**低资源开销Frodo密码芯片**

**设计文档**

版本：Ver. 1.1

审核人：

批准人：

编写人：

版本记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **版本号** | **生成时间** | **主 要 修 改 记 录** | **作者** |
|  | V1.0 | 20241118 | 创建 |  |
|  | V1.1 | 20241125 | 完成报告初版 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 系统架构设计 4](#_Toc14451)

[1.1. 系统整体架构设计 4](#_Toc5555)

[1.2. 模块资源开销 4](#_Toc13872)

[2. 算法指令设计及存储规划 5](#_Toc16561)

[2.1. 指令设计 5](#_Toc24373)

[2.1.1. absorbload指令 5](#_Toc958)

[2.1.2. absorb指令 5](#_Toc10068)

[2.1.3. squeezestore指令 5](#_Toc5300)

[2.1.4. squeeze指令 6](#_Toc24861)

[2.1.5. mat指令 6](#_Toc10102)

[2.2. 算法流程分析与指令使用 7](#_Toc22041)

[2.2.1. Key Generation 7](#_Toc30682)

[2.2.2. Encapsulation 8](#_Toc12682)

[2.2.3. Decapulation 10](#_Toc28623)

[2.3. 矩阵运算流程设计 11](#_Toc23157)

[2.4. SRAM存储规划 14](#_Toc13264)

[3. 子模块原理与结构设计 16](#_Toc32458)

[3.1. Hash系数生成模块 16](#_Toc28648)

[3.1.1. 接口设计 16](#_Toc24795)

[3.1.2. 结构与功能设计 17](#_Toc20942)

[3.2. 高斯采样模块 18](#_Toc16773)

[3.2.1. 接口设计 18](#_Toc16608)

[3.2.2. 结构与功能设计 19](#_Toc21692)

[3.3. 矩阵运算PE阵列 21](#_Toc11022)

[3.3.1. 接口设计 21](#_Toc23384)

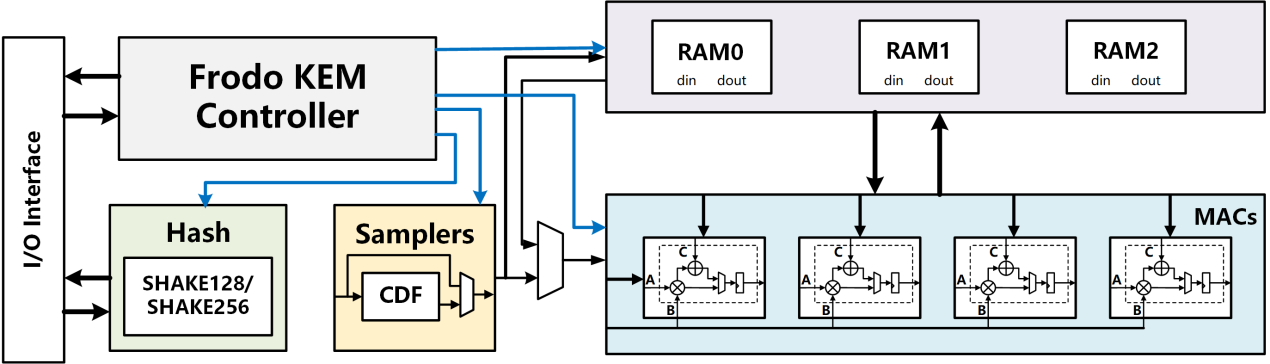
[3.3.2. 结构与功能设计 22](#_Toc13803)

[3.4. 控制模块 23](#_Toc5198)

[3.4.1. 功能设计 23](#_Toc23938)

# 系统架构设计

## 系统整体架构设计



**图1 系统整体架构图**

如图1所示，低资源开销Frodo芯片主要包括**控制模块**、**SRAM存储模块**、**PE阵列**、**Hash系数生成模块**、**采样模块**和**I/O接口**。各部分的功能描述如下：

控制模块：负责对指令进行解码，并控制计算状态转移，同时为SRAM存储模块、矩阵运算PE阵列、Hash系数生成模块、采样模块提供相应的控制信号。

SRAM存储模块：使用多块SRAM存储Frodo计算过程的中间值和结果。

矩阵运算PE阵列：用于核心矩阵乘法加法操作。

Hash系数生成模块：负责实现SHA3函数，用于生成摘要矩阵（如A）、生成用于采样的摘要、生成数据摘要作为随机数种子（如seedA）、或生成摘要等待传输（如pkh或ss）

采样模块：负责对Hash模块生成的随机数进行高斯采样，输出满足特定高斯分布特性的随机数，用于矩阵S, E, S', E', E''的生成。

## 模块资源开销

核心运算子模块的硬件结构已设计完成，下面给出各个核心子模块的硬件资源开销。

**表1 模块资源开销表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **模块名称** | **资源1** | **使用情况** | **资源2** | **使用情况** | **资源3** | **使用情况** |
| Hash系数生成模块 | FF | 14021 | LUT | 6383 | RAM |  |
| 高斯采样模块 | FF | 126 | LUT | 32 | RAM |  |
| 矩阵运算PE阵列 | FF | 1012 | LUT | 242 | RAM |  |
| SRAM存储模块 | FF |  | LUT |  | RAM | 4块  4096×32bit |

# 算法指令设计及存储规划

## 指令设计

为降低控制开销，低资源开销的Frodo采用指令进行控制。本设计中指令为27位，具体的指令设计如下。

### absorbload指令

descript

**图2 Key Gen流程**

说明：

* Addr：数据首地址
* Length：需要吸收的数据长度
* Level：安全等级
* Ex：是否执行吸收

功能描述：

用于单次吸收时，数据源来自不同的地址，或是数据量小于一轮吸收可处理最大数据量。

### absorb指令

descript

说明：

* Addr:：数据首地址
* Round：进行吸收的轮数
* Level：安全等级

功能描述：

用于数据量大于一轮吸收可处理最大数据量，需要对一批数据进行多轮吸收。

### squeezestore指令

descript

说明：

* Index：存储矩阵的索引（地址由查表获取）
* Trans：存储时是否转置
* Addr：是否更新存储地址
* Length：需要挤压的数据长度
* Level：安全等级
  + Sample：是否采样
  + EX：是否执行挤压

功能描述：

需要存储的数据量小于一次挤压输出的数据量或一次挤压的数据需要存储在不同的地址。

### squeeze指令

descript

说明：

* Index：存储矩阵的索引（地址由查表获取）
* Trans：存储时是否转置
* Addr：是否更新存储地址
* Round：挤压轮数
* Level：安全等级
* Sample：是否采样

