



# PIL: Polynomial Identity Language

Approaching state machines to the community

---

Héctor Masip Ardevol

Polygon Hermes  
Universitat Politècnica de Catalunya (UPC)

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

Advanced Features

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

Advanced Features

# The R1CS-SAT and PlonK-SAT Problems

- Given three  $m \times n$  matrices  $A = \{a_{i,j}\}$ ,  $B = \{b_{i,j}\}$ ,  $C = \{c_{i,j}\}$ , the **R1CS-SAT** problem asks if there exists a vector  $z \in \mathbb{F}^n$  such that, for all  $i \in [m]$ :

$$\left( \sum_{j=1}^n a_{i,j} \cdot z_j \right) \cdot \left( \sum_{j=1}^n b_{i,j} \cdot z_j \right) = \sum_{j=1}^n c_{i,j} \cdot z_j.$$

- Given five vectors  $q_L, q_R, q_O, q_M, q_C \in \mathbb{F}^n$  and a permutation  $\sigma$ , the **PlonK-SAT** problem asks if there exists three vectors  $a, b, c \in \mathbb{F}^n$  such that, for all  $i \in [n]$ :

$$(q_L)_i \cdot a_i + (q_R)_i \cdot b_i + (q_O)_i \cdot c_i + (q_M)_i \cdot a_i \cdot b_i + (q_C)_i = 0,$$
$$(a, b, c) = \sigma(a, b, c).$$

# Limitations in the Expressiveness of R1CS and PlonK

- The main problem with the expressiveness of R1CS and PlonK constraint systems is that they are very limited.
- Both R1CS and PlonK does not allow for constraints of degree greater than two.
- Hence, for instance, the following constraints are not directly possible:

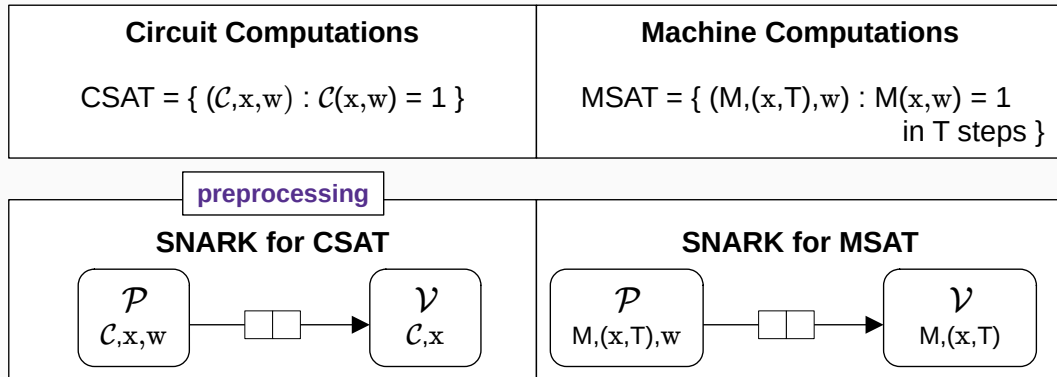
$$1 \leq a_2 \leq 2^8 - 1,$$

$$a_5^2 \cdot b_5 = c_5,$$

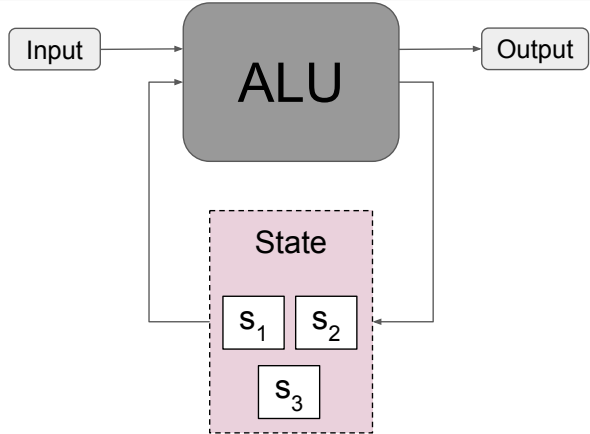
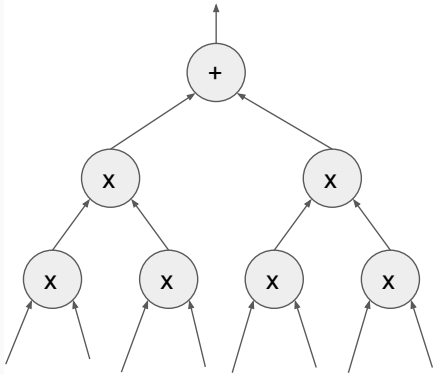
$$a_3 + b_9 = c_4.$$

