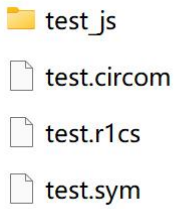
Project3： 用 circom 实现 poseidon2 哈希算法的电路
1） poseidon2 哈希算法参数参考参考文档 1 的 Table1，用(n, t, d)=(256, 3, 5)或(256, 2, 5)
2）电路的公开输入用 poseidon2 哈希值，隐私输入为哈希原象，哈希算法的输入只考虑一个 block 即可。
3） 用 Groth16 算法生成证明

一. Circom 基础

编写一个简单的乘法电路 安装完所需工具后尝试编译

```
PS D:\sdusummer\project3\test> circom2 test.circom --r1cs --wasm --sym
template instances: 1
non-linear constraints: 1
linear constraints: 0
public inputs: 0
private inputs: 2
public outputs: 1
wires: 4
labels: 4
Written successfully: ./test.r1cs
Written successfully: ./test.sym
Written successfully: ./test_js/test.wasm
Everything went okay
```

成功由 circom 文件得到电路约束文、WASM 计算模块和调试符号



二. 编写文件

实验目标
目标是证明可以找到一个输入使得其 Poseidon2 哈希等于某个公开值
在电路上表现为：验证 Hash(preimage) == public_hash，但不泄露 preimage

编写思路
路构建依赖于单向哈希函数的基本特性。Poseidon2 作为专为零知识证明优化的哈希函数，具备严格的单向性和抗碰撞性，这确保了从公开的哈希值无法反推原始输入，也无法伪造不同的输入产生相同哈希值。电路通过将哈希计算过程转化为算术电路，将"已知 preimage 使得 Hash(preimage)=public_hash"这一知识陈述，编码为一组可验证的数学约束。
其次，电路编译阶段使用 R1CS（Rank-1 约束系统）将计算过程表示为线性代数关系。每个逻辑门操作都被转换为约束方程，其中状态向量 z 包含所有输入、输出和中间变量。这种表示方法既保留了计算过程的完整性，又为后续的零知识证明提供了标准化接口。
在零知识性保障方面，系统采用 Groth16/PLONK 等非交互式证明协议。这些协议通过多项式承诺和双线性配对技术，使得验证者仅能验证约束系统的满足性，而无法获取关于隐私输入

preimage 的任何信息。电路中的隐私输入被明确声明为 private 信号，确保其始终处于加密保护状态。

编译过程

由于写入权限限制，将文件复制到临时文件夹并编译

```
PS D:\zkp_clean_test_20250808182115> npx circom2 poseidon2.circom --r1cs --wasm --output . --verbose
template instances: 71
non-linear constraints: 216
linear constraints: 199
public inputs: 0
private inputs: 1
public outputs: 1
wires: 417
labels: 583
Written successfully: poseidon2.r1cs
Written successfully: poseidon2_js/poseidon2.wasm
Everything went okay
```

编译得到 wtns 文件

```
PS D:\zkp_clean_test_20250808182115> node .\poseidon2_js\generate_witness.js .\poseidon2_js\poseidon2.wasm .\input.jso
.\witness.wtns 2>&1 | Tee-Object -Variable output
PS D:\zkp_clean_test_20250808182115>
PS D:\zkp_clean_test_20250808182115> # 检查结果
PS D:\zkp_clean_test_20250808182115> if (Test-Path .\witness.wtns) {
>>     $size = (Get-Item .\witness.wtns).Length
>>     Write-Host "✅ 生成成功！文件大小：$size 字节" -ForegroundColor Green
>> } else {
>>     Write-Host "❌ 生成失败，错误信息：" -ForegroundColor Red
>>     $output
>> }
✅ 生成成功！文件大小：13420 字节
```

三．用 Groth16 协议进行零知识证明的验证

Groth16 是一种高效、简洁的零知识证明协议，主要用于单个电路的证明生成和验证。每个电路都需要单独生成 .zkey 文件：如果修改电路，必须重新进行可信设置。

