SIEVE Circuit Intermediate Representation

See *Introduction to the SIEVE Intermediate Representation* for a list of contributors

Last Updated: 2023-07-17

This material is based upon work supported by DARPA under Contracts No. HR001120C0087, HR001120C0086, HR001120C0085 and Agreement No. HR00112020021. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

Contents

1 Introduction

The Circuit-IR has been a workhorse of the SIEVE Program, having been tested during the Phase II testing event. The SIEVE Circuit-IR encodes a circuit as a sequence of gates, each emitting one or more wires. This document specifies the format and semantics of the Circuit-IR, the public and private input streams used with it, and the Circuit Configuration Communication (CCC). For a quick example try jumping ahead to the example in section 3.8.

2 Headers

Please see the *Headers* section of *Introduction to the SIEVE Intermediate Representation*.

3 Circuit-IR

3.1 Header

As stated in Section 2, the Circuit-IR header starts with the version number and resource type.

```
version 2.0.0;
circuit;
```

Next, all plugins (Section 4), types, and conversion gates that are used in the file are declared up front in the header, before the `@begin` keyword. The order of these declarations must be sorted. Plugins must appear first, followed by types, and then conversion gates come last.

A type declaration specifies the types that are used in the rest of the file. Types can specify either a `field`, `ext_field`, or `ring`, as defined in Section 3.2. Type declarations also implicitly specify a type-index, assigned incrementally as each type is specified. The maximum number of types that can be defined is 256.

It is a failure of resource validity to declare the same type more than once. This property allows backends to compare types by comparing indices, which is less computationally expensive (in particular when dealing with large primes or structured types defined by plugins).

```
// index 0: Boolean
@type field 2;
// index 1: 2^61 - 1
@type field 2305843009213693951;
// index 2: GF(2^63) with polynomial modulus x^63 + x + 1
@type ext_field 0 63 9223372036854775811;
// index 3: Ring over 2^32
@type ring 32;
```

This example declares two field types for the Boolean and $2^{61} - 1$ prime fields, extension field type for $GF(2^{63})$, and a ring type for 2^{32}, respectively indexed by 0, 1, 2, and 3.

All conversion gates used in the file must be declared in the header. Each conversion gate declaration specifies which fields are converted between and how many of each field is input and output. Note that the length of input and output ranges must be greater than 0. Here are a few examples.

```
// Convert Booleans to Mersenne61 and back
@convert(@out: 1:1, @in: 0:61);
@convert(@out: 0:61, @in: 1:1);
// Convert Mersenne61 to 25519 and back
@convert(@out: 1:5, @in: 2:1);
@convert(@out: 2:1, @in: 1:5);
```

Conversion gates are fully specified in section 3.6.

3.2 Types

The Circuit-IR supports multiple types, as specified below.

- `@type field <p>`: This type specifies a field modulo prime `p`.

  Examples:

  ```
  // index 0: Boolean
  @type field 2;
  // index 1: 2^61 - 1
  @type field 2305843009213693951;
  ```

- `@type ext_field <index> <degree> <modulus>`: This type specifies an extension field $GF(p^n)$, where `index` denotes the type index of the (non-extension) field p, `degree` denotes the degree n of the extension field's polynomial modulus, and `modulus` is an integer constant which denotes the irreducible polynomial modulus of the extension field, as explained below. The value of `modulus` MUST be less than p^n, and the value of `index` MUST be associated with an already defined type.

  Constant representation. For extension fields, constants (including the `modulus` value above) are represented as base-10 integers, where the coefficients are the digits of the integer when interpreted in the base field p. The coefficient for the X^k term is given by the kth digit of the constant written in base p, where the 0th digit is the least significant.

  As an example, for $GF(2^{63})$ the constant 9223372036854775811 denotes the polynomial $X^{63} + X + 1$.

  Examples:

  ```
  // index 0: Boolean
  @type field 2;
  // index 1: GF(2^63) with polynomial modulus x^63 + x + 1
  @type ext_field 0 63 9223372036854775811;
  ```

- `@type ring <width>`: This type specifies a ring, $\mathbb{Z}_{2^n}$. Practically, these act like conventional unsigned base-2 integers.

  Examples:

  ```
  // index 0: unsigned 32-bit integers
  @type ring 32;
  ```

3.3 Public and Private Inputs

Inputs to the circuit are provided through separate resources (See Section 5) and accessed as streams. There are two streams per type, one for public inputs and one for private (prover only) inputs. Stream access uses the following syntax for public inputs:

```
$out_0 [... $out_n] <- @public([type_idx]);
```

And the following syntax for private inputs:

```
$out_0 [... $out_n] <- @private([type_idx]);
```

Items are read from the stream corresponding to the type index and assigned to the associated output wires. If the type index is not specified, it defaults to zero.

3.4 Memory Management

Backends that consume SIEVE IR are highly optimized. To minimize their overhead in managing memory, the IR exposes primitives to allocate and deallocate ranges of memory. There are strict restrictions on memory management: many operations require a range of input or output wires that are not only consecutive but also are part of the same allocation. This allows optimized backends to ensure that the input or output wires are stored in contiguous memory. Two wires may have consecutive wire-numbers, but be non-contiguous in memory, depending on the backend's internal memory management strategy.

Each type is given its own numbering space, with wire-numbers in the range of $0...2^{64} - 1$. Most directives will use a type-index parameter to select in which type, and in which numbering-space, they will act. For example, `0: $123` and `1: $123` may both be defined, with each wire residing in different numbering space due to their different types.

To allocate a range of wires explicitly, the `@new([ type_idx: ] $first ... $last);` directive may be used. This creates a new allocation containing exactly the wire numbers `$first ... $last`, but does not assign values to those wires. Reads from uninitialized wires is a failure of resource validity. The new allocation must not overlap any previous allocation. If the type index is not specified, it defaults to zero.

```
@new(1: $100 ... $200);
```

Directives that assign to wires will implicitly allocate the output wires if needed. For a directive that assigns to a range of output wires `type_idx: $first ... $last`, if all wires in the range are unallocated, it first creates an allocation as by the directive `@new(type_idx: $first ... $last)` and then assigns to the newly-allocated wires. If, instead, any of those wires were previously allocated, then they all must be part of a single allocation; if only some of the wires in the range were allocated, or different wires are part of different allocations, this is a failure of resource validity. This applies even for single wires; a single output wire such as `type_idx: $wire` is treated as a one-element range `type_idx: $wire ... $wire`. If a directive has multiple output wire ranges, each is handled independently, even if the ranges use consecutive numbers.

```
// Assume no wires are previously allocated.
// Implicitly allocates the range $100 ... $199
$100 ... $199 <- @call(foo1);

// Implicitly allocates the single wire $200
$200 <- @call(foo2);

@new($300 ... $399);
// All wires $300 ... $399 are already allocated, so this does
// not implicitly allocate.
$300 ... $399 <- @call(foo3);

@new($400 ... $449);
// Error: only some of the wires are allocated.
$400 ... $499 <- @call(foo4);

@new($500 ... $549);
@new($550 ... $599);
// Error: wires are not all part of a single allocation.
$500 ... $599 <- @call(foo5);

// This creates a separate allocation for each output range.
$600 ... $649, $650 ... $699 <- @call(foo6);
```

