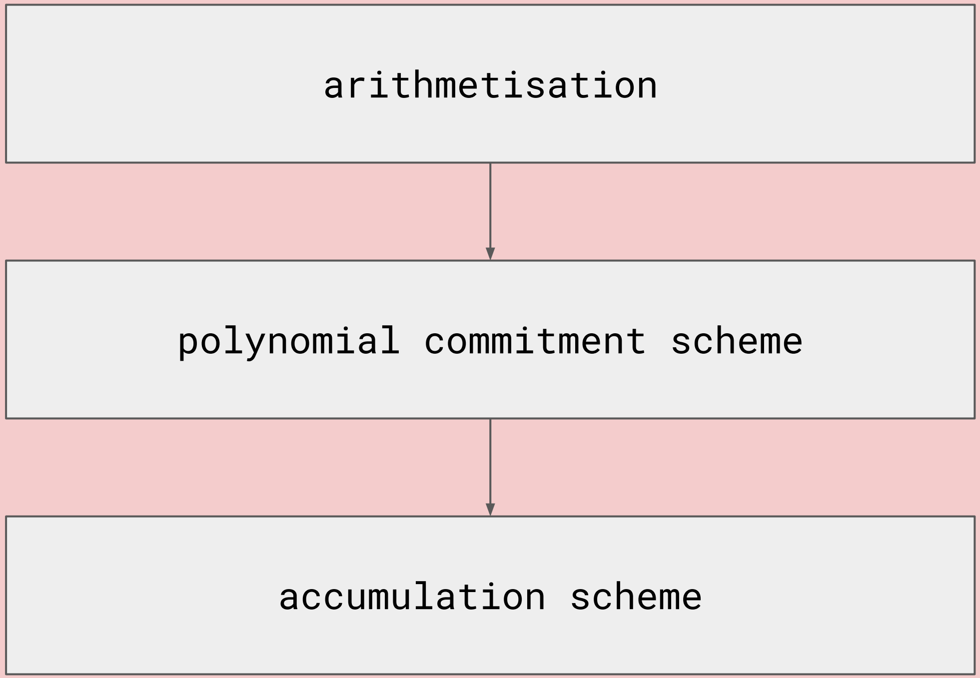
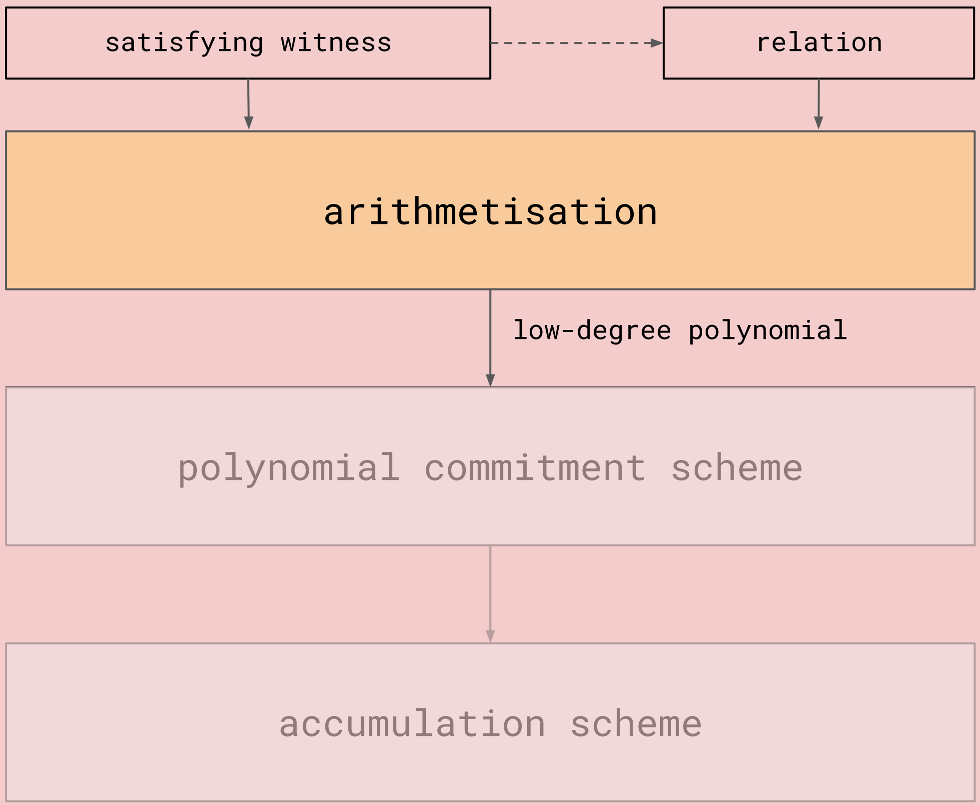
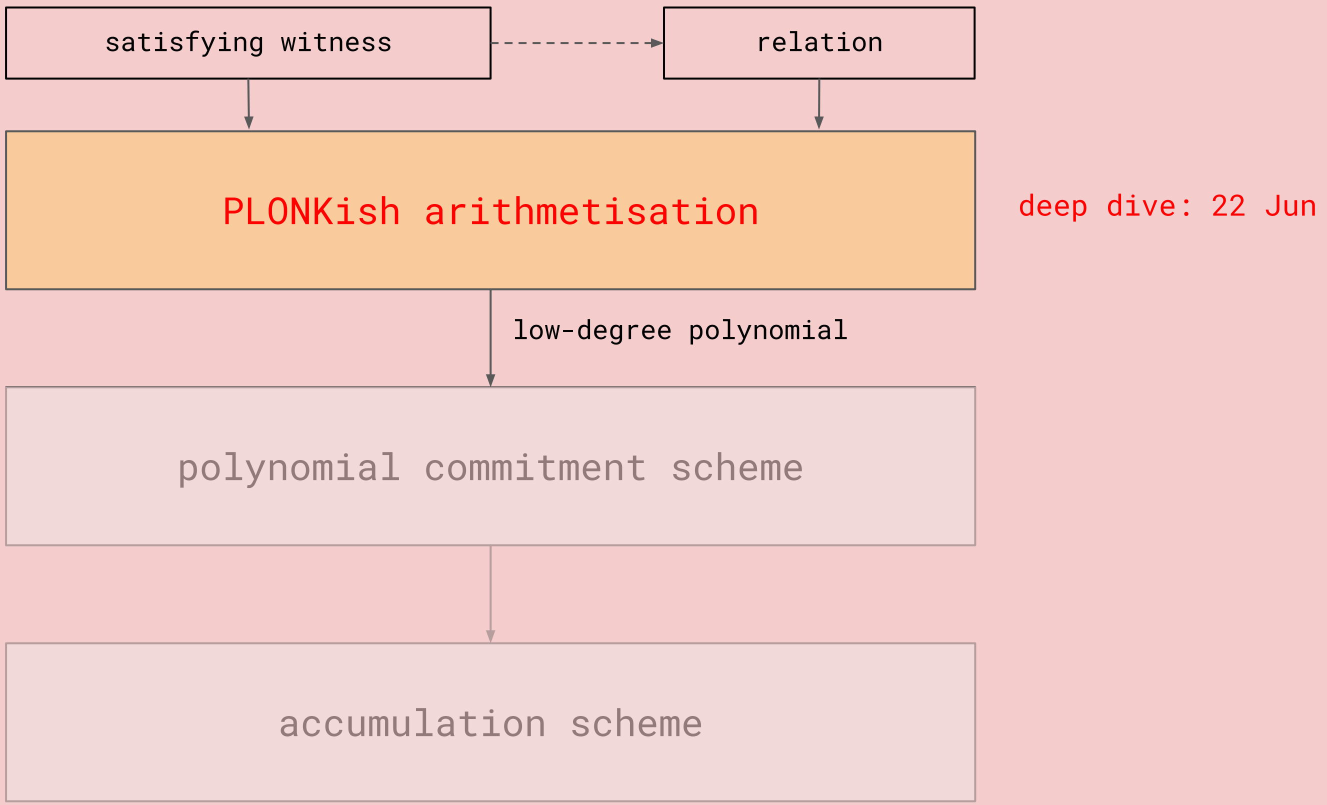
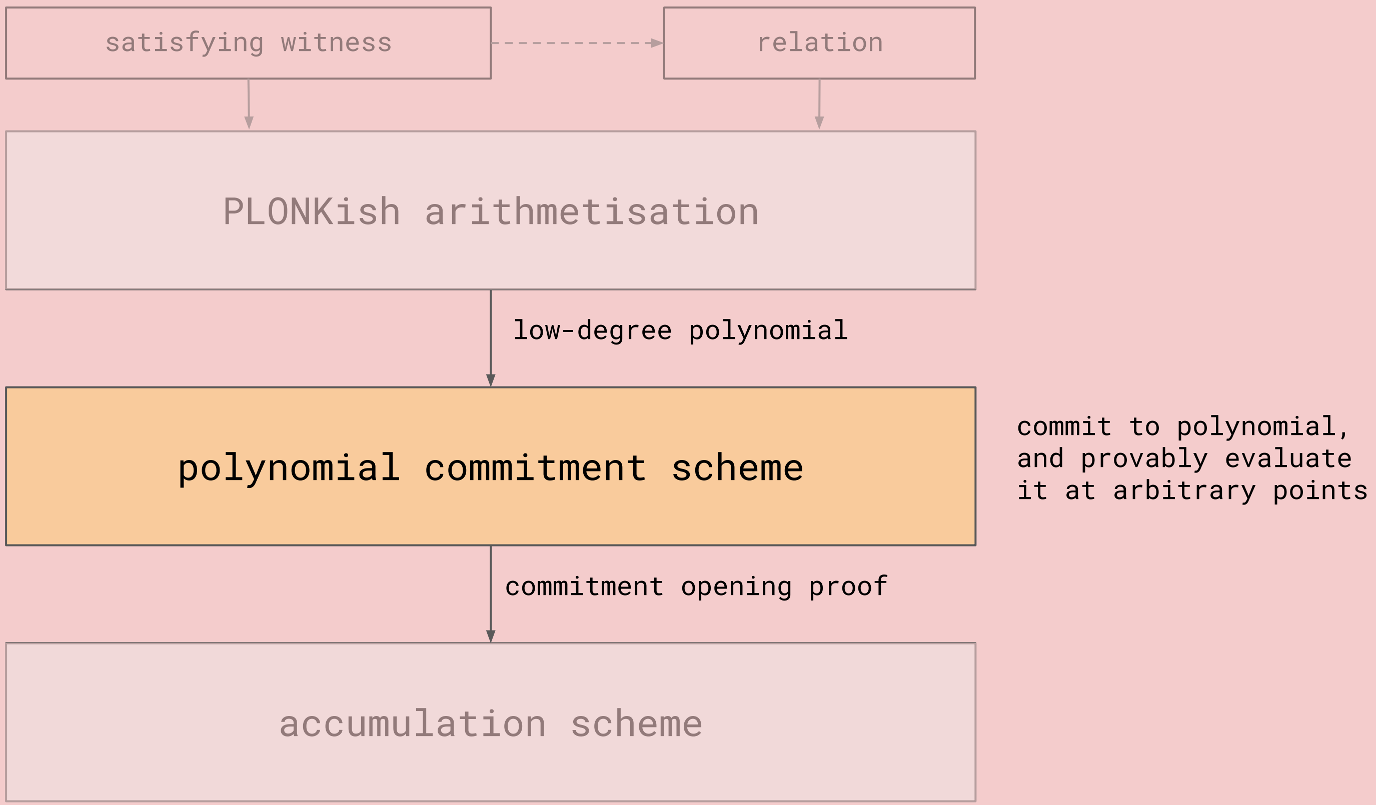
Intro to PLONKish/halo2

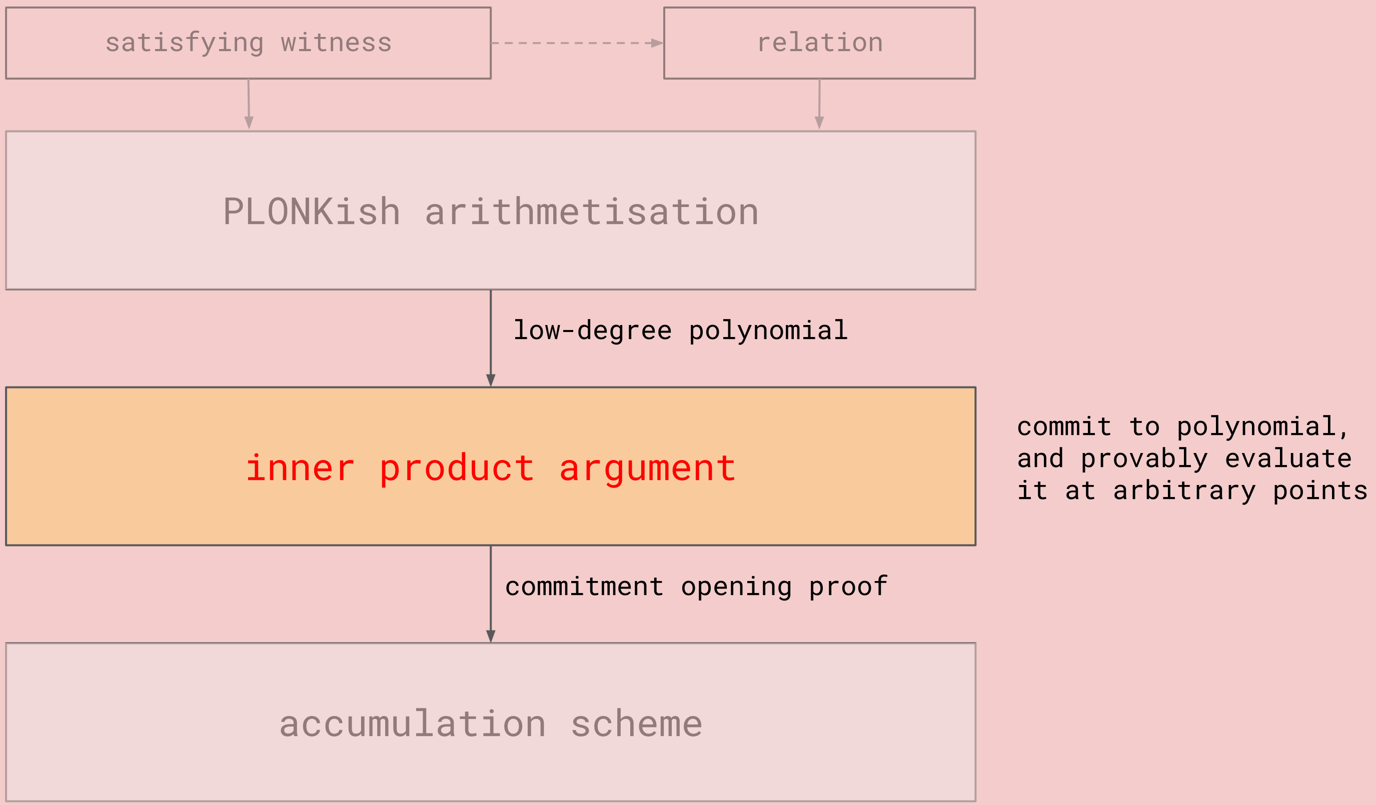
0xPARC Halo2 Learning Group
13 Jun 2022

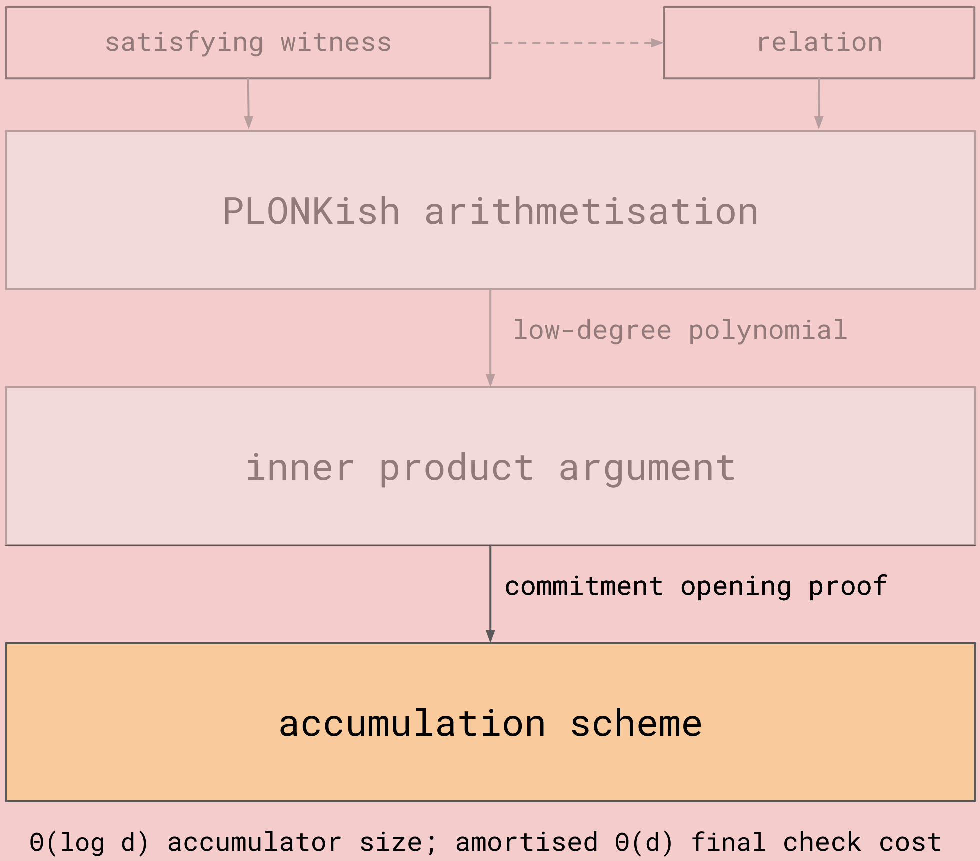

arithmetisation
polynomial commitment scheme
accumulation scheme

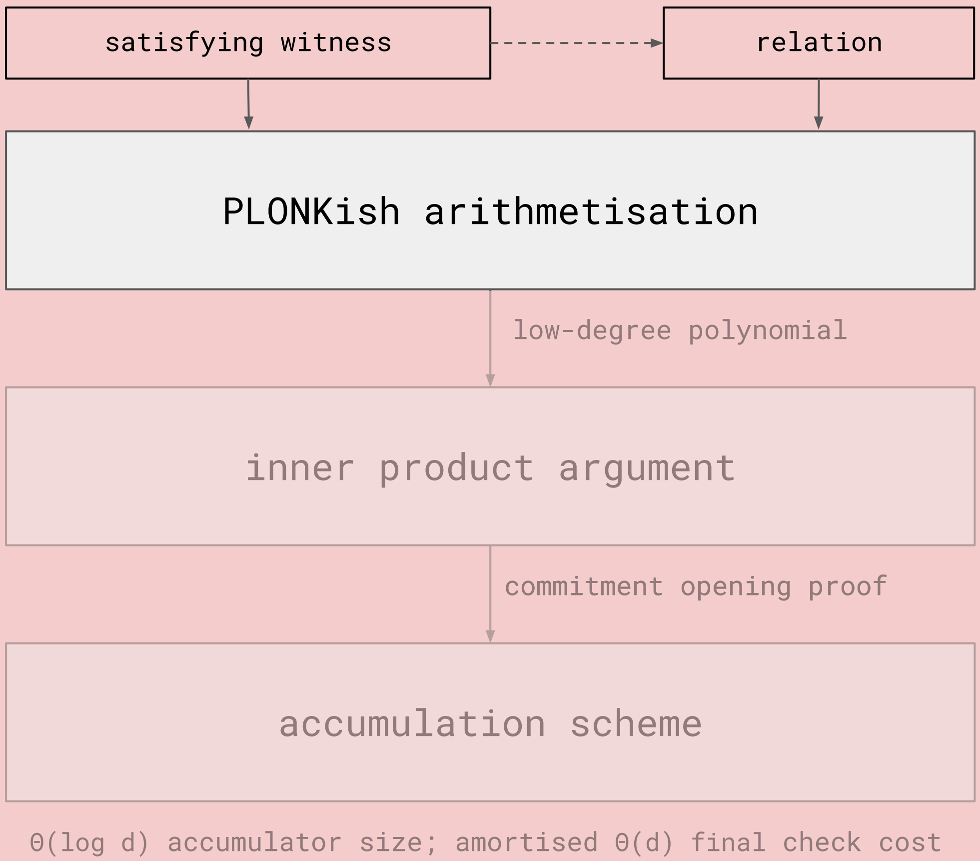

satisfying witness
relation
arithmetisation
low-degree polynomial
polynomial commitment scheme
accumulation scheme


satisfying witness
relation
PLONKish arithmetisation
deep dive: 22 Jun
low-degree polynomial
polynomial commitment scheme
accumulation scheme

