华南理工大学

South China University of Technology

专业学位硕士学位论文

适用于 RISC-V 的高面积效率浮点运算器设计

作 者 姓 名	赖书浩
学 位 类 别	电子信息硕士
指 导 教 师	贺小勇 副教授
	胡剑琛 高级工程师
所 在 学 院	微电子学院
论文提交日期	2023 年 4 月

Design of floating-point unit with high area efficiency for RISC-V

A Dissertation Submitted for the Degree of Master

Candidate：Lai Shuhao

Supervisor：Prof. He Xiaoyong

Senior Engineer Hu Jianchen

South China University of Technology

Guangzhou, China

分类号：TN4　　学校代号：10561

学　号：202021059461

华南理工大学硕士学位论文

适用于 RISC-V 的高面积效率浮点运算器设计

作者姓名：赖书浩　　指导教师姓名、职称：贺小勇副教授、胡剑琛高级工程师

申请学位级别：电子信息硕士　　学科专业名称：电子信息

研究方向：数字集成电路设计

论文提交日期：2023 年 4 月 24 日　　论文答辩日期：2023 年 5 月 27 日

学位授予单位：华南理工大学　　学位授予日期：　　年　　月　　日

答辩委员会成员：

主席：

委员：

摘　要

随着信息化时代的到来，从低端移动设备到嵌入式领域，或是高端的航天航空设备，无不对数据的处理提出了更高的要求。浮点数相比定点数表示精度更高、表示范围更大，更满足信息化场景的需求，但浮点运算相比定点运算更为复杂导致硬件设计困难，对浮点运算器进行研究是有必要的。

本文旨在设计适用于 RISC-V 的高面积效率浮点运算器，高面积效率可实现更低成本。其符合 IEEE 754-2019 标准，可执行 RISC-V 中的多种浮点运算包括加、减、乘、乘加、乘减、除、开平方和浮点数与整数的互相转换，并支持 RISC-V 规定的舍入模式和异常处理。其支持多种精度的向量浮点运算，可并行执行四个半精度，或两个单精度，或一个双精度浮点运算。

本文细致分析浮点算法，着重优化运算器的面积和速度以提高面积效率，合理规划流水线。为优化浮点加减法中对阶延时长的问题，基于“并行化”思想设计并行移位器。另外在乘加中基于前导 0 和后导 0 检测提出一种用于负结果修正的快速附加位求解算法，并将修正引起的尾数加 1 与舍入合并处理以将负结果修正从长延时路径移除，减少加 1 电路的使用。为减小加法器开销，将除法和开平方的求阶逻辑统一并优化乘加使用的大位宽加法器。针对过往研究中为支持向量运算和多种精度而导致大面积的问题，基于“硬件隔离”的思想，采用多种精度融合的尾数向量运算结构，设计尾数向量加减、向量乘、向量除与开平方和向量移位电路。硬件可根据不同位宽尾数的运算要求进行合理拆分，各尾数运算模块只需使用一套硬件资源便可实现多种精度浮点数的尾数向量运算，从而实现更小面积。

本课题采用 Chisel 进行设计，使用 System Verilog 搭建验证平台以进行功能验证。在 TSMC 7nm 工艺下完成综合，本文设计的浮点运算器最高工作频率为 2631.58MHz，相比现有浮点运算器提高 5%~37%，面积为 5963.93μm^2，相比现有浮点运算器减小 33%~58%，经计算面积效率提高 63%~144%。最后通过形式验证和 FPGA 板级验证，实现一个适用于 RISC-V 的高面积效率浮点运算器。

关键词：浮点运算器；RISC-V；IEEE 754；多种精度；向量运算

Abstract

With the advent of the information age, from low-end mobile devices to embedded fields, or high-end aerospace equipment, all put forward higher requirements for data processing. Compared with fixed-point number, floating-point number has higher precision and wider range of representation, which can better meet the requirements of information scenarios. However, floating-point arithmetic is more complex than fixed-point arithmetic, resulting in difficulties in hardware design. It is necessary to research floating-point unit.

The aim of this paper is to design a high area efficiency floating-point unit for RISC-V, where high area efficiency can achieve lower cost. It is compliant with the IEEE 754-2019 standard, which can perform various floating-point operations in RISC-V, including addition, subtraction, multiplication, multiply-add, multiply-subtract, division, square root, and conversion between floating-point numbers and integers, and supports the rounding modes and exception handling specified by RISC-V. It supports multiple-precision vector floating-point operations, which can perform four half-precision, or two single-precision, or one double-precision floating-point operation in parallel.

This article provides a detailed analysis of the floating-point algorithm and focuses on optimizing the area and speed of the arithmetic unit to improve area efficiency, and reasonably plans the pipeline. In order to optimize the issue of long delay caused by exponent matching in floating-point addition and subtraction, a parallel shifter is designed based on the idea of "parallelization". In addition, a fast additional bit solving algorithm for negative results is proposed based on leading zero and trailing zero detection in multiply-add. The addition of 1 to the significand caused by the correction is combined with rounding to remove the negative result correction from the long delay path and reduce the use of the circuit for adding 1. To reduce the cost of the adder, the calculating exponent logic of division and square root is unified and the large bit-width adder used in multiply-add is optimized. Aiming at the problem of large area caused by supporting vector operation and multiple-precision in previous research, based on the idea of "hardware isolation", a multiple-precision fused significand vector operation structure is adopted to design significand vector addition and subtraction, vector multiplication, vector division and square root, and vector shift circuits. The hardware can be reasonably divided according to the operation requirements of different bit-width significand, and each significand operation module only needs to use one set of hardware resources to realize the significand vector operation of multiple-precision floating-point number, so as to achieve smaller area.

This project is designed using Chisel, and the testbench is built using System Verilog to perform functional verification. The synthesis is completed under the TSMC 7nm process. The maximum operating frequency of the floating-point unit designed in this paper is 2631.58MHz, which is 5%-37% higher than existing floating-point units, and the area is 5963.93μm^2, which is 33%-58% lower than existing floating-point units, and the calculated area efficiency is 63%-144% higher. Finally, it passes the formal verification and FPGA board level verification, and realizes a floating-point unit with high area efficiency which can be used for RISC-V.

Keywords: Floating-Point Unit; RISC-V; IEEE 754; Multiple-Precision; Vector Operation

目 录

第一章 绪论.....1
1.1 研究背景和意义.....1
1.2 国内外研究现状.....3
1.2.1 浮点运算器的出现和发展.....3
1.2.2 RISC-Ⅴ浮点运算器的现状.....5
1.2.3 存在的问题.....8
1.3 主要研究内容.....8
1.4 论文结构安排.....9
第二章 浮点协议标准.....11
2.1 IEEE 754 标准概述.....11
2.2 浮点数表示格式.....11
2.3 浮点数数值分类.....12
2.4 基于 RISC-Ⅴ的浮点数舍入规则.....13
2.5 基于 RISC-Ⅴ的浮点异常生成.....14
2.6 本章小结.....16
第三章 浮点运算算法研究.....17
3.1 浮点加减算法.....17
3.1.1 浮点加减算法流程.....17
3.1.2 前导 0 预测.....18
3.1.3 前导 0 检测.....22
3.2 浮点乘加算法.....23
3.2.1 浮点乘法算法流程.....23
3.2.2 浮点乘加算法流程.....24
3.2.3 适用于乘加的对阶移位.....26
3.2.4 阵列乘法器.....27
3.2.5 基 4 的 Booth 乘法器.....28
3.3 浮点除法算法.....30
3.3.1 浮点除法算法流程.....30
3.3.2 恢复余数除法器.....32
3.3.3 不恢复余数除法器.....33
3.3.4 Newton-Raphson 除法器.....35
3.3.5 尾数除法算法对比.....36
3.4 浮点开平方算法.....37
3.4.1 浮点开平方算法流程.....37

3.4.2 不恢复余数开平方器.....37
3.4.3 Newton-Raphson 开平方器.....39
3.5 浮点数与整数的互相转换.....39
3.5.1 浮点数转整数.....39
3.5.2 整数转浮点数.....41
3.6 本章小结.....42
第四章 浮点运算器的设计与优化.....43
4.1 浮点运算器的整体架构.....43
4.2 浮点加减法与格式转换模块的设计与优化.....44
4.2.1 浮点加减法与格式转换模块整体架构.....44
4.2.2 基于速度优化的并行右移器.....46
4.2.3 尾数加减法的 SIMD 优化.....48
4.2.4 尾数左移器的 SIMD 优化.....50
4.3 浮点乘加模块的设计与优化.....52
4.3.1 浮点乘加模块的整体架构.....52
4.3.2 部分积的位宽优化.....55
4.3.3 尾数乘法的 SIMD 优化.....55
4.3.4 基于 4-2CSA 的部分积累加.....57
4.3.5 大位宽加法优化.....58
4.3.6 负结果修正优化.....59
4.4 浮点除法与开平方模块的设计与优化.....60
4.4.1 浮点除法与开平方模块的整体架构.....60
4.4.2 求阶码电路的设计与优化.....63
4.4.3 尾数除法与开平方的 SIMD 优化.....65
4.4.4 流水线阻塞.....68
4.5 本章小结.....69
第五章 系统仿真与验证.....70
5.1 浮点运算器的功能验证.....70
5.1.1 验证平台设计.....70
5.1.2 浮点运算功能验证结果.....71
5.2 浮点运算器的逻辑综合.....75
5.3 浮点运算器的形式验证.....77
5.4 浮点运算器的板级验证.....77
5.5 本章小结.....82
总结与展望.....83
1. 全文总结.....83
2. 工作展望.....84
参考文献.....85

图表清单

图 1-1 浮点指令使用频率分布 3
图 1-2 CELL FPU 架构图 4
图 1-3 FPnew 向量运算的 paraller 实现 6
图 1-4 FPnew 向量运算的 merge 实现 7
图 2-1 浮点数表示格式 12
图 3-1 浮点加减法流程图 17
图 3-2 前导 0 优先编码逻辑 22
图 3-3 浮点乘法流程图 23
图 3-4 浮点乘加流程图 25
图 3-5 e_c-e_{ab}≥56 时 f_c 的移位过程 26
图 3-6 e_c-e_{ab}<56 时 f_c 的移位过程 26
图 3-7 阵列乘法器运算流程 27
图 3-8 阵列乘法器硬件结构图 27
图 3-9 尾数乘法部分积生成逻辑 29
图 3-10 七级 CSA 网络 30
图 3-11 浮点除法流程图 31
图 3-12 恢复余数和不恢复余数算法电路结构 32
图 3-13 可控加减法单元结构图 34
图 3-14 不恢复余数除法器 CAS 阵列 35
图 3-15 浮点数转整数流程图 40
图 3-16 整数转浮点数流程图 41
图 4-1 浮点运算器整体架构 43
图 4-2 浮点数存储格式 44
图 4-3 浮点加减法与格式转换模块整体架构 45
图 4-4 并行右移器结构图 47
图 4-5 加减法器的数据结构 48
图 4-6 向量加减法器硬件结构 49
图 4-7 向量左移器的数据结构 51

图 4-8 32bits 复合左移器结构图 52
图 4-9 浮点乘加模块整体架构 53
图 4-10 Booth 部分积符号位扩展优化 55
图 4-11 用于生成部分积的数据结构 56
图 4-12 尾数乘法共享部分积示意图 57
图 4-13 4-2CSA 结构图 58
图 4-14 改进的大位宽加法器 59
图 4-15 浮点除法与开平方模块整体架构 61
图 4-16 reg_q 数据结构 66
图 4-17 执行 10 比特尾数开平方的 CAS 输入 67
图 4-18 执行两个 4 比特尾数开平方的 CAS 输入 67
图 4-19 执行 10 比特尾数除法的 CAS 输入 67
图 4-20 执行两个 4 比特尾数除法的 CAS 输入 67
图 4-21 单级 CAS 阵列结构图 68
图 5-1 验证环境框图 70
图 5-2 浮点加减法运算仿真波形图 71
图 5-3 有符号整数转浮点数运算仿真波形图 71
图 5-4 浮点数转有符号整数运算仿真波形图 72
图 5-5 无符号整数转浮点数运算仿真波形图 72
图 5-6 浮点数转无符号整数运算仿真波形图 73
图 5-7 浮点乘加、乘减运算仿真波形图 73
图 5-8 浮点除法运算仿真波形图 74
图 5-9 浮点开平方运算仿真波形图 74
图 5-10 形式验证测试结果 77
图 5-11 浮点加减法板级验证结果 78
图 5-12 有符号整数转浮点数板级验证结果 78
图 5-13 浮点数转有符号整数板级验证结果 79
图 5-14 无符号整数转浮点数板级验证结果 79
图 5-15 浮点数转无符号整数板级验证结果 80
图 5-16 浮点乘加、乘减板级验证结果 80

图 5-17 浮点除法板级验证结果 81
图 5-18 浮点开平方板级验证结果 81
表 2-1 RISC-V 的舍入模式编码 13
表 2-2 源浮点数非 sNaN 的无效操作 15
表 3-1 浮点减法 $a>b$ 情况下的前导 0 预测表 20
表 3-2 浮点减法 $a<b$ 情况下的前导 0 预测表 21
表 3-3 基 4-Booth 乘法编码表 28
表 3-4 低两个比特 CAS 真值表 38
表 3-5 单精度浮点数转整数的范围和无效输入的行为 41
表 4-1 浮点除法与开平方运算吞吐率表 69
表 5-1 浮点运算大规模功能验证结果 74
表 5-2 各 FPU 中运算模块面积（μm^2） 75
表 5-3 各 FPU 速度和面积效率对比 76
表 5-4 各 FPU 功耗情况（mW） 77

第一章 绪论

1.1 研究背景和意义

目前正处于一个信息爆炸的时代，随着各种高精尖科学技术和社会经济的快速发展，社会已经进入信息化的新时代。特别是计算机技术的出现和发展，直接推动了整个信息化产业的快速发展，也推进了集成电路技术的不断进步。在对数据量要求大、对数据精度要求高、对数据范围要求广的图像处理、信号传输和航天航空等信息化领域，集成电路芯片都发挥着重要的作用。

处理器研发是计算机技术中关键的一环，处理器中的数值通常用二进制的 0 和 1 来表示，分为两种类型，即定点和浮点。定点数的小数点位置固定不变，浮点数的格式则与科学计数法是相似的。定点数的优势在于运算逻辑简单，这使得定点运算的部件比较容易设计，劣势就是定点数的数值表示范围较小，对数据的可表示精度也较低。而浮点数可以动态地调整小数点的位置，从而可以更准确地表示数据。浮点数可表示的数据范围比定点数要大得多，且精度更高，但浮点运算算法比定点算法复杂得多，在硬件设计上存在更大的困难。在对于精度和数据范围要求较低的场景下，定点数可以满足要求。而一些对数字处理有着高速实时要求的领域，对于运算的数值范围和精度的要求都比较高，例如图像处理、气候建模、傅里叶变换和航空航天、神经网络等领域，此时使用浮点数很有必要[1, 2]。在这种背景下，浮点运算器（Floating-Point Unit，FPU）必不可少。

对于浮点运算器的实现，目前主要有三种实现方式。在最开始对浮点运算有需求的阶段，许多浮点运算器的功能可以通过软件库来仿真和实现。此阶段对数据处理的实时性要求较低，且芯片的硬件资源相对较少，软件实现是最可取的方式。但随着对计算速度的要求越来越高，软件实现越来越难以满足要求，为了提高系统性能，需要为浮点运算设计专门的硬件模块，即第二种实现方法。这些浮点模块会被挂载到主处理器上，当主处理器识别到浮点数时，便将它送往这些模块进行处理。这种硬件结构也称为协处理浮点运算器，虽然有些耗费硬件资源，但以此来保证浮点运算的高效执行。而随着集成电路工艺和半导体技术的高速发展，工艺特征尺寸越来越小，实现片上系统（System On Chip，SOC）成为可能。浮点运算器开始作为功能部件直接集成到处理器中，这种方法可以实现更快的数据传输速度，但也对于面积提出了更高的要求。

对于高速应用场景，很多浮点运算器通过增加运算单元的数量来提高并行度，以面积换速度，此时相比于速度，其他指标并不是那么重要。反之，一些应用场景下却更看

重面积开销。对于嵌入式设备和低功耗的移动设备，由于设备内部的硬件资源有限，浮点运算器必须实现更小面积。而随着信息技术的发展，这些设备在实现小面积的同时也对速度提出了更高的要求，增加了设计难度。

一些信息化场景要求支持多种精度的浮点运算，不同精度的浮点数位宽不一样，计算误差也不一样。对于低端 DSP 应用或对单个节点要求精度不高的神经网络计算，半精度可满足要求，且低精度具有更高效的运算速率。对于计算的数值范围大且要求结果精确的图像处理、人工智能和科学计算等领域，双精度和单精度浮点数应用更为广泛。支持多种精度使运算器可根据工作场景即时切换浮点精度，但芯片的空间资源有限，多套独立精度的浮点运算部件尽管保证了足够快的速度，但势必占用大量的硬件资源[3]。

标量运算器只能单指令完成一个浮点操作，而随着数据量的增加，以及对速度的高要求，人们倾向于实现向量运算器。64 位的向量运算器可单指令完成一个双精度，或两个单精度，或四个半精度浮点操作，这也称为单指令多数据流（Single Instruction Multiple Data，SIMD）。实时 3D 图形显示要求单精度性能大大超过传统处理器，而向量运算是提高数据吞吐率最低成本的方法，具有更高效的运算能力[4, 5]。AMD 和 Intel 处理器便采用单指令多数据流扩展指令集（Streaming SIMD Extensions，SSE）取代了早期的 x87 指令集[6]。硬件设计上常通过增加多套运算逻辑实现多种精度的向量运算，这往往会增加运算器的面积。而通过循环迭代多次运算实现向量化能够实现小面积，但对速度和吞吐率的提升非常有限。

综上浮点运算器需要在精度范围、速度和面积上取得平衡，本课题在实现多种精度的向量浮点运算的基础上，着重优化面积和速度，提高浮点运算器的面积效率。面积效率即性能/面积，相同速度下高面积效率运算器的面积开销更小，可实现更低的成本，具备实用意义，本文以速度与面积的比值评估面积效率。区别于传统的 ARM、MIPS 等架构，RISC-V 是加州大学伯克利分校的团队所提出的全新处理器架构，它完全开源，且对传统架构中暴露的问题已进行充分的研究并加以规避[7]。RISC-V 与闭源、授权费高昂的 ARM 等架构形成差异化竞争，且架构精简而易于实现，吸引了国外的麻省理工学院、印度理工学院和国内的中科院等研究机构投入到 RISC-V 相关芯片的研发。NVIDIA 等科技巨头也对 RISC-V 抱以厚望并加大研发投入，各国也相继推出自研的 RISC-V 处理器，研究适用于 RISC-V 的实用和高面积效率芯片是具有市场竞争力的[8]。RISC-V 包含浮点运算指令，本课题对 RISC-V 的浮点运算功能进行实现，实现的运算器可兼容其他 RISC-V 架构的芯片，作为浮点运算模块使用，具备研究性和实用性。

1.2 国内外研究现状

1.2.1 浮点运算器的出现和发展

基于信息化发展的需要，浮点运算器已是大多数硬件设计中重要的定制应用之一。Intel 最早着眼于为浮点制定统一标准，8087 是 Intel 公司于 1980 年设计的第一款 x87 系列浮点协处理器，用于和 x86 系列的 Intel 8086 微处理器协同工作以加速处理器中的浮点运算，x87 本质上是 x86 的浮点处理单元。x87 给后世产品提供了一个可参考的标准，使用 x87 指令可以实现 32 位、64 位和 80 位的一些基本的浮点运算，图 1-1 展示了浮点运算的种类以及它们在浮点系统中的使用频率。之后 Intel 相继开发了新的产品，开始尝试直接将浮点加速单元集成在处理器中，包括 80486DX、Pentium 等产品，这对于提高运算速度以及节约功耗是非常有效的，但也对浮点运算器的面积提出了更高的要求。

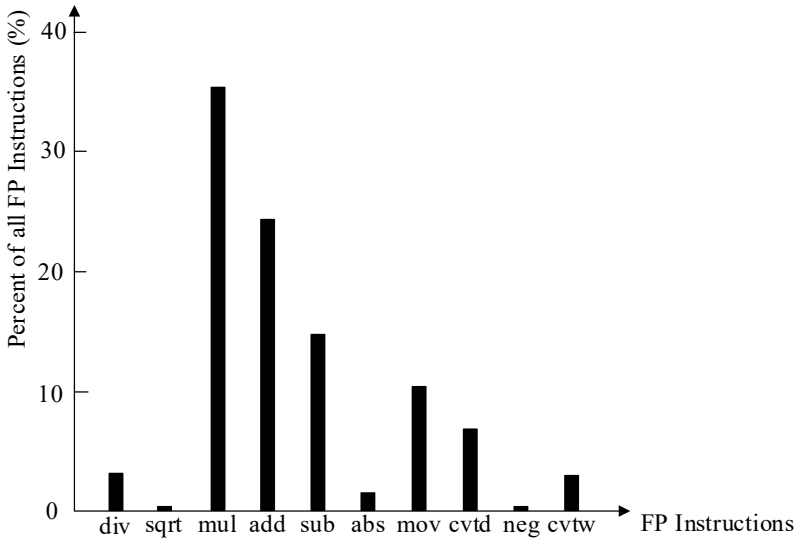

图 1-1 浮点指令使用频率分布[9]

IBM 的浮点运算器则基于 POWER 指令集架构进行开发，POWER7 的 FPU 整体架构采用六级流水线，兼容标量运算和向量运算，包含四路并行单精度 SIMD 以及两路并行的双精度 SIMD，并实现了浮点乘加指令（Floating-Point Multiply-Add 或 Fused Multiply-Add，FMA），可见对浮点运算器的需求考虑是比较全面的[10]。如图 1-1，浮点指令中使用最频繁的是浮点加减法和乘法，且在程序中乘法运算之后常会接着执行加减法。为了提高浮点处理器的运算能力，现代处理器倾向于加入浮点乘加单元，将 *opa*×*opb*+*opc* 只通过一条指令实现以提高速度，且由于乘加省去了一次舍入操作，精度损失也更小，一些设计则使用乘加单元完全替代了浮点加法器和乘法器[11]。

2005 年 IBM 联合 Toshiba 和 Sony 设计的 Cell 芯片最高可在 4GHz 下工作，其 FPU 内配备四套单精度 FPU（SPfpu）和一套双精度 FPU（DPfpu），作为协处理单元加速浮

点运算，流水线深度为六或七级[4]。FPU 架构如图 1-2 所示，实现了向量运算但面积仍较大。随后在 2006 年，Intel 公布了世界上第一款速度可以达到每秒万亿次浮点运算的“万亿级”研究原型芯片，此阶段浮点运算器在运行速度和吞吐率上取得了不错的成果。

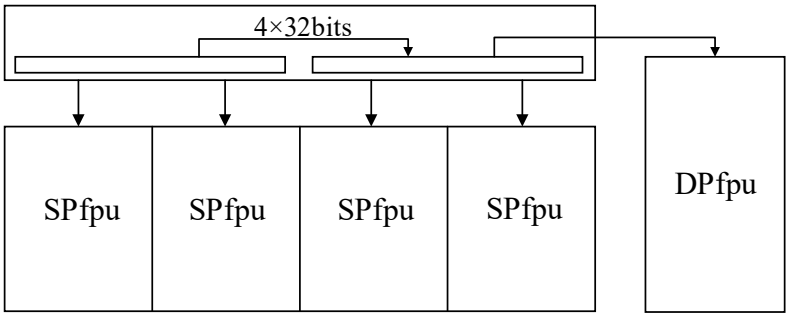

图 1-2 CELL FPU 架构图

为提高浮点运算能力，一些设计采用集成多个浮点运算单元的方法。如 NVIDIA 基于 Kepler 架构设计的 GK104 GPU 芯片，内部包含 1536 个单精度浮点运算单元和 64 个双精度浮点运算单元，实现了高的并行度，强化了浮点运算能力[12]。但这么多运算单元的引入势必会增加芯片面积，要求单个浮点运算单元的面积应尽量小。

而我国对于浮点运算器的研究起步是比较晚的，但随着国家对于科学技术发展的重视，特别是近年来对于集成电路产业的不断支持，我国也取得了不少成果。

2004 年，清华大学设计的 THUMP 芯片面世，最高工作频率为 500MHz，这是我国自主研发的 32 位微处理器。随后不少的国产芯片陆续发布，我国的中科院计算所先后推出龙芯 1 号、龙芯 2 号和龙芯 3 号。龙芯 2 号于 2005 年发布，支持单精度和双精度的浮点向量运算，能执行大部分的浮点运算，内部包含浮点加减模块、乘法模块和除法与开平方模块[13]。双精度的浮点运算速度最高可达 39.93 亿次每秒，是当时世界上除美日外最快的处理器芯片，我国目前对于浮点运算器的研究也多侧重于高速应用。

