Ola: A ZKVM-based, High-performance and Privacy-focused Layer2 platform

Sin7Y, Applied R&D Team*

April 2023

Abstract

We are developing a ZK-ZKVM platform for Ethereum [18] that will offer the following features: (1) Programmable privacy: supporting a vast variety of privacy design options for smart contracts. dApp developers can at an application level, develop private and public functions, and users can use these dApps to send both public and private transactions according to their needs; (2) Robust architecture: through the means of viewing keys which can be shared to third parties, we create a compliance-friendly landscape. Ola provides the possibillity for any entity to track the actual whereabouts of private assets, avoiding the disadvantages of untraceable assets, such as Tornado Cash [42]; (3) Flexible data sharing: indefinitely exposing your viewing key to a third party allows for continouous monitoring and tracking of an accounts status; (4) Data ownership: empowering users by returning the control of their own asset information and on-chain data; (5) High programmability: a customized GPL language which has more advanced features and is more programmable than DSL [16]; (6) High performance: customized full-featured zk-friendly ZKVM, OlaVM [28], can quickly execute transactions and generate proofs due to, amongst other, its non-deterministic computation and succinct instruction set; (7) Better language scalability: based on the LLVM compilation framework, it is easier to achieve compatibility with other advanced programming languages; (8) Developer-friendly language: Ola-lang [27] is customized for smart contract development with syntax similar to Rust, making it convenient for developers quickly adopt it; (9) Developer-friendly suite of tools: Ola-lang language plug-in is integrated into Visual Studio Code, making it convenient for traditional developers to start building. Ola is a rapidly developing project and this is the second whitepaper which will describe the design philosophy and specifications of our full-featured zk-friendly ZKVM, OlaVM [28] and developer-friendly general purpose smart contract language, Ola-lang [27] in details. Implementation details related to privacy will be released in the upcoming third whitepaper upon releasing the test net.

Keywords: Privacy, Programmability, ZKVM, Plonky2, Starky, Smart Contract, Data Ownership.

*https://twitter.com/Sin7Y_Labs

Contents

1 Introduction

In the past few years, we have witnessed a rapid development of the blockchain industry, influx of capital, users, and builders has enabled unprecedented growth and potential of the entire industry. However, blockchain is still somewhat in its infancy, infrastructure is still maturing. One of the largest challenges is the low transactional throughput of

public blockchain systems, especially the largest public programmable blockchain Ethereum [18], which in the best case scenario [19], processes up to 23 transactions per second, far less than traditional centralized systems.

There are many reasons for the congestion of the Ethereum network, such as the POW [36] consensus algorithm (The Merge upgrade [41] to POS [35] consensus now), the repeated execution of verification process of all transactions by nodes, and the serial execution of transactions; for the above reasons, many projects are trying to scale Ethereum through different means. Approach 1: Launching new public blockchains, independent of Ethereum, using faster consensus algorithms or faster transaction execution to achieve higher TPS, such as BSC [11], Solana [37], Aptos [4], etc; Approach 2: Ethereum Sidechain, using faster consensus algorithms and then regularly synchronizing to Ethereum, such as Polygon [33], Optimism [29], Arbitrum [5], etc; Approach 3: ZK(E)VM, a Layer2 network of Ethereum, using ZK technology to solve the problem of repeated execution of transactions, such as Polygon Hermez [34], Zksync [49] etc.

Scaling is not the only problem that has to be solved in the long term for Ethereum and the wider blockchain space. There are many excellent teams working rigorously on scaling solutions, the ola team believe that the next crucial feature to be implemented is privacy. Blockchains being a public ledger, all transactions taking place on-chain, meaning state changes containing the asset information related to an address or account is open and transparent.

Excessive information transparency leads to:

1. MEV (Maximal Extractable Value) issues. Miners can selectively package transactions based on its fee, resulting in transactions with lower fees having a lower likelihood of being processed, if ever, forcing users to increase their fees. More concerning:
2. Front running and censorship attacks, through the means of MEV. Block producers in programmable blockchains can use MEV to exploit certain smart contracts deployed on the blockchain (Such as DEXes and other DeFi products). They may also censor certain addresses. Such an attack could be that the miner or block producer sees a buy order in the memory pool (where transactions that are not yet appended to the blockchain are stored) and adds their own buy order on that particular DEX, thus increasing the price of the given token. The other transaction that is added is a sell order of given token and the third order is the buy order of the third party for the new increased price. Resulting in huge security issues and in the past year hackers have managed to steal assets roughly amounting to 2 billion dollars.
3. User data ownership problems, both the assets information and transaction information of addresses are open to being monitored and used, which is contrary to the vision of Web3 [43]. Therefore, when the scaling problem is solved, privacy will become the next urgent feature to achieve. In the world of blockchain, privacy is not a new topic, it has long been studied and supported by the Zcash team [45].

1.1 Non Programmable Privacy

In addition to Zcash, there are other public blockchains facilitating private transactions, such as Monero [25], Dash [15], Grin [23], etc. However, these blockchains only support basic private asset transfers but lack programmability. As a result, their ecosystem development significantly trails Ethereum, which has become the largest public blockchain in the ecosystem due to its programmability. Nevertheless, Ethereum does not have privacy features. To address this issue, some projects have started exploring ways to introduce privacy to Ethereum, such as the *ZK-ZKRollup* application zk.money, developed by the Aztec team. Unfortunately, the zk.money product has been discontinued, primarily because its privacy features were limited to basic single transfer scenarios. Given the current surge in decentralized finance

(DeFi) applications, asset transfers represent just one of the simplest financial scenarios. Consequently, the user base is limited, while maintenance costs continue to accumulate.

Therefore, some projects have begun to explore ways to introduce privacy to Ethereum, such as the *ZK-ZKRollup* application *zk.money* [48] developed by the *Aztec* [6] team. However, the current *zk.money* product has been discontinued, mainly because its privacy features only apply to simple single transfer scenarios. Given the current explosion of DeFi applications, asset transfer is only one of the simplest financial scenarios, and therefore the user base is limited, while the maintenance costs continue to accrue.

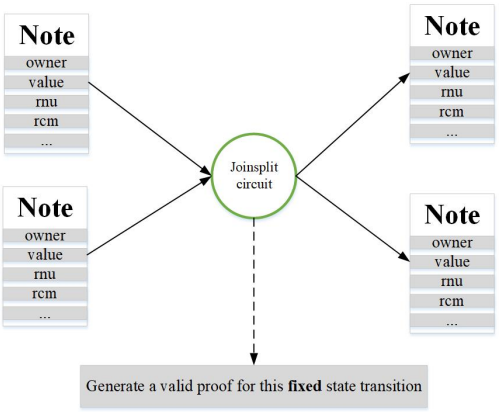


Figure 1: Example of Non Programmable Privacy

Figure 1 shows the simple logic of non-programmable privacy. The value change logic corresponding to the input and output notes in Section 5.2.3 are also fixed, generally in the form of "$A + B = C + D$". *Manta Network* [24] is a public blockchain that supports user-defined token privacy transfers, and the privacy transaction constraint circuits of all fungible tokens can be used to reuse the above logic.

A ZK-ZKRollup of a single-use case application, is similar to a corresponding ZKRollup of a single-use case application. If you want to use an asset in another application, you must bridge the asset through another protocol, which is generally a poor user experience. Therefore, just as ZKRollups need to transition to ZK(E)VMs, ZK-ZKRollup also needs to transition to ZK-ZKVMs (Appendix 7.2 explains how to get solidity compatibility).

1.2 Programmable Privacy

ZK-ZKVM has two main features:

1. It's a privacy-first platform, not privacy-only. Meaning that users can choose the transaction type, public or private, depending on their need. Just as in Zcash;
2. Programmability, you can deploy any smart contract, public or private, depending on the needs of the project side. Compared with non programmable privacy, the main difference is the logic of state transitions in a note 5.2.3, **Figure 2** simply shows the difference

Current projects focusing on programmable privacy are Aleo [1] and Aztec. Aleo is a privacy-first public blockchain, from Bitcoin [12] to Ethereum to Zcash [45] to Aleo. It brings the public blockchain into a new era, supporting programmable privacy. It has reached the testnet stage and allow developers to deploy privacy contracts on it; Aztec focuses on doing Layer2 programmable privacy for Ethereum, a project called Aztec3 [7], is still in development.

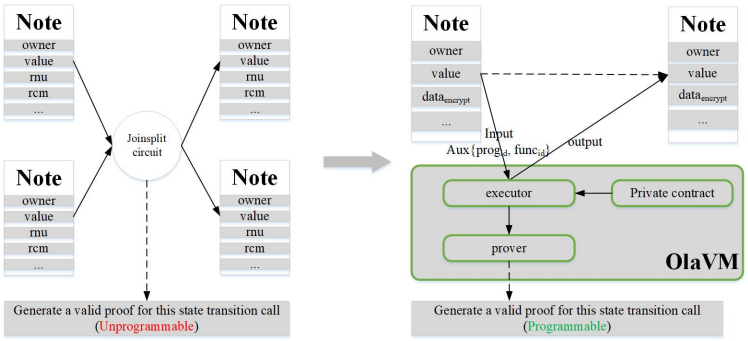


Figure 2: Difference between Non Programmable Privacy and Programmable Privacy

Before we clarify the different approaches on achieving programmability, we will clarify what *Domain Specific Language* (DSL) and *General Purpose Language* (GPL) [16] are. DSL is defined as a computer programming language of limited expressiveness focused on a particular domain, limited expressiveness means it just supports a bare minimum of features needed to support its domain. You can't build an entire software system in a DSL; rather, you use a DSL for one particular aspect of a system. While GPL is defined as a general-purpose programming language.

So there are often two ways to achieve programmability, one is to design a DSL, such as Circom [14], Pil [31], Noir [26], etc; the other is *Smart Contract Language* SCL, such as Cairo1.0 [13], Solidity [38], Ola lang [27] and so on. As we have mentioned before, the main difference is that SCL supports more complex structures and has higher abstraction, it's more suitable for writing complex business logic and meanwhile, and DSL is more suitable for some simple computational expressions. Take Pil [31] language as an example, you can directly use it to define a simple micro-op, such as "$A * B + C$", or "$A * B * C + D$" and other simple combinations. **Table 1** briefly shows some of the differences between DSL and SCL.

Type	Abstraction	Process	Difficulty	Examples	Notes
DSL	low	program -> arith-ops -> ops gadgets	normal	circom noir cairo	1. semantic analysis 2. codeGen optimization
SCL (ISA/VM)	high	program -> bytecodes -> cpu circuit	hard	solidity cairo1.0 ola lang	1. need a compiler 2. re-use LLVM framework

Table 1: Difference between DSL and SCL

If you want to prove a program written by DSL is executed correctly (this is what ZKDSL means), you may need to predefine some common operators, each of them corresponding to a circuit, called a gadget [21]. Developers can use these operators to implement desired functions, but it's difficult to handle the call and return logic between functions. Meanwhile, If you want to prove a program written by SCL is executed correctly (this is what ZKVM means), you need to design corresponding constraints for each instruction, collectively referred to as CPU circuit; therefore, Any program will be compiled into bytecodes composed of these instructions, and then constrained by the CPU circuit.

Ola achieved a customized SCL to get programmability even though it's harder to implement than DSL, because we could get benefits from it as follows:

- Higher abstraction and programmable language, allowing developers to write smart contracts with arbitrary

logic;

- A full-featured zk-friendly VM can be designed to achieve higher system performance;
- LLVM-based compiler can be more easily compatible with other advanced programming languages.

1.3 Full-featured ZK-friendly ZKVM

To achieve programmability, the optimal approach is to create a complete zk-friendly ZKVM with a customized instruction set, smart contract language, compiler, and more. A ZKVM is a virtual machine capable of executing any program while simultaneously generating a zero-knowledge proof of the correctness of the execution process. Therefore, the speed of proof generation is crucial, as it directly affects the system's performance.

The key to obtaining a full-featured zk-friendly ZKVM lies in obtaining (

1. The smallest possible execution trace
2. The most succinct state transition constraints
3. The fastest zero-knowledge proof algorithm

The smallest execution trace refers to the ability of OlaVM [28] to use the fewer instructions for the same computational logic, achieved through non-deterministic computation support at the computational level and register-based design at the memory access level.

The most succinct state transition constraints refer to the ability to constrain the entire execution trace of OlaVM for the same computational logic with the fewest polynomials and the smallest order, accomplished through Algebraic RISC architecture. The instruction sets size determines the CPU circuits complexity.

A faster zero-knowledge proof algorithm refers to the ability to complete the proof generation process in a shorter time for the same calculation using OlaVM. This primarily depends on the Goldilocks [22] field, a finite field less than 64-bit. Compared to the SNARK system based on large bit-width elements of elliptic curves, the STARK algorithm based on the Goldilocks [22] field is executed faster.

The subsequent chapters will elaborate on Ola's design philosophy and specifications for obtaining a full-featured zk-friendly ZKVM. Ola, a programmable privacy Layer2 network based on ZKVM, Ola will support the following features and use cases:

- Projects:
  - Developers can freely choose to deploy public contracts (Account-based), privacy contracts (Note-based), and ordinary contracts (Account and Note-based).
    1. For public contracts, Ola functions as a ZKVM;
    2. For privacy contracts, Ola functions as a ZK-ZKVM;
    3. For ordinary contracts, Ola functions as a ZK-ZKVM or ZKVM, depending on the user's transaction type;
  - Transfer of assets between public and private accounts
  - Intra-contract, no bridge contract is required, supported by default;
  - Cross-contract, a bridge contract is required;
- Users:
  - For ordinary contract types, users can freely choose the transaction type;
  - For public/private contract types, users can only execute transactions of the corresponding type;
  - Users have a view key to disclose executed private transactions;

– Ola supports the update of the view key so that after the view key is exposed, the privacy transactions executed by the user in the future will always be parsed;
– Ola supports asset transfers between public and private accounts.

1.4 Outline

Ola is a full stack developer framework for zero-knowledge applications, the whole framework is shown as **Figure 3**:

Figure 3: Ola framework

The green borders refers to the modules we have completed implementation, and others stand for the modules we will implement in the future. In the remaining sections of this whitepaper, we will introduce those modules as follows:

- Section 2 mainly describes the design of Ola's virtual machine, including our zk-friendly design schemes;
- Section 3 mainly describes the design of the customized smart contract language, Ola-lang and the framework of Ola compiler based LLVM;
- Section 4 mainly describes the design of Ola compiler;
- Section 5 mainly describes key points on achieving privacy;
- Section 6 mainly describes algorithms used in Ola, including zk and hardware acceleration algorithms;
- Section 7 mainly describes the key features we researched to be supported in the future and our frameworks around that;
- Section 8 mainly describes some basic notations;

2 Ola-VM: A Full-featured ZK-friendly ZKVM

2.1 Design Principles

2.1.1 Design of Instruction Set

OlaVM utilizes the *Reduced Instruction Set Computer* (RISC) architecture, which is characterized by a smaller instruction set compared to the *Complex Instruction Set Computer* (CISC) architecture. The contrast between RISC and CISC architectures can be further explored in the RISC vs. CISC comparison.

1. A reduced instruction set reduces the number of constraint polynomials

In ZKVM, there is a very critical constraint, the CSTC (CPU State Transition Constraint); it is mainly used to constrain the validity of VM state changes before and after the execution of an instruction.

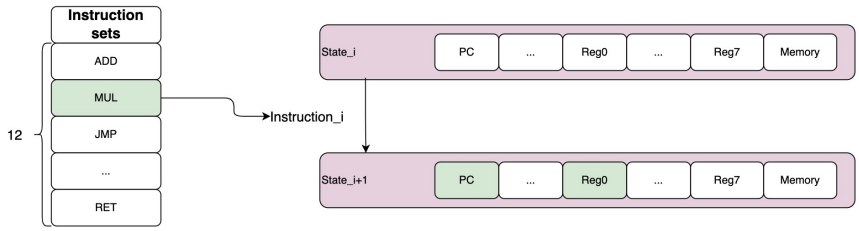


Figure 4: Instruction-induced cpu state transition

As shown in **Figure 4**, there are many registers in the VM. When the VM executes an instruction, it will cause the state of some registers to change, in order to ensure the correctness of the VM execution, we need to constrain the following:

- The change of register state involved in the execution of the instruction is valid;
- The status of registers not involved in the instruction execution remains unchanged.

Let's look at a simple example to explain the constraints on register state changes:

clk	pc	instruction	...	reg0	reg1	reg2	...
...	...	...	...	...	...	...	...
78	8	ADD reg0 reg0 reg1	...	1	2	0	...
79	9	MOV reg2 3	...	3	0	0	...
80	11	JMP 2	...	3	0	3	...
81	2	...	...	3	0	3	...

Table 2: Example of cpu state transitions

Table 2 briefly presents the changes of some registers after the VM executes three instructions; taking the PC register as an example, the logic of the updates of the three instructions are:

- ADD: $\mathrm{pc}_{i+1} = \mathrm{pc}_i + 1$
- MOV: $\mathrm{pc}_{i+1} = \mathrm{pc}_i + 2$
- JMP: $\mathrm{pc}_{i+1} = 2$

When considering constraints, we cannot predict which instruction the VM will execute at any given time. Therefore, we must design a constraint capable of handling all instructions. This constraint is known as the CPU State Transition Constraint (CSTC). In the aforementioned simple example, the update logic of the PC can be designed as follows:

$$\mathrm{pc}_{i+1} = \mathrm{sel\_jmp}_i \cdot \mathrm{imm\_value} + (1 - \mathrm{sel\_jmp}) \cdot (\mathrm{pc}_i + 1 + \mathrm{op1\_imm})$$

We express all CPU state transitions in this constrained form. The inclusion of each new instruction introduces additional selectors, which increases the complexity of the constraints. To manage this complexity, we limit the instruction size to 19, including built-ins.

Despite having a reduced instruction set architecture, it is still necessary to maintain Turing-completeness to ensure that we can perform any computation with these simple instructions. Combining these instructions with the

prophet mechanism described in Section 2.1.3, we have developed Ola-lang, a Turing-complete language that supports loops and recursive operations.

2. Most of the operations will use a small part of the instruction set

Under the CISC architecture, a vast instruction set is defined. However, in practice, only a small subset of the instructions is used for most computations. From a constraint standpoint, this represents a significant inefficiency. This is because the constraint logic for all instructions must be included in the constraint system (CSTC), and in the worst-case scenario, each instruction corresponds to a constraint factor. This complexity can lead to a large number of polynomials and high-order constraints, making the entire constraint system challenging to scale.

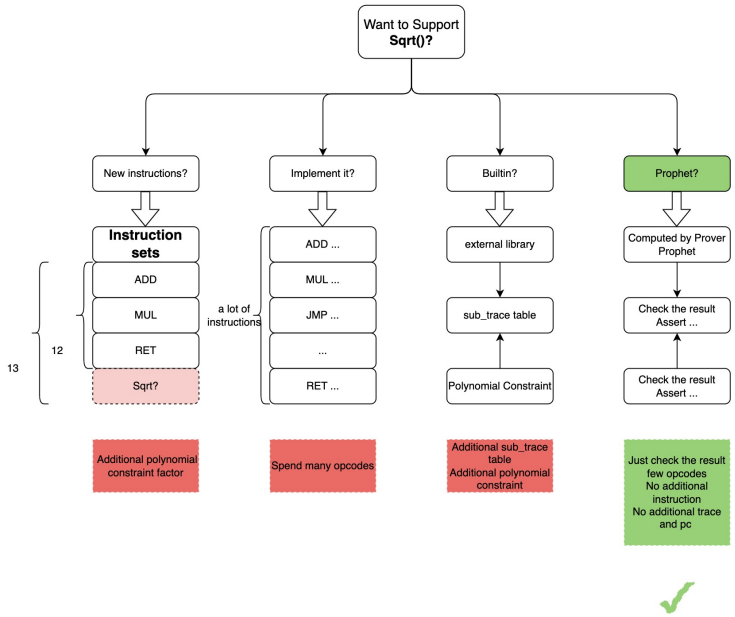


Figure 5: The best way to support ZK-Unfriendly computation

In many cases, when performing certain calculations, it is preferable to use the existing instruction set instead of introducing a new instruction. While this approach may increase the program size (in terms of opcodes), redundant data can be added during the verification process to facilitate FFT. However, for complex computations that require a significant number of instructions, other solutions are available. These include introducing Builtins and our Prophet module, which will be discussed in more detail in subsequent chapters. For now, it is important to note that they can significantly reduce the number of instructions needed to perform these complex computations, as demonstrated in **Figure 5**.

2.1.2 Design of Registers

As discussed in Section 2.1.1, in ZKVM, we must not only constrain the state of the registers associated with the instruction being updated correctly, but we must also ensure that the state of the registers not involved in the current instruction remains unchanged. While register-based VMs offer certain advantages, having a larger number of registers is not necessarily better from a verification standpoint. There is a trade-off that must be considered:

- How many registers will be sufficient for most of the calculations?

- For some calculations, is the increased number of memory accesses acceptable when there are not enough registers?

If the number of registers is small enough and memory accesses don't need to be constrained, this would be the ideal scenario for verification efficiency. The design of the Cairo VM exemplifies this approach, with no general-purpose registers and all instruction operands originating from memory. To avoid consistency checks on memory, a write-once model is employed, meaning memory can't be rewritten at the execution level, obviating the need for checks.

However, the write-once model's downside is that it's unfriendly to Dapp developers, and the memory model's limitations require developers to be extra mindful of memory usage. The traditional memory model is the read-write model.

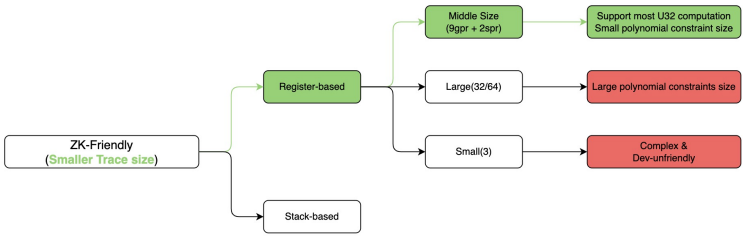


Figure 6: How to get ZK-Friendly(Smaller Trace) from register

As illustrated in Figure 6, we have opted for an optimal solution that takes into account both ZK-Friendly and Dev-Friendly perspectives. When it comes to the number of registers, we have considered the fact that the minimum type of computation that can be supported on OlaVM is U32 integer computation. With some boundary conditions such as the number of loops, loop termination conditions, etc., we have determined that nine general-purpose registers can support most of the U32 computation. Furthermore, both U64 and U256 integer computations can be implemented through the U32 library. OlaVM utilizes two special registers (PC and PSP) as well, as explained in Section 2.2.1.

If the number of registers is insufficient, we need to use memory for caching, which necessitates memory access-related instructions (MLOAD, MSTORE). It is important to note that in order to facilitate FFT, Provers often add redundant data in Trace to meet the size of Trace to the power of 2. These redundant data can be replaced with valid data, as demonstrated in Figure 7.

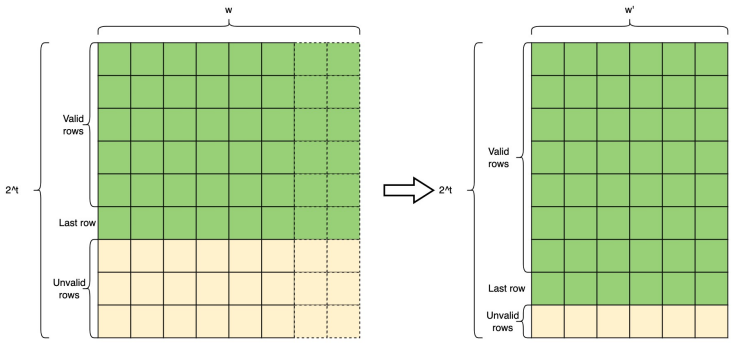


Figure 7: Better layout of Trace

2.1.3 Prophet

OlaVM employs a reduced instruction set, which limits its ability to support certain complex computations. The conventional approach to address this is to implement such computations using existing instructions. However, this method may consume a large number of opcodes, and there is an upper limit on the size of Trace. Therefore, to accommodate more computations in Trace, each opcode should occupy fewer trace cells. Alternatively, one can take inspiration from the design of EVM and use a pre-compiled contract to support complex computations. In this mode, there is no need to use EVM instructions to implement the computation; instead, one can add the corresponding pre-compiled contract, which can be written in any high-level language such as Rust or Go. However, this approach significantly increases the proof workload, requiring a separate trace table for the complex computation, and performing constraint proof only on the output of this computation and its input recorded in the CPU trace.

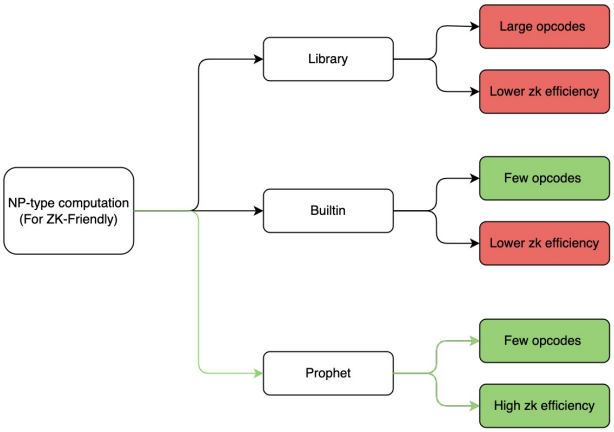


Figure 8: How the Prophet can get ZK-Friendly

It is desirable to support complex computations using few opcodes without increasing the complexity of the proof. To achieve this, we have devised a method called the "Prophet", depicted in **Figure 8**, which follows a prophecy and confirmation approach. This method fits well with the overall design of OlaVM.

For complex computations, the result of the complex computation is first computed locally by someone else (often Prover himself), without any constraints. Subsequently, the VM performs a validity check on the computed result before proceeding with the rest of the computation. This verification process is implemented using instructions and is included in the proof process.

For example, consider the task of calculating the square root of a radicand. While implementing Newton's method requires more than 20 instructions, verifying if a result is indeed the square root of a radicand can be done succinctly. ZKVM, for instance, only needs to multiply the result with itself to obtain a product and then verify if the product is equal to the radicand. This process requires only two instructions. With the second algorithm, how to compute the square root is not a concern for the ZK circuit and user. The computation process is wrapped up in the prophet segment and does not need to be verified by the ZK circuit. The ZK circuit only needs to verify the succinct logic $\texttt{result} \cdot \texttt{result} = \texttt{radicand}$. By using OlaVM to execute both algorithms and generate ZK proof, the second algorithm is ten times faster than the first traditional algorithm. (See Section 2.2.3 for more details.)

To ensure that OlaVM remains developer-friendly at a language level, it currently does not support developers to write their own Prophet implementations. Instead, the Prophet library is officially maintained and updated. Additionally,

to ensure security, memory access during Prophet execution is read-only. It's worth noting that not all complex calculations are suitable for this model. Calculations that are complex to implement but efficient to verify, such as square root and cube root, are more suitable for this approach. However, there are also some calculations that are not complex but still ZK-unfriendly, such as bitwise operations, range checks, hashing, and other calculations. These types of calculations are not suitable for Prophet's design. Instead, we use the pre-compiled approach to support them, which is similar to the pre-compiled contract function in EVM. We call this approach "Builtin". In the subsequent chapters, we will introduce the type of Builtin and the way of constraint in detail.

2.1.4 Memory

The memory design is influenced by the prophet. To ensure that the memory trace table can be written by prophet without use instructions, the memory space need be partitioned into several segment. The prophet segment is write-once for constraint, reference: Section 2.2.2

2.2 OlaVM

OlaVM is a zero-knowledge virtual machine (ZKVM). It executes the program compiled from the Ola-lang to generate trace tables. The traces are used by ZK prover to construct the ZK proof. The program format provided by Ola-lang compiler consists of two fields (bytecode and prophets):

- `bytecode` consists of instructions and can not be empty.
- `prophets` is a array contain all prophets and can be empty.

The program snippet is as below:

```
{
  "bytecode": "0x6000080400000000\n0x5\n0x2010000001000000\n0xffffffffffffffff\n0x
      4000000840000000\n0x0\n0x0030000001000000\n",
  "prophets": [
    {
      "host": 41,
      "code": "%{\n    entry() {\n        cid.y = sqrt(cid.x);\n    }\n%}",
      "inputs": [
        {
          "name": "cid.x",
          "stored_in": "reg",
          "anchor": "r1",
          "offset": 0
        }
      ],
      "outputs": [
        "cid.y"
      ]
    }
  ]
}
```

OlaVM contains the following modules:

- `instructions set`

- `prophet`
- `builtin`
- `memory`
- `trace generator`

2.2.1 Instruction Set

OlaVM uses different spaces between instructions and memory addresses. It is because the OlaVM has sufficient space for use, and using a unified space can make the column for storing constraints smaller, reducing overhead.

Register Currently, there are two main models for designing the L0 layer of the OlaVM memory hierarchy in the industry, one is the register model and the other is the stack model. For the stack model, state updates are implemented according to the first-in-last-out model of the stack.The constraints are relatively simple. However, the disadvantage is that accessing the state is not user-friendly. It is not possible to use one instruction to randomly access data at a specific address in the stack.

In contrast, the register model only requires one instruction to complete a randomly read operation, while the stack model requires multiple pop and push operations. Alternatively, to improve efficiency, swap-like instructions are added in the MidenVM design to optimize randomly accessed data in the stack. For the register model, any registers can be randomly accessed. However, the disadvantage is that the constraint model requires copy constraints for the upper and lower instructions and more columns for register selectors. The columns in the trace table are larger than the stack model.

As designed in the previous version of the whitepaper,OlaVM selects the register model.OlaVM includes 9 general-purpose registers. The symbol denotes as: $\mathtt{r}_0 - \mathtt{r}_8$. The $\mathtt{r}_8$ is used as `fp` (frame pointer). `fp` is the alias of $\mathtt{r}_8$.

The PC (program counter) pointer of a loaded program initially points to the instruction at address 0. As instructions are executed, the value of the address pointed to by the PC changes. If a jump or call instruction is not executed, the address stored in the PC register is incremented by 1 for each instruction executed, unless the instruction uses an immediate value, in which case the PC is incremented by 2.

The PSP (prophet stack pointer) register is used to alloc the prophet memory segment when OlaVM execute prophet code.In the initial state, set the prophet memory segment start address ($2^{64} - 2 \cdot 2^{32}$) to this register.

OlaVM instructions To minimize the degree of constraints, OlaVM use reduced instruction set.

OlaVM use Words (64bits) to encode instructions. An instruction is encoded with an opcode and up to three operands, which can be either a register or an immediate value. An instruction encoding consists of the following 8 fields:

- Field1: Field name: NULL.1 bit width,not used.
- Field2: Field name: OP1 immediate flag (OP1_IMM, imm). 1-bit width. Set to 0, indicates that A is a register; set to 1, indicates that A is an immediate value.
- Field3: Field name: operation0 register (OP0_REG, reg_src1). 9-bit width. Indicates source op0 register index, index range from 0 to 8. Set to 1 determine operating the register corresponding to the index, otherwise set to 0.

- Field4: Field name: operation1 register (OP1_REG, reg_src2). 9-bit width. Indicates source op1 register index, index range from 0 to 8. Set to 1 determine operating the register corresponding to the index, otherwise set to 0.If all 9 bits set to 0, indicates use PSP (prophet stack pointer) register.
- Field5: Field name: destination register (DST_REG, reg_dst). 9-bit width. Indicates destination register index, index range from 0 to 8. Set to 1 determine operating the register corresponding to the index, otherwise set to 0.
- Field6: Field name: opcode selector (opcode_sel). 21-bit width. Indicates the opcode type, set to 1 determine executing the opcode, otherwise set to 0.
- Field7: Field name: padding bits (paddings). 14-bit width. All bits padding by 0, will be used in the future.
- Field8: Field name: immediate data (immediate). This field is an option, determined by field2 if field2 is set to 1. This field indicates an immediate value, This field is 2 Word wide.

The instruction set of OlaVM is as below, A denoted intermediate or register:

Type	*Instruction*	*Operands*	*Description*	*flag*
Field Operation	ADD	ri rj A	Compute rj + A and store the result in ri	
	MUL	ri rj A	Compute rj * A and store the the result in ri	
	NOT	ri A	Compute not A and store the the result in ri	
Cmp	EQ	ri rj A	Equality comparison and store the the result in ri	rj = A
	NEQ	ri rj A	check whether rj is not equal A and store the the result in ri	rj != A
	ASSERT	ri A	Assert ri == A and vm will hang up if assertion fail	
Move	MOV	ri A	Copy the data of A to ri	
Flow	JMP	A	Set PC equal A	
	END		only appears at the end of the program	
	CJMP	rj A	If rj = 1, set PC equal A, else increment PC as usual	
	CALL	A	The Call instruction consists of the following steps 1. store return PC to the frame address [fp-1]. 2. jump to the address A	
	RET		The Ret instruction consists of the following two steps 1. use the memory address stored in the fp register to find the returned PC and jump to the location of the returned PC . 2. update the fp register to the fp before the call	
RAM	MLOAD	ri [A, imm]	Read the value from memory [A+imm] and store it into ri. imm is optional, can not represent. A can be a intermediate data or register value. if A is a register, can follow a immediate data.	
	MSTORE	[A, imm] rj	Read the value from register rj and store it into memory [A+imm]. imm is optional, can not represent. A can be a intermediate data or register value. if A is a register, can follow a immediate data.	
BUILTIN	RANGE CHECK	rj	range check the value in rj	
	AND	ri rj A	Compute rj and A and store the the result in ri	
	OR	ri rj A	Compute rj or A and store the the result in ri	
	XOR	ri rj A	Compute rj xor A and store the the result in ri	
	GTE	ri rj A	check whether rj is great than or equal to A and store the the result in ri	rj >= A

Table 3: Instruction set

As per the above definition: instruction encoding uses 2W width when there is no immediate value, and 4W width when there is an immediate value. The alloc principle of operand: when register state changes, the register is set as DST_REG operand. when the register state does not change, the register is set as OP0_REG operand and OP1_REG operand. If only use one register state not changes, use OP1_REG operand. The instruction encodes format as the table below:

NULL	OP1_IMM	OP0_R8	OP0_R7	OP0_R6	OP0_R5	OP0_R4	OP0_R3	OP0_R2	OP0_R1	OP0_R0	OP1_R8	OP1_R7	OP1_R6	OP1_R5	OP1_R4
OP1_R3	OP1_R2	OP1_R1	OP1_R0	DST_R8	DST_R7	DST_R6	DST_R5	DST_R4	DST_R3	DST_R2	DST_R1	DST_R0	ADD	MUL	EQ
ASSERT	MOV	JMP	CJMP	CALL	RET	MLOAD	MSTORE	END	RANGE_CHECK	AND	OR	XOR	NOT	NEQ	GTE
padding															
immediate data 48-63 bits															
immediate data 32-47 bits															
immediate data 16-31 bits															
immediate data 0-15 bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 4: OlaVM instruction encode format

The pseudocode for the state transition of executing instructions is as follows:

```
new_op:
opcode = opcode_list[instuction[pc].opcode_sel]
imm = instuction[pc].imm
ri = reg_dst
rj = reg_src1

if imm == 1:
    next = pc + 2
    A = immediate
else
    next = pc + 1
    A = reg_src2
endif

switch opcode:
    ADD:
        ri = rj + A
        break
    MUL:
        ri = rj * A
        break
    EQ:
        if rj == A:
            ri = 1
        else
            ri = 0
        break
    MOV:
        ri = A
        break
    JMP:
        pc = A
        goto new_op
    CJMP:
        if rj == 1:
            pc = A
        goto new_op
    CALL:
        [fp-1] = next
        pc = A
        goto new_op
    RET:
        fp = [fp-2]
        pc = [fp-1]
        goto new_op
    MLOAD:
        ri = [A]
        break
    MSTORE:
        [A] = ri
```

```
        break
    ASSERT:
        if ri != A:
           panic("ASSERT failed")
    RANGE_CHECK:
        insert ri value to rangetable trace table
        break
    AND:
        ri = rj & A
        break
    OR:
        ri = rj | A
        break
    XOR:
        ri = rj xor A
        break
    NOT:
        ri = not
        break
    NEQ:
        ri = rj != A
        break
    GTE:
        ri = rj >= A
        break
pc = next
goto new_op
```

procedure call standard Stack space to function. When executing a ret instruction, the fp decrease and reclaim stack space allocated to the function. After executing a call instruction, fp points to the callee function stack. The return PC address stored in:`[fp-1]`, the caller fp address stored in `[fp-2]`, the first three input parameters stored in $r_1 - r_3$. From the fourth parameter, the parameters are stored in descending order: `[fp-3]`, `[fp-4]` $\cdots$. The local variables are stored in the stack from `[fp]` in ascending order. The return value stored in r_0.

For example: function `foo(a: felt, b: felt, c: felt, d: felt, e: felt)`, input parameters: `a=0x1, b=0x2, c=0x3, d=0x4, e=0x5`.

```
func foo(a: felt, b: felt, c: felt, d: felt, e: felt)
    let sum = 0;
    sum = a+b;
    sum = sum * c;
    sum = sum + d + e
    return sum;
```

The `fp, PC` and memory state transition when function call is executed is as **Figure 9** (yellow represent memory address, red represent instruction address, blue represent registers):

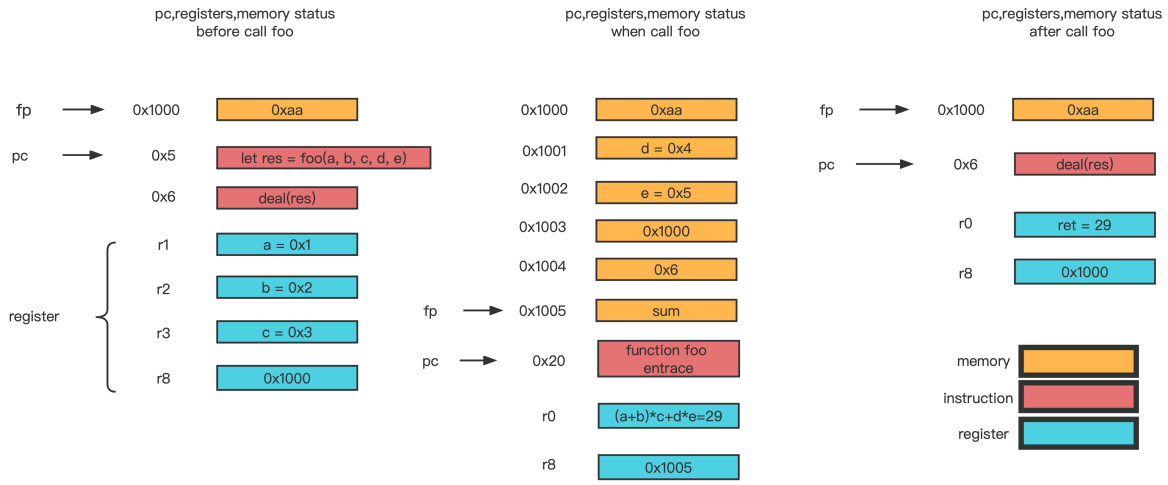


Figure 9: OlaVM function call model

2.2.2 Memory

The memory structure of OlaVM consists of two types (Read-Write and Write-Once) and mapping to disjoint regions of the memory space.

Memory Seg	*Type*	*Range*
random access memory	read-write	$[0, 2^{64} - 3 \cdot 2^{32} - 1]$
ecdsa	write-once	$[2^{64} - 3 \cdot 2^{32}, 2^{64} - 2 \cdot 2^{32} - 1]$
poseidon	write-once	$[2^{64} - 2 \cdot 2^{32}, 2^{64} - 2^{32} - 1]$
prophet	write-once	$[2^{64} - 2^{32}, 2^{64} - 1]$

Table 5: Memory segment range

OlaVM uses the B-Tree algorithm to manage the memory address and support random access memory.The B-Tree algorithm sorts the memory by address and CPU clock when inserting data into the memory by address.

1. When random access memory, the time complexity of search is $O(\log N)$.
2. When generate memory trace table by address and clock in ascending order,the memory can direct visit, not need sort again.

The memory structure illustration is as **Figure 10**:

2.2.3 Prophet Interpreter

The prophet interpreter runs the prophet code in the field `prophets` of program. Prophet interpreter does not change the state of PC and $\mathtt{r}_0 - \mathtt{r}_8$. It runs the prophet code and stores the state of context id in the prophet segment of memory and updates the PSP to free prophet memory in ascending order. The prophet code snippet is as below, the entry procedure is the beginning when the prophet interpreter runs, and the variable with the prefix of `cid` is the context id:

```
%{
```

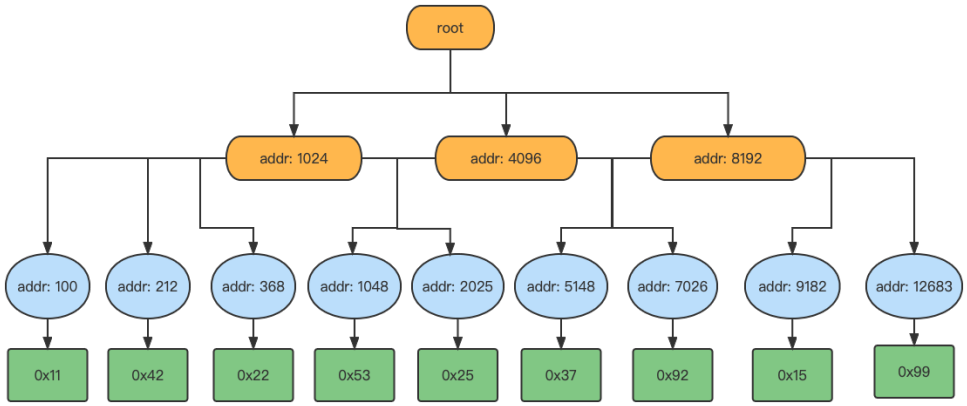


Figure 10: OlaVM memory structure

```
    function mod(felt x, felt y) -> felt {
        return x % y;
    }

    entry() {
        cid.r = mod(cid.x, cid.y);
    }
%}
```

The prophet interpreter illustration is as below. The pink blocks are the logic related to the prophet interpreter. The blue blocks are the logic related to OlaVM executor. The green blocks are the beginning and end. Prophet interpreter is embedded to OlaVM executor as a module. Olavm execute the program the process is shown as **Figure 11**:

2.2.4 builtin

OlaVM implements builtin to support range check, hash, etc. Builtin code need to be encoded in the instruction set, the definition in builtin rows of **Table 3**.

- **Range Check**

  Range check builtin is used to check u32 type data.OlaVM put check data into range check trace table when execting the range check builtin. The trace table reference: **Table 12**.

- **Bitwise**

  Bitwise builtin is used to implement the and, or, xor three bitwise operations of u32 type data.OlaVM runs the builtin logic and put operation result into bitwise trace table.the trace table reference: **Table 14**. Meanwhile, put two source operands into range check trace table.

- **Comparison**

  Comparison builtin is used to implement gte (great or equal) comparison.OlaVM runs the builtin logic and operation result into a comparison trace table. The trace table reference: **Table 13**. Meanwhile, put two source operands into range check trace table.

