Lesson 12

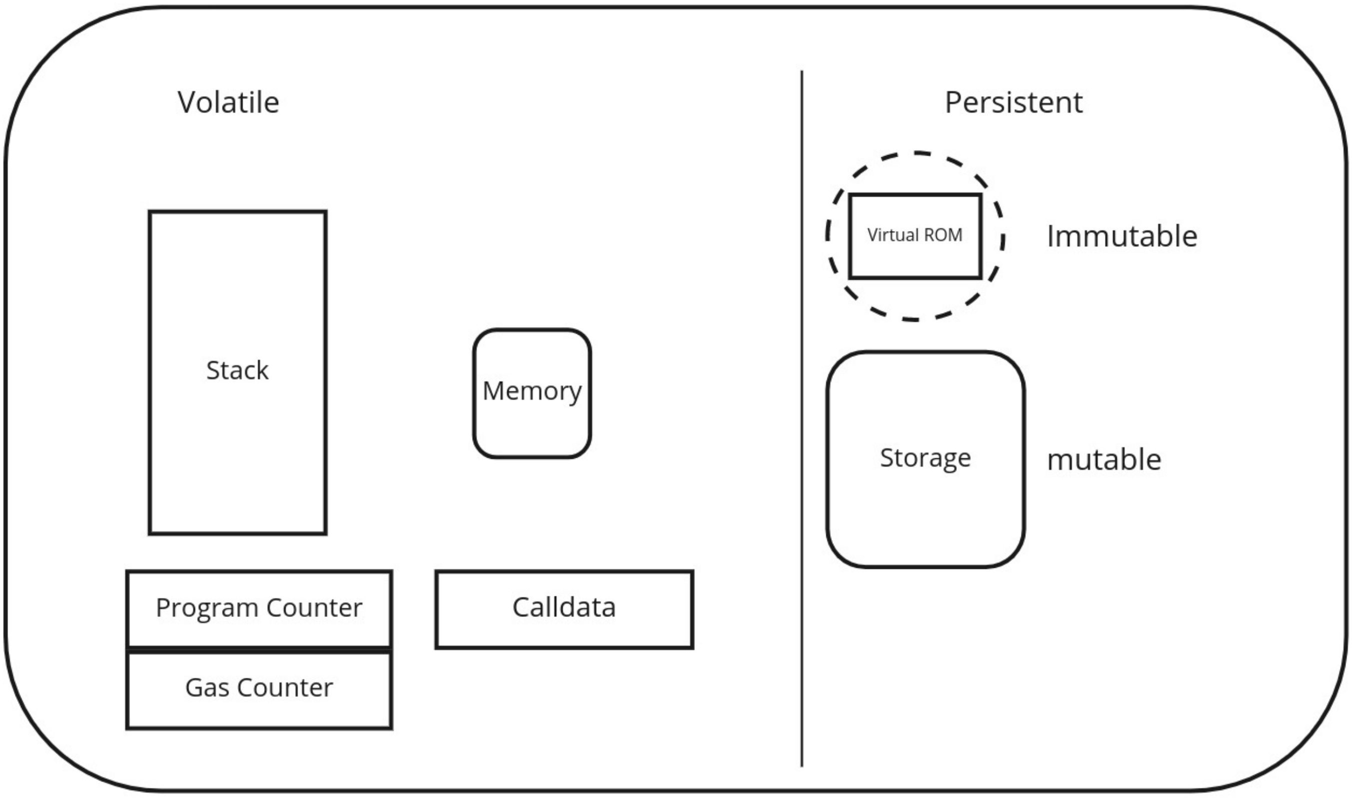
Week 3

Lesson 9 - What's next in L2 part 2 : L3s/Hyperchains

Lesson 10 - Privacy in Layer 2

Lesson 11 - What are ZK EVMs part 1 - overview

Lesson 12- What are ZK EVMs part 2 - universal circuits/circuit compiler

> ### EVM Operations
>
> Before we look at specific circuit details it is useful to look at the processing in the EVM and hence what are requirements are.
>
> 
>
>
> The state transition we are interested in is the change in storage, but that is the result of interactions between the

different compenents of the EVM.

The core of the computation occurs on the stack since the EVM is a stack based VM (rather than register based).

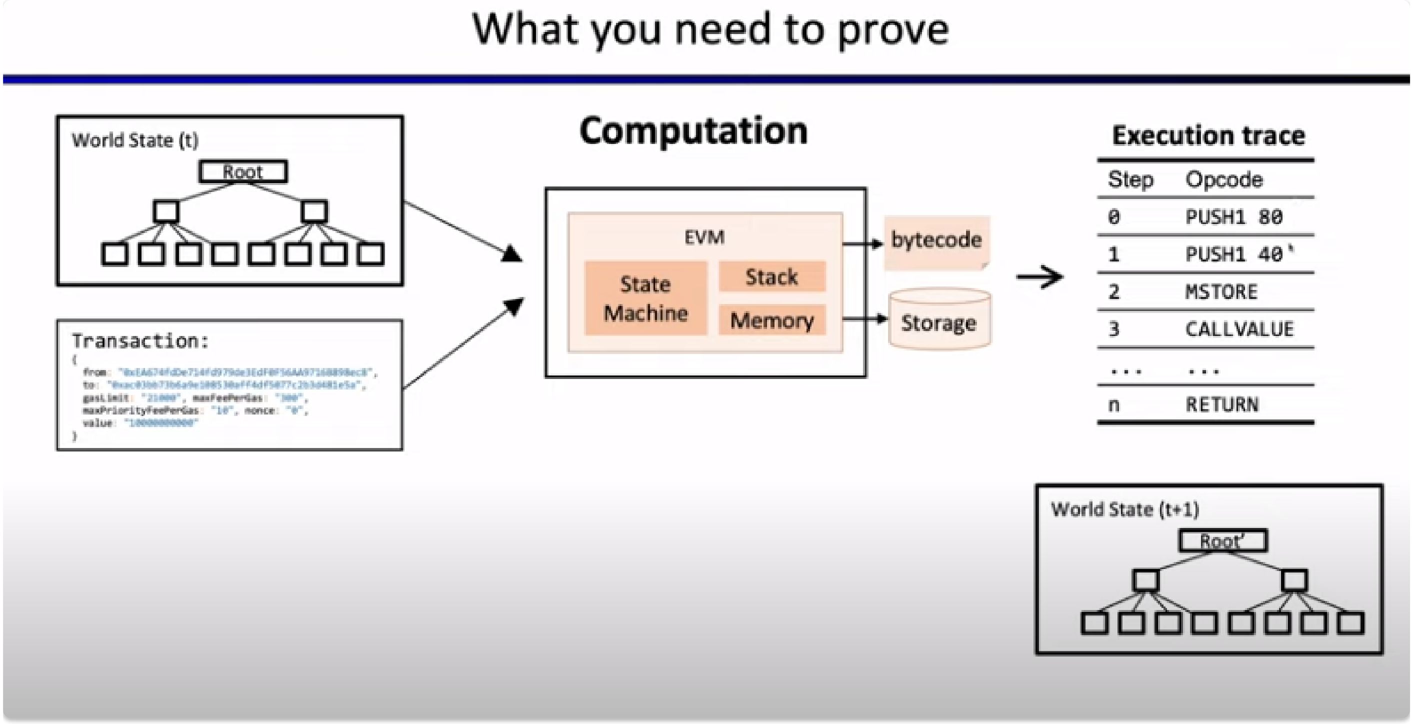
For example to add the numbers 2 and 4
we have to first push them onto the stack, then call the ADD opcode.

```
 PUSH1 2
PUSH1 4
ADD
```

```
 PUSH1 2
|__2__|


PUSH1 4
|__4__|
|__2__|


ADD
|__6__|
```



What you need to prove

World State (t)

Root

```
Transaction:
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aFF4dF5077c2b3d481e5a",
  gasLimit: "21000", maxFeePerGas: "300",
  maxPriorityFeePerGas: "10", nonce: "0",
  value: "10000000000"
}
```

