Draft: axVM STARK Architecture

1 Preliminaries

1.1 Notations

In this document, we adopt the following notations:

- **Base Field** $\mathbb{F}$: The base field is the prime field $\mathbb{F}_{2^{31}-1}$, also known as the *Mersenne-31 bit prime field*. This field contains $2^{31}-1$ distinct elements, where each element is represented as an integer modulo $2^{31}-1$.
- **Extension Field** $\mathbb{F}_{\mathbf{ext}}$: We denote by $\mathbb{F}_{\text{ext}}$ the degree-4 extension field of the base field $\mathbb{F}$. The extension field $\mathbb{F}_{\text{ext}}$ is defined as $\mathbb{F}_{(2^{31}-1)^4}$, and elements in this field can be expressed as linear combinations of the form $a_0 + a_1\theta + a_2\theta^2 + a_3\theta^3$, where $a_0, a_1, a_2, a_3 \in \mathbb{F}$, and θ is the generator of the extension field.
- **Generator θ of the Extension Field**: We denote by $\theta \in \mathbb{F}_{\text{ext}}$ the generator of the degree-4 extension field $\mathbb{F}_{\text{ext}}$. This generator is used to express all elements of $\mathbb{F}_{\text{ext}}$ as polynomials of degree up to 3 in θ.
- **Subgroup** H: 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 the evolution of a computation over time. The AIR trace matrix captures the values of different variables or registers 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 (or computation step) and* j *indicates the column (or variable).*

- ***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. Each column corresponds to a specific variable or register in the computation, with the total number of variables (or registers) being represented by* m *columns.*
- ***Cell Values****: Each cell in the AIR trace matrix contains an element from the base field* $\mathbb{F}_{2^{31}-1}$*. These values represent the state of the variables or registers at each step of the computation. The cells hold values from* $\mathbb{F}_{2^{31}-1}$ *unless specified otherwise.*
- ***Extension Field Representation****: If an element from the extension field* $\mathbb{F}_{ext}$ *(the degree-4 extension of* $\mathbb{F}_{2^{31}-1}$*) is used in the computation, it is represented by four continuous cells across a single row. These four cells store the coefficients of the extension field element, where each coefficient is an element of the base field* $\mathbb{F}_{2^{31}-1}$*.*

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* 3*.*

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 $p = 2^{31} - 1$, 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:*

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

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* $1 - \binom{\sum_{j=1}^{p} k_i \cdot |H_j|}{2} \cdot \frac{\mathsf{max}_{i\in[1,k_i]}\ell_i}{|\mathbb{F}|} - \frac{t-1}{|\mathbb{F}_{ext}|}$*, 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. □

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. The virtual machine is designed to execute instructions using the RV32 instruction set. We define the state of the virtual machine, the transition function that evolves the machine state, and the correctness of the execution.

1.6.1 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.*
- $\mathsf{Mem}:(\mathbb{F}\times\mathbb{F})\to\mathbb{F}$ *is a map representing the memory, where:*
  - *Each memory address is represented as a tuple* $(space, pointer)$*.*
  - *The address space identifier* $(space\in\mathbb{F})$ *differentiates between various types of memory:*
    - $space=1$*: Registers.*
    - $space=2$*: Program memory.*
    - $space=3$*: Data memory.*
  - *The pointer* $(pointer\in\mathbb{F})$ *specifies the exact location within the chosen address space.*
  - *Each memory cell holds a value in the base field* $\mathbb{F}$*.*

Representation of Registers Registers are treated as part of the memory using address space 1. For example, the register reg_i is accessed as:

$$\mathsf{Mem}(1,i)\in\mathbb{F}.$$

This allows uniform handling of both registers and data memory cells through the memory map. Thus, loading and writing registers are equivalent to loading and writing data in memory.

Representation of Instructions Each instruction in the program memory (address space 2) is structured as a tuple:

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

where:

- $\mathsf{opcode} \in \mathbb{F}$ specifies one of the instructions in the RV32 instruction set.
- operands $= (\text{op}_1, \text{op}_2, \ldots, \text{op}_n)$ is a fixed-length vector of field elements representing addresses or values and immediates.