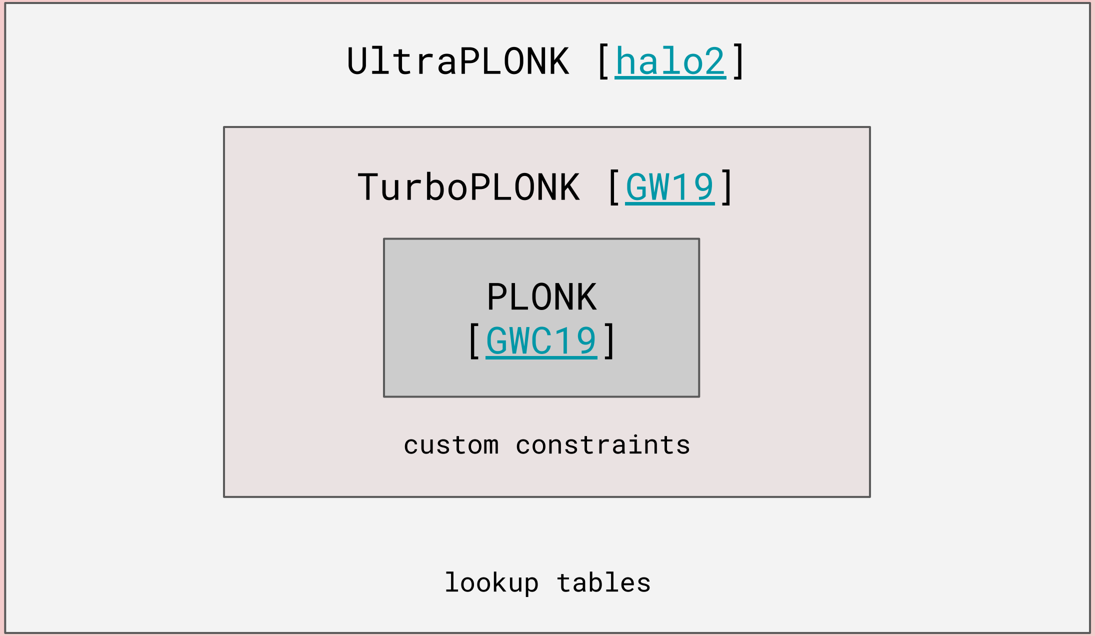

satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
polynomial commitment scheme
commit to polynomial,
and provably evaluate
it at arbitrary points
commitment opening proof
accumulation scheme


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
inner product argument
commit to polynomial, and provably evaluate it at arbitrary points
commitment opening proof
accumulation scheme


satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
inner product argument
commitment opening proof
accumulation scheme
O(log d) accumulator size; amortised O(d) final check cost

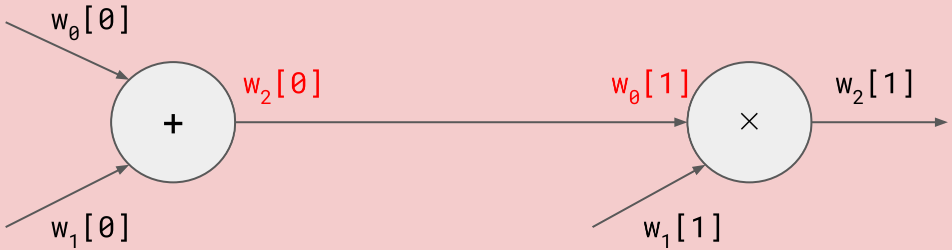

satisfying witness
relation
PLONKish arithmetisation
low-degree polynomial
inner product argument
commitment opening proof
accumulation scheme
Θ(log d) accumulator size; amortised Θ(d) final check cost

PLONKish arithmetisation



PLONKish arithmetisation (vanilla)

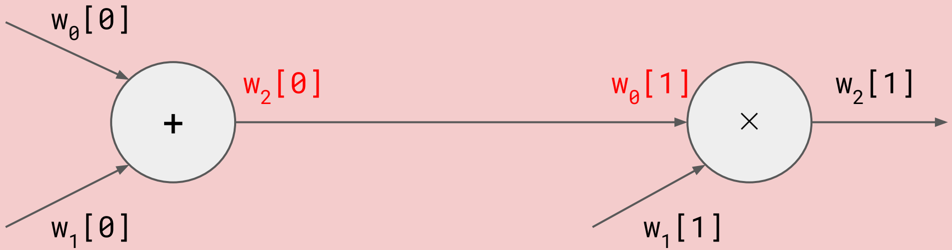
PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]

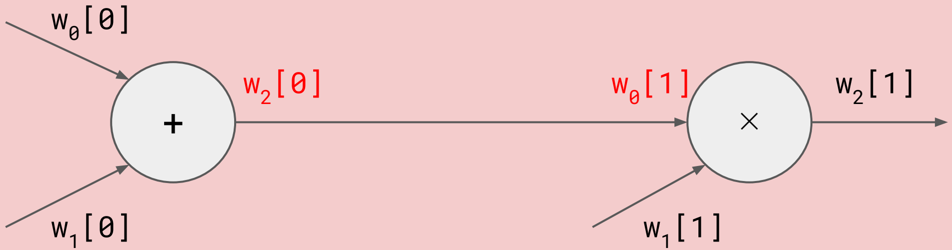


- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



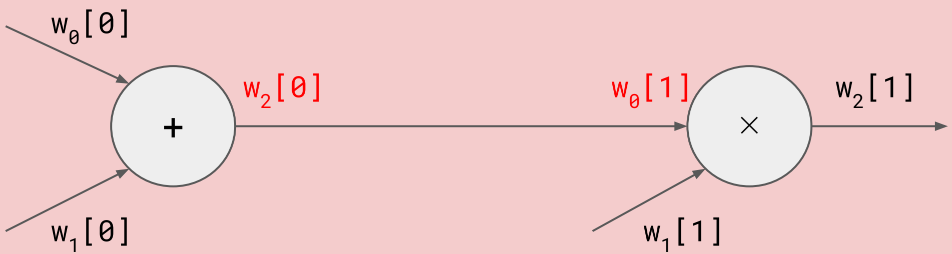
- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

PLONKish arithmetisation (vanilla)

PLONK [GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

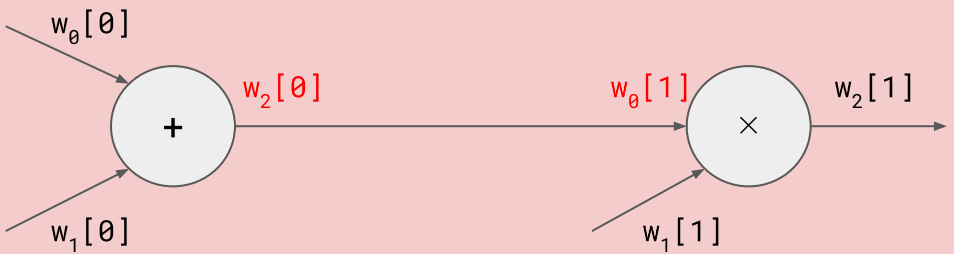
"local" consistency check: are all gate equations satisfied?

$$\boxed{q_L} \cdot x_a + \boxed{q_R} \cdot x_b + \boxed{q_O} \cdot x_c + \boxed{q_M} \cdot (x_a x_b) = 0$$

preprocessed selector polynomials

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

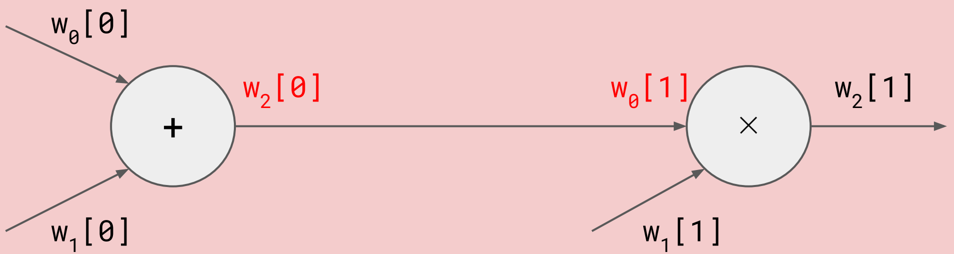
"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

add: $1 \cdot x_a + 1 \cdot x_b + (-1) \cdot x_c + 0 \cdot (x_a x_b) = 0$

PLONKish arithmetisation (vanilla)

PLONK
[GWC19]



- **gates** take two values as **inputs**, either **add** or **multiply** them, and then emit the result through an **output** wire;

"local" consistency check: are all gate equations satisfied?

$$q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$$

add: $1 \cdot x_a + 1 \cdot x_b + (-1) \cdot x_c + 0 \cdot (x_a x_b) = 0$

mul: $0 \cdot x_a + 0 \cdot x_b + (-1) \cdot x_c + 1 \cdot (x_a x_b) = 0$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} + \underbrace{q_{bool} \cdot (a_0 \cdot a_0 - a_0)}_{\text{bool gate}} = 0$$

PLONKish arithmetisation (custom gates)

TurboPLONK
[GW19]

vanilla PLONK gate: $q_L \cdot x_a + q_R \cdot x_b + q_O \cdot x_c + q_M \cdot (x_a x_b) = 0$

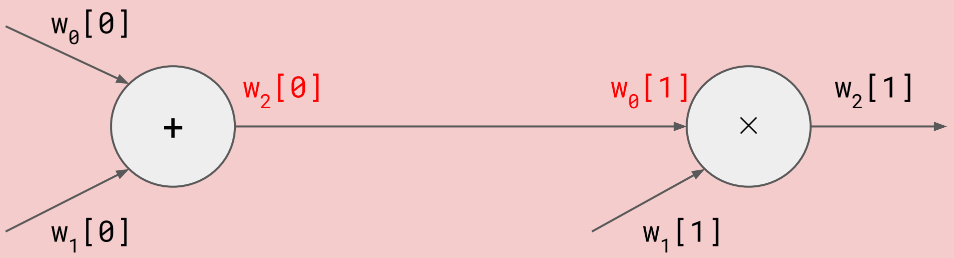
custom gates (arbitrary linear combinations):

$$\underbrace{q_{add} \cdot (a_0 + a_1 - a_2)}_{\text{add gate}} + y \cdot \underbrace{q_{mul} \cdot (a_0 \cdot a_1 - a_2)}_{\text{mul gate}} + y^2 \cdot \underbrace{q_{bool} \cdot (a_0 \cdot a_0 - a_0)}_{\text{bool gate}} = 0$$

verifier challenge to keep gates linearly independent

PLONKish arithmetisation (permutation)

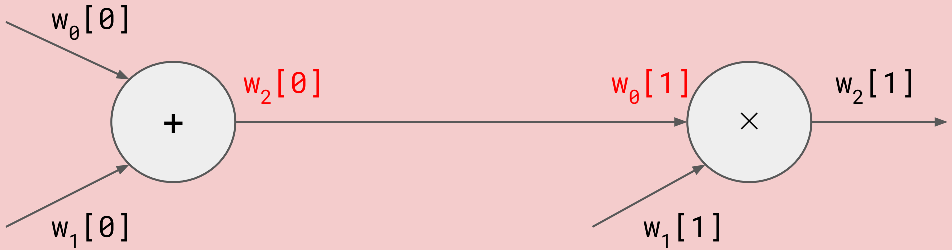
PLONK
[GWC19]



- **wires** carry values into and out of gates

PLONKish arithmetisation (permutation)

PLONK
[GWC19]



- **wires** carry values into and out of gates

"global" consistency check: do the wires correctly join the gates together?

* in Groth16, routing is baked into the trusted setup; we can't do this for universal SNARKs

PLONKish arithmetisation (permutation)

PLONK
[GWC19]

w_0	w_1	w_2	gate
$w_0[0]$	$w_1[0]$	$w_2[0]$	+
$w_0[1]$	$w_1[1]$	$w_2[1]$	×

each wire (column i) is encoded as a Lagrange polynomial w_i over the powers (rows) of an n^{th} root of unity $\{1, \omega, \ldots, \omega^{n-1}\}$, where $\omega^n = 1$:

$$\boxed{w_i(\omega^j) = w_i[j]}$$

PLONKish arithmetisation (permutation)

PLONK
[GWC19]

w_0	w_1	w_2	gate
$w_0[0]$	$w_1[0]$	$w_2[0]$	+
$w_0[1]$	$w_1[1]$	$w_2[1]$	×

each wire (column i) is encoded as a Lagrange polynomial w_i over the powers (rows) of an n^{th} root of unity $\{1, \omega, \ldots, \omega^{n-1}\}$, where $\omega^n = 1$:

$$\boxed{w_i(\omega^j) = w_i[j]}$$

to enforce equality of wires, use permutation argument (deep-dive); show that swapping $w_2(\omega^0)$ with $w_0(\omega^1)$ doesn't change the polynomials.

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1
42	SHA(42)
0	0
69	SHA(69)
…	…
0	0

problem: SHA is expensive to do in-circuit

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

solution: load precomputed SHA (e.g. for 8-bit values) as lookup table

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

$(q_{lookup} \cdot w_0,\ t_0)$
$(q_{lookup} \cdot w_1,\ t_1)$

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

$$(q_{lookup} \cdot w_0 + (1 - q_{lookup}) \cdot 0, \quad t_0)$$
$$(q_{lookup} \cdot w_1 + (1 - q_{lookup}) \cdot \mathrm{SHA}(0), \quad t_1)$$

lookup default value when q_{lookup} is not enabled,
so that lookup argument passes on every row

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

the lookup argument is a more permissive version of the permutation argument. it enforces that:

every cell in a set of **input columns** is equal to
some cell in a set of **table columns**

PLONKish arithmetisation (lookup)

UltraPLONK
[halo2]

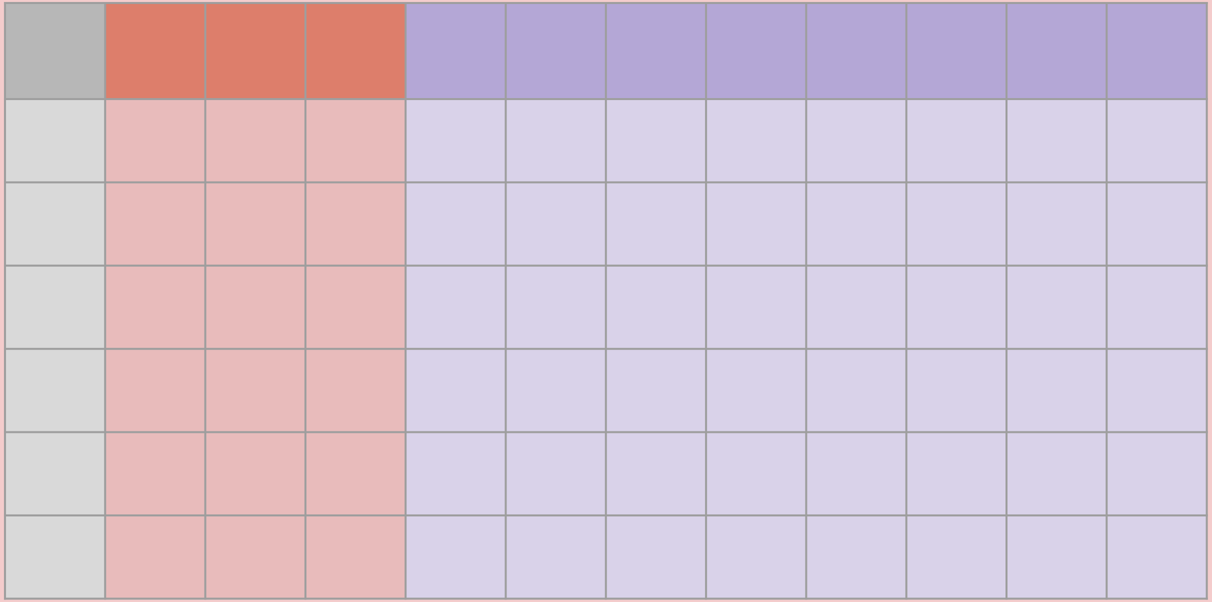
w_0	w_1	q_{lookup}	t_0	t_1
42	SHA(42)	1	0	SHA(0)
0	0	0	1	SHA(1)
69	SHA(69)	1	2	SHA(2)
…	…	…	…	…
0	0	0	255	SHA(255)

the lookup argument is a more permissive version of the permutation argument. it enforces that:

every expression in a set of **input columns** is equal to
some expression in a set of **table columns**

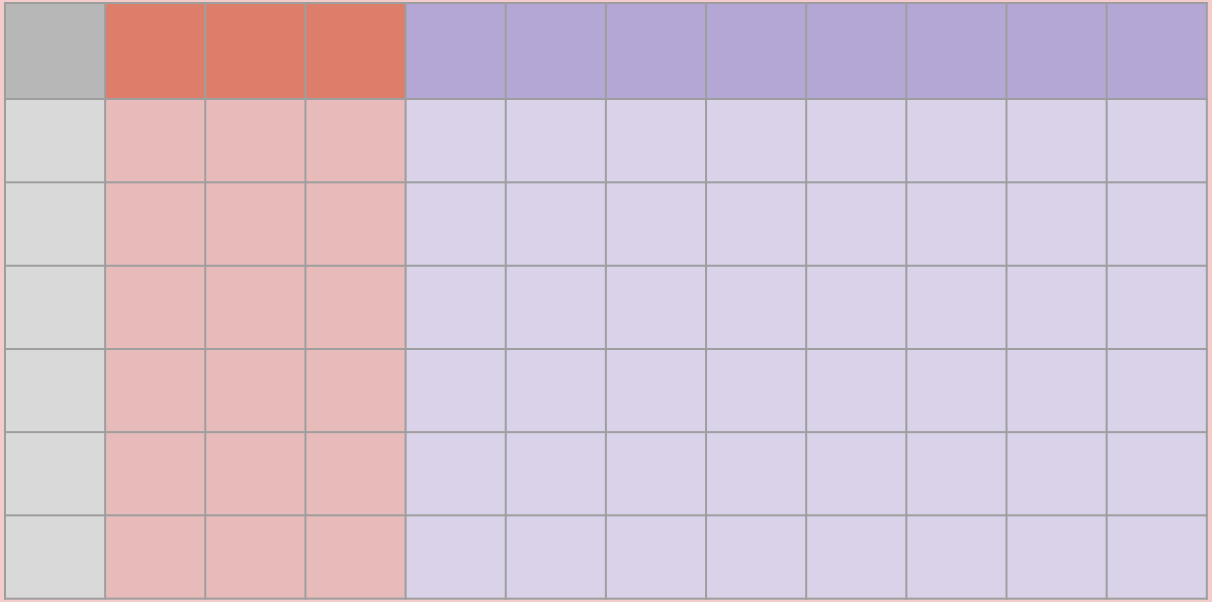
more on this in the deep-dive!

