ZKVM 101

INGONYAMA

Outline

1. Introduction: ZKP and frontends ASIC vs CPU approach
2. ASIC Approach toy example R1CS
3. RAM computational model
4. CPU approach: (toy?) example TinyRAM
5. ZKEVM $\subset$ ZKVM
6. Aspects of ZKVM design

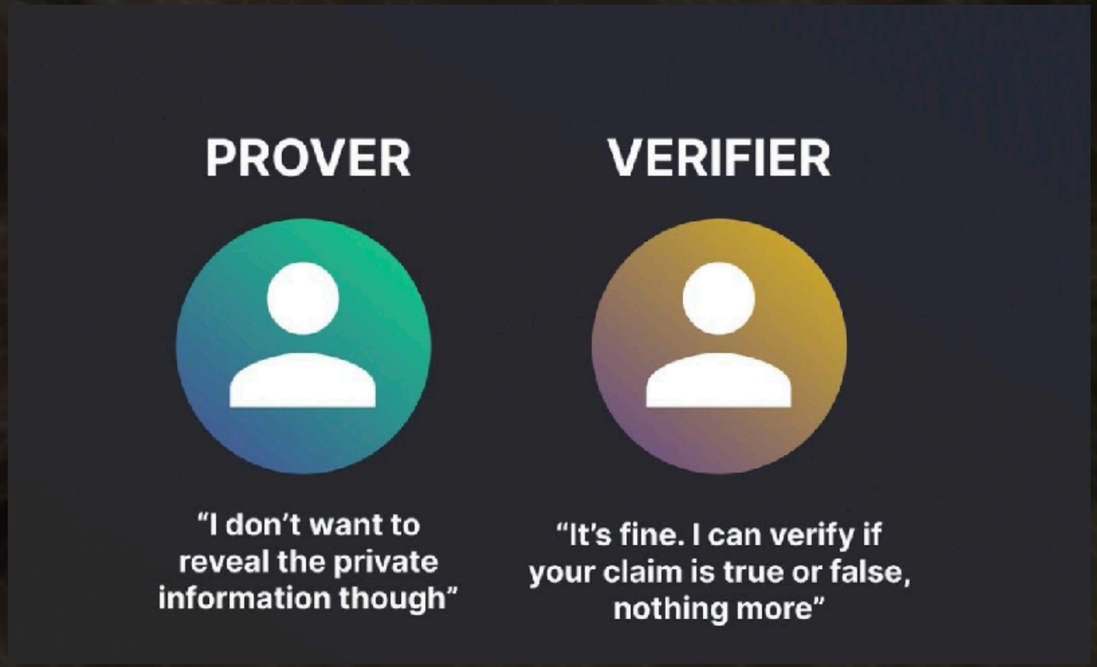
1. Introduction

ZKP in a 

- ZKP (Zero Knowledge Proofs) are cryptographic protocols that enable one party (prover) to convince another party (verifier) about the truth of a statement without revealing any private information (witness)

Public: y, x

Witness: w



Public: y, x

0/1

$$y \stackrel{?}{=} C(x, w)$$

Statement

Eagle's view: overarching picture

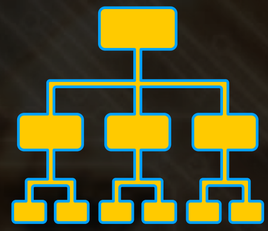
Application Logic

Public: Intermediate representation

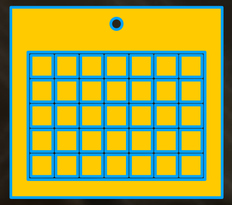
Frontend: Generate private trace

Backend: Cryptographic Proof

Eg: Block chain smart contracts



Eg: Merkle tree of hashes of user id



Execution Trace table

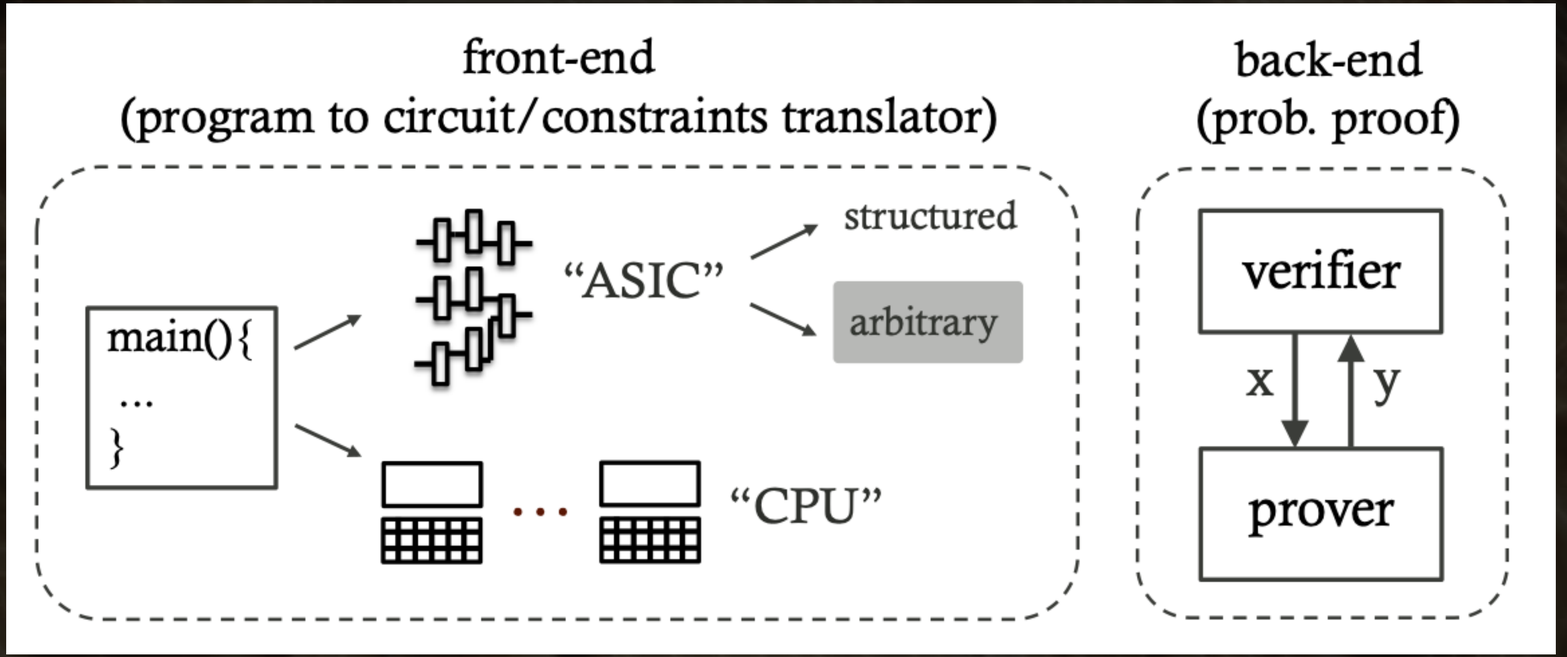
Arithmetization - Encode application logic as conditions on application specific trace table memory registers

Witness generation: Run Arithmetized application and fill the table

Prove: I filled the table as per the protocol specified.

Hence, The values I filled in the table, satisfy the constraints prescribed by the application logic

Frontends: ASIC vs CPU approach



Walfish et al,
BIU13 school

TABLE 3
Front end paradigms ASIC vs CPU approach

Feature	ASIC approach	CPU approach
Program logic	deterministic	non-deterministic
Circuit encoding	hard coded arithmetic circuits	Compile to virtual Machine (VM) Instruction Set Architecture (ISA)
Advantages	Efficiency community templates for circuits Optimizations	Flexiblity Program in high level languages less cryptography knowledge
Disadvantages	Low level languages for circuit Large surface area for bugs	difficult to optimize rely on compilers for ISA
Backend function	prove CSAT	prove ISA execution **

polyAPI paper

2. ASIC approach: toy example

Front end compiler: ASIC approach

R1CS (Rank One Constraint System)

```
// Define declares the circuit logic. The compiler then produces a list of constraints
// which must be satisfied (valid witness) in order to create a valid zk-SNARK
func (circuit *Circuit) Define(api frontend.API) error {
    // compute x**3 and store it in the local variable x3.
    x3 := api.Mul(circuit.X, circuit.X, circuit.X)

    // compute x**3 + x + 5 and store it in the local variable res
    res := api.Add(x3, circuit.X, 5)

    // assert that the statement x**3 + x + 5 == y is true.
    api.AssertIsEqual(circuit.Y, res)
    return nil
}
-- witness.json --
{
    "x": 3,
    "Y": 35
}
```

