摘要

Abstract

Thesis 不是paper

目录

[摘要 2](#_Toc448392758)

[Abstract 3](#_Toc448392759)

[目录 4](#_Toc448392760)

[第一章 绪论 6](#_Toc448392761)

[1.1 研究背景 6](#_Toc448392762)

[1.1.1 可重构计算概述 6](#_Toc448392763)

[1.1.2 密码算法综述 6](#_Toc448392764)

[1.2 国内外研究现状 7](#_Toc448392765)

[1.2.1 可重构密码架构综述 7](#_Toc448392766)

[1.2.2 可重构密码PE概述 8](#_Toc448392767)

[1.3 论文研究内容及意义 10](#_Toc448392768)

[1.4 论文组织结构 12](#_Toc448392769)

[第二章 分组密码算法与密码可重构架构 13](#_Toc448392770)

[2.1 分组密码算法 13](#_Toc448392771)

[2.1.1 分组密码简介 13](#_Toc448392772)

[2.1.2 分组密码算法的数学模型 13](#_Toc448392773)

[2.1.3 分组密码的整体结构特征及代表算法 14](#_Toc448392774)

[2.2 面向分组密码算法可重构系统架构 25](#_Toc448392775)

[2.2.1 配置控制器 26](#_Toc448392776)

[2.2.2 计算引擎 27](#_Toc448392777)

[2.3 本章小结 27](#_Toc448392778)

[第三章 算法建模与算子特征提取 29](#_Toc448392779)

[3.1 算法建模 29](#_Toc448392780)

[3.2 算法的算子特征 30](#_Toc448392781)

[3.2.1 参数特征 31](#_Toc448392782)

[3.2.2 组合特征 33](#_Toc448392783)

[3.2.3 位置特征 34](#_Toc448392784)

[第四章 PE设计方案 35](#_Toc448392785)

[4.1 设计方法 35](#_Toc448392786)

[4.2 阵列拓扑结构 35](#_Toc448392787)

[4.3 行间互连 37](#_Toc448392788)

[4.4 异构组 38](#_Toc448392789)

[4.5 处理单元 40](#_Toc448392790)

[4.6 功能单元 41](#_Toc448392791)

[4.6.1 逻辑单元 42](#_Toc448392792)

[4.6.2 S盒替代单元 42](#_Toc448392793)

[4.6.3 算术单元 42](#_Toc448392794)

[4.6.4 置换单元 43](#_Toc448392795)

[4.6.5 移位单元 44](#_Toc448392796)

[4.6.6 有限域乘法单元 45](#_Toc448392797)

[4.7 本章小结 47](#_Toc448392798)

[第五章 算法映射分析与PE方案优化 48](#_Toc448392799)

[5.1 可重构架构映射概述 48](#_Toc448392800)

[5.1.1 问题模型 48](#_Toc448392801)

[5.1.2 研究现状 49](#_Toc448392802)

[5.2 基于子图同构的映射方案 50](#_Toc448392803)

[5.2.1 子图同构基本概念 50](#_Toc448392804)

[5.2.2 VF2子图同构算法 51](#_Toc448392805)

[5.2.3 架构图建模 53](#_Toc448392806)

[5.2.4 基于VF2算法的映射方案 55](#_Toc448392807)

[5.3 映射结果分析与PE方案优化 58](#_Toc448392808)

[5.3.1 算法映射结果分析 58](#_Toc448392809)

[5.3.2 基于映射结果的PE方案优化 58](#_Toc448392810)

[5.4 本章小结 58](#_Toc448392811)

[第六章 优化PE方案的验证与分析 59](#_Toc448392812)

[6.1 PE实现与性能分析 59](#_Toc448392813)

[6.2 算法映射结果 59](#_Toc448392814)

[6.3 与不同架构的对比 59](#_Toc448392815)

[6.4 本章小结 62](#_Toc448392816)

[第七章 总结与展望 63](#_Toc448392817)

[7.1 总结 63](#_Toc448392818)

[7.2 展望 63](#_Toc448392819)

1. 绪论
   1. 研究背景
      1. 可重构计算概述

随着科技水平不断发展，人们生活的需求越来越复杂，各种新型应用层出不穷。它们普遍具有运算复杂度高、处理数据量大等特点，在性能、功耗、灵活性、集成度和成本等方面对嵌入式移动终端提出了十分苛刻的要求。为了实现这些计算密集型的应用，主要采用两种方法：一种是在专用集成电路（Application Specific Integrated Circuit，ASIC）上用硬件实现，另一种是在通用处理器（General Purpose Processor，GPP）上用软件实现。ASIC具有高的能量效率、面积效率，在性能、功耗、芯片面积等方面能够进行很好的优化，但是它只能针对某一特定应用或者某一特定算法设计，灵活性低下，很难满足更多的新型应用需求，难以升级，重用性差；GPP则具有很强的灵活性，可适用于各种不同的应用领域，但指令流驱动的执行方式导致其整体性能和功耗并不理想，并且运行速度慢、效率低，难以满足应用的性能要求。

在这种背景之下，可重构计算技术应运而生。可重构计算同时兼顾了通用处理器与ASIC的优点，既保留了通用处理器的灵活性，也具有ASIC的高效性，能够比较好地满足众多复杂应用的计算需求。早在20世纪60年代，美国加州大学的Gerald Estrin就提出了可重构计算的概念**错误!未找到引用源。**，将一个通用处理器连接一个计算阵列，通用处理器负责控制计算阵列运行，计算阵列可以接收配置信息执行特定的功能，两者结合起来，通过通用处理器对计算阵列的配置信息进行管理，就可以让计算阵列执行不同的任务，从而灵活性大大提高。限制于工艺水平，直到上世纪90年代可重构计算才重新获得重视，成为学术界和产业界的热点。1999年加州大学伯克利分校可重构技术研究中心的Wawrzynek和Dehon提出可重构计算的另外一种定义**错误!未找到引用源。**，它将其视为一类计算机组织结构，有区别于其他组织结构的两类突出特点：制造后芯片的定制能力，即硅实现以后计算功能依旧可以按需改变，区别于传统的专用集成电路；能针对很多算法完成计算引擎的空间映射（区别于传统的指令驱动处理器）。

按照重构粒度划分，可以分为细粒度和粗粒度，这里的粒度是指可重构计算数据通路中运算单元的数据位宽。一般情况下，重构粒度越大可重构计算处理所需的配置信息就越少，重构的速度就越快，相应的功能灵活性也越低。粗粒度可重构架构适用于计算密集型的应用，如通信应用**错误!未找到引用源。错误!未找到引用源。**、密码处理应用**错误!未找到引用源。**和多媒体应用**错误!未找到引用源。错误!未找到引用源。**等；近些年来，已经有很多国内外的公司和科研机构提出了多种不同的可重构架构，如REMAR**错误!未找到引用源。**、PipeRench**错误!未找到引用源。**、ADRE**错误!未找到引用源。**、MorphySy**错误!未找到引用源。**、XPP**错误!未找到引用源。**、Zippy**错误!未找到引用源。**等。

* + 1. 密码算法综述

随着计算机技术和网络通信技术的发展，信息安全问题也逐渐成为人们关注的社会问题，密码技术是保证信息的可用性、机密性和安全性等安全要求的基本手段。密码算法是各种安全应用的基础，也是信息系统安全性的根本所在，高效灵活的密码算法是各种高性能信息系统的重要指标和基本保障，因此成为信息安全领域的重要课题。

密码算法又可分为公钥密码、私钥密码和哈希函数三大类，其中，公钥密码主要是大数的模幂操作，私钥密码和哈希函数则以逻辑运算、移位、置换、替换为基础的迭代操作。密码算法按照密钥特点又被分为对称密码和非对称密码，对称密码在加密和解密时都使用相同的密钥，也称为传统密码，是20世纪70年代在公钥密码产生之前唯一的加密类型，迄今为止，在两种加密类型中仍然是使用最为广泛的加密类型，对称密码主要分为流密码算法和分组密码算法。非对称密码在加密和解密时则会使用不同的密钥，并且很难从一个推出另一个，主要有基于离散对数问题的ECC算法、基于大数分解的RSA算法、杂凑函数（散列算法）等。

密码算法应用常常需要处理较大的信息量，或具有较大的计算强度，往往是各种通信系统中计算密集型环节，影响整个系统的吞吐率。常规的GPP无法满足其速度要求，安全性也不如专用硬件，因此目前国内外对密码处理专用硬件的研究和开发十分活跃。密码算法的硬件加速方式可分为三类：密码算法处理器、特定密码算法ASIC和可重构密码处理结构。

随着移动互联网的飞速发展，对系统安全性的要求也越来越迫切，同时随着互联网大数据通信时代的到来，对安全处理器的性能要求也越来越高，保障系统安全所需投入的处理资源将越来越多，安全应用的范围也会越来越广，密码算法与可重构技术的结合，可以满足性能和安全方面的需求，具体地，可重构密码系统架构在这一应用领域的优势体现在以下几点**错误!未找到引用源。**：

（1）可重构系统可以根据实际需求实现不同的密码算法，具有很大的算法灵活性；

（2）可重构系统的计算能力能够满足密码算法的高性能需求，结构能够根据特定的算法集定制硬件，使算法执行更加高效；

（3）可重构系统具有扩展性，能够适应不断被提出新的更安全的算法，同时支持随时修改密钥，满足某些特殊情况下的白片需求。

* 1. 国内外研究现状
     1. 可重构密码架构综述

近年来，能够同时满足密码算法应用的高性能、高灵活性和低成本需求的可重构密码处理器已成为研究热点。密码算法的实现结果的考核可以以面积效率为标准，由于密码算法的数据依赖性较低，和媒体、通信算法不同，密码算法的实现可以通过增加硬件开销来获得算法实现性能的提升。已有的面向密码的可重构架构在处理单元（Process Element，PE）设计上或多或少存在一些设计上的不足，导致整个架构变得臃肿，存在比较多的计算资源浪费。已有研究为了提高面积效率进行不懈努力，但与专用硬件实现的结果相比仍有显著差距。

COBRA**错误!未找到引用源。**是一款面向对称密钥算法提出的指令级分布式可重构处理器，通过对多种对称密钥算法的映射实现进行验证，AES算法的实现面积效率为0.216Gbps/Mgates。为了保证充分的灵活性和并行计算能力，COBRA架构在每个处理单元中包含了各种算法所需要的所有算子，包括置换单元、S盒等资源开销很大的操作，从而造成了极大的硬件冗余；处理单元中的很多算子被串行组织，最长的路径上串联了13个算子单元，这使处理单元的电路延迟很大，整个处理器的主频非常低，很难实现高性能。

Celator**错误!未找到引用源。**是由艾克斯—马赛大学研发的面向分组密码算法和哈希函数的可重构架构，其计算阵列基于脉动结构设计，采用二维互联结构完成计算单元间的数据传输，每个计算单元支持逻辑操作和算术操作，通过有限状态机控制计算阵列的数据访问和计算操作，有效控制了整体架构的硬件资源开销，DES算法的实现面积效率为0.25Gbps/mm2。但是该架构中处理单元的算子功能很少，支持的算法有限；并且基于指令的设计使得每一个操作都需要重新获取指令进行指令译码再进行数据处理，也无法实现算法的流水设计，算法实现性能不高。

Cyptoraptor**错误!未找到引用源。**是由德州大学奥斯汀分校研发的一款高性能、低功耗、高灵活的密码处理器，面向分组密码、流密码、哈希函数三类百余种对称密钥算法进行架构探索，是目前研究支持算法最多的一款，DES和AES算法的实现面积效率分别为6.75Gbps/mm2和20.25Gbps/mm2。该架构通过灵活的计算单元和阵列互联模块，大幅提升了阵列的流水效率，同时对计算单元支持的算子操作进行组合，有效缩短了关键路径时延和流水级数，从而提升了计算性能。然而，基于该架构映射的对称密钥算法中，比较多的算法实现硬件利用率不高，影响了面积效率，其原因在于未能考虑不同算法间的特征差异，导致硬件架构存在过度设计。

ProDFA[18]是由国防科技大学提出的一种基于可编程数据流计算的体系结构框架，DES和AES算法的实现面积效率分别为6.09Gbps/mm2和5.83Gbps/mm2。通过分析架构可编程性和数据流计算特性与控制逻辑属性的关系在一个处理单元中通过多周期运算，实现完整的算法计算。迭代结构可以节省电路开销，多个处理单元集成可以实现高吞吐率。在处理单元设计时，每一个算子都单独设计，没有考虑与其它算子进行串行组合，这导致完成算法需要很多个处理周期，因此单个处理单元完成算法的性能很低，要想实现高性能需要很多的处理单元集成。

RCPA[19]是由解放军信息工程大学研究提出的一款高性能和高灵活的密码处理器，该架构利用处理单元阵列和阵列间的灵活互连，支持分组密码算法的流水展开映射实现，同时也考虑了算子单元之间的简单组合，缩短映射的流水级数，从而提升了计算性能。与Cyptoraptor架构类似，处理单元中算子单元的过度设计使算法实现时存在很多的闲置单元，硬件利用率不高，影响了面积效率。

综上所述，现有的面向密码算法的可重构架构在设计处理单元时，在算子组合上存在过度组合导致处理单元延迟过大的问题，或者不进行算子组合，存在映射周期过多的问题；处理单元中包含了算法所需的所有算子单元，算法映射时大部分算子单元被闲置，存在硬件利用率不高的问题。算子组合关系到处理器性能，算子利用率关系到芯片面积，因此建立一个统一的算法模型，充分挖掘算法的算子组合特征和算子位置特征，根据提取的特征，提出一套更合理的处理单元设计方案，优化算子组合，提高算子利用率，从而提升整个架构的面积效率。

* + 1. 可重构密码PE概述

可重构密码PE作为可重构架构的核心功能部件，是可重构密码架构设计中的关键环节。架构要完成的所有的运算都会在PE上执行，架构中的其它模块都只是使这些PE能正常高效工作的保障。PE的功能决定了架构的功能，PE的性能决定了架构的性能。当前的几类可重构密码PE方案在PE功能设计上过于臃肿，硬件利用率很低，影响了整个架构的面积效率。可重构密码PE内部包含了完成密码算法所必须的各种功能单元，这些功能单元按照组合的方式不同可以分为串行组合、并行组合、串并混合：

* 串行组合

串行组合方式中，各可重构功能单元按照一定的次序串行连接，对不需要执行运算的进行旁路。早期的可重构密码处理器大多采用串行组合，这种组合方式在设计中要结合密码算法的算子组合特点，对算法的处理顺序进行分析并找出最佳的连接顺序；这种组合方式每次可以依次执行多种运算，提高了执行效率。但串行组合设计使路径延迟达到了最大，降低了整体的最高运算时钟频率，架构的整体性能很低；同时串联组合设计需要与算法中算子的组合紧密关联，但是没有一种组合策略可以兼顾所有算法，过度的串行组合会使算子利用率变得很低。

* 并行组合

并行连接方式中，各可重构功能单元并行连接，通过数据选择器选择相应的功能单元参与运算。这种设计简单且易于实现，并且使路径延迟最小，整个处理器可以有很高的主频。这种结构的缺点是每次只能执行一种的运算，完成算法的完整流程需要很多PE参与，因此这种组合方式一般出现在可重构密码阵列架构中，用更多的PE数量来弥补单个PE功能组合上的缺陷，通过增大面积开销来保证很高的性能。同样，并行组合方式并没有减少资源冗余，它用牺牲面积来换取更高的性能。

* 串并混合

串并混合方式中，整体上采用并行组合的模型，在某些并行通路上采用简单的串行组合。相关度低的功能单元将被并行组合，相关度高的功能单元将被串行组合，同时考虑并行通路之间的延迟平衡。在并连时，通过数据选择器选择所需的运算的结果，串连时则对不需要执行运算的功能单元进行旁路。

三种PE结构在性能和执行效率上各有差异，但是它们都面临着功能利用率极低的问题。表1-1对三种PE方案的代表架构中PE面积占整个架构的面积比例以及在映射不同算法时各种功能单元的利用率进行了总结，可以发现：

表1- 1 不同架构的功能单元利用率

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 架构  名称 | PE方案 | PE占整个架构面积比例 | 功能单元利用率 | | | | | | | | |
| 算  法 | 功能单元 | | | | | | | |
| 算术  单元 | 移位  单元 | 置换  单元 | 逻辑  单元 | S盒 | 有限域乘法 | 模乘  单元 | 平均利用率 |
| COBRA | 串行组合 | 40% | AES | 0% | 0% | 0% | 17% | 100% | 50% | 0% | 14% |
| DES | 0% | 0% | 13% | 4% | 25% | 0% | 0% | 3% |
| RC5 | 13% | 8% | 0% | 8% | 0% | 0% | 3% | 7% |
| RCPA | 并行组合 | 90% | AES | 0% | 0% | 0% | 25% | 25% | 50% | 0% | 19% |
| DES | 0% | 0% | 17% | 8% | 8% | 0% | 0% | 6% |
| RC5 | 13% | 13% | 0% | 6% | 0% | 0% | 0% | 5% |
| Cyptoraptor | 串并混合 | 72% | AES | 0% | 0% | 0% | 50% | 14% | / | / | 14% |
| DES | 0% | 0% | 17% | 17% | 2% | / | / | 5% |
| RC5 | 13% | 13% | 0% | 13% | 0% | / | / | 3% |

* PE占架构面积的主要部分

表1-1第三列列出了三种架构中PE在整个架构中的面积占比，串行组合结构由于没有形成处理阵列，占比较低为40%；对于并行组合和串并混合结构，PE面积占比分别达到了90%和72%；对于这两类架构，PE成了整个架构面积的主体，是提升整体面积效率的最大入手点。

* PE中的功能单元利用率很低

表1-1的分功能单元利用率部分列出了三种代表架构在实现三种典型密码算法时各种功能单元的利用率。表中列出了PE中最常见的几种功能单元：算术单元、移位单元、置换单元、逻辑单元、S盒替换单元、有限域乘法单元、模乘单元各自的利用率，以及架构中所有功能单元的平均利用率。各种功能单元利用率在0%到100%之间，很多功能单元都没有被使用，利用率为0%，使用到的功能单元利用率也大多在20%以下。三种架构在平均利用率上也很相似，AES的平均利用率为15%左右，其它两个算法的的平均利用率只有5%左右。

在架构设计时，为了兼顾所有的算法，一定的功能单元冗余是必然的；但是当前的可重构密码架构在运行各单一算法时冗余比例达到了85%-95%，也就是说在运行某一个算法时，整个芯片有70%-85%的面积是一直被闲置的，这些冗余严重影响了算法的面积效率。

* 1. 论文研究内容及意义

本文针对分组密码算法的可重构实现提出了一套新的PE设计方案，用于减小功能单元的冗余，提高资源利用率，从而提高整个可重构平台的面积效率。

如图1-1所示，整个研究过程可以分为八个部分：

1. 选取目标算法集，研究这些算法的详细定义；本文选取常见的几十种分组密码算法作为课题研究的目标算法集合。
2. 建立了一个统一的模型来描述算法，使后续对大量算法的一系列自动化处理成为可能；课题中采用有向图来描述分组密码算法。



图1- 1 研究方案

1. 基于建立的算法图模型，提取分组密码算法的一系列特征，包括算法算子的参数特征、组合特征和位置特征，这些特征和架构设计息息相关。
2. 确定PE设计初始方案，根据算法集中提取的算子参数特征、组合特征和位置特征分别确定PE方案中功能单元的功能参数、功能组合和功能分布。
3. 对可重构阵列架构建立有向图建模，同时将架构中的功能单元和互连单元参数化；一方面这个模型作为一个超图，算法图可以这个超图中完成匹配映射，另一方面，参数化的功能和互连模型可以很方便地完成架构调整。
4. 提出一套映射方案，完成目标算法集合中的算法图到架构超图的映射；本文通过对算法映射的研究分析，将算法映射问题归纳成图论中的子图同构问题，通过对子图同构算法的调整，同时定义一系列资源成本约束函数，将子图同构问题的解决方法成功应用到算法映射中。
5. 分析目标算法集合的映射结果，统计初始架构中各个功能单元的使用情况，针对架构中的冗余单元进行进一步优化。
6. 最后对优化后的架构进行RTL实现验证，并且与其它可重构密码架构进行对比分析。

本文提出的PE研究方案致力于改善当前可重构密码架构中功能单元利用率低，架构整体面积效率偏低的问题，因此在进行最终的实验考核中，主要有两个指标，一个是功能单元的利用率，另一个是架构的最终性能面积比。由于课题中涉及到大量的分组密码算法，课题中的这两个指标都采用算法集合中所有算法映射结果的平均值。

如表1-2所示，功能单元利用率相对于其它所有架构平台平均提高50%以上，面积效率相对于其它所有架构平台平均提高30%以上。

表1-1 考核指标

|  |  |
| --- | --- |
| 考核指标 | 相应提升要求 |
| 功能单元平均利用率 | 相较于其它所有架构平台提高50%以上 |
| 性能面积比 | 相较于其它所有架构平台提高30%以上 |

