OlaVM: An Ethereum compatible ZKVM

Sin7Y, Applied R&D Team[*]

July, 2022

Abstract

We designed the architecture for OlaVM, a customized ZKVM, with ambitions to become fully compatible with the existing Ethereum ecosystem, providing seamless migration for projects, enabled at compiler level rather than circuit constraint level. The OlaVM instruction set design strikes a graceful balance between trade-offs in execution trace size and amount of constraints. Utilizing a register-based VM, the overall execution traces of OlaVM are much smaller than that of a stack-based VM. The design of OlaVM revolves around the following main features: (1) A custom virtual machine with a simplified instruction set designed to improve the execution process and ZK verification. (2) Register-based structure, which greatly reduces memory access overhead during the execution process, effectively reducing the scale of the entire execution trace. (3) Finite Field Word, the Word of OlaVM is a field element, the only type of computations that can be performed are field operations, which enables OlaVM to obtain a set of concise state transition constraints. (4) Modular design, dividing the entire execution trace into multiple sub traces based on operation type and processing them separately. (5) FPGA acceleration, utilizing the FPGA acceleration logic of the main calculation module of the ZK algorithm. (6) Zero Knowledge without FFT, eliminating the most computationally expensive module, FFT, in implementing ZK calculations on FPGAs. (7) Other tricks, improving computational efficiency of OlaVM, and of ZK verification through non-deterministic features.

Contents

[*]https://twitter.com/Sin7Y_Labs

1 Introduction

The introductory section will introduce the fundamental design ideas, advantages and disadvantages of similar projects within the space, the main features of OlaVM design, as well as outlining the content of subsequent sections. In this paper we call all Ethereum compatiable virtual machines as ZKEVM.

1.1 Industry Progress

The blockchain industry has seen a tremendous pace of development over the past few years, exploring applications and opportunities built on these systems. With this rapid expansion, the industry faces a major technical challenge in improving processing capacity of the blockchain to achieve higher throughput while still maintaining its security. Currently, some very prominent teams have made great progress to achieve this, such as PSE [21], Matter Labs [16], Polygon Hermez [20], StarkWare [30], Risc0 [22] and Scroll [24].

We are able to divide the above mentioned projects into three different levels based on differences in their compatibility with EVM: Consensus-level, Bytecode-level, and Language-level.

First up is PSE, that has a Consensus-level compatibility with EVM, meaning it is fully compatible with EVM. The ZK constraints are consistent with the current behaviour of EVM, this includes the update of the state root, which is advantageous in that you only need to design the circuit to constrain all the rows of EVM. Secondly, Scroll and Polygon has a Bytecode-level compatibility with EVM, which directly interprets the bytecode of EVM, whilst using a different account model to increase ZK efficiency, resulting in different state roots. The advantage of this is that the compatibility with EVM is still guaranteed and the proof efficiency of state root is improved. Due to the introduction of different account models, additional changes at EVM level are required compared to a Consensus-level scheme. Last up is Language-level compatibility, utilized by projects such as StarkWare, Matter Labs and Risc0. They define a ZK-friendly VM, which includes a custom instruction set, custom memory model and so on. They convert Solidity (or Yul) into custom VM-supported instructions through a compiler. The advantage of this solution is achieving that allows for deploying most contracts directly without modification and at the same time providing the highest ZK efficiency in its smaller execution traces and simple state transition constraints. However, trade-offs lie in huge workloads involving constraint design, VM design and compiler design, amongst others. Following we will briefly introduce the general framework of the three mentioned schemes.

1.1.1 Consensus-level Compatibility

The main advantage of this scheme is that it maintains full compatibility to the current behaviour of EVM. All behaviors of EVM are constrained by circuits, including but not limited to memory access, instruction execution, and world state updates. Execution traces are generated by EVM Runner, then it is generating a proof that EVM produces combined with its corresponding ZK constraint. The framework of this scheme is shown in Figure 1.

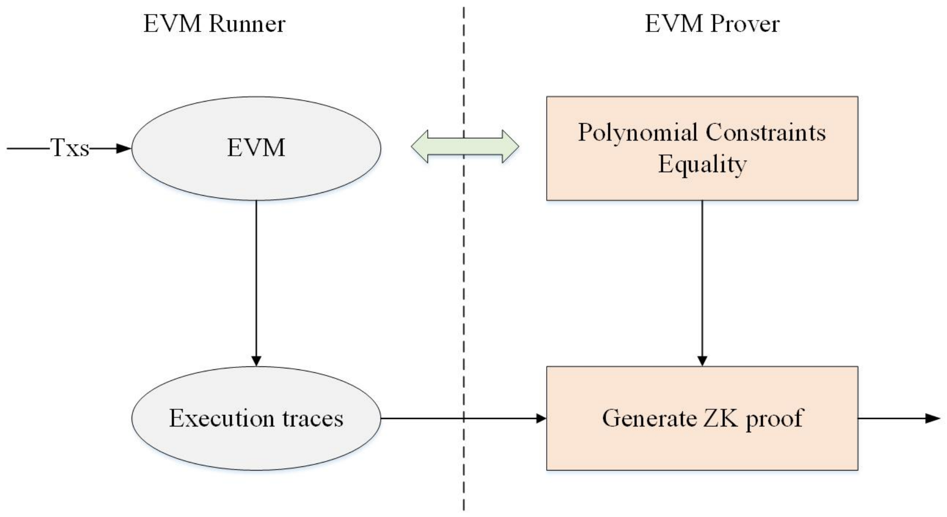


Figure 1: ZKEVM architecture based on Consensus-level

The main workload of this scheme lies in the design of ZK constraints which does not require any modifications to EVM. Which is advantageous, but it is also disadvantageous because there are ZK-unfriendly operations in EVM that also must be proven by ZK-constraints, which makes the overall ZK efficiency low.

1.1.2 Bytecode-level Compatibility

Intuitively you may assume that the feature of this scheme is that it can process all the instructions of EVM on Bytecode-level (That the bytecodes of a Solidity contract can be directly used as input), however, other parts than the instructions can be different, such as getting a different state root and using a different hash algorithm. The framework of the scheme is shown in Figure 2.

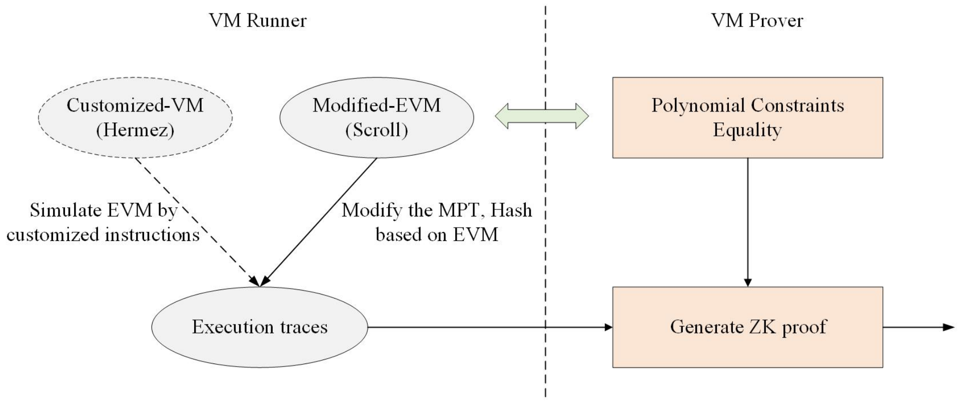

VM Runner
VM Prover
Customized-VM (Hermez)
Modified-EVM (Scroll)
Polynomial Constraints Equality
Simulate EVM by customized instructions
Modify the MPT, Hash based on EVM
Execution traces
Generate ZK proof

Figure 2: ZKEVM architecture based on Bytecode-level

As you can tell by this scheme it requires designing or modifying the VM in addition to designing ZK constraints. As an example, Scroll may only need to make corresponding modifications based on the Ethereum client, but Hermez is slightly more complicated due to them customizing a VM. Compared with the Consensus-level ZKEVM scheme, the ZK efficiency of this scheme is slightly better.

1.1.3 Language-level Compatibility

Compared to previous two schemes, Language-level compatibility is the weakest. It does not have to be 100% compatible with EVM, which means that simple modifications to some contracts could result in it needing a lot of support. However, the overall scheme has a ZK-friendly design, e.g, the memory model, instruction set, account model and so on, hence it has a better ZK efficiency. The framework of this scheme is shown in Figure 3.

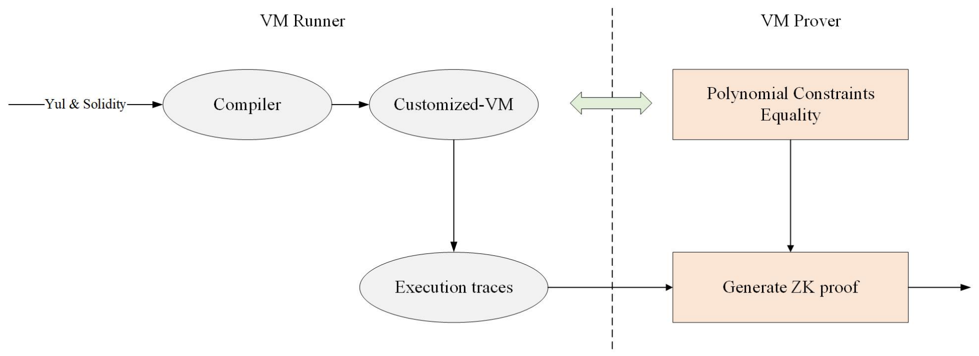


Figure 3: ZKEVM architecture based on Language-level

Out of all the schemes this has the highest workload, involving the compiler, VM & constraint design and so on. However, it is also achieving the best ZK efficiency. An example would be StarkWare's Cairo VM, which has a surprisingly good performance in terms of capacity expansion. StarkWare team has come quite far with the Cairo design in terms of being ZK-friendly since the inception of teams pursuing developing a ZKEVM solution.

1.2 Our Work

Designing and implementing a fully functional ZKEVM is a very daunting and challenging task. We have thoroughly studied and followed the progress of other teams in this space, and mastered how to design an efficient ZK-friendly EVM.

OlaVM is designed to have strong EVM compatibility without compromising its ZK-friendliness. To maintain its ZK-friendliness we've defined a simple instruction set, which makes the underlying state transition constraints very concise. In addition to this, OlaVM is a register-based VM, which significantly reduces the execution trace (avoiding intensive PUSH and POP operations compared to a stack-based VM). The Word of OlaVM is an element defined in a finite field, that's how we are able to obtain a simplified set of instructions. Finally, OlaVM supports some neat tricks to prevent the VM itself from performing complex calculations, reducing the execution trace and thereby improving overall ZK efficiency. We have defined integer calculation logic of uint256 to achieve solid EVM compatibility. In order to reduce the execution trace of integer calculation, we use BLS12-381 curve, which is defined on a 381-bit characteristic finite field.

In terms of design, based on the technical characteristics described above, OlaVM generates smaller execution traces and more concise state transition constraints, which enables us to achieve better ZK efficiency.

In addition to that we utilize other technical means such as reducing the number of selector polynomials by Combined Selector technology, as well as reducing the constraint scale using Lookup Argument technology, in order to improve ZK efficiency. For detailed principle, please refer to Section 11. To further improve ZK efficiency, we have got a ZK hardware acceleration design framework. In order to ensure scalability of the design, we only accelerate the core module in ZK algorithm – MSM (Multi-Scalar Multiplication), so that our design still applies to other ZK algorithms.

OlaVM uses a ZK algorithm without FFT in our design (for detailed principle, please refer to Section 9), not simply a classic ZK algorithm, such as Halo2 [11], Plonk [6], Plonky2 [19] and Groth16 [8]. Reasoning behind this is, with ZK algorithms, although FFT/IFFT and MSM calculations occupy most of the execution time of the entire algorithm, due to the computational complexity of FFT being $O(N \log N)$ and the MSM being $O(N/\log N)$ (Do note that at this complexity, a lot of hardware buffers are required), as the circuit size increases, FFT/IFFT will consume more computing resources than MSM (MSM has more multipliers than FFT/IFFT for a single multiplication even if for the bit width).

It should be noted that compared to the traditional ZK algorithms of R1CS system, such as BCTV14 [2], Halo [3] and Fractal [4], current ZK algorithms without FFT based on R1CS system, such as Nova [15] and Spartan [26], has more efficient performance in proving, verifying, recursive performance and so on.

We believe that a ZK algorithm without FFT based on Plonkish constraint system will have better performance, although it is not considering optimal now, we will continue studying and researching this, as well as showcasing the hardware acceleration scheme of its main computing module, MSM, in advance.

1.3 Outline of the Paper

The remaining sections of this paper are summarized as follows:

- Section 2 mainly describes the design of the virtual machine of OlaVM, which is the basis for OlaVM to obtain smaller execution traces and state transition constraints;
- Section 3 mainly describes the design of OlaVM constraints, covering the design idea of OlaVM architecture;
- Section 4 mainly describes the data structure of execution traces and the structure description of related sub-traces;
- Section 5 mainly describes the underlying constraints of OlaVM, primarily constraints relating to the instruction set;
- Section 6 mainly describes the custom constraints supported in OlaVM, called Builtins;
- Section 7 mainly describes the account model used in OlaVM and the corresponding constraint logic;
- Section 8 mainly describes uint256 integer calculation logic supported in OlaVM;
- Section 9 mainly describes the principle of ZK without FFT algorithm;
- Section 10 mainly describes MSM algorithm and the design framework of FPGA-accelerated MSM;
- Section 11 mainly describes the key technologies used in the entire design of OlaVM.

2 Overview of VM

This section introduces the basic design ideas of OlaVM, primarily focusing on (1) why we choose a register-based VM instead of a stack-based VM; (2) how to design function calls on a register-based VM; (3) defining VM instructions on a finite field for better proving efficiency; (4) simplified VM instruction set design; (5) structure description of the VM instructions.

2.1 Register-based VM

Among virtual machine architectures, stack-based and register-based are common implementations. Stack-based implementation is used by e.g, Java VM [9] and EVM [1], and register-based virtual machine implementations by e.g, Lua VM [13] and Cairo VM [7]. These two types of virtual machines adopt the mode of simulating physical CPU to execute instructions, but there are significant differences in execution efficiency and the speed of compiling and generating code in this specific implementation process. Take a simple addition instruction as an example to illustrate the difference between two virtual machine architectures in the process of executing instructions. The storage data structure of a stack-based virtual machine adopts the operation form of FIFO (First in, First out). First, the operand is popped out of the stack, and then the operation is performed, and the operation result will be pushed into the stack again. The instructions involved are

```
POP (1)
POP (2)
ADD 1, 2, result
PUSH result
```

As can be seen in the addition operation, 4 instructions are required, which is relatively inefficient, however, the advantage is that you don't need to know the specific address of the operand, and you can get the operand by using instruction `POP`.

Register-based VMs do not have such an operand stack to store operands. Each instruction will contain the register address of the operand. The above example involves the following instruction

```
ADD R0, R1, R2
```

The obvious advantage of register-based virtual machines is that the addition can be completed with a single instruction. The disadvantage is that the address of the operand needs to be included in the instruction, which makes the instruction length longer.

Two virtual machine architectures each have their own advantages and disadvantages, further comparisons are shown in Table 1. Since our goal is to design a ZK-friendly virtual machine, we aim to achieve smaller execution traces of a given program with the trade-off increased instruction length. Furthermore, register-based virtual machines are easier to optimize during its execution of the instructions, hence, execution speed will see a significant improvement. Therefore we have opted for using a register-based virtual machine.

Stack-based VM vs. Register-based VM	**Comparison**
Number of instructions	Register-based $<$ Stack-based
Instruction length	Register-based $>$ Stack-based
Difficulty of the code generation	Register-based $>$ Stack-based
Instruction optimization	Register-based VM is easier to optimize code
Execution speed	Register-based VMs have a faster execution pace

Table 1: Stack-based VMs vs. Register-based VMs

2.2 Function Call by RPR

OlaVM adopts the register-based model. In order to easier simulate the effect of a function call stack it is necessary to save the next `pc` that currently calls the initiating function to a dedicated read-only memory, RPR (Return PC ROM), which is convenient to find the `pc` position that the last call should return to after the execution of the sub-function. The function call involves two instructions, `CALL` and `RET`, and a special register `fp`, which points to the address of RPR. Figure 4 shows a function call example.

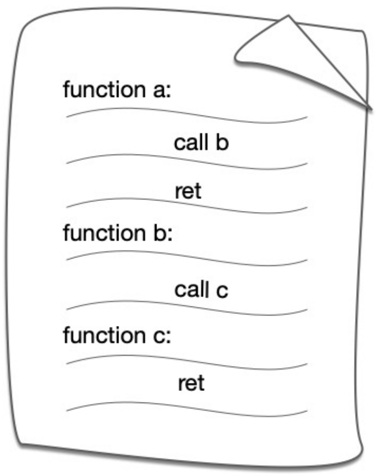


Figure 4: Function call example

The stack frame of the above function call is shown in Figure 5.

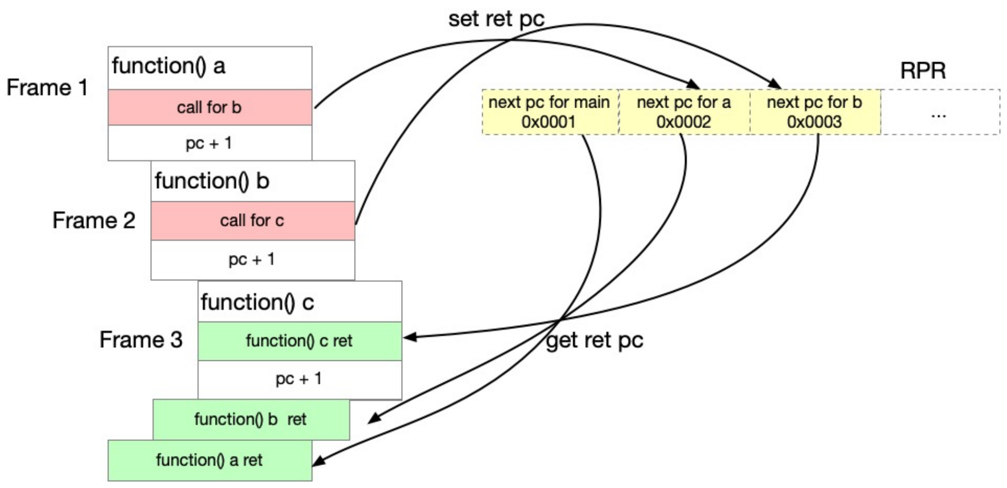


Figure 5: Function call frame

- RPR stores 3 return `pc` positions: `a return -> main pc`, `b return -> a pc`, `c return -> b pc`;
- When function `main` utilizes instruction `CALL` to call function `a`, the value of `pc` will be updated to the value saved in register `r0`;
- Set the value of ROM location pointed by the pointer stored in register `fp` to current `pc+1`, `fp+1` points to the next empty memory area.

- When the function a completes the execution, it will take the value pointing to the correct position of ROM through the pointer stored in register `fp`. Set current `pc` to this value, so the remaining program fragments can be executed when returning to function main;
- `fp` points to the position of `fp-1` in RPR;
- For nested calls of function `b` and function `c`, the change process of `pc` and `fp` is the same as above.

2.3 Word in Finite Field

In order to make the processing logic of the VM simple, to obtain the smallest basic constraint unit (which can constrain any instruction of the VM), we defined the VM over a finite field so that the Word of the VM is a field element. Therefore, the calculation logic of the VM only supports a few simple field operations, such as addition, subtraction (equivalent to adding an additive inverse), multiplication, and division (equivalent to multiplying a multiplicative inverse). Meanwhile, we have also implemented integer calculation logic in order to support integer operations (OlaVM supports uint256 integer calculations to strengthen compatibility with Ethereum). The relationship between integer calculation and field calculation is shown in Figure 6.

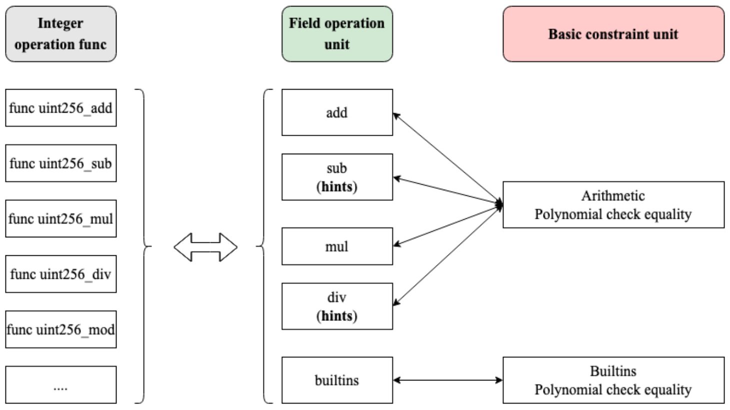


Figure 6: Field operation

As you can tell, OlaVM instruction set defined over a finite field are very concise, with only a few instructions, hence the number of constraint units of the circuit will be very small. In PSE scheme, the underlying constraint unit needs to process the logic of all EVM instructions, which adds up to roughly 100 instructions, vastly increasing its complexity. In addition to this, we have defined some builtins in order to support more complex logic. Thanks to this, given complex calculations that needs to be implemented, we don't need to utilize these VM instructions, we can pre-implement the corresponding custom constraint logic and call that directly.

Note: Current choice of finite field is BLS12-381 curve, which is planned to support uint256 integer calculations (refer to the design of StarkWare [30]).

2.4 Instruction Set

Whilst designing a ZKVM we also considered the complexity of verification, in addition to the execution efficiency of the VM itself, which differs from the traditional design principles of the instruction set. Purely pursuing efficient execution of instructions is not our goal, we have other fundamental design objectives, such as achieving short and concise verification complexity (occupying fewer trace cell units).

