zero-knowledge programmable cryptography for verifiable computation

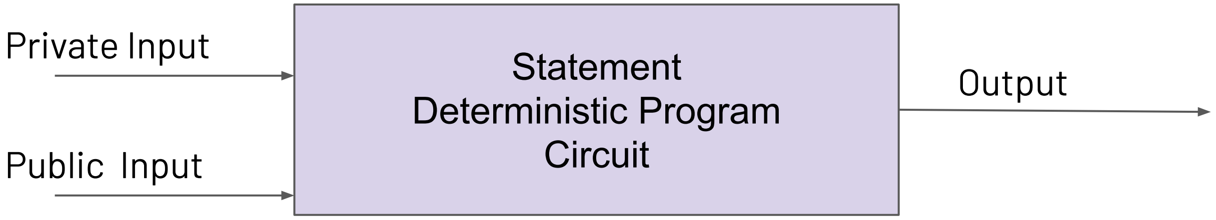
October 17th,2023

Universidad Complutense de Madrid

Jordi Baylina

@jbaylina

What’s a non interactive zero knowledge proof

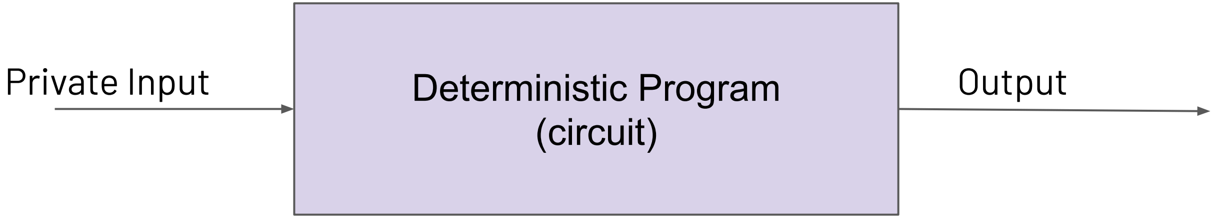


Given a circuit and its input, the prover can run the circuit and generate a proof.

Given the proof and the public input/output, I can proof to a verifier that who generated that proof knew the private input and executed the circuit.

This proof does not reveal anything about the private input

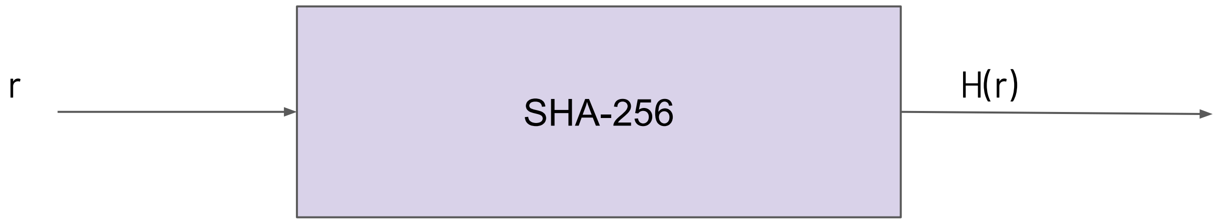
What's a zero knowledge proof



Proof = F(Circuit, PrivateInput)

V(Circuit, Output, Proof) = true/false

Example



Proof = F(SHA-256, r)

V(SHA-256, H(r), Proof) = true

Simple Circuit

Truth table

s1	s2	s3
0	0	1
0	1	1
1	0	1
1	1	0

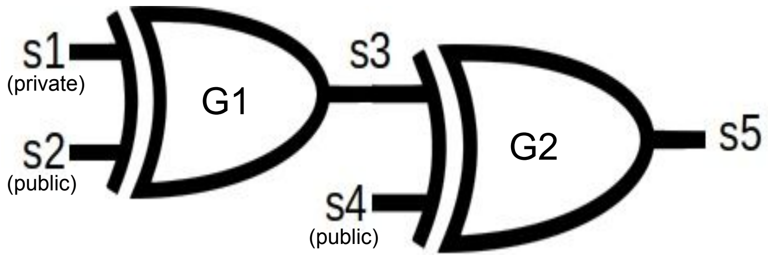
Constraints system

$$s_3 = 1 - s_1 s_2$$

Circom circuit

```
template NAND() {
    signal input s1;
    signal input s2;
    signal output s3;

    s3 <== 1 - s1*s2;
}

component main = NAND()
```

Composite Circuit



Constraints system

$$\begin{cases} s_3 = s_1 + s_2 - 2s_1s_2 \\ s_5 = s_3 + s_4 - 2s_3s_4 \end{cases}$$

nand.circom

```
template NAND() {
    signal input a;
    signal input b;
    signal output out;

    out <== 1 - a*b;
}
```

composite.circom

```
include "nand.circom"

template Composite() {
    signal private input s1;
    signal input s2;
    signal input s3;
    signal output s5;
    signal s4

    component G1 = NAND();
    component G2 = NAND();

    s1 ==> G1.a;
    s2 ==> G1.b;
    G1.out ==> G2.a;
    s3 ==> G2.b;
    G2.out ==> s5;
}

component main =
Composite();
```

Touching metal

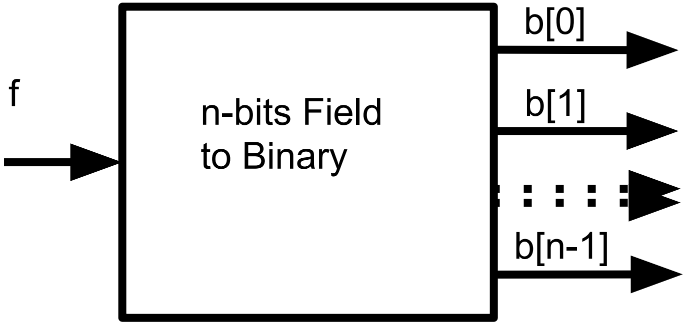
Constraints system

$$
\begin{cases}
b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 & = & f \\
b_0 \cdot (b_0 - 1) & = & 0 \\
b_1 \cdot (b_1 - 1) & = & 0 \\
b_2 \cdot (b_2 - 1) & = & 0
\end{cases}
$$

```
template Num2Bits3() {
    signal input f;
    signal output b[3];

     for (i=0; i<3; i++) {
         b[i] <-- (f>>i) & 1;
         b[i] * (b[i]-1) === 0;
     }
     b[0] + 2*b[1] + 4*b[2] ===
f;
}
```

```
==>   ✗   -->
          ===

<==   ✗   <--
          ===
```