Computation

EVM

State Machine

Stack

Memory

bytecode

Storage

Execution trace

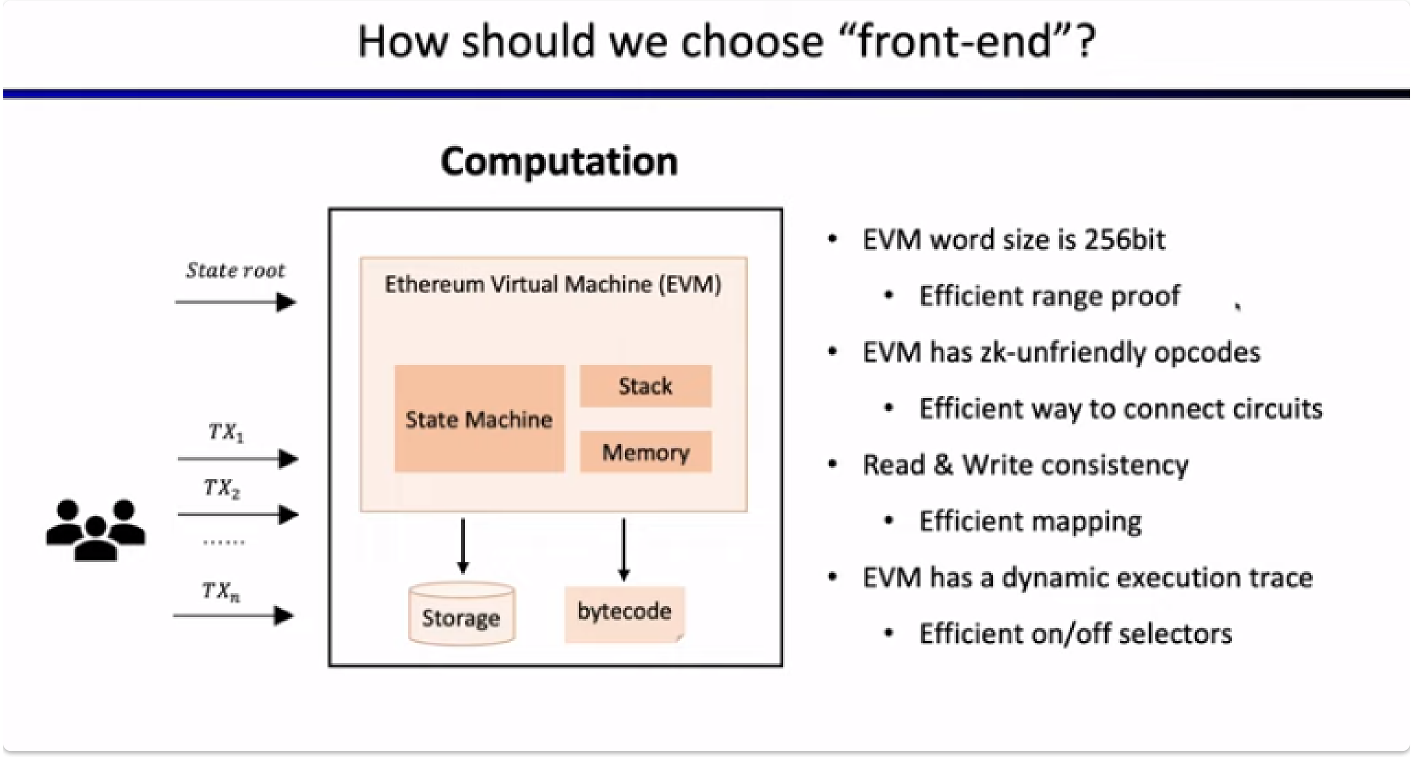
Step	Opcode
0	PUSH1 80
1	PUSH1 40
2	MSTORE
3	CALLVALUE
...	...
n	RETURN

World State (t+1)

Root'

> **zkSync zkEVM Architecture**
>
> See Primer
>
> The EraVM has
>
> - registers: 16 general-purpose registers: `r0`, `r1`, …, `r15`.
>   `r0` is a special constant register: reading it yields 0, storing to it is ignored.
> - flags: three distinct boolean registers LT (less-than), EQ (equals, the result is zero) and GT (greater-than). Instructions may set or clear flags depending on computation results.
> - data stack: holds 216 words, is free to use.
> - heap: for data that we want to pass around between functions and contracts. Heap is bounded, accesses are only free inside the bound, and we have to pay for growing the bound.
> - code memory: stores code of currently running contracts. May also be used as a constant pool.

Design Challenges



How should we choose "front-end"?

- EVM word size is 256bit
  - Efficient range proof
- EVM has zk-unfriendly opcodes
  - Efficient way to connect circuits
- Read & Write consistency
  - Efficient mapping
- EVM has a dynamic execution trace
  - Efficient on/off selectors

Building a zkEVM presents several significant challenges for the following reasons:

1. **Limited Support for Elliptic Curves in EVM**: Currently, EVM only supports BN254 pairing, making it challenging to perform proof recursion since cyclic elliptic curves are not directly supported. This limitation also hinders the use of specialized protocols, requiring the verification algorithm to be EVM-friendly.
2. **Mismatched Field Sizes**: EVM operates with 256-bit integers, while zk proofs naturally work over prime fields. Performing "mismatched field arithmetic" within a circuit necessitates range proofs, which substantially increase the size of the EVM circuit with

approximately 100 constraints per EVM step, causing a two-fold increase in circuit complexity.

3. **Special Opcodes in EVM**: EVM differs from traditional virtual machines with numerous special opcodes like `CALL` and specific error types related to execution context and gas. These idiosyncrasies introduce new challenges in circuit design.
4. **Stack-Based Model**: EVM is a stack-based virtual machine, contrasting with architectures like Cario that utilize a register-based model. These alternatives define their own intermediate representations and employ specialized compilers to translate smart contract code into a zk-friendly IR. This makes it more difficult to prove for the stack-based EVM model and to provide direct support for the native toolchain.
5. **Ethereum Storage Overhead**: Ethereum's storage layout heavily relies on Keccak and the Merkle Patricia Tree (MPT), both of which are not zk-friendly and impose substantial proving overhead. For instance, Keccak hash is 1000 times larger in circuit size compared to Poseidon hash. Replacing Keccak with another hash could create compatibility issues within the existing Ethereum infrastructure.
6. **Machine-Based Proof Overhead**: Even after addressing the aforementioned challenges, efficiently

combining them to create a complete EVM circuit remains a monumental task.

These challenges are being met however the most significant technological advancements stem from the following key areas:

Utilization of Polynomial Commitment Schemes

In recent years, most succinct zero-knowledge proof protocols adhered to R1CS with PCP queries encoded in a trusted setup tailored to specific applications.
This approach often resulted in significant circuit size inflation, limiting opportunities for customized optimizations due to the constraints' degree being restricted to 2 (as bilinear pairings only support one exponentiation). With the adoption of polynomial commitment schemes, constraints can be elevated to higher degrees using a universal or even transparent setup. This newfound flexibility enables a wider range of backend choices.