For example: when you have two instruction sets A and B, there is an "is_zero" instruction in instruction set A, which is used to judge "if x = 0, res = 1, otherwise, res = 0", and except for this instruction, instruction set B is identical to instruction set A. We denote the number of track units occupied by a CPU based on instruction set A to execute an instruction by a (for simplicity, assuming that all instructions consume the same number of units), and the number of track units required to execute one step based on instruction set B by b (where $a > b$ due to the added complexity of additional instructions). On the other hand, if a deterministic program is written based on instruction set A, it will execute k_A; and if it's written based on instruction set B, it will execute k_B ($k_B > k_A$, it is possible to utilize more instructions to execute "is_zero"). If $a \cdot k_A < b \cdot k_B$, instruction set A is more suitable for this program; conversely, instruction set B is more suitable. When deciding whether to intervene in an instruction, we should consider the relationship between the additional cost of each step (a/b) and the additional steps added (k_A/k_B).

Based on the principles above, the instruction sets we designed are as shown in Table 2, `flag` is a special register that holds the flag of `overflow` or `borrow`.

Type	Sub Type	Instruction	Operands	Description	flag
Logic	arithmetic	ADD	r0 r1 r2	Compute [r1] + [r2] and store the result in r0	overflow
		SUB	r0 r1 r2	Compute [r1] - [r2] and store the result in r0	borrow
		MUL	r0 r1 r2	Compute [r1] * [r2] and store the least significant bits of the result in r0	overflow
		DIV	r0 r1 r2	Compute [r1] / [r2] and store the result in r0	[r2] = 0
Flow	jump	JMP	r0	Set pc to [r0]	
		CJMP	r0	If flag = 1, set pc to [r0], else increment pc as usual	
	call	CALL	r0	Call instruction is usually used to make a call from one function to another Set pc to [r0], and store next pc in RPR Update the value saved in RPR position pointed to by fp to next pc, then fp = fp + 1	
	ret	RET		Update the value saved by pc to the next pc value of RPR pointed to by fp, then fp = fp - 1	
	end	END		END means the program is terminated and all registers are cleared	
Move	mov	MOV	r0 r1	Store [r1] in r0	
		MOVI	r0 imm0	Store immediate value imm0 in r0	
RAM	memory	MLOAD	r0 r1	Load the Word in memory begin with offset [r1] and store the Word into r0	
		MSTORE	r1 r0	Store [r0] at the start of memory where the offset is [r1]	
	storage	SLOAD	r0 r1	Use [r1] as key, query the value of the key from storage (the key will be hashed and prefixed with a splice) and store it in r0	
		SSTORE	r1 r0	Use [r1] as key, [r0] as value, store the key and the value into storage (the key will be hashed and prefixed with a splice)	
Input	input	READ	r0	Read the external input from VM freeInput, and store one Word into r0	

Table 2: Instruction set

Note: We haven't designed "comparison instruction" because comparison operations can be implemented directly through multiple Range Checks based on the particular design and parameter selection of OlaVM. Specific principles of this will be introduced in later sections.

2.5 Instruction Structure

Based on the VM instruction set defined in Section 2.4, the VM is able to execute programs. OlaVM proof system can constrain the semantics of each instruction according to the execution trace of the VM to ensure that the program has been executed correctly. However, we also need to ensure that the actual program being executed has not been maliciously modified, meaning, we have to ensure it executed the original, correct program. As for how to check that the instructions involved in the execution trace are all present in the original program fragment and if they are executed in the correct `pc` order, we've decided to compare whether or not each instruction in the execution trace is included in the original program. In order to do that, we encode each instruction, then checking if the encoding of the execution instruction is included in the encoding of the program instruction.

Instruction encoding form as shown in Figure 7 with each instruction encoding occupying 32 bits.

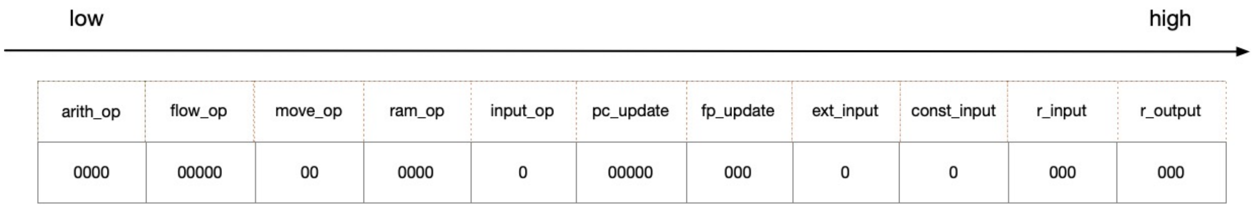


Figure 7: Instruction encoding

As shown in Table 7, the instruction will set the bit value of each used item to 1 and set the unused item to 0 when encoding. The specific meaning of each item is explained as follows:

- `arith_op` is the logic of arithmetic operation, which occupies 4 bits and is used to select the 4 instructions of `ADD`, `SUB`, `MUL` and `DIV`;
- `flow_op` is the control logic, which occupies 5 bits and is used to indicate the selection of control-related instructions;
- `mov_op` is the register move logic, which occupies 2 bits and is used to indicate the selection of `mov`-related instructions;
- `ram_op` is the operation logic of memory and storage, which occupies 4 bits and is used to indicate the selection of specific operations;
- `input_op` is the input logic, which occupies 1 bit and is used to indicate the selection of the program to read the input from outside;
- `pc_update`, occupying 5 bits, is used to indicate the relevant logic of `pc` update. `pc` can be updated through `flow_op` or updated normally, and the `pc_update` value of normal update is 00000;
- `fp_update`, occupying 3 bits, is used to indicate whether `fp` is updated. Instructions `CALL`, `RET` and `END` will update `fp`;
- `ext_input`, occupying 1 bit, is used to indicate whether there is an external input;
- `const_input`, occupying 1 bit, is used to indicate whether the instruction has immediate value input;
- `r_input`, occupying 3 bits, is used to indicate whether the instruction read input values from three registers `r0`, `r1` and `r2`;
- `r_output`, occupying 3 bits, is used to indicate whether the instruction will update the value of three registers `r0`, `r1` and `r2`.

The encoding forms of `arith_op` are

```
0001 -> ADD
0010 -> SUB
0100 -> MUL
```

```
1000 -> DIV
```

The encoding forms of `flow_op` are

```
00001 -> JMP
00010 -> CJMP
00100 -> CALL
01000 -> RET
10000 -> END
```

The encoding forms of `mov_op` are

```
01 -> MOV
10 -> MOVI
```

The encoding forms of `ram_op` are

```
0001 -> MSTORE
0010 -> MLOAD
0100 -> SSTORE
1000 -> SLOAD
```

The encoding form of `input_op` is

```
1 -> READ
```

The encoding forms of `pc_update` are

```
00001 -> indicates pc changes caused by JMP
00010 -> indicates pc changes caused by CJMP
00100 -> indicates pc changes caused by CALL
01000 -> indicates pc changes caused by RET
10000 -> indicates pc changes caused by END
```

The encoding forms of `pc_update` are

```
01 -> indicates the change of the fp register pointer caused by the CALL instruction
10 -> indicates the change of the fp register pointer caused by the RET instruction
```

The encoding forms of `fp_update` are

```
001 -> indicates the content change of the fp register caused by the CALL
    instruction
010 -> indicates the content change of the fp register caused by the RET instruction
100 -> indicates the content change of the fp register caused by the END instruction
```

The encoding forms of `ext_input` are

```
1 -> indicates the existence of external input data
```

The encoding form of `const_input` is

```
1 -> indicates that mov instruction has immediate value input
```

The encoding form of `r_input` is as follows

```
001/010/100 -> indicates the input of register r0, r1 and r2 respectively
```

The encoding form of `r_output` is as follows

```
001/010/100 -> indicates the output of register r0, r1 and r2 respectively
```

The description above contains instruction encoding forms, without immediate value encoding forms of the instruction. If an instruction involves immediate value, we will encode the immediate value after the instruction. At this time, `pc` is increased by the size of instruction each time, namely `pc = pc + instruction_size`.

With the above code rules, we can prove that the executed program fragment is indeed derived from the original program. In order to ensure the execution of the correct program sequence in case of program jumps, an additional `pc` code to the instruction is required to prove the program is executed in the correct sequence.

3 Overview of ZKVM Architecture

Our architectural approach is modular, designing the ZKVM to divide traces into main and sub traces. This section will describe the constraint list and consistency between main/sub traces, as well as between sub traces.

3.1 Modular Design

The modular design of OlaVM greatly improves the efficiency of the prover. Before we dive into a detailed description, let us briefly analyze the time consumption distribution chart of the VM executing a transaction, as show in Figure 8.

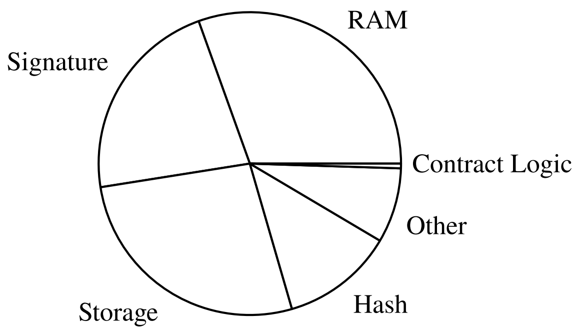


Figure 8: VM Execution proportions

As can be seen from Figure 8, the entire transaction execution process can be roughly divided into following categories: Contract Logic, RAM, Signature Verification, Storage Read/Write, Hash, and so on. If we prove these calculation types in a circuit, the entire constraint design will become quite complicated and difficult to handle, so we need to “divide and conquer” and finally processing them together, which is the source of the modular idea.

The execution trace of the VM (hereinafter referred to as the main trace for convenience) can be seen in Figure 9. With the approach mentioned above we divide the main trace table, resulting in several smaller tables, referred to as sub trace tables. Each sub trace table corresponds to a specific type of operation. Example given, the RAM sub trace table only contains RAM operations, and the signature sub trace table only contains signature operations. Therefore, we only need to design the corresponding constraint system for each module. The overall design framework is shown in Figure 9.

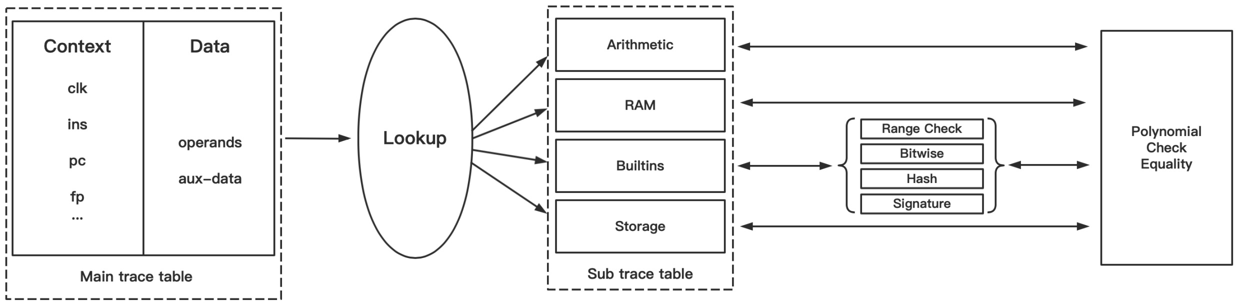


Figure 9: Architecture of ZKVM

The modular design approach allows for proofs among different modules to be executed in parallel, because the verification process is independent of each other, and the data correlation is only guaranteed by Permutation Argument and Lookup Argument, there are no data dependencies between each other. This is way more efficient than executing all the proofs in a single circuit, furthermore, multiple proofs corresponding to different sub-modules can be aggregated into a single proof, the idea of using recursive proofs can utilized to improve the system efficiency of proofs between different blocks.

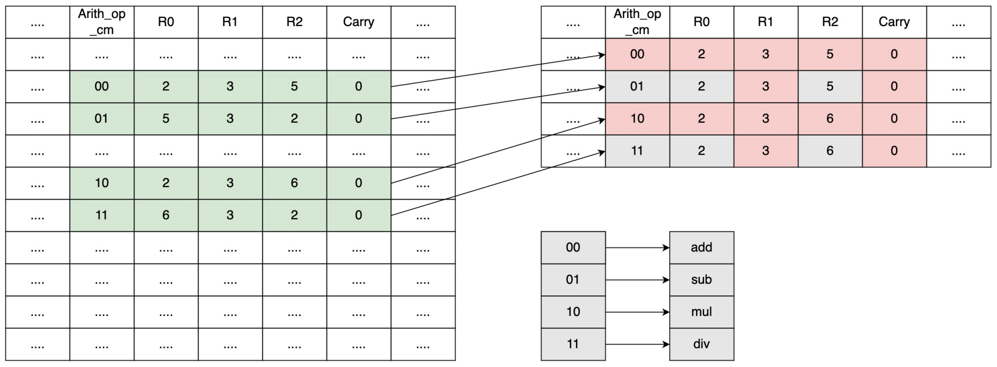
These common technologies are all applied in OlaVM architecture.

3.2 Consistency Check

We need to check the consistency between main/sub traces to ensure that the trace data is consistent.

3.2.1 Consistency between Main Trace and Sub Traces

According to the definition of main/sub trace tables, elements of sub trace tables should be taken from the main trace table. For example, the RAM sub trace table is composed of all RAM operations in main trace table, the signature sub trace table is composed of all signature verification operations in the main trace table. Now we need to make sure that these are executed correctly. Lookup Argument gracefully achieves this through successfully binding the data relationship between the main trace table and the corresponding sub trace tables.



....	Arith_op_cm	R0	R1	R2	Carry	....
....	....	....	....	....	....	....
....	00	2	3	5	0	....
....	01	5	3	2	0	....
....	....	....	....	....	....	....
....	10	2	3	6	0	....
....	11	6	3	2	0	....
....	....	....	....	....	....	....
....	....	....	....	....	....	....
....	....	....	....	....	....	....
....	....	....	....	....	....	....

....	Arith_op_cm	R0	R1	R2	Carry	....
....	00	2	3	5	0	....
....	01	2	3	5	0	....
....	10	2	3	6	0	....
....	11	2	3	6	0	....

00	add
01	sub
10	mul
11	div

Figure 10: Lookup between main trace and sub trace tables

Note: We have adjusted subtraction and division for convenience (The gray marked part on the right side of Figure 10), so that addition and subtraction can share one set of constraints and multiplication and division can share another set of constraints, which minimizes the overall constraints scale.

We utilize Lookup technology to ensure data consistency between the main trace table on the left and sub trace tables on the right, such as

$$\{\text{arith\_op\_cm}, \text{R0}, \text{R1}, \text{R2}, \text{Carry}\}_{\text{sub trace table}} \subset \{\text{arith\_op\_cm}, \text{R0}, \text{R1}, \text{R2}, \text{Carry}\}_{\text{main trace table}}$$

Utilizing the following constraints to verify the validity of the data in sub trace tables

$$f_{\text{arith\_op\_cm}}(x) \prod (3 - f_{\text{arith\_op\_cm}}(x)) (4 - f_{\text{arith\_op\_cm}}(x)) (f_{\text{R0}}(x) + f_{\text{R1}}(x) - (f_{\text{R2}}(x) + \text{Bound} \cdot f_{\text{Carry}}(x))) = 0$$
$$f_{\text{arith\_op\_cm}}(x) \prod (1 - f_{\text{arith\_op\_cm}}(x)) (2 - f_{\text{arith\_op\_cm}}(x)) (f_{\text{R0}}(x) \cdot f_{\text{R1}}(x) - (f_{\text{R2}}(x) + \text{Bound} \cdot f_{\text{Carry}}(x))) = 0$$

Note: The idea of Combined Selector is adopted here to reduce the number of selector polynomials. Please refer to Section 11 for detailed principles.

3.2.2 Consistency between Sub Traces

As mentioned in the previous section, we ensure data consistency between the main/sub trace tables through Lookup Arguments, however, data consistency must also be maintained between sub trace tables. For example, we store the result of instruction `ADD` in a result register, we must be able to ensure that the value written in this register is indeed the result of the execution of that previous instruction. Then we utilize Permutation Arguments to ensure data consistency between the arithmetic sub trace table and the RAM sub trace table.

During the execution of the entire VM, some of the operands of the instructions derive from intermediate values of the VM execution, when utilizing Permutation Argument to ensure data consistency as mentioned above, some derive from public inputs, such as the initial value and result of the program, we use instruction `EQ` to ensure that the instruction does operate the correct value.

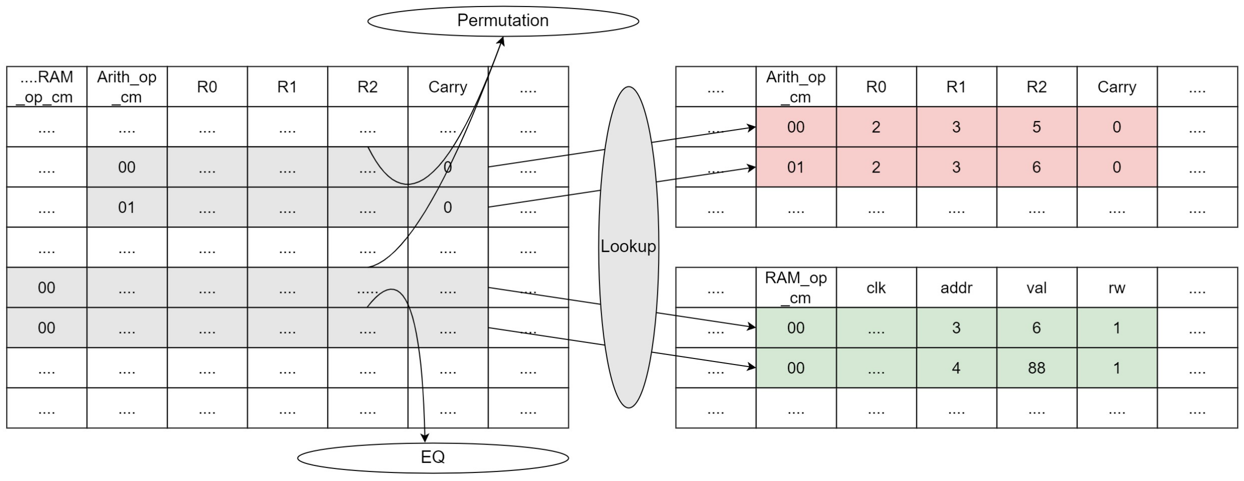


Figure 11: Consistency between sub trace tables

3.3 Constraint List

Because of the modular design, each module handles a specific function, resulting in specific constraints for each module, described in detail in later sections. Furthermore, we added context constraints, primarily used to constrain the change of context after the VM executes each instruction, this constraint applies to all instructions, hence applied to the main trace table.

According to the instruction structure we defined, it occupies 31 valid bits, enabling us to state the following definitions:

Arithmetic operations:

$$f_{\text{ADD}} = f_0,\ f_{\text{SUB}} = f_1,\ f_{\text{MUL}} = f_2,\ f_{\text{DIV}} = f_3,$$

Control flows:

$$f_{\text{JMP}} = f_4,\ f_{\text{CJMP}} = f_5,\ f_{\text{CALL}} = f_6,\ f_{\text{RET}} = f_7,\ f_{\text{END}} = f_8,$$

Moves:

$$f_{\text{MOV}} = f_9,\ f_{\text{MOVI}} = f_{10},$$

Memory and storage operations:

$$f_{\text{MLOAD}} = f_{11},\ f_{\text{MSTORE}} = f_{12},\ f_{\text{SLOAD}} = f_{13},\ f_{\text{SSTORE}} = f_{14},$$

External inputs:

$$f_{\text{INPUT}} = f_{15},\ f_{\text{EX\_INPUT}} = f_{23},$$

`pc`:

$$f_{\text{PC\_CALL}} = f_{16},\ f_{\text{PC\_RET}} = f_{17},\ f_{\text{PC\_JMP}} = f_{18},\ f_{\text{PC\_CJMP}} = f_{19},\ f_{\text{PC\_END}} = f_{20},$$

`fp` and `const`:

$$f_{\text{FP\_CALL}} = f_{21},\ f_{\text{FP\_RET}} = f_{22},\ f_{\text{CONST}} = f_{24}$$

Inputs:

$$f_{\text{INPUT\_R0}} = f_{25},\ f_{\text{INPUT\_R1}} = f_{26},\ f_{\text{INPUT\_R2}} = f_{27},$$

Outputs:

$$f_{\text{UPDATE\_R0}} = f_{28},\ f_{\text{UPDATE\_R1}} = f_{29},\ f_{\text{UPDATE\_R2}} = f_{30}.$$

3.3.1 Inst-decode Check

Upon execution, the VM decodes the instruction, then performs the corresponding operation according to the parsed flag. Therefore, we need to verify that the instruction executed by the VM is valid, i.e. the value of the instruction encoding is consistent with the corresponding flag. For simplicity, we will not allocate a separate column (the length is N) for all flags, however, we will use a virtual column to represent (the length is $32N$). Therefore we define $\widetilde{f_i} = \sum_{j=i}^{30} 2^{j-i} f_j$, so $\widetilde{f_0} = \sum_{j=0}^{30} 2^{j-0} f_j$, which represents the value of the first 31 bits of the instruction and $\widetilde{f_{31}} = 0$.

In order to obtain the value of f_i from $\{\widetilde{f_i}\}_{i=0}^{31}$, we utilize the equation

$$\begin{aligned}
\widetilde{f_i} - 2\widetilde{f_{i+1}} &= \sum_{j=i}^{30} 2^{j-i} f_j - 2 \sum_{j=i+1}^{30} 2^{j-i-1} f_j \\
&= \sum_{j=i}^{30} 2^{j-i} f_j - \sum_{j=i+1}^{30} 2^{j-i} f_j \\
&= f_i
\end{aligned}$$

The corresponding constraints are as follows:

Instruction:

$$\text{instruction} = \widetilde{f_0}$$

Bit:

$$(\widetilde{f_i} - 2\widetilde{f_{i+1}})(\widetilde{f_i} - 2\widetilde{f_{i+1}} - 1) = 0 \qquad \forall i \in [0, 31)$$

Last value is zero:

$$\widetilde{f_{31}} = 0$$

3.3.2 Context Update

If the instruction is `JMP` or `CJMP`, `pc` is updated to the specified location `dst_pc`, and register `fp` stores the address of RPR.

