# halo2

## Usage

This repository contains the halo2_proofs and halo2_gadgets crates, which should be used directly.

## Minimum Supported Rust Version

Requires Rust **1.60** or higher.

Minimum supported Rust version can be changed in the future, but it will be done with a minor version bump.

## Controlling parallelism

`halo2` currently uses rayon for parallel computation. The `RAYON_NUM_THREADS` environment variable can be used to set the number of threads.

You can disable `rayon` by disabling the `"multicore"` feature. Warning! Halo2 will lose access to parallelism if you disable the `"multicore"` feature. This will significantly degrade performance.

## License

### Contribution

Unless you explicitly state otherwise, any contribution intentionally submitted for inclusion in the work by you, as defined in the Apache-2.0 license, shall be dual licensed as above, without any additional terms or conditions.

# Concepts

First we'll describe the concepts behind zero-knowledge proof systems; the *arithmetization* (kind of circuit description) used by Halo 2; and the abstractions we use to build circuit implementations.

# Proof systems

The aim of any ***proof system*** is to be able to prove interesting mathematical or cryptographic ***statements***.

Typically, in a given protocol we will want to prove families of statements that differ in their ***public inputs***. The prover will also need to show that they know some ***private inputs*** that make the statement hold.

To do this we write down a ***relation***, $\mathcal{R}$, that specifies which combinations of public and private inputs are valid.

> The terminology above is intended to be aligned with the ZKProof Community Reference.

To be precise, we should distinguish between the relation $\mathcal{R}$, and its implementation to be used in a proof system. We call the latter a ***circuit***.

The language that we use to express circuits for a particular proof system is called an ***arithmetization***. Usually, an arithmetization will define circuits in terms of polynomial constraints on variables over a field.

> The *process* of expressing a particular relation as a circuit is also sometimes called "arithmetization", but we'll avoid that usage.

To create a proof of a statement, the prover will need to know the private inputs, and also intermediate values, called ***advice*** values, that are used by the circuit.

We assume that we can compute advice values efficiently from the private and public inputs. The particular advice values will depend on how we write the circuit, not only on the high-level statement.

The private inputs and advice values are collectively called a ***witness***.

> Some authors use "witness" as just a synonym for private inputs. But in our usage, a witness includes advice, i.e. it includes all values that the prover supplies to the circuit.

For example, suppose that we want to prove knowledge of a preimage $x$ of a hash function $H$ for a digest $y$:

- The private input would be the preimage $x$.
- The public input would be the digest $y$.
- The relation would be $\{(x, y) : H(x) = y\}$.
- For a particular public input $Y$, the statement would be: $\{(x) : H(x) = Y\}$.
- The advice would be all of the intermediate values in the circuit implementing the hash function. The witness would be $x$ and the advice.

A ***Non-interactive Argument*** allows a ***prover*** to create a ***proof*** for a given statement and witness. The proof is data that can be used to convince a ***verifier*** that *there exists* a witness for which the statement holds. The security property that such proofs cannot falsely convince a verifier is called ***soundness***.

A ***Non-interactive Argument of Knowledge*** (***NARK***) further convinces the verifier that the prover *knew* a witness for which the statement holds. This security property is called ***knowledge soundness***, and it implies soundness.

In practice knowledge soundness is more useful for cryptographic protocols than soundness: if we are interested in whether Alice holds a secret key in some protocol, say, we need Alice to prove that *she knows* the key, not just that it exists.

Knowledge soundness is formalized by saying that an ***extractor***, which can observe precisely how the proof is generated, must be able to compute the witness.

> This property is subtle given that proofs can be ***malleable***. That is, depending on the proof system it may be possible to take an existing proof (or set of proofs) and, without knowing the witness(es), modify it/them to produce a distinct proof of the same or a

related statement. Higher-level protocols that use malleable proof systems need to take this into account.

Even without malleability, proofs can also potentially be ***replayed***. For instance, we would not want Alice in our example to be able to present a proof generated by someone else, and have that be taken as a demonstration that she knew the key.

If a proof yields no information about the witness (other than that a witness exists and was known to the prover), then we say that the proof system is ***zero knowledge***.

If a proof system produces short proofs —i.e. of length polylogarithmic in the circuit size— then we say that it is ***succinct***. A succinct NARK is called a ***SNARK*** (***Succinct Non-Interactive Argument of Knowledge***).

By this definition, a SNARK need not have verification time polylogarithmic in the circuit size. Some papers use the term ***efficient*** to describe a SNARK with that property, but we'll avoid that term since it's ambiguous for SNARKs that support amortized or recursive verification, which we'll get to later.

A ***zk-SNARK*** is a zero-knowledge SNARK.

# PLONKish Arithmetization

The arithmetization used by Halo 2 comes from PLONK, or more precisely its extension UltraPLONK that supports custom gates and lookup arguments. We'll call it ***PLONKish***.

***PLONKish circuits*** are defined in terms of a rectangular matrix of values. We refer to ***rows***, ***columns***, and ***cells*** of this matrix with the conventional meanings.

A PLONKish circuit depends on a ***configuration***:

- A finite field $\mathbb{F}$, where cell values (for a given statement and witness) will be elements of $\mathbb{F}$.
- The number of columns in the matrix, and a specification of each column as being ***fixed***, ***advice***, or ***instance***. Fixed columns are fixed by the circuit; advice columns correspond to witness values; and instance columns are normally used for public inputs (technically, they can be used for any elements shared between the prover and verifier).
- A subset of the columns that can participate in equality constraints.
- A ***maximum constraint degree***.

- A sequence of ***polynomial constraints***. These are multivariate polynomials over $\mathbb{F}$ that must evaluate to zero *for each row*. The variables in a polynomial constraint may refer to a cell in a given column of the current row, or a given column of another row relative to this one (with wrap-around, i.e. taken modulo $n$). The maximum degree of each polynomial is given by the maximum constraint degree.
- A sequence of ***lookup arguments*** defined over tuples of ***input expressions*** (which are multivariate polynomials as above) and ***table columns***.

A PLONKish circuit also defines:

- The number of rows $n$ in the matrix. $n$ must correspond to the size of a multiplicative subgroup of $\mathbb{F}^\times$; typically a power of two.
- A sequence of ***equality constraints***, which specify that two given cells must have equal values.
- The values of the fixed columns at each row.

From a circuit description we can generate a ***proving key*** and a ***verification key***, which are needed for the operations of proving and verification for that circuit.

> Note that we specify the ordering of columns, polynomial constraints, lookup arguments, and equality constraints, even though these do not affect the meaning of the circuit. This makes it easier to define the generation of proving and verification keys as a deterministic process.

Typically, a configuration will define polynomial constraints that are switched off and on by ***selectors*** defined in fixed columns. For example, a constraint $q_i \cdot p(\ldots) = 0$ can be switched off for a particular row $i$ by setting $q_i = 0$. In this case we sometimes refer to a set of constraints controlled by a set of selector columns that are designed to be used together, as a ***gate***. Typically there will be a ***standard gate*** that supports generic operations like field multiplication and division, and possibly also ***custom gates*** that support more specialized operations.

# Chips

The previous section gives a fairly low-level description of a circuit. When implementing circuits we will typically use a higher-level API which aims for the desirable characteristics of auditability, efficiency, modularity, and expressiveness.

Some of the terminology and concepts used in this API are taken from an analogy with integrated circuit design and layout. As for integrated circuits, the above desirable

characteristics are easier to obtain by composing ***chips*** that provide efficient pre-built implementations of particular functionality.

For example, we might have chips that implement particular cryptographic primitives such as a hash function or cipher, or algorithms like scalar multiplication or pairings.

In PLONKish circuits, it is possible to build up arbitrary logic just from standard gates that do field multiplication and addition. However, very significant efficiency gains can be obtained by using custom gates.

Using our API, we define chips that "know" how to use particular sets of custom gates. This creates an abstraction layer that isolates the implementation of a high-level circuit from the complexity of using custom gates directly.

> Even if we sometimes need to "wear two hats", by implementing both a high-level circuit and the chips that it uses, the intention is that this separation will result in code that is easier to understand, audit, and maintain/reuse. This is partly because some potential implementation errors are ruled out by construction.

Gates in PLONKish circuits refer to cells by ***relative references***, i.e. to the cell in a given column, and the row at a given offset relative to the one in which the gate's selector is set. We call this an ***offset reference*** when the offset is nonzero (i.e. offset references are a subset of relative references).

Relative references contrast with ***absolute references*** used in equality constraints, which can point to any cell.

The motivation for offset references is to reduce the number of columns needed in the configuration, which reduces proof size. If we did not have offset references then we would need a column to hold each value referred to by a custom gate, and we would need to use equality constraints to copy values from other cells of the circuit into that column. With offset references, we not only need fewer columns; we also do not need equality constraints to be supported for all of those columns, which improves efficiency.

In R1CS (another arithmetization which may be more familiar to some readers, but don't worry if it isn't), a circuit consists of a "sea of gates" with no semantically significant ordering. Because of offset references, the order of rows in a PLONKish circuit, on the other hand, *is* significant. We're going to make some simplifying assumptions and define some abstractions to tame the resulting complexity: the aim will be that, at the gadget level where we do most of our circuit construction, we will not have to deal with relative references or with gate layout explicitly.

We will partition a circuit into ***regions***, where each region contains a disjoint subset of cells, and relative references only ever point *within* a region. Part of the responsibility of a chip

implementation is to ensure that gates that make offset references are laid out in the correct positions in a region.

Given the set of regions and their ***shapes***, we will use a separate ***floor planner*** to decide where (i.e. at what starting row) each region is placed. There is a default floor planner that implements a very general algorithm, but you can write your own floor planner if you need to.

Floor planning will in general leave gaps in the matrix, because the gates in a given row did not use all available columns. These are filled in —as far as possible— by gates that do not require offset references, which allows them to be placed on any row.

Chips can also define lookup tables. If more than one table is defined for the same lookup argument, we can use a ***tag column*** to specify which table is used on each row. It is also possible to perform a lookup in the union of several tables (limited by the polynomial degree bound).

## Composing chips

In order to combine functionality from several chips, we compose them in a tree. The top-level chip defines a set of fixed, advice, and instance columns, and then specifies how they should be distributed between lower-level chips.

In the simplest case, each lower-level chips will use columns disjoint from the other chips. However, it is allowed to share a column between chips. It is important to optimize the number of advice columns in particular, because that affects proof size.

The result (possibly after optimization) is a PLONKish configuration. Our circuit implementation will be parameterized on a chip, and can use any features of the supported lower-level chips via the top-level chip.

Our hope is that less expert users will normally be able to find an existing chip that supports the operations they need, or only have to make minor modifications to an existing chip. Expert users will have full control to do the kind of circuit optimizations that ECC is famous for 🙂.

# Gadgets

When implementing a circuit, we could use the features of the chips we've selected directly. Typically, though, we will use them via ***gadgets***. This indirection is useful because, for reasons of efficiency and limitations imposed by PLONKish circuits, the chip interfaces will often be dependent on low-level implementation details. The gadget interface can provide a more convenient and stable API that abstracts away from extraneous detail.

For example, consider a hash function such as SHA-256. The interface of a chip supporting SHA-256 might be dependent on internals of the hash function design such as the separation between message schedule and compression function. The corresponding gadget interface can provide a more convenient and familiar `update` / `finalize` API, and can also handle parts of the hash function that do not need chip support, such as padding. This is similar to how accelerated instructions for cryptographic primitives on CPUs are typically accessed via software libraries, rather than directly.

Gadgets can also provide modular and reusable abstractions for circuit programming at a higher level, similar to their use in libraries such as libsnark and bellman. As well as abstracting *functions*, they can also abstract *types*, such as elliptic curve points or integers of specific sizes.

# User Documentation

You're probably here because you want to write circuits? Excellent!

This section will guide you through the process of creating circuits with halo2.

# Developer tools

The `halo2` crate includes several utilities to help you design and implement your circuits.

## Mock prover

`halo2_proofs::dev::MockProver` is a tool for debugging circuits, as well as cheaply verifying their correctness in unit tests. The private and public inputs to the circuit are constructed as would normally be done to create a proof, but `MockProver::run` instead creates an object that will test every constraint in the circuit directly. It returns granular error messages that indicate which specific constraint (if any) is not satisfied.

## Circuit visualizations

The `dev-graph` feature flag exposes several helper methods for creating graphical representations of circuits.

On Debian systems, you will need the following additional packages:

```
sudo apt install cmake libexpat1-dev libfreetype6-dev libcairo2-dev
```

## Circuit layout

`halo2_proofs::dev::CircuitLayout` renders the circuit layout as a grid:

```
fn main() {
    // Prepare the circuit you want to render.
    // You don't need to include any witness variables.
    let a = Fp::random(OsRng);
    let instance = Fp::ONE + Fp::ONE;
    let lookup_table = vec![instance, a, a, Fp::zero()];
    let circuit: MyCircuit<Fp> = MyCircuit {
        a: Value::unknown(),
        lookup_table,
    };

    // Create the area you want to draw on.
    // Use SVGBackend if you want to render to .svg instead.
    use plotters::prelude::*;
    let root = BitMapBackend::new("layout.png", (1024, 768)).into_drawing_area();
    root.fill(&WHITE).unwrap();
    let root = root
        .titled("Example Circuit Layout", ("sans-serif", 60))
        .unwrap();

    halo2_proofs::dev::CircuitLayout::default()
        // You can optionally render only a section of the circuit.
        .view_width(0..2)
        .view_height(0..16)
        // You can hide labels, which can be useful with smaller areas.
        .show_labels(false)
        // Render the circuit onto your area!
        // The first argument is the size parameter for the circuit.
        .render(5, &circuit, &root)
        .unwrap();
}
```

- Columns are laid out from left to right as instance, advice and fixed. The order of columns is otherwise without meaning.
    - Instance columns have a white background.
    - Advice columns have a red background.
    - Fixed columns have a blue background.
- Regions are shown as labelled green boxes (overlaying the background colour). A region may appear as multiple boxes if some of its columns happen to not be adjacent.
- Cells that have been assigned to by the circuit will be shaded in grey. If any cells are assigned to more than once (which is usually a mistake), they will be shaded darker than

the surrounding cells.

### Circuit structure

`halo2_proofs::dev::circuit_dot_graph` builds a DOT graph string representing the given circuit, which can then be rendered with a variety of layout programs. The graph is built from calls to `Layouter::namespace` both within the circuit, and inside the gadgets and chips that it uses.

```
fn main() {
    // Prepare the circuit you want to render.
    // You don't need to include any witness variables.
    let a = Fp::rand();
    let instance = Fp::one() + Fp::one();
    let lookup_table = vec![instance, a, a, Fp::zero()];
    let circuit: MyCircuit<Fp> = MyCircuit {
        a: None,
        lookup_table,
    };

    // Generate the DOT graph string.
    let dot_string = halo2_proofs::dev::circuit_dot_graph(&circuit);

    // Now you can either handle it in Rust, or just
    // print it out to use with command-line tools.
    print!("{}", dot_string);
}
```

## Cost estimator

The `cost-model` binary takes high-level parameters for a circuit design, and estimates the verification cost, as well as resulting proof size.

```
Usage: cargo run --example cost-model -- [OPTIONS] k

Positional arguments:
  k                       2^K bound on the number of rows.

Optional arguments:
  -h, --help              Print this message.
  -a, --advice R[,R..]    An advice column with the given rotations. May be
repeated.
  -i, --instance R[,R..]  An instance column with the given rotations. May be
repeated.
  -f, --fixed R[,R..]     A fixed column with the given rotations. May be
repeated.
  -g, --gate-degree D     Maximum degree of the custom gates.
  -l, --lookup N,I,T      A lookup over N columns with max input degree I and max
table degree T. May be repeated.
  -p, --permutation N     A permutation over N columns. May be repeated.
```

For example, to estimate the cost of a circuit with three advice columns and one fixed column (with various rotations), and a maximum gate degree of 4:

```
> cargo run --example cost-model -- -a 0,1 -a 0 -a-0,-1,1 -f 0 -g 4 11
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/examples/cost-model -a 0,1 -a 0 -a 0,-1,1 -f 0 -g 4 11`
Circuit {
    k: 11,
    max_deg: 4,
    advice_columns: 3,
    lookups: 0,
    permutations: [],
    column_queries: 7,
    point_sets: 3,
    estimator: Estimator,
}
Proof size: 1440 bytes
Verification: at least 81.689ms
```

# A simple example

Let's start with a simple circuit, to introduce you to the common APIs and how they are used. The circuit will take a public input $c$, and will prove knowledge of two private inputs $a$ and $b$ such that

$$a^2 \cdot b^2 = c.$$

## Define instructions

Firstly, we need to define the instructions that our circuit will rely on. Instructions are the boundary between high-level gadgets and the low-level circuit operations. Instructions may be as coarse or as granular as desired, but in practice you want to strike a balance between an instruction being large enough to effectively optimize its implementation, and small enough that it is meaningfully reusable.

For our circuit, we will use three instructions:

- Load a private number into the circuit.
- Multiply two numbers.
- Expose a number as a public input to the circuit.

We also need a type for a variable representing a number. Instruction interfaces provide associated types for their inputs and outputs, to allow the implementations to represent these in a way that makes the most sense for their optimization goals.

```
trait NumericInstructions<F: Field>: Chip<F> {
    /// Variable representing a number.
    type Num;

    /// Loads a number into the circuit as a private input.
    fn load_private(&self, layouter: impl Layouter<F>, a: Value<F>) ->
Result<Self::Num, Error>;

    /// Loads a number into the circuit as a fixed constant.
    fn load_constant(&self, layouter: impl Layouter<F>, constant: F) ->
Result<Self::Num, Error>;

    /// Returns `c = a * b`.
    fn mul(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    /// Exposes a number as a public input to the circuit.
    fn expose_public(
        &self,
        layouter: impl Layouter<F>,
        num: Self::Num,
        row: usize,
    ) -> Result<(), Error>;
}
```

## Define a chip implementation

For our circuit, we will build a chip that provides the above numeric instructions for a finite field.

```
/// The chip that will implement our instructions! Chips store their own
/// config, as well as type markers if necessary.
struct FieldChip<F: Field> {
    config: FieldConfig,
    _marker: PhantomData<F>,
}
```

Every chip needs to implement the `Chip` trait. This defines the properties of the chip that a `Layouter` may rely on when synthesizing a circuit, as well as enabling any initial state that the chip requires to be loaded into the circuit.

```
impl<F: Field> Chip<F> for FieldChip<F> {
    type Config = FieldConfig;
    type Loaded = ();

    fn config(&self) -> &Self::Config {
        &self.config
    }

    fn loaded(&self) -> &Self::Loaded {
        &()
    }
}
```

## Configure the chip

The chip needs to be configured with the columns, permutations, and gates that will be required to implement all of the desired instructions.

```
/// Chip state is stored in a config struct. This is generated by the chip
/// during configuration, and then stored inside the chip.
#[derive(Clone, Debug)]
struct FieldConfig {
    /// For this chip, we will use two advice columns to implement our
instructions.
    /// These are also the columns through which we communicate with other parts
of
    /// the circuit.
    advice: [Column<Advice>; 2],

    /// This is the public input (instance) column.
    instance: Column<Instance>,

    // We need a selector to enable the multiplication gate, so that we aren't
placing
    // any constraints on cells where `NumericInstructions::mul` is not being
used.
    // This is important when building larger circuits, where columns are used by
    // multiple sets of instructions.
    s_mul: Selector,
}

impl<F: Field> FieldChip<F> {
    fn construct(config: <Self as Chip<F>>::Config) -> Self {
        Self {
            config,
            _marker: PhantomData,
        }
    }

    fn configure(
        meta: &mut ConstraintSystem<F>,
        advice: [Column<Advice>; 2],
        instance: Column<Instance>,
        constant: Column<Fixed>,
    ) -> <Self as Chip<F>>::Config {
        meta.enable_equality(instance);
        meta.enable_constant(constant);
        for column in &advice {
            meta.enable_equality(*column);
        }
        let s_mul = meta.selector();

        // Define our multiplication gate!
        meta.create_gate("mul", |meta| {
            // To implement multiplication, we need three advice cells and a
selector
            // cell. We arrange them like so:
            //
            // | a0  | a1  | s_mul |
            // |-----|-----|-------|
            // | lhs | rhs | s_mul |
```

```
            // | out |     |       |
            //
            // Gates may refer to any relative offsets we want, but each distinct
            // offset adds a cost to the proof. The most common offsets are 0 (the
            // current row), 1 (the next row), and -1 (the previous row), for
which
            // `Rotation` has specific constructors.
            let lhs = meta.query_advice(advice[0], Rotation::cur());
            let rhs = meta.query_advice(advice[1], Rotation::cur());
            let out = meta.query_advice(advice[0], Rotation::next());
            let s_mul = meta.query_selector(s_mul);

            // Finally, we return the polynomial expressions that constrain this
gate.
            // For our multiplication gate, we only need a single polynomial
constraint.
            //
            // The polynomial expressions returned from `create_gate` will be
            // constrained by the proving system to equal zero. Our expression
            // has the following properties:
            // - When s_mul = 0, any value is allowed in lhs, rhs, and out.
            // - When s_mul != 0, this constrains lhs * rhs = out.
            vec![s_mul * (lhs * rhs - out)]
        });

        FieldConfig {
            advice,
            instance,
            s_mul,
        }
    }
}
```

# Implement chip traits

```
/// A variable representing a number.
#[derive(Clone)]
struct Number<F: Field>(AssignedCell<F, F>);

impl<F: Field> NumericInstructions<F> for FieldChip<F> {
    type Num = Number<F>;

    fn load_private(
        &self,
        mut layouter: impl Layouter<F>,
        value: Value<F>,
    ) -> Result<Self::Num, Error> {
        let config = self.config();

        layouter.assign_region(
            || "load private",
            |mut region| {
                region
                    .assign_advice(|| "private input", config.advice[0], 0, ||
value)
                    .map(Number)
            },
        )
    }

    fn load_constant(
        &self,
        mut layouter: impl Layouter<F>,
        constant: F,
    ) -> Result<Self::Num, Error> {
        let config = self.config();

        layouter.assign_region(
            || "load constant",
            |mut region| {
                region
                    .assign_advice_from_constant(|| "constant value",
config.advice[0], 0, constant)
                    .map(Number)
            },
        )
    }

    fn mul(
        &self,
        mut layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error> {
        let config = self.config();
```

```
        layouter.assign_region(
            || "mul",
            |mut region: Region<'_, F>| {
                // We only want to use a single multiplication gate in this
region,
                // so we enable it at region offset 0; this means it will
constrain
                // cells at offsets 0 and 1.
                config.s_mul.enable(&mut region, 0)?;

                // The inputs we've been given could be located anywhere in the
circuit,
                // but we can only rely on relative offsets inside this region. So
we
                // assign new cells inside the region and constrain them to have
the
                // same values as the inputs.
                a.0.copy_advice(|| "lhs", &mut region, config.advice[0], 0)?;
                b.0.copy_advice(|| "rhs", &mut region, config.advice[1], 0)?;

                // Now we can assign the multiplication result, which is to be
assigned
                // into the output position.
                let value = a.0.value().copied() * b.0.value();

                // Finally, we do the assignment to the output, returning a
                // variable to be used in another part of the circuit.
                region
                    .assign_advice(|| "lhs * rhs", config.advice[0], 1, || value)
                    .map(Number)
            },
        )
    }

    fn expose_public(
        &self,
        mut layouter: impl Layouter<F>,
        num: Self::Num,
        row: usize,
    ) -> Result<(), Error> {
        let config = self.config();

        layouter.constrain_instance(num.0.cell(), config.instance, row)
    }
}
```

# Build the circuit

Now that we have the instructions we need, and a chip that implements them, we can finally build our circuit!

```
/// The full circuit implementation.
///
/// In this struct we store the private input variables. We use `Option<F>`
because
/// they won't have any value during key generation. During proving, if any of
these
/// were `None` we would get an error.
#[derive(Default)]
struct MyCircuit<F: Field> {
    constant: F,
    a: Value<F>,
    b: Value<F>,
}

impl<F: Field> Circuit<F> for MyCircuit<F> {
    // Since we are using a single chip for everything, we can just reuse its
config.
    type Config = FieldConfig;
    type FloorPlanner = SimpleFloorPlanner;

    fn without_witnesses(&self) -> Self {
        Self::default()
    }

    fn configure(meta: &mut ConstraintSystem<F>) -> Self::Config {
        // We create the two advice columns that FieldChip uses for I/O.
        let advice = [meta.advice_column(), meta.advice_column()];

        // We also need an instance column to store public inputs.
        let instance = meta.instance_column();

        // Create a fixed column to load constants.
        let constant = meta.fixed_column();

        FieldChip::configure(meta, advice, instance, constant)
    }

    fn synthesize(
        &self,
        config: Self::Config,
        mut layouter: impl Layouter<F>,
    ) -> Result<(), Error> {
        let field_chip = FieldChip::<F>::construct(config);

        // Load our private values into the circuit.
        let a = field_chip.load_private(layouter.namespace(|| "load a"), self.a)?;
        let b = field_chip.load_private(layouter.namespace(|| "load b"), self.b)?;

        // Load the constant factor into the circuit.
        let constant =
            field_chip.load_constant(layouter.namespace(|| "load constant"),
self.constant)?;
```

```
        // We only have access to plain multiplication.
        // We could implement our circuit as:
        //     asq  = a*a
        //     bsq  = b*b
        //     absq = asq*bsq
        //     c    = constant*asq*bsq
        //
        // but it's more efficient to implement it as:
        //     ab   = a*b
        //     absq = ab^2
        //     c    = constant*absq
        let ab = field_chip.mul(layouter.namespace(|| "a * b"), a, b)?;
        let absq = field_chip.mul(layouter.namespace(|| "ab * ab"), ab.clone(),
ab)?;
        let c = field_chip.mul(layouter.namespace(|| "constant * absq"), constant,
absq)?;

        // Expose the result as a public input to the circuit.
        field_chip.expose_public(layouter.namespace(|| "expose c"), c, 0)
    }
}
```

## Testing the circuit

`halo2_proofs::dev::MockProver` can be used to test that the circuit is working correctly. The private and public inputs to the circuit are constructed as we will do to create a proof, but by passing them to `MockProver::run` we get an object that can test every constraint in the circuit, and tell us exactly what is failing (if anything).

```
    // The number of rows in our circuit cannot exceed 2^k. Since our example
    // circuit is very small, we can pick a very small value here.
    let k = 4;

    // Prepare the private and public inputs to the circuit!
    let constant = Fp::from(7);
    let a = Fp::from(2);
    let b = Fp::from(3);
    let c = constant * a.square() * b.square();

    // Instantiate the circuit with the private inputs.
    let circuit = MyCircuit {
        constant,
        a: Value::known(a),
        b: Value::known(b),
    };

    // Arrange the public input. We expose the multiplication result in row 0
    // of the instance column, so we position it there in our public inputs.
    let mut public_inputs = vec![c];

    // Given the correct public input, our circuit will verify.
    let prover = MockProver::run(k, &circuit, vec!
[public_inputs.clone()]).unwrap();
    assert_eq!(prover.verify(), Ok(()));

    // If we try some other public input, the proof will fail!
    public_inputs[0] += Fp::one();
    let prover = MockProver::run(k, &circuit, vec![public_inputs]).unwrap();
    assert!(prover.verify().is_err());
```

### Full example

You can find the source code for this example here.

# Lookup tables

In normal programs, you can trade memory for CPU to improve performance, by pre-computing and storing lookup tables for some part of the computation. We can do the same thing in halo2 circuits!

A lookup table can be thought of as enforcing a *relation* between variables, where the relation is expressed as a table. Assuming we have only one lookup argument in our constraint system, the total size of tables is constrained by the size of the circuit: each table entry costs one row, and it also costs one row to do each lookup.

TODO

# Gadgets

# Tips and tricks

This section contains various ideas and snippets that you might find useful while writing halo2 circuits.

## Small range constraints

A common constraint used in R1CS circuits is the boolean constraint: $b * (1 - b) = 0$. This constraint can only be satisfied by $b = 0$ or $b = 1$.

In halo2 circuits, you can similarly constrain a cell to have one of a small set of values. For example, to constrain $a$ to the range $[0..5]$, you would create a gate of the form:

$$a \cdot (1 - a) \cdot (2 - a) \cdot (3 - a) \cdot (4 - a) = 0$$

while to constrain $c$ to be either 7 or 13, you would use:

$$(7 - c) \cdot (13 - c) = 0$$

> The underlying principle here is that we create a polynomial constraint with roots at each value in the set of possible values we want to allow. In R1CS circuits, the maximum supported polynomial degree is 2 (due to all constraints being of the form $a * b = c$). In halo2 circuits, you can use arbitrary-degree polynomials - with the proviso that higher-degree constraints are more expensive to use.

Note that the roots don't have to be constants; for example $(a - x) \cdot (a - y) \cdot (a - z) = 0$ will constrain $a$ to be equal to one of $\{x, y, z\}$ where the latter can be arbitrary polynomials, as long as the whole expression stays within the maximum degree bound.

## Small set interpolation

We can use Lagrange interpolation to create a polynomial constraint that maps $f(X) = Y$ for small sets of $X \in \{x_i\}, Y \in \{y_i\}$.

For instance, say we want to map a 2-bit value to a "spread" version interleaved with zeros. We first precompute the evaluations at each point:

$$\begin{aligned} 00 \to 0000 \quad &\implies \quad 0 \to 0 \\ 01 \to 0001 \quad &\implies \quad 1 \to 1 \\ 10 \to 0100 \quad &\implies \quad 2 \to 4 \\ 11 \to 0101 \quad &\implies \quad 3 \to 5 \end{aligned}$$

Then, we construct the Lagrange basis polynomial for each point using the identity:

$$l_j(X) = \prod_{0 \le m < k,\, m \neq j} \frac{x - x_m}{x_j - x_m},$$

where $k$ is the number of data points. ($k = 4$ in our example above.)

Recall that the Lagrange basis polynomial $l_j(X)$ evaluates to $1$ at $X = x_j$ and $0$ at all other $x_i, j \neq i$.

Continuing our example, we get four Lagrange basis polynomials:

$$\begin{aligned} l_0(X) &= \frac{(X-3)(X-2)(X-1)}{(-3)(-2)(-1)} \\ l_1(X) &= \frac{(X-3)(X-2)(X)}{(-2)(-1)(1)} \\ l_2(X) &= \frac{(X-3)(X-1)(X)}{(-1)(1)(2)} \\ l_3(X) &= \frac{(X-2)(X-1)(X)}{(1)(2)(3)} \end{aligned}$$

Our polynomial constraint is then

$$\begin{aligned} & f(0) \cdot l_0(X) + f(1) \cdot l_1(X) + f(2) \cdot l_2(X) + f(3) \cdot l_3(X) - f(X \\ \implies & 0 \cdot l_0(X) + 1 \cdot l_1(X) + 4 \cdot l_2(X) + 5 \cdot l_3(X) - f(X \end{aligned}$$

# Using halo2 in WASM

Since halo2 is written in Rust, you can compile it to WebAssembly (wasm), which will allow you to use the prover and verifier for your circuits in browser applications. This tutorial takes you through all you need to know to compile your circuits to wasm.

Throughout this tutorial, we will follow the repository for Zordle for reference, one of the first known webapps based on Halo 2 circuits. Zordle is ZK Wordle, where the circuit takes as advice values the player's input words and the player's share grid (the grey, yellow and green squares) and verifies that they match correctly. Therefore, the proof verifies that the player knows a "preimage" to the output share sheet, which can then be verified using just the ZK proof.

## Circuit code setup

The first step is to create functions in Rust that will interface with the browser application. In the case of a prover, this will typically input some version of the advice and instance data, use it to generate a complete witness, and then output a proof. In the case of a verifier, this will typically input a proof and some version of the instance, and then output a boolean indicating whether the proof verified correctly or not.

In the case of Zordle, this code is contained in wasm.rs, and consists of two primary functions:

### Prover

```
#[wasm_bindgen]
pub async fn prove_play(final_word: String, words_js: JsValue, params_ser:
JsValue) -> JsValue {
  // Steps:
  // - Deserialise function parameters
  // - Generate the instance and advice columns using the words
  // - Instantiate the circuit and generate the witness
  // - Generate the proving key from the params
  // - Create a proof
}
```

While the specific inputs and their serialisations will depend on your circuit and webapp set up, it's useful to note the format in the specific case of Zordle since your use case will likely be similar:

This function takes as input the `final_word` that the user aimed for, and the words they attempted to use (in the form of `words_js`). It also takes as input the parameters for the circuit, which are serialized in `params_ser`. We will expand on this in the Params section below.

Note that the function parameters are passed in `wasm_bindgen`-compatible formats: `String` and `JsValue`. The `JsValue` type is a type from the `Serde` library. You can find much more details about this type and how to use it in the documentation here.

The output is a `Vec<u8>` converted to a `JSValue` using Serde. This is later passed in as input to the the verifier function.

## Verifier

```
#[wasm_bindgen]
pub fn verify_play(final_word: String, proof_js: JsValue, diffs_u64_js: JsValue,
params_ser: JsValue) -> bool {
  // Steps:
  // - Deserialise function parameters
  // - Generate the instance columns using the diffs representation of the columns
  // - Generate the verifying key using the params
  // - Verify the proof
}
```

Similar to the prover, we take in input and output a boolean true/false indicating the correctness of the proof. The `diffs_u64_js` object is a 2D JS array consisting of values for each cell that indicate the color: grey, yellow or green. These are used to assemble the instance columns for the circuit.

## Params

Additionally, both the prover and verifier functions input `params_ser`, a serialised form of the public parameters of the polynomial commitment scheme. These are passed in as input (instead of being regenerated in prove/verify functions) as a performance optimisation since these are constant based only on the circuit's value of `K`. We can store these separately on a static web server and pass them in as input to the WASM. To generate the binary serialised form of these (separately outside the WASM functions), you can run something like:

```
fn write_params(K: u32) {
    let mut params_file = File::create("params.bin").unwrap();
    let params: Params<EqAffine> = Params::new(K);
    params.write(&mut params_file).unwrap();
}
```

Later, we can read the `params.bin` file from the web-server in Javascript in a byte-serialised format as a `Uint8Array` and pass it to the WASM as `params_ser`, which can be deserialised in Rust using the `js_sys` library.

Ideally, in future, instead of serialising the parameters we would be able to serialise and work directly with the proving key and the verifying key of the circuit, but that is currently not supported by the library, and tracked as issue #449 and #443.

# Rust and WASM environment setup

Typically, Rust code is compiled to WASM using the `wasm-pack` tool and is as simple as changing some build commands. In the case of halo2 prover/verifier functions however, we need to make some additional changes to the build process. In particular, there are two main changes:

- **Parallelism**: halo2 uses the `rayon` library for parallelism, which is not directly supported by WASM. However, the Chrome team has an adapter to enable rayon-like parallelism using Web Workers in browser: `wasm-bindgen-rayon`. We'll use this to enable parallelism in our WASM prover/verifier.
- **WASM max memory**: The default memory limit for WASM with `wasm-bindgen` is set to 2GB, which is not enough to run the halo2 prover for large circuits (with `K` > 10 or so). We need to increase this limit to the maximum allowed by WASM (4GB!) to support larger circuits (up to `K = 15` or so).

Firstly, add all the dependencies particular to your WASM interfacing functions to your `Cargo.toml` file. You can restrict the dependencies to the WASM compilation by using the WASM target feature flag. In the case of Zordle, this looks like:

```
[target.'cfg(target_family = "wasm")'.dependencies]
getrandom = { version = "0.2", features = ["js"]}
wasm-bindgen = { version = "0.2.81", features = ["serde-serialize"]}
console_error_panic_hook = "0.1.7"
rayon = "1.5"
wasm-bindgen-rayon = { version = "1.0"}
web-sys = { version = "0.3", features = ["Request", "Window", "Response"] }
wasm-bindgen-futures = "0.4"
js-sys = "0.3"
```

Next, let's integrate `wasm-bindgen-rayon` into our code. The README for the library has a great overview of how to do so. In particular, note the changes to the Rust compilation pipeline. You need to switch to a nightly version of Rust and enable support for WASM atomics. Additionally, remember to export the `init_thread_pool` in Rust code.

Next, we will bump up the default 2GB max memory limit for `wasm-pack`. To do so, add `"-C", "link-arg=--max-memory=4294967296"` Rust flag to the wasm target in the `.cargo/config` file. With the setup for `wasm-bindgen-rayon` and the memory bump, the `.cargo/config` file should now look like:

```
[target.wasm32-unknown-unknown]
rustflags = ["-C", "target-feature=+atomics,+bulk-memory,+mutable-globals", "-C",
"link-arg=--max-memory=4294967296"]
...
```

Shoutout to @mattgibb who documented this esoteric change for increasing maximum memory in a random GitHub issue here.[1]

[1] Off-topic but it was quite surprising for me to learn that WASM has a hard maximum limitation of 4GB memory. This is because WASM currently has a 32-bit architecture, which was quite surprising to me for such a new, forward-facing assembly language. There are, however, some open proposals to move WASM to a larger address space.

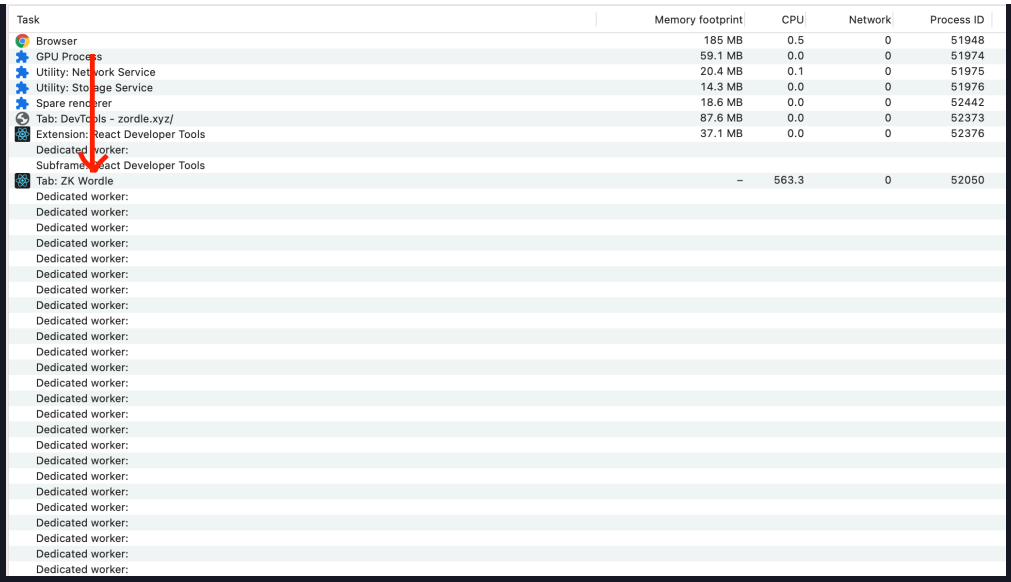
Now that we have the Rust set up, you should be able to build a WASM package simply using `wasm-pack build --target web --out-dir pkg` and use the output WASM package in your webapp.

## Webapp setup

Zordle ships with a minimal React test client as an example (that simply adds WASM support to the default `create-react-app` template). You can find the code for the test client here. I would recommend forking the test client for your own application and working from there.

The test client includes a clean WebWorker that interfaces with the Rust WASM package. Putting the interface in a WebWorker prevents blocking the main thread of the browser and allows for a clean interface from React/application logic. Checkout `halo-worker.ts` for the WebWorker code and see how you can interface with the web worker from React in `App.tsx`.

If you've done everything right so far, you should now be able to generate proofs and verify them in browser! In the case of Zordle, proof generation for a circuit with `K = 14` takes about a minute or so on my laptop. During proof generation, if you pop open the Chrome/Firefox task manager, you should additionally see something like this:




Zordle and its test-client set the parallelism to the number of cores available on the machine by default. If you would like to reduce this, you can do so by changing the argument to `initThreadPool`.

If you'd prefer to use your own Worker/React setup, the code to fetch and serialise parameters, proofs and other instance and advice values may still be useful to look at!

## Safari

Note that `wasm-bindgen-rayon` library is not supported by Safari because it spawns Web Workers from inside another Web Worker. According to the relevant Webkit issue, support for this feature had made it into Safari Technology Preview by November 2022, and indeed the Release Notes for Safari Technology Preview Release 155 claim support, so it is worth checking whether this has made it into Safari if that is important to you.

## Debugging

Often, you'll run into issues with your Rust code and see that the WASM execution errors with `Uncaught (in promise) RuntimeError: unreachable`, a wholly unhelpful error for debugging. This is because the code is compiled in release mode which strips out error messages as a performance optimisation. To debug, you can build the WASM package in debug mode using the flag `--dev` with `wasm-pack build`. This will build in debug mode, slowing down execution significantly but allowing you to see any runtime error messages in the browser console. Additionally, you can install the `console_error_panic_hook` crate (as is done by Zordle) to also get helpful debug messages for runtime panics.

## Credits

This guide was written by Nalin. Thanks additionally to Uma and Blaine for significant work on figuring out these steps. Feel free to reach out to me if you have trouble with any of these steps.

# Developer Documentation

You want to contribute to the Halo 2 crates? Awesome!

This section covers information about our development processes and review standards, and useful tips for maintaining and extending the codebase.

# Feature development

Sometimes feature development can require iterating on a design over time. It can be useful to start using features in downstream crates early on to gain experience with the APIs and functionality, that can feed back into the feature's design prior to it being stabilised. To enable this, we follow a three-stage `nightly -> beta -> stable` development pattern inspired by (but not identical to) the Rust compiler.

## Feature flags

Each unstabilised feature has a default-off feature flag that enables it, of the form `unstable-*`. The stable API of the crates must not be affected when the feature flag is disabled, except for specific complex features that will be considered on a case-by-case basis.

Two meta-flags are provided to enable all features at a particular stabilisation level:

- `beta` enables all features at the "beta" stage (and implicitly all features at the "stable" stage).
- `nightly` enables all features at the "beta" and "nightly" stages (and implicitly all features at the "stable" stage), i.e. all features are enabled.
- When neither flag is enabled (and no feature-specific flags are enabled), then in effect only features at the "stable" stage are enabled.

## Feature workflow

- If the maintainers have rough consensus that an experimental feature is generally desired, its initial implementation can be merged into the codebase optimistically behind a feature-specific feature flag with a lower standard of review. The feature's flag is added to the `nightly` feature flag set.
  - The feature will become usable by downstream published crates in the next general release of the `halo2` crates.
  - Subsequent development and refinement of the feature can be performed in-situ via additional PRs, along with additional review.
  - If the feature ends up having bad interactions with other features (in particular, already-stabilised features), then it can be removed later without affecting the stable or beta APIs.
- Once the feature has had sufficient review, and is at the point where a `halo2` user considers it production-ready (and is willing or planning to deploy it to production), the feature's feature flag is moved to the `beta` feature flag set.
- Once the feature has had review equivalent to the stable review policy, and there is rough consensus that the feature is useful to the wider `halo2` userbase, the feature's feature flag is removed and the feature becomes part of the main maintained codebase.

> For more complex features, the above workflow might be augmented with `beta` and `nightly` branches; this will be figured out once a feature requiring this is proposed as a candidate for inclusion.

## In-progress features

| Feature flag | Stage | Notes |
|---|---|---|
| `unstable-sha256-gadget` | `nightly` | The SHA-256 gadget and chip. |

# Design

## Note on Language

We use slightly different language than others to describe PLONK concepts. Here's the overview:

1. We like to think of PLONK-like arguments as tables, where each column corresponds to a "wire". We refer to entries in this table as "cells".
2. We like to call "selector polynomials" and so on "fixed columns" instead. We then refer specifically to a "selector constraint" when a cell in a fixed column is being used to control whether a particular constraint is enabled in that row.
3. We call the other polynomials "advice columns" usually, when they're populated by the prover.
4. We use the term "rule" to refer to a "gate" like

$$A(X) \cdot q_A(X) + B(X) \cdot q_B(X) + A(X) \cdot B(X) \cdot q_M(X) + C(X) \cdot q_C(X) = 0.$$

    - TODO: Check how consistent we are with this, and update the code and docs to match.

# Proving system

The Halo 2 proving system can be broken down into five stages:

1. Commit to polynomials encoding the main components of the circuit:
    - Cell assignments.
    - Permuted values and products for each lookup argument.
    - Equality constraint permutations.
2. Construct the vanishing argument to constrain all circuit relations to zero:
    - Standard and custom gates.
    - Lookup argument rules.
    - Equality constraint permutation rules.
3. Evaluate the above polynomials at all necessary points:
    - All relative rotations used by custom gates across all columns.
    - Vanishing argument pieces.
4. Construct the multipoint opening argument to check that all evaluations are consistent with their respective commitments.
5. Run the inner product argument to create a polynomial commitment opening proof for the multipoint opening argument polynomial.

These stages are presented in turn across this section of the book.

## Example

To aid our explanations, we will at times refer to the following example constraint system:

- Four advice columns $a, b, c, d$.

- One fixed column $f$.
- Three custom gates:
  - $a \cdot b \cdot c_{-1} - d = 0$
  - $f_{-1} \cdot c = 0$
  - $f \cdot d \cdot a = 0$

## tl;dr

The table below provides a (probably too) succinct description of the Halo 2 protocol. This description will likely be replaced by the Halo 2 paper and security proof, but for now serves as a summary of the following sub-sections.

| Prover | | Verifier |
|---|---|---|
| | $\leftarrow$ | $t(X) = (X^n - 1)$ |
| | $\leftarrow$ | $F = [F_0, F_1, \ldots, F_{m-1}]$ |
| $\mathbf{A} = [A_0, A_1, \ldots, A_{m-1}]$ | $\rightarrow$ | |
| | $\leftarrow$ | $\theta$ |
| $\mathbf{L} = [(A'_0, S'_0), \ldots, (A'_{m-1}, S'_{m-1})]$ | $\rightarrow$ | |
| | $\leftarrow$ | $\beta, \gamma$ |
| $\mathbf{Z_P} = [Z_{P,0}, Z_{P,1}, \ldots]$ | $\rightarrow$ | |
| $\mathbf{Z_L} = [Z_{L,0}, Z_{L,1}, \ldots]$ | $\rightarrow$ | |
| | $\leftarrow$ | $y$ |
| $h(X) = \frac{\text{gate}_0(X) + \cdots + y^i \cdot \text{gate}_i(X)}{t(X)}$ | | |
| $h(X) = h_0(X) + \cdots + X^{n(d-1)} h_{d-1}(X)$ | | |
| $\mathbf{H} = [H_0, H_1, \ldots, H_{d-1}]$ | $\rightarrow$ | |
| | $\leftarrow$ | $x$ |
| $evals = [A_0(x), \ldots, H_{d-1}(x)]$ | $\rightarrow$ | |
| | | Checks $h(x)$ |
| | $\leftarrow$ | $x_1, x_2$ |
| Constructs $h'(X)$ multipoint opening poly | | |
| $U = \text{Commit}(h'(X))$ | $\rightarrow$ | |
| | $\leftarrow$ | $x_3$ |

| Prover | | Verifier |
|---|---|---|
| $\mathbf{q}_{\text{evals}} = [Q_0(x_3), Q_1(x_3), \dots]$ | $\rightarrow$ | |
| $u_{\text{eval}} = U(x_3)$ | $\rightarrow$ | |
| | $\leftarrow$ | $x_4$ |

Then the prover and verifier:

- Construct $\text{finalPoly}(X)$ as a linear combination of $\mathbf{Q}$ and $U$ using powers of $x_4$;
- Construct $\text{finalPolyEval}$ as the equivalent linear combination of $\mathbf{q}_{\text{evals}}$ and $u_{\text{eval}}$; and
- Perform $\text{InnerProduct}(\text{finalPoly}(X), x_3, \text{finalPolyEval})$.

TODO: Write up protocol components that provide zero-knowledge.

# Lookup argument

Halo 2 uses the following lookup technique, which allows for lookups in arbitrary sets, and is arguably simpler than Plookup.

## Note on Language

In addition to the general notes on language:

- We call the $Z(X)$ polynomial (the grand product argument polynomial for the permutation argument) the "permutation product" column.

## Technique Description

For ease of explanation, we'll first describe a simplified version of the argument that ignores zero knowledge.

We express lookups in terms of a "subset argument" over a table with $2^k$ rows (numbered from 0), and columns $A$ and $S$.

The goal of the subset argument is to enforce that every cell in $A$ is equal to *some* cell in $S$. This means that more than one cell in $A$ can be equal to the *same* cell in $S$, and some cells in $S$ don't need to be equal to any of the cells in $A$.

