RISC Zero Prover Protocol & Analysis

Tomer Solberg
tomer@ingonyama.com

April 2023

1 Overview

The RISC Zero [JBtRZT23] prover uses a STARK to construct a transparent non-interactive proof of execution of a RISC-V program [1]. It consists of a set-up phase that needs to be done once for each program, a RAP (randomized preprocessing) arithmetization phase, and an AIR-FRI phase which in itself consists of two phases - a DEEP-ALI phase and a batched FRI phase. The protocol works over two finite fields:

1. A prime field $\mathbb{F} = \mathbb{F}_p$ where $p - 2^{31} - 2^{27} + 1$ is the Baby Bear prime
2. An extension field $\mathbb{K} = \mathbb{F}[x]/(x^4 + 11)$

For the purposes of this analysis, we will ignore the set-up phase, as that can be handled by the compiler which compiles the RISC-V program, and does not need to run with each run, so that there is no need for it to be optimized for real-time. The set-up phase generates constraint polynomials, which are used to check that the trace generated by the program was indeed generated correctly.

2 RAP phase

The RAP phase consists of the following steps

1. Trace generation. The trace consists of two tables which we denote t_c, t_d. These tables take values in $\mathbb{F}$, and they encode the values of control (t_c) and data (t_d) registers for every clock cycle in the runtime of the program. Each of their n_c, n_d columns encodes a single register, and each of their 2^h rows encodes a clock cycle (if the number of clock cycle is not a power of 2 then the trace is padded by zeros to the next power of 2).
2. RS-encoding. Each column of the trace is encoded with a Reed-Solomon code of rate $\rho = 1/4$ into a coset called the evaluation domain. This is done via the following steps:
    2.1. Interpolate (=perform an iNTT of) each column of the trace, to get u_c, u_d. This requires the root of unity ω^4, where ω is the root of unity of order 2^{h+2}.

[1] https://github.com/risc0/risc0

2.2. Multiply u_c, u_d by consecutive powers of a constant number $\beta \in \mathbb{F}$ which is not a power of ω.

2.3. Zero-pad the result so that each column is now of length 2^{h+2}.

2.4. Evaluate (=perform an NTT of) each column. The resulting tables w_c, w_d are RS-coded trace.

3. Commit w_c, w_d using Merkle trees. Merkle trees commitments use a hash function which we will assume to be the Poseidon hash function. The poseidon hash function takes 24 inputs in $\mathbb{F}$ and mixes them into 24 outputs in $\mathbb{F}$, of which we usually take only the first 8 to get a 248-bit result. This number of bits gives the maximal soundness of the commitments (in this case gives a security parameter of 124 bits). The commitments to each of the two tables is computed in two steps

   3.1. Hash rows: Each row is divided into blocks of 16 numbers in $\mathbb{F}$ (the last one padded by zeros if needed) which we label b_i, $i = 1, 2, \ldots, \lfloor n/16 \rfloor$, where n is the number of rows. The first hash function takes as inputs the first block, appended by 8 zeros, and returns 24 values. Those are summed with the next block (appended by 8 zeros), and inserted as input to the next hash function. The process repeats iteratively until all of the blocks have been hashed, and the first 8 outputs of the last hash function for each row gives the result for each row. In total this step requires $2^{h+2} * \lfloor n/16 \rfloor$ Poseidon hash computations for each table, and the output is a table of 2^{h+2} rows by 8 columns regardless of n.

   3.2. Hash fold: This step takes the output of the previous step and treats it as a single vector of length 2^{h+2} with 256-bit values. Poseidon is then used as a binary $(2 \to 1)$ function to construct a Merkle Tree. This requires $2^{h+2} - 1$ hash computations, and the top ℓ layers of the tree are taken to be the commitment for each table, which we denote by $[w_c], [w_d]$. The value of ℓ is set such that 2^ℓ is larger than the number of queries made to the Merkle tree, which we set to be 50, so that $\ell = 6$.