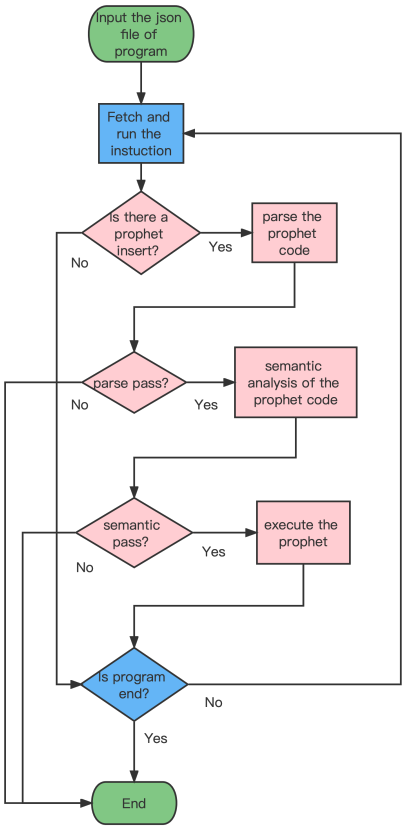


Figure 11: OlaVM prophet interpreter flow

2.3 Trace Tables

In general, the constraints of each table in OlaVM and the constraints between different tables are shown in the figure 12

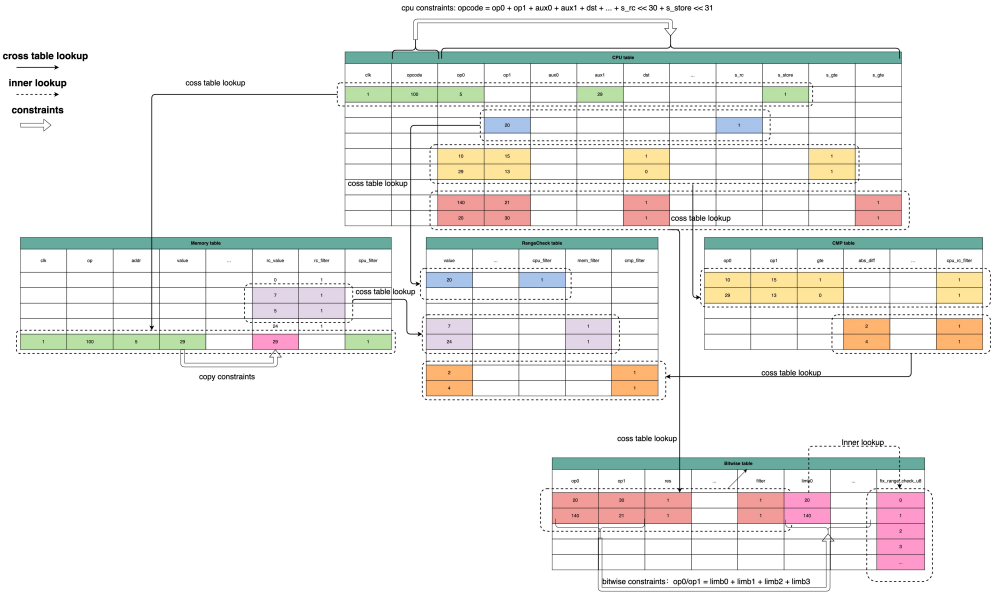


Figure 12: OlaVM STARK tables

2.3.1 Cpu Table

Context related columns:

clk	pc	reg0	reg1	reg2	reg3	reg4	reg5	reg6	reg7	reg8 (fp)

Table 6: Cpu table columns, context related

- clk: Cpu clock, 1 increment per line.
- pc: Program counter, indicates the line number of the currently executed instruction in the program.
- reg0 ~reg8: Register values. Reg8 may work as general register as well as frame pointer.

Instruction related columns:

instruction	op1_imm	opcode	immediate_value

Table 7: Cpu table columns, instruction related

- instruction: The currently executing instruction.
- op1_imm: Indicates if op1 is a immediate value.
- opcode: The currently executing instruction opcode.
- immediate_value: If current instruction contains immediate value, this should be it.

Columns for register selector:

- op0: Value of operand 0, should be a copy of a register.

op0	op1	dst	aux0	aux1	sel_op0_r0	···	sel_op0_r8	···	sel_op1_r8	sel_dst_r0	···	sel_dst_r8

Table 8: Cpu table columns for register selector

- op1: Value of operand 1, could be a copy of a register or equal to the immediate value.
- dst: Value of result, should be a copy of a register.
- aux0: Auxillary column, some instruction may use this column.
- aux1: Auxillary column, some instruction may use this column.
- sel_op0_ri: The i-th register used for op0.
- sel_op1_ri: The i-th register used for op1.
- sel_dst_ri: The i-th register used for result.

Columns for opcode selector:

sel_add	sel_mul	sel_eq	sel_assert	sel_mov	sel_jmp	sel_cjmp	sel_call	sel_ret	sel_mload	sel_mstore

Table 9: Cpu table columns for opcode selector

- sel_add: Selector for opcode add.
- sel_mul: Selector for opcode mul.
- sel_eq: Selector for opcode eq.
- sel_assert: Selector for opcode assert.
- sel_mov: Selector for opcode mov.
- sel_jmp: Selector for opcode jmp.
- sel_cjmp: Selector for opcode cjmp.
- sel_call: Selector for opcode call.
- sel_ret: Selector for opcode ret.
- sel_mload: Selector for opcode mload.
- sel_mstore: Selector for opcode mstore.

Columns for builtin selector:

sel_range_check	sel_and	sel_or	sel_xor	sel_not	sel_neq	sel_gte	sel_poseidon	sel_ecdsa

Table 10: Cpu table columns for builtin selector

- sel_range_check: Selector for range_check.
- sel_and: Selector for bitwise and.
- sel_or: Selector for bitwise or.
- sel_xor: Selector for bitwise xor.
- sel_not: Selector for bitwise not.
- sel_neq: Selector for comparison neq.
- sel_gte: Selector for comparison gte.
- sel_poseidon: Selector for poseidon hash function.
- sel_ecdsa: Selector for ecdsa.

2.3.2 Memory Table

is_rw	addr	clk	opcode	is_write	value	diff_addr	diff_ addr_inv	diff_clk	diff_add r_cond	filter_looked _for_main	rv_addr _unchanged	region _prophet	region _poseidon	region _ecdsa	rc_value	filter_ looking_rc
1	100	5	mstore	1	55	0	0	0	0	1	0	0	0	0	0	1
1	100	12	mload	0	55	0	0	7	0	1	1	0	0	0	7	1
1	100	17	mstore	1	300	0	0	5	0	1	1	0	0	0	5	1
1	100	26	mload	0	300	0	0	9	0	1	1	0	0	0	9	1
1	124	36	mstore	1	20000	24	(1/24)	0	0	1	0	0	0	0	24	1
1	124	37	call	0	20000	0	0	1	0	1	1	0	0	0	1	1
1	124	50	ret	0	20000	0	0	13	0	1	1	0	0	0	13	1
1	125	37	call	1	30000	1	1	0	0	1	0	0	0	0	1	1
1	125	50	ret	0	30000	0	0	13	0	1	1	0	0	0	13	1
1	150	20	mstore	1	78	25	(1/25)	0	0	1	0	0	0	0	25	1
1	150	23	mload	0	78	0	0	3	0	1	1	0	0	0	3	1
1	200	8	mstore	1	99	50	(1/50)	0	0	1	0	0	0	0	50	1
1	200	15	mload	0	99	0	0	7	0	1	1	0	0	0	7	1
0	p”’	0	0	1	in	p”’-200	0	0	p”-addr	0	0	0	0	1	p”-addr	1
0	p”’+1	0	0	1	in	1	0	0	p”-addr	0	0	0	0	1	p”-addr	1
0	p”’+2	0	0	1	out	1	0	0	p”-addr	0	0	0	0	1	p”-addr	1
0	p”’+2	44	mload	0	out	0	0	0	p”-addr	1	0	0	0	1	p”-addr	1
0	p”’+3	0	0	1	out	1	0	0	p”-addr	0	0	0	0	1	p”-addr	1
0	p”’+3	45	mload	0	out	0	0	0	p”-addr	1	0	0	0	1	p”-addr	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	p”	0	0	1	in	...	0	0	p’-addr	0	0	0	1	0	p’-addr	1
0	p”+1	0	0	1	in	1	0	0	p’-addr	0	0	0	1	0	p’-addr	1
0	p”+2	0	0	1	out	1	0	0	p’-addr	0	0	0	1	0	p’-addr	1
0	p”+2	55	mload	0	out	0	0	0	p’-addr	1	0	0	1	0	p’-addr	1
0	p”+3	0	0	1	out	1	0	0	p’-addr	0	0	0	1	0	p’-addr	1
0	p”+3	56	mload	0	out	0	0	0	p’-addr	1	0	0	1	0	p’-addr	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	p’	0	0	1	123	...	0	0	p-addr	0	0	1	0	0	p-addr	1
0	p’	7	mload	0	123	0	0	0	p-addr	1	0	1	0	0	p-addr	1
0	p’	9	mload	0	123	0	0	0	p-addr	1	0	1	0	0	p-addr	1
0	p’+1	0	0	1	456	1	0	0	p-addr	0	0	1	0	0	p-addr	1
0	p’+1	27	mload	0	456	0	0	0	p-addr	1	0	1	0	0	p-addr	1

Table 11: Memory Trace Table

- is_rw: Indicates whether the current memory is in read-write segment.
- addr: Address of memory.
- clk: Clock when access memory this time.
- opcode: The opcode that triggered this memory access.In write-once segment, no corresponding opcode for

write access, and should be 0 here.

- is_write: Indicates whether this memory access is write.
- value: Value in the memory cell.
- diff_addr: Address difference between two rows.
- diff_addr_inv: The reciprocal of the address difference between the two rows.Only used in read-write segment, in write-once segment, should be 0.
- diff_clk: Clock difference between two rows.Only used in read-write segment, in write-once segment, should be 0.
- diff_addr_cond:
  - Read-write segment: Should be 0 here.
  - Ecdsa segment: $\mathrm{p}'' - \mathrm{addr}$.
  - Poseidon segment: $\mathrm{p}' - \mathrm{addr}$.
  - Prophet segment: $\mathrm{p} - \mathrm{addr}$.
- filter_looked_for_main: Filter column for cross table lookup between cpu table and memory table, should be 0 when opcode is 0 and 1 otherwise.
- rw_addr_unchanged: Indicates current address is in read-write segment, and address is the same as the last row.
- region_prophet: Indicates current address is in prophet segment.
- region_poseidon: Indicates current address is in poseidon segment.
- region_ecdsa: Indicates current address is in ecdsa segment.
- rc_value: Value used for range check.
  - Read-write segment: Should be diff_clk(addr not changed) or diff_addr(addr changed).
  - Write-once segments: Should be diff_addr_cond.
- filter_looking_rc: Filter column for cross table lookup between memory table and range-check table, should all be 1.

2.3.3 Builtin Tables

filter_looked_for_cpu	filter_looked_for_memory	filter_looked_for_cmp	val	limb_lo	limb_high
1	0	0	0xffff0000	0x0	0xffff
0	1	0	0x1	0x0	0x1
0	0	1	0xe00a0	0xe	0xa0

Table 12: Range Check Table

2.3.3.1 Range check Table

- filter_looked_for_cpu: Filter column for cross table lookup between cpu table and range-check table.
- filter_looked_for_memory: Filter column for cross table lookup between memory table and range-check table.
- filter_looked_for_cmp: Filter column for cross table lookup between comparison table and range-check table.
- val: Value to be range checked.
- limb_lo: Low 16 bits of val.
- limb_high: Hight 16 bits of val.

In addition to these columns, there are also columns for fixed table lookup including: limb_lo_permuted, limb_hi_permuted, fix_rc_u16, fix_rc_permuted_lo, fix_rc_permuted_hi.

op0	op1	gte	abs_diff	abs_diff_inv	filter_looking_rc
5	3	1	2	(1/2)	1
2	7	0	5	(1/5)	1
6	6	1	0	0	1

Table 13: Comparison Table

2.3.3.2 Comparison Table

- op0: Comparison first operand.
- op1: Comparison second operand.
- gte: Indicates whether this row is constrained for gte or lt.
- abs_diff: Absolute value of the difference between two operands.
- abs_diff_inv: The reciprocal of abs_diff.
- fliter_looking_rc: Filter column for cross table lookup between comparison table and range-check table.

opcode	op0	op1	res	op0_0	op0_1	op0_2	op0_3	op1_0	op1_1	op1_2	op1_3	res_0	res_1	res_2	res_3
and	30	10	10	30	0	0	0	10	0	0	0	10	0	0	0
or	2	30	30	2	0	0	0	30	0	0	0	30	0	0	0
xor	10	3	9	10	0	0	0	3	0	0	0	9	0	0	0

Table 14: Bitwise Table

2.3.3.3 Bitwise Table

- opcode: Can be and/or/xor.
- op0: Bitwise first operand.
- op1: Bitwise second operand.
- res: Bitwise result.
- op0_i: i-th limb of op0.
- op1_i: i-th limb of op1.
- res_i: i-th limb of res.

In addition to these columns, there are also some columns used for fixed table lookup.

2.4 Constraints

2.4.1 Constraints for Cpu Table

Instruction encode related constraints

Related columns include instruction, opcode, op_imm, sel_opi_rj, and opcode/builtins selector columns.

op_imm should be binary:

$$\text{op\_imm}_i * (1 - \text{op\_imm}_i) = 0$$

Selectors should be binary:

$$\text{sel\_op0\_r0}_i \cdot (1 - \text{sel\_op0\_r0}_i) = 0$$

$$\text{sel\_op0\_r1}_i \cdot (1 - \text{sel\_op0\_r1}_i) = 0$$

$$\dots$$

$$\text{sel\_op0\_r8}_i \cdot (1 - \text{sel\_op0\_r8}_i) = 0$$

$$\text{sel\_op1\_r0}_i \cdot (1 - \text{sel\_op1\_r0}_i) = 0$$

$$\text{sel\_op1\_r1}_i \cdot (1 - \text{sel\_op1\_r1}_i) = 0$$

$$\text{sel\_op1\_r2}_i \cdot (1 - \text{sel\_op1\_r2}_i) = 0$$

$$\dots$$

$$\text{sel\_op1\_r8}_i \cdot (1 - \text{sel\_op1\_r8}_i) = 0$$

$$\text{sel\_dst\_r0}_i \cdot (1 - \text{sel\_dst\_r0}_i) = 0$$

$$\text{sel\_dst\_r1}_i \cdot (1 - \text{sel\_dst\_r1}_i) = 0$$

$$\dots$$

$$\text{sel\_dst\_r8}_i \cdot (1 - \text{sel\_dst\_r8}_i) = 0$$

$$\text{sel\_add}_i \cdot (1 - \text{sel\_add}_i) = 0$$

$$\text{sel\_mul}_i \cdot (1 - \text{sel\_mul}_i) = 0$$

$$\text{sel\_eq} \cdot (1 - \text{sel\_eq}_i) = 0$$

$$\text{sel\_assert}_i \cdot (1 - \text{sel\_assert}_i) = 0$$

$$\text{sel\_mov}_i \cdot (1 - \text{sel\_mov}_i) = 0$$

$$\text{sel\_jmp}_i \cdot (1 - \text{sel\_jmp}_i) = 0$$

$$\text{sel\_cjmp}_i \cdot (1 - \text{sel\_cjmp}_i) = 0$$

$$\text{sel\_call}_i \cdot (1 - \text{sel\_call}_i) = 0$$

$$\text{sel\_ret}_i \cdot (1 - \text{sel\_ret}_i) = 0$$

$$\text{sel\_mload} \cdot (1 - \text{sel\_mload}_i) = 0$$

$$\text{sel\_mstore}_i \cdot (1 - \text{sel\_mstore}_i) = 0$$

$$\text{sel\_end}_i \cdot (1 - \text{sel\_end}_i) = 0$$

$$\text{sel\_range\_check}_i \cdot (1 - \text{sel\_range\_check}_i) = 0$$

$$\text{sel\_and}_i \cdot (1 - \text{sel\_and}_i) = 0$$

$$\text{sel\_or}_i \cdot (1 - \text{sel\_or}_i) = 0$$

$$\text{sel\_xor}_i \cdot (1 - \text{sel\_xor}_i) = 0$$

$$\text{sel\_not}_i \cdot (1 - \text{sel\_not}_i) = 0$$

$$\text{sel\_neq}_i \cdot (1 - \text{sel\_neq}_i) = 0$$

$$\text{sel\_gte}_i \cdot (1 - \text{sel\_gte}_i) = 0$$

$$\text{sel\_poseidon}_i \cdot (1 - \text{sel\_poseidon}_i) = 0$$

$$\text{sel\_ecdsa}_i \cdot (1 - \text{sel\_ecdsa}_i) = 0$$

Opcode encode constraint:

$$\begin{aligned}\text{opcode}_i - (&\text{sel\_add}_i \cdot 2^{34} + \text{sel\_mul}_i \cdot 2^{33} + \text{sel\_eq} \cdot 2^{32} + \text{sel\_assert}_i \cdot 2^{31} + \text{sel\_mov}_i \cdot 2^{30} + \text{sel\_jmp}_i \cdot 2^{29} \\ &+ \text{sel\_cjmp}_i \cdot 2^{28} + \text{sel\_call}_i \cdot 2^{27} + \text{sel\_ret}_i \cdot 2^{26} + \text{sel\_mload} \cdot 2^{25} + \text{sel\_mstore}_i \cdot 2^{24} \\ &+ \text{sel\_end}_i \cdot 2^{23} + \text{sel\_range\_check}_i \cdot 2^{22} + \text{sel\_and}_i \cdot 2^{21} + \text{sel\_or}_i \cdot 2^{20} + \text{sel\_xor}_i \cdot 2^{19} \\ &+ \text{sel\_not}_i \cdot 2^{18} + \text{sel\_neq}_i \cdot 2^{17} + \text{sel\_gte}_i \cdot 2^{16} + \text{sel\_poseidon}_i \cdot 2^{15} + \text{sel\_ecdsa}_i \cdot 2^{14}) = 0\end{aligned}$$

Instruction encode constraint:

$$\begin{aligned}\text{instruction}_i - (&\text{op1\_imm}_i \cdot 2^{62} + \text{sel\_op0\_r8}_i \cdot 2^{61} + \text{sel\_op0\_r7}_i \cdot 2^{60} + \text{sel\_op0\_r6}_i \cdot 2^{59} + \text{sel\_op0\_r5}_i \cdot 2^{58} \\ &+ \text{sel\_op0\_r4}_i \cdot 2^{57} + \text{sel\_op0\_r3}_i \cdot 2^{56} + \text{sel\_op0\_r2}_i \cdot 2^{55} + \text{sel\_op0\_r1}_i \cdot 2^{54} + \text{sel\_op0\_r0}_i \cdot 2^{53} \\ &+ \text{sel\_op1\_r8}_i \cdot 2^{52} + \text{sel\_op1\_r7}_i \cdot 2^{51} + \text{sel\_op1\_r6}_i \cdot 2^{50} + \text{sel\_op1\_r5}_i \cdot 2^{49} + \text{sel\_op1\_r4}_i \cdot 2^{48} \\ &+ \text{sel\_op1\_r3}_i \cdot 2^{47} + \text{sel\_op1\_r2}_i \cdot 2^{46} + \text{sel\_op1\_r1}_i \cdot 2^{45} + \text{sel\_op1\_r0}_i \cdot 2^{44} + \text{sel\_dst\_r8}_i \cdot 2^{43} \\ &+ \text{sel\_dst\_r7}_i \cdot 2^{42} + \text{sel\_dst\_r6}_i \cdot 2^{41} + \text{sel\_dst\_r5}_i \cdot 2^{40} + \text{sel\_dst\_r4}_i \cdot 2^{39} + \text{sel\_dst\_r3}_i \cdot 2^{38} \\ &+ \text{sel\_dst\_r2}_i \cdot 2^{37} + \text{sel\_dst\_r1}_i \cdot 2^{36} + \text{sel\_dst\_r0}_i \cdot 2^{35} + \text{opcode}_i) = 0\end{aligned}$$

Operands and their selector constraints

Define the following virtual columns:

$$\widetilde{\text{enable\_op0}}_i = \sum_{x=0}^{8} \text{sel\_op0\_rx}_i$$

$$\widetilde{\text{enable\_op0}}_i = \sum_{x=0}^{8} \text{sel\_op0\_rx}_i$$

$$\widetilde{\text{enable\_op0}}_i = \sum_{x=0}^{8} \text{sel\_op0\_rx}_i$$

Register selector constraints:

$$\widetilde{\text{enable\_op0}}_i \cdot (1 - \widetilde{\text{enable\_op0}}_i) = 0$$

$$\widetilde{\text{enable\_op1}}_i \cdot (1 - \widetilde{\text{enable\_op1}}_i) = 0$$

$$\widetilde{\text{enable\_dst}}_i \cdot (1 - \widetilde{\text{enable\_dst}}_i) = 0$$

Copy constraints between and related register:

$$\widetilde{\text{enable\_op0}}_i \cdot (\text{op0}_i - \sum_{x=0}^{8} \text{sel\_op0\_rx}_i \cdot \text{regx}_i) = 0$$

$$\widetilde{\text{enable\_op1}}_i \cdot (\text{op1}_i - \sum_{x=0}^{8} \text{sel\_op1\_rx}_i \cdot \text{regx}_i) = 0$$

$$\widetilde{\text{enable\_dst}}_i \cdot (\text{dst}_i - \sum_{x=0}^{8} \text{sel\_dst\_rx}_i \cdot \text{regx}_{i+1}) = 0$$

If op1_imm is 1, then op1 equals the immediate value:

$$\text{op1\_imm}_i \cdot (\text{op1}_i - \text{immediate\_value}_i) = 0$$

Only one operation selector can be enabled at a time:

$$\begin{aligned} 1 - (&\text{sel\_add}_i + \text{sel\_mult}_i + \text{sel\_eq}_i + \text{sel\_assert}_i + \text{sel\_mov}_i + \text{sel\_jmp}_i + \text{sel\_cjmp}_i + \text{sel\_call}_i \\ &+ \text{sel\_ret}_i + \text{sel\_mload}_i + \text{sel\_mstore}_i + \text{sel\_end}_i + \text{sel\_range\_check}_i + \text{sel\_and}_i + \text{sel\_or}_i + \text{sel\_xor}_i \\ &+ \text{sel\_not}_i + \text{sel\_neq}_i + \text{sel\_gte}_i + \text{sel\_poseidon}_i + \text{sel\_ecdsa}_i) = 0 \end{aligned}$$

Context state transition constraints

clk:

$$(1 - \text{sel\_end}_i) \cdot (\text{clk}_{i+1} - (\text{clk}_i + 1)) = 0$$

Registers:

For common register, its value changes only when the register is used as dst:

$$(1 - \text{sel\_dst\_r0}_i) \cdot (\text{reg0}_{i+1} - \text{reg0}_i) = 0$$

$$(1 - \text{sel\_dst\_r1}_i) \cdot (\text{reg1}_{i+1} - \text{reg1}_i) = 0$$

$$(1 - \text{sel\_dst\_r2}_i) \cdot (\text{reg2}_{i+1} - \text{reg2}_i) = 0$$

$$(1 - \text{sel\_dst\_r3}_i) \cdot (\text{reg3}_{i+1} - \text{reg3}_i) = 0$$

$$(1 - \text{sel\_dst\_r4}_i) \cdot (\text{reg4}_{i+1} - \text{reg4}_i) = 0$$

$$(1 - \text{sel\_dst\_r5}_i) \cdot (\text{reg5}_{i+1} - \text{reg5}_i) = 0$$

$$(1 - \text{sel\_dst\_r6}_i) \cdot (\text{reg6}_{i+1} - \text{reg6}_i) = 0$$

$$(1 - \text{sel\_dst\_r7}_i) \cdot (\text{reg7}_{i+1} - \text{reg7}_i) = 0$$

Reg8 is usually used as fp:

$$\text{sel\_ret}_i \cdot (\text{reg8}_{i+1} - \text{aux0}_i) + (1 - \text{sel\_ret}_i) \cdot (1 - \text{sel\_dst\_r8}_i) \cdot (\text{reg8}_{i+1} - \text{reg8}_i) = 0$$

pc:

Opcodes related to pc state transitions include: jmp, cjmp, call, ret.

Define virtual column $\widetilde{\text{instruction\_size}}$ as instruction size which equals pc default increment:

$$\widetilde{\text{instruction\_size}}_i = (1 - \text{sel\_mload}_i - \text{sel\_mstore}_i) \cdot (1 + \text{op1\_imm}_i) + (\text{sel\_mload}_i + \text{sel\_mstore}_i) \cdot 2$$

When current opcode is jmp, the constraint fragment is:

$$\text{sel\_jmp}_i \cdot \text{op1}_i$$

When current opcode is cjmp, the constraint fragment is:

$$\text{sel\_cjmp}_i \cdot ((1 - \text{op0}_i) \cdot (\text{pc}_i + \widetilde{\text{instruction\_size}}_i) + \text{op0}_i \cdot \text{op1}_i)$$

And cjmp need op0 to be binary:

$$\text{sel\_cjmp}_i \cdot \text{op0}_i \cdot (1 - \text{op0}_i) = 0$$

When current opcode is call, related trace cells in cpu table are in **Table 15**:

op0	op1	dst	aux0	aux1
fp-1	target_pc	ret_pc	fp-2	ret_pc

Table 15: Trace of call

Op0 is the address of fp-1, dst is the value of fp-1 in memory.Similarly, aux0 is the address of fp-2, aux1 is the value of fp-2 in memory.This correspondence requires a cross table lookup using clk-opcode-op0-dst and clk-opcode-aux0-aux1 between cpu table and memory table.And the constraint fragment for call is:

$$\text{sel\_call}_i \cdot \text{op1}_i$$

Also we need to constraint ret_pc for call:

$$\text{ret\_pc}_i - pc_i - \widetilde{\text{instruction\_size}})_i = 0$$

When current opcode is ret, related trace cells in cpu table are in **Table 16**:

op0	op1	dst	aux0	aux1
fp-1	0	ret_pc	fp-2	ret_pc

Table 16: Trace of ret

We also need to add cross table lookup for ret, and constraint fragment in cpu table for ret is:

$$\text{sel\_ret}_i \cdot \text{op1}_i$$

Putting the above together, the constraint on the pc state transition is:

$$\begin{aligned}(1 - \text{sel\_end}_i) \cdot (\text{pc}_{i+1} - ((1 - (\text{sel\_jmp}_i + \text{sel\_cjmp}_i + \text{sel\_call}_i + \text{sel\_ret}_i)) \cdot \widetilde{\text{instruction\_size}}_i \\ + \text{sel\_jmp}_i \cdot \text{op1}_i + \text{sel\_cjmp}_i \cdot ((1 - \text{op0}_i) \cdot \widetilde{\text{instruction\_size}}_i + \text{op0}_i \cdot \text{op1}_i) + \text{sel\_call}_i \cdot \text{op1}_i \\ + \text{sel\_ret}_i \cdot \text{op1}_i)) = 0\end{aligned}$$

Constraints on specific opcode:

ADD:

$$\text{sel\_add}_i \cdot (\text{dst}_{i+1} - (\text{op0}_i + \text{op1}_i)) = 0$$

MUL:

$$\text{sel\_mul}_i \cdot (\text{dst}_{i+1} - (\text{op0}_i \cdot \text{op1}_i)) = 0$$

EQ/NEQ:

For opcode EQ/NEQ, related trace cells in cpu table are in **Table 17**:

opcode	dst	op0	op1	aux0
(eq)	1	5	5	0
(eq)	0	19	6	(1/(19-6))
(neq)	0	5	5	0
(neq)	1	19	6	(1/(19-6))

Table 17: Trace of EQ

Related constraints:

$$\text{sel\_eq}_i \cdot (\text{dst}_i \cdot (\text{op0}_i - \text{op1}_i) + (1 - \text{dst}_i) \cdot (1 - (\text{op0}_i - \text{op1}_i) \cdot \text{aux0}_i)) = 0$$

$$\text{sel\_neq}_i \cdot ((1 - \text{dst}_i) \cdot (\text{op0}_i - \text{op1}_i) + (1 - \text{dst}_i) \cdot (1 - (\text{op0}_i - \text{op1}_i) \cdot \text{aux0}_i)) = 0$$

ASSERT:

$$\text{sel\_assert}_i \cdot (\text{op1}_i - \text{op2}_i) = 0$$

MOV:

$$\text{sel\_mov}_i \cdot (\text{dst}_i - \text{op1}_i) = 0$$

CALL:

$$\text{sel\_call}_i \cdot ((\text{op0}_i - (\text{fp}_i - 1)) + (\text{op1\_imm}_i \cdot (\text{op1}_i - \text{pc}_i - 2) + (1 - \text{op1\_imm}_i) \cdot (\text{op1}_i - \text{pc}_i - 1))) = 0$$

RET:

$$\text{sel\_ret}_i \cdot ((\text{op0}_i + 1 - \text{fp}_i) + (\text{op1}_i - \text{pc}_i) + (\text{dst}_i + 2 - \text{fp}_i) + (\text{aux0}_i - \text{fp}_i)) = 0$$

MLOAD/MSTORE:

mstore and mload support relative addressing, related trace cells are in **Table 18**:

opcode	op0	op1	dst	aux0	aux1
(mload)	...	anchor_addr	value	offset	addr
(mstore)	value	anchor_addr	...	offset	addr

Table 18: Trace of EQ

When op1_imm = 1, aux0 need to be 0:

$$\text{sel\_mload}_i \cdot \text{op1\_imm}_i * \text{aux0}_i = 0$$

$$\text{sel\_mstore}_i \cdot \text{op1\_imm}_i * \text{aux0}_i = 0$$

When 0p1_imm = 0, aux0 should be the immediate value in the instruction:

$$\text{sel\_mload}_i \cdot (1 - \text{op1\_imm}_i) \cdot (\text{aux0}_i - \text{immediate\_value}_i) = 0$$

$$\text{sel\_mstore}_i \cdot (1 - \text{op1\_imm}_i) \cdot (\text{aux0}_i - \text{immediate\_value}_i) = 0$$

2.4.2 Constraints for Memory Table

Regional division of memory

In general, the memory table is divided into two areas, read-write and write-once, which are indicated in green and yellow respectively in **Table 11**. In the write-once area, it is subdivided into three small areas according to different uses: prophet area, poseidon area, and ecdsa area, and the available size of each small area is: $span = 2^{32} - 1$.Define $p = 2^{64} - 2^{32} + 1$, $p' = p - span$, $p'' = p' - span$, $p''' = p'' - span$.The address upper limit for each of these three small areas are $p - 1$, $p' - 1$, $p'' - 1$, The starting addresses of these three small areas are p', p'', p'''. Each small area has a corresponding flag column, the correctness of the flag column is ensured by a range check of the difference between the upper limit and address.Finally, the read-write area is in a four-choice relationship with the three small areas mentioned above, and the flag of the read-write area can be constrained by this relationship.

Constraints

opcode:

$$\text{opcode}_i \cdot (\text{opcode}_i - \mathit{op\_mload}) \cdot (\text{opcode}_i - \mathit{op\_mstore}) \cdot (\text{opcode}_i - \mathit{op\_call}) \cdot (\text{opcode}_i - \mathit{op\_ret}) = 0$$

When opcode is 0, is_rw must be 0:

$$(\text{opcode}_i - op\_mload) \cdot (\text{opcode}_i - op\_mstore) \cdot (\text{opcode}_i - op\_call) \cdot (\text{opcode}_i - op\_ret) \cdot \text{is\_rw}_i = 0$$

The relationship between is_write and opcode:

For the write case, the opcode may be mstore, call, prophet write; For read, opcode may be mload, call and ret; When opcode is call, it may be read or write, so we need not constraint is_write for call:

$$(\text{opcode}_i - op\_mload) \cdot (\text{opcode}_i - op\_ret) \cdot (\text{opcode}_i - \text{op\_call}) \cdot (1 - \text{is\_write}_i) = 0$$

$$\text{opcode}_i \cdot (\text{opcode}_i - op\_mstore) \cdot (\text{opcode}_i - \text{op\_call}) \cdot \text{is\_write}_i = 0$$

filter_looked_for_main:

$$\text{opcode}_i \cdot (1 - \text{filter\_lookup\_for\_main}_i) = 0$$

Constraints for diff_addr:

diff_addr is mainly used in the rw segment to perform a range check on the difference of addresses to ensure the correctness of the incremental sorting of addresses.The corresponding auxiliary column diff_addr_inv is the reciprocal of diff_addr (or 0) and is used to limit diff from unpredictable values to 0/1 to constrain it with rw_addr_changed.

The diff_addr constraint will span the boundary between the two regions, except that the diff_addr after the span does not need to be range checked.

$$\text{addr}_{i+1} - \text{addr}_i - \text{diff\_addr}_{i+1} = 0$$

For rw_addr_changed:

$$\text{rw\_addr\_unchanged}_0 = 0$$

$$\text{is\_rw}_i \cdot (1 - \text{rw\_addr\_changed}_i - \text{diff\_addr}_i \cdot \text{diff\_addr\_inv}_i) = 0$$

For region select:

$$\text{is\_rw}_i + \text{region\_prophet}_i + \text{region\_poseidon}_i + \text{region\_ecdsa}_i - 1 = 0$$

Different regions of diff_addr_cond have different meanings:

$$\text{region\_prophet}_i \cdot (p - \text{addr}_i - \text{diff\_addr\_cond}_i) = 0$$

$$\text{region\_poseidon}_i \cdot (p' - \text{addr}_i - \text{diff\_addr\_cond}_i) = 0$$

$$\text{region\_ecdsa}_i \cdot (p'' - \text{addr}_i - \text{diff\_addr\_cond}_i) = 0$$

Use region_prophet, region_poseidon, and region_ecdsa as filters to perform range check on diff_addr_cond.

Binary constraints:

$$\text{is\_rw}_i \cdot (1 - \text{is\_rw}_i) = 0$$

$$\text{region\_prophet}_i \cdot (1 - \text{region\_prophet}_i) = 0$$

$$\text{region\_poseidon}_i \cdot (1 - \text{region\_poseidon}_i) = 0$$

$$\text{region\_ecdsa}_i \cdot (1 - \text{region\_ecdsa}_i) = 0$$

Constraints for write-once region: In the write-once region, the address is incremented by 1 or remains unchanged:

$$(1 - \text{is\_rw}_i) \cdot (1 - \text{region\_prophet}_{i+1} + \text{region\_prophet}_i) \cdot (1 - \text{region\_poseidon}_{i+1} + \text{region\_poseidon}_i) \cdot (\text{addr}_{i+1} - \text{addr}_i) \cdot (\text{addr}_{i+1} - \text{addr}_i - 1) = 0$$

Write-once constraint, the opcode must be mload if the address is not incremented by 1:

$$(1 - \text{is\_rw}_i) \cdot (1 - \text{region\_prophet}_{i+1} + \text{region\_prophet}_i) \cdot (1 - \text{region\_poseidon}_{i+1} + \text{region\_poseidon}_i) \cdot (\text{addr}_{i+1} - \text{addr}_i - 1) \cdot \text{is\_write}_{i+1} = 0$$

Constraints for read or write: First opcode for each address must be write(mstore or 0) and the next row must be mload and value remains unchanged:

$$\text{is\_write}_0 - 1 = 0$$

$$(\text{addr}_{i+1} - \text{addr}_i) \cdot (1 - \text{is\_write}_{i+1}) = 0$$

$$(1 - \text{is\_write}_{i+1}) \cdot (\text{value}_{i+1} - \text{value}_i) = 0$$

Constraints for rc_value: The value of rc_value in the line where is_rw is 1 is the non-zero value of diff_addr and diff_clk (0 if both are 0); rc_value equals diff_addr_cond when is_rw is 0.

$$\text{is\_rw}_i \cdot (\text{diff\_addr}_i + \text{diff\_clk}_i - \text{rc\_value}_i) = 0$$

$$(1 - \text{is\_rw}_i) \cdot (\text{diff\_addr\_cond}_i - \text{rc\_value}_i) = 0$$

For filter_looking_rc, it must be 1 in read-write region and must be 1 for read in write-once region.

$$\text{is\_rw}_i \cdot (1 - \text{filter\_looking\_rc}_i) = 0$$

$$(1 - \text{is\_rw}_i) \cdot (1 - \text{is\_write}_i) \cdot (1 - \text{filter\_looking\_rc}_i) = 0$$

2.4.3 Constraints for Builtins

Range check table constraints: Constraint val is the combination of limb_lo and limb_hi:

$$\text{val}_i - \text{limb\_lo} - \text{limb\_hi} \cdot 2^{16}$$

In addition, limb_lo and limb_hi need to perform lookup with fixed table which contains 0 to $2^{16} - 1$.

Comparison table constraints: gte is binary:

$$\text{gte}_i \cdot (1 - \text{gte}_i) = 0$$

Constraints for abs_diff:

$$\text{gte}_i \cdot (\text{op0}_i - \text{op1}_i - \text{abs\_diff}_i) = 0$$

$$(1 - \text{gte}_i) \cdot (\text{op1}_i - \text{op0}_i - \text{abs\_diff}_i) = 0$$

Constrains for abs_diff_inv:

$$(1 - \text{gte}_i) \cdot (1 - \text{abs\_diff}_i \cdot \text{abs\_diff\_inv}_i) = 0$$

Lookups:

- op0, op1, dst in CPU table need to perform cross table lookup with op0, op1, gte in comparison table.
- abs_diff column needs to perform cross table lookup with value column in range check table.

Bitwise table constraints:

$$\text{op0}_i - \text{op0\_0}_i - \text{op0\_1}_i \cdot 2^8 - \text{op0\_2}_i \cdot 2^{16} - \text{op0\_3}_i \cdot 2^{24} = 0$$

$$\text{op1}_i - \text{op1\_0}_i - \text{op1\_1}_i \cdot 2^8 - \text{op1\_2}_i \cdot 2^{16} - \text{op1\_3}_i \cdot 2^{24} = 0$$

$$\text{dst}_i - \text{dst\_0}_i - \text{dst\_1}_i \cdot 2^8 - \text{dst\_2}_i \cdot 2^{16} - \text{dst\_3}_i \cdot 2^{24} = 0$$

Lookups:

- opcode, op0, op1, dst in cpu table need to perform cross table lookup with opcode, op0, op1, res in bitwise table.
- Bitwise computation needs to lookup with fixed table.

3 Ola-Lang: Dev-friendly Smart Contract Language

Ola-lang is a high-level programming language for developing OlaVM smart contracts. It is Turing complete and can be used to write arithmetic programs.

The computing process is proven by the OlaVM back-end proof system, which verifies that the OlaVM processing is accurate.

Most of the existing programming languages in the ZKP field require fundamental knowledge of the circuit field, which is not universal, or the execution process is difficult to proven and verified by ZKP.

3.1 Language Introduction

From a developer's point of view, Ola's syntax is close to that of C/C++ and Rust, which allows most developers to learn and develop various Dapps using Ola quickly. At the same time, the Ola language is also a ZK-friendly language, but we do not expose the ZK-friendly design directly to the developers. We try to make the Ola language as smooth to use as the traditional high-level languages such as Rust, leaving the ZK-friendly design support to be handled by the compiler and OlaVM.

Ola's primary design goal:

- Security: The results of the program executions are deterministic, declared at the language level/syntax, and guaranteed at the compiler level.
- Efficiency: The speed of execution and proof of the program is well balanced to be efficient.
- Universal: Easy to use and readable code, it is developer friendly with minimal learning thresholds for developers familiar with mainstream programming languages such as C++/Rust, to quickly get into writing arithmetic programs.
- Turing complete: Ola can be used for complex programs such as loops, recursion, etc.

3.2 Language Elements

3.2.1 Variables

Identifier

Identifier consist of numbers (`0-9`), ASCII uppercase and lowercase letters (`a-zA-Z`), underscores (`_`). Identifier cannot start with a number.

```
fn foo() {
    // declare and ine `_variable`
    u32 _variable = 2;   // identifiers start with "_"
    // u32 0a = 2;  define error, identifiers can't start with number
}
```

Declaration

Variables need to be declared before used. To avoid variables being undefined, it needs to be initialized at declaration time.

```
fn foo() {
    // declare and define `a`
    u32 a = 2;
    // redefine `a`
    a = 3;
}
```

Scope

For security reasons, variable definitions do not support Shadowing. If you need multiple adjacent variables with similar logical meanings, use a variable or type suffix.

```
fn foo() {
    u32 a = 5;
    {
        u32 a = 25; // compile error: redeclared variable 'a'
    };
    u32 a = 25; // compile error: redeclared variable 'a'

    a = 25; // ok
}
```

Variables have a scope that is restricted to the current function, whereas constants do not.

Global variables are not supported.

```
fn foo() -> (u32) {
    // return a; <- not allowed
    return 2;
}

fn bar() -> (u32) {
    u32 a = 2;
    return foo();
}
```

Variables in a `for`-loop are scoped only inside the loop.

```
fn foo() -> (u32) {
    u32 a = 0;
    for (u32 i = 0; i < 5; i++) {
        a = a + i;
    }
    // return i; <- not allowed
    return a;
}
```

3.2.2 Data Type

Ola is a statically typed language, and variable types must be known at compile time to avoid most runtime exceptions. Two basic types and multiple complex types are supported.

Basic Types

The basic types include integer and boolean types.

There are several types of integer types: `u32`, `u64`, and `u256`, and currently only u32 operations are supported now. All types are built based on the `field` type. Ola provides the above-mentioned basic libs of various integer types based on the field implementation, which is convenient for developers to write complex logic.

```
u32 a = 2; // u32
u64 b = 2; // u64
u64 b = 0xffffl; // u64
u256 d = 10241111l; // u256
```

```
bool a = true;
bool b = false;
```

Arrays

Ola supports statically typed arrays. The data types of array elements must be consistent, and the array size must be determined at compile time.

Array elements are numbered from zero and are accessed using `[index]` for addressing.

Array declarations must be initialized, and the array declaration format is `type` and `[] (type [])`, and the array size must be specified. Two ways to initialize arrays are provided:

- Split the list of elements by commas, `[array_element1,array_element2,...]`.
- Array declaration and initialization with consistent array elements, `[array_value; size]`.

```
u32[5] a = [1, 2, 3, 4, 5];
bool[3] b = [true; 3]; // initialize a bool array with value true
```

Two-dimensional Arrays

Two-dimensional arrays are declared and used similarly to one-dimensional arrays, except that the internal elements of a two-dimensional array are also one-dimensional arrays.

Declare `type [row_size][col_size]`, and initialize `[[],[],...]`.

```
// Array of two elements of an array of 3 elements

u32[2][4] a = [[1, 2, 3, 4],[4, 5, 6, 7]];

u32[4] b = a[1]; // should be [4, 5, 6, 7]
```

Array Slicing

Similar to Rust, arrays can be created by slicing an array to copy the generated array,

```
u32[5] a = [1, 2, 3, 4, 5];
u32[2] b = a[2..4];   // initialize an array copying a slice from `a`
// array b is [3, 4, 5]
```

Tuples

A collection composed of multiple types of elements, each element can be accessed through `.` (for example, `t.0`, `t.1`, etc.).

```
fn main() -> (bool) {
    (u32[2], bool) v = ([1, 2], true);
    v.0 = [2, 3];
    return v.1;
}
```

Structs

A combination of multiple data types forms a new custom combination type. Struct members are accessed via `.` (`struct_name.struct_field`)

```
struct Person {
    age: u32,
    id: u64,
}

fn foo() {
    Person person = Person {
        age: 18,
        id: 123456789,
    };
    person.age = 25;
}
```

Enumerations

The enumeration type is defined by the keyword `enum`.

```
contract Foo {
    u256 const x = 56;
    enum ActionChoices {
        GoLeft,
        GoRight,
        GoStraight,
        Sit
    }
    ActionChoices const choices = ActionChoices.GoLeft;
}
```

Type Alias

To increase code readability, defining a type alias for each type is supported. At compile time, the type alias will be replaced with real types.

```
type balance = u256;

fn main() -> (balance) {
    balance a = 32;
    a -= 2;
    return a;
}
```

3.2.3 Constant

Constants can only be declared as constant expressions when defined with the `const` keyword.

Compile time determination cannot be redeclared and assigned, that is, once defined, it can only be used within its scope, and it is recommended to declare with all capital letters and _ concatenation.

```
const u32 ONE = 1;
const u32 TWO = ONE + ONE;

const u32 HASH_SIZE = 256;

fn hash_size() -> (u32) {
    return HASH_SIZE;
}
```

3.2.4 Operators

Ola Provides operators such as arithmetic, boolean, relational, bitwise, and so on.

Operator	Example	Explanation
`+`	`a + b`	Integer addition
`-`	`a - b`	Integer subtraction
`*`	`a * b`	Integer multiplication
`/`	`a / b`	Integer division
`%`	`a % b`	The modulo of integer division
`**`	`a ** b`	Power

Table 19: Arithmetic operators

Operator	Example	Explanation
`&&`	`a && b`	Boolean operator AND
`||`	`a || b`	Boolean operator OR
`!`	`!a`	Boolean operator NEGATION

Table 20: Boolean operators

Arithmetic operators

Arithmetic operators can be combined with the assignment operator `=` to form new compound operators `+=`, `-=`, `*=`, `/=`, `%=`, with arithmetic operators having higher priority than compound operators.

Boolean operators

Support with OR (`||`), NOT (`!`), AND (`&&`) with the latter having higher priority.

Relational operators

The return result of the relational operator is type `bool`

Bitwise operators

Bitwise operators containing bitwise or and non and shift operations.

Bitwise operators can be combined with the assignment operator `=` to form the new compound operators `&=`, `|=`, `^=`, `<<=`, `>>=`, with bitwise operators taking precedence over compound operators.

