R1CS Programming

ZK0x04 Workshop Notes

Daniel Lubarov Brendan Farmer

October 21, 2019

Contents

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_{inv}. To verify the result, we enforce

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

2 Zero testing

To assert $x = 0$, we simply enforce

$$(x)(1) = (0) \tag{2}$$

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*

$$y \coloneqq \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise,} \end{cases} \tag{3}$$

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

$$(x)(m) = (y), \tag{4}$$

$$(1 - y)(x) = (0). \tag{5}$$

Outside of the circuit, the prover generates y as in Equation 3, and generates m as

$$m \coloneqq \begin{cases} 1 & \text{if } x = 0, \\ y/x & \text{otherwise.} \end{cases} \tag{6}$$

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

$$(b)(b - 1) = (0). \tag{7}$$

To "split" a field element x into its binary encoding, $(b_1, \ldots, b_n)$, 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

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

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 $(b_1, \ldots, b_n) < |F|$. Such binary comparisons are covered in Section 9.

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

$$x \coloneqq \sum_{i=0}^{n-1} 2^i b_i. \tag{9}$$

This does not require any constraints, unless we don't know whether $(b_1, \ldots, b_n)$ 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

$$z \coloneqq \begin{cases} x & \text{if } s = 0, \\ y & \text{if } s = 1. \end{cases} \tag{10}$$

We can compute this as

$$z \coloneqq x + s(y - x). \tag{11}$$

This requires two constraints: one "is boolean" asertion (Equation 7), assuming s was not already known to be boolean, and another for the multiplication.

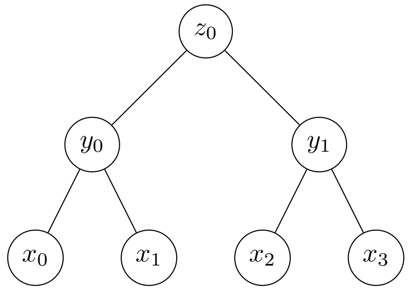
5 Random access

Suppose we have a sequence $S = (x_0, \ldots, x_n)$, and we wish to access the ith element. One approach would be to multiply each element by an equality test (Section 2) comparing that element's index to i, then sum up those products. In particualar,

$$S_i = \sum_{j=0}^{n-1} (i = j) x_j \tag{12}$$

This requires $3n$ constraints: $2n$ for the zero tests and n for the products.

A better approach is to split i into its binary encoding, $(i_0, \ldots, i_k)$, where $k = \lceil \log_2 n \rceil$. Then we can view $(i_0, \ldots, i_k)$ as a path through a binary tree with $(x_0, \ldots, x_n)$ as its leaves. For example, if $n = 4$, we could imagine the following binary tree:



Then for each non-leaf node, we would select one of its two children using the method from Section 4. In the example, we have

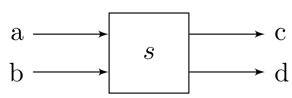
$$y_0 \coloneqq x_0 + b_0(x_1 - x_0), \tag{13}$$
$$y_1 \coloneqq x_2 + b_0(x_3 - x_2), \tag{14}$$
$$z_0 \coloneqq y_0 + b_1(y_1 - y_0). \tag{15}$$

If n is a power of 2, this method takes $\log_2 n + 1$ constraints for splitting i and $n - 1$ constraints for the selection operations, for a total cost of $n + \log_2 n$. If $(x_0, \ldots, x_n)$ is comprised of constant values, then selecting between two leaf nodes becomes a free linear combination, saving us $n/2$ constraints.

If n is not a power of 2, this method can still be used, but we need to think about how to handle out-of-bounds indices. We could assert $i \leq n$ (Section 9), but this may not be necessary depending on the context.

6 2x2 switch

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



In particular, if $s = 0$ then the outputs should be identical 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

$$d \coloneqq a + b - c \tag{16}$$

which does not require any additional constraints.

7 Permutations

Say we want to verify that two sequences, $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$, 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 $\log_2(n!)$.

8 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 R1CS circuit to sort a sequence, we have the prover supply the ordered sequence. Using a permutation network (Section 7), 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 $x_i \leq x_{i+1}$.

9 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

$$z = 2^n + x - y. \tag{17}$$

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 nth bit is our result.

If x and y are unbounded, then the technique described above does not help. A neat solution was described by Ahmed Kosba. First, we split x and y into their binary forms, $(x_0, \ldots, x_n)$ and $(y_0, \ldots, y_n)$. 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 17.

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.

10 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.

10.1 Addition

Recall the addition law for twisted Edwards curves,

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_1 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right). \tag{18}$$

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

However, the operation becomes much cheaper when 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

$$[2](x, y) = \left(\frac{2xy}{ax^2 + y^2}, \frac{y^2 - ax^2}{2 - ax^2 - y^2} \right) \tag{19}$$

which requires 5 constraints.

10.2 Multiplication

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

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 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 1–9, ACM, 1983.

[1] 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: $(q)(y) = (x)$.