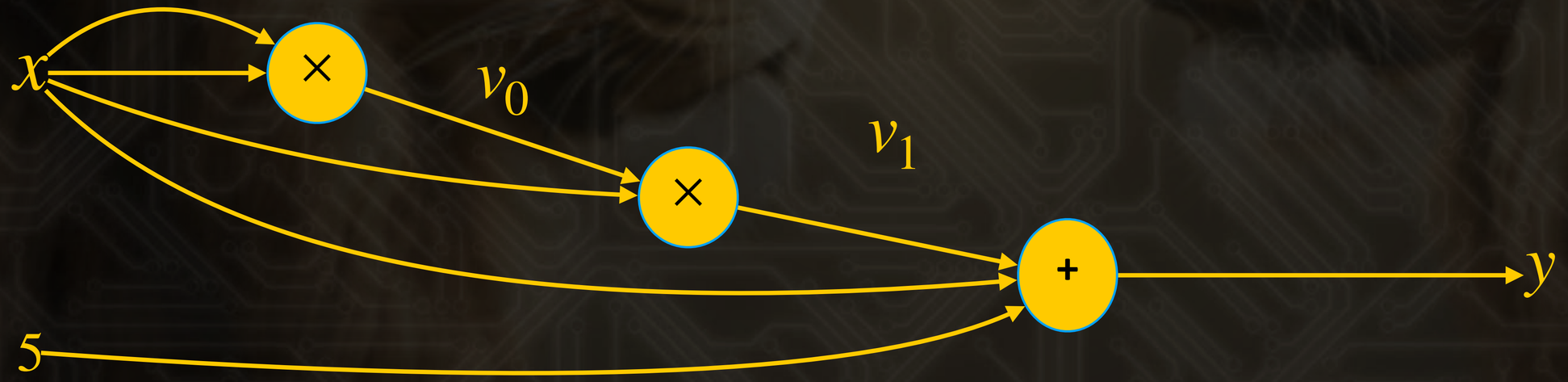
Witness

Frontend: trace

$L \cdot R == 0$

L	R	O
x	x	v0
v0	x	v1
1	Y	5 + x + v1

Gnark R1CS tool



Public: Intermediate representation

$$x^3 + x + 5 = y$$

Front end compiler: ASIC approach

Hard coded constraints of trace table

$$x \cdot x = v_0$$
$$v_0 \cdot x = v_1$$
$$1 \cdot y = 5 + x + v_1$$

Witness vector: $z = [pub, priv, gen]$

Trace table satisfies R1CS constraints equation

$$(A \,.\, z) \circ (B \,.\, z) = (C \,.\, z)$$

		L·R == O
L	**R**	**O**
x	x	v0
v0	x	v1
1	Y	5 + x + v1

Gnark R1CS tool

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

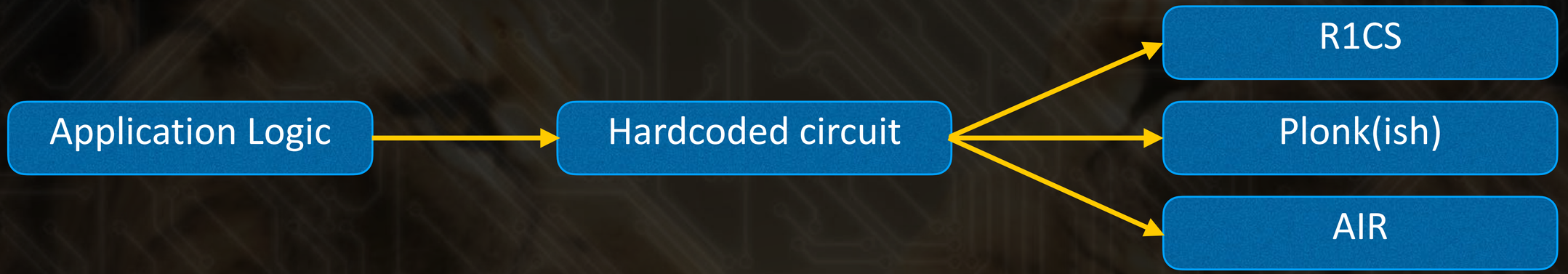
$$z = [1,5,y, x, v_0, v_1]$$

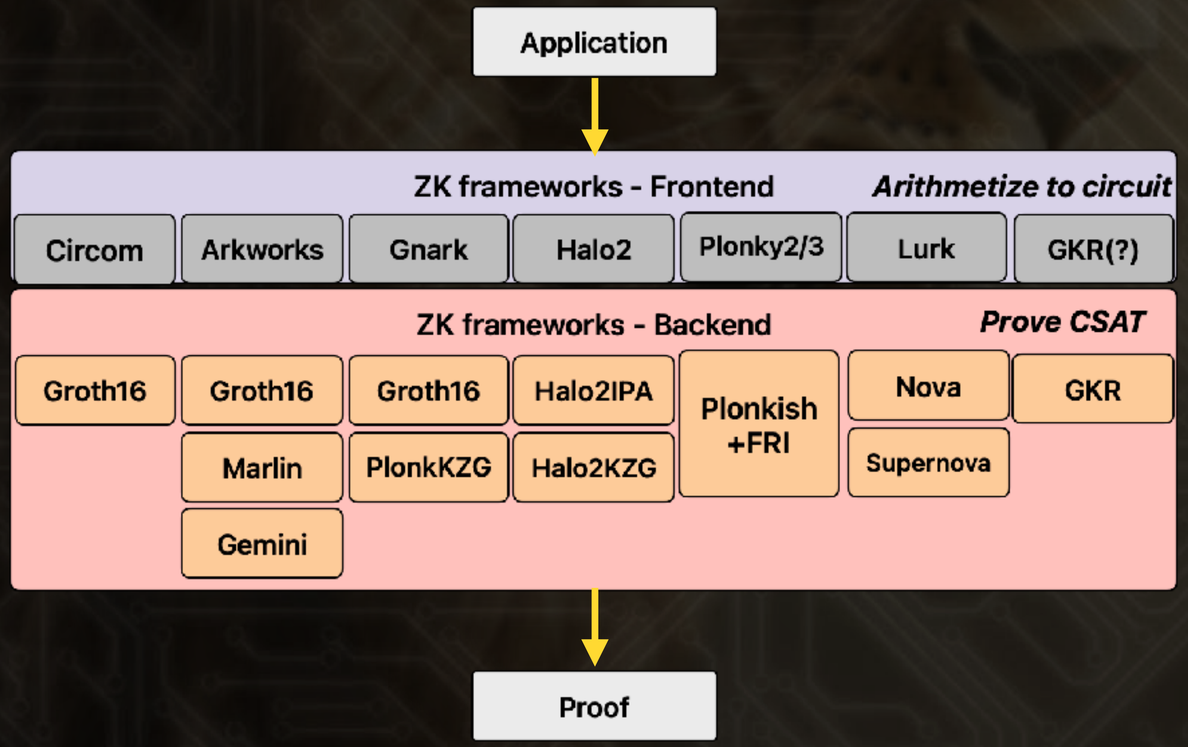
Problem: Needs to be hardcoded, source of 90% bugs in ZKP systems

Problem: needs to be done per program, significant developer effort

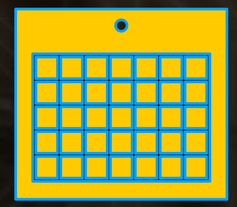
Front end compiler: ASIC approach

Constraint system





Execution Trace table



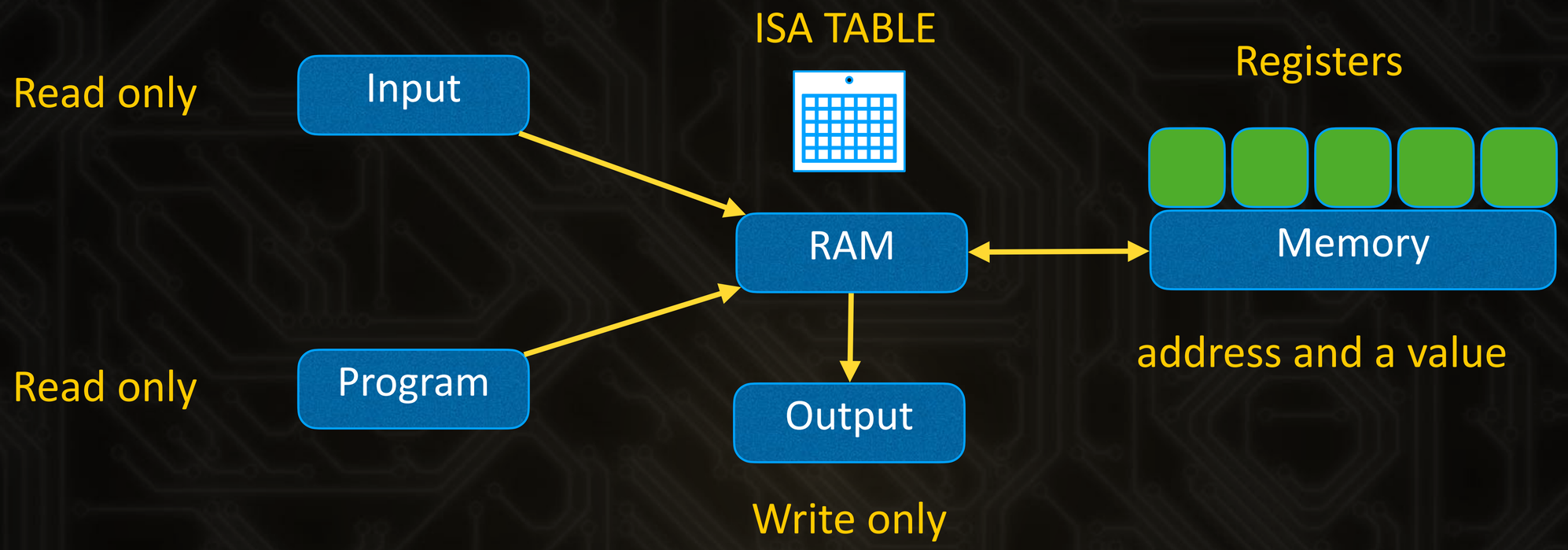
Frontend	Supported Backends
Arkworks [22] (R1CS)	Groth16 [4], Marlin [31], Gemini [32]
Circom [33] (R1CS)	Groth16
Gnark [11] (R1CS/Plonk)	Groth16, Plonk [5] +KZG [13]
Halo2 (Plonkish)	Halo2 IPA , Halo2 KZG [6]
Plonky2/3 [7], [8] (Plonkish)	Plonkish + STARK [9]
ZoKrates [34]	Groth16, Marlin

PolyAPI paper

3. RAM computational model

Theory - RAM computation model

RAM - Random Access Machine is a computational model of a CPU

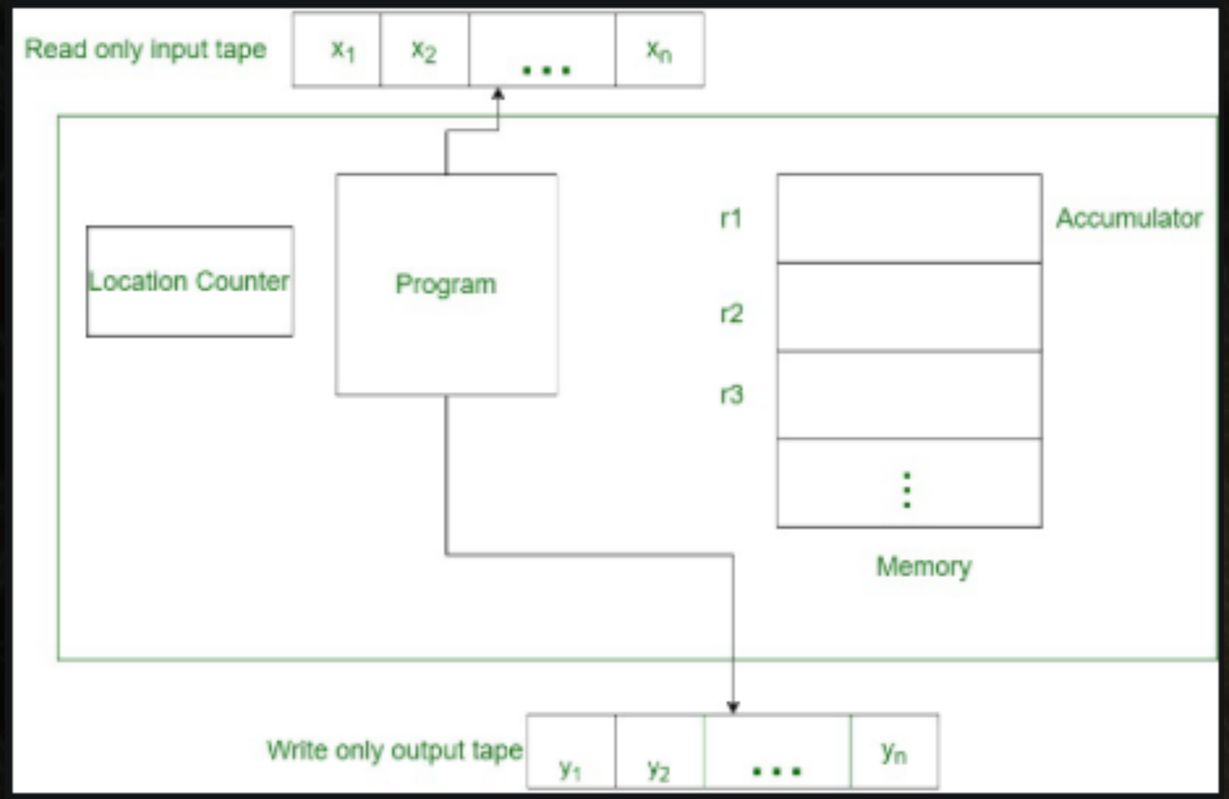


“Simple” operations take one 1 time step

Loops count as often as they run

“Memory” access is free, assumed to be infinite.

RAM computation model - Formal representation



Read only input tape, write only output tape

HV: Program not stored on memory,
VN: On memory

For each read, pointer moves one step

After each write pointer moves one step.

