Formal Verification of the OpenVM RISC-V zkVM

Technical Report

NETHERMIND

1st of February, 2026

1 Executive Summary

This report provides an overview of the engagement performed by Nethermind on formally verifying, in the Lean proof assistant, the OpenVM RISC-V zkVM implementation of the RV32IM instruction set against the official Lean RISC-V specification.

The work done in this engagement was based on the the official OpenVM repository (commit `645460f`, dating from December 8th, 2025), and the engagement has resulted in the following four deliverables:

1. a mechanism for extracting Plonky3 constraints from OpenVM chips, described throughout §3 and publicly available at:
   - `https://github.com/NethermindEth/openvm` (OpenVM fork at commit `645460f`); and
   - `https://github.com/NethermindEth/openvm-stark-backend` (stark-backend fork at commit `972f5db`);
2. a comprehensive verification infrastructure written in the Lean proof assistant (version 4.26.0), which is publicly available at `github.com/openvm-org/openvm-fv`; and
3. as part of this infrastructure, Lean proofs that the OpenVM zk implementations of the following 45 RISC-V RV32IM opcodes satisfy their Lean RISC-V specifications:
   - 27 ALU opcodes: ADD, ADDI, SUB, XOR, XORI, OR, ORI, AND, ANDI, SLT, SLTI, SLTU, SLTUI, SLL, SLLI, SRL, SRLI, SRA, SRAI, MUL, MULH, MULHU, MULHSU, DIV, DIVU, REM, and REMU;
   - 10 control-flow opcodes: AUIPC, BEQ, BNE, BLT, BGE, BLTU, BGEU, LUI, JAL, and JALR; and
   - 8 memory manipulation opcodes: LW, LH, LHU, LB, LBU, SW, SH, and SB.
4. as part of this infrastructure, a proof of execution and memory consistency of RV32IM-related OpenVM chips.

During this process, we have not found any bugs in the implementation of the RV32IM-related OpenVM chips.

Running the proofs. Assuming that an installation of Lean's lake package manager is present on the system, the infrastructure and all of the proofs can be built using:

```
lake update
lake exe cache get!
lake build
```

in the root folder of the repository, where the first two lines are superfluous but allow for a faster initial build.

1.1 Verification context, assumptions, and caveats

The verification was performed in the context of future participation of OpenVM in Ethereum L1 proving, for which the Ethereum Foundation (EF) is developing a proposal for the standardisation of the RISC-V target architecture for Ethereum with the goal of defining a minimal RISC-V instruction set for zkEVMs that would like to prove Ethereum blocks. This proposal, at the time of writing this report, makes the following recommendations that are relevant for this engagement:

EF1: that the supported set of instructions should be RV64IM: OpenVM currently supports RV32IM, which is the instruction set that we have verified in full as part of this engagement;

EF2: that the only supported privileged mode should be machine mode (M);

EF3: that no interaction with the environment using system calls should be required;

EF4: that the memory should be treated as flat, without advanced use of the memory management unit (MMU) and related concepts, such as paging.

On the other hand, the Lean RISC-V specification models almost all of the mechanisms, both mandatory and optional, described by the RISC-V standard and various extensions. In line with EF recommendations and the design choices made by OpenVM developers, we perform the following direct modifications to the Lean RISC-V specification:

S1: We disable the Zca RISC-V extension by setting its support to `false` here. The Zca extension enables the RISC-V C instruction set, which allows for higher data compression. This extension is not implemented by OpenVM and is not required for L1 proving.

S2: We disable the Zicfilp RISC-V extension by setting its support to `false` here. The Zicfilp extension enforces forward-edge control-flow integrity by introducing a landing-pad instruction, LPAD, which must be placed at the program locations that are valid targets of indirect jumps or calls. OpenVM does not support the LPAD instruction, and there is no indication that this support is required for L1 proving. In the proofs, this is relevant only for the JALR opcode.

S3: We disable the RISC-V core-local interruptor (CLINT), which is a memory-mapped peripheral that handles software and timer interrupts, by setting its size to zero here. This mechanism is not required for L1 proving, and disabling it is also in line with recommendation EF4 and the uniform way in which OpenVM models memory. In the proofs, this is relevant for all of the memory-manipulating opcodes.

S4: We disable the RISC-V Physical Memory Protection (PMP) mechanism here. The PMP is not modelled by OpenVM, and disabling it is in line with recommendation EF4. In the proofs, this is relevant for all of the memory-manipulating opcodes.

Further, we make the following assumptions on the RISC-V state in our Lean development (here and below):

A1: We 1) set the privileged mode to be machine mode; and 2) do not allow for this privilege to be overridden by setting the `MPRV` bit of the `mstatus` register to zero. This is in line with EF2.

A2: We set up the Physical Memory Attributes (PMA) in line with EF2. Specifically:

1. We declare there to be a single PMA region. This is to make sure that all correctly aligned memory accesses will be executed, disallowing cases in which the load or store would cross beyond the end of a PMA region (for example, if the size of a PMA region is not a multiple of four, and a four-byte load was requested from the address that is the multiple of four closest to its end).
2. We set the base address of this single PMA region to be zero, and its size to be at least 2^{29} bytes. In doing so, we ensure that the address space of OpenVM, which is $[0, 2^{29})$, is supported.
3. We set this entire single PMA region to be readable, writable, and disallow any misaligned accesses. This last point is not strictly relevant, as OpenVM only supports correctly aligned memory accesses, but the Lean RISC-V specification requires an explicit choice of how such accesses should be treated.

A3: We disable the host/target mechanism of RISC-V. This is in line with EF3, as support for this mechanism should not be required for L1 proving.

A4: In order for all opcodes to be executable by the Lean RISC-V specification, we assume the existence of: 1) the `misa` register, required for most control-flow opcodes; and 2) the `mseccfg` register, required for JALR.

```
def RISC_V_assumptions
  (state : PreSail.SequentialState RegisterType Sail.trivialChoiceSource)
  (mstatus : RegisterType Register.mstatus)
  (pmaRegion : PMA_Region)
  (misa : RegisterType Register.misa)
  (mseccfg : RegisterType Register.mseccfg)
: Prop :=
  -- Assumption A1.1: machine privilege
  Sail.readReg Register.cur_privilege state = EStateM.Result.ok Privilege.Machine state ∧
  -- Assumption A1.2: MPRV bit of the mstatus register not set
  (Sail.readReg Register.mstatus state = EStateM.Result.ok mstatus state ∧
   BitVec.extractLsb 17 17 mstatus = 0#1) ∧
  -- A2.1 : Single PMA region
  Sail.readReg Register.pma_regions state = EStateM.Result.ok [ pmaRegion ] state ∧
  -- A2.2 : with base 0 and at least 2^29 bytes in size
  pmaRegion.base = 0 ∧ OpenVM_address_space_size ≤ pmaRegion.size.toNat ∧
  -- A2.3 : with all addresses readable and writable, and misaligned accesses treated as errors
  pmaRegion.attributes.readable ∧ pmaRegion.attributes.writable ∧
  pmaRegion.attributes.misaligned_fault = misaligned_fault.AlignmentFault ∧
  -- Assumption A3: no host-target interface
  Sail.readReg Register.htif_tohost_base state = EStateM.Result.ok .none state ∧
  -- Assumption A4.1: misa register exists
```

```
  state.regs.get? Register.misa = .some misa ∧
  -- Assumption A4.2: mseccfg register exists
  Sail.readReg Register.mseccfg state = EStateM.Result.ok mseccfg state
```

We note that not supporting misaligned accesses is a legitimate choice made by OpenVM developers that is in line with the RISC-V specification, which requires mandatory support only for aligned writes and leaves the handling of misaligned accesses to the implementation.

Finally, when it comes to the underlying infrastructure, we assume the following:

I1: correctness of the theory and implementation of the OpenVM lookup argument and bus interaction mechanism;

I2: correctness of the theory and implementation of the proof system utilised by OpenVM;

I3: correctness of the RISC-V Lean specification with respect to the official RISC-V standard; and

I4: correctness of Lean's kernel.

2 Verification methodology

We discuss the key aspects of our methodology: formalisation of transpilation, the treatment of buses (both general and OpenVM-specific), and the workflow of per-opcode verification.

2.1 Transpilation

OpenVM transpiles RISC-V instructions into native OpenVM instructions, taking care of RISC-V specific requirements, such as the special treatment of register 0, in the process. This transpilation is formalised, for the opcodes covered by this engagement, in the `Transpiler.lean` file. As part of the transpilation, we formalise the conditions under which transpilation is successful. Specifically:

- as per this and this transpiler Rust code, that program counters are always divisible by 4; and
- as per this transpiler Rust code, that program counters are smaller than `MAX_ALLOWED_PC` $= 2^{30}$.

The formalisation of the transpiler is presented in more detail in the remainder of this report.

2.2 General treatment of buses

The OpenVM framework uses a number of buses for inter-chip communication and LogUp-based lookup-table information retrieval. A general bus entry over a field `F` is, in principle, of the form:

```
⟨multiplicity : F, data : Vector F⟩
```

consisting of a multiplicity (most commonly equalling 0, 1, or -1) and a vector of data, which can have different lengths and carry different meaning for different buses.

We formalise the notion of a bus as a Lean class, as follows:

```
/-- Type class for generic bus entries -/
class BusEntry (α : Type*) where
  multiplicity : α → F

  data_length : ℕ
  data : α → Vector F data_length

  axioms : α → Prop         -- Axioms on bus entries

  wf_properties  : α → Prop -- Well-formedness properties of bus entries
  wf_assume_cond : α → Prop -- Condition under which well-formedness properties are assumed
  wf_assert_cond : α → Prop -- Condition under which well-formedness properties need to be asserted

  -- Assuming and proving of well-formedness properties
  assume (a : α) := wf_assume_cond a → wf_properties a
  assert (a : α) := wf_assert_cond a → wf_properties a

  -- Serialisation, deserialisation, properties
  serialise (x : α) := (multiplicity x, data x)
  serialiseToList (x : α) := (multiplicity x, (data x).toList)
  deserialise : F × Vector F data_length → α

  inv_ser_deser (x : F × Vector F data_length) : serialise (deserialise x) = x
  inv_deser_ser {x : α} : deserialise (serialise x) = x
```

providing the ability to specify:

- the multiplicity, the length of the data, and the data itself;
- the bus entry axioms, that is, the constraints that are enforced by the OpenVM implementation and/or the constraints that can only be inferred from multi-chip reasoning;
- the bus entry well-formedness (WF) properties, that is, the constraints that are assumed to hold for every bus entry a chip reads and must be proven to hold for every bus entry a chip writes;

- the conditions under which WF-properties should be assumed and asserted;
- how to generically serialise a bus entry into a tuple and a list; and
- how to deserialise a bus entry correctly.

Importantly, this approach is not OpenVM-specific and can be repurposed for at least all Plonky3-based zkVMs.

2.2.1 On bus balancing

For the results related to balancing of buses, we instantiate the field `F` to the BabyBear field[1] and equip the bus data (lists of BabyBear values) with a measure μ:

```
μ : List FBB → ℕ
```

and define the notion of a *rising pair* with respect to μ as follows:

```
def rising_pair
  (dr ds : List FBB)
  (_ : μ dr < μ ds)
: List (FBB × List FBB) :=
  [ (-1, dr), ( 1, ds) ]
```

that is, as a pair for which the measure of the data that is put on the bus (`ds`) is strictly larger than the measure of the data that is read from the bus (`dr`). We define data read from the bus as the data with multiplicity -1 and the data put on the bus as the data with multiplicity 1, as that is the case for the two relevant buses in OpenVM (the `Execution` and `Memory` buses, as discussed shortly).