- $S$ might be fixed, but it doesn't need to be. That is, we can support looking up values in either fixed or variable tables (where the latter includes advice columns).
- $A$ and $S$ can contain duplicates. If the sets represented by $A$ and/or $S$ are not naturally of size $2^k$, we extend $S$ with duplicates and $A$ with dummy values known to be in $S$.
    - Alternatively we could add a "lookup selector" that controls which elements of the $A$ column participate in lookups. This would modify the occurrence of $A(X)$ in the permutation rule below to replace $A$ with, say, $S_0$ if a lookup is not selected.

Let $\ell_i$ be the Lagrange basis polynomial that evaluates to $1$ at row $i$, and $0$ otherwise.

We start by allowing the prover to supply permutation columns of $A$ and $S$. Let's call these $A'$ and $S'$, respectively. We can enforce that they are permutations using a permutation argument with product column $Z$ with the rules:

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

$$\ell_0(X) \cdot (1 - Z(X)) = 0$$

i.e. provided that division by zero does not occur, we have for all $i \in [0, 2^k)$:

$$Z_{i+1} = Z_i \cdot \frac{(A_i + \beta) \cdot (S_i + \gamma)}{(A'_i + \beta) \cdot (S'_i + \gamma)}$$

$$Z_{2^k} = Z_0 = 1.$$

This is a version of the permutation argument which allows $A'$ and $S'$ to be permutations of $A$ and $S$, respectively, but doesn't specify the exact permutations. $\beta$ and $\gamma$ are separate challenges so that we can combine these two permutation arguments into one without worrying that they might interfere with each other.

The goal of these permutations is to allow $A'$ and $S'$ to be arranged by the prover in a particular way:

1. All the cells of column $A'$ are arranged so that like-valued cells are vertically adjacent to each other. This could be done by some kind of sorting algorithm, but all that matters is that like-valued cells are on consecutive rows in column $A'$, and that $A'$ is a permutation of $A$.
2. The first row in a sequence of like values in $A'$ is the row that has the corresponding value in $S'$. Apart from this constraint, $S'$ is any arbitrary permutation of $S$.

Now, we'll enforce that either $A'_i = S'_i$ or that $A'_i = A'_{i-1}$, using the rule

$$(A'(X) - S'(X)) \cdot (A'(X) - A'(\omega^{-1} X)) = 0$$

In addition, we enforce $A'_0 = S'_0$ using the rule

$$\ell_0(X) \cdot (A'(X) - S'(X)) = 0$$

(The $A'(X) - A'(\omega^{-1}X)$ term of the first rule here has no effect at row $0$, even though $\omega^{-1}X$ "wraps", because of the second rule.)

Together these constraints effectively force every element in $A'$ (and thus $A$) to equal at least one element in $S'$ (and thus $S$). Proof: by induction on prefixes of the rows.

## Zero-knowledge adjustment

In order to achieve zero knowledge for the PLONK-based proof system, we will need the last $t$ rows of each column to be filled with random values. This requires an adjustment to the lookup argument, because these random values would not satisfy the constraints described above.

We limit the number of usable rows to $u = 2^k - t - 1$. We add two selectors:

- $q_{blind}$ is set to $1$ on the last $t$ rows, and $0$ elsewhere;
- $q_{last}$ is set to $1$ only on row $u$, and $0$ elsewhere (i.e. it is set on the row in between the usable rows and the blinding rows).

We enable the constraints from above only for the usable rows:

$$\big(1 - (q_{last}(X) + q_{blind}(X))\big) \cdot \big(Z(\omega X) \cdot (A'(X) + \beta) \cdot (S'(X) + \gamma) - Z(X) \cdot (A(X) + \beta$$

$$\big(1 - (q_{last}(X) + q_{blind}(X))\big) \cdot (A'(X) - S'(X)) \cdot (A'(X) - A'(\omega^{-1}X)) = 0$$

The rules that are enabled on row $0$ remain the same:

$$\ell_0(X) \cdot (A'(X) - S'(X)) = 0$$

$$\ell_0(X) \cdot (1 - Z(X)) = 0$$

Since we can no longer rely on the wraparound to ensure that the product $Z$ becomes $1$ again at $\omega^{2^k}$, we would instead need to constrain $Z(\omega^u)$ to $1$. However, there is a potential difficulty: if any of the values $A_i + \beta$ or $S_i + \gamma$ are zero for $i \in [0, u)$, then it might not be possible to satisfy the permutation argument. This occurs with negligible probability over choices of $\beta$ and $\gamma$, but is an obstacle to achieving *perfect* zero knowledge (because an adversary can rule out witnesses that would cause this situation), as well as perfect completeness.

To ensure both perfect completeness and perfect zero knowledge, we allow $Z(\omega^u)$ to be either zero or one:

$$q_{last}(X) \cdot (Z(X)^2 - Z(X)) = 0$$

Now if $A_i + \beta$ or $S_i + \gamma$ are zero for some $i$, we can set $Z_j = 0$ for $i < j \leq u$, satisfying the constraint system.

Note that the challenges $\beta$ and $\gamma$ are chosen after committing to $A$ and $S$ (and to $A'$ and $S'$), so the prover cannot force the case where some $A_i + \beta$ or $S_i + \gamma$ is zero to occur. Since this case occurs with negligible probability, soundness is not affected.

## Cost

- There is the original column $A$ and the fixed column $S$.
- There is a permutation product column $Z$.
- There are the two permutations $A'$ and $S'$.
- The gates are all of low degree.

## Generalizations

Halo 2's lookup argument implementation generalizes the above technique in the following ways:

- $A$ and $S$ can be extended to multiple columns, combined using a random challenge. $A'$ and $S'$ stay as single columns.
    - The commitments to the columns of $S$ can be precomputed, then combined cheaply once the challenge is known by taking advantage of the homomorphic property of Pedersen commitments.
    - The columns of $A$ can be given as arbitrary polynomial expressions using relative references. These will be substituted into the product column constraint, subject to the maximum degree bound. This potentially saves one or more advice columns.
- Then, a lookup argument for an arbitrary-width relation can be implemented in terms of a subset argument, i.e. to constrain $\mathcal{R}(x, y, \ldots)$ in each row, consider $\mathcal{R}$ as a set of tuples $S$ (using the method of the previous point), and check that $(x, y, \ldots) \in \mathcal{R}$.
    - In the case where $\mathcal{R}$ represents a function, this implicitly also checks that the inputs are in the domain. This is typically what we want, and often saves an additional range check.
- We can support multiple tables in the same circuit, by combining them into a single table that includes a tag column to identify the original table.
    - The tag column could be merged with the "lookup selector" mentioned earlier, if this were implemented.

These generalizations are similar to those in sections 4 and 5 of the Plookup paper. That is, the differences from Plookup are in the subset argument. This argument can then be used in all the same ways; for instance, the optimized range check technique in section 5 of the Plookup paper can also be used with this subset argument.

# Permutation argument

Given that gates in halo2 circuits operate "locally" (on cells in the current row or defined relative rows), it is common to need to copy a value from some arbitrary cell into the current row for use in a gate. This is performed with an equality constraint, which enforces that the source and destination cells contain the same value.

We implement these equality constraints by constructing a permutation that represents the constraints, and then using a permutation argument within the proof to enforce them.

## Notation

A permutation is a one-to-one and onto mapping of a set onto itself. A permutation can be factored uniquely into a composition of cycles (up to ordering of cycles, and rotation of each cycle).

We sometimes use cycle notation to write permutations. Let $(a\ b\ c)$ denote a cycle where $a$ maps to $b$, $b$ maps to $c$, and $c$ maps to $a$ (with the obvious generalization to arbitrary-sized cycles). Writing two or more cycles next to each other denotes a composition of the corresponding permutations. For example, $(a\ b)\ (c\ d)$ denotes the permutation that maps $a$ to $b$, $b$ to $a$, $c$ to $d$, and $d$ to $c$.

## Constructing the permutation

### Goal

We want to construct a permutation in which each subset of variables that are in a equality-constraint set form a cycle. For example, suppose that we have a circuit that defines the following equality constraints:

- $a \equiv b$
- $a \equiv c$

- $d \equiv e$

From this we have the equality-constraint sets $\{a, b, c\}$ and $\{d, e\}$. We want to construct the permutation:

$$(a\ b\ c)\ (d\ e)$$

which defines the mapping of $[a, b, c, d, e]$ to $[b, c, a, e, d]$.

## Algorithm

We need to keep track of the set of cycles, which is a set of disjoint sets. Efficient data structures for this problem are known; for the sake of simplicity we choose one that is not asymptotically optimal but is easy to implement.

We represent the current state as:

- an array **mapping** for the permutation itself;
- an auxiliary array **aux** that keeps track of a distinguished element of each cycle;
- another array **sizes** that keeps track of the size of each cycle.

We have the invariant that for each element $x$ in a given cycle $C$, $\mathsf{aux}(x)$ points to the same element $c \in C$. This allows us to quickly decide whether two given elements $x$ and $y$ are in the same cycle, by checking whether $\mathsf{aux}(x) = \mathsf{aux}(y)$. Also, $\mathsf{sizes}(\mathsf{aux}(x))$ gives the size of the cycle containing $x$. (This is guaranteed only for $\mathsf{sizes}(\mathsf{aux}(x))$, not for $\mathsf{sizes}(x)$.)

The algorithm starts with a representation of the identity permutation: for all $x$, we set $\mathsf{mapping}(x) = x$, $\mathsf{aux}(x) = x$, and $\mathsf{sizes}(x) = 1$.

To add an equality constraint $\mathit{left} \equiv \mathit{right}$:

1. Check whether $\mathit{left}$ and $\mathit{right}$ are already in the same cycle, i.e. whether $\mathsf{aux}(\mathit{left}) = \mathsf{aux}(\mathit{right})$. If so, there is nothing to do.
2. Otherwise, $\mathit{left}$ and $\mathit{right}$ belong to different cycles. Make $\mathit{left}$ the larger cycle and $\mathit{right}$ the smaller one, by swapping them iff $\mathsf{sizes}(\mathsf{aux}(\mathit{left})) < \mathsf{sizes}(\mathsf{aux}(\mathit{right}))$.
3. Set $\mathsf{sizes}(\mathsf{aux}(\mathit{left})) := \mathsf{sizes}(\mathsf{aux}(\mathit{left})) + \mathsf{sizes}(\mathsf{aux}(\mathit{right}))$.
4. Following the mapping around the right (smaller) cycle, for each element $x$ set $\mathsf{aux}(x) := \mathsf{aux}(\mathit{left})$.
5. Splice the smaller cycle into the larger one by swapping $\mathsf{mapping}(\mathit{left})$ with $\mathsf{mapping}(\mathit{right})$.

For example, given two disjoint cycles $(A\ B\ C\ D)$ and $(E\ F\ G\ H)$:

```
A +---> B
^       +
|       |
+       v
D <---+ C       E +---> F
                ^       +
                |       |
                +       v
                H <---+ G
```

After adding constraint $B \equiv E$ the above algorithm produces the cycle:

```
A +---> B +-------------+
^                       |
|                       |
+                       v
D <---+ C <---+ E       F
                ^       +
                |       |
                +       v
                H <---+ G
```

### Broken alternatives

If we did not check whether $left$ and $right$ were already in the same cycle, then we could end up undoing an equality constraint. For example, if we have the following constraints:

- $a \equiv b$
- $b \equiv c$
- $c \equiv d$
- $b \equiv d$

and we tried to implement adding an equality constraint just using step 5 of the above algorithm, then we would end up constructing the cycle $(a\ b)\ (c\ d)$, rather than the correct $(a\ b\ c\ d)$.

## Argument specification

We need to check a permutation of cells in $m$ columns, represented in Lagrange basis by polynomials $v_0, \ldots, v_{m-1}$.

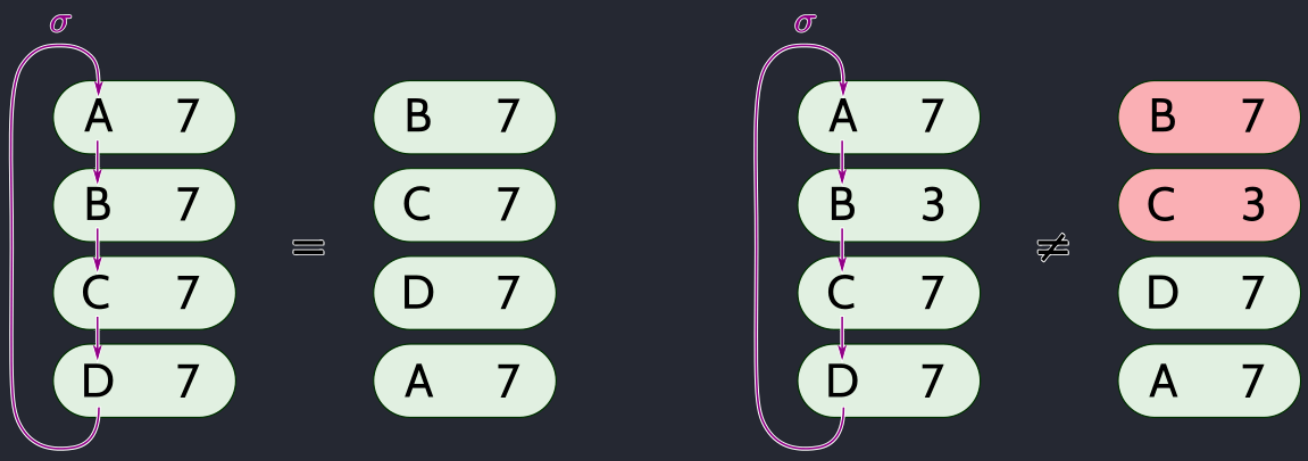
We will label *each cell* in those $m$ columns with a unique element of $\mathbb{F}^\times$.

Suppose that we have a permutation on these labels,

$$\sigma(\mathsf{column} : i, \mathsf{row} : j) = (\mathsf{column} : i', \mathsf{row} : j').$$

in which the cycles correspond to equality-constraint sets.

If we consider the set of pairs $\{(label, value)\}$, then the values within each cycle are equal if and only if permuting the label in each pair by $\sigma$ yields the same set:




Since the labels are distinct, set equality is the same as multiset equality, which we can check using a product argument.

Let $\omega$ be a $2^k$ root of unity and let $\delta$ be a $T$ root of unity, where $T \cdot 2^S + 1 = p$ with $T$ odd and $k \leq S$. We will use $\delta^i \cdot \omega^j \in \mathbb{F}^\times$ as the label for the cell in the $j$th row of the $i$th column of the permutation argument.

We represent $\sigma$ by a vector of $m$ polynomials $s_i(X)$ such that $s_i(\omega^j) = \delta^{i'} \cdot \omega^{j'}$.

Notice that the identity permutation can be represented by the vector of $m$ polynomials $\mathsf{ID}_i(\omega^j)$ such that $\mathsf{ID}_i(\omega^j) = \delta^i \cdot \omega^j$.

We will use a challenge $\beta$ to compress each $(label, value)$ pair to $value + \beta \cdot label$. Just as in the product argument we used for lookups, we also use a challenge $\gamma$ to randomize each term of the product.

Now given our permutation represented by $s_0, \ldots, s_{m-1}$ over columns represented by $v_0, \ldots, v_{m-1}$, we want to ensure that:

$$\prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \left( \frac{v_i(\omega^j) + \beta \cdot \delta^i \cdot \omega^j + \gamma}{v_i(\omega^j) + \beta \cdot s_i(\omega^j) + \gamma} \right) = 1$$

Here $v_i(\omega^j) + \beta \cdot \delta^i \cdot \omega^j$ represents the unpermuted $(label, value)$ pair, and $v_i(\omega^j) + \beta \cdot s_i(\omega^j)$ represents the permuted $(\sigma(label), value)$ pair.

Let $Z_P$ be such that $Z_P(\omega^0) = Z_P(\omega^n) = 1$ and for $0 \leq j < n$:

$$\begin{aligned} Z_P(\omega^{j+1}) &= \prod_{h=0}^{j} \prod_{i=0}^{m-1} \frac{v_i(\omega^h)+\beta\cdot\delta^i\cdot\omega^h+\gamma}{v_i(\omega^h)+\beta\cdot s_i(\omega^h)+\gamma} \\ &= Z_P(\omega^j) \prod_{i=0}^{m-1} \frac{v_i(\omega^j)+\beta\cdot\delta^i\cdot\omega^j+\gamma}{v_i(\omega^j)+\beta\cdot s_i(\omega^j)+\gamma} \end{aligned}$$

Then it is sufficient to enforce the rules:

$$Z_P(\omega X) \cdot \prod_{i=0}^{m-1} (v_i(X) + \beta \cdot s_i(X) + \gamma) - Z_P(X) \cdot \prod_{i=0}^{m-1} \left(v_i(X) + \beta \cdot \delta^i \cdot X + \gamma\right) = 0$$

$$\ell_0 \cdot (1 - Z_P(X)) = 0$$

This assumes that the number of columns $m$ is such that the polynomial in the first rule above fits within the degree bound of the PLONK configuration. We will see below how to handle a larger number of columns.

The optimization used to obtain the simple representation of the identity permutation was suggested by Vitalik Buterin for PLONK, and is described at the end of section 8 of the PLONK paper. Note that the $\delta^i$ are all distinct quadratic non-residues, provided that the number of columns that are enabled for equality is no more than $T$, which always holds in practice for the curves used in Halo 2.

## Zero-knowledge adjustment

Similarly to the lookup argument, we need an adjustment to the above argument to account for the last $t$ rows of each column being filled with random values.

We limit the number of usable rows to $u = 2^k - t - 1$. We add two selectors, defined in the same way as for the lookup argument:

- $q_{blind}$ is set to $1$ on the last $t$ rows, and $0$ elsewhere;
- $q_{last}$ is set to $1$ only on row $u$, and $0$ elsewhere (i.e. it is set on the row in between the usable rows and the blinding rows).

We enable the product rule from above only for the usable rows:

$$\big(1 - (q_{last}(X) + q_{blind}(X))\big) \cdot \left(Z_P(\omega X) \cdot \prod_{i=0}^{m-1} (v_i(X) + \beta \cdot s_i(X) + \gamma) - Z_P(X) \cdot \prod_{i=0}^{m-1} \big(v_i(X) + \beta \cdot \delta^i \cdot X + \gamma\big)\right) = 0$$

The rule that is enabled on row $0$ remains the same:

$$\ell_0(X) \cdot (1 - Z_P(X)) = 0$$

Since we can no longer rely on the wraparound to ensure that each product $Z_P$ becomes $1$ again at $\omega^{2^k}$, we would instead need to constrain $Z(\omega^u) = 1$. This raises the same problem that was described for the lookup argument. So we allow $Z(\omega^u)$ to be either zero or one:

$$q_{last}(X) \cdot (Z_P(X)^2 - Z_P(X)) = 0$$

which gives perfect completeness and zero knowledge.

## Spanning a large number of columns

The halo2 implementation does not in practice limit the number of columns for which equality constraints can be enabled. Therefore, it must solve the problem that the above approach might yield a product rule with a polynomial that exceeds the PLONK configuration's degree bound. The degree bound could be raised, but this would be inefficient if no other rules require a larger degree.

Instead, we split the product across $b$ sets of $m$ columns, using product columns $Z_{P,0}, \ldots Z_{P,b-1}$, and we use another rule to copy the product from the end of one column set to the beginning of the next.

That is, for $0 \leq a < b$ we have:

$$\big(1 - (q_{last}(X) + q_{blind}(X))\big) \cdot \left(Z_{P,a}(\omega X) \cdot \prod_{i=am}^{(a+1)m-1} (v_i(X) + \beta \cdot s_i(X) + \gamma) - Z_P(X) \cdot \prod_{i=am}^{(a+1)m-1} \big(v_i(X) + \beta \cdot \delta^i \cdot X + \right. \\ \left. = 0\right.$$

> For simplicity this is written assuming that the number of columns enabled for equality constraints is a multiple of $m$; if not then the products for the last column set will have

fewer than $m$ terms.

For the first column set we have:

$$\ell_0 \cdot (1 - Z_{P,0}(X)) = 0$$

For each subsequent column set, $0 < a < b$, we use the following rule to copy $Z_{P,a-1}(\omega^u)$ to the start of the next column set, $Z_{P,a}(\omega^0)$:

$$\ell_0 \cdot (Z_{P,a}(X) - Z_{P,a-1}(\omega^u X)) = 0$$

For the last column set, we allow $Z_{P,b-1}(\omega^u)$ to be either zero or one:

$$q_{last}(X) \cdot \left(Z_{P,b-1}(X)^2 - Z_{P,b-1}(X)\right) = 0$$

which gives perfect completeness and zero knowledge as before.

# Circuit commitments

## Committing to the circuit assignments

At the start of proof creation, the prover has a table of cell assignments that it claims satisfy the constraint system. The table has $n = 2^k$ rows, and is broken into advice, instance, and fixed columns. We define $F_{i,j}$ as the assignment in the $j$th row of the $i$th fixed column. Without loss of generality, we'll similarly define $A_{i,j}$ to represent the advice and instance assignments.

> We separate fixed columns here because they are provided by the verifier, whereas the advice and instance columns are provided by the prover. In practice, the commitments to instance and fixed columns are computed by both the prover and verifier, and only the advice commitments are stored in the proof.

To commit to these assignments, we construct Lagrange polynomials of degree $n - 1$ for each column, over an evaluation domain of size $n$ (where $\omega$ is the $n$th primitive root of unity):

- $a_i(X)$ interpolates such that $a_i(\omega^j) = A_{i,j}$.
- $f_i(X)$ interpolates such that $f_i(\omega^j) = F_{i,j}$.

We then create a blinding commitment to the polynomial for each column:

$$\mathbf{A} = [\text{Commit}(a_0(X)), \ldots, \text{Commit}(a_i(X))]$$

$$\mathbf{F} = [\text{Commit}(f_0(X)), \ldots, \text{Commit}(f_i(X))]$$

$\mathbf{F}$ is constructed as part of key generation, using a blinding factor of $1$. $\mathbf{A}$ is constructed by the prover and sent to the verifier.

## Committing to the lookup permutations

The verifier starts by sampling $\theta$, which is used to keep individual columns within lookups independent. Then, the prover commits to the permutations for each lookup as follows:

- Given a lookup with input column polynomials $[A_0(X), \ldots, A_{m-1}(X)]$ and table column polynomials $[S_0(X), \ldots, S_{m-1}(X)]$, the prover constructs two compressed polynomials

$$A_{\text{compressed}}(X) = \theta^{m-1} A_0(X) + \theta^{m-2} A_1(X) + \cdots + \theta A_{m-2}(X) + A_{m-1}(X)$$

$$S_{\text{compressed}}(X) = \theta^{m-1} S_0(X) + \theta^{m-2} S_1(X) + \cdots + \theta S_{m-2}(X) + S_{m-1}(X)$$

- The prover then permutes $A_{\text{compressed}}(X)$ and $S_{\text{compressed}}(X)$ according to the rules of the lookup argument, obtaining $A'(X)$ and $S'(X)$.

The prover creates blinding commitments for all of the lookups

$$\mathbf{L} = [(\text{Commit}(A'(X))), \text{Commit}(S'(X))), \ldots]$$

and sends them to the verifier.

After the verifier receives $\mathbf{A}$, $\mathbf{F}$, and $\mathbf{L}$, it samples challenges $\beta$ and $\gamma$ that will be used in the permutation argument and the remainder of the lookup argument below. (These challenges can be reused because the arguments are independent.)

## Committing to the equality constraint permutation

Let $c$ be the number of columns that are enabled for equality constraints.

Let $m$ be the maximum number of columns that can be accommodated by a column set without exceeding the PLONK configuration's maximum constraint degree.

Let $u$ be the number of "usable" rows as defined in the Permutation argument section.

Let $b = \mathsf{ceiling}(c/m)$.

The prover constructs a vector $\mathbf{P}$ of length $bu$ such that for each column set $0 \leq a < b$ and each row $0 \leq j < u$,

$$\mathbf{P}_{au+j} = \prod_{i=am}^{\min(c,(a+1)m)-1} \frac{v_i(\omega^j) + \beta \cdot \delta^i \cdot \omega^j + \gamma}{v_i(\omega^j) + \beta \cdot s_i(\omega^j) + \gamma}.$$

The prover then computes a running product of $\mathbf{P}$, starting at $1$, and a vector of polynomials $Z_{P,0..b-1}$ that each have a Lagrange basis representation corresponding to a $u$-sized slice of this running product, as described in the Permutation argument section.

The prover creates blinding commitments to each $Z_{P,a}$ polynomial:

$$\mathbf{Z_P} = [\mathrm{Commit}(Z_{P,0}(X)), \ldots, \mathrm{Commit}(Z_{P,b-1}(X))]$$

and sends them to the verifier.

## Committing to the lookup permutation product columns

In addition to committing to the individual permuted lookups, for each lookup, the prover needs to commit to the permutation product column:

- The prover constructs a vector $P$:

$$P_j = \frac{(A_{\text{compressed}}(\omega^j) + \beta)(S_{\text{compressed}}(\omega^j) + \gamma)}{(A'(\omega^j) + \beta)(S'(\omega^j) + \gamma)}$$

- The prover constructs a polynomial $Z_L$ which has a Lagrange basis representation corresponding to a running product of $P$, starting at $Z_L(1) = 1$.

$\beta$ and $\gamma$ are used to combine the permutation arguments for $A'(X)$ and $S'(X)$ while keeping them independent. The important thing here is that the verifier samples $\beta$ and $\gamma$ after the prover has created $\mathbf{A}$, $\mathbf{F}$, and $\mathbf{L}$ (and thus committed to all the cell values used in lookup columns, as well as $A'(X)$ and $S'(X)$ for each lookup).

As before, the prover creates blinding commitments to each $Z_L$ polynomial:

$$\mathbf{Z_L} = [\mathrm{Commit}(Z_L(X)), \ldots]$$

and sends them to the verifier.

# Vanishing argument

Having committed to the circuit assignments, the prover now needs to demonstrate that the various circuit relations are satisfied:

- The custom gates, represented by polynomials $\text{gate}_i(X)$.
- The rules of the lookup arguments.
- The rules of the equality constraint permutations.

Each of these relations is represented as a polynomial of degree $d$ (the maximum degree of any of the relations) with respect to the circuit columns. Given that the degree of the assignment polynomials for each column is $n - 1$, the relation polynomials have degree $d(n - 1)$ with respect to $X$.

In our example, these would be the gate polynomials, of degree $3n - 3$:

- $\text{gate}_0(X) = a_0(X) \cdot a_1(X) \cdot a_2(X\omega^{-1}) - a_3(X)$
- $\text{gate}_1(X) = f_0(X\omega^{-1}) \cdot a_2(X)$
- $\text{gate}_2(X) = f_0(X) \cdot a_3(X) \cdot a_0(X)$

A relation is satisfied if its polynomial is equal to zero. One way to demonstrate this is to divide each polynomial relation by the vanishing polynomial $t(X) = (X^n - 1)$, which is the lowest-degree polynomial that has roots at every $\omega^i$. If relation's polynomial is perfectly divisible by $t(X)$, it is equal to zero over the domain (as desired).

This simple construction would require a polynomial commitment per relation. Instead, we commit to all of the circuit relations simultaneously: the verifier samples $y$, and then the prover constructs the quotient polynomial

$$h(X) = \frac{\text{gate}_0(X) + y \cdot \text{gate}_1(X) + \cdots + y^i \cdot \text{gate}_i(X) + \dots}{t(X)},$$

where the numerator is a random (the prover commits to the cell assignments before the verifier samples $y$) linear combination of the circuit relations.

- If the numerator polynomial (in formal indeterminate $X$) is perfectly divisible by $t(X)$, then with high probability all relations are satisfied.
- Conversely, if at least one relation is not satisfied, then with high probability $h(x) \cdot t(x)$ will not equal the evaluation of the numerator at $x$. In this case, the numerator polynomial would not be perfectly divisible by $t(X)$.

## Committing to $h(X)$

$h(X)$ has degree $d(n-1)-n$ (because the divisor $t(X)$ has degree $n$). However, the polynomial commitment scheme we use for Halo 2 only supports committing to polynomials of degree $n-1$ (which is the maximum degree that the rest of the protocol needs to commit to). Instead of increasing the cost of the polynomial commitment scheme, the prover split $h(X)$ into pieces of degree $n-1$

$$h_0(X) + X^n h_1(X) + \cdots + X^{n(d-1)} h_{d-1}(X),$$

and produces blinding commitments to each piece

$$\mathbf{H} = [\text{Commit}(h_0(X)), \text{Commit}(h_1(X)), \ldots, \text{Commit}(h_{d-1}(X))].$$

## Evaluating the polynomials

At this point, we have committed to all properties of the circuit. The verifier now wants to see if the prover committed to the correct $h(X)$ polynomial. The verifier samples $x$, and the prover produces the purported evaluations of the various polynomials at $x$, for all the relative offsets used in the circuit, as well as $h(X)$.

In our example, this would be:

- $a_0(x)$
- $a_1(x)$
- $a_2(x)$, $a_2(x\omega^{-1})$
- $a_3(x)$
- $f_0(x)$, $f_0(x\omega^{-1})$
- $h_0(x)$, ..., $h_{d-1}(x)$

The verifier checks that these evaluations satisfy the form of $h(X)$:

$$\frac{\text{gate}_0(x) + \cdots + y^i \cdot \text{gate}_i(x) + \ldots}{t(x)} = h_0(x) + \cdots + x^{n(d-1)} h_{d-1}(x)$$

Now content that the evaluations collectively satisfy the gate constraints, the verifier needs to check that the evaluations themselves are consistent with the original circuit commitments, as well as $\mathbf{H}$. To implement this efficiently, we use a multipoint opening argument.

# Multipoint opening argument

Consider the commitments $A, B, C, D$ to polynomials $a(X), b(X), c(X), d(X)$. Let's say that $a$ and $b$ were queried at the point $x$, while $c$ and $d$ were queried at both points $x$ and $\omega x$. (Here, $\omega$ is the primitive root of unity in the multiplicative subgroup over which we constructed the polynomials).

To open these commitments, we could create a polynomial $Q$ for each point that we queried at (corresponding to each relative rotation used in the circuit). But this would not be efficient in the circuit; for example, $c(X)$ would appear in multiple polynomials.

Instead, we can group the commitments by the sets of points at which they were queried:

| $\{x\}$ | $\{x, \omega x\}$ |
|---|---|
| $A$ | $C$ |
| $B$ | $D$ |

For each of these groups, we combine them into a polynomial set, and create a single $Q$ for that set, which we open at each rotation.

## Optimization steps

The multipoint opening optimization takes as input:

- A random $x$ sampled by the verifier, at which we evaluate $a(X), b(X), c(X), d(X)$.
- Evaluations of each polynomial at each point of interest, provided by the prover: $a(x), b(x), c(x), d(x), c(\omega x), d(\omega x)$

These are the outputs of the vanishing argument.

The multipoint opening optimization proceeds as such:

1. Sample random $x_1$, to keep $a, b, c, d$ linearly independent.

2. Accumulate polynomials and their corresponding evaluations according to the point set at which they were queried: `q_polys`:

$$\begin{aligned} q_1(X) &= a(X) + x_1 b(X) \\ q_2(X) &= c(X) + x_1 d(X) \end{aligned}$$

`q_eval_sets`:

```
        [
            [a(x) + x_1 b(x)],
            [
                c(x) + x_1 d(x),
                c(\omega x) + x_1 d(\omega x)
            ]
        ]
```

NB: `q_eval_sets` is a vector of sets of evaluations, where the outer vector corresponds to the point sets, which in this example are $\{x\}$ and $\{x, \omega x\}$, and the inner vector corresponds to the points in each set.

3. Interpolate each set of values in `q_eval_sets`: `r_polys`:

$$\begin{array}{rcl} r_1(X) s.t. & & \\ \quad r_1(x) & = & a(x) + x_1 b(x) \\ r_2(X) s.t. & & \\ \quad r_2(x) & = & c(x) + x_1 d(x) \\ \quad r_2(\omega x) & = & c(\omega x) + x_1 d(\omega x) \end{array}$$

4. Construct `f_polys` which check the correctness of `q_polys`: `f_polys`

$$\begin{array}{rcl} f_1(X) & = & \frac{q_1(X) - r_1(X)}{X - x} \\ f_2(X) & = & \frac{q_2(X) - r_2(X)}{(X - x)(X - \omega x)} \end{array}$$

If $q_1(x) = r_1(x)$, then $f_1(X)$ should be a polynomial. If $q_2(x) = r_2(x)$ and $q_2(\omega x) = r_2(\omega x)$ then $f_2(X)$ should be a polynomial.

5. Sample random $x_2$ to keep the `f_polys` linearly independent.

6. Construct $f(X) = f_1(X) + x_2 f_2(X)$.

7. Sample random $x_3$, at which we evaluate $f(X)$:

$$\begin{array}{rcccl} f(x_3) & = & f_1(x_3) & + & x_2 f_2(x_3) \\ & = & \frac{q_1(x_3) - r_1(x_3)}{x_3 - x} & + & x_2 \frac{q_2(x_3) - r_2(x_3)}{(x_3 - x)(x_3 - \omega x)} \end{array}$$

8. Sample random $x_4$ to keep $f(X)$ and `q_polys` linearly independent.

9. Construct `final_poly`,

$$final\_poly(X) = f(X) + x_4 q_1(X) + x_4^2 q_2(X),$$

which is the polynomial we commit to in the inner product argument.

# Inner product argument

Halo 2 uses a polynomial commitment scheme for which we can create polynomial commitment opening proofs, based around the Inner Product Argument.

> TODO: Explain Halo 2's variant of the IPA.
>
> It is very similar to $\mathrm{PC}_{\mathrm{DL}}.\mathrm{Open}$ from Appendix A.2 of BCMS20. See this comparison for details.

# Comparison to other work

## BCMS20 Appendix A.2

Appendix A.2 of BCMS20 describes a polynomial commitment scheme that is similar to the one described in BGH19 (BCMS20 being a generalization of the original Halo paper). Halo 2 builds on both of these works, and thus itself uses a polynomial commitment scheme that is very similar to the one in BCMS20.

The following table provides a mapping between the variable names in BCMS20, and the equivalent objects in Halo 2 (which builds on the nomenclature from the Halo paper):

| BCMS20 | Halo 2 |
|---|---|
| $S$ | $H$ |
| $H$ | $U$ |
| $C$ | `msm` or $P$ |
| $\alpha$ | $\iota$ |
| $\xi_0$ | $z$ |
| $\xi_i$ | `challenge_i` |
| $H'$ | $[z]U$ |
| $\bar{p}$ | `s_poly` |
| $\bar{\omega}$ | `s_poly_blind` |
| $\bar{C}$ | `s_poly_commitment` |
| $h(X)$ | $g(X)$ |

| BCMS20 | Halo 2 |
|---|---|
| $U$ | $G$ |
| $\omega'$ | `blind` / $\xi$ |
| $\mathbf{c}$ | $\mathbf{a}$ |
| $c$ | $a = \mathbf{a}_0$ |
| $v'$ | $ab$ |

Halo 2's polynomial commitment scheme differs from Appendix A.2 of BCMS20 in two ways:

1. Step 8 of the $\mathrm{Open}$ algorithm computes a "non-hiding" commitment $C'$ prior to the inner product argument, which opens to the same value as $C$ but is a commitment to a randomly-drawn polynomial. The remainder of the protocol involves no blinding. By contrast, in Halo 2 we blind every single commitment that we make (even for instance and fixed polynomials, though using a blinding factor of 1 for the fixed polynomials); this makes the protocol simpler to reason about. As a consequence of this, the verifier needs to handle the cumulative blinding factor at the end of the protocol, and so there is no need to derive an equivalent to $C'$ at the start of the protocol.
    - $C'$ is also an input to the random oracle for $\xi_0$; in Halo 2 we utilize a transcript that has already committed to the equivalent components of $C'$ prior to sampling $z$.
2. The $\mathrm{PC}_{\mathrm{DL}}.\mathrm{SuccinctCheck}$ subroutine (Figure 2 of BCMS20) computes the initial group element $C_0$ by adding $[v]H' = [v\xi_0]H$, which requires two scalar multiplications. Instead, we subtract $[v]G_0$ from the original commitment $P$, so that we're effectively opening the polynomial at the point to the value zero. The computation $[v]G_0$ is more efficient in the context of recursion because $G_0$ is a fixed base (so we can use lookup tables).

# Protocol Description

## Preliminaries

We take $\lambda$ as our security parameter, and unless explicitly noted all algorithms and adversaries are probabilistic (interactive) Turing machines that run in polynomial time in this security parameter. We use $\mathrm{negl}(\lambda)$ to denote a function that is negligible in $\lambda$.

## Cryptographic Groups

We let $\mathbb{G}$ denote a cyclic group of prime order $p$. The identity of a group is written as $\mathcal{O}$. We refer to the scalars of elements in $\mathbb{G}$ as elements in a scalar field $\mathbb{F}$ of size $p$. Group elements are written in capital letters while scalars are written in lowercase or Greek letters. Vectors of scalars or group elements are written in boldface, i.e. $\mathbf{a} \in \mathbb{F}^n$ and $\mathbf{G} \in \mathbb{G}^n$. Group operations are written additively and the multiplication of a group element $G$ by a scalar $a$ is written $[a]G$.

We will often use the notation $\langle \mathbf{a}, \mathbf{b} \rangle$ to describe the inner product of two like-length vectors of scalars $\mathbf{a}, \mathbf{b} \in \mathbb{F}^n$. We also use this notation to represent the linear combination of group elements such as $\langle \mathbf{a}, \mathbf{G} \rangle$ with $\mathbf{a} \in \mathbb{F}^n$, $\mathbf{G} \in \mathbb{G}^n$, computed in practice by a multiscalar multiplication.

We use $\mathbf{0}^n$ to describe a vector of length $n$ that contains only zeroes in $\mathbb{F}$.

**Discrete Log Relation Problem.** The advantage metric

$$\mathsf{Adv}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n}(\mathcal{A}, \lambda) = \Pr\left[\mathsf{G}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n}(\mathcal{A}, \lambda)\right]$$

is defined with respect the following game.

$$\begin{array}{l} \underline{\mathbf{Game}\ \mathsf{G}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n}(\mathcal{A}, \lambda):} \\ \mathbf{G} \leftarrow \mathbb{G}^n_\lambda \\ \mathbf{a} \leftarrow \mathcal{A}(\mathbf{G}) \\ \text{Return } (\langle \mathbf{a}, \mathbf{G} \rangle = \mathcal{O} \wedge \mathbf{a} \neq \mathbf{0}^n) \end{array}$$

Given an $n$-length vector $\mathbf{G} \in \mathbb{G}^n$ of group elements, the *discrete log relation problem* asks for $\mathbf{g} \in \mathbb{F}^n$ such that $\mathbf{g} \neq \mathbf{0}^n$ and yet $\langle \mathbf{g}, \mathbf{G} \rangle = \mathcal{O}$, which we refer to as a *non-trivial* discrete log relation. The hardness of this problem is tightly implied by the hardness of the discrete log problem in the group as shown in Lemma 3 of [JT20]. Formally, we use the game $\mathsf{G}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n}$ defined above to capture this problem.

## Interactive Proofs

*Interactive proofs* are a triple of algorithms $\mathrm{IP} = (\mathrm{Setup}, \mathcal{P}, \mathcal{V})$. The algorithm $\mathrm{Setup}(1^\lambda)$ produces as its output some *public parameters* commonly referred to by **pp**. The prover $\mathcal{P}$ and verifier $\mathcal{V}$ are interactive machines (with access to **pp**) and we denote by $\langle \mathcal{P}(x), \mathcal{V}(y) \rangle$ an algorithm that executes a two-party protocol between them on inputs $x, y$. The output of this protocol, a *transcript* of their interaction, contains all of the messages sent between $\mathcal{P}$ and $\mathcal{V}$. At the end of the protocol, the verifier outputs a decision bit.

## Zero knowledge Arguments of Knowledge

Proofs of knowledge are interactive proofs where the prover aims to convince the verifier that they know a witness $w$ such that $(x, w) \in \mathcal{R}$ for a statement $x$ and polynomial-time decidable relation $\mathcal{R}$. We will work with *arguments* of knowledge which assume computationally-bounded provers.

We will analyze arguments of knowledge through the lens of four security notions.

- **Completeness:** If the prover possesses a valid witness, can they *always* convince the verifier? It is useful to understand this property as it can have implications for the other security notions.
- **Soundness:** Can a cheating prover falsely convince the verifier of the correctness of a statement that is not actually correct? We refer to the probability that a cheating prover can falsely convince the verifier as the *soundness error*.
- **Knowledge soundness:** When the verifier is convinced the statement is correct, does the prover actually possess ("know") a valid witness? We refer to the probability that a cheating prover falsely convinces the verifier of this knowledge as the *knowledge error*.
- **Zero knowledge:** Does the verifier learn anything besides that which can be inferred from the correctness of the statement and the prover's knowledge of a valid witness?

First, we will visit the simple definition of completeness.

**Perfect Completeness.** An interactive argument $(\mathrm{Setup}, \mathcal{P}, \mathcal{V})$ has *perfect completeness* if for all polynomial-time decidable relations $\mathcal{R}$ and for all non-uniform polynomial-time adversaries $\mathcal{A}$

$$Pr\left[(x, w) \notin \mathcal{R} \vee \langle \mathcal{P}(\mathsf{pp}, x, w), \mathcal{V}(\mathsf{pp}, x)\rangle \text{ accepts} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathrm{Setup}(1^\lambda) \\ (x, w) \leftarrow \mathcal{A}(\mathsf{pp}) \end{array}\right] = 1$$

### Soundness

Complicating our analysis is that although our protocol is described as an interactive argument, it is realized in practice as a *non-interactive argument* through the use of the Fiat-Shamir transformation.

**Public coin.** We say that an interactive argument is *public coin* when all of the messages sent by the verifier are each sampled with fresh randomness.

**Fiat-Shamir transformation.** In this transformation an interactive, public coin argument can be made *non-interactive* in the *random oracle model* by replacing the verifier algorithm with a cryptographically strong hash function that produces sufficiently random looking output.

This transformation means that in the concrete protocol a cheating prover can easily "rewind" the verifier by forking the transcript and sending new messages to the verifier. Studying the concrete security of our construction *after* applying this transformation is important. Fortunately, we are able to follow a framework of analysis by Ghoshal and Tessaro ([GT20]) that has been applied to constructions similar to ours.

We will study our protocol through the notion of *state-restoration soundness*. In this model the (cheating) prover is allowed to rewind the verifier to any previous state it was in. The prover wins if they are able to produce an accepting transcript.

**State-Restoration Soundness.** Let $\mathrm{IP}$ be an interactive argument with $r >= r(\lambda)$ verifier challenges and let the $i$th challenge be sampled from $\mathsf{Ch}_i$. The advantage metric

$$\mathsf{Adv}^{\mathrm{SRS}}_{\mathrm{IP}}(\mathcal{P}, \lambda) = \Pr\left[\mathrm{SRS}^{\mathrm{IP}}_{\mathcal{P}}(\lambda)\right]$$

of a state restoration prover $\mathcal{P}$ is defined with respect to the following game.

$\underline{\mathbf{Game}\ \mathrm{SRS}^{\mathcal{P}}_{\mathrm{IP}}(\lambda):}$
$\mathrm{win} \leftarrow \texttt{false};$
$\mathrm{tr} \leftarrow \epsilon$
$\mathsf{pp} \leftarrow \mathrm{IP.Setup}(1^\lambda)$
$(x, \mathsf{st}_{\mathcal{P}}) \leftarrow \mathcal{P}_\lambda(\mathsf{pp})$
$\mathrm{Run}\ \mathcal{P}^{\mathcal{O}_{\mathrm{SRS}}}_{\lambda}(\mathsf{st}_{\mathcal{P}})$
$\mathrm{Return\ win}$

$\underline{\mathbf{Oracle}\ \mathcal{O}_{\mathrm{SRS}}(\tau = (a_1, c_1, ..., a_{i-1}, c_{i-1}), a_i):}$
$\mathrm{If}\ \tau \in \mathrm{tr}\ \mathrm{then}$
$\quad \mathrm{If}\ i \leq r\ \mathrm{then}$
$\quad\quad c_i \leftarrow \mathsf{Ch}_i; \mathrm{tr} \leftarrow \mathrm{tr}||(\tau, a_i, c_i); \mathrm{Return}\ c_i$
$\quad \mathrm{Else\ if}\ i = r + 1\ \mathrm{then}$
$\quad\quad d \leftarrow \mathrm{IP}.\mathcal{V}(\mathsf{pp}, x, (\tau, a_i)); \mathrm{tr} \leftarrow (\tau, a_i)$
$\quad\quad \mathrm{If}\ d = 1\ \mathrm{then\ win} \leftarrow \texttt{true}$
$\quad\quad \mathrm{Return}\ d$
$\mathrm{Return} \perp$

As shown in [GT20] (Theorem 1) state restoration soundness is tightly related to soundness after applying the Fiat-Shamir transformation.

### Knowledge Soundness

We will show that our protocol satisfies a strengthened notion of knowledge soundness known as *witness extended emulation*. Informally, this notion states that for any successful prover

algorithm there exists an efficient *emulator* that can extract a witness from it by rewinding it and supplying it with fresh randomness.

However, we must slightly adjust our definition of witness extended emulation to account for the fact that our provers are state restoration provers and can rewind the verifier. Further, to avoid the need for rewinding the state restoration prover during witness extraction we study our protocol in the algebraic group model.

**Algebraic Group Model (AGM).** An adversary $\mathcal{P}_{\text{alg}}$ is said to be *algebraic* if whenever it outputs a group element $X$ it also outputs a *representation* $\mathbf{x} \in \mathbb{F}^n$ such that $\langle \mathbf{x}, \mathbf{G} \rangle = X$ where $\mathbf{G} \in \mathbb{G}^n$ is the vector of group elements that $\mathcal{P}_{\text{alg}}$ has seen so far. Notationally, we write $\{X\}$ to describe a group element $X$ enhanced with this representation. We also write $\{X\}_i^{\mathbf{G}}$ to identify the component of the representation of $X$ that corresponds with $\mathbf{G}_i$. In other words,

$$X = \sum_{i=0}^{n-1} \left[\{X\}_i^{\mathbf{G}}\right] \mathbf{G}_i$$

The algebraic group model allows us to perform so-called "online" extraction for some protocols: the extractor can obtain the witness from the representations themselves for a single (accepting) transcript.

**State Restoration Witness Extended Emulation** Let $\mathrm{IP}$ be an interactive argument for relation $\mathcal{R}$ with $r = r(\lambda)$ challenges. We define for all non-uniform algebraic provers $\mathcal{P}_{\text{alg}}$, extractors $\mathcal{E}$, and computationally unbounded distinguishers $\mathcal{D}$ the advantage metric

$$\mathsf{Adv}^{\text{sr-wee}}_{\mathrm{IP},\mathcal{R}}(\mathcal{P}_{\text{alg}}, \mathcal{D}, \mathcal{E}, \lambda) = \Pr\left[\text{WEE-real}^{\mathcal{P},\mathcal{D}}_{\mathrm{IP},\mathcal{R}}(\lambda)\right] - \Pr\left[\text{WEE-ideal}^{\mathcal{E},\mathcal{P},\mathcal{D}}_{\mathrm{IP},\mathcal{R}}(\lambda)\right]$$

is defined with the respect to the following games.

