基于 SpinalHDL 和 Cocotb 的 Poseidon 哈希算法硬件加速器的敏捷开发

翁万正, 吴迪, 王璞

(北京达坦科技有限公司 北京 100000)
(wanzheng.weng@datenlord.com)

摘 要: Poseidon 是一种全新的面向零知识证明(ZKP:Zero-Knowledge Proof)密码学协议设计的哈希算法. 相比同类算法, 如经典的 SHA-256 和 Pedersen 哈希函数, 在零知识证明的应用场景下, Poseidon 能够显著地降低证明生成和验证的计算复杂度, 极大提升零知识证明系统的运行效率. Poseidon 哈希函数的计算涉及高位宽模乘和矩阵乘法运算, 需要消耗大量计算资源. 为了提升哈希计算的效率, 本文基于流水线和折叠技术提出了一种面向 FPGA 平台的 Poseidon 硬件加速器架构. 在该整体架构下, 针对高位宽模乘, 我们基于 Karatsuba 乘法拆分算法实现了一种高性能蒙哥马利模乘器. 针对函数中的向量—矩阵乘法计算, 本文基于脉动矩阵结构提出了一种高吞吐率的硬件实现. 最终, 在搭载有 Xilinx Virtex Ultrascale+ FPGA 芯片的 Varium C1100 加速卡上, 硬件加速器每秒最高能完成 0.99M 次哈希计算, 达到 AMD Ryzen 5900X 处理器 Poseidon 计算速度的两倍.

此外, 本文还详细介绍了 SpinalHDL 和 Cocotb 在 Poseidon 加速器设计和验证中的应用. SpinalHDL 是一种基于 Scala 的硬件生成器语言 HCL(HCL: Hardware Construction Language). 借助 Scala 中各种高级语言特性, 如函数式编程、递归和面向对象等, 以及其自身提供的丰富的电路抽象, SpinalHDL 能够极大地提升硬件设计的效率和质量. 而 Cocotb 是基于 Python 的数字电路验证框架, 和传统基于 System Verilog 的验证流程相比, Python 丰富的生态和高效简洁的语法特性, 能够显著地减少搭建参考模型的工作量, 提升硬件验证的效率.

关键词: 区块链; 零知识证明; Poseidon 哈希函数; FPGA; SpinalHDL; Cocotb; 芯片敏捷开发;

Agile Development Of Poseidon Hasher Hardware Accelerator Based On SpinalHDL And Cocotb

Weng Wanzheng, Wu Di, and Wang Pu

(*DatenLord, Beijing* 100000)

Abstract: Poseidon is a novel hash function designed for zero-knowledge Proof (ZKP) cryptography protocols. Compared with fellow algorithms, such as the classic SHA-256 and Pedersen hash function, Poseidon can significantly reduce the computational complexity of proof generation and verification in zero-knowledge proof application, and greatly improve the efficiency of the whole proof system. The computation of Poseidon hash function involves modular multiplications of big integers and vector-matrix multiplications, which require a great amount of computing resources. In order to improve the efficiency of hashing process, this paper proposes a Poseidon hardware accelerator architecture for FPGA platform based on pipeline and folding technology. Based on this architecture, we implement a high-performance Montgomery modular multiplier based on Karatsuba algorithm. And a high-throughput hardware implementation of vector-matrix multiplication based on systolic array is proposed in this paper. In the end, the hardware accelerator has been implemented on Varium C1100 card, which has a Xilinx Virtex Ultrascale+ FPGA chip, and is able to perform up to 0.99M hash calculations per second, twice as fast as AMD's Ryzen 5900X processor.

Additionally, the application of SpinalHDL and Cocotb in Poseidon accelerator design and verification is introduced in detail. SpinalHDL is a Scala-based HCL(Hardware Construction Language). With the high-level language features of Scala, such as functional programming, recursion, and object orientation, as well as the rich circuit library it provides, SpinalHDL can greatly improve the efficiency and quality of hardware design. Cocotb is a Python-based digital hardware verification framework. Compared with the traditional System Verilog based verification process, Python's

rich community and its efficient and succinct syntax can significantly reduce the workload of reference model construction and improve the efficiency of hardware verification.

Key words: Blockchain; Zero-Knowledge Proof; Poseidon Hasher; FPGA; SpinalHDL; Cocotb; Agile Chip Developme

1 引 言

Poseidon 是一种全新的面向零知识证明(ZKP: Zero-Knowledge Proof)密码学协议设计的哈希算法. 相比同类算法，包括经典的 SHA-256、SHA-3 以及 Pedersen 哈希函数，在零知识证明的应用场景下，Poseidon 能够显著降低证明生成和验证的计算复杂度，极大地提升证明系统整体的运行效率. 基于上述优点, Poseidon 目前已被广泛应用在了各种区块链项目当中，包括去中心化存储系统 Filecoin 以及加密货币 Mina Protocol 和 Dusk Network 等，主要用于加速其中的零知识证明计算.

Poseidon 哈希函数的计算流程中涉及高位宽的模乘以及复杂的矩阵乘法运算，需要消耗大量的计算资源，目前主要采用 GPU 对其计算过程进行加速. 由于 Poseidon 哈希函数提出的时间较晚，目前并没有开源的针对该算法的硬件电路实现以及公开发表的相关学术研究的论文.

1.1 Poseidon 和零知识证明

零知识证明(ZKP: Zero-Knowledge Proof)是一类用于证明计算完整性(Computational Integrity)的密码学协议.其基本的思想是：使证明者(Prover)能够在不泄露任何有效信息的情况下向验证者(Verifier)证明某个计算等式(Computational Statement)的成立. 一个具体的证明对象可以表示为如下形式：

$$y = f(x, w) \tag{1}$$

其中f代表某个函数或程序，x代表该函数中公开的输入，w表示需要保密的函数输入，即只能有证明者知晓. 在零知识证明系统下，证明者能够在不泄露 w 的情况下向验证者证明该等式的成立. 由以上描述可见，零知识证明最显著的特点也是其最大的优势在于隐私保护性，证明者可以在不泄露任何重要私密信息的情况下完成对某个函数或程序计算结果的证明.

由于零知识证明能够兼顾计算完整性证明和隐私保护，其具备广泛的应用场景，具体包括匿名线上拍卖[1]、匿名电子投票[2]、云数据库 SQL 查询验证[3]、去中心化数据存储系统等[5]. 尤其是近年来，随着加密货币比特币(Bitcoin)的出现，区块链技术得到了越来越广泛的应用，而零知识证明凭借其出色的隐私保护特性，近年来获得了大量区块链开发者和设计者的青睐，同时也涌现出了许多具体的实现算法，如 ZK-SNARKs, Bulletproofs 以及 ZK-STARKs 等. 其中 ZK-SNARKs 算法已经被应用到了许多实际的区块链项目当中，包括加密货币 Zcash[4] 和 Pinocchio 以及基于 IPFS(IPFS: InterPlanetary File System)协议建立的分布式存储系统 Filecoin 等。

为了保证证明的通用性，零知识证明算法通常不会针对某一特定函数 f 而设计. 因此，如果需要验证公式 1 的成立，需要将具体的 f 函数转换到零知识证明算法所规定的受限表达式系统(Constraint System)当中，这一过程可以理解为“程序编译”，即将待证明的程序转换成另一种由受限表达式(Constraints)组合成的具备相同功能的程序。例如，对于 ZK-SNARKs 算法，需要将待证函数转换成通过特殊的二次多项式 R1CS (R1CS: Rank-1 Quadratic Constraints) 进行描述. 而转换后的函数形式也被称为算数电路(Arithmetic Circuit)，证明者和验证者都需要在该电路的基础上进行一系列的算术运算来完成证明生成和验证的工作.