If the instruction is `CALL`, `pc` is updated to the value in `r0`, and the value of the location where `fp` points to the RPR is set to `pc+instruction_size`. `fp+1` points to an empty memory area.

If the instruction is `RET`, the function call ends, and `pc` is updated to the value saved by the RPR pointed to by `fp`, and `fp` is updated to the value of `fp-1` in RPR.

$$\begin{aligned}
f_{\text{pc}}(x\omega) - f_{\text{PC\_*JMP}}(f_{\text{dst\_pc}}(x) + f_{\text{PC\_CALL}}(f_{\text{dst\_pc}}(x)) + f_{\text{PC\_RET}}(f_{\text{fp}}(x)) + \\
(1 - f_{\text{PC\_*JMP}} - f_{\text{PC\_CALL}} - f_{\text{PC\_RET}})(f_{\text{pc}}(x) + \text{instruction\_size})) = 0
\end{aligned}$$

$$f_{\text{FP\_CALL}}(f_{\text{fp}}(x\omega) - (f_{\text{pc}}(x) + \text{instruction\_size})) = 0$$

3.3.3 Register Check

We have three general-purpose registers. Most of the instructions are based on registers. During execution of instructions, some register states may change while others stay the same, based on which the instruction is being executed. We define flags to indicate the register read/write state corresponding to the current instruction. We can constrain these general-purpose registers through flags:

$$\begin{aligned}
(1 - f_{\text{r0\_output}})(f_{\text{r0}}(x\omega) - f_{\text{r0}}(x)) = 0 \\
(1 - f_{\text{r1\_output}})(f_{\text{r1}}(x\omega) - f_{\text{r1}}(x)) = 0 \\
(1 - f_{\text{r2\_output}})(f_{\text{r2}}(x\omega) - f_{\text{r2}}(x)) = 0
\end{aligned}$$

The above constraints ensures that if the current instruction did not update the corresponding register, the value inside the register should stay the same, if the value of a certain register changed, we will ensure the validity of the value change on the corresponding sub trace table. For example, when executing the instruction `ADD r0,r1,r2`, we will verify that the value inside register `r1` and register `r2` remains the same, then check `r0=r1+r2` on the arithmetic sub trace table.

3.3.4 Execute Correct Program

The constraints described in previous sections guarantee the correctness of all the instructions executed by the VM, but there is no guarantee that the programs corresponding to these instructions are correct, i.e. corresponding to our original program. Therefore, we also need to verify that we have executed the correct program. Example given, we have the following program and store it in ROM:

pc	**Program Bytecodes**	**program_encode**
0	READ R0	0000000000000001000000001000010000
1	MOV R1 3	0000000001000000000000000100001001
2	ADD R0, R1, R2	0001000000000000000000000011000110
3	END	0000100000000000100000000000011111

Table 3: Simple VM example

Given an input, we get the execution trace table as shown in Table 4.

clk	program_encode	arith_op	flow_op	move_op	ram_op	input_op	pc_update	fp_update	ext_input	const_input	r_input	r_output	pc
0	0000000000000001000000001000010000	0000	00000	00	0000	1	00000	000	1	0	000	100	0
1	0000000001000000000000000100001001	0000	00000	10	0000	0	00000	000	0	1	000	010	1
2	0001000000000000000000000011000110	0001	00000	00	0000	0	00000	000	0	0	110	001	2
3	0000100000000000100000000000011111	0000	10000	00	0000	0	10000	000	0	0	000	111	3
4	…	…	…	…	…	…	…	…	…	…	…	…	…

Table 4: Execution trace table example

For simplification, the maximum `pc` is set to 4, we will utilize Lookup Argument and program encoding form shown below to ensure the execution of the correct program:

$$\text{trace\_program\_encode} = 2^2 \cdot \text{instruction\_encode} + \text{pc}$$

$$\{\text{trace\_program\_encode}\} \subset \{\text{ROM\_program\_encode}\}$$

Note:

1. The correctness of the instruction is verified by Inst-decode check;
2. The immediate value after the instruction does not need to be encoded, but `pc` needs to be encoded into the instruction, and the correctness of the immediate value after the instruction is guaranteed by the equality circuit.

4 Structure of Execution Trace Table

As an initial step, by dividing the instructions, data and other contents involved in the program execution trace into one main trace and multiple sub traces we are able to prove that the program is being executed correctly. There is a hierarchical relationship between these sub traces and the main trace. When there are instructions that cannot be processed in the main trace, these instructions will be executed in a sub trace. These sub traces are associated with the main trace through Lookup Argument. The complexity of polynomial representation of the main trace table has been reduced and the speed of proof generation is improved through this division of main and sub traces.

This section primarily introduces the main trace table and the RAM trace table, and others like arithmetic sub trace table, binary sub trace table and so on will be introduced in other sections.

In this section you'll find a description of the main trace table and RAM trace table, before that we've listed a few variables that will be referenced to. Some variables that have been described in Section 2.5 will not be repeated below.

- `clk`: The global clock of the program execution tracking. Every time an instruction is executed, `clk` automatically increases by 1;
- `instruction`: The encoded representation of an instruction;
- `pc`: A program counter. Under normal circumstances, the `pc` update logic is: `pc = pc + instruction_size`, in other cases, such as `flow_op`, the `pc` will be changed according to the specific logic;
- `fp`: Special register used in CALL and RET instructions, holds a pointer to the RPR memory location;
- `const`: Used to store the immediate value of the instruction;
- `r0`, `r1`, `r2`: Three general-purpose registers used for the storage of instruction input and output data;
- `carry`: Indicates whether there is a carry;
- `address`: The address operated by RAM instructions;
- `value`: The value operated by RAM instructions;
- `oldRoot`: World state representation of Verkle Tree before update;
- `newRoot`: World state representation of Verkle Tree after update.

4.1 Main Trace Table

In this section we are going to explain how the main trace table is generated, below you'll find a simplified execution program that we will be using as an example in order to walk you through the process.

```
READ R2(7)
MOV R1 3
ADD R0, R1, R2
END
```

This program would generate the main trace table as shown in Table 5.

clk	instruction	arith_op	flow_op	move_op	ram_op	input_op	pc_update	fp_update	ext_input	const_input	r_input	r_output
	tape	pc	fp	const	r0	r1	r2	carry	address	value	oldRoot	newRoot
0	00000000000000010000000010000100	0000	00000	00	0000	1	00000	000	1	0	000	100
	7	0	0x00	0	0	0	0	0	0x00	0	0x00	0x00
1	00000000010000000000000001000010	0000	00000	10	0000	0	00000	000	0	1	000	010
	0	1	0x00	3	0	0	0	0	0x00	0	0x00	0x00
2	00010000000000000000000000110001	0001	00000	00	0000	0	00000	000	0	0	110	001
	0	2	0x00	0	0	3	7	0	0x00	0	0x00	0x00
3	00001000000000001000000000000111	0000	10000	00	0000	0	10000	000	0	0	000	111
	0	3	0x00	0	10	3	7	0	0x00	0	0x00	0x00
4	...	...	...	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...	...	...	...

Table 5: Main trace table

The main trace table demonstrates the changes made in the main state machine. Starting from its initial state, a new state is generated after each instruction is executed. Whenever `clk` changes, the state of the last executed instruction will be related to the currently executed instruction. Taking `pc` column as an example, we regard each value appearing in `pc` column as the evaluation of a polynomial, e.g. ω^0 is 0, ω^1 is 1, ω^2 is 2 and so on. Similarly, each column can be regarded as a polynomial. Finally, we only need to combine these polynomials together to prove the correctness of the main trace table, thus proving that the program is executed correctly.

4.2 RAM Trace Table

In order to facilitate the correctness constraints for memory and storage operations, we utilize sub traces to represent all read and write operations for memory and storage. Since the state machines of memory and storage are very similar, we can combine them together, called RAM trace collectively, and distinguish between memory/storage operations through different `ram_ops`. Table 6 demonstrates the RAM trace table of a simple example program.

clk	**ram_op**	**Read**	**Write**	**Address**	**Value**
1	MSTORE	0	1	`0x80`	value1
2	SSTORE	0	1	`0x7f22`	value2
3	SLOAD	1	0	`0x7f22`	value2
4	MLOAD	1	0	`0x80`	value1
...	...	...	...	...	...

Table 6: RAM trace table

The above RAM trace table shows the memory read and write operations of one address and the storage read and write operations of another address. All instructions are listed in the order of `clk`, read and write operations are represented by separate columns. Considering that the VM will randomly read and write memory and storage during the execution process of reading and writing, it's difficult to make a unified circuit constraint in this case (it would require backtracking and recording the operation process continuosly for each tag, finally receiving all operations for a specific tag). This leads us to sorting the RAM trace table in a specific order, which is: `ram_op -> address -> clk`. The sorted RAM trace table is shown in Table 7.

clk	**ram_op**	**Read**	**Write**	**Address**	**Value**
1	MSTORE	0	1	`0x80`	value1
4	MLOAD	1	0	`0x80`	value1
2	SSTORE	0	1	`0x7f22`	value2
3	SLOAD	1	0	`0x7f22`	value2
...	...	...	...	...	...

Table 7: Sorted RAM trace table

Some additional circuit constraints required for the RAM trace table are shown below:

- The value of read and write columns must be of the Boolean type, read as 0, write as 1.
- `ram_op` operation must have only 4 types, namely MSTORE, MLOAD, SSTORE, SLOAD, among which MSTORE, MLOAD are classified as the same type of tags and tags of the same type are not sorted.

- RAM trace table must be sorted according to the rules

$$\text{ram\_op(MSTORE/MLOAD)}_i \leq \text{ram\_op(SSTORE/SLOAD)}_{i+1}$$

When tags are the same, the sorting rules are: $\text{address}_i \leq \text{address}_{i+1}$; when addresses are the same, the sorting rules are: $\text{clk}_i \leq \text{clk}_{i+1}$.
- For an operation of the same address, if it is a read operation, the value read must be the last value written to the address.

5 Circuit Constraints

5.1 Field Arithmetic Instruction Constraints

When describing directive constraints, we use the following notations:

- `pc`: The `pc` address stored in register `pc`;
- `fp`: The RPR address stored in register `fp`;
- `[fp]`: The `pc` address stored in RPR pointed by register `fp`;
- `r_src`: Register `src`;
- `r_dst`: Register `dst`;
- `r_flag`: Register `flag`;
- `[r_x]`: The value of register `x`;
- $\texttt{[r\_src}_i\texttt{]}$: The value of register `src` on clock i;
- $\texttt{[r\_src}_{i+1}\texttt{]}$: The value of register `src` on clock $i+1$;
- `mem[addr]`: The value of `addr` on memory;
- `storage[addr]`: The value of `addr` on storage;
- `imm0`: The immediate value.

5.1.1 ADD

Instruction: `ADD r_dst, r_src1, r_src2`

Defines field addition of the value of register `src1` and register `src2`, storing the result in register `dst`.

The VM arithmetic logic is

$$\texttt{[r\_dst] = [r\_src1] + [r\_src2]}$$

The constraints are

$$\texttt{[r\_dst}_{i+1}\texttt{]} - (\texttt{[r\_src1}_i\texttt{] + [r\_src2}_i\texttt{]}) = 0$$

$$\texttt{[r\_src1}_{i+1}\texttt{]} - \texttt{[r\_src1}_i\texttt{]} = 0$$

$$\texttt{[r\_src2}_{i+1}\texttt{]} - \texttt{[r\_src2}_i\texttt{]} = 0$$

The value of other registers remain unchanged.

5.1.2 MUL

Instruction: `MUL r_dst, r_src1, r_src2`

Defines field multiplication of the value of register `src1` and register `src2`, storing the result in register `dst`. The VM arithmetic logic is

$$\texttt{[r\_dst] = [r\_src1]} \cdot \texttt{[r\_src2]}$$

The constraints are

$$\begin{aligned}&\texttt{[r\_dst}_{i+1}\texttt{]} - \texttt{[r\_src1}_i\texttt{]} \cdot \texttt{[r\_src2}_i\texttt{]} = 0\\&\texttt{[r\_src1}_{i+1}\texttt{]} - \texttt{[r\_src1}_i\texttt{]} = 0\\&\texttt{[r\_src2}_{i+1}\texttt{]} - \texttt{[r\_src2}_i\texttt{]} = 0\end{aligned}$$

The value of other registers remain unchanged.

5.2 Flow Instruction Constraints

5.2.1 JMP

Instruction: `JMP r_dst`

The VM logic is

$$\texttt{pc = [r\_dst]}$$

The constraints are

$$\begin{aligned}&\texttt{pc}_{i+1} - \texttt{[r\_dst}_i\texttt{]} = 0\\&\texttt{[r\_dst}_{i+1}\texttt{]} - \texttt{[r\_dst}_i\texttt{]} = 0\end{aligned}$$

The value of other registers remain unchanged.

5.2.2 CJMP

Instruction: `CJMP r_dst`

The VM logic is

```
if [r_flag] == 1
  pc = [r_dst]
else
  pc = pc + 1
```

The constraints are

$$\begin{aligned}&\texttt{[r\_flag]}(\texttt{pc}_{i+1} - \texttt{[r\_dst}_i\texttt{]}) + (1 - \texttt{[r\_flag]})(\texttt{pc}_{i+1} - \texttt{pc}_i - 1) = 0\\&\texttt{[r\_dst}_{i+1}\texttt{]} = \texttt{[r\_dst}_i\texttt{]}\\&\texttt{[r\_flag}_{i+1}\texttt{]} = \texttt{[r\_flag}_i\texttt{]}\end{aligned}$$

The value of other registers remain unchanged.

5.2.3 CALL

Instruction: `CALL r_dst`

The VM logic is

```
[fp] = pc + 1
fp = fp + 1
pc = [r_dst]
```

The constraints are

$$\texttt{[fp}_i\texttt{]} - (\texttt{pc}_i + 1) = 0$$
$$\texttt{fp}_{i+1} - (\texttt{fp}_i + 1) = 0$$
$$\texttt{pc}_{i+1} - \texttt{[r\_dst}_i\texttt{]} = 0$$
$$\texttt{[r\_dst}_{i+1}\texttt{]} - \texttt{[r\_dst}_i\texttt{]} = 0$$

The value of other registers remain unchanged.

5.2.4 RET

Instruction: `RET`

The VM logic is

```
fp = fp - 1
pc = [fp]
```

The constraints are

$$\texttt{fp}_{i+1} - (\texttt{fp}_i - 1) = 0$$
$$\texttt{pc}_{i+1} - \texttt{[fp}_i\texttt{ - 1]} = 0$$

The value of other registers remain unchanged.

5.2.5 END

Instruction: `END`

The VM logic is

```
fp = 0
pc = 0
all registers set to 0
```

The constraints are

$$\texttt{fp}_{i+1} = 0$$
$$\texttt{pc}_{i+1} = 0$$
$$\texttt{[r\_x}_{i+1}\texttt{]} = 0,\ x \in \{0, 1, 2, \ldots\}$$

5.2.6 MOV

Instruction: `MOV r_dst, r_src`

The VM logic is

$$\texttt{[r\_dst] = [r\_src]}$$

The constraints are

$$\texttt{[r\_src}_{i+1}\texttt{]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

$$\texttt{[r\_dst}_{i+1}\texttt{]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

The value of other registers remain unchanged.

5.2.7 MOVI

Instruction: `MOVI r_dst, imm0`

The VM logic is

$$\texttt{[r\_dst] = imm0}$$

The constraint is

$$\texttt{[r\_dst}_{i+1}\texttt{]} = \texttt{imm0}$$

The value of other registers remain unchanged.

5.3 Memory Instruction Constraints

5.3.1 MLOAD

Instruction: `MLOAD r_dst, r_src`

The VM logic is

$$\texttt{[r\_dst] = mem[[r\_src]]}$$

The constraints are

$$\texttt{[r\_src}_{i+1}\texttt{]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

$$\texttt{[r\_dst}_{i+1}\texttt{]} - \texttt{mem[[r\_src}_{i}\texttt{]]} = 0$$

The value of other registers remain unchanged.

5.3.2 MSTORE

Instruction: `MSTORE r_dst, r_src`

The VM logic is

$$\texttt{mem[[r\_dst]] = [r\_src]}$$

The constraints are

$$\texttt{[r\_src}_{i+1}\texttt{]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

$$\texttt{mem[[r\_dst}_{i+1}\texttt{]]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

The value of other registers remain unchanged.

5.3.3 SLOAD

Instruction: `SLOAD r_dst, r_src`
The VM logic is

$$\texttt{[r\_dst] = store[[r\_src]]}$$

The constraints are

$$\texttt{[r\_src}_{i+1}\texttt{]} = \texttt{[r\_src}_{i}\texttt{]}$$

$$\texttt{[r\_dst}_{i+1}\texttt{]} = \texttt{store[[r\_src}_{i}\texttt{]]}$$

The value of other registers remain unchanged.

5.3.4 SSTORE

Instruction: `SSTORE r_dst, r_src`
The VM logic is

$$\texttt{store[[r\_dst]] = [r\_src]}$$

The constraints are

$$\texttt{[r\_src}_{i+1}\texttt{]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

$$\texttt{store[[r\_dst}_{i+1}\texttt{]]} - \texttt{[r\_src}_{i}\texttt{]} = 0$$

The value of other registers remain unchanged.

5.3.5 Memory Constraint

The trace of memory is shown in Table 8.

ctx	type	addr	clk	rw	v	d0	d1	t

Table 8: Memory trace table

In order to handle different contexts of contracts, we need to add context fields. For example, `ctx` of contract address 5 and contract address 6 are 5 and 6 respectively. The meaning of each column of the table is as follows:

- `ctx`: Context ID. Values in this column must increase monotonically but there can be gaps between two consecutive values of up to 2^{256}. Two consecutive values can be the same. Abbreviated to `c` in subsequent constraint expressions.
- `type`: Memory type, such as memory, storage. Currently we define 0 for memory and 1 for storage. Abbreviated to `p` in subsequent constraint expressions.
- `addr`: Memory/storage address. Values in this column must increase monotonically for a given context but there can be gaps between two consecutive values of up to 2^{256}. Two consecutive values can be the same. Abbreviated to `a` in subsequent constraint expressions.
- `clk`: Clock cycle at which the memory operation happened. Values in this column must increase monotonically for a given context and memory address but there can be gaps between two consecutive values of up to 2^{256}.
- `rw`: Read/write operation flag, 0 for read and 1 for write.

- `v`: Field element stored at a given context/address/clock cycle after memory/storage operation.
- `d0` and `d1`: Lower and upper 128 bits of the delta between two consecutive context IDs, addresses, or clock cycles.
- `t`: According to the different meanings of the state changes of `ctx`, `adddr` and `clk`, the priorities are as follows:
  1. If `ctx` changes, `d0` represents the lower 128 bits of the difference of `ctx`, and `t` represents the inverse of the difference of `ctx`.
  2. If `ctx` remains unchanged but type changes, `d0` represents the lower 128 bits of the difference of `type`, and `t` represents the inverse of the difference of type.
  3. If `ctx` and `type` remain unchanged, but `addr` changes, `d0` represents the lower 128 bits of the difference of address, and `t` represents the inverse of the difference of address.
  4. If `ctx`, `type` and `addr` remain unchanged, but `clk` changes, `d0` represents the lower 128 bits of the difference of `clk` minus 1, and `t` represents the inverse of the difference minus 1 of `clk`, namely $(\texttt{clk}_{i+1} - \texttt{clk}_i - 1)^{-1}$.

First we define n_0, n_1 and n_2

$$n_0 = (c_{i+1} - c_i) \cdot t_{i+1}$$

$$n_1 = (p_{i+1} - p_i) \cdot t_{i+1}$$

$$n_2 = (a_{i+1} - a_i) \cdot t_{i+1}$$

Constrain the change of `ctx` to be consistent with priority 1, and then get the following constraints 1 and 2:

$$n_0^2 - n_0 = 0$$

$$(1 - n_0)(c_{i+1} - c_i) = 0$$

Constrain the change of `type` to be consistent with priority 2, and then get the following constraints 3 and 4:

$$(1 - n_0)(n_1^2 - n_1) = 0$$

$$(1 - n_0)(1 - n_1)(p_{i+1} - p_i) = 0$$

Constrain the change of `addr` to be consistent with priority 3, and then get the following constraints 5 and 6:

$$(1 - n_0)(1 - n_1)(n_2^2 - n_2) = 0$$

$$(1 - n_0)(1 - n_1)(1 - n_2)(a_{i+1} - a_i) = 0$$

Satisfy the constraint of priority 1 $n_0 = 1$ and then get the following constraint 7:

$$n_0\left((c_{i+1} - c_i) - (2^{128} \cdot \texttt{d1}_{i+1} + \texttt{d0}_{i+1})\right) = 0$$

Satisfy the constraint of priority 2 $n_0 = 0, n_1 = 1$ and then get the following constraint 8:

$$(1 - n_0)n_1\left((p_{i+1} - p_i) - (2^{128} \cdot \texttt{d1}_{i+1} + \texttt{d0}_{i+1})\right) = 0$$

Satisfy the constraint of priority 3 $n_0 = 0, n_1 = 0, n_2 = 1$ and then get the following constraint 9:

$$(1 - n_0)(1 - n_1)n_2\left((a_{i+1} - a_i) - (2^{128} \cdot \texttt{d1}_{i+1} + \texttt{d0}_{i+1})\right) = 0$$

Satisfy the constraint of priority 4 $n_0 = 0, n_1 = 0, n_2 = 0$ and then get the following constraint 10:

$$(1 - n_0)(1 - n_1)(1 - n_2)\left((\texttt{clk}_{i+1} - \texttt{clk}_i - 1) - (2^{128} \cdot \texttt{d1}_{i+1} + \texttt{d0}_{i+1})\right) = 0$$

Changes of `ctx` or `type` or `addr` will cause the memory value under the new `clk` to be initialized to 0, resulting in constraint 11:

$$\left(n_0 + (1 - n_0)(n_1 + (1 - n_1)n_2)\right)v_i = 0$$

Changes of `ctx`, `type` or `addr` will cause the `rw` under the new `clk` to be initialized to 1, resulting in constraint 12:

$$\big(n_0 + (1-n_0)(n_1 + (1-n_1)n_2)\big)(1-\texttt{rw}) = 0$$

When `ctx`, `type` and `addr` remain unchanged and `rw` is the `read` type, the old value of the new `clk` cycle of memory is equal to the new value of the previous `clk` cycle. The combination $(\texttt{ctx}, \texttt{type}, \texttt{addr}, \texttt{clk})$ is unique, then obtain constraint 13:

$$(1-n_0)(1-n_1)(1-n_2)(1-\texttt{rw})(v_{i+1}-v_i) = 0$$

6 Builtins

From a provers perspective, the difficulty of different function modules in a ZKVM varies, hence, we are utilizing a modular architecture, decoupling the different modules to the maximum extent possible. When dealing with complex modules, the circuit scale vastly grows in size increasing the cost of the proof, in addition to potentially not being frequently used, creating inefficiencies. We utilize builtins to enhance this. Through Plookup technology builtins we are able to store the circuit overhead upon usage, enabling us to flexibly handling the overhead problem of complex circuits.