$\underline{\textbf{Game}\ \text{WEE-real}^{\mathcal{P}_{\text{alg}},\mathcal{D}}_{\text{IP},\mathcal{R}}(\lambda):}$
$\text{tr} \leftarrow \epsilon$
$\mathsf{pp} \leftarrow \text{IP.Setup}(1^\lambda)$
$(x, \mathsf{st}_{\mathcal{P}}) \leftarrow \mathcal{P}_{\text{alg}}(\mathsf{pp})$
$\text{Run}\ \mathcal{P}_{\text{alg}}{}^{\mathcal{O}_{\text{real}}}(\mathsf{st}_{\mathcal{P}})$
$b \leftarrow \mathcal{D}(\text{tr})$
$\text{Return}\ b = 1$

$\underline{\textbf{Game}\ \text{WEE-ideal}^{\mathcal{E},\mathcal{P}_{\text{alg}},\mathcal{D}}_{\text{IP},\mathcal{R}}(\lambda):}$
$\text{tr} \leftarrow \epsilon$
$\mathsf{pp} \leftarrow \text{IP.Setup}(1^\lambda)$
$(x, \mathsf{st}_{\mathcal{P}}) \leftarrow \mathcal{P}_{\text{alg}}(\mathsf{pp})$
$\mathsf{st}_{\mathcal{E}} \leftarrow (1^\lambda, \mathsf{pp}, x)$
$\text{Run}\ \mathcal{P}_{\text{alg}}{}^{\mathcal{O}_{\text{ideal}}}(\mathsf{st}_{\mathcal{P}})$
$w \leftarrow \mathcal{E}(\mathsf{st}_{\mathcal{E}}, \perp)$
$b \leftarrow \mathcal{D}(\text{tr})$
$\text{Return}\ (b = 1)$
$\wedge(\text{Acc}(\text{tr}) \implies (x, w) \in \mathcal{R})$

$\underline{\textbf{Oracle}\ \mathcal{O}_{\text{real}}(\tau = (a_1, c_1, ..., a_{i-1}, c_{i-1}), a_i)}$
$\text{If}\ \tau \in \text{tr}\ \text{then}$
$\ \text{If}\ i \leq r\ \text{then}$
$\ \ c_i \leftarrow \mathsf{Ch}_i; \text{tr} \leftarrow \text{tr}||(\tau, a_i, c_i); \text{Return}\ c_i$
$\ \text{Else if}\ i = r + 1\ \text{then}$
$\ \ d \leftarrow \text{IP}.\mathcal{V}(\mathsf{pp}, x, (\tau, a_i)); \text{tr} \leftarrow (\tau, a_i)$
$\ \ \text{If}\ d = 1\ \text{then win} \leftarrow \texttt{true}$
$\ \ \text{Return}\ d$
$\ \text{Return}\ \perp$

$\underline{\textbf{Oracle}\ \mathcal{O}_{\text{ideal}}(\tau, a):}$
$\text{If}\ \tau \in \text{tr}\ \text{then}$
$\ (r, \mathsf{st}_{\mathcal{E}}) \leftarrow \mathcal{E}(\mathsf{st}_{\mathcal{E}}, [(\tau, a)])$
$\ \text{tr} \leftarrow \text{tr}||(\tau, a, r)$
$\ \text{Return}\ r$
$\text{Return}\ \perp$

### Zero Knowledge

We say that an argument of knowledge is *zero knowledge* if the verifier also does not learn anything from their interaction besides that which can be learned from the existence of a valid $w$. More formally,

**Perfect Special Honest-Verifier Zero Knowledge.** A public coin interactive argument $(\text{Setup}, \mathcal{P}, \mathcal{V})$ has *perfect special honest-verifier zero knowledge* (PSHVZK) if for all polynomial-time decidable relations $\mathcal{R}$ and for all $(x, w) \in \mathcal{R}$ and for all non-uniform polynomial-time adversaries $\mathcal{A}_1, \mathcal{A}_2$ there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that

$$Pr\left[\mathcal{A}_1(\sigma, x, \text{tr}) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \text{Setup}(1^\lambda); \\ (x, w, \rho) \leftarrow \mathcal{A}_2(\mathsf{pp}); \\ tr \leftarrow \langle \mathcal{P}(\mathsf{pp}, x, w), \mathcal{V}(\mathsf{pp}, x, \rho) \rangle \end{array}\right]$$

$$= \quad Pr\left[\mathcal{A}_1(\sigma, x, \text{tr}) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \text{Setup}(1^\lambda); \\ (x, w, \rho) \leftarrow \mathcal{A}_2(\mathsf{pp}); \\ tr \leftarrow \mathcal{S}(\mathsf{pp}, x, \rho) \end{array}\right]$$

where $\rho$ is the internal randomness of the verifier.

In this (common) definition of zero-knowledge the verifier is expected to act "honestly" and send challenges that correspond only with their internal randomness; they cannot adaptively respond to the prover based on the prover's messages. We use a strengthened form of this definition that forces the simulator to output a transcript with the same (adversarially provided) challenges that the verifier algorithm sends to the prover.

## Protocol

Let $\omega \in \mathbb{F}$ be a $n = 2^k$ primitive root of unity forming the domain $D = (\omega^0, \omega^1, ..., \omega^{n-1})$ with $t(X) = X^n - 1$ the vanishing polynomial over this domain. Let $n_g, n_a, n_e$ be positive integers with $n_a, n_e < n$ and $n_g \geq 4$. We present an interactive argument $\mathsf{Halo} = (\mathrm{Setup}, \mathcal{P}, \mathcal{V})$ for the relation

$$\mathcal{R} = \left\{ \begin{array}{l} \left( \begin{array}{l} (g(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-1}\left(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-2}(X)\right)\right) \\ (a_0(X), a_1(X, C_0, a_0(X)), ..., a_{n_a-1}\left(X, C_0, ..., C_{n_a-1}, a_0(X), ..., a_{n_a-2}(X)\right)\right) \end{array} \right. \\ \\ g(\omega^i, \cdots) = 0 \quad \forall i \in [0, 2^k) \end{array} \right.$$

where $a_0, a_1, ..., a_{n_a-1}$ are (multivariate) polynomials with degree $n-1$ in $X$ and $g$ has degree $n_g(n-1)$ at most in any indeterminates $X, C_0, C_1, ....$

$\mathrm{Setup}(\lambda)$ returns $\mathsf{pp} = (\mathbb{G}, \mathbb{F}, \mathbf{G} \in \mathbb{G}^n, U, W \in \mathbb{G})$.

For all $i \in [0, n_a)$:

- Let $\mathbf{p_i}$ be the exhaustive set of integers $j$ (modulo $n$) such that $a_i(\omega^j X, \cdots)$ appears as a term in $g(X, \cdots)$.
- Let $\mathbf{q}$ be a list of distinct sets of integers containing $\mathbf{p_i}$ and the set $\mathbf{q_0} = \{0\}$.
- Let $\sigma(i) = \mathbf{q}_j$ when $\mathbf{q}_j = \mathbf{p_i}$.

Let $n_q \leq n_a$ denote the size of $\mathbf{q}$, and let $n_e$ denote the size of every $\mathbf{p_i}$ without loss of generality.

In the following protocol, we take it for granted that each polynomial $a_i(X, \cdots)$ is defined such that $n_e + 1$ blinding factors are freshly sampled by the prover and are each present as an evaluation of $a_i(X, \cdots)$ over the domain $D$. In all of the following, the verifier's challenges cannot be zero or an element in $D$, and some additional limitations are placed on specific challenges as well.

1. $\mathcal{P}$ and $\mathcal{V}$ proceed in the following $n_a$ rounds of interaction, where in round $j$ (starting at $0$ )

- $\mathcal{P}$ sets $a'_j(X) = a_j(X, c_0, c_1, ..., c_{j-1}, a_0(X, \cdots), ..., a_{j-1}(X, \cdots, c_{j-1}))$
- $\mathcal{P}$ sends a hiding commitment $A_j = \langle \mathbf{a'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{a'}$ are the coefficients of the univariate polynomial $a'_j(X)$ and $\cdot$ is some random, independently sampled blinding factor elided for exposition. (This elision notation is used throughout this protocol description to simplify exposition.)
- $\mathcal{V}$ responds with a challenge $c_j$.

2. $\mathcal{P}$ sets $g'(X) = g(X, c_0, c_1, ..., c_{n_a - 1}, \cdots)$.
3. $\mathcal{P}$ sends a commitment $R = \langle \mathbf{r}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{r} \in \mathbb{F}^n$ are the coefficients of a randomly sampled univariate polynomial $r(X)$ of degree $n - 1$.
4. $\mathcal{P}$ computes univariate polynomial $h(X) = \frac{g'(X)}{t(X)}$ of degree $n_g(n-1) - n$.
5. $\mathcal{P}$ computes at most $n - 1$ degree polynomials $h_0(X), h_1(X), ..., h_{n_g - 2}(X)$ such that $h(X) = \sum_{i=0}^{n_g - 2} X^{ni} h_i(X)$.
6. $\mathcal{P}$ sends commitments $H_i = \langle \mathbf{h_i}, \mathbf{G} \rangle + [\cdot]W$ for all $i$ where $\mathbf{h_i}$ denotes the vector of coefficients for $h_i(X)$.
7. $\mathcal{V}$ responds with challenge $x$ and computes $H' = \sum_{i=0}^{n_g - 2} [x^{ni}] H_i$.
8. $\mathcal{P}$ sets $h'(X) = \sum_{i=0}^{n_g - 2} x^{ni} h_i(X)$.
9. $\mathcal{P}$ sends $r = r(x)$ and for all $i \in [0, n_a)$ sends $\mathbf{a_i}$ such that $(\mathbf{a_i})_j = a'_i(\omega^{(\mathbf{p_i})_j} x)$ for all $j \in [0, n_e - 1]$.
10. For all $i \in [0, n_a)$ $\mathcal{P}$ and $\mathcal{V}$ set $s_i(X)$ to be the lowest degree univariate polynomial defined such that $s_i(\omega^{(\mathbf{p_i})_j} x) = (\mathbf{a_i})_j$ for all $j \in [0, n_e - 1)$.
11. $\mathcal{V}$ responds with challenges $x_1, x_2$ and initializes $Q_0, Q_1, ..., Q_{n_q - 1} = \mathcal{O}$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{V}$ sets $Q_{\sigma(i)} := [x_1] Q_{\sigma(i)} + A_i$.
- $\mathcal{V}$ finally sets $Q_0 := [x_1^2] Q_0 + [x_1] H' + R$.

12. $\mathcal{P}$ initializes $q_0(X), q_1(X), ..., q_{n_q - 1}(X) = 0$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ sets $q_{\sigma(i)} := x_1 q_{\sigma(i)} + a'(X)$.
- $\mathcal{P}$ finally sets $q_0(X) := x_1^2 q_0(X) + x_1 h'(X) + r(X)$.

13. $\mathcal{P}$ and $\mathcal{V}$ initialize $r_0(X), r_1(X), ..., r_{n_q - 1}(X) = 0$.

- Starting at $i = 0$ and ending at $n_a - 1$ $\mathcal{P}$ and $\mathcal{V}$ set $r_{\sigma(i)}(X) := x_1 r_{\sigma(i)}(X) + s_i(X)$.
- Finally $\mathcal{P}$ and $\mathcal{V}$ set $r_0 := x_1^2 r_0 + x_1 h + r$ and where $h$ is computed by $\mathcal{V}$ as $\frac{g'(x)}{t(x)}$ using the values $r, \mathbf{a}$ provided by $\mathcal{P}$.

14. $\mathcal{P}$ sends $Q' = \langle \mathbf{q'}, \mathbf{G} \rangle + [\cdot]W$ where $\mathbf{q'}$ defines the coefficients of the polynomial

$$q'(X) = \sum_{i=0}^{n_q-1} x_2^i \left( \frac{q_i(X) - r_i(X)}{\prod_{j=0}^{n_e-1} \left( X - \omega^{(\mathbf{q_i})_j} x \right)} \right)$$

15. $\mathcal{V}$ responds with challenge $x_3$.
16. $\mathcal{P}$ sends $\mathbf{u} \in \mathbb{F}^{n_q}$ such that $\mathbf{u}_i = q_i(x_3)$ for all $i \in [0, n_q)$.
17. $\mathcal{V}$ responds with challenge $x_4$.
18. $\mathcal{V}$ sets $P = Q' + x_4 \sum_{i=0}^{n_q-1} [x_4^i] Q_i$ and $v =$

$$\sum_{i=0}^{n_q-1} \left( x_2^i \left( \frac{\mathbf{u}_i - r_i(x_3)}{\prod_{j=0}^{n_e-1} \left( x_3 - \omega^{(\mathbf{q_i})_j} x \right)} \right) \right) + x_4 \sum_{i=0}^{n_q-1} x_4 \mathbf{u}_i$$

19. $\mathcal{P}$ sets $p(X) = q'(X) + [x_4] \sum_{i=0}^{n_q-1} x_4^i q_i(X)$.
20. $\mathcal{P}$ samples a random polynomial $s(X)$ of degree $n - 1$ with a root at $x_3$ and sends a commitment $S = \langle \mathbf{s}, \mathbf{G} \rangle + [\cdot] W$ where $\mathbf{s}$ defines the coefficients of $s(X)$.
21. $\mathcal{V}$ responds with challenges $\xi, z$.
22. $\mathcal{V}$ sets $P' = P - [v] \mathbf{G}_0 + [\xi] S$.
23. $\mathcal{P}$ sets $p'(X) = p(X) - p(x_3) + \xi s(X)$ (where $p(x_3)$ should correspond with the verifier's computed value $v$).
24. Initialize $\mathbf{p'}$ as the coefficients of $p'(X)$ and $\mathbf{G'} = \mathbf{G}$ and $\mathbf{b} = (x_3^0, x_3^1, ..., x_3^{n-1})$. $\mathcal{P}$ and $\mathcal{V}$ will interact in the following $k$ rounds, where in the $j$th round starting in round $j = 0$ and ending in round $j = k - 1$:

- $\mathcal{P}$ sends $L_j = \langle \mathbf{p'}_{\text{hi}}, \mathbf{G'}_{\text{lo}} \rangle + [z \langle \mathbf{p'}_{\text{hi}}, \mathbf{b}_{\text{lo}} \rangle] U + [\cdot] W$ and $R_j = \langle \mathbf{p'}_{\text{lo}}, \mathbf{G'}_{\text{hi}} \rangle + [z \langle \mathbf{p'}_{\text{lo}}, \mathbf{b}_{\text{hi}} \rangle] U + [\cdot] W$.
- $\mathcal{V}$ responds with challenge $u_j$ chosen such that $1 + u_{k-1-j} x_3^{2^j}$ is nonzero.
- $\mathcal{P}$ and $\mathcal{V}$ set $\mathbf{G'} := \mathbf{G'}_{\text{lo}} + u_j \mathbf{G'}_{\text{hi}}$ and $\mathbf{b} := \mathbf{b}_{\text{lo}} + u_j \mathbf{b}_{\text{hi}}$.
- $\mathcal{P}$ sets $\mathbf{p'} := \mathbf{p'}_{\text{lo}} + u_j^{-1} \mathbf{p'}_{\text{hi}}$.

25. $\mathcal{P}$ sends $c = \mathbf{p'}_0$ and synthetic blinding factor $f$ computed from the elided blinding factors.
26. $\mathcal{V}$ accepts only if $\sum_{j=0}^{k-1} [u_j^{-1}] L_j + P' + \sum_{j=0}^{k-1} [u_j] R_j = [c] \mathbf{G'}_0 + [c \mathbf{b}_0 z] U + [f] W$.

## Zero-knowledge and Completeness

We claim that this protocol is *perfectly complete*. This can be verified by inspection of the protocol; given a valid witness $a_i(X, \cdots) \forall i$ the prover succeeds in convincing the verifier with probability $1$.

We claim that this protocol is *perfect special honest-verifier zero knowledge*. We do this by showing that a simulator $\mathcal{S}$ exists which can produce an accepting transcript that is equally distributed with a valid prover's interaction with a verifier with the same public coins. The simulator will act as an honest prover would, with the following exceptions:

1. In step $1$ of the protocol $\mathcal{S}$ chooses random degree $n-1$ polynomials (in $X$) $a_i(X, \cdots) \forall i$.
2. In step $5$ of the protocol $\mathcal{S}$ chooses a random $n-1$ degree polynomials $h_0(X), h_1(X), ..., h_{n_g-2}(X)$.
3. In step $14$ of the protocol $\mathcal{S}$ chooses a random $n-1$ degree polynomial $q'(X)$.
4. In step $20$ of the protocol $\mathcal{S}$ uses its foreknowledge of the verifier's choice of $\xi$ to produce a degree $n-1$ polynomial $s(X)$ conditioned only such that $p(X) - v + \xi s(X)$ has a root at $x_3$.

First, let us consider why this simulator always succeeds in producing an *accepting* transcript. $\mathcal{S}$ lacks a valid witness and simply commits to random polynomials whenever knowledge of a valid witness would be required by the honest prover. The verifier places no conditions on the scalar values in the transcript. $\mathcal{S}$ must only guarantee that the check in step $26$ of the protocol succeeds. It does so by using its knowledge of the challenge $\xi$ to produce a polynomial which interferes with $p'(X)$ to ensure it has a root at $x_3$. The transcript will thus always be accepting due to perfect completeness.

In order to see why $\mathcal{S}$ produces transcripts distributed identically to the honest prover, we will look at each piece of the transcript and compare the distributions. First, note that $\mathcal{S}$ (just as the honest prover) uses a freshly random blinding factor for every group element in the transcript, and so we need only consider the *scalars* in the transcript. $\mathcal{S}$ acts just as the prover does except in the mentioned cases so we will analyze each case:

1. $\mathcal{S}$ and an honest prover reveal $n_e$ openings of each polynomial $a_i(X, \cdots)$, and at most one additional opening of each $a_i(X, \cdots)$ in step $16$. However, the honest prover blinds their polynomials $a_i(X, \cdots)$ (in $X$) with $n_e + 1$ random evaluations over the domain $D$. Thus, the openings of $a_i(X, \cdots)$ at the challenge $x$ (which is prohibited from being $0$ or in the domain $D$ by the protocol) are distributed identically between $\mathcal{S}$ and an honest prover.
2. Neither $\mathcal{S}$ nor the honest prover reveal $h(x)$ as it is computed by the verifier. However, the honest prover may reveal $h'(x_3)$ --- which has a non-trivial relationship with $h(X)$ --- were it not for the fact that the honest prover also commits to a random degree $n-1$

polynomial $r(X)$ in step $3$, producing a commitment $R$ and ensuring that in step $12$ when the prover sets $q_0(X) := x_1^2 q_0(X) + x_1 h'(X) + r(X)$ the distribution of $q_0(x_3)$ is uniformly random. Thus, $h'(x_3)$ is never revealed by the honest prover nor by $\mathcal{S}$.
3. The expected value of $q'(x_3)$ is computed by the verifier (in step $18$) and so the simulator's actual choice of $q'(X)$ is irrelevant.
4. $p(X) - v + \xi s(X)$ is conditioned on having a root at $x_3$, but otherwise no conditions are placed on $s(X)$ and so the distribution of the degree $n - 1$ polynomial $p(X) - v + \xi s(X)$ is uniformly random whether or not $s(X)$ has a root at $x_3$. Thus, the distribution of $c$ produced in step $25$ is identical between $\mathcal{S}$ and an honest prover. The synthetic blinding factor $f$ also revealed in step $25$ is a trivial function of the prover's other blinding factors and so is distributed identically between $\mathcal{S}$ and an honest prover.

Notes:

1. In an earlier version of our protocol, the prover would open each individual commitment $H_0, H_1, \ldots$ at $x$ as part of the multipoint opening argument, and the verifier would confirm that a linear combination of these openings (with powers of $x^n$) agreed to the expected value of $h(x)$. This was done because it's more efficient in recursive proofs. However, it was unclear to us what the expected distribution of the openings of these commitments $H_0, H_1, \ldots$ was and so proving that the argument was zero-knowledge is difficult. Instead, we changed the argument so that the *verifier* computes a linear combination of the commitments and that linear combination is opened at $x$. This avoided leaking $h_i(x)$.
2. As mentioned, in step $3$ the prover commits to a random polynomial as a way of ensuring that $h'(x_3)$ is not revealed in the multiopen argument. This is done because it's unclear what the distribution of $h'(x_3)$ would be.
3. Technically it's also possible for us to prove zero-knowledge with a simulator that uses its foreknowledge of the challenge $x$ to commit to an $h(X)$ which agrees at $x$ to the value it will be expected to. This would obviate the need for the random polynomial $s(X)$ in the protocol. This may make the analysis of zero-knowledge for the remainder of the protocol a little bit tricky though, so we didn't go this route.
4. Group element blinding factors are *technically* not necessary after step $23$ in which the polynomial is completely randomized. However, it's simpler in practice for us to ensure that every group element in the protocol is randomly blinded to make edge cases involving the point at infinity harder.
5. It is crucial that the verifier cannot challenge the prover to open polynomials at points in $D$ as otherwise the transcript of an honest prover will be forced to contain what could be portions of the prover's witness. We therefore restrict the space of challenges to include all elements of the field except $D$ and, for simplicity, we also prohibit the challenge of $0$.

## Witness-extended Emulation

Let $\mathsf{Halo} = \mathsf{Halo}[\mathbb{G}]$ be the interactive argument described above for relation $\mathcal{R}$ and some group $\mathbb{G}$ with scalar field $\mathbb{F}$. We can always construct an extractor $\mathcal{E}$ such that for any non-uniform algebraic prover $\mathcal{P}_{\text{alg}}$ making at most $q$ queries to its oracle, there exists a non-uniform adversary $\mathcal{H}$ with the property that for any computationally unbounded distinguisher $\mathcal{D}$

$$\mathsf{Adv}^{\text{sr-wee}}_{\mathsf{Halo},\mathcal{R}}(\mathcal{P}_{\text{alg}}, \mathcal{D}, \mathcal{E}, \lambda) \leq q\epsilon + \mathsf{Adv}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n+2}(\mathcal{H}, \lambda)$$

where $\epsilon \leq \frac{n_g \cdot (n-1)}{|\mathsf{Ch}|}$.

*Proof.* We will prove this by invoking Theorem 1 of [GT20]. First, we note that the challenge space for all rounds is the same, i.e. $\forall i\ \mathsf{Ch} = \mathsf{Ch}_i$. Theorem 1 requires us to define:

- $\mathsf{BadCh}(\text{tr}') \in \mathsf{Ch}$ for all partial transcripts $\text{tr}' = (\mathsf{pp}, x, [a_0], c_0, \ldots, [a_i])$ such that $|\mathsf{BadCh}(\text{tr}')|/|\mathsf{Ch}| \leq \epsilon$.
- an extractor function $e$ that takes as input an accepting extended transcript $\text{tr}$ and either returns a valid witness or fails.
- a function $p_{\text{fail}}(\mathsf{Halo}, \mathcal{P}_{\text{alg}}, e, \mathcal{R})$ returning a probability.

We say that an accepting extended transcript $\text{tr}$ contains "bad challenges" if and only if there exists a partial extended transcript $\text{tr}'$, a challenge $c_i \in \mathsf{BadCh}(\text{tr}')$, and some sequence of prover messages and challenges $([a_{i+1}], c_{i+1}, \ldots [a_j])$ such that $\text{tr} = \text{tr}' \,||\, (c_i, [a_{i+1}], c_{i+1}, \ldots [a_j])$.

Theorem 1 requires that $e$, when given an accepting extended transcript $\text{tr}$ that does not contain "bad challenges", returns a valid witness for that transcript except with probability bounded above by $p_{\text{fail}}(\mathsf{Halo}, \mathcal{P}_{\text{alg}}, e, \mathcal{R})$.

Our strategy is as follows: we will define $e$, establish an upper bound on $p_{\text{fail}}$ with respect to an adversary $\mathcal{H}$ that plays the $\mathsf{dl\text{-}rel}_{\mathbb{G},n+2}$ game, substitute these into Theorem 1, and then walk through the protocol to determine the upper bound of the size of $\mathsf{BadCh}(\text{tr}')$. The adversary $\mathcal{H}$ plays the $\mathsf{dl\text{-}rel}_{\mathbb{G},n+2}$ game as follows: given the inputs $U, W \in \mathbb{G}$, $\mathbf{G} \in \mathbb{G}^n$, the adversary $\mathcal{H}$ simulates the game $\text{sr-wee}_{\mathsf{Halo},\mathcal{R}}$ to $\mathcal{P}_{\text{alg}}$ using the inputs from the $\mathsf{dl\text{-}rel}_{\mathbb{G},n+2}$ game as public parameters. If $\mathcal{P}_{\text{alg}}$ manages to produce an accepting extended transcript $\text{tr}$, $\mathcal{H}$ invokes a function $h$ on $\text{tr}$ and returns its output. We shall define $h$ in such a way that for an accepting extended transcript $\text{tr}$ that does not contain "bad challenges", $e(\text{tr})$ *always* returns a valid witness whenever $h(\text{tr})$ does *not* return a non-trivial discrete log relation. This means that the probability $p_{\text{fail}}(\mathsf{Halo}, \mathcal{P}_{\text{alg}}, e, \mathcal{R})$ is no greater than $\mathsf{Adv}^{\mathsf{dl\text{-}rel}}_{\mathbb{G},n+2}(\mathcal{H}, \lambda)$, establishing our claim.

### Helpful substitutions

We will perform some substitutions to aid in exposition. First, let us define the polynomial

$$\kappa(X) = \prod_{j=0}^{k-1}(1 + u_{k-1-j}X^{2^j})$$

so that we can write $\mathbf{b}_0 = \kappa(x_3)$. The coefficient vector $\mathbf{s}$ of $\kappa(X)$ is defined such that

$$\mathbf{s}_i = \prod_{j=0}^{k-1} u_{k-1-j}^{f(i,j)}$$

where $f(i, j)$ returns $1$ when the $j$th bit of $i$ is set, and $0$ otherwise. We can also write $\mathbf{G'}_0 = \langle \mathbf{s}, \mathbf{G} \rangle$.

## Description of function $h$

Recall that an accepting transcript $\mathbf{tr}$ is such that

$$\sum_{i=0}^{k-1}[u_j^{-1}]\{L_j\} + \{P'\} + \sum_{i=0}^{k-1}[u_j]\{R_j\} = [c]\mathbf{G'}_0 + [cz\mathbf{b}_0]U + [f]W$$

By inspection of the representations of group elements with respect to $\mathbf{G}, U, W$ (recall that $\mathcal{P}_{\text{alg}}$ is algebraic and so $\mathcal{H}$ has them), we obtain the $n$ equalities

$$\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}_i^{\mathbf{G}} + \{P'\}_i^{\mathbf{G}} + \sum_{i=0}^{k-1} u_j\{R_j\}_i^{\mathbf{G}} = c\mathbf{s}_i \forall i \in [0, n)$$

and the equalities

$$\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}^{U} + \{P'\}^{U} + \sum_{i=0}^{k-1} u_j\{R_j\}^{U} = cz\kappa(x_3)$$

$$\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}^{W} + \{P'\}^{W} + \sum_{i=0}^{k-1} u_j\{R_j\}^{W} = f$$

We define the linear-time function $h$ that returns the representation of

$$\begin{aligned}
\sum_{i=0}^{n-1} & \left[\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}_i^{\mathbf{G}} + \{P'\}_i^{\mathbf{G}} + \sum_{i=0}^{k-1} u_j\{R_j\}_i^{\mathbf{G}} - c\mathbf{s}_i\right] & \mathbf{G}_i \\
+ & \left[\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}^{U} + \{P'\}^{U} + \sum_{i=0}^{k-1} u_j\{R_j\}^{U} - cz\kappa(x_3)\right] & U \\
+ & \left[\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}^{W} + \{P'\}^{W} + \sum_{i=0}^{k-1} u_j\{R_j\}^{W} - f\right] & W
\end{aligned}$$

which is always a discrete log relation. If any of the equalities above are not satisfied, then this discrete log relation is non-trivial. This is the function invoked by $\mathcal{H}$.

#### The extractor function $e$

The extractor function $e$ simply returns $a_i(X)$ from the representation $\{A_i\}$ for $i \in [0, n_a)$. Due to the restrictions we will place on the space of bad challenges in each round, we are guaranteed to obtain polynomials such that $g(X, C_0, C_1, \cdots, a_0(X), a_1(X), \cdots)$ vanishes over $D$ whenever the discrete log relation returned by the adversary's function $h$ is trivial. This trivially gives us that the extractor function $e$ succeeds with probability bounded above by $p_{\text{fail}}$ as required.

#### Defining $\mathsf{BadCh}(\text{tr}')$

Recall from before that the following $n$ equalities hold:

$$\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}_i^{\mathbf{G}} + \{P'\}_i^{\mathbf{G}} + \sum_{i=0}^{k-1} u_j\{R_j\}_i^{\mathbf{G}} = c\mathbf{s}_i \forall i \in [0, n)$$

as well as the equality

$$\sum_{i=0}^{k-1} u_j^{-1}\{L_j\}^{U} + \{P'\}^{U} + \sum_{i=0}^{k-1} u_j\{R_j\}^{U} = cz\kappa(x_3)$$

For convenience let us introduce the following notation

$$\begin{aligned}
\mathcal{M}_i^{\mathbf{G}}(m) &= \sum_{i=0}^{m-1} u_j^{-1}\{L_j\}_i^{\mathbf{G}} + \{P'\}_i^{\mathbf{G}} + \sum_{i=0}^{m-1} u_j\{R_j\}_i^{\mathbf{G}} \\
\mathcal{M}^{U}(m) &= \sum_{i=0}^{m-1} u_j^{-1}\{L_j\}^{U} + \{P'\}^{U} + \sum_{i=0}^{m-1} u_j\{R_j\}^{U}
\end{aligned}$$

so that we can rewrite the above (after expanding for $\kappa(x_3)$) as

$$\mathcal{M}_i^{\mathbf{G}}(k) = c\mathbf{s}_i \forall i \in [0, n)$$

$$\mathcal{M}^{U}(k) = cz\prod_{j=0}^{k-1}(1 + u_{k-1-j}x_3^{2^j})$$

We can combine these equations by multiplying both sides of each instance of the first equation by $\mathbf{s}_i^{-1}$ (because $\mathbf{s}_i$ is never zero) and substituting for $c$ in the second equation, yielding the following $n$ equalities:

$$\mathcal{M}^U(k) = \mathcal{M}_i^{\mathbf{G}}(k) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{k-1}(1 + u_{k-1-j}x_3^{2^j}) \forall i \in [0, n)$$

**Lemma 1.** If $\mathcal{M}^U(k) = \mathcal{M}_i^{\mathbf{G}}(k) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{k-1}(1 + u_{k-1-j}x_3^{2^j}) \forall i \in [0, n)$ then it follows that $\{P'\}^U = z \sum_{i=0}^{2^k-1} x_3^i \{P'\}_i^{\mathbf{G}}$ for all transcripts that do not contain bad challenges.

*Proof.* It will be useful to introduce yet another abstraction defined starting with

$$\mathcal{Z}_k(m, i) = \mathcal{M}_i^{\mathbf{G}}(m)$$

and then recursively defined for all integers $r$ such that $0 < r \leq k$

$$\mathcal{Z}_{k-r}(m, i) = \mathcal{Z}_{k-r+1}(m, i) + x_3^{2^{k-r}} \mathcal{Z}_{k-r+1}(m, i + 2^{k-r})$$

This allows us to rewrite our above equalities as

$$\mathcal{M}^U(k) = \mathcal{Z}_k(k, i) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{k-1}(1 + u_{k-1-j}x_3^{2^j}) \forall i \in [0, n)$$

We will now show that for all integers $r$ such that $0 < r \leq k$ that whenever the following holds for $r$

$$\mathcal{M}^U(r) = \mathcal{Z}_r(r, i) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-1}(1 + u_{k-1-j}x_3^{2^j}) \forall i \in [0, 2^r)$$

that the same *also* holds for

$$\mathcal{M}^U(r-1) = \mathcal{Z}_{r-1}(r-1, i) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-2-j}x_3^{2^j}) \forall i \in [0, 2^{r-1})$$

For all integers $r$ such that $0 < r \leq k$ we have that $\mathbf{s}_{i+2^{r-1}} = u_{r-1}\mathbf{s}_i \forall i \in [0, 2^{r-1})$ by the definition of $\mathbf{s}$. This gives us $\mathbf{s}_{i+2^{r-1}}^{-1} = \mathbf{s}_i^{-1} u_{r-1}^{-1} \forall i \in [0, 2^{r-1})$ as no value in $\mathbf{s}$ nor any challenge $u_r$ are zeroes. We can use this to relate one half of the equalities with the other half as so:

$$\begin{aligned}\mathcal{M}^U(r) &= \mathcal{Z}_r(r,i) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-1}(1 + u_{k-1-j}x_3^{2^j}) \\ &= \mathcal{Z}_r(r,i+2^{r-1}) \cdot \mathbf{s}_i^{-1} u_{r-1}^{-1} z \prod_{j=0}^{r-1}(1 + u_{k-1-j}x_3^{2^j}) \\ &\forall i \in [0, 2^{r-1})\end{aligned}$$

Notice that $\mathcal{Z}_r(r,i)$ can be rewritten as $u_{r-1}^{-1}\{L_{r-1}\}_i^{\mathbf{G}} + \mathcal{Z}_r(r-1,i) + u_{r-1}\{R_{r-1}\}_i^{\mathbf{G}}$ for all $i \in [0, 2^r)$. Thus we can rewrite the above as

$$\begin{aligned}\mathcal{M}^U(r) &= \left(u_{r-1}^{-1}\{L_{r-1}\}_i^{\mathbf{G}} + \mathcal{Z}_r(r-1,i) + u_{r-1}\{R_{r-1}\}_i^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-1}(1 + u_{k-1-j}x_3^{2^j}) \\ &= \left(u_{r-1}^{-1}\{L_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}} + \mathcal{Z}_r(r-1,i+2^{r-1}) + u_{r-1}\{R_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} u_{r-1}^{-1} z \prod_{j=0}^{r-1}(1 + u_{k-1-j}x_3^{2^j}) \\ &\forall i \in [0, 2^{r-1})\end{aligned}$$

Now let us rewrite these equalities substituting $u_{r-1}$ with formal indeterminate $X$.

$$\begin{aligned}&X^{-1}\{L_{r-1}\}^U + \mathcal{M}^U(r-1) + X\{R_{r-1}\}^U \\ &= \left(X^{-1}\{L_{r-1}\}_i^{\mathbf{G}} + \mathcal{Z}_r(r-1,i) + X\{R_{r-1}\}_i^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j}x_3^{2^j})(1 + x_3^{2^{r-1}}X) \\ &= \left(X^{-1}\{L_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}} + \mathcal{Z}_r(r-1,i+2^{r-1}) + X\{R_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j}x_3^{2^j})(X^{-1} + x_3^{2^{r-1}}) \\ &\forall i \in [0, 2^{r-1})\end{aligned}$$

Now let us rescale everything by $X^2$ to remove negative exponents.

$$\begin{aligned}&X\{L_{r-1}\}^U + X^2\mathcal{M}^U(r-1) + X^3\{R_{r-1}\}^U \\ &= \left(X^{-1}\{L_{r-1}\}_i^{\mathbf{G}} + \mathcal{Z}_r(r-1,i) + X\{R_{r-1}\}_i^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j}x_3^{2^j})(X^2 + x_3^{2^{r-1}}X^3) \\ &= \left(X^{-1}\{L_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}} + \mathcal{Z}_r(r-1,i+2^{r-1}) + X\{R_{r-1}\}_{i+2^{r-1}}^{\mathbf{G}}\right) \\ &\cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j}x_3^{2^j})(X + x_3^{2^{r-1}}X^2) \\ &\forall i \in [0, 2^{r-1})\end{aligned}$$

This gives us $2^{r-1}$ triples of maximal degree-$4$ polynomials in $X$ that agree at $u_{r-1}$ despite having coefficients determined prior to the choice of $u_{r-1}$. The probability that two of these polynomials would agree at $u_{r-1}$ and yet be distinct would be $\frac{4}{|\mathsf{Ch}|}$ by the Schwartz-Zippel lemma and so by the union bound the probability that the three of these polynomials agree and yet any of them is distinct from another is $\frac{8}{|\mathsf{Ch}|}$. By the union bound again the probability that any of the $2^{r-1}$ triples have multiple distinct polynomials is $\frac{2^{r-1} \cdot 8}{|\mathsf{Ch}|}$. By restricting the challenge space for $u_{r-1}$ accordingly we obtain $|\mathsf{BadCh}(\mathrm{tr}'|_{u_r})|/|\mathsf{Ch}| \leq \frac{2^{r-1} \cdot 8}{|\mathsf{Ch}|}$ for integers $0 < r \leq k$ and thus $|\mathsf{BadCh}(\mathrm{tr}'|_{u_k})|/|\mathsf{Ch}| \leq \frac{4n}{|\mathsf{Ch}|} \leq \epsilon$.

We can now conclude an equality of polynomials, and thus of coefficients. Consider the coefficients of the constant terms first, which gives us the $2^{r-1}$ equalities

$$0 = 0 = \mathbf{s}_i^{-1} z \left( \prod_{j=0}^{r-2} (1 + u_{k-1-j} x_3^{2^j}) \right) \cdot \{L_{r-1}\}^{\mathbf{G}}_{i+2^{r-1}} \forall i \in [0, 2^{r-1})$$

No value of $\mathbf{s}$ is zero, $z$ is never chosen to be $0$ and each $u_j$ is chosen so that $1 + u_{k-1-j} x_3^{2^j}$ is nonzero, so we can then conclude

$$0 = \{L_{r-1}\}^{\mathbf{G}}_{i+2^{r-1}} \forall i \in [0, 2^{r-1})$$

An identical process can be followed with respect to the coefficients of the $X^4$ term in the equalities to establish $0 = \{R_{r-1}\}^{\mathbf{G}}_i \forall i \in [0, 2^{r-1})$ contingent on $x_3$ being nonzero, which it always is. Substituting these in our equalities yields us something simpler

$$\begin{aligned}
&X\{L_{r-1}\}^U + X^2 \mathcal{M}^U(r-1) + X^3\{R_{r-1}\}^U \\
&= \left(X^{-1}\{L_{r-1}\}^{\mathbf{G}}_i + \mathcal{Z}_r(r-1, i)\right) \\
&\cdot \mathbf{s}_i^{-1} z \textstyle\prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j})(X^2 + x_3^{2^{r-1}} X^3) \\
&= \left(\mathcal{Z}_r(r-1, i+2^{r-1}) + X\{R_{r-1}\}^{\mathbf{G}}_{i+2^{r-1}}\right) \\
&\cdot \mathbf{s}_i^{-1} z \textstyle\prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j})(X + x_3^{2^{r-1}} X^2) \\
&\forall i \in [0, 2^{r-1})
\end{aligned}$$

Now we will consider the coefficients in $X$, which yield the equalities

$$\begin{aligned}
\{L_{r-1}\}^U \quad &= \mathbf{s}_i^{-1} z \textstyle\prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j}) \cdot \{L_{r-1}\}^{\mathbf{G}}_i \\
&= \mathbf{s}_i^{-1} z \textstyle\prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j}) \cdot \mathcal{Z}_r(r-1, i+2^{r-1}) \\
&\forall i \in [0, 2^{r-1})
\end{aligned}$$

which for similar reasoning as before yields the equalities

$$\{L_{r-1}\}^{\mathbf{G}}_i = \mathcal{Z}_r(r-1, i+2^{r-1}) \forall i \in [0, 2^{r-1})$$

Finally we will consider the coefficients in $X^2$ which yield the equalities

$$\begin{aligned}
\mathcal{M}^U(r-1) \quad &= \mathbf{s}_i^{-1} z \textstyle\prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j}) \cdot \left(\mathcal{Z}_r(r-1, i) + \{L_{r-1}\}^{\mathbf{G}}_i x_3^{2^{r-1}}\right) \\
&\forall i \in [0, 2^{r-1})
\end{aligned}$$

which by substitution gives us $\forall i \in [0, 2^{r-1})$

$$\mathcal{M}^U(r-1) = \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j} x_3^{2^j}) \cdot \left(\mathcal{Z}_r(r-1, i) + \mathcal{Z}_r(r-1, i+2^{r-1}) x_3^{2^{r-1}}\right)$$

Notice that by the definition of $\mathcal{Z}_{r-1}(m, i)$ we can rewrite this as

$$\mathcal{M}^U(r-1) = \mathcal{Z}_{r-1}(r-1,i) \cdot \mathbf{s}_i^{-1} z \prod_{j=0}^{r-2}(1 + u_{k-1-j}x_3^{2^j}) \forall i \in [0, 2^{r-1})$$

which is precisely in the form we set out to demonstrate.

We now proceed by induction from the case $r = k$ (which we know holds) to reach $r = 0$, which gives us

$$\mathcal{M}^U(0) = \mathcal{Z}_0(0,0) \cdot \mathbf{s}_0^{-1} z$$

and because $\mathcal{M}^U(0) = \{P'\}^U$ and $\mathcal{Z}_0(0,0) = \sum_{i=0}^{2^k-1} x_3^i \{P'\}_i^{\mathbf{G}}$, we obtain $\{P'\}^U = z \sum_{i=0}^{2^k-1} x_3^i \{P'\}_i^{\mathbf{G}}$, which completes the proof.

Having established that $\{P'\}^U = z \sum_{i=0}^{2^k-1} x_3^i \{P'\}_i^{\mathbf{G}}$, and given that $x_3$ and $\{P'\}_i^{\mathbf{G}}$ are fixed in advance of the choice of $z$, we have that at most one value of $z \in \mathsf{Ch}$ (which is nonzero) exists such that $\{P'\}^U = z \sum_{i=0}^{2^k-1} x_3^i \{P'\}_i^{\mathbf{G}}$ and yet $\{P'\}^U \neq 0$. By restricting $|\mathsf{BadCh}(\mathrm{tr}'|_z)|/|\mathsf{Ch}| \leq \frac{1}{|\mathsf{Ch}|} \leq \epsilon$ accordingly we obtain $\{P'\}^U = 0$ and therefore that the polynomial defined by $\{P'\}^{\mathbf{G}}$ has a root at $x_3$.

By construction $P' = P - [v]\mathbf{G}_0 + [\xi]S$, giving us that the polynomial defined by $\{P + [\xi]S\}^{\mathbf{G}}$ evaluates to $v$ at the point $x_3$. We have that $v, P, S$ are fixed prior to the choice of $\xi$, and so either the polynomial defined by $\{S\}^{\mathbf{G}}$ has a root at $x_3$ (which implies the polynomial defined by $\{P\}^{\mathbf{G}}$ evaluates to $v$ at the point $x_3$) or else $\xi$ is the single solution in $\mathsf{Ch}$ for which $\{P + [\xi]S\}^{\mathbf{G}}$ evaluates to $v$ at the point $x_3$ while $\{P\}^{\mathbf{G}}$ itself does not. We avoid the latter case by restricting $|\mathsf{BadCh}(\mathrm{tr}'|_\xi)|/|\mathsf{Ch}| \leq \frac{1}{|\mathsf{Ch}|} \leq \epsilon$ accordingly and can thus conclude that the polynomial defined by $\{P\}^{\mathbf{G}}$ evaluates to $v$ at $x_3$.

The remaining work deals strictly with the representations of group elements sent previously by the prover and their relationship with $P$ as well as the challenges chosen in each round of the protocol. We will simplify things first by using $p(X)$ to represent the polynomial defined by $\{P\}^{\mathbf{G}}$, as it is the case that this $p(X)$ corresponds exactly with the like-named polynomial in the protocol itself. We will make similar substitutions for the other group elements (and their corresponding polynomials) to aid in exposition, as the remainder of this proof is mainly tedious application of the Schwartz-Zippel lemma to upper bound the bad challenge space size for each of the remaining challenges in the protocol.

Recall that $P = Q' + x_4 \sum_{i=0}^{n_q-1} [x_4^i] Q_i$, and so by substitution we have $p(X) = q'(X) + x_4 \sum_{i=0}^{n_q-1} x_4^i q_i(X)$. Recall also that

$$v = \sum_{i=0}^{n_q-1} \left( x_2^i \left( \frac{\mathbf{u}_i - r_i(x_3)}{\prod_{j=0}^{n_e-1} \left( x_3 - \omega^{(\mathbf{q_i})_j} x \right)} \right) \right) + x_4 \sum_{i=0}^{n_q-1} x_4 \mathbf{u}_i$$

We have already established that $p(x_3) = v$. Notice that the coefficients in the above expressions for $v$ and $P$ are fixed prior to the choice of $x_4 \in \mathsf{Ch}$. By the Schwartz-Zippel lemma we have that only at most $n_q + 1$ possible choices of $x_4$ exist such that these expressions are satisfied and yet $q_i(x_3) \neq \mathbf{u}_i$ for any $i$ or

$$q'(x_3) \neq \sum_{i=0}^{n_q-1} \left( x_2^i \left( \frac{\mathbf{u}_i - r_i(x_3)}{\prod_{j=0}^{n_e-1} \left( x_3 - \omega^{(\mathbf{q_i})_j} x \right)} \right) \right)$$

By restricting $|\mathsf{BadCh}(\mathrm{tr}'|_{x_4})|/|\mathsf{Ch}| \leq \frac{n_q+1}{|\mathsf{Ch}|} \leq \epsilon$ we can conclude that all of the aforementioned inequalities are untrue. Now we can substitute $\mathbf{u}_i$ with $q_i(x_3)$ for all $i$ to obtain

$$q'(x_3) = \sum_{i=0}^{n_q-1} \left( x_2^i \left( \frac{q_i(x_3) - r_i(x_3)}{\prod_{j=0}^{n_e-1} \left( x_3 - \omega^{(\mathbf{q_i})_j} x \right)} \right) \right)$$

Suppose that $q'(X)$ (which is the polynomial defined by $\{Q'\}^{\mathbf{G}}$, and is of degree at most $n - 1$ ) does *not* take the form

$$\sum_{i=0}^{n_q-1} x_2^i \left( \frac{q_i(X) - r_i(X)}{\prod_{j=0}^{n_e-1} \left( X - \omega^{(\mathbf{q_i})_j} x \right)} \right)$$

and yet $q'(X)$ agrees with this expression at $x_3$ as we've established above. By the Schwartz-Zippel lemma this can only happen for at most $n - 1$ choices of $x_3 \in \mathsf{Ch}$ and so by restricting $|\mathsf{BadCh}(\mathrm{tr}'|_{x_3})|/|\mathsf{Ch}| \leq \frac{n-1}{|\mathsf{Ch}|} \leq \epsilon$ we obtain that

$$q'(X) = \sum_{i=0}^{n_q-1} x_2^i \left( \frac{q_i(X) - r_i(X)}{\prod_{j=0}^{n_e-1} \left( X - \omega^{(\mathbf{q_i})_j} x \right)} \right)$$

Next we will extract the coefficients of this polynomial in $x_2$ (which are themselves polynomials in formal indeterminate $X$) by again applying the Schwartz-Zippel lemma with respect to $x_2$; again, this leads to the restriction $|\mathsf{BadCh}(\mathrm{tr}'|_{x_2})|/|\mathsf{Ch}| \leq \frac{n_q}{|\mathsf{Ch}|} \leq \epsilon$ and we obtain the following polynomials of degree at most $n-1$ for all $i \in [0, n_q - 1)$

$$\frac{q_i(X) - r_i(X)}{\prod_{j=0}^{n_e-1} \left(X - \omega^{(\mathbf{q_i})_j} x\right)}$$

Having established that these are each non-rational polynomials of degree at most $n-1$ we can then say (by the factor theorem) that for each $i \in [0, n_q - 1]$ and $j \in [0, n_e - 1]$ we have that $q_i(X) - r_i(X)$ has a root at $\omega^{(\mathbf{q_i})_j} x$. Note that we can interpret each $q_i(X)$ as the restriction of a *bivariate* polynomial at the point $x_1$ whose degree with respect to $x_1$ is at most $n_a + 1$ and whose coefficients consist of various polynomials $a'_i(X)$ (from the representation $\{A'_i\}^{\mathbf{G}}$) as well as $h'(X)$ (from the representation $\{H'_i\}^{\mathbf{G}}$) and $r(X)$ (from the representation $\{R\}^{\mathbf{G}}$). By similarly applying the Schwartz-Zippel lemma and restricting the challenge space with $|\mathsf{BadCh}(\mathrm{tr}'|_{x_1})|/|\mathsf{Ch}| \leq \frac{n_a+1}{|\mathsf{Ch}|} \leq \epsilon$ we obtain (by construction of each $q'_i(X)$ and $r_i(X)$ in steps 12 and 13 of the protocol) that the prover's claimed value of $r$ in step 9 is equal to $r(x)$; that the value $h$ computed by the verifier in step 13 is equal to $h'(x)$; and that for all $i \in [0, n_q - 1]$ the prover's claimed values $(\mathbf{a_i})_j = a'_i(\omega^{(\mathbf{p_i})_j} x)$ for all $j \in [0, n_e - 1]$.