The `@delete` directive deallocates wires with the form `@delete([ type_idx: ] $first ... $last );`. All wires in the range `type_idx: $first ... $last` must be assigned and must not have been previously deleted. The range may span multiple allocations, but it must cover each allocation in full; deallocating only part of an allocation is a failure of resource validity. If the type index is not specified, it defaults to zero.

```
// Set up some allocations
@new(1: $100 ... $199)
@new(1: $200 ... $299)
// Implicit allocations are treated the same as @new
$300 ... $399 <- @call(foo)

// Error: @delete includes wires that were never allocated
@delete(1: $300 ... $499)

// Error: @delete covers only part of the $300 ... $399 allocation
@delete(1: $300 ... $310)

// Delete the entire $100 ... $199 allocation
@delete(1: $100 ... $199)

// Delete the $200 ... $299 and $300 ... $399 allocations
@delete(1: $200 ... $399)
```

Once a wire has been deleted, its wire number may not be reused and it may not be deleted again.

IMPORTANT Notice that the form of nearly all ranges in the IR is `first ... last` rather than `first ... length`. Ranges are inclusive on both ends.

3.5 Standard Gates

The form of most gates, unless otherwise specified, is:

```
$out <- gate_name([ type_idx:] $left_in, $right_in);
```

For a circuit to be well-formed, the type index must refer to a valid declared type. The type index is optional and defaults to type 0 when omitted. Other gates have variations on this form, and are described as necessary.

- `@add` field or ring addition
- `@mul` field or ring multiplication
- `@addc` field or ring addition by a constant

  ```
  $out <- @addc([type_idx:] $left_in, <right_constant>);
  ```

  Note: If `type_idx` specifies an extension field, then `right_constant` contains a decimal encoding of the extension field value, as is explained in Section 3.2.

- `@mulc` field or ring multiplication by a constant

  ```
  $out <- @mulc([type_idx:] $left_in, <right_constant>);
  ```

  Note: If `type_idx` specifies an extension field, then `right_constant` contains a decimal encoding of the extension field value, as is explained in Section 3.2.

- Copy the input wires to the output wires. The total number of input and output wires must be the same. Input and output ranges for copy gates must conform to the same memory contiguity constraints as input and output ranges for function calls as defined in Section 3.7.1.

  ```
  $out_0 [... $out_n] <- [type_idx_0:]
                         $in_first_0 [... $in_last_0]
                         [, $in_first_m [ .. $in_last_m]];
  ```

- Assign the input constant to the output wire

  ```
  $out <- [type_idx:] <constant>;
  ```

  Note: If `type_idx` specifies an extension field, then `constant` contains a decimal encoding of the extension field value, as is explained in Section 3.2.

- `@assert_zero` assert that a field element is zero

  ```
  @assert_zero([type_idx:] $wire);
  ```

For simplicity Boolean gates (in $GF(2)$) are replaced with mathematically equivalent arithmetic operations. This table summarizes alternative gates.

Boolean Gate	**Arithmetic Replacement**
`@and`	`@mul`
`@xor`	`@add`
`@not`	`@addc(x, <1>)`

For a circuit to be well formed, two rules must be obeyed when using and assigning wires. First, **topological ordering** requires that when a wire is used as the input to a gate, it must have been previously defined by an earlier gate in the scope. Second, **single static assignment (SSA)** requires that within a scope a particular wire is never redefined after its original assignment, even if it removed with the `@delete` directive.

3.6 Conversion Gates

Conversion gates enable conversion of wires from one type to another. Conversion gates are only supported between two `field` types or between an `ext_field` type and its associated `field` type.

3.6.1 Conversions Between Field Types

Conceptually a list of wires in field A is converted to a list of wires in field B. Within the circuit, a conversion gate has the form:

```
out_type_idx: $out_first [... $out_last] <- @convert(
    in_type_idx: $in_first [... $in_last] [, @modulus|@no_modulus]);
```

The optional `@modulus` / `@no_modulus` specifier specifies the semantics of the conversion; any other value denotes an error. If not specified, we default to `@no_modulus`. The conversion's fields and number of wires must match a conversion specification from the front matter. If it is not the case, there is a resource invalidity. Here is an example that uses conversion gates:

```
version 2.0.0;
circuit;
// field 0: Boolean
@type field 2;
// field 1: 2^61 - 1
@type field 2305843009213693951;
// field 2: 2^255 - 19
@type field 5789604461865809771178549250434395392663499233282028201972879200
// Declare used convert gates
@convert(@out: 1:1, @in: 0:61);
```

```
@convert(@out: 1:5, @in: 2:1);
@begin
    ...
    // convert Booleans to a single Mersenne61
    1: $0 <- @convert(0: $1 ... $61);
    // convert a single 25519 to 5 Mersenne61s
    1: $1 ... $5 <- @convert(2: $0);
    ...
@end
```

The input range `in_type_idx: $in_first ... $in_last` must be part of a single allocation. The output range `out_type_idx: $out_first ... $out_last` must either be part of a single allocation or be unallocated; if it is unallocated, the range will be implicitly allocated, as with `@new`.

Conversion Semantics. Here, we define in detail the specification of a `@convert` gate. Inputs and outputs are expressed in big endian representation. There are two possible semantics for conversions:

- `@no_modulus` (overflow semantics): To convert p wires $x_1 ... x_p$ in field A into q wires $y_1 ... y_q$ in field B, we first convert the p wires in field A into a natural number $N = \sum_{i=1}^{p} x_i \times A^{p-i}$. Then we try to represent N using q wires in field B $y_1 ... y_q$ as $N = \sum_{i=1}^{q} y_i \times B^{q-i}$. If N cannot be represented in this fashion, this constitutes a proof failure.

- `@modulus` (modulus semantics): To convert p wires $x_1 ... x_p$ in field A into q wires $y_1 ... y_q$ in field B, we first convert the p wires in field A into a natural number $N = \sum_{i=1}^{p} x_i \times A^{p-i} \mod B^q$. Then we represent N into q wires in field B $y_1 ... y_q$: $N = \sum_{i=1}^{q} y_i \times B^{q-i}$.

3.6.2 Conversions Between Extension Field and its Base Field

For extension fields one can only convert between an extension field $GF(p^n)$ and its base field p.

```
  out_type_idx: $out_first [... $out_last] <-
      @convert(in_type_idx: $in_first [... $in_last]);
```

Note that unlike for conversion gates for the field type, there is no optional specifier in this case.