Next, we define the notion of a *rising bus* as the bus whose data consists of rising pairs, as follows:

```
def rising_bus
  (bus_data : List (List FBB × List FBB))
  (lt_proofs : ∀ entry ∈ bus_data, μ entry.1 < μ entry.2)
: List (FBB × List FBB) :=
  List.flatten
    (List.map
      (fun (x : { y // y ∈ bus_data }) ↦
        let ⟨ ⟨ dr, ds ⟩ , pf ⟩ := x
        rising_pair μ dr ds (lt_proofs (dr, ds) pf )) bus_data.attach)
```

2.2.2 The rising bus balancing theorem

The main bus balancing theorem that we prove is as follows:

```
lemma rising_bus_with_single_balancer_characterisation
  {ldata rdata : List FBB}
  (bus_data : List (List FBB × List FBB))
  (lt_proofs : ∀ entry ∈ bus_data, μ entry.1 < μ entry.2)
  (h_not_empty : 0 < bus_data.length)
  (h_len_bus : ([(1, ldata)] ++ (rising_bus μ bus_data lt_proofs) ++ [(-1, rdata)]).length < BB_prime)
  (h_balance : InteractionList.is_balanced
                ([((1 : FBB), ldata)] ++ (rising_bus μ bus_data lt_proofs) ++ [((-1 : FBB), rdata)]))
:
  ∃ xs, bus_data.Perm (List.zip (ldata :: xs) (xs ++ [rdata])) ∧
        List.Pairwise (fun x₁ x₂ ↦ μ x₁ < μ x₂) (ldata :: xs ++ [rdata])
```

and it states that, if we have a non-empty rising bus, and if this rising bus is balanced out by a pair of the form $[\langle 1, \texttt{ldata}\rangle, \langle -1, \texttt{rdata}\rangle]$, and if the length of this balanced-out bus is less than `BB_prime` $- 2 = 2013265919$ pairs, then the the rising bus can be rearranged so that the measure is strictly increasing, the measure of `ldata` is the smallest measure of the bus, and the measure of `rdata` is the largest measure on the bus. More importantly, if we think of the measure as time, it also means that every read (entry with multiplicity -1) reads from the latest write (entry with multiplicity 1), which corresponds to the well-known notion of consistency. This theorem is key for proving execution and memory consistency (cf. §4).

[1] The BabyBear field is the finite field used by OpenVM. Its size equals `BB_prime` $= 2013265921$.

2.3 The OpenVM buses

OpenVM makes use of the following six buses in the implementation of the RISC-V opcodes relevant for this engagement. These are identified by their index (a natural number), and are formalised in Lean as follows:

```
abbrev ExecutionBus : ℕ := 0
abbrev MemoryBus : ℕ := 1
abbrev RangeCheckerBus : ℕ := 4
abbrev ProgramBus : ℕ := 8
abbrev BitwiseBus : ℕ := 9
abbrev RangeTupleCheckerBus : ℕ := 11
```

We discuss each of them in the upcoming subsections. The important points to remember are that:

- the semantics of the bus entries is effectively given by the axioms and well-formedness properties; and
- by construction, the chips read contents from all six buses but write only to the `Execution` and `Memory` buses, which means that these are the only two buses for which we will need to prove well-formedness properties.

2.3.1 The `Execution` bus

The entry structure and semantics of the Execution bus are formalised as follows:

```
structure ExecutionBusEntry where
  multiplicity : F
  pc : F
  timestamp : F

instance ExecutionBusEntryInstance : BusEntry FBB (ExecutionBusEntry FBB) := {
  ...
  axioms := fun ⟨mul, pc, timestamp⟩ ⇒ ¬ mul = 0 → pc < 2^30 ∧ pc % 4 = 0 ∧ timestamp < 2 ^ 29
  wf_properties  := fun _ ⇒ True
  ...
}
```

which tells us that:

- the `Execution` bus entries carry two pieces of information: the program counter (pc) and the timestamp;
- the program counters are axiomatically declared to be smaller than 2^{30} and aligned to multiples of four;
- the timestamps are axiomatically declared to be smaller than 2^{29}; and that
- there are no well-formedness properties for the entries of this bus;

Justification of axiomatic properties. The axiomatic properties in question are the properties guaranteed to hold for any program counter on the `Program` bus as per the transpilation process. They are justified on paper as follows:

E1) every row of every RV32IM-related chip puts on the execution bus either two zero-multiplicity entries or a pair of entries of the form $[\langle -1, [\mathtt{pc}, \mathtt{timestamp}]\rangle, \langle 1, [\mathtt{pc}', \mathtt{timestamp} + \mathtt{increment}]\rangle]$ where $1 \leq \mathtt{increment} \leq 3$;

E2) this Rust code of the connector chip enforces that the `start_timestamp` of the execution bus equals 1;

E3) this Rust code of the connector chip enforces that the `end_timestamp` of the execution bus is smaller than 2^{29};

E4) the verifier enforces that there are less than `BB_prime` overall interactions;

E5) every row of every RV32IM-related chip, if it puts entries on the execution bus, then it also emits more than `increment` overall interactions;

E6) all timestamps are smaller then $\mathtt{BB\_prime} - 3$: otherwise, given (E1), there would have to exist a chain of timestamps with maximal increment of 3 from 1 to somewhere in $[\mathtt{BB\_prime} - 3, \mathtt{BB\_prime})$, which is not possible given (E4), and (E5);

E7) from (E1) and (E6), we get that the execution bus is a rising bus;

E8) this Rust code additionally puts the entries $[\langle -1, [\mathtt{end\_pc}, \mathtt{end\_timestamp}]\rangle, \langle 1, [\mathtt{start\_pc}, \mathtt{start\_timestamp}]\rangle]$ on the execution bus;

E9) given the previous points, the balancing of the execution bus, and the rising bus balancing theorem, we obtain that all of the timestamps are less than 2^{29}, justifying the timestamp-related axiomatic property;

E10) every chip obtains pc from the `Program` bus, from which we get, given transpiler properties, that $\text{pc} < 2^{30}$ and that $\text{pc} \,\%\, 4 = 0$, justifying the pc-related axiomatisation for the execution bus entries with multiplicity -1;

E11) given the previous points and the rising bus balancing theorem, we get that all of the pcs on the execution bus with multiplicity 1, *except the one with the highest timestamp*, must have a matching pc with multiplicity -1, which all originate from the `Program` bus and, therefore, satisfy the pc-related axiomatic properties;

E12) this Rust code of the connector chip enforces that the pc with the highest timestamp can be looked up on the `Program` bus, which establishes the axiomatic pc-related property for this pc as well, concluding the justification.

2.3.2 The `Memory` bus

The entry structure and semantics of the `Memory` bus are formalised as follows:

```
structure MemoryBusEntry where
  multiplicity : F
  as : F        -- address space
  ptr : F       -- memory address for 32-bit read/write
  x0 : F        -- first byte read/written
  x1 : F        -- second byte read/written
  x2 : F        -- third byte read/written
  x3 : F        -- fourth byte read/written
  timestamp : F -- timestamp

instance MemoryBusEntryInstance : BusEntry FBB (MemoryBusEntry FBB) := {
  ...
  axioms := fun ⟨mul, _, _, _, _, _, _, timestamp⟩ ⇒ ¬ mul = 0 → timestamp < 2 ^ 29
  wf_properties :=
    fun ⟨_, as, ptr, x0, x1, x2, x3, _⟩ ⇒
      as.val < 3 ∧ ptr.val < 2 ^ 29 ∧ x0.val < 256 ∧ x1.val < 256 ∧ x2.val < 256 ∧ x3.val < 256
  wf_assume_cond := fun entry ⇒ entry.1 = -1,
  wf_assert_cond := fun entry ⇒ entry.1 = 1,
  ...
}
```

which tells us:

- that the `Memory` bus entries carry seven pieces of information, described in code comments above;
- what the bounds on all of these seven parameters should be (for the timestamp axiomatically and for the others through the WF-properties); and
- that the WF-properties should be assumed/asserted if the multiplicity equals -1/1 (observed empirically).

Justification of axiomatic properties. The axiomatic property of the timestamps is justified on paper as follows. Without loss of generality, take a fixed pointer in memory, `ptr`, for which there are entries on the memory bus. Then:

M1) this Rust code writes the initial value at `ptr` at initial timestamp 0;

M2) every row of every of every RV32IM-related chip that interacts with `ptr` puts on the memory bus pairs of entries with timestamps of the form $(\texttt{prev\_timestamp}, \texttt{cur\_timestamp})$, where `prev_timestamp` corresponds to a read, `cur_timestamp` corresponds to a write, where all of the `current_timestamps` are all unique and all in the range $[\texttt{timestamp}, \texttt{timestamp} + n)$, and where the execution bus entry corresponding to the row in question of the chip in question is $[\langle -1, [\text{pc}, \texttt{timestamp}]\rangle, \langle 1, [\text{pc}', \texttt{timestamp} + n]\rangle]$;

M3) this Rust code reads the final value at `ptr` at the final timestamp;

M4) by (E1)-(E8) and (M1)-(M3), we have that all of the writes for `ptr` have timestamps smaller than 2^{29};

M5) by the balancing of the memory bus per-pointer, then all of the reads for `ptr` must also be smaller than 2^{29}, making all of the timestamps on the memory bus associated with `ptr` smaller than 2^{29};

M6) as `ptr` is arbitrarily chosen, given previous points, we have that all timestamps on the memory bus are smaller than 2^{29}, which is the desired property.

Justification of WF-properties. These are the invariants that must hold for the initial memory and need to be proven to be preserved by the memory writes of each RV32IM-related chip. They are guaranteed to hold for the initial memory by the boundary chips, and the preservation proofs are provided in Lean and discussed in §3.1.7.

2.3.3 The `RangeChecker` bus

The entry structure and semantics of the `RangeChecker` bus are formalised as follows:

```
structure RangeCheckerBusEntry where
  multiplicity : F
  val : F
  deg : F

instance RangeCheckerBusEntryInstance : BusEntry FBB (RangeCheckerBusEntry FBB) := {
  ...
  axioms         := fun _             ⇒ True
  wf_properties  := fun ⟨_, val, deg⟩ ⇒ deg.val < 31 ∧ val.val < 2 ^ deg.val
  wf_assume_cond := fun ⟨mul, _, _⟩   ⇒ mul = 1,
  ...
}
```

which tells us that:

- the `RangeChecker` bus entries carry two pieces of information: `deg` and `val`;
- it holds that $\texttt{val} < 2^{\texttt{deg}}$ (hence the range-checking name of the bus); and
- the WF-properties should be assumed if multiplicity equals 1 (observed empirically).

Justification of WF-properties. These properties follow from the construction of the range-checking lookup table.

2.3.4 The `Program` bus

The entry structure and semantics of the `Program` bus are formalised as follows:

```
def ProgramBusEntry.operand_properties (entry : ProgramBusEntry FBB) : Prop :=
  ∃ instruction data,
    (Transpiler.transpile_op instruction entry.multiplicity entry.pc = .some data) ∧
    (ProgramBusEntry.deserialise FBB data = entry)

structure ProgramBusEntry where
  multiplicity : F
  pc : F
  opcode : F
  xa xb xc xd xe xf xg : F

instance ProgramBusEntryInstance : BusEntry FBB (ProgramBusEntry FBB) := {
  ...
  axioms := fun _ ⇒ True
  wf_properties := ProgramBusEntry.operand_properties
  wf_assume_cond := fun entry ⇒ entry.1 = 1,
  wf_assert_cond := fun entry ⇒ entry.1 = -1,
  ...
}
```

which tells us that:

- the `Program` bus entries carry nine pieces of information: the program counter, `pc`; the opcode, `opcode`; and other parameters which denote, e.g., the registers from which data should be retrieved, immediate values, etc.;
- that the WF-properties are obtained from the transpilation of the opcode in question; and
- the WF-properties should be assumed if multiplicity equals 1 (observed empirically).

Justification of WF-properties. These properties follow from the definition of the transpiler.

2.3.5 The `Bitwise` bus

The entry structure and semantics of the `Bitwise` bus are formalised as follows:

```
structure BitwiseBusEntry where
  multiplicity : F
  a : F
  b : F
  c : F
  op : F

instance BitwiseBusEntryInstance : BusEntry FBB (BitwiseBusEntry FBB) := {
  ...
  axioms         := fun _                ⇒ True
  wf_properties  := fun ⟨_, a, b, c, op⟩ ⇒ deg.val < 31 ∧ val.val < 2 ^ deg.val
  wf_assume_cond := fun ⟨mul, _, _⟩      ⇒ mul = 1,
  ...
}
```

which tells us that:

- the `Bitwise` bus entries carry four pieces of information: `a`, `b`, `c`, and `op`;
- it holds that:
  - `a.val` < 256, `b.val` < 256, and `op` equals either 0 or 1;
  - if `op` $= 0$, then `c.val` $= 0$; and
  - if `op` $= 1$, then `c.val` $=$ `a.val` $\oplus$ `b.val`;
- the WF-properties should be assumed if multiplicity equals 1 (observed empirically).

What is interesting about this bus is that the flexibility of the `op` parameter allows it to sometimes be used only as a range checker, and other times to also compute the exclusive disjunction.

