OpenVM Circuit Architecture

1 Preliminaries

1.1 Notations

In this document, we adopt the following notations:

- **Base Field** $\mathbb{F}$: The base field is a prime finite field.
- **Extension Field** $\mathbb{F}_{\mathbf{ext}}$: We denote by $\mathbb{F}_{\text{ext}}$ the extension field of the base field $\mathbb{F}$. The extension field is used for Fiat-Shamir randomness and must have sufficient bits for target security.
- **Subgroup** H: Given n, the interpolation domain is represented by the subgroup H, which is a subgroup of the base field $\mathbb{F}$ of size 2^n. The generator of the subgroup H is denoted by ω, and the elements of H are given by $\omega^0, \omega^1, \ldots, \omega^{2^n-1}$.

1.2 AIR Trace

Definition 1 (AIR Trace)**.** *An Algebraic Intermediate Representation (AIR) trace is a matrix, denoted by* $\mathbf{Tr}$*, used to represent a computation over time. The AIR trace matrix captures the values of different variables at each step in the computation, allowing algebraic constraints to be enforced over the trace. Each row of the matrix is denoted by* $\mathbf{Tr}[i]$*, and each entry within a row by* $\mathbf{Tr}[i][j]$*, where* i *indicates the row and* j *indicates the column.*

- ***Height of the AIR Trace****: The height of the AIR trace is denoted by* 2^n*, representing the number of rows in the matrix. Each row corresponds to a single step in the computation, with the total number of steps being* 2^n*.*
- ***Width of the AIR Trace****: The width of the AIR trace is denoted by* m*, representing the number of columns.*
- ***Cell Values****: Each cell in the AIR trace matrix contains an element from the base field* $\mathbb{F}$*.*
- ***Extension Field Representation****: If an element from the extension field* $\mathbb{F}_{ext}$ *of degree* D *is used, it is represented by* D *continuous cells across a single row. These* D *cells store the coefficients of the extension field element, where each coefficient is an element of the base field* $\mathbb{F}$*.*

Remark 1.1. *In this document, we will frequently work with multiple AIR traces, each potentially with different heights and widths, depending on the specific computation and constraints being represented. We use* $k \in [1, N]$ *to denote the* k*-th AIR trace, with height* 2^{n_k} *and width* m_k*.*

1.3 Virtual Columns and Constraints

In the AIR framework, each column of the trace matrix can be represented as a univariate polynomial. More specifically:

Each column of the AIR trace, say $\mathbf{Tr}[i]$ with entries $\mathbf{Tr}[i][0], \mathbf{Tr}[i][1], \ldots, \mathbf{Tr}[i][2^n - 1]$, is represented as a univariate polynomial $\mathbf{Tr}[i](x)$, where x ranges over the points $\omega^0, \omega^1, \ldots, \omega^{2^n-1}$ in the interpolation domain. Here, ω is the generator of the subgroup $H \subseteq \mathbb{F}$. Thus, each column can be viewed as a polynomial evaluated at distinct points in the domain.

Definition 2 (Virtual Column)**.** *A virtual column denoted by $\mathbf{P}(y_1, \ldots, y_m)$, is a multivariate polynomial in which each variable represents either a univariate polynomial corresponding to a column in the AIR trace (as defined above), or a rotation of that univariate polynomial.*

The rotation of a polynomial $\mathbf{Tr}[i](x)$ is defined as $\mathbf{Tr}[i]^{\mathsf{rot}}(x) = \mathbf{Tr}[i](x \cdot \omega)$, where ω is the generator of the subgroup H. In explicit column notation, this rotation means that $\mathbf{Tr}[i]^{\mathsf{rot}}[j] = \mathbf{Tr}[i][j+1]$, shifting the entries by one position in the trace.

For example, suppose we have three columns $\mathbf{Tr}[i_1]$, $\mathbf{Tr}[i_2]$, and $\mathbf{Tr}[i_3]$ in the trace. A virtual column $\mathbf{P}(y_1, y_2, y_3)$ might be defined as follows:

$$\mathbf{P}(y_1, y_2, y_3) = y_1 + y_2 - y_3,$$

where $y_1 = \mathbf{Tr}[i_1](x)$, $y_2 = \mathbf{Tr}[i_2](x)$, and $y_3 = \mathbf{Tr}[i_3](x)$ represent the univariate polynomials of these columns.

Definition 3 (Degree of Virtual Columns)**.** *The degree of each virtual polynomial is measured in terms of its total degree, which is the highest degree of any term when considering all variables. In the above example, the total degree of this polynomial is* 1*.*

Definition 4 (Constraint Polynomials)**.** *A constraint polynomial enforces a relation or constraint over one or more columns of the AIR trace. These polynomials are designed to ensure that each step of the computation adheres to the algebraic rules of the system being verified. Most constraints involve multiple columns.*

Without loss of generality, all constraints can be represented by virtual polynomials, which are multivariate polynomials designed to evaluate to zero over a specific subset $D \subseteq H$ of the interpolation domain.

For example, in our previous example with the virtual polynomial $\mathbf{P}(y_1, y_2, y_3) = \mathbf{Tr}[i_1](x) + \mathbf{Tr}[i_2](x) - \mathbf{Tr}[i_3](x)$, enforcing $\mathbf{P}$ to be zero over the entire domain $x \in H$ ensures that $\mathbf{Tr}[i_3]$ is always the sum of the other two columns, $\mathbf{Tr}[i_1]$ and $\mathbf{Tr}[i_2]$, across all steps of the computation.

1.4 Multiset over Finite Field and LogUp Identity

Definition 5 (Finite Field Multiset)**.** *A finite field multiset over a finite field $\mathbb{F}$ is a function $M : \mathbb{F} \to \mathbb{F}$, where $M(a)$ represents the multiplicity of each element $a \in \mathbb{F}$ in the multiset.*

In this context, multiplicities are defined within the finite field $\mathbb{F}$ rather than the natural numbers $\mathbb{N}$. For a field $\mathbb{F}$ with characteristic p, such as $p = 15 \cdot 2^{27} + 1$, a multiplicity of $a \cdot p$ (where $a \in \mathbb{F}$) is equivalent to a multiplicity of 0*, even though $a \cdot p \neq 0$ as a natural number. Thus, multiplicity values are considered modulo p, following the arithmetic rules of the field $\mathbb{F}$.*

The cardinality of a finite field multiset M, denoted $|M|$, is defined as the size of its support, which is the number of elements in $\mathbb{F}$ with non-zero multiplicity, i.e., $|M| = |\{a \in \mathbb{F} \mid M(a) \neq 0\}|$.

Definition 6 (Empty Finite Field Multiset)**.** *An empty finite field multiset over a finite field $\mathbb{F}$ is a finite field multiset $M : \mathbb{F} \to \mathbb{F}$ such that $M(a) = 0$ for all $a \in \mathbb{F}$. In other words, M is the* 0*-identity function.*

Definition 7 (LogUp α-Fraction)**.** *Over any given finite field $\mathbb{F}$, a finite field multiset $M : \mathbb{F} \to \mathbb{F}$, and any element $a \in \mathbb{F}$, its LogUp α-Fraction takes the form*

$$\frac{M(a)}{a + \alpha}.$$

Here, $\alpha \in \mathbb{F}$ is an indeterminate variable.

Theorem 1 (LogUp Identity)**.** *Let $M : \mathbb{F} \to \mathbb{F}$ be a finite field multiset over a finite field $\mathbb{F}$. If the sum of all α-fractions of M over each element $a \in \mathbb{F}$ equals zero, that is,*

$$\sum_{a \in \mathbb{F}} \frac{M(a)}{a + \alpha} = 0,$$

where α is an indeterminate in $\mathbb{F}$, then $M(a) = 0$ for all $a \in \mathbb{F}$. Thus, M is an empty finite field multiset.