The conversion semantics are straightforward: when converting to the base field, the extension field $GF(p^n)$ is decomposed into a sequence of n wires of field type p. When converting to the extension field, the n wires of field type p are used to represent the polynomial coefficients in $GF(p^n)$. For both conversion directions the base field wires use little endian ordering; that is, `$out_first` represents the coefficient for X^n and `$out_last` represents the coefficient for X^0.

```
version 2.0.0;
circuit;
// index 0: Boolean
@type field 2;
// index 1: GF(2^63)
@type ext_field 0 63 9223372036854775811;
// Declare used convert gates
```

```
@convert(@out: 0:63, @in: 1:1);
@convert(@out: 1:1, @in: 0:63);
@begin
  ...
  // convert Booleans to polynomial in GF(2^63)
  1: $0 <- @convert(0: $1 ... $63)
  // convert GF(2^63) to polynomial coefficients
  0: $1 ... $63 <- @convert(1: $0)
  ...
@end
```

3.6.3 Conversion Between Rings and Fields

Ring to ring, ring to field, and field to ring conversion use the same syntax as field to field conversions. Each ring wire is to be treated as a vector of n-many $GF(2)$ values, in most significant bit first order, where n is the bit-width specified in its type declaration. Conversion can then proceed as a field to field conversion.

3.7 Function Gates

Function gates define a sub-circuit which may be reused multiple times. The function's outputs and inputs are given as ranges mapped sequentially, and by type, into the function's scopes. In the function's signature, each range is defined by a length and a type index. When the function is invoked, each range is mapped into its scope incrementally from 0.

Function declaration and invocation have the following forms:

```
@function(function_name,
    [@out: out_type_idx_0: out_field_count_0
            [, out_type_idx_n: out_field_count_n],]
    [@in: in_type_idx_0: in_field_count_0
            [, in_type_idx_n: in_field_count_n],]
    )
  /* gate list */
@end

[$out_first_0 [ ... $out_last_0 ]
  [, $out_first_n [ ... $out_last_n ] ] <- ]
  @call(function_name [, $in_first_0 [ ... $in_last_0 ]
      [, $in_first_n [ ... $in_last_n ] ] ]);
```

The length of input and output ranges must be greater than 0 (for all `i`, $\texttt{out\_field\_count\_i} > 0$ and $\texttt{in\_field\_count\_i} > 0$).

Note that function invocations do not specify the type index for inputs and outputs since they can be inferred from the function signature.

3.7.1 Function Gate Example

```
@function(dot_prod_10, @out: 1:1, @in: 1:10, 1:10)
  // omitted
@end

@new(1: $0 ... $9);
@new(1: $10 ... $22);
// assign $0 ... $19

$25 <- @call(dot_prod_10, $0 ... $9, $10 ... $19);
```

The `@call` directive must have one range of input wires for each input range declared in the `@function` declaration. Each range of input wires must be part of a single allocation. Similarly, the `@call` must have one range of output wires for each output range declared in the `@function`. Each range of output wires must either be part of a single allocation or be unallocated; if it is unallocated, the range will be implicitly allocated, as with `new`.

3.7.2 Function Declaration Ordering and Recursion

Functions are declared at the top level of the circuit. Function names come into scope after their declaration. This prevents recursive functions and allows type checking while processing the file as a stream. For example, the following invocation is valid.

```
@function(a) /* ... */ @end

@function(b)
  @call(a);
@end

@call(b)
```

The next example is invalid since the function `a` has not been declared and is not yet in scope when `b` is defined.

```
@function(b)
  @call(a);
@end

@function(a) /* ... */ @end

@call(b)
```

3.8 Example

Here is a full example of a right-triangle using the Circuit-IR.

```
version 2.0.0;
circuit;
```

```
@type field 7;
@type field 127;
@convert(1:1, 0:1);

@begin
  // mod 7 hypotenuse
  $0 <- @public(0);
  // mod 7 legs
  $1 <- @private(0);
  $2 <- @private(0);

  // mod 7 is too small to square them
  1:$0 <- @convert(0:$0);
  1:$1 <- @convert(0:$1);
  1:$2 <- @convert(0:$2);

  // square them
  $3 <- @mul(1: $0, $0);
  $4 <- @mul(1: $1, $1);
  $5 <- @mul(1: $2, $2);
  $6 <- @add(1: $4, $5);

  // invert the hypotenuse
  $7 <- @mulc(1: $3, <126>);

  // assert equal
  $8 <- @add(1: $6, $7);
  @assert_zero(1: $8);
@end
```

3.9 Circuit Semantics and Validity

When working with the Circuit-IR there are three levels of semantics and validity to be considered. Each level builds upon the prior level.

1. **Syntactic Validity:** The IR resource is recognizable in the language defined by the IR's grammar (see appendix A and B).
2. **Resource Validity:** The IR resource obeys semantic rules which are falsifiable with just the single resource.
3. **Evaluation Validity:** Three IR resources (relation, public inputs, and private inputs) obey semantic rules which are only falsifiable in tandem.

While syntactic validity is important, it is easy to check using off the shelf parsing tools. The focus of this subsection is on resource validity and evaluation validity.

Each resource is checked individually for **Circuit Well-formedness** or **Stream Well-formedness** (See Section 5 for details). Circuit Well-formedness focuses on ensuring that wires are connected correctly – a "broken" wire would make the circuit poorly-formed – and that all declared types are unique.

Type Uniqueness No type is declared more than once.

Topological Ordering For a wire to be the input to a gate, it must have previously been assigned as an output wire within the same scope.

Static Single Assignment Each wire which is allocated must be assigned exactly once within its scope.

Allocation of Range Arguments When passing a range of wires (as either an input or an output), all wires in the range must belong to the same allocation, and the range's cardinality must match the called function or conversion gate's specification.

Deletion of Whole Allocations When passing a range of wires to a `@delete` directive, all wires within the range must have previously been assigned and all allocates within the range must be whole allocations. E.g. a `@delete` directive may not split an allocation into smaller portions.

To meet **Evaluation Validity**, all three resources are evaluated together, and the following conditions must be met.

Assertions Each input to an `@assert_zero` directive must carry the value 0.

Stream Length Requirement When the end of the circuit is reached each stream has exactly zero items remaining: it must not have run out of items before reaching the end, and there may not be any extra items.

4 Plugins

4.1 Motivation

In the previous section, we describe the Circuit-IR which contains the core IR functionalities. In this intermediate representation, we would like to add some (complex) features (e.g. RAM operations). Unfortunately, each update in the basic syntax forces frontends and backends to update their IR generator and parser. We would like to avoid this burden while increasing expressibility of the language. The goal of plugins is to allow IR extensions without changing the core IR.

4.2 Plugin Syntax

Plugins allow a circuit to refer to specific functionalities. Those functionalities are defined in a document. Only backends that have an implementation of a plugin can evaluate statements containing that plugin.

In the circuit's syntax, plugins are similar to functions except the function body is replaced by the plugin. The declaration of a plugin function starts with the signature of a function followed by the use of a `@plugin` directive with plugin parameters that includes the plugin name, the operation name, and its generic parameters. Invocation will remain the same as for functions. A function bound to a plugin must be declared before its invocation.

The names of plugins used must be specified in the header. This ensures that backends can easily check which plugins are used and reject the circuit if needed, prior to starting circuit evaluation.

