An Introduction to the Official Formal Specification of the RISC-V Instruction Set Architecture

Rishiyur S. Nikhil, PhD [1]
CTO and co-founder, Bluespec, Inc.
(nikhil@bluespec.com)

3rd RISC-V Meeting, March 30-31, 2021, France

[1] Chaired the RISC-V Foundation's Technical Committee that selected/developed the ISA Formal Spec

Introduction

- The RISC-V ISA Formal Spec is intended to be the ***official*** and ***definitive*** specification of the RISC-V ISA (Instruction Set Architecture)
- I.e., it is the “ultimate truth” against which the (functional) correctness of all implementations should be measured
  - Hardware implementations
  - Simulators and emulators
  - Compilers

Disclaimer: These are not official positions of RISC-V International Association.
Any inaccuracies here are solely the responsibility of the speaker.

What is an “ISA Formal Spec”?

- It is a description of each instruction, written in a formal language (not English prose)
- Describes:
  - Architectural machine state
  - Bit encoding of instructions
  - The “meaning” of each opcode, i.e., “what does this instruction do?”
  - (Nice to have:) An assembly syntax
  - All the different allowed parameterizations
  - All the different allowed non-deteriminisms
    - Including, in particular, interactions with RISC-V Weak Memory Models

- We also want the formal language to have these properties:
  - Have simple unambiguous semantics (so that instruction semantics are clear)
  - Be easily machine readable and manipulable (so we can connect to formal tools, perform formal analyses and transformations, etc.)
  - Be executable (so we can use the formal spec directly as a “correct” simulator)
  - Be approachable and understandable every day by working engineers

A RISC-V ISA simulator written in C/C++ would satisfy some, but not all of these requirements.

The RISC-V ISA Formal Spec is written in *Sail* [1]

- Developed by Prof. Peter Sewell's group at U.Cambridge, UK
- Sail is purpose-designed to describe ISAs
  - Has also been used to describe ISAs of ARM, MIPS, parts of PowerPC and x86
  - Has been deliberately designed to be similar to traditional ISA specs (written with English prose and sometimes also with pseudo-code), with great effort to make it familiar to and easily usable by working engineers:
    - All aspects of an instruction (bit encoding, fields, execution semantics, assembly syntax) are in a textually contiguous group
- The Sail implementation (i.e., code that processes Sail code) is written in OCaml
  - Back-ends produce simulators in C and OCaml
  - Back-ends connect to other formal tools: Coq, Isabelle, HOL4
- Public repository for Sail: https://github.com/rems-project/sail

[1] *Sail* is just a name, not an acronym

Example: Sail spec for LUI and AUIPC instructions

The text spec document for the RISC-V ISA shows that RISC-V instructions are encoded in a few 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 LUI (“Load Upper Immediate”) and AUIPC (“Add Upper Immediate to PC”) instructions have the “U-type” encoding format. The two instructions are distinguished by the 7-bit “opcode” field. The 20-bit immediate value is loaded (LUI) or added to the current PC and loaded (AUIPC) into the destination register (named with a 5-bit index “rd”) after suitable alignment, sign-extension, etc.

Example: Sail spec for LUI and AUIPC instructions: encoding

imm[31:12]	rd	opcode	U-type

The Sail code first defines an AST (Abstract Syntax Tree) for the U-type format:

```
enum uop = {RISCV_LUI, RISCV_AUIPC}
union clause ast = UTYPE : (bits(20), regidx, uop)
```

Declared elsewhere as `bits(5)`

Then, a bit-encoding for the 7-bit 'opcode' part of the instruction:

```
mapping encdec_uop : uop <-> bits(7) = {  RISCV_LUI   <-> 0b0110111,
                                          RISCV_AUIPC <-> 0b0010111 }
```

Then, the bit-encoding for whole instruction:

bit-concatenation

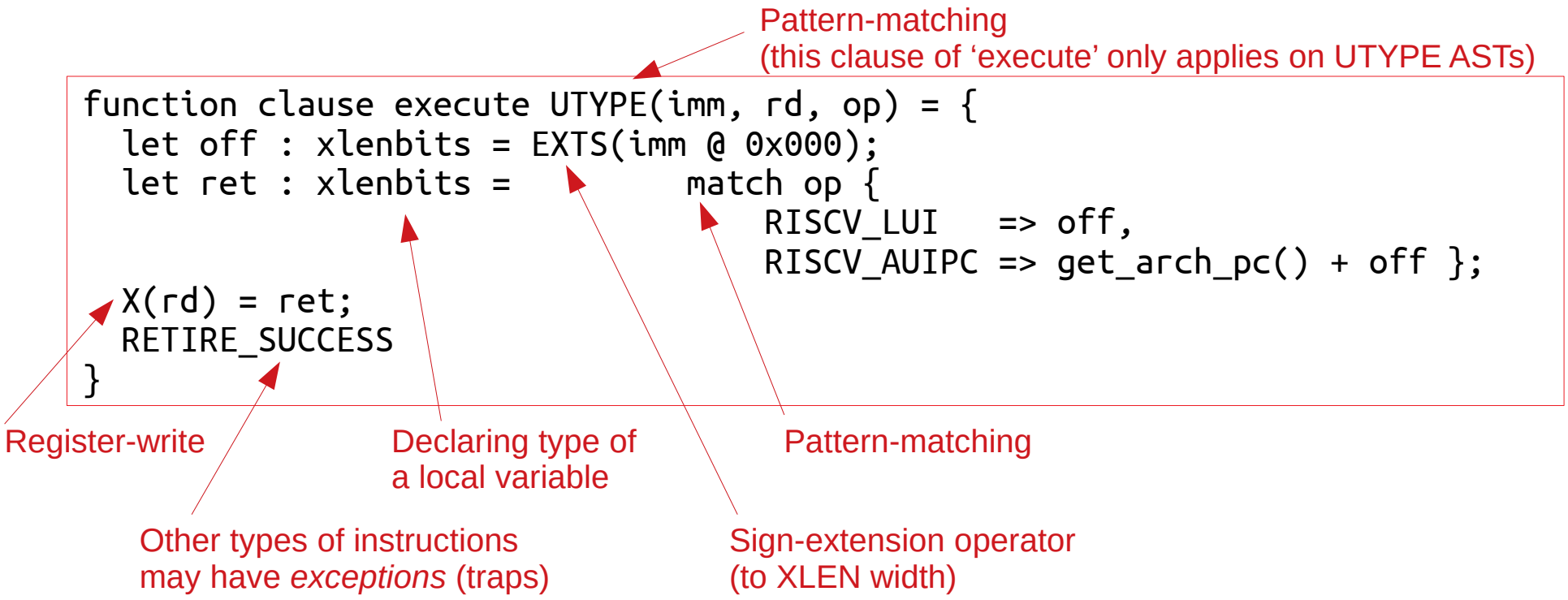
```
mapping clause encdec = UTYPE(imm, rd, op) <-> imm @ rd @ encdec_uop(op)
```

The AST

The bit representation

Example: Sail spec LUI and AUIPC: execution semantics

The execution semantics is defined by a function on the AST (on a global architectural state):



Here the specs for LUI and AUIPC are given together since they share much structure, but this is a stylistic choice based on readability; they can be given individually.

Example: Sail spec LUI and AUIPC: assembly syntax

The assembly syntax is defined by a mapping between the ASTs and strings:

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

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

AST
String

String concatenation

string space

string separator
(e.g., comma)

Sail: “Scattered” definitions

In most programming languages, a data type declaration must contain all its “clauses”:

```
union ast = UTYPE : (bits(20), regidx, uop)
          | BTYPE : (bits(13), regidx, regidx, bop)
          | ...
```

In Sail, the clauses can be “scattered” across the program text:

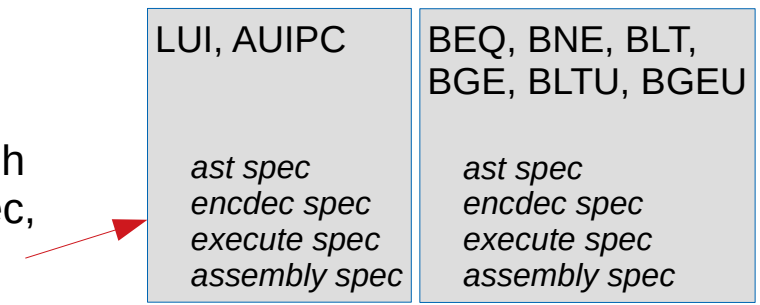
```
union scattered ast
```

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

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

Similarly, functions and mappings can also be “scattered”.

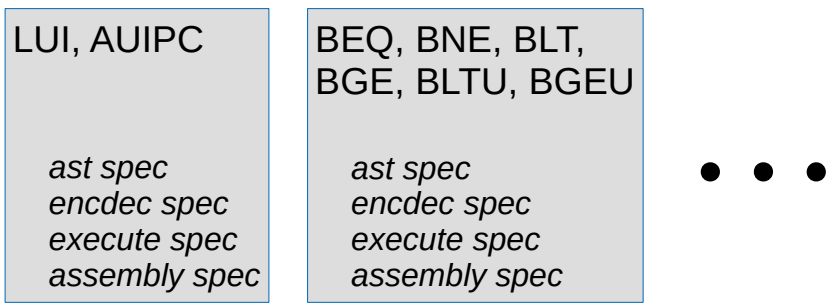
Thus, the program text can group together the clauses for each opcode, so the spec is organized just like a traditional ISA spec, with all aspects of each opcode grouped into its own section.