Introduction of Lookup Table Arguments and Customised Gates:

Another substantial optimisation arises from the use of lookup tables, initially proposed in Arya and further refined in Plookup. This technique offers substantial savings for zk-unfriendly primitives, such as bitwise operations like AND and XOR. Customized gadgets facilitate the efficient handling of high-degree constraints. TurboPlonk and

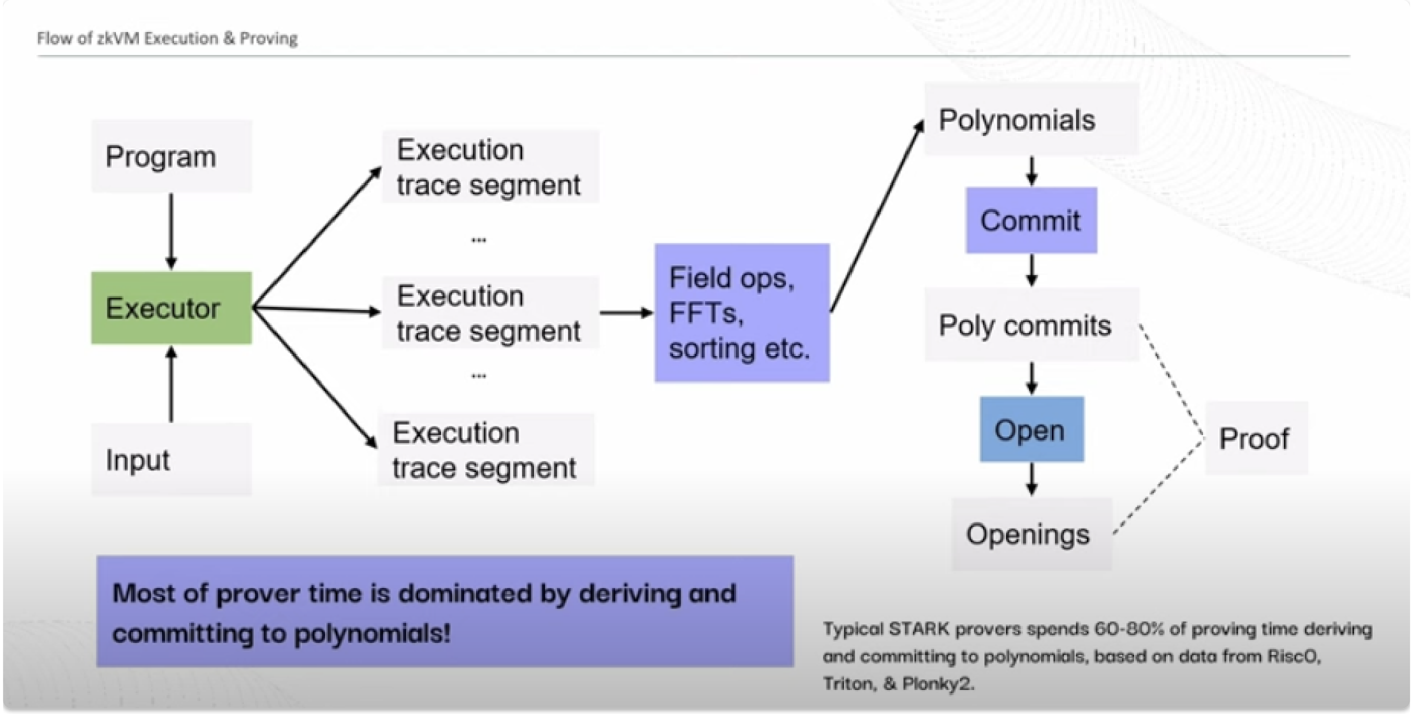
> UltraPlonk introduce elegant program syntax to simplify the use of lookup tables and define customized gadgets. These innovations can significantly reduce the overhead of EVM circuits.
>
> ##### Advancements in Recursive Proofs
>
> Recursive proofs have historically incurred significant overhead due to their reliance on special pairing-friendly cyclic elliptic curves, such as the MNT curve-based construction. This reliance introduced substantial computational costs. However, recent techniques have made recursive proofs feasible without sacrificing efficiency.
>
> ##### Hardware Acceleration Enhances Proving Efficiency
>
> Remarkable progress has been made in hardware acceleration to improve the efficiency of proving. The fastest GPU and ASIC/FPGA accelerators for provers have been developed, including an ASIC prover
> The GPU prover is approximately 5x-10x faster than Filecoin's implementation, significantly enhancing the computational efficiency of provers.

zkProcess in general

The process of creating a proof is one of transformation. We start with a DSL, and from this create a circuit, to which we can add public and private (witness) inputs.
Our goal is to transform this into some mathematical objects that we can reason about.

(Setup)
Arithmetisation
Transform DSL to a circuit
Transform to polynomials
Prover <-> Verifier interaction
Make non-interactive

Here we will focus on the creation and use of circuits. The nature of the arithmetisation process is different for SNARKS and STARKS, today we will concentrate on SNARKS.



Flow of zkVM Execution & Proving

Program

Executor

Input

Execution trace segment

...

Execution trace segment

...

Execution trace segment

Field ops, FFTs, sorting etc.

Polynomials

Commit

Poly commits

Open

Openings

Proof

Most of prover time is dominated by deriving and committing to polynomials!

Typical STARK provers spends 60-80% of proving time deriving and committing to polynomials, based on data from Risc0, Triton, & Plonky2.

> ### Architecture of zkEVM circuits
>
> Overview of zkEVM arithmetisation
>
> Recall that zk circuits are typically composed of general purpose addition and multiplication gates.
> Optimisations are added to this to produce more specific gates for particular tasks.
> We would like to build circuits to produce the constraints for op codes, but many of these op codes are not zk friendly.
> The circuit representation also changes, the above diagram is a useful logical view, but in fact to have these as a data structure we represent the gates as rows in a table.
> The way that we represent our circuit will vary from project to project

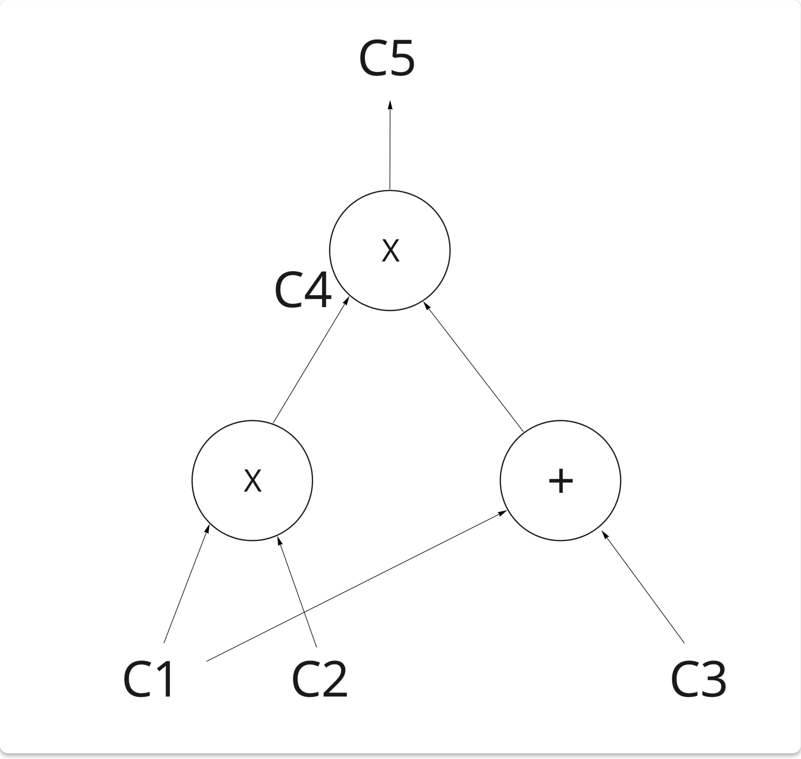
Gates and selectors

Arithmetic Circuits review

In this representation our program is transformed into a (large) number of gates similar to an electrical circuit.
In the simplest form, these are addition or multiplication gates, though as we shall see, we can also have custom gates.

The motivation for this, is that we can represent any algorithm as a series of gates, and further the gates can then represent polynomials, which is our goal.

We could think of a simple circuit as follows, but this is just a useful abstraction, and we actually represent it differently.



It is important to remember that we have moved beyond the idea of an algorithm, when we thinkk of the circuit, we think of it as a whole and we want it to show that the inputs to the circuit satisfy the circuit, that is that the circuit as a whole makes sense.

For example in the above diagram, we are saying that for the circuit to be satisfied we need to find values for C1.. C5 so that

C1.C2 . (C1 +C3) == C5

We have effectively created constraints on the inputs to our circuit.

