

# R1CS Programming

## ZK0x04 Workshop Notes

Daniel Lubarov      Brendan Farmer

October 22, 2019

### Contents

<b>1</b>	<b>Multiplicative inverse</b>	<b>2</b>
<b>2</b>	<b>Zero testing</b>	<b>2</b>
<b>3</b>	<b>Binary</b>	<b>2</b>
<b>4</b>	<b>Selection</b>	<b>3</b>
<b>5</b>	<b>Random access</b>	<b>3</b>
<b>6</b>	<b>Embedded curve operations</b>	<b>4</b>
6.1	Addition . . . . .	4
6.2	Multiplication . . . . .	5
<b>7</b>	<b>2x2 switch</b>	<b>5</b>
<b>8</b>	<b>Permutations</b>	<b>5</b>
<b>9</b>	<b>Sorting</b>	<b>6</b>
<b>10</b>	<b>Comparisons</b>	<b>6</b>

## 1 Multiplicative inverse

Deterministically computing  $1/x$  in an R1CS circuit would be expensive. Instead, we can have the prover compute  $1/x$  outside of the circuit and supply the result as a witness element, which we will call  $x_{\text{inv}}$ . To verify the result, we enforce

$$(x)(x_{\text{inv}}) = (1) \quad (1)$$

## 2 Zero testing

To assert  $x = 0$ , we simply enforce

$$(x)(1) = (0) \quad (2)$$

Asserting  $x \neq 0$  is similarly easy: we compute  $1/x$  (non-deterministically, as in [Section 1](#)). The result can be ignored; the mere fact that an inverse exists implies  $x \neq 0$ .

On the other hand, if we want to *evaluate*

$$y := \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise,} \end{cases} \quad (3)$$

we can do so by introducing another variable,  $m$ , and enforcing

$$(x)(m) = (y), \quad (4)$$

$$(1 - y)(x) = (0). \quad (5)$$

Outside of the circuit, the prover generates  $y$  as in [Equation 3](#), and generates  $m$  as

$$m := \begin{cases} 1 & \text{if } x = 0, \\ y/x & \text{otherwise.} \end{cases} \quad (6)$$

This method is from [\[1\]](#).

We can also use this technique to test for equality, since  $x = y$  if and only if  $x - y = 0$ .

## 3 Binary

To assert  $b \in \{0, 1\}$ , we enforce

$$(b)(b - 1) = (0). \quad (7)$$

To “split” a field element  $x$  into its binary encoding,  $(b_1, \dots, b_n)$ , we have the prover generate the binary encoding out-of-band. We then verify it by applying [Equation 7](#) to each  $b_i$ , and enforcing

$$(x)(1) = \left( \sum_{i=0}^{n-1} 2^i b_i \right), \quad (8)$$

assuming a little-ending ordering of the bits.

Note that [Equation 8](#) permits two encodings of certain field elements. In  $\mathbb{F}_{13}$  for example, the element 1 can be represented as either 0001 or 1110. If a canonical encoding is required, we can prevent “overflowing” encodings by asserting that  $(b_1, \dots, b_n) < |F|$ . Such binary comparisons are covered in [Section 10](#).

To “join” a sequence of bits into the field element they encode, we simply take a weighted sum of the bits:

$$x := \sum_{i=0}^{n-1} 2^i b_i. \quad (9)$$

This does not require any constraints, unless we don’t know whether  $(b_1, \dots, b_n)$  is the canonical encoding of some field element, in which case we may want a comparison to assert that overflow does not occur.