Proof. We begin with the assumption that

$$\sum_{a \in \mathbb{F}} \frac{M(a)}{a + \alpha} = 0,$$

where α is an indeterminate in $\mathbb{F}$ and $M(a)$ represents the multiplicity of each element $a \in \mathbb{F}$.

Since α is indeterminate, the expression $\sum_{a \in \mathbb{F}} \frac{M(a)}{a+\alpha}$ must be identically zero as a function of α. For this sum to be identically zero for all values of α, each term $\frac{M(a)}{a+\alpha}$ in the sum must independently contribute zero.

Suppose, for contradiction, that there exists an element $b \in \mathbb{F}$ such that $M(b) \neq 0$. Then the term $\frac{M(b)}{b+\alpha}$ would introduce a non-zero contribution to the sum for values of $\alpha \neq -b$, which would prevent the sum from being identically zero. This contradicts our assumption that the entire sum is zero for all values of α.

Therefore, we conclude that $M(a) = 0$ for every $a \in \mathbb{F}$. This implies that M is an empty finite field multiset. □

In this paper, we also consider multisets over the ℓ-dimensional vector space $\mathbb{F}^\ell$:

Definition 8 (Finite Field Multiset over $\mathbb{F}^\ell$). *An ℓ-dimensional finite field multiset over $\mathbb{F}^\ell$ is a function $M : \mathbb{F}^\ell \to \mathbb{F}$, where $M(\mathbf{a})$ represents the multiplicity of each vector $\mathbf{a} \in \mathbb{F}^\ell$.*

We deduce another LogUp identity for ℓ-dimensional finite field multiset using a well-known technique called universal hashing:

Definition 9 (Universal Hashing). *Let $\mathbb{F}$ be a finite field, and let $S \subseteq \mathbb{F}^\ell$ be a set of ℓ-dimensional vectors. A universal hash function $h^\beta : \mathbb{F}^\ell \to \mathbb{F}$ maps each vector $\mathbf{a} = (a_1, a_2, \ldots, a_\ell) \in \mathbb{F}^\ell$ to a single field element in $\mathbb{F}$ using powers of a randomly chosen constant $\beta \in \mathbb{F}$ as follows:*

$$h^\beta(\mathbf{a}) = \sum_{i=1}^{\ell} \beta^{i-1} a_i.$$

Claim 1. *It's well known that for any two distinct vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}^\ell$, the probability (over the random choice of β) that $h^\beta(\mathbf{a}) = h^\beta(\mathbf{b})$ is at most $\frac{\ell}{|\mathbb{F}|}$.*

Let $M : \mathbb{F}^\ell \to \mathbb{F}$ be a finite field multiset over the ℓ-dimensional vector space $\mathbb{F}^\ell$. Using the universal hash function $h^\beta : \mathbb{F}^\ell \to \mathbb{F}$, we can construct a one-dimensional multiset $M' : \mathbb{F} \to \mathbb{F}$ such that M and M' are isomorphic with respect to their multiplicities, specifically over the elements of M with non-zero multiplicity.

Theorem 2. *Define $M' : \mathbb{F} \to \mathbb{F}$ where $M'(h^\beta(\mathbf{a})) = M(\mathbf{a})$ only for those $\mathbf{a} \in \mathbb{F}^\ell$ where $M(\mathbf{a}) \neq 0$ and set $M'(b) = 0$ for all other $b \in \mathbb{F}$. Then with probability at least $1 - \binom{|M|}{2} \cdot \frac{\ell}{|\mathbb{F}|}$, M' is well-defined function, meaning no element in $\mathbb{F}$ is assigned multiple values in M'.*

Proof. To ensure M' is well-defined, we need h^β to be injective on the support of M, i.e., the subset of $\mathbb{F}^\ell$ where $M(\mathbf{a}) \neq 0$. Let $|M|$ denote the cardinality of this support. Consider two distinct vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}^\ell$ with non-zero multiplicities. By claim 1, the probability that $h^\beta(\mathbf{a}) = h^\beta(\mathbf{b})$ for any given pair is at most $\frac{\ell}{|\mathbb{F}|}$. Using the union bound over all possible pairs of distinct vectors in the support of M, the probability that h^β remains injective is at least

$$1 - \binom{|M|}{2} \cdot \frac{\ell}{|\mathbb{F}|}.$$

□

Combining theorem 1 with theorem 2, we obtain the following theorem:

Theorem 3 (LogUp Identity for ℓ-dimensional Multiset). *Let $M : \mathbb{F}^\ell \to \mathbb{F}$ be a finite field multiset over the ℓ-dimensional vector space $\mathbb{F}^\ell$, and let $M' : \mathbb{F} \to \mathbb{F}$ be the one-dimensional multiset constructed using the* *universal hash function h^β as aforementioned. Then, with probability at least $1 - \binom{|M|}{2} \cdot \frac{\ell}{|\mathbb{F}|}$ over the random choice of (α, β), if the LogUp sum over M' is zero,*

$$\sum_{b \in \mathbb{F}} \frac{M'(b)}{b + \alpha} = \sum_{M(\mathbf{a}) \neq 0} \frac{M(\mathbf{a})}{h^\beta(\mathbf{a}) + \alpha} = 0,$$

then M is also an empty ℓ-dimensional multiset.

Proof. By theorem 2, M' is well-defined. Now, suppose the LogUp sum over M' equals zero, by the LogUp identity in theorem 1, this implies that M' is the zero function, meaning $M'(b) = 0$ for all $b \in \mathbb{F}$. Since $M'(b)$ iterates over all non-zero values of M, M must also be the zero function, completing the proof. □

LogUp Identity for Multiset Across Different Dimensions In some cases, we need to verify that several ℓ-dimensional multisets—each potentially with different dimensions—are empty. Suppose we have k multisets $M_1, M_2, \ldots, M_k$, where each $M_i : \mathbb{F}^{\ell_i} \to \mathbb{F}$ is a finite field multiset over the vector space $\mathbb{F}^{\ell_i}$ for some dimension ℓ_i. To simplify this verification, we aim to represent the union of these multisets as a single one-dimensional multiset.

A naive approach would be to define a one-dimensional multiset $M' : \mathbb{F} \to \mathbb{F}$ iteratively as follows: for each ℓ_i-dimensional multiset M_i, set $M'(h^\beta(\mathbf{a})) = M_i(\mathbf{a})$ for all $\mathbf{a} \in \mathbb{F}^{\ell_i}$ where $M_i(\mathbf{a}) \neq 0$, and finally set $M'(b) = 0$ for all other $b \in \mathbb{F}$. However, applying universal hashing directly to each M_i without accounting for their dimensional differences introduces ambiguities. For example, a vector (1) in $\mathbb{F}^1$ will hash to the same value as a vector $(1, 0)$ in $\mathbb{F}^2$ no matter the choice of β, causing collisions that make M' not well-defined.

To resolve this, we use *dimensional encoding*, where each vector is extended by appending a constant term (specifically, 1) to signify its dimension. For a vector $\mathbf{a}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,\ell_i}) \in \mathbb{F}^{\ell_i}$, we encode it as:

$$\mathbf{a}'_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,\ell_i}, 1).$$

This encoding ensures that vectors from different dimensions are distinct under universal hashing.

