A Tour of the RISC-V ISA Formal Specification

RISC-V Foundation ISA Formal Spec Technical Committee

At RISC-V Summit, San Jose
December 12, 2019

Dear reader, *Caveat!* we must confess,
This slide deck remains a work in progress,
We suggest, until the target date, please take a recess,
Your cognitive serenity so as not to distress.

Please come back on December 12, 2019.

Location of these slides and additional material

These slides are in "`Slides_Tutorial.pdf`" in this GitHub repository:
`https://github.com/rsnikhil/RISCV_ISA_Spec_Tour`

The repository also contains "`Slides_Installation.pdf`" which describes
Step A that is sufficient for reading the ISA formal spec, and
Step B that is needed to create an executable version of the spec.

Please at least perform Step A, which is just to clone the following repository:
`https://github.com/rems-project/sail-riscv`
This is sufficient for most of this tutorial.

Objectives for this tutorial

- *Primary objective*: To give you a basic *reading literacy* of the RISC-V ISA Formal Specification, i.e., where you are comfortable reading and consulting the formal spec on your own on a daily basis.
- *Secondary objective*: To show you how to *execute* the formal spec on RISC-V binaries, such as standard ISA tests, the Compliance Suite, and your own compiled programs. Typically, you'd do this to compare a RISC-V implementation (simulator, architectural model, actual hardware implementation) to see if the implementation's behavior matches the formal spec accurately.

Target audience: Working RISC-V engineers (e.g., CPU designers, compiler writers, simulator/emulator writers, ...) who consult the spec to clarify understanding of instruction semantics. We do not assume any prior experience with formal languages or formal methods.

This tutorial may not be suitable for afficionados of formal methods, except as an initial familiarization with Sail and the RISC-V formal spec.

Credits

The RISC-V ISA Formal Spec, in the Sail language, was written by:

> *Prashanth Mundkur, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Rishiyur Nikhil*

The Sail language itself was principally designed and implemented by the authors of this paper:

> *ISA Semantics for ARMv8-A, RISC-V, and Cheri-MIPS*,
> Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, Peter Sewell
> in *Proc. 46th ACM SIGPLAN Symp. on Principles of Programming Languages (POPL), Cascais/Lisbon, Portugal, Jan 13-19, 2019*, pp. 71:1–71:31.

Thanks also to all members of the Technical Group for the ISA Formal Spec in the RISC-V Foundation who volunteered their time for discussions paced by weekly con-calls. And to the Technical Committee of the RISC-V Foundation who provided feedback and guidance on process, based on our monthly status updates.

Outline

Introduction

Context, briefly

What is it?

- The formal spec of the RISC-V ISA is intended to be the *definitive* reference for RISC-V instructions:
  - Encoding
  - Execution semantics (what executing each instruction is supposed to do)

  More authoritative than the English prose spec (complete, precise, unambiguous).
- Excutable: can be run as a simulator executing RISC-V binaries, providing definitive execution behaviors
- Readable and usable by, and useful to, ordinary mortals who don't do formal stuff for a living.
  - Casual reading, as a reference guide to RISC-V instructions.
  - Executable "golden reference model" to check implementation correctness.
- For those who *do* formal stuff for a living, usable with formal tools for proofs of correctness of compilers, CPU implementations, automatic generation of tests, test coverage, etc.

Downloading and Installing

Although the slides in this deck are self-contained, containing code fragments, we recommend that, in parallel, you view the actual code from the repository in a text viewer or editor. The fragments here are excerpts, contain elisions, and cannot show their larger context.

Each code fragment in this deck shows the file from which it is taken.

Downloading and Installing

Please see the separate slide deck `Slides_Installation.pdf` for instructions on how to download the spec, and optionally to build an executable version of the spec.

Reading the ISA to understand semantics of each RISC-V instruction

About Sail, and the RISC-V ISA Formal Spec written in Sail

- The RISC-V ISA Formal Spec is written in the language Sail, which is a DSL (Domain-Specific Language) designed for purpose, i.e., for writing ISA specs.
- Sail has been used to describe ISAs for RISC-V, ARMv8 (complete spec!), MIPS, parts of x86 and IBM POWER, and more.
- The Sail manual can be found in the main Sail repository
  `https://github.com/rems-project/sail`
- In this tutorial we won't study Sail separately; we'll jump into studying the RISC-V Spec written in Sail, explaining the Sail notation as we go along.
- The RISC-V spec in Sail has its own repository:
  `https://github.com/rems-project/sail-riscv`
  We will be studying the code in the `model/` directory.

A first taste of the semantics

File riscv_insts_base.sail

```
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_SLL  => if   sizeof(xlen) == 32
                  then rs1_val << (rs2_val[4..0])
                  else rs1_val << (rs2_val[5..0]),
    ... };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

- The semantics of each instruction is given by an "execute" instruction, a fragment of which is shown above.
- The function argument says that it is an "R-format" instruction containing source register fields rs1 and rs2, destination register field rd, and an "op" sub-opcode identifying the specific operation within the group of "R-format" instructions.
- It shows reading the two source registers rs1 and rs2, performing the operation specified by "op", and writing the result to register rd (here showing the ADD and SLL sub-opcodes).

Spec source files (there are a lot of them!)

```
--12:50:52--Dell-mation: ~/git_clones/sail-riscv/model
$ ls
main.sail                       riscv_insts_aext.sail    riscv_platform.sail           riscv_termination_duo.sail
prelude_mapping.sail            riscv_insts_base.sail    riscv_pmp_control.sail        riscv_termination_rv32.sail
prelude_mem_metadata.sail       riscv_insts_begin.sail   riscv_pmp_regs.sail           riscv_termination_rv64.sail
prelude_mem.sail                riscv_insts_cext.sail    riscv_pte.sail                riscv_types_ext.sail
prelude.sail                    riscv_insts_end.sail     riscv_ptw.sail                riscv_types.sail
README.md                       riscv_insts_mext.sail    riscv_regs.sail               riscv_vmem_common.sail
riscv_addr_checks_common.sail   riscv_insts_next.sail    riscv_reg_type.sail           riscv_vmem_rv32.sail
riscv_addr_checks.sail          riscv_insts_rmem.sail    riscv_step_common.sail        riscv_vmem_rv64.sail
riscv_analysis.sail             riscv_insts_zicsr.sail   riscv_step_ext.sail           riscv_vmem_sv32.sail
riscv_csr_ext.sail              riscv_jalr_rmem.sail     riscv_step_rvfi.sail          riscv_vmem_sv39.sail
riscv_csr_map.sail              riscv_jalr_seq.sail      riscv_step.sail               riscv_vmem_sv48.sail
riscv_decode_ext.sail           riscv_mem.sail           riscv_sync_exception.sail     riscv_vmem_tlb.sail
riscv_duopod.sail               riscv_misa_ext.sail      riscv_sys_control.sail        riscv_vmem_types.sail
riscv_ext_regs.sail             riscv_next_control.sail  riscv_sys_exceptions.sail     riscv_xlen32.sail
riscv_fetch_rvfi.sail           riscv_next_regs.sail     riscv_sys_regs.sail           riscv_xlen64.sail
riscv_fetch.sail                riscv_pc_access.sail     riscv_termination_common.sail rvfi_dii.sail
```

Sail does not have a package/module structure–the full Sail program is just the concatenation of the source files.

We organize them into files according to our convenience.

In this tutorial we will look at excerpts of some of these files.

A word about types and type-checking

- Sail is a strongly-typed language, and does its type-checking statically (i.e., on the source code, without running the code).
- Many types are familiar from other languages (particularly functional programming languages): vectors, structs, algebraic types/tagged unions, ...
- Perhaps the most unfamiliar for many people will be the use of numbers as types.
  - In ISAs (unlike most software programming languages) we deal with representations (e.g., bit-vectors) of many different sizes, and the precise size is important.
  - Moreover, sizes of various entities are often related. E.g., the shift amount in RV32 (respectively, RV64) should be a 5-bit value (respectively, a 6-bit value). In Sail, such relationships can be expressed in types, and are type-checked.
- Sail also statically keeps track of *effects* (for example, does a certain expression read any registers, write any registers, ...). More about this later.

Let's start with some small, easy files

We'd use one of these two files, depending on whether we are considering RV32 or RV64.

File riscv_xlen32.sail

