学校代码：10286
分类号：TN47
密　级：公开
U D C：621.38
学　号：131199

东南大学
SOUTHEAST UNIVERSITY

# 硕士学位论文

# 面向分组加密算法的可重构阵列处理单元优化与设计

研究生姓名：**李小泉**
导 师 姓 名：**孙伟锋 教授**

申请学位类别 **工学硕士**　学位授予单位 **东 南 大 学**
一级学科名称 **电子科学与技术**　论文答辩日期 **2016 年　月　日**
二级学科名称 **微电子学与固体电子学**　学位授予日期 **2016 年　月　日**
答辩委员会主席　评　阅　人

2016 年　月　日

東南大學

# 硕士学位论文

面向分组加密算法的可重构阵列处理单元优化与设计

**专 业 名 称：** 微电子学与固体电子学

**研究生姓名：** 李 小 泉

**导 师 姓 名：** 孙 伟 锋

# Optimization and Design of Processing Unit of Reconfigurable Array for Block Cipher Algorithm

A Thesis Submitted to

Southeast University

For the Academic Degree of Master of Engineering

BY

Li XiaoQuan

Supervised by

Professor Sun WeiFeng

School of Electronic Science and Engineering

Southeast University

May 2016

# 东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

研究生签名：_____日期：_____

# 东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括以电子信息形式刊登）论文的全部内容或中、英文摘要等部分内容。论文的公布（包括以电子信息形式刊登）授权东南大学研究生院办理。

研究生签名：_____导师签名：_____日期：_____

# 摘要

可重构系统兼顾了灵活性和高效性，适合于数据密集型的应用。密码算法处理数据量大，和可重构技术的结合可以满足密码算法在性能和安全方面的需求。为了适应密码算法不断提升的性能需求，密码可重构在向大规模处理单元阵列发展，在当前的密码可重构阵列架构中存在功能冗余量大、利用率低的问题，严重影响了整个系统的面积效率。本文重点研究了密码可重构阵列设计中的冗余功能单元优化。

本文以面向分组密码算法的可重构系统架构为基础平台，提出了一套冗余功能单元优化的密码可重构 PE 阵列设计方案。首先建立了一个分组密码算法图分析模型用来研究大量分组密码算法，提取与架构设计相关的算法特征：模式特征、组合特征和次序特征；根据提取的算法特征提出阵列的初始设计方案，对架构进行建模，以算法模型和架构模型作为输入，提出基于 VF2 子图同构的算法映射方案，完成大量分组密码算法的自动化映射，根据大量算法映射后的功能使用分布情况，对初始架构存在的冗余功能进行优化，多轮迭代最终得到无冗余的阵列设计方案。

本文电路实现采用 TSMC 40nm CMOS 工艺，电路运行主频为 500MHz；选取了 30 种比较常用分组密码算法作为实验测试集合，与其它面向分组密码算法的可重构处理器阵列架构相比，本文提出的可重构阵列架构方案功能单元平均利用率提高 83.2%~168.5%，面积效率提高 57.1%~643.5%。

**关键词：**可重构系统，分组密码算法，冗余优化，子图同构，算法映射

# Abstract

# 目录

# 第一章 绪论

## 1.1 研究背景

### 1.1.1 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[1]，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[2]，它将其视为一类计算机组织结构，有区别于其它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，这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，这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[3][4]、密码处理应用[5]和多媒体应用[6][7]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[8]、PipeRench[9]、ADRE[10]、MorphySy[11]、XPP[12]、Zippy[13]等。

### 1.1.2 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[14]：

（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的新的更安全的算法，同时支持随时修改密钥，满足某些特殊情况下的白片需求。

## 1.2 国内外研究现状

### 1.2.1 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[15]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[16]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/mm$^2$。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。

Cyptor[17]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/mm$^2$ 和 20.25Gbps/mm$^2$。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算子特征差异，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/mm$^2$ 和 5.83Gbps/mm$^2$。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 Cyptor 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，在算子组合上要么存在过度组合导致处理单元延迟过大的问题；要么不进行算子组合，存在映射周期过多的问题。在功能单元分布上，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，本文对分组密码算法建立了一个统一的图分析模型，充分挖掘算法的算子组合特征和算子次序特征，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。

### 1.2.2 密码可重构 PE 概述

密码可重构 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，而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。在串联组合中，PE 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，过度的串行组合也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，它用面积牺牲来换取更高的性能。

- 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。在并联时，通过数据选择器选择所需的运算的结果，串联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 方案的代表架构在映射不同算法时各种功能单元的利用率进行了总结。

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 | 平均利用率 |
| COBRA[15] | 串行组合 | 40% | AES | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| | | | RC5 | 0% | 8% | 0% | 6% | 17% | 0% | 5% |
| | | | CAST128 | 19% | 4% | 0% | 13% | 25% | 0% | 10% |
| RCPA[19] | 并行组合 | 90% | AES | 0% | 0% | 0% | 25% | 50% | 50% | 21% |
| | | | DES | 0% | 0% | 17% | 13% | 17% | 0% | 8% |
| | | | RC5 | 13% | 13% | 0% | 6% | 0% | 0% | 5% |
| Cyptor[17] | 串并混合 | 72% | AES | 0% | 0% | 0% | 50% | 50% | / | 30% |
| | | | DES | 0% | 0% | 17% | 25% | 17% | / | 12% |
| | | | RC5 | 13% | 13% | 0% | 13% | 0% | / | 8% |
| | | | BLOWFISH | 17% | 0% | 0% | 50% | 33% | / | 20% |
| | | | TWOFISH | 20% | 15% | 0% | 5% | 10% | / | 12% |

可以从表 1-1 发现：

- 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 的功能单元利用率部分列出了三种代表架构在实现几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，大部分在 30%以下。三种架构在平均利用率上也很相似，AES 的平均利用率较高，Cyptor 架构达到了 30%，其它算法的平均利用率都比较低，都在 20%以下。

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 70%-95%，也就是说在运行某一个算法时，整个芯片有 50%-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。

## 1.3 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 7 部分：

1) 选取目标算法集，研究这些算法的详细定义；本文选取了比较常见的 36 种分组密码算法作为研究的目标算法集合；
2) 为分组密码算法建立了一个统一的图分析模型，为后续的算法自动化特征提取、映射提供输入；
3) 基于建立的算法图模型，提取分组密码算法的一系列特征，包括算法算子的模式特征、组合特征和次序特征，这些特征和架构设计息息相关；
4) 确定 PE 初始设计方案，根据算法集中提取的算子模式特征、组合特征和次序特征分别确定 PE 方案中功能单元的功能模式、功能组合和功能分布；
5) 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；
6) 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问题的研究分析，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；

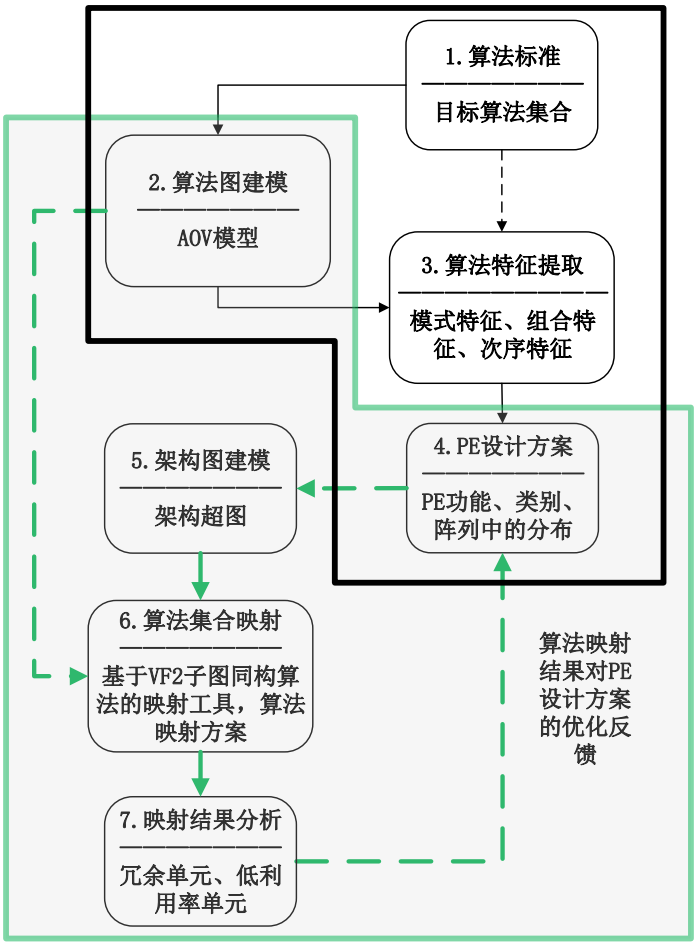



图 1-1 研究方案

7) 分析目标算法集合的映射结果，根据大量算法映射后的功能使用分布情况，对初始架构存在的冗余功能进行优化，多轮迭代最终得到无冗余的阵列设计方案；并对最终方案进行电路实现，功能验证和对比分析。

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，架构整体面积效率不高的问题，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，另一个是架构的在实现算法时的性能面积比。由于涉及到大量的分组密码算法，本文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文献中的其它架构平均提高 50%以上，面积效率相对于文献中的其它架构平均提高 30%以上。

表 1-2 考核指标

| 考核指标 | 相应提升要求 |
|---|---|
| 功能单元平均利用率 | 相对于文献中的其它架构平均提高 50%以上 |
| 性能面积比 | 相对于文献中的其它架构平均提高 30%以上 |

图 1-1 中的加黑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。灰色框标示的是本文在探索架构设计时提出的新内容，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对大量算法实施映射验证，并根据映射结果进行了反馈优化设计，同

时建立了参数化的架构模型，对架构进行多轮迭代式的设计调整并在自动化工具中验证。在本文的工作中主要有以下创新点：

1) 算法和架构的建模分析：对算法和架构建立一致的图分析模型，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) 更多的算法特征提取：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模式特征和组合特征；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次序特征，分析不同算法中这些算子次序特征的共性，根据次序特征的共性来安排 PE 设计中的功能单元分布；其它文献中的架构在 PE 设计时由于缺乏算子次序相关的特征分析，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) 自动化映射分析：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，这导致最终设计出来的架构具有很明显的算法偏向性；
4) 反馈优化：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，根据大量算法映射后的功能使用分布情况，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。

## 1.4 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，并在这个图模型上提取了分组密码算法算子的模式特征、组合特征和次序特征；

第四章根据提取的算子模式特征、组合特征和次序特征分别确定 PE 方案中功能单元的功能模式、功能组合和功能分布，提出 PE 设计初始方案；

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 VF2 算法应用到算法映射中，并通过对大量的算法映射分析，根据大量算法映射后的功能使用分布情况，找出冗余单元，完成 PE 设计的反馈优化。

第六章对本文的 PE 方案进行了电路实现、仿真验证和对比分析，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。

# 第二章 分组密码算法与密码可重构架构

## 2.1 分组密码算法

现代密码算法主要分为对称密码和非对称密码两大类，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。

### 2.1.1 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[20]。

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它组，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[20]。

### 2.1.2 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 $x_1$，$x_2$，…划分成长为 m 的组 x=($x_1$，$x_2$，…，$x_m$)，每个明文组（长为 m 的向量）分别在密钥 k=（$k_1$，$k_2$，…，$k_t$）的控制下变换成等长的输出数字序列 y=（$y_1$，$y_2$，…，$y_n$）（长为 n 的向量），整个运行过程的的数学模型如图 2-1 所示。

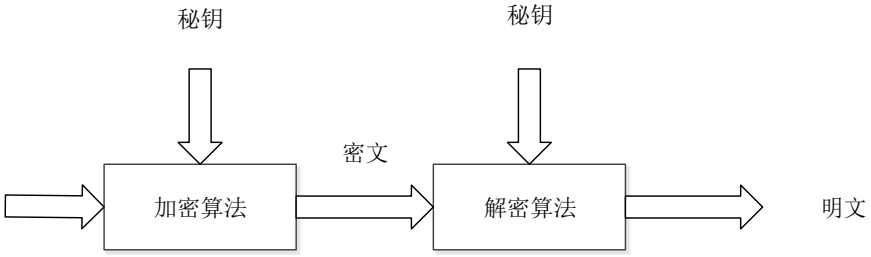



图 2-1 分组密码加解密模型

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$F_2$是二元域，$F_2^s$表示$F_2$上的 s 维向量空间。假定明文空间和密文空间均为$F_2^m$，密钥空间$S_k$是$F_2^t$的一个子集。m 是明文和密文的分组长度，t 是密钥长度。

### 2.1.3 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，已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。

#### 2.1.3.1 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：

$$L_i = R_{i-1} \tag{2.1}$$

$$R_i = L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \tag{2.2}$$

$K_i$ 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：

$$L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \oplus F\left(R_{i-1}, K_i\right) = L_{i-1} \tag{2.3}$$

因此只要在每轮中对 F 的输入重新构造，使用了该结构的密码算法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。

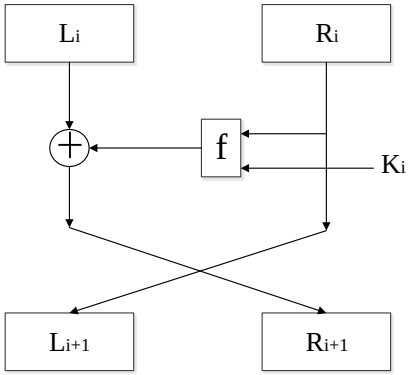



图 2-2 Festel 网络型结果

- DES 密码算法

DES 算法是 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 $K_i$ 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。

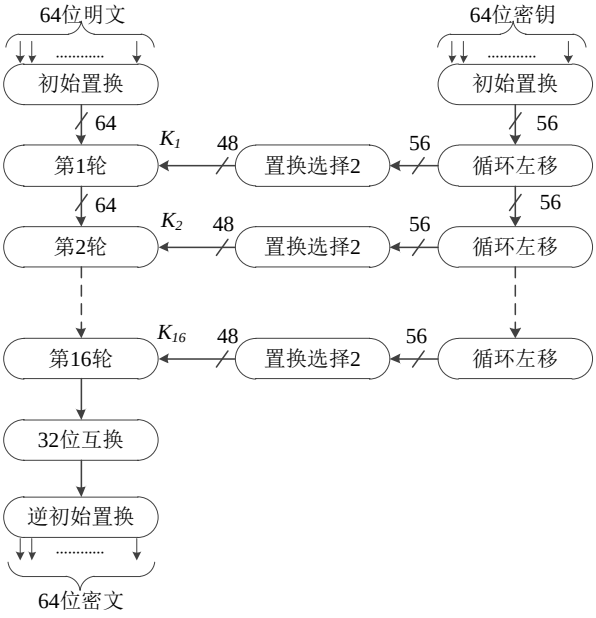



图 2-3 DES 加密算法的总体描述

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 2-1(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（2.4）和式（2.5）：

$$L_i = R_{i-1} \tag{2.4}$$

$$R_i = L_{i-1} \oplus F\left(R_{i-1}, K_i\right) \tag{2.5}$$

轮密钥 $K_i$ 长 48 位，R 为 32 位。首先将 R 按照表 2-1(c)定义的置换扩展为 48 位数据，其中有 16 位内容是重复的。将输出的 48 位数据与 $K_i$ 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。

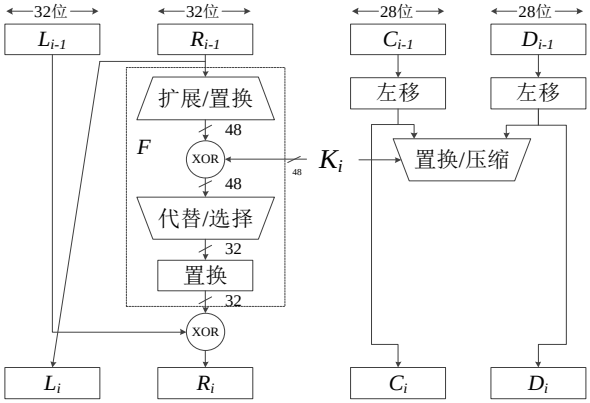



图 2-4 DES 算法一轮迭代的过程

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 $S_i$ 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。

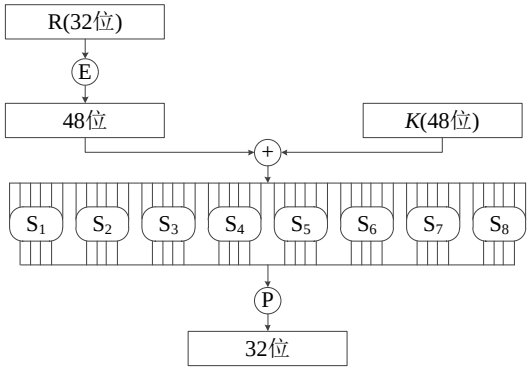



图 2-5 DES 加密算法中的 F 函数

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 |

#### 2.1.3.2 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。

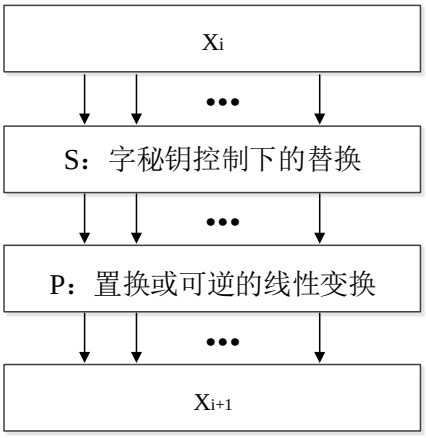



图 2-6 SP 网络型结构

- AES 算法

AES 算法是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。

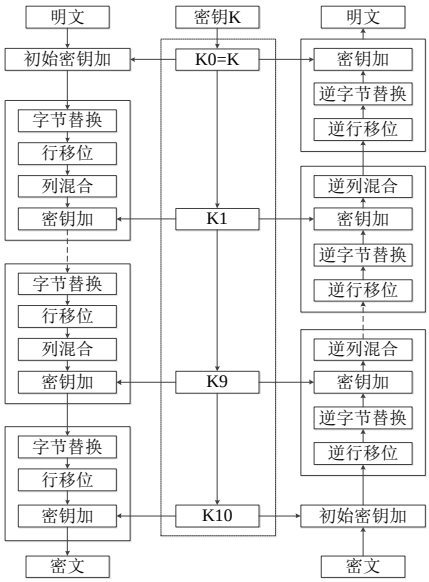



图 2-7 AES 加密算法的执行流程

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，$S_{1,1}$ 通过字节替换操作，转换为了 $S'_{1,1}$。

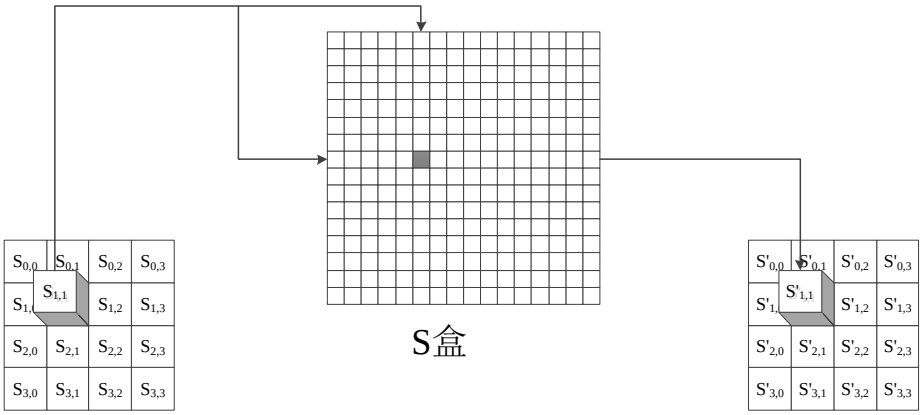



图 2-8 AES 字节替换

AES 定义了一个 S 盒，见表 2-3，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。

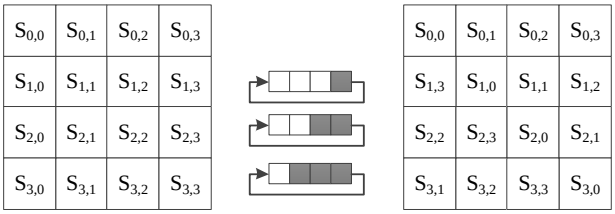



图 2-9 AES 的行移位变换

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($2^8$)上的元素。这样 128bit 的输入被当成了有限域 GF($2^8$)上的一个 4×4 方阵。视矩阵的一列为有限域 GF($2^8$)上的 4 维列向量，然后使用有限域 GF($2^8$)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($2^8$)上的一次矩阵乘法。

