zkLLVM Circuit Compiler

Zero-Knowledge Proof Systems Circuits Compiler.

Nikita Kaskov
=nil; Foundation
nbering@nil.foundation

Mikhail Komarov
=nil; Foundation
nemo@nil.foundation

Mikhail Aksenov
=nil; Foundation
maksenov@nil.foundation

April 6, 2024

Abstract

This document describes zkLLVM toolchain – an LLVM-based high-level programming languages compiler into provable computations protocols input representations (i.e. zero-knowledge proof system circuits or fully-homomorphic encryption operations circuits). It can be used to generate input for any arbitrary zero-knowledge proof system or protocol, which accepts input data in form of algebraic circuits. Its initial impelemtation is supposed to be based on top of Placeholder proof system, but is not limited to it.

Such a toolchain, as its usage grows, will allow to build new zk- (zero- knowledge) projects with minimal effort by reusing prepared set of components, which can always be extended in the form of a community-driven project.

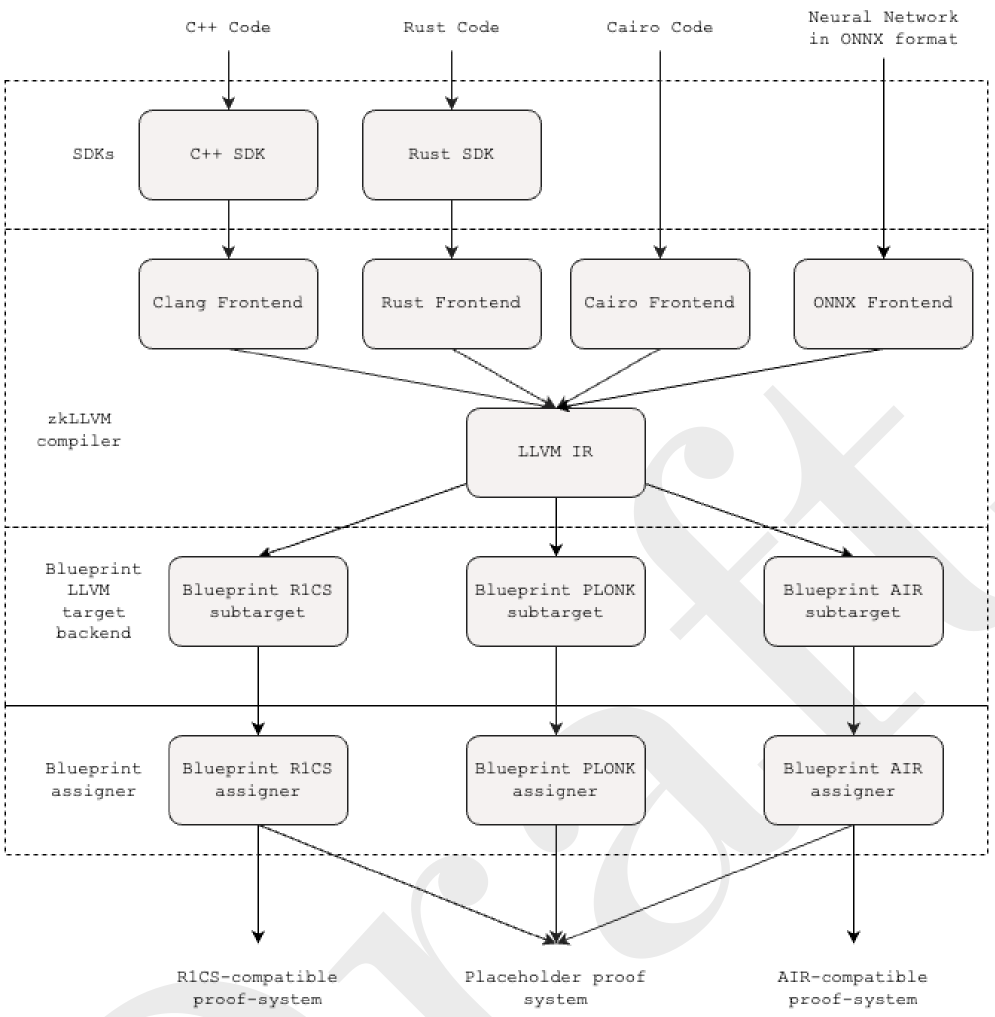
1 Introduction

Provable computations experience noticeable growth as they allow to decrease trust assumptions toward the execution actor, thus providing a way to build complex application logic. Unfortunately, most of the ways to define provable logic require to describe the algorithm via some custom DSL (domain specific language), which significantly decreases zk application sustainability and upgradability, slows down the implementation and increases time to market for projects. It also requires to be familiar with the specifics of many different DSLs as each zk-project has its own.

Based on these reasons zkLLVM circuit compiler project was designed. It is intended to transform arbitrary algorithm implemented in mainstream Turing-complete language into an algebraic **circuit** so it further could be used as input by **Placeholder** proof system [**placeholder-spec**] or any other arithmetization-compatible proof system. The compiler is

zkLLVM circuit compiler is implemented as an extension of the LLVM (Low Level Virtual Machine) project [**llvm-website**], uses its perfect three-phase compiler design, which allows to implement a separate compiler backend for the flexible target circuit architecture and get support of many existing source language frontends with minimal modifications on the frontend side.

Hereinafter we use **Blueprint** as the name of algebraic circuit-specific LLVM-compatible compilation target.



The compiler toolchain requires a set of circuit **components** providing arithmetizaion constraints definition. Current implementation is based on the [**zkllvm-blueprint-git**] circuit components library; instead of it, any other components library of required components can be used.

Full list of required and optional components is given in the appendix. The ultimate goal of the project is to develop a set of universal interfaces and components witch can be reused for different purposes. The components set might be extended over time with new custom components, including, for example, ML-specific functions or cryptographic hashes optimized for particular target.

In terms of zkLLVM tool-chain, there is no difference between building a zk-Oracle, a consensus zk-proof or a zk-VM. All of these are being built from the same set of components. If the set of components is stable and well audited, then the task to insure in the security of the app is pretty straightforward - only the code in the high-level language requres an audit.

1.1 Short introduction to Placeholder

Placeholder is a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) based on PlonK-style arithmetization. Its internal components, such as commitment schemes and types of arithmetization, are replaceable and configurable. Low-level Placeholder circuits can adapt to selected parameters, such as table size, date degree, and lookup options. These qualities enable the flexible configuration of Placeholder with trade-offs between circuit parameters, trust assumptions, and efficiency of proof generation. Due to this flexibility, Placeholder can accommodate particular cases, consistently achieving efficient results.

1.2 Arithmetizations

To prove an algorithm, it needs to be represented in a form compatible with provable computation protocol. This form is called **arithmetization** and can be one of those:

- R1CS
- PLONK
- PLONK with Custom Gates
- AIR

Different arithmetizations represent components logic differently. What takes hundreads of constraints in PLONK arithmetization can take hundreads thousands in R1CS. But in terms of zkLLVM no higher level are being affected by the arithmetization and do not require any changes when arithmetization changes.

1.3 Plonk Arithmetization

Here we describe our instantiation of PLONK with custom gates.

The computation sequence that needs to be proved is represented as *Circuit*. The Circuit is defined by fixed parameters and various sets of constraints (*Basic*, *Copy*, *Lookup*) on the *Execution trace* of the computation. Note that it is assumed that the Verifier does not know the entire Execution trace. A more formal description is as follows.

The Execution trace stores values used during computations. It is represented by a rectangular matrix $\boldsymbol{\mathcal{T}}$ (which we'll refer as *Table*) with $\mathsf{N}_{\mathsf{rows}}$ rows and $\mathsf{N}_{\mathsf{col}}$ columns:

$$\boldsymbol{\mathcal{T}} = \left[\boldsymbol{\tau}_0^T, \ldots, \boldsymbol{\tau}_{\mathsf{N}_{\mathsf{col}}-1}^T\right].$$

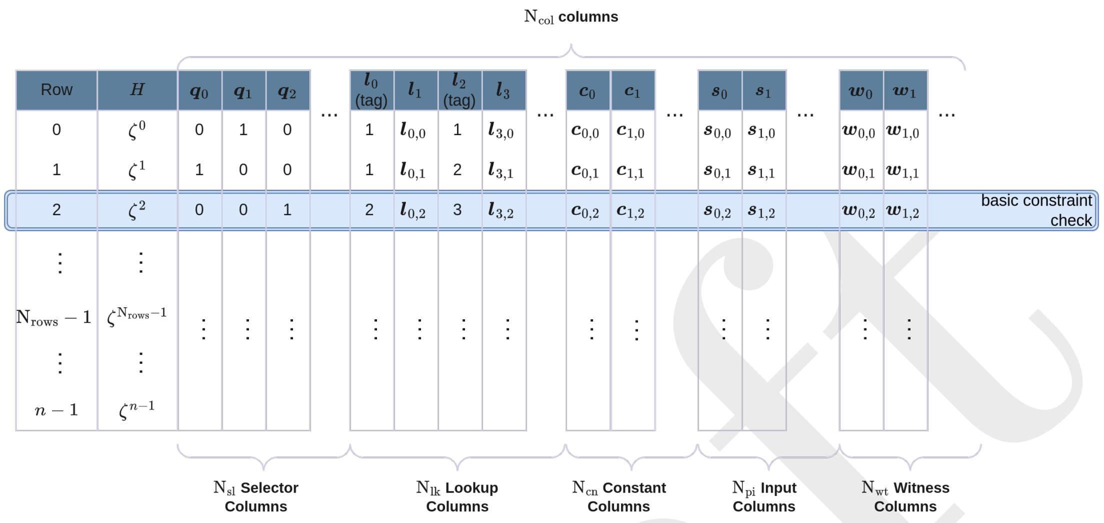
There are 5 types of columns $\boldsymbol{\mathcal{T}}$.