By construction of $h'(X)$ (from the representation $\{H'\}^{\mathbf{G}}$) in step 7 we know that $h'(x) = h(x)$ where by $h(X)$ we refer to the polynomial of degree at most $(n_g - 1) \cdot (n - 1)$ whose coefficients correspond to the concatenated representations of each $\{H_i\}^{\mathbf{G}}$. As before, suppose that $h(X)$ does *not* take the form $g'(X)/t(X)$. Then because $h(X)$ is determined prior to the choice of $x$ then by the Schwartz-Zippel lemma we know that it would only agree with $g'(X)/t(X)$ at $(n_g - 1) \cdot (n - 1)$ points at most if the polynomials were not equal. By restricting again $|\mathsf{BadCh}(\mathrm{tr}'|_{x})|/|\mathsf{Ch}| \leq \frac{(n_g-1)\cdot(n-1)}{|\mathsf{Ch}|} \leq \epsilon$ we obtain $h(X) = g'(X)/t(X)$ and because $h(X)$ is a non-rational polynomial by the factor theorem we obtain that $g'(X)$ vanishes over the domain $D$.

We now have that $g'(X)$ vanishes over $D$ but wish to show that $g(X, C_0, C_1, \cdots)$ vanishes over $D$ at all points to complete the proof. This just involves a sequence of applying the same technique to each of the challenges; since the polynomial $g(\cdots)$ has degree at most $n_g \cdot (n-1)$ in any indeterminate by definition, and because each polynomial $a_i(X, C_0, C_1, ..., C_{i-1}, \cdots)$ is determined prior to the choice of concrete challenge $c_i$ by similarly bounding $|\mathsf{BadCh}(\mathrm{tr}'|_{c_i})|/|\mathsf{Ch}| \leq \frac{n_g\cdot(n-1)}{|\mathsf{Ch}|} \leq \epsilon$ we ensure that $g(X, C_0, C_1, \cdots)$ vanishes over $D$, completing the proof.

# Implementation

# Halo 2 proofs

## Proofs as opaque byte streams

In proving system implementations like `bellman`, there is a concrete `Proof` struct that encapsulates the proof data, is returned by a prover, and can be passed to a verifier.

`halo2` does not contain any proof-like structures, for several reasons:

- The Proof structures would contain vectors of (vectors of) curve points and scalars. This complicates serialization/deserialization of proofs because the lengths of these vectors depend on the configuration of the circuit. However, we didn't want to encode the lengths of vectors inside of proofs, because at runtime the circuit is fixed, and thus so are the proof sizes.
- It's easy to accidentally put stuff into a Proof structure that isn't also placed in the transcript, which is a hazard when developing and implementing a proving system.
- We needed to be able to create multiple PLONK proofs at the same time; these proofs share many different substructures when they are for the same circuit.

Instead, `halo2` treats proof objects as opaque byte streams. Creation and consumption of these byte streams happens via the transcript:

- The `TranscriptWrite` trait represents something that we can write proof components to (at proving time).
- The `TranscriptRead` trait represents something that we can read proof components from (at verifying time).

Crucially, implementations of `TranscriptWrite` are responsible for simultaneously writing to some `std::io::Write` buffer at the same time that they hash things into the transcript, and similarly for `TranscriptRead` / `std::io::Read`.

As a bonus, treating proofs as opaque byte streams ensures that verification accounts for the cost of deserialization, which isn't negligible due to point compression.

# Proof encoding

A Halo 2 proof, constructed over a curve $E(\mathbb{F}_p)$, is encoded as a stream of:

- Points $P \in E(\mathbb{F}_p)$ (for commitments to polynomials), and
- Scalars $s \in \mathbb{F}_q$ (for evaluations of polynomials, and blinding values).

For the Pallas and Vesta curves, both points and scalars have 32-byte encodings, meaning that proofs are always a multiple of 32 bytes.

The `halo2` crate supports proving multiple instances of a circuit simultaneously, in order to share common proof components and protocol logic.

In the encoding description below, we will use the following circuit-specific constants:

- $k$ - the size parameter of the circuit (which has $2^k$ rows).
- $A$ - the number of advice columns.
- $F$ - the number of fixed columns.
- $I$ - the number of instance columns.
- $L$ - the number of lookup arguments.
- $P$ - the number of permutation arguments.
- $\mathsf{Col}_P$ - the number of columns involved in permutation argument $P$.
- $D$ - the maximum degree for the quotient polynomial.
- $Q_A$ - the number of advice column queries.
- $Q_F$ - the number of fixed column queries.
- $Q_I$ - the number of instance column queries.
- $M$ - the number of instances of the circuit that are being proven simultaneously.

As the proof encoding directly follows the transcript, we can break the encoding into sections matching the Halo 2 protocol:

- PLONK commitments:
  - $A$ points (repeated $M$ times).
  - $2L$ points (repeated $M$ times).
  - $P$ points (repeated $M$ times).
  - $L$ points (repeated $M$ times).
- Vanishing argument:
  - $D - 1$ points.
  - $Q_I$ scalars (repeated $M$ times).
  - $Q_A$ scalars (repeated $M$ times).
  - $Q_F$ scalars.

    - $D - 1$ scalars.

- PLONK evaluations:
    - $(2 + \mathsf{Col}_P) \times P$ scalars (repeated $M$ times).
    - $5L$ scalars (repeated $M$ times).
- Multiopening argument:
    - 1 point.
    - 1 scalar per set of points in the multiopening argument.
- Polynomial commitment scheme:
    - $1 + 2k$ points.
    - $2$ scalars.

# Fields

The Pasta curves that we use in `halo2` are designed to be highly 2-adic, meaning that a large $2^S$ multiplicative subgroup exists in each field. That is, we can write $p - 1 \equiv 2^S \cdot T$ with $T$ odd. For both Pallas and Vesta, $S = 32$; this helps to simplify the field implementations.

## Sarkar square-root algorithm (table-based variant)

We use a technique from Sarkar2020 to compute square roots in `halo2`. The intuition behind the algorithm is that we can split the task into computing square roots in each multiplicative subgroup.

Suppose we want to find the square root of $u$ modulo one of the Pasta primes $p$, where $u$ is a non-zero square in $\mathbb{Z}_p^\times$. We define a $2^S$ root of unity $g = z^T$ where $z$ is a non-square in $\mathbb{Z}_p^\times$, and precompute the following tables:

$$gtab = \begin{bmatrix} g^0 & g^1 & \dots & g^{2^8-1} \\ (g^{2^8})^0 & (g^{2^8})^1 & \dots & (g^{2^8})^{2^8-1} \\ (g^{2^{16}})^0 & (g^{2^{16}})^1 & \dots & (g^{2^{16}})^{2^8-1} \\ (g^{2^{24}})^0 & (g^{2^{24}})^1 & \dots & (g^{2^{24}})^{2^8-1} \end{bmatrix}$$

$$invtab = \begin{bmatrix} (g^{-2^{24}})^0 & (g^{-2^{24}})^1 & \dots & (g^{-2^{24}})^{2^8-1} \end{bmatrix}$$

Let $v = u^{(T-1)/2}$. We can then define $x = uv \cdot v = u^T$ as an element of the $2^S$ multiplicative subgroup.

Let $x_3 = x, x_2 = x_3^{2^8}, x_1 = x_2^{2^8}, x_0 = x_1^{2^8}$.

### i = 0, 1

Using $invtab$, we lookup $t_0$ such that

$$x_0 = (g^{-2^{24}})^{t_0} \implies x_0 \cdot g^{t_0 \cdot 2^{24}} = 1.$$

Define $\alpha_1 = x_1 \cdot (g^{2^{16}})^{t_0}$.

### i = 2

Lookup $t_1$ s.t.

$$\begin{aligned} \alpha_1 = (g^{-2^{24}})^{t_1} &\implies x_1 \cdot (g^{2^{16}})^{t_0} = (g^{-2^{24}})^{t_1} \\ &\implies x_1 \cdot g^{(t_0 + 2^8 \cdot t_1) \cdot 2^{16}} = 1. \end{aligned}$$

Define $\alpha_2 = x_2 \cdot (g^{2^8})^{t_0 + 2^8 \cdot t_1}$.

### i = 3

Lookup $t_2$ s.t.

$$\begin{aligned} \alpha_2 = (g^{-2^{24}})^{t_2} &\implies x_2 \cdot (g^{2^8})^{t_0 + 2^8 \cdot t_1} = (g^{-2^{24}})^{t_2} \\ &\implies x_2 \cdot g^{(t_0 + 2^8 \cdot t_1 + 2^{16} \cdot t_2) \cdot 2^8} = 1. \end{aligned}$$

Define $\alpha_3 = x_3 \cdot g^{t_0 + 2^8 \cdot t_1 + 2^{16} \cdot t_2}$.

### Final result

Lookup $t_3$ such that

$$\begin{aligned} \alpha_3 = (g^{-2^{24}})^{t_3} &\implies x_3 \cdot g^{t_0 + 2^8 \cdot t_1 + 2^{16} \cdot t_2} = (g^{-2^{24}})^{t_3} \\ &\implies x_3 \cdot g^{t_0 + 2^8 \cdot t_1 + 2^{16} \cdot t_2 + 2^{24} \cdot t_3} = 1. \end{aligned}$$

Let $t = t_0 + 2^8 \cdot t_1 + 2^{16} \cdot t_2 + 2^{24} \cdot t_3$.

We can now write

$$\begin{array}{rcccl} x_3 \cdot g^t = 1 & \implies & x_3 & = & g^{-t} \\ & \implies & uv^2 & = & g^{-t} \\ & \implies & uv & = & v^{-1} \cdot g^{-t} \\ & \implies & uv \cdot g^{t/2} & = & v^{-1} \cdot g^{-t/2}. \end{array}$$

Squaring the RHS, we observe that $(v^{-1}g^{-t/2})^2 = v^{-2}g^{-t} = u$. Therefore, the square root of $u$ is $uv \cdot g^{t/2}$; the first part we computed earlier, and the second part can be computed with three multiplications using lookups in $gtab$.

# Selector combining

Heavy use of custom gates can lead to a circuit defining many binary selectors, which would increase proof size and verification time.

This section describes an optimization, applied automatically by halo2, that combines binary selector columns into fewer fixed columns.

The basic idea is that if we have $\ell$ binary selectors labelled $1, \ldots, \ell$ that are enabled on disjoint sets of rows, then under some additional conditions we can combine them into a single fixed column, say $q$, such that:

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

However, the devil is in the detail.

The halo2 API allows defining some selectors to be "simple selectors", subject to the following condition:

> Every polynomial constraint involving a simple selector $s$ must be of the form $s \cdot t = 0$, where $t$ is a polynomial involving *no* simple selectors.

Suppose that $s$ has label $k$ in some set of $\ell$ simple selectors that are combined into $q$ as above. Then this condition ensures that replacing $s$ by $q \cdot \prod_{1 \le h \le \ell,\, h \neq k} (h - q)$ will not change the meaning of any constraints.

> It would be possible to relax this condition by ensuring that every use of a binary selector is substituted by a precise interpolation of its value from the corresponding combined selector. However,

- the restriction simplifies the implementation, developer tooling, and human understanding and debugging of the resulting constraint system;
- the scope to apply the optimization is not impeded very much by this restriction for typical circuits.

Note that replacing $s$ by $q \cdot \prod_{1 \leq h \leq \ell,\, h \neq k} (h - q)$ will increase the degree of constraints selected by $s$ by $\ell - 1$, and so we must choose the selectors that are combined in such a way that the maximum degree bound is not exceeded.

## Identifying selectors that can be combined

We need a partition of the overall set of selectors $s_0, \ldots, s_{m-1}$ into subsets (called "combinations"), such that no two selectors in a combination are enabled on the same row.

Labels must be unique within a combination, but they are not unique across combinations. Do not confuse a selector's index with its label.

Suppose that we are given **max_degree**, the degree bound of the circuit.

We use the following algorithm:

1. Leave nonsimple selectors unoptimized, i.e. map each of them to a separate fixed column.
2. Check (or ensure by construction) that all polynomial constraints involving each simple selector $s_i$ are of the form $s_i \cdot t_{i,j} = 0$ where $t_{i,j}$ do not involve any simple selectors. For each $i$, record the maximum degree of any $t_{i,j}$ as $d_i^{\mathsf{max}}$.
3. Compute a binary "exclusion matrix" $X$ such that $X_{j,i}$ is $1$ whenever $i \neq j$ and $s_i$ and $s_j$ are enabled on the same row; and $0$ otherwise.

   > Since $X$ is symmetric and is zero on the diagonal, we can represent it by either its upper or lower triangular entries. The rest of the algorithm is guaranteed only to access only the entries $X_{j,i}$ where $j > i$.

4. Initialize a boolean array $\mathsf{added}_{0..k-1}$ to all **false**.

   > $\mathsf{added}_i$ will record whether $s_i$ has been included in any combination.

5. Iterate over the $s_i$ that have not yet been added to any combination:

- a. Add $s_i$ to a fresh combination $c$, and set $\mathsf{added}_i = \mathsf{true}$.
- b. Let mut $d := d_i^{\mathsf{max}} - 1$.

  > $d$ is used to keep track of the largest degree, *excluding* the selector expression, of any gate involved in the combination $c$ so far.

- c. Iterate over all the selectors $s_j$ for $j > i$ that can potentially join $c$, i.e. for which $\mathsf{added}_j$ is false:
  - i. (Optimization) If $d + \mathsf{len}(c) = \mathsf{max\_degree}$, break to the outer loop, since no more selectors can be added to $c$.
  - ii. Let $d^{\mathsf{new}} = \mathsf{max}(d, d_j^{\mathsf{max}} - 1)$.
  - iii. If $X_{j,i'}$ is $\mathsf{true}$ for any $i'$ in $c$, or if $d^{\mathsf{new}} + (\mathsf{len}(c) + 1) > \mathsf{max\_degree}$, break to the outer loop.

    > $d^{\mathsf{new}} + (\mathsf{len}(c) + 1)$ is the maximum degree, *including* the selector expression, of any constraint that would result from adding $s_j$ to the combination $c$.

  - iv. Set $d := d^{\mathsf{new}}$.
  - v. Add $s_j$ to $c$ and set $\mathsf{added}_j := \mathsf{true}$.
- d. Allocate a fixed column $q_c$, initialized to all-zeroes.
- e. For each selector $s' \in c$:
  - i. Label $s'$ with a distinct index $k$ where $1 \leq k \leq \mathsf{len}(c)$.
  - ii. Record that $s'$ should be substituted with $q_c \cdot \prod_{1 \leq h \leq \mathsf{len}(c),\, h \neq k} (h - q_c)$ in all gate constraints.
  - iii. For each row $r$ such that $s'$ is enabled at $r$, assign the value $k$ to $q_c$ at row $r$.

The above algorithm is implemented in halo2_proofs/src/plonk/circuit/compress_selectors.rs. This is used by the `compress_selectors` function of halo2_proofs/src/plonk/circuit.rs which does the actual substitutions.

## Writing circuits to take best advantage of selector combining

For this optimization it is beneficial for a circuit to use simple selectors as far as possible, rather than fixed columns. It is usually not beneficial to do manual combining of selectors, because

the resulting fixed columns cannot take part in the automatic combining. That means that to get comparable results you would need to do a global optimization manually, which would interfere with writing composable gadgets.

Whether two selectors are enabled on the same row (and so are inhibited from being combined) depends on how regions are laid out by the floor planner. The currently implemented floor planners do not attempt to take this into account. We suggest not worrying about it too much — the gains that can be obtained by cajoling a floor planner to shuffle around regions in order to improve combining are likely to be relatively small.

# Gadgets

In this section we document the gadgets and chip designs provided in the `halo2_gadgets` crate.

> Neither these gadgets, nor their implementations, have been reviewed, and they should not be used in production.

# Elliptic Curves

## EccChip

`halo2_gadgets` provides a chip that implements `EccInstructions` using 10 advice columns. The chip is currently restricted to the Pallas curve, but will be extended to support the Vesta curve in the near future.

### Chip assumptions

A non-exhaustive list of assumptions made by `EccChip`:

- $0$ is not an $x$-coordinate of a valid point on the curve.
  - Holds for Pallas because $5$ is not square in $\mathbb{F}_q$.
- $0$ is not a $y$-coordinate of a valid point on the curve.
  - Holds for Pallas because $-5$ is not a cube in $\mathbb{F}_q$.

### Layout

The following table shows how columns are used by the gates for various chip sub-areas:

- $W$ - witnessing points.
- $AI$ - incomplete point addition.
- $AC$ - complete point addition.
- $MF$ - Fixed-base scalar multiplication.
- $MVI$ - variable-base scalar multiplication, incomplete rounds.
- $MVC$ - variable-base scalar multiplication, complete rounds.
- $MVO$ - variable-base scalar multiplication, overflow check.

| Sub-area | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $W$ | $x$ | $y$ | | | | | | | | |
| $AI$ | $x_p$ | $y_p$ | $x_q$ | $y_q$ | | | | | | |
| | | | $x_r$ | $y_r$ | | | | | | |
| $AC$ | $x_p$ | $y_p$ | $x_q$ | $y_q$ | $\lambda$ | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ | |
| | | | $x_r$ | $y_r$ | | | | | | |
| $MF$ | $x_p$ | $y_p$ | $x_q$ | $y_q$ | window | $u$ | | | | |
| | | | $x_r$ | $y_r$ | | | | | | |
| $MVI$ | $x_p$ | $y_p$ | $\lambda_2^{lo}$ | $x_A^{hi}$ | $\lambda_1^{hi}$ | $\lambda_2^{hi}$ | $z^{lo}$ | $x_A^{lo}$ | $\lambda_1^{lo}$ | $z^{hi}$ |
| $MVC$ | $x_p$ | $y_p$ | $x_q$ | $y_q$ | $\lambda$ | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ | $z^{complete}$ |
| | | | $x_r$ | $y_r$ | | | | | | |

# Witnessing points

We represent elliptic curve points in the circuit in their affine representation $(x, y)$. The identity is represented as the pseudo-coordinate $(0, 0)$, which we assume is not a valid point on the curve.

## Non-identity points

To constrain a coordinate pair $(x, y)$ as representing a valid point on the curve, we directly check the curve equation. For Pallas and Vesta, this is:

$$y^2 = x^3 + 5$$

| Degree | Constraint |
|---|---|
| 4 | $q_{\text{point}}^{\text{non-id}} \cdot (y^2 - x^3 - 5) = 0$ |

## Points including the identity

To allow $(x, y)$ to represent either a valid point on the curve, or the pseudo-coordinate $(0, 0)$, we define a separate gate that enforces the curve equation check unless both $x$ and $y$ are zero.

| Degree | Constraint |
|---|---|
| 5 | $(q_{\text{point}} \cdot x) \cdot (y^2 - x^3 - 5) = 0$ |
| 5 | $(q_{\text{point}} \cdot y) \cdot (y^2 - x^3 - 5) = 0$ |

We will use formulae for curve arithmetic using affine coordinates on short Weierstrass curves, derived from section 4.1 of Hüseyin Hışıl's thesis.

## Incomplete addition

- Inputs: $P = (x_p, y_p), Q = (x_q, y_q)$
- Output: $R = P \div Q = (x_r, y_r)$

The formulae from Hışıl's thesis are:

- $x_3 = \left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - x_1 - x_2$
- $y_3 = \frac{y_1 - y_2}{x_1 - x_2} \cdot (x_1 - x_3) - y_1.$

Rename $(x_1, y_1)$ to $(x_q, y_q)$, $(x_2, y_2)$ to $(x_p, y_p)$, and $(x_3, y_3)$ to $(x_r, y_r)$, giving

- $x_r = \left(\frac{y_q - y_p}{x_q - x_p}\right)^2 - x_q - x_p$
- $y_r = \frac{y_q - y_p}{x_q - x_p} \cdot (x_q - x_r) - y_q$

which is equivalent to

- $x_r + x_q + x_p = \left(\frac{y_p - y_q}{x_p - x_q}\right)^2$
- $y_r + y_q = \frac{y_p - y_q}{x_p - x_q} \cdot (x_q - x_r).$

Assuming $x_p \neq x_q$, we have

$$\begin{aligned}
x_r + x_q + x_p &= \left(\frac{y_p - y_q}{x_p - x_q}\right)^2 \\
\iff (x_r + x_q + x_p) \cdot (x_p - x_q)^2 &= (y_p - y_q)^2 \\
\iff (x_r + x_q + x_p) \cdot (x_p - x_q)^2 - (y_p - y_q)^2 &= 0
\end{aligned}$$

and

$$\begin{aligned}
y_r + y_q &= \frac{y_p - y_q}{x_p - x_q} \cdot (x_q - x_r) \\
\iff (y_r + y_q) \cdot (x_p - x_q) &= (y_p - y_q) \cdot (x_q - x_r) \\
\iff (y_r + y_q) \cdot (x_p - x_q) - (y_p - y_q) \cdot (x_q - x_r) &= 0.
\end{aligned}$$

So we get the constraints:

- $(x_r + x_q + x_p) \cdot (x_p - x_q)^2 - (y_p - y_q)^2 = 0$
  - Note that this constraint is unsatisfiable for $P \div (-P)$ (when $P \neq \mathcal{O}$), and so cannot be used with arbitrary inputs.
- $(y_r + y_q) \cdot (x_p - x_q) - (y_p - y_q) \cdot (x_q - x_r) = 0.$

### Constraints

| Degree | Constraint |
|---|---|
| 4 | $q_{\text{add-incomplete}} \cdot \left((x_r + x_q + x_p) \cdot (x_p - x_q)^2 - (y_p - y_q)^2\right) = 0$ |
| 3 | $q_{\text{add-incomplete}} \cdot \left((y_r + y_q) \cdot (x_p - x_q) - (y_p - y_q) \cdot (x_q - x_r)\right) = 0$ |

## Complete addition

$$\begin{aligned}
\mathcal{O} &+ \mathcal{O} &&= \mathcal{O} \\
\mathcal{O} &+ (x_q, y_q) &&= (x_q, y_q) \\
(x_p, y_p) &+ \mathcal{O} &&= (x_p, y_p) \\
(x, y) &+ (x, y) &&= [2](x, y) \\
(x, y) &+ (x, -y) &&= \mathcal{O} \\
(x_p, y_p) &+ (x_q, y_q) &&= (x_p, y_p) \div (x_q, y_q), \text{ if } x_p \neq x_q.
\end{aligned}$$

Suppose that we represent $\mathcal{O}$ as $(0, 0)$. ($0$ is not an $x$-coordinate of a valid point because we would need $y^2 = x^3 + 5$, and $5$ is not square in $\mathbb{F}_q$. Also $0$ is not a $y$-coordinate of a valid point because $-5$ is not a cube in $\mathbb{F}_q$.)

$$
\begin{aligned}
P + Q &= R \\
(x_p, y_p) + (x_q, y_q) &= (x_r, y_r) \\
\lambda &= \frac{y_q - y_p}{x_q - x_p} \\
x_r &= \lambda^2 - x_p - x_q \\
y_r &= \lambda(x_p - x_r) - y_p
\end{aligned}
$$

For the doubling case, Hışıl's thesis tells us that $\lambda$ has to instead be computed as $\frac{3x^2}{2y}$.

Define $\mathsf{inv0}(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1/x, & \text{otherwise.} \end{cases}$

Witness $\alpha, \beta, \gamma, \delta, \lambda$ where:

$$
\begin{aligned}
\alpha &= \mathsf{inv0}(x_q - x_p) \\
\beta &= \mathsf{inv0}(x_p) \\
\gamma &= \mathsf{inv0}(x_q) \\
\delta &= \begin{cases} \mathsf{inv0}(y_q + y_p), & \text{if } x_q = x_p \\ 0, & \text{otherwise} \end{cases} \\
\lambda &= \begin{cases} \frac{y_q - y_p}{x_q - x_p}, & \text{if } x_q \neq x_p \\ \frac{3{x_p}^2}{2y_p} & \text{if } x_q = x_p \wedge y_p \neq 0 \\ 0, & \text{otherwise.} \end{cases}
\end{aligned}
$$

## Constraints

| Degree | Constraint | | | Meaning |
|---|---|---|---|---|
| 4 | $q_{add} \cdot (x_q - x_p) \cdot ((x_q - x_p) \cdot \lambda - (y_q - y_p))$ | $=$ | $0$ | $x_q \neq x_p \implies \lambda =$ |
| 5 | $q_{add} \cdot (1 - (x_q - x_p) \cdot \alpha) \cdot (2y_p \cdot \lambda - 3{x_p}^2)$ | $=$ | $0$ | $\begin{cases} x_q = x_p \wedge y_p \neq \\ x_q = x_p \wedge y_p = \end{cases}$ |
| 6 | $q_{add} \cdot x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda^2 - x_p - x_q - x_r)$ | $=$ | $0$ | $x_p \neq 0 \wedge x_q \neq 0 /$ |
| 6 | $q_{add} \cdot x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r)$ | $=$ | $0$ | $x_p \neq 0 \wedge x_q \neq 0 /$ |
| 6 | $q_{add} \cdot x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda^2 - x_p - x_q - x_r)$ | $=$ | $0$ | $x_p \neq 0 \wedge x_q \neq 0 /$ |
| 6 | $q_{add} \cdot x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r)$ | $=$ | $0$ | $x_p \neq 0 \wedge x_q \neq 0 /$ |
| 4 | $q_{add} \cdot (1 - x_p \cdot \beta) \cdot (x_r - x_q)$ | $=$ | $0$ | $x_p = 0 \implies x_r =$ |
| 4 | $q_{add} \cdot (1 - x_p \cdot \beta) \cdot (y_r - y_q)$ | $=$ | $0$ | $x_p = 0 \implies y_r =$ |
| 4 | $q_{add} \cdot (1 - x_q \cdot \gamma) \cdot (x_r - x_p)$ | $=$ | $0$ | $x_q = 0 \implies x_r =$ |
| 4 | $q_{add} \cdot (1 - x_q \cdot \gamma) \cdot (y_r - y_p)$ | $=$ | $0$ | $x_q = 0 \implies y_r =$ |
| 4 | $q_{add} \cdot (1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot x_r$ | $=$ | $0$ | $x_q = x_p \wedge y_q = -$ |
| 4 | $q_{add} \cdot (1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot y_r$ | $=$ | $0$ | $x_q = x_p \wedge y_q = -$ |

Max degree: 6

# Analysis of constraints

1. $(x_q - x_p) \cdot ((x_q - x_p) \cdot \lambda - (y_q - y_p)) = 0$

   At least one of $x_q - x_p = 0$
   or $(x_q - x_p) \cdot \lambda - (y_q - y_p) = 0$
   must be satisfied for the constraint to be satisfied.

   If $x_q - x_p \neq 0$, then $(x_q - x_p) \cdot \lambda - (y_q - y_p) = 0$, and
   by rearranging both sides we get $\lambda = (y_q - y_p)/(x_q - x_p)$.

   Therefore:
   $$x_q \neq x_p \implies \lambda = (y_q - y_p)/(x_q - x_p).$$

2. $(1 - (x_q - x_p) \cdot \alpha) \cdot (2y_p \cdot \lambda - 3x_p^2) = 0$

   At least one of $(1 - (x_q - x_p) \cdot \alpha) = 0$
   or $(2y_p \cdot \lambda - 3x_p^2) = 0$
   must be satisfied for the constraint to be satisfied.

   If $x_q = x_p$, then $1 - (x_q - x_p) \cdot \alpha = 0$ has no solution for $\alpha$,
   so it must be that $2y_p \cdot \lambda - 3x_p^2 = 0$.

   If $x_q = x_p$ and $y_p = 0$ then $x_p = 0$, and the constraint is satisfied.

   If $x_q = x_p$ and $y_p \neq 0$ then by rearranging both sides
   we get $\lambda = 3x_p^2/2y_p$.

   Therefore:
   $$(x_q = x_p) \wedge y_p \neq 0 \implies \lambda = 3x_p^2/2y_p.$$

3. a) $x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda^2 - x_p - x_q - x_r) = 0$
   b) $x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r) = 0$
   c) $x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda^2 - x_p - x_q - x_r) = 0$
   d) $x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r) = 0$

   At least one of $x_p = 0$
   or $x_p = 0$
   or $(x_q - x_p) = 0$
   or $(\lambda^2 - x_p - x_q - x_r) = 0$
   must be satisfied for constraint (a) to be satisfied.

If $x_p \neq 0 \wedge x_q \neq 0 \wedge x_q \neq x_p$,

• Constraint (a) imposes that $x_r = \lambda^2 - x_p - x_q$.
• Constraint (b) imposes that $y_r = \lambda \cdot (x_p - x_r) - y_p$.

If $x_p \neq 0 \wedge x_q \neq 0 \wedge y_q \neq -y_p$,

• Constraint (c) imposes that $x_r = \lambda^2 - x_p - x_q$.
• Constraint (d) imposes that $y_r = \lambda \cdot (x_p - x_r) - y_p$.

Therefore:

$$(x_p \neq 0) \wedge (x_q \neq 0) \wedge ((x_q \neq x_p) \vee (y_q \neq -y_p))$$
$$\implies (x_r = \lambda^2 - x_p - x_q) \wedge (y_r = \lambda \cdot (x_p - x_r) - y_p).$$

4. a) $(1 - x_p \cdot \beta) \cdot (x_r - x_q) = 0$
   b) $(1 - x_p \cdot \beta) \cdot (y_r - y_q) = 0$

At least one of $1 - x_p \cdot \beta = 0$
or $x_r - x_q = 0$
must be satisfied for constraint (a) to be satisfied.

If $x_p = 0$ then $1 - x_p \cdot \beta = 0$ has no solutions for $\beta$,
and so it must be that $x_r - x_q = 0$.

Similarly, constraint (b) imposes that if $x_p = 0$
then $y_r - y_q = 0$.

Therefore:

$$x_p = 0 \implies (x_r, y_r) = (x_q, y_q).$$

5. a) $(1 - x_q \cdot \gamma) \cdot (x_r - x_p) = 0$
   b) $(1 - x_q \cdot \gamma) \cdot (y_r - y_p) = 0$

At least one of $1 - x_q \cdot \gamma = 0$
or $x_r - x_p = 0$
must be satisfied for constraint (a) to be satisfied.

If $x_q = 0$ then $1 - x_q \cdot \gamma = 0$ has no solutions for $\gamma$,
and so it must be that $x_r - x_p = 0$.

Similarly, constraint (b) imposes that if $x_q = 0$
then $y_r - y_p = 0$.

Therefore:

$$x_q = 0 \implies (x_r, y_r) = (x_p, y_p).$$

6. a) $(1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot x_r = 0$
   b) $(1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot y_r = 0$

At least one of $1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta = 0$
or $x_r = 0$
must be satisfied for constraint (a) to be satisfied,
and similarly replacing $x_r$ by $y_r$.

If $x_r \neq 0$ or $y_r \neq 0,$ then it must be that $1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta = 0.$

However, if $x_q = x_p \land y_q = -y_p,$ then there are no solutions for $\alpha$ and $\delta$.

Therefore:

$$x_q = x_p \land y_q = -y_p \implies (x_r, y_r) = (0, 0).$$

**Propositions:**

(1) $x_q \neq x_p \implies \lambda = (y_q - y_p)/(x_q - x_p)$
(2) $(x_q = x_p) \land y_p \neq 0 \implies \lambda = 3x_p^2/2y_p$
(3) $(x_p \neq 0) \land (x_q \neq 0) \land ((x_q \neq x_p) \lor (y_q \neq -y_p))$
$\implies (x_r = \lambda^2 - x_p - x_q) \land (y_r = \lambda \cdot (x_p - x_r) - y_p)$
(4) $x_p = 0 \implies (x_r, y_r) = (x_q, y_q)$
(5) $x_q = 0 \implies (x_r, y_r) = (x_p, y_p)$
(6) $x_q = x_p \land y_q = -y_p \implies (x_r, y_r) = (0, 0)$

**Cases:**

$(x_p, y_p) + (x_q, y_q) = (x_r, y_r)$

Note that we rely on the fact that 0 is not a valid $x$-coordinate or $y$-coordinate of a point on the Pallas curve other than $\mathcal{O}$.

- $(0, 0) + (0, 0)$
    - Completeness:

$(1)$ holds because $x_q = x_p$
$(2)$ holds because $y_p = 0$
$(3)$ holds because $x_p = 0$
$(4)$ holds because $(x_r, y_r) = (x_q, y_q) = (0, 0)$
$(5)$ holds because $(x_r, y_r) = (x_p, y_p) = (0, 0)$
$(6)$ holds because $(x_r, y_r) = (0, 0)$.

- Soundness: $(x_r, y_r) = (0, 0)$ is the only solution to $(6)$.

- $(x, y) + (0, 0)$ for $(x, y) \neq (0, 0)$
  - Completeness:

$(1)$ holds because $x_q \neq x_p,$ therefore $\lambda = (y_q - y_p)/(x_q - x_p)$ is a solution
$(2)$ holds because $x_q \neq x_p,$ therefore $\alpha = (x_q - x_p)^{-1}$ is a solution
$(3)$ holds because $x_q = 0$
$(4)$ holds because $x_p \neq 0,$ therefore $\beta = x_p^{-1}$ is a solution
$(5)$ holds because $(x_r, y_r) = (x_p, y_p)$
$(6)$ holds because $x_q \neq x_p,$ therefore $\alpha = (x_q - x_p)^{-1}$ and $\delta = 0$ is a soluti

  - Soundness: $(x_r, y_r) = (x_p, y_p)$ is the only solution to $(5)$.

- $(0, 0) + (x, y)$ for $(x, y) \neq (0, 0)$
  - Completeness:

$(1)$ holds because $x_q \neq x_p,$ therefore $\lambda = (y_q - y_p)/(x_q - x_p)$ is a solution
$(2)$ holds because $x_q \neq x_p,$ therefore $\alpha = (x_q - x_p)^{-1}$ is a solution
$(3)$ holds because $x_p = 0$
$(4)$ holds because $x_p = 0$ only when $(x_r, y_r) = (x_q, y_q)$
$(5)$ holds because $x_q \neq 0,$ therefore $\gamma = x_q^{-1}$ is a solution
$(6)$ holds because $x_q \neq x_p,$ therefore $\alpha = (x_q - x_p)^{-1}$ and $\delta = 0$ is a soluti

  - Soundness: $(x_r, y_r) = (x_q, y_q)$ is the only solution to $(4)$.

- $(x, y) + (x, y)$ for $(x, y) \neq (0, 0)$
  - Completeness:

$(1)$ holds because $x_q = x_p$
$(2)$ holds because $x_q = x_p \wedge y_p \neq 0,$ therefore $\lambda = 3x_p^2/2y_p$ is a solution
$(3)$ holds because $x_r = \lambda^2 - x_p - x_q \wedge y_r = \lambda \cdot (x_p - x_r) - y_p$ in this case
$(4)$ holds because $x_p \neq 0,$ therefore $\beta = x_p^{-1}$ is a solution
$(5)$ holds because $x_p \neq 0,$ therefore $\gamma = x_q^{-1}$ is a solution
$(6)$ holds because $x_q = x_p$ and $y_q \neq -y_p,$ therefore $\alpha = 0$ and $\delta = (y_q + y_$

- Soundness: $\lambda$ is computed correctly, and $(x_r, y_r) = (\lambda^2 - x_p - x_q, \lambda \cdot (x_p - x_r) - y_p)$ is the only solution.

- $(x, y) + (x, -y)$ for $(x, y) \neq (0, 0)$
  - Completeness:

    $(1)$ $\text{holds because } x_q = x_p$
    $(2)$ $\text{holds because } x_q = x_p \wedge y_p \neq 0, \text{ therefore } \lambda = 3x_p^2/2y_p \text{ is a solution (although } \lambda \text{ is not used in this case)}$
    $(3)$ $\text{holds because } x_q = x_p \text{ and } y_q = -y_p$
    $(4)$ $\text{holds because } x_p \neq 0, \text{ therefore } \beta = x_p^{-1} \text{ is a solution}$
    $(5)$ $\text{holds because } x_q \neq 0, \text{ therefore } \gamma = x_q^{-1} \text{ is a solution}$
    $(6)$ $\text{holds because } (x_r, y_r) = (0, 0)$

  - Soundness: $(x_r, y_r) = (0, 0)$ is the only solution to $(6)$.

- $(x_p, y_p) + (x_q, y_q)$ for $(x_p, y_p) \neq (0, 0)$ and $(x_q, y_q) \neq (0, 0)$ and $x_p \neq x_q$
  - Completeness:

    $(1)$ $\text{holds because } x_q \neq x_p, \text{ therefore } \lambda = (y_q - y_p)/(x_q - x_p) \text{ is a solution}$
    $(2)$ $\text{holds because } x_q \neq x_p, \text{ therefore } \alpha = (x_q - x_p)^{-1} \text{ is a solution}$
    $(3)$ $\text{holds because } x_r = \lambda^2 - x_p - x_q \wedge y_r = \lambda \cdot (x_p - x_r) - y_p \text{ in this case}$
    $(4)$ $\text{holds because } x_p \neq 0, \text{ therefore } \beta = x_p^{-1} \text{ is a solution}$
    $(5)$ $\text{holds because } x_q \neq 0, \text{ therefore } \gamma = x_q^{-1} \text{ is a solution}$
    $(6)$ $\text{holds because } x_q \neq x_p, \text{ therefore } \alpha = (x_q - x_p)^{-1} \text{ and } \delta = 0 \text{ is a soluti}$

  - Soundness: $\lambda$ is computed correctly, and $(x_r, y_r) = (\lambda^2 - x_p - x_q, \lambda \cdot (x_p - x_r) - y_p)$ is the only solution.

# Fixed-base scalar multiplication

There are $6$ fixed bases in the Orchard protocol:

- $\mathcal{K}^{\text{Orchard}}$, used in deriving the nullifier;
- $\mathcal{G}^{\text{Orchard}}$, used in spend authorization;
- $\mathcal{R}$ base for $\mathsf{NoteCommit}^{\text{Orchard}}$;
- $\mathcal{V}$ and $\mathcal{R}$ bases for $\mathsf{ValueCommit}^{\text{Orchard}}$; and
- $\mathcal{R}$ base for $\mathsf{Commit}^{\text{ivk}}$.

# Decompose scalar

We support fixed-base scalar multiplication with three types of scalars:

## Full-width scalar

A $255$-bit scalar from $\mathbb{F}_q$. We decompose a full-width scalar $\alpha$ into $85$ $3$-bit windows:

$$\alpha = k_0 + k_1 \cdot (2^3)^1 + \cdots + k_{84} \cdot (2^3)^{84}, k_i \in [0..2^3).$$

The scalar multiplication will be computed correctly for $k_{0..84}$ representing any integer in the range $[0, 2^{255})$ - that is, the scalar is allowed to be non-canonical.

We range-constrain each $3$-bit word of the scalar decomposition using a polynomial range-check constraint:

| Degree | Constraint |
|---|---|
| $9$ | $q_{\texttt{mul\_fixed\_full}} \cdot \texttt{range\_check}(\text{word}, 2^3) = 0$ |

where $\texttt{range\_check}(\text{word}, \texttt{range}) = \text{word} \cdot (1 - \text{word}) \cdots (\texttt{range} - 1 - \text{word})$.

## Base field element

We support using a base field element as the scalar in fixed-base multiplication. This occurs, for example, in the scalar multiplication for the nullifier computation of the Action circuit $\mathsf{DeriveNullifier}_{\mathsf{nk}} = \mathsf{Extract}_{\mathbb{P}}\left(\left[(\mathsf{PRF}^{\mathsf{nfOrchard}}_{\mathsf{nk}}(\rho) + \psi) \bmod q_{\mathbb{P}}\right] \mathcal{K}^{\mathsf{Orchard}} + \mathsf{cm}\right)$: here, the scalar

$$\left[(\mathsf{PRF}^{\mathsf{nfOrchard}}_{\mathsf{nk}}(\rho) + \psi) \bmod q_{\mathbb{P}}\right]$$

is the result of a base field addition.

Decompose the base field element $\alpha$ into three-bit windows, and range-constrain each window, using the short range decomposition gadget in strict mode, with $W = 85, K = 3$.

If $k_{0..84}$ is witnessed directly then no issue of canonicity arises. However, because the scalar is given as a base field element here, care must be taken to ensure a canonical representation, since $2^{255} > p$. That is, we must check that $0 \leq \alpha < p$, where $p$ the is Pallas base field modulus

$$p = 2^{254} + t_p = 2^{254} + 45560315531419706090280762371685220353.$$

Note that $t_p < 2^{130}$.

To do this, we decompose $\alpha$ into three pieces:

$$\alpha = \alpha_0 \ (252 \text{ bits}) \ || \ \alpha_1 \ (2 \text{ bits}) \ || \ \alpha_2 \ (1 \text{ bit}) \ .$$

We check the correctness of this decomposition by:

| Degree | Constraint |
|---|---|
| 5 | $q_{\text{canon-base-field}} \cdot \texttt{range\_check}(\alpha_1, 2^2) = 0$ |
| 3 | $q_{\text{canon-base-field}} \cdot \texttt{bool\_check}(\alpha_2) = 0$ |
| 2 | $q_{\text{canon-base-field}} \cdot \left(z_{84} - (\alpha_1 + \alpha_2 \cdot 2^2)\right) = 0$ |

If the MSB $\alpha_2 = 0$ is not set, then $\alpha < 2^{254} < p$. However, in the case where $\alpha_2 = 1$, we must check:

- $\alpha_2 = 1 \implies \alpha_1 = 0$;
- $\alpha_2 = 1 \implies \alpha_0 < t_p$:
  - $\alpha_2 = 1 \implies 0 \leq \alpha_0 < 2^{130}$,
  - $\alpha_2 = 1 \implies 0 \leq \alpha_0 + 2^{130} - t_p < 2^{130}$

To check that $0 \leq \alpha_0 < 2^{130}$, we make use of the three-bit running sum decomposition:

- Firstly, we constrain $\alpha_0$ to be a $132$-bit value by enforcing its high $120$ bits to be all-zero. We can get $\mathsf{alpha\_0\_hi\_120}$ from the decomposition:

$$\begin{aligned} z_{44} &= k_{44} + 2^3 k_{45} + \cdots + 2^{3\cdot(84-44)} k_{84} \\ \implies \mathsf{alpha\_0\_hi\_120} &= z_{44} - 2^{3\cdot(84-44)} k_{84} \\ &= z_{44} - 2^{3\cdot(40)} z_{84}. \end{aligned}$$

- Then, we constrain bits $130..{=}131$ of $\alpha_0$ to be zeroes; in other words, we constrain the three-bit word $k_{43} = \alpha[129..{=}131] = \alpha_0[129..{=}131] \in \{0, 1\}$. We make use of the running sum decomposition to obtain $k_{43} = z_{43} - z_{44} \cdot 2^3$.

Define $\alpha_0' = \alpha_0 + 2^{130} - t_p$. To check that $0 \leq \alpha_0' < 2^{130}$, we use 13 ten-bit lookups, where we constrain the $z_{13}$ running sum output of the lookup to be $0$ if $\alpha_2 = 1$.

| Degree | Constraint | Comment |
|---|---|---|
| 2 | $q_{\text{canon-base-field}} \cdot (\alpha_0' - (\alpha_0 + 2^{130} - t_{\mathbb{P}})) = 0$ | |
| 3 | $q_{\text{canon-base-field}} \cdot \alpha_2 \cdot \alpha_1 = 0$ | $\alpha_2 = 1 \implies \alpha_1 = 0$ |
| 3 | $q_{\text{canon-base-field}} \cdot \alpha_2 \cdot \mathsf{alpha\_0\_hi\_120} = 0$ | Constrain $\alpha_0$ to be a 132-bit va |
| 4 | $q_{\text{canon-base-field}} \cdot \alpha_2 \cdot \texttt{bool\_check}(k_{43}) = 0$ | Constrain $\alpha_0[130..{=}131]$ to 0 |
| 3 | $q_{\text{canon-base-field}} \cdot \alpha_2 \cdot z_{13}(\texttt{lookup}(\alpha_0', 13)) = 0$ | $\alpha_2 = 1 \implies 0 \leq \alpha_0' < 2^{130}$ |

### Short signed scalar

A short signed scalar is witnessed as a magnitude $m$ and sign $s$ such that

$$\begin{aligned} s &\in \{-1, 1\} \\ m &\in [0, 2^{64}) \\ \mathsf{v}^{\mathsf{old}} - \mathsf{v}^{\mathsf{new}} &= s \cdot m. \end{aligned}$$

This is used for $\mathsf{ValueCommit}^{\mathsf{Orchard}}$. We want to compute $\mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}} - \mathsf{v}^{\mathsf{new}}) = [\mathsf{v}^{\mathsf{old}} - \mathsf{v}^{\mathsf{new}}]\mathcal{V} + [\mathsf{rcv}]\mathcal{R}$, where

$$-(2^{64} - 1) \leq \mathsf{v}^{\mathsf{old}} - \mathsf{v}^{\mathsf{new}} \leq 2^{64} - 1$$

$\mathsf{v}^{\mathsf{old}}$ and $\mathsf{v}^{\mathsf{new}}$ are each already constrained to $64$ bits (by their use as inputs to $\mathsf{NoteCommit}^{\mathsf{Orchard}}$).

Decompose the magnitude $m$ into three-bit windows, and range-constrain each window, using the short range decomposition gadget in strict mode, with $W = 22, K = 3$.

We have two additional constraints:

| Degree | Constraint | Comment |
|---|---|---|
| 3 | $q_{\texttt{mul\_fixed\_short}} \cdot \texttt{bool\_check}(k_{21}) = 0$ | The last window must be a single bi |
| 3 | $q_{\texttt{mul\_fixed\_short}} \cdot \left(s^2 - 1\right) = 0$ | The sign must be $1$ or $-1$. |

where $\texttt{bool\_check}(x) = x \cdot (1 - x)$.

## Load fixed base

Then, we precompute multiples of the fixed base $B$ for each window. This takes the form of a window table: $M[0..W)[0..8)$ such that:

- for the first (W-1) rows $M[0..(W-1))[0..8)$:

$$M[w][k] = [(k+2) \cdot (2^3)^w]B$$

- in the last row $M[W-1][0..8)$:

$$M[w][k] = [k \cdot (2^3)^w - \sum_{j=0}^{83} 2^{3j+1}]B$$

The additional $(k+2)$ term lets us avoid adding the point at infinity in the case $k = 0$. We offset these accumulated terms by subtracting them in the final window, i.e. we subtract

$\sum_{j=0}^{W-2} 2^{3j+1}$.

> Note: Although an offset of $(k+1)$ would naively suffice, it introduces an edge case when $k_0 = 7, k_1 = 0$. In this case, the window table entries evaluate to the same point:
>
> - $M[0][k_0] = [(7+1) * (2^3)^0]B = [8]B$,
> - $M[1][k_1] = [(0+1) * (2^3)^1]B = [8]B$.
>
> In fixed-base scalar multiplication, we sum the multiples of $B$ at each window (except the last) using incomplete addition. Since the point doubling case is not handled by incomplete addition, we avoid it by using an offset of $(k+2)$.

For each window of fixed-base multiples $M[w] = (M[w][0], \cdots, M[w][7]), w \in [0..(W-1))$:

- Define a Lagrange interpolation polynomial $\mathcal{L}_x(k)$ that maps $k \in [0..8)$ to the $x$-coordinate of the multiple $M[w][k]$, i.e.

$$\mathcal{L}_x(k) = \begin{cases} ([(k+2) \cdot (2^3)^w]B)_x & \text{for } w \in [0..(W-1)); \\ ([k \cdot (2^3)^w - \sum_{j=0}^{83} 2^{3j+1}]B)_x & \text{for } w = 84; \text{ and} \end{cases}$$

- Find a value $z_w$ such that $z_w + (M[w][k])_y$ is a square $u^2$ in the field, but the wrong-sign $y$-coordinate $z_w - (M[w][k])_y$ does not produce a square.

Repeating this for all $W$ windows, we end up with:

- an $W \times 8$ table $\mathcal{L}_x$ storing 8 coefficients interpolating the $x-$coordinate for each window. Each $x$-coordinate interpolation polynomial will be of the form

$$\mathcal{L}_x[w](k) = c_0 + c_1 \cdot k + c_2 \cdot k^2 + \cdots + c_7 \cdot k^7,$$

  where $k \in [0..8), w \in [0..85)$ and $c_k$'s are the coefficients for each power of $k$; and
- a length-$W$ array $Z$ of $z_w$'s.

We load these precomputed values into fixed columns whenever we do fixed-base scalar multiplication in the circuit.

# Fixed-base scalar multiplication

Given a decomposed scalar $\alpha$ and a fixed base $B$, we compute $[\alpha]B$ as follows:

1. For each $k_w, w \in [0..85), k_w \in [0..8)$ in the scalar decomposition, witness the $x$- and $y$-coordinates $(x_w, y_w) = M[w][k_w]$.
2. Check that $(x_w, y_w)$ is on the curve: $y_w^2 = x_w^3 + b$.
3. Witness $u_w$ such that $y_w + z_w = u_w^2$.
4. For all windows but the last, use incomplete addition to sum the $M[w][k_w]$'s, resulting in $[\alpha - k_{84} \cdot (2^3)^{84} + \sum_{j=0}^{83} 2^{3j+1}]B$.
5. For the last window, use complete addition $M[83][k_{83}] + M[84][k_{84}]$ and return the final result.