PLONKish arithmetisation



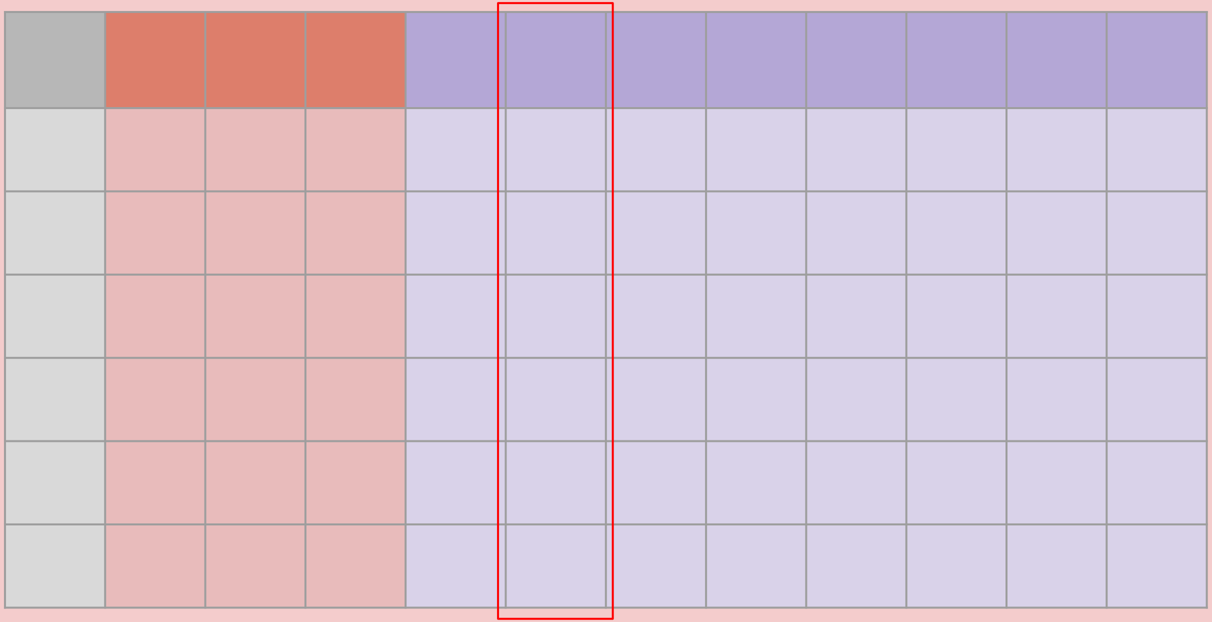
we conceptualise the circuit as a **matrix** of *m* columns and *n* rows

PLONKish arithmetisation



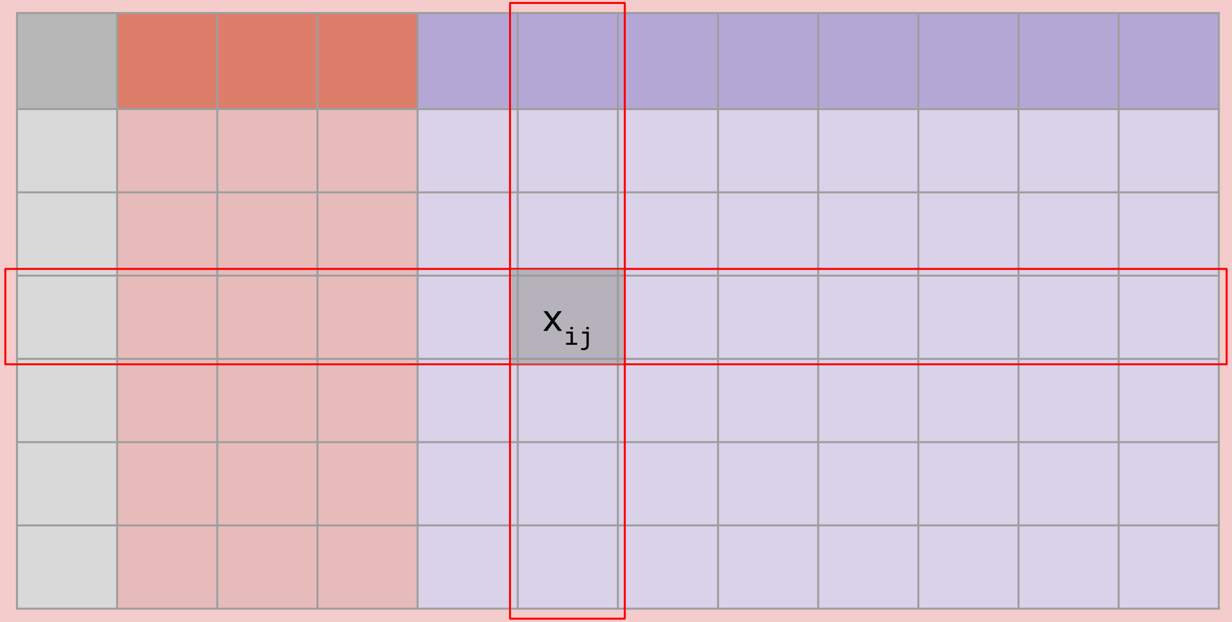
we conceptualise the circuit as a **matrix** of m columns and n rows, over a given **finite field** $\mathbb{F}$ (so the cells contain elements of $\mathbb{F}$)

PLONKish arithmetisation



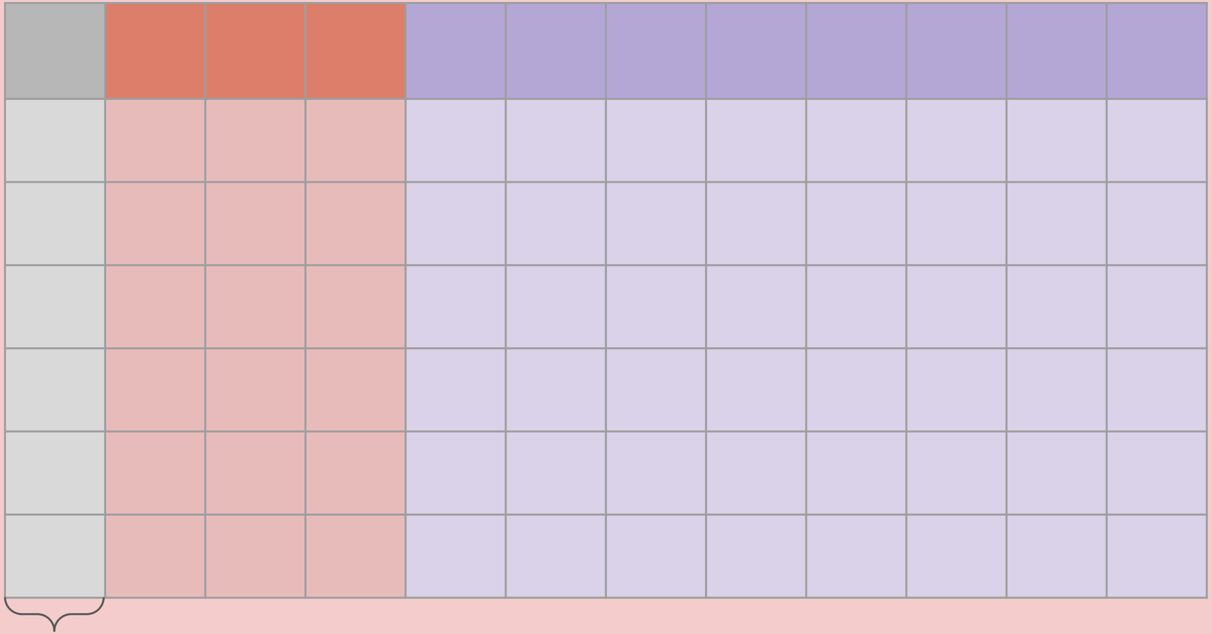
each column j corresponds to a Lagrange interpolation polynomial $p_j(X)$

PLONKish arithmetisation



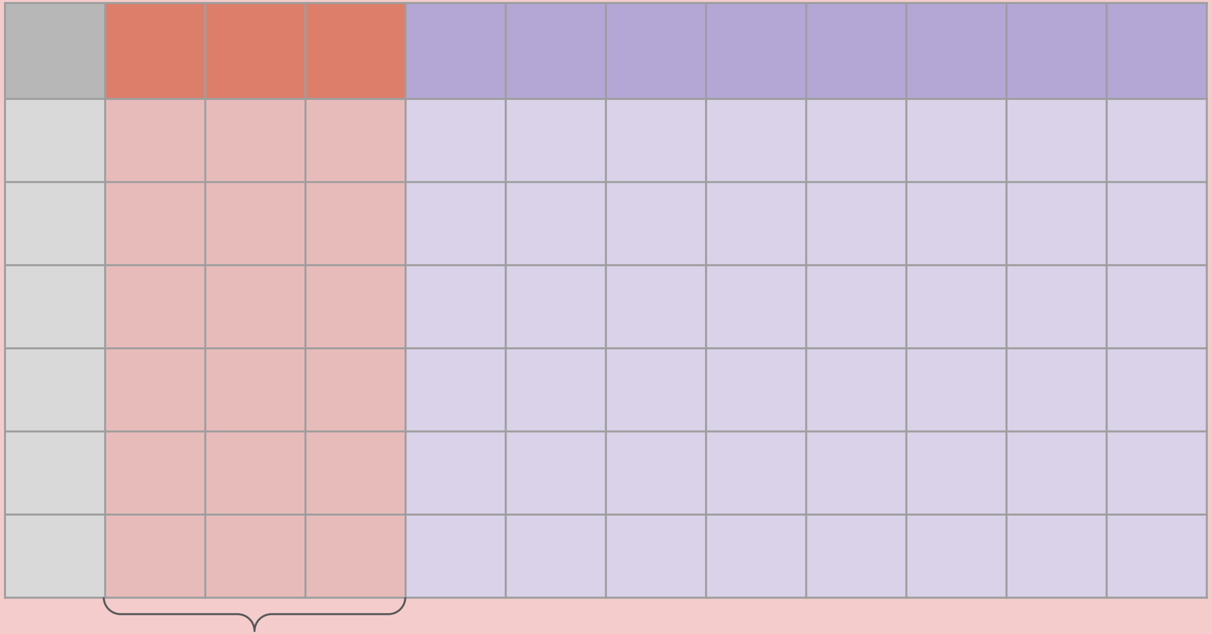
each column j corresponds to a Lagrange interpolation polynomial $p_j(X)$ evaluating to $\mathbf{p_j(\omega^i) = x_{ij}}$, where ω is the n^{th} primitive root of unity.

PLONKish arithmetisation



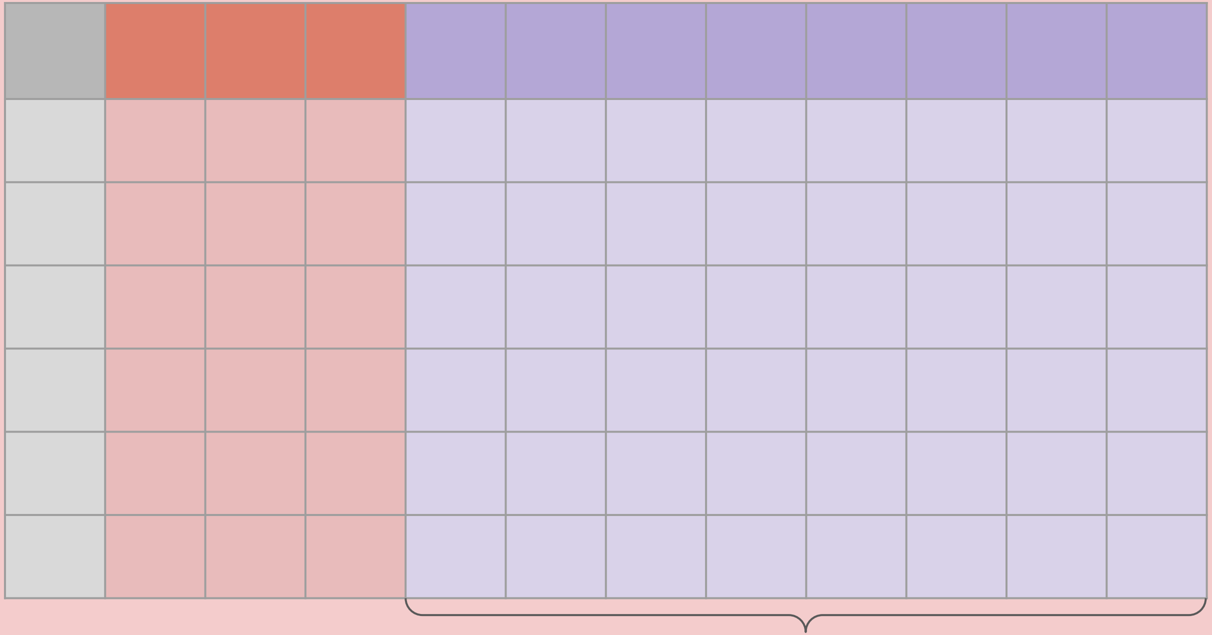
instance columns contain inputs **shared** between prover/verifier (e.g. public inputs)

PLONKish arithmetisation



advice columns contain private values witnessed by the prover

PLONKish arithmetisation



example: Fibonacci sequence

write this in tomorrow's session!