龙芯 3A 是国产的商用四核处理器系列，主要面向计算机、低功耗服务器和工业控制等应用场景。2015 年中科院发布基于 GS464E 架构的龙芯 3A2000 芯片，实测工作频率可以达到 1.5GHz，支持单精度和双精度浮点运算。GS464E 架构对于浮点运算的实现是比较全面的，内部流水线融合了多个浮点执行单元，包括比较单元，乘加单元、除法单元和开平方单元，以及格式转换单元[14]。在 SPEC 2000 测试下，3A2000 的浮点性能得分超过 1100 分每 GHz，是国产处理器芯片中单核 SPEC 测试性能比较高的芯片。

另外非商用的浮点运算器也取得了一些成果，例如文献所提出的浮点协处理器 FPVC，支持标量和向量计算[15]。FPVC 最大的不足是仅支持单精度浮点数的运算，而现今对于数据处理的要求越来越高，支持多种精度的浮点运算对于各种应用程序的正确和

快速运行是必不可少的[16]。国内也有研究人员设计了适用于嵌入式领域的浮点运算器，一定程度上改善了面积效率，但仍只支持单精度浮点数[17]。

1.2.2 RISC-V 浮点运算器的现状

上述浮点运算器被应用于多种计算机架构，在很长一段时间里，处理器都是基于非开源架构进行设计的，包括 ARM、x86 等传统架构。由于非开源的特性，高昂的授权费对科研工作造成了一定限制，且这些传统架构在现代处理器的发展过程中不断引入新的需求，存在指令定义复杂和数量冗余等缺点，对新系统的开发产生阻碍，且在一些设计细节上不同研发人员有不同的实现方式，导致兼容性问题。

针对传统架构所存在的问题，加州大学伯克利分校提出了一种全新且完全开源的指令集架构，即 RISC-V 架构。使用 RISC-V 无需支付高额的授权费，这对于构建开源的硬件生态环境是非常有用的。得益于后发优势，RISC-V 在设计之初已考虑了兼容性的问题。IEEE 754 标准是使用最为广泛的浮点标准，为许多处理器与浮点运算器所采用。但此标准为保证通用性，在许多定义上并没有限制唯一值，例如无效信号（Not a Number，NaN）可以有多种编码方式来存放回溯诊断信息，或是对于下溢的判断时间可取舍入前或是舍入后[18]。传统架构由于未采用统一的标准，在硬件设计上可能对于这些定义所采取的实现方案不同，这使得基于传统架构设计的浮点运算器虽不违反协议要求却存在兼容性问题。RISC-V 对这些可选的定义均进行统一的规定，包括舍入和异常处理等问题，这使得基于 RISC-V 所设计的浮点运算单元可以更好地协作和兼容，目前越来越多的硬件产品也倾向于使用 RISC-V 来进行设计[19]。

RISC-V 作为热门的新兴开源架构，已吸引了多家企业参与研究和设计，国内目前比较知名的是芯来科技所推出的蜂鸟 E200 系列处理器，该系列在功耗和面积的表现优于同级的 ARM Cortex-M，主要的应用场景为低功耗的嵌入式领域。其中 E203 处理器采用 RISC-V 的 RV32EMAC 和 RV32IMAC 架构，且完全开源，非常适合进行 RISC-V FPU 的相关研究，例如潘树朋等人就将设计的浮点运算器与 E203 进行了集成[17]。

RISC-V 指令集支持浮点运算，目前常见的 RISC-V 系统大多数集成了浮点运算单元。伯克利分校在发布 RISC-V 的同时推出了开源的 64 位 RISC-V 处理器生成器 Rocket Chip，可用它来生成处理器核 Rocket Core[20, 21]。Rocket Core 采用五级流水线设计，内部集成了浮点运算器，该 FPU 符合 IEEE 754-2008 标准。Rocket Core 只支持标量运算，内部包含格式转换部件、浮点乘加部件以及除法与开平方部件。对于多种精度运算，Rocket Core 采用了多模块的实现方式，乘加模块分为单精度乘加模块和双精度乘加模

块，除法与开平方模块同理，一定程度上造成了面积的浪费，且并不支持半精度运算。

近年来各国相继推出自研的处理器芯片，作为印度的顶级高校，印度理工学院于2016年启动处理器Shakti的研发工作[22]。Shakti基于RISC-V架构，并针对不同市场规划了多个系列，包括E-Class、C-Class、I-Class、M-Class、S-Class、H-Class、T-Class和N-Class，它们在面积、性能、功耗方面各有优点，在商用处理器领域具有竞争力。其中C-Class可被配置为32或64位标量处理器，支持RISC-V的C扩展，内部包含一个浮点运算器，只支持标量运算。此FPU的设计思想和Rocket Core类似，采用多模块的方式实现多种精度运算，其中包含单精度和双精度运算模块，面积仍存在优化空间。

苏黎世瑞士联邦理工学院则开源了基于RISC-V的SOC平台pulp-platform，主要面向低功耗的可穿戴设备和物联网应用，包括多核计算系统PULP，以及更通用的单核版本PULPino[23, 24, 25]。为满足pulp-platform浮点运算的需要，有多款FPU可供选择，其中包括已流片验证的ETHZfpu[26]。该FPU包括浮点加、乘、乘加、除法与开平方，以及格式转换等模块，仅能实现单精度的标量运算。另外有可满足向量运算需要的FPnew，FPnew为可实现向量运算的多种精度可配置浮点运算器，可选择实现8、16、32和64比特的浮点数运算，且流水线深度可配以更好地平衡速度和寄存器资源的使用[27]。FPnew通过parallel和merge两种方式实现向量化，图1-3为半精度FMA slice，展示了向量化的parallel实现，可执行半精度向量FMA运算，在这种实现方式下每个精度对应一个slice。完成四个半精度的乘加运算，该slice中需要四个并行的FMA模块，这也是目前硬件设计中最常用的向量化实现方法，非常消耗面积。

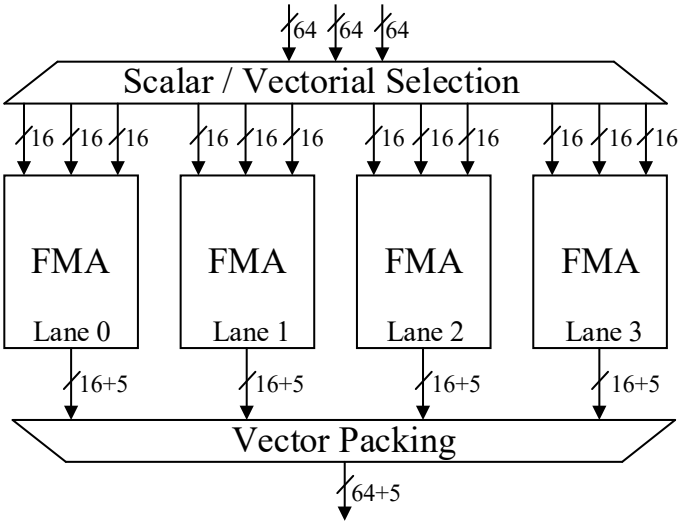

图1-3 FPnew向量运算的paraller实现[27]

图1-4展示了向量化的merge实现方式，这种实现方法不像parallel需要每个精度

都独立使用一套 slice，而是所有精度共用一套 slice，slice 内不同位宽的运算共享一套硬件资源，且运算模块的位宽逐个递减。fp64、fp32、fp16 和 fp8 分别表示 64、32、16 和 8 比特的浮点数，以图 1-4 为例，此时 Lane 0 可完成一个 fp64、fp32、fp16 或 fp8 运算，Lane 1 可完成一个 fp32、fp16 或 fp8 运算，Lane 2 和 Lane 3 可完成一个 fp16 或 fp8 运算，Lane 4 到 Lane7 则只执行 fp8 运算。假设面积的增长和实现精度的总位宽成正比例变化，实现与图 1-4 相同的功能，parallel 方式所需的面积是 merge 的 1.6 倍，可见 FPnew 的这种 merge 实现方式对于面积效率是有一定优化效果的，但仍无法实现只使用一套运算逻辑来完成所有精度的向量运算。

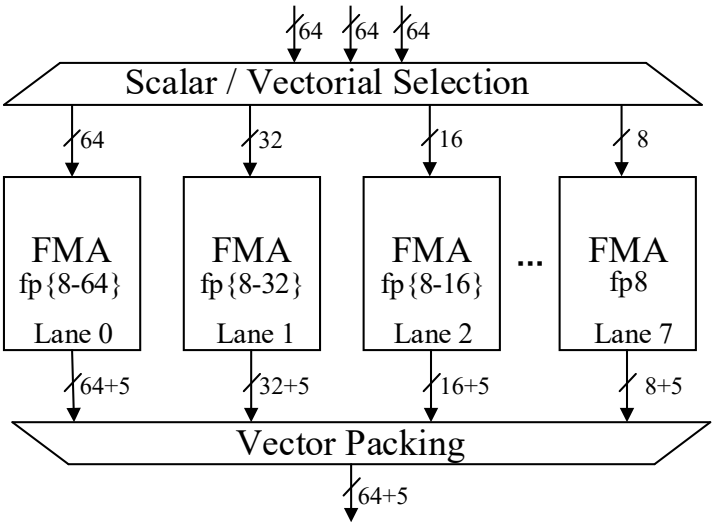

图 1-4 FPnew 向量运算的 merge 实现[27]

“香山”处理器是国内优秀的 RISC-V 超标量乱序处理器，内部浮点单元包括 4 个 FMAC 和 2 个 FMISC，前者用于执行乘加功能，后者完成其他浮点操作，例如除法和格式转换等[28]。“香山”通过舍弃一部分面积来保证浮点向量运算的执行速度，仍存在一定的优化空间。但不可否认，它是国内对于 RISC-V 架构的处理器一个很好的探索，具有很大的发展潜力。另外一些设计则是通过“批处理”的方式实现向量化，即每次只执行一个浮点运算，直至所有操作数循环执行完毕后再一起输出，这种方法使得指令的执行周期翻倍增长，也称为硬件迭代法[29, 30]。

有研究人员则将注意力集中到浮点算法的优化上，特别是对于延时的优化，例如对于浮点加法和乘加，在尾数加法时并行执行前导 0 预测（Leading Zero Anticipator，LZA）相比于传统的前导 0 检测（Leading Zero Detector，LZD）可以一定程度上缩短时序路径，但也可能带来一个比特的误差[31, 32]。另外对于浮点乘加单元常采用双通路的设计方法，在硬件上根据指数差划分 close 和 far 路径以缩短关键路径是常用的优化思路，但这种方

法通常会导致大面积的问题。而除法和开平方模块的设计则更多的集中于对尾数运算的处理算法上，包括恢复与不恢复余数算法、Newton-Raphson 算法等，这些算法各有优缺点。例如 Miriam 和 Peter 等人在研究中指出 Newton-Raphson 算法和 Goldschmidt 算法在实现上可以复用浮点乘法器的尾数乘法模块，但由于需要增加新的控制逻辑，容易引起时序问题而导致性能瓶颈[33]。也有研究人员在除法中引入 SRT 算法以实现更快收敛速度的浮点除法器，SRT 算法本质上是不恢复余数算法的一种变形[34]。SRT 算法优化了不恢复算法迭代次数的问题，随着基数的提高，每次迭代所得到的商的位数增大，但相应的硬件资源消耗也增大[35]。一般来说，除法和开平方等运算器所消耗的时钟周期和面积的变化趋势是相反的，在设计上对算法的选择应根据具体的设计目标而定。

1.2.3 存在的问题

综上目前 RISC-V 浮点运算器的研究仍存在一些问题，主要包括以下几个方面：

（1）支持精度较少，支持多种精度需要消耗更大的面积。现有的浮点运算器支持精度少，多只支持一或两种精度的浮点数运算，少数已商业化的产品可以支持多种精度运算但并不开源，难以提供参考。为实现多种精度，很多设计采用多模块的实现方式，造成面积浪费。

（2）侧重速度的提升而导致大面积的问题。现阶段对于大规模浮点运算芯片的研究较多，为了实现高速，基于面积换时间的思想，很多都集成了大量浮点运算单元，或是采用多通路的实现方法，导致更大的硬件开销。也有的研究从算法层面入手，但通常来说，快速算法的引入会增加面积。浮点运算器多采用流水线设计，流水线深度增大可提高工作频率，但也增加了寄存器资源的使用，进而影响面积。

（3）向量化实现困难，难以实现高面积效率。向量化是浮点运算器发展的必然趋势，现阶段部分研究仍只支持标量运算，而现有的向量浮点单元也多难以兼顾速度和面积，支持多种浮点精度也使得向量化实现更加困难。很多运算器通过例化多个运算单元来实现多种精度的向量运算，导致面积增加。而部分设计则舍弃速度，采用“批处理”的方式迭代更多的周期数，面积减小了但吞吐率大大降低。

本课题将着重对以上问题进行研究，实现一个适用于 RISC-V 的高面积效率浮点运算器，具备研究和应用价值。

1.3 主要研究内容

本课题基于 RISC-V 协议，对其中的浮点运算操作进行实现，在 IEEE 754-2019 标

准下实现高面积效率的浮点运算器。其可实现 RISC-V 的双精度和单精度浮点运算，并扩展实现半精度浮点运算以满足低精度运算的需要，且支持浮点向量运算。其可执行浮点加、减、乘、乘加、乘减、除和开平方运算，以及浮点数和无符号整数、浮点数和有符号整数的互相转换，并支持 RISC-V 规定的五种舍入模式和五种异常处理。

本文将综合考虑面积和速度的需求，使用流水线技术以提高设计的最高工作频率，细致分析浮点运算算法，在保证一定速度的前提下合理设计浮点运算器的硬件架构以减小整体面积，提高面积效率。面积效率即性能/面积，高面积效率意味着更低的成本，本文以速度与面积的比值进行评估。并考虑浮点运算的向量实现，所设计的浮点运算器可并行执行四个半精度浮点操作，或并行执行两个单精度浮点操作，或执行一个双精度浮点操作。本课题将对模块的硬件架构进行深度分析，对运算模块进行 SIMD 优化，使运算模块只需使用一套硬件资源便可实现多种精度的向量运算，以减少运算器的面积，实现高的面积效率。

由于开源 RISC-V 处理器多使用 Chisel 语言进行硬件开发，为更好地与 RISC-V 设计兼容和协作，本课题也将采用 Chisel 进行设计。设计完成后采用 Synopsys 公司的 VCS 和 Verdi 工具进行功能验证，以确保所设计的浮点运算器功能正确。之后利用 Design Compiler（DC）工具进行综合，确保时序收敛，得到最高工作频率和面积数据，从而对面积效率进行合理的评估，并与其他知名的 RISC-V 浮点运算器进行对比。最后将利用 Formality 工具完成形式验证以提高可靠性，并将开发代码下载到 FPGA 板上完成板级验证。

1.4 论文结构安排

本文工作分六个部分进行阐述，下面简单地对各个章节的内容进行介绍:

第一章为绪论，主要讲述课题背景和浮点运算器的研究现状，介绍国内外已有的一些浮点运算器，并指出现有研究的不足之处，以阐明课题的研究意义，说明本课题具备一定的研究价值，并对本文各个章节的主要内容进行概括。

第二章为浮点协议标准，本章主要介绍进行浮点运算器设计时所必须遵循的标准和协议。首先对 IEEE 754 标准进行简述，并介绍了浮点数的表示格式和数值分类，后半部分则介绍了 RISC-V 协议所规定的浮点运算的舍入模式和异常处理。适用于 RISC-V 的浮点运算器必须遵循这些标准和协议进行设计。

第三章为浮点运算算法研究，对 RISC-V 中的浮点运算操作进行介绍。主要介绍浮

点加减法、浮点乘法、浮点乘加、浮点除法和浮点开平方以及浮点数和整数互相转换的基本算法流程，并对一些可能被应用于硬件设计的算法模型进行阐述和对比，例如前导 0 预测算法、Booth 乘法等等。本课题的硬件设计将基于这些算法，本章为后文介绍具体的硬件设计奠定基础。

第四章为浮点运算器的设计与优化，主要对本课题的硬件设计部分进行阐述。对 RISC-V 中的浮点运算操作进行硬件实现，首先对浮点运算器的整体架构进行介绍，之后分小节对各个运算模块的设计思路进行阐述，包括浮点加减法与格式转换模块、浮点乘加模块，以及浮点除法与开平方模块。着重对区别于过往研究的优化部分进行介绍，以体现本文工作的创新性，最终完成适用于 RISC-V 的高面积效率浮点运算器设计。

第五章为系统仿真与验证，包括功能验证、逻辑综合和形式验证以及 FPGA 的板级验证，保证浮点运算功能正确，并得到硬件设计的最高工作频率和面积数据。将数据与现有的 RISC-V 浮点运算器进行对比，以评估本设计在相关数据上是否有所优化。

最后为总结与展望，本章起总结全文的作用，阐述所设计的浮点运算器的优点以体现创新性，另一方面则指出仍存在的不足与可改进之处。

第二章 浮点协议标准

本章主要介绍在设计适用于 RISC-V 的浮点运算器时必须遵循的标准和规则。首先介绍 IEEE 754 浮点标准，这是被浮点系统最广泛使用的协议，定义了浮点数的表示格式和数值分类。之后介绍 RISC-V 协议所规定的浮点数舍入规则和异常处理规则，这些规则符合 IEEE 754 标准，且进行了更加具体的规定，适用于 RISC-V 的浮点运算器应遵循这些规定。

2.1 IEEE 754 标准概述

IEEE 754 即 IEEE 浮点数算术标准，于 1985 年为二进制浮点系统开发，无论是基于 ARM 架构还是 RISC-V 架构，浮点处理单元均需要依照此标准进行实现。该标准后续也发布了多个修订版本，本课题实现的浮点运算器依据最新的 IEEE 754-2019 版进行开发[18]。

此标准的目的是提供一种与实现方式无关的浮点数运算方法，可以由硬件实现、软件实现或者软硬件共同实现。在不同的实现中，给定相同的输入数据，计算的结果以及计算中报告的错误和错误条件是相同的。

该标准规定了基本和扩展的浮点数格式、浮点算术运算、整数和浮点数格式之间的相互转换和舍入算法，以及浮点异常控制，并指定了这些情况的处理方法。IEEE 规定的浮点数格式包含有限数，由符号位、尾数位和阶码位三个字段组成，其中又分为规格化数和非规格化数，另外还规定了两个无穷大值“+∞”和“−∞”以及非数 NaN，NaN 表示未被定义或无法表示的数值。

2.2 浮点数表示格式

IEEE 754 浮点标准由公式(2-1)的形式来表示一个浮点数。

$$M = (-1)^{s} \times 2^{e-bias} \times (1.f) \tag{2-1}$$

浮点数 M 的各个字母的含义如下：

（1）s：表示浮点数的符号，决定该浮点数是负数还是正数，与整数的表示相同，s 为 1 表示负数，反之为 0 表示正数。

（2）f：表示浮点数的尾数，以正的二进制小数进行存储，对于 n 位的尾数 f 存储为 $f_{n-1}f_{n-1}....f_1f_0$。在浮点系统中，尾数是一个定点值，实际数值为 $x.f$。对于规格化数，x 一定为 1，由公式(2-1)所示，而非规格化数的 x 一定为 0，因此 IEEE 754 标准省去了 x

的存储以减小位宽，*x* 也称为隐藏位，规格化数和非规格化数的含义在后文进行介绍。

（3）*e*：表示浮点数的阶码，作用是对浮点数进行加权，权重是 2 的 *e-bias*（可能是负数）次幂，*bias* 为偏置值。假设阶码的宽度为 *n*，则偏置值 *bias* 为 2^{n-1}-1。由于 *bias* 在单一精度浮点数中是一个固定值，因此 IEEE 754 规定只需要存储 *e* 的值即可，*bias* 的存在使得 *M* 可以去表示数值范围更大的小数。对于半精度浮点数，*bias* 的值是 15，对于单精度是 127, 双精度是 1023。随着精度的提高，浮点数能表示的数值范围也更大。

半精度、单精度和双精度浮点数在计算机系统中分别存储为 64、32 和 16 比特的二进制小数，上述各个字段的存储位置和位宽在 IEEE 754 标准中的存储如图 2-1 所示，由上至下分别为半精度浮点数（fp16）、单精度浮点数（fp32）和双精度浮点数（fp64）。

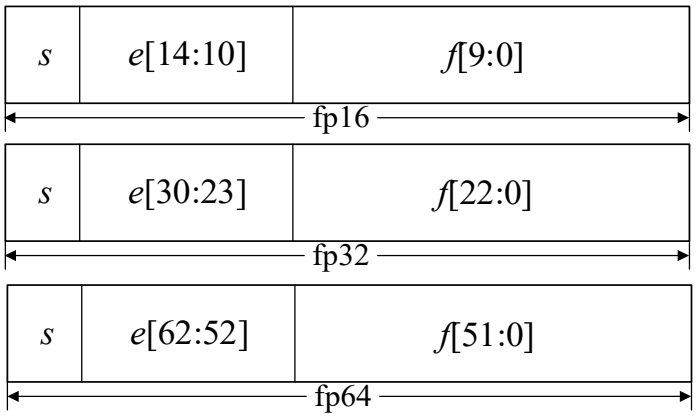

图 2-1 浮点数表示格式

2.3 浮点数数值分类

在 IEEE 754 标准中规定，如果一个浮点数的阶码 *e* 满足 0<*e*<2^n-1，其中 *n* 为阶码的位宽，那么称其为规格化数，其表示的数值大小 *M* 见式(2-1)。规格化数尾数的隐藏位一定为 1，这个位并不需要使用 *f* 进行存储。

单精度浮点数的阶码 *e* 是连续的 8 比特二进制数，那么指数值 *e-bias* 的范围在-126 至+127。根据以上定义可知，对于单精度浮点数，最小的规格化数二进制表示为 s_00000001_00000000000000000000000，实际表示的绝对值大小为 $2^{-126}\times1.0$。如果需要表示一个比这个数值还要小的非 0 值，则可以使用非规格化数。对于非规格化数，其阶码 *e* 存储为 0，表示的数值大小 *M* 见式(2-2)：

$$M = (-1)^s \times 2^{1-bias} \times (0.f) \tag{2-2}$$

非规格化数虽然 *e* 取为 0，但指数值取为 1-*bias* 而不是-*bias*，这使得非规格化数与规格化数的过渡更加平滑。当阶码位 *e* 和尾数位 *f* 均为 0 时，该浮点数被定义为 0。

另外 IEEE 754 还规定了两种特殊的浮点数，即无穷大（INF，也可表示为∞）和非数（NaN）。当 *e* 的每一比特均为 1 时，如果此时 *f* 为 0 时，它表示无穷大，并根据符号位 *s* 可分为正无穷大（+∞）和负无穷大（-∞）；而如果此时 *f* 不为 0，则表示该浮点数不是一个有效的浮点数，称为 NaN，该数值可作为一些不合规范的浮点操作的结果，例如+∞+(-∞)，或是∞×0，均为无效操作，结果输出 NaN。

2.4 基于 RISC-V 的浮点数舍入规则

经过运算的浮点数可能会超出位宽的限制，此时需要把多余的位舍弃，称为浮点数舍入。除非另有说明，否则在执行浮点操作时，都应先产生一个精确到无限精度和无限范围的中间结果，然后根据选择的舍入方向去舍入该结果。基于 IEEE 754，RISC-V 浮点指令集规定了多种舍入模式如表 2-1 所示，“—”表示无具体规定。

表 2-1 RISC-V 的舍入模式编码[19]

舍入模式	助记符	含义
000	RNZ	向最接近的值舍入，首选“偶数”值
001	RTZ	向零舍入
010	RDN	向下含入（向-∞）
011	RUP	向上含入（向+∞）
100	RMM	向最接近的值合入，首选最大值
101	—	非法，保留给未来使用
110	—	非法，保留给未来使用
111	—	选择动态舍入模式寄存器，非法

舍入模式由 3 比特的输入信号进行控制，RISC-V 目前共规定了五种可选的舍入模式，本课题将实现这五种舍入模式，分别包括以下几种：

（1）向偶数舍入的就近舍入（RNZ）：舍入到最接近无限精度结果的浮点数，如果存在两个最接近的浮点数，则舍入到最低有效数字为偶数的那一个。假设有一个 8 比特的尾数为 00100100，如果要舍入为 5 比特的有效尾数结果，由于多余的 3 比特值 100 正好为 3 比特数所能表示的所有数的中间值，因此其有两种舍入的可能，舍入后可能为 00100 或者 00101，如果选择该舍入模式，最终结果将为 00100。这额外的 3 比特称为附

加位，从高位到低位分别称为 *G*（*guard*）、*R*（*round*）和 *S*（*sticky*），在浮点运算流程中将附加位以变量 *GRS* 进行存储并放于有效尾数的最后，直至规格化后进行舍入。