> Note: complete addition is required in the final step to correctly map $[0]B$ to a representation of the point at infinity, $(0, 0)$; and also to handle a corner case for which the last step is a doubling.

Constraints:

| Degree | Constraint |
|---|---|
| 8 | $q_{\text{mul-fixed}} \cdot (\mathcal{L}_x[w](k_w) - x_w) = 0$ |
| 4 | $q_{\text{mul-fixed}} \cdot \left(y_w^2 - x_w^3 - b\right) = 0$ |
| 3 | $q_{\text{mul-fixed}} \cdot \left(u_w^2 - y_w - Z[w]\right) = 0$ |

where $b = 5$ (from the Pallas curve equation).

## Signed short exponent

Recall that the signed short exponent is witnessed as a $64-$bit magnitude $m$, and a sign $s \in 1, -1$. Using the above algorithm, we compute $P = [m]\mathcal{B}$. Then, to get the final result $P'$, we conditionally negate $P$ using $(x, y) \mapsto (x, s \cdot y)$.

Constraints:

| Degree | Constraint |
|---|---|
| 3 | $q_{\texttt{mul\_fixed\_short}} \cdot \left(P'_y - P_y\right) \cdot \left(P'_y + P_y\right) = 0$ |
| 3 | $q_{\texttt{mul\_fixed\_short}} \cdot \left(s \cdot P'_y - P_y\right) = 0$ |

## Layout

| $x_P$ | $y_P$ | $x_{QR}$ | $y_{QR}$ | $u$ | window | $L_{0..=7}$ | fixed_z |
|---|---|---|---|---|---|---|---|
| $x_{P,0}$ | $y_{P,0}$ | | | $u_0$ | $\text{window}_0$ | $L_{0..=7,0}$ | $\text{fixed\_z}_0$ |
| $x_{P,1}$ | $y_{P,1}$ | $x_{Q,1} = x_{P,0}$ | $y_{Q,1} = y_{P,0}$ | $u_1$ | $\text{window}_1$ | $L_{0..=7,1}$ | $\text{fixed\_z}_1$ |
| $x_{P,2}$ | $y_{P,2}$ | $x_{Q,2} = x_{R,1}$ | $y_{Q,2} = y_{R,1}$ | $u_2$ | $\text{window}_2$ | $L_{0..=7,1}$ | $\text{fixed\_z}_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Note: this doesn't include the last row that uses complete addition. In the implementation this is allocated in a different region.

# Variable-base scalar multiplication

In the Orchard circuit we need to check $\mathsf{pk_d} = [\mathsf{ivk}]\mathsf{g_d}$ where $\mathsf{ivk} \in [0, p)$ and the scalar field is $\mathbb{F}_q$ with $p < q$.

We have $p = 2^{254} + t_p$ and $q = 2^{254} + t_q$, for $t_p, t_q < 2^{128}$.

## Witness scalar

We're trying to compute $[\alpha]T$ for $\alpha \in [0, q)$. Set $k = \alpha + t_q$ and $n = 254$. Then we can compute

$$\begin{aligned}[2^{254} + (\alpha + t_q)]T &= [2^{254} + (\alpha + t_q) - (2^{254} + t_q)]T \\ &= [\alpha]T\end{aligned}$$

provided that $\alpha + t_q \in [0, 2^{n+1})$, i.e. $\alpha < 2^{n+1} - t_q$ which covers the whole range we need because in fact $2^{255} - t_q > q$.

Thus, given a scalar $\alpha$, we witness the boolean decomposition of $k = \alpha + t_q$. (We use big-endian bit order for convenient input into the variable-base scalar multiplication algorithm.)

$$k = k_{254} \cdot 2^{254} + k_{253} \cdot 2^{253} + \cdots + k_0.$$

## Variable-base scalar multiplication

We use an optimized double-and-add algorithm, copied from "Faster variable-base scalar multiplication in zk-SNARK circuits" with some variable name changes:

```
Acc := [2] T
for i from n-1 down to 0 {
    P := k_{i+1} ? T : -T
    Acc := (Acc + P) + Acc
}
return (k_0 = 0) ? (Acc - T) : Acc
```

It remains to check that the x-coordinates of each pair of points to be added are distinct.

When adding points in a prime-order group, we can rely on Theorem 3 from Appendix C of the Halo paper, which says that if we have two such points with nonzero indices wrt a given odd-prime order base, where the indices taken in the range $-(q-1)/2..(q-1)/2$ are distinct disregarding sign, then they have different x-coordinates. This is helpful, because it is easier to reason about the indices of points occurring in the scalar multiplication algorithm than it is to reason about their x-coordinates directly.

So, the required check is equivalent to saying that the following "indexed version" of the above algorithm never asserts:

```
acc := 2
for i from n-1 down to 0 {
    p = k_{i+1} ? 1 : -1
    assert acc ≠ ± p
    assert (acc + p) ≠ acc    // X
    acc := (acc + p) + acc
    assert 0 < acc ≤ (q-1)/2
}
if k_0 = 0 {
    assert acc ≠ 1
    acc := acc - 1
}
```

The maximum value of `acc` is:

```
    <--- n 1s --->
  1011111...111111
= 1100000...000000 - 1
```

= $2^{n+1} + 2^n - 1$

> The assertion labelled X obviously cannot fail because $p \neq 0$. It is possible to see that acc is monotonically increasing except in the last conditional. It reaches its largest value when $k$ is maximal, i.e. $2^{n+1} + 2^n - 1$.

So to entirely avoid exceptional cases, we would need $2^{n+1} + 2^n - 1 < (q-1)/2$. But we can use $n$ larger by $c$ if the last $c$ iterations use complete addition.

The first $i$ for which the algorithm using **only** incomplete addition fails is going to be $252$, since $2^{252+1} + 2^{252} - 1 > (q-1)/2$. We need $n = 254$ to make the wraparound technique above work.

```
sage: q = 0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001
sage: 2^253 + 2^252 - 1 < (q-1)//2
False
sage: 2^252 + 2^251 - 1 < (q-1)//2
True
```

So the last three iterations of the loop ($i = 2..0$) need to use complete addition, as does the conditional subtraction at the end. Writing this out using $\dotplus$ for incomplete addition (as we do in the spec), we have:

```
Acc := [2] T
for i from 253 down to 3 {
    P := k_{i+1} ? T : -T
    Acc := (Acc ∔ P) ∔ Acc
}
for i from 2 down to 0 {
    P := k_{i+1} ? T : -T
    Acc := (Acc + P) + Acc  // complete addition
}
return (k_0 = 0) ? (Acc + (-T)) : Acc  // complete addition
```

## Constraint program for optimized double-and-add

Define a running sum $\mathbf{z_j} = \sum_{i=j}^{n}(\mathbf{k}_i \cdot 2^{i-j})$, where $n = 254$ and:

$$
\begin{aligned}
&\mathbf{z}_{n+1} = 0, \\
&\mathbf{z}_n = \mathbf{k}_n, \qquad (\text{most significant bit}) \\
&\mathbf{z}_0 = k.
\end{aligned}
$$

Initialize $A_{254} = [2]T$.

for $i$ from 254 down to 4 :
$\quad \texttt{bool\_check}(\mathbf{k}_i) = 0$
$\quad \mathbf{z}_i = 2\mathbf{z}_{i+1} + \mathbf{k}_i$
$\quad x_{P,i} = x_T$
$\quad y_{P,i} = (2\mathbf{k}_i - 1) \cdot y_T \qquad \text{(conditionally negate)}$
$\quad \lambda_{1,i} \cdot (x_{A,i} - x_{P,i}) = y_{A,i} - y_{P,i}$
$\quad x_{R,i} = \lambda_{1,i}^2 - x_{A,i} - x_{P,i}$
$\quad (\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - x_{R,i}) = 2y_{\mathsf{A},i}$
$\quad \lambda_{2,i}^2 = x_{A,i-1} + x_{R,i} + x_{A,i}$
$\quad \lambda_{2,i} \cdot (x_{A,i} - x_{A,i-1}) = y_{A,i} + y_{A,i-1}.$

The helper $\texttt{bool\_check}(x) = x \cdot (1 - x)$. After substitution of $x_{P,i}, y_{P,i}, x_{R,i}, y_{A,i}$, and $y_{A,i-1}$, this becomes:

Initialize $A_{254} = [2]T$.

for $i$ from 254 down to 4 :
$\quad // \text{ let } \mathbf{k}_i = \mathbf{z}_i - 2\mathbf{z}_{i+1}$
$\quad // \text{ let } y_{A,i} = \frac{(\lambda_{1,i}+\lambda_{2,i}) \cdot (x_{A,i} - (\lambda_{1,i}^2 - x_{A,i} - x_T))}{2}$
$\quad \texttt{bool\_check}(\mathbf{k}_i) = 0$
$\quad \lambda_{1,i} \cdot (x_{A,i} - x_T) = y_{A,i} - (2\mathbf{k}_i - 1) \cdot y_T$
$\quad \lambda_{2,i}^2 = x_{A,i-1} + \lambda_{1,i}^2 - x_T$
$\quad \begin{cases} \lambda_{2,i} \cdot (x_{A,i} - x_{A,i-1}) = y_{A,i} + y_{A,i-1}, & \text{if } i > 4 \\ \lambda_{2,4} \cdot (x_{A,4} - x_{A,3}) = y_{A,4} + y_{A,3}^{\text{witnessed}}, & \text{if } i = 4. \end{cases}$

Here, $y_{A,3}^{\text{witnessed}}$ is assigned to a cell. This is unlike previous $y_{A,i}$'s, which were implicitly derived from $\lambda_{1,i}, \lambda_{2,i}, x_{A,i}, x_T$, but never actually assigned.

The bits $\mathbf{k}_{3\ldots1}$ are used in three further steps, using complete addition:

for $i$ from 3 down to 1 :
$\quad // \text{ let } \mathbf{k}_i = \mathbf{z}_i - 2\mathbf{z}_{i+1}$
$\quad \texttt{bool\_check}(\mathbf{k}_i) = 0$
$\quad (x_{A,i-1}, y_{A,i-1}) = ((x_{A,i}, y_{A,i}) + (x_T, y_T)) + (x_{A,i}, y_{A,i})$

If the least significant bit $\mathbf{k_0} = 1$, we set $B = \mathcal{O}$, otherwise we set $B = -T$. Then we return $A + B$ using complete addition.

Let $B = \begin{cases} (0, 0), & \text{if } \mathbf{k_0} = 1, \\ (x_T, -y_T), & \text{otherwise.} \end{cases}$

Output $(x_{A,0}, y_{A,0}) + B$.

(Note that $(0, 0)$ represents $\mathcal{O}$.)

## Incomplete addition

We need six advice columns to witness $(x_T, y_T, \lambda_1, \lambda_2, x_{A,i}, \mathbf{z}_i)$. However, since $(x_T, y_T)$ are the same, we can perform two incomplete additions in a single row, reusing the same $(x_T, y_T)$. We split the scalar bits used in incomplete addition into $hi$ and $lo$ halves and process them in parallel. This means that we effectively have two for loops:

- the first, covering the $hi$ half for $i$ from $254$ down to $130$, with a special case at $i = 130$; and
- the second, covering the $lo$ half for the remaining $i$ from $129$ down to $4$, with a special case at $i = 4$.

| $x_T$ | $y_T$ | $z^{hi}$ | $x_A^{hi}$ | $\lambda_1^{hi}$ | $\lambda_2^{hi}$ | $q_1^{hi}$ | $q_2^{hi}$ | $q_3^{hi}$ | $z^{lo}$ | $x$ |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $\mathbf{z}_{255} = 0$ | | $y_{A,254} = 2[T]_y$ | | $1$ | $0$ | $0$ | $\mathbf{z}_{130}$ | |
| $x_T$ | $y_T$ | $\mathbf{z}_{254}$ | $x_{A,254} = 2[T]_x$ | $\lambda_{1,254}$ | $\lambda_{2,254}$ | $0$ | $1$ | $0$ | $\mathbf{z}_{129}$ | $x_A$ |
| $x_T$ | $y_T$ | $\mathbf{z}_{253}$ | $x_{A,253}$ | $\lambda_{1,253}$ | $\lambda_{2,253}$ | $0$ | $1$ | $0$ | $\mathbf{z}_{128}$ | $x_A$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | |
| $x_T$ | $y_T$ | $\mathbf{z}_{130}$ | $x_{A,130}$ | $\lambda_{1,130}$ | $\lambda_{2,130}$ | $0$ | $0$ | $1$ | $\mathbf{z}_5$ | $x$ |
| $x_T$ | $y_T$ | | $x_{A,129}$ | $y_{A,129}$ | | | | | $\mathbf{z}_4$ | $x$ |
| | | | | | | | | | | $x$ |

For each $hi$ and $lo$ half, we have three sets of gates. Note that $i$ is going from $255.. = 3$; $i$ is NOT indexing the rows.

### $q_1 = 1$

This gate is only used on the first row (before the for loop). We check that $\lambda_1, \lambda_2$ are initialized to values consistent with the initial $y_A$.

| Degree | Constraint |
|---|---|
| $4$ | $q_1 \cdot \left(y_{A,n}^{\text{witnessed}} - y_{A,n}\right) = 0$ |

where

$$y_{A,n} = \frac{(\lambda_{1,n} + \lambda_{2,n}) \cdot (x_{A,n} - (\lambda_{1,n}^2 - x_{A,n} - x_T))}{2},$$

$y_{A,n}^{\text{witnessed}}$ is witnessed.

## $q_2 = 1$

This gate is used on all rows corresponding to the for loop except the last.

| Degree | Constraint |
|---|---|
| 2 | $q_2 \cdot (x_{T,cur} - x_{T,next}) = 0$ |
| 2 | $q_2 \cdot (y_{T,cur} - y_{T,next}) = 0$ |
| 3 | $q_2 \cdot \texttt{bool\_check}(\mathbf{k}_i) = 0, \text{ where } \mathbf{k}_i = \mathbf{z}_i - 2\mathbf{z}_{i+1}$ |
| 4 | $q_2 \cdot (\lambda_{1,i} \cdot (x_{A,i} - x_{T,i}) - y_{A,i} + (2\mathbf{k}_i - 1) \cdot y_{T,i}) = 0$ |
| 3 | $q_2 \cdot \left(\lambda_{2,i}^2 - x_{A,i-1} - x_{R,i} - x_{A,i}\right) = 0$ |
| 3 | $q_2 \cdot (\lambda_{2,i} \cdot (x_{A,i} - x_{A,i-1}) - y_{A,i} - y_{A,i-1}) = 0$ |

where

$$\begin{aligned}
x_{R,i} &= \lambda_{1,i}^2 - x_{A,i} - x_T, \\
y_{A,i} &= \frac{(\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - (\lambda_{1,i}^2 - x_{A,i} - x_T))}{2}, \\
y_{A,i-1} &= \frac{(\lambda_{1,i-1} + \lambda_{2,i-1}) \cdot (x_{A,i-1} - (\lambda_{1,i-1}^2 - x_{A,i-1} - x_T))}{2},
\end{aligned}$$

## $q_3 = 1$

This gate is used on the final iteration of the for loop, handling the special case where we check that the output $y_A$ has been witnessed correctly.

| Degree | Constraint |
|---|---|
| 3 | $q_3 \cdot \texttt{bool\_check}(\mathbf{k}_i) = 0, \text{ where } \mathbf{k}_i = \mathbf{z}_i - 2\mathbf{z}_{i+1}$ |
| 4 | $q_3 \cdot (\lambda_{1,i} \cdot (x_{A,i} - x_{T,i}) - y_{A,i} + (2\mathbf{k}_i - 1) \cdot y_{T,i}) = 0$ |
| 3 | $q_3 \cdot \left(\lambda_{2,i}^2 - x_{A,i-1} - x_{R,i} - x_{A,i}\right) = 0$ |
| 3 | $q_3 \cdot \left(\lambda_{2,i} \cdot (x_{A,i} - x_{A,i-1}) - y_{A,i} - y_{A,i-1}^{\text{witnessed}}\right) = 0$ |

where

$$
\begin{aligned}
x_{R,i} &= \lambda_{1,i}^2 - x_{A,i} - x_T, \\
y_{A,i} &= \frac{(\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - (\lambda_{1,i}^2 - x_{A,i} - x_T))}{2}, \\
y_{A,i-1}^{\text{witnessed}} &\text{ is witnessed.}
\end{aligned}
$$

## Complete addition

We reuse the complete addition constraints to implement the final $c$ rounds of double-and-add. This requires two rows per round because we need 9 advice columns for each complete addition. In the 10th advice column we stash the other cells that we need to correctly implement the double-and-add:

- The base $y$ coordinate, so we can conditionally negate it as input to one of the complete additions.
- The running sum, which we constrain over two rows instead of sequentially.

### Layout

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $q_{\texttt{mul\_decompose\_var}}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $x_T$ | $y_p$ | $x_A$ | $y_A$ | $\lambda_1$ | $\alpha_1$ | $\beta_1$ | $\gamma_1$ | $\delta_1$ | $z_{i+1}$ | $0$ |
| $x_A$ | $y_A$ | $x_q$ | $y_q$ | $\lambda_2$ | $\alpha_2$ | $\beta_2$ | $\gamma_2$ | $\delta_2$ | $y_T$ | $1$ |
| | | $x_r$ | $y_r$ | | | | | | $z_i$ | $0$ |

### Constraints

In addition to the complete addition constraints, we define the following gate:

| Degree | Constraint |
|---|---|
| | $q_{\texttt{mul\_decompose\_var}} \cdot \texttt{bool\_check}(\mathbf{k}_i) = 0$ |
| | $q_{\texttt{mul\_decompose\_var}} \cdot \texttt{ternary}(\mathbf{k}_i, y_T - y_p, y_T + y_p) = 0$ |

where $\mathbf{k}_i = \mathbf{z}_i - 2\mathbf{z}_{i+1}$.

# LSB

## Layout

| $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $q_{\texttt{mul\_lsb}}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $x_p$ | $y_p$ | $x_A$ | $y_A$ | $\lambda$ | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ | $z_1$ | $1$ |
| $x_T$ | $y_T$ | $x_r$ | $y_r$ | | | | | | $z_0$ | $0$ |

## Constraints

| Degree | Constraint |
|---|---|
| | $q_{\texttt{mul\_lsb}} \cdot \texttt{bool\_check}(\mathbf{k}_0) = 0$ |
| | $q_{\texttt{mul\_lsb}} \cdot \texttt{ternary}(\mathbf{k}_0, x_p, x_p - x_T) = 0$ |
| | $q_{\texttt{mul\_lsb}} \cdot \texttt{ternary}(\mathbf{k}_0, y_p, y_p + y_T) = 0$ |

where $\mathbf{k}_0 = \mathbf{z}_0 - 2\mathbf{z}_1$.

# Overflow check

$\mathbf{z}_i$ cannot overflow for any $i \geq 1$, because it is a weighted sum of bits only up to $2^{n-1} = 2^{253}$, which is smaller than $p$ (and also $q$).

However, $\mathbf{z}_0 = \alpha + t_q$ *can* overflow $[0, p)$.

> Note: for full-width scalar mul, it may not be possible to represent $\mathbf{z}_0$ in the base field (e.g. when the base field is Pasta's $\mathbb{F}_p$ and $p < q$). In that case, we need to special-case the row that would mention $\mathbf{z}_0$ so that it is correct for whatever representation we use for a full-width scalar. Our representation for $k$ will be the pair $(\mathbb{k}_{254}, k' = k - 2^{254} \cdot \mathbb{k}_{254})$. We'll use $k'$ in place of $\alpha + t_q$ for $\mathbf{z}_0$, constraining $k'$ to 254 bits so that it fits in an $\mathbb{F}_p$ element. Then we just have to generalize the argument below to work for $k' \in [0, 2 \cdot t_q)$ (because the maximum value of $\alpha + t_q$ is $q - 1 + t_q = 2^{254} + t_q - 1 + t_q$).

Since overflow can only occur in the final step that constrains $\mathbf{z}_0 = 2 \cdot \mathbf{z}_1 + \mathbf{k}_0$, we have $\mathbf{z}_0 = k \pmod p$. It is then sufficient to also check that $\mathbf{z}_0 = \alpha + t_q \pmod p$ (so that $k = \alpha + t_q \pmod p$) and that $k \in [t_q, p + t_q)$. These conditions together imply that $k = \alpha + t_q$ as an integer, and so $2^{254} + k = \alpha \pmod q$ as required.

Note: the bits $\mathbf{k}_{254..0}$ do not represent a value reduced modulo $q$, but rather a representation of the unreduced $\alpha + t_q$.

## Optimized check for $k \in [t_q, p + t_q)$

Since $t_p + t_q < 2^{130}$ (also true if $p$ and $q$ are swapped), we have

$$[t_q, p + t_q) = [t_q, t_q + 2^{130}) \ \cup \ [2^{130}, 2^{254}) \ \cup \ \big([2^{254}, 2^{254} + 2^{130}) \ \cap \ [p + t_q - 2^{130}, p + t_q)$$

We may assume that $k = \alpha + t_q \pmod p$.

(This is true for the use of variable-base scalar mul in Orchard, where we know that $\alpha < p$. If is also true if we swap $p$ and $q$ so that we have $p > q$. It is *not* true for a full-width scalar $\alpha \geq p$ when $p < q$.)

Therefore,

$$\begin{aligned}
k \in [t_q, p + t_q) \quad &\Leftrightarrow \quad \big(k \in [t_q, t_q + 2^{130}) \ \vee \ k \in [2^{130}, 2^{254})\big) \vee \\
&\qquad \big(k \in [2^{254}, 2^{254} + 2^{130}) \ \wedge \ k \in [p + t_q - 2^{130}, p + t_q)\big) \\
&\Leftrightarrow \quad \big(\mathbf{k}_{254} = 0 \implies (k \in [t_q, t_q + 2^{130}) \ \vee \ k \in [2^{130}, 2^{254}))\big) \wedge \\
&\qquad \big(\mathbf{k}_{254} = 1 \implies (k \in [2^{254}, 2^{254} + 2^{130}) \ \wedge \ k \in [p + t_q - 2^{130}, p + \\
&\Leftrightarrow \quad \big(\mathbf{k}_{254} = 0 \implies (\alpha \in [0, 2^{130}) \ \vee \ k \in [2^{130}, 2^{254}))\big) \wedge \\
&\qquad \big(\mathbf{k}_{254} = 1 \implies (k \in [2^{254}, 2^{254} + 2^{130}) \ \wedge \ (\alpha + 2^{130}) \bmod p \in [0,
\end{aligned}$$

Given $k \in [2^{254}, 2^{254} + 2^{130})$, we prove equivalence of $k \in [p + t_q - 2^{130}, p + t_q)$ and $(\alpha + 2^{130}) \bmod p \in [0, 2^{130})$ as follows:

- shift the range by $2^{130} - p - t_q$ to give $k + 2^{130} - p - t_q \in [0, 2^{130})$;
- observe that $k + 2^{130} - p - t_q$ is guaranteed to be in $[2^{130} - t_p - t_q, 2^{131} - t_p - t_q)$ and therefore cannot overflow or underflow modulo $p$;
- using the fact that $k = \alpha + t_q \pmod p$, observe that $(k + 2^{130} - p - t_q) \bmod p = (\alpha + t_q + 2^{130} - p - t_q) \bmod p = (\alpha + 2^{130}) \bmod p$.

(We can see in a different way that this is correct by observing that it checks whether $\alpha \bmod p \in [p - 2^{130}, p)$, so the upper bound is aligned as we would expect.)

Now, we can continue optimizing from Ⓐ:

$$
\begin{aligned}
k \in [t_q, p + t_q) \quad &\Leftrightarrow \quad \big(\mathbf{k}_{254} = 0 \implies (\alpha \in [0, 2^{130}) \ \vee \ k \in [2^{130}, 2^{254})\big) \ \wedge \\
&\quad\ \ \big(\mathbf{k}_{254} = 1 \implies (k \in [2^{254}, 2^{254} + 2^{130}) \ \wedge \ (\alpha + 2^{130}) \bmod p \in [0,\\
\\
&\Leftrightarrow \quad \big(\mathbf{k}_{254} = 0 \implies (\alpha \in [0, 2^{130}) \ \vee \ \mathbf{k}_{253..130} \text{ are not all } 0)\big) \ \wedge \\
&\quad\ \ \big(\mathbf{k}_{254} = 1 \implies (\mathbf{k}_{253..130} \text{ are all } 0 \ \wedge \ (\alpha + 2^{130}) \bmod p \in [0, 2^{130})
\end{aligned}
$$

Constraining $\mathbf{k}_{253..130}$ to be all-0 or not-all-0 can be implemented almost "for free", as follows.

Recall that $\mathbf{z}_i = \sum_{h=i}^{n}(\mathbf{k}_h \cdot 2^{h-i})$, so we have:

$$
\begin{aligned}
\mathbf{z}_{130} &= \sum_{h=130}^{254}(\mathbf{k}_h \cdot 2^{h-130}) \\
\mathbf{z}_{130} &= \mathbf{k}_{254} \cdot 2^{254-130} + \sum_{h=130}^{253}(\mathbf{k}_h \cdot 2^{h-130}) \\
\mathbf{z}_{130} - \mathbf{k}_{254} \cdot 2^{124} &= \sum_{h=130}^{253}(\mathbf{k}_h \cdot 2^{h-130})
\end{aligned}
$$

So $\mathbf{k}_{253..130}$ are all 0 exactly when $\mathbf{z}_{130} = \mathbf{k}_{254} \cdot 2^{124}$.

Finally, we can merge the $130$-bit decompositions for the $\mathbf{k}_{254} = 0$ and $\mathbf{k}_{254} = 1$ cases by checking that $(\alpha + \mathbf{k}_{254} \cdot 2^{130}) \bmod p \in [0, 2^{130})$.

### Overflow check constraints

Let $s = \alpha + \mathbf{k}_{254} \cdot 2^{130}$. The constraints for the overflow check are:

$$
\begin{aligned}
\mathbf{z}_0 &= \alpha + t_q \pmod p \\
\mathbf{k}_{254} = 1 &\implies (\mathbf{z}_{130} = 2^{124} \ \wedge \ s \bmod p \in [0, 2^{130})) \\
\mathbf{k}_{254} = 0 &\implies (\mathbf{z}_{130} \neq 0 \ \vee \ s \bmod p \in [0, 2^{130}))
\end{aligned}
$$

Define $\mathsf{inv0}(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1/x, & \text{otherwise.} \end{cases}$

Witness $\eta = \mathsf{inv0}(\mathbf{z}_{130})$, and decompose $s \bmod p$ as $\mathbf{s}_{129..0}$.

Then the needed gates are:

| Degree | Constraint |
|---|---|
| 2 | $q_{\texttt{mul\_overflow}} \cdot \big(s - (\alpha + \mathbf{k}_{254} \cdot 2^{130})\big) = 0$ |
| 2 | $q_{\texttt{mul\_overflow}} \cdot (\mathbf{z}_0 - \alpha - t_q) = 0$ |
| 3 | $q_{\texttt{mul\_overflow}} \cdot \big(\mathbf{k}_{254} \cdot (\mathbf{z}_{130} - 2^{124})\big) = 0$ |
| 3 | $q_{\texttt{mul\_overflow}} \cdot \left(\mathbf{k}_{254} \cdot (s - \sum_{i=0}^{129} 2^i \cdot \mathbf{s}_i)/2^{130}\right) = 0$ |
| 5 | $q_{\texttt{mul\_overflow}} \cdot \left((1 - \mathbf{k}_{254}) \cdot (1 - \mathbf{z}_{130} \cdot \eta) \cdot (s - \sum_{i=0}^{129} 2^i \cdot \mathbf{s}_i)/2^{130}\right) = 0$ |

where $(s - \sum_{i=0}^{129} 2^i \cdot \mathbf{s}_i)/2^{130}$ can be computed by another running sum. Note that the factor of $1/2^{130}$ has no effect on the constraint, since the RHS is zero.

#### Running sum range check

We make use of a $10$-bit lookup range check in the circuit to subtract the low $130$ bits of $\mathbf{s}$. The range check subtracts the first $13 \cdot 10$ bits of $\mathbf{s}$, and right-shifts the result to give $(s - \sum_{i=0}^{129} 2^i \cdot \mathbf{s}_i)/2^{130}$.

## Overflow check (general)

Recall that we defined $\mathbf{z}_j = \sum_{i=j}^{n} (\mathbf{k}_i \cdot 2^{i-j})$, where $n = 254$.

$\mathbf{z}_j$ cannot overflow for any $j \geq 1$, because it is a weighted sum of bits only up to and including $2^{n-j}$. When $n = 254$ and $j = 1$ this sum can be at most $2^{254} - 1$, which is smaller than $p$ (and also $q$).

However, for full-width scalar mul, it may not be possible to represent $\mathbf{z}_0$ in the base field (e.g. when the base field is Pasta's $\mathbb{F}_p$ and $p < q$). In that case $\mathbf{z}_0 = \alpha + t_q$ *can* overflow $[0, p)$.

So, we need to special-case the row that would mention $\mathbf{z}_0$ so that it is correct for whatever representation we use for a full-width scalar.

Our representation for $k$ will be the pair $(\mathbf{k}_{254}, k' = k - 2^{254} \cdot \mathbf{k}_{254})$. We'll use $k'$ in place of $\alpha + t_q$ for $\mathbf{z}_0$, constraining $k'$ to 254 bits so that it fits in an $\mathbb{F}_p$ element.

Then we just have to generalize the overflow check used for variable-base scalar mul in the Orchard circuit to work for $k' \in [0, 2 \cdot t_q)$ (because the maximum value of $\alpha + t_q$ is $q - 1 + t_q = 2^{254} + t_q - 1 + t_q$).

> Note: the bits $\mathbf{k}_{254..0}$ do not represent a value reduced modulo $q$, but rather a representation of the unreduced $\alpha + t_q$.

Overflow can only occur in the final step that constrains $\mathbf{z}_0 = 2 \cdot \mathbf{z}_1 + \mathbf{k}_0$, and only if $\mathbf{z}_1$ has the bit with weight $2^{253}$ set (i.e. if $\mathbf{k}_{254} = 1$). If we instead set $\mathbf{z}_0 = 2 \cdot \mathbf{z}_1 - 2^{254} \cdot \mathbf{k}_{254} + \mathbf{k}_0$, now $\mathbf{z}_0$ cannot overflow and should be equal to $k'$.

It is then sufficient to also check that $\mathbf{z}_0 + 2^{254} \cdot \mathbf{k}_{254} = \alpha + t_q$ as an integer where $\alpha \in [0, q)$.

Represent $\alpha$ as $2^{254} \cdot \boldsymbol{\alpha}_{254} + 2^{253} \cdot \boldsymbol{\alpha}_{253} + \alpha''$ where we constrain $\alpha'' \in [0, 2^{253})$ and $\boldsymbol{\alpha}_{253}$ and $\boldsymbol{\alpha}_{254}$ to boolean. For this to be a canonical representation we also need $\boldsymbol{\alpha}_{254} = 1 \implies (\boldsymbol{\alpha}_{253} = 0 \ \wedge \ \alpha'' \in [0, t_q))$.

Let $\alpha' = 2^{253} \cdot \boldsymbol{\alpha}_{253} + \alpha''$.

If $\boldsymbol{\alpha}_{254} = 1$:

- constrain $\mathbf{k}_{254} = 1$ and $\mathbf{z}_0 = \alpha' + t_q$. This cannot overflow because in this case $\alpha' \in [0, t_q)$ and so $\mathbf{z}_0 \in [0, 2 \cdot t_q)$.

If $\boldsymbol{\alpha}_{254} = 0$:

- we should have $\mathbf{k}_{254} = 1$ iff $\alpha' \in [2^{254} - t_q, 2^{254})$, i.e. witness $\mathbf{k}_{254}$ as boolean and then
  - If $\mathbf{k}_{254} = 0$ then constrain $\alpha' \notin [2^{254} - t_q, 2^{254})$.
    - This can be done by constraining either $\boldsymbol{\alpha}_{253} = 0$ or $\alpha'' + t_q \in [0, 2^{253})$. ( $\alpha'' + t_q$ cannot overflow.)
  - If $\mathbf{k}_{254} = 1$ then constrain $\alpha' \in [2^{254} - t_q, 2^{254})$.
    - This can be done by constraining $\alpha' - (2^{254} - t_q) \in [0, 2^{130})$ and $\alpha' - 2^{254} + 2^{130} \in [0, 2^{130})$.

## Overflow check constraints (general)

Represent $\alpha$ as $2^{254} \cdot \boldsymbol{\alpha}_{254} + 2^{253} \cdot \boldsymbol{\alpha}_{253} + \alpha''$ as above.

The constraints for the overflow check are:

$$
\begin{aligned}
\alpha'' &\in [0, 2^{253}) \\
\boldsymbol{\alpha}_{253} &\in \{0, 1\} \\
\boldsymbol{\alpha}_{254} &\in \{0, 1\} \\
\mathbf{k}_{254} &\in \{0, 1\} \\
\boldsymbol{\alpha}_{254} = 1 &\implies \boldsymbol{\alpha}_{253} = 0 \\
\boldsymbol{\alpha}_{254} = 1 &\implies \alpha'' \in [0, 2^{130}) \ ❁ \\
\boldsymbol{\alpha}_{254} = 1 &\implies \alpha'' + 2^{130} - t_q \in [0, 2^{130}) \ ❁ \\
\boldsymbol{\alpha}_{254} = 1 &\implies \mathbf{k}_{254} = 1 \\
\boldsymbol{\alpha}_{254} = 1 &\implies \mathbf{z}_0 = \alpha' + t_q \\
\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \boldsymbol{\alpha}_{253} = 1 \ \wedge\ \mathbf{k}_{254} = 0 &\implies \alpha'' + t_q \in [0, 2^{253}) \\
\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \mathbf{k}_{254} = 1 &\implies \alpha' - 2^{254} + t_q \in [0, 2^{130}) \ ❁ \\
\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \mathbf{k}_{254} = 1 &\implies \alpha' - 2^{254} + 2^{130} \in [0, 2^{130}) \ ❁
\end{aligned}
$$

Note that the four 130-bit constraints marked ❁ are in two pairs that occur in disjoint cases. We can therefore combine them into two 130-bit constraints using a new witness variable $u$; the other constraint always being on $u + 2^{130} - t_q$:

$$
\begin{aligned}
\boldsymbol{\alpha}_{254} = 1 &\implies u = \alpha'' \\
\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \mathbf{k}_{254} = 1 &\implies u = \alpha' - 2^{254} + t_q \\
u &\in [0, 2^{130}) \\
u + 2^{130} - t_q &\in [0, 2^{130})
\end{aligned}
$$

($u$ is unconstrained and can be witnessed as $0$ in the case $\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \mathbf{k}_{254} = 0$.)

### Cost

- 25 10-bit and one 3-bit range check, to constrain $\alpha''$ to 253 bits;
- 25 10-bit and one 3-bit range check, to constrain $\alpha'' + t_q$ to 253 bits when $\boldsymbol{\alpha}_{254} = 0 \ \wedge\ \boldsymbol{\alpha}_{253} = 1 \ \wedge\ \mathbf{k}_{254} = 0$;
- two times 13 10-bit range checks.

# Sinsemilla

## Overview

Sinsemilla is a collision-resistant hash function and commitment scheme designed to be efficient in algebraic circuit models that support lookups, such as PLONK or Halo 2.

The security properties of Sinsemilla are similar to Pedersen hashes; it is **not** designed to be used where a random oracle, PRF, or preimage-resistant hash is required. **The only claimed security property of the hash function is collision-resistance for fixed-length inputs.**

Sinsemilla is roughly 4 times less efficient than the algebraic hashes Rescue and Poseidon inside a circuit, but around 19 times more efficient than Rescue outside a circuit. Unlike either of these hashes, the collision resistance property of Sinsemilla can be proven based on cryptographic assumptions that have been well-established for at least 20 years. Sinsemilla can also be used as a computationally binding and perfectly hiding commitment scheme.

The general approach is to split the message into $k$-bit pieces, and for each piece, select from a table of $2^k$ bases in our cryptographic group. We combine the selected bases using a double-and-add algorithm. This ends up being provably as secure as a vector Pedersen hash, and makes advantageous use of the lookup facility supported by Halo 2.

## Description

This section is an outline of how Sinsemilla works: for the normative specification, refer to §5.4.1.9 Sinsemilla Hash Function in the protocol spec. The incomplete point addition operator, $\divideontimes$, that we use below is also defined there.

Let $\mathbb{G}$ be a cryptographic group of prime order $q$. We write $\mathbb{G}$ additively, with identity $\mathcal{O}$, and using $[m]P$ for scalar multiplication of $P$ by $m$.

Let $k \geq 1$ be an integer chosen based on efficiency considerations (the table size will be $2^k$). Let $n$ be an integer, fixed for each instantiation, such that messages are $kn$ bits, where $2^n \leq \frac{q-1}{2}$. We use zero-padding to the next multiple of $k$ bits if necessary.

**Setup**: Choose $Q$ and $P[0..2^k - 1]$ as $2^k + 1$ independent, verifiably random generators of $\mathbb{G}$, using a suitable hash into $\mathbb{G}$, such that none of $Q$ or $P[0..2^k - 1]$ are $\mathcal{O}$.

> In Orchard, we define $Q$ to be dependent on a domain separator $D$. The protocol specification uses $\mathcal{Q}(D)$ in place of $Q$ and $\mathcal{S}(m)$ in place of $P[m]$.

$\mathsf{Hash}(M)$:

- Split $M$ into $n$ groups of $k$ bits. Interpret each group as a $k$-bit little-endian integer $m_i$.
- let $\mathsf{Acc}_0 := Q$
- for $i$ from $0$ up to $n - 1$:
  - let $\mathsf{Acc}_{i+1} := (\mathsf{Acc}_i \mathbin{\vdots} P[m_{i+1}]) \mathbin{\vdots} \mathsf{Acc}_i$
- return $\mathsf{Acc}_n$

Let $\mathsf{ShortHash}(M)$ be the $x$-coordinate of $\mathsf{Hash}(M)$. (This assumes that $\mathbb{G}$ is a prime-order elliptic curve in short Weierstrass form, as is the case for Pallas and Vesta.)

> It is slightly more efficient to express a double-and-add $[2]A + R$ as $(A + R) + A$. We also use incomplete additions: it is shown in the Sinsemilla security argument that in the case where $\mathbb{G}$ is a prime-order short Weierstrass elliptic curve, an exceptional case for addition would lead to finding a discrete logarithm, which can be assumed to occur with negligible probability even for adversarial input.

### Use as a commitment scheme

Choose another generator $H$ independently of $Q$ and $P[0..2^k - 1]$.

The randomness $r$ for a commitment is chosen uniformly on $[0, q)$.

Let $\mathsf{Commit}_r(M) = \mathsf{Hash}(M) \mathbin{\vdots} [r]H$.

Let $\mathsf{ShortCommit}_r(M)$ be the $x$-coordinate of $\mathsf{Commit}_r(M)$. (This again assumes that $\mathbb{G}$ is a prime-order elliptic curve in short Weierstrass form.)

Note that unlike a simple Pedersen commitment, this commitment scheme (Commit or ShortCommit) is not additively homomorphic.

# Efficient implementation

The aim of the design is to optimize the number of bits that can be processed for each step of the algorithm (which requires a doubling and addition in $\mathbb{G}$) for a given table size. Using a single table of size $2^k$ group elements, we can process $k$ bits at a time.

## Incomplete addition

In each step of Sinsemilla we want to compute $A_{i+1} := (A_i \divideontimes P_i) \divideontimes A_i$. Let $R_i := A_i \divideontimes P_i$ be the intermediate result such that $A_{i+1} := A_i \divideontimes R_i$. Recalling the incomplete addition formulae:

$$
\begin{aligned}
x_3 &= \left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - x_1 - x_2 \\
y_3 &= \frac{y_1 - y_2}{x_1 - x_2} \cdot (x_1 - x_3) - y_1
\end{aligned}
$$

Let $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$. Substituting the coordinates for each of the incomplete additions in turn, and rearranging, we get

$$
\begin{aligned}
\lambda_{1,i} &= \frac{y_{A,i} - y_{P,i}}{x_{A,i} - x_{P,i}} \\
&\implies y_{A,i} - y_{P,i} = \lambda_{1,i} \cdot (x_{A,i} - x_{P,i}) \\
&\implies y_{P,i} = y_{A,i} - \lambda_{1,i} \cdot (x_{A,i} - x_{P,i}) \\
x_{R,i} &= \lambda_{1,i}^2 - x_{A,i} - x_{P,i} \\
y_{R,i} &= \lambda_{1,i} \cdot (x_{A,i} - x_{R,i}) - y_{A,i}
\end{aligned}
$$

and

$$
\begin{aligned}
\lambda_{2,i} &= \frac{y_{A,i} - y_{R,i}}{x_{A,i} - x_{R,i}} \\
&\implies y_{A,i} - y_{R,i} = \lambda_{2,i} \cdot (x_{A,i} - x_{R,i}) \\
&\implies y_{A,i} - (\lambda_{1,i} \cdot (x_{A,i} - x_{R,i}) - y_{A,i}) = \lambda_{2,i} \cdot (x_{A,i} - x_{R,i}) \\
&\implies 2 \cdot y_{A,i} = (\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - x_{R,i}) \\
x_{A,i+1} &= \lambda_{2,i}^2 - x_{A,i} - x_{R,i} \\
y_{A,i+1} &= \lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) - y_{A,i}.
\end{aligned}
$$

### Constraint program

Let $\mathcal{P} = \{(j, x_{P[j]}, y_{P[j]}) \text{ for } j \in \{0..2^k - 1\}\}$.

Input: $m_{1..=n}$. (The message words are 1-indexed here, as in the protocol spec, but we start the loop from $i = 0$ so that $(x_{A,i}, y_{A,i})$ corresponds to $\mathsf{Acc}_i$ in the protocol spec.)

Output: $(x_{A,n}, y_{A,n})$.

- $(x_{A,0}, y_{A,0}) = Q$
- for $i$ from $0$ up to $n - 1$:
  - $y_{P,i} = y_{A,i} - \lambda_{1,i} \cdot (x_{A,i} - x_{P,i})$
  - $x_{R,i} = \lambda_{1,i}^2 - x_{A,i} - x_{P,i}$
  - $2 \cdot y_{A,i} = (\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - x_{R,i})$
  - $(m_{i+1}, x_{P,i}, y_{P,i}) \in \mathcal{P}$
  - $\lambda_{2,i}^2 = x_{A,i+1} + x_{R,i} + x_{A,i}$
  - $\lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) = y_{A,i} + y_{A,i+1}$

## PLONK / Halo 2 constraints

### Message decomposition

We have an $n$-bit message $m = m_1 + 2^k m_2 + ... + 2^{k \cdot (n-1)} m_n$. (Note that the message words are 1-indexed as in the protocol spec.)

Initialise the running sum $z_0 = \alpha$ and define $z_{i+1} := \frac{z_i - m_{i+1}}{2^K}$. We will end up with $z_n = 0$.

Rearranging gives us an expression for each word of the original message $m_{i+1} = z_i - 2^k \cdot z_{i+1}$, which we can look up in the table. We position $z_i$ and $z_{i+1}$ in adjacent rows of the same column, so we can sequentially apply the constraint across the entire message.

In other words, $z_{n-i} = \sum_{h=0}^{i-1} 2^{kh} \cdot m_{h+1}$.

> For a little-endian decomposition as used here, the running sum is initialized to the scalar and ends at 0. For a big-endian decomposition as used in variable-base scalar multiplication, the running sum would start at 0 and end with recovering the original scalar.

### Efficient packing

The running sum only applies to message words within a single field element. That means if $n \geq$ `PrimeField::NUM_BITS` then we will need several disjoint running sums. A longer message can be constructed by splitting the message words across several field elements, and then running several instances of the constraints below.

The expression for $m_{i+1}$ above requires $n + 1$ rows in the $z_i$ column, leaving a one-row gap in adjacent columns and making $\mathsf{Acc}_i$ tricker to accumulate. In order to support chaining multiple field elements without a gap, we use a slightly more complicated expression for $m_{i+1}$ that includes a selector:

$$m_{i+1} = z_i - 2^k \cdot q_{run,i} \cdot z_{i+1}$$

This effectively forces $\mathbf{z}_n$ to zero for the last step of each element, which allows the cell that would have been $\mathbf{z}_n$ to be used to reinitialize the running sum for the next element.

With this sorted out, the incomplete addition accumulator can eliminate $y_{A,i}$ almost entirely, by substituting for $x$ and $\lambda$ values in the current and next rows. The two exceptions are at the start of Sinsemilla (where we need to constrain the accumulator to have initial value $Q$), and the end (where we need to witness $y_{A,n}$ for use outside of Sinsemilla).

### Selectors

We need a total of four logical selectors to:

- Control the Sinsemilla gate and lookup.
- Distinguish between the last message word in a running sum and its earlier words.
- Mark the start of Sinsemilla.
- Mark the end of Sinsemilla.

We use regular selector columns for the Sinsemilla gate selector $q_{S1}$ and Sinsemilla start selector $q_{S4}$. The other two selectors are synthesized from a single fixed column $q_{S2}$ as follows:

$$q_{S3} = q_{S2} \cdot (q_{S2} - 1)$$
$$q_{run} = q_{S2} - q_{S3}$$

| $q_{S2}$ | $q_{S3}$ | $q_{run}$ |
|---|---|---|
| $0$ | $0$ | $0$ |
| $1$ | $0$ | $1$ |
| $2$ | $2$ | $0$ |

We set $q_{S2}$ to $1$ on most Sinsemilla rows, and $0$ for the last step of each element, except for the last element where it is set to $2$. We can then use $q_{S3}$ to toggle between constraining the substituted $y_{A,i+1}$ on adjacent rows, and the witnessed $y_{A,n}$ at the end of Sinsemilla:

$$\lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) = y_{A,i} + \frac{2 - q_{S3}}{2} \cdot y_{A,i+1} + \frac{q_{S3}}{2} \cdot y_{A,n}$$

## Generator lookup table

The Sinsemilla circuit makes use of $2^{10}$ pre-computed random generators. These are loaded into a lookup table:

| $table_{idx}$ | $table_x$ | $table_y$ |
|---|---|---|
| $0$ | $x_{P[0]}$ | $y_{P[0]}$ |
| $1$ | $x_{P[1]}$ | $y_{P[1]}$ |
| $2$ | $x_{P[2]}$ | $y_{P[2]}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $2^{10} - 1$ | $x_{P[2^{10}-1]}$ | $y_{P[2^{10}-1]}$ |

## Layout

| Step | $x_A$ | $x_P$ | $bits$ | $\lambda_1$ | $\lambda_2$ | $q_{S1}$ | $q_{S2}$ | $q_{S4}$ | fixed_y_Q |
|---|---|---|---|---|---|---|---|---|---|
| $0$ | $x_Q$ | $x_{P[m_1]}$ | $z_0$ | $\lambda_{1,0}$ | $\lambda_{2,0}$ | $1$ | $1$ | $1$ | $y_Q$ |
| $1$ | $x_{A,1}$ | $x_{P[m_2]}$ | $z_1$ | $\lambda_{1,1}$ | $\lambda_{2,1}$ | $1$ | $1$ | $0$ | $0$ |
| $2$ | $x_{A,2}$ | $x_{P[m_3]}$ | $z_2$ | $\lambda_{1,2}$ | $\lambda_{2,2}$ | $1$ | $1$ | $0$ | $0$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $1$ | $1$ | $0$ | $0$ |
| $n-1$ | $x_{A,n-1}$ | $x_{P[m_n]}$ | $z_{n-1}$ | $\lambda_{1,n-1}$ | $\lambda_{2,n-1}$ | $1$ | $0$ | $0$ | $0$ |
| $0'$ | $x'_{A,0}$ | $x_{P[m'_1]}$ | $z'_0$ | $\lambda'_{1,0}$ | $\lambda'_{2,0}$ | $1$ | $1$ | $0$ | $0$ |
| $1'$ | $x'_{A,1}$ | $x_{P[m'_2]}$ | $z'_1$ | $\lambda'_{1,1}$ | $\lambda'_{2,1}$ | $1$ | $1$ | $0$ | $0$ |
| $2'$ | $x'_{A,2}$ | $x_{P[m'_3]}$ | $z'_2$ | $\lambda'_{1,2}$ | $\lambda'_{2,2}$ | $1$ | $1$ | $0$ | $0$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $1$ | $1$ | $0$ | $0$ |
| $n-1'$ | $x'_{A,n-1}$ | $x_{P[m'_n]}$ | $z'_{n-1}$ | $\lambda'_{1,n-1}$ | $\lambda'_{2,n-1}$ | $1$ | $2$ | $0$ | $0$ |
| $n'$ | $x'_{A,n}$ | | | $y_{A,n}$ | | $0$ | $0$ | $0$ | $0$ |

$x_Q$, $z_0$, $z'_0$, etc. are copied in using equality constraints.

