PLONK with lookups (draft)

Ariel Gabizon
Aztec

Zachary J. Williamson
Aztec

February 12, 2020

1 Introduction

We can use a reduced form of PLONK's permutation argument, to validate that two polynomials share the same roots. This in turn can be used to verify two polynomials share the same Lagrange coefficients.

For two degree-n polynomials $A(X) = \sum_{i=1}^{n} a_i L_i(X), B(X) = \sum_{i=1}^{n} b_i L_i(X)$. Using a random challenge $\gamma \in \mathbb{F}_p$, we can verify that the two sets $\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_n\}$ are identical:

$$\prod_{i=1}^{n} \frac{(a_i + \gamma)}{(b_i + \gamma)} = 1$$

In addition, we can validate the correctness of subsets within larger sets. For l subsets, we introduce a new challenge β and two selector polynomials $q_1(X) = \sum_{i=1}^{n} \sigma_1(i) L_i(X), q_2(X) = \sum_{i=1}^{n} \sigma_2(i) L_i(X)$, where both $\sigma_1(i)$ and $\sigma_2(i)$ evaluate on H to l unique values. The modified permutation argument becomes

$$\prod_{i=1}^{n} \frac{(a_i + \sigma_1(i)\beta + \gamma)}{(b_i + \sigma_2(i)\beta + \gamma)} = 1$$

2 Implementing arbitrary lookup tables

Consider a PLONK circuit with a program width of m and a depth of n. We can represent witnesses to the circuit's memory cells via w_i for all $i \in [1, \ldots, mn]$.

We wish to prove that a subset of memory cells are copies of elements in a size k precomputed lookup table. We represent table values via t_i for all $i \in [1, \ldots, k]$.

To ensure that all polynomial identities are evaluated modulo the vanishing polynomial of a size-n multiplicative subgroup H, we can split the lookup table into size-n

subsets. i.e. our PLONK circuit now has a *table* width of v, where $vn >= k$.

To validate a sequence of lookups, the high level outline is as follows

1. create a key/value mapping that maps an index in the lookup table to a value
2. apply the key/value mapping to the table reads in program memory
3. isolate *unique* table reads in program memory
4. isolate values from the lookup table t that were copied into program memory
5. validate a subset permutation on the set of unique table reads and the set of used table lookup values

3 Using randomized constraints for key/value mappings

We can use randomized constraints to create key/value mappings for arbitrary combinations of keys and values. This is useful for situations where a key and a value require more values than the set of integers $[1, \ldots, p]$, where p is the order of the circuit's native field arithmetic.

After committing to program memory values, the verifier generates a random challenge α and sends it to the prover. We can discriminate between l distinct lookup tables using the l'th powers of α.

The prover can now commit to a sorted list of table reads. We can define the number of table reads as $e = sn$, where $s <= w$. The set of table reads is represented via $\{r_1, \ldots, r_e\}$, where each table read is a linear combination of two memory cells w_1, w_2, where $r = w_1 + \alpha w_2$. The specific mapping between memory cells and table read elements is described using l randomized constraints, represented by identities $E_1, \ldots, E_l$.

We can now validate a permutation between the set of all table reads

$$r_1, \ldots, r_e$$

, and the set of values produced by applying

$$E_1, \ldots, E_l$$

to every row in program memory.

example: XOR gates, width-4 circuit. Consider the scenario where we wish to evaluate $a^b = c$, where a, b, c are in the range $[0, \ldots, 255]$. All possible XOR combinations

can be represented by 2^16 values in a lookup table. For a row i of program memory, the XOR constraint is

$$E_1(\{w_{(i-1)m+1}, \ldots, w_{im}\}) = w_{(i-1)m+1} + 2^8 w_{(i-1)m+2} + \alpha w_{(i-1)m+3} + \beta + \gamma$$

where $w_{(i-1)m+1} = a$, $w_{(i-1)m+2} = b$, $w_{(i-1)m+3} = c$. Instead of directly mapping memory cells to values in $r_1, \ldots, r_e$, we map the output of the identities $E_1, \ldots, E_l$ to values in $r_1, \ldots, r_e$ instead.

We can use selector polynomials to toggle our randomized constraints on/off for a given row in program memory. This allows us to also mask out locations in program memory that are not table reads when required.