Having these constraints we can transform these further into polynomials, which then form the basis for our proving system.

If we have a gate, this can represent a constraint, let us specify our gate as having inputs a, and b and output c.

For an addition gate the following must be true if the constraint is satisfied.

$a + b - c = 0$

We prefer to write it as equating to zero, as this helps later.

If we have a number of gates we could index them with i and generalise

a_i+ b_i - c_i = 0

similarly for multiplication gates

$a_i.b_i$ - c_i = 0

Better yet would be a common way to represent both gates, for this we use selectors S

In a table this would be

	a	b	c	S
1	a_1	b_1	c_1	1
2	a_2	b_2	c_2	1
3	a_3	b_3	c_3	0

here S is set to 1 for an addition gate and zero for a multiplication gate, this gives a common equation for all rows in our table as

$$S_i(a_i + b_i) + (1 - S_i)a_i.b_i - c_i = 0$$

In PLONK these are called the *gate constraints* because they refer to the equalities for a particular gate.

We can also have *copy constraints* where we have a relationship between values that are not on the same gate, for example it may be the case that

a_7 = b_5 for a particular circuit, in fact this is how we link the gates together.

The use of a selector is a common technique within zk circuits and its selecting process adds

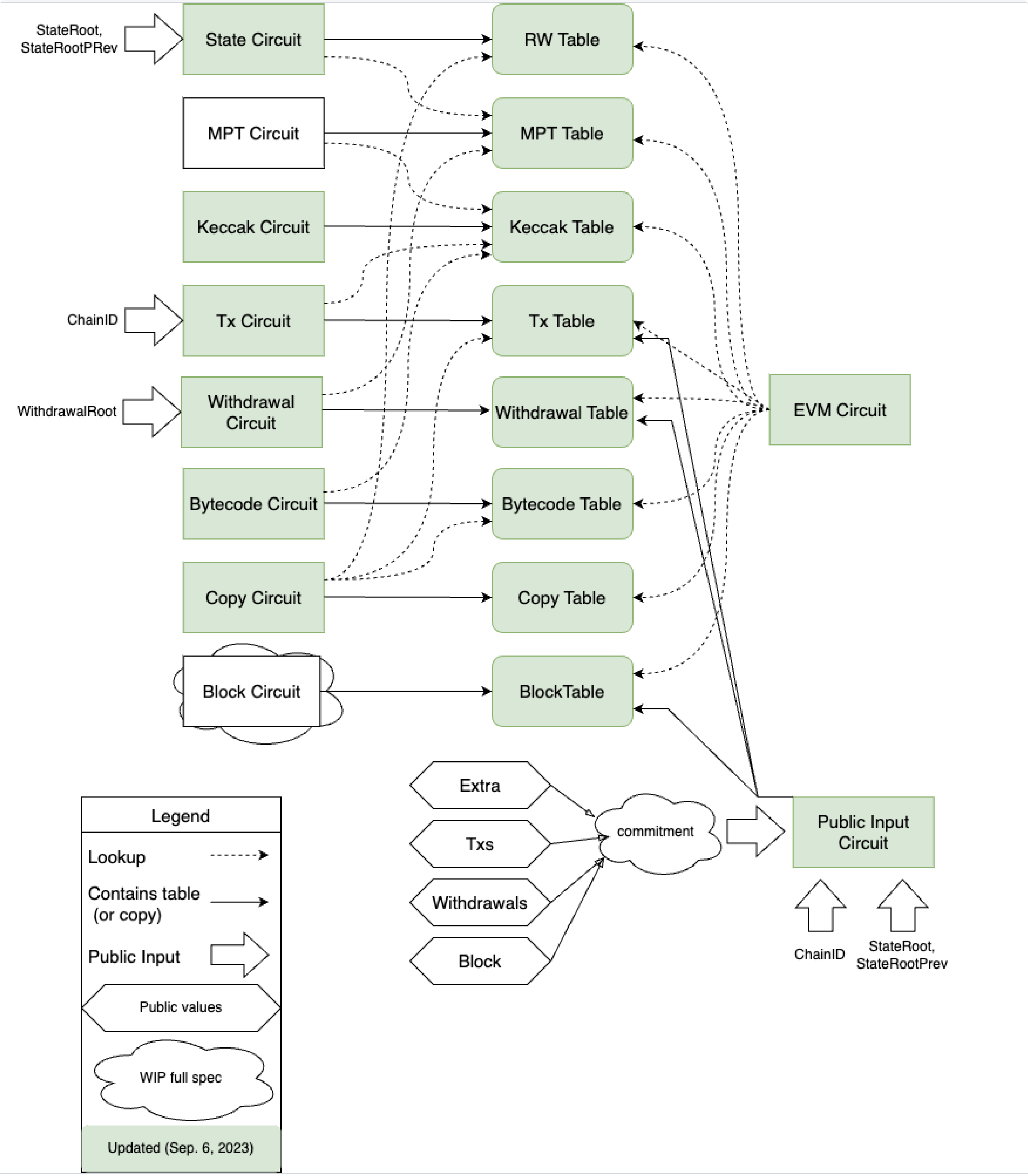
an additional constraint that if we add all the S values together for a row (which will represent many gates) then the total of S can be at most 1, since we are selecting only one item.

> **Custom gates**
>
> Another optimisation is to use specially crafted custom gates to perform operations that would not be optimal with generic gates.

> ### Lookup tables
>
> Many operations are not optimally represented by simple gates, for example bitwise operations or small range checks.
>
> Rather than ask the compiler to produce general purpose gates to represent these, we can create lookup tables and prepopulate them with values. Our proof then becomes a matter of showing that the requisite value exists in the table.
>
> For example if we want to check that a value x is between 1 and 10, it may be more optimal to populate a table with values 1 to 10 and then show that our x value is in that table.
>
> A number of schemes to do this process efficiently have been developed including Plookup and Caulk (See Paper, zk study club Video, Slides)
>
> We are not able to prepopulate values for a hash function (this would be equivalent to a brute force) so we need a keccak circuit to check the values.

Combining the component circuits

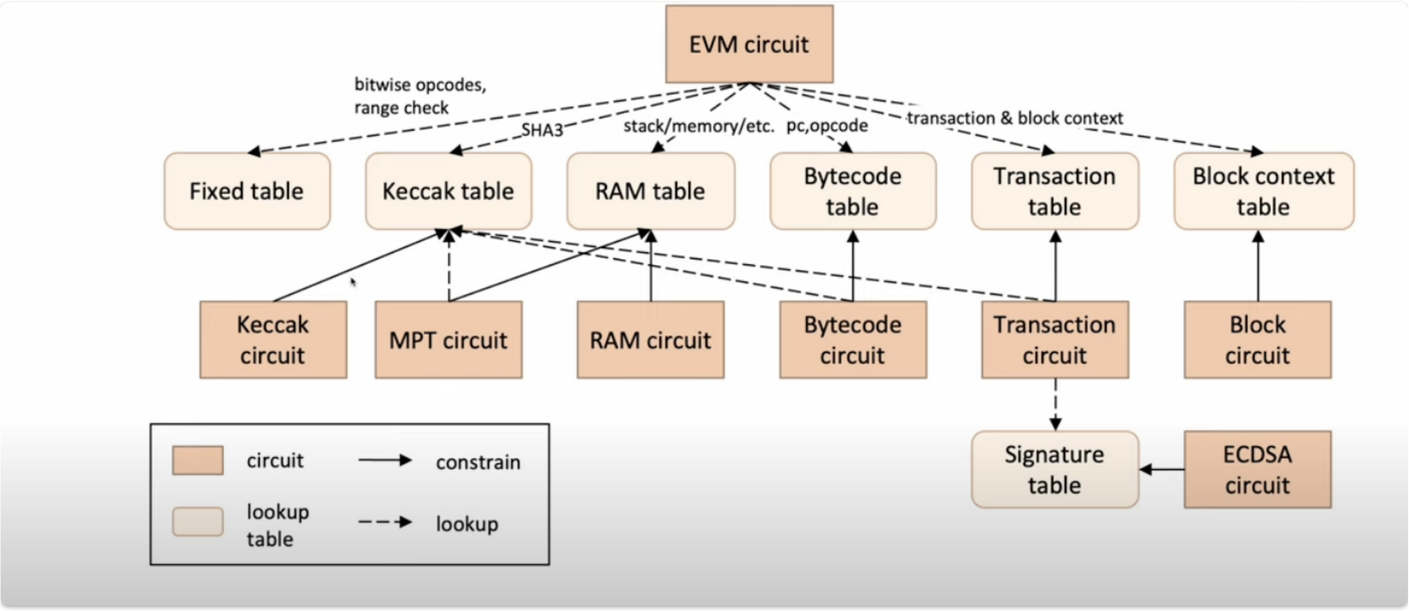
We will need circuits for the various components and also for their interaction.
The PSE group show the following 'super circuit'