### Optimized Sinsemilla gate

$$\begin{aligned}\text{For } i \in [0, n), \text{ let} \quad x_{R,i} &= \lambda_{1,i}^2 - x_{A,i} - x_{P,i} \\ Y_{A,i} &= (\lambda_{1,i} + \lambda_{2,i}) \cdot (x_{A,i} - x_{R,i}) \\ y_{P,i} &= Y_{A,i}/2 - \lambda_{1,i} \cdot (x_{A,i} - x_{P,i}) \\ m_{i+1} &= z_i - q_{run,i} \cdot z_{i+1} \cdot 2^k \\ q_{run} &= q_{S2} - q_{S3} \\ q_{S3} &= q_{S2} \cdot (q_{S2} - 1)\end{aligned}$$

The Halo 2 circuit API can automatically substitute $y_{P,i}$, $x_{R,i}$, $Y_{A,i}$, and $Y_{A,i+1}$, so we don't need to do that manually.

- $x_{A,0} = x_Q$
- $2 \cdot y_Q = Y_{A,0}$
- for $i$ from $0$ up to $n - 1$:
  - $(m_{i+1},\, x_{P,i},\, y_{P,i}) \in \mathcal{P}$
  - $\lambda_{2,i}^2 = x_{A,i+1} + x_{R,i} + x_{A,i}$
  - $4 \cdot \lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) = 2 \cdot Y_{A,i} + (2 - q_{S3}) \cdot Y_{A,i+1} + 2q_{S3} \cdot y_{A,n}$

Note that each term of the last constraint is multiplied by $4$ relative to the constraint program given earlier. This is a small optimization that avoids divisions by $2$.

By gating the lookup expression on $q_{S1}$, we avoid the need to fill in unused cells with dummy values to pass the lookup argument. The optimized lookup value (using a default index of $0$) is:

$$\begin{pmatrix} q_{S1} \cdot m_{i+1}, \\ q_{S1} \cdot x_{P,i} + (1 - q_{S1}) \cdot x_{P,0}, \\ q_{S1} \cdot y_{P,i} + (1 - q_{S1}) \cdot y_{P,0} \end{pmatrix}$$

This increases the degree of the lookup argument to $6$.

| Degree | Constraint |
|---|---|
| $4$ | $q_{S4} \cdot (2 \cdot y_Q - Y_{A,0}) = 0$ |
| $6$ | $q_{S1,i} \Rightarrow (m_{i+1},\, x_{P,i},\, y_{P,i}) \in \mathcal{P}$ |
| $3$ | $q_{S1,i} \cdot \left(\lambda_{2,i}^2 - (x_{A,i+1} + x_{R,i} + x_{A,i})\right)$ |
| $5$ | $q_{S1,i} \cdot (4 \cdot \lambda_{2,i} \cdot (x_{A,i} - x_{A,i+1}) - (2 \cdot Y_{A,i} + (2 - q_{S3,i}) \cdot Y_{A,i+1} + 2 \cdot q_{S3,i} \cdot y_{A,n}$ |

# MerkleCRH

## Message decomposition

SinsemillaHash is used in the MerkleCRH$^{\mathsf{Orchard}}$ hash function. The input to SinsemillaHash is:

$$l\star \,||\, \mathsf{left}\star \,||\, \mathsf{right}\star,$$

where:

- $l\star = \mathsf{I2LEBSP}_{10}(l) = \mathsf{I2LEBSP}_{10}(\mathsf{MerkleDepth}^{\mathsf{Orchard}} - 1 - \mathsf{layer})$,
- $\mathsf{left}\star = \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(\mathsf{left})$,
- $\mathsf{right}\star = \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(\mathsf{right})$,

with $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}} = 255$. $\mathsf{left}\star$ and $\mathsf{right}\star$ are allowed to be non-canonical $255$-bit encodings of $\mathsf{left}$ and $\mathsf{right}$.

Sinsemilla operates on multiples of 10 bits, so we start by decomposing the message into chunks:

$$\begin{aligned}
l\star &= a_0 \\
\mathsf{left}\star &= a_1 \,||\, b_0 \,||\, b_1 \\
&= (\text{bits } 0..=239 \text{ of } \mathsf{left}) \,||\, (\text{bits } 240..=249 \text{ of } \mathsf{left}) \,||\, (\text{bits } 250..=254 \text{ of } \mathsf{left}) \\
\mathsf{right}\star &= b_2 \,||\, c \\
&= (\text{bits } 0..=4 \text{ of } \mathsf{right}) \,||\, (\text{bits } 5..=254 \text{ of } \mathsf{right})
\end{aligned}$$

Then we recompose the chunks into `MessagePiece`s:

| Length (bits) | Piece |
|---|---|
| 250 | $a = a_0 \,||\, a_1$ |
| 20 | $b = b_0 \,||\, b_1 \,||\, b_2$ |
| 250 | $c$ |

Each message piece is constrained by SinsemillaHash to its stated length. Additionally, $\mathsf{left}$ and $\mathsf{right}$ are witnessed as field elements, so we know that they are canonical. However, we need additional constraints to enforce that the chunks are the correct bit lengths (or else they could overlap in the decompositions and allow the prover to witness an arbitrary SinsemillaHash message).

Some of these constraints can be implemented with reusable circuit gadgets. We define a custom gate controlled by the selector $q_{\mathsf{decompose}}$ to hold the remaining constraints.

## Bit length constraints

Chunk $c$ is directly constrained by Sinsemilla. We constrain the remaining chunks with the following constraints:

$a_0, a_1$

$z_{1,a}$, the index-1 running sum output of $\mathsf{SinsemillaHash}(a)$, is copied into the gate. $z_{1,a}$ has been constrained by $\mathsf{SinsemillaHash}$ to be $240$ bits, and is precisely $a_1$. We recover chunk $a_0$ using $a, z_{1,a}$ :

$$\begin{aligned} z_{1,a} &= \frac{a - a_0}{2^{10}} \\ &= a_1 \\ \implies a_0 &= a - z_{1,a} \cdot 2^{10}. \end{aligned}$$

$b_0, b_1, b_2$

$z_{1,b}$, the index-1 running sum output of $\mathsf{SinsemillaHash}(b)$, is copied into the gate. $z_{1,b}$ has been constrained by $\mathsf{SinsemillaHash}$ to be $10$ bits. We witness the subpieces $b_1, b_2$ outside this gate, and constrain them each to be $5$ bits. Inside the gate, we check that

$$b_1 + 2^5 \cdot b_2 = z_{1,b}.$$

We also recover the subpiece $b_0$ using $(b, z_{1,b})$:

$$\begin{aligned} z_{1,b} &= \frac{b - b_{0..=10}}{2^{10}} \\ \implies b_0 &= b - (z_{1,b} \cdot 2^{10}). \end{aligned}$$

### Constraints

| Degree | Constraint |
|---|---|
| | $\texttt{short\_lookup\_range\_check}(b_1, 5)$ |
| | $\texttt{short\_lookup\_range\_check}(b_2, 5)$ |
| 2 | $q_{\mathsf{decompose}} \cdot (z_{1,b} - (b_1 + b_2 \cdot 2^5)) = 0$ |

where $\texttt{short\_lookup\_range\_check}()$ is a short lookup range check.

## Decomposition constraints

We have now derived or witnessed every subpiece, and range-constrained every subpiece:

- $a_0$ ($10$ bits), derived as $a_0 = a - 2^{10} \cdot z_{1,a}$;
- $a_1$ ($240$ bits), equal to $z_{1,a}$;
- $b_0$ ($10$ bits), derived as $b_0 = b - 2^{10} \cdot z_{1,b}$;
- $b_1$ ($5$ bits) is witnessed and constrained outside the gate;
- $b_2$ ($5$ bits) is witnessed and constrained outside the gate;
- $c$ ($250$ bits) is witnessed and constrained outside the gate.
- $b_1 + 2^5 \cdot b_2$ is constrained to equal $z_{1,b}$.

We can now use them to reconstruct the original field element inputs:

$$
\begin{aligned}
l &= a_0 \\
\mathsf{left} &= a_1 + 2^{240} \cdot b_0 + 2^{250} \cdot b_1 \\
\mathsf{right} &= b_2 + 2^5 \cdot c
\end{aligned}
$$

| Degree | Constraint |
|---|---|
| 2 | $q_{\mathsf{decompose}} \cdot (a_0 - l) = 0$ |
| 2 | $q_{\mathsf{decompose}} \cdot (a_1 + (b_0 + b_1 \cdot 2^{10}) \cdot 2^{240} - \mathsf{left}) = 0$ |
| 2 | $q_{\mathsf{decompose}} \cdot (b_2 + c \cdot 2^5 - \mathsf{right}) = 0$ |

## Region layout

| | | | | | $q_{\mathsf{decompose}}$ |
|---|---|---|---|---|---|
| $a$ | $b$ | $c$ | $\mathsf{left}$ | $\mathsf{right}$ | $1$ |
| $z_{1,a}$ | $z_{1,b}$ | $b_1$ | $b_2$ | $l$ | $0$ |

## Circuit components

The Orchard circuit spans $10$ advice columns while the **Sinsemilla** chip only uses $5$ advice columns. We distribute the path hashing evenly across two **Sinsemilla** chips to make better use of the available circuit area. Since the output from the previous layer hash is copied into the next layer hash, we maintain continuity even when moving from one chip to the other.

# Decomposition

Given a field element $\alpha$, these gadgets decompose it into $W$ $K$-bit windows

$$\alpha = k_0 + 2^K \cdot k_1 + 2^{2K} \cdot k_2 + \cdots + 2^{(W-1)K} \cdot k_{W-1}$$

where each $k_i$ a $K$-bit value.

This is done using a running sum $z_i, i \in [0..W)$. We initialize the running sum $z_0 = \alpha$, and compute subsequent terms $z_{i+1} = \frac{z_i - k_i}{2^K}$. This gives us:

$$\begin{aligned}
z_0 &= \alpha \\
&= k_0 + 2^K \cdot k_1 + 2^{2K} \cdot k_2 + 2^{3K} \cdot k_3 + \cdots, \\
z_1 &= (z_0 - k_0)/2^K \\
&= k_1 + 2^K \cdot k_2 + 2^{2K} \cdot k_3 + \cdots, \\
z_2 &= (z_1 - k_1)/2^K \\
&= k_2 + 2^K \cdot k_3 + \cdots, \\
&\vdots \\
&\downarrow \text{(in strict mode)} \\
z_W &= (z_{W-1} - k_{W-1})/2^K \\
&= 0 \text{ (because } z_{W-1} = k_{W-1})
\end{aligned}$$

### Strict mode

Strict mode constrains the running sum output $z_W$ to be zero, thus range-constraining the field element to be within $W \cdot K$ bits.

In strict mode, we are also assured that $z_{W-1} = k_{W-1}$ gives us the last window in the decomposition.

# Lookup decomposition

This gadget makes use of a $K$-bit lookup table to decompose a field element $\alpha$ into $K$-bit words. Each $K$-bit word $k_i = z_i - 2^K \cdot z_{i+1}$ is range-constrained by a lookup in the $K$-bit table.

The region layout for the lookup decomposition uses a single advice column $z$, and two selectors $q_{lookup}$ and $q_{running}$.

| $z$ | $q_{lookup}$ | $q_{running}$ |
|---|---|---|
| $z_0$ | $1$ | $1$ |
| $z_1$ | $1$ | $1$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $z_{n-1}$ | $1$ | $1$ |
| $z_n$ | $0$ | $0$ |

## Short range check

Using two $K$-bit lookups, we can range-constrain a field element $\alpha$ to be $n$ bits, where $n \leq K$. To do this:

1. Constrain $0 \leq \alpha < 2^K$ to be within $K$ bits using a $K$-bit lookup.
2. Constrain $0 \leq \alpha \cdot 2^{K-n} < 2^K$ to be within $K$ bits using a $K$-bit lookup.

The short variant of the lookup decomposition introduces a $q_{bitshift}$ selector. The same advice column $z$ has here been renamed to **word** for clarity:

| word | $q_{lookup}$ | $q_{running}$ | $q_{bitshift}$ |
|---|---|---|---|
| $\alpha$ | $1$ | $0$ | $0$ |
| $\alpha'$ | $1$ | $0$ | $1$ |
| $2^{K-n}$ | $0$ | $0$ | $0$ |

where $\alpha' = \alpha \cdot 2^{K-n}$. Note that $2^{K-n}$ is assigned to a fixed column at keygen, and copied in at proving time. This is used in the gate enabled by the $q_{bitshift}$ selector to check that $\alpha$ was shifted correctly:

| Degree | Constraint |
|---|---|
| $2$ | $q_{bitshift} \cdot ((\alpha \cdot 2^{K-n}) - \alpha')$ |

### Combined lookup expression

Since the lookup decomposition and its short variant both make use of the same lookup table, we combine their lookup input expressions into a single one:

$$q_{lookup} \cdot \big(q_{running} \cdot (z_i - 2^K \cdot z_{i+1}) + (1 - q_{running}) \cdot \text{word}\big)$$

where $z_i$ and **word** are the same cell (but distinguished here for clarity of usage).

## Short range decomposition

For a short range (for instance, $[0, \texttt{range})$ where $\texttt{range} \leq 8$), we can range-constrain each word using a degree-$\texttt{range}$ polynomial constraint instead of a lookup:

$$\texttt{range\_check}(word, range) = \texttt{word} \cdot (1 - \texttt{word}) \cdots (\texttt{range} - 1 - \texttt{word}).$$

# SHA-256

## Specification

SHA-256 is specified in NIST FIPS PUB 180-4.

Unlike the specification, we use $\boxplus$ for addition modulo $2^{32}$, and $+$ for field addition. $\oplus$ is used for XOR.

## Gadget interface

SHA-256 maintains state in eight 32-bit variables. It processes input as 512-bit blocks, but internally splits these blocks into 32-bit chunks. We therefore designed the SHA-256 gadget to consume input in 32-bit chunks.

## Chip instructions

The SHA-256 gadget requires a chip with the following instructions:

```
const BLOCK_SIZE: usize = 16;
const DIGEST_SIZE: usize = 8;

pub trait Sha256Instructions: Chip {
    /// Variable representing the SHA-256 internal state.
    type State: Clone + fmt::Debug;
    /// Variable representing a 32-bit word of the input block to the SHA-256
compression
    /// function.
    type BlockWord: Copy + fmt::Debug;

    /// Places the SHA-256 IV in the circuit, returning the initial state
variable.
    fn initialization_vector(layouter: &mut impl Layouter<Self>) ->
Result<Self::State, Error>;

    /// Starting from the given initial state, processes a block of input and
returns the
    /// final state.
    fn compress(
        layouter: &mut impl Layouter<Self>,
        initial_state: &Self::State,
        input: [Self::BlockWord; BLOCK_SIZE],
    ) -> Result<Self::State, Error>;

    /// Converts the given state into a message digest.
    fn digest(
        layouter: &mut impl Layouter<Self>,
        state: &Self::State,
    ) -> Result<[Self::BlockWord; DIGEST_SIZE], Error>;
}
```

TODO: Add instruction for computing padding.

This set of instructions was chosen to strike a balance between the reusability of the instructions, and the scope for chips to internally optimise them. In particular, we considered splitting the compression function into its constituent parts (Ch, Maj etc), and providing a compression function gadget that implemented the round logic. However, this would prevent chips from using relative references between the various parts of a compression round. Having an instruction that implements all compression rounds is also similar to the Intel SHA extensions, which provide an instruction that performs multiple compression rounds.

# 16-bit table chip for SHA-256

This chip implementation is based around a single 16-bit lookup table. It requires a minimum of $2^{16}$ circuit rows, and is therefore suitable for use in larger circuits.

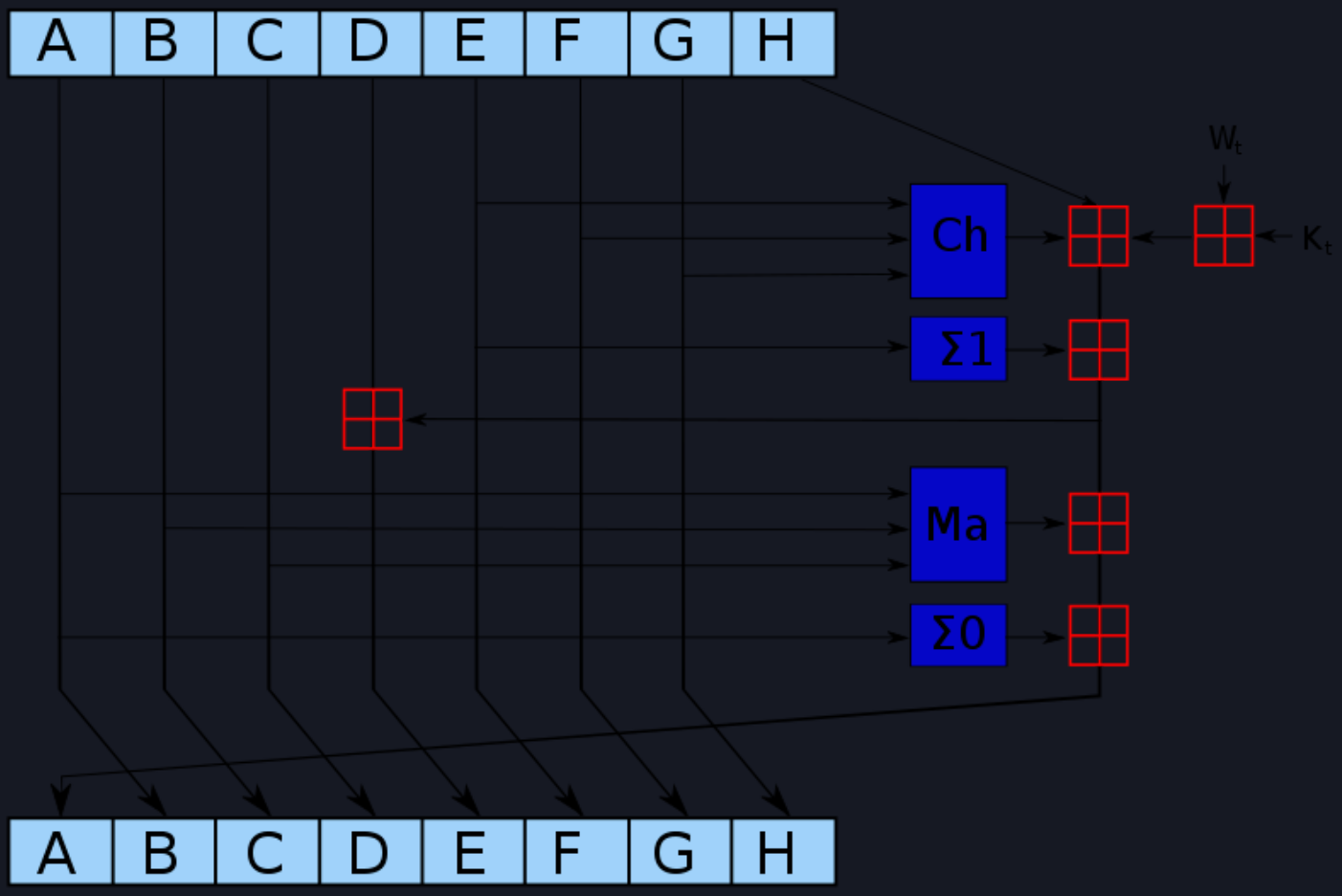
We target a maximum constraint degree of $9$. That will allow us to handle constraining carries and "small pieces" to a range of up to $\{0..7\}$ in one row.

## Compression round

There are $64$ compression rounds. Each round takes 32-bit values $A, B, C, D, E, F, G, H$ as input, and performs the following operations:

$$
\begin{aligned}
Ch(E, F, G) &= (E \wedge F) \oplus (\neg E \wedge G) \\
Maj(A, B, C) &= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \\
&= count(A, B, C) \geq 2 \\
\Sigma_0(A) &= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \\
\Sigma_1(E) &= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25) \\
H' &= H + Ch(E, F, G) + \Sigma_1(E) + K_t + W_t \\
E_{new} &= reduce_6(H' + D) \\
A_{new} &= reduce_7(H' + Maj(A, B, C) + \Sigma_0(A))
\end{aligned}
$$

where $reduce_i$ must handle a carry $0 \leq carry < i$.



Define **spread** as a table mapping a $16$-bit input to an output interleaved with zero bits. We do not require a separate table for range checks because **spread** can be used.

### Modular addition

To implement addition modulo $2^{32}$, we note that this is equivalent to adding the operands using field addition, and then masking away all but the lowest 32 bits of the result. For example, if we have two operands $a$ and $b$:

$$a \boxplus b = c,$$

we decompose each operand (along with the result) into 16-bit chunks:

$$(a_L : \mathbb{Z}_{2^{16}}, a_H : \mathbb{Z}_{2^{16}}) \boxplus (b_L : \mathbb{Z}_{2^{16}}, b_H : \mathbb{Z}_{2^{16}}) = (c_L : \mathbb{Z}_{2^{16}}, c_H : \mathbb{Z}_{2^{16}}),$$

and then reformulate the constraint using field addition:

$$\textsf{carry} \cdot 2^{32} + c_H \cdot 2^{16} + c_L = (a_H + b_H) \cdot 2^{16} + a_L + b_L.$$

More generally, any bit-decomposition of the output can be used, not just a decomposition into 16-bit chunks. Note that this correctly handles the carry from $a_L + b_L$.

This constraint requires that each chunk is correctly range-checked (or else an assignment could overflow the field).

- The operand and result chunks can be constrained using **spread**, by looking up each chunk in the "dense" column within a subset of the table. This way we additionally get the "spread" form of the output for free; in particular this is true for the output of the bottom-right $\boxplus$ which becomes $A_{new}$, and the output of the leftmost $\boxplus$ which becomes $E_{new}$. We will use this below to optimize $Maj$ and $Ch$.
- **carry** must be constrained to the precise range of allowed carry values for the number of operands. We do this with a small range constraint.

### Maj function

$Maj$ can be done in $4$ lookups: $2$ **spread** $* 2$ chunks

- As mentioned above, after the first round we already have $A$ in spread form $A'$. Similarly, $B$ and $C$ are equal to the $A$ and $B$ respectively of the previous round, and therefore in the steady state we already have them in spread form $B'$ and $C'$. In fact we can also

assume we have them in spread form in the first round, either from the fixed IV or from the use of **spread** to reduce the output of the feedforward in the previous block.
- Add the spread forms in the field: $M' = A' + B' + C'$;
  - We can add them as $32$-bit words or in pieces; it's equivalent
- Witness the compressed even bits $M_i^{even}$ and the compressed odd bits $M_i^{odd}$ for $i = \{0..1\}$;
- Constrain $M' = \mathbf{spread}(M_0^{even}) + 2 \cdot \mathbf{spread}(M_0^{odd}) + 2^{32} \cdot \mathbf{spread}(M_1^{even}) + 2^{33} \cdot \mathbf{spread}(M_1^{odd})$, where $M_i^{odd}$ is the $Maj$ function output.

Note: by "even" bits we mean the bits of weight an even-power of $2$, i.e. of weight $2^0, 2^2, \ldots$. Similarly by "odd" bits we mean the bits of weight an odd-power of $2$.

### Ch function

TODO: can probably be optimized to $4$ or $5$ lookups using an additional table.

$Ch$ can be done in $8$ lookups: $4$ $\mathbf{spread} * 2$ chunks

- As mentioned above, after the first round we already have $E$ in spread form $E'$. Similarly, $F$ and $G$ are equal to the $E$ and $F$ respectively of the previous round, and therefore in the steady state we already have them in spread form $F'$ and $G'$. In fact we can also assume we have them in spread form in the first round, either from the fixed IV or from the use of **spread** to reduce the output of the feedforward in the previous block.
- Calculate $P' = E' + F'$ and $Q' = (evens - E') + G'$, where $evens = \mathbf{spread}(2^{32} - 1)$.
  - We can add them as $32$-bit words or in pieces; it's equivalent.
  - $evens - E'$ works to compute the spread of $\neg E$ even though negation and **spread** do not commute in general. It works because each spread bit in $E'$ is subtracted from $1$, so there are no borrows.
- Witness $P_i^{even}, P_i^{odd}, Q_i^{even}, Q_i^{odd}$ such that $P' = \mathbf{spread}(P_0^{even}) + 2 \cdot \mathbf{spread}(P_0^{odd}) + 2^{32} \cdot \mathbf{spread}(P_1^{even}) + 2^{33} \cdot \mathbf{spread}(P_1^{odd})$, and similarly for $Q'$.
- $\{P_i^{odd} + Q_i^{odd}\}_{i=0..1}$ is the $Ch$ function output.

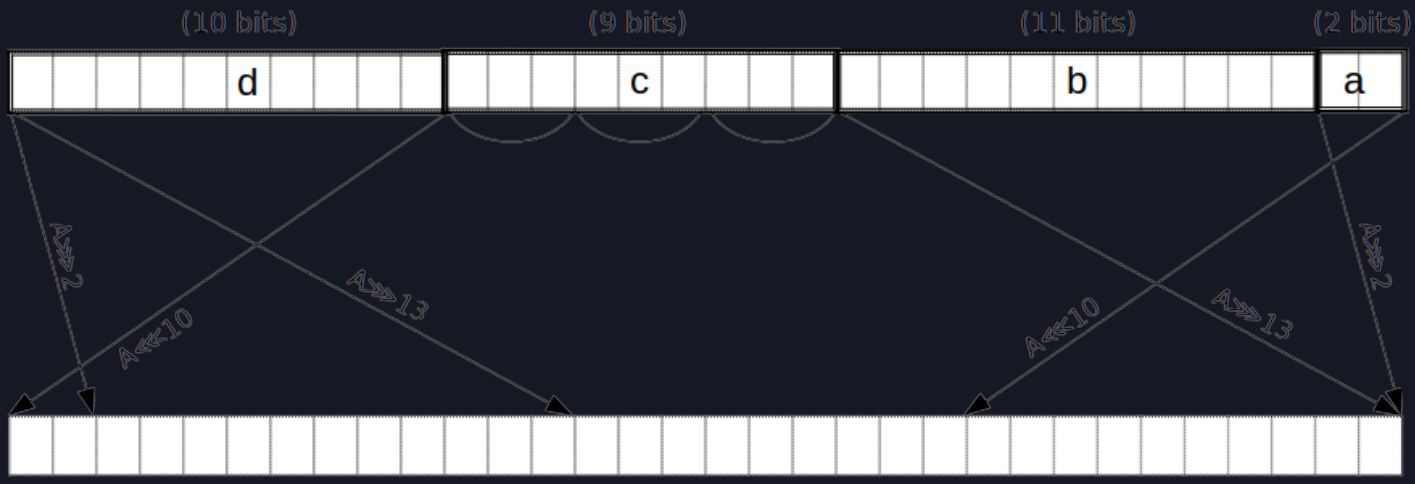
### Σ_0 function

$\Sigma_0(A)$ can be done in $6$ lookups.

To achieve this we first split $A$ into pieces $(a, b, c, d)$, of lengths $(2, 11, 9, 10)$ bits respectively counting from the little end. At the same time we obtain the spread forms of these pieces. This can all be done in two PLONK rows, because the $10$ and $11$-bit pieces can be handled using **spread** lookups, and the $9$-bit piece can be split into $3 * 3$-bit subpieces. The latter and the remaining $2$-bit piece can be range-checked by polynomial constraints in parallel with the two lookups, two small pieces in each row. The spread forms of these small pieces are found by interpolation.

Note that the splitting into pieces can be combined with the reduction of $A_{new}$, i.e. no extra lookups are needed for the latter. In the last round we reduce $A_{new}$ after adding the feedforward (requiring a carry of up to $7$ which is fine).

$(A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$ is equivalent to $(A \ggg 2) \oplus (A \ggg 13) \oplus (A \lll 10)$ :




Then, using $4$ more **spread** lookups we obtain the result as the even bits of a linear combination of the pieces:

$$
\begin{array}{rcccccccc}
 & (a & || & d & || & c & || & b) & \oplus \\
 & (b & || & a & || & d & || & c) & \oplus \\
 & (c & || & b & || & a & || & d) & \\
 & & & & \Downarrow & & & & \\
R' = & 4^{30}a & + & 4^{20}d & + & 4^{11}c & + & b & + \\
 & 4^{21}b & + & 4^{19}a & + & 4^{9}d & + & c & + \\
 & 4^{23}c & + & 4^{12}b & + & 4^{10}a & + & d &
\end{array}
$$

That is, we witness the compressed even bits $R_i^{even}$ and the compressed odd bits $R_i^{odd}$, and constrain

$$R' = \mathbf{spread}(R_0^{even}) + 2 \cdot \mathbf{spread}(R_0^{odd}) + 2^{32} \cdot \mathbf{spread}(R_1^{even}) + 2^{33} \cdot \mathbf{spread}(R_1^{odd})$$

where $\{R_i^{even}\}_{i=0..1}$ is the $\Sigma_0$ function output.

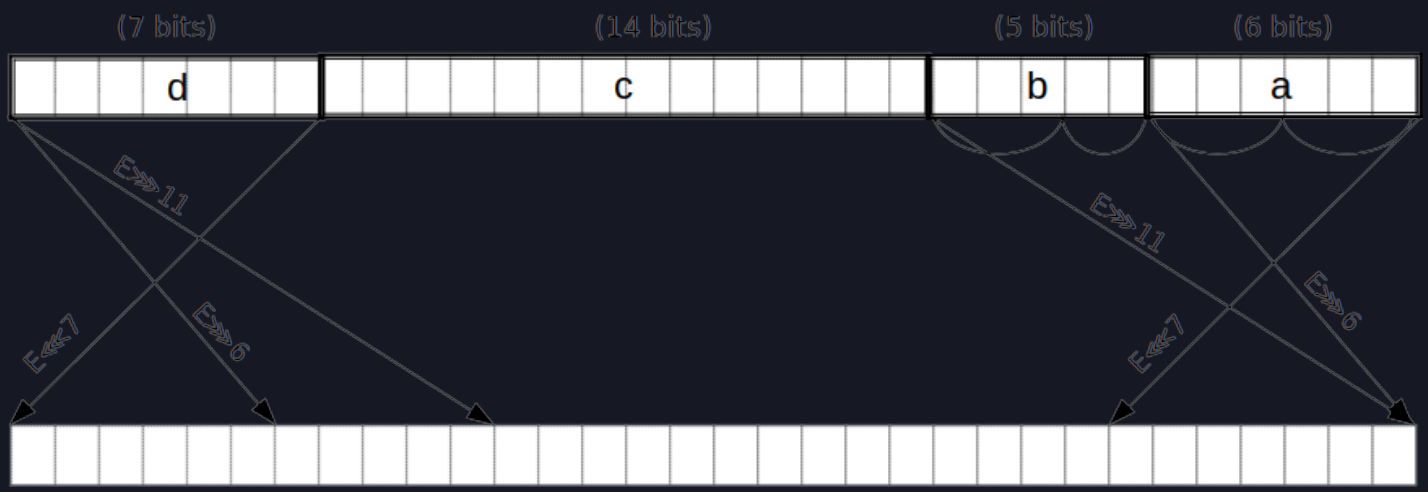
## Σ_1 function

$\Sigma_1(E)$ can be done in $6$ lookups.

To achieve this we first split $E$ into pieces $(a, b, c, d)$, of lengths $(6, 5, 14, 7)$ bits respectively counting from the little end. At the same time we obtain the spread forms of these pieces. This can all be done in two PLONK rows, because the $7$ and $14$-bit pieces can be handled using **spread** lookups, the $5$-bit piece can be split into $3$ and $2$-bit subpieces, and the $6$-bit piece can be split into $2 * 3$-bit subpieces. The four small pieces can be range-checked by polynomial constraints in parallel with the two lookups, two small pieces in each row. The spread forms of these small pieces are found by interpolation.

Note that the splitting into pieces can be combined with the reduction of $E_{new}$, i.e. no extra lookups are needed for the latter. In the last round we reduce $E_{new}$ after adding the feedforward (requiring a carry of up to $6$ which is fine).

$(E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$ is equivalent to $(E \ggg 6) \oplus (E \ggg 11) \oplus (E \lll 7)$.




Then, using $4$ more **spread** lookups we obtain the result as the even bits of a linear combination of the pieces, in the same way we did for $\Sigma_0$:

$$
\begin{array}{rccccccccc}
 & (a & || & d & || & c & || & b) & \oplus \\
 & (b & || & a & || & d & || & c) & \oplus \\
 & (c & || & b & || & a & || & d) & \\
 & & & & \Downarrow & & & & \\
R' = & 4^{26}a & + & 4^{19}d & + & 4^{5}c & + & b & + \\
 & 4^{27}b & + & 4^{21}a & + & 4^{14}d & + & c & + \\
 & 4^{18}c & + & 4^{13}b & + & 4^{7}a & + & d &
\end{array}
$$

That is, we witness the compressed even bits $R_i^{even}$ and the compressed odd bits $R_i^{odd}$, and constrain

$$R' = \texttt{spread}(R_0^{even}) + 2 \cdot \texttt{spread}(R_0^{odd}) + 2^{32} \cdot \texttt{spread}(R_1^{even}) + 2^{33} \cdot \texttt{spread}(R_1^{odd})$$

where $\{R_i^{even}\}_{i=0..1}$ is the $\Sigma_1$ function output.

## Block decomposition

For each block $M \in \{0,1\}^{512}$ of the padded message, $64$ words of $32$ bits each are constructed as follows:

- The first $16$ are obtained by splitting $M$ into $32$-bit blocks

$$M = W_0 || W_1 || \cdots || W_{14} || W_{15};$$

- The remaining $48$ words are constructed using the formula:

$$W_i = \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16},$$

  for $16 \leq i < 64$.

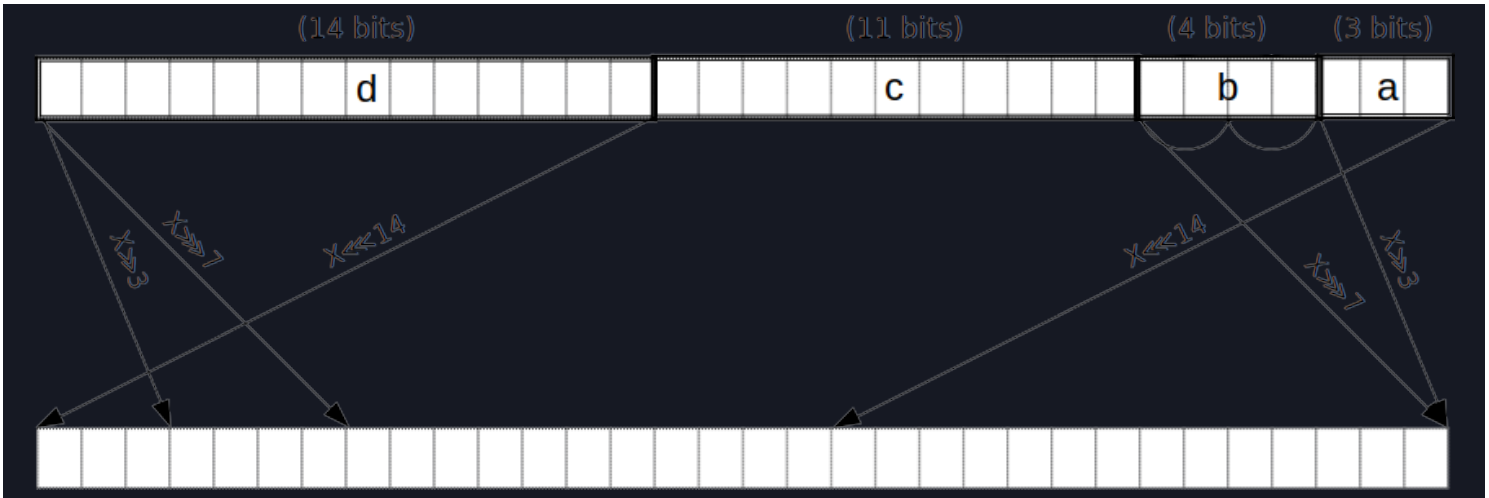
> Note: $0$-based numbering is used for the $W$ word indices.

$$\begin{aligned} \sigma_0(X) &= (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3) \\ \sigma_1(X) &= (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10) \end{aligned}$$

> Note: $\gg$ is a right-**shift**, not a rotation.

### σ_0 function

$(X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3)$ is equivalent to $(X \ggg 7) \oplus (X \lll 14) \oplus (X \gg 3)$.

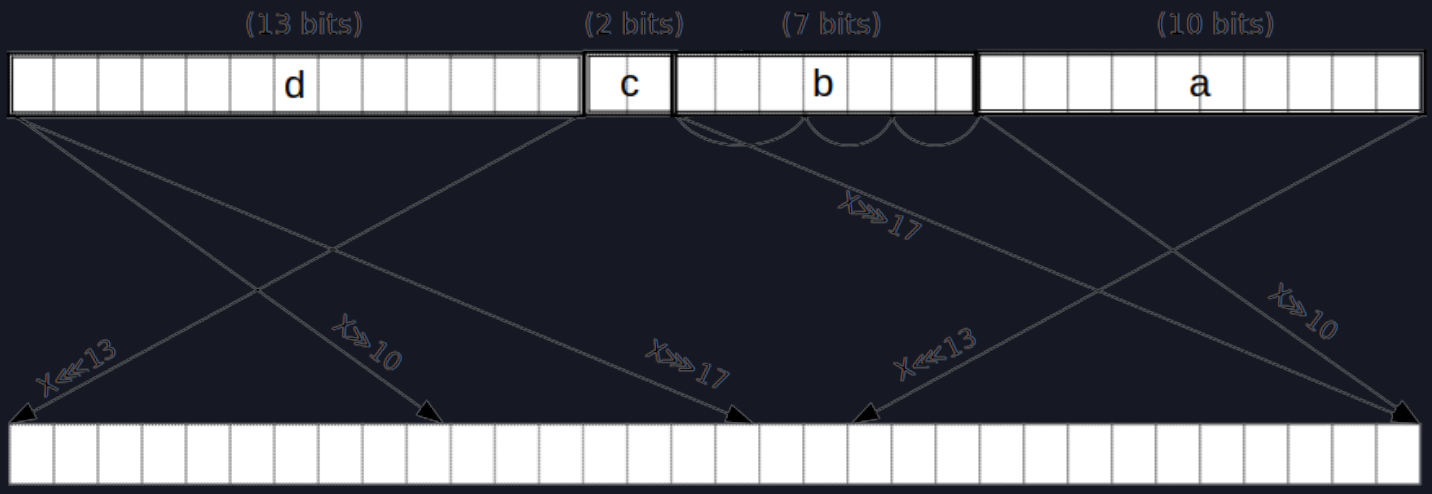



As above but with pieces $(a, b, c, d)$ of lengths $(3, 4, 11, 14)$ counting from the little end. Split $b$ into two 2-bit subpieces.

$$
\begin{array}{rcccccccc}
 & (0^{[3]} & || & d & || & c & || & b) & \oplus \\
 & (\ b & || & a & || & d & || & c) & \oplus \\
 & (\ c & || & b & || & a & || & d) & \\
 & & & & \Downarrow & & & & \\
R' = & & & 4^{15}d & + & 4^{4}c & + & b & + \\
 & 4^{28}b & + & 4^{25}a & + & 4^{11}d & + & c & + \\
 & 4^{21}c & + & 4^{17}b & + & 4^{14}a & + & d &
\end{array}
$$

## σ_1 function

$(X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10)$ is equivalent to $(X \lll 15) \oplus (X \lll 13) \oplus (X \gg 10)$.



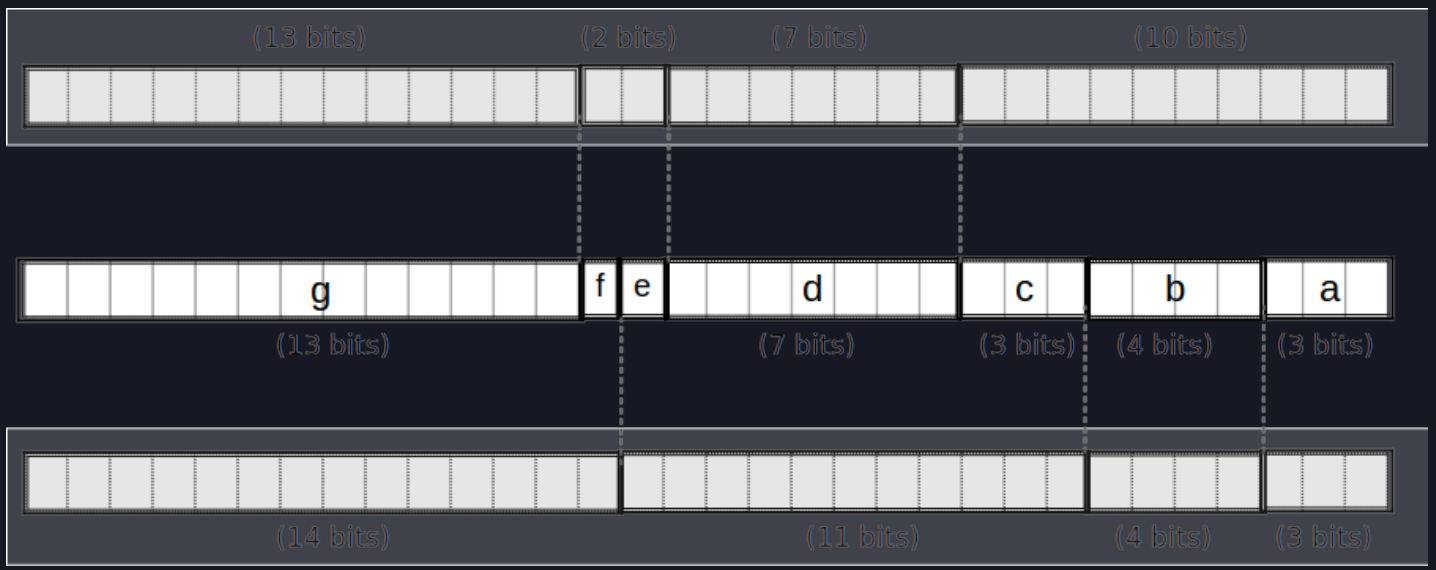

TODO: this diagram doesn't match the expression on the right. This is just for consistency with the other diagrams.

As above but with pieces $(a, b, c, d)$ of lengths $(10, 7, 2, 13)$ counting from the little end. Split $b$ into $(3, 2, 2)$-bit subpieces.

$$
\begin{array}{rccccccccc}
 & (0^{[10]} & || & d & || & c & || & b) & \oplus \\
 & (\ b & || & a & || & d & || & c) & \oplus \\
 & (\ c & || & b & || & a & || & d) & \\
 & & & & \Downarrow & & & & \\
R' = & & & 4^9 d & + & 4^7 c & + & b & + \\
 & 4^{25} b & + & 4^{15} a & + & 4^2 d & + & c & + \\
 & 4^{30} c & + & 4^{23} b & + & 4^{13} a & + & d &
\end{array}
$$

## Message scheduling

We apply $\sigma_0$ to $W_{1..48}$, and $\sigma_1$ to $W_{14..61}$. In order to avoid redundant applications of `spread`, we can merge the splitting into pieces for $\sigma_0$ and $\sigma_1$ in the case of $W_{14..48}$. Merging the piece lengths $(3, 4, 11, 14)$ and $(10, 7, 2, 13)$ gives pieces of lengths $(3, 4, 3, 7, 1, 1, 13)$.




If we can do the merged split in $3$ rows (as opposed to a total of $4$ rows when splitting for $\sigma_0$ and $\sigma_1$ separately), we save $35$ rows.

> These might even be doable in $2$ rows; not sure. —Daira

We can merge the reduction mod $2^{32}$ of $W_{16..61}$ into their splitting when they are used to compute subsequent words, similarly to what we did for $A$ and $E$ in the round function.

We will still need to reduce $W_{62..63}$ since they are not split. (Technically we could leave them unreduced since they will be reduced later when they are used to compute $A_{new}$ and $E_{new}$ --

but that would require handling a carry of up to $10$ rather than $6$, so it's not worth the complexity.)

The resulting message schedule cost is:

- $2$ rows to constrain $W_0$ to $32$ bits
  - This is technically optional, but let's do it for robustness, since the rest of the input is constrained for free.
- $13 * 2$ rows to split $W_{1..13}$ into $(3, 4, 11, 14)$-bit pieces
- $35 * 3$ rows to split $W_{14..48}$ into $(3, 4, 3, 7, 1, 1, 13)$-bit pieces (merged with a reduction for $W_{16..48}$)
- $13 * 2$ rows to split $W_{49..61}$ into $(10, 7, 2, 13)$-bit pieces (merged with a reduction)
- $4 * 48$ rows to extract the results of $\sigma_0$ for $W_{1..48}$
- $4 * 48$ rows to extract the results of $\sigma_1$ for $W_{14..61}$
- $2 * 2$ rows to reduce $W_{62..63}$
- $= 547$ rows.

## Overall cost

For each round:

- $8$ rows for $Ch$
- $4$ rows for $Maj$
- $6$ rows for $\Sigma_0$
- $6$ rows for $\Sigma_1$
- $reduce_6$ and $reduce_7$ are always free
- $= 24$ per round

This gives $24 * 64 = 1792$ rows for all of "step 3", to which we need to add:

- $547$ rows for message scheduling
- $2 * 8$ rows for $8$ reductions mod $2^{32}$ in "step 4"

giving a total of $2099$ rows.

## Tables

We only require one table **`spread`**, with $2^{16}$ rows and $3$ columns. We need a tag column to allow selecting $(7, 10, 11, 13, 14)$-bit subsets of the table for $\Sigma_{0..1}$ and $\sigma_{0..1}$.

### `spread` table

| row | tag | table (16b) | spread (32b) |
|---|---|---|---|
| $0$ | 0 | 0000000000000000 | 00000000000000000000000000000000 |
| $1$ | 0 | 0000000000000001 | 00000000000000000000000000000001 |
| $2$ | 0 | 0000000000000010 | 00000000000000000000000000000100 |
| $3$ | 0 | 0000000000000011 | 00000000000000000000000000000101 |
| ... | 0 | ... | ... |
| $2^7 - 1$ | 0 | 0000000001111111 | 00000000000000000001010101010101 |
| $2^7$ | 1 | 0000000010000000 | 00000000000000000100000000000000 |
| ... | 1 | ... | ... |
| $2^{10} - 1$ | 1 | 0000001111111111 | 00000000000001010101010101010101 |
| ... | 2 | ... | ... |
| $2^{11} - 1$ | 2 | 0000011111111111 | 00000000010101010101010101010101 |
| ... | 3 | ... | ... |
| $2^{13} - 1$ | 3 | 0001111111111111 | 00000001010101010101010101010101 |
| ... | 4 | ... | ... |
| $2^{14} - 1$ | 4 | 0011111111111111 | 00000101010101010101010101010101 |
| ... | 5 | ... | ... |
| $2^{16} - 1$ | 5 | 1111111111111111 | 01010101010101010101010101010101 |

For example, to do an $11$-bit `spread` lookup, we polynomial-constrain the tag to be in $\{0, 1, 2\}$. For the most common case of a $16$-bit lookup, we don't need to constrain the tag. Note that we can fill any unused rows beyond $2^{16}$ with a duplicate entry, e.g. all-zeroes.