证明体积极小的证明体积：证明仅包含 3 个椭圆曲线点（A、B、C），验证速度快。

基于配对的验证：验证时只需 1 次配对运算，计算量低。

```
PS D:\zkp_clean_test_20250808182115> snarkjs groth16 setup .\poseidon2.r1cs .\pot10_final.ptau .\poseidon2_groth.zkey
[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Reading tauG1
[INFO]  snarkJS: Reading tauG2
[INFO]  snarkJS: Reading alphatauG1
[INFO]  snarkJS: Reading betatauG1
[INFO]  snarkJS: Circuit hash:
                9261e963 aa29d123 90a50c2c 0f9c46e7
                ef3a296d 587c8506 2b8d7546 d8c8525b
                5c214e5b 99757e76 b62e714f 9017fe47
                04d8398f 37eefde9 1cc6d04d 5827b613
PS D:\zkp_clean_test_20250808182115> snarkjs zkey export verificationkey .\poseidon2_groth.zkey .\groth16_verification_k
ey.json
[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: groth16
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
PS D:\zkp_clean_test_20250808182115> snarkjs groth16 prove .\poseidon2_groth.zkey .\witness.wtns .\groth16_proof.json .\
groth16_public.json
PS D:\zkp_clean_test_20250808182115> snarkjs groth16 verify .\groth16_verification_key.json .\groth16_public.json .\grot
h16_proof.json
[INFO]  snarkJS: OK!
```

四．用 Plonk 协议进行零知识证明的验证

PLONK 是一种通用、灵活的零知识证明协议，适用于多个电路共享可信设置。
通用可信设置:只需一次全局可信设置，适用于所有电路，修改电路时无需重新设置。

多项式承诺：使用 KZG 承诺（基于椭圆曲线配对）或 FRI（基于哈希）来压缩证明。

证明体积稍大，比 Groth16 稍大，但验证速度仍然很快。

生成了一个 10 阶的 ptau 文件

```
PS D:\zkp_clean_test_20250808182115> snarkjs powersoftau contribute pot10_0000.ptau pot10_0001.ptau --name="First Contri
butor" -v
Enter a random text. (Entropy): 1357924680
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: processing: tauG1: 0/2047
[DEBUG] snarkJS: processing: tauG2: 0/1024
[DEBUG] snarkJS: processing: alphaTauG1: 0/1024
[DEBUG] snarkJS: processing: betaTauG1: 0/1024
[DEBUG] snarkJS: processing: betaTauG2: 0/1
[INFO]  snarkJS: Contribution Response Hash imported:
                 19ab64cb aaaa5811 0a876138 dd1b41ee
                 b937b8a2 fb0a0037 41eb5577 a548ca12
                 592ed053 9c3d45c2 8ed61948 5ba2331d
                 a7040df7 52d59704 c3e595e2 e85fa627
[INFO]  snarkJS: Next Challenge Hash:
                 007a533d 11925a2a b692127e f560d184
                 2c4dfa1c 51889b82 154eeaa2 942c4b6f
                 838a3282 71b06dd2 5d9f98eb 0ef8669d
                 a7fd432f cbec60cf a6c66166 18e14f38
```

进行验证

```
PS D:\zkp_clean_test_20250808182115> snarkjs plonk prove .\poseidon2_plonk.zkey .\witness.wtns .\proof.json .\public.jso
n
PS D:\zkp_clean_test_20250808182115> snarkjs zkey export verificationkey .\poseidon2_plonk.zkey .\plonk_verification_key
.json
[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: plonk
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
PS D:\zkp_clean_test_20250808182115> snarkjs plonk verify .\plonk_verification_key.json .\public.json .\proof.json
[INFO]  snarkJS: PLONK VERIFIER STARTED
[INFO]  snarkJS: OK!
```