Parametric templates

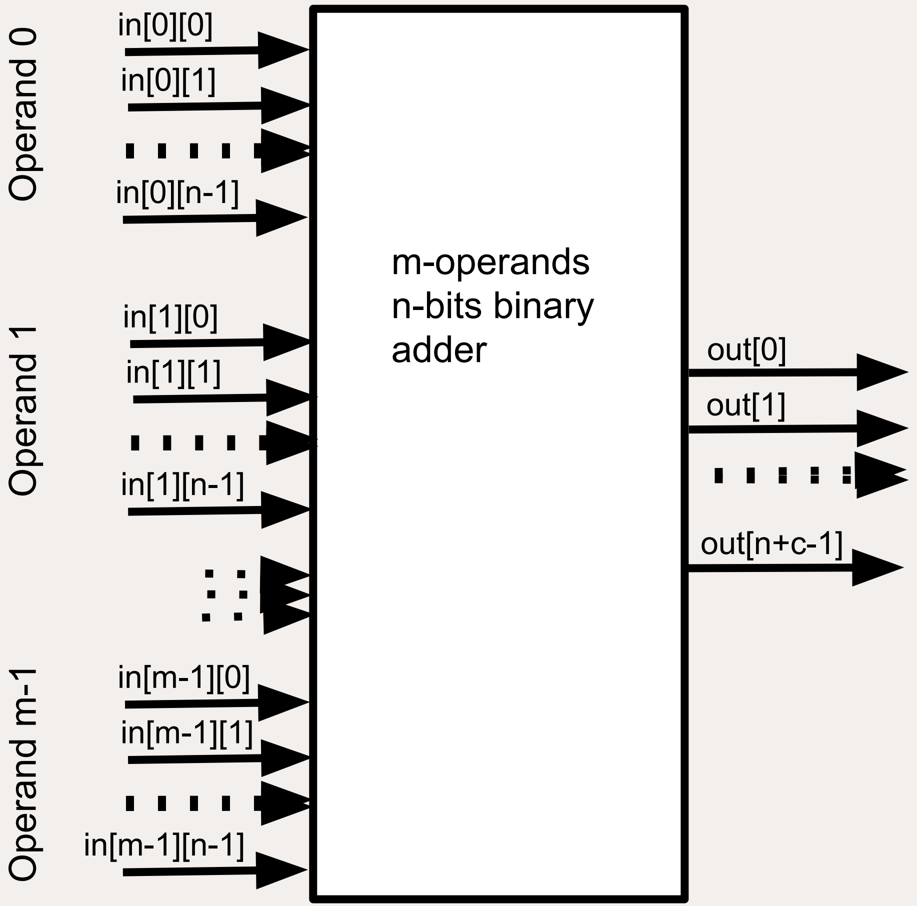


Constraints system

$$\begin{cases} b_0 \cdot 2^0 + b_1 \cdot 2^1 + \quad \dots \quad + b_{n-1} \cdot 2^{n-1} = f \\ b_0 \cdot (b_0 - 1) = 0 \\ b_1 \cdot (b_1 - 1) = 0 \\ \dots \\ b_{n-1} \cdot (b_{n-1} - 1) = 0 \end{cases}$$

```
template Num2Bits(n) {
    signal input f;
    signal output b[n];
    var lc1=0;

    for (var i = 0; i<n; i++) {
        b[i] <-- (f >> i) & 1;
        b[i] * (b[i] - 1) === 0;
        lc1 += b[i] * 2**i;
    }

    lc1 === f;
}

component main = Num2Bits(253);
```

Binary adder



Constraints system for generic binary adder

$$
\begin{cases}
\quad in_{0,0} \cdot 2^0 + in_{0,1} \cdot 2^1 + \quad \dots \quad + \quad in_{0,n-1} \cdot 2^{n-1} \quad + \\
+ \quad in_{1,0} \cdot 2^0 + in_{1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{1,n-1} \cdot 2^{n-1} \quad + \\
\quad \dots \qquad\qquad\qquad\qquad\qquad \dots \\
+ \quad in_{m-1,0} \cdot 2^0 + in_{m-1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{m-1,n-1} \cdot 2^{n-1} \quad = \\
= \quad out_0 \cdot 2^0 + out_1 \cdot 2^1 + \quad \dots \quad + \quad out_{n+c-1} \cdot 2^{n+c-1} \\
\\
out_0 \cdot (out_0 - 1) = 0 \\
out_1 \cdot (out_1 - 1) = 0 \\
\dots \\
out_{n+c-1} \cdot (out_{n+c-1} - 1) = 0
\end{cases}
$$

```
function nbits(a) {
    var n = 1;
    var r = 0;
    while (n-1<a) {
        r++;
        n *= 2;
    }
    return r;
}

template BinSum(n, m) {
    var nout = nbits((2**n -1)*m);
    signal input in[m][n];
    signal output out[nout];

    var lin = 0;
    var lout = 0;

    var k;
    var j;

    for (k=0; k<n; k++) {
        for (j=0; j<m; j++) {
            lin += in[j][k] * 2**k;
        }
    }

    for (k=0; k<nout; k++) {
        out[k] <-- (lin >> k) & 1;

        // Ensure out is binary
        out[k] * (out[k] - 1) === 0;

        lout += out[k] * 2**k;
    }

    // Ensure the sum;
    lin === lout;
}

component main = BinSum(64,2)
```

Baby Jub

The Order of the field is the order of the bn128 curve.

Field operations matches with the field of the circuit.

The curve is safe.

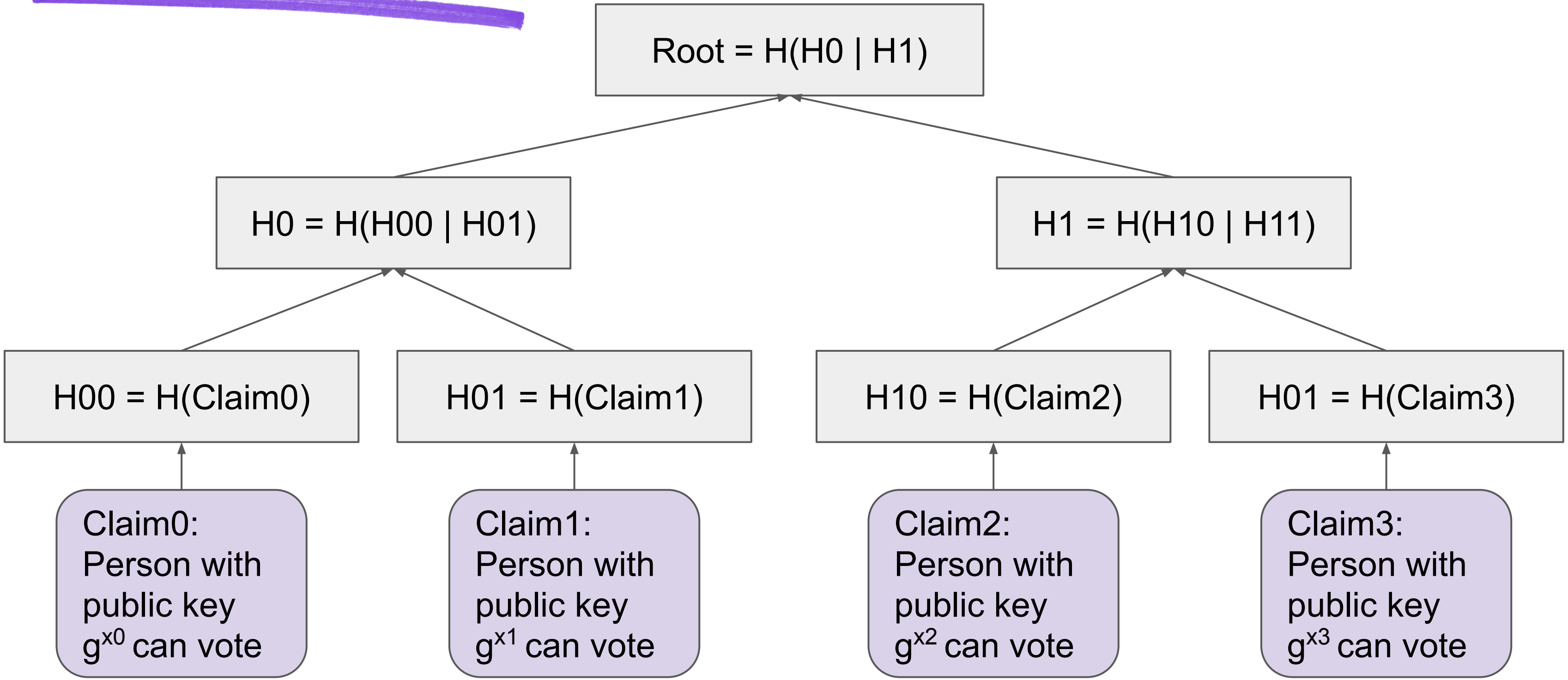
$$E_{E,a,d}: ax^2 + y^2 = 1 + dx^2y^2$$

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

$a=1$
$d=9706598848417545097372247223557719406784115219466060233080913168975159366771$

