Grant Report: Ethereum Foundation

Verification of a zkVM Plonky3 chip in Rocq

Date: 2025-08-28

To: Ethereum Foundation https://ethereum.foundation/

From: Formal Land (Arae) https://formal.land/

Contact: guillaume.claret@formal.land

> This grant report aims to answer the third milestone "Formally verify the soundness and correctness of a relevant zkVM component(e.g., an SP1 chip)" of our grant "Plonky3 in Rocq".

Summary

For this grant milestone, we formally verified in Rocq the soundness and correctness of the Plonky3 chip branch_eq of OpenVM.

We chose this chip after discussing with the OpenVM team, which told us this was a relevant item to look at. To make sure our Rocq model is equivalent to the Rust implementation of the circuit, we also built a translation mechanism from Plonky3 to our Garden circuit framework.

Extraction from Plonky3 to Rocq

We first tried to do the translation from Plonky3 to Rocq using coq-of-rust, as the circuits are described in an embedded DSL in Rust. But this approach was too time-consuming, as we were regularly encountering new Rust features or standard library definitions that were not supported yet in `coq-of-rust`.

In order to reduce the risk of building a too complex solution, we went with the direct pretty-printing of the constraints. Indeed, as the circuit is generated once and for all for a given zkVM, it is enough to make sure we have the same set of constraints on the Rust and Rocq sides to ensure we are verifying the actual implementation. We pretty-print the constraints on both sides and make sure they are producing the exact same file.

Here is what the pretty-printing of a circuit looks like:

```
Trace 🐾
  Message 🦜
    eval_row
  Message 🦜
    eval_row::flags
  AssertZero:
    Mul:
      Variable: 0
      Sub:
        Variable: 0
        Constant: 1
  AssertZero:
    Mul:
      Variable: 1
      Sub:
        Variable: 1
        Constant: 1
  AssertZero:
    Mul:
      Variable: 2
      Sub:
        Variable: 2
        Constant: 1
```

A challenge of the pretty-printing approach is that the produced constraints can be very large. For example, for the `sha256` circuit of OpenVM on which we ran an experiment, the output was around 40,000 lines long, for a source Rust file of less than a thousand lines.

We made these two design choices to simplify the comparison of the constraints files:

1. Adding a logging command to Plonky3. This command has no effect but inserts a string message in the list of constraints produced at the end, so that we can recognize in which function or code block a particular constraint comes from. This information is generally enough to quickly identify where the mistakes were in the model, according to our experiment on the `sha256` circuit.
2. Flattening the associative operators. The addition and multiplication of polynomial expressions are the most common operators used to build constraints. We flatten these binary operators so that they apply to a list of parameters instead of two at a time. This reduced most of the nested levels of the polynomial expressions we were printing, and clarified the `diff` comparison between the two files of constraints.

Pretty-printing in Rust

The code for the pretty-print is available in our pull request add command to pretty-print circuits. The main part is the introduction of a mock `AirBuilder` named `RocqAirBuilder` that does nothing but accumulate the constraints to pretty-print them at the end of a circuit's construction:

```rust
enum Constraint<E> {
    AssertZero(E),
    Message(String),
    Interaction(Interaction<E>),
}

struct RocqAirBuilder {
    main: RowMajorMatrix<SymbolicVariable<Goldilocks>>,
    constraints: Vec<Constraint<SymbolicExpression<Goldilocks>>>,
}
```

We handle three kinds of constraints:

- `AssertZero`: a direct equality constraint between a polynomial expression and zero.
- `Message`: a pretty-printed string that clarifies the source of the constraints.
- `Interaction`: a special constraint for OpenVM's buses.

We use symbolic expressions to keep their structures explicit and to be able to print their syntax trees.

Pretty-printing in Rocq

As Rocq is a strict, purely functional language, what we actually do is to define a function that translates a list of circuit constraints to a string, which we later extract into a snapshot file.

Here are the main technical decisions we made:

1. Using the PrimString module instead of the usual `String` module. It enables much better performance by using native OCaml strings internally, instead of building strings using an explicit inductive definition going down to the bit level.
2. Designing a type-class adding a pretty-printing capability to all data structures we want to display.
3. Designing a new type of monad, with a deep embedding of the polynomial variables and expressions, instead of using a shallow embedding like in the rest of the Garden framework for circuits. This was required to be able to pretty-print the structure of the expressions.