Operator	Example	Explanation
`==`	`a == b`	equal
`!=`	`a != b`	not equal
`<`	`a < b`	less than
`>`	`a > b`	greater than
`<=`	`a <= b`	less than or equal to
`>=`	`a >= b`	greater than or equal to

Table 21: Relational operators

Operator	Example	Explanation
&	a & b	Bitwise and
|	a | b	Bitwise or
^	a ^ b	Bitwise xor
<<	a << 3	left shift
>>	a >> 3	right shift
~	~a	Bitwise complement

Table 22: Bitwise operators

3.2.5 Control Flow

Conditional statement

Control conditional branch and select different branch programs to execute according to different conditions. If the expression value is nonzero, the branch body is executed. It comes in two forms:

- Contains only single branch `if`: `if conditional_expression {statements}`.
- Contains multiple branches of `if` and `else`: `if conditional_expression {statements} else {statements}`.

```
fn foo(u32 a) -> u32 {

    // Similar to rust, the result of a conditional expression
    // can be received directly by the variable
    u32 b = if (a + 1 == 2) { 1 } else { 3 };
    return b;
}
```

Note: Conditional statements support ternary conditional operators.

```
fn foo(u32 a) -> u32 {
    u32 b = a + 1 == 2 ? 1 : 3;
    return b;
}
```

Loop statement

Repeats the statement within the loop for a specified number of times based on the loop condition.

`for`-loop statement is supported. Its syntax is

`for (init_expression; conditional_expression; loop_expression) {statements}`.

The execution process is:

1. Calculate the `init_expression`, namely the loop initialization.
2. Calculate the `conditional_expression`. If the result is `true`, the loop body `statements` is executed, followed by the `loop_expression`.
3. If the result is `false`, `for`-loop statement terminates. Sequential execution starts with the next `statement`.

```
fn foo() -> u32 {
```

```
    u32 res = 0;
    for (u32 i = 0; i <= 10; i++) {
        res = res + i;
    }
    return res;
}
```

3.2.6 Functions

It is the basic module unit of Ola, containing declarations and statements.

If the `fn` keyword is used, the function name must be explicitly provided. parameters, and return values are optional, and parameters are passed by value.

The function return type must be specified after `->`.

- When a function call occurs, control permissions is passed from the calling function to the called function, and the parameters are passed to the called function by value.
- The called function return to the calling function through the `return` statement, and returns the return value to the calling function.

The basic syntax is:

```
fn function_name(parameter_declaration_list) -> return_parameter_list {
    // compound-statement
    statements
    return_statement
}
```

e.g.:

```
fn foo() -> u32 {
    return sum(1u, 2u);
}

fn sum(u32 a, u32 b) -> u32 {
/*
 *  Unlike rust, the return value of
 *  a function must be a combination of return and return value
 */
    return a + b;
}
```

3.2.7 Imports

To use the code from other files, we can import them into our program using the keyword `import` and `as` with the corresponding file name. Using `import` makes it easier for us to import some modular libs, eliminating the need for repeated development. The basic syntax is as follows, `path-spec` can be an absolute path (the full path of the source file) or relative path (file path starts with `./` or `../`).

```
import "path-spec";
```

```
import "path-spec" as alias_name;
```

e.g.:

```
import "./math/uint256.ola";
import "crypto/sha256.ola" as sha256;
```

3.2.8 Comment Lines

They are in-code documentation. When comments are inserted into the code, the compiler simply ignores them. Comment lines only serve as an aid in understanding the code.

Single-line comments start with `//` and multi-line subsubsubsection comments start with `/*` and end with `*/`.

Single line using `//`:

```
// Using this, we can comment on a line.
fn main(u32 a) -> u32 {
    u32 b = a + 1 == 2 ? 1 : 3;
    return b;
}
```

Multi-line subsubsubsection comments using `/*` and `*/`:

```
fn sum(u32 a, u32 b) -> u32 {
/*
 *  Unlike rust, the return value of
 *  a function must be a combination of return and return value
 */
    return a + b;
}
```

3.2.9 Keywords and Reservation Words

Keywords

The following **Table 23** shows the keywords and reserved words for ola-lang.

Reservation keywords

```
pub
impl
while
do
loop
use
match
static
u128
in
```

Keywords	Explanation
const	Constants declaration
type	Type alias declaration
struct	Structure declaration
enum	Definition of a enum
fn	Function declaration
for	Conditional loop based on the result of the expression
if	Result selection branches based on conditional expressions
else	Candidate statements for 'if' control flow
return	Function returns results
bool	Boolean value
u32	uint32 value
u64	uint64 value
u256	uint256 value
true	boolean true
false	boolean false
assert	Assertion of the input expression
import	Importing other files
contract	Definition of a smart contract

Table 23: Ola language keywords

3.3 Grammar

Ola's grammar is defined using the EBNF format file. The content of this EBNF file includes contract definitions, import instructions, type definitions, variable declarations, structure definitions, enumeration definitions, function definitions, and so on.

Rule SourceUnit

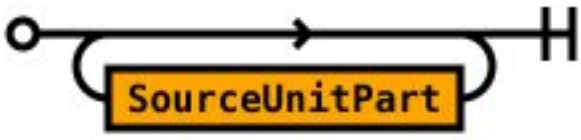


Figure 13: SourceUnit

```
rule SourceUnit ::=
    SourceUnitPart *
  ;
```

The SourceUnit rule represents the top-level structure of a smart contract source file. It consists of a sequence of source unit parts.

Rule SourceUnitPart

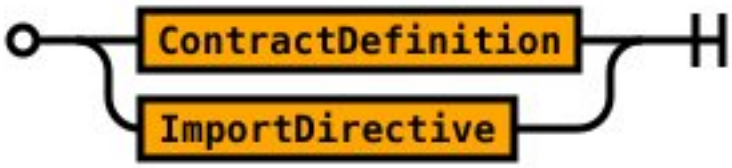


Figure 14: SourceUnitPart

```
rule SourceUnitPart ::=
    ContractDefinition
  | ImportDirective
  ;
```

The SourceUnitPart rule describes the elements that can appear at the top level of a source file. These elements can be either contract definitions or import directives.

Rule ImportDirective

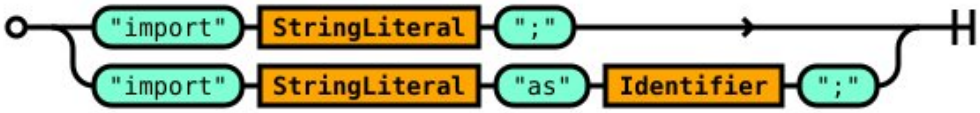


Figure 15: ImportDirective

```
rule ImportDirective ::=
     'import' StringLiteral  ';'
  |  'import' StringLiteral  'as' Identifier  ';'
  ;
```

The ImportDirective rule defines the syntax for importing other source files into the current file. There are two forms of import directives: a simple import and an import with an alias.

Rule Type

```
rule Type ::=
     'bool'
  |  'u32'
  |  'u64'
  |  'u256'
  ;
```

The Type rule defines the basic types available in the language. These include boolean values (bool), 32-bit unsigned integers (u32), 64-bit unsigned integers (u64), and 256-bit unsigned integers (u256).

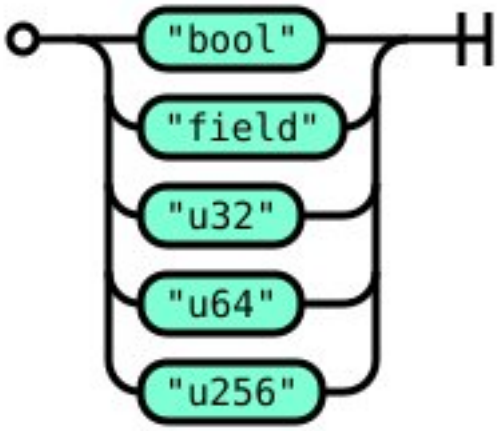


Figure 16: Type

Rule IdentifierOrError

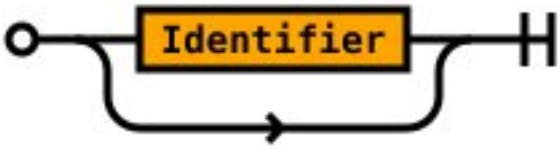


Figure 17: IdentifierOrError

```
rule IdentifierOrError ::=
    Identifier
  |
  ;
```

The IdentifierOrError rule represents either an identifier or an empty element, which can be useful for optional parts of the grammar.

Rule VariableDeclaration

Figure 18: VariableDeclaration

```
rule VariableDeclaration ::=
    Precedence0 IdentifierOrError
  ;
```

The VariableDeclaration rule defines the syntax for declaring a variable. A variable is declared by specifying a type, followed by an optional identifier.

Rule StructDefinition

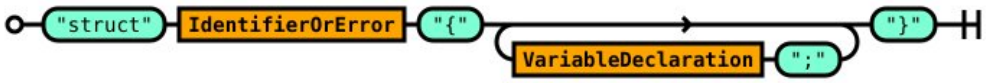


Figure 19: StructDefinition

```
rule StructDefinition ::=
     'struct' IdentifierOrError  '{' ( VariableDeclaration  ';' ) *   '}'
  ;
```

The StructDefinition rule defines the syntax for declaring a struct type. A struct is a composite data type that groups together variables under a single name.

Rule ContractPart

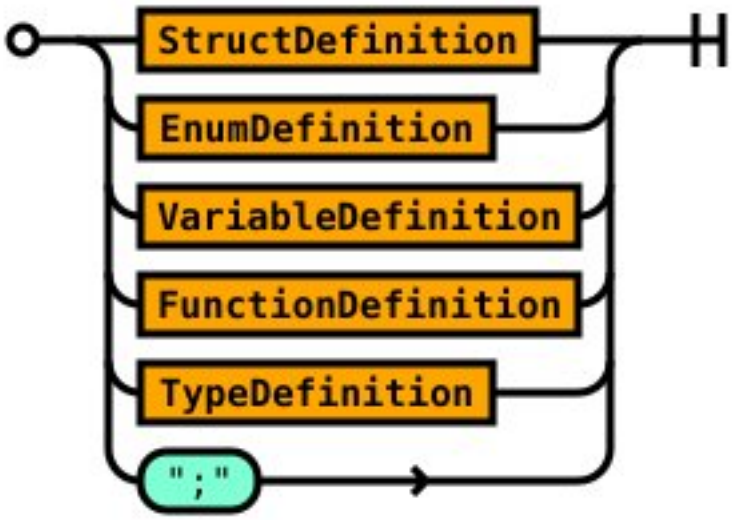


Figure 20: ContractPart

```
rule ContractPart ::=
    StructDefinition
  | EnumDefinition
  | VariableDefinition
  | FunctionDefinition
  | TypeDefinition
  |  ';'
  ;
```

The ContractPart rule defines the elements that can appear within a contract definition. These elements include struct definitions, enum definitions, variable definitions, function definitions, and type definitions.

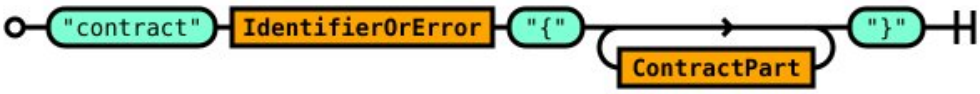


Figure 21: ContractDefinition

Rule ContractDefinition

```
rule ContractDefinition ::=
     'contract' IdentifierOrError  '{' ( ContractPart ) *   '}'
  ;
```

The ContractDefinition rule defines the syntax for declaring a smart contract. A contract is a collection of data and functions that can be executed on a blockchain.

Rule EnumDefinition

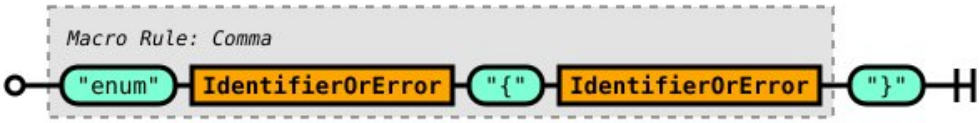


Figure 22: EnumDefinition

```
rule EnumDefinition ::=
     'enum' IdentifierOrError  '{' Comma!(IdentifierOrError)  '}'
  ;
```

The EnumDefinition rule defines the syntax for declaring an enumeration type. An enumeration is a data type consisting of a set of named values.

Rule VariableDefinition

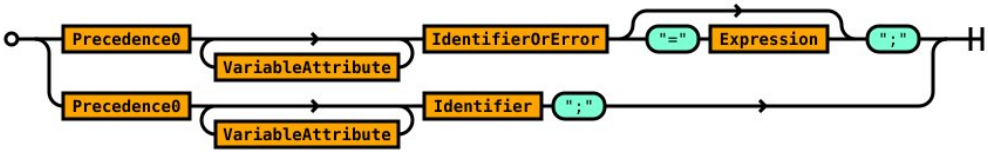


Figure 23: VariableDefinition

```
rule VariableDefinition ::=
    Precedence0 VariableAttribute *  IdentifierOrError (  '=' Expression ) ?   ';'
  | Precedence0 VariableAttribute *  Identifier  ';'
  ;
```

The VariableDefinition rule defines the syntax for defining a variable within a contract. Variables can have attributes such as "const" or "mut" to indicate whether they are constant or mutable.

Rule TypeDefinition

Figure 24: TypeDefinition

```
rule TypeDefinition ::=
     'type' Identifier  '=' Precedence0  ';'
  ;
```

The TypeDefinition rule defines the syntax for creating a new type alias. A type alias is a way to give a new name to an existing type, which can be useful for improving code readability and maintainability.

Rule VariableAttribute

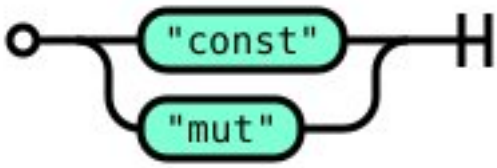


Figure 25: VariableAttribute

```
rule VariableAttribute ::=
'const'
| 'mut'
;
```

The VariableAttribute rule defines the attributes that can be applied to a variable. There are two possible attributes: `const` and `mut`. The `const` attribute indicates that the variable's value cannot be changed after it is initialized, while the `mut` attribute indicates that the variable's value can be modified.

Rule Expression

Figure 26: Expression

```
rule Expression ::=
Precedence14
;
```

The Expression rule represents any valid expression in the language. In this case, it refers to the highest level of precedence, Precedence14.

Rule Precedence14

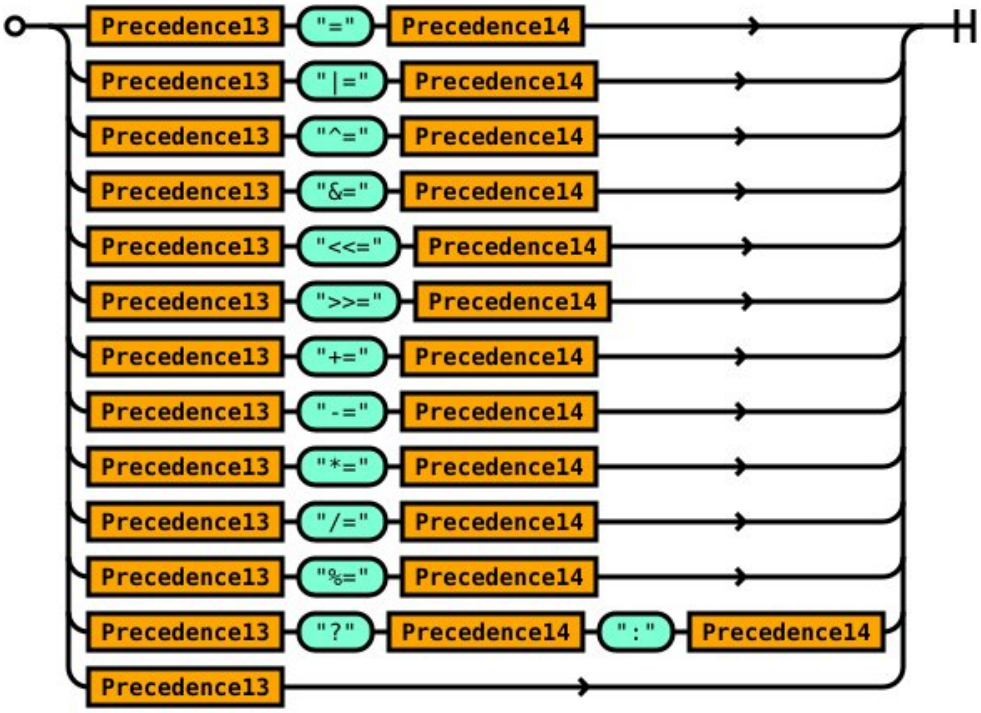


Figure 27: Precedence14

```
rule Precedence14 ::=
Precedence13 '=' Precedence14
| Precedence13 '|=' Precedence14
| Precedence13 '^=' Precedence14
| Precedence13 '&=' Precedence14
| Precedence13 '<<=' Precedence14
| Precedence13 '>>=' Precedence14
| Precedence13 '+=' Precedence14
| Precedence13 '-=' Precedence14
| Precedence13 '*=' Precedence14
| Precedence13 '/=' Precedence14
| Precedence13 '%=' Precedence14
| Precedence13 '?' Precedence14 ':' Precedence14
| Precedence13
;
```

The Precedence14 rule defines the various assignment and conditional operators with the highest precedence level. This includes assignment, bitwise OR assignment, bitwise XOR assignment, bitwise AND assignment, left shift assignment, right shift assignment, addition assignment, subtraction assignment, multiplication assignment, division assignment, modulo assignment, and the ternary conditional operator.

Rule Precedence13

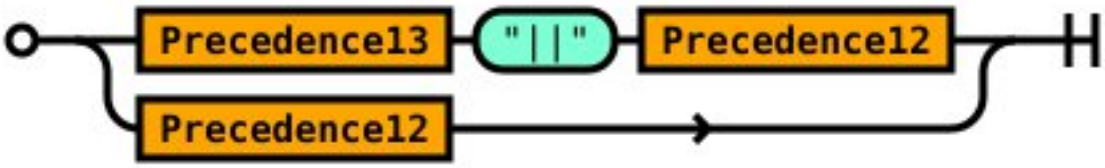


Figure 28: Precedence13

```
rule Precedence13 ::=
Precedence13 '||' Precedence12
| Precedence12
;
```

The Precedence13 rule represents the logical OR operator, which has lower precedence than assignment and conditional operators. It allows combining multiple conditions using the "||" symbol.

Rule Precedence12

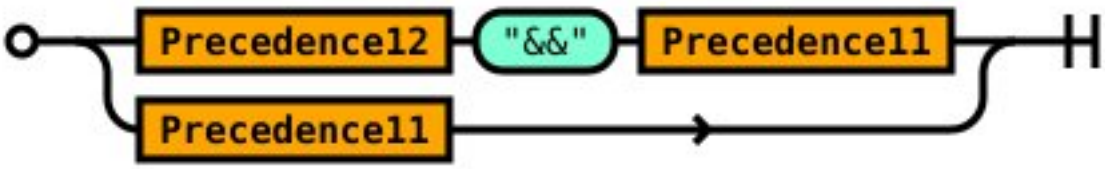


Figure 29: Precedence12

```
rule Precedence12 ::=
Precedence12 '&&' Precedence11
| Precedence11
;
```

The Precedence12 rule defines the logical AND operator, which has a lower precedence than the logical OR operator. It is used to combine multiple conditions using the "&&" symbol.

Rule Precedence11

```
rule Precedence11 ::=
    Precedence11  '==' Precedence10
  | Precedence11  '!=' Precedence10
  | Precedence10
  ;
```

The Precedence11 rule defines a non-terminal that represents operators with the same precedence level. It has two productions: the first production specifies the equality operator ('==') and the next higher precedence non-terminal Precedence10, while the second production specifies the inequality operator ('!=') and Precedence10. If neither of

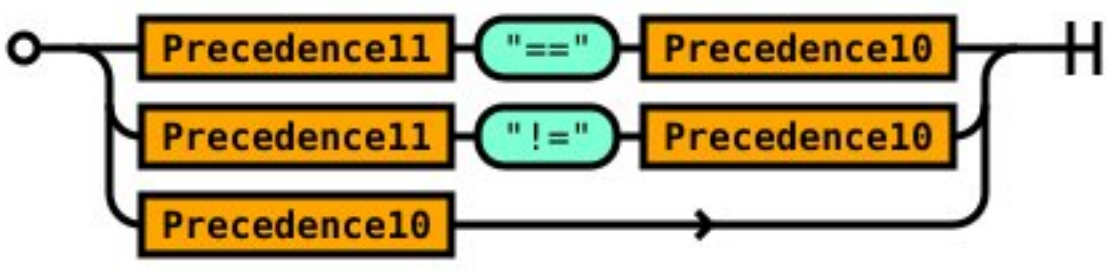


Figure 30: Precedence11

these cases applies, Precedence10 is directly considered as the result of this rule. This rule is a useful part of writing a parser or compiler for a programming language, as it can be used to specify operator precedence and associativity.

Rule Precedence10

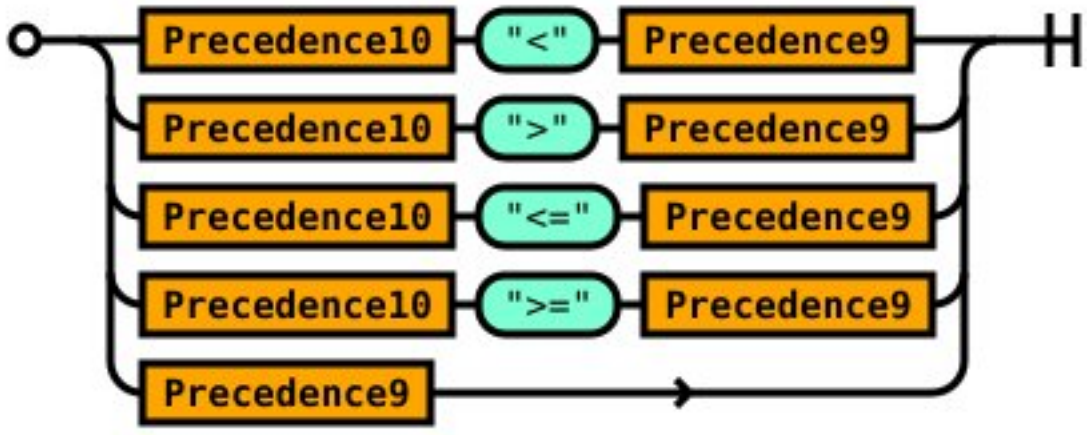


Figure 31: Precedence10

```
rule Precedence10 ::=
    Precedence10  '<' Precedence9
  | Precedence10  '>' Precedence9
  | Precedence10  '<=' Precedence9
  | Precedence10  '>=' Precedence9
  | Precedence9
  ;
```

The Precedence10 rule defines a non-terminal that represents operators with the same precedence level. It has five productions, each representing a comparison operator (‘<’, ‘>’, ‘<=’, ‘>=’), followed by the next higher precedence non-terminal Precedence9. If none of these cases apply, Precedence9 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of comparison operators.

Rule Precedence9

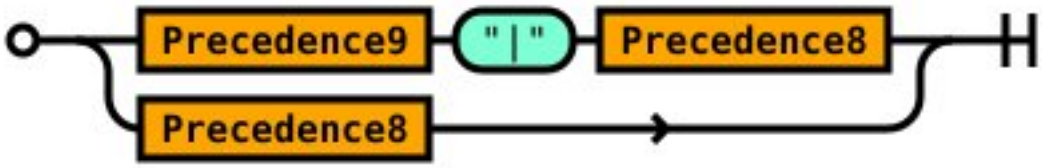


Figure 32: Precedence9

```
rule Precedence9 ::=
    Precedence9  '|' Precedence8
  | Precedence8
  ;
```

The Precedence9 rule defines a non-terminal that represents operators with the same precedence level. It has two productions: the first production specifies the logical OR operator ('|') and the next higher precedence non-terminal Precedence8, while the second production simply specifies Precedence8. If neither of these cases applies, Precedence8 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of logical OR operators.

Rule Precedence8

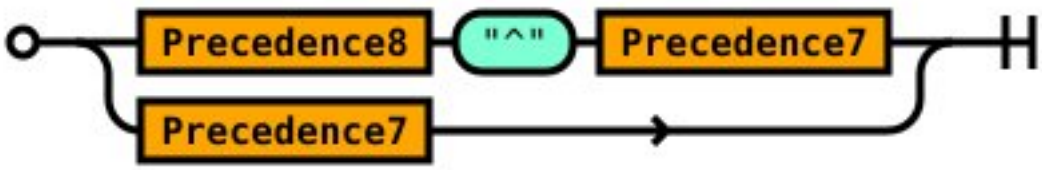


Figure 33: Precedence8

```
rule Precedence8 ::=
    Precedence8  '^' Precedence7
  | Precedence7
  ;
```

The Precedence8 rule defines a non-terminal that represents operators with the same precedence level. It has two productions: the first production specifies the bitwise XOR operator ('^') and the next higher precedence non-terminal Precedence7, while the second production simply specifies Precedence7. If neither of these cases applies, Precedence7 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of bitwise XOR operators.

Rule Precedence7

```
rule Precedence7 ::=
    Precedence7  '&' Precedence6
  | Precedence6
```

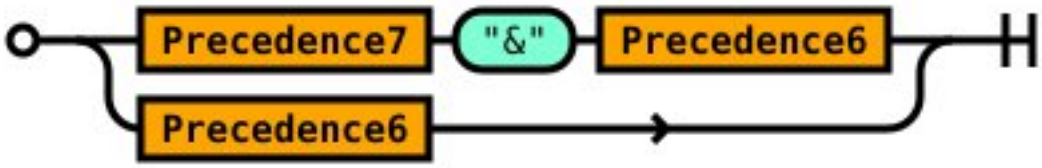


Figure 34: Precedence7

```
  ;
```

The Precedence7 rule defines a non-terminal that represents operators with the same precedence level. It has two productions: the first production specifies the bitwise AND operator (‘&’) and the next higher precedence non-terminal Precedence6, while the second production simply specifies Precedence6. If neither of these cases applies, Precedence6 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of bitwise AND operators.

Rule Precedence6

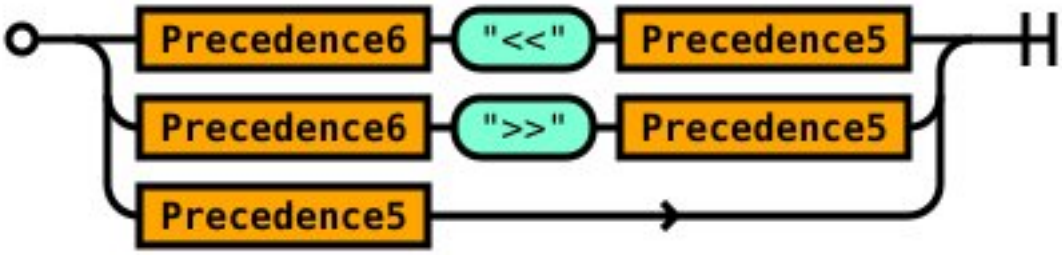


Figure 35: Precedence6

```
rule Precedence6 ::=
    Precedence6  '<<' Precedence5
  | Precedence6  '>>' Precedence5
  | Precedence5
  ;
```

The Precedence6 rule defines a non-terminal that represents operators with the same precedence level. It has three productions: the first production specifies the bitwise left shift operator (‘<<’) and the next higher precedence non-terminal Precedence5, the second production specifies the bitwise right shift operator (‘>>’) and Precedence5, while the third production simply specifies Precedence5. If neither of the first two cases applies, Precedence5 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of bitwise left shift and bitwise right shift operators.

Rule Precedence5

```
rule Precedence5 ::=
    Precedence5  '+' Precedence4
```

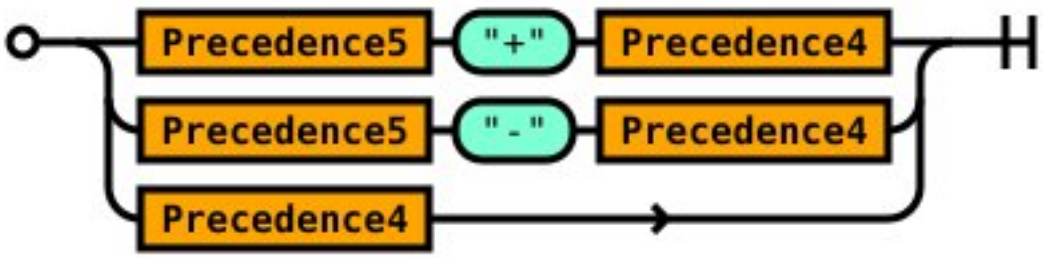


Figure 36: Precedence5

```
  | Precedence5  '-' Precedence4
  | Precedence4
  ;
```

The Precedence5 rule defines a non-terminal that represents operators with the same precedence level. It has three productions: the first production specifies the addition operator (‘+’) and the next higher precedence non-terminal Precedence4, the second production specifies the subtraction operator (‘-’) and Precedence4, while the third production simply specifies Precedence4. If neither of the first two cases applies, Precedence4 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of addition and subtraction operators.

Rule Precedence4

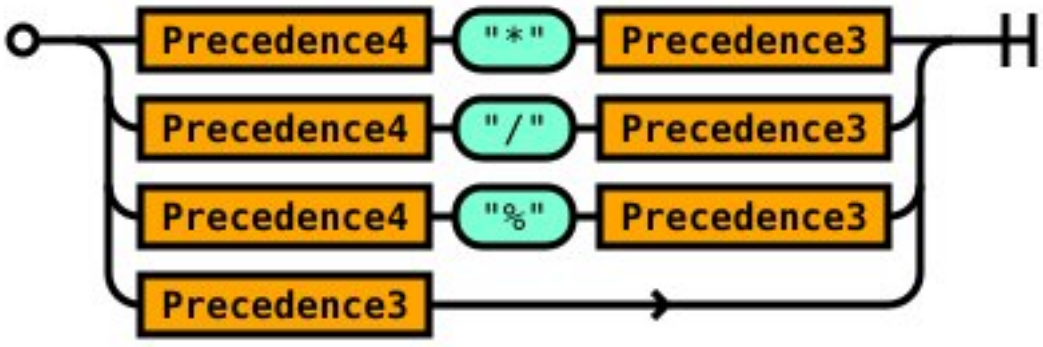


Figure 37: Precedence4

```
rule Precedence4 ::=
    Precedence4  '*' Precedence3
  | Precedence4  '/' Precedence3
  | Precedence4  '%' Precedence3
  | Precedence3
  ;
```

The Precedence4 rule defines a non-terminal that represents operators with the same precedence level. It has four productions: the first production specifies the multiplication operator (‘*’) and the next higher precedence non-terminal Precedence3, the second production specifies the division operator (‘’/’) and Precedence3, the third production specifies

the modulus (or remainder) operator (‘%’) and Precedence3, while the fourth production simply specifies Precedence3. If none of the first three cases apply, Precedence3 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of multiplication, division, and modulus operators.

Rule Precedence3

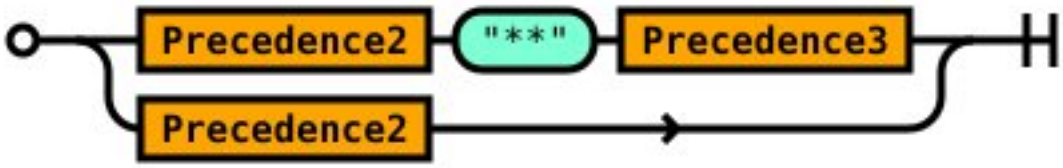


Figure 38: Precedence3

```
rule Precedence3 ::=
    Precedence2  '**' Precedence3
  | Precedence2
  ;
```

The Precedence3 rule defines a non-terminal that represents operators with the same precedence level. It has two productions: the first production specifies the exponentiation operator (‘**’) and the next higher precedence non-terminal Precedence2, while the second production simply specifies Precedence2. If the first case applies, Precedence3 is considered as the result of the operation Precedence2 raised to the power of Precedence3. Otherwise, Precedence2 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of exponentiation operators.

Rule Precedence2

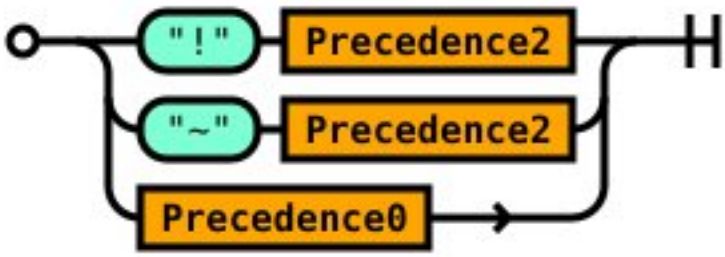


Figure 39: Precedence2

```
rule Precedence2 ::=
     '!' Precedence2
  |  '~' Precedence2
  | Precedence0
  ;
```

The Precedence2 rule defines a non-terminal that represents operators with the same precedence level. It has three productions: the first production specifies the logical NOT operator (‘!’) and the same precedence non-terminal Precedence2, the second production specifies the bitwise NOT operator (‘~’) and Precedence2, while the third production simply specifies Precedence0. If either of the first two cases applies, the operator is applied to the result of Precedence2. Otherwise, Precedence0 is directly considered as the result of this rule. This rule is often used in writing a parser or compiler for a programming language, as it specifies the precedence and associativity of logical NOT and bitwise NOT operators.

Rule NamedArgument

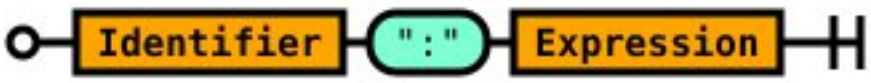


Figure 40: NamedArgument

```
rule NamedArgument ::=
    Identifier  ':' Expression
  ;
```

The NamedArgument rule defines a non-terminal that represents a named argument in a function or method call. It consists of an Identifier followed by a colon and an Expression. The Identifier represents the name of the argument, while the Expression represents its value. This rule is often used in programming languages that support named arguments, allowing for more readable and self-documenting code.

Rule FunctionCall

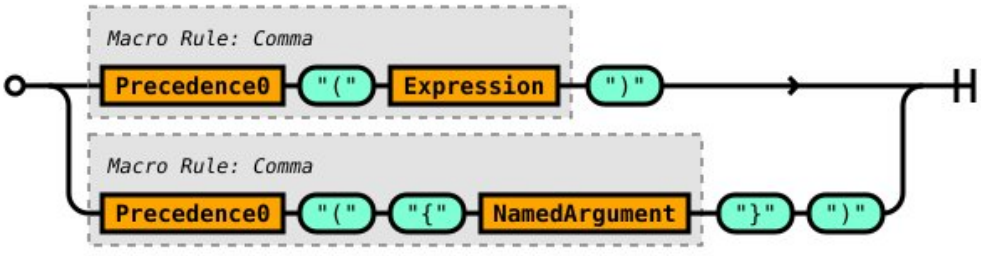


Figure 41: FunctionCall

```
rule FunctionCall ::=
    Precedence0  '(' Comma!(Expression)  ')'
  | Precedence0  '('  '{' Comma!(NamedArgument)  '}'  ')'
  ;
```

The FunctionCall rule defines a non-terminal that represents a function or method call. It has two productions: the first production specifies an ordered argument list, where the function or method is represented by Precedence0, and the arguments are represented by a comma-separated list of Expression. The second production specifies a named

argument list, where the function or method is represented by Precedence0, and the arguments are represented by a comma-separated list of NamedArgument enclosed in braces. This rule is often used in programming languages to represent function or method calls, with the syntax varying depending on the language's support for named arguments and other features.

Rule Precedence0

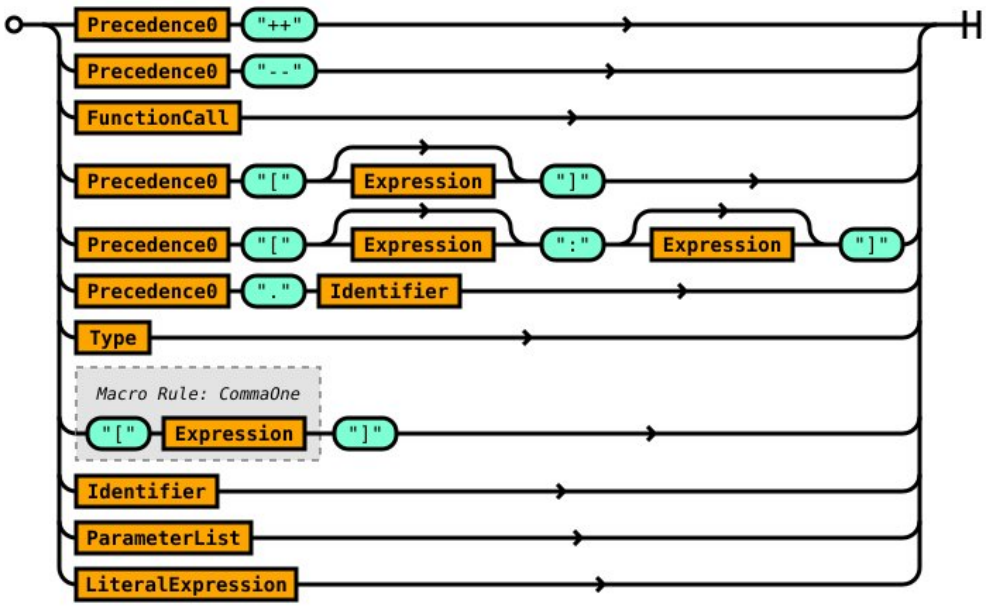


Figure 42: Precedence0

```
rule Precedence0 ::=
    Precedence0  '++'
  | Precedence0  '--'
  | FunctionCall
  | Precedence0  '[' Expression ?   ']'
  | Precedence0  '[' Expression ?   ':' Expression ?   ']'
  | Precedence0  '.' Identifier
  | Type
  |  '[' CommaOne!(Expression)  ']'
  | Identifier
  | ParameterList
  | LiteralExpression
  ;
```

The Precedence0 rule defines a non-terminal that represents primary expressions, which are the building blocks of more complex expressions. It has multiple productions, including the increment and decrement operators ('++' and '--'), function or method calls (represented by the non-terminal FunctionCall), array indexing with an Expression inside square brackets, slice notation with two optional Expression separated by a colon inside square brackets, accessing a member of an object using a dot and an Identifier, type names (represented by the non-terminal Type), an array literal with one or more comma-separated Expression enclosed in square brackets, an Identifier, a ParameterList, and

a LiteralExpression. These productions specify the most basic elements that can be combined to form more complex expressions in a programming language.

Rule LiteralExpression

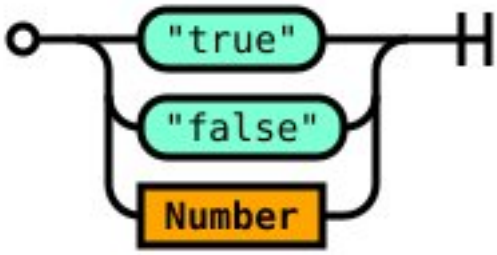


Figure 43: LiteralExpression

```
rule LiteralExpression ::=
     'true'
  |  'false'
  | Number
  ;
```

The LiteralExpression rule defines a non-terminal that represents literal values in a programming language. It has three productions: the first production specifies the Boolean value true, the second production specifies the Boolean value false, and the third production specifies a numeric literal value represented by the non-terminal Number. This rule is often used in programming languages to represent literal values that can be used in expressions, such as Boolean values and numeric constants.

Rule Parameter

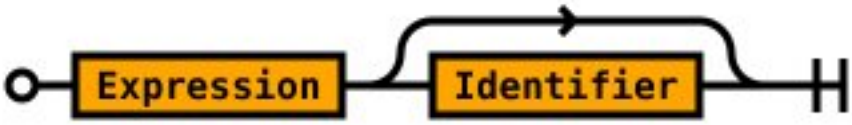


Figure 44: Parameter

```
rule Parameter ::=
    Expression Identifier ?
  ;
```

The Parameter rule defines a non-terminal that represents a parameter in a function or method declaration. It consists of an Expression followed by an optional Identifier. The Expression represents the default value of the parameter, while the Identifier represents its name. If the Identifier is not specified, the parameter is treated as an anonymous parameter. This rule is often used in programming languages to define the parameters of functions or methods, allowing the caller to pass arguments to the function or method at runtime.

Rule OptParameter

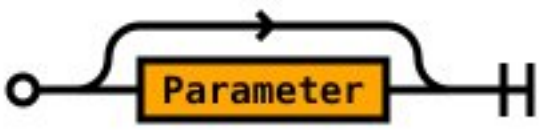


Figure 45: OptParameter

```
rule OptParameter ::=
    Parameter ?
  ;
```

The OptParameter rule defines a non-terminal that represents an optional parameter in a function or method declaration. It consists of an optional Parameter. If the Parameter is present, it specifies a parameter with a default value and an optional name, while if it is absent, there is no default value and no name. This rule is often used in programming languages to provide optional parameters to functions or methods, allowing the caller to omit them if they are not needed.

Rule ParameterList

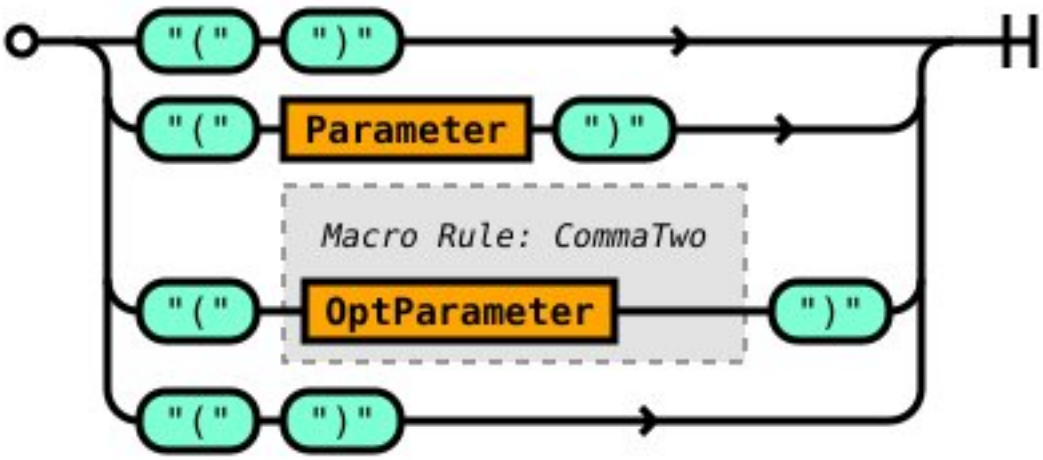


Figure 46: ParameterList

```
rule ParameterList ::=
     '('  ')'
  |  '(' Parameter  ')'
  |  '(' CommaTwo!(OptParameter)  ')'
  |  '('  ')'
  ;
```

The ParameterList rule defines a non-terminal that represents a list of parameters in a function or method declaration. It has four productions: the first production specifies an empty parameter list, the second production

specifies a single parameter, the third production specifies two or more parameters separated by commas, and the fourth production again specifies an empty parameter list. Each parameter is represented by the OptParameter non-terminal, which itself represents an optional parameter with an optional name and default value. This rule is often used in programming languages to define the parameters of functions or methods, allowing the caller to pass arguments to the function or method at runtime.

Rule BlockStatementOrSemiColon

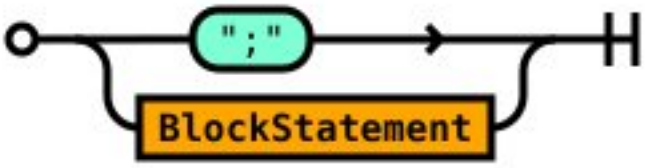


Figure 47: BlockStatementOrSemiColon

```
rule BlockStatementOrSemiColon ::=
     ';'
  | BlockStatement
  ;
```

The BlockStatementOrSemiColon rule defines a non-terminal that represents either a semicolon or a block statement in a programming language. It has two productions: the first production specifies a semicolon, and the second production specifies a BlockStatement, which can contain one or more statements enclosed in curly braces. This rule is often used in programming languages to represent the end of a statement or the beginning of a block of statements. If the BlockStatementOrSemiColon is a semicolon, it is typically used to indicate the end of a single statement. If it is a block statement, it is typically used to group multiple statements into a single block that can be executed together.

Rule FunctionDefinition

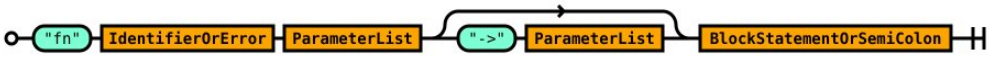


Figure 48: FunctionDefinition

```
rule FunctionDefinition ::=
     'fn' IdentifierOrError ParameterList (  '->' ParameterList ) ?
         BlockStatementOrSemiColon
  ;
```

The FunctionDefinition rule defines a non-terminal that represents a function definition in a programming language. It consists of the keyword 'fn', followed by an identifier that represents the name of the function, followed by a ParameterList that represents the parameters of the function, and an optional ParameterList that represents the return type of the function. Finally, it ends with a BlockStatementOrSemiColon that represents the body of the function. This

rule is often used in programming languages to define functions, which are reusable blocks of code that can be called from other parts of the program. The parameters of the function represent the input values that the function accepts, while the return type represents the output value that the function returns.