6.1 Range Check

In order to implement Range Check we are utilizing Lookup Argument technology. Please refer to Section 11 for detailed design. The basic fixed table is an 8-bit lookup table, i.e. 0-255. For 8-bit numbers, the validity is checked using Plookup. The utilization of Plookup check is shown in Figure 12.

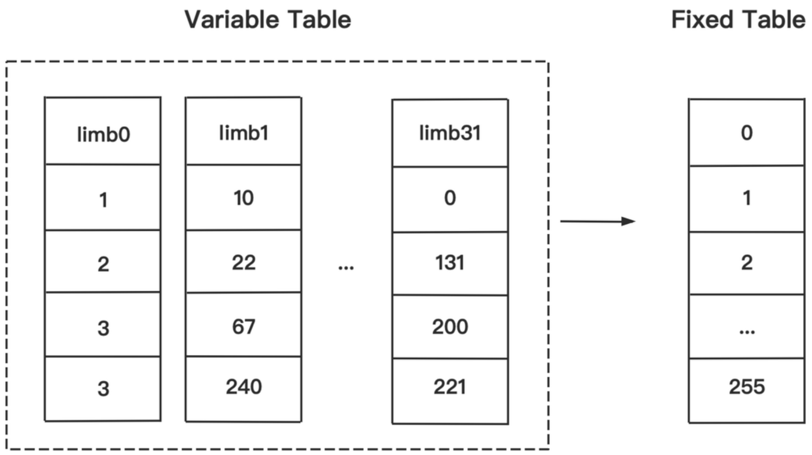


Figure 12: Fixed/Variable lookup table

For numbers greater than 8 bits, break them down to multiple 8-bit limbs and perform Plookup respectively, and then check the constraints of sum and limbs. Take 256-bit as an example, $\text{integer256} = \text{limb}_0 + \text{limb}_1 \cdot 2^8 + \text{limb}_2 \cdot (2^8)^2 + \text{limb}_3 \cdot (2^8)^3 + \cdots + \text{limb}_{31} \cdot (2^8)^{31}$.

6.2 Bitwise

Lookup Arguments are also utilized to implement most of the bitwise operations. The basic lookup table is based on 8-bit, containing three columns, two inputs and one bitwise output, as shown in Table 9.

	lhs	**rhs**	**out**
BitwiseAND table	lhs = 8-bit	rhs = 8-bit	lhs AND rhs
BitwiseOR table	lhs = 8-bit	rhs = 8-bit	lhs OR rhs
BitwiseXOR table	lhs = 8-bit	rhs = 8-bit	lhs XOR rhs

Table 9: Bitwise lookup table

Bitwise operations larger than 8 bits are broken down into multiple 8-bit operations with Plookup ran on each operation. Take the 32-bit AND of two numbers as an example,

$$\begin{aligned} a \wedge b &= (a_0 + a_1 \cdot 2^8 + a_2 \cdot (2^8)^2 + a_3 \cdot (2^8)^3) \wedge (b_0 + b_1 \cdot 2^8 + b_2 \cdot (2^8)^2 + b_3 \cdot (2^8)^3) \\ &= (a_0 \wedge b_0) + (a_1 \wedge b_1) \cdot 2^8 + (a_2 \wedge b_2) \cdot (2^8)^2 + (a_3 \wedge b_3) \cdot (2^8)^3 \end{aligned}$$

Therefore, for two n-bit numbers a and b, let $m = n/8$, and a_i, b_i are their limbs, then

$$a \wedge b = \sum_{i=0}^{m-1} (2^{8i} \cdot (a_i \wedge b_i))$$

$$a \vee b = \sum_{i=0}^{m-1} (2^{8i} \cdot (a_i \vee b_i))$$

$$a \oplus b = \sum_{i=0}^{m-1} (2^{8i} \cdot (a_i \oplus b_i))$$

The implementation of NOT is slightly different. The NOT to constrain a is

$$\neg a + a = 2^{256} - 1$$

6.3 Sinsemilla Hash

We use Halo2 [11] Sinsemilla Hash scheme.

For detailed design of the hash algorithm and the parameters used below, please refer to Section 11. Following is the main calculation process of Sinsemilla:

1. Set the initial value $\text{Acc}_0 = Q$
2. Perform loop calculation of i from 0 to $n-1$:

$$\text{Acc}_{i+1} = (\text{Acc}_i + P_{m_{i+1}}) + \text{Acc}_i$$

Now, let $R_i = \text{Acc}_i + P_{m_{i+1}}$, then $\text{Acc}_{i+1} = R_i + \text{Acc}_i$ then according to previous definition, we have

$$\lambda_{1,i} = \frac{y_{Acc,i} - y_{P,i}}{x_{Acc,i} - x_{P,i}}$$
$$x_{R,i} = \lambda_{1,i}^2 - x_{\text{Acc},i} - x_{P,i}$$
$$y_{R,i} = \lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - y_{\text{Acc},i}$$
$$\lambda_{2,i} = \frac{y_{\text{Acc},i} - y_{R,i}}{x_{\text{Acc},i} - x_{R,i}}$$
$$x_{\text{Acc},i+1} = \lambda_{2,i}^2 - x_{\text{Acc},i} - x_{R,i}$$
$$y_{\text{Acc},i+1} = \lambda_{2,i}(x_{\text{Acc},i} - x_{\text{Acc},i+1}) - y_{\text{Acc},i}$$

So, we have constrains as in Table 10.

Degree	Constraints	Notes
3	$q_{\text{incomplete-add}} \cdot (x_{R,i} + x_{\text{Acc},i} + x_{P,i} - \lambda_{1,i}^2) = 0$	$x_{R,i} = \lambda_{1,i}^2 - x_{\text{Acc},i} - x_{P,i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - (y_{\text{Acc},i} + y_{R,i})) = 0$	$y_{R,i} = \lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - y_{\text{Acc},i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{1,i}(x_{\text{Acc},i} - x_{P,i}) - (y_{\text{Acc},i} - y_{P,i})) = 0$	$\lambda_{1,i} = (y_{\text{Acc},i} - y_{P,i})/(x_{\text{Acc},i} - x_{P,i})$
3	$q_{\text{incomplete-add}} \cdot (x_{\text{Acc},i+1} + x_{\text{Acc},i} + x_{R,i} - \lambda_{2,i}^2) = 0$	$x_{\text{Acc},i+1} = \lambda_{2,i}^2 - x_{\text{Acc},i} - x_{R,i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{2,i}(x_{\text{Acc},i} - x_{\text{Acc},i+1}) - (y_{\text{Acc},i} + y_{\text{Acc},i+1})) = 0$	$y_{\text{Acc},i+1} = \lambda_{2,i}(x_{\text{Acc},i} - x_{\text{Acc},i+1}) - y_{\text{Acc},i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{2,i}(x_{\text{Acc},i} - x_{R,i}) - (y_{\text{Acc},i} - y_{R,i})) = 0$	$\lambda_{2,i} = (y_{\text{Acc},i} - y_{R,i})/(x_{\text{Acc}_i} - x_{R,i})$

Table 10: Sinsemilla Hash constraints

$$\lambda_{2,i} = \frac{y_{\text{Acc},i} - y_{R,i}}{x_{\text{Acc},i} - x_{R,i}}$$
$$\implies y_{\text{Acc},i} - y_{R,i} = \lambda_{2,i}(x_{\text{Acc},i} - x_{R,i})$$
$$y_{R,i} = \lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - y_{\text{Acc},i}$$
$$\implies y_{\text{Acc},i} - (\lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - y_{\text{Acc},i}) = \lambda_{2,i}(x_{\text{Acc},i} - x_{R,i})$$
$$\implies 2y_{\text{Acc},i} = (\lambda_{1,i} + \lambda_{2,i})(x_{\text{Acc},i} - x_{R,i})$$

Therefore, the check for $\lambda_{2,i} = (y_{\text{Acc}_i} - y_{R,i})/(x_{\text{Acc}_i} - x_{R,i})$ can be replaced by

$$q_{\text{incomplete-add}} \cdot (\lambda_{2,i}(x_{\text{Acc},i} - x_{R,i}) - (y_{\text{Acc},i} - y_{R,i})) = 0$$

then

$$q_{\text{incomplete-add}} \cdot ((\lambda_{2,i} + \lambda_{1,i})(x_{\text{Acc},i} - x_{R,i}) - 2y_{\text{Acc},i}) = 0$$

We haven't used $y_{R,i}$ of $R_i = \{x_{R,i}, y_{R,i}\}$ in the second step of checking for $\text{Acc}_{i+1} = R_i + \text{Acc}_i$. Therefore, remove the constraint on $y_{R,i}$ in the first step and the entire constraint develop into what is displayed in Table 11.

Degree	Constraints	Notes
3	$q_{\text{incomplete-add}} \cdot (x_{R,i} + x_{\text{Acc},i} + x_{P,i} - \lambda_{1,i}^2) = 0$	$x_{R,i} = \lambda_{1,i}^2 - x_{\text{Acc},i} - x_{P,i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{1,i}(x_{\text{Acc},i} - x_{P,i}) - (y_{\text{Acc},i} - y_{P,i})) = 0$	$\lambda_{1,i} = (y_{\text{Acc},i} - y_{P,i})/(x_{\text{Acc},i} - x_{P,i})$
3	$q_{\text{incomplete-add}} \cdot (x_{\text{Acc},i+1} + x_{\text{Acc},i} + x_{R,i} - \lambda_{2,i}^2) = 0$	$x_{\text{Acc},i+1} = \lambda_{2,i}^2 - x_{\text{Acc},i} - x_{R,i}$
3	$q_{\text{incomplete-add}} \cdot (\lambda_{2,i}(x_{\text{Acc},i} - x_{\text{Acc},i+1}) - (y_{\text{Acc},i} + y_{\text{Acc},i+1})) = 0$	$y_{\text{Acc},i+1} = \lambda_{2,i}(x_{\text{Acc},i} - x_{\text{Acc},i+1}) - y_{\text{Acc},i}$
3	$q_{\text{incomplete-add}} \cdot ((\lambda_{2,i} + \lambda_{1,i}(x_{\text{Acc},i} - x_{R,i}) - 2y_{\text{Acc},i}) = 0$	$\lambda_{2,i} = (y_{\text{Acc},i} - y_{R,i})/(x_{\text{Acc},i} - x_{R,i})$

Table 11: Simplified Sinsemilla Hash constraints

6.4 Comparison

Range Check is used to perform GTE comparison, using Cairo's scheme [7].

On a circuit level, we only need to support 3 comparisons: EQ, GTE and NEQ. We will perform a 256-bit Range Check for two numbers to be compared, and then perform a 256-bit Range Check for the difference between two numbers.

EQ checks the equality of two 256-bit numbers a_{256}, b_{256}. The constraint is $a_{256} - b_{256} = 0$

NEQ checks the inequality of two 256-bit numbers. The constraint is $(a - b) \cdot (a - b)^{-1} = 1$

GTE checks whether two 256-bit numbers a and b satisfy $a \geq b$.

First, use Range Check to constrain each number in 256 bits, and then compare them according to the difference. If the difference exceeds 256 bits, it means that they don't satisfy $a \geq b$. The constraint pseudo code is

```
range_check(integer256_a, range256bit)
range_check(integer256_b, range256bit)
let diff = integer256_a - integer256_b
range_check(diff, range256bit)
```

We can utilize GTE and NEQ bitwise operations to perform other comparisons, such as Table 12.

Application level	Circuit level
$a \geq b$	a GTE b
$a \leq b$	b GTE a
$a > b$	(a GTE b) && (a NEQ b)
$a < b$	(b GTE a) && (b NEQ a)

Table 12: Comparison level

6.5 Signature

We support two different signature schemes, ECDSA [14] and Schnorr [23].

6.5.1 ECDSA Signature

The hash function we choose is Sinsemilla Hash, which is the same function used in signature generation. With the following notations:

- The signature is (r, s).
- m is the message to verify.
- $Q(Q_x, Q_y)$ is the public-key curve point.
- h is the hash result of m.
- G is the elliptic curve base point.
- n is the order of G.
- $\mathcal{O}$ is the identity element.
- L_n is the bit length of n.

The constraints are the following:

1. Check that Q is not equal to $\mathcal{O}$.

2. Check that Q lies on the curve, which means: $Q_y^2 = Q_x^3 + \alpha Q_x + \beta$.
3. Check that $nQ = \mathcal{O}$.
4. Check that r and s are in $[1, n-1]$.
5. Let z be the leftmost L_n bits of h (we use Sinsemilla Hash builtin circuit to check $h \equiv \text{HASH}(m)$).
6. Calculate $u_1 = zs^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$.
7. Calculate curve point $(x_1, y_1) = u_1 G + u_2 Q$, check $(x_1, y_1) \neq \mathcal{O}$.
8. Check that $r \equiv x_1 \pmod{n}$.

6.5.2 Schnorr Signature

Schnorr signature is known for its simplicity, Bitcoin used Schnorr signature as well. The hash function we choose is Sinsemilla Hash. With the following notations:

- (s, e) is the signature.
- m is the message to verify, it is represented as list of finite bit strings.
- h is the hash result of m.
- g is the generator.
- y is the public verification key.

The constraints are the following:

1. Check $r_v = g^s y^e$.
2. Check $e_v = \text{Hash}(r_v \,||\, m)$, where $||$ denotes concatenation and r_v is represented as a bit string (Like above, we use Sinsemilla Hash builtin circuit to check it).
3. Check that $e_v \equiv e$.

Schnorr signature also supports Multi-Signatures [17], which could provide more security. Given

- A list of public keys $L = \{X_1, X_2, \ldots, X_n\}$.
- An aggregated public key $X = \prod_{i=1}^{n} H(L, X_i) X_i$.
- A message m.
- A signature (R, s).
- A list of random numbers $R_i = g^{r_i}$ and $R = \prod_{i=1}^{n} R_i$.

The constraints are the following:

1. Check $c = \text{Hash}(X, R, m)$ (As always, this can be done in Hash builtin circuit).
2. Check $u = cX$.
3. Check $sG \equiv R + u$.

7 Storage Module

7.1 Verkle Tree Structure

We use Verkle Tree to store storage data. The width of Verkle Tree is 256, and the nodes are divided into two types: InternalNode and LeafNode. We use KZG commitments for vector commitments. The order $n = 2^{251} + 17 \cdot 2^{192} + 1$ of the curve we selected is a 252-bit prime number, so we can only make safe commitments for 251-bit values. Each key consists of 32-bytes. The structure of Verkle Tree is shown in Figure 13. For LeafNode, each value V_i is 32-bytes. To

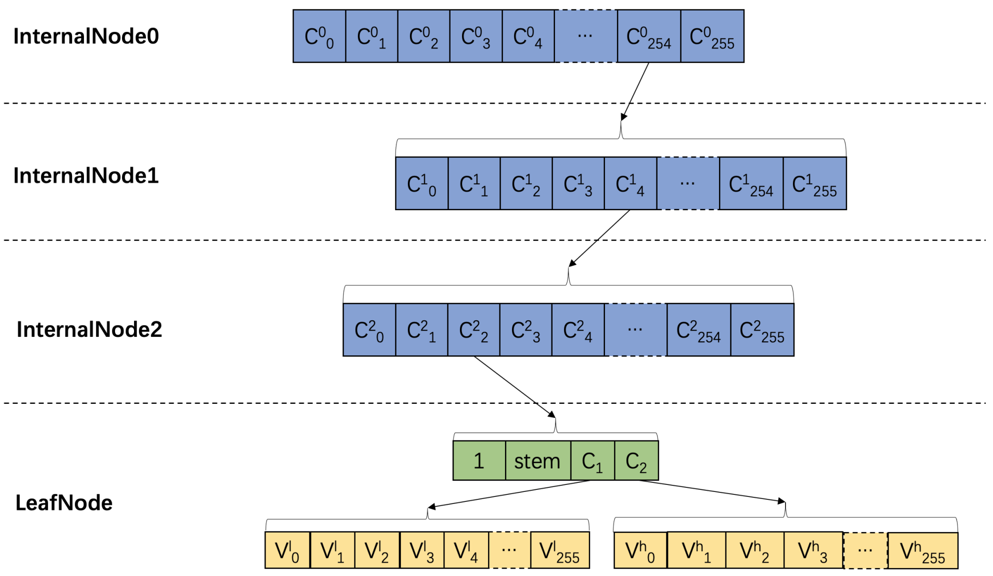

InternalNode0
C^0_0 C^0_1 C^0_2 C^0_3 C^0_4 ... C^0_{254} C^0_{255}
InternalNode1
C^1_0 C^1_1 C^1_2 C^1_3 C^1_4 ... C^1_{254} C^1_{255}
InternalNode2
C^2_0 C^2_1 C^2_2 C^2_3 C^2_4 ... C^2_{254} C^2_{255}
LeafNode
1 stem C_1 C_2
V^l_0 V^l_1 V^l_2 V^l_3 V^l_4 ... V^l_{255}
V^h_0 V^h_1 V^h_2 V^h_3 ... V^h_{255}

Figure 13: Structure of Verkle Tree

make commitments to vector $\{V_i\}$, we split each value V_i into two 16-byte values V_i^l and V_i^h, and make commitments to $\{V_i^l\}$ and $\{V_i^h\}$ as $C_1 = \text{Commit}(V_0^l, V_1^l, V_2^l, \ldots, V_{255}^l)$, $C_2 = \text{Commit}(V_0^h, V_1^h, V_2^h, \ldots, V_{255}^h)$ respectively. It should especially be noted that the node will not be deleted at the data level, but use the 128th bit of V_i^l as the marker bit of the deleted node. If the 128th bit of V_i^l is set to 1, it means the node has been deleted. The commitment of LeafNode is composed of flag bit 1, stem and two sub-commitments C_1, C_2, i.e.

$$C = \text{Commit}(1, \text{stem}, C_1, C_2)$$

The commitment to InternalNode is relatively simpler that the commitment to empty node is 0 and if the node is non-empty, just directly make commitments to all 256 commitment values, namely

$$C = \text{Commit}(C_0, C_1, C_2, \ldots, C_{255})$$

7.2 Verkle Tree Vector Commitment

Define that the random number selected by SRS is R. The Lagrange basis for evaluating polynomials of 256 points is δ_i, i.e.

$$\delta_i = \prod_{\substack{k=0 \\ k \neq i}}^{255} \frac{R-k}{i-k}$$

Commitment to the InternalNode:

$$C = \text{Commit}(C_0, C_1, C_2, \ldots, C_{255}) = \sum_{i=0}^{255} C_i \delta_i$$

Two child commitments to the LeafNode:

$$C_1 = \text{Commit}(V_0^l, V_1^l, V_2^l, \ldots, V_{255}^l) = \sum_{i=0}^{255} V_i^l \delta_i$$

$$C_2 = \text{Commit}(V_0^h, V_1^h, V_2^h, \ldots, V_{255}^h) = \sum_{i=0}^{255} V_i^h \delta_i$$

Commitment to the LeafNode:

$$C = \text{Commit}(1, \text{stem}, C_1, C_2) = \delta_0 + \text{stem} \cdot \delta_1 + C_1\delta_2 + C_2\delta_3 + \sum_{i=3}^{255} C_i\delta_i$$

$$C_i = 0 \quad \forall i \geq 3$$

7.3 Design of Verkle Tree Constraints

Our constraints are designed for a key change, when the key changes, we constrain the node changes of the corresponding path. When the value of the constraint key changes from V_1 to V_2, the root of Verkle Tree should change from root_1 to root_2.

- the value of key V_1 corresponds to the root of storage as root_1;
- the value of key V_2 corresponds to the root of storage as root_2.

Since we do not delete keys at data level, we're adding a "deletion flag", Verkle Tree modification actually has only two kinds of operations, update and insert.

7.3.1 Constraints for Node Update

Figure 14 shows the value update constraint schematic.

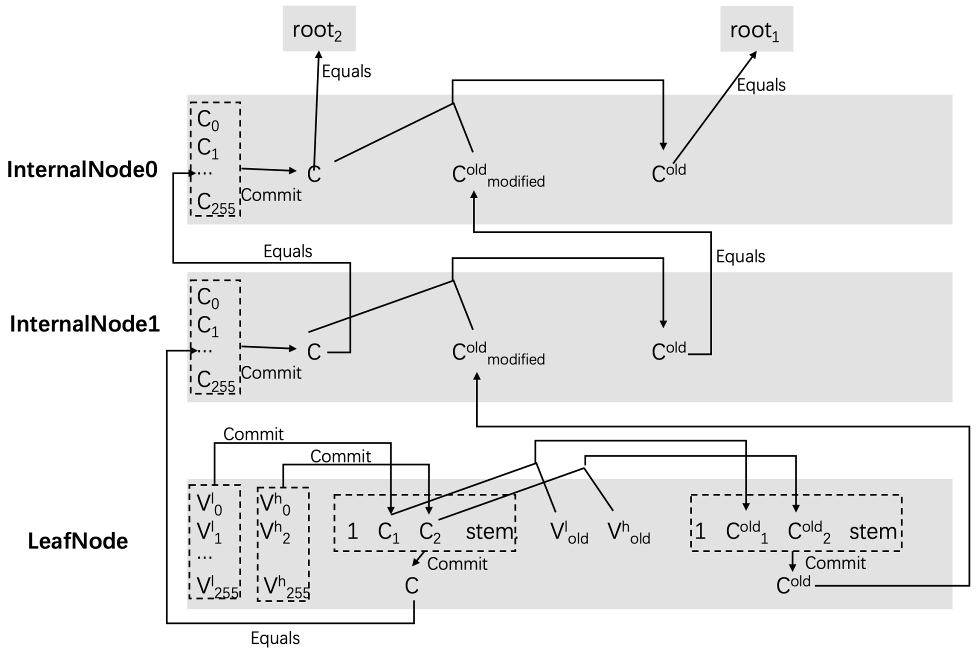


Figure 14: Verkle Tree Update Constraint Schematic

As for the path related to the updated value, trace table records the completed updated value, i.e. $\{V_i^l\}, \{V_i^h\}$ of LeafNode and $\{C_i\}$ of InternalNode. The value before update is represented as V_{old}^l and V_{old}^h in LeafNode, $C_{\text{modified}}^{\text{old}}$ in InternalNode.