## 4 Selection

Suppose we have a boolean value  $s$ , and we wish to compute

$$z := \begin{cases} x & \text{if } s = 0, \\ y & \text{if } s = 1. \end{cases} \quad (10)$$

We can compute this as

$$z := x + s(y - x). \quad (11)$$

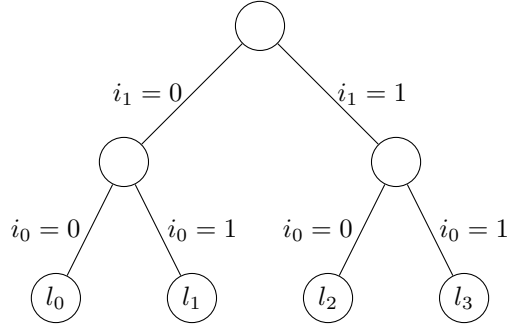
This requires two constraints: one “is boolean” assertion ([Equation 7](#)), assuming  $s$  was not already known to be boolean, and another for the multiplication.

## 5 Random access

Suppose we have a sequence of field elements,  $S = (x_0, \dots, x_n)$ , and we wish to access its  $i$ th element. For simplicity, we will assume  $n = 2^k$ , although our techniques can be generalized to arbitrary sequence lengths.

We start by splitting  $i$  into its binary encoding,  $(i_0, \dots, i_k)$ . One may think of  $(i_0, \dots, i_k)$  as a path through a binary tree, where  $i_j$  indicates the direction of descent at height  $j$ . For each leaf  $j$ , we construct a polynomial  $l_j : (i_0, \dots, i_k) \rightarrow \{0, 1\}$  which evaluates whether  $j$  is the selected index.

If  $k = 2$  for example, we would imagine the following binary tree:



and construct the leaf polynomials as

$$\begin{aligned}
l_0 &:= (1 - i_1)(1 - i_0) = 1 - i_0 - i_1 + i_0 i_1, \\
l_1 &:= (1 - i_1)(i_0) = i_0 - i_0 i_1, \\
l_2 &:= (i_1)(1 - i_0) = i_1 - i_0 i_1, \\
l_3 &:= (i_1)(i_0) = i_0 i_1.
\end{aligned}$$

Once we have these leaf polynomials, the result of the lookup is

$$\sum_{j=0}^{n-1} x_j l_j \tag{12}$$

TODO: Describe trick to get an extra layer for cheap.

TODO: Discuss lookups for  $r$ -tuples of field elements.

## 6 Embedded curve operations

Embedded curves have several uses in SNARKs. A few examples are Schnorr signatures, Pedersen hashes, and recursive SNARK verifiers. Here we will focus on twisted Edwards curves such as Jubjub.

### 6.1 Addition

Recall the addition law for twisted Edwards curves,

$$(x_1, y_1) + (x_2, y_2) = \left( \frac{x_1 y_1 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right). \tag{13}$$

Applying the law directly takes 7 constraints: 4 for the products in the numerators, one for the product in the denominator, and one<sup>1</sup> for each of the two quotients.

<sup>1</sup>In general, computing a quotient  $q := x/y$  takes two constraints:  $(y)(y_{\text{inv}}) = (1)$  and  $(x)(y_{\text{inv}}) = (q)$ . In this case, however, we can multiply both sides by the denominator since we know it will never be zero. This yields a single constraint:  $(q)(y) = (x)$ .

However, the operation becomes much cheaper when one of the points is constant. Without loss of generality, suppose  $(x_1, y_1)$  is constant while  $(x_2, y_2)$  is variable. Then the numerators become “free” linear combinations, while the denominators require a single multiplication. The quotients add 2 constraints as before, resulting in 3 constraints total.

Finally, when doubling a point, the addition law simplifies to

$$[2](x, y) = \left( \frac{2xy}{ax^2 + y^2}, \frac{y^2 - ax^2}{2 - ax^2 - y^2} \right) \quad (14)$$

which requires 5 constraints.

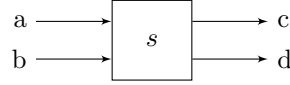
## 6.2 Multiplication

Consider the problem of computing  $[s]p$ , where  $p$  is a constant point and  $s$  is a variable scalar.

TODO: Discuss multiplication by doubling, along with its variants like windowed multiplication.

## 7 2x2 switch

Suppose we wish to implement a switch with the following structure:



In particular, if  $s = 0$  then the outputs should be identical to the inputs:  $(c, d) = (a, b)$ . If  $s = 1$  then the inputs should be swapped:  $(c, d) = (b, a)$ .