To validate the correctness of the values $\{r_1, \ldots, r_e\}$, we validate that the following permutation holds:

$$\frac{\prod_{i=1}^{n}(\sum_{j=1}^{l} E_j(\{w_{(i-1)m+1}, \ldots, w_{im}\}))}{\prod_{i=1}^{n}\prod_{j=1}^{s}(r_{(i-1)s+j} + q_{(i-1)s+j}\beta + \gamma)}$$

We can assume that, for any given $i \in [1, \ldots, n]$, only one randomized constraint will be active, which ensures that we are still evaluating a 'grand product'.

When the circuit is constructed, the number of table reads per sub-table will be known, which means that we can use a precomputed selector q to map each element of $\{r_1, \ldots, r_n\}$ to their respective sub-table.

Finally, we should note that each identity can map more than one cell in program memory into the lookup read table. For example, we could index a range table using the following constraint:

$$E_{range} = (w_1 + \beta^{range} + \gamma)(w_2 + \beta^{range} + \gamma)(w_3 + \beta^{range} + \gamma)$$

Because we have introduced a selector polynomial to toggle each constraint, each identity can only contain (for a width 4 program) cubic terms, without increasing the degree of PLONK's quotient polynomial $T(X)$.

4 From a sorted list to a lookup argument

We are now at the stage where we can validate that all of a program's table reads have been mapped into the set of values $R = \{r_1, \ldots, r_e\}$, split over s degree-n commitments.

We now need to provide the prover with a mechanism to remove duplicate elements. We can assume that the elements of R are sorted in monotonically increasing order (for each subset). If this is not the case, our final permutation argument will be rejected by the verifier.

We introduce s **mask** commitments, that contain elements $\{m_1, \ldots, m_e\}$. Where all elements take boolean values. N.B. there seems like there should be a way to not need this mask thingy, but it's extremely cheap to commit to using a Lagrange-base SRS.

For a given mask element, if $m_i = 1$, the prover is making a claim that two adjacent values in R are adjacent.

The verifier validate this by evaluating the following:

$$m_i^2 - m_i = 0$$

$$(r_{i-1} - r_i)m_i = 0$$

The latter identity always holds if $m_i = 0$. If $m_i = 1$, then $r_{i-1} = r_i$.

Note that this does not prevent the prover from *not* masking duplicates. This is not necessary because the final permutation argument will fail if duplicates have not been removed.

5 The final permutation check

We almost have a complete lookup argument. The final step is the validate that elements $\{r_1 m_1, \ldots, r_e m_e\}$ is a *subset* of the lookup table values $\{t_1, \ldots, t_k\}$.

We need to accommodate for situations where not all table elements have been read from in a given proof - a strict permutation won't suffice.

To this end, we introduce a second set of l masking polynomials $y_1(X), \ldots, y_l(X)$, containing elements $y_1, \ldots, y_k$.

We use a transition constraint to validate that

$$y_i^2 - y_i = 0$$

We represent our lookup table via l commitments to table keys, $\{tk_1(X), \ldots, tk_l(X)\}$, l commitments to table values $\{tv_1(X), \ldots, tv_l(X)\}$ and l commitments to subtable indices $\{ts_1(X), \ldots, ts_l(X)\}$.

The final permutation check is the following:

$$\frac{\prod_{i=1}^{n}\prod_{j=1}^{l}\left(y_{(i-1)l+j}(tk_{(i-1)l+j} + \alpha tv_{(i-1)l+j} + \beta ts_{(i-1)l+j} + \gamma) + (1 - y_{(i-1)l+j})\right)}{\prod_{i=1}^{n}\prod_{j=1}^{s}\left((1 - m_{(i-1)s+j})(r_{(i-1)s+j} + q_{(i-1)s+j}\beta + \gamma) + m_{(i-1)s+j}\right)}$$

6 Efficiency Analysis

Adding this lookup argument into PLONK incurs the following fixed overheads for the prover:

1. Two new grand product polynomials, to validate the two permutations (TODO: can these be combined somehow without exploding T(X)?)
2. A commitment to $M(X)$
3. A commitment to $Y(X)$

Assuming these are committed to using a Lagrange-base SRS, the number of scalar multiplications will be equal to the number of reads, for the grand product polynomials. Both $M(X)$ and $Y(X)$ can be committed to using group additions instead of group exponentiations, so their overall contribution to prover time is small.

We can also generously assume that less than 1/4 of all memory cells will be table reads. This sets $s = 1$. Similarly, we can likely assume that the set of all lookup table values will be less than n, which sets $l = 1$.