# SNARK Paradigms

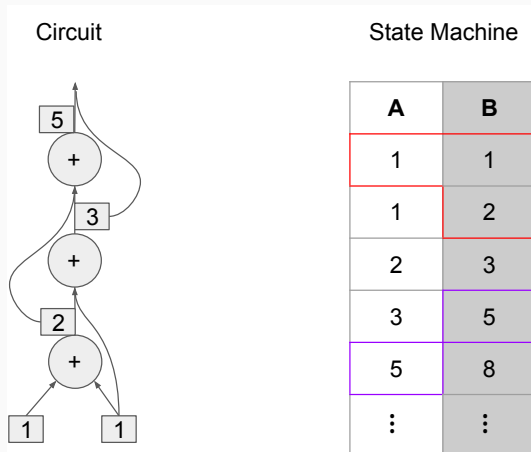
Fast verification in SNARKs is achieved in different ways, depending on the type of computation being checked.



# Arithmetic Circuits vs State Machines

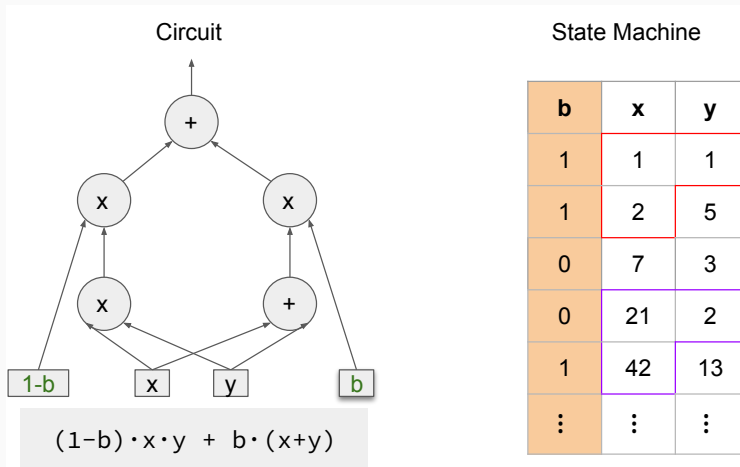


# Looping: Circuits Become Crowdy



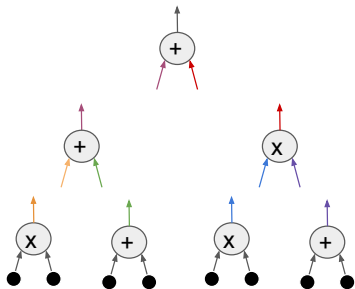


# Branching: A Big Difference

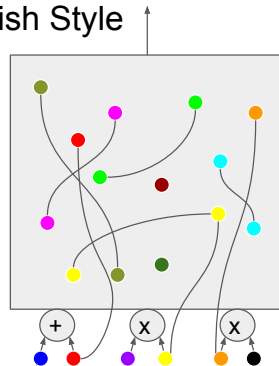


# Solution for Arithmetic Circuits: The PlonKish Style

PlonK Style



PlonKish Style



# Solution for State Machines: Polynomial Identity Language (PIL)

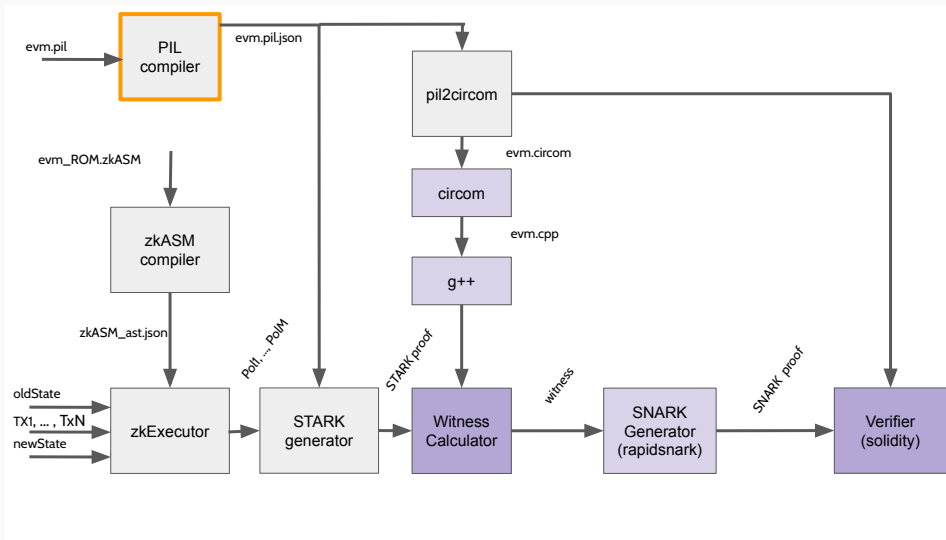
The **Polynomial Identity Language (PIL)** is a novel domain-specific language useful for defining state machines. Some of the key properties of PIL consist of:

- Providing namespaces for naming the essential parts that constitutes state machines.
- Denote whether the polynomials are committed or constant.
- Expressing polynomial relations, including identities and lookup arguments.
- (\*) Specify the type of elements that a polynomial evaluate to: bool, u32, field, etc.
- Represent values of polynomials at certain roots of unity.
- ...

