Sin7Y, Applied R&D Team*

November 2022

Contents

*https://twitter.com/Sin7Y_Labs

1 Circuit config

1.1 Circuit config

1.1.1 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,
        },
    }
}
```

1.1.2 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,
        },
    }
}
```

1.1.3 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,
        },
    }
}
```

1.1.4 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,
        },
    }
}
```

2 Gates

Each gate is a row with 135 columns. As different custom gate has different complexity, for some complex gate 135 columns may only constraints one operation while for other simple gate 135 columns may constraints several

operations. The index of operation in a row is called slot.

Steps to use a custom gate:

- Determine the type of gate;
- Find row and slot using gate type;
- Wire the operation parameters to cells found;

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

2.1 Custom gates

2.1.1 arithmetic_base

ArithmeticGate is a gate which 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 structure of gate is shown in **Figure 1**.

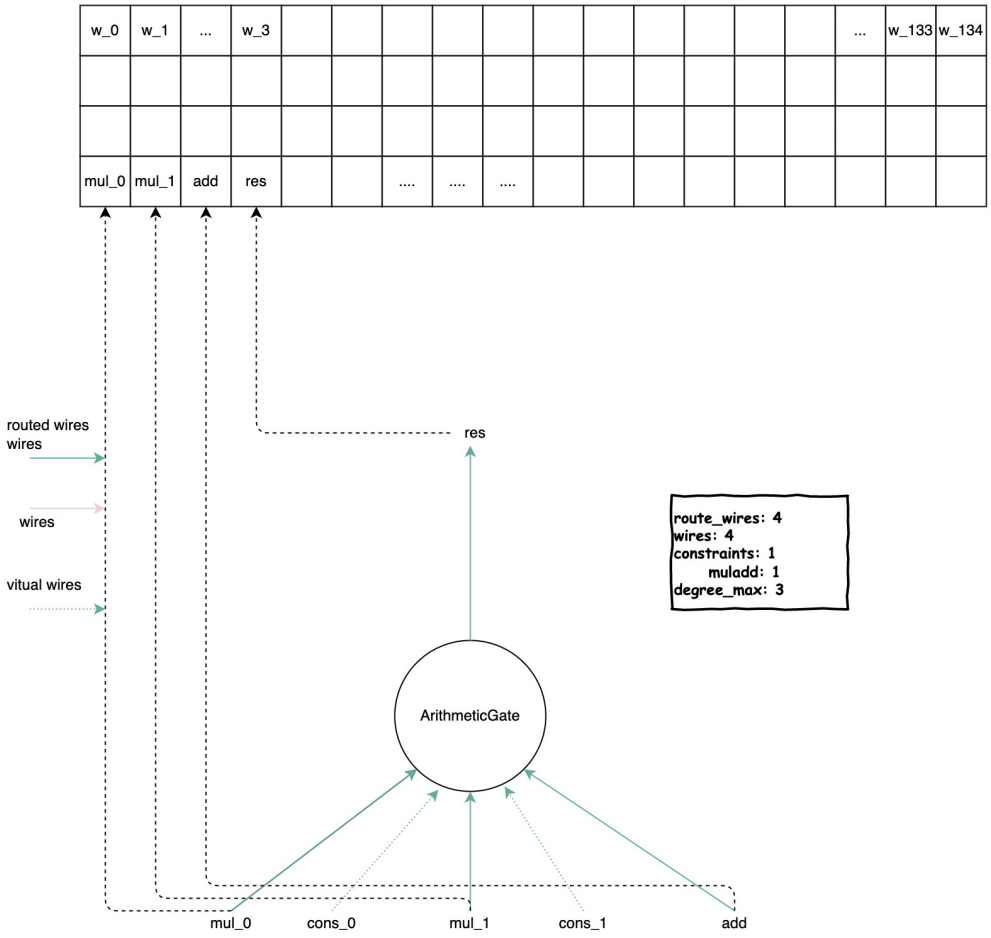


Figure 1: ArithmeticGate

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

2.1.2 arithmetic_extension

To understand the design principle of this Gate, we must first understand Field extension.

Taking Plonky2's GoldilocksField as an example, we give the extension field elements under quadratic, quartic and quintic extensions respectively in **Figure 2**.

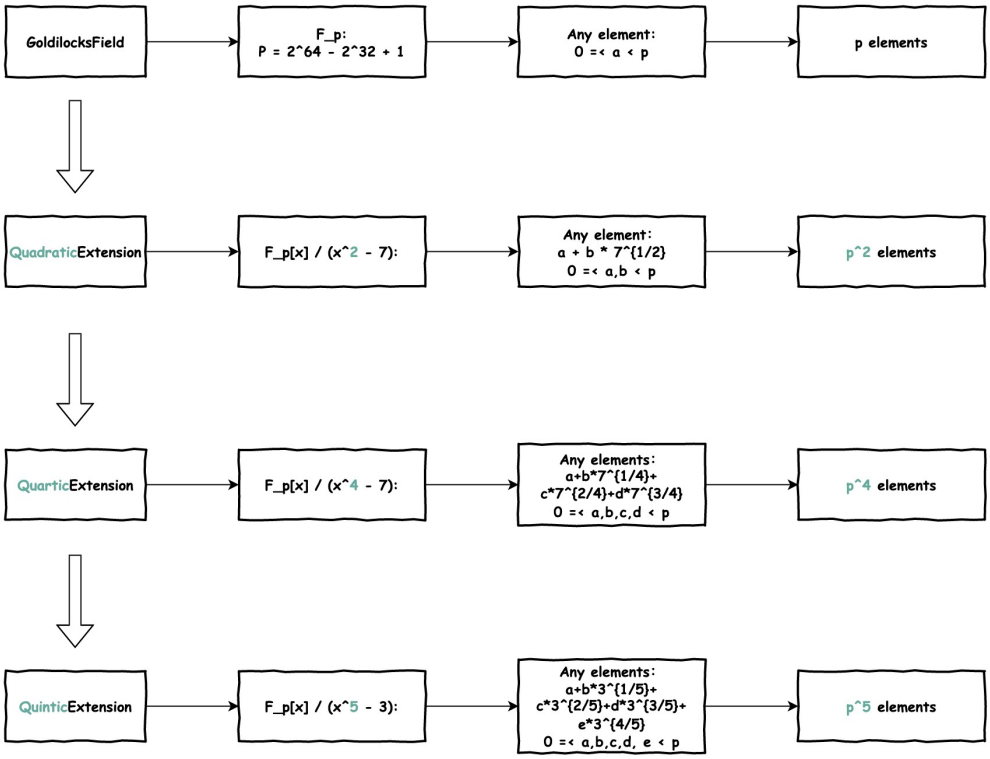


Figure 2: GoldilocksField Extension

It is easy to see that for QuadraticExtension 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 domain, there are p^2 elements and the original domain is a subset of the quadratic extension domain.

ArithmeticExtensionGate is also a gate which 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 gate is shown in **Figure 3**.

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

2.1.3 base_sum

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

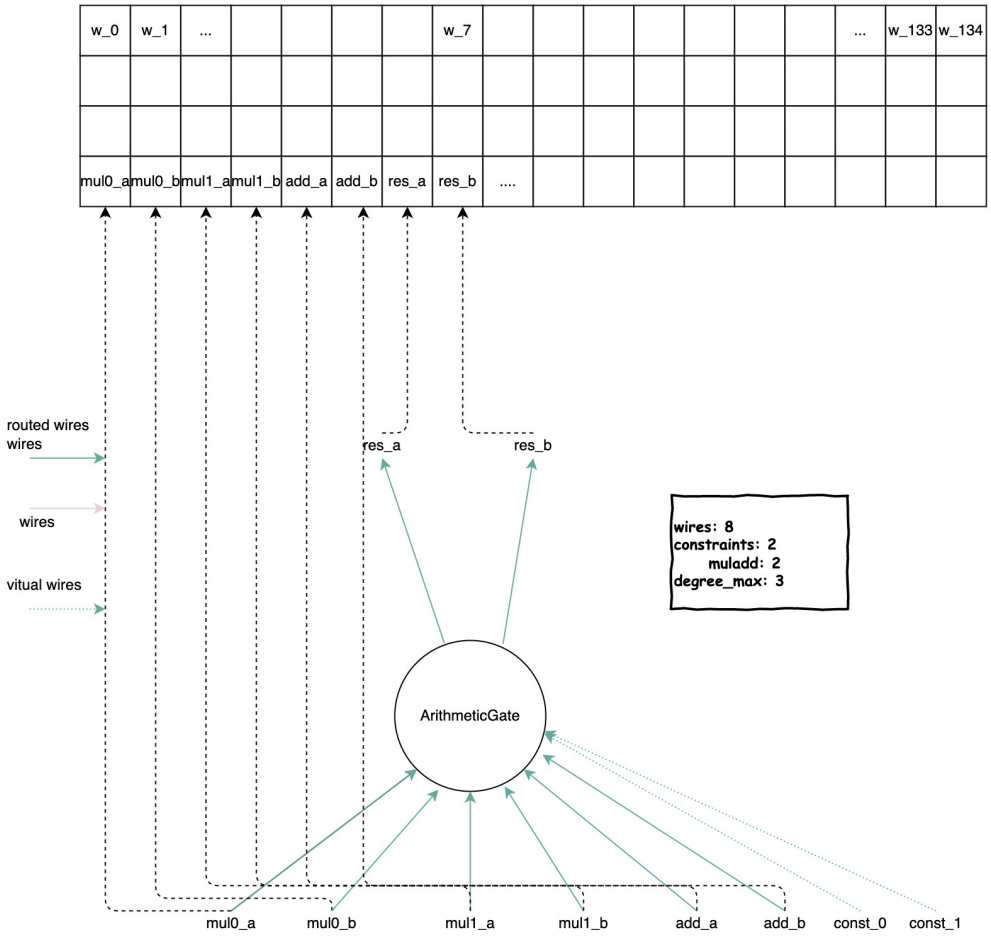


Figure 3: ArithmeticExtensionGate

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 gate is shown in **Figure 4**.

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

2.1.4 exponentiation

ExponentiationGate is a gate for raising a value to a power. Trace table contains 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

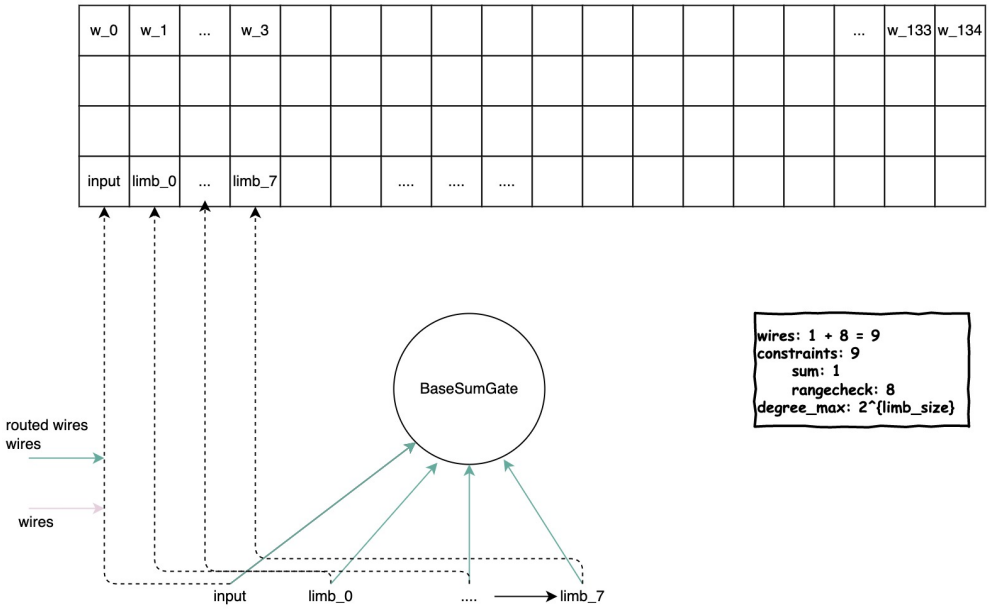


Figure 4: BaseSumGate

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 gate is shown in **Figure 5**.

