Jolt-QED: Formally Verifying Bytecode Expansions In Lean

Ari Biswas, *University of Warwick & Layer Zero Labs*, tcs@randomwalks.xyz
Quang Dao, *Carnegie Mellon University & Layer Zero Labs*, qvd@andrew.cmu.edu
Daniel Ross, *Independent Researcher*, daniel23@toastertime.net
Justin Thaler, *a16z crypto & Georgetown University*, justin.r.thaler@gmail.com

Abstract

In this work, we formally verify using the Lean4 proof assistant, that jolt zk-VM's bytecode expansion process faithfully emulates a RISC-V CPU. Prior work translated the official Sail specification of RISC-V into Lean. We extend this model to define the Jolt ISA in Lean. We then write a transpiler to automatically extract Jolt's bytecode expansions into Lean from their Rust definitions. Finally, we prove these expansions equivalent to their corresponding RISC-V instructions. Additionally, this lays the groundwork for proving the upstream components of Jolt are complete and sound.

1 Introduction

Jolt [2] is a state-of-the-art zero-knowledge virtual machine (zk-VM) that powers the proving machinery of blockchains in research and industry [9]. A traditional virtual machine executes a guest program compiled for a specified instruction set architecture (ISA), such as RISC-V, ARM, x86, etc. by emulating the guest CPU in software. A zk-VM is a virtual machine with the added responsibility that it is required to output an efficiently[1]checkable cryptographic proof[2] that it executed every instruction of the guest program correctly. Thus, Jolt being a zk-VM, can be logically split into the tracer and the prover. The tracer takes a RISC-V guest program, denoted here with x, executes instructions in order, and generates a witness w. In the language of computational complexity theory, the guest program x is often referred to as the problem instance, and the honest trace witness w is referred to as the NP witness which can be compared with the program to validate that the output was generated by executing the program's actual instructions. Generating an NP witness on its own accomplishes little. A client wishing to confirm the result came from the program could simply run the program to generate its own witness w', and compare it against the prover's witness w. The challenge is to generate a substantially cheaper procedure for validating program execution. The Jolt prover's task, then, can be viewed as compressing the witness w to output a proof $\pi := \pi(w, x)$ that a client can efficiently process to validate whether the program was in fact run correctly.

To efficiently generate a proof, the Jolt zk-VM leverages the fact that *all* instructions in the guest program's ISA satisfy *decomposability* condition [2: page 7]. Intuitively, an instruction is decomposable if it can be evaluated on inputs $(\vec{x}, \vec{y})$ by decomposing the inputs into smaller chunks, evaluating a function on each chunk, and then efficiently combining the outputs of each chunk into the final output. The `AND` instruction, for example, is decomposable. Let $m \in \mathbb{N}$, then for any $\vec{x}, \vec{y} \in \{0,1\}^m$, let $f_{\text{AND}}(\vec{x}, \vec{y}) = (x_{m-1} \parallel ... \parallel x_0) \ \& \ (y_{m-1} \parallel ... \parallel y_0)$ be the bitwise AND operation. We can decompose the computation of the AND function as follows:

[1]By efficient we mean that, if T is the time taken to run the program, then we want to verification to take time $\text{poly}(\log T)$.

[2]These proofs are often referred to as Succinct Arguments of Knowledge (SNARK) in the literature. The *zero knowledge* in the zk-VM is misleading and is an artefact of poor nomenclature originating in early days of SNARK populism. The defining property of the proof is that it is otherwise efficiently verifiable. The proof may or may not be zero knowledge.

$$f_{\text{AND}}(\vec{x}, \vec{y}) = \sum_{i=0}^{m-1} 2^i \cdot f_{\text{AND}}(x_i, y_i) \tag{1}$$

Unfortunately, not all native RISC-V instructions are decomposable in this way. For example, the `SLL rd, rs1, rs2` instruction models the computation $f(x, y) = x \ll y$ where $x, y \in \{0,1\}^{64}$ represent the contents of registers `rs1` and `rs2` respectively. $\ll$ denotes the logical left bit shift operator, and the value $f(x, y) \in \{0,1\}^{64}$ is written into the contents of `rd`. Here, in order to know the bit i of $f(x, y)$, we need to know one bit of x and **every** bit of y. Thus, there is no way to decompose `SLL`.

But the input to Jolt is still a RISC-V program, and the Jolt prover needs decomposability. So we need some mechanism to decompose non-decomposable instructions for efficient proof generation. The approach the Jolt zk-VM takes is to re-write the guest RISC-V program into a program in the Jolt ISA, which consists solely of decomposable instructions, but is still expressive enough to permit all RISC-V instructions to be efficiently rewritten into Jolt ISA instructions. Thus, Jolt re-writes all troublesome non-decomposable RISC-V instructions with a sequence of one or more decomposable Jolt instructions. In order to aid this emulation process the Jolt CPU includes the usual RISC-V registers plus some additional "virtual" general-purpose registers in which it stores intermediate results.

Continuing with the same example from above, the `SLL` RISC-V instruction is replaced with the following sequence of Jolt instructions.

```
VirtualPow2  v1,  rs2
MUL          rd, rs1, v1
```

where `VirtualPow2 v1 rs2` is a Jolt ISA instruction that represents computation which sets the contents of virtual register `v1` with 2^x, where x represents the contents of register `rs2`. As `VirtualPow2` is not present in the RISC-V ISA, it is referred to as a virtual instruction. `MUL` on the other hand is the extension of the RISC-V instruction `MUL` to the Jolt ISA by allowing it to operate on both RISC-V general-purpose registers and Jolt ISA virtual registers. It is not too hard to see that the above sequence is equivalent to the original `SLL` instruction in that value in `rd` is the same whether we run one `SLL` instruction, or the expanded sequence. In summary, the Jolt ISA can be viewed as a strict subset of the RISC-V ISA, plus some virtual instructions, that can operate on RISC-V registers, plus some virtual registers.

Reflecting back on the job of the tracer, in practice, witness generation proceeds in two steps. The Jolt tracer first translates all RISC-V instructions into one or more Jolt ISA instructions (the Jolt documentation refers to this process as *bytecode expansion*). Once translation is complete, the Jolt CPU executes Jolt instructions to generate the witness. *Thus, the Jolt proof guarantees that it ran every Jolt (not RISC-V) instruction of the transformed guest program correctly.* As the honest witness is generated by executing decomposable instructions, it is amenable to be compressed into an efficiently checkable proof. This added bytecode expansion step however, creates another potential for failure. RISC-V execution semantics are complex, and it is now necessary to ensure that the expanded Jolt ISA versions of all RISC-V instructions implement equivalent semantics.

> Our primary contribution in this work is to use the Lean 4 proof assistant to prove that each Jolt bytecode expansion faithfully implements its corresponding RISC-V instruction. In other words, when starting from the same initial machine state, executing the expansion yields the same final RISC-V CPU state as executing the original instruction.

Note that if the above equivalence were not true, then the program that the prover claims to have correctly executed is *different* from the initial guest program. This renders all downstream guarantees of Jolt to be meaningless. Program equivalence is paramount to the soundness of any Jolt claim. Now while the importance of zk-VM correctness is easily justified, two immediate questions arise.

First, why apply formal verification methods for verifying Jolt's bytecode expansion phase? One might ask whether fuzzing or differential testing could provide comparable assurance with far less effort. Randomised testing has uncovered hundreds of compiler bugs [20], and recent work by Christoph Hochrainer, Valentin Wuestholz, and Maria Christakis. [8] found eleven soundness and completeness bugs in three of six tested zk-VMs. Although fuzzing and differential testing are valuable tools for finding concrete failures, they do not guarantee the absence of bugs. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. [3] argue that testing and fuzzing are insufficient to establish functional correctness and cryptographic security, and use the Coq proof assistant to verify an OpenSSL HMAC implementation against its FIPS functional specification. Furthermore, Jolt's unit tests already use differential testing as a security check. Despite these tests, the incompleteness error described in **Section 6** went undetected until it was found through formal verification in this work. Another motivating factor is that Ethereum Foundation. [5] identifies formal verification as part of the security foundation for L1 zkEVMs and invests in the verification of critical components, security proofs, and specifications that match deployed code. Current SNARK implementations are riddled with bugs [15], and formal verification is deemed necessary to establish their correctness and security.