Adding new instructions (e.g., custom instructions)

This “all aspects of an instruction together” makes it easy to add new instructions as new Sail code without disturbing Sail code for existing instructions.

Existing instruction code can be used as an “example” or “template” for adding new instructions.



Sail: Types and Type-checking

Sail is a strongly typed, statically type-checked language

- Types include
  - enums (e.g., `enum bop = {BEQ, BNE, BLT, BGE, BLTU, BGEU}`)
  - algebraic types (like `UTYPE` and `BTYPE`, shown earlier)
    - (algebraic types are also known as “tagged unions”)
  - tuples (like the 3-tuple component of a `UTYPE`, shown earlier)
  - bit-vectors (like `bits(20)`, shown earlier)
    - bit-vector sizes are part of the type, and are type-checked: it is impossible to accidentally pass a bit-vector of the wrong size
  - functions, and so on
- Types can be polymorphic (e.g., a function operating on different bit-vectors of different sizes)

A more complex example: Sail ‘execute’ spec for LOAD (1)

The LOAD group of instructions encompass:

- an rs1 source register for the address, and a 12-bit immediate offset to the address
- an rd destination register for the loaded value
- different sizes: byte (8b), halfword (16b), word (32b), doubleword (64b)
- whether the loaded value should be zero- or sign-extended into rd
- “acquire” and “release” annotations (for weak-memory models)

```
function clause execute(LOAD(imm, rs1, rd, is_unsigned, width, aq, rl)) = {
  let offset : xlenbits = EXTS(imm);
  match ext_data_get_addr(rs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
… (continued) …
```

Sign-extend 12b offset

Compute address to be loaded

Argument indicating this is a ‘read’ to ‘data memory’ (not instruction mem)

Pattern-match on whether addr computation failed or succeeded with (virtual) address vaddr

Trap actions

A more complex example: Sail ‘execute’ spec for LOAD (2)

The spec can be configured to model traps on misaligned addresses

```
… (continuing) …
  if   check_misaligned(vaddr, width)
  then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
  else match translateAddr(vaddr, Read(Data)) {
… (continued) …
```

Trap actions

Compute the physical address from the virtual address (if virtual memory is currently active).

The function also models TLBs (Translation-Lookaside Buffers) because these are potentially architecturally visible, needing the SFENCE.VMA instruction.

A more complex example: Sail ‘execute’ spec for LOAD (3)

Virtual-to-physical translation can fail

```
… (continuing) …
  else match translateAddr(vaddr, Read(Data)) {
         TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
         TR_Address(addr, _) =>
           match (width, sizeof(xlen)) {
             (BYTE, _)   =>
                process_load(rd, vaddr,
                             mem_read(Read(Data), addr, 1, aq, rl, false),
                             is_unsigned),
             (HALF, _)   =>
                process_load(rd, vaddr,
                             mem_read(Read(Data), addr, 2, aq, rl, false),
                             is_unsigned),
            … (and so on for WORD and DWORD) …
```

Physical address

Move memory value into register rd

Do the actual read from the memory model

Configuration options (“implementation-defined”)

RISC-V permits a number of “implementation-defined” options:

- In the unprivileged ISA. Examples:
  - RV32I vs. RV64I
  - Each of the optional extensions: M (integer multiply/divide), A (atomics), FD (single-and-double-precision floating point), C (compressed), … and more in the future
  - Whether or not a LOAD/STORE traps if the address is misaligned

- And many more in the privileged ISA: Examples:
  - Many CSRs (Control and Status Registers) have “write-any, read-legal” (WARL) fields.
    An implementation can define how an illegal write-value is transformed into a legal read-value.
  - A (“accessed”) and D (“dirty”) bits in a PTE (Page Table Entry) can be maintained in the implementation or raise a trap.

The Sail spec is easy to configure for a specific set of choices, at the moment with some editing of Sail source files.