Each step result constraint with intermediate values, and output is constrained with the final intermediated value, a total of $bits + 1$ constraints. 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);
```

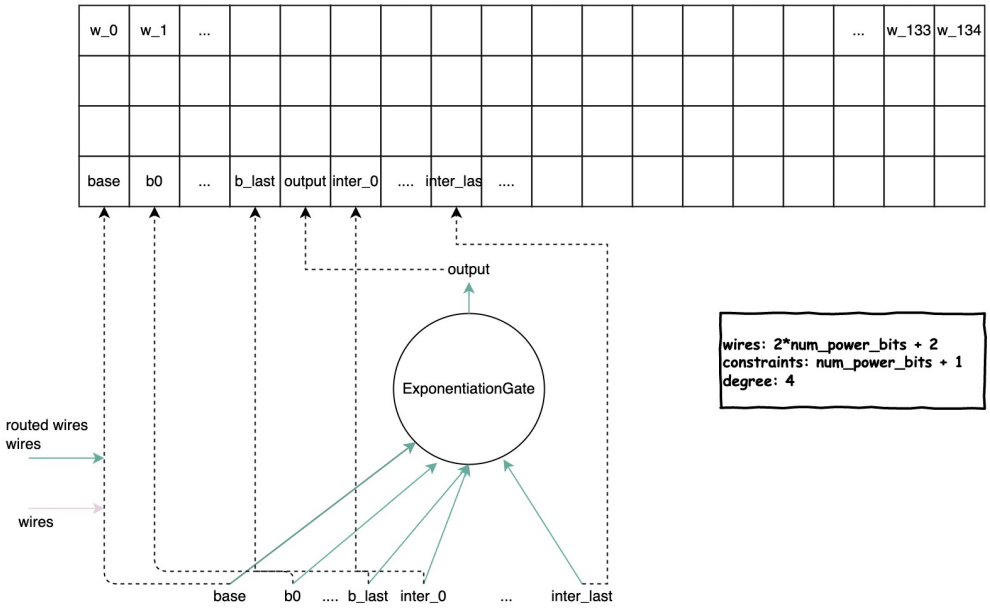


Figure 5: ExponentiationGate

2.1.5 poseidon

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

Each round function of poseidon permutation consists of the following three compoments.

1. ARC(.): AddRoundConstants
2. S: SubWords
3. M(.): MixLayer

Trace of the gate is like:

- 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.
- 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);
}
```

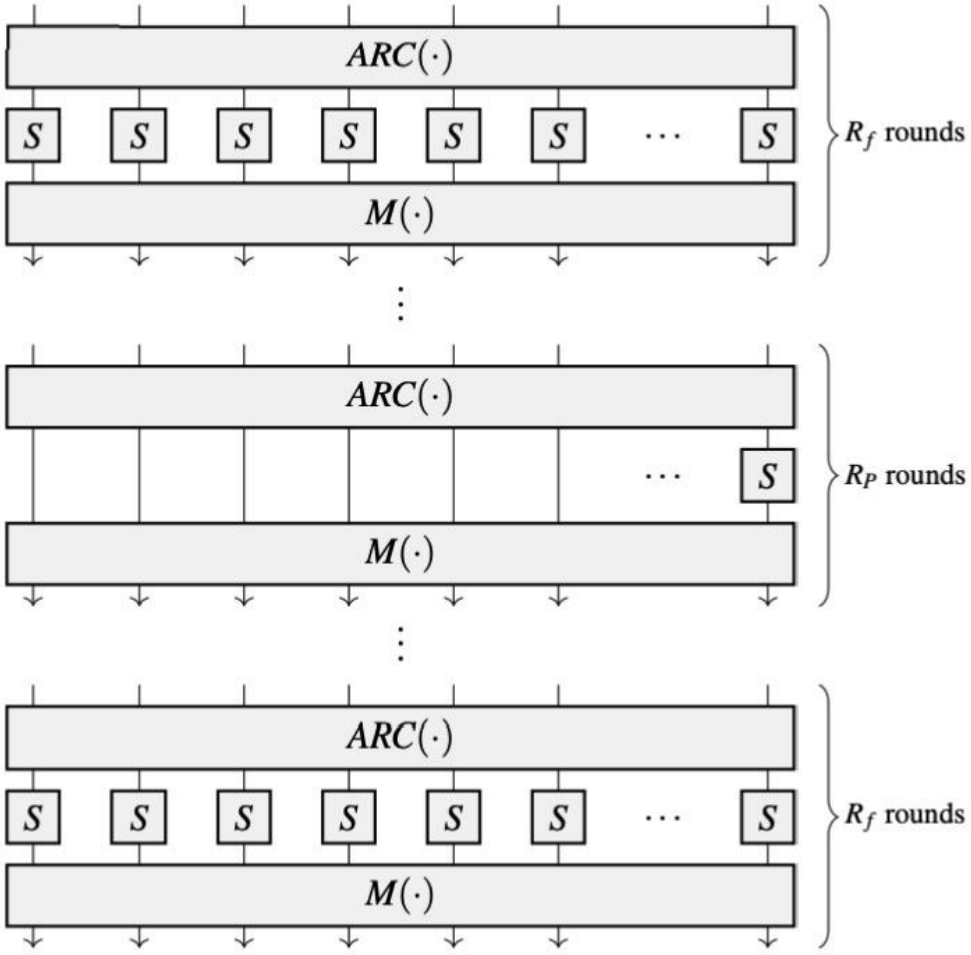


Figure 6: Construction of poseidon

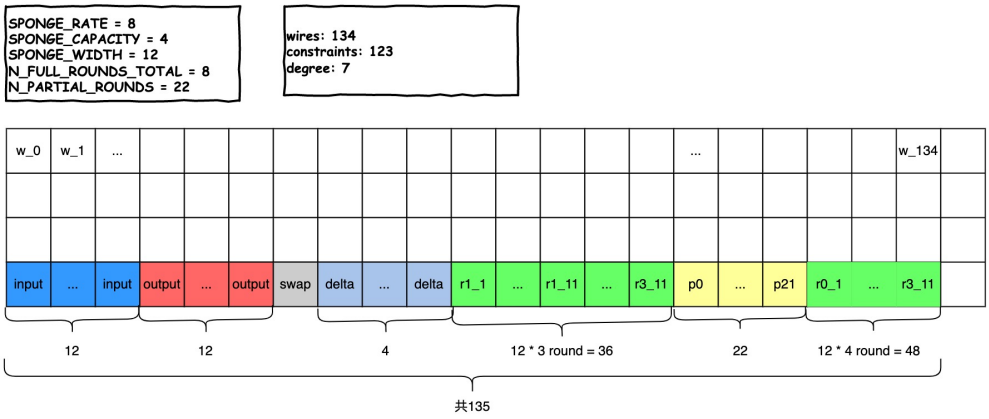


Figure 7: PoseidonGate

- Initialize state: when swap=0, state=input; when swap=1, state is swapped input:

```
for i in 0..4 {
    ...
```

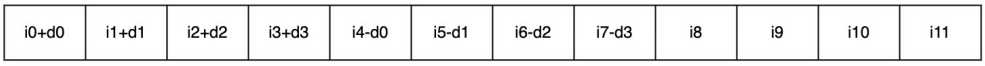


Figure 8: Poseidon State Init

```
    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, totally 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 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 parial 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 22th 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 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)]);
}
```

Constraints of this gate in total is 123, degree is 7 (when performing s-box, making state$[i] \mapsto$ state$[i]^7$).

2.1.6 poseidon_mds

This gate is used for constrain outputs of poseidon mds.

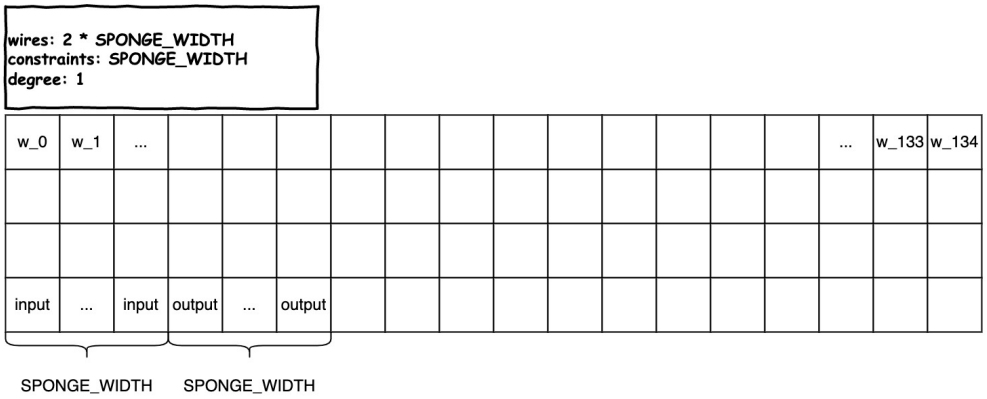


Figure 9: PoseidonMdsGate

computed_output is calculated from input, and constrained with output element by element, total 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()
```

2.1.7 random_access

RandomAccessGate is used for verify that an element matches a value in the list.

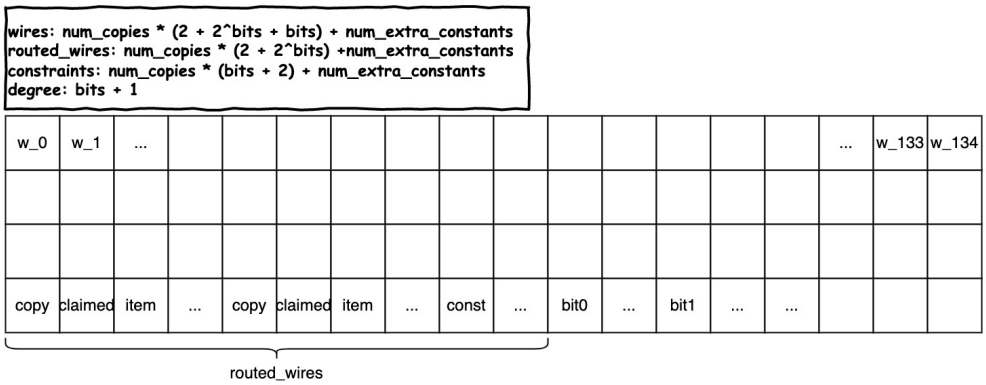


Figure 10: RandomAccessGate

- item: list items.
- copy: index of 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 bits, reconstruct items with 2-elements-tuple, select first element when bit is 0 and select second when bit is 1. After the bits round, only one element remains, that is, the index element corresponding to bits, constraint it with claimed.

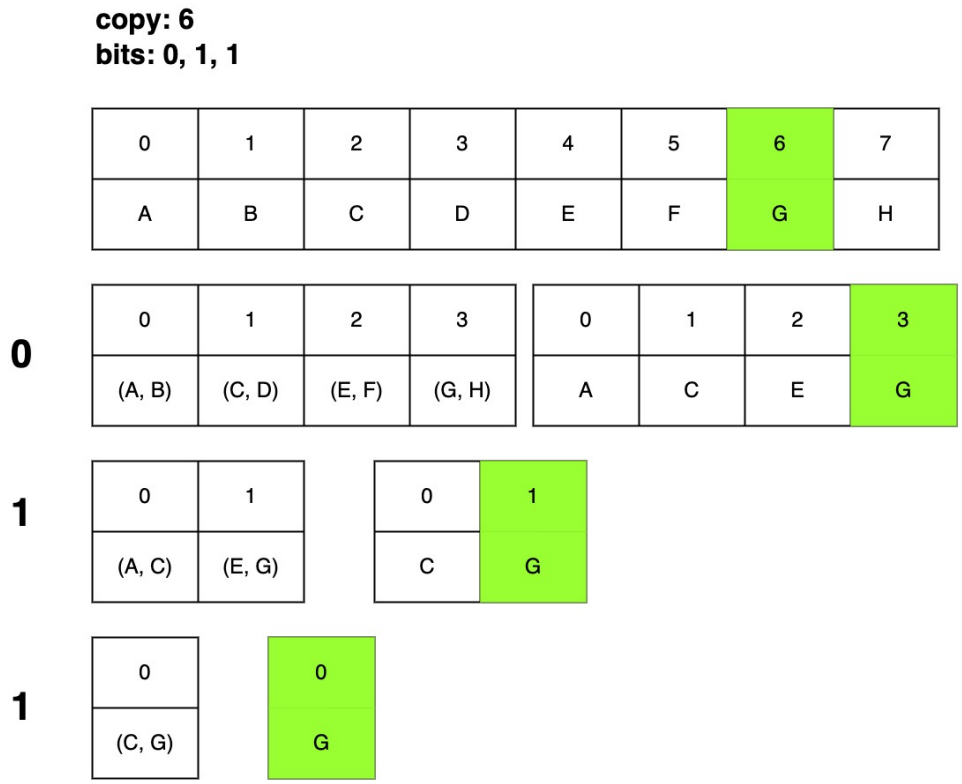


Figure 11: Random Access Example