功能描述：

进行多次挤压并存储数据

### mat指令

descript

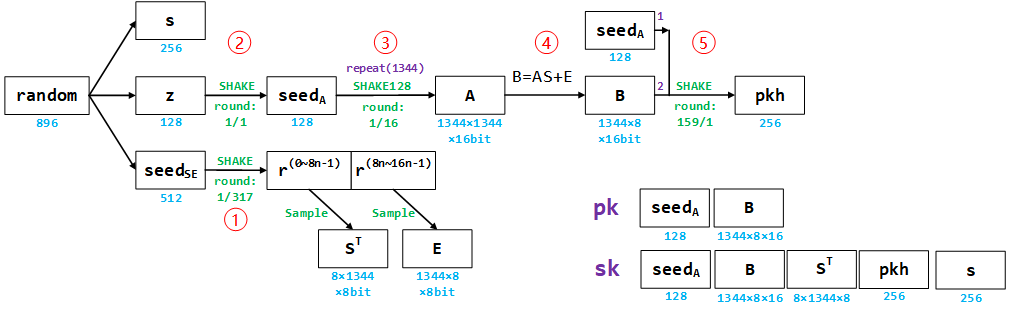
说明：

* Dst\_index：目标矩阵地址索引
* Src0\_index：源矩阵0地址索引
* Src1\_index：源矩阵1地址索引
* func：指明使用的函数
  + func=00：matmul，执行Dst = Src0 \* Src1 + Dst
  + func=01：matadd, 执行 Dst = Src0 + Dst
  + func=10：encode，执行 Dst = Encode(Src0)
  + func=11：decode，执行 Dst = Decode(Src0)

## 算法流程分析与指令使用

本设计通过对FrodoKEM算法流程的研究，设计了FrodoKEM硬件实现的总体算法流程。同时对原算法文档中的流程进行优化，有利于提高资源利用率，加快算法执行速度。

### Key Generation



**图2 Key Gen流程**

过程1需要对进行一次吸收，长度小于一轮吸收的最大长度，使用absorbload指令进行吸收。吸收完毕后，多轮挤压输出S，S需要转置存储，首次挤压时，更新控制模块内的地址。在S输出的最后一轮挤压，需要分步输出E，因此使用squeezestore指令断开，此时地址不进行更新，挤压输出S的剩余部分。E输出时，首轮不进行挤压，输出剩余的数据，且更新控制模块内的地址，之后正常多轮挤压输出，最后一轮数据长度不足使用squeezestore指令。过程1指令实现如下：

|  |
| --- |
| absorbload seed\_SE 8 Level Ex=True  squeeze S Trans=True Addr\_update=True Length Level Sample=True  squeezestore S Trans=True Addr\_update=False Length Level Sample=True Ex=True  squeezestore E Trans=False Addr\_update=True Length Level Sample=True Ex=False  squeeze E Trans=False Addr\_update=False Length Level Sample=True  squeezestore E Trans=False Addr\_update=False Length Level Sample=True Ex=True |

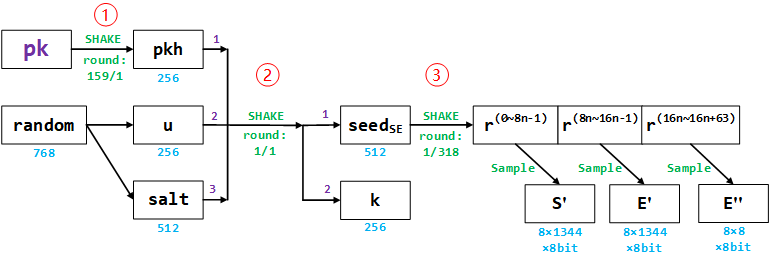
过程2较为简单，吸收和挤压均只需一轮。在本设计中，A矩阵在计算时实时生成，因此过程3、4融合为一条指令，B矩阵覆盖E矩阵的位置，因此不出现在指令中。过程2、3、4指令实现如下：

|  |
| --- |
| absorbload z 2 Level Ex=True  squeezestore seed\_A Trans=False Addr\_update=True 8 Level Sample=False Ex=True  matmul E A S |

过程5需要对和B进行拼接后再进行吸收，在本设计中只需要利用absorbload指令进行多步吸收即可，第一次吸收时仅载入数据不进行吸收。吸收完毕后，执行一轮挤压输出pkh。过程5指令实现如下：

|  |
| --- |
| absorbload seedA 2 Level Ex=False  absorbload B Length Level Ex=True  absorb B Length Level  absorbload B Length Level Ex=True  squeezestore pkh Trans=False Addr\_update=True 16 Level Sample=False Ex=True |

### Encapsulation



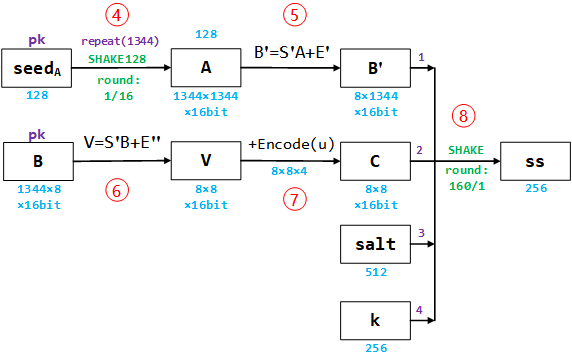
**图3 Enc流程1**

过程1与Key Generation中相同。过程2使用absorbload指令对pkh、u、salt进行分步载入吸收，再使用squeezestore指令挤压后分步输出。过程1、2指令实现如下：

|  |
| --- |
| absorb pkh Length Level  absorbload pkh Length Level  squeezestore pkh Trans=False Addr\_update=True 16 Level Sample=False Ex=True  absorbload pkh 8 Level  absorbload u 8 Level  absorbload salt 16 Level  squeezestore seed\_SE Trans=False Addr\_update=True 16 Level Sample=False Ex=True  squeezestore k Trans=False Addr\_update=True 8 Level Sample=False Ex=False |

过程3与Key Generation中S和E的生成类似，都是一轮吸收后，多轮挤压输出目标不同的数据。与之前类似，使用squeezestore指令实现挤压后分步输出，中间需要更新两次地址。过程3指令实现如下：

|  |
| --- |
| absorbload seed\_SE 8 Level Ex=True  squeeze S' Trans=False Addr\_update=True Length Level Sample=True  squeezestore S' Trans=False Addr\_update=False Length Level Sample=True Ex=True  squeezestore E' Trans=False Addr\_update=True Length Level Sample=True Ex=True  squeeze E' Trans=False Addr\_update=False Length Level Sample=True  squeezestore E' Trans=False Addr\_update=False Length Level Sample=True Ex=True  squeezestore E'' Trans=False Addr\_update=True Length Level Sample=True Ex=False |