The second question that may arise is why we use the Lean 4 proof assistant. One motivating factor is that researchers from the University of Cambridge, Galois, University of Edinburgh, and Lindy Labs have developed a Sail-to-Lean transpiler that generates Lean definitions from the official RISC-V Sail model [6, 13]. Sail is an industry-accepted, domain-specific language for describing ISAs. The resulting Lean definitions type-check and provide a *trusted* representation of the RISC-V specification against which we can state and prove the equivalence of each Jolt expansion and its corresponding RISC-V instruction. This generated Lean specification has already been used as a reference in formal-verification efforts for other zkVMs. Succinct and Nethermind verified the core RV64 chips of SP1 Hypercube against the official Sail RISC-V specification [14]. Earlier, Nethermind used Lean to formalise selected components of the RISC Zero zkVM [11] and to prove that extracted constraints for SP1's `AddSub` chip conform to its specification [12]. The Ethereum Foundation has funded Lean-based verification work beyond the RISC-V model, including projects on the Jolt zkVM, SP1 ALU chips, and the ArkLib SNARK library [4]. ArkLib is being developed as a modular Lean framework for formalising SNARKs in general, with current work targeting completeness and round-by-round knowledge soundness for selected protocols, including sum-check, Spartan, FRI, STIR, WHIR, and Binius [18]. VCVio provides a separate Lean framework for formalising cryptographic security arguments, including a formally verified forking lemma and Fiat–Shamir transform [17]. Mathlib, which has become the gold standard for automated theorem proving in mathematics, provides the general-purpose library of formalised mathematics on which Lean developments can build [10]. As a consequence of these projects, we believe Lean 4 is a practical environment for formalising the algebraic and cryptographic components used by Jolt.

The remainder of the document is organised as follows. The phrase "formal verification" has become quite overloaded in recent years. Given the promise of strong guarantees of proof assistants, it is tempting to use it as a broad umbrella to advertise the complete security of a zk-VM. The fact still remains that Jolt is written in the rust programming language, and the formal verification claims are using the Lean proving assistant. How one guarantees correctness of high level functions written in rust, with theorem statements in Lean is a nuanced mixture of ecological trust, explicit assumptions, and other ingredients. Thus, in **Section 2**,

we detail exactly how the Lean theorems were generated, and what components of the proof generation of pipeline are assumed to be trusted by default. Only after establishing the exact trust surface, do we in Section 3, describe the Lean representation of the RISC-V CPU, including its machine state and execution semantics, before describing how to define the Jolt ISA on top of it. Theorems are only true, if the corresponding assumptions they make also hold. In Section 4, we make explicit the assumptions with references to the Jolt codebase under which our equivalence theorems hold. These assumptions are just as important as the final theorem statement, and their presence *must* be justified. In Section 5, we describe how we get Lean expansions of RISC-V instructions from their rust definitions. Finally, in Section 6, we summarise the bugs and architectural mis-specifications uncovered during our work and discuss the limitations of the guarantees established by formal verification.

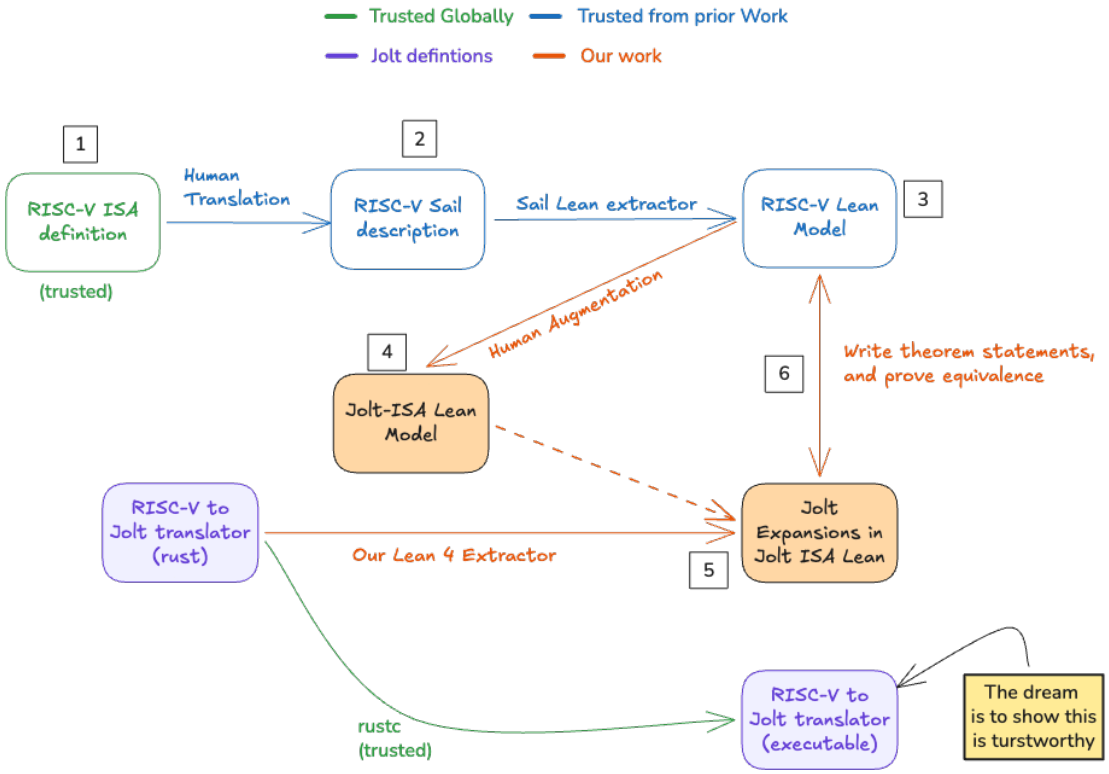

Figure 1: Items are green are considered trusted by the computer science community at large. These inlude programs the `rustc` compiler and the RISC-V ISA specification. Items in blue, are critical prior work that most recent efforts at formally verifying zk-VMs rely upon. These include artefacts like the RISC-V extraction from Sail to Lean. Items in purple describe the Jolt source code and executable, the thing we wish to deem trustworthy. Items in orange, describe Lean souce generated by this work, to establish that certain components of Jolt are trustworthy.

2 The Process of Verification

Formal verification of software is intended to create improved trust in code that one wishes to rely upon. Before we describe how we formally verified equivalence of the specific software artefacts under study, we will seek to describe what those software artefacts are, how they relate to the software artefacts that are intended for widespread use (and therefore require improved trust), and how our verification project generates trust in those artefacts. In particular, this process should highlight what should be thought of

as "proven", what we are considering as trusted due to ecology (e.g. the rust compiler), and what other artefacts are bearing the weight of assumed trust (and thus, may be beneficial areas for future work).

Figure 1 summarises the process of trust propagation. It outlines what we are verifying and how it relates to things we need to trust for our desired outcome – that the final executable that performs Jolt Bytecode expansions, faithfully emulate RISC-V instructions. Below we describe the process of trust propagation with item numbers acting as references back to the figure.

1. The starting point is the RISC-V ISA architectural specification defined by the chip manufacturers [19].
2. The RISC-V specification is written in human accessible natural language. From this specification, prior work [13] translate the RISC-V ISA into Sail [1], a language for defining the instruction-set architecture (ISA) semantics of processors. This Sail specification of the RISC-V chip has been adopted by RISC-V International.
3. Given the Sail specification of the RISC-V ISA, as mentioned earlier, researchers at Galois, Lindy Labs, University of Edinburgh, and Cambridge university have written a Sail compiler to auto-generate a Lean 4 model of the RISC-V CPU [6]. The focus of our work is to prove equivalence against **this** Lean4 definition of the RISC-V CPU. We remark that the above compiler generates Lean code that is non-executable, but perfectly suitable for proving theorems[3]. Thus, all our formal guarantees depend on the assumption that this auto-extracted Lean RISC-V is faithful to the actual RISC-V specification.
4. As we have described in **Section 1**, the Jolt ISA is a rather minimal extension of the RISC-V ISA, and therefore we do small amount of *manual work* to generate a Lean 4 model of the Jolt ISA by extending this Lean 4 model of the RISC-V ISA (see **Section 3** for details on how this has been done). This process gives us a non-executable Jolt CPU definition in Lean. It should be noted that any misspecifications during the process of defining the Jolt ISA model, that results in deviation from functionality described the Jolt Rust code base would render final theorems useless.
5. The next step is to describe expansions of RISC-V instructions that are not natively supported by the Jolt ISA.The exact expansions are defined in the Jolt rust codebase, which when compiled along with the rest of Jolt using `rustc`, results in machine-executable binary code that performs the bytecode expansion at speed. *This artefact is what want to prove to be trustworthy*. Instead, we write the equivalent of a procedural macro, to translate the high-level rust code that models bytecode expansions to Lean[4]. Although, this step now gives us bytecode expansions of RISC-V instructions in Lean, defined using the Jolt ISA we defined in Step 4, we point out that it is contingent of us trusting the translation from rust to lean.
6. The formal verification we perform, then, validates that (with a few exceptions) for any given RISC-V instruction, its execution in the RISC-V Lean 4 model has the same effects on the model's CPU state as the execution of the sequence of Jolt ISA instructions that that RISC-V instruction expands to has within the Jolt ISA Lean 4 model.

2.1 How does this generate trust in the bytecode expander binary code?

Figure 1 makes it abundantly clear that there is some distance between the executable binary code we would like to trust and the Lean 4 models for which we are verifying equivalence. However, we have also outlined relations that allows us to deduce that showing that the executable is trustworthy rests on components we have verified, components that are small (e.g. our small Rust-to-Lean translation macros), and components that have a large amount of community buy-in (e.g. `rustc`).

