二、了解 circom

1. 项目文件夹中给出了 TUTORIAL.md 文件，该文件为本项目教程。然后，阅读circuits/example.circom 中的电路示例。并回答 artifacts/writeup.md 中的相关问题。

> artifacts/writeup.md

Name: []

Question 1

在下面 `Num2Bits` 代码片段中，看起来 `sum_of_bits` 可能是一个信号的乘积和，使得随后的约束不是一级的。说明为什么 `sum_of_bits` 实际上是一个信号的**线性组合**。

```
        sum_of_bits += (2 ** i) * bits[i];
```

Answer 1

`sum_of_bits` 是一个信号的线性组合，因为和里的每一项都由一个定值（二的幂，`2 ** i`）与一个信号（`bits[i]`）乘等构成。在电路限制中，线性组合指的是每项都由定值与信号相乘后相加构成。由于 `2 ** i` 是在编译时定义的幂值，不依赖于信号的值，因此这个和满足线性组合的要求。它没有包含两个信号乘等，故而不是非线性的。

Question 2

用你自己的言语说明 `<==` 这个运算符的意义。

Answer 2

`<==` 运算符在 `circom` 中用于定义两个信号或表达式之间的相等约束。它确保表达式左边和右边在执行时的计算结果相等。这个运算符不会为左边的信号赋值；而是创建一个约束，需要在电路验证中满足才能被认为有效。

这与 `<--` 运算符有所不同，`<--` 用于为信号赋值，但不创建任何约束。

Question 3

假设你正在阅读一个 `circom` 程序，你看到下列内容：

```
    signal input a;
    signal input b;
    signal input c;
    (a & 1) * b === c;
```

说明为什么这段代码是无效的。

Answer 3

这段代码是无效的，因为表达式 `(a & 1)` 包含了一个低位操作（`&`），这在 `circom` 中不是直接支持的。 `circom` 中的限制表达式需要用有限体量上的多项式表达，而低位操作如 `&` 无法转换为这类多项式。

此外，`(a & 1)` 的结果需要被表示为一个信号，或者通过合法的电路逻辑进行计算。在没有这样的表示情况下，此代码将无法编译或正常执行。

如果需要实现该功能，必须显心设计一个电路，将低位操作转化为其他限制的集成部分。

2. 根据你对 circom 和 snarkjs 的理解，使用 SmallOddFactorization 电路为7×17×19=2261 创建一个证明。并将验证密钥（verifier key）保存到 artifacts/verifier_key_factor.json 中，将证明保存到 artifacts/proof_factor.json 中。

- 编译 circom 电路，将其生成为 json 文件；

> example.circom

```
// Finally, we set the `main` circuit for this file, which is the circuit that
// `circom` will synthesize.
component main = SmallOddFactorization(3, 8);
```

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$ circom example.circom -o example.json
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$
```

- 创建可信设置，得到 verification_key.json 文件与 proving_key.json 文件。

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs setup -c example.json
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$
```

设置过程将生成两个文件： proving_key.json 和 verification_key.json 。

- 创建 input.json 文件：

```
{
    "product": 2261,
    "factors": [7,17,19]
}
```

- 计算 witness：

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs calculatewitness -c example.json
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$
```

- 生成 witness 后可以创建证明：

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs proof
```

此命令会默认使用 proving_key.json 和 witness.json 文件生成 proof.json 和 public.json 。

proof.json 文件包含实际的证明。

public.json 文件包含公开输入和输出的值。

- 验证证明

正在验证我们知道一个 witness， 其中公共输入和输出与 public.json 文件中的输入相匹配。如果证明有效，输出 OK ，如果无效，则为 INVALID 。

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/circuits$ snarkjs verify
OK
```

> 按照要求将验证密钥（ verifier key ）保存到artifacts/verifier_key_factor.json 中，将证明保存到 artifacts/proof_factor.json 中。

五、赎回证明

生成input.json文件

```
node ../../src/compute_spend_inputs.js -o input.json 10
../compute_spend_inputs/transcript3.txt 10137284576094
```

和之前证明命令类似

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/test/circuits$ snarkjs proof
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6/test/circuits$ snarkjs verify
OK
```

验证密钥（ verifier key ）保存到了 artifacts/verifier_key_spend.json 中，证明保存到了 artifacts/proof_spend.json 中。

六、测试

```
lijialu@lijialu-virtual-machine:~/Ex6/Ex6_包含全部依赖库/Ex6$ npm test

> cs251-cash@0.1.0 test
> mocha -s 1s -t 5s test/if_then_else.js test/selective_switch.js test/compute_spend_inputs.js test/spend.js



  IfThenElse
    ✓ should give `false_value` when `condition` = 0
    ✓ should give `true_value` when `condition` = 1
    ✓ should enforce that s in {0, 1}

  SelectiveSwitch
    ✓ should not switch when s = 0
    ✓ should switch when s = 1
    ✓ should enforce that s in {0, 1}

  computeInput
    ✓ transcript0.txt, depth 0, nullifier 1
    ✓ transcript1.txt, depth 4, nullifier 4
    ✓ transcript2.txt, depth 25, nullifier 7

  Spend
    ✓ witness computable for depth 0
    ✓ witness computable for depth 1
    ✓ witness computable for depth 2 (4021ms)
    ✓ witness not computable for bad input (2546ms)


  13 passing (7s)
```