Circom Basic

HDLs: From Digital to Arithmetic

Circom is an HDL(Hardware description Language) for R1CS

- circom wires : R1CS variables
- circom gates: R1CS constraints

a circom circuit does 2 things:

- sets variable values
- creates R1CS constraints

Circom Online tool

Online Circom test : https://zkrepl.dev/

https://zkrepl.dev/ allows you to test your circuit very rapidly on inputs in comment JSON format :

```cpp
/* INPUT = {
    "a": "5",
    "b": "77"
} */
```

Example :

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";
// include "https://github.com/0xPARC/circom-secp256k1/blob/master/circuits/bigint.circom";

template Example () {
    signal input a;
    signal input b;
    signal output c;

    var unused = 4;
    c <== a * b;
    assert(a > 2);

    component hash = Poseidon(2);
    hash.inputs[0] <== a;
    hash.inputs[1] <== b;

    log("hash", hash.out);
}

component main { public [ a ] } = Example();

/* INPUT = {
    "a": "5",
    "b": "77"
} */
```

role of each file in circom

- Circom `.circom`
- Input `.json`
- PoT `.ptau`
  - power of tau : 大随机数, 约束量越大, 文件越大, 直接决定了电路的安全性
  - STARK 里的 Transparent : 我不需要这样一个 tau 也可以进行安全证明 (只是 Proof 会比较大)
- Circuit `.wasm` 程序生成, 可供电路进行约束验证的证据
- Proving key `.zky`
- Verification key `.vkey`

signal

In circom, all $output$ **signals** of the main component are **public** (and cannot be made private)

the $input$ signals of the main component are private if not stated otherwise using the keyword **public** as above. The rest of signals are all private and cannot be made public. (如果没有使用关键字 public，main component 的输入信号是 Private 私有的。其余的信号也都是私有的，不能公开)

Thus, from the programmer's point of view, only public input and output signals are visible from outside the circuit, and hence no intermediate signal can be accessed. (因此，从程序员的角度来看，从电路外部只能看到公共输入和输出信号，因此无法访问中间信号)

Circom syntax

Circom 的语法类似 javascript 和 C ，提供一些基本的数据类型和操作，例如 `for、while、>>、array` 等。

关键字 :

- `signal`： 信号变量，要转换为电路的变量，可以是 `private` 或 `public`
- `template`： 模板，用于函数定义，就像 JS / Solidity 中的 `function`
- `component`： 组件变量，可以把组件变量想象成对象，而信号变量是对象的公共成员

Circom 也提供了一些操作符用于操作信号变量:

- `<--, -->`： 这些操作符为信号变量赋值，但不会生成约束条件
- `===` ： 这个操作符用来定义约束, it Must be `rank-1`
- `<==, ==>`： 这两个操作符用于连接信号变量，同时定义约束

```
ab <-- a*b
ab === a*b
c <-- ab * ab
c === ab * ab
```

1. 指令 `a <-- a*b` 意味着在执行阶段，把 `a*b` 赋值给 `ab`，并在编译约束时忽略掉
2. 另一方面， `ab === a*b`，它在编译期间被编译为电路结构，但在执行期间被忽略 (?)
3. 所以说在编写 Circom 时，是在同时编写属于 2 种不同编程范式的 2 个不同程序
4. `<==` means `<--` + `===` , 即 `赋值` + `定义约束`
5. `==>` 是一种连接运算符，用于实现信号的传递和组件的互连。它表示将左侧的 `output` 连接到右侧信号端口的 `input` 上：