**图4 Enc流程2**

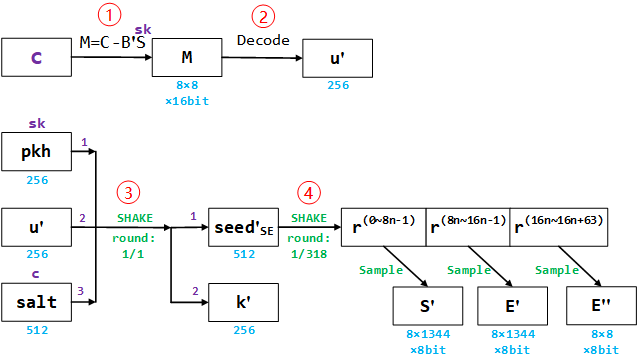
过程4、5是一个带A的矩阵乘法，因此只需要一条指令实现。过程6、7是不含A的矩阵操作，使用相应的指令实现即可。

|  |
| --- |
| matmul E' S' A  matmul E'' S' B  encode Enu u  matadd Enu V |

过程8是首先多轮吸收同一数据源B，在最后一轮吸收需要拼接B、C、salt、k，使用absorbload进行分步的载入吸收。吸收完毕经过一轮挤压输出ss。过程8指令实现如下：

|  |
| --- |
| absorb B' Round Level  absorbload B' Length Level Ex=False  absorbload C 16 Level Ex=False  absorbload salt 8 Level Ex=False  absorbload k 4 Level Ex=True  squeezestore ss Trans=False Addr\_update=True 16 Level Sample=False Ex=True |

### Decapulation



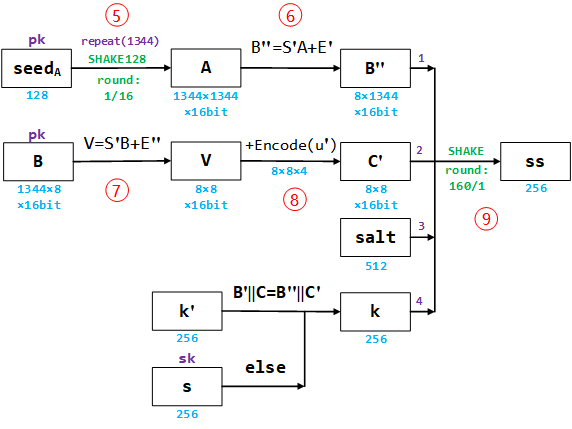
**图5 Dec流程1**

过程1是一个减法，在本设计中我们通过对乘法的一个输入取其补码实现。过程2使用decode指令对M进行decode即可。过程1、2指令实现如下：

|  |
| --- |
| matadd C -B' S  decode u' M |

过程3、4与Encapsulation相同，指令实现如下：

|  |
| --- |
| absorbload pkh 8 Level  absorbload u' 8 Level  absorbload salt 16 Level  squeezestore seed\_SE’ Trans=False Addr\_update=True 16 Level Sample=False Ex=True  squeezestore k Trans=False Addr\_update=True 8 Level Sample=False Ex=False  absorbload seed\_SE’ 8 Level Ex=True  squeeze S' Trans=False Addr\_update=True Length Level Sample=True  squeezestore S' Trans=False Addr\_update=False Length Level Sample=True Ex=True  squeezestore E' Trans=False Addr\_update=True Length Level Sample=True Ex=True  squeeze E' Trans=False Addr\_update=False Length Level Sample=True  squeezestore E' Trans=False Addr\_update=False Length Level Sample=True Ex=True  squeezestore E'' Trans=False Addr\_update=True Length Level Sample=True Ex=False |



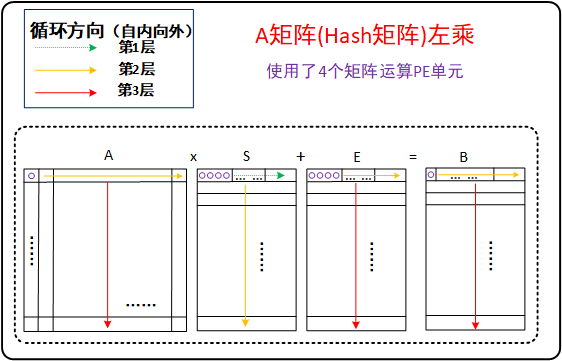
**图6 Dec流程2**

过程5、6、7、8、9与Encapsulation均重复，具体指令实现如下：

|  |
| --- |
| matmul E' S' A  matmul E'' S' B  encode Enu' u'  matadd Enu' V  absorb B' Round Level  absorbload B' Length Level Ex=False  absorbload C 16 Level Ex=False  absorbload salt 8 Level Ex=False  absorbload k 4 Level Ex=True  squeezestore ss Trans=False Addr\_update=True 16 Level Sample=False Ex=True |

## 矩阵运算流程设计

在本设计中，为节省存储空间，公共矩阵A不进行存储，而是在计算时进行实时生成，这给矩阵运算带来了难点，因为公共矩阵A必须由Hash模块按行生成，但是在封装和解封装的过程中，公共矩阵A都会作为右乘矩阵需要按列进行读取。受限于Hash模块的数量，难以使用传统的数据流进行矩阵计算，且需要额外的控制开销。考虑到在低资源开销的Frodo中，计算的并行度也有所降低，本设计最终采用了类似weight store(ws)的脉动阵列的数据流。下面针对三种矩阵运算类型，A矩阵左乘、A矩阵右乘，A矩阵不参与运算，对矩阵运算流程设计进行详细说明。

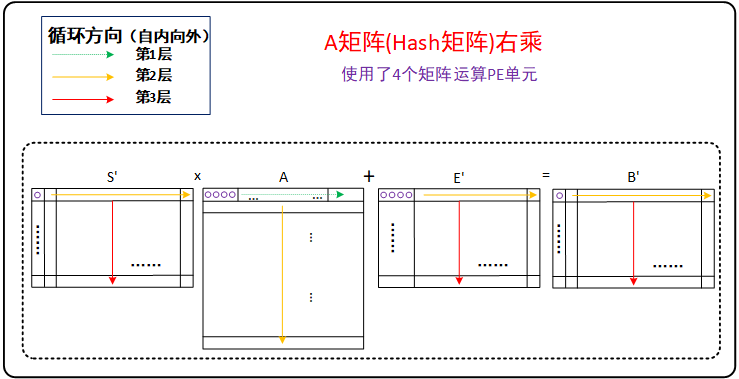
、

**图7 A左乘流程设计图**

如图所示，针对A×S+E这一A矩阵左乘运算，使用4个矩阵PE运算单元，每一个运算单元可进行针对16比特的a×b+c（a、b、c分别为A、S、E矩阵中的元素）数学运算。

