AIR-ICICLE : Plonky3 on I...
Overview
Methodology
Fibonacci Example
Conclusion & Future work

Ingonyama · Follow
Last edited by Omer Shlomovits on Feb 14, 2025
Contributed by

AIR-ICICLE : Plonky3 on ICICLE, part 1

In this article, we present an integration of the ICICLE field library (rust wrappers) for trace generation and symbolic constraints for AIR arithmetization using the Plonky3 framework. This enables users to

- write AIR circuits in the Plonky3 AIR language and easily interface with ICICLE APIs,
- generate trace data in ICICLE field types in any device using ICICLE's device agnostic API's,
- generate symbolic constraints.

The user can parse the symbolic constraints and show that the trace satisfies the constraints by taking advantage of ICICLE backend APIs such as the NTT, Polynomials, VecOps, Hash, MerkleTree and FRI (upcoming). In short, this enables users to write custom STARK provers that meet their application requirements.

As a proof of concept of this integration, we have adapted the Plonky3 examples

- Fibonacci AIR
- Blake3 AIR
- Keccak AIR

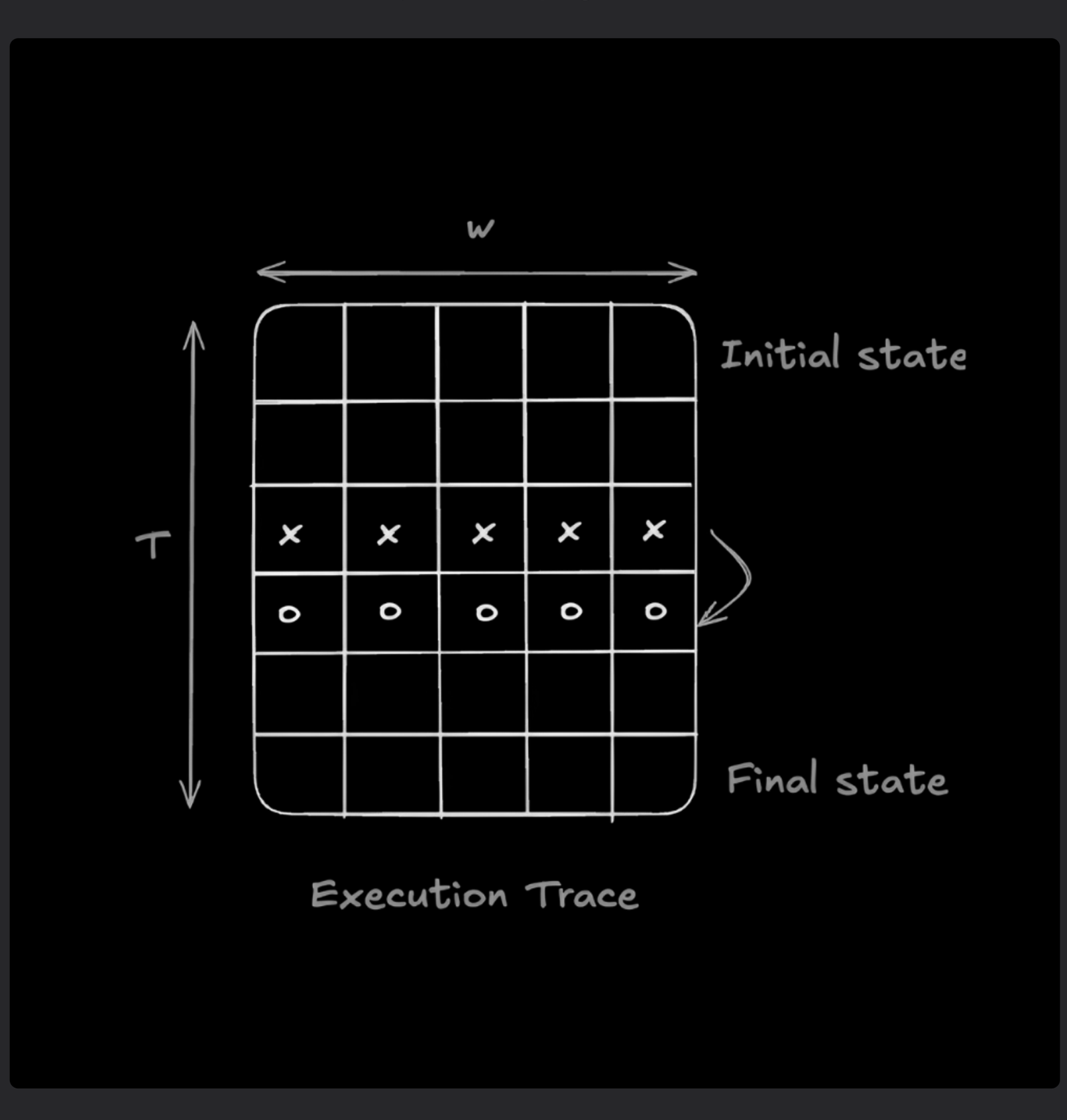
and produced the trace and symbolic constraints in ICICLE data types. Our code is open source and can be found here. As always, we will make our roadmap transparent and would love to collaborate! In order for this integration to be functional we had to redefine some properties of the Field trait in ICICLE, and the ICICLE branch used for this purpose can be found here.