```
for b in bits {
    list_items = list_items
        .iter()
        .tuples()
        .map(|(&x, &y)| builder.select_ext_generalized(b, y, x))
        .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. Degree is bits $+ 1$ which happens when repeatedly folding the list.

2.1.8 reducing

ReducingGate is used for computes output $=$ old_acc $+ \sum C_i * \alpha^i$ in base field.

Trace structure for this gate is like:

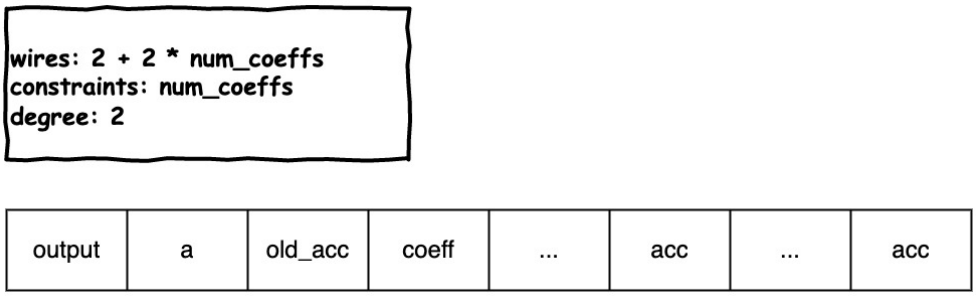


Figure 12: ReducingGate

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. 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 extension field, constrain logic is all the same.

2.1.9 high_degree_interpolation

InterpolationGate is used for interpolation 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 less gates (than LowDegreeInterpolationGate).

HighDegreeInterpolationGate trace is shown in **Figure 13**.

Constraints:

- 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 evaluation point, and constrain with ev. – 1 constraint.

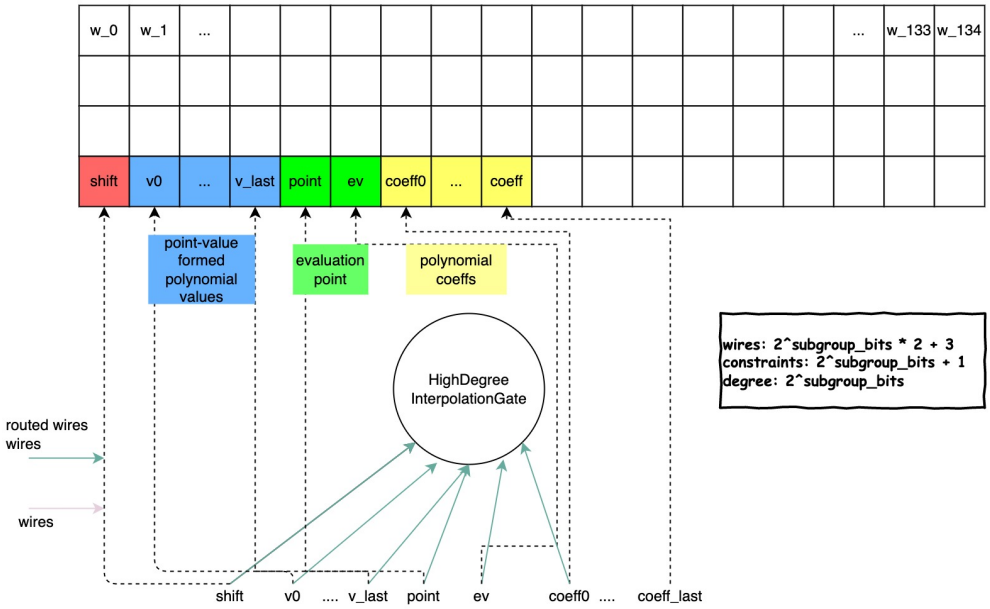


Figure 13: HighDegreeInterpolationGate

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.

Number of constraints equals num_points $+$ 1: num_points for consistency between the coefficients and the point-value pairs, 1 constraints for the evaluation value.

2.1.10 low_degree_interpolation

InterpolationGate is used for interpolation 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 14**.

Constraints:

- 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 evalation point. – A total of $2^{\text{subgroup\_bits}} - 2$ constraints.
- Evaluate the coefficient-form polynomial at evaluation point, and constrain with ev. – 1 constraint.

As can be seen from the above constraint description, number of constraints is $3 * 2^{\text{subgroup\_bits}} - 3$, degree of LowDegreeInterpolationGate is 2.

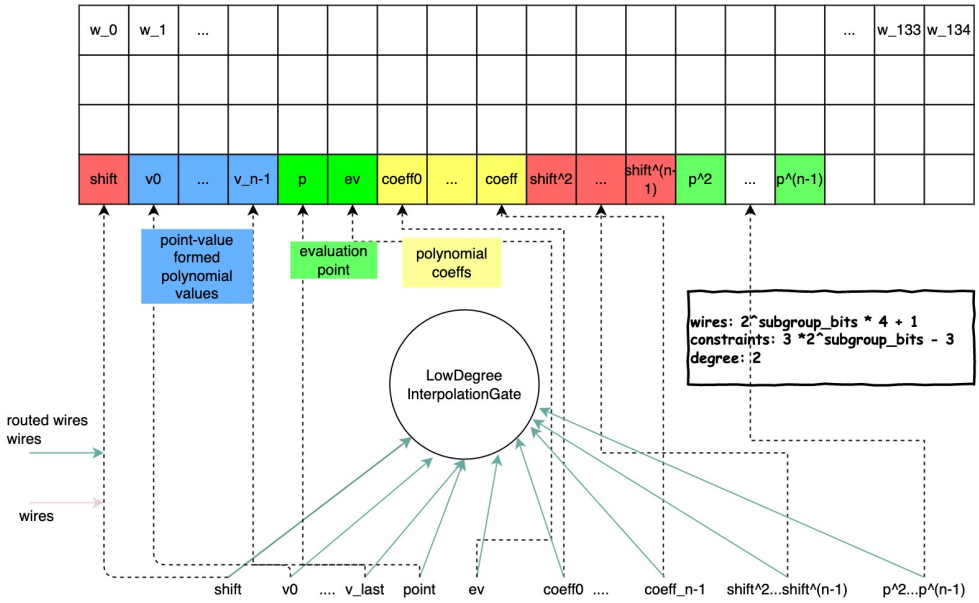


Figure 14: LowDegreeInterpolationGate

2.2 U32 gates

2.2.1 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.

Gate structure is like **Figure 15**.