In LeafNode, $\{V_i^l\}$ is committed to C_1 and $\{V_i^h\}$ is committed to C_2. According to the commitment protocol, we can compute the pre-update commitment C_1^{old} using C_1 and V_{old}^l, and compute the pre-update commitment C_2^{old} using C_2 and V_{old}^h. Finally make a commitment to LeafNode to get the updated commitment C and the commitment before update C^{old}, and also constrain C to be equal to the commitment of the change line of parent, C^{old} to be equal to the pre-change commitment of the change line of parent.

In InternalNode, commit $\{C_i\}$ to C, and then the commitment protocol utilizes C and $C_{\text{modified}}^{\text{old}}$ to calculate the pre-updated commitment C^{old} of this node. Constrain them to be equal to the corresponding values in parent. If the node is on the top-level, constrain $C = \text{root}_2$, $C^{\text{old}} = \text{root}_1$.

7.3.2 Constraints for Node Insert

We have three cases for inserting nodes:

1. The node is inserted at the position of EmptyNode in InternalNode.
2. The node is inserted into the LeafNode position of the InternalNode, and the newly inserted LeafNode has the same stem as the LeafNode at the original position.
3. The node is inserted into the LeafNode position of the InternalNode, and the newly inserted LeafNode has the different stem from the LeafNode at the original position.

There are no differences between the first/second cases and the constraint logic of node update, which will not be discussed again here. For the third case of inserting a node, the logic of inserting is shown in Figure 15.

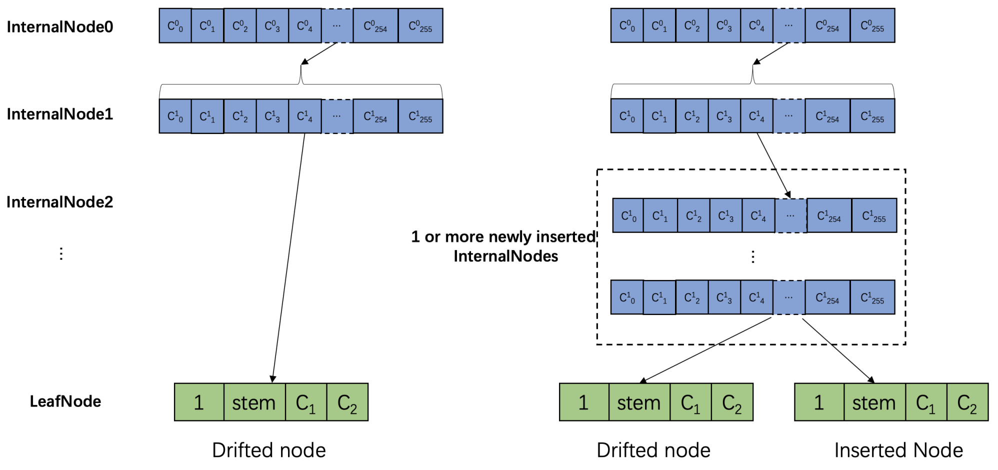


Figure 15: Verkle Tree Update Constraint Schematic

The position of the newly inserted node in InternalNode has been occupied by a LeafNode, called DriftedNode. Meanwhile, the stem of the newly inserted node is different from that of the original DriftedNode. At this time, one or more InternalNodes need to be added until the paths of the newly inserted LeafNode and the DriftedNode are on different nodes of the InternalNode. Finally, insert the DriftedNode and the newly inserted LeafNode.

In this case, we are supposed to perform constrains to DriftedNode in addition to insert full paths of the node when constraining as Figure 16.

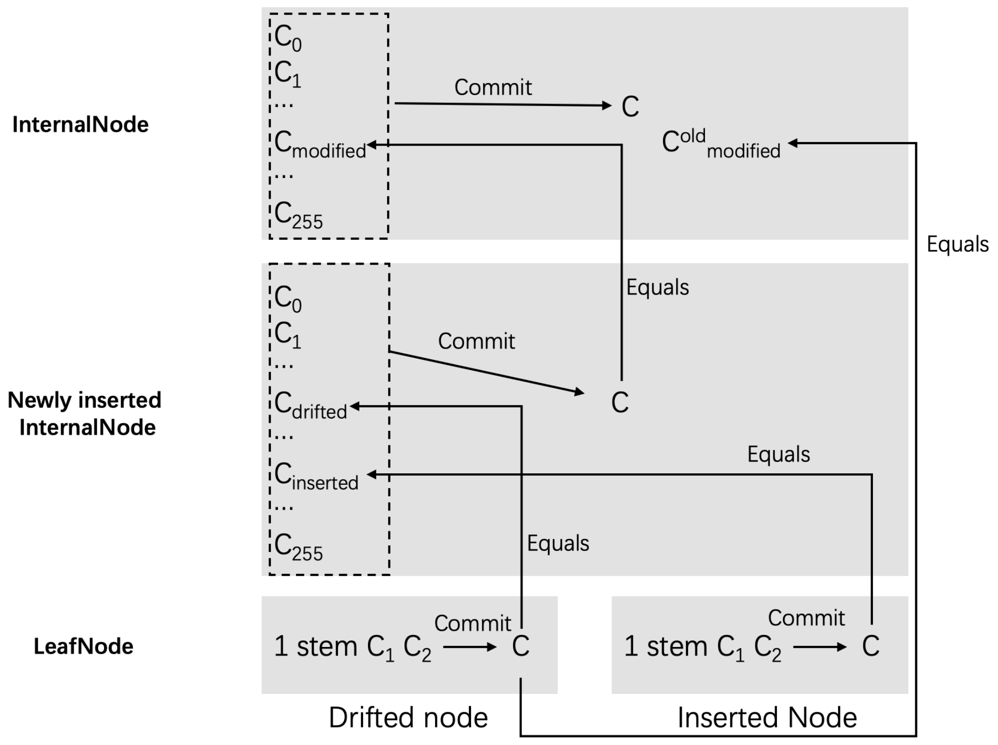


Figure 16: Verkle Tree update constraint schematic

7.4 Verkle Tree Trace Table and Constraints

7.4.1 Constraints for InternalNode

Figure 13 shows the structure of the trace section of InternalNode.

Each node is divided into 257 rows, of which the first 256 rows are `child_rows`, and the last row is `final_row`.

In `child_rows`:

- `data`: Polynomial commitment of the child node;
- `is_modified`: Whether the row has been updated;
- `pre_data`: Value before the row is updated, or 0 if it is not updated;
- `basis`: Lagrange basis;
- `acc_c`: Cumulative value of evaluating the committed polynomials, and the value of the column in the last row is the polynomial commitment of the node;
- `acc_diff`: Difference between the committed polynomial evaluation before update and the committed polynomial evaluation after update;
- `is_leaf`: Whether it's a LeafNode;
- `is_first_child_row`: Whether it's the first `child_row`.

In `final_row`:

data	data_aux	pre_data	pre_data_aux	is_modified	basis	acc_c	acc_c_aux	acc_diff
C_0	0	0	0	0	δ_0	$C_0\delta_0$	0	0
C_1	0	0	0	0	δ_1	$C_0\delta_0 + C_1\delta_1$	0	0
...	...	...	...	...	...	...	...	...
C_k	0	$C_{\text{old\_k}}$	0	1	δ_k	$\sum_{i=0}^{k} C_i\delta_i$	0	$-C_k \cdot \delta_k + C_k^{\text{old}}\delta_k$
...	...	...	...	...	...	...	...	...
C_{255}	0	0	0	0	δ_{255}	$\sum_{i=0}^{255} C_i\delta_i$	0	$-C_k \cdot \delta_{255} + C_k^{\text{old}}\delta_k$
C_{curr}	$C_{\text{old\_curr}}$	0	0	0	0	0	0	0

acc_diff_aux	is_leaf	is_drifted	is_child_row	is_first_child_row	modified_rlc	modified_rlc_mult
0	1	0	1	1	0	R
0	0	0	1	0	0	R
...	...	...	...	...	...	...
0	0	0	1	0	$C_k R + C_{\text{old\_k}} R^2$	R
...	...	...	...	...	...	...
0	0	0	1	0	$C_k R + C_{\text{old\_k}} R^2$	R^3
0	1	0	0	0	$C_k R + C_{\text{old\_k}} R^2$	R^3

Table 13: Trace table for InternalNode

- `data`: C_{curr}, polynomial commitment of the node;
- `data_aux`: $C_{\text{old\_curr}}$, polynomial commitment of the node before update.

Accumulate evaluations of constraint polynomials:

$$\text{is\_child\_row}_i \cdot (1 - \text{is\_modified}_i) \cdot \text{pre\_data}_i = 0$$

$$\text{is\_child\_row}_i \cdot (1 - \text{is\_first\_row}_i) \cdot (\text{acc\_c}_i - \text{acc\_c}_{i-1} - \text{basis}_i \cdot \text{data}_i) = 0$$

$$\text{is\_child\_row}_i \cdot \text{is\_first\_row}_i \cdot (\text{acc\_c}_i - \text{basis}_i \cdot \text{data}_i) = 0$$

$$\text{is\_child\_row}_i \cdot (1 - \text{is\_first\_row}_i) \cdot (\text{acc\_diff}_i - \text{acc\_diff}_{i-1} + \text{data} \cdot \text{basis}_i - \text{pre\_data} \cdot \text{basis}_i) = 0$$

$$\text{is\_child\_row}_i \cdot \text{is\_first\_row}_i \cdot (\text{acc\_diff}_i + \text{data} \cdot \text{basis}_i - \text{pre\_data} \cdot \text{basis}_i) = 0$$

$$(1 - \text{is\_leaf}_i) \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{data}_i - \text{acc\_c}_{i-1}) = 0$$

$$(1 - \text{is\_leaf}_i) \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{data\_aux}_i - \text{acc\_c}_{i-1} - \text{acc\_diff}_{i-1}) = 0$$

7.4.2 Constraints for LeafNode

The trace table structure of LeafNode is the same as that of InternalNode, just as shown in Figure 14, except that the definitions of some columns are different.

In `child_rows`:

- `data`: Value of the lower 32 bytes of the node, i.e. V^l;

data	data_aux	pre_data	pre_data_aux	is_modified	basis	acc_c	acc_c_aux	acc_diff
V_0^l	V_0^h	0	0	0	δ_0	$V_0^l\delta_0$	$V_0^h\delta_0$	0
V_1^l	V_1^h	0	0	0	δ_1	$V_0^l\delta_0 + V_1^l\delta_1$	$V_0^h\delta_0 + V_1^h\delta_1$	0
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
V_k^l	V_k^h	$V_{\text{old\_k}}^l$	$V_{\text{old\_k}}^h$	1	δ_k	$\sum_0^k V_i^l\delta_i$	$\sum_0^k V_i^h\delta_i$	$-V_k^l\delta_k + V_{\text{old\_k}}^l\delta_k$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
V_{255}^l	V_{255}^h	0	0	0	δ_{255}	$\sum_0^{255} V_i^l\delta_i$	$\sum_0^{255} V_i^h\delta_i$	$-V_k^l\delta_k + V_{\text{old\_k}}^l\delta_k$
1	stem	C_1	C_2	$C_{\text{old\_1}}$	$C_{\text{old\_2}}$	C_{curr}	$C_{\text{old\_curr}}$	0

acc_diff_aux	is_leaf	is_drifted	is_child_row	is_first_child_row	modified_rlc	modified_rlc_mult
0	1	0	1	1	0	R
0	1	0	1	0	0	R
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$-V_k^h\delta_k + V_{\text{old\_k}}^h\delta_k$	1	0	1	0	0	R
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$-V_k^h\delta_k + V_{\text{old\_k}}^h\delta_k$	1	0	1	0	0	R
0	1	0	0	0	0	R

Table 14: Trace table for LeafNode

- `data_aux`: Value of the upper 32 bytes of the node, i.e. V^h;
- `pre_data`: Value of the lower 32 bytes of the row before update, i.e. V_{old}^l, and 0 if it is not updated;
- `pre_data_aux`: Value of the upper 32 bytes of the row before updated, i.e. V_{old}^h, and 0 if it is not updated;
- `acc_c`: Cumulative value of evaluating the committed polynomials $\{V^l\}$, and the value of the column in the last row is the commitment of $\{V^l\}$;
- `acc_aux`: Cumulative value of evaluating the committed polynomials $\{V^h\}$, and the value of the column in the last row is the commitment of $\{V^h\}$;
- `acc_diff`: Difference between the committed polynomial $\{V^l\}$ evaluation before the update and the committed polynomial $\{V^l\}$ evaluation after the update;
- `acc_diff_aux`: Difference between the committed polynomial $\{V^h\}$ evaluation before the update and the committed polynomial $\{V^h\}$ evaluation after the update.

In `final_row`:

- `data`: 1;
- `data_aux`: Stem;
- `pre_data`: C_1, the commitment of $\{V^l\}$;
- `is_modified`: Whether the row has been updated;
- `basis`: Lagrange basis;

- `acc_c`: C_{curr}, polynomial commitment of this node;
- `acc_aux`: $C_{\text{old\_curr}}$, polynomial commitment of this node.

In addition to the constraints mentioned in the discussion of InternalNode in the previous subsection, we still need to add some constraints:

$$\text{is\_leaf}_i \cdot \text{is\_child\_row}_i \cdot (1 - \text{is\_first\_row}_i) \cdot (\text{acc\_c\_aux}_i - \text{acc\_c\_aux}_{i-1} - \text{basis}_i \cdot \text{data\_aux}_i) = 0$$

$$\text{is\_leaf}_i \cdot \text{is\_child\_row}_i \cdot \text{is\_first\_row}_i \cdot (\text{acc\_c\_aux}_i - \text{basis}_i \cdot \text{data\_aux}_i) = 0$$

$$\text{is\_leaf}_i \cdot \text{is\_child\_row}_i \cdot (1 - \text{is\_first\_row}_i) \cdot (\text{acc\_diff\_aux}_i - \text{acc\_diff\_aux}_{i-1} + \text{data} \cdot \text{basis}_i - \text{pre\_data\_aux} \cdot \text{basis}_i) = 0$$

$$\text{is\_leaf}_i \cdot \text{is\_child\_row}_i \cdot \text{is\_first\_row}_i \cdot (\text{acc\_diff\_aux}_i + \text{data} \cdot \text{basis}_i - \text{pre\_data\_aux} \cdot \text{basis}_i) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{pre\_data}_i - \text{acc\_c}_{i-1}) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{pre\_data\_aux}_i - \text{acc\_c\_aux}_{i-1}) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{is\_modified}_i - \text{acc\_c}_{i-1} - \text{acc\_diff}_{i-1}) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{basis}_i - \text{acc\_c\_aux}_{i-1} - \text{acc\_diff\_aux}_{i-1}) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{acc\_c}_i - \text{data}_i \cdot \text{basis}_{i-256} - \text{data\_aux} \cdot \text{basis}_{i-255} - \text{pre\_data} \cdot \text{basis}_{i-254} - \text{pre\_data\_aux} \cdot \text{basis}_{i-253}) = 0$$

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{acc\_c\_aux}_i - \text{data}_i \cdot \text{basis}_{i-256} - \text{data\_aux} \cdot \text{basis}_{i-255} - \text{is\_modified} \cdot \text{basis}_{i-254} - \text{basis} \cdot \text{basis}_{i-253}) = 0$$

7.4.3 Constraints between Child Node and Parent Node

In InternalNode:

- `modified_rlc`: rlc of the modified row;
- `modified_rlc_mult`: random number to be multiplied currently in the modified row.

Guarantee the calculation of parent node rlc:

$$\text{is\_first\_child\_row}_i \cdot (\text{modified\_rlc\_mult}_i - R) = 0$$

$$(1 - \text{is\_first\_child\_row}_i) \cdot (\text{modified\_rlc\_mult}_i - \text{modified\_rlc\_mult}_{i-1} \cdot \text{is\_modified}_{i-1} \cdot R^2) = 0$$

$$(1 - \text{is\_leaf}_i) \cdot (\text{modified\_rlc}_i - \text{data}_i \cdot \text{modified\_rlc\_mult}_i - \text{pre\_data} \cdot \text{modified\_rlc\_mult}_i \cdot R) = 0$$

If child node is InternalNode, we need to ensure

$$(1 - \text{is\_leaf}_i) \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{data}_i \cdot R + \text{data\_aux}_i \cdot R^2 - \text{modified\_rlc}_{i-257}) = 0$$

If child node is LeafNode, we need to ensure

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{acc\_c}_i \cdot R + \text{acc\_c\_aux}_i \cdot R^2 - \text{modified\_rlc}_{i-257}) = 0$$

7.5 Verkle Tree Key Constraints

The key of Verkle Tree is reflected in the trace of the root node. The trace table of the root node has 33 rows, as shown in Table 15. The first 32 rows are `child_rows` and the 33rd row is `final_row`. The root value and the root value before modification are stored in the 33rd row, i.e. `final_row`. Meanwhile, `final_row` also stores stem, path, and rlc of stem.

index	data	data_aux	pre_data	pre_data_aux	is_root
0	n_0	1	$n_0 \cdot 256^{30}$	256^{30}	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	n_k	1	$\sum_{i=0}^{k} n_i \cdot 256^{30-i}$	256^{30-k}	1
$k+1$	n_{k+1}	0	$\sum_{i=0}^{k+1} n_i \cdot 256^{30-i}$	256^{30-k-1}	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
30	n_{30}	0	$\sum_{i=0}^{30} n_i \cdot 256^{30-i}$	256^0	1
31	n_{31}	0	$\sum_{i=0}^{30} n_i \cdot 256^{30-i}$	0	1
32	root	root_{old}	0	stem	1

index	is_child_row	is_first_child_row	key_rlc_mult	key_path_rlc	key_path_acc
0	1	1	R	$n_0 \cdot R$	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	1	1	R^{k+1}	$\sum_{i=0}^{k} \text{data}_i \cdot \text{key\_rlc\_mult}_i$	0
$k+1$	1	1	R^{k+1}	$\sum_{i=0}^{k} \text{data}_i \cdot \text{key\_rlc\_mult}_i$	0
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
31	1	1	R^{k+1}	$\sum_{i=0}^{k} \text{data}_i \cdot \text{key\_rlc\_mult}_i$	0
32	0	0	1	$n_{31} \cdot R^{k+2} + \text{stem} \cdot R^{k+3} + \sum_{i=0}^{31} \text{data}_i \cdot \text{key\_rlc\_mult}_i$	0

Table 15: Related trace table of root node in Verkle Tree

In the trace table of root node:

- `data`: Divides the key into 32 nibbles, and each nibble is 1 byte. This column is the encoded value of each nibble (0x00 – 0xFF);
- `data_aux`: Whether the nibble corresponds to an InternalNode;
- `pre_data`: Cumulative calculation of key. The last row of `child_rows` takes `final_row` as the key of updated node;
- `pre_data_aux`: Used to assist in calculating stem values;
- `is_root`: Whether it is the root node;
- `key_rlc_mult`: Computes the random number of the relative path rlc of the key. This column is used to assist in the calculation of `key_path_acc` in InternalNode and LeafNode;
- `key_path_rlc`: Relative path rlc of key. The last row of `child_rows` is the accumulated result of rlc corresponding to InternalNode, and the last byte of stem and key are added to rlc in `final_row`;

- `key_path_acc`: Used to save the intermediate process of calculating the relative path of key. In `final_row` of LeafNode, the value of this column should be consistent with `key_path_rlc`.

