AirAssembly

A low-level language for encoding Algebraic Intermediate Representation (AIR) of computations.

https://github.com/GuildOfWeavers/AirAssembly

Outline

AIR arithmetization

AirAssembly language

AirAssembly examples

Types of computations

	Circuit Computations	Machine Computations
Representation	Arithmetic circuits	Transition functions
Arithmetization	Rank 1 Constraint Satisfiability (R1CS)	Algebraic Intermediate Representation (AIR)
Benefits	Easy reduction from GPC, simple composition	Fast proof generation and verification times
Languages	Snarky, Circom, ZoKrates	AirScript, AirAssembly
Used in	Most SNARKs (Groth16 etc.)	STARKs

Algebraic Intermediate Representation (AIR)

Main concepts

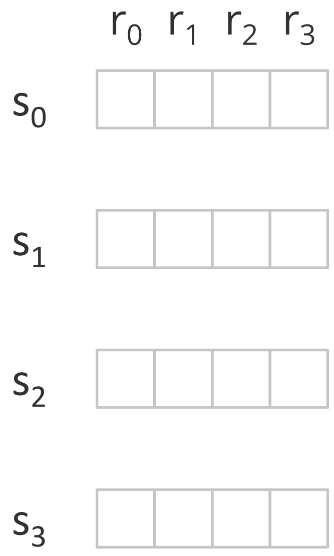
- Computation state

r_0	r_1	r_2	r_3

Algebraic Intermediate Representation (AIR)

Main concepts

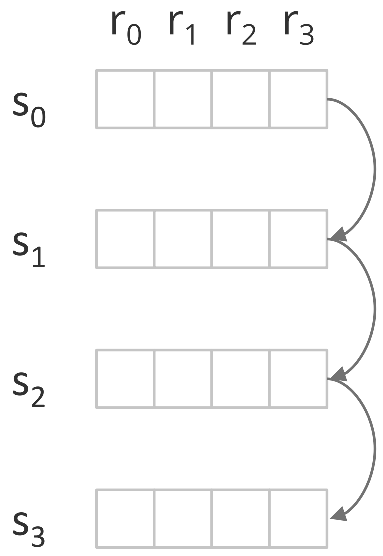
- Computation state
- Execution trace



Algebraic Intermediate Representation (AIR)

Main concepts

- Computation state
- Execution trace
- Transition function



$$s_1 = f(s_0)$$

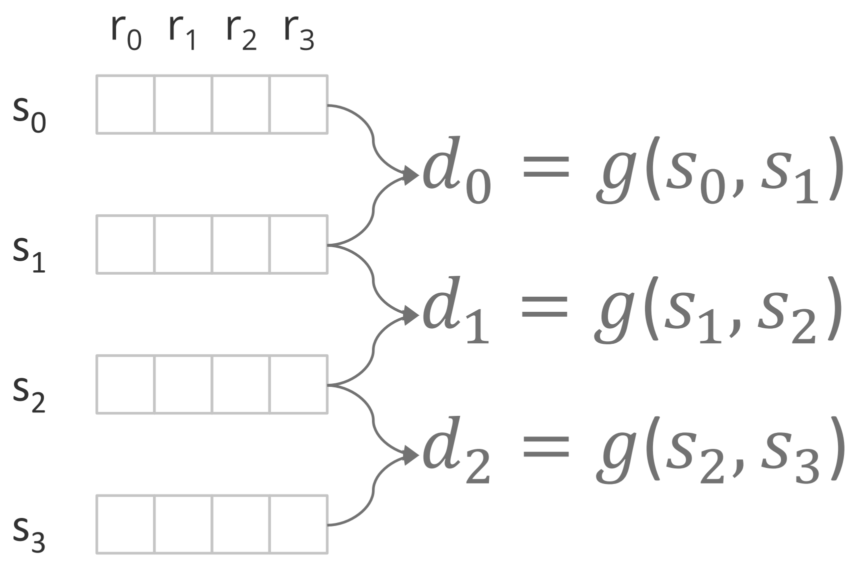
$$s_2 = f(s_1)$$

$$s_3 = f(s_2)$$

Algebraic Intermediate Representation (AIR)

Main concepts

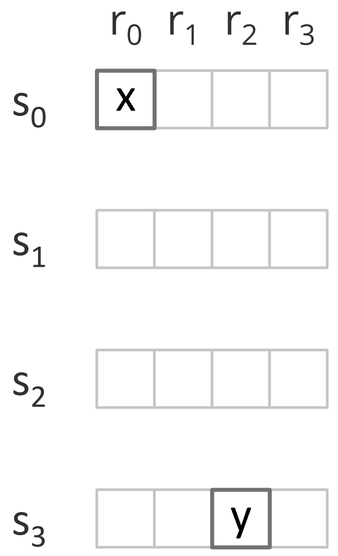
- Computation state
- Execution trace
- Transition function
- Transition constraints



Algebraic Intermediate Representation (AIR)