[3] The code typechecks, and theorem proving in Lean is based on a variant of type theory.
[4] The code is entirely self contained in **https://github.com/a16z/jolt/blob/main/crates/jolt-lean-gen/src/main.rs**

In more detail, first, we remark that verifying emulators according to ISA specifications is a fairly standard business involving test vectors that are, largely, ISA-driven and thus can be centralised. For our purposes, the Sail Lean extractor is being treated as trusted, and we leave any validation of that to future work. This allows us to trust (subject to the caveats concomitant to the above discussion) that the RISC-V Lean 4 model faithfully represents the RISC-V ISA. The Jolt ISA Lean 4 model is a very lightweight adaptation of the RISC-V Lean 4 model (few hundred lines of Lean code), and thus is easily checked for issues. The expanded RISC-V instructions written in the Jole ISA in lean are automatically translated from our Rust codebase by a small (and therefore also readily checkable) amount of code. See **Section 5** for details. Finally, because `rustc` is largely regarded as trusted, it is reasonable to assume that accurately modelling rust semantics in Lean implies that we have accurately modelled the executables behaviour also.

Now if we were to prove (item 6) in Lean, that the Jolt expansions acutally implement RISC-V instructions, then the Rust code that was straightforwardly translated to generate that implementation as well as the Jolt ISA model that was straightforwardly augmented from the RISC-V ISA model are likely themselves trustworthy (otherwise the proofs would unlikely pass).

3 Modelling Stateful Computation In Lean

As discussed in **Section 1**, one of the motivating factors behind formalising Jolt in Lean is that we already have a RISC-V specification written in Lean. Given that we are going to demonstrate techniques for proving things about program execution in Lean, it will help to explain in some detail what modelling a RISC-V CPU in Lean looks like.

3.1 The Sail RISC-V Lean Model

As the Jolt ISA builds on top of a RISC-V ISA, our starting point will be to understand the RISC-V Lean model generated by the Sail compiler as described in **Section 2**. While the RISC-V Sail specification is quite full-featured, modelling elements such as TLB cache and multiple harts, we will limit our explanation to subset of types/structure elements relevant for our concerns: registers, memory, and CPU error states. As such, our description will faithfully represent that subset as reified in the Sail Lean model, but will not be exhaustive. We remark that every code snippet described in this subsection is part of the trusted transpilation. We did not write the Lean code below. We describe it in order to better understand how we extend it, when we do write Lean code for the Jolt ISA.

3.1.1 RISC-V Machine State in Lean

The RISC-V lean model represents its machine state as `SequentialState`, a structure parameterised by a dependent register-type function `RegisterType : Register → Type`. (Note that by using `SequentialState` we assume that instructions are executed in order, and we do not model concurrent instruction execution. For the purpose of proving equivalence to Jolt bytecode expansions, this distinction does not matter.)

```
structure SequentialState (RegisterType : Register → Type) where
  regs : Std.ExtDHashMap Register RegisterType
  mem : Std.ExtHashMap Nat (BitVec 8)
  sailOutput : Array String
  -- ... other elements such as current CPU states, etc.
```

Thus in this model, memory is modelled as byte-addressable (i.e., the values are `BitVec 8`), but the addresses—the keys of the hash map—are modelled as of type `Nat`.

Registers are modelled using Lean's built-in dependent hash map type, which accounts for the fact that different sorts of registers may have different value types (e.g. some registers might be 32-bit whereas others might be 64-bit). Thus the type signature of `Std.ExtDHashMap` receives the key type (`Register`), which will be an inductive type whose terms simply enumerate every RISC-V register, and not the value type, but a function which specifies, for each possible key type, what the type of the corresponding value will be:

```
inductive Register : Type where
  | PC
  | x1 | x2 | x3 | ... | x31   -- general-purpose registers
  -- ... CSRs, vector registers, debug state, ~170 variants total ...
```

```
abbrev RegisterType : Register → Type
  | .PC => (BitVec 64)
  | .x1 => (BitVec 64)   -- all x1–x31 map to BitVec 64
  | .x2 => (BitVec 64)
  -- ... other register types such as hart state, TLB entry, etc.
```

After instantiating `SequentialState` with the `RegisterType` function defined above, we write the resulting RISC-V machine state as `SailState` .

```
abbrev SailState := SequentialState RegisterType
```

3.1.2 Program Execution in Lean

Modelling CPU instruction execution is an inherently stateful process. As Lean is a pure functional language, the Sail model encodes the stateful nature of CPU execution using Lean's built-in error-state monad, `EStateM` .

The relevant parameters for the EStateM monad are are a state type and an error type, (and an answer type, which we will not make use of). The state we will use is the `SailState` introduced above, and the error type will be an inductive type that enumerates the various error states a CPU can enter (e.g. `Exit` for when the code requests a halt, or `OutOfMemoryRange` for when code executes outside the valid memory range, or a further `User` type for specific processor implementations.[5]):

```
inductive Error (ue : Type) where
  | Exit
  | OutOfMemoryRange (n : Nat)
  | User (e : ue)
  -- .. other error conditions
```

The idea is that each instruction execution will either result in an updated CPU state (if the instruction retires normally, for example), or an error state which should halt execution (such as if the program itself requested a halt).

Specifically, then, a each RISC-V instruction will be of type `EStateM Error SailState ExecutionResult` , i.e. a function receiving a `SailState` (representing the current CPU state) and returning either a `EStateM.Result.ok SailState` after successful execution (wrapping the new CPU state after the instruction's completion) or a `EStateM.Result.error Error SailState` in the event of failure (wrapping the error describing

[5]The `Error` type definition comes from the generic Lean library that is used across all Lean processor models generated compiled by Sail, and thus leaves the processor-specific exception types to be specified on an ISA-by-ISA basis.

the specific failure condition, as well as an updated state, since an instruction may modify registers before raising an exception.)

Successful instruction execution returns an `ExecutionResult`, which enumerates the possible outcomes of a single instruction execution. Most instructions retire normally with `Retire_Success`. The remaining variants cover traps, memory exceptions, and privilege changes, most of which we do not have to deal with when arguing equivalence.

```
inductive ExecutionResult where
  | Retire_Success (_ : Unit)
  | ExecuteAs (_ : instruction)
  | Enter_Wait (_ : WaitReason)
  | Illegal_Instruction (_ : Unit)
  | Virtual_Instruction (_ : Unit)
  | Trap (_ : (Privilege × ctl_result × xlenbits))
  | Memory_Exception (_ : (virtaddr × ExceptionType))
  | Ext_CSR_Check_Failure (_ : Unit)
  | Ext_ControlAddr_Check_Failure (_ : ext_control_addr_error)
  | Ext_DataAddr_Check_Failure (_ : ext_data_addr_error)
  | Ext_XRET_Priv_Failure (_ : Unit)
```

Together, these instantiate `SailM` — the monad in which all transpiled RISC-V instructions are written:

```
abbrev SailM := EStateM (Error exception) SailState
```

More simply, the `SailM` monad is an arrow type, whose terms represent the different ways we can step the RISC-V CPU. Shown below is an example of how the trusted translation defines how the `ADDIW` instruction updates the RISC-V state.

```
def execute_ADDIW (imm : (BitVec 12)) (rs1 : regidx) (rd : regidx) : SailM ExecutionResult := do
  let result ← do (pure ((← (rX_bits rs1)) + (sign_extend (m := 64) imm)))
  (wX_bits rd (sign_extend (m := 64) (Sail.BitVec.extractLsb result 31 0)))
  (pure RETIRE_SUCCESS)
```

3.2 Modelling Jolt State

Having explained the Lean model of a RISC-V CPU, we now proceed to explain our extension of this to a Lean model of a Jolt ISA CPU. All code[6] snippets described in this section are *new*. They were *not* part of the trusted Sail transpilation.

3.2.1 The Jolt ISA

The Jolt ISA is a very simple modification of the RISC-V ISA. The CPU state of a Jolt CPU includes a RISC-V CPU state, along with an additional 128 64-bit general-purpose registers referred to as "virtual" registers.

The instructions within the Jolt ISA consist of the subset of RISC-V instructions that are decomposable, along with a further set of "Virtual" instructions that are also decomposable, and whose purpose is to render the non-decomposable RISC-V instructions implementable as relatively straightforward compositions of Jolt ISA instructions. In addition, all Jolt ISA instructions can use as operands both virtual and RISC-V general-purpose registers.