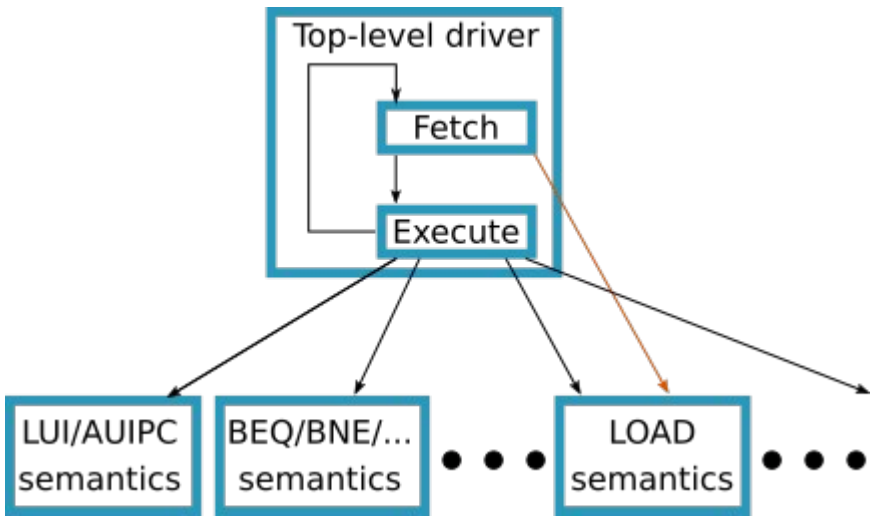
A more systematic way of *specifying* all officially-allowed implementation choices, and automatically incorporating this into the Sail spec, is under development.

From individual instructions to programs (sequences of instructions)

- Sequential semantics is easy (and is adequate for many uses).
- Concurrent semantics (modeling concurrency of pipelines, superscalarity, out-of-order, speculation, multi-harts (Hardware Threads) etc. is more complex and subtle because of Weak Memory Models, which allow certain kinds of non-determinism to become visible in the semantics.

Sequential semantics (simple) → sequential simulator

To create a simple one-instruction-at-a-time sequential simulator from the formal spec, we simply write a Sail top-level fetch-execute loop that invokes per-instruction semantic functions.



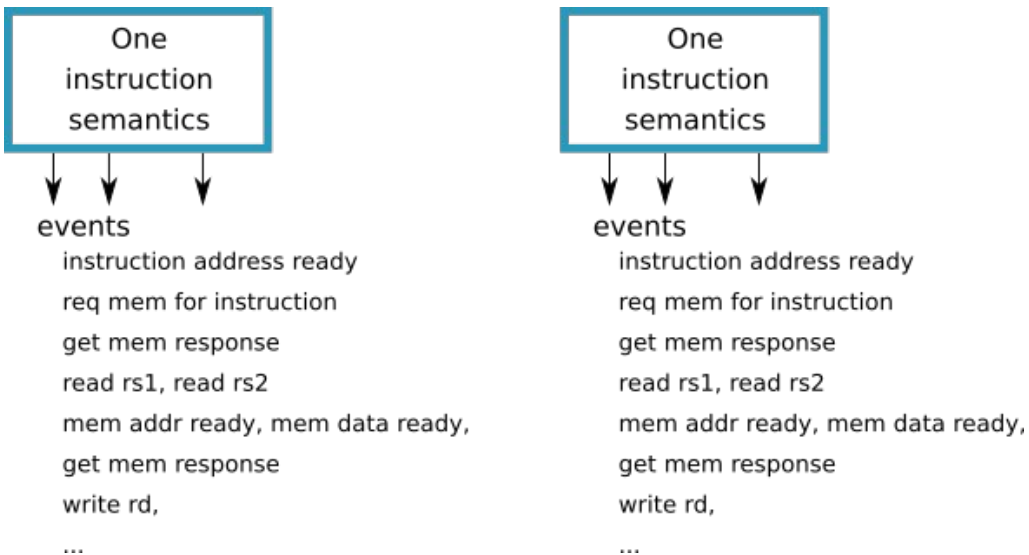
Concurrent semantics and Weak Memory Models

Implementations have various kinds of concurrency: pipelineing, speculation, superscalarity, out-of-order execution, and multi-harts (multi-thread, multi-core; hart = “hardware thread”).

With instruction concurrency and weak memory models, we can no longer treat each instruction execution as an atomic action (as in the simple sequential semantics).

Execution of an instruction involves a series of events,
and the events of concurrent instructions can interleave,
exposing non-deterministic behavior.

(Hence the need for instructions like FENCE, FENCE.I, SFENCE.VMA)