虽然零知识证明能在提供计算完整性验证的同时兼顾隐私的保护，但其代价是提高了证明生成和验证的计算复杂度.目前，不论对于证明者还是验证者完成一次零知识证明都需要消耗大量的计算资源. 通常情况下，证明生成和验证的计算复杂度和待证函数 f 转换到受限表达式系统后的复杂度，即转换结果所包含的受限表达式的数量成正比关系. 函数经过“编译”后生成的受限表达式的数量越多，所需的证明生成和验证的计算时间越长.

在区块链应用中，待验证的函数 f 通常为某种哈希函数。而本文所加速的 Poseidon 哈希算法对零知识证明做了针对性的优化，使得其对受限表达式的适配程度显著优于同类的算法，例如, Poseidon 转换后所需的受限表达式的数量是 Pedersen 哈希函数的 1/8 左右.

1.2 Filecoin 分布式存储网络

本文所研究的加速器器主要针对的是应用在 Filecoin 分布式存储网络中的 Poseidon 哈希计算实例。

Filecoin 在提供存储服务的过程中需要对数据进行 Poseidon 哈希运算，这一计算过程需要消耗大量的算力，是整个 Filecoin 系统运行的性能瓶颈之一。

1.2.1 Filecoin 概述

Filecoin 是基于区块链技术建立的一种去中心化、分布式、开源的云存储网络[14]。其主网络已平稳运行一年，并展现出了高速、低成本和稳定的特点。由于其开源、去中心化的优点，目前 Filecoin 已经得到了广泛的应用，包括著名的维基百科(Wikipedia)已将其网站的数据库存储在 Filecoin 网络上。

与传统由数据中心提供的云存储服务不同，任何具备空闲存储资源的设备均可以在 Filecoin 网络上提供服务。因此，为了保证存储服务的质量，Filecoin 中设计了包括复制证明 PoRep(Proof Of Replication)和时空证明 PoSt(Proof Of Spacetime)等机制来保证数据存储的完整性和持久性，这些机制中涉及大量的密码学算法，包括 SHA-256 和 Poseidon 哈希函数以及零知识证明 zk-SNARK 等，并需要大量的算力支持。目前 Filecoin 主要是基于 GPU 对这些计算过程进行加速，并没有相关开源 FPGA 硬件加速方案。

1.2.2 Filecoin 软件实现 Lotus

目前，应用最为广泛的 Filecoin 软件实现是由 Protocol Labs 基于 Go 语言编写的 Lotus. 具体来讲，Lotus 是一个在 Linux 操作系统上运行的命令行程序，其所提供的功能包括: 1)上传和下载文件；2)出售存储资源；3)检验存储数据的完整性等。同时，Lotus 中提供了用于测试计算机硬件在 Filecoin 计算负载下性能表现的基准程序 Lotus-Bench。本次课题研究将在 Poseidon 加速电路实现的基础上，为 Lotus 提供访问底层硬件加速器的软件接口，并通过 Lotus-Bench 对 FPGA 加速器的计算性能进行测试和比较。

2 Poseidon 哈希算法

准确地来说，Poseidon 指代的是一类具有相似计算流程的专门针对零知识证明而优化的哈希函数的集合[7].类比编程语言中面向对象的概念,可以将 Poseidon 理解成一个哈希类，在实际应用中，需要初始化 Poseidon 哈希函数类的成员变量生成具体的哈希函数实例. 不同参数对应生成的 Poseidon 实例在计算流程上基本相似，但其安全性、计算量和复杂度都有一定程度的差异. 在使用过程中允许开发者根据不同场景选择最佳的参数组合，生成最适合当前应用需求的哈希算法实例。

Poseidon 哈希函数类所包含的具体参数如表 2-1 和 2-2 所示。其中表 2-1 列出了实例化一个 Poseidon 类需要指定的所有参数，这三个参数可确定一个唯一的哈希计算实例. 而表 2-2 中所列参数均由表 2-1 中的三个参数经过某种运算后得到，它们分别确定了 Poseidon 计算流程的各项具体参数，包括迭代次数、模幂运算的指数、模加运算中的常数等.

表 2-**1** Poseidon 实例化需要的基本参数

参数	含义
p	函数中所有操作数所在有限域的模数
M	哈希函数的安全等级
t	中间状态向量包含的有限域元素的个数

表 2-**2** Poseidon 计算流程的具体参数

参数	含义
α	S-Box 阶段模幂运算的指数，由参数 p 确定
R_F	Full Round 迭代次数，由参数 p 和 M 确定
R_P	Partial Round 迭代次数，由参数 p 和 M 确定
$RoundConstants$	Add Round Constants 阶段的常数，由 p, M, t 共同确定
$\mathcal{M}$	MDS Mixing 阶段矩阵乘法中的常数矩阵，由 t 确定

本文将针对 Filecoin 中使用的 Poseidon 计算实例设计相应加速电路，其三个基本的实例化参数取值如下:

(1) 有限域模数 p:

参数 p 位宽为 255 位，哈希函数中所有算数运算操作均在 p 确定的有限域内完成，即所有的算术运算均为模数为p的模运算，包括 255-bit 的模加和模乘；同时,参数 p 还确定了 S-Box 阶段模幂运算的指数 $\alpha = 5$. p 的具体取值为:

p=0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

(2) 中间状态向量元素个数 t :

参数 $t \in \{3,5,9,12\}$，即 Filecoin Poseidon 实例的中间状态可以包含 3、5、9 或 12 个有限域元素;

(3) 安全等级 M :

Filecoin Poseidon 哈希实例的安全等级 $M = 128\ Bits$，安全等级 M 和参数 p 共同确定了 Full Round 迭代次数 $R_F = 8$ 以及 Partial Round 迭代次数 $R_P \in \{55,56,57,57\}$ (依次对应 t 的四个取值)；除了具体参数外，Poseidon 计算流程中还涉及到两类常数，分别是 Add Round Constants 阶段与中间状态元素相加的常数 $RoundConstants$ 和MDS Mixing阶段与中间状态完成矩阵乘法的常数矩阵 $\mathcal{M}$ ，这两类常数同样由 (p, M, t) 三个基本参数唯一确定；由于计算这两类常数的算法较为复杂，且与哈希函数本身的计算流程以及后续的加速电路的设计没有太大的关联，本文将不对此进行详细地介绍，具体的常数计算可参考 Filecoin Poseidon 实例的官方文档[6]。

2.1 Poseidon 哈希函数特点

上文提到，Poseidon 可以看成是一类相似的哈希函数实例组成的集合，这类哈希函数计算流程的共同特点如下：

(1) 所有运算均在有限域内完成，即均为模运算；
(2) 计算过程中涉及大量的常数；
(3) 整体计算流程由多次重复的迭代组成，这些迭代可以分成 Full Round 和 Partial Round 两类；