  Relative constraints in root node: public inputs in `final_row`:

$$\text{is\_root}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{data}_i - \text{root}) = 0$$
$$\text{is\_root}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{data\_aux}_i - \text{root}_{\text{old}}) = 0$$
$$\text{is\_root}_i \cdot (1 - \text{is\_child\_row}) \cdot (\text{pre\_data\_aux}_i - \text{stem}) = 0$$

Relative constraints of stem:

$$\text{is\_root}_i \cdot \text{is\_first\_child\_row}_i \cdot (\text{pre\_data}_i - \text{data}_i \cdot \text{pre\_data\_aux}_i) = 0$$
$$\text{is\_root} \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{pre\_data\_aux}_i - \text{pre\_data}_{i-1}) = 0$$

Constraints of relative path rlc of key:

$$\text{is\_root}_i \cdot \text{is\_first\_child\_row}_i \cdot (\text{key\_rlc\_mult}_i - R) = 0$$

$$\begin{aligned}\text{is\_root}_i \cdot (1 - \text{is\_first\_child\_row}_i) \cdot \text{is\_child\_row}_i &\\ \cdot (1 - \text{is\_first\_child\_row}) \cdot (\text{key\_rlc\_mult}_i - R \cdot \text{data\_aux}_{i-1}) &= 0\end{aligned}$$
$$\text{is\_root}_i \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{key\_rlc\_mult}_i - 1) = 0$$

$$\text{is\_root}_i \cdot \text{is\_first\_child\_row}_i \cdot (\text{key\_path\_rlc}_i - \text{data}_i \cdot \text{key\_rlc\_mult}_i) = 0$$

$$\begin{aligned}\text{is\_root}_i \cdot (1 - \text{is\_first\_child\_row}_i) \cdot \text{is\_child\_row}_i &\\ \cdot (\text{key\_path\_rlc}_i - \text{key\_path\_rlc}_{i-1} - \text{data\_aux} \cdot \text{data}_i \cdot \text{key\_rlc\_mult}_i) &= 0\end{aligned}$$

$$\begin{aligned}\text{is\_root}_i \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{key\_path\_rlc}_i - \text{data}_{i-1} \cdot R \cdot \text{key\_rlc\_mult}_{i-1} &\\ - \text{pre\_data\_aux}_i \cdot R^2 \cdot \text{key\_rlc\_mult}_{i-1} - \text{key\_path\_rlc}_{i-1}) &= 0\end{aligned}$$

  In InternalNode and LeafNode:

  `key_rlc_mult` only changes on the first row of node. Multiply it based on its parent by R:

$$(1 - \text{is\_root}_i) \cdot (1 - \text{is\_leaf}_i) \cdot \text{is\_first\_child\_row}_i \cdot (\text{key\_rlc\_mult}_i - \text{key\_rlc\_mult}_{i-1} \cdot R) = 0$$
$$(1 - \text{is\_root}_i) \cdot (1 - \text{is\_leaf}_i) \cdot (1 - \text{is\_first\_child\_row}_i) \cdot (\text{key\_rlc\_mult}_i - \text{key\_rlc\_mult}_{i-1}) = 0$$

`key_path_rlc` remains unchanged:

$$(1 - \text{is\_root}_i) \cdot (\text{key\_path\_rlc}_i - \text{key\_path\_rlc}_{i-1}) = 0$$

`key_path_acc` only changes in is_modified:

$$(1 - \text{is\_root}_i) \cdot (1 - \text{is\_modified}_i) \cdot (\text{key\_path\_acc}_i - \text{key\_path\_acc}_{i-1}) = 0$$
$$(1 - \text{is\_root}_i) \cdot \text{is\_modified}_i \cdot (\text{key\_path\_acc}_i - \text{key\_path\_acc}_{i-1} - \text{index}_i \cdot \text{key\_rlc\_mult}_i) = 0$$

In `final_row` of child node, `key_path_acc` column is equal to `key_path_acc` column:

$$\text{is\_leaf}_i \cdot (1 - \text{is\_child\_row}_i) \cdot (\text{key\_path\_acc}_i - \text{key\_path\_acc}_i) = 0$$

8 Integer Arithmetic Operations

In order to support arithmetic operations of uint256 in OlaVM, we use two field elements to represent the lower and the upper 128 bits of a uint256 number. When describing the VM operating logic and constraints of an instruction, we use the following notations:

- `r_a`: Register `a` which stores uint256 input a;
- `r_a_lo`: Register `a_lo` which stores the lower 128 bits of uint256 input a;
- `r_a_hi`: Register `a_hi` which stores the upper 128 bits of uint256 input a;
- `r_b`: Register `b` which stores another uint256 input b;
- `r_b_lo`: Register `b_lo` which stores the lower 128 bits of uint256 input b;
- `r_b_hi`: Register `b_hi` which stores the upper 128 bits of uint256 input b;
- `r_dst`: Register `dst` which stores the uint256 output;
- `r_dst_lo`: Register `dst_lo` which stores the lower 128 bits of uint256 operation output;
- `r_dst_hi`: Register `dst_hi` which stores the upper 128 bits of uint256 operation output;
- `[r_x]`: The value of register `x`;
- `[carry_lo]`: The carry flag for lower 128-bit arithmetic operation;
- `[carry_hi]`: The carry flag for upper 128-bit arithmetic operation;
- `%{...%}`: Hints, a concept borrrowed from Cairo [7], which defines a block of some other programming language which will be executed by the prover right before the next instruction. This block is not implemented by OlaVM instructions.

8.1 uint256 Arithmetic Operations

8.1.1 ADD

Library function interface: `uint256_add(uint256 a, uint256 b)`

1. Split two uint256 inputs a and b into two parts of the upper 128 bits and the lower 128 bits, and perform Range Check.
2. Perform field addition between the lower 128 bits of two numbers where reuse the constraints of field addition in Section 5.1, then we get the lower 128 bits of the uint256 result and the carry flag `carry_lo`.
3. Perform field addition between the upper 128 bits and add `carry_lo`, then we get the upper 128 bits of uint256 and carry flag `carry_hi`.

The instructions corresponding to the addition of uint256 are

```
split128 r_a, r_a_lo, r_a_hi
split128 r_b, r_b_lo, r_b_hi
[r_src1] = [r_a_lo]
[r_src2] = [r_b_lo]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
ADD r_dst_lo, r_src1, r_src2

if r_dst_lo >= 2^128:
    carry_lo = 1
    %{
        r_dst_lo = r_dst_lo - 2^128
    %}
else:
    carry_lo = 0

[r_src1] = [r_a_hi]
[r_src2] = [r_b_hi]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
ADD r_dst_hi, r_src1, r_src2
[r_carry] = carry_lo
ADD r_dst_hi, r_dst_hi, r_carry

if r_dst_hi >= 2^128:
    carry_hi = 1
    %{
        r_dst_hi = r_dst_hi - 2^128
    %}
else:
    carry_hi = 0

[r_carry] = carry_hi
%{
    [r_dst] = [r_dst_hi] * 2^128 + [r_dst_lo]
%}
return [r_dst], [r_carry]
```

The corresponding constraints are

$$
\begin{aligned}
&\texttt{r\_lo} \in [0, 2^{128})\\
&\texttt{r\_hi} \in [0, 2^{128})\\
&\texttt{carry\_lo} \cdot (\texttt{carry\_lo} - 1) = 0\\
&\texttt{carry\_hi} \cdot (\texttt{carry\_hi} - 1) = 0\\
&\texttt{dst\_lo} + \texttt{carry\_lo} \cdot 2^{128} - (\texttt{a\_lo} + \texttt{b\_lo}) = 0\\
&\texttt{dst\_hi} + \texttt{carry\_hi} \cdot 2^{128} - (\texttt{a\_hi} + \texttt{b\_hi}) = 0
\end{aligned}
$$

8.1.2 NOT

Library function interface: `uint256_not(uint256 a)`

The instructions corresponding to bitwise NOT of uint256 are

```
split128 r_a, r_a_lo, r_a_hi
[r_src1] = [r_a_lo]
[r_src2] = [r_a_hi]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
[r_u128] = 2^128 - 1
[r_dst_lo] = [r_u128] - [r_src1]
```

```
[r_dst_hi] = [r_u128] - [r_src2]
%{
    [r_dst] = [r_dst_hi] * 2^128 + [r_dst_lo]
%}
return [r_dst]
```

The corresponding constraints are

$$\begin{aligned} \texttt{r\_lo} &\in [0, 2^{128}) \\ \texttt{r\_hi} &\in [0, 2^{128}) \\ \texttt{[r\_dst\_lo]} + \texttt{[r\_a\_lo]} - 2^{128} + 1 &= 0 \\ \texttt{[r\_dst\_hi]} + \texttt{[r\_a\_hi]} - 2^{128} + 1 &= 0 \end{aligned}$$

8.1.3 NEG

Library function interface: `uint256_neg(uint256 a)`

The negation of uint256 relies on two uint256 operations, ADD and NOT, so the constraints of these two operations will also be used, but will not be enumerated again under this operation. The corresponding instructions are

```
[r_not_num] = uint256_not(a)
[r_dst] = uint256_add(r_not_num, 1)
return [r_dst]
```

The corresponding constraint is

$$\texttt{[r\_dst]} + \texttt{[r\_a]} - 2^{256} = 0$$

8.1.4 SUB

Library function interface: `uint256_sub(uint256 a, uint256 b)`

The instructions corresponding to the subtraction of uint256 are

```
split128 r_a, r_a_lo, r_a_hi
split128 r_b, r_b_lo, r_b_hi
[r_src1] = [r_a_lo]
[r_src2] = [r_b_lo]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)

if r_src1 < r_src2:
    borrow_lo = 0
else:
    borrow_lo = 1
    [r_u128] = 2^128
    add r_src1, r_src1, r_u128
SUB r_dst_lo, r_src1, r_src2

[r_src1] = [r_a_hi]
[r_src2] = [r_b_hi]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
[r_borrow] = borrow_lo
ADD r_src2, r_src2, r_borrow

if r_src1 < r_src2:
    borrow_hi = 0
else:
    borrow_hi = 1
    [r_u128] = 2^128
    ADD r_src1, r_src1, r_u128
SUB r_dst_hi, r_src1, r_src2

[r_borrow] = borrow_hi
%{
    [r_dst] = [r_dst_hi] * 2^128 + [r_dst_lo]
%}
return [r_dst], [r_borrow]
```

The corresponding constraints are

$$\begin{aligned}
&\texttt{r\_lo} \in [0, 2^{128}) \\
&\texttt{r\_hi} \in [0, 2^{128}) \\
&\texttt{carry\_lo} \cdot (\texttt{carry\_lo} - 1) = 0 \\
&\texttt{carry\_hi} \cdot (\texttt{carry\_hi} - 1) = 0 \\
&\texttt{dst\_lo} + \texttt{borrow\_lo} \cdot 2^{128} - (\texttt{a\_lo} + \texttt{b\_lo}) = 0 \\
&\texttt{dst\_hi} + \texttt{borrow\_hi} \cdot 2^{128} - (\texttt{a\_hi} + \texttt{b\_hi}) = 0
\end{aligned}$$

8.1.5 MUL

Library function interface: `uint256_mul(uint256 a, uint256 b)`

The range of the multiplication result of two uint256 numbers is $[0, 2^{512})$, so the final result needs to be stored in two registers and the numbers are splitted into 128 bits for multiplication and addition in calculation process. The principle is as Figure 17.

The instructions corresponding to the multiplication of uint256 are

```
split128 r_a, r_a_lo, r_a_hi
split128 r_b, r_b_lo, r_b_hi
[r_src1] = [r_a_lo]
[r_src2] = [r_b_lo]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
MUL r_dst_al_bl, r_src1, r_src2

[r_src1] = [r_a_hi]
[r_src2] = [r_b_lo]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
MUL r_dst_ah_bl, r_src1, r_src2
```

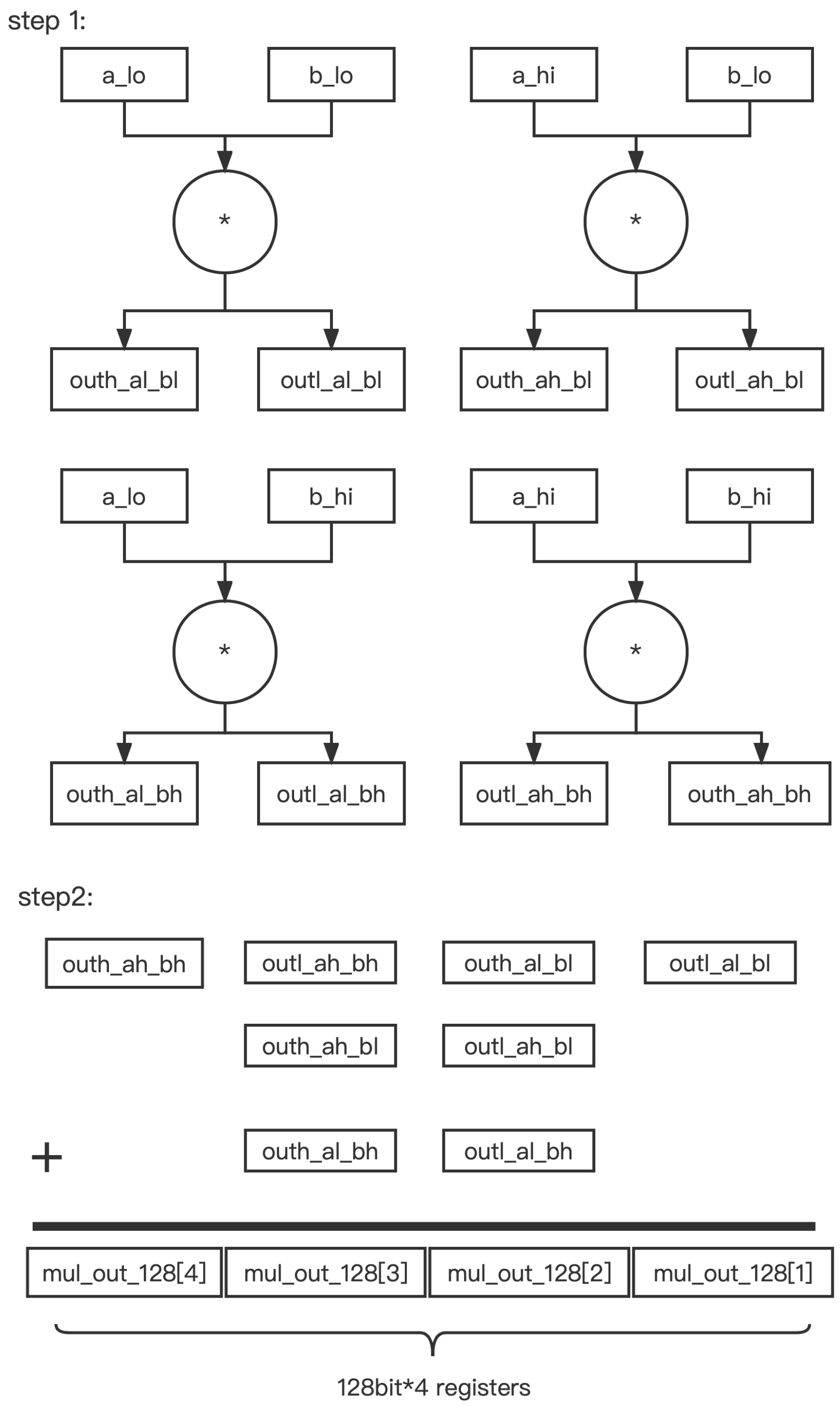


Figure 17: uint256 multiplication algorithm

```
[r_src1] = [r_a_lo]
[r_src2] = [r_b_hi]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
MUL r_dst_al_bh, r_src1, r_src2

[r_src1] = [r_a_hi]
[r_src2] = [r_b_hi]
range_check(r_src1, MAX_uint128)
range_check(r_src2, MAX_uint128)
MUL r_dst_ah_bh, r_src1, r_src2

split128 r_dst_al_bl, r_dst_al_bl_lo, r_dst_al_bl_hi
split128 r_dst_ah_bl, r_dst_ah_bl_lo, r_dst_ah_bl_hi
split128 r_dst_al_bh, r_dst_al_bh_lo, r_dst_al_bh_hi
split128 r_dst_ah_bh, r_dst_ah_bh_lo, r_dst_ah_bh_hi

range_check(r_dst_al_bl_hi, MAX_uint128)
range_check(r_dst_ah_bl_lo, MAX_uint128)
[r_mul_out_128_1] = [r_dst_al_bl]

ADD r_mul_out_128_2, r_dst_al_bl_hi, r_dst_al_bl_hi
ADD r_mul_out_128_2, r_mul_out_128_2, r_dst_al_bh_lo

split128 r_mul_out_128_2, r_mul_out_128_2, r_mul_out_128_2_carry

ADD r_mul_out_128_3, r_mul_out_128_2_carry, r_dst_ah_bl_hi
ADD r_mul_out_128_3, r_mul_out_128_3, r_dst_al_bh_hi
ADD r_mul_out_128_3, r_mul_out_128_3, r_dst_ah_bh_lo

split128 r_mul_out_128_3, r_mul_out_128_3, r_mul_out_128_3_carry

ADD r_mul_out_128_4, r_mul_out_128_3_carry, r_dst_ah_bh_hi

%{
    r_mul_out_256_2 = r_mul_out_128_4 * 2^128 + r_mul_out_128_3
    r_mul_out_256_1 = r_mul_out_128_2 * 2^128 + r_mul_out_128_1
%}

return [r_mul_out_256_2], [r_mul_out_256_1]
```

The multiplication of uint256 is composed of several instructions. The instructions are constrained by corresponding constraints in Section 5.1. The independent constraint of multiplication is

$$\texttt{[r\_mul\_out\_256\_2]} \cdot 2^{256} + \texttt{[r\_mul\_out\_256\_1]} - \texttt{a} \cdot \texttt{b} = 0$$

8.1.6 DIV

Library function interface: `uint256_div(uint256 a, uint256 div)`

The instructions corresponding to the division of uint256 are

```
if div == 0:
    return 0, 0
%{
    quotient = a / div
    remainder = a % div
%}
[r_quotient] = quotient
(r_mul_out_256_2, r_mul_out_256_1) = uint256_mul(r_quotient, div)
[r_zero] = 0
EQ r_mul_out_256_2, r_zero

[r_rem] = remainder

(r_a_check, r_add_carry) = uint256_add(r_mul_out_256_1, r_rem)
EQ r_a_check, r_a
EQ r_add_carry, r_zero
return [r_quotient], [r_rem]
```

The division of uint256 is composed of several instructions. The instructions are constrained by corresponding constraints in Section 5.1. The independent constraint of division is

$$\texttt{[r\_quotient]} \cdot \texttt{[r\_div]} + \texttt{[r\_rem]} - \texttt{[r\_a]} = 0$$

9 ZK Without FFT

9.1 Lagrange Basis Polynomial

According to Lagrange Interpolation Algorithm, given n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$, there exists a unique polynomial with a degree less than n passing through these n interpolation points, which can be expressed as

$$f(x) = \sum_{i=0}^{n-1} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Define Lagrange basis polynomial as

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Then the above formula can be written as $f(x) = \sum_{i=0}^{n-1} y_i L_i(x)$. Let $v(x) = \prod_{i=0}^{n-1}(x - x_i)$, using the formal derivative of $v(x)$

$$v'(x) = \sum_{i=0}^{n-1} \prod_{j \neq i} (x_i - x_j)$$

$$v'(x_i) = \prod_{j \neq i} (x_i - x_j)$$

we can simplify the Lagrange basis to

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} = \frac{v(x)}{v'(x)(x - x_i)}$$

All polynomials with a degree less than n, form an n-dimensional vector space under polynomial addition. For a polynomial $f(x)$ of a degree less than n,

$$f(x) = \sum_{i=0}^{n-1} a_i x^i$$

$1, x, \dots, x^{n-1}$ is a basis of the vector space, and the coordinate a_i is called the coefficient form of $f(x)$. On the other hand,

$$f(x) = \sum_{i=0}^{n-1} f(x_i) L_i(x)$$

Lagrange basis $L_i(x)$ is also a basis of the vector space, and the coordinate $f_i = f(x_i)$ is called the evaluation form of $f(x)$. The FFT/IFFT algorithm enables the transformation of the coordinates between the two bases.

Next, we will study how to calculate exact division when the divisor is a polynomial of degree 1, i.e. given $f(a) = 0$, calculate the value of $q(x) = f(x)/(x-a)$ at the interpolation point x_i. When $x_m \neq a$, $q(x_m) = f_m/(x_m - a)$, when $x_m = a$, since the degree of $q(x) = f(x)/(x - x_m)$ is less than $n-1$, we only need $n-1$ values of interpolation points $x_0, \dots, x_{m-1}, x_{m+1}, \dots, x_{n-1}$ to uniquely determine $q(x)$.

$$\begin{aligned}
q(x) &= \frac{f(x)}{x - x_m} \\
&= \sum_{i \neq m} \frac{f_i}{x_i - x_m} \prod_{j \neq i,m} \frac{x - x_j}{x_i - x_j} \\
&= \sum_{i \neq m} \frac{f_i}{x - x_m} \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \\
&= \sum_{i \neq m} \frac{f_i}{x - x_m} L_i(x) \\
&= \sum_{i \neq m} \frac{f_i}{x - x_m} \frac{v(x)}{v'(x)(x - x_i)} \\
&= \sum_{i \neq m} \frac{f_i}{v'(x_i)(x - x_i)} \cdot \prod_{i \neq m} (x - x_i)
\end{aligned}$$

Then

$$\begin{aligned}
q_m = q(x_m) &= \sum_{i \neq m} \frac{f_i}{v'(x_i)(x_m - x_i)} \cdot \prod_{i \neq m} (x_m - x_i) \\
&= \sum_{i \neq m} \frac{f_i}{x_m - x_i} \cdot \frac{v'(x_m)}{v'(x_i)}
\end{aligned}$$

For non-interpolation points $z \notin \{x_0, \dots, x_{n-1}\}$, we can also quickly calculate its value

$$\begin{aligned}
f(z) = \sum_{i=0}^{n-1} f_i L_i(z) &= \sum_{i=0}^{n-1} \frac{v(z) f_i}{v'(z)(z - x_i)} \\
&= v(z) \sum_{i=0}^{n-1} \frac{f_i}{v'(x_i)(z - x_i)}
\end{aligned}$$

If the interpolation point takes the n-th root of unity: $x_i = \omega^i$, then we have

$$v(x) = x^n - 1 \qquad v'(x) = n x^{n-1} \qquad v'(x_i) = n\omega^{-i}$$

$$q_m = \sum_{i \neq m} \frac{f_i}{\omega^m - \omega^i} \cdot \frac{n\omega^{-m}}{n\omega^{-i}} = \sum_{i \neq m} \frac{\omega^{i-m} f_i}{\omega^m - \omega^i}$$

$$f(z) = (z^n - 1)\sum_{i=0}^{n-1} \frac{f_i}{n\omega^{-i}(z-\omega^i)} = \frac{z^n-1}{n}\sum_{i=0}^{n-1}\frac{f_i\omega^i}{z-\omega^i}$$

9.2 KZG Without FFT

We can represent a polynomial using Lagrange basis in KZG commitment, thus avoiding the use of FFT. The specific changes can be found in Table 16.