- *Witness* columns contain witness input and intermediate calculations.
  Witness columns differ between proof instances (because they depend on input). They are not known to the verifier. We denote witness columns by $\boldsymbol{w} = (\boldsymbol{w}_0, \ldots, \boldsymbol{w}_{\mathsf{N}_{\mathsf{wt}}-1})$ where $\mathsf{N}_{\mathsf{wt}} \in \mathbb{N},\ \boldsymbol{w}_i \in \mathbb{F}^{\mathsf{N}_{\mathsf{rows}}}$ for $i = 0, \ldots, \mathsf{N}_{\mathsf{wt}} - 1$.
- *Public* columns contain public input for computation.
  Public Columns differ between proof instances (because they depend on input). They are known to the verifier. We denote public columns by $\boldsymbol{s} = (\boldsymbol{s}_0, \ldots, \boldsymbol{s}_{\mathsf{N}_{\mathsf{pi}}-1})$ where $\mathsf{N}_{\mathsf{pi}} \in \mathbb{N},\ \boldsymbol{s}_i \in \mathbb{F}^{\mathsf{N}_{\mathsf{rows}}}$ for $i = 0, \ldots, \mathsf{N}_{\mathsf{pi}} - 1$.
- *Constant* columns contain circuit-depended data.
  Constant columns do not differ between proof instances. They are known to the verifier. We denote constant columns by $\boldsymbol{c} = (\boldsymbol{c}_0, \ldots, \boldsymbol{c}_{\mathsf{N}_{\mathsf{cn}}-1})$ where $\mathsf{N}_{\mathsf{cn}} \in \mathbb{N},\ \boldsymbol{c}_i \in \mathbb{F}^{\mathsf{N}_{\mathsf{rows}}}$ for $i = 0, \ldots, \mathsf{N}_{\mathsf{cn}} - 1$.
- *Selector* columns define which rows of the Table the basic constraint is applied.
  Selectors' values can be only ones or zeroes. We denote selector columns by $\boldsymbol{q} = (\boldsymbol{q}_0, \ldots, \boldsymbol{q}_{\mathsf{N}_{\mathsf{sl}}-1})$ where $\mathsf{N}_{\mathsf{sl}} \in \mathbb{N},\ \boldsymbol{q}_i \in \{0,1\}^{\mathsf{N}_{\mathsf{rows}}}$ for $i = 0, \ldots, \mathsf{N}_{\mathsf{sl}} - 1$.
- *Lookup* columns define tables for membership testing. We denote lookup columns by $\boldsymbol{l} = (\boldsymbol{l}_0, \ldots, \boldsymbol{l}_{\mathsf{N}_{\mathsf{lk}}-1})$ where $\mathsf{N}_{\mathsf{lk}} \in \mathbb{N},\ \boldsymbol{l}_i \in \mathbb{F}_p{}^{\mathsf{N}_{\mathsf{rows}}}$ for $i = 0, \ldots, \mathsf{N}_{\mathsf{lk}} - 1$.

Assume without loss of generality that

$$\boldsymbol{\mathcal{T}} = \left[\boldsymbol{q}_0^T, \ldots, \boldsymbol{q}_{\mathsf{N_{sl}}-1}^T, \boldsymbol{l}_0^T, \ldots, \boldsymbol{l}_{\mathsf{N_{lk}}-1}^T, \boldsymbol{c}_0^T, \ldots, \boldsymbol{c}_{\mathsf{N_{cn}}-1}^T, \boldsymbol{s}_0^T, \ldots, \boldsymbol{s}_{\mathsf{N_{pi}}-1}^T, \boldsymbol{w}_0^T, \ldots, \boldsymbol{w}_{\mathsf{N_{wt}}-1}^T\right],$$

where $\mathsf{N_{sl}} + \mathsf{N_{lk}} + \mathsf{N_{cn}} + \mathsf{N_{pi}} + \mathsf{N_{wt}} = \mathsf{N_{col}}$ (see figure 1).



$\mathrm{N_{col}}$ columns
Row
H
$\boldsymbol{q}_0$
$\boldsymbol{q}_1$
$\boldsymbol{q}_2$
$\boldsymbol{l}_0$ (tag)
$\boldsymbol{l}_1$
$\boldsymbol{l}_2$ (tag)
$\boldsymbol{l}_3$
$\boldsymbol{c}_0$
$\boldsymbol{c}_1$
$\boldsymbol{s}_0$
$\boldsymbol{s}_1$
$\boldsymbol{w}_0$
$\boldsymbol{w}_1$
0 ζ^0 0 1 0 1 $\boldsymbol{l}_{0,0}$ 1 $\boldsymbol{l}_{3,0}$ $\boldsymbol{c}_{0,0}$ $\boldsymbol{c}_{1,0}$ $\boldsymbol{s}_{0,0}$ $\boldsymbol{s}_{1,0}$ $\boldsymbol{w}_{0,0}$ $\boldsymbol{w}_{1,0}$
1 ζ^1 1 0 0 1 $\boldsymbol{l}_{0,1}$ 2 $\boldsymbol{l}_{3,1}$ $\boldsymbol{c}_{0,1}$ $\boldsymbol{c}_{1,1}$ $\boldsymbol{s}_{0,1}$ $\boldsymbol{s}_{1,1}$ $\boldsymbol{w}_{0,1}$ $\boldsymbol{w}_{1,1}$
2 ζ^2 0 0 1 2 $\boldsymbol{l}_{0,2}$ 3 $\boldsymbol{l}_{3,2}$ $\boldsymbol{c}_{0,2}$ $\boldsymbol{c}_{1,2}$ $\boldsymbol{s}_{0,2}$ $\boldsymbol{s}_{1,2}$ $\boldsymbol{w}_{0,2}$ $\boldsymbol{w}_{1,2}$
basic constraint check
$\mathrm{N_{rows}} - 1$ $\zeta^{\mathrm{N_{rows}}-1}$
$n-1$ ζ^{n-1}
$\mathrm{N_{sl}}$ Selector Columns
$\mathrm{N_{lk}}$ Lookup Columns
$\mathrm{N_{cn}}$ Constant Columns
$\mathrm{N_{pi}}$ Input Columns
$\mathrm{N_{wt}}$ Witness Columns

Figure 1: *Structure of execution trace table* $\boldsymbol{\mathcal{T}}$

Sets of constraints define relationships between values of $\boldsymbol{\mathcal{T}}$.

- *Basic* constraints are expressions for table values of a certain row (and possibly several adjacent ones).
  Let $\mathbf{o}$ – set of offsets for the row indices involved in the constraint. Usually, $\mathbf{o} = \{-1, 0, 1\}$. The j-th constraint ($0 \leq j < \mathsf{C_{bs}}$) is given in the form of a multivariate polynomial $\mathcal{C}'_j$ of total degree $\mathsf{C_{dg}}$ over the table values:

  $$\mathcal{C}'_j(\{\boldsymbol{w}_{0,i+o'}\}_{o' \in \mathbf{o}}, \ldots, \{\boldsymbol{w}_{\mathsf{N_{wt}}-1,i+o'}\}_{o' \in \mathbf{o}}) = 0, \text{ where } i - \text{number of row}, i \in [\mathsf{N_{rows}}] \quad (1)$$

  An example of a basic constraint could be the following:

  $$\boldsymbol{w}_{j,i} \cdot \boldsymbol{w}_{j+1,i} + \boldsymbol{w}_{j+1,i+1} - 1 = 0. \quad (2)$$

  Selectors are used to include/exclude a Basic Constraint check to/from the Row. Selectors are included as a part of the assertion to do this. A set of Basic Constraints used with the same Selector is called *Gate*. A gate may contain one or more constraints. Each Row has to satisfy all Gates of the Circuit.
- *Copy* constraints defines equality assertions between Cells.
  Such constraints have following form $\boldsymbol{\mathcal{T}}_{i,j} = \boldsymbol{\mathcal{T}}_{i',j'}$.
- *Lookup* constraints assert that the chosen tuples of cells of the Table are equal to some rows in lookup table L.
  Note that Lookup Constraint does not define the precise place of the tuple in the Lookup Table. It is the main difference between Lookup and Copy constraints. Such constraints have following form $(\boldsymbol{\mathcal{T}}_{i,j}, \boldsymbol{\mathcal{T}}_{i+1,j}, \boldsymbol{\mathcal{T}}_{i+2,j+1}) \in L$.

In order to arithmetize basic constraints we need to define how gates univariate polynomials are constructed from constraints. Remind that gates contains a set of constraints with the same selector.

In practice, the table with $\mathsf{N}_{\mathsf{rows}}$ rows is padded to get the number of rows $n = 2^d$ for some $d \in \mathbb{N}$.

Let $H \subseteq \mathbb{F}_p^*$ – cyclic group with generator ζ: $H = \{\zeta^0, ..., \zeta^{n-1}\}$. The bijection $[n] \to H$ make it possible to consider $\boldsymbol{\tau}_i$ as values of some polynomial on evaluation domain H. For each table column we define interpolant polynomial from $\mathbb{F}_p^{<n}[X]$ using Lagrange basis. For j-th witness column $j \in [\mathsf{N}_{\mathsf{wt}}]$ we have

$$w_j(x) = \sum_{i=0}^{n-1} \boldsymbol{w}_{j,i} \cdot L_i(x).$$

Selector $\{s_i(x)\}$, constant $\{c_i(x)\}$, lookup-tables $\{l_i(x)\}$ and public input $\{s_i(x)\}$ polynomials are given analogously.

Each constraint $\mathcal{C}'$ (see equation 1) may be written as polynomial

$$\mathcal{C}_j(X) = \mathcal{C}'_j(\{w_j(\zeta^{o'} \cdot X)\}_{o' \in \mathbf{o},\ j \in [\mathsf{N}_{\mathsf{wt}}]}),\ \ X \in H.$$

Then example 2 has the following form $w_j(\zeta^i) \cdot w_{j+1}(\zeta^i) + w_{j+1}(\zeta^{i+1}) - 1 = 0$.

For constraint polynomials $\mathcal{C}_0, \ldots, \mathcal{C}_{k-1}$ used in i-th gate and corresponding random challenge θ_i, we can represent i-th gate as:

$$\mathcal{G}_i(X) = q_i(X) \cdot (\theta_i^0 \mathcal{C}_0(X) + \cdots + \theta_i^{k-1} \mathcal{C}_{k-1}(X)).$$

For details on Copy and Lookup constraints arithmetization, we refer the reader to sections the **Placeholder** specification document.

2 SDKs and Frontend

2.1 Frontend

Since zkLLVM project is based on the LLVM, it is possible to use it with different frontends. The only required frontend features are custom types (such as Galois field elements and multiprecision integer number) and intrinsics support. Full list of custom types can be found in the IR specification section 3.1.

2.1.1 SDKs

Time-consuming provable computations algorithms implementations mostly take place when some advanced mathematics apears. Common use-cases include:

- Cryptographic signatures, ciphers and hashes.
- Matrix arithmetic.
- Provable random generators.
- Key derivation function.
- Verifiable delay functions.
- ML and data analysis funcitons

Since such logic is commonly-used and many of these algorithms operate with specific data types, it makes sence to put them all in the SDK by implementing with use of types usages corresponding to zkLLVM IR custom types 3.1.

For C++ code such an SDK is crypto3 cryptography library [**crypto3-suite-git**], which contains a wide list of compatible with zkLLVM advanced mathematics algorithms implementations.

3 Intermediate representation

3.1 Custom types support

The following types assumed to be defined in the IR to represent types, which can be efficently operated by **Blueprint** target instructions:

3.2 Galois field element types

Galois field element type for all supported by algebra [**crypto3-algebra-git**] field types.

