

Little Things I've Learned from Developing Halo2 Circuits

How do you think like a circuit dev?

Chih-Cheng Liang(CC)

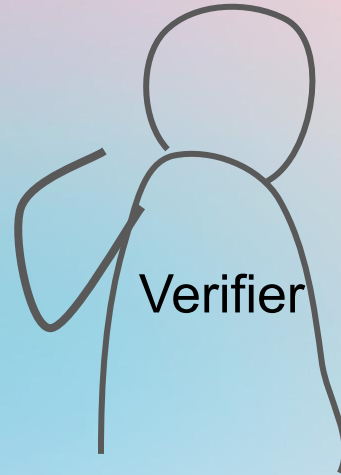
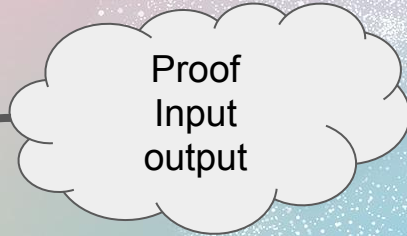
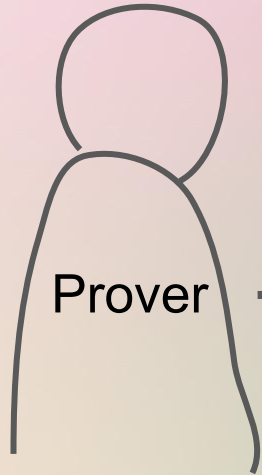
Privacy and Scaling Exploration

```
def algorithm(input):
```

```
    ...
```

```
    return output
```

I don't rerun the computation. But I'm convinced that the output is correctly computed



Setup time



Verifying key

Verifier

Proving key

Prover

Define constraints

Send keys to Provers & Verifiers

Proving time



Proving key

Prover

Proof

Ok /
Not ok

Verifier

Verifying key

Proof

Fill in computation trace

Generate Proof

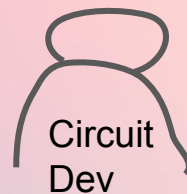
Verify Proof

The Pain: A circuit dev has a different brain



```
def foo(opt: bool):  
    a = 1  
    b = 1  
    if opt:  
        a = 2  
        b = 2  
    for _ in range(10):  
        a = a + b  
        b, a = a, b  
    return b
```

- Arithmetization
- Computational trace



```
2 4    a_next - opt * a_cur - 1 == 0  
4 6    b_next - opt * b_cur - 1 == 0  
6 10   a_next - a_cur - b_cur == 0  
10 16  
16 26  
26 42  
42 68  
68 110  
110 178  
178 288
```



Rules of the game

Computation is represented in Finite Field Arithmetics

- An integer less than a number p
- Say $p = 3$
- Wrap over
 - $2 + 2 = 1 \pmod{3}$
- p is usually a very large number. For example, 254 bits

The grid: Layout computation traces here

- A cell: you can fill in a field element
- Columns: extra column would induce extra proving/verifying cost
- Rows: use as much as you can. But there's a cap. You can only use up to 2^{18} rows.