A以自上至下自左至右的遍历方向，自矩阵A取出第1行第1个元素，S矩阵以自左至右方向将第1行元素逐个与第1个A矩阵元素进行乘法运算，E矩阵以自左至右方向将第1行元素逐个与乘积进行加法运算，此时结果矩阵第一行每个元素仅是其中间结果。之后自矩阵A取出第1行第2个元素，S矩阵以自左至右方向将第2行元素逐个与第一个A矩阵元素进行乘法运算，乘积逐个与对应的中间运算结果相加。重复此过程直至A的第一行遍历结束，此时结果矩阵第一行元素生成完成。对于A的每一行都采用此流程（仅在A本行第1个元素时E遍历一行），直至A矩阵遍历完毕，此时结果矩阵每一个元素生成完成。A矩阵不存储，通过Hash函数按行生成A的元素，每次生成一个，按流程运算后生成下一个。

总结而言，对于A的一行，S遍历整个矩阵一轮，对于A每一行的第一个元素，E遍历对应的一行。



**图8 A右乘流程设计图**

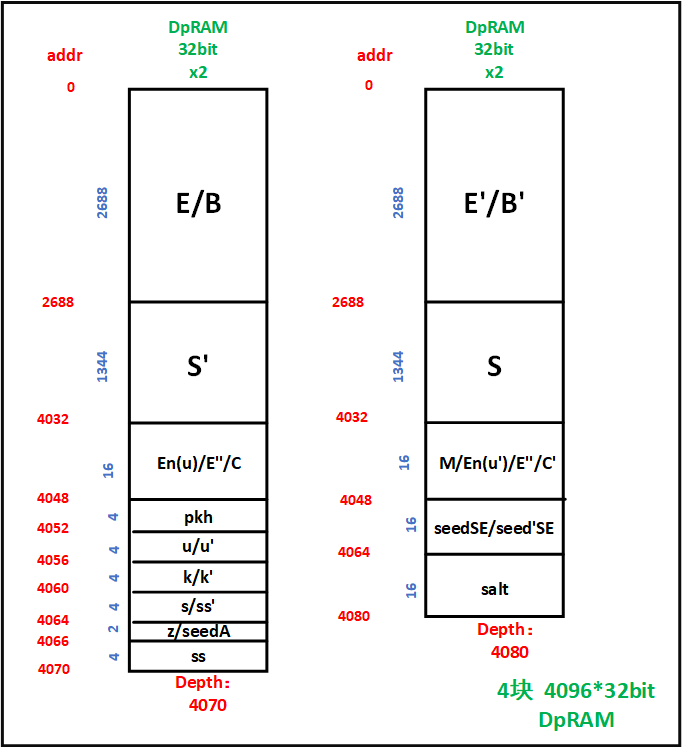
如图所示，针对S'×A+E'这一类型的A矩阵右乘运算，使用4个矩阵PE运算单元。总体运算流程与A矩阵类似，不同之处在于，左乘矩阵S’在运算前已被生成完毕并存储，右乘矩阵A需要通过Hash模块按行生成，在此类运算下，对于S'的每一个元素，都需要遍历A的对应一行，所以A的一行需要多次生成，每一行S'元素都需要整个A矩阵的一次生成。此方案保持了A左乘右乘控制的一致性，解决了A按行生成按列读取的问题。

总结而言，对于S'的一行，A需要生成一次整个矩阵，遍历一次，对于S'每一行的第一个元素，E'遍历对应的一行。

针对A不参与的矩阵运算，其参与乘法的矩阵均在运算前已被生成并存储，可以直接采用类似的运算流程，由于不需控制Hash模块，控制更加简单。

总体而言，通过对矩阵运算流程的设计，解决了A按行生成按列读取的问题，实现了采用一种运算流程控制所有矩阵运算，有利于简化控制逻辑，降低控制所需的硬件资源。

## SRAM存储规划

**图9 存储地址示意图**

在本文中，共使用了4块4096\*32bit大小的DpRAM，用于存储数据。FrodoKEM算法的硬件实现，必须存储的数据包括公钥（sk），私钥（sk），共享秘密（ss、ss‘），密文（c）。而对于矩阵运算过程中的中间数据存储，可进行优化设计，通过新数据覆盖旧数据的方法，达到不同中间数据公用同一块存储空间的效果，进而减小存储资源的使用。

多个数据复用一个RAM存储空间的详细说明：

1. 矩阵E与矩阵B，矩阵E'与矩阵B'，其矩阵维度均一致。矩阵E参与矩阵B的加法运算，其数据仅需读取一次，矩阵E'同理。设计E与B共用一块存储资源，E'与B'共用一块存储资源。
2. 矩阵En(u)、矩阵E''、矩阵C，四者矩阵维度一致，且分别为同一个矩阵运算过程下的待加矩阵与结果矩阵，设计四者共用一块存储资源。
3. 矩阵En(u')、矩阵E''、矩阵C'、矩阵M四者矩阵维度一致，且分别为同一个矩阵运算过程下的待加矩阵与结果矩阵，设计四者共用一块存储资源。
4. 比特串u与u'，二者数据大小一致，在算法中功能相似，均作为输入消息，哈希产生SeedSE或Seed'SE，设计二者共用一块存储资源。
5. 比特串SeedSE与Seed'SE，二者数据大小一致，均作为输入消息，哈希产生需采样的数据，设计二者共用一块存储资源。
6. k与k'，二者数据大小一致，在相似的运算过程中作为哈希值输出，设计二者共用一块存储资源。
7. s与ss‘，二者数据大小一致，s在KeyGen中从输出的哈希中被截取出来，之后并没有参与其他运算，设计二者共用一块存储资源，在ss‘生成后覆盖原s的存储位置。
8. z与seedA，二者数据大小一致，z作为输入消息，通过Hash过程产生seedA后不参与其他运算，设计二者共用一块存储资源，在seedA生成后覆盖原z的存储位置。