After hashing each dimension-specific multiset M_i wrapped with such encoding, we obtain a single one-dimensional multiset $M' : \mathbb{F} \to \mathbb{F}$ that combines all M_i's multiplicities:

Claim 2. *The one-dimensional multiset $M' : \mathbb{F} \to \mathbb{F}$, constructed by hashing each dimension-specific multiset M_i with h^β and applying dimensional encoding, is well-defined with probability at least*

$$1 - \binom{\sum_{i=1}^k |M_i|}{2} \cdot \frac{\mathsf{max}_i \ell_i}{|\mathbb{F}|}.$$

Proof. The dimensional encoding ensures that each vector from each multiset M_i maps to a unique hash value as a function of β. The probability calculation follows directly from theorem 2, applying a union bound over all pairs of distinct vectors across the multisets $M_1, \ldots, M_k$. □

Theorem 4 (LogUp Identity across Different Dimensional Multisets)**.** *Let $M_1, \ldots, M_k$ be finite field multisets over vector spaces $\mathbb{F}^{\ell_1}, \ldots, \mathbb{F}^{\ell_k}$, respectively, and let $M' : \mathbb{F} \to \mathbb{F}$ be the one-dimensional multiset obtained by hashing each dimension-specific multiset M_i using h^β with dimensional encoding. Then, with probability at least $1 - \binom{\sum_{i=1}^k |M_i|}{2} \cdot \frac{\mathsf{max}_i \ell_i}{|\mathbb{F}|}$ over the random choice of (α, β), if the LogUp sum over M' is zero, i.e.,*

$$\sum_{b \in \mathbb{F}} \frac{M'(b)}{b + \alpha} = \sum_{i=1}^k \sum_{\mathbf{a} \in \mathbb{F}^{\ell_i}} \frac{M_i(\mathbf{a})}{h^\beta(\mathbf{a}') + \alpha} = 0,$$

then each original ℓ_i-dimensional multiset M_i is empty.

Proof. The proof can be easily established based on claim 2 and theorem 3. □

1.5 AIR Buses

Definition 10 (AIR Buses)**.** *An AIR Bus (denoted by* Airbus*) is a collection of multisets* $M_1, \ldots, M_k$*, each defined over different dimensions* $\ell_1, \ldots, \ell_k$*. Each multiset* M_i *corresponds to a set of* ℓ_i *virtual columns* $P_{i,1}, \ldots, P_{i,\ell_i}$ *over some AIR trace, representing the* ℓ_i*-dimensional data associated with that multiset.*

The construction of each multiset M_i *is as follows:*

- *Each virtual column* $P_{i,j}$ *is represented as a multivariate polynomial over the columns in the AIR trace. Since each column in a trace can be represented by a univariate polynomial over* x *in the interpolation domain* H*, we can iterate over each virtual column* $P_{i,j}$ *by evaluating it at each* $x \in H$*. This allows us to denote the values of the* ℓ_i *virtual columns collectively as* $P_i(x) = (P_{i,1}(x), \ldots, P_{i,\ell_i}(x))$*.*

- *To fully define the multiset* M_i*, we introduce a multiplicity column* $m_i(x)$*, which is itself another virtual column. For each* $x \in H$*, the multiplicity of the tuple* $(P_{i,1}(x), \ldots, P_{i,\ell_i}(x))$ *is specified by* $M_i(P_{i,1}(x), \ldots, P_{i,\ell_i}(x)) = m_i(x)$*, where* x *iterates over* H*. We then set* $M_i(\mathbf{a}) = 0$ *for any tuple* $\mathbf{a} \in \mathbb{F}^{\ell_i}$ *not generated by* $P_i(x)$ *for some* $x \in H$*.*

To summarize, an $\mathsf{Airbus} = \{(P_i, m_i)\}_{i\in[1,k]}$ *comprises the multisets* $M_1, \ldots, M_k$*, where each* M_i *is specified by:*

1. *A unique set of virtual columns* $P_{i,1}, \ldots, P_{i,\ell_i}$*, corresponding to each dimension* ℓ_i *specifying the elements with non-zero multiplicity.*

2. *A multiplicity virtual column* $m_i(x)$*, which assigns the multiplicity to each element above.*

Definition 11 (AIR Bus Constraint)**.** *Given any specific trace* $\mathbf{Tr}$ *and specific* $\mathsf{Airbus} = \{(P_i, m_i)\}_{i\in[1,k]}$*, the corresponding AIR Bus Constraint* $\mathcal{C}_{\mathsf{Airbus},\mathbf{Tr}}$ *is a specific type of constraint that enforces the emptiness of multiple multisets* $M_1, \ldots, M_k$ *specified by a single* Airbus *defined over the given trace* $\mathbf{Tr}$*. Let* H *be the interpolation domain for* $\mathbf{Tr}$*. The constraint is an LogUp identity which iterates over two nested loops:*

$$\mathcal{C}_{\mathsf{Airbus},\mathbf{Tr}} = \sum_{x\in H}\sum_{i=1}^{k} \frac{m_i(x)}{h^{\beta}(P_i'(x)) + \alpha^{\mathsf{ID}_{\mathsf{Airbus}}}} = 0,$$

where $\mathsf{ID}_{\mathsf{Airbus}}$ is a unique identity (a field element) which is given as the identifier of such Airbus.

Recall that $P_i'(x)$ denotes the dimensional encoding of $P_i(x)$. In the case where $k = 1$, we can drop the dimensional encoding, and use $P_i(x)$ directly in the AIR Bus constraint.

Definition 12 (Multi-AIR Multi-Bus Constraint)**.** *Let* $\mathbf{Tr}_1, \ldots, \mathbf{Tr}_p$ *be a set of AIR traces, each with an associated interpolation domain* H_j *for* $j \in [1, p]$*. Let* $\mathsf{Airbus}_1, \ldots, \mathsf{Airbus}_t$ *be a collection of AIR buses.*

The Multi-AIR Multi-Bus Constraint is a constraint that enforces the emptiness of all multisets specified by all AIR buses across all AIR traces. This constraint is an extension of the AIR Bus Constraint, incorporating two additional nested summations:

- *The outermost summation iterates over each AIR bus* Airbus_i*.*

- *The next summation iterates over each AIR trace* $\mathbf{Tr}_j$*, using its specific interpolation domain* H_j*.*

- *For each given AIR bus* Airbus_i *and each given trace* $\mathbf{Tr}_j$*, the inner summations correspond to an AIR Bus constraint* $\mathcal{C}_{\mathsf{Airbus}_i,\mathbf{Tr}_j}$*.*

The Multi-AIR Multi-Bus constraint is expressed as:

$$\sum_{i=1}^{t}\sum_{j=1}^{p} \mathcal{C}_{\mathsf{Airbus}_i,\mathbf{Tr}_j} = 0,$$

We now give a theorem over the above Multi-AIR Multi-Bus constraint:

Theorem 5 (LogUp Identity over Multi-AIR Multi-Bus)**.** *If the above Multi-AIR Multi-Bus constraint is satisfied, then with probability at least* $1 - \binom{\sum_{j=1}^{p} k_i\cdot|H_j|}{2} \cdot \frac{\max_{i\in[1,k_i]}\ell_i}{|\mathbb{F}|} - \frac{t-1}{|\mathbb{F}_{ext}|}$*, for each AIR bus* Airbus_i *over the set of AIR traces, the multisets defined in* Airbus_i *with respect to all traces* $\mathbf{Tr}_1, \ldots, \mathbf{Tr}_p$ *must be empty.*

Proof. First, unroll the Multi-AIR Multi-Bus constraint explicitly as

$$\sum_{i=1}^{t}\sum_{j=1}^{p}\sum_{x\in H_j}\sum_{k=1}^{k_i}\frac{m_{i_k}(x)}{h^{\beta}(P'_{i_k}(x))+\alpha^{\mathsf{ID}_{\mathsf{Airbus}_i}}}=0.$$

This expression can be viewed as a sum of t terms, where each term is a Multi-AIR Bus constraint as follows:

$$\sum_{k=1}^{k_i}\sum_{j=1}^{p}\sum_{x\in H_j}\frac{m_{i_k}(x)}{h^{\beta}(P'_{i_k}(x))+\alpha^{\mathsf{ID}_{\mathsf{Airbus}_i}}}$$

In particular, each term is associated with a unique indeterminate $\alpha^{\mathsf{ID}_{\mathsf{Airbus}_i}}$ due to each Airbus has a unique identifier.