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1 +	1 =	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	2	1
	2	3	5	1
13	5	8	13	0

$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$

$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$

$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$

global permutation: $a_2[i] = a_0[i + 1]$

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
1	1	1	$\boxed{2}$	1
	$\boxed{2}$	3	$\boxed{5}$	1
13	$\boxed{5}$	8	13	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

global permutation: $a_2[i] = a_0[i + 1]$

exercise: can you see how to constrain this locally (using q_{fib})?

example: Fibonacci sequence

i_0	a_0	a_1	a_2	q_{fib}
$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	2	1
	2	3	5	1
$\boxed{13}$	5	8	$\boxed{13}$	0

$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{2,cur}) = 0$$

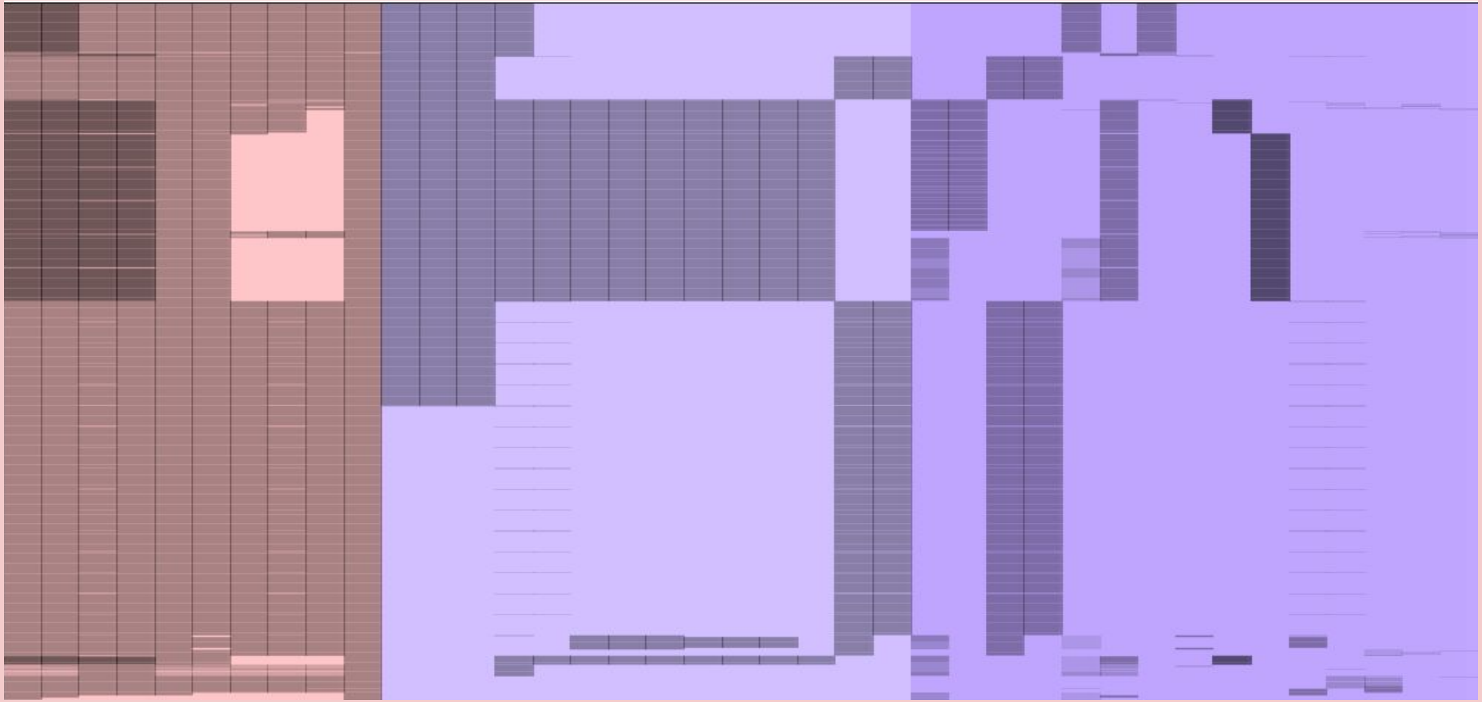
$$q_{fib} \cdot (a_{0,cur} + a_{1,cur} - a_{0,next}) = 0$$

$$q_{fib} \cdot (a_{1,cur} + a_{2,cur} - a_{1,next}) = 0$$

global permutation:

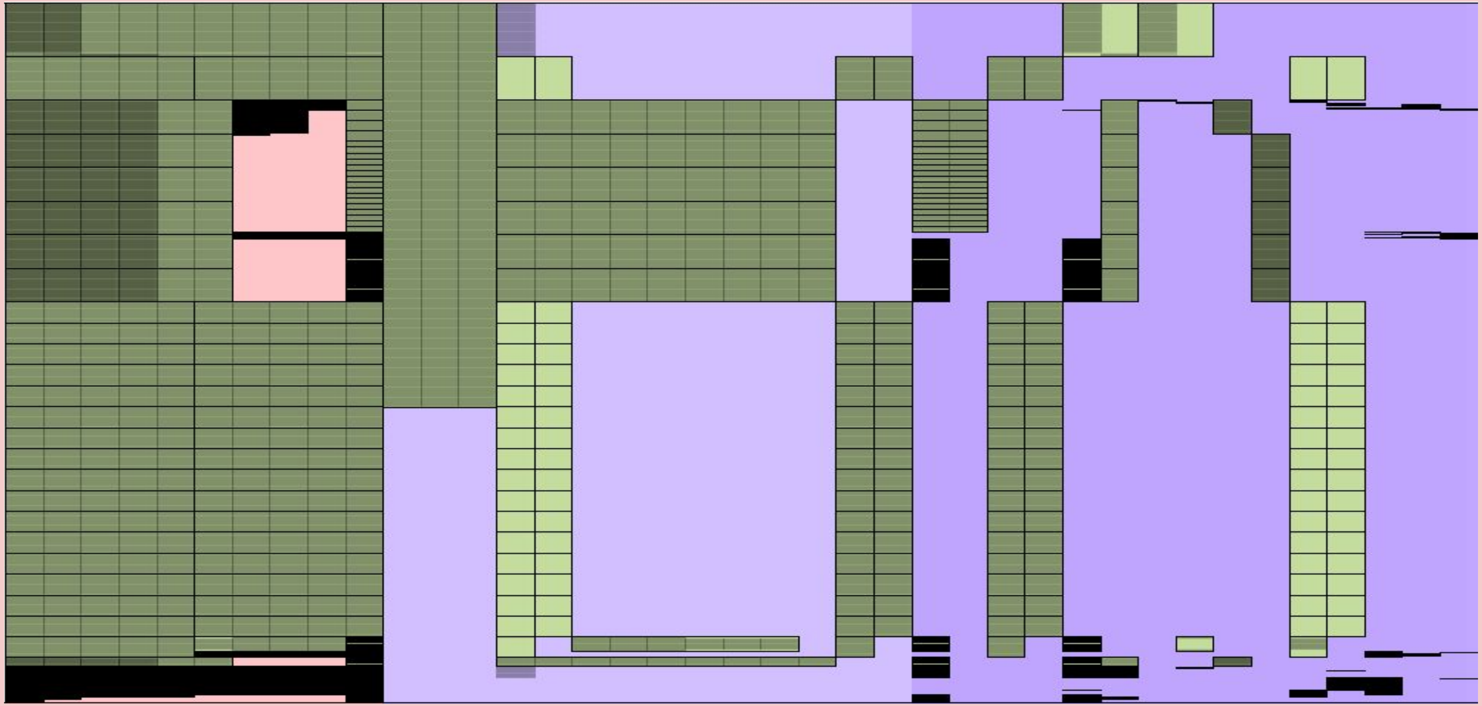
- $i_0[0] = a_0[0]$ // initialisation
- $i_0[0] = a_1[0]$ // initialisation
- $i_0[2] = a_2[2]$ // output

PLONKish arithmetisation



how do we optimise **global** layout while reasoning about **local** offsets?

PLONKish arithmetisation



regions!

regions

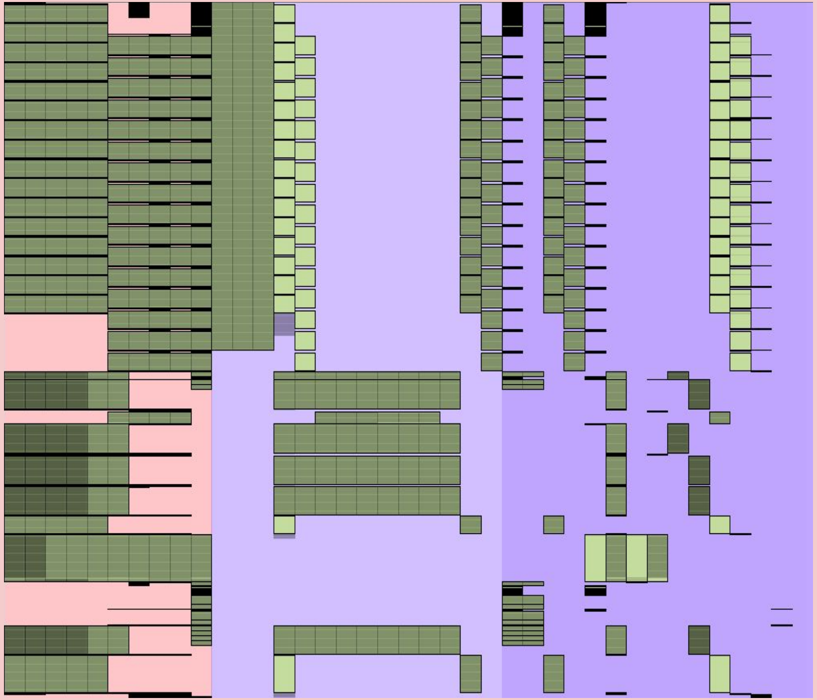
"**regions**" are the boundary between **gates** and the global circuit **layouter**

- a block of assignments preserving **relative offsets**: easy to reason about how gates apply within the region

- not affected by offsets in other regions: can be **freely rearranged** to optimise global space usage

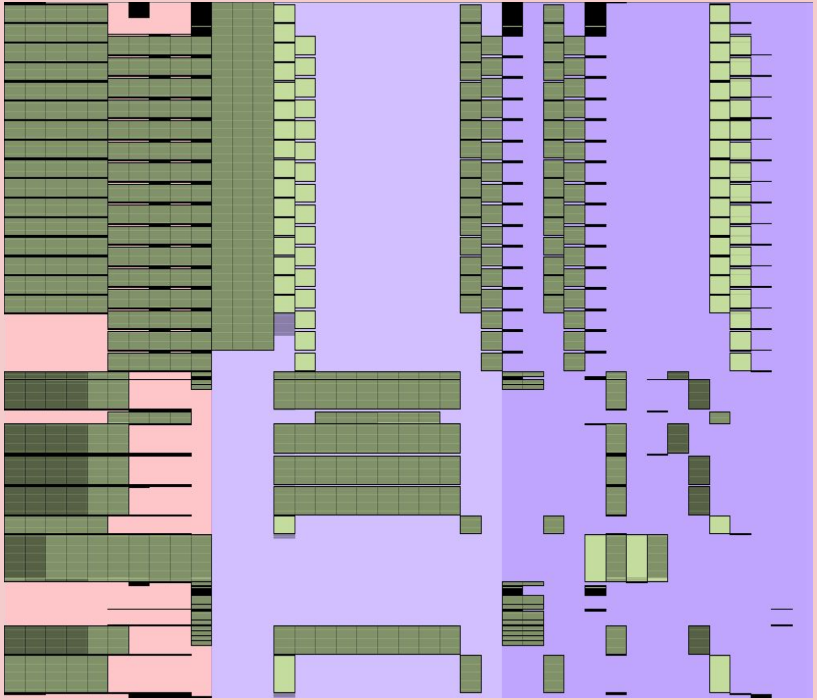
layouter

single-pass layouter (2^{12} rows)

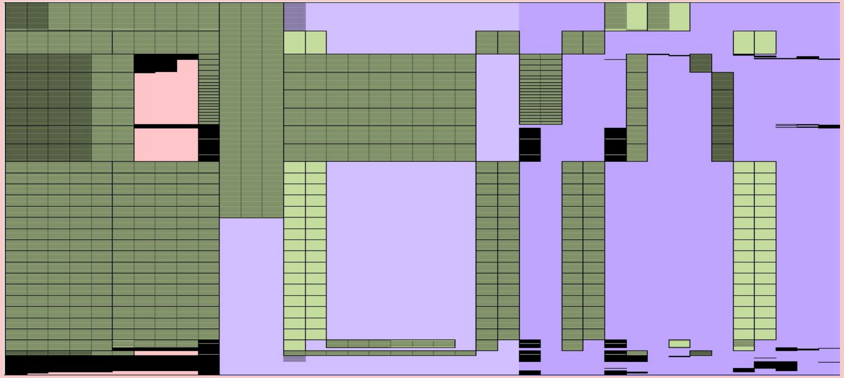


layouter

single-pass layouter (2^{12} rows)

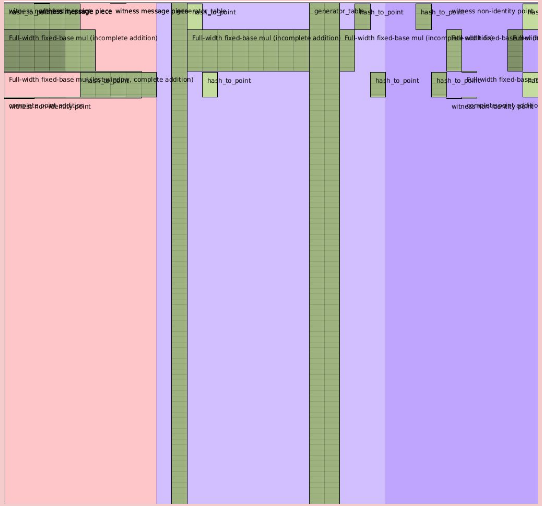


dual-pass layouter (2^{11} rows)



halo2_gadgets

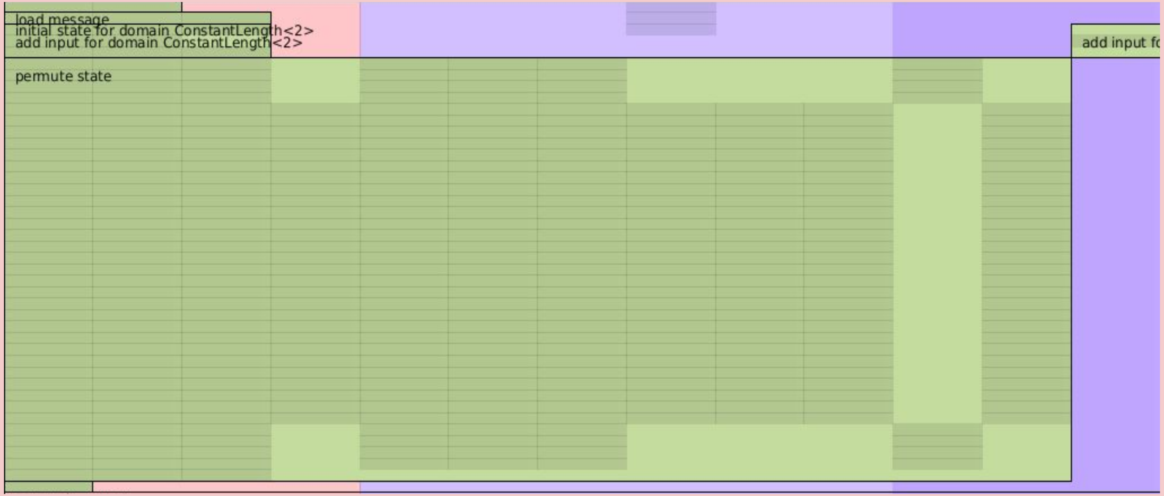
Sinsemilla



2^{10} lookup table of indexed Sinsemilla generators:

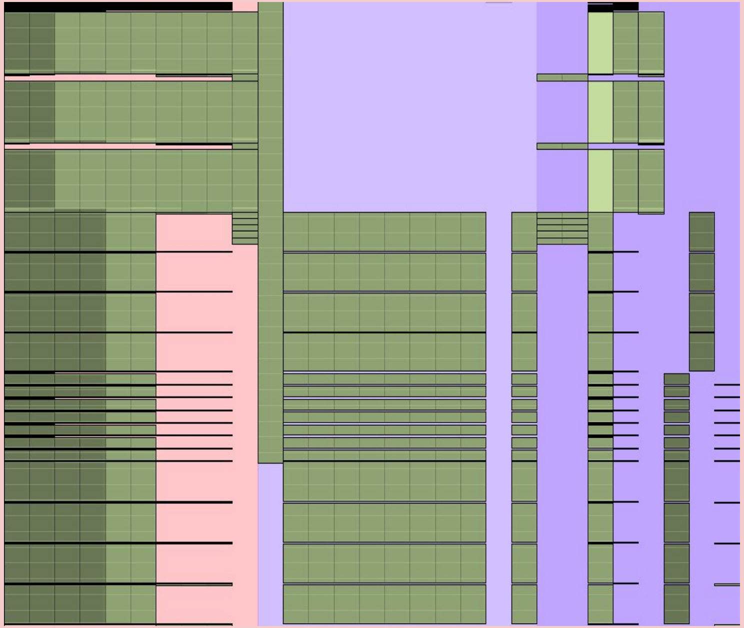
$(i, P[i]_x, P[i]_y)$

Poseidon

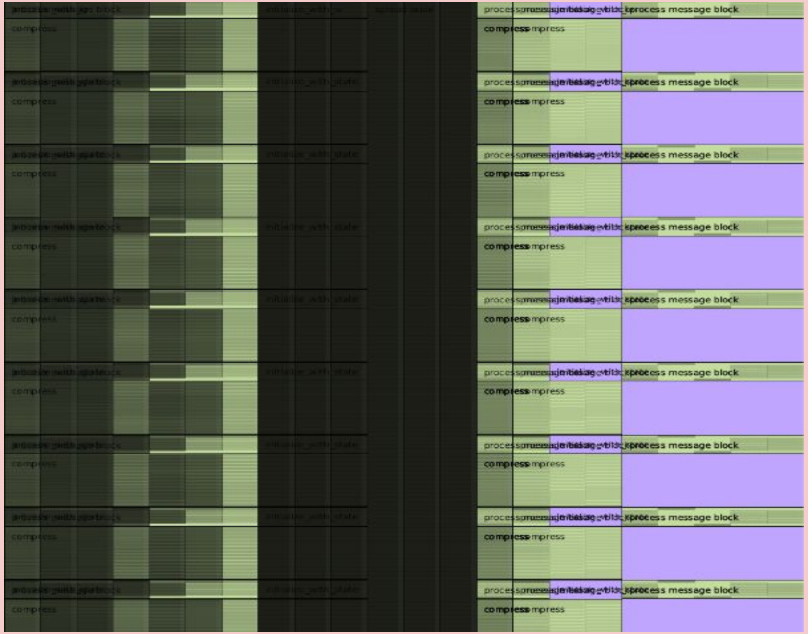


halo2_gadgets

ECC



SHA256 (experimental)



open problems / wishlist

- DSL / intermediate representation
  - add an API to construct a Halo 2 circuit from a set of constraints (halo2#550)
  - improve connection between gate configuration and assignment (halo2#365)
- multi-phase prover (halo2#593)
- dynamic lookup tables (halo2#534)

- what other features would you like to see?

arithmetisation (cont.)

1. **arithmetise** statement using UltraPLONK circuit

arithmetisation (cont.)

1. arithmetise statement using UltraPLONK circuit
2. **commit to polynomials** encoding the main components of the circuit;

$$\mathrm{Commit}(p) = \sum^{n} [p[i]]G_i,$$

where i = [0..=n], and G_i's are commitments to the Lagrange basis polynomials

$a_i(X)$

advice polys

$f_i(X)$

fixed polys

arithmetisation (cont.)

1. arithmetise statement using UltraPLONK circuit
2. commit to polynomials encoding the main components of the circuit;
3. construct **vanishing argument** to constrain all circuit relations to zero;

$$h(X) = \frac{\mathrm{gate}_0(X) + y \cdot \mathrm{gate}_1(X) + \ldots + y^n \cdot \mathrm{gate}_n(X)}{t(X)}$$

where $t(X)$ is the vanishing polynomial on the domain $\{\omega^0, \ldots, \omega^{n-1}\}$; in other words, $t(X) = X^n - 1$

arithmetisation (cont.)

1. arithmetise statement using UltraPLONK circuit
2. commit to polynomials encoding the main components of the circuit;
3. construct vanishing argument to constrain all circuit relations to zero;
4. **evaluate** the above polynomials at all necessary points;

$a_0(x),\ a_0(\omega x),\ a_1(\omega x), \dots \qquad f_0(x),\ f_1(\omega x),\ f_1(\omega^{-1} x), \dots$

$$h(x) = \frac{gate_0(x) + y \cdot gate_1(x) + \dots + y^n \cdot gate_n(x)}{t(x)}$$

arithmetisation (cont.)