Main concepts

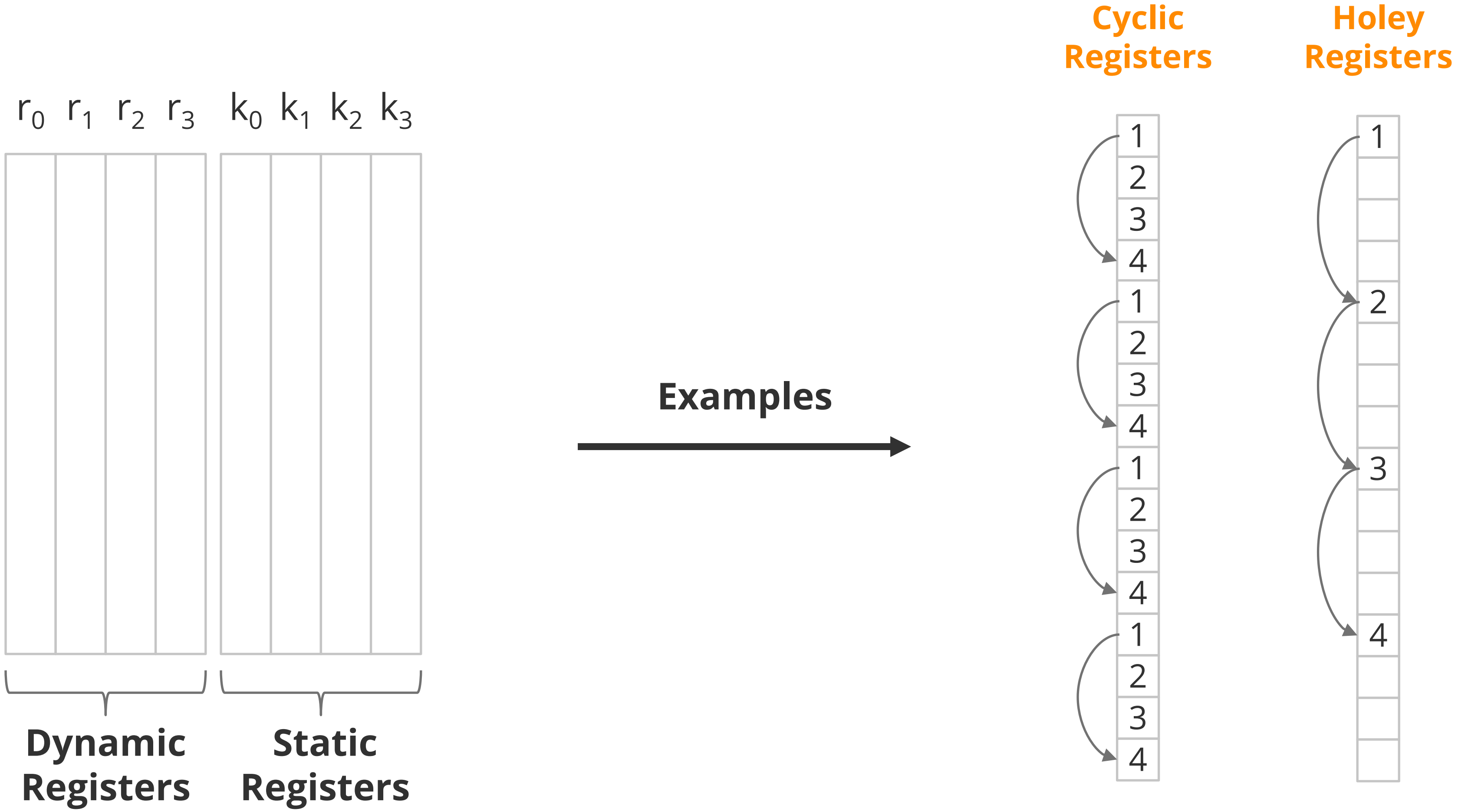
- Computation state
- Execution trace
- Transition function
- Transition constraints
- Boundary constraints



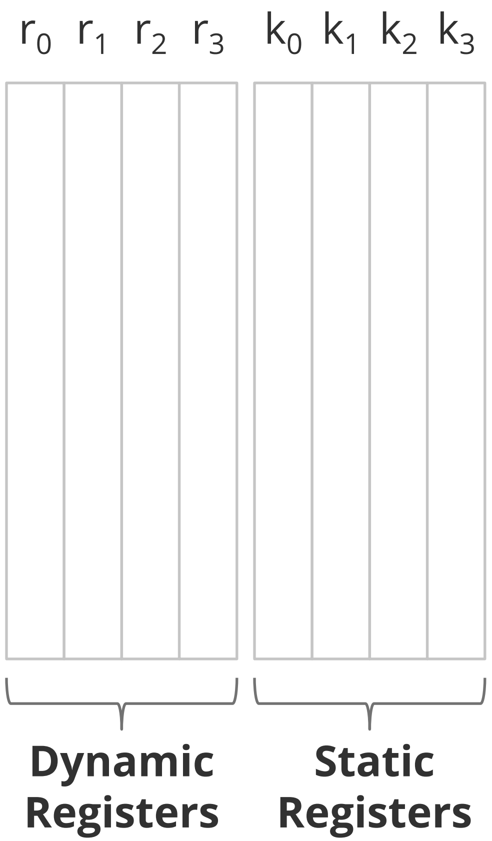
$r_{0,0} = x$

$r_{2,3} = y$

Execution trace



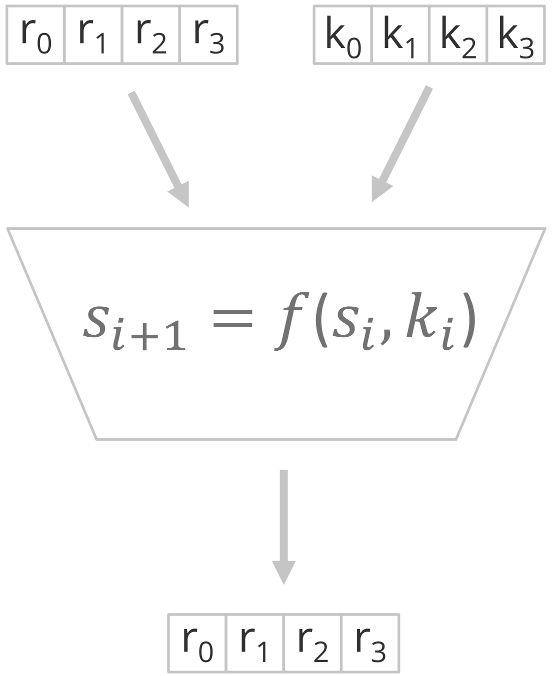
Execution trace



Considerations

- Trace length must be 2^n for some n
- Static registers can be public or secret
- Field must have high order roots of unity