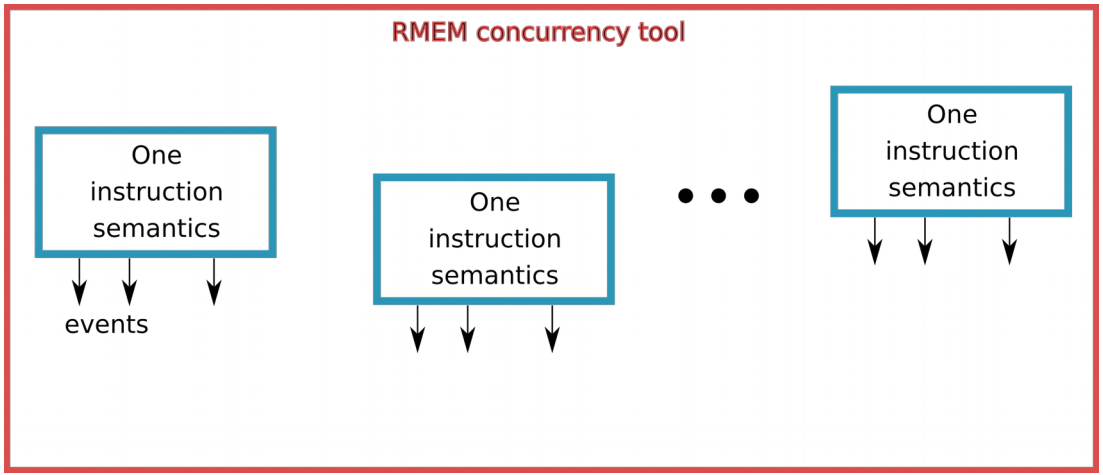


Instruction concurrency and Weak Memory Models

The Sail formal spec can be used to model the full range of allowed non-determinisms.

- The semantics are written in terms of an “API” for basic actions, such as reading/writing a register, address calculation, reading/writing from memory, etc.
- For sequential semantics, these are implemented in the natural, intuitive way.
- For concurrent semantics, these can also announce/record the “events” during an instruction’s execution.
- An alternative top-level can interleave events from concurrent instructions.

The Sail project includes the REM concurrency tool which has been used to specify the RISC-V Weak Memory Model(s), and which can interact with the instruction semantics.



Using the RISC-V ISA Formal Spec

There are many ways to use the ISA Formal Spec

- It is ready for some uses *now*, in everyday activities
- Some are more researchy, but open the door for future formal assurance about designs (correctness, security, ...)

Use-case: As an everyday document to consult

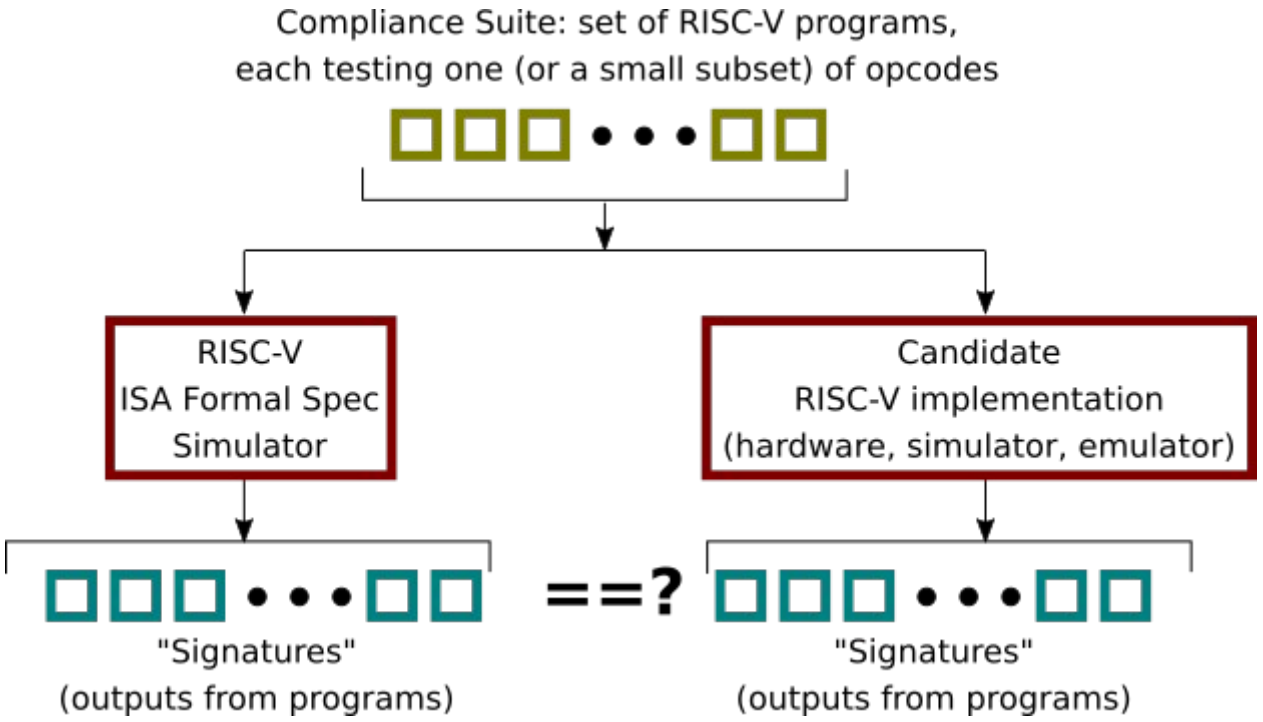
- I hope this will be the most common use-case for the ISA Formal Spec.
- *Every* engineer who works with RISC-V (hardware CPU designer, hardware CPU verification engineer, compiler writer targeting RISC-V, simulator writer, …) should
  - Bookmark the Formal Spec link in their browser
  - Consult the Formal Spec to clarify any doubts about any RISC-V instruction

- Anyone teaching the RISC-V ISA, and anyone learning the RISC-V ISA, should use the Formal Spec as an integral part of their activity.