The bottomline is: The user can use the Plonky3 AIR script that they are familiar with but generate trace data directly and compatible with ICICLE backend library.
In short, AIR construction is identical to how one would write an AIR circuit in Plonky3.

> **Note:** We haven't currently implemented a backend prover and will do so in future work. We encourage users to try different air circuits in this framework and build their own STARK provers using the ICICLE framework.

Overview

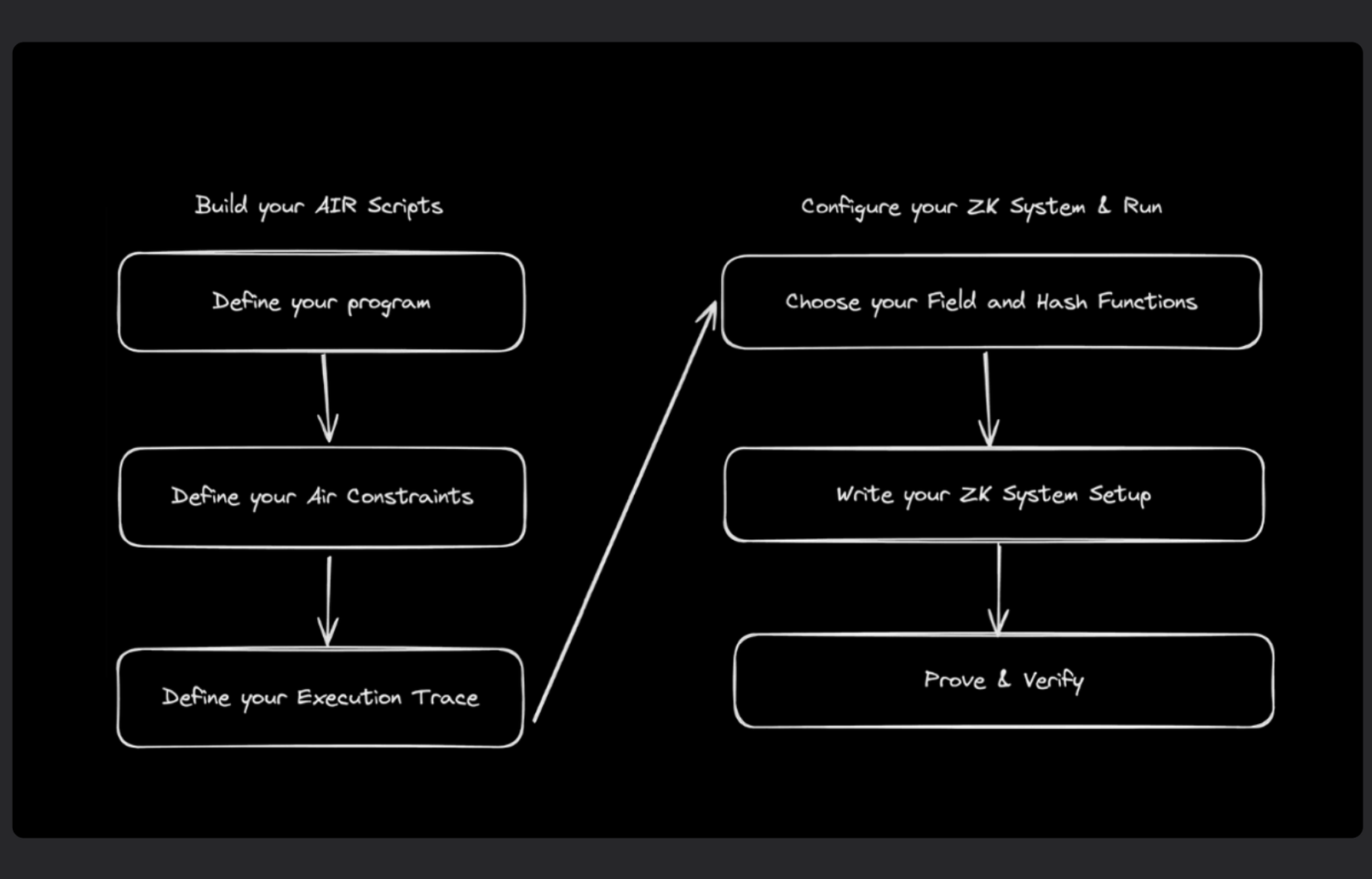
AIR (Algebraic Intermediate Representation) is the arithmetization scheme used by the majority of STARK based protocols. In AIR arithmetization, a collection of registers represents the state of the program and records the values in the registers over the execution cycle of the program. The recorded sequence of values is referred to as an Execution Trace which is a matrix of Field elements FTw with "T" rows, and "w" columns, where "T" is the total number of cycles of the program and "w" the width of the state.