Registers (scratch) is assumed infinite

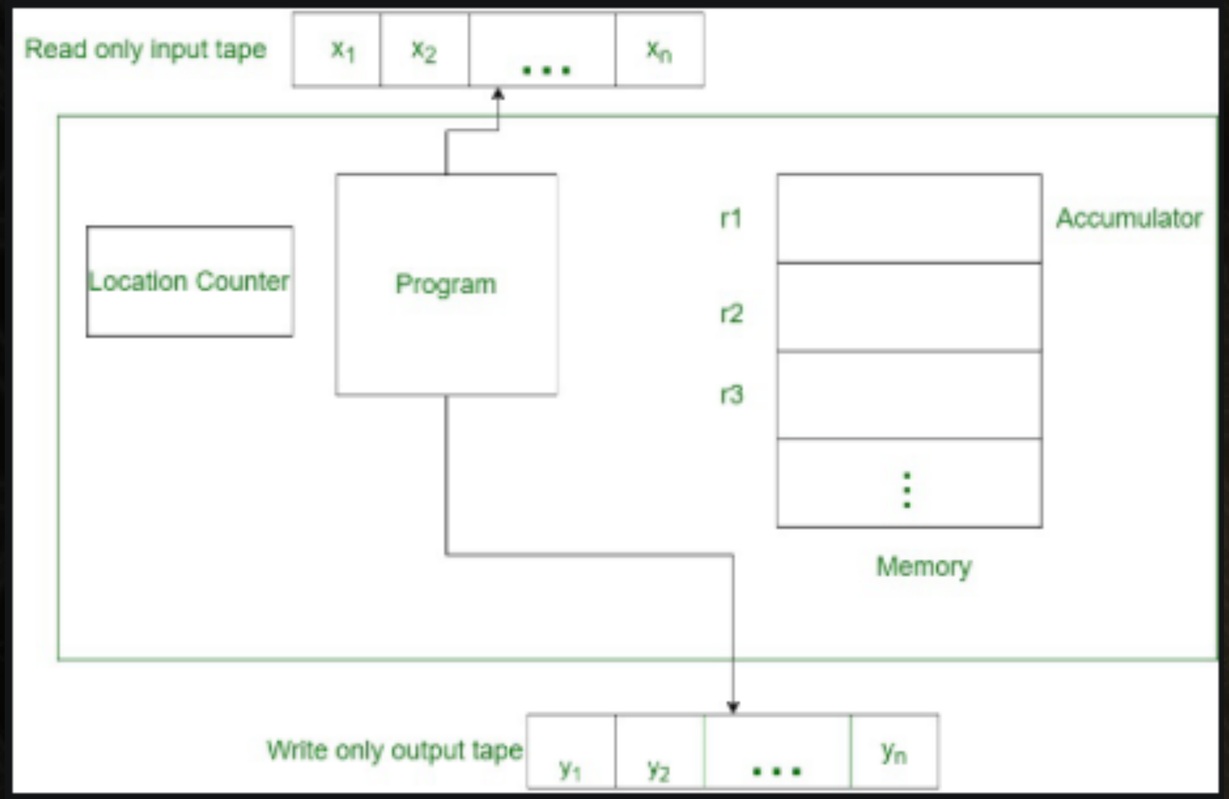
RAM has a defined dictionary of Instructions

In RAM, it is assumed that every program is compiled from this dictionary

Functionality: RAM uses a given instruction from the ISA set and derives a target register address from the instruction itself or the content of the register specified in the instruction.

RAM computation model - Formal representation

Register-to-register ("read-modify-write") model of Cook and Reckhow (1973)



$LOAD(C, r_d)$; $C \rightarrow r_d$

$ADD(r_{s_1}, r_{s_2}, r_d)$; $[r_{s_1}] + [r_{s_2}] \rightarrow r_d$

$SUB(r_{s_1}, r_{s_2}, r_d)$; $[r_{s_1}] - [r_{s_2}] \rightarrow r_d$

$COPY(d, r_s, i, r_p)$; $[r_s] \rightarrow [r_p]$

$JNZ(r, I_z)$ $\quad$ $If\ [r] > 0\ jump\ to\ inst\ I_z$ $else\ continue$

$READ(r_d)$ $\quad$ Read from input tape

$PRINT(r_d)$ $\quad$ Copy to output tape

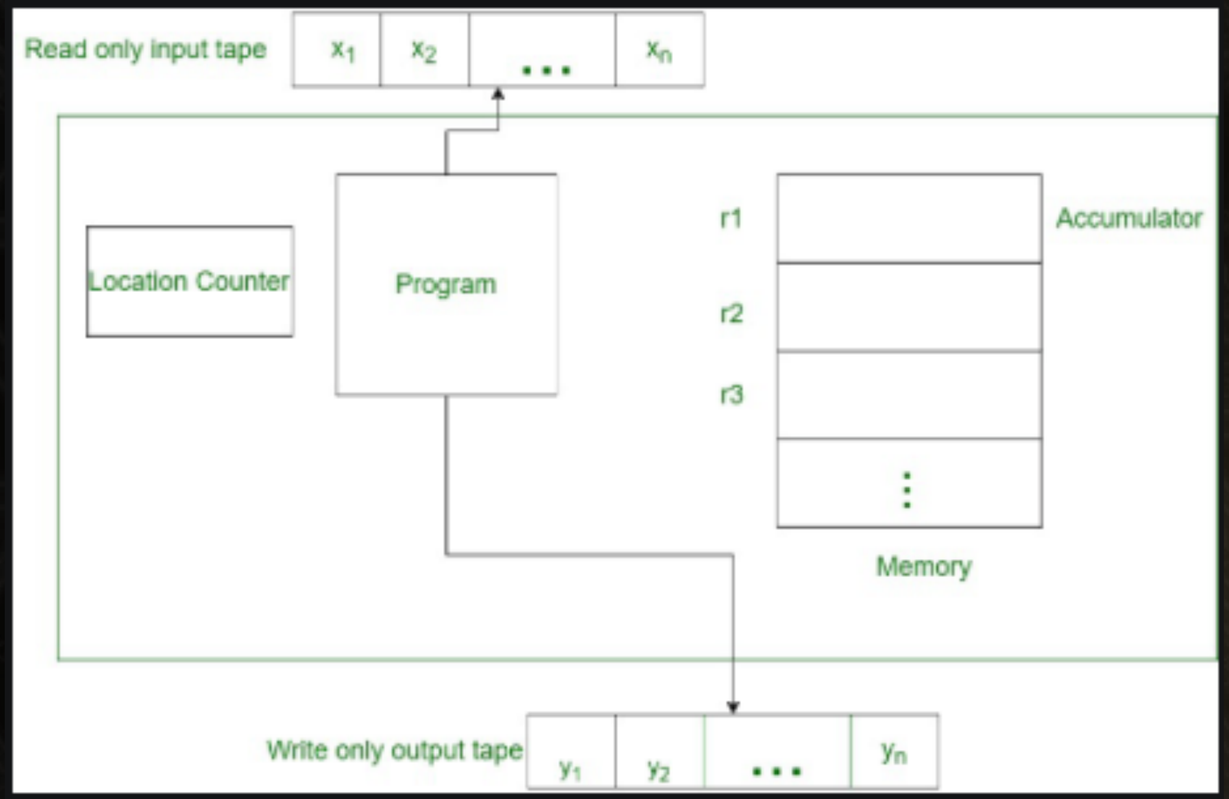
$LOAD(0,3)$ $\quad$ Clear reg3

$SUB(3,3,3)$ $\quad$ Clear reg3

$ADD(3,3,3)$ $\quad$ Double reg 3

RAM computation model - Provability

At each instant of time T, the RAM has a state, specified by its values in the registers



An execution trace is a sequence of RAM states

A valid trace if each step follows from the rules of RAM

Eg: registers do not change value if no opcode is executed

What we need to prove?

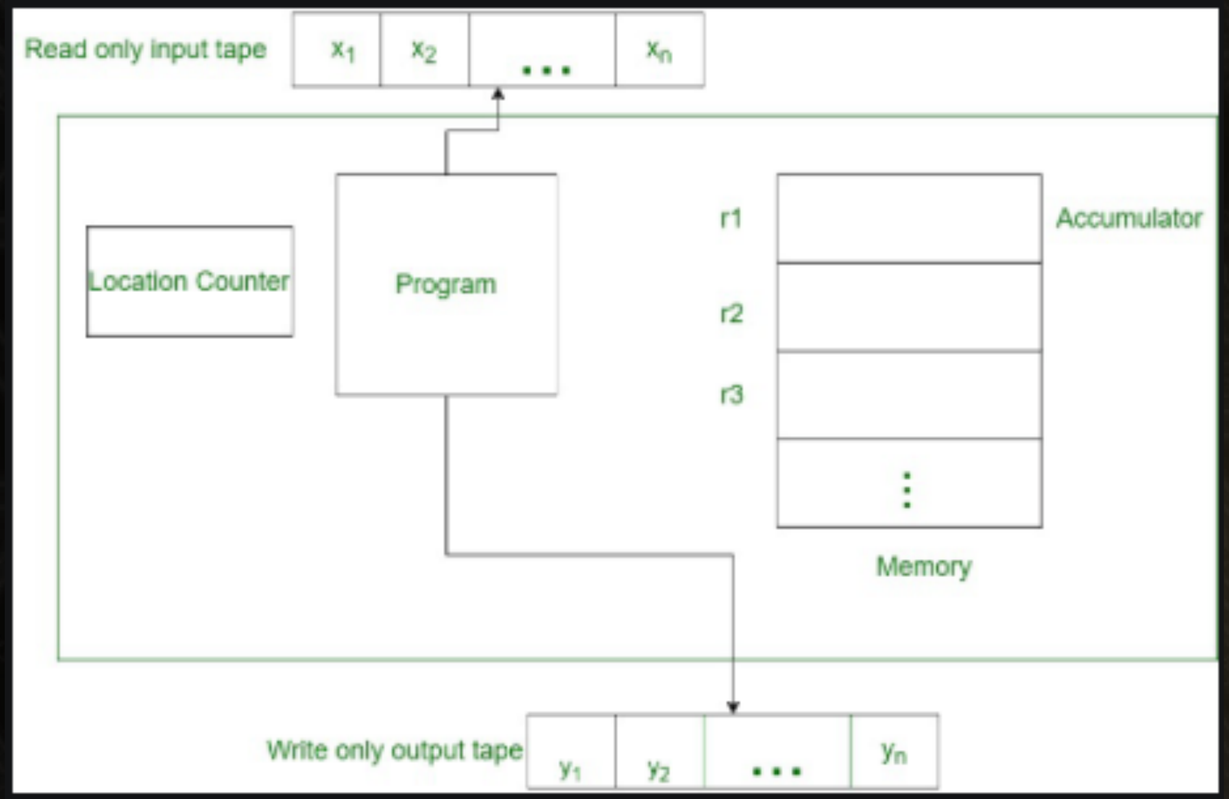
Prove that: a fetched instruction was executed as per RAM model.

Prove that :At each time step, the correct instruction was fetched from memory (VN) or from program (Hv)

Prove that: each load from memory retrieves the last value stored there.

RAM computation model - Provability

We can imagine a circuit that does all these checks



$$ADD(r_{s_1}, r_{s_2}, r_d) \;;\; [r_{s_1}] + [r_{s_2}] \rightarrow r_d$$

Can do with addition gates

What about multiplication?

Ok can unroll it into many additions - inefficient

What about division modulo?

One option is to try and implement, as “software” above the RAM level but again will be unrolled into huge numbers of opcodes

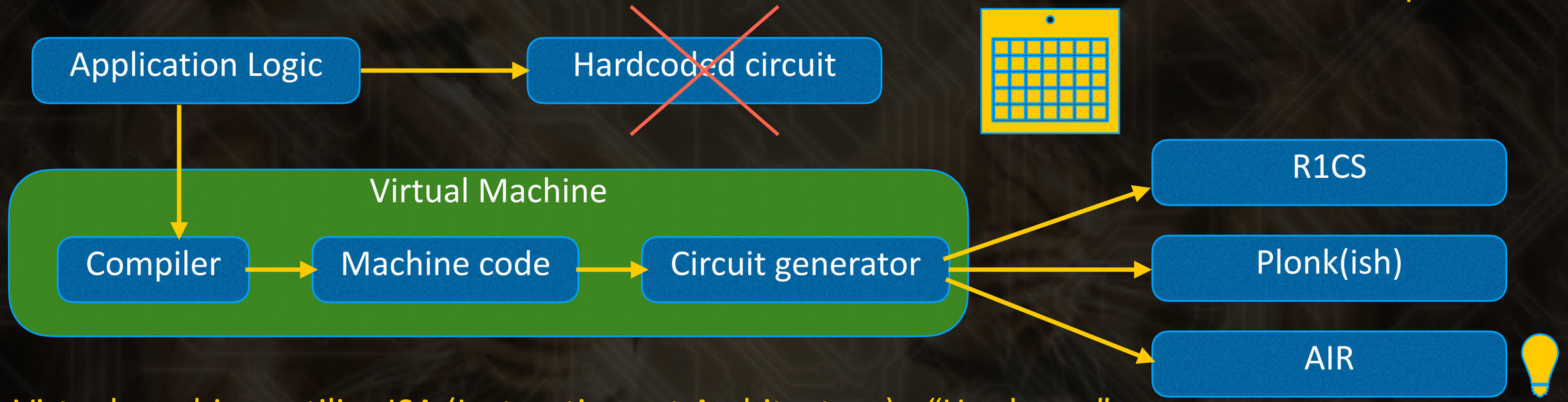
Generally speaking, one needs to identify a broad class of ISA that will solve the problem you are interested in, eg: what is the minimum ISA needed for efficient ZK computation?