# 子模块原理与结构设计

## Hash系数生成模块

### **接口设计**

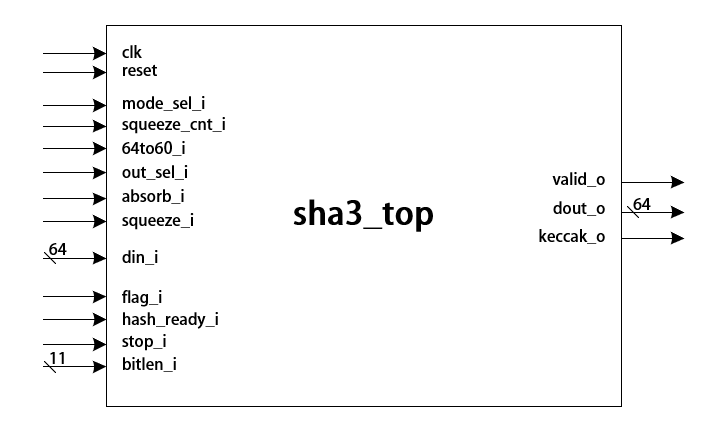
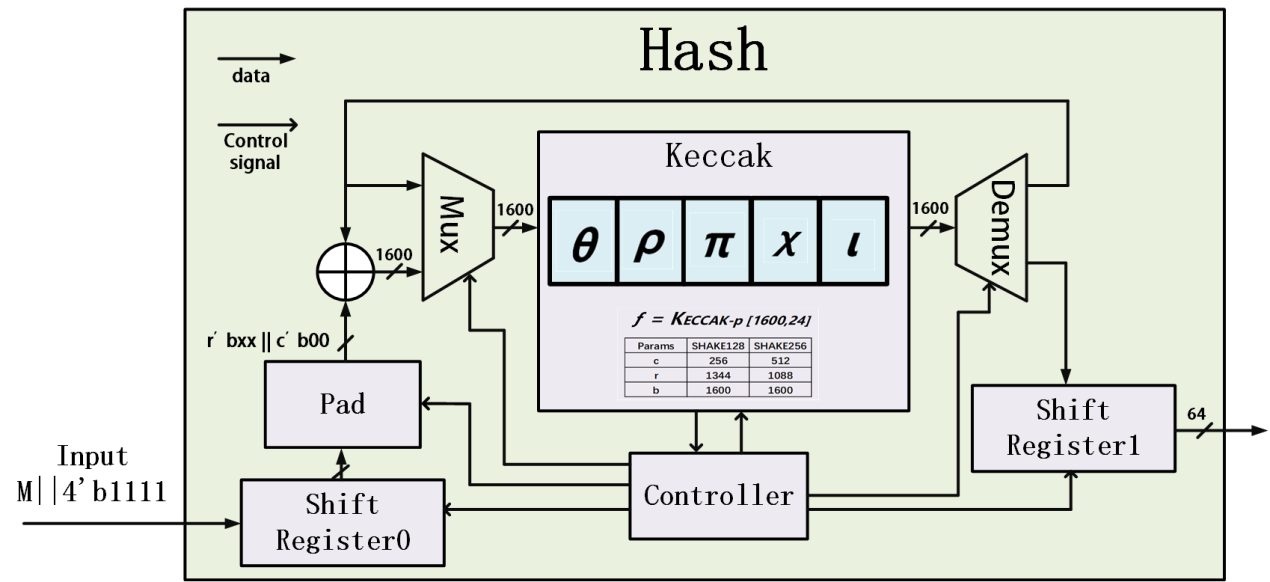


图10 Hash系数生成模块接口图

**表2 Hash模块接口定义**

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名称** | **I/O** | **位宽** | **定义** |
| clk | input | 1 | 时钟 |
| reset | input | 1 | 复位信号 |
| absorb\_i | input | 1 | 吸收使能 |
| squeeze\_i | input | 1 | 挤压使能 |
| mode\_sel\_i | input | 1 | 模式选择：SHAKE128/256 |
| bitlen\_i | input | 11 | 输入长度指示信号 |
| 64to60\_i | input | 1 | 位宽转换指示信号 |
| hash\_ready\_i | input | 1 | 输入使能 |
| din\_i | input | 64 | 输入数据 |
| flag\_i | input | 1 | 输入移位寄存器清零信号 |
| squeeze\_cnt\_i | input | 2 | 挤压轮数指示信号 |
| out\_sel\_i | input | 2 | 输出模式指示信号 |
| stop\_o | input | 1 | 输出暂停信号 |
| dout\_o | output | 64 | 输出数据 |
| keccak\_o | output | 1 | keccak函数迭代结束信号 |
| valid\_o | output | 1 | 输出有效 |

### 结构与功能设计



**图11 Hash系数生成模块硬件架构图**

在FrodoKEM算法中，可通过两种方法进行矩阵A元素的生成，一种为通过AES函数，另一种为通过SHAKE函数。由于在其他过程中SHAKE函数被多次使用（如seedA，ss的生成），本设计选择使用SHAKE函数。最核心的运算过程之一为使用Hash函数进行数据处理，本算法中选择使用的Hash函数为SHAKE128与SHAKE256。本文设计了Hash算子的硬件结构，支持Hash算法家族中的SHAKE128与SHAKE256。

如图所示，Hash模块主要两个移位寄存器，填充（Pad）模块，核心运算模块（Keccak），以及一个控制模块组成（Controller）。在Keccak模块的输入端与输出端分别连接多路选择器与解多路选择器，通过两个移位寄存器分别对输入数据和输出数据进行缓冲处理。内部结构的系统工作由一个控制模块控制。

模块同时支持SHAKE128与SHAKE256两种类型的SHA-3函数，同时提供安全等级切换信号，可实现在不同安全等级下使用对应函数。通过详细研究算法流程，可以发现在同一安全等级下Hash模块的挤压操作次数只有四种情况，以FrodoKEM-1344等级为例子，仅有1次、16次、317次、318次四种情况，基于这个特性，设计Hash函数自动执行挤压过程的功能。具体而言，提供一个Keccak挤压操作执行次数的选择信号，在外部控制进行一次挤压操作后，Hash模块将自行执行后续的多轮挤压，直至达到目标轮次，这降低了控制的复杂性，有利于整个算法控制模块的设计。针对Hash矩阵的左乘右乘两种情况，设计了Hash函数的可控数据输出功能，可一次输出4个矩阵有效数据，也可一次输出1个矩阵有效数据，并可以通过输入信号，控制数据输出的暂停与否，适配本文设计的矩阵乘法流程控制方案。在Frodo—640安全等级下，矩阵B,B',C的元素位宽规定为15bit（640等级运算在模32768下进行），但在本设计中，被以16bit形式储存，如果直接输入Hash模块进行pkh与ss生成，会由于第16位的存在产生错误，针对此问题，在Hash模块添加对应指示信号，设计Hash可在此信号有效下，将输入的64bit分为4组，每一组仅取60bit然后拼接后再次输入模块内部。

Hash模块运行中先进行吸收阶段，待输入的消息根据其长度被分为一次或数次输入进模块的移位寄存器0中，在一次输入结束后，本次输入的所有数据会整体被送进Pad模块中根据其长度进行填充并连接零串构成1600bit，与Keccak模块中经过24轮运算的结果数据进行异或操作后再度被送入Keccak模块运算，重复此过程直到所有数据被传入模块并且完成吸收。在挤压阶段，Keccak模块运算结果数据分为两路，一路送入输出移位寄存器在控制下输出，一路直接再次送入Keccak模块，直至挤压过程结束。