Here is an example use of the `vectors_v1` plugin that provides a `mul` operation over ranges of wires.

```
...
@plugin vectors_v1;
...
@begin
  ...
  // declare the function signature with a plugin body
  @function(vec_mul_4, @out: 0:4, @in: 0:4, 0:4)
      @plugin(vectors_v1, mul);
  ...
  // call the vec_mul_4 plugin function
  $8 ... $11 <- @call(vec_mul_4, $0 ... $3, $4 ... $7);
  ...
@end
```

Plugin operations are defined elsewhere, so the `@plugin` binding takes a list of plugin defined arguments after the required plugin and operation names. Function signatures and their calls (`@call`) remain well-specified in the IR. Each instantiation of a plugin operation requires a separate function declaration and `@plugin` binding. Plugin binding arguments consist of a comma separated sequence of identifiers and numeric literals. In practice, this enables plugins to specify parameters like fields, lengths, and other functionality. Providing other functions' names as generic arguments can enable higher order operations like maps and folds.

```
/* ... */
@plugin vectors_v1;
@plugin iter_v0;
/* ... */
@begin
  /* ... */
  // Multiple instantiations of the same plugin (vector, mul)
  @function(vec_mul_4, @out: 0:4, @in: 0:4, 0:4)
      @plugin(vectors_v1, mul);
  @function(vec_mul_2, @out: 0:2, @in: 0:2, 0:2)
      @plugin(vectors_v1, mul);

  // Numeric parameter used as a circuit constant by the plugin
  @function(vec_double_4, @out: 0:4, @in: 0:4)
```

```
    @plugin(vectors_v1, mulc, 2);

  // Higher order map operation.
  @function(plus1_0, @out: 0:1, @in: 0:1) /* ... */ @end
  // An identifier parameter for the sub-function name,
  // A numeric parameter describing function arguments by their order
  // A numeric parameter for the iteration count
  @function(vec_plus1_4, @out: 0:4, @in: 0:4)
    @plugin(iter_v0, map, plus1_0, 0, 4);
  /* ... */
@end
```

Each plugin operation has a signature, which is defined as part of the standard for the plugin. If the backend sees a `@plugin` for a known plugin, and the function signature doesn't match the expected signature of the operation it is being bound to, then the circuit is invalid. Further, the plugin's name must have been declared in the circuit header. Either of these errors would make the circuit **poorly formed**.

Plugin operations may consume some public and private inputs. If a plugin operation consumes input from the standard public or private streams, its plugin binding must contain the count of the number of consumed public or private inputs per field. However, a plugin may be sophisticated enough to produce its own public or private inputs and consume them immediately, in which case input usage need not be declared. Here is an example use of an `assert_equal` plugin that checks that the five inputs are equal to the five next private inputs.

```
/* ... */
@plugin assert_equal;
/* ... */
@begin
  /* ... */
  // declare the function signature with a plugin body
  @function(equal_to_private, @in: 0:5)
   @plugin(assert_equal, private, 0, 5, @private: 0:5);
  /* ... */
  // call the equal_to_private plugin
  @call(equal_to_private, $4 ... $8);
  /* ... */
@end
```

4.3 Plugin Types

For some plugins, it is useful to declare additional types that are distinct from ordinary field types. Plugins can define new types by using the `@type @plugin(plugin_id, type_id, ...)` directive in the circuit header, which again takes a list of plugin-specified parameters. The first two parameters name the plugin, and a particular type within the plugin. Subsequent parameters may be identifiers or numbers, and their semantics are defined by the plugin. Types declared by a plugin can be manipulated via its plugin functions, or the plugin may overload built-in gates for its types. Here is an example demonstrating how the `ram_arith_v1` plugin uses a plugin type to implement a buffer.

```
version 2.0.0;
circuit;
@plugin ram_arith_v1;
// Wire type 0 is the field mod 127
@type field 127;
// Wire type 1 is the ram plugin type, with indexes and
// elements both drawn from field 0.
@type @plugin(ram\_arith\_v1, state, 0);

@begin
  // Declare RAM operations as abstract functions.
  // ram.init creates a buffer of fixed size and
  // initializes each element to the same input value.
  @function(ram_init, @out: 1:1, @in: 0:1)
    @plugin(ram, init, 10);

  // ram.read takes an RAM buffer and an address
  // and returns the value at that address.
  @function(ram_read, @out: 0:1, @in: 1:1, 0:1)
    @plugin(ram_arith_v1, read);

  // ram_write takes a RAM buffer, an address, and
  // a new value and writes the value to the address.
  @function(ram_write, @in: 1:1, 0:1, 0:1)
    @plugin(ram, write);

  @function(assert_eq, @in: 0:1, 0:1)
    $2 <- @addc($0, <126>); // p-1
    $3 <- @add($1, $2);
    @assert_zero($3);
  @end

  // Initialize all elements to 0 (type 0) in a newly
  // allocated RAM buffer (type 1)
  $0 <- 0: <0>;
  $0 <- @call(ram_init, $0);

  // Write something to the address
  $1 <- @private_in(); // address
  $2 <- @private_in(); // value

  //             1:$0 RAM Buffer
  //                    0:$1 address wire
  //                        0:$2 value wire
  @call(ram_write, $0, $1, $2);

  // Read from the same address and check correctness.
  $3 <- @call(ram_read, $0, $1);
  @call(assert_eq, $2, $3);
@end
```

5 Input Streams

Public and private inputs are provided as separate resources. We have one input file per type and per visibility level (`public_input` or `private_input`). These input files start with the same headers as described in section 2: the version, the resource type (`public_input` or `private_input`), and one type. Then, a sequence of numeric literals representing type elements is provided between `@begin` and `@end` tags. These sequences act as a stream, and certain directives in the circuit consume a value from one of these streams. If values in either stream are exhausted, this is a failure of evaluation validity. If values remain in a stream after processing, then this is also an evaluation invalidity. Here is an example for public and private inputs.

```
version 2.0.0;
public_input;
@type field 7;
@begin
  < 5 >;
@end
```

```
version 2.0.0;
public_input;
@type field 19;
@begin
  < 2 >;
  < 15 >;
@end
```

```
version 2.0.0;
private_input;
@type field 7;
@begin
  < 3 >;
  < 4 >;
@end
```

6 Circuit Configuration Communication (CCC)

6.1 Motivation

To ease interoperation each backend should provide a configuration file declaring available types and plugins that the frontend may use. This *Circuit Configuration Communication*, or *CCC* for short, is a static configuration file provided with the installation of a backend, and the frontend can configure from it while generating a statement. The CCC starts with supported plugins, then types, conversions, and it may end with plugin constraints. A compatible frontend will only use

- the types supported by the targeted backend according to the CCC file,
- the standard gates (Section 3.5),
- the conversion gates supported by the targeted backend according to the CCC file, and
- the plugins supported by the targeted backend according to the CCC file.

All SIEVE IR compatible backends must provide a CCC, and all SIEVE IR compatible frontends must ingest a backend's CCC to generate a statement for the backend. In the case that a backend cannot provide a feature (such as a type or a plugin) which the frontend would require, the frontend will be unable to generate a statement, and the pair of frontend and backend would be mutually incompatible.

6.2 CCC content and syntax

A CCC file starts with the standard IR header (Section 2), declaring its resource type as `configuration`. It list the plugins, families of types, and conversion gates supported by the backend.