Since each term depends uniquely on α raised to a unique power, they are linearly independent functions over α. Therefore, the only way this linear combination yields zero is if each term is the zero function. In particular, when we replace the indeterminate α with random element in $\mathbb{F}$, this fact still holds with probability at least $1-\frac{t-1}{|\mathbb{F}|}$. Now notice that the partial sum $\sum_{j=1}^{p}\sum_{x\in H_j}\frac{m_{i_k}(x)}{h^{\beta}(P'_{i_k}(x))+\alpha^{\mathsf{ID}_{\mathsf{Airbus}_i}}}$ can be viewed as $\sum_{\mathbf{a}\in\mathbb{F}^{\ell_i}}\frac{M_{i_k}(\mathbf{a})}{h^{\beta}(\mathbf{a}')+\alpha^{\mathsf{ID}_{\mathsf{Airbus}_i}}}$, where as $\mathbf{a}$ iterates over $\mathbb{F}^{\ell_i}$, $M_{i_k}(\mathbf{a})$ is precisely the kth multiset $(P_{i_k}, m_{i,k})$ specified by Airbus_i that iterates over all traces $\mathbf{Tr}_1,\ldots,\mathbf{Tr}_p$. The rest of proof then follows easily from theorem 4. $\square$

1.6 Virtual Machine Execution

This section describes the execution model of a virtual machine built on a Harvard architecture, which separates program memory from data memory. In particular, we split the memory into read-only memory (ROM) and random access memory (RAM).

1.6.1 ROM: Program Memory

The ROM is used to store the program memory, which contains the bytecode to be executed by the virtual machine. Each instruction in the ROM is encoded as a tuple consisting of an opcode and operands, and is fetched sequentially or via control flow instructions like branches or jumps. The ROM is read-only, ensuring that the program memory remains static throughout execution.

Instruction Format Instructions in the virtual machine are encoded in the ROM as a tuple consisting of an opcode and operands. Each instruction is structured as an $(n+1)$-length tuple:

$$\mathsf{Inst}=(\mathsf{opcode},\text{operands}),$$

where:

- $\mathsf{opcode}\in\mathbb{F}$: Specifies one of the instructions in the instruction set.
- operands $=(\mathrm{op}_1,\mathrm{op}_2,\ldots,\mathrm{op}_n)$: A fixed-length vector of field elements representing addresses, values, or immediates. By default, we use 7 operands. Unused operands are set to zero by default.

1.6.2 RAM: Data Memory and Block Access

The RAM is organized into several address spaces and supports both single-cell and block accesses. Each memory cell in this space holds a value in $\mathbb{F}$. Block accesses are supported with the following restrictions:

- The number of cells in a block access is restricted to powers of two: $1, 2, 4, 8, 16, 32, 64$.

We define the state of the virtual machine, the transition function that evolves the machine state, and the correctness of the execution.

1.6.3 State of the Virtual Machine

We define the state of the virtual machine as a tuple containing the current program counter (pc) and a memory map (Mem).

Definition 13 (Virtual Machine State)**.** *The state of the virtual machine, denoted by* State*, is defined as:*

$$\mathsf{State} = (\mathsf{pc}, \mathsf{Mem}),$$

where:

- $\mathsf{pc} \in \mathbb{F}$ *is the program counter, which holds the address of the next instruction to be fetched from program memory. The program counter* pc *is a single field element and points to the location of the instruction being executed.* [1]

- Mem *is a memory map. The memory is comprised of addressable cells, each cell containing a single field element. Instructions of the virtual machine may access (read or write) memory as single cells or as contiguous lists of cells, called blocks.*

  The memory map is defined as $\mathsf{Mem} : (\mathbb{F} \times \mathbb{F}) \to \mathbb{F}$*, where each memory address is represented as a tuple* $(space, pointer)$*, where* $space \in \mathbb{F}$ *is the address space identifier and* $pointer \in \mathbb{F}$ *specifies the exact location within the chosen address space. Each memory cell holds a value in the base field* $\mathbb{F}$*. For ease of notation, addressing conventions are as follows:*

  - $\mathsf{Mem}[a]_d$ *denotes the single-cell value at pointer location* a *in address space* d*. This is a single field element.*
  - $\mathsf{Mem}[a : N]_d$ *denotes the slice* $[a..a + N]_d$*, which is a length-*N *array of field elements.*

Since the ROM is read-only, its state will remain unchanged during VM execution. For such reason, we will use such mutable memory map to specifically represent the RAM, and we will think of ROM as immutable state of the VM, which is included in its initial state.

Remark 1.2 (VM Registers)**.** *Registers are treated as pointers to a separate address space, comprised of addressable cells. This allows general-purpose registers to emulate word-sized memory units, and they are stored in their own address space within the memory map.*

Remark 1.3 (Address Space in RAM)**.** *The virtual machine supports multiple configurable address spaces, each with its own range of pointers. The number of address spaces and their sizes are configurable constants of the VM.*

The following fixed address spaces are supported by default:

Address Space	***Name***
0	*Immediates (special, not a real address space;* $[a]_0 = a$*)*
1	*Registers*
2	*User Data Memory*
3	*User IO*
4	*Native Kernel*

1.6.4 Definition of the Virtual Machine

A virtual machine is specified by a triple: $\mathsf{VM} = (\mathsf{State}_0, \mathsf{State}_{\text{final}}, \mathsf{Step})$, where

- The **initial state**, denoted by $\mathsf{State}_0 = (\mathsf{pc}_0, \overline{\mathsf{Mem}_0})$, consists of:
  - pc_0 is set to the starting address of the program.
  - $\overline{\mathsf{Mem}_0}$ consists of two parts: $\mathsf{Mem}_0{}^*$ and Mem_0. $\mathsf{Mem}_0{}^*$ represents the ROM, which is initialized with the program instructions and treated as an invariant. Mem_0 denotes the initial state of RAM, which contains any initial data (e.g., user input/outputs).

[1] Program code is addressed by any field element in the range $[0, 2^{\mathsf{PC\_BITS}})$, where $\mathsf{PC\_BITS} = 30$. By default, instructions are stored at multiples of $\mathsf{DEFAULT\_PC\_STEP} = 4$, following RISC-V conventions.

- A **final state**, denoted by $\mathsf{State}_{\text{final}}$, reached when a terminating instruction (e.g., `TERMINATE`) is executed.
- A **transition function** Step that defines how the virtual machine evolves from one state to another:

$$\mathsf{Step} : \mathsf{State}_{\text{curr}} \to \mathsf{State}_{\text{next}}.$$

In more detail, given the current state $\mathsf{State}_{\text{curr}} = (\mathsf{pc}, \mathsf{Mem})$, the transition function proceeds as follows:

1. **Instruction Fetch**:

$$(\mathsf{opcode}, \mathsf{operands}) \leftarrow \mathsf{Fetch}(\mathsf{Mem}, \mathsf{pc})$$

The instruction is fetched from program memory using the current program counter, and subsequently decoded into its opcode and operands.

2. **Instruction Execution**: The virtual machine executes the instruction according to the decoded opcode. Consider the following examples:
   - For an arithmetic operation (e.g., addition):

$$\mathsf{Mem}[a_3]_1 = \mathsf{Mem}[a_1]_1 + \mathsf{Mem}[a_2]_1$$

   - For a single-cell read(load) operation:

$$\mathsf{Mem}[a_3]_1 = \mathsf{Mem}[a_2]_2$$

   - For a block-level write(store) operation:

$$\mathsf{Mem}[a_2 : 8]_2 = \mathsf{Mem}[a_3 : 8]_1$$

We denote by $\mathsf{Mem}_{\text{next}}$ the new memory state after all the memory accesses have been completed during such instruction.