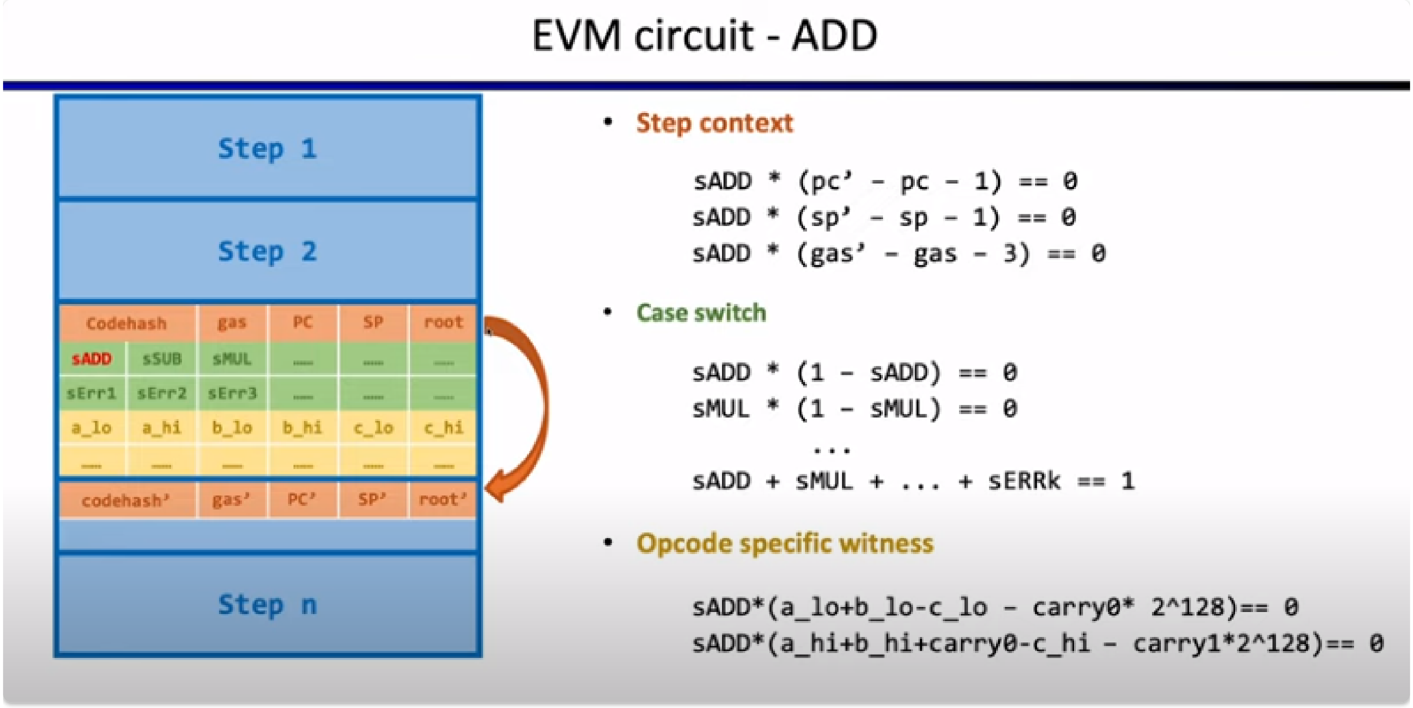
> made of the following circuits
>
> EVM Circuit
>
> State Circuit
>
> MPT Circuit
>
> Keccak Circuit
>
> Tx Circuit
>
> Bytecode Circuit
>
> Copy Circuit
>
> Block Circuit
>
> PublicInputs Circuit
>
> Withdrawal Circuit
>
> and the following tables
>
> RW Table
>
> MPT Table
>
> Keccak Table
>
> Tx Table
>
> Bytecode Table
>
> Copy Table
>
> Block Table
>
> Withdrawal Table
>
> We require read / write tables to check that items that have been written to say storage are being read correctly.

Compiling a simple add operation

Imagine we have a large table in which we can hold relevant variables , maybe from storage, memory etc.
In total there would be many tables and circuits needed, this diagram from Scroll (from video about arithmetisation) gives an indication of what is involved



Taking some of these circuits and tables



EVM circuit - ADD

Step 1

Step 2

Codehash		gas	PC	SP	root
sADD	sSUB	sMUL	—	—	—
sErr1	sErr2	sErr3	—	—	—
a_lo	a_hi	b_lo	b_hi	c_lo	c_hi
—	—	—	—	—	—
codehash'		gas'	PC'	SP'	root'

Step n

- Step context

```
sADD * (pc' - pc - 1) == 0
sADD * (sp' - sp - 1) == 0
sADD * (gas' - gas - 3) == 0
```

- Case switch

```
sADD * (1 - sADD) == 0
sMUL * (1 - sMUL) == 0
            ...
sADD + sMUL + ... + sERRk == 1
```

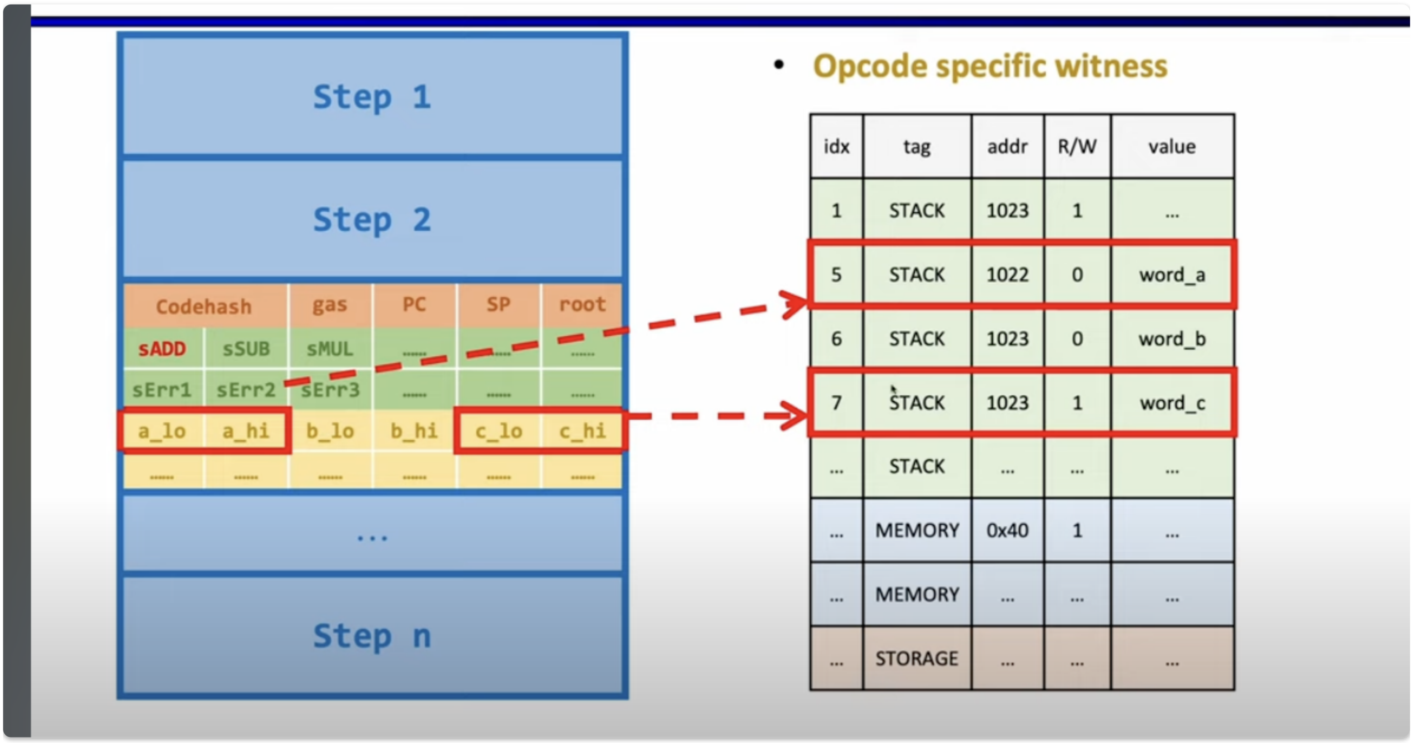
- Opcode specific witness

```
sADD*(a_lo+b_lo-c_lo - carry0* 2^128)== 0
sADD*(a_hi+b_hi+carry0-c_hi - carry1*2^128)== 0
```