Each available plugin is specified by name using the same `@plugin plugin_name;` syntax as the Circuit IR. A plugins presence in the CCC indicates its availability, meaning that the frontend is allowed to use it, but it does not require the frontend to use it. The frontend may use any subset of the backend's available plugins. Similarly, a backend may indicate that no plugins are supported, by leaving the plugins list empty.

Here is an example plugin list indicating the availability of the `mux_v0`, `arith_ram_v0`, and `arith_ram_v1` plugins. Presumably, a frontend could then choose if to use the `mux_v0` plugin, and if it needs RAM it could choose either version of the ram plugin. The frontend could also use both versions of the RAM plugin, although that would be redundant and confusing.

```
@plugin mux_v0;
@plugin ram_arith_v0;
@plugin ram_arith_v1;
```

6.3 Type Families

Types are grouped into "type families" in the CCC. Each type family encompasses a set of types which are somehow related. For example all fields which use Mersenne primes may be grouped into a "Mersenne family".

Each type family is specified first by what kind of type it is, then by predicates which constrain the set of possible types in the family. A type family may be referenced later in the CCC by its index in the order in which they appear, same as how Circuit IR types are referenced. There are four kinds of types, each having a different mathematical structure and accordingly different meanings for predicates.

- A `field` type is defined by a single prime characteristic. Implicitly, the characteristic must be prime, and the predicates of a `field` family enforce further restrictions on it. A `field` family has the form `@type field(`*`predicate`*` [, `*`predicates...`*`]);`. An empty list of predicates (e.g. empty parenthesis) indicates that any prime is supported (including, for example, 7).

- An `ext_field` type is defined by a base field, a polynomial order, and a numeric encoding of polynomial coefficients (see Section 3.2). There are three groups of predicates to an `ext_field` family. The first group is a list of previously declared base `field` families, referenced by index. The base type of a member of this `ext_field` family must be a member of one of the allowed base `field` families, but need not be a member of all (in fact, two base `field` families may be distinct of each other). The second group is predicates upon the order of the extension field. The order must be greater than one, and predicates may define additional restrictions. A third group defines predicates upon the numerically encoded polynomial coefficients. The polynomial must be irreducible, and predicates may define additional restrictions. An `ext_field` family has the form `@type ext_field (`*`base_idx`*` [, `*`base_idxs...`*`]) (`*`predicate`*` [, `*`predicates...`*`]) (`*`predicate`*` [, `*`predicates...`*`]);`. The family must define at least one *`base_idx`*, but both the order predicates and polynomial predicates may be empty.

- A `ring` type uses a ring over a 2^n modulus, providing the familiar bit representation for n-bit integers. The predicates of a `ring` family enforce restrictions on n. A `ring` family has the form `@type ring(`*`predicate`*` [, `*`predicates...`*`]);`. An empty list of predicates (e.g. empty parenthesis) indicates that any n is supported.

- A `plugin` type is defined using plugins. Plugin type families use plugin constraints (see below) instead of predicates. A `plugin` family has the form `@type @plugin `*`plugin_name`*`;`.

6.4 Predicates

Predicates define constraints upon numeric parameters of a type. The following predicates exist.

- `less_than( numeric )` constructs a predicate requiring the parameter to be less than the specified value.

- `greater_than( numeric )` constructs a predicate requiring the parameter to be greater than the specified value.

- `equals( numeric )` constructs a predicate requiring the parameter to equal the specified value.

- `is_mersenne` is a predicate requiring that the parameter be a Mersenne prime.
- `is_proth` is a predicate requiring that the parameter be a Proth prime.
- `is_power_of_2` is a predicate requiring that the parameter be a power of two.

When a type uses multiple predicates, each predicate is ANDed with the others. To produce an OR relation between predicates, simply create multiple type families.

6.5 Conversion Declarations

Conversion gate declarations specify which type families the backend supports conversions between. Conversion gates are declared with `@convert(@out: family_index_out, @in: family_index_in)` where the type families converted to and from are specified by their indices. It is currently not possible to configure constraints on the number of input or output wires in a conversion.

6.6 Plugin Constraints

Certain plugins may define additional constraints, such as constraints on vector widths or RAM address and value types. For each such plugin that the backend supports, the CCC should contain a *constraint block* delimited by `@plugin_constraint(ram) ... @end`, containing `@constraint(...)` lines specific to that plugin. The arguments to each `@constraint` line consist of comma-separated identifiers and integers, similar to the arguments of a function's `@plugin` binding declaration. Predicates may also be used within a `@constraint`, however their associativity and combinations with arbitrary identifiers or integers is plugin defined. Each plugin that uses constraints will specify the structure and semantics of `@constraint` lines allowed within the plugin's constraint block.

6.7 Example

```
version 2.0.0;
configuration;

// The following plugins may be used
@plugin mux_v0;
@plugin ram_arith_v0;
@plugin ram_arith_v1;

// 0: The field of exactly 2
@type field(equals(2));

// 1: Any prime greater than 1 million
@type field(greater_than(1000000));

// 2: Any mersenne prime which is greater than 100 and less than 1000.
// To avoid confusion, for a Mersenne Prime p such that p = 2**m − 1,
// 100 < p < 1000, rather than m.
@type field(is_mersenne, greater_than(100), less_than(1000));

// 3: An extension field over GF(2**4)
@type ext_field (0) (equals(4)) ();

// 4: a 16, 32, or 64−bit unsigned integer
@type ring(is_power_of_2, less_than(65), greater_than(15));

// 5, 6: RAM Types for either the v0 or v1 RAM plugin
@type @plugin ram_arith_v0;
@type @plugin ram_arith_v1;

// Conversion from large fields (type 1) to Booleans (type 0)
@convert(@out: 0, @in: 1);

// Conversion from large primes (type 1) to Mersennes (type 2)
@convert(@out: 2, @in: 1);

// Conversions from the extension field (type 3) to Booleans (type 0)
// Note, that the IR only defines conversions involving extension
// fields to or from their base fields.
@convert(@out: 0, @in: 3);

// Bidirectional conversions of fields (type 1) and rings (type 4)
@convert(@out: 4, @in: 1);
@convert(@out: 1, @in: 4);

// Note that different versions of the same plugin are
// unlikely to be compatible
// INVALID: @convert(@out: 5, @in: 6);

// Note that although constraints are shown for the vectors_v1
// plugin, they are shown for demonstration purpose and are not
// standardized. Frontends are unlikely to recognize these
// constraints.

@plugin_constraints(vectors_v1)
  // Element type must match either of the first two @field lines.
  // The plugin spec defines the meaning of repeated element_type
  // constraints; in this case, we assume repeated lines are
  // allowed and that the constraint is satisfied if at least one
  // line is satisfied.
  @constraint(element_type, 0);
  @constraint(element_type, 1);

  // Vector width must be a power of two between 2 and 16.  The
  // meaning of these constraints is defined by the plugin spec.
  @constraint(vector_width, is_power_of_2, 2, 16);
@end
```

Appendix A Textual Syntax

This appendix uses a right-recursive BNF grammar to define the exact syntax of the text format.