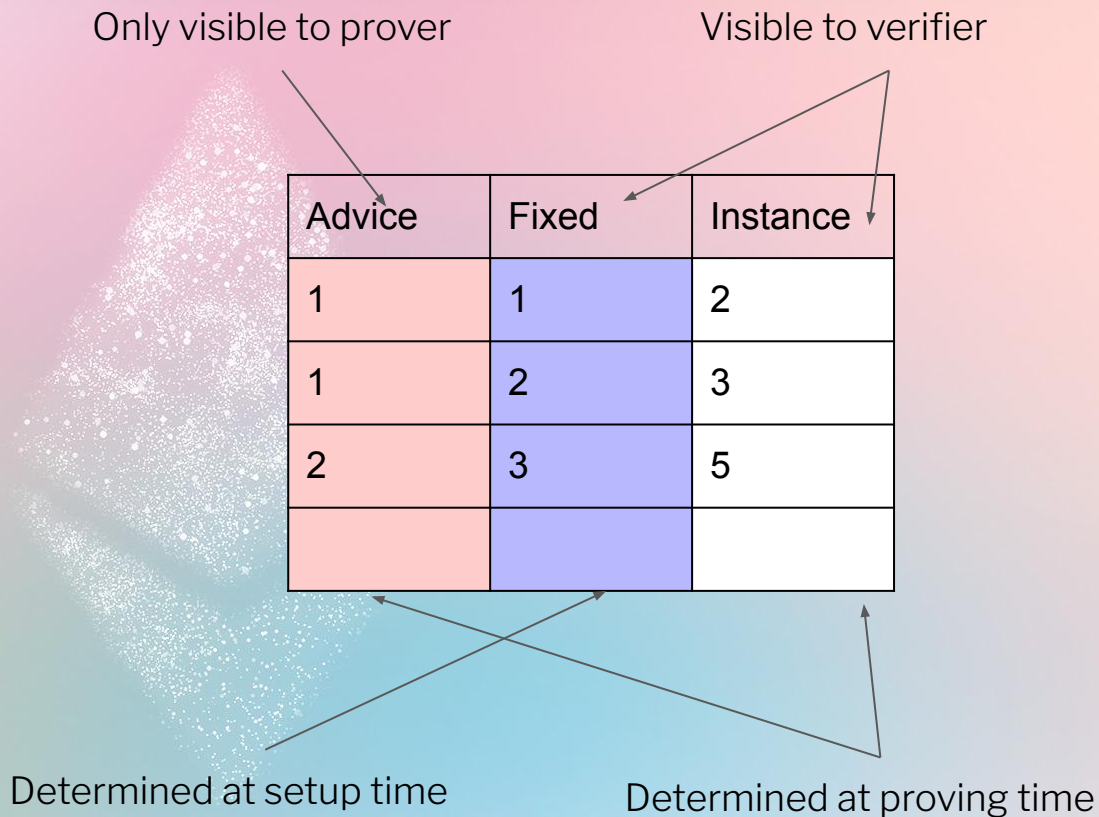
Free but capped →

More costly →

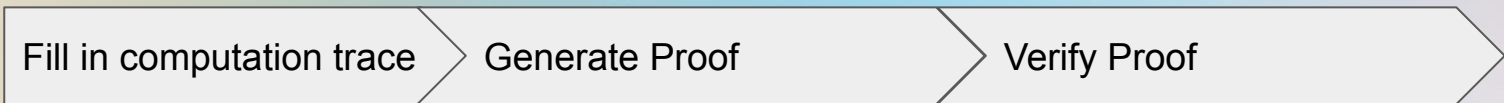
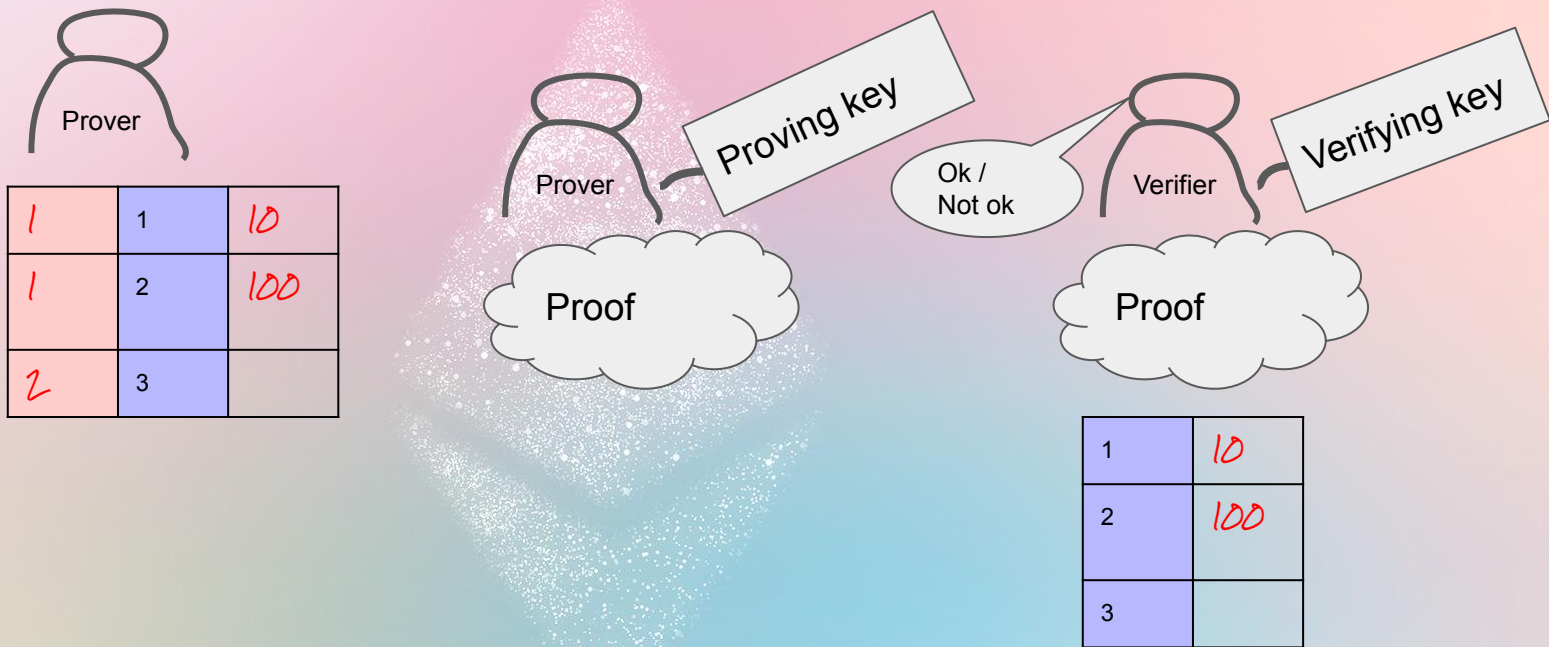
1	1	2
1	2	3
2	3	5

Types of columns: Who sees what and when?

- Pink columns are advice columns. The prover assigns private values at proving time.
- Purple columns are fixed columns. Values are assigned at setup time.
- White columns are instance columns. The prover assigns public values at proving time.



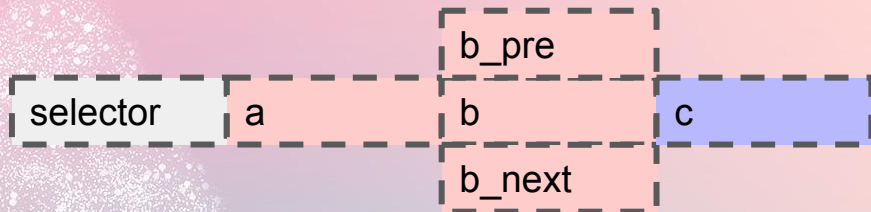
Proving time



Constraints:
What values in the cells are valid?

Custom gates: Define constraints in polynomials

- Define an expression that cells in the gate must hold
- Selector is an implicit fixed column. It has value 0 or 1, it can enable or disable the gate
- We can refer to previous cell, next cell, or cell that's 20 rows later. It comes with costs of kzg opening.



$$\text{Selector} * (a - b - b_{\text{next}}) = 0$$

Everything is a polynomial.
I'm a polynomial.