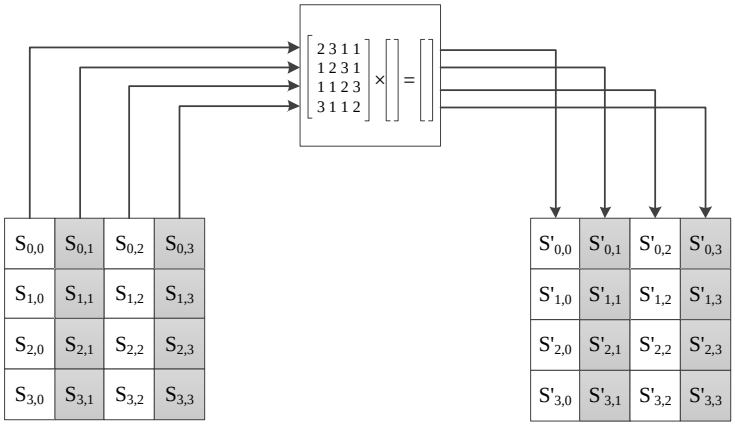



图 2-10 AES 的列混合变换

也就是通过左乘矩阵实现列混合，参见式（2.6），通过化简可以得到式（2.7）：

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,0} & s_{1,0} & s_{1,0} \\ s_{2,0} & s_{2,0} & s_{2,0} & s_{2,0} \\ s_{3,0} & s_{3,0} & s_{3,0} & s_{3,0} \end{bmatrix} \tag{2.6}$$

$$\begin{aligned} s'_{0,j} &= (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j}) \\ s'_{3,j} &= (3 \bullet s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \bullet s_{3,j}) \end{aligned} \tag{2.7}$$

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。

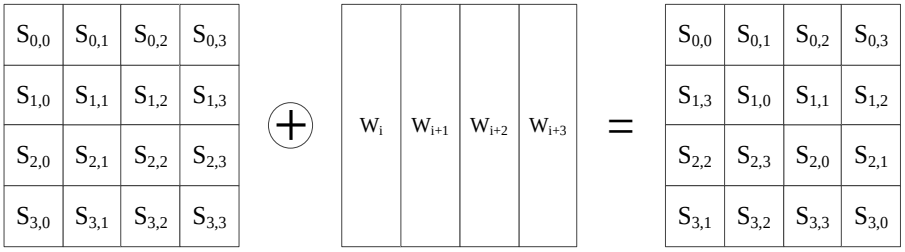



图 2-11 AES 的密钥加操作

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。

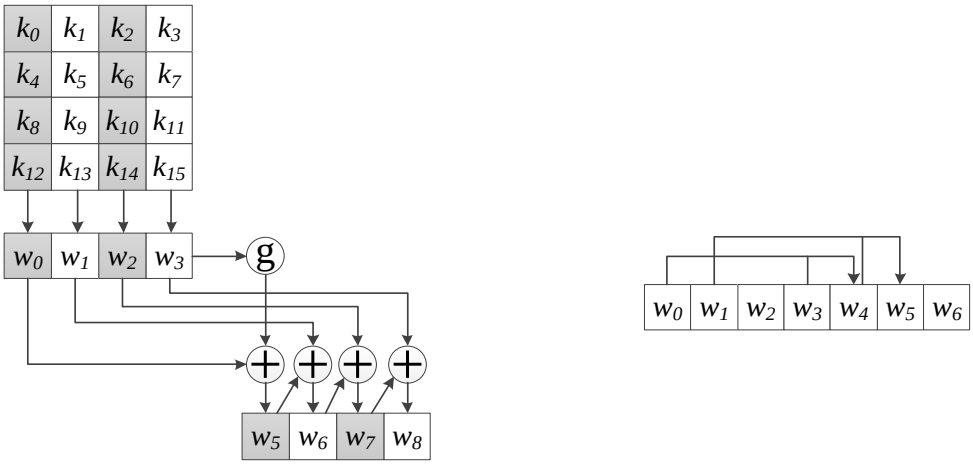



图 2-12 AES 密钥扩展

密钥扩展可以总结为式（2.8）：

$$\begin{aligned} &w_0 \oplus g(w_1) = w_4 \\ &w_1 \oplus w_4 = w_5 \\ &w_2 \oplus w_5 = w_6 \\ &w_3 \oplus w_6 = w_7 \\ &w_4 \oplus g(w_7) = w_8 \end{aligned} \tag{2.8}$$

其中的 g 函数可以参见图 2-13 所示。

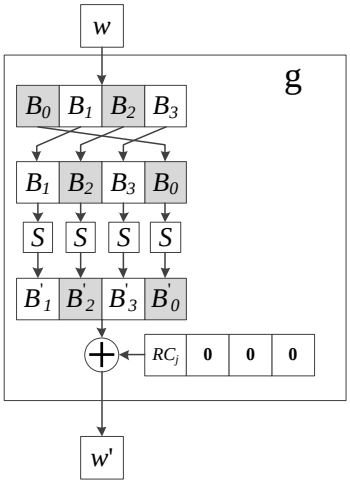



图 2-13 AES 密钥扩展中 g 函数结构

### 2.1.3.3 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。

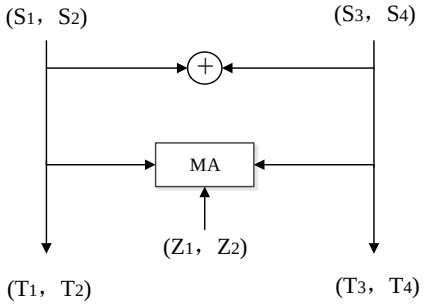



图 2-14 LM 型结构中的轮函数

- 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，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 个子密钥进行运算。

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 $2^{16}+1$ 乘；

（2）X1 和第二个子密钥（K2） 模 $2^{16}$ 加；

（3）X2 和第三个子密钥（K3） 模 $2^{16}$ 加；

（4）X4 和第四个子密钥（K4） 模 $2^{16}+1$ 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 $2^{16}+1$ 乘；

（8）将第（6）步和第（7）步的结果模 $2^{16}$ 加；

（9）将第（8）步的结果与第六个子密钥 模 $2^{16}+1$ 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；

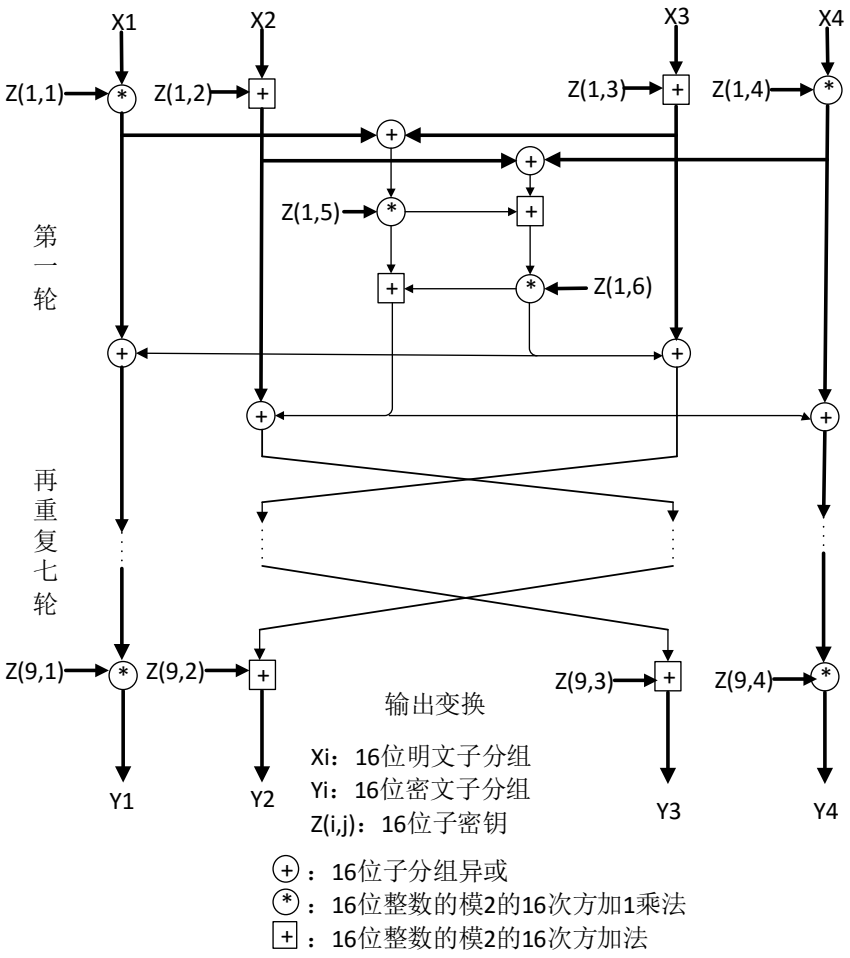



图 2-15 IDEA 算法流程

（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)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 $2^{16}+1$ 乘。

（2）X1 和第二个额外子密钥模 $2^{16}$ 加。

（3）X2 和第三个额外子密钥模 $2^{16}$ 加。

（4）X3 和第四个额外子密钥模 $2^{16}+1$ 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。

#### 2.1.3.4 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。

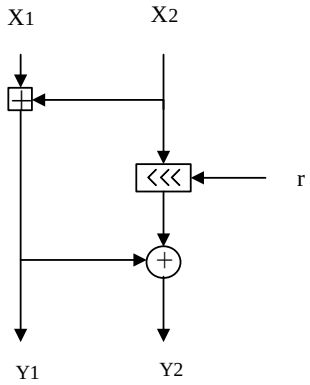



图 2-16 ARX 结构

- SPECK 算法

SPECK 密码算法 82[21]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。

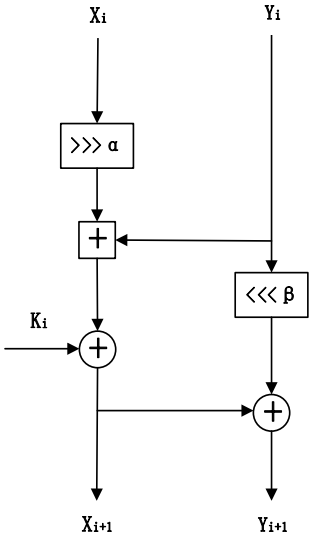



图 2-17 SPECK 算法的第 i 轮加密过程

设 $X_i$ 和 $Y_i$ 为第 i 轮的输入分组，$X_{i+1}$ 和 $Y_{i+1}$ 为第 i 轮的输出分组，$K_i$ 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 $X_i$ 向右循环移 α 位，然后与右端输入 $Y_i$ 进行模加运算，所得结果与子密钥 $K_i$ 进行异或运算，得到左端的输出 $X_{i+1}$；右端输入 $Y_i$ 向左循环移 β 位，再与左端输出 $X_{i+1}$ 进行异或运算，得到右端的输出 $Y_{i+1}$，这样就完成了整个轮函数。计算公式如式(2.9)所示：

$$\begin{aligned} \mathrm{X}_{i+1} &= ((\mathrm{X}_i >>> \alpha) + \mathrm{Y}_i) \oplus \mathrm{K}_i \\ \mathrm{Y}_{i+1} &= (\mathrm{Y}_i <<< \beta) \oplus \mathrm{X}_{i+1} \end{aligned} \tag{2.9}$$

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 $k_i$，设 K=（$l_{m-2}$，…，$l_0$，$k_0$）是算法的主密钥，其中 $l_0$，$k_0$ 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 $k_0$，$k_1$，…，$k_{T-1}$。计算 $k_i$ 和 $l_i$ 的公式(2.10)：

$$\begin{aligned} \mathrm{l}_{i+m-2} &= (k_i + (l_i >>> \alpha)) \oplus i \\ \mathrm{k}_{i+1} &= (\mathrm{k}_i <<< \beta) \oplus \mathrm{l}_{i+m-2} \end{aligned} \tag{2.10}$$

## 2.2 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。

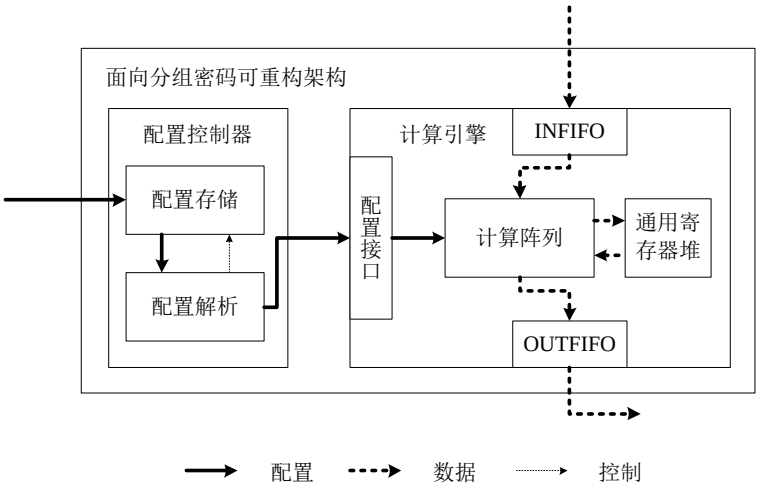



图 2-18 面向分组密码算法的可重构架构总体框图

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。

### 2.2.1 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图如图 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。

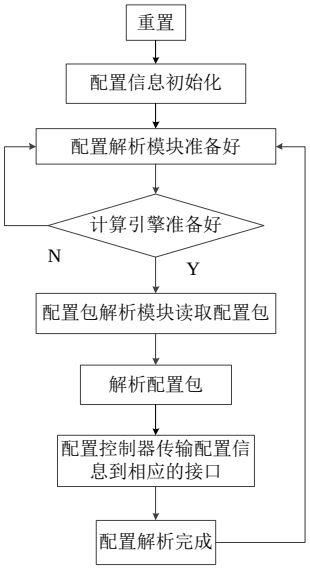



图 2-29 配置控制器工作流程图

### 2.2.2 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。

#### 2.2.2.1 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。

#### 2.2.2.2 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。

#### 2.2.2.3 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。

#### 2.2.2.4 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。

## 2.3 本章小结

本章对分组密码算法进行了简单的介绍，包括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。

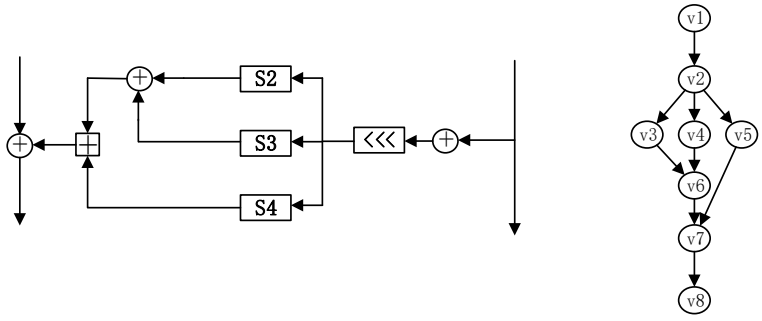
# 第三章 算法建模与算子特征提取

## 3.1 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以对算法进行人工观察统计的方式来获取算法特征，进而指导架构设计，最后通过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次序信息，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[22]（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。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），另一种是顶点活动网络[23]（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运算属性，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。




(a)算法流程图　　(b)AOV表示

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。

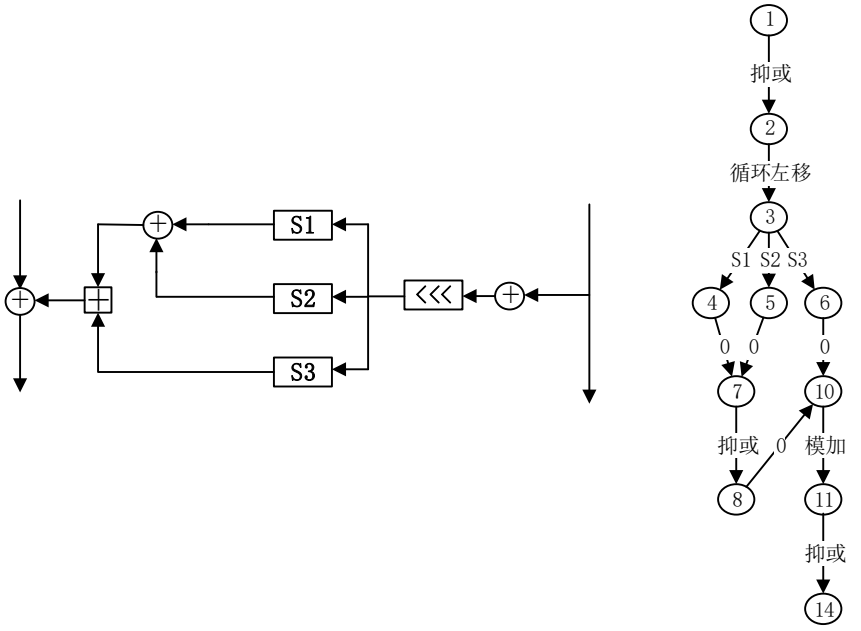
如图 3-1 所示，图中(a)是算法中的数据流程图的一部分，它标示了算法的各种操作以及数据方向，(b)则是与数据流图对应的算法图模型。表 3-1 列出了对应节点的属性，其中操作属性中，xor 表示异或，sh 表示移位，lut 表示查找表，au 表示算术，通过给(b)的节点赋予这些属性，(b)就能够描述算法。

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用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，用来获取算子的次序信息。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，进而提高映射效率；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在 AOE 网络上进行的，AOE 网络的有向图中，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。




(a)算法流程图　　(b)AOE表示

图 3-2 算法图的 AOE 表示

## 3.2 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数学理论和结构模型，涉及的操作类型有较大的交集。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、有限域乘法以及算术运算等。

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 设计时需要参考的关键特征：

- 模式特征

模式特征主要指算法的一些基本特征和基本操作的功能模式特征，比如算法的分组大小、轮数、密钥长度，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。

- 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度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时，对于算法映射实现，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。

- 次序特征

次序特征是指算子在算法轮函数中的出现的次序；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放置置换单元。在架构设计时，根据次序特征，在需要的位置放置这些算子对应的功能单元，可以有效减少功能单元的数量

### 3.2.1 模式特征

模式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。这些模式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[24]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 3-2 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 3-2 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模式，统计不同算子的使用频率

和对应的模式特征，具体如表 3-3 所示。

表 3-3 基本运算操作的模式特征

| 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 | 任意 | 36 | 位宽兼容 |
| | 非 | 任意 | 0 | |
| | 或 | 任意 | 1 | |
| | 与 | 任意 | 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 $2^{16}$ |
| | | 32 | 14 | 模 $2^{32}$ |
| | 模减 | 32 | 1 | 模 $2^{32}$ |
| | 模乘 | 16 | 1 | 模$(2^{16}-1)$ |
| | | 32 | 1 | 模 $2^{32}$ |
| 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$(2^8)$ |
| 置换 | | / | 6 | 输入-输出（64-64、32-48、32-32、32-40、128-128） |

通过表 3-3 可以发现分组密码算法的基本算子具有如下模式特征：

(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，取摸操作中的模数都是 2 的幂次，多为 $2^8$、$2^{16}$、$2^{32}$。

(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、4-4、8-8、8-32、6-2、10-8；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($2^8$)上，而且一般都是以矩阵乘法的形式出现。

### 3.2.2 组合特征

模式特征关注的是单个功能单元的不同运算模式，组合特征则关注多个功能之间的关系，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之前首先需要提取算法流程中的关键路径，关键路径上的算子组合是本文的关注点。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。

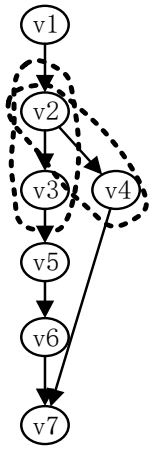



图 3-3 组合选择冲突示例

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[25]。

定义 Ve[i]为每一个点的最早发生时间；Vl[i]为每一个点的最迟发生时间；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 3-4 列出了部分算子的前后缀组合特征，其中 SH 表示移位算子，AU 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 表示按字节循环移位算子。

功能组合一般不超过 3 级，3 级以上的组合往往延迟很大而且出现的概率也相对较低，表 3-4 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、算术算子-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 3-4 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-AU | 20% | XOR-SH | 67% | AU-XOR-SH | 27% |
| | SH-AU | 33% | SH-XOR-SH | 13% | AU-SH | 27% | XOR-LUT-SH | 20% |
| | SH-AND | 13% | SH-AU-XOR | 13% | LUT-SH | 27% | XOR-XOR-SH | 13% |
| AU | AU-XOR | 75% | AU-AU-XOR | 31% | XOR-AU | 56% | XOR-SH-AU | 25% |
| | AU-AU | 31% | AU-XOR-XOR | 25% | SH-AU | 31% | AU-XOR-AU | 25% |
| | AU-SH | 25% | AU-XOR-SH | 25% | AU-AU | 31% | XOR-AU-AU | 19% |
| LUT | LUT-XOR | 37% | LUT-XOR-BR | 19% | XOR-LUT | 78% | XOR-XOR-LUT | 26% |
| | LUT-BR | 32% | LUT-SH-XOR | 11% | AU-LUT | 11% | XOR-AU-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% |

### 3.2.3 次序特征

次序特征指算子在算法轮函数中出现前后次序的统计特征，在架构设计时，根据次序特征，在需要的位置放置这些算子对应的功能单元，可以有效减少功能单元的数量。

在论文 2.2 节中介绍了整个可重构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，可以通过提取这些算子在轮函数中的次序特征来决定这些算子对应的功能单元在异构组中的位置分布。在算法建模时算法中的每一个算子都有与横纵次序相关的信息（x，y），相当于一个算子的坐标系。将分组密码算法的轮函数简单分为三个连续阶段，6 类算子表现出如表 3-6 所示的次序特征。

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；对于 S 盒，在中间操作上出现 28 次，终结操作上出现 2 次，不出现在初始操作中；对于有限域乘法，在终结操作上出现 10 次，中间操作上出现 1 次，不出现在初始操作中；算术操作、移位操作、逻辑操作出现的次序比较平均，没有明显的次序倾向性。