# The Key Component: Modularity

- PIL aims to provide developers a holistic framework to construct state machines through an easy-to-use interface and abstracting the complexity of the proving mechanisms.
- This is why one of the main particularities of PIL is its **modularity**.
- Modularity allows programmers to define parametrizable state machines, called **namespaces**, which can be instantiated from larger state machines.
- The idea of building state machines in a modular manner makes it easier to test, review, audit, or formally verify large and complex state machines.
- In this regard, PIL developers can create their own custom namespaces or instantiate namespaces from some public library.

# How does PIL Fits in the Polygon zkEVM? (\*)



# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

Advanced Features

# State Machine to Multiply Two Numbers i

step	freeIn <sub>1</sub>	freeIn <sub>2</sub>	out
0	4	2	8
1	3	1	3
2	0	9	0
3	7	3	21
4	4	4	16
5	5	6	30
⋮	⋮	⋮	⋮

- This state machine takes two numbers and multiply them:

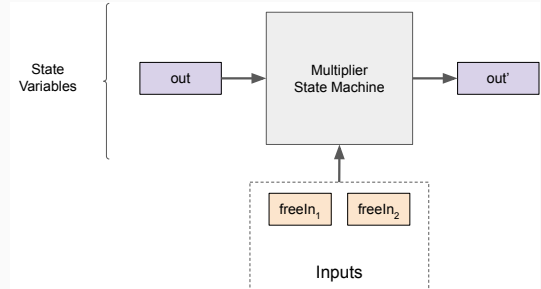
$$f(x, y) = x \cdot y.$$

- The input to this computation is feed into the *free input* columns **freeIn<sub>i</sub>**.
- The output of this computation is set in the *output* column **out**, which contain the product of the input columns.

# State Machine to Multiply Two Numbers ii

The columns of this state machine are divided in:

- a) **Input Polynomials.**
- b) **State Variables.**





## State Machine to Multiply Two Numbers iii

```
1 namespace Multiplier;
2 // Input Polynomials
3 pol commit freeIn1;
4 pol commit freeIn2;
5
6 // State Variables
7 pol commit out;
8
9 // Constraints
10 out = freeIn1*freeIn2;
```

- To achieve the correct behaviour of this state machine, one constraint must be satisfied:

$$\text{out} = \text{freeIn}_1 \cdot \text{freeIn}_2.$$

- The problem with this design is that the number of committed polynomials grow linearly with the number of multiplications/operations.
- We can do much better with the introduction of **constant** polynomials.

# Compiling PIL Code

- You can compile the previous PIL code using [pilcom](#):

```
$ node ../pilcom/src/pil.js multiplier1.pil -o multiplier1.json
```

- It generates a JSON file with the state machine data, the most relevant being:
  1. Number of Committed Polynomials.
  2. Number of Constant Polynomials.
  3. Number of Intermediate Polynomials.
  4. Number of Quotient Polynomials.
  5. Number of Polynomial Identities.
  6. Number of Plookup Identities.
  7. Number of Permutation Identities.
  8. Number of Connection Identities.

# Optimizing the Multiplier State Machine i

step	SET	freeIn	out
0	1	4	0
1	0	2	4
2	1	3	8
3	0	1	3
4	1	9	3
5	0	0	9
⋮	⋮	⋮	0

- This state machine takes two numbers and multiply them:

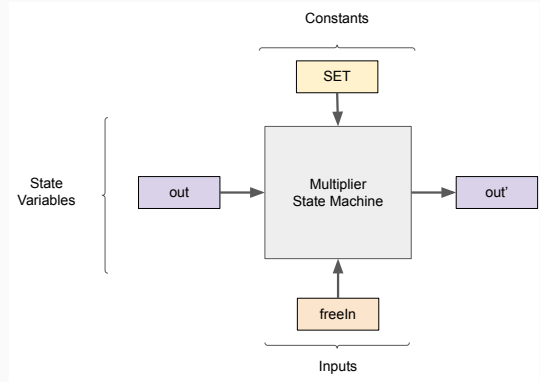
$$f(x, y) = x \cdot y.$$

- The input to this computation is feed into the free input column **freeIn**.
- In the first step, the first input  $x$  is moved from **freeIn** to **out**.
- In a second step,  $x$  is multiplied by the second input  $y$  and moved to **out**.