4. CPU (VM) approach: tinyRAM

Front end compiler: CPU approach - VM

Replaces hardcoded circuits and wiring with “virtual machines” or CPU

Constraint system needs to be defined apriori!

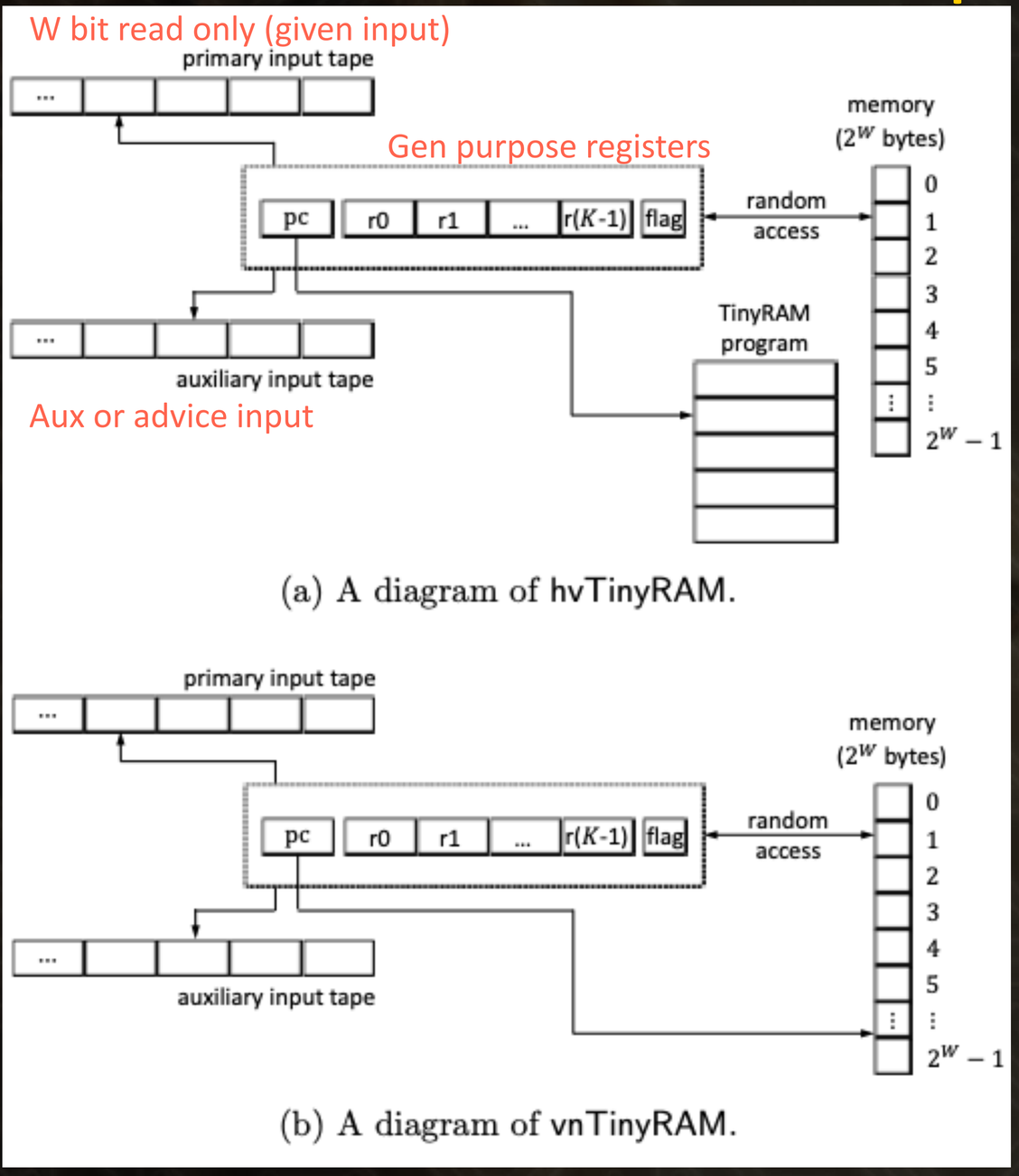


Virtual machines utilize ISA (Instruction set Architecture) - “Hardware”

Any program logic directly supported by ISA is directly expressed in this “Hardware language” is efficient

Any program logic not directly supported by ISA is written in “software” - additional machine code, and computational overhead

Front end compiler: eg VM - TinyRAM



(a) A diagram of hvTinyRAM.

(b) A diagram of vnTinyRAM.

Word size W= 8 | 2^k

PC - program counter - W bits

K gen purpose registers

1 bit flag (condition)

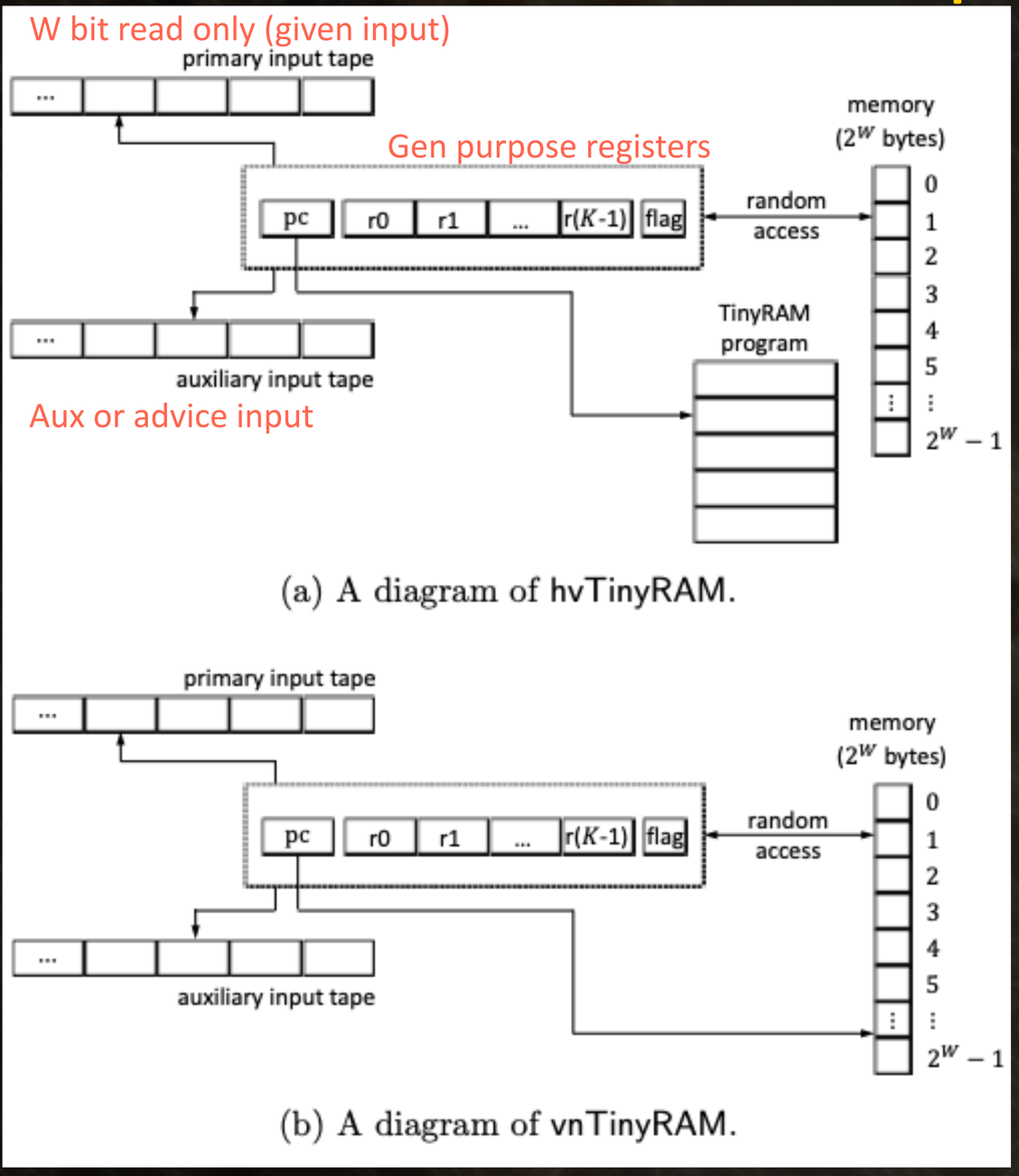
Memory: 2^W bytes

Tapes: Read only in one direction

- VN - program resides on memory (same address space as data) - a prog can modify its own code
- HVRD - Program resides in separate address space

TinyRAM [BCGTV13/20] architecture

Front end compiler: eg VM - TinyRAM



The input tapes model streams of data that can be read exactly once

VN: Primary tape - program and data

VN: Aux tape - “advice” to speed up calculations

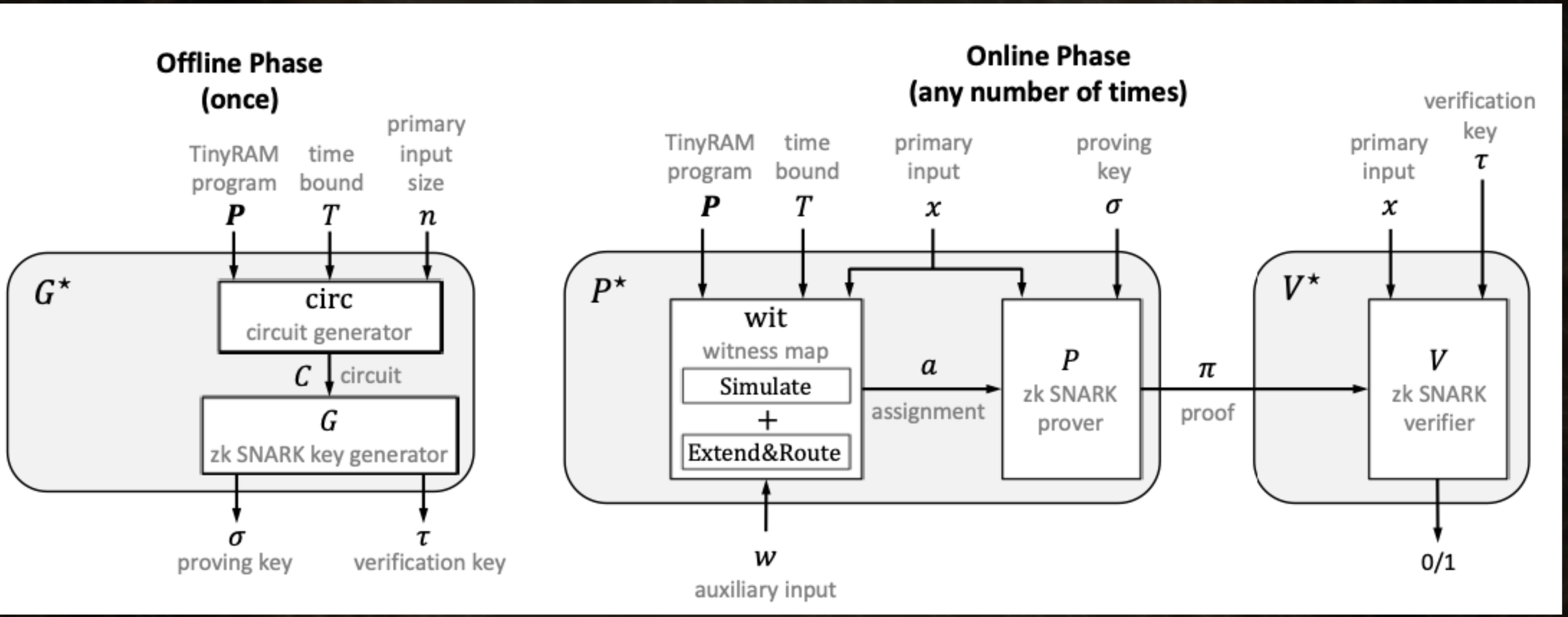
ZK friendly ISA, with 16 bit or 32 bit reg sizes

Prove that: a fetched instruction was executed as per vnTinyRAM model.