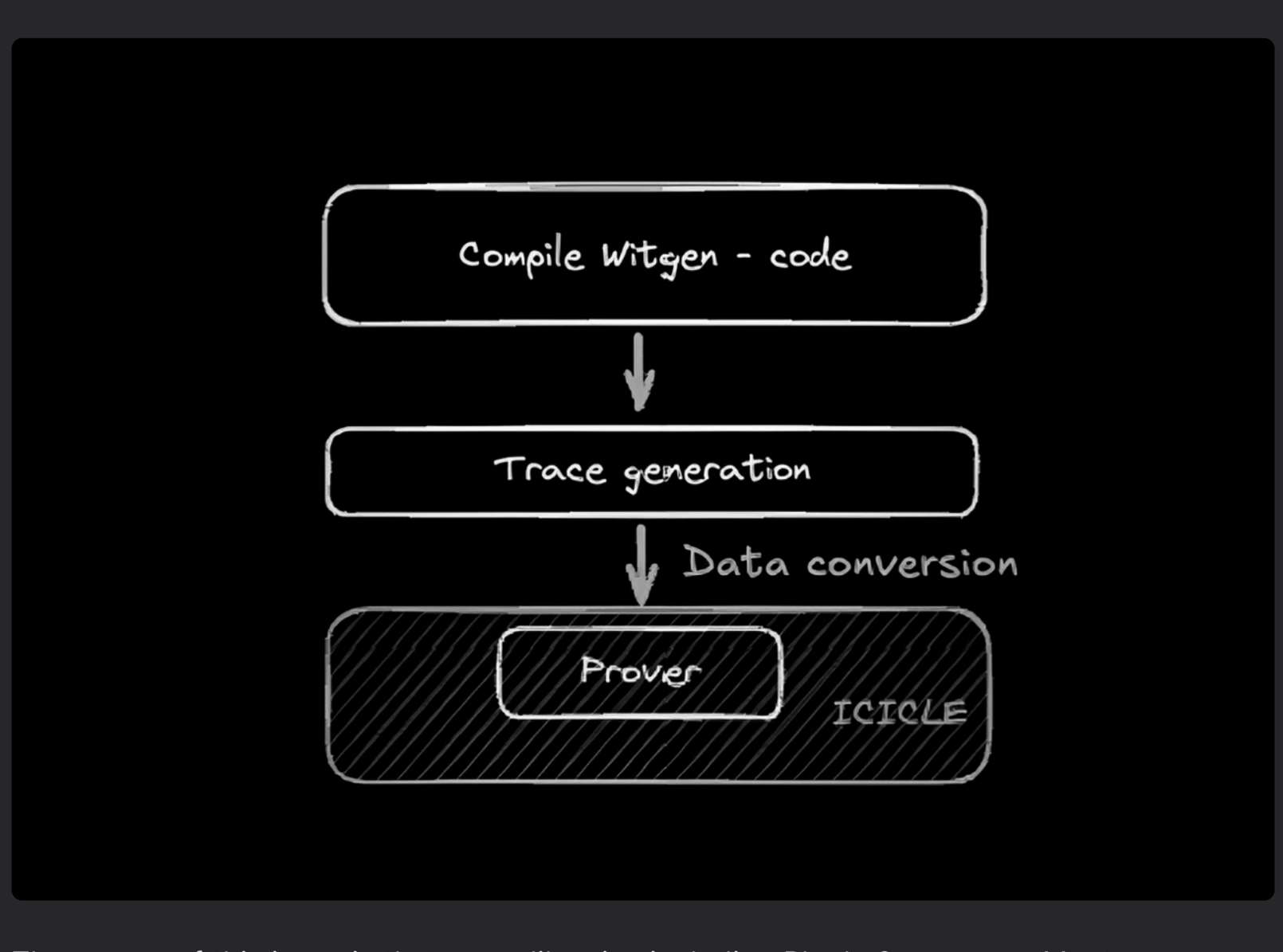
The trace satisfies two types of constraints:

- Boundary constraints $T[i, j] = \alpha \text{ for } \alpha \in F$
- Transition constraints $f(T[j,1], T[j,2], \ldots, T[j,w], T[j+1,1], T[j+1,2], \ldots, T[j+1,w]) = 0$ across the state at cycle t and the state at cycle t + 1.

The Plonky3 library is a modular framework that enables users to write their programs, define AIR constraints, execution trace, choose fields, hashes and prove the correctness of the program execution using the unistark framework. The typical workflow in a project that uses Plonky3 is



A user who wishes to enjoy GPU acceleration with the ICICLE library will typically generate a trace using Plonky3 and convert the data to ICICLE field types. This is typically wasteful and as the sizes of computation grow often becomes an overhead on its own.

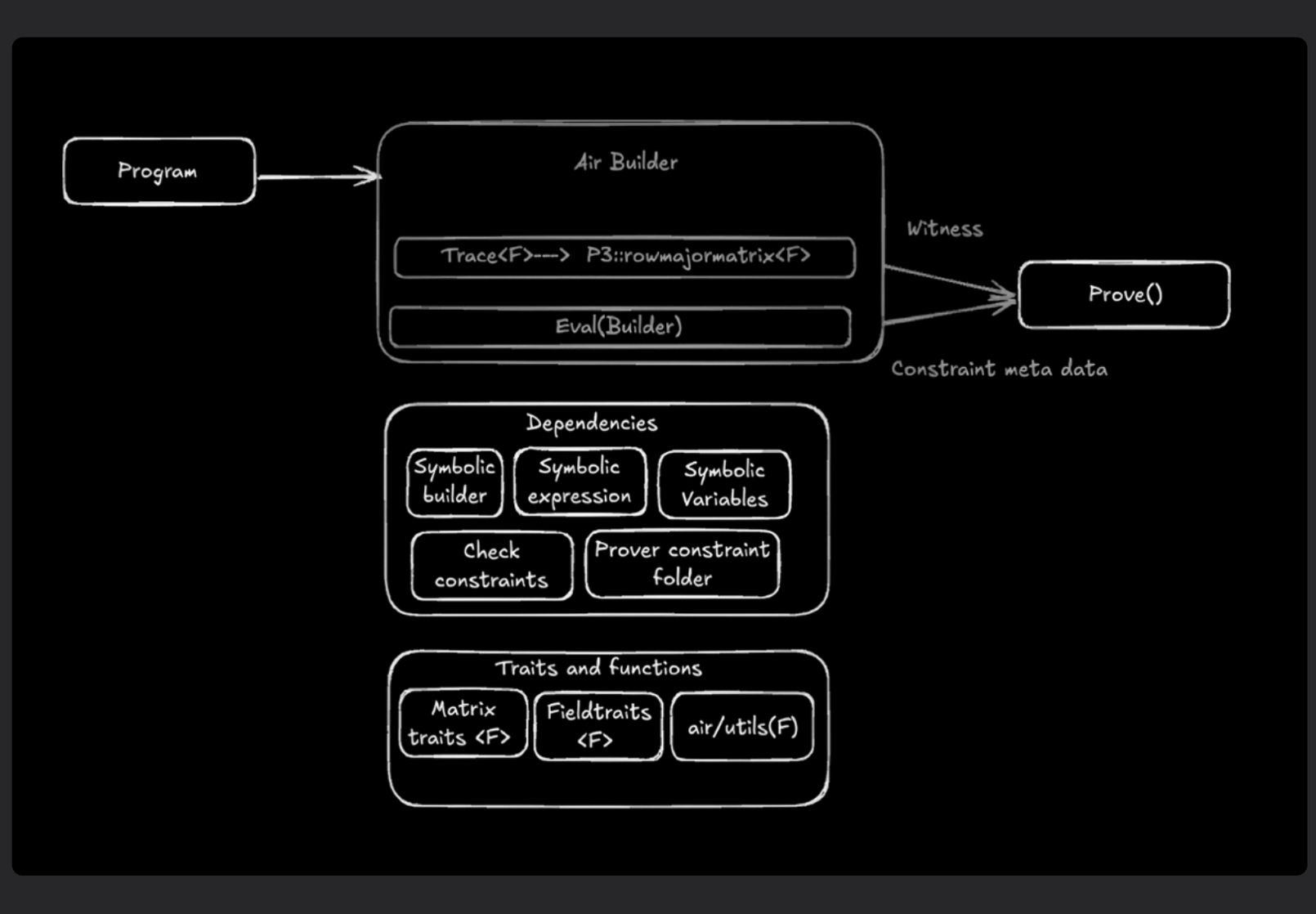


The source of this issue is that many libraries including Plonky3 assume a Montgomery representation for field arithmetic, and employ field representations that are efficient for CPU architectures. Whereas, ICICLE uses efficient data types for field arithmetic, and especially a highly efficient Barett multiplier that works across multiple backends (such as CPU/GPU/Metal).

For circuit sizes in the order of $n > 2^{20}$ and for different field types, data conversions can be a real pain point for developers. Moreover, they are forced to write their programs, arithmetizations in one framework and pay the price of conversion for acceleration using another framework.

Methodology

We begin by observing the dependencies of a typical AIR program



Air Builder

The Air builder is a modular structure that is used to both generate trace and define constraints. It contains the components

- **BaseAir<T>**: This trait allows the user to define a trace structure for a program that generates values in a datatype T.
    - The user writes the witness generation program as an implementation in the BaseAir, using a generate_trace_rows<T> struct and writes the logic for generating the witness values in the trace.
    - The output of the implementation is a matrix struct of the type RowMajorMatrix<F>.
- **Air Builder struct<F:Field+FieldAlgebra, M: Rowmajormatrix>:** The Air builder struct provides the **eval(builder)** method to generate the constraints satisfied by the trace.
    - It implements if/when assert functions that define constraints under certain conditions such as first row or last row etc…
    - Different types of air builder constructs, such as air builder with public values, PAIR (pre processed AIR) builder, Extension air builder, Permutation Air builder, FIltered Air builder etc…

Much of the air builder depends on the abstractions (note: Not the actual field implementations) of field definitions in plonky3. In particular:

- Field and FieldAlgebra traits
- ExtensionField and ExtensionFieldAlgebra traits
- Matrix traits<T>
- Utility functions.

