# Poseidon2 哈希算法电路实现

### 1.2 项目目标

本项目旨在使用 circom 语言实现 Poseidon2 哈希算法的电路，具体目标如下：

1. 基于参数 (n,t,d)=(256,3,5) 实现 Poseidon2 哈希算法，其中 n 为输出哈希值长度，t 为状态大小，d 为 S 盒的次数。
2. 电路设计需满足：公开输入为 Poseidon2 哈希值，隐私输入为哈希原象，且仅处理单个 block 的输入。
3. 利用 Groth16 算法生成零知识证明，实现对哈希原象与哈希值关系的验证。

## 二、Poseidon2 哈希算法原理

### 2.1 算法概述

Poseidon2 是一种基于 SPN 结构的迭代型哈希算法，其核心思想是通过多轮的替代（Substitution）和置换（Permutation）操作，实现对输入数据的混淆和扩散，最终生成哈希值。算法的主要特点包括：

* 采用非平衡的 SPN 结构，在保证安全性的同时减少电路约束数量。
* 结合完全轮（Full Rounds）和部分轮（Partial Rounds）操作，平衡安全性和效率。
* 使用幂函数作为 S 盒，在有限域上实现高效的非线性变换。

### 2.2 数学基础

#### 2.2.1 有限域运算

Poseidon2 算法在有限域GF(p)上运行，本实现采用 BN254 椭圆曲线对应的有限域，其素数p为：

p = 21888242871839275222246405745257275088548364400416034343698204186575808495617

在有限域上的基本运算包括：

1. 加法：对于
2. 乘法：对于
3. 幂运算：对于)和整数

#### 2.2.2 状态表示

Poseidon2 算法的状态用一个长度为t的向量表示，即。对于参数t=3，状态向量为，其中：

* 为率元素，用于接收输入数据。
* 为容量元素，用于维持算法的安全性。

### 2.3 算法流程

Poseidon2 哈希算法的流程包括初始化、迭代轮操作和输出三个阶段，具体如下：

#### 2.3.1 初始化

对于输入数据（其中r = t - 1为率），初始化状态向量：

#### 2.3.2 迭代轮操作

迭代轮操作由完全轮和部分轮组成，对于参数 (n,t,d)=(256,3,5)，共包含 8 轮完全轮（前 4 轮和后 4 轮）和 4 轮部分轮。每一轮操作包括以下三个步骤：

1. **添加轮常量**：

对状态向量中的每个元素添加对应的轮常量RC，即：

其中表示第round轮的第i个轮常量。

1. **S 盒替换**：

* 在完全轮中，对状态向量中的所有元素应用 S 盒变换：
* 在部分轮中，仅对状态向量的第一个元素应用 S 盒变换：其余元素保持不变。

1. **线性混合**：

通过一个的线性变换矩阵M对状态向量进行混合，即：

其中矩阵乘法为有限域上的运算。

#### 2.3.3 输出

迭代轮操作完成后，取状态向量中的容量元素作为哈希值，即。

## 三、电路设计与实现

### 3.1 电路总体结构

本项目设计的 Poseidon2 哈希电路由以下几个主要模块组成：

1. 基础运算模块：实现有限域上的加法、乘法和幂运算。
2. 轮操作模块：实现 AddRoundConstant、SubWords 和 MixLayer 操作。
3. 哈希函数模块：组合轮操作，实现完整的 Poseidon2 哈希计算。
4. 主电路模块：定义隐私输入、公开输入，并验证哈希计算结果与预期哈希值的一致性。

### 3.2 模块详细设计

#### 3.2.1 基础运算模块

1. **模 p 加法**：

|  |
| --- |
| template AddModP() {  signal input a;  signal input b;  signal output out;    out <== (a + b) % p;  } |

功能：实现有限域上的加法运算，输出a + b \mod p。

1. **模 p 乘法**：

|  |
| --- |
| template MulModP() {  signal input a;  signal input b;  signal output out;    out <== (a \* b) % p;  } |