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);
```

- 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. Degree of the gate is 4 which is needed by 4-bits limbs range check.

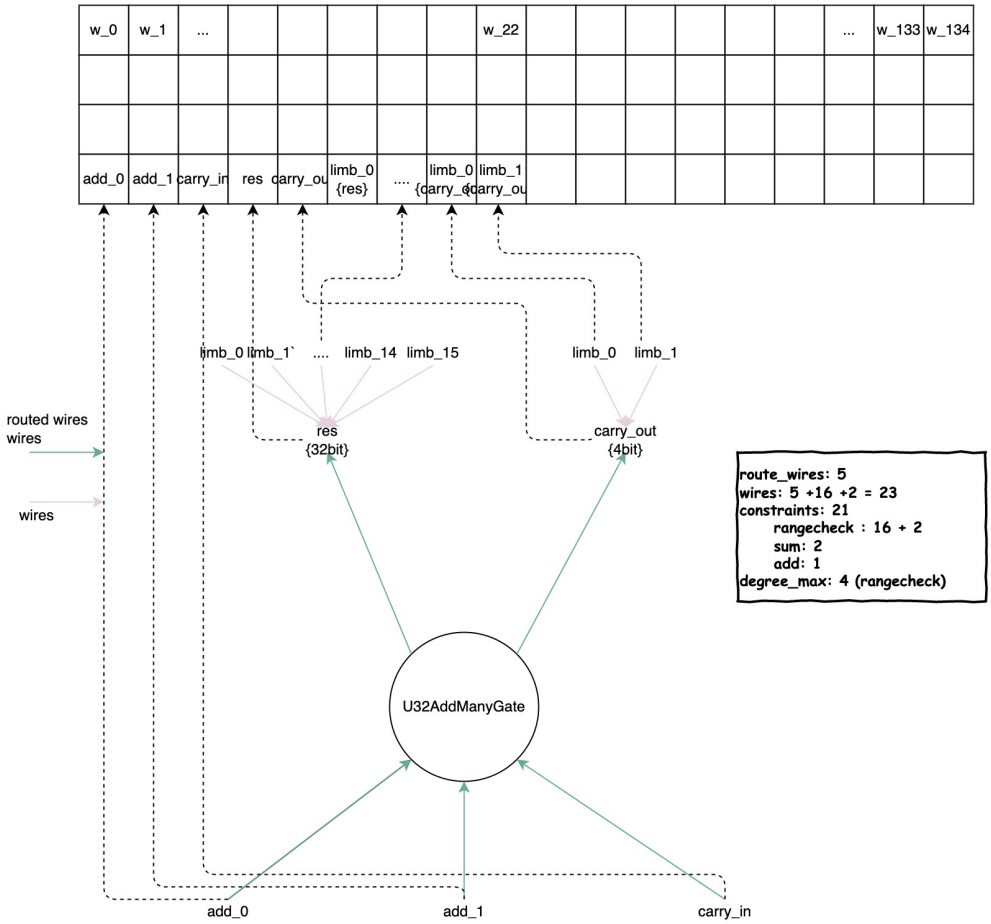


Figure 15: U32AddManyGate

2.2.2 arithmetic_u32

U32ArithmeticGate gate is used for compute res $=$ mul_0 $\times$ mul_1 $+$ add. Res is store in res_lo and res_hi each of which can be represented by 15 limbs.

Gate structure is like **Figure 16**.

Constraints for each operation:

- Constrain res is not overflow (less equal than `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. Degree of the gate is 4 which is needed by 4-bits limbs range check.

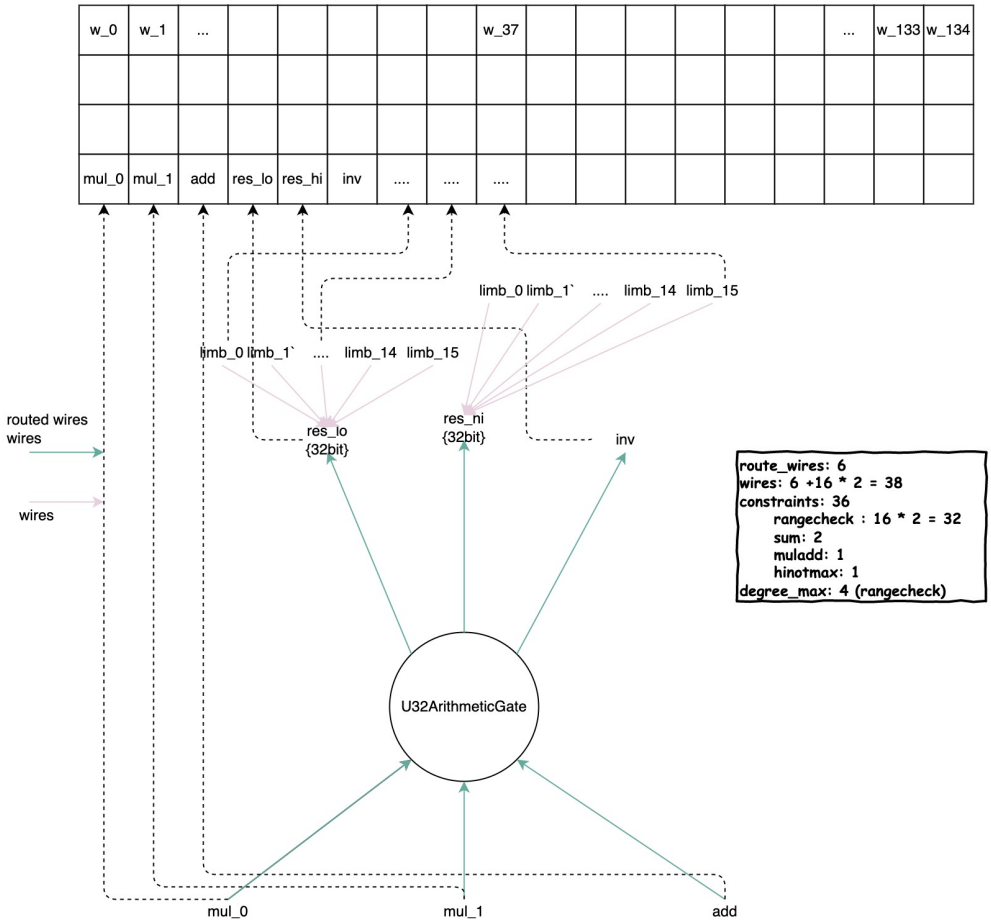


Figure 16: U32ArithmeticGate

2.2.3 comparison

ComparisonGate is a gate for checking that one value is less than or equal to another. Compared numbers are divided into chunks, chunk size and number is configable.

For a ComparisonGate with 8 4-bits chunks is like **Figure 17**.

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.

2.2.4 range_check_u32

U32RangeCheckGate is a gate which can decompose a number into base B little-endian limbs.

Gate structure is like **Figure 18**.

Constraints for each input_limb:

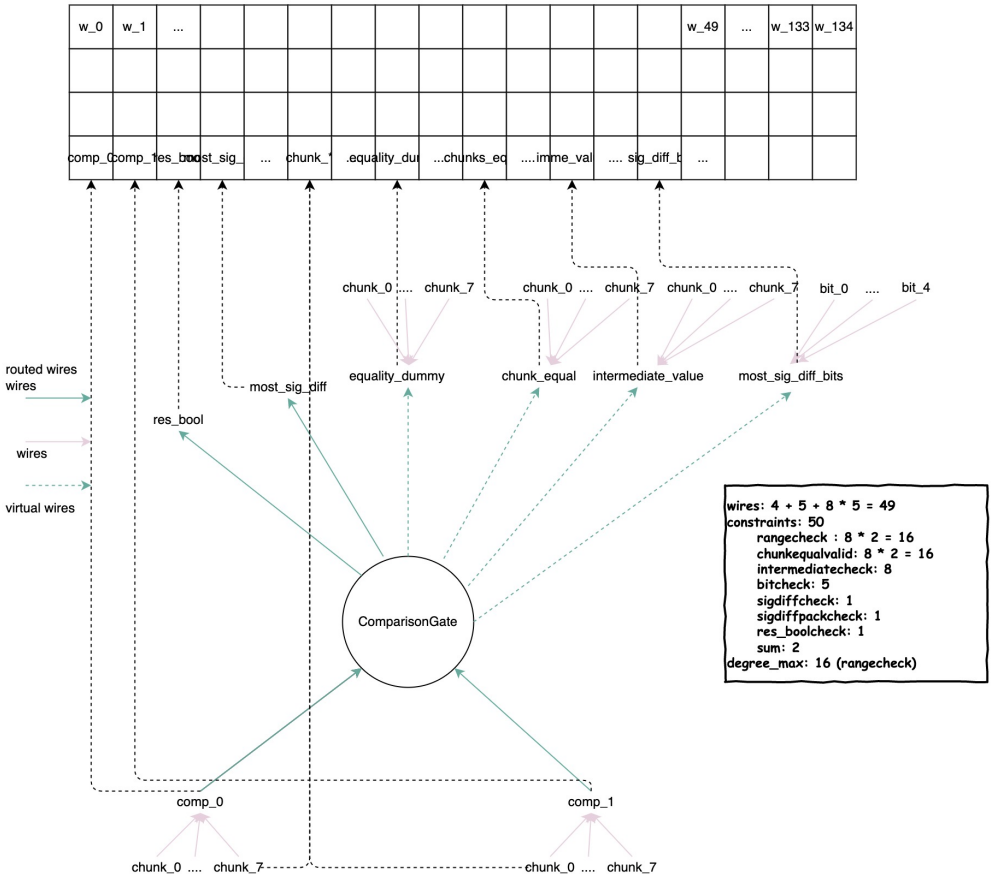


Figure 17: ComparisonGate

- 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 ($(x-0)(x-1)\cdots(x-\text{BASE}+1)$).

In summary, there are 17 constraints per input limbs, a total of num_input_limbs $\times$ 17 constraints. Degree of the gate equals BASE for range check.

2.2.5 subtraction_u32

U32SubtractionGate is a gate to perform a subtraction on 32-bit limbs: given 'x', 'y', and 'borrow', it returns the result $x - y -$ borrow and, if this underflows, a new 'borrow'.

Gate structure is like **Figure 19**.

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));
```

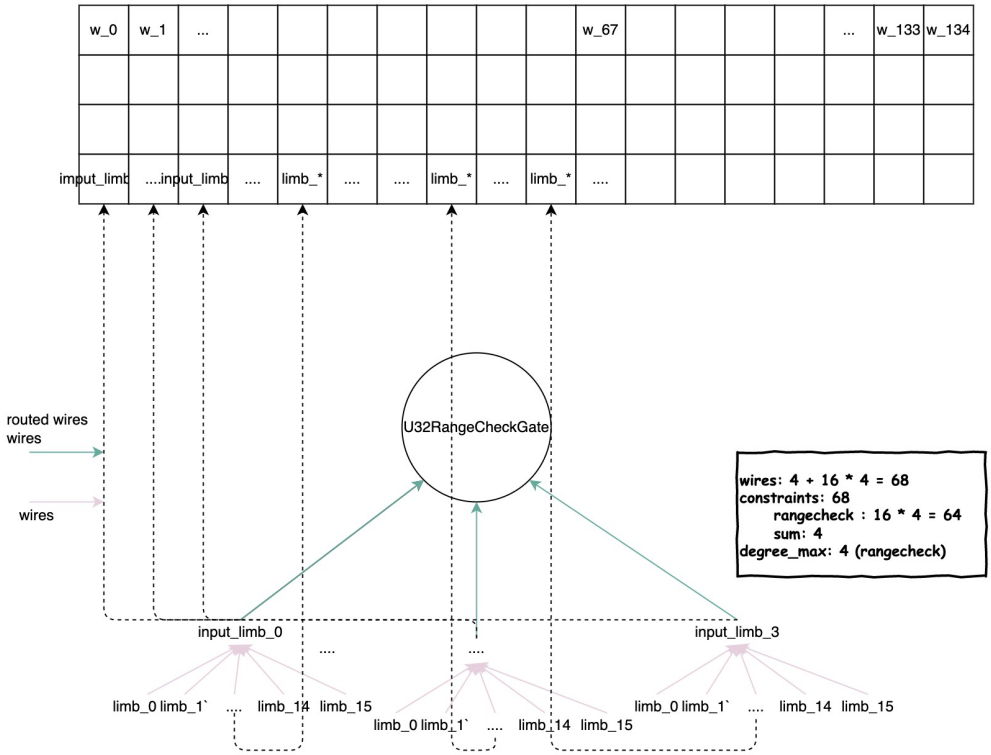


Figure 18: U32RangeCheckGate

- 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. Degree of the gate is 4 which is needed by 4-bits limbs range check.

3 Gadgets

3.1 biguint

3.1.1 biguint-add

Target Implement the addition of two biguints.

Constraints logic

- Equation for gates;
- Sumcheck between output and limbs;
- Rangecheck for limbs.

Circuit layout See **Figure 20**.

Trace layout See **Figure 21**.

Constraints info and costs

- gate type num: 1 (U32AddManyGate)