表 3-6 算子在算法轮中出现的次序特征

| 算子 | 操作位置 | 出现次数 | 算子 | 操作位置 | 出现次数 |
|---|---|---|---|---|---|
| 算术操作 | 初始操作 | 8 | S 盒 | 初始操作 | 0 |
| | 中间操作 | 15 | | 中间操作 | 28 |
| | 终结操作 | 10 | | 终结操作 | 2 |
| 移位操作 | 初始操作 | 9 | 有限域乘法 | 初始操作 | 0 |
| | 中间操作 | 12 | | 中间操作 | 1 |
| | 终结操作 | 4 | | 终结操作 | 10 |
| 逻辑操作 | 初始操作 | 26 | 置换 | 初始操作 | 3 |
| | 中间操作 | 10 | | 中间操作 | 0 |
| | 终结操作 | 16 | | 终结操作 | 4 |

## 3.3 本章小结

本章首先选取了 36 个比较常用的分组密码算法作为研究对象，为分组密码算法建立了统一的图分析模型，基于这个图分析模型，分别对这些分组密码算法算子的模式特征、组合特征和次序特征进行了总结，分析了这 3 类特征和可重构阵列架构设计的关系。

# 第四章 PE 设计方案

## 4.1 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、功能单元类别、互连方式等。第三章中对分组密码算法的 3 类特征进行了总结，这 3 类统计特征将成为本章 PE 方案的设计依据。

本文致力于在保证架构的高算法支持度和高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功能单元冗余设计。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单元；但是对于具体算法而言，某一种功能单元只会出现有限的几次，架构中的这些功能单元在实际的映射中使用率很低，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：

1） 在确定初始架构时对于次序信息比较明显的算子，只在有明确需求的位置上放置对应功能单元；

2） 对于次序特征不明显的功能单元，在制定初始架构方案时不进行位置确定，如传统的同构架构一样，所有的 PE 都包含这些功能单元，进一步的优化在算法映射后根据大量算法映射后的功能使用分布情况，再对初始架构存在的冗余功能进行优化。

## 4.2 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。每个异构组包含 3 个 PE 行和 3 个互连结构，每个 PE 行包含 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使算法达到最大性能，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，因此本文中的架构采用 4 路并行 PE 的设计方案。

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构成，与算法的轮函数迭代正好一一对应。有些轮函数较复杂，无法在一个异构组中完成，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设计就可以把大多数算法约束到一个异构组中完成计算。

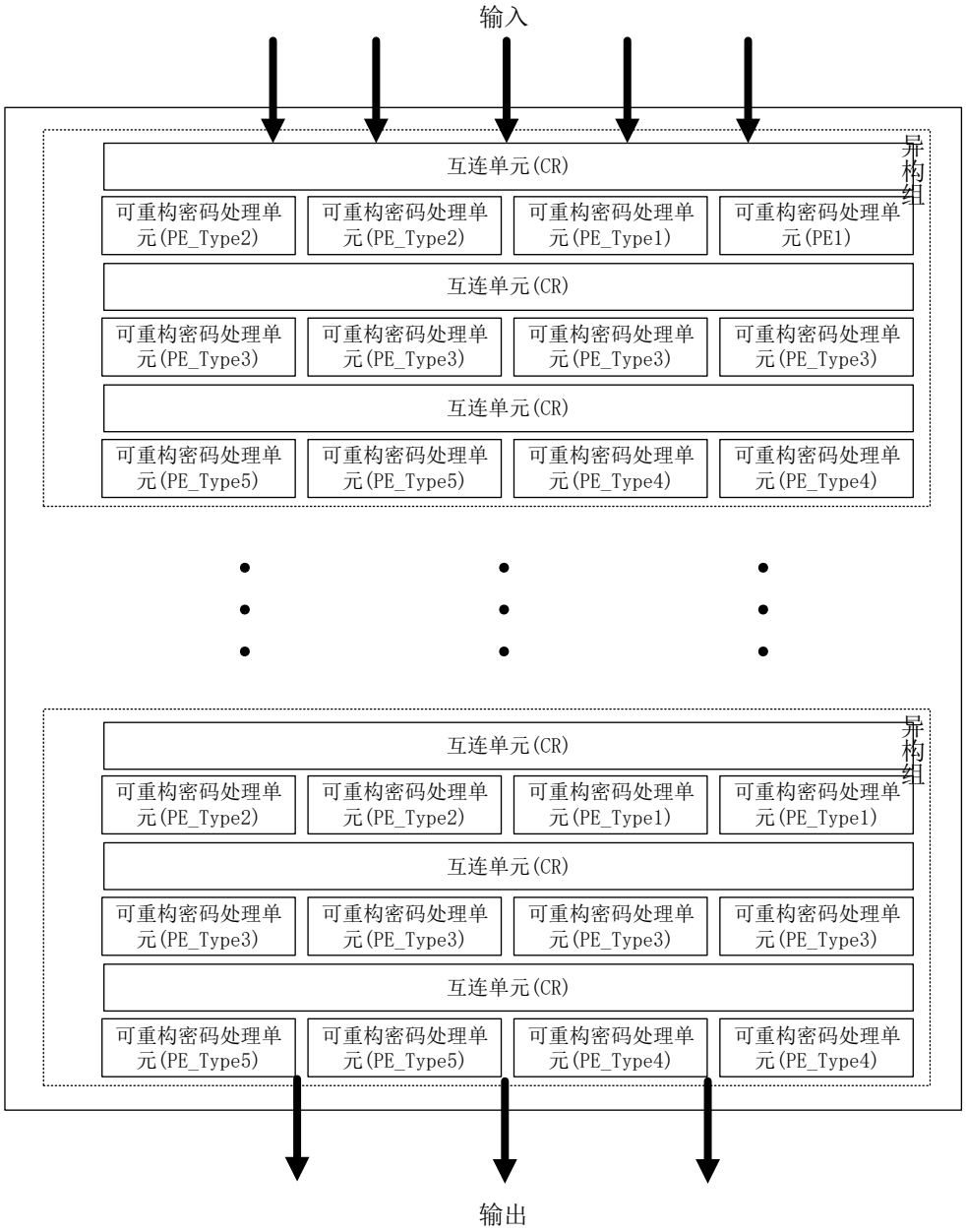



图 4-1 可重构密码架构阵列拓扑图

## 4.3 行间互连

可重构阵列互连单元主要完成连续 PE 行的数据互连。如**错误!未找到引用源。**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。加上第一层，整个互连单元需要 112bit 配置信息。

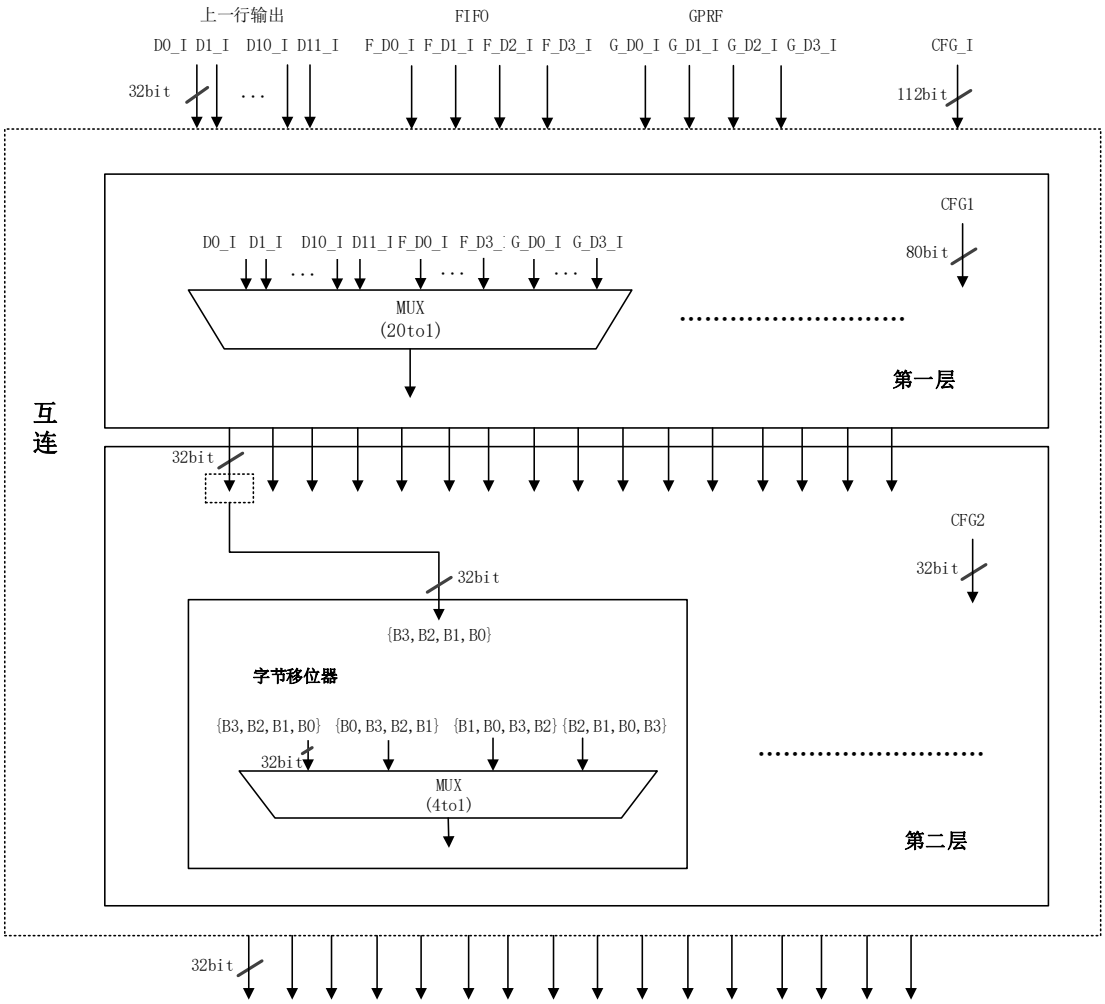



图 4-2 互联单元结构图

## 4.4 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。

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次序特征，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**错误!未找到引用源。**所示。

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密钥加操作；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，因此异构组中只有第 2 行包含了 S 盒查表单元。

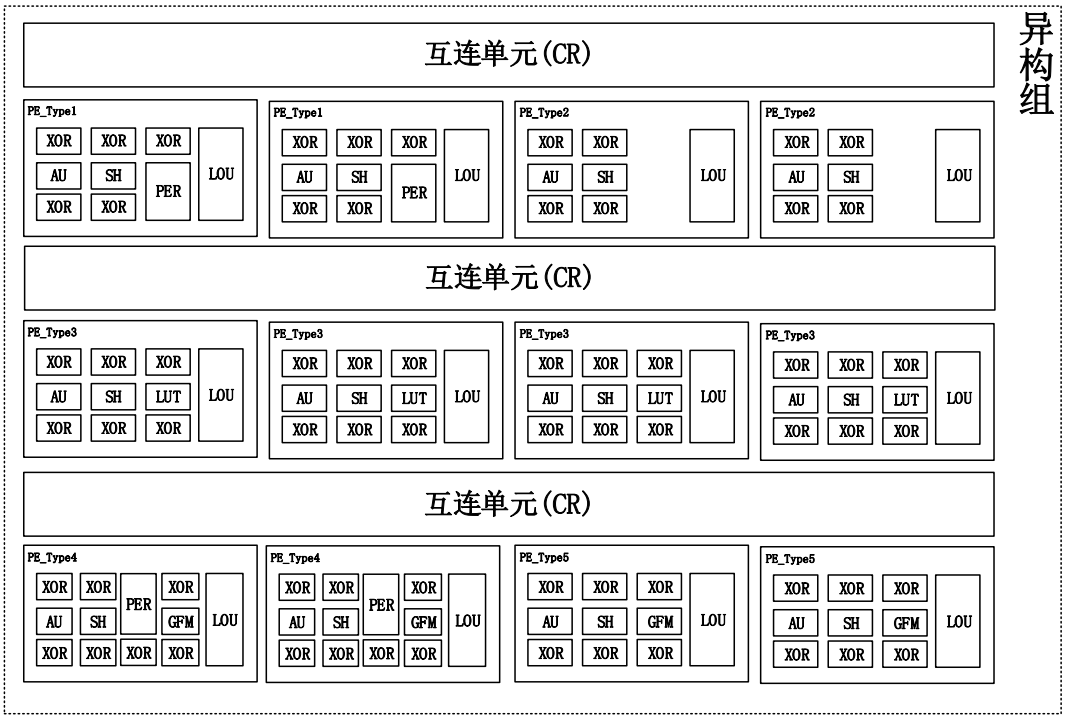



图 4-3 异构组功能拓扑图

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 S 盒查表操作行之后，对应于异构组中的第 3 行。

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 | | | √ | | |

如表 3-4 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。

## 4.5 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。

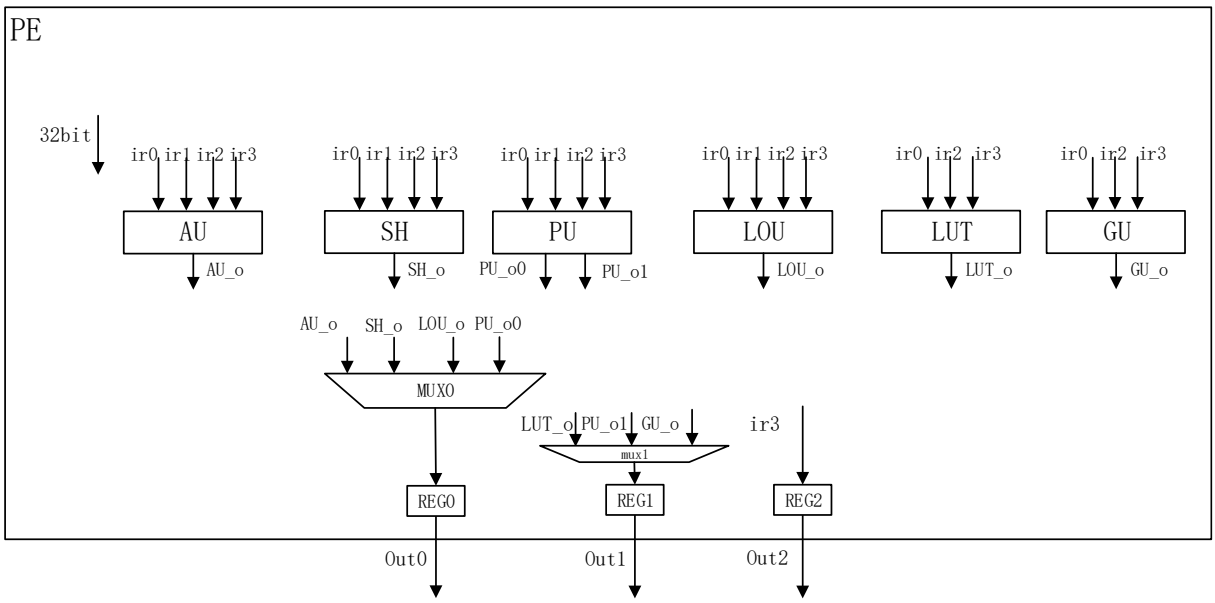



图 4-4 带全功能的处理单元

图 4-4 是一个带全功能的处理单元，根据 4.4 中对异构组的描述可知，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。

处理单元中有 4 个 32 位的输入：in0、in1、in2、in3，这些输入来自上一行的互连输出，传输上一行处理后的数据。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；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 MUX 来选择输出。

## 4.6 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。

### 4.6.1 逻辑单元

第三章中算法分析总结的逻辑操作算子特征如表 4-2 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的算法使用，没有算法使用了非逻辑操作。

表 4-2 逻辑操作统计特征

| 逻辑操作 | 使用频率 | 逻辑操作 | 使用频率 |
|---|---|---|---|
| XOR | 100.00% | AND | 8.30% |
| NOT | 0.00% | XOR->XOR | 30.56% |
| OR | 2.80% | 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**错误!未找到引用源。**所示，逻辑单元中对异或操作进行了可选择的三级级联。

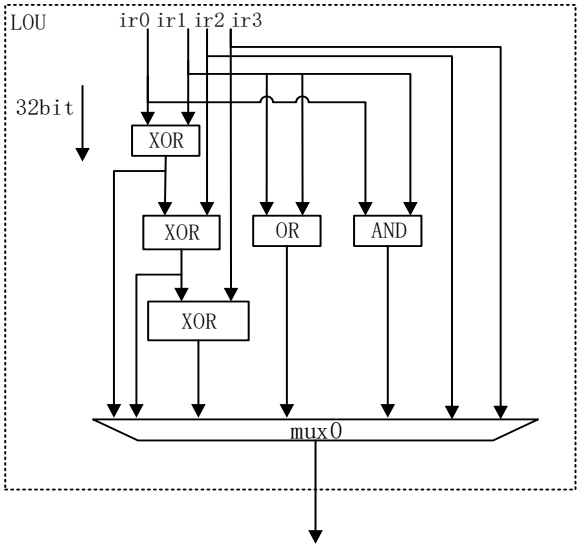



图 4-5 逻辑单元结构

如**错误!未找到引用源。**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 4-所示。

表 4-3 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 |

### 4.6.2 S 盒替代单元

S 盒是许多分组密码算法中唯一的非线性部件，提供了算法所需要到混乱作用。分组算法中常常需要并行查找相同或不同的 S 盒，S 盒代替的输入粒度差别较大，以不大于 8 位者居多；S 盒的大小也很不相同，从 512bit 到 80kbit。第三章对大量的分组密码算法分析可以得出 S 盒查找表的输入输出常见有 4-4、6-4、8-8、8-32、6-2、10-8 六种模式。表 4-1 列出了不同 S 盒查找表模式的使用频度。

表 4-1 S 盒输入输出模式使用频度表

| 输入输出模式 | 使用频度 | 输入输出模式 | 使用频度 |
|---|---|---|---|
| 4-4 | 14.8% | 8-32 | 3.7% |
| 6-4 | 3.7% | 6-2 | 3.7% |
| 8-8 | 70.4% | 10-8 | 3.7% |

根据表 4-2 的数据可知 4-4、6-4、8-8、8-32、6-2 五种查表模式占到了整个 S 盒查找表方式的 96.3%，对于 10-8 模式，无法与前面五种模式进行兼容，而且占用比例很小，因此本文只对前面五

种模式提供支持。

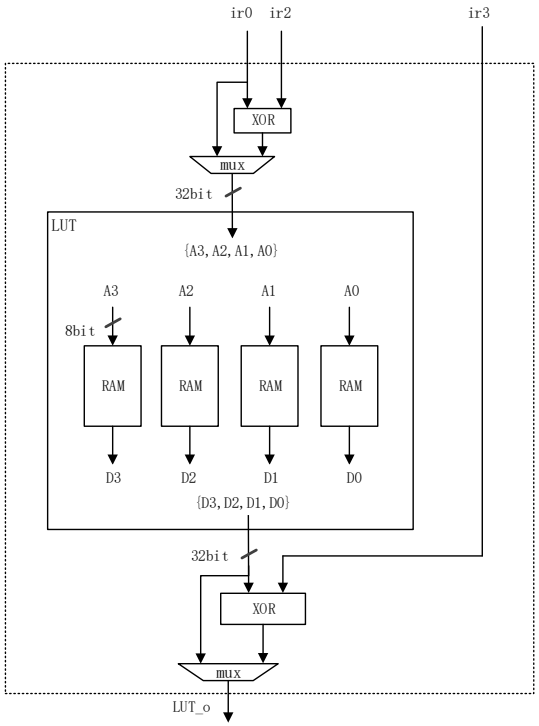



图 4-6 S 盒查找表结构

根据上面对 S 盒的模式研究，提出如**错误!未找到引用源。**所示的 S 盒结构。使用了四个 $2^8 \times 8$ RAM 并联电路,查表前后支持异或逻辑操作，32 位输入数据被拆分成 4 个 8bit 位的地址位，输出的 4 个 8bit 数据又重新组合成一个 32bit 数据输出。

### 4.6.3 算术单元

算法分析表明，47.22%的算法包含算术运算，其中有 78.2%是 32 位运算，紧跟模 $2^{32}$ 操作；11.2%是 16 位运算，紧跟 $2^{16}$ 运算；10%是 8 位运算，紧跟 $2^8$ 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。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。

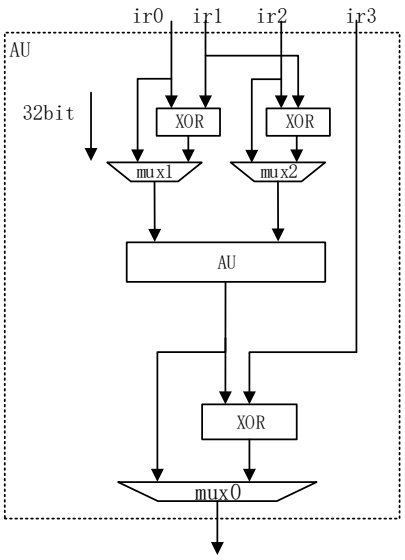



图 4-7 算术单元结构

算术单元中有 3 个 2 输入 MUX，需要 3 位配置，AU 需要两位配置；共需要 5 位配置信息。如表 4-3 所示，最高 2 位配置信息控制 AU 的模式，余下的 3 位配置信息控制 3 个 2 输入 MUX 完成异或操作选择。