Rule BlockStatement

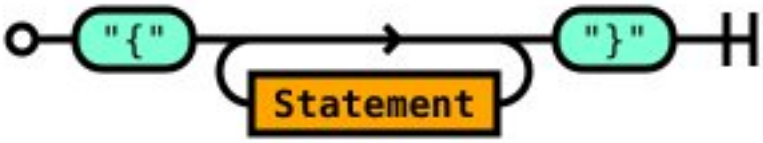


Figure 49: BlockStatement

```
rule BlockStatement ::=
     '{' Statement *   '}'
  ;
```

The BlockStatement rule defines a non-terminal that represents a block of statements in a programming language. It consists of an opening curly brace, followed by zero or more statements, and finally, a closing curly brace. The statements can be any valid statement in the programming language. This rule is often used in programming languages to group multiple statements together into a single block that can be executed as a unit. A block statement can be used in a variety of contexts, such as in a function body, a loop body, or an if statement body. The block statement allows the programmer to treat multiple statements as a single entity and can be used to make the code more organized and easier to read.

Rule OpenStatement

```
rule OpenStatement ::=
     'if'  '(' Expression  ')' Statement
  |  'if'  '(' Expression  ')' ClosedStatement  'else' OpenStatement
  |  'for'  '(' SimpleStatement ?   ';' Expression ?   ';' SimpleStatement ?   ')'
      OpenStatement
  ;
```

The OpenStatement rule defines a non-terminal that represents an open statement in a programming language. It has three productions: the first production specifies an “if” statement with a single statement in its body; the second production specifies an “if-else” statement with two sub-statements, one for the “if” condition and one for the “else” condition; the third production specifies a “for” loop with an optional initialization statement, an optional condition, and an optional post-statement, followed by an open statement body. This rule is often used in programming languages

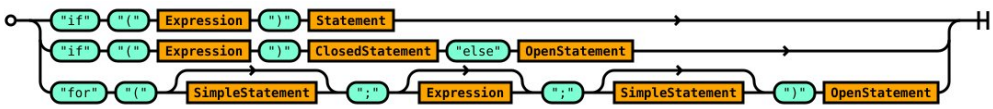


Figure 50: OpenStatement

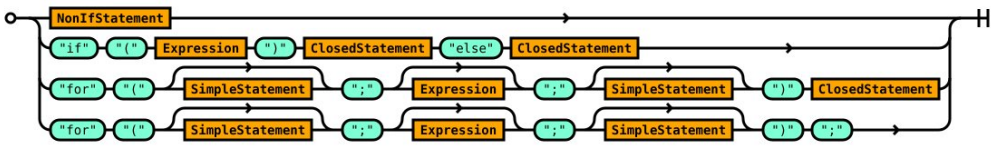


Figure 51: ClosedStatement

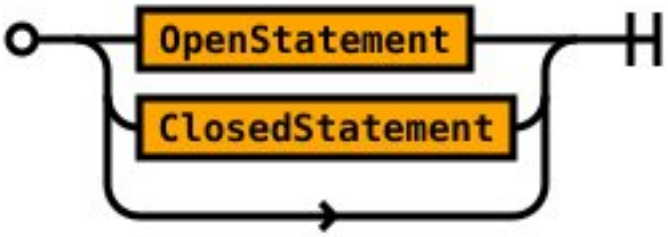


Figure 52: Statement

to control the flow of execution based on certain conditions or to perform iterative operations on a set of data. The "if" statement is used to execute a block of code if a certain condition is true. The "if-else" statement is used to execute one of two blocks of code based on whether a certain condition is true or false. The "for" loop is used to execute a block of code multiple times while a certain condition is true, and it includes an optional initialization statement, an optional condition, and an optional post-statement.

Rule ClosedStatement

```
rule ClosedStatement ::=
    NonIfStatement
  | 'if' '(' Expression ')' ClosedStatement 'else' ClosedStatement
  | 'for' '(' SimpleStatement ? ';' Expression ? ';' SimpleStatement ? ')'
    ClosedStatement
  | 'for' '(' SimpleStatement ? ';' Expression ? ';' SimpleStatement ? ')'
    ';'
  ;
```

The ClosedStatement rule defines a non-terminal that represents a closed statement in a programming language. It has four productions: the first production specifies a non-"if" statements, such as a loop or a function call; the second production specifies an "if-else" statement with two sub-statements, both of which are closed statements; the third production specifies a "for" loop with an optional initialization statement, an optional condition, and an optional post-statement, followed by a closed statement body; the fourth production specifies a "for" loop with an optional initialization statement, an optional condition, and an optional post-statement, followed by a semicolon. This rule is often used in programming languages to control the flow of execution based on certain conditions or to perform iterative operations on a set of data. The closed statement differs from the open statement in that it contains a complete sub-statement within its body.

Rule Statement

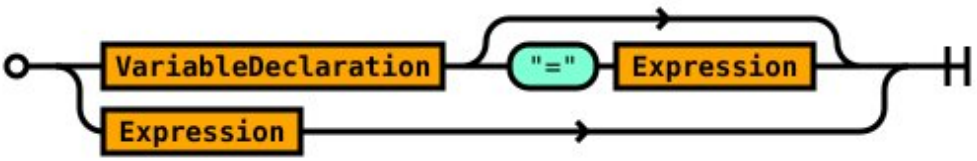


Figure 53: SimpleStatement

```
rule Statement ::=
    OpenStatement
  | ClosedStatement
  |
  ;
```

This rule is used to define the structure of statements in a programming language. Statements are used to express an action that needs to be carried out, such as assigning a value to a variable or looping over a set of data. The Statement rule is often used in the context of defining control structures like loops and conditionals.

Rule SimpleStatement

```
rule SimpleStatement ::=
    VariableDeclaration (  '=' Expression ) ?
  | Expression
  ;
```

The SimpleStatement rule is used to define a simple statement in a programming language. It has two possible productions. The first production specifies a VariableDeclaration, which may or may not be followed by an = sign and an Expression. This allows for the declaration of a variable with an optional initial value. The second production specifies an Expression, which is a piece of code that evaluates to a value.

Rule NonIfStatement

```
rule NonIfStatement ::=
    BlockStatement
  | SimpleStatement  ';'
  |  'continue'  ';'
  |  'break'  ';'
  |  'return'  ';'
  |  'return' Expression  ';'
  ;
```

The NonIfStatement rule is used to define a non-conditional statement in a programming language. It has several possible productions, including a BlockStatement and a SimpleStatement followed by a semicolon.

The BlockStatement production allows for a block of code to be executed as a single unit and is often used to group multiple statements. The SimpleStatement followed by a semicolon production allows for a single statement to be executed.

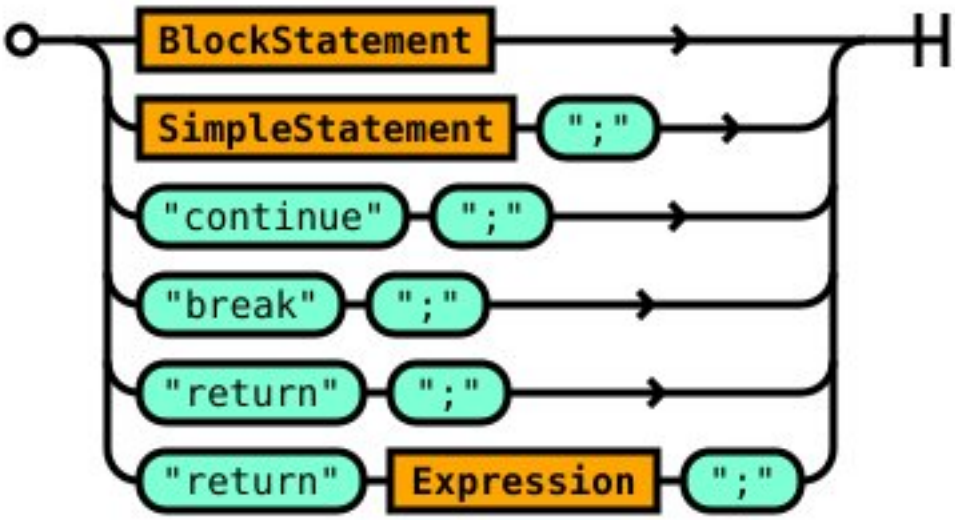


Figure 54: NonIfStatement

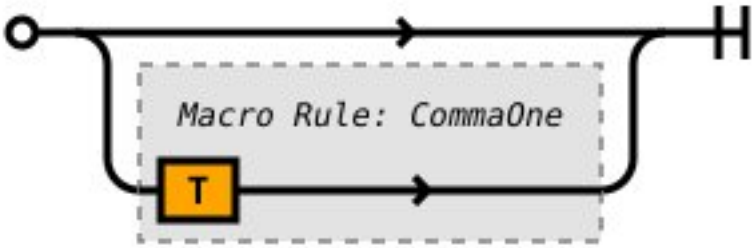


Figure 55: Comma

In addition, the rule specifies three special types of statements: continue, break, and return. The continue statement is used to skip to the next iteration of a loop, while the break statement is used to exit a loop. The return statement is used to exit a function and optionally return a value.

Overall, the NonIfStatement rule is an important part of defining the syntax of a programming language and is used to express a wide range of statements and behaviors.

Macro Comma

```
macro Comma<T> ::=

  | CommaOne!(T)
  ;
```

This is a macro definition for a comma-separated list of elements of type T. It allows for zero or more elements separated by commas. The `!` symbol after the macro name indicates that the macro is left-recursive, meaning it can be applied repeatedly until it no longer matches. The `CommaOne!` in the definition means that the macro should match at least one element before the optional commas.

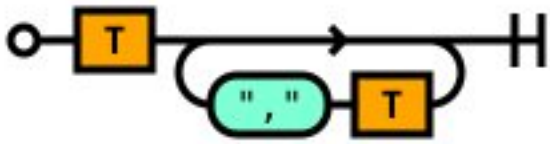


Figure 56: CommaOne

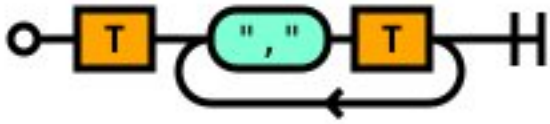


Figure 57: CommaTwo

Macro CommaOne

```
macro CommaOne<T> ::=
    T (  ',' T ) *
  ;
```

This is a macro definition for a comma-separated list of one or more elements of type `T`. It first matches a single element of type `T`, followed by zero or more occurrences of a comma followed by another element of type `T`. The `*` after the second parentheses indicates that this sequence can repeat zero or more times. This macro enforces that there must be at least one element in the comma-separated list.

Macro CommaTwo

```
macro CommaTwo<T> ::=
    T (  ',' T ) +
  ;
```

The CommaTwo macro is used to match one or more occurrences of the same non-terminal symbol separated by commas.

Rule Number

```
rule Number ::=
     'r#([1-9][0-9]*|0)#'
```

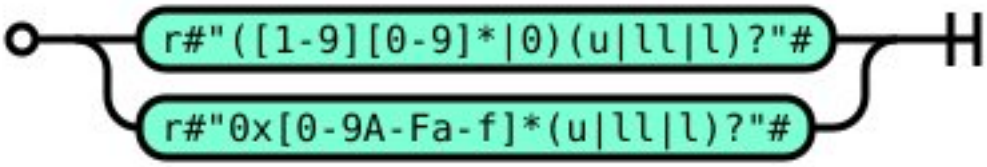


Figure 58: Number

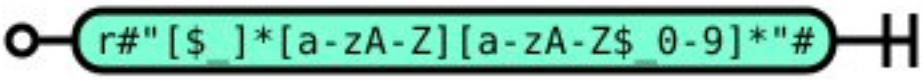


Figure 59: Identifier

Figure 60: StringLiteral

```
  |  'r#0x[0-9A-Fa-f]*#'
  ;
```

The Number rule is used to define a number in a programming language. It has two possible productions. The first production matches a decimal number, while the second production matches a hexadecimal number.

Rule Identifier

```
rule Identifier ::=
'r#[$_][a-zA-Z][a-zA-Z$_0-9]#'
;
```

The Identifier rule is used to define an identifier in a programming language. It matches a sequence of characters that starts with a letter or underscore, followed by zero or more letters, numbers, underscores, or dollar signs.

Rule StringLiteral

```
rule StringLiteral ::=
'r#\[^\]*\#'
;
```

The StringLiteral rule is used to define a string literal in a programming language. It matches a sequence of characters surrounded by square brackets.

3.4 Smart Contracts

Ola contracts allow users to write complex business logic that will be deployed to Ola's Layer2 network, and cross-contract calls can be written between different contracts just like solidity.

3.4.1 Simple Examples

The following example shows a recursive and non-recursive Ola smart contract implementation of Fibonacci numbers.

```
contract Fibonacci {
```

```
    fn main() {
       fib_non_recursive(10);
    }

    fn fib_recursive(u32 n) -> (u32) {
        if (n == 0) {
            return 0;
        }
        if (n == 1) {
            return 1;
        }
        return fib_recursive(n - 1) + fib_recursive(n - 2);
    }

    fn fib_non_recursive(u32 n) -> (u32) {
        if (n == 0) {
            return 0;
        }
        u32 first = 0;
        u32 second = 1;
        u32 third = 1;
        for (u32 i = 2; i <= n; i++) {
             third = first + second;
             first = second;
             second = third;
        }
        return third;
    }

}
```

The following shows a simple Person contract that contains a person structure, assigns a value to the Person structure, and reads the status of the person.

```
contract Person {
    enum Sex {
        Man,
        Women
    }

    struct Person {
        Sex s;
        u32 age;
        u256 id;
    }

    Person p;

    fn newPerson(Sex s, u32 age, u256 id) {
        p = Person(s, age, id);
    }
```

```
    fn getPersonId() -> (u256) {
        return p.id;
    }

    fn getAge() -> (u32) {
        return p.age;
    }
}
```

3.4.2 Multiple files

For better project organization and clearer logic, it is common to split the contents of a file into multiple files. Ola language supports the import of another contract within a contract through the `import` keyword.

An example of a multi-file contract is shown below.

Contract RectangularCalculator

```
contract RectangularCalculator {
    fn rectangle(u32 w, u32 h) -> (u32 s, u32 p) {
        s = w * h;
        p = 2 * (w + h);
        // Returns a variable with the same name, the return can be ignored
        // return (s, p)
    }
}
```

Contract ShapeCalculator

```
contract SquareCalculator {

    fn square(u32 w) -> (u32 s, u32 p) {
        s = w * w;
        p = 4 * w;
        return (s, p);
    }
}
```

Contract Calculator

```
import "./RectangularCalculator";
import "./SquareCalculator";

contract Calculator {

    fn sum(u32 w, u32 h) -> (u32 s, u32 p) {
        (u32 rectangle_s, u32 rectangle_p) = rectangle(w, h);
        (u32 square_s, u32 square_p) = square(w);
        return (rectangle_s + square_s, rectangle_p + square_p);
    }
}
```

4 Ola-Compiler: A LLVM IR Based Compiler to Generate RISC Assembly

4.1 Ola Compiler Introduction

The Ola Compiler compiles the high-level Ola contract code into the assembly code supported by OlaVM. The general pipeline process is shown as **Figure 61**:

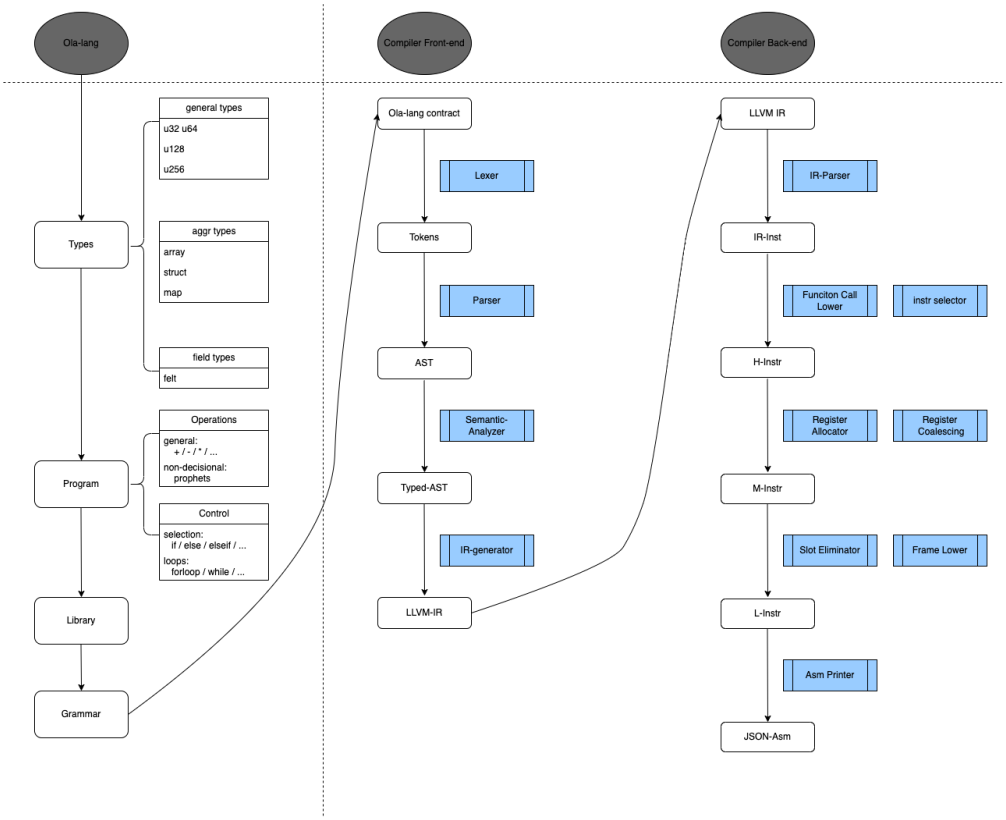


Figure 61: Ola-lang Language and Compiler Pipeline

As can be seen from the above figure, the frontend of the compiler takes the high-level contract program as input and then compiles it into LLVM Intermediate Representation (IR); and the backend of the compiler takes the LLVM IR generated by the frontend as input and then compiles it into Ola assembly code.

The assembly code is eventually assembled, linked, loaded, and executed by OlaVM through the toolchain pipeline to generate a trace.

To illustrate the compiler pipeline, the following is an example of a typical u32 type sqrt operation with instructions and Prophet two different versions to show the code generation process.

An example Ola-lang high-level language program for computing sqrt of type u32 with Prophet version is as follows:

```
contract SqrtContract {
    fn main() {
        sqrt_test(4);
    }
```

```
    fn sqrt_test(u32 n) -> (u32) {
        u32 b = u32_sqrt(n);
        return b;
    }
}
```

An example ola-lang high-level language program for computing sqrt of type u32 with instructions version is as follows:

```
contract SqrtContract {
    fn main() {
       sqrt_test(4);
    }

    fn sqrt_test(u32 a) -> (u32) {
        u32 result = 0;
        if (a > 3) {
            result = a;
            u32 x = a / 2 + 1;
            // assume the maximum iteration is 100
            for (u32 i = 0; i < 100; i++) {
                if (x >= result) break;
                result = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            result = 1;
        }
        return result;
    }
}
```

4.2 Ola Language Compiler Frontend

This section introduces the key components and functionalities of the Ola language compiler frontend. We will discuss the process of lexical analysis, syntax parsing, Abstract Syntax Tree (AST) generation, semantic analysis, and LLVM Intermediate Representation (IR) generation in detail.

The processing flow of the compiler frontend is shown in the following figure62:

4.2.1 Ola Language Parser

Lexical Analysis

Lexical analysis is the first stage of the compiler frontend. In this phase, the goal is to break down the source code into a series of tokens. The Ola language lexer will handle the following elements:

- Keywords
- Identifiers
- Operators

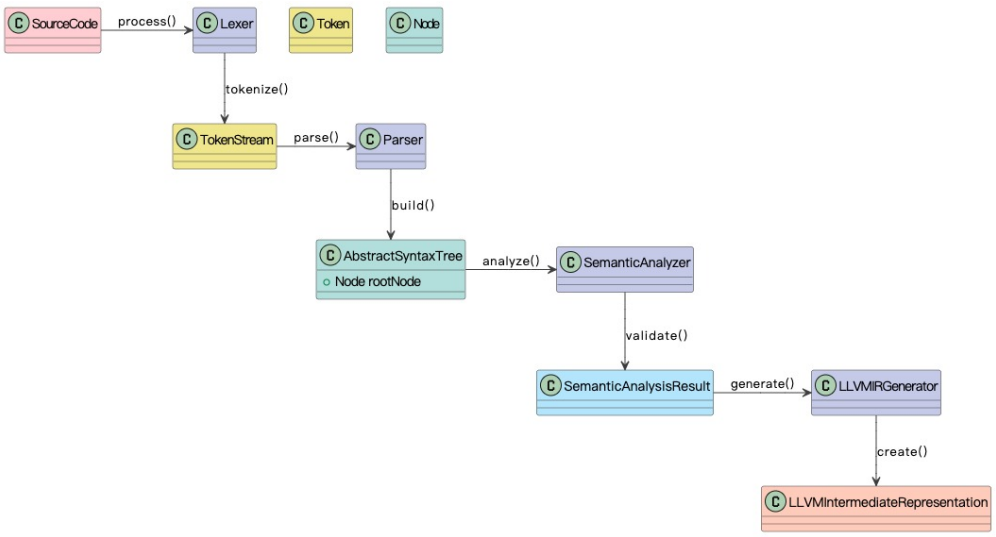


Figure 62: Ola-lang Compiler Frontend

- Literals (such as strings, numbers, and boolean values)
- Comments
- Delimiters (such as parentheses and commas)

Additionally, the lexer will eliminate whitespace and comments, ensuring a clean token stream for the next phase.

Syntax Parsing

Syntax parsing is the process of transforming the tokens generated in the lexical analysis phase into an Abstract Syntax Tree (AST). The Ola language compiler will implement a top-down parser, such as a Recursive Descent Parser, to support Ola language's grammar.

This section will also discuss the implementation of error handling and recovery mechanisms, ensuring that the parser can handle syntax errors gracefully and provide helpful error messages to the user.

Abstract Syntax Tree (AST) Generation

During the syntax parsing phase, the parser will generate an AST representing the program's structure. This section will explain the design of the AST data structures and the process of constructing the AST during parsing. Additionally, it will cover the benefits of using an AST, such as enabling easier manipulation and analysis of the code's structure.

The Ola compiler seamlessly integrates the Lexical Analysis, Syntax Parsing, and Abstract Syntax Tree (AST) generation processes, forming a cohesive and efficient pipeline. By leveraging the LALRPOP framework, these phases work in harmony, transforming the Ola source code into an AST representation that is suitable for subsequent compiler phases. This unified approach not only simplifies the implementation but also enhances the performance and robustness of the Ola compiler.

By following these steps, the compiler can efficiently convert the Ola source code text into a sequence of tokens:

- The first step in implementing the lexical analysis phase of the Ola compiler is to define lexer rules for various token categories, including keywords, identifiers, literals, and operators. These rules should be based on the

provided EBNF grammar rules. We created a file named `Ola.lalrpop` that describes Ola's EBNF grammar rules.

- After defining the lexer rules, the next step is to integrate the lexer with the parser. This can be achieved by using the `lexer` attribute in LALRPOP grammar rules. The `lexer` attribute specifies which lexer rule should be used to recognize a particular grammar production.
- Ola compiler provides error handling and reporting. If the lexer encounters an unexpected character or a malformed token, it generates an error with the corresponding position in the input text. This information can be used to provide helpful error messages to the user.

Once the lexical analysis phase is complete, the generated sequence of tokens can be passed to the parser, which will construct an abstract syntax tree (AST) based on the defined grammar rules. This AST can then be further processed by subsequent phases of the Ola compiler, such as semantic analysis, optimization, and code generation, ultimately producing executable code for the target platform.

By leveraging the powerful LALRPOP framework, the Ola compiler can efficiently perform lexical analysis and provide robust error handling, ensuring that the compiler is user-friendly and capable of handling complex Ola source code.

4.2.2 Semantic Analysis

The Semantic Analysis phase of the Ola compiler is an extensive process that ensures the program's correctness and consistency. As previously mentioned, this phase consists of several sub-tasks. Here, we delve deeper into each sub-task, providing a more detailed and comprehensive explanation of the process.

Symbol Resolution

The compiler analyzes the program's scope and context to resolve symbols accurately. It distinguishes between local and global variables, function declarations, and type definitions. The symbol table, which holds information about each symbol, is updated as the compiler traverses the AST. During this process, the compiler also checks for naming conflicts and multiple declarations, ensuring that the program adheres to Ola's scoping rules.

LibFunction Identification

In the semantic analysis phase, we will identify all libFunction names that users call. We will also construct prototype code for these LibFunctions and verify whether the parameters used to call them match the parameter types and numbers in the prototype code. If there is a match, we will record them for easy processing of IR generation for Lib Functions in subsequent LLVM IR generation phases.

Type Checking

The compiler ensures that each operation and expression in the program involves operands of compatible types. In this stage, the compiler also infers the types of expressions when necessary and enforces type constraints for function calls, assignments, and arithmetic operations.

Control Flow Analysis

In addition to checking for unreachable code and infinite loops, the control flow analysis process verifies that:

- All code paths in a function that should return a value must end with a return statement.
- Break and continue statements appear only within loops.
- Variables are declared before they are used.

Constant Expression Evaluation

During this step, the compiler performs the following tasks:

- Evaluates arithmetic and bitwise operations on constant expressions at compile-time, ensuring that the generated code is more efficient.
- Detects potential errors, such as array index out-of-bounds, by evaluating expressions that involve constants.
- Folds constant expressions, such as mathematical operations or string concatenations, reducing the code size and improving execution efficiency.

Semantic Validation

The final step of the semantic analysis phase consists of several validation checks, including:

- Verifying that variables are initialized before they are used.
- Ensuring that variables, functions, and types are declared only once within a given scope.
- Checking that all required function arguments are provided and that excess arguments are not supplied.
- Validating that return statements are used correctly within functions.

The Semantic Analysis phase is crucial for the robustness and correctness of the Ola compiler. By performing these comprehensive checks, the compiler can guarantee that the generated code adheres to the language's semantic rules and is free from errors that might lead to unexpected behavior during execution. With a semantically verified AST, the Ola compiler proceeds to the subsequent phases of the compilation process, ensuring the efficient translation of the source code into executable code tailored for the OlaVM.

4.2.3 LLVM IR Generation

The LLVM IR Generation phase is a crucial step in the Ola compiler, as it translates the Abstract Syntax Tree (AST) obtained from the semantic analysis into LLVM Intermediate Representation (IR). This phase leverages the Inkwell framework, a powerful and user-friendly library that simplifies the process of generating LLVM IR code in Rust.

Inkwell Initialization

Initialize the Inkwell context by creating a new `Context` object. Set up the target information by querying the target machine properties, such as target triple, data layout, and target-specific optimization levels. This information helps in generating the correct LLVM IR code tailored for the target architecture.

The pseudocode is shown below:

```
// Create a new LLVM context
let context = Context::create();
let module = context.create_module(name);
let builder = context.create_builder();
```

Lib IR Generation

Before generating LLVM IR for user-defined functions, the compiler will first generate LLVM IR for previously called lib functions, prophet functions, and builtin functions. These functions have fixed logic that Ola contract developers will not change.

The pseudocode is shown below:

```
declare_builtins(bin)
declare_prophets(bin)

for each lib function in ns.called_lib_functions:
    if lib function == "u32_sqrt":
        define function u32_sqrt(value: i64) -> i64:
            root = prophet_u32_sqrt(value)
            builtin_range_check(root)
            root_squared = root * root
            builtin_assert(root_squared, value)
            return root
    else if lib function == "u32_xxx":
        // do nothing
    else:
        // do nothing
```

AST Traversal

Traverse the AST using a depth-first approach. Focus on visiting function nodes, as they represent the main building blocks of the Ola program. For each function node encountered, generate its corresponding LLVM IR.

The pseudocode is shown below:

```
fn traverse_ast(node: &AstNode) {
    if let AstNode::Function(func) = node {
        generate_function(&func);
    }

    for child in node.children() {
        traverse_ast(child);
    }
}
```

Function Generation

For each Ola function, create a corresponding LLVM function by invoking the `add_function` method on the Inkwell module. Map the Ola function's return type and argument types to their corresponding LLVM types using a custom type mapping function. Add the function arguments with their respective types to the LLVM function using the `append_parameter` method.

The pseudocode is shown below:

```
fn generate_function(func: &OlaFunction) -> LLVMFunction {
    // Map Ola function type to LLVM function type
    let func_type = map_ola_type_to_llvm_type(&func.return_type);
    let llvm_func = context.create_function(func.name, func_type);

    // Add function arguments
    for arg in &func.arguments {
        let llvm_arg_type = map_ola_type_to_llvm_type(&arg.type);
        llvm_func.add_argument(llvm_arg_type, arg.name);
    }

    // Create a builder and position it at the entry block
    let builder = context.create_builder();
    builder.position_at_end(&llvm_func.entry);

    // Process each statement in the function body
    for stmt in &func.body {
        process_statement(stmt, &builder);
    }

    // Return the generated LLVM function
    llvm_func
}
```

Statement Processing

Based on the type of statement encountered in the function body, generate the appropriate LLVM IR instructions. For expression statements, process the expression and generate the corresponding LLVM IR code. For variable declaration statements, allocate memory for the variable on the stack using the `build_alloca` method, and store its initial value using the `build_store` method. For control flow constructs, such as loops and conditionals, generate appropriate branching instructions using methods like `build_conditional_branch` and `build_loop`.

The pseudocode is shown below:

```
fn process_statement(stmt: &Statement, builder: &Builder) {
    match stmt {
        Statement::Expression(expr) => process_expression(expr, builder),
        Statement::VariableDeclaration(var_decl) => process_variable_declaration(
            var_decl, builder),
        // ... other statement types
    }
```

```
}
```

Expression Processing

Process each expression encountered in the function body and generate the corresponding LLVM IR instructions. This includes handling arithmetic operations, logical operations, and control flow constructs (such as if-else expressions). For each type of expression, call the appropriate function to generate the LLVM IR code. For example, for binary expressions, generate the appropriate arithmetic or logical operation using methods like `build_add`, `build_mul`, or `build_and`. For function calls, generate the `call` instruction using the `build_call` method.

The pseudocode is shown below:

```
fn process_expression(expr: &Expression, builder: &Builder) -> LLVMValue {
    match expr {
        Expression::BinaryExpression(op, lhs, rhs) => generate_binary_expression(op,
            lhs, rhs, builder),
        Expression::IfExpression(cond, then_expr, else_expr) =>
            generate_if_expression(cond, then_expr, else_expr, builder),
        // ... other expressions
    }
}
```

Type Mapping

Type Mapping is a crucial function that needs to be implemented in order to convert Ola types into their corresponding LLVM types. This mapping function should handle all Ola types, including basic (such as u32, u64, and u256) and complex types (such as structs and arrays). It's important to note that the Ola language is based on the Field type at its core, with various integer types built upon it. The Field Order is `0xFFFFFFFF00000001`. To fully represent a field element, we create an integer type using the method `context.i64_type()`. For complex types such as structs and arrays, custom LLVM types can be created using methods like `context.struct_type()` and `context.array_type()`.

The pseudocode is shown below:

```
fn map_ola_type_to_llvm_type(ola_type: &OlaType) -> LLVMType {
    match ola_type {
        OlaType::Field => context.i64_type(),
        OlaType::U32 => context.i32_type(),
        OlaType::U64 => context.i64_type(),
        OlaType::U256 => context.array_type(context.i64_type(), 4),
        OlaType::Struct(name) => context.struct_type(&structs[name].fields, false),
        OlaType::Array(elem_type, size) => context.array_type(
            map_ola_type_to_llvm_type(elem_type), *size),
        // ... other types
    }
}
```

4.3 Ola Compiler Backend: from LLVM IR to OlaVM assembly

Ola compiler backend bridge IR structure of module, function and Instruction Set Architecture(ISA) related callconv, registers and instructions. Its main features are code generation related lower and optimization related passes.

Target ISA mainly contains custom target instructions, registers which contain register class and register information, call convention and datalayout information.

Module in addition to inheritance IR is parsed out as Module structure, the description of its function and differ significantly with LLVM IR. Data information of Basic Block (BB) in the instructions target for the instruction, the register contains VRegs (Virtual Registers) and RegUnit (Register Unit) two categories, and contains a VRegs to Instructions (Insts) mapping. At the same time, instruction in data layout is referred to target instruction. Note that the structure of slot which contains stack base pointer and stack offset describe the memory access operations of parameters, local variables.

The lower provides the process of downgrading IR instruction to target instruction. Specifically it also requires copy parameters to VRegs for function call.

The pass module contains the register allocation (RegAlloc, RA) and spiller for analyzing the liveness of the pass and the function pass.

4.3.1 Backend Introduction

The backend of the Ola compiler compiles IR into target assembly code. It takes the standard LLVM IR generated by the frontend as input and the Ola assembly code as output.

Its pipeline process is as follows figure 63:

An example ola-lang assembly code for computing sqrt of type u32 with prophet version is as follows:

```
{
  "program": "u32_sqrt:\n.LBL3_0:\n  mov r3 r1\n  mov r1 r3\n.PROPHET3_0:\n  mov r0
      psp\n  mload r0 [r0,0]\n  range r0\n  mul r2 r0 r0\n  assert r2 r3\n  ret\
      nmain:\n.LBL4_0:\n  add r8 r8 4\n  mstore [r8,-2] r8\n  mov r1 4\n  call
      sqrt_test\n  add r8 r8 -4\n  end\nsqrt_test:\n.LBL5_0:\n  add r8 r8 6\n
      mstore [r8,-2] r8\n  mov r0 r1\n  mstore [r8,-3] r0\n  mload r1 [r8,-3]\n
      call u32_sqrt\n  mstore [r8,-4] r0\n  mload r0 [r8,-4]\n  add r8 r8 -6\n  ret\
      n",
  "prophets": [
    {
      "label": ".PROPHET3_0",
      "code": "%{\n    entry() {\n        cid.y = sqrt(cid.x);\n    }\n%}",
      "inputs": [
        "cid.x"
      ],
      "outputs": [
        "cid.y"
      ]
    }
  ]
}
```

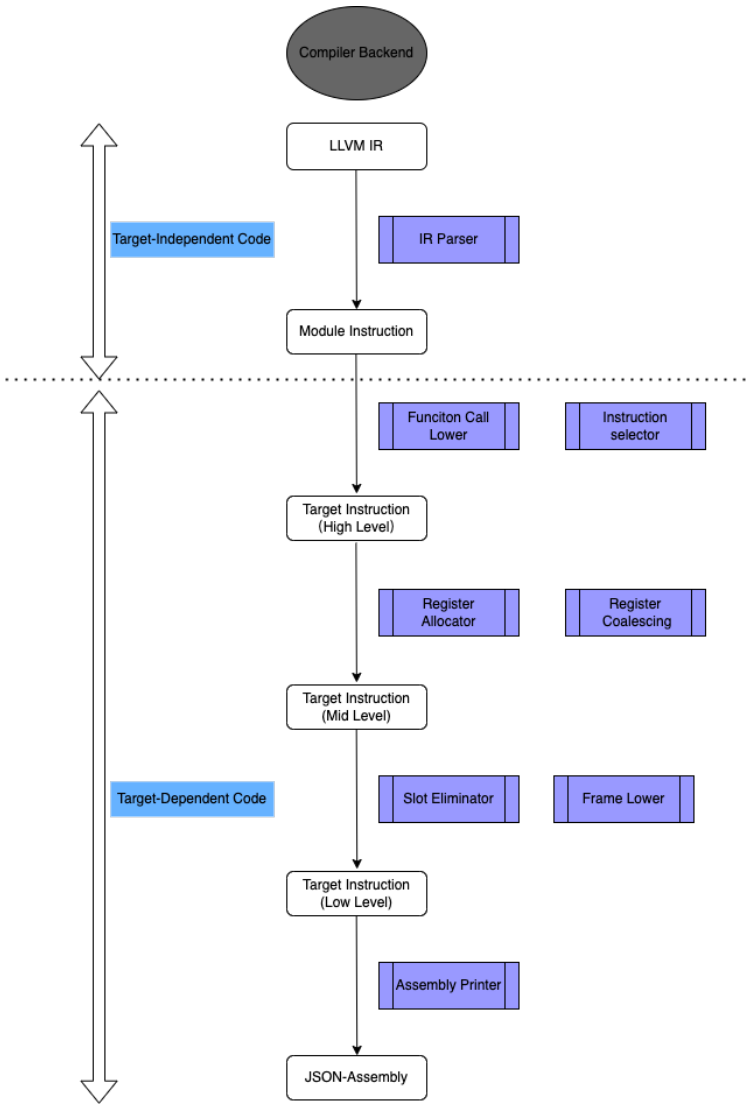


Figure 63: Ola-lang Backend Pipeline

The backend data structure contains mainly lists, module instructions, and target instructions. The Application Binary Interface(ABI) then contains Function call specification and mapping relationships to virtual memory.

The backend codegen pipeline process is briefly as follows:

- IR parsing
- Function call lowering
- Instruction selection
- Register allocation
- Slot elimination
- Stack frame handling
- Assembly printing

4.3.2 Parser: Parse LLVM IR to Module Instruction

- IR Module

  IR is composed of the following multi level structures:

  ```
  module -> function -> value -> types
  ```

  (1) Module structure contains Target which contains triple and datalayout information, Function, Attribute and GlobalVariable.

  (2) Function structure contains Name, Type information, Visibility, Attribute and Parameter list basic information and DataLayout.

  (3) Datalayout contains the sequential logical relationship between BasicBlock and the Instruction within them.

  (4) Data contains the specific Value, Instruction, BasicBlock list instances.

  (5) BasicBlock is identified by BasicBlockId and consists of two parts: name and number. Each BB usually contains one predecessor and one successor except for the entrance BB, which just only has one successor but not predecessor.

  (6) Instruction is identified by BasicBlockId, InstructionId and usually consists of opcode, source operands, dest operand and type of its operation.

  (7) Value contains Instruction, Argument, Constant and Inline Assembly types.

  (8) Due to the characteristics of the instruction set of OlaVM, The types currently supported by the compiler are mainly `i64` and the occasional `i1`.

- IR Parser

  The role of IR parser is to parse LLVM IR to Module Instruction. Its parsing briefly process is as follows:

  (1) Parser parse target DataLayout and Triple, the result is target data information.

  (2) Parser parse attribute group, the result is attribute information of module.

  (3) Parser parse local types in module, the result is registered type in module.

  (4) Parser parse global variables, the result is global variables symbol table.

  (5) Parser parse function which is mainly divided into arguments list and function body, the result is function structure in module instruction.

  (6) Parser parse metadata, the result is metadata map in module.

  Its pipeline process is as **Figure 64**:

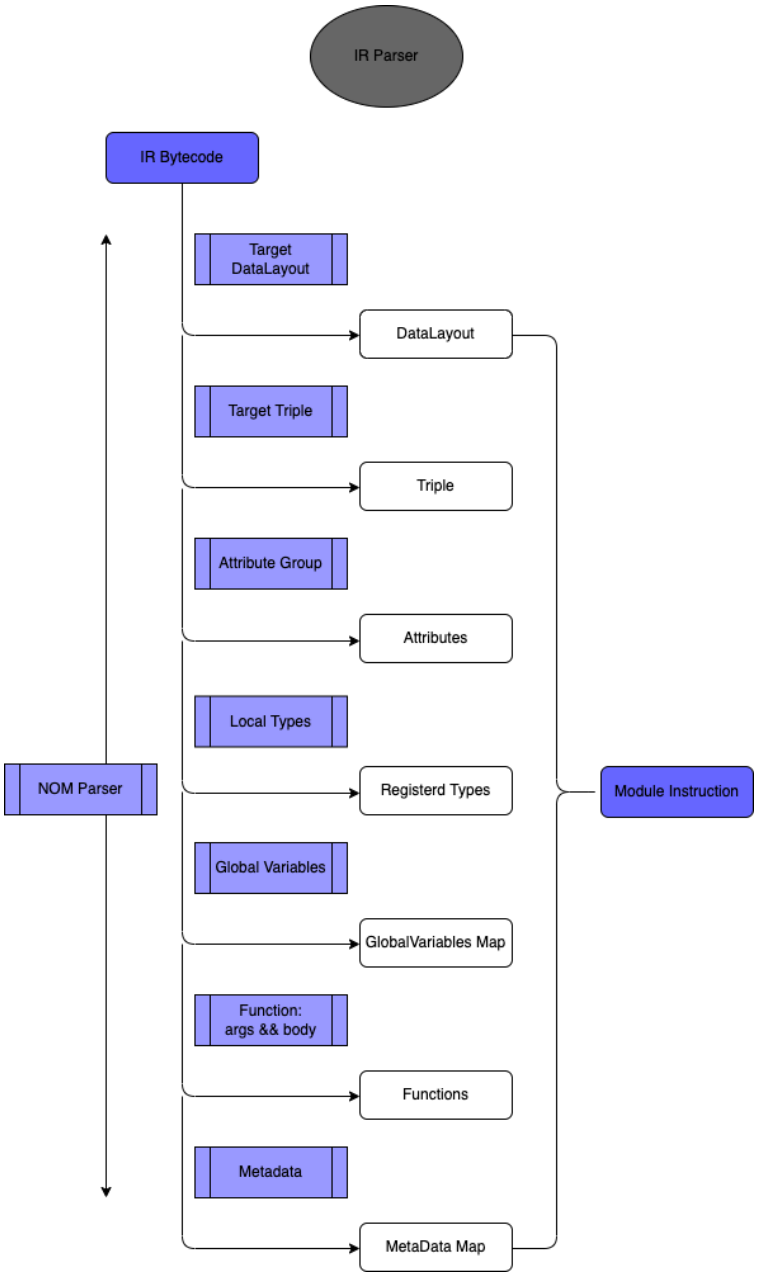


Figure 64: Ola-lang Backend Parser Pipeline

4.3.3 Optimizer: Optimization Passes on Parsed IR

Usually there are two kinds of compiler Optimization (Opt) passes, one is analysis passes and the other is transform passes. Currently our analysis pass is mainly Dominator Tree analysis pass, while transform passes contains Dead Code Elimination (DCE), Promote Memory to Register (Mem2Reg), Sparse Conditional Constant Propagation (SCCP).

Its pipeline process is as **Figure 65**:

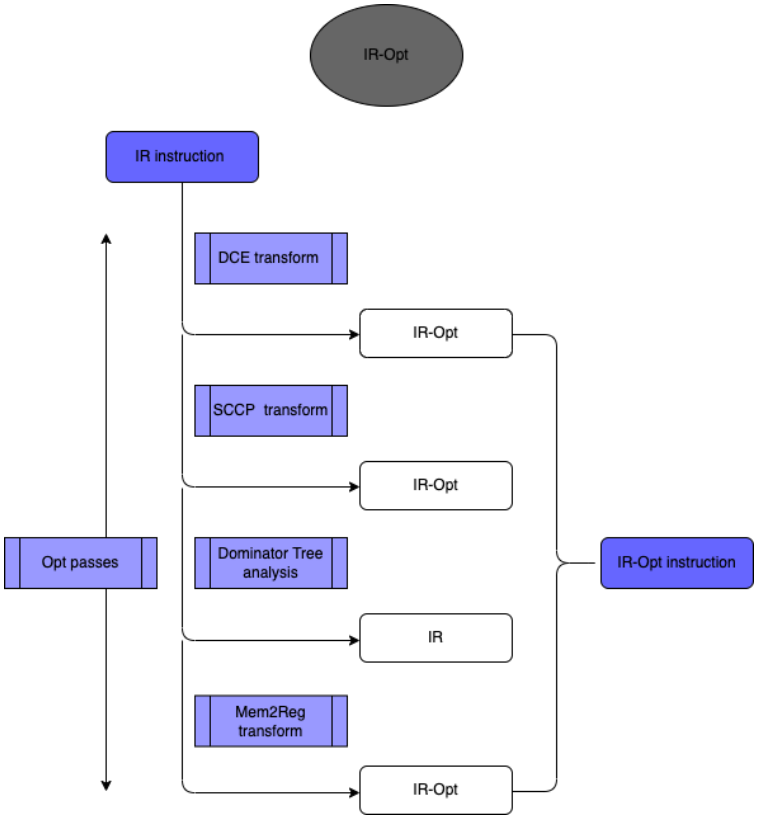


Figure 65: Ola-lang Backend IR Optimization

4.3.4 ISA Description: Register and Instruction

The register description is as follows:

Type	*Description*	*Register Group*
general registers	General used by program	`[r0-r8]`
return register	Return value for return to caller	`[r0]`
parameters rigsters	Parameters value for passing arguments	`[r1, r2, r3]`
temporary registers	Temporary alloc for local variables	`[r4, r5, r6, r7]`
stack pointer	Function's stack pointer	`[r8]`
special registers	Interact with vm: pc for program counter and psp for prophet pointer	`[pc, psp]`

Table 24: Register Description

Instruction description contains opcode and operands, note that the opcode here contains both register and

immediate number types in addition to the usual operator.