A.1 Special Tokens

The following special tokens are defined by regular expressions.

```
⟨number⟩ => 0 ;
⟨number⟩ => [1-9][0-9]* ;
⟨number⟩ => 0x[0-9a-fA-F]+ ;
⟨number⟩ => 0X[0-9a-fA-F]+ ;
⟨number⟩ => 0o[0-7]+ ;
⟨number⟩ => 0O[0-7]+ ;
⟨number⟩ => 0b[0-1]+ ;
⟨number⟩ => 0B[0-1]+ ;

// interpret ‘$’ as a literal dollar sign (U+0024)
⟨wire-number⟩ => $[1-9][0-9]* ;
⟨wire-number⟩ => $0x[0-9a-fA-F]+ ;
⟨wire-number⟩ => $0X[0-9a-fA-F]+ ;
⟨wire-number⟩ => $0o[0-7]+ ;
⟨wire-number⟩ => $0O[0-7]+ ;
⟨wire-number⟩ => $0b[0-1]+ ;
⟨wire-number⟩ => $0B[0-1]+ ;

// interpret ‘.’ as a literal point or dot (U+002E)
⟨identifier⟩ => [a-zA-Z_][a-zA-Z0-9_]*((.|::)[a-zA-Z_][a-zA-Z0-9_]*)* ;
```

Whitespace is defined with the following regular expression. Do note that in certain cases, non-empty whitespace may be necessary to break larger tokens.

```
// * is repetition. \* is a star character.
// \n, \r, \t are newline, carriage return, and tab.
⟨whitespace⟩ => ( |\n|\r|\t|//[^\n]*\n|/\*([^\*]|\*[^/])*\*/)* ;
```

A.2 Header

The following grammar is given for the IR Header.

```
⟨header⟩ => ⟨version-spec⟩ ⟨resource-id⟩ ;

⟨version-spec⟩ => ‘version’ ⟨number⟩ ‘.’ ⟨number⟩ ‘.’ ⟨number⟩ ‘;’ ;

⟨resource-id⟩ => ‘circuit’ ‘;’ ;
⟨resource-id⟩ => ‘translation’ ‘;’ ;
⟨resource-id⟩ => ‘public_input’ ‘;’ ;
⟨resource-id⟩ => ‘private_input’ ‘;’ ;
⟨resource-id⟩ => ‘configuration’ ‘;’ ;
```

A.3 Circuit-IR

The following grammar is given for the Circuit-IR (resource id `circuit`).

```
⟨circuit⟩ => ⟨header⟩ ⟨circuit-header⟩ ⟨circuit-body⟩ EOF ;

⟨circuit-header⟩ => ⟨type-list⟩ ;
⟨circuit-header⟩ => ⟨type-list⟩ ⟨conversion-list⟩ ;

⟨type-list⟩ => ⟨type⟩ ⟨type-list⟩ ;
⟨type-list⟩ => ⟨type⟩ ;
⟨type⟩ => ‘@type’ ‘field’ ⟨number⟩ ‘;’ ;
⟨type⟩ => ‘@type’ ‘ext_field’ ⟨number⟩ ⟨number⟩ ⟨number⟩ ‘;’ ;
⟨type⟩ => ‘@type’ ‘ring’ ⟨number⟩ ‘;’ ;

// note that in @out: t:n (and likewise @in: t:n), n > 0
⟨conversion-list⟩ => ⟨conversion⟩ ⟨conversion-list⟩ ;
⟨conversion-list⟩ => ⟨conversion⟩ ;
⟨conversion⟩ => ‘@convert’ ‘(’
    ‘@out’ ‘:’ ⟨number⟩ ‘:’ ⟨number⟩ ‘,’
    ‘@in’ ‘:’ ⟨number⟩ ‘:’ ⟨number⟩ ‘,’
    ‘)’ ‘;’ ;

⟨circuit-body⟩ => ‘@begin’ ⟨top-scope⟩ ‘@end’ ;

⟨top-scope⟩ => ⟨top-scope-item⟩ ⟨top-scope⟩ ;
⟨top-scope⟩ => ⟨top-scope-item⟩ ;

⟨top-scope-item⟩ => ⟨function-declaration⟩ ;
⟨top-scope-item⟩ => ⟨directive⟩ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@out’ ‘:’ ⟨parameter-list⟩ ‘,’
    ‘@in’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨function-scope⟩
    ‘@end’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@in’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨function-scope⟩
    ‘@end’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@out’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨function-scope⟩
    ‘@end’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘)’
    ⟨function-scope⟩
    ‘@end’ ;

// note that in parameter t:n, n > 0
⟨parameter-list⟩ => ⟨parameter⟩ ‘,’ ⟨parameter-list⟩ ;
⟨parameter-list⟩ => ⟨parameter⟩ ;
⟨parameter⟩ => ⟨number⟩ ‘:’ ⟨number⟩ ;

⟨function-scope⟩ => ⟨function-scope-item⟩ ⟨function-scope⟩ ;
⟨function-scope⟩ => ⟨function-scope-item⟩ ;
⟨function-scope-item⟩ => ⟨directive⟩ ;

⟨directive⟩ => ⟨new-directive⟩ ;
⟨directive⟩ => ⟨delete-directive⟩
⟨directive⟩ => ⟨binary-gate⟩ ;
⟨directive⟩ => ⟨binary-const-gate⟩ ;
⟨directive⟩ => ⟨copy-directive⟩ ;
⟨directive⟩ => ⟨assign-directive⟩ ;
⟨directive⟩ => ⟨input-gate⟩ ;
⟨directive⟩ => ⟨assert-zero-directive⟩ ;
⟨directive⟩ => ⟨convert-gate⟩ ;
⟨directive⟩ => ⟨call-directive⟩ ;

⟨new-directive⟩ => ‘@new’ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-number⟩ ‘...’ ⟨wire-number⟩ ‘)’ ‘;’ ;
⟨new-directive⟩ => ‘@new’ ‘(’
    ⟨wire-number⟩ ‘...’ ⟨wire-number⟩ ‘)’ ‘;’ ;

⟨delete-directive⟩ => ‘@delete’ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-number⟩ ‘...’ ⟨wire-number⟩ ‘)’ ‘;’ ;
⟨delete-directive⟩ => ‘@delete’ ‘(’
    ⟨wire-number⟩ ‘...’ ⟨wire-number⟩ ‘)’ ‘;’ ;

⟨binary-gate⟩ => ⟨wire-number⟩ ‘<-’ ⟨binary-gate-operation⟩ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-number⟩ ‘,’ ⟨wire-number⟩ ‘)’ ‘;’ ;
⟨binary-gate⟩ => ⟨wire-number⟩ ‘<-’ ⟨binary-gate-operation⟩ ‘(’
    ⟨wire-number⟩ ‘,’ ⟨wire-number⟩ ‘)’ ‘;’ ;

⟨binary-gate-operation⟩ => ‘@add’ ;
⟨binary-gate-operation⟩ => ‘@mul’ ;

⟨binary-const-gate⟩ => ⟨wire-number⟩ ‘<-’ ⟨binary-const-gate-operation⟩ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-number⟩ ‘,’ ‘<’ ⟨number⟩ ‘>’ ‘)’ ‘;’ ;
⟨binary-const-gate⟩ => ⟨wire-number⟩ ‘<-’ ⟨binary-const-gate-operation⟩ ‘(’
    ⟨wire-number⟩ ‘,’ ‘<’ ⟨number⟩ ‘>’  ‘)’ ‘;’ ;

⟨binary-const-gate-operation⟩ => ‘@addc’ ;
⟨binary-const-gate-operation⟩ => ‘@mulc’ ;

⟨copy-directive⟩ => ⟨wire-range⟩ ‘<-’ ⟨number⟩ ‘:’ ⟨wire-range-list⟩ ‘;’ ;
⟨copy-directive⟩ => ⟨wire-range⟩ ‘<-’ ⟨wire-range-list⟩ ‘;’ ;
⟨assign-directive⟩ => ⟨wire-number⟩ ‘<-’ ⟨number⟩ ‘:’ ‘<’ ⟨number⟩ ‘>’ ‘;’ ;
⟨assign-directive⟩ => ⟨wire-number⟩ ‘<-’ ‘<’ ⟨number⟩ ‘>’ ‘;’ ;

⟨input-gate⟩ => ⟨wire-range⟩ ‘<-’ ⟨input-gate-operation⟩ ‘(’ ⟨number⟩ ‘)’ ‘;’ ;
⟨input-gate⟩ => ⟨wire-range⟩ ‘<-’ ⟨input-gate-operation⟩ ‘(’ ‘)’ ‘;’ ;

⟨input-gate-operation⟩ => ‘@public’ ;
⟨input-gate-operation⟩ => ‘@private’ ;

⟨assert-zero-directive⟩ => ‘@assert_zero’ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-number⟩ ‘)’ ‘;’ ;
⟨assert-zero-directive⟩ => ‘@assert_zero’ ‘(’ ⟨wire-number⟩ ‘)’ ‘;’ ;

⟨convert-gate⟩ => ⟨number⟩ ‘:’ ⟨wire-range⟩ ‘<-’ ‘@convert’ ‘(’
    ⟨number⟩ ‘:’ ⟨wire-range⟩ ‘)’ ‘;’ ;

⟨wire-range⟩ => ⟨wire-number⟩ ;
⟨wire-range⟩ => ⟨wire-number⟩ ‘...’ ⟨wire-number⟩ ;

⟨call-directive⟩ => ⟨wire-range-list⟩ ‘<-’ ‘@call’ ‘(’
    ⟨identifier⟩ ⟨wire-range-list⟩ ‘)’ ‘;’ ;
⟨call-directive⟩ => ‘@call’ ‘(’ ⟨identifier⟩ ⟨wire-range-list⟩ ‘)’ ‘;’ ;
⟨call-directive⟩ => ⟨wire-range-list⟩ ‘<-’ ‘@call’ ‘(’ ⟨identifier⟩ ‘)’ ‘;’ ;
```

```
⟨call-directive⟩ => ‘@call’ ‘(’ ⟨identifier⟩ ‘)’ ‘;’ ;

⟨wire-range-list⟩ => ⟨wire-range⟩ ‘,’ ⟨wire-range-list⟩ ;
⟨wire-range-list⟩ => ⟨wire-range⟩ ;
```