Prove that :At each time step, the correct instruction was fetched from memory

Prove that: each load from memory retrieves the last value stored there.

BCGTV12: C++ to tinyRAM



We dont worry about the backend prover in this talk, but to the curious, this uses a QAP model like Pinocchio for the SNARK backend and is incredibly inefficient.

Front end compiler: eg VM - TinyRAM

instruction mnemonic	operands	effects	flag
`and`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute bitwise AND of $[\mathtt{r}j]$ and $[A]$ and store result in $\mathtt{r}i$	result is 0^W
`or`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute bitwise OR of $[\mathtt{r}j]$ and $[A]$ and store result in $\mathtt{r}i$	result is 0^W
`xor`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute bitwise XOR of $[\mathtt{r}j]$ and $[A]$ and store result in $\mathtt{r}i$	result is 0^W
`not`	$\mathtt{r}i$ A	compute bitwise NOT of $[A]$ and store result in $\mathtt{r}i$	result is 0^W
`add`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute $[\mathtt{r}j]_\mathrm{u} + [A]_\mathrm{u}$ and store result in $\mathtt{r}i$	overflow
`sub`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute $[\mathtt{r}j]_\mathrm{u} - [A]_\mathrm{u}$ and store result in $\mathtt{r}i$	borrow
`mull`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$ and store least significant bits of result in $\mathtt{r}i$	overflow
`umulh`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute $[\mathtt{r}j]_\mathrm{u} \times [A]_\mathrm{u}$ and store most significant bits of result in $\mathtt{r}i$	overflow
`smulh`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute $[\mathtt{r}j]_\mathrm{s} \times [A]_\mathrm{s}$ and store most significant bits of result in $\mathtt{r}i$	over/underflow
`udiv`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute quotient of $[\mathtt{r}j]_\mathrm{u}/[A]_\mathrm{u}$ and store result in $\mathtt{r}i$	$[A]_\mathrm{u} = 0$
`umod`	$\mathtt{r}i$ $\mathtt{r}j$ A	compute remainder of $[\mathtt{r}j]_\mathrm{u}/[A]_\mathrm{u}$ and store result in $\mathtt{r}i$	$[A]_\mathrm{u} = 0$
`shl`	$\mathtt{r}i$ $\mathtt{r}j$ A	shift $[\mathtt{r}j]$ by $[A]_\mathrm{u}$ bits to the left and store result in $\mathtt{r}i$	MSB of $[\mathtt{r}j]$
`shr`	$\mathtt{r}i$ $\mathtt{r}j$ A	shift $[\mathtt{r}j]$ by $[A]_\mathrm{u}$ bits to the right and store result in $\mathtt{r}i$	LSB of $[\mathtt{r}j]$
`cmpe`	$\mathtt{r}i$ A	none ("compare equal")	$[\mathtt{r}i] = [A]$
`cmpa`	$\mathtt{r}i$ A	none ("compare above", unsigned)	$[\mathtt{r}i]_\mathrm{u} > [A]_\mathrm{u}$
`cmpae`	$\mathtt{r}i$ A	none ("compare above or equal", unsigned)	$[\mathtt{r}i]_\mathrm{u} \geq [A]_\mathrm{u}$
`cmpg`	$\mathtt{r}i$ A	none ("compare greater", signed)	$[\mathtt{r}i]_\mathrm{s} > [A]_\mathrm{s}$
`cmpge`	$\mathtt{r}i$ A	none ("compare greater or equal", signed)	$[\mathtt{r}i]_\mathrm{s} \geq [A]_\mathrm{s}$
`mov`	$\mathtt{r}i$ A	store $[A]$ in $\mathtt{r}i$	
`cmov`	$\mathtt{r}i$ A	if flag $= 1$, store $[A]$ in $\mathtt{r}i$	
`jmp`	A	set pc to $[A]$	
`cjmp`	A	if flag $= 1$, set pc to $[A]$ (else increment pc as usual)	
`cnjmp`	A	if flag $= 0$, set pc to $[A]$ (else increment pc as usual)	
`store.b`	A $\mathtt{r}i$	store the least-significant byte of $[\mathtt{r}i]$ at the $[A]_\mathrm{u}$-th byte in memory	
`load.b`	$\mathtt{r}i$ A	store into $\mathtt{r}i$ (with zero-padding in front) the $[A]_\mathrm{u}$-th byte in memory	
`store.w`	A $\mathtt{r}i$	store $[\mathtt{r}i]$ at the word in memory that is aligned to the $[A]_\mathrm{w}$-th byte	
`load.w`	$\mathtt{r}i$ A	store into $\mathtt{r}i$ the word in memory that is aligned to the $[A]_\mathrm{w}$-th byte	
`read`	$\mathtt{r}i$ A	if the $[A]_\mathrm{u}$-th tape has remaining words then consume the next word, store it in $\mathtt{r}i$, and set flag $= 0$; otherwise store 0^W in $\mathtt{r}i$ and set flag $= 1$	←
`answer`	A	stall or halt (and the return value is $[A]_\mathrm{u}$)	

(1) All but the first two tapes are empty: if $[A]_\mathrm{u} \notin \{0, 1\}$ then store 0^W in $\mathtt{r}i$ and set flag $= 1$.
(2) `answer` causes a stall (i.e., not increment pc) or a halt (i.e., the computation stops); the choice between the two is u

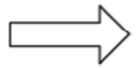
TinyRAM [BCGTV13/20] 29 Instructions

- ISA: simple load and store instructions
- simple integer, shift, logical, compare, move, and jump instructions,
- control flow, loops, sub routines, recursion.
- Complex instructions, such as floating-point arithmetic, are not directly supported and can be implemented "in software" — these increase machine code size and overhead.
- Specify with opcode and operand

Front end compiler: eg reducing C++ to tiny RAM

Input: (P, x, T) , program, x - string of W bit words, T - time bound such that $\exists\ w$ for which $P(x, w)$ accepts (answer 0) in T steps. [BCGTV13]

```
void sumarray(int size,
              int* A,
              int* B,
              int* C)
{
  int i;
  for (i=0; i<size; i++) {
    C[i] = A[i] + B[i];
  }
}
```



```
_sumarray:
    cmpe  r4, r5
    cjmp  _end
    load  r6, r1
    load  r7, r2
    add   r8, r7, r5
    store r3, r8
    add   r1, r1, 1
    add   r2, r2, 1
    add   r3, r3, 1
    add   r4, r4, 1
    jmp   _sumarray
_end:
```

BCGTV12 C++ to tinyRAM

Local state S: (W+KW+1) bits at any instant

Transition

$$\Pi_{tr}(P, S, S') = \begin{cases} 1 \ \ iff\ S \rightarrow^{P} S' \\ 0 \end{cases}$$

Execution trace $tr = (S_1, S_2, \ldots S_T)$

An execution trace tr is valid if there exists an auxiliary input w such that the sequence of states induced by P running with inputs x, w is tr.

TinyRAM compiler: Designs a circuit that verifies tr, and that tr is as small as possible.

Front end compiler: eg reducing C++ to tiny RAM

BCGTV12 C++ to tinyRAM

- Code consistency: C_{TF} Circuit that implements transition function Π_{TF}

$$C_{TF}(P, S, S') = 1 \ \textit{iff} \ \Pi_{TF}(P, S, S') = 1$$

Invoke sequentially and check $\Pi_{TF}(P, S_i, S_{i+1}) = 1 \ \forall \ i = 1, \ldots T - 1$

- Memory consistency: Every load from an address retrieves the value of last store

Maintaining an entire snapshot of memory is inefficient and leads to $\Omega(T^2)$ behavior

Trace sorted in order of accessed addresses, and use timestamps to break ties [BCGTV13]

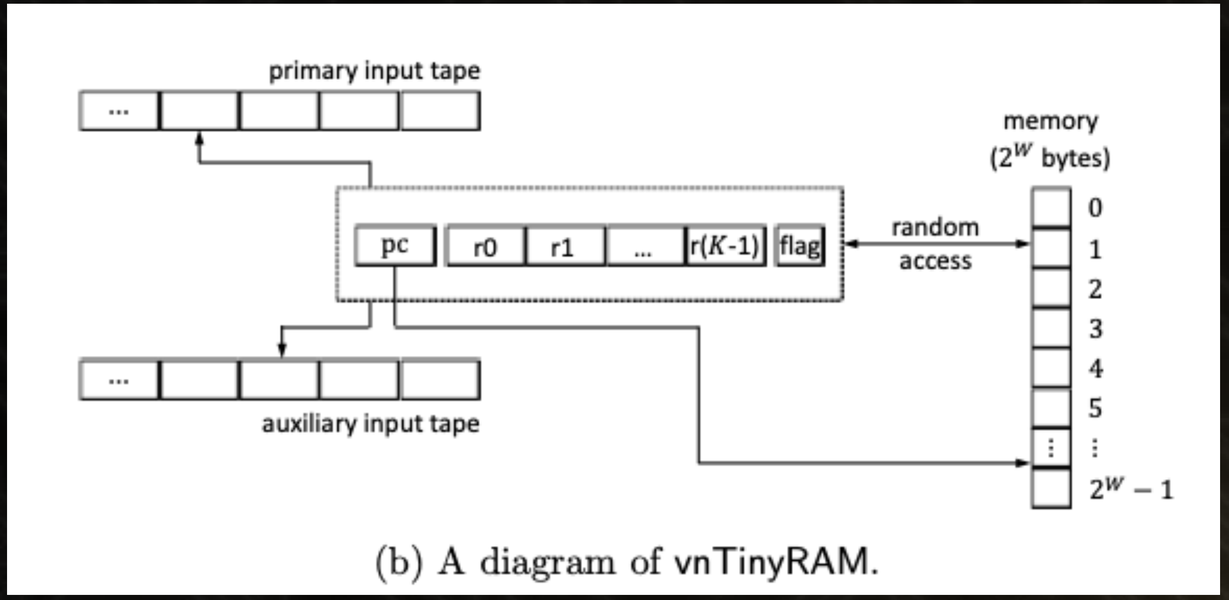
- Routing network: Represent the states as a directed graph with T sources, sinks and nodes such that any permutation $[T] \rightarrow [T]$

Switch settings on network = graph coloring on network

This at the heart of all VM

Any arbitrary program is a certain coloring on the routing network graph

How to do Memory consistency check?



(b) A diagram of vnTinyRAM.

How do you simulate a memory bank?

State: Time stamp in CPU cycles

Merkle root over contents of “external” memory

Program counter and register set

Flag denoting if program is accepted for proving

CPU state can be kept small, irrespective of size of RAM memory

Any data loaded from this “external memory”, is loaded in auxiliary advice tape, and provide with a Merkle authentication path that it is indeed the correct value stored in the “external memory”

This can be chained to load-then-store requests as well, by updating the root.

