SHA as the union of two sets of constraints:

As Dmitry says, SHA can be expressed in terms of two types of constraints,

- a) “pure lookups” (“I know tuple x being in this table T”). Call the committed to the looked-up values *clop*.
- b) wirings (“hey, see that looked up value over there? Well, it also shows up over there in another tuple we are looking up”).

(*Not important, but in my understanding, sometimes depending on the implementation of SHA, this also wiring may involve slightly more than pure “pointing to values”. It may require a bona-fide arithmetic constraint (e.g., “I’m performing a field addition between these two looked up values; its result will be looked up later”))

Approaches to prove the above

The stand-alone possibility One possibility one has is to use a stand-alone proof system that supports all sort of gates: lookups, polynomial gates, etc. Something Plonky-ish, in other words.

Limitations of lookup gates in standalone PLONKish systems: we pay in linear size of the table in the prover. “Pure” lookup systems such as Caulk[+], Baloo, cq, allow one to prove in time o(|T|) (assuming a preprocessing).

Now, one may think of using one of these pure lookup schemes. Except that one cannot. That is, one cannot use them on their own (for SHA): they are guaranteeing only that a set of committed “values” is looked where you expect them to be looked. What is left is: is the relation among these committed looked-up values the one from the “wiring” (item (b) in our description above).

So, my understanding is that, under the assumption that we want to exploit a pure lookup scheme, one would need to prove that the wires committed in *clop* do also satisfy some additional relation. **Our (hypothetical) choice for that additional proof is Groth16** (Why Groth16? It is not the best choice in absolute terms, but it is the most engineered choice we have in-house for now.).

But how efficient is this lookup+groth16 for sha?

So, so far, we picked one lookup scheme—Baloo for concreteness—and (Lego)Groth16 for the wiring (as well as for anything else we want to prove on that SHA result). Performance of the bundle?

- We can measure lookup proving efficiency
    - We have an approximate number of lookups and can benchmark Baloo on them
- We do not know how to measure the “wiring” among the lookups because we do not understand how many wiring constraints, _as_ R1CS_ constraints, there are.

And why are we interested in this?

Because from preliminary numbers from measuring Baloo, it is possible this approach may save 4-8x compared to Groth16 standalone if the additional wiring constraints are not many (e.g. if they are <5K)

Example of what is besides lookups in SHA256

The following is an example from
https://zcash.github.io/halo2/design/gadgets/sha256/table16.html

Constraints:

- `s_upp_sigma_0` (Σ_0 constraint): $LHS - RHS + tag + decompose = 0$

$$
\begin{aligned}
tag &= constrain_1(a_0\omega^{-1}) + constrain_2(a_0\omega) \\
decompose &= a(2) + 2^2 b(11) + 2^{13} c(9)^{lo} + 2^{16} c(9)^{mid} + 2^{19} c(9)^{hi} + 2^{22} d(10) - A \\
LHS &= \mathtt{spread}(R_0^{even}) + 2 \cdot \mathtt{spread}(R_0^{odd}) + 2^{32} \cdot \mathtt{spread}(R_1^{even}) + 2^{33} \cdot \mathtt{spread}(R_1^{odd})
\end{aligned}
$$

$$
\begin{aligned}
RHS = \; & 4^{30}\mathtt{spread}(a(2)) + 4^{20}\mathtt{spread}(d(10)) + 4^{17}\mathtt{spread}(c(9)^{hi}) + 4^{14}\mathtt{spread}(c(9)^{mid}) + 4^{11}\mathtt{spread}(c(9)^{lo}) + \mathtt{spread}(b(11)) \\
& 4^{21}\mathtt{spread}(b(11)) + 4^{19}\mathtt{spread}(a(2)) + 4^{9}\mathtt{spread}(d(10)) + 4^{6}\mathtt{spread}(c(9)^{hi}) + 4^{3}\mathtt{spread}(c(9)^{mid}) + \mathtt{spread}(c(9)^{lo}) \\
& 4^{29}\mathtt{spread}(c(9)^{hi}) + 4^{26}\mathtt{spread}(c(9)^{mid}) + 4^{23}\mathtt{spread}(c(9)^{lo}) + 4^{12}\mathtt{spread}(b(11)) + 4^{10}\mathtt{spread}(a(2)) + \mathtt{spread}(d(10))
\end{aligned}
$$

- **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 $c(9)^{lo}, c(9)^{mid}, c(9)^{hi}$

The “spread” are lookup invocations. They are assembled together through additions and multiplication to constant here.
The lines above is easy to convert into R1CS constraints by itself. Our challenge is doing that for the whole sha chip (it would require sort of reimplementing it isolating lookups from all the rest). We wonder if there are easier approaches or anybody has suggestions on how to go at that economically.