Justification of WF-properties. These properties follow from the construction of the bitwise lookup table.

2.3.6 The `RangeTupleChecker` bus

The entry structure and semantics of the `RangeChecker` bus are formalised as follows:

```
structure RangeTupleCheckerBusEntry where
  multiplicity : F
  x1 : F
  x2 : F

instance RangeTupleCheckerBusEntryInstance : BusEntry FBB (RangeTupleCheckerBusEntry FBB) := {
  ...
  axioms         := fun _           ⇒ True
  wf_properties  := fun ⟨_, x1, x2⟩ ⇒ x1 < 256 ∧ x2 < 2048
  wf_assume_cond := fun ⟨mul, _, _⟩ ⇒ mul = 1,
  ...
}
```

which tells us that:

- the `RangeTupleChecker` bus entries carry two pieces of information: `x1` and `x2`;
- it holds that `x1` $< 2^8 = 256$ and `x2` $< 2^{11} = 2048$, range-checking both pieces of data at the same time; and
- the WF-properties should be assumed if multiplicity equals 1 (observed empirically).

Justification of WF-properties. The WF-properties follow from the definition of the tuple-range-checking lookup table. We note that these range checks can, in principle, be instantiated by the user, and it happens that, for the opcodes verified in this engagement, the upper bounds are set to 2^8 and 2^{11}.

2.4 Opcode verification workflow

For every verified opcode verified, we perform the following workflow in Lean, noting that there are parts of the workflow that are re-used for multiple opcodes, as OpenVM chips almost never implement a single opcode:

1. implement the transpilation of the opcode in our formalisation of the OpenVM transpiler;
2. formalise the core air of the chip that implements the opcode;
3. formalise the associated core air adapter, which connects the chip to the various buses;
4. formalise the air wrapper, which brings together the core air and its adapter;
5. extract the constraints and bus contents of the chip in question;
6. transform the extracted constraints and bus contents into human-readable form;
7. prove that the extracted constraints imply a pure human-readable RISC-V specification of the opcode, closely related to the behaviour specified in the official Lean RISC-V specification; and, finally
8. connect the structural part of the Lean RISC-V opcode specification to the pure specification of (7), effectively proving overall correctness of the opcode implementation.

The main 45 correctness theorems, one per each analysed opcode, can all be found in the Equivalence.lean file.

3 An in-depth look into verification

In this section, we provide detailed, illustrative examples of the verification process. In particular: we show the verification workflow as described in §2.4 in detail for the ADD/ADDI opcodes (implemented by the base ALU chip), and also discuss the specificities of some branching opcodes and memory-manipulating opcodes.

3.1 Example 1: ADD/ADDI

3.1.1 Step 1: Transpilation

As a first step, we extend the transpiler with support for the ADD and ADDI opcodes:

```
def transpile_op
  (inst : instruction) -- instruction, as per the Lean RISC-V spec
  (multiplicity : FBB) -- multiplicity
  (pc : FBB)           -- program counter
:
  Option (FBB × Vector FBB 9) -- Program bus entry
:=
  if pc % 4 = 0 ∧ pc < 2 ^ 30 then
    match inst with
    ...
    | .RTYPE (rs2, rs1, rd, rop.ADD) ⇒
      .some (if rd == regidx.Regidx 0
       then (multiplicity, #v[pc, 1, 0, 0, 0, 0, 0, 0, 0])
       else (multiplicity, #v[pc, 512, ind rd, ind rs1, ind rs2, 1, 1, 0, 0]))
    ...
    | .ITYPE (imm, rs1, rd, iop.ADDI) ⇒
      .some (if rd == regidx.Regidx 0
       then (multiplicity, #v[pc, 1, 0, 0, 0, 0, 0, 0, 0])
       else (multiplicity, #v[pc, 512, ind rd, ind rs1, utof (sign_extend_24 imm), 1, 0, 0, 0]))
```

where

```
  def ind           (rd : regidx)    : FBB       := ⟨4 * (regidx_to_fin rd).val, by omega⟩
  def sign_extend_24 (bv : BitVec 12) : BitVec 24 := bv.signExtend 24
  def utof          (bv : BitVec n). : FBB       := bv.toNat
```

noting:

- that the transpiler receives the instruction to transpile directly from the Lean RISC-V spec `instruction` type
- that the transpiler returns the corresponding bus entry that should be put on the `Program` bus; and
- how the transpilation branches on whether or not the target register is 0, in which case no operation is performed, mirroring the RISC-V specification; and
- in particular for ADDI, how the 12-bit immediate `imm` is sign-extended to 24-bits and then cast into the BabyBear field; the constraints of the base ALU chip further sign-extend this value to 32 bits.

3.1.2 Step 2: Core air

The ADD/ADDI opcodes are implemented as part of the base ALU chip. In particular, the structural part of the base ALU core air is implemented in the Rust codebase as follows:

```
pub struct BaseAluCoreCols<T, const NUM_LIMBS: usize, const LIMB_BITS: usize> {
  pub a b c : [T; NUM_LIMBS],

  pub opcode_add_flag: T,
  pub opcode_sub_flag: T,
  pub opcode_xor_flag: T,
  pub opcode_or_flag : T,
  pub opcode_and_flag: T,
}
```

We transfer this structure to Lean, maintaining strong eyeball correspondence between the two definitions. We achieve this by implementing a series of macros as part of our Lean zk-circuit infrastructure that allow users to define air-related structures in a modular way. In particular, the two macros that are used for the generation of specialised circuit types are called `#define_air` and `#define_subair`: `#define_subair` is used for creating nested structure within a trace, whereas `#define_air` is used for generating the overall structure of the circuit.

With this in mind, the structure above is formalised in Lean, in the `BaseAluCoreAir.lean` file, as:

```
#define_subair "BaseAluCoreAir" using "openvm_encapsulation" where
  Column["a_0"]
  Column["a_1"]
  Column["a_2"]
  Column["a_3"]
  Column["b_0"]
  Column["b_1"]
  Column["b_2"]
  Column["b_3"]
  Column["c_0"]
  Column["c_1"]
  Column["c_2"]
  Column["c_3"]
  Column["opcode_add_flag"]
  Column["opcode_sub_flag"]
  Column["opcode_xor_flag"]
  Column["opcode_or_flag"]
  Column["opcode_and_flag"]
```

The correspondence between the two is almost one-to-one, with the only difference being that the Rust arrays are explicitly deconstructed per-index: that is, for example, the Rust `pub a: [T; NUM_LIMBS]`, as `NUM_LIMBS = 4` for 32-bit OpenVM, corresponds to the four Lean columns `a_0`, `a_1`, `a_2`, and `a_3`. From this macro, we then automatically generate two structures, `Raw_BaseAluCoreAir` and `Valid_BaseAluCoreAir`, as follows:

```
structure Raw_BaseAluCoreAir (F : Type) : Type
number of parameters: 1
fields:
  Raw_BaseAluCoreAir.columns : ℕ → ℕ → ℕ → F
  Raw_BaseAluCoreAir.a_0 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.a_1 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.a_2 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.a_3 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.b_0 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.b_1 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.b_2 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.b_3 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.c_0 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.c_1 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.c_2 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.c_3 : ℕ → ℕ → F
  Raw_BaseAluCoreAir.opcode_add_flag : ℕ → ℕ → F
  Raw_BaseAluCoreAir.opcode_sub_flag : ℕ → ℕ → F
  Raw_BaseAluCoreAir.opcode_xor_flag : ℕ → ℕ → F
  Raw_BaseAluCoreAir.opcode_or_flag  : ℕ → ℕ → F
  Raw_BaseAluCoreAir.opcode_and_flag : ℕ → ℕ → F
constructor:
  Raw_BaseAluCoreAir.mk {F : Type} (columns : ℕ → ℕ → ℕ → F)
    (a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3 c_0 c_1 c_2 c_3 opcode_add_flag opcode_sub_flag
     opcode_xor_flag opcode_or_flag opcode_and_flag : ℕ → ℕ → F) : Raw_BaseAluCoreAir F
```

```
@[reducible] def Valid_BaseAluCoreAir : Type → Type :=
  fun F ↦ { c // c.isValid }
```

where:

– the `columns` field purely represents a circuit (collection of traces): specifically, `columns col row rotation` is

meant to denote the value in column `col` of row $(\texttt{row} + \texttt{rotation}) \% \texttt{trace\_length}$, which is enforced at the `#define_air` level, as illustrated shortly;

- the remaining 17 typed members of the structure represent the 17 columns, which are named as in the original Rust code: for example, `a_0`, `opcode_add_flag`, etc.; and
- the `isValid` predicate states the these 17 named columns (that is, sub-air members) are indeed equal to the appropriate locations in the appropriate traces, and, more generally, that any nested sub-airs are also valid.

Next, we look at the Rust code of the `eval` function of the base ALU chip and build a model of it in Lean. This amounts to defining and folding the definitions of various local variables used by `eval`. For example, in the Rust codebase there is the array `carry_add` that holds the carries from the addition:

```
let mut carry_add: [AB::Expr; NUM_LIMBS] = array::from_fn(|_| AB::Expr::ZERO);
let carry_divide = AB::F::from_canonical_usize(1 << LIMB_BITS).inverse();
...
for i in 0..NUM_LIMBS {
  ...
  carry_add[i] = AB::Expr::from(carry_divide) * (b[i] + c[i] - a[i] +
                  if i > 0 { carry_add[i - 1].clone() } else { AB::Expr::ZERO });
  ...
}
```

a part of which is encoded into Lean as follows, again index-by-index:

```
def Valid_BaseAluCoreAir.carry_divide (c : Valid_BaseAluCoreAir FBB) : FBB := 2005401601

def Valid_BaseAluCoreAir.carry_add_0 (c : Valid_BaseAluCoreAir FBB) (row rotation : ℕ) : FBB :=
  c.carry_divide * (c.b_0 row rotation + c.c_0 row rotation - c.a_0 row rotation)

def Valid_BaseAluCoreAir.carry_add_1 (c : Valid_BaseAluCoreAir FBB) (row rotation : ℕ) : FBB :=
  c.carry_divide *
    (c.b_1 row rotation + c.c_1 row rotation - c.a_1 row rotation + c.carry_add_0 row rotation)

...
```

where the eyeball correspondence is once more straightforward, noting the above-mentioned row and rotation. This process is relatively simple to do by hand, but would be complex to do automatically as it would require reasoning about arbitrary rust code. The general strategy is to create a Lean definition for each let statement in the Rust constraints and a corresponding proof that the full expression is equal to this newly defined member. For the two carries shown above, these proofs are as follows:

```
@[openvm_encapsulation]
lemma BaseAluCoreAir.carry_add_0_def
  [Field F] (c : Valid_BaseAluCoreAir F) (row rotation: ℕ)
:
  (2005401601 : F) * (c.b_0 row rotation + c.c_0 row rotation - c.a_0 row rotation) =
    c.carry_add_0 row rotation
:= rfl

@[openvm_encapsulation]
lemma BaseAluCoreAir.carry_add_1_def
  [Field F] (c : Valid_BaseAluCoreAir F) (row rotation: ℕ)
:
  (2005401601 : F) * (c.b_1 row rotation + c.c_1 row rotation - c.a_1 row rotation +
    c.carry_add_0 row rotation) =
      c.carry_add_1 row rotation
:= rfl
```

Note how these proofs gets included in the set of those that the simplification proofs can use, denoted by `openvm_encapsulation`, and now when the full expression appears in the constraint, it will get folded up into our newly defined member. The effect of this process is that the constraints will ultimately look almost exactly like the Rust assertions, making them easier to reason about.

3.1.3 Step 3: Core air adapter

The next step is to encode the adapter air for the base ALU chip. Its structural part is implemented in the Rust codebase as follows:

```
pub struct Rv32BaseAluAdapterCols<T> {
    pub from_state: ExecutionState<T>,
    pub rd_ptr: T,
    pub rs1_ptr: T,
    pub rs2: T,    /// Pointer if rs2 was a read, immediate value otherwise
    pub rs2_as: T, /// 1 if rs2 was a read, 0 if an immediate
    pub reads_aux: [MemoryReadAuxCols<T>; 2],
    pub writes_aux: MemoryWriteAuxCols<T, RV32_REGISTER_NUM_LIMBS>,
}
```

which, as we can see, contains three sub-airs, `from_state`, `from_state`, and `from_state`, together with several other columns. In Lean, this structure is encoded as follows, in the `Rv32BaseAluAdapterAir.lean` file:

```
#define_subair "Rv32BaseAluAdapterAir" using "openvm_encapsulation" where
  SubAir["from_state": "ExecutionState" width := 2]
  Column["rd_ptr"]
  Column["rs1_ptr"]
  Column["rs2"]
  Column["rs2_as"]
  SubAir["reads_aux_0": "MemoryReadAuxCols" width := 3]
  SubAir["reads_aux_1": "MemoryReadAuxCols" width := 3]
  SubAir["writes_aux": "MemoryWriteAuxCols_4" width := 7]
```

where the auxiliary airs, `ExecutionState`, `MemoryReadAuxCols`, and `MemoryWriteAuxCols_4` are defined separately in `ExecutionState.lean` and `Memory.lean`. This particular encoding above showcases how easy is it to combine airs in our verification infrastructure, mirroring their Rust implementations.