图1-1中的实线框标示的过程是一般可重构密码架构研究的过程：算法选取-特征提取-确立方案-RTL实现验证。如图1-1的虚线框标示所示，本文提出的研究方法加入了对算法和架构的建模分析，开发了一系列基于所建模型的自动化分析工具，扩大目标算法集合，提取更多的算法特征；基于子图同构的映射方法可以对大量算法实施映射验证，并根据映射结果进行反馈设计，同时参数化的架构模型可以很方便地对架构进行设计调整并在自动化工具中得到验证。在整个课题工作中主要有以下创新点：

1. 对算法和架构建立一致的图模型，使大量算法的特征提取和自动化映射分析成为可能。自动化分析不仅可以扩大目标算法集合，还能提取很多无法通过人工直接观察获取的算法特征；同时参数化的架构模型可以很方便地对架构进行设计调整并在自动化工具中得到验证；
2. 在以往架构平台的PE设计中只考虑了分组密码算法中算子的参数特征和组合特征；本文在此基础上，开发自动化分析工具，提取了不同算子在算法轮中的位置特征，分析不同算法中这些算子位置特征的共性，根据位置特征的共性来安排PE设计中的功能单元分布；其它架构在PE设计时由于缺乏算子位置相关的特征分析，只能对功能单元进行全功能分布，这种全功能分布是功能单元利用率低下的主要原因，这造成了很大的资源浪费；
3. 提出了一套基于图模型的自动化映射方案，完成目标算法集合中的算法图到架构超图的映射；本文通过对算法映射的研究分析，将算法映射问题归纳成图论中的子图同构问题，通过对子图同构算法的调整，同时定义一系列资源成本约束函数，将子图同构问题的解决方法成功应用到算法映射中。在以往的架构设计中，算法的映射通过手工完成，由于算法映射过程繁杂，映射只能针对AES、DES等少数几个算法，无法对更多的算法进行映射验证，架构设计具有很明显的算法偏向性；
4. 最终PE设计方案确定是多轮迭代优化的结果，如图1-1所示，算法集合在初始架构中完成映射，通过映射结果分析，统计初始架构中各个功能单元的使用情况，针对架构中的冗余单元进行进一步优化，并对优化后的架构进行映射验证，重复上述过程，直到架构中不存在冗余单元。在传统的设计中，只对少数的几个算法进行功能验证，没有针对大量算法映射结果的冗余分析，也不存在针对冗余优化的反馈设计。

论文的研究意义在于：以分组密码算法的高面积效率为目标，提出了一套设计粗粒度可重构PE方案的研究方法，并对方案进行了验证；在一定程度上解决了当前可重构密码架构平台普遍面临的功能单元利用率低和整体面积效率不高的问题，充分发挥粗粒度可重构系统在以分组密码算法为代表的数据密集计算领域的独特优势。

* 1. 论文组织结构

本文共分为7章，各章的主要内容具体如下：

第一章为绪论，从可重构计算、密码算法角度介绍本文的相关研究背景，介绍了可重构密码架构和可重构密码PE的国内外研究现状，阐述了论文的研究内容和意义；

第二章简单介绍了分组密码算法的四种一般结构：Feistel、SP、LM和ARX，并分别介绍了三种结构的代表算法DES、AES、IDEA和SPECK。本章还给出了面向分组密码算法的可重构架构的结构介绍，重点介绍了可重构系统中计算引擎部分的结构，包括计算阵列、通用寄存器堆、输入输出通道等；

第三章建立了分组密码算法的统一图模型，在这个图模型上提取分组密码算法算子的参数特征、组合特征和位置特征；

第四章根据提取的算子参数特征、组合特征和位置特征分别确定PE方案中功能单元的功能参数、功能组合和功能分布，提出PE初始设计方案；

第五章提出算法映射方案，将算法映射问题归纳成图论中的子图同构问题，通过对子图同构算法的调整，同时定义一系列资源成本约束函数，将子图同构问题的解决方法成功应用到算法映射中。通过大量的算法映射分析，完成PE设计的反馈优化。

第六章对本文的PE设计进行了验证，分析了方案在功能单元利用率和面积效率方面的实验结果，并与其它几种最新的可重构密码平台在功能单元利用率和面积效率上进行对比分析；

第七章对本文的工作进行了总结，并对未来进一步的工作进行了展望。

1. 分组密码算法与密码可重构架构
   1. 分组密码算法

现代密码算法主要分为两大类对称密码和非对称密码，对称密码根据加解密操作位数的不同分为分组密码和序列密码两类非对称密码加解密密钥不同，又称公钥密码。分组密码和公钥密码是现代安全系统中使用最多的两类算法，很多主流分组密码和公钥密码算法是公开的，为本论文的研究提供了基础。

* + 1. 分组密码简介

分组密码（Block Cipher）又称秘密钥密码（Secret Key Cipher）是用于数据加解密的主要算法。利用分组密码对明文进行加密时，首先需要对明文进行分组，每组的长度都相同，然后对每组明文分别加密得到等长的密文。分组密码在设计上的特点是加密密钥与解密密钥相同。分组密码的安全性应该主要依赖于密钥，而不依赖于对加密算法和解密算法的保密。因此，分组密码的加密和解密算法是可以公开的[1]。

分组密码具有速度快、易于标准化和便于软硬件实现等特点，通常是信息与网络安全中实现数据加密、数字签名、认证及密钥管理的核心体制，可以用于重点信息的加密。使用分组密码容易实现同步，因为一个密文组的传输错误不会影响其他组，丢失一个明密文不会对其后续组解密的正确性产生影响。

分组密码通常具有比较整齐的数据位宽，需要大量重复的操作，执行速度比较快，很适合硬件实现，在密码领域的使用频度最大。分组密码的加解密算法结构非常规整，加密和解密过程的计算结构相同，只是某些对应操作和使用的常数、S盒等略有不同。分组密码的加解密处理过程通常分为三部分初始变换、中间变换和末尾变换。初始变换和末尾变换只执行一次，中间变换反复执行多轮，每轮的计算结构大体相同，加解密过程利用密钥扩展得到的子密钥对明文或密文进行一系列算术、逻辑、置换及代替等操作，密码算法的安全性主要依靠中间变换的强度来体现。很多情况下，密码算法的加解密过程要使用大于密钥量的子密钥参与处理，子密钥是由密钥扩展形成的，有些算法的子密钥就等于算法密钥，有些算法的密钥扩展过程比较简单，但有些算法密钥扩展需要的计算量很大。需要复杂计算的密钥扩展过程通常会使用和密码算法相同或相似的运算类型及操作模块[1]。

* + 1. 分组密码算法的数学模型

任何可计算的算法都具有基本的数学模型，其算法的描述也都可以用数学符号来表述。分组密码算法的数学模型可抽象为明文数据，密钥，加密过程，密文数据，解密过程。

其过程为：将明文信息编码形成的序列分成等长的分组，即输入明文数据或称待加密数据，同时输入密钥序列，在加密密钥的控制下通过加密算法变换成等长的输出密文数据（加密过程）。加密过程生成的密文数据，经过存储或传输，再输入解密密钥序列。在解密密钥的控制下通过解密算法变换成等长的输出明文数据（解密过程）。

分组密码是将明文消息编码后的数字（通常是0与1）序列x1，x2，…划分成长为m的组x=(x1，x2，…，xm)，每个明文组（长为m的向量）分别在密钥k=（k1，k2，…，kt）的控制下变换成等长的输出数字序列y=（y1，y2，…，yn）（长为n的向量），整个运行过程的的数学模型如图2-1所示。

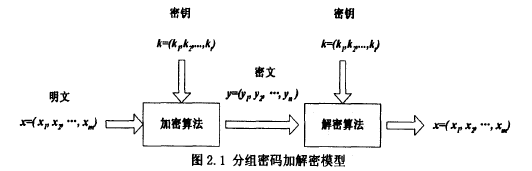


图2-1中，若n>m，则为有数据扩展的分组密码；如n<m，则为有数据压缩的分组密码；如n=m，则为无数据扩展和压缩的分组密码，通常研究的均为这种情况。本文设明文x和密文y均为二元（0与1）的序列。设是二元域，表示上的s维向量空间。假定明文空间和密文空间均为，秘钥空间是的一个子集。m是明文和密文的分组长度，t是秘钥长度。

* + 1. 分组密码的整体结构特征及代表算法

目前分组密码所采用的整体结构可分为Feistel网络结构，SP网络结构，LM结构以及ARX结构。Feistel结构的由于DES算法的公布而广为人知，己被许多分组密码所采用。Feistel结构的最大优点是容易保证加解密相似，扩散较慢。SP结构比较难做到加解密相似，但是扩散特性比较好。LM结构则基于不同代数群的混合运算来设计的。

* + - 1. Feistel网络结构及其代表算法
* Feistel网络结构

很多分组密码算法都是网络或扩展网络型结构，其典型代表是DES密码算法，它体现了现代密码学理论设计密码算法的原则。。取长度为n的分组，把它分成长度为n/2的两部分L和R，n必须是偶数。可以定义出一个迭代型的分组密码算法，其第i轮的输入来自于前一轮的输出：

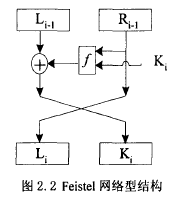




Ki是第i轮使用的子密钥，F是任意轮函数。除了DES算法以外，常见的CAST、Twofish、RC6算法都属于Feistel网络型分组密码算法。Feistel网络型结构保证了算法的可逆性，用异或来合并左半部分和轮函数的输出，它满足：



因此只要在每轮中F的输入能重新构造，使用了该结构的密码就可保证它是可逆的。它不管F函数如何，也不需要它是可逆的。我们能将函数F设计成我们希望的那样复杂，并且不必实现两个不同算法（分别用于加解密）。Feistel网络结构将自动实现这些。总的来说：Feistel网络的优点在于加解密结构的相似性，缺点在于扩散较慢，算法至少需要两轮才能改变每一个比特。该结构如图2-2所示。



* DES密码算法

DES算法时Feistel结构的代表算法，它由美国国家标准局在1997年提出，算法的分组长度为64位的，密钥长度为56位。DES算法的输入数据宽度为64位，经过处理输出64位数据。解密过程和加密过程是完全相同的。

图2-3是DES算法的执行流程图。DES算法有两个输入：明文和密钥。从图2-3左半部分可见明文的处理经过了三个阶段：首先，输入64位明文经过初始置换（IP）处理，数据被重新排列；接着数据经过16轮轮函数处理，这些轮函数的内容都包含置换操作和替换操作，最后一轮轮函数输出64位数据；最后数据经过初始逆置换（IP-1）处理输出64位密文。图2-3的右半部分给出了密钥生成的过程，初始密钥经过置换后，再经过循环左移操作和另一个置换操作，得到每轮轮函数对应的子密钥Ki用于各轮轮函数。DES算法的密钥生成部分的置换操作内容都是一样的，但各轮循环左移的位数互不相同。



图2-3 DES加密算法的总体描述

DES加密算法可以分为加密和密钥生成两个部分，以下分开介绍：

1、DES分组密码算法的加密部分

DES算法的初始置换和逆置换的内容见表2-1(a）和表2-(b)。输入数据按照位数被分别标记为1到64，共64位。置换表中的每个元素的数值表明了某个输入位在64位输出中的位置。

表2-1 DES置换表

(a)初始置换

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 | 60 | 52 | 44 | 36 | 28 | 20 | 12 | 4 |
| 62 | 54 | 46 | 38 | 30 | 22 | 14 | 6 | 64 | 56 | 48 | 40 | 32 | 24 | 16 | 8 |
| 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 | 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 61 | 53 | 45 | 37 | 29 | 21 | 13 | 5 | 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 |

(b)逆初始置换

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 40 | 8 | 48 | 16 | 56 | 24 | 64 | 32 | 39 | 7 | 47 | 15 | 55 | 23 | 63 | 31 |
| 38 | 6 | 46 | 14 | 54 | 22 | 62 | 30 | 37 | 5 | 45 | 13 | 53 | 21 | 61 | 29 |
| 36 | 4 | 44 | 12 | 52 | 20 | 60 | 28 | 35 | 3 | 43 | 11 | 51 | 19 | 59 | 27 |
| 34 | 2 | 42 | 10 | 50 | 18 | 58 | 26 | 33 | 1 | 41 | 9 | 49 | 17 | 57 | 25 |

(c)扩展置换

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 32 | 1 | 2 | 3 | 4 | 5 | 6 | 5 | 6 | 7 | 8 | 9 |
| 8 | 9 | 10 | 11 | 12 | 13 | 12 | 13 | 14 | 156 | 16 | 17 |
| 16 | 17 | 18 | 19 | 20 | 21 | 20 | 21 | 22 | 23 | 24 | 25 |
| 24 | 25 | 26 | 27 | 28 | 29 | 28 | 29 | 30 | 31 | 32 | 1 |

(d)置换函数

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 7 | 20 | 21 | 29 | 12 | 28 | 17 |
| 1 | 15 | 23 | 26 | 5 | 18 | 31 | 10 |
| 2 | 8 | 24 | 14 | 32 | 27 | 3 | 9 |
| 19 | 13 | 30 | 6 | 22 | 11 | 4 | 25 |

图2-4给出了一轮变换的内部结构。在左半部分，64位中间数据的左右两部分作为独立的32位数据，分别记为L和R。每轮变换的整个过程可以写为式和式：





轮密钥Ki长48位，R为32位。首先将R按照表2-(c)定义的置换扩展为48位数据，其中有16位内容重复的。将输出的48位数据与Ki进行异或操作，之后通过一个替换函数处理，获得32位输出数据，最后按照表2-1(d)定义的置换处理之后输出。



图2-4 DES算法一轮迭代的过程

图2-5解释了F函数中的S盒操作。替代函数由8个S盒来组成，每个S盒都输入6位，输出4位。这些变换参见表2-2，其解释如下：盒Si输入的第一位和最后一位组成一个2位的二进制数，用来选择S盒4行代替值中的一行，中间4位用来选择16列中的某一列。行列交叉处的十进制1(01)，列是12(1100)，该处的值是9，所以输出为1001。



图2-5 DES加密算法中的F函数

2、DES分组密码算法的密钥生成部分

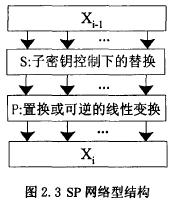
密钥生成可以参见图2-3和图2-4，可以看到算法输入了64位的密钥，见图2-3右半部分密钥通过置换表格得到56位数据，分为两个28位数据C和D。每轮迭代中C和D分别循环左移一位或者两位。移位后作为下一轮的输入，同时作为置换选择的输入，将产生的一个48位的输出作为函数F的输入。

表2-2 DES加密算法的S盒定义

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S1 | | | | | | | | | | | | | | | |
| 14 | 4 | 13 | 1 | 2 | 15 | 11 | 8 | 3 | 10 | 6 | 12 | 5 | 9 | 0 | 7 |
| 0 | 15 | 7 | 4 | 14 | 2 | 13 | 1 | 10 | 6 | 12 | 11 | 9 | 5 | 3 | 8 |
| 4 | 1 | 14 | 8 | 13 | 6 | 2 | 11 | 15 | 12 | 9 | 7 | 3 | 10 | 5 | 0 |
| 15 | 12 | 8 | 2 | 4 | 9 | 1 | 7 | 5 | 11 | 3 | 14 | 10 | 0 | 6 | 13 |
| S2 | | | | | | | | | | | | | | | |
| 15 | 1 | 8 | 14 | 6 | 11 | 3 | 4 | 9 | 7 | 2 | 13 | 12 | 0 | 5 | 10 |
| 3 | 13 | 4 | 7 | 15 | 2 | 8 | 14 | 12 | 0 | 1 | 10 | 6 | 9 | 11 | 5 |
| 0 | 14 | 7 | 11 | 10 | 4 | 13 | 1 | 5 | 8 | 12 | 6 | 9 | 3 | 2 | 15 |
| 13 | 8 | 10 | 1 | 3 | 15 | 4 | 2 | 11 | 6 | 7 | 12 | 0 | 5 | 14 | 9 |
| S3 | | | | | | | | | | | | | | | |
| 10 | 0 | 9 | 14 | 6 | 3 | 15 | 5 | 1 | 13 | 12 | 7 | 11 | 4 | 2 | 8 |
| 13 | 7 | 0 | 9 | 3 | 4 | 6 | 10 | 2 | 8 | 5 | 14 | 12 | 11 | 15 | 1 |
| 13 | 6 | 4 | 9 | 8 | 15 | 3 | 0 | 11 | 1 | 2 | 12 | 5 | 10 | 14 | 7 |
| 1 | 10 | 13 | 0 | 6 | 9 | 8 | 7 | 4 | 15 | 14 | 3 | 11 | 5 | 2 | 12 |
| S4 | | | | | | | | | | | | | | | |
| 7 | 13 | 14 | 3 | 0 | 6 | 9 | 10 | 1 | 2 | 8 | 5 | 11 | 12 | 4 | 15 |
| 13 | 8 | 11 | 5 | 6 | 15 | 0 | 3 | 4 | 7 | 2 | 12 | 1 | 10 | 14 | 9 |
| 10 | 6 | 9 | 0 | 12 | 11 | 7 | 13 | 15 | 1 | 3 | 14 | 5 | 2 | 8 | 4 |
| 3 | 15 | 0 | 6 | 10 | 1 | 13 | 8 | 9 | 4 | 5 | 11 | 12 | 7 | 2 | 14 |
| S5 | | | | | | | | | | | | | | | |
| 2 | 12 | 4 | 1 | 7 | 10 | 11 | 6 | 8 | 5 | 3 | 15 | 13 | 0 | 14 | 9 |
| 14 | 11 | 2 | 12 | 4 | 7 | 13 | 1 | 5 | 0 | 15 | 10 | 3 | 9 | 8 | 6 |
| 4 | 2 | 1 | 11 | 10 | 13 | 7 | 8 | 15 | 9 | 12 | 5 | 6 | 3 | 0 | 14 |
| 11 | 8 | 12 | 7 | 1 | 14 | 2 | 13 | 6 | 15 | 0 | 9 | 10 | 4 | 5 | 3 |
| S6 | | | | | | | | | | | | | | | |
| 12 | 1 | 10 | 15 | 9 | 2 | 6 | 8 | 0 | 13 | 3 | 4 | 14 | 7 | 5 | 11 |
| 10 | 15 | 4 | 2 | 7 | 12 | 9 | 5 | 6 | 1 | 13 | 14 | 0 | 11 | 3 | 8 |
| 9 | 14 | 15 | 5 | 2 | 8 | 12 | 3 | 7 | 0 | 4 | 10 | 1 | 13 | 11 | 6 |
| 4 | 3 | 2 | 12 | 9 | 5 | 15 | 10 | 11 | 14 | 1 | 7 | 6 | 0 | 8 | 13 |
| S7 | | | | | | | | | | | | | | | |
| 4 | 11 | 2 | 14 | 15 | 0 | 8 | 13 | 3 | 12 | 9 | 7 | 5 | 10 | 6 | 1 |
| 13 | 0 | 11 | 7 | 4 | 9 | 1 | 10 | 14 | 3 | 5 | 12 | 2 | 15 | 8 | 6 |
| 1 | 4 | 11 | 13 | 12 | 3 | 7 | 14 | 10 | 15 | 6 | 8 | 0 | 5 | 9 | 2 |
| 6 | 11 | 13 | 8 | 1 | 4 | 10 | 7 | 9 | 5 | 0 | 15 | 14 | 2 | 3 | 12 |
| S8 | | | | | | | | | | | | | | | |
| 13 | 2 | 8 | 4 | 6 | 15 | 11 | 1 | 10 | 9 | 3 | 14 | 5 | 0 | 12 | 7 |
| 1 | 15 | 13 | 8 | 10 | 3 | 7 | 4 | 12 | 5 | 6 | 11 | 0 | 14 | 9 | 2 |
| 7 | 11 | 4 | 1 | 9 | 12 | 14 | 2 | 0 | 6 | 10 | 13 | 15 | 3 | 5 | 8 |
| 2 | 1 | 14 | 7 | 4 | 10 | 8 | 13 | 15 | 12 | 9 | 0 | 3 | 5 | 6 | 11 |

* + - 1. SP网络结构及其代表算法
* SP网络结构

SP网络是Feistel网络的一种推广结构，网络中的S和P分别是Substitution和Permutation的首字母，AES、SAFER、SHARK等著名密码算法都采用SP结构。在SP结构算法的每一轮中，首先轮输入被作用于一个由子密钥控制的可逆函数S（非线性变换），然后再被作用于一个置换（或一个可逆的线性变换）P。S一般被称为混淆层，提供了分组密码算法所必须的混淆作用。P一般被称为扩散层，主要是提供快速的扩散，实现雪崩效应。该结构中轮函数一般由S盒层和P置换（或线性变换）组成，分别提供必要的混淆性和扩散性，所以通常也称S盒层为混淆层，称P置换（或线性变换）为扩散层。SP网络与Feistel网络相比的主要优势在于可以得到更快的扩散速度。其缺点是加解密很难达到一致。该结构如图2-6所示。