（2）向 0 舍入（RTZ）：朝向零点方向进行舍入，实际上就是丢弃 *GRS* 而不向有效尾数进行加 1 操作。

（3）向下舍入（RDN）：朝向负无穷方向舍入，如果结果为负数且 *GRS* 不为 0，那么对有效尾数进行加 1 操作。

（4）向上舍入（RUP）：朝向正无穷方向舍入，类似向下舍入，如果结果为正数且 *GRS* 不为 0，那么对有效尾数进行加 1 操作。

（5）向更大量级的就近舍入（RMM）：舍入到最接近无限精度结果的浮点数，区别于向偶数舍入，如果存在两个最接近的浮点数，则舍入到绝对值更大的那个数，对于 8 比特的尾数 00100100，最终会舍入到 00101。

2.5 基于 RISC-V 的浮点异常生成

基于 IEEE 754 标准，RISC-V 对浮点操作定义了五种异常信号并进行了更加详细的规定，当出现了异常情况时，硬件需要做相应的处理，并拉高正确的异常指示信号，其共有 5 比特，分别对应以下五种异常情况[19]。

（1）无效操作异常

当且仅当执行的浮点操作无法得到有效定义的结果时输出 NaN，并根据需要生成无效操作异常指示信号。

在浮点系统中，NaN 分为 qNaN 和 sNaN，两者的阶码每一比特均为 1，IEEE 754 规定 qNaN 的尾数最高位为 1，而 sNaN 的尾数最高位为 0，且其他位至少有一个位为 1 以区别于无穷大。RISC-V 在 IEEE 754 的基础上规定任何需要生成 NaN 的操作，结果均输出 qNaN，且输出的符号位一定为 0，尾数除最高位为 1，其他位均为 0。这种统一方便了不同 RISC-V 系统的交互，RISC-V 组织者也曾考虑过传递 NaN 有效载荷（即不统一尾数信息），但这会增加硬件的代价，且不方便移植，因此最终选择生成统一的 NaN 信号。

并不是所有会输出 NaN 的操作都会将指示无效操作异常的信号置位至高电平。如果所执行的指令有一个源浮点数是 sNaN，则会产生无效操作异常指示，否则在正常的运算操作下，如果源浮点数是 qNaN，则输出 qNaN，且不产生无效操作异常指示。

另外对于源浮点数不包括 sNaN，但无法得到确定结果的操作，也会产生无效操作

异常指示，所有情况汇总于表 2-2。

表 2-2 源浮点数非 sNaN 的无效操作

浮点运算	源浮点数非 sNaN 的无效操作
加法、减法、乘加、乘减	对 INF 进行绝对值相减
乘法、乘加、乘减	执行 INF 与 0 相乘
除法	执行 INF 和 INF 相除或 0 和 0 相除
开平方	被开平方浮点数小于 0
浮点数转整数	当浮点数是 qNaN、INF 或舍入后超出目标格式表示范围的有穷浮点数

（2）除 0 异常

在浮点除法中，当除数为零且被除数为有限的非零数时，除零的默认结果应该为无穷大，且无穷大的符号由被除数和除数的符号位异或得到。

（3）上溢异常

当浮点操作得到的结果舍入后的绝对值超出了目标格式所能表示的最大有穷数时，浮点运算器应发出上溢异常信号。其中上溢只发生在结果目标格式为浮点数的操作，因此浮点数向整数的转换并不会产生上溢异常。当上溢异常发生时，运算器应根据所选择的不同舍入模式来输出规定的值。

当舍入模式为就近舍入，无论是向偶数舍入，还是向更大量级舍入，当上溢异常发生时，运算器均应输出无穷大，符号取决于此刻执行的浮点操作种类。

当舍入模式为向 0 舍入，当上溢发生时，输出目标浮点格式所能表示的最大有穷数。以单精度为例，即阶码最低位为 0，而其他位均为 1，尾数则所有比特值均为 1，符号取决于此刻执行的浮点操作种类。

当舍入模式为向上舍入，如果发生上溢异常的浮点结果为正数，则输出正无穷大，反之如果为负数，则输出最大负有穷数。

当舍入模式为向下舍入，与向上舍入相反，如果发生上溢异常的浮点结果为负数，则输出负无穷大，反之如果为正数，则输出最大正有穷数。

（4）下溢异常

当浮点操作得到的结果数值过小以至于不能维持精度而导致精度受损时，需要按规

定产生下溢异常。这一般发生在结果小于规格化数所能表示的最小值时，或者说生成非规格化数时，但并不是所有生成非规格化数的操作都会产生下溢异常。

IEEE 754 标准规定对于下溢异常判断的时间点可以是舍入之前或者舍入之后，基于 RISC-V 设计的浮点运算器采用在舍入之后进行下溢异常判断，这是由于在舍入之后再检测极小可以减少虚假的下溢信号。

只有在舍入的结果是不精确的值时，才会真正地设置下溢标志，即 *GRS* 要求非 0。如果舍入的结果是精确的，即使此时生成了非规格化数，仍不会产生下溢异常，这是 RISC-V 异常处理中唯一不会设置相应状态标志的情况。实际上，下溢发生时，运算器仍会正常输出运算得到的值，只是这个数值相比于规格化数更小。尽管此时精度有一定程度的受损，但大部分情况并不会影响运算器的正常工作，RISC-V 这样规定可避免一些不必要的异常生成。

（5）不精确异常

当舍入之后的结果与舍入之前不同，即舍入之前如果 *GRS* 有任一比特不是 0 时，运算器会发出不精确异常信号。另外当上溢发生时，也会发出不精确异常信号，因为此时输出的都是非精确值。而对于下溢操作，由于只有当结果不精确时才会将下溢异常信号拉高，因此当发出下溢异常时，运算器应同步拉高不精确异常信号。

2.6 本章小结

本章介绍了本课题在设计过程所必须遵循的标准和规则，浮点系统应遵循 IEEE 754 标准进行设计，且由于本文实现的浮点运算器适用于 RISC-V，因此还必须遵循 RISC-V 的浮点协议，以进行正确的舍入和异常处理。RISC-V 对 IEEE 754 协议中可选的定义均进行了更加精确的规定，便利了硬件系统的设计和研究。

第三章 浮点运算算法研究

本章对本课题实现的 RISC-V 中的浮点运算种类和运算基本流程进行阐述，并对其中关键模块的实现算法进行介绍，了解和选择符合设计目标的算法对于硬件设计是至关重要的。浮点数由符号 s、阶码 e 和尾数 f 组成，为方便叙述本章以 $\{s,e,f\}$ 来表示浮点数。

3.1 浮点加减算法

3.1.1 浮点加减算法流程

浮点数的加法和减法运算是浮点系统中使用频率较高的操作，在浮点运算部件中，经统计浮点加法和浮点减法的使用次数接近所有浮点操作指令总量的二分之一[9]。当运算器执行 $opa+opb$ 时，操作符保持为“+”，执行加法还是减法由浮点数 opa 和 opb 的符号位控制。当符号位相同时执行浮点加法，当符号位相反则执行浮点减法，加法与减法的算法流程均可由图 3-1 表示，统称为浮点加减法。浮点加减法功能模块的性能好坏将直接影响 FPU 的性能进而影响处理器的浮点运算能力。相比于定点数的加减法，由于阶码的引入，浮点加减法要复杂得多，增加了硬件设计的困难。

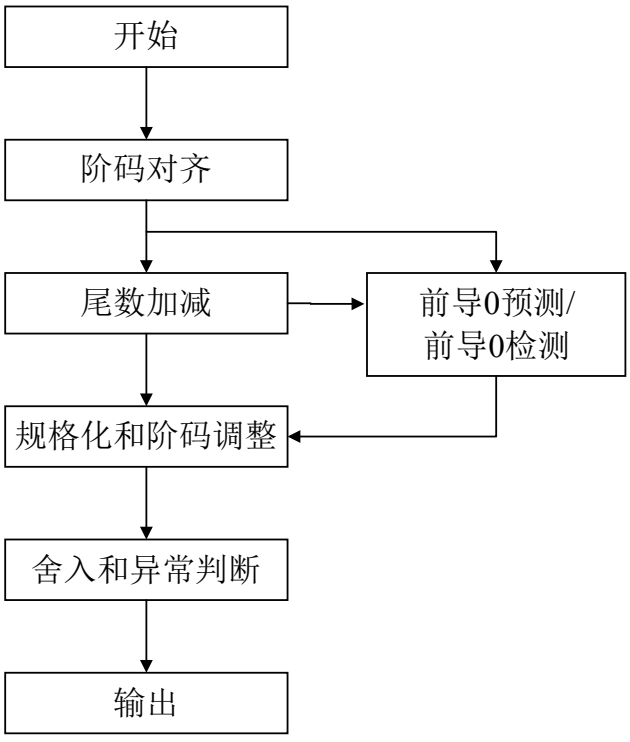

图 3-1 浮点加减法流程图

假设 $opa=\{s_a,e_a,f_a\}$，$opb=\{s_b,e_b,f_b\}$，浮点加减算法 $opa+opb$ 主要包括以下流程：

（1）阶码对齐：将浮点数的小数点位置对齐，统一临时阶码 e_{tmp}，也称为对阶。需

要先判断出浮点数绝对值的大小关系，如果 $opa \geq opb$，那么 e_{tmp} 统一为 e_a，并将 f_b 向右移位，等效于将 f_b 的小数点向左移位；反之 e_{tmp} 取 e_b，并将 f_a 向右移位。

（2）尾数加减：执行定点加减法，用绝对值更大的浮点数的尾数去与对阶好的尾数进行相加减。

（3）前导 0 预测、检测：计算尾数加减后的结果第一个 1 的位置，得到在这个 1 前 0 的数量，这些 0 称为前导 0，而第一个 1 称为前导 1。预测和检测的区别在于前者可与尾数加减并行执行，为并行模块，后者必须等尾数加减运算完成后才可执行，为串行模块。该流程是影响浮点加减法运算速度的重点。

（4）规格化和阶码调整：尾数加减后的结果可能不满足浮点数尾数的表示规则，应对尾数进行规格化操作。如果结果判断为规格化数，尾数应处理为 $1.f$ 的形式，利用（3）求得的前导 0 数量将尾数左移至最高位为 1，且尾数每左移一位，阶码应减一。如果结果判断为非规格化数，也需将尾数左移，直至阶码为 0，完成移位的尾数最高位不一定为 1。如果整数部分大于 1，那么应将尾数右移一位，这种情况下只需通过位选择实现硬连接，无需使用移位器，相应地阶码加一。

（5）舍入和异常判断：RISC-V 对于浮点数的舍入规定了五种舍入方法，包括就近舍入（向偶数舍入和向更大量级舍入）、向 0 舍入、向下舍入和向上舍入，本课题实现的浮点运算器支持这五种舍入模式。在完成舍入后，根据 RISC-V 的要求进行异常判断以生成 5 比特的异常指示信号，并输出运算结果。

基于 RISC-V 实现的浮点加减法运算一定不会产生下溢异常。这是由于要产生下溢异常需要满足两个条件，一是舍入后的结果处于非规格化数所规定的区间，二是该结果是不精确的，即 *GRS* 不为 0。对于非规格化数相加减，结果的 *GRS* 必定为 0，不满足下溢的产生条件，尽管结果为非规格化数。而对于规格化数与规格化数相加减，或是规格化数与非规格化数相加减，结果舍入后必定为规格化数，并不会产生下溢的情况。

可见浮点加减法的流程相比定点加减法复杂得多，这很大程度是由于阶码的存在，但也恰恰是阶码使浮点数相比定点数所能表示的数值范围大大增加。在整个算法流程中，移位器和尾数加减法器的使用频率是最高的，若要实现更小的面积，复用这类逻辑器件是整个设计的关键，且应保证一定的速度以提高运算的吞吐率。

3.1.2 前导 0 预测

前导 0 预测是对两个定点数相减后的结果进行预测，预测的目标是从左边的最高位开始往右边数，预测当第一个 1 出现的时候，在 1 前面的 0 的个数。在常规流程中，需

要在尾数加减完成后才可进行第一个 1 的查找，这个查找的过程称为前导 0 检测，也称 LZD。LZD 延时是比较大的，如果和大延时的尾数加减模块串行，那么整个算法的速度将大大降低。

为减小LZD的延时所带来的影响，前导0预测是可行的方案，前导0预测也称LZA。LZA 由前导 0 编码和 LZD 组成，LZA 会将加减法前的数据输入到编码电路中得到新的序列，之后对这个编码序列进行 LZD。由于编码序列的生成比加减法要快得多，从而将 LZD 的开始时间提前从而减小整体延时。LZA 尽管增加了一部分面积，但对速度的优化是非常可观的，本课题将这种方法引入到浮点加减模块的设计中。尾数加法和减法的 LZA 编码逻辑有所不同，相比加法，减法的 LZA 更为复杂。

（1）尾数减法 LZA

假设有两个位宽为 n 的整数 a 和 b，分别编码为 $a_{n-1}a_{n-2}a_{n-3}\ldots a_1a_0$ 和 $b_{n-1}b_{n-2}b_{n-3}\ldots b_1b_0$。以 $r=r_{i-1}r_{i-2}r_{i-3}\ldots r_1r_0$ 表示 a-b 的结果，如果不考虑借位，r 的每个比特有三种可能的情况，即 0、1 和-1。另外定义 z_i、p_i 和 n_i，r_i 为 0、1 和-1 分别对应 z_i、p_i 和 n_i 为 1，可由公式(3-1)、公式(3-2)和公式(3-3)求得：

$$p_i = a_i\overline{b_i} \tag{3-1}$$

$$z_i = a_ib_i + \overline{a_i}\,\overline{b_i} \tag{3-2}$$

$$n_i = \overline{a_i}b_i \tag{3-3}$$

考虑三种情况，分别为 $a>b$，$a<b$ 和 $a=b$。首先考虑 $a>b$，此时减法结果一定大于 0，所以序列 r 中第一个不为 0 的值一定是 1，r 的表示见式(3-4)：

$$r = 0^k1x \tag{3-4}$$

其中 0^k 表示连续 k 比特是 0，x 表示任意的序列。如果此时 r 为 0^k11x，那么即使 $x<0$，也无法影响到第一个 1 的位置，设 r 的前导 0 数量为 lzd_r，此时 lzd_r 取 k。为了确定 k 的值，只需要从最高位到最低位去寻找 r 中的第一个子序列 2'b11 的位置即可得到结果。

如果 r 为 0^k10x，那么存在两种情况，如果此时 $x<0$，则最后将 r 化为标准的二进制表示时，x 一定需要向前面进行借位，此时 lzd_r 取 $k+1$。而如果 $x\geq0$，则将 r 化为标准的二进制时，x 一定不需要向前面借位，此时不会影响到 1 的位置，所以 lzd_r 取 k。对于这种可能有两种预测结果的情况，LZA 的处理方式是将两种情况统一起来，均预测 lzd_r 为 k，而在规格化时再判断是否需要再左移一个比特以进行预测补偿。因此对于 r 为 0^k10x 的情况，按照从最高位到最低位的方向，寻找 r 中第一个子序列 2'b10 的位置即可得到

预测结果。

对于最后一种情况 r 为 $0^k1(-1)^j(0/1)x$，即第一个 1 后面跟着 j 个-1，0/1 表示可能为 0 或 1，那么此时将 r 的已知部分转化为二进制表示时，r 必然为 $0^{k+j}1(0/1)x$。若 0/1 取 1，则 x 的值并不会影响 lzd_r，lzd_r 取 $k+j$；若 0/1 取 0，那么当 $x<0$ 时，lzd_r 取 $k+j+1$，反之当 $x\geq 0$，那么 lzd_r 取 $k+j$，与 r 为 0^k10x 的情况是类似的，此时统一预测 lzd_r 为 $k+j$。为了确定 k 的值，寻找 r 中第一个子序列 2'b(-1)0 或 2'b(-1)1 即可得到预测结果。将上述内容以表 3-1 进行总结：

表 3-1 浮点减法 $a>b$ 情况下的前导 0 预测表

序列 r 的值	前导 0 预测个数	特征子序列
0^k11x	k	2'b11
$0^k10x, x\geq 0$	k	2'b10
$0^k10x, x<0$	k（实际为 $k+1$）	2'b10
$0^k1(-1)^j1x$	$k+j$	2'b(-1)1
$0^k1(-1)^j0x, x\geq 0$	$k+j$	2'b(-1)0
$0^k1(-1)^j0x, x<0$	$k+j$（实际为 $k+j+1$）	2'b(-1)0

对 r 进行检测，只要找到表 3-1 中任意一个特征子序列即可对前导 0 的个数进行预测，基于以上逻辑对输入序列 a 和 b 进行 LZA 编码，生成序列 e，e 的二进制表达形式为 $e_{n-1}e_{n-2}e_{n-3}\ldots e_1e_0$，当 r 的 i 和 i-1 位符合任意一个特征子序列时，e_i 为 1，由此 e_i 可由公式(3-5)求得：

$$e_i = p_i p_{i-1} + p_i z_{i-1} + n_i p_{i-1} + n_i z_{i-1} = (p_i + n_i)(p_{i-1} + z_{i-1}) \tag{3-5}$$

z_i、p_i 和 n_i 的计算见式(3-1)、式(3-2)和式(3-3)，代入公式(3-5)可得 e_i 的编码逻辑，由公式(3-6)表示：

$$e_i = \begin{cases} (a_i \oplus b_i)(a_{i-1} + \overline{b_{i-1}}), & i \neq 0 \\ a_i \oplus b_i, & i = 0 \end{cases} \tag{3-6}$$

只需要从序列 e 中按照高位至低位的方向，找出第一个 1 出现的位置即可，即对 e 进行 LZD，由于 e 的生成比 a-b 提早很多，从而实现提前预测 a-b 前导 0 的数量，提高整个设计的速度。

而对于 $a<b$ 的情况，此时减法结果小于 0，序列 r 中第一个不为 0 的值一定是-1，r

的表示见式(3-7)：

$$r = 0^k(-1)x \tag{3-7}$$

为方便讨论，对 r 按位取负值得到 d，即如果某比特为 1，将其取负得到-1，如果某比特为-1，则取负得到 1，0 保持不变，d 实际上为 b-a 的值，那么 d 可由公式(3-4)表示。

在浮点减法中，本课题的尾数加减法模块硬件设计上固定使用大的尾数去减小的尾数，因此结果一定是正数，a 和 b 的大小关系只会影响最终输出的浮点数加减法结果的符号位，实际上是对 d 进行 LZD，此时对于 d 前导 0 的个数的预测情况分析和（1）的情况基本相同，总结后可得到表 3-2：

表 3-2 浮点减法 $a<b$ 情况下的前导 0 预测表

序列 r 的值	序列 d 的值	前导 0 预测个数	特征子序列
0^k(-1)(-1)x	0^k11x	k	2'b(-1)(-1)
0^k(-1)0x, $x\leq 0$	0^k10x, $x\geq 0$	k	2'b(-1)0
0^k(-1)0x, $x>0$	0^k10x, $x<0$	k（实际为 k+1）	2'b(-1)0
0^k(-1)1^j(-1)x	$0^k1(-1)^j1x$	k+j	2'b1(-1)
0^k(-1)1^j0x, $x\leq 0$	$0^k1(-1)^j0x$, $x\geq 0$	k+j	2'b10
0^k(-1)1^j0x, $x>0$	$0^k1(-1)^j0x$, $x<0$	k+j（实际为 k+j+1）	2'b10

类似地，总结上表可得到当 i 非 0 时 e_i 的编码逻辑由公式(3-8)表示：

$$e_i = (a_i \oplus b_i)(\overline{a_{i-1}} + b_{i-1}) \tag{3-8}$$

对于 a=b 的情况，此时 a-b 的前导 0 数量为 a 或 b 的位宽，由于 a-b 结果为 0，实际在后续规格化和舍入过程中，无论移位多少比特结果均为 0，前导 0 的数量不会对最终结果产生影响，因此无需对此情况进行判断，从而可省去部分判断逻辑以减小面积。

（2）尾数加法 LZA

对于 a+b，如果是规格化数和规格化数相加，或者规格化数和非规格化数相加，由于相加的结果一定为规格化数，由于规格化数的隐藏位为 1，那么前导 0 的个数一定为 0，无需进行预测。

对于非规格化数之间的相加，由于非规格化数的阶码为 0，尾数部分为 0.f，且无需对阶移位，那么两个尾数相加后的结果一定不超过 2，即结果只有 1.f 和 0.f 两种情况。1.f 为规格化数，无需进行左移；而 0.f 仍为非规格化数，且此时阶码为 0，也无需左移，

两种情况均将前导 0 的数量视为 0 即可。综上所述，可直接简化 *a*+*b* 的 LZA 逻辑以减小不必要的面积开销。

3.1.3 前导 0 检测

前导 0 预测编码得到的序列需经过前导 0 检测才能得到最终的值，本小节对两种前导 0 检测的算法进行介绍，一种是直接由输入序列得到前导 0 数量的选择逻辑，另外一种则是在得到前导 0 数量的同时完成左移操作，直至将第一个 1 左移至最高位。

第一种方法实现简单，只使用基本的选择器进行搭建，本质是一个优先编码器。对一个 *n* 比特二进制字符串 *e*，当第 *n*-1 比特为 1 时，代表前导 0 的数量为 0；若第 *n*-1 比特为 0，则对第 *n*-2 比特进行判断，如果其为 1，则前导 0 的数量为 1，以此类推。图 3-2 展示了 8 比特的前导 0 检测优先编码逻辑，设前导 0 数量为 *z*。

8'b1???????	*z*=0
8'b01??????	*z*=1
8'b001?????	*z*=2
8'b0001????	*z*=3
8'b00001???	*z*=4
8'b000001??	*z*=5
8'b0000001?	*z*=6
8'b00000001	*z*=7

图 3-2 前导 0 优先编码逻辑

另外一种前导 0 检测算法则引入位或运算并将 *z* 逐位输出，且并行执行左移位操作。假设 *e* 的位宽为 *n* 比特，那么表示前导 0 数量 *z* 所需的位宽为 $\log_2(n)$，且应向上取整。例如对一个 26 比特的二进制数，*z* 是一个 5 比特位宽的变量，z_i 即为第 *i* 比特值，且算法过程中会产生 *m* 个位宽为 *n* 的中间变量，定义为 *t*[*m*][*n*]，求解逻辑由公式(3-9)和公式(3-10)表示：

$$z_i=\begin{cases}\overline{|\,e[n-1:n-2^i]}, & i=m-1\\ |\,t[i+1][n-1:n-2^i], & i<m-1\end{cases} \tag{3-9}$$

$$t[i]=\begin{cases}t[i+1]<<2^i, & z_i=1\\ t[i+1], & z_i=0\end{cases} \tag{3-10}$$

其中 *i* 初始值为 *m*-1，之后每循环一次减 1，直至 *i* 等于 0，最终得到的 *t*[0]即为已完成左移位的二进制字符串，*z* 即为解得的前导 0 数量。

第一种电路在得到前导 0 的数量后还需要经过移位电路，也需要生成中间值 *t*，最

终总面积和时序都比第二种稍差。其好处在于将前导 0 数量的求解和前导 1 移位分开运算，单独计算 z 的数值，图 3-2 的电路所需延时比由公式(3-9)所表示的电路更短。在一些场景下更适合用第一种，例如如果前导 0 逻辑正好处于关键路径的末尾，其中一种改善时序的办法就是将前导 0 数量的求解和前导 0 移位分别于流水线寄存器的前后级实现以切断时序路径。而如果前导 0 逻辑在寄存器同一拍实现，那么由公式(3-9)和公式(3-10)所表示的电路更为适合，应根据具体的使用场景选择更为合适的前导 0 检测电路。

3.2 浮点乘加算法

3.2.1 浮点乘法算法流程

本小节对浮点乘法算法进行阐述以更好地与下文的乘加算法进行对比，图 3-3 展示了浮点乘法的算法流程，区别于浮点加减法，浮点乘法不需要对阶，在规格化之前，阶码和尾数并行执行于两条不同的通路。