[6]The full source code can be found at https://github.com/LayerZero-Labs/jolt-qed/

3.2.2 The Jolt CPU model in Lean

To implement this in Lean, we define the Jolt CPU State as a structure that contains the `SailState` from the RISC-V CPU model above:

```
structure SailJoltState where
  sail  : SailState
  vregs : BitVec 7 → BitVec 64 := fun _ => 0
```

The `sail` field holds the full RISC-V architectural state. The `vregs` field maps each 7-bit index to a 64-bit word with the default value initialised to 0[7]. Unlike `sail.regs`, which is a dependent hashmap, `vregs` is a plain function and every virtual register has type `BitVec 64`. This implies that we do not need to assume that the register is readable. Unlike a hash-map which might have missing keys, a function can be evaluated successfully for every term in its domain (see **Section 4** for more details). Similar to `SailM`, Jolt computations live in `JoltMonad`, which is the `EStateM` monad with `SailJoltState` as the type that describes machine state.

```
abbrev JoltMonad (α : Type) := EStateM (Error exception) SailJoltState α
```

To define the Jolt instructions in Lean, we use an inductive type `JoltISA.Instr`. Its constructors fall into two groups:

1. *RISC-V-like instructions*, which correspond directly to RISC-V opcodes.
2. *Virtual instructions* (prefixed `Virtual`), which have no RISC-V counterpart and exist only in the Jolt model.

```
inductive Instr where
  -- RISC-V-like instructions
  | ADDI  (dst : Dst) (src : Src) (imm : BitVec 12)
  | ANDI  (dst : Dst) (src : Src) (imm : BitVec 12)
  | ADD   (dst : Dst) (lhs rhs : Src)
  | SUB   (dst : Dst) (lhs rhs : Src)
  | BEQ   (lhs rhs : Src) (imm : BitVec 13)
  | LD    (vd base : VReg) (imm : BitVec 12)
  | SD    (base value : VReg) (imm : BitVec 12)
  -- ...
  -- Virtual instructions
  | VirtualSignExtendWord    (dst : Dst) (src : Src)
  | VirtualAssertLoadAlignment (base : regidx) (imm : BitVec 12) (mask : BitVec 64)
  | VirtualAdvice            (vd : VReg) (value : BitVec 64)
  | VirtualAssertEQ          (lhs rhs : VReg)
  | VirtualChangeDivisor     (dst : VReg) (dividend divisor : regidx)
  -- ... (~40 Virtual constructors total)
```

As the Jolt CPU can read an write from both RISC-V general purpose registers, and virtual registers, we define a generic inductive type called `Src` and `Dst` to model source and destination registers.

```
inductive Src where
| vreg : VReg → Src
| xreg : regidx → Src
```

[7] In the Rust codebase, they use virtual registers 0-31 to model RISC-V general-purpose registers. However, in Lean, the general-purpose registers are already defined to be inductive types that key into the hash map. Thus, to stay faithful to the Sail model and be able to prove theorems, we model RISC-V registers using the same inductive types and never use virtual registers 0-31 to prove theorems.

```
inductive Dst where
| vreg : VReg → Dst
| xreg : regidx → Dst

def readSrc : Src → JoltMonad (BitVec 64)
  | .vreg vr => readVReg vr          -- direct function application, always .ok
  | .xreg rs => liftSail (rX_bits rs) -- hashmap lookup, requires read succeeds assumption

def writeDst : Dst → BitVec 64 → JoltMonad Unit
  | .vreg vr, v => writeVReg vr v    -- always .ok
  | .xreg rd, v => liftSail (wX_bits rd v) -- always .ok
```

We also write helper methods `readSrc` and `writeDst` to read and write to virtual or general purpose RISC-V registers. The `liftSail` method takes in some RISC-V computation as input, and runs the computation inside the Jolt monad (this is always possible as the JoltState contains RISC-V state inside it).

Above we describe the instruction signatures. We still have to define how these instructions affect machine state i.e the state transition function for each of these instructions. Instruction semantics are given by `execInstr`, which maps each `Instr` to a `JoltMonad ExecutionResult`.

```
-- jolt-qed/JoltBytecode/JoltISA/Semantics.lean
def execInstr : Instr → JoltMonad ExecutionResult
  | .ADDI dst src imm => do
      let x ← readSrc src
      writeDst dst (x + sign_extend (m := 64) imm)
      pure RETIRE_SUCCESS
  | .ADD dst lhs rhs => do
      let x ← readSrc lhs
      let y ← readSrc rhs
      writeDst dst (x + y)
      pure RETIRE_SUCCESS
  -- ... one case per constructor
```

The logic for the instruction semantics is specified in the `tracer` crate of Jolt in the `tracer/src/instruction/<instr-name>.rs`. Each file contains an `exec` method that describes how the instruction updates the Jolt CPU.

```
impl ADDI {
    fn exec(&self, cpu: &mut Cpu, _: &mut <ADDI as RISCVInstruction>::RAMAccess) {
        cpu.write_register(
            self.operands.rd as usize,
            cpu.sign_extend(
                cpu.x[self.operands.rs1 as usize]
                    .wrapping_add(normalize_imm(self.operands.imm, &cpu.xlen)),
            ),
        );
    }
}
```

> **Remark:** The Lean execution semantics for Jolt instructions are hand-transcribed. That is, we have no automated procedure to go from the rust implementation above to the following Lean implementation. In

> **Section 2**, we mentioned that to describe the Jolt ISA in Lean, we needed a little bit of human-augmentation. As CPU semantics are given by short, simple blocks of code, we do not envisage this translation to introduce significant risk to the final guarantees. Additionally, the proof of equivalence against the trusted Sail description of the same instructions, gives us confidence that at least the native RISC-V instructions have been implemented correctly. Still, it is imperative we mention this clearly off the bat, to ensure we are fully transparent about what is the final artefact we are verifying.

Now we describe the bytecode expansion process. Each RISC-V instruction that is not a constructior of `Jolt.Instr` type must have an expansion with type `Program` given below.

```
inductive Program where
  | done  (result : ExecutionResult)
  | instr (instr : Instr) (next : Program)
```

A program is recursively defined to be either `done` or a Jolt instruction followed by another Jolt program. As a concrete example, the RISC-V `SLL rd, rs1, rs2` instruction (shift-left-logical) has no direct `Instr` constructor. Its Jolt expansion is the two-instruction `sllProgram`:

```
def sllProgram (rs2 rs1 rd : regidx) : Program :=
  .instr (.VirtualPow2W (.vreg 0) (.xreg rs2)) <|  -- v0 := 2 ^ rs2[4:0]
  .instr (.MUL (.xreg rd) (.xreg rs1) (.vreg 0)) <| -- rd := rs1 * v0
  .done RETIRE_SUCCESS
```

As a point of comparison, shown below is the rust expansion for the same instruction[8].

```
asm.expand_i(
    SourceInstructionKind::VirtualPow2,
    v_pow2.operand(),
    reg(rs2(instruction)?),
    0,
);
asm.emit_r(
    Kind::MUL,
    reg(rd(instruction)?),
    reg(rs1(instruction)?),
    v_pow2.operand(),
);
```

> **Remark:** Unlike instruction semantics, these expansions are *not* hand transcribed. They are instead auto-extracted from the Jolt source code. See **Section 5** for details on how we extract rust to lean.

A `Program` is just a concatentation of Jolt instructions. `execProgram` recurisvely describes how this instruction sequence changes the Jolt CPU state. For each instruction, we invoke the `execInstr` method to run the instruction. If an instruction retires with `RETIRE_SUCCESS` the interpreter continues with the remaining instructions. Otherwise result is returned immediately and the remaining instructions are skipped.

[8] https://github.com/abiswas3/jolt/blob/3bde0cbd1a5ec5d7b9da72396d5076a0dd72b37f/crates/jolt-program/src/expand/shifts/sll.rs#L14-L25

```
def execProgram : Program → JoltMonad ExecutionResult
  | .done result    => pure result
  | .instr instr rest => do
      match ← execInstr instr with
      | .Retire_Success () => execProgram rest
      | result             => pure result  -- short-circuit on any non-retire outcome
```

3.2.3 Relating the Jolt and RISC-V CPU models

To state equivalence between a Jolt computation and a RISC-V computation, we need to project a Jolt Monad result down to a `SailM` result. That is, after running a sequence of state transitions of the Jolt CPU, we should be able to ask what the RISC-V CPU looks like now. The method that allows us to do this is the `systemProjectResult` shown below.

```
def systemProjectResult
    (r : EStateM.Result (Error exception) SailJoltState α) :
    EStateM.Result (Error exception) SailState α :=
  match r with
  | .ok a js' => .ok a (systemProject js')
  | .error e js' => .error e (systemProject js')
```

In simple English, it says if you give me the result of a Jolt computation (successful or not), then we return a result of a RISC-V computation which we know has type `EStateM.Result (Error exception) SailState α` (the output type). The body of the method says that this is achieved by taking the Jolt computation's updated Jolt state and projecting it down to a RISC-V state with the `systemProject` method described below.

```
def systemProject (js : SailJoltState) : SailState :=
  { js.sail with
    regs :=
      ((((((js.sail.regs
        |>.insert Register.mtvec (js.vregs JoltISA.trapHandlerVReg))
        |>.insert Register.mscratch (js.vregs JoltISA.mscratchVReg))
        |>.insert Register.mepc (js.vregs JoltISA.mepcVReg))
        |>.insert Register.mcause (js.vregs JoltISA.mcauseVReg))
        |>.insert Register.mtval (js.vregs JoltISA.mtvalVReg))
        |>.insert Register.mstatus (js.vregs JoltISA.mstatusVReg)) }
```

At first glance, the above code block appears unnecessarily complicated. Intuitively, the projection should just return the RISC-V part of the Jolt state, i.e., we should just output `js.sail`. This is exactly what we do, but before we output the RISC-V state, we write the values in 6 virtual registers to 6 specific locations in the register file hashmap. As mentioned earlier, in Jolt the control and status registers are modelled with virtual register indices 32-39. The trusted RISC-V state register file hashmap has keys that model these CSRs explicitly. Thus, if a Jolt program ever writes to these CSRs via virtual register writes, this write should be reflected in the RISC-V state. Without this prelude, a Jolt program can write to virtual registers modelling CSRs while the RISC-V program does not, and a theorem can still pass. So far, we have defined Jolt state and how to project down from Jolt to RISC-V state.