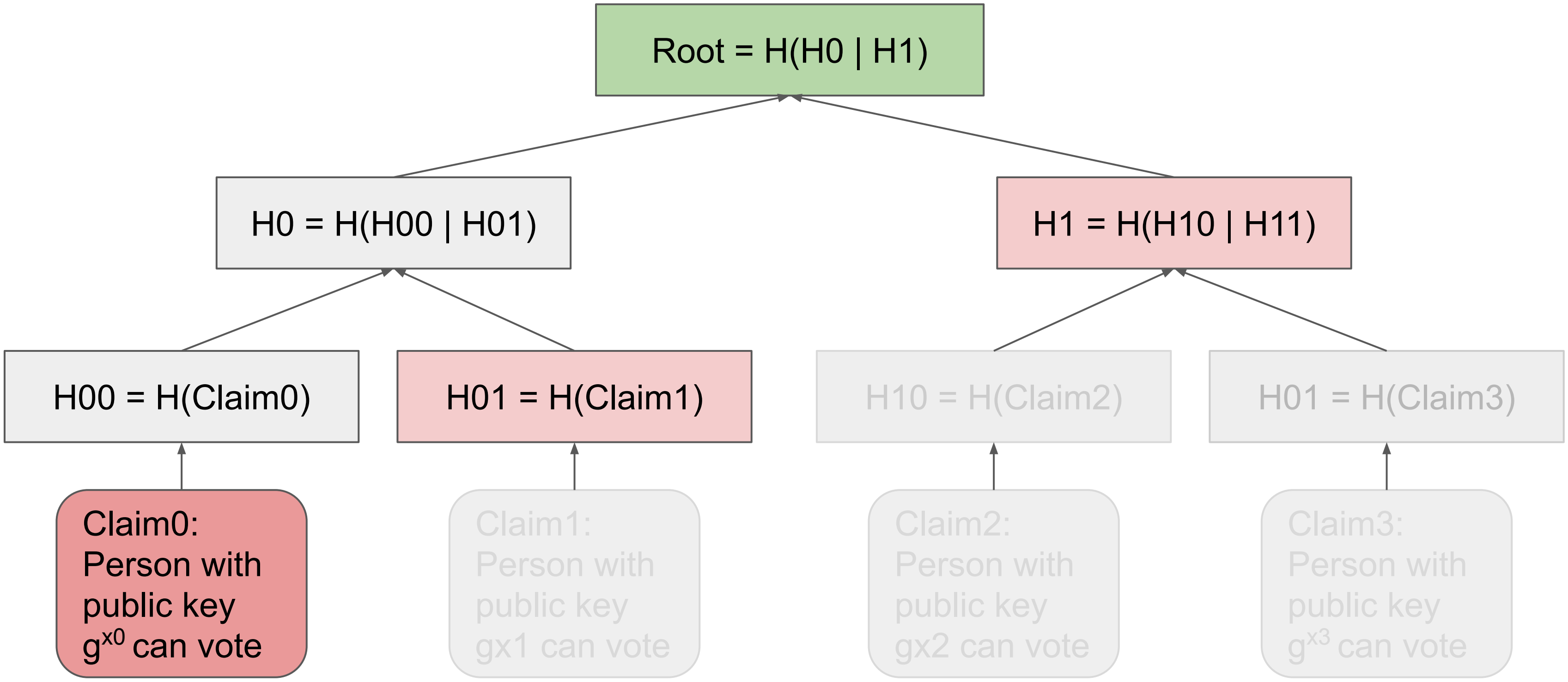
```
template BabyAdd() {
    signal input x1;
    signal input y1;
    signal input x2;
    signal input y2;
    signal output xout;
    signal output yout;

    signal beta;
    signal gamma;
    signal delta;
    signal tau;

    var a = 168700;
    var d = 168696;

    beta <== x1*y2;
    gamma <== y1*x2;
    delta <== (-a*x1+y1)*(x2 + y2);
    tau <== beta * gamma;

    xout <-- (beta + gamma) / (1+ d*tau);
    (1+ d*tau) * xout === (beta + gamma);

    yout <-- (delta + a*beta - gamma) / (1-d*tau);
    (1-d*tau)*yout === (delta + a*beta - gamma);
}
```

Merkle tree

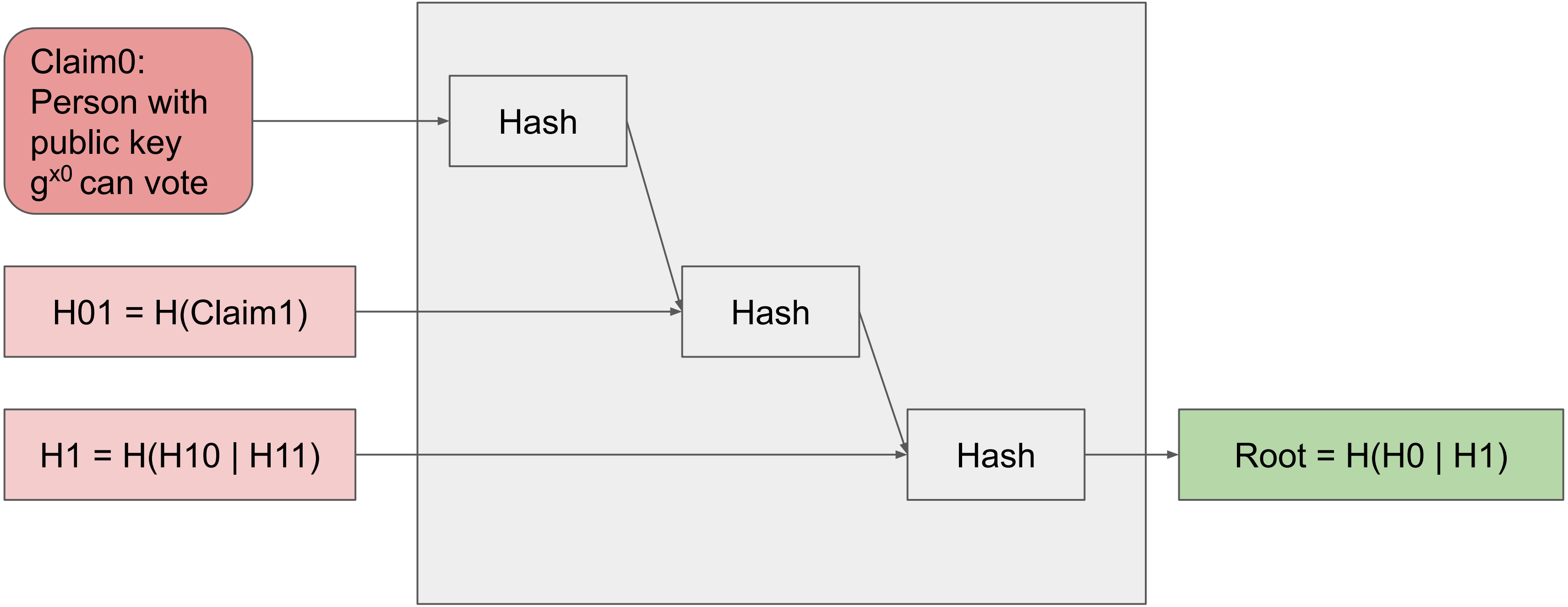


Root = H(H0 | H1)
H0 = H(H00 | H01)
H1 = H(H10 | H11)
H00 = H(Claim0)
H01 = H(Claim1)
H10 = H(Claim2)
H01 = H(Claim3)
Claim0: Person with public key g^{x0} can vote
Claim1: Person with public key g^{x1} can vote
Claim2: Person with public key g^{x2} can vote
Claim3: Person with public key g^{x3} can vote