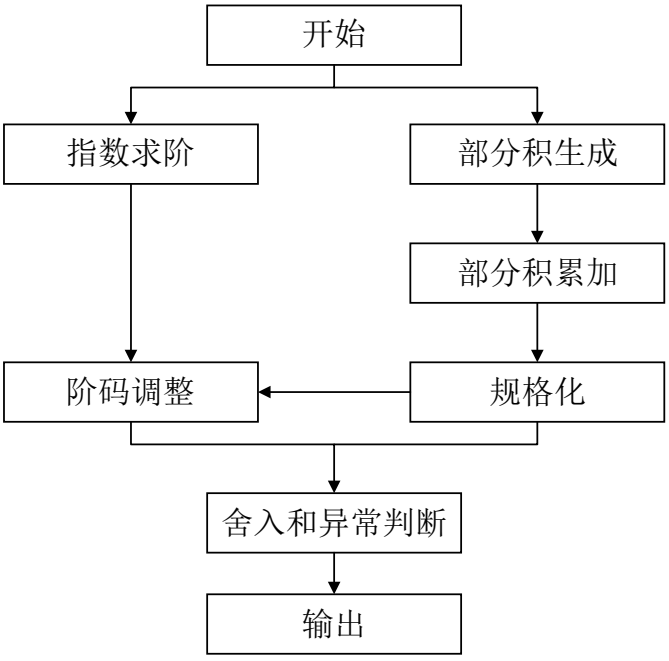

图 3-3 浮点乘法流程图

假设 $opb=\{s_a,e_a,f_a\}$，$opb=\{s_b,e_b,f_b\}$，浮点乘法 $opa\times opb$ 主要包括以下操作：

（1）指数求阶：由 e_a+e_b-bias 求得临时阶码 e_{tmp}，结果如果超过阶码允许的最大值将发生正溢出，应生成上溢异常指示，并输出 INF；如果求得临时阶码为负数，则需要在规格化阶段利用尾数进行调整。

（2）部分积（Partial Product，PP）生成和累加：执行定点乘法 $f_a\times f_b$，结果扩展为尾数位宽的两倍，进行乘法时用乘数的每一比特去乘被乘数，每次乘得的积即部分积，

之后将这些部分积累加。

（3）规格化和阶码调整：如果 e_{tmp} 为正值，将尾数乘法结果的第一个 1 左移至最高位，反之向右移位。区别于浮点加减法，浮点乘法规格化时使用的移位器需能实现两个方向的移位。在移位时对阶码进行调整，尾数左移时阶码应减小，反之增大。

（4）舍入和异常判断：同浮点加减法，对尾数多余的比特位进行舍入，并进行异常处理。

3.2.2 浮点乘加算法流程

本课题基于 RISC-V 协议，需要实现浮点乘加运算，单指令完成浮点乘法和浮点加减法操作，可由公式(3-11)表示：

$$result = opa \times opb + opc \tag{3-11}$$

与浮点加减法类似，*opa*×*opb* 的符号与 *opc* 相同时执行浮点乘加，此时浮点数 *opa*、*opb* 和 *opc* 的符号位异或结果为 0；反之当 *opa*×*opb* 的符号与 *opc* 相反时执行浮点乘减，此时符号位异或为 1，为方便叙述两种运算统称为浮点乘加。传统硬件设计上会先对 *opa* 和 *opb* 执行浮点乘法，之后与 *opc* 执行浮点加减法，但会由于进行两次舍入而导致精度损失。现代浮点运算器中会单独设计浮点乘加器，乘法得到的结果不进行规格化和舍入，而是直接与处理好的 *opc* 相加减，减小精度损失。

乘加将乘法和加减法合并处理以减少硬件的延迟和整体执行周期。另一方面，浮点乘加也是对浮点乘法的一种优化，当 *opc* 为 0 时乘加模块执行乘法运算。由于加减法的引入，当浮点乘法所求得的临时阶码为负数时，临时阶码将被 *opc* 的阶码所替代，最终所得到的临时阶码一定大于等于 0。这简化了规格化的移位设计，此时移位器只需要考虑左移的情况，特别是在向量运算中，这个优势更为明显。当执行多个小精度的浮点乘法时，它们有的需要左移而有的需要右移，这给硬件设计增加了难度，而乘加可以很好地规避这个问题，本课题也将设计浮点乘加模块。

当 *opa* 或 *opb* 为 1 时乘加器可执行加减法，综上乘加器可兼容浮点加减法和浮点乘法运算。本课题目标实现浮点数与整数的互相转换，由下文可知其所需的硬件资源与浮点加减法相近，因此本课题仍将实现浮点加减法模块以融合格式转换功能。在合理的设计下这并不会增加太多的面积开销，但可以使浮点运算器增加一套浮点加减法资源，使得浮点加减法模块和浮点乘加模块可以并行执行两种运算从而提高吞吐率，符合高面积效率的设计目标。这种做法也简化了浮点乘加器的控制逻辑，缩短时序路径，是一种对速度和面积进行权衡的做法，图 3-4 展示了浮点乘加算法的流程。

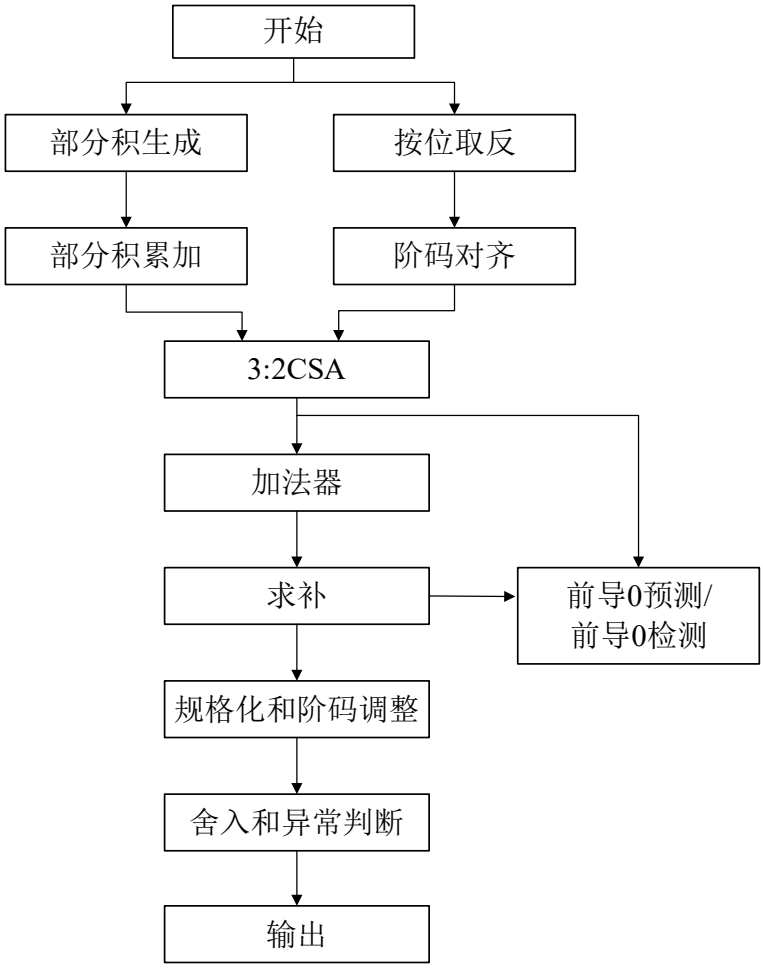

图 3-4 浮点乘加流程图

假设 $opa=\{s_a,e_a,f_a\}$，$opb=\{s_b,e_b,f_b\}$，$opc=\{s_c,e_c,f_c\}$，浮点乘加 $opa\times opb+opc$ 主要包括以下流程：

（1）部分积生成和累加：执行定点乘法 $f_a\times f_b$，并将部分积累加得到和（*sum*）和进位（*carry*）而不直接输出乘法结果。

（2）按位取反和阶码对齐：在浮点加减法中对阶是对绝对值小的浮点数的尾数进行移位。由于尾数相乘的延时较长，在浮点乘加中统一对 f_c 进行移位以使对阶移位与尾数相乘并行，从而缩短时序路径。这会导致移位器位宽增加，后续通路的加法器等硬件模块的位宽也随之增加，对速度和面积都会有不利的影响。如果执行的是浮点乘减，则应先对 f_c 按位取反，临时阶码的值于 3.2.3 给出。

（3）3:2CSA、加法器和求补：将（1）得到的 *sum* 和 *carry* 以及移位好的 f_c 通过 CSA（Carry Save Adder）缩减为两个加数后输入到加法器中。结果如果是负值应进行求补，类似浮点加减法，需要执行前导 0 预测或检测。

（4）规格化、舍入和异常判断：与浮点加减法流程类似。

部分积生成和累加的本质是完成尾数的定点乘法，与加减法运算相比，乘法要消耗更多的时间和硬件资源[36]。且如 3.2.3 所述，浮点乘加中所需移位器的位宽将增加，后续在对数据进行加法时也需要使用大位宽的加法器。如何在不影响速度的前提下，减少这些硬件资源的开销以减小面积是浮点乘加器的设计难点。

3.2.3 适用于乘加的对阶移位

在浮点加减法器的设计中通常采用“小阶对大阶”的方式进行对阶，即对于两个浮点数，需要首先判断出哪个浮点数绝对值更小，然后对这个浮点数的尾数进行移位。在乘加操作中，其中一个加数通过两数相乘得到，而乘法的延时是非常长的，如果仍按照这种方法进行对阶，则必须等待乘法执行完成，导致时序路径变长。为优化这个问题，乘加算法固定对 *opa*×*opb*+*opc* 中的浮点数 *opc* 的尾数 f_c 进行移位，以双精度浮点乘加为例，图 3-5 和图 3-6 表示 f_c 的移位过程。

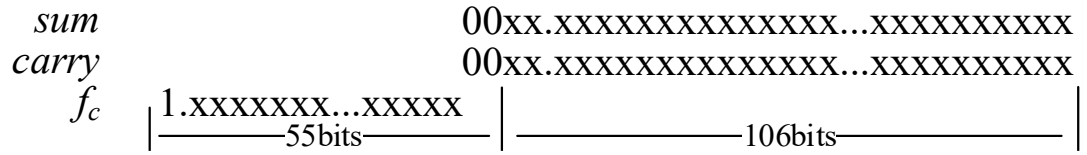

图 3-5 e_c-e_{ab}≥56 时 f_c 的移位过程

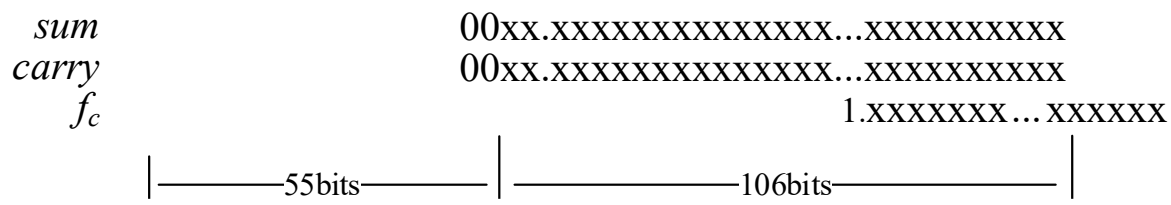

图 3-6 e_c-e_{ab}<56 时 f_c 的移位过程

sum 和 *carry* 由 f_a 和 f_b 相乘得到，假设 *opa* 和 *opb* 相乘后的阶码为 e_{ab}，当 e_c-e_{ab}≥56 时，f_c 和 *sum*、*carry* 的相对位置如图 3-5 所示，此时 f_c 无需移位，只需在 f_c 的 *GR* 位补 0 后将 *sum* 和 *carry* 置于 *R* 位之后，且对阶后的阶码取 e_c。当 e_c-e_{ab}<56 时，则将 f_c 向右进行移位，如图 3-6 所示，并考虑阶码的偏置值 *bias*，在双精度浮点数中，*bias* 取 1023，得到移位的位数 *rshiftnum* 由公式(3-12)表示：

$$rshiftnum = 56 + e_a + e_b - e_c - 1023 = e_a + e_b - e_c - 967 \tag{3-12}$$

通过这种方法实现将 f_c 的右移与尾数相乘并行以减小对阶所需的延时，提高硬件的速度。阶码也应根据移位的数值进行调整，此时临时阶码取为 e_c 加上 *rshiftnum* 的值。

单精度和半精度浮点数的尾数移位规则也是类似的，只是移位器所需的宽度有所区别。对于双精度，由图 3-5 和图 3-6 可得出所需的右移器位宽 161 比特（不考虑 *sum* 和 *carry* 的 *GRS* 时），同理要完成 1 个单精度的运算所需的移位器位宽为 74 比特，对于半

精度则为 35 比特。由于本课题的目标是实现高面积效率，为优化面积，只使用一个 161 比特的右移器来实现半精度、单精度和双精度的向量移位。

3.2.4 阵列乘法器

在浮点乘加器中，实现尾数乘法的定点乘法器占据很大的面积资源，且由于部分积的数量较大，增大了时延路径的长度，可能导致时序收敛困难，其是设计的难点。下面介绍几种定点乘法算法，阵列乘法器是最为传统的定点乘法器，存在很大的局限性。

$$
\begin{array}{lcccccccc}
A & & a_3 & a_2 & a_1 & a_0 & & & \\
B & \times & b_3 & b_2 & b_1 & b_0 & & & \\
\hline
 & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & & & \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & & & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & & \\
\hline
A\times B & d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0
\end{array}
$$

图 3-7 阵列乘法器运算流程

以位宽为 4 的二进制定点数 a 和 b 为例，假设执行 $a\times b$，相乘的流程可用图 3-7 表示，需要生成 4 个部分积，得到每个部分积的每一个比特 a_ib_i 使用一个与门，之后将每一列使用半加器或全加器进行相加，结果表示为 d_i。每一列产生的进位传递至相邻的一列参与计算，将该过程以硬件电路实现可得到图 3-8。

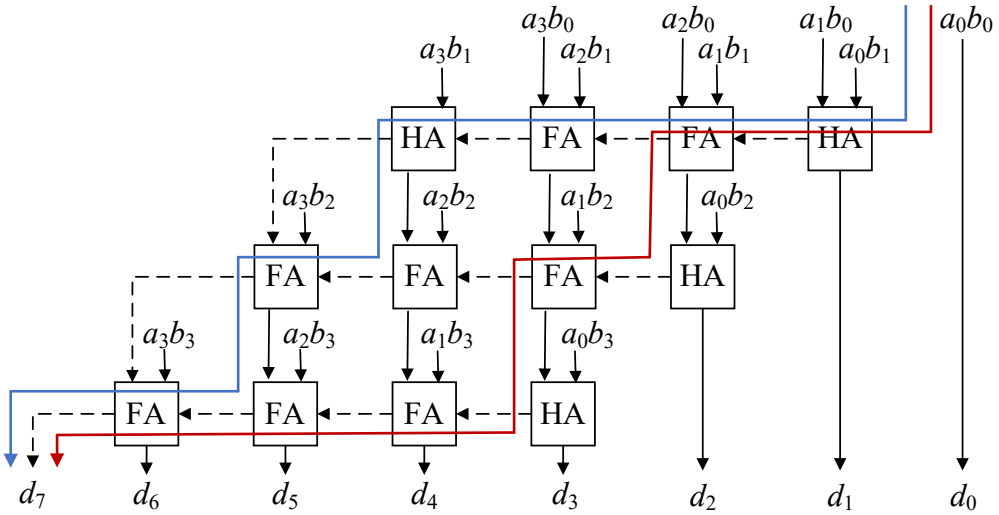

图 3-8 阵列乘法器硬件结构图

图 3-8 中 HA（Half Adder）表示半加器，FA（Full Adder）表示全加器，虚线箭头表示进位传播的路线。要实现 m 和 n 比特位宽的两数相乘，阵列乘法器将消耗资源 mn 个与门，n 个半加器和 mn-m-n 个全加器，蓝线和红线表示阵列乘法器中延时最长的路径。

如果在浮点乘法中使用这种乘法器阵列，对于双精度浮点乘法，尾数位宽为 53 位，那么这个阵列对于资源的消耗将是非常大的，需要 2809 个与门，53 个半加器和 2703 个全加器，显然这与高面积效率的设计目标是不符的。另外由于进位传播链的延时非常长，这会导致整个运算器只能在很低的工作频率下工作，而利用 Booth 乘法和华莱士树（Wallace Tree）可以很好地优化这个问题。

3.2.5 基 4 的 Booth 乘法器

乘法器的设计对大规模芯片的面积、速度和功耗有很大的影响，其中 Booth 算法多被用于高速和小面积设计中[37, 38]。Booth 算法对于乘法的优化是通过减少运算中加法的次数实现的，它是实现二进制乘法运算的高效算法。定点乘法运算主要由两部分组成，即生成部分积和部分积累加。Booth 算法可以大大减少部分积的个数从而加速累加，对于面积和速度都能起到很好的优化效果，本节对该乘法器进行阐述。

对于定点乘法 $a\times b$，将有符号数 b 展开后见式(3-13)：

$$b=-b_{n-1}2^{n-1}+\left(\sum_{i=0}^{n-2}b_i\times 2^i\right)=\sum_{i=0,i|2}^{n-2}\left[\left(-2b_{i+1}+b_i+b_{i-1}\right)\times 2^i\right] \tag{3-13}$$

其中 i=0,$i|2$ 表示 i 的初始值为 0，且 i 能被 2 整除，b_{-1} 取 0。阵列乘法器的部分积个数取决于乘数展开后 2 的幂次项的数量，Booth 算法则将乘数重新编码，由公式(3-13)可知，此时 b 展开后只剩下幂次为偶数的项，从而减少部分积的数量。基 4-Booth 乘法对 $-2b_{i+1}+b_i+b_{i-1}$ 进行编码，共有五种编码情况，以数集表示即{−2,−1,0,1,2}，如表 3-3 所示。

表 3-3 基 4-Booth 乘法编码表

b_{i+1}	b_i	b_{i-1}	$-2b_{i+1}+b_i+b_{i-1}$	部分积操作
0	0	0	+0	0
0	0	1	+1	a
0	1	0	+1	a
0	1	1	+2	$2a$
1	0	0	-2	$-2a$
1	0	1	-1	$-a$
1	1	0	-1	$-a$
1	1	1	-0	0

Booth 乘法的基表示编码位宽的最高位权重，对于基 4-Booth 乘法来说，每次将乘数的三个连续比特作为 Booth 编码模块的输入，该模块的输出选择对被乘数 a 的正确操作，包括移位和反转、反转、等于零、无操作或者移位，即在{-2a,-a,0,a,2a}中选择正确的结果输出部分积，这个过程在硬件上只涉及移位和补码运算。对于双精度格式，乘法器有 53 位，通过该算法进行重新编码，部分积的数量由 53 个减少到 27 个，部分积的个数大大减少，极大地优化了面积。图 3-9 显示了双精度运算下 Booth 乘法的部分积生成，PP_i 表示各个部分积，此处 s_i 表示部分积的符号。

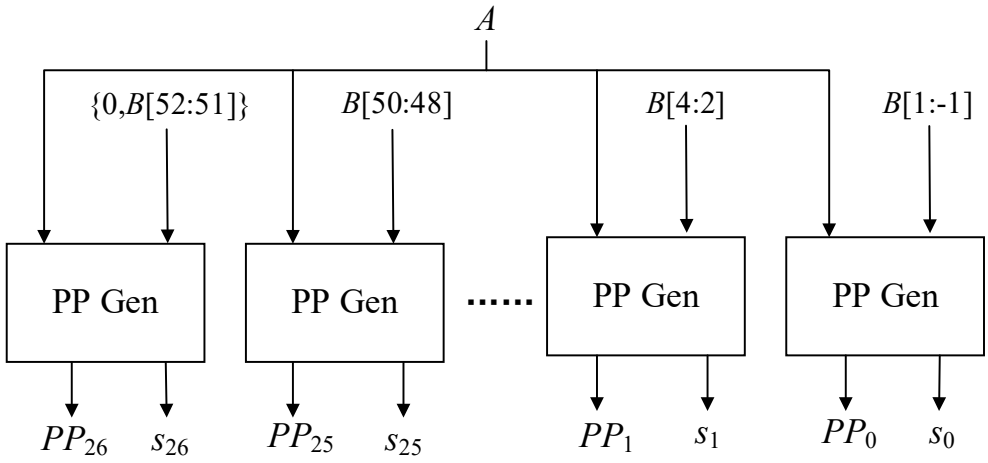

图 3-9 尾数乘法部分积生成逻辑

在生成部分积之后，利用华莱士树可以对部分积的数量进行进一步压缩，否则对大量的部分积进行累加意味着需要消耗大量的进位传播加法器（Carry Propagation Adder，CPA）资源。硬件设计上主要是利用进位保存加法器（Carry Save Adder，CSA），减少部分积的个数，从而减小长进位链导致的延时，由 CSA 单元组成的部分积累加网络即华莱士树。

3-2CSA 可以将输入的三个加数缩减为两个，单比特运算逻辑与全加器等效，但区别于 CPA，Wallace Tree 中 CSA 单元各比特彼此独立而无需将进位端相连[39]。CSA 实现的基本思想是将进位 *carry* 和和 *sum* 分别计算保存，且 *carry* 和 *sum* 的每一比特都是独立计算而互不干扰的，所以速度极快。假设计算三个 n 比特的二进制数 x、y 和 z 相加，*sum* 和 *carry* 在硬件上的求解逻辑由公式(3-14)和公式(3-15)得到，其中 *carry* 的低位填入 0，此处{}表示拼接符：

$$sum = x \oplus y \oplus z \tag{3-14}$$

$$carry = \{x \,\&\, y \mid y \,\&\, z \mid x \,\&\, z, 0\} \tag{3-15}$$

另外 3-2CSA 也可用来计算 $x+y-z$，减去 z 在硬件系统中通常处理为加上 z 的补码，

即加上 z 按位取反后的值再加 1。利用 3-2CSA 可得到 $x+y-z$ 的逻辑表达式见式(3-16)和式(3-17)，通过将 *carry* 的最低比特取为 1 实现加 1 操作，可以省去减法运算时求补码所需的加 1 电路从而提高面积效率。

$$sum = x \oplus y \oplus \bar{z} \tag{3-16}$$

$$carry = \left\{ x \,\&\, y \mid y \,\&\, \bar{z} \mid x \,\&\, \bar{z}, 1 \right\} \tag{3-17}$$

每三个部分积输入到 3-2CSA，然后输出 *sum* 和 *carry* 至下一级，依此类推。对于 27 个部分积，需要七级的 CSA，CSA 网络的结构如图 3-10 所示。

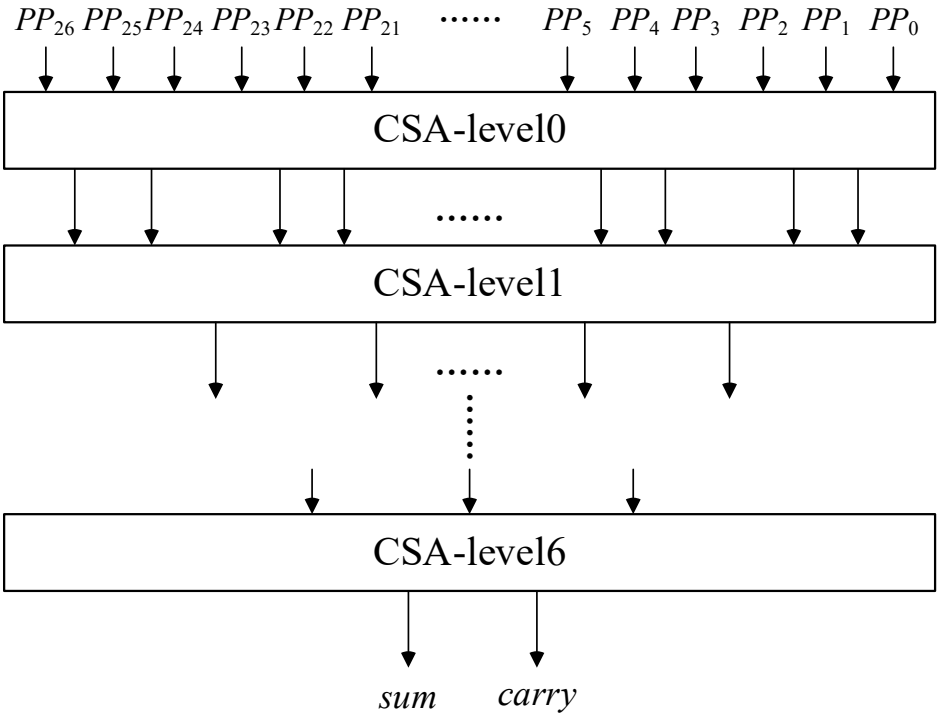

图 3-10 七级 CSA 网络

3.3 浮点除法算法

3.3.1 浮点除法算法流程

一般来说，浮点除法操作的执行周期在浮点系统中是最长的，这是由于流程中的尾数除法需要迭代多个周期才能完成运算。如果想减少迭代的周期数，那么一个周期内数据需要经过的组合逻辑路径也更长，对速度和面积都将产生不利的影响，这导致浮点运算器的关键路径多出现在尾数除法模块中。

浮点除法是浮点系统中的复杂运算，浮点除法模块也是影响浮点运算器面积和性能的一个主要单元[40]。浮点除法在规格化前可将阶码和尾数的运算并行处理，这方面与浮点乘法是类似的，整体流程如图 3-11 所示。