3. **Program Counter Update**: If no jump or branch instruction is executed, the program counter is then incremented to fetch the next sequential instruction. By default, we update:

$$\mathsf{pc}_{\text{next}} = \mathsf{pc} + 4$$

4. **State Update**: The next state is:

$$\mathsf{State}_{\text{next}} = (\mathsf{pc}_{\text{next}}, \mathsf{Mem}_{\text{next}})$$

1.6.5 Correct Execution of the Virtual Machine

Definition 14 (Correct Execution)**.** *The execution of the virtual machine is correct if, given a discrete time span defined by a global timer* timer*, starting from a given initial state* State_0 *at* timer_0*, it reaches a specified final state* State_{final} *at* timer_{final}*, such that there exists an execution trace during the entire time span* $[\mathsf{timer}_0, \ldots, \mathsf{timer}_{final}]$*:*

$$\mathsf{State}_0 \xrightarrow{\mathsf{Step}} \mathsf{State}_1 \xrightarrow{\mathsf{Step}} \mathsf{State}_2 \xrightarrow{\mathsf{Step}} \ldots \xrightarrow{\mathsf{Step}} \mathsf{State}_{final},$$

and the correct transition behavior is applied to every consecutive transition Step*. In particular, the following conditions must hold:*

1. ***Global Timer Association:*** *There exists a sequence of timestamps such that each state* State_i *is associated with a unique timestamp* timer_i*, such that:*

$$\mathsf{timer}_0 < \mathsf{timer}_1 < \mathsf{timer}_2 < \cdots < \mathsf{timer}_{final}.$$

2. ***Instruction Execution Timing**: Each instruction execution (denoted as the i-th execution) spans a time interval* $[\mathsf{timer}_i, \mathsf{timer}_{i+1}]$*, where:*
   - timer_i *is the time at which the instruction is fetched (based on* pc*).*
   - timer_{i+1} *is the time at which the program counter is updated, and the state transitions to* State_{next}*.*
   - *All intermediate events during the instruction execution (e.g., memory reads and writes) occur at timestamps* $t_{checkpoint,j}$*, such that:*

     $$\mathsf{timer}_i < t_{checkpoint,1} < t_{checkpoint,2} < \cdots < t_{checkpoint,k} < \mathsf{timer}_{i+1},$$

     where k is the total number of memory accesses (reads or writes) during the instruction. The number of these memory accesses must follow the ordering dictated by the instruction opcode (e.g., read-read-write).

3. ***Instruction Fetch Correctness**: Each instruction fetched from program memory must match the value stored at the corresponding address in the ROM:*

   $$(\mathsf{opcode}, operands) = \mathsf{Fetch}(\mathsf{Mem}, \mathsf{pc}),$$

   where $\mathsf{Mem}[\mathsf{pc}]_0$ *refers to the ROM's value at the address* pc*, and* opcode *and operands correctly decode this value.*

4. ***Memory Access Timing and Consistency**:*
   - ***Access Timing:** All memory accesses (reads or writes) must occur at valid checkpoints within the current instruction's time interval* $[\mathsf{timer}_i, \mathsf{timer}_{i+1}]$*. Specifically:*

     $$\mathsf{timer}_i < t_{checkpoint,1} < t_{checkpoint,2} < \cdots < t_{checkpoint,k} < \mathsf{timer}_{i+1},$$

     where $t_{checkpoint,j}$ *corresponds to the timestamp of the j-th memory access as dictated by the current instruction at* timer_i*. The values read or written at each checkpoint must align with the instruction's semantics as defined by the opcode.*
   - ***Consistency:** For each memory read access (single-cell or block-level), the returned memory state at a specific timestamp* $t_{checkpoint,j}$ *must reflect the content of the most recent write to that address or block at a prior timestamp* $t_{checkpoint,k}$*, such that:*

     $$t_{checkpoint,k} < t_{checkpoint,j},$$

     and no intermediate updates occur in the interval $t_{checkpoint,k} < t < t_{checkpoint,j}$*.*

5. ***Program Counter Update Correctness**: After executing each instruction, the program counter is updated correctly:*

   $$\mathsf{pc}_{\mathsf{timer_{i+1}}} = \begin{cases} \textit{branch target or jump address,} & \textit{if the instruction modifies control flow;} \\ \mathsf{pc}_{\mathsf{timer_i}} + 4, & \textit{otherwise.} \end{cases}$$

1.7 Modular VM Execution Across Multiple Chips

We utilize a no-CPU design philosophy. This design is motivated by the observation that a centralized CPU leads to a trace matrix with rows that grow with the total number of clock cycles in program execution. By eliminating the CPU, we reduce the number of required trace cells added per opcode execution to its theoretical minimum.

In our virtual machine, we utilize a modular architecture where the transition function (Step) is not executed by a central CPU. Instead, the system is divided across multiple specialized chips, each dedicated to handling specific opcodes from the instruction set. More specifically:

- Each chip independently fetches its specialized instructions from the program memory (ROM). Only the operands relevant to the chip's execution are fetched, with unused operands set to zero, ensuring efficient operand handling. Then each chip is responsible for correctly executing each instruction.
- The $\mathsf{new\_pc}$ is constrained by each chip based on the opcode it handles:
  - For sequential instructions, $\mathsf{new\_pc} = \mathsf{pc} + 4$.
  - For control flow instructions, $\mathsf{new\_pc}$ is updated based on the branch or jump target.
- Each chip also enforces that each memory read/write access of each instruction execution occurs during the correct checkpoint in the instruction time interval.

1.8 AIR Traces for Virtual Machine

To record the execution trace of each chip in our virtual machine, we utilize Algebraic Intermediate Representation (AIR) traces. Each chip maintains its own AIR trace, capturing the evolution of its state as it executes instructions. These traces are essential for verifying the correctness of the modular execution of the virtual machine, ensuring consistency in memory access, instruction fetching, program counter updates, and opcode execution.

Required (Virtual) Columns for AIR Traces Each AIR trace for a chip includes the following columns. We point out that one or more of those columns may itself span multiple columns in AIR trace, and sometimes could be virtual columns, as defined in definition 2.

Column Name	Description
$\mathbf{Tr}_{\mathsf{pc}}$	Program counter (pc)
$\mathbf{Tr}_{\mathsf{opcode}}$	Opcode of the instruction
$\mathbf{Tr}_{\text{operands}}$	Operands for the instruction
$\mathbf{Tr}_{\text{pointer,space}}$	RAM address space and pointer
$\mathbf{Tr}_{\mathsf{Mem}[\text{pointer}]_{\text{space}}}$	RAM value
$\mathbf{Tr}_{\mathsf{timer}_{\text{curr}}}$	The time when the current instruction execution timing begins
$\mathbf{Tr}_{\mathsf{timer}_{\text{next}}}$	The time when the current instruction execution timing ends
$\mathbf{Tr}_{t_{\text{checkpoint,prev}}}$	The checkpoint where certain memory cell/block is last updated
$\mathbf{Tr}_{t_{\text{checkpoint,curr}}}$	The checkpoint corresponding to the current time when an update is issued to that memory cell

Table 1: Columns in the AIR Trace for Each Chip

1.9 Three AIR Buses: $\mathsf{Airbus}_{\mathsf{Mem}}, \mathsf{Airbus}_{\mathsf{Prog}}, \mathsf{Airbus}_{\mathsf{Exe}}$

1.9.1 Memory Bus

To ensure the correctness of memory accesses (reads and writes) in our modular virtual machine architecture, we utilize a Memory Bus ($\mathsf{Airbus}_{\mathsf{Mem}}$). This is achieved through offline memory checking [BEG$^+$94], which relies on multiset equality to verify memory consistency.

In the offline memory checking process, each memory operation is recorded in the form (pointer, space, $\mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint}}$), where:

- $(\text{pointer}, \text{space})$ specifies the memory address, where space identifies the address space, and pointer indicates the exact location within that space.
- $\mathsf{Mem}[\text{pointer}]_{\text{space}}$ represents the value being read or written. It is a vector of elements from the base field $\mathbb{F}$. The length of this vector is variable, allowing flexibility depending on the data size (single-cell or contiguous block). Initially, we focus on single-cell memory access (where the vector has length 1). Later, we introduce adapter chips that enable block memory access.
- $t_{\text{checkpoint}} \in \mathbb{F}$ is the checkpoint associated with the memory operation. Specifically:

- When reading a value, $t_{\text{checkpoint}}$ corresponds to $t_{\text{checkpoint,prev}}$, which is the checkpoint of the most recent write to $(\text{pointer}, \text{space})$. This value must be provided as non-deterministic input (hint) to each chip.
- $t_{\text{checkpoint,curr}}$ will then replace $t_{\text{checkpoint,prev}}$ as the new checkpoint, signifying the (newest) most recent access to $(\text{pointer}, \text{space})$.