上述三点简要概括了 Poseidon 哈希计算的主要特点，也是对应加速电路设计与实现中需要着重考虑的几个地方。

2.2 Poseidon 详细计算流程

从输入和输出的角度看，Poseidon 哈希函数将输入 $(t-1)$ 个有限域元素映射到一个有限域元素输出。其中间计算流程的表述如下:

(1) 初始化：将输入(preimage)的$(t-1)$个元素通过拼接(Concatenation)和填充(Padding)的方式转换成包含 t 个有限域元素的中间状态($state$);
(2) $R_F/2$ 次 Full Round 循环：每次循环需要对中间状态向量 $state$ 的每个元素完成 Add Round Constant，S-Box 和 MDS Mixing 操作，其中 Add Round Constants 需要完成一次常数模加，S-Box 需要计算五次模幂, MDS Mixng 计算 $state$ 向量和常数矩阵 $\mathcal{M}$相乘;
(3) R_P 次 Partial Round 循环：Partial Round 循环和 Full Round 循环的计算流程基本一致，不同点在于 Partial Round 在 S-Box 阶段不需要对$state$向量中所有元素进行计算，只需完成第一个元素的模幂运算即可；
(4) $R_F/2$ Full Round 循环: 与步骤 2 一致；

在依次完成上述步骤 1-4 中总共 $R_F + R_P$ 次循环后，将中间状态向量 $state$ 中的第二个元素作为哈希运算的结果输出.

详细的 Poseidon 哈希算法定义如下：

Algorithm1: Poseidon Hash Function：

$\boldsymbol{R = R_F + R_P;\ R_f = R_F/2;}$
INPUT: $preimage: Z_p^{[t-1]}$
Output: $digest: Z_p$
1. $state: Z_p^{[t]} = DomainTag||preimage||Padding$
2. $For\ r \in [0, R)$:
3. For i ∈ [0, t):
4. state$[i]$ = state$[i]$ ⊕ RoundConstant$_r[i]$
5. If $r \in [0, R_f)$ or $r \in [R_f + R_P, R)$
6. For $i \in [0, t)$: $state[i] = state[i]^{\alpha}$
7. Else: $state[0] = state[0]^{\alpha}$
8. $state = state \times \mathcal{M}$
9. $digest: Z_p = state[1]$
10. Return $digest$

Poseidon 哈希算法的主体部分，即上述算法中 2-8 步所对应的计算流程图如下:

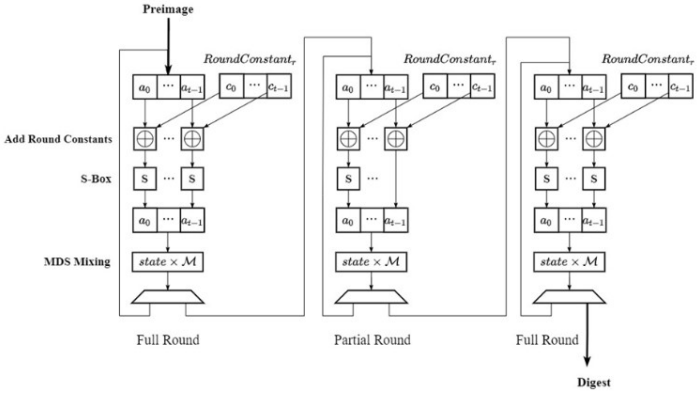

图 2-1 Poseidon 哈希函数计算流程图

3 基于 SpinalHDL 和 Cocotb 的硬件开发

近年来，随着芯片制程的缩小逐渐逼近物理极限，通过工艺迭代获得芯片性能提升的方式变得不再有效。因而针对具体应用场景设计专用加速电路来满足性能需求的方式逐渐成为芯片设计的趋势。但同时领域专用架构 DSA(DSA: Domain Specific Architecture)的设计方式也对硬件开发的效率和质量提出了更高要求，而目前传统的基于 Verilog 和 System Verilog 的设计和验证方式已经很难满足人们对高效开发的需求。

因此，如何实现芯片的敏捷开发获得了越来越广泛的关注，与此同时也涌现出了以 Chisel 为代表的各种新兴的硬件描述语言和验证工具.与 Chisel 类似，本文使用的 SpinalHDL 是一种基于 Scala 的硬件生成器语言 HCL(HCL:Hardware Construction Language)。而 Cocotb 则是基于 Python 的硬件验证框架。这两个新兴的设计和验证工具的使用极大提升了 Poseidon 加速器的开发效率和质量。

本节将结合 Poseidon 电路设计阐述 SpinalHDL 和 Cocotb 在硬件设计和验证上的优势：

3.1 SpinalHDL 和 Cocotb 概述

3.1.1 SpinalHDL 概述

SpinalHDL 是一种基于高级编程语言 Scala 的硬件生成器语言 HCL(HCL: Hardware Construction Language). 对于开发者来说，SpinalHDL 通常可以分成两部分: Scala 语法和 SpinalHDL 提供的电路元件库。SpinalHDL 基于 Scala 语法中的类和函数提供了硬件设计所需的基本电路元件的抽象，包括寄存器、逻辑门、多路选择器、译码器和算数运算电路等。而开发者完成的工作是：基于 Scala 语法来描述电路的整体结构，即描述各个基本电路元件间的连接关系。由于 Scala 是一门多范式的编程语言，其支持函数式编程、面向对象、递归等高级的语言特性以及丰富的

集合类型，能够赋予设计者强大的结构建模能力和代码参数化的能力。基于 SpinalHDL 的详细硬件开发流程如图 3-1 所示.

3.1.2 Cocotb 概述

Cocotb 是一种开源的基于 Python 的数字电路验证框架，该验证框架的具体工作模式如图 3-2 所示。基于 Cocotb 的电路验证可以分为如下两个部分:

(1) 基于 Python 编写的测试平台代码：基于 Python 中的协程(Coroutine)并行地完成：产生待测电路输入端的激励信号；将相同的激励信号传递给参考模型得到标准输出；监测待测电路的输出端口，并检查输出结果是否和标准输出一致；

(2) 支持标准 GPI 编程接口的电路仿真器：通过 GPI 编程接口接收 Python 测试代码生成的输入端激励信号，并对待测电路 DUT(DUT: Design Under Test) 进行功能仿真

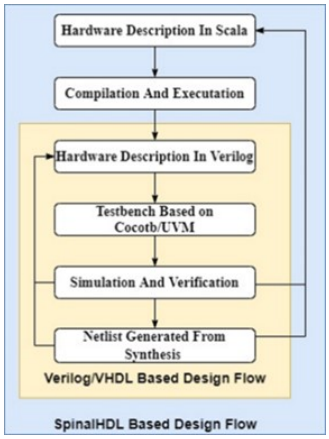

图 3-1 基于 SpinalHDL 的硬件开发流程

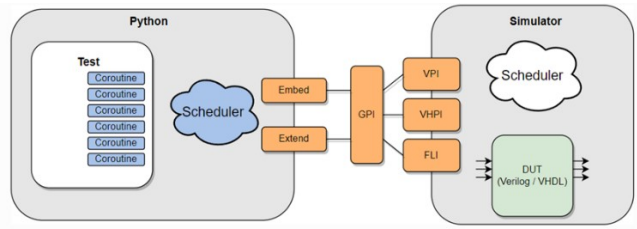

图 3-2 Cocotb 验证框架[12]