# Gates

## Choice gate

Input from previous operations:

- $E', F', G'$, 64-bit spread forms of 32-bit words $E, F, G$, assumed to be constrained by previous operations

- in practice, we'll have the spread forms of $E', F', G'$ after they've been decomposed into 16-bit subpieces
- $evens$ is defined as $\texttt{spread}(2^{32} - 1)$
  - $evens_0 = evens_1 = \texttt{spread}(2^{16} - 1)$

**E ∧ F**

| s_ch | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $P_0^{even}$ | $\texttt{spread}(P_0^{even})$ | $\texttt{spread}(E^{lo})$ | $\texttt{spread}(E^{h}$ |
| 1 | {0,1,2,3,4,5} | $P_0^{odd}$ | $\texttt{spread}(P_0^{odd})$ | $\texttt{spread}(P_1^{odd})$ | |
| 0 | {0,1,2,3,4,5} | $P_1^{even}$ | $\texttt{spread}(P_1^{even})$ | $\texttt{spread}(F^{lo})$ | $\texttt{spread}(F^{h}$ |
| 0 | {0,1,2,3,4,5} | $P_1^{odd}$ | $\texttt{spread}(P_1^{odd})$ | | |

**¬E ∧ G**

| s_ch_neg | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a$ |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $Q_0^{even}$ | $\texttt{spread}(Q_0^{even})$ | $\texttt{spread}(E_{neg}^{lo})$ | $\texttt{sprea}$ |
| 1 | {0,1,2,3,4,5} | $Q_0^{odd}$ | $\texttt{spread}(Q_0^{odd})$ | $\texttt{spread}(Q_1^{odd})$ | |
| 0 | {0,1,2,3,4,5} | $Q_1^{even}$ | $\texttt{spread}(Q_1^{even})$ | $\texttt{spread}(G^{lo})$ | $\texttt{sprea}$ |
| 0 | {0,1,2,3,4,5} | $Q_1^{odd}$ | $\texttt{spread}(Q_1^{odd})$ | | |

Constraints:

- `s_ch` (choice): $LHS - RHS = 0$
  - $LHS = a_3\omega^{-1} + a_3\omega + 2^{32}(a_4\omega^{-1} + a_4\omega)$
  - $RHS = a_2\omega^{-1} + 2 * a_2 + 2^{32}(a_2\omega + 2 * a_3)$
- `s_ch_neg` (negation): `s_ch` with an extra negation check
- **spread** lookup on $(a_0, a_1, a_2)$
- permutation between $(a_2, a_3)$

Output: $Ch(E, F, G) = P^{odd} + Q^{odd} = (P_0^{odd} + Q_0^{odd}) + 2^{16}(P_1^{odd} + Q_1^{odd})$

## Majority gate

Input from previous operations:

- $A', B', C'$, 64-bit spread forms of 32-bit words $A, B, C$, assumed to be constrained by previous operations
  - in practice, we'll have the spread forms of $A', B', C'$ after they've been decomposed into $16$-bit subpieces

| s_maj | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $M_0^{even}$ | $\texttt{spread}(M_0^{even})$ | | $\texttt{spread}$ |
| 1 | {0,1,2,3,4,5} | $M_0^{odd}$ | $\texttt{spread}(M_0^{odd})$ | $\texttt{spread}(M_1^{odd})$ | $\texttt{spread}$ |
| 0 | {0,1,2,3,4,5} | $M_1^{even}$ | $\texttt{spread}(M_1^{even})$ | | $\texttt{spread}$ |
| 0 | {0,1,2,3,4,5} | $M_1^{odd}$ | $\texttt{spread}(M_1^{odd})$ | | |

Constraints:

- `s_maj` (majority): $LHS - RHS = 0$
  - $LHS = \texttt{spread}(M_0^{even}) + 2 \cdot \texttt{spread}(M_0^{odd}) + 2^{32} \cdot \texttt{spread}(M_1^{even}) + 2^{33} \cdot \texttt{spread}(M_1^{odd})$
  - $RHS = A' + B' + C'$
- $\texttt{spread}$ lookup on $(a_0, a_1, a_2)$
- permutation between $(a_2, a_3)$

Output: $Maj(A, B, C) = M^{odd} = M_0^{odd} + 2^{16} M_1^{odd}$

### Σ_0 gate

$A$ is a 32-bit word split into $(2, 11, 9, 10)$-bit chunks, starting from the little end. We refer to these chunks as $(a(2), b(11), c(9), d(10))$ respectively, and further split $c(9)$ into three 3-bit chunks $c(9)^{lo}, c(9)^{mid}, c(9)^{hi}$. We witness the spread versions of the small chunks.

$$\begin{aligned} \Sigma_0(A) &= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \\ &= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \lll 10) \end{aligned}$$

| s_upp_sigma_0 | $a_0$ | $a_1$ | $a_2$ | $a_3$ | |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\texttt{spread}(R_0^{even})$ | $c(9)^{lo}$ | s |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\texttt{spread}(R_0^{odd})$ | $\texttt{spread}(R_1^{odd})$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\texttt{spread}(R_1^{even})$ | $a(2)$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\texttt{spread}(R_1^{odd})$ | | |

Constraints:

- `s_upp_sigma_0` ($\Sigma_0$ constraint): $LHS - RHS + tag + decompose = 0$

$$\begin{aligned} tag &= constrain_1(a_0\omega^{-1}) + constrain_2(a_0\omega) \\ decompose &= a(2) + 2^2 b(11) + 2^{13} c(9)^{lo} + 2^{16} c(9)^{mid} + 2^{19} c(9)^{hi} + 2^{22} d(10) \\ LHS &= \mathtt{spread}(R_0^{even}) + 2 \cdot \mathtt{spread}(R_0^{odd}) + 2^{32} \cdot \mathtt{spread}(R_1^{even}) + 2^{33} \cdot \mathtt{spr} \end{aligned}$$

$$\begin{aligned} RHS = \quad & 4^{30}\mathtt{spread}(a(2)) + 4^{20}\mathtt{spread}(d(10)) + 4^{17}\mathtt{spread}(c(9)^{hi}) + \\ & 4^{21}\mathtt{spread}(b(11)) + 4^{19}\mathtt{spread}(a(2)) + 4^{9}\mathtt{spread}(d(10)) + \\ & 4^{29}\mathtt{spread}(c(9)^{hi}) + 4^{26}\mathtt{spread}(c(9)^{mid}) + 4^{23}\mathtt{spread}(c(9)^{lo}) + \end{aligned}$$

- **spread** lookup on $a_0, a_1, a_2$
- 2-bit range check and 2-bit spread check on $a(2)$
- 3-bit range check and 3-bit spread check on $c(9)^{lo}, c(9)^{mid}, c(9)^{hi}$

(see section Helper gates)

Output: $\Sigma_0(A) = R^{even} = R_0^{even} + 2^{16} R_1^{even}$

## Σ_1 gate

$E$ is a 32-bit word split into $(6, 5, 14, 7)$-bit chunks, starting from the little end. We refer to these chunks as $(a(6), b(5), c(14), d(7))$ respectively, and further split $a(6)$ into two 3-bit chunks $a(6)^{lo}, a(6)^{hi}$ and $b$ into (2,3)-bit chunks $b(5)^{lo}, b(5)^{hi}$. We witness the spread versions of the small chunks.

$$\begin{aligned} \Sigma_1(E) &= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25) \\ &= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \lll 7) \end{aligned}$$

| s_upp_sigma_1 | $a_0$ | $a_1$ | $a_2$ | $a_3$ | |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\mathtt{spread}(R_0^{even})$ | $b(5)^{lo}$ | s |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\mathtt{spread}(R_0^{odd})$ | $\mathtt{spread}(R_1^{odd})$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\mathtt{spread}(R_1^{even})$ | $a(6)^{lo}$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\mathtt{spread}(R_1^{odd})$ | | |

Constraints:

- `s_upp_sigma_1` ($\Sigma_1$ constraint): $LHS - RHS + tag + decompose = 0$

$$\begin{aligned} tag &= a_0\omega^{-1} + constrain_4(a_0\omega) \\ decompose &= a(6)^{lo} + 2^3 a(6)^{hi} + 2^6 b(5)^{lo} + 2^8 b(5)^{hi} + 2^{11} c(14) + 2^{25} d(7) - \\ LHS &= \mathtt{spread}(R_0^{even}) + 2 \cdot \mathtt{spread}(R_0^{odd}) + 2^{32} \cdot \mathtt{spread}(R_1^{even}) + 2^{33} \cdot \mathtt{spr} \end{aligned}$$

$$RHS = 4^{29}\texttt{spread}(a(6)^{hi}) + 4^{26}\texttt{spread}(a(6)^{lo}) + 4^{19}\texttt{spread}(d(7)) + 4$$
$$4^{29}\texttt{spread}(b(5)^{hi}) + 4^{27}\texttt{spread}(b(5)^{lo}) + 4^{24}\texttt{spread}(a(6)^{hi}) + 4$$
$$4^{18}\texttt{spread}(c(14)) + 4^{15}\texttt{spread}(b(5)^{hi}) + 4^{13}\texttt{spread}(b(5)^{lo}) + 4$$

- **spread** lookup on $a_0, a_1, a_2$
- 2-bit range check and 2-bit spread check on $b(5)^{lo}$
- 3-bit range check and 3-bit spread check on $a(6)^{lo}, a(6)^{hi}, b(4)^{hi}$

(see section Helper gates)

Output: $\Sigma_1(E) = R^{even} = R_0^{even} + 2^{16} R_1^{even}$

## σ_0 gate

### v1

v1 of the $\sigma_0$ gate takes in a word that's split into $(3, 4, 11, 14)$-bit chunks (already constrained by message scheduling). We refer to these chunks respectively as $(a(3), b(4), c(11), d(14))$. $b(4)$ is further split into two 2-bit chunks $b(4)^{lo}, b(4)^{hi}$. We witness the spread versions of the small chunks. We already have $\texttt{spread}(c(11))$ and $\texttt{spread}(d(14))$ from the message scheduling.

$(X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3)$ is equivalent to $(X \ggg 7) \oplus (X \lll 14) \oplus (X \gg 3)$.

| s_low_sigma_0 | $a_0$ | $a_1$ | $a_2$ | $a_3$ | |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\texttt{spread}(R_0^{even})$ | $b(4)^{lo}$ | s |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\texttt{spread}(R_0^{odd})$ | $\texttt{spread}(R_1^{odd})$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\texttt{spread}(R_1^{even})$ | $0$ | 0 |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\texttt{spread}(R_1^{odd})$ | | |

Constraints:

- `s_low_sigma_0` ($\sigma_0$ v1 constraint): $LHS - RHS = 0$

$$LHS = \texttt{spread}(R_0^{even}) + 2 \cdot \texttt{spread}(R_0^{odd}) + 2^{32} \cdot \texttt{spread}(R_1^{even}) + 2^{33} \cdot \texttt{spread}(R$$

$$RHS = 4^{15}d(14) + 4^4 c(11) + 4^2 b(4)^{hi} + b(4)^{lo} +$$
$$4^{30}b(4)^{hi} + 4^{28}b(4)^{lo} + 4^{25}a(3) + 4^{11}d(14) + c(11) +$$
$$4^{21}c(11) + 4^{19}b(4)^{hi} + 4^{17}b(4)^{lo} + 4^{14}a(3) + d(14)$$

- check that `b` was properly split into subsections for 4-bit pieces.
  - $W^{b(4)lo} + 2^2 W^{b(4)hi} - W = 0$
- 2-bit range check and 2-bit spread check on $b(4)^{lo}, b(4)^{hi}$
- 3-bit range check and 3-bit spread check on $a(3)$

**v2**

v2 of the $\sigma_0$ gate takes in a word that's split into $(3, 4, 3, 7, 1, 1, 13)$-bit chunks (already constrained by message scheduling). We refer to these chunks respectively as $(a(3), b(4), c(3), d(7), e(1), f(1), g(13))$. We already have $\texttt{spread}(d(7)), \texttt{spread}(g(13))$ from the message scheduling. The 1-bit $e(1), f(1)$ remain unchanged by the spread operation and can be used directly. We further split $b(4)$ into two 2-bit chunks $b(4)^{lo}, b(4)^{hi}$. We witness the spread versions of the small chunks.

$(X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3)$ is equivalent to $(X \ggg 7) \oplus (X \lll 14) \oplus (X \gg 3)$.

| s_low_sigma_0_v2 | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\texttt{spread}(R_0^{even})$ | $b(4)^{lo}$ |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\texttt{spread}(R_0^{odd})$ | $\texttt{spread}(R_1^{odd})$ |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\texttt{spread}(R_1^{even})$ | $a(3)$ |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\texttt{spread}(R_1^{odd})$ | |

Constraints:

- `s_low_sigma_0_v2` ($\sigma_0$ v2 constraint): $LHS - RHS = 0$

$$LHS = \texttt{spread}(R_0^{even}) + 2 \cdot \texttt{spread}(R_0^{odd}) + 2^{32} \cdot \texttt{spread}(R_1^{even}) + 2^{33} \cdot \texttt{spread}(R$$

$$\begin{aligned} RHS = & \quad 4^{16} g(13) + 4^{15} f(1) + 4^{14} e(1) + 4^{7} d(7) + \\ & 4^{30} b(4)^{hi} + 4^{28} b(4)^{lo} + 4^{25} a(3) + 4^{12} g(13) + 4^{11} f(1) + \\ & 4^{31} e(1) + 4^{24} d(7) + 4^{21} c(3) + 4^{19} b(4)^{hi} + 4^{17} b(4)^{lo} + \end{aligned}$$

- check that `b` was properly split into subsections for 4-bit pieces.
  - $W^{b(4)lo} + 2^2 W^{b(4)hi} - W = 0$
- 2-bit range check and 2-bit spread check on $b(4)^{lo}, b(4)^{hi}$
- 3-bit range check and 3-bit spread check on $a(3), c(3)$

## σ_1 gate

### v1

v1 of the $\sigma_1$ gate takes in a word that's split into $(10, 7, 2, 13)$-bit chunks (already constrained by message scheduling). We refer to these chunks respectively as $(a(10), b(7), c(2), d(13))$. $b(7)$ is further split into $(2, 2, 3)$-bit chunks $b(7)^{lo}, b(7)^{mid}, b(7)^{hi}$. We witness the spread versions of the small chunks. We already have $\texttt{spread}(a(10))$ and $\texttt{spread}(d(13))$ from the message scheduling.

$(X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10)$ is equivalent to $(X \lll 15) \oplus (X \lll 13) \oplus (X \gg 10)$.

| s_low_sigma_1 | $a_0$ | $a_1$ | $a_2$ | $a_3$ | |
|---|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\texttt{spread}(R_0^{even})$ | $b(7)^{lo}$ | s |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\texttt{spread}(R_0^{odd})$ | $\texttt{spread}(R_1^{odd})$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\texttt{spread}(R_1^{even})$ | $c(2)$ | s |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\texttt{spread}(R_1^{odd})$ | | |

Constraints:

- `s_low_sigma_1` ($\sigma_1$ v1 constraint): $LHS - RHS = 0$

$$LHS = \texttt{spread}(R_0^{even}) + 2 \cdot \texttt{spread}(R_0^{odd}) + 2^{32} \cdot \texttt{spread}(R_1^{even}) + 2^{33} \cdot \texttt{spre}$$

$$\begin{aligned} RHS = & \quad 4^9 d(13) + 4^7 c(2) + 4^4 b(7)^{hi} + 4^2 b(7)^{n} \\ & 4^{29} b(7)^{hi} + 4^{27} b(7)^{mid} + 4^{25} b(7)^{lo} + 4^{15} a(10) + 4^2 d(13 \\ & 4^{30} c(2) + 4^{27} b(7)^{hi} + 4^{25} b(7)^{mid} + 4^{23} b(7)^{lo} + 4^{13} a(1( \end{aligned}$$

- check that `b` was properly split into subsections for 7-bit pieces.
  - $W^{b(7)lo} + 2^2 W^{b(7)mid} + 2^4 W^{b(7)hi} - W = 0$
- 2-bit range check and 2-bit spread check on $b(7)^{lo}, b(7)^{mid}, c(2)$
- 3-bit range check and 3-bit spread check on $b(7)^{hi}$

### v2

v2 of the $\sigma_1$ gate takes in a word that's split into $(3, 4, 3, 7, 1, 1, 13)$-bit chunks (already constrained by message scheduling). We refer to these chunks respectively as

$(a(3), b(4), c(3), d(7), e(1), f(1), g(13))$. We already have $\texttt{spread}(d(7)), \texttt{spread}(g(13))$ from the message scheduling. The 1-bit $e(1), f(1)$ remain unchanged by the spread operation and can be used directly. We further split $b(4)$ into two 2-bit chunks $b(4)^{lo}, b(4)^{hi}$. We witness the spread versions of the small chunks.

$(X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10)$ is equivalent to $(X \lll 15) \oplus (X \lll 13) \oplus (X \gg 10)$.

| s_low_sigma_1_v2 | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|---|
| 0 | {0,1,2,3,4,5} | $R_0^{even}$ | $\texttt{spread}(R_0^{even})$ | $b(4)^{lo}$ |
| 1 | {0,1,2,3,4,5} | $R_0^{odd}$ | $\texttt{spread}(R_0^{odd})$ | $\texttt{spread}(R_1^{odd})$ |
| 0 | {0,1,2,3,4,5} | $R_1^{even}$ | $\texttt{spread}(R_1^{even})$ | $a(3)$ |
| 0 | {0,1,2,3,4,5} | $R_1^{odd}$ | $\texttt{spread}(R_1^{odd})$ | |

Constraints:

- `s_low_sigma_1_v2` ($\sigma_1$ v2 constraint): $LHS - RHS = 0$

$$LHS = \texttt{spread}(R_0^{even}) + 2 \cdot \texttt{spread}(R_0^{odd}) + 2^{32} \cdot \texttt{spread}(R_1^{even}) + 2^{33} \cdot \texttt{spread}(R$$

$$\begin{aligned} RHS = & & & & & & 4^9 g(13) & + & 4^8 f(1) & + & 4 \\ & 4^{25} d(7) & + & 4^{22} c(3) & + & 4^{20} b(4)^{hi} & + & 4^{18} b(4)^{lo} & + & 4^{15} a & + & 4^2 \\ & 4^{31} f(1) & + & 4^{30} e(1) & + & 4^{23} d(7) & + & 4^{20} c(3) & + & 4^{18} b(4)^{hi} & + & 4^{1} \end{aligned}$$

- check that `b` was properly split into subsections for 4-bit pieces.
  - $W^{b(4)lo} + 2^2 W^{b(4)hi} - W = 0$
- 2-bit range check and 2-bit spread check on $b(4)^{lo}, b(4)^{hi}$
- 3-bit range check and 3-bit spread check on $a(3), c(3)$

## Helper gates

### Small range constraints

Let $constrain_n(x) = \prod_{i=0}^{n}(x - i)$. Constraining this expression to equal zero enforces that $x$ is in $[0..n]$.

### 2-bit range check

$$(a - 3)(a - 2)(a - 1)(a) = 0$$

| sr2 | $a_0$ |
|---|---|
| 1 | a |

**2-bit spread**

$$l_1(a) + 4 * l_2(a) + 5 * l_3(a) - a' = 0$$

| ss2 | $a_0$ | $a_1$ |
|---|---|---|
| 1 | a | a' |

with interpolation polynomials:

- $l_0(a) = \frac{(a-3)(a-2)(a-1)}{(-3)(-2)(-1)}$ ($\texttt{spread}(00) = 0000$)
- $l_1(a) = \frac{(a-3)(a-2)(a)}{(-2)(-1)(1)}$ ($\texttt{spread}(01) = 0001$)
- $l_2(a) = \frac{(a-3)(a-1)(a)}{(-1)(1)(2)}$ ($\texttt{spread}(10) = 0100$)
- $l_3(a) = \frac{(a-2)(a-1)(a)}{(1)(2)(3)}$ ($\texttt{spread}(11) = 0101$)

**3-bit range check**

$$(a-7)(a-6)(a-5)(a-4)(a-3)(a-2)(a-1)(a) = 0$$

| sr3 | $a_0$ |
|---|---|
| 1 | a |

**3-bit spread**

$$l_1(a) + 4 * l_2(a) + 5 * l_3(a) + 16 * l_4(a) + 17 * l_5(a) + 20 * l_6(a) + 21 * l_7(a) - a' = 0$$

| ss3 | $a_0$ | $a_1$ |
|---|---|---|
| 1 | a | a' |

with interpolation polynomials:

- $l_0(a) = \frac{(a-7)(a-6)(a-5)(a-4)(a-3)(a-2)(a-1)}{(-7)(-6)(-5)(-4)(-3)(-2)(-1)}$ ($\texttt{spread}(000) = 000000$)
- $l_1(a) = \frac{(a-7)(a-6)(a-5)(a-4)(a-3)(a-2)(a)}{(-6)(-5)(-4)(-3)(-2)(-1)(1)}$ ($\texttt{spread}(001) = 000001$)
- $l_2(a) = \frac{(a-7)(a-6)(a-5)(a-4)(a-3)(a-1)(a)}{(-5)(-4)(-3)(-2)(-1)(1)(2)}$ ($\texttt{spread}(010) = 000100$)
- $l_3(a) = \frac{(a-7)(a-6)(a-5)(a-4)(a-2)(a-1)(a)}{(-4)(-3)(-2)(-1)(1)(2)(3)}$ ($\texttt{spread}(011) = 000101$)
- $l_4(a) = \frac{(a-7)(a-6)(a-5)(a-3)(a-2)(a-1)(a)}{(-3)(-2)(-1)(1)(2)(3)(4)}$ ($\texttt{spread}(100) = 010000$)

- $l_5(a) = \frac{(a-7)(a-6)(a-4)(a-3)(a-2)(a-1)(a)}{(-2)(-1)(1)(2)(3)(4)(5)}$ ($\mathtt{spread}(101) = 010001$)
- $l_6(a) = \frac{(a-7)(a-5)(a-4)(a-3)(a-2)(a-1)(a)}{(-1)(1)(2)(3)(4)(5)(6)}$ ($\mathtt{spread}(110) = 010100$)
- $l_7(a) = \frac{(a-6)(a-5)(a-4)(a-3)(a-2)(a-1)(a)}{(1)(2)(3)(4)(5)(6)(7)}$ ($\mathtt{spread}(111) = 010101$)

**reduce_6 gate**

Addition $\pmod{2^{32}}$ of 6 elements

Input:

- $E$
- $\{e_i^{lo}, e_i^{hi}\}_{i=0}^{5}$
- $carry$

Check: $E = e_0 + e_1 + e_2 + e_3 + e_4 + e_5 \pmod{32}$

Assume inputs are constrained to 16 bits.

- Addition gate (sa):
  - $a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 - a_7 = 0$
- Carry gate (sc):
  - $2^{16}a_6\omega^{-1} + a_6 + [(a_6 - 5)(a_6 - 4)(a_6 - 3)(a_6 - 2)(a_6 - 1)(a_6)] = 0$

| sa | sc | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | $e_0^{lo}$ | $e_1^{lo}$ | $e_2^{lo}$ | $e_3^{lo}$ | $e_4^{lo}$ | $e_5^{lo}$ | $-carry * 2^{16}$ | $E^{lo}$ |
| 1 | 1 | $e_0^{hi}$ | $e_1^{hi}$ | $e_2^{hi}$ | $e_3^{hi}$ | $e_4^{hi}$ | $e_5^{hi}$ | $carry$ | $E^{hi}$ |

Assume inputs are constrained to 16 bits.

- Addition gate (sa):
  - $a_0\omega^{-1} + a_1\omega^{-1} + a_2\omega^{-1} + a_0 + a_1 + a_2 + a_3\omega^{-1} - a_3 = 0$
- Carry gate (sc):
  - $2^{16}a_3\omega + a_3\omega^{-1} = 0$

| sa | sc | $a_0$ | $a_1$ | $a_2$ | $a_3$ |
|---|---|---|---|---|---|
| 0 | 0 | $e_0^{lo}$ | $e_1^{lo}$ | $e_2^{lo}$ | $-carry * 2^{16}$ |
| 1 | 1 | $e_3^{lo}$ | $e_4^{lo}$ | $e_5^{lo}$ | $E^{lo}$ |
| 0 | 0 | $e_0^{hi}$ | $e_1^{hi}$ | $e_2^{hi}$ | $carry$ |
| 1 | 0 | $e_3^{hi}$ | $e_4^{hi}$ | $e_5^{hi}$ | $E^{hi}$ |

**reduce_7 gate**

Addition $\pmod{2^{32}}$ of 7 elements

Input:

- $E$
- $\{e_i^{lo}, e_i^{hi}\}_{i=0}^{6}$
- $carry$

Check: $E = e_0 + e_1 + e_2 + e_3 + e_4 + e_5 + e_6 \pmod{32}$

Assume inputs are constrained to 16 bits.

- Addition gate (sa):
  - $a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 - a_8 = 0$
- Carry gate (sc):
  - $2^{16}a_7\omega^{-1} + a_7 + [(a_7 - 6)(a_7 - 5)(a_7 - 4)(a_7 - 3)(a_7 - 2)(a_7 - 1)(a_7)] = 0$

| sa | sc | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | $e_0^{lo}$ | $e_1^{lo}$ | $e_2^{lo}$ | $e_3^{lo}$ | $e_4^{lo}$ | $e_5^{lo}$ | $e_6^{lo}$ | $-carry * 2^{16}$ | $E$ |
| 1 | 1 | $e_0^{hi}$ | $e_1^{hi}$ | $e_2^{hi}$ | $e_3^{hi}$ | $e_4^{hi}$ | $e_5^{hi}$ | $e_6^{hi}$ | $carry$ | $E$ |

## Message scheduling region

For each block $M \in \{0,1\}^{512}$ of the padded message, $64$ words of $32$ bits each are constructed as follows:

- the first $16$ are obtained by splitting $M$ into $32$-bit blocks

$$M = W_0 || W_1 || \cdots || W_{14} || W_{15};$$

- the remaining $48$ words are constructed using the formula:

$$W_i = \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16},$$

  for $16 \leq i < 64$.

| sw | sd0 | sd1 | sd2 | sd3 | ss0 | ss0_v2 | ss1 | ss1_v2 | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |

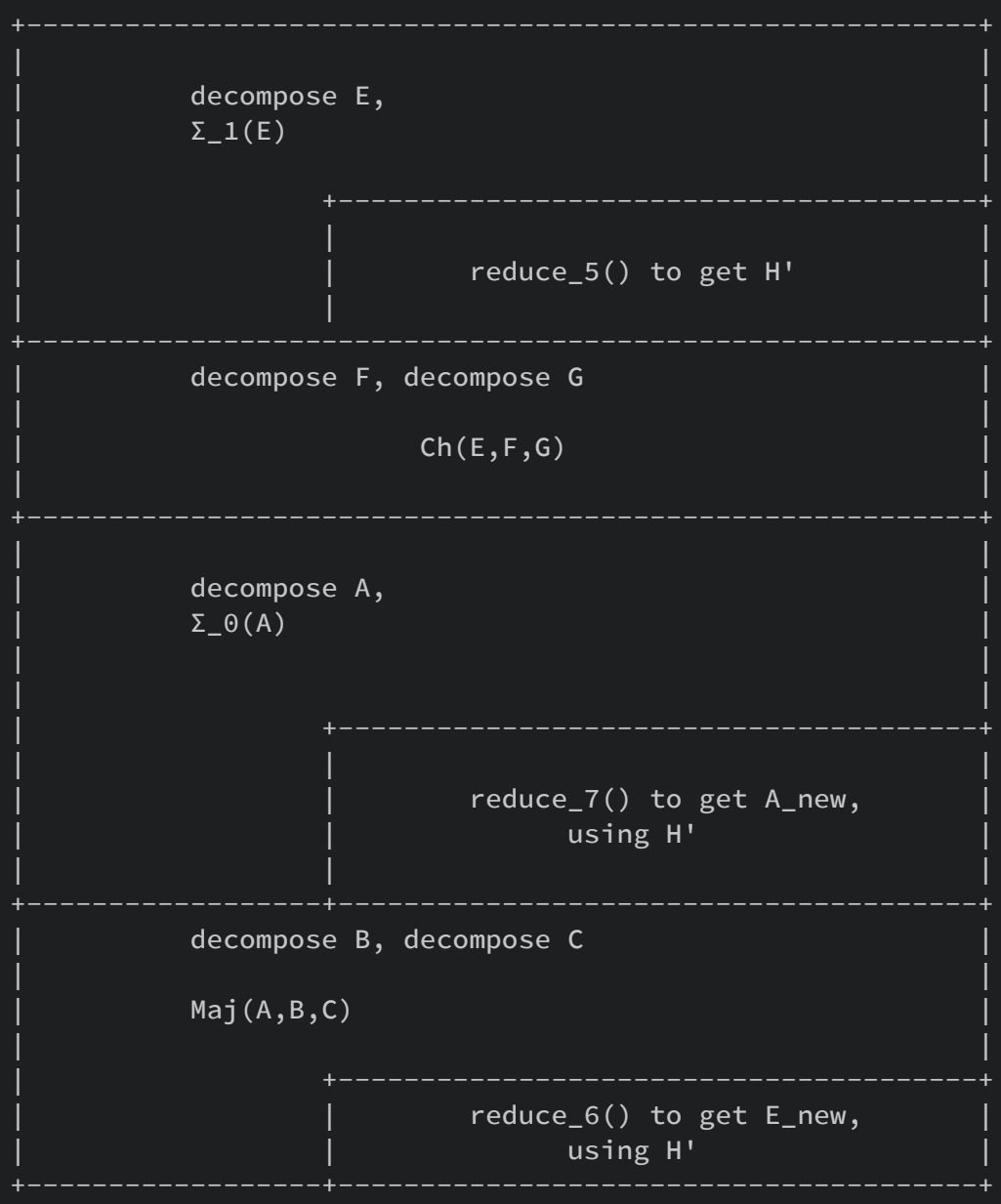
| sw | sd0 | sd1 | sd2 | sd3 | ss0 | ss0_v2 | ss1 | ss1_v2 | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2} |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1} |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |

| sw | sd0 | sd1 | sd2 | sd3 | ss0 | ss0_v2 | ss1 | ss1_v2 | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | {0,1,2,3 |

Constraints:

- `sw` : construct word using $reduce_4$
- `sd0` : decomposition gate for $W_0, W_{62}, W_{63}$
  - $W^{lo} + 2^{16} W^{hi} - W = 0$
- `sd1` : decomposition gate for $W_{1..13}$ (split into $(3, 4, 11, 14)$-bit pieces)
  - $W^{a(3)} + 2^3 W^{b(4)lo} + 2^5 W^{b(4)hi} + 2^7 W^{c(11)} + 2^{18} W^{d(14)} - W = 0$
- `sd2` : decomposition gate for $W_{14..48}$ (split into $(3, 4, 3, 7, 1, 1, 13)$-bit pieces)
  - $W^{a(3)} + 2^3 W^{b(4)lo} + 2^5 W^{b(4)hi} + 2^7 W^{c(11)} + 2^{10} W^{d(14)} + 2^{17} W^{e(1)} + 2^{18} W^{f(1)} + 2^{19} W^{g(13)} - W = 0$
- `sd3` : decomposition gate for $W_{49..61}$ (split into $(10, 7, 2, 13)$-bit pieces)
  - $W^{a(10)} + 2^{10} W^{b(7)lo} + 2^{12} W^{b(7)mid} + 2^{15} W^{b(7)hi} + 2^{17} W^{c(2)} + 2^{19} W^{d(13)} - W = 0$

## Compression region




### Initial round:

| s_digest | sd_abcd | sd_efgh | ss0 | ss1 | s_maj | s_ch_neg | s_ch |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| s_digest | sd_abcd | sd_efgh | ss0 | ss1 | s_maj | s_ch_neg | s_ch |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Steady-state:**

| s_digest | sd_abcd | sd_efgh | ss0 | ss1 | s_maj | s_ch_neg | s_ch |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Final digest:**

| s_digest | sd_abcd | sd_efgh | ss0 | ss1 | s_maj | s_ch_neg | s_ch |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| s_digest | sd_abcd | sd_efgh | ss0 | ss1 | s_maj | s_ch_neg | s_ch |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Background Material

This section covers the background material required to understand the Halo 2 proving system. It is targeted at an ELI15 (Explain It Like I'm 15) level; if you think anything could do with additional explanation, let us know!

# Fields

A fundamental component of many cryptographic protocols is the algebraic structure known as a field. Fields are sets of objects (usually numbers) with two associated binary operators $+$ and $\times$ such that various field axioms hold. The real numbers $\mathbb{R}$ are an example of a field with uncountably many elements.

Halo makes use of *finite fields* which have a finite number of elements. Finite fields are fully classified as follows:

- if $\mathbb{F}$ is a finite field, it contains $|\mathbb{F}| = p^k$ elements for some integer $k \geq 1$ and some prime $p$;
- any two finite fields with the same number of elements are isomorphic. In particular, all of the arithmetic in a prime field $\mathbb{F}_p$ is isomorphic to addition and multiplication of integers modulo $p$, i.e. in $\mathbb{Z}_p$. This is why we often refer to $p$ as the *modulus*.

We'll write a field as $\mathbb{F}_q$ where $q = p^k$. The prime $p$ is called its *characteristic*. In the cases where $k > 1$ the field $\mathbb{F}_q$ is a $k$-degree extension of the field $\mathbb{F}_p$. (By analogy, the complex numbers $\mathbb{C} = \mathbb{R}(i)$ are an extension of the real numbers.) However, in Halo we do not use extension fields. Whenever we write $\mathbb{F}_p$ we are referring to what we call a *prime field* which has a prime $p$ number of elements, i.e. $k = 1$.

Important notes:

- There are two special elements in any field: $0$, the additive identity, and $1$, the multiplicative identity.
- The least significant bit of a field element, when represented as an integer in binary format, can be interpreted as its "sign" to help distinguish it from its additive inverse

(negation). This is because for some nonzero element $a$ which has a least significant bit $0$ we have that $-a = p - a$ has a least significant bit $1$, and vice versa. We could also use whether or not an element is larger than $(p-1)/2$ to give it a "sign."

Finite fields will be useful later for constructing polynomials and elliptic curves. Elliptic curves are examples of groups, which we discuss next.

## Groups

Groups are simpler and more limited than fields; they have only one binary operator $\cdot$ and fewer axioms. They also have an identity, which we'll denote as $1$.

Any non-zero element $a$ in a group has an *inverse* $b = a^{-1}$, which is the *unique* element $b$ such that $a \cdot b = 1$.

For example, the set of nonzero elements of $\mathbb{F}_p$ forms a group, where the group operation is given by multiplication on the field.

**(aside) Additive vs multiplicative notation**

If $\cdot$ is written as $\times$ or omitted (i.e. $a \cdot b$ written as $ab$), the identity as $1$, and inversion as $a^{-1}$, as we did above, then we say that the group is "written multiplicatively". If $\cdot$ is written as $+$, the identity as $0$ or $\mathcal{O}$, and inversion as $-a$, then we say it is "written additively".

It's conventional to use additive notation for elliptic curve groups, and multiplicative notation when the elements come from a finite field.

When additive notation is used, we also write

$$[k]A = \underbrace{A + A + \cdots + A}_{k \text{ times}}$$

for nonnegative $k$ and call this "scalar multiplication"; we also often use uppercase letters for variables denoting group elements. When multiplicative notation is used, we also write

$$a^k = \underbrace{a \times a \times \cdots \times a}_{k \text{ times}}$$

and call this "exponentiation". In either case we call the scalar $k$ such that $[k]g = a$ or $g^k = a$ the "discrete logarithm" of $a$ to base $g$. We can extend scalars to negative integers by inversion, i.e. $[-k]A + [k]A = \mathcal{O}$ or $a^{-k} \times a^k = 1$.

The *order* of an element $a$ of a finite group is defined as the smallest positive integer $k$ such that $a^k = 1$ (in multiplicative notation) or $[k]a = \mathcal{O}$ (in additive notation). The order *of the group* is the number of elements.

Groups always have a generating set, which is a set of elements such that we can produce any element of the group as (in multiplicative terminology) a product of powers of those elements. So if the generating set is $g_{1..k}$, we can produce any element of the group as $\prod_{i=1}^{k} g_i^{a_i}$. There can be many different generating sets for a given group.

A group is called cyclic if it has a (not necessarily unique) generating set with only a single element — call it $g$. In that case we can say that $g$ generates the group, and that the order of $g$ is the order of the group.

Any finite cyclic group $\mathbb{G}$ of order $n$ is isomorphic to the integers modulo $n$ (denoted $\mathbb{Z}/n\mathbb{Z}$), such that:

- the operation $\cdot$ in $\mathbb{G}$ corresponds to addition modulo $n$;
- the identity in $\mathbb{G}$ corresponds to $0$;
- some generator $g \in \mathbb{G}$ corresponds to $1$.

Given a generator $g$, the isomorphism is always easy to compute in the $\mathbb{Z}/n\mathbb{Z} \to \mathbb{G}$ direction; it is just $a \mapsto g^a$ (or in additive notation, $a \mapsto [a]g$). It may be difficult in general to compute in the $\mathbb{G} \to \mathbb{Z}/n\mathbb{Z}$ direction; we'll discuss this further when we come to elliptic curves.

If the order $n$ of a finite group is prime, then the group is cyclic, and every non-identity element is a generator.

## The multiplicative group of a finite field

We use the notation $\mathbb{F}_p^\times$ for the multiplicative group (i.e. the group operation is multiplication in $\mathbb{F}_p$) over the set $\mathbb{F}_p - \{0\}$.

A quick way of obtaining the inverse in $\mathbb{F}_p^\times$ is $a^{-1} = a^{p-2}$. The reason for this stems from Fermat's little theorem, which states that $a^p = a \pmod p$ for any integer $a$. If $a$ is nonzero, we can divide by $a$ twice to get $a^{p-2} = a^{-1}$.

Let's assume that $\alpha$ is a generator of $\mathbb{F}_p^\times$, so it has order $p-1$ (equal to the number of elements in $\mathbb{F}_p^\times$). Therefore, for any element in $a \in \mathbb{F}_p^\times$ there is a unique integer $i \in \{0..p-2\}$ such that $a = \alpha^i$.

Notice that $a \times b$ where $a, b \in \mathbb{F}_p^\times$ can really be interpreted as $\alpha^i \times \alpha^j$ where $a = \alpha^i$ and $b = \alpha^j$. Indeed, it holds that $\alpha^i \times \alpha^j = \alpha^{i+j}$ for all $0 \leq i, j < p-1$. As a result the multiplication of nonzero field elements can be interpreted as addition modulo $p-1$ with respect to some fixed generator $\alpha$. The addition just happens "in the exponent."

This is another way to look at where $a^{p-2}$ comes from for computing inverses in the field:

$$p - 2 \equiv -1 \pmod{p-1},$$

so $a^{p-2} = a^{-1}$.

### Montgomery's Trick

Montgomery's trick, named after Peter Montgomery (RIP) is a way to compute many group inversions at the same time. It is commonly used to compute inversions in $\mathbb{F}_p^\times$, which are quite computationally expensive compared to multiplication.

Imagine we need to compute the inverses of three nonzero elements $a, b, c \in \mathbb{F}_p^\times$. Instead, we'll compute the products $x = ab$ and $y = xc = abc$, and compute the inversion

$$z = y^{p-2} = \frac{1}{abc}.$$

We can now multiply $z$ by $x$ to obtain $\frac{1}{c}$ and multiply $z$ by $c$ to obtain $\frac{1}{ab}$, which we can then multiply by $a, b$ to obtain their respective inverses.

This technique generalizes to arbitrary numbers of group elements with just a single inversion necessary.

## Multiplicative subgroups

A *subgroup* of a group $G$ with operation $\cdot$, is a subset of elements of $G$ that also form a group under $\cdot$.

In the previous section we said that $\alpha$ is a generator of the $(p-1)$-order multiplicative group $\mathbb{F}_p^\times$. This group has *composite* order, and so by the Chinese remainder theorem[1] it has strict subgroups. As an example let's imagine that $p = 11$, and so $p-1$ factors into $5 \cdot 2$. Thus,

there is a generator $\beta$ of the $5$-order subgroup and a generator $\gamma$ of the $2$-order subgroup. All elements in $\mathbb{F}_p^\times$, therefore, can be written uniquely as $\beta^i \cdot \gamma^j$ for some $i$ (modulo $5$) and some $j$ (modulo $2$).

If we have $a = \beta^i \cdot \gamma^j$ notice what happens when we compute

$$a^5 = (\beta^i \cdot \gamma^j)^5 = \beta^{i \cdot 5} \cdot \gamma^{j \cdot 5} = \beta^0 \cdot \gamma^{j \cdot 5} = \gamma^{j \cdot 5};$$

we have effectively "killed" the $5$-order subgroup component, producing a value in the $2$-order subgroup.

Lagrange's theorem (group theory) states that the order of any subgroup $H$ of a finite group $G$ divides the order of $G$. Therefore, the order of any subgroup of $\mathbb{F}_p^\times$ must divide $p - 1$.

PLONK-based proving systems like Halo 2 are more convenient to use with fields that have a large number of multiplicative subgroups with a "smooth" distribution (which makes the performance cliffs smaller and more granular as circuit sizes increase). The Pallas and Vesta curves specifically have primes of the form

$$T \cdot 2^S = p - 1$$

with $S = 32$ and $T$ odd (i.e. $p - 1$ has 32 lower zero-bits). This means they have multiplicative subgroups of order $2^k$ for all $k \leq 32$. These 2-adic subgroups are nice for efficient FFTs, as well as enabling a wide variety of circuit sizes.

## Square roots

In a field $\mathbb{F}_p$ exactly half of all nonzero elements are squares; the remainder are non-squares or "quadratic non-residues". In order to see why, consider an $\alpha$ that generates the $2$-order multiplicative subgroup of $\mathbb{F}_p^\times$ (this exists because $p - 1$ is divisible by $2$ since $p$ is a prime greater than $2$) and $\beta$ that generates the $t$-order multiplicative subgroup of $\mathbb{F}_p^\times$ where $p - 1 = 2t$. Then every element $a \in \mathbb{F}_p^\times$ can be written uniquely as $\alpha^i \cdot \beta^j$ with $i \in \mathbb{Z}_2$ and $j \in \mathbb{Z}_t$. Half of all elements will have $i = 0$ and the other half will have $i = 1$.

Let's consider the simple case where $p \equiv 3 \pmod 4$ and so $t$ is odd (if $t$ is even, then $p - 1$ would be divisible by $4$, which contradicts $p$ being $3 \pmod 4$). If $a \in \mathbb{F}_p^\times$ is a square, then there must exist $b = \alpha^i \cdot \beta^j$ such that $b^2 = a$. But this means that

$$a = (\alpha^i \cdot \beta^j)^2 = \alpha^{2i} \cdot \beta^{2j} = \beta^{2j}.$$

In other words, all squares in this particular field do not generate the $2$-order multiplicative subgroup, and so since half of the elements generate the $2$-order subgroup then at most half

of the elements are square. In fact exactly half of the elements are square (since squaring each nonsquare element gives a unique square). This means we can assume all squares can be written as $\beta^m$ for some $m$, and therefore finding the square root is a matter of exponentiating by $2^{-1} \pmod t$.

In the event that $p \equiv 1 \pmod 4$ then things get more complicated because $2^{-1} \pmod t$ does not exist. Let's write $p - 1$ as $2^k \cdot t$ with $t$ odd. The case $k = 0$ is impossible, and the case $k = 1$ is what we already described, so consider $k \geq 2$. $\alpha$ generates a $2^k$-order multiplicative subgroup and $\beta$ generates the odd $t$-order multiplicative subgroup. Then every element $a \in \mathbb{F}_p^\times$ can be written as $\alpha^i \cdot \beta^j$ for $i \in \mathbb{Z}_{2^k}$ and $j \in \mathbb{Z}_t$. If the element is a square, then there exists some $b = \sqrt{a}$ which can be written $b = \alpha^{i'} \cdot \beta^{j'}$ for $i' \in \mathbb{Z}_{2^k}$ and $j' \in \mathbb{Z}_t$. This means that $a = b^2 = \alpha^{2i'} \cdot \beta^{2j'}$, therefore we have $i \equiv 2i' \pmod{2^k}$, and $j \equiv 2j' \pmod t$. $i$ would have to be even in this case because otherwise it would be impossible to have $i \equiv 2i' \pmod{2^k}$ for any $i'$. In the case that $a$ is not a square, then $i$ is odd, and so half of all elements are squares.

In order to compute the square root, we can first raise the element $a = \alpha^i \cdot \beta^j$ to the power $t$ to "kill" the $t$-order component, giving

$$a^t = \alpha^{it \pmod{2^k}} \cdot \beta^{jt \pmod t} = \alpha^{it \pmod{2^k}}$$

and then raise this result to the power $t^{-1} \pmod{2^k}$ to undo the effect of the original exponentiation on the $2^k$-order component:

$$(\alpha^{it \bmod 2^k})^{t^{-1} \pmod{2^k}} = \alpha^i$$

(since $t$ is relatively prime to $2^k$). This leaves bare the $\alpha^i$ value which we can trivially handle. We can similarly kill the $2^k$-order component to obtain $\beta^{j \cdot 2^{-1} \pmod t}$, and put the values together to obtain the square root.

It turns out that in the cases $k = 2, 3$ there are simpler algorithms that merge several of these exponentiations together for efficiency. For other values of $k$, the only known way is to manually extract $i$ by squaring until you obtain the identity for every single bit of $i$. This is the essence of the Tonelli-Shanks square root algorithm and describes the general strategy. (There is another square root algorithm that uses quadratic extension fields, but it doesn't pay off in efficiency until the prime becomes quite large.)

## Roots of unity

In the previous sections we wrote $p - 1 = 2^k \cdot t$ with $t$ odd, and stated that an element $\alpha \in \mathbb{F}_p^\times$ generated the $2^k$-order subgroup. For convenience, let's denote $n := 2^k$. The elements

$\{1, \alpha, \ldots, \alpha^{n-1}\}$ are known as the $n$th roots of unity.

The **primitive root of unity**, $\omega$, is an $n$th root of unity such that $\omega^i \neq 1$ except when $i \equiv 0 \pmod{n}$.

Important notes:

- If $\alpha$ is an $n$th root of unity, $\alpha$ satisfies $\alpha^n - 1 = 0$. If $\alpha \neq 1$, then

$$1 + \alpha + \alpha^2 + \cdots + \alpha^{n-1} = 0.$$

- Equivalently, the roots of unity are solutions to the equation

$$X^n - 1 = (X - 1)(X - \alpha)(X - \alpha^2) \cdots (X - \alpha^{n-1}).$$

- $\boxed{\omega^{\frac{n}{2}+i} = -\omega^i}$ **("Negation lemma")**. Proof:

$$\begin{aligned} \omega^n = 1 &\implies \omega^n - 1 = 0 \\ &\implies (\omega^{n/2} + 1)(\omega^{n/2} - 1) = 0. \end{aligned}$$

  Since the order of $\omega$ is $n$, $\omega^{n/2} \neq 1$. Therefore, $\omega^{n/2} = -1$.

- $\boxed{(\omega^{\frac{n}{2}+i})^2 = (\omega^i)^2}$ **("Halving lemma")**. Proof:

$$(\omega^{\frac{n}{2}+i})^2 = \omega^{n+2i} = \omega^n \cdot \omega^{2i} = \omega^{2i} = (\omega^i)^2.$$

  In other words, if we square each element in the $n$th roots of unity, we would get back only half the elements, $\{(\omega_n^i)^2\} = \{\omega_{n/2}\}$ (i.e. the $\frac{n}{2}$th roots of unity). There is a two-to-one mapping between the elements and their squares.

## References

[1] Friedman, R. (n.d.) "Cyclic Groups and Elementary Number Theory II" (p. 5).

# Polynomials

Let $A(X)$ be a polynomial over $\mathbb{F}_p$ with formal indeterminate $X$. As an example,

$$A(X) = a_0 + a_1 X + a_2 X^2 + a_3 X^3$$

defines a degree-$3$ polynomial. $a_0$ is referred to as the constant term. Polynomials of degree $n-1$ have $n$ coefficients. We will often want to compute the result of replacing the formal indeterminate $X$ with some concrete value $x$, which we denote by $A(x)$.

> In mathematics this is commonly referred to as "evaluating $A(X)$ at a point $x$". The word "point" here stems from the geometrical usage of polynomials in the form $y = A(x)$, where $(x, y)$ is the coordinate of a point in two-dimensional space. However, the polynomials we deal with are almost always constrained to equal zero, and $x$ will be an element of some field. This should not be confused with points on an elliptic curve, which we also make use of, but never in the context of polynomial evaluation.