Now that we have given the definition of a RISC-V CPU and a Jolt CPU in Lean, as well as a way to take a Jolt CPU state and determine the corresponding RISC-V CPU state, we are now ready to describe the main set of theorems that we prove here. This will be the work of the next section.

4 Theorem Statements And Assumptions

For every RISC-V instruction that is *not* native to the Jolt ISA, we give a machine proof of the following statement - given inputs to the instruction and an assumption bundle, starting from the same machine state, the expansion and the original instruction produce the same final RISC-V CPU state. The details are best illustrated by the theorem for the `SLL` expansion in **Section 1**.

```
def sllProgramEqSailStatement
    -- Arguments for the SLL instruction
    (rs2 : regidx)
    (rs1 : regidx)
    (rd : regidx)
    -- Initial Jolt State
    (js : SailJoltState)
    -- Assumption Bundle
    (_h : BinarySourceReadWithLinkedCSRs rs2 rs1 js) : Prop :=
    -- Theorem statement
  System.systemProjectResult
      ((JoltISA.execProgram (JoltISA.sllProgram rs2 rs1 rd)).run js) =
    (execute_RTYPE rs2 rs1 rd rop.SLL).run js.sail
```

The input arguments are direct: `js` is the Jolt state before executing the Jolt program, and `js.sail` is the corresponding RISC-V state before executing `SLL`. The theorem statement is:

```
System.systemProjectResult
      ((JoltISA.execProgram (JoltISA.sllProgram rs2 rs1 rd)).run js) =
    (execute_RTYPE rs2 rs1 rd rop.SLL).run js.sail
```

We now break down the theorem statement.

`(execute_RTYPE rs2 rs1 rd rop.SLL).run` is the RISC-V computation generated by trusted transpilation and run on the RISC-V component of the Jolt state. This monadic computation returns a value `v` and an updated RISC-V state `s`. The expression `((JoltISA.execProgram (JoltISA.sllProgram rs2 rs1 rd)).run js)` executes the Jolt program on the complete initial state and returns a value `v'` and an updated Jolt state. `System.systemProjectResult` projects from that final Jolt state to a value `v'` and RISC-V state `s'`. Thus, the proof must establish `v = v'` and `s = s'`.

> For every RISC-V instruction *not* natively in the Jolt ISA, we have an equivalent theorem statement. For instructions that are native to both RISC-V and Jolt ISA, we also prove an equivalence theorem that shows that our semantic definition *hand translated* from the Rust implementation matches the RISC-V state transition specification.

The remaining question is the role of `_h: BinarySourceReadWithLinkedCSRs`. It is an **assumption bundle**, a conjunction of the assumptions needed to close the particular equivalence theorem. We define and justify the core assumptions in **Section 4.2**; each is supported by the Jolt implementation or by the trusted RISC-V model.

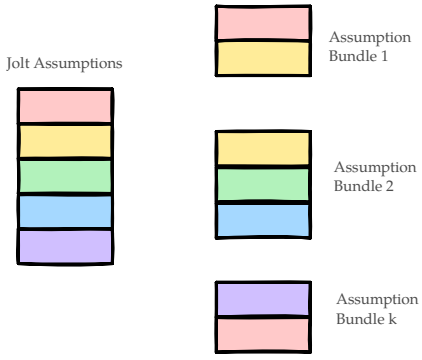

Figure 2: Jolt defines a fixed set of core assumptions in `Assumptions.lean`. Each assumption bundle is a subset of these assumptions. This allows for an ergonomic way to list every assumption made in Jolt. Once each assumption in `assumption.lean` is audited, we can be safe that a bundle never uses any assumption outside this list.

Before describing the assumptions and justifying their presence, we describe the two changes made to the trusted transpiled code.

4.1 Changes to Trusted Transpilation

The RISC-V component is generated from the official Sail specification [13] using the LeanRV64D transpilation [6]. We treat this generated model as trusted, except for the two changes below, which align its behavior with the Jolt CPU.

```
-- WARNING: CHANGE IN TRANSPILED CODE
def plat_enable_misaligned_access : Bool := false
```

We say that the RISC-V CPU should not support misaligned memory addressing. The trusted transpilation has this flag set to true. Jolt does not support misaligned addressing, and it would be impossible to prove equivalence with the Sail spec without this change. The Ethereum zkVM Standards[9] recommend that all zk-VMs use `RV64IM+Zicclsm` as the target baseline, where `Zicclsm` is included to support misaligned loads and stores. We highlight this Jolt drawback off the bat.

```
-- WARNING: (change to sail, Jolt does not support Zicfilp extension).
| Ext_Zicfilp => false
-- WARNING: (change to sail, Jolt does not support Zicfiss extension).
| Ext_Zicfiss => false
```

Jolt does not implement the Zicfiss and Zicfilp extensions, whereas the trusted transpilation enables them by default. We therefore disable both extensions in the generated model. These changes are the *only* changes we make to the trusted transpiled Lean code.

4.2 Jolt Assumptions

For the remainder of this section, we list the assumptions used in the Jolt equivalence proofs and justify each one with evidence from the Jolt implementation. We re-iterate that these are the only assumptions used to prove theorems are the ones listed below.

[9] https://zkevm.ethereum.foundation/blog/zkevm-standards-v0-release

4.2.1 Privilege Is Always Machine

Based on the specification, Jolt runs its CPU in machine mode throughout execution[10]. The following hypothesis says precisely that the value in the `curr_privilege` register is set to `Privilege.Machine`.

```
structure CurPrivilegeMachine (s : SailState) : Prop where
  value : s.regs.get? Register.cur_privilege =
    some (Privilege.Machine : RegisterType Register.cur_privilege)
```

4.2.2 MPP Bit Set to Machine

Jolt uses virtual register 39 to model the `mstatus` control status register. The Jolt `mret` implementation[11] establishes that execution remains in machine mode and that the MPP field of `mstatus` is set to `Machine`.

```
structure MstatusMppMachine (js : SailJoltState) : Prop where
  value_eq :
    _get_Mstatus_MPP (js.vregs JoltISA.mstatusVReg) =
      privLevel_to_bits Privilege.Machine
```

4.2.3 MISA User Mode Enabled

Jolt initializes `misa` to `0x8000000008014312f` in its CPU emulator[12]. The `_get_Misa_U` accessor reads bit 20, which is set to one based on the above value. According to the RISC-V /specification [13], this indicates that user mode is implemented.

```
-- The Misa register is populated in the hash-map and bit 20 is set to 1
structure MisaUserEnabled (s : SailState) : Prop where
  exists_value : ∃ misa : BitVec 64,
    s.regs.get? Register.misa = some (misa : RegisterType Register.misa) ∧
    _get_Misa_U misa = 1#1
```

4.2.4 MPRV Bit

The `MPRV` bit of `mstatus` is zero in Jolt's MMU[13]. As a consequence of this assumption, one can formally derive the fact that virtual addresses are the same as physical addresses in Jolt i.e. address translation is the identity function[14]

```
-- Mstatus is ppulated and Mstatus MPRV bit is 0
structure MstatusMprvZero (s : SailState) : Prop where
  value : ∃ mval : RegisterType Register.mstatus,
    s.regs.get? Register.mstatus = some mval ∧
    _get_Mstatus_MPRV mval = 0#1
```

4.2.5 Zicfilp Extension

Jolt does not support the Zicfilp extension.

```
structure ZicfilpDisabled (s : SailState) : Prop where
  value : currentlyEnabled extension.Ext_Zicfilp s = .ok false s
```

[10] https://github.com/abiswas3/jolt/blob/main/tracer/src/emulator/cpu.rs#L329-L353
[11] https://github.com/abiswas3/jolt/blob/main/tracer/src/instruction/mret.rs#L7-L18
[12] https://github.com/abiswas3/jolt/blob/main/tracer/src/emulator/cpu.rs:399
[13] https://github.com/abiswas3/jolt/blob/main/tracer/src/emulator/mmu.rs#L870-L898
[14] See `JoltBytecode/InstructionEquivalence/ProofSupport/BundleLemmas.lean`.