Thus “external memory” operations across the trace can be reliably proven

Under the hood tinyRAM VM is Still CSAT but on a routing network!

```
_sumarray:
     cmpe   r4, r5
     cjmp   _end
     load   r6, r1
     load   r7, r2
     add    r8, r7, r5
     store  r3, r8
     add    r1, r1, 1
     add    r2, r2, 1
     add    r3, r3, 1
     add    r4, r4, 1
     jmp    _sumarray
_end:
```

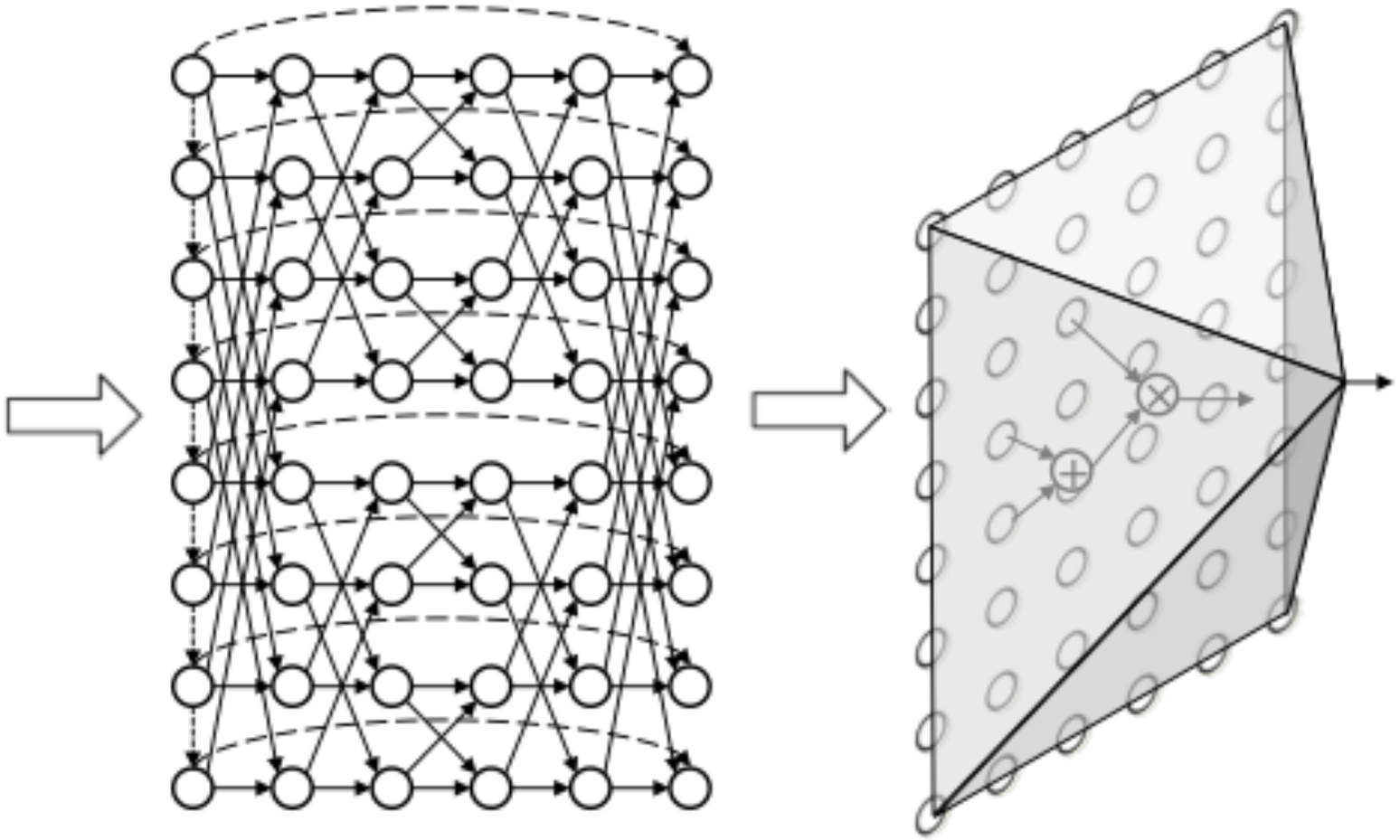


Figure 6: Verifying the correct execution of a given piece of TinyRAM code (left) is reduced to satisfiability of a certain constraint-satisfaction problem on a routing network (middle). Then, an arithmetic circuit (right), given an assignment as input, verifies all the constraints of the constraint-satisfaction problem.

BCGTV12 C++ to tinyRAM

Final remarks: vn TinyRAM

1. The generated circuit, “vnTinyRAM universal” implements one cycle of the CPU

2. Takes input as a prev CPU state, and the current state, along with a SNARK proof of the previous state

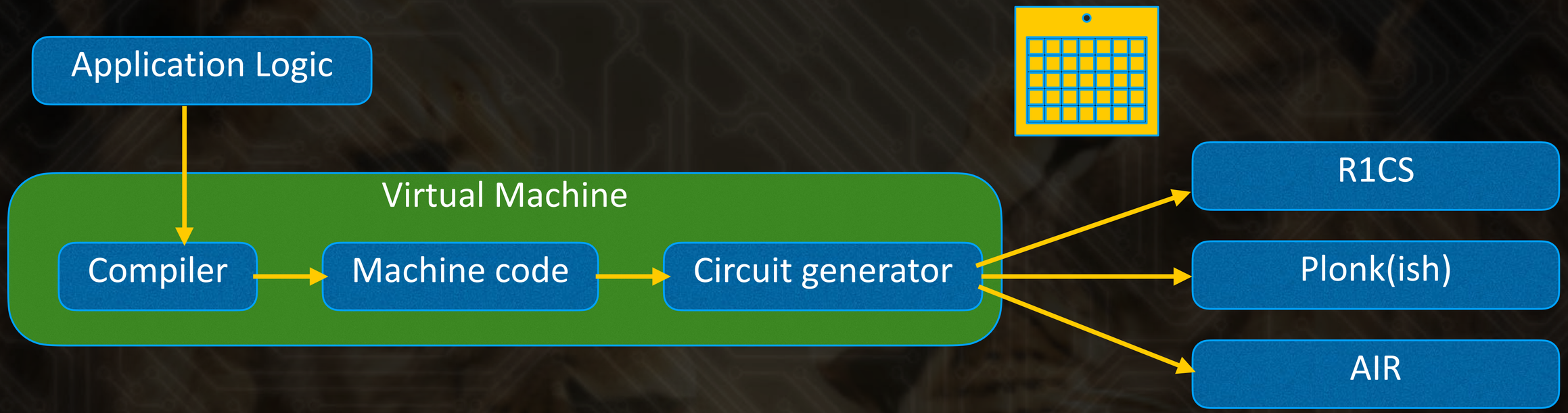
3.Circuit checks, the prior proof and that the current state transition is valid.

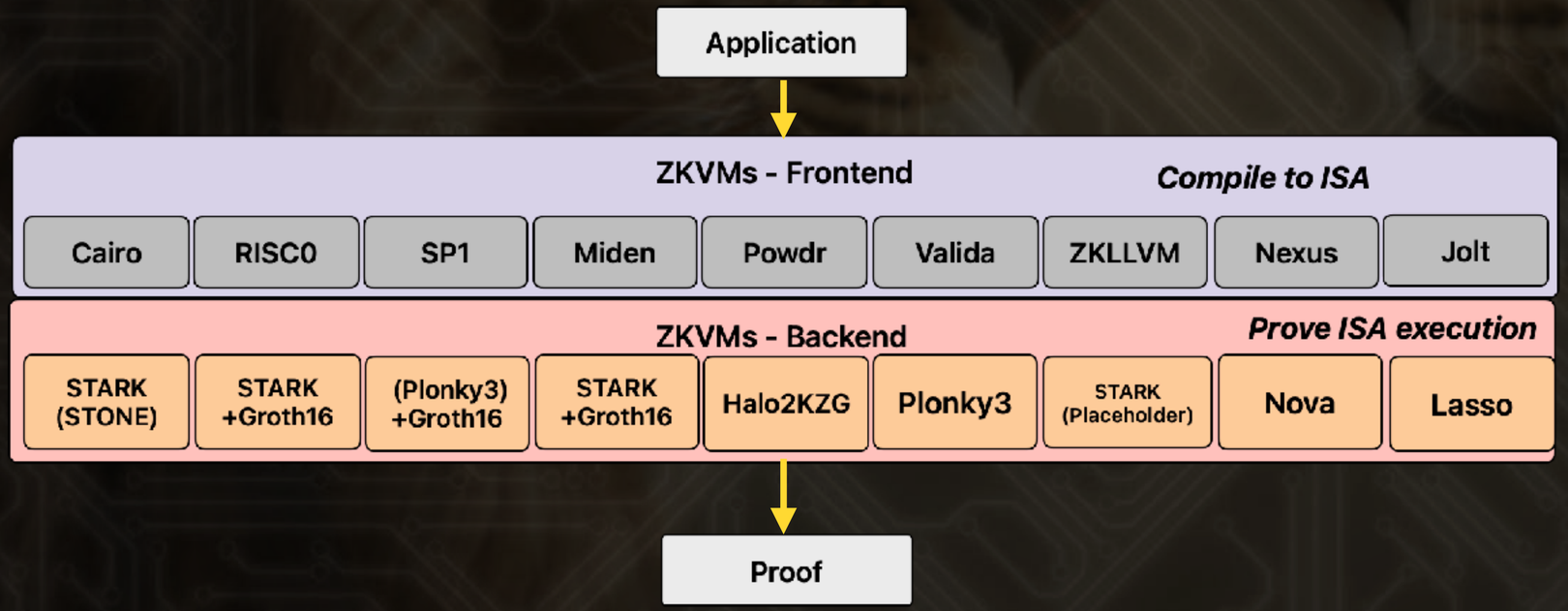
4. The back end generates an updated proof, which can be fed into the universal circuit again for the next clock cycle.

5.recursion was implemented using QAP and cycles of curves in BCGTV13

We should really go back and read the old papers, all the fancy languages used now trace back to fundamental papers like BCGTV13

Front end compiler CPU approach: summary





zkVM	ISA	Backend
CAIRO [35]	RISC (Custom)	STARK [9]
Jolt [36]	RISC-V	LASSO [37]
Lurk [38]	Lurk (Custom)	Nova [10]
Miden [39]	Assembly (Custom) [40]	STARK based [41]
Nexus [42]	NVM (Custom)	Nova [10], Hypernova [43]
Powdr [44]	Assembly (Custom)	Halo2KZG [6]
RISC0 [45]	RISC-V	STARK+Groth16 [46]
SP1 [47]	RISC-V	Plonky3 [8]
Valida [48]	RISC (Custom)	Plonky3 [8]

PolyAPI paper

4. ZKEVM ⊂ ZKVM

What exactly is a ZKEVM

The Ethereum VM is the VM that runs Ethereum smart contracts in solidity language

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted. The zero-th power of zero 0^0 is defined to be one.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$

10s: Comparison & Bitwise Logic Operations

Value	Mnemonic	δ	α	Description
0x10	LT	2	1	Less-than comparison. $\mu'_s[0] \equiv \begin{cases} 1 & \text{if } \mu_s[0] < \mu_s[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\mu'_s[0] \equiv \begin{cases} 1 & \text{if } \mu_s[0] > \mu_s[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\mu'_s[0] \equiv \begin{cases} 1 & \text{if } \mu_s[0] < \mu_s[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.

30s: Environmental Information

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Get address of currently executing account. $\mu'_s[0] \equiv I_a$
0x31	BALANCE	1	1	Get balance of the given account. $\mu'_s[0] \equiv \begin{cases} \sigma[\mu_s[0] \bmod 2^{160}]_b & \text{if } \sigma[\mu_s[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$ $A'_a \equiv A_a \cup \{\mu_s[0] \bmod 2^{160}\}$
0x32	ORIGIN	0	1	Get execution origination address. $\mu'_s[0] \equiv I_o$ This is the sender of original transaction; it is never an account with non-empty associated code.