# Optimizing the Multiplier State Machine ii

The columns of the state machine are divided in:

- a) **Constant Polynomials.**
- b) **Input Polynomials.**
- c) **State Variables.**



## Optimizing the Multiplier State Machine iii

```
1 namespace Multiplier2;
2 // Constant Polynomials
3 pol constant SET; // 1, 0, 1, 0, ...
4
5 // Input Polynomials
6 pol commit freeIn;
7
8 // State Variables
9 pol commit out;
10
11 // Intermediate Computations
12 pol mul = out*freeIn;
13
14 // Constraints
15 out' = SET*freeIn + (1 - SET)*mul;
```

- To achieve the correct behaviour of this state machine, one constraint must be satisfied:  
$$\text{out}' = \text{SET} \cdot \text{freeIn} + (1 - \text{SET}) \cdot (\text{out} \cdot \text{freeIn}).$$
- Notice how the SET column helps out with the branching.

- As a convention, we use a tick ' to denote the “next” iteration.
- In the case of polynomials defined over the roots of unity:

$$f'(X) := f(\omega X),$$

## State Machine to Generate 4-Byte Numbers i

step	SET	freeIn	out
0	1	0xba04	0x00000000
1	0	0x3ff2	0x0000ba04
2	1	0x4443	0xba043ff2
3	0	0xc1d1	0x00004443
4	1	0xd11e	0x4443c1d1
5	0	0x6ab9	0x0000d11e
⋮	⋮	⋮	0xd11e6ab9

- This state machine takes two 2-byte numbers and generates a 4-byte number from them.
- The logic is similar to the previous one.
- In the first step, the first input *x* is moved from **freeIn** to **out**.
- In a second step, *x* is concatenated to the second input *y* and moved to **out**.

# State Machine to Generate 4-Byte Numbers ii

Filename: config.pil

```
1      constant %N = 2**10;
```

Filename: global.pil

```
1      include "config.pil";
2
3      namespace Global(%N);
4      pol constant BYTE2; // All 2-byte numbers
5      // -- more pols --
```

Filename: byte4.pil

```
1      include "config.pil";
2      include "global.pil";
3
4      namespace Byte4(%N);
5      // Constant Polynomials
6      pol bool constant SET;
7
8      // Input Polynomials
9      pol u16 commit freeIn;
10
11     // State Variables
12     pol u32 commit out;
13
14     // Constraints
15     freeIn in Global.BYTE2; // Check that input is of
16                             // 2 bytes
17     out' = SET*freeIn + (1 - SET)*(2**16*out + freeIn)
18         ;
```

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

Advanced Features



# Namespaces i

```
1 namespace Name(param_1, param_2, ...);
```

- State machines in PIL are organized in **namespaces**.
- Namespaces are written with the keyword **namespace** followed by the name of the state machine.
- Namespaces can optionally include some parameters.
- Every component of a state machine is included within its namespace.
- There is a one-to-one correspondence between state machine and namespaces.
- In the previous example, a state machine called **Name** is created.

## Namespaces ii

- The same name cannot be used twice between state machines that are directly or indirectly related (more later):

```
1 namespace Name1;  
2  
3 // --code--  
4  
5 namespace Name2;  
6  
7 // --code--  
8  
9 namespace Name99;  
10  
11 // --code--  
12  
13 namespace Name1; // <-- This is not allowed
```

# Polynomials

```
1 namespace Name;  
2   pol constant A; // <-- This is a constant polynomial  
3   pol commit b;  // <-- This is a committed polynomial
```

- **Polynomials** are the key component of PIL.
- Polynomials have to be compared with state machine's columns: in PIL, they are the same thing.
- More precisely, polynomials are just the interpolation of the columns over all the rows of the computational trace.
- They are initialized with the keyword **pol** and they need to be explicitly set to be **constant** (A.K.A preprocessed) or **commit**.
- Consequently, in PIL there exist two types of polynomials: **constant** and **committed**.

# Type of the Elements of a Polynomial

```
1 namespace Name;  
2   pol bool constant A;    // <-- This is the same as: A(x) in {0,1} for all x  
3   pol u16 commit b;      // <-- This is the same as: b(x) in {0,1,...,65535} for all x  
4   pol field commit c;    // <-- This is the same as: c(x) in F for all x  
5   // ...
```

- A polynomial definition can also contain a keyword indicating the type of elements a polynomial is composed by.
- Types include **bool**, **u16**, **field**, and much more.