* AES算法

AES算法时目前最被广泛使用的对称分组密码算法，它是SP网络结构的代表算法。美国国家标准技术研究所在2001年发布了AES，旨在取代DES成为广泛使用的标准。如图2-7展示了AES加密过程的总体结构。明文分组长度为128位即16字节，密钥长度可以为16，24或32字节（128，192或192位）。根据密钥的长度，算法被成为AES-128，AES-192或者AES-256，根据密钥长度的不同，AES迭代的论述也不相同，AES-128迭代10轮，AES-192迭代12轮，AES-256迭代14轮。图2-7所示为AES-128的结构。

如图2-7所示AES加密算法由轮函数和密钥生成组成。轮迭代又由四个部分组成，分别是字节替换、行移位、列混合和密钥加。加密和解密类似，只是执行顺序相反。



图2-7 AES加密算法的执行流程

1、字节替换

AES算法的字节替换变换是通过一个简单的查表操作实现，见图2-8。字节替换是一个查表过程，输入是字节，输出也是字节。如图，S1,1通过字节替换操作，转换为了S’1,1。



图2-8 AES字节替换

AES定义了一个S盒，见表2-33，它是由16×16个字节组成的矩阵，包含了8位所能表示的256个数的一个置换。输入一个8位的查表数据，把高4位作为行值，低4位作为列值，以这些行列值作为索引从S盒的对应位置取出元素作为输出。例如十六进制数95所对应的S盒的行值是9，列值是5，S盒中在此位置的值是2A，相应地，95被映射为2A。

表2-3 AES的S盒定义

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | y | | | | | | | | | | | | | | | |
|  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| x | 0 | 63 | 7C | 77 | 7B | F2 | 6B | 6F | C5 | 30 | 01 | 67 | 2B | FE | D7 | AB | 76 |
| 1 | CA | 82 | C9 | 7D | FA | 59 | 47 | F0 | AD | D4 | A2 | AF | 9C | A4 | 72 | C0 |
| 2 | B7 | FD | 93 | 26 | 36 | 3F | F7 | CC | 34 | A5 | E5 | F1 | 71 | D8 | 31 | 15 |
| 3 | 04 | C7 | 23 | C3 | 18 | 96 | 05 | 9A | 07 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| 4 | 09 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 5 | 53 | D1 | 00 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 02 | 7F | 50 | 3C | 9F | A8 |
| 7 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| 8 | CD | 0C | 13 | EC | 5F | 97 | 44 | 17 | C4 | A7 | 7E | 3D | 64 | 5D | 19 | 73 |
| 9 | 60 | 81 | 4F | DC | 22 | 2A | 90 | 88 | 46 | EE | B8 | 14 | DE | 5E | 0B | DB |
| A | E0 | 32 | 3A | 0A | 49 | 06 | 24 | 5C | C2 | D3 | AC | 62 | 91 | 95 | E4 | 79 |
| B | E7 | C8 | 37 | 6D | 8D | D5 | 4E | A9 | 6C | 56 | F4 | EA | 65 | 7A | AE | 08 |
| C | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | DD | 74 | 1F | 4B | BD | 8B | 8A |
| D | 70 | 3E | B5 | 66 | 48 | 03 | F6 | 0E | 61 | 35 | 57 | B9 | 86 | C1 | 1D | 9E |
| E | E1 | F8 | 98 | 11 | 69 | D9 | 8E | 94 | 9B | 1E | 87 | E9 | CE | 55 | 28 | DF |
| F | 8C | A1 | 89 | 0D | BF | E6 | 42 | 68 | 41 | 99 | 2D | 0F | B0 | 54 | BB | 16; |

2、行移位

行移位的具体操作如图2-9。输入为一个4列字节组成的数组，其中每列由4个字节组成。在加密过程中，输入数据的第一行的4个字节保持不变，第二行的4个字节通过循环移位一个字节，第三行的4个字节通过循环移位两个字节，最后一行的4个字节通过循环移位三个字节，得到新的数据输出，行移位实质上是一种按字节进行的置换操作。



图2-9 AES的行移位变换

3、列混合

列混合的具体内容如图2-10所示，首先把128bit的操作数按字节排列成4×4的方阵，然后每个字节看做是有限域GF(28)上的元素。这样128bit的输入被当成了有限域GF(28)上的一个4×4方阵。视矩阵的一列为有限域GF(28)上的4维列向量，然后使用有限域GF(28)上的可逆矩阵A对该列向量进行线性变换，所得结果就是该列进行列混淆变换得结果。整个变换过程相当于有限域GF(28)上的一次矩阵乘法。



图2-10 AES的列混合变换

也就是通过左乘矩阵实现列混合，参见式，通过化简可以得到式：





4、密钥加

轮密钥加变换比较简单，如图2-11所示，只需把128位的状态按位与128位的子密钥异或即可。由于是异或运算，所以解密时的密钥加变换仍然是把128位的状态按位与128位的解密子密钥异或。



图2-11 AES的密钥加操作

5、密钥生成

AES的密钥生成的轮数同样与密钥的长度有关，对密钥长为128位的AES算法，密钥扩展总共需要进行10轮迭代。每轮迭代中包含一次密钥加运算，需要一个128位，或16字节，或4个字（每个字由4个字节组成）的子密钥，10轮迭代总共需要40个字。另外，在开始迭代前，需要做一次初始密钥加，需要4个字。所以，整个加密过程需要总量为44个字的子密钥。密钥扩展的目的是如何由128位的原始密钥（也称主密钥）产生总量为44个字或176个字节的子密钥。密钥扩展可以参见图2-12所示。



图2-12 AES密钥扩展

密钥扩展可以总结为式：



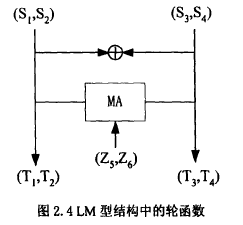
其中的g函数可以参加图2-13所示。



图2-13 AES密钥扩展中g函数结构

* + - 1. LM结构及其代表算法
* LM结构

从操作特性上看，分组密码中有一类密码算法是基于多种代数群的混合运算来设计的，多由算术运算组成。Xuejia Lai和James Massey在1990年公布的PES算法和1991年公布的IDEA算法[1][2]都属于LM结构，如图2-14所示，它与SP网络和Feistel网络都有区别，把这种类型的密码单独归为LM结构，这类密码的混乱与扩散都依赖于MA结构，MA结构采用了混合不同群的运算。当S1、S2、S3和S4用T1、T2、T3和T4代替时，图2-14中表示的MA结构的输入不变，当Z5和Z6不变时，该MA结构的输出也不变，这种结构就保证了加解密是相似的，其安全性同样也基于MA结构。由于MA结构采用的数学运算较多，因此在用软件实现时，在一部分处理器上的执行效率可能不是太高，比AES和DES算法的实现效率低，但是这样的设计使攻击者对算法进行分析变得困难。



* IDEA算法

LM结构的算法比较少，其中的代表算法是国际数据加密算法（International Data Encryption Algorithm，IDEA），它的第一版是由Xuejia Lai和James Massey于1990年公布的，叫做推荐加密标准（Proposed Encryption Standard，PES）。在Biham和Shamir演示了差分密码分析之后，第二年设计者为抗此攻击，增加了密码算法的强度，称新算法为改进型推荐加密标准（Improved Proposedtional Data Encryption Algorithm，IPES）， IPES在1992年又改名为IDEA。

图2-15是IDEA算法的加密流程，64位数据分组分成4个16位的子分组：X0，X1，X2，X3。这4个子分组作为算法的第一轮输入，总共有8轮。每一轮中，这4个子分组间相异或、相加、相乘，且与6个16位的子密钥相异或、相加、相乘。每轮之间间，第二和第三个子分组交换。最后在输出变换中4个子分组与4个子密钥运算。



图2-15 IDEA算法流程

在算法的每一轮中按顺序执行如下操作：

1. X0和第一个子密钥（K1） 模216 +1乘；
2. X1和第二个子密钥（K2） 模216加；
3. X2和第三个子密钥（K3） 模216加；
4. X4和第四个子密钥（K4） 模216 +1乘。；
5. 第步和第（3）步的结果相异或；
6. 将第（2）步和第（4）步的结果相异或；
7. 将第（5）步的结果与第五个子密钥模216 +1乘；
8. 将第（6）步和第（7）步的结果模216加；
9. 将第（8）步的结果与第六个子密钥 模216 +1乘；
10. 将第（7）步和第（9）步的结果模216加；
11. 将第（1）步和第（9）步的结果相异或；
12. 将第（3）步和第（9）步的结果相异或；
13. 将第（2）步和第（10）步的结果相异或；
14. 将第（4）步和第（10）步的结果相异或。

将第(11)、(12)、(13)和(14) 步的结果形成的4个子分组X0,X1,X2,X3作为输出,然后将中间两个分组(X1,X2)交换(最后一轮除外)后，作为为下一轮的输入。

经过8轮运算之后，有一个最终的输出X0,X1,X2,X3,对这4个输出子分组进行如下操作：

1. X0和第一个额外子密钥模216 +1乘。
2. X1和第二个额外子密钥模216加。
3. X2和第三个额外子密钥模216加。
4. X3和第四个额外子密钥模216 +1乘。

最后，这4个子分组重新连接到一起产生密文。

解密过程基本上一样，只是子密钥需要求逆且有微小差别，解密子密钥要么是加密子密钥的加法逆或是乘法逆。计算子密钥要花点时间,但对每一个解密密钥，只需做一次。

产生子密钥也很容易，由128bits的密钥产生52个16bits 的子密钥，即首先将128bits密钥分成8个16bits子密钥（前6个用于第一轮加密，后2个用于第二轮加密的头两个），然后密钥向左循环移25位后再分成8个子密钥（开始4个用在第二轮，后4个用在第三轮），如此六次产生48个子密钥，再一次密钥向左循环移25位，只在前64位生成4个16bits子密钥，合计52个子密钥完成加密过程。IDEA的加密运算都在16bits子分组上运行，只使用三种加密运算算法，而没有位置换。

* + - 1. ARX结构及其代表算法
* ARX结构

ARX结构，是指由模加（Modular Addition）、循环移位（Rotation）和异或（Xor）三种运算通过一定顺序组合构成的一类密码学结构，如图2-16所示。在 ARX 结构中，模加运算为其提供了非线性，循环移位运算提供了运算字内的扩散性，而异或运算则提供了两个运算字间的扩散性。

很多流密码算法使用ARX结构，但是由于ARX结构的密码算法在硬件、软件上实现都表现出相对高的性能和更低的成本，基于该结构的分组密码算法也在最近几年被提出，如SPECK、Simon、Treefish等。这些算法作为轻量级的分组密码算法被应用到移动互联领域和嵌入式设备中。

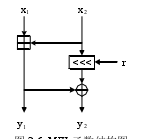


图2-16 ARX结构

* SPECK算法

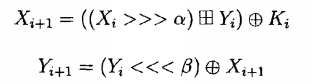
SPECK密码算法[1]是由Ray Beaulieu等人在2013年提出的一种轻量级分组密码算法。SPECK系列分组密码算法的分组长度为32、48、64、96或128比特，密钥长度为64、72、96、128、144、192或256比特，即该算法的分组长度和密钥长度不是固定的，可以根据具体的安全性要求、性能要求以及应用环境等来选择合适的分组长度和密钥长度。

SPECK系列分组密码算法的轮函数结构比较简单，主要由模加运算、循环移位操作和异或运算组成，第i轮加密函数的具体操作过程如图2-17所示。



图2-17 SPECK算法的第i轮加密过程

设Xi和Yi为第i轮的输入分组，Xi+1和Yi+1为第i轮的输出分组，Ki表示第i轮的子密钥，长度均为n比特。轮函数中，首先将左端输入Xi向右循环移位，然后与右端输入Yi进行模加运算，所得结果与子密钥Ki进行异或运算，得到左端的输出五Xi+1；右端输入Yi向左循环移位，再与左端输出Xi+1进行异或运算，得到右端的输出Yi+1，这样就完成了整个轮函数。计算公式如下：



SPECK密码算法的密钥生成函数利用轮函数来生成所需的子密钥ki，设K=（lm-2，…，l0，k0）是算法的主密钥，其中l0，k0是GF(2)n上的值。密钥生成函数的输入为主密钥K，输出为T个子密钥k0，k1，…，kT-1。计算ki和li的公式如下：



[1]Ray Beaulieu，Douglas Shors，Jason Smith，Stefan Treatman-Clark，Bryan Weeks，and Louis Wingers.The SIMON and SPECK Families ofLightweight Block Ciphers，Cryptology ePrint Archive，Report 2013/404，2013.http://eprint.iacr.org/.

* 1. 面向分组密码算法可重构系统架构

本小节给出了面向分组密码算法的可重构架构。整个系统主要由计算引擎模块和配置控制模块两个部分组成，其主要功能模块划分如图2-18所示，其中计算阵列模块的设计是本文的研究内容，会在第四章和第五章中进行详细介绍。



图2-18 面向分组密码算法的可重构架构总体框图

整个系统的工作机制如下：

* 1. 系统上电，配置信息由片外加载到片上配置存储器中；
  2. 配置解析单元解析配置，从配置存储器中选择相应的配置对可重构阵列进行配置；
  3. 阵列从INFIFO中读取数据进行计算，计算结果写入到OUTFIFO中；
  4. 可重构阵列与功能模块计算的中间结果数据只与通用寄存器堆进行交互；
  5. 阵列计算的中间结果通过通用寄存器堆缓存。
     1. 配置控制器

配置控制器由两个部分组成，即为配置存储器和配置解析器。配置控制器的工作流程如图2-29所示，首先系统重置完成之后，完成配置控制器中的配置信息初始化；其次在计算阵列处于可配置状态时，配置解析模块解析配置信息；最后配置控制器发送配置信息给计算阵列从而实现计算引擎的配置，在计算过程中，同样会负责配置信息的切换。



图2-29 配置控制器工作流程图

* + 1. 计算引擎

计算引擎为可重构处理器中主要运算模块，其主要模块包括可重构阵列、通用寄存器堆（General Purpose Register File，GPRF）、数据接口和配置接口，结构框图如图2-18所示。

* + - 1. 可重构阵列

可重阵列的基本单元是行，每行由4个32位的PE单元组成。每一行的输出和下一行的输入模块直接相连。每一行都可以从FIFO或者GPRF读入数据，并且都可以将数据读出到FIFO或者GPRF中。除首行外，每行均可选择由上一行输出作为输入；除尾行外，每行输出均可作为下一行输入。每三行PE单元组成一个异构组，异构组迭代扩展构成可重构阵列。

* + - 1. 通用寄存器堆

通用寄存器堆的容量为64×128bit。其分别有10个独立的128bit读端口与10个独立的128bit写端口，这些端口可以实现对寄存器堆中10个128bit数据的读写操作，而且其写端口都支持以字为单位的mask操作，即每个128bit写端口都由4位的mask信号控制。mask信号高位定义为有效，例如，当某个写端口的mask信号为0010时，写操作仅将数据写入对应128bit寄存器的第3个字，而其余3字的寄存器值保持不变。本架构禁止多个端口同时对同一个地址进行读写操作。

* + - 1. 数据接口

本架构的数据接口比较简单，通过两个FIFO与外部数据进行交互，分别是负责明文输入的INFIFO和密文输出的OUTFIFO，两者的参数一样，宽度都为128bit、深度为32，总计大小为0.5KB的FIFO。整个系统开始工作除了要接收配置控制器的开始信号，还要输入INFIFO处于非空状态，输出OUTFIFO处于非满状态，加密算法的明文通过输入INFIFO进入可重构阵列，经过计算完成之后，加密算法的密文被写入输出OUTFIFO中，再被外部读走，完成整个加密过程，与此同时，加密算法的一些初始化数据也从INFIO输入，包括初始密钥和一些常数等。

* + - 1. 配置接口

配置接口用于可重构阵列与配置控制器之间的配置信息与使能信号的传递。配置接口会把接收到的配置信息，写入到可重构架构的配置寄存器中，而配置寄存器中的配置信息会进一步被可重构阵列的各个模块读取；配置接口同样负责把可重构阵列中的控制单元握手信号传递给配置控制器，同时把配置控制器的使能信号传递给计算阵列。

* 1. 本章小结

本章对分组密码算法进行了简单的介绍，介绍了分组密码算法的基本数学模型和四种不同的算法结构，并分别介绍了四种结构对应的代表算法DES、AES、IDEA和SPECK，同时给出了面向分组密码算法的可重构系统架构的整体结构，简单介绍了架构中的配置控制器和计算引擎。

[1] 姜晶菲.可重构密码处理结构的研究与设计[D].长沙：国防科学技术大学，2004

1. 算法建模与算子特征提取
   1. 算法建模

通用密码可重构处理器旨在提供一个通用的可重构平台同时适用于多种密码算法。这里的多种根据设计目标的不同可能是几种、几十种甚至是几百种。当前的密码可重构架构在设计时通常通过对算法人工观察统计的方式来获取算法特征，进而指导架构设计，最后同构手工映射分析来对架构进行验证。这种人工的方式很难处理大量的算法，特别是对针对大量算法的映射验证分析，往往力不从心。另一方面，很多算法的特征无法简单的通过人工观察来获取，比如关键路径上的功能组合，算法集合表现出来的算子整体位置信息，这些特征需要对算法集合中每一个算法的整体拓扑进行对比才能发现，这样的工作需要计算机去完成。

算法的数据流图（Data Flow Graph，DFG）本身就是一个有向图，因此可以很方便地为算法建立一个有向图模型，给顶点和边赋予一定的属性，使这个有向图能包含该算法的所有信息。课题中的特征提取和算法映射分别使用了有向图的两种表示，一种是边表示的活动网络（Activity On Edges network，AOE），另一种是基于顶点活动网络（Activity On Vertex network，AOV）。

在算法DFG中，将算法的操作用顶点表示，算法的数据依赖用边来表示，这和有向图的描述是一致的，因此可以很方便地将算法的数据流图转换成有向图的AOV表示，数据流图和AOV网络拥有完全一样的结构，只需要将数据流图的操作转换成AOV网络中的点，对这些点赋予对应的属性，然后将数据流图的数据流向转换成AOV网络中的有向边，这样就完成了数据流图到AOV网络的转换。



图3-1算法图的AOV表示

**定义3.1（算法图AOV表示）：**一个架构图是一个二元组ALG=(V,E)，其中V是节点集，表示算法中的操作；E是边集，表示操作之间的数据依赖关系。

如图3-1所示，图中(a)是算法中的数据流程图的一部分，它标示了算法的各种操作以及数据方向，(b)则是与数据流图对应的算法图模型。表3-1列出了对应节点的属性，通过给(b)的节点赋予这些属性，(b)能够描述算法的完整信息。

表3-1算法图结点属性

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 节点 | x | y | 操作属性 | 节点 | x | y | 操作属性 |
| v1 | 1 | 1 | xor | v5 | 3 | 3 | lut |
| v2 | 2 | 1 | sh | v7 | 4 | 1 | xor |
| v3 | 3 | 1 | lut | v8 | 5 | 1 | au |
| v4 | 3 | 2 | lut | v9 | 6 | 1 | xor |

表3-1中节点属性中的y属性标示了各个操作在并行上左右顺序，这个信息会被用到算法映射中的成本函数中，来找出没有功能交叉的映射，这部分会在第五章详述。X属性对应纵向上的顺序，用来获取大算子的位置信息。节点的操作属性用来完成映射时的功能匹配，在算法模型和架构模型中，操作属性都用字符串来描述，通过字串包含的方式来验证功能包含。

在进行算法组合特征提取时，关键路径上的算子组合才是有效的，因为只有通过算子组合减少关键路径的长度才有可能减少算法的映射规模提高映射效率因此需要对算法的关键路径进行分析。图论中的关键路径算法是AOE网络上进行的，AOE网络的向图中，用有向边来表示一个工程中的各项活动（Activity），用有向边上的权值表示活动的持续时间（Duration），用顶点表示事件（Event）。对应算法的AOE网络，如图3-2所示，用顶点表示算法的不同阶段，用边表示算法的的各项操作，边的权值表示各操作的电路延迟，边的方向表示数据流向。



图3-2算法图的AOE表示

* 1. 算法的算子特征

通过第二章中对分组密码的数学模型和结构特征分析可以发现，目前大多数分组密码算法的设计都基于一些相似的设计理论和结构模型，这些算法在处理结构也很相似，涉及的操作类型有较大的交集[1]。因此我们可以得出这样的结论：很多不同的分组密码算法具有相同或相似的基本操作成分，或者说同一基本操作成分在不同的算法中出现的频率很高。如：异或、移位、置换、S盒、有限域乘法以及模加/减运算等。

对于架构设计人员，充分了解算法，提取尽可能多的有用特征来指导架构设计是十分有必要的。在进行架构的PE设计时，主要关注算法的算子特征，以下几类算子特征是PE设计的决定性因素：

* 参数特征

参数特征主要指算法的一些基本特征和基本操作的功能特征，比如算法的分组大小、轮数、秘钥长度，移位操作的左右移位、循环移位等操作模式、S盒的不同输入输出位宽等。通过对算法集中算法进行统计，可以总结这些特征。

* 组合特征

组合特征指算法的关键路径中频繁出现的算子组合，很多算子会在算法中组合出现，比如S盒操作后面一般会接着进行异或操作。对于这种组合操作，可以在上一行PE中先进行S盒操作，下一行PE中执行异或操作。另一种更好的方式是在一个PE中将S盒和异或串联组合设计，这样就可以在一个PE中完成两个操作，有效减少资源消耗。