1.6.2 Definition of the Virtual Machine

A virtual machine VM consists of:

- An **initial state**, denoted by $\mathsf{State}_0 = (\mathsf{pc}_0, \mathsf{Mem}_0)$, where:
  - pc_0 is set to the starting address of the program.
  - Mem_0 is initialized with the program instructions in address space 2 and any initial data in address space 3.
- A **final state**, denoted by $\mathsf{State}_{\text{final}}$, reached when a terminating instruction (e.g., `HALT`) is executed.
- A **transition function** Step that defines how the virtual machine evolves from one state to another:

$$\mathsf{Step} : \mathsf{State} \to \mathsf{State}.$$

Transition Function Given the current state $(\mathsf{pc}, \mathsf{Mem})$, the transition function proceeds as follows:

1. **Instruction Fetch**:
$$(\mathsf{opcode}, \text{operands}) = \mathsf{Mem}(2, \mathsf{pc})$$
The instruction is fetched from program memory (address space 2) using the current program counter.
2. **Instruction Decode**: The fetched instruction is decoded into its opcode and operands based on the RV32 instruction set.
3. **Instruction Execution**: The virtual machine executes the instruction according to the decoded opcode:
   - For an arithmetic operation (e.g., addition):
   $$\mathsf{Mem}(1, \text{destination}) = \mathsf{Mem}(1, \text{op}_1) + \mathsf{Mem}(1, \text{op}_2)$$
   - For a load operation:
   $$\mathsf{Mem}(1, \text{destination}) = \mathsf{Mem}(3, \text{address})$$
   - For a store operation:
   $$\mathsf{Mem}(3, \text{address}) = \mathsf{Mem}(1, \text{source})$$
   - For a jump or branch instruction:
   $$\mathsf{pc}_{\text{new}} = \text{target address}$$
4. **Program Counter Update**: If no jump or branch instruction is executed, the program counter is incremented to fetch the next sequential instruction:
$$\mathsf{pc}_{\text{new}} = \mathsf{pc} + 4$$
5. **State Update**: The new state is:
$$\mathsf{State}_{\text{new}} = (\mathsf{pc}_{\text{new}}, \mathsf{Mem})$$

1.6.3 Correct Execution of the Virtual Machine

Definition 14 (Correct Execution)**.** *The execution of the virtual machine is correct if, starting from a given initial state* State_0*, it reaches a specified final state* State_{final} *such that there exists an execution trace:*

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

and the following conditions hold for each transition Step*.*

- *Every instruction is fetched correctly from data memory (address space* 2*) and decoded according to the RV32 instruction set.*
- *Register operations (address space* 1*) are correctly performed.*
- *Memory consistency is maintained:*
  - *All reads from data memory (address space* 3*) return the most recently written value.*
  - *All reads from program memory (address space* 2*) return the most recently written value.*
  - *All reads from register memory (address space* 1*) return the most recently written value.*
- *The program counter updates correctly based on the type of instruction (e.g., sequential increment or jump).*

1.6.4 Distributed Execution Across Multiple Chips

In our virtual machine, we utilize a distributed 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 RV32 instruction set.

Chip-Specific Instruction Fetching Each chip is responsible for independently fetching instructions from program memory (address space 2). For simplicity, for now we assume that each chip implicitly fetches only those instructions that correspond to the opcodes it is specialized to handle.

Overview of Chip Specialization

- **Arithmetic Operations**: Each arithmetic opcode (e.g., addition, subtraction, multiplication) is managed by a dedicated chip. This specialization allows for optimized execution of these operations.
- **Other RV32 Instructions**: Instructions such as logical operations, immediate arithmetic, and shifts are similarly distributed among specialized chips.
- **Locally Handled Instructions**: Each chip is capable of handling **memory read/write** and **branch/jump** instructions locally.

RV32 Opcode Distribution The table below lists all the opcodes in the RV32 instruction set, indicating which opcodes are handled by specialized chips and which are managed locally by all chips.

1.7 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 distributed execution of the virtual machine, ensuring consistency in memory access, instruction fetching, program counter updates, and opcode execution.

Required Columns for AIR Traces Each AIR trace for a chip includes the following columns:

Opcode	Description	Chip Responsibility
0b0110011	Arithmetic (ADD, SUB, MUL, DIV)	Specialized Chip
0b0010011	Immediate Arithmetic (ADDI, ANDI, ORI)	Specialized Chip
0b0000011	Load (LB, LH, LW)	Handled Locally
0b0100011	Store (SB, SH, SW)	Handled Locally
0b1100011	Branch (BEQ, BNE, BLT, BGE)	Handled Locally
0b1101111	Jump and Link (JAL)	Handled Locally
0b1100111	Jump and Link Register (JALR)	Handled Locally
0b0110111	Load Upper Immediate (LUI)	Handled Locally
0b0010111	Add Upper Immediate to PC (AUIPC)	Handled Locally
0b1110011	System Instructions (ECALL, EBREAK)	Specialized Chip
0b0001111	Fence Instructions (FENCE, FENCE.I)	Specialized Chip