4. Generate accumulation table t_a. This table has 2^h rows and $n_a = n_p + n_l$ columns, where the p columns encode permutation constraints on the control and data tables, and the l columns encode lookup constraints on them. These are generated in a few steps:

   4.1. Random number generation: n_a random numbers in $\mathbb{K}$ are generated, one for each column. This is done by running the Poseidon hash function over the previous transcript $([w_c], [w_d])$.

   4.2. Permutation column generation: given two columns a, b of the data table that must be a permutation of each other, a column c is computed using the random number α such that its i-th value is given by

$$c_i = \frac{\prod_{j=i}^{i-1}(\alpha + a_j)}{\prod_{j=i}^{i-1}(\alpha + b_j)} \tag{2.1}$$

   4.3. Lookup column generation: given a columns a of the data table that must take values that appear in some lookup table b, a column d is computed using the

random numbers γ, δ such that its i-th value is given by

$$d_i = \frac{(1+\delta)^{i-1}\prod_{l=1}^{i-1}(\gamma + a_l)(\gamma(1+\delta) + b_l + \delta b_{l+1})}{\prod_{l=1}^{i-1}(\gamma(1+\delta) + s_l + \delta s_{l+1})(\gamma(1+\delta) + s_{N+l} + \delta s_{N+l+1})} \tag{2.2}$$

where s is the sorted union of a, b.

5. RS-encode and commit t_a: Repeat steps 2,3 on t_a to compute $u_a, w_a, [w_a]$.

This phase is described schematically in Figure 1.