## 高斯采样模块

### 接口设计

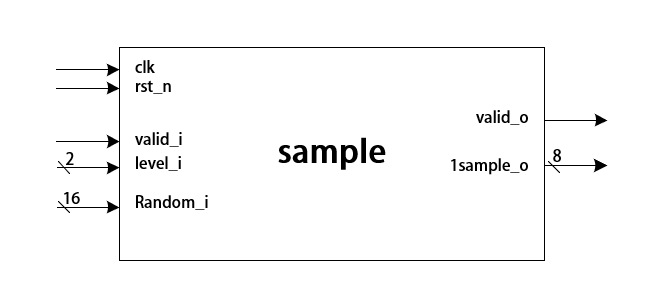


图12 高斯采样模块接口图

**表3 采样器模块接口定义**

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名称** | **I/O** | **位宽** | **定义** |
| clk | input | 1 | 时钟 |
| rst\_n | input | 1 | 下降沿复位信号 |
| valid\_i | input | 1 | 使能信号 |
| level\_i | input | 2 | 安全等级选择信号 |
| 1random\_i | input | 16 | 待采样数据(1个) |
| 1sample\_o | output | 8 | 采样结果（1个） |
| valid\_o | output | 1 | 输出有效 |

### 结构与功能设计

****

**图13 概率密度函数硬件结构**

****

**图14 高斯采样硬件结构**

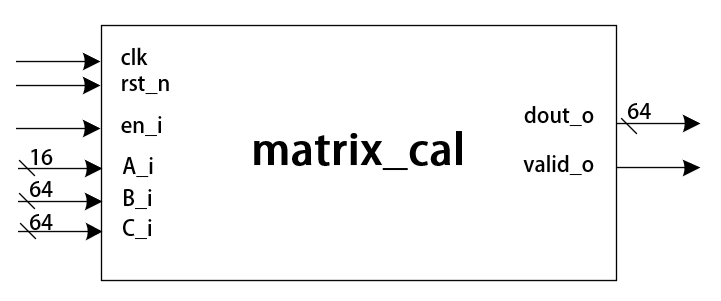
哈希函数生成的部分数据需在经过高斯采样才可后组成矩阵。针对高斯采样过程，本文设计了高斯采样器的硬件结构，实现了对数据进行高斯采样后输出的功能。

如图所示，采样器仅由1个CDF模块组成，CDF模块主要结构由数个数值比较器，加法树，符号处理模块组成。采用将误差分布表中数据提前储存在硬件单元中的方法，降低控制难度，有利于减少控制所需的硬件资源。

数据在输入采样器后，最低有效位数据会输入符号处理模块，其余有效位组成的数值在安全等级信号得控制下，查找对应的误差分布表，逐个与表中数据比较大小，小于此数值的表中数据个数即为采样结果的绝对值。绝对值进入符号处理模块后，根据最低有效位得数值进行正负符号的添加，最终得到采样数据输出。