对于算法中出现的每一个算子，发掘每一类算子的前驱、后继算子的种类和出现频率对于架构设计是十分有必要的，它为PE的组合功能设计提供依据。当架构的PE中有大量合理的组合功能存在，在算法实现时，很少的PE阵列规模就可以实现原来需要很大规模才能实现的算法，提高效率。

* 位置特征

位置特征是指算子在算法轮函数中的出现的位置；比如置换操作，它一般只出现在轮函数的开始或结束的位置完成初始置换和终结置换，因此在架构设计时只需要在映射轮函数的开始和结束的位置存在置换单元。位置特征主要针对大算子，这里的“大”是指这些算子对应的电路面积大，如所示。因此这些大算子对应的功能单元往往是阵列面积的主要部分，比如S盒操作，它一般会占据阵列总面积的一半以上，减少这类复杂单元的冗余数量对于降低阵列面积是十分有效的。

表3-2文献[1]中的部分功能单元面积对比

|  |  |  |  |
| --- | --- | --- | --- |
| 功能单元 | 面积µm2 | 等效门数 | 相对值 |
| 移位单元 | 19286 | 1929 | 1.0 |
| 模加/减单元 | 12826 | 1283 | 0.7 |
| 逻辑运算 | 3426 | 343 | 0.2 |
| 有限域乘法 | 85473 | 8547 | 4.4 |
| 置换单元 | 49816 | 4982 | 2.6 |
| S盒 | 1160372 | 116037 | 60.2 |

算法的参数特征、组合特征、位置特征分别为架构设计中的功能单元设计、组合单元设计和复杂单元在阵列中的位置确定提供依据。参数特征只是为了保证架构中的PE能够提供算法中算子所需要的功能，同时也避免在PE中添加不必要的功能。组合特征对应的组合功能是降低阵列规模的一个有效方法，很多通用可重构中，一个PE往往只会完成一个算子，这种设计虽然简单，但是在完成同等运算的情况下，PE的规模会很大，对应的存储、互连、配置等都会变得更加复杂。位置特征是从阵列的面积占比出发，针对复杂功能单元的优化，如表3-2所示，在密码算法中这样的复杂功能单元主要有S盒、置换、有限域乘法。这些功能单元的电路面积很大，这些功能单元的冗余造成不必要的阵列面积开销。对于简单的功能单元虽然也有优化的空间，但是往往没有什么明显的效果，因此本文在优化上只针对S盒、置换、有限域乘法这三类复杂功能单元。

[1]姜晶菲.可重牧密码处理结构的研究与设计[D]。长沙；国防防科学技术大学，2004；

* + 1. 参数特征

参数特征决定了阵列的一些基本属性，比如阵列大小、PE的位宽、PE内部功能单元的种类等。这些参数特征在不同的分组密码算法中都表现出相同或相似性，密码算法的这些共性也是可重构密码处理器的基本理论依据。

文献[1]对DES、IDEA、AES候选算法等41种公开的分组密码的基本运算操作进行了统计，结果如表3-3所示。从表中可以发现，对于这些分组密码算法，它们的操作只有有限的几种，这意味着PE设计时，只需要对这几类基本运算进行支持就足以实现这些算法。

表3-3 基本运算操作及其使用频率表

|  |  |  |  |
| --- | --- | --- | --- |
| 基本运算操作 | 使用频率 | 基本运算操作 | 使用频率 |
| 逻辑运算 | 97.56% | 算术运算 | 48.78% |
| S盒 | 73.17% | 置换运算 | 24.39% |
| 移位运算 | 85.36% | 有限域乘法运算 | 17.07% |

虽然算法有着相同的基本运算操作，但是不同的算法在同一个运算上的要求是不一样的，这些基本运算有着不同的运算模式，在进行架构设计时需要支持具体的运算模式。通过遍历目标算法集合中算法图的每一个节点，查询节点的运算功能属性和对应的功能参数，统计不同算子的使用频率和对应的参数特征，具体如表3-4所示。

表3-4 基本运算操作的参数特征

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 基本运算操作 | | 运算位宽 | 对应算法个数 | 运算模式 |
| 逻辑操作 | 异或 | 32 | 36 | 位宽兼容 |
| 非 |  | 0 |
| 或 | 32 | 1 |
| 与 | 32 | 3 |
| 移位 | 移位 | 32 | 4 | 左右移位 |
| 循环移位 | 8 | 3 |
| 32 | 11 |
| 模运算 | 模加 | 16 | 2 | 模216 |
| 32 | 14 | 模232 |
| 模减 | 32 | 1 | 模232 |
| 模乘 | 16 | 1 | 模(216-1) |
| 32 | 1 | 模232 |
| S盒 | |  | 10 | 输入-输出  4-4 6-4 8-8  8-32 6-2 10-8 |
| 有限域乘法 | | 32 | 10 | GF(28) |
| 置换 | |  |  | 输入-输出  64-64 32-48 32-32 32-40  128-128 |

通过表3-4可以发现分组密码算法的基本算子具有如下参数特征：

1. 分组密码算法中虽然大量使用逻辑操作，但是一般只会用到异或操作，只有少量的算法会使用其它逻辑操作（与、或、非）。
2. 分组密码算法中大量使用了逻辑移位和循环移位，其移位模式既有固定移位模式（每次移位位数固定不变）又有可变移位模式（每次移位数依赖寄存器的值），移位的位数一般从1位到32位，数据位宽大多为8、32。
3. 分组密码算法涉及的算术运算（乘、加/减）和逻辑操作（异或、与、或）位宽大多是字节或字节的整数倍，且算术运算大多带有取模操作，取摸操作都是2的幂次，多为28、216、232。
4. 表的S盒部分的运算模式中列出了几种不同的S盒的输入输出模式，常见的有6-4 S盒、4-4 S盒、8-8S盒、8-32 S盒、6-2 S盒、10-8 S盒；其中出现最多的是8-8的S盒。而且根据变化程度和使用方式的不同，分组密码使用的S盒可分为两种方式：一种是每轮操作使用相同的S盒，如DES、AES、Blowfish等；另一种是每轮操作使用不同的S盒，如SERPENT。
5. 置换操作的位宽会比较大，因为更大的数据位宽能有效扩展输出对输入的依赖性。置换操作的位宽会出现64、48比特，甚至128比特的置换在分组密码设计中也较为常见，但是一般128比特的置换都是基于字节的置换，而基于比特置换的位宽最大的只有64比特。
6. 分组密码算法中出现的有限域乘法主要集中在有限域GF(28)上，而且一般都是以矩阵乘法的形式出现。
   * 1. 组合特征

参数特征关注的是单个功能单元的各个参数，组合特征则关注多个功能之间的关系特征，涉及到整个算法的运算流程。本章的3.1小节中对算法进行了建模，其中算法有向图的AOE表示被用来分析算法的组合特征。在进行算子组合特征分析时首先需要对提取算法流程中的关键路径，关键路径上的算子组合本文的关注点。如图3-3所示，v2可以和v3组合成（v2，v3），也可以和v4组合成（v2，v4），但是在设计时只能选择一种进行实现。其实从DFG图中很容易发现（v2，v3）组合设计是一个有意义的组合，因为含有这个组合的路径更长，假设其它操作都不进行组合，那么（v2，v3）组合设计可以将整个DFG流程映射到5行PE阵列上。但如果（v2，v4）组合，没有改变长路径，映射依然需要6行PE。



图3-3组合选择冲突示例

从图3-3中的示例可以发现，有效的组合必须是关键路径上的组合，只有将关键路径上的某些操作组合才能在映射算法时有效降低映射所需的资源规模。因此需要找出算法流程中的关键路径，从关键路径上提取有效的操作组合。在图的AOE表示中可以很容易地通过正逆两次拓扑排序确定关键路径。

定义Ve[i]为每一个点的最早发生时间；Vl[i]为每一个点的最晚发生开始时间；整个过程如下：

1. 拓扑排序，并求得ve[]：从源点V0出发，令Ve[0]=0，按拓扑有序求其余各顶点的最早发生时间Ve[i]。如果得到的拓扑有序序列中顶点个数小于网中顶点数n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤2。
2. 拓扑逆序，求得Vl[]：从汇点Vn出发，令Vl[n-1] = Ve[n-1]，按逆拓扑有序求其余各顶点的最迟发生时间Vl[i]。
3. 比较Ve[]和Vl[]，求得关键路径：根据各顶点的Ve[i]和Vl[i]值，求每条边的最早开始时间e(s)和最迟开始时间l(s)。若某条弧满足条件e(s) = l(s)，则为关键活动。其中：e(i) = ve(j)，l(i) = vl(k) - weight(<j,k>)

获取关键路径后，对关键路径上的所有的算子进行临近组合，每一个有效组合作为一个组合元，所有的组合元构成有效组合的集合，遍历所有算法的关键路径统计这些组合元出现的频度。表3-5列出了部分算子的前后缀组合特征，其中SH表示移位算子，MAS表示算术算子，LUT表示S盒算子，GFM表示有限域乘法算子，PER表示置换算子，BR为按字节循环移位算子。

表3-5列出了算子的前后缀的2、3级组合中出现概率比较高的几种组合，从表中可以发现：在后缀组合中，算子后面跟异或逻辑算子的概率很高，而且具有一般性；在移位算子和算术算子中，移位算子-算术算子、算子算子-移位算子的两两组合虽然没有与异或逻辑算子组合的概率高，但也有相当的概率；S盒算子、有限域乘法算子、置换算子后面一般接异或逻辑算子，与其它算子组合的概率不高，而且在异或操作的后面接字节循环移位算子，组成概率相当高的3级组合。对于前缀组合，特征与后缀组合比较相似，大部分算子的前缀算子都是异或逻辑算子，一个比较有用的前缀特征是有限域乘法的前缀最大概率算子是S盒算子。第四章在确定具体的功能单元组合结构时会详细讨论这些算子组合特征与对应功能单元串行组合之间的关系。

表3-5 部分算子组合特征

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算子 | 后缀组合 | | | | 前缀组合 | | | |
| 2级组合 | 概率 | 3级组合 | 概率 | 2级组合 | 概率 | 3级组合 | 概率 |
| SH | SH-XOR | 47% | SH-XOR-MAS | 20% | XOR-SH | 67% | MAS-XOR-SH | 27% |
| SH-MAS | 33% | SH-XOR-SH | 13% | MAS-SH | 27% | XOR-LUT-SH | 20% |
| SH-AND | 13% | SH-MAS-XOR | 13% | LUT-SH | 27% | XOR-XOR-SH | 13% |
| MAS | MAS-XOR | 75% | MAS-MAS-XOR | 31% | XOR-MAS | 56% | XOR-SH-MAS | 25% |
| MAS-MAS | 31% | MAS-XOR-XOR | 25% | SH-MAS | 31% | MAS-XOR-MAS | 25% |
| MAS-SH | 25% | MAS-XOR-SH | 25% | MAS-MAS | 31% | XOR-MAS-MAS | 19% |
| LUT | LUT-XOR | 37% | LUT-XOR-BR | 19% | XOR-LUT | 78% | XOR-XOR-LUT | 26% |
| LUT-BR | 32% | LUT-SH-XOR | 11% | MAS-LUT | 11% | XOR-MAS-LUT | 11% |
| LUT-SH | 7% | LUT-XOR-SH | 7% | SH-LUT | 7% | / | / |
| GFM | GFM-XOR | 70% | GFM-XOR-BR | 56% | LUT-GFM | 60% | XOR-LUT-GFM | 50% |
| GFM-BR | 23% | GFM-XOR-XOR | 10% | XOR-GFM | 10% | / | / |
| PER | PER-XOR | 83% | PER-XOR-BR | 17% | XOR-PER | 67% | XOR-BR-PER | 17% |

* + 1. 位置特征

位置特征指算子在算法轮函数中的出现的位置统计特征，位置特征主要针对大算子，这类算子不会在算法中大量存在，同时这类算子在架构中的数量直接决定了架构的电路面积。因此进一步挖掘这类算子的位置特征，在架构设计时，根据位置特征，在需要的位置放置这些算子对应的功能单元，可以有效低降低整体面积。

算法中这类大算子主要有S盒算子、置换算法和有限域乘法算子。在论文2.2节中介绍了整个架构的阵列是以异构组的形式扩展的，这个异构组对应算法的轮函数，因此这些算子在轮函数中的位置对应着在异构组中的位置，因此可以通过提取这些算子在轮函数中的位置特征来决定这些算子对应的功能单元在异构组中的关系。

在算法建模时算法中的每一个算子都有与位置相关的信息（x，y），相当于一个算子位置的坐标系，提取上述三种算子在坐标系中的位置特征。

表3-6置换操作在轮函数中的位置特征

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | | Des | Present | Square | ICE | Grand\_Cru | Q |
| 置换操作出现位置 | 轮函数初始 | **√** |  | **√** | **√** | **√** | **√** |
| 轮函数终结 | **√** | **√** |  | **√** |  |  |

如表3-6对于置换算子，对应的6种算法中，轮函数的初始和结束位置中出现置换算子，其它位置不存在置换算子。

在目标算法集合中有10个算法中含有有限域乘法算子：AES、TWOFISH、CLEFIA、ARIA、SQUARE、SHARK、GRAND\_CRU、KHAZAD、HIEROCRYPT\_L1和HIEROCRYPT\_3，所有的有限域乘法算子都出现在了这些算法轮函数的终结位置，并且在这十个算法中有7个算法的有限域乘法算子与S盒算子组合出现，有限域乘法算子紧跟在S盒算子的后面出现。

目标算法集合中含有S盒算子的算法占了75%，S算子作为算法中的非线性变换一般出现中轮函数的中间位置，S盒算子的前后都会有其它操作，这与置换算子和有限域乘法算子差异很大。

[1] Elbirt, Adam J. "Reconfigurable computing for symmetric-key algorithms." (2002).

1. PE设计方案

本章将在算法特征分析的基础上，对架构中PE设计提出一个初始的方案。这个初始方案确定了可重构密码处理器的执行阵列中异构组的大小，确定了异构组中每一个位置异构PE的种类以及这些不同种类PE的结构组成。根据S盒替换、置换、有限域乘法这三种操作在算法特征分析中表现出来的明显特征，初始方案中确定了S盒替换、置换、有限域乘法这三种运算单元的个数以及在异构组中的位置，这个结果是会在第五章的映射分析中进行验证。对于加法、移位、逻辑等操作，统计特征不明显，在初始架构中的每一个PE中都包含了这些单元，进一步的优化会在第五章的映射分析之后进行。章节的最后对这个初始PE方案进行了基于图的建模，这个图模型是第五章中映射算法的目标输入。

* 1. 设计方法

对于一个通用的可重构密码处理器，密码算法的统计特征决定了处理器的设计特征，比如处理器位宽、功能单元、互连方式等特征与算法的数据分组大小、算子种类、数据加密流程等密切相关。第三章中对算法集合中的分组密码算法进行了建模分析，算法的统计特征成为本章中PE方案的依据。

课题致力于在保证架构的算法支持和映射性能的前提下尽可能地减少架构的设计面积。算法的统计分析表明传统的可重构密码架构中存在很多单元冗余设计。在传统的同构设计中，在架构的任何位置的PE都包含了算法所需的全部功能；但是对于算法而言，某一种功能单元只会出现有限的几次，架构中这些功能单元的使用率很低，甚至有些功能单元根本不会被使用。本文从两个方面避免这种功能单元冗余；首先在确定初始架构时对于位置信息比较明显的功能单元，只在有明确需求的位置上放置这些功能单元；其次对于那些位置特征不明显的功能单元，在制定初始架构方案时不进行确定，如传统的同构架构一样，所有的PE都包含这些功能单元，进一步的删除在进行算法映射后进行，算法映射统计结果为进一步删除这些位置信息不明显的功能单元提供依据。

* 1. 阵列拓扑结构

如图4-1所示，可重构密码处理器阵列由多个异构组重复迭代组成。每3个PE行和3个互连结构组成1个异构组，每1个PE行中包含4个PE。

对于87.2%的加密算法来说，4路并行的数据通路就可以是算法达到最大性能，只有少数的加密算法需要8路或16路的并行PE。在架构设计中，并行PE数的增加意味着阵列规模的线性增加；对于8路或16路的阵列，在大多数算法下，整个阵列的硬件利用率将非常低，因此架构中采用4路并行PE的设计方案。



图4-1可重构密码架构阵列拓扑图

所有的分组密码算法都有一个共同的特征，算法的加解密过程都是由多个相同的轮函数迭代完成的，因此真正表征算法的特征的是这个轮函数。对于同构架构，所有的PE中都被堆砌了算法所需要的功能单元，因此它可以不用考虑算法中功能单元的位置信息。对于异构架构来说，理想的映射方式是一个算法轮函数映射到阵列的一个异构组中，阵列以异构组重复迭代单元，对应于算法的轮函数迭代。不多轮函数较复杂，无法再一个异构组中完成，那么只能用一个新的异构组去完成余下的部分，这种情况看起来比较糟糕，但是实际上只要对异构组进行合理实际就可以把大多数算法约束到一个异构组中完成。

* 1. 行间互连

可重构阵列行单元主要完成连续PE行的数据互连。如图4-2所示是所用互联单元的结构图，整个互联单元由两个部分组成：

第一层主要实现数据的全互联选择，由16个20输入MUX构成，每一个MUX的20输入分别为上一个PE行的输出（12个）、通用寄存器堆（General Purpose Register File，GPRF）（4个）和外部输入接口（First In First Out，FIFO）（4个）。每一个20输入MUX需要5bit配置信息，第一层总共需要80bit配置信息。

第二层实现4个字节数据按字节重组，由16个字节移位器构成。每一个字节移位器的输入为第一层中某一个MUX的输出。每一个字节移位器需要2bit配置信息，第二层总共需要32bit配置信息。



图4-2互联单元结构图

* 1. 异构组

异构组重复迭代构成处理单元阵列，是整个架构设计的核心，如图4-3所示，每个异构组中包含3个互连行和3个PE行。3个PE行共有12个PE，这12个PE分成5个不同的种类。



图4-3异构组拓扑图

在确定初始PE方案时，根据算法中S盒替换、置换、有限域乘法这三种操作表现出来的统计特征，对对应的功能单元进行了位置裁剪。5类异构处理单元的不同表现在对S盒替换、置换、有限域乘法三类处理单元的包含上，具体见表4-1所示。

表4-1 5类异构处理单元对比

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 处理单元类别 | | PE\_Type1 | PE\_Type2 | PE\_Type3 | PE\_Type4 | PE\_Type5 |
| 包含操作 | 逻辑单元 | **√** | **√** | **√** | **√** | **√** |
| 算术单元 | **√** | **√** | **√** | **√** | **√** |
| 移位单元 | **√** | **√** | **√** | **√** | **√** |
| 置换单元 | **√** |  |  | **√** |  |
| S盒子替换单元 |  |  | **√** |  |  |
| 有限域乘法单元 |  |  |  | **√** | **√** |

表4-1可以看出，5类异构PE中都包含了逻辑、算术、移位操作单元；PE\_Type1、PE\_Type4中包含了置换单元，PE\_Type3包含了S盒替代单元，PE\_Type4、PE\_Type5包含了有限域乘法单元。

算法集合中有6个算法的轮函数中有置换操作，这些置换操作表现出了很一致的位置特征，置换只出现在轮函数的开始或结束的位置，如表3-6所示，而且出现最大的比特级置换是64位，如表3-4所示。因此置换单元支持最大64bit置换。架构支持的最大数据位宽是128位，因此最多只会并行使用两个置换单元，因此确定在异构组的第1、3行的第1、2个PE中包含置换单元。

S盒替换单元是整个架构中的重要单元，75%的分组密码算法使用S盒替换操作来完成算法中的非线性变换，而且S盒替换单元的面积开销占据了整个架构的相当部分，因此很有必要对S盒替换单元进行位置优化。算法的统计特征表明，S盒替换操作一般出现的位置是轮函数的中间位置，77.78%的算法在查表前进行了秘钥加操作；查表后则主要进行移位、字节变换、有限域乘法等操作；没有一个算法将S盒替换操作作为轮函数的开始或者结束。因此在异构组的第1、3行放置S盒查表单元是不合理的，异构组中只有第2行中含有S盒查表单元。

有限域乘法在AES算法中引入，随后出现的加密算法广泛使用该结构；包含有限乘法的算法有AES、TWOFISH、CLEFIA 、ARIA、SQUARE、SHARK 、GRAND\_CRU、KHAZAD、HIEROCRYPT\_L1、HIEROCRYPT\_3。在这些算法中有限域乘法都紧跟在S盒查表操作之后，因此有限域乘法应该出现在紧跟在S盒查表操作行之后，对应于异构组中的第3行。

如表3-5所示，在轮函数中，各类处理单元的前后出现概率最高的操作是异或操作，因此除了逻辑操作单元本身除外，其它的处理单元的前后都串行连接了异或操作，这样可以有效减少算法映射时所需的行数。对于移位算子和算术算子之间的组合，虽然概率也比较高，但是这两个算子对应功能单元的组合设计会成为处理单元的关键路径，虽然这种组合可以在一定程度上提高阵列的处理效率到也损害了阵列性能，因此本设计忽略这两种算子的组合设计。