功能：实现有限域上的乘法运算，输出a \times b \mod p。

1. **模 p 幂运算**：

|  |
| --- |
| template Pow5ModP() {  signal input x;  signal output out;  signal t1, t2;    t1 <== (x \* x) % p; // x^2  t2 <== (t1 \* x) % p; // x^3  out <== (t2 \* t1) % p; // x^5  } |

功能：实现有限域上的 5 次幂运算，输出x^5 \mod p，用于 S 盒变换。

#### 3.2.2 轮操作模块

1. **添加轮常量**：

|  |
| --- |
| template AddRoundConstant(t) {  signal input state[t];  signal input constants[t];  signal output out[t];    for (var i = 0; i < t; i++) {  component add = AddModP();  add.a <== state[i];  add.b <== constants[i];  out[i] <== add.out;  }  } |

功能：对状态向量中的每个元素添加对应的轮常量，其中t为状态大小。

1. **S 盒替换**：

|  |
| --- |
| template SubWords(t, partialRound) {  signal input state[t];  signal output out[t];    for (var i = 0; i < t; i++) {  if (partialRound == 0 || i == 0) { // 完全轮所有元素都替换，部分轮只替换第一个元素  component sbox = Pow5ModP();  sbox.x <== state[i];  out[i] <== sbox.out;  } else {  out[i] <== state[i];  }  }  } |

功能：根据轮类型（完全轮或部分轮）对状态向量应用 S 盒变换，其中partialRound为 0 表示完全轮，为 1 表示部分轮。

1. **线性混合**：

|  |
| --- |
| template MixLayer(t) {  signal input state[t];  signal output out[t];    for (var i = 0; i < t; i++) {  signal sum;  sum <== 0;  for (var j = 0; j < t; j++) {  component mul = MulModP();  mul.a <== state[j];  mul.b <== M[i][j];  component add = AddModP();  add.a <== sum;  add.b <== mul.out;  sum <== add.out;  }  out[i] <== sum;  }  } |

功能：通过线性变换矩阵M对状态向量进行混合，实现数据的扩散。

#### 3.2.3 哈希函数模块

|  |
| --- |
| template Poseidon2Hash(t, r, fullRounds, partialRounds) {  signal input inputs[r]; // 隐私输入：哈希原象（rate个元素）  signal output hash; // 公开输出：哈希值    // 初始化状态  signal state[t];  for (var i = 0; i < r; i++) {  state[i] <== inputs[i];  }  state[r] <== 0; // capacity初始化为0    // 前半部分完全轮  for (var round = 0; round < fullRounds/2; round++) {  component arc = AddRoundConstant(t);  for (var i = 0; i < t; i++) {  arc.state[i] <== state[i];  arc.constants[i] <== RC[round][i];  }    component sw = SubWords(t, 0); // 0表示完全轮  for (var i = 0; i < t; i++) {  sw.state[i] <== arc.out[i];  }    component ml = MixLayer(t);  for (var i = 0; i < t; i++) {  ml.state[i] <== sw.out[i];  }    for (var i = 0; i < t; i++) {  state[i] <== ml.out[i];  }  }    // 部分轮  for (var round = 0; round < partialRounds; round++) {  component arc = AddRoundConstant(t);  for (var i = 0; i < t; i++) {  arc.state[i] <== state[i];  arc.constants[i] <== RC[fullRounds/2 + round][i];  }    component sw = SubWords(t, 1); // 1表示部分轮  for (var i = 0; i < t; i++) {  sw.state[i] <== arc.out[i];  }    component ml = MixLayer(t);  for (var i = 0; i < t; i++) {  ml.state[i] <== sw.out[i];  }    for (var i = 0; i < t; i++) {  state[i] <== ml.out[i];  }  }    // 后半部分完全轮  for (var round = 0; round < fullRounds/2; round++) {  component arc = AddRoundConstant(t);  for (var i = 0; i < t; i++) {  arc.state[i] <== state[i];  arc.constants[i] <== RC[fullRounds/2 + partialRounds + round][i];  }    component sw = SubWords(t, 0); // 0表示完全轮  for (var i = 0; i < t; i++) {  sw.state[i] <== arc.out[i];  }    component ml = MixLayer(t);  for (var i = 0; i < t; i++) {  ml.state[i] <== sw.out[i];  }    for (var i = 0; i < t; i++) {  state[i] <== ml.out[i];  }  }    // 输出哈希值（取capacity部分）  hash <== state[r];  } |