Galois field element type represents arithmetic logic by some prime number, such as BLS12-381 or BN-128 base fields.

3.3 Multiprecision modular number types

Multiprecision modular number is a fixed-precision type for modular arithmetic. It has to support numbers by modulus with bitness higher than 128 bits.

Since Galois field by a prime number has similar behaviour as modular number arithmetic by the modulus of this prime number, it might make sence to define multiprecision modular number type only for some common use-cases, which are not Galois fields. List of common is to be defined and can include different powers of two.

3.4 Curve group element types

zkLLVM IR has to include curve group element types for all supported by algebra [**crypto3-algebra-git**] curves types, for all forms and coordinates. Forms and coordinates description can be found in [**hyperelliptic**].

3.5 Multiprecision integer types

Multiprecision integral number type for extended bitness. For example, we need uint256 to efficiently represent EVM-compatible instructions.

4 Blueprint Assigner

Blueprint assigner emits gates list and assignment table in a form described in 1.2.

It uses components descriptions from blueprint module [**crypto3-blueprint-git**], which define constraints logic for every instruction, and contain information about assignments emiting. Each **component** follows trait described in the section below.

4.1 Assigner target parameters

Arithmetization parameters are hihgly tunable and, depending on the particular use case, it may be required to have the emited circuit being limited by some metric, since some circuit params directly affect the computational complexity of a **Placeholder** prover later 1.1. For PLONK subtarget it may be limitation by amount of witness N_w or constant N_c columns, or by assignment rows amount N_rows.

Such limitations are crucial for the instruction emition and assignment generation processes, since they directly affect physical variables allocation and possible late optimizations.

Common subtargets description for PLONK arithmetization may look this way 1.3:

- $N_w \leq 15$: Witness amount N_w for every gate and the whole circuit table (a.k.a. witness columns amount) is not higher than 15, other columns amounts are not limited. Rows amount is not limited.
- $N_w \leq 9$: Witness amount N_w for every gate and the whole circuit table (a.k.a. witness columns amount) is not higher than 9, other columns amounts are not limited. Rows amount is not limited.
- $\forall c \in \{constraints(g) | g \in \{Gates\}\} : degree(c) \leq n$: Constraints degree in circuit gates is not higher than some predefined n.

4.2 Blueprint field

Since blueprint **component** and the whole circuit presents all the logic in form of constraints, it operates with variables holding values in some Galois field. Such a field is fixed for specific instance of an **arithmetization** and within this document we refer to it as **Blueprint field**.

4.3 Component trait

Component is an internal blueprint entity representing part of circuit logic for a backend instruction. Each instruction has one corresponding component, but many instructions may be reflected into one blueprint component. Thus, a component represents some more or less atomic part of the final circuit.

Component operates with **blueprint variables**, wich are absolute references to **Assignment table** cells. It takes blueprint variables as input and results blueprint variables as output. Since all the cells of an **Assignment table** are the elements of a fixed prior to one instance blueprint field $\mathbb{F}_{bp}$, all components also operate with of this blueprint field $\mathbb{F}_{bp}$ elements.

Every components has to implement two functions: one for constraints/gates generation, one for corresponding part of assignment table generation.

4.3.1 PLONK component

In the context of the circuit, PLONK component represents a set of gates. Each gate is a set of constraints united under common selector. As described in 1.3, selector defines on which rows of **Assignment table** should the corresponding constraint be satisfied.

$$\texttt{component} = \{\texttt{gate}_i, i = \overline{0, \texttt{gates\_amount}}\},$$

$$\forall i = \overline{0, \texttt{gates\_amount}} : \texttt{gate}_i = \langle \texttt{selector}, \{\texttt{constraint}_j, j = \overline{0, \texttt{constraints\_amount}^i}\} \rangle$$

5 Instructions sets

Instructions represent different transforms on IR types 3.1, mostly - arithmetic and logical ones.

A common design pattern is to also add many intrinsics, such as hashes or signatures. Nevertheless, in zkLLVM all the complex logic circuits assumed to be built from the given low-level instructions. This approach is much more difficult to implement, since it requires to have precise understanding of arithmetization's features. But it reduces the price of SDKs extension, making it easier to widen zkLLVM usages.

Though zkLLVM has some extra instructions, which are pretty high-level. Such instructions mostly include matrix-based algorithms, since they are the most difficult to present in form of constraints.

During the lowering process every instruction exist in different states for different passes. Here is full list of such states:

- Blueprint instruction with virtual IR types arguments;
- Blueprint instruction with virtual legalized types arguments;
- Parameterized blueprint instruction with allocated variables arguments;
- List of final circuit gates together with part of assignment table.

5.1 Branch instructions

For some **arithmetizations**, branch instructions may not present in form of some separate circuit component.

For PLONK **arithmetization** unsupported service instructions are:

- JUMP and conditional JUMP;
- CALL

Despite the fact that these instructions are not presented in form of components, they are being handeled by specific arithmetization features. Thus PLONK subtarget branch instructions are handeled by selectors logic and require to keep some additional information until final selectors expansion pass (6.6)

5.2 Blueprint instruction with virtual IR types arguments

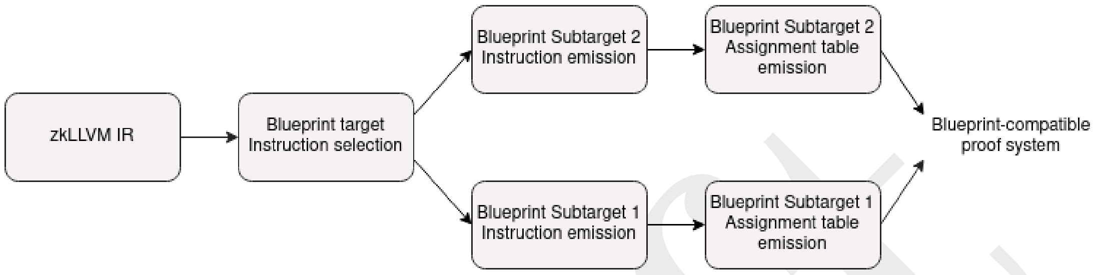
5.3 Blueprint instruction with virtual legalized types arguments

Every instruction's argument is a circuit variable. The value of this variable is always an element of **blueprint field**, because it's used in constraints operating on **blueprint field** elements.

5.4 Parameterized blueprint instruction with allocated variables arguments;

5.5 Circuit component gates together with part of assignment table.

6 PLONK lowering passes


zkLLVM IR
Blueprint target Instruction selection
Blueprint Subtarget 2 Instruction emission
Blueprint Subtarget 2 Assignment table emission
Blueprint Subtarget 1 Instruction emission
Blueprint Subtarget 1 Assignment table emission
Blueprint-compatible proof system

6.1 Instruction selection

Pass type: mandatory.
Run-time input: not required.
Input: zkLLVM IR
Ouput: list of Blueprint instructions with virtual IR types arguments

During instruction selection process, LLVM IR instructions are being substituted by Blueprint instructions, which are supported by particular subtarget.

Since zkLLVM uses a custom version of LLVM IR, some custom types 3.1 are supported during the Instruction selection process. Instruction selection is mostly based on the type of argument passed. Thus, arithmetic instructions for field elements are being choosen for LLVM IR field elements arithmetic, and the same is true for curve element arithmetic.

6.2 Types legalization

Pass type: mandatory.
Run-time input: not required.
Input: list of Blueprint instructions with virtual IR types arguments
Ouput: list of Blueprint instructions with virtual legalized types arguments

The final circuit operates on variables of some blueprint field $\mathbb{F}_{bp}$ element type 1.3. Even the field element zkLLVM IR internal types must be legalized, if they differ from the blueprint field $\mathbb{F}_{bp}$ element type (they might be the same – then no legalization is needed). The amount and algorithm of legalization depends on how does the origin zkLLVM type fit into blueprint field $\mathbb{F}_{bp}$ element type.

That means, that all instructions input types need to be legalized from zkLLVM IR internal types (3.1) into $\mathbb{F}_{bp}$ field element type:

6.2.1 Native Galois field element legalization

The simplest case is native Galois $\mathbb{F}_{origin}$ field element legalization. Since this type matches blueprint field $\mathbb{F}_bp$ element type, it does not require any transformations.

$$\mathbb{F}_{origin} == \mathbb{F}_{bp} \Rightarrow$$
$$legalize\_it : \mathbb{F}_{origin} \rightarrow \mathbb{F}_{bp}$$

6.2.2 Non-native Galois field element legalization

Non-native Galois field element legalization algorithm depends on the bit size of the origin field type and of the blueprint field $\mathbb{F}_bp$ element type.

$$\mathbb{F}_{origin} \neq \mathbb{F}_{bp} \Rightarrow$$
$$legalize\_it : \mathbb{F}_{origin} \rightarrow \{\mathbb{F}_{bp}\}^n, \text{ -- n depends on fields relative bit size}$$

6.2.3 Native curve group element in 2-coordinates form legalization

Native curve group element in 2-coordinates form type: two blueprint field $\mathbb{F}_{bp}$ elements.

$$\mathbb{C}^{affine}_{origin}, base\_field(C_{origin}) \neq \mathbb{F}_{bp} \Rightarrow,$$
$$legalize\_it : \mathbb{C}^{affine}_{origin} \rightarrow \{\mathbb{F}_{bp}\}^2$$

6.2.4 Native curve group element in 3-coordinates form legalization

Native curve group element in 3-coordinates form type: three blueprint field $\mathbb{F}_{bp}$ elements;

$$\mathbb{C}^{non-affine}_{origin}, base\_field(C_{origin}) \neq \mathbb{F}_{bp} \Rightarrow,$$
$$legalize\_it : \mathbb{C}^{non-affine}_{origin} \rightarrow \{\mathbb{F}_{bp}\}^3$$

6.2.5 Non-native curve group element in 2-coordinates form legalization

Non-native curve group element in 2-coordinates (usually affine)form type: ???

$$\mathbb{C}^{affine}_{origin}, base\_field(C_{origin}) = \mathbb{F}_{origin} \neq \mathbb{F}_{bp} \Rightarrow,$$
$$legalize\_it : \mathbb{C}^{affine}_{origin} \rightarrow \{\mathbb{F}_{bp}\}^n, \text{ -- n depends on fields relative bit size}$$

6.2.6 Non-native curve group element in 3-coordinates form legalization

Non-native curve group element in aby but not affine coordinates form type: ???;

$$\mathbb{C}^{non-affine}_{origin}, base\_field(C_{origin}) = \mathbb{F}_{origin} \neq \mathbb{F}_{bp} \Rightarrow,$$
$$legalize\_it : \mathbb{C}^{non-affine}_{origin} \rightarrow \{\mathbb{F}_{bp}\}^n, \text{ -- n depends on fields relative bit size}$$

6.2.7 Multiprecision integral number legalization

Multiprecision integral number type: ???.