在3级组合中，一个有效的组合是按字节循环移位算子与其它算子构成的组合，在表3-5中可以发现，这类组合出现的频率很高，几乎在所有的大算子结束位置都有概率出现按字节循环移位算子。在本文的设计中，按字节循环移位功能被放置在行间互连的第二层，如图4-2所示，这种设计中每一个PE都可以与按字节循环移位进行组合。

* 1. 处理单元

处理单元是架构中的最小执行单元，它包含算法需求的各种功能单元：算术单元（AU）、移位单元（SH）、置换单元（PER）、S盒替换单元（LUT）、有限域乘法单元（GFM）、逻辑单元（LOU）以及这些单元串联的异或操作。



图xxx 带全功能的处理单元

图xxx是一个带全功能的处理单元，根据4.1.4中对异构组的分析可知，异构组中共有五类不同的执行单元，这些执行单元在置换单元（PER）、S盒替换单元（LUT）和有限域乘法单元（GFM）的包含上会有所差异。从这个全功能的处理单元中删除某些功能单元就可以得到异构组中的五类处理单元。为了易于描述，下面会描述这个全功能的处理单元。

处理单元中有5个32位的输入：in0、in1、in2、in3、Reg\_in0；其中in0、in1、in2、in3这4个输入来自上一行的互连，传输上一行处理后的数据；Reg\_in0来自寄存器堆，主要用来传输寄存的中间数据以及算法的秘钥。处理单元中有4串联的数据通路最多需要四个输入，因此用4个2输入mux从5个输入中选出4个待处理数据。

处理单元有两个输出out0和out1，输出到互连单元作为下一行的输入；异构组的第三行处理单元有一个额外的输出Reg\_out0，输出到寄存器堆，用来寄存运算中间数据。每一个输出数据可以来自处理单元并行通路中任何一个，因此在输出端有2个6输入mux来选择输出。

*表xxx 处理单元控制信号分配*

|  |  |  |
| --- | --- | --- |
| *控制信号* | *位数* | *对应功能单元* |
| *control[xx:xx]* | *x* | *算术单元* |
| *control[xx:xx]* | *x* | *逻辑单元* |
| *control[xx:xx]* | *x* | *移位单元* |
| *control[xx:xx]* | *x* | *置换单元* |
| *control[xx:xx]* | *x* | *S盒替换单元* |
| *control[xx:xx]* | *x* | *有限域乘法单元* |

*对处理单元的映射分配做说明*

* 1. 功能单元

处理单元中有6条独立的数据通路，这6条数据通路都对应着不同的功能单元：算术单元（AU）、移位单元（SH）、置换单元（PER）、S盒替换单元（LUT）、有限域乘法单元（GFM）、逻辑单元（LOU）。接下来分别介绍这这些功能单元的结构。

* + 1. 逻辑单元

对算法分析获取的逻辑操作使用特征如表xxx所示；所有的算法都使用了异或操作，与操作和或操作只有很少数的算法使用，没有算法使用到非操作。

表xxx 逻辑操作统计信息

|  |  |
| --- | --- |
| 逻辑操作 | 使用频率 |
| XOR | 100.00% |
| NOT | 0.00% |
| OR | 2.80% |
| AND | 8.30% |
| XOR->XOR | 30.56% |
| XOR->XOR->XOR | 17.00% |

对于逻辑操作的级联，有30.56%的算法中含有双异或操作，有17%的算法中含有三异或操作；为了支持多异或操作，如图xxx所示，逻辑单元中对异或操作进行了可选择的三级级联。



图xxx 逻辑单元结构

如图xxx所示，结构中共有7路数据，因此需要一个8输入mux，共需要3位配置位。对应的配置信息和操作功能如表xxx所示。

表xxx 逻辑单元的配置信息分配

|  |  |
| --- | --- |
| 控制信号 | 操作 |
| 000 | ir0⊕ir1 |
| 001 | ir0⊕ir1⊕ir2 |
| 010 | ir0⊕ir1⊕ir2⊕ir3 |
| 011 | ir0|ir1 |
| 100 | ir0&ir1 |
| 101 | Ir2直通 |
| 110 | Ir3直通 |

* + 1. S盒替代单元
    2. 算术单元

算法分析表明，有47.22%的算法使用了算术单元，其中有88.2%是32位运算，对应模运算，11.2%是16位运算，对应模运算。因此在进行结构设计时同时兼容了8、16和32三种位宽的运算需求。如图xxx所示，算术单元可以根据算法需求配置成模、模和模运算。算术运算前后出现异或操作的概率分别是56%和75%，因此与其它功能单元一样，算术单元的前后都串联了异或操作。



图xxx 算术单元结构

算术单元中有3个2输入mux，需要3位配置位，AU需要两位配置位；共需要5位配置信息。

表xxx 算术单元的配置信息分配（部分）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 控制信号 | | | | 操作 |
| AU | Mux2 | Mux1 | Mux0 |
| 00 | 0 | 0 | 0 | (ir0+ir2) modulo |
| 00 | 0 | 0 | 1 | ((ir0⊕ir1)+ir2) modulo |
| 00 | 0 | 1 | 1 | ((ir0⊕ir1)+ (ir1⊕ir2)) modulo |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

* + 1. 置换单元

比特置换被DES、PRESENT等算法用来完成非线性变换，同时还有组合，扩展的功能。输入为2个32位数据，输出为2个32为数据或1个64位数据。置换还可以完成某些不容易实现的操作，基于比特的置换操作可以轻松完成很多不容易实现的复杂逻辑操作。



图xxx 置换单元

置换单元由一个64位的BENES（无阻塞）网络和4组异或逻辑构成，每一组异或逻辑还对应一组2输入mux。如式xxx所示，64位BENES所需的配置位为352位，因此整个置换单元共需352+4=356位配置信息。

BENES (N).length = (N/2)\*(2\*log2 (N)-1)

* + 1. 移位单元

在分组密码算法中使用最多的移位操作是32比特的移位和循环移位，占总共的83.3%。只有16.7%的算法的移位运算时8比特的，因此课题中提出的移位只支持32比特的操作，少量使用到8比特移位运算的算法可以使用置换单元来代替实现。

如图xxx所示，移位单元输入两个32比特的数据，其中一个是操作数，一个是移位比特位数。移位单元支持左移位、右移位、左循环移位和右循环移位。



图xxx 移位单元

整个数据通路需要4比特的控制信息，具体的含义如下表所示。

表xxx 移位单元控制信息(部分)

|  |  |  |  |
| --- | --- | --- | --- |
| 控制信号 | | | 操作 |
| mux1 | SH | mux2 |
| 0 | 00 | 0 | ir0<<ir2 |
| 0 | 00 | 1 | (ir0⊕ir1)<<ir2 |
| 0 | 01 | 1 | (ir0⊕ir1)>>ir2 |
| 0 | 11 | 1 | (ir0⊕ir1)>>>ir2 |
| 1 | 11 | 1 | ((ir0⊕ir1)>>>ir2)⊕ir3 |

* + 1. 有限域乘法单元

有限域乘在分组密码算法中的出现频率比较高，比较常见的使用了有限域乘法的算法有AES、TWOFISH、CLEFIA、ARIA、SQUARE、SHARK等。这些算法中出现的运算都是在域上，因此本文中的有限域乘法只支持上的乘法运算。

* 有限域上的乘法

在上，我们设f(x)是不可约多项式，则f(x)被称为域多项式，中的元素可以看作是小于n次的多项式。设上的两个元素为：

定义两者的乘积为：

其中：

根据乘法定义，上的乘法本质上是多项式中对应的系数进行运算，目前实行的基本方法是采用“Shift-and-add”算法[xx,xx]。算法执行多次移位，且加法是串行操作。算法实质上是根据乘数，将被乘数不断移位后，将每个结果作模2加，得到原始乘积，再进行模运算，就可以得到最终乘法结果。在现实中，移位分为左移与右移两种方式，并且通常采用边移位边做模运算的方式进行。使用向左移的方式时，根据乘数的第i位的值来确定移位的位数，若第i位上的数为1，那么被乘数左移i位；若第i位上的数为0，那么被乘数不运算。由上述分析可知，利用逻辑左移可以很容易实现被乘数移位，同时对每次的移位结果进行模运算，最后将每个结果模2加得到最终结果。“Shift-and-add”算法描述如图xxx所示：

根据有限域的性质以及移位相加“Shift-and-add”算法，对于具有任意不可约多项式的有限域上的乘法，假设多项式x乘以a(x)有：

(xxx)

由上式（xxx）可得：

(xxx)

|  |
| --- |
| Shift-and-add 算法描述 |
| Input：二元多项式a(x)、b(x)，最高次项系数最多为(n-1)  Output：c(x)=a(x)b(x) mod f(x)  (1)c<-0  (2)for i=n-1 to 1 do  if =1 then c<-c+b  c=cx mod f(x)  (3)if =1 then c<-c+b  return (c) |

图xxx Shift-and-add 算法描述

由式（xxx）可获得任意不可约多项式的域的乘法电路。将x乘法电路依次串联起来，便可得到域上任意不可约多项式的基本乘法运算电路。将这样的多个基本乘法运算电路并联以及异或操作，便组成了乘法矩阵运算电路[xx]。

* 电路结构

域上的乘法单元电路结构如图xxx所示。电路分为两部分，分别是域矩阵乘法电路和静态配置寄存器。域矩阵乘法电路模块负责完成域上的矩阵乘法运算工作。静态配置寄存器的功能是完成域上乘数多项式及不可约多项式信息的存储工作，这也是图中静态配置寄存器的作用。

分组密码算法实际应用中，域上的乘法通常会使用在矩阵乘法形式表示，乘数多项式以及不可约多项式是较为固定的。因此，对这两种运算参数采取静态配置的形式读入到有限域乘法运算电路中。对分组算法的分析发现，在一个4X4的矩阵中，乘数为4x4x8=128比特，不可约多项式信息则为8比特。架构中有四条通路，因此静态配置寄存器大小为4x136比特。



图xxx 域乘法电路整体电路图

根据公式xxx可以得到对于任意不可约多项式域上的x乘法电路。如图xxx所示。



图xxx 域x乘法电路

（f7,…,f1,f0）是不可约多项式的低8位，将图xxx描述的电路定义为xtime，那么将7个xtime电路依次串联起来，再将结果进行三级异或，就可以得到如图xxx所示的域上任意不可约多项式的乘法运算电路。称它为基本有限域乘法电路，记为GF28Mult。



图xxx 基本有限域乘法电路

f[7:0]即为xtime电路中的（f7,…,f1,f0），是不可约多项式的低8位，a[7:0]和b[7:0]表示域上的两个相乘的多项式。由电路结构图可以发现，a、b和f都是可以实时输入的，因此该电路结构可以完成域的任意不可约多项式的乘法。

在分组密码算法中通常将所有的8比特的乘数多项式以固定的矩阵方式给出，表示为矩阵乘法的形式。因此，如图xxx所示，四个基本有限域乘法电路并联，输出结果再进行异或操作，这样便得到了矩阵乘法中l行×l列的8比特效果。迸一步，将该电路进行4路并联，输入128比特的乘数多项式、32比特的运算数据以及8比特的不可约多项式便可以完成完整的矩阵乘法运算。



图xxx 有限域矩阵乘法电路(1行x1列)

* 1. 本章小结

1. 算法映射分析与PE方案优化
   1. 可重构架构映射概述

应用映射是可重构设计的一个主要难题，在可重构架构上处理数据密集型应用要求较高的吞吐量和并行性，采用手动的方法将应用映射到可重构架构上不仅费时并且容易出错，当算法集合增大时，大量的映射变得非常繁琐。手工映射需要映射人员对架构有清晰的认识，不容易掌握，而且在架构进行微调时，所有的映射要重新进行，出现大量的重复性工作。另一方面，拥有高效的自动化映射工具允许架构设计人员对架构进行大规模的应用样本映射测试，根据映射结果反馈，对架构进行优化调整。因而设计高效的自动化工具是可重构架构设计的一个关键问题。

* + 1. 问题模型

应用可用数据流图（Data Flow Graph，DFG）表示，在DFG中每个节点表示应用中的某一个运算操作，有向边表示两个操作间的数据依赖关系和数据流向。应用到可重构阵列上的映射包括三个部分：DFG中的节点到可重构阵列上的PE的映射；DFG中的有向边到PE间互连的映射；数据到局部存储的映射。图xxx给出了一个应用到可重构阵列上的映射，图中的R表示该PE不进行运算，作为路由使用。



图5- 1

通常将应用映射到可重构阵列上需要考虑体系结构上的几点制约因素：

* 阵列

阵列的规模，即阵列上的资源数量，直接决定应用操作的并行度和吞吐量，较大的应用映射到资源数较少的阵列上时，需要根据资源数量对应用算法进行划分，在不同划分里的运算需要对可重构阵列进行配置切换才能继续执行。另一个关键因素是阵列的拓扑结构，包括阵列的横向执行单元的个数，这决定了应用可以在阵列上映射的数据宽度；阵列的纵向深度，这在应用进行流水展开时决定了可以映射的流水级数。

* 处理单元

每一个都有多个功能单元资源，功能资源包括单操作的功能单元例如移位器和支持多种操作的功能单元如一些功能单元组合操作。在异构的可重构阵列中，一些面积开销大的功能单元如S盒替代单元只存在阵列中的部分处理单元中，因而不是所有的操作都能映射到任意的处理单元上，部分操作只能由特定的处理单元完成。

* 局部存储器

通常上多个通过总线来共享一个局部存储器，总线的带宽有限，不能在一个周期来完成所有PE的数据获取，因此在映射时要考虑这种总线带宽限制。

* 互连网络

可重构阵列中PE间的互连资源是有限的，不是所有的PE间都可以直接互连；在某些时候还需要将某一个PE作为路由使用。互连网络结构是应用映射时通信路径选择的主要约束因素。

本文提出的可重构密码处理器上应用映射的特点

1. 可重构密码处理器阵列的互连是有导向的，数据从阵列的第一行输入，第一行的输出结果给第二行，数据经过整个阵列从最后一行输出。因此在映射时从阵列的第一行出发，数据流向和阵列的互连导向一致。
2. 可重构密码处理器阵列上的映射属于空间映射，密码算法会被循环展开，在架构上进行流水展开映射。
3. 行间的互连不是硬连线，而是数据选择。可以选择某条路径到达下一行的任何位置，但不可以同时到达所有的位置。也就是从某一个点出发的所有连线是互斥的，不可约同时存在，每一次只能互连到一个位置。
4. PE内部功能更复杂，不只是单个运算到单个PE的映；PE内部有很多组合功能设计，应用中的多个组合的功能会被同时映射到一个PE中。
5. 单个PE的输入不唯一、功能不唯一，也就是说在输入允许的情况下，横向上并行的多个功能可以被映射到同一个PE。
6. 可重构密码处理器阵列被设计成行间异构、列间异构。这样的不一致性结构在架构描述、映射算法上都有特异性。

决定方法的变化

* + 1. 研究现状

将循环映射到时，根据映射目标架构的不同，有两种常用的映射方法：时间映射（temporal mapping）和空间映射（spatial mapping）。

时间映射适用于处理单元只有一个或者一行的可重构处理器的架构。时间映射的方法将整个循环体映射到一个PE或一行PE上，每PE个或每行执行不同迭代次数的循环。PE要顺序执行多个操作，每执行完循环的一次操作就要进行一次重构，动态配置下一级的操作功能。时间映射方法的特点有：

* 由于循环的所有操作都在一个PE或一行内完成，不需要考虑PE间的数据互连；
* 可以直接利用传统的编程编译方法完成循环的映射；

这类CGRA映射问题[ V1-V3]的方案来自于VLIW架构的编译技术，它利用了VLIW中的时间流水模调度算法和VLIW中的存储共享特征。

空间映射的方法适用于可重构阵列架构，它将循环中的每个操作一次性映射到CGRA的PE阵列上，每个PE只绑定循环中的一个操作，只执行一种固定的操作，整个循环执行的过程中只用配置一次，不需要在经过重新的配置执行其他操作。空间映射的方法有以下特点：

* 能够充分利用CGRA计算资源和并行运算的能力，并行执行循环体的多次循环；
* PE只需要完成一个固定的操作，重构的开销小，整个循环的过程不需要进行重构；
* 操作节点的布局要充分考虑各个操作间的数据依赖关系和CGRA的互连资源，布局复杂，并且要使用较多的互连资源；

这类CGRA映射问题的研究更为广泛，最开始的方案灵感来自FPGA[v4-v5]综合里的布局布线技术，但是CGRA本身和FPGA有很大不同的，具体来说，CGRA中的互连线大部分是固定的，而FPGA中是可配置的；因此基于FPGA综合里的布局布线技术在固定的互连在寻找路由变得很困难。后来的一些映射方法主要来自于图论领域，[V6 ]利用子图同构算法获取DFG图到架构图之间的映射候选集；SPKM [V7] 引入分裂和外扩的方式 [V8] ，它将应用看成一个集合，每一次向外扩展一个点，余下的点作为新的扩展集合，一直到集合中的点全部外扩完毕。类似的图论方案还有如[V9]中的子图同胚[V10(134)]以及EPIMap[V10(56)]中的图满射技术。

本文中的可重构密码处理器是架构中包含PE阵列，因映射时将密码算法循环展开以流水的形式映射到阵列上，因此本文中的应用映射问题为空间映射。根据上小节总结的架构特征，本文选用适用性更好的子图同构算法作为映射的依据，子图同构算法能够在异构的不规整架构中找出所有的可能映射方案，然后再通过一定的映射规则找出资源使用最少的方案作为最终映射方案。

* 1. 基于子图同构的映射方案
     1. 子图同构基本概念
* 基本定义

**定义5.1（图）：**一个图是一个四元组，其中，

1. 称为G的顶点集，其元素称为顶点或结点；
2. 称为G的边集，其元素称为边；
3. 为图G的顶点标记函数，说明顶点与其标记的对应关系；
4. 为图G的边标记函数，说明边与其标记的对应关系。

**定义5.2（子图）：**一个图 为图的一个子图，记为，如果有：

1. ；
2. )；
3. ；
4. 。

此时，我们也称图为图的一个超图。

**定义5.3（图同构）：**图和图是同构的，记为，如果存在一个双射函数，使得：

* 1. ；
  2. 。

这样的函数也称为图与的图同构。例如，图xxx中A和B两个图同构。



图5- 2

图 xxx 图同构的示例

**定义5.4（子图同构）：**给定图与的一个图同构以及另一个图，如果，则为图与的一个子图同构。图xxx是一个子图同构的示例，S是G的一个子图，它包含了G中顶点集的一个子集，而且S中顶点的连接方式与G中相同，所以称S是G的一个同构的子图。



图5- 3

图 xxx 子图同构的示例

* 复杂度分析

在理想情况下，问题都可以在多项式时间（也称为P时间）内解决。这意味着对于解决一个含有n项的问题，存在c，z，g几个常量，算法T所需时间受以下公式下公式[Vxx]的限制：

其中常量z是我们真正感兴趣的，它通常被叫做“增长因子”。它表示当n增加时算法预期增多的额外时间的数量。例如当z =1，n加倍时，算法所需的时间也加倍。

但是子图同构被证明是一个NP完全问题，即在最坏情况下，判定两个图子图同构所需时间与图中所包含的节点数量成指数增长关系。也就是说，最坏情况下解决子图同构问题的时间复杂度为，其中 n 为规模较大的图中的节点个数。

虽然子图同构问题具有先天的复杂性，但是由于可重构密码处理器映射本身的一些特性可以使问题的复杂度降低。

1. 算法图和架构图都是有向图，而且有明确的起点和终点，这样架构图中的算法图候选集非常有限。
2. 架构图中没一个结点（PE）的输入输出很有限，而且结点只有相邻才有互连，因此图的边集很小，这也简化了算法的复杂度。
3. 在本架构中一共只有5类PE，而且边没有属性，降低了图标记的难度。
   * 1. VF2子图同构算法

图的同构判定是图论学科的基本同题之一，文献[x-x]对这个问题进行了充分的研究。文献[x]提出了Ullmann算法，它是一个可前向剪枝的带有回溯的树搜索过程；文献[x]提出的VF2算法同样基于搜索方法，它利用一个快速计算的启发式规则进行剪枝，这使其性能得到显著提升。

表5-1 Ullmann算法和VF2算法对比



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | VF2 | | Ullmann | |
| 复杂度 | 最好情况 | 最坏情况 | 最好情况 | 最坏情况 |
| 时间 | O(N2) | O(N!N) | O(N3) | O(N!N2) |
| 空间 | O(N) | O(N) | O(N3) | O(N3) |

VF2算法的核心思想是搜索加剪枝，重点就在于如何剪枝。状态s存储搜索过程中的部分匹配，以及算法需要的其它数据。代表中间状态，和表示当前状态s的部分匹配中图和图中的点，整个算法流程如图5-4所示。

|  |
| --- |
| 1) PROCEDURE Match(s)  2) 输入：中间状态，初始状态满足  3) 输出：两个图之间的所有匹配  4) IF M(s) 覆盖了的所有结点  5) 输出M(s)  6) ELSE  7) 计算可以被包含到M(s)中的候选点对集合P(s)  8) FOREACH P(s)中的点对p  9) IF 候选点对满足加入M(s)的所有约束规则  10) 将p加入M(s)，计算  11) CALL Match()  12) END IF  13) END FOREACH  14) 恢复数据状态  15) END IF  16) END PROCEDURE Match |