3.2 SpinalHDL 在硬件设计中的优势

3.2.1 结构级电路描述方式

和高层次综合 HLS 不同，SpinalHDL 虽然也是基于高级编程语言 Scala 进行电路设计，但其本质上仍然是从结构级对电路进行描述，Verilog 中所有可综合语法元素在 SpinalHDL 中都有对应的实现. 对于开发者，如果不使用 Scala 中的高级语言特性以及 SpinalHDL 中抽象层次较高的一些电路模型，如 FIFO、计数器和总线仲裁器等，完全可以像使用 Verilog 一样对电路进行建模。

3.2.2 更可靠的电路描述方式

使用 SpinalHDL 进行电路描述能够提升设计的可靠性，显著降低代码出错的概率。而代码可靠性的提升不仅能减轻电路验证的压力，同时对后期的代码维护也有很大的帮助。SpinalHDL 所带来的可靠性的提升主要表现在如下几个方面:

(1) 更加准确的电路模型：以寄存器的实现为例，Verilog 中并没有直接对应的语法元素，Verilog 关键字“reg”既可以被综合成时序逻辑也可用于实现组合逻辑。而 SpinalHDL 将寄存器模型直接加入到了语法层面上，使用“Reg/RegNext”关键字即可直接实例化出寄存器电路. 避免了 Verilog 中重复实现“reg”对应的“always”块代码. SpinalHDL 在信号的类型上也有更加精确的区分，和 Verilog 中只有“wire”关键字不同，SpinalHDL 可以根据信号的具体行为对其进行进一步地划分，包括“Bool”—布尔值、“UInt”—无符号整型、“SInt”—有符号整型，“Bits”—多比特信号等;

(2) 设计规则 DRC(DRC: Design Rule Check)检查：SpinalHDL 可以在 Scala 代码的编译和运行阶段进行电路设计规则的检查，规避电路实现中容易出错的地方，包括：1) 不同位宽信号的连接；2) 将寄存器实现成锁存器；3) 和信号类型不符的行为，如对“Bits”信号进行算数运算操作。

3.2.3 更强的建模和表达能力

SpinalHDL 对电路强大的建模和表达能力主要源于 Scala 提供的各种丰富的高级语言特性，包括函数式编程、面向对象、递归等:

(1) 函数式编程：硬件设计描述的往往是一种结构而非一个串行的流程。而函数式编程则是基于变量之间的映射关系对问题进行建模，这种建模思维更加适用于电路结构的设计。

(2) 递归语法：数字电路的结构在很多情况下具备非常规整的递归结构，尤其是数字信号处理相关的模块。Scala 中递归的编程特性能够帮助设计者高效地完成相关电路结构的描述，同时也有助于提升代码的复用能力。

此外，SpinalHDL 自身提供的丰富的电路抽象使开发者避免了很多底层模块的重复实现，进而能够从更高的层次构建电路。SpinalHDL 中不仅提供了最基本的电路元件如信号、逻辑运算符、算数运算符和寄存器等，同时也实现了诸如 FIFO、AXI 总线、计数器和状态机等较高层级的电路模型。

3.2.4 更好的代码复用能力

除了更加强大的电路建模能力外，SpinalHDL 还能为硬件设计带来更好的复用性。基于 Scala 面向对象的语法特点，开发者可以对设计进行高度的参数化，如本文在 Poseidon 的代码开发中将表 2-1 和 2-2 中的参数统一实现在一个配置类当中，有助于处理参数之

间的依赖关系、检查参数值的合法性以及方便参数的调用等。

3.3 Cocotb 在硬件验证中的优势

3.3.1 Python 高效简洁的语法

相比传统硬件验证使用的语言，包括 Verilog、VHDL 和 System Verilog，Python 具备更加高效和简洁的表达能力。即使是和 C/C++以及 Java 这些软件编程中的高级语言相比，Python 在代码开发效率上也有很大的优势。此外，作为一门高级编程语言，Python 同样支持诸如面向对象等高级语言特性，能提供很好的抽象和参数化的能力，帮助验证人员构建复用性更高的验证代码。

以本次课题中 Poseidon 加速的验证为例，Poseidon 哈希算法需要完成大位宽数据(255 位)的算术运算。而包括 C/C++以及 Java 等大部分编程语言一般最高只支持 64 位整数运算，而 Python 中的整数类型支持任意位宽的数据，这一点大大降低了 Poseidon 软件参考模型实现的复杂度

3.3.2 Python 丰富的软件生态

同时 Python 拥有强大的生态和丰富的开源代码库。使用 Cocotb 进行硬件设计的验证，可以很方便地复用这些现有的基于 Python 的算法实现，作为硬件电路的参考模型。例如，在设计深度学习模型加速器时，可以基于 Pytorch、Tensorflow 等 Python 库搭建验证用的参考模型；对于 SoC 设计中涉及的各种总线协议，如 AXI 协议等，Cocotb 中也提供了相应的开源软件实现；使用 Python 现成的开源软件包不仅能够极大地减轻搭建参考模型的工作量，同也可降低代码出错的概率。

除了在参考模型搭建上的优势，借助许多开源的 Python 包开发者更好地编写、组织和执行验证代码。例如，**cocotb-test** 使开发者能够以 Python 函数的方式与仿真器进行交互；而在软件开发中常用的测试框架 **pytest** 也能够帮助验证人员更好的管理和组织硬件电路的测试; **pytest-xdist** 和 **pytest-parallel** 包则可以实现在多核处理器中并行地运行验证代码。

4 加速系统整体设计

针对 Filecoin 中 Poseidon 哈希函数的加速，本文所设计的整体加速系统如图 4-1 所示。整个系统主要分为 CPU 服务器和 FPGA 加速卡两个主体。CPU 服务器运行 Filecoin 具体软件实现 Lotus 程序提供数据存储服务，当需要进行 Poseidon 哈希函数的计算时，服务器通过 FPGA 加速器的软件接口函数，以 DMA 的方式将数据传输到 Poseidon 硬件加速器当中进行哈希计算。Poseidon 加速器完成计算后以相同的方式将哈希结果传回处理器内存中。

其中，FPGA 硬件系统主要由三个部分组成：

1. Xilinx XDMA IP：实现数据从 CPU 侧内存到 FPGA 的搬运；同时负责 PCIe 外设总线到 FPGA 片内 AXI-Stream 总线的转换;
2. 异步 FIFO：实现 XDMA IP 到 Poseidon 加速器间跨时钟域的数据传输；XDMA 输出的总线时钟固定为 250MHz，而 Poseidon 加速器 IP 的工作频率在 100-200MHz;
3. Poseidon 加速器：整个 FPGA 加速卡的核心部分，负责 Poseidon 哈希函数的计算加速;目前，系统中只实现了 Poseidon 哈希函数的加速器，还可以通过增加 XDMA 通道数以接入其他算法的硬件加速单元；