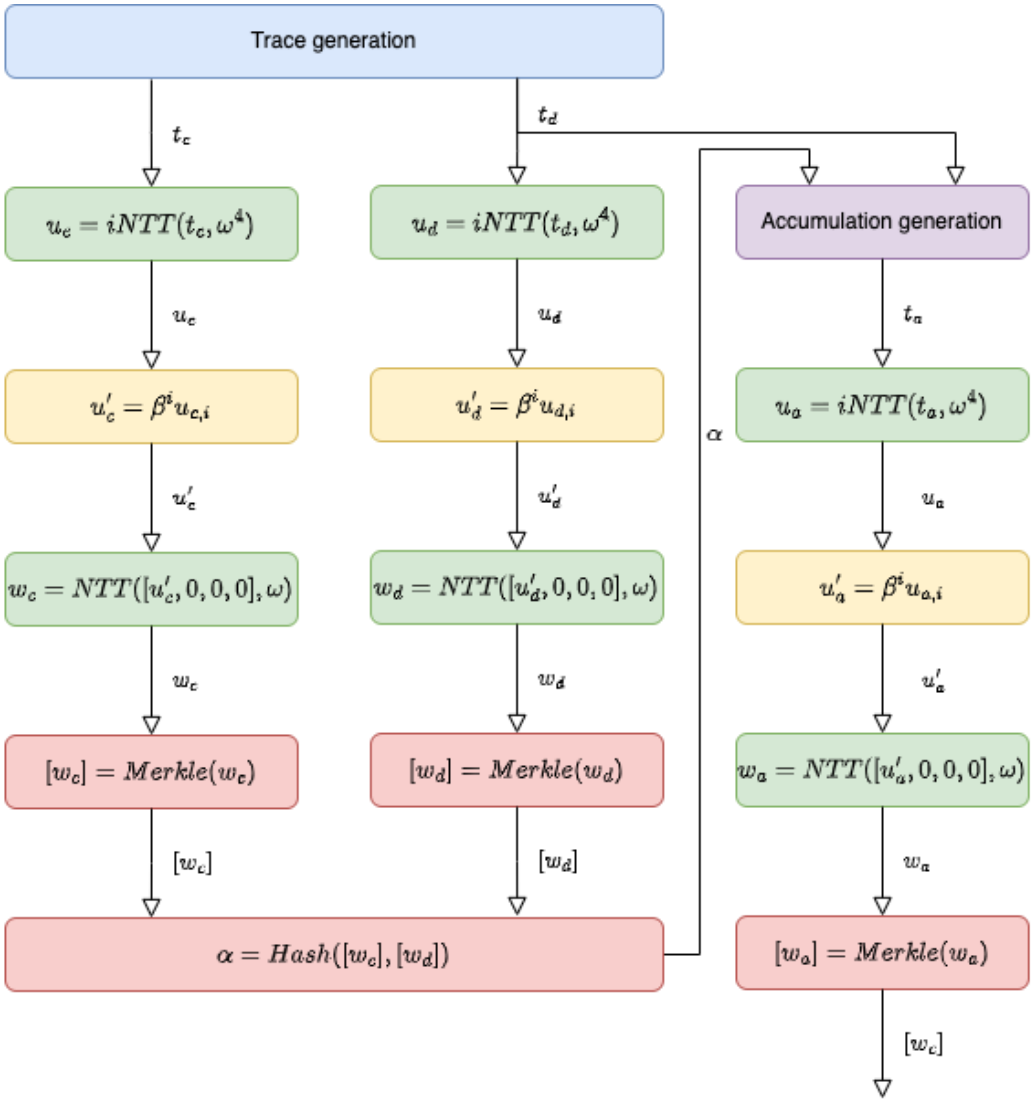

Figure 1: Schematic dataflow of the RAP phase. Colors indicate the type of computation performed in each block: red for hashes, green for NTT, Yellow for linear combination with powers, blue for trace generation, purple for accumulation table

3 DEEP-ALI phase

The DEEP-ALI phase takes as input the constraints polynomials that were used in the set-up phase, mixes them together, and allows for their verification. It consists of the following steps

1. Generate a random number $\alpha_c \in \mathbb{K}$ using $Hash([w_c], [w_d], [w_a])$.
2. Mix the constraint polynomials $c_i(x)$ by taking the sum $C(x) = \sum_i \alpha_c^i c_i(x)$. This will be a $5 * 2^h$ degree polynomial
3. Divide the vanishing polynomial $Z(x) = x^{2^h} - 1$ and break into 4 polynomials of degree 2^h. Which we denote u_v.
4. Evaluate u_v on the coset and commit by repeating steps 2.2-2.4,3 of the RAP phase to get $w_v, [w_v]$.
5. Generate a random number $z \in \mathbb{K}$ using $Hash([w_c], [w_d], [w_a], [w_v])$.
6. For each column $P_i(x)$ in u_c, u_d, u_a, u_v, evaluate $\bar{P}_i(x) = P_i(z) + \frac{P_i(\omega^{-4}z) - P_i(z)}{z\omega^{-4} - z}(x - z)$
7. For each column $P_i(x)$ generate the DEEP polynomial $d_i(x)$ as

$$d_i(x) = \frac{P_i(x) - \bar{P}_i(x)}{(x - z)(x - \omega^{-4}z)} \tag{3.1}$$

The transcript at the end of this phase consists of

$$[w_c], [w_d], [w_a], [w_v], \bar{P}_i(x) \tag{3.2}$$

This phase is described schematically in Figure 2.

4 Batched FRI

This phase is used to test that the polynomials $d_i(x)$ are indeed of low degree. It consists of the following steps

1. Generate a random number $\alpha_f \in \mathbb{K}$ using $Hash([w_c], [w_d], [w_a], [w_v], \bar{P}_i(x))$.
2. Mix together $d_i(x)$ by $d(x) = \sum_i \alpha_f^i d_i(x)$
3. Re-index by defining $f^{(0)}(x) = d(\beta^{-1}x)$
4. Evaluate $f^{(0)}(x)$ on the coset and commit by repeating steps 2.3-2.4,3 of the RAP phase to get $[f^{(0)}]$.
5. Perform $r = \lfloor (h + 2)/4 - 2 \rfloor$ rounds of FRI-commit (this number ensures that for a 16-fold each round, FRI will stop once the degree is 256 or lower. Each round consists of
   5.1. Generate a random number $\alpha_j \in \mathbb{K}$ using Hash of the previous transcript