6.3 Instructions de-duplication

Pass type: optimization.
Run-time input: not required.
Input:
Ouput: de-duplicated instanced instructions list.

One of the main advantages of PLONK components is that it's gates may be reused. Gates of one component are fixed within the arithmetization and usually different components do not implement same gates (otherwise such gates may be carry out into a separate reusable component).

Once proof system and arithmetization params have been choosen (**??**), set of instruction instances is fixed.

Write about rows allocation and selectors

If an instruction is used more than once in the code, then these usings require some form of aggregation. During the de-duplication pass, all repeating instances of an instruction are being substituted by `CALL` service instruction.

6.4 Components selectors preprocessing

Pass type: optimization.
Run-time input: not required.
Input: full list of instanced instructions.
Ouput: set of preprocessed selector tables.

Where is columns information saved?

Apart from de-duplication of repeating **Blueprint instructions** in the generated code, another form of de-duplication is to be made.

Every used more or less comlex **Blueprint component** wraps inside itself generation another underlying **components**. These wrapped **components** are also requirred to be taken into account during result selectors aggregation.

De-duplication is made via used **Blueprint components** call-graph visiting. All wrapped instances of the same **component** (**component** is defined by used columns set) are being aggregated into preprocessed selectors.

6.5 Assignment table emission

Pass type: mandatory.
Run-time input: required.
Input:
Ouput: PLONK assignment table.

6.6 Selectors post-processing

Pass type: mandatory.
Run-time input: required.
Input: set of preprocessed selector tables, de-duplicated instanced instructions list.
Ouput: aggregated selector table, full circuit without service.

Some part of preimage code may require run-time input before being transformed into final circuit:

- Conditional loops;
- Conditional branching (if, switch etc.);

7 Aggregated mode

Placeholder proof system is able to work in aggregated mode, and so does the zkLLVM toolchain. In that mode, user marks parts of code for different provers in the same manner as it usually is being done for parallelization. After that, the generated table is being devided in chunks corresponding to different provers. Apart from the part of the assignment table corresponding to the prover's local computations, additional shared part of the table is being created, where the data intended to be used by all provers are located.

7.1 Assignment table partitioning for aggregated mode

On the higher level (for C++ code) user denotes a new part for aggregation by adding designated label.

To use the aggregation mode user needs to define which prover should execute each blocks of code by adding

```
#pragma zk_multi_prover PROVER_INDEX
```

PROVER_INDEX may have value [0, MAX_NUM_PROVERS). The pragma affects the code block enclosed in {...}. For example:

```
a = b + c;
#pragma zk_multi_prover 1
{
    a = a * 10;
    a = a + c * 10;
}
```

The code under different labels is being designated to different provers. If no label denotes a part of code, than the first available prover is designated to this part.

Each one of aggregated provers owns an instance of assignment table, where all the data for his part of the circuit is being stored.

Following the restriction of the Placeholder proof system, all the inputs and outputs of aggregated sub-circuits are stored in a public column called shared column. All the data accessed by more then one prover are also being stored in that shared column.

8 Lowering extra optimizations

8.1 Components adjustment and collocation

Pass type: optimization.
Run-time input: required.

In some cases it makes sense to put several components constraint into one row to use all its cells. This process is being called collocation within this document. For example purposes, let's assume, that on proof system params tuning step (**??**), 6-witness PLONK arithmetization has been chosen. Both `MNFADD` and `MNFMUL`instructions may be instanced in 3 witness columns form, so both of them can be putted in one row.

In addition to collocation, components may be adjusted to use more or less columns and rows. Represented in polynomial form, component's constraints depend on the number of columns, which are being used to instance the component. The polynomials for different columns amount can also differ in the number of variables and in the number of monomials.

Based on the algorithm and the data structures used in it, the components can be adjusted and then collocated. The algorithm is being implemented as a part of the `Blueprint` library.

8.2 Loops circification

Pass type: optimization.
Run-time input: required.

There are two types of loops: those which have constant number of iterations and those which have variable number of iterations dependent on run-time input.

The first type of loops may be easily circified, which means that all loop iterations are being unrolled and the resulting circuit is being transformed into a circuit without loops.

The loops of second type require more complex approach. The core idea is to circify the loop with the maximum number of iterations and then to modify the instruction inside the loop block to be able to break the loop if the condition is unmet. This is being done by adding additional boolean variable to the loop block, which is being set to 1 if the condition is met and to 0 otherwise. Then the instruction is being modified to be able to break the loop if the boolean variable is set to 0.

The instruction modification is being implemented by adding additional multiplier gate to the instruction, which multiplies the instruction constraint by the boolean variable. Thus, if the boolean variable is set to 0, the instruction constraint is being multiplied by 0 and the instruction is being ignored.

8.3 Branching circification

Pass type: optimization.
Run-time input: required.

Same as loops circification, branching circification is being done by unrolling the branch block and adding additional boolean variable to the block, which is being set to 1 if the condition is met and to 0 otherwise. Then the instruction is being modified to be able to break the loop if the boolean variable is set to 0.

Appendices

A PLONK instructions set

Basic instructions represent essential instructions supported by every arithmetization backend regardless of it's peculiarities.

A.1 Curve addition

Adds two curve group elements A and B in affine coordinates representation into the result one C.

```
CADD v0 v1 .  v2 v3 .  v4 v5
```

Input: [v0, v1] - curve group element in affine coordinates A , [v2, v3] - curve group element in affine coordinates B.

Output: [v4, v5] - curve group element in affine coordinates C.

`v0`: X coordinate of A
`v1`: Y coordinate of A
`v2`: X coordinate of B
`v3`: Y coordinate of B
`v4`: X coordinate of C
`v5`: Y coordinate of C

Circuit

Curve addition is handled by the flexible 15-wire complete addition and doubling circuit defined as follows:

Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	x_1	y_1	x_2	y_2	x_3	y_3	$\texttt{inv}_{x_1}$	$\texttt{inv}_{x_2}$	$\texttt{inv}_{x_2-x_1}$	δ	λ	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Evaluations:

- $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$
- $\texttt{inv}_a = a^{-1}$, if $a \neq 0$, $\texttt{inv}_a = 0$ otherwise
- $\delta = \texttt{inv}_{y_1+y_2}$, if $x_1 = x_2$, $\delta = 0$ otherwise
- $\lambda = \frac{y_2-y_1}{x_2-x_1}$ if $x_1 \neq x_2$, $\lambda = \frac{3x_1^2}{2y_1}$, if $x_1 = x_2$ and$y_1 \neq 0$, otherwise $\lambda = 0$

Constraints (max degree $= 3$**):**

- $(w_2 - w_0) \cdot ((w_2 - w_0) \cdot w_{10} - (w_3 - w_1))$
- $(1 - (w_2 - w_0) \cdot w_8) \cdot (2w_1 \cdot w_{10} - 3w_0 \cdot w_0)$
- $(w_0 \cdot w_2 \cdot w_2 - w_0 \cdot w_2 \cdot w_0) \cdot (w_{10} \cdot w_{10} - w_0 - w_2 - w_4)$
- $(w_0 \cdot w_2 \cdot w_2 - w_0 \cdot w_2 \cdot w_0) \cdot (w_{10} \cdot (w_0 - w_4) - w_1 - w_5)$
- $(w_0 \cdot w_2 \cdot w_3 + w_0 \cdot w_2 \cdot w_1) \cdot (w_{10} \cdot w_{10} - w_0 - w_2 - w_4)$
- $(w_0 \cdot w_2 \cdot w_3 + w_0 \cdot w_2 \cdot w_1) \cdot (w_{10} \cdot (w_0 - w_4) - w_1 - w_5)$
- $(1 - w_0 \cdot w_6) \cdot (w_4 - w_2)$
- $(1 - w_0 \cdot w_6) \cdot (w_5 - w_3)$

- $(1 - w_2 \cdot w_7) \cdot (w_4 - w_0)$
- $(1 - w_2 \cdot w_7) \cdot (w_5 - w_1)$
- $(1 - (w_2 - w_0) \cdot w_8 - (w_3 + w_1) \cdot w_9) \cdot w_4$
- $(1 - (w_2 - w_0) \cdot w_8 - (w_3 + w_1) \cdot w_9) \cdot w_5$

Details The gate uses basic group law formulae. Let $P = (x_1, y_1), Q = (x_2, y_2), R = (x_3, y_3)$ and $R = P + Q$. Then:

- $(x_2 - x_1) \cdot s = y_2 - y_1$
- $s^2 = x_1 + x_2 + x_3$
- $y_3 = s \cdot (x_1 - x_3) - y_1$

For point doubling $R = P + P = 2P$:

- $2s \cdot y_1 = 3x_1^2$
- $s^2 = 2x_1 + x_3$
- $y_3 = s \cdot (x_1 - x_3) - y_1$

A.2 Curve multiplication

Performs scalar multiplication of curve group element A in affine coordinates representation onto a number B. Result is a curve group element in affine coordinates C.

```
CMUL v0 v1 .  v2 .  v3 v4
```

Input: [v0, v1] - curve group element in affine coordinates A , v2 - integral B.
Output: [v3, v4] - curve group element in affine coordinates C.
`v0`: X coordinate of A
`v1`: Y coordinate of A
`v2`: integral B
`v3`: X coordinate of C
`v4`: Y coordinate of C

Circuit

Curve multiplication is handled by the flexible 15-wire variable base scalar multiplication circuit defined as follows:

For $R = [r]T$, $k = \frac{r - 2^{255} - 1}{2}$, $k = [k_0, k_1, \ldots, k_{n-1}]$, $n = 255$ [1],
Let $u = k - 2^{254} - t_p + 2^{130}$ for Vesta curve and $u = k - 2^{254} - t_q + 2^{130}$ for Pallas curve $u = [u_0, u_1, \ldots, u_{43}]$, $u = \sum_{i=0}^{42} u_i \cdot 2^{127 - i \cdot 3} + u_{43}$, $u_i \in \{0, \ldots, 7\}, i = 0, \ldots, 42$, $u_{43} \in \{0, 1\}$:

```
1. $P = [2]T$
2. for $i$ from $n - 1$ to 0:
    2.1 $Q = k_i \, ? \, T : -T$
    2.2 $P = P + Q + P$
```

The first step of the alforithm are verified by the complete addition circuit.