3.1.4 Step 4: Base ALU air wrapper

The last structural step is to connect the core air with the adapter air using the `VmAirWrapper` mechanism, implemented in the Rust codebase as follows:

```
pub type Rv32BaseAluAir =
    VmAirWrapper<Rv32BaseAluAdapterAir, BaseAluCoreAir<RV32_REGISTER_NUM_LIMBS, RV32_CELL_BITS>>;
```

and encoded in Lean as follows (in the `Alu/VmAirWrapper_alu.lean` file):

```
#define_air "VmAirWrapper_alu" using "openvm_encapsulation" where
  MainSubAir["adapter": "Rv32BaseAluAdapterAir" width := 19]
  MainSubAir["core": "BaseAluCoreAir" width := 17]
```

which results, analogously to the sub-air definitions above, in an automatic creation of the following Lean structure representing an air:

```
structure Raw_VmAirWrapper_alu (F ExtF : Type) : Type
number of parameters: 2
fields:
  Raw_VmAirWrapper_alu.buses : ℕ → List (F x List F)
  Raw_VmAirWrapper_alu.challenge : ℕ → ExtF
  Raw_VmAirWrapper_alu.exposed : ℕ → ExtF
  Raw_VmAirWrapper_alu.main : ℕ → ℕ → ℕ → ℕ → F
  Raw_VmAirWrapper_alu.permutation : ℕ → ℕ → ℕ → ExtF
  Raw_VmAirWrapper_alu.preprocessed : ℕ → ℕ → ℕ → F
  Raw_VmAirWrapper_alu.public_values : ℕ → F
  Raw_VmAirWrapper_alu.last_row : Nat
  Raw_VmAirWrapper_alu.adapter : Raw_Rv32BaseAluAdapterAir F
  Raw_VmAirWrapper_alu.core : Raw_BaseAluCoreAir F
constructor:
  Raw_VmAirWrapper_alu.mk {F ExtF : Type}
    (buses : ℕ → List (F x List F)) (challenge exposed : ℕ → ExtF)
```

```
    (main : ℕ → ℕ → ℕ → ℕ → F) (permutation : ℕ → ℕ → ℕ → ExtF)
    (preprocessed : ℕ → ℕ → ℕ → F) (public_values : ℕ → F)
    (last_row : ℕ) (adapter : Raw_Rv32BaseAluAdapterAir F) (core : Raw_BaseAluCoreAir F) :
    Raw_VmAirWrapper_alu F ExtF
```

where the most important fields for this engagement, in addition to the adapter and the core, are:

- `buses`, which contains the data that is put on the buses by the base ALU chip;
- `main`, which contains the main trace of the base ALU chip;
- `last_row`, which tells us how many rows the main trace has, in the sense that the above-mentioned `trace_length` corresponds to `last_row` $+ 1$.

as well as the wrapper `Valid_VmAirWrapper_alu` structure:

```
@[reducible]
def Valid_VmAirWrapper_alu : Type → Type → Type :=
fun F ExtF ↦ { c // c.isValid }
```

where the `is_Valid` predicate states, in addition to sub-air validity and column matching, that the `rotation` parameter behaves as expected.

This air definition is accompanied by the manually written definitions and proofs that connect the columns of the base ALU core air to the columns of the adapter air as per the Rust code, such as, for example:

```
let rs2_imm =
  rs2_limbs[0].clone() +
  rs2_limbs[1].clone() * AB::Expr::from_canonical_usize(1 << RV32_CELL_BITS) +
  rs2_sign.clone() * AB::Expr::from_canonical_usize(1 << (2 * RV32_CELL_BITS));
```

which is encoded in Lean as follows:

```
def Valid_VmAirWrapper_alu.rs2_imm
  [Field ExtF]
  (c : Valid_VmAirWrapper_alu FBB ExtF)
  (row rotation : ℕ)
: FBB :=
  c.rs2_limbs row rotation 0 + c.rs2_limbs row rotation 1 * 256 + c.rs2_sign row rotation * 65536
```

3.1.5 Step 5: Constraint extraction

The next file of relevance is `Extraction/VmAirWrapper_alu.lean`, which is generated with the help of our constraint extraction system and contains all of the extracted main-trace constraints, permutation-related constraints, and the contents placed on all of the buses by the base ALU chip.

In particular, we extract the constraints by modifying the `stark-backend` crate that OpenVM uses. Precisely, in the `AirKeygenBuilder::generate_pk` method, the `SymbolicRapBuilder` is already being used to generate symbolic constraints for the air in order to calculate the degree of those constraints. This `SymbolicRapBuilder` also creates a list of symbolic bus interactions generated during the evaluation of the air. These symbolic constraints and bus interactions are in the form of Abstract Syntax Trees for arithmetic expressions. We hook into this process, taking the generated symbolic constraints and interactions and printing them to the standard output so that they can be retrieved without having to further modify the control flow of the crate. We insert this logic into the stark-backend crate so that it can easily be run by simply running the tests for each circuit.

From the arithmetic-expression ASTs obtained from the stark backend, we create a purposefully simple and verbose flattening of each AST into Lean arithmetic expressions, avoiding any optimisation or simplification which could introduce bugs, preferring instead to allow Lean to do such things verifiably. This is done for all generated constraints first, and then for the bus interactions. As the permutation trace (that is, the trace in the extension field) is only used in these circuits for the constraints generated to enforce interactions, the extractor takes all generated constraint expressions and converts any that reference the permutation trace into comments. Again, great care is taken to make sure that any bugs would be immediately obvious, in the sense that constraints are commented out rather than deleted so that we can double check that only the correct constraints are being ignored.

When it comes to bus interactions, we create a list representation that includes all bus interactions specialised to coming from the first row, then the second, etc. This simplifies reasoning as we tend to reason about one row at a time. We also need to separate the interactions into the individual buses, which is done by constructing a hash map keyed on bus index and first inserting all interactions into there. As seen earlier, bus indices are left purely as integers, again for the sake of reducing processing done outside of Lean.

We illustrate the outcome of the extraction process using the constraint arising from the last (fourth) iteration of the following loop:

```
for i in 0..NUM_LIMBS {
  ...
  carry_sub[i] = AB::Expr::from(carry_divide)
    * (a[i] + c[i] - b[i] +
       if i > 0 { carry_sub[i - 1].clone() } else { AB::Expr::ZERO });
  builder.when(cols.opcode_sub_flag).assert_bool(carry_sub[i].clone());
}
```

which states that, when the `opcode_sub_flag` is non-zero, `carry_sub[3]` equals either zero or one. This constraint, which happens to be the 13th one, gets extracted to the following Lean code:

```
def constraint_13
  {C : Type → Type → Type}
  {F ExtF : Type} [Field F] [Field ExtF]
  [Circuit F ExtF C]
  (c : C F ExtF) (row: ℕ)
:=
  ((Circuit.main c (id := 0) (column := 32) (row := row) (rotation := 0)) * ((2005401601 *
  ((((Circuit.main c (id := 0) (column := 22) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 30) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 26) (row := row) (rotation := 0))) + (2005401601 *
  ((((Circuit.main c (id := 0) (column := 21) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 29) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 25) (row := row) (rotation := 0))) + (2005401601 *
  ((((Circuit.main c (id := 0) (column := 20) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 28) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 24) (row := row) (rotation := 0))) + (2005401601 *
  ((((Circuit.main c (id := 0) (column := 19) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 27) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 23) (row := row) (rotation := 0))) + 0)))))))) *
  ((2005401601 * ((((Circuit.main c (id := 0) (column := 22) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 30) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 26) (row := row) (rotation := 0))) +
  (2005401601 * ((((Circuit.main c (id := 0) (column := 21) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 29) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 25) (row := row) (rotation := 0))) +
  (2005401601 * ((((Circuit.main c (id := 0) (column := 20) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 28) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 24) (row := row) (rotation := 0))) +
  (2005401601 * ((((Circuit.main c (id := 0) (column := 19) (row := row) (rotation := 0)) +
  (Circuit.main c (id := 0) (column := 27) (row := row) (rotation := 0))) -
  (Circuit.main c (id := 0) (column := 23) (row := row) (rotation := 0))) + 0)))))))) - 1))) = 0
```

where, as can be seen, all of the variable definitions have been fully unfolded.

On the other hand, the contents of the buses is described as follows (for readability only showing the contents of the `Program` bus, whose index is 8, in full):

```
def constrain_interactions
  {C : Type → Type → Type}
  {F ExtF : Type} [Field F] [Field ExtF]
  [Circuit F ExtF C]
  (c : C F ExtF)
:=
  Circuit.buses c = fun index ⇒
    -- Execution bus
```

```
    if index = 0 then (List.range (Circuit.last_row c + 1)).flatMap (fun row ⇒ [ ... ])
    -- Memory bus
    else if index = 1 then (List.range (Circuit.last_row c + 1)).flatMap (fun row ⇒ [ ... ])
    -- Range-checking bus
    else if index = 4 then (List.range (Circuit.last_row c + 1)).flatMap (fun row ⇒ [ ... ])
    -- Program bus
    else if index = 8 then (List.range (Circuit.last_row c + 1)).flatMap (fun row ⇒
      [
        (((((0 + (Circuit.main c (id := 0) (column := 31) (row := row) (rotation := 0))) +
                 (Circuit.main c (id := 0) (column := 32) (row := row) (rotation := 0))) +
                 (Circuit.main c (id := 0) (column := 33) (row := row) (rotation := 0))) +
                 (Circuit.main c (id := 0) (column := 34) (row := row) (rotation := 0))) +
                 (Circuit.main c (id := 0) (column := 35) (row := row) (rotation := 0))),
        [
          (Circuit.main c (id := 0) (column := 0) (row := row) (rotation := 0)),
          (512 + ((((((0 + ((Circuit.main c (id := 0) (column := 31) (row := row) (rotation := 0)) * 0)) +
                           ((Circuit.main c (id := 0) (column := 32) (row := row) (rotation := 0)) * 1)) +
                           ((Circuit.main c (id := 0) (column := 33) (row := row) (rotation := 0)) * 2)) +
                           ((Circuit.main c (id := 0) (column := 34) (row := row) (rotation := 0)) * 3)) +
                           ((Circuit.main c (id := 0) (column := 35) (row := row) (rotation := 0)) * 4))),
          (Circuit.main c (id := 0) (column := 2) (row := row) (rotation := 0)),
          (Circuit.main c (id := 0) (column := 3) (row := row) (rotation := 0)),
          (Circuit.main c (id := 0) (column := 4) (row := row) (rotation := 0)),
          1,
          (Circuit.main c (id := 0) (column := 5) (row := row) (rotation := 0)),
          0,
          0
        ]
      ]
    )
    -- Bitwise bus
    else if index = 9 then (List.range (Circuit.last_row c + 1)).flatMap (fun row ⇒ [ ... ])
    -- Other buses
    else []
```

All of the constraints are pasted into a template file in the Lean project for this basic extraction output, yielding the `Extraction/VmAirWrapper_alu.lean` file. At this level, the Circuit structure referenced in the expressions (which contains the various traces) is completely generic and so the only human interaction is to name the namespace and simp categories for these constraints. Also, at this point, the logic of the constraints is entirely in Lean.