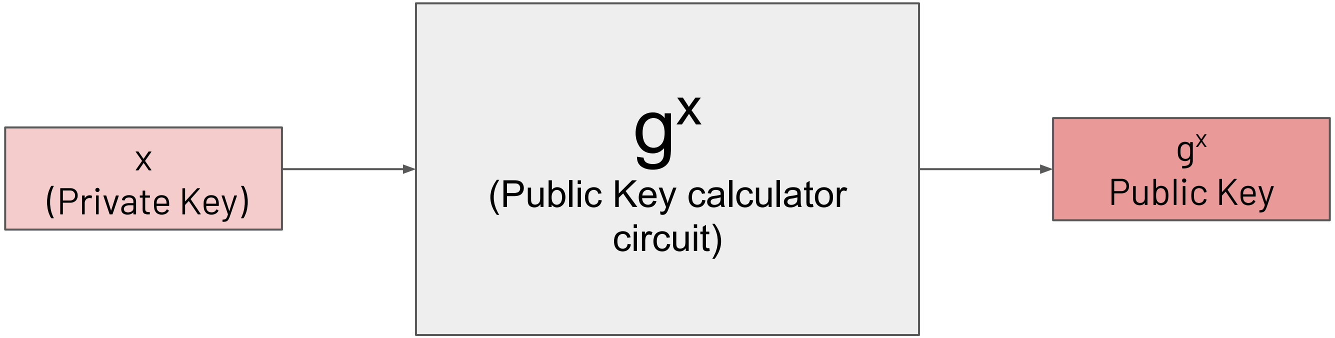
Merkle proof



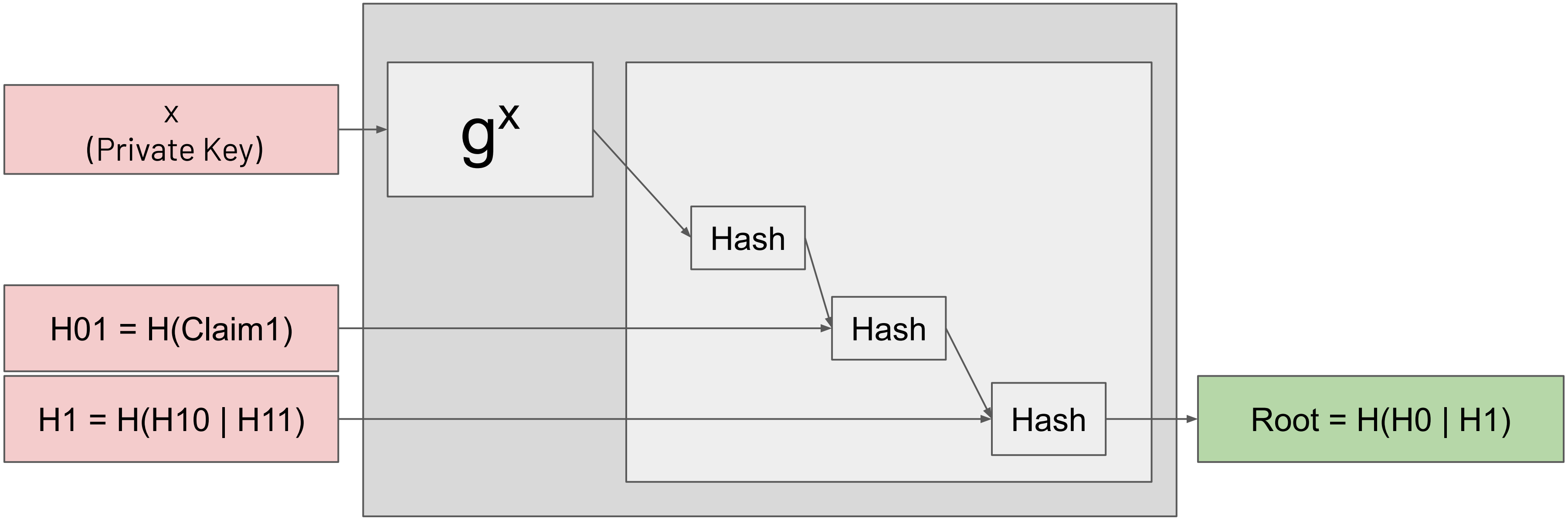
Merkle proof verifier circuit



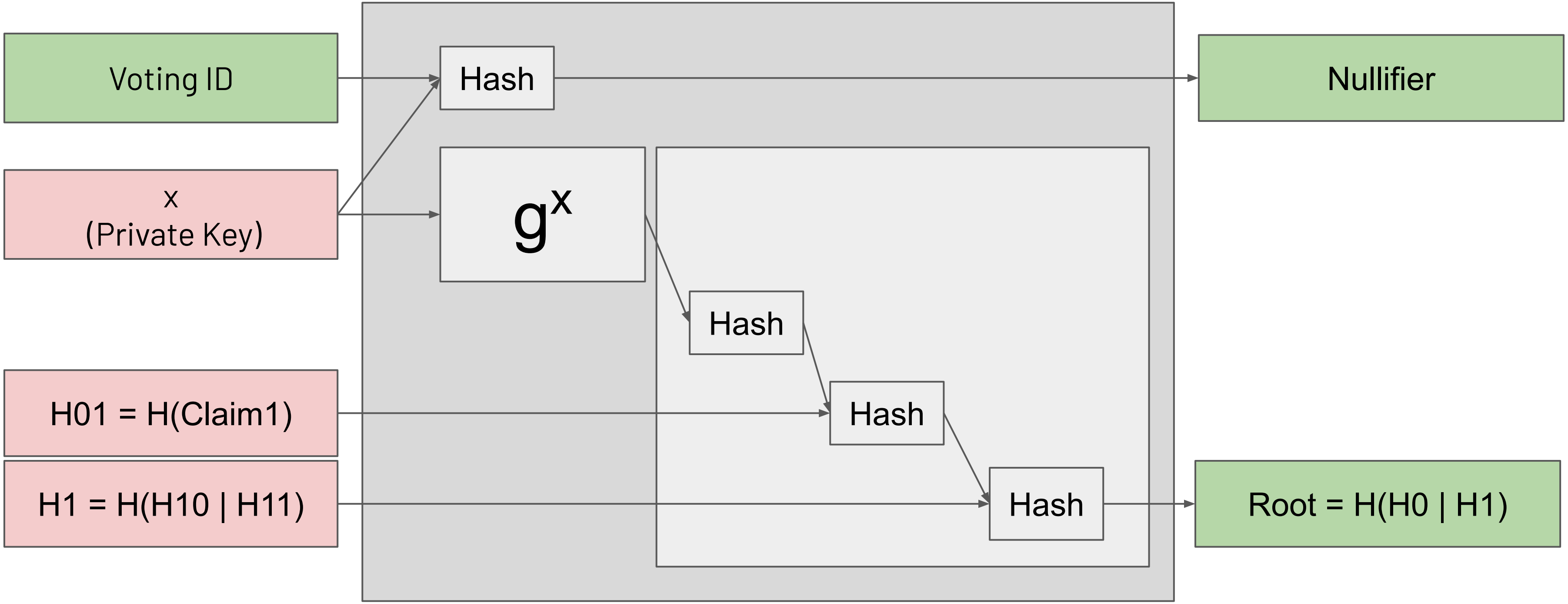
Proof of private key



Proof of belonging to a census



Preventing to vote more than once



CircomLib

https://github.com/iden3/circomlib

- Binary to Field and Field to Binary converters (With strict option)
- BabyJub
  - Addition / Constant scalar multiplication / Variable scalar multiplication
  - Edwards and Montgomery conversion
  - Point compression/decompression
- EdDSA
- Pedersen commitments
- MiMC7 Hash
- Sparse merkle trees processors to add/update/remove elements.
- Sparse merkle tree verifiers to verify inclusion and exclusion.
- Comparators
- Logical operators like adders and Binary Gates.
- SHA256 Hash function.
- ... and more

snarkJS:
https://github.com/iden3/snarkjs

circom:
https://github.com/iden3/circom

Polynomials representation

Coefficients representation

$p(x) = a_0 + a_1x + a_2x^2 \dots. + a_nx^n$

$[a_0, a_1, a_2, \dots. , a_n]$

fft →

← fft^{-1}

Evaluation representation

$p(x) = A_0L_0(x) + A_1L_1(x) + A_2L_2(x) + \dots. + A_nL_n(x)$

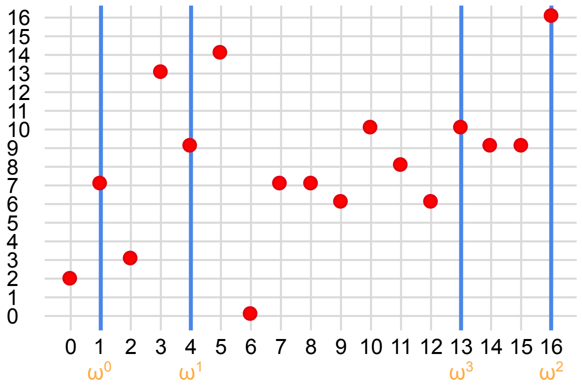
$\langle A_0, A_1, A_2, \dots. , A_n \rangle$

Numerical example

q=17
ω=4

x	f(x)
0	2
1 = ω^0	7
2	3
3	13
4 = ω^1	9
5	14
6	0
7	7
8	7
9	6
10	10
11	8
12	6
13 = ω^3	10
14	9
15	9
16 = ω^2	16

$f(x) = 2 + 3x + x^2 + x^3$



Coefficients representation: [2,3,1,1]

Evaluation representation: <7,9,16,10>

fft([2,3,1,1]) = <7,9,16,10>

fft^{-1}(<7,9,16,10>) = [2,3,1,1]

Polynomial protocols

Prover

cm(a(x))

cm(b(x))

cm(c(x))

Fiat Shamir heuristic to convert this protocol to non interactive:

r = HASH(cm(a(x) | cm(b(x) | com(c(x))

r

Verifier

a(r), opening(cm, r, a(r))

b(r), opening(cm, r, b(r))

c(r), opening(cm, r, c(r))

Verify a(r)+b(r) = c(r)

Verifiable relationships between polynomials

- General Polynomial relationship
  - Example:
    - $a(x) \cdot b(x) + 44 \cdot c(x) \cdot x^2 === (d(x) + 1)^2$
- Relationships with neighbor points
  - Example:
    - a(ωx) = a(x)^2 <1, 2, 4 ,16>
    - a(ωx) = b(x)+3*c(x)
- Specific Value at some point
  - Example:
    - Z(1) = 1
- Permutation between two polynomials in the roots of unity.
- Inclusion (Plookup)

Hello World: Fibonacci Series

$$\mathbb{F}_{\texttt{0xffffffff00000001}}$$

a_0	a_1	a_2	a_3	a_4		a_{1024}
1	2	3	5	8	………	**11696381471667068125**

$$a_{i+1} = a_{i-1} + a_i$$

fibonacci.circom

```
pragma circom 2.0.6;

template Fibonacci(n) {
    signal input a0;
    signal input a1;
    signal output out;

    signal im[n-1];

    for (var i=0; i<n-1; i++) {
        if (i==0) {
            im[i] <== a0 + a1;
        } else if (i==1) {
            im[i] <== a1 + im[0];
        } else {
            im[i] <== im[i-2] + im[i-1];
        }
    }
    out <== im[n-2];
}
component main = Fibonacci(1024);
```