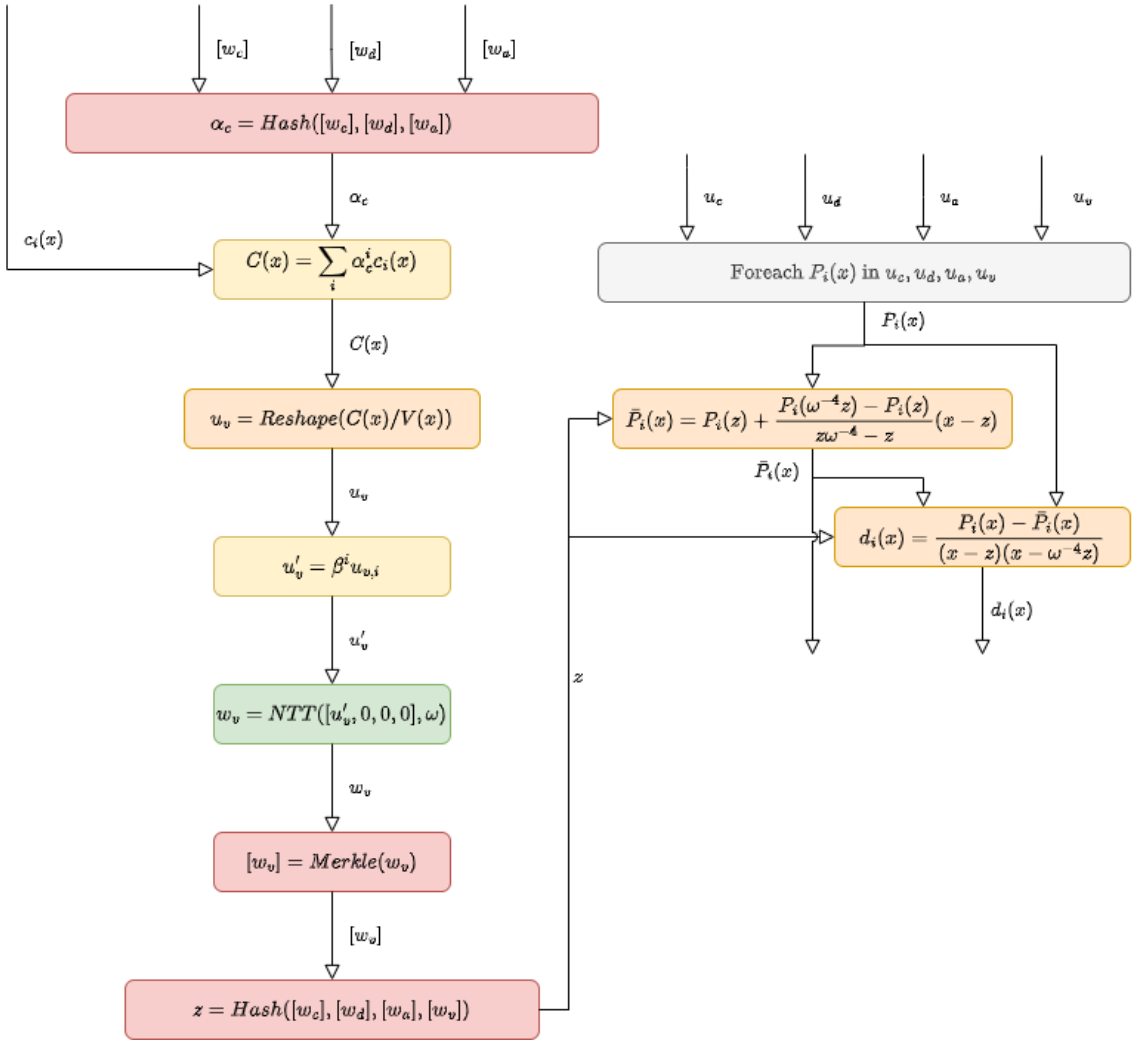

Figure 2: Schematic dataflow of the DEEP ALI phase. Colors indicate the type of computation performed in each block: red for hashes, green for NTT, Yellow for linear combination with powers, orange for field and polynomial division

5.2. Split $f^{(j-1)}$ into 16 parts and sum them with coefficients which are consecutive powers of α_j to get $f^{(j)}(x)$.