```
// 将输入信号 `in` 连接到组件 `myComponent` 的输入上，
// 并将组件 `myComponent` 的输出连接到输出信号 `out` 上。
in ==> myComponent.in;
myComponent.out ==> out;
```

虽然通常会编写等效的赋值和约束，但有时需要将它们分开。一个很好的例子就是电路 `IsZero`

`IsZero` 电路

Circom 为什么不能直接支持类似 `in==0` 的判断 ?

1. Circom 使用算术电路来构造零知识证明,每个门对应一条多项式计算。
2. 相等判断不满足电路门的输入输出需求,因此无法构造等式门。
3. 另外, 传统的编程语言支持的 `if/else` 分支,也不是电路门的形式,因此 Circom 不直接支持 if/else
4. 所以不能直接判断 `in==0` , 而要转换为 `inv` 的计算,来实现等价的多项式关系。

Fortunately，Circom 有一个好用的库，包括一个 `IsZero` 返回 `0` 或 `1` 取决于输入是否为零的电路，如下所示：

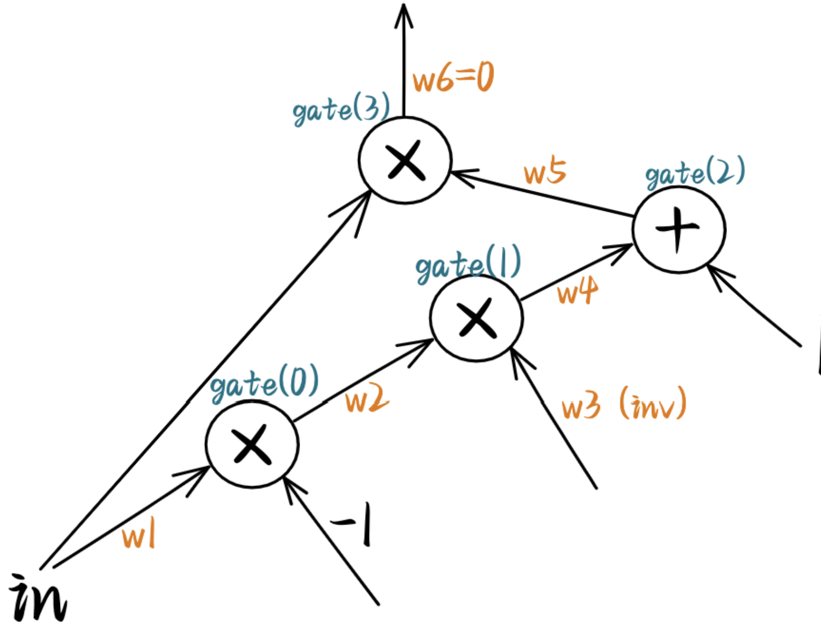
```
template IsZero () {
    signal input in;
    signal output out;
    signal inv;

    inv <-- in!=0 ? 1/in : 0;  // 赋值, inv 即 inverse, 倒数
    out <-- -in*inv +1;   // 赋值
    out === -in*inv +1;   // 电路约束 gate0, gate1, gate2
    in*out === 0;         // 电路约束 gate3
}
```

请注意 `inv <-- in!=0 ? 1/in : 0` 这个表达式没有用 `==` 约束, 所以它不会被计入电路结构, 所以可以使用判等 , 可以使用三元运算符判断, 这里和普通的图灵完备编程语言没什么区别

- if `in` 是 0 ，那么 $out = -in \times 0 + 1 = 1$
- if `in` 是 1 ，那么 $out = -in \times \frac{1}{in} = 0$
- ↑ 可以看到这行表达式可以让 `in * out` 始终 `== 0`

而 `out === -in*inv +1` 和 `in*out === 0` —— 这 2 行是电路结构，Draw the arithmetic `circuit` representation of `IsZero` :



- w_i is `wire` (电线), $gate(i)$ 是电路门
- in 是输入值, -1, 1 是输入常量

如图可见 `out === -in*inv +1;` 和 `in*out === 0` 这 2 个约束语句已经将电路描述出来了

以上代码可以进一步优化成：