This new monad is named `MExpr.t` and is located in the file Garden/Plonky3/MExpr.v. We define with it equivalence rules to compare a circuit in the deep-embedding form to its version in the shallow embedding form:

```
Module Eq.
  (** Equality between the [MExpr.t] and the [M.t] monad. *)
  Inductive t {A1 A2 : Set} `{Eval.C A1 A2} {p} `{Prime p} (env : Env.t) :
    MExpr.t A1 -> M.t A2 -> Prop :=
  | Pure (value : A1) value' :
    Eval.eval env value = value' ->
    t env (Pure value) (M.Pure value')
  | AssertZero (expr : Expr.t) expr' (value : A1) value' :
    Eval.eval env expr = expr' ->
    Eval.eval env value = value' ->
    t env (AssertZero expr value) (M.AssertZero expr' value')
  | Call (e : MExpr.t A1) (e' : M.t A2) :
    t env e e' ->
    t env (MExpr.Call e) (M.Call e')
  | Let {B1 B2 : Set} `{Eval.C B1 B2}
      (e : MExpr.t B1) (k : B1 -> MExpr.t A1)
      (e' : M.t B2) (k' : B2 -> M.t A2) :
    t env e e' ->
    (forall (value : B1),
      t env (k value) (k' (Eval.eval env value))
    ) ->
    t env (MExpr.Let e k) (M.Let e' k')
  | When (condition : Expr.t) condition' (body : MExpr.t A1) (body' : M.t A2) :
    Eval.eval env condition = condition' ->
    t env body body' ->
    t env (MExpr.When condition body) (M.When condition' body').
End Eq.
```

These rules are very straightforward to apply and state that the evaluation function of a polynomial expression, given some values assigned to each of the variables, commutes with the rest of the code. We have done the verification work to ensure that our definition of the `branch_eq` circuit in `M.t` is equivalent to its definition in `MExpr.t`, which is itself equivalent to the Rust implementation of the circuit when pretty-printed.

The part with the `MExpr.t` definition and proof of equivalence is quite verbose, and we are working on a new mechanism to pretty-print the constraints directly from the shallow definition in `M.t`, using meta-programming in Rocq.

Definition of the circuit in Rocq

The definition of the `branch_eq` circuit in `M.t` is available in the file Garden/OpenVM/BranchEq/core_with_monad.v. Here is the definition of its main function `eval`:

```
Definition eval {p} `{Prime p} {NUM_LIMBS : Z}
    (self : BranchEqualCoreAir.t NUM_LIMBS)
    (local : BranchEqualCoreCols.t NUM_LIMBS Z)
    (from_pc : Z) :
    M.t (AdapterAirContext.t NUM_LIMBS Z) :=
  let flags : list Z := [
    local.(BranchEqualCoreCols.opcode_beq_flag);
    local.(BranchEqualCoreCols.opcode_bne_flag)
  ] in

  let* is_valid : Z :=
    M.List.fold_left
      (fun acc flag =>
        let* _ := M.assert_bool flag in
        M.pure (BinOp.add acc flag)
      )
      Z.zero
      flags in
  let* _ := M.assert_bool is_valid in
  let* _ := M.assert_bool local.(BranchEqualCoreCols.cmp_result) in

  let a : Array.t Z NUM_LIMBS := local.(BranchEqualCoreCols.a) in
  let b : Array.t Z NUM_LIMBS := local.(BranchEqualCoreCols.b) in
  let inv_marker : Array.t Z NUM_LIMBS := local.(BranchEqualCoreCols.diff_inv_ma

  let* cmp_eq : Z :=
    M.pure (
      BinOp.add
        (BinOp.mul local.(BranchEqualCoreCols.cmp_result) local.(BranchEqualCore(
        (BinOp.mul (M.not local.(BranchEqualCoreCols.cmp_result)) local.(BranchE(
    ) in

  let* _ := M.for_in_zero_to_n NUM_LIMBS (fun i =>
    M.assert_zero (BinOp.mul cmp_eq (BinOp.sub (Array.get a i) (Array.get b i)))
  ) in
  let sum : Z := sum_for_in_zero_to_n_starting_from NUM_LIMBS (fun i =>
    BinOp.mul (BinOp.sub (Array.get a i) (Array.get b i)) (Array.get inv_marker i
  ) cmp_eq in
  let* _ := M.when is_valid (M.assert_one sum) in

  let flags_with_opcode_integer : list (Z * Z) :=
    [
      (local.(BranchEqualCoreCols.opcode_beq_flag), 0);
      (local.(BranchEqualCoreCols.opcode_bne_flag), 1)
    ] in
  let expected_opcode : Z :=
    Lists.List.fold_left
      (fun acc '(flag, opcode) =>
        BinOp.add acc (BinOp.mul flag opcode)
      )
      flags_with_opcode_integer
      0 in
  let expected_opcode : Z :=
    BinOp.add expected_opcode self.(BranchEqualCoreAir.offset) in

  let to_pc : Z :=
    BinOp.add
```

```
      (BinOp.add
        from_pc
        (BinOp.mul local.(BranchEqualCoreCols.cmp_result) local.(BranchEqualCore
      )
      (BinOp.mul (M.not local.(BranchEqualCoreCols.cmp_result)) self.(BranchEqua
    in

  M.pure {|
    AdapterAirContext.to_pc := Some to_pc;
    AdapterAirContext.reads := [a; b];
    AdapterAirContext.writes := [];
    AdapterAirContext.instruction := {|
      ImmInstruction.is_valid := is_valid;
      ImmInstruction.opcode := expected_opcode;
      ImmInstruction.immediate := local.(BranchEqualCoreCols.imm);
    |};
  |}.
```