Grouping of Memory Access Records Memory access records $(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint}})$ are grouped by the length of $\mathsf{Mem}[\text{pointer}]_{\text{space}}$. In the most general case, when dealing with block memory access, for each distinct block length $\ell = |\mathsf{Mem}[\text{pointer}]_{\text{space}}| + 3$, we define a vector representation:

$$(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint}})$$

We then define a collection of multisets for different dimensions ℓ, each corresponding to a specific block size.

Defining Multisets To ensure that each memory operation adheres to the consistency rules (i.e., reads return the most recent write value), we use two distinct multisets for each dimension ℓ:

- **Receive Multiset** $M^{\ell}_{\text{receive}}$: Captures the previous state of memory at the time of a read operation. It logs the value associated with each memory address at the time $t_{\text{checkpoint,prev}}$:

$$M^{\ell}_{\text{receive}} = \#\{(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,prev}})\}.$$

- **Send Multiset** M^{ℓ}_{send}: Captures the updated state of memory after an access. It logs the current (newest) value written to each memory address at the current time $t_{\text{checkpoint,curr}}$:

$$M^{\ell}_{\text{send}} = \#\{(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,curr}})\}.$$

Offline Memory Checking: Handling Read and Write Operations Each chip in the system independently maintains memory consistency by interacting with the **Receive** and **Send** multisets during both read and write operations. These interactions are defined as follows. For simplicity, we first focus on single-cell memory access.

- **Read Operation**:
  1. The chip logs the prior state of the memory address by adding to the **Receive Multiset**:

$$M^{\ell}_{\text{receive}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,prev}}),$$

  2. The chip confirms the read operation by adding to the **Send Multiset**:

$$M^{\ell}_{\text{send}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,curr}}),$$

  ensuring that:

$$t_{\text{checkpoint,curr}} > t_{\text{checkpoint,prev}}.$$

- **Write Operation**:
  1. Before writing a new value $\mathsf{Mem}[\text{pointer}]^{\text{new}}_{\text{space}}$, the chip logs the previous state by adding to the **Receive Multiset**:

$$M^{\ell}_{\text{receive}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,prev}}),$$

  2. After updating the memory with the new value, the chip logs the updated state by adding to the **Send Multiset**:

$$M^{\ell}_{\text{send}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer}]^{\text{new}}_{\text{space}}, t_{\text{checkpoint,curr}}),$$

  ensuring that:

$$t_{\text{checkpoint,curr}} > t_{\text{checkpoint,prev}}.$$

Balancing Initial Sends and Final Receives This directly relates to subsubsection 1.6.4, where the initial and final states of the virtual machine are defined. For a virtual machine VM, the initial and final states include:

- An **initial state** $\mathsf{State}_0 = (\mathsf{pc}_0, \mathsf{Mem}_0)$, where:
  - pc_0 is set to the starting address of the program.
  - Mem_0 is initialized with initial user input/data in RAM.
- A **final state** $\mathsf{State}_{\text{final}} = (\mathsf{pc}_{\text{final}}, \mathsf{Mem}_{\text{final}})$, reached upon execution of a terminating instruction (e.g., `TERMINATION`).

To ensure memory consistency throughout the program execution, we enforce the following constraints:

- An **initial entry** for each accessed memory address:

$$(\text{pointer}, \text{space}, \mathsf{Mem}_0[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,init}}) \in M_{\text{send}},$$

  where $t_{\text{checkpoint,init}} = \mathsf{timer}_0$, representing the initial state.

- A **final entry** for each memory address accessed during execution:

$$(\text{pointer}, \text{space}, \mathsf{Mem}_{\text{final}}[\text{pointer}]_{\text{space}}, t_{\text{checkpoint,final}}) \in M_{\text{receive}},$$

  where $\mathsf{Mem}_{\text{final}}[\text{pointer}]_{\text{space}}$ represents the expected final value at $(\text{pointer}, \text{space})$, and $t_{\text{checkpoint,final}}$ is the checkpoint of its final access (This value is address specific).

Theorem 6 (Consistency Rule [BEG+94])**.** *The memory state transitions are valid and that all reads return the most recent written value, if and only if the receive and send multisets must satisfy the following equality:*

$$M^{\ell}_{receive} = M^{\ell}_{send}.$$

Definition 15 (Memory Bus ($\mathsf{Airbus}_{\mathsf{Mem}}$))**.** *A Memory Bus is a specialized AIR Bus designed to ensure memory access consistency. Each multiset $M \in \mathsf{Airbus}_{\mathsf{Mem}}$ captures differences between the Receive and Send multisets for varying dimensions. For each dimension ℓ, we define:*

$$M(x) = M^{\ell}_{receive}(P_1(x), \ldots, P_\ell(x)) - M^{\ell}_{send}(P_1(x), \ldots, P_\ell(x)),$$

where:

- $P_1(x), \ldots, P_\ell(x) = (\mathbf{Tr}_{pointer,space}(x), \mathbf{Tr}_{\mathsf{Mem}[pointer]_{space}}(x), (\mathbf{Tr}_{t_{checkpoint,prev}}(x))$ *or* $(\mathbf{Tr}_{t_{checkpoint,curr}}(x))$ *are individual columns from the AIR trace.*
- $M(x)$ *is the multiplicity column computed as the difference between the Receive and Send multisets.*

Corollary 7 (Read-Write Consistency in Memory Bus [BEG+94])**.** *Given the initial state State_0 and final state State_{final} of the virtual machine, let the collection of multisets M^{ℓ}_{send} and $M^{\ell}_{receive}$ be constructed as above. Assuming the condition $t_{checkpoint,curr} > t_{checkpoint,prev}$ is maintained correctly everywhere, then memory consistency is guaranteed (every read from RAM returns the most recently written value), if and only if the collection of $M(x)$ are empty multisets.*

1.9.2 Block Access to RAM

In addition to single-cell memory access, our virtual machine supports block-level memory access to RAM. Each block of size N is associated with a timestamp indicating the last time the block was accessed. This subsection describes how block memory access is handled, including the mechanisms for splitting and merging blocks to maintain memory consistency.

Motivation for Split and Merge When a block of memory is accessed, the system must ensure that only the relevant portion of the block is modified, leaving the untouched portions intact. This is achieved through the **split** operation, which divides the block into smaller sub-blocks, each with its own timestamp. Conversely, when two contiguous sub-blocks are accessed together, the **merge** operation combines them into a single parent block with a unified timestamp. The timestamp of the parent block is set to the maximum of the timestamps of the left and right sub-blocks. We introduce the **Access Adapter Chip** which plays a crucial role in enforcing these constraints as follows. This ensures that the send and receive multisets on the memory bus still balance at the end of execution.

Split Operation This operation divides a parent block into two sub-blocks, left and right. This ensures fine-grained control over memory access. During a split operation, the following steps take place:

- **Receive Multiset (Parent Block)**: The parent block is fetched from the memory bus and added to the **Receive Multiset**:

  $$M^{\ell}_{\text{receive}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer:N}]_{\text{space}}, t_{\text{checkpoint,parent, prev}})$$

  Here, $t_{\text{checkpoint,parent, prev}}$ must be provided as non-deterministic input (hint).

- **Send Multiset (Left and Right Blocks)**: The left and right sub-blocks created from the split are added to the **Send Multiset**:

  $$M^{\ell/2}_{\text{send}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer:N/2}]_{\text{space}}, t_{\text{checkpoint,parent, prev}})$$
  $$M^{\ell/2}_{\text{send}}(\text{pointer} + N/2, \text{space}, \mathsf{Mem}[\text{pointer+N/2:N/2}]_{\text{space}}, t_{\text{checkpoint,parent, prev}})$$

  Here, the left and right sub-blocks inherit the timestamp of the parent block.