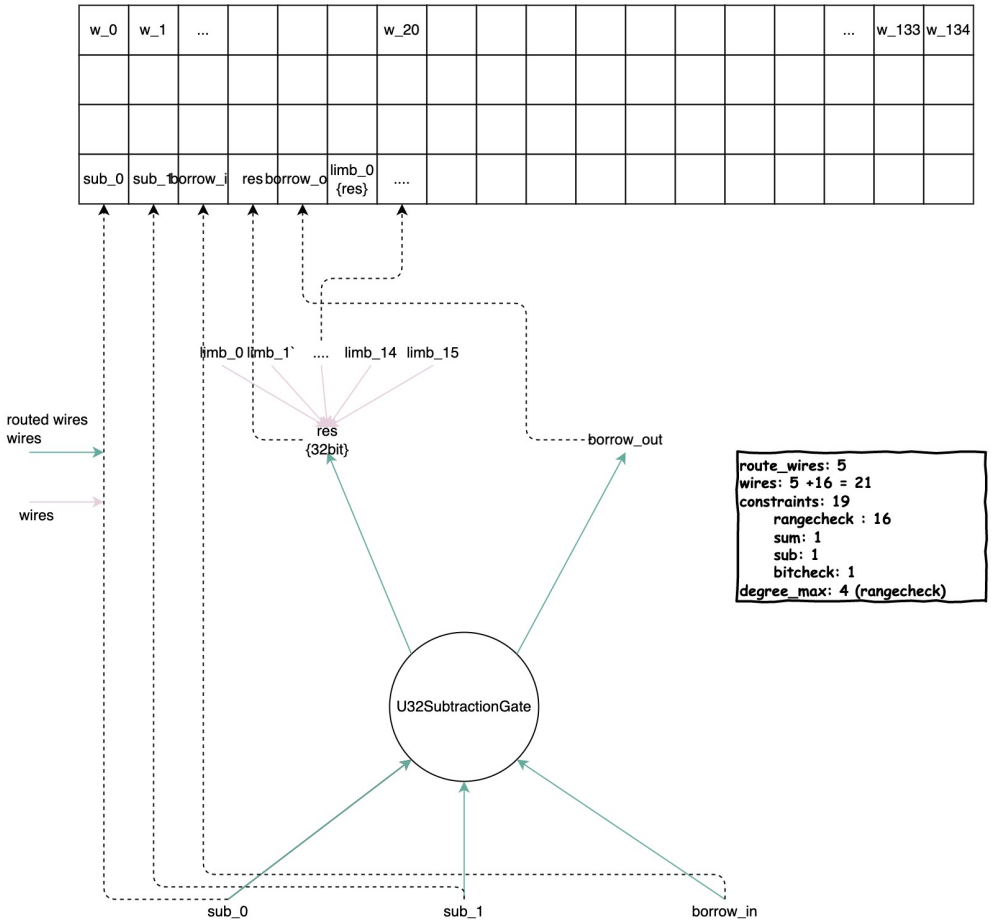


Figure 19: U32SubtractionGate

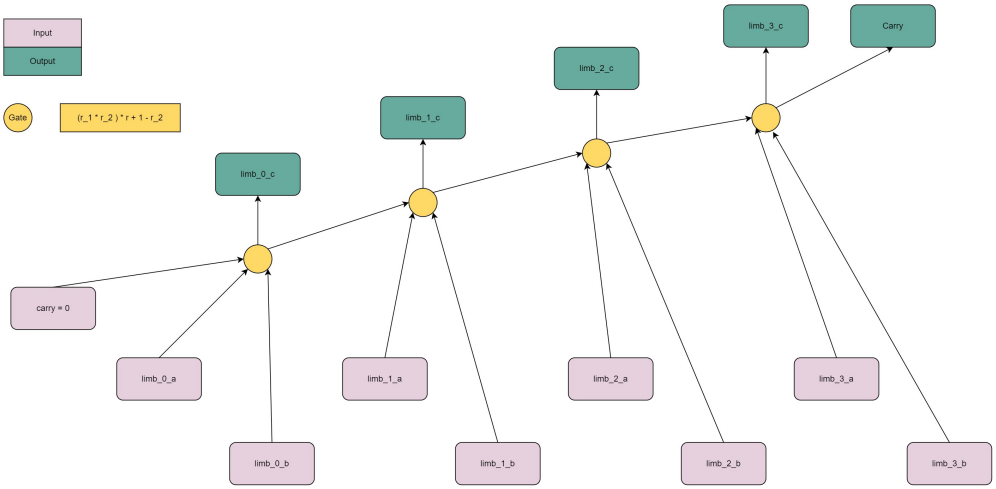


Figure 20: biguint-add circuit layout

- gate ops num: limbs-num
- gate instance num: ceil(limbs-num / gate.ops)

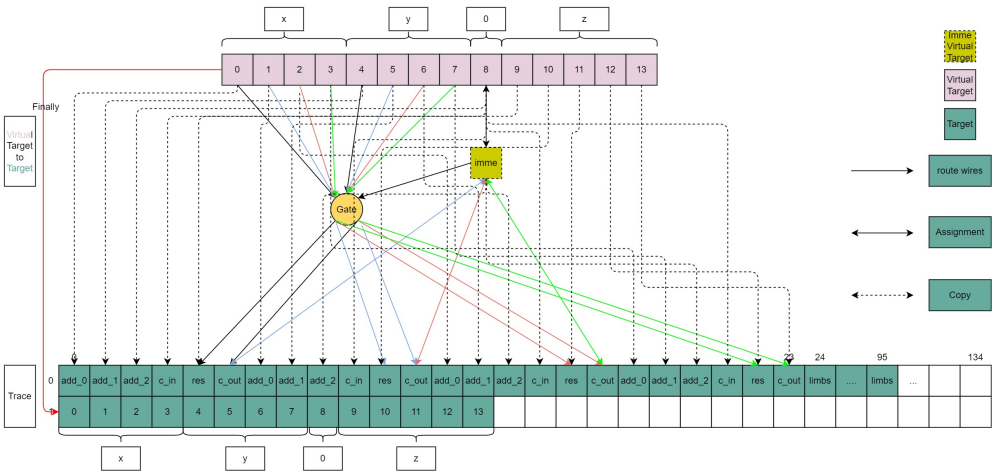


Figure 21: biguint-add trace layout

- copy-constraints: limbs-num * 4
- max-degree: 4 (`1 << limb-bits`)

Questions

- Why not make rangecheck constraint for inputs?
- Why not make copy constraint between cur-c-in and last-c-out?

3.1.2 biguint-sub

Target Implement the substraction of two biguints.

Constraints logic

- Equation for gate;
- Sumcheck for ouptput;
- Rangecheck for limbs.

Circuit layout See **Figure 22**.

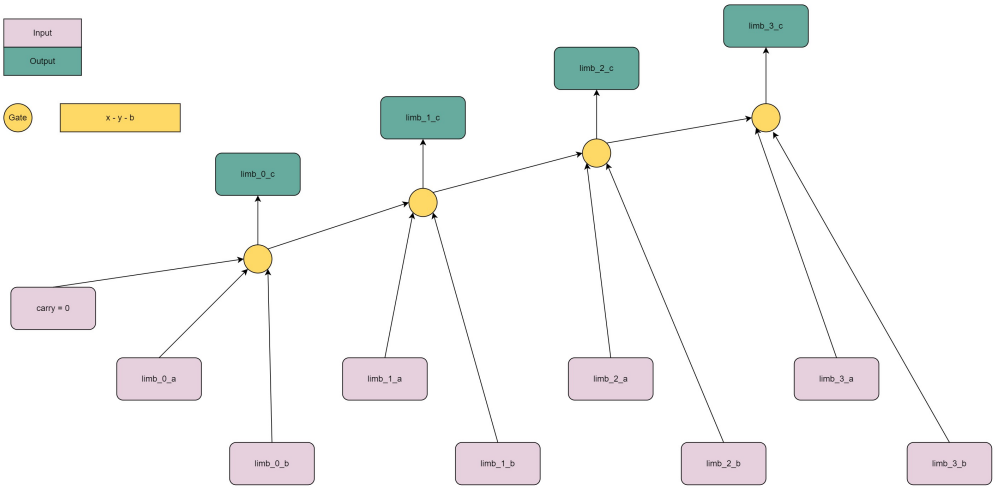


Figure 22: biguint-sub circuit layout

Trace layout See **Figure 23**.

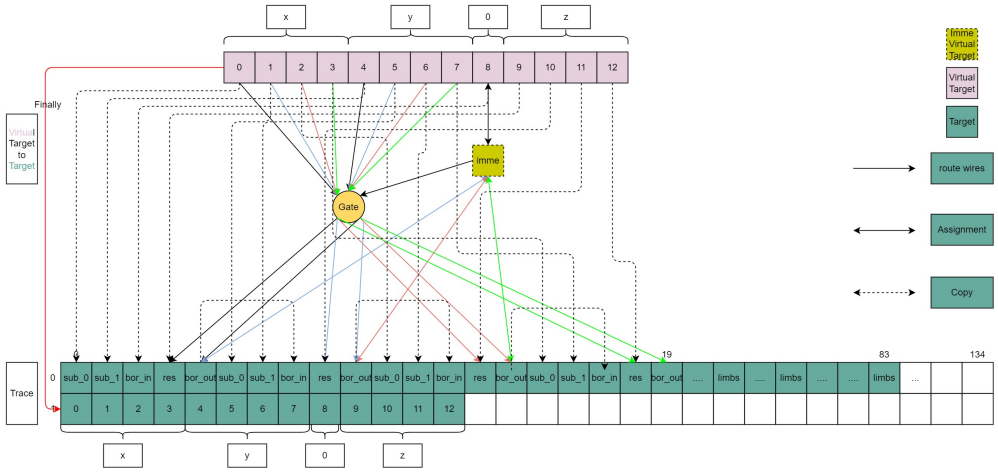


Figure 23: biguint-sub trace layout

Constraints info and costs

- constraints-num: $6 \times (3 + 32/2) = 114$
- copy-constraints: 16
- max-degree: 4
- wires-num: $6 \times (5 + 16) = 126$

Questions

- Why not make rangecheck constraint for inputs?
- Could try to use the same constraint with add-gate.

3.1.3 biguint-mul

Target Implement the multiplication of two biguints.

Constraints logic

- Compute mul-factors first, use U32ArithmeticGate;
- Add mul-factors from low bits, use U32AddManyGate.

Process layout See **Figure 24**

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

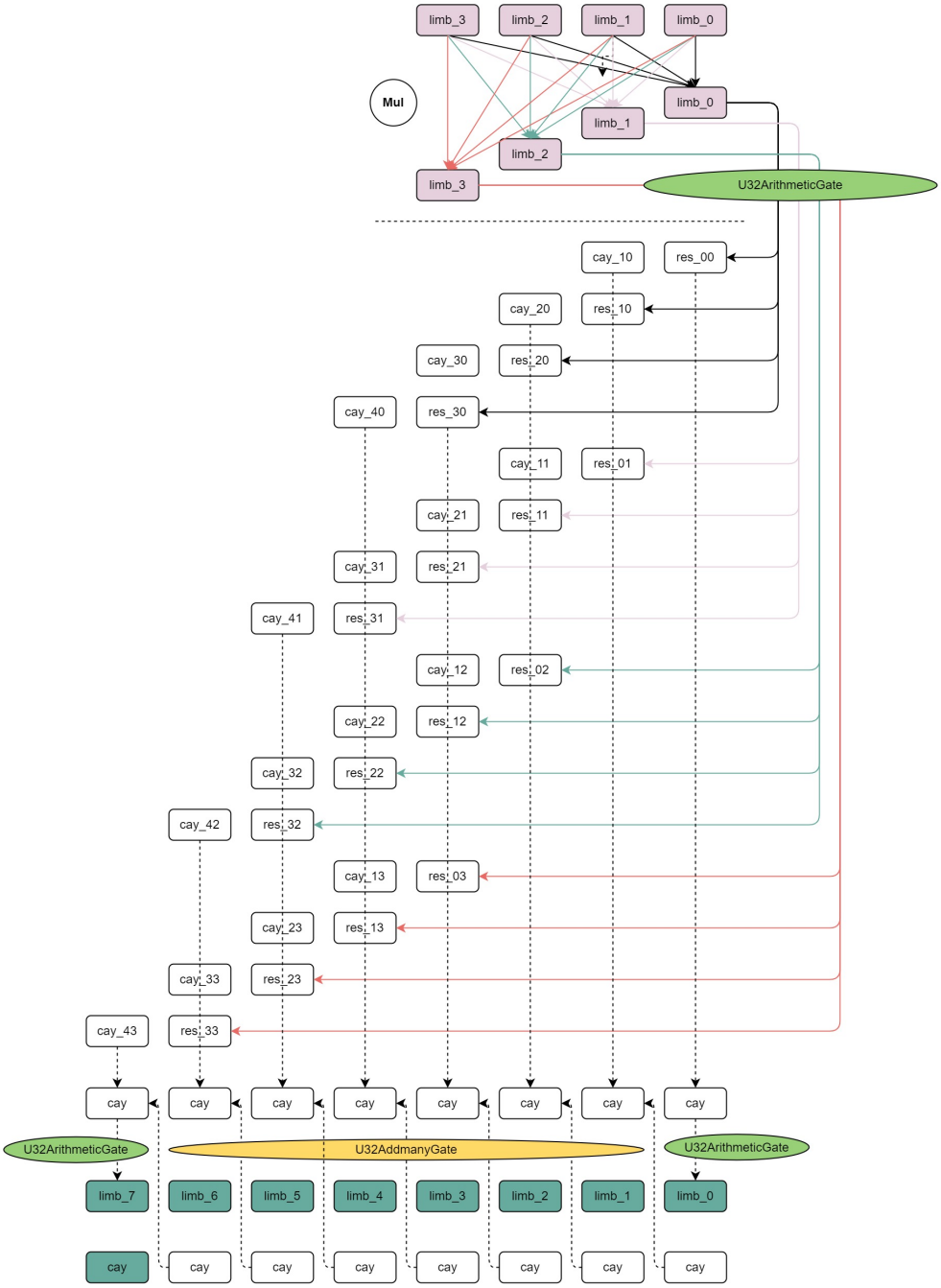


Figure 24: biguint-mul layout

3.1.4 biguint-div

Note that div-rem has the same constraints logic with div

Target Implement the division of two biguints.