9.3 FRI Decomposition

We can write a polynomial $f(x)$ as the sum of odd and even power terms: $f(x) = f_e(x^2) + xf_o(x^2)$. Let $\tilde{f}(x) = f(-x)$, then

$$f_e(x^2) = \frac{f(x)+f(-x)}{2} = \frac{f(x)+\tilde{f}(x)}{2}, \quad f_o(x^2) = \frac{f(x)-f(-x)}{2x} = \frac{f(x)-\tilde{f}(x)}{2x}$$

Lemma 9.1. *Let H be an even-order subgroup of $\mathbb{F}_p^*$, a, b, c, d be polynomials on the Lagrange basis of H, and r be a random number. Let $H' = \{h^2 : h \in H\}$,*

$$\begin{aligned} a' &= a_e + ra_o \\ b' &= b_e + rb_o \\ c' &= c_e + rc_o \\ d' &= r^2 d_e/x + rd_o \end{aligned}$$

which are polynomials on the Lagrange basis of H', then

$$\begin{cases} ab = c + d \\ a\tilde{b} = c - d \end{cases} \iff \begin{cases} a'b' = c' + d' \\ a'\tilde{b}' = c' - d' \end{cases}$$

The proof of Lemma 9.1 can refer to [10].

We can build a recursive proof protocol without FFT using Lemma 9.1. Set ω to be a primitive root of unity of order 2^m, $H_0 = \langle\omega\rangle$ be the subgroup generated by ω. In order to prove the polynomial equation $ab = h$, we construct the polynomial on the Lagrange basis of H_0: $a^{(0)} = a$, $b^{(0)} = b$. The prover first calculates the polynomials $c^{(0)}$, $d^{(0)}$, which produces

$$\begin{cases} a^{(0)}b^{(0)} = c^{(0)} + d^{(0)} \\ a^{(0)}\tilde{b}^{(0)} = c^{(0)} - d^{(0)} \end{cases}$$

and sends the KZG commitment of a_0, b_0, c_0, d_0 to the verifier.

The recursive proof requires m rounds in total. In the k-th round, the prover starts from the commitments of $a_e^{(k)}, a_o^{(k)}, b_e^{(k)}, b_o^{(k)}, c_e^{(k)}, c_o^{(k)}, d_e^{(k)}/x, d_o^{(k)}$ that correspond to four polynomials $a^{(k)}, b^{(k)}, c^{(k)}, d^{(k)}$ on the Lagrange basis of $H_k = \langle\omega^{2^k}\rangle$ and sends them to the verifier. The verifier replies with a random number r and computes the corresponding commitment of a', b', c', d'. In the next round, replace H_k with H_{k+1} and replace a, b, c, d with $a^{(k+1)} = a', b^{(k+1)} = b', c^{(k+1)} = c', d^{(k+1)} = d'$ until $a^{(m)}, b^{(m)}, c^{(m)}, d^{(m)}$ are reduced to constants after m rounds and $|H_m| = 1$.

Finally, the verifier randomly picks a point z and performs multiple consistency verifications. For each polynomial

Coefficient Form	**Evalutaion Form**
Notations	
$\mathbb{F}$: finite field λ: security parameters $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$: groups of prime order p over ECC $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$: symmetric bilinear pairing g, h: generators of $\mathbb{G}_1, \mathbb{G}_2$ t: max degree ω: primitive n-th root of unity in $\mathbb{F}$	$\mathbb{F}$: finite field λ: security parameters $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$: groups of prime order p over ECC $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$: symmetric bilinear pairing g, h: generators of $\mathbb{G}_1, \mathbb{G}_2$ t: max degree ω: primitive n-th root of unity in $\mathbb{F}$ L_i: Lagrange basis, such that $L_j(\omega^i) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$
Input	
$\{(\omega^0, \phi(\omega^0)), (\omega^1, \phi(\omega^1)), \ldots, (\omega^{n-1}, \phi(\omega^{n-1}))\}$	$\{(\omega^0, \phi(\omega^0)), (\omega^1, \phi(\omega^1)), \ldots, (\omega^{n-1}, \phi(\omega^{n-1}))\}$
Setup	
SK: $\alpha \in \mathbb{F}_p$, generated by trusted authority PK: $(g, \alpha g, \ldots, \alpha^t g) \in \mathbb{G}_1^{t+1},\ (h, \alpha h) \in \mathbb{G}_2^2$	SK: $\alpha \in \mathbb{F}_p$, generated by trusted authority PK: $(g, \alpha g, \ldots, \alpha^t g) \in \mathbb{G}_1^{t+1},\ (h, \alpha h) \in \mathbb{G}_2^2$
Commitment(PK, $\phi(x)$)	
Polynomial: $\phi(x) = \sum_{j=0}^{\deg\phi} \phi_j x^j$ (IFFT) Commitment: $C = \sum_{j=0}^{\deg\phi} \phi_j(\alpha^j g)$	Polynomial: $\phi(x) = \sum_{j=0}^{n-1} \phi(\omega^j) L_j(x)$ Commitment: $C = \sum_{j=0}^{n-1} \phi(\omega^j) L_j(\alpha) g$
CreateWitness(PK, $\phi(z)$, z) (Query point z out of domain, used in IOPP)	
Compute quotient polynomial: $\psi(x) = \dfrac{\phi(x) - \phi(z)}{x - z}$ Commitment: $w_z = \sum_{j=0}^{\deg\psi} \psi_j(\alpha^j g)$	Evaluate out of domain: $\phi(z) = \dfrac{z^n - 1}{n} \sum_{i=0}^{n-1} \dfrac{\psi(\omega^i)\omega^i}{z - \omega^i}$ Compute quotient polynomial: $\psi(\omega^j) = \dfrac{\phi(\omega^j) - \phi(z)}{\omega^j - z}$ Commitment: $w_z = \sum_{j=0}^{n-1} \psi(\omega^j) L_j(\alpha) g$
CreateWitness(PK, $\phi(z)$, z) (Query point z in of domain, used in Verkle Tree)	
Evaluate in the domain: $\phi(x) = \sum_{j=0}^{\deg\phi} \phi_j z^j$ Compute quotient polynomial: $\psi(x) = \dfrac{\phi(x) - \phi(z)}{x - z}$ Commitment: $w_z = \sum_{j=0}^{\deg\psi} \psi_j(\alpha^j g)$	Compute quotient polynomial $\psi(\omega^j) = \begin{cases} \frac{\phi(\omega^j) - \phi(z)}{\omega^j - z} & \omega^j \neq z \\ \sum_{\omega^i \neq z} \frac{(\phi(\omega^i) - \phi(z))\omega^i}{z(z - \omega^i)} & \omega^j = z \end{cases}$ Commitment: $w_z = \sum_{j=0}^{n-1} \psi(\omega^j) L_j(\alpha) g$
VerifyEval(PK, C, w_i, z, $\phi(z)$)	
Verify: $e(w_z, (\alpha - z)h) = e(C - \phi(z)g, h)$	Verify: $e(w_z, (\alpha - z)h) = e(C - \phi(z)g, h)$

Table 16: KZG Commitment without FFT

$a^{(k)}, b^{(k)}, c^{(k)}$, the verifier needs to verify

$$\begin{cases} f(z) = f_e(z^2) + zf_o(z^2) \\ f(-z) = f_e(z^2) - zf_o(z^2) \end{cases}$$

For polynomial $d^{(k)}$, the verifier needs to verify

$$\begin{cases} f(z) = z^2(f_e/x)(z^2) + zf_o(z^2) \\ f(-z) = z^2(f_e x)(z^2) - zf_o(z^2) \end{cases}$$

f represents some a, b, c, d, and z is a random point.

The verifier also needs to verify the boundary conditions

$$\begin{cases} a^{(m)}(z)b^{(m)}(z) = c^{(m)}(z) + d^{(m)}(z) \\ a^{(m)}(z)\tilde{b}^{(m)}(z) = c^{(m)}(z) - d^{(m)}(z) \end{cases}$$

10 ZK Acceleration

The Zero Knowledge calculation process primarily consists of MSM, FFT/IFFT and polynomial evaluation. Given a circuit scale of approximately 2^{25}, the rough computational consumption ratios are as shown in Figure 18, ratios will vary for each unique scenario and algorithm.

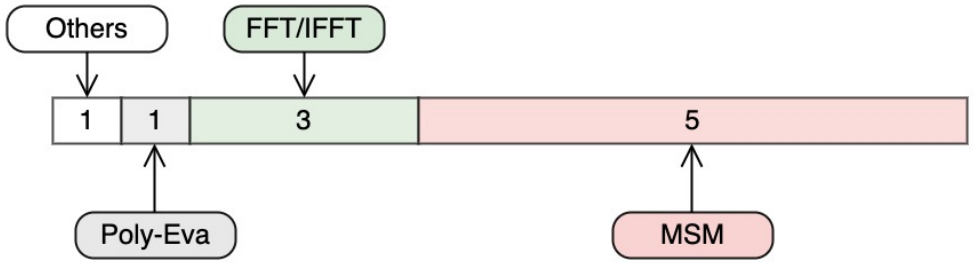


Figure 18: Consumption ratios of each module of ZK calculation when $n \approx 2^{25}$

Do note that with an increased circuit size, consumption ratio of FFT/IFFT increases as well (because of the better parallelism implementation of MSM than FFT/IFFT), therefore, simply accelerating MSM and FFT/IFFT modules by GPU/FPGA to improve the speed of the prover is limited. We need a ZK algorithm without FFT, where the proportion of FFT/IFFT is so small that it can be ignored, so that the final consumption ratio of each module is close to Figure 19.

We will introduce additional polynomials to eliminate FFT/IFFT calculations, thus generating additional MSM and polynomial evaluation operations, then using hardware-accelerated designs for MSM and polynomial evaluation operations.

10.1 MSM of Variable Basis

Given two vectors $(a_1, a_2, \ldots, a_n)$, $(G_1, G_2, \ldots, G_n)$, where a_i is a field element, and G_i is a point on the elliptic curve, MSM algorithm is to compute the following expressions: $\sum_{i=1}^{n} a_i G_i$.

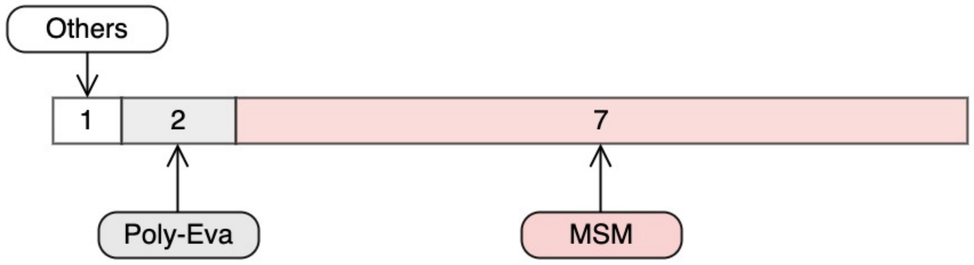


Figure 19: The consumption target ratio of each module of ZK calculation

10.1.1 Windowing Method

Assuming that the bit width of a_i is b, we split it into multiple sub-modules with a bit width of c. Such sub-modules have a total of $k = \lceil b/c \rceil$, then

$$a_iG_i = a_{i,0}G_i + 2^c a_{i,1}G_i + \cdots + 2^{(k-1)c}a_{i,k-1}G_i$$

For all i we have

$$\begin{aligned} a_1G_1 &= a_{1,0}G_1 + 2^c a_{1,1}G_1 + \cdots + 2^{(k-1)c}a_{1,k-1}G_1 \\ a_2G_2 &= a_{2,0}G_2 + 2^c a_{2,1}G_2 + \cdots + 2^{(k-1)c}a_{2,k-1}G_2 \\ &\cdots \\ a_nG_n &= a_{n,0}G_n + 2^c a_{n,1}G_n + \cdots + 2^{(k-1)c}a_{n,k-1}G_n \end{aligned}$$

So

$$\sum_{i=1}^{n} a_iG_i = \sum_{i=1}^{n}\sum_{j=0}^{k-1} a_{i,j}2^{jc}G_i = \sum_{j=0}^{m-1}\left(\sum_{i=1}^{n} a_{i,j}G_i\right)2^{jc}$$

For $a_{i,j} \in [0, 2^c)$, we can precompute

$$\begin{matrix} 1G_1 & 2G_1 & \cdots & 2^{c-1}G_1 \\ 1G_2 & 2G_2 & \cdots & 2^{c-1}G_2 \\ \vdots & \vdots & \ddots & \vdots \\ 1G_n & 2G_n & \cdots & 2^{c-1}G_n \end{matrix}$$

10.1.2 Endomorphism

For a cyclic group $\mathbb{G}$ over an elliptic curve $y^2 = x^3 + ax + b$ on a finite field $\mathbb{F}_p$, if one can find such a group endomorphism φ: there exists $\alpha, \beta \in \mathbb{F}_p$ such that $\varphi(x, y) = (\alpha x, \beta y)$ holds for all points on $\mathbb{G}$. It is easy to prove that such an automorphism is a multiplicative map, i.e. we can find a λ such that $\varphi(P) = \lambda P$ holds for all points P on $\mathbb{G}$. This means that when we know the coordinates of a point, we only need to multiply the x-coordinate and y-coordinate by a number in $\mathbb{F}_p$ to become the coordinates of another point, which can be used for further optimization of the algorithm. If the parameters of the elliptic curve are special, for example, BLS curves can be written as $y^2 = x^3 + b$, and $p \equiv 1 \pmod 3$, taking an element α of order 3 in $\mathbb{F}_p^*$, there exits a corresponding λ such that $\lambda(x, y) = (\alpha x, y)$, then the multiplication operation can be optimized as

$$\begin{aligned} mP &= (m_1 + m_2\lambda)P \\ &= m_1P + m_2(\lambda P) \\ &= m_1P + m_2\varphi(P) \end{aligned}$$

From [27] we can find small m_1 and m_2 to make the above equation hold, i.e. $|m_1|, |m_2| < \sqrt{3 \cdot |\mathbb{G}|}$, which can reduce a multiplication operation of b bits into two multiplication operations of $b/2$ bits. Applied to the windowing method, the number of group operations is reduced to

$$\frac{b}{c}(n + 2^c - 2) + b - c + \frac{b}{c} - 1 \approx \frac{b}{c}(n + 2^c)$$

When $n = 10^5$, it can save about 5.5% of group operations.

10.2 FPGA Acceleration

The main flow of MSM calculation in FPGA is shown in Figure 20. When performing MSM of a large number of curve points, we mainly use Pippenger's algorithm [18] to reduce the calculation of point doubling in MSM calculation process to improve calculation efficiency. Then perform a point addition on the buckets we get from the calculation.

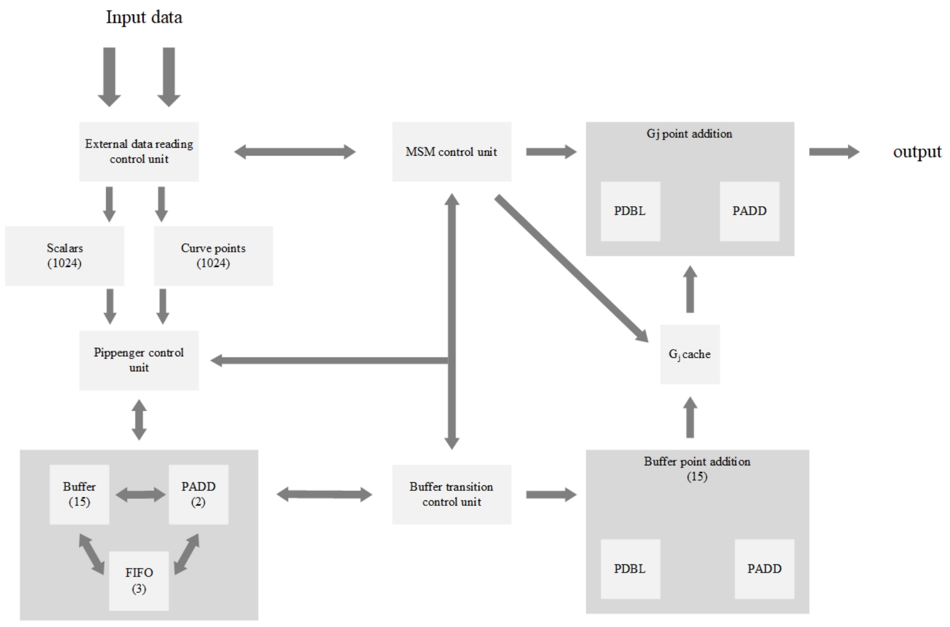


Figure 20: FPGA acceleration

During the whole calculation process, we need a main control logic: MSM control logic, which completes the triggering of the external data reading logic, monitoring the status of the Pippenger control logic, Buffer transition and reading, point addition and point doubling control of the output results of Pippenger's algorithm. This is the brain of MSM.

The core process of the whole calculation is Pippenger's algorithm. In Pippenger's algorithm, firstly, read 1024 scalar values and 1024 curve points from the external memory through the external data reading control logic. Divide the scalar values into 4 bits, and then scan the curve points to complete PADD operation operation. It involves four modules in Pippenger's algorithm: the Pippenger control logic, Buffer, FIFO group and PADD operation. The Pippenger control logic mainly completes the division of sliding windows for scalar values, the cyclic extraction of curve point coordinates and the allocation, control of FIFO reading and writing, and the input and output of PADD. The Buffer buffers the output of PADD operation, while the FIFO group buffers the input of PADD operation.

Due to the limited on-chip memory of FPGA, Pippenger's algorithm can only perform MSM calculations of 1024 points each time. First up, extract the lowest 4 bits of the scalar values and perform point addition of 1024 points. After the addition operations, transfer the cache points in the Buffer, which requires the buffer transition control unit to complete. Secondly, after the point addition operations of 1024 scalars, the next Pippenger of 1024 points needs the Buffer transition control unit to load the Buffer point corresponding to the lowest 4 bits of last time. Then continue the operation, which will also be completed by the Buffer transition control unit.

After the point multiplication operations using Pippenger's algorithm, we need to perform point doubling and point addition operations on the result of the Buffer cache points we get from calculations. Finally, output the operation result of MSM.

11 Key Technologies

This section introduces the key technical principles used in the design of OlaVM.

11.1 Permutation Argument

Step 1: Label and define permutation index polynomial S

Permutation Argument explicitly specifies the exact location of the equal value, just as $V_3(1) = V_5(0)$ shown in Figure 21. Therefore, we need to define a new polynomial to represent the index correspondence between different polynomials.

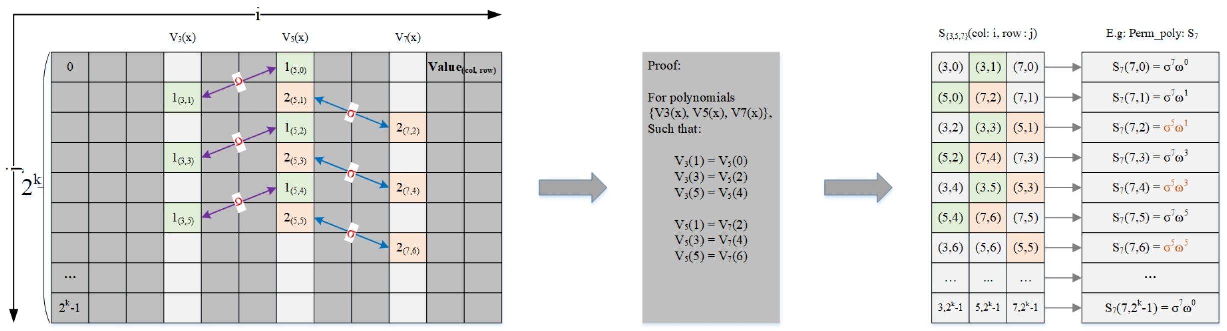


Figure 21: Permutation index polynomial

Since different polynomials correspond to different column indices, we can distinguish different evaluation domains by introducing column indices (distinguished through introducing a domain of size n in Plonk, i.e. if there are k polynomials, the domain size of `perm_index_poly` is kn). The mapping can be expressed as

$$S(\text{col:}i, \text{row:}j) = \sigma^{i'}\omega^{j'}$$

Taking the polynomial $V_7(X)$ of column index = 7 as an example, the truth value of its corresponding permutation index polynomial $S_7(\text{col:}i, \text{row:}j)$ is shown in the right side of Figure 21.

Step 2: Generate the proof

For the permutation index polynomial set $S = \{S_i\}$ generated based on step 1, we need to ensure the following equation holds:

$$\prod_{j=0}^{n-1} \frac{\left(V_3(\omega^j) + \beta \cdot \sigma^3 \cdot \omega^j + \gamma\right)\left(V_5(\omega^j) + \beta \cdot \sigma^5 \cdot \omega^j + \gamma\right)\left(V_7(\omega^j) + \beta \cdot \sigma^7 \cdot \omega^j + \gamma\right)}{\left(V_3(\omega^j) + \beta \cdot S(3, \omega^j) + \gamma\right)\left(V_5(\omega^j) + \beta \cdot S(5, \omega^j) + \gamma\right)\left(V_7(\omega^j) + \beta \cdot S(7, \omega^j) + \gamma\right)} = 1$$

Note: If we reorder the indices of three polynomials (starting from 0 regardless of the original order, then the polynomial indices will be 0, 1 and 2 successively), the above equation can be simply expressed as

$$\prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \frac{V_i(\omega^j) + \beta \cdot \sigma^i \cdot \omega^j + \gamma}{V_i(\omega^j) + \beta \cdot S_i(\omega^j) + \gamma} = 1$$

where the permutation index polynomial is defined as

$$S_{\text{col}:i}\left(\omega^{\text{row}:j}\right) = \sigma^{i'} \omega^{j'}$$

The principle is shown in Figure 22.