Looking at the constraints, we see items concerning

- the program counter
- the gas counter
- the selector for the type of operation (ADD or MUL)
- range checks for high and low parts of representation of a 256 bit number.

We can also have similar items to check the operation of the stack



Step 1

Step 2

Codehash		gas	PC	SP	root
sADD	sSUB	sMUL	......	......	......
sErr1	sErr2	sErr3	......	......	......
a_lo	a_hi	b_lo	b_hi	c_lo	c_hi
......	......	......	......	......	......

...

Step n

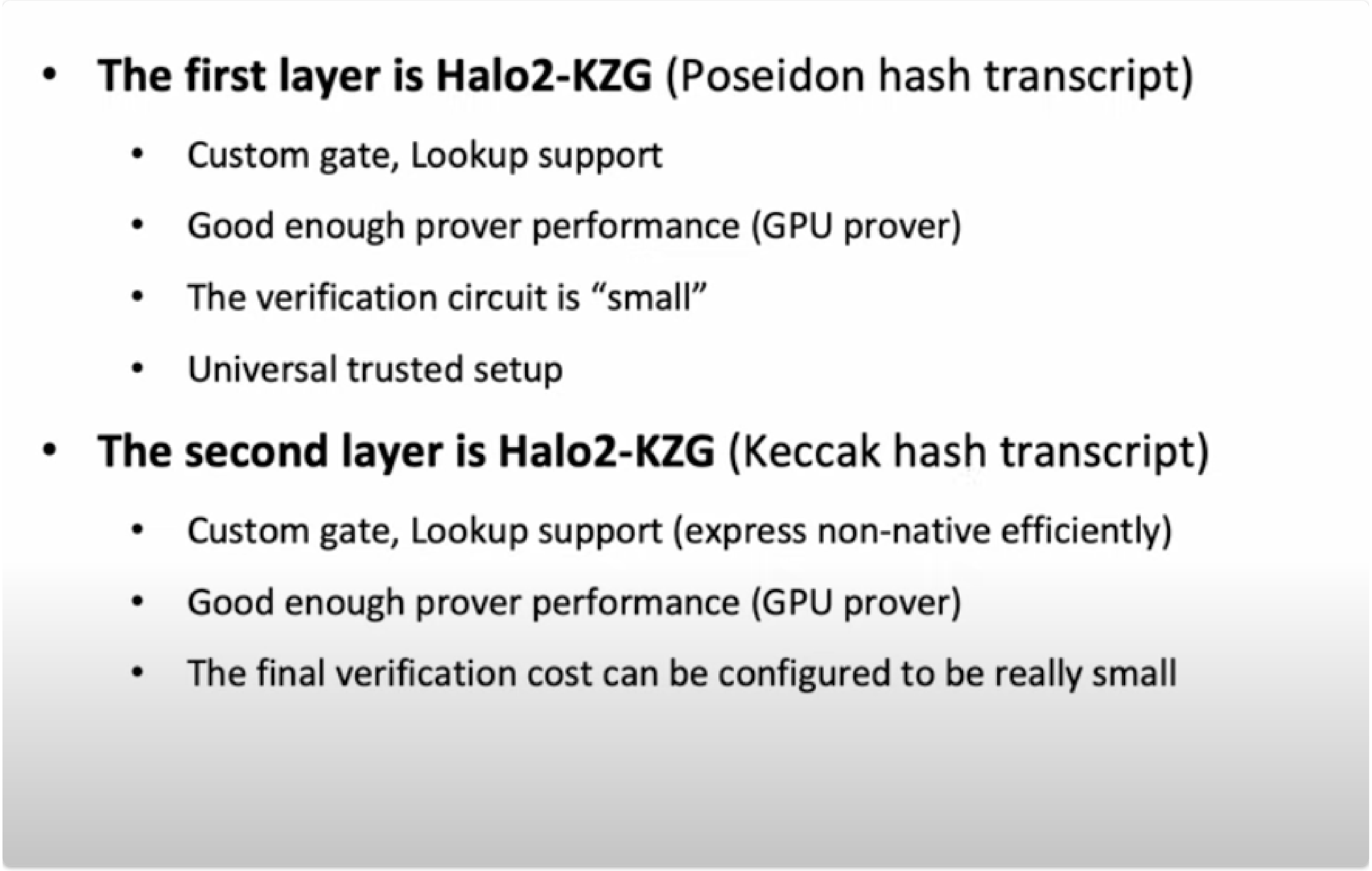
- Opcode specific witness

idx	tag	addr	R/W	value
1	STACK	1023	1	...
5	STACK	1022	0	word_a
6	STACK	1023	0	word_b
7	STACK	1023	1	word_c
...	STACK	...	...	...
...	MEMORY	0x40	1	...
...	MEMORY	...	...	...
...	STORAGE	...	...	...