Polynomial Identities State Machine

x	ISLAST(x)	aBeforeLast(x)	aLast(x)
1	0	1	2
ω	0	2	3
ω^2	0	3	5
ω^3	0	5	8
	...	...	...
ω^{1022}	0	16804231586740408223	13338893954341244223
ω^{1023}	1	13338893954341244223	11696381471667068125

aBeforeLast(ωx) = aLast(x)
aLast(ωx) = aBeforeLast(x) + aLast(x)

Hello world

fibonacci.pil

```
constant %N = 1024;

namespace Fibonacci(%N);
   pol constant ISLAST;                // 0,0,0,0,.....,1
   pol commit aBeforeLast, aLast;

   (1-ISLAST) * (aBeforeLast' - aLast) = 0;
   (1-ISLAST) * (aLast' - (aBeforeLast + aLast)) = 0;

   public out = aLast(%N-1);
   ISLAST *( aLast- :out) = 0;
```

fibonacci.js

```
const { FGL } = require("pil-stark");

module.exports.buildConstants = async function
(pols) {

   const N = pols.ISLAST.length;

   for ( let i=0; i<N; i++) {
       pols.ISLAST[i] = (i == N-1) ? 1n : 0n;
   }
}

module.exports.execute = async function (pols,
input) {

   const N = pols.aLast.length;

   pols.aBeforeLast[0] = BigInt(input[0]);
   pols.aLast[0] = BigInt(input[1]);

   for (let i=1; i<N; i++) {
       pols.aBeforeLast[i] =pols.aLast[i-1];
       pols.aLast[i] = FGL.add(
            pols.aBeforeLast[i-1],
            pols.aLast[i-1]
       );
   }
   return pols.aLast[N-1];
}
```

fibonacci.test.js

```
const assert = require("assert");
const path = require("path");
const { FGL, starkSetup, starkGen, starkVerify } =
      require("pil-stark");
const { newConstantPolsArray, newCommitPolsArray,
      compile, verifyPil } = require("pilcom");
const smFibonacci = require("../src/fibonacci.js");

describe("test fibonacci sm", async function () {
   this.timeout(10000000);
   let constPols, cmPols, pil;

   it("It should create the pols main", async () => {
       pil = await compile(
          FGL, path.join(__dirname, "../src/fibonacci.pil"));

      constPols =  newConstantPolsArray(pil);

       await smFibonacci.buildConstants(
          constPols.Fibonacci);

       cmPols = newCommitPolsArray(pil);

       const result = await smFibonacci.execute(
          cmPols.Fibonacci, [1,2]);

       console.log("Result: " + result);

       const res = await verifyPil(
             FGL, pil, cmPols, constPols);
       assert(res.length == 0);
   });
   it("It should generate and verify the stark", async () => {
       const starkStruct = {
           nBits: 10,
           nBitsExt: 14,
           nQueries: 32,
           verificationHashType : "GL",
           steps: [
               {nBits: 14},
               {nBits: 9},
               {nBits: 4}
           ]
       };
       const setup = await starkSetup(
             constPols,
             pil,
             starkStruct
       );
       const resP = await starkGen(
                   cmPols,
                   constPols,
                   setup.constTree,
                   setup.starkInfo
       );
       const resV = await starkVerify(
                   resP.proof,
                   resP.publics,
                   setup.constRoot,
                   setup.starkInfo
       );
       assert(resV==true);
   });
});
```

Permutation Checks

x	a(x)	b(x)
1	3	1
ω	2	2
ω^2	6	3
ω^3	5	4
ω^4	4	5
ω^5	8	6
ω^6	7	7
ω^7	1	8

```
namespace PermutationExample(%N);
   pol commit a, b;

   a is b;
```

Higher Complexity Permutation Checks

x	selA(x)	a1(x)	a2(x)	SELB(x)	B1(x)	B2(x)
1	1	3	333	1	1	111
ω	1	2	222	1	2	222
ω^2	0			1	3	333
ω^3	0			1	4	444
ω^4	1	4	444	0		
ω^5	0			0		
ω^6	0			0		
ω^7	1	1	111	0		

```
namespace PermutationExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   selA { a1, a2 } is SELB { B1, B2 };
```

Plookup

x	a(x)	b(x)
1	3	1
ω	3	2
ω^2	6	3
ω^3	5	4
ω^4	6	5
ω^5	6	6
ω^6	1	7
ω^7	1	8

```
namespace PlookupExample(%N);
   pol commit a, b;

   a in b;
```

Higher Complexity Plookup

x	selA(x)	a1(x)	a2(x)	SELB(x)	B1(x)	B2(x)
1	1	3	333	1	1	111
ω	1	3	333	1	2	222
ω^2	0			1	3	333
ω^3	0			1	4	444
ω^4	1	4	444	0		
ω^5	0			0		
ω^6	0			0		
ω^7	1	1	111	0		

```
namespace PlookupExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   selA { a1, a2 } in SELB { B1, B2 };
```

Connection Checks

x	a(x)	S(x)
1	3	ω^5
ω	66	ω^6
ω^2	1833	ω^7
ω^3	3	1
ω^4	3	ω^3
ω^5	3	ω^4
ω^6	66	ω
ω^7	1833	ω^2

```
namespace ConnectionExample(%N);
    pol constant S;
    pol commit a;

    a connect S;
```

Higher Complexity Connection Checks

x	a(x)	b(x)	c(x)	S1(x)	S2(x)	S3(x)
1	1	2	3	1	k_1	ω^3
ω	3	4	5	k_2	$k_1\ \omega$	$k_1\ \omega^2$
ω^2	3	5	6	ω	$k_2\ \omega$	$k_1\ \omega^3$
ω^3	3	6	7	ω^2	$k_2\ \omega^2$	$k_2\ \omega^3$
ω^4				ω^4	$k_1\ \omega^4$	$k_2\ \omega^4$
ω^5				ω^5	$k_1\ \omega^5$	$k_2\ \omega^5$
ω^6				ω^6	$k_1\ \omega^6$	$k_2\ \omega^6$
ω^7				ω^7	$k_1\ \omega^7$	$k_2\ \omega^7$

```
namespace PermutationExample(%N);
   pol constant SELB, B1, B2;
   pol commit selA, a1, a2;

   { a1, a2, a3 } connect { S1, S2, S3 };
```

Plonk Example

```
// Plonk circuit

namespace main;

    pol committed a, b, c
    pol constant Sa, Sb, Sc;
    pol constant Ql, Qr, Qm, Qo, Qc;
    pol constant L1;                            // 1, 0, 0, …

    public publicInput = a(0);

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

    pol ab = a*b;
    Ql*a + Qr*b + Qo*c + Qm*ab + Qc = 0;

    L1 * (a - :publicInput) = 0;
```

Custom Gates in CIRCOM

```
pragma circom 2.0.6;
pragma custom_templates;

template custom CMul() {
   signal input ina[3];
   signal input inb[3];
   signal output out[3];

   var A = (ina[0] + ina[1])  * (inb[0] + inb[1]);
   var B = (ina[0] + ina[2])  * (inb[0] + inb[2]);
   var C = (ina[1] + ina[2])  * (inb[1] + inb[2]);
   var D = ina[0]*inb[0];
   var E = ina[1]*inb[1];
   var F = ina[2]*inb[2];
   var G = D-E;

   out[0] <-- C+G-F;
   out[1] <-- A+C-E-E-D;
   out[2] <-- B-G;
}
```