Opcode with register or immediate number is as follows:

```
ADDri,
ADDrr,
MULri,
MULrr,

EQri,
EQrr,
ASSERTri,
ASSERTrr,

MOVri,
MOVrr,

JMPi,
JMPr,
CJMPi,
CJMPr,
CALL,
RET,
END,

MLOADi,
MLOADr,
MSTOREi,
MSTOREr,

RANGECHECK,
AND,
OR,
XOR,
NOT,
NEQ,
GTE,

PROPHET,

Phi,
```

In particular, it is important to note that Operand data types are very different from the standard LLVM IR such as two register types, memory slot, block and label.

Operand data type in instructions selection is as follows:

```
Reg(Reg),
VReg(VReg),
Int8(i8),
Int32(i32),
Int64(i64),
MemStart,
```

```
    Slot(SlotId),
    Block(BasicBlockId),
    Label(String),
    GlobalAddress(String),
    None,
```

4.3.5 ABI Lower: Lowering Function Call

Ola Procedure Call Standard (OPCS) are as follows:

The stack initialization points to the first address of the frame stack after the `fp` register is loaded.

The address will be increased when the `call` instruction is executed later. When the `ret` instruction is executed, the `fp` register points to the address and falls back.

The Calling process is as follows:

- call label

  Caller use `call` instruction to call a callee as `call functionLabel`, and `fp` points to the new frame.

  The `pc` address returned by the callee is placed in `fp-1` which is detected by VM but not visible by the compiler backend.

  Its instructions pattern are as follows:

```
main:
.LBL0_0:
  ...
  call foo
  ...
foo:
.LBL1_0:
  ...
```

- function address

  The address pointed to by `fp` before the function call is placed in `fp-2` as `mstore [r8,-2] r8`.

  Its instructions pattern are as follows:

```
mstore [r8,-2] r8
```

- passing arguments

  Function parameter processing: the first three input parameters are placed in the three registers `r1`, `r2`, and `r3`. If there are more than 3 parameters, start with the fourth input parameter and descend accordingly in `fp-3`, `fp-4`, `....`

  Its instructions pattern are as follows:

```
mov r1 vreg1
mload r2 [r8,offset]
mov r3 vreg2
```

- local variables

  Local variables inside the function start at old `fp`, and their addresses are stored incrementally.

The single return value is stored in `r0`. If there are multi return values, it needs to be returned by a memory pointer that return the package data.

Instruction pattern for single return value is as follows:

```
mov r0 vreg3
```

The call stack frames layout is as follows 66:

For prophet library functions, its instructions pattern as:

```
.PROPHET{funcNum}_{prophetNum}:  // bind to prophet label
mov r0 psp  // interact with prophet read-only memory, get return value from prophet
     pointer
mload r0 [r0,0]  // used returned r0 as indexed addressing
```

First `.PROPHET` label binds to the prophet instance in assembly output. Then the program interacts with prophet read-only memory, get the return value from prophet pointer `[psp]` and write the result into `r0`. At last, we use `r0` as indexed addressing to load return values from prophet memory.

4.3.6 Instruction Selection: Match Pattern from IR to Target Instruction

It finds opcode and operands of instruction according to the given pattern, then select the best target single or multi instructions to match with it.

Several common patterns briefly are as follows table:

Pattern Type	*Description*
Alloca	params and vars allocation, selected to memmory operations
IntBinary	bianary operator
Load	memory load, containing base or offset
Store	memory store, containing base or offset
Call	A (caller) call B (callee)
Return	B (callee) return to A (caller)
Branch	branch control flow, selected to jump operations
Conditional Branch	conditional branch control flow, selected to compare and jump operations

Table 25: Instruction Selection Pattern

Let's take the Conditional Branch more specifically, for example, it's target instructions are as follows table 26:

4.3.7 Slot Elimination

This pass handles the stack slot for local variables.

Its pipeline is as follows:

```
VistModule
    | VisitFunction layout
        | VisitBasicBlock
            | Match inst's data operand is Slot type
                | workList: push inst
```

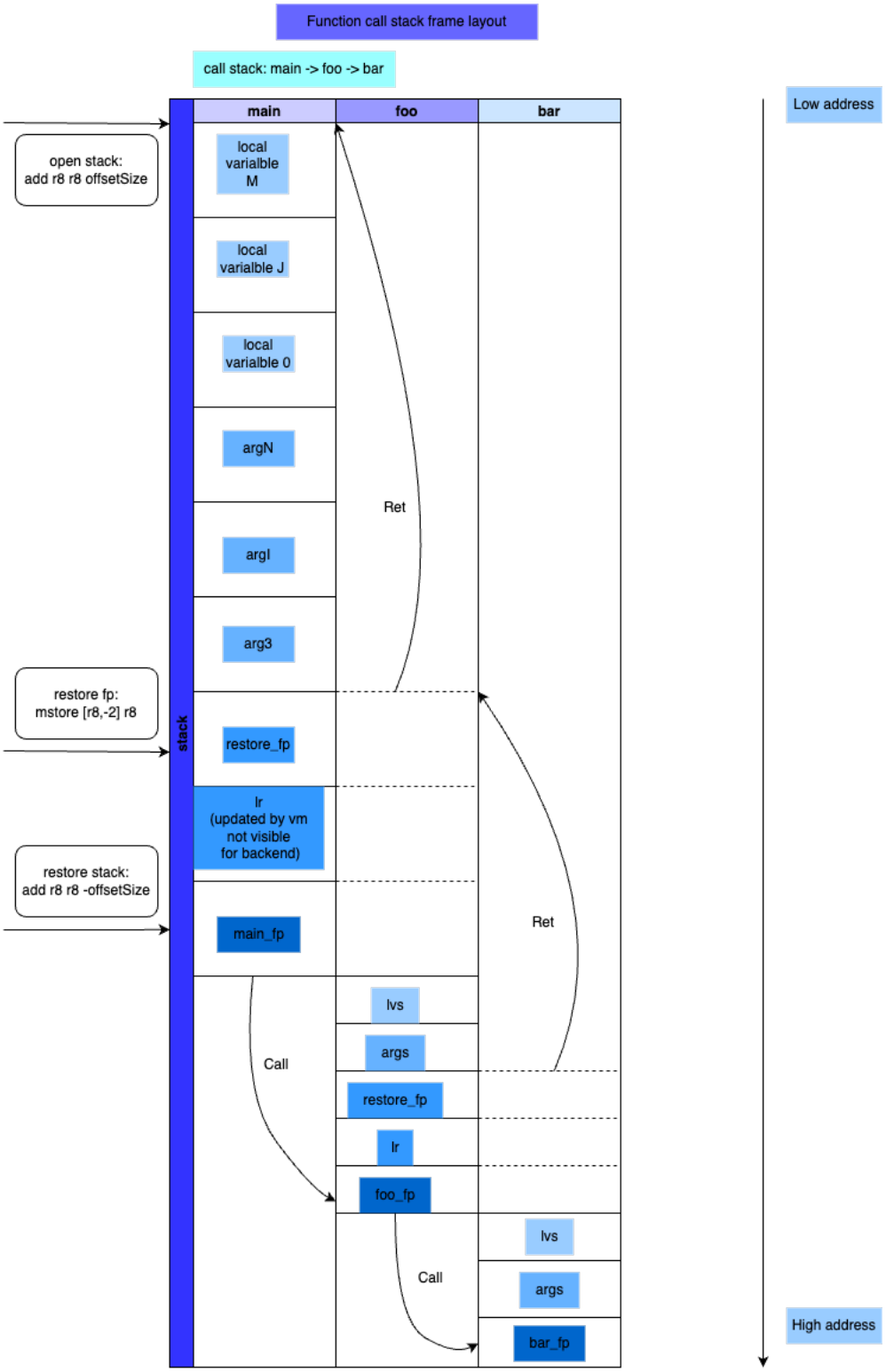


Figure 66: Ola-lang Function Call Stack Frames

Operator	*Reg and Imm*	*Reg and Reg*	*Cycles*
==	mov tmpReg imm eq tmpReg regA tmpReg cjmp tmpReg labelTrue	eq tmpReg regA regB cjmp tmpReg labelTrue	3inst + 2reg 2inst + 3reg
<	mov tmpReg1 imm gte tmpReg1 tmpReg1 regA neq tmpReg2 tmpReg1 regA and tmpReg2 tmpReg2 tmpReg1 cjmp tmpReg2 labelTrue	gte tmpReg1 regB regA neq tmpReg2 regA regB and tmpReg2 tmpReg1 tmpReg2 cjmp tmpReg2 labelTrue	5inst + 3reg 4inst + 4reg
<=	mov tmpReg imm gte tmpReg tmpReg regA cjmp tmpReg labelTrue	gte tmpReg regA regB cjmp tmpReg labelTrue	3inst + 2reg 2inst + 3reg
>	mov tmpReg1 imm gte tmpReg1 regA tmpReg1 neq tmpReg2 tmpReg1 regA and tmpReg2 tmpReg2 tmpReg1 cjmp tmpReg2 labelTrue	gte tmpReg1 regA regB neq tmpReg2 regA regB and tmpReg2 tmpReg1 tmpReg2 cjmp tmpReg2 labelTrue	5inst + 3reg 4inst + 4reg
>=	mov tmpReg imm gte tmpReg regA tmpReg cjmp tmpReg labelTrue	gte tmpReg regA regB cjmp tmpReg labelTrue	3inst + 2reg 2inst + 3reg
!=	mov tmpReg imm neq tmpReg regA tmpReg cjmp tmpReg labelTrue	neq tmpReg regA regB cjmp tmpReg labelTrue	3inst + 2reg 2inst + 3reg

Table 26: Conditional Branch Selection Pattern

```
    | Computer slot offset
    | foreach workList
        | fixup inst's operand with offset and size
```

4.3.8 Target Instruction Insertion: Prologue and Epilogue

When the processing is completed after the parameters of the function and the function body, as a part of the function it needs to do the corresponding stack space processing at the entrance and exit, respectively. That is, first calculate the stack size, then the stack is opened at the entrance, and the stack is recycled at the exit.

When there is no function call in the function body, entrance is one add instruction such as:

```
InstructionData {
    opcode: Opcode::ADDri,
    operands: vec![
        Operand::input_output(GR::R8.into()),
        Operand::input_output(GR::R8.into()),
        Operand::input(adj.into()),
    ],
}
```

When there is one or multi function call in the function body, entrance is one add inst and one mstore instruction such as:

```
InstructionData {
    opcode: Opcode::ADDri,
    operands: vec![
        Operand::input_output(GR::R8.into()),
        Operand::input_output(GR::R8.into()),
        Operand::input(adj.into()),
    ],
}

InstructionData {
    opcode: Opcode::MSTOREr,
    operands: vec![
        Operand::output(GR::R8.into()),
        Operand::input(GR::R8.into()),
    ],
}
```

While the exit is one sub operator which is expressed as add instruction such as:

```
InstructionData {
    opcode: Opcode::ADDri,
    operands: vec![
        Operand::output(GR::R8.into()),
        Operand::input_output(GR::R8.into()),
        Operand::input((-adj).into()),
    ],
}
```

4.3.9 Register Allocation and Coalescing

Register allocation use linear scan method, its briefly steps as follows:

(1) we analyze liveness in function, for input and output find live in and live out.

(2) we insert spill and reload code, push it to worklist.

(3) we rewrite the virtual register for the target register.

While the steps for register coalescing is as follows:

(1) We traverse the `movrr` target instructions at basic block of function on the module.

(2) If the two registers of operands are the same, then we push the instructions into the work list.

(3) We can then remove the instructions in the work list from the function.

4.3.10 Assembly Printing

Program basic format

The basic format of the Ola assembly language is as follows:

```
{symbol} {instruction | directive | pseudo-instruction} {; | // comment}
```

- Symbol indicates a symbol, which must start at the beginning of the line.
- Instruction indicates an instruction, it is usually preceded by two spaces.
- Directive indicates a pseudo operation.
- Pseudo instruction means a pseudo instruction.
- Directives, pseudo operations, and pseudo instruction helpers are all case-sensitive, but cannot be mixed.

Assembly instructions

For simplicity, pseudo operations and pseudo instructions like `.global` is not considered for now. Function entries that start with `funcName` and end with `:` are treated as label. For example, `main:` defines a label for a function named main.

Note: The symbols that usually start with `.` symbols that begin with `.` indicate pseudo directives or pseudo operations, such as different segments. Symbols ending with `:` indicate labels, such as function names and basic block numbers.

Instruction Format

The format of the internal assembly instruction is in the form of a three-address code:

```
    <opcode> <Rd> <Rn> <shifter_operand>
```

- `Opcode` indicates the instruction helper, usually the instruction helper defined by OlaVM.
- `Rd` indicates the instruction operation destination register, which is usually the register defined by OlaVM.
- `Rn` indicates the first source operand of the instruction, usually a register defined by OlaVM.
- `shifter_operand` indicates the instruction data processing operand, usually an immediate or register defined by OlaVM.

Memory layout

After the program is loaded, `pc` points to the zero address and the function stack frame is switched according to the hierarchy of function calls, and the memory address stack grows from a low address to high address. When prophets are present in program, an indexed addressing register is required to interact with the prophet memory.

4.4 Library Functions

4.4.1 Ola-lang Library Goal

- efficiency

  The goal of the Ola-lang high-level language library is to provide a set of high-level APIs that can be used to quickly develop applications. The library provides commonly used functions and modules, such as Ola Standard Library, integer type operations, math calculations, which can greatly improve the development efficiency of programmers.

  In addition, the library also will provide third-party modules that can be used to extend the functionality of the library.
- integer types functions

  Another goal is that since the native type of the ola language is the field type, operators of other types such as u32 and other integer types need to be simulated and converted through field-type operations to Implement their functions.

4.4.2 Language Features

For functionality, efficiency, and performance, our library functions usually combine Program, Builtin, and Prophet these three language features in their implementation.

The ola language uses standard syntax such as operations and control flow to describe the algorithmic process of library functions.

Ir generates the standard llvm ir, and also generates some builtin and prophet functions; these two types of functions generate calls and declarations but do not generate function bodies for function definitions.

The assembly code generates the standard olavm instructions; the builtin functions are converted to standard instructions such as rangecheck assert; the prophet functions to generate a label in the body of the program, while the function body, input, and output information are expanded to facilitate interpretation and execution in the VM.

- Program

  The ola language describes the algorithmic process, Ir generates llvm ir containing builtin and prophet libFunctions, and The assembly code generates the standard olavm instructions and prophets.

- Builtin

  The ola language is non-sensitive, Ir generates builtin libFunctions such as rangecheck and assert.

- Prophet

  The ola language is non-sensitive, Ir generates prophet libFunctions such as sqrt, div, and mod, and verifies the result. The assembly code generates prophet APIs for the VM such as function body, inputs, and outputs. and load psp to get the result.

Example of Assembly Pseudocode for a Typical Library Functions

Its pseudo-code format is as follows:

```
{
  "program": " // be directly executed by vm
    funcName:
    .LBL{funcNum}_{bbNum}:
      insts // such as mov add mload mstore call general instructions
      .PROPHET{funcNum}_{prophetNum}:  // bind to prophet label
      mov r0 psp  // interact with prophet read-only memory, get return value from
          prophet pointer
      mload r0 [r0,0]  // used returned r0 as indexed addressing
      rangecheck  // rangecheck builtin inst
      insts
      assert  // assert builtin inst
      insts
      return",
  "prophets": [  // be indirectly executed by the embedded interpreter in vm
    {
      "label": ".PROPHET{funcNum}_{prophetNum}", // according with prophet label in
          program
      "code": "%{\n    entry() {\n        cid.y = prophet_function_name(cid.x);\n
            }\n%}",  // prophet body, like solidity syntax
      "inputs": [  // prophet params
```

```
        "cid.x"
      ],
      "outputs": [  // prophet return values
        "cid.y"
      ]
    }
  ]
}
```

The comments in json asm describe the program, builtin, and prophet and their interaction in compiler output olavm assembly code.

- Program

  The program will be directly executed by VM after assembler encoding. It contains mainly instructions, rangecheck and assert builtin instructions, and call prophet displayed as prophet label format. It interacts with prophet read-only memory and gets a return value from the prophet pointer. Then use the return value register as indexed addressing to go on next program logic.

- Prophets

  The other part as the prophets list will be indirectly executed by an embedded interpreter in VM. The API field contains labels according to the prophet label in the program, prophet body which code is like solidity syntax, input as prophet params and outputs as prophet return values indirectly passed to the program.

4.4.3 U32 Library Functions List

Take u32 integer type libFunctions for example, its implementation in the language, the frontend and backend of the compiler is roughly as follows table 27:

5 ZK-ZKVM

A ZKVM (Section 2.2) which is a system that uses ZK technology to implement a verifiable circuit system for general computation. However, it has some issues where privacy is required, for example, quotation data of commercial competition, anonymous auction, etc. The generation of each program's proof executed in the ZKVM leaks the witness data, such as the name and function of the called contract, parameters, and so on.

To address this privacy concern, the ZK-ZKVM system has been developed. This system builds on top of ZKVM but adds an extra layer of privacy by ensuring that all witness data is private and does not reveal any information. This is achieved through the use of a key system consists of different permissioned private keys and a note mechanism similar to ZCash's UTXO model.

The basic principle of ZK-ZKVM is the use of different permissioned private keys in the key system. These keys are used to encrypt and decrypt the witness data, ensuring that it remains private and secure. Additionally, the note mechanism is used to further enhance the privacy of the system. Each note represents a certain amount of value that is spendable by the recipient, and is transmitted in ciphertext. It is impossible to deduce any information about the transaction from this ciphertext.

The importance of ZK-ZKVM lies in its ability to address the privacy concerns that exist in ZKVM. With ZK-ZKVM, users can conduct transactions on public blockchains with the assurance that their sensitive information is

Functions	*Notes*	*Ola-lang*	*IR Logic*	*Assembly Logic*
u32_add	add rangecheck	a+b	add opcode rangecheck builtin	add instruction rangecheck builtin instruction
u32_sub	sub rangecheck	a-b	sub opcode rangecheck builtin	add instruction rangecheck builtin instruction
u32_mul	mul rangecheck	a*b	mul opcode rangecheck builtin	mul instruction rangecheck builtin instruction
u32_div	rangecheck div prophet assert	a/b	q=prophet_div(a, b) assert builtin	instructions: passing params a,b code: prophet_div, prophet_mod instructions: get q from psp register assert instruction
u32_mod	rangecheck mod prophet assert	a%b	r=prophet_mod(a, b) assert builtin	instructions: passing params a,b code: prophet_div, prophet_mod instructions: get r from psp register assert instruction
u32_increment	add rangecheck	a++	add opcode rangecheck builtin	add instruction rangecheck builtin instruction
u32_decrement	sub rangecheck	a–	sub opcode rangecheck builtin	add instruction rangecheck builtin instruction
u32_or	conditional branch with or	a||b	separate to two brcond opcodes	cmp, cjmp and jmp instructions
u32_and	conditional branch with and	a&&b	separate to two brcond opcodes	cmp, cjmp and jmp instructions
u32_bitwise_or	bitwise or	a|b	or opcode	or builtin instruction
u32_bitwise_and	bitwise and	a&b	and opcode	and builtin instruction
u32_bitwise_xor	bitwise xor	aˆb	xor opcode	xor builtin instruction
u32_equal	eq compare	a==b	icmp eq opcode	eq instruction
u32_not_equal	ne compare	a!=b	icmp ne opcode	neq builtin instruction
u32_more	ugt compare	a>b	icmp ugt opcode	gte and neq builtin instructions
u32_more_equal	uge compare	a>=b	icmp uge opcode	gte builtin instructions
u32_less	ult compare	a<b	icmp ult opcode	gte and neq builtin instructions
u32_less_equal	ule compare	a<=b	icmp ule opcode	gte builtin instruction
u32_shift_left	i=2b̂ which call u32_power a=a*i	a«b	loop unroll related opcodes and mul opcode rangecheck builtin	jmp, cjmp and mul related opcodes rangecheck builtin instruction
u32_shift_right	i=2b̂ which call u32_power a=a/i which use div prophet	a»b	loop unroll related opcodes rangecheck builtin div prophet	jmp and cjmp related opcodes rangecheck builtin instruction div prophet assert instruction
u32_not	eq compare to zero	a	icmp eq opcode with the last operand is zero	eq instruction with the last operand is zero
u32_complement	u32_max sub a	-a	sub opcode with the first operand is u32_max	add instruction with the first operand is u32_max rangecheck builtin instruction
u32_power	forloop pattern rangecheck	a**b	loop unroll related opcodes rangecheck builtin	jmp and cjmp related opcodes rangecheck builtin instruction
u32_sqrt	rangecheck sqrt prophet assert	b=u32_sqrt(a)	rangecheck builtin sqrt prophet assert builtin	rangecheck builtin instruction sqrt prophet assert instruction

Table 27: U32 Library Functions List

secure and private. This makes it suitable for a wide range of use cases that require high levels of privacy, such as financial transactions or data sharing.

Furthermore, ZK-ZKVM also offers the same scalability benefits as ZKVM. By allowing for off-chain computation and verification, it reduces the burden on the main blockchain and increases transaction throughput. This makes it a more efficient and effective solution for blockchain scaling than traditional solutions.

In Section 5.1, we provide an in-depth explanation of the key system design in our ZK-ZKVM system. In the following section, Section 5.2, we explore the note design within our ZK-ZKVM system. Moving on, in Sections 5.3 and 5.4, we conduct a comparative analysis of two distinct approaches to proof generation. Finally, in Section 5.5, we present the fundamental framework of ZK-ZKVM.

5.1 Keys and Addresses

Encryption schemes and signature schemes in cryptography are the basis for achieving privacy. Encryption schemes include symmetric encryption and asymmetric encryption. In symmetric encryption, the sender and recipient first agree with a secret between them using key agreement scheme, and then derive a symmetric key from the secret for encryption and decryption. In asymmetric encryption, the sender encrypts the plaintext with the recipient's public key, and then the recipient decrypts the ciphertext with his own private key. The signature scheme is just the opposite, the sender signs plaintext with his own private key, and others can use the sender's public key to verify the signature.

A variety of encryption and signature schemes are used in Ola to protect privacy. At the same time, in order to enable the separation of spending and viewing permissions, some new keys are derived based on the elliptic curve algorithm, which constitutes Ola's key system.

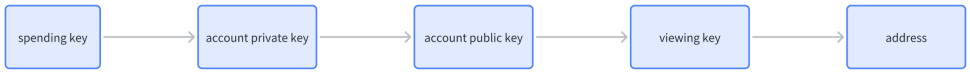


Figure 67: Key system of Ola

5.1.1 Spending Key

The spending key is the most important key in the key system. Similar to the private key in Bitcoin, whoever owns the spending key can spend and view the balance and transaction history in the address associated with this key. The spending key is essentially a 256-bit random number, which makes it as secure as Bitcoin in some respects, and is used to derive other keys.

5.1.2 Account Private Key

The account private key is derived from the associated spending key, and is also a private key like the spending key and can not be leaked to others. An account private key can be used to sign transactions and generate note commitments. It mainly consists of three parts:

- Random commitment key
  - A random number used in commitment scheme.
- Signature secret key

  - A key used in signature scheme.
- Random seed
  - A random number seed
  - The signature secret key and the random commitment key are all derived from the random seed.

The random commitment key is a scalar used in commitment schemes. The signature secret key is also a scalar used for the private key in signature schemes. The random seed is a random number generated by a random number generator, which is an element in a finite field and is 32 bytes in size.

5.1.3 Account Public Key

The account public key is derived from the associated account private key, which mainly consists of:

- Signature public key
- Signature public random key
- PRF secret key

The signature public key is a public key derived from a signature secret key and a group generator on an elliptic curve. It is essentially also a group element on the elliptic curve, and is used to verify the signature in a signature scheme.

$$\mathrm{Sig\_Pub\_Key} = G^{\mathrm{Sig\_Pri\_Key}}$$

The signature public random key is a random number derived from random commitment key for a signature scheme, and is essentially an element of the group defined on the elliptic curve.

$$\mathrm{Sig\_Pub\_Rand} = G^{\mathrm{rcm}}$$

The PRF secret key is a key used in the Pseudo Random Function to generate pseudo-random numbers.

$$\mathrm{PRF}_{sk}(x) = \mathrm{Hash}(sk \,||\, x)$$

5.1.4 Viewing Key

The viewing key is derived from the associated account private key, and each viewing key can generate one or more addresses. A viewing key can decrypt all transaction records associated with its address, that is, arbitrarily read transaction data under the address, so they can be used by regulatory departments to audit the historical transactions of an account. It can also be handed over to an delegate to generate zero-knowledge proofs. After the viewing key is leaked, all transaction data will be traced, but no assets will be lost (only the account private key can spend the user's assets). Viewing keys can only be issued to trusted institutions.

The viewing key is derived directly from the user's account private key

$$\mathrm{viewing\_key} = \mathrm{private\_key.sk} + \mathrm{private\_key.rcm} + \mathrm{public\_key.sk\_prf}$$

The main reason for creating a viewing key is to separate spending permissions and viewing permissions for user addresses. Users can share the viewing key with others to help themselves when necessary (such as asking delegates to generate zero-knowledge proofs for themselves), while keeping the spend key private.

5.1.5 Address

An address is generated by the associated account public key or viewing key to receive transfers. Addresses in Ola are either transparent or shielded (private). Transparent addresses are publicly visible on the blockchain, in the same way that Bitcoin addresses are viewable. Shielded addresses are invisible and transactions between shielded addresses do not reveal either address, the transaction value or the content of the encrypted field.

Zcash has defined a mechanism for generating hierarchical deterministic wallets in ZIP 32 [47], just like BIP 32 [9]. It can help users who want to generate multiple addresses per account. All derivations produce an opaque binary spending key, from which the keys and addresses are then derived.

- ZIP 32
  - Account 0
    - Diversified address 0
    - Diversified address 1
    - ...
  - Account 1
    - Diversified address 0
    - Diversified address 1
    - ...
  - ...

Ola will also design a similar mechanism in the future to implement a lightweight diversified address derivation scheme, which provides different payment addresses to different senders when users receive money to better protect the privacy of users.

5.2 Notes

5.2.1 Introduction

There are mainly two storage models in blockchain: the UTXO model used by Bitcoin and the account model use by Ethereum. UTXO is the abbreviation of Unspent Transaction Outputs, it mainly focuses on the equality of input and output. Transferring or updating contract status in the UTXO model performs consuming old notes and producing new notes. The account model, on the contrary, focuses on data such as the balance of the account stored in a location associated with the account. Transferring or updating the contract state can directly modify the data in the same location. Ola uses the UTXO model to implement a private account system, and also supports the account model to implement a public accounts system.

The main reasons why the private account system is implemented using the UTXO model rather than directly based on the account model are:

1. The account model updates directly update the state in the location of the account, which makes it easy to leak user identity information.
2. The account model supports privacy requires updating states of all users and contracts at the same time, which is less efficient.

The UTXO model can address the above two problems and enables efficient implementation of privacy features. In the private account system, a user's balance information and state data are stored in a data structure called Note

(refer to Zcash). A note mainly contains:

- Account public key
  - The public key of the owner of the note, that is, the holder of the private key corresponding to this public key can consume this note.
- Amount
  - The amount of tokens in the note, that is, the value that this note can spend.
- Payload
  - Data stored in this note.
- Birth predicate
  - Conditions that need to be met to create this note.
- Death predicate
  - Conditions that need to be met to consume this note.
- Random commitment key
  - Used to generate note commitment corresponding to this note.
- Nullifier key
  - Used to generate nullifier corresponding to this note.

The above note is represented by the vector group, which can be expressed as: (spend public key, amount, payload, birth, death, random commitment key, nullifier key).

5.2.2 Note Commitments and Nullifiers

When each note is created, a corresponding note commitment is generated, and the note commitment promises the validity of all parameters in the note.

$$\text{note\_commitment} = \text{Commit}(\text{note})$$

Note commitments and notes have a one-to-one bijection relationship, that is, there are no two different notes with the same note commitment, and vice versa. Unlike the account model, there is not a Merkle tree on the chain to store all states, but the ciphertext of notes is scattered among transactions in all blocks, and sequencers only maintain a notes commitment tree. When creating a note, users need to prove that the note commitment of this new note is on the current merkle tree through a zero-knowledge proof.

Every note has a nullifier, generated by the nullifier key and note commitment. In addition to the note commitment tree, sequencers also maintain a nullifier set. When a note is spent, the nullifier of the note needs to be revealed publicly. The node needs to check that the nullifier is not added to the nullifier set, otherwise it will cause the double spending problem.

$$\text{nullifier} = \text{Hash}(\text{nullifier\_key} \,||\, \text{note\_commitment})$$

5.2.3 Sending notes

In order to protect user privacy, the notes in the ledger can not be updated once they are generated. Instead, old notes need to be marked as spent by revealing their nullifiers, and produce new notes and append them to the ledger. After the user selects old notes and produces new notes, they need to encapsulate the note ciphertexts into a transaction

and submit it to a sequencer. There are some other fields in the transaction, such as an aggregated zero-knowledge proof. The process of sending notes is as follows:

1. Select the notes to be spent and the recipient's shielded address.
2. Decrypt the note ciphertexts to get note plaintexts of the selected old notes.
3. Get the random commitment keys and the nullifier keys of old notes from their plaintexts.
4. Generate note commitments of old notes using the random commitment keys.
5. Generate proofs of the existence of old notes on the merkle tree.
6. Generate nullifiers of old notes through their nullifier keys.
7. Generate an ephemeral key pair, which consists of an ephemeral private key and an ephemeral public key.
8. Use the spending public key and the ephemeral private key as inputs of the key agreement scheme to generate a shared secret.
9. Derive a symmetric key sym_key from the shared secret.
10. Construct new note plaintexts

$$np = (\text{spending\_public\_key}, \text{value}, \text{payload}, \text{random\_commitment\_key}, \text{nullifier\_key})$$

11. Encrypt note plaintexts using sym_key to get note ciphertexts.
12. Generate a statement depending on primary and auxiliary inputs.
13. Generate a zero-knowledge proof of the statement.
14. Assemble all old notes, new notes, the ephemeral public key and the proof into a transaction.
15. Encode the transaction and send it to the network.

5.2.4 Receiving notes

Recipients need to scan all blocks to receive notes sent to them. In each block, iterate over each transaction in it, and try to decrypt all note ciphertexts. If a note ciphertext can be decrypted by the recipient's viewing key, this means the note is sent to the recipient, and the recipient can then view the value and payload in the note and add it to his spent set. The process of scanning the blockchain is as follows:

1. Enumerate every transaction in the block.
2. Get the ephemeral public key in the transaction.
3. Use the spending private key and the ephemeral public key as inputs to the key agreement scheme to get a shared secret.
4. Derive a symmetric key sym_key from the shared secret.
5. Use sym_key to decrypt a note ciphertext in the transaction.
6. If decryption gets note plaintext np:

$$np = (\text{spending\_public\_key}, textvalue, \text{payload}, \text{random\_commitment\_key}, \text{nullifier\_key})$$

   (a) Generate the nullifier using nullifier_key and random commitment key.
   (b) Generate the note commitment using the random commitment key.
   (c) Check if the nullifier is already in the nullifier set to prevent double spending.
   (d) Check if the note commitment is in the note commitment tree.
   (e) Add the valid note to a spent set.
7. If it cannot be decrypted, try the next note ciphertext.

8. Return the spent set.

The user's balance and all on-chain state is the sum of the notes in the spent set.

5.3 Proof Generation

Sections 5.1 and 5.2 serve as core building blocks of our ZK-ZKVM system. In this section, we will explain our proof generation scheme. For a high-level overview of the system design, please refer to Section 5.5.

Proof generation is a critical component of the ZK-ZKVM platform, designed to facilitate private and efficient execution of smart contracts on the blockchain. The platform offers both public and private functions, with private functions executed and proven on the user-side, while public functions are executed and verified on blockchain nodes.

To enable this functionality, ZK-ZKVM relies on advanced cryptographic techniques such as the key system discussed in Section 5.1 and note design discussed in Section 5.2. These tools allow users to generate compact and efficient proofs that demonstrate their authorization to execute private functions without revealing any sensitive information about those functions.

The proof generation process involves several steps. First, in a single contract function call, there may be private and public functions. The user executes and generates a set of data (including private and public inputs) for the private function, along with a callback hook for the public function. They then use this data to generate a zero-knowledge proof that demonstrates their authorization to execute the function and validates the execution.

Next, the user submits this proof, along with their public inputs, to the blockchain node for verification. The node verifies the proof using ZK-ZKVM's public parameters (referred to as the verification key in ZK-SNARKs) to ensure that the private function is executed correctly.

Once the proof is verified, the network executes any relevant public functions on behalf of the user with the provided callback hook. Throughout this process, all sensitive information about the private function remains hidden from public view, ensuring strong privacy guarantees for all parties involved. Public state information, such as Note Commitments Tree and Nullifier Set, is stored on the node side and public functions are handled on the node as well.

Procedure. A Transaction is constructed as **Figure 68**:

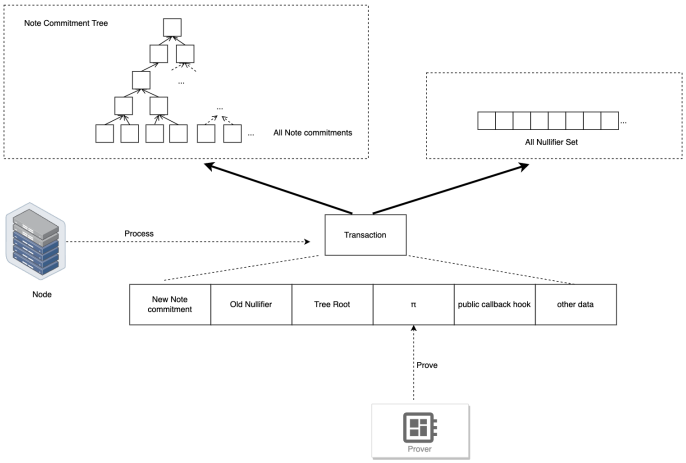


Figure 68: Transaction construction

As previously mentioned, proving the value of π is a crucial aspect of our transaction. Therefore, this section will

primarily focus on the generation of proofs. Below are several critical proof statements, which, although brief, hold great significance:

$\exists$ old Notes $(Note_{old,1}, Note_{old,2}, ..., Note_{old,n})$

old secret keys $(ask_{old,1}, ask_{old,2}, ..., ask_{old,n})$

new Notes $(Note_{new,1}, Note_{new,2}, ..., Note_{new,m})$

other auxiliary input aux

such that:

each old note $Note_{old,i}$:

- has a commitment that is in note commitments tree with tree root R_L.
- is owned by secret key $ask_{old,i}$
- has a nullifier nf_i

each new note $Note_{new,i}$ has commitment cm_j, and new commitment is unique.

When all the required parts match, including any other necessary components, the prover is able to produce a valid proof π for the transaction.

Here is a more detailed explanation proof generation phase:

Input:

- public parameters pp
- old
$$\begin{cases} \text{notes } [n_i]_1^n \\ \text{address secret keys } [ask_i]_1^n \end{cases}$$
- new
$$\begin{cases} \text{address public keys } [apk_j]_1^m \\ \text{note payload } [payload_i]_1^m \\ \text{note birth predicates } [\Phi_{b,j}]_1^m \\ \text{note death predicates } [\Phi_{d,j}]_1^m \end{cases}$$
- auxiliary predicate input aux
- transaction memorandum $memo$

Output: new notes $[n_j]_1^m$ and transaction tx that contains public function callback hooks $[hook_j]_1^m$ (if public functions are called)

1. For each $i \in \{1, ..., n\}$, process the i-th old note as follows:
   (a) Parse old note n_i as $(apk_i, payload_i, \rho, predicates(\Phi_{b,i}, \Phi_{d,i}), commitment\ cm_i)$
   (b) Compute **Note Commitments Tree(NCT) membership witness** for commitment: $W_{L,i} \leftarrow NCT.Prove(cm_i)$
   (c) Compute **nullifier number:** $nf_i \leftarrow PRF(\rho)$
2. For each $j \in \{1, ..., m\}$, construct the j-th new note as follows:
   (a) Compute **nullifier nonce:** $\rho_j := Hash(pp_{CRH}||nf_1||...||nf_n)$
   (b) Construct **new note:** $cm_j \leftarrow ConstructNote(apk_j, payload_j, \Phi_{b,j}, \Phi_{d,j}, \rho_j)$
3. Retrieve current Note Commitments Tree root: $R_L \leftarrow NCT.Root$.
4. Construct instance X_e for curcuit: $X_e := (R_L, [nf_i]_1^n, [cm_j]_1^m, memo, [hook_j]_1^m)$
5. Construct witness W_e for circuit: $W_e := ([n_i]_1^n, [W_{L,i}]_1^n, [ask_i]_1^n, [n_j]_1^m, aux)$

6. Generate **proof for:** $\pi \leftarrow Starky.Prove(pp, X_e, W_e)$
7. Construct **transaction:** $tx := ([nf_j]_1^n, [cm_j]_1^m, memo, R_L, \pi, [hook_j]_1^m)$
8. New notes and new constructed transaction: $([n_j]_1^m, tx)$

For additional information on the inner workings of Starky and how it produces proofs, please refer to this resource 6.1.

5.4 Delegable Proof Generation

The Section 5.3 scheme highlights a key challenge in generating cryptographic proofs for transactions in a blockchain network. As the complexity of the transaction and the number of calls involved increases, so does the cost of creating and including the proof in the transaction. This poses a significant problem for users, particularly those using weaker devices such as mobile phones or hardware tokens.

To address this issue, a solution called Delegable Proof Generation Scheme (DPGS) has been proposed. DPGS allows users to delegate the task of generating cryptographic proofs to a third party in a privacy-preserving manner, such as a server or a more powerful device, while still maintaining the security and validity of the transaction. This means that users can conduct complex transactions without incurring the high cost of generating proofs themselves.

The key system and note design used in DPGS ensures that transactions remain private and secure, even when proof generation is delegated to a third party. The system creates notes, which represent a certain unit of state, and each note is assigned a unique key. When a user wants to send a private transaction, they create a new note with a new stealth key and send it to the receiver. To ensure that the transaction is valid, a cryptographic proof is required to show that the note has not been previously spent or duplicated.

5.4.1 Current design

We compared the proof generation schemes of ZCash [46], ZEXE [10], VERI-ZEXE [44], Aleo [2], Aztec3 [7], and Efficient Private Delegation of ZK-SNARK Provers [17] as shown in **Table 28** below:

	Schemes
ZCash	User-side proof generation
ZEXE/VERI-ZEXE	Delegable DPC
Aleo	User-side proof generation for testnet3
Aztec	* Private function circuit proof generation at user-side, * Public function circuit proof generation at sequencer of rollup side
EPDZP	MPC for PIOP and KZG based proof system, core building block: Additive Secret Shares

Table 28: Current Proof Generation Schemes

As we can see from above figure, ZCash's proof generation scheme is user-side proof generation, which means ZCash has only one type of transaction, which is the transfer of ZEC. The transaction is not complex and the corresponding circuit size is not very large. Therefore, when it comes to proof generation, such as for shielded transactions, it can be done on the user side, for example, generating proofs within a wallet.

The delegable DPC protocol in ZEXE [10] allows for the delegation of computations while maintaining security. The protocol involves a delegator who sends input parameters to a delegatee, who then performs an offline computation and generates a proof of correctness using zero-knowledge proofs. The delegatee sends this proof along with the output of the computation back to the delegator, who can use it to generate a transaction that attests to the correctness of the computation. This transaction can be publicly verified by anyone without revealing any additional information about it.

To ensure privacy, ZEXE uses a combination of cryptographic techniques such as zero-knowledge proofs and randomizable signatures. The randomizable signature scheme is used to prevent linking across multiple signatures, which is important for maintaining security in the delegable DPC protocol that the delegatee can not impersonate the delegator, e.g., by producing further transactions that the delegator never authorized.

VERI-ZEXE and ZEXE use the same delegable DPC scheme, we will not repeat it. Aleo's current implementation (testnet3 [2]) is still user-side and uses SNARK-powered circuits to generate proofs, but does not yet implement delegable DPC.

Aztec3 [7] uses a recursive circuit to generate the final proof, and each contract function is a circuit when a proof needs to be generated. The recursion uses two system circuits, private kernel circuits and public kernel circuits. All private related function calls and their proofs must be entered into private kernel circuits to generate proofs, while all public related function calls and their proofs must be entered into public kernel circuits to generate proofs. In short, the proof of the private circuit is generated on the client side, while the proof of the public circuit is generated in Rollup's Squencer.

Efficient Private Delegation of zkSNARK Provers [17] uses MPC to do outsouring proveing. While it keeps witness private and proving efficient, it is designed for Polynomial IOPs and MPC friendly polynomial commitments proof system, while our ZK-ZKVM uses Starky as its internal proof system, which is not suitable for it.

5.4.2 Our work

Our design based on ZEXE, and the main procedure of delegable transactions as shown in the figure69 below:

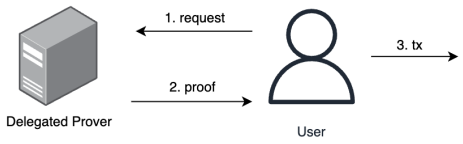


Figure 69: Delegable transactions

Although ZEXE does not allow the delegatee to impersonate the delegator, the witness remains visible to all participants on the public blockchain. This lack of privacy does not serve our purpose for ZK-ZKVM. To address this issue, we incorporated an external cryptographic primitive called the Secret Share Scheme.

We generate a shared secret by combining the delegatee's public key with the delegator's private key. We then use this shared key to symmetrically encrypt the witness on the delegator's side, and send the encrypted witness to the delegatee. The delegatee uses their private key and the delegator's public key to retrieve the shared key and decrypt the witness.

Once the witness data is decrypted, the subsequent process is identical to that of ZEXE.

Let's say User Alice is the delegator, Prover Bob is the delegatee. We briefly explain our delegable prove scheme:

- Alice generates shared key $key_{shared} = sk_{alice} \cdot pk_{bob}$
- Alice encrypt the witness, the encrypted witness is $witness_{enc} = enc(key_{shared}, witness)$
- Alice send the encrypted witness to Bob, while other participants on the public blockchain knows nothing.
- Bob generates shared key $key'_{shared} = sk_{bob} \cdot pk_{alice}$, which is same as key_{shared}
- Bob decrypt the encrypted witness, get raw witness, $witness = dec(key'_{shared}, witness_{enc})$
- Bob use decrypted witness generating proof π, send proof back to Alice.
- Alice uses her signature private key to construct a transaction within proofs, send it to the blockchain.

Our high-level description of our scheme as shown in the figure70 below:

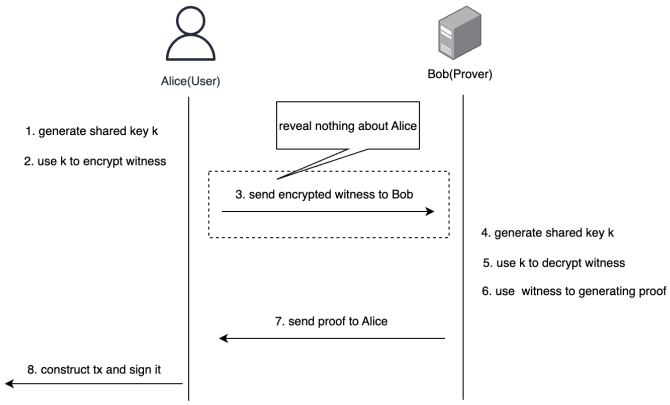


Figure 70: Our Proof Generation Scheme

5.5 Framework

Our high-level description of our framework as shown in the figure71 below:

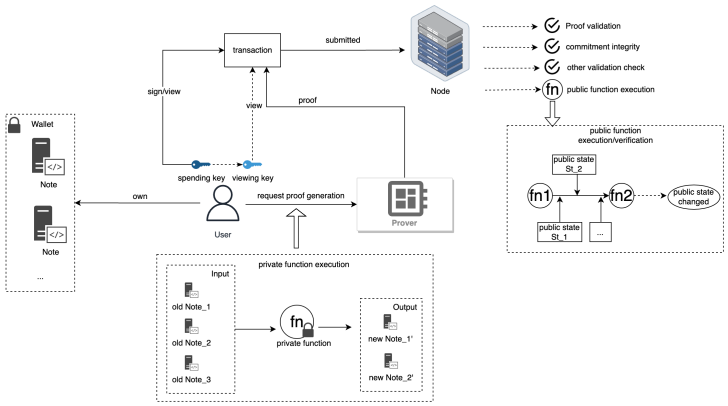


Figure 71: ZK-ZKVM Framework

And the building blocks of our ZK-ZKVM are:

Key System.