## 矩阵运算PE阵列

### 接口设计

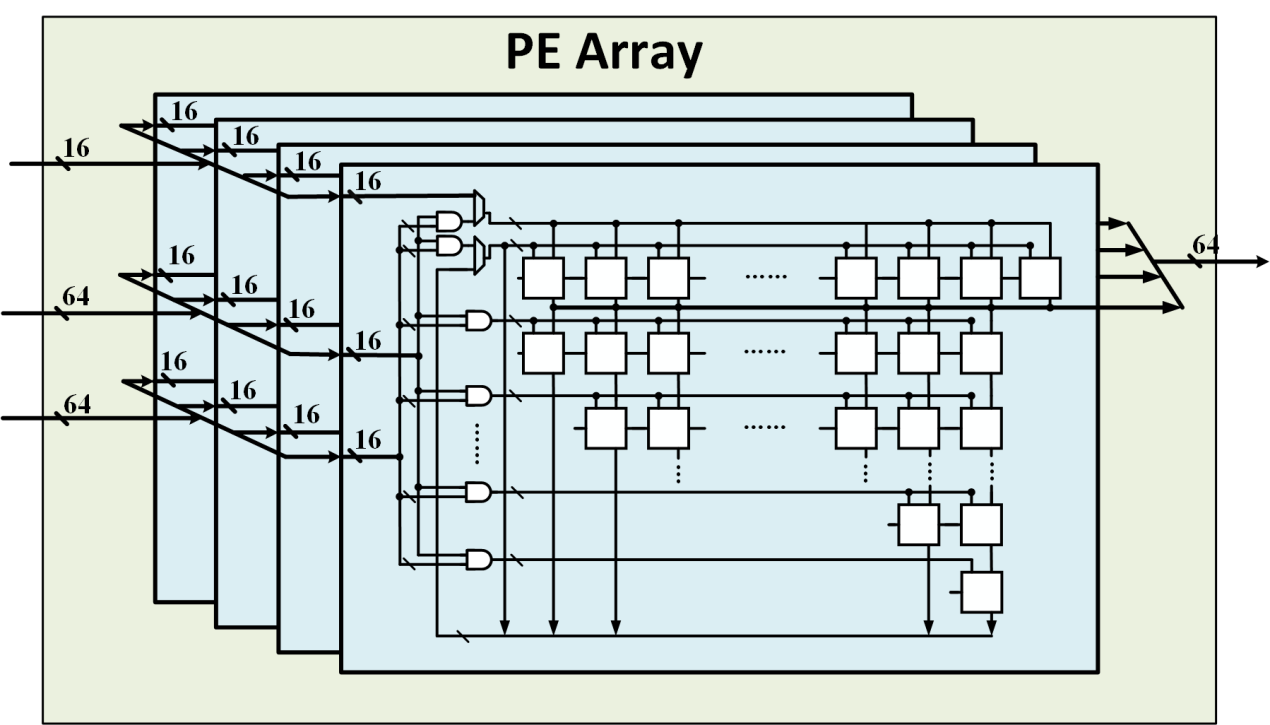


**图15 矩阵运算模块接口图**

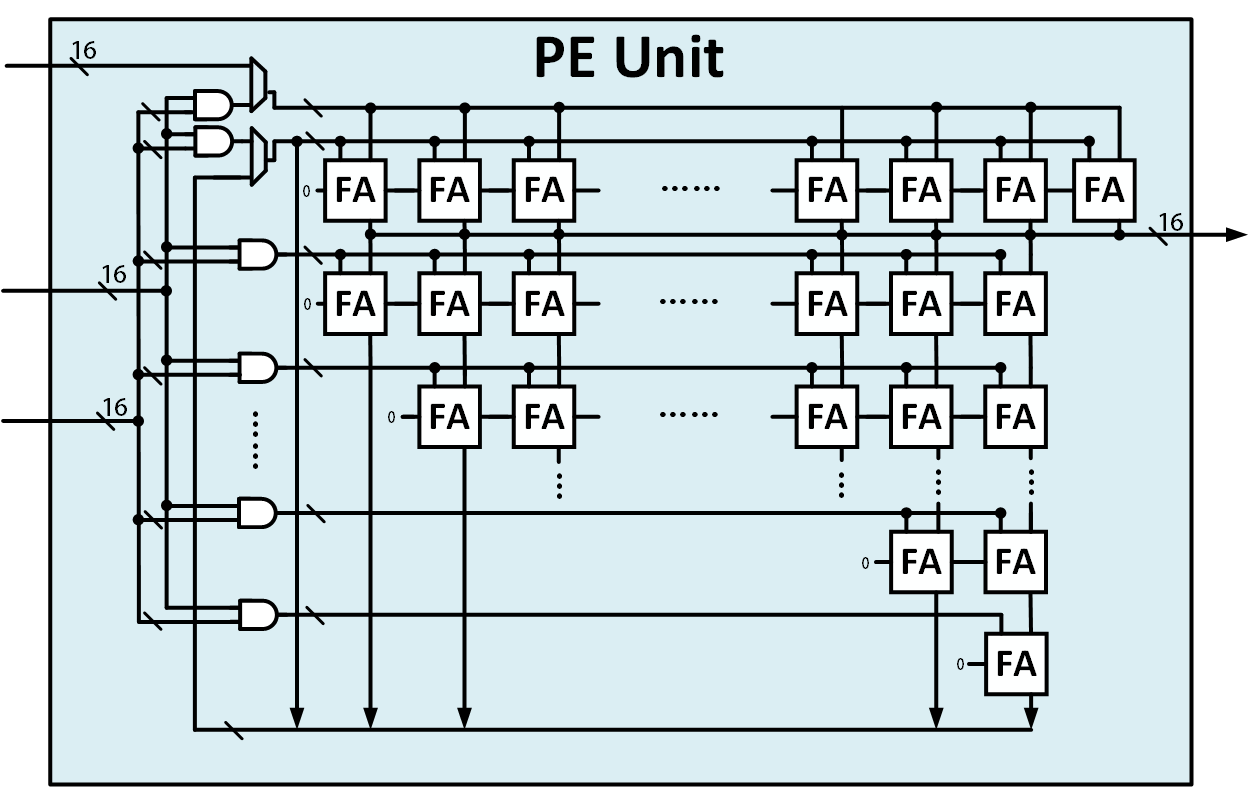
**表4 矩阵运算PE阵列接口定义**

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名称** | **I/O** | **位宽** | **定义** |
| clk | input | 1 | 时钟 |
| rst\_n | input | 1 | 复位 |
| en\_i | input | 1 | 使能 |
| A\_i | input | 16 | 左乘矩阵1个元素 |
| B\_i | input | 64 | 右乘矩阵4个元素(1个16bit) |
| C\_i | input | 64 | 待加矩阵4个元素(1个16bit) |
| dout\_o | output | 64 | 输出4个结果矩阵元素(1个16bit) |
| valid\_o | output | 1 | 输出有效 |

### 结构与功能设计



**图16 PE运算单元阵列**



**图17 全加器阵列（乘加器结构**）

在FrodoKEM算法运行过程中，需要对大维度矩阵进行运算，包括矩阵乘法与矩阵加法，需要设计专用的矩阵运算PE阵列，本设计采用根据运算数据与运算流程的特点，通过手动设计全加器（Full Adder，FA）阵列的方式，实现乘法器加法器公用结构，有利于减少硬件资源的使用。

在FrodoKE算法中，矩阵运算分为两个内容，矩阵数据模乘与矩阵数据模加，即A x B + C(A、B、C为参与运算的矩阵)。矩阵运算PE阵列由四个PE运算单元组成，每一个运算单元可均以实现对输入其的三个数据进行 （a x b + c） mod 65535 的数学运算，整个PE阵列可以并行4个PE运算单元，在控制模块的控制下，PE阵列可以完成对于FrodoKEM中矩阵的运算。

如图所示，PE阵列由4个PE运算单元组成，PE单元的本质是一个由全加器构成的阵列，根据FrodoKEM算法中的数据运算特点，在传统全加器阵列乘法器上进行优化设计。算法中的矩阵数据在运算结束后需通过Pack操作，舍弃高有效位数据，仅保留16比特。在本设计中，为降低乘法器加法器资源开销，将其中的乘法运算设计为模乘与模加运算，对应在全加器阵列中删除运算高有效位数据的全加器，构成适用于算法的模乘器，并使用模乘器中的部分全加器进行模加运算，实现了模乘模加运算公用一套硬件结构，完成在此运算单元上的低资源开销设计。

数据在输入PE运算单元后，首先两个乘数每一位经过与运算，得到多个部分和，分别被输入到整个全加器阵列中，实现模乘乘运算。之后得到的乘积与加数会再次被输入加法器阵列，使用阵列中得部分全加器阵列实现模加运算，最终输出模乘模加运算结果。