The functionality of the Airbuilder is accessed just before proof generation begins. For example, in the unistark prover and using the functionalities in

- symbolic air builder
- symbolic expression
- symbolic variable

The prover accesses the trace length, constraint degrees, types and the conditions that the trace values need to be checked for. This metadata is essentially parsed and the constraint system is evaluated using the Plonky3 backend.

We note that the symbolic functionalities are once again only tied to the Field abstractions and not Field implementations. Our observation is that if we ensure the compatibility of the abstractions, we can use any field implementation we want to generate the trace. This is largely possible due to the modular nature of both Plonky3 and ICICLE. In the next section we discuss the relevance of symbolic constraints in the quotient polynomial computation.

Quotient/Composition Polynomial Computation and the relevance of symbolic constraints

One of the key challenges with implementing a STARK prover in GPU is computing the composition polynomial. Each program has a different set of constraints which makes it difficult to design a general strategy for parallelising this computation. The Plonky3 framework can help alleviate this problem using an abstraction called symbolic constraints. Symbolic constraints are an abstract representation of the AIR constraint. Let's look at an example. Suppose your trace at rows i and (i + 1) is:

```
| a  | b  | c  |
|----|----|----|
| a' | b' | c' |
```

and the constraint is: (a * a' + b * b' - c - c') = 0. This constraint can be rewritten as:

```
[SUB,
  [ADD,
    [MUL, (i, 0), (i + 1, 0)],
    [MUL, (i, 1), (i + 1, 1)]
  ],
  [ADD, (i, 2), (i + 1, 2)]
]
```

where ((i, j)) denotes the value at index ((i, j)) in the execution trace. This seemingly straightforward representation is very useful because now if we have the execution trace and the symbolic constraints both on GPU, we can easily parallelize computation of the composition polynomial (without any additional data movement to and from CPU). Of course, this means we need to support operations like **ADD, MUL, SUB** (and more) on the GPU.

In the next section, we walk through the precise changes in trait definitions that we used for the integration.

Trait compatibility in Plonky3 and ICICLE

The Plonky3 code differentiates between the Field and FieldAlgebra traits, in terms of functionality. On the other hand, ICICLE has traits called FieldImpl and Arithmetic which are designed slightly differently from the traits of Plonky3. This results in a practical difficulty in integrating the ICICLE back-end with Plonky3 front-end (for the AIR arithmetization). As a result, we had to modify the ICICLE traits slightly to ensure compatibility with the Plonky3 design.

For instance, in the p3 branch in ICICLE we removed the copy + from + into traits to enable symbolic constraints functionality with minimum changes in ICICLE, and this has a consequence in trace generation as values need to be cloned explicitly instead of the more efficient implicit copy. However, in our benches we did not notice a significant performance drop. This is because the underlying field elements or symbolic expressions are not large data structures, so an explicit clone() is not much different from an implicit Copy. In the longer term though, it is preferable to use implicit Copy when we integrate ICICLE back-end for the end-to-end Plonky3.

Note is that ICICLE calls extension fields by defining the num_limbs parameter as defined here

```
#[repr(C)]
pub struct Field<const NUM_LIMBS: usize, F: FieldConfig> {
    limbs: [u32; NUM_LIMBS],
    p: PhantomData<F>,
}
```

This definition is efficient for device agnostic use cases, and is somewhat different from the extension field abstractions used in Plonky3. Nevertheless, since most of the core FIeld properties are identical, we could do any user-defined arithmetic using the ICICLE arithmetic trait for extension fields as well. We will consider extension field air builder and permutation air builder for future iterations.

Fibonacci Example

Let's revisit the famous Fibonacci example, where the prover claims to know the Fibonacci series till the value **21**.

With starting values **[0,1]**, and the final value **21** denoted as public values (PI = [0,1,21]).