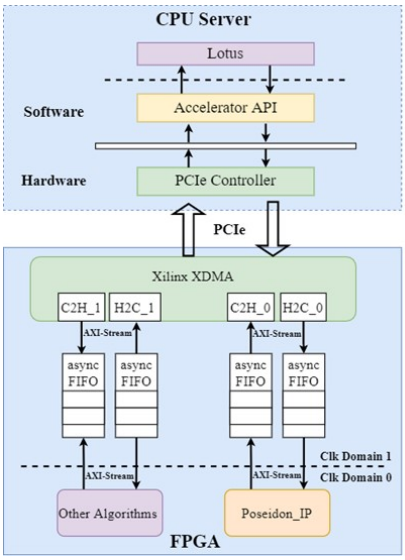

图 4-1 硬件加速系统设计

5 Poseidon 加速器 IP 设计

在上述加速系统整体架构设计的基础上，本节将介绍其中的核心模块 Poseidon 算法硬件加速 IP 的设计与实现细节。

对于任意算法的硬件加速器，其设计与实现大体上都可以分成单元运算电路和整体硬件架构两部分。对于单元运算电路而言，如加法器、乘法器和模乘器等，其设计的主要目标是如何在更少的硬件开销下达到更高的计算性能。而在单元运算电路的基础上，硬件架构设计需要考虑的是如何组织好每个运算单元以提高其利用率，进而使加速器整体达到更高的数据吞吐率。

5.1 单元运算电路设计

Poseidon 哈希函数的基本运算涉及模加和模乘两种，下文将分别介绍其具体的实现细节。

5.1.1 模加电路设计

模加运算的数学定义如下:

$$a \oplus b = (a + b) \bmod p$$

任意的模运算都可以分解成普通算术运算和取余两部分. 对于模加，需要先完成一次普通的加法，然后还需将加法结果缩减到模数所限制的范围内. 最基础的实现方式如算法 3 所示，通过减法将操作数的和缩减到模数范围内. 其对应硬件电路由加法器、减法器、比较器和多路选择器组成，详细的数据通路为：输入操作数经过一个加法器后得到加法结果，将加法结果同时传递给减法器和比较器，分别得到减去模值和与模值比较的结果，将比较结果作为多路选择器的选通信号对加法器和减法器的输出进行仲裁后输出，详细电路结构如图 5-1(a)所示。

另一种电路实现方式的原理如算法 4 所示. 这种实现方式将操作数的和与$(2^k - p)$再次相加完成取余. 其对应的硬件电路仅由两个加法器和一个多路选择器组成，具体数据通路为：输入操作数经过第一个加法器，输出两数之和与进位，两数之和继续输入第二个加法器与 $(2^k - p)$ 相加后得到对应的和与进位，将两次加法的进位位相或后，作为多路选择器的选通信号对两次加法的和进行总裁后输出，详细电路结构见图 5-1 (b).

Algorithm 4: Adder Based Modular Addition

Modulus: $p\ k-bit$
INPUT: $a, b \in [0, p)$
OUTPUT: $(a + b)\ mod\ p$
1. $carry_0, sum_0 = a + b$
2. $carry_1, sum_1 = (sum_0 \bmod 2^k) + (2^k - p)$
3. **IF** $carry_0$ ***OR*** $carry_1$:
4. **Return** $\boldsymbol{sum_1}$
5. **ELSE:**
6. **Return** sum_0

上述两种实现方式都是建立在两个加/减法器和一个多路选择器的基础上，两个加/减法器分别实现普通加法和取余操作，然后通过多路选择器仲裁后输出. 而第一种实现方式需要额外的比较器产生选通信号，在电路实现上，比较器通常有较大的硬件开销并且会产生较长的组合逻辑延时. 尤其是对于 Poseidon 当中 255 比特高位宽的操作数，比较器对整体模加电路的性能—面积比的影响会更加显著。同时，额外的比较器也不利于对模加电路进行流水线化的设计。因此，本文采用了第二种实现方式。

在具体电路实现中，由于 Poseidon 操作数位宽为 255 比特，单周期高位宽加法会产生较长的组合逻辑延时. 为了提升电路工作频率，我们将图 5-1 (b) 中的两个 255 比特加法器进行了流水线处理，每个加法器设置了五个周期的流水线延迟，并在多路选择器的输出端添加一级寄存器，使得整体模加电路总共包含 11 级流水延迟. 流水线处理后的电路结构如图 5-1 (c)所示.

Algorithm 3: Modular Addition

Modulus: p
INPUT: $a, b \in [0, p)$
OUTPUT: $(a + b)\ mod\ p$
1. $sum = a + b$
2. **IF** $sum \geq p$:
3. **Return** $sum = sum - p$
4. **ELSE:**
5. **Return** sum

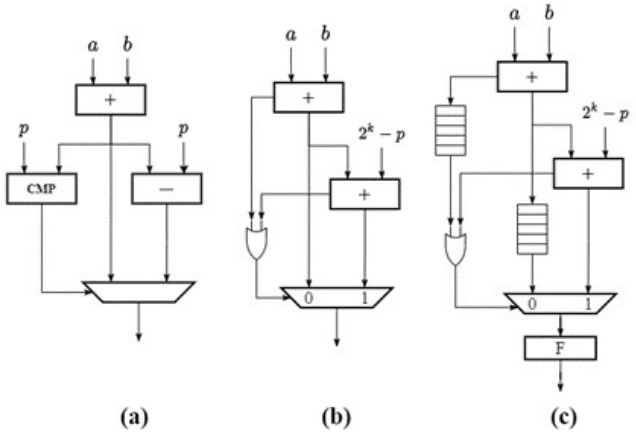

图 5-1 模加器电路结构

5.1.2 模乘电路设计

模乘同样可以分成一次普通乘法和取余运算两部分. 但和模加不同，其取余通常需要在除法的基础上完成，为了避免除法器实现导致大量的硬件资源消耗和延时，本文选择了基于 Montgomery 算法的模乘实现。

Montgomery 算法的基本思想是通过数域转换的方式，将操作数和运算结果转换到 Montgomery 域内，在该数域内取余的除法操作可以简化成移位运算. 具体的数域转换方式：当模运算的模数为 p 时，操作数 a, b 对应 Montgomery 域内的取值$a^{'}, b^{'}$为：

$$a^{'} = a \cdot R\ mod\ p$$
$$b^{'} = b \cdot R\ mod\ p$$

其中 R 需要满足 $R = 2^k > p$ 且 R 和 p 互质，即 $gcd(R, p) = 1$；在该数域内的乘法，即 Montgomery 乘法操作的定义如下：

由$R = 2^k$可知，上述算法中 2 - 4 步的取余和除法操作均可以由移位代替，而 p' 在R 和 p 固定的情况下可以提前计算，5-6 步的取余操作可以通过一次加法或减法实现，因此，整个 Montgomery 模乘算法总共需要完成三次乘法和两次加法/减法. 对于 Montgomery 模乘输出 c' 可以将其通过以下公式，即计算 MontPro$(c', 1)$, 将结果转换回常数域，也可以继续在 Montgomery 域内完成后续乘法运算。

$$c = c' \cdot R^{-1} \ mod \ p$$

该算法对应硬件实现主要包括两个部分，首先需要完成 255 比特高位宽普通乘法器的设计；其次，在乘法器的基础上，根据 Montgomery 算法流程搭建模乘器的整体架构。