```
1  /* Define the XLEN value for the architecture. */
2
3  type xlen       : Int = 32
4  type xlen_bytes : Int = 4
5  type xlenbits          = bits(xlen)
```

File riscv_xlen64.sail

```
1  /* Define the XLEN value for the architecture. */
2
3  type xlen       : Int = 64
4  type xlen_bytes : Int = 8
5  type xlenbits          = bits(xlen)
```

In lines 3-4, we are defining new *types* that are numeric.

In line 5 we are defining a new type for bit-vectors of size `xlen`. The type `bits(`t`)` represents the type of bit-vectors of size t. It's parameter t must be a numeric type (here, we instantiate it as `xlen`).

Values in integer registers

File riscv_reg_type.sail

```
1  /* default register type */
2  type regtype = xlenbits
3
4  /* default zero register */
5  let zero_reg : regtype = EXTZ(0x0)
```

In line 2 we're defining the *type* of values in registers; it's the same type as `xlenbits`.

In line 5 we're defining a specific *value* of this type, using the library function `EXTZ` to zero-extend the constant 0x0 to the appropriate length. Because of strong type-checking (including some amount of type inference), Sail knows exactly how much extension is needed.

Note: the keyword `type` introduces a type definition, the keyword `let` introduces a value definition.

Integer registers

File riscv_regs.sail

```
1 register PC       : xlenbits
2
3 register x1  : regtype
4 register x2  : regtype
5 ...
6 register x31 : regtype
```

In line 1 with keyword `register` we declare `PC` to be a register, and we specify the type of values it can contain, `xlenbits`. The remaining lines similarly declare registers `x1`...`x31`.

(There's no `x0` register because it's a constant 0.)

Reading from integer registers

File riscv_regs.sail

```
1  val rX : forall 'n, 0 <= 'n < 32. regno('n) -> xlenbits effect {rreg, escape}
2  function rX r = {
3    let v : regtype =
4      match r {
5        0 => zero_reg,
6        1 => x1,
7        ...
8        31 => x31,
9        _  => {assert(false, "invalid register number"); zero_reg}
10     };
11   regval_from_reg(v)
12 }
```

This defines a function `rX` that takes a register number `r` as argument and returns the value contained in that register. Line 1, introduced by the `val` keyword, specifies the *type* of the function. It can be read as:

> *For all n in the range 0..31, it takes an argument n that is a register number, and returns a value of type* `xlenbits`. *Executing this function can have two possible effects,* `rreg` *(reading a register) and* `escape` *(abort due to illegal register number).*

Reading from integer registers (contd.)

File riscv_regs.sail

```
1  val rX : forall 'n, 0 <= 'n < 32. regno('n) -> xlenbits effect {rreg, escape}
2  function rX r = {
3    let v : regtype =
4      match r {
5        0 => zero_reg,
6        1 => x1,
7        ...
8        31 => x31,
9        _  => {assert(false, "invalid register number"); zero_reg}
10     };
11   regval_from_reg(v)
12 }
```

Line 2, introduced by the `function` keyword, defines the function `rX` itself, with argument `r`.

The `let` binding in line 3 introduces a local variable `v` and binds it to the value of the "pattern-matching" expression in Lines 4-10. This matches the value `r` with each of the subsequent patterns 0, 1, 2, ... 31, returning the value of the right-hand side on first match.

Reading from integer registers (contd.)

File riscv_regs.sail

```
1  val rX : forall 'n, 0 <= 'n < 32. regno('n) -> xlenbits effect {rreg, escape}
2  function rX r = {
3    let v : regtype =
4      match r {
5        0 => zero_reg,
6        1 => x1,
7        2 => x2,
8        ...
9        _  => {assert(false, "invalid register number"); zero_reg}
10     };
11   regval_from_reg(v)
12 }
```

The type of `v` is `regtype`, i.e., it is a register, and so in Line 11 the `regval_from_reg(v)` application reads out the register value, of type `xlenbits`.

In Sail, a block is a series of expressions in in braces, and the value of the last expression is treated as the value of the whole block; here, that is also the result of the function.

Observation: improved type-checking and pattern analysis in the Sail compiler should allow us to omit the `assert` statement. This, in turn, should allow us to omit the `escape` effect.

Writing integer registers

File riscv_regs.sail

```
1  val wX : forall 'n, 0 <= 'n < 32. (regno('n), xlenbits) -> unit effect {wreg, escape}
2  function wX (r, in_v) = {
3    let v = regval_into_reg(in_v);
4    match r {
5      0  => (),
6      1  => x1 = v,
7      ...
8      31 => x31 = v,
9      _  => assert(false, "invalid register number")
10   };
11 }
```

This is similar to the `rX` read-function. The function type-declaration in line 1 says it takes two arguments, one a register number and the second a value of type `xlenbits`, and its result type is `unit` which is like the "void" type in C, indicating a value of no particular interest, since this is a pure side effect. Its effects include `wreg` (writing a register) and `escape`.

Using Sail overloading to simplify notation

File riscv_regs.sail

```
1 overload X = {..., rX, wX}
```

This allows "`X(r)`" to be used to read a register (invoking the function "`rX()`"),
and "`X(r)=v`" to write a register (invoking the function "`wX()`").

"Scattered" organization of instruction specs

In a traditional programming language, we might have:

- A type definition showing all the different variants of instructions (opcodes, register fields, immediate fields, ...).
- A decode function that describes how to take a 32-bit value into into each of the different instruction variants.
- An execute function that describes how to execute each variant of instruction.

Traditional instruction set manuals "scatter" this same information differently—a page (or a few) per instruction variant, showing:

- Its fields (opcode, register fields, immediate fields, ...).
- How a 32-bit instruction is decoded/encoded.
- How it is executed.

Sail supports the latter more traditional, familiar organization. For each type of instruction, all its relevant information is collected in one place.

Introducing things whose clauses will be scattered

We must first introduce the generic information about entities whose individual definition-clauses will given later in scattered fashion; specifically, types:

File `riscv_insts_begin.sail`

```
1  scattered union ast
2
3  val encdec : ast <-> bits(32)
4  scattered mapping encdec
5
6  val assembly : ast <-> string
7  scattered mapping assembly
```

Line 1 introduces the type "ast" which is a *union* of all the different variants of instructions. Each variant will follow later, in a scattered fashion. Here, "ast" stands for Abstract Syntax Tree, the decoded view of an instruction.

Line 3 declares the type of the "`encdec`" mapping, which is a pair of functions converting from a 32-bit value (instruction) to is decoded view (ast), and vice versa.

Line 6 declares the type of the "`assembly`" mapping that converts from a string to a decoded instruction and vice versa.

The "`scattered`" annotation and lines indicate that the clauses of each of these entities will follow later, in a scattered manner.

Introducing things whose clauses will be scattered (contd.)

File riscv_insts_begin.sail