## Warning!

The type is strictly informative! This means that to enforce the elements of some polynomial to be restricted over some smaller domain, one should include a constraint reflecting it.

# Constant Polynomials

```
1 namespace Name;  
2   pol constant A;
```

- Also known as **preprocessed polynomials**, these are polynomials that are known prior to the execution of the state machine.
- They correspond to polynomials that do not change during the execution and are known to both the prover  $\mathcal{P}$  and the verifier  $\mathcal{V}$ .
- They can be thought as the preprocessed polynomials of an arithmetic circuit.
- A typical use of these polynomials is the inclusion of selectors.
- To create a constant polynomial, initialize a polynomial with the keyword **constant** and they are typically written in uppercase.

# Committed Polynomials

```
1 namespace Name;  
2   pol commit a;
```

- These are polynomials that are **NOT** known prior to the execution of the state machine.
- They correspond to polynomials that change during the execution and are **only** known to the prover  $\mathcal{P}$ .
- To create a constant polynomial, initialize a polynomial with the keyword **committed**.
- These polynomials are typically used as **input polynomials** and as **state variables**.

## Note

Both free input polynomials and polynomials representing state variables are considered to be committed.

# Free Input Polynomials

- Free input polynomials are used to introduce data to the state machines.
- The state machine computes its logic over the data introduced by them.
- This data is considered the output of some state transition (or the output of the state machine, if it is the last transition).
- These polynomials are introduced by the prover and not known to the verifier.
- From all the previous reason, they are labelled as **committed**.

# State Variables

- State variables are a set of values considered to be the state of the state machine.
- These polynomials and the pivotal role of the state machine: the prover focus to the generation of a proof of the correct evolution of the state variables.
- The output of the computation in each state transition is included in the state variables.
- The state of the state variables in the last transition is the **output** of the computation.
- State variables depend on the input and the constant polynomials.
- They are therefore labeled as committed.



# Constraints

```
1 namespace Name;  
2   pol constant A;  
3   pol commit b;  
4   pol commit c;  
5  
6   // Constraints  
7   b in A;           // <-- This is a range check  
8   c' = b + 5*c;     // <-- This is an identity
```

- The set of **constraints** is the most important part of a PIL code
- The constraints are the set of relations between polynomials that dictate the correct evolution of the state machine in every step.
- A state machine does what it does because the set of constraints reflects it.
- There are some types of constraints.

# Type of Constraints

Constraints can be of the following types (to be changed):

- Identity

```
1 a' = a + 5*b + c;
```

- Containment

```
1 a in b;
```

- Permutation

```
1 {a, b} is {c, d};
```

- Copy-Constraint

```
1 {a, b, c} connect {Sa, Sb, Sc};
```

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

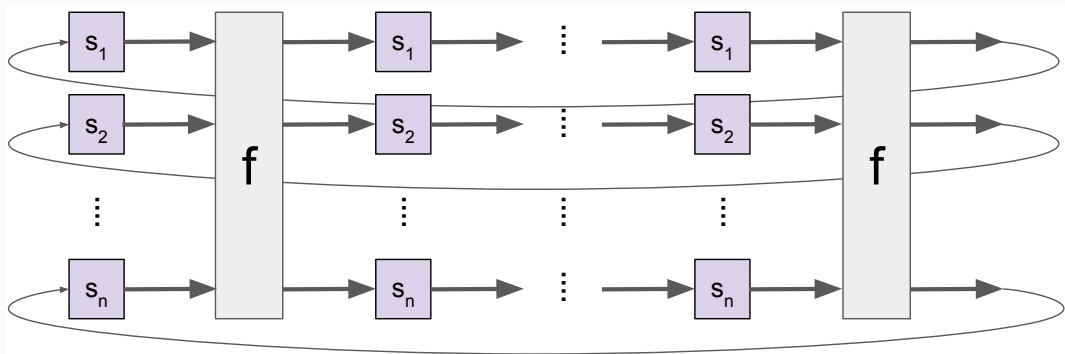
Advanced Features

- There is one implicit complexity in the design of state machines:

**State machines should have a cyclical nature.**

- This means that the description (in terms of constraints) of a state machine is not correct if the appropriate constraints are not satisfied **everywhere**.
- This is an important aspect that has to be taken care of when designing the set of constraints of a state machine.
- It is normally done by adding artificial values in latter evaluations of some polynomials as the last transition of the computation.

# Diagram of the Cyclic States




# Cyclical Nature: Example i

- Say we have the following state machine with its respective computational trace:

```
1 namespace Cyclic1;  
2   pol commit a;  
3   pol commit b;  
4  
5   pol inter = (a+1)*a;  
6  
7   inter*(a-1) = 0;  
8   b' = b+a;
```

a	b
1	1
0	2
-1	2
1	1
1	1
1	2
-1	3
-1	2



a	b
1	1
0	2
-1	2
1	1
1	2
-1	3
-1	2

- Then, some values have to be adapted so that the constraint  $b' = b + a$  is also satisfied in the transition that goes from the last row to the first one.