In other words, people should use the Formal Spec as often and naturally as a traditional text ISA spec.

Use-case: “Golden Reference” for Compliance Suite

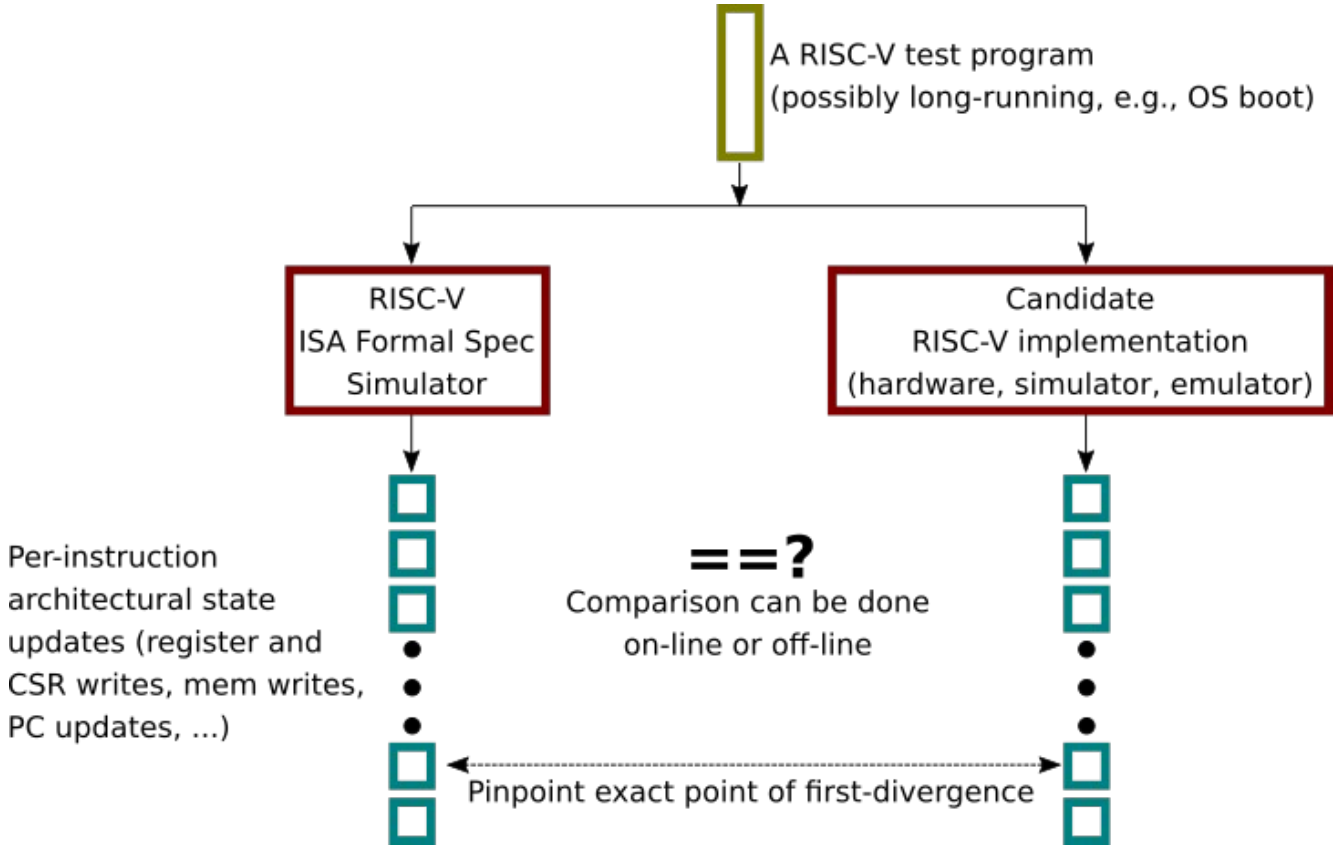
- This may be the next most common use-case for the ISA Formal Spec.
- Before a commercial implementation can claim that it “is a RISC-V implementation”, it must pass the RISC-V Compliance Suite.



In general, the ISA Formal Spec simulator can be seen as a “golden reference model” during verification of functional correctness of any RISC-V implementation.