The next set of assumptions are about writing to control status registers. The RISC-V specification says all writes must be legalised before writing to a control status register. In Jolt the legalisation function is *always* the identity function. In RISC-V, the legalise function is identity under a strict set of constraints. Below we assume that those conditions are satisfied, and justify why having legalisation set to identity is acceptable, and how this does not violate the RISC-V specifications.

4.2.6 `mtvec` Writes Use Direct Mode

The RISC-V specification requires values to be legalised before they are written to control and status registers. If the two least significant bits of the contents of `rs1` are `00` or `01` then we write the contents of `rs1` to `mtvec` unchanged. Otherwise the value must be legalised. In Jolt, writing to the control status registers is done by ZeroOS. ZeroOS *always* writes `_trap_handler` to `mtvec` using `csrw mtvec, t0`, and defines[15] `_trap_handler` under `.align 2`. As a result the address is four-byte aligned. Based on the boot code[16] we can assume the write can pass through unchanged.

```
structure MtvecWriteDirectMode (value : BitVec 64) : Prop where
  mode_eq : _get_Mtvec_Mode value = 0b00#2
```

4.2.7 Program Counter Alignment

Jolt supports compressed instructions (`+c`, or RV64IMAC), so the least significant bit of the program counter is set to 0. . Accordingly, the relevant assumption is the lowest bit of the `mepc` register is set to 0. Under this assumption, PC legalization is a no-op.

```
structure MepcReadAligned (value : BitVec 64) (s : SailState) : Prop where
  value_eq : align_pc value s = .ok value s
```

The write-side counterpart concerns the new `rs1` value written by CSRRW, rather than only the previously stored `mepc`.

```
structure MepcWriteLegalized (value : BitVec 64) : Prop where
  value_eq : legalize_xepc value = value
```

4.2.8 `mstatus` Writes Are Already Legalized

```
structure MstatusWriteLegalized
    (old value : BitVec 64) (s : SailState) : Prop where
  value_eq : legalize_mstatus old value s = .ok value s
```

Jolt[17] does not perform legalization before writing `mstatus`. In the Jolt CSR expansions, `CSRRW` writes `rs1` directly to the CSR virtual register, while `CSRRS` writes[18] the bitwise OR of the old CSR value and `rs1`.

4.2.9 Virtual-Register Correspondence

Jolt models persistent CSRs using virtual registers. We could write directly to the Sail registers in the Jolt semantics, but doing so would require us to change the Jolt expansions. Instead, the semantics writes the virtual registers and this assumption states that the virtual registers are the same as the control status

[15] https://github.com/LayerZero-Labs/ZeroOS/blob/main/platforms/spike-platform/src/boot.rs#L13-L19
[16] https://github.com/LayerZero-Labs/ZeroOS/blob/main/crates/zeroos-arch-riscv/src/trap.rs#L308-L316
[17] https://github.com/abiswas3/jolt/blob/main/crates/jolt-program/src/expand/control_flow/csrrw.rs#L17-L63
[18] https://github.com/abiswas3/jolt/blob/main/crates/jolt-program/src/expand/control_flow/csrrs.rs#L26-L74

registers in the hash map[19]¡.

```
/-- Sail `mstatus` agrees with Jolt's persistent `mstatus` virtual register. -/
structure MstatusVRegMatchesSail (js : SailJoltState) : Prop where
  value_eq :
    js.sail.regs.get? Register.mstatus =
      some (js.vregs JoltISA.mstatusVReg)

/-- Sail `mtvec` agrees with Jolt's persistent trap-handler virtual register. -/
structure MtvecVRegMatchesSail (js : SailJoltState) : Prop where
  value_eq :
    js.sail.regs.get? Register.mtvec =
      some (js.vregs JoltISA.trapHandlerVReg)

-- 4 more similar assumptions about Mscratch, Mepc, Mcause, Mtval
```

4.2.10 Memory Assumptions

```
structure DwordPresent (addr : BitVec 64) (s : SailState) : Prop where
  bytes :
    ∃ bytes : Fin 8 → BitVec 8,
      ∀ k : Fin 8, s.mem.get? (addr.toNat + k.val) = some (bytes k)
```

Lean models memory with a hash map from natural numbers to eight-bit bit strings. To prove facts about the contents in the hash map, Lean requires that we give a proof that the key is present in the hash maps In an actual physical CPU would never have to make such an assumption, so this assumption simply says that the 64 bits at given physical address `addr` is readable.

```
abbrev LoadPmpOk (addr : BitVec 64) (width : Nat) (s : SailState) : Prop :=
  phys_access_check (Load Data) Privilege.Machine
    (physaddr.Physaddr addr) width false s = .ok none s

structure LoadPmpOkWindow
    (base : BitVec 64) (width : Nat) (s : SailState) : Prop where
  ok :
    ∀ offset accessWidth : Nat, offset + accessWidth ≤ width →
      LoadPmpOk (base + BitVec.ofNat 64 offset) accessWidth s

abbrev StorePmpOk (addr : BitVec 64) (width : Nat) (s : SailState) : Prop :=
  phys_access_check (Store Data) Privilege.Machine
    (physaddr.Physaddr addr) width false s = .ok none s

structure StorePmpOkWindow
    (base : BitVec 64) (width : Nat) (s : SailState) : Prop where
  ok :
    ∀ offset accessWidth : Nat, offset + accessWidth ≤ width →
      StorePmpOk (base + BitVec.ofNat 64 offset) accessWidth s

abbrev AtomicPmpOk
    (op : amoop) (addr : BitVec 64) (width : Nat) (s : SailState) : Prop :=
  phys_access_check (Atomic (op, Data, Data)) Privilege.Machine
    (physaddr.Physaddr addr) width true s = .ok none s
```

[19] https://github.com/abiswas3/jolt/tree/main/crates/jolt-program/src/expand/allocator.rs

```
structure AtomicPmpOkWindow
    (op : amoop) (base : BitVec 64) (width : Nat) (s : SailState) : Prop where
  ok :
    ∀ offset accessWidth : Nat, offset + accessWidth ≤ width →
      AtomicPmpOk op (base + BitVec.ofNat 64 offset) accessWidth s
```

Jolt's Rust MMU explicitly does not implement memory protection[20]. These assumptions tell the RISC-V Cpu that memory protection has been turned off. The Sail specification also distinguishes reads, writes and atomic read/writes separately, and hence we need three separate assumptions. The window forms are needed because Jolt always loads double word at the given address, so we have to say that all eight addresses are not protected.

The generated RISC-V model also distinguishes ordinary RAM from memory-mapped I/O. Jolt does not support device MMIO such as UART, CLINT, PLIC, or disks; accesses to such regions are unsupported by the tracer. We therefore require the relevant memory window to avoid readable or writable MMIO.

```
abbrev NotReadableMmio (addr : BitVec 64) (width : Nat) (s : SailState) : Prop :=
  within_mmio_readable (physaddr.Physaddr addr) width s = .ok false s

structure NotReadableMmioWindow
    (base : BitVec 64) (width : Nat) (s : SailState) : Prop where
  ok :
    ∀ offset accessWidth : Nat, offset + accessWidth ≤ width →
      NotReadableMmio (base + BitVec.ofNat 64 offset) accessWidth s

abbrev NotWritableMmio (addr : BitVec 64) (width : Nat) (s : SailState) : Prop :=
  within_mmio_writable (physaddr.Physaddr addr) width s = .ok false s

structure NotWritableMmioWindow
    (base : BitVec 64) (width : Nat) (s : SailState) : Prop where
  ok :
    ∀ offset accessWidth : Nat, offset + accessWidth ≤ width →
      NotWritableMmio (base + BitVec.ofNat 64 offset) accessWidth s
```

4.2.11 Register Assumptions

```
structure XRegReadable (r : regidx) (s : SailState) : Prop where
  exists_value : ∃ value : BitVec 64, rX_bits r s = .ok value s

structure SailRegReadable (r : Register) (s : SailState) : Prop where
  exists_value : ∃ value : RegisterType r, s.regs.get? r = some value

/-- Assumptions for an instruction that reads one architectural source register. -/
structure UnarySourceReadAssumptions (rs1 : regidx) (s : SailState) where
  rs1_val : BitVec 64
  rs1_read : rX_bits rs1 s = .ok rs1_val s

/-- Assumptions for an instruction that reads two architectural source registers. -/
structure BinarySourceReadAssumptions
    (rs2 rs1 : regidx) (s : SailState) where
  rs1_val : BitVec 64
```

[20] https://github.com/abiswas3/jolt/blob/main/tracer/src/emulator/mmu.rs#L17

```
  rs1_read : rX_bits rs1 s = .ok rs1_val s
  rs2_val : BitVec 64
  rs2_read : rX_bits rs2 s = .ok rs2_val s
```

Much like memory, these structures state that the RISC-V registers can be read successfully. The `UnarySourceReadAssumptions` are for used by instructions that read from 1 register only, whereas the `BinarySourceReadAssumptions` say both source registers are readable.