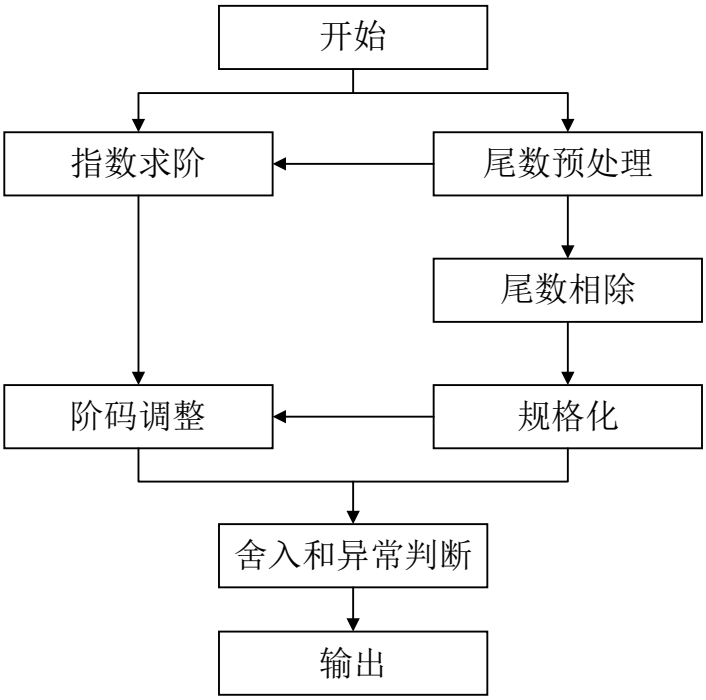

图 3-11 浮点除法流程图

假设 $opa=\{s_a,e_a,f_a\}$，$opb=\{s_b,e_b,f_b\}$，浮点除法 *opa*/*opb* 主要包括以下流程：

（1）指数求阶：由 e_a-e_b+bias 计算出临时阶码 e_{tmp}，e_{tmp} 可能为负数，也可能产生上溢，需先在阶码高位增加双符号位。

（2）尾数预处理：根据所使用的尾数除法算法而对尾数进行预处理的过程，并控制指数求阶模块对 e_{tmp} 进一步调整。当被除数左移一个比特时，e_{tmp} 应减 1，而当除数左移一个比特时，e_{tmp} 应加 1。

（3）尾数相除：对两个尾数执行定点除法操作，这是浮点除法算法的核心流程，该流程耗费了整个浮点算法执行的大部分时间，迭代多个周期直至算出所需精度的商。

（4）规格化和阶码调整：类似浮点乘法，该规格化也包括左移和右移两种情况，区别于浮点乘加算法处理成规格化时只进行左移，浮点除法算法更多时候会处理成只需对尾数进行右移，并在移位过程中对指数进行调整。

（5）舍入和异常判断：与前文其他运算类似。

浮点除法模块在执行尾数相除时需要迭代多个周期，硬件上需要大面积和长延时的定点除法器，这是整个浮点除法模块设计的关键。本文在尾数除法上主要研究了恢复余数除法算法、不恢复余数除法算法和 Newton-Raphson 算法，最终选择以不恢复余数算法进行硬件实现，下面具体介绍这三种尾数除法器。

3.3.2 恢复余数除法器

恢复余数除法算法流程类似于传统的定点除法，区别在于运算时只需要考虑每个比特只有 0 和 1 两种情况。当当前余数大于等于除数时，用当前余数减去除数后，剩余余数左移一个比特；而当当前余数小于除数时，为简化硬件设计，仍用当前余数减去除数，之后对剩余余数的符号位进行判断。如果判断出剩余余数为负，则将除数加回，剩下的余数再左移一个比特，这种情况需要多执行一次加法，因此也称为恢复余数算法。

恢复余数算法需要重复多次的加减操作以实现除法运算，这将导致路径延时和执行周期增加，且需要在硬件上增加判断和恢复逻辑，使得硬件设计复杂化，难以实现目标的面积效率，恢复余数除法器并不符合本课题的研究目标。

假设执行 4 比特有符号数除法，图 3-12 展示了每计算商的一比特时恢复和不恢复余数算法的硬件电路，左图为恢复余数算法的电路结构，右图为不恢复余数算法[41]。其中 $r_3r_2r_1r_0$ 为上一轮运算得到的余数（即当前余数），$r'_3r'_2r'_1r'_0$ 为此轮运算后得到的余数（即剩余余数），$y_3y_2y_1y_0$ 为除数，q_n 即为此轮所求得的商的一比特值。对于恢复余数算法来说，每经过一轮减法器的运算后，还需要根据余数的符号来选择输出减法器的结果或者是当前余数，余数的符号位 r_3 和 r'_3 一定是 0。实现这个功能需要一个与减法器同位宽的选择器，这增加了硬件面积和路径延时。右图则为不恢复余数算法除法器，省去了选择逻辑，且利用可控加减法单元（Controllable Addition And Subtraction Unit，CAS）可以在不增加硬件逻辑门数的基础上实现求余数所需的加减法控制，从而实现更高的面积效率。其无需保证余数为正数，因此余数符号位不一定为 0，下文介绍此算法。

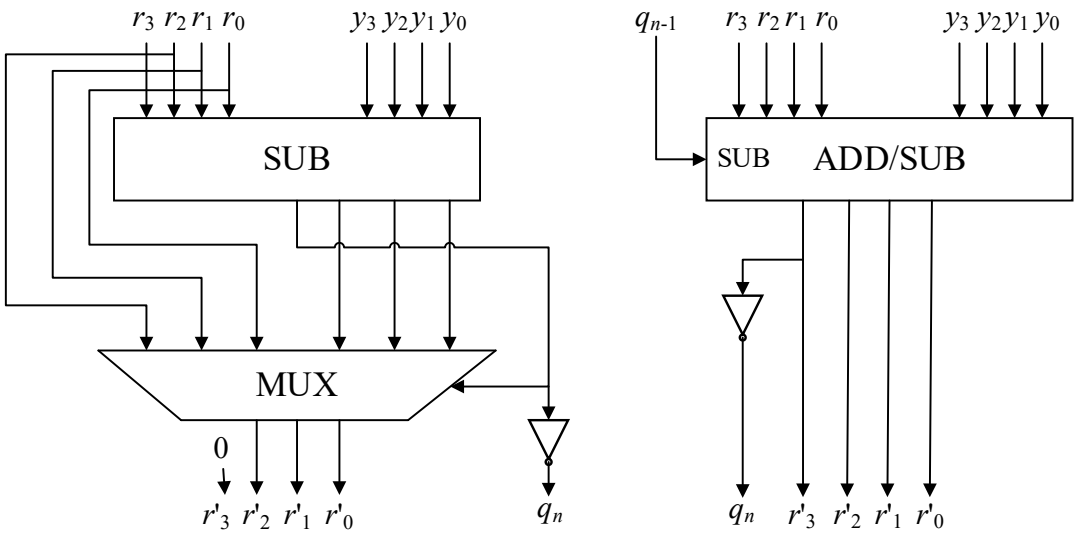

图 3-12 恢复余数和不恢复余数算法电路结构

3.3.3 不恢复余数除法器

假设执行定点除法 a/b 的过程中，第 n-1 轮计算得到的剩余余数为 r，当 r 大于等于 0 时，第 n 轮得到的余数 r'是将 r 左移一位后再减去除数 b，此时商的一比特值 q_n 取 1，r'由公式(3-18)计算：

$$r' = 2r - b \tag{3-18}$$

而当 r 小于 0 时，在恢复余数算法中会先恢复余数，然后再左移一位后减去除数 b，那么下一个余数 r'由公式(3-19)计算：

$$r' = 2(r + b) - b = 2r + b \tag{3-19}$$

r 小于 0 的情况下，不恢复余数算法定义此时商的一比特 q_n 取-1。由此可将恢复余数的步骤定量化，在进行除法时，余数或者左移一位后加上 b，或者左移一位后减去 b，而无需在每次运算后判断是否需要进行余数的恢复，因此称为不恢复余数算法。该算法也称为加减交替算法，只需要在每次加减操作后对余数符号进行判断。以数集的角度考虑，不恢复余数算法的每一比特商是从{1,-1}选取的，而恢复余数算法则从{1,0}进行选取。

由于-1 无法在二进制中表示，需要采用其他方法将不恢复余数算法的商集{1,-1}也调整到{1,0}上。如果求得 q_n 为-1，那么当化为标准的二进制表示时其需要向 q_{n-1} 借位，不恢复余数算法在实现为硬件时，通常会使除法阵列在运算的同时也完成借位操作。设第 n 轮接收到第 n-1 轮的剩余余数 r，如果 r 大于等于 0，那么一比特商 q_{n-1} 取 1，并将 r 左移一位后减去除数 b 得到新余数；反之如果 r 小于 0，那么 q_{n-1} 取 0，将 r 左移一位后再加上除数 b 得到新余数。

如图 3-11 右图所示，不恢复余数算法在硬件设计上更具有优势，每产生商的一个比特位只用完成一次加法或减法操作，而使用可控加减法单元 CAS 就可以实现这个功能，CAS 是不恢复余数除法器最基本的执行单元。在硬件设计里，减法通常是通过使用加法器去加上减数的补码实现的，而 CAS 只需要将求补码的按位取反改为异或即可在不增加逻辑门数的基础上实现加减法的切换。不恢复余数算法在减法操作后不需要进行余数恢复，从而省去恢复余数所需的选择器，不仅使得整体的延时更短，也使得面积开销更小。不恢复余数算法有利于实现高面积效率，本文以该算法为基础进行硬件实现。

CAS 的硬件结构如图 3-13 所示，硬件上由全加器和二输入异或门组成[42]。它有四个输入和四个输出，其中包括本位输入 x_i 和 y_i，来自低位的进位或借位 c_i，p 控制该单元执行加法或者减法。p 为低电平时，执行加法操作，反之执行减法，此时相当于将 x

减 y 转化为 x 加上 y 按位取反的值后再加 1。其中加 1 操作通过控制最低位的 CAS 输入的 c_i 为 1 实现，另外输出信号包括本位和或差 r_i 及进位 c_{i+1}。

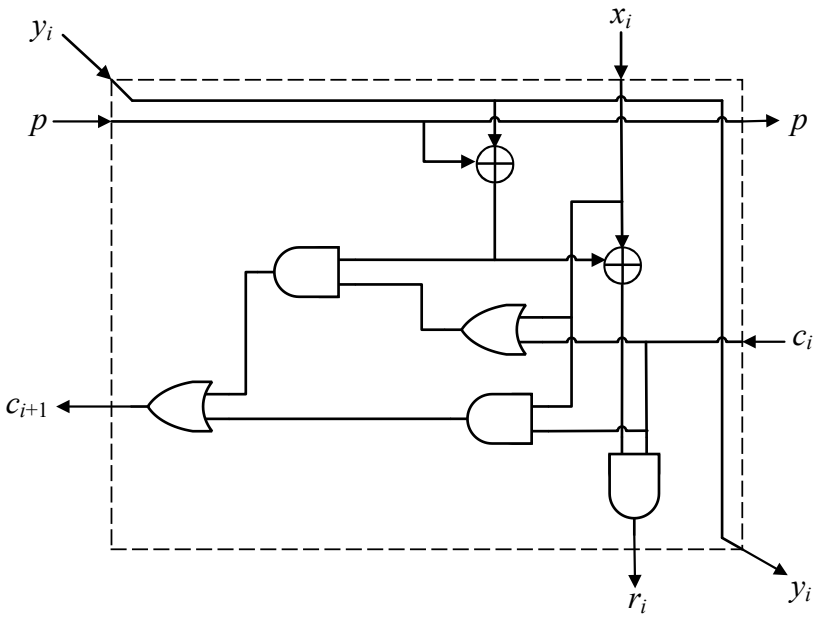

图 3-13 可控加减法单元结构图

CAS 单元的输入与输出逻辑关系由公式(3-20)和公式(3-21)表示[43]：

$$r_i = x_i \oplus (y_i \oplus p) \oplus c_i \tag{3-20}$$

$$c_{i+1} = (x_i + c_i)(y_i \oplus p) + x_i c_i \tag{3-21}$$

实现两个位宽为 k 的二进制数相除，只需要在阵列除法器的每一级将 k 个 CAS 级联起来即可，而每一级运算后再将 CAS 的输出信号输入至阵列除法器的下一级，整个阵列除法器的级数取决于商值的位宽。

假设执行 4 比特除法，被除数 x 为 $x_3x_2x_1x_0$，除数 y 为 $y_3y_2y_1y_0$，阵列除法器的架构如图 3-14 所示。第一级固定执行减法，因此第一级所有 CAS 的 p 值为 1，并将最低位 CAS 输出的 p 再输入到 c_i。第一级本质上对 x 和 y 进行大小的比较，当最高位的 CAS 输出 c_{i+1} 为 1 时，代表 x 大于等于 y，此时剩余余数为正值，下一级的 CAS 执行减法；当最高位的 CAS 输出 c_{i+1} 为 0 时，则代表 x 小于 y，此时剩余余数为负值，那么下一级的 CAS 依照不恢复余数算法的要求应执行加法，其他级同理，此 c_{i+1} 的值即为所求得商的一比特值 q_n。在硬件上可以通过将每一级最高位 CAS 输出的 c_{i+1} 作为 p 输入至下一级所有 CAS 即可实现自动判断除法器阵列的每一级应执行加法还是减法操作。$q_3q_2q_1q_0$ 为所求得的商，若需要求得更大精度的商则可利用 $r_3r_2r_1r_0$ 继续往下求，只需要增加 CAS 阵列的级数。

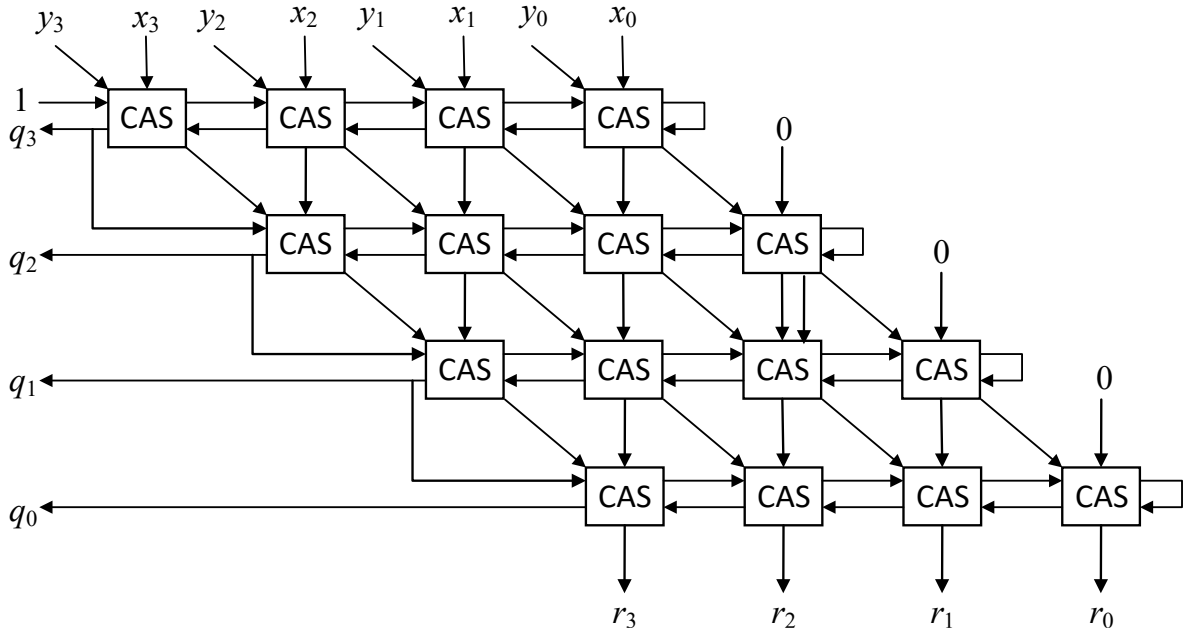

图 3-14 不恢复余数除法器 CAS 阵列[43]

由于 CAS 阵列每一级的结构相同，假设要求一个周期求得 $q_3q_2q_1q_0$，那么可以采用图 3-14 的四级阵列；而如果允许两个周期实现，则可以采用二级阵列，当前周期最后一级 CAS 的输出寄存后作为下一周期第一级 CAS 的输入即可。如果不考虑寄存器面积，除法器阵列的面积会减小一半，而工作最高频率则约增加一倍。这种阵列的设计非常灵活，可根据执行周期、面积和工作最高频率的不同需求进行调整。

3.3.4 Newton-Raphson 除法器

相比于恢复余数和不恢复余数算法使用加减法来实现除法，另外一种方法则是用乘法来替代除法运算，这种算法称为 Newton-Raphson 算法。

Newton-Raphson 算法的实现逻辑是先求出除数的倒数，然后再用其与被除数相乘，最后得到所需精度的商。假设执行定点除法 a/b，基本思想是假设有函数 $f(x)$，猜测存在 x_i，初始为 x_0，那么 $f(x)$在 x_0 的切线方程由公式(3-22)表示：

$$y - f(x_0) = f'(x)(x - x_0) \tag{3-22}$$

利用该切线方程可以得到使 y 为 0 的新的点 x_1，且 x_1 更加接近最后的精确值，如此循环迭代即可得到最后的 x_n，x_{i+1} 的迭代逻辑由公式(3-23)表示：

$$x_{i+1} = x_i - f(x_i)/f'(x_i) \tag{3-23}$$

将这个思想应用到除法上即可实现 Newton-Raphson 除法器，令 $f(x)=1/x-b$，那么要使 $f(x)=0$，x 应取 b 的倒数 $1/b$，每一轮迭代得到 x_{i+1}，运算逻辑由公式(3-24)表示：

$$x_{i+1} = x_i(2 - x_i b) \tag{3-24}$$

综上，Newton-Raphson 除法器的实现流程如下：

（1）对除数 b 进行移位，使其处于[0.5,1)的范围。

（2）使用 b 的若干高比特位查表得到 $1/b$ 的一个近似值 x_0。

（3）由公式(3-24)进行迭代，直到所求 x_n 可以满足所要求的精度。

（4）由于 $x_n=1/b$，因此计算 $a\times x_n$ 得到所需的商，且应把结果相对于（1）进行反向移位，以消除（1）产生的影响。

3.3.5 尾数除法算法对比

由公式(3-24)可知，Newton-Raphson 除法器每轮计算所得到的结果的精度是上一轮的两倍，与单级 CAS 阵列相比，Newton-Raphson 除法器所需的迭代次数更少，这适用于要求总周期数更少的硬件设计中，这类算法也称为函数迭代法，通常可以实现高性能。但该算法执行复杂度高，要实现除法需引入乘法电路，其中$(2-x_ib)$还引入了求补码电路，且需要消耗一定的存储资源以实现（2）所需的查找表资源，因此 Newton-Raphson 除法器所需的硬件资源相比于不恢复余数算法更多，面积也更大。在执行周期数相同的情况下，Newton-Raphson 除法器单周期所经过的组合逻辑路径更长，可能会受到建立时间的限制而导致时序问题。

尽管可以考虑复用浮点乘加模块中的尾数乘法单元，但乘法单元为实现向量运算已引入了较多的控制逻辑，本身为长延时模块，如果兼容除法将加剧时序问题。另外还可能造成资源冲突，由于 Newton-Raphson 算法需要迭代多个周期得到所需精度的结果，在此期间将一直占用尾数乘法模块，运算器将无法处理浮点乘加运算而导致阻塞问题，降低了运算器的吞吐率。且函数迭代法所提供的商的最低位并不准确，可能导致舍入操作无效，需要进行修正以满足 IEEE 754 标准对精度的要求，导致额外的面积开销[44]。

恢复余数算法和不恢复余数算法同属于数字循环算法，这类算法以加、减为基本运算，通过线性收敛来得出最后结果。同属数字循环算法的还有基 2 的 SRT 算法，它将不恢复余数算法的商集从{1,-1}调整到{1,0,-1}，当商位选择 0 时，只需完成移位而无需进行加减运算，从而减少了迭代过程的平均延时以加速除法实现，但这种操作需要通过在数据通路增加选择逻辑实现。如前文所述，浮点除法中尾数除法一般通过迭代多个周期实现，平均延时减小并不会影响所需的执行周期数，SRT 算法的优势并不明显。SRT 算法的基数决定了单级可得到多少比特的商，设基数为 r，那么单次循环得到 $\log_2(r)$位商，基 2 单次循环和不恢复余数算法一样得到商的一比特值，而基 4 可以得到两比特商。随着基数的提高，所需迭代次数也减少，但相应的硬件资源消耗也增大[35]。

不恢复余数除法器设计灵活，可根据执行周期和面积等要求调整CAS阵列的级数，最大的优点是所需的硬件成本低，并可实现低延迟，对工作频率和面积都有积极的影响，更适合高面积效率设计[33]。综上本课题选用不恢复算法进行尾数除法实现，且由3.4可知，该算法可以兼容开平方操作以节省面积。

3.4 浮点开平方算法

3.4.1 浮点开平方算法流程

除了常用的加、减、乘、除，开平方也是运算系统必不可少的操作。RISC-V浮点指令集包含开平方指令，开平方操作虽然只占到整个程序很小的一部分但又不可或缺，对整个处理器的性能有很大的影响，本课题研究的浮点运算器将实现开平方运算的功能。

浮点开平方的流程与浮点除法类似，只需将图 3-11 中的尾数除法模块替换为尾数开平方模块以进行尾数的定点开平方，而指数求阶的推导于 4.4.2 给出。尾数开平方模块是实现浮点开平方的核心模块，直接影响整个浮点开平方单元的面积和速度。

无论是规格化浮点数还是非规格化浮点数数开方，结果均为规格化数。对于规格化处理，在设计时可将尾数开平方模块处理为输出的结果最高位一定为1，这样处理后可省去规格化模块的使用，这对于高面积效率设计是有利的。

定点开平方运算本质上是一种特殊的除法运算，因此适用于定点除法的算法大部分也可以应用于定点开平方运算，只需要另外增加部分控制逻辑。这也是本课题选择实现开平方运算的一个原因，在硬件上可以设法复用定点除法模块，只需要增加很小的面积和硬件资源开销，便可实现定点开平方功能，这符合高面积效率的设计目标。

3.4.2 不恢复余数开平方器

本节对不恢复余数开平方算法具体进行阐述，不恢复余数算法也可用于定点开平方操作，但输入到CAS阵列的数据与执行除法有所不同[41]。

定义结果寄存器变量 reg_q，如果所求平方根 $q=q_{n-1}q_{n-2}...q_3q_2q_1q_0$，那么 reg_q 总位宽为 n，寄存的初值为0。另外假设CAS阵列算得的余数存储在余数寄存器变量 reg_r 中，位宽与 reg_q 相同，寄存的初值也为0。区别于除法在进行每级运算时需要将上级得到的余数左移一位后补0，开平方需要将余数左移两位后在低位补上被开平方尾数还未进行运算的最高两个比特（开平方时每一轮运算需使用被开方数的两个比特）。假设被开平方的尾数为 $1a_0.b_{m-1}b_{m-2}b_{m-3}\ldots b_1b_0$ 或 $01.b_{m-1}b_{m-2}b_{m-3}...b_1b_0$，其中 a_i 和 b_i 为0或者1，使用尾数寄存器变量 reg_f 寄存该尾数，该变量位宽应大于等于 m+2。

基于以上假设，CAS 阵列第一级的输入分别为：（1）reg_r 低位拼接 reg_f 的最高两位；（2）reg_q 低位拼接 01 或者 11，取决于此级执行减法还是加法，减法取前者;（3）来自上级最高位 CAS 输出的 c_{i+1}，作为当级 CAS 的 p 输入控制执行加法或者减法，由于第一级一定执行减法，因此第一级的 p 值默认为 1，从而可以将（2）统一为在 reg_q 低位拼接{~p,1}。第一级得到余数 r，硬件逻辑由公式(3-25)表示：

$$r = \left\{reg_r[n-1,0], reg_f[m+1,m]\right\} - \left\{reg_q[n-1,0], \overline{p}, 1\right\} \tag{3-25}$$

其中当执行减法时 p 为 1，反之则执行加法，r 为求得的余数。由公式(3-25)可知，执行开平方时 CAS 阵列每一级需要的 CAS 数量至少为 n+2，当取比 n+2 大的值则相当于在 reg_q 和 reg_r 的高位存入 0，不影响最终结果。第一级执行结束后，将 r 的低 n 位存入 reg_r 中更新其值，而 reg_q 则左移一位后并在最低位存入当级最高位 CAS 输出的 c_{i+1}，该值即为所求得的一比特平方根，reg_f 更新为左移两位后的值。如果当级最高位 CAS 输出的 c_{i+1} 为 1，该 c_{i+1} 作为 p 信号控制下一级 CAS 阵列进行减法运算，反之执行加法，CAS 阵列其他各级的运算逻辑与第一级所描述的方法是类似的。