```
inv <-- in!=0 ? 1/in : 0;
out <== -in*inv +1;      // 优化: [赋值] 和 [约束] 合并
in*out === 0;
```

IsZero Tips

约束 `in * out === 0` 将在编译时约束 `in` 或 `out` 为零

> 在编译期间将其转换为 R1CS 方程，并在执行期间用于验证电路的正确性。

表达式 `inv <-- in != 0 ? 1/in : 0` 是用于执行阶段的 hint，用于在生成证明过程中计算辅助信号 inv 的值。这个表达式并不会被直接编译到约束中 (gpt-4)

我们需要告诉 Circom 如何在 `Witness generation` 阶段计算 `inv` 的值：这就是 `inv <-- in!=0 ? 1/in : 0` 表达式的用处。Cairo 恰当地称这种表达式为 hints(暗示)。

请注意，我们在这里使用的运算比二次表达式复杂得多：我们有条件、比较和除法。这是因为**执行阶段与编译阶段生成的约束无关**，因此**不受生成 ZKP 的限制**。

因此，在**执行**阶段工作时，我们又回到了非常常见的编程方式，可以访问额外的运算符、控制流语句，甚至辅助函数。

约束不足的 Bug

然而，执行阶段和约束生成阶段之间的这种差异可能会导致非常严重的错误。回到 `IsZero` 例子，如果我们忘记写一个约束，比如说 `in*out === 0`，会发生什么？

让我们按照已知的安全工作流程进行操作，假设 `in=3` 输入：

```
inv <-- in!=0 ? 1/in : 0;  // 赋值, inv 即 inverse, 倒数
out <-- -in*inv +1;        // 赋值
out === -in*inv +1;         // 电路约束 gate0, gate1, gate2
        // in*out === 0;      <--- 约束不足 (Underconstrained)
```

1. 电路编译得很好，因为没有错误
2. `Witness generation` 将遵循执行指令，并正确计算值 $inv = \frac{1}{3}$, $out = 0$
3. 由于约束 `out === -in*inv +1` 成立，证明者会根据 Witness 和公共输出生成 proof
4. 验证者会接受它是有效的，相信证明者有一个私有的非零值。

到目前为止，一切都很好

但是，如果一个**恶意的证明者想要让我们相信他们拥有一个为 0 的值**，而实际他只有一个 `in=3`，会发生什么呢？

1. 恶意用户可以手动制作一个 `in=3, inv=0 (原本 inv 应等于 1/3), out=1` 的 `witness`.
   1. 因为缺乏 `in*out === 0` 约束, 所以作弊者可以去伪造不符合约束的 `Witness`
2. 由于 witness 满足编译出的电路约束 `out === -in * inv +1` ，因此 `Prover` 成功为 `Witness` 生成 `Proof`
3. 验证者欣然接受，**认为用户有一个为 0 的值，而实际上没有**

这种问题被称为欠缺约束计算，特别难以捕捉。

任何将电路视为黑盒的单元测试永远不会触发它们，因为它们按照电路的执行逻辑来构建见证 (witness) .
重现它们的唯一方法是手动组装具有自定义逻辑的见证，这需要你确切地知道正在寻找的攻击是谁。

或者，有一些努力来验证你编写的关于要编码的功能的电路的可靠性，例如Ecne。

这里的底线是，**当你编写与约束生成代码分开的执行代码时**，应该特别小心，因为它为欠缺约束的 ZK 程序打开了大门

剪刀石头布的 Circom 实现

用 Circom 编写剪刀石头布问题的电路很有趣。由于不可能在约束中编写条件表达式，因此需要求助于基于数学的技巧。作为一个简单的例子，第一个电路通过检查它是 0、1 或 2 来验证游戏的输入，如下所示：

```
// input must be 0/1/2 (石头/剪刀/布)
template AssertIsRPS() {
  signal input x;
  signal isRP <== (x-0) * (x-1);
  isRP * (x-2) === 0;
}
```

下面的代码用类似的结构来计算单轮得分， `IsEqual` 电路来自 CircomLib：