[1]Using the results from https://arxiv.org/pdf/math/0208038.pdf

Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	x_T	y_T	x_0	y_0	$n = 0$	n'	$-$	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
$i+1$	x_5	y_5	b_0	b_1	b_2	b_3	b_4	s_0	s_1	s_2	s_3	s_4	$-$	$-$	$-$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i+58$	x_T	y_T	x_0	y_0	n	n'	$u' = 0$	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
$i+59$	x_5	y_5	b_0	b_1	b_2	b_3	b_4	s_0	s_1	s_2	s_3	s_4	u_0	u_1	u''
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i+100$	x_T	y_T	x_0	y_0	n	n'	u'	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
$i+101$	x_5	y_5	b_0	b_1	b_2	b_3	b_4	s_0	s_1	s_2	s_3	s_4	u_0	u_1	u''
$i+102$	x	y	t_0	t_1	t_2	n'	x_T	y_T	m	e_1	e_2	k	u	$-$	$-$

Let $a_{-1} = 0x224698fc0994a8dd8c46eb2100000000$ be a circuit's value for a scalar -1, $a_0 = 0x200000000000000000000000000000003369e57a0e5efd4c526a60b180000001$ for 0 and $a_1 = 0x224698fc0994a8dd8c46eb2100000001$ for 1. Evaluations:

- b_i are bits of the k, first b_0 is the most significant bit of k, n is an accumulator of b_i.
- $(x_1, y_1) - (x_0, y_0) = (x_0, y_0) + (x_T, (2b_1 - 1)y_T)$
- $(x_2, y_2) - (x_1, y_1) = (x_1, y_1) + (x_T, (2b_1 - 1)y_T)$
- $(x_3, y_3) - (x_2, y_2) = (x_2, y_2) + (x_T, (2b_1 - 1)y_T)$
- $(x_4, y_4) - (x_3, y_3) = (x_3, y_3) + (x_T, (2b_1 - 1)y_T)$
- $(x_5, y_5) - (x_4, y_4) = (x_4, y_4) + (x_T, (2b_1 - 1)y_T)$
- $s_0 = \frac{y_0 - (2b_0 - 1) \cdot y_T}{x_0 - x_T}$
- $s_1 = \frac{y_1 - (2b_1 - 1) \cdot y_T}{x_1 - x_T}$
- $s_2 = \frac{y_2 - (2b_2 - 1) \cdot y_T}{x_2 - x_T}$
- $s_3 = \frac{y_3 - (2b_3 - 1) \cdot y_T}{x_3 - x_T}$
- $s_4 = \frac{y_4 - (2b_4 - 1) \cdot y_T}{x_4 - x_T}$
- $m = (n' - a_{-1}) \cdot (n' - a_0) \cdot (n' - a_1)$
- $t_0 = \texttt{inv}_m$
- $t_1 = \texttt{inv}_{(n' - a_{-1})}$
- $t_2 = \texttt{inv}_{(n' - a_1)}$
- (x, y) - is the circuit's result.

Constraints:

- $\texttt{next}(w_2) \cdot (\texttt{next}(w_2) - 1) = 0$
- $\texttt{next}(w_3) \cdot (\texttt{next}(w_3) - 1) = 0$
- $\texttt{next}(w_4) \cdot (\texttt{next}(w_4) - 1) = 0$
- $\texttt{next}(w_5) \cdot (\texttt{next}(w_5) - 1) = 0$
- $\texttt{next}(w_6) \cdot (\texttt{next}(w_6) - 1) = 0$

- $(w_2 - w_0) \cdot \texttt{next}(w_7) = w_3 - (2 \cdot \texttt{next}(w_2) - 1) \cdot w_1$
- $(w_7 - w_0) \cdot \texttt{next}(w_8) = w_8 - (2 \cdot \texttt{next}(w_3) - 1) \cdot w_1$
- $(w_9 - w_0) \cdot \texttt{next}(w_9) = w_{10} - (2 \cdot \texttt{next}(w_4) - 1) \cdot w_1$
- $(w_{11} - w_0) \cdot \texttt{next}(w_{10}) = w_{12} - (2 \cdot \texttt{next}(w_5) - 1) \cdot w_1$
- $(w_{13} - w_0) \cdot \texttt{next}(w_{11}) = w_{14} - (2 \cdot \texttt{next}(w_6) - 1) \cdot w_1$

- $(2 \cdot w_3 - \texttt{next}(w_7) \cdot (2 \cdot w_2 - \texttt{next}(w_7)^2 + w_0))^2 = (2 \cdot w_2 - \texttt{next}(w_7)^2 + w_0)^2 \cdot (w_7 - w_0 + \texttt{next}(w_7)^2)$

- $(2 \cdot w_8 - \texttt{next}(w_8) \cdot (2 \cdot w_7 - \texttt{next}(w_8)^2 + w_0))^2 = (2 \cdot w_7 - \texttt{next}(w_8)^2 + w_0)^2 \cdot (w_9 - w_0 + \texttt{next}(w_8)^2)$
- $(2 \cdot w_{10} - \texttt{next}(w_9) \cdot (2 \cdot w_9 - \texttt{next}(w_9)^2 + w_0))^2 = (2 \cdot w_9 - \texttt{next}(w_9)^2 + w_0)^2 \cdot (w_{11} - w_0 + \texttt{next}(w_9)^2)$
- $(2 \cdot w_{12} - \texttt{next}(w_{10}) \cdot (2 \cdot w_{11} - \texttt{next}(w_{10})^2 + w_0))^2 = (2 \cdot w_{11} - \texttt{next}(w_{10})^2 + w_0)^2 \cdot (w_{13} - w_0 + \texttt{next}(w_{10})^2)$
- $(2 \cdot w_{14} - \texttt{next}(w_{11}) \cdot (2 \cdot w_{13} - \texttt{next}(w_{11})^2 + w_0))^2 = (2 \cdot w_{13} - \texttt{next}(w_{11})^2 + w_0)^2 \cdot (\texttt{next}(w_0) - w_0 + \texttt{next}(w_{11})^2)$

- $(w_8 + w_3) \cdot (2 \cdot w_2 - \texttt{next}(w_7)^2 + w_0) = (w_2 - w_7) \cdot (2 \cdot w_3 - \texttt{next}(w_7) \cdot (2 \cdot w_2 - \texttt{next}(w_7)^2 + w_0))$
- $(w_{10} + w_8) \cdot (2 \cdot w_7 - \texttt{next}(w_8)^2 + w_0) = (w_7 - w_9) \cdot (2 \cdot w_8 - \texttt{next}(w_8) \cdot (2 \cdot w_7 - \texttt{next}(w_8)^2 + w_0))$
- $(w_{12} + w_{10}) \cdot (2 \cdot w_9 - \texttt{next}(w_9)^2 + w_0) = (w_9 - w_{11}) \cdot (2 \cdot w_{10} - \texttt{next}(w_9) \cdot (2 \cdot w_9 - \texttt{next}(w_9)^2 + w_0))$
- $(w_{14} + w_{12}) \cdot (2 \cdot w_{11} - \texttt{next}(w_{10})^2 + w_0) = (w_{11} - w_{13}) \cdot (2 \cdot w_{12} - \texttt{next}(w_{10}) \cdot (2 \cdot w_{11} - \texttt{next}(w_{10})^2 + w_0))$
- $(\texttt{next}(w_1) + w_{14}) \cdot (2 \cdot w_{13} - \texttt{next}(w_{11})^2 + w_0) = (w_{13} - \texttt{next}(w_0)) \cdot (2 \cdot w_{14} - \texttt{next}(w_{11}) \cdot (2 \cdot w_{13} - \texttt{next}(w_{11})^2 + w_0))$

- $w_5 = 32 \cdot (w_4) + 16 \cdot \texttt{next}(w_2) + 8 \cdot \texttt{next}(w_3) + 4 \cdot \texttt{next}(w_4) + 2 \cdot \texttt{next}(w_5) + \texttt{next}(w_6)$

Constraints for the last 3 rows:

- $w_2 \cdot (w_2 - 1) = 0$
- $w_3 \cdot (w_3 - 1) = 0$
- $w_4 \cdot (w_4 - 1) = 0$
- $w_5 \cdot (w_5 - 1) = 0$
- $w_6 \cdot (w_6 - 1) = 0$

- $(\texttt{prev}(w_2) - \texttt{prev}(w_0)) \cdot w_7 = \texttt{prev}(w_3) - (2 \cdot w_2 - 1) \cdot \texttt{prev}(w_1)$
- $(\texttt{prev}(w_7) - \texttt{prev}(w_0)) \cdot w_8 = \texttt{prev}(w_8) - (2 \cdot w_3 - 1) \cdot \texttt{prev}(w_1)$
- $(\texttt{prev}(w_9) - \texttt{prev}(w_0)) \cdot w_9 = \texttt{prev}(w_{10}) - (2 \cdot w_4 - 1) \cdot \texttt{prev}(w_1)$
- $(\texttt{prev}(w_{11}) - \texttt{prev}(w_0)) \cdot w_{10} = \texttt{prev}(w_{12}) - (2 \cdot w_5 - 1) \cdot \texttt{prev}(w_1)$
- $(\texttt{prev}(w_{13}) - \texttt{prev}(w_0)) \cdot w_{11} = \texttt{prev}(w_{14}) - (2 \cdot w_6 - 1) \cdot \texttt{prev}(w_1)$

- $(2 \cdot \texttt{prev}(w_3) - w_7 \cdot (2 \cdot \texttt{prev}(w_2) - (w_7)^2 + \texttt{prev}(w_0)))^2 = (2 \cdot \texttt{prev}(w_2) - (w_7)^2 + \texttt{prev}(w_0))^2 \cdot (\texttt{prev}(w_7) - \texttt{prev}(w_0) + (w_7)^2)$
- $(2 \cdot \texttt{prev}(w_8) - w_8 \cdot (2 \cdot \texttt{prev}(w_7) - (w_8)^2 + \texttt{prev}(w_0)))^2 = (2 \cdot \texttt{prev}(w_7) - (w_8)^2 + \texttt{prev}(w_0))^2 \cdot (\texttt{prev}(w_9) - \texttt{prev}(w_0) + (w_8)^2)$
- $(2 \cdot \texttt{prev}(w_{10}) - w_9 \cdot (2 \cdot \texttt{prev}(w_9) - (w_9)^2 + \texttt{prev}(w_0)))^2 = (2 \cdot \texttt{prev}(w_9) - (w_9)^2 + \texttt{prev}(w_0))^2 \cdot (\texttt{prev}(w_{11}) - \texttt{prev}(w_0) + (w_9)^2)$
- $\Big(\big(2 \cdot \texttt{prev}(w_{12}) - w_{10} \cdot (2 \cdot \texttt{prev}(w_{11}) - (w_{10})^2 + \texttt{prev}(w_0))\big)^2 - \big((2 \cdot \texttt{prev}(w_{11}) - (w_{10})^2 + \texttt{prev}(w_0))^2 \cdot (\texttt{prev}(w_{13}) - \texttt{prev}(w_0) + (w_{10})^2)\big)\Big) \cdot (\texttt{next}(w_8) \cdot \texttt{next}(w_2)) = 0$
- $\Big(\big(2 \cdot \texttt{prev}(w_{14}) - w_{11} \cdot (2 \cdot \texttt{prev}(w_{13}) - (w_{11})^2 + \texttt{prev}(w_0))\big)^2 - \big(2 \cdot \texttt{prev}(w_{13}) - (w_{11})^2 + \texttt{prev}(w_0))^2 \cdot (w_0 - \texttt{prev}(w_0) + (w_{11})^2)\big)\Big) \cdot (\texttt{next}(w_8) \cdot \texttt{next}(w_2)) = 0$