图5-4 VF2子图同构算法流程

初始化时状态是，是空集，即还没有任何匹配；之后递归的进行搜索。如果当前状态s代表的部分匹配包含了中的所有节点，则已经找到了在中同构的子图，搜索结束；否则，在当前的局部匹配基础上，再匹配一个点。找出所以可能进行匹配点对集合；对于每一个匹配对p，检查加入匹配p是否可行，即加入p后，两个图还是否同构，以及加入p之后，是否还有扩展的可能性；如果加入匹配p可行，则将p加入s，递归调用Match()，继续搜索。如果若干次调用Match()后都没有找到同构的子图，则说明当前状态不可能扩展出可行的子图同构匹配；所以将生成新状态时加入的两点匹配p从s中删除，回溯到上一个状态。

在上述算法流程中，新加入的匹配p，我们要检验其加入的可行性，从而对搜索空间进行剪枝，来提高算法的效率。

先约定几个符号：和表示图和图中的点集。n和m分别表示图和图中的点。表示点n在图G中的前驱，表示点n在图G中的后继。和表示状态s在图和图中，指向当前已经匹配的点集的所有边的源点集合。和表示状态s在图和图中，从当前已经匹配的点集出发的所有边的终点集合。，即当前状态s在图中已经匹配的点集的所有一步邻居。，即图中，除了s中已经匹配的点，和这些点的一步邻居以外的点。

新加入的点p需要同时满足五个规则：











和保证加入新的匹配对p后，两个子图仍然是同构的。设新加入的匹配对是(n,m)，则对于n在图中的所有前驱（或后继），必须能在图中m的前驱（或后继）里有相应的点与之对应。同样，对于m在图中的所有前驱（或后继），也必须能在图中n的前驱（或后继）里有相应的点与之对应。

、和都是剪枝策略。其中表示求集合中元素的个数。和表示n在（或）中的前驱（或后继）的数目，必须大于等于m在（或）中的前驱（或后继）的数目。如果不满足，则说明对于图中新匹配的点m，其邻居个数是大于图中n的邻居个数的，最终必然无法完全匹配图中所有的点。

跟和思想类似，只不过考虑的两步邻居。具体是，和中的考虑的邻居是和中的邻居。这些点既跟n相邻，又跟当前匹配中的其他点相邻。而考虑的邻居，是只跟n相邻，跟当前匹配中其他店不相邻的邻居。这样细粒度的考虑的好处是可以更细粒度的剪枝，从而提高剪枝效率。

的的地方的分段算法的搜索空间主要由候选点对集合P(s)决定，在没一个状态s下都对应不同的搜索空间。候选点对的规则为：

1. 如果和都不为空，则取这两个集合中的所有点两两组合，生成候选匹配对集合；
2. 如果和两个集合都为空，若和都不为空，则取这两个集合中的所有点两两组合，生成候选匹配对集合；
3. 如果上面四个集合都为空（对于非连通图会出现这种情况）。则只能找两个图中所有没有匹配的点两两组合，生成候选匹配对集合。

细粒度的分类讨论，可以尽量减少单次生成的候选匹配对的数量。否则如果每次都按上面第三种方式生成，则每次递归都会生成很多之前生成过的匹配对，造成重复计算。

Conte, Donatello, et al. "Thirty years of graph matching in pattern recognition."*International journal of pattern recognition and artificial intelligence* 18.03 (2004): 265-298.

Ullmann, Julian R. "An algorithm for subgraph isomorphism." *Journal of the ACM (JACM)* 23.1 (1976): 31-42.

Cordella, Luigi P., et al. "A (sub) graph isomorphism algorithm for matching large graphs." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26.10 (2004): 1367-1372.

* + 1. 架构图建模

上一小节中采用有向图对算法进行建模，利用图的节点描述算法中算子，图中的边描述数据依赖。完成算法的图建模后就可以利用图论算法来提取算法的某些关键特征为架构设计提供依据。在可重构架构中，同样可以对架构进行图建模，利用节点描述PE，边描述架构中的互连。这样算法和架构有了相同的基于图的描述方式，利用图论中的匹配算法可以完成算法到图的匹配映射[1][2]，第五章将会详细讨论算法在架构图上的映射验证问题。

**定义5.5（架构图）：**一个架构图是一个二元组ARG=(V,E)，其中V是节点集，表示架构中的PE；E是边集，表示架构中的数据互连。

和算法图相比，架构图中的节点具有更复杂的功能属性和位置属性，架构图中的边网络也更加庞大。

初始架构中定义了5类PE，因此架构图中的节点有五类不同的功能属性，在架构图中用字符串进行描述。如表5-2所示，PE\_type1的功能属性的字符串描述了第一类PE的19种功能，表中的au表示算术运算，xor表示逻辑功能，sh表示移位功能，bn表示置换功能，lut表示S盒功能，gfm表示有限域乘法功能，字符串组合表示功能的组合，每一个功能字符串之间用“\_”字符隔开，在进行算法图节点和架构图节点进行匹配时，算法图节点的功能属性必须是架构图节点功能属性的子集，也就是架构图节点对应的PE能够提供算法节点所需的功能，在属性比较时，算法图节点属性字符串是架构图节点属性字符串的子串。架构图中的（x，y）位置属性标识了该节点对应的PE在架构中的位置。在完成算法映射匹配时根据这些位置信息从多个匹配中选择出占用最少行、列资源的结果。

表5-2架构图结点属性

|  |  |  |  |
| --- | --- | --- | --- |
| PE（节点） | x | y | 功能属性 |
| PE\_type1 | 1 | 1,2 | au\_xorau\_auxor\_xorauxor\_xorxorau\_xorxorauxor\_sh\_xorsh\_shxor\_xorshxor\_xor\_xorxor\_xorxorxor\_or\_and\_bn\_xorbn\_bnxor\_xorbnxor\_ |
| PE\_type2 | 1 | 3,4 | au\_xorau\_auxor\_xorauxor\_xorxorau\_xorxorauxor\_sh\_xorsh\_shxor\_xorshxor\_xor\_xorxor\_xorxorxor\_or\_and\_ |
| PE\_type3 | 2 | 1,2,3,4 | au\_xorau\_auxor\_xorauxor\_xorxorau\_xorxorauxor\_sh\_xorsh\_shxor\_xorshxor\_xor\_xorxor\_xorxorxor\_or\_and\_lut\_xorlut\_lutxor\_xorlutxor\_ |
| PE\_type4 | 3 | 1,2 | au\_xorau\_auxor\_xorauxor\_xorxorau\_xorxorauxor\_sh\_xorsh\_shxor\_xorshxor\_xor\_xorxor\_xorxorxor\_or\_and\_bn\_xorbn\_bnxor\_xorbnxor\_gfm\_xorgfm\_gfmxor\_xorgfmxor\_ |
| PE\_type5 | 4 | 3,4 | au\_xorau\_auxor\_xorauxor\_xorxorau\_xorxorauxor\_sh\_xorsh\_shxor\_xorshxor\_xor\_xorxor\_xorxorxor\_or\_and\_gfm\_xorgfm\_gfmxor\_xorgfmxor\_ |

如图xxx所示为一个异构组的图模型，每一行的4个PE被抽象成四个点，每个点有与PE对应的功能属性，行与行之间的互连被抽象成从上一行到下一行的有向边，架构中的行间互连支持数据从上一行的PE到下一行的任意一个PE，因此在图模型中上一行的每一个点都有一条有向边到下一行的任意一个点。



图5-5异构组的图模型

[1]沈来信, 曾国荪, and 王伟. "基于分层超图的可重构体系结构模型." *计算机科学*40.4 (2013): 26-30.

[2]沈来信, and 王伟. "基于算粒感知的可重构体系结构." *Computer Engineering*39.9 (2013).

* + 1. 基于VF2算法的映射方案
       1. VF2算法适应性调整

子图同构是指在模板图中存在一个和查询图同构的子图，根据公式xxx可知同构是一个双射关系。在算法图和架构图匹配中，目标是从架构图中找出一个图，这个图包含算法图，但不要求同构，包含是一个从算法图到架构图的单射关系。因此在约束上，映射匹配的约束比子图同构弱。

图xxx描述了架构图和算法图的包含关系，映射{(v1+v2, a1), (v3, a5), (v4, a6), (v5, a7), (v6+v7+v8, a9)}中架构图的子图{a1, a5 ,a6, a7, a9}和对应的算法图不同构，子图中的多条边在算法图中没有对应的边存在，这只满足单射关系。

VF2算法的和规则对图同构的双射关系进行了约束，因此对于映射算法必须对这两个放宽这两个规则，新的规则如下：





在新的规则中，只要求图（架构图）中包含图（算法图）对应的前驱和后继，反向则不再约束。



图5- 6架构图和算法图的包含关系

单纯的VF2算法不考虑点的属性问题，规则和只保证了对应点的前驱和后继的对应关系，它不保证架构图中的点能提供算法图对应的点所包含的操作属性。在进行算法图和架构图建模时，用字符串来描述图中节点的功能属性，因此架构图节点功能属性字符串要包含算法图的操作属性字符串，增加一个属性约束规则如式所示，其中attr(x)表示节点x的功能属性。



* + - 1. 最优匹配成本约束规则

架构图对算法图满足包含关系的子图可能成千上万，这些子图都满足基本的映射要求，但并不都是我们所需要的。映射要考虑行资源、列资源、互联资源的成本，还要考虑映射聚集等因素。在本文提出的映射方案中主要考虑了5个成本约束，满足这5个成本约束的最终结果满足使用最少的行、列资源，最简单的数据互连，以及尽可能往架构的右上聚集。

对算法图中的节点加入拓扑信息a，v.a表示算法图中的点v出现在横向上的位置；对架构图中的节点加入拓扑信息（x, y），其中v.x表示架构图中点v代表的PE出现在纵向上的位置，v.y表示架构图中点v代表的PE出现在横向上的位置。S表示匹配算法找出的所有可能的匹配集合， 为S中的一个元素，表示一个有效的匹配；(,)表示一个有效匹配中的一个匹配点对。五个成本约束如式~所示。











式是映射最优选择的第一优先级约束，它保证选出的候选集的映射占有最少的行数。首先统计每一个可行映射中行数的最大值，然后从这些最大值中选出行数最少的映射集合。算法在可重构阵列上按行映射，因此最小的资源单位是行，映射的行数最少表示使用的资源最少。

式是映射最优选择的第二优先级约束，它保证选出的候选集的映射占有最少的列数。首先统计每一个可行映射中列数的最大值，然后从这些最大值中选出列数最少的映射集合。对于很多算法，映射时不需要阵列上的所有列参与运算，对于这样的算法可以同时进行多路映射，找出列数最少的映射，总列数除以这个值就是这个算法允许同时执行的路数。

通过式和这两个约束选择出的映射同时满足行列最小化，这已经是最优映射；式、和确定的是映射工具的映射风格。式和分别约束在算法映射时的功能聚集方向，在能完成映射的前提下，功能总会向架构的左上边聚集。式约束功能乱序，在映射时并行功能在并行上的顺序和在算法本身中的顺序一致。式、和虽然和最终的资源最优无关，但是这三个规则使所有算法在架构上映射时都有统一的映射风格，保证所有算法不会胡乱使用资源，这种资源使用的倾向性可以进一步指导架构设计的功能资源倾向性，这对于进行一步的架构优化十分重要。

构建了算法图和架构图后，采用下面几篇论文中使用的子图同构（或者近似子图同构）的映射方案

[1]Clark, Nathan, et al. "Scalable subgraph mapping for acyclic computation accelerators." Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. ACM, 2006.

[2]Hamzeh, Mahdi, Aviral Shrivastava, and Sarma Vrudhula. "EPIMap: using epimorphism to map applications on CGRAs." Proceedings of the 49th Annual Design Automation Conference. ACM, 2012.

[3]Tuhin, Mohammed Ashraful Alam, and Theodore S. Norvell. "Compiling parallel applications to coarse-grained reconfigurable architectures." Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on. IEEE, 2008.

[4]Brenner, Janina A., Sándor P. Fekete, and Jan C. Van Der Veen. "A minimization version of a directed subgraph homeomorphism problem."Mathematical Methods of Operations Research 69.2 (2009): 281-296.

* 1. 算法映射分析与PE方案优化

本文的目标算法集合中有36个常用的分组密码算法，分别是：AES、DES、IDEA、BLOWFISH、CAMELLIA、CAST128、GOST、RC5、SEED、TWOFISH、SM4、RC6、SERPENT、TEA、XTEA、SKIPJECT、SPECK、SIMON、LUCIFER、CLEFIA、ARIA、C2、PRESENT、MACGUFFIN、SQUARE、M6、ICE、SHARK、CS\_CIPHER、NUSH、GRAND\_CRU、Q、E2、KHAZAD、HIEROCRYPT\_L1、HIEROCRYPT\_3。配置这些算法的图模型，将这些算法的算法图映射到架构图中。在这36个算法中有30个算法能够完成映射，另外6个算法存在一些本架构不支持的运算单元，如乘法单元、条件判断等，无法完成映射。统计异构组架构中的各个PE在完成30个算法映射后功能单元的使用情况，如表xxx和表xxx所示，表xxx是常规功能单元使用统计，表xxx是异构功能单元使用统计。

表xxx 常规功能单元使用统计

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 功能  单元 | 内部功能 | PE1 | PE2 | PE3 | PE4 | PE5 | PE6 | PE7 | PE8 | PE9 | PE10 | PE11 | PE12 |
| 算术  单元 | au | 5 | 2 | **1** | **1** | 8 | 2 | 0 | 0 | 5 | 0 | 0 | 0 |
| xorau | 1 | 0 | **0** | **0** | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| auxor | 4 | 0 | **0** | **0** | 1 | 1 | 0 | 0 | 3 | 0 | 0 | 0 |
| xorauxor | 0 | 0 | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| xorxorau | 0 | 0 | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| xorxorauxor | 0 | 0 | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 移位  单元 | sh | 8 | 3 | **1** | 0 | 1 | 3 | 0 | 0 | 3 | 3 | **1** | 0 |
| xorsh | 3 | 1 | **0** | 0 | 0 | 0 | 0 | 0 | 2 | 1 | **0** | 0 |
| shxor | 0 | 0 | **0** | 0 | 1 | 0 | 0 | 0 | 2 | 0 | **0** | 0 |
| xorshxor | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** | 0 |
| 逻辑  单元 | xor | 4 | 3 | **1** | **1** | 3 | **0** | 0 | 0 | 5 | **1** | 0 | 0 |
| xorxor | 2 | 3 | **0** | **0** | 1 | **1** | 0 | 0 | 1 | **0** | 0 | 0 |
| xorxorxor | 3 | 0 | **0** | **0** | 0 | **0** | 0 | 0 | 1 | **0** | 0 | 0 |
| or | 0 | 0 | **0** | **0** | 0 | **0** | 0 | 0 | 0 | **0** | 0 | 0 |
| and | 0 | 0 | **0** | **0** | 1 | **0** | 0 | 0 | 3 | **0** | 0 | 0 |

表xxx异构功能单元使用统计

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 功能单元 | 内部功能 | PE1 | PE2 | PE9 | PE10 |
| 置换单元 | bn | 1 | 1 | 1 | 0 |
| xorbn | 0 | 0 | 0 | 0 |
| bnxor | 1 | 0 | 3 | 0 |
| xorbnxor | 0 | 0 | 0 | 0 |
| S盒替代单元 |  | PE5 | PE6 | PE7 | PE8 |
| lut | 8 | 4 | 2 | 2 |
| xorlut | 20 | 17 | 7 | 7 |
| lutxor | 0 | 0 | 0 | 0 |
| xorlutxor | 0 | 0 | 0 | 0 |
| 有限域乘法单元 |  | PE9 | PE10 | PE11 | PE12 |
| gfm | 12 | 12 | 6 | 6 |
| xorgfm | 0 | 0 | 0 | 0 |
| gfmxor | 0 | 0 | 0 | 0 |
| xorgfmxor | 0 | 0 | 0 | 0 |

在表xxx和表xxx中，灰色背景标记部分是所有功能都闲置的单元，加粗字体标记部分是使用频率低的单元，可以对异构组架构PE功能单元使用情况总结如下：

1. 由于映射方式的左上聚集特征，架构中后两列PE功能单元使用频率比前两列的要低很多，存在很多的闲置单元，而前两列中的闲置单元要少很多；
2. 除了利用率为0的闲置单元以外，很多单元的使用频率为1，表示只有一个算法使用了该功能；
3. 常规功能单元使用频率比异构功能单元使用频率要低很多，在异构功能单元中，只有一个功能单元是闲置的，其它功能单元都有比较好的利用率；
4. 组合功能利用率中，有些功能利用比较频率比较高，但仍然有大部分被闲置。

PE中的闲置单元表示在映射所有的算法后这些单元都没被使用过，这样的功能单元从架构中删除后对算法映射毫无影响；功能利用低表示在映射所有的算法后只有一两个算法使用了该功能，对于这样的功能单元可以尝试删除，如果对应的算法能从别的功能路径上映射通过，且不影响映射规模，那么该功能单元可以删除，否则即使利用率低也要保留该单元。异构组中12个PE中的闲置单元和低利用率单元如表xxx所示。

表xxx PE中的闲置单元和低利用率单元

|  |  |  |
| --- | --- | --- |
| PE | 闲置单元 | 低利用率单元 |
| PE1 | 无 | 无 |
| PE2 | 无 | 无 |
| PE3 | 无 | 算术单元、移位单元、逻辑单元 |
| PE4 | 移位单元 | 算术单元、逻辑单元 |
| PE5 | 无 | 无 |
| PE6 | 无 | 逻辑单元 |
| PE7 | 算术单元、移位单元、逻辑单元 | 无 |
| PE8 | 算术单元、移位单元、逻辑单元 | 无 |
| PE9 | 无 | 无 |
| PE10 | 算术单元、置换单元 | 移位单元 |
| PE11 | 算术单元、逻辑单元 | 移位单元 |
| PE12 | 算术单元、移位单元、逻辑单元 | 无 |

PE3和PE4中，算术单元的使用算法是LUCIFER，删除算术单元重新映射时LUCIFER算法的映射规模增加了三行，因此放弃对算术单元的优化；PE3中的移位单元删除后，对应使用了该单元的TEA算法从另外的路径映射成功且没有增加映射规模，因此将移位单元优化掉，与此类似的还有逻辑单元。其它PE中的低利用率单元在删除后对应算法的映射规模都增加了，因此确定方案时选择不对这些单元进行优化。对于组合功能中很多闲置的组合功能，虽然有一定的优化空间，但是组合中串联的是异或逻辑操作，异或逻辑消耗的电路资源十分有限，优化效果也十分有限，因此保留了这些组合功能。

综上所述，相对初始PE方案，最终的优化方案减少了5个算术单元，5个移位单元，6个逻辑单元和1个置换单元。新的PE功能包含如表xxx所示，PE间的异构不只在置换单元、S盒和有限域乘法，PE间在算术单元、移位单元和逻辑单元的上也是异构的，PE的种类由初始建构中的5类扩展成了8类。

表xxx 优化后PE的功能包含

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PE | 算术单元 | 移位单元 | 逻辑单元 | 置换单元 | S盒 | 有限域乘法 |
| PE1 | **√** | **√** | **√** | **√** | **×** | **×** |
| PE2 | **√** | **√** | **√** | **√** | **×** | **×** |
| PE3 | **√** | **×** | **×** | **×** | **×** | **×** |
| PE4 | **√** | **×** | **×** | **×** | **×** | **×** |
| PE5 | **√** | **√** | **√** | **×** | **√** | **×** |
| PE6 | **√** | **√** | **√** | **×** | **√** | **×** |
| PE7 | **×** | **×** | **×** | **×** | **√** | **×** |
| PE8 | **×** | **×** | **×** | **×** | **√** | **×** |
| PE9 | **√** | **√** | **√** | **√** | **×** | **√** |
| PE10 | **×** | **√** | **√** | **×** | **×** | **√** |
| PE11 | **×** | **√** | **×** | **×** | **×** | **√** |
| PE12 | **×** | **×** | **×** | **×** | **×** | **√** |

* 1. 本章小结

1. 优化PE方案的验证与分析
   1. PE方案实现

根据第五章给出的PE优化方案，使用Verilog HDL语言描述了该优化方案，完成可重构PE阵列设计，使用Synopsys公司的Verilog Complier Simulator（VCS）工具进行功能和时序仿真，通过设计的功能验证。基于TSMC 40nm CMOS工艺标准单元库对PE设计进行逻辑综合，综合工具使用Synopsys公司的Design Compiler（DC），生成面积、时序等参数，最终按照500MHz主频对硬件设计进行时序约束，整个异构组的面积为217797μm2，各个功能单元的面积和延迟如表xxx所示。