```
1 val execute : ast -> Retired effect {escape, wreg, rreg, rmem, ...}
2 scattered function execute
```

Line 1 declares the type of the "`execute`" function, which takes a decoded instruction (ast) and returns a "`Retired`" result which indicates whether it should be counted as a retired instruction or not. It also specifies all the possible effects of an instruction, such as aborting ("`escape`"), writing and reading registers ("`wreg, rreg`"), reading memory, and so on.

The "`scattered`" line indicates that the clauses of this function will follow later, in a scattered manner.

Type definitions for instruction fields

The top of each page in *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, Chapter 25 *Instruction Set Listings* shows the RISC-V formats:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- The least-significant 7 bits provide a major opcode.
- The funct3 and funct7 fields (and sometimes the immediate fields) often specify sub-opcodes.
- The "rs1", "rs2" and "rd" fields are 5-bit values specifying source and destination registers.
- Immediate values are often composed from non-trivial permutation of "imm" instruction fields.

Type definitions for instruction fields (contd.)

We declare convenient types for instruction fields.

File riscv_types.sail

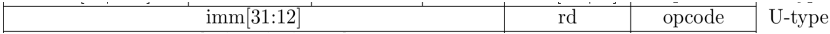
```
1  type regidx  = bits(5)
2  type csreg   = bits(12)   /* CSR addressing */
3  ...
4  type opcode = bits(7)
5  type imm12  = bits(12)
6  type imm20  = bits(20)
7  ...
```

These are definitions for register indexes, CSR register addresses, major opcodes, and 12-bit and 20-bit immediates.

LUI and AUIPC: decoded view

Earlier we declared "`ast`" to be a "`union`" type, i.e., a type with several variants. We also declared that the variants would be provided later in scattered clauses.

We now provide one of those clauses, for U-format instructions (LUI and AUIPC):



File `riscv_insts_base.sail`

```
1 union clause ast = UTYPE : (bits(20), regidx, uop)
```

This says: one variant of the `ast` type is called `UTYPE`. It contains 3 fields (identified positionally, not with keywords) whose types are, respectively, a bit-vector of 20 bits, a register index, and a `uop` (which identifies whether it's an LUI or AUIPC).

Note: Sail unions are similar to "algebraic types" or "tagged unions" in other programming langauges. Each value of a tagged union carries a way (a "tag") by which we can query which variant this value encodes.

In Sail, as is common in functional programming languages, values of union type are usually analyzed in "pattern-matching" statements, which are like case/switch statements where each clause matches a variant of the union.

LUI and AUIPC: Decoding

Earlier, we declared a mapping (a function and its inverse) "encdec" with this type.

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

and further declared that the mapping body would be provided later in scattered clauses.

We now provide one such clause, showing how to encode/decode LUI and AUIPC instructions.

File riscv_insts_base.sail

```
1  mapping encdec_uop : uop <-> bits(7) = {
2    RISCV_LUI   <-> 0b0110111,
3    RISCV_AUIPC <-> 0b0010111
4  }
5
6  mapping clause encdec = UTYPE(imm, rd, op)
7    <-> imm @ rd @ encdec_uop(op)
```

Lines 1-4 define a new, local mapping between the bit-encodings of the 7-bit opcode in a U-format instruction to a value of uop type, i.e., the symbolic names for the corresponding instructions.

Lines 6-7 add a scattered clause to the "encdec" mapping. The left-hand-side of "<->" in Line shows the decoded view. The right-hand side shows a bit-concatenation. The prior declarations allow Sail to infer that "imm", "rd", and "encddec_op(op)" are are 20-bit, 5-bit and 7-bit fields, respectively, and that the concatenation is a 32-bit value,

LUI and AUIPC: execution

Earlier, we declared a function "execute" with this type.

```
val execute :  ast -> Retired effect {escape, wreg, rreg, rmem, ...}
```

and further declared that the function body would be provided later in scattered clauses. Here is the definition of the Retired type:

File riscv_types.sail

```
1  enum Retired = {RETIRE_SUCCESS, RETIRE_FAIL}
```

We could have used the "bool" type for this, but (a) these provide more readable names, and (b) this prevents accidental confusion of random booleans where an Retired value is expected.

LUI and AUIPC: execution (contd.)

We now provide one of the clauses for `execute`, for LUI and AUIPC instructions.

File riscv_insts_base.sail

```
1 function clause execute UTYPE(imm, rd, op) = {
2   let off : xlenbits = EXTS(imm @ 0x000);
3   let ret : xlenbits = match op {
4     RISCV_LUI   => off,
5     RISCV_AUIPC => get_arch_pc() + off
6   };
7   X(rd) = ret;
8   RETIRE_SUCCESS
9 }
```

In Line 1, the argument to the "`execute`" function is given as a *pattern* "`UTYPE(imm, rd, op)`". Remember `execute` can be applied to any value of type `ast`. The pattern here ensures that this clause will only be relevant to those `ast` values that are of the `UTYPE` variant. On a successful match, it also binds the names `imm`, `rd` and `op` to the three fields of the decoded instruction, so we can use these variables in the body of the function.

LUI and AUIPC: execution (contd.)

File riscv_insts_base.sail

```
1  function clause execute UTYPE(imm, rd, op) = {
2    let off : xlenbits = EXTS(imm @ 0x000);
3    let ret : xlenbits = match op {
4      RISCV_LUI   => off,
5      RISCV_AUIPC => get_arch_pc() + off
6    };
7    X(rd) = ret;
8    RETIRE_SUCCESS
9  }
```

Strong-typing assures us that `imm` is of type `bits(20)`, i.e., a bit-vector of length 20. In Line 2, we concatenate this with the 12-bit value 0x000, giving us a 32-bit value. Then, we use `EXTS` to sign-extend it as necessary. This does nothing in RV32, since it's already a 32-bit value, and it sign-extends it to 64 bits in RV64. The result is bound to the local variable `off` of type `xlenbits`.

LUI and AUIPC: execution (contd.)

File riscv_insts_base.sail

```
1  function clause execute UTYPE(imm, rd, op) = {
2    let off : xlenbits = EXTS(imm @ 0x000);
3    let ret : xlenbits = match op {
4      RISCV_LUI   => off,
5      RISCV_AUIPC => get_arch_pc() + off
6    };
7    X(rd) = ret;
8    RETIRE_SUCCESS
9  }
```

Line 3 binds local variable `ret`, of type `xlenbits`, to the right-hand side, which is a pattern-matching expression examining `op`. When it matches `RISCV_LUI`, the value is just `off`. When it matches `RISCV_AUIPC`, the value is added to `get_arch_pc()`, which retrieves the value of the program counter in the current machine state.

Line 7 assigns this value to register `rd`, using the overloading of `X` we saw earlier.

Finally, Line 8 is the constant expresssion `RETIRE_SUCCESS`. Being the last expression in the block, and the block being the body of the function, this is the value returned by the function.

LUI and AUIPC: assembly language parsing and printing

We first define a mapping (function and its inverse) to convert the sub-opcode `uop` to a string and back:

File riscv_insts_base.sail

```
1 mapping utype_mnemonic : uop <-> string = {
2   RISCV_LUI   <-> "lui",
3   RISCV_AUIPC <-> "auipc"
4 }
```

Then, we add a scattered clause to our previously introduced `assembly` mapping:

File riscv_insts_base.sail

```
1 mapping clause assembly = UTYPE(imm, rd, op)
2   <-> utype_mnemonic(op) ^ spc() ^ reg_name(rd) ^ sep() ^ hex_bits_20(imm)
```

- the caret operator concatenates strings; `spc()` and `sep()` return strings for spaces and commas;
- `reg_name(r)` returns the string name for its register-number argument;
- `hex_bits_20()` returns a string showing a hex printing of a 20-bit value.

Conditional branch instructions: ast clause

Conditional branch instructions include BEQ, BNE, BLT, BGE, BLTU, and BGEU: we define symbolic names:

File riscv_types.sail

```
1 enum bop = {RISCV_BEQ, RISCV_BNE, RISCV_BLT,
2             RISCV_BGE, RISCV_BLTU, RISCV_BGEU}    /* branch ops */
```

Branch instructions are encoded in the B-format:

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

Our abstract (decoded) ast view is:

File riscv_insts_base.sail