表 4-3 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 $2^8$ |
| 00 | 0 | 0 | 1 | ((ir0⊕ir1)+ir2) modulo $2^8$ |
| 00 | 0 | 1 | 1 | ((ir0⊕ir1)+ (ir1⊕ir2)) modulo $2^8$ |
| 01 | 0 | 0 | 0 | (ir0+ir2) modulo $2^{16}$ |
| 01 | 0 | 0 | 1 | ((ir0⊕ir1)+ir2) modulo $2^{16}$ |
| 01 | 0 | 1 | 1 | ((ir0⊕ir1)+ (ir1⊕ir2)) modulo $2^{16}$ |
| 10 | 0 | 0 | 0 | (ir0+ir2) modulo $2^{32}$ |
| 10 | 0 | 0 | 1 | ((ir0⊕ir1)+ir2) modulo $2^{32}$ |
| 10 | 0 | 1 | 1 | ((ir0⊕ir1)+ (ir1⊕ir2)) modulo $2^{32}$ |

### 4.6.4 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 位数据或 1 个 64 位数据。置换还可以完成某些不容易实现的操作，比如移位操作、复杂逻辑操作等。

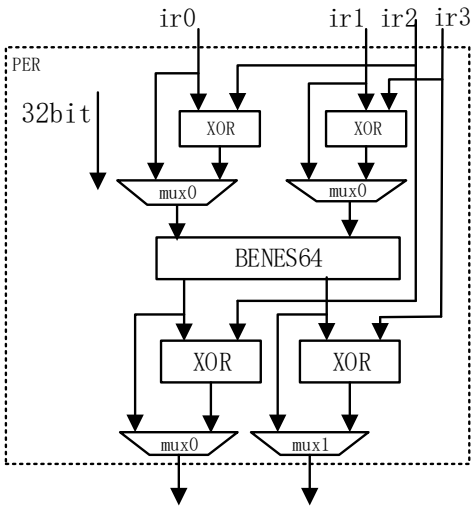



图 4-8 置换单元

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（4.1）所示，64 位 BENES 所需的配置为 352 位，因此整个置换单元共需 352+4=356 位配置信息。

$$BENES(N).length = \frac{2*(2*log2\ (N)-1)}{N} \tag{4.1}$$

### 4.6.5 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%的算法的移位运算是 8 比特的，因此本文设计的移位单元只支持 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 4-9 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。

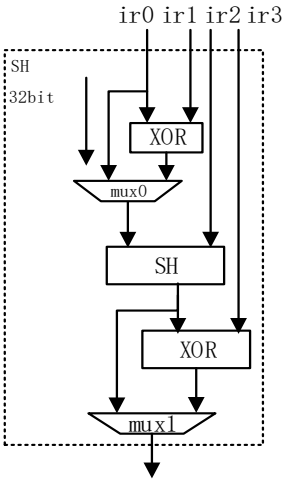



图 4-9 移位单元结构

整个数据通路需要 4 比特的控制信息，具体的含义如表 4-4 所示，两 bit 控制移位单元的移位模式，两 bit 控制两个 2 输入 MUX 完成前后异或的组合功能选择。

表 4-4 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 |

### 4.6.6 有限域乘法单元

有限域乘法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。这些算法中出现的有限域都是 GF($2^8$)域，因此本文中的有限域乘法单元只支持 GF($2^8$)上的乘法运算。

- 有限域 GF($2^n$)上的乘法

在 GF($2^n$)上，我们设 f(x)是不可约多项式，也被称为域多项式，GF($2^n$)中的元素可以看作是小于 n 次的多项式。设 GF($2^n$)上的两个元素为：

$$a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \tag{4.2}$$

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_1x + b_0 \tag{4.3}$$

定义两者的乘积为：

$$c(x) = a(x)b(x) \bmod f(x) = c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + ... + c_1x + c_0 \tag{4.4}$$

其中： $f(x) = x^n + f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + ... + f_1x + f_0$

根据有这个乘法定义，GF($2^n$)上的乘法本质上是多项式中对应的系数进行运算，目前实现的基本方法是采用“Shift-and-add”算法[26][27]，该算法执行多次移位，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 4-1010 所示：

根据有限域 GF($2^8$)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 GF($2^8$)上的乘法，多项式 a(x)乘以 x 为：

$$r(x) = (a_7x^7 + a_6x^6 + ... + a_1x + a_0) \cdot x \bmod (x^8 + f_7x^7 + f_6x^6 + ... + f_1x + f_0) \tag{4.5}$$

由式（4.5）可得：

$$r(x) = (a_7f_7 + a_6)x^7 + (a_7f_6 + a_5)x^6 + (a_7f_5 + a_4)x^5 + ... + (a_7f_1 + a_0)x + a_7f_0 \tag{4.6}$$

| 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)<br>Output：c(x)=a(x)b(x) mod f(x)<br>c = 0<br>for i=n-1 to 1 do<br>　　if $a_i$=1 then c=c+b<br>　　c=c·x mod f(x)<br>if $a_0$=1 then c=c+b<br>return (c) |

图 4-10 Shift-and-add 算法描述

由式（4.6）可获得任意不可约多项式的 GF($2^8$)域的乘法电路。将 x 乘法电路依次串联起来，便可得到 GF($2^8$)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再异或，便组成了 GF($2^8$)乘法矩阵运算电路[28]。

- 电路结构

GF($2^8$)域上的乘法单元电路结构如图 4-111 所示。电路分为两部分，分别是 GF($2^8$)域矩阵乘法

电路和静态配置寄存器。GF($2^8$)域矩阵乘法电路模块负责完成 GF($2^8$)域上的矩阵乘法运算工作。静态配置寄存器的功能是完成 GF($2^8$)域上乘数多项式及不可约多项式信息的存储工作。

分组密码算法实际应用中，GF($2^8$)域上的乘法通常会用矩阵乘法形式表示，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。对分组密码算法的分析发现，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=544 比特。

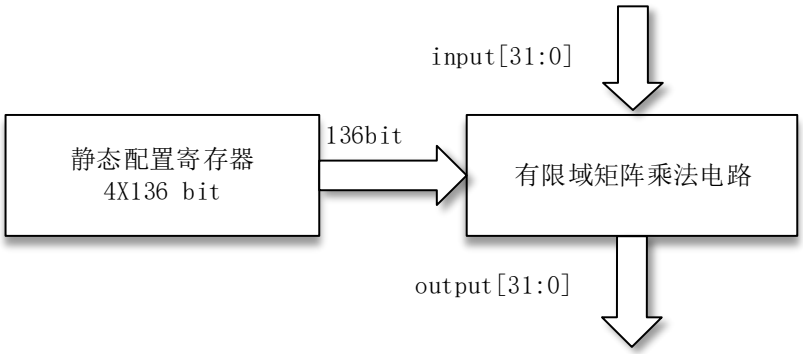



图 4-11 GF($2^8$)域乘法电路整体结构

根据公式（4.6）可以得到对于任意不可约多项式 GF($2^8$)域上的 x 乘法电路。如图 4-122 所示。

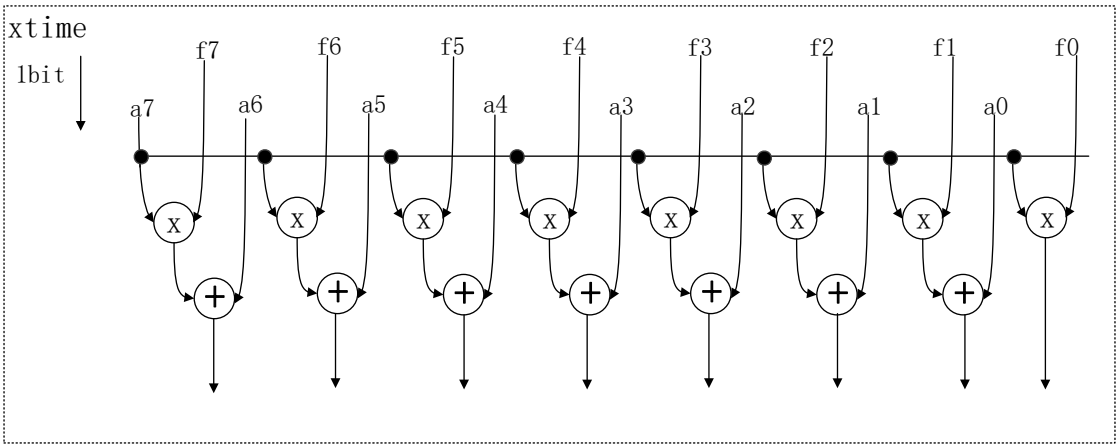



图 4-12 GF($2^8$)域 x 乘法电路

其中（f7,…,f1,f0）是不可约多项式的低 8 位，将图 4-122 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 4-143 所示的 GF($2^8$)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。

在图 4-123 中，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。从图 4-13 的单元整体结构图可以知道，a 是操作数，来自运算数据，b 和 f 是被乘数和不可约多项式，从配置端输入，因此该电路结构可以完成 GF($2^8$)域的任意不可约多项式的乘法。

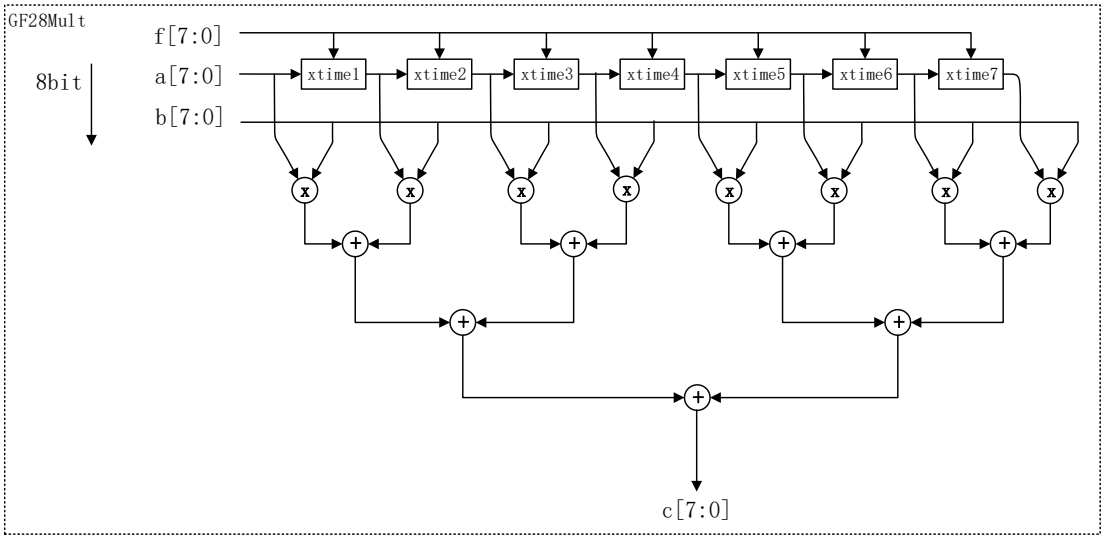



图 4-14 基本有限域乘法电路

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 4-154 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 比特运算。进一步，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 4x1 矩阵乘法运算。

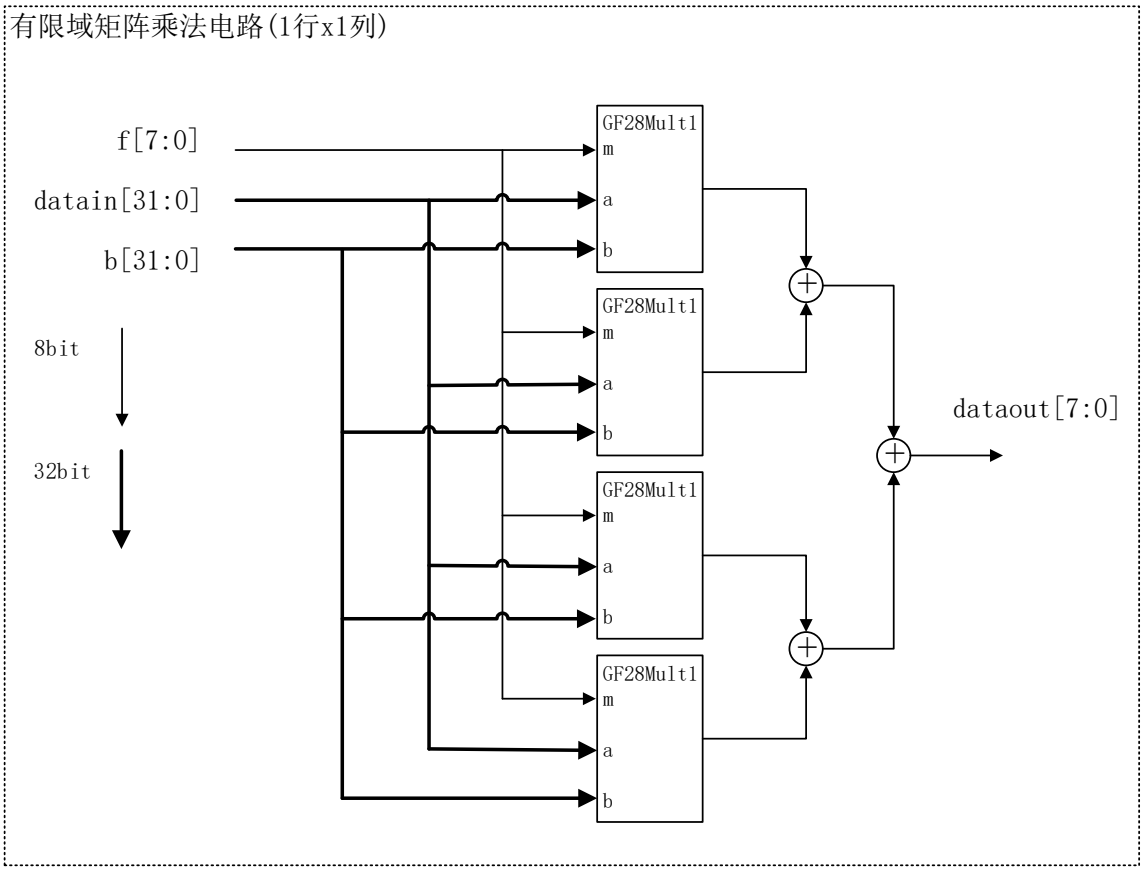



图 4-15 有限域矩阵乘法电路(1 行 x1 列)

## 4.7 本章小结

本章在第三章总结的算法特征的基础上，对架构中 PE 设计提出了一个初始的方案。这个初始方

案确定了可重构密码处理器的计算阵列设计，计算阵列由异构组重复迭代构成，因此重点讨论了单个异构组的设计，包括异构组中的 5 类运算单元设计，二级结构带字节循环移位功能的互连单元设计以及 6 类功能（算术、移位、逻辑、S 盒替换、置换、有限域乘法）设计。根据 S 盒替换、置换、有限域乘法这三类算子在算法特征分析中表现出来的次序特征，对这三种运算单元进行了分布优化，减少了单元个数；对于加法、移位、逻辑操作，统计特征不明显，进行了全包含设计，进一步的优化会在第五章的映射分析之后进行。

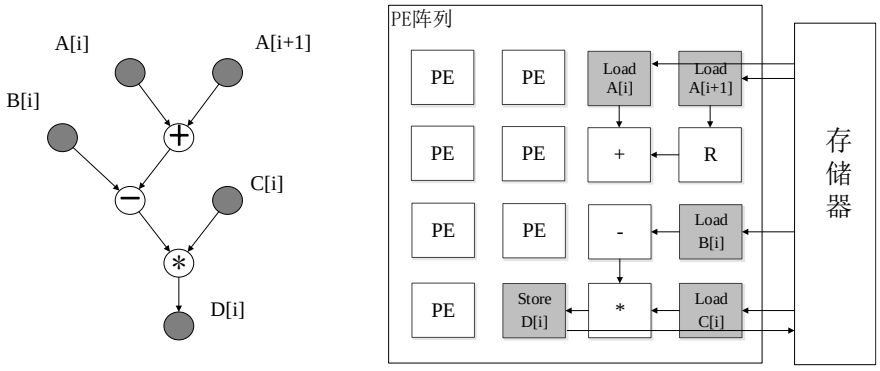
# 第五章 算法映射分析与 PE 方案优化

## 5.1 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，当应用集合增大时，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。与之对比，拥有高效的自动化映射工具不仅可以快速高效地完成大量应用的映射实验，并且允许架构设计人员对架构进行微调测试，根据测试结果反馈到架构设计，对架构进行优化调整。因而设计高效的自动化映射工具是可重构架构设计的一个必要且关键的问题。

### 5.1.1 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 5-1 给出了一个应用到可重构阵列上的映射，数据先从存储器读入，在 PE 上完成运算，通过互连传递数据，直到获取最终结果然后存入存储器，图中的 R 表示该 PE 不进行运算，作为路由使用。




(a) 数据流图　　(b) 映射

图 5-1 应用到可重构阵列上的映射

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

每一个处理单元都包含多个功能单元，例如移位、算术、逻辑等。在进行应用映射时需要对应的处理单元能够提供应用节点的操作；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。

- 局部存储器

通常多个处理单元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。

分组密码算法作为一种特定的应用，密码可重构作为一种特定的可重构处理器，分组密码算法到密码可重构的映射具有自己的特点：

1) 可重构密码处理器阵列的互连是有导向的，加密数据从阵列的第一行输入，第一行运算后的输出结果又输入到下一行，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，将密码算法的轮函数迭代展开，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 的映射，PE 内部有很多组合功能设计，应用中的多个功能会被打包到一起映射到一个 PE 中；
5) 本文的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建模描述、运算功能匹配上会更加复杂。

### 5.1.2 研究现状

将一个循环体映射到可重构架构，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 PE 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：

1) 循环体的所有操作都在一个 PE 或一行 PE 内完成，不需要考虑 PE 间的数据互连；
2) 可以直接利用传统的程序编译方法完成循环的映射。

这类 CGRA 映射问题[29][30][31]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 中的存储共享方法。

空间映射的方法适用于可重构阵列架构，它将循环中的所有或部分操作一次性映射到 CGRA 的

PE 阵列上，每个 PE 绑定循环中的一个操作，整个循环执行的过程中只用配置一次，不需要再经过重新配置执行其它操作。空间映射的方法有以下特点：

1) 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；
2) 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；
3) 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[32][33]综合里的布局布线技术，但是 CGRA 本身和 FPGA 有很大不同，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上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，文献[34]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 [38] 引入分裂和外扩的方式[39]，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文献[40]中的子图同胚[41]以及 EPIMap[42]中的图满射技术。

本文中的可重构密码处理器是 PE 阵列架构，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。根据架构特征分析，本文将密码算法的映射问题归纳为子图同构问题的一个变种，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，然后再通过一定的选优规则找出资源使用最少的方案作为最终映射方案。

## 5.2 基于子图同构的映射方案

### 5.2.1 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$\mathrm{G}=(\mathrm{V},\mathrm{E},\alpha,\beta)$，其中：

1) $\mathrm{V}\neq\emptyset$ 称为 G 的顶点集，其元素称为顶点或结点；
2) E⊆V×V 称为 G 的边集，其元素称为边；
3) α：$\mathrm{V}\rightarrow\sum_V$ 为图 G 的顶点标记函数，说明顶点与其标记的对应关系；
4) β：$\mathrm{E}\rightarrow\sum_E$ 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 $G_1=(V_1,\mathrm{E}_1,\alpha_1,\beta_1)$ 为图 $G_2=(V_2,\mathrm{E}_2,\alpha_2,\beta_2)$ 的一个子图，记为 $G_1\subseteq G_2$，如果有：

1) $V_1\subseteq V_2$；
2) $E_1\subseteq E_2\cap(V_1\times V_1)$；
3) $\alpha_1(x)=\alpha_2(x)$，$\forall x\in V_1$；
4) $\beta_1((x,y))=\beta_2((x,y))$，$\forall(x,y)\in E_1$。

此时，我们也称图 $G_2$ 为图 $G_1$ 的一个超图。

**定义 5.3（图同构）：**图 $G_1=(V_1,\mathrm{E}_1,\alpha_1,\beta_1)$ 和图 $G_2=(V_2,\mathrm{E}_2,\alpha_2,\beta_2)$ 是同构的，记为 $G_1\cong G_2$，如果存在一个双射函数 $f$：$V_1\rightarrow V_2$，使得：

a) $\alpha_1(x)=\alpha_2(f(x))$，$\forall x\in V_1$；

b) $\beta_1((x,y)) = \beta_2((f(x),f(y))$，$\forall(x,y) \in E_1$。

这样的函数 $f$ 也称为图 $G_1$ 与 $G_2$ 的图同构。例如，图 5-2 中 A 和 B 两个图同构，顶点如图中的虚线箭头一一对应。

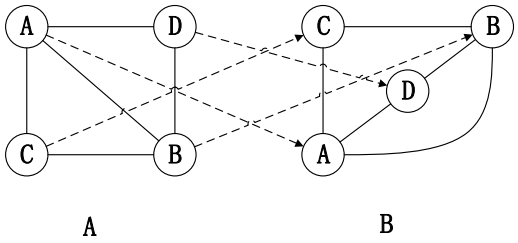



图 5-2 图同构的示例

**定义 5.4（子图同构）：**给定图 $G_1$ 与 $G_2$ 的一个图同构 $f$ 以及另一个图 $G_3$，如果$G_2 \subseteq G_3$，则 $f$ 为图 $G_1$ 与 $G_3$ 的一个子图同构。图 5-3 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。