3.1.6 Step 6: Air wrapper with human-readable constraints

Next, we recover human-readable information from the extracted constraints and bus contents, which we do in the Constraints/VmAirWrapper_alu.lean file, starting from the constraints. For example, we prove that the fully unfolded constraint 13 from the previous section (`VmAirWrapper_alu.extraction.constraint_13 air row`) is equivalent to its human-readable form (`constraint_13 air row`) as follows:

```
@[VmAirWrapper_alu_constraint_and_interaction_simplification]
def constraint_13 (air : Valid_VmAirWrapper_alu F ExtF) (row : ℕ) : Prop :=
  air.core.opcode_sub_flag row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 1

@[VmAirWrapper_alu_air_simplification]
lemma constraint_13_of_extraction
  (air : Valid_VmAirWrapper_alu F ExtF) (row : ℕ)
: VmAirWrapper_alu.extraction.constraint_13 air row ↔ constraint_13 air row := by
  simp_all [openvm_encapsulation,
            VmAirWrapper_alu_constraint_and_interaction_simplification]
```

These simplification lemmas and their proofs are auto-generated with placeholders for the simplified constraints, and then these placeholders can be computed by running the proofs. Using the example at hand, we would have:

```
@[VmAirWrapper_alu_constraint_and_interaction_simplification]
def constraint_13 (air : Valid_VmAirWrapper_alu F ExtF) (row : ℕ) : Prop := sorry
```

and the above lemma. Its proof will fail, but the state after the simplifications

```
F ExtF : Type
inst†¹ : Field F
inst† : Field ExtF
air : Valid_VmAirWrapper_alu F ExtF
row : ℕ
⊢ air.core.opcode_sub_flag row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 1 ↔ sorry
```

tells us that `constraint_13` should equal

```
air.core.opcode_sub_flag row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 0 ∨ air.core.carry_sub_3 row 0 = 1
```

When it comes to the buses, we also restate their contents in more readable terms, as follows:

```
@[VmAirWrapper_alu_constraint_and_interaction_simplification]
def constrain_interactions (air : Valid_VmAirWrapper_alu FBB ExtF) : Prop :=
  air.buses = fun index ↦
  if index = ExecutionBus then (List.range (air.last_row + 1)).flatMap (executionBus_row air)
    else if index = MemoryBus then (List.range (air.last_row + 1)).flatMap (memoryBus_row air)
    else if index = RangeCheckerBus then (List.range (air.last_row + 1)).flatMap (rangeCheckerBus_row air)
    else if index = Program then (List.range (air.last_row + 1)).flatMap (programBus_row air)
    else if index = BitwiseBus then (List.range (air.last_row + 1)).flatMap (bitwiseBus_row air)
    else []
```

where, for instance, the contents of the `Program` bus are given by:

```
def programBus_row [Field ExtF]
  (air : Valid_VmAirWrapper_alu FBB ExtF)
  (row : ℕ)
: List (FBB × List FBB) :=
[
  air.core.is_valid row 0,
  [
    air.adapter.from_state.pc row 0, (air.core.ctx row 0).instruction.opcode,
    air.adapter.rd_ptr row 0, air.adapter.rs1_ptr row 0,
    air.adapter.rs2 row 0, 1, air.adapter.rs2_as row 0, 0, 0
  ]
]
```

which can be matched directly against the fully unfolded contents from the previous section. For example, given the following Rust code:

```
let flags = [
  cols.opcode_add_flag, cols.opcode_sub_flag, cols.opcode_xor_flag,
  cols.opcode_or_flag, cols.opcode_and_flag,
];

let is_valid = flags.iter().fold(AB::Expr::ZERO, |acc, &flag| {
  ...
  acc + flag.into()
});
```

and the definition of the airs at hand, we have that

```
air.core.is_valid row 0
```

corresponds precisely to

```
(((((0 + (Circuit.main c (id := 0) (column := 31) (row := row) (rotation := 0))) +
        (Circuit.main c (id := 0) (column := 32) (row := row) (rotation := 0))) +
        (Circuit.main c (id := 0) (column := 33) (row := row) (rotation := 0))) +
        (Circuit.main c (id := 0) (column := 34) (row := row) (rotation := 0))) +
        (Circuit.main c (id := 0) (column := 35) (row := row) (rotation := 0)))
```

The Constraints/VmAirWrapper_alu.lean file further contains lemmas that establish equivalence of unfolded and human-readable constraints and bus contents, as well as some basic properties about the constraints, such as that at most one opcode can be non-zero and that the chip opcodes must be in a certain range.

Axioms, assumptions, assertions. Finally, we define, based on our approach to bus entries as described in 2.2, the related axioms, assumptions, and assertions per row of the base ALU chip, as follows:

```
-- Axiomatic properties
def axioms [Interaction.BusEntry FBB a] (rowData : List a) : Prop :=
  List.Forall id (rowData.map (Interaction.BusEntry.axioms FBB))

-- Well-formedness properties to be assumed
def propertiesToAssume [Interaction.BusEntry FBB a] (rowData : List a) : Prop :=
  List.Forall id (rowData.map (Interaction.BusEntry.assume FBB))

-- Well-formedness properties to be asserted
def propertiesToAssert [Interaction.BusEntry FBB a] (rowData : List a) : Prop :=
  List.Forall id (rowData.map (Interaction.BusEntry.assert FBB))

-- All of the axiomatic properties per row
def axiomsPerRow [Field ExtF]
  (air : Valid_VmAirWrapper_alu FBB ExtF) (row : ℕ)
: Prop :=
  axioms (_executionBus_row air row)    ∧ axioms (_memoryBus_row air row) ∧
  axioms (_rangeCheckerBus_row air row) ∧ axioms (_programBus_row air row) ∧
  axioms (_bitwiseBus_row air row)

-- All of the well-formedness properties to be assumed per row
def wf_propertiesToAssumePerRow [Field ExtF] (air : Valid_VmAirWrapper_alu FBB ExtF) (row : ℕ)
: Prop :=
  propertiesToAssume (_executionBus_row air row)    ∧ propertiesToAssume (_memoryBus_row air row) ∧
  propertiesToAssume (_rangeCheckerBus_row air row) ∧
  propertiesToAssume (_progrmBus_row air row) ∧ propertiesToAssume (_bitwiseBus_row air row)

-- All of the well-formedness properties to be asserted per row
def wf_propertiesToAssertPerRow [Field ExtF] (air : Valid_VmAirWrapper_alu FBB ExtF) (row : ℕ)
: Prop :=
  propertiesToAssert (_executionBus_row air row)    ∧ propertiesToAssert (_memoryBus_row air row) ∧
  propertiesToAssert (_rangeCheckerBus_row air row) ∧
  propertiesToAssert (_programBus_row air row) ∧ propertiesToAssert (_bitwiseBus_row air row)
```

3.1.7 Step 7: Pure RISC-V specification

As part of our next step, in the Spec/BaseALU.lean file, we first prove that padding rows (that is, the rows for which the execution bus entries have multiplicity zero) do not affect the program and memory buses, that is, that no instruction is executed and that the memory is not modified:

```
/-- On non-valid rows, all interaction multiplicities on the
    execution, memory, and program buses equal zero -/
lemma non_valid_row_exec_mem_program_multiplicities_zero
:
  forall entry,
  entry ∈
    executionBus_row air row
     ++ memoryBus_row air row
     ++ programBus_row air row
    → entry.1 = 0
```

Next, we prove that, for non-padding rows (that is, the rows for which the execution bus entries have non-zero multiplicites (that is, when `air.core.is_valid row 0 = 1`) for the base ALU chip):

– if the values read from the buses satisfy the desired properties, then the values put on the buses also satisfy these properties; and

– the constraints imply the desired human-readable specification of the five base ALU opcodes, both for immediate and non-immediate variants.

For these proofs, we make use of the following hypotheses:

```
variable
  (ExtF : Type) [Field ExtF]
  (air : Valid_VmAirWrapper_alu FBB ExtF)
  (row : ℕ)
  (row_in_range : row <= air.last_row)
  (constraints : VmAirWrapper_alu.constraints.allHold air row row_in_range)
  (row_valid : air.core.is_valid row 0 = 1)
  (assumptions : axiomsPerRow air row)
  (propertiesToAssume : wf_propertiesToAssumePerRow air row)
```

Bus content properties. First, we prove that the properties that need to be asserted about the content that is put on the buses by the base ALU chip actually hold:

```
include
  row_valid constraints axioms propertiesToAssume
in
lemma wf_propertiesToAssert : wf_propertiesToAssertPerRow air row
```

Out of these properties, the only one that is noteworthy is the one that states that the four values denoting the result of executing the opcode are, in fact, bytes:

```
(air.core.a_0 row 0).val < 256 ∧ (air.core.a_1 row 0).val < 256 ∧
(air.core.a_2 row 0).val < 256 ∧ (air.core.a_3 row 0).val < 256
```

Human-readable RISC-V specification. From this point on, we no longer treat the chip as a whole, focusing instead on the specific opcodes implemented by the chip, but still For this example, we focus on the ADD and ADDI opcodes, noting that the related correctness theorems are still phrased generally for all of the base ALU opcodes, given that a substantial amount of the reasoning is the same. The theorem that covers the ADD opcode is as follows:

```
/-- From non-immediate ALU opcode to RISC-V opcode -/
def rop_of_ALU_opcode (opcode : FBB) : rop :=
  if opcode = 512 then .ADD else
  if opcode = 513 then .SUB else
  if opcode = 514 then .XOR else
  if opcode = 515 then .OR  else .AND

include
  row_valid constraints axioms propertiesToAssume
in
/-- The non-immediate variants of the five base ALU opcodes are implemented as per the RISC-V spec --/
theorem spec_base_ALU_non_imm
  (_ : air.adapter.rs2_as row 0 = 1)
:
  U32.toBV #v[(air.core.a_0 row 0).val,
              (air.core.a_1 row 0).val,
              (air.core.a_2 row 0).val,
              (air.core.a_3 row 0).val]
    =
  execute_RTYPE_pure
    (U32.toBV #v[(air.core.b_0 row 0).val,
                 (air.core.b_1 row 0).val,
                 (air.core.b_2 row 0).val,
                 (air.core.b_3 row 0).val])
    (U32.toBV #v[(air.core.c_0 row 0).val,
                 (air.core.c_1 row 0).val,
                 (air.core.c_2 row 0).val,
                 (air.core.c_3 row 0).val])
    (rop_of_ALU_opcode (air.core.ctx row 0).instruction.opcode)
```

where execute_RTYPE_pure captures the pure part of the executing RTYPE opcodes as per the Lean RISC-V spec:

```
/-- Pure part of 32-bit `execute_RTYPE` -/
def execute_RTYPE_pure (op1 : BitVec 32) (op2 : BitVec 32) (op : rop) :=
  match op with
  | .ADD ⇒ op1 + op2
  | .SUB ⇒ op1 - op2
  | .XOR ⇒ op1 ^^^ op2
  | .OR ⇒ op1 ||| op2
  | .AND ⇒ op1 &&& op2
  ...
```

Within the proof of the theorem, we branch on the five opcodes of the base ALU, providing a separate proof for each. The ADD-specific goal is as follows:

```
⊢ U32.toBV
    #v[BitVec.ofNat 8 ↑(air.core.a_0 row 0), BitVec.ofNat 8 ↑(air.core.a_1 row 0),
       BitVec.ofNat 8 ↑(air.core.a_2 row 0), BitVec.ofNat 8 ↑(air.core.a_3 row 0)] =
  U32.toBV
      #v[BitVec.ofNat 8 ↑(air.core.b_0 row 0), BitVec.ofNat 8 ↑(air.core.b_1 row 0),
        BitVec.ofNat 8 ↑(air.core.b_2 row 0), BitVec.ofNat 8 ↑(air.core.b_3 row 0)] +
    U32.toBV
      #v[BitVec.ofNat 8 ↑(air.core.c_0 row 0), BitVec.ofNat 8 ↑(air.core.c_1 row 0),
        BitVec.ofNat 8 ↑(air.core.c_2 row 0), BitVec.ofNat 8 ↑(air.core.c_3 row 0)]
```

and is discharged by case analysis on the carries:

```
  simp_all [← BitVec.toNat_inj, U32.toBV_toNat, U32.toNat]
  clear *- add_0 add_1 add_2 add_3 ub_a0 ub_a1 ub_a2 ub_a3
         ub_b0 ub_b1 ub_b2 ub_b3 ub_c0 ub_c1 ub_c2 ub_c3
  rcases add_0 <;> rcases add_1 <;>
  rcases add_2 <;> rcases add_3 <;>
  simp_all <;> omega
```

The proof of ADDI is done as part of an analogous theorem, targeting the appropriate immediate variants of the base ALU opcodes (ADDI, XORI, ORI, and ANDI):

```
/-- From immediate ALU opcode to RISC-V opcode -/
def iop_of_ALU_opcode (opcode : FBB) : iop :=
  if opcode = 512 then .ADDI else
  if opcode = 514 then .XORI else
  if opcode = 515 then .ORI else .ANDI

include row_valid constraints axioms propertiesToAssume in
/-- The immediate variants of the five base ALU opcodes are implemented as per the RISC-V spec -/
theorem spec_base_ALU_imm
  (h_imm : air.adapter.rs2_as row 0 = 0)
:
  U32.toBV #v[(air.core.a_0 row 0).val, (air.core.a_1 row 0).val,
              (air.core.a_2 row 0).val, (air.core.a_3 row 0).val]
    =
  execute_ITYPE_pure
    (BitVec.ofNat 12 (air.adapter.rs2 row 0).val)
    (U32.toBV #v[(air.core.b_0 row 0).val, (air.core.b_1 row 0).val,
                 (air.core.b_2 row 0).val, (air.core.b_3 row 0).val])
    (iop_of_ALU_opcode (air.core.ctx row 0).instruction.opcode)
```

where execute_ITYPE_pure corresponds to the pure part of the execution of the ITYPE opcodes as per the official Lean RISC-V spec:

```
/-- Pure part of 32-bit `execute_ITYPE` -/
def execute_ITYPE_pure (imm : BitVec 12) (op1 : BitVec 32) (op : iop) :=
  let immext := sign_extend (m := 32) imm
  execute_RTYPE_pure op1 immext (rop_of_iop op)
```

3.1.8 Step 8: Equivalence with RISC-V specification

The final step in proving correctness of opcode implementation is to connect the associated constraints with the structural part of the RISC-V specification.