Table 1: Distribution of RV32 Opcodes Across Chips

Column Name	Description
$\mathbf{Tr}_{\mathsf{pc}}$	Program counter (pc)
$\mathbf{Tr}_{\text{opcode}}$	Opcode of the instruction
$\mathbf{Tr}_{\text{operands}}$	Operands for the instruction
$\mathbf{Tr}_{\text{reg}}$	Register values (address space 1)
$\mathbf{Tr}_{\text{mem\_addr}}$	Memory address (address space 3)
$\mathbf{Tr}_{\text{mem\_val}}$	Data memory value
$\mathbf{Tr}_{\text{curr\_timestamp}}$	Current timestamp for each virtual machine state
$\mathbf{Tr}_{\text{prev\_timestamp}}$	Timestamp indicating the last time certain memory cell is accessed

Table 2: Columns in the AIR Trace for Each Chip

Explanation of Columns

- **Program Counter ($\mathbf{Tr}_{\mathsf{pc}}$)**: Records the program counter value for each computation step, ensuring the correct sequence of instruction fetching.
- **Opcode ($\mathbf{Tr}_{\mathbf{opcode}}$)**: This column defaults to the chip's specialized opcode. However, it changes dynamically to handle other opcodes like memory read/write and branch/jump instructions.
- **Operands ($\mathbf{Tr}_{\mathbf{operands}}$)**: Stores the operands used by the current instruction, represented as a fixed-length vector.
- **Registers ($\mathbf{Tr}_{\mathbf{reg}}$)**: Represents values read from or written to registers, treated as memory accesses within address space 1.
- **Memory Address ($\mathbf{Tr}_{\mathbf{mem\_addr}}$)** and **Memory Value ($\mathbf{Tr}_{\mathbf{mem\_val}}$)**:
  - $\mathbf{Tr}_{\text{mem\_addr}}$ records the address being accessed in data memory (address space 3).
  - $\mathbf{Tr}_{\text{mem\_val}}$ captures the value being read or written, represented as a vector with variable length to accommodate different data sizes.
- **Current Timestamp ($\mathbf{Tr}_{\mathbf{curr\_timestamp}}$)** and **Previous Timestamp ($\mathbf{Tr}_{\mathbf{prev\_timestamp}}$)**: They together ensure that operations, especially memory reads and writes, occur in the correct temporal order, allowing the system to enforce memory consistency.

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

1.8.1 Memory Bus

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

In the offline memory checking, each memory operation is recorded in the form $(\text{mem\_addr}, \text{mem\_val}, \text{timestamp})$, where:

- $\text{mem\_addr} = (\text{space}, \text{pointer})$.
- mem_val 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 based on data size.
- timestamp $\in \mathbb{F}$ is either the previous or new timestamp of the value which is being read or written. Here, previous means the last time (prev_timestamp) mem_addr has been accessed. Subsequently, prev_timestamp is replaced with the current timestamp which corresponds to the current state timestamp curr_timestamp.

Grouping of Memory Access Records Memory access records $(\text{mem\_addr}, \text{mem\_val}, \text{timestamp})$ are grouped by the length of mem_val. For each distinct length, we define a vector of length $\ell = |\text{mem\_val}| + 3$, which includes:

$$(\text{mem\_addr}, \text{mem\_val}, \text{timestamp})$$

We will then define a collection of multisets encompassing all different dimensions based on the above grouping.

Verification Using Multisets To ensure that each memory operation correctly aligns with the most recent write to the same address, for each dimension ℓ, we utilize two distinct multisets:

- **Receive Multiset** $M^{\ell}_{\text{receive}}$: Logs the previous value associated with each memory address, recorded at the previous timestamp (prev_timestamp). For each length $\ell = |\text{mem\_val}| + 3$, we define:

  $M^{\ell}_{\text{receive}}(\text{mem\_addr}\dots) = \#\{(\text{mem\_addr}\dots) \mid \text{mem\_val is most recent value read frommem\_addr at prev\_timestamp}\}.$

- **Send Multiset** M^{ℓ}_{send}: Records the new values written to memory, using the current timestamp (curr_timestamp). For each length $\ell = |\text{mem\_val}| + 3$, we define:

  $M^{\ell}_{\text{send}}(\text{mem\_addr}\dots) = \#\{(\text{mem\_addr}\dots) \mid \text{mem\_val is new value written to mem\_addr at current state}\}.$

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:

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

     $$M^{\ell}_{\text{receive}}(\text{mem\_addr}, \text{mem\_val}, \text{prev\_timestamp}),$$

     where:
     - mem_val is the return value previously stored at mem_addr.
     - prev_timestamp is the timestamp of the last access, which is received as a non-deterministic input.
  2. The chip then confirms the read operation by adding to the **Send Multiset**:

     $$M^{\ell}_{\text{send}}(\text{mem\_addr}, \text{mem\_val}, \text{curr\_timestamp}),$$

     ensuring that:

     $$\text{curr\_timestamp} > \text{prev\_timestamp}.$$

- **Write Operation**:

1. Before writing a new value new_val, the chip logs the previous state by adding to the **Receive Multiset**:
$$M^{\ell}_{\text{receive}}(\text{mem\_addr}, \text{mem\_val}, \text{prev\_timestamp}),$$
where:
   - mem_val is the previous value stored at mem_addr.
   - prev_timestamp is its timestamp of the last access, provided as a non-deterministic input.
2. After updating the memory with the new value, the chip adds the updated entry to the **Send Multiset**:
$$M^{\ell}_{\text{send}}(\text{mem\_addr}, \text{new\_val}, \text{curr\_timestamp}),$$
ensuring that:
$$\text{curr\_timestamp} > \text{prev\_timestamp}.$$

Balancing Initial Sends and Final Receives This directly relates to subsubsection 1.6.2, where the initial and final states of the virtual machine are defined. For a virtual machine VM with:

- 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 instructions in address space 2 and data in address space 3.
- A **final state** $\mathsf{State}_{\text{final}}$, reached upon execution of a terminating instruction (e.g., `HALT`).

To enforce memory consistency, we also include:

- An **initial entry** $(\text{mem\_addr}, \text{mem\_val}_{\text{init}}, 0)$ in M_{send} for each accessed address, representing the initial state where all timestamps are initialized to 0.
- A **final entry** $(\text{mem\_addr}, \text{mem\_val}_{\text{final}}, \text{mem\_timestamp}_{\text{final}})$ in M_{receive} for each address, where $\text{mem\_val}_{\text{final}}$ is the expected final value of its final access at time $\text{mem\_timestamp}_{\text{final}}$.

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:

- $P_1(x), \ldots, P_\ell(x) = (\mathbf{Tr}_{mem\_addr}(x), \mathbf{Tr}_{mem\_val}(x), \mathbf{Tr}_{curr\_timestamp}(x))$ *or* $\mathbf{Tr}_{prev\_timestamp}(x))$ *are individual columns from the AIR trace.*
- $M(x)$ *is the multiplicity column computed as the difference between the Receive and Send multisets.*

Theorem 6 (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. Then every read from data memory (address space 3) and register memory (address space 1) returns the most recently written value, if and only if the collection of $M(x)$ are empty multisets.*

1.8.2 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**).

Similar to the memory bus, the Program Bus uses multisets to enforce consistency in the sequence of fetched instructions from the program memory, ensuring that each fetch operation aligns with the correct program counter and timestamp. The slight difference is that since the program memory is read-only, we will disallow any write operations.

Program Access Records Program access records $(\mathbf{pc}, \text{opcode}||\text{operands}, \text{timestamp})$ are grouped as follows:,

- **pc**: The program counter representing the address from which the instruction is fetched.
- opcode||operands: The full instruction data, consisting of an operation code followed by its operands.
- timestamp: The timestamp indicating the previos time or the current time where the given instruction is fetched from the program memory.

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 ($\mathsf{Airbus}_{\mathsf{Mem}}$). Therefore, a single dimension suffices to represent all entries in the Program Bus.

We then similarly define M_{receive} and M_{send}, two distinct multisets which holds the program access records to ensure that the fetched instructions are consistent with the program memory. Each chip will perform read operations only. Finally, we will again balance the receive and send multisets base on the **initial state** $\mathsf{State}_0 = (\mathsf{pc}_0, \mathsf{Mem}_0)$ and **final state** $\mathsf{State}_{\text{final}}$.