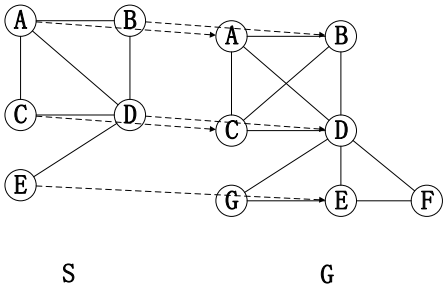



图 5-3 子图同构的示例

- 复杂度分析

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 $o(e^n)$，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上的算法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，架构图中的算法图候选集非常有限；
2) 架构图中每一个结点（PE）的输入输出很有限，而且只有相邻节点才有互连，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。

### 5.2.2 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[40]对这个问题进行了充分的研究。文献[41]提出了 Ullmann 算法，它是一个前向剪枝的带有回溯的树搜索过程；文献[42]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[41] | | Ullmann[42] | |
|---|---|---|---|---|
| 复杂度 | 最好情况 | 最坏情况 | 最好情况 | 最坏情况 |
| 时间 | $O(N^2)$ | $O(N!N)$ | $O(N^3)$ | $O(N!N^2)$ |
| 空间 | $O(N)$ | $O(N)$ | $O(N^3)$ | $O(N^3)$ |

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。M(s)代表中间状态，$M_1(s)$和 $M_2(s)$表示当前状态 s 的部分匹配中图 $G_1$ 和图 $G_2$ 中的点，整个算法流程如图 5-所示。

```
1)  PROCEDURE Match(s)
2)      输入：中间状态 s，初始状态 s0 满足
3)      输出：两个图之间的所有匹配
4)        IF M(s) 覆盖了 G2 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 s'
11)                   CALL Match(s')
12)               END IF
13)           END FOREACH
14)           恢复数据状态
15)       END IF
16) END PROCEDURE Match(s)
```

图 5-4 VF2 子图同构算法流程

初始化时状态是 $s_0$，$M(s_0)$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 M(s)包含了 $G_2$ 中的所有节点，则已经找到了 $G_2$ 在 $G_1$ 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 P(s)；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，两个图是否保证同构，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匹配 p 从 s 中删除，回溯到上一个状态。

在上述算法流程中，对于每一个新加入的匹配 p，我们要检验其加入的可行性，从而对搜索空间进行剪枝，提高算法的效率。

先约定几个符号：$N_1$ 和 $N_2$ 表示图 $G_1$ 和图 $G_2$ 中的点集。n 和 m 分别表示图 $G_1$ 和图 $G_2$ 中的点。Pred(G，n)表示点 n 在图 G 中的前驱，Succ(G，n)表示点 n 在图 G 中的后继。$T_1^{in}(s)$和$T_2^{in}(s)$表示状态 s 在图 $G_1$ 和图 $G_2$ 中指向当前已经匹配的点集的所有边的源点集合。$T_1^{in}(s)$和$T_2^{in}(s)$表示状态 s 在

图 $G_1$ 和图 $G_2$ 中，从当前已经匹配的点集出发的所有边的终点集合。$T_1(s) = T_1^{in}(s) \cup T_1^{out}(s)$，表示当前状态 s 在图$G_1$中已经匹配的点集的所有一步邻居。$\tilde{N}_1(s) = N_1 - M_1(s) - T_1(s)$，表示图$G_1$中，除了 s 中已经匹配的点和这些点的一步邻居以外的点。

新加入的点对 p 需要同时满足五个规则：

$$\begin{cases} \mathrm{Pred}_{2to1} \Leftrightarrow \forall n' \in \mathrm{M}_1(s) \bigcap \mathrm{Pred}(\mathrm{G}_1, n), \ \exists m' \in \mathrm{Pred}(\mathrm{G}_2, m) \,|\, (n', m') \in \mathrm{M}(s) \\ \mathrm{Pred}_{1to2} \Leftrightarrow \forall m' \in \mathrm{M}_2(s) \bigcap \mathrm{Pred}(\mathrm{G}_2, m), \ \exists n' \in \mathrm{Pred}(\mathrm{G}_1, n) \,|\, (n', m') \in \mathrm{M}(s) \\ R_{pred}(s, n, m) \Leftrightarrow \mathrm{Pred}_{2to1} \wedge \mathrm{Pred}_{1to2} \end{cases} \tag{4.7}$$

$$\begin{cases} \mathrm{Succ}_{2to1} \Leftrightarrow \forall n' \in \mathrm{M}_1(s) \bigcap \mathrm{Succ}(\mathrm{G}_1, n), \ \exists m' \in \mathrm{Succ}(\mathrm{G}_2, m) \,|\, (n', m') \in \mathrm{M}(s) \\ \mathrm{Succ}_{1to2} \Leftrightarrow \forall m' \in \mathrm{M}_2(s) \bigcap \mathrm{Succ}(\mathrm{G}_2, m), \ \exists n' \in \mathrm{Succ}(\mathrm{G}_1, n) \,|\, (n', m') \in \mathrm{M}(s) \\ R_{succ}(s, n, m) \Leftrightarrow \mathrm{Succ}_{2to1} \wedge \mathrm{Succ}_{1to2} \end{cases} \tag{4.8}$$

$$\begin{cases} \mathrm{Card\_in}_{succ} \Leftrightarrow \mathrm{Card}(\mathrm{Succ}(\mathrm{G}_1, n) \bigcap T_1^{in}(s)) \geq \mathrm{Card}(\mathrm{Succ}(\mathrm{G}_2, m) \bigcap T_2^{in}(s)) \\ \mathrm{Card\_in}_{pred} \Leftrightarrow \mathrm{Card}(\mathrm{Pred}(\mathrm{G}_1, n) \bigcap T_1^{in}(s)) \geq \mathrm{Card}(\mathrm{Pred}(\mathrm{G}_2, m) \bigcap T_2^{in}(s)) \\ R_{in}(s, n, m) \Leftrightarrow \mathrm{Card\_in}_{succ} \wedge \mathrm{Card\_in}_{pred} \end{cases} \tag{4.9}$$

$$\begin{cases} \mathrm{Card\_out}_{succ} \Leftrightarrow \mathrm{Card}(\mathrm{Succ}(\mathrm{G}_1, n) \bigcap T_1^{out}(s)) \geq \mathrm{Card}(\mathrm{Succ}(\mathrm{G}_2, m) \bigcap T_2^{out}(s)) \\ \mathrm{Card\_out}_{pred} \Leftrightarrow \mathrm{Card}(\mathrm{Pred}(\mathrm{G}_1, n) \bigcap T_1^{out}(s)) \geq \mathrm{Card}(\mathrm{Pred}(\mathrm{G}_2, m) \bigcap T_2^{out}(s)) \\ R_{out}(s, n, m) \Leftrightarrow \mathrm{Card\_in}_{succ} \wedge \mathrm{Card\_out}_{pred} \end{cases} \tag{4.10}$$

$$\begin{cases} \mathrm{Card}_{pred} \Leftrightarrow \mathrm{Card}(\tilde{N}_1(s) \bigcap \mathrm{Pred}(\mathrm{G}_1, n) \geq \mathrm{Card}(\tilde{N}_2(s) \bigcap \mathrm{Pred}(\mathrm{G}_2, n) \\ \mathrm{Card}_{succ} \Leftrightarrow \mathrm{Card}(\tilde{N}_1(s) \bigcap \mathrm{Succ}(\mathrm{G}_1, n) \geq \mathrm{Card}(\tilde{N}_2(s) \bigcap \mathrm{Succ}(\mathrm{G}_2, n) \\ R_{new}(s, n, m) \Leftrightarrow \mathrm{Card}_{pred} \wedge \mathrm{Card}_{succ} \end{cases} \tag{4.11}$$

(4.12)和(4.13)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 $G_1$ 中的所有前驱（或后继），必须能在图 $G_2$ 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 $G_2$ 中的所有前驱（或后继），也必须能在图 $G_1$ 中 n 的前驱（或后继）里有相应的点与之对应。

(4.14)、(4.15)和(4.16)都是剪枝策略。其中Card()表示求集合中元素的个数。(4.17)和(4.18)表示 n 在$T_1^{in}(s)$（或$T_1^{out}(s)$）中的前驱（或后继）的数目，必须大于等于 m 在$T_2^{in}(s)$（或$T_2^{out}(s)$）中的前驱（或后继）的数目。如果不满足，则说明对于图 $G_2$ 中新匹配的点 m，其邻居个数是大于图 $G_1$ 中 n 的邻居个数的，最终必然无法完全匹配图 $G_2$ 中所有的点。

(4.19)跟(4.20)和(4.21)思想类似，只不过考虑的是两步邻居。具体来说，(4.22)和(4.23)中的考虑的邻居是$T_1^{in}(s)$和$T_1^{out}(s)$中的邻居。这些点既跟 n 相邻，又跟当前匹配中的其它点相邻。而(4.24)考虑的邻居是只跟 n 相邻，而跟当前匹配中其它点不相邻的邻居。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。

算法的搜索空间主要由候选点对集合 P(s)决定，在每一个状态 s 下都对应不同的搜索空间。候选点对的生成规则为：

1) 如果$T_1^{out}(s)$和$T_2^{out}$都不为空，则取这两个集合中的所有点两两组合，生成候选匹配点对集合；

2) 如果$T_1^{out}$(s)和$T_2^{out}$两个集合都为空，若$T_1^{in}$(s)和$T_2^{in}$(s)都不为空，则取这两个集合中的所有点两两组合，生成候选匹配点对集合；

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，生成候选匹配点对集合。

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点对集合，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。

### 5.2.3 架构图建模

在第三章中介绍了用有向图对算法建模的方法，利用图的节点描述算法中的算子，图中的边描述数据依赖。对于可重构架构，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图到架构图的匹配映射[35]。

**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。

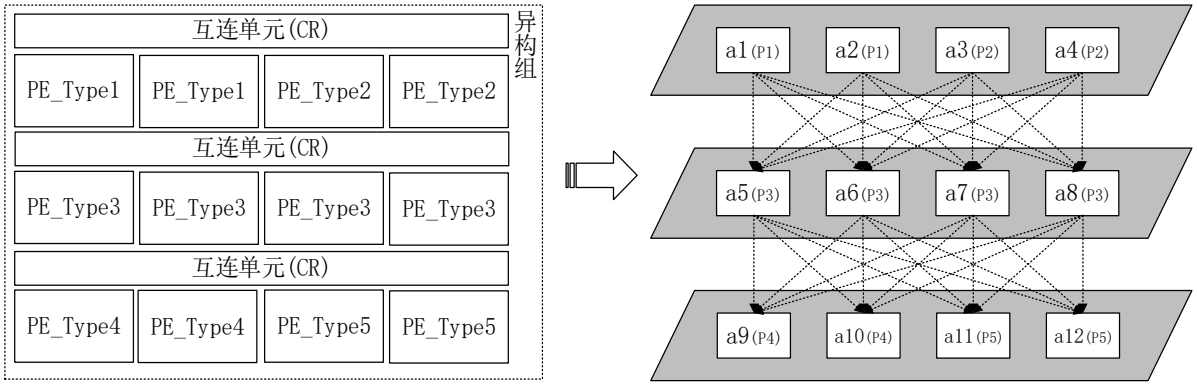
表 5-2 架构图结点属性

| 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_ |

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 表示算术功能，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，也就是说架构图节点对应的 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。

如图 5-5 异构组的图模型所示，在异构组的图模型中，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。




(a)异构组　　(b)图模型

图 5-5 异构组的图模型

## 5.2.4 基于 VF2 子图同构算法的映射方案

### 5.2.4.1 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(5.1)和(5.2)可知同构是一个双射关系。在算法图和架构图匹配的目标是从架构图中找出一个子图，这个子图包含算法图，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 5-6 描述了架构图和算法图的包含关系，{(v1+v2，a1)，(v3，a5)，(v4，a6)，(v5，a7)，(v6+v7+v8，a9)，(v9, a10)}是一个从算法图到架构图的映射，但是{a1，a5，a6，a7，a9，a10}构成的子图和算法图并不满足双射的同构关系，子图中的多条边在算法图中没有对应的边存在，这只是一个从算法图到架构子图的单射。

VF2 算法的 $R_{pred}$ 和 $R_{succ}$ 规则对图同构的双射关系进行了约束，因此对于算法映射必须放宽这两个规则，新的规则如下：

$$R_{succ\_m}(\mathrm{s},\mathrm{n},\mathrm{m}) \Leftrightarrow \forall m' \in \mathrm{M}_2(\mathrm{s}) \bigcap \mathrm{Succ}(\mathrm{G}_2,\mathrm{m}),\ \exists n' \in \mathrm{Succ}(\mathrm{G}_1,\ n) \mid (n',\ m') \in \mathrm{M}(\mathrm{s}) \quad (4.25)$$

$$R_{succ\_m}(\mathrm{s},\mathrm{n},\mathrm{m}) \Leftrightarrow \forall m' \in \mathrm{M}_2(\mathrm{s}) \bigcap \mathrm{Succ}(\mathrm{G}_2,\mathrm{m}),\ \exists n' \in \mathrm{Succ}(\mathrm{G}_1,\ n) \mid (n',\ m') \in \mathrm{M}(\mathrm{s}) \quad (4.26)$$

在新的规则中，只要求图 $G_1$（架构图）中包含图 $G_2$（算法图）对应的前驱和后继，反向则不再约束。

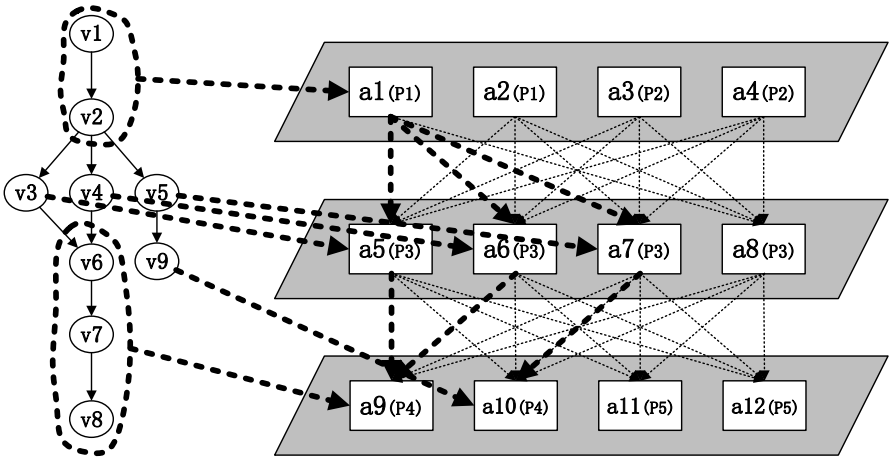



图 5-6 架构图和算法图的包含关系

单纯的 VF2 算法不考虑点的属性问题，规则 $R_{pred\_m}$ 和 $R_{succ\_m}$ 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是否提供算法图对应的点的操作属性。在进行算法图和架构图建模时，用字符串描述节点的功能属性，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 $R_{attr}$ 如式(4.27)所示，其中 attr(n)表示节点 n 的功能属性。

$$R_{attr}(\mathrm{s},\mathrm{n},\mathrm{m}) \Leftrightarrow \forall(\mathrm{n},\mathrm{m}) \in \mathrm{M(s)}, \mathrm{attr(m)} \subseteq \mathrm{attr(n)} \tag{4.28}$$

#### 5.2.4.2 最优匹配成本约束规则

架构图中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，5 个成本约束保证了最终入选的映射结果使用最少的行、列资源，最简单的数据互连，左上聚集的资源使用方式。

算法图中的节点包含了拓扑信息 a，v.a 表示算法图中的点 v 出现在横向上的位置；架构图中的节点也包含了拓扑信息（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，$s_i$ 为 S 中的一个元素，表示一个有效的匹配；$(m_j, n_j)$表示一个有效匹配 $s_i$ 中的一个匹配点对。五个成本约束如式(4.29)~(4.30)所示。

$$\begin{cases} R\max_{s_i} = \max(\mathrm{n}_j.\mathrm{x}) \\ R_{\min\_row}(\mathrm{S}) = \min(\mathrm{Rmax}_{s_i}) \end{cases} \tag{4.31}$$

$$\begin{cases} R\max_{s_i} = \max(\mathrm{n}_j.\mathrm{y}) \\ R_{\min\_col}(\mathrm{S}) = \min(\mathrm{Rmax}_{s_i}) \end{cases} \tag{4.32}$$

$$\begin{cases} L_{s_i} = \sum_{1}^{|s_i|} \mathrm{n}_j.\mathrm{y} \\ \mathrm{L(S)} = \min(L_{s_i}) \end{cases} \tag{4.33}$$

$$\begin{cases} U_{s_i} = \sum_{1}^{|s_i|} \mathrm{n}_j \,.\, \mathrm{x} \\ U_{\min}(\mathrm{S}) = \min(U_{s_i}) \end{cases} \tag{4.34}$$

$$\begin{cases} M_{s_i} = \sum_{1}^{|s_i|} |\mathrm{n}_j \,.\, \mathrm{y}\text{-}\mathrm{m}_j \,.\, \mathrm{a}| \\ M_{\min}(\mathrm{S}) = \min(M_{s_i}) \end{cases} \tag{4.35}$$

式(4.36)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用最少的资源行。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，最小的资源单位是行，因此映射的行数最少表示使用的行资源最少。

式(4.37)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(4.38)和(4.39)这两个约束选择出同时满足行列最小化的映射，这已经是最优映射；而式(4.40)、(4.41)和(4.42)确定的是映射工具映射时使用资源的聚集倾向。式(4.43)和(4.44)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聚集。式(4.45)约束功能乱序，保证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(4.46)、(4.47)和(4.48)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，保证这些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分布倾向性，这对于进一步的架构优化十分重要。

## 5.3 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[43]、CAMELLIA[44]、CAST128[45]、GOST[46]、RC5[47]、SEED[48]、TWOFISH[49]、SM4[50]、RC6[51]、SERPENT[52]、TEA[53]、XTEA[54]、SKIPJECT[55]、SPECK[56]、SIMON[56]、LUCIFER[57]、CLEFIA[58]、ARIA[59]、C2[60]、PRESENT[61]、MACGUFFIN[62]、SQUARE[63]、M6[64]、ICE[65]、SHARK[66]、CS_CIPHER[67]、NUSH[68]、GRAND_CRU[69]、Q[70]、E2[71]、KHAZAD[72]、HIEROCRYPT_L1[73]、HIEROCRYPT_3[73]。配置这些算法的图模型，然后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 个算法能够在本文的初始架构方案中完成映射，另外 6 个算法存在一些本架构不支持的运算操作，如乘法操作、条件判断等，因而不能被架构所支持，无法完成映射。

对异构组中各个 PE 在完成算法映射后功能单元的使用情况进行了总结，如表 5-3 所示，可以发现：

1) 由于映射方式的左上聚集特征，架构中 2、3 列 PE 功能单元使用频率比 1、2 列的要低很多，存在很多的闲置单元，而 1、2 列中的闲置单元相对要少很多；
2) 除了利用率为 0 的闲置单元以外，还存在很多使用频率为 1 的单元，即只有一个算法使用了该功能，这类单元属于低利用率单元；

3) 算术、移位和逻辑这三类功能单元中存在较多冗余闲置，而在初始架构设计时经过位置优化的置换、S 盒替代和有限域乘法这三类功能单元中只有一个功能单元是闲置的，普遍具有比较高的利用率。

表 5-3 异构组 PE 内部功能单元使用频度统计

| 内部功能 | PE 1_1 | PE 1_2 | PE 1_3 | PE 1_4 | PE 2_1 | PE 2_2 | PE 2_3 | PE 2_4 | PE 3_1 | PE 3_2 | PE 2_3 | PE 2_4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 算术单元 | 10 | 2 | 1 | 1 | 11 | 3 | 0 | 0 | 9 | 0 | 0 | 0 |
| 移位单元 | 11 | 4 | 1 | 0 | 2 | 3 | 0 | 0 | 7 | 4 | 1 | 0 |
| 逻辑单元 | 9 | 6 | 1 | 1 | 5 | 1 | 0 | 0 | 10 | 1 | 0 | 0 |
| 置换单元 | 2 | 2 | / | / | / | / | / | / | 4 | 0 | / | / |
| S 盒替代单元 | / | / | / | / | 28 | 21 | 9 | 9 | / | / | / | / |
| 有限域乘法单元 | / | / | / | / | / | / | / | / | 12 | 12 | 6 | 6 |

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；而对于功能利用率低的单元，算法映射时只有一两个算法使用了该功能，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，那么表示该功能单元可以删除，否则即使功能单元利用率低也要保留该单元。异构组 12 个 PE 中的闲置单元和低利用率单元如表 5-4 所示。

表 5-4 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_1 | 无 | 无 |
| PE1_2 | 无 | 无 |
| PE1_3 | 无 | 算术单元、移位单元、逻辑单元 |
| PE1_4 | 移位单元 | 算术单元、逻辑单元 |
| PE2_1 | 无 | 无 |
| PE2_2 | 无 | 逻辑单元 |
| PE2_3 | 算术单元、移位单元、逻辑单元 | 无 |
| PE2_4 | 算术单元、移位单元、逻辑单元 | 无 |
| PE3_1 | 无 | 无 |
| PE3_2 | 算术单元、置换单元 | 移位单元 |
| PE3_3 | 算术单元、逻辑单元 | 移位单元 |
| PE3_4 | 算术单元、移位单元、逻辑单元 | 无 |