- Vitalik



Custom Gate Example: Fibonacci

1, 1, 2, 3, 5, 8, 13

q_fib	left	right	sum
1	1	1	2
1	1	2	3
1	2	3	5
0			

q_fib	left	right	sum
-------	------	-------	-----

$$q_fib * (left + right - sum) = 0$$

Example borrowed from

<https://github.com/therealyingtong/halo2-hope/blob/main/src/fibonacci.rs>

Custom Gate Example: Fibonacci

1, 1, 2, 3, 5, 8, 13

q_fib	left	right
1	1	1
1	1	2
1	2	3
0		5

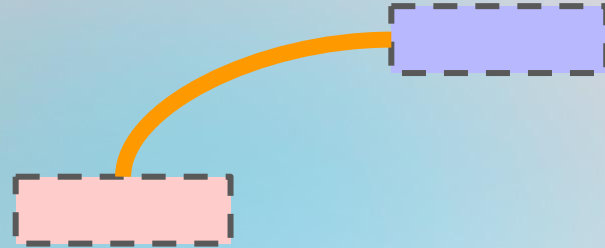
q_fib	left	right
		right_next

$$q_fib * (left + right - right_next) = 0$$

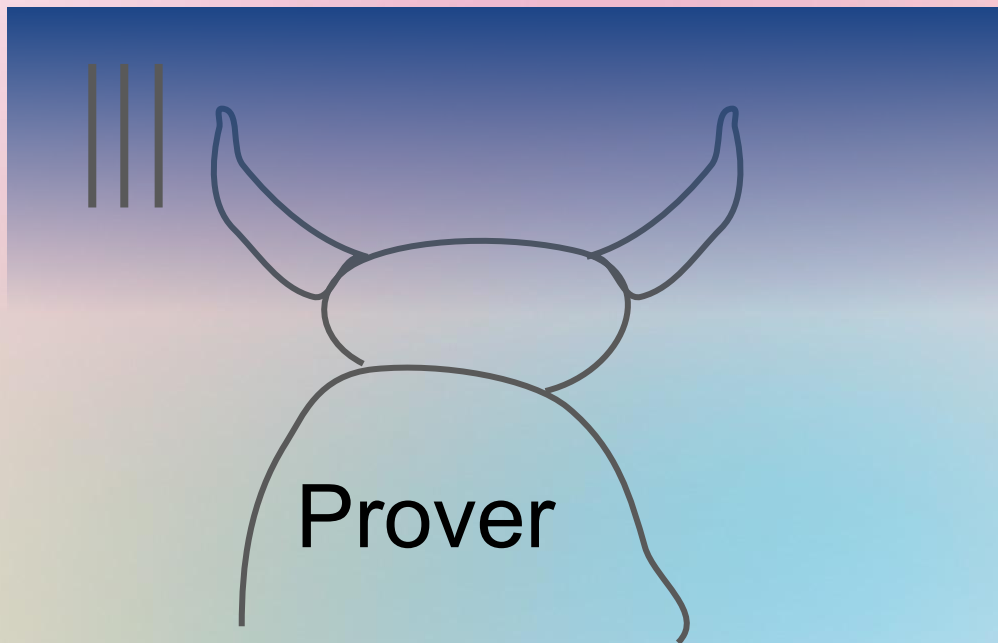
Copy Constraints/ Equality check

- A copy constraint binds two cells so that their assigned values must be the same.
- Cheapest constraint. Use as much as we can.
- Cell positions are determined at setup time. Can't change it at proving time.

10	10
1	100
100	



Rule: Prover can be evil





Tricks



Limiting options

Limiting cell values

Want: The value in the cell can be only one of 1, 2, 3

Gate expression: $(x - 1)(x - 2)(x - 3) = 0$

3



100



$$(100 - 1)(100 - 2)(100 - 3) \neq 0$$

2



0



$$(0 - 1)(0 - 2)(0 - 3) \neq 0$$



Converting If-else

Example

```
def foo(a:int, b: int, happy: bool):  
    if happy:  
        return a + b  
    else:  
        return a * b
```

5	5 (A)
6	6 (B)
1	1 (HAPPY)
11	11 (OUTPUT)