Opcode input/output/behaviour. First, we isolate the input, the output, and the behaviour of each opcode. For ADD, this is done in the add.lean file, through which we go in detail, giving explanations in comments:

```
-- Input required for executing the ADD opcode
structure AddInput where
  r1_val : BitVec 32 -- first operand of ADD
  r2_val : BitVec 32 -- second operand of ADD
  rd     : Fin 32    -- intended output register (between 0 and 31 inclusive)
  PC     : BitVec 32 -- current program counter

-- Output of the ADD opcode
structure AddOutput where
  nextPC : BitVec 32                        -- next program counter
  rd : Option (Finset.Icc 1 31 × BitVec 32) -- result of executing the ADD operation

-- What does it mean to execute the ADD opcode?
def execute_RTYPE_add_pure (input : AddInput) : AddOutput := {
  -- next program counter is always the current one + 4
  nextPC := input.PC + 4#32
  -- the result of executing the operation is either:
  rd := if h : input.rd = 0
    -- no result, if the intended output register equals zero
    then .none
    -- bit-vector addition, which is stored in register rd, which we now know is non-zero
    else .some (
      ⟨ input.rd.val, by apply Finset.mem_Icc.mpr; omega ⟩,
      input.r1_val + input.r2_val
    ) : AddOutput
}

-- Equivalence between actual RISC-V execution of ADD and execute_RTYPE_add_pure
lemma execute_RTYPE_add_pure_equiv
  -- For a given AddInput
  (add_input : AddInput)
  -- and registers r1, r2, and rd, such that
  (r1 r2 rd : regidx)
  -- the first operand of the AddInput is stored in register r1 in the RISC-V state,
  (h_input_r1: read_xreg (regidx_to_fin r1) state = EStateM.Result.ok (add_input.r1_val) state)
  -- the second operand of the AddInput is stored in register r2 in the RISC-V state,
  (h_input_r2: read_xreg (regidx_to_fin r2) state = EStateM.Result.ok (add_input.r2_val) state)
  -- the intended output register of the AddInput is rd,
  (h_input_rd: add_input.rd = regidx_to_fin rd)
  -- and the program counter of the AddInput is stored in the PC register in the RISC-V state
  (h_input_pc: state.regs.get? Register.PC = .some add_input.PC)
: -- then
  -- executing the RISC-V add on a given state
  ( do
    Sail.writeReg Register.nextPC (Sail.BitVec.addInt (← Sail.readReg Register.PC) 4)
    LeanRV32D.Functions.execute (instruction.RTYPE (r2, r1, rd, rop.ADD))
  ) state
  -- is equivalent to, on the same state
  =
  -- running execute_RTYPE_add_pure
  let add_output : AddOutput := execute_RTYPE_add_pure add_input
  ( do
    -- storing the nextPC of the AddOutput in the PC register
    Sail.writeReg Register.nextPC add_output.nextPC
    -- and updating the intended output register with the sum of the two operands
    -- if the intended output register does not equal zero
    match add_output.rd with
```

```
      | .some (rd, rd_val) ⇒ write_xreg rd rd_val
      | .none ⇒ pure ()
    pure (ExecutionResult.Retire_Success ())
  ) state
```

This process for the ADDI opcode is similar (cf. addi.lean), and below we highlight the differences only:

```
structure AddiInput where
  r1_val : BitVec 32
  imm    : BitVec 12 -- 12-bit immediate value
  rd     : Fin 32
  PC     : BitVec 32

structure AddiOutput where
  nextPC : BitVec 32
  rd : Option (Finset.Icc 1 31 × BitVec 32)

def execute_ITYPE_addi_pure (input : AddiInput) : AddiOutput := {
  nextPC := input.PC + 4#32
  rd := if h: input.rd = 0
    then .none
    else .some (
      ⟨ input.rd.val, by apply Finset.mem_Icc.mpr; omega ⟩,
      -- bit-vector addition of the first operand and the sign-extended 12-bit immediate
      input.r1_val + BitVec.signExtend 32 input.imm
    ) : AddiOutput
}

lemma execute_ITYPE_addi_pure_equiv
  (addi_input : AddiInput)
  (imm : BitVec 12) -- 12-bit immediate value
  (r1 rd : regidx)
  (h_input_r1: read_xreg (regidx_to_fin r1) state = EStateM.Result.ok (addi_input.r1_val) state)
  -- must match the AddiInput immediate
  (h_input_imm: addi_input.imm = imm)
  (h_input_rd: addi_input.rd = regidx_to_fin rd)
  (h_input_pc: state.regs.get? Register.PC = .some addi_input.PC)
:
  ( do
    Sail.writeReg Register.nextPC (Sail.BitVec.addInt (← Sail.readReg Register.PC) 4)
    LeanRV32D.Functions.execute (instruction.ITYPE (imm, r1, rd, iop.ADDI))
  ) state =
  let addi_output := execute_ITYPE_addi_pure addi_input
  ( do
    Sail.writeReg Register.nextPC addi_output.nextPC
    match addi_output.rd with
      | .some (rd, rd_val) ⇒ write_xreg rd rd_val
      | .none ⇒ pure ()
    pure (ExecutionResult.Retire_Success ())
  ) state
```

Full Equivalence. Finally, for a given opcode, we prove equivalence between executing that opcode in the RISC-V specification and in OpenVM.

On the one hand, executing an opcode in the Lean RISC-V specification is captured as follows:

```
def execute_instruction
  (instr : instruction)
  (state : PreSail.SequentialState RegisterType Sail.trivialChoiceSource)
:=
  ( do
    Sail.writeReg Register.nextPC (Sail.BitVec.addInt (← Sail.readReg Register.PC) 4)
    LeanRV32D.Functions.execute instr
  ) state
```

This execution involves setting the value of the next program counter (`Register.nextPC`) to the location of the next linear instruction (at `Register.PC` $+ 4$), and then executes the instruction `instr` from current state `state`. This is the essence of how the Lean RISC-V specification executes basic opcodes.

On the other hand, opcode execution in OpenVM is captured only with respect to the content put on the execution and memory buses, as follows, with commentary inlined in the code. The `bus_effect` function processes the execution and memory bus entries and accumulates the assumptions learned from the reads and the state changes resulting from the writes. The assumptions that the function makes are respected by all of the OpenVM chips under the over-arching memory consistency assumption (I1).

```
/-- `bus_effect` captures the effect of execution and memory bus contents
    on the RISC-V state, returning:
    - the hypotheses obtained from the reads; and
    - the state after the writes.

    Assumes that:
    - the execution bus holds two entries, a read and a write; and
    - the memory bus entries have been sorted in chronological order

    - the multiplicities of each entry are in { -1, 0, 1 }; and
    - the address space is either 1 (registers) or 2 (memory) -/
def bus_effect
  (execution_bus : List (Interaction.ExecutionBusEntry FBB))
  (memory_bus : List (Interaction.MemoryBusEntry FBB))
  (state : PreSail.SequentialState RegisterType Sail.trivialChoiceSource)
:
  Prop × (EStateM.Result (Sail.Error exception) (PreSail.SequentialState RegisterType
  ↪  Sail.trivialChoiceSource) ExecutionResult)
:=
  -- Execution bus well-formedness assumption
  if execution_bus.length = 2 ∧ execution_bus[0]!.multiplicity = -1 ∧ execution_bus[1]!.multiplicity = 1
  then
    -- From the first entry on the execution bus, we learn the current program counter
    let initial_result : Prop × (EStateM.Result (Sail.Error exception) (PreSail.SequentialState RegisterType
    ↪  Sail.trivialChoiceSource) ExecutionResult) :=
      ⟨ Sail.readReg Register.PC state = EStateM.Result.ok (register_type_pc_equiv ▸ (BitVec.ofNat 32
      ↪  (execution_bus[0]!.pc).val)) state, EStateM.Result.ok (ExecutionResult.Retire_Success ()) state ⟩
    -- Then we compute the effects of the memory bus
    let post_memory : Prop × (EStateM.Result (Sail.Error exception) (PreSail.SequentialState RegisterType
    ↪  Sail.trivialChoiceSource) ExecutionResult) :=
      List.foldl
        (fun ⟨ cond, result ⟩ entry ⇒
          match result with
          -- Previously got an error: impossible under assumptions
          | .error _ _ ⇒ ⟨ cond, result ⟩
          | .ok _ state ⇒
            -- Read
            if (entry.multiplicity = (-1 : FBB)) then
              -- Register read adds the assumption that the value of the read register is equal
              -- to the 32-bit interpretation of the appropriate memory bus entry values
              (if (entry.as.val = 1) then
                let val := U32.toBV #v[entry.x0, entry.x1, entry.x2, entry.x3]
                let reg_idx := Transpiler.wrap_to_regidx entry.ptr
                ⟨ cond ∧ read_xreg reg_idx state = EStateM.Result.ok val state, result ⟩
              -- Memory read adds the assumption that the four values in memory starting from
              -- the given pointer are equal to the appropriate memory bus entry values
              else if (entry.as.val = 2) then
                ⟨ cond ∧
                  state.mem[entry.ptr.toNat]? = .some entry.x0 ∧
                  state.mem[entry.ptr.toNat + 1]? = .some entry.x1 ∧
                  state.mem[entry.ptr.toNat + 2]? = .some entry.x2 ∧
                  state.mem[entry.ptr.toNat + 3]? = .some entry.x3
                , result ⟩
              -- Neither a register nor a memory read: impossible under assumptions
              else ⟨ cond, EStateM.Result.error Sail.Error.Unreachable state ⟩)
```

```
          -- Write
          else if (entry.multiplicity = (1 : FBB)) then
            (if (entry.as.val = 1) then
              -- Register write writes the four values from the memory bus entry
              -- to the given register as a 32-bit value
              if h : Transpiler.wrap_to_regidx entry.ptr = 0 then
                ⟨ cond, result ⟩
              else
                let val := U32.toBV #v[entry.x0, entry.x1, entry.x2, entry.x3]
                let reg_idx := ⟨ (Transpiler.wrap_to_regidx entry.ptr).val, by simp; omega ⟩
                ⟨ cond, EStateM.Result.map
                        (fun x ↦ ExecutionResult.Retire_Success ())
                          (write_xreg reg_idx val state) ⟩
            -- Memory write writes the four values from the memory bus entry
            -- to memory starting from the given pointer
            else if (entry.as.val = 2) then
              ⟨
                cond
              , EStateM.Result.ok (ExecutionResult.Retire_Success ()) {
                  state with mem :=
                    ((((state.mem.insert entry.ptr.toNat entry.x0
                    ).insert (entry.ptr.toNat + 1) entry.x1
                    ).insert (entry.ptr.toNat + 2) entry.x2
                    ).insert (entry.ptr.toNat + 3) entry.x3)
                }⟩
            -- Address space not 1 or 2 : impossible under assumptions
            else ⟨ cond, EStateM.Result.error Sail.Error.Unreachable state ⟩)
          -- Multiplicity zero: no effect
          else if (entry.multiplicity = (0 : FBB)) then ⟨ cond, result ⟩
          -- Neither a register nor a memory read nor a no-effect: impossible under assumptions
          else ⟨ cond, EStateM.Result.error Sail.Error.Unreachable state ⟩
        )
        initial_result
        memory_bus
    match post_memory.2 with
    -- the second entry on the execution bus sets the next program counter
    | .ok _ state ⇒ ⟨ post_memory.1,
                      EStateM.Result.map
                        (fun x ↦ ExecutionResult.Retire_Success ())
                        (Sail.writeReg
                          Register.nextPC
                          (register_type_pc_equiv ▸ (BitVec.ofNat 32 (execution_bus[1]!.pc).val))
                          state) ⟩
    -- Memory pass returned an error: impossible under assumptions
    | _ ⇒ post_memory
  else
    -- Malformed execution bus: impossible under assumptions
    ⟨ True, EStateM.Result.error Sail.Error.Unreachable state ⟩
```