5.3. Evaluate $f^{(j)}(x)$ on the coset and commit by repeating steps 2.3-2.4,3 of the RAP phase to get $[f^{(j)}]$.

6. Add to the transcript the coefficients of $f^{(r)}(x)$

7. Perform a number (Typically 50) of FRI queries. Each query consists of

7.1. Generate g_0 which is a random power of ω, and g_j for every $j = 1, \ldots, r$ by $g_j = g_j^{16}$

7.2. Generate a coset βg_j^k for each $j = 0, \ldots, r-1$

7.3. Evaluate $f^{(j)}(\beta g_j^k)$ and $f^{(r)}(g_r)$ and add them to the transcript

This phase is described schematically in Figure 3.

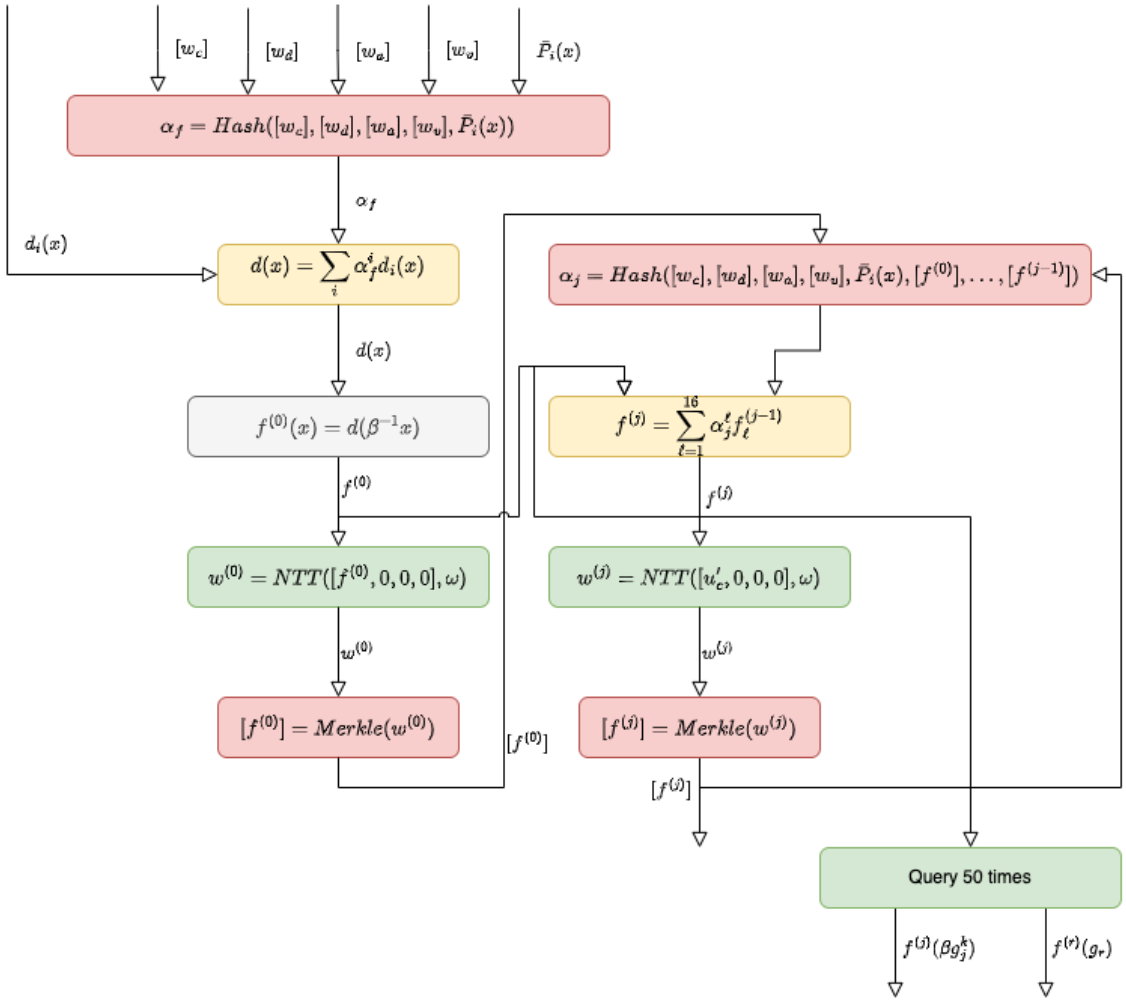

Figure 3: Schematic dataflow of the FRI phase. Colors indicate the type of computation performed in each block: red for hashes, green for NTT, Yellow for linear combination with powers

5 Performance analysis

Risc Zero supply several example programs of different sizes, along with a performance analysis. A typical example that we use for our analysis is the "chess" example, which has $2^{18} = 256k$ clock cycles. The number of columns for the control, data, and accumulation tables are 16, 222, and 36 respectively. Performance analysis sets the prover time of this example at 3.4 seconds on a RTX-3090Ti GPU. Examining the times of different components of the prover run time we get

- Set-up takes about 6%
- Trace generation takes about 13%

– NTTs take about 17%

– Hashes take about 37%

– Accumulation table generation takes about 5%

– Other computations take about 22%

We see that the main bottleneck is the Hash computation. For the "chess" example, one can count how many Poseidon hashes are performed, and get about 24M hashes per proof, which take a total of $0.37 * 3.4 = 1.26$ seconds on a RTX-3090Ti GPU.

6 Poseidon hashes on FPGA