表6- 1 架构中的电路单元在DC上的综合结果

|  |  |  |
| --- | --- | --- |
| 功能单元 | 面积/μm2 | 延迟/ns |
| 逻辑单元 | 1585 | 1.27 |
| 算术单元 | 1877 | 0.97 |
| 移位单元 | 399 | 0.95 |
| 置换单元 | 7065 | 1.25 |
| S盒子替换单元 | 12797 | 1.42 |
| 有限域乘法单元 | 8506 | 1.17 |
| 互连单元 | 28253 | 0.57 |
| 按字节循环移位单元 | 448 | 0.16 |

* 1. 算法映射

本文对30个分组密码算法进行了映射分析，分别是：AES、DES、BLOWFISH、CAMELLIA、CAST128、GOST、RC5、SEED、TWOFISH、SM4、SERPENT、TEA、XTEA、SPECK、SIMON、LUCIFER、CLEFIA、ARIA、C2、PRESENT、MACGUFFIN、SQUARE、M6、SHARK、NUSH、GRAND\_CRU、E2、KHAZAD、HIEROCRYPT\_L1、HIEROCRYPT\_3。分组密码算法按轮函数重复迭代完成运算，因此在算法映射时只针对算法的一轮来分析。由于算法数量巨大，无法在有限的篇幅里列出每一个算法的映射分析，本节只针对AES、DES和SPECK三个算法进行详细映射分析，同时给出所有算法的映射结果统计分析。

* + 1. AES算法映射结果

AES算法的轮函数共有4个连续操作：秘钥加、字节替换、行移位和列混淆，分别对应于异或逻辑运算、S盒、字节循环移位和有限域乘法运算；将算法流程转换成有向图，共有12个点和12条数据关联边，如图6-1中(b)所示；算法中字节替换和行移位可以组合在一行中完成，(c)中第一行的4个PE完成异或逻辑运算，第二行的4个PE完成S盒+字节循环移位运算，第三行的4个PE完成有限域乘法运算；行间PE互连如(c)中加粗虚线所示，都是直连方式。整个算法轮函数在三行PE中完成映射，对应一个PE异构组。



图6- 1 AES算法映射

AES算法轮函数中共有4个逻辑运算，4个查表运算和4个有限域乘法运算；PE方案在优化后，一个异构组中含有7个算术单元，7个移位单元，3个置换单元，6个逻辑单元，4个S盒盒4个有限域乘法单元；对应的各个单元的利用率如表6-2的第4行所示，最后一列的综合利用率是指所有单元的平均利用率。

表6- 2 AES算法映射结果分析

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 功能单元 | 算术  单元 | 移位  单元 | 置换  单元 | 逻辑  单元 | S盒 | 有限域乘法 | 综合 |
| AES算法包含运算 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| 架构包含功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| 利用率 | 0% | 0% | 0% | 67% | 100% | 100% | 39% |

* + 1. DES算法映射结果

DES算法的轮函数如图6-2中(a)所示，包含5个连续运算：扩展、秘钥加、代替、置换和左右异或，分别对应于置换、异或逻辑运算、S盒、置换和异或逻辑运算；将算法流程转换成有向图，共有7个点和7条数据关联边，如图6-2中(b)所示。算法中异或运算和S盒可以组合在一行中完成，同样的情况还有置换运算和异或运算。(c)中第一行的第1个PE完成置换运算，其它3个PE空闲，第二行的2个PE完成异或+S盒运算，第三行的1个PE完成置换+异或运算，行间PE互连如(c)中加粗虚线所示。整个算法轮函数在三行PE中完成映射，对应一个PE异构组。

图6- 2 DES算法映射

DES算法轮函数中共有3个逻辑运算，2个查表运算和2个置换运算；PE方案在优化后，一个异构组中含有7个算术单元，7个移位单元，3个置换单元，6个逻辑单元，4个S盒盒4个有限域乘法单元；对应的各个单元的利用率如表6-3的第4行所示。

表6- 3 DES算法映射结果分析

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 功能单元 | 算术  单元 | 移位  单元 | 置换  单元 | 逻辑  单元 | S盒 | 有限域乘法 | 综合 |
| DES算法包含运算 | 0 | 0 | 2 | 3 | 2 | 0 | 7 |
| 架构包含功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| 利用率 | 0% | 0% | 67% | 50% | 50% | 0% | 23% |

* + 1. SPECK算法映射结果

SPECK算法的轮函数如图6-3中(a)所示，包含2个移位运算，一个算术运算和2个逻辑运算；将算法流程转换成有向图，共有5个点和5条数据关联边，如图6-3中(b)所示。算法中算术查找和异或运算可以组合在一行中完成。(c)中第一行的2个PE完成移位操，第二行的1个PE完成算术+异或运算，还有1个PE被用来数据直通，第三行的1个PE完成异或运算，行间PE互连如(c)中加粗虚线所示。整个算法轮函数在三行PE中完成映射，对应一个PE异构组。



图6- 3 SPECK算法映射

SPECK算法轮函数中共有2个逻辑运算，2个移位运算和1个算术运算；PE方案在优化后，一个异构组中含有7个算术单元，7个移位单元，3个置换单元，6个逻辑单元，4个S盒盒4个有限域乘法单元；对应的各个单元的利用率如表6-4的第4行所示。

表6- 4 SPECK算法映射结果分析

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 功能单元 | 算术  单元 | 移位  单元 | 置换  单元 | 逻辑  单元 | S盒 | 有限域乘法 | 综合 |
| SPECK算法包含运算 | 1 | 2 | 0 | 2 | 0 | 0 | 5 |
| 架构包含功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| 利用率 | 14% | 29% | 0% | 33% | 0% | 0% | 16% |

* 1. 与其它可重构平台对比分析
     1. 算法集合在不同架构平台上的映射结果

表6- 5 算法在不同架构平台上的映射结果（部分）

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | 架构 | 功能单元利用率 | | | | | | | | 性能 | | |
|  | 算术单元 | 移位单元 | 置换单元 | 逻辑单元 | S盒 | 有限域乘法 | 综合 | 性能/Gbps | 面积/mm2 | 性能面积比 |
| AES |  | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 59.5 | 0.9 | 67.9 |
| 资源利用率 | 0% | 0% | 0% | 8% | 7% | 7% | 9% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 32.9 | 0.4 | 78.4 |
| 资源利用率 | 0% | 0% | 0% | 25% | 25% | 50% | 19% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| RPU | 映射资源消耗 | 8 | 16 | 4 | 48 | 16 | 0 | 92 | 64.0 | 0.4 | 182.1 |
| 资源利用率 | 0% | 0% | 0% | 8% | 25% | / | 13% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **67%** | **100%** | **100%** | **39%** |
| DES |  | 算法包含操作 | 0 | 0 | 2 | 3 | 2 | 0 | 7 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 0% | 0% | 17% | 4% | 2% | / | 4% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 0% | 0% | 17% | 13% | 8% | 0% | 7% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 0% | 0% | 25% | 3% | 6% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **67%** | **50%** | **50%** | **0%** | **23%** |
| S  PECK |  | 算法包含操作 | 1 | 2 | 0 | 2 | 0 | 0 | 5 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 8% | 17% | 0% | 3% | 0% | / | 3% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 8% | 17% | 0% | 8% | 0% | 0% | 5% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 15.9 |
| 资源利用率 | 6% | 8% | 0% | 8% | 0% | 0% | 6% |
| RPU | 映射资源消耗 | 12 | 24 | 6 | 72 | 24 | 0 | 138 | 32.0 | 0.5 | 60.7 |
| 资源利用率 | 8% | 8% | 0% | 3% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **14%** | **29%** | **0%** | **33%** | **0%** | **0%** | **16%** |

* + 1. 对比分析

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 架构 | 算法 | AES | DES | SM4 | TWOFISH | RC5 | CAST128 | SERPENT | BLOWFISH |
| Cyptoraptor | 功能单元综合利用率 | 9% | 4% | 2% | 4% | 2% | 3% | 12% | 12% |
| 性能面积比(Gbps/mm2) | 67.9 | 22.6 | 34.0 | 34.0 | 34.0 | 13.6 | 41.0 | 22.6 |
| RCPA | 功能单元综合利用率 | 19% | 7% | 2% | 6% | 5% | 5% | 20% | 20% |
| 性能面积比(Gbps/mm2) | 78.4 | 26.1 | 13.1 | 13.1 | 39.2 | 15.7 | 39.3 | 26.1 |
| COBRA | 功能单元综合利用率 | 14％ | / | 5% | 10% | 7% | 10% | 24% | 24% |
| 性能面积比(Gbps/mm2) | 31.8 | / | 21.2 | 21.2 | 31.8 | 15.9 | 25.2 | 31.8 |
| RPU | 功能单元综合利用率 | 13% | 4% | 3% | 4% | 3% | 3% | 15% | 15% |
| 性能面积比(Gbps/mm2) | 182.1 | 45.5 | 91.1 | 91.1 | 91.1 | 36.4 | 91.4 | 45.5 |
| **本文** | **功能单元综合利用率** | **39%** | **23%** | **19%** | **39%** | **19%** | **26%** | **77%** | **12%** |
| **性能面积比** | **293.9** | **146.9** | **293.9** | **293.9** | **220.4** | **73.5** | **132.3** | **146.9** |
| 架构平台 | 算法 | SEED | CAMELLIA | GOST | TEA | XTEA | SPECK | SIMON | LUCIFER |
| Cyptoraptor | 功能单元综合利用率 | 3% | 4% | 2% | 3% | 4% | 3% | 5% | 5% |
| 性能面积比(Gbps/mm2) | 13.6 | 34.0 | 22.6 | 17.0 | 22.6 | 22.6 | 32.7 | 45.3 |
| RCPA | 功能单元综合利用率 | 5% | 5% | 4% | 6% | 7% | 5% | 10% | 10% |
| 性能面积比(Gbps/mm2) | 14.2 | 26.1 | 26.1 | 19.6 | 26.1 | 26.1 | 33.6 | 52.2 |
| COBRA | 功能单元综合利用率 | 6% | 8% | 10% | 5% | 8% | 6% | 12% | 0 |
| 性能面积比(Gbps/mm2) | 9.1 | 21.2 | 31.8 | 8.0 | 14.6 | 15.9 | 20.9 | 0 |
| RPU | 功能单元综合利用率 | 3% | 4% | 2% | 4% | 4% | 4% | 4% | 5% |
| 性能面积比(Gbps/mm2) | 30.4 | 60.7 | 45.5 | 45.5 | 47.4 | 60.7 | 50.5 | 91.1 |
| **本文** | **功能单元综合利用率** | **11%** | **16%** | **13%** | **19%** | **23%** | **16%** | **23%** | **32%** |
| **性能面积比** | **58.8** | **146.9** | **146.9** | **110.2** | **146.9** | **146.9** | **146.9** | **293.9** |
| 架构平台 | 算法 | CLEFIA | ARIA | C2 | PRESENT | MACGUFFIN | SQUARE | M6 | SHARK |
| Cyptoraptor | 功能单元综合利用率 | 5% | 8% | 3% | 4% | 4% | 4% | 2% | 5% |
| 性能面积比(Gbps/mm2) | 45.3 | 67.9 | 13.6 | 34 | 34 | 34 | 7.5 | 34 |
| RCPA | 功能单元综合利用率 | 10% | 16% | 6% | 8% | 8% | 8% | 3% | 9% |
| 性能面积比(Gbps/mm2) | 52.2 | 78.4 | 15.7 | 39.2 | 39.2 | 39.2 | 8.7 | 39.2 |
| COBRA | 功能单元综合利用率 | 12% | 24% | 7% | 0 | 6% | 6% | 6% | 14% |
| 性能面积比(Gbps/mm2) | 31.8 | 63.7 | 10.6 | 0 | 15.9 | 15.9 | 8 | 31.8 |
| RPU | 功能单元综合利用率 | 5% | 5% | 3% | 5% | 5% | 5% | 2% | 3% |
| 性能面积比(Gbps/mm2) | 91.1 | 91.1 | 30.4 | 91.1 | 91.1 | 91.1 | 20.2 | 45.5 |
| **本文** | **功能单元综合利用率** | **32%** | **32%** | **15%** | **16%** | **16%** | **16%** | **11%** | **19%** |
| **性能面积比** | **293.9** | **293.9** | **73.5** | **146.9** | **146.9** | **146.9** | **49** | **146.9** |
| 架构平台 | 算法 | NUSH | GRAND CRU | E2 | KHAZAD | HIEROCRYPT-L1 | HIEROCRYPT-3 | **平均** | **提升倍数** |
| Cyptoraptor | 功能单元综合利用率 | 3% | 7% | 3% | 5% | 5% | 9% | **4%** | **5.33** |
| 性能面积比(Gbps/mm2) | 45.3 | 45.3 | 22.6 | 34 | 17 | 34 | **24.46** | **6.03** |
| RCPA | 功能单元综合利用率 | 5% | 15% | 6% | 9% | 9% | 19% | **7%** | **5.33** |
| 性能面积比(Gbps/mm2) | 52.2 | 52.2 | 26.1 | 39.2 | 19.6 | 39.2 | **22.96** | **6.42** |
| COBRA | 功能单元综合利用率 | 6% | 0 | 0 | 14% | 14% | 29% | **10%** | **2.32** |
| 性能面积比(Gbps/mm2) | 31.8 | 0 | 0 | 31.8 | 15.9 | 31.8 | **19.19** | **7.68** |
| RPU | 功能单元综合利用率 | 4% | 8% | 7% | 3% | 3% | 7% | **5%** | **4.67** |
| 性能面积比(Gbps/mm2) | 121.4 | 91.1 | 78.5 | 45.5 | 22.8 | 45.5 | **55.52** | **2.66** |
| **本文** | **功能单元综合利用率** | **16%** | **45%** | **13%** | **19%** | **19%** | **39%** | **22%** |  |
| **性能面积比** | **293.9** | **293.9** | **98** | **146.9** | **73.5** | **146.9** | **147.42** |  |

功能单元减少，这些减少的功能单元占原来架构的百分比（也就是说这种新设计可以为原架构减多少面积）

1. 功能单元消耗对比

算法集在不同架构的映射下阵列所需要的功能单元统计

课题中的架构优化的粒度是功能单元，因此这个表能直接说明课题的优化方案达到的效果

1. 面积、性能对比

这个对比是一般论文都会有的对比，也是杨博在汇报时提的要求。和论文中的对比会出现工艺对齐的问题，这些论文中都给出了阵列的门数，面积的对比有一定的参考意义。但是不同的工艺下性能是没有可比性的。