- $(\texttt{prev}(w_8) + \texttt{prev}(w_3)) \cdot (2 \cdot \texttt{prev}(w_2) - (w_7)^2 + \texttt{prev}(w_0)) = (\texttt{prev}(w_2) - \texttt{prev}(w_7)) \cdot (2 \cdot \texttt{prev}(w_3) - w_7 \cdot (2 \cdot \texttt{prev}(w_2) - (w_7)^2 + \texttt{prev}(w_0)))$
- $(\texttt{prev}(w_{10}) + \texttt{prev}(w_8)) \cdot (2 \cdot \texttt{prev}(w_7) - (w_8)^2 + \texttt{prev}(w_0)) = (\texttt{prev}(w_7) - \texttt{prev}(w_9)) \cdot (2 \cdot \texttt{prev}(w_8) - w_8 \cdot (2 \cdot \texttt{prev}(w_7) - (w_8)^2 + \texttt{prev}(w_0)))$

- $(\texttt{prev}(w_{12}) + \texttt{prev}(w_{10})) \cdot (2 \cdot \texttt{prev}(w_9) - (w_9)^2 + \texttt{prev}(w_0)) = (\texttt{prev}(w_9) - \texttt{prev}(w_{11})) \cdot (2 \cdot \texttt{prev}(w_{10}) - w_9 \cdot (2 \cdot \texttt{prev}(w_9) - (w_9)^2 + \texttt{prev}(w_0)))$
- $\Big((\texttt{prev}(w_{14}) + \texttt{prev}(w_{12})) \cdot (2 \cdot \texttt{prev}(w_{11}) - (w_{10})^2 + \texttt{prev}(w_0))) - ((\texttt{prev}(w_{11}) - \texttt{prev}(w_{13})) \cdot (2 \cdot \texttt{prev}(w_{12}) - w_{10} \cdot (2 \cdot \texttt{prev}(w_{11}) - (w_{10})^2 + \texttt{prev}(w_0)))\Big) \cdot (\texttt{next}(w_8) \cdot \texttt{next}(w_2)) = 0$
- $\Big(((w_1 + \texttt{prev}(w_{14})) \cdot (2 \cdot \texttt{prev}(w_{13}) - (w_{11})^2 + \texttt{prev}(w_0))) - ((\texttt{prev}(w_{13}) - w_0) \cdot (2 \cdot \texttt{prev}(w_{14}) - w_{11} \cdot (2 \cdot \texttt{prev}(w_{13}) - (w_{11})^2 + \texttt{prev}(w_0))))\Big) \cdot (\texttt{next}(w_8) \cdot \texttt{next}(w_2)) = 0$
- $\texttt{prev}(w_5) = 32 \cdot \texttt{prev}(w_4) + 16 \cdot w_2 + 8 \cdot w_3 + 4 \cdot w_4 + 2 \cdot w_5 + w_6$

- $(\texttt{next}(w_8) \cdot \texttt{next}(w_2) - 1) \cdot \texttt{next}(w_8)$
- $((\texttt{next}(w_5) - a_{-1}) \cdot \texttt{next}(w_3) - 1) \cdot (\texttt{next}(w_5) - a_{-1}) = 0$
- $((\texttt{next}(w_5) - a_1) \cdot \texttt{next}(w_4) - 1) \cdot (\texttt{next}(w_5) - a_1)$
- $(\texttt{next}(W_8) \cdot \texttt{next}(w_2) \cdot w_0) + ((\texttt{next}(w_5) - a_{-1}) \cdot \texttt{next}(w_3) - (\texttt{next}(w_5) - a_1) \cdot \texttt{next}(w_4)) \cdot ((\texttt{next}(w_5) - a_{-1}) \cdot \texttt{next}(w_3) - (\texttt{next}(w_5) - a_1) \cdot \texttt{next}(w_4)) \cdot \texttt{next}(w_6) - \texttt{next}(w_0) = 0$
- $(\texttt{next}(w_8) \cdot \texttt{next}(w_2) \cdot w_1) + ((\texttt{next}(w_5) - a_{-1}) \cdot \texttt{next}(w_3) - (\texttt{next}(w_5) - a_1) \cdot \texttt{next}(w_4)) \cdot \texttt{next}(w_7) - \texttt{next}(w_1) = 0$
- $\texttt{next}(w_8) - ((\texttt{next}(w_5) - a_{-1}) \cdot (\texttt{next}(w_5) - a_0) \cdot (\texttt{next}(w_5) - a_1))$

Additional range checks.

- **Evaluations:**
  - u_i are triplets of bits of the u, first u_0 is the most significant triplet, u'' is an accumulator of u_i (and u' is its previous value)
  - $e_1 = k_0$
  - $e_2 = \sum_{j=254}^{130} k_{254-j} \cdot 2^{j-130}$
- textbfConstraints: for rows $i + 58, i + 60, \ldots, i + 98$:
  - $\texttt{next}(w_{12}) \cdot (\texttt{next}(w_{12}) - 1) \cdot (\texttt{next}(w_{12}) - 2) \cdot (\texttt{next}(w_{12}) - 3) \cdot (\texttt{next}(w_{12}) - 4) \cdot (\texttt{next}(w_{12}) - 5) \cdot (\texttt{next}(w_{12}) - 6) \cdot (\texttt{next}(w_{12}) - 7) = 0$
  - $\texttt{next}(w_{13}) \cdot (\texttt{next}(w_{13}) - 1) \cdot (\texttt{next}(w_{13}) - 2) \cdot (\texttt{next}(w_{13}) - 3) \cdot (\texttt{next}(w_{13}) - 4) \cdot (\texttt{next}(w_{13}) - 5) \cdot (\texttt{next}(w_{13}) - 6) \cdot (\texttt{next}(w_{13}) - 7) = 0$
  - $\texttt{next}(w_{14}) = 2^6 \cdot w_6 + 2^3 \cdot \texttt{next}(w_{12}) + \texttt{next}(w_{13})$

  for row $i + 101$:

  - $w_{12} \cdot (w_{12} - 1) \cdot (w_{12} - 2) \cdot (w_{12} - 3) \cdot (w_{12} - 4) \cdot (w_{12} - 5) \cdot (w_{12} - 6) \cdot (w_{12} - 7) = 0$
  - $w_{13} \cdot (w_{13} - 1) = 0$
  - $w_{14} = 2^4 \cdot \texttt{prev}(w_6) + 2 \cdot w_{12} + w_{13}$
  - $\texttt{next}(w_9) \cdot (w_{14} - \texttt{next}(w_{12})) = 0$
  - $\texttt{next}(w_9) \cdot (\texttt{next}(w_{10}) - 2^{124}) = 0$
  - $\texttt{next}(w_{12}) - \texttt{next}(w_{11}) + \texttt{next}(w_9) \cdot 2^{254} + t_p - 2^{130} = 0$

Copy Constraints:

- (x_T, y_T) in row j are copy constrained with (x_T, y_T) in row $j + 2$ (for $j \in \{i, i + 2, i + 4, \ldots, i + 98\}$) and with last row.
- (x_0, y_0) in row i are copy constrained with values from the first doubling circuit
- (x_0, y_0) in row $j, j \neq i$ are copy constrained with (x_5, y_5) in row $j - 1$
- $n = 0$ in row i is copy-constrained with the zero value
- n in the row $j, j \neq i$ is copy contrained with n' in the row $j - 2$

- n' in the row $i + 102$ is copy constrained with n' in the row $i + 100$
- k is copy constrained with n' in the row $i + 100$
- $u' = 0$ in row $i + 58$ is copy-constrained with the zero value
- u' in the row $j, i < j \leq i + 100$ is copy contrained with u'' in the row $j - 1$
- b_0 in row $i + 1$ is copy-constrained with w_9 in row $i + 102$
- k is copy constrained with w_{11} in the row $i + 102$
- w_5 in row $i + 48$ is copy-constrained with w_{10} in row $i + 102$.

A.3 FADD

Adds two **Blueprint field** elements A and B into the result one C.
`FADD v0 .  v1 .  v2`
Input: v0 - field element A , v1 - field element B.
Output: v2 - field element C.
`v0`: field element A
`v1`: field element B
`v2`: field element C

A.4 FSUB

Subtracts one **Blueprint field** element B from another one A and puts the result into **Blueprint field** element C.
`FSUB v0 .  v1 .  v2`
Input: v0 - field element A , v1 - field element B.
Output: v2 - field element C.
`v0`: field element A
`v1`: field element B
`v2`: field element C

A.5 FMUL

Multiplies two **Blueprint field** elements A and B into the result one C.
`FMUL v0 .  v1 .  v2`
Input: v0 - field element A , v1 - field element B.
Output: v2 - field element C.
`v0`: field element A
`v1`: field element B
`v2`: field element C

A.6 FDIV

Divides one **Blueprint field** element A to another one B and puts the result into **Blueprint field** element C.
`FDIV v0 .  v1 .  v2`
Input: v0 - field element A , v1 - field element B.
Output: v2 - field element C.
`v0`: field element A

`v1`: field element B
`v2`: field element C

A.7 HSHA256

HSHA256 instruction gets two blocks of SHA2-256 hash as input and outputs result of hashing into new block. Blocks are represented in form of variables. Amount of variables required to represent one block depends on used Blueprint field bitsize.

TODO: clarify operated field description.

For case of Pasta curve base field as **Blueprint field** each of input and ouput blocks can be fits into 2 variables.

```
HSHA256 v0 v1 .  v2 v3 .  v4 v5
```

Input: [v0, v1, v2, v3] - two block to hash
Output: [v4, v5] - block with hashing result

TODO: clarify arguments description.

`v0`: Bits from ??? to ??? in ??? endianness of first input block.
`v1`: Bits from ??? to ??? in ??? endianness of first input block.
`v2`: Bits from ??? to ??? in ??? endianness of second input block.
`v3`: Bits from ??? to ??? in ??? endianness of second input block.
`v4`: Bits from ??? to ??? in ??? endianness of result block.
`v5`: Bits from ??? to ??? in ??? endianness of result block.

A.7.1 Circuit

`HSHA256` is handled by the flexible 9-wire circuit defined as follows.