Merge Operation Thisoperation combines two contiguous sub-blocks into a single parent block. This ensures that block-level access is efficiently handled when the entire block is accessed together. During a merge operation, the following steps take place:

- **Receive Multiset (Left and Right Sub-Blocks)**: The left and right sub-blocks are fetched from the memory bus and added to the **Receive Multiset**:

  $$M^{\ell/2}_{\text{receive}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer:N/2}]_{\text{space}}, t_{\text{checkpoint,left, prev}})$$
  $$M^{\ell/2}_{\text{receive}}(\text{pointer} + N/2, \text{space}, \mathsf{Mem}[\text{pointer+N/2:N/2}]_{\text{space}}, t_{\text{checkpoint,right, prev}})$$

  Here, both $t_{\text{checkpoint,left, prev}}$ and $t_{\text{checkpoint,right, prev}}$ must be provided as non-deterministic inputs (hints).

- **Send Multiset (Parent Block)**: The parent block created from the merge is added to the **Send Multiset**:

  $$M^{\ell}_{\text{send}}(\text{pointer}, \text{space}, \mathsf{Mem}[\text{pointer:N}]_{\text{space}}, t_{\text{checkpoint,parent}})$$

  The parent block's timestamp is set to:

  $$t_{\text{checkpoint,parent}} = \mathsf{max}(t_{\text{checkpoint,left, prev}}, t_{\text{checkpoint,right, prev}})$$

  ensuring consistency in timestamp propagation.

Extension to Block-Level Operations While the above discussion focuses on block memory access, the same principles apply to higher-level operations that involve multiple blocks. By recursively applying split and merge operations, the virtual machine can handle arbitrary memory access patterns while ensuring correctness and consistency.

1.9.3 Program Bus

The Program Bus ($\mathsf{Airbus}_{\mathsf{Prog}}$) ensures the correctness of instruction fetching in our virtual machine system. This bus verifies that instructions are fetched sequentially and correctly from program memory (address space 2), based on the current program counter (pc) and associated timestamps.

Similar to the Memory Bus ($\mathsf{Airbus}_{\mathsf{Mem}}$), the Program Bus uses multisets to enforce consistency in the sequence of fetched instructions, ensuring that each fetch operation aligns with the correct program counter and timestamp. However, since program memory is read-only, the Program Bus exclusively handles read operations, disallowing any write operations.

Program Access Records Program access records $(\mathsf{pc}, \mathsf{opcode}||\text{operands}, t_{\text{checkpoint}})$ are structured as follows:

- pc: The program counter representing the address from which the instruction is fetched.
- $\mathsf{opcode}||\text{operands}$: The full instruction data, consisting of an operation code followed by its operands.
- $t_{\text{checkpoint}}$: The checkpoint indicating the timestamp associated with the instruction fetch:
  - $t_{\text{checkpoint,prev}}$: The last time the same instruction was fetched.
  - $t_{\text{checkpoint,curr}}$: The current time at which the instruction is being fetched.

Since instructions have a fixed format (with a single opcode and a constant number of operands), the data captured in the Program Bus does not require grouping by variable dimensions, unlike the Memory Bus ($\mathsf{Airbus}_{\mathsf{Mem}}$). A single dimension suffices to represent all entries in the Program Bus.

Receive and Send Multisets To verify instruction-fetch consistency, the Program Bus defines two multisets:

- **Receive Multiset** M_{receive}: Logs the state of the program ROM and unconstrained multiplicity for how many times a particular program address was requested:

$$M_{\text{receive}}(\mathsf{pc}, \mathsf{opcode}||\text{operands}, \textit{multiplicity})$$

  ensures that the fetched instruction matches the most recent prior state of the program memory.

- **Send Multiset** M_{send}: Logs the instruction lookup:

$$M_{\text{send}}(\mathsf{pc}, \mathsf{opcode}||\text{operands}, \textit{multiplicity})$$

  ensures that the program ROM is accessed consistently according to program counter.

Each chip will add to the send multisets each time it fetches its instruction from ROM. Since the instructions are all stored in the ROM, we will only use $\mathsf{Mem}_0{}^*$ to create the receive multiset.

Definition 16 (Program Bus ($\mathsf{Airbus}_{\mathsf{Prog}}$))**.** *A Program Bus is a specialized AIR Bus designed to ensure instruction fetch consistency from program memory (ROM). The multiset $M \in \mathsf{Airbus}_{\mathsf{Prog}}$ captures differences between the Receive and Send multisets. We define:*

$$M(x) = M_{receive}(P_1(x), P_2(x), P_3(x)) - M_{send}(P_1(x), P_2(x), P_3(x)),$$

where $P_1(x), P_2(x), P_3(x) = (\mathbf{Tr}_{\mathsf{pc}}(x), (\mathbf{Tr}_{\mathsf{opcode}}(x), \mathbf{Tr}_{operands}(x)), \mathbf{Tr}_{multiplicity}(x))$ are individual columns from the AIR trace.

Theorem 8 (Instruction Fetch Consistency in Program Bus)**.** *Given the initial state State_0 containing the program code in ROM ($\mathsf{Mem}_0{}^*$), and the final state State_{final} of the virtual machine, let the two multisets M_{send} and $M_{receive}$ be constructed as above. Then all instructions fetched from program memory are consistent with the initial state of ROM if and only if:*

$$M(x) = 0.$$

1.9.4 Execution Bus

The Execution Bus ($\mathsf{Airbus_{Exe}}$) ensures that the program counter (pc) are correctly updated across all chips in the system, based on the instruction execution timing which is logged via timer.

Updating the Program Counter and Timestamps In our modular architecture, each chip fetches instructions and updates its own program counter (pc) autonomously. However, instructions such as branches or jumps may update the program counter differently based on specific conditions or target addresses. The timestamps (timer_i) ensure that each state transition is correctly sequenced in time, with every instruction execution spanning from $\mathsf{timer}_i = \mathsf{timer}_{\text{curr}}$ (current) to $\mathsf{timer}_{i+1} = \mathsf{timer}_{\text{next}}$ (next).

Ensuring Correct Updates with the Execution Bus The Execution Bus uses two multisets to ensure that each chip correctly updates its program counter and maintains the correct order of execution:

- **Receive Multiset** M_{receive}: Records the observed pairs $(\mathsf{pc}_i, \mathsf{timer}_i)$ representing the program counter and timestamp values of the current state. Each chip must observe its current program counter pc_i at the current timestamp timer_i, which marks the beginning of the instruction execution timing. At this point, the chip fetches the current instruction based on pc_i, and the pair $(\mathsf{pc}_i, \mathsf{timer}_i)$ is added to the **Receive Multiset**. Formally:
$$M_{\text{receive}} = \{(\mathsf{pc}_i, \mathsf{timer}_i)\}\,.$$
- **Send Multiset** M_{send}: Captures the expected pairs $(\mathsf{pc}_{i+1}, \mathsf{timer}_{i+1})$ after the instruction execution is completed. At the end of the instruction execution timing, marked by timer_{i+1}, the chip updates the program counter pc_i to the next program counter pc_{i+1}. The updated pair $(\mathsf{pc}_{i+1}, \mathsf{timer}_{i+1})$ is then added to the **Send Multiset**. Formally:
$$M_{\text{send}} = \left\{(\mathsf{pc}_{i+1}, \mathsf{timer}_{i+1})\right\}\,.$$
  To ensure the flow of time and maintain consistency in the execution trace, we enforce:
$$\mathsf{timer}_{i+1} > \mathsf{timer}_i,$$
  which guarantees that the timestamps strictly progress. Additionally, to prevent overflow and maintain the integrity of the timestamp history, we must ensure that the difference $\mathsf{timer}_{i+1} - \mathsf{timer}_i$ does not exceed a large value, bounded by the field characteristic.[2]

Definition of the Execution Bus The Execution Bus is defined as the multiset difference between the receive and send sets:
$$M(x) = M_{\text{receive}}(\mathbf{Tr}_{\mathsf{pc}}(x), \mathbf{Tr}_{\mathsf{timer}_{\text{curr}}}(x)) - M_{\text{send}}(\mathbf{Tr}_{\mathsf{pc}_{\text{next}}}(x), \mathbf{Tr}_{\mathsf{timer}_{\text{next}}}(x)),$$
where:

- $\mathbf{Tr}_{\mathsf{pc}}(x)$: The program counter ($\mathsf{pc}_i$) from the current state.
- $\mathbf{Tr}_{\mathsf{timer}_{\text{curr}}}(x)$: The current timestamp (timer_i) for the state.
- $\mathbf{Tr}_{\mathsf{pc}_{\text{next}}}(x)$: The program counter ($\mathsf{pc}_{i+1}$) for the next state.
- $\mathbf{Tr}_{\mathsf{timer}_{\text{next}}}(x)$: The next timestamp ($\mathsf{timer}_{i+1}$) for the state.