功能：组合轮操作，实现完整的 Poseidon2 哈希计算。其中t为状态大小，r为率，fullRounds为完全轮数，partialRounds为部分轮数。

#### 3.2.4 主电路模块

|  |
| --- |
| template Main() {  // 配置参数  constant t = 3; // 状态大小  constant r = 2; // 率 (t-1)  constant fullRounds = 8; // 完全轮数  constant partialRounds = 4; // 部分轮数    // 隐私输入：哈希原象（2个元素）  signal private input preimage[r];    // 公开输入：预期的哈希值  signal public input expectedHash;    // 计算哈希值  component poseidon = Poseidon2Hash(t, r, fullRounds, partialRounds);  for (var i = 0; i < r; i++) {  poseidon.inputs[i] <== preimage[i];  }    // 验证计算的哈希值与预期的哈希值是否一致  expectedHash === poseidon.hash;  }  // 实例化主电路  component main = Main(); |

功能：定义隐私输入（哈希原象）和公开输入（预期哈希值），调用 Poseidon2Hash 模块计算哈希值，并验证计算结果与预期哈希值的一致性。

### 3.3 参数设置

根据参考文档 1 的 Table1，本实现采用的参数如下：

* 输出哈希值长度n = 256比特
* 状态大小t = 3
* 率r = t - 1 = 2
* S 盒次数d = 5
* 完全轮数fullRounds = 8（前 4 轮和后 4 轮）
* 部分轮数partialRounds = 4

轮常量RC和混合矩阵M是 Poseidon2 算法的关键参数，本实现中使用的参数仅为示例，实际应用中需根据参考文档 1 的规范精确设置。

## 四、零知识证明生成

### 4.1 Groth16 算法概述

Groth16 是一种高效的零知识证明算法，具有证明体积小、验证速度快等优点，非常适合在区块链等资源受限的环境中应用。Groth16 算法的工作流程包括：

1. 信任设置：生成公共参考字符串（CRS），包括证明密钥和验证密钥。
2. 证明生成：根据私有输入、公共输入和证明密钥生成证明。
3. 证明验证：根据公共输入、证明和验证密钥验证证明的有效性。

### 4.2 证明生成步骤

本项目使用 snarkjs 工具实现 Groth16 算法的证明生成和验证，具体步骤如下：

1. **编译电路**：

|  |
| --- |
| circom poseidon2.circom --r1cs --wasm --sym |

功能：将 circom 电路编译为 R1CS（Rank-1 Constraint System）约束系统、WASM 模块和符号文件。

1. **生成见证（Witness）**：

|  |
| --- |
| node poseidon2\_js/generate\_witness.js poseidon2\_js/poseidon2.wasm input.json witness.wtns |

功能：根据输入文件 input.json（包含隐私输入和公开输入）生成见证，即满足 R1CS 约束的解。

1. **信任设置**：

|  |
| --- |
| snarkjs groth16 setup poseidon2.r1cs pot12\_0000.ptau poseidon2\_0000.zkey  snarkjs zkey contribute poseidon2\_0000.zkey poseidon2\_0001.zkey --name="First contribution" -v |

功能：使用 powers of tau 仪式生成初始 ZKey，然后通过贡献者参与更新 ZKey，最终生成包含证明密钥的 ZKey 文件。

1. **导出验证密钥**：

|  |
| --- |
| snarkjs zkey export verificationkey poseidon2\_0001.zkey verification\_key.json |

功能：从 ZKey 文件中导出验证密钥，用于证明验证。

1. **生成证明**：

|  |
| --- |
| snarkjs groth16 prove poseidon2</doubaocanvas> |