Except for differences in notations, the code is very similar to its Plonky3 version in Rust. The main difference is the introduction of a new constraint, which is a side-effect in both Rust and Rocq. For the Rust version, we mutate the `builder` structure to add a new equality constraint to the existing ones:

```
builder.when(is_valid.clone()).assert_one(sum);
```

On the Rocq side, we make a side-effect in the `M.t` monad:

```
let* _ := M.when is_valid (M.assert_one sum) in
```

Also, we do not need to manage the memory with `.clone()` calls or the use of references `&`, as the memory management it automatic.

Determinism

The determinism of the circuit (also called the absence of under-constraints) is probably the most important property, as otherwise an attacker could forge an invalid execution trace with forbidden transactions, and this property is almost impossible to check by testing.

We state the determinism of a circuit by giving a function relating the values of the output variables to the values of the input variables if all the constraints of the circuit are satisfied.

Here is the statement for the `branch_eq` circuit, appearing below its definition in the source Rocq file:

```
{{ eval self local from_pc 🔽
  {|
    AdapterAirContext.to_pc :=
      Some (BinOp.add from_pc (
        if expected_cmp_result then
          local.(BranchEqualCoreCols.imm)
        else
          self.(BranchEqualCoreAir.pc_step)
      ));
    AdapterAirContext.reads := [local.(BranchEqualCoreCols.a); local.(BranchEqua
    AdapterAirContext.writes := [];
    AdapterAirContext.instruction := {|
      ImmInstruction.is_valid := 1;
      ImmInstruction.opcode :=
        BinOp.add local.(BranchEqualCoreCols.opcode_bne_flag) self.(BranchEqualC
      ImmInstruction.immediate := local.(BranchEqualCoreCols.imm)
    |};
  |},
  local.(BranchEqualCoreCols.cmp_result) = Z.b2z expected_cmp_result
}}.
```

It gives the explicit expression of the result of the `branch_eq` circuit, which in Rust is of type:

```
pub struct AdapterAirContext<T, I: VmAdapterInterface<T>> {
    /// Leave as `None` to allow the adapter to decide the `to_pc` automatically
    pub to_pc: Option<T>,
    pub reads: I::Reads,
    pub writes: I::Writes,
    pub instruction: I::ProcessedInstruction,
}
```

For this chip, it basically says that depending on the result of the equality comparison between the two parameters `a` and `b`, which are integers decomposed into limbs, the instruction will either make a jump of `imm` offset, or proceed to the next instruction incrementing the Program Counter (PC) by the regular amount `pc_step`.

There is only one output variable for this chip: `cmp_result`. We say that it must be the field representation (`0` or `1`) of the boolean `expected_cmp_result` which is explicitly computed by:

```
let expected_cmp_result : bool :=
  match branch_equal_opcode with
  | BranchEqualOpcode.BEQ =>
    if Array.Eq.dec local.(BranchEqualCoreCols.a) local.(BranchEqualCoreCols.b)
      true
    else
      false
  | BranchEqualOpcode.BNE =>
    if Array.Eq.dec local.(BranchEqualCoreCols.a) local.(BranchEqualCoreCols.b)
      false
    else
      true
  end
```

The expected result is reversed depending on the value of `branch_equal_opcode` given through the input variables, which we expect to be well-formed.

Proof of determinism

We use two ingredients to prove the determinism of the circuit:

1. Our reasoning rules to progress through the monadic code of the circuit, accumulating the constraints along the way.
2. Our field arithmetic tactics to show that the polynomial equations from the constraints imply the expected equations on the output variables.

At the outer layer of the proof, we use the tactic:

```
eapply Run.LetAccumulate with (value := ...) (P1 := ...).
```

It applies to each of the top-level `let*` bindings in the computation of the circuit, and enables us to state:

- An expression of the result of the binding, generally the unit value.
- An expression for the predicate implied by the constraints of the binding.

Inside each `let*` binding, we reason by rewriting to simplify the polynomial equations, and by cases to explore all possible branches for the equality or inequality of the parameters. At the end of the proof, we aggregate the properties implied by each `let*` into the main statement using the tactic `tauto`.