Definition 16 (Program Bus ($\mathsf{Airbus}_{\mathsf{Prog}}$))**.** *The Program Bus is a specialized AIR Bus used to verify the consistency of instruction fetching. It consists of a single multiset $M \in \mathsf{Airbus}_{\mathsf{Prog}}$, defined as follows:*

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

where:

- $P_1(x), \ldots, P_\ell(x) = (\mathbf{Tr}_{pc}(x), \mathbf{Tr}_{opcode}(x)||\mathbf{Tr}_{operands}(x), \mathbf{Tr}_{curr/prev\_timestamp}(x))$ *are individual columns from the AIR trace.*
- $M(x)$ *is the multiplicity column representing the difference between the Receive and Send multisets.*

Theorem 7 (Instruction Fetch Consistency in Program Bus)**.** *Given the initial state State_0 and 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 (address space 2) are consistent with the initial state if and only if $M(x)$ is the empty multiset.*

1.8.3 Execution Bus

The Execution Bus ($\mathsf{Airbus}_{\mathsf{Exe}}$) ensures that the program counter (pc) and timestamps are correctly updated across all chips in the system.

Updating the Program Counter In our distributed architecture, each chip fetches instructions and updates its own program counter (pc) autonomously. For example, for most instructions in the RV32 architecture, the program counter is updated sequentially:

$$\mathsf{pc}_{\text{new}} = \mathsf{pc} + 4,$$

However, instructions such as branches or jumps may update the program counter differently based on specific conditions or target addresses.

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 actual pairs $(\mathsf{pc}, \text{curr\_timestamp})$ observed in the system, representing the program counter and timestamp values from the current states. Formally:

$$M_{\text{receive}} = \{(\mathsf{pc}, \text{curr\_timestamp})\}.$$

- **Send Multiset** M_{send}: Captures the expected pairs $(\mathsf{pc}_{\text{new}}, \text{next\_timestamp})$ after executing an instruction. These pairs represent the updated program counter and the timestamp. Formally:

$$M_{\text{send}} = \{(\mathsf{pc}_{\text{new}}, \text{next\_timestamp})\},$$

  where next_timestamp is usually curr_timestamp + 1, which indicates the new program counter value at the subsequent state.

Definition 17. *The Execution Bus is defined as the multiset which is taken as the difference between these multisets:*

$$M(x) = M_{receive}(\mathbf{Tr}_{pc}(x), \mathbf{Tr}_{curr\_timestamp}(x)) - M_{send}(\mathbf{Tr}_{pc\_new}(x), \mathbf{Tr}_{curr\_timestamp}(x) + 1),$$

where $\mathbf{Tr}_{\text{pc\_new}}(x)$ is a virtual column indicating the new program counter values.

Theorem 8 (Program Counter Consistency in the Execution Bus)**.** *Given the initial state* State_0 *and final state* State_{final} *of the virtual machine, let the two multisets* M_{send} *and* $M_{receive}$ *be constructed as above. Assuming that the local updates to pc_new are correct, then the updates to the program counter (*pc*) are consistent and correct across all chips in the system, with initial program counter matching* pc_0*, if and only if* $M(x)$ *is the empty multiset.*

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

$$\mathsf{pc} = \mathsf{pc}_{\text{new}}, \quad \text{curr\_timestamp} = \text{next\_timestamp}.$$

This ensures:

1. Sequential updates are correctly reflected as $\mathsf{pc}_{\text{new}} = \mathsf{pc} + 4$ or adjusted appropriately for branch/jump instructions.
2. The timestamps maintain the correct order, i.e., next_timestamp = curr_timestamp + 1.

□

1.9 Correct Execution of the Virtual Machine

Corollary 9 (Correct Execution of the Virtual Machine)**.** *Given the initial state* State_0 *and final state* State_{final} *of the virtual machine, if the following conditions are satisfied:*

- *The* ***Read-Write Consistency in Memory Bus*** *(Theorem 6) ensures that all reads from data memory (address space* 3*) and register memory (address space* 1*) return the most recently written value.*
- *The* ***Instruction Fetch Consistency in Program Bus*** *(Theorem 7) ensures that all instructions fetched from program memory (address space* 2*) are consistent with the initial state.*
- *The* ***Program Counter Consistency in the Execution Bus*** *(Theorem 8) guarantees that the updates to the program counter (*pc*) are correct and consistent across all chips.*
- *Each chip enforces the correct parsing of its specialized opcodes and operands, and performs register operations (address space* 1*) correctly.*

then the execution of the virtual machine is correct according to the definition of correct execution (14). Specifically, the machine reaches a specified final state State_{final} *starting from the initial state* State_0 *through a sequence of correctly executed transitions:*

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

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.