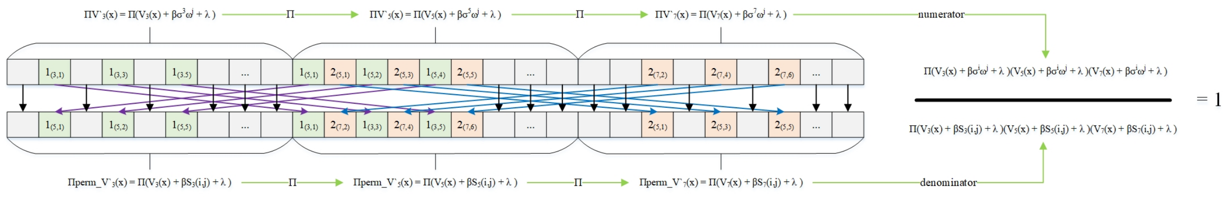

ΠV`3(x) = Π(V3(x) + βσ³ωʲ + λ)
Π
ΠV`5(x) = Π(V5(x) + βσ⁵ωʲ + λ)
Π
ΠV`7(x) = Π(V7(x) + βσ⁷ωʲ + λ)
numerator
1(3,1) 1(3,3) 1(3,5) ... 1(5,1) 2(5,1) 1(5,2) 2(5,3) 1(5,4) 2(5,5) ... 2(7,2) 2(7,4) 2(7,6) ...
1(5,1) 1(5,2) 1(5,5) ... 1(3,1) 2(7,2) 1(3,3) 2(7,4) 1(3,5) 2(7,6) ... 2(5,1) 2(5,3) 2(5,5) ...
Π(V3(x) + βσⁱωʲ + λ)(V5(x) + βσⁱωʲ + λ)(V7(x) + βσⁱωʲ + λ)
= 1
Π(V3(x) + βS3(i,j) + λ)(V5(x) + βS5(i,j) + λ)(V7(x) + βS7(i,j) + λ)
Πperm_V`3(x) = Π(V3(x) + βS3(i,j) + λ)
Π
Πperm_V`5(x) = Π(V5(x) + βS5(i,j) + λ)
Π
Πperm_V`7(x) = Π(V7(x) + βS7(i,j) + λ)
denominator

Figure 22: Accumulator polynomial

Due to the introduction of random numbers β and γ, we can guarantee the two cells connected by purple lines and blue lines in Figure 22 to be equal with a probability close to 1 if and only if the copy constraints are satisfied according to Schwartz-Zippel Lemma. Finally, the multiplication result of the numerators is equal to that of denominators. Define the multiplicative polynomial Z_P as follows:

$$Z_P(W^0) = 1$$

$$Z_P(W^{j+1}) = \prod_{h=0}^{j} \prod_{i=0}^{m-1} \frac{V_i(\omega^h) + \beta \cdot \sigma^i \cdot \omega^h + \gamma}{V_i(\omega^h) + \beta \cdot S_i(\omega^h) + \gamma}$$

$$= Z_P(W^j) \prod_{i=0}^{m-1} \frac{V_i(\omega^j) + \beta \cdot \sigma^i \cdot \omega^j + \gamma}{V_i(\omega^j) + \beta \cdot S_i(\omega^j) + \gamma}$$

It should satisfy the following constraints

$$Z_P(\omega X) \prod_{i=0}^{m-1} \left(V_i(x) + \beta \cdot S_i(X) + \gamma\right) - Z_P(X) \prod_{i=0}^{m-1} \left(V_i(X) + \beta \cdot \sigma^i \cdot X + \gamma\right) = 0$$

$$l_0 \cdot (1 - Z_P(X)) = 0$$

For further understanding of the principle, see Permutation Argument [29].

11.2 Lookup Argument

Lookup Argument is used to prove that the elements of the two sets are of the same type, while the size can differ, therefore, for two sets A and S, if we want to prove that they contain the same elements, first we need to remove duplicates and then we obtain two new sets A' and S'. By performing LDT on them, we obtain two polynomials $A'(X)$

and $S'(X)$. According to the Schwartz-Zippel Lemma, we can know that the following constraints stand if and only if the set A' contain the same elements as S'.

$$Z(\omega X)(A'(X)+\beta)(S'(X)+\gamma)-Z(X)(A(X)+\beta)(S(X)+\gamma)=0$$
$$(1-Z(X))l_0(X)=0$$

For further understanding of the principle, refer to Lookup Argument [28] and Plookup [5].

11.3 Combined Selector

This section mainly introduces how to improve the efficiency of ZK by combining selectors. First up, it introduces why selectors appear, and then how to combining multiple selectors.

11.3.1 Custom Gate

When designing ZKVM circuit, many binary selectors are introduced due to the vast amount of custom gates. Looking at the field division gate as an example, we plan to design a gate to verify the equation $q = x/y$ between three field elements q, x, y. For convenience, we will not implement field division operation at circuit level, but by checking the equations

$$x \cdot y^{-1} = q$$
$$y^{-1} \cdot y = 1 \quad \text{(ensure } y \neq 0)$$

They are equivalent, so we have the following trace table:

clk	s_*	$\cdots$	s_{div}	$\cdots$	w_0	w_1	w_2	w_3	Notes
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	0	0	1	0	x	y	q	y^{-1}	division
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table 17: A simple example of execution trace

For columns w_0, w_1, w_2, w_3, define polynomials $w_0(x), w_1(x), w_2(x), w_3(x)$, then on the row corresponding to the division operation, they should satisfy

$$w_0(x) \cdot w_3(x) - w_2(x) = 0$$
$$w_1(x) \cdot w_3(x) - 1 = 0$$

In order to ensure the satisfaction of the above relationships on the corresponding row, it is necessary to enable the corresponding division to constrain the verification through a selector, for which we introduce a new column, like $s_{\text{div}} = \{0, 0, \dots, 1, \dots, 0\}$. After converting to polynomial $s_{\text{div}}(x)$, the above equations become

$$s_{\text{div}}(x) \cdot (w_0(x) \cdot w_3(x) - w_2(x)) = 0$$
$$s_{\text{div}}(x) \cdot (w_1(x) \cdot w_3(x) - 1) = 0$$

11.3.2 Combined Selector

According to the above example, we know that whenever we define a new custom gate, we need to introduce a selector column s_* related to the gate, called $t_*(x)$, then we have the following constraints

$$s_{\text{add}}(x) \cdot t_{\text{add}}(x) = 0$$
$$s_{\text{div}}(x) \cdot t_{\text{div}}(x) = 0$$
$$s_{\text{cube}}(x) \cdot t_{\text{cube}}(x) = 0$$
$$s_{\text{sqrt}}(x) \cdot t_{\text{sqrt}}(x) = 0$$

Since the prover needs to make commitments to all polynomials when generating proofs, the introduction of too many selector polynomials will increase the workload of both the prover and verifier. Therefore, we need to optimize the number of selectors, which requires to meet two conditions:

1. Without losing the meaning of selector polynomials, i.e. only specific gates can be allowed;
2. Fewer selector polynomials.

Plonky2 [19] shows a Binary-Tree based Selector optimization method, which reduces the number of selector polynomials to $\log(k)$, where k is the number of custom gates; in Halo2 [11], ZCash team shares a new optimization method that may achieve a smaller number of polynomials (which is related to the constraint polynomial $t_*(x)$ and the parameter `max_degree` set to the constraint polynomial).

We've taken forth a simple scenario to ease the understanding (for detailed algorithm, please refer to Selector Combining [25]).

clk	s_{add}	s_{div}	s_{cube}	s_{sqrt}	w_0	w_1	w_2	w_3	Notes
0	1	0	0	0	a_0	b_0	c_0	d_0	addition
1	0	1	0	0	a_1	b_1	c_1	d_1	division
2	0	0	1	0	a_2	b_2	c_2	d_2	cube
3	0	0	0	1	a_3	b_3	c_3	d_3	sqrt

Table 18: Multi-operation example of execution trace

As can be seen, we have set 4 selector columns for the four custom gates, which are not what we want, thus increasing the workload of the prover and verifier. Then we define a new column q, satisfying

$$q = \begin{cases} k & \text{if the selector labelled } k \text{ is } 1 \\ 0 & \text{if all the selectors are } 0 \end{cases}$$

If we define a set $\{s_{\text{add}}, s_{\text{div}}, s_{\text{cube}}, s_{\text{sqrt}}\}$ for selectors $s_{\text{add}}, s_{\text{div}}, s_{\text{cube}}, s_{\text{sqrt}}$ (we call this set disjoint, because the rows enabled are not intersecting), combining the definition of column q, we have

clk	s_{add}	s_{div}	s_{cube}	s_{sqrt}	q	w_0	w_1	w_2	w_3	Notes
0	1	0	0	0	1	a_0	b_0	c_0	d_0	addition
1	0	1	0	0	2	a_1	b_1	c_1	d_1	division
2	0	0	1	0	3	a_2	b_2	c_2	d_2	cube
3	0	0	0	1	4	a_3	b_3	c_3	d_3	sqrt

Table 19: Add combined column q

Then we define a new selector polynomial form based on column q. The k-th selector polynomial is

$$q(x) \prod_{\substack{h=1 \\ h \neq k}}^{\text{len(selectors)}} (h - q(x))$$

For example, for constraint $s_{\text{add}} \cdot t_{\text{add}}(x) = 0$, we can rewrite it as

$$q(x) \prod_{\substack{h=1 \\ h \neq 1}}^{\text{len(selectors)}} (h - q(x)) \cdot t_{\text{add}}(x) = 0$$

The above equation can be expanded into

$$q(x) \cdot (2 - q(x)) \cdot (3 - q(x)) \cdot (4 - q(x)) \cdot t_{\text{add}}(x) = 0$$

Define

$$\text{combined}_q(x) = q(x) \cdot (2 - q(x)) \cdot (3 - q(x)) \cdot (4 - q(x))$$

then we can derive its truth table:

clk	$\text{combined}_q(x)$	$s_{\text{add}}(x)$
0	1	1
1	0	0
2	0	0
3	0	0

Table 20: Consistency between q and s_{add}

This achieves the same function as the original selector, therefore, for constraints

$$s_{\text{add}}(x) \cdot t_{\text{add}}(x) = 0$$
$$s_{\text{div}}(x) \cdot t_{\text{add}}(x) = 0$$
$$s_{\text{cube}}(x) \cdot t_{\text{cube}}(x) = 0$$
$$s_{\text{sqrt}}(x) \cdot t_{\text{sqrt}}(x) = 0$$

we can rewrite them as

$$q(x) \prod_{\substack{h=1 \\ h \neq 1}}^{\text{len(selectors)}} (h - q(x)) \cdot t_{\text{add}}(x) = 0$$
$$q(x) \prod_{\substack{h=1 \\ h \neq 2}}^{\text{len(selectors)}} (h - q(x)) \cdot t_{\text{add}}(x) = 0$$
$$q(x) \prod_{\substack{h=1 \\ h \neq 3}}^{\text{len(selectors)}} (h - q(x)) \cdot t_{\text{cube}}(x) = 0$$
$$q(x) \prod_{\substack{h=1 \\ h \neq 4}}^{\text{len(selectors)}} (h - q(x)) \cdot t_{\text{sqrt}}(x) = 0$$

For constraints of the types above, we only need to make commitments to the polynomial $q(x)$. But we should note that this method also increases the degree of constraints.

So far, we have achieved the two points mentioned above:

1. Without losing the meanings of selector polynomials, i.e. only specific gates can be allowed;
2. Fewer selector polynomials.

Of course, due to the limitation of degree of constraints in the protocol, the selectors that can be combined are limited. The number of selectors combined at a time depends on the degree of the constraint polynomial $t_*(x)$ and the boundary value `max_degree`. Therefore, we may need multiple combined columns, even so the number of which is much smaller than the original number.

11.4 Sinsemilla Hash

Sinsemilla Hash is a Lookup Argument-friendly hash algorithm based on the complexity of discrete logarithm problem and collision resistance (the length of input is fixed). Compared with other algebraic hash algorithms, such as Rescue and Poseiden, Sinsemilla Hash has certain advantages and disadvantages:

- Advantage: Sinsemilla Hash executes 19 times faster than Rescue and Poseiden Hash when executed out of circuit;
- Disadvantage: When executing with circuits, Sinsemilla Hash is 4 times slower than Rescue and Poseiden Hash.

Considering the local execution of hash calculation when generating the execution trace, Sinsemilla Hash is a better option from the perspective of the overall performance (ranging from data preparation to proof generation). Next let's take a look at the algorithm details of Sinsemilla Hash.

11.4.1 Setup

1. Select the split parameter k;
2. Generate $2^k + 1$ independent points $\{Q, P_0, P_1, \ldots, P_{2^k-1}\}$.

11.4.2 Hash(M)

1. Split M into n groups of k bits, the i-th group is represented as m_i in little endian;
2. Set the initial value $\mathrm{Acc}_0 = Q$.
3. For i from 0 to $n-1$, execute loop calculation $\mathrm{Acc}_{i+1} = (\mathrm{Acc}_i + P_{m_{i+1}}) + \mathrm{Acc}_i$.
4. Finally we get Acc_n.
5. $\mathrm{ShortHash}(M)$ returns to the x-coordinate of $\mathrm{Hash}(M)$.

11.4.3 Incomplete Addition

In step 3 of $\mathrm{Hash}(M)$, we have performed operation $\mathrm{Acc}_{i+1} = (\mathrm{Acc}_i + P_{m_{i+1}}) + \mathrm{Acc}_i$, where the elliptic curve addition algorithm used here is Incomplete addition, i.e., it can only process elliptic curve addition in valid input scenarios (the inputs are different points on the elliptic curve). The other one is a complete addition. The specific differences of the two calculation forms are shown in Incomplete and complete addition – The halo2 Book [11].

- Input: $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$
- Output: $R = P + Q = (x_R, y_R)$

According to the introduction in Section 4.1 of [12], the Incomplete addition formula can be expressed as

$$x_R = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_Q - x_P$$
$$y_R = \frac{y_Q - y_P}{x_Q - x_P} \cdot (x_Q - x_R) - y_Q$$

According to the definition of Incomplete addition, we have $x_Q \neq x_P$, so the previous formula can be transformed into

$$\begin{aligned}
x_R &= \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_Q - x_P \\
\implies x_R + x_Q + x_P &= \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 \\
\implies (x_R + x_Q + x_P)(x_Q - x_P)^2 &- (y_Q - y_P)^2 = 0 \\
y_R &= \frac{y_Q - y_P}{x_Q - x_P} \cdot (x_Q - x_R) - y_Q \\
\implies (y_R + y_Q)(x_Q - x_P) &- (y_Q - y_P)(x_Q - x_R) = 0
\end{aligned}$$

So, for Incomplete addition of elliptic curve points, we have the following constraints:

Degree	Constraint
4	$q_{\text{incomplete-add}} \cdot ((x_R + x_Q + x_P)(x_Q - x_P)^2 - (y_Q - y_P)^2) = 0$
3	$q_{\text{incomplete-add}} \cdot ((y_R + y_Q)(x_Q - x_P) - (y_Q - y_P)(x_Q - x_R)) = 0$

Table 21: Constraints for incomplete addition

Let $\lambda = (y_Q - y_P)/(x_Q - x_P)$, the above constraints can be changed to:

Degree	Constraint
3	$q_{\text{incomplete-add}} \cdot (x_R + x_Q + x_P - \lambda^2) = 0$
3	$q_{\text{incomplete-add}} \cdot (\lambda(x_Q - x_R) - (y_Q + y_R)) = 0$
3	$q_{\text{incomplete-add}} \cdot (\lambda(x_Q - x_P) - (y_Q - y_P)) = 0$

Table 22: Improved constraints for incomplete addition

11.4.4 Lookup for Sinsemilla

According to the calculation process of Sinsemilla, we need to split the original information M into n sub-information m_i with a bit width of k (since Sinsemilla has fixed-length input requirements, it's necessary to pad M to a fixed length). Let $P_{m_i} = (x_{P,i}, y_{P,i})$, since $m_i \in [0, 2^k)$, k will not be large in general, then we can precompute a table for query as Table 23.

12 Summary

We covered the complete design of OlaVM in previous sections, including VM design, ZKVM constraint design, constraint ideas, logic behind each module and much more. We believe that this content will give a fundamental understanding of ZKVM design process. Due to the limited space, we are working on a new, detailed, "*module by*

i	x-coordinate	y-coordinate
0	$x_{P,0}$	$y_{P,0}$
1	$x_{P,1}$	$y_{P,1}$
...	...	...
$2^k - 1$	$x_{P,2^k-1}$	$y_{P,2^k-1}$

Table 23: Lookup table for Sinsemilla Hash

module" paper, focused on the subsequent engineering implementation process, which includes the design on how to support further EVM instructions in OlaVM. Related ZKVM technologies are under constant development and we continuously keep up to date with recent research, such as, e.g. new ZK-friendly Hash and more efficient Lookup Argument [31], in order to include this in OlaVM design.

In addition to this, we want to show our gratitude for the hard work of all the prominent teams in the space of Zero Knowledge, of which, naming a few, PSE, Matter Labs, Polygon Hermez and StarkWare. We have learnt a lot through their contributions in open source documentation, code and online sharing amongst other, on how to design, improve ZK-efficiency and construct a ZKVM. We want to direct a special thanks towards Justin Drake[1], Barry Whitehat[2] and others, whom we've had educational and inspirational information exchanges with, enlightening us on certain aspects in ZKEVM design, providing us with a better understanding on how to proceed. On an ending note, there is still a lot to be done, research to be conducted, knowledge to be acquired and room for improvement to be identified.

Bibliography

[1] **Awesome Ethereum Virtual Machine**. URL: https://github.com/pirapira/awesome-ethereum-virtual-machine.

[2] Eli Ben-Sasson et al. "Succinct {Non-Interactive} zero knowledge for a von neumann architecture". In: **23rd USENIX Security Symposium (USENIX Security 14)**. 2014, pp. 781–796.

[3] Sean Bowe, Jack Grigg, and Daira Hopwood. "Halo: Recursive Proof Composition without a Trusted Setup". In: **IACR Cryptol. ePrint Arch.** 2019 (2019), p. 1021.

[4] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. "Fractal: Post-quantum and Transparent Recursive Proofs from Holography". In: **Advances in Cryptology – EUROCRYPT 2020**. Ed. by Anne Canteaut and Yuval Ishai. Cham: Springer International Publishing, 2020, pp. 769–793. ISBN: 978-3-030-45721-1.

[5] Ariel Gabizon and Zachary J. Williamson. "plookup: A simplified polynomial protocol for lookup tables". In: **IACR Cryptol. ePrint Arch.** (2020), p. 315. URL: https://eprint.iacr.org/2020/315.

[6] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". In: **IACR Cryptol. ePrint Arch.** 2019 (2019), p. 953.

[7] Lior Goldberg, Shahar Papini, and Michael Riabzev. **Cairo – a Turing-complete STARK-friendly CPU architecture**. Cryptology ePrint Archive, Paper 2021/1063. https://eprint.iacr.org/2021/1063. 2021. URL: https://eprint.iacr.org/2021/1063.

[8] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: **Advances in Cryptology – EUROCRYPT 2016**. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326. ISBN: 978-3-662-49896-5.

[9] Ulrik Günther et al. "scenery: Flexible Virtual Reality Visualization on the Java VM". In: **2019 IEEE Visualization Conference (VIS)**. 2019, pp. 1–5. DOI: 10.1109/VISUAL.2019.8933605.

[1] https://twitter.com/drakefjustin

[2] https://twitter.com/barrywhitehat

[10] **Hadamard checks from the Lagrange basis without FFTs**. URL: https://notes.ethereum.org/Il4z42lmQtaUYFigsjsk2Q.

[11] **halo2 - the Halo2 book**. URL: https://zcash.github.io/halo2/.

[12] Hüseyin Hişil. "Elliptic curves, group law, and efficient computation". PhD thesis. Queensland University of Technology, 2010. URL: https://eprints.qut.edu.au/33233/.

[13] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. "The Implementation of Lua 5.0." In: **J. Univers. Comput. Sci.** 11.7 (2005), pp. 1159–1176.

[14] Don Johnson, Alfred Menezes, and Scott Vanstone. **The Elliptic Curve Digital Signature Algorithm (ECDSA)**. 2001. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[15] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: **IACR Cryptol. ePrint Arch.** 2021 (2021), p. 370.

[16] **Matter Labs**. URL: https://github.com/matter-labs.

[17] Gregory Maxwell et al. **Simple Schnorr Multi-Signatures with Applications to Bitcoin**. Cryptology ePrint Archive, Paper 2018/068. https://eprint.iacr.org/2018/068. 2018. URL: https://eprint.iacr.org/2018/068.

[18] Nicholas Pippenger. "On the evaluation of powers and related problems". In: **17th Annual Symposium on Foundations of Computer Science (sfcs 1976)**. 1976, pp. 258–263. DOI: 10.1109/SFCS.1976.21.

[19] **Plonky2: Fast Recursive Argument with PLONK and FRI**. URL: https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf.

[20] **Polygon Hermez**. URL: https://github.com/hermeznetwork.

[21] **Privacy & Scaling Explorations (formerly known as appliedzkp)**. URL: https://github.com/privacy-scaling-explorations.

[22] **risc0/risc0: RISC Zero is a zero-knowledge verifiable general computing platform based on zk-STARKs and the RISC-V microarchitecture.** URL: https://github.com/risc0/risc0.

[23] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards". In: **J. Cryptology** 4 (1991), pp. 161–174. DOI: 10.1007/BF00196725.

[24] **Scroll**. URL: https://github.com/scroll-tech.

[25] **Selector combining**. URL: https://zcash.github.io/halo2/design/implementation/selector-combining.html.

[26] Srinath Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: Aug. 2020, pp. 704–737. ISBN: 978-3-030-56876-4. DOI: 10.1007/978-3-030-56877-1_25.

[27] Francesco Sica, Mathieu Ciet, and Jean-Jacques Quisquater. "Analysis of the Gallant-Lambert-Vanstone Method Based on Efficient Endomorphisms: Elliptic and Hyperelliptic Curves". In: **Selected Areas in Cryptography**. Ed. by Kaisa Nyberg and Howard Heys. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 21–36. ISBN: 978-3-540-36492-4.

[28] **Sin7Y Tech Review (14): Lookup Argument**. URL: https://hackmd.io/CCW1KrhiQueapJ_jT33brw.

[29] **Sin7Y Tech Review (15): Permutation Argument**. URL: https://hackmd.io/SWrXK1l3QluG-TT8MpUAKg.

[30] **StarkWare Libs**. URL: https://github.com/starkware-libs.

[31] Arantxa Zapico et al. **Caulk: Lookup Arguments in Sublinear Time**. Cryptology ePrint Archive, Paper 2022/621. https://eprint.iacr.org/2022/621. 2022. URL: https://eprint.iacr.org/2022/621.