Suppose that input data is in the 32-bits form, which is already padded to the required size. We suppose that the checking that chunked input data corresponds to the original data out of the circuit. However, we do not need to range constrain these chunks as we get them for free from the SHA2-256 circuit.

Thus, the preprocessing constraints for the SHA2-256 circuit is a decomposition of k message blocks to 32 bits chunks without range proofs.

Lookup tables We use the following lookup tables:

1. **SHA2-256 NORMALIZE4** with 2 columns and 2^{14} rows. The first column contains all possible 14-bits words. The second column contains corresponding sparse representations with base 4. The constraints can be used for the range check and sparse representation simultaneously.
2. **SHA2-256 NORMALIZE7** with 2 columns and 2^{14} rows. The first column contains all possible 14-bits words. The second column contains corresponding sparse representations with base 7. The constraints can be used for the range check and sparse representation simultaneously.

3. **SHA2-256 NORMALIZE MAJ** with 2 columns and 2^8 rows. The first column contains all possible 8-bits words. The second column contains corresponding sparse representations with base 4.
4. **SHA2-256 NORMALIZE CH** with 2 columns and 2^8 rows. The first column contains all possible 8-bits words. The second column contains corresponding sparse representations with base 7.

Message scheduling For each block of 512 bits of the padded message the 64 words are constructed in the following way:

- The first 16 words are obtained by splitting the message.
- The last 48 words are obtained by using the functions σ_0, σ_1:

$$W_i = \sigma_1(W_{i-2}) \oplus W_{i-7} \oplus \sigma_0(W_{i-15}) \oplus W_{i-16} \tag{3}$$

Each round of the message scheduling has the following table:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
j + 0	a	a_0	a_1	a_2	a_3	$\hat{a}_1$	$\hat{a}_2$	a'_0	
j + 1	W_i	W_j	a'_1	a'_2	a'_3	s'_0	s'_1	s'_2	s'_3
j + 2	w	s_0	s_1	s_2	s_3	s_0	s_1	s_2	s_3
j + 3		b'_0	b'_1	b'_2	b'_3	s'_0	s'_1	s'_2	s'_3
j + 4	b	b_0	b_1	b_2	b_3	$\hat{b}_0$	$\hat{b}_1$	$\hat{b}_3$	

Evaluations:

Let b be W_{i-2} and a be W_{i-15} from 3. The values W_i and W_j in the table corresponds to W_{i-7} and W_{i-16} respectively from 3. From the round $r = 2$ the copy constraints are used for values b and w from round $r - 2$. The copy constraints for W_{i-7}, W_{i-15} and W_{i-16} are used in a similar way. The output of round W_i from 3 is w.

The first 16 words require a range check. We get it fo free from range-constraining chunks inside functions σ_0 and σ_1. Thus, for i from 16 to 63:

1. Apply σ_0 to W_{i-15}.
2. Add the following constraint for W_i:

$$w_{0,j+2} = w_{0,j+1} + w_{1,j+1} + w_{1,j+2} + w_{2,j+2} \cdot 2^3 + w_{3,j+2} \cdot 2^7 + w_{4,j+2} \cdot 2^{18} + w_{5,j+2} + w_{6,j+2} \cdot 2^{10} + w_{7,j+2} \cdot 2^{17} + w_{8,j+2} \cdot 2^{19},$$

3. Apply σ_1 to W_{i-2}.

Thus, the message schedule takes $5 \cdot 48 = 240$ rows.

The function σ_0 contains sparse mapping with base 4. Let a be divided to chunks a_0, a_1, a_2, a_3 which equals to $3, 4, 11, 14$ bits respectively. The values a'_0, a'_1, a'_2, a'_3 are in sparse form, and a' is a sparse a. **SHA2-256 NORMALIZE4** lookup table is used for mapping to sparse representation and range-constraining for each chunk a_i, where bit-length of $a_i > 3$. If a chunk is 14 bits long, then it is constrained for free. Else the prover has to calculate the sparse representation $\hat{a}_i$ for $2^j \cdot a_i$, where $j + \texttt{len}(a_i) = 14$ and $\texttt{len}(a_i)$ is bit-length of a_i. The tuple $\{s'_0, s'_1, s'_2, s'_3\}$ is a sparse representation of the result of σ_0 and the tuple $\{s_0, s_1, s_2, s_3\}$ is a normal representation. The size of elements of these tuples equals to $\{14, 14, 2, 2\}$ bits respectively.

Constraints:

$$w_{0,j+0} = w_{1,j+0} + w_{2,j+0} \cdot 2^3 + w_{3,j+0} \cdot 2^7 + w_{4,j+0} \cdot 2^{18}$$
$$(w_{1,j+0} - 7) \cdot (w_{1,j+0} - 6) \cdot \ldots \cdot w_{1,j+0} = 0$$
$$w_{5,j+1} + w_{6,j+1} \cdot 4^{14} + w_{7,j+1} \cdot 4^{28} + w_{8,j+1} \cdot 2^{30} = w_{2,j+1} + w_{3,j+1} \cdot 4^4 + w_{4,j+1} \cdot 4^{15} + w_{3,j+1} + w_{4,j+1} \cdot 4^{11} + w_{7,j+0} \cdot 4^{25} + w_{2,j+1} \cdot 4^{28} + w_{4,j+1} + w_{7,j+0} \cdot 4^{14} + w_{2,j+1} \cdot 4^{17} + w_{3,j+1} \cdot 4^{21}$$
$$(w_{7,j+1} - 3) \cdot (w_{7,j+1} - 2) \cdot (w_{7,j+1} - 1) \cdot w_{7,j+1} = 0$$
$$(w_{8,j+1} - 3) \cdot (w_{8,j+1} - 2) \cdot (w_{8,j+1} - 1) \cdot w_{8,j+1} = 0$$

10 plookup constraints:

$$(w_{1,j+0}, w_{7,j+0}), (2^{10} \cdot w_{2,j+0}, w_{5,j+0}), (w_{2,j+0}, w_{2,j+1}), (2^3 \cdot w_{3,j+0}, w_{6,j+0}), (w_{3,j+0}, w_{3,j+1}), (w_{4,j+0}, w_{4,j+1}), (w_{1,j+2}, w_{5,j+1}), (w_{2,j+2}, w_{6,j+1}), (w_{3,j+2}, w_{7,j+2}), (w_{4,j+2}, w_{8,j+2})$$

The function σ_1 contains sparse mapping subcircuit with base 4. Let a be divided to chunks a_0, a_1, a_2, a_3 which equals to $10, 7, 2, 13$ bits respectively. The values a'_0, a'_1, a'_2, a'_3 are in sparse form and a' is a sparse a. **SHA2-256 NORMALIZE4** lookup table is used for mapping to sparse representation and range-constraining in the same way as for σ_0. The tuple $\{s'_0, s'_1, s'_2, s'_3\}$ is a sparse representation of the result of σ_1 and the tuple $\{s_0, s_1, s_2, s_3\}$ is a normal representation. The size of elements of these tuples equals to $\{14, 14, 2, 2\}$ bits respectively.

Constraints:

$$w_{0,j+3} = w_{1,j+3} + w_{2,j+3} \cdot 2^{10} + w_{3,j+3} \cdot 2^{17} + w_{4,j+3} \cdot 2^{19} (w_{3,j+3} - 3) \cdot (w_{3,j+3} - 2) \cdot (w_{3,j+3} - 1) \cdot w_{3,j+3} = 0 w_{5,j+3} + w_{6,j+3} \cdot 4^{14} + w_{7,j+3} \cdot 4^{28} + w_{8,j+3} \cdot 2^{30} = w_{2,j+3} + w_{3,j+3} \cdot 4^7 + w_{4,j+3} \cdot 4^9 + w_{3,j+3} + w_{4,j+3} \cdot 4^2 + w_{1,j+3} \cdot 4^{15} + w_{2,j+3} \cdot 4^{25} + w_{4,j+3} + w_{1,j+3} \cdot 4^{13} + w_{2,j+3} \cdot 4^{23} + w_{3,j+3} \cdot 4^{30} (w_{7,j+3} - 3) \cdot (w_{7,j+3} - 2) \cdot (w_{7,j+3} - 1) \cdot w_{7,j+3} = 0 (w_{8,j+3} - 3) \cdot (w_{8,j+3} - 2) \cdot (w_{8,j+3} - 1) \cdot w_{8,j+3} = 0$$

11 plookup constraints:

$$(2^4 \cdot (w_{1,j+3}, w_{5,j+3}), (2^7 \cdot w_{2,j+3}, w_{6,j+3}), (2 \cdot w_{4,j+3}, w_{7,j+3}), (w_{1,j+3}, w_{1,j+2}), (w_{2,j+3}, w_{2,j+2}), (w_{3,j+3}, w_{3,j+2}), (w_{4,j+3}, w_{4,j+2}), (w_{5,j+2}, w_{5,j+3}), (w_{6,j+2}, w_{6,j+3}), (w$$

Compression There are 64 rounds of compression. Each round of compression has the following table:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
j + 0	e	e'_0	e_0	e_1	e_2	e_3	$\hat{e}_1$	$\hat{e}_2$	$\hat{e}_3$
j + 1	e'	f'	e'_1	e'_2	e'_3	s'_0	s'_1	s'_2	s'_3
j + 2	$ch_{0,sparse}$	$ch_{1,sparse}$	$ch_{2,sparse}$	$ch_{3,sparse}$	$-$	s_0	s_1	s_2	s_3
j + 3	g'	d	h	W_r	e_{new}	ch_0	ch_1	ch_2	ch_3
j + 4	$maj_{0,sparse}$	$maj_{1,sparse}$	$maj_{2,sparse}$	$maj_{3,sparse}$	a_{new}	maj_3	maj_0	maj_1	maj_2
j + 5	a'	b'			c'	s_0	s_1	s_2	s_3
j + 6	s'_1	s'_2	a'_0	a'_1	a'_2	a'_3	s'_3	s'_4	
j + 7	a		a_0	a_1	a_2	a_3	$\hat{a}_0$	$\hat{a}_1$	$\hat{a}_3$

The working variables a, b, c, d, e, f, g, h equals to the fixed initial $SHA - 256$ values for the first chunk and to the sum of previous output and initial values for the rest of chunks. The values for chunk $c, c\neg 1$ are copy-constrained with output from previous round. The variables with quotes are corresponded sparse representation. For each chunk, the following rows are used:

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
j + 0	a	a'	b	b'	d	$-$	$-$	$-$	$-$
j + 1	c	c'	e	e'	h	$-$	$-$	$-$	$-$
j + 2	f	f'	g	g'	$-$	$-$	$-$	$-$	$-$

For the first round, $a, a', b', c', d, e, e', f', g', h$ are copy constrained with corresponded values from the table above.