Important notes:

- Multiplication of polynomials produces a product polynomial that is the sum of the degrees of its factors. Polynomial division subtracts from the degree.

$$\deg(A(X)B(X)) = \deg(A(X)) + \deg(B(X)),$$

$$\deg(A(X)/B(X)) = \deg(A(X)) - \deg(B(X)).$$

- Given a polynomial $A(X)$ of degree $n-1$, if we obtain $n$ evaluations of the polynomial at distinct points then these evaluations perfectly define the polynomial. In other words, given these evaluations we can obtain a unique polynomial $A(X)$ of degree $n-1$ via polynomial interpolation.
- $[a_0, a_1, \cdots, a_{n-1}]$ is the **coefficient representation** of the polynomial $A(X)$. Equivalently, we could use its **evaluation representation**

$$[(x_0, A(x_0)), (x_1, A(x_1)), \cdots, (x_{n-1}, A(x_{n-1}))]$$

  at $n$ distinct points. Either representation uniquely specifies the same polynomial.

> **(aside) Horner's rule**
>
> Horner's rule allows for efficient evaluation of a polynomial of degree $n-1$, using only $n-1$ multiplications and $n-1$ additions. It is the following identity:
>
> $$\begin{aligned} &a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1} \\ &\quad = a_0 + X\Big(a_1 + X\Big(a_2 + \cdots + X(a_{n-2} + X a_{n-1})\Big)\Big), \end{aligned}$$

# Fast Fourier Transform (FFT)

The FFT is an efficient way of converting between the coefficient and evaluation representations of a polynomial. It evaluates the polynomial at the $n$th roots of unity $\{\omega^0, \omega^1, \cdots, \omega^{n-1}\}$, where $\omega$ is a primitive $n$th root of unity. By exploiting symmetries in the roots of unity, each round of the FFT reduces the evaluation into a problem only half the size. Most commonly we use polynomials of length some power of two, $n = 2^k$, and apply the halving reduction recursively.

## Motivation: Fast polynomial multiplication

In the coefficient representation, it takes $O(n^2)$ operations to multiply two polynomials $A(X) \cdot B(X) = C(X)$:

$$
\begin{aligned}
A(X) &= a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1}, \\
B(X) &= b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}, \\
C(X) &= a_0 \cdot (b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}) \\
&\quad + a_1 X \cdot (b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}) \\
&\quad + \cdots \\
&\quad + a_{n-1} X^{n-1} \cdot (b_0 + b_1 X + b_2 X^2 + \cdots + b_{n-1} X^{n-1}),
\end{aligned}
$$

where each of the $n$ terms in the first polynomial has to be multiplied by the $n$ terms of the second polynomial.

In the evaluation representation, however, polynomial multiplication only requires $O(n)$ operations:

$$
\begin{aligned}
A &: \{(x_0, A(x_0)), (x_1, A(x_1)), \cdots, (x_{n-1}, A(x_{n-1}))\}, \\
B &: \{(x_0, B(x_0)), (x_1, B(x_1)), \cdots, (x_{n-1}, B(x_{n-1}))\}, \\
C &: \{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \cdots, (x_{n-1}, A(x_{n-1})B(x_{n-1}))\},
\end{aligned}
$$

where each evaluation is multiplied pointwise.

This suggests the following strategy for fast polynomial multiplication:

1. Evaluate polynomials at all $n$ points;
2. Perform fast pointwise multiplication in the evaluation representation ($O(n)$);
3. Convert back to the coefficient representation.

The challenge now is how to **evaluate** and **interpolate** the polynomials efficiently. Naively, evaluating a polynomial at $n$ points would require $O(n^2)$ operations (we use the $O(n)$ Horner's rule at each point):

$$\begin{bmatrix} A(1) \\ A(\omega) \\ A(\omega^2) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\cdot 2} & \dots & \omega^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

For convenience, we will denote the matrices above as:

$$\hat{\mathbf{A}} = \mathbf{V}_\omega \cdot \mathbf{A}.$$

($\hat{\mathbf{A}}$ is known as the *Discrete Fourier Transform* of $\mathbf{A}$; $\mathbf{V}_\omega$ is also called the *Vandermonde matrix*.)

## The (radix-2) Cooley-Tukey algorithm

Our strategy is to divide a DFT of size $n$ into two interleaved DFTs of size $n/2$. Given the polynomial $A(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1}$, we split it up into even and odd terms:

$$\begin{aligned} A_{\text{even}} &= a_0 + a_2 X + \cdots + a_{n-2} X^{\frac{n}{2}-1}, \\ A_{\text{odd}} &= a_1 + a_3 X + \cdots + a_{n-1} X^{\frac{n}{2}-1}. \end{aligned}$$

To recover the original polynomial, we do $A(X) = A_{\text{even}}(X^2) + X A_{\text{odd}}(X^2)$.

Trying this out on points $\omega_n^i$ and $\omega_n^{\frac{n}{2}+i}$, $i \in [0..\frac{n}{2} - 1]$, we start to notice some symmetries:

$$\begin{aligned} A(\omega_n^i) &= A_{\text{even}}((\omega_n^i)^2) + \omega_n^i A_{\text{odd}}((\omega_n^i)^2), \\ A(\omega_n^{\frac{n}{2}+i}) &= A_{\text{even}}((\omega_n^{\frac{n}{2}+i})^2) + \omega_n^{\frac{n}{2}+i} A_{\text{odd}}((\omega_n^{\frac{n}{2}+i})^2) \\ &= A_{\text{even}}((-\omega_n^i)^2) - \omega_n^i A_{\text{odd}}((-\omega_n^i)^2) \leftarrow (\text{negation lemma}) \\ &= A_{\text{even}}((\omega_n^i)^2) - \omega_n^i A_{\text{odd}}((\omega_n^i)^2). \end{aligned}$$

Notice that we are only evaluating $A_{\text{even}}(X)$ and $A_{\text{odd}}(X)$ over half the domain $\{(\omega_n^0)^2, (\omega_n)^2, \cdots, (\omega_n^{\frac{n}{2}-1})^2\} = \{\omega_{n/2}^i\}, i = [0..\frac{n}{2} - 1]$ (halving lemma). This gives us all the terms we need to reconstruct $A(X)$ over the full domain $\{\omega^0, \omega, \cdots, \omega^{n-1}\}$: which means we have transformed a length-$n$ DFT into two length-$\frac{n}{2}$ DFTs.

We choose $n = 2^k$ to be a power of two (by zero-padding if needed), and apply this divide-and-conquer strategy recursively. By the Master Theorem[1], this gives us an evaluation algorithm with $O(n \log_2 n)$ operations, also known as the Fast Fourier Transform (FFT).

### Inverse FFT

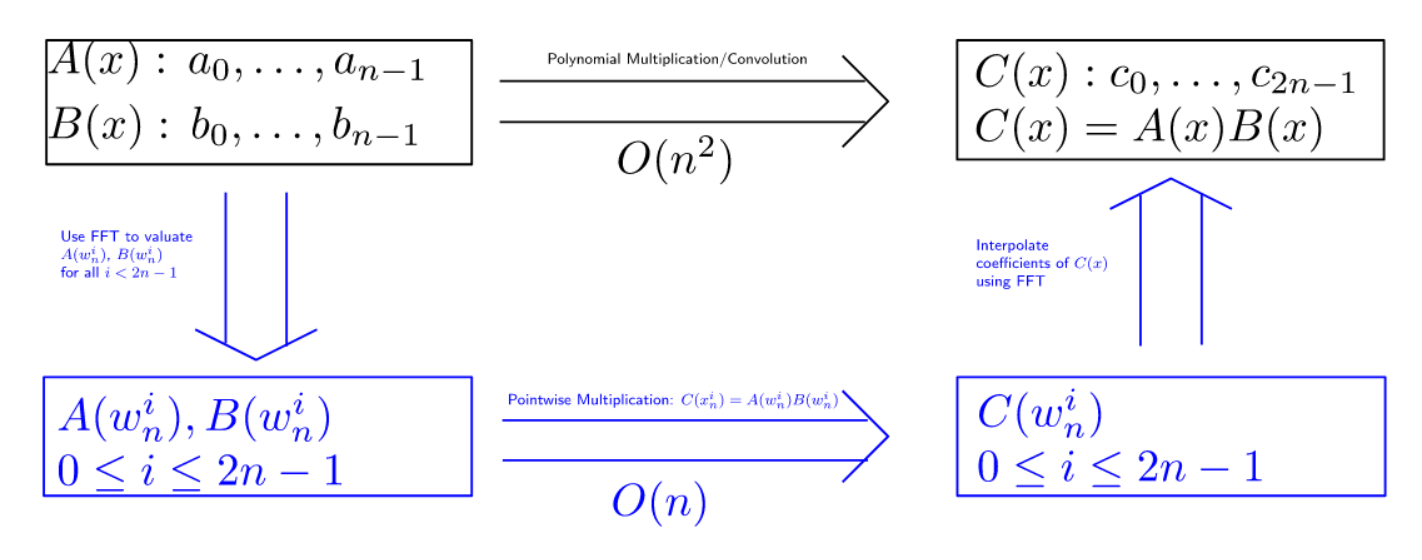
So we've evaluated our polynomials and multiplied them pointwise. What remains is to convert the product from the evaluation representation back to coefficient representation. To do this, we simply call the FFT on the evaluation representation. However, this time we also:

- replace $\omega^i$ by $\omega^{-i}$ in the Vandermonde matrix, and
- multiply our final result by a factor of $1/n$.

In other words:

$$\mathbf{A} = \frac{1}{n}\mathbf{V}_{\omega^{-1}} \cdot \hat{\mathbf{A}}.$$

(To understand why the inverse FFT has a similar form to the FFT, refer to Slide 13-1 of [2]. The below image was also taken from [2].)




## The Schwartz-Zippel lemma

The Schwartz-Zippel lemma informally states that "different polynomials are different at most points." Formally, it can be written as follows:

Let $p(x_1, x_2, \cdots, x_n)$ be a nonzero polynomial of $n$ variables with degree $d$. Let $S$ be a finite set of numbers with at least $d$ elements in it. If we choose random $\alpha_1, \alpha_2, \cdots, \alpha_n$ from $S$,

$$\Pr[p(\alpha_1, \alpha_2, \cdots, \alpha_n) = 0] \leq \frac{d}{|S|}.$$

In the familiar univariate case $p(X)$, this reduces to saying that a nonzero polynomial of degree $d$ has at most $d$ roots.

The Schwartz-Zippel lemma is used in polynomial equality testing. Given two multi-variate polynomials $p_1(x_1, \cdots, x_n)$ and $p_2(x_1, \cdots, x_n)$ of degrees $d_1, d_2$ respectively, we can test if $p_1(\alpha_1, \cdots, \alpha_n) - p_2(\alpha_1, \cdots, \alpha_n) = 0$ for random $\alpha_1, \cdots, \alpha_n \leftarrow S$, where the size of $S$ is at least $|S| \geq (d_1 + d_2)$. If the two polynomials are identical, this will always be true, whereas if the two polynomials are different then the equality holds with probability at most $\frac{\max(d_1, d_2)}{|S|}$.

## Vanishing polynomial

Consider the order-$n$ multiplicative subgroup $\mathcal{H}$ with primitive root of unity $\omega$. For all $\omega^i \in \mathcal{H}, i \in [n-1]$, we have $(\omega^i)^n = (\omega^n)^i = (\omega^0)^i = 1$. In other words, every element of $\mathcal{H}$ fulfils the equation

$$\begin{aligned} Z_H(X) &= X^n - 1 \\ &= (X - \omega^0)(X - \omega^1)(X - \omega^2) \cdots (X - \omega^{n-1}), \end{aligned}$$

meaning every element is a root of $Z_H(X)$. We call $Z_H(X)$ the **vanishing polynomial** over $\mathcal{H}$ because it evaluates to zero on all elements of $\mathcal{H}$.

This comes in particularly handy when checking polynomial constraints. For instance, to check that $A(X) + B(X) = C(X)$ over $\mathcal{H}$, we simply have to check that $A(X) + B(X) - C(X)$ is some multiple of $Z_H(X)$. In other words, if dividing our constraint by the vanishing polynomial still yields some polynomial $\frac{A(X)+B(X)-C(X)}{Z_H(X)} = H(X)$, we are satisfied that $A(X) + B(X) - C(X) = 0$ over $\mathcal{H}$.

# Lagrange basis functions

TODO: explain what a basis is in general (briefly).

Polynomials are commonly written in the monomial basis (e.g. $X, X^2, ...X^n$). However, when working over a multiplicative subgroup of order $n$, we find a more natural expression in the Lagrange basis.

Consider the order-$n$ multiplicative subgroup $\mathcal{H}$ with primitive root of unity $\omega$. The Lagrange basis corresponding to this subgroup is a set of functions $\{\mathcal{L}_i\}_{i=0}^{n-1}$, where

$$\mathcal{L}_i(\omega^j) = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

We can write this more compactly as $\mathcal{L}_i(\omega^j) = \delta_{ij}$, where $\delta$ is the Kronecker delta function.

Now, we can write our polynomial as a linear combination of Lagrange basis functions,

$$A(X) = \sum_{i=0}^{n-1} a_i \mathcal{L}_i(X), X \in \mathcal{H},$$

which is equivalent to saying that $A(X)$ evaluates to $a_0$ at $\omega^0$, to $a_1$ at $\omega^1$, to $a_2$ at $\omega^2, \cdots$, and so on.

When working over a multiplicative subgroup, the Lagrange basis function has a convenient sparse representation of the form

$$\mathcal{L}_i(X) = \frac{c_i \cdot (X^n - 1)}{X - \omega^i},$$

where $c_i$ is the barycentric weight. (To understand how this form was derived, refer to [3].) For $i = 0$, we have $c = 1/n \implies \mathcal{L}_0(X) = \frac{1}{n}\frac{(X^n-1)}{X-1}$.

Suppose we are given a set of evaluation points $\{x_0, x_1, \cdots, x_{n-1}\}$. Since we cannot assume that the $x_i$'s form a multiplicative subgroup, we consider also the Lagrange polynomials $\mathcal{L}_i$'s in the general case. Then we can construct:

$$\mathcal{L}_i(X) = \prod_{j \neq i} \frac{X - x_j}{x_i - x_j}, i \in [0..n-1].$$

Here, every $X = x_j \neq x_i$ will produce a zero numerator term $(x_j - x_j)$, causing the whole product to evaluate to zero. On the other hand, $X = x_i$ will evaluate to $\frac{x_i - x_j}{x_i - x_j}$ at every term,

resulting in an overall product of one. This gives the desired Kronecker delta behaviour $\mathcal{L}_i(x_j) = \delta_{ij}$ on the set $\{x_0, x_1, \cdots, x_{n-1}\}$.

### Lagrange interpolation

Given a polynomial in its evaluation representation

$$A : \{(x_0, A(x_0)), (x_1, A(x_1)), \cdots, (x_{n-1}, A(x_{n-1}))\},$$

we can reconstruct its coefficient form in the Lagrange basis:

$$A(X) = \sum_{i=0}^{n-1} A(x_i)\mathcal{L}_i(X),$$

where $X \in \{x_0, x_1, \cdots, x_{n-1}\}$.

## References

[1] Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). "Algorithms" (ch. 2). New York: McGraw-Hill Higher Education.

[2] Golin, M. (2016). "The Fast Fourier Transform and Polynomial Multiplication" [lecture notes], COMP 3711H Design and Analysis of Algorithms, Hong Kong University of Science and Technology.

[3] Berrut, J. and Trefethen, L. (2004). "Barycentric Lagrange Interpolation."

# Cryptographic groups

In the section Inverses and groups we introduced the concept of *groups*. A group has an identity and a group operation. In this section we will write groups additively, i.e. the identity is $\mathcal{O}$ and the group operation is $+$.

Some groups can be used as *cryptographic groups*. At the risk of oversimplifying, this means that the problem of finding a discrete logarithm of a group element $P$ to a given base $G$, i.e. finding $x$ such that $P = [x]G$, is hard in general.

# Pedersen commitment

The Pedersen commitment [P99] is a way to commit to a secret message in a verifiable way. It uses two random public generators $G, H \in \mathbb{G}$, where $\mathbb{G}$ is a cryptographic group of order $q$. A random secret $r$ is chosen in $\mathbb{Z}_q$, and the message to commit to $m$ is from any subset of $\mathbb{Z}_q$. The commitment is

$$c = \text{Commit}(m, r) = [m]G + [r]H.$$

To open the commitment, the committer reveals $m$ and $r$, thus allowing anyone to verify that $c$ is indeed a commitment to $m$.

Notice that the Pedersen commitment scheme is homomorphic:

$$\begin{aligned}\text{Commit}(m, r) + \text{Commit}(m', r') &= [m]G + [r]H + [m']G + [r']H \\ &= [m + m']G + [r + r']H \\ &= \text{Commit}(m + m', r + r').\end{aligned}$$

Assuming the discrete log assumption holds, Pedersen commitments are also perfectly hiding and computationally binding:

- **hiding**: the adversary chooses messages $m_0, m_1$. The committer commits to one of these messages $c = \text{Commit}(m_b, r), b \in \{0, 1\}$. Given $c$, the probability of the adversary guessing the correct $b$ is no more than $\frac{1}{2}$.
- **binding**: the adversary cannot pick two different messages $m_0 \neq m_1$, and randomness $r_0, r_1$, such that $\text{Commit}(m_0, r_0) = \text{Commit}(m_1, r_1)$.

## Vector Pedersen commitment

We can use a variant of the Pedersen commitment scheme to commit to multiple messages at once, $\mathbf{m} = (m_0, \cdots, m_{n-1})$. This time, we'll have to sample a corresponding number of random public generators $\mathbf{G} = (G_0, \cdots, G_{n-1})$, along with a single random generator $H$ as before (for use in hiding). Then, our commitment scheme is:

$$\begin{aligned}\text{Commit}(\mathbf{m}; r) &= \text{Commit}((m_0, \cdots, m_{n-1}); r) \\ &= [r]H + [m_0]G_0 + \cdots + [m_{n-1}]G_{n-1} \\ &= [r]H + \sum_{i=0}^{n-1} [m_i]G_i.\end{aligned}$$

TODO: is this positionally binding?

## Diffie–Hellman

An example of a protocol that uses cryptographic groups is Diffie–Hellman key agreement [DH1976]. The Diffie–Hellman protocol is a method for two users, Alice and Bob, to generate a shared private key. It proceeds as follows:

1. Alice and Bob publicly agree on two prime numbers, $p$ and $G$, where $p$ is large and $G$ is a primitive root $\pmod p$. (Note that $g$ is a generator of the group $\mathbb{F}_p^\times$.)
2. Alice chooses a large random number $a$ as her private key. She computes her public key $A = [a]G \pmod p$, and sends $A$ to Bob.
3. Similarly, Bob chooses a large random number $b$ as his private key. He computes his public key $B = [b]G \pmod p$, and sends $B$ to Alice.
4. Now both Alice and Bob compute their shared key $K = [ab]G \pmod p$, which Alice computes as

$$K = [a]B \pmod p = [a]([b]G) \pmod p,$$

   and Bob computes as

$$K = [b]A \pmod p = [b]([a]G) \pmod p.$$

A potential eavesdropper would need to derive $K = [ab]g \pmod p$ knowing only $g, p, A = [a]G$, and $B = [b]G$: in other words, they would need to either get the discrete logarithm $a$ from $A = [a]G$ or $b$ from $B = [b]G$, which we assume to be computationally infeasible in $\mathbb{F}_p^\times$.

More generally, protocols that use similar ideas to Diffie–Hellman are used throughout cryptography. One way of instantiating a cryptographic group is as an elliptic curve. Before we go into detail on elliptic curves, we'll describe some algorithms that can be used for any group.

## Multiscalar multiplication

### TODO: Pippenger's algorithm

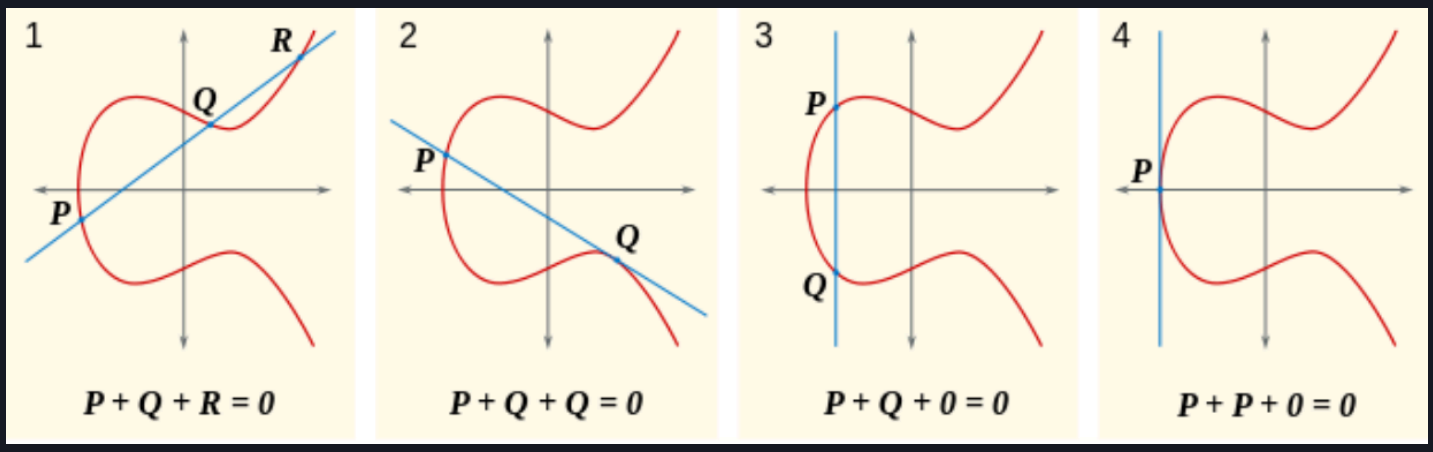
Reference: https://jbootle.github.io/Misc/pippenger.pdf

# Elliptic curves

Elliptic curves constructed over finite fields are another important cryptographic tool.

We use elliptic curves because they provide a cryptographic group, i.e. a group in which the discrete logarithm problem (discussed below) is hard.

There are several ways to define the curve equation, but for our purposes, let $\mathbb{F}_p$ be a large (255-bit) field, and then let the set of solutions $(x, y)$ to $y^2 = x^3 + b$ for some constant $b$ define the $\mathbb{F}_p$-rational points on an elliptic curve $E(\mathbb{F}_p)$. These $(x, y)$ coordinates are called "affine coordinates". Each of the $\mathbb{F}_p$-rational points, together with a "point at infinity" $\mathcal{O}$ that serves as the group identity, can be interpreted as an element of a group. By convention, elliptic curve groups are written additively.




*"Three points on a line sum to zero, which is the point at infinity."*

The group addition law is simple: to add two points together, find the line that intersects both points and obtain the third point, and then negate its $y$-coordinate. The case that a point is being added to itself, called point doubling, requires special handling: we find the line tangent to the point, and then find the single other point that intersects this line and then negate. Otherwise, in the event that a point is being "added" to its negation, the result is the point at infinity.

The ability to add and double points naturally gives us a way to scale them by integers, called *scalars*. The number of points on the curve is the group order. If this number is a prime $q$, then the scalars can be considered as elements of a *scalar field*, $\mathbb{F}_q$.

Elliptic curves, when properly designed, have an important security property. Given two random elements $G, H \in E(\mathbb{F}_p)$ finding $a$ such that $[a]G = H$, otherwise known as the discrete log of $H$ with respect to $G$, is considered computationally infeasible with classical computers. This is called the elliptic curve discrete log assumption.

If an elliptic curve group $\mathbb{G}$ has prime order $q$ (like the ones used in Halo 2), then it is a finite cyclic group. Recall from the section on groups that this implies it is isomorphic to $\mathbb{Z}/q\mathbb{Z}$, or equivalently, to the scalar field $\mathbb{F}_q$. Each possible generator $G$ fixes the isomorphism; then an element on the scalar side is precisely the discrete log of the corresponding group element with respect to $G$. In the case of a cryptographically secure elliptic curve, the isomorphism is hard to compute in the $\mathbb{G} \to \mathbb{F}_q$ direction because the elliptic curve discrete log problem is hard.

> It is sometimes helpful to make use of this isomorphism by thinking of group-based cryptographic protocols and algorithms in terms of the scalars instead of in terms of the group elements. This can make proofs and notation simpler.
>
> For instance, it has become common in papers on proof systems to use the notation $[x]$ to denote a group element with discrete log $x$, where the generator is implicit.
>
> We also used this idea in the "distinct-x theorem", in order to prove correctness of optimizations for elliptic curve scalar multiplication in Sapling, and an endomorphism-based optimization in Appendix C of the original Halo paper.

# Curve arithmetic

## Point doubling

The simplest situation is doubling a point $(x_0, y_0)$. Continuing with our example $y^2 = x^3 + b$, this is done first by computing the derivative

$$\lambda = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{3x^2}{2y}.$$

To obtain expressions for $(x_1, y_1) = (x_0, y_0) + (x_0, y_0)$, we consider

$$\begin{aligned} \frac{-y_1 - y_0}{x_1 - x_0} = \lambda &\implies -y_1 = \lambda(x_1 - x_0) + y_0 \\ &\implies \boxed{y_1 = \lambda(x_0 - x_1) - y_0}. \end{aligned}$$

To get the expression for $x_1$, we substitute $y = \lambda(x_0 - x) - y_0$ into the elliptic curve equation:

$$\begin{aligned} y^2 = x^3 + b &\implies (\lambda(x_0 - x) - y_0)^2 = x^3 + b \\ &\implies x^3 - \lambda^2 x^2 + \cdots = 0 \leftarrow (\text{rearranging terms}) \\ &= (x - x_0)(x - x_0)(x - x_1) \leftarrow (\text{known roots } x_0, x_0, x_1) \\ &= x^3 - (x_0 + x_0 + x_1)x^2 + \cdots . \end{aligned}$$

Comparing coefficients for the $x^2$ term gives us $\lambda^2 = x_0 + x_0 + x_1 \implies \boxed{x_1 = \lambda^2 - 2x_0}$.

## Projective coordinates

This unfortunately requires an expensive inversion of $2y$. We can avoid this by arranging our equations to "defer" the computation of the inverse, since we often do not need the actual affine $(x', y')$ coordinate of the resulting point immediately after an individual curve operation. Let's introduce a third coordinate $Z$ and scale our curve equation by $Z^3$ like so:

$$Z^3 y^2 = Z^3 x^3 + Z^3 b$$

Our original curve is just this curve at the restriction $Z = 1$. If we allow the affine point $(x, y)$ to be represented by $X = xZ$, $Y = yZ$ and $Z \neq 0$ then we have the homogenous projective curve

$$Y^2 Z = X^3 + Z^3 b.$$

Obtaining $(x, y)$ from $(X, Y, Z)$ is as simple as computing $(X/Z, Y/Z)$ when $Z \neq 0$. (When $Z = 0$, we are dealing with the point at infinity $O := (0 : 1 : 0)$.) In this form, we now have a convenient way to defer the inversion required by doubling a point. The general strategy is to express $x', y'$ as rational functions using $x = X/Z$ and $y = Y/Z$, rearrange to make their denominators the same, and then take the resulting point $(X, Y, Z)$ to have $Z$ be the shared denominator and $X = x'Z, Y = y'Z$.

> Projective coordinates are often, but not always, more efficient than affine coordinates. There may be exceptions to this when either we have a different way to apply Montgomery's trick, or when we're in the circuit setting where multiplications and inversions are about equally as expensive (at least in terms of circuit size).

The following shows an example of doubling a point $(X, Y, Z) = (xZ, yZ, Z)$ without an inversion. Substituting with $X, Y, Z$ gives us

$$\lambda = \frac{3x^2}{2y} = \frac{3(X/Z)^2}{2(Y/Z)} = \frac{3X^2}{2YZ}$$

and gives us

$$\begin{aligned}
x' &= \lambda^2 - 2x \\
&= \lambda^2 - \frac{2X}{Z} \\
&= \frac{9X^4}{4Y^2Z^2} - \frac{2X}{Z} \\
&= \frac{9X^4 - 8XY^2Z}{4Y^2Z^2} \\
&= \frac{18X^4YZ - 16XY^3Z^2}{8Y^3Z^3}
\end{aligned}$$

$$\begin{aligned}
y' &= \lambda(x - x') - y \\
&= \lambda(\frac{X}{Z} - \frac{9X^4 - 8XY^2Z}{4Y^2Z^2}) - \frac{Y}{Z} \\
&= \frac{3X^2}{2YZ}(\frac{X}{Z} - \frac{9X^4 - 8XY^2Z}{4Y^2Z^2}) - \frac{Y}{Z} \\
&= \frac{3X^3}{2YZ^2} - \frac{27X^6 - 24X^3Y^2Z}{8Y^3Z^3} - \frac{Y}{Z} \\
&= \frac{12X^3Y^2Z - 8Y^4Z^2 - 27X^6 + 24X^3Y^2Z}{8Y^3Z^3}
\end{aligned}$$

Notice how the denominators of $x'$ and $y'$ are the same. Thus, instead of computing $(x', y')$ we can compute $(X, Y, Z)$ with $Z = 8Y^3Z^3$ and $X, Y$ set to the corresponding numerators such that $X/Z = x'$ and $Y/Z = y'$. This completely avoids the need to perform an inversion when doubling, and something analogous to this can be done when adding two distinct points.

## Point addition

We now add two points with distinct $x$-coordinates, $P = (x_0, y_0)$ and $Q = (x_1, y_1)$, where $x_0 \neq x_1$, to obtain $R = P + Q = (x_2, y_2)$. The line $\overline{PQ}$ has slope

$$\lambda = \frac{y_1 - y_0}{x_1 - x_0} \implies y - y_0 = \lambda \cdot (x - x_0).$$

Using the expression for $\overline{PQ}$, we compute $y$-coordinate $-y_2$ of $-R$ as:

$$-y_2 - y_0 = \lambda \cdot (x_2 - x_0) \implies \boxed{y_2 = \lambda(x_0 - x_2) - y_0}.$$

Plugging the expression for $\overline{PQ}$ into the curve equation $y^2 = x^3 + b$ yields

$$\begin{aligned} y^2 = x^3 + b &\implies (\lambda \cdot (x - x_0) + y_0)^2 = x^3 + b \\ &\implies x^3 - \lambda^2 x^2 + \cdots = 0 \leftarrow (\text{rearranging terms}) \\ &= (x - x_0)(x - x_1)(x - x_2) \leftarrow (\text{known roots } x_0, x_1, x_2) \\ &= x^3 - (x_0 + x_1 + x_2)x^2 + \cdots . \end{aligned}$$

Comparing coefficients for the $x^2$ term gives us $\lambda^2 = x_0 + x_1 + x_2 \implies$ $\boxed{x_2 = \lambda^2 - x_0 - x_1}$.

---

Important notes:

- There exist efficient formulae[1] for point addition that do not have edge cases (so-called "complete" formulae) and that unify the addition and doubling cases together. The result of adding a point to its negation using those formulae produces $Z = 0$, which represents the point at infinity.
- In addition, there are other models like the Jacobian representation where $(x, y) = (xZ^2, yZ^3, Z)$ where the curve is rescaled by $Z^6$ instead of $Z^3$, and this representation has even more efficient arithmetic but no unified/complete formulae.
- We can easily compare two curve points $(X_1, Y_1, Z_1)$ and $(X_2, Y_2, Z_2)$ for equality in the homogenous projective coordinate space by "homogenizing" their Z-coordinates; the checks become $X_1 Z_2 = X_2 Z_1$ and $Y_1 Z_2 = Y_2 Z_1$.

## Curve endomorphisms

Imagine that $\mathbb{F}_p$ has a primitive cube root of unity, or in other words that $3|p - 1$ and so an element $\zeta_p$ generates a $3$-order multiplicative subgroup. Notice that a point $(x, y)$ on our example elliptic curve $y^2 = x^3 + b$ has two cousin points: $(\zeta_p x, y), (\zeta_p^2 x, y)$, because the computation $x^3$ effectively kills the $\zeta$ component of the $x$-coordinate. Applying the map $(x, y) \mapsto (\zeta_p x, y)$ is an application of an endomorphism over the curve. The exact mechanics involved are complicated, but when the curve has a prime $q$ number of points (and thus a prime "order") the effect of the endomorphism is to multiply the point by a scalar in $\mathbb{F}_q$ which is also a primitive cube root $\zeta_q$ in the scalar field.

## Curve point compression

Given a point on the curve $P = (x, y)$, we know that its negation $-P = (x, -y)$ is also on the curve. To uniquely specify a point, we need only encode its $x$-coordinate along with the sign of

its $y$-coordinate.

### Serialization

As mentioned in the Fields section, we can interpret the least significant bit of a field element as its "sign", since its additive inverse will always have the opposite LSB. So we record the LSB of the $y$-coordinate as `sign`.

Pallas and Vesta are defined over the $\mathbb{F}_p$ and $\mathbb{F}_q$ fields, which elements can be expressed in $255$ bits. This conveniently leaves one unused bit in a 32-byte representation. We pack the $y$-coordinate `sign` bit into the highest bit in the representation of the $x$-coordinate:

```
         <---------------------------------- x ---------------------------------->
Enc(P) = [_ _ _ _ _ _ _ _] [_ _ _ _ _ _ _ _] ... [_ _ _ _ _ _ _ _] [_ _ _ _ _ _ _
sign]
          ^                <------------------------------------>
^
         LSB                                30 bytes
MSB
```

The "point at infinity" $\mathcal{O}$ that serves as the group identity, does not have an affine $(x, y)$ representation. However, it turns out that there are no points on either the Pallas or Vesta curve with $x = 0$ or $y = 0$. We therefore use the "fake" affine coordinates $(0, 0)$ to encode $\mathcal{O}$, which results in the all-zeroes 32-byte array.

### Deserialization

When deserializing a compressed curve point, we first read the most significant bit as `ysign`, the sign of the $y$-coordinate. Then, we set this bit to zero to recover the original $x$-coordinate.

If $x = 0, y = 0$, we return the "point at infinity" $\mathcal{O}$. Otherwise, we proceed to compute $y = \sqrt{x^3 + b}$. Here, we read the least significant bit of $y$ as `sign`. If `sign == ysign`, we already have the correct sign and simply return the curve point $(x, y)$. Otherwise, we negate $y$ and return $(x, -y)$.

## Cycles of curves

Let $E_p$ be an elliptic curve over a finite field $\mathbb{F}_p$, where $p$ is a prime. We denote this by $E_p/\mathbb{F}_p$. and we denote the group of points of $E_p$ over $\mathbb{F}_p$, with order $q = \#E(\mathbb{F}_p)$. For this curve, we

call $\mathbb{F}_p$ the "base field" and $\mathbb{F}_q$ the "scalar field".

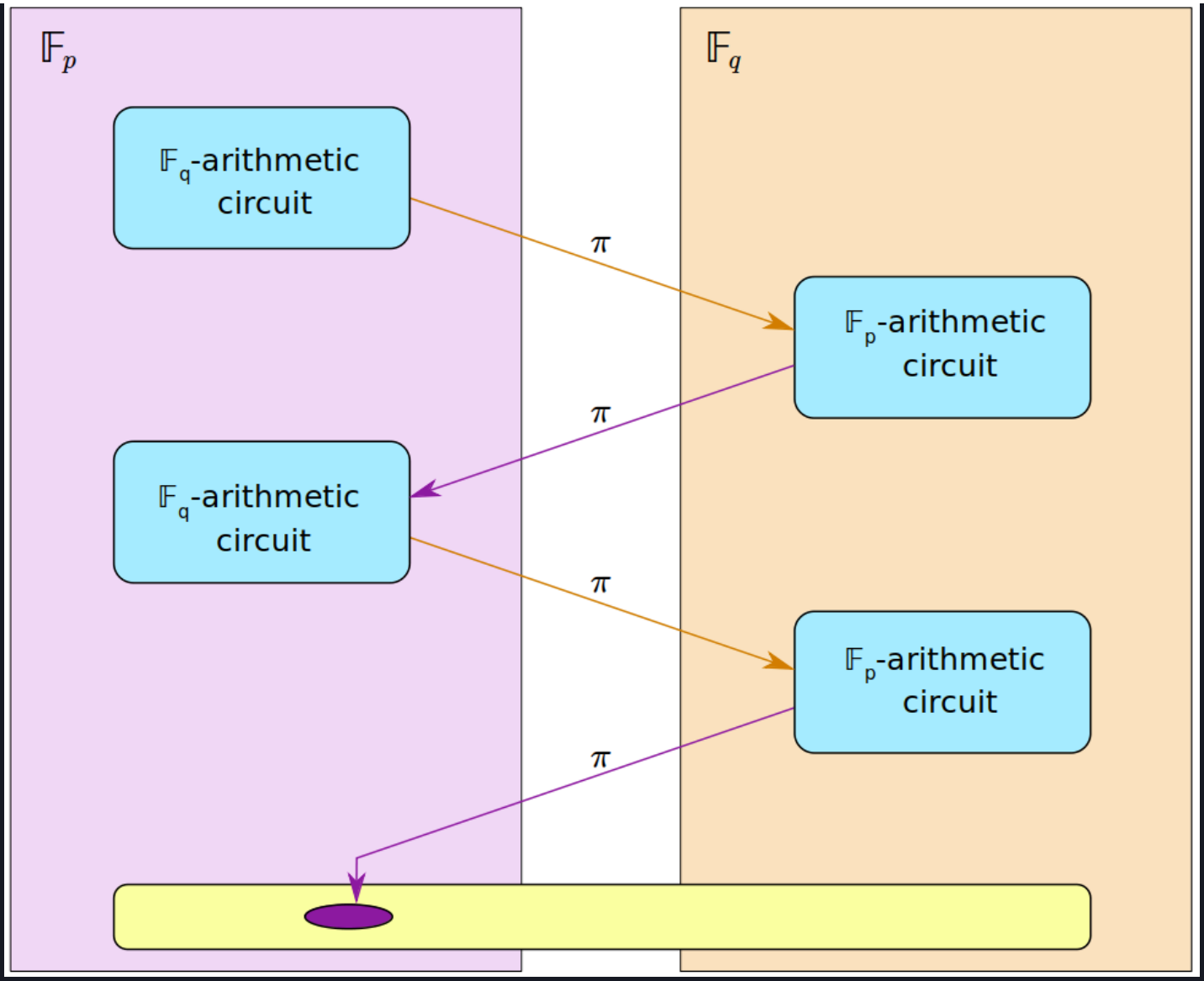
We instantiate our proof system over the elliptic curve $E_p/\mathbb{F}_p$. This allows us to prove statements about $\mathbb{F}_q$-arithmetic circuit satisfiability.

> **(aside) If our curve $E_p$ is over $\mathbb{F}_p$, why is the arithmetic circuit instead in $\mathbb{F}_q$?** The proof system is basically working on encodings of the scalars in the circuit (or more precisely, commitments to polynomials whose coefficients are scalars). The scalars are in $\mathbb{F}_q$ when their encodings/commitments are elliptic curve points in $E_p/\mathbb{F}_p$.

However, most of the verifier's arithmetic computations are over the base field $\mathbb{F}_p$, and are thus efficiently expressed as an $\mathbb{F}_p$-arithmetic circuit.

> **(aside) Why are the verifier's computations (mainly) over $\mathbb{F}_p$?** The Halo 2 verifier actually has to perform group operations using information output by the circuit. Group operations like point doubling and addition use arithmetic in $\mathbb{F}_p$, because the coordinates of points are in $\mathbb{F}_p$.

This motivates us to construct another curve with scalar field $\mathbb{F}_p$, which has an $\mathbb{F}_p$-arithmetic circuit that can efficiently verify proofs from the first curve. As a bonus, if this second curve had base field $E_q/\mathbb{F}_q$, it would generate proofs that could be efficiently verified in the first curve's $\mathbb{F}_q$-arithmetic circuit. In other words, we instantiate a second proof system over $E_q/\mathbb{F}_q$, forming a 2-cycle with the first:




**TODO: Pallas-Vesta curves**

Reference: https://github.com/zcash/pasta

## Hashing to curves

Sometimes it is useful to be able to produce a random point on an elliptic curve $E_p/\mathbb{F}_p$ corresponding to some input, in such a way that no-one will know its discrete logarithm (to any other base).

This is described in detail in the Internet draft on Hashing to Elliptic Curves. Several algorithms can be used depending on efficiency and security requirements. The framework used in the Internet Draft makes use of several functions:

- `hash_to_field` : takes a byte sequence input and maps it to a element in the base field $\mathbb{F}_p$
- `map_to_curve` : takes an $\mathbb{F}_p$ element and maps it to $E_p$.

### TODO: Simplified SWU

Reference: https://eprint.iacr.org/2019/403.pdf

## References

[1] Renes, J., Costello, C., & Batina, L. (2016, May). "Complete addition formulas for prime order elliptic curves." In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 403-428). Springer, Berlin, Heidelberg.

# Polynomial commitment using inner product argument

We want to commit to some polynomial $p(X) \in \mathbb{F}_p[X]$, and be able to provably evaluate the committed polynomial at arbitrary points. The naive solution would be for the prover to simply send the polynomial's coefficients to the verifier: however, this requires $O(n)$ communication. Our polynomial commitment scheme gets the job done using $O(\log n)$ communication.

### `Setup`

Given a parameter $d = 2^k$, we generate the common reference string $\sigma = (\mathbb{G}, \mathbf{G}, H, \mathbb{F}_p)$ defining certain constants for this scheme:

- $\mathbb{G}$ is a group of prime order $p$;
- $\mathbf{G} \in \mathbb{G}^d$ is a vector of $d$ random group elements;
- $H \in \mathbb{G}$ is a random group element; and
- $\mathbb{F}_p$ is the finite field of order $p$.

### `Commit`

The Pedersen vector commitment $\mathrm{Commit}$ is defined as

$$\mathrm{Commit}(\sigma, p(X); r) = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H,$$

for some polynomial $p(X) \in \mathbb{F}_p[X]$ and some blinding factor $r \in \mathbb{F}_p$. Here, each element of the vector $\mathbf{a}_i \in \mathbb{F}_p$ is the coefficient for the $i$th degree term of $p(X)$, and $p(X)$ is of maximal degree $d - 1$.

### `Open` (prover) and `OpenVerify` (verifier)

The modified inner product argument is an argument of knowledge for the relation

$$\boxed{\{((P, x, v); (\mathbf{a}, r)) : P = \langle \mathbf{a}, \mathbf{G} \rangle + [r]H, v = \langle \mathbf{a}, \mathbf{b} \rangle\}},$$

where $\mathbf{b} = (1, x, x^2, \cdots, x^{d-1})$ is composed of increasing powers of the evaluation point $x$. This allows a prover to demonstrate to a verifier that the polynomial contained "inside" the commitment $P$ evaluates to $v$ at $x$, and moreover, that the committed polynomial has maximum degree $d - 1$.

The inner product argument proceeds in $k = \log_2 d$ rounds. For our purposes, it is sufficient to know about its final outputs, while merely providing intuition about the intermediate rounds. (Refer to Section 3 in the Halo paper for a full explanation.)

Before beginning the argument, the verifier selects a random group element $U$ and sends it to the prover. We initialize the argument at round $k$, with the vectors $\mathbf{a}^{(k)} := \mathbf{a}$, $\mathbf{G}^{(k)} := \mathbf{G}$ and $\mathbf{b}^{(k)} := \mathbf{b}$. In each round $j = k, k-1, \cdots, 1$:

- the prover computes two values $L_j$ and $R_j$ by taking some inner product of $\mathbf{a}^{(j)}$ with $\mathbf{G}^{(j)}$ and $\mathbf{b}^{(j)}$. Note that are in some sense "cross-terms": the lower half of $\mathbf{a}$ is used with the higher half of $\mathbf{G}$ and $\mathbf{b}$, and vice versa:

$$L_j = \langle \mathbf{a}^{(j)}_{\mathbf{lo}}, \mathbf{G}^{(j)}_{\mathbf{hi}} \rangle + [l_j]H + [\langle \mathbf{a}^{(j)}_{\mathbf{lo}}, \mathbf{b}^{(j)}_{\mathbf{hi}} \rangle]U$$
$$R_j = \langle \mathbf{a}^{(j)}_{\mathbf{hi}}, \mathbf{G}^{(j)}_{\mathbf{lo}} \rangle + [r_j]H + [\langle \mathbf{a}^{(j)}_{\mathbf{hi}}, \mathbf{b}^{(j)}_{\mathbf{lo}} \rangle]U$$

- the verifier issues a random challenge $u_j$;
- the prover uses $u_j$ to compress the lower and higher halves of $\mathbf{a}^{(j)}$, thus producing a new vector of half the original length

$$\mathbf{a}^{(j-1)} = \mathbf{a}^{(j)}_{\mathbf{hi}} + \mathbf{a}^{(j)}_{\mathbf{lo}} \cdot u_j^{-1}.$$

  The vectors $\mathbf{G}^{(j)}$ and $\mathbf{b}^{(j)}$ are similarly compressed to give $\mathbf{G}^{(j-1)}$ and $\mathbf{b}^{(j-1)}$ (using $u_j$ instead of $u_j^{-1}$).
- $\mathbf{a}^{(j-1)}$, $\mathbf{G}^{(j-1)}$ and $\mathbf{b}^{(j-1)}$ are input to the next round $j - 1$.

Note that at the end of the last round $j = 1$, we are left with $a := \mathbf{a}^{(0)}$, $G := \mathbf{G}^{(0)}$, $b := \mathbf{b}^{(0)}$, each of length 1. The intuition is that these final scalars, together with the challenges $\{u_j\}$ and "cross-terms" $\{L_j, R_j\}$ from each round, encode the compression in each round. Since the prover did not know the challenges $U, \{u_j\}$ in advance, they would have been unable to manipulate the round compressions. Thus, checking a constraint on these final terms should enforce that the compression had been performed correctly, and that the original $\mathbf{a}$ satisfied the relation before undergoing compression.

Note that $G, b$ are simply rearrangements of the publicly known $\mathbf{G}, \mathbf{b}$, with the round challenges $\{u_j\}$ mixed in: this means the verifier can compute $G, b$ independently and verify that the prover had provided those same values.

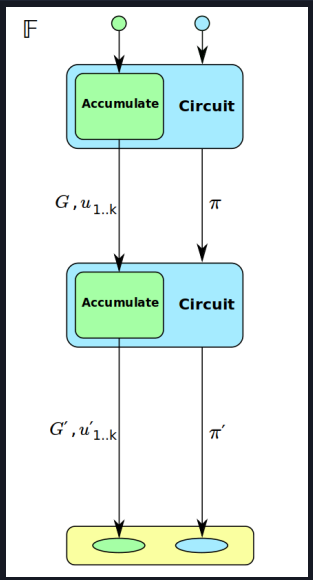
## Recursion

Alternative terms: Induction; Accumulation scheme; Proof-carrying data

However, the computation of $G$ requires a length-$2^k$ multiexponentiation $\langle \mathbf{G}, \mathbf{s} \rangle$, where $\mathbf{s}$ is composed of the round challenges $u_1, \cdots, u_k$ arranged in a binary counting structure. This is the linear-time computation that we want to amortise across a batch of proof instances. Instead of computing $G$, notice that we can express $G$ as a commitment to a polynomial

$$G = \text{Commit}(\sigma, g(X, u_1, \cdots, u_k)),$$

where $g(X, u_1, \cdots, u_k) := \prod_{i=1}^{k} (u_i + u_i^{-1} X^{2^{i-1}})$ is a polynomial with degree $2^k - 1$.




Since $G$ is a commitment, it can be checked in an inner product argument. The verifier circuit witnesses $G$ and brings $G, u_1, \cdots, u_k$ out as public inputs to the proof $\pi$. The next verifier instance checks $\pi$ using the inner product argument; this includes checking that $G = \text{Commit}(g(X, u_1, \cdots, u_k))$ evaluates at some random point to the expected value for the given challenges $u_1, \cdots, u_k$. Recall from the previous section that this check only requires $\log d$ work.

At the end of checking $\pi$ and $G$, the circuit is left with a new $G'$, along with the $u'_1, \cdots, u'_k$ challenges sampled for the check. To fully accept $\pi$ as valid, we should perform a linear-time computation of $G' = \langle \mathbf{G}, \mathbf{s}' \rangle$. Once again, we delay this

computation by witnessing $G'$ and bringing $G', u'_1, \cdots, u'_k$ out as public inputs to the proof $\pi'$.

This goes on from one proof instance to the next, until we are satisfied with the size of our batch of proofs. We finally perform a single linear-time computation, thus deciding the validity of the whole batch.

We recall from the section Cycles of curves that we can instantiate this protocol over a two-cycle, where a proof produced by one curve is efficiently verified in the circuit of the other curve. However, some of these verifier checks can actually be efficiently performed in the native circuit; these are "deferred" to the next native circuit (see diagram below) instead of being immediately passed over to the other curve.