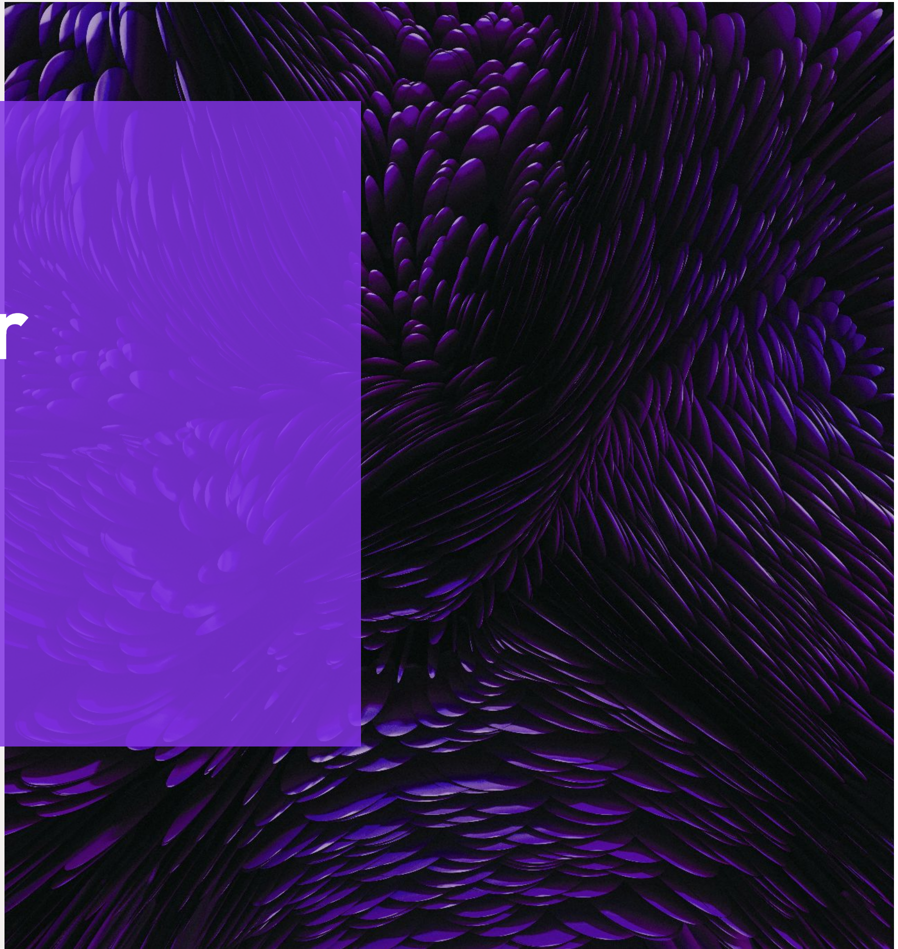
Custom Gates in CIRCOM

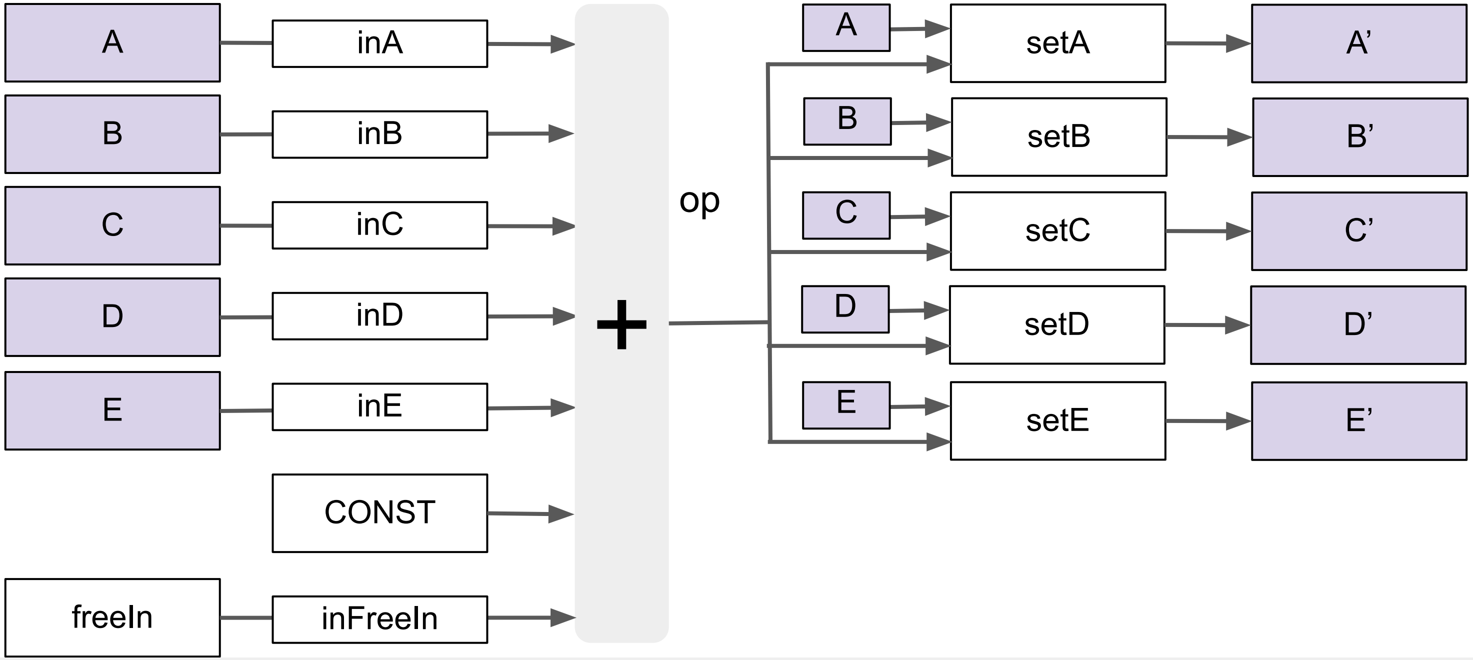
- Can be used as normal templates.
- Witness calculator is generated by CIRCOM like any other template.
- No constraints are allowed.
- All the custom gates are exported to the .r1cs file.
- This allows to do a circuit in circom and proof/verify it with a STARK!
- Supported primes in circom: BN128, BLS-12381, Goldilocks

Plonk Example

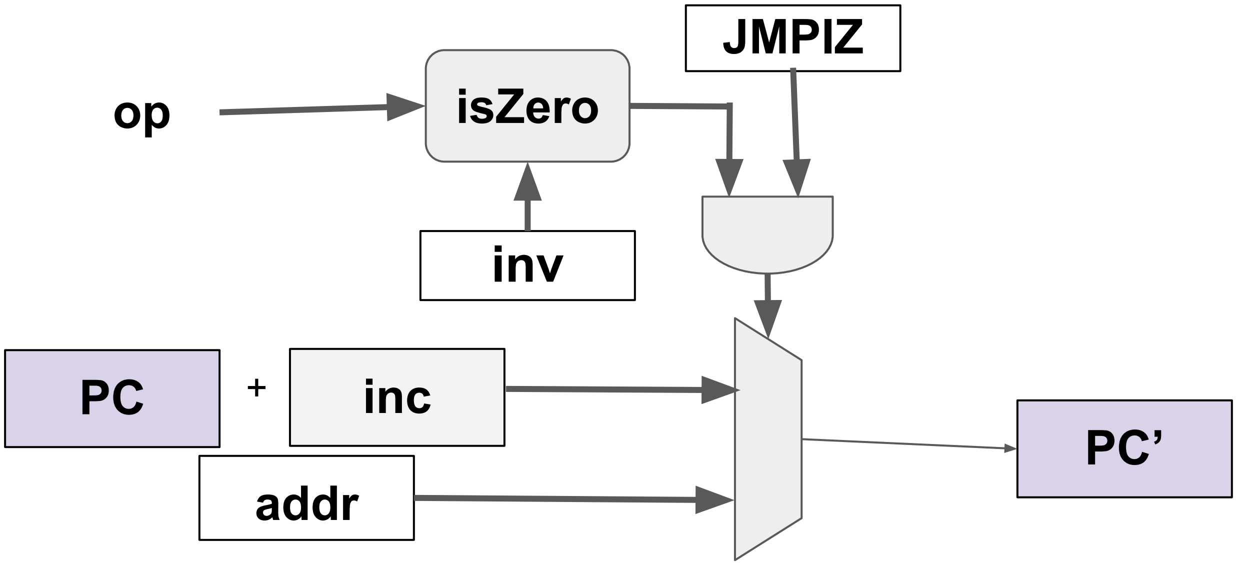
```
// Plonk circuit

namespace main;

    pol committed a, b, c
    pol constant Sa, Sb, Sc;
    pol constant Ql, Qr, Qm, Qo, Qc;
    pol constant L1;                          // 1, 0, 0, …

    public publicInput = a(0);

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

    pol ab = a*b;
    Ql*a + Qr*b + Qo*c + Qm*ab + Qc = 0;

    L1 * (a - :publicInput) = 0;
```

Building a processor with arithmetics






```
pol op = A*inA + B*inB + C*inC + D*inD + E*inE + freeIn*inFreeIn + CONST;
A’ = (op - A) * setA + A;
B’ = (op - B) * setB + B;
C’ = (op - C) * setC + C;
D’ = (op - D) * setD + D;
E’ = (op - E) * setE + E;
```




```
pol commit inv;
pol isZero = 1 -op * inv;
isZero * op = 0;
pol jmp = JMPIZ*isZero;

PC’ = jmp * (addr - (PC+INC)) + (PC+INC);
```