```
// Returns the score for a single round, given the plays by x and y
template Round() {
  signal input x, y;
  signal output out;

  // ensure that each input is within 0,1,2
  AssertIsRPS()(x);
  AssertIsRPS()(y);

  // check if match was a draw (平局)
  signal isDraw <== IsEqual()([x, y]); // IsEqual from circomlib

  signal diffYX <== (y+3)-x;   // +3 : circom 不操作负数

  // y wins if y-x = 1 mod 3
  signal yWins1 <== (diffYX-1) * (diffYX-4);
  signal yWins <== IsZero()(yWins1);

  // x wins if y-x = 2 mod 3
  signal xWins1 <== (diffYX-2) * (diffYX-5);
  signal xWins <== IsZero()(xWins1);

  // check that exactly one of xWins, yWins, isDraw is true
  // we probably can do without these constraints
  signal xOrYWins <== (xWins - 1) * (yWins - 1);
  xOrYWins * (isDraw - 1) === 0;
  xWins + yWins + isDraw === 1;

  // score is 6 if y wins, 3 if draw, 0 if x wins
  // plus 1, 2, 3 depending on the choice of RPS
  out <== yWins * 6 + isDraw * 3 + y + 1;
}
```

最后，最外层的电路循环遍历可参数化的轮数 `n` ，并汇总分数。
请注意，这里可以使用循环，因为它取决于 `n`，这是编译时已知的模板参数。

> 编译电路的时候, 由用户传入

```
template Game(n) {
  signal input xs[n];
  signal input ys[n];
  signal scores[n];
```

```
  signal output out;

  var score = 0;
  for (var i = 0; i < n; i++) {
    scores[i] <== Round()(xs[i], ys[i]);
    score += scores[i];
  }

  out <== score;
}
```

算术电路及常见设计方法：

算术电路是零知识证明电路的核心

零知识程序和其他程序的实现不太一样。首先，你要解决的问题需要先转化成多项式，再进一步转化成电路。例如，多项式 $x^3 + x + 5$ 可以表示成如下的电路：

```
sym_1 = x * x // sym_1 = x²
sym_2 = sym_1 * x // sym_2 = x³
y = sym_2 + x // y = x³ + x
~out = y + 5
```

Circom 编译器将逻辑转换为电路。通常我们不需要自己设计基础电路。

如果你需要一个哈希函数或签名函数，可以在 circomlib(一个 JS 库) 找到。

提示 (hints/advice)

由于算术电路的约束性, 每个门电路都会转换为约束(Constrains) , 进而增加 Prove 和 Verify 的工作量, 我们可以将复杂的计算过程变为**预先计算的提示值**, 在电路中对提示值进行验证, 从而降低 Prove 和 Verify 的工作量

`IsZero` 函数：

```
  inv <-- in != 0 ? 1/in : 0; // 赋值
  out <-- -in * inv +1;       // 赋值 ①
  out === -in * inv +1;       // 约束 ②
```

1. 利用提示值 `inv` 计算输出 `Output`
2. 验证 输入 输出 符合约束

情况	输入 $input$	提示值 inv	输出 $output$	$input \times output$
非零输入	4	1/4	0	0
为零输入	0	0	1	0
非零时，恶意提示	4	1/5	1/5	4/5
为零时，恶意提示	0	1/5	1	0

恶意提示即伪造提示值 `inv` , 如 input = 4 & inv = 1/5 , 不满足电路的约束 : $input \times output = 0$

如果不用提示方法, 需要用费马小定理 $input^{p-1} = 1$ 证明, `P` 很大时, 计算次数非常多

> Circom 中的 `IsZero` 函数用于检查电路中的某个值是否为零。在零知识证明中，我们希望尽可能地保持信息的私密性。费马小定理（Fermat's Little Theorem）可以在不透露实际值的情况下实现这一目标。
>
> 费马小定理指出，如果 p 是一个素数，且 a 是一个不被 p 整除的整数，那么 $a^{p-1} \equiv 1 \pmod p$。换句话说，a^{p-1} 除以 p 的余数是 1。
>
> 在这里，我们可以利用费马小定理来检查一个值是否为零。如果 input 是零，则 $0^{p-1} \equiv 0 \pmod p$，而不是 1。当 input 不为零时，费马小定理成立，即 $input^{p-1} \equiv 1 \pmod p$。通过检查 $input^{p-1}$ 是否等于 1，我们可以在不透露 input 实际值的情况下判断它是否为零。
>
> 然而，费马小定理并不能直接证明 input 为零的情况，而提示方法（Hint Method）可以提供一个额外的“提示”值，使证明者可以证明 input 为零，同时保持零知识。在实际应用中，根据证明的需求和性能考虑，可能会使用提示方法或费马小定理的 IsZero 函数。 (GPT)

二进制化 (Num2Bits)