1. arithmetise statement using UltraPLONK circuit
2. commit to polynomials encoding the main components of the circuit;
3. construct vanishing argument to constrain all circuit relations to zero;
4. evaluate the above polynomials at all necessary points;
5. construct the **multipoint opening argument** to check that all evaluations are consistent with their respective commitments;

multipoint opening argument

a. group commitments by the **sets of points** at which they were queried:

$\{x\}$	$\{x, \omega x\}$
A_0	A_2
A_1	A_3

b. construct polynomials to accumulate polynomials at each point set; sample x_1 to keep them linearly independent:

$$q_0(X) = A_0(X) + x_1 \cdot A_1(X)$$
$$q_1(X) = A_2(X) + x_1 \cdot A_3(X)$$

c. evaluate the q_i's at their respective points: $q_0(x)$, $q_1(x)$, $q_1(\omega x)$

multipoint opening argument

d. at each query point, interpolate the relevant q_i evaluations:

$$r_0(X) \text{ s.t. } r_0(x) = A_0(x) + x_1 \cdot A_1(x); \quad r_1(x) = A_2(x) + x_1 \cdot A_3(x)$$

$$r_1(X) \text{ s.t. } r_1(\omega x) = A_2(\omega x) + x_1 \cdot A_3(\omega x)$$

e. construct polynomials to check the correctness of q_i polynomials

$$f_0(X) = \frac{q_0(X) - r_0(X)}{X - x}$$

$$f_1(X) = \frac{q_1(X) - r_1(X)}{(X - x)(X - \omega x)}$$

f. construct $f(X) = f_0(X) + x_2 \cdot f_1(X)$, with random x_2 to keep f_i polynomials linearly independent

multipoint opening argument

g. construct

$$\text{final\_poly}(X) = f(X) + x_4 \cdot q_0(X) + x_4^2 \cdot q_1(X),$$

with random x_4 challenge to keep polynomials linearly independent.

this checks that all evaluations are consistent with their respective commitments.

added in "Math Building Blocks" (15 Jun)

polynomial commitment scheme

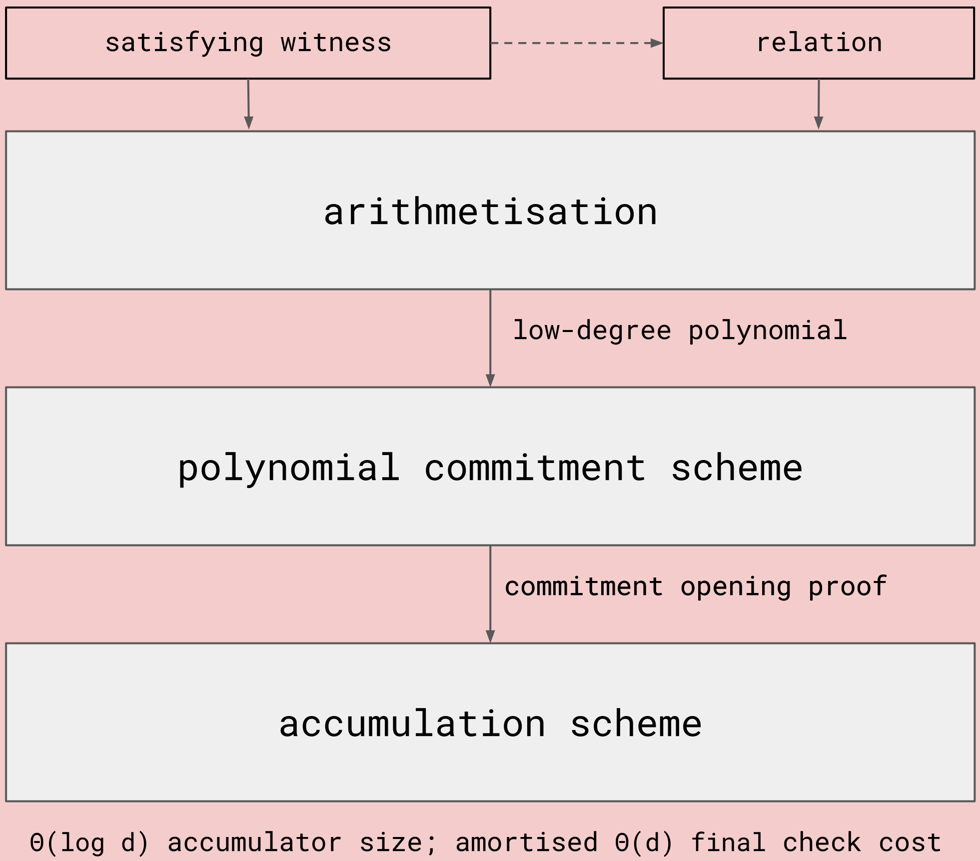
Setup(): generates a setup pk

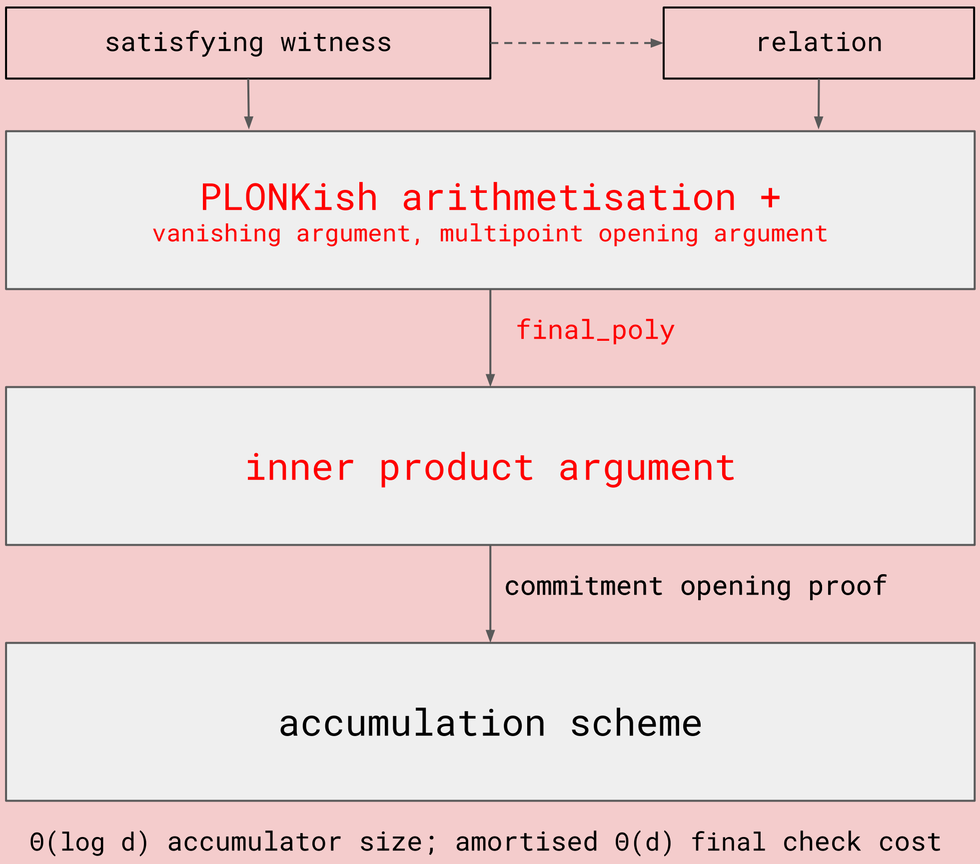
Commit(pk, P): creates a commitment c to P

VerifyPoly(pk, c, P): checks c is a valid commitment to P

Open(pk, P, x): generates an opening proof π

VerifyOpen(pk, c, x, y, π): checks if $y = P(x)$ using π


satisfying witness
relation
arithmetisation
low-degree polynomial
polynomial commitment scheme
commitment opening proof
accumulation scheme
Θ(log d) accumulator size; amortised Θ(d) final check cost


satisfying witness
relation
PLONKish arithmetisation +
vanishing argument, multipoint opening argument
final_poly
inner product argument
commitment opening proof
accumulation scheme
Θ(log d) accumulator size; amortised Θ(d) final check cost

added in "Math Building Blocks" (15 Jun)

inner product argument

Setup$(1^\lambda, d)$ produces a crs $\sigma = (\mathbb{G}, \mathbb{F}_p, \mathbf{G}, H)$ for group $\mathbb{G}$ of prime order p, with random $\mathbf{G} \in \mathbb{G}^d$ and $H \in \mathbb{G}$.

NOT trusted!

Commit$(\sigma, p(X); r)$: $P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$, where $\mathbf{a}_i \in \mathbb{F}$ is the coeff for the i^{th} deg term in $p(X)$, $i \in [0, d)$

VerifyPoly(pk, P, p): checks that $P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$

Open(pk, p, x): generates an opening proof π for $y = p(x)$
$= \langle \mathbf{a}, (1, x, \ldots, x^{d-1}) \rangle$

VerifyOpen$(\mathrm{pk}, C, x, y, \pi)$: checks $y = \langle \mathbf{a}, (1, x, \ldots, x^{d-1}) \rangle$ using π

added in "Math Building Blocks" (15 Jun)

inner product argument

Setup$(1^\lambda, d)$ produces a crs $\sigma = (\mathbb{G}, \mathbb{F}_p, \mathbf{G}, H)$ for group $\mathbb{G}$ of prime order p, with random $\mathbf{G} \in \mathbb{G}^d$ and $H \in \mathbb{G}$.

Commit$(\sigma, p(X); r)$: $P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$, where $\mathbf{a}_i \in \mathbb{F}$ is the coeff for the i^{th} deg term in $p(X)$, $i \in [0, d)$

VerifyPoly(pk, P, p): checks that $P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$

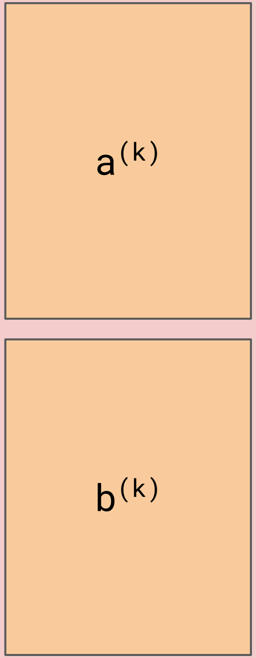
naive opening proof: send $\mathbf{a}$ ($O(n)$ communication)
want: π with $O(\log d)$ communication

$$\boxed{\textbf{Open}(\mathrm{pk}, p, x)\text{: generates an opening proof } \pi \text{ for } y = p(x) = \langle \mathbf{a}, (1, x, \ldots, x^{d-1}) \rangle}$$

VerifyOpen$(\mathrm{pk}, C, x, y, \pi)$: checks $y = \langle \mathbf{a}, (1, x, \ldots, x^{d-1}) \rangle$ using π

added in "Math Building Blocks" (15 Jun)

inner product argument
(originally from [Bootle et al, 2016])



we have vectors **a**, **b** each of length $d = 2^k$. we want to prove that a given inner product c and a commitment P are related as:

$$y = \langle \mathbf{a}, \mathbf{b} \rangle$$

$$P = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle$$

where **G**, **H** are length-n vectors of random group elements.

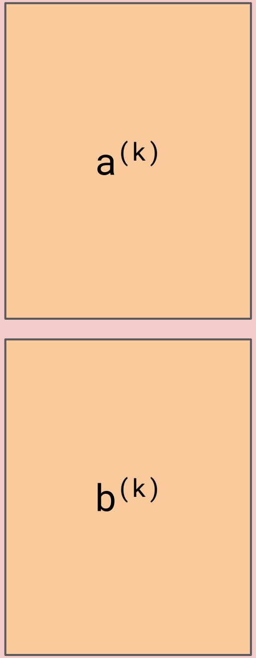
we can combine this into a single commitment P_k using a random group element U:

$$P_k = P + [y]U = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle + [\langle \mathbf{a}, \mathbf{b} \rangle]U$$

the naive solution would be to send the verifier **a**, **b** – but this requires sending $2d$ scalars. instead, the inner product argument uses $O(\log(d)) = O(k)$ communication cost.

added in "Math Building Blocks" (15 Jun)

modified inner product argument



we have vectors **a**, **b** each of length $d = 2^k$.

fix $\mathbf{b} = (1, x, x^2, \ldots, x^{d-1})$ for a chosen evaluation point x, known to both prover and verifier. since **b** is known, no random vector **H** is needed.

we want to prove that v, which is an evaluation of **a** at x, and a commitment P are related as:

$$v = \langle \mathbf{a}, (1, x, x^2, \ldots, x^{d-1}) \rangle$$

$$P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H$$

where **G** is a length-n vector of random group elements.

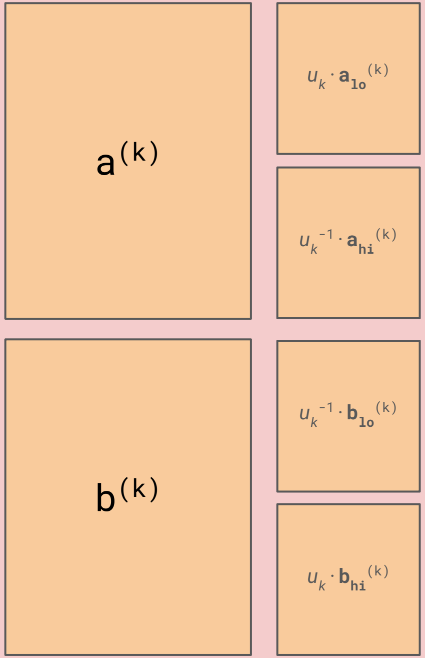
we can combine this into a single commitment P_k using a random group element U:

$$P_k = P + [v]U = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H + [\langle \mathbf{a}, \mathbf{b} \rangle]U$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument

round k



we start at the k^{th} round, where we split **a**, **b**, **G** into **lo** and **hi** halves. we introduce a random challenge u_k and compress the vectors by adding the left and the right halves separated by u_k:

$$\mathbf{a}^{(k-1)} = u_k \cdot \mathbf{a}_{lo}^{(k)} + u_k^{-1} \cdot \mathbf{a}_{hi}^{(k)}$$

$$\mathbf{b}^{(k-1)} = u_k^{-1} \cdot \mathbf{b}_{lo}^{(k)} + u_k \cdot \mathbf{b}_{hi}^{(k)}$$

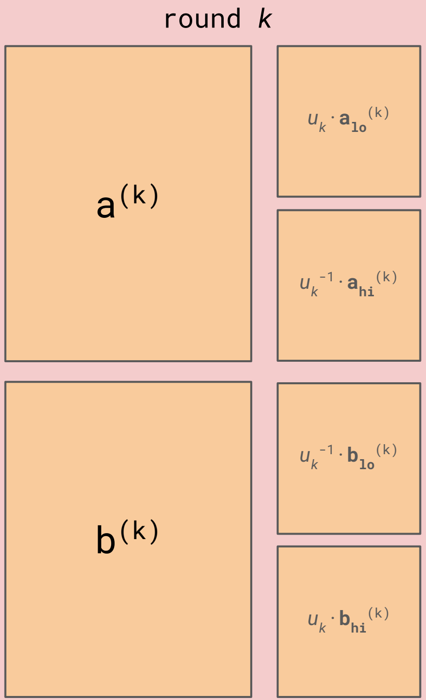
$$\mathbf{G}^{(k-1)} = u_k^{-1} \cdot \mathbf{G}_{lo}^{(k)} + u_k \cdot \mathbf{G}_{hi}^{(k)}$$

now, we can write a commitment P_{k-1} (of the same form as P_k) using the compressed vectors:

$$P_{k-1} = \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle]\ U$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument

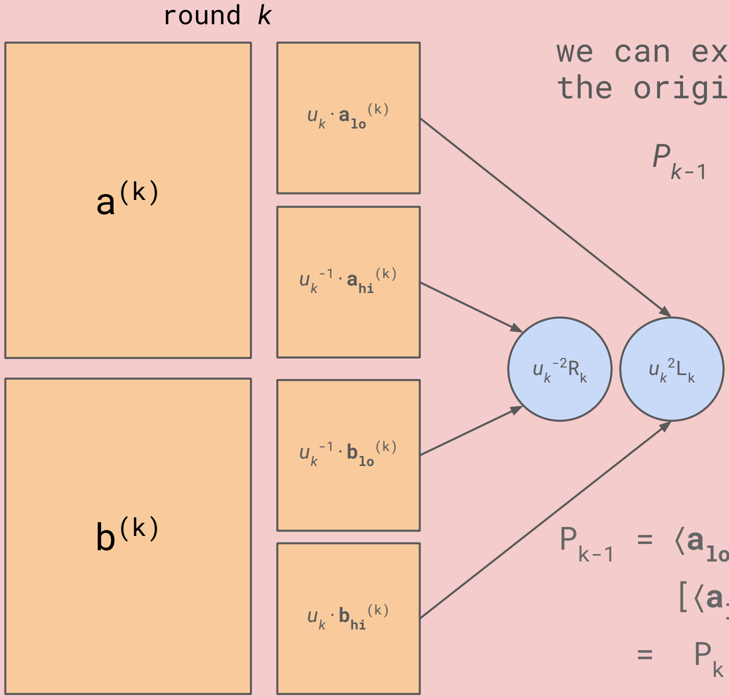


we can examine the compressed P_{k-1} equation in terms of the original vectors:

$$
\begin{aligned}
P_{k-1} &= \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle] U \\
&= \langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, u_k^{-1} \cdot \mathbf{G}_{lo} + u_k \cdot \mathbf{G}_{hi} \rangle + \\
&\quad [\langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, u_k^{-1} \cdot \mathbf{b}_{lo} + u_k \cdot \mathbf{b}_{hi} \rangle] U
\end{aligned}
$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument



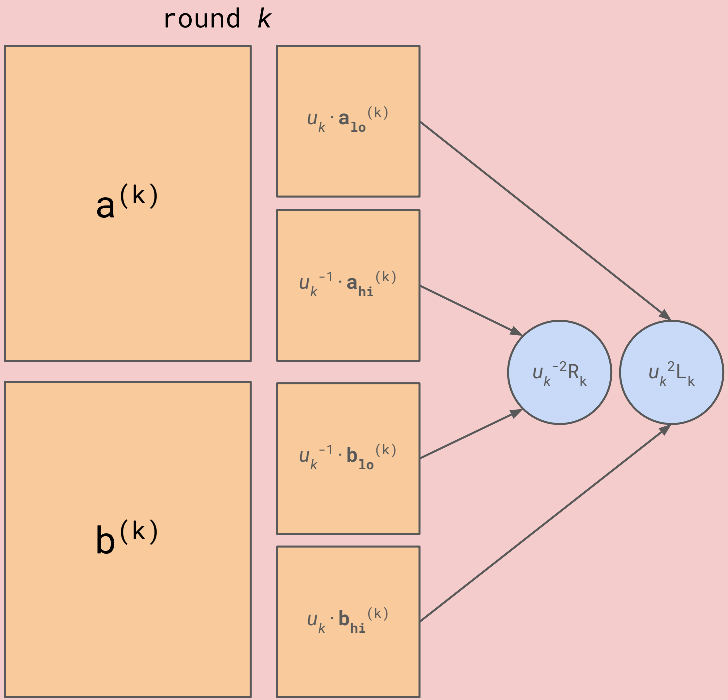
we can examine the compressed P_{k-1} equation in terms of the original vectors:

$$
\begin{aligned}
P_{k-1} &= \langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \rangle + [\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \rangle] U \\
&= \langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, u_k^{-1} \cdot \mathbf{G}_{lo} + u_k \cdot \mathbf{G}_{hi} \rangle + \\
&\quad [\langle u_k \cdot \mathbf{a}_{lo} + u_k^{-1} \cdot \mathbf{a}_{hi}, u_k^{-1} \cdot \mathbf{b}_{lo} + u_k \cdot \mathbf{b}_{hi} \rangle] U
\end{aligned}
$$

$$
\begin{aligned}
P_{k-1} &= \langle \mathbf{a}_{lo}, \mathbf{G}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{G}_{hi} \rangle + u_k^2 \langle \mathbf{a}_{lo}, \mathbf{G}_{hi} \rangle + u_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{G}_{lo} \rangle + \\
&\quad [\langle \mathbf{a}_{lo}, \mathbf{b}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{b}_{hi} \rangle)] U + [u_k^2 \langle \mathbf{a}_{lo}, \mathbf{b}_{hi} \rangle + u_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{b}_{lo} \rangle] U \\
&= P_k + [u_k^2] L_k + [u_k^{-2}] R_k
\end{aligned}
$$

added in "Math Building Blocks" (15 Jun)

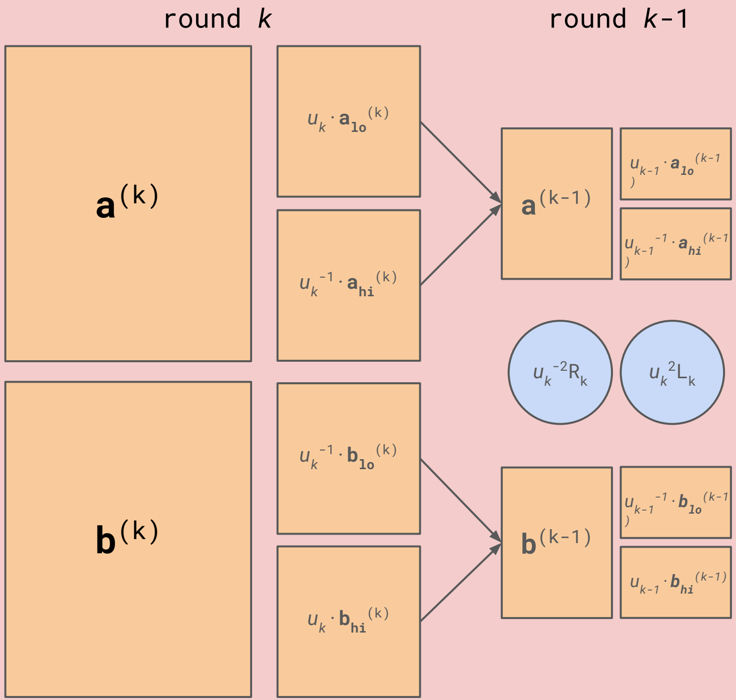
modified inner product argument



$$P_{k-1} = P_k + [u_k^2]L_k + [u_k^{-2}]R_k$$

P_{k-1} is the sum of P_k and the cross-terms L_k, R_k (with coefficients from the round challenge u_k)

modified inner product argument

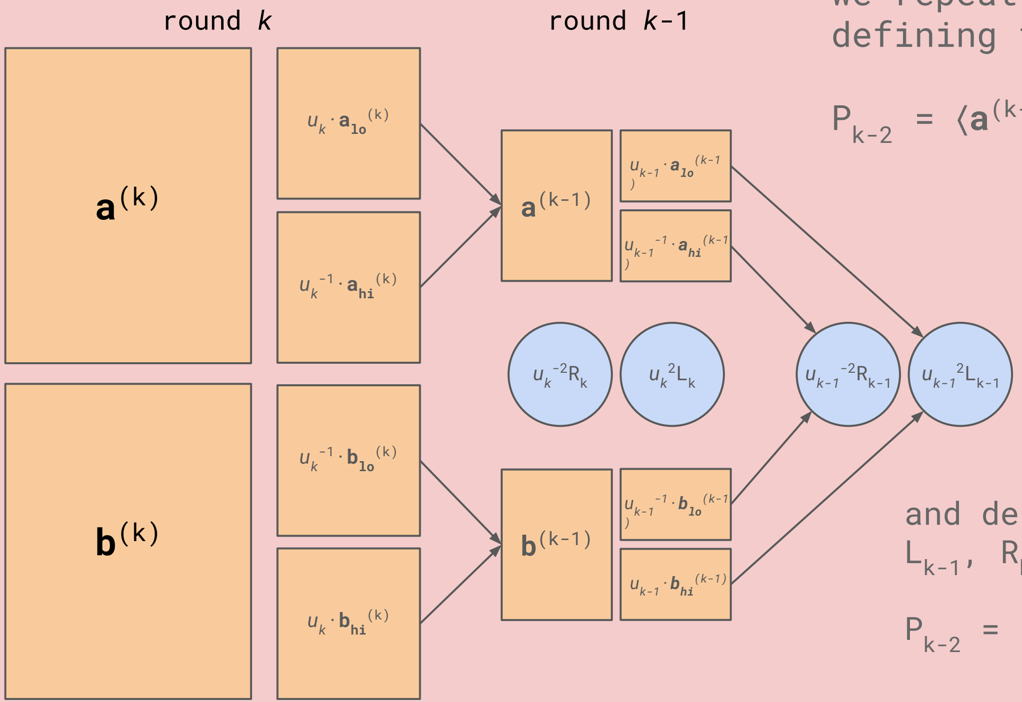


we repeat this in the k-1st round, defining the new compressed eqn

$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] U$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument



we repeat this in the k-1st round, defining the new compressed eqn

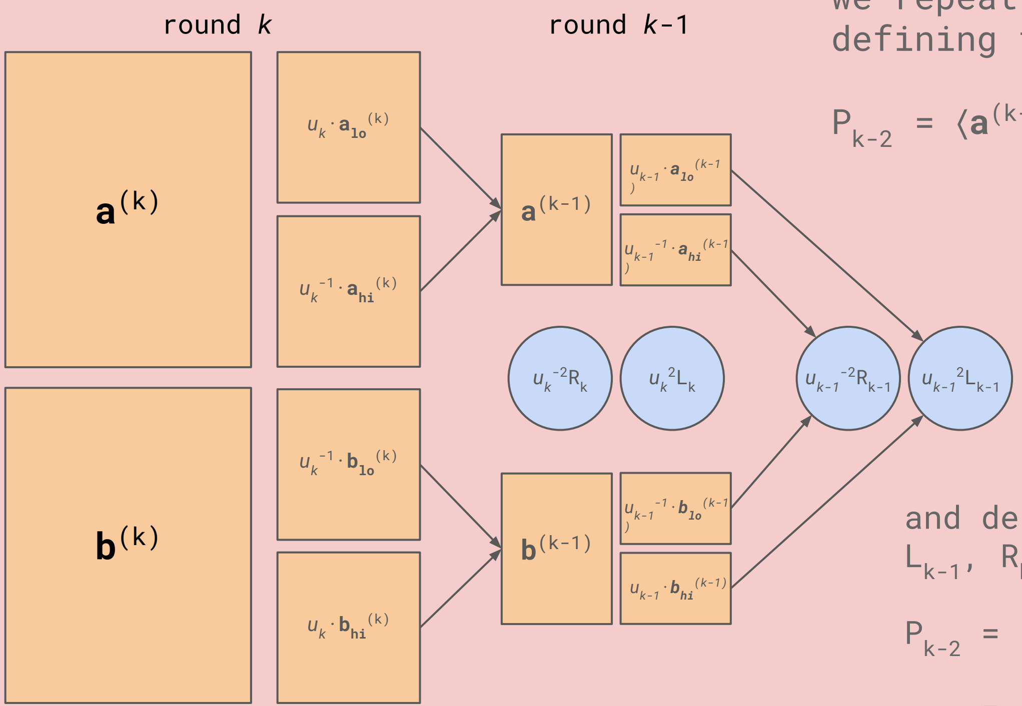
$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] U$$

and deriving new cross-terms L_{k-1}, R_{k-1} such that

$$P_{k-2} = P_{k-1} + [u_{k-1}^{2}]L_{k-1} + [u_{k-1}^{-2}]R_{k-1}$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument



we repeat this in the k-1st round, defining the new compressed eqn

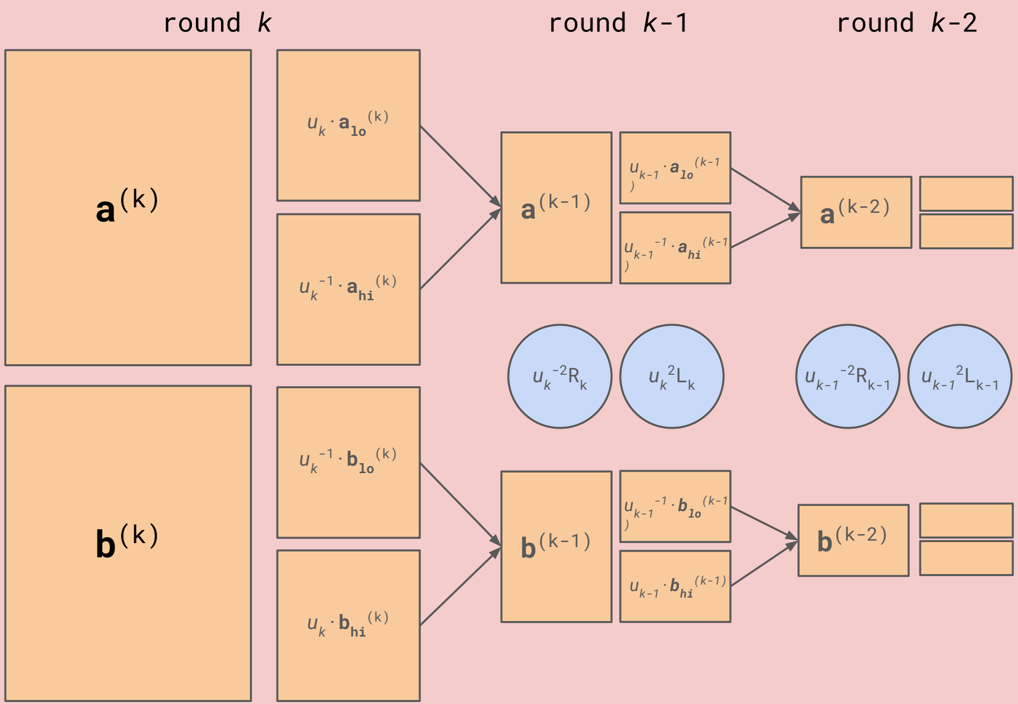
$$P_{k-2} = \langle \mathbf{a}^{(k-2)}, \mathbf{G}^{(k-2)} \rangle + [\langle \mathbf{a}^{(k-2)}, \mathbf{b}^{(k-2)} \rangle] U$$

and deriving new cross-terms L_{k-1}, R_{k-1} such that

$$\begin{aligned} P_{k-2} &= P_{k-1} + [u_{k-1}^{2}]L_{k-1} + [u_{k-1}^{-2}]R_{k-1} \\ &= P_k + [u_k^{2}]L_k + [u_k^{-2}]R_k \\ &\quad + [u_{k-1}^{2}]L_{k-1} + [u_{k-1}^{-2}]R_{k-1} \end{aligned}$$

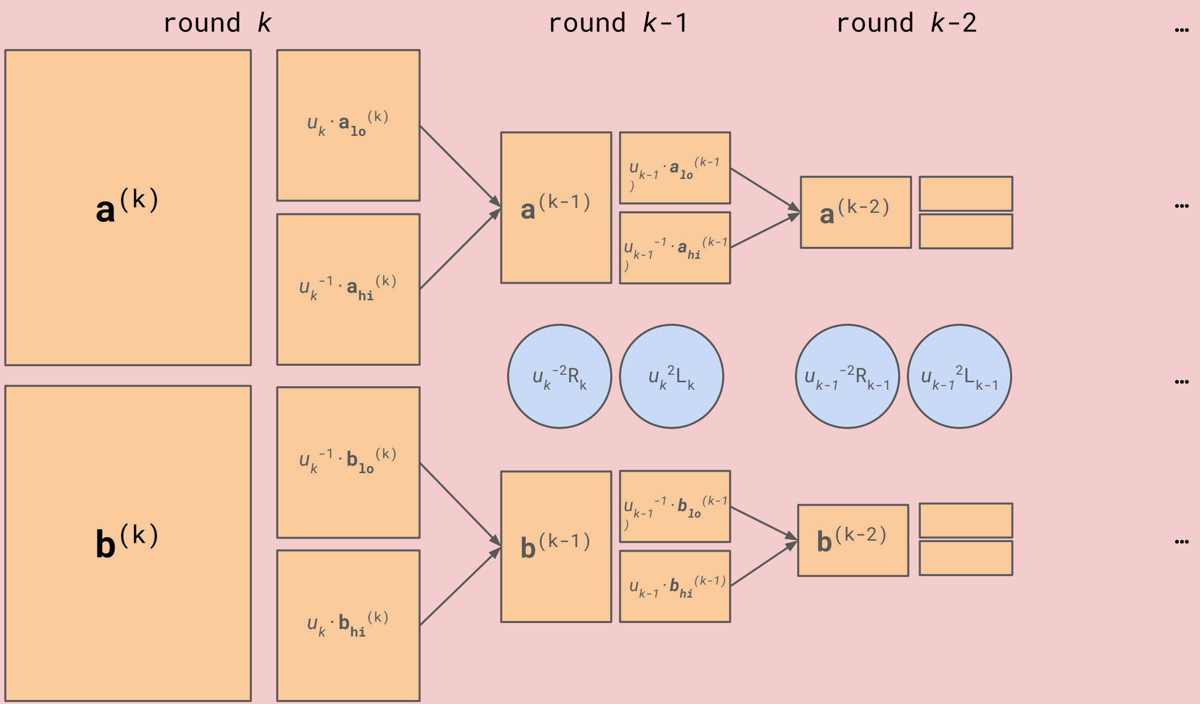
added in "Math Building Blocks" (15 Jun)

modified inner product argument



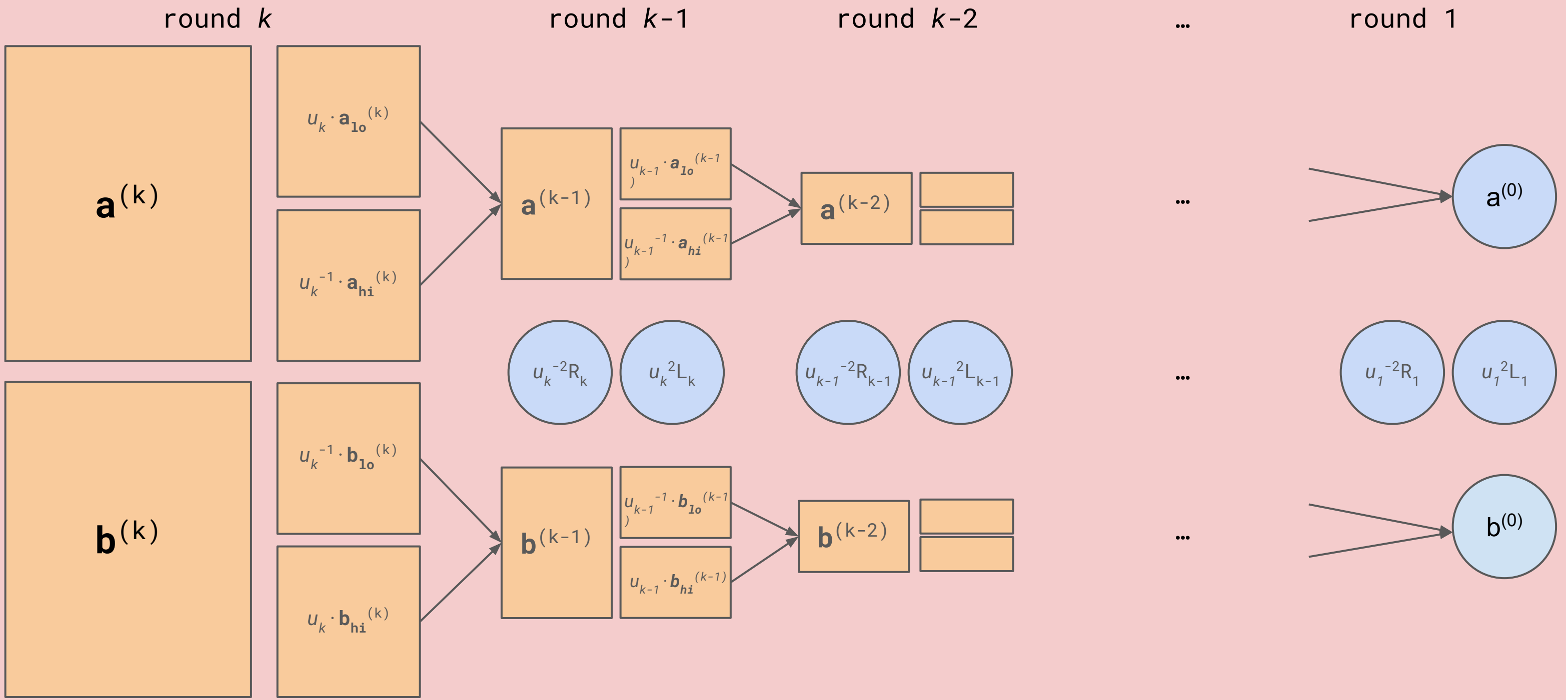
added in "Math Building Blocks" (15 Jun)