## Cyclical Nature: Example ii

- Another option would have been the introduction of a selector:

```
1 namespace Cyclic2;  
2 pol constant SEL; // 1, 0, 1,  
   0, ...  
3  
4 pol commit a;  
5 pol commit b;  
6  
7 pol inter = (a+1)*a;  
8  
9 inter*(a-1) = 0;  
10 b' = SEL*(b+a) + (1-SEL);
```

SEL	a	b
1	1	1
1	0	2
1	-1	2
1	1	1
0	1	2

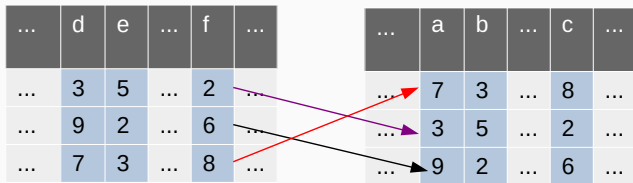
- A third option (when possible) is taking advantage of some existing selectors to accommodate latter values.

# Achieving Modularity: Divide and Conquer

- We could keep adding polynomials to our state machine to express more operations but that would make the design hard to test, audit or formally verify.
- To avoid this, PIL let you use a divide and conquer technique:
  - a) Instead of one (big) state machine, a typical architecture consists on different state machines.
  - b) Each state machine is devoted to proving the execution of a specific task, each with its own set of constraints.
  - c) Then, relevant polynomials on different state machines are related using lookup or permutation arguments.
  - d) This guarantees consistency as if it would have been a single state machine.
- We will refer to this state machine design as **modular** and refer to this state machine property as its **modularity**.

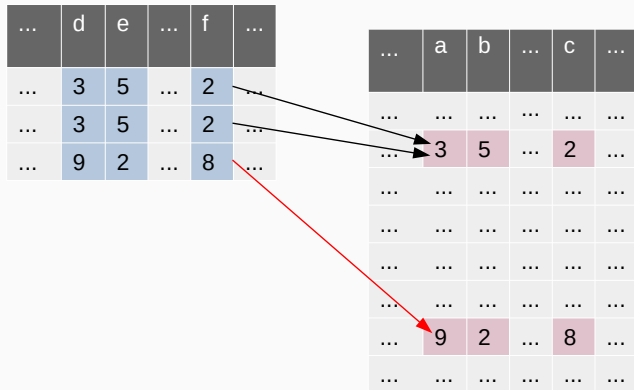


# Constraining a Permutation



```
1 namespace sm1;  
2   pol commit a, b, c;  
3  
4 namespace sm2;  
5   pol commit d, e, f;  
6   {d, e, f} is {sm1.a, sm1.b, sm1.c};
```

# Constraining an Inclusion



```
1 namespace sm1;
2   pol commit a, b, c;
3
4 namespace sm2;
5   pol commit d, e, f;
6   {d, e, f} in {sm1.a, sm1.b, sm1.c};
```

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

**More Examples**

Advanced Features

# Connecting State Machines: A Complete Example

To illustrate this important process:

1. First, we design a state machine to manage arithmetic operations over 2-byte elements.
2. Then, we will connect this state machine with another state machines (that needs to perform arithmetic operations) via a lookup argument.

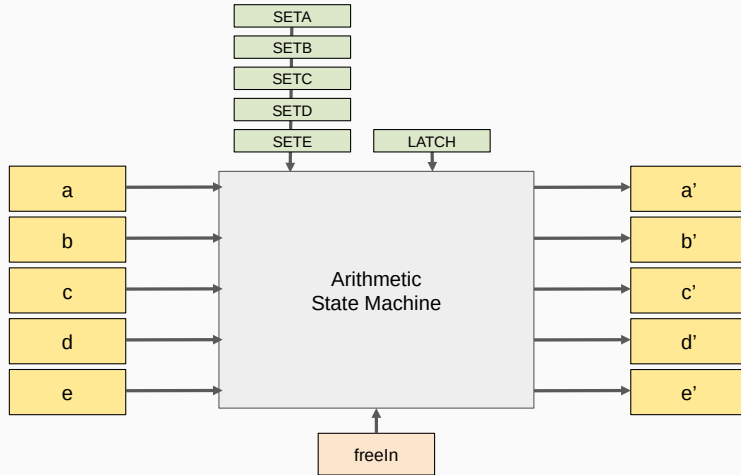
# The Arithmetic State Machine

- The **Arithmetic State Machine** is in charge of checking that some arithmetic operations like additions and multiplications are correctly performed over 2-byte elements.
- For this, we want to guarantee that the polynomials **a**, **b**, **c**, **d**, and **e** fulfil the identity:

$$\mathbf{a} \cdot \mathbf{b} + \mathbf{c} = 2^{16} \cdot \mathbf{d} + \mathbf{e}.$$