5 Auto-extraction

Figure **Figure 3** summarises the extraction process[21]. Our goal was to re-use as much as the rust jolt production code as possible, and write a thin easily checkable wrapper around it. By introducing as little code as possible, we do not broaden the trust surface any more than needed.

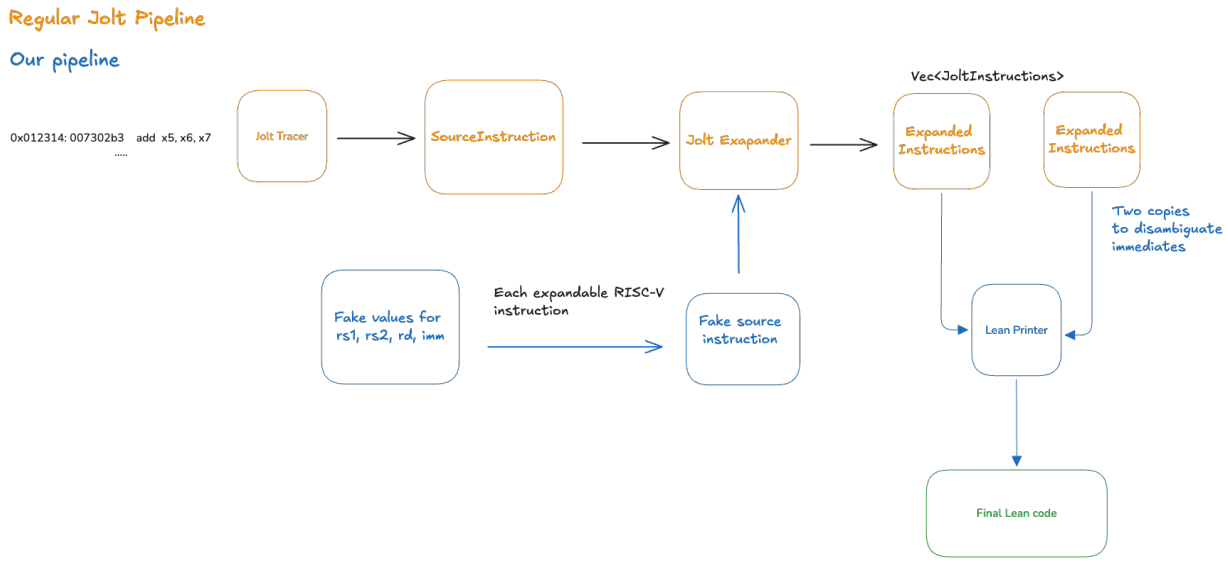

Figure 3: An outline of the how we re-use the Jolt expansion pipeline to obtain Lean translations.

We first describe the production Jolt expansion pipeline written in rust. The Jolt tracer takes as input the guest elf file with RISC-V instructions, and represents each RISC-V source instruction with an intermediate data structure called `SourceInstruction`, which is stored in memory of the host machine running Jolt. This data structure is passed into the Jolt expansion pipeline which returns a vector of Jolt instructions, another intermediate data structure stored in memory. This vector corresponds to the expanded sequence of Jolt instructions for the given RISC-V instruction, where each row of the vector has type `JoltInstructionRow` which represents one Jolt instruction. Now this intermediate data structure of Jolt instructions contains "nearly" all the information needed to translate into Lean.

```
pub struct JoltInstructionRow {
    pub instruction_kind: JoltInstructionKind,
    pub address: usize,
    pub operands: NormalizedOperands,
    pub virtual_sequence_remaining: Option<u16>,
```

[21] The entire translation code is contained in https://github.com/abiswas3/jolt/tree/rust-to-lean/crates/jolt-lean-gen

```
    pub is_first_in_sequence: bool,
    pub is_compressed: bool,
}

pub struct NormalizedOperands {
    pub rs1: Option<u8>,
    pub rs2: Option<u8>,
    pub rd: Option<u8>,
    pub imm: i128,
}
```

For example, consider the Lean description of the `LD` RISC-V instruction, which is natively supported by the Jolt-ISA.

```
.instr
    (.LD
      .normal
      (.vreg (BitVec.ofNat 7 41))
      (.vreg (BitVec.ofNat 7 41))
      (0 : BitVec 12)) <|
```

All the information needed to write the above string is already stored in `JoltInstructionRow`. The `kind` field stores the name of the instruction. The `normalized_operands` contain information about instruction inputs. We can also infer whether the registers are virtual or real. Thus, if someone were to hand our translator this intermediate representation, we could write a minimal printer that just translates this data into Lean code.

So the game now is to obtain this intermediate vector of Jolt instructions for every expandable RISC-V instruction. Towards this, we fix a dummy value for `rd, rs1, rs2` and, `imm` and create a fake RISC-V instruction in the `SourceInstruction` format for each expandable RISC-V instruction. We then invoke the production Jolt expander pipeline with this newly crafted data structure. The jolt expander treats this instruction as it would for any instruction in the guest program, and therefore expands it into a vector of Jolt Instructions. The translator just takes this intermediate data structure of Jolt instructions, and prints strings that pass the Lean typechecker. The only artefact that is brand new is this printer, which if it passes type checking is likely fine. Either ways, our translator could not have artificially written an incorrect expansion, as the printer does not control expansion logic. Thus, we argue that this extraction process hardly changes the trust surface in a meaningful way.

Observe that in excerpt above we said that the translator "nearly" has all information needed to perform translation. A small issue is that the intermediate data structure which is a vector of Jolt instructions stores all immediate operands as raw values. Thus, looking at the just the instruction, one cannot tell if value 37 is a hard-coded offset in the expansion algorithm or it was an intermediate value passed by the guest program. The two cases are different, and affect how the translator prints the expansion. To resolve this ambiguity, we create two dummy instructions with different immediate operands, say `imm:= 13` and `imm:=5`. For the immediate operands that are hard-coded by the expansions, they should remain the same in both runs. In contrast the user specified immediate values should change. We track this information across two runs, and this lets the translator know for which constants to use the string `imm`, and for which constants to translate raw integer values into strings.

We complete the above discussion with a concrete example. The `LB` instruction which expands to

```
VirtualAlignAddr   v41, x2, 37
LD                 v41, v41, 0
VirtualWindowMaskB v40, x2, 37
VirtualPextSigned  x1, v41, v40
```

Our transpiler generates the following Lean code.

```
    .instr (.VirtualAlignAddr (.vreg (BitVec.ofNat 7 41)) (.xreg rs1) imm) <|
    .instr (.LD .normal (.vreg (BitVec.ofNat 7 41)) (.vreg (BitVec.ofNat 7 41)) (0 : BitVec 12)) <|
    .instr (.VirtualWindowMaskB (.vreg (BitVec.ofNat 7 40)) (.xreg rs1) imm) <|
    .instr (.VirtualPextSigned (.xreg rd) (.vreg (BitVec.ofNat 7 41)) (.vreg (BitVec.ofNat 7 40)))
<|
    .done RETIRE_SUCCESS
```

Thus, in this way we are able to extract Lean descriptions of Jolt Programs that correspond to a single RISC-V instruction.

6 Results

The Jolt ISA implements 67 RISC-V instructions via expansion. We show equivalence for 60 out of 67 expanded instructions, and *all* of the native instructions[22]. In the process, we found one incorrect expansion that renders the proof system incomplete. Additionally, we found an expansion that deviates from the RISC-V specification, and thus is unprovable. Below we describe our findings in detail.

6.1 Incompleteness Issue

We identified a bug in the original implementation of Jolt's `DIVW` sequence. Given a dividend in `rs1` and a divisor in `rs2`, the way Jolt implements division is it asks the tracer to provide the correct values for quotient and remainder as advice. These values are treated as untrusted, and the Jolt sequence validates that these values are correct. Completeness implies that as long as long the tracer uses the honest advice, the program should never crash.

For `rs1 = i32::MIN` and `rs2 = 0`, the tracer correctly provides quotient `-1` and remainder `|i32::MIN| = 2^31` as untrusted advice (these values are correct as per the RISC-V specification). However, one of instructions in the sequence checks `SRAI(remainder, 31) == 0`, which implicitly requires the remainder to be less than 2^{31}; so the assertion fails and panics. This leads to an incompleteness issue. Although the tracer is acting in good faith, the expansion is rejected by the verifier. The expansion sequence has since been fixed[23], and the proofs pass.

6.2 Unprovable Instructions

RISC-V instructions `ECall, EBreak, LRD, LRW, SCD, SCW` are not fully described in the trusted Lean representations. Each of the above method descriptions contain black box axioms that ensure the trusted RISC-V Lean code compiles, but we cannot unfold these instructions to prove equivalence. More specifically, the trusted Lean code implementation of `execute LOADRES` contains a method named `load_reservation` in its body which does not have an implementation. Only the input and output types of the function is provided

[22] Remember that the Jolt ISA semantics for instruction are hand transcribed from Rust. Thus, showing that the RISC-V semantics are identical to the Jolt semantics for instructions common to both CPU's gives us an added layer of security that we did this translation correctly.