- The system will use a private key pair (sk, vk) to enable spending and viewing of funds.
- The spending key sk will be used to **sign** transactions such as transfer funds.
- The viewing key vk will allow the holder to view transactions without being able to spend the funds nor change the state of owner.
- The system will ensure that transactions can only be signed using the correct spending key sk.

Note Model.

- The system will use a UTXO (Unspent Transaction Output) model to track the ownership of state/funds.
- Each private transaction will create one or more new Notes, which can be spent in future transactions.
- The system will ensure that transactions can only spend Notes that are owned by the signer of the transaction.
- The Note reveals nothing but a meaningless string to other users.

Private/Public execution.

- There are two kinds of functions for users in our system: private or public functions.
- Private functions have return values (also known as notes), while public functions do not.
- Private functions will be executed and proved by the signer of the transaction, using Key System and Notes as core techniques described above.
- Private state can be decrypted by the owner's key, while it can only be spent by the spending key and is read-only when decrypted by the viewing key.
- Public functions have a callback hook, which is used in the Node's execution and verification phase.
- Private functions can call Public functions, but not the other way around.

Smart Contract. As we said in earlier Section 5, this is a ZK-ZKVM, which means it supports arbitrary smart contracts.

6 Algorithms

Ola uses ZKP to ensure the correct execution of the program and protect privacy. ZKP enables one party to prove to another party that a statement is true without revealing any private information.

At present, there are many implementations of the ZKP algorithms, such as Groth16, Halo2, Marlin, Plonky2, Starky, etc., and they have different technical characteristics. We thought about Ola's requirements for the ZKP algorithm. Firstly, as a ZKVM, we hope that the algorithm is more secure and more decentralized to use. Secondly, any smart contract can be more succinct to be constrained. Thirdly, proof generation and verification have good performance.

After careful consideration, we chose Starky as our proof generation algorithm and Plonky2 as our recursion circuit algorithm. What they have in common is that they are both based on STARK, which means neither require trusted third-party setup. Additionally, they both use the Goldilocks field to perform field operations. The difference is that Starky is AIR-based, which makes it more suitable for ZKVM, while Plonky2 is Plonkish-based, which makes it more suitable for specific computation, such as verification process of Starky.

Goldilocks field is a small prime field of order $p = 2^{64} - 2^{32} + 1$, and has the following properties:

1. Each u32 operation requires only one VM cycle using the Goldilocks field.
2. Field elements are within 64-bit, making operations on Goldilocks field fast on modern CPUs.
3. There is no overflow problem in the modular multiplication operation of two 32-bit integers.
4. It's very efficient to check whether four 16-bit values can construct a valid field element.

According to the test, a single multiplication operation on Goldilocks field takes only 2~3 cycles on modern CPUs. In addition, the delay of multiplication is also lower and resource utilization is higher than other fields like BN254.

6.1 Starky

At present, a zero-knowledge proof system consists of modules such as arithmetic, constraint system and polynomial commitment scheme. Ola's proof system is based on Starky. Starky uses the same finite field and hash functions as Plonky2 but without the computation-heavy arithmetizations. The arithmetic and constraint system in Starky is implemented using AIR. That is, the calculation process of the program is represented by several STARK tables, and there are predefined constraints between rows and columns for each table, and the same as between different tables. The polynomial commitment scheme in Starky is DEEP-FRI, a non-interactive protocol that proves a committed polynomial having a bounded degree.

6.1.1 STARK Tables Generation

The program in Ola will generate multiple original STARK tables after execution. There are several tables currently, such as CPU, Memory, Bitwise, RangeCheck, CMP, etc.

Different tables define a different number of columns for storing data such as original data, fixed data, auxiliary data. The values of some columns are determined when the program is executed, and the values of the remaining columns need to be calculated using the program execution results. When the cells of all STARK tables have values, we get the complete STARK tables, and then we can use FFT to interpolate each column in each STARK table into a polynomial. The operation process for each table is as follows:

1. Iterate each row of the table, calculating the values of all instruction-related columns;
2. Calculate the value of the permutation argument related columns;
3. Calculate the value of the lookup related columns;
4. Interpolate all columns into polynomials using FFT;
5. Returns the column polynomial array and public input.

6.1.2 Starky Proof Generation

Once we have the polynomial array and public input for each STARK table as described in **Figure 6.1.1**, we can start the proving process. Each STARK table in Starky must generate a ZK-STARK proof separately. The proof process for each table is as follows:

1. Compute Polynomials and Commitments.
   (a) Generate a Merkle commitment from polynomials of STARK table columns;
   (b) Divide the instance of the permutation argument into multiple batches, each batch generates a polynomial $Z(X)$;
   (c) Calculate $Z(X)$ polynomials for cross table lookups;

(d) Generate a Merkle commitment from all $Z(X)$ polynomials above.

2. Using trace polynomials and $Z(X)$ to generate quotient polynomials:

$$\text{quotient}(X) = \frac{\sum C(X) + (P\_Z(gX)\prod rhs - P\_Z(X)\prod lhs) + (CTL\_Z(gX) - CTL\_Z(X)\text{selector}(gX))}{X^n - 1}$$

3. Divide the quotient polynomials into small polynomials, the degrees of which are `quotient_degree_factor`;
4. Generate a Merkle commitment from quotient polynomials;
5. Calculate the three points of the challenge ζ, $g \cdot \zeta$ and the last point g^{n-1} in the group H;
6. Open the challenge points on the polynomial, respectively, and get openings;
   (a) Point ζ is opened on
      - Trace polynomials
      - Permutation and cross table lookups polynomials
      - Quotient polynomials

   (b) Point $g \cdot \zeta$ is opened on
      - Trace polynomials
      - Permutation and cross table lookups polynomials

   (c) Point g^{n-1} is opened only on
      - Cross table lookups polynomials

7. Generate the final polynomial;
   (a) Compress the polynomial opened at each point into one polynomial

$$F_i(X) = \sum_j \alpha^j \cdot f_{ij}, i \in [3]$$

   (b) Calculate the above $F_i(X)$ quotient, where $z \in \{\zeta, g \cdot \zeta, g^{n-1}\}$

$$Q_i(X) = \frac{F_i(X) - F_i(z)}{X - z}$$

   (c) Compress all the points to $Q_i(X)$ get the polynomial of the final FRI

$$\text{Final}(X) = \sum_{k=0}^{3} \alpha^k Q_i(X)$$

8. Generate a FRI proof for the final polynomial fri_proof;
9. Finally got the proof of a single stark, proof consists of
   - trace_cap
   - permutation_ctl_zs_cap
   - quotient_polys_cap
   - openings
   - fri_proof

10. The STARK proofs and public input of all STARK tables are converted into a final proof of the program, called a Starky proof.

There are now a total of 3 Merkle commitments

1. A Merkle commitment of trace polynomials;
2. A Merkle commitment of permutation_ctl_zs polynomials;
3. A Merkle commitment of quotient polynomials.

The order of a quotient polynomial is determined by the related constraints and trace degree:

$$\text{quotient\_degree\_factor} \cdot \text{degree}$$

The last point g^{n-1} is for cross table lookup checking. That is, the product of the last point value of multiple looking tables is equal to the last point value of the looked table.

6.1.3 Starky Proof Verification

Proving that an Ola program executes correctly requires verifying the starky proof generated in **Figure 6.1.2**. Verification mainly includes that each STARK proof is valid and that the cross table constraints between different STARK tables are correct. The process is as follows:

1. Get all challenges for generating STARK proofs;
2. Get the number of Z polynomials in permutation arguments;
3. Generate the data required to verify the Z polynomials of the cross table lookups;
   (a) The values of Z polynomials at ζ and $g \cdot \zeta$ at each table generated;
   (b) Generate the required β, γ used in generation of Z polynomials;
   (c) Columns and selectors used for cross table lookups in each STARK table.
4. Verify each stark proof individually.
   (a) Check the number of polynomials and cap_height of the stark proof are valid;
   (b) Get the value of 3 points (ζ, $g \cdot \zeta$ and g^{n-1}) opened on column polynomials;
   (c) Calculate the values of constraints, permutation arguments, and cross table lookup polynomials at the above three points;
   (d) Check $\text{quotient}(\zeta) \cdot Z\_H(\zeta) = \text{vanishing}(\zeta)$;
   (e) Check FRI proof is valid.

6.1.4 FRI

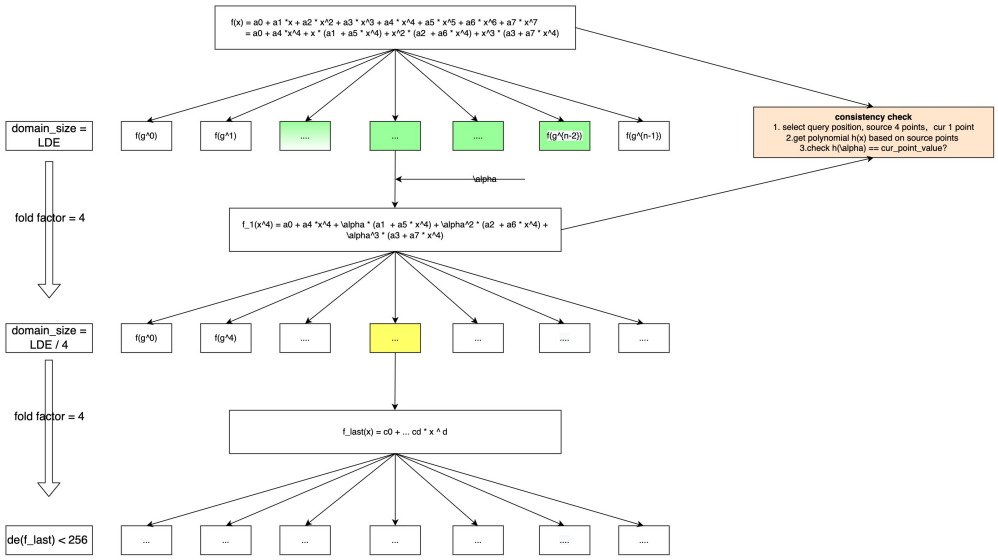


Figure 72: FRI protocol

FRI is a protocol that proves a committed polynomial having a bounded degree. Ola uses Deep-FRI for its polynomial commitment scheme. **Figure 72** simply expresses the calculation process of FRI, anyone can read DEEP-FRI [8] for more details.

6.1.5 Benchmark

Many optimizations have not yet been applied, and we expect to see more performance improvements as we devote more time to optimization. The benchmarks below should only be used as a rough guide to expected future performance.

In the benchmarks below, the VM executes the same Fibonacci calculator program for 2^{20} cycles at 100-bit target security level on an advanced 64-core CPU.

VM cycles	Execution time	Proving time	RAM consumption	Proof size
2^{18}	81.115 ms	2.932 s	5.6 GB	175 KB
2^{19}	159.80 ms	6.143 s	11.1 GB	181 KB
2^{20}	318.08 ms	12.688 s	23.2 GB	187 KB
2^{21}	627.38 ms	29.923 s	45.3 GB	195 KB
2^{22}	1240.4 ms	61.834 s	86.6 GB	208 KB
2^{23}	2453.8 ms	128.62 s	176 GB	216 KB

6.2 Plonky2

Plonky2 [32] = Plonkish contraints system + DEEP-FRI commitment + Goldilocks field. We plan to use Plonky2 as our recursive Layer2 and Layer3 (supported in the future) because it's more suitable for specific computation than Starky which is more suitable for ZKVM. It should be noted that the designs described in this are the original version of Plonky2, which will be more convenient to learn for the reader.

6.2.1 Circuit config

There are some types of configure in Plonky2, let us make some clarification on this first:

- `num_wires` is the maximum column number in one row;
- `routed_num_wires` is the maximum wires of gates in one row;
- `num_constants` is the maximum constants in one row;
- `use_base_arithmetic_gate` if the flag of base gate or extension gate;
- `security_bit` is the security level of prove system;
- `num_challenges` is the security parameter to get security_bit level;
- `zero_knowledge` is the flag of blind;
- `max_quotient_degree_factor` is the order parameter of constraint;
- `rate_bits` is the extension parameters for domain extension
- `cap_height` is the special parameter of Merkle tree;
- `proof_of_work_bit` is the difficulty parameter of query points;
- `reduction_strategy` is the parameters of FRI-reduction;

standard-recursion-config

```
pub fn standard_recursion_config() -> Self {
    Self {
        num_wires: 135,
        num_routed_wires: 80,
        num_constants: 2,
        use_base_arithmetic_gate: true,
        security_bits: 100,
        num_challenges: 2,
        zero_knowledge: false,
        max_quotient_degree_factor: 8,
        fri_config: FriConfig {
            rate_bits: 3,
            cap_height: 4,
            proof_of_work_bits: 16,
            reduction_strategy: FriReductionStrategy::ConstantArityBits(4, 5),
            num_query_rounds: 28,
        },
    }
}
```

standard-recursion-zk-config

```
pub fn standard_recursion_zk_config() -> Self {
    Self {
        num_wires: 135,
        num_routed_wires: 80,
        num_constants: 2,
        use_base_arithmetic_gate: true,
        security_bits: 100,
        num_challenges: 2,
        zero_knowledge: true,
        max_quotient_degree_factor: 8,
        fri_config: FriConfig {
            rate_bits: 3,
            cap_height: 4,
            proof_of_work_bits: 16,
            reduction_strategy: FriReductionStrategy::ConstantArityBits(4, 5),
            num_query_rounds: 28,
        },
    }
}
```

standard-ecc-config

```
pub fn standard_ecc_config() -> Self {
    Self {
        num_wires: 136,
        num_routed_wires: 80,
```

```
        num_constants: 2,
        use_base_arithmetic_gate: true,
        security_bits: 100,
        num_challenges: 2,
        zero_knowledge: false,
        max_quotient_degree_factor: 8,
        fri_config: FriConfig {
            rate_bits: 3,
            cap_height: 4,
            proof_of_work_bits: 16,
            reduction_strategy: FriReductionStrategy::ConstantArityBits(4, 5),
            num_query_rounds: 28,
        },
    }
}
```

wide-ecc-config

```
pub fn wide_ecc_config() -> Self {
    Self {
        num_wires: 234,
        num_routed_wires: 80,
        num_constants: 2,
        use_base_arithmetic_gate: true,
        security_bits: 100,
        num_challenges: 2,
        zero_knowledge: false,
        max_quotient_degree_factor: 8,
        fri_config: FriConfig {
            rate_bits: 3,
            cap_height: 4,
            proof_of_work_bits: 16,
            reduction_strategy: FriReductionStrategy::ConstantArityBits(4, 5),
            num_query_rounds: 28,
        },
    }
}
```

6.2.2 Gates

Each gate is a row with 135 columns. As different custom gate has different complexity, for some complex gates 135 columns may only constrain one operation while for other simple gates, 135 columns may constrain several operations. The index of operation in a row is called a slot index.

Steps to using a custom gate:

- Ensure the type of gate;
- Ensure the row and slot location of the gate;
- Ensure the wires layout of the gate;

- Ensure the consistency between public data and wires of gate;
- Ensure the row location of public data;
- Finalize the trace table;
- Constraint for trace table;
- Generate proof.

For the convenience of description, the trace tables in this paper only show one operation which is slot 0.

6.2.2.1 arithmetic_base

ArithmeticGate is a gate that can perform a multiply-add with constants, i.e.

$$\text{res} = \text{cons\_0} \times \text{mul\_0} \times \text{mul\_1} + \text{cons\_1} \times \text{add}.$$

The structure of the gate is shown in **Figure 73**.

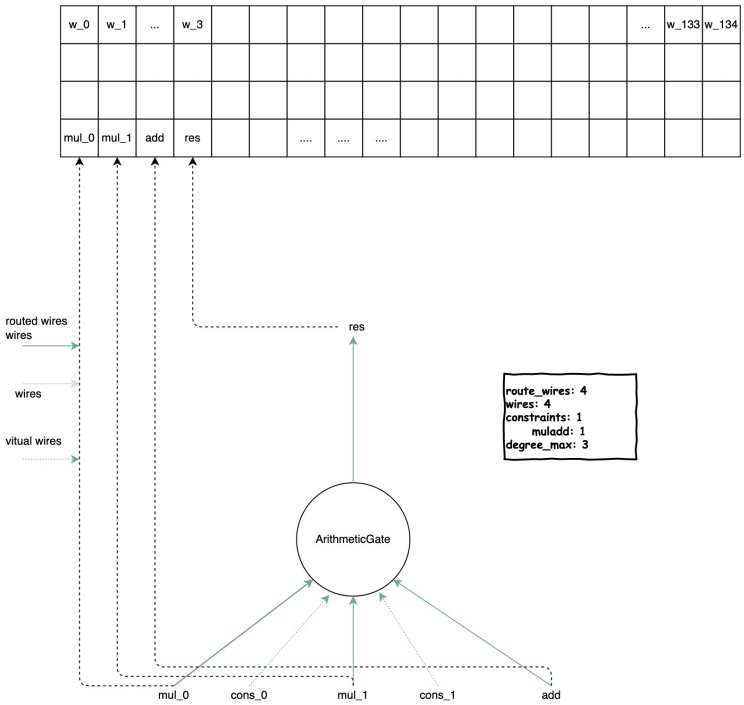


Figure 73: ArithmeticGate

There's only one constraint per operation, and the degree is 3.

6.2.2.2 arithmetic_extension

To understand the design principle of this Gate, we must first understand field extensions.

Taking Plonky2's Goldilocks field as an example, we give the extension field elements under quadratic, quartic, and quintic extensions respectively in **Figure 74**.

It is easy to see that for quadratic extension field, the elements on its domain take the form $a + b\sqrt{7},\ a, b \in \mathbb{F}_p$. It can be seen that on the quadratic extension field, there are p^2 elements and the original field is a subset of the quadratic extension field.

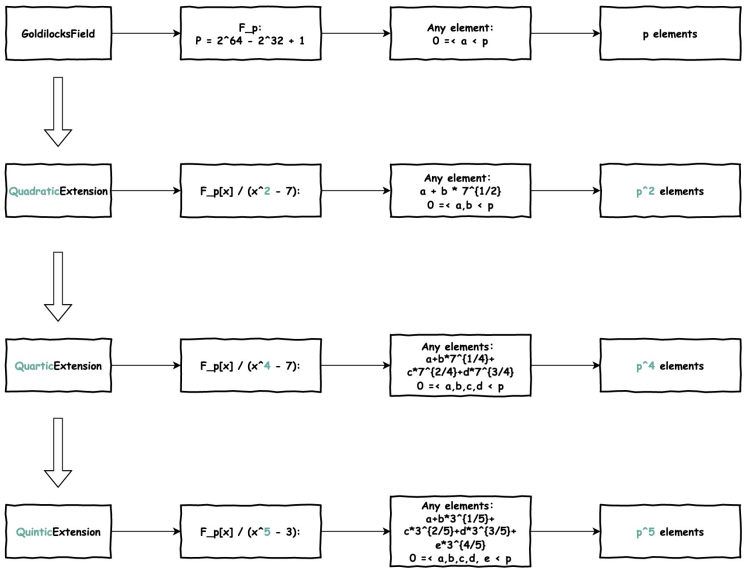


Figure 74: Goldilocks Field Extension

ArithmeticExtensionGate is also a gate that can perform a weighted multiply-add, i.e.

$$\text{res} = \text{cons\_0} \times \text{mul\_0} \times \text{mul\_1} + \text{cons\_1} \times \text{add}$$

The elements of the QuadraticExtension Field are represented in the form $[a, b]$, so the Gate design for arithmetic_extension has the following form:

The structure of the gate is shown in **Figure 75**. There's only one constraint per operation, and the degree is 3.

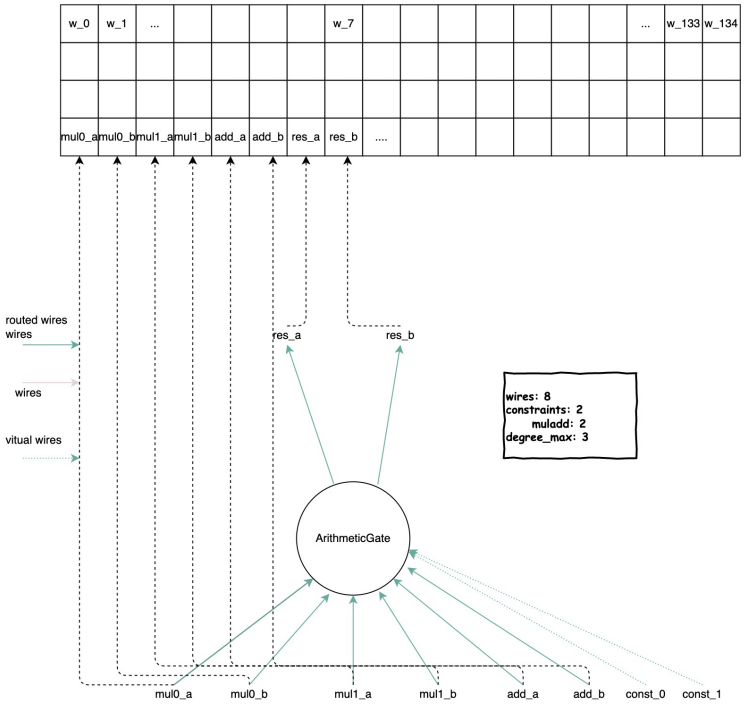


Figure 75: ArithmeticExtensionGate

6.2.2.3 base_sum

BaseSumGate is used to constrain the input to be composed of limbs that are arranged in little-endian. There are two kinds of constraints:

For each limb, limb is in range $[0, \text{base})$:

$$\sum_{i=0}^{\text{base}} (\text{limb}_i - i) = 0.$$

Input is composed of limbs:

$$\text{input} = \sum_{i=0}^{n-1} \text{limb}_{n-1-i} \times \text{base}^i.$$

The structure of the gate is shown in **Figure 76**.

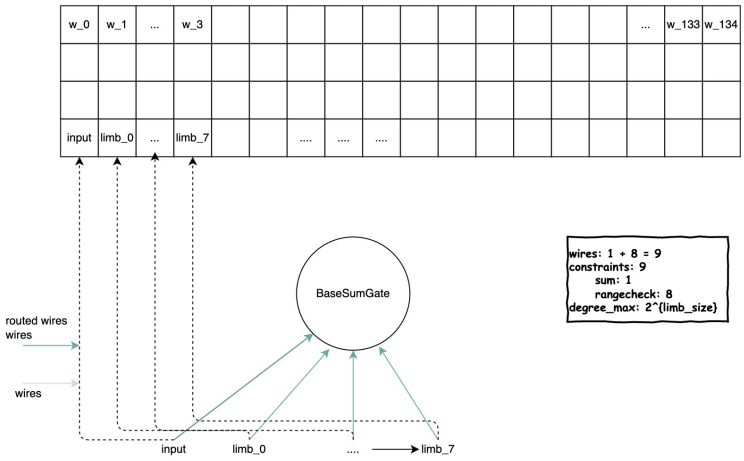


Figure 76: BaseSumGate

There's 1 constraint for the sum check and 8 constraints for the limbs' range check. The degree of the gate is $2^{\text{limb\_size}}$.

6.2.2.4 exponentiation

ExponentiationGate is a gate for raising a value to a power. The trace table contains the base, bits of the exponent, output, and intermediate value of the bits.

Take $A^{21} = A^{10101_b}$ for example to describe intermediate value, the bits are [1, 0, 1, 0, 1].

1. Current bit = 1, we start from 1, and times A^{bit} we get A
2. Current bit = 0,
   - Square prev_intermediate_value $A^{1_b<<1} = A^{10_b}$
   - Then times A^{bit} we get $A^{10_b} \times A^0 = A^{10_b}$
3. Current bit = 1,
   - Square prev_intermediate_value $A^{10_b<<1} = A^{100_b}$
   - Then times A^{bit} we get $A^{100_b} \times A = A^{101_b}$
4. Current bit = 0,
   - Square prev_intermediate_value $A^{101_b<<1} = A^{1010_b}$
   - Then times A^{bit} we get $A^{1010_b} \times 1 = A^{1010_b}$

5. Current bit = 1,
    - Square prev_intermediate_value $A^{1010_b << 1} = A^{10100_b}$
    - Then times A^{bit} we get $A^{10100_b} \times A = A^{10101_b}$

And we get the last intermediate value A^{10101_b} which should be equal to the output.

Let's take another example of a specific number $2^{13} = 2^{1101_b}$, and have a look at the trace cell:

base	b_0	b_1	b_2	b_3	output	inter_0	inter_1	inter_2	inter_3
2	1	0	1	1	8192	2	8	64	8192

The structure of the gate is shown in **Figure 77**.

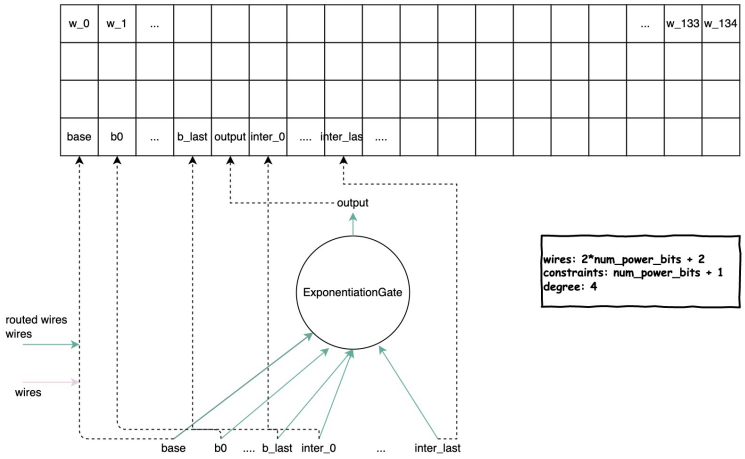


Figure 77: ExponentiationGate

Each step result is constrainted with intermediate values, and output is constrained with the final intermediated value, a total of $bits + 1$ constraints. The degree of the gate is 4, which is determined by the intermediate calculation:

```
let computed_intermediate_value =
            prev_intermediate_value * (cur_bit * base + not_cur_bit);
```

6.2.2.5 Poseidon

Poseidon is a hash function designed for the Zero-Knowledge proof system. Its calculation process is rough as follows:

Each round function of Poseidon permutation consists of the following three components.

1. ARC(.): AddRoundConstants
2. S: SubWords
3. M(.): MixLayer

The Trace of the gate is like this:

- input: components of the input, 12 elements.
- output: components of the output, 12 elements.
- swap: 0 or 1, Indicates whether the first four elements of the input are swapped with the last four elements.
- delta: used when swap is 1, $\text{delta}_i = \text{swap} \times (\text{input}_{\text{rhs}} - \text{input}_{\text{lhs}})$.
- green region: full rounds, ri_1 ri_11 is the state of each round.

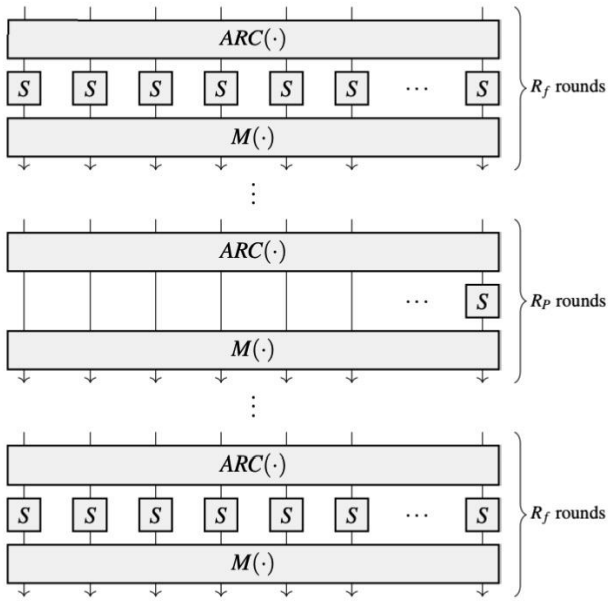


Figure 78: Construction of poseidon

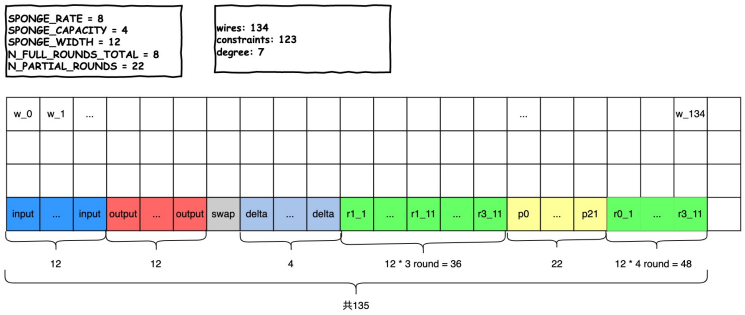


Figure 79: PoseidonGate

- yellow region: partial rounds, each element is state[0] of each round.

Calculation process and related constraints:

- Assert that swap is binary. (1 constraint)

```
constraints.push(swap * (swap - F::Extension::ONE))
```

- Assert $\text{delta}_i = \text{swap} \times (\text{rhs} - \text{lhs})$. (4 constraints)

```
for i in 0..4 {
    ....
    constraints.push(swap * (input_rhs - input_lhs) - delta_i);
}
```

- Initialize state: when swap=0, state=input; when swap=1, state is swapped input:

i0+d0	i1+d1	i2+d2	i3+d3	i4-d0	i5-d1	i6-d2	i7-d3	i8	i9	i10	i11

Table 29: Poseidon State Init

```
for i in 0..4 {
    ...
    state[i] = vars.local_wires[input_lhs] + delta_i;
    state[i + 4] = vars.local_wires[input_rhs] - delta_i;
}
for i in 8..SPONGE_WIDTH {
    state[i] = vars.local_wires[Self::wire_input(i)];
}
```

- Begin first full rounds calculation, for each round r (which is 0–3):
  - Perform ARC: Add each element of state to the pre-generated value at a particular position in the array.

```
for i in 0..WIDTH {
    state[i] += F::from_canonical_u64(ALL_ROUND_CONSTANTS[i + WIDTH * round_ctr]);
}
```

  - Except for r=0, constrain each element of the state calculated in the previous round (the green part of the first slice of the figure). (12 constraints per round, a total of 36 constraints)
  - Perform SubWords: Turn state element by element x into $x \mapsto x^7$
  - Perform MixLayer: Each element of the state is updated according to itself and a pre-generated array.

```
// r is the index of state elements here.
let mut res = F::ZERO;
    for i in 0..WIDTH {
    res += v[(i + r) % WIDTH] * F::from_canonical_u64(Self::MDS_MATRIX_CIRC[i]);
}
res += v[r] * F::from_canonical_u64(Self::MDS_MATRIX_DIAG[r]);
```

- Perform partial rounds:
  - Perform ARC

```
for i in 0..12 {
    if i < WIDTH {
        state[i] += F::from_canonical_u64(Self::FAST_PARTIAL_FIRST_ROUND_CONSTANT[
            i]);
    }
}
```

  - Processing of state with 11×11 MDS (maximum distance separable) matrix.

```
result[0] = state[0];
for r in 1..12 {
    if r < WIDTH {
        for c in 1..12 {
            if c < WIDTH {
                let t = F::from_canonical_u64(
                    Self::FAST_PARTIAL_ROUND_INITIAL_MATRIX[r - 1][c - 1],
                );
                result[c] += state[r] * t;
            }
        }
```

```
    }
}
result
```

  - Perform 22 round sbox, for the first 21 rounds (r = 0–21):
    * Take sbox_in(yellow elements in the figure), and constrains state[0]=sbox_in – 21 rounds totally 21 constraints.
    * `state[0] = state[0]^7`
    * `state[0] += FAST_PARTIAL_ROUND_CONSTANTS[r]`
    * Perform mds to state.
  - For the 22nd round:
    * `state[0] = sbox_in` (i constraint)
    * `state[0] = state[0]^7`
    * Perform mds to state.
- Perform second round "Full rounds", same with the first round. (the green part of the second slice of the figure). (12 constraints, 4 rounds totally of 48 constraints)
- Asserts computed result equals output. (12 constraints)

```
for i in 0..SPONGE_WIDTH {
    constraints.push(state[i] - vars.local_wires[Self::wire_output(i)]);
}
```

The constraints of this gate in total is 123, the degree is 7 (when performing s-box, making state$[i] \mapsto$ state$[i]^7$).

6.2.2.6 poseidon_mds

This gate is used for constraining outputs of Poseidon mds.

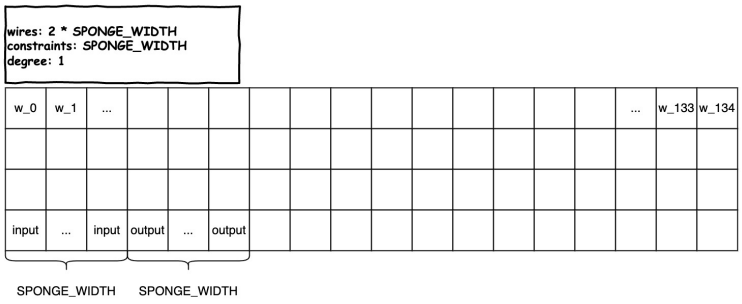


Figure 80: PoseidonMdsGate

computed_output is calculated from input, and constrained with output element by element, a total of 12 constraints (degree 1).

```
let inputs: [_; SPONGE_WIDTH] = (0..SPONGE_WIDTH)
    .map(|i| vars.get_local_ext_algebra(Self::wires_input(i)))
    .collect::<Vec<_>>()
    .try_into()
    .unwrap();
let computed_outputs = Self::mds_layer_algebra(&inputs);
```

```
(0..SPONGE_WIDTH)
    .map(|i| vars.get_local_ext_algebra(Self::wires_output(i)))
    .zip(computed_outputs)
    .flat_map(|(out, computed_out)| (out - computed_out).to_basefield_array())
    .collect()
```

6.2.2.7 random_access

RandomAccessGate is used to verify that an element matches a value in the list.

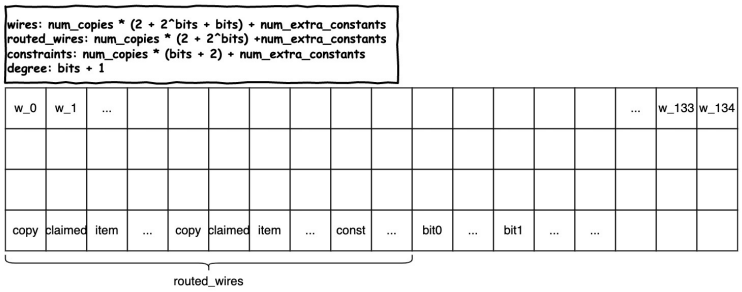


Figure 81: RandomAccessGate

- item: list items.
- copy: index of the target element in the list
- claimed: target element
- bit_i: bits for the i-th copy

For each copy:

- Constrain bits are 0 or 1. – bits constraints for each copy, A total of num_copied*bits constraints.

```
for &b in &bits {
    constraints.push(builder.mul_sub_extension(b, b, b));
}
```

- Constraint copy consists of bits. – 1 constraint for each copy, A total of num_copied constraints.

```
let reconstructed_index = bits
    .iter()
    .rev()
    .fold(zero, |acc, &b| builder.mul_add_extension(acc, two, b));
constraints.push(builder.sub_extension(reconstructed_index, access_index));
```

- For each bit, reconstruct items with a 2-elements-tuple, select the first element when the bit is 0, and select the second when the bit is 1. After the bits round, only one element remains, that is, the index element corresponding to bits, constraint it with claimed.

```
for b in bits {
    list_items = list_items
        .iter()
        .tuples()
        .map(|(&x, &y)| builder.select_ext_generalized(b, y, x))
```

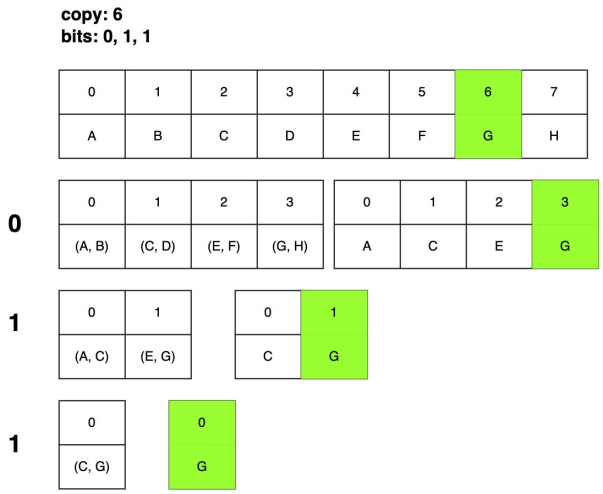


Figure 82: Random Access Example

```
        .collect()
}
// Check that the one remaining element after the folding is the claimed element.
debug_assert_eq!(list_items.len(), 1);
constraints.push(builder.sub_extension(list_items[0], claimed_element));
```

Finally, the constant is constrained – A total of num_extra_constraints constraints.

In summary, there're num_copies $\times$ (bits $+ 2$) $+$ num_extra_constants constraints. The degree is bits $+ 1$ which happens when repeatedly folding the list.

6.2.2.8 reducing

ReducingGate is used for computes output $=$ old_acc $+ \sum C_i \cdot \alpha^i$ in the base field.

The trace structure for this gate is like this:

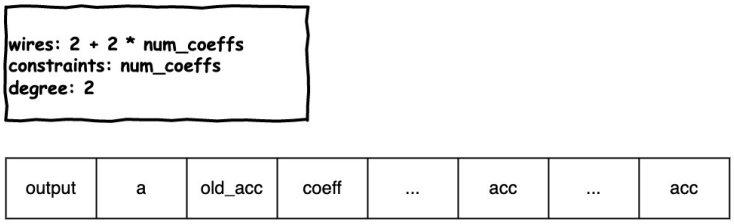


Figure 83: ReducingGate

The constraint flow is relatively intuitive, initializing acc to old_acc, then cumulative computation of polynomials by coeff in turn, and constraining the intermediate results of each step with acc.

```
for i in 0..self.num_coeffs {
    let coeff = builder.convert_to_ext_algebra(coeffs[i]);
    let mut tmp = builder.mul_add_ext_algebra(acc, alpha, coeff);
    tmp = builder.sub_ext_algebra(tmp, accs[i]);
    constraints.push(tmp);
```

```
    acc = accs[i];
}
```

The number of constraints is equal to the number of coefficients. The polynomial degree is 2 which happens when calculating $\text{coeff}_i * \alpha$, sum up does not increase degree.

ReducingExtensionGate is like ReducingGate, just computations happen in the extension field, and constrain logic is all the same.

6.2.2.9 high_degree_interpolation

InterpolationGate is used for the interpolation of a polynomial, whose points are a (base field) coset of the multiplicative subgroup with the given size, and whose values are extension field elements. As for HighDegreeInterpolationGate, allows constraints of variable degree, up to `1 << subgroup_bits`. The higher degree is a tradeoff for fewer gates (than LowDegreeInterpolationGate).

HighDegreeInterpolationGate trace is shown in **Figure 84**.

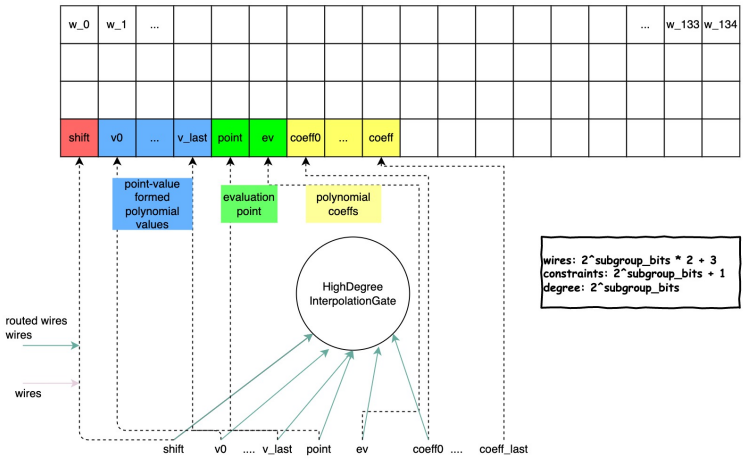


Figure 84: HighDegreeInterpolationGate

Constraints as follows:

- Bring each point (from the point-value pairs) into the coefficient polynomial to compute the computed_value and compare the constraint with the value (from the point-value pairs). – A total of $2^{\text{subgroup\_bits}}$ constraints.

```
for (i, point) in coset.into_iter().enumerate() {
    let value = vars.get_local_ext_algebra(self.wires_value(i));
    let computed_value = interpolant.eval_base(point);
    constraints.extend((value - computed_value).to_basefield_array());
}
```

  coset: $[sg, sg^2, ..., sg^{2^{\text{subgroup\_bits}}}]$, $s = \text{shift}$.

- Evaluate the coefficient-form polynomial at the evaluation point and constrain it. – 1 constraint.

The degree of this gate equals the number of points (num_points): max point power is num_points $-$ 1, and multiplication by coefficient adds 1 degree.

The number of constraints equals num_points $+$ 1: num_points for consistency between the coefficients and the point-value pairs, 1 constraint for the evaluation value.

6.2.2.10 low_degree_interpolation

InterpolationGate is used for the interpolation of a polynomial, whose points are a (base field) coset of the multiplicative subgroup with the given size, and whose values are extension field elements. As for LowDegreeInterpolationGate, all constraints are degree <= 2, low degree is a tradeoff for more gates (than HighDegreeInterpolationGate).

LowDegreeInterpolationGate trace is shown in **Figure 85**.

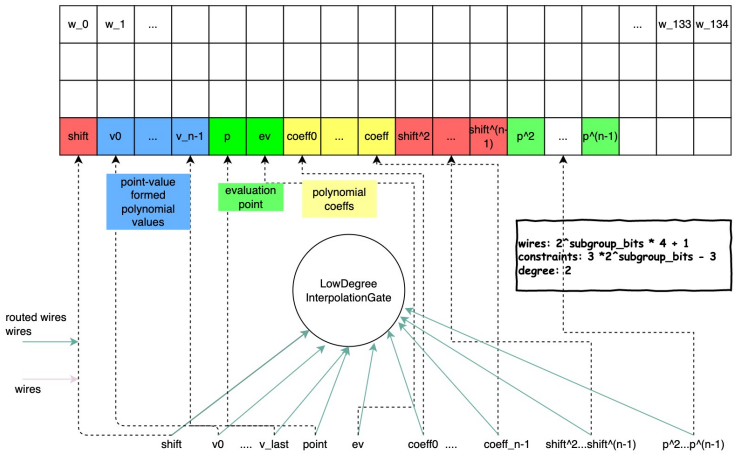


Figure 85: LowDegreeInterpolationGate

Constraints as follows:

- Constrain powers of shift, from shift^2 to shift^{n-1}, a total of $2^{\text{subgroup\_bits}} - 2$ constraints.

```
for i in 1..self.num_points() - 1 {
    constraints.push(powers_shift[i - 1] * shift - powers_shift[i]);
}
```

- Bring each point (from the point-value pairs) into the coefficient polynomial to compute the computed_value and compare the constraint with the value (from the point-value pairs). – A total of $2^{\text{subgroup\_bits}}$ constraints.
- Constrain powers of evaluation point. – A total of $2^{\text{subgroup\_bits}} - 2$ constraints.
- Evaluate the coefficient-form polynomial at the evaluation point and constrain it. – 1 constraint.

As can be seen from the above constraint description, the number of constraints is $3 \cdot 2^{\text{subgroup\_bits}} - 3$, degree of LowDegreeInterpolationGate is 2.

6.2.2.11 add_many_u32

U32AddManyGate is a gate to perform addition on num_addends different 32-bit values, plus a small carry. There can be up to 16 operations per gate.

The gate structure is like **Figure 86**.