The Poseidon implementation in RISC Zero takes 24 values in $\mathbb{F}$, runs them through 4 full rounds, then 21 partial rounds, and finally 4 more full rounds. A full round consists of an s-box (x^7) on every data wire, which take $4 * 24$ 32-bit modular multipliers, and an MDS matrix, which take 24^2 32-bit modular multipliers. A partial round consists of an s-box on a single data wire, which takes 4 32-bit modular multipliers, and an optimized MDS matrix which can be calculated using 47 32-bit modular multipliers. In total this requires $8 * (4 * 24 + 24^2) + 21 * (4 + 47) = 6447$ modular multipliers. This can be optimized by considering that when multiplying the data vector by the MDS matrix, we can take the 24^2 multipliers to be integer (not modular), and only take the modulu of the 24 results at the end. A modular multiplier uses 3 integer multipliers, so that the total number of modular multipliers used in MDS matrix multiplication in full rounds can be reduced from 24^2 to $24 * (8 + 2)$, and in partial rounds from 47 to 32, reducing the total number of 32-bit modular multipliers to $8 * (4 * 24 + 24 * 10) + 21 * (4 + 32) = 3444$.

Using Karatsuba, a 32-bit integer multiplier can be implemented with 3 16-bit DSP blocks, so that a 32-bit modular multiplier requires 9 DSP blocks. A C1100 Xilinx FPGA has about 6000 DSP blocks, so that one can fit on it about 1/8 of the required multipliers for Poseidon calculation. This can be used to implement a machine which pipelines Poseidon calculation and produce a hash every 8 clock cycles. Such a machine can calculate the 24M hashes required by the example analyzed above in (assuming 500MHz clock frequency)

$$\frac{24M * 8}{500MHz} \approx 0.4sec \tag{6.1}$$

About 3x faster than the GPU.

Acknowledgements

The author would like to thank Michael Asa and Jeremy Felder for their insights and assistance in analyzing the RISC Zero protocol and writing this review, as well as to the RISC Zero team

References

[JBtRZT23] Paul Gafni Jeremy Bruestle and the RISC Zero Team. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity, 2023. `https://www.risczero.com/proof-system-in-detail.pdf`.