```
1 union clause ast = BTYPE : (bits(13), regidx, regidx, bop)
```

Note:

- The branch offset immediate value is *13 bits* composed from 12 bits in the instruction, with 0 appended as the least-significant bit.
- The 12 bits come from non-contiguous 7-bit and 5-bit fields in the instruction.
- Our ast (decoded) view holds the 13-bit offset (computed in the `encdec` function to be shown shortly).

Conditional branch instructions: encoding/decoding

We define a mapping converting the 3-bit "funct3" field in the instruction to its abstract names:

File riscv_types.sail

```
1 mapping encdec_bop : bop <-> bits(3) = {
2   RISCV_BEQ  <-> 0b000,    RISCV_BNE  <-> 0b001,
3   RISCV_BLT  <-> 0b100,    RISCV_BGE  <-> 0b101,
4   RISCV_BLTU <-> 0b110,    RISCV_BGEU <-> 0b111
5 }
```

Then, we add a scattered clause to the encdec mapping:

File riscv_types.sail

```
1 mapping clause encdec = BTYPE(imm7_6 @ imm5_0 @ imm7_5_0 @ imm5_4_1 @ 0b0, rs2, rs1, op)
2 <-> imm7_6 : bits(1) @ imm7_5_0 : bits(6)
3     @ rs2 @ rs1 @ encdec_bop(op)
4     @ imm5_4_1 : bits(4) @ imm5_0 : bits(1)
5     @ 0b1100011
```

Observe 13-bit offset is composed by extracting bits from various places in the instruction.

Conditional branch instructions: execution

We add a scattered clause to the execute function. The first part is straightforward:

File riscv_types.sail

```
1  function clause execute (BTYPE(imm, rs2, rs1, op)) = {
2    let rs1_val = X(rs1);
3    let rs2_val = X(rs2);
4    let taken : bool = match op {
5      RISCV_BEQ  => rs1_val == rs2_val,     RISCV_BNE  => rs1_val != rs2_val,
6      RISCV_BLT  => rs1_val <_s rs2_val,    RISCV_BGE  => rs1_val >=_s rs2_val,
7      RISCV_BLTU => rs1_val <_u rs2_val,    RISCV_BGEU => rs1_val >=_u rs2_val
8    };
9    let t : xlenbits = PC + EXTS(imm);
10   ...
11 }
```

- Line 4 computes "taken", indicating whether the branch is taken or not. It does a pattern-match on the sub-opcode op. Note that BLT and BLTU are supposed to interpret their argument as signed and unsigned values, respectively. This is encoded by using different Sail pre-defined comparison operators <_s and <_u, respectively.
- Line 9 computes the branch target, adding a sign-extension of the immediate to the PC.

Conditional branch instructions: execution (contd.)

The second part of the `execute` function clause performs different actions depending on whether the branch is taken or not:

File riscv_types.sail

```
1  function clause execute (BTYPE(imm, rs2, rs1, op)) = {
2    ...
3    if taken then {
4      ...
5      ...
6    } else RETIRE_SUCCESS
7  }
```

If the branch is not taken, there is no further action and the result is `RETIRE_SUCCESS`.

Conditional branch instructions: execution (contd.)

If the branch would be taken, we first check that the branch target PC is valid.

File riscv_types.sail

```
1  if taken then {
2    ... <some code elided> ...
3        if bit_to_bool(target[1]) & (~ (haveRVC())) then {
4          handle_mem_exception(target, E_Fetch_Addr_Align());    RETIRE_FAIL;
5        } else {
6          set_next_pc(target);    RETIRE_SUCCESS
7        }
```

- Line 3 checks the requirement that, without the "C" ISA extension (compressed instructions), the branch target must be 4-byte aligned, i.e., bit [1] must be 0. `bit_to_bool` converts a value of `bits(1)` type to `bool` type (we could have also used "==1"). `haveRVC` checks if the C extension is active. If the target is not ok, on Line 4 we invoke function `handle_mem_exception` to perform exception actions and return failure. If the target is ok, Line 6 assigns it to the next PC and we return success.
- Our "<some code elided>" on Line 2 contains additional checks for target validity that may be required by any other extensions.

Conditional branch instructions: Assembly language parsing and printing

We first define a mapping (function and its inverse) to convert the sub-opcode bop to a string and back:

File riscv_insts_base.sail

```
1 mapping btype_mnemonic : bop <-> string = {
2   RISCV_BEQ  <-> "beq",     RISCV_BNE  <-> "bne",
3   RISCV_BLT  <-> "blt",     RISCV_BGE  <-> "bge",
4   RISCV_BLTU <-> "bltu",    RISCV_BGEU <-> "bgeu"
5 }
```

Then, we add a scattered clause to our previously introduced assembly mapping:

File riscv_insts_base.sail

```
1 mapping clause assembly = BTYPE(imm, rs2, rs1, op)
2   <-> btype_mnemonic(op) ^ spc() ^ reg_name(rs1) ^ sep() ^ reg_name(rs2) ^
3           sep() ^ hex_bits_13(imm)
```

- the caret operator concatenates strings; spc() and sep() return strings for spaces and commas;
- reg_name(r) returns the string name for its register-number argument;
- hex_bits_13() returns a string showing a hex printing of a 13-bit value.

Taking stock ...

The general scheme for adding a new instruction, or new class of instructions, should be clear by now:

- Define an enum and mapping for any sub-opcodes in the class (if the class contains more than one instruction)
- Augment the "`ast`" type by adding a scattered clause to describe this new class
- Augment the "`encdec`" mapping by adding a scattered clause to describe this new class
- Augment the "`execute`" function by adding a scattered clause to describe this new class
- Augment the "`assembly`" mapping by adding a scattered clause to describe this new class

It is a stylistic judgement call whether you define a class with sub-opcodes, or just define a separate clause for each instruction in the class. E.g., we could have defined separate "`ast`" , "`encdec`", "`execute`" and "`assembly`" clauses for BEQ, BNE, BLT, ...

A class with sub-opcodes makes sense when the instructions share structure and semantics. For example, BEQ/BNE/BLT/... differ only in the particular comparison operator; using a class with sub-opcodes captures this similarity.

For the remaining examples we'll focus on the "`execute`" function.

LOAD instructions: execution

Memory-access instructions involve many more steps, since they can involve alignment checks, virtual address-to-physical address translation, physical memory protection checks, ordering relationships with other memory accesses, and so on. Many of these can trap (raise an exception).

The header of a memory-load instruction:

File riscv_insts_base.sail

```
1 function clause execute(LOAD(imm, rs1, rd, is_unsigned, width, aq, rl)) = {
```

The arguments are the

- the immediate, rs1 and rd fields from the instruction;
- whether the loaded value is treated as signed or unsigned, i.e., whether the loaded value should be sign-extended or zero-extended to the width of the destination register;
- the width: byte, halfword (2 bytes), word (4 bytes) or double (8 bytes);
- the acquire/release semantics for memory ordering.

LOAD instructions: execution (contd.)

The next step is to compute the actual (virtual) address to be accessed:

File riscv_insts_base.sail

```
1  let offset : xlenbits = EXTS(imm);
2  match ext_data_get_addr(rs1, offset, Read(Data), width) {
3    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
4    Ext_DataAddr_OK(vaddr) =>
5        ...
```

After computing the offset by sign-extending the immediate value, it invokes the function "`ext_data_get_addr`" to perform a signed addition to the contents of rs1. This function is defined in "`riscv_addr_checks.sail`". By encapsulating this addition in a function, we allow future extensibility to new ISA extensions that may peform additional checks/transformations on the address.

This function can return an error, but in the normal simple case without additional ISA extensions it returns "`Ext_DataAddr_OK(vaddr)` containing the effective virtual address. We use a `match` expression to distinguish these two outcomes.

LOAD instructions: execution (contd.)

Next:

File riscv_insts_base.sail