Execution Trace

PROGRAM COUNTER REGISTER	INSTRUCTION
0	ADD
1	JMP 5
5	MUL
6	JMP 5
5	MUL
6	JMP 5

$\subset$

ROM

PROGRAM LINE	INSTRUCTION
0	ADD
1	JMP 5
2	ADD
3	ADD
5	MUL
6	JMP 5

Execution Trace

COUNT	INSTRUCTION	ADDR	VALUE
0			
1	WR	5	2
2	WR	3	8
3	RD	5	2
4			
5	RD	5	2
6	RD	3	8
7	WR	5	34
8			
9	RD	5	34

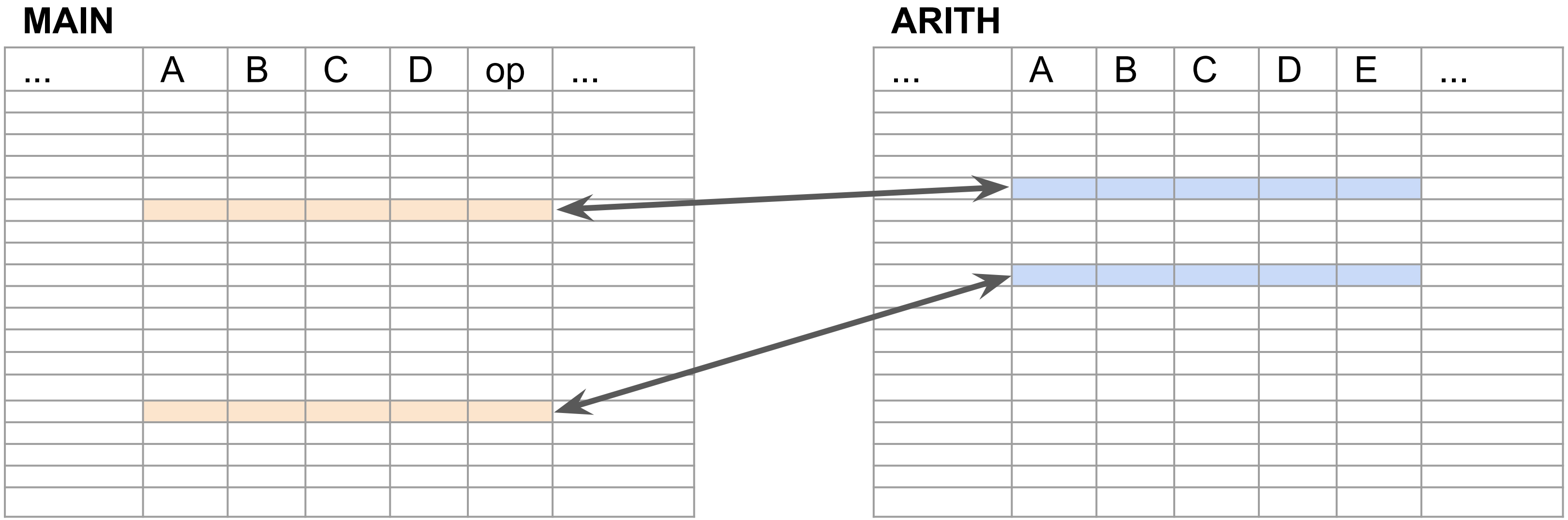
$\subset$

Memory

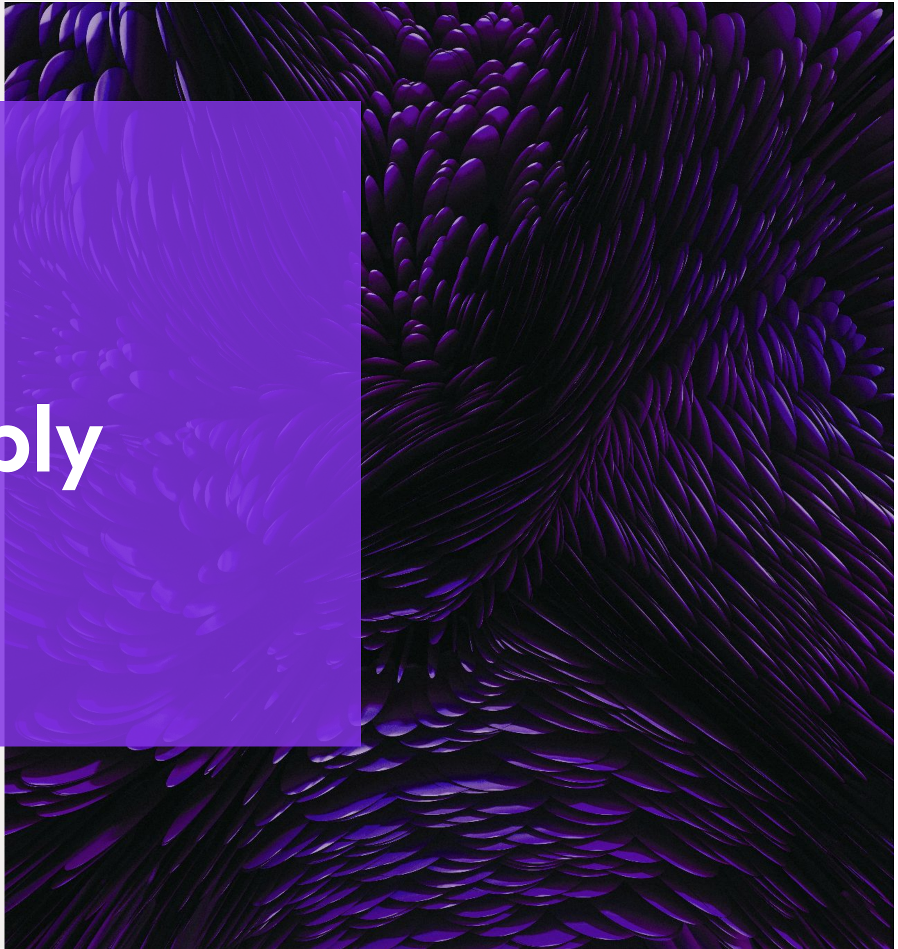
ADDR	COUNT	INSTRUCTION	VALUE
3	2	WR	2
3	6	RD	2
5	1	WR	8
5	3	RD	8
5	5	RD	8
5	7	WR	34
5	9	RD	34

Connecting two state machines with Plookup

```
MAIN.arith {MAIN.A , MAIN.B , MAIN.C , MAIN.D, MAIN.op} in
ARITH.latch {ARITH.A , ARITH.B , ARITH.C , ARITH.D , ARITH.E};
```



zkROM Writting programs in assembly



ZKASM-ROM

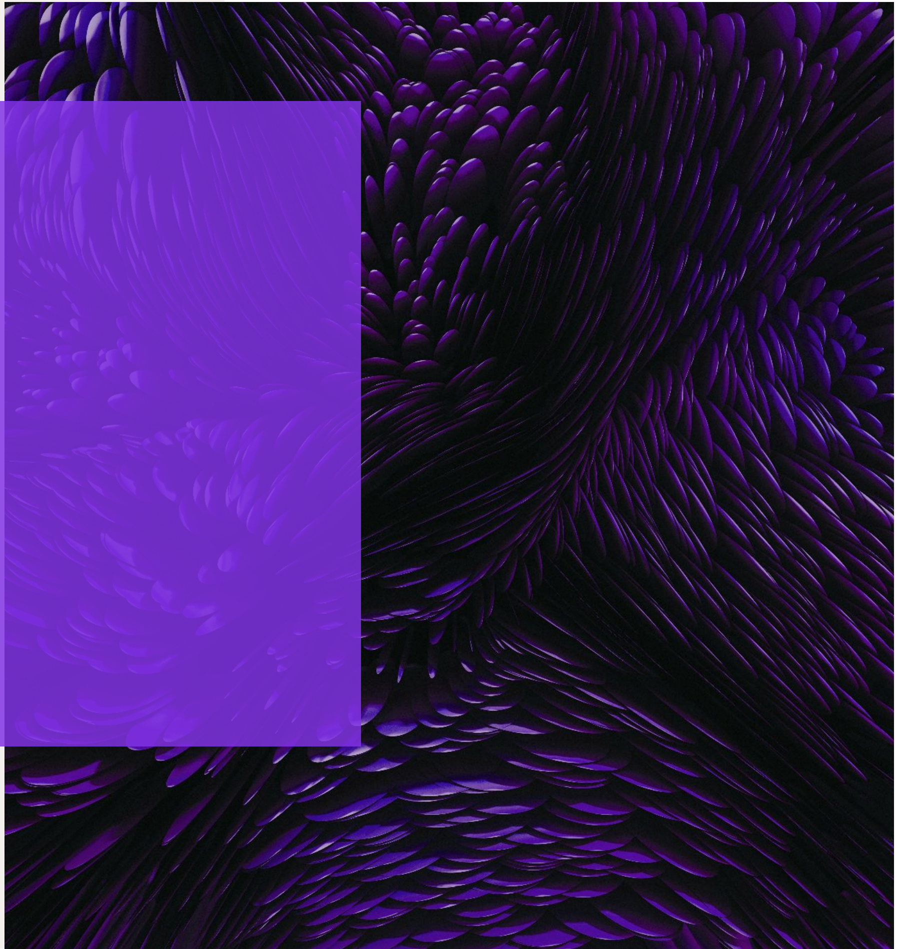
- Ethereum Transaction processor
- FREE Input the Transactions and the hash must match.
- zkCounters to prevent the proof to fail (DoS).