modified inner product argument



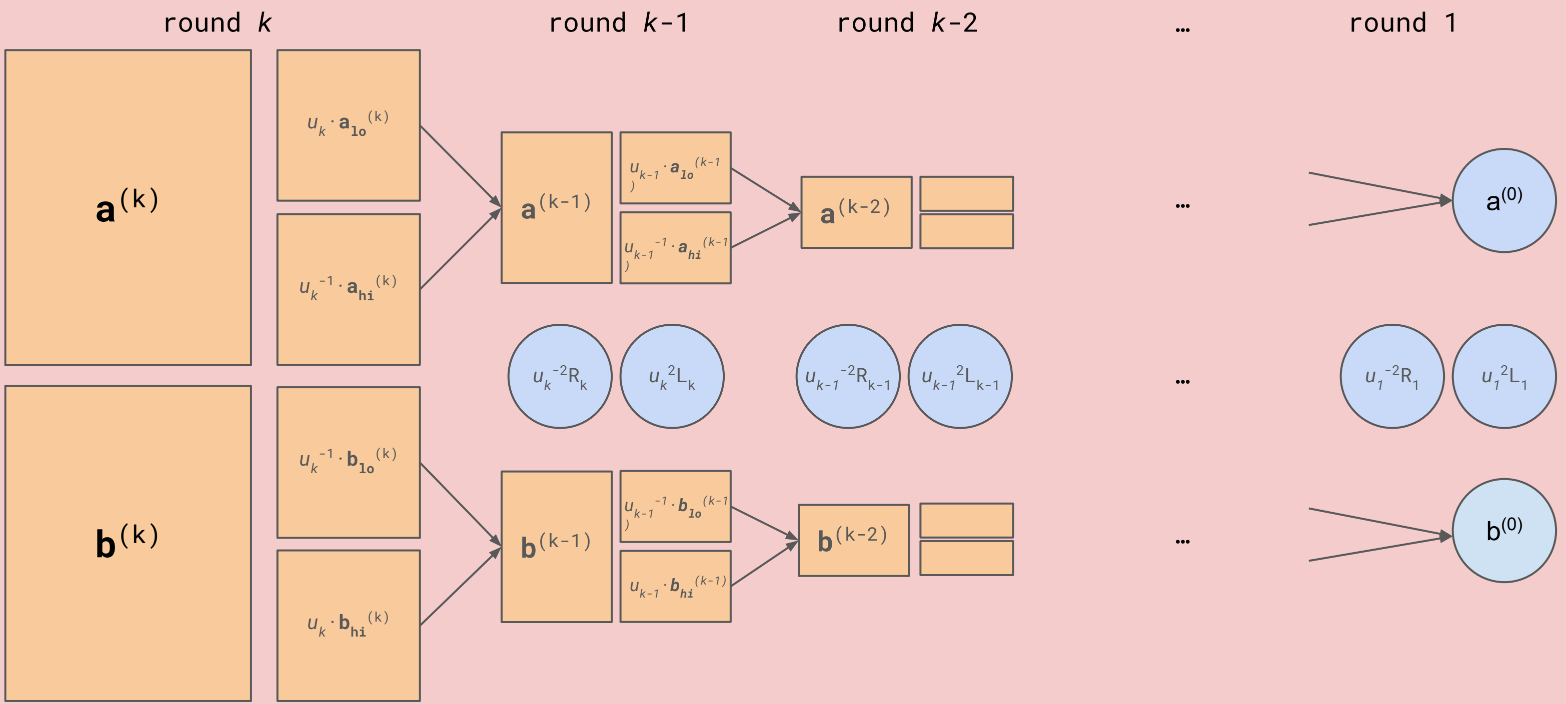
added in "Math Building Blocks" (15 Jun)

modified inner product argument



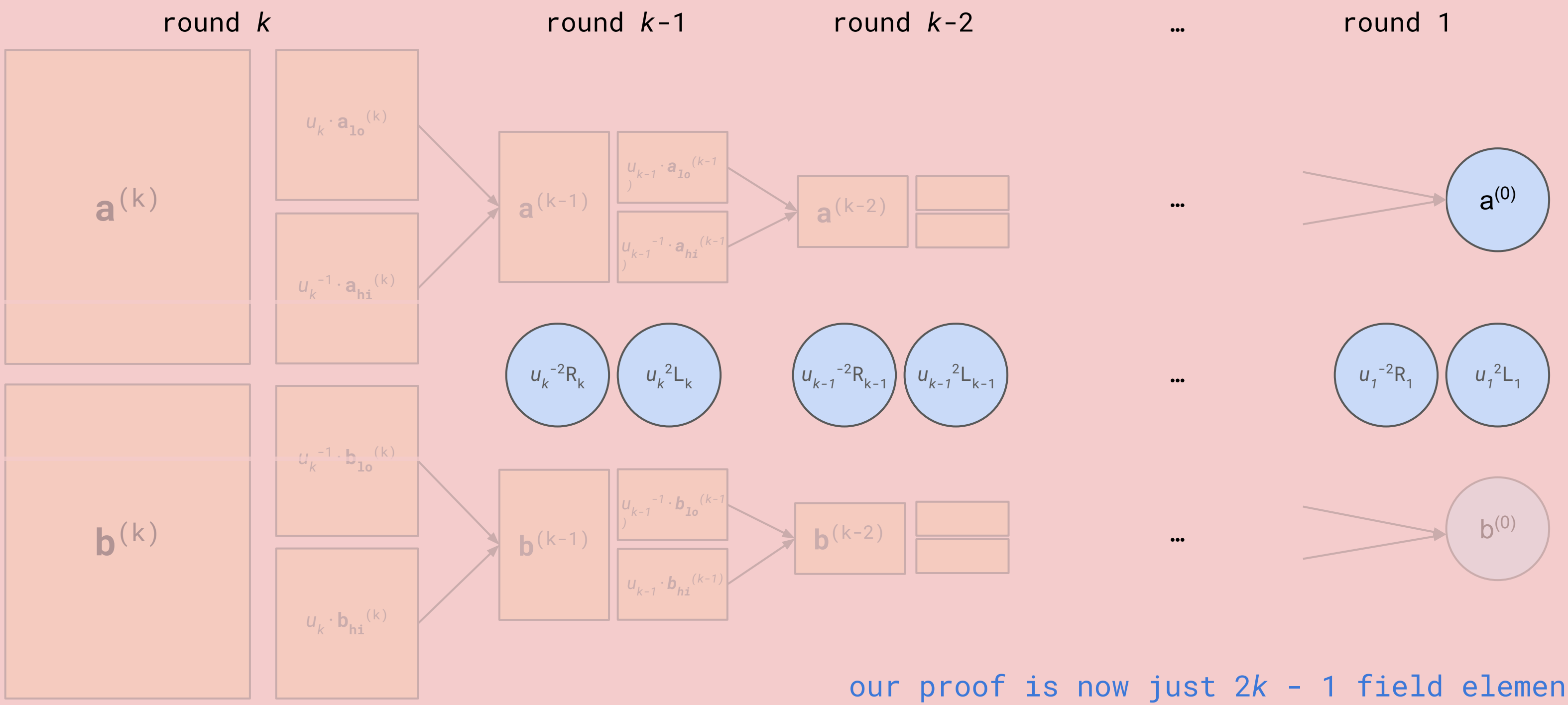
added in "Math Building Blocks" (15 Jun)

modified inner product argument



added in "Math Building Blocks" (15 Jun)

modified inner product argument



our proof is now just 2k - 1 field elements!

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\ U$$

but $P_0 = \sum^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \sum^k([u_j^{\,-2}]\ R_j)$

verifier checks this equivalence

$$[\mathrm{a}]\mathrm{G} + [\mathrm{a} \cdot \mathrm{b}]\ U == \sum^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \sum^k([u_j^{\,-2}]\ R_j)$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\ U$$

but $P_0 = \sum^k([u_j{}^2]\ L_j) + \mathrm{P}_k + \sum^k([u_j{}^{-2}]\ R_j)$

verifier checks this equivalence

from prover

$$[\boxed{a}]G + [\boxed{a}\cdot b]\ U == \sum^k([u_j{}^2]\ \boxed{L_j}) + \boxed{\mathrm{P}_k} + \sum^k([u_j{}^{-2}]\ \boxed{R_j})$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\, U$$

but $P_0 = \sum^k([u_j^{2}]\, L_j) + P_k + \sum^k([u_j^{-2}]\, R_j)$

verifier checks this equivalence

from prover

$$[a]G + [a \cdot b]\, U == \sum^k([u_j^{2}]\, L_j) + P_k + \sum^k([u_j^{-2}]\, R_j)$$

calculated by verifier

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\ U$$

but $P_0 = \Sigma^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \Sigma^k([u_j^{\,-2}]\ R_j)$

where $\mathbf{s} = \begin{pmatrix} u_1^{-1}\, u_2^{-1}\, u_3^{-1} \ldots u_k^{-1}, \\ u_1\ u_2^{-1}\, u_3 \ldots u_k^{-1}, \\ u_1^{-1}\, u_2\ u_3^{-1} \ldots u_k, \\ \ldots \\ u_1\ u_2\ u_3 \ldots u_k \end{pmatrix}$

verifier checks this equivalence

$$[\mathrm{a}]\underbrace{\boxed{\mathrm{G}}}_{\langle \mathbf{G}, \mathbf{s}\rangle} + [\mathrm{a} \cdot \underbrace{\boxed{\mathrm{b}}}_{\langle \mathbf{b}, \mathbf{s}\rangle = g(x,\ u_1,\ \ldots,\ u_k)}]\ U == \Sigma^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \Sigma^k([u_j^{\,-2}]\ R_j)$$

where $g(X,\ u_1,\ \ldots,\ u_k) = \Pi^k(u_i + u_i^{-1}X^2)$

(can compute in $O(\log(d))$ steps)

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\ U$$

but $P_0 = \sum^k([u_j^2]\ L_j) + P_k + \sum^k([u_j^{-2}]\ R_j)$

where $\mathbf{s} = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \dots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3 & \dots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \dots & u_k, \\ & & \dots & & \\ u_1 & u_2 & u_3 & \dots & u_k \end{pmatrix}$

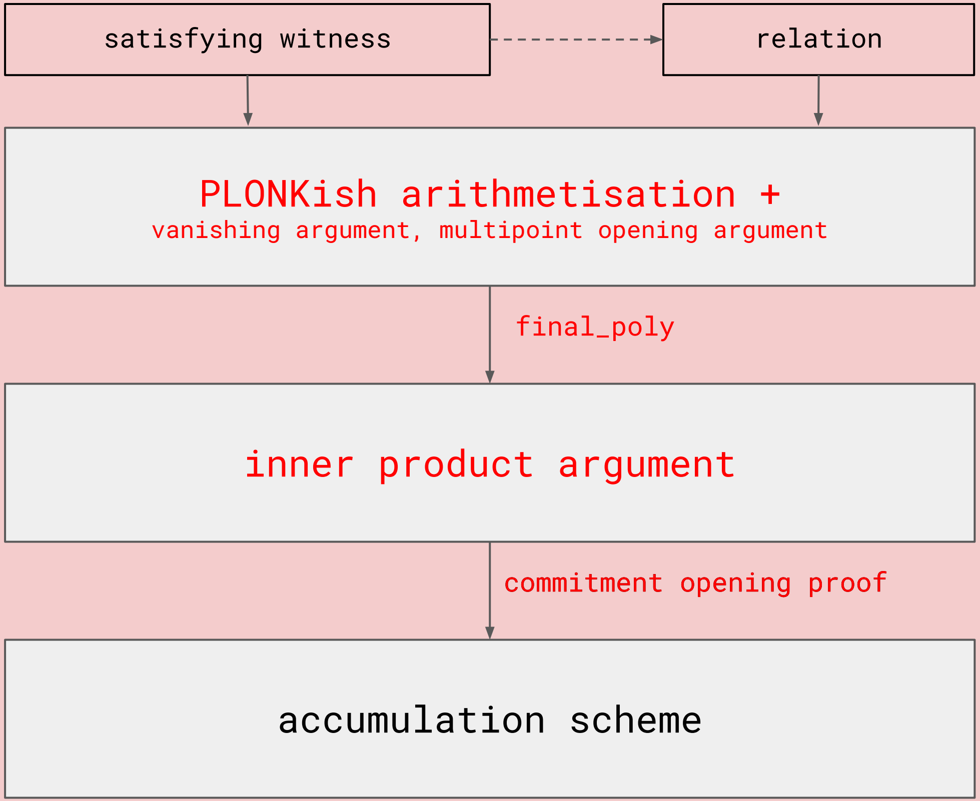
verifier checks this equivalence

$$[a]\boxed{G} + [a \cdot \boxed{b}]\ U == \sum^k([u_j^2]\ L_j) + P_k + \sum^k([u_j^{-2}]\ R_j)$$

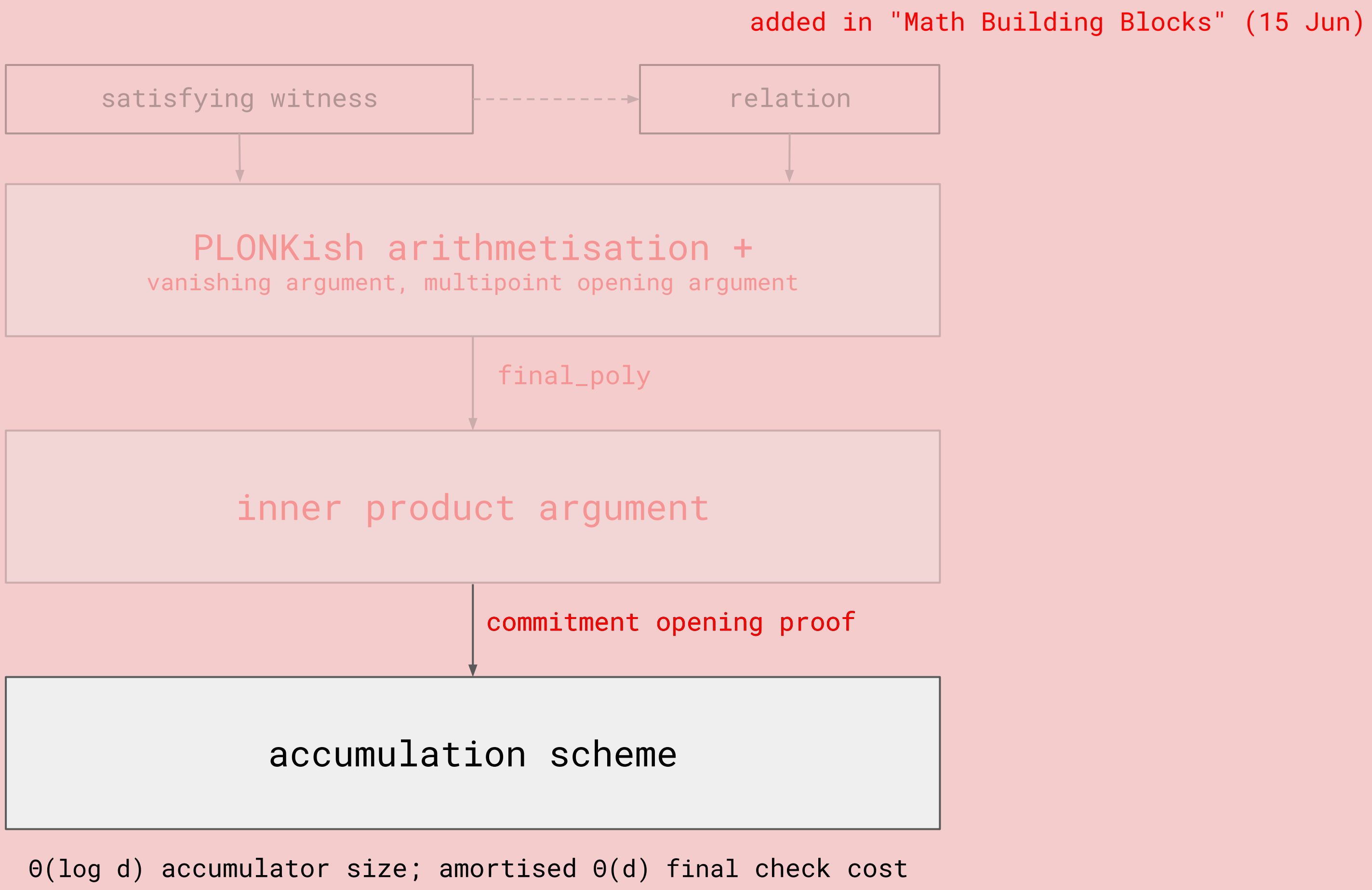
$\langle \mathbf{G}, \mathbf{s} \rangle$ needs a **linear-time multi-scalar multiplication**. this is a problem for recursive proof composition: a linear-size verification circuit will yield fixed-depth recursion at best.

instead, we use an **accumulation scheme** to amortise this linear cost across a batch of proof instances.

added in "Math Building Blocks" (15 Jun)



$\Theta(\log d)$ accumulator size; amortised $\Theta(d)$ final check cost



added in "Math Building Blocks" (15 Jun)
satisfying witness
relation
PLONKish arithmetisation +
vanishing argument, multipoint opening argument
final_poly
inner product argument
commitment opening proof
accumulation scheme
Θ(log d) accumulator size; amortised Θ(d) final check cost

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\ U$$

but $P_0 = \sum^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \sum^k([u_j^{\,-2}]\ R_j)$

where $\mathbf{s} = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \dots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3 & \dots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \dots & u_k\ , \\ & & \dots & & \\ u_1 & u_2 & u_3 & \dots & u_k \end{pmatrix}$

verifier checks this equivalence

$$g(X, u_1, \dots, u_k) = \prod^k (u_i + u_i^{-1} X^2\ )$$

$$[\mathrm{a}]\boxed{\mathrm{G}} + [\mathrm{a} \cdot \boxed{\mathrm{b}}]\ U == \sum^k([u_j^{\,2}]\ L_j) + \mathrm{P}_k + \sum^k([u_j^{\,-2}]\ R_j)$$