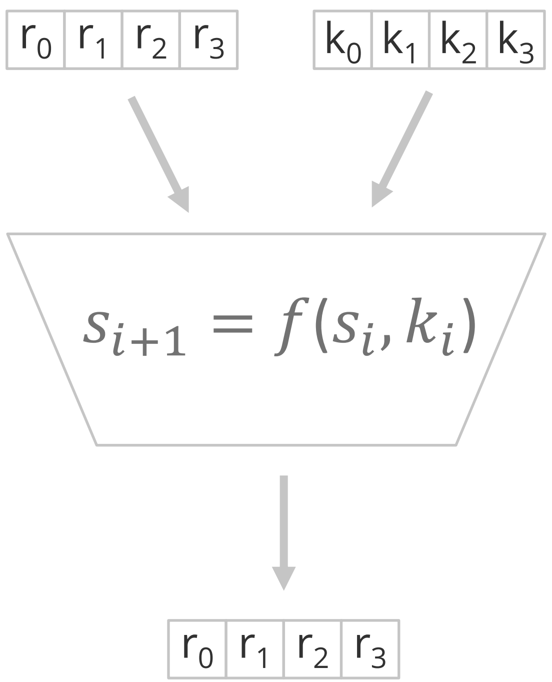
Transition function



Arithmetic Switching

$$s_{i+1} = \underbrace{k_{i,0} * f_1(s_i, k_i)}_{\textbf{Option 1}} + \underbrace{(1 - k_{i,0}) * f_2(s_i, k_i)}_{\textbf{Option 2}}$$

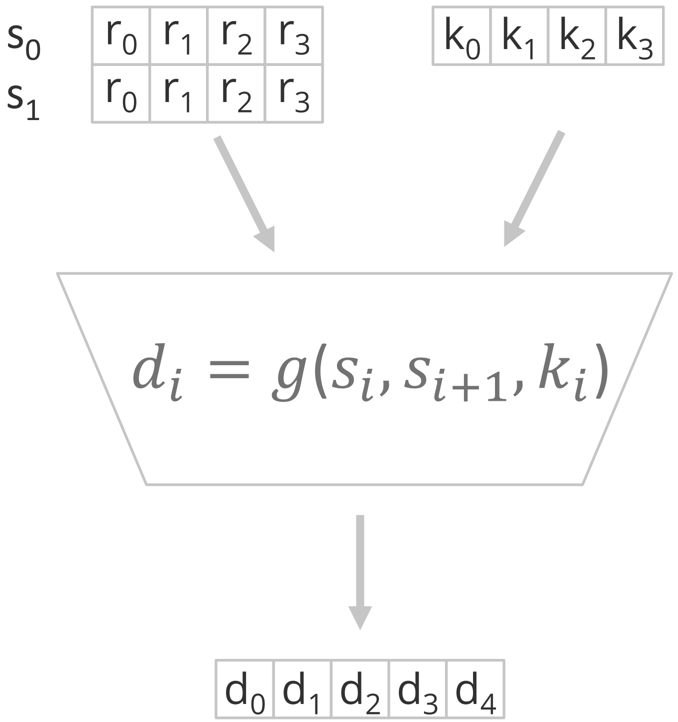
Transition function



Considerations

- All arithmetic operations are allowed
- Degree grows when:
  - A register is raised to a power
  - Two registers are multiplied
- "Long-range" transition functions work with more than 2 consecutive states

Transition constraints



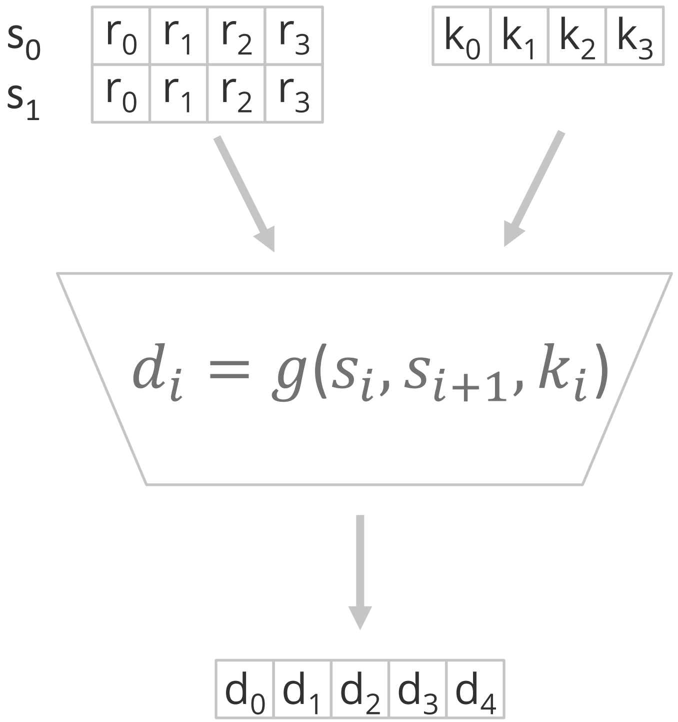
Frequently just

$$d_i = s_{i+1} - f(s_i, k_i)$$

But can also be more complex

$$d_i = g_1(s_{i+1}, k_i) - g_2(s_i, k_i)$$

Transition constraints



Considerations

- Division not allowed (but can be emulated)
- Degree may or may not be the same as degree of transition function
- “Long-range” constraints work with more than 2 consecutive states