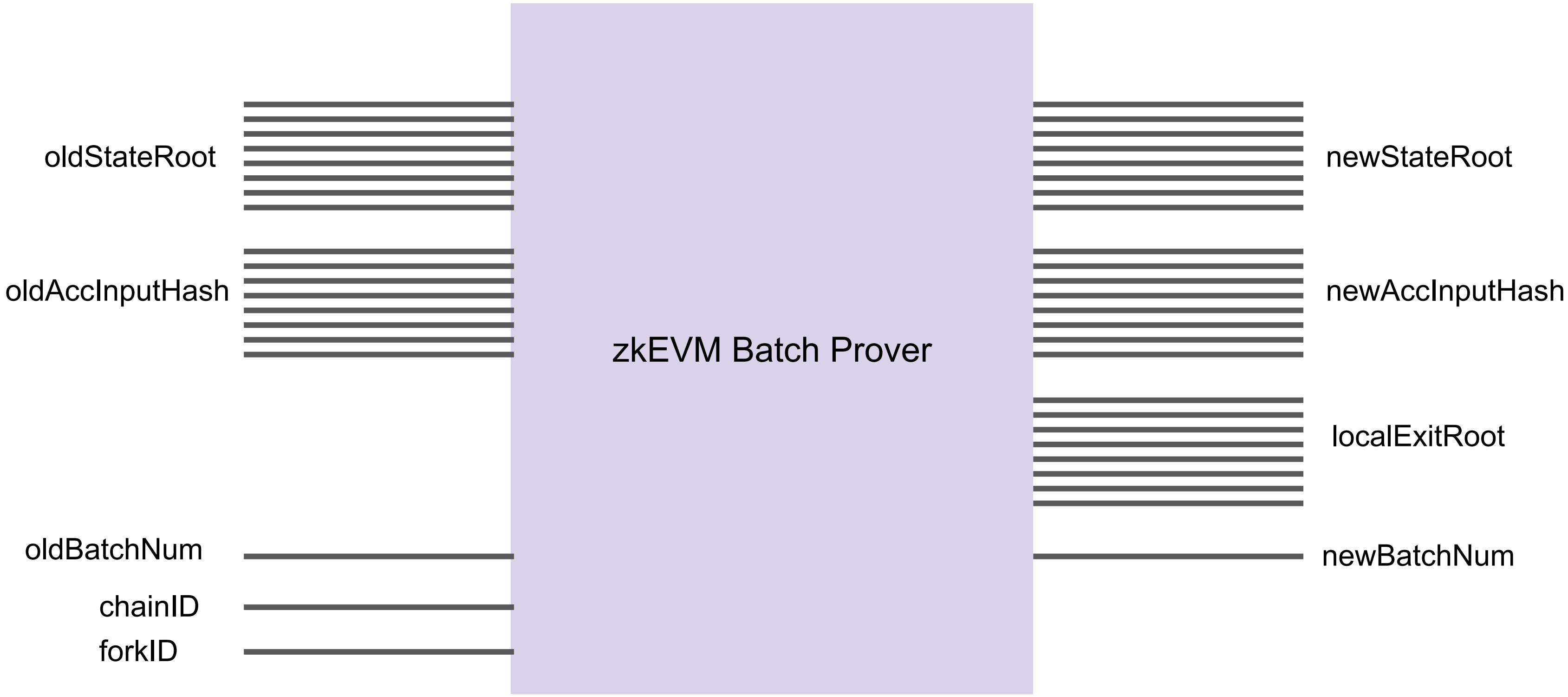
Some examples:

- Opcodes
- RLP Processing

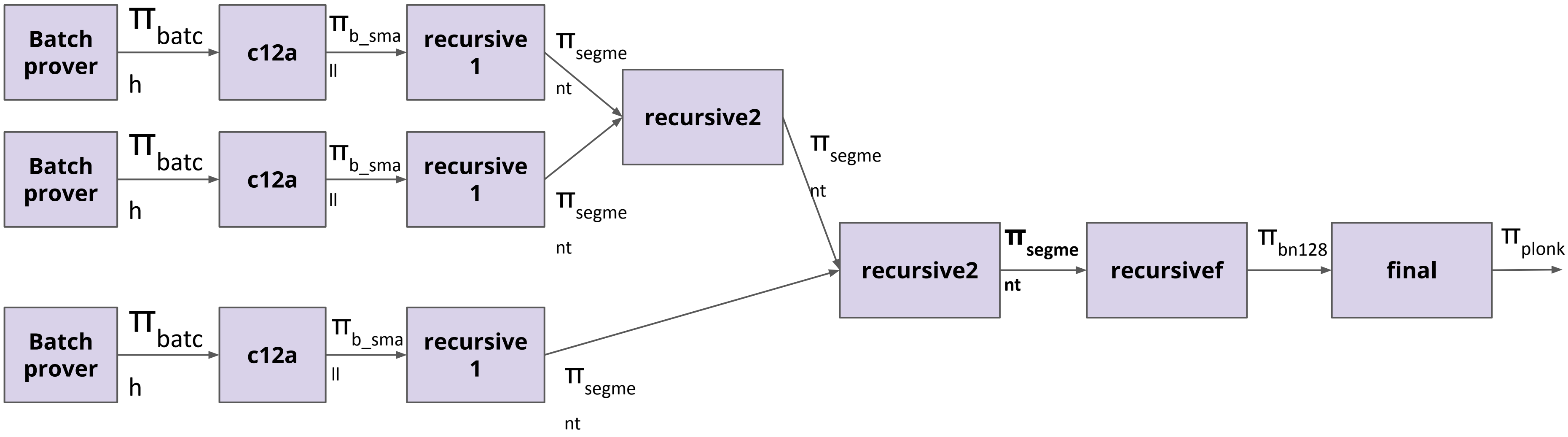
```
opPUSH31:
    31 => D
    $ => B                              :MLOAD(isCreateContract)
    0 - B                               :JMPN(opAuxPUSHB)
                                        :JMP(opAuxPUSHA)

opPUSH32:
    32 => D
    $ => B                              :MLOAD(isCreateContract)
    0 - B                               :JMPN(opAuxPUSHB)
                                        :JMP(opAuxPUSHA)

opDUP1:

    %MAX_CNT_STEPS - STEP - 120 :JMPN(outOfCounters)

    SP - 1 => SP    :JMPN(stackUnderflow)
    $ => A          :MLOAD(SP++)
    1024 - SP       :JMPN(stackOverflow)
    A               :MSTORE(SP++)
    1024 - SP       :JMPN(stackOverflow)
    GAS-3 => GAS    :JMPN(outOfGas)
                    :JMP(readCode)

opDUP2:

    %MAX_CNT_STEPS - STEP - 120 :JMPN(outOfCounters)

    SP - 2 => SP    :JMPN(stackUnderflow)
    $ => A          :MLOAD(SP)
```

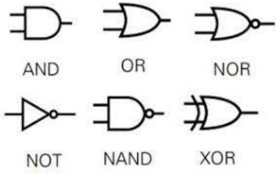
Recursion




oldStateRoot
oldAccInputHash
oldBatchNum
chainID
forkID
zkEVM Batch Prover
newStateRoot
newAccInputHash
localExitRoot
newBatchNum

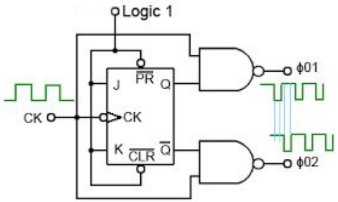
Recursion and on-chain verification





R1CS

circom



Polynomial Identities/ State Machines

PIL

Polynomial Identity Language

zkASM

polygon zkEVM

Build a better world!

polygon zkEVM

Thank you

@jbaylina