对于 256 比特高位宽乘法器，本文提出的解决方案是将其分解成多个并行的低位宽乘法器，然后通过调用 FPGA DSP 单元中嵌入的低位宽乘法器实现.最简单的乘法拆分方式是基于经典的级联算法实现，每次拆分将一个乘法器分解为四个并行的位宽减半的乘法器。对于 256-bit 乘法器，经过四次递归的拆分，可通过 256 个 16 比特的乘法器实现。这种拆分方式虽然实现的电路结构简单规整，但拆分后所需乘法单元的数量仍然不够理想。本文采用了 Karatusba-Ofman 算法进行乘法器拆分[23]，这种分解方式虽然在电路结构上相对复杂并且会引入额外的加法器，但每次分解只需要三个位宽减半的乘法器，对于 256 比特乘法，三次拆分后总共需要 81 个 16 比特乘法器，大概是普通级联算法的三分之一。Karatusba-Ofman 算法定义如下：

Algorithm 5: $Karatsuba - Ofman \ Multiplication$

INPUT: $x \ n-bit, \ y \ n-bit$
OUTPUT: $x \times y \ 2n-bit$
1. $m = n/2; \ x = x_1 2^m + x_0; \ y = y_1 2^m + y_0$
2. $x \times y$
3. $= (x_1 2^m + x_0)(y_1 2^m + y_0)$
4. $= x_1 y_1 2^{2m} + (x_0 y_1 + x_1 y_0) 2^m + x_0 y_0$
5. $= x_1 y_1 2^{2m} + [(x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0] 2^m + x_0 y_0$

上述算法所对应的乘法器拆分结构如图 5-2 所示，每次拆分后乘法器位宽减半，但数量上增加两倍，同时需要引入额外的加/减法器. 基于递归的思想，图中的三个乘法器还可以不断地进行拆分，直至位宽满足需求. 本文对 Poseidon 加速器的 255 比特乘法器进行了三次拆分，最终分解为 27 个并行的 34 比特乘法器。而 34 比特的乘法器则通过调用 Xilinx 提供的乘法器 IP 实现。具体的 IP 配置中，本文中选择了基于 DSP 的实现方式，每个 34 比特的乘法器 IP 由 4 个 DSP 模块中 27×8 乘法器组合而成。

Function: $\boldsymbol{MontPro(a', b')}$

INPUT: a', b'
OUTPUT: $c' = a' \cdot b' \cdot R^{-1} \ mod \ p$
1. $t = a' \cdot b'$
2. $t' = t \ mod \ R$
3. $m = t' \cdot p' \ mod \ R$
4. $u = (t + m \cdot p)/R$
5. **IF** $u \geq p$: **return** $u - p$
6. **ELSE: return** $\boldsymbol{u}$

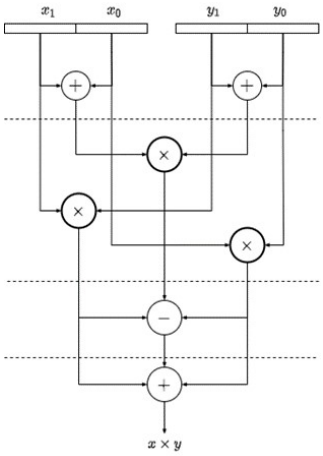

图 5-2 Karatsuba 乘法器拆分结构

为了提升乘法器的工作频率，Poseidon 加速器中对 Karatsuba 乘法器拆分架构进行了流水线处理，图 5-2 中的虚线代表一级寄存器，每进行一次拆分乘法器内增加 3 级流水，而最底层的 Xilinx 乘法器 IP 内设置了 5 级流水。因此，255-bit 的乘法器经过三级拆分后总共的流水线级数为 14.

在 255 比特普通乘法电路的基础上，实现 Montgomery 模乘器的另一步是根据 Montgomery 模乘算法设计整体的硬件架构. 由上文所示的 Montgomery 模乘算法可知，其计算流程总共包括三次级联的普通乘法和两次加法运算. 具体的架构实现思路可以大致分成两类: 1)折叠: 通过时分复用的方式，在单个普通乘法器上完成三次级联的乘法运算；2)展开: 实现三个串行的普通乘法器，每个乘法器完成对应的一次乘法操作；对于折叠的设计方式，每个 Montgomery 模乘器只需要消耗一个普通乘法器，但由于采用时分复用的方式完成三次乘法操作，每完成一次 Montgomery 模乘都至少需要三个时钟周期，同时在电路设计上，需要添加额外的寄存器暂存乘法输出，以及设计相应的控制电路对乘法器输入进行仲裁. 对于展开的设计思路，三个级联的普通乘法操作在电路上分别对应三个串行的乘法器，虽然在硬件开销上是第一种设计的三倍，但这种设计方式每个周期都能产生一个 Montgomery 模乘结果，且电路结构简单不需要实现复杂的控制逻辑. 基于计算性能上的考虑，Poseidon 加速器中采用了第二种实现方式，

具体的电路结构如图 5-3 所示。

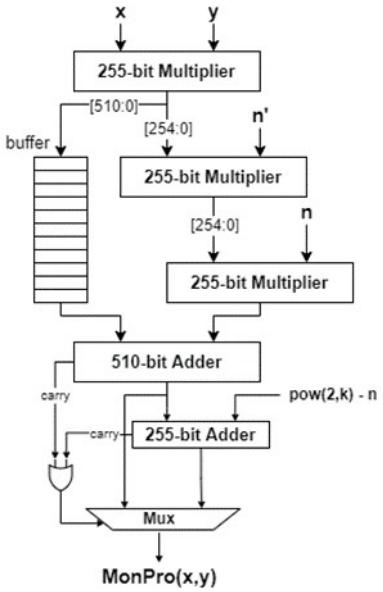

图 5-3 Montgomery 乘法器整体架构

5.2 向量-矩阵乘法电路

Poseidon 算法每次迭代的 MDS Mixing 阶段都需要完成中间状态向量 $state$ 和常数矩阵相乘的运算，本文基于脉动矩阵架构设计了一种结构规整、高吞吐率的向量—矩阵乘法电路.

脉动矩阵架构指的是由一系列相同处理单元 PE(Processing Engine)按照规则的排列和连接方式构成的完成某种计算功能的电路[24]，主要用于实现卷积、矩阵乘法和滤波等非常规整的计算流程. 脉动矩阵有两个非常显著的特点：1) 每个计算节点 PE 完全相同; 2)每个节点都只与其相邻的节点进行通信. 由于脉动矩阵规整和模块化的特点，非常适合于大规模电路的综合实现. 因此，基于脉动矩阵架构的电路设计通常能达到非常高的工作频率和数据吞吐率。

Poseidon 哈希函数所要完成的向量—矩阵乘法的数学定义如下:

$$\text{state} \times M = [a_0 \quad \cdots \quad a_{n-1}] \times \begin{bmatrix} b_{0,0} & \cdots & b_{0,n-1} \\ \vdots & \ddots & \vdots \\ b_{n-1,0} & \cdots & b_{n-1,n-1} \end{bmatrix}$$