Specifically, for the ADD and ADDI opcodes, the corresponding theorems, which can be found in the Equivalence.lean file, are formulated as follows:

```
variable
  -- We are working with a valid row of the base ALU air
  (air : Valid_VmAirWrapper_alu FBB ExtF)
  (row : ℕ)
  (h_row : row ≤ air.last_row)
  (h_is_valid : air.core.is_valid row 0 = 1)
  -- for which the constraints, bus-related axioms, bus-related well-formedness properties,
  -- and the information about the state learned from the reads on the execution and memory bus hold
  (h_constraints : allHold air row h_row)
  (h_bus_axioms : axiomsPerRow air row)
  (h_bus_wellformedness : wf_propertiesToAssumePerRow air row)
  (h_bus : (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).1)
```

```
theorem equiv_ADD
  (h_opcode : (air.core.ctx row 0).instruction.opcode = 512) -- ADD OpenVM opcode
  (h_imm : air.adapter.rs2_as row 0 = 1)                     -- Non-immediate version
:
  -- If we take the destination and operand registers from the program bus,
  let rd_ptr := (_programBus_row air row)[0]!.xa
  let rs1_ptr := (_programBus_row air row)[0]!.xb
  let rs2_ptr := (_programBus_row air row)[0]!.xc
  -- adjust them for the Lean RISC-V specification,
  let r1 : regidx := ⟨ (Transpiler.wrap_to_regidx rs1_ptr).val, by simp ⟩
  let r2 : regidx := ⟨ (Transpiler.wrap_to_regidx rs2_ptr).val, by simp ⟩
  let rd : regidx := ⟨ (Transpiler.wrap_to_regidx rd_ptr).val, by simp ⟩
  -- and construct the appropriate ADD instruction
  let instr : instruction := instruction.RTYPE (r2, r1, rd, rop.ADD)
  -- then, executing this instruction in the Lean RISC-V specification
  execute_instruction instr state =
  -- is the same as executing it in OpenVM
  (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).2

theorem equiv_ADDI
  (h_opcode : (air.core.ctx row 0).instruction.opcode = 512)
  (h_imm : air.adapter.rs2_as row 0 = 0)
:
  -- If we take the destination, first operand register, and the immediate from the program bus,
  let rd_ptr := (_programBus_row air row)[0]!.xa
  let rs1_ptr := (_programBus_row air row)[0]!.xb
  let imm := (_programBus_row air row)[0]!.xc
  -- adjust them for the Lean RISC-V specification,
  let r1 : regidx := ⟨ (Transpiler.wrap_to_regidx rs1_ptr).val, by simp ⟩
  let imm : BitVec 12 := BitVec.ofNat 12 imm.toNat
  let rd : regidx := ⟨ (Transpiler.wrap_to_regidx rd_ptr).val, by simp ⟩
  -- and construct the appropriate ADDI instruction
  let instr : instruction := instruction.ITYPE (imm, r1, rd, iop.ADDI)
  -- then, executing this instruction in the Lean RISC-V specification
  execute_instruction instr state =
  -- is the same as executing it in OpenVM
  (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).2
```

To conclude this part, we note that the ALU-opcode proofs do not use assumptions S1-S4 and A1-A4 (cf. §1.1).

3.2 Example 2: Control-flow opcodes

The verification process of control-flow opcodes is, for the most part, entirely analogous to that of ALU-related opcodes. The main point of divergence is that control-flow opcodes may affect the program counter in more complex ways and interact more with the RISC-V state. We illustrate this on the BEQ example, jumping directly into the last steps of the verification (as the previous steps do not have notable differences), in which we link the proven pure specification to the structural part of the Lean RISC-V opcode specification.

The pure specification of BEQ captures the expected branching, and is as follows:

```
include
  row_valid constraints axioms propertiesToAssume
in
theorem spec_BEQ_BNE_pc :
  -- BEQ case
  (air.core.expected_opcode row 0 = 544 →
    BitVec.ofNat 32 (air.to_pc row 0) =
      if U32.toBV #v[(air.core.a_0 row 0).val, (air.core.a_1 row 0).val,
                     (air.core.a_2 row 0).val, (air.core.a_3 row 0).val] ==
         U32.toBV #v[(air.core.b_0 row 0).val, (air.core.b_1 row 0).val,
                     (air.core.b_2 row 0).val, (air.core.b_3 row 0).val]
    -- operands equal, jump to pc + immediate
    then BitVec.ofNat 32 (air.adapter.from_state.pc row 0).val +
           BitVec.signExtend 32 (BitVec.ofInt 13 (BabyBear.toInt (air.core.imm row 0)))
```

```
    -- operands not equal, proceed to next instruction
    else BitVec.ofNat 32 (air.adapter.from_state.pc row 0).val + 4#32) ∧
  ...
```

The structures that capture the input and the output for BEQ are defined as follows:

```
structure BeqInput where
  imm.   : BitVec 13 -- 13-bit immediate value
  r1_val : BitVec 32 -- first operand of BEQ
  r2_val : BitVec 32 -- second operand of BEQ
  PC     : BitVec 32 -- current program counter
  misa   : BitVec 32 -- misa register of the RISC-V machine

structure BeqOutput where
  nextPC  : BitVec 32 -- next program counter
  success : Bool      -- success indicator
  throws  : Bool      -- exception indicator

def execute_BEQ_pure (input : BeqInput) : BeqOutput :=
  -- non-success checks do not need to be performed if operands are not equal
  let skip := !(input.r1_val == input.r2_val)
  -- BEQ throws if the operands are equal and next PC is not divisible by 2
  let throws := !skip && BitVec.ofBool (input.PC + BitVec.signExtend 32 input.imm)[0] == 1#1
  -- BEQ fails either if it throws or if the operands are equal and the next PC is not divisible by 4
  -- but the misa register indicates that compression (2-byte aligned PCs) is not allowed
  let fails := throws ||
               (!skip &&
                BitVec.ofBool (input.PC + BitVec.signExtend 32 input.imm)[1] == 1#1 && !input.misa[2])
  {
    -- The program counter moves to either
    nextPC := if skip || fails
      -- the next instruction, if operands are not equal or BEQ fails
      then (input.PC + 4)
      -- or to the sum of the program counter and the immediate value
      else (input.PC + BitVec.signExtend 32 input.imm)
    success := !fails -- BEQ succeeds if it does not fail
    throws := throws
    : BeqOutput
  }
```

noting how we need to break down and make explicit the control flow into the success, failure, and exception cases. Importantly, as the transpilation process ensures that all program counters used in the execution are divisible by four (cf. 2.3.1), the throw and failure cases do not get triggered in OpenVM.

Finally, and analogously to the ADD/ADDI, we prove the equivalence between the RISC-V and OpenVM executions of BEQ, as follows:

```
theorem equiv_BEQ
  (h_opcode : air.core.expected_opcode row 0 = 544)
:
  let rs1_ptr := (_programBus_row air row)[0]!.xa
  let rs2_ptr := (_programBus_row air row)[0]!.xb
  let imm := (_programBus_row air row)[0]!.xc
  let r1 : regidx := ⟨ (Transpiler.wrap_to_regidx rs1_ptr).val, by simp ⟩
  let r2 : regidx := ⟨ (Transpiler.wrap_to_regidx rs2_ptr).val, by simp ⟩
  let imm : BitVec 13 := BitVec.ofInt 13 (BabyBear.toInt imm)
  let instr : instruction := instruction.BTYPE (imm, r2, r1, bop.BEQ)
  execute_instruction instr state =
  (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).2
```

To conclude this part, we note that the control-flow-opcode proofs do not use assumptions S3-S4 and A2-A3 (cf. §1.1).

3.3 Example 3: Memory-manipulating opcodes

The verification of memory-manipulating opcodes is different from that of the rest of the opcodes, as it requires reasoning about RISC-V memory reads and writes. We illustrate this process using the LW and SW opcodes; the remaining ones are analogous.

3.3.1 The LW opcode

As for the previous examples, we start by describing the input and the output of LW:

```
structure LwInput where
  r1  : BitVec 5  -- register of first operand
  imm : BitVec 12 -- 12-bit immediate
  rd  : BitVec 5  -- intended output register
  r1_val : BitVec 32 -- value stored in register of first operand
  PC     : BitVec 32 -- current program counter
  -- four bytes, from lowest to highest, that are loaded from the memory
  data0 : BitVec 8
  data1 : BitVec 8
  data2 : BitVec 8
  data3 : BitVec 8
```

```
structure LwOutput where
  nextPC : BitVec 32                       -- next program counter
  rd : Option (Finset.Icc 1 31 × BitVec 32) -- result of executing the LW opcode
```

Next, we capture the intended (pure) behaviour of LW as follows:

```
def execute_LOADW_pure (input : LwInput) : LwOutput := {
  nextPC := input.PC + 4#32 -- the next program counter is always the current one + 4
  rd := if h: input.rd = 0
    then .none              -- no result, if the intended output register equals zero
    else .some (            -- otherwise, the returned 32-bit value is obtained by
      ⟨                     -- concatenating the four bytes of data, highest to lowest
        input.rd.toNat,
        range input.rd h
      ⟩,
      input.data3 ++ input.data2 ++ input.data1 ++ input.data0
    ) : LwOutput
}
```

Next, and in contrast to non-memory-related opcodes, we define the LW-specific assumptions that we must take into consideration for this proof:

```
def lw_state_assumptions
  (i : LwInput)
  (state : PreSail.SequentialState RegisterType Sail.trivialChoiceSource) : Prop :=
  -- Connecting the registers to the pure data
  state.regs.get? Register.PC = .some i.PC ∧
  LeanRV32D.Functions.rX_bits (regidx.Regidx i.r1) state = EStateM.Result.ok i.r1_val state ∧
  -- Connecting the memory to the pure data
  state.mem[i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat]? = .some i.data0 ∧
  state.mem[i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat + 1]? = .some i.data1 ∧
  state.mem[i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat + 2]? = .some i.data2 ∧
  state.mem[i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat + 3]? = .some i.data3 ∧
  -- Memory address is not outside address space
  i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat < OpenVM_address_space_size ∧
  -- Memory address alignment
  (4 : ℤ) ∣ i.r1_val.toNat + (BitVec.signExtend 32 i.imm).toNat
```

Then, we prove that, given input data and under the general RISC-V assumptions and LW-specific assumptions, executing the LW opcode using the Lean RISC-V specification is equivalent to appropriately updating the RISC-V state using the output data, as follows:

```
lemma execute_LOADW_pure_equiv
  (input : LwInput)
  (risc_v_assumptions : RISC_V_assumptions state mstatus pmaRegion misa mseccfg)
  (h_opcode_assumptions : lw_state_assumptions input state)
:
  ( do
      Sail.writeReg Register.nextPC (Sail.BitVec.addInt (← Sail.readReg Register.PC) 4)
      LeanRV32D.Functions.execute (
        instruction.LOAD(input.imm, regidx.Regidx input.r1, regidx.Regidx input.rd, true, 4)
      )
  ) state =
  let output := execute_LOADW_pure input
  ( do
    -- Update the nextPC register to be the nextPC of the output
    Sail.writeReg Register.nextPC output.nextPC
    match output.rd with
      -- Update the rd register with the read value, noting that rd_val = input.data3 ++ input.data2 ++
      -- input.data1 ++ input.data0, as per the definition of execute_LOADW_pure
      | .some (rd, rd_val) ⇒ write_xreg rd rd_val
      | .none ⇒ pure ()
    pure (ExecutionResult.Retire_Success ())
  ) state
```

Finally, we use this equivalence proof in the proof of the execution equivalence theorem, which is formulated in the same way as for other opcodes:

```
theorem equiv_LOADW
  (h_opcode : air.core.expected_opcode row 0 = 528)
:
  let rd_ptr := (_programBus_row air row)[0]!.xa
  let rs1_ptr := (_programBus_row air row)[0]!.xb
  let imm := (_programBus_row air row)[0]!.xc
  let rd : regidx := ⟨ (Transpiler.wrap_to_regidx rd_ptr).val, by simp ⟩
  let rs1 : regidx := ⟨ (Transpiler.wrap_to_regidx rs1_ptr).val, by simp ⟩
  let imm : BitVec 12 := BitVec.ofNat 12 imm
  let instr : instruction := .LOAD (imm, rs1, rd, true, 4)
  execute_instruction instr state =
  (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).2
```

noting that all of the LW-specific assumptions made in the pure equivalence proof hold given the chip constraints and execution and memory bus reads.