Constraints logic

- Not implement div-algrithem directly;
- Use nondeterministic feature to check div-logic;

- Check `div * b + rem = a`;
- Check `rem < b`.

Process layout See **Figure 25**.

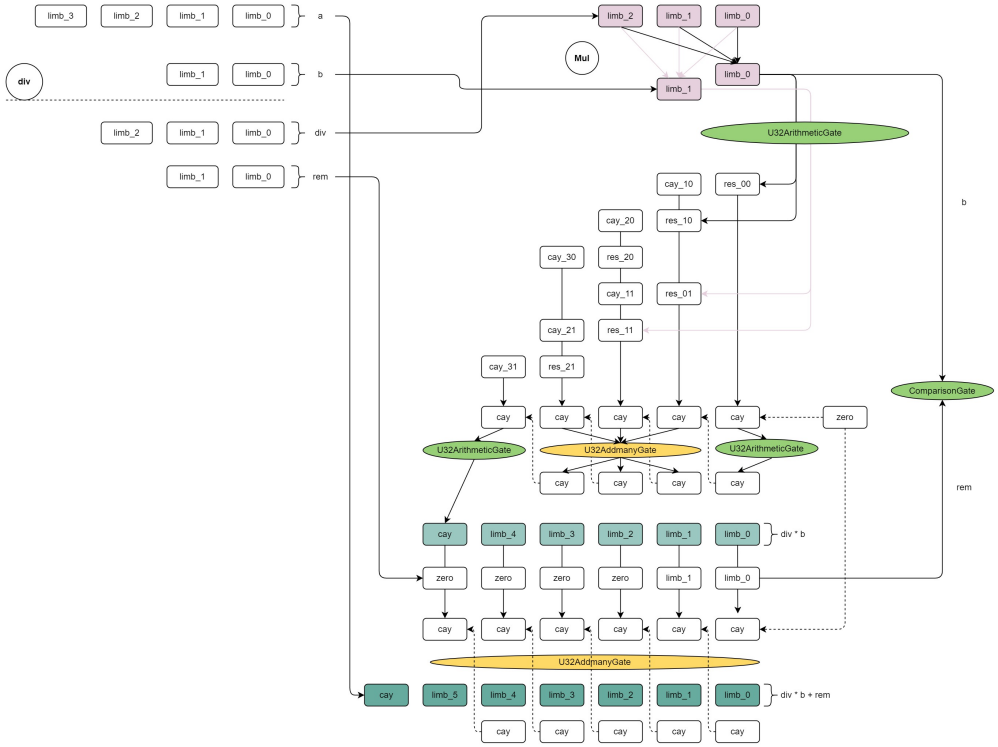


Figure 25: biguint-div layout

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

3.1.5 biguint-cmp

Target Implement the comparison of two biguints.

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.

Process layout See **Figure 26**.

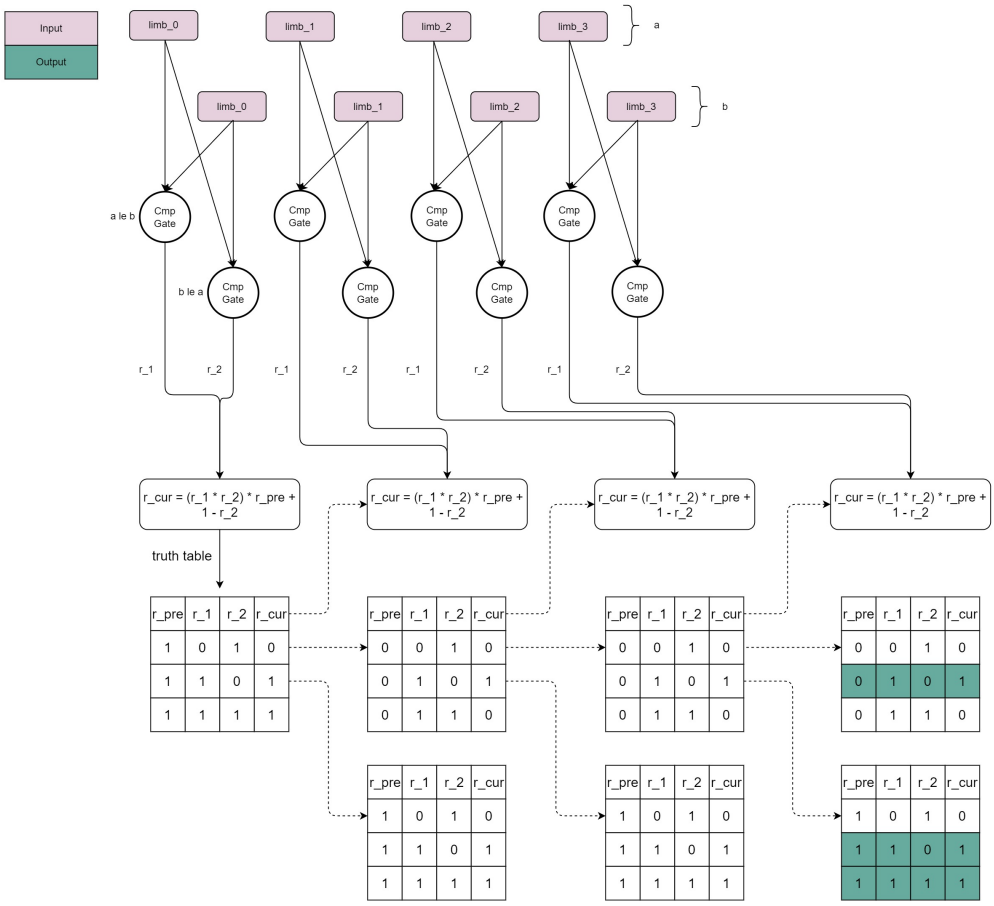


Figure 26: biguint-cmp layout

Circuit layout See **Figure 27**.

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

3.2 nonnative

3.2.1 nonnative-add

Target Check the additional relation among three nonnative target objects.

Constraints logic

- Check equation for gadget: `a + b = c + modular * overflow`;
- Check that "c lt modular".

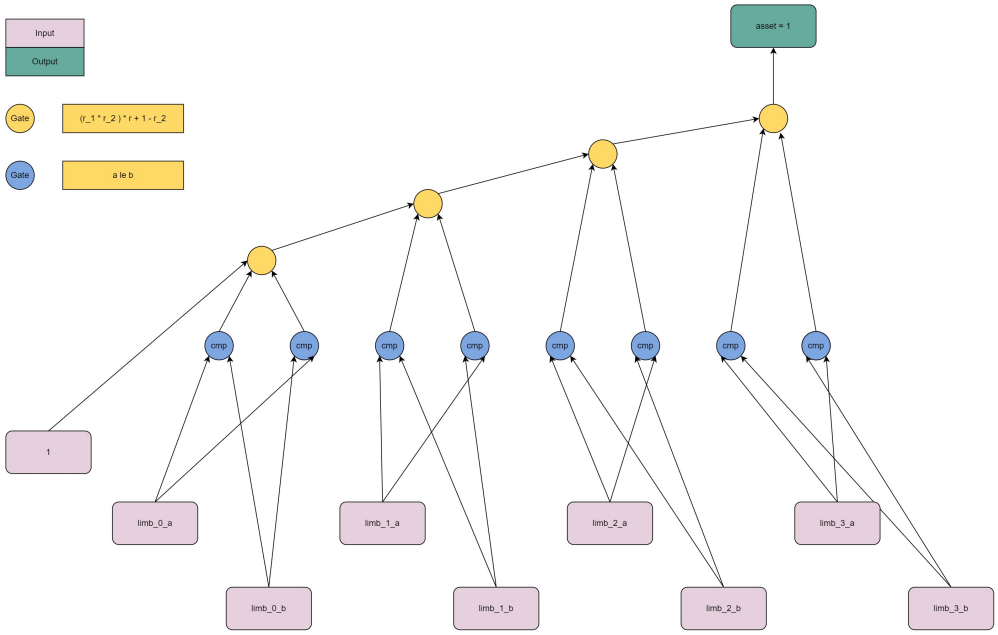


Figure 27: biguint-cmp circuit layout

Process layout See **Figure 28**.

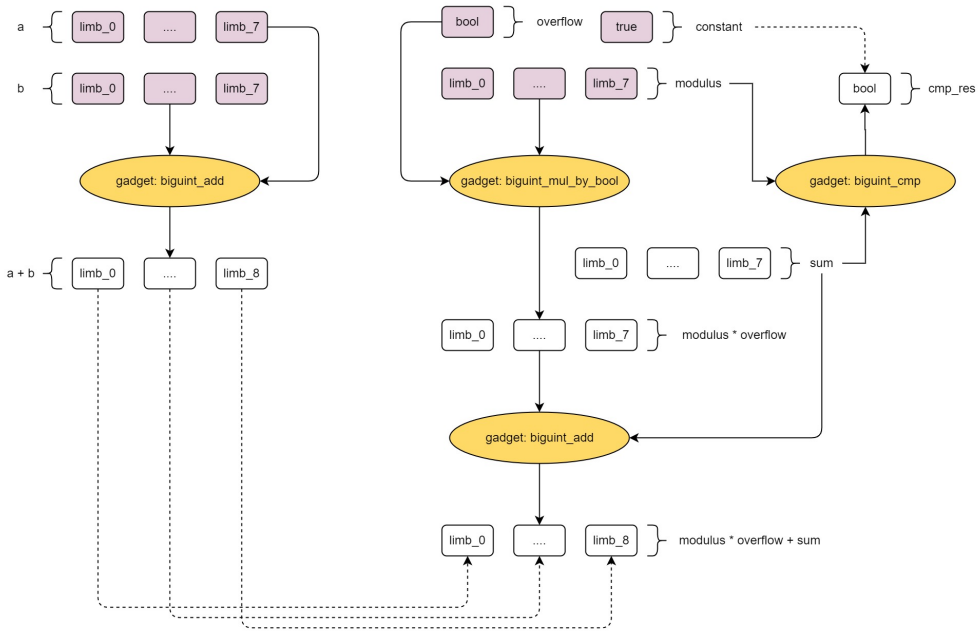


Figure 28: nonnative-add layout

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

Questions

- When set value to sum?

3.2.2 nonnative-sub

target Check the substract relation among three nonnative target objects.

Constraints logic

- Check equation for gadget: `diff + b = a + modular * overflow`;
- Check that "overflow is bool";
- Check that "diff.limbs is range U32".