杨博要求和清华目前的PE进行对比，这个是可以做到的，到时可以使用相同的工艺库进行综合。

* 1. 本章小结

1. 总结与展望
   1. 总结

总结自己的工作，不是行内

* 1. 展望

自己的不足，而不是领域

两点

附录A

表A- 1算法在不同架构平台上的映射结果

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | 架构 | 功能单元利用率 | | | | | | | | 性能 | | |
|  | 算术单元 | 移位单元 | 置换单元 | 逻辑单元 | S盒 | 有限域乘法 | 综合 | 性能 | 面积/mm2 | 性能面积比 |
| AES |  | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 59.5 | 0.9 | 67.9 |
| 资源利用率 | 0% | 0% | 0% | 8% | 7% | 7% | 9% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 32.9 | 0.4 | 78.4 |
| 资源利用率 | 0% | 0% | 0% | 25% | 25% | 50% | 19% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| RPU | 映射资源消耗 | 8 | 16 | 4 | 48 | 16 | 0 | 92 | 64.0 | 0.4 | 182.1 |
| 资源利用率 | 0% | 0% | 0% | 8% | 25% | / | 13% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **67%** | **100%** | **100%** | **39%** |
| DES |  | 算法包含操作 | 0 | 0 | 2 | 3 | 2 | 0 | 7 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 0% | 0% | 17% | 4% | 2% | / | 4% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 0% | 0% | 17% | 13% | 8% | 0% | 7% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 0% | 0% | 25% | 3% | 6% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **67%** | **50%** | **50%** | **0%** | **23%** |
| SM4 |  | 算法包含操作 | 0 | 3 | 0 | 2 | 1 | 0 | 6 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 59.5 | 1.8 | 34.0 |
| 资源利用率 | 0% | 19% | 0% | 2% | 1% | / | 2% |
| RCPA | 映射资源消耗 | 48 | 48 | 48 | 96 | 96 | 48 | 384 | 32.9 | 2.5 | 13.1 |
| 资源利用率 | 0% | 6% | 0% | 2% | 1% | 0% | 2% |
| COBRA | 映射资源消耗 | 24 | 36 | 12 | 36 | 6 | 12 | 126 | 9.9 | 0.5 | 21.2 |
| 资源利用率 | 0% | 8% | 0% | 6% | 17% | 0% | 5% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 0% | 9% | 0% | 2% | 3% | / | 3% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **43%** | **0%** | **33%** | **25%** | **0%** | **19%** |
| TWOFISH |  | 算法包含操作 | 4 | 3 | 0 | 1 | 2 | 2 | 12 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 20 | 20 | 20 | 120 | 140 | 0 | 320 | 59.5 | 1.8 | 34.0 |
| 资源利用率 | 20% | 15% | 0% | 1% | 1% | / | 4% |
| RCPA | 映射资源消耗 | 24 | 24 | 24 | 48 | 48 | 24 | 192 | 32.9 | 2.5 | 13.1 |
| 资源利用率 | 17% | 13% | 0% | 2% | 4% | 8% | 6% |
| COBRA | 映射资源消耗 | 24 | 36 | 12 | 36 | 6 | 12 | 126 | 9.9 | 0.5 | 21.2 |
| 资源利用率 | 17% | 8% | 0% | 3% | 33% | 17% | 10% |
| RPU | 映射资源消耗 | 24 | 48 | 12 | 144 | 48 | 0 | 276 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 17% | 6% | 0% | 1% | 4% | / | 4% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **29%** | **21%** | **0%** | **8%** | **25%** | **25%** | **39%** |
| RC5 |  | 算法包含操作 | 2 | 2 | 0 | 2 | 0 | 0 | 6 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 59.5 | 1.8 | 34.0 |
| 资源利用率 | 13% | 13% | 0% | 2% | 0% | / | 2% |
| RCPA | 映射资源消耗 | 16 | 16 | 16 | 32 | 32 | 16 | 128 | 32.9 | 0.8 | 39.2 |
| 资源利用率 | 13% | 13% | 0% | 6% | 0% | 0% | 5% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 13% | 8% | 0% | 8% | 0% | 0% | 7% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 13% | 6% | 0% | 2% | 0% | / | 3% |
| **本文** | **映射资源消耗** | **9** | **9** | **4** | **8** | **5** | **5** | **31** | **64.0** | **0.3** | **220.4** |
| **资源利用率** | **21%** | **21%** | **0%** | **25%** | **0%** | **0%** | **19%** |
| CAST  1  2  8 |  | 算法包含操作 | 3 | 1 | 0 | 3 | 1 | 0 | 8 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 20 | 20 | 20 | 120 | 140 | 0 | 320 | 29.8 | 2.2 | 13.6 |
| 资源利用率 | 15% | 5% | 0% | 3% | 1% | / | 3% |
| RCPA | 映射资源消耗 | 20 | 20 | 20 | 40 | 40 | 20 | 160 | 16.5 | 1.0 | 15.7 |
| 资源利用率 | 15% | 5% | 0% | 8% | 3% | 0% | 5% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 15.9 |
| 资源利用率 | 19% | 4% | 0% | 13% | 25% | 0% | 10% |
| RPU | 映射资源消耗 | 20 | 40 | 10 | 120 | 40 | 0 | 230 | 32.0 | 0.9 | 36.4 |
| 资源利用率 | 15% | 3% | 0% | 3% | 3% | / | 3% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **31** | **32.0** | **0.4** | **73.5** |
| **资源利用率** | **21%** | **7%** | **0%** | **25%** | **13%** | **0%** | **26%** |
| SERPENT |  | 算法包含操作 | 0 | 8 | 0 | 12 | 4 | 0 | 24 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 28 | 12 | 12 | 72 | 84 | 0 | 208 | 59.5 | 1.5 | 41.0 |
| 资源利用率 | 0% | 67% | 0% | 17% | 5% | / | 12% |
| RCPA | 映射资源消耗 | 36 | 12 | 12 | 24 | 24 | 12 | 120 | 32.9 | 0.8 | 39.3 |
| 资源利用率 | 0% | 67% | 0% | 50% | 17% | 0% | 20% |
| COBRA | 映射资源消耗 | 32 | 24 | 8 | 24 | 4 | 8 | 100 | 9.9 | 0.4 | 25.2 |
| 资源利用率 | 0% | 33% | 0% | 50% | 100% | 0% | 24% |
| RPU | 映射资源消耗 | 32 | 24 | 6 | 72 | 24 | 0 | 158 | 64.0 | 0.7 | 91.4 |
| 资源利用率 | 0% | 33% | 0% | 17% | 17% | / | 15% |
| **本文** | **映射资源消耗** | **21** | **21** | **9** | **18** | **12** | **12** | **31** | **64.0** | **0.5** | **132.3** |
| **资源利用率** | **0%** | **38%** | **0%** | **67%** | **33%** | **0%** | **77%** |
| BLOWFISH |  | 算法包含操作 | 2 | 0 | 0 | 6 | 4 | 0 | 12 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 17% | 0% | 0% | 8% | 5% | / | 6% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 17% | 0% | 0% | 25% | 17% | 0% | 13% |
| COBRA | 映射资源消耗 | 8 | 12 | 4 | 12 | 2 | 4 | 42 | 5.0 | 0.2 | 31.8 |
| 资源利用率 | 25% | 0% | 0% | 50% | 200% | 0% | 29% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 13% | 0% | 0% | 6% | 13% | / | 7% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **29%** | **0%** | **0%** | **100%** | **100%** | **0%** | **39%** |
| SEED |  | 算法包含操作 | 3 | 0 | 0 | 11 | 3 | 0 | 17 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 40 | 40 | 40 | 240 | 280 | 0 | 640 | 59.5 | 4.4 | 13.6 |
| 资源利用率 | 8% | 0% | 0% | 5% | 1% | / | 3% |
| RCPA | 映射资源消耗 | 44 | 44 | 44 | 88 | 88 | 44 | 352 | 32.9 | 2.3 | 14.2 |
| 资源利用率 | 7% | 0% | 0% | 13% | 3% | 0% | 5% |
| COBRA | 映射资源消耗 | 56 | 84 | 28 | 84 | 14 | 28 | 294 | 9.9 | 1.1 | 9.1 |
| 资源利用率 | 5% | 0% | 0% | 13% | 21% | 0% | 6% |
| RPU | 映射资源消耗 | 48 | 96 | 24 | 288 | 96 | 0 | 552 | 64.0 | 2.1 | 30.4 |
| 资源利用率 | 6% | 0% | 0% | 4% | 3% | / | 3% |
| **本文** | **映射资源消耗** | **35** | **35** | **15** | **30** | **20** | **20** | **155** | **64.0** | **1.1** | **58.8** |
| **资源利用率** | **9%** | **0%** | **0%** | **37%** | **15%** | **0%** | **11%** |
| CAMELL  I  A |  | 算法包含操作 | 0 | 0 | 0 | 8 | 2 | 0 | 10 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 59.5 | 1.8 | 34.0 |
| 资源利用率 | 0% | 0% | 0% | 8% | 2% | / | 4% |
| RCPA | 映射资源消耗 | 24 | 24 | 24 | 48 | 48 | 24 | 192 | 32.9 | 1.3 | 26.1 |
| 资源利用率 | 0% | 0% | 0% | 17% | 4% | 0% | 5% |
| COBRA | 映射资源消耗 | 24 | 36 | 12 | 36 | 6 | 12 | 126 | 9.9 | 0.5 | 21.2 |
| 资源利用率 | 0% | 0% | 0% | 22% | 33% | 0% | 8% |
| RPU | 映射资源消耗 | 24 | 48 | 12 | 144 | 48 | 0 | 276 | 64.0 | 1.1 | 60.7 |
| 资源利用率 | 0% | 0% | 0% | 6% | 4% | / | 4% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **62** | **64.0** | **0.4** | **146.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **67%** | **25%** | **0%** | **16%** |
| GOST |  | 算法包含操作 | 1 | 1 | 0 | 1 | 1 | 0 | 4 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 8% | 8% | 0% | 1% | 1% | / | 2% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 8% | 8% | 0% | 4% | 4% | 0% | 4% |
| COBRA | 映射资源消耗 | 8 | 12 | 4 | 12 | 2 | 4 | 42 | 5.0 | 0.2 | 31.8 |
| 资源利用率 | 13% | 8% | 0% | 8% | 50% | 0% | 10% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 6% | 3% | 0% | 1% | 3% | / | 2% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **14%** | **14%** | **0%** | **17%** | **25%** | **0%** | **13%** |
| TEA |  | 算法包含操作 | 4 | 2 | 0 | 2 | 0 | 0 | 8 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 29.8 | 1.8 | 17.0 |
| 资源利用率 | 25% | 13% | 0% | 2% | 0% | / | 3% |
| RCPA | 映射资源消耗 | 16 | 16 | 16 | 32 | 32 | 16 | 128 | 16.5 | 0.8 | 19.6 |
| 资源利用率 | 25% | 13% | 0% | 6% | 0% | 0% | 6% |
| COBRA | 映射资源消耗 | 32 | 48 | 16 | 48 | 8 | 16 | 168 | 5.0 | 0.6 | 8.0 |
| 资源利用率 | 13% | 4% | 0% | 4% | 0% | 0% | 5% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 25% | 6% | 0% | 2% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **9** | **9** | **4** | **8** | **5** | **5** | **41** | **32.0** | **0.3** | **110.2** |
| **资源利用率** | **43%** | **21%** | **0%** | **25%** | **0%** | **0%** | **19%** |
| XTEA |  | 算法包含操作 | 3 | 2 | 0 | 2 | 0 | 0 | 7 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 25% | 17% | 0% | 3% | 0% | / | 4% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 25% | 17% | 0% | 8% | 0% | 0% | 7% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 14.6 |
| 资源利用率 | 19% | 8% | 0% | 8% | 0% | 0% | 8% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 47.4 |
| 资源利用率 | 19% | 6% | 0% | 2% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **43%** | **29%** | **0%** | **33%** | **0%** | **0%** | **23%** |
| S  PECK |  | 算法包含操作 | 1 | 2 | 0 | 2 | 0 | 0 | 5 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 29.8 | 1.3 | 22.6 |
| 资源利用率 | 8% | 17% | 0% | 3% | 0% | / | 3% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 16.5 | 0.6 | 26.1 |
| 资源利用率 | 8% | 17% | 0% | 8% | 0% | 0% | 5% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 15.9 |
| 资源利用率 | 6% | 8% | 0% | 8% | 0% | 0% | 6% |
| RPU | 映射资源消耗 | 12 | 24 | 6 | 72 | 24 | 0 | 138 | 32.0 | 0.5 | 60.7 |
| 资源利用率 | 8% | 8% | 0% | 3% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **14%** | **29%** | **0%** | **33%** | **0%** | **0%** | **16%** |
| S  IMON |  | 算法包含操作 | 0 | 3 | 0 | 4 | 0 | 0 | 7 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 8 | 8 | 48 | 56 | 0 | 132 | 29.8 | 0.9 | 32.7 |
| 资源利用率 | 0% | 38% | 0% | 8% | 0% | / | 5% |
| RCPA | 映射资源消耗 | 16 | 8 | 8 | 16 | 16 | 8 | 72 | 16.5 | 0.5 | 33.6 |
| 资源利用率 | 0% | 38% | 0% | 25% | 0% | 0% | 10% |
| COBRA | 映射资源消耗 | 24 | 12 | 4 | 12 | 2 | 4 | 58 | 5.0 | 0.2 | 20.9 |
| 资源利用率 | 0% | 25% | 0% | 33% | 0% | 0% | 12% |
| RPU | 映射资源消耗 | 8 | 32 | 8 | 96 | 32 | 0 | 176 | 32.0 | 0.6 | 50.5 |
| 资源利用率 | 0% | 9% | 0% | 4% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **43%** | **0%** | **67%** | **0%** | **0%** | **23%** |
| LUC  I  FER |  | 算法包含操作 | 4 | 0 | 1 | 1 | 4 | 0 | 10 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 59.5 | 1.3 | 45.3 |
| 资源利用率 | 33% | 0% | 8% | 1% | 5% | / | 5% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 32.9 | 0.6 | 52.2 |
| 资源利用率 | 33% | 0% | 8% | 4% | 17% | 0% | 10% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 25% | 0% | 13% | 1% | 13% | / | 5% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **57%** | **0%** | **33%** | **17%** | **100%** | **0%** | **32%** |
| CLEF  I  A |  | 算法包含操作 | 0 | 0 | 0 | 6 | 2 | 2 | 10 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 59.5 | 1.3 | 45.3 |
| 资源利用率 | 0% | 0% | 0% | 8% | 2% | / | 5% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 32.9 | 0.6 | 52.2 |
| 资源利用率 | 0% | 0% | 0% | 25% | 8% | 17% | 10% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 25% | 50% | 25% | 12% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 0% | 0% | 0% | 6% | 6% | / | 5% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **100%** | **50%** | **50%** | **32%** |
| AR I  A |  | 算法包含操作 | 0 | 0 | 0 | 6 | 2 | 2 | 10 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 59.5 | 0.9 | 67.9 |
| 资源利用率 | 0% | 0% | 0% | 13% | 4% | / | 8% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 32.9 | 0.4 | 78.4 |
| 资源利用率 | 0% | 0% | 0% | 38% | 13% | 25% | 16% |
| COBRA | 映射资源消耗 | 8 | 12 | 4 | 12 | 2 | 4 | 42 | 9.9 | 0.2 | 63.7 |
| 资源利用率 | 0% | 0% | 0% | 50% | 100% | 50% | 24% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 0% | 0% | 0% | 6% | 6% | / | 5% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **100%** | **50%** | **50%** | **32%** |
| C2 |  | 算法包含操作 | 2 | 2 | 1 | 3 | 1 | 0 | 9 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 20 | 20 | 20 | 120 | 140 | 0 | 320 | 29.8 | 2.2 | 13.6 |
| 资源利用率 | 10% | 10% | 5% | 3% | 1% | / | 3% |
| RCPA | 映射资源消耗 | 20 | 20 | 20 | 40 | 40 | 20 | 160 | 16.5 | 1.0 | 15.7 |
| 资源利用率 | 10% | 10% | 5% | 8% | 3% | 0% | 6% |
| COBRA | 映射资源消耗 | 24 | 36 | 12 | 36 | 6 | 12 | 126 | 5.0 | 0.5 | 10.6 |
| 资源利用率 | 8% | 6% | 8% | 8% | 17% | 0% | 7% |
| RPU | 映射资源消耗 | 24 | 48 | 12 | 144 | 48 | 0 | 276 | 32.0 | 1.1 | 30.4 |
| 资源利用率 | 8% | 4% | 8% | 2% | 2% | / | 3% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **62** | **32.0** | **0.4** | **73.5** |
| **资源利用率** | **14%** | **14%** | **17%** | **25%** | **13%** | **0%** | **15%** |
| PRESENT |  | 算法包含操作 | 0 | 0 | 1 | 2 | 2 | 0 | 5 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 29.8 | 0.9 | 34.0 |
| 资源利用率 | 0% | 0% | 13% | 4% | 4% | / | 4% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 16.5 | 0.4 | 39.2 |
| 资源利用率 | 0% | 0% | 13% | 13% | 13% | 0% | 8% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 8 | 16 | 4 | 48 | 16 | 0 | 92 | 32.0 | 0.4 | 91.1 |
| 资源利用率 | 0% | 0% | 25% | 4% | 13% | / | 5% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **33%** | **33%** | **50%** | **0%** | **16%** |
| MACGUFFIN |  | 算法包含操作 | 0 | 0 | 0 | 3 | 2 | 0 | 5 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 29.8 | 0.9 | 34.0 |
| 资源利用率 | 0% | 0% | 0% | 6% | 4% | / | 4% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 16.5 | 0.4 | 39.2 |
| 资源利用率 | 0% | 0% | 0% | 19% | 13% | 0% | 8% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 15.9 |
| 资源利用率 | 0% | 0% | 0% | 13% | 50% | 0% | 6% |
| RPU | 映射资源消耗 | 8 | 16 | 4 | 48 | 16 | 0 | 92 | 32.0 | 0.4 | 91.1 |
| 资源利用率 | 0% | 0% | 0% | 6% | 13% | / | 5% |
| 本文 | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **50%** | **50%** | **0%** | **16%** |
| SQUARE |  | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 59.5 | 1.3 | 45.3 |
| 资源利用率 | 0% | 0% | 0% | 6% | 5% | / | 6% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 32.9 | 0.6 | 52.2 |
| 资源利用率 | 0% | 0% | 0% | 17% | 17% | 33% | 13% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 0% | 0% | 0% | 4% | 13% | / | 7% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **67%** | **100%** | **100%** | **39%** |
| M6 |  | 算法包含操作 | 6 | 3 | 0 | 1 | 0 | 0 | 10 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 36 | 36 | 36 | 216 | 252 | 0 | 576 | 29.8 | 3.9 | 7.5 |
| 资源利用率 | 17% | 8% | 0% | 0% | 0% | / | 2% |
| RCPA | 映射资源消耗 | 36 | 36 | 36 | 72 | 72 | 36 | 288 | 16.5 | 1.9 | 8.7 |
| 资源利用率 | 17% | 8% | 0% | 1% | 0% | 0% | 3% |
| COBRA | 映射资源消耗 | 32 | 48 | 16 | 48 | 8 | 16 | 168 | 5.0 | 0.6 | 8.0 |
| 资源利用率 | 19% | 6% | 0% | 2% | 0% | 0% | 6% |
| RPU | 映射资源消耗 | 36 | 72 | 18 | 216 | 72 | 0 | 414 | 32.0 | 1.6 | 20.2 |
| 资源利用率 | 17% | 4% | 0% | 0% | 0% | / | 2% |
| **本文** | **映射资源消耗** | **21** | **21** | **9** | **18** | **12** | **12** | **93** | **32.0** | **0.7** | **49.0** |
| **资源利用率** | **29%** | **14%** | **0%** | **6%** | **0%** | **0%** | **11%** |
| SHARK |  | 算法包含操作 | 0 | 0 | 0 | 2 | 2 | 2 | 6 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 29.8 | 0.9 | 34.0 |
| 资源利用率 | 0% | 0% | 0% | 4% | 4% | / | 5% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 16.5 | 0.4 | 39.2 |
| 资源利用率 | 0% | 0% | 0% | 13% | 13% | 25% | 9% |
| COBRA | 映射资源消耗 | 8 | 12 | 4 | 12 | 2 | 4 | 42 | 5.0 | 0.2 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 0% | 0% | 0% | 2% | 6% | / | 3% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **33%** | **50%** | **50%** | **19%** |
| NUSH |  | 算法包含操作 | 2 | 1 | 0 | 2 | 0 | 0 | 5 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 59.5 | 1.3 | 45.3 |
| 资源利用率 | 17% | 8% | 0% | 3% | 0% | / | 3% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 32.9 | 0.6 | 52.2 |
| 资源利用率 | 17% | 8% | 0% | 8% | 0% | 0% | 5% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 13% | 4% | 0% | 8% | 0% | 0% | 6% |
| RPU | 映射资源消耗 | 12 | 24 | 6 | 72 | 24 | 0 | 138 | 64.0 | 0.5 | 121.4 |
| 资源利用率 | 17% | 4% | 0% | 3% | 0% | / | 4% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **29%** | **14%** | **0%** | **33%** | **0%** | **0%** | **16%** |
| GRAND CRU |  | 算法包含操作 | 0 | 0 | 2 | 4 | 4 | 4 | 14 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 12 | 12 | 12 | 72 | 84 | 0 | 192 | 59.5 | 1.3 | 45.3 |
| 资源利用率 | 0% | 0% | 17% | 6% | 5% | / | 7% |
| RCPA | 映射资源消耗 | 12 | 12 | 12 | 24 | 24 | 12 | 96 | 32.9 | 0.6 | 52.2 |
| 资源利用率 | 0% | 0% | 17% | 17% | 17% | 33% | 15% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.7 | 91.1 |
| 资源利用率 | 0% | 0% | 25% | 4% | 13% | / | 8% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **64.0** | **0.2** | **293.9** |
| **资源利用率** | **0%** | **0%** | **67%** | **67%** | **100%** | **100%** | **45%** |
| E2 |  | 算法包含操作 | 0 | 0 | 0 | 8 | 4 | 0 | 12 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 24 | 24 | 24 | 144 | 168 | 0 | 384 | 59.5 | 2.6 | 22.6 |
| 资源利用率 | 0% | 0% | 0% | 6% | 2% | / | 3% |
| RCPA | 映射资源消耗 | 24 | 24 | 24 | 48 | 48 | 24 | 192 | 32.9 | 1.3 | 26.1 |
| 资源利用率 | 0% | 0% | 0% | 17% | 8% | 0% | 6% |
| COBRA | 映射资源消耗 | / | / | / | / | / | / | / | / | / | / |
| 资源利用率 | / | / | / | / | / | / | / |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 64.0 | 0.8 | 78.5 |
| 资源利用率 | 0% | 0% | 0% | 8% | 13% | / | 7% |
| **本文** | **映射资源消耗** | **21** | **21** | **9** | **18** | **12** | **12** | **93** | **64.0** | **0.7** | **98.0** |
| **资源利用率** | **0%** | **0%** | **0%** | **44%** | **33%** | **0%** | **13%** |
| KHAZAD |  | 算法包含操作 | 0 | 0 | 0 | 2 | 2 | 2 | 6 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 8 | 8 | 8 | 48 | 56 | 0 | 128 | 29.8 | 0.9 | 34.0 |
| 资源利用率 | 0% | 0% | 0% | 4% | 4% | / | 5% |
| RCPA | 映射资源消耗 | 8 | 8 | 8 | 16 | 16 | 8 | 64 | 16.5 | 0.4 | 39.2 |
| 资源利用率 | 0% | 0% | 0% | 13% | 13% | 25% | 9% |
| COBRA | 映射资源消耗 | 8 | 12 | 4 | 12 | 2 | 4 | 42 | 5.0 | 0.2 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| RPU | 映射资源消耗 | 16 | 32 | 8 | 96 | 32 | 0 | 184 | 32.0 | 0.7 | 45.5 |
| 资源利用率 | 0% | 0% | 0% | 2% | 6% | / | 3% |
| **本文** | **映射资源消耗** | **7** | **7** | **3** | **6** | **4** | **4** | **31** | **32.0** | **0.2** | **146.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **33%** | **50%** | **50%** | **19%** |
| HIEROCRYPT-L1 |  | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 29.8 | 1.8 | 17.0 |
| 资源利用率 | 0% | 0% | 0% | 4% | 4% | / | 5% |
| RCPA | 映射资源消耗 | 16 | 16 | 16 | 32 | 32 | 16 | 128 | 16.5 | 0.8 | 19.6 |
| 资源利用率 | 0% | 0% | 0% | 13% | 13% | 25% | 9% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 5.0 | 0.3 | 15.9 |
| 资源利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| RPU | 映射资源消耗 | 32 | 64 | 16 | 192 | 64 | 0 | 368 | 32.0 | 1.4 | 22.8 |
| 资源利用率 | 0% | 0% | 0% | 2% | 6% | / | 3% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **62** | **32.0** | **0.4** | **73.5** |
| **资源利用率** | **0%** | **0%** | **0%** | **33%** | **50%** | **50%** | **19%** |
| HIEROCRYPT-3 |  | 算法包含操作 | 0 | 0 | 0 | 8 | 8 | 8 | 24 |  |  |  |
| Cyptoraptor | 映射资源消耗 | 16 | 16 | 16 | 96 | 112 | 0 | 256 | 59.5 | 1.8 | 34.0 |
| 资源利用率 | 0% | 0% | 0% | 8% | 7% | / | 9% |
| RCPA | 映射资源消耗 | 16 | 16 | 16 | 32 | 32 | 16 | 128 | 32.9 | 0.8 | 39.2 |
| 资源利用率 | 0% | 0% | 0% | 25% | 25% | 50% | 19% |
| COBRA | 映射资源消耗 | 16 | 24 | 8 | 24 | 4 | 8 | 84 | 9.9 | 0.3 | 31.8 |
| 资源利用率 | 0% | 0% | 0% | 33% | 200% | 100% | 29% |
| RPU | 映射资源消耗 | 32 | 64 | 16 | 192 | 64 | 0 | 368 | 64.0 | 1.4 | 45.5 |
| 资源利用率 | 0% | 0% | 0% | 4% | 13% | / | 7% |
| **本文** | **映射资源消耗** | **14** | **14** | **6** | **12** | **8** | **8** | **62** | **64.0** | **0.4** | **146.9** |
| **资源利用率** | **0%** | **0%** | **0%** | **67%** | **100%** | **100%** | **39%** |

参考文献

Hütter, Markus, Johann Großschädl, and Guy-Armand Kamendje. "A versatile and scalable digit-serial/parallel multiplier architecture for finite fields GF (2 m)."*Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on*. IEEE, 2003.

Cordella, Luigi P., et al. "A (sub) graph isomorphism algorithm for matching large graphs." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26.10 (2004): 1367-1372.

[V1] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. Analysis of the performance of coarse-grain reconfigurable architectures withdifferent processing element configurations. In *Workshop on ApplicationSpecific Processors, held in conjunction with the International Symposiumon Microarchitecture (MICRO)*, 2003.

[V2] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei.Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In Proceedings of the 2008 ACM SIGPLAN-SIGBEDConference on Languages, Compilers, and Tools for Embedded Systems,pages 151–160. ACM, 2008.

[V3] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker,Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the 17th annual ACM/SIGDA InternationalSymposium on Field Programmable Gate Arrays*, pages 191–200. ACM,2009.

[V4] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan.Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *Proceedings of the 2010 International Conference on Soft Computing and Pattern Recognition*, pages 145–151. IEEE,2010.

[V5] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithmfor a coarse-grain reconfigurable array template. In *Proceedings of 2007International Parallel and Distributed Processing Symposium*, pages 1–8.IEEE, 2007.

[V6] Bingfeng Mei, S Vernalde, D Verkest, H De Man, and R Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the 2003 Conference onDesign, Automation and Test in Europe*, pages 296–301. IEEE, 2003.