A.4 Input Streams

The following grammar is given for input streams (resources `private_input` and `public_input`).

```
⟨stream⟩ => ⟨header⟩ ⟨stream-header⟩ ⟨stream-body⟩ EOF ;

⟨stream-header⟩ => ‘@type’ ‘field’ ⟨number⟩ ‘;’ ;

⟨stream-body⟩ => ‘@begin’ ⟨stream-list⟩ ‘@end’ ;
⟨stream-body⟩ => ‘@begin’ ‘@end’ ;

⟨stream-list⟩ => ⟨stream-item⟩ ⟨stream-list⟩ ;
⟨stream-list⟩ => ⟨stream-item⟩ ;

⟨stream-item⟩ => ‘<’ ⟨number⟩ ‘>’ ‘;’ ;
```

A.5 Circuit Plugins

The following additions to the Circuit-IR grammar will enable plugins.

```
⟨circuit-header⟩ => ⟨plugin-list⟩ ⟨type-list⟩ ;
⟨circuit-header⟩ => ⟨plugin-list⟩ ⟨type-list⟩ ⟨conversion-list⟩ ;

⟨plugin-list⟩ => ⟨plugin⟩ ⟨plugin-list⟩ ;
⟨plugin-list⟩ => ⟨plugin⟩ ;
⟨plugin⟩ => ‘@plugin’ ⟨identifier⟩ ‘;’ ;

⟨type⟩ => ‘@type’ ⟨plugin-binding⟩ ‘;’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@out’ ‘:’ ⟨parameter-list⟩ ‘,’
    ‘@in’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨plugin-binding⟩ ‘;’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@in’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨plugin-binding⟩ ‘;’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘,’
    ‘@out’ ‘:’ ⟨parameter-list⟩ ‘)’
    ⟨plugin-binding⟩ ‘;’ ;

⟨function-declaration⟩ => ‘@function’ ‘(’ ⟨identifier⟩ ‘)’
    ⟨plugin-binding⟩ ‘;’ ;

⟨plugin-binding⟩ => ‘@plugin’ ‘(’ ⟨identifier⟩ ‘,’ ⟨identifier⟩
    ‘,’ ⟨plugin-arguments⟩ ⟨stream-count⟩ ‘)’ ;
⟨plugin-binding⟩ => ‘@plugin’ ‘(’ ⟨identifier⟩ ‘,’ ⟨identifier⟩
    ⟨stream-count⟩ ‘)’ ;

⟨plugin-arguments⟩ => ⟨plugin-argument⟩ ⟨plugin-arguments⟩ ;
⟨plugin-arguments⟩ => ⟨plugin-argument⟩ ;

⟨plugin-argument⟩ => ⟨number⟩ ;
⟨plugin-argument⟩ => ⟨identifier⟩ ;

⟨stream-count⟩ => ‘,’ ‘@private’ ‘:’ ⟨stream-count-items⟩ ;
⟨stream-count⟩ => ‘,’ ‘@public’ ‘:’ ⟨stream-count-items⟩ ;
⟨stream-count⟩ => ‘,’ ‘@private’ ‘:’ ⟨stream-count-items⟩ ‘,’
    ‘@public’ ‘:’ ⟨stream-count-items⟩ ;
⟨stream-count⟩ => NULL ;

⟨stream-count-items⟩ => ⟨stream-count-item⟩ ‘,’ ⟨stream-count-items⟩ ;
⟨stream-count-items⟩ => ⟨stream-count-item⟩ ;
⟨stream-count-item⟩ => ⟨number⟩ ‘:’ ⟨number⟩ ;
```

Appendix B Binary Syntax

B.1 FlatBuffer Schema

The binary serialization of Circuit-IR will be described here using the open-source FlatBuffers cross-platform serialization library, originally developed by Google. FlatBuffers is a metaformat that specifies the superficial aspects of the syntax, such as representations of literals, structured data and arrays. It moreover supports formal schemas that concretely define what elements (e.g., structures and arrays) can appear in the specific format. FlatBuffers was chosen for the following reasons:

- It offers an existing compact encoding of the format with efficient (de)serialization.
- It is supported by a wide-range of community-based tools and libraries for the most common languages (and is also easy to parse from scratch).