输出向量的每个元素是输入向量元素与常数矩阵对应列中元素相乘累加后的结果. 基于上述算法定义，脉动矩阵中每个处理单元 PE 的电路结构如图 5-4 所示，输入输出信号的关系如下:

$$sum = sum + element \cdot constan$$
$$element = element$$

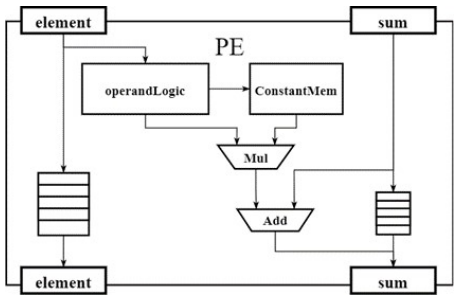

图 5-4 向量-矩阵乘法中每个 PE 的电路结构

在上述处理单元 PE 的基础上，脉动阵列的组织方式如下图所示. 整个阵列由 12 个处理单元 PE 级联而成，输入向量的元素从第一个 PE 的 element 端口依次输入矩阵. 每个 PE 在每个周期完成一次乘累加运算并将结果传递给下一个处理单元. 最后一个 PE 的 sum 端口依次输出结果向量的每个元素.

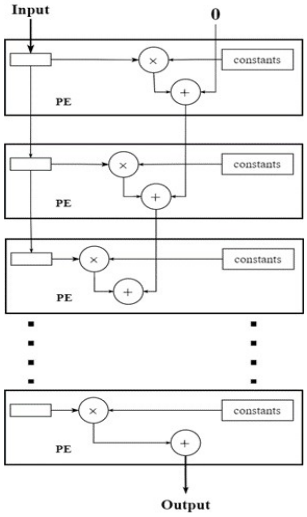

图 5-5 脉动矩阵整体结构

5.3 加速器架构设计

本文所加速的 Filecoin Poseidon 哈希实例的输入为 $t \in \{3,5,9,12\}$ 个有限域元素，每个元素的的位宽为 255 比特. 具体的计算流程由 R_F 次 Full Round 循环和 R_P 次 Partial Round 循环组成。两种循环的计算流程基本相似，都依次包括 AddRoundConstant、SBox 和 MDSMixing 三个阶段，这三个阶段分别完成常数模加、五次方模幂和向量—矩阵乘法，两者唯一的区别在于 Partial Round 在 Sbox 阶段只需要完成中间状态第一个元素的计算. Full Round 和 Partial Round 一次循环的计算流程分别如图 5-6(a)和(b)所示.

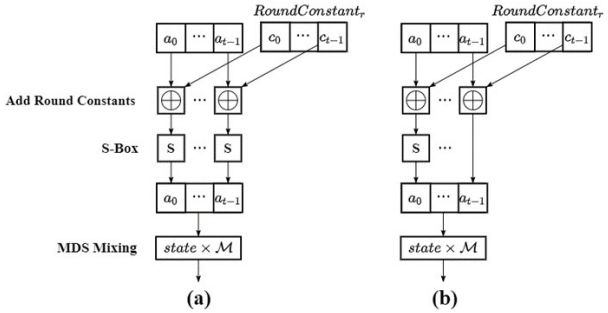

图 5-6 Full Round 和 Partial Round 计算流程

基于上述 Poseidon 算法流程的定义，本文所设计的加速器 IP 的具体硬件架构如图 5-7 所示. 在图 5-6 所示的单次迭代算法流程的基础上，硬件实现针对具体的 FPGA 架构特点和资源限制做了如下几点优化：

(1) 流水线处理：

在数据通路的实现中，为了提高加速器的工作频率，设计中对其进行了流水线化的处理。数据通路主要由模乘和模加单元组成，Poseidon 加速器对每个模运算单元的组合逻辑路径都进行了充分的切割，其中模加运算包含了 11 级流水，而 Montgomery 模乘器分割成了 43 级流水，整个计算通路总共由两百级左右的流水组成.

(2) 串行数据处理:

在 Poseidon 硬件加速器的实现中，中间状态的 t 个元素串行通过每个硬件计算单元，即一个周期只完成一个有限域元素的计算。而采用串行数据处理方式的主要原因包括:

1) 硬件资源的限制: MDS Mixing 阶段中如果并行地完成向量-矩阵乘法运算则需要同时实现 12×12 个模乘器，其硬件资源的开销远远超过 FPGA 板卡的限制.
2) 输入数据的形式：Poseidon 加速器由 XDMA IP 提供输入数据，而 XDMA 输出总线位宽为 255 比特，即每个周期最多只能完成一个有限域元素的传输.
3) 运算单元利用率: 中间状态向量的元素个数 t 有四种取值，加速器需要兼容不同的大小的中间状态，而在这种情况下，串行数据处理的方式能够最大化每个计算单元的利用率.

(3) 折叠的设计思路

Poseidon 哈希算法总共由 $R_F + R_P$ 次迭代组成，对于 Filecoin Poseidon 实例，每次哈希计算，根据中间状态大小的不同，总共需要 63 至 65 次循环. 由于硬件资源的限制，无法将所有循环都在硬件上展开. 因此需要基于折叠的思路，即时分复用的方式，在有限的循环单元上完成 Poseidon 哈希算法的所有迭代.

本次课题所设计的硬件加速器总共实现了两个并行的 Poseidon 迭代单元 PoseidonLoop，每个 PoseidonLoop 都可以完成图 5-6 所示的一次 Partial Round 或 Full Round 计算. 整个 PoseidonLoop 的数据通路呈环状结构，数据输入该模块后，在环状数据通路中不断地流动，直至所有迭代完成后输出。

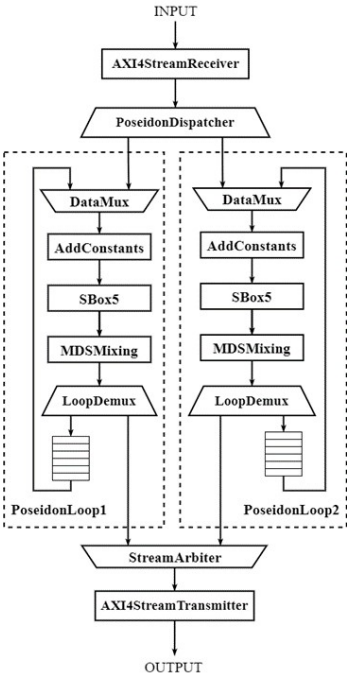

图 5-7 Poseidon 加速器器架构

6 性能测试结果

在上述电路设计的基础上，我们将整体硬件系统通过 Vivado 集成开发环境实现在了 Varium C1100 FPGA 加速卡上，该板卡搭载了 Xilinx Virtex UltraScale+系列的 FPGA 芯片，具体芯片型号为 XCU55N-FSVH2892-2L-E.

6.1 资源利用和时序信息

整体硬件电路综合后的资源消耗如下表所示：

表 6-1 Poseidon 加速器综合后硬件资源使用情况

Resources	Used	Total	Percentage
CLB LUT	533050	871680	61.15%
CLB Registers	835492	1743360	47.92%
LUT As Memory	87548	403200	21.72%
Block Ram Tile	61.5	1344	5%
DSP Slices	4108	5952	70.01%
Carry 8	54657	108960	50.16%
F7 Muxes	1213	435840	0.20%
F8 Muxes	37	217920	0.01%

各项资源中 DSP Slices(70.01%)和 LUT(61.15%)的消耗最多，主要用于 255 比特模乘电路的实现上。这两项资源的不足也限制了在加速器中配置更多模运算电路来提升计算的并行度和整体的加速性能.