```
5 -> 101
```

$$\begin{cases} out_1 \times (out_1 - 1) = 0 \\ out_2 \times (out_2 - 1) = 0 \\ out_3 \times (out_3 - 1) = 0 \\ out_1 \cdot 2^0 + out_2 \cdot 2^1 + out_3 \cdot 2^2 = input \end{cases}$$

> 前 3 个约束：`out_#` 只能是 `0` 或 `1`

由于算术电路的丰富性，输入均为有限域 F 上的数字，将其转换为二进制表示，在很多方面(比如比较大小)都有很重要的作用

与传统思路不同地方在于，将数字转化为二进制的过程，实际上是利用提示 (hint) 技术对已经转化好的数字做约束验证的过程。

Num2Bits

- Parameters: `nBits`
- Input signal(s): `in`
- Output signal(s): `b[nBits]`

The output signals should be an array of bits of length `nBits` equivalent to the binary representation of `in`. `b[0]` is the least significant bit. (`b[0]` 是最低有效位)

Solution :

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1=0;  // 累加器, make sure 最终的二进制转换结果的整数值与 n 相等

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1; // 将 `in` 右移 `i` 位, 然后与 1 进行按位与操作
        out[i] * (out[i] -1 ) === 0; // 约束值为 0 or 1
        lc1 += out[i] * e2;  // 累加器累加结果
        e2 = e2+e2;  // e2 * 2: 其值: 1,2,4,8,16 ....
    }

    lc1 === in;
}
```

```
template Num2Bits_strict() {
    signal input in;
    signal output out[254];

    component aliasCheck = AliasCheck();
    component n2b = Num2Bits(254);
    in ==> n2b.in;

    for (var i=0; i<254; i++) {
        n2b.out[i] ==> out[i];
        n2b.out[i] ==> aliasCheck.in[i];
    }
}
```

相等 (IsEqual)

比较 2 个数是否相等，只需要看相减之后结果是否为 **0**（复用 `IsZero`）

Specification: If `in[0]` is equal to `in[1]`, `out` should be `1`. Otherwise, `out` should be `0`.

- Input signal(s): `in[2]`
- Output signal(s): `out`

```
template IsEqual() {
    signal input in[2];
    signal output out;

    component isz = IsZero();

    isz.in <== in[1] - in[0];

    out <== isz.out;
```

```
}

component main { public [ in ] } = IsEqual();

/* INPUT = {
    "in": [2,1]
} */
```

比较 (LessThan)