```
1       if   check_misaligned(vaddr, width)
2       then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
3       else match translateAddr(vaddr, Read(Data)) {
4           ...
```

The function "`check_misaligned(vaddr, width)`" optionally checks if the access is aligned for the requested width. This function is defined a little earlier in the file and returns true it is misaligned *and if we've configured the model to disallow misaligned accesses*.

If ok, it invokes "`translateAddr(vaddr, Read(Data))`" to optionally translate virtual addresses to physical addresses. This function is defined in a collection of files:

`riscv_vmem_rv32.sail`, `riscv_vmem_sv32.sail`
`riscv_vmem_rv64.sail`, `riscv_vmem_sv39.sail`, `riscv_vmem_sv48.sail`
`riscv_vmem_tlb.sail`

different subsets of which are used depending on whether we're modeling RV32 or RV64, and the Sv32, Sv39 or Sv48 virtual memory schemes.

The function simply returns the address as-is if not running with virtual memory.

In the virtual-memory translation functions, you'll notice that they also model a TLB (Translation Lookaside Buffer). This is because TLBs are visible in the semantics via the SFENCE.VMA instruction.

LOAD instructions: execution (contd.)

Finally:

File riscv_insts_base.sail

```
1       else match translateAddr(vaddr, Read(Data)) {
2         TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
3         TR_Address(addr, _) =>
4           match (width, sizeof(xlen)) {
5             (BYTE, _)   =>
6               process_load(rd, vaddr, mem_read(Read(Data), addr, 1, aq, rl, false), is_unsigned),
7             (HALF, _)   =>
8               process_load(rd, vaddr, mem_read(Read(Data), addr, 2, aq, rl, false), is_unsigned),
9             (WORD, _)   =>
10              process_load(rd, vaddr, mem_read(Read(Data), addr, 4, aq, rl, false), is_unsigned),
11            (DOUBLE, 64) =>
12              process_load(rd, vaddr, mem_read(Read(Data), addr, 8, aq, rl, false), is_unsigned)
```

If the virtual-to-physical translation was successful, we invoke "`mem_read`" to perform the raw memory read, and pass the result to "`process_load`" to process the result (which could be an exception, e.g, if there is no memory at that address).

The first three clauses of the "`match`" expression use the wildcard pattern "`_`" in the second component, since these sizes are valid in RV32 and RV64. The fourth clause will only match when the second component is 64, i.e., it restricts it to RV64.

Exceptions

RISC-V has

- interrupts (asyncnronous exceptions, conceptually "between" any two instructions)
- traps (synchronous exceptions, due to execution of an instruction)

The different kinds of interrupts:

File riscv_types.sail

```
1 enum InterruptType = {  I_U_Software,    I_S_Software,    I_M_Software,
2                         I_U_Timer,       I_S_Timer,       I_M_Timer,
3                         I_U_External,    I_S_External,    I_M_External    }
```

The file also contains a function to convert bit-encodings to these symbolic names:

File riscv_types.sail

```
1 val interruptType_to_bits : InterruptType -> bits (8)
2 function interruptType_to_bits (i) = match (i) {   I_U_Software => 0x00,
3                                                    I_S_Software => 0x01,  ... }
```

Comment: A mapping would be more expressive than a function, but since we don't decode interrupts/exceptions, we don't need the inverse function.

Exceptions (contd.)

The different kinds of traps, and converting to bits:

File riscv_types.sail

```
union ExceptionType = { E_Fetch_Addr_Align    : unit,     E_Fetch_Access_Fault : unit,
                        E_Illegal_Instr       : unit,     E_Breakpoint         : unit,
                        E_Load_Addr_Align     : unit,     E_Load_Access_Fault  : unit,
                        E_SAMO_Addr_Align     : unit,     E_SAMO_Access_Fault  : unit,
                        E_U_EnvCall           : unit,     E_S_EnvCall          : unit,
                        E_Reserved_10         : unit,     E_M_EnvCall          : unit,
                        E_Fetch_Page_Fault    : unit,     E_Load_Page_Fault    : unit,
                        E_Reserved_14         : unit,     E_SAMO_Page_Fault    : unit }

val exceptionType_to_bits : ExceptionType -> exc_code
function exceptionType_to_bits(e) = match (e) { E_Fetch_Addr_Align()   => 0x00,
                                                E_Fetch_Access_Fault() => 0x01, ...}
```

Comment: I think this could also have been written as an enum. The "`unit`" type is like `void`, so these union variants don't contain any interesting data with each tag.

Exceptions (contd.)

Some traps may carry additional information. In Sail (and OCaml), optional information is usually expressed using the "option" predefined type:

predefined

```
1 union option ('a : Type) = { Some : 'a,
2                              None : unit }
```

i.e., the "Some" variant carries some additional information (generic/polymorphic type 'a), and the "None" variant carries no additional information.

File riscv_sync_exception.sail

```
1 struct sync_exception = { trap    : ExceptionType,
2                           excinfo : option(xlenbits),
3                           ext     : option(ext_exception)   /* for extensions */ }
```

The "trap" field is necessary information. The other two fields carry optional information, for standard traps (such as an address that provoked a trap), and also for future standard or non-standard ISA extensions.

Handling certain exceptions

The "handle_mem_exception" action function we saw earlier in conditional branches with illegal branch targets is:

File riscv_sys_control.sail

```
1  function handle_mem_exception(addr : xlenbits, e : ExceptionType) -> unit = {
2    let t : sync_exception = struct { trap    = e,
3                                      excinfo = Some(addr),
4                                      ext     = None() } in
5    set_next_pc(exception_handler(cur_privilege, CTL_TRAP(t), PC))
6  }
```

Lines 1-3 construct a "sync_exception" value, filling in the address as optional exception info, and binds it to the local variable "t".

In Line 4 we invoke a more general "exception_handler" (next slide).

Handling exceptions (contd.)

File riscv_sys_control.sail

```
1  function exception_handler(cur_priv : Privilege, ctl : ctl_result,
2                             pc: xlenbits) -> xlenbits = {
3    match (cur_priv, ctl) {
4      (_, CTL_TRAP(e)) => {
5        let del_priv = exception_delegatee(e.trap, cur_priv);
6        ...
7        trap_handler(del_priv, false, exceptionType_to_bits(e.trap), pc, e.excinfo, e.ext)
8      },
9      (_, CTL_MRET())  => { ... }
10     (_, CTL_SRET())  => { ... }
11     (_, CTL_URET())  => { ... } }
```

Line 5 checks if the current trap, at the current privilege level, is being delegated to be handled at a different privilege level (returning that, or the current, privilege level).

Line 7 invokes an even more general trap handler (next slide).

Lines 9-11 handle exception returns from the Machine, Supervisor and User privilege levels, respectively.

Handling exceptions (contd.)

File riscv_sys_control.sail