Outline

AIR arithmetization

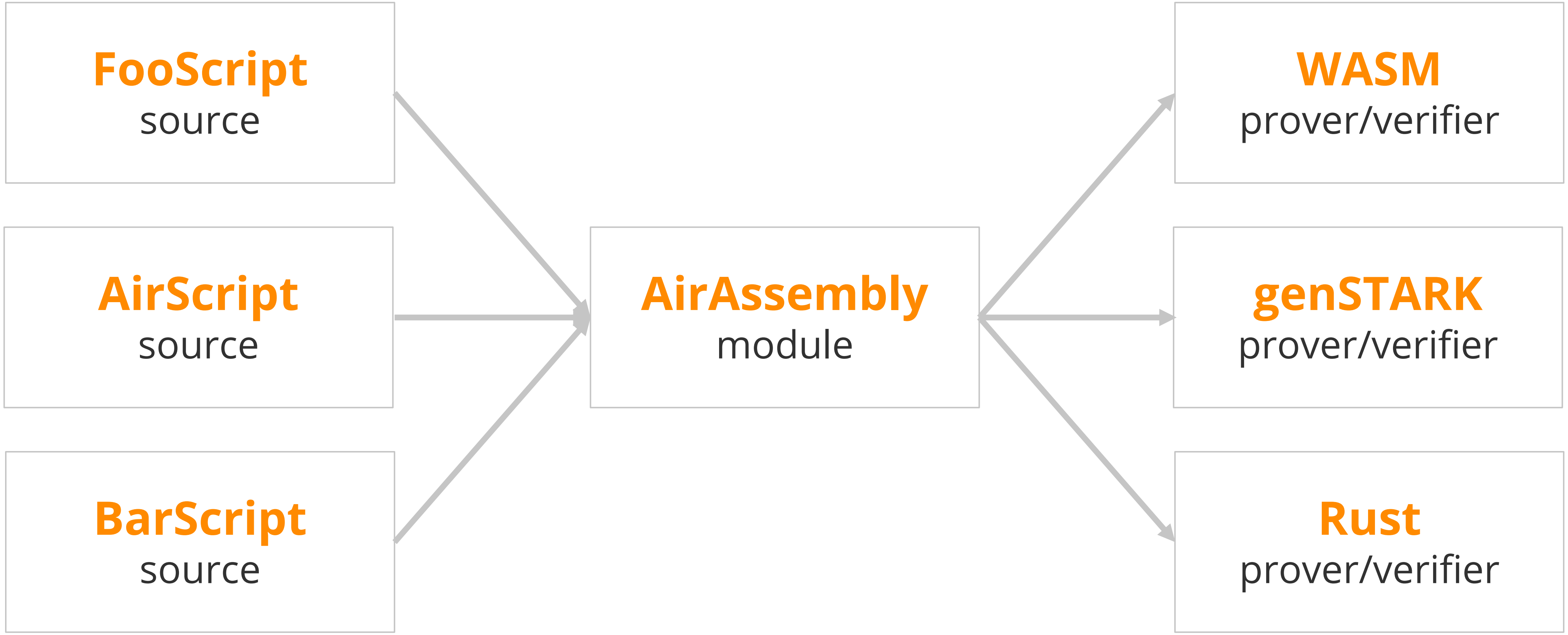
AirAssembly language

AirAssembly examples

AirAssembly aims to describe

1. Logic for generating execution trace
2. Logic for generating constraint evaluations
3. Logic for interpreting inputs

Toolchain model



AirAssembly syntax

S-expression-based syntax modeled after WebAssembly:

```
(operation
  <operand 1> <operand 2> ...)
```

3 data types
`scalar`, `vector`, `matrix`

8 arithmetic operations
`add`, `sub`, `mul`, `div`, `exp`, `prod`, `inv`, `neg`

2 vector operations
`get`, `slice`

1 function call expression
`call`

6 load/store operations
`load.const`, `load.param`, `load.local`, `load.trace`, `load.static`, `store.local`

Data types

```
(scalar 3)
```

new scalar → 3

```
(vector
  (scalar 1) (scalar 2))
```

new vector →

1	2

```
(vector
  (scalar 1)
  (vector
    (scalar 2) (scalar 3)))
```

new vector →

1	2	3

```
(matrix
  ((scalar 1) (scalar 2))
  ((vector
    (scalar 3) (scalar 4))))
```

new matrix →

1	2
3	4

Arithmetic operations

```
(add
  (scalar 1) (scalar 2))
```

add scalars → 3

```
(add
  (vector (scalar 1) (scalar 2))
  (vector (scalar 3) (scalar 4)))
```

add vector elements →

4	6

```
(add
 (vector (scalar 1) (scalar 2))
 (scalar 2))
```

add scalar to vector elements →

3	4

All arithmetic operations:

`add`, `sub`, `mul`, `div`, `exp`, `prod`, `inv`, `neg`

Product operation

```
(prod
  (vector (scalar 1) (scalar 2))
  (vector (scalar 3) (scalar 4)))
```

linear combination

11

```
(prod
  (matrix
    ((scalar 1) (scalar 2))
    ((scalar 3) (scalar 4)))
  (matrix
    ((scalar 5) (scalar 6))
    ((scalar 7) (scalar 8))))
```

matrix multiplication

19	22
43	50

```
(prod
  (matrix
    ((scalar 1) (scalar 2))
    ((scalar 3) (scalar 4)))
  (vector (scalar 1) (scalar 2)))
```

matrix-vector multiplication

5	11

Vector operations

```
(get
  (vector
    (scalar 1)
    (scalar 2)
    (scalar 3))
  1)
```

get vector element → 2

```
(slice
  (vector
    (scalar 1)
    (scalar 2)
    (scalar 3))
  1 2)
```

slice vector → | 2 | 3 |

AirAssembly module

```
module
  field
  constants
  functions
  components
    static registers
    trace initializer
    transition function
    constraint evaluator
```

Finite field

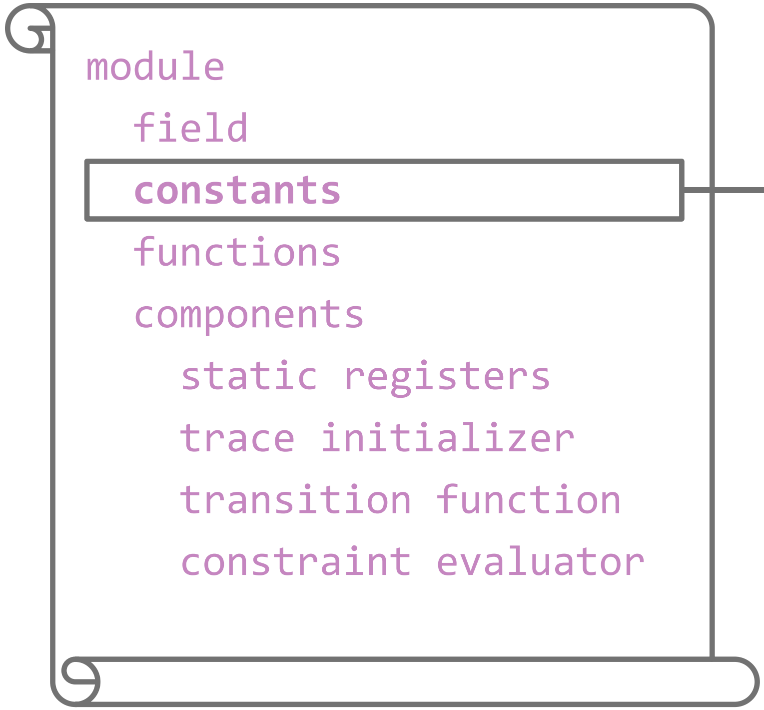
```
module
  field
  constants
  functions
  components
    static registers
    trace initializer
    transition function
    constraint evaluator
```

Finite field

Specifies the finite field to be used for all arithmetic operations within the module.

```
(field prime
  340282366920938463463374607393113505793)
```

Constants

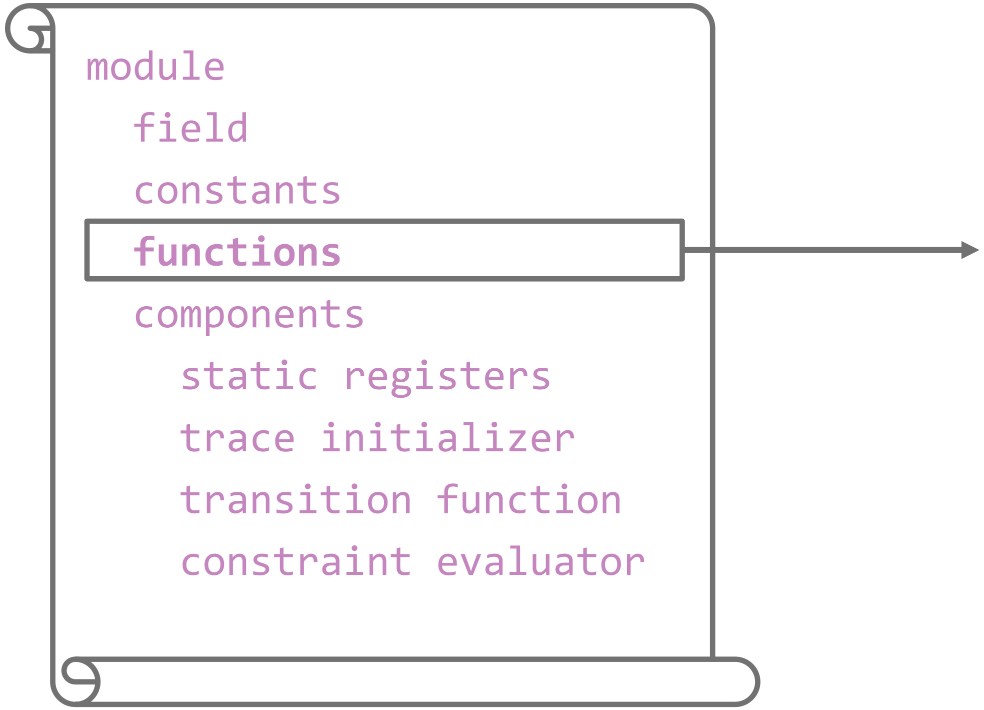


Module constants

Defines a set of constants which can be used in arithmetic operations within the module.

```
(const $foo scalar 123)
(const $bar vector 1 2 3 4)
(const $baz matrix (1 2) (3 4))
```

Functions



Module functions

Defines a set of functions which can be used to encapsulate common arithmetic expressions.

```
(function $mimcRound
  (result vector 1)
  (param $state vector 1) (param $key scalar)
  (add
    (exp (load.param $state) (scalar 3))
    (load.param $roundKey))))
```