This requires two constraints: one “is boolean” assertion ([Equation 7](#)), and another for selecting the value of  $c$  ([Equation 11](#)). Once we have  $c$ , we can compute  $d$  as

$$d := a + b - c \quad (15)$$

which does not require any additional constraints.

## 8 Permutations

Say we want to verify that two sequences,  $(x_1, \dots, x_n)$  and  $(y_1, \dots, y_n)$ , are permutations of one another. This can be done efficiently using routing networks, which used a fixed (for a fixed  $n$ ) network of 2x2 switches.

AS-Waksman networks [\[2\]](#) are a particularly useful construction, since they support arbitrary permutation sizes. They use about  $n \log_2(n) - n$  switches, which is close to the theoretical lower bound of  $\log_2(n!)$ .

## 9 Sorting

Like permutation networks, sorting networks use a fixed network of gates. In particular, a sorting network is comprised of several 2x2 comparator gates, each of which takes two inputs and sorts them. It is theoretically possible to construct a sorting network for  $n$  elements using  $\mathcal{O}(n \log n)$  gates [3], but practical constructions use  $\mathcal{O}(n \log^2 n)$  gates. Since each comparison adds  $\mathcal{O}(\log |F|)$  constraints, this approach is fairly expensive.

A better solution is to leverage non-determinism: instead of creating an RICS circuit to sort a sequence, we have the prover supply the ordered sequence. Using a permutation network (Section 8), we can efficiently verify that the two sequences are permutations of one another. Then for each contiguous pair of elements in the ordered sequence,  $x_i$  and  $x_{i+1}$ , we assert  $x_i \leq x_{i+1}$ .

## 10 Comparisons

Suppose we want to evaluate  $x < y$ . For now, assume that both values fit in  $n$  bits, where  $2^{n+1} < |F|$ . We compute

$$z = 2^n + x - y. \quad (16)$$

Notice that  $2^n$  has a single 1 bit with index  $n$ , which will be cleared if and only if  $x < y$ . Thus we split  $z$  into  $n + 1$  bits, and the negation of the  $n$ th bit is our result.

If  $x$  and  $y$  are unbounded, then the aforementioned technique does not apply. A neat solution was described by Ahmed Kosba. First, we split  $x$  and  $y$  into their binary forms,  $(x_0, \dots, x_n)$  and  $(y_0, \dots, y_n)$ . Next we divide each binary sequence into  $c$ -bit chunks, and join each chunk of bits into a field element (Equation 9).

The prover non-deterministically identifies the most significant chunk index at which  $x$  and  $y$  differ, if any. We add constraints to verify that the identified chunks do indeed differ, and that any more significant chunks are equal. Finally, we compare the two identified chunks using Equation 16.

If we choose  $c, l \in \mathcal{O}(\sqrt{n})$ , this approach requires  $\mathcal{O}(\sqrt{n})$  constraints in addition to the cost of splitting the inputs (if they were not already in binary form). A couple optimizations are possible in particular circumstances:

1. To assert (not evaluate)  $x < y$ , we can split  $x$  non-canonically and split  $y$  canonically. The prover is forced to use  $x$ 's canonical encoding anyway, otherwise  $x_{\text{bin}} \geq |F| > y_{\text{bin}}$ , making the assertion unsatisfiable.
2. To assert  $x < c$  for some constant  $c \ll |F|$ , we can split  $x$  into just  $\lceil \log_2 c \rceil$  bits.

## References

- [1] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” in *2013 IEEE Symposium on Security and Privacy*, pp. 238–252, IEEE, 2013.
- [2] B. Beauquier and E. Darrot, “On arbitrary size waksman networks and their vulnerability,” *Parallel Processing Letters*, vol. 12, no. 03n04, pp. 287–296, 2002.
- [3] M. Ajtai, J. Komlós, and E. Szemerédi, “An  $O(n \log n)$  sorting network,” in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 1–9, ACM, 1983.