Author: 0xParc
Type: #source #class
Link: https://learn.0xparc.org/halo2/
Topics: cryptography Computer Science

Session 2: Halo2 API & Building a Basic Fibonacci Circuit

Class

Slides: Link to Slides

Halo2 Columns Type:

- **Advice Columns:** Private inputs (vary for each proof) + witness values
- **Instance Columns:** Public Inputs (vary for each proof)
- **Fixed Columns:** Constants & lookup tables (fixed circuit configuration)
- **Selector Columns:** Special instance of the fixed columns, binary values that control if specific gates will be turned on/off (fixed circuit configuration)

The Layouter participates when you assign witness into the table. A quick heuristic for the Halo2 cost-model includes rows, columns, and polynomial degree. Interestingly, adding more advice columns can significantly reduce the number of rows, aiding the layouter in optimizing the circuits.

Since the prover operations, including Fast Fourier Transforms (FFT) and certain elliptic curve operations, will be applied on the vectors (columns), it is computationally wise to reduce the number of rows at the expense of the number of columns. However, the area remains unchanged in that exercise.

To use the permutation argument, you need to call `enable_equality` on the witness at the advice column level.

The Rust Halo2 has the following structure:

- A so-called Chip, which is essentially some sort of "zk chip" where the witnesses are manipulated and constrained. This is done by creating and arranging a set of advice and selectors into gates, which are returned in a config format by the `configure` function.

- A so-called Circuit, which takes witnesses and instantiates the chip(s). It's important to note that the synthesizer assigns values to columns (the witnesses) by calling a function that resorts to the layouter, so each "region" (think of the table of advice, instance, etc.) can be compressed.

The mental model is that the circuit and the chip could have been a single object. But by splitting them, it allows for better abstraction and the possibility to combine chips. The steps are as follows:

1. Create a circuit without witness.
2. Configure the circuit at keygen.
3. Synthesize: makes the actual circuit, called both at keygen and proving time!

Session 3: Halo2 API & Building a Basic Fibonacci Circuit (Part 2)

`assign_advice`: witnessing something

`assign_advice_from_instance`: copying something from the instance column

This session showcases an optimization technique. Instead of making a row per Fibonacci, we work on the column dimension. The column then gets compressed into a single region versus one region per Fibonacci step. One heuristic to understand optimization is to reduce loops in the synthesizer, which means that the least region must be created, as each loop seems to create a region with the layouter being passed again. Another takeaway is to use chips as selectors for another chip. For example, the `is_zero_circuit` can be used as a selector within other circuits!

Session 4: Circuit Exercise (Part 1)

When querying a selector, we don't specify the rotation because we always query at the current rotation, but advices are then indexed around the current selector location.

`fold`: takes an initial value, as well as a closure that takes an accumulator and an element, and will return the value of the accumulator on the next iteration, simply by combining it with the provided element.

We can look up arbitrary expressions instead of single cells! Even if we're searching for values, it remains more efficient as these values can be precomputed and checked for during proving tasks (e.g., a range check, where we can verify that a byte is indeed a byte by checking the table).

Session 6: Circuit Exercise (Part 3)

Steps:

1. Ensure config elements are composed of headers and referenced tables of a drawing of your circuit table.
2. Write the lookup argument/constraints (`chip::configure`).
3. Write assignments.

Recall on selector: Any selector that we intend to use as a lookup must be a complex selector so that it doesn't participate in circuit optimization and retain its binary state.

Key takeaways from building the `decomposer_range_check` include optimization considerations. Take the case where `num_bits` is not divisible by the `LOOKUP_NUM_BITS` discussed at the end of the videos. In this case, you could quickly inflate your constraints since you need to hardcode the `k` value and will have to increase it to handle higher `num_bits`.

The lookup is interesting but needs to be balanced: Do you pass it around circuits? Does it meaningfully reduce rows vs no lookup table, or reduce polynomial size?

Also, consider edge cases of enabling equality to handle inter-row assignments, complex selectors to avoid optimizations making them non-binary, etc. The Halo2 API is not straightforward, and one should be careful when building circuits!

Session 7: Custom Gates

(Empty Section)

Halo2 Docs

Advice: All intermediary values generated in the circuit to reach the final proof.

Witness: Public/private inputs + advice.

A circuit is chunked into regions: each region contains a disjoint subset of cells, and relative references only ever point within a region.