Process layout See **Figure 29**

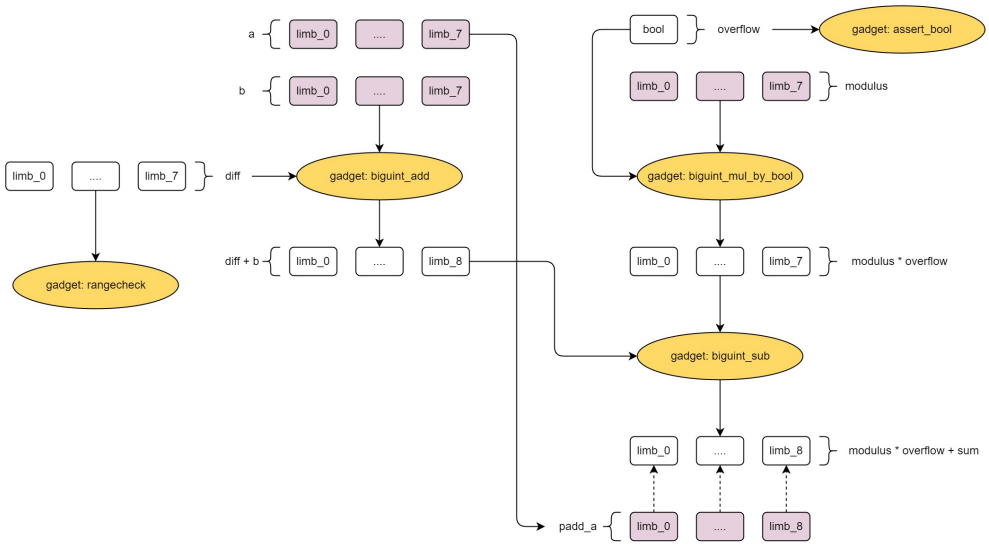


Figure 29: nonnative-sub layout

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 = 32biguint-add num + 27U32SubtractionGate + 9biguint-mul-by-bool + 8u32rangecheck + 4assert-bool + 9

Questions

- Why not constraint for overflow at nonnative-add?
- Why not make u32rangecheck for input at nonnative-add?

3.2.3 nonnative-mul

Target Check the multiplication relation among three nonnative target objects.

Constraints logic

- Check equation for gadget: `a * b = prod + modular * overflow`;
- Check that "overflow.limb is U32";
- Check that "prod.limb is U32".

Process layout See **Figure 30**.

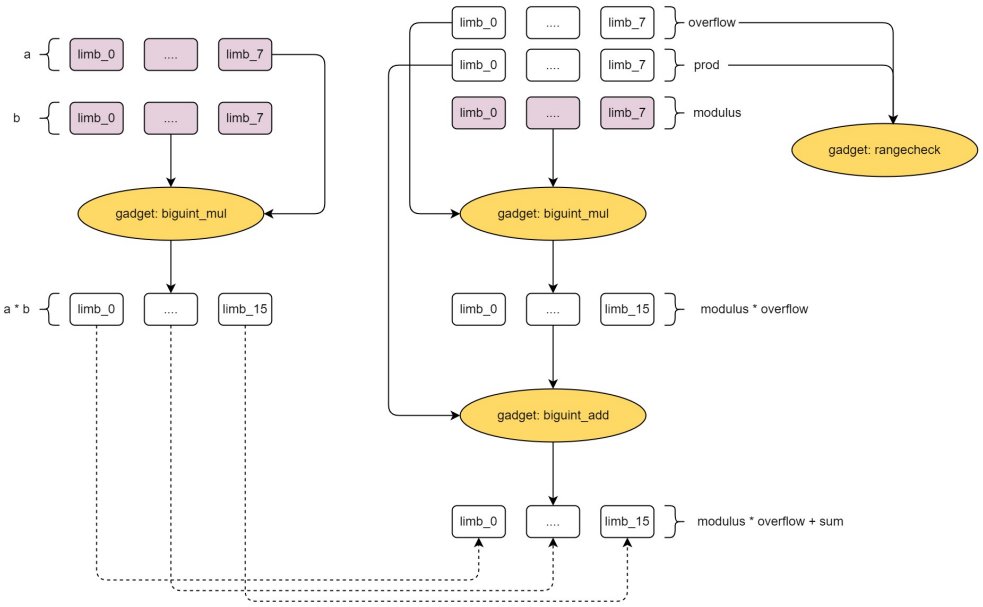


Figure 30: nonnative-mul layout

Constraints info and costs

- 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

3.2.4 nonnative-inv

Target Check the modular inverse relation among three nonnative target objects.

Constraints logic

- Check equation for gadget: `a * inv_a = 1 + modular * div`.

Process layout See **Figure 31**.

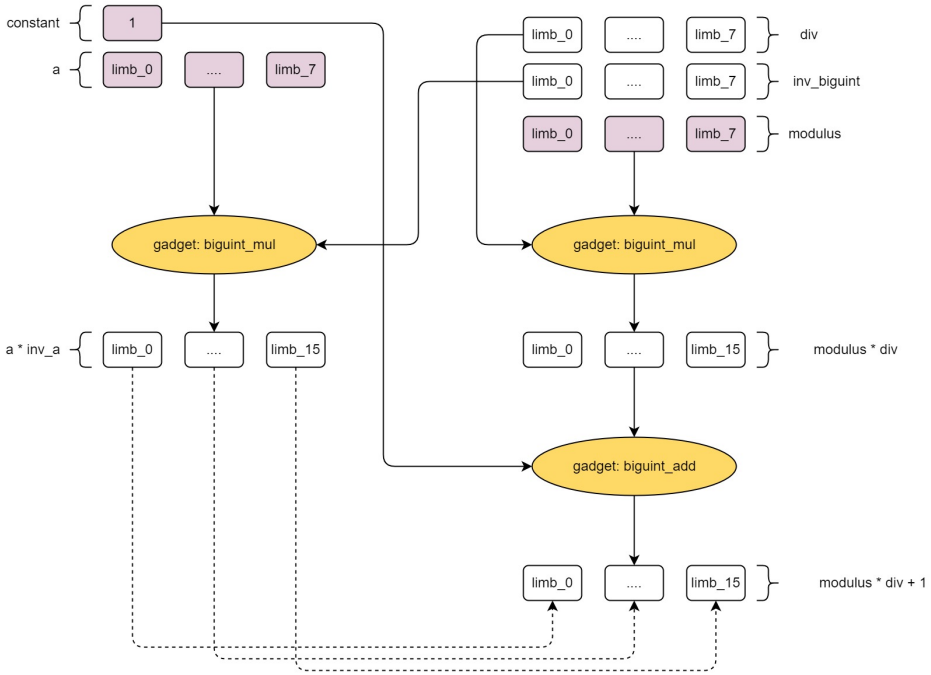


Figure 31: nonnative-inv layout

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

3.3 curve

3.3.1 curve-add

Target Implement the addition of two different curve points. this is a incomplete addition, you can refer to The halo2 Book [1] to learn more about it.

Constraints logic $(x_1, y_1) \neq (x_2, y_2)$, See **Figure 32**.

Process layout See **Figure 33**

Constraints info and costs

- gadget-sub-nonnative num: 5
- gadget-add-nonnative num: 1
- gadget-mul-nonnative num: 3
- gadget-inv-nonnative num: 1

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 32: curve-add

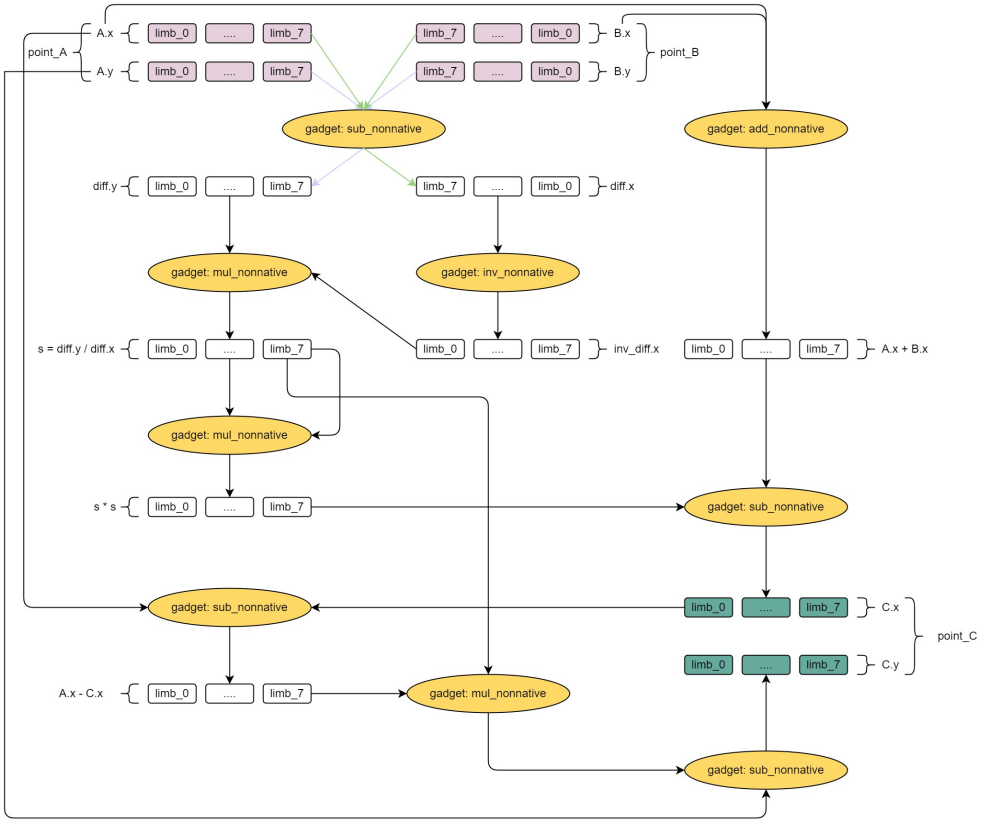


Figure 33: curve-add layout

- 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}

3.3.2 curve-double

target Implement the addition of two same curve points. this is a incomplete addition, you can refer to The halo2 Book [1] to learn more about it.

Constraints logic $(x_1, y_1) = (x_2, y_2)$. See **Figure 32**.

Process layout See **Figure 34**.

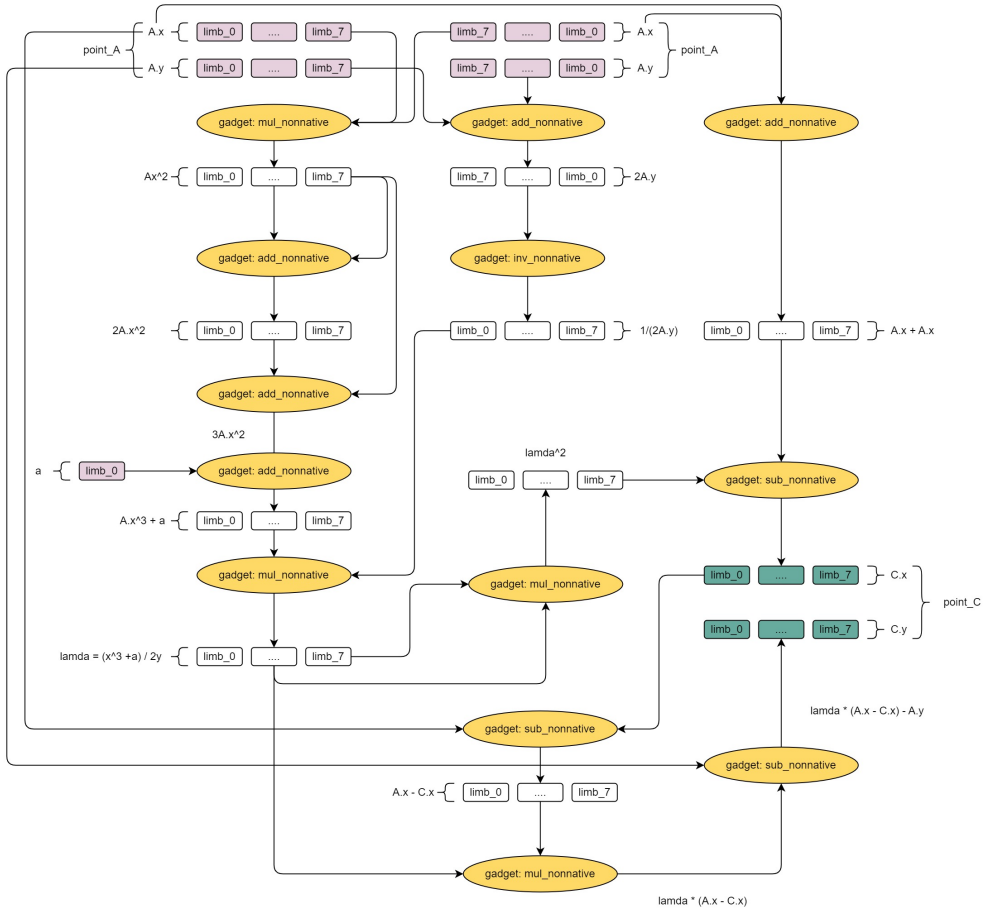


Figure 34: curve-double layout

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}

3.3.3 curve-assert-valid

Target Check point is on the curve $y^2 = x^3 + ax + b$.