5	5
6	6
0	0
30	30

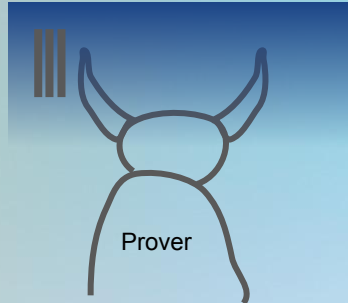
$$1 * (5 + 6) + (1 - 1) (5 * 6) - 11 = 0$$

$$0 * (5 + 6) + (1 - 0) (5 * 6) - 30 = 0$$

Gate expression:

$$\text{happy} * (a + b) + (1 - \text{happy}) (a * b) - \text{output} = 0$$

$$\text{happy} (1 - \text{happy}) = 0$$



5	5
6	6
3	3
-27	-27

$$3 * (5 + 6) + (1 - 3) (5 * 6) - (-27) = 0$$



Converting For loops

Example: trivial one

```
def foo():  
    r = 0  
    for _ in range(5):  
        r += 5  
    return r
```

r	const	out
0	0	25
5		
10		
15		
20		
25		

Gate expression:

$$r_{\text{next}} - r_{\text{cur}} - 5 = 0$$

How about ...

```
def foo(n: int):  
    r = 0  
    for _ in range(n):  
        r += 5  
    return r
```

computation trace

r	const	in/out
0	0	5
5		25
10		
15		
20		
25		

r	const	in/out
0	0	3
5		15
10		
15		
15		
15		

```
def foo(n: int):  
    r = 0  
    for _ in range(n):  
        r += 5  
    return r
```

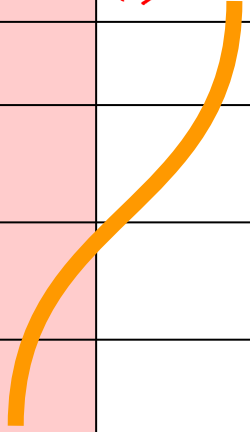
- We don't have unlimited rows for **n**. So we need to cap **n** to a max possible value, say 5
- Checking the range of **n** is out of scope

Copy cells

- r0 === const0
- r5 === out1

Attempt: Maybe try add a prover witnessable selector

s	r	in/out
1	0	3
1	5	15
1	10	
0	15	
0	15	
0	15	

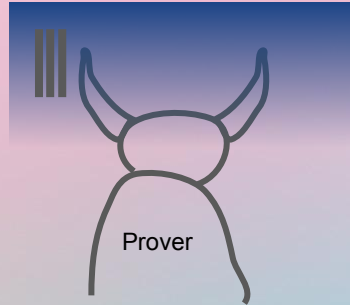


Gate expression:

$$s * (r_{\text{next}} - r - 5) + (1 - s) * (r_{\text{next}} - r) = 0$$

How do you prevent the prover from doing this?