当执行除法时 CAS 的数量和被除数和除数的位宽有关，而执行开平方时 CAS 的数量则跟 reg_q 和 reg_r 的位宽有关，两者不一定相等。为匹配 CAS 阵列执行除法和开平方时所需的 CAS 数量，本课题采用一种优化思路，使开平方运算时单级阵列的 CAS 数量减少 2，在下文进行硬件实现时将使用这种方法以实现两种运算下 CAS 数量一致。

当执行定点减法时，硬件上通过被减数加上减数的补码来实现的。由公式(3-25)可知，最低两位执行加法时将 reg_f 的最高两位与{1,1}相加，而执行减法时则与{0,1}相减，硬件上则是加上{0,1}按位取反后的值再加 1。如果只看每一级最低的两个 CAS，实际上无论是执行加法还是减法，最终这两个比特都是与 2'b11 相加，真值表如表 3-4 所示。

表 3-4 低两个比特 CAS 真值表

$reg_f[m+1]$	$reg_f[m]$	$r[1]$	$r[0]$
0	0	1	1
0	1	0	0
1	0	0	1
1	1	1	0

由此开平方时可以省去最低两个 CAS 单元，取而代之使用简单的逻辑门进行计算，

运算由公式(3-26)、公式(3-27)和公式(3-28)表示：

$$c_0 = reg_f[m+1] \mid reg_f[m] \tag{3-26}$$

$$r[1] = \sim \left(reg_f[m+1] \oplus reg_f[m]\right) \tag{3-27}$$

$$r[0] = \sim reg_f[m] \tag{3-28}$$

如果想要实现除法器和开平方器复用同一个不恢复余数算法 CAS 阵列，以上这种处理方式是非常有用的，且相比于 CAS 的逻辑更为简单，一定程度上减小了延时和面积，优化了面积效率。

3.4.3 Newton-Raphson 开平方器

Newton-Raphson 算法同样可用于实现开平方操作，设对定点数 d 进行开方，假设 q 为所求的平方根，令 $f(x)=1/x^2-d$，则当 $f(x)$为 0 时，$x=1/q$，利用 Newton-Raphson 算法可得到迭代逻辑由公式(3-29)表示：

$$x_{i+1} = \frac{x_i\left(3 - x_i^{\,2} d\right)}{2} \tag{3-29}$$

得到满足所需精度要求的 x_n 后，再乘以 d 即可得到所求平方根值。尽管其与除法实现的本质思想相同，但由公式(3-24)和公式(3-29)可知，两者实现逻辑差别较大，开平方时需要多执行一次乘法操作和右移操作，使得浮点除法和浮点开平方复用同一套除法资源存在不小的难度，也对流水线的划分造成一定的困难，且 Newton-Raphson 算法实现为硬件存在资源消耗大的问题。

在 3.3.5 已对比了多种算法实现尾数的定点除法时的优缺点，而这些算法执行尾数开平方时的优缺点也是类似的。不恢复余数算法执行除法和开平方时控制逻辑简单且结果准确，硬件可移植性好，与其他算法比较，该算法占用资源极少，处理速度较快[45,46]。本课题在进行尾数开平方模块的硬件实现时同样采用不恢复余数算法，复用不恢复余数除法所使用 CAS 阵列以减少硬件开销，提高面积效率。

3.5 浮点数与整数的互相转换

本课题实现相同位宽的浮点数和整数间的互相转换，其中包括浮点数和有符号整数以及浮点数和无符号整数间的互相转换。

3.5.1 浮点数转整数

浮点数向整数转换是程序编程时使用频率较高的操作，RISC-V 浮点指令集也包含了浮点数和整数相互间转换的指令，浮点数转整数的算法流程如图 3-15 所示。

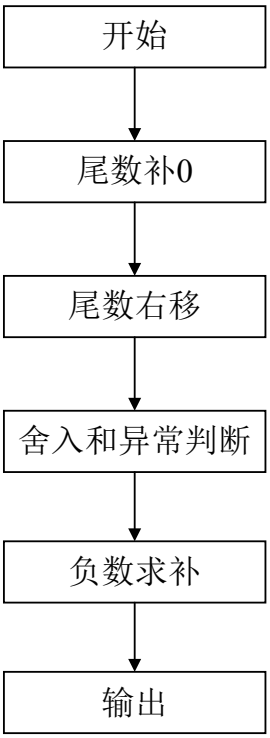

图 3-15 浮点数转整数流程图

以单精度浮点数 $opa=\{s_a,e_a,f_a\}$转 32 位整数为例，主要包括以下流程：

（1）尾数补 0：由于浮点尾数只有 23 位，加上隐藏位为 24 位，而目标整数的位宽为 32，因此首先在 f_a 低位以 0 进行填充直至位宽与整数位宽相同。

（2）尾数右移：尾数补 0 后假设尾数的小数点也右移到最右端，相应地 e_a 应减去 31，这要求 $e_a \geq bias+31$，而当 $e_a<bias+31$ 时，则应将补 0 后的尾数右移，此时求出移位数为 $bias+31-e_a$，$bias+31$ 定义为转换常量 con。

（3）舍入和异常判断：根据舍入模式进行舍入，该过程无需进行规格化，因为目标格式并不是浮点数。此类目标格式为整数的浮点操作只会产生无效异常和不精确异常，异常处理不同于目标格式为浮点数的浮点操作。

（4）负数取补：若目标整数为有符号整数，如果转换结果为负数应进行取补操作后再输出。

该算法的核心模块为步骤（2）使用的尾数右移，可以复用浮点数加减法中的对阶移位模块以提高资源利用率，提高面积效率，要注意的是其异常判断和处理与浮点加减法有所不同。

由于浮点数的表示范围比整数大得多，因此在转换过程中可能会出现转换结果无法在目标格式中表示的情况。当待转换浮点数为 NaN、无穷大或者其他转换后超出目标格式表示范围的有穷浮点数时，运算器应生成无效操作异常信号。区别于目标格式为浮点

数的操作，浮点数转整数不会产生上溢和下溢异常。而当转换操作的结果值可以用目标格式表示但为非精确值时，基于 RISC-V 协议规定生成不精确异常。

RISC-V 规定的单精度浮点数转整数的最大和最小有效输入如表 3-5 所示，当输入超出有效输入范围或为 NaN 时，应按表 3-5 的规定进行输出，其他精度的转换同理。

表 3-5 单精度浮点数转整数的范围和无效输入的行为

	浮点数转有符号整数	浮点数转无符号整数
舍入后的最小有效输入	-2^{31}	0
舍入后的最大有效输入	$2^{31}-1$	$2^{32}-1$
超出范围的负输入时的输出	-2^{31}	0
-∞时的输出	-2^{31}	0
超出范围的正输入时的输出	$2^{31}-1$	$2^{32}-1$
+∞或 NaN 时的输出	$2^{31}-1$	$2^{32}-1$

3.5.2 整数转浮点数

除了 16 位无符号整数在转换成浮点数的过程中可能产生上溢，其余格式的整数转换为浮点数均不会产生异常，整数转浮点数的算法实现流程如图 3-16 所示。

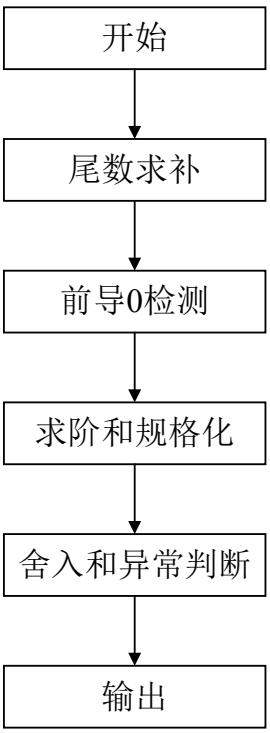

图 3-16 整数转浮点数流程图

以 32 位整数转单精度浮点数 $opa=\{s_a,e_a,f_a\}$为例，主要包括以下流程：

（1）尾数求补：由于尾数都以正值存储，如果是对有符号整数进行转换且为负值，应先对其进行求补。假设小数点在隐藏位后面，此时临时阶码 e_{tmp} 取为 $31+bias$。

（2）前导 0 检测：规格化数尾数的隐藏位一定是 1，因此需要对整数进行前导 0 检测以找出第一个 1 的位置，记第一个 1 前有 z 个前导 0。

（3）求阶和规格化：对尾数进行左移直至隐藏位为 1，考虑规格化尾数左移对阶码的影响，可得到临时阶码 e_{tmp} 为 $31+bias-z$，即 $con-z$。

（4）舍入和异常判断：从 32 位整数舍入到 24 位尾数（包含隐藏位），可能生成不精确异常。如果是 16 位的无符号整数转浮点数，还可能生成上溢异常，这通常发生在无符号整数为 16 个比特均为 1 的情况。

整数转浮点数时所需使用的硬件资源包括前导 0 检测模块和尾数规格化的左移模块，与浮点数转整数操作类似，此操作也将复用浮点加减法的硬件资源实现以提高资源利用率。

3.6 本章小结

本章主要对本课题预期实现的 RISC-V 中的浮点运算操作进行介绍，包括浮点加减法、乘法、乘加、除法和开平方，以及浮点数和整数的互相转换等基本算法流程，浮点运算模块在这些算法流程上进行实现。同时对各个浮点运算中的关键模块所采用的算法也进行细致阐述，包括浮点加减法中的前导 0 预测和检测算法、浮点乘加中的基 4-Booth 算法等，并对比了多种尾数除法和开平方算法。可认为，本章为本课题进行浮点运算器的硬件实现提供了理论基础。

第四章 浮点运算器的设计与优化

本章对 RISC-V 中的浮点运算操作进行硬件实现，主要对本课题的硬件设计部分进行阐述。首先介绍运算器整体架构，之后重点阐述各个浮点运算模块的实现，包括浮点加减法与格式转换模块、浮点乘加模块和浮点除法与开平方模块，并着重对其中的关键子模块进行详述。

4.1 浮点运算器的整体架构

浮点运算器的整体架构如图 4-1 所示，主要包括输入控制逻辑、执行逻辑和输出控制逻辑，输入三个 64 比特的源浮点数 *opa*、*opb* 和 *opc* 后最终得到 64 比特的结果浮点数 *result* 和 5 比特的异常指示信号值 *flag*。

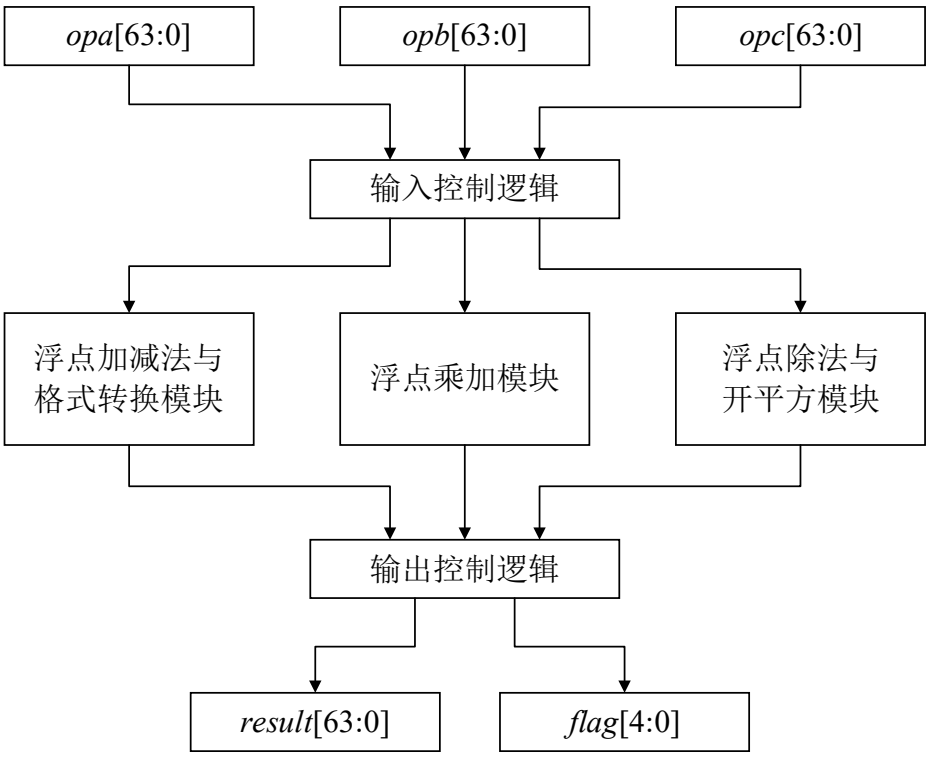

图 4-1 浮点运算器整体架构

输入控制逻辑将接收到的源浮点数 *opa*、*opb* 和 *opc* 送往执行逻辑中的子模块进行处理并产生相应的使能信号，本课题目标实现 RISC-V 协议所规定的浮点运算功能，包括浮点加法、减法、乘法、乘加、乘减、除法、开平方、浮点数与无符号整数互相转换以及浮点数与有符号整数互相转换运算，输入控制逻辑根据不同的运算需求去使能不同的子模块。

另外本课题以支持多种浮点精度为实现目标，可实现半精度、单精度和双精度三种精度的浮点运算，应在输入选择相应的精度，且本课题实现的浮点运算器可执行向量运

算。以 *opa* 为例，数据结构如图 4-2 所示，每一个 64 比特的浮点数可存储一个 64 比特的双精度浮点数 $fp64_0$，或者两个 32 比特的单精浮点数 $fp32_0$ 和 $fp32_1$，或是四个 16 比特的半精度浮点数 $fp16_0$、$fp16_1$、$fp16_2$ 和 $fp16_3$。

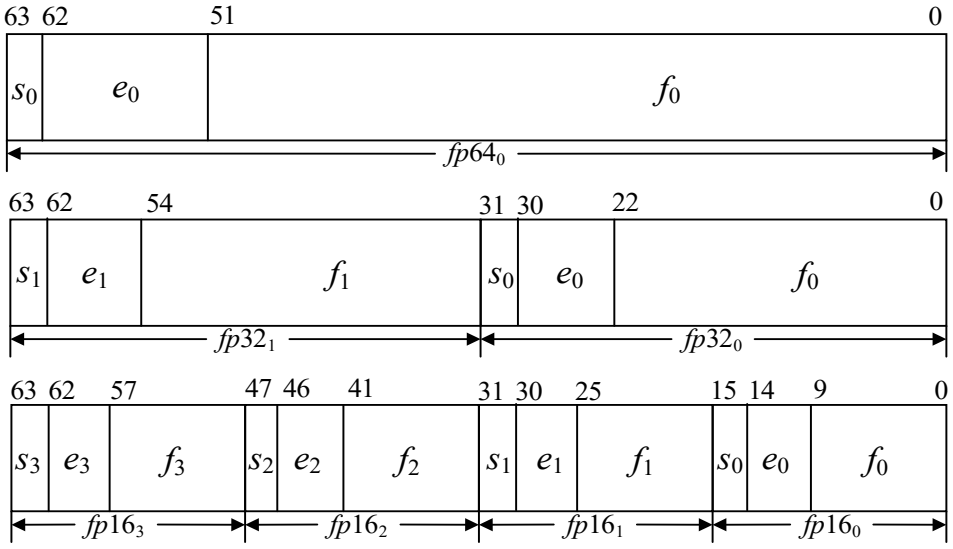

图 4-2 浮点数存储格式

执行逻辑是整个浮点运算器的核心，以高面积效率为目标进行研究和设计，遵循 IEEE 754 和 RISC-V 的规定，执行选定的浮点运算。模块的划分考虑了不同运算所需硬件资源的匹配度，若两种运算执行时所需资源相近则划分到同一模块实现。最终此部分逻辑由三个子模块组成，包括浮点加减法与格式转换模块（FADDCVT）、浮点乘加模块（FMA）和浮点除法与开平方模块（FDIVSQRT），下文对这三个模块进行具体阐述。

输出控制逻辑的本质为选择逻辑，将选定的子模块输出的运算结果作为结果浮点数 *result* 进行输出，*result* 遵循图 4-2 的格式，并输出 5 比特的异常指示信号 *flag*，*flag* 从高位到低位分别指示无效操作异常、除 0 异常、上溢异常、下溢异常和不精确异常。

4.2 浮点加减法与格式转换模块的设计与优化

4.2.1 浮点加减法与格式转换模块整体架构

基于浮点加减法的算法流程，该模块以四级流水线进行规划，整体架构如图 4-3 所示，虚线表示流水线寄存器，并复用浮点加减法的硬件资源融合实现浮点数和整数的互相转换功能。该模块可实现一个双精度，或两个单精度，或四个半精度浮点加减法，当源浮点数的符号位异或为 0 时执行浮点数加法，反之执行减法。另外也可实现一个双精度浮点数与 64 位整数，或两个单精度浮点数与 32 位整数，或四个半精度浮点数与 16 位整数的互相转换，整数格式包括有符号和无符号整数。

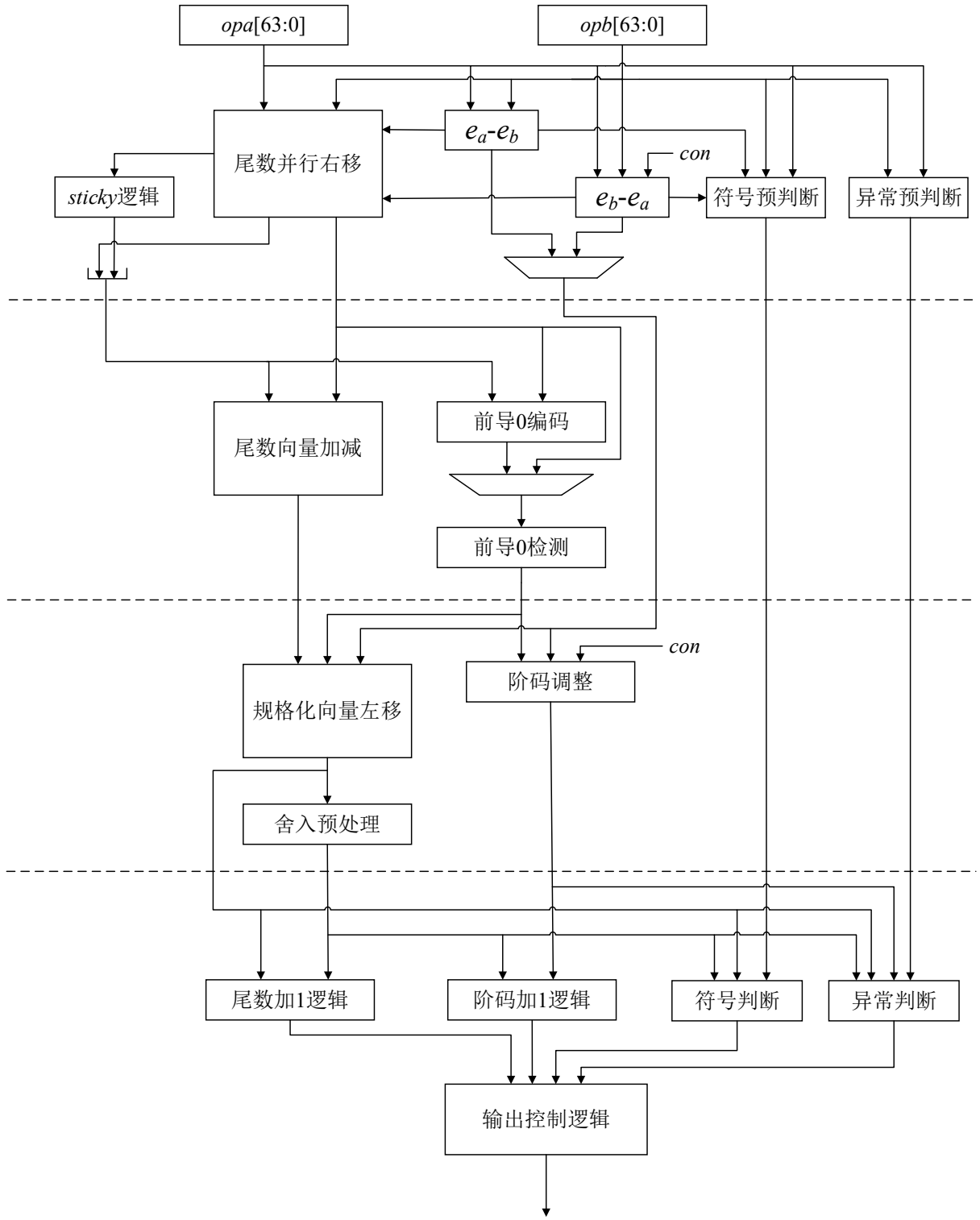

图 4-3 浮点加减法与格式转换模块整体架构

第一级通过计算两个浮点数阶码的差值来确定尾数对齐需要向右移位的位数，结果浮点数的符号位可进行预判断，取为实际数值大的浮点数的符号位，临时阶码也取为更大的浮点数的阶码。在这个过程中，所需要的硬件资源包括小位宽的减法器以求出阶码

差、大位宽的移位器以实现尾数右移。其中 *GRS* 的 *S* 位由被右移出的比特值相或得到，图中以 *sticky* 逻辑表示。这和浮点数转整数时的移位操作所需的硬件资源是一致的，将两者统一起来以实现资源复用，此时移位值由转换常量 *con* 减去 *opa* 的阶码得到。对于双精度浮点数转整数来说，*con* 的值为 1086，对于单精度为 158，对于半精度则是 30。同时在该级进行异常信号的预判断，根据 RISC-V 规定判断结果是否可能为 NaN、INF 等非正常值，生成使能信号并传递至第四级以生成异常指示信号。

第二级使用 SIMD 优化的加减法器完成尾数加减，并计算结果的前导 0 数量，为规格化做好预处理。为了缩短时序路径，利用 3.1.2 所阐述的前导 0 预测算法，使该流程和尾数加减并行，缩短第二级的组合逻辑延时以优化速度。前导 0 检测模块采用图 3-2 的实现方法，并可被整数转浮点数操作复用，从而减少硬件资源的浪费以减小面积。

第三级的核心是使用 SIMD 优化的左移器以实现规格化操作的向量化。假设 e_{tmp} 为临时阶码，z 为前导 0 的数量，当 $e_{tmp}>z$ 时规格化结果为规格化数，如果左移后最高位仍为 0，代表预测有一比特的误差，尾数再左移一位以进行预测补偿。反之当 $e_{tmp}\leq z$ 时，规格化后的结果仍为非规格化数，此时移位位数由临时阶码决定而与前导 0 的个数无关，无需进行预测补偿。假设左移位数为 *lshiftnum*，求解由公式(4-1)表示：

$$lshiftnum=\begin{cases} e_{tmp}-1, & e_{tmp}\leq z \\ z, & e_{tmp}>z \end{cases} \tag{4-1}$$

在移位的过程中对阶码进行调整，对于整数转浮点数操作，阶码等于转换常量 *con* 减去前导 0 的数量。同时在第三级对 *GRS* 进行舍入预处理，根据所选择的舍入模式判断是否需要对规格化的结果进行加 1 操作，需要则拉高加 1 使能信号。

第四级根据接收到的加 1 使能信号选择是否进行舍入加 1，并对结果浮点数的符号进行最终判断。当两个具有相反符号的浮点数的和或者两个具有相同符号的浮点数的差为精确零（即无不精确异常），且此时舍入模式为向下舍入时，输出的浮点数符号位应取为 1。此级需依照 RISC-V 规定进行异常处理，最终输出正确的结果。

对于源浮点数 *opa* 和 *opb*，当执行浮点加减法 *opa*+*opb* 时应注意以下情况：（1）当任一源浮点数为 NaN 时结果输出 qNaN;（2）当两个源浮点数均为∞且执行有效减法时，例如+∞+(-∞)，虽源浮点数没有 NaN 但结果仍输出 qNaN。其中（1）只有在源浮点数存在 sNaN 的情况下会将无效操作异常指示置位高电平，而（2）只要发生即将其拉高。

4.2.2 基于速度优化的并行右移器

对阶移位时一般只对更小的浮点数的尾数进行右移，这个比较关系可以通过在 e_a 和

e_b 最高位增加符号位后相减得到，这也意味着需要等待整个阶码相减计算完成后才能利用差值的最高位来判断浮点数的大小关系，继而进行尾数右移，降低了硬件的速度。

利用并行化思想，针对阶码对齐时组合逻辑延时较长的问题，本课题采用一种可对速度进行优化的并行右移器，整体结构如图 4-4 所示，两个操作数的阶码分别定义为 n 比特的 e_a 和 e_b，尾数分别为 f_a 和 f_b。当模块接收到 e_a 和 e_b 时，并行执行两个减法，即 $\{0,e_a\}$-$\{0,e_b\}$和 e_b-e_a，前者高位补 0 是为了利用差值的最高位来判断阶码的大小关系。对 f_a 和 f_b 分别使用分步右移器进行移位，由于减法结果一定是从低位到高位逐位输出，当分步右移器首先接收到阶码差的最低比特结果时，如果其为 1，右移器将相应尾数右移一位，否则保持不变。之后接收到阶码差的第二位时，如果其为 1，右移器则进一步将上一步的结果右移两位。每得到阶码差的一个比特，右移器进行一次移位。