For the second round, b', c', d, f', g', h are copy constrained with a', b', c, e', f', g from the table. The values a, e are copy constrained with a_{new}, e_{new} from the previous round.

For the third round, c', d, g', h are copy constrained with a', b, e', f. The values a, e are copy constrained with a_{new}, e_{new} from the previous round. The values b', f' are copy constrained with a', e' from the previous round.

In the rest of the rounds the following 'non-special' copy constraints are used:

1. The values a, e are copy constrained with a_{new}, e_{new} from the previous round.
2. The values b', f' are copy constrained with a', e' from the previous round.
3. The values c', g' are copy constrained with b', c' from the previous round.
4. The values d, h are copy constrained with a', e' from the round $r - 3$, where r is current round.

The Σ_0 function contains subcircuit with base 4. Let a be divided to chunks a_0, a_1, a_2, a_3 which equals to $2, 11, 9, 10$ bits respectively. The values a'_0, a'_1, a'_2, a'_3 are in sparse form and a' is a sparse a. The tuple $\{s'_0, s'_1, s'_2, s'_3\}$ is a sparse representation of the result of Σ_0 and the tuple $\{s_0, s_1, s_2, s_3\}$ is a normal representation. The size of elements of these tuples equals to $\{14, 14, 2, 2\}$ bits respectively. **SHA2-256 NORMALIZE4** lookup table is used for mapping to sparse representation and range-constraining in the same way as for σ_0.

Constraints:

$$w_{0,j+7} = w_{2,j+7} + w_{3,j+7} \cdot 2^2 + w_{4,j+7} \cdot 2^{13} + w_{5,j+7} \cdot 2^{22} w_{0,j+5} = w_{2,j+6} + w_{3,j+6} \cdot 4^2 + w_{4,j+6} \cdot 4^{13} + w_{5,j+6} \cdot 4^{22} (w_{2,j+6} - 3) \cdot (w_{2,j+6} - 2) \cdot (w_{2,j+6} - 1) \cdot w_{2,j+6} = 0 w_{0,j+6} + w_{1,j+6} \cdot 4^{14} + w_{6,j+6} \cdot 4^{28} + w_{7,j+6} \cdot 2^{30} = w_{3,j+6} + w_{4,j+6} \cdot 4^{11} + w_{5,j+6} \cdot 4^{20} + w_{1,j+6} \cdot 2^{30} + w_{4,j+6} + w_{5,j+6} \cdot 4^9 + w_{2,j+6} \cdot 4^{19} + w_{3,j+6} \cdot 4^{21} + w_{5,j+6} + w_{2,j+6} \cdot 4^{10} + w_{3,j+6} \cdot 4^{12} + w_{4,j+6} \cdot 4^{23} (w_{6,j+6} - 3) \cdot (w_{6,j+6} - 2) \cdot (w_{6,j+6} - 1) \cdot w_{6,j+6} = 0 (w_{7,j+6} - 3) \cdot (w_{7,j+6} - 2) \cdot (w_{7,j+6} - 1) \cdot w_{7,j+6} = 0$$

11 plookup constraints:

$$(2^3 \cdot (w_{3,j+6}, w_{6,j+6}), (2^5 \cdot w_{4,j+6}, w_{7,j+6}), (2^4 \cdot w_{5,j+6}, w_{8,j+6}), (w_{2,j+6}, w_{2,j+5}), (w_{3,j+6}, w_{3,j+5}), (w_{4,j+6}, w_{4,j+5}), (w_{5,j+6}, w_{5,j+5}), (w_{5,j+5}, w_{0,j+6}), (w_{6,j+5}, w_{1,j+6}), (w$$

The Σ_1 function contains subcircuit with base 7. Let a be divided to chunks a_0, a_1, a_2, a_3 which equals to $6, 5, 14, 7$ bits respectively. The values a'_0, a'_1, a'_2, a'_3 are in sparse form, and a' is a sparse a. The tuple $\{s'_0, s'_1, s'_2, s'_3\}$ is a sparse representation of the result of Σ_1 and the tuple $\{s_0, s_1, s_2, s_3\}$ is a normal representation. The size of elements of these tuples equals to $\{14, 14, 2, 2\}$ bits respectively. **SHA2-256 NORMALIZE7** lookup table is used for mapping to sparse representation and range-constraining in the same way as for σ_0.

Constraints:

$$w_{0,j+0} = w_{2,j+0} + w_{3,j+0} \cdot 2^6 + w_{4,j+0} \cdot 2^{11} + w_{5,j+0} \cdot 2^{25} w_{0,j+1} = w_{1,j+0} + w_{2,j+1} \cdot 7^6 + w_{3,j+1} \cdot 7^{11} + w_{4,j+1} \cdot 7^{25} w_{5,j+1} + w_{6,j+1} \cdot 4^{14} + w_{7,j+1} \cdot 4^{28} + w_{8,j+1} \cdot 2^{30} = w_{2,j+1} + w_{3,j+1} \cdot 4^5 + w_{4,j+1} \cdot 4^{19} + w_{1,j+0} \cdot 2^{26} + w_{3,j+1} + w_{4,j+1} \cdot 4^{14} + w_{1,j+0} \cdot 4^{21} + w_{2,j+1} \cdot 4^{27} + w_{4,j+1} + w_{1,j+0} \cdot 4^7 + w_{2,j+1} \cdot 4^{13} + w_{3,j+1} \cdot 4^{27} (w_{3,j+1} - 3) \cdot (w_{3,j+1} - 2) \cdot (w_{3,j+1} - 1) \cdot w_{3,j+1} = 0 (w_{4,j+1} - 3) \cdot (w_{4,j+1} - 2) \cdot (w_{4,j+1} - 1) \cdot w_{4,j+1} = 0$$

11 plookup constraints:

$$(2^8 \cdot (w_{2,j+0}, w_{1,j+0}), (2^9 \cdot w_{3,j+0}, w_{2,j+1}), (2^7 \cdot w_{5,j+0}, w_{4,j+1}), (w_{2,j+0}, w_{1,j+0}), (w_{3,j+0}, w_{2,j+1}), (w_{4,j+0}, w_{3,j+1}), (w_{5,j+0}, w_{4,j+1}), (w_{5,j+2}, w_{5,j+1}), (w_{6,j+2}, w_{6,j+1}), (w$$

The Maj function contains subcircuit with base 4 for a, b, c. **SHA2-256 NORMALIZE MAJ** lookup table is used for mapping to sparse representation in the same way as for σ_0.

The value of the maj function is stored in chunks of 8 bits $\{maj_0, maj_1, maj_2, maj_3\}$ and the corresponded sparse value is $\{maj_{0,sparse}, maj_{1,sparse}, maj_{2,sparse}, maj_{3,sparse}\}$

Constraints:

$$w_{0,j+4} + w_{1,j+4} \cdot 4^8 + w_{2,j+4} \cdot 4^{8 \cdot 2} + w_{3,j+4} \cdot 4^{8 \cdot 3} = w_{0,j+5} + w_{1,j+5} + w_{4,j+5}$$

4 plookup constraints:

$$(w_{5,j+4}, w_{0,j+4}), (w_{6,j+4}, w_{1,j+4}), (w_{7,j+4}, w_{2,j+4}), (w_{8,j+4}, w_{3,j+4})$$

The Ch function contains sparse mapping subcircuit with base 7 for e, f, g. **SHA2-256 NORMALIZE CH** lookup table is used for mapping to sparse representation in the same way as for σ_0. The value of the ch function is stored in chunks of 8 bits $\{ch_0, ch_1, ch_2, ch_3\}$ and the corresponded sparse value is $\{ch_{0,sparse}, ch_{1,sparse}, ch_{2,sparse}, ch_{3,sparse}\}$

Constraints:

$$w_{0,j+2} + w_{1,j+2} \cdot 7^8 + w_{2,j+2} \cdot 7^{8 \cdot 2} + w_{3,j+2} \cdot 7^{8 \cdot 3} = w_{0,j+1} + 2 \cdot w_{1,j+1} + 3 \cdot w_{0,j+3}$$

4 plookup constraints:

$$(w_{5,j+3}, w_{0,j+2}), (w_{6,j+3}, w_{1,j+2}), (w_{7,j+3}, w_{2,j+2}), (w_{8,j+3}, w_{3,j+2})$$

Update the values a and e The value W_r is a word, where r is a number of round. It has to be copy-constrained with the word W_r in the message scheduling.

Constraints:

$$w_{4,j+3} = w_{1,j+3} + w_{2,j+3} + w_{5,j+2} + w_{6,j+2} \cdot 2^{14} + w_{7,j+2} \cdot 2^{28} + w_{8,j+2} \cdot 2^{30} + w_{5,j+3} + w_{6,j+3} \cdot 2^8 + w_{7,j+3} \cdot 2^{8 \cdot 2} + w_{8,j+3} \cdot 2^{8 \cdot 3} + k[r] + w_{3,j+3}, \text{ where } r \text{ is a number of round.}$$

$$w_{4,j+4} = w_{4,j+3} - w_{1,j+3} + w_{5,j+5} + w_{6,j+5} \cdot 2^{14} + w_{7,j+5} \cdot 2^{28} + w_{8,j+5} \cdot 2^{30} + w_{5,j+4} + w_{6,j+4} \cdot 2^8 + w_{7,j+4} \cdot 2^{8 \cdot 2} + w_{8,j+4} \cdot 2^{8 \cdot 3}$$

Output of the round

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
j + 0	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	–	–	–
j + 1	h_0	h_1	h_2	h_3	h_4	h_5	–	–	–
j + 2	a	b	c	d	e	f	–	–	–
j + 3	h_6	h_7	$\overline{g}$	$\overline{h}$	g	h	–	–	–

Evaluations:

The values $\overline{\xi}$ copy constrained with initial working variables of this round. The values a, b, c, d, e, f, g, h copy constrained with variables from the compression. The output of the round is $h_0, h_1, .., h_7$

Constraints:

$$\begin{aligned}
w_{0,j+1} &= w_{0,j+0} + w_{0,j+2} \\
w_{1,j+1} &= w_{1,j+0} + w_{1,j+2} \\
w_{2,j+1} &= w_{2,j+0} + w_{2,j+2} \\
w_{3,j+1} &= w_{3,j+0} + w_{3,j+2} \\
w_{4,j+1} &= w_{4,j+0} + w_{4,j+2} \\
w_{5,j+1} &= w_{5,j+0} + w_{5,j+2} \\
w_{0,j+3} &= w_{2,j+3} + w_{4,j+3} \\
w_{1,j+3} &= w_{3,j+3} + w_{5,j+3}
\end{aligned}$$

Cost The total value of rows is $48 \cdot 5 + 8 \cdot 64 + 3 = 755$ per chunk.