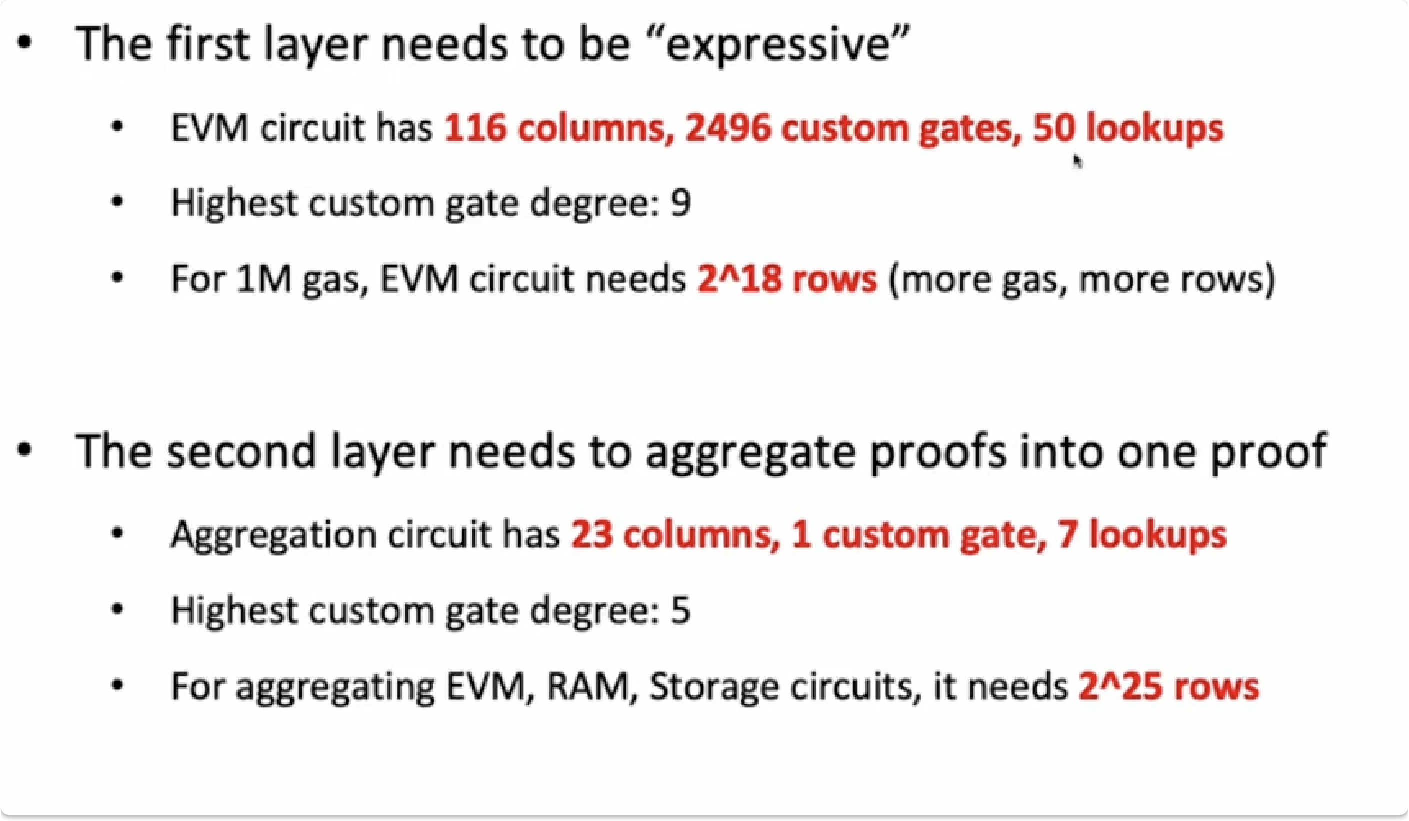
Further techniques

We may chose different proving systems for their relevant optimisations, and be able to combine them in a way to make the most of each's strengths.

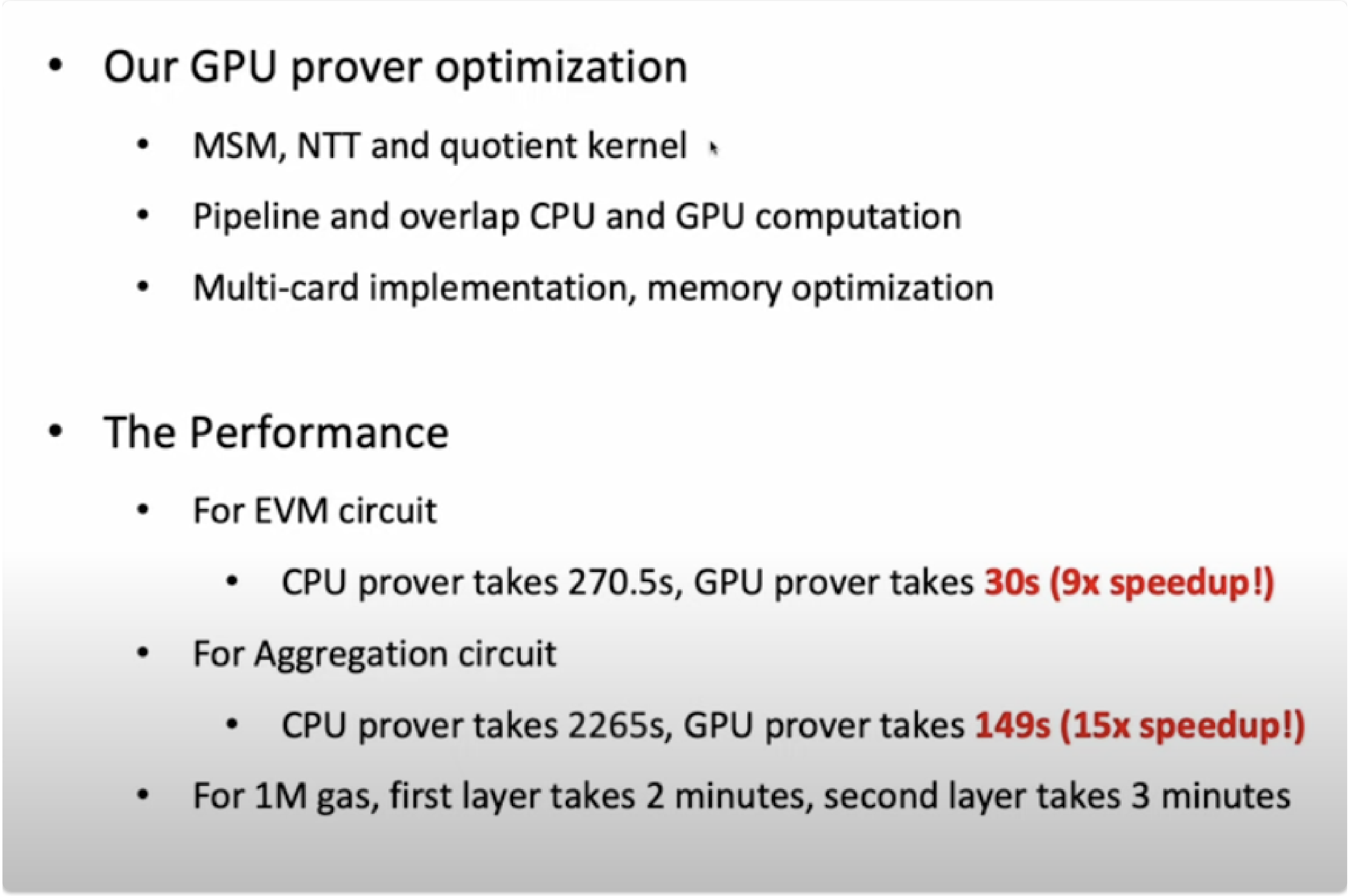
From Scroll



- **The first layer is Halo2-KZG** (Poseidon hash transcript)
  - Custom gate, Lookup support
  - Good enough prover performance (GPU prover)
  - The verification circuit is “small”
  - Universal trusted setup
- **The second layer is Halo2-KZG** (Keccak hash transcript)
  - Custom gate, Lookup support (express non-native efficiently)
  - Good enough prover performance (GPU prover)
  - The final verification cost can be configured to be really small



- The first layer needs to be “expressive”
  - EVM circuit has **116 columns, 2496 custom gates, 50 lookups**
  - Highest custom gate degree: 9
  - For 1M gas, EVM circuit needs **2^18 rows** (more gas, more rows)
- The second layer needs to aggregate proofs into one proof
  - Aggregation circuit has **23 columns, 1 custom gate, 7 lookups**
  - Highest custom gate degree: 5
  - For aggregating EVM, RAM, Storage circuits, it needs **2^25 rows**



- Our GPU prover optimization
  - MSM, NTT and quotient kernel
  - Pipeline and overlap CPU and GPU computation
  - Multi-card implementation, memory optimization
- The Performance
  - For EVM circuit
    - CPU prover takes 270.5s, GPU prover takes **30s (9x speedup!)**
  - For Aggregation circuit
    - CPU prover takes 2265s, GPU prover takes **149s (15x speedup!)**
  - For 1M gas, first layer takes 2 minutes, second layer takes 3 minutes