$\langle \mathbf{G}, \mathbf{s} \rangle$ can itself be rewritten as a polynomial commitment:

$$G = \mathbf{G}^{(0)} = \mathrm{Commit}(\sigma,\ g(X, u_1, \dots, u_k))$$

added in "Math Building Blocks" (15 Jun)

modified inner product argument

$$P_0 = [\mathbf{a}^{(0)}]\mathbf{G}^{(0)} + [\mathbf{a}^{(0)} \cdot \mathbf{b}^{(0)}]\, U$$

but $P_0 = \sum^k([u_j^2]\, L_j) + P_k + \sum^k([u_j^{-2}]\, R_j)$

where $\mathbf{s} = \begin{pmatrix} u_1^{-1} & u_2^{-1} & u_3^{-1} & \dots & u_k^{-1}, \\ u_1 & u_2^{-1} & u_3 & \dots & u_k^{-1}, \\ u_1^{-1} & u_2 & u_3^{-1} & \dots & u_k, \\ & & \dots & & \\ u_1 & u_2 & u_3 & \dots & u_k \end{pmatrix}$

verifier checks this equivalence

$$g(X, u_1, \dots, u_k) = \prod^k (u_i + u_i^{-1} X^{2^{i}})$$

$$[a]\boxed{G} + [a \cdot \boxed{b}]\, U == \sum^k([u_j^2]\, L_j) + P_k + \sum^k([u_j^{-2}]\, R_j)$$

$\langle \mathbf{G}, \mathbf{s} \rangle$ can itself be rewritten as a polynomial commitment:

$$G = \mathbf{G}^{(0)} = \mathrm{Commit}(\sigma,\ g(X, u_1, \dots, u_k))$$

which means we can invoke an inner product argument for G.

added in "Math Building Blocks" (15 Jun)

accumulation scheme

in an accumulation scheme, it is expensive to “decide” validity of an instance, but cheap to combine two (or more) instances.

accumulator	accumulation step	decider
P, x, v, $\boxed{a, G, L, R \; \{u_1, u_2, \dots, u_k\}}$	polynomial commitment opening of G ($O(\log(d)$ field operations) the **old accumulator** and **new instance** (same form as the accumulator) are accumulated into a new accumulator	check (in $O(d)$ time) that $G = \mathrm{Commit}(g(X, u_1, u_2, \dots, u_k))$ $= \langle \mathbf{s}, \mathbf{G} \rangle$

π evaluation proof for the claim that $P(x) = v$

added in "Math Building Blocks" (15 Jun)

accumulation scheme

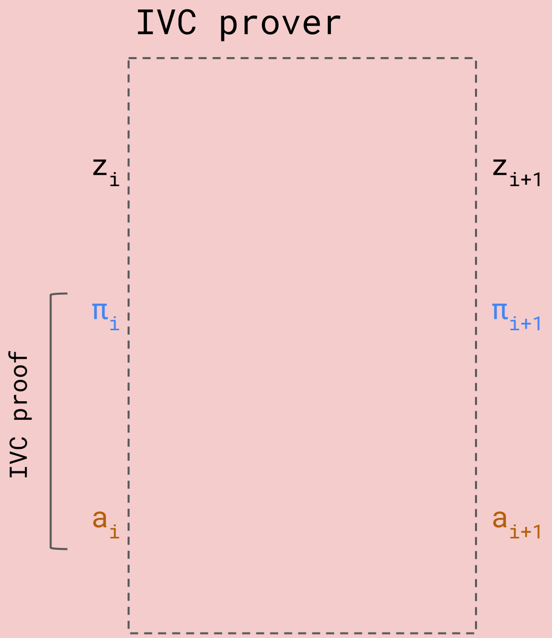
in an accumulation scheme, it is expensive to "decide" validity of an instance, but cheap to combine two (or more) instances.

accumulator	accumulation step	decider
P, x, v, $\boxed{a,\ G,\ L,\ R \ \{u_1, u_2, \ldots, u_k\}}$ π evaluation proof for the claim that $P(x) = v$	polynomial commitment opening of G ($O(\log(d)$ field operations) the **old accumulator** and **new instance** (same form as the accumulator) are accumulated into a new accumulator	check (in $O(d)$ time) that $G = \mathrm{Commit}(g(X, u_1, u_2, \ldots, u_k))$ $= \langle \mathbf{s}, \mathbf{G} \rangle$

instead of trying to compute $G = \langle \mathbf{s},\ \mathbf{G} \rangle$ in the circuit, the verifier instead asks the prover to supply the purported G as part of their witness.

the recursive verifier then checks (in $O(\log(d)$ field operations) that G opens at some random point to the expected value for the given challenges $\{u_1, u_2, \ldots, u_k\}$.

accumulation scheme



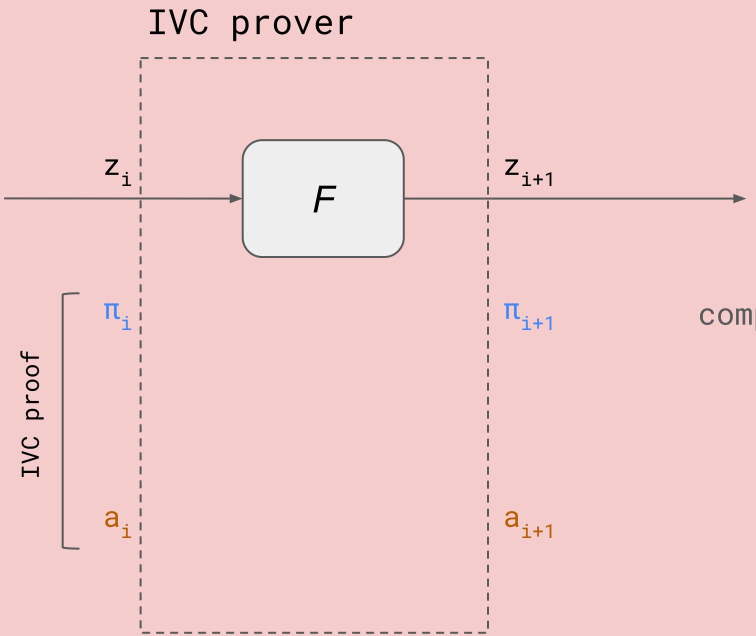
IVC proof consists of:

- SNARK proof π_i,
- accumulator a_i

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

accumulation scheme

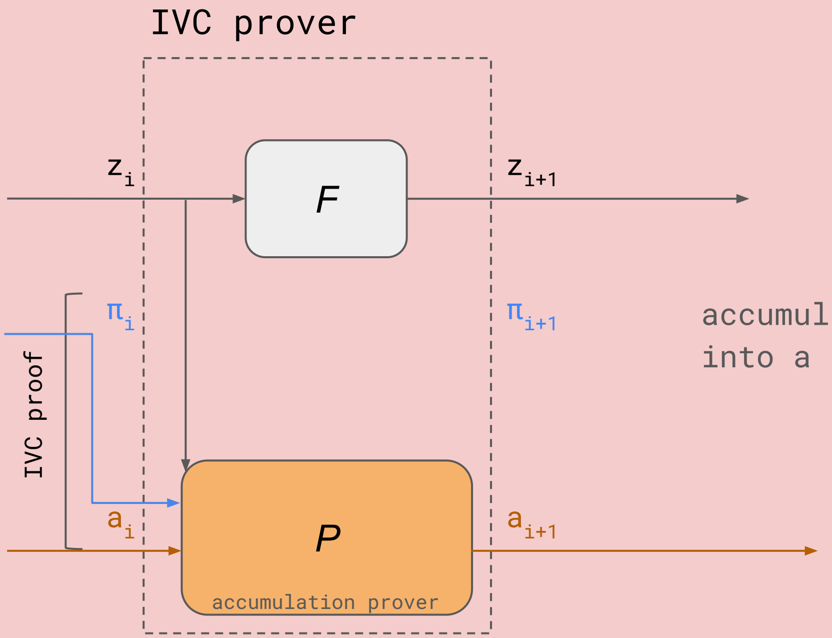


compute state transition function

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

accumulation scheme

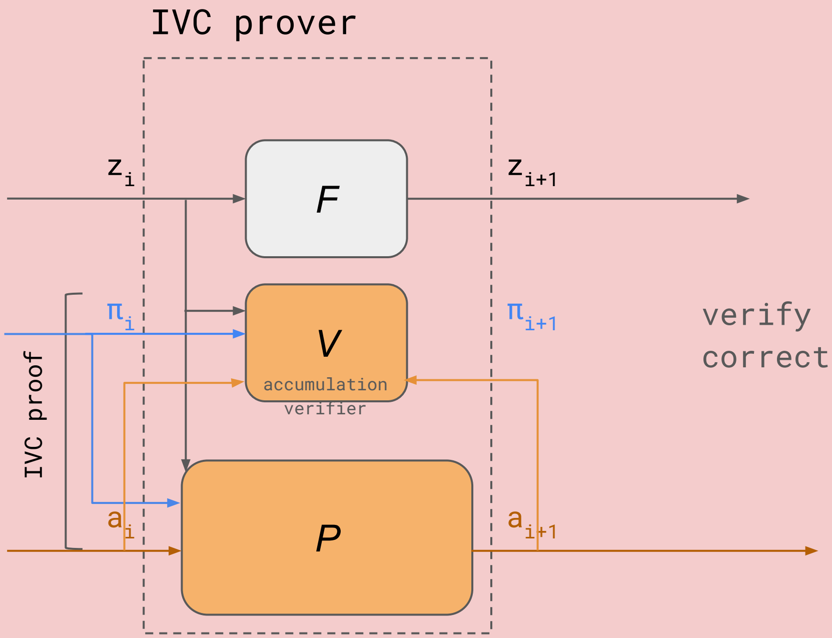


accumulate previous inputs (z_i, π_i, a_i)
into a new accumulator a_{i+1}

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

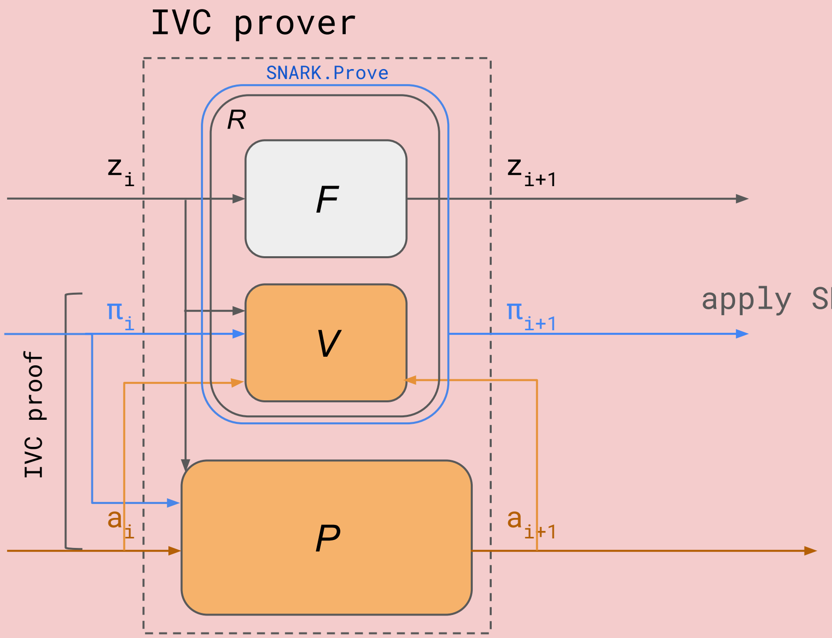
accumulation scheme



verify that new accumulator was updated correctly

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

accumulation scheme

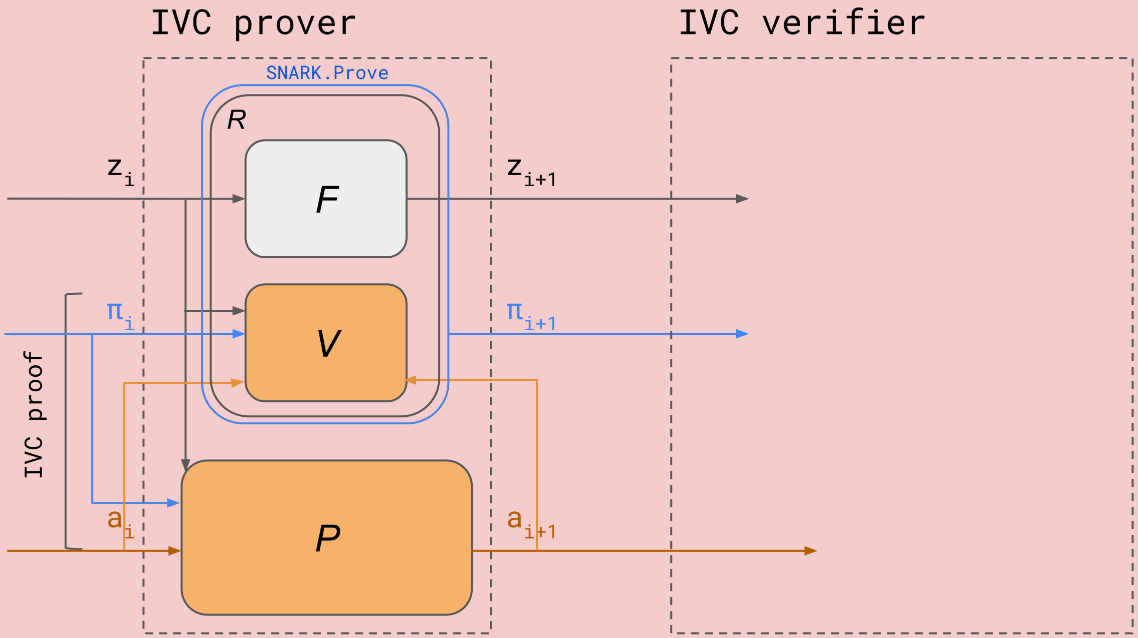


apply SNARK prover to the circuit R

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

accumulation scheme

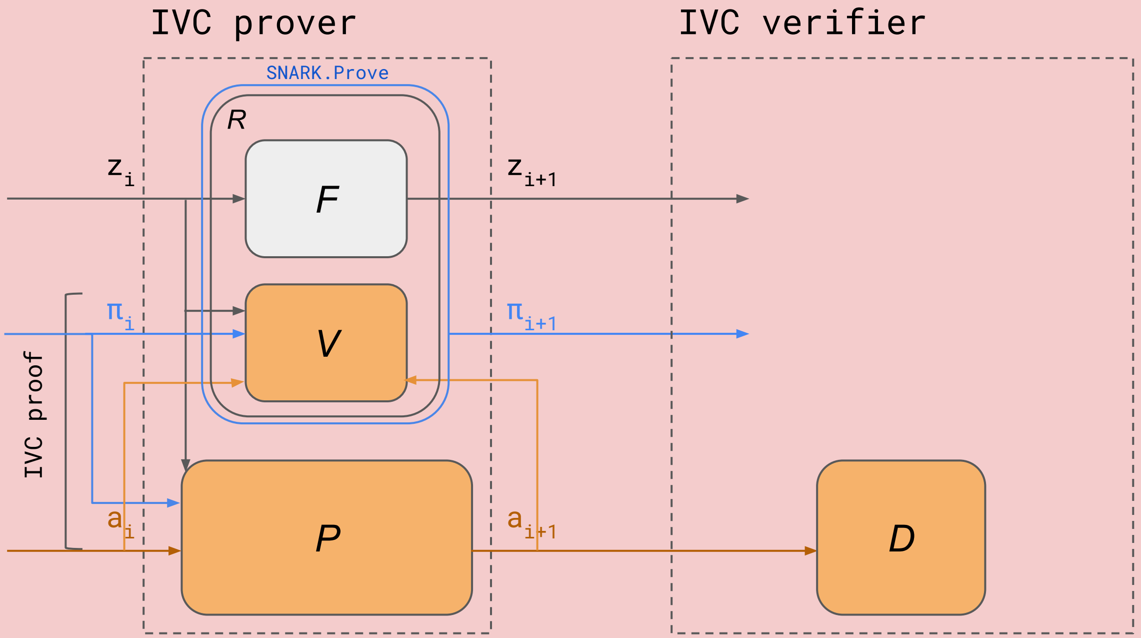


IVC verifier takes in some z, π, a

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

accumulation scheme

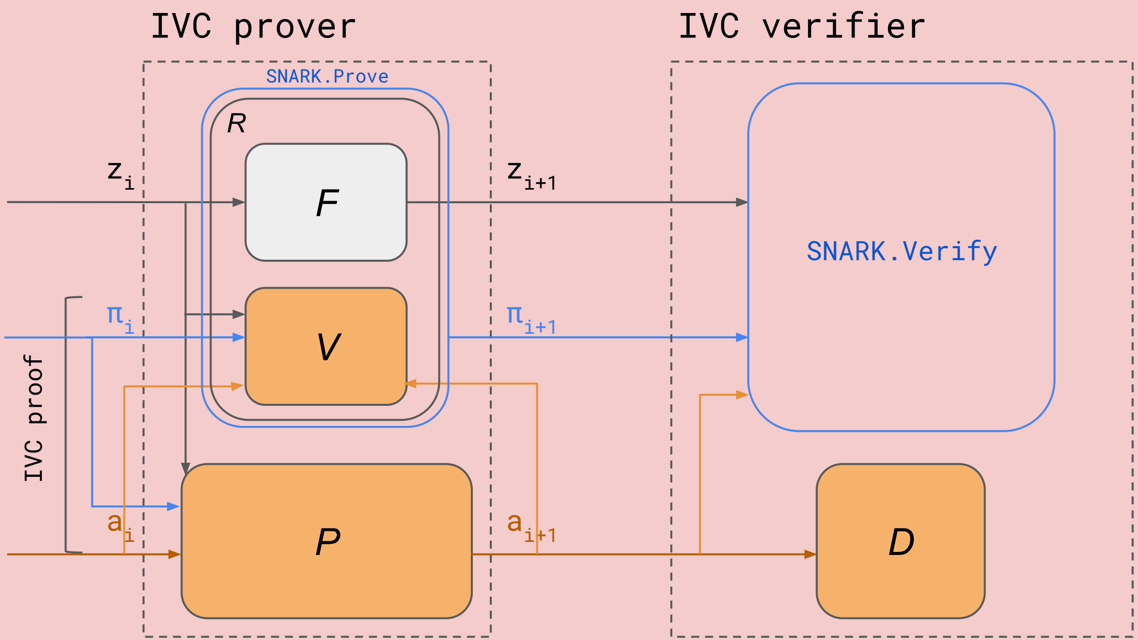


IVC verifier runs decider *D* to check that accumulator is well-formed

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

added in "Math Building Blocks" (15 Jun)

accumulation scheme



IVC verifier runs SNARK verifier

diagram adapted from https://iacr.org/cryptodb/data/paper.php?pubkey=30579

thank you!

any questions?