Completeness and correctness

The functional correctness of this circuit is given by the explicit expression for its result and for its output variable `cmp_result`.

For the completeness property, we introduce a new predicate:

```
{{ e ✅ value }}
```

stating that a circuit `e` reduces to the value `value`, validating all the constraints it encounters when reducing its expression. The rules are:

```
Inductive t {A : Set} : M.t A -> A -> Prop :=
| Pure (value : A) :
  {{ M.Pure value ✅ value }}
| AssertZero (x : Z) (value : A) :
  x = 0 ->
  {{ M.AssertZero x value ✅ value }}
| Call (e : M.t A) (value : A) :
  {{ e ✅ value }} ->
  {{ M.Call e ✅ value }}
| Let {B : Set} (e : M.t B) (value : B) (k : B -> M.t A) (value_k : A) :
  {{ e ✅ value }} ->
  {{ k value ✅ value_k }} ->
  {{ M.Let e k ✅ value_k }}
| When (condition : Z) (e : M.t A) (value : A) :
  (condition <> 0 -> {{ e ✅ value }}) ->
  {{ M.When condition e ✅ value }}
```

We prefer to keep this predicate as separated from the determinism one, as the reasoning behind each proof tends to be different. In the determinism case, we want to show that the constraints imply an explicit value for the output variables. For the completeness property, we want to show that a given expression for the output variable indeed makes all the constraints valid.

The completeness statement for the `eval` function is also in the file Garden/OpenVM/BranchEq/core_with_monad.v. We assume that we are given a correct value for the oracle `diff_inv_marker` and say that we return the expected output (the same as in the determinism statement), validating all the constraints:

```
Lemma eval_complete `{Prime goldilocks_prime} {NUM_LIMBS : Z}
    (self : BranchEqualCoreAir.t NUM_LIMBS)
    (a' : Array.t Z NUM_LIMBS)
    (b' : Array.t Z NUM_LIMBS)
    (imm' : Z)
    (diff_inv_marker' : Array.t Z NUM_LIMBS)
    (from_pc' : Z)
    (branch_equal_opcode : BranchEqualOpcode.t)
    (H_NUM_LIMBS : 0 <= NUM_LIMBS) :
  let a := M.map_mod a' in
  let b := M.map_mod b' in
  let imm := M.map_mod imm' in
  let diff_inv_marker := M.map_mod diff_inv_marker' in
  let from_pc := M.map_mod from_pc' in
  let expected_cmp_result := get_expected_cmp_result branch_equal_opcode a b in
  let local :=
    {|
      BranchEqualCoreCols.a := a;
      BranchEqualCoreCols.b := b;
      BranchEqualCoreCols.cmp_result := Z.b2z expected_cmp_result;
      BranchEqualCoreCols.imm := imm;
      BranchEqualCoreCols.opcode_beq_flag :=
        match branch_equal_opcode with
        | BranchEqualOpcode.BEQ => 1
        | BranchEqualOpcode.BNE => 0
        end;
      BranchEqualCoreCols.opcode_bne_flag :=
        match branch_equal_opcode with
        | BranchEqualOpcode.BEQ => 0
        | BranchEqualOpcode.BNE => 1
        end;
      BranchEqualCoreCols.diff_inv_marker := diff_inv_marker;
    |} in
  forall
    (* We assume a [diff_inv_marker] oracle with the following property *)
    (H_diff_inv_marker :
      if Array.Eq.dec a b then
        (* It can be anything in case of equality *)
        True
      else
        (* Otherwise it is the inverse of the difference in exactly one case, ze
        exists k, 0 <= k < NUM_LIMBS /\
        forall i, 0 <= i < NUM_LIMBS ->
        if i =? k then
          BinOp.mul (BinOp.sub (a.[i]) (b.[i])) diff_inv_marker.[i] = 1
        else
          diff_inv_marker.[i] = 0
    ),
  {{ eval self local from_pc ✅
    get_expected_result self local from_pc expected_cmp_result
  }}.
```

We do the proof by step through the code using the definition of the `✅` predicate, with a proof by induction for the loop iterating over the limbs of the arrays `a`, `b`, and `diff_inv_marker`.

Conclusion

We have seen how to translate an example of Plonky3 chip of an existing zkVM to Rocq, and how to formally verify it, first for the soundness, then for the completeness, the functional correctness being included in these two properties with our representation. The `branch_eq` circuit was interesting as it features the three main kinds of variables, input, output, and oracle, and it returns some field expressions in addition to enforcing the constraints. We hope the work on this example will enable verifying more customer circuits in the future.

We are thankful to the Ethereum Foundation for funding this work and this opportunity, and we are happy to hear your thoughts.

🌲