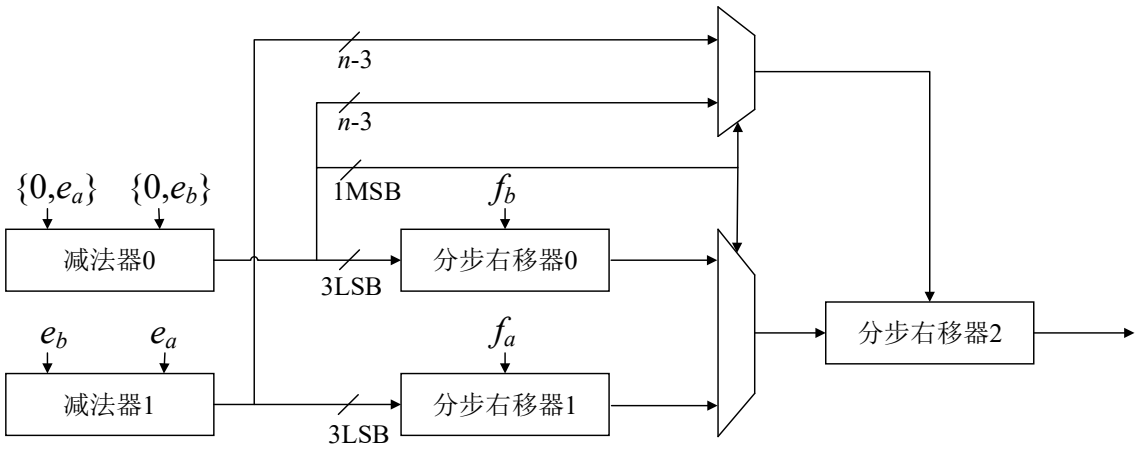

图 4-4 并行右移器结构图

如果 f_a 和 f_b 都响应阶码差的所有比特进行右移，移位器的资源消耗将会很大，对面积的消极影响将超过对速度的积极影响，面积效率反而下降。为了优化这个问题，注意到分步移位器的实现逻辑本质上是选择逻辑，其每次从不移位和移位的两个结果中选择出正确的结果输出，相比于减法器得到输出的一比特的速度，选择逻辑的速度要慢得多。利用这个思想，使与 f_a 和 f_b 相连的右移器只响应阶码差的最低几个比特，在本设计中取三个比特，当执行完三次右移操作后，利用$\{0,e_a\}$-$\{0,e_b\}$的最高位结果进行选择。当其为 1 时，代表 e_b>e_a，那么选择 f_a 右移的结果以进行下一步移位，并选择减法器 1 输出的除最低三位比特的 n-3 比特的值来控制这个移位操作；反之为 0 时，则取 f_b 右移的结果进行下一步移位，并选择减法器 0 输出的 n-3 比特的值来控制。通过这种方法，分步右移器 0 和 1 只需要控制是否需要右移 1、2、4 个比特，所需的硬件资源减少，从而降低并行化对于面积所带来的消极影响。

综上可确定右移器的位宽，对于双精度浮点数的尾数来说，包含隐藏位的位宽为 53 比特，加上 *GR* 位（*S* 位由 *sticky* 逻辑得到），移位器所需的最小位宽为 55 比特。另外为了与双精度浮点数转整数算法的尾数右移实现资源共享，执行此运算时尾数低位补 0 直至位宽等于整数位宽 64 比特，加上 *GR* 位，可确定右移器的位宽为 66 比特，它可对一个双精度、单精度或半精度浮点数的尾数进行右移。为实现向量运算，还需要 34 比特位宽的右移器对一个单精度或半精度浮点数的尾数进行右移，以及两个 18 比特位宽的右移器对两个半精度浮点数的尾数进行右移。

4.2.3 尾数加减法的 SIMD 优化

合理的数据结构可以简化控制逻辑，使得不同精度的浮点数的尾数可以共享控制信号，优化面积效率。对于双精度浮点数的加减法，包含隐藏位的尾数位宽为 53 比特，加上 *GRS*，在进行尾数加减时需要一个 56 比特的加减法器，而大位宽的加减法器也会占据大的面积。本课题对尾数加减法进行 SIMD 优化，以实现在向量运算中可复用这个加减法器完成一个双精度，或两个单精度，或四个半精度浮点数的尾数加减法操作。

包含隐藏位和 *GRS*，双精度浮点数的尾数加减法位宽为 56 比特，单精度为 27 比特，半精度则为 14 比特。只需在单精度浮点数的尾数低位补一个 0，那么双精度与单精度运算，以及单精度与半精度运算所需的加法器位宽均为两倍位宽的关系，从而实现位宽匹配。图 4-5 表示尾数的数据结构，f_d、f_s 和 f_h 分别表示双精度、单精度和半精度浮点数尾数，硬件上将尾数以这种格式组合之后输入至加减法器进行计算。前导 0 预测与尾数的加减法并行执行，将图 4-5 的数据输入前导 0 预测模块后，得到的输出既是 f_{d0} 的前导 0 数量，也是 f_{s0} 与 f_{h0} 的前导 0 数量。这也是单精度尾数补 0 在低位而不是高位的原因，且它们可以共享一部分加减法器的控制信号，f_{s1} 和 f_{h2} 同理。

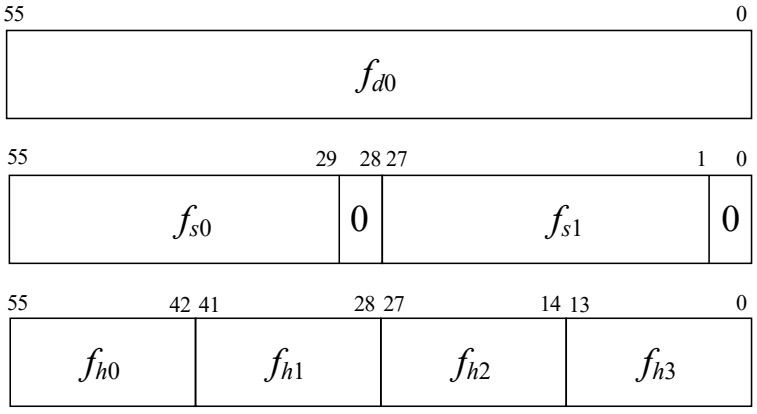

图 4-5 加减法器的数据结构

本课题的尾数向量加减电路基于“硬件隔离”的思想进行设计，通过在大位宽的设

计中添加部分控制逻辑以将数据通路分割为多条并行的小位宽的数据通路，从而满足多种精度下向量运算的需求，实现多种精度融合的尾数向量运算结构。本课题将大位宽的加法器分解为四个小位宽加法器，通过串行实现尾数加减法，且这四个小位宽加法器执行加法或者减法是独立可控的，这对于向量运算是必须的。例如进行单精度运算时，四个尾数并不一定都执行加法或都执行减法。其结构如图 4-6 所示，包含四个级联的 14 比特加减法器 adder，加减法器的内部结构由 3.3.3 所提出的 CAS 级联构成，这种单元很适合用于小面积设计。对于 n 比特的加法器，只需要增加 n 个异或门就可以实现减法功能，且加减法的切换是可控的，无需通过增加求补码电路来实现减法。在浮点数加法中，两个同位宽加法可能在最高位产生一比特的溢出，此时在规格化时会将尾数右移一位且阶码加 1，而通过硬件隔离可以很方便地获取到这些溢出信号，它们就是每个小位宽加法器所输出的进位信号 c_{i+1}。

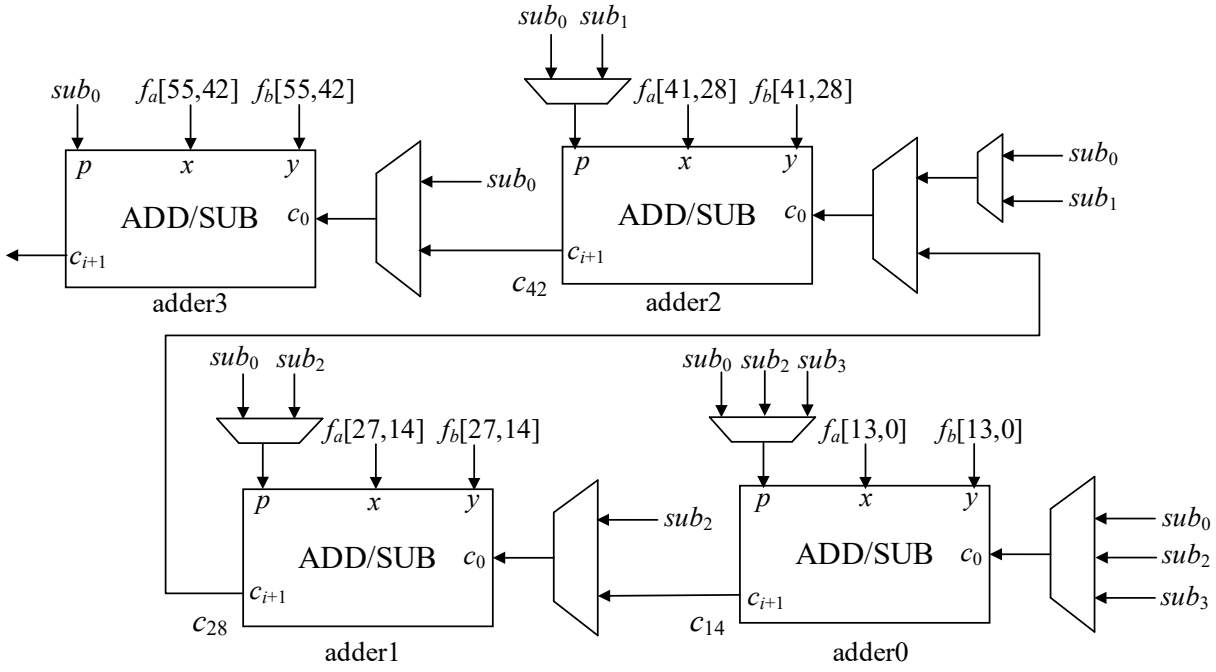

图 4-6 向量加减法器硬件结构

假设对阶后的尾数为 f_a 和 f_b，它们都遵循图 4-5 的数据结构，sub 指示当前数据块执行加法还是减法操作，f_{d0}、f_{s0} 和 f_{h0} 共享 sub_0，f_{h1} 受 sub_1 控制，而 f_{s1} 和 f_{h2} 共享 sub_2，f_{h3} 则受 sub_3 控制。adder0 首先计算尾数的最低 14 个比特，如果执行的是双精度浮点加减法，那么选择器将选择 sub_0 来进行控制执行加法还是减法，以此类推。如果此时产生溢出，那么 adder0 最高位的 CAS 输出的 c_{i+1}（即 c_{14}）被置为高电平，c_{14} 只有在计算单精度或双精度浮点数操作时才会传递至下一个加减法器 adder1。对于半精度浮点数，adder1 的进位信号受 sub_2 控制而不选择 c_{14}，图中所有选择器均受当前执行的浮点数精

度控制。此时就完成了一个半精度浮点数尾数的加减法运算，或是一个单精度浮点数尾数加减运算的二分之一，对于双精度则是四分之一。之后 adder1 继续完成接下去的 14 比特加减法运算，并在操作格式为双精度浮点数的情况下将最高位的 CAS 输出的 c_{i+1}（即 c_{28}）传递至 adder2，其他精度下，adder2 的输入 c_0 则取决于其执行的是加法还是减法。adder2 和 adder3 的功能是类似的，不做过多阐述，直至完成 56 比特的完整运算。每个 adder 最高位 CAS 输出的进位信号 c_{i+1} 传入流水线下一级以供规格化操作使用。在设计中保证减法操作一定是大的尾数减去小的尾数，所以减法并不会产生溢出，传入规格化级的进位信号也一定为 0。

这种结构既可以实现四个或两个小位宽加减法独立可控，也可以完整地完成一个大位宽的加减法，最大程度地复用了加减法器资源。只需要额外引入几个选择器进行控制，就可以实现只使用一套运算资源实现半精度、单精度和双精度浮点数的尾数向量加减法运算，减少资源的浪费，对提高面积效率是非常有利的。

4.2.4 尾数左移器的 SIMD 优化

在对阶移位时为了实现速度优化使用了 4 个并行右移器，而当时序裕量比较充足时，以更小面积为目标，可以通过引入额外的控制逻辑实现只使用一个移位器完成多组移位操作，实现向量化移位。本课题通过 SIMD 优化，设计向量移位器来实现浮点数加减的尾数向量左移位和整数转浮点数时的整数向量左移位操作。

移位器的位宽不应小于所需移位数据的最大位宽，本课题移位数据位宽最大的情况为整数转双精度浮点数操作，此情况下需要对 64 比特位宽的整数进行左移位，因此向量左移器的位宽定义为 64 比特。该移位器也可对两个 32 比特位宽的整数左移以转换为单精度浮点数，或是四个 16 比特的整数左移以转换成半精度浮点数。而在浮点数的加减法操作中，双精度浮点数的尾数位宽为 56 比特，在左移操作中保持高位对齐，在尾数低位补八个 0 以匹配位宽，单精度和半精度情况同理。

基于以上讨论，移位器的数据结构如图 4-7 所示，其中 *int*64、*int*32 和 *int*16 分别表示 64、32 和 16 比特位宽的整数，其他字母含义同图 4-5。本设计中共有四个不同位宽的前导 0 检测模块，当执行整数转浮点数功能时，第一个前导 0 检测模块接收图 4-7 的数据，而当执行浮点加减时，则接收前导 0 编码输出的序列，检测得到的前导 0 数量定义为 z_0，前导 0 计算已于 3.1.2 和 3.1.3 给出。z_0 可表示 $int64$、$int32_0$、$int16_0$、f_{d0}、f_{s0} 和 f_{h0} 的前导 0 的数量，这也是采用高位对齐的原因，可通过一个检测模块得到多种数据格式的前导 0 数量。$int16_1$ 和 f_{h1} 的前导 0 数量则为 z_1，z_2 和 z_3 同理

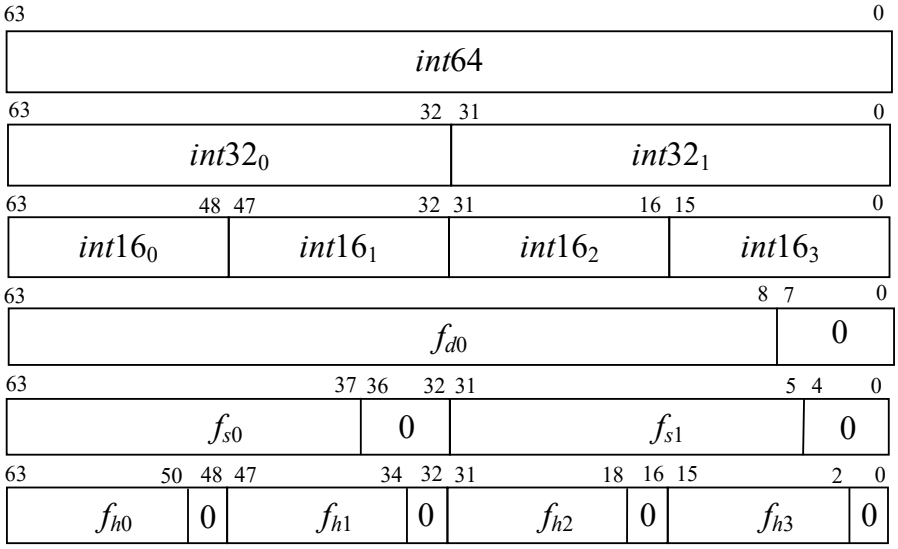

图 4-7 向量左移器的数据结构

为了实现在进行向量左移时各部分移位独立可控且实现更小面积，在进行规格化左移时，基于“硬件隔离”的思想，向量化的左移位操作通过将一个大位宽左移器分割为四个小位宽的左移器实现，实现多种精度融合的尾数向量移位结构，以减小硬件开销。

向量移位器在设计上基于 4.2.2 所提到的分步移位器的思想，每一级的移位操作（本质上为一个二选一的选择器）受移位位数 *lshiftnum*（简写为 *l*）的一比特信号控制，*l* 的计算见式(4-1)，具体的控制信号根据当前运算执行的精度进行选择。向量左移器的结构如图 4-8 所示，此处以低 32 比特的左移器结构为例进行阐述。假设对图 4-7 所示的数据进行移位，设为 s_0，以第一级为例，如果此时执行半精度浮点数相关（包括半精度浮点数加减法和 16 位整数转半精度浮点数操作）的向量移位，s_0[15:0]的移位受 l_3[0]控制，而如果执行的是单精度浮点数相关的运算，则受 l_2[0]所控制，双精度则受 l_0[0]控制，左移位时低位均移入 0。s_0[31:16]的移位逻辑也是类似的，只是低位并不是像前者一样一定移入 0。如果此时执行半精度浮点数相关运算，那么 s_0[31:16]和 s_0[15:0]的移位操作相互独立，此时 s_0[31:16]左移时仍移入 0；反之如果为单精度或双精度相关运算，那么 s_0[31:0]应视作一个整体，此时 s_0[31:16]左移时应移入低位部分的最高位，即 s_0[15]，第一级完成移位后得到 s_1。第二至第四级同理，区别在于左移的位数分别为 2、4 和 8 个比特。对于半精度浮点数相关运算，数据位宽最大为 16，那么前导 0 数量 *z* 除第零至第三位外均为 0 值，即在第四级移位后已能得到正确的移位结果。如果此时执行的是单精度或双精度相关运算，那么可将移位器的左右两部分 16 比特合并为 32 比特以继续进行更大位宽的左移位。

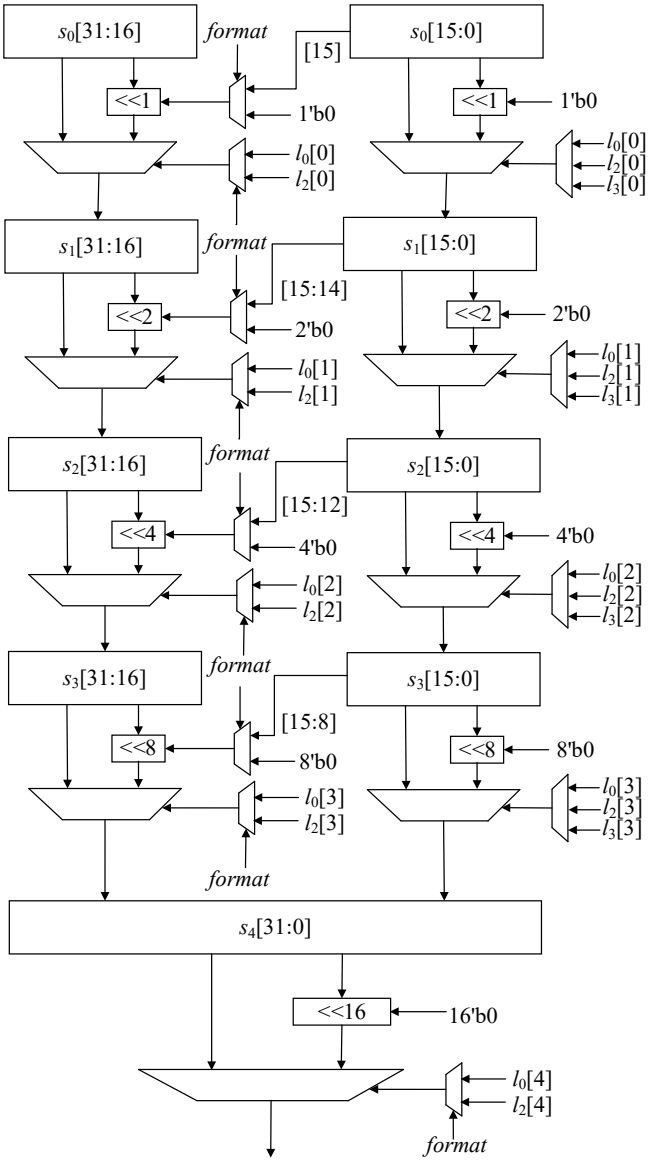

图 4-8 32bits 复合左移器结构图

基于以上讨论，本设计使用 64 比特的向量左移器，只需要将图 4-8 中每一级的硬件资源扩展一倍以对 s[63:32]执行类似操作，最终总级数为六级。基于这种结构，只需要引入少量的选择逻辑进行控制，就可实现只使用一个向量移位器完成浮点数加减法和整数转浮点数运算所需的向量化移位操作，并支持半精度、单精度和双精度浮点数，达到面积优化的目的。

4.3 浮点乘加模块的设计与优化

4.3.1 浮点乘加模块的整体架构

相比于浮点乘法模块，目前处理器更倾向于使用浮点乘加器，其可以提高机器学习和图像处理等领域的浮点性能[47]。RISC-V 也规定了浮点乘加操作，如 3.2.2 所述，本课题除设计浮点加减法模块，也对浮点乘加模块进行设计。当源浮点数的符号位异或为 0

执行浮点乘加，反之执行浮点乘减。浮点乘加器使用四级流水线，整体架构如图 4-9 所示，虚线表示流水线寄存器，本节对流水线各级所执行的操作进行阐述。

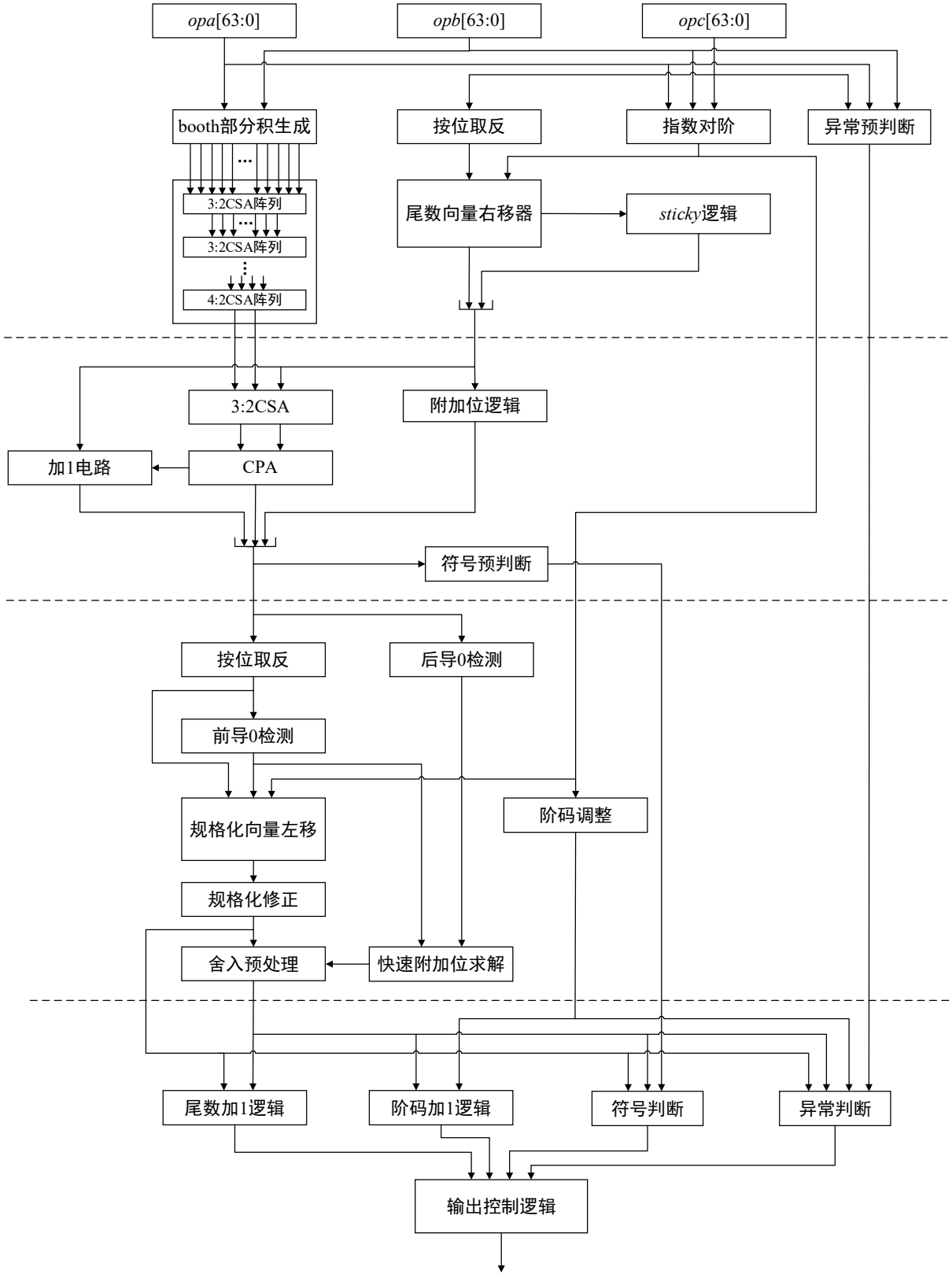

图 4-9 浮点乘加模块整体架构

第一级完成尾数乘法操作，本课题采用基 4-Booth 乘法，*opa* 和 *opb* 的尾数输入到

Booth 部分积生成模块以生成 27 个部分积，之后输入到 CSA 阵列构成的华莱士网络以缩减部分积数量，进行部分积累加。区别于浮点加减法需要先比较两数大小后才能确定对哪个操作数进行移位，在 *opa*×*opb*+*opc* 中固定对 *opc* 的尾数 f_c 进行移位，移位的数值由 3.2.3 阐述的对阶逻辑得出，并算出临时阶码的值。注意如果执行浮点乘减，则应先将 f_c 按位取反。向量移位操作通过一个大位宽的向量右移器实现，实现上与 4.2.4 所阐述的方法类似，只是将左移更改为右移。