3.3.2 The SW opcode

The definitions of the input and output of SW, as well as intended pure behaviour are as follows, with the difference with respect to other opcodes being that the value to be stored is explicitly deconstructed in the output from the value read in the `rs2` register:

```
structure SwInput where
  -- operands
  r1     : BitVec 5  -- register of first operand
  imm    : BitVec 12 -- 12-bit immediate
  r2     : BitVec 5  -- register of value to be stored
  r1_val : BitVec 32 -- value stored in register of first operand
  r2_val : BitVec 32 -- value to be stored
  PC     : BitVec 32 -- current program counter
```

```
structure SwOutput where
  nextPC : BitVec 32 -- next program counter
   -- four address-byte pairs, with the intended meaning that
   -- the given byte is supposed to be stored at the given address
  data0 : ℕ × BitVec 8
  data1 : ℕ × BitVec 8
```

```
  data2 : ℕ × BitVec 8
  data3 : ℕ × BitVec 8
```

```
def execute_STOREW_pure (i : SwInput) : SwOutput := {
  nextPC := input.PC + 4#32 -- the next program counter is always the current one + 4
  -- The four pieces of data to be stored, obtained by extracting
  -- the four bytes of r2_val from the lowest to the highest
  data0 := ((i.r1_val + BitVec.signExtend 32 i.imm).toNat,     BitVec.extractLsb  7  0 i.r2_val)
  data1 := ((i.r1_val + BitVec.signExtend 32 i.imm).toNat + 1, BitVec.extractLsb 15  8 i.r2_val)
  data2 := ((i.r1_val + BitVec.signExtend 32 i.imm).toNat + 2, BitVec.extractLsb 23 16 i.r2_val)
  data3 := ((i.r1_val + BitVec.signExtend 32 i.imm).toNat + 3, BitVec.extractLsb 31 24 i.r2_val)
  : SwOutput
}
```

Noting that the assumptions are defined in the same way as for LW, we next prove that, given input data and under the general memory assumptions and SW-specific assumptions, executing the SW opcode using the Lean RISC-V specification is equivalent to appropriately updating the RISC-V state using the output data, as follows:

```
lemma execute_STOREW_pure_equiv
  (input : SwInput)
  (h_assumptions : Local.Assumptions.general_memory_assumptions state)
  (h_sw_assumptions : sw_state_assumptions input state)
:
  ( do
      Sail.writeReg Register.nextPC (Sail.BitVec.addInt (← Sail.readReg Register.PC) 4)
      LeanRV32D.Functions.execute (
        instruction.STORE (input.imm, regidx.Regidx input.r2, regidx.Regidx input.r1, 4)
      )
  ) state =
  let output := execute_STOREW_pure input
  ( do
    -- Update the nextPC register to be the nextPC of the output
    Sail.writeReg Register.nextPC output.nextPC
    set (
      let s := (← get) -- Get current state
      { s with mem :=  -- and update only the memory at the locations and with the values given by the output
          ((((s.mem.insert output.data0.1 output.data0.2
          ).insert output.data1.1 output.data1.2)
          ).insert output.data2.1 output.data2.2
          ).insert output.data3.1 output.data3.2
        : PreSail.SequentialState RegisterType Sail.trivialChoiceSource }
    )
    pure (ExecutionResult.Retire_Success ())
  ) state
```

observing that we had to explicitly update the RISC-V memory as part of the lemma statement, and that we state that the only modifications happen to the memory whereas all of the other state components stay the same. As for LW, we use this pure equivalence in the proof of the execution equivalence theorem:

```
theorem equiv_LOADW
  (h_opcode : air.core.expected_opcode row 0 = 528)
:
  let rd_ptr := (_programBus_row air row)[0]!.xa
  let rs1_ptr := (_programBus_row air row)[0]!.xb
  let imm := (_programBus_row air row)[0]!.xc
  let rd : regidx := ⟨ (Transpiler.wrap_to_regidx rd_ptr).val, by simp ⟩
  let rs1 : regidx := ⟨ (Transpiler.wrap_to_regidx rs1_ptr).val, by simp ⟩
  let imm : BitVec 12 := BitVec.ofNat 12 imm
  let instr : instruction := .LOAD (imm, rs1, rd, true, 4)
  execute_instruction instr state =
  (bus_effect (_executionBus_row air row) (_memoryBus_row air row) state).2
```

noting once more that all of the SW-specific assumptions made in the pure equivalence proof hold given the chip constraints and execution and memory bus reads.

4 Execution and Memory Consistency

We prove the memory consistency of the RV32IM-related chips of OpenVM in MemoryConsistency.lean. First, we define the general properties that a chip needs to satisfy:

```
/-- A well formed chip with respect to a measure μ provides:
  - pairs of read-write entries put on the execution bus
  - pairs of read-write entries put on the memory bus
  - the proof that each memory bus read-write entry is for the same pointer
  - a proof that the execution bus entries are rising
  - a proof that the memory bus entries are rising -/
class WFConstraints (α : Type) (air : α) (μ : List FBB → ℕ) where
  execution_bus_entries : List (List FBB × List FBB)
  memory_bus_entries : List (List FBB × List FBB)
  memory_bus_entries_wf : ∀ a b, (a, b) ∈ memory_bus_entries → (a[1]!.val = b[1]!.val)
  rising_pairs_on_execution_bus : ∀ a b, (a, b) ∈ execution_bus_entries → μ a < μ b
  rising_pairs_on_memory_bus : ∀ a b, (a, b) ∈ memory_bus_entries → μ a < μ b
```

and then provide an instance of this class for every RV32IM-chip (Auipc, BaseALU, BranchEqual, BranchLessThan, DivRem, JalLui, JalR, LoadSignExtend, LoadStore, Lt, Mul, Mulh, and Shift) where the joint measure function on bus entries, μ, is defined as the timestamp of the execution and memory bus entries, as follows:

```
-- The actual measure takes the timestamp from the execution and the memory bus.
-- This happens to be the last element of the associated data for both of these buses.
def μ (data : List FBB) : ℕ :=
  match data with
  | [] ⇒ 0
  | _  ⇒ data.reverse.head!.val
```

Next, we define a general well-formed (WF) chip with respect to a given measure μ as follows:

```
/-- A general well-formed chip -/
structure WFChip (μ : List FBB → ℕ) where
  ChipType : Type
  chip : ChipType
  [inst_wf : WFConstraints ChipType chip μ]
```

and the full execution/memory bus of a given collection of WF-chips simply as the concatenation of all of the execution/memory buses of the individual chips:

```
def execution_bus (chips : List (WFChip μ)) : List (List FBB × List FBB)
 := List.flatMap (fun chip ↦ chip.inst_wf.execution_bus_entries) chips

def memory_bus (chips : List (WFChip μ)) : List (List FBB × List FBB)
 := List.flatMap (fun chip ↦ chip.inst_wf.memory_bus_entries) chips
```

and also isolate the subset of the memory bus relevant to a given pointer, `ptr`, as follows:

```
def memory_bus_per_ptr (chips : List (WFChip μ)) (ptr : Fin OpenVM_address_space_size)
: List (List FBB × List FBB) :=
  List.filter (fun (a, _) ↦ a[1]! = ptr.val) (memory_bus chips)
```

We then prove that the execution bus and the memory bus per-pointer are rising buses:

```
lemma execution_bus_is_rising_bus
  (chips : List (WFChip μ))
: ∀ entry ∈ (execution_bus chips), μ entry.1 < μ entry.2

lemma memory_bus_per_ptr_is_rising_bus
  (chips : List (WFChip μ))
  (ptr : Fin OpenVM_address_space_size)
: ∀ entry ∈ (memory_bus_per_ptr chips ptr), μ entry.1 < μ entry.2
```

which allows us to, assuming that the combined number of interactions on the balanced execution bus and the unbalanced memory bus is less than BB_prime (which holds from (E4) and (E5)) and the balancing of the execution bus and of the memory bus per-pointer (which hold given the properties of the backend), apply the rising bus theorem to obtain the following two statements:

```
variable
  -- The chips that participate in the execution
  (chips : List (WFChip μ))
  -- The number of overall lookups on the execution and memory buses,
  -- balancing included, does not exceed BabyBear (guaranteed by the verifier)
  (bus_length : 2 * (execution_bus chips).length +
                2 * (memory_bus chips).length + 2 < BB_prime)

include bus_length in
theorem execution_bus_consistency
  -- The execution is not empty
  (execution_not_empty : 0 < (execution_bus chips).length)
  -- When the execution is not empty, execution bus is balanced by single balancers
  (balanced_execution_bus :
     InteractionList.is_balanced
       ([((1 : FBB), lbound)] ++
        (rising_bus μ (execution_bus chips) (execution_bus_is_rising_bus chips)) ++
        [((-1 : FBB), rbound)]))
:
  -- The execution bus is accessed in a consistent manner
  ∃ xs, (execution_bus chips).Perm (List.zip (lb :: xs) (xs ++ [ub])) ∧
    List.Pairwise (fun x₁ x₂ ↦ μ x₁ < μ x₂) (lb :: xs ++ [ub])

include bus_length in
theorem memory_bus_consistency_per_ptr
  (ptr : Fin OpenVM_address_space_size)
  -- Memory bus per-pointer is either empty or is balanced by single balancers per-pointer
  (balanced_memory_bus_per_ptr :
    memory_bus_per_ptr chips ptr = [] ∨
    (0 < (memory_bus_per_ptr chips ptr).length ∧
      InteractionList.is_balanced
        ([((1 : FBB), lbm)] ++
         (rising_bus μ (memory_bus_per_ptr chips ptr) (memory_bus_per_ptr_is_rising_bus chips ptr)) ++
         [((-1 : FBB), ubm)])))
:
  -- The memory at `ptr` is either not accessed during the execution
  (memory_bus_per_ptr chips ptr = []) ∨
    -- or is accessed in a consistent manner
    ∃ xs, (memory_bus_per_ptr chips ptr).Perm (List.zip (lb :: xs) (xs ++ [ub])) ∧
      List.Pairwise (fun x₁ x₂ ↦ μ x₁ < μ x₂) (lb :: xs ++ [ub])
```

What do these two theorems mean? As discussed in §2.2.2, the rising bus balancing theorem, when the measure corresponds to time, means that the bus read-write entry pairs $x_1 = (r_1, w_1), \ldots, x_n = (r_n, w_n)$, can be rearranged so that:

1. the reads, $r_1, \ldots, r_n$, are ordered chronologically;
2. the writes, $w_1, \ldots, w_n$, are ordered chronologically;
3. the read r_{i+1} balances out the write w_i, for $i \in \{1, ..., n-1\}$, meaning that all of these reads effectively read from the latest write;
4. the read r_1 balances out the initial write w_0, which, given (3), means that all reads read from the latest write.

In the context of the execution bus, this means that we can establish a chronological sequence of instructions. In the context of the memory bus, this means that, for every pointer ptr that is touched by the execution, the associated memory accesses are consistent, and that, therefore, the memory is treated overall in a consistent manner.

5 Disclaimer

This Report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the tasks outlined in the Executive Summary. The content of this Report is intended to provide an analysis of the formal verification process undertaken, and the results obtained, based on the information and data available to Nethermind up to the time of completion. The formal verification process conducted by the team of engineers was based on the specific parameters, methodologies, and data available at the time of the analysis. Any changes in the underlying parameters, methodologies, or data may lead to different results.

Nethermind has provided the review and this Report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to any other areas outside the scope of work agreed with you in our Agreement. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the Report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.