```
1  function trap_handler(del_priv : Privilege, intr : bool, c : exc_code,
2                        pc : xlenbits, info : option(xlenbits), ext : option(ext_exception))
3                       -> xlenbits = {
4    cancel_reservation();    /* for LR/SC */
5    match (del_priv) {
6      Machine => { mcause->IsInterrupt() = bool_to_bits(intr);
7                   mcause->Cause()       = EXTZ(c);
8                   mstatus->MPIE()       = mstatus.MIE();    mstatus->MIE() = 0b0;
9                   mstatus->MPP()        = privLevel_to_bits(cur_privilege);
10                  mtval                 = tval(info);       mepc           = pc;
11                  cur_privilege         = del_priv;
12                  prepare_trap_vector(del_priv, mcause) },
13     Supervisor => { ... }
14     User => { ... }
```

This is an intricate but otherwise unremarkable assignment of certain values to certain CSRs.

Line 15 invokes "prepare_trap_vector" (in file "riscv_sys_extensions.sail") which returns the PC that is in "mtvec", "stvec", or "utvec", as appropriate.

Executing complete programs

Executing complete programs

- So far, we've only talked about the decode and execute function for individual instructions. We've said nothing about how and when these get invoked, nor about how instructions are fetched.
- This separation is deliberate. We may wish to build several different processor models: pipelined, superscalar, multi-hart, and so on. Each of these would be a different top-level system, with its own system-level semantics, but they can all share the individual instruction semantics discussed so far.
- In the slides that follow, we'll sketch one such encapsulating model, which is used in the default simulators built from the model. This model, shown in files "`main.sail`" and "`riscv_step.sail`" implement a simple, sequential, unpipelined, one-instruction-at-a-time fetch-execute loop ().

Executing complete programs (contd.)

The top-level function initializes the PC, initializes the model as a whole (including certain CSRs and registers), and performs the fetch-execute loop:

File main.sail

```
1  function main () : unit -> unit = {
2    PC = sail_zero_extend(0x1000, sizeof(xlen));
3    init_model();
4    loop()
5  }
```

The loop, in turn, repeatedly performs a fetch-execute step:

File riscv_step.sail

```
1  function loop () : unit -> unit = {
2    while (...) do {
3      let stepped = step(step_no);
4      ...
5    }
6  }
```

Executing complete programs (contd.)

File riscv_step.sail

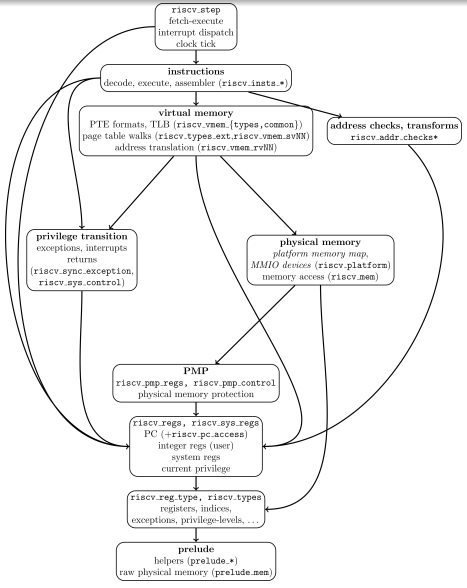
```
1  function step(step_no : int) -> bool = {
2    let (retired, stepped) : (Retired, bool) =
3      match dispatchInterrupt(cur_privilege) {
4        Some(intr, priv) => { handle_interrupt(intr, priv); (RETIRE_FAIL, false) },
5        None() => {
6          let f : FetchResult = ext_fetch_hook(fetch());
7          match f {
8            F_RVC(h) => { let ast = decodeCompressed(h);
9                          (execute(ext_post_decode_hook(ast)), true) ... }
10           F_Base(w) => { let ast = decode(w);
11                          nextPC = PC + 4;
12                          (execute(ext_post_decode_hook(ast)), true) ... }
13 }
```

- The step first checks for interrupts and handles it if there is one.
- If not, it fetches and instruction and decides whether its an RVC (compressed) instruction or a base instruction.
- In each case, it decodes it and executes it.

Taking stock ...

By this time we hope you're getting the hang of reading the Sail code that expresses the semantics of RISC-V instructions. Some observations:

- In many senses, Sail is "just another" programming language. Many of its notations and features are taken from or inspired by the functional programming language OCaml (which, in turn, was inspired by SML).
- Expressing the semantics of RISC-V instructions is an exercise in coding in this programming language.
- Features like numeric types with type-checking, scattered definitions, mappings, bit-vectors with type-encoded lengths all make it into a DSL (Domain Specific Language) for expressing ISAs.
- Sail's simple, clean, semantics make it suitable for connecting to well-known formal-method tools (such as Coq, Isabelle, HOL4).



Overview of module dependencies in the Sail RISC-V spec.

(Original: `doc/figs/riscvspecdeps.svg` in repo `https://github.com/rems-project/sail-riscv`)

Executing ISA Tests

Ensure you have downloaded/built/installed executable versions of the formal spec

If you have not already done so, please follow both Step A and Step B described in slide deck "`Slides_Installation.pdf`" in the repository:

`https://github.com/rsnikhil/RISCV_ISA_Spec_Tour`

Step A clones `https://github.com/rems-project/sail-riscv`, with the Sail RISC-V spec in the "`model`" directory (we've been studying this code so far in this tutorial).

Step B takes you through these steps:

- Install Opam, the package manager for OCaml;
- Using Opam, install OCaml
- Using Opam, install Sail
- Using Ocaml and Sail, build executable versions of the Sail RISC-V spec.

As a result, you should now have the following executables in your sail-riscv repository:

in your sail-riscv repository clone

```
$ pwd; ls c_emulator/riscv_sim_RV*
/home/nikhil/git_clones/sail-riscv
c_emulator/riscv_sim_RV32*  c_emulator/riscv_sim_RV64*
```

Executing standard RISC-V ISA tests

The directory "`sail-riscv/test/riscv-tests/`" has a full suite of pre-compiled standard RISC-V ISA tests. Each has an ELF file (RISC-V binary) and a disassembly (text file) of the test. Examples:

Example of ISA test ELF file and disassembly text file

```
$ ls -1 test/riscv-tests/rv64ui-p-add*
test/riscv-tests/rv64ui-p-add.elf          RISC-V executable
test/riscv-tests/rv64ui-p-add.dump         Disassembly text file
test/riscv-tests/rv64ui-p-addi.elf
test/riscv-tests/rv64ui-p-addi.dump
test/riscv-tests/rv64ui-p-addiw.elf
test/riscv-tests/rv64ui-p-addiw.dump
test/riscv-tests/rv64ui-p-addw.elf
test/riscv-tests/rv64ui-p-addw.dump
```

Executing standard RISC-V ISA tests (contd.)

Using the C-based simulator we can execute, for example, the "`rv64ui-p-add`" ISA test for the ADD instruction:

in your sail-riscv repository clone

```
$ pwd
/home/nikhil/git_clones/sail-riscv

$ ./c_emulator/riscv_sim_RV64  test/riscv-tests/rv64ui-p-add.elf
Tue Dec 10 07:37:05 2019
...
Running file test/riscv-tests/rv64ui-p-add.elf.
ELF Entry @ 0x80000000
CSR mstatus <- 0x0000000A00000000 (input: 0x0000000000000000)
mem[X,0x0000000000001000] -> 0x0297
mem[X,0x0000000000001002] -> 0x0000
[0] [M]: 0x0000000000001000 (0x00000297) auipc t0, 0
...
[1] [M]: 0x0000000000001004 (0x02028593) addi a1, t0, 32
...
[2] [M]: 0x0000000000001008 (0xF1402573) csrrs a0, zero, mhartid
...
[477] [M]: 0x0000000080000044 (0xFC3F2023) sw gp, 4032(t5)
htif[0x0000000080001000] <- 0x00000001
htif-syscall-proxy cmd: 0x000000000001
SUCCESS
```

During execution of the RISC-V binary, it prints out a trace of instructions executed (PC, instruction, assembly).

Executing standard RISC-V ISA tests (contd.)

Another example: the "`rv32um-v-mulhsu`" test for the MULHSU instruction in virtual-memory mode:

in your sail-riscv repository clone