- The boundary conditions are:

$$T[0,0] = PI_0, \quad T[0,1] = PI_1, \quad T[7,1] = PI_2$$

- The transition conditions for (j \neq 0):

$$T[j,0] - T[j-1,1] = 0$$

$$T[j,1] - T[j-1,0] - T[j-1,1] = 0$$

The boundary conditions and transition conditions altogether make up **five constraints** for the trace to be valid. The generated trace in **ICICLE** is given by:

```
Trace DenseMatrix{
values: {
        0x00000000,
        0x00000001,
        0x00000001,
        0x00000001,
        0x00000001,
        0x00000002,
        0x00000002,
        0x00000003,
        0x00000003,
        0x00000005,
        0x00000005,
        0x00000008,
        0x00000008,
        0x0000000d,
        0x0000000d,
        0x00000015,
    },
    width: 2,
    _phantom: PhantomData<icicle_core::field::Field<1, icicle_babybear::field::Scal
};
```

The [s]ymbolic constraint generator](https://github.com/ingonyama-zk/air-icicle/blob/aee75c80a87a5e8e647352e8f99ff376bfaddfce/icicle-trace/examples/fib_icicle_trace_sym.rs#L99) that generates the meta data for the five constraints is given by

```
symbolic constraints [
   Mul { x: IsFirstRow,
         y: Sub { x: Variable( SymbolicVariable {entry: Main {offset: 0,}, index:
         y: Variable(SymbolicVariable { entry: Public, index: 0, _phantom: Phantom
           degree_multiple: 1,},degree_multiple: 2,},

   Mul { x: IsFirstRow,
         y: Sub { x: Variable( SymbolicVariable {entry: Main {offset: 0,}, index:
         y: Variable(SymbolicVariable { entry: Public, index: 1, _phantom: Phantom
           degree_multiple: 1,},degree_multiple: 2,},

   Mul {x: IsTransition,
        y: Sub {x: Variable( SymbolicVariable {entry: Main {offset: 0,},index: 1,
        y: Variable(SymbolicVariable {entry: Main {offset: 1,},index: 0,_phantom:
           degree_multiple: 1,},degree_multiple: 1,},

   Mul {x: IsTransition ,
        y: Sub {x: Add {x: Variable(SymbolicVariable {entry: Main { offset: 0,},in
        y: Variable(SymbolicVariable {entry: Main {offset: 0,},index: 1,_phantom:
           degree_multiple: 1,},
        y: Variable(SymbolicVariable {entry: Main {offset: 1,},index: 1,_phantom:
           degree_multiple: 1,},degree_multiple: 1,},

   Mul {x: IsLastRow,
        y: Sub {x: Variable(SymbolicVariable { entry: Main {offset: 0,},index: 1,
        y: Variable(SymbolicVariable {entry: Public,index: 2,_phantom: PhantomData
           degree_multiple: 1,},degree_multiple: 2,},
]
Constraint count: 5
Constraint degree: 2
Trace eval domain: 8
```

which in a more readable format looks like

Constraint Type	Condition	Operation	Degree Multiple
Constraint 1	`IsFirstRow`	`Variable(Main[0], 0) - Variable(Public, 0)`	2
Constraint 2	`IsFirstRow`	`Variable(Main[0], 1) - Variable(Public, 1)`	2
Constraint 3	`IsTransition`	`Variable(Main[0], 1) - Variable(Main[1], 0)`	1
Constraint 4	`IsTransition`	`(Variable(Main[0], 0) + Variable(Main[0], 1)) - Variable(Main[1], 1)`	1
Constraint 5	`IsLastRow`	`Variable(Main[0], 1) - Variable(Public, 2)`	2

This meta data can be easily parsed by a backend prover into a set of instructions and evaluate efficiently.

Conclusion & Future work

In this work, we demonstrated a promising first step towards the integration of ICICLE back-end for Plonky3. Longer term, we envision that an end-to-end integration of ICICLE as back-end to a powerful and modular toolkit like Plonky3 can open interesting avenues for developers to get the best of both worlds: writing circuits using the modular Plonky3 toolkit and seamlessly getting the back-end prover using the ICICLE library. In Part2 we will detail showcase an end-to-end RiscV ZKVM written in ICICLE with a Plonky3 frontend.

For questions, collaboration ideas and feedback: feel free to reach us at hi@ingonyama.com, open an issue in the repo or reach the authors directly at karthik@ingonyama.com, suyash@ingonyama.com

Acknowledgements:

We would like to thank *Leon Hibnik*, *Yuval Domb* and *Tomer Solberg* for helpful discussions.