在时序上，实现(Implementation)后 Poseidon 加速器刚好能够满足 100MHz 工作频率的要求。关键路径上，建立(set up)时间的余量为 0.069ns，保持(hold)时间的余量为 0.01ns.

6.2 性能测试结果

在上述 FPGA 实现的基础上，本文分别测试了其在 Poseidon 计算负载下的数据吞吐率以及在 Lotus-Bench 上的性能表现.

6.2.1 数据吞吐率测试

本文在 Xilinx 提供的 XDMA 驱动的基础上编写了简单的性能测试程序. 该测试程序向 FPGA 加速器写入一定数量某一长度的输入数据，并记录加速器完成所有数据哈希运算所需要的时间。当输入数据的大小为 arity2，即中间状态大小为 3 时，加速器在 0.877 秒内完成了 850000 次的哈希运算，数据吞吐率可达到 29.1651MB/s，即每秒大约能够完成 1M 次 Poseidon 哈希计算.

表 6-2 不同长度输入下加速器的数据吞吐率

Size	Num	Duration(s)	Transfer Rate(MB/s)	Hash Rate
Arity 2	850000	0.877	29.1651	0.99M
Arity 8	250000	0.697	11.4776	358K
Arity 11	100000	0.328	9.7505	305K

6.2.2 Lotus-Bench 性能测试：

Filecoin 的软件实现 Lotus 中提供了测试计算机硬件在 Filecoin 计算负载下性能表现的基准程序 Lotus-Bench. 与自己编写的测试代码相比，Lotus-Bench 的测试更加接近实际的工作负载，能够得到更加有效的测试结果. 基于 Lotus-Bench，本文分别测试比较了 CPU, GPU 和 FPGA 在 Filecoin preCommit 阶段(该阶段主要完成 Poseidon 哈希函数的计算)处理 512MB 数据所需的时间. FPGA 在 Lotus-Bench 测试下的算力可达到 15.65MB/s，大约是 AMD Ryzen 5900X CPU 实现的 2 倍，但和 RTX 3070 GPU 相比仍有很大的提升空间；

表 6-3 Lotus-Bench 性能测试结果

Device	Model	Size (MB)	Duration (s)	Transfer Rate (MB/s)
GPU	Nvidia RTX 3070	512	10.855	47.17
CPU	AMD Ryzen 5900X	512	40.723	7.41
FPGA	Xilinx Varium C1100	512	32.713	15.65

7 总结

本文主要介绍了基于 SpinalHDL 和 Cocotb 的 Poseidon 哈希算法硬件加速器的敏捷设计. SpinalHDL 是基于 Scala 的一种硬件生成器语言，而 Cocotb 是基于 Python 的数字电路验证框架. 两者都是新兴的硬件设计和验证工具，它们的基本思想都是通过引入软件开发中先进的设计理念和各种高级编程语言特性来提升硬件开发的效率。

Poseidon 哈希函数是一种新兴的面向零知识证明应用的哈希函数。由于其计算过程中涉及大位宽的模乘以及矩阵运算，需要消耗大量的计算资源。为了提升哈希计算的效率，本文基于流水线和折叠技术提出了一种面向 FPGA 平台的 Poseidon 硬件加速器架构. 在该架构的基础上，我们基于 Karatsuba 乘法拆分算法实现了一种高性能的蒙哥马利模乘器.同时利用脉动矩阵架构实现了向量—矩阵乘法电路. 最终，在搭载有 Xilinx Virtex Ultrascale+ FPGA 芯片的 Varium C1100 加速卡上，该硬件加速器每秒最高能完成 0.99M 次哈希计算，是 AMD Ryzen 5900X 处理器 Poseidon 计算速度的两倍.

参考文献(References):

[1] Galal H S, Youssef A M. Verifiable sealed-bid auction on the ethereum blockchain[C]//International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2018: 265-278.

[2] Zhao Z, Chan T H H. How to vote privately using bitcoin[C]//International Conference on Information and Communications Security. Springer, Cham, 2015: 82-96.

[3] Zhang Y, Genkin D, Katz J, et al. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 863-880.

[4] Hopwood D, Bowe S, Hornby T, et al. Zcash protocol specification[J]. GitHub: San Francisco, CA, USA, 2016: 1.

[5] Psaras Y, Dias D. The interplanetary file system and the filecoin network[C]//2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S). IEEE, 2020: 80-80.

[6] Filecoin's Poseidon Specification. GitHub: https://github.com/filecoin-project/neptune/blob/master/spec/poseidon_spec.pdf.

[7] Grassi L, Khovratovich D, Rechberger C, et al. Poseidon: A New Hash Function for {Zero-Knowledge} Proof Systems[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 519-535.

[8] Zhang Y, Wang S, Zhang X, et al. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021: 416-428.

[9] Reitwiessner C. zkSNARKs in a nutshell[J]. Ethereum blog, 2016, 6: 1-15.

[10] Bachrach J, Vo H, Richards B, et al. Chisel: constructing hardware in a scala embedded language[C]//DAC Design automation conference 2012. IEEE, 2012: 1212-1221.

[11] " SpinalHDL: a scala-based HDL", https://github.com/SpinalHDL/SpinalHDL

[12] "Cocotb: a coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python", https://github.com/cocotb/cocotb

[13] Loeliger J, McCullough M. Version Control with Git: Powerful tools and techniques for collaborative software development[M]. " O'Reilly Media, Inc.", 2012.

[14] Psaras Y, Dias D. The interplanetary file system and the filecoin network[C]//2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S). IEEE, 2020: 80-80.

[15] Protocol Labs. Lotus Documentation. https://lotus.filecoin.io/

[16] Xilinx, Inc. (DS1003) Varium C1100 Compute Adaptor Data

Sheet v1.0 [R]. September 17, 2021.
[17] Xilinx, Inc. (UG1525) Varium C1100 Compute Adaptor Installation Guide v1.0 [R]. September 17, 2021.
[18] Xilinx, Inc. (PG195) DMA/Bridge Subsystem for PCI Express Product Guide v4.1[R]. April 29, 2021.
[19] Knezevic M, Vercauteren F, Verbauwhede I. Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods[J]. IEEE Transactions on Computers, 2010, 59(12): 1715-1721.
[20] Montgomery P L. Modular multiplication without trial division[J]. Mathematics of computation, 1985, 44(170): 519-521.
[21] Beuchat J L. Some modular adders and multipliers for field programmable gate arrays[C]//Proceedings International Parallel and Distributed Processing Symposium. IEEE, 2003: 8 pp.
[22] Xilinx, Inc. (UG579) UltraScale Architecture DSP slices User Guide v1.11[R]. August 30, 2021.
[23] Gong Y, Li S. High-throughput FPGA implementation of 256-bit Montgomery modular multiplier[C]//2010 Second International Workshop on Education Technology and Computer Science. IEEE, 2010, 3: 173-176.
[24] Kung H T. Why systolic architectures?[J]. Computer, 1982, 15(01): 37-46.