```
1  Tue Dec 10 07:46:33 2019
2  Running file test/riscv-tests/rv32um-v-mulhsu.elf.
3  ELF Entry @ 0x80000000
4  [0] [M]: 0x00001000 (0x00000297) auipc t0, 0
5  ...
6  [20652] [S]: 0xFFC02270 (0x0106A023) sw a6, 0(a3)
7  htif[0x80001000] <- 0x00000001
8  htif-syscall-proxy cmd: 0x000000000001
9
10 SUCCESS
```

During execution of the RISC-V binary, it prints out a trace of instructions executed (PC, instruction, assembly).

You can execute any of the "`*.elf`" tests in directory "`sail-riscv/test/riscv-tests/`" in the same way.

Executing standard RISC-V ISA tests (contd.)

FYI, for those who wish to explore the OCaml-based simulators and/or connections to various formal tools.

The "`make`" commmand in Step B.4 of the "`Slides_Installation.pdf`" (without the "`csim`" argument) also makes:

- OCaml-based executable versions of the spec, in directory "`./ocaml_emulator/`". These are run in the same way as the C-based simulators of the previous examples.
- Material to connect to formal tools Coq, Isabelle, HOL4, etc. Please see documentation in the repository about these options.

Executing Compliance Tests

Ensure you have downloaded/built/installed executable versions of the formal spec

[This is a repeat of Slide 59]

If you have not already done so, please follow both Step A and Step B described in slide deck "`Slides_Installation.pdf`" in the repository:

`https://github.com/rsnikhil/RISCV_ISA_Spec_Tour`

Step A clones `https://github.com/rems-project/sail-riscv`, with the Sail RISC-V spec in the "`model`" directory (we've been studying this code so far in this tutorial).

Step B takes you through these steps:

- Install Opam, the package manager for OCaml
- Using Opam, install OCaml
- Using Opam, install Sail
- Using Ocaml and Sail, build executable versions of the Sail RISC-V spec.

As a result, you should now have the following executables in your sail-riscv repository:

in your sail-riscv repository clone

```
1 $ pwd; ls c_emulator/riscv_sim_RV*
2 /home/nikhil/git_clones/sail-riscv
3 c_emulator/riscv_sim_RV32*  c_emulator/riscv_sim_RV64*
```

Executing Compliance tests: preparation

Clone a copy of the RISC-V Foundation's "Compliance" repository:

```
$ git clone  https://github.com/riscv/riscv-compliance
```

Set up your environment for RISC-V compiler tools (the Compliance scripts will use this to re-compile compliance tests).

```
$ export RISCV=<your toolchain_installation_dir>/riscv64
$ export PATH=$RISCV/bin:$PATH
```

Spot check that we've got the toolchain setup:

```
$ which riscv64-unknown-elf-gcc
/home/nikhil/Projects/RISCV/Tests/RISCV_Gnu_Toolchain/riscv64/bin/riscv64-unknown-elf-gcc

$ riscv64-unknown-elf-gcc  --version
riscv64-unknown-elf-gcc (GCC) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Executing Compliance tests: preparation (contd.)

Setup up your PATH environment variable to include a your clone-directory of the "sail-riscv" repository, so that the Compliance scripts know where to find the executable versions of the Sail RISC-V spec:

```
1
2  $ export SAIL_RISCV=<path to your clone of sail-riscv repository>/sail-riscv
3  $ export PATH=$SAIL_RISCV/c_emulator:${PATH}
```

Check:

```
1  $ which riscv_sim_RV32  riscv_sim_RV64
2  /home/nikhil/git_clones/sail-riscv/c_emulator/riscv_sim_RV32
3  /home/nikhil/git_clones/sail-riscv/c_emulator/riscv_sim_RV64
```

Executing Compliance tests (contd.)

Finally, the following will execute all relevant variants of the Compliance test suite:

In your riscv-compliance repository clone

```
$ pwd
/home/nikhil/git_clones/riscv-compliance

$ make RISCV_TARGET=sail-riscv-c all_variant
for isa in rv32i rv32im rv32imc rv32mi rv32si rv32ua rv32uc rv32ui rv64i rv64im; do \
    ...
...
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 ...
riscv_sim_RV32  --test-signature=...  ....elf 2>  ....log 1>&2
...
Compare to reference files ...

Check          I-ADD-01 ... OK
Check         I-ADDI-01 ... OK
...
--------------------------------
OK: 48/48
...
```

Executing Compliance tests (contd.)

If you examine the transcript, you will see the results for each of the ISA groups mentioned in the "for isa in rv32i, rv32im, ..." line at the top:

In your riscv-compliance repository clone

```
$ make RISCV_TARGET=sail-riscv-c all_variant
for isa in rv32i rv32im rv32imc rv32mi rv32si rv32ua rv32uc rv32ui rv64i rv64im; do \
... OK: 48/48
... OK: 8/8
... OK: 25/25
... OK: 9/9
... OK: 6/6
... OK: 10/10
... OK: 1/1
... OK: 39/39
... OK: 9/9
... OK: 4/4
```

(One test, "`lrsc`", is marked IGNORE instead of OK. I'm not sure why; perhaps because it is uninteresting in this single-hart sequential execution?)

Extra Topic: Adding new ISA extensions to the Formal Spec

Adding new ISA Extensions

This slide repeats information from Slide 41

The general scheme for adding a new instruction, or new class of instructions, should be clear by now:

- Define an enum and mapping for any sub-opcodes in the class (if the class contains more than one instruction)
- Augment the "`ast`" type by adding a scattered clause to describe this new class
- Augment the "`encdec`" mapping by adding a scattered clause to describe this new class
- Augment the "`execute`" function by adding a scattered clause to describe this new class
- Augment the "`assembly`" mapping by adding a scattered clause to describe this new class

It is a stylistic judgement call whether you define a class with sub-opcodes, or just define a separate clause for each instruction in the class. E.g., we could have defined separate "`ast`" , "`encdec`", "`execute`" and "`assembly`" clauses for BEQ, BNE, BLT, ...

A class with sub-opcodes makes sense when the instructions share structure and semantics. For example, BEQ/BNE/BLT/... differ only in the particular comparison operator; using a class with sub-opcodes captures this similarity.

Case study: adding ISA Extensions F and D (single and double-precision floating point)

This is still in a branch of the repo, nearing final development, will be merged into the main master soon.
You can view the source files by switching to the development branch:

In your riscv-compliance repository clone

```
$ git branch --all
* master
  ...
  remotes/origin/rsnikhil

$ git checkout  remotes/origin/rsnikhil
$ git branch
* (HEAD detached at origin/rsnikhil)
  master
```

Case study: adding ISA Extensions F and D (contd.)

Since F,D define a new register width "`FLEN`", we created two new files by analogy with "`riscv_xlen32.sail`" and "`riscv_xlen64.sail`". (Not necessary for extensions that don't define new widths).

In sail-riscv/model

```
1 riscv_flen_F.sail    riscv_flen_D.sail
```

Since F,D define new registers, we created two new files by analogy with "`riscv_reg_type.sail`" and "`riscv_regs.sail`". (Not necessary for extensions that don't define new registers).

In sail-riscv/model

```
1 riscv_freg_type.sail
2 riscv_fdext_regs.sail
```

Since F,D define new CSRs, we created a new file by analogy with "`riscv_sys_control.sail`". (Not necessary for extensions that don't define new CSRs).

In sail-riscv/model

```
1 riscv_fdext_control.sail
```

Case study: adding ISA Extensions F and D (contd.)

Finally, the semantics of F and D instructions are in two new files:

In your `riscv-compliance` repository clone

```
1 riscv_insts_fext.sail
2 riscv_insts_dext.sail
```

They follow the standard pattern:

- Define an enum and mapping for any sub-opcodes in the class (if the class contains more than one instruction)
- Augment the "`ast`" type by adding a scattered clause to describe this new class
- Augment the "`encdec`" mapping by adding a scattered clause to describe this new class
- Augment the "`execute`" function by adding a scattered clause to describe this new class
- Augment the "`assembly`" mapping by adding a scattered clause to describe this new class

Case study: adding ISA Extensions F and D (contd.)