[23] https://github.com/a16z/jolt/pull/1473

for the type-checker to pass, and the project to compile. The actual body of the function is black-boxed. Similarly for conditional stores methods used in `execute STORECON` are blackboxed as well.

```
axiom load_reservation : Arch.pa → Nat → SailM Unit
axiom match_reservation : Arch.pa → Bool
axiom cancel_reservation : Unit → SailM Unit
axiom valid_reservation : Unit → Bool
```

For `ECALL` and `EBREAK` no body is provided at all, so we have nothing to prove our Jolt expansions against.

6.2.1 Control Status Registers (CSR's)

The jolt CPU only implements 6 control status registers, namely, `mscratch, mcause, mtval, mtvec, mepc` and `mstatus`. As we have mentioned in **Section 4**, RISC-V asks for writes to CSR's be first legalised. In Jolt, there is no legalisation process, we directly write values from the source register into the control status registers. This works fine when writing to `mscratch, mcause, mtval` where the legalisation function is the identity function always. However, for the remaining writes, the legalisation function is *only* identity if list of constraints are satisfied (for e.g. the last bit of the PC value being written in `mepc` is 0). We explicitly assume that these assumptions hold, to prove equivalence for the remaining three control status registers.

6.2.2 Mret

As mentioned in **Section 4**, Jolt writes to the `misa` register such that user mode is enabled. As soon as one enables user mode, the RISC-V specification says that when returning from a TRAP, the CPU must be eventually returned to user mode from machine mode. Although, Jolt implements user mode by writing 1 to the `misa`, in its implementation, it *never* switches modes. In fact, Jolt's privilege is always set to machine. This mismatch, prevents us from closing this theorem and proving equivalence. The table below illustrates the main proof bottlenecks.

Before `MRET`	
Sail	**Jolt**
`PC` = current `pc`	`pc` = current `pc`
`nextPC` = old `nextPC`	`nextPC` = old `nextPC`
`cur_privilege` = Machine	`cur_privilege` = Machine
`mepc` = return target	`mepc` = return target
`mstatus.MPP` = Machine	`mstatus.MPP` = Machine
`mstatus.MIE` = old `MIE`	`mstatus.MIE` = old `MIE`
`mstatus.MPIE` = old `MPIE`	`mstatus.MPIE` = old `MPIE`
`mstatus.MPRV` = old `MPRV`	`mstatus.MPRV` = old `MPRV`
After `MRET`	
Sail	**Jolt**
`nextPC` = `mepc`	`pc` / `nextPC` = `mepc`
`cur_privilege` = Machine from old `mstatus.MPP`	`cur_privilege` = Machine
`mepc` = unchanged	`mepc` = unchanged
`mstatus.MPP` = User unresolvable mismatch	`mstatus.MPP` = Machine unchanged
`mstatus.MIE` = old `MPIE` mismatch unless old `MIE` = old `MPIE`, but resolvable	`mstatus.MIE` = old `MIE` unchanged
`mstatus.MPIE` = 1 mismatch unless old `MPIE` = 1, but resolvable	`mstatus.MPIE` = old `MPIE` unchanged
`mstatus.MPRV` = old `MPRV` unchanged because privilege remains Machine	`mstatus.MPRV` = old `MPRV` unchanged

Table 1: Sail and Jolt machine state before and after `MRET`.

7 Future Work And Conclusion.

So far we have demonstrated, under certain assumptions that Jolt's bytecode expansion faithfully emulates a RISC-V CPU. We also argue, while the assumptions are justifiable, there still exists a gap between what we want to prove to be secure, and the artefact we provide guarantees for. The question of how to further reduce this trust surface is an interesting research direction. As mentioned in **Section 5**, software tools like Aeneas [7] are efforts to automatically translate Rust to Lean. For simple code snippets like those of CPU semantics, lowering down to a custom intermediate representation instead of using a procedural macro seems overkill, and potentially introduces a much larger trust surface. Aeneas does not yet provide an interface to say, translate *just this portion of code* into lean. It needs the whole Jolt binary.

Upstream components of Jolt are not simple four line imperative blocks of code without loops. We will have to find a more way to extract rust into Lean, as we verify the more complicated upstream components in Jolt. Looking ahead, bytecode expansions is the first foundational layer of the proving pipeline. Recall, that the tracer needs takes in a program written in the Jolt ISA, and generates the NP witness. To generate this witness, it must know how to decode jolt instructions, and that is exactly what our Jolt ISA tells it how to do. Equipped with the decoder, we are able write down the NP witness in Lean. This work is currently in progress in the *JoltConstraints* branch of the repository. Once we have this witness, we are able to define constraints that are then checked by Jolt's sumchecks [16]. In the next phase, we will show that the Jolt's NP relation is meaningful.

$$\text{Honest NP witness} \Leftrightarrow \text{All Jolt Constraints Satisfied} \tag{2}$$

Note the above guarantee has no randomness associated with it. It simply says the NP relation, that Jolt is claiming to be an argument of knowledge for is meaningful. Once we have this step, the sum-checks are just randomised tests for constraint satisfaction above. Initially we can show the sum-checks are indeed randomised tests of the above constraints, by assuming a perfect polynomial opening oracle. Finally, once all the work is done, we can isolate the task of proving the polynomial commitment scheme used in Jolt securely emulates the oracle. Thus, the proving equivalence to RISC-V and defining the Jolt ISA in Lean, are *fundamental* to formally verifying Jolt in Lean.

References

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Kathryn Gray, Robert Norton-Wright, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. Sail: A Language for ISA Semantics. Retrieved from https://github.com/rems-project/sail

[2] Arasu Arun, Srinath Setty, and Justin Thaler. 2024. Jolt: SNARKs for Virtual Machines via Lookups. In *Advances in Cryptology – EUROCRYPT 2024*, 2024.

[3] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium* (*USENIX Security 15*), August 2015. USENIX Association, Washington, D.C., 207–221. Retrieved from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[4] Ethereum Foundation. 2025. Allocation Update – Q3 2025. Retrieved from https://blog.ethereum.org/en/2025/12/02/allocation-q3-25

[5] Ethereum Foundation. 2025. Shipping an L1 zkEVM #2: The Security Foundations. Retrieved from https://blog.ethereum.org/2025/12/18/zkevm-security-foundations

[6] Tobias Grosser, Leo Stefanesco, James Parker, Lee Newcomb, Ben Selfridge, Ben Hamlin, Jakob von Raumer, and Ryan Lahfa. 2024. LeanRV64D: Lean 4 Model of RISC-V RV64GC Generated from the Sail Specification. Retrieved from https://github.com/opencompl/sail-riscv-lean

[7] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (August 2022). https://doi.org/10.1145/3547647

[8] Christoph Hochrainer, Valentin Wuestholz, and Maria Christakis. 2026. Arguzz: Testing zkVMs for Soundness and Completeness Bugs. In *USENIX Security Symposium*, 2026.

[9] LayerZero. 2026. Zero: Technical Positioning Paper. Retrieved from https://layerzero.network/blog/zero-technical-positioning-paper

[10] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020. 367–378. https://doi.org/10.1145/3372885.3373824

[11] Nethermind Security Formal Verification Team. 2025. Towards Formal Verification of the First RISC-V zkVM. Retrieved from https://www.nethermind.io/blog/towards-formal-verification-of-the-first-risc-v-zkvm

[12] Nethermind Security Formal Verification Team. 2025. Proof-of-Concept Formal Verification of SP1 zk Chips. Retrieved from https://www.nethermind.io/blog/proof-of-concept-formal-verification-of-sp1-zk-chips

[13] RISC-V International and REMS Project, University of Cambridge. The RISC-V Sail Model: Official Formal Specification of the RISC-V Architecture. Retrieved from https://github.com/riscv/sail-riscv

[14] Succinct Labs. 2025. Formal Verification of SP1 Hypercube. Retrieved from https://blog.succinct.xyz/nethermind-lean/

[15] Justin Thaler. 2024. Getting the Bugs out of SNARKs: The Road Ahead. Retrieved from https://a16zcrypto.com/posts/article/getting-bugs-out-of-snarks/

[16] Justin Thaler. 2025. Sum-check Is All You Need: An Opinionated Survey on Fast Provers in SNARK Design. Retrieved from https://eprint.iacr.org/2025/2041

[17] Devon Tuma and Nicholas Hopper. 2024. VCVio: A Formally Verified Forking Lemma and Fiat–Shamir Transform, via a Flexible and Expressive Oracle Representation. Retrieved from https://eprint.iacr.org/2024/1819

[18] Verified zkEVM. 2025. ArkLib: Formally Verified Arguments of Knowledge in Lean. Retrieved from https://github.com/Verified-zkEVM/ArkLib

[19] Andrew Waterman and Krste Asanović (Eds). 2019. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. Retrieved from https://docs.riscv.org/reference/isa/v20191213/_attachments/riscv-unprivileged.pdf

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. 283–294. https://doi.org/10.1145/1993316.1993532