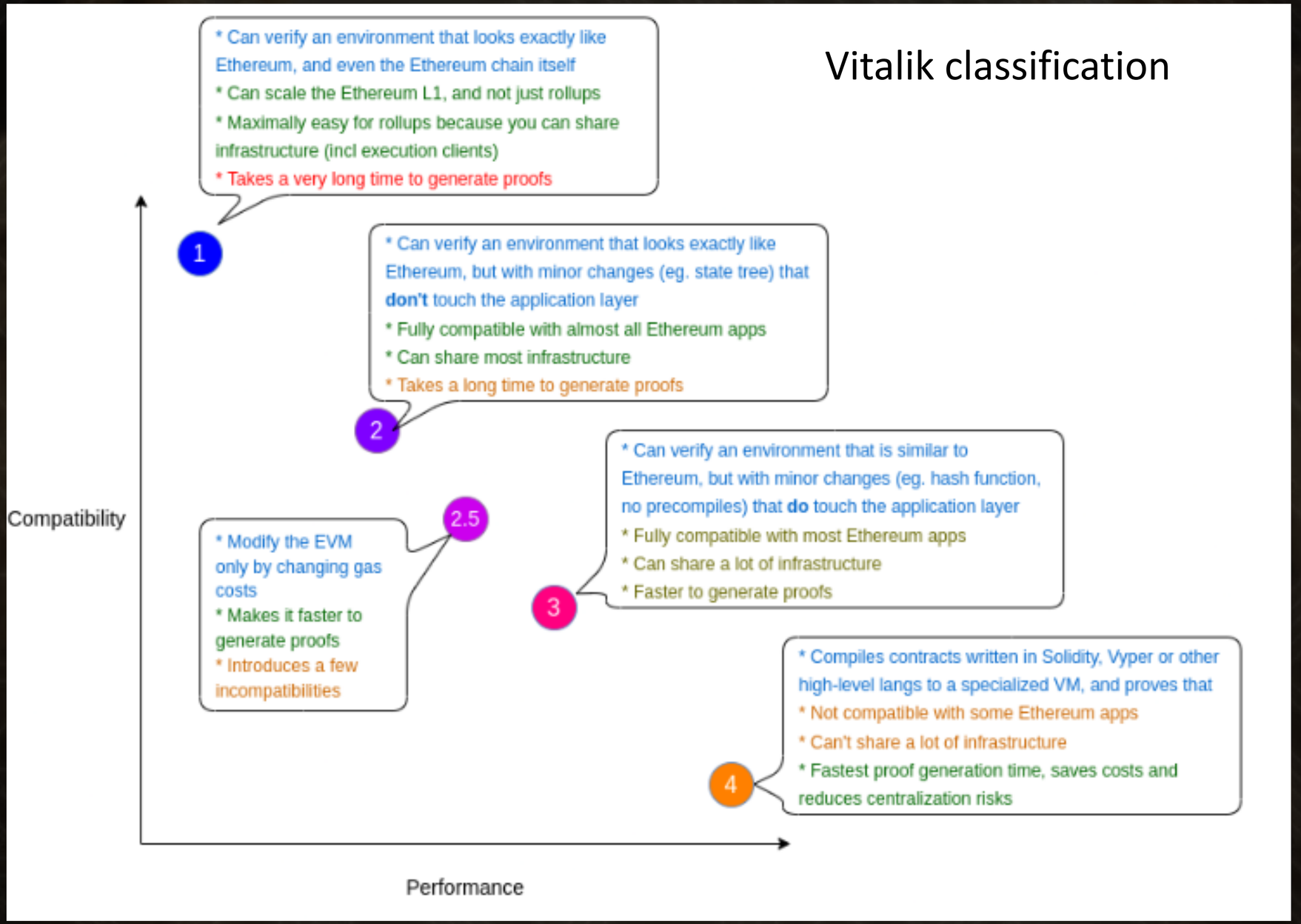
40s: Block Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\mu'_s[0] \equiv P(I_{H_p}, \mu_s[0], 0)$ where P is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than or equal to the current block number or more than 256 blocks behind the current block. $P(h, n, a) \equiv \begin{cases} 0 & \text{if } n > H_i \vee a = 256 \vee h = 0 \\ h & \text{if } n = H_i \\ P(H_p, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined from its hash h unless h is zero (as is the case for the parent hash of the genesis block).
0x41	COINBASE	0	1	Get the current block's beneficiary address. $\mu'_s[0] \equiv I_{H_c}$

EVM uses these opcodes such as above to performs all the transactions on Ethereum blockchain (World CPU)

Ethereum yellow paper

What exactly is a ZKEVM



It is very confusing to really understand the exact definition of the ZKEVM, it varies

Detour: Precompiles

Precompiles, these are contracts in Ethereum include complex cryptographic computations, but do not require the overhead of the EVM - native execution in ethereum.

They do not execute inside a smart contract, they are part of the Ethereum client specification.

Precompiles:
ecrecover,
sha256,
sha3FIPS256,
ripemd-160,
Bn128Add,
Bn128Mul,
Bn128Pairing,
identity function,
modular exponentiation.

Fixed gas cost for execution per precompile

This hashing function returns the SHA256 hash from the given data. To test this precompile, you can use this SHA256 Hash Calculator tool to calculate the SHA256 hash of any string you want. In this case, you'll do so with `Hello World!`. You can head directly to Remix and deploy the following code, where the calculated hash is set for the `expectedHash` variable:

```
pragma solidity ^0.7.0;

contract Hash256 {
    bytes32 public expectedHash =
        0x7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069;

    function calculateHash() internal pure returns (bytes32) {
        string memory word = "Hello World!";
        bytes32 hash = sha256(bytes(word));

        return hash;
    }

    function checkHash() public view returns (bool) {
        return (calculateHash() == expectedHash);
    }
}
```

Once the contract is deployed, you can call the `checkHash()` method that returns **true** if the hash returned by `calculateHash()` is equal to the hash provided.

Behave like “smart contracts” but execute on eth client

Prover simply outsources it to onchain checks

ZKEVM ⊂ ZKVM

ZKVM tells you, I ran this machine code, I can give you a receipt for running it eg: Risc0

can also run ethereum related programs eg EVM, eg solidity but may require coding the logic on software, since the “hardware ISA” doesn’t support it natively

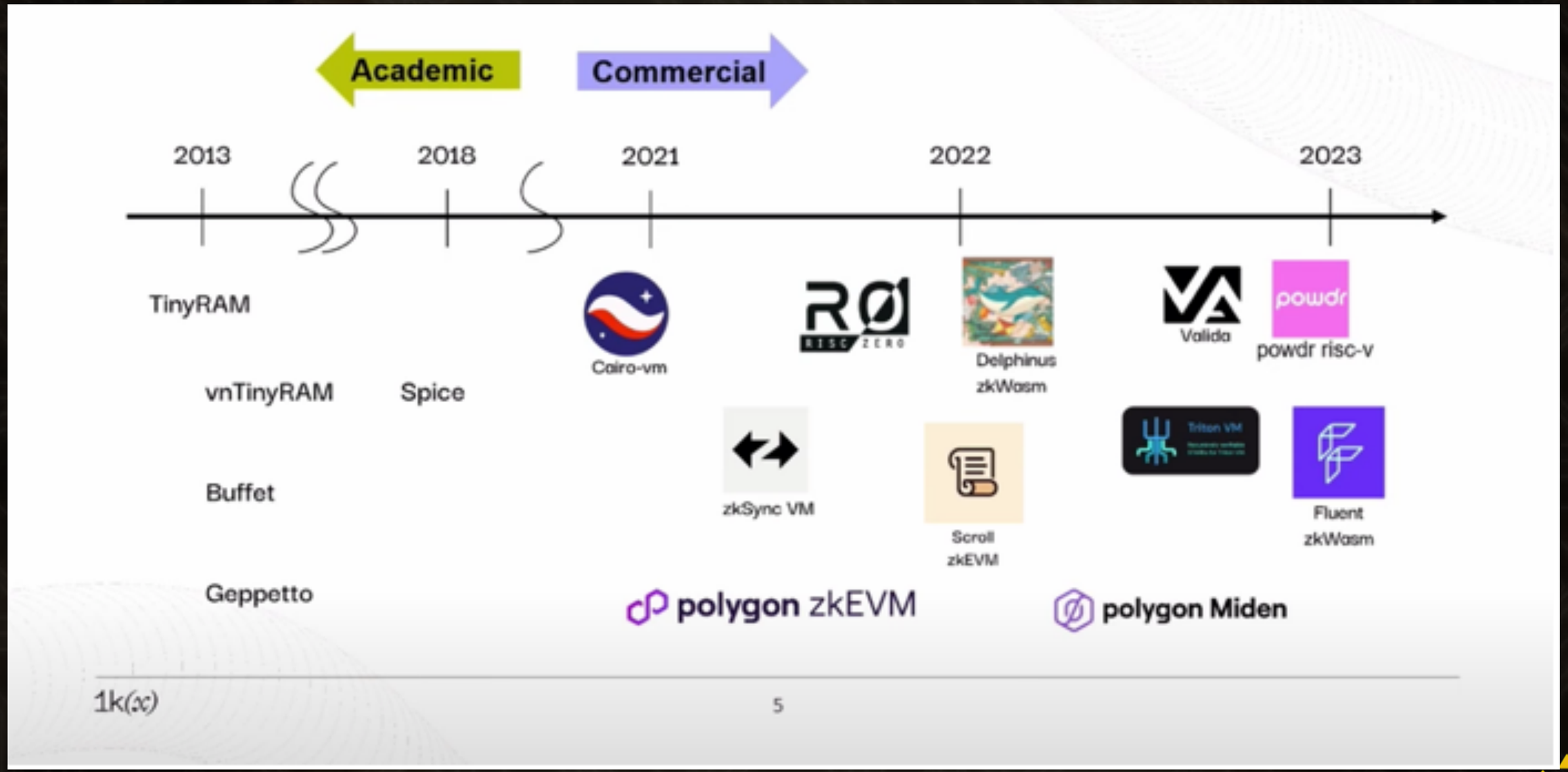
RISC0 - uses the RISC-V architecture which is a set of instructions used for general-purpose computing

RISC-V eg doesn’t have a notion of wallet addresses or other blockchain constructs or anything in the EVM opcode - u can always encode this logic using RISCV ISA

RISCV ISA is mostly composed of operators that move data between memory locations and do mathematical operations on data.

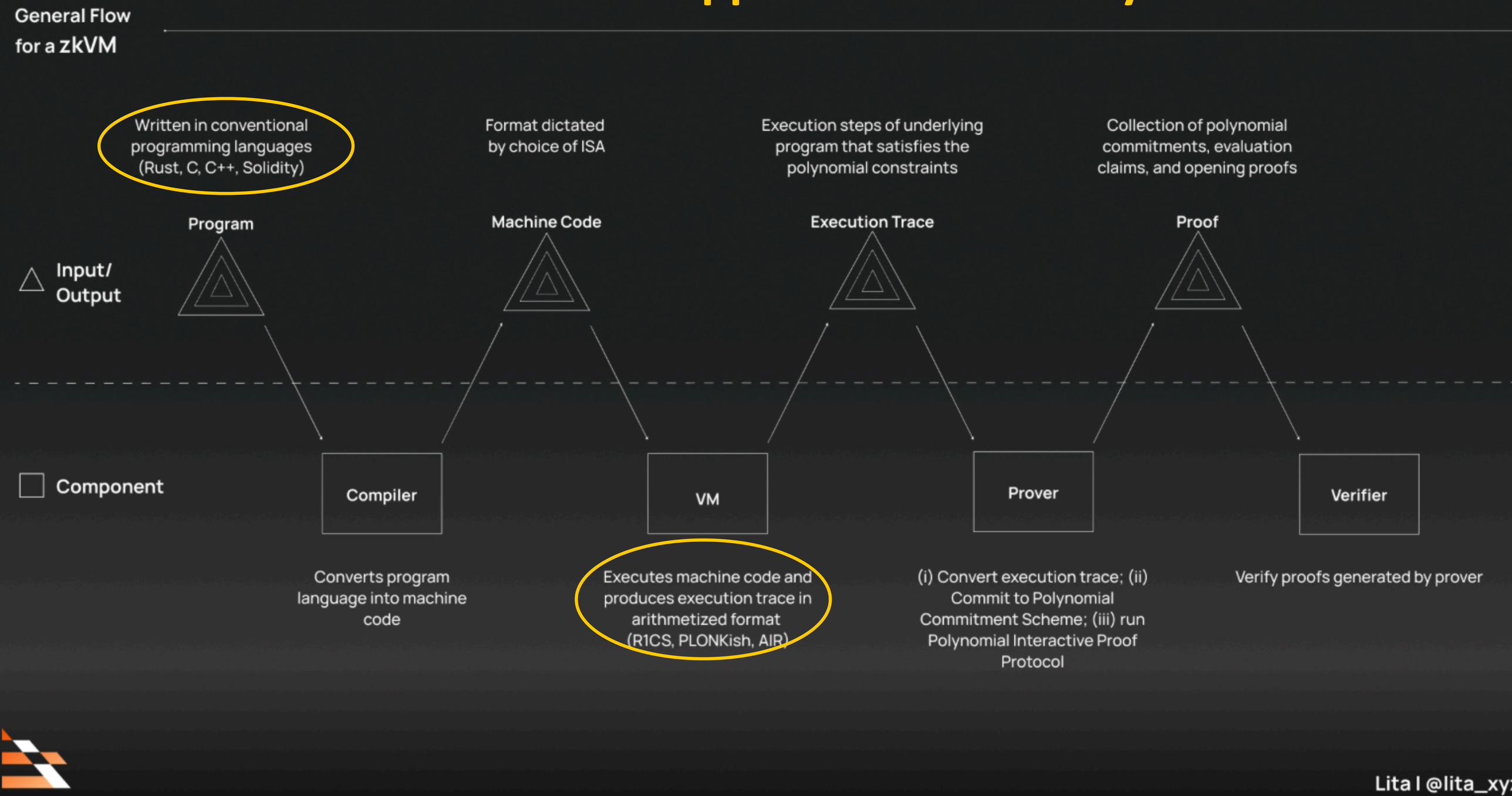
One can write high level logic in any language, rust,c++,go as long as ISA can compile it.