Constraints for each operation:

- Constrain the result of addends summation equals results of res and carry_out calculation. – 1 constraint with degree 1

```
let base = F::Extension::from_canonical_u64(1 << 32u64);
let combined_output = output_carry * base + output_result;
constraints.push(combined_output - computed_output);
```

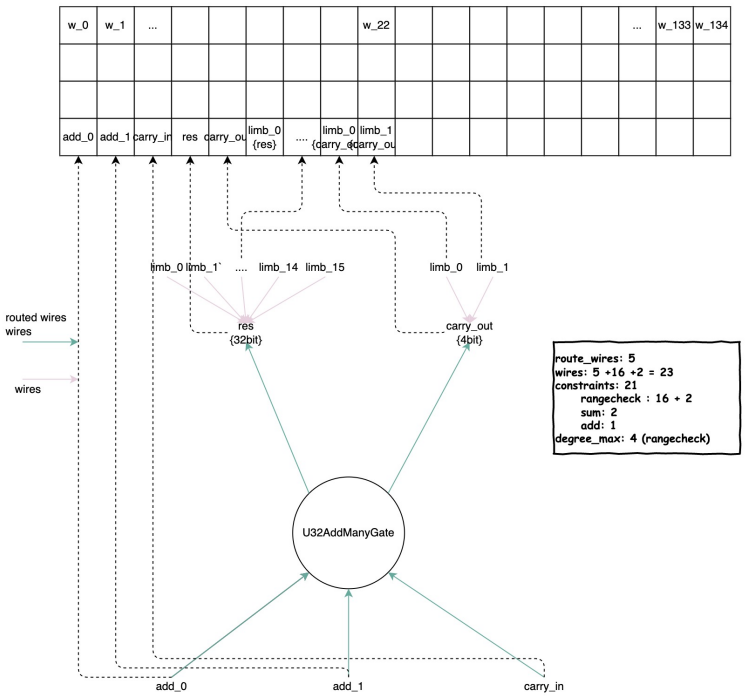


Figure 86: U32AddManyGate

- Limbs range check. – 18(limbs) constraints with degree 4.(limbs are all 2-bits)

```
let product = (0..max_limb)
    .map(|x| this_limb - F::Extension::from_canonical_usize(x))
    .product();
constraints.push(product);
```

- Constrain limbs for res and carry_out. – 2 constraints with degree 1.

In summary, there are 21 constraints for each operation. The degree of the gate is 4 which is needed by the 4-bits limbs range check.

6.2.2.12 arithmetic_u32

U32ArithmeticGate gate is used for compute res $=$ mul_0 $\times$ mul_1 $+$ add. `res` is stored in res_lo and res_hi each of which can be represented by 15 limbs.

The gate structure is like **Figure 87**.

Constraints for each operation:

- Constrain res is not overflow (less than or equal to `max_u32 * max_u32 + max_u32`). – 1 constraint with degree 2.
- Constrain combined (by res_lo and res_hi) output equals computed (by `mul_0 * mul_1 + add`) output. – 1 constraint with degree 2.
- Limbs range check. – 32 (limbs) constraints with degree 4. (limbs are all 2-bits)
- Constrain limbs for res_lo and res_hi. – 2 constraints with degree 1.

In summary, there are 36 constraints for each operation. The degree of the gate is 4 which is needed by the 4-bits

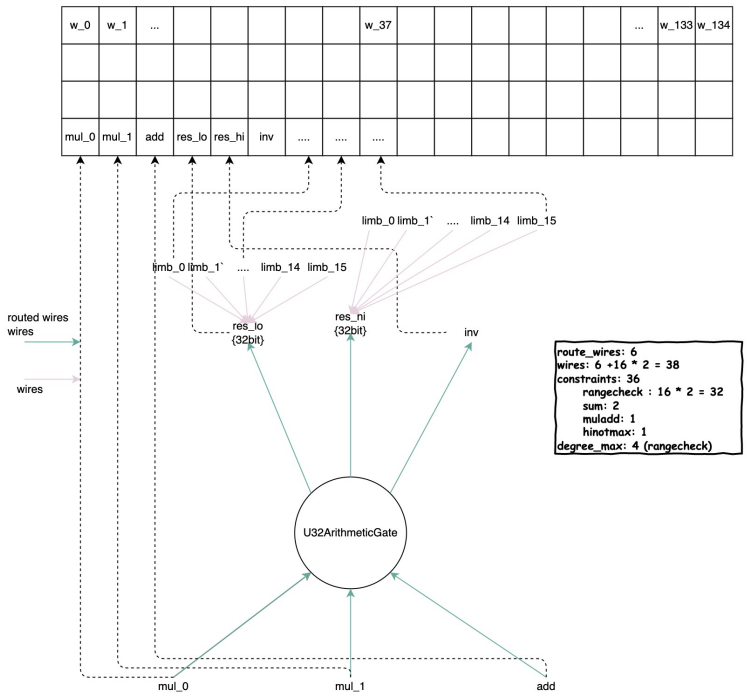


Figure 87: U32ArithmeticGate

limbs range check.

6.2.2.13 comparison

ComparisonGate is a gate for checking that one value is less than or equal to another. Compared numbers are divided into chunks, and chunk size and the number are configurable.

For a ComparisonGate with 8 4-bits chunks is like **Figure 88**.

The main idea of constraints:

- Consistency of the sliced chunks and the original values.
- Range check for each chunk.
- If the chunks are equal, the difference is 0 and there is no inverse.
- Chunk by chunk so for most significant diff equals related intermediate_value.
- If first <= second, the top n+1-th bit of $(2^n + \text{most\_significant\_diff})$ will be 1.

6.2.2.14 range_check_u32

U32RangeCheckGate is a gate that can decompose a number into base B little-endian limbs.

The gate structure is like **Figure 89**.

Constraints for each input_limb:

- Each input_limb consists of its aux_limbs. – 1 constraint with degree 1.

```
let computed_sum = reduce_with_powers(&aux_limbs, base);
constraints.push(computed_sum - input_limb);
```

- aux_limbs range check. – 16 (aux_limbs) constraints with degree BASE $((x-0)(x-1)\cdots(x-\text{BASE}+1))$.

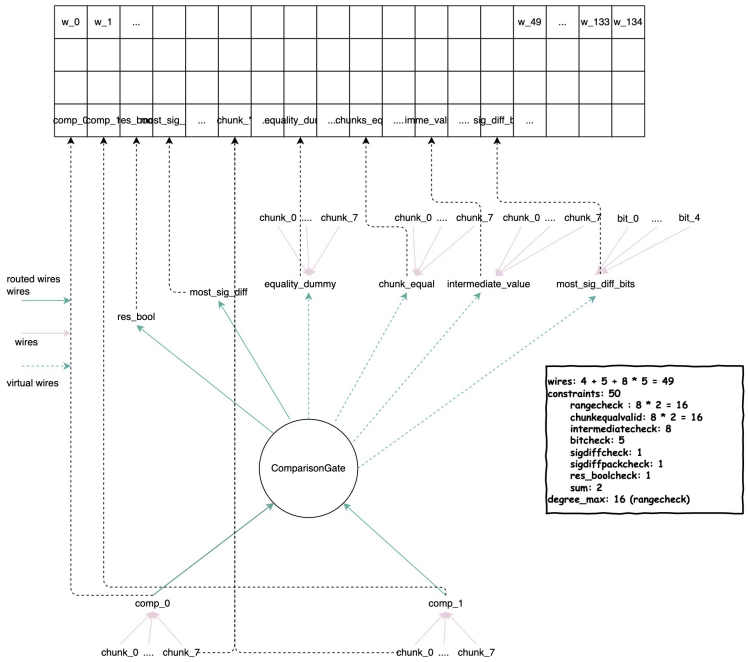


Figure 88: ComparisonGate

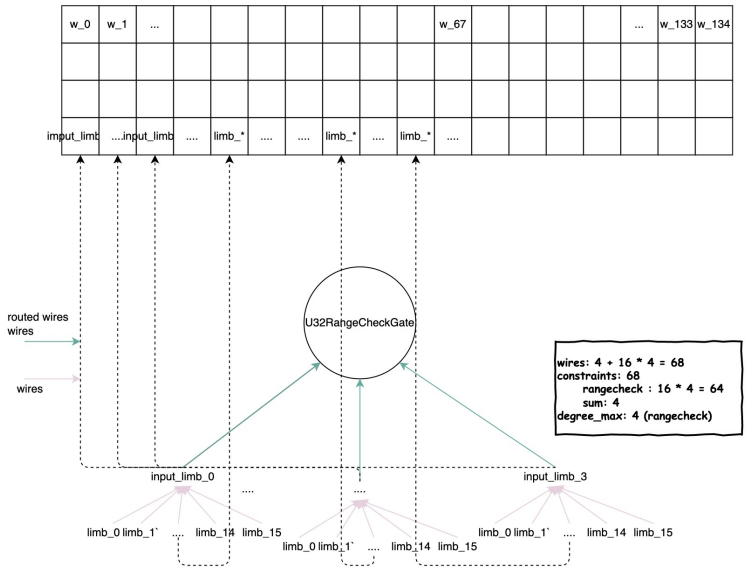


Figure 89: U32RangeCheckGate

In summary, there are 17 constraints per input limb, a total of num_input_limbs $\times$ 17 constraints. The degree of the gate equals BASE for range check.

6.2.2.15 subtraction_u32

U32SubtractionGate is a gate to perform subtraction on 32-bit limbs: given 'x', 'y', and 'borrow', it returns the result $x - y -$ borrow and, if this underflows, a new 'borrow'.

The gate structure is like **Figure 90**.

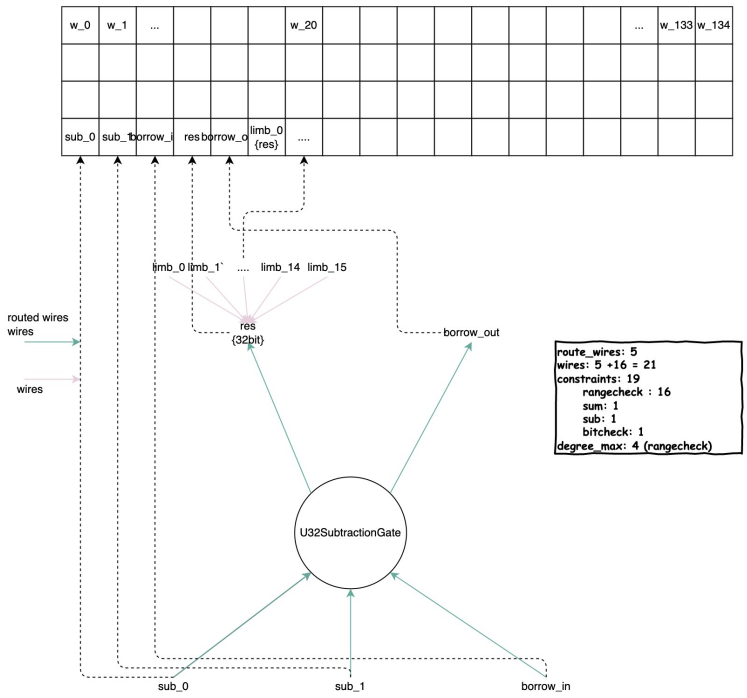


Figure 90: U32SubtractionGate

Constraints for each operation:

- Constrain the calculation. – 1 constraint with degree 2.

```
let result_initial = input_x - input_y - input_borrow;
...
constraints.push(output_result - (result_initial + base * output_borrow));
```

- Limbs range check. – 16 (limbs) constraints with degree 4. (limbs are all 2-bits)
- Constrain limbs for res. – 1 constraint with degree 1.
- Constrain borrow_out to be one bit. – 1 constraint with degree 1.

In summary, there are 19 constraints for each operation. The degree of the gate is 4 which is needed by the 4-bits limbs range check.

6.2.3 Gadgets

6.2.3.1 biguint-add

1. `Target`: Implement the addition of two biguints.
2. `Constraints logic`:
   - Equation for gates;
   - Sumcheck between output and limbs;
   - Rangecheck for limbs.
3. `Circuit layout`: See **Figure 91**.
4. `Trace layout`: See **Figure 92**.
5. `Constraints info and costs`

- gate type num: 1 (U32AddManyGate)
- gate ops num: limbs-num
- gate instance num: ceil(limbs-num / gate.ops)
- copy-constraints: limbs-num * 4
- max-degree: 4 (`1 << limb-bits`)

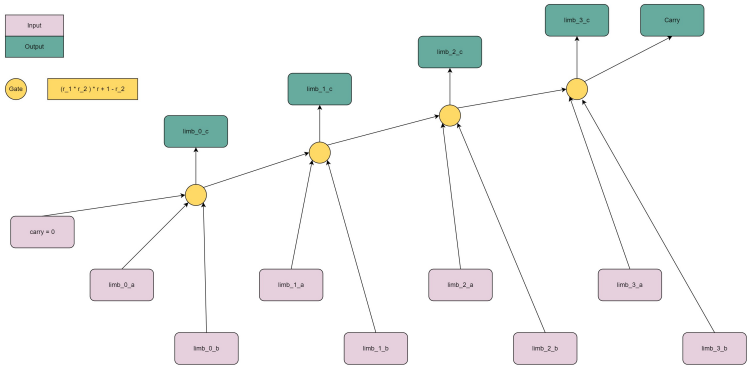


Figure 91: biguint-add circuit layout

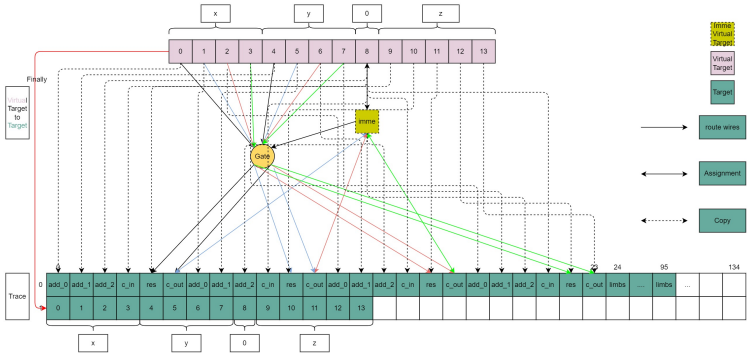


Figure 92: biguint-add trace layout

6.2.3.2 biguint-sub

1. `Target`: Implement the substraction of two biguints.
2. `Constraints logic`:
   - Equation for gate;
   - Sumcheck for ouptput;
   - Rangecheck for limbs.
3. `Circuit layout`: See **Figure 93**.
4. `Trace layout`: See **Figure 94**.
5. `Constraints info and costs`:
   - constraints-num: $6 \times (3 + 32/2) = 114$
   - copy-constraints: 16
   - max-degree: 4

- wires-num: $6 \times (5 + 16) = 126$

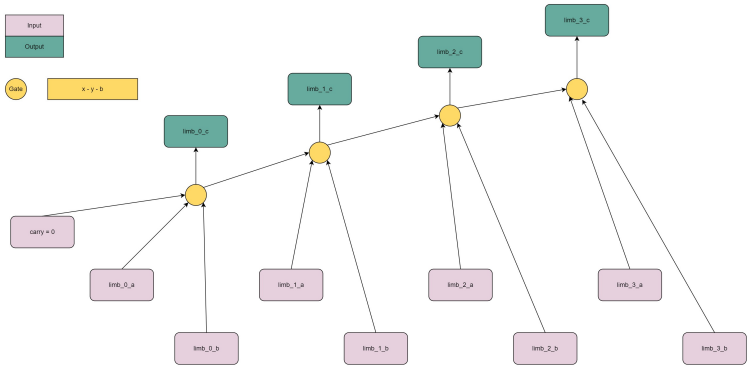


Figure 93: biguint-sub circuit layout

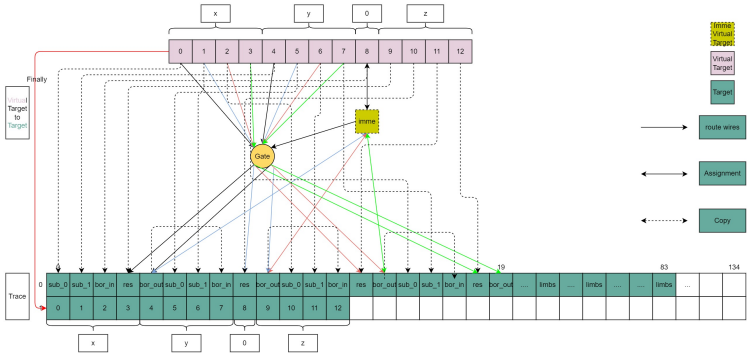


Figure 94: biguint-sub trace layout

6.2.3.3 biguint-mul

1. `Target`: Implement the multiplication of two biguints.
2. `Constraints logic`:
   - Compute mul-factors first, use U32ArithmeticGate;
   - Add mul-factors from low bits, use U32AddManyGate.
3. `Process layout`: See **Figure 95**
4. `Constraints info and costs`:
   - Gate type num: 4 (U32ArithmeticGate, U32AddManyGate(num-addends: 4), U32AddManyGate(num-addends: 6), U32AddManyGate(num-addends: 8))
   - Gate instance num: 9
   - U32ArithmeticGate num: 6
   - U32AddManyGate num: 3
   - copy-constraints: $18 \times 3 + (4 + 6 + 8) \times 2 + 9 = 99$
   - max-degree: 4

6.2.3.4 biguint-div

Note that div-rem has the same constraints logic with div

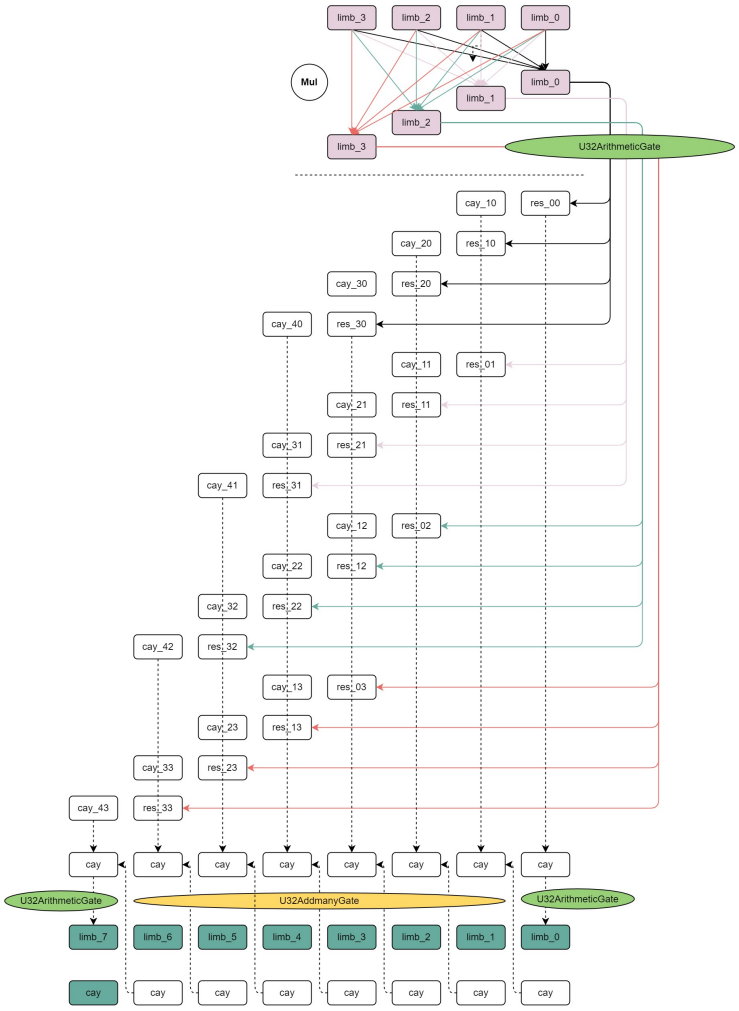


Figure 95: biguint-mul layout

1. `Target`: Implement the division of two biguints.
2. `Constraints logic`:
    - Not implement div-algrithem directly;
    - Use nondeterministic feature to check div-logic;
    - Check `div * b + rem = a`;
    - Check `rem < b`.
3. `Process layout`: See **Figure 96**.
4. `Constraints info and costs`:
    - Gate type num: 5 (U32ArithmeticGate, U32AddManyGate (num-addends: 3), U32AddManyGate (num-addends: 4), ComparisionGate, ArithmeticGate)
    - Gate instance num: $3 + 3 + 4 + 3 = 13$
    - U32ArithmeticGate num: 3
    - U32AddManyGate num: 3
    - ComparisionGate num: 4

- ArithmeticGate num: 3
- copy-constraints: $3 \times 8 + 4 + 5 + 4 + 4 \times 6 + 7 + 1 + 26 + 5 = 100$
- max-degree: 4

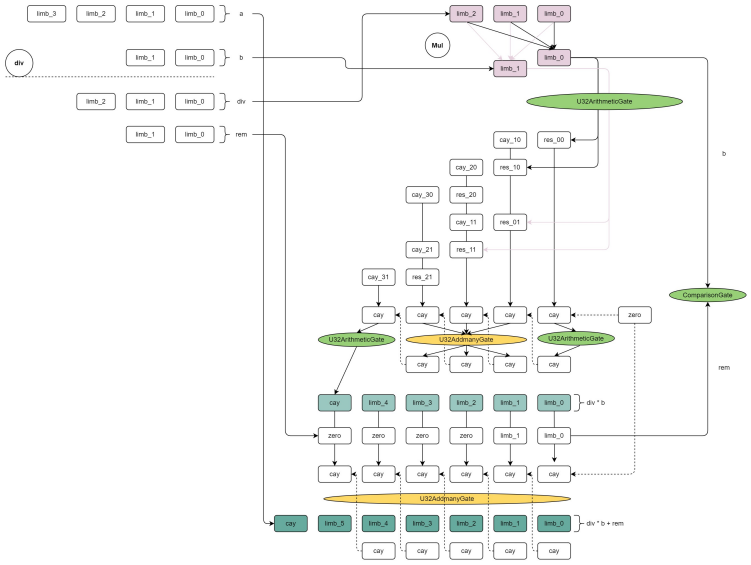


Figure 96: biguint-div layout

6.2.3.5 biguint-cmp

1. `Target`: Implement the comparison of two biguints.
2. `Constraints logic`
   - Split the input to many limbs, such that: `limbs_num = bits / chunks`;
   - Execute comparison for low bits limbs;
   - Ensure that the result is determined by the highest limbs which are not equal.
3. `Process layout`: See **Figure 97**.
4. `Circuit layout`: See **Figure 98**.
5. `Constraints info and costs`:
   - Gate type num: 2 (ComparisionGate, ArithmeticGate)
   - Gate instance num: $4 \times 2 + 3 = 11$
   - ComparisionGate num: 8
   - ArithmeticGate num: 3
   - copy-constraints: $(4 + 9) \times 4 + 1 = 53$
   - max-degree: 4

6.2.3.6 non-native-add

1. `Target`: Check the additional relation among three non-native target objects.
2. `Constraints logic`:
   - Check equation for gadget: `a + b = c + modular * overflow`;
   - Check that "c < modular".

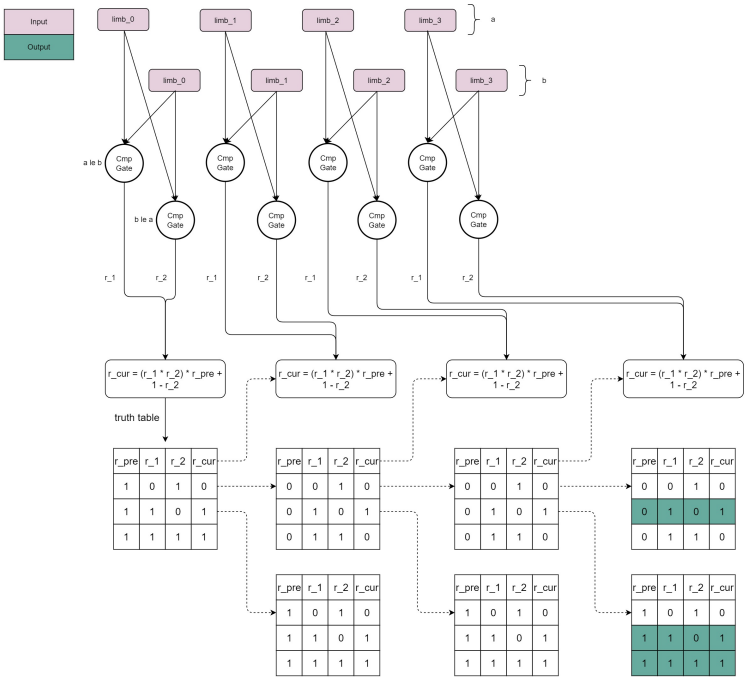


Figure 97: biguint-cmp layout

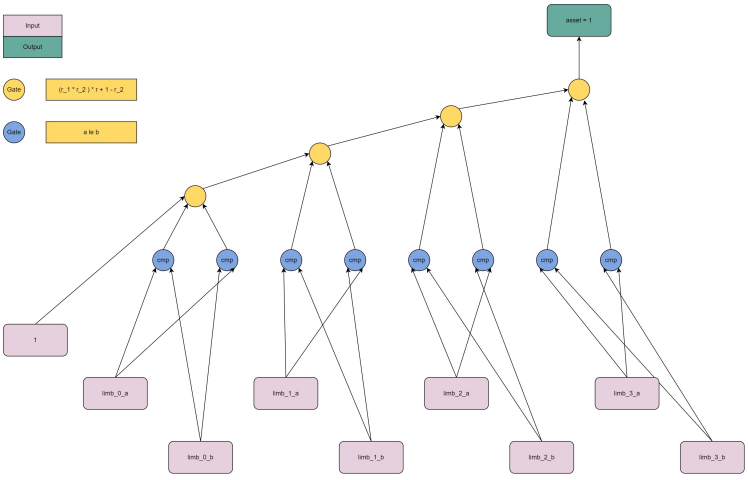


Figure 98: biguint-cmp circuit layout

3. `Process layout`: See **Figure 99**.
4. `Constraints info and costs`:
   - gadget biguint-add num: 2
   - gadget biguint-mul-by-bool num: 1
   - gadget biguint-cmp num: 1
   - gate type num: 3 (U32AddManyGate, ComparisonGate, ArithmeticGate)
   - gate instance num: 23 = 3 (U32AddManyGate) + 16 (ComparisonGate) + 2 (ArithmeticGate (1,0)) + 1 (ArithmeticGate(1,-1)) + 1 (ArithmeticGate(1,1))
   - copy-constraints: 186 = 32 * 2biguint-add + 9biguint-mul-by-bool + 9 + (4 + 9) * 8biguint-cmp = 186

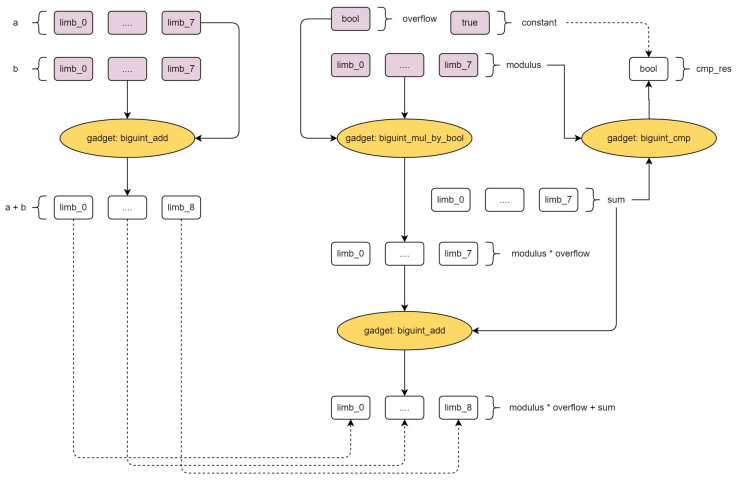


Figure 99: non-native-add layout

6.2.3.7 non-native-sub

1. `Target`: Check the substrate relation among three non-native target objects.
2. `Constraints logic`:
    - Check equation for gadget: `diff + b = a + modular * overflow`;
    - Check that "overflow is bool";
    - Check that "diff.limbs is range U32".
3. `Process layout`: See **Figure 100**
4. `Constraints info and costs`:
    - gadget biguint-add num: 1
    - gadget biguint-sub num: 1
    - gadget biguint-mul-by-bool num: 1
    - gadget u32rangecheck num: 1
    - gadget assert-bool num: 1
    - gate type num: 4 (U32AddManyGate, U32SubtractionGate, U32RangeCheckGate, ArithmeticGate)
    - gate instance num: 7 = 2 (U32AddManyGate) + 2 (U32SubtractionGate) + 1 (U32RangeCheckGate) + 1 (ArithmeticGate(1,0)) + 1 (ArithmeticGate(1,-1))
    - copy-constraints: 89 = 32 (biguint-add num) + 27 (U32SubtractionGate) + 9(biguint-mul-by-bool) + 8 (u32rangecheck) + 4 (assert-bool) + 9

6.2.3.8 non-native-mul

1. `Target`: Check the multiplication relation among three non-native target objects.
2. `Constraints logic`:
    - Check equation for gadget: `a * b = prod + modular * overflow`;
    - Check that "overflow.limb is U32";
    - Check that "prod.limb is U32".
3. `Process layout`: See **Figure 101**.
4. `Constraints info and costs`:

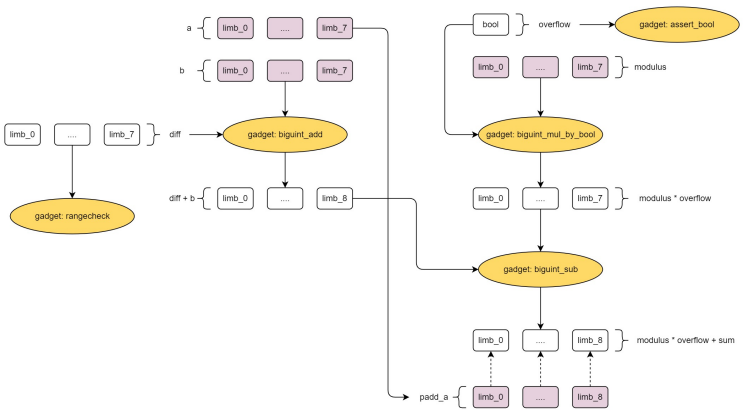


Figure 100: non-native-sub layout

- gadget biguint-add num: 1
- gadget biguint-mul num: 2
- gadget u32rangecheck num: 2
- gate type num: 9 = 7 (U32AddManyGate{3,5,7,9,11,13,15}) + 1 (U32RangeCheckGate) + 1 (U32ArithmeticGate)
- gate instance num: 37 = 2 (u32rangecheck) + 8 (biguint-mul: constant-input) + 22 (biguint-mul) + 1 + 3 (biguint-add)
- copy-constraints: 583 = 8 * 2 (u32rangecheck) + 3 * 3 + (4 + 6 + 8 + 10 + 12 + 14 + 16) * 4 + (8 * 8) * 3 + 17 * 4 + 18

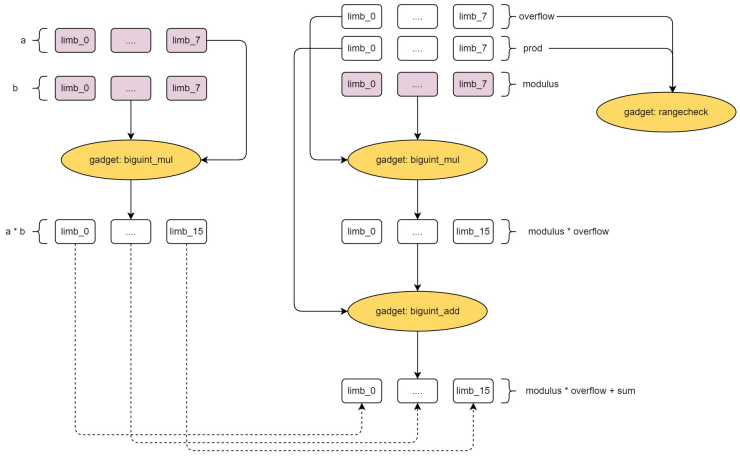


Figure 101: non-native-mul layout

6.2.3.9 non-native-inv

1. `Target`: Check the modular inverse relation among three non-native target objects.
2. `Constraints logic`:
   - Check equation for gadget: `a * inv_a = 1 + modular * div`.
3. `Process layout`: See **Figure 102**.
4. `Constraints info and costs`:

- gadget biguint-add num: 1
- gadget biguint-mul num: 2
- gate type num: 8 = 7 (U32AddManyGate{3,5,7,9,11,13,15}) + 1 (U32ArithmeticGate)
- gate instance num: 56 = (8 * 8 + 2) * 2 / 3 + 5 (U32AddManyGate3) + 5 + 2 (U32AddManyGate15)
- copy-constraints: 762 = (8 * 8 + 2) * 2 * 3 + 21 * 4 + (6 + 8 + 10 + 12 + 14) * 4 + 4 * 16 + 18

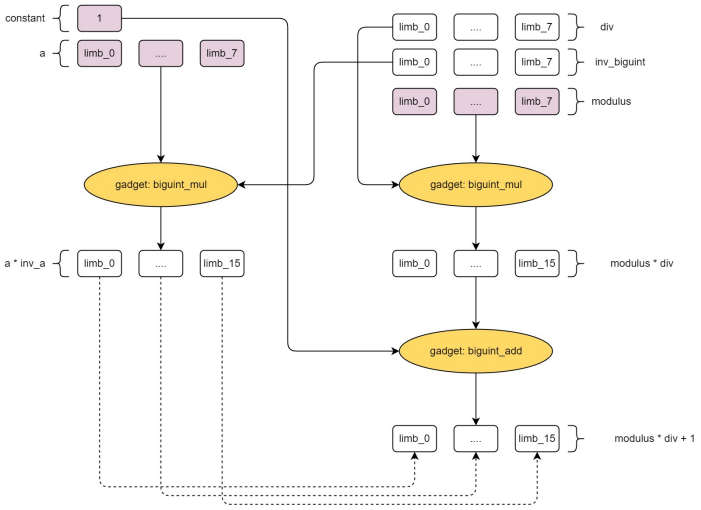


Figure 102: non-native-inv layout

6.2.3.10 curve-add

1. `Target`: Implement the addition of two different curve points. this is an incomplete addition, you can refer to The halo2 Book [40] to learn more about it.
2. `Constraints logic`: $(x_1, y_1) \neq (x_2, y_2)$, See **Figure 103**.
3. `Process layout`: See **Figure 104**
4. `Constraints info and costs`:
   - gadget-sub-nonnative num: 5
   - gadget-add-nonnative num: 1
   - gadget-mul-nonnative num: 3
   - gadget-inv-nonnative num: 1
   - gate type num: 14
     - 8: U32AddManyGate{2,3,5,7,9,11,13,15}
     - 1: ComparisonGate
     - 1: ArithmeticGate
     - 1: U32ArithmeticGate
     - 1: U32SubtractionGate
     - 2: U32RangeCheckGate{0,8}

6.2.3.11 curve-double

specified.

Let E be a Weierstrass form elliptic curve $y^2 = x^3 + ax + b$. Let (x_1, y_1) be a point in $E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$. Then, $(x_1, y_1) + (x_1, -y_1) = \mathcal{O}$. Further let (x_2, y_2) be a point in $E(\mathbb{F}_q)\backslash\{\mathcal{O}\}$ such that $y_2 \neq 0$ and $(x_2, y_2) \neq (x_1, -y_1)$. Then, $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ where

$$x_3 = \lambda^2 - x_1 - x_2, \tag{7.1}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \tag{7.2}$$

with

$$\lambda = \begin{cases} (y_1 - y_2)/(x_1 - x_2) & \text{if } (x_1, y_1) \neq (x_2, y_2) \\ (3x_1^2 + a)/(2y_1) & \text{if } (x_1, y_1) = (x_2, y_2) \end{cases},$$

Figure 103: curve-add

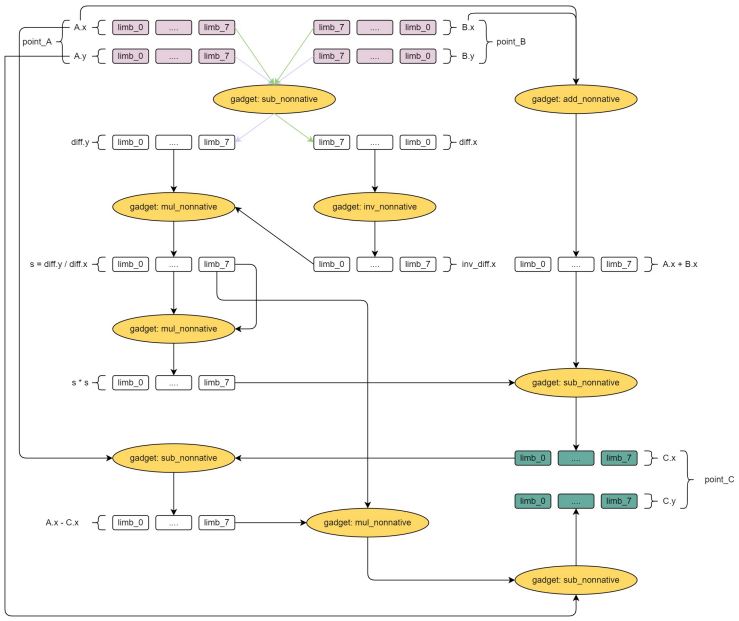


Figure 104: curve-add layout

1. `Target`: Implement the addition of two same curve points. this is an incomplete addition, you can refer to The halo2 Book [40] to learn more about it.
2. `Constraints logic`: See **Figure 105**.
3. `Constraints info and costs`:
   - gadget-sub-nonnative num: 3
   - gadget-add-nonnative num: 5
   - gadget-mul-nonnative num: 4
   - gadget-inv-nonnative num: 1
   - gate type num: 14
     - 8: U32AddManyGate{2,3,5,7,9,11,13,15}
     - 1: ComparisonGate
     - 1: ArithmeticGate
     - 1: U32ArithmeticGate
     - 1: U32SubtractionGate
     - 2: U32RangeCheckGate{0,8}

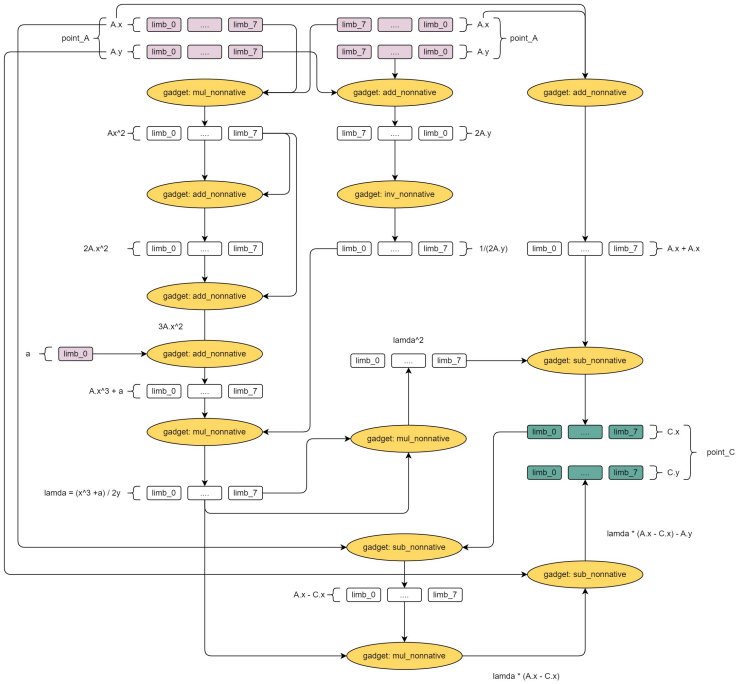


Figure 105: curve-double layout

6.2.3.12 curve-assert-valid

1. `Target`: Check the point is on the curve $y^2 = x^3 + ax + b$.
2. `Process layout`: See **Figure 106**.
3. `Constraints info and costs`:
    - gadget-add-nonnative num: 3
    - gadget-mul-nonnative num: 3
    - gate type num: 13
        - 8: U32AddManyGate{2,3,5,7,9,11,13,15}
        - 1: ComparisonGate
        - 1: ArithmeticGate
        - 1: U32ArithmeticGate
        - 2: U32RangeCheckGate{0,8}

6.2.3.13 curve-msm

1. `Target`: Implement the multiplication scalar multiplication (MSM).
2. `Constraints logic`: See **Figure 107**.
3. `Constraints info and costs`:
    - gate type num: 14
    - gate instance num: 147553

6.2.3.14 curve-scalar

1. `Target`: Implement the multiplication between scalar and point.

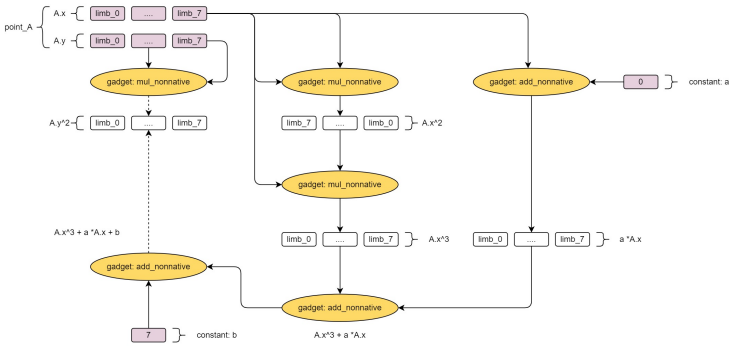


Figure 106: curve-assert valid layout

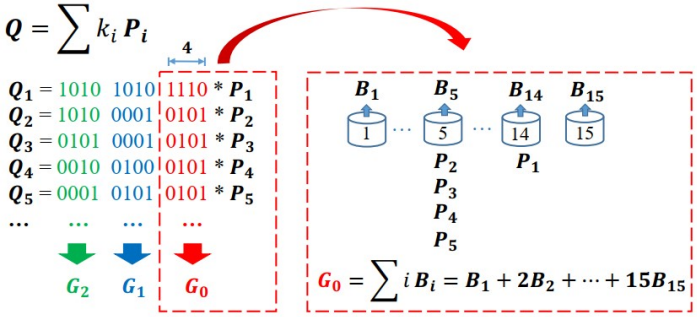


Figure 107: curve-msm

2. `Constraints logic`: See **Figure 108**.
3. `Constraints info and costs`:
   - gate type num: 13
   - gate instance num: 180364

Let E be an elliptic curve over $\mathbb{K}$. A point P can be multiplied by a scalar as $[k]P = \sum_{i=1}^{k} P$. Efficient computation of $[k]P$ is an active research area in ECC implementations since a majority of the computational power is spent on this operation. A standard double-and-add technique to compute k-folds of P is presented in Algorithm 2.4.1.

Algorithm 2.4.1: Left-to-right binary method for scalar multiplication

input : $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{K})$.
output: $Q \leftarrow [k]P$.

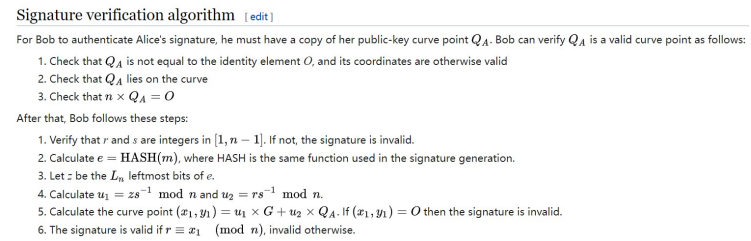
```
Q ← O.
for i = t − 2 to 0 do
    Q ← [2]Q.
    if k_i = 1 then
        Q ← Q + P.
    end
end
return Q.
```

Figure 108: curve-scalar

6.2.3.15 ecdsa

1. `Target`: Implement ecdsa verify process.
2. `Constraints logic`: See **Figure 109**.

3. `Constraints info and costs`:
    - gate type num: 16
    - gate instance num: 98039



Signature verification algorithm [edit]

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid
2. Check that Q_A lies on the curve
3. Check that $n \times Q_A = O$

After that, Bob follows these steps:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $u_1 = zs^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$.
5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
6. The signature is valid if $r \equiv x_1 \pmod n$, invalid otherwise.

Figure 109: ecdsa

6.3 Inner Table Lookup

Inner-table lookup can be thought of as a subset argument technology between columns I and T, proving that cells in column I all appear in column T at least once, and it does not matter whether the cells in column T are in I. To simplify the calculation, the columns in the STARK table are all 2^k rows, and when the cells of columns I and T are not 2^k, they need to be extended.