PE1_3 和 PE1_4 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，因此放弃对该算术单元的优化；PE1_3 中的移位单元删除后，对应的 TEA 算法从另外的路径映射成功且没有增加映射规模，因此将该移位单元优化掉，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有一定的增加，因此确定方案时选择不对这些单元进行优化。

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 5-5 所示，PE 间的异构不只存在于置换单元、S 盒和有限域乘法单元，PE 间在算术单元、移位单元和逻辑单元上也存在了异构，PE 的种类由初始架构中的 5 类扩展成了 8 类。

表 5-5 最终方案中的异构组 PE 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_1 | √ | √ | √ | √ | × | × |
| PE1_2 | √ | √ | √ | √ | × | × |
| PE1_3 | √ | × | × | × | × | × |
| PE1_4 | √ | × | × | × | × | × |
| PE2_1 | √ | √ | √ | × | √ | × |
| PE2_2 | √ | √ | √ | × | √ | × |
| PE2_3 | × | × | × | × | √ | × |
| PE2_4 | × | × | × | × | √ | × |
| PE3_1 | √ | √ | √ | √ | × | √ |
| PE3_2 | × | √ | √ | × | × | √ |
| PE3_3 | × | √ | × | × | × | √ |
| PE3_4 | × | × | × | × | × | √ |

## 5.4 本章小结

本章通过分析可重构映射的问题模型，提出了基于 VF2 子图同构的算法映射方案，对 36 种分组密码算法进行了映射分析，总结了这些算法在映射后的功能单元使用分布，并根据功能分布对异构组中的冗余单元和低利用率单元进行了优化。

最终的优化方案相对初始架构减少了 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，异构组的功能单元数量减少到了 31，与之对应的同构架构包含 72 个功能单元，相比减少了 57%。

# 第六章 PE 方案验证与分析

## 6.1 PE 方案电路实现结果

对第五章给出的 PE 方案进行电路实现，使用 Verilog HDL 语言描述了该 PE 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 217797μm$^2$，各个功能单元的面积和延迟如表 6-1 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 |

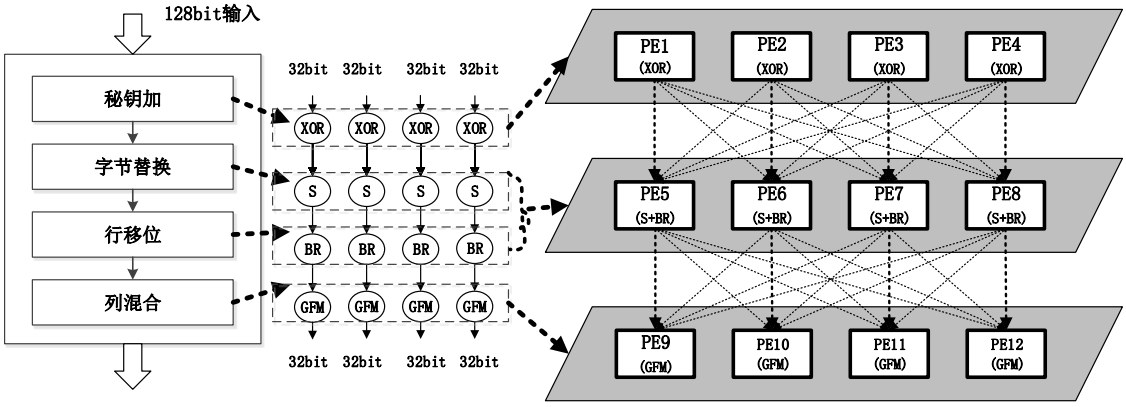
## 6.2 算法映射

本文完成了 30 个比较常用的分组密码算法的映射分析，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，完整算法的映射是轮函数映射的重复。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 SP 结构、Feistel 结构和 ARX 结构的代表算法 AES 算法、DES 算法和 SPECK 算法进行详细映射分析，其它算法的映射结果见附录 A。

### 6.2.1 AES 算法映射结果

AES 算法的轮函数共有 4 个连续操作：密钥加、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；PE 行间互连如(c)中加粗虚线所示，都为直连方式。整个算法轮函数在三行 PE 中完成映射，对应于一个 PE 异构组。




(a)算法流程　　(b)算法有向图模型　　(c)算法图在架构超图中映射

图 6-1 AES 算法映射

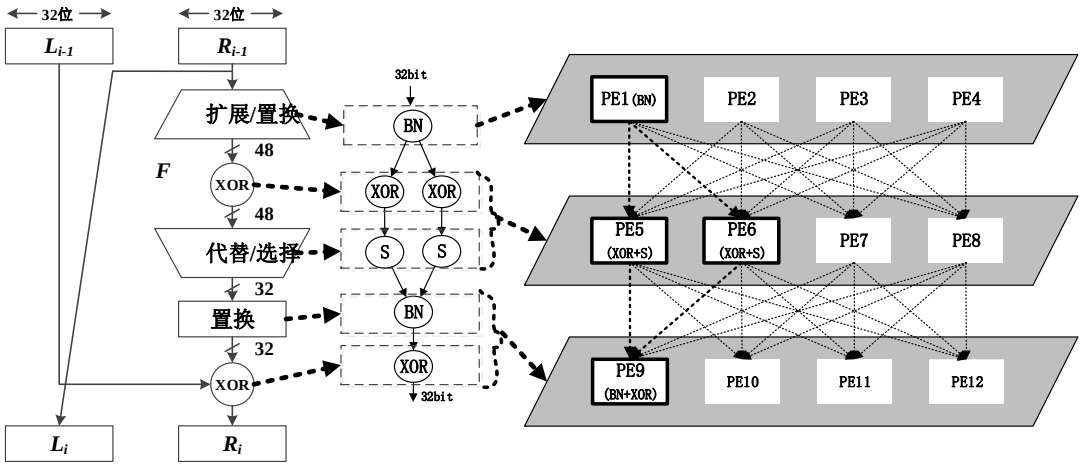
AES 算法轮函数中共有 4 个逻辑运算，4 个查表运算和 4 个有限域乘法运算；PE 方案在优化后，一个异构组中含有 7 个算术单元，7 个移位单元，3 个置换单元，6 个逻辑单元，4 个 S 盒和 4 个有限域乘法单元；对应的各个单元的利用率如表 6-2 的第 4 行所示，算法中没有算术运算、移位运算和置换运算，因此对应的功能单元利用率为 0，逻辑单元、S 盒和有限域乘法单元的利用率分别为 67%、100%和 100%；最后一列的综合利用率是指所有单元的统计结果，对于 AES，算法中有 12 个操作，映射架构有 31 个功能单元，利用率为 39%。

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% |

### 6.2.2 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、密钥加、代替、置换和左右异或，分别对应于置换、异或逻辑、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的组合。(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，PE 行间互连如(c)中加粗虚线所示。整个算法轮函数在三行 PE 中完成映射，对应于一个 PE 异构组。




(a)算法流程　　(b)算法有向图模型　　(c)算法图在架构超图中映射

图 6-2 DES 算法映射

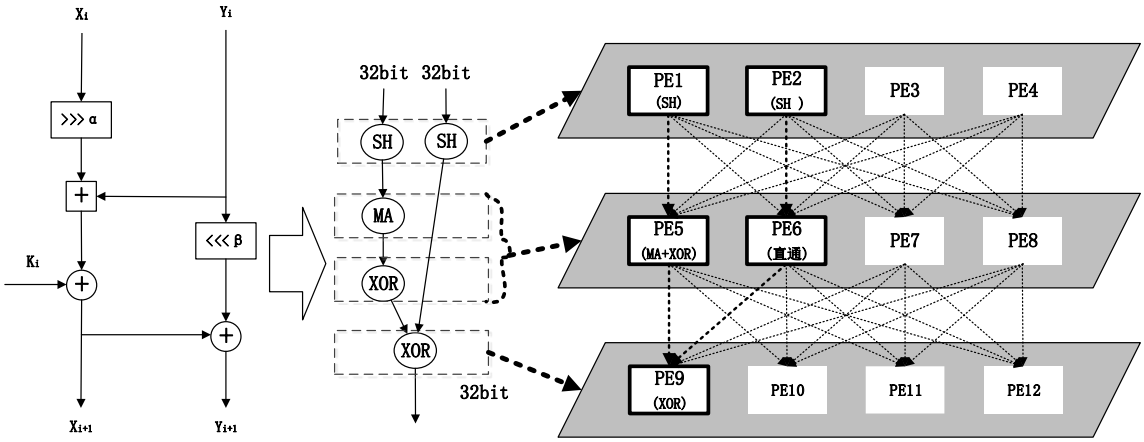
DES 算法轮函数中共有 3 个逻辑运算，2 个查表运算和 2 个置换运算；PE 方案在优化后，一个异构组中含有 7 个算术单元，7 个移位单元，3 个置换单元，6 个逻辑单元，4 个 S 盒和 4 个有限域乘法单元；算法中没有算术运算、移位运算和有限域乘法运算，因此对应的功能单元利用率为 0，置换单元、逻辑单元和 S 盒的利用率分别为 67%、50%和 50%； DES 算法中有 7 个操作，映射架构有 31 个功能单元，综合利用率为 23%。

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% |

### 6.2.3 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，1 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。算法中算术运算和异或运算可以组合在一行中完成。(c)中第一行的 2 个 PE 完成移位操作，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，PE 行间互连如(c)中加粗虚线所示。整个算法轮函数在三行 PE 中完成映射，对应于一个 PE 异构组。




(a) 算法流程　　(b) 算法有向图模型　　(c) 算法图在架构超图中映射

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 盒和 4 个有限域乘法单元。算法中没有置换运算、S 盒运算和有限域乘法运算，因此对应的功能单元利用率为 0，算术单元、移位单元和逻辑单元的利用率分布为 14%、29%和 33%； SPECK 算法中有 5 个操作，映射架构有 31 个功能单元，综合利用率为 16%。

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% |

## 6.3 与其它可重构方案对比分析

### 6.3.1 功能单元利用率

功能单元利用率是指算法包含运算数与映射架构单元数的比值，算法包含的操作是固定的，因此如果功能单元利用率提高就意味着架构用更少的功能单元完成了同样的算法运算，是架构设计优劣的直接表现。

附录 A 的表 A-1 列出了 30 个比较常用的分组密码算法在各个架构方案上的映射结果，包括算法的功能单元数，算法映射后的映射单元数，不同功能单元的利用率和所有功能单元的综合利用率。表中共有五种不同架构的映射结果，其中 Cyptor[17]、RCPA[18]和 COBRA[15]架构来自近几年的期刊论文，RPU 架构是实验室与研究机构在 2015 年合作研发的一款针对分组密码算法的可重构密码芯片，表中的部分映射结果来自于论文原始数据，由于论文中的映射算法有限，很多算法的映射结果无法从论文中直接获取，本文对这些算法在对应的架构上进行了补全映射分析，使得所有的架构有相同的算法映射集合。在五种架构中，除了 COBRA 架构由于不支持置换操作只能映射其中 24 个算法以外，其它四个架构均成功映射算法集中的 30 个分组密码算法。

由于算法数量大，对表 A-1 的所有算法映射结果进行了平均，分别计算出不同功能单元的平均使用数量、平均利用率，并且与本文的架构进行了对比，计算出不同功能单元减少的比例，如表 6-5 所示。

表 6-5 不同架构方案的功能单元平均使用数量对比

| 架构 | 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 数量 | 减少比例 | 数量 | 减少比例 | 数量 | 减少比例 | 数量 | 减少比例 | 数量 | 减少比例 | 数量 | 减少比例 | 数量 | 减少比例 |
| **Cyptor [17]** | 15.5 | 30% | 15.5 | 30% | 15.5 | 70% | 15.5 | 40% | 15.5 | 60% | 0.0 | / | 77.3 | 43% |
| **RCPA [19]** | 17.5 | 38% | 17.5 | 38% | 17.5 | 73% | 34.9 | 73% | 17.5 | 64% | 17.5 | 64% | 122.3 | 64% |
| **COBRA [15]** | 19.7 | 45% | 29.5 | 63% | 9.8 | 53% | 29.5 | 68% | 4.9 | -27% | 9.8 | 37% | 103.3 | 58% |
| **RPU** | 18.9 | 42% | 37.9 | 71% | 9.5 | 51% | 18.9 | 51% | 18.9 | 67% | 0.0 | / | 104.1 | 58% |
| **本文** | 10.9 | / | 10.9 | / | 4.7 | / | 9.3 | / | 6.2 | / | 6.2 | / | 43.7 | / |

从表 6-5 可以看到，本文的方案算法映射所使用的各种功能单元平均数量比其它 4 个架构方案要少得多，减少比例在 30%到 73%之间，唯一的例外是 COBRA 架构的 S 盒单元，COBRA 中含有很少的 S 盒单元，它采用复杂的串联结构规避这个弊端，这导致了其它功能单元开销变大，从表中可以看出，COBRA 的逻辑单元和算术单元都比其它架构要多；表 6-5 最后两列列出了架构功能单元的综合数量，本文方案的功能单元综合数目是 43.7，相对其它架构减少了 43%到 64%，大幅减少了架构上的功能单元数量。

表 6-6 不同架构方案的功能单元平均利用率对比

| 架构 | 算术单元 | | 移位单元 | | 置换单元 | | 逻辑单元 | |
|---|---|---|---|---|---|---|---|---|
| | 平均利用率 | 提升比例 | 平均利用率 | 提升比例 | 平均利用率 | 提升比例 | 平均利用率 | 提升比例 |
| **Cyptor[17]** | 7.2% | 63% | 6.2% | 67% | 2.0% | 266% | 27.8% | 63% |
| **RCPA[19]** | 7.0% | 67% | 5.3% | 96% | 2.0% | 266% | 13.1% | 246% |
| **COBRA[15]** | 7.0% | 67% | 4.1% | 149% | 0.3% | 1980% | 16.1% | 182% |
| **RPU** | 6.3% | 87% | 3.1% | 236% | 3.2% | 126% | 22.7% | 100% |
| **本文** | 11.7% | / | 10.3% | / | 7.2% | / | 45.4% | / |
| **架构** | **S 盒** | | **有限域乘法** | | **综合** | | | |
| | 平均利用率 | 提升比例 | 平均利用率 | 提升比例 | 平均利用率 | 提升比例 | | |
| **Cyptor[17]** | 16.7% | 140% | / | / | 13.9% | 83.2% | | |
| **RCPA[19]** | 16.3% | 147% | 9.7% | 131% | 9.5% | 168.5% | | |
| **COBRA[15]** | 51.9% | -23% | 16.3% | 38% | 11.2% | 128.4% | | |
| **RPU** | 12.7% | 216% | / | / | 10.2% | 150.8% | | |
| **本文** | 40.2% | / | 22.5% | / | 25.5% | / | | |

表 6-56 列出了架构中不同功能单元的平均利用率，和其它 4 个架构对比，本文的方案在各种功能单元平均利用率上的提升比例在 38%到 1980%之间，综合功能单元利用率提升在 83.2%到 168.5%之间，大幅提高了功能单元的利用率。

### 6.3.2 面积效率

对于密码算法实现，系统的面积效率越高意味着在有限的面积下算法的性能越高。本文对比了5种不同的密码可重构架构在实现30种不同分组密码算法时的面积、性能和性能面积比。如**错误!未找到引用源。**所示，表中列出了各种架构平台在实现不同算法的性能、面积和性能面积比。由于本文关注的是系统的计算阵列部分的优化设计，下表中的面积仅仅是完成算法所需的计算阵列的面积，忽略系统的其它部分。由于不同架构的工艺水平存在差异性，对于Cyptor[17]、RCPA[18]和COBRA[15]架构，本文查询了相关的架构文献，对这些架构的计算阵列进行了对应的功能复现，复现的架构与原始架构方案在功能单元层次上保证一致性，本文在功能单元层次对架构中的功能冗余进行优化，与功能单元的具体设计独立的，因此对于功能单元以下的电路级设计存在的差异性可以忽略。

表6-7 不同架构方案在运行分组密码算法的性能、面积和面积效率对比

| 架构 | 算法 | AES | DES | SM4 | TWOFISH | RC5 | CAST128 | SERPENT | BLOWFISH |
|---|---|---|---|---|---|---|---|---|---|
| Cyptor | 性能（Gbps） | 59.5 | 29.8 | 59.5 | 59.5 | 59.5 | 29.8 | 59.5 | 29.8 |
| | 面积($mm^2$) | 0.25 | 0.37 | 0.49 | 0.62 | 0.49 | 0.62 | 0.86 | 0.37 |
| | 性能面积比(Gbps/$mm^2$) | 241.7 | 80.6 | 120.9 | 96.7 | 120.9 | 48.3 | 69.1 | 80.6 |
| RCPA | 性能（Gbps） | 32.9 | 16.5 | 32.9 | 32.9 | 32.9 | 16.5 | 32.9 | 16.5 |
| | 面积($mm^2$) | 0.32 | 0.48 | 1.91 | 0.95 | 0.64 | 0.79 | 1.43 | 0.48 |
| | 性能面积比(Gbps/$mm^2$) | 103.6 | 34.5 | 17.3 | 34.5 | 51.8 | 20.7 | 23.0 | 34.5 |
| COBRA | 性能（Gbps） | 9.9 | / | 9.9 | 9.9 | 9.9 | 5.0 | 9.9 | 5.0 |
| | 面积($mm^2$) | 0.31 | / | 0.47 | 0.47 | 0.31 | 0.31 | 0.62 | 0.16 |
| | 性能面积比(Gbps/$mm^2$) | 31.8 | / | 21.2 | 21.2 | 31.8 | 15.9 | 15.9 | 31.8 |
| RPU | 性能（Gbps） | 64.0 | 32.0 | 64.0 | 64.0 | 64.0 | 32.0 | 64.0 | 32.0 |
| | 面积($mm^2$) | 0.23 | 0.47 | 0.47 | 0.70 | 0.47 | 0.58 | 0.93 | 0.47 |
| | 性能面积比(Gbps/$mm^2$) | 274.6 | 68.7 | 137.3 | 91.5 | 137.3 | 54.9 | 68.7 | 68.7 |
| 本文 | 性能（Gbps） | 64.0 | 32.0 | 64.0 | 64.0 | 64.0 | 32.0 | 64.0 | 32.0 |
| | 面积($mm^2$) | 0.22 | 0.22 | 0.22 | 0.44 | 0.29 | 0.44 | 0.48 | 0.22 |
| | 性能面积比(Gbps/$mm^2$) | 293.9 | 146.9 | 293.9 | 146.9 | 220.4 | 73.5 | 132.3 | 146.9 |
| 架构平台 | 算法 | SEED | CAMELLIA | GOST | TEA | XTEA | SPECK | SIMON | LUCIFER |
| Cyptor | 性能（Gbps） | 59.5 | 59.5 | 29.8 | 29.8 | 29.8 | 29.8 | 29.8 | 59.5 |
| | 面积($mm^2$) | 1.23 | 0.49 | 0.37 | 0.49 | 0.37 | 0.37 | 0.37 | 0.37 |
| | 性能面积比(Gbps/$mm^2$) | 48.3 | 120.9 | 80.6 | 60.4 | 80.6 | 80.6 | 80.6 | 161.1 |
| RCPA | 性能（Gbps） | 32.9 | 32.9 | 16.5 | 16.5 | 16.5 | 16.5 | 16.5 | 32.9 |
| | 面积($mm^2$) | 1.75 | 0.95 | 0.48 | 0.64 | 0.48 | 0.48 | 0.64 | 0.48 |
| | 性能面积比(Gbps/$mm^2$) | 18.8 | 34.5 | 34.5 | 25.9 | 34.5 | 34.5 | 25.9 | 69.1 |
| COBRA | 性能（Gbps） | 9.9 | 9.9 | 5.0 | 5.0 | 5.0 | 5.0 | 5.0 | / |
| | 面积($mm^2$) | 1.09 | 0.47 | 0.16 | 0.62 | 0.34 | 0.31 | 0.47 | / |
| | 性能面积比(Gbps/$mm^2$) | 9.1 | 21.2 | 31.8 | 8.0 | 14.6 | 15.9 | 10.6 | / |
| RPU | 性能（Gbps） | 64.0 | 64.0 | 32.0 | 32.0 | 32.0 | 32.0 | 32.0 | 64.0 |
| | 面积($mm^2$) | 1.40 | 0.70 | 0.47 | 0.47 | 0.44 | 0.35 | 0.23 | 0.47 |
| | 性能面积比(Gbps/$mm^2$) | 45.8 | 91.5 | 68.7 | 68.7 | 73.1 | 91.5 | 137.3 | 137.3 |
| 本文 | 性能（Gbps） | **64.0** | **64.0** | **32.0** | **32.0** | **32.0** | **32.0** | **32.0** | **64.0** |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **面积(mm²)** | **1.09** | **0.44** | **0.22** | **0.29** | **0.22** | **0.22** | **0.22** | **0.22** |
| | **性能面积比(Gbps/mm²)** | **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** |
| **Cyptor** | **性能（Gbps）** | 59.5 | 59.5 | 29.8 | 29.8 | 29.8 | 59.5 | 29.8 | 29.8 |
| | **面积(mm²)** | 0.37 | 0.25 | 0.62 | 0.25 | 0.25 | 0.37 | 1.11 | 0.25 |
| | **性能面积比(Gbps/mm²)** | 161.1 | 241.7 | 48.3 | 120.9 | 120.9 | 161.1 | 26.9 | 120.9 |
| **RCPA** | **性能（Gbps）** | 32.9 | 32.9 | 16.5 | 16.5 | 16.5 | 32.9 | 16.5 | 16.5 |
| | **面积(mm²)** | 0.48 | 0.32 | 0.79 | 0.32 | 0.32 | 0.48 | 1.43 | 0.32 |
| | **性能面积比(Gbps/mm²)** | 69.1 | 103.6 | 20.7 | 51.8 | 51.8 | 69.1 | 11.5 | 51.8 |
| **COBRA** | **性能（Gbps）** | 9.9 | 9.9 | 5.0 | / | 5.0 | / | 5.0 | 5.0 |
| | **面积(mm²)** | 0.31 | 0.16 | 0.47 | / | 0.31 | / | 0.62 | 0.16 |
| | **性能面积比(Gbps/mm²)** | 31.8 | 63.7 | 10.6 | / | 15.9 | / | 8.0 | 31.8 |
| **RPU** | **性能（Gbps）** | 64.0 | 64.0 | 32.0 | 32.0 | 32.0 | 64.0 | 32.0 | 32.0 |
| | **面积(mm²)** | 0.47 | 0.47 | 0.70 | 0.23 | 0.23 | 0.47 | 1.05 | 0.47 |
| | **性能面积比(Gbps/mm²)** | 137.3 | 137.3 | 45.8 | 137.3 | 137.3 | 137.3 | 30.5 | 68.7 |
| **本文** | **性能（Gbps）** | **64.0** | **64.0** | **32.0** | **32.0** | **32.0** | **64.0** | **32.0** | **32.0** |
| | **面积(mm²)** | **0.22** | **0.22** | **0.44** | **0.22** | **0.22** | **0.22** | **0.65** | **0.22** |
| | **性能面积比(Gbps/mm²)** | **293.9** | **293.9** | **73.5** | **146.9** | **146.9** | **293.9** | **49.0** | **146.9** |
| **架构平台** | **算法** | **NUSH** | **GRAND CRU** | **E2** | **KHAZAD** | **HIEROCRYPT-L1** | **HIEROCRYPT-3** | **平均** | **优化百分比** |
| **Cyptor** | **性能（Gbps）** | 59.5 | 59.5 | 59.5 | 29.8 | 29.8 | 59.5 | **44.7** | **7.5%** |
| | **面积(mm²)** | 0.37 | 0.37 | 0.74 | 0.25 | 0.49 | 0.49 | **0.48** | **-30.0%** |
| | **性能面积比(Gbps/mm²)** | 161.1 | 161.1 | 80.6 | 120.9 | 60.4 | 120.9 | **109.3** | **57.1%** |
| **RCPA** | **性能（Gbps）** | 32.9 | 32.9 | 32.9 | 16.5 | 16.5 | 32.9 | **24.7** | **94.5%** |
| | **面积(mm²)** | 0.48 | 0.48 | 0.95 | 0.32 | 0.64 | 0.64 | **0.69** | **-51.9%** |
| | **性能面积比(Gbps/mm²)** | 69.1 | 69.1 | 34.5 | 51.8 | 25.9 | 51.8 | **44.3** | **287.3%** |
| **COBRA** | **性能（Gbps）** | 9.9 | / | / | 5.0 | 5.0 | 9.9 | **7.2** | **562.4%** |
| | **面积(mm²)** | 0.31 | / | / | 0.16 | 0.31 | 0.31 | **0.38** | **-13.5%** |
| | **性能面积比(Gbps/mm²)** | 31.8 | / | / | 31.8 | 15.9 | 31.8 | **23.1** | **643.5%** |
| **RPU** | **性能（Gbps）** | 64.0 | 64.0 | 64.0 | 32.0 | 32.0 | 64.0 | **48.0** | **0.0%** |
| | **面积(mm²)** | 0.35 | 0.47 | 0.58 | 0.47 | 0.93 | 0.93 | **0.55** | **-39.9%** |
| | **性能面积比(Gbps/mm²)** | 183.1 | 137.3 | 110.5 | 68.7 | 34.3 | 68.7 | **101.7** | **68.9%** |
| **本文** | **性能（Gbps）** | **64.0** | **64.0** | **64.0** | **32.0** | **32.0** | **64.0** | **48.0** | |
| | **面积(mm²)** | **0.22** | **0.22** | **0.65** | **0.22** | **0.44** | **0.44** | **0.33** | |
| | **性能面积比(Gbps/mm²)** | **293.9** | **293.9** | **98.0** | **146.9** | **73.5** | **146.9** | **171.7** | |