Theorem 9 (Program Counter and Timestamp Consistency in the Execution Bus)**.** *Given the initial state* $\mathsf{State}_0 = (\mathsf{pc}_0, \mathsf{timer}_0)$ *and the final state* $\mathsf{State}_{final} = (\mathsf{pc}_{final}, \mathsf{timer}_{final})$*, let the two multisets* M_{send} *and* $M_{receive}$ *be constructed as above. Assuming that every local update to* pc_{i+1} *is correct and* $\mathsf{timer}_{i+1} > \mathsf{timer}_i$ *is maintained everywhere, then all the updates to the program counter (*pc*) are correctly sequenced in time across all chips in the system, with the initial program counter matching* pc_0*, if and only if:*
$$M(x) = 0.$$

[2]This constraint avoids inconsistencies that may arise due to large timestamp gaps or field overflow.

Proof. If $M(x) = M_{\text{receive}} - M_{\text{send}} = 0$, this implies that for every expected update $(\mathsf{pc}_{i+1}, \mathsf{timer}_{i+1})$, there exists a corresponding observed entry $(\mathsf{pc}_i, \mathsf{timer}_i)$ such that:

$$\mathsf{pc}_{i+1} = \mathsf{pc}_i \quad \text{and} \quad \mathsf{timer}_{i+1} > \mathsf{timer}_i.$$

This ensures:

1. **Sequential Updates**: For sequential instructions, the updates are correctly reflected as $\mathsf{pc}_{i+1} = \mathsf{pc}_i + 4$. For branch or jump instructions, pc_{i+1} is updated appropriately according to the control flow logic specified by the opcode and operands.

2. **Timestamp Progression**: The timestamps progress in a strictly increasing order, i.e., $\mathsf{timer}_{i+1} > \mathsf{timer}_i$. Typically, timer_{i+1} is close to $\mathsf{timer}_i + 1$, but large gaps are disallowed, ensuring that the difference $\mathsf{timer}_{i+1} - \mathsf{timer}_i$ does not exceed a reasonable bound to avoid field overflow or inconsistencies.

Thus, the program counter and timestamps are consistent if and only if $M(x) = 0$, verifying the correctness of the execution bus and ensuring that all updates adhere to the defined execution flow. □

1.10 Correctness of the Modular VM Execution

The modular architecture of our virtual machine ensures that the program executes correctly by leveraging the interactions between the chips and the three core buses: the **Program Bus**, the **Memory Bus**, and the **Execution Bus**. This section formally states and proves the correctness of the modular VM execution.

Overview of the Design In the modular VM, each chip is responsible for fetching, executing, and updating its own instructions independently. The system operates under the following principles:

- Each chip fetches its instruction at a unique timestamp timer_i which is received from the **Program Bus**.
- Memory access timing is enforced through intermediate checkpoints $t_{\text{checkpoint},j}$ between timer_i and timer_{i+1}, following the ordering dictated by the instruction's opcode specification.
- The program counter and timestamps are updated autonomously by each chip, with global consistency ensured via the **Execution Bus**.
- The **Memory Bus** ensures that all memory accesses (reads/writes) are consistent across all chips.

The guarantees provided by the three buses collectively ensure that the entire program executes correctly in the modular VM.

Theorem 10 (Correct Execution of the Modular VM)**.** *Let the virtual machine operate under a modular architecture equipped with the **Program Bus**, the **Memory Bus**, and the **Execution Bus**. Suppose:*

- *Each chip executes its assigned instructions atomically within the time intervals* $[\mathsf{timer}_i, \mathsf{timer}_{i+1}]$.
- *For each atomic instruction execution, intermediate checkpoints for memory accesses $t_{checkpoint,j}$ are placed between timer_i and timer_{i+1}, following the ordering dictated by the instruction's opcode specification.*
- *For each atomic instruction execution, each chip performs all non-memory operations correctly as dictated by the instruction's opcode specification.*
- *The chips interact correctly with the three buses, ensuring:*
  1. *Correct instruction fetching from ROM via the **Program Bus**, as guaranteed by theorem 8.*
  2. *Read-write consistency across RAM, sequenced by $t_{checkpoint}$ via the **Memory Bus**, as guaranteed by theorem 7.*

3. *Correct program counter updates sequenced by* timer *via the* ***Execution Bus****, as guaranteed by theorem 9.*

Then the entire program in the virtual machine executes correctly, satisfying all conditions of correct execution as defined in definition 14.

Proof. To prove the theorem, we analyze how the interaction with each bus ensures the conditions of definition 14:

1. **Global Timer Association:** Each chip executes its instructions independently and updates the timestamp via the **Execution Bus**. By theorem 9, the timestamps timer_i and timer_{i+1} satisfy $\mathsf{timer}_{i+1} > \mathsf{timer}_i$, ensuring a strictly increasing global timer sequence. The condition that the two timers do not differ by a large value (exceeding the field characteristic) ensures the global timer is consistent and bounded.

2. **Instruction Execution Timing:** Each instruction spans the interval $[\mathsf{timer}_i, \mathsf{timer}_{i+1}]$, with intermediate memory accesses occurring at checkpoints $t_{\text{checkpoint},j}$. The ordering of these checkpoints is enforced by the chip according to the specification of the instruction's opcode. The execution timing within each chip is inherently enforced by its design, ensuring that all intermediate events (memory accesses, computations, etc.) occur within the specified interval.

3. **Instruction Fetch Correctness:** By theorem 8, the **Program Bus** ensures that every instruction fetched by each chip matches the program's sequence in ROM. Specifically, the fetched instruction $(\mathsf{opcode}, \text{operands})$ corresponds to the value stored at the program counter address (pc_i) in ROM, guaranteeing instruction fetch correctness.

4. **Memory Access Timing and Consistency:**

   - **Enforced directly by each chip:** All memory accesses (reads or writes) occur at valid intermediate checkpoints $t_{\text{checkpoint},j}$, satisfying the condition:

   $$\mathsf{timer}_i < t_{\text{checkpoint},1} < t_{\text{checkpoint},2} < \cdots < t_{\text{checkpoint},k} < \mathsf{timer}_{i+1}.$$

   - **Enforced directly by each chip:** The ordering of these memory accesses corresponds to the instruction's opcode specification.
   - By theorem 7, the **Memory Bus** ensures that for each memory read, the value returned corresponds to the most recent write to the same address, ensuring read-write consistency.

5. **Program Counter Update Correctness:** Each chip ensures that the program counter (pc_i) is updated correctly after each instruction execution. Furthermore, by theorem 9, the **Execution Bus** ensures that the updates are sequenced according to the global timer acorss all the chips in the modular VM.

Since all conditions of definition 14 are satisfied through the interactions with the three buses and the atomic execution of instructions within each chip, the VM program must execute correctly in our modular VM. □

References

[BEG+94] Manuel Blum, W. Evans, Peter Gemmell, Sampath Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 09 1994. `doi:10.1007/BF01185212`.