We will use the below FlatBuffers schema to represent `circuit`, `public_input`, and `private_input` resources. This schema is isomorphic to the Circuit-IR representation presented in Section 3.

```
// This is a FlatBuffers schema.
// See https://google.github.io/flatbuffers/
namespace sieve_ir;

// REGEX used:
// - VERSION_REGEX = "^\d+.\d+.\d+$"
// - STRING_REGEX = "^[a-zA-Z_][\w]*((\.|::)[a-zA-Z_][\w]*)*$"
// - NUMBER_REGEX = "^((\d+)|(0x[0-9a-fA-F]+))$"

// ==== Message types that can be exchanged. ====
union Message {
  Relation,
  PublicInputs,
  PrivateInputs,
}

// The 'version' field must match VERSION_REGEX
// Each string in the 'plugins' list must match STRING_REGEX
table Relation {
  version       :string;
  plugins       :[string];
  types         :[Type];
  conversions   :[Conversion];
  directives    :[Directive];
}

// The 'version' field must match VERSION_REGEX
table PublicInputs {
  version   :string;
  type      :Type;
  inputs    :[Value];
}

// The 'version' field must match VERSION_REGEX
table PrivateInputs {
  version   :string;
  type      :Type;
  inputs    :[Value];
}

// ==== Helper types ====
// Type element is encoded in a vector of bytes in little-endian
// order. There is no minimum or maximum length; trailing zeros
// may be omitted.
table Value {
  value  :[ubyte];
}

struct Count {
  type_id   :ubyte;
  count     :uint64;
}

// ==== Directive ====
union DirectiveSet {
  Gate,
  Function,
}

table Directive {
  directive   :DirectiveSet;
}

// ==== Conversion ====
// The 'count' field of these 'Count's must be > 0
struct Conversion {
  output_count   :Count;
  input_count    :Count;
}

// ==== Type ====
union TypeU {
  Field,
  ExtField,
  Ring,
  PluginType
}

table Type {
  element   :TypeU;
}

table Field {
  modulo   :Value;
}

table ExtField {
  index    :ubyte;
  degree   :uint64;
  modulus  :uint64;
}
```

```
table Ring {
  nbits    :uint64;
}

// 'name' and 'operation' must match STRING_REGEX
// Strings of the 'params' list must match either
// STRING_REGEX or NUMBER_REGEX
table PluginType {
  name        :string;
  operation   :string;
  params      :[string];
}

// ==== Gate types ====
table GateConstant {
  type_id    :ubyte;
  out_id     :uint64;
  // 'constant' is encoded in a vector of bytes in little-endian
  // order. There is no minimum or maximum length; trailing zeros
  // may be omitted.
  constant   :[ubyte];
}

table GateAssertZero {
  type_id   :ubyte;
  in_id     :uint64;
}

table GateCopy {
  type_id   :ubyte;
  out_id    :WireRange;
  in_id     :[WireRange];
}

table GateAdd {
  type_id   :ubyte;
  out_id    :uint64;
  left_id   :uint64;
  right_id  :uint64;
}

table GateMul {
  type_id   :ubyte;
  out_id    :uint64;
  left_id   :uint64;
  right_id  :uint64;
}

table GateAddConstant {
  type_id    :ubyte;
  out_id     :uint64;
  in_id      :uint64;
  // 'constant' is encoded in a vector of bytes in little-endian
  // order. There is no minimum or maximum length; trailing zeros
  // may be omitted.
  constant   :[ubyte];
}

table GateMulConstant {
  type_id    :ubyte;
  out_id     :uint64;
  in_id      :uint64;
  // 'constant' is encoded in a vector of bytes in little-endian
  // order. There is no minimum or maximum length; trailing zeros
  // may be omitted.
  constant   :[ubyte];
}

table GatePublic {
  type_id   :ubyte;
  out_id    :WireRange;
}

table GatePrivate {
  type_id   :ubyte;
  out_id    :WireRange;
}

// To allocate in a contiguous space all wires between
// first_id and last_id inclusive.
table GateNew {
  type_id    :ubyte;
  first_id   :uint64;
  last_id    :uint64;
}

table GateDelete {
  type_id    :ubyte;
  first_id   :uint64;
  last_id    :uint64;
}

table GateConvert {
  out_type_id    :ubyte;
  out_first_id   :uint64;
  out_last_id    :uint64;
  in_type_id     :ubyte;
  in_first_id    :uint64;
  in_last_id     :uint64;
  modulus        :bool;
}

// ==== Function declaration ====
union FunctionBody {
  Gates,
  PluginBody,
}

table Gates {
  gates   :[Gate];
}

// 'name' and 'operation' must match STRING_REGEX
// Strings of the 'params' list must match either
// STRING_REGEX or NUMBER_REGEX
table PluginBody {
  name            :string;
  operation       :string;
  params          :[string];
  public_count    :[Count]; // Each type_id must be unique
  private_count   :[Count]; // Each type_id must be unique
}

// Declare a Function gate as a custom computation or from a plugin
// The 'name' must match STRING_REGEX
// The 'output_count' and 'input_count' must be > 0
table Function {
  name           :string;
  output_count   :[Count];
  input_count    :[Count];
  body           :FunctionBody;
}

struct WireRange {
  first_id   :uint64;
  last_id    :uint64;
}

// Invokes a previously defined Function gate
// The 'name' must match STRING_REGEX
table GateCall {
  name      :string;
  out_ids  :[WireRange];
  in_ids   :[WireRange];
}

union GateSet {
  GateConstant,
  GateAssertZero,
  GateCopy,
  GateAdd,
  GateMul,
  GateAddConstant,
  GateMulConstant,
  GatePublic,
  GatePrivate,
  GateNew,
  GateDelete,
  GateConvert,
  GateCall,
}

table Gate {
  gate   :GateSet;
}

// ==== Flatbuffers details ====
// All message types are encapsulated in the FlatBuffers root table.
table Root {
  message   :Message;
}
root_type Root;

// When storing messages to files, this extension and identifier
// should be used.
file_extension "sieve";
file_identifier "siev"; // a.k.a. magic bytes.

// Message framing:
```

```
//
// All messages must be prefixed by its size in bytes,
// as a 4-bytes little-endian unsigned integer.
```

As a structured format, the FlatBuffers schema provides a concrete, readable and typed syntax, ensuring syntactic validity. However, it does not provide resource or evaluation validity as it is not a language. Refer to Section 3.9 to a description of syntactic, resource and evaluation validity.

B.2 Multi Gigabyte Flatbuffer Limitations

A limitation of the Flatbuffer technology is its 32-bit internal pointer representation, which prevents it from storing buffers larger than approximately 2GB. The IR specifies the following workaround for this limitation.

- The `Root` message may be repeated within a file or stream as many times as is necessary: each message holding a portion of the IR resource.
- Each message must be prefixed by its length in bytes, as a 4-byte unsigned little-endian number. (See FinishSizePrefix).
- Each message's `version` attribute must be the same as the first message's `version`. All other attributes must be empty except for the resource's body.

Unfortunately, there is no way for a Flatbuffer to hold a single function which is larger than 2GB.