> **Scroll zkEVM Design**
>
> See Video

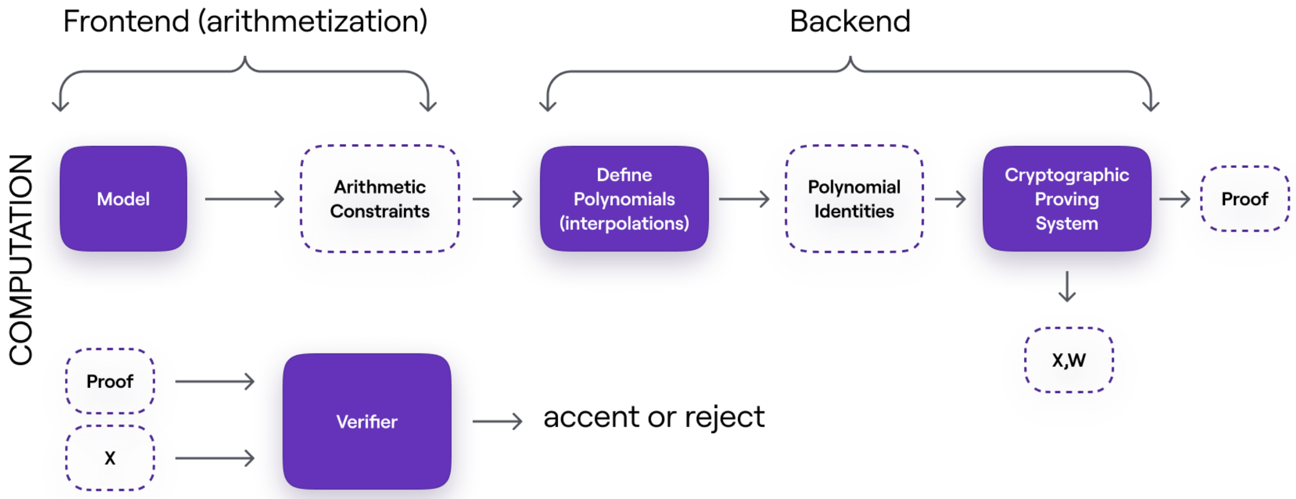
---

zkEVM Design

From Polygon documentation

The general approach to designing the zkProver so as to realise a proof system based on State Machines is as follows:

1. Turn the required deterministic computation into a state machine computation.
2. Describe state transitions in terms of algebraic constraints. These are like rules that every state transition must satisfy.
3. Use interpolation of state values to build polynomials that describe the state machine.
4. Define polynomial identities that all state values must satisfy.
5. A specially designed cryptographic proving system (e.g. a STARK, a SNARK, or a combination of the two) is used to produce a verifiable proof, which anyone can verify.


Frontend (arithmetization)
Backend
COMPUTATION
Model
Arithmetic Constraints
Define Polynomials (interpolations)
Polynomial Identities
Cryptographic Proving System
Proof
X,W
Proof
X
Verifier
accent or reject

> **zkEVMBenchmarks**
>
> Benchmarks from Celer
>
> Miden and Risc Zero benchmarks

Under constrained code

When we write code in our DSL we need to ensure that the assertions or constraints within our code is sufficient to completely test the claim that the prover is making.
If we have under constrained code this could lead to security problems, in that a proof is accepted as showing a particular witness is known for example, but the DSL is not sufficiently testing this.

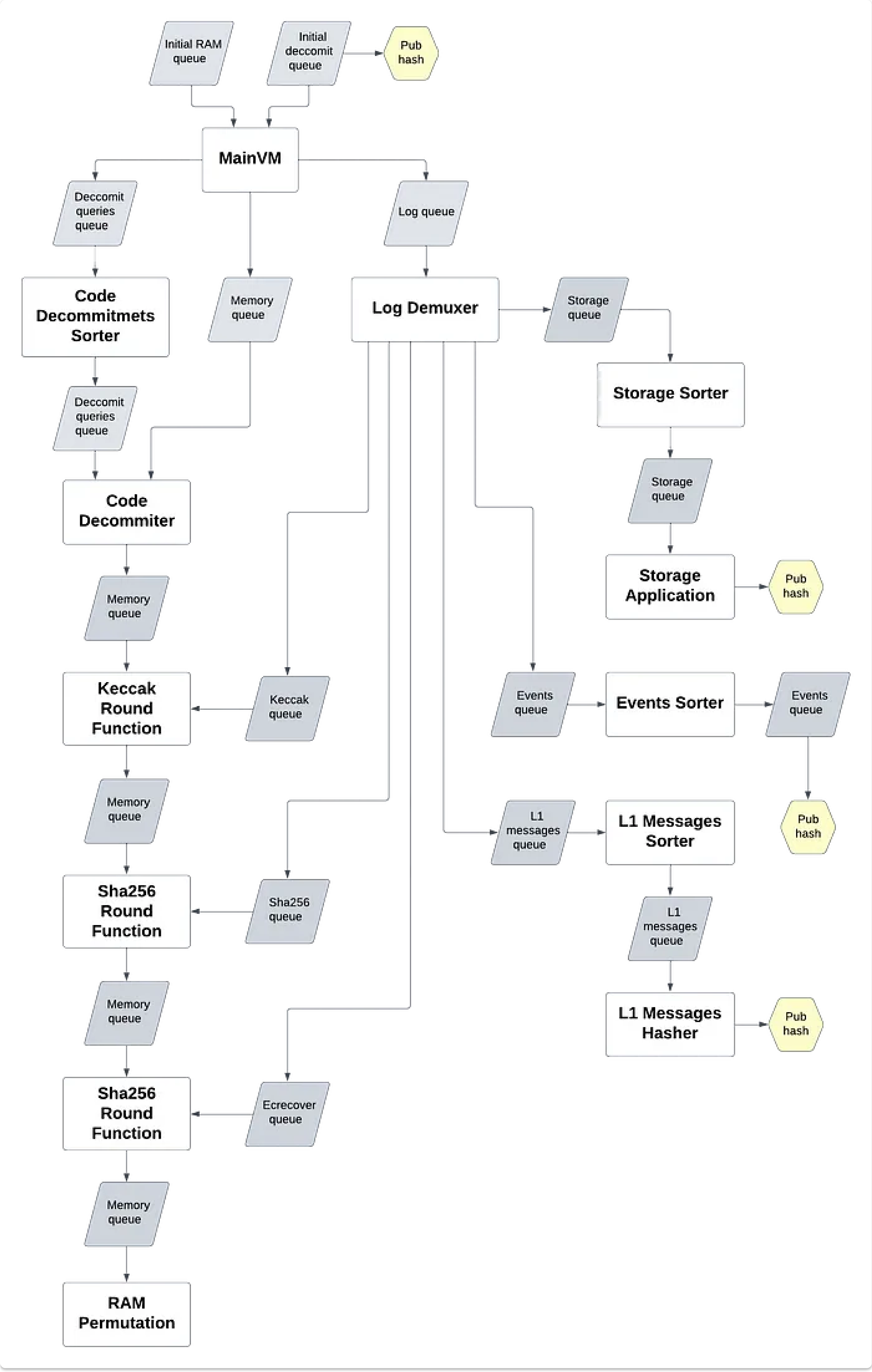
We also need to be careful with arithmetic, since we use modular arithmetic over finite fields, there could be multiple values that would satisfy a constraint, of which we are not aware.

Examples

See write up of zkSync
See POC for zkSync
See dYdX bug

A example of this was found in zkSync Era

Architecture of the VM


Initial RAM queue
Initial deccomit queue
Pub hash
MainVM
Deccomit queries queue
Log queue
Code Decommitmets Sorter
Memory queue
Log Demuxer
Storage queue
Storage Sorter
Deccomit queries queue
Storage queue
Code Decommiter
Storage Application
Pub hash
Memory queue
Keccak Round Function
Keccak queue
Events queue
Events Sorter
Events queue
Memory queue
L1 messages queue
L1 Messages Sorter
Pub hash
Sha256 Round Function
Sha256 queue
L1 messages queue
Memory queue
L1 Messages Hasher
Pub hash
Sha256 Round Function
Ecrecover queue
Memory queue
RAM Permutation

> The problem was in the Memory Queue.
> It was found that a constraint was missing, this constraint was being used to enforce that the upper 128 bits of a memory variable would be zero.