将

```
let y = s1 > s2 ? 1 : 0 ;
```

转化为电路：

```
1. y  = s1 + 2^n - s2
2. y 二进制化取高位
```

1. 比较大小的朴素想法是将 2 个数字相减, 将结果二进制化后, 根据符号位进行判断, 但, 由于数字是群元素, 没有负数, 所以要将数字 ➕ 最大值
2. 二进制并取最高位：例如输入 s_1 / s_2 分别是 $3/4$:

$$n = 3, \;\; y = 3 + 2^3 - 4 = 7$$

7 转为二进制: $(7)_{10} = (0111)_2$, 最高位是 0 , 所以 3 < 4

同理, 例如输入 s_1 / s_2 分别是 $3/2$:

$$n = 3, \;\; y = 3 + 2^3 - 2 = 9$$

9 转为二进制: $(7)_{10} = (1001)_2$, 最高位是 1 , 所以 3 > 2

LessThan

- Input signal(s): `in[2]` . Assume that it is known ahead of time that these are at most 2252−12252−1.
- Output signal(s): `out`

Specification: If `in[0]` is strictly less than `in[1]` , `out` should be `1` . Otherwise, `out` should be `0` (如果 `in[0]` 严格小于 `in[1]` ， `out` 应该是 `1` ，否则为 `0`)

```
template LessThan(n) {
    assert(n <= 252); // 需要 n <= 252
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(n+1);

    n2b.in <== in[0]+ (1<<n) - in[1];

    out <== 1-n2b.out[n];
}
```

选择 (Selector)

将 `let y = s ? (a+b) : (a * b)` 转化为电路语言：

```
1. s · (1-s) = 0
2. y = s · (a+b) + (1-s) · (a·b)
```

1. 由于算术电路的丰富性, 需对 s 进行约束检查
2. 利用一个二进制位 s , 作为计算有效性的选择开关

Selector Circom :

- 参数： `nChoices`
- 输入信号： `in[nChoices]` , `index`
- 输出： `out`

要求：输出 `out` 应该等于 `in[index]` 。 如果 `index` 越界（不在 `[0, nChoices)` 中）， `out` 应该是 `0` 。

```
include "CalculateTotal.circom"

template QuinSelector(choices) {
    signal input in[choices];
    signal input index;
    signal output out;

    // Ensure that index < choices
    component lessThan = LessThan(4);
    lessThan.in[0] <== index;
    lessThan.in[1] <== choices;
    lessThan.out === 1;

    component calcTotal = CalculateTotal(choices);
    component eqs[choices];

    // For each item, check whether its index equals the input index.
    for (var i = 0; i < choices; i ++) {
        eqs[i] = IsEqual();
        eqs[i].in[0] <== i;
        eqs[i].in[1] <== index;

        // eqs[i].out is 1 if the index matches. As such, at most one input to
        // calcTotal is not 0.
        calcTotal.in[i] <== eqs[i].out * in[i];
    }

    // Returns 0 + 0 + 0 + item
    out <== calcTotal.out;
}
```

循环

由于算术电路的固定性, 电路只能设计为**支持最大输入数量,** 根据实际输入数量的不同, 利用选择器技术将部分计算功能关闭, 以达到不同数量的循环功能

> 不是说输入少, 计算就快, 因为电路是固定的

```
for(let i=0; i<=N; i++){
    y += 1;
}
```

1. $s = \begin{cases} 1, & i < n \\ 0, & else \end{cases}$
2. $y = s \cdot (y+1) + (1-s) \cdot (y)$

当 $i < n$ 时, $s == 1$; $y = y + 1$
当 不满足循环条件时, $s == 0$, $y == y$

1. 利用**比较**方法, 为临时变量 s 赋值
2. 利用选择方法, 分别启用循环中的计算; 或恒等原值, 即未启用

交换

```
if(s){
    o1 = i2;
    o2 = i1;
} else {
    o1 = i1;
    o2 = i2;
}
```

$$\begin{cases} o_1 = (i_2 - i_1) \times s + i_1 \\ o_2 = (i_1 - i_2) \times s + i_2 \end{cases}$$

通过一个交换标识 $s \in \{0, 1\}$ 来标记是否要交换 2 个输入

- $s == 1$: 交换
- $s == 0$: 不交换(维持原样)