- Cells in column I can be extended with elements in column I or column T
- Cells in column T can only be extended with cells in column T, usually the last one

The inner-table lookup technology is to first generate the corresponding permutation columns I' and T' from columns I and T, respectively. The permutation relationship between I and I', T and T' can be checked through the permutation argument during the proving phase:

$$Z(\omega X) \cdot (I'(X) + \beta) \cdot (T'(X) + \gamma) - Z(X) \cdot (I(X) + \beta) \cdot (T(X) + \gamma) = 0$$
$$1 - Z(\omega^0) = 1 - Z(\omega^k) = 0$$

Then sort columns I' and T' as follows:

1. Column I' is ordered by a sorting algorithm so that cells of the same value are adjacent to each other.
2. Column T' is ordered so that the first row in a sequence of the same values in I' has the same value in T' at the same position.

Now, we'll enforce either $I'_i = T'_i$ or that $I'_i = I'_{i-1}$, using the rule:

$$(I'(X) - T'(X)) \cdot (I'(\omega X) - I'(X)) = 0$$
$$I'(\omega^0) - T'(\omega^0) = 0$$

These constraints effectively force every element in I' (and thus I) to equal at least one element in T' (and thus T).

6.4 Cross Table Lookup

The goal of cross table lookup is different from that of inner-table lookup, which is not only a subset argument, but a mix of subset argument and permutation argument.

Cross table lookup indicates the multi-table lookup in the plookup [20], that is, the data in a looked table comes from multiple looking tables. When implementing STARK tables, Starky should define all the STARK cross table

lookup relationships, that is, which other STARK tables each STARK table needs to look from, and which columns and rows are involved (filtered) in the looked table and looking table respectively during the lookup. Each STARK table needs to perform cross table lookups, so there are multiple CrossTableLookups in Starky, and each cross table lookup instance consists of a looked table and multiple looking tables.

We generate a $Z(X)$ polynomial from calculating the permutation product as the grand product argument polynomial for the permutation argument in a cross table lookup instance, and then enforce that they are permutations using a permutation argument with $Z(X)$ polynomial

1. Initialize cross table lookup instances and random numbers β, γ
2. Calculate the partial products of the looking table and looked table, compressing the values of multiple columns into one:
$$C = \prod(c_i \beta^i) + \gamma$$
3. Calculate $Z(X)$ polynomial from partial products
$$Z(\omega X) = Z(X) \cdot (C \cdot \text{filter} + 1 - \text{filter})$$
4. Verify that the product of the last element of the $Z(X)$ polynomials of all looking tables is equal to the last element of the $Z(X)$ of the looked table
$$\prod_i Z_i^{\text{looking}}(\omega^{k_i - 1}) = Z^{\text{looked}}(\omega^{k_{\text{looked}} - 1})$$

Cross table lookup enforces STARK tables are correctly generated from the program, and reduces the rows in CPU STARK table.

6.5 GPU Acceleration

The polynomial computations of ZK mainly consists of multiple NTTs and iNTTs. The large-size NTTs result in sigificant challenges for both off-chip memory accesses and on-chip compute resources, due to the irregular strided access patterns similar to classical FFTs. The GPU acceleration section takes full advantage of GPU multi-threaded parallel processing to speed up NTT calculations.

6.5.1 NTT Computations

The NTT computation is defined as

$$\hat{a} = NTT(a)$$

Where a and $\hat{a}$ are two N size arrays, and $\hat{a}[i] = \sum_{j=0}^{N-1} a[j]\omega_N^{ij}$. Here $a[i]$ and $\hat{a}[i]$ are λ-bit scalars in a finite field. And ω_N is the N-th root of unity in the same field, also is called twiddle factor, which is a constant value for a specific size of N.

Similar to classical FFTs, optimized and improved performance implementations of the NTT are of necessity and fundamental essentials. Cooley-Tukey processing is a commonly used NTT algorithm in science and engineering. For example, the Cooley-Tukey algorithm reduces the computational complexity from $O(N^2)$ to $O(N \log N)$. Its fundamental idea is to recursively decompose a large-size NTT into several linear combinations of smaller NTTs. The butterfly network of Cooley-Tukey NTT is a decimation-in-time network whereby its twiddle factor appears on the input side of calculation and input data are arranged in bit-reversed order. **Figure 110** illustrates the butterfly networks of Cooley-Tukey NTT. The NTT optimization mainly involves butterfly network and butterfly computing.

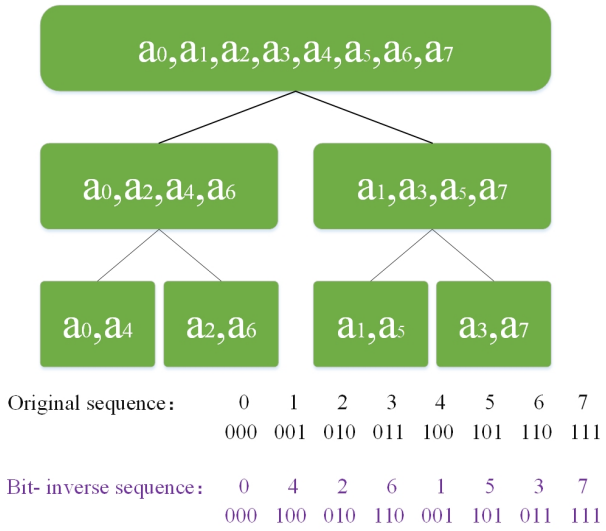


Figure 110: The butterfly networks of Cooley-Tukey NTT

6.5.2 2-D NTT

NTT is an important kernel commonly used in our ZK scheme. In fact, there exist many accelerator designs for NTT. However, the polynomial computations in our ZK scheme have substantially larger scales than those addressed in other ZK. It requires multiple NTTs of up to 2^{23} or 2^{24} elements, Such large sizes can hardly be satisfied by any previous GPU design.

To overcome the large-sizes challenges, we adopt a 2-D NTT algorithm to decompose a large NTT into multiple smaller NTT kernels (4096/8192). This allows GPU to only implement smaller NTT modules, which can fit into the compute resources (e.g. the stream processor number of on GPU). Then the original large-size NTT can be calculated by iteratively using the smaller NTT.

The overview of 2-D NTT algorithm is shown in **Figure 111**. First of all, a large NTT size N is decomposed into P rows and Q columns, with $N = P \times Q$. Then the N-size NTT can be calculated by several P-size and Q-size smaller NTTs. Firstly, the P-size NTT is parallel computing using the same twiddle factors for each of the Q columns. The output is still a P-size vector. Before the second dimensional NTT, a corresponding twiddle factor ω^{ij} is multiplied to each element of the output vector. Finally, a Q-size NTT for each of the P rows is parallel computing. The output column-major order elements is the result of the large-size NTT.

6.5.3 GPU and CUDA

Currently, the GPU is no longer limited to graphics processing. Thanks to the rapid evolution of their architecture that made it straightforward to access their underlying powerful parallelism to address different categories of data parallel algorithms to be handled efficiently by this platform. The GPU is designed to process a huge amount of numbers operations in parallel relying on thousands of computing cores running concurrently. CUDA is a general-purpose parallel computing architecture that delivers a novel parallel programming paradigm and a new instruction set architecture that can easily leverage the parallel power of the computing engine embedded in the CUDA-enabled GPUs to solve complex scientific and engineering applications. It becomes straightforward to decompose the computations

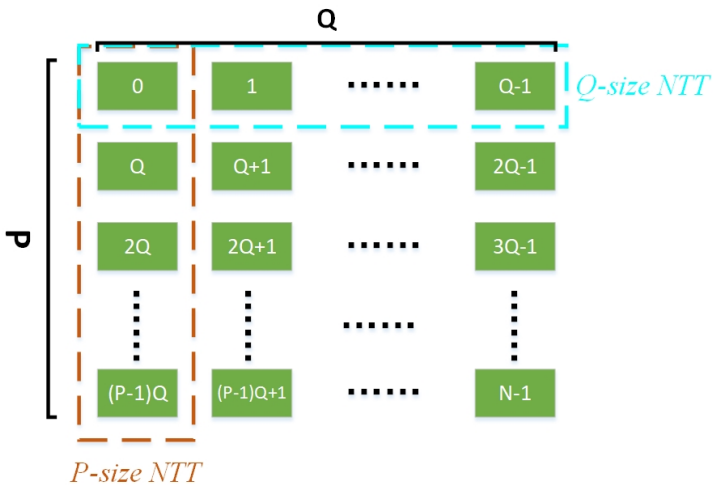


Figure 111: The 2-D NTT algorithm

into fine-grained threads and execute theses threads on the device in parallel. CUDA refers to the software platform and the hardware architecture used to execute the kernels. A kernel launches thousands of threads for parallel execution organized in three levels: grids, blocks, and warps. The grids are two- or three-dimensional arrangements of threads blocks where every block consists of an upper limit of threads 512 or 1024 depending on the compute capability version of the device, and finally each group of threads form a wrap. Such architecture is called Single Instruction Multiple Thread (SIMT); which allows the GPU to execute single instruction on multiple concurrent threads operating on different data. Every running thread has access to some built-in variable assigned by the CUDA run time system and used for thread indexing. CUDA also has an efficient memory hierarchy divided in global, shared, register, texture, and constant memory.

6.5.4 Implement of 2-D NTT with GPU

To overcome the large-sizes challenges on GPU, the 2-D NTT algorithm is adopted. For a particular GPU model, the number of stream processors has been set to a fixed value, so the total number of threads running simultaneously on this GPU is limited. In order to make full use of the GPU thread resources, we set NTT's two-dimensional running framework as shown in **Figure 112** with a 2^{23} size NTT. The NTT sequence is divided into 8192 rows and 1024 columns. For each column, 8192 threads (This number can be adjusted based on the number of stream processors to achieve maximum GPU utilization) are allocated to calculate 8192-size 1D NTT in parallel. After processing each column, the NTT of the second dimension can process 8 rows at a time through the Cooperative Groups (CG) mechanism. The Cooperative Groups programming model describes synchronization patterns both within and across CUDA thread blocks. It provides CUDA device code APIs for defining, partitioning, and synchronizing groups of threads. It also provides host-side APIs to launch grids whose threads are all guaranteed to be executing concurrently to enable synchronization across thread blocks. These primitives enable new patterns of cooperative parallelism within CUDA, including producer-consumer parallelism and global synchronization across the entire thread grid or even multiple GPUs.

Therefore, the total running time of the GPU is 1024 8192-size NTTs plus 1024 1024-size NTTs.

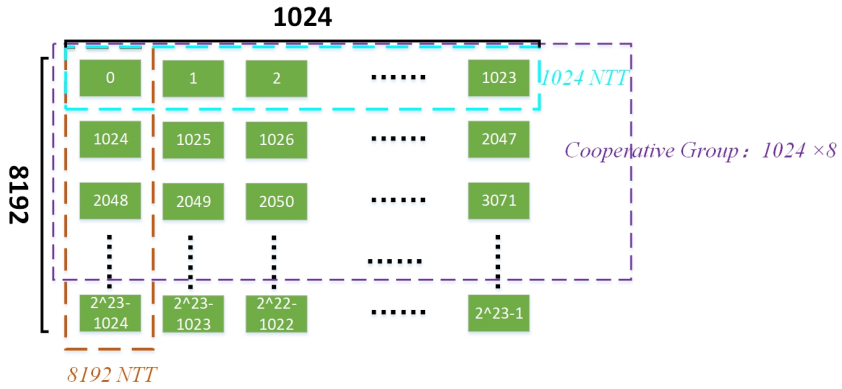


Figure 112: Implement of 2-D NTT with GPU

6.5.5 Memory Access

The following **Figure 113** shows the CUDA memory model. The large-size NTT data is copied from the host into global memory. Then it is sent to shared memory into the compute queue. For faster access, precomputed twiddle factors are stored in texture memory. When the twiddle factor is used, it is copied sequentially to the registers according to the index. The results of one-dimensional NTTs are saved in shared memory. When the NTT is calculated, the output is copied from shared memory to global memory and uploaded to the host.

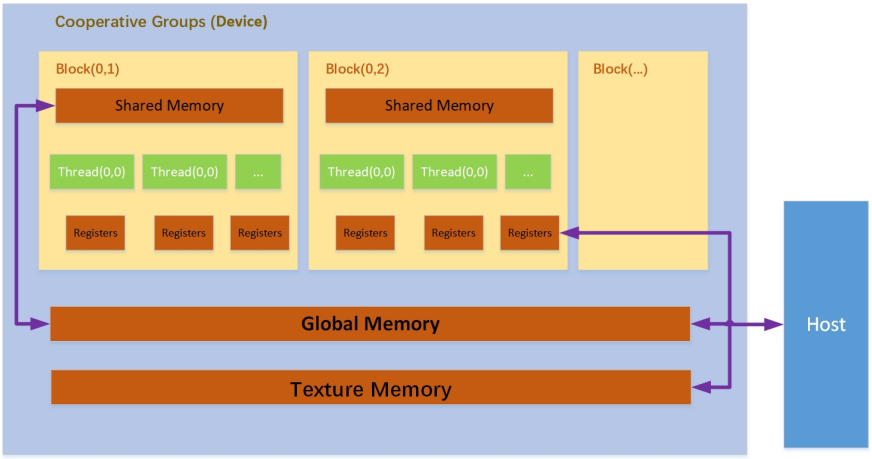


Figure 113: CUDA Memory Model

6.6 FPGA Acceleration

The NTT acceleration design consists of two main parts: the NTT engine and the DMA data transfer engine. The NTT engine performs calculations on given-length data points, while the DMA data transfer engine efficiently exchanges data between the NTT engine and DDR.

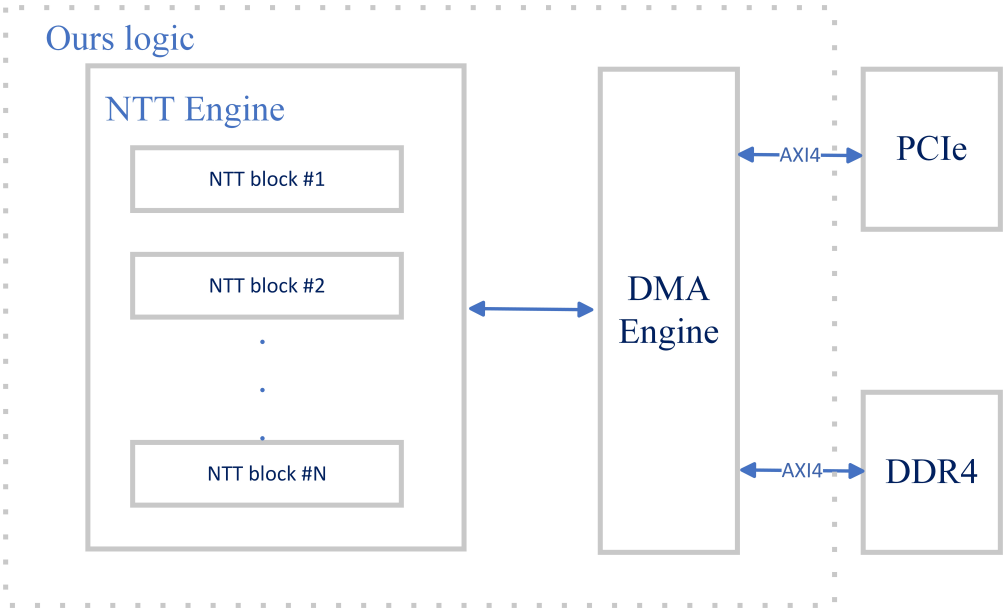


Figure 114: The design of NTT acceleration implemented in FPGA

6.6.1 NTT engine

In this design, the NTT engine consists of multiple parallel-running NTT modules. The maximum supported data length for a single NTT module is 2^{12}. Moreover, a modular multiplication unit is connected to the output of each NTT module to support the extension of two-dimensional or higher-dimensional NTTs. To improve scalability performance, variable-length NTT calculations can be achieved through parameter configuration through the AXI4 interface.

In practical applications, processing in parallel with multiple NTT modules can greatly improve computational performance. For a single NTT module, one point reading and one storage operation are required per clock cycle. When 8 NTTs are processed in parallel in the engine, the total access bandwidth required is 8 * 8 byte/point * (1x read + 1x write) * 100MHz = 11.92 GB/sec with assuming a working clock of 100MHz. That is, the calculation rate of NTT is 8 * 100MHz / 2^{24} / 2= 25 times/sec. For the AWS FPGA platform, the user logic can use three DDR4 channels with a interface frequency up to 2100MHz. The total supported simultaneous access bandwidth is 3 * 8 byte/DDR * 2100MHz = 46.9 GB/sec. Therefore, the maximum supported NTT calculation rate is 25 * 46.9 / 11.92 = 98 times/sec.

NTT calculation module To support NTT calculation with the length of 2^{24}, it is better to consider the 2^{24} points as a 2D array with a size of 2^{12} points by 2^{12} points. Using the Gentleman-Sande algorithm, the first pass through the engine computes 2^{12} NTTs column-wise of the 2D array using the carefully designed NTT module with a length of 2^{12}. For this module, a recursive NTT Algorithm is adopted, which contains 12 stages of butterfly with varying kernel sizes. Users can adjust the sizes of both columns and rows in the 2D array by setting parameters to meet FPGA running requirements. Finally, the output of the NTT block performs modular multiplication according to the 2D NTT algorithm.

NTT kernel The NTT block contains several kernels, each of which is mainly composed of a butterfly unit and two RAMs. The butterfly unit is carefully designed to take maximum advantage of the FPGA DSPs, especially by utilizing

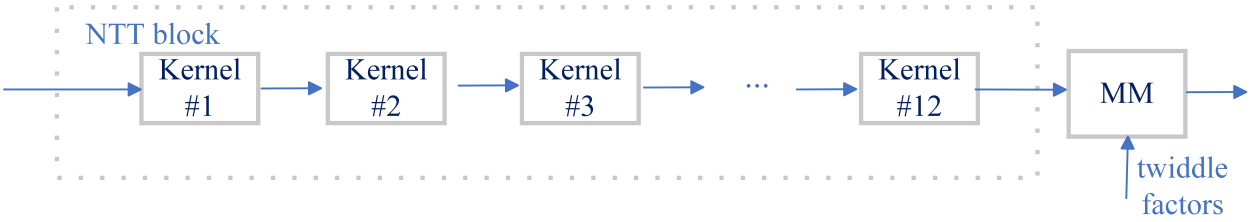


Figure 115: Various-size NTT block (MM: modular multiplication)

the tricks of the Goldilocks field multiply to improve computational efficiency over modular reduction. One of the RAMs is used to buffer the input data, while the other stores the twiddle factors required for butterfly calculation.

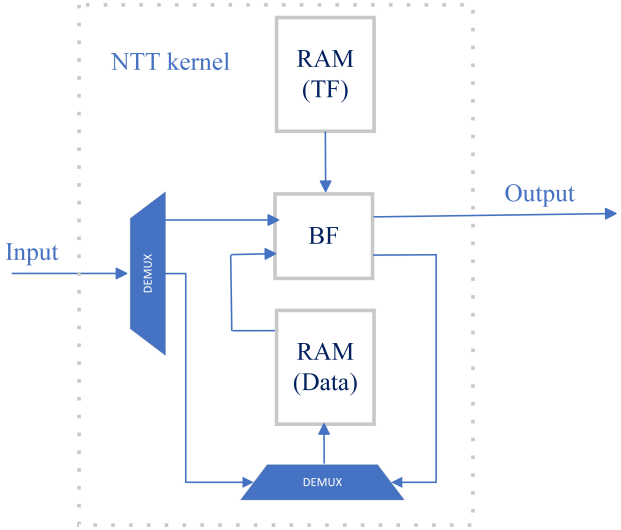


Figure 116: NTT kernel (TF: twiddle factor; BF: Butterfly)

Generation of twiddle factors To realize the 2D NTT, the outputs of the 1st pass with the NTT engine should be multiplied by the corresponding twiddle factors, as shown in Fig.2. Since the twiddle factors required for the NTT output of different columns of the 2D array are different, these twiddle factors must be updated for every column. To reduce the necessary DDR bandwidth, a combination of LUT and computation is used to generate these twiddle factors. The LUT provides 2^{12} twiddles needed for modular multiplication before the 2nd pass.Then, the updating twiddle factor is computed by multiplying the LUT twiddles with an accumulated running twiddle factor and then writing back to the LUTs at the same of modular multiplying. Additionally, the twiddle factors needed in the NTT block are provided by LUTs and loaded once before performing the NTT. Therefore, the total number of multiplication/reduction units per block is 13: 11 for the 12 butterfly stages and 2 for each twiddle updating and modulation multiplication out of the NTT block.

Bit Reversal The NTT engine performs bit reversal during NTT computation, which is achieved by bit reversing each pass of the NTT at the end of the block (as shown in the above NTT blocks). A final step is required to complete the bit reversal across the two passes. This can be achieved by writing back the results of the 2nd pass using the column-major access pattern instead of the row-major access pattern.

6.6.2 DMA Engine

The sole responsibility of the DMA Engine is to exchange points between the NTT Engine and the memory outside of the FPGA at the NTT Engine's burst processing rate. With the NTT Engine capable of consuming and producing 8 points per cycle when 8 NTT blocks are working concurrently, and an NTT Engine clock rate nearing 300MHz, the feed and sink rates are both 17.9 GB/sec. The FPGA's DDR4 memory, with a peak bandwidth of 46.9 GB/sec, can support the combined feed and sink bandwidth need of 35.8 GB/sec while also providing enough storage for 2^{24} points. In addition, to fully utilize the bandwidth of DDR4 and continuously provide high data bandwidth to the NTT engine, a large data buffer RAM is introduced in the DMA engine to maintain a high data throughput rate for the AXI4 interface.

6.6.3 Simulation

To verify the feasibility of the designed NTT acceleration scheme, it was implemented in Vivado using Verilog HDL language and both functional and timing simulations were performed. The simulation results are shown in the following figure.

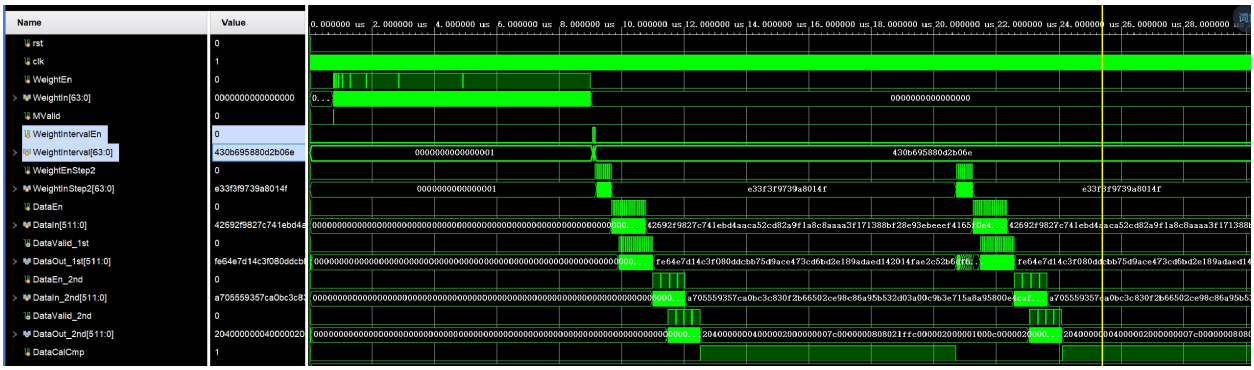


Figure 117: NTT simulation

In the Figure: the signals of WeightEn and WeightIn are the configurations for the twiddle factors of the NTT module; WeightEnStep2 and WeightInStep2 are twiddle factors for the output of large number modular multiplication in the NTT module; signals of WeightIntervalEn and WeightIntervalIn are used to be accumulated twiddle factors when updating the twiddle to multiply the outputs of NTT blocks; DataIn and DataEn are the input data for the first dimension of the NTT, while DataOut_1st and DataValid_1st are the computed results outputted; DataIn_2nd and DataEn_2nd are the input data for the second dimension of the NTT, and DataOut_2nd and DataValid_2nd are the computed results outputted; DataCalCmp is used for indicate the comparison of the computed NTT results with the theoretical ones.

In the implementation, 8-way parallel NTT acceleration is used. According to the timing, the NTT engine is respectively configured with LUT twiddle factors, LUT twiddle factors of modular multiplication, and accumulated twiddle factors. To compute a 4096-point NTT, the size of the 2D array is 8 * 16 * 32 for the first passs and 8 * 4 * 128 for the second one. Based on the comparison signal DataCalCmp, the results obtained from the 2D NTT operation are consistent with the theoretical values. When performing the second 4096-point NTT acceleration, it is only necessary to reload the modular multiplication LUT twiddle factors.

7 Appendix

7.1 Parallel Execution

7.1.1 Challenges in Parallel Transaction Execution

The transaction execution process for most blockchains is as follows: the execution module sequentially retrieves transactions from the block and executes them one by one. During execution, the latest world state is modified. When a transaction is completed, the state is accumulated until the latest world state is reached after the completion of the block. The execution of the next block strictly depends on the world state after the execution of the previous block. Therefore, it is difficult to optimize parallel execution in the traditional linear transaction execution process.

In traditional blockchain projects, transactions cannot be directly executed in parallel, mainly due to the following types of race conditions:

1. Account conflicts: If two threads update the balance or other attributes of the same account simultaneously, the result must be consistent with sequential processing, meaning the world state is a deterministic finite state machine.
2. Storage conflicts: Two contract variables modify the storage of the same global variable simultaneously.
3. Cross-contract call conflicts: Contract A must be deployed first, and Contract B calls Contract A after Contract A is deployed. However, after parallel transaction execution, the order no longer exists, result in a conflict.

7.1.2 OlaVM Parallel Transaction Execution Solution

The implementation of OlaVM is as follows: each transaction within the same block starts from the world state of the previous block and is executed in parallel. During execution, all three types of race conditions encountered in the ideal execution path are recorded. After the parallel execution phase is finished, the merge phase begins. In the merge phase the world states are sequentially combined. When merging each transaction, First, determine if there is a conflict. If there are no conflicts, merge directly; if conflicts exist, re-execute the transaction from the starting point of the merged world state to obtain a new world state.

If the current node needs to process blocks synchronized from other nodes, the processing flow is as follows:

1. The node starts the parallel transaction execution process.
2. After the parallel transaction execution is completed, the node merges the world state.
3. The node verifies the final merged world state using the synchronized block hash.
4. If the verification is correct, the node broadcasts the block to the network.
5. If the verification is incorrect, the node re-executes the block sequentially.

In general, if the computing task can be fully parallelized, the scalability of a single chain will be enormous. This is because nodes can add more CPU cores. However, the actual situation is not like this. The maximum theoretical speed is limited by Amdahl's Law [3]: the speedup limit of the system depends on the reciprocal of the non-parallelizable part. For example, if 99% parallelism can be achieved, the speed can be increased 100 times; if only 95% parallelism can be achieved, the speed can be increased 20 times.

For the replay of all Ethereum transactions, about 80% of transactions can be parallelized, while the remaining 20% cannot be parallelized. Therefore, the theoretical speed-up limit of Ethereum transaction processing using this

solution is about 5 times.

For block producers, since they can be deployed on higher-performance machines, If they can accurately identify the conflicting conditions of different transactions and sort these transactions according to the best strategy, the parallelism may be greatly improved, thus achieving higher speedup.

The transaction process is executed in parallel as shown in the figure 118.

The comparison between parallel execution and sequential execution is shown in the figure 119.

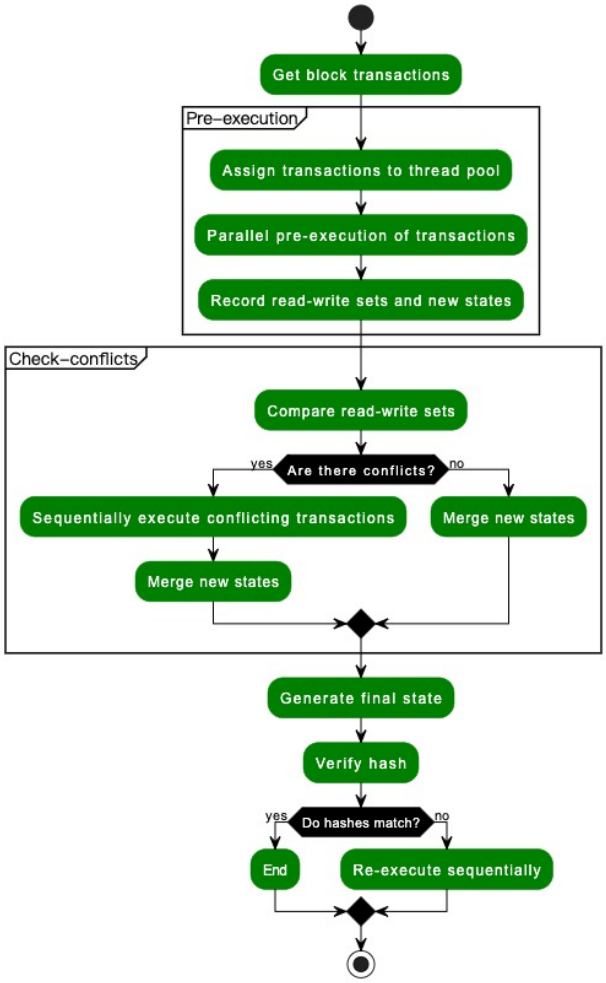


Figure 118: Parallel Transaction Execution Process

7.1.3 Conflict Transaction Marking

In EVM, conflicts mainly occur during the process when certain instructions perform read and write operations on storage. By recording these read-and-write operations, a read-and-write set can be formed. However, static code analysis cannot ensure that all these processes are recorded. Therefore, when processing transactions in each block, each transaction needs to be pre-executed in parallel. Through the pre-execution process, it can be identified whether these transactions perform read and write operations on the same account or storage, and then a read-set and write-set can be generated for each transaction. The pre-execution process first copies the world state multiple times as the initial

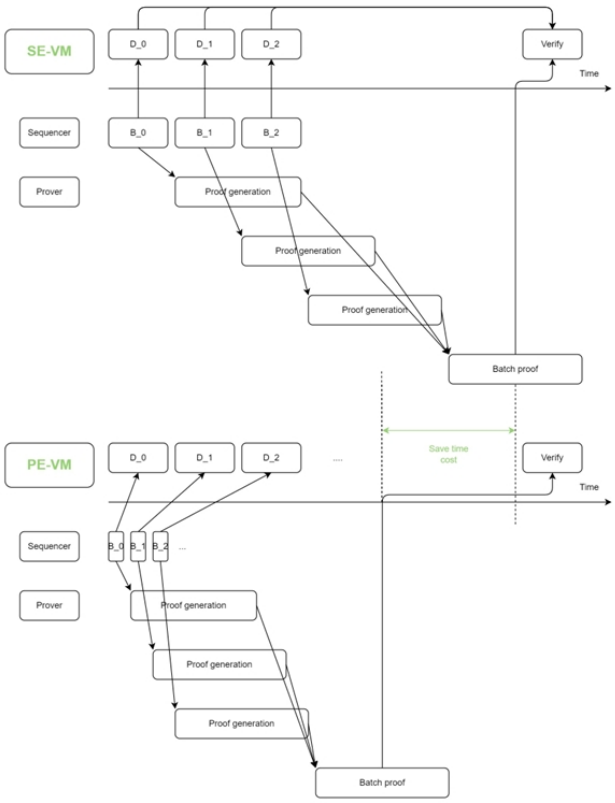


Figure 119: Sequential Execution vs Parallel Execution

state for all transactions. Suppose there are 100 transactions in the blockchain; these 100 transactions can be executed in parallel through a thread pool. Each contract has the same initial world state, and during execution, 100 read-sets and write-sets will be generated, along with 100 new states.

After the pre-execution is finished, the next phase begins. Ideally, if these 100 read-sets and write-sets have no conflicts, they can be directly merged to produce the final world state after all transactions in the block have been executed. However, conflicts in merger transactions usually do not proceed smoothly. The correct way is to first compare the read-sets and write-sets after the execution of the first and second transactions to see if they perform read and write operations on the same account or storage. If there is a conflict, it means that these two transactions have conflicts. At this point, the state after the completion of the first transaction will be used as the starting state for the second transaction, and the second transaction will be re-executed. In this way, as the state machines are continuously merged, the conflict set will continue to accumulate. As long as subsequent transactions have conflicts with previous transactions, they will be executed sequentially until all transactions are completed.

7.1.4 Advantages and Disadvantages of the Solution

Advantages: Improved parallel speed, and enhanced security without using locks. Since the state merging itself is not the focus of time consumption, it basically won't form a bottleneck.

Disadvantages: Putting all transaction validations at the end of the processing phase may lead to performance degradation when there are many transaction conflicts. However, for Ethereum transactions, there are relatively few conflicting items, so the performance is expected to increase by about 3 to 5 times.

7.2 Solidity Compatibility

Solidity is an advanced programming language specifically designed for smart contract development. It is widely used on the Ethereum network and supported by other blockchain projects that are compatible with the Ethereum Virtual Machine (EVM). The following advantages of Solidity make it necessary for some blockchain projects to support it:

- Solidity has a large developer community and user base, with many excellent smart contract projects and cases that can provide reference and inspiration for other blockchain projects. Supporting Solidity can attract these existing developers and users, and increase their activity and liquidity.
- Solidity has a rich development toolset and ecosystem including compilers, debuggers, testing frameworks, code analysis tools, code libraries, etc, which can improve development efficiency and code quality. Supporting Solidity enables developers to enjoy these mature tools and resources thereby enhancing their security and stability.

Therefore, some blockchain projects need to support Solidity to attract more developers and users; enhance their competitiveness and influence; contribute towards the development and innovation of smart contracts.

7.2.1 How OlaVM Supports Solidity?

The Ola compiler generates an abstract syntax tree (AST) by parsing the syntax of Solidity. In Solidity, contracts are composed of various functions. Therefore, the Ola compiler first parses the function definitions of Solidity contracts and converts each function into a corresponding AST node. For each function, the Ola compiler parses the function's parameters, returns values, visibility, and other attributes, and represents them as properties of AST nodes. When parsing the function body, the Ola compiler decomposes the function body into basic statement blocks, such as assignment statements, conditional statements, and loop statements, and converts them into corresponding AST nodes. During this process, the Ola compiler performs semantic analysis to check the correctness of statements and the type matching. For example, when parsing an assignment statement, the Ola compiler checks whether the types on both sides match. If not, a compilation error is generated.

In addition to function definitions, Solidity also supports advanced syntax structures such as structures, enumerations, and events. When parsing these structures, the Ola compiler converts them into corresponding AST nodes and generates the corresponding Ola opcodes. In general, the Solidity parsing process in the Ola compiler can be divided into the following steps:

- Lexical analysis: Decompose Solidity code into basic lexical units (such as keywords, variable names, and numbers).
- Syntax analysis: Combine lexical units into an abstract syntax tree according to Solidity's syntax rules.
- Semantic analysis: Check the correctness and type matching of the syntax tree, and generate the corresponding Ola opcodes.

After parsing, the AST tree is obtained. The process of generating LLVM IR from the AST tree includes the following steps:

- The Ola compiler converts the Solidity AST nodes into corresponding LLVM IR syntax tree nodes and maps the variables, types, functions, and other information in Solidity to the types, global variables, functions, and other elements in LLVM IR.

- After generating the LLVM IR syntax tree, the Ola compiler optimizes it and then compiles it into Ola opcodes, which is the instruction set that OlaVM can execute directly. Specifically, the Ola compiler converts the LLVM IR syntax tree into Static Single Assignment (SSA)[39] form, where each variable is assigned only once, making it easier for subsequent analysis and optimization.

Then, the Ola compiler converts the LLVM IR syntax tree in SSA form into Ola opcodes, which mainly includes the following steps:

- Convert the basic blocks in the LLVM IR syntax tree into basic blocks in Ola opcodes.
- Convert the instructions in the LLVM IR syntax tree into Ola opcodes. Ola opcodes are simpler than LLVM IR instructions and only include basic operations, branching, and jumping instructions.
- Convert the PHI instructions [30] in the SSA form of the LLVM IR syntax tree into PHI instructions in Ola opcodes.

After completing the above process, the Ola compiler generates Ola opcodes that can be executed on OlaVM. By introducing the intermediate representation form LLVM IR, the Ola compiler can compile Solidity into LLVM IR, and then compile LLVM IR into Ola opcodes, making OlaVM support not only Ola language but also many programming languages that can be compiled to LLVM IR, with better flexibility and extensibility.

The above process is shown in the figure120.

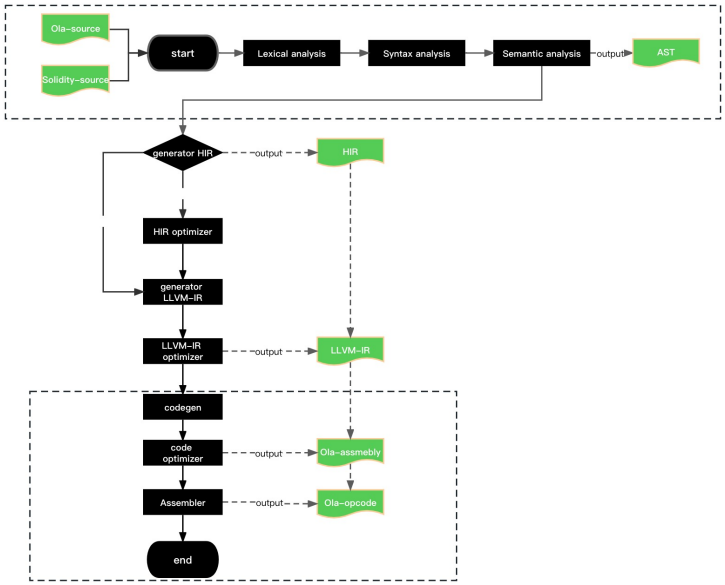


Figure 120: Solidity Compilation Process

7.2.2 Further Discussion

Although compiling Solidity into LLVM IR is feasible, there are many details in between that are worth considering. This is also a key direction that needs improvement.

- Data type mapping: Solidity has many proprietary data types, such as uint, int, address, bytes, bool, etc. When converting Solidity code to LLVM IR, these data types need to be mapped to data types supported by LLVM IR.
- Function calls and message passing: Solidity contracts communicate with each other through message passing

and support both external and internal function calls. When compiling Solidity to LLVM IR, these calls and message-passing mechanisms need to be handled correctly.

- Storage and memory: Solidity has two types of data storage: persistent storage and memory. The compiler needs to be able to correctly handle these storage types and map them to the corresponding data structures in LLVM IR.
- Exception handling: Solidity uses statements like revert, require, and assert to handle exceptions. When compiling Solidity to LLVM IR, these exception-handling mechanisms need to be mapped to the exception-handling structures in LLVM IR.
- Access control and visibility: Solidity supports access control and visibility modifiers, such as private, internal, public, and external. These modifiers need to be correctly converted to access control and visibility structures in LLVM IR.

Fortunately, some of the issues can be solved by defining certain built-in functions and built-in structures, but some problems cannot be directly resolved, such as:

- EVM-specific features: Solidity has some features specific to the Ethereum Virtual Machine (EVM), such as self-destruct, gas-left, and block-hash. These features may not apply to other platforms based on LLVM IR and may require alternative solutions or limitations on their use.
- Full ABI support: Solidity's Application Binary Interface (ABI) is used for encoding and decoding data between contracts. Compiling Solidity to LLVM IR may require an additional implementation to support the full ABI.
- Optimization and Performance: Compiling Solidity to LLVM IR may result in performance loss, as we introduce non-deterministic computation in the Ola language. This solves most of the difficult-to-compute but easy-to-verify problem, which cannot be gained by directly processing Solidity.

8 Glossary

- `ZKP`: Zero-Knowledge Proof.
- `MPC`: Multi-Party Computation.
- `IOP`: Interactive Oracle Proof.
- `ZK-SNARK`: Zero-Knowledge Succinct Non-Interactive Argument of Knowledge.
- `ZK-STARK`: Zero-Knowledge Scalable Transparent Argument of Knowledge.
- `ZKVM`: Zero-Knowledge Virtual Machine.
- `AIR`: Arithmetic Intermediate Representations.
- `FRI`: Fast Reed-Solomon Interactive Oracle Proof.
- `DEEP-FRI`: Domain Extending for Eliminating Pretenders of FRI.

Bibliography

[1] **Aleo**. URL: https://www.aleo.org/.

[2] **Aleo snarkVM code repository**. URL: https://github.com/AleoHQ/snarkVM/tree/testnet3.

[3] **Amdahl's law**. URL: https://en.wikipedia.org/wiki/Amdahl%27s_law.

[4] **Aptos**. URL: https://aptoslabs.com/.

[5] **Arbitrum**. URL: https://arbiscan.io/.

[6] **Aztec**. URL: https://aztec.network/.

[7] **Aztec3**. URL: https://docs.aztec.network/aztec3.

[8] Eli Ben-Sasson et al. **DEEP-FRI: Sampling Outside the Box Improves Soundness**. Cryptology ePrint Archive, Paper 2019/336. https://eprint.iacr.org/2019/336. 2019. URL: https://eprint.iacr.org/2019/336.

[9] **BIP 32**. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[10] Sean Bowe et al. **Zexe: Enabling Decentralized Private Computation**. Cryptology ePrint Archive, Paper 2018/962. https://eprint.iacr.org/2018/962. 2018. URL: https://eprint.iacr.org/2018/962.

[11] **BSC**. URL: https://bscscan.com/.

[12] **BTC**. URL: https://bitcoin.org/en/.

[13] **Cairo1.0**. URL: https://medium.com/starkware/open-sourcing-cairo-1-0-b3100a664bb0.

[14] **Circom**. URL: https://docs.circom.io/circom-language/basic-operators/.

[15] **Dash**. URL: https://www.dash.org/.

[16] **DSL**. URL: https://github.com/TonnnnnnyLiu/Books/blob/master/%E9%A2%86%E5%9F%9F%E7%89%B9%E5%AE%9A%E8%AF%AD%E8%A8%80.pdf.

[17] **Efficient Private Delegation of zkSNARK Provers**. URL: https://www.youtube.com/live/SCIuwh9ya8U?feature=share&t=1263.

[18] **Ethereum**. URL: https://ethereum.org/en/.

[19] **Etherscan chart**. URL: https://etherscan.io/chart/tx.

[20] Ariel Gabizon and Zachary J. Williamson. **plookup: A simplified polynomial protocol for lookup tables**. Cryptology ePrint Archive, Paper 2020/315. https://eprint.iacr.org/2020/315. 2020. URL: https://eprint.iacr.org/2020/315.

[21] **Gadget**. URL: https://zcash.github.io/halo2/user/gadgets.html.

[22] **Goldilocks**. URL: https://cronokirby.com/notes/2022/09/the-goldilocks-field/.

[23] **Grin**. URL: https://grin.mw/.

[24] **Manta network**. URL: https://manta.network/.

[25] **Monero**. URL: https://www.getmonero.org/.

[26] **Noir**. URL: https://noir-lang.org/index.html.

[27] **Ola lang**. URL: https://github.com/Sin7Y/ola-lang.

[28] **OlaVM**. URL: https://github.com/Sin7Y/olavm.

[29] **Optimism**. URL: https://optimistic.etherscan.io/.

[30] **PHI Instruction**. URL: https://llvm.org/docs/LangRef.html#phi-instruction.

[31] **Pil**. URL: https://github.com/0xPolygonHermez/pil-stark.

[32] **Plonky2 Code repository**. URL: https://github.com/Sin7Y/plonky2.

[33] **Polygon**. URL: https://polygon.technology/.

[34] **Polygon Hermez**. URL: https://github.com/0xPolygonHermez/zkevm-techdocs.

[35] **POS**. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[36] **POW**. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/.

[37] **Solana**. URL: https://solana.com/zh.

[38] **Solidity**. URL: https://docs.soliditylang.org/en/latest/.

[39] **SSA**. URL: https://en.wikipedia.org/wiki/Static_single-assignment_form.

[40] **The halo2 Book**. URL: https://zcash.github.io/halo2/design/gadgets/ecc/addition.html.

[41] **The Merge upgrade**. URL: https://ethereum.org/en/roadmap/merge/.

[42] **Tornado cash**. URL: https://www.sanctions.io/blog/ofacs-tornado-cash-sanctions.

[43] **Web3**. URL: https://en.wikipedia.org/wiki/Web3.

[44] Alex Luoyuan Xiong et al. **VERI-ZEXE: Decentralized Private Computation with Universal Setup**. Cryptology ePrint Archive, Paper 2022/802. https://eprint.iacr.org/2022/802. 2022. URL: https://eprint.iacr.org/2022/802.

[45] **Zcash**. URL: https://z.cash/.

[46] **Zcash Protocol Specification NU5**. URL: https://zips.z.cash/protocol/protocol.pdf.

[47] **ZIP 32**. URL: https://zips.z.cash/zip-0032.

[48] **zk.money**. URL: https://zk.money/.

[49] **Zksync**. URL: https://zksync.io/.