- a) Notice that the multiplication between **a** and **b**, which are 2-byte elements, can be expressed with **e** and **d**, where these are also 2-byte elements.
- b) Also notice that we have to enforce that all the evaluations of **a**, **b**, **c**, **d** and **e** are 2-byte elements.

# Arithmetic State Machine: Diagram



# Arithmetic State Machine: Computational Trace

SETA	SETB	SETC	SETD	SETE	LATCH	freeIn	a	b	c	d	e
1	0	0	0	0	0	0x0003	0	0	0	0	0
0	1	0	0	0	0	0x0002	0x0003	0	0	0	0
0	0	1	0	0	0	0x0004	0x0003	0x0002	0	0	0
0	0	0	1	0	0	0x0000	0x0003	0x0002	0x0004	0	0
0	0	0	0	1	0	0x0017	0x0003	0x0002	0x0004	0x0000	0
1	0	0	0	0	1	0x1111	0x0003	0x0002	0x0004	0x0000	0x0017
0	1	0	0	0	0	0x2222	0x1111	0x0002	0x0004	0x0000	0x0017
0	0	1	0	0	0	0x3333	0x1111	0x2222	0x0004	0x0000	0x0017
0	0	0	1	0	0	0x0246	0x1111	0x2222	0x3333	0x0000	0x0017
0	0	0	0	1	0	0xb975	0x1111	0x2222	0x3333	0x0246	0x0017
1	0	0	0	0	1	0x7777	0x1111	0x2222	0x3333	0x0246	0xb975
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.

- We use LATCH to flag when the operation is ready.
- Notice that SETA, SETB, SETC, SETD, SETE and LATCH are constant polynomials.
- freeIn is committed and contains the values for which we want to perform the arithmetic operations.
- Polynomials a, b, c, d and e compose the state variables.

# Arithmetic State Machine: Constraints

The polynomial identities that define the Arithmetic State Machine are the following:

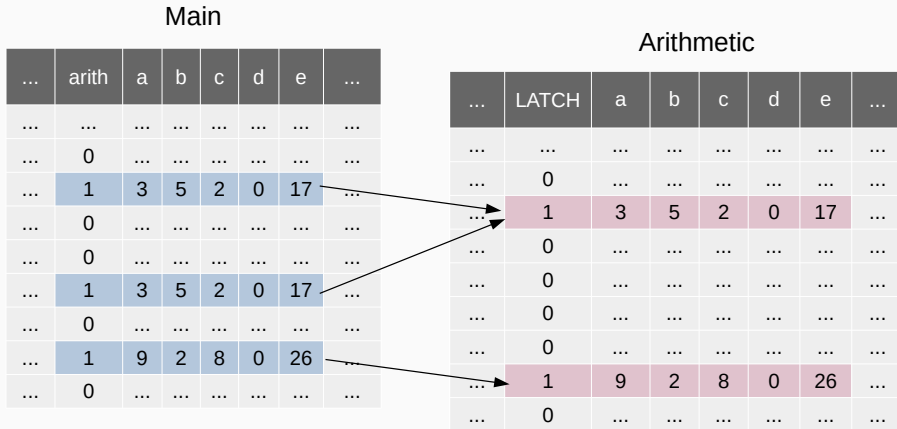
$$\begin{aligned}\text{freeIn} &\subset [0, 2^{16} - 1], \\ a' &= \text{SETA} \cdot (\text{freeIn} - a) + a, \\ b' &= \text{SETB} \cdot (\text{freeIn} - b) + b, \\ c' &= \text{SETC} \cdot (\text{freeIn} - c) + c, \\ d' &= \text{SETD} \cdot (\text{freeIn} - d) + d, \\ e' &= \text{SETE} \cdot (\text{freeIn} - e) + e, \\ 0 &= [a \cdot b + c - (2^{16} \cdot d + e)] \cdot \text{LATCH}.\end{aligned}$$