逻辑 与或非

```
let y = a & b;
let y = !a;
let y = a | b;
let y = a ^ b; // XOR 异或
```

```
// 前提: 二进制约束: a·(1-a) === 0
y = a · b;
y = 1 - a;
y = 1 - (1-a)·(1-b);
y = (a + b) - 2·a·b;
```

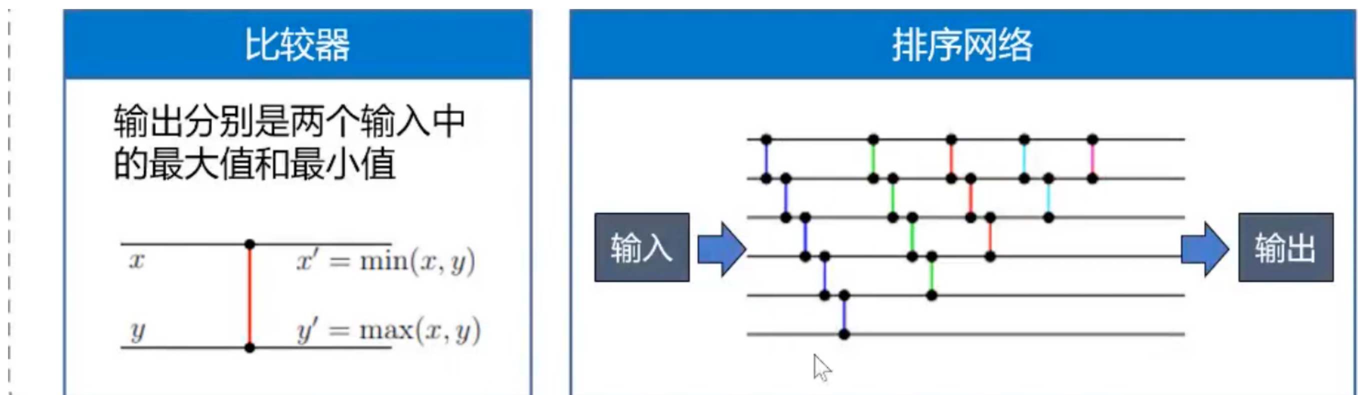
1. 逻辑运算可以通过简单的数学运算获得；
2. 二进制约束：$a \cdot (1 - a) === 0$

排序

```
// 冒泡排序
for(let i=0; i < len(arr); i++){
    for(let j=0; j < len(arr) - i - 1; i++){
        if(arr[j]> arr[j+1]){
            swap(arr[j], arr[j+1])
                }
    }
}
```

在电路中实现的方法：

1. 在算术电路上排序, 可以借用排序网络的概念
2. 利用多个比较器形成排序网络进行排序



Exercise

代码修改：如下代码存在问题：

```
// f(x) = (x1 + x2) / x3 - x4
y1 <== x1 + x2;
y2 <== y1 / x3;
out <== y2 - x4;
```

修改 ✅

```
// f(x) = (x1 + x2) / x3 - x4
y1 <== x1 + x2;
y2 <-- y1 / x3;  // 赋值, 但不约束
y1 === y2 * x3;  // 约束 : 将除法约束 转化为 乘法约束
out <== y2 - x4;
```

Tips

5 的 inverse 是不是 1/5 呀? 为什么是这么大的数字呢? 8755297148735710088898562298102910035419345760166413737479281674630323

> 因为是有限域里的 `-1` 次方

---

tau 的次数和电路约束的次数有关, 电路的约束越多, 需要越高次数的 tau 来和它匹配

> **Hash 函数特别耗费约束**

Ref :

- https://medium.com/coinmonks/hands-on-your-first-zk-application-70fe3a0c0d82
- https://zhuanlan.zhihu.com/p/143519030
- https://zhuanlan.zhihu.com/p/556765252
- https://www.youtube.com/watch?v=CTJ1JkYLiyw # ZK Shanghai Workshop Lesson 2