selector	r	in/out
0	0	3
1	0	10
0	5	
1	5	
0	10	
1	10	



I can convince you $5 \cdot 3 = 10$

Observe some instances

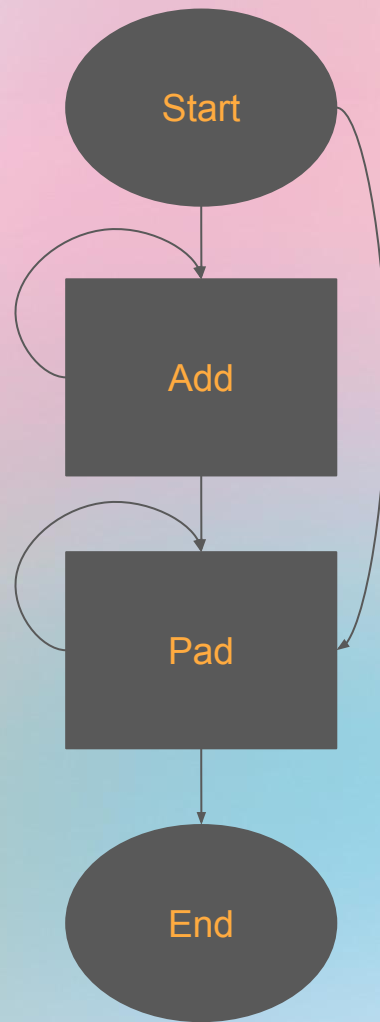
s	r	io
0	0	0
0	0	0
0	0	
0	0	
0	0	
0	0	

s	r	io
1	0	3
1	5	15
1	10	
0	15	
0	15	
0	15	

s	r	io
1	0	5
1	5	25
1	10	
1	15	
1	20	
0	25	

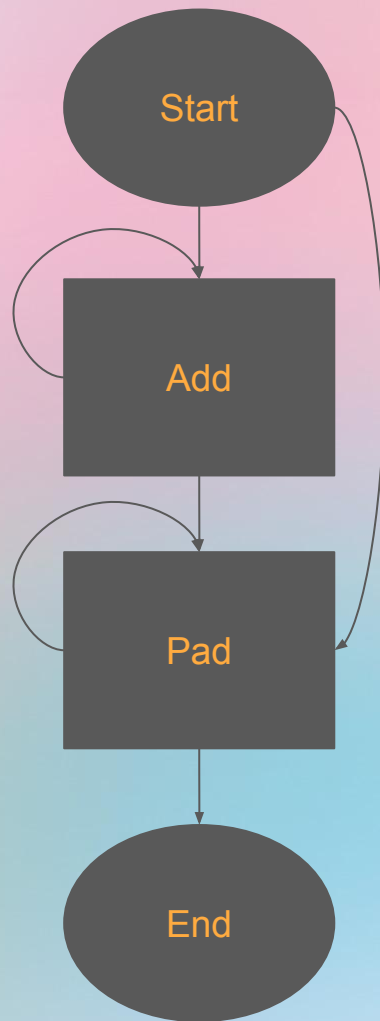
State transition trick

r	io
0	3
5	15
10	
15	
15	
15	



State transition trick

s	r	io
1	0	3
1	5	15
1	10	
0	15	
0	15	
0	15	



Gate expression:

$$s * (r_{next} - r - 5) + (1 - s) * (r_{next} - r) = 0$$

selector should be boolean

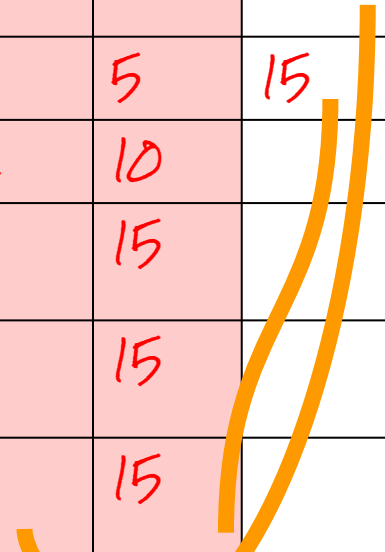
$$s * (1 - s) = 0$$

if current selector is 0, then the next has to be 0

$$(1-s) s_{next} = 0$$

Final touch: adding input check

s	in	r	io
1	0	0	3
1	1	5	15
1	2	10	
0	3	15	
0	3	15	
0	3	15	



Gate expression:

$$s * (in_next - in - 1) + (1-s) * (in_next - in) = 0$$

Copy cells

- in5 == io0
- r5 == io1

Gate expression:

$$s * (r_next - r - 5) + (1 - s) * (r_next - r) = 0$$

selector should be boolean
 $s * (1 - s) = 0$

if current selector is 0, then the next has to be 0

$$(1-s) s_next = 0$$

Why are we doing all these?

Quoting Edu

s	in	r	io
1	0	0	3
1	1	5	15
1	2	10	
0	3	15	
0	3	15	
0	3	15	

"In a CPU we have a set of instructions that take variable time, and in each cycle we run one instruction."

"When we have an if/else the execution only follows one path"

"In an arithmetic circuit we don't have cycles! We must "flatten" all the possible execution paths, and "run" them all at the same time and discard the paths that we don't need by multiplying by 0."



Take away

- Challenges for circuit developers
 - working with "computation trace" instead of the execution.
 - **Need to flatten the path**
 - working with field element arithmetics instead of bits and bytes.
 - **need to work with maths and equations.**
 - working with verification mindset
 - **need to prevent the prover from cheating**
- Tricks
 - if-else constraint boolean and do $s^*(\text{true path}) + (1-s)^*(\text{false path})$
 - complicate loops: identify state transition and design constraints for them.