Static registers

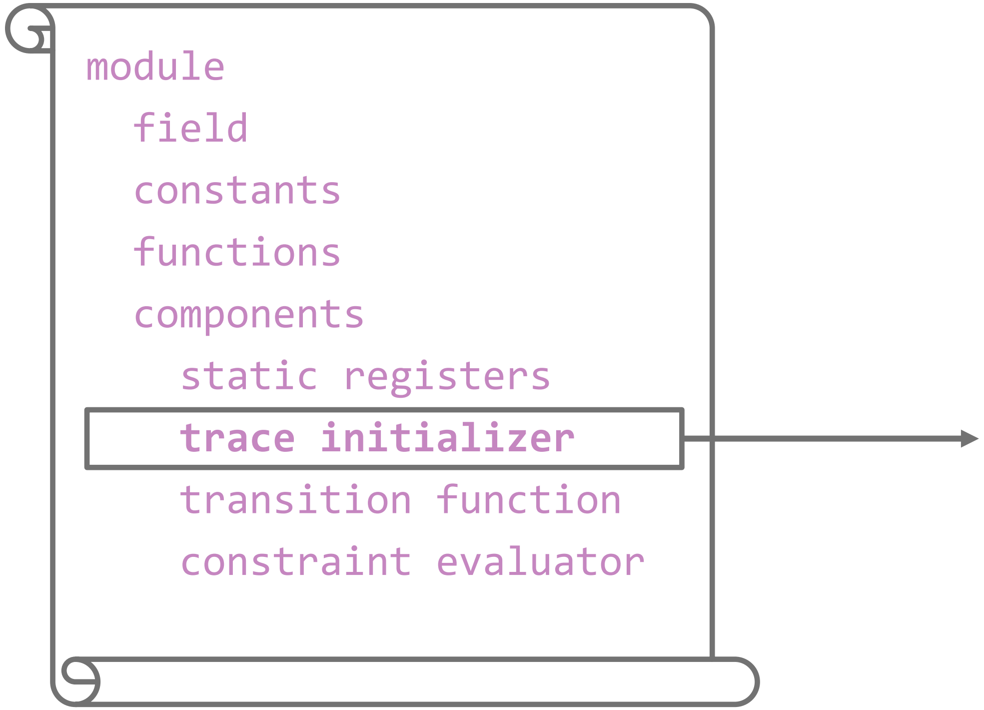
```
module
  field
  constants
  functions
  components
    static registers
    trace initializer
    transition function
    constraint evaluator
```

Static registers

Describes logic for building static registers, including logic for interpreting non-scalar inputs.

```
(static
  (input public (steps 64))
  (mask inverted (input 0))
  (cycle (prng sha256 0x4d694d43 64)))
```

Trace initializer

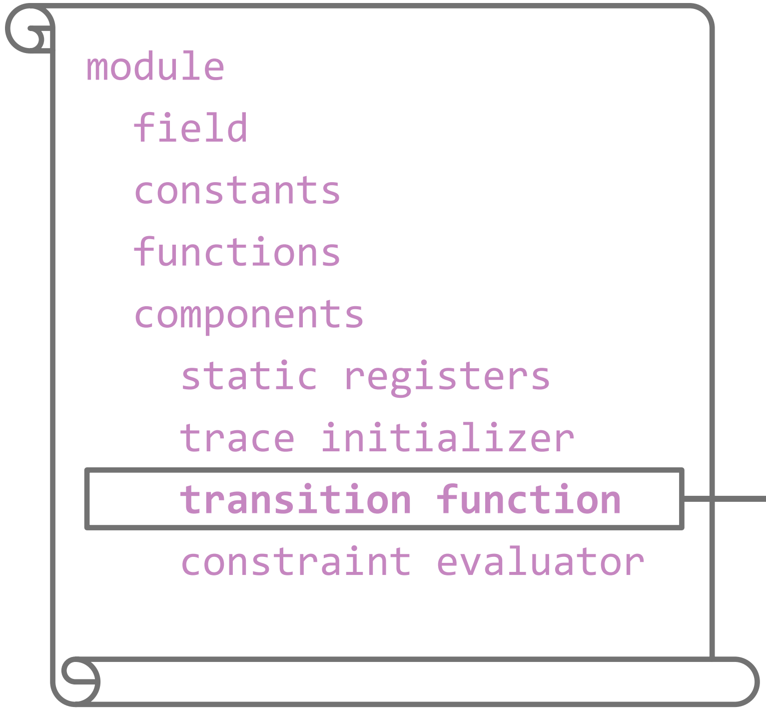


Trace initializer

Describes logic for initializing the first row of the execution trace, including logic for interpreting scalar inputs.

```
(init
  (param $seed vector 1)
  (load.param $seed))
```