Process layout See **Figure 35**.

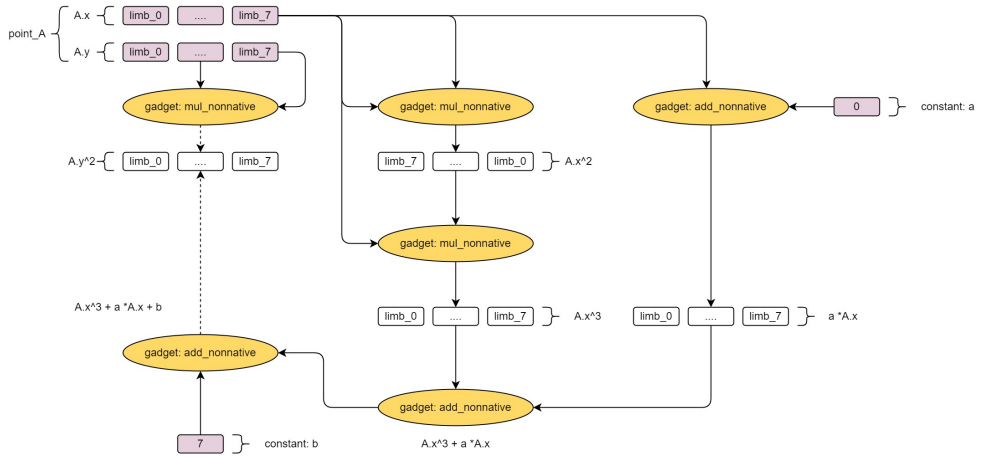


Figure 35: curve-assert valid layout

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}

3.3.4 curve-msm

target Implement the multiplication scalar multiplication (MSM).

Constraints logic See **Figure 36**.

Constraints info and costs

- gate type num: 14
- gate instance num: 147553

3.3.5 curve-scalar

Target Implement the multiplication between scalar and point.

Constraints logic See **Figure 37**.

Constraints info and costs

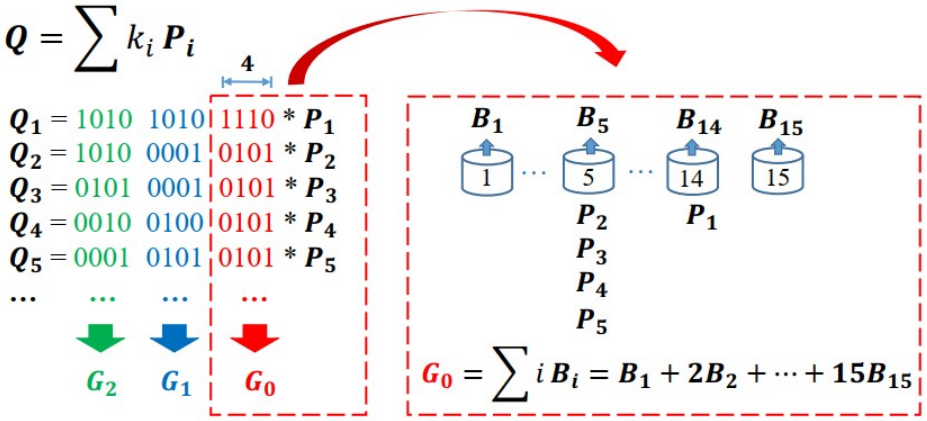


Figure 36: curve-msm

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: $Q \leftarrow [k]P$.

$Q \leftarrow \mathcal{O}$.
for $i = t - 2$ **to** 0 **do**
  $Q \leftarrow [2]Q$.
  if $k_i = 1$ **then**
    $Q \leftarrow Q + P$.
  end
end
return Q.

Figure 37: curve-scalar

- gate type num: 13
- gate instance num: 180364

3.4 ecdsa

3.4.1 ecdsa

Target Implement ecdsa verify process.

Constraints logic

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 38: ecdsa

4 Protocol

4.1 protocol

4.1.1 circuit-build

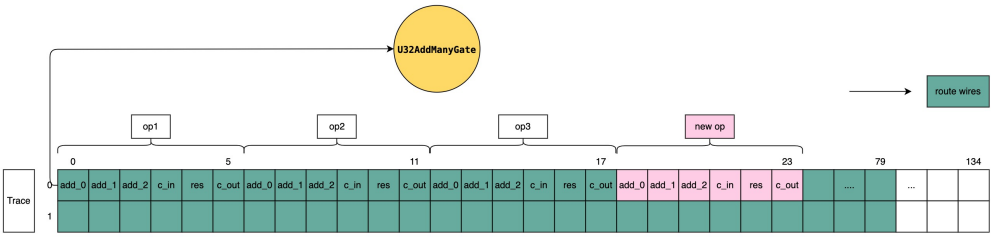
Plonky2 builds a full circuit with both prover and verifier data. The circuit description in Plonky2 is a data structure that describes the constraint relationship within the gate and among the gates, which is the same with Plonk.

One of the difference is Plonky2 makes extensive use of custom gates and generates a trace table with all used custom gates. Each row in the trace table represents an instance of a type of cutom gates, which can perform a specified operation designed by developers.

If the config supports enough routed wires, an instance of an arithmetic gate can support several such operations. When performing an arithmetic operation, Plonky2 will find an available slot, of the form (row and op) for it and first check if there is an exist gate instance in the circuit to use, rather than just add a new one.

For example, U32AddManyGate is a custom gate to perform addition on `num_addends` different 32-bit values, plus a small carry. Defaultly, each gate has 80 routed wires and 135 wires, if `num_addends` is two, then a U32AddManyGate can support five addition operations.

Suppose we need to perform a new addition of two 32-bit values, and there is an gate instance of such U32AddManyGate in the circuit, with only three operations used. Then we use its fourth op to perform the addition operation.



If no instance of such U32AddManyGate in the circuit or all instances of U32AddManyGate are full, then we need to add a new instance of U32AddManyGate with two addends to the circuit, which also adds a new row to the trace table.

Once all instances of custom gates are added to the circuit, the constraints of all gates are also determined in the trace table. To simplify generate filter polynomials in prove phase, see combined selector, Plonky2 calculates all

selector polynomials and related information in advance.

Before computing the selector polynomials, Plonky2 sorts all gates by their degrees and ID to make the ordering deterministic. Then partition the gates into several groups. For each group G, the number of gates in it plus each gate's degree are less than or equal to the max degree of configuration:

$$|G| + \max_{g \in G}\{\text{g.degree()}\} \leq \text{max\_degree}.$$

The reason behind it is in prove phase, Plonky2 will calculate the filter polynomials and multiply them with custom gates' constraint polynomials, the result is a polynomial whose degree is equal to $|G|$ + gate.degree(). These groups are constructed greedily from the list of gates sorted by degree and ID.

After finishing partitioning groups, Plonky2 builds a selector polynomial for each group. For instance, a circuit uses four ordered gates: add, sub, div and mul. The four gates are partitioned into two groups: g1 { add, sub } and g2 { div, mul }. Each gate has and only has one instance, that is there are four rows in its trace:

row	w0	w1	w2	w3	gate
0	a0	b0	c0	d0	div
1	a1	b1	c1	d1	mul
2	a2	b2	c2	d2	add
3	a3	b3	c3	d3	sub

To understand the process of building polynomials from groups, we first add a new column to the trace table to simulate the polynomial of group g1. For each row in the trace table, the value of the cell is the gate index if the gate is in g1, otherwise unused, which defined by Plonky2 is u32::MAX:

row	g1	w0	w1	w2	w3	gate
0	unused	a0	b0	c0	d0	div
1	unused	a1	b1	c1	d1	mul
2	0	a2	b2	c2	d2	add
3	1	a3	b3	c3	d3	sub

Then we add another column of g2 to the trace and fill the values with same rule:

row	g1	g2	w0	w1	w2	w3	gate
0	unused	2	a0	b0	c0	d0	div
1	unused	3	a1	b1	c1	d1	mul
2	0	unused	a2	b2	c2	d2	add
3	1	unused	a3	b3	c3	d3	sub

Now we have two new columns with four values, we can construct two selector polynomials from them easily.

Apart from selector polynomials, Plonky2 also generates several constant polynomials and sigma polynomials, then commits these polynomials all together and generates a merkle tree, which is used in FRI. The last thing calculated in circuit build phase is generators, which store all information of performing operations and generating constants in the circuit. They are used to generate full witness in prove phase.

At the end of circuit build, Plonky2 returns circuit data incluing three data structures:

Common data:

```
let common = CommonCircuitData {
    config: self.config, // circuit config
    fri_params, // fri config
    gates, // types of gates used in circuit
    selectors_info, // information of groups and selector polynomials
    quotient_degree_factor, // the degree of the PLONK quotient polynomial
    num_gate_constraints, // the largest number of constraints imposed by any gate
    num_constants, // the number of constant wires
    num_public_inputs, // the number of public inputs
    k_is, // shifts used in lde
    num_partial_products, // number of partial products in generating Z(x)
}
```

Prover data:

```
let prover_only = ProverOnlyCircuitData {
    generators: self.generators, // store information of all operations and used to
        generate full witness
    generator_indices_by_watches, // used to perform operations dependent on other
        operations
    constants_sigmas_commitment, // commitment of selector polynomials, constant
        polynomials and sigma polynomials
    sigmas: transpose_poly_values(sigma_vecs), // the transpose of the list of sigma
        polynomials
    subgroup, // subgroup of order `degree`
    public_inputs: self.public_inputs, // targets to be made public
    representative_map: forest.parents, // store information of copy constrains
    fft_root_table: Some(fft_root_table), // intermediate values used in fft
    circuit_digest, // a digest of the circuit, which can be used to seed Fiat-Shamir
}
```

Verifier data:

```
let verifier_only = VerifierOnlyCircuitData {
    constants_sigmas_cap, // a commitment to each constant polynomial and each
        permutation polynomial
    circuit_digest, // a digest of the circuit, which can be used to seed Fiat-Shamir
}
```

4.1.2 prove

4.1.3 verify

Bibliography

[1] **The halo2 Book**. URL: https://zcash.github.io/halo2/design/gadgets/ecc/addition.html.