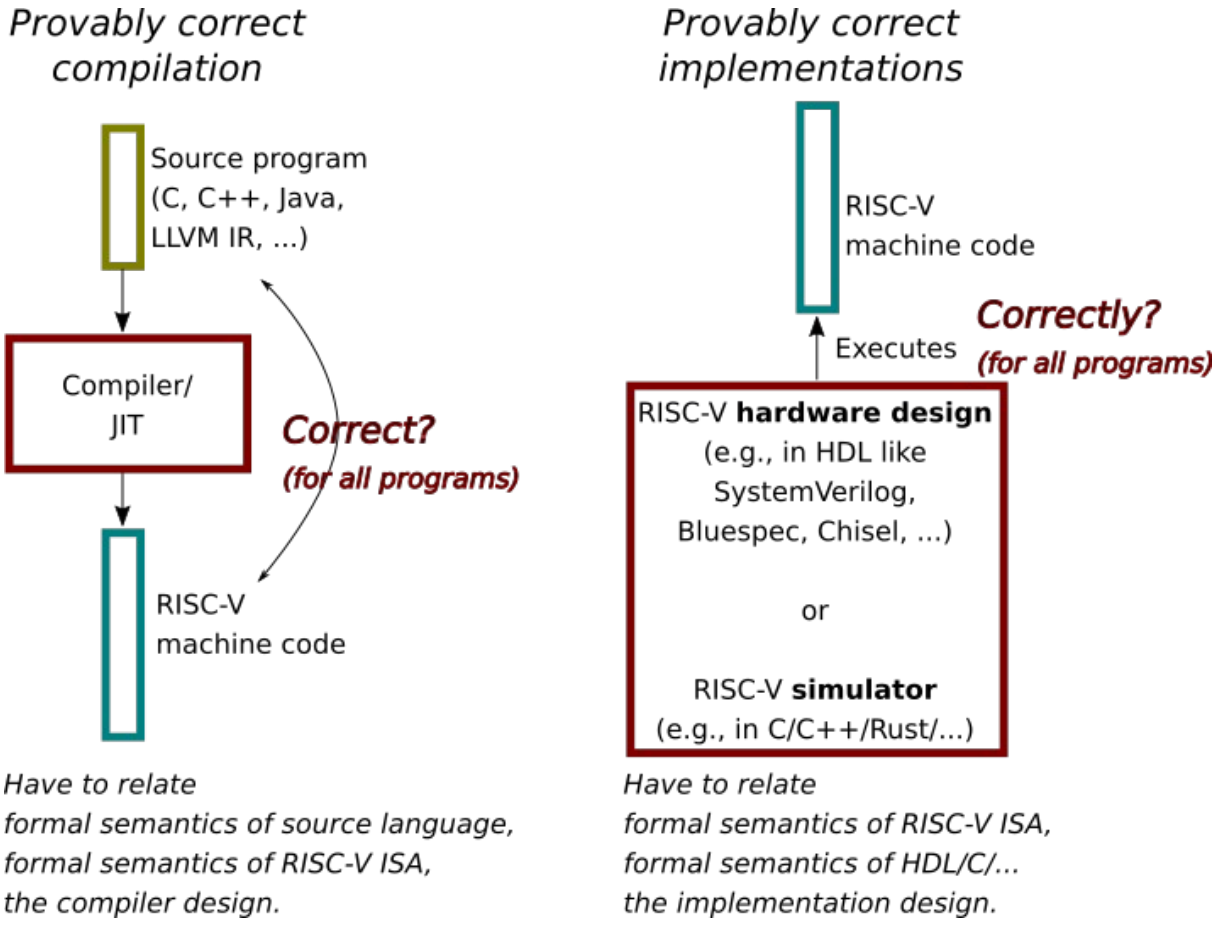
Use-case: “Tandem Verification”

Tandem Verification uses the ISA Formal Spec as a golden reference in a more fine-grain (instruction-by-instruction) manner, so that faulty behavior can be pinpointed precisely.



Use-cases: Formal proofs of correctness/equivalence

Several research groups are pursuing these ideas.



RISC-V ISA Formal Spec: Status and Plans

The Formal Spec currently covers:

- Unprivileged RV32I, RV64I
- Extensions M (int mult/div), A (atomics) FD (single-, double-precision float), C (compressed)
  - FD currently use capture all RISC-V-specific aspects (floating point registers, CSRs, load/store) but invoke Berkeley “Softfloat” for actual computation; this should change in future
- Privileged spec with M, S and U privileges, Sv32, Sv39 and Sv48 virtual memory

The simulator has been used to boot Linux, FreeBSD, Sel4, FreeRTOS.