# Arithmetic State Machine: PIL

```
1  include "global.pil";
2
3  namespace Arith;
4      // Constant Polynomials
5      pol bool constant SETA, SETB, SETC, SETD, SETE;
6      pol bool constant LATCH;
7
8      // Input Polynomials
9      pol u16 commit freeIn;
10
11     // State Variables
12     pol u16 commit a, b, c, d, e;
13
14     // Constraints
15     freeIn in Global.BYTE2;
16
17     a' = SETA*(freeIn - a) + a;
18     b' = SETB*(freeIn - b) + b;
19     c' = SETC*(freeIn - c) + c;
20     d' = SETD*(freeIn - d) + d;
21     e' = SETE*(freeIn - e) + e;
22
23     pol mulSum = a*b + c - (d * 2**32 + e);
24     LATCH * mulSum = 0;
```

# Connecting The Main and Arithmetic State Machines



Main.arith · [Main.a, Main.b, Main.c, Main.d, Main.e]

⊂

Arith.LATCH · [Arith.a, Arith.b, Arith.c, Arith.d, Arith.e]

# Table of Contents

Introduction

Hello Word Examples

PIL Components

Cyclical Nature and Modularity

More Examples

Advanced Features

# Public Inputs

```
1 namespace Public;
2   // Constant Polynomials
3   pol constant L1; // 1, 0, ..., 0
4
5   // State Variables
6   pol commit a;
7
8   // Public Inputs
9   public publicInput = a(0);
10
11  // Constraints
12  L1 * (a - :publicInput) = 0;
```

- Here, we introduce the concept of **public inputs**.
- Public inputs are values of a polynomial that are known prior to the execution of a state machine.
- In the previous example, the public input is set to be the first element of the polynomial **a** and a colon **:** is used to indicate it to the compiler.

# Permutation Check

```
1 namespace sm1;  
2   pol commit a, b, c;  
3  
4 namespace sm2;  
5   pol commit a, b, c;  
6   {a, b, c} is {sm1.a, sm1.b, sm1.c};
```

- In this example we use the `is` keyword to denote that `[sm1.a, sm1.b, sm1.c]` and `[sm2.a, sm2.b, sm2.c]` are a permutation of each other.
- This is useful to connect distinct state machines, since this constrain is forcing that polynomials belonging to different state machines are the same (up to permutation).

# The Connect Keyword

```
1 namespace Connect;  
2   // Constant Polynomials  
3   pol constant SA, SB, SC;  
4  
5   // State Variables  
6   pol commit a, b, c;  
7  
8   {a, b, c} connect {SA, SB, SC};
```

- Here, we have introduced the **connect** keyword to denote that the copy constraint is applied to  $[a, b, c]$  using the permutation created by  $[SA, SB, SC]$ .
- In particular, this allows one to check that the permutation induced by  $[SA, SB, SC]$  is satisfied by  $[a, b, c]$ .

- The previous feature can be used to describe an entire PlonK circuit in PIL:

```
1 namespace Plonk;
2 // Constant Polynomials
3 pol constant SA, SB, SC;
4 pol constant QL, QR, QM, QO, QC;
5 pol constant L1; // 1, 0, ..., 0
6
7 // State Variables
8 pol commit a, b, c;
9
10 // Public Input
11 public publicInput = a(0);
12
13 // Intermediate Computations
14 pol ab = a*b;
15
16 {a, b, c} connect {SA, SB, SC};
17
18 QL*a + QR*b + QM*ab + QO*c + QC = 0;
19
20 L1 * (a - :publicInput) = 0;
```

- Here are some vectors for which the **connect**, **in** and **is** functionalities are designed for:

(1, 1, 1, 1, 3, 2)	connect	(1, 2, 4, 3, 5, 6)
(3, 9, 3, 1, 12, 6, 9)	connect	(3, 7, 1, 4, 5, 6, 2)
(3, 2)	in	(1, 2, 3, 4)
(1, 5, 5, 5, 8, 1, 1, 2)	in	(1, 2, 4, 5, 8)
(3, 2, 3, 1)	is	(1, 2, 3, 3)
(5, 5, 6, 9, 0)	is	(6, 5, 9, 0, 5).



# RapidUp: Permutation Check with Multiple-Domain

```
1 namespace sm1(2**28);
2   pol commit a, b, c;
3   pol bool commit sel;
4
5 namespace sm2(2**23);
6   pol commit d, e, f;
7   pol bool commit latch;
8
9   latch {d, e, f} is sm1.sel {sm1.a, sm1.b, sm1.c};
10  // {d, e, f} is sm1.sel {sm1.a, sm1.b, sm1.c};    <-- No selector on the first
11  // latch {d, e, f} is {sm1.a, sm1.b, sm1.c};      <-- No selector on the second
```

- Another important feature is the possibility to prove that polynomials of distinct state machines are the same (up to permutation) in a subset of its elements.
- This helps to improve efficiency when state machines are defined over subgroups of distinct size.
- The selectors choose the subset of elements to be included in the permutation argument.

# RapidUp: Diagram