通过不同架构方案间的对比分析，可以发现：

1) 在性能上，本文的方案与近年的 Cyptor 和 RPU 架构对比并没有明显的提升，和 Cyptor 对比提升了 7.5%，和 RPU 对比则完全一样；与本文类似，这两种架构也采用了功能并行设计方案，整个阵列的主频可以很高。与 RCPA 和 COBRA 架构对比，性能提升比较明显，分别为 94.5%和 562.4%，这两种架构采用功能串行设计，整个系统的主频较低，这是功能并行方案对功能串行方案在性能上的优势；

2) 在面积上，本文通过次序特征优化和映射反馈设计优化消除了架构中的很多冗余单元，因此本文的方案相对于其它的架构都有着比较明显的优势，与 Cyptor、RCPA、COBRA 和 RPU 对比分别减少了 30.0%、51.9%、13.5%和 39.9%；

3) 在性能面积比上，与 RCPA 和 COBRA 架构对比，由于本文在性能和面积上都有着很大的优势，因此面积性能比提升很大，提升比例分别为 287.3%和 643.5%；与 Cyptor 和 RPU 架构对比，虽然性能提升不大，但是本文在面积上进行了深入的优化，面积优势明显，因此整体的性能面积比也有着比较大的提升，提升比例分别为 57.1%和 68.9%。

综上所述，本文的架构方案在面积效率方面优于同类型密码可重构平台，实现结果与同类可重构架构比较，30 种算法的平均面积效率比已有文献提高了 57.1%~643.5%。

## 6.4 本章小结

本章对 PE 方案进行了电路实现，仿真验证，并综合获取性能、面积等方面的数据，对 30 种分组密码算法进行了映射分析，给出了这些算法的映射结果，包括功能单元使用数量、功能单元利用率以及性能、面积、性能面积比。将这些数据与同类型的密码可重构架构进行对比，结果表明：本文的 PE 方案，30 种算法的平均功能单元利用率比已有文献提高了 83.2%~168.5%，面积效率提高了 57.1%~643.5%，在面积效率方面优于同类型可重构平台。

# 第七章 总结与展望

## 7.1 总结

为了解决当前密码可重构阵列方案中功能单元冗余量大、利用率低，整体面积效率不高的问题，本文从算法算子次序特征和算法映射反馈设计两方便对密码可重构的冗余功能单元进行了优化，给出了一套无功能冗余的密码可重构 PE 阵列设计方案，提升了整个系统的面积效率。

本文完成的主要工作如下：

1) 算法特征分析：选取 36 种比较常用分组密码算法作为算法研究对象，为分组密码算法建立了统一的图分析模型，基于这个图模型提取了架构设计相关的算法算子模式特征，组合特征和次序特征，为本文后续的架构设计提供依据；
2) 确定初始架构：根据提取的算法特征，确定 PE 初始设计方案，算子模式特征指导功能单元运算模式设计，组合特征指导功能单元组合设计，次序特征指导功能单元在 PE 阵列中的分布，这个初始架构作为后续算法映射反馈优化设计的基础模板架构；
3) 架构建模：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对架构调整验证，完成架构后续的探索优化。
4) 基于算法映射的反馈优化：提出基于 VF2 子图同构的映射方案，完成目标算法集合中的算法图到架构超图的映射；根据算法集合的映射结果，分析功能使用分布情况，优化初始架构中的冗余功能单元。

实验结果证明本文的架构方案在面积效率方面优于同类型可重构平台，实现结果与同类可重构架构比较，30 种算法的平均功能单元利用率比已有文献提高了 83.2%~168.5%，面积效率提高了 57.1%~643.5%。

## 7.2 展望

本文提出了一套无功能冗余的密码可重构 PE 阵列设计方案，在一定程度上减少了密码可重构阵列的功能冗余，提高了整体的面积效率。后续还可以从以下几方面开展进一步的研究工作：

1) 本文针对的是面向分组密码算法的可重构阵列设计优化，可重构配置控制器作为可重构系统的另一个重要部分也存在很大的优化空间，比如配置切换时间优化、配置信息组织优化等，通过对可重构配置控制器的进一步优化，可以对整个可重构系统的面积进一步优化、运行频率进一步提升，从而提升整个平台的性能和面积效率；
2) 本文面向的是分组密码算法，而分组密码算法只是整个密码算法领域中的一部分内容，为了提升整个可重构系统平台的适用性，后续可以将更多类别的密码算法纳入研究范围，以使得整个可重构系统平台适用于其它密码算法。

# 致谢

时光荏苒，转眼间三年的研究生生活就要结束了，感觉自己在生活和学习上都收获了很多，值此论文即将完成之际，谨向所有在我的学习和生活上给予关心、指导和帮助的老师、同学、亲人致以衷心的感谢！

首先感谢曹鹏老师，在生活上和工作上都给与了我很多帮助。在开展项目过程中，与我共同解决遇到的问题，课题研究上，对我严格要求，很多处理问题的思路对我影响很深。在论文的研究和撰写过程中，给我提出了很多宝贵的意见和建议，并指出论文中的不足和研究中的缺陷，使我进一步完善论文，完成论文最后的定稿，在此对他再次致以深深的谢意。

其次要感谢师兄杨锦江博士和已经毕业的胡建兵师兄，感谢两位师兄在项目工作和课题研究中给我的指导和在北京一起出差时对我的照顾和关心。

此外还要特地感谢闵婧同学和申艾麟、李兆奇、尹玲三位师弟师妹在课题实验方面给予的大力帮助。同时感谢可重构项目组的所有兄弟姐妹们，让我时刻感受到家一般的温暖，我会永远铭记和他们一起度过的那些美好时光。

最后感谢我的父母、我的哥哥和我的女朋友，不管我在学习生活中遇到何种困难和挫折，他们都一直默默地支持和鼓励我，让我重新找回自信，使我能够顺利地完成学业，感谢她们对我的无私奉献。

# 附录 A 算法在不同架构平台的映射结果

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 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 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 | | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 50% | / | 30% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 50% | 50% | 21% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| | RPU | 所需功能单元 | 8 | 16 | 4 | 8 | 8 | 0 | 44 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 50% | / | 27% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 67% | 100% | 100% | 39% |
| DES | | 算法包含操作 | 0 | 0 | 2 | 3 | 2 | 0 | 7 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 0% | 0% | 17% | 25% | 17% | / | 12% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 0% | 0% | 17% | 13% | 17% | 0% | 8% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 25% | 19% | 13% | / | 8% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 67% | 50% | 50% | 0% | 23% |
| SM4 | | 算法包含操作 | 0 | 3 | 0 | 2 | 1 | 0 | 6 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 0% | 19% | 0% | 13% | 6% | / | 8% |
| | RCPA | 所需功能单元 | 48 | 48 | 48 | 96 | 48 | 48 | 336 |
| | | 功能单元利用率 | 0% | 6% | 0% | 2% | 2% | 0% | 2% |
| | COBRA | 所需功能单元 | 24 | 36 | 12 | 36 | 6 | 12 | 126 |
| | | 功能单元利用率 | 0% | 8% | 0% | 6% | 17% | 0% | 5% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 9% | 0% | 13% | 6% | / | 7% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 43% | 0% | 33% | 25% | 0% | 19% |
| TWOFI | | 算法包含操作 | 4 | 3 | 0 | 1 | 2 | 2 | 12 |
| | Cyptor | 所需功能单元 | 20 | 20 | 20 | 20 | 20 | 0 | 100 |
| | | 功能单元利用率 | 20% | 15% | 0% | 5% | 10% | / | 12% |
| | RCPA | 所需功能单元 | 24 | 24 | 24 | 48 | 24 | 24 | 168 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| SH | | 功能单元利用率 | 17% | 13% | 0% | 2% | 8% | 8% | 7% |
| | COBRA | 所需功能单元 | 24 | 36 | 12 | 36 | 6 | 12 | 126 |
| | | 功能单元利用率 | 17% | 8% | 0% | 3% | 33% | 17% | 10% |
| | RPU | 所需功能单元 | 24 | 48 | 12 | 24 | 24 | 0 | 132 |
| | | 功能单元利用率 | 17% | 6% | 0% | 4% | 8% | / | 9% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 31 |
| | | 功能单元利用率 | 29% | 21% | 0% | 8% | 25% | 25% | 39% |
| RC5 | | 算法包含操作 | 2 | 2 | 0 | 2 | 0 | 0 | 6 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 13% | 13% | 0% | 13% | 0% | / | 8% |
| | RCPA | 所需功能单元 | 16 | 16 | 16 | 32 | 16 | 16 | 112 |
| | | 功能单元利用率 | 13% | 13% | 0% | 6% | 0% | 0% | 5% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 13% | 8% | 0% | 8% | 0% | 0% | 7% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 13% | 6% | 0% | 13% | 0% | / | 7% |
| | 本文 | 所需功能单元 | 9 | 9 | 4 | 8 | 5 | 5 | 31 |
| | | 功能单元利用率 | 21% | 21% | 0% | 25% | 0% | 0% | 19% |
| CAST128 | | 算法包含操作 | 3 | 1 | 0 | 3 | 1 | 0 | 8 |
| | Cyptor | 所需功能单元 | 20 | 20 | 20 | 20 | 20 | 0 | 100 |
| | | 功能单元利用率 | 15% | 5% | 0% | 15% | 5% | / | 8% |
| | RCPA | 所需功能单元 | 20 | 20 | 20 | 40 | 20 | 20 | 140 |
| | | 功能单元利用率 | 15% | 5% | 0% | 8% | 5% | 0% | 6% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 19% | 4% | 0% | 13% | 25% | 0% | 10% |
| | RPU | 所需功能单元 | 20 | 40 | 10 | 20 | 20 | 0 | 110 |
| | | 功能单元利用率 | 15% | 3% | 0% | 15% | 5% | / | 7% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 31 |
| | | 功能单元利用率 | 21% | 7% | 0% | 25% | 13% | 0% | 26% |
| SERPENT | | 算法包含操作 | 0 | 8 | 0 | 12 | 4 | 0 | 24 |
| | Cyptor | 所需功能单元 | 28 | 12 | 12 | 12 | 12 | 0 | 76 |
| | | 功能单元利用率 | 0% | 67% | 0% | 100% | 33% | / | 32% |
| | RCPA | 所需功能单元 | 36 | 12 | 12 | 24 | 12 | 12 | 108 |
| | | 功能单元利用率 | 0% | 67% | 0% | 50% | 33% | 0% | 22% |
| | COBRA | 所需功能单元 | 32 | 24 | 8 | 24 | 4 | 8 | 100 |
| | | 功能单元利用率 | 0% | 33% | 0% | 50% | 100% | 0% | 24% |
| | RPU | 所需功能单元 | 32 | 24 | 6 | 12 | 12 | 0 | 86 |
| | | 功能单元利用率 | 0% | 33% | 0% | 100% | 33% | / | 28% |
| | 本文 | 所需功能单元 | 21 | 21 | 9 | 18 | 12 | 12 | 31 |
| | | 功能单元利用率 | 0% | 38% | 0% | 67% | 33% | 0% | 77% |
| BLOWFI | | 算法包含操作 | 2 | 0 | 0 | 6 | 4 | 0 | 12 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 17% | 0% | 0% | 50% | 33% | / | 20% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 17% | 0% | 0% | 25% | 33% | 0% | 14% |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| SH | COBRA | 所需功能单元 | 8 | 12 | 4 | 12 | 2 | 4 | 42 |
| | | 功能单元利用率 | 25% | 0% | 0% | 50% | 200% | 0% | 29% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 13% | 0% | 0% | 38% | 25% | / | 14% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 29% | 0% | 0% | 100% | 100% | 0% | 39% |
| SEED | | 算法包含操作 | 3 | 0 | 0 | 11 | 3 | 0 | 17 |
| | Cyptor | 所需功能单元 | 40 | 40 | 40 | 40 | 40 | 0 | 200 |
| | | 功能单元利用率 | 8% | 0% | 0% | 28% | 8% | / | 9% |
| | RCPA | 所需功能单元 | 44 | 44 | 44 | 88 | 44 | 44 | 308 |
| | | 功能单元利用率 | 7% | 0% | 0% | 13% | 7% | 0% | 6% |
| | COBRA | 所需功能单元 | 56 | 84 | 28 | 84 | 14 | 28 | 294 |
| | | 功能单元利用率 | 5% | 0% | 0% | 13% | 21% | 0% | 6% |
| | RPU | 所需功能单元 | 48 | 96 | 24 | 48 | 48 | 0 | 264 |
| | | 功能单元利用率 | 6% | 0% | 0% | 23% | 6% | / | 6% |
| | 本文 | 所需功能单元 | 35 | 35 | 15 | 30 | 20 | 20 | 155 |
| | | 功能单元利用率 | 9% | 0% | 0% | 37% | 15% | 0% | 11% |
| CAMELLIA | | 算法包含操作 | 0 | 0 | 0 | 8 | 2 | 0 | 10 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 13% | / | 13% |
| | RCPA | 所需功能单元 | 24 | 24 | 24 | 48 | 24 | 24 | 168 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 8% | 0% | 6% |
| | COBRA | 所需功能单元 | 24 | 36 | 12 | 36 | 6 | 12 | 126 |
| | | 功能单元利用率 | 0% | 0% | 0% | 22% | 33% | 0% | 8% |
| | RPU | 所需功能单元 | 24 | 48 | 12 | 24 | 24 | 0 | 132 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 8% | / | 8% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 62 |
| | | 功能单元利用率 | 0% | 0% | 0% | 67% | 25% | 0% | 16% |
| GOST | | 算法包含操作 | 1 | 1 | 0 | 1 | 1 | 0 | 4 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 8% | 8% | 0% | 8% | 8% | / | 7% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 8% | 8% | 0% | 4% | 8% | 0% | 5% |
| | COBRA | 所需功能单元 | 8 | 12 | 4 | 12 | 2 | 4 | 42 |
| | | 功能单元利用率 | 13% | 8% | 0% | 8% | 50% | 0% | 10% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 6% | 3% | 0% | 6% | 6% | / | 5% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 14% | 14% | 0% | 17% | 25% | 0% | 13% |
| TEA | | 算法包含操作 | 4 | 2 | 0 | 2 | 0 | 0 | 8 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 25% | 13% | 0% | 13% | 0% | / | 10% |
| | RCPA | 所需功能单元 | 16 | 16 | 16 | 32 | 16 | 16 | 112 |
| | | 功能单元利用率 | 25% | 13% | 0% | 6% | 0% | 0% | 7% |
| | COBRA | 所需功能单元 | 32 | 48 | 16 | 48 | 8 | 16 | 168 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 功能单元利用率 | 13% | 4% | 0% | 4% | 0% | 0% | 5% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 25% | 6% | 0% | 13% | 0% | / | 9% |
| | 本文 | 所需功能单元 | 9 | 9 | 4 | 8 | 5 | 5 | 41 |
| | | 功能单元利用率 | 43% | 21% | 0% | 25% | 0% | 0% | 19% |
| XTEA | | 算法包含操作 | 3 | 2 | 0 | 2 | 0 | 0 | 7 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 25% | 17% | 0% | 17% | 0% | / | 12% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 25% | 17% | 0% | 8% | 0% | 0% | 8% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 19% | 8% | 0% | 8% | 0% | 0% | 8% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 19% | 6% | 0% | 13% | 0% | / | 8% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 43% | 29% | 0% | 33% | 0% | 0% | 23% |
| SPECK | | 算法包含操作 | 1 | 2 | 0 | 2 | 0 | 0 | 5 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 8% | 17% | 0% | 17% | 0% | / | 8% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 8% | 17% | 0% | 8% | 0% | 0% | 6% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 6% | 8% | 0% | 8% | 0% | 0% | 6% |
| | RPU | 所需功能单元 | 12 | 24 | 6 | 12 | 12 | 0 | 66 |
| | | 功能单元利用率 | 8% | 8% | 0% | 17% | 0% | / | 8% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 14% | 29% | 0% | 33% | 0% | 0% | 16% |
| SIMON | | 算法包含操作 | 0 | 3 | 0 | 4 | 0 | 0 | 7 |
| | Cyptor | 所需功能单元 | 12 | 8 | 8 | 8 | 8 | 0 | 44 |
| | | 功能单元利用率 | 0% | 38% | 0% | 50% | 0% | / | 16% |
| | RCPA | 所需功能单元 | 16 | 8 | 8 | 16 | 8 | 8 | 64 |
| | | 功能单元利用率 | 0% | 38% | 0% | 25% | 0% | 0% | 11% |
| | COBRA | 所需功能单元 | 24 | 12 | 4 | 12 | 2 | 4 | 58 |
| | | 功能单元利用率 | 0% | 25% | 0% | 33% | 0% | 0% | 12% |
| | RPU | 所需功能单元 | 8 | 32 | 8 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 0% | 9% | 0% | 25% | 0% | / | 9% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 43% | 0% | 67% | 0% | 0% | 23% |
| LUCIFER | | 算法包含操作 | 4 | 0 | 1 | 1 | 4 | 0 | 10 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 33% | 0% | 8% | 8% | 33% | / | 17% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 33% | 0% | 8% | 4% | 33% | 0% | 12% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 25% | 0% | 13% | 6% | 25% | / | 11% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 57% | 0% | 33% | 17% | 100% | 0% | 32% |
| CLEFIA | | 算法包含操作 | 0 | 0 | 0 | 6 | 2 | 2 | 10 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 17% | / | 17% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 17% | 17% | 12% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 50% | 25% | 12% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 38% | 13% | / | 11% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 100% | 50% | 50% | 32% |
| ARIA | | 算法包含操作 | 0 | 0 | 0 | 6 | 2 | 2 | 10 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 0% | 75% | 25% | / | 25% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 0% | 38% | 25% | 25% | 18% |
| | COBRA | 所需功能单元 | 8 | 12 | 4 | 12 | 2 | 4 | 42 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 100% | 50% | 24% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 38% | 13% | / | 11% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 100% | 50% | 50% | 32% |
| C2 | | 算法包含操作 | 2 | 2 | 1 | 3 | 1 | 0 | 9 |
| | Cyptor | 所需功能单元 | 20 | 20 | 20 | 20 | 20 | 0 | 100 |
| | | 功能单元利用率 | 10% | 10% | 5% | 15% | 5% | / | 9% |
| | RCPA | 所需功能单元 | 20 | 20 | 20 | 40 | 20 | 20 | 140 |
| | | 功能单元利用率 | 10% | 10% | 5% | 8% | 5% | 0% | 6% |
| | COBRA | 所需功能单元 | 24 | 36 | 12 | 36 | 6 | 12 | 126 |
| | | 功能单元利用率 | 8% | 6% | 8% | 8% | 17% | 0% | 7% |
| | RPU | 所需功能单元 | 24 | 48 | 12 | 24 | 24 | 0 | 132 |
| | | 功能单元利用率 | 8% | 4% | 8% | 13% | 4% | / | 7% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 62 |
| | | 功能单元利用率 | 14% | 14% | 17% | 25% | 13% | 0% | 15% |
| PRESENT | | 算法包含操作 | 0 | 0 | 1 | 2 | 2 | 0 | 5 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 13% | 25% | 25% | / | 13% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 13% | 13% | 25% | 0% | 9% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |
| | RPU | 所需功能单元 | 8 | 16 | 4 | 8 | 8 | 0 | 44 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 功能单元利用率 | 0% | 0% | 25% | 25% | 25% | / | 11% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 33% | 33% | 50% | 0% | 16% |
| MACGUFFIN | | 算法包含操作 | 0 | 0 | 0 | 3 | 2 | 0 | 5 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 0% | 38% | 25% | / | 13% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 0% | 19% | 25% | 0% | 9% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 50% | 0% | 6% |
| | RPU | 所需功能单元 | 8 | 16 | 4 | 8 | 8 | 0 | 44 |
| | | 功能单元利用率 | 0% | 0% | 0% | 38% | 25% | / | 11% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 50% | 0% | 16% |
| SQUARE | | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 33% | / | 20% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 33% | 33% | 14% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 25% | / | 14% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 67% | 100% | 100% | 39% |
| M6 | | 算法包含操作 | 6 | 3 | 0 | 1 | 0 | 0 | 10 |
| | Cyptor | 所需功能单元 | 36 | 36 | 36 | 36 | 36 | 0 | 180 |
| | | 功能单元利用率 | 17% | 8% | 0% | 3% | 0% | / | 6% |
| | RCPA | 所需功能单元 | 36 | 36 | 36 | 72 | 36 | 36 | 252 |
| | | 功能单元利用率 | 17% | 8% | 0% | 1% | 0% | 0% | 4% |
| | COBRA | 所需功能单元 | 32 | 48 | 16 | 48 | 8 | 16 | 168 |
| | | 功能单元利用率 | 19% | 6% | 0% | 2% | 0% | 0% | 6% |
| | RPU | 所需功能单元 | 36 | 72 | 18 | 36 | 36 | 0 | 198 |
| | | 功能单元利用率 | 17% | 4% | 0% | 3% | 0% | / | 5% |
| | 本文 | 所需功能单元 | 21 | 21 | 9 | 18 | 12 | 12 | 93 |
| | | 功能单元利用率 | 29% | 14% | 0% | 6% | 0% | 0% | 11% |
| SHARK | | 算法包含操作 | 0 | 0 | 0 | 2 | 2 | 2 | 6 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 25% | / | 15% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 25% | 25% | 11% |
| | COBRA | 所需功能单元 | 8 | 12 | 4 | 12 | 2 | 4 | 42 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 13% | / | 7% |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 50% | 50% | 19% |
| NUSH | | 算法包含操作 | 2 | 1 | 0 | 2 | 0 | 0 | 5 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 17% | 8% | 0% | 17% | 0% | / | 8% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 17% | 8% | 0% | 8% | 0% | 0% | 6% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 13% | 4% | 0% | 8% | 0% | 0% | 6% |
| | RPU | 所需功能单元 | 12 | 24 | 6 | 12 | 12 | 0 | 66 |
| | | 功能单元利用率 | 17% | 4% | 0% | 17% | 0% | / | 8% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 29% | 14% | 0% | 33% | 0% | 0% | 16% |
| GRANDCRU | | 算法包含操作 | 0 | 0 | 2 | 4 | 4 | 4 | 14 |
| | Cyptor | 所需功能单元 | 12 | 12 | 12 | 12 | 12 | 0 | 60 |
| | | 功能单元利用率 | 0% | 0% | 17% | 33% | 33% | / | 23% |
| | RCPA | 所需功能单元 | 12 | 12 | 12 | 24 | 12 | 12 | 84 |
| | | 功能单元利用率 | 0% | 0% | 17% | 17% | 33% | 33% | 17% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 25% | 25% | 25% | / | 16% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |
| | | 功能单元利用率 | 0% | 0% | 67% | 67% | 100% | 100% | 45% |
| E2 | | 算法包含操作 | 0 | 0 | 0 | 8 | 4 | 0 | 12 |
| | Cyptor | 所需功能单元 | 24 | 24 | 24 | 24 | 24 | 0 | 120 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 17% | / | 10% |
| | RCPA | 所需功能单元 | 24 | 24 | 24 | 48 | 24 | 24 | 168 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 17% | 0% | 7% |
| | COBRA | 所需功能单元 | / | / | / | / | / | / | / |
| | | 功能单元利用率 | / | / | / | / | / | / | / |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 25% | / | 14% |
| | 本文 | 所需功能单元 | 21 | 21 | 9 | 18 | 12 | 12 | 93 |
| | | 功能单元利用率 | 0% | 0% | 0% | 44% | 33% | 0% | 13% |
| KHAZAD | | 算法包含操作 | 0 | 0 | 0 | 2 | 2 | 2 | 6 |
| | Cyptor | 所需功能单元 | 8 | 8 | 8 | 8 | 8 | 0 | 40 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 25% | / | 15% |
| | RCPA | 所需功能单元 | 8 | 8 | 8 | 16 | 8 | 8 | 56 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 25% | 25% | 11% |
| | COBRA | 所需功能单元 | 8 | 12 | 4 | 12 | 2 | 4 | 42 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| | RPU | 所需功能单元 | 16 | 32 | 8 | 16 | 16 | 0 | 88 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 13% | / | 7% |
| | 本文 | 所需功能单元 | 7 | 7 | 3 | 6 | 4 | 4 | 31 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 50% | 50% | 19% |
| HIEROCRYPT-L1 | | 算法包含操作 | 0 | 0 | 0 | 4 | 4 | 4 | 12 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 25% | / | 15% |
| | RCPA | 所需功能单元 | 16 | 16 | 16 | 32 | 16 | 16 | 112 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 25% | 25% | 11% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 17% | 100% | 50% | 14% |
| | RPU | 所需功能单元 | 32 | 64 | 16 | 32 | 32 | 0 | 176 |
| | | 功能单元利用率 | 0% | 0% | 0% | 13% | 13% | / | 7% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 62 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 50% | 50% | 19% |
| HIEROCRYPT-3 | | 算法包含操作 | 0 | 0 | 0 | 8 | 8 | 8 | 24 |
| | Cyptor | 所需功能单元 | 16 | 16 | 16 | 16 | 16 | 0 | 80 |
| | | 功能单元利用率 | 0% | 0% | 0% | 50% | 50% | / | 30% |
| | RCPA | 所需功能单元 | 16 | 16 | 16 | 32 | 16 | 16 | 112 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 50% | 50% | 21% |
| | COBRA | 所需功能单元 | 16 | 24 | 8 | 24 | 4 | 8 | 84 |
| | | 功能单元利用率 | 0% | 0% | 0% | 33% | 200% | 100% | 29% |
| | RPU | 所需功能单元 | 32 | 64 | 16 | 32 | 32 | 0 | 176 |
| | | 功能单元利用率 | 0% | 0% | 0% | 25% | 25% | / | 14% |
| | 本文 | 所需功能单元 | 14 | 14 | 6 | 12 | 8 | 8 | 62 |
| | | 功能单元利用率 | 0% | 0% | 0% | 67% | 100% | 100% | 39% |