The generated simulator via C is fast enough to boot Linux/FreeBSD in a few seconds/minutes.

Ongoing work to improve accessibility:

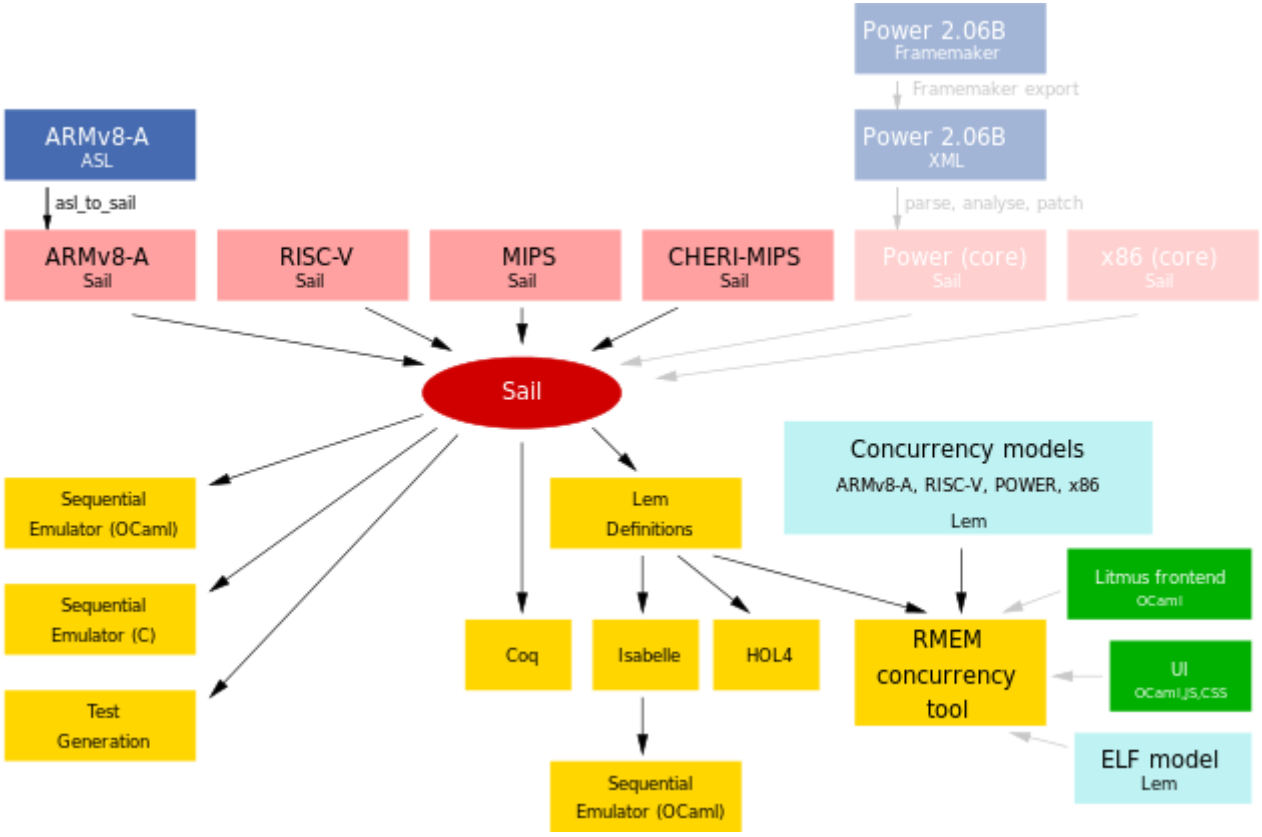
- Integration in-line with existing unprivileged and privileged text spec documents
- General documentation, tutorials, outreach

Ongoing work:

- Integration with RISC-V Compliance Suite
- Formalization of “implementation choices”
- Every new ISA extension will require an extension to the ISA Formal Spec

Please feel free to join in!

The ecosystem around the RISC-V ISA Formal Spec and Sail



Many ISAs have been coded in Sail

The Sail implementation has many back-ends to produce simulators (in C and OCaml) and to connect to other formal tools

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

Thank you!

Questions?

Useful links:

- https://github.com/rems-project/sail-riscv
  The actual RISC-V ISA Formal Spec, written in Sail
- https://github.com/rsnikhil/RISCV_ISA_Spec_Tour
  These slides; longer 3-hour introduction and reading-tour of the actual spec code; instructions on how to download and build an executable version of the spec; how to execute the spec on the RISC-V Compliance Suite
- https://github.com/riscv/riscv-compliance
  The official RISC-V “ISA Compliance Suite” (in development)

For those interested in Sail itself, and its compiler (written in Ocaml) to C, Ocaml, Coq, ...

- https://github.com/rems-project/sail