## 控制模块

### 接口设计

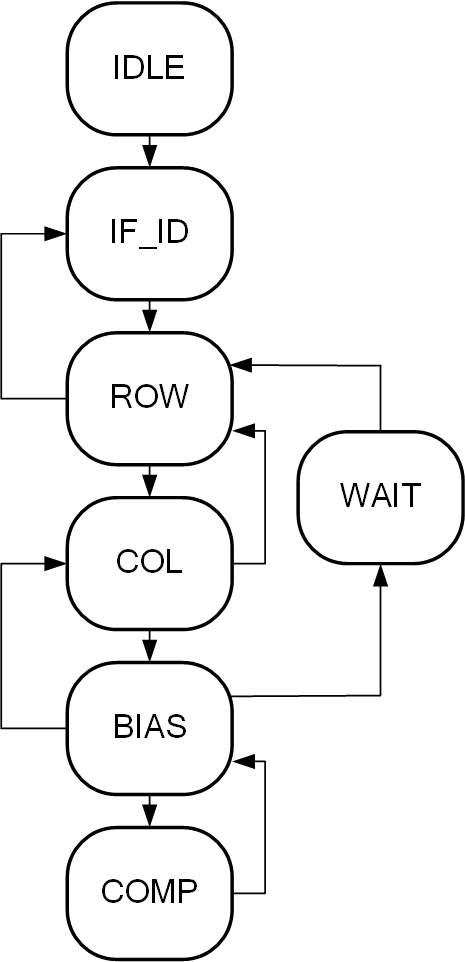
**表5 控制模块接口定义**

|  |  |  |  |
| --- | --- | --- | --- |
| **端口名称** | **I/O** | **位宽** | **定义** |
| clk | input | 1 | 时钟 |
| reset | input | 1 | 复位信号 |
| absorb\_i | output | 1 | Hash吸收使能 |
| squeeze\_i | output | 1 | Hash挤压使能 |
| mode\_sel\_i | output | 1 | 模式选择：SHAKE128/256 |
| bitlen\_i | output | 11 | 输入长度指示信号 |
| 64to60\_i | output | 1 | 位宽转换指示信号 |
| hash\_ready\_i | output | 1 | 输入使能 |
| din\_i | output | 64 | 输入数据 |
| flag\_i | output | 1 | 输入移位寄存器清零信号 |
| squeeze\_cnt\_i | output | 2 | 挤压轮数指示信号 |
| out\_sel\_i | output | 2 | 输出模式指示信号 |
| stop\_o | output | 1 | 输出暂停信号 |
| valid\_i | output | 1 | 采样使能信号 |
| level\_i | output | 2 | 安全等级选择信号 |
| en\_i | output | 1 | PE阵列使能 |
| A\_i | output | 16 | 左乘矩阵1个元素 |
| B\_i | output | 64 | 右乘矩阵4个元素(1个16bit) |
| C\_i | output | 64 | 待加矩阵4个元素(1个16bit) |
| dout\_o | input | 64 | 输出4个结果矩阵元素(1个16bit) |
| ram0\_addra | output | 13 | RAM0端口a地址 |
| ram0\_addrb | output | 13 | RAM0端口b地址 |
| ram0\_dina | input | 64 | RAM0端口a输入数据 |
| ram0\_dinb | input | 64 | RAM0端口b输入数据 |
| ram0\_douta | output | 64 | RAM0端口a输出数据 |
| ram0\_doutb | output | 64 | RAM0端口b输出数据 |
| ram0\_ena | output | 1 | RAM0端口a写使能 |
| ram0\_enb | output | 1 | RAM0端口a写使能 |
| ram1\_addra | output | 13 | RAM1端口a地址 |
| ram1\_addrb | output | 13 | RAM1端口b地址 |
| ram1\_dina | input | 64 | RAM1端口a输入数据 |
| ram1\_dinb | input | 64 | RAM1端口b输入数据 |
| ram1\_douta | output | 64 | RAM1端口a输出数据 |
| ram1\_doutb | output | 64 | RAM1端口b输出数据 |
| ram1\_ena | output | 1 | RAM1端口a写使能 |
| ram1\_enb | output | 1 | RAM1端口a写使能 |
| ram2\_addra | output | 13 | RAM2端口a地址 |
| ram2\_addrb | output | 13 | RAM2端口b地址 |
| ram2\_dina | input | 64 | RAM2端口a输入数据 |
| ram2\_dinb | input | 64 | RAM2端口b输入数据 |
| ram2\_douta | output | 64 | RAM2端口a输出数据 |
| ram2\_doutb | output | 64 | RAM2端口b输出数据 |
| ram2\_ena | output | 1 | RAM2端口a写使能 |
| ram2\_enb | output | 1 | RAM2端口a写使能 |

### 功能设计

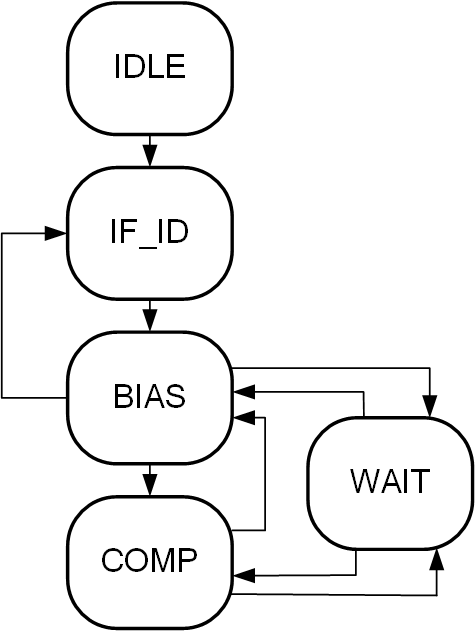
控制模块的功能为通过对输入的读取，产生其他模块的控制信号，并提供数据读写所需的RAM地址，对一些数据进行预处理。在本设计中，所有的指令都与矩阵的生成、读写、运算相关，因此控制模块使用4个计数器控制一个4层循环，ROW计数器（11位），COL计数器（9位），BIAS计数器（2位），COMP计数器（9位）。循环的边界值可由指令译码获得，根据指令不同，控制模块的状态机有下面4种模式，不同模式间状态共用。在所有的模式中，IDLE均为待机状态，IF\_ID进行取指译码。

模式0适用指令matmul和matadd,进行矩阵计算的相关操作。此时四个计数器，ROW计数器记录左乘矩阵的行号，COL计数器记录左乘矩阵的列号，因为存储器的位宽大于一次处理的左乘矩阵的数据位宽，BIAS计数器记录块内偏置量，COMP计数器记录右乘矩阵的列号。同名的各个状态进行各计数器达到边界值时对应的操作，COMP状态负责计算。当公共矩阵A作为操作矩阵时，会进入WAIT状态等待矩阵A的生成。没有A的参与时，为减少额外的控制逻辑仍保留了WAIT状态。



**图18 模式0状态转移图**

模式1适用于使用squeeze，squeezestore存储需要进行转置的矩阵。COMP计数器记录行的累加，也就算地址的累加，BIAS计数器记录块内的偏置，在每个COMP状态，控制模块会从存储器读取数据后，拼接新的数据再重新存入。与之前相同，在一次挤压的数据全部输出后会进入WAIT状态进行等待下一轮的结果。



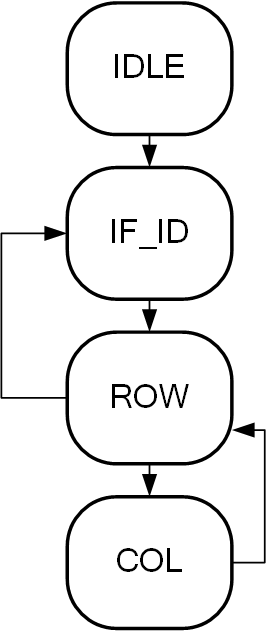
**图19 模式1状态转移图**

模式2适用于absorb，absorbload指令以及矩阵不需要偏置的squeeze和squeezestore指令。ROW计数器记录需要读取/存储矩阵的行号，COL计数器记录列号，BIAS记录块内偏置（一次输出的数据不满足存储器的位宽），WAIT阶段等待Hash模块的操作。



**图20 模式2状态转移图**

模式3适用于encode和decode指令。ROW计数器记录矩阵的行号，COL计数器记录矩阵的列号（非矩阵数据我们也会安装矩阵的格式读写），读取后进行相应操作再写回。



**图21 模式3状态转移图**