Transition function

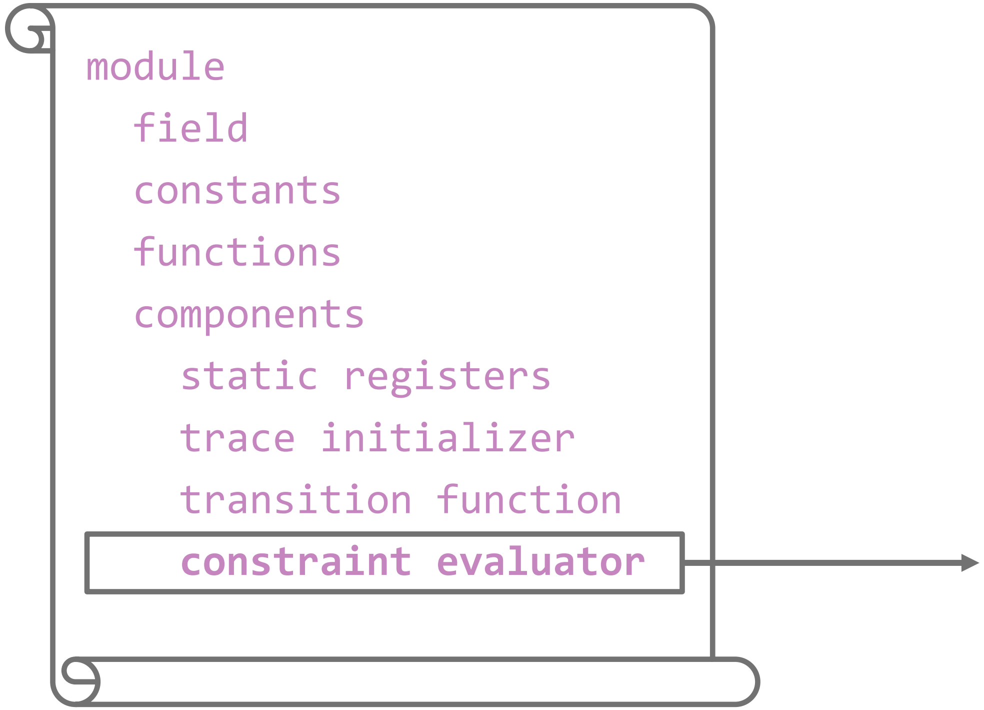


Transition function

Describes state transition logic for the computation. The value returned by the transition function becomes the next row in the execution trace table.

```
(transition
  (add
    (exp (load.trace 0) (scalar 3))
    (load.static 0)))
```

Constraint evaluator

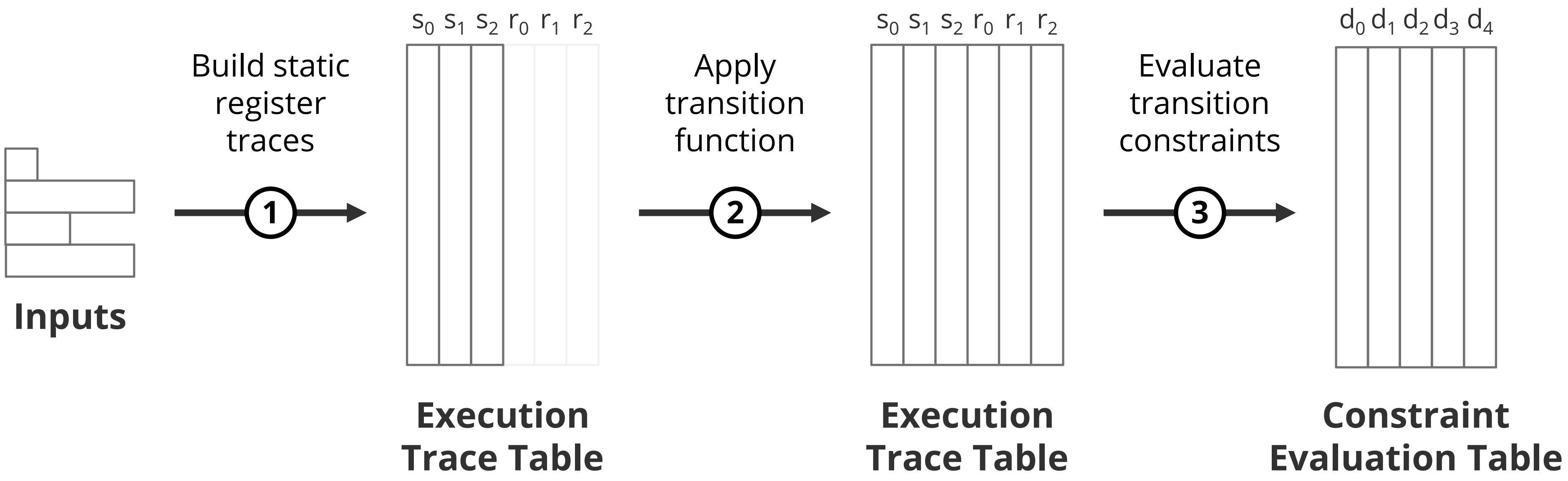


Constraint evaluator

Describes algebraic relation between steps of the computation. The value returned from the constraint evaluator becomes the next row in the constraint evaluation table

```
(evaluation
  (sub
    (load.trace 1)
    (add
      (exp (load.trace 0) (scalar 3))
      (load.static 0))))
```

AirAssembly execution



Building static register traces

```
(static
  (input public)
  (input public (parent 0) (steps 4))
  (mask inverted (input 1))
  (cycle 1 2 3 4))
```

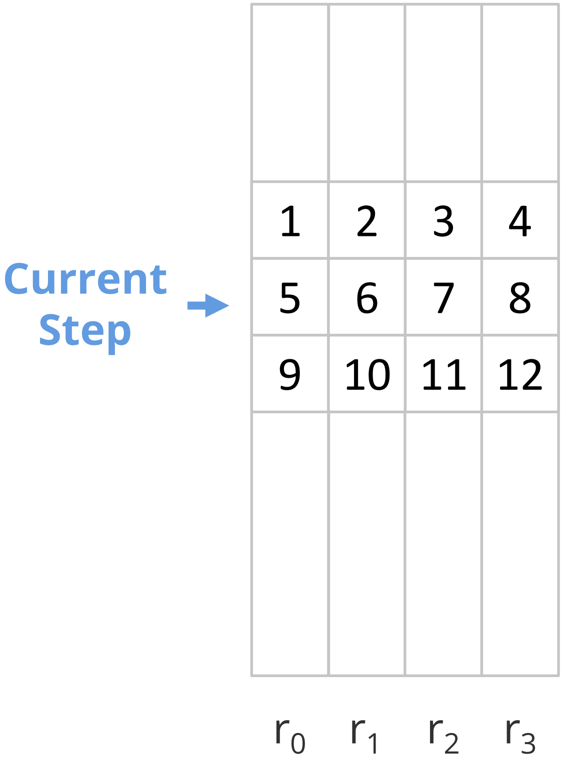
```
With inputs:

  Register 0: [1, 2]
  Register 1: [[3, 4], [5, 6]]
```