# 参考文献

[1] Estrin G，Bussell B，Turn R，et al．Parallel Processing in a Restructurable Computer System[J]．IEEE Transactions on Electronic Computers，1963，EC-12(6)：747-755

[2] DeHon A，Wawrzynek I．Reconfigurable computing what， why， and implications for design autimation[C]．Procedings of the 36th Annual ACM on IEEE Design Automation Conference，1999：610-615

[3] Wien M，Variable block-size transforms for H.264/AVC[J]．IEEE Transactions on Circuits and Systems for Video Technology，2003，13(7)：604-613

[4] Palkovic M，Cappelle H，Glassee M，et al．Mapping of 40 MHz MIMO SDM-OFDM Baseband Processing on Multi-Processor SDR Platform[C]．DDECS 200811th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems，2008：1-6

[5] Majzoub S，Saleh R，Diab H．Reconfigurable Platform Evaluation Through Application Mapping And Performance Analysis[C]．IEEE International Symposium on Signal Processing and Information Technology，2006：496-501

[6] Lee M-H，Singh H，Lu G，et al．Design and Implementation of the MorphoSys Reconfigurable Computing Processor [M]．Field-Programmable Custom Computing Technology： Architectures，Tools，and Applications，2000：21-38

[7] Berekovic M，Kanstein A，Mei BF，et al．Mapping of nomadic multimedia applications on the ADRES reconfigurable array processor[J]．Microprocessors and Microsystems，2009，33(4)：290-294

[8] Miyamori T，Olukotun U．A quantitative analysis of reconfigurable coprocessors for multimedia applications[C]．IEEE Symposium on FPGAs for Custom Computing Machines，1998：2-11

[9] Goldstein S C，Schmit H，Budiu M，et al．PipeRench: a reconfigurable architecture and compiler[J]．IEEE Transactions on Computers，2000，33(4)：70-77

[10] Mei B，Vernalde S，Verkest D，et al．ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix[M]．Field Programmable Logic and Application．Springer Berlin/Heidelberg，2003：61-70

[11] Singh H，Ming-Hau L，Guangming L，et al．MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications[J]．IEEE Transactions on Computers，2000，49(5)：465-481

[12] Baumgarte V，Ehlers G，May F，et al．PACT XPP—A Self-Reconfigurable Data Processing Architecture[J]．The Journal of Supercomputing，2003，26(2)：167-184

[13] Plessl C，Platzner M．Zippy a coarse-grained reconfigurable array with support for hardware virtualization[C]．16th IEEE International Conference on Application-Specific Systems，Architecture Processors，2005：213-218

[14] 王莉．密码算法的可重构系统实现研究[D]：[博士学位论文]．湖南：国防科学技术大学，2004

[15] J Elbirt and C . Paar . An instruction-level distributed processor for symmetric-key cryptography[J]. IEEE Transactions on Parallel and Distributed Systems，2005，16(5)：468-480

[16] D Fronte ， A Perez ， and E Payrat . Celator: A Multi-algorithm Cryptographic Co-processor[C]. International Conference on Reconfigurable Computing and FPGAs，2008：438-443

[17] G Sayilar and D Chiou . Cryptoraptor: high throughput reconfigurable cryptographic processor[C]. IEEE/ACM International Conference on Computer-Aided Design，2014：155-161

[18] Y Ming，Y Ziyu，L Lei，and L Sikun. ProDFA: Accelerating Domain Applications with a Coarse-Grained Runtime Reconfigurable Architecture[C]. Parallel and Distributed Systems (ICPADS)，2012：834-839

[19] Dai Z，Yang X，Ren Q，et al. The research and design of reconfigurable cipher processing architecture targeted at block cipher[C]. IEEE 7th International Conference on ASICON'07，2007：814-817

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

[21] 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，2013：404-407

[22] Van Eijndhoven J T J，Stok L. A data flow graph exchange standard[C] Design Automation，1992. Proceedings. [3rd] European Conference on. IEEE，1992：193-199

[23] Moon，Chiung, et al. "Integrated process planning and scheduling in a supply chain" Computers & Industrial Engineering 54. 4 (2008)：1048-1061

[24] Elbirt A J. Reconfigurable computing for symmetric-key algorithms[J]. 2002：103-108

[25] 刘芳，王玲. 基于动态规划思想求解关键路径的算法[J]. 计算机应用，2006，26(6)：1440-1442

[26] Han Y，Leong P C，Tan P C，et al. Fast algorithms for elliptic curve cryptosystems over binary finite field[M]. Advances in Cryptology-ASIACRYPT99. Springer Berlin Heidelberg，1999：75-85

[27] Hütter M，Großschädl J， Kamendje G A. A versatile and scalable digit-serial/parallel multiplier architecture for finite fields GF ($2^m$)[C]. Information Technology：Coding and Computing [Computers and Communications]，2003. Proceedings. ITCC 2003. International Conference on. IEEE，2003：692-700

[28] Staake T R . Design of a Reconfigurable Hardware Architecture for Encryption Algorithms[D]. Darmstadt University of Technology，2003. 104-106

[29] Nikhil Bansal， Sumit Gupta，Nikil Dutt，and Alexandru Nicolau. Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations . In Workshop on ApplicationSpecific Processors，held in conjunction with the International Symposiumon Microarchitecture (MICRO)，2003：677-683

[30] 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. ACM，

2008：151–160

[31] 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．ACM，2009：191–200

[32] 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，IEEE，2010：145–151

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

[34] 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．IEEE，2003：296–301

[35] 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

[36] Ullmann，Julian R．"An algorithm for subgraph isomorphism．" Journal of the ACM (JACM) 23．1 (1976)：31-42

[37] 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

[38] Gerard K．Rauwerda．Paul M．Heysters．Gerard J．M．Smit．2004．Mapping Wireless Communication Algorithms onto a Reconfigurable Architecture[J]．The Journal of Supercomputing，30：263-282．

[39] YuanqingGuo．Gerard J．M．Smit．Hajo Broersma．et al．A Graph Covering Algorithm for a Coarse Grain Reconfigurable System[C]．ACM SIGPLAN conference on Language，compiler，and tool for embedded systems．2003：332-341

[40] Paul M．Heysters．Gerard K．Rauwerda．Gerard J．M．Smit．Implementation of a HiperLAN/2 receiver on the reconfigurable montium architecture[C]．Parallel and Distributed Processing Symposium．2004：245-254

[41] Gerard J．M．Smit．Paul M．Heysters．Michel Rosien．et al．Lessons Learned from Designing the MONTIUM-a Coarse-Grained Reconfigurable Processing Tile[C]．International Symposium on System-on-Chip．2004：265-271

[42] Gerard K．Rauwerda．Paul M．Heysters．Gerard J．M．Smit．Towards Software Defined Radios Using Coarse-Grained Reconfigurable Hardware[C]．IEEE Transactions On Very Large Scale Integration (VLSI) Systems．2008：324-331

[43] Schneier B．Description of a new variable-length key，64-bit block cipher (Blowfish)[C]//Fast Software Encryption．Springer Berlin Heidelberg，1993：191-204

[44] Biryukov A．Block Ciphers and Stream Ciphers：The State of the Art[J]．IACR Cryptology ePrint Archive，2004，2004：94

[45] Adams C M．Constructing symmetric ciphers using the CAST design procedure[C]//Selected Areas in Cryptography．Springer US，1997：71-104

[46] Fleischmann E，Gorski M，Huhne J H，et al．Key recovery attack on full GOST block cipher with zero time and memory[J]．Published as ISO/IEC JTC，2009：1-32

[47] Rivest R L．The RC5 encryption algorithm[C]//Fast Software Encryption．Springer Berlin Heidelberg，1994：86-96

[48] Kim H，Huh J H，Anderson R．On the Security of Internet Banking in South Korea–a lesson in how not to regulate security[J]．2011：33-54

[49] Schneier B，Kelsey J，Whiting D，et al．The Twofish encryption algorithm：a 128-bit block cipher[M]．John Wiley & Sons，Inc.，1999

[50] Liu F，Ji W，Hu L，et al．Analysis of the SMS4 block cipher[C]//Information Security and Privacy．Springer Berlin Heidelberg，2007：158-170

[51] Rivest R L，Robshaw M J B，Sidney R，et al．The RC6TM block cipher[C]//First Advanced Encryption Standard (AES) Conference．1998：44-54

[52] Biham E，Anderson R，Knudsen L．Serpent：A new block cipher proposal[C]//Fast Software Encryption．Springer Berlin Heidelberg，1998：222-238

[53] Kelsey J，Schneier B，Wagner D．Related-key cryptanalysis of 3-way，biham-des，cast，des-x，newdes，rc2，and tea[J]．Information and Communications Security，1997：233-246

[54] Lu J．Related-key rectangle attack on 36 rounds of the XTEA block cipher[J]．International Journal of Information Security，2009，8(1)：1-11

[55] Phan R C W．Cryptanalysis of full Skipjack block cipher[J]．Electronics Letters，2002，38(2)：1

[56] Beaulieu R，Shors D，Smith J，et al．The SIMON and SPECK lightweight block ciphers[C]//Proceedings of the 52nd Annual Design Automation Conference．ACM，2015：175

[57] Sorkin A．Lucifer，a cryptographic algorithm[J]．Cryptologia，1984，8(1)：22-42

[58] Shirai T，Shibutani K，Akishita T，et al．The 128-bit blockcipher CLEFIA[C]//Fast software encryption．Springer Berlin Heidelberg，2007：181-195

[59] Kwon D，Kim J，Park S，et al．New block cipher：ARIA[M]//Information Security and Cryptology-ICISC 2003．Springer Berlin Heidelberg，2003：432-445

[60] Borghoff J，Knudsen L R，Leander G，et al．Cryptanalysis of C2[M]//Advances in Cryptology-CRYPTO 2009．Springer Berlin Heidelberg，2009：250-266

[61] Bogdanov A，Knudsen L R，Leander G，et al．PRESENT：An ultra-lightweight block cipher[M]．Springer Berlin Heidelberg，2007

[62] Blaze M，Schneier B．The MacGuffin block cipher algorithm[C]//Fast Software Encryption．Springer Berlin Heidelberg，1994：97-110

[63] Daemen J，Knudsen L，Rijmen V．The block cipher Square[C]//Fast software encryption．Springer Berlin Heidelberg，1997：149-165

[64] Kelsey J，Schneier B，Wagner D．Mod n cryptanalysis，with applications against RC5P and M6[C]//Fast

Software Encryption．Springer Berlin Heidelberg，1999：139-155

[65] Kwan M．The design of the ICE encryption algorithm[C]//Fast Software Encryption．Springer Berlin Heidelberg，1997：69-82

[66] Rijmen V，Daemen J，Preneel B，et al．The cipher SHARK[C]//Fast Software Encryption．Springer Berlin Heidelberg，1996：99-111

[67] Stern J，Vaudenay S．Cs-cipher[C]//Fast Software Encryption．Springer Berlin Heidelberg，1998：189-204

[68] Wu W，Feng D．Linear cryptanalysis of NUSH block cipher[J]．Science in China Series F：Information Sciences，2002，45(1)：59-67

[69] Borst J．The block cipher：Grand cru[J]．Primitive submitted to NESSIE，2000：4

[70] Keliher L，Meijer H，Tavares S．High probability linear hulls in Q[C]//Proceedings of Second Open NESSIE Workshop，Royal Holloway College，University of London，Egham，UK．2001

[71] Kanda M，Moriai S，TAKASHIMA Y，et al．E2--a new 128-bit block cipher[J]．IEICE TRANSACTIONS on Fundamentals of Electronics，Communications and Computer Sciences，2000，83(1)：48-59

[72] Barreto P，Rijmen V．The Khazad legacy-level block cipher[J]．Primitive submitted to NESSIE，2000，97

[73] Ohkuma K，Muratani H，Sano F，et al．The block cipher Hierocrypt[C]//Selected Areas in Cryptography．Springer Berlin Heidelberg，2000：72-88