Cambrian explosion of ZK(E)VMs starting from tinyRAM



5. Aspects of ZKVM design

Modern ZKVM approach follows tinyRAM



Modern ZKVM implementations follow the crowd

Favors rust based compilers, love stark and small fields. I’m still for good old c++

System	Proof System	ISA (VM)	Compiler	Programming Language(s)
Starknet	STARKs	Cairo	Proprietary	Cairo
Lita	STARKs	Valida	LLVM	C, C++, Rust, Solidity
Risc Zero	STARKs	RISC-V	Rust Compiler	Rust
Succinct Labs	STARKs	RISC-V / Valida	Rust Pre-compiles	Rust
NEXUS	STARKs	NVM (Nexus Virtual Machine)	Rust	Rust
Polygon Miden	STARKs	Miden	Rust	Rust

ZKVM general structure

C++

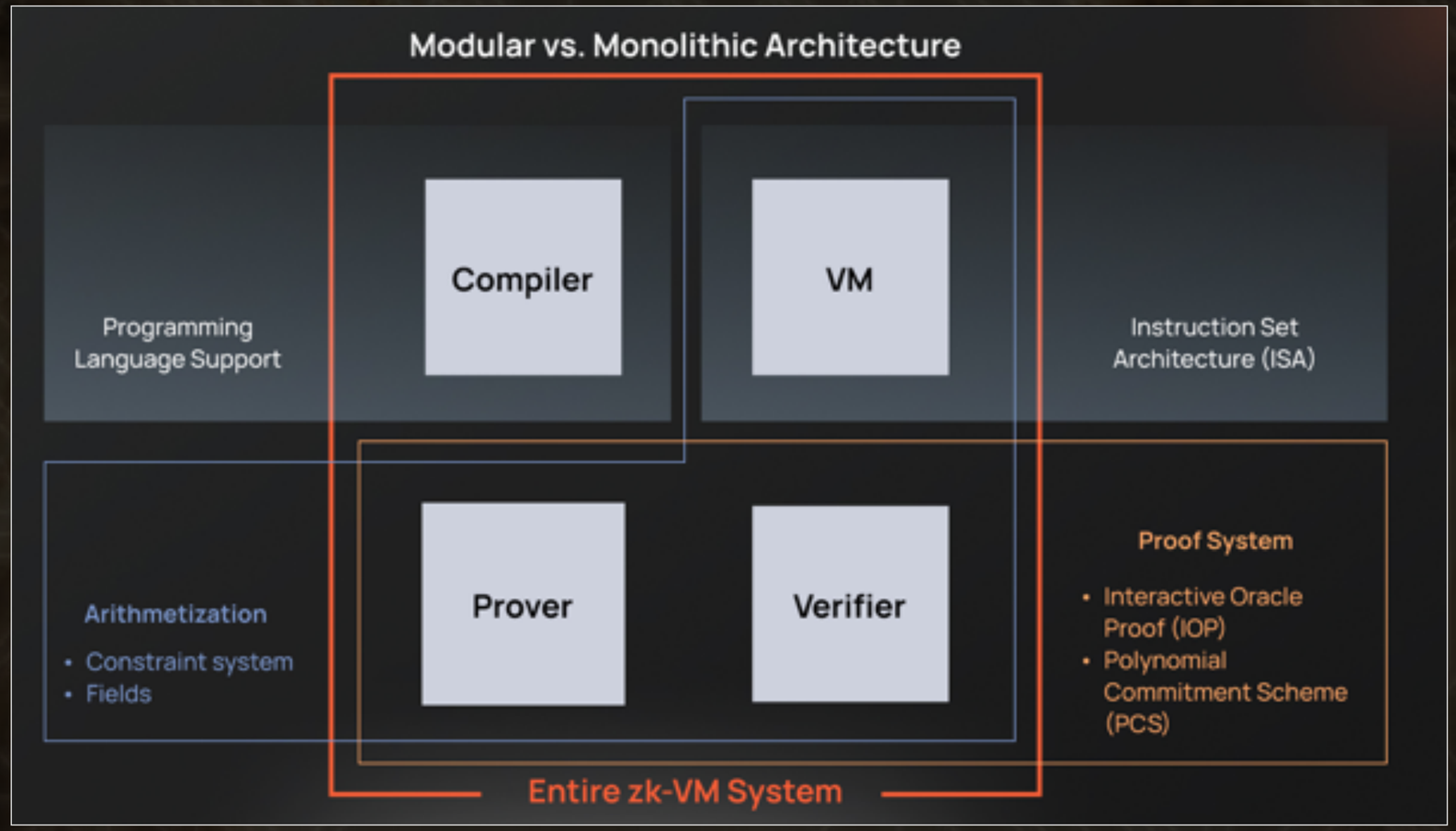
Go

Rust

AIR

R1CS

Plonkish



Memory model

VN vs Har

32-bit vs. 64-bit ISAs

EC vs hash

UnivarIOP vs MLEIOP

Quotient vs sumcheck

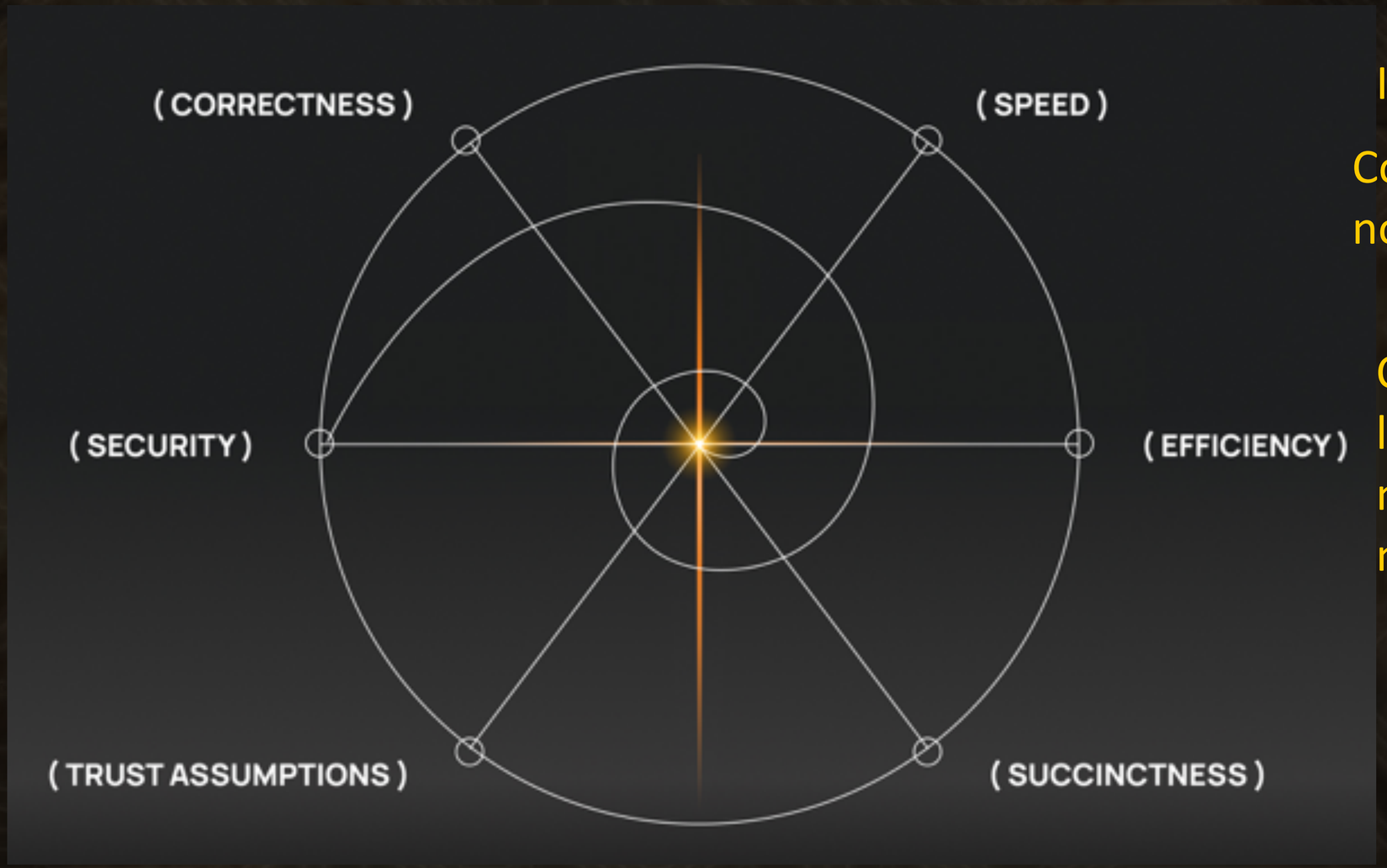
Performance seems to be generally affected by all of the design choices above - not clear

Which is a good ZKVM?

Completeness
Soundness
ZK

N bits of security, = tolerance of 1 / (2^N)

Trusted setup?



Tricky

ISA complexity

Cost per instruction x no of instruction

Can affect length/ memory of machine code

More design choices

zk-VM : Modular vs Monolithic

	Modular zk-VM	Monolithic zk-VM
Architecture	Built partially out of pluggable components	Closed system
Extensibility	Extensible via a potentially infinite number of pluggable components	Cannot be extended by the user
Configurability	Can be instantiated in a potentially infinite number of different configurations	Lacks the flexibility for user-defined configurations
Instruction Set	Open-ended	Closed-ended, updated by the zk-VM project itself
Code Architecture	Clean separation of concerns, facilitating economical customizations	Often lacks clean separation, making customization difficult

Eg: Design space

Key Aspects of Valida zk-VM Design

VM	Valida
Instruction Sets	Custom-made RISC-inspired instruction set
Flow Control	Harvard Architecture
Native Field	32-bit Machine
ZK Stack	**Plonky3**
Polynomial Interactive Oracle Proofs (PIOPs)	Univariate STARK
Polynomial Commitment Scheme (PCS)	FRI-based
Field Choices	Mersenne31*, BabyBear, Goldilocks
Group/Hash	Rescue, Poseidon, Poseidon 2, Blake3, Keccak-256, Monolith
Compiler	**Valida Compiler**
Frontend	Conventional Language compiler to LLVM IR
Backend	Custom built LLVM IR to Valida VM Bytecode compiler

*to be supported

LLVM (Low level virtual machine) is a compiler toolchain that talks to x86,ARM and the likes

Thank you!