Static Register Traces

s_0	s_1	s_2	s_3
1	3	0	1
0	0	1	2
0	0	1	3
0	0	1	4
0	4	0	1
0	0	1	2
0	0	1	3
0	0	1	4
2	5	0	1
0	0	1	2
0	0	1	3
0	0	1	4
0	6	0	1
0	0	1	2
0	0	1	3
0	0	1	4

Loading trace register

Execution Trace



Load values of all registers at the current, next, and previous steps:

```
(load.trace  0) -> [5, 6, 7, 8]
(load.trace  1) -> [9, 10, 11, 12]
(load.trace -1) -> [1, 2, 3, 4]
```

Load value of the second register at the current step:

```
(get (load.trace 0) 1) -> 6
```

Outline

AIR arithmetization

AirAssembly language

$\boxed{\text{AirAssembly examples}}$

MiMC arithmetization

$$\underbrace{\underbrace{r_{i+1,0}}_{\text{Next state}} = \underbrace{r_{i,0}^3}_{\text{Current state}}}_{\text{1 dynamic register}} + \underbrace{\underbrace{k_{i,0}}_{\text{Round constant}}}_{\text{1 static register}}$$

MiMC execution trace

$$r_{i+1,0} = r_{i,0}^3 + k_{i,0}$$

Parameters

- Input value: 3
- Static register cycle: [1, 2, 3, 4]
- Computation steps: 64
- Field modulus: $2^{32} - 3 * 2^{25} + 1$

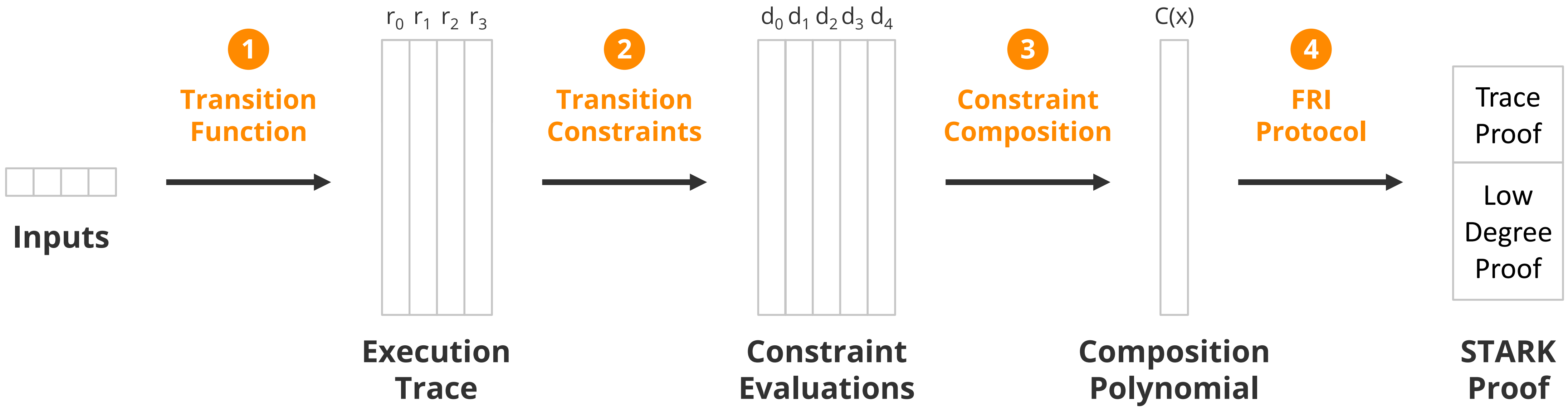
Step	k0	r0
0	1	3
1	2	28
2	3	21954
3	4	3312868145
4	1	2594339824
5	2	2328384290
6	3	1974036709
7	4	2601710651
…	…	…
63	4	4012694445

MiMC example

```
(module
    (field prime 4194304001)
    (const $alpha scalar 3)
    (function $mimcRound
        (result vector 1)
        (param $state vector 1) (param $roundKey scalar)
        (add
            (exp (load.param $state) (load.const $alpha))
            (load.param $roundKey)))
    (export mimc
        (registers 1) (constraints 1) (steps 64)
        (static
            (cycle 1 2 3 4))
        (init
            (param $seed vector 1)
            (load.param $seed))
        (transition
            (call $mimcRound (load.trace 0) (get (load.static 0) 0)))
        (evaluation
            (sub
                (load.trace 1)
                (call $mimcRound (load.trace 0) (get (load.static 0) 0))))))
```

Appendix

STARK basics



STARK basics

1. Transition Function
2. Transition Constraints
3. Constraint Composition
4. FRI Protocol

Computation-specific (1–2)

Computation-agnostic (3–4)

STARK basics

1 **Transition Function**

2 **Transition Constraints**

3 **Constraint Composition**

4 **FRI Protocol**

AirAssembly

genSTARK