In addition to writing the Sail code, you have to update the Makefiles so that your new files are included. They look something like this:

In your riscv-compliance repository clone

```
$ git diff Makefile

+# Currently, we only have F with RV32, and both F and D with RV64.
 ifeq ($(ARCH),RV32)
   SAIL_XLEN := riscv_xlen32.sail
+  SAIL_FLEN := riscv_flen_F.sail
 else ifeq ($(ARCH),RV64)
   SAIL_XLEN := riscv_xlen64.sail
+  SAIL_FLEN := riscv_flen_D.sail
 else
   $(error '$(ARCH)' is not a valid architecture, must be one of: RV32, RV64)
 endif

...
```

Case study: adding ISA Extensions F and D (contd.)

In addition to writing the Sail code, you have to update the Makefiles so that your new files are included. They look something like this:

In your riscv-compliance repository clone

```
$ git diff Makefile

...

+SAIL_DEFAULT_INST += riscv_softfloat_interface.sail  riscv_insts_fext.sail
+ifeq ($(ARCH),RV64)
+SAIL_DEFAULT_INST +=riscv_insts_dext.sail
+endif


+SAIL_SYS_SRCS += riscv_fdext_regs.sail riscv_fdext_control.sail

+PRELUDE = prelude.sail prelude_mapping.sail $(SAIL_XLEN) $(SAIL_FLEN) ...

+SAIL_REGS_SRCS = riscv_reg_type.sail riscv_freg_type.sail riscv_regs.sail ...

+SAIL_ARCH_SRCS += $(SAIL_FD_SRCS)
```

Extra Topic: The larger context of Sail usage

The larger context of Sail usage

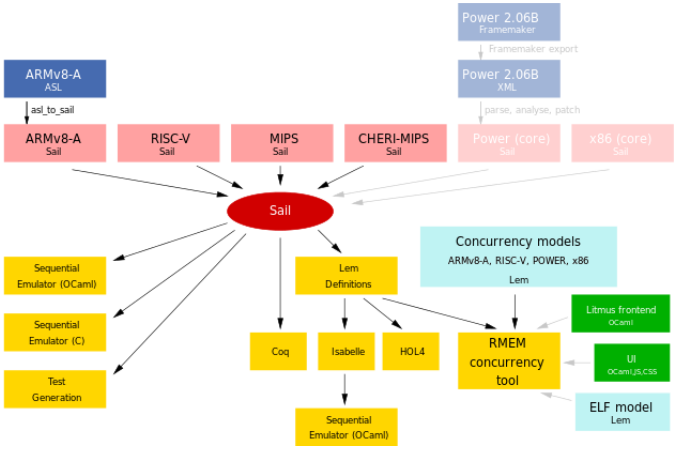
In addition to RISC-V, Sail has also been used to specify other ISAs. In addition to the so-called "C-based" back-end we've seen in this tutorial, it has more back-ends to create an OCaml-based simulator, and to connect to various formal environments such as Coq, Isabelle, HOL4, etc.

Recommended reference with more details:

> *ISA Semantics for ARMv8-A, RISC-V, and Cheri-MIPS*,
> Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, Peter Sewell
> in *Proc. 46th ACM SIGPLAN Symp. on Principles of Programming Languages (POPL), Cascais/Lisbon, Portugal, Jan 13-19, 2019*, pp. 71:1–71:31.

See also the figure on the next slide.

Sail: the larger picture (contd.)

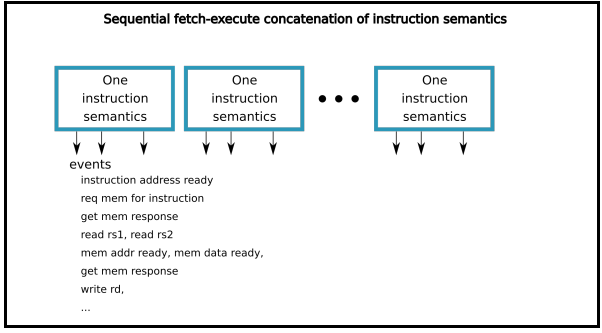


(from: `https://github.com/rems-project/sail`)

Extra Topic: Concurrency and connection to RISC-V Weak Memory Model

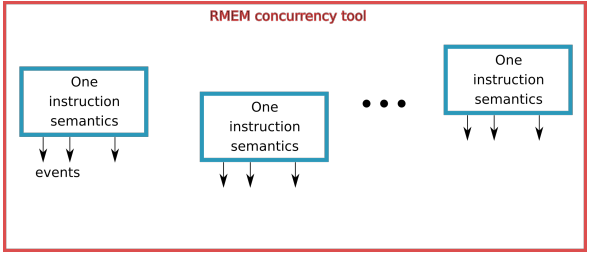
Concurrency

- In this introductory tutorial we deliberately stayed away from questions of concurrency, which is a more advanced topic.
- Each instruction semantics can be regarded as a small sequential thread performing that instruction's semantics. There are various "events" during this thread's progress.
- In our simple one-instruction-at-a-time fetch-execute loop model shown in this tutorial (and in the default simulators built), these threads are simply concatenated into an overall single sequential thread.



Concurrency (contd.)

- A parallel model can overlap these threads:
  - a pipeline model may launch the next instruction's thread before the current one has finished; in fact can launch it speculatively based only on PC of previous instruction;
  - a superscalar model may launch two or more of these threads together;
  - an out-of-order model may have many of these threads running concurrently;
  - register read/write events can model renamed registers;
  - memory address/read/write events can interact with a model of weakly orderer memory;
  - and so on.
- The RMEM concurrency tool is meant for these purposes (`https://github.com/rems-project/rmem`)



Connection to official RISC-V Memory Model

- RISC-V's Weak Memory Model was developed by a separate RISC-V Foundation Technical Group, chaired by Dan Lustig of NVidia.
- One of their formalizations was indeed using these Sail and RMEM system and models. As mentioned in the previous slide, this uses a concurrent fetch-execute model where multiple instructions may be in flight concurrently, with concurrent interactions with the weak memory model.
- (These should be covered in another, more advanced tutorial.)

Status and Plans

Status and Plans

- The RISC-V Foundation set up a Technical Group (TG) in 2017 to develop a Formal Specifcation for the RISC-V ISA, which will become the *definitive* specification of the ISA. There are currently 110 members in the TG, of which 15-20 were regular attendees of weekly con-calls.
- The group encompassed 6 different, parallel, efforts, including this one in Sail. In March/April 2019 we put out a public review of these efforts. Based on the feedback and discussions in the TG, we decided to move forward with Sail as the main canonical spec. It is also the most advanced of the 6 efforts in terms of feature coverage and modeling concurrency. The other 5 efforts continue, and over time could achieve peer status with this Sail spec.
- The immediate priority is to clean up, polish, document, and make it easily accessible to the community (of which this tutorial is a part).

How people use ISA formal specs

People are already using and will use the ISA formal spec in various ways.

- As a reference to clarify the intended semantics of each type of instruction.
- As a "golden reference model" against which to compare other implementations (simulation to hardware designs) for correctness. Specific examples include official Compiance Tests, and Tandem Verification.
- In a tool to generate ISA tests automatically.
- In a tool to measure instruction coverage automatically.
- In a tool to formally prove a separately-written implementation correct, by directly correlating the ISA formal semantics with the sematics of the language of the implementation (C, C++, SystemVerilog, ...). Implementations may be complex: pipelined, superscalar, out of order, speculative, ...
- In a tool to formally and systematically derive an implementation from the ISA formal spec using a series of derivations, each formally proved correct ("correct-by-construction implementation").