The final number of group exponentiations required by the prover for each table read is 3 - one to commit to $R(X)$, one for each new grand product polynomial.

7 Examples: XOR

We can perform a 32-bit XOR in 4 gates (using an 8bit xor and not the funky sparse bit trick), using a lookup table and one randomized constraint.

$$\begin{array}{cccc} a_0 & b_0 & c_0 & - - - \\ a_1 & b_1 & c_1 & - - - \\ a_2 & b_2 & c_2 & - - - \\ a_3 & b_3 & c_3 & - - - \end{array}$$

With the above structure, we can ensure that all of a_i, b_i, c_i track accumulating sums instead of raw 8-bit values - the randomized constraint can extract the 8-bit slices from the differences between elements. This means we don't have to swap between 8-bit representations and native representations of 32 bit integers, which makes it much easier to combine different arithmetic operations.

Total cost is 4 gates and 4 table reads. The 4 reads contribute 12 scalar multiplications. A gate's base 'cost' is 11 scalar multiplications, making the 32-bit XOR cost approximately 5 gates.

The current best case is the quaternary XOR custom gate, which requires 17 gates. This yields a ¿3x speed increase.

7.1 example: bit rotations on 32 bit integers

Assume we have a 32-bit unsigned integer, represented using 1 memory cell with value a. We wish to evaluate a bit rotation by a fixed number of bits. E.g. $a >>> 5$.

Any rotation by an odd number of bits will 'slice' one 16-bit value in two. In this case into a 11-bit value b_0, a 16-bit value b_1 and a 5-bit value b_2. i.e.

$$a >>> 5 = b_0 + 2^{16-5}b_1 + 2^{32-5}b_2$$

We can now validate the following:
$b = b_0 + 2^{16-5}b_1 + 2^{32-5}b_2$ $a = b_2 + 2^5 b_0 + 2^1 6 b_1$

This requires 3 table lookups, b_0 must come from an 11-bit range, b_1 from a 16-bit range, b_2 from a 5-bit range. Both of above conditions can be verified using a single gate (but would require the ability to validate 2 arithmetic statements per gate. Otherwise 2 gates).

Total cost is 1 gate and 3 reads, which is 2 native gates.

7.2 example: unsigned addition

Let's say we wish to evaluate $a + b = c$. Instead of using a basic add gate, we can use a lookup to ensure that c is within the range $[0, \ldots, 2^{32}]$. Specifically we validate:

$$a + b = c_0 + 2^{16}c_1 + 2^{32}c_2$$

We use a simple transition constraint to validate that $c_2 \in [0, 1]$. We use lookup table reads to validate that c_0 and c_1 are 16-bit values.

Like in the XOR and bit rotation case, we can represent c via accumulating sums in program memory, instead of raw 16-bit slices.

Total cost is 1 gate and 2 reads.

7.3 example: Blake2s

The Blake2s round function applies the following operation to three 32-bit variables a, b, c, where d is a constant rotation value:

$$e = (a + b)^c >>> d$$

By combining the above three sub-programs, we can evaluate this 'add xor rotate' operation in 4 gates and 4 table reads!

Layout in program memory is the following:

$$\left|\begin{array}{|c|c|c|c|}(a+b)_0 & c_0 & e_0 & - - - \\ (a+b)_1 & c_1 & e_1 & b \\ (a+b)_2 & c_2 & e_2 & (b+overflow) \\ (a+b)_3 & c_3 & e_3 & a\end{array}\right|$$

Step 1: all values 'a+b', 'c', 'e' track accumulating sums. So $(a+b)_3 = a+b \bmod 2^{32}$.
Step 2: we use a lookup table that combines an XOR with a bit rotation, instead of performing these in discrete steps.
Step 3: use a transition constraint that validates

$$a + b_3 = a + b + overflow$$

Step 4: use a transition constraint that validates

$$(b + overflow - b)^2 - (b + overlow - b) = 0$$

A Blake2s round consists of 4 of these, which is 16 gates and 16 table reads.

The blake2s hash function requires 80 applications of this round function, which would be 1,280 gates and 1,280 table reads. The table reads would cost 3,840 scalar multiplications, which is equivalent to 350 native gates. So the total cost would be $\approx 1,650$ 'gates'. Current best estimate with quaternary decomposition constraints is $\approx 6,700$. R1CS requires $> 20,000$.