第二级完成加法相关的操作，得到 CSA 阵列输出的两个加法项后，与对阶移位的结果通过 3-2CSA 缩减为两个加法项后，输入进位传播加法器中完成加法操作。为减少大位宽加法器的资源消耗，高位通过加 1 电路完成加法，而低位的 *GRS* 通过附加位逻辑处理，具体原理在 4.3.5 介绍。由于此处尾数位宽达 161 比特，考虑到前导 0 预测模块对面积的占用较大，本设计不使用前导 0 预测，而直接将图 3-2 实现的前导 0 检测模块置于流水线第三级以缩短第二级的组合逻辑延时。

第三级对尾数进行规格化左移操作，实现上采用 4.2.4 提出的向量左移器，移位数由公式(4-1)得到，并对阶码进行调整。由于前级对于负结果只进行按位取反而没有加 1，当所求得的负数为连续的 1 接上连续的 0，例如 11111000，按位取反后为 00000111，此时求得的前导 0 数量为 5，但对于负数求补还应加 1，正确的前导 0 数量为 4。这种情况下所求得的前导 0 数量存在 1 的误差，这导致左规时多左移一个比特，需要将规格化尾数右移一个比特以进行规格化修正。

另外由于未对负结果进行加 1 操作，可能导致数值错误，除了在第三级进行规格化移位，还应进行负结果修正。此处使用一种快速附加位算法并将修正可能导致的有效尾数加 1 与舍入加 1 合并处理，具体原理于 4.3.6 阐述。

第四级根据舍入预处理模块所输出的使能信号选择是否对尾数和阶码加 1。同时该级根据 RISC-V 规定完成异常判断并生成异常指示信号，对于乘加来说，异常可能来自于乘法操作，也可能来自于加法操作。

对于被乘数 *opa* 和乘数 *opb*，需要注意以下情况：（1）当其中一个源浮点数是∞，而另一个源浮点数既不是 0 也不是 NaN 时，*opa*×*opb* 的结果为∞；（2）只要有一个源浮点数是 NaN，*opa*×*opb* 的结果为 qNaN；（3）∞和 0 相乘，*opa*×*opb* 的结果为 qNaN。其中（2）只有任一源浮点数为 sNaN，才将无效操作异常指示拉高，而（3）只要发生即将其拉高。之后将 *opa*×*opb* 视为一个整体，根据浮点加减法的异常处理判断与 *opc* 可能产生的异常即可。

4.3.2 部分积的位宽优化

执行尾数乘法所使用的定点乘法器通常会消耗大量的硬件资源，其是面积效率优化的重点[48]。在执行尾数乘法时，由于定点乘法的乘积为被乘数和乘数的位宽之和，那么 Booth 算法生成的部分积在运算中将会进行符号位扩展以将位宽扩展至乘积的位宽。如果能对这部分扩展的符号位进行优化，或者说把它们处理为 0，那么在综合时工具就可以省去这部分逻辑电路，从而减小面积的浪费。

通过对部分积引入符号位 *s*，正值取 0，反之取 1，可将不必要的高位全部置 0，部分积无需再进行符号位扩展，从而简化硬件逻辑以优化面积，运算结构如图 4-10 所示。对于偶数位宽的无符号乘法，以 16 比特的无符号乘法为例，共生成九个部分积，不考虑符号位扩展的部分积位宽为 17 比特。为减小符号位扩展带来的资源浪费，对于第一个部分积，在高位添加序列{~*s*,*s*,*s*}，{}表示拼接符，而第八个部分积则高位补{~*s*}，最后一个部分积无需操作，剩余其他部分积均在高位添加序列{1,~*s*}。另外考虑到部分积可能为负数，对于这种情况，部分积按位取反，而求补码所需的加 1 操作则通过在下一个部分积低位处补上 *s* 实现。

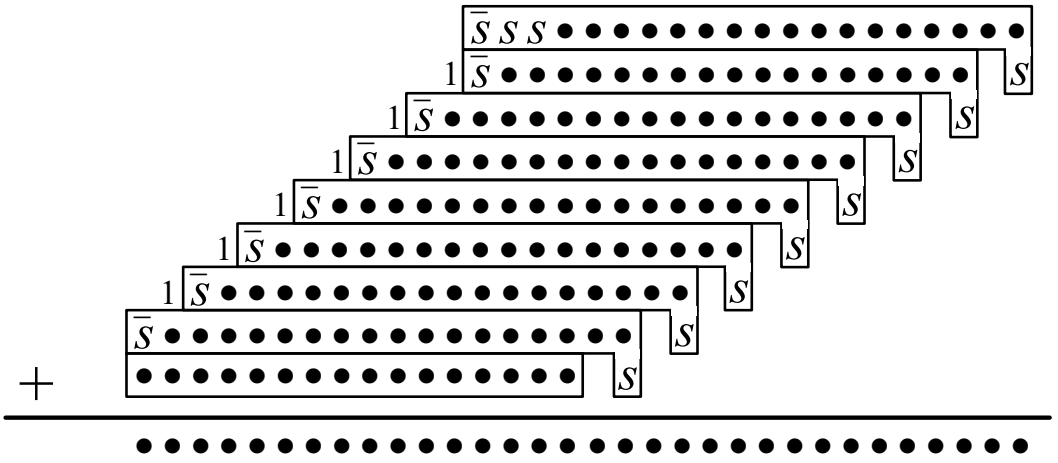

图 4-10 Booth 部分积符号位扩展优化[49]

而对于奇数位宽的无符号乘法，最后一个部分积则取两倍被乘数，且倒数第二个部分积高位应添加常量 1 和 *s* 的反值，其余操作与偶数位宽的无符号乘法相同。

4.3.3 尾数乘法的 SIMD 优化

本课题所设计的尾数乘法电路结构基于基 4-Booth 乘法算法，并通过合理的硬件资源分配实现尾数的向量乘法，其中关键在于对部分积的合理分配。

执行尾数的定点乘法 $a\times b$ 时，每一个部分积根据 $-2b_{i+1}+b_i+b_{i-1}$ 由表 3-3 从{−2,−1,0,1,2}

中选取一个系数与 a 相乘得到。为使生成的部分积可以被双精度、单精度和半精度浮点数的尾数向量乘法使用，a 的数据结构如图 4-11 所示，f_d、f_s 和 f_h 分别表示双精度、单精度和半精度浮点数的尾数。以单精度和半精度运算为例，这种数据结构使得在计算 f_{s1} 与 $-2b_{i+1}+b_i+b_{i-1}$ 相乘的结果时也得到了 f_{h2} 和 f_{h3} 与 $-2b_{i+1}+b_i+b_{i-1}$ 相乘的结果。且如 4.3.2 中所述，部分积的高位需要填充符号位相关的序列，采用高位对齐使得不同精度的部分积可以共享这些填充序列，从而简化控制逻辑以减小面积，因此将 f_{d0}、f_{s0} 和 f_{h0} 高位对齐，f_{s1} 和 f_{h2} 同理。

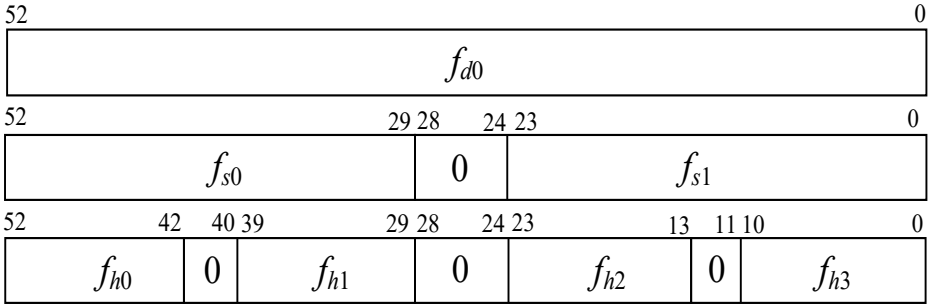

图 4-11 用于生成部分积的数据结构

同一精度下的不同尾数间隔一定数量的 0，一方面在尾数低位补 0 可以实现位宽匹配，另一方面可以保证在进行部分积累加时，向量运算下不同尾数对应的部分积互不干扰。基于这种数据结构，所有精度下的 Booth 乘法将共享一套部分积，便于硬件以更小面积实现向量乘法。

以实现一个 24 比特的无符号乘法为例，共生成 13 个部分积，为实现多种精度的向量运算，基于“硬件隔离”的思想，本课题设计的乘法电路可复用这些部分积以实现两个 11 比特的无符号乘法。图 4-12 展示了适用于向量运算的 Booth 部分积结构，为方便叙述，假设每个部分积高位均扩展{1,~s}。此处不考虑部分积低位处添加的 s 位，考虑到部分积可能是被乘数的两倍，最终 24 比特的无符号乘法生成的部分积位宽为 27 比特，使用图 4-12 的所有部分积资源。而复用其执行 11 比特的向量乘法时，对于第一个 11 比特的乘法，共生成 6 个 14 比特的部分积，使用了右上角的硬件资源，而另一个 11 比特乘法则使用了左下角的资源。图中深色方框代表了 11 比特乘法的部分积使用情况，在小位宽模式下，不需要的比特位均应该被置零，图中以白色填充。基于图 4-11 的数据结构，两个 11 比特乘法的部分积会间隔一定距离 d 而不会互相干扰，根据图 4-12 可以算得 d 为 3 比特，由于图 4-12 中假设所有部分积位宽相同，实际上对于第一个 11 比特的乘法，最后一个部分积的位宽应该为 12 比特，那么 d 的实际数值是要比 3 比特更大的。

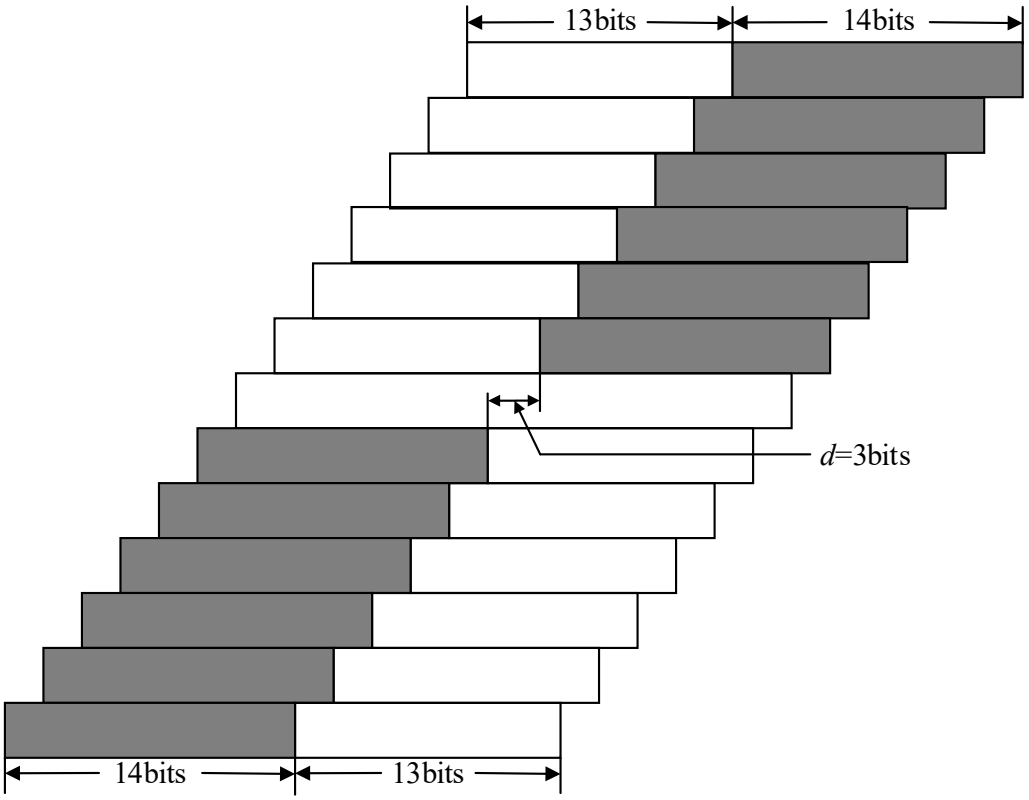

图 4-12 尾数乘法共享部分积示意图

通过以上设计，实现将双精度浮点运算时的尾数相乘的部分积阵列复用于两个单精度浮点运算的尾数相乘，或者四个半精度的尾数相乘，实现多种精度融合的尾数向量乘法运算结构，只使用一套硬件资源，以最小的面积实现不同精度下尾数乘法的向量运算。

4.3.4 基于 4-2CSA 的部分积累加

当得到部分积后，为缩减部分积的数量以加速累加，以 CSA 替代 CPA 以缩短时序路径。3.2.5 介绍了 3-2CSA，其可将三个输入项缩减为 *sum* 和 *carry*。如果只使用 3-2CSA 完成部分积缩减，网络如图 3-10 所示，需经过七级 3-2CSA 阵列，时序路径较长。注意到其中在第三级后部分积的个数为 4 的倍数，那么可以使用 4-2CSA 来减少 CSA 的级数，4-2CSA 可将四个输入项缩减为 *sum* 和 *carry*，进一步加速累加。

假设对四个加法项 x_i、y_i、z_i、w_i 进行缩减，且接收到来自上一个比特计算得到的进位值 c_i。4-2CSA 实际有五个输入和三个输出，它的实现方式多样，传统的设计会采用两个 3-2CSA 组合的办法，但这样会产生资源浪费，且路径延时为四个异或门，与直接使用 3-2CSA 无异。图 4-13 表示每计算 *sum* 和 *carry* 的一比特所需 4-2CSA 的硬件结构，目前应用中更多地会采用异或门和选择逻辑组合搭建的优化电路，实现更小的路径延时，且相比于传统的实现方式也更加省面积[50, 51]。

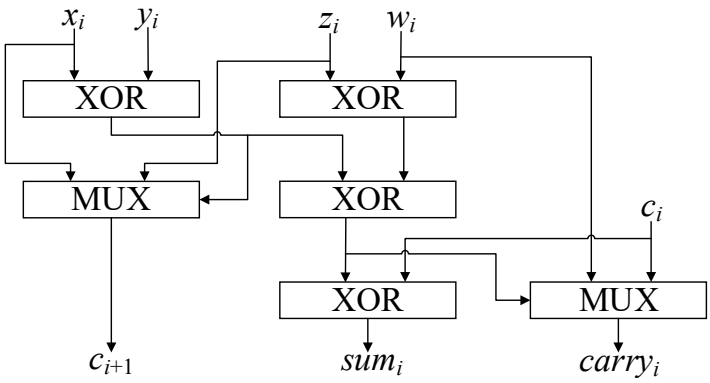

图 4-13 4-2CSA 结构图

不同于串行加法器，4-2CSA 的输出的 c_{i+1} 并不依赖于进位 c_i 而只由 x_i、y_i、z_i、w_i 计算得到，其延时不像串行加法器一样随着位宽的增加而增加。3-2CSA 的延时大致为两个异或门级联的延时，而本文所采用的 4-2CSA 最长的路径延时为三级异或门，注意选择逻辑的延时与异或门相同。基于以上讨论，将图 3-10 的最后四级 3-2CSA 阵列替换为两级 4-2CSA 阵列，从而把华莱士网络的总级数减少为五级，八个异或门延时被优化为六个，缩短时序路径以优化速度。

4.3.5 大位宽加法优化

要实现将华莱士网络输出的 *sum*、*carry* 和移位好的尾数 f_c 相加，需要使用 161 比特位宽的大位宽加法器，考虑符号位和 *GRS* 则需要 165 位的加法器。这会导致一定程度的面积浪费，基于这个问题对其进行面积优化。

首先考虑附加位 *GRS*，当执行加法时，结果的最低三个比特即为 *GRS* 值；当执行减法时，由于 f_c 移位前已进行按位取反，则取 *GRS*+1。而 *GRS* 运算产生向 161 比特加法进位的情况只有一种，此时执行减法且 *G*、*R* 和 *S* 均为 1。由于图 4-14 输出的 *carry* 的最低位一定是 0，因此可将这个进位置于 *carry* 的最低位进行处理。当 *GRS* 满足进位条件时，*carry* 的最低位填入 1，而 *GRS* 根据此时执行加法或是减法控制是否需要加 1。通过引入以上附加位逻辑，加法器位宽可减小 3 比特。

以双精度操作为例，考虑 165 比特加法，加法器的结构如图 4-14 所示。基于图 3-5 和图 3-6 可以发现，对于高 56 比特的加法，f_c 并不与乘法器输出的 *sum* 和 *carry* 重合，高位 56 比特加法的结果只与 f_c 和来自低 106 比特（不包括 *GRS*）加法的进位有关。对 f_c 的低 106 比特、*sum* 和 *carry* 完成加法，为了保存向高 56 比特加法输出的进位信号，需使用 107 比特位宽的加法器，包括一级进位保存加法器和一级二输入的进位传播加法器，然后利用进位信号去选择 f_c 的高 56 比特是否需要加 1。如果其为 1，则选择 f_c 的高

56 比特加 1 后的结果，否则选择不加 1 的结果。加 1 电路硬件上通过级联的半加器实现，这相比于 CSA 和 CPA 的面积要小得多，从而优化面积。且通过引入选择逻辑，使高 56 比特加法与低 106 比特加法可以并行计算，从而将 56 比特加法从关键路径上移除，大大缩短时序路径，速度得以优化。

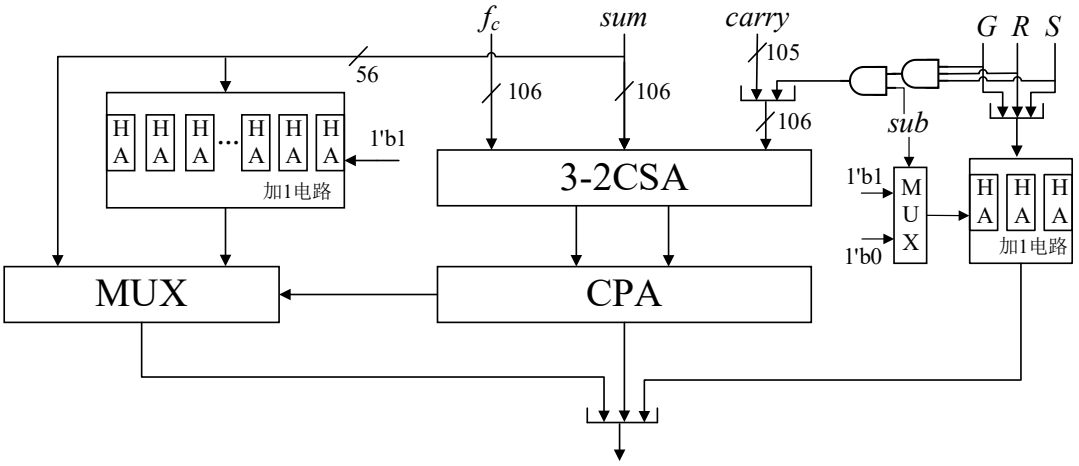

图 4-14 改进的大位宽加法器

为减少硬件开销以减小面积，本课题在只使用这一个大位宽加法器的条件下完成向量运算。在双精度运算时，与 *sum* 和 *carry* 有关的加法操作置于 107 比特的两级加法器完成，而单精度操作应包括两个 49 比特的加法，对于半精度则应该计算四个 23 比特加法，并将这些小位宽加法的最高位值作为选择信号保存下来。而高位 56 比特的加 1 电路也需进行向量化实现，单精度操作时执行两个 27 比特的加 1 操作，对于半精度则执行四个 14 比特的加 1 操作。其向量化实现与 4.2.3 类似，将加法单元拆分为四个更小位宽的加法单元，但将 CAS 替换为半加器，根据当前运算精度来选择半加器的输入来自前级半加器或者常量 1，最后利用低 107 比特加法输出的选择信号来控制输出加 1 或者不加 1 的结果。

4.3.6 负结果修正优化

当执行有效减法时，多数情况下所引入的加 1 可合并到 CSA 输出的 *carry* 的末位，而作为输入 f_c 只需要按位取反，从而省去大位宽加 1 电路的使用。而还有一种情况，即当有效减法的结果是负数值时，需把结果取补码以得到原码，此时仍只执行按位取反，但加 1 无法再合并到 CSA 进行处理，需要在后续通路中加 1 以进行修正，这种情况称为负结果修正。

负结果修正在规格化级完成，此时有效尾数取为隐藏位加上 IEEE 754 所规定的尾

数位。负结果修正和舍入不会同时对有效尾数加 1，该论述可从三个方面证明：（1）如果在规格化中，由于尾数位宽的限制所丢弃的低位均为 0，那么此时负结果修正所引入的末位加 1 操作的效果是使 *GRS* 的 *S* 位从 0 变成 1，不会影响到有效尾数；（2）如果规格化中所丢弃的低位均为 1，那么末位加 1 将导致这些位全部变成 0，而且向有效尾数的最低位进位，*GRS* 变为 0；（3）丢弃的低位不全为 0，这种情况下末位加 1 可能影响 *GRS*，但不会影响到有效尾数，只有通过舍入对有效尾数加 1。

综上负结果修正可能影响 *GRS*，但不会出现与舍入向有效尾数加两次 1 的情况，可与舍入合并处理。在规格化前，对负结果进行按位取反，且负结果在左移时低位应补 1 而不是 0。左移结束后对 *S* 进行修正，如果此时 *S* 和比 *S* 更低的位均为 1，可认为 *S* 的实际取值为 0，但通过这种方式求 *S* 最长会在数据通路上引入 109 个与门级联的延时。为移除这部分延时，采用一种快速附加位算法，利用前导 0 检测和后导 0 检测（Trailing Zero Detector，TZD）计算 *S*。对图 4-14 的加法电路输出的结果进行 TZD，假设得到的后导 0 数量为 tzd_f。如果加法结果为负数则对尾数按位取反，反之无需操作，之后进行 LZD，得到前导 0 数量为 lzd_{invf}。假设有效尾数的位宽为 *width*，那么 *S* 位的实际取值由公式(4-3)求得：

$$S = \left(lzd_{invf} + tzd_f\right) < \left(2 \times width + 3\right) \tag{4-3}$$

小于关系成立代表在规格化左移修正后 *S* 及 *S* 后存在某一位为 1，那么 *S* 取值为 1。由于该算法与规格化左移并行，相比于在左移后再通过级联的与门计算 *S*，该快速算法不会在数据通路引入新的延时从而能更快获取 *S* 的修正值。如果图 4-14 求得负结果且 *S* 修正值为 0，此时 *S* 应向 *R* 进 1，如果此时 *R* 为 1，则将 *R* 修正为 0，反之 *R* 为 0 则修正为 1。如果为正结果或 *S* 修正值非 0，则仍取规格化移位后的 *R* 值，*G* 位的求法同理。

当有效减法的结果为负值且求得 *GRS* 修正值为 0，则应向有效尾数加 1，产生的效果和舍入加 1 是一样的，由此在舍入级选择输出尾数加 1 或尾数不加 1 的结果。基于以上讨论，当有效减法求得负值时，只需对结果按位取反而无需加 1，*GRS* 另外通过快速附加位算法求出并判断负结果修正是否会引起有效尾数加 1，将该加 1 与舍入合并处理，从而减少加 1 电路的数量，这对于面积和速度都是有积极影响的，从而优化面积效率。

4.4 浮点除法与开平方模块的设计与优化

4.4.1 浮点除法与开平方模块的整体架构

浮点除法与开平方是影响数字计算机性能和信号处理速度的关键算术运算[52]。本文

的浮点除法与开平方模块采用五级流水线实现，结构如图 4-15 所示，虚线表示流水线寄存器。在该模块中，区别于前文其他浮点操作在不同精度下执行周期相同，浮点除法和浮点开平方操作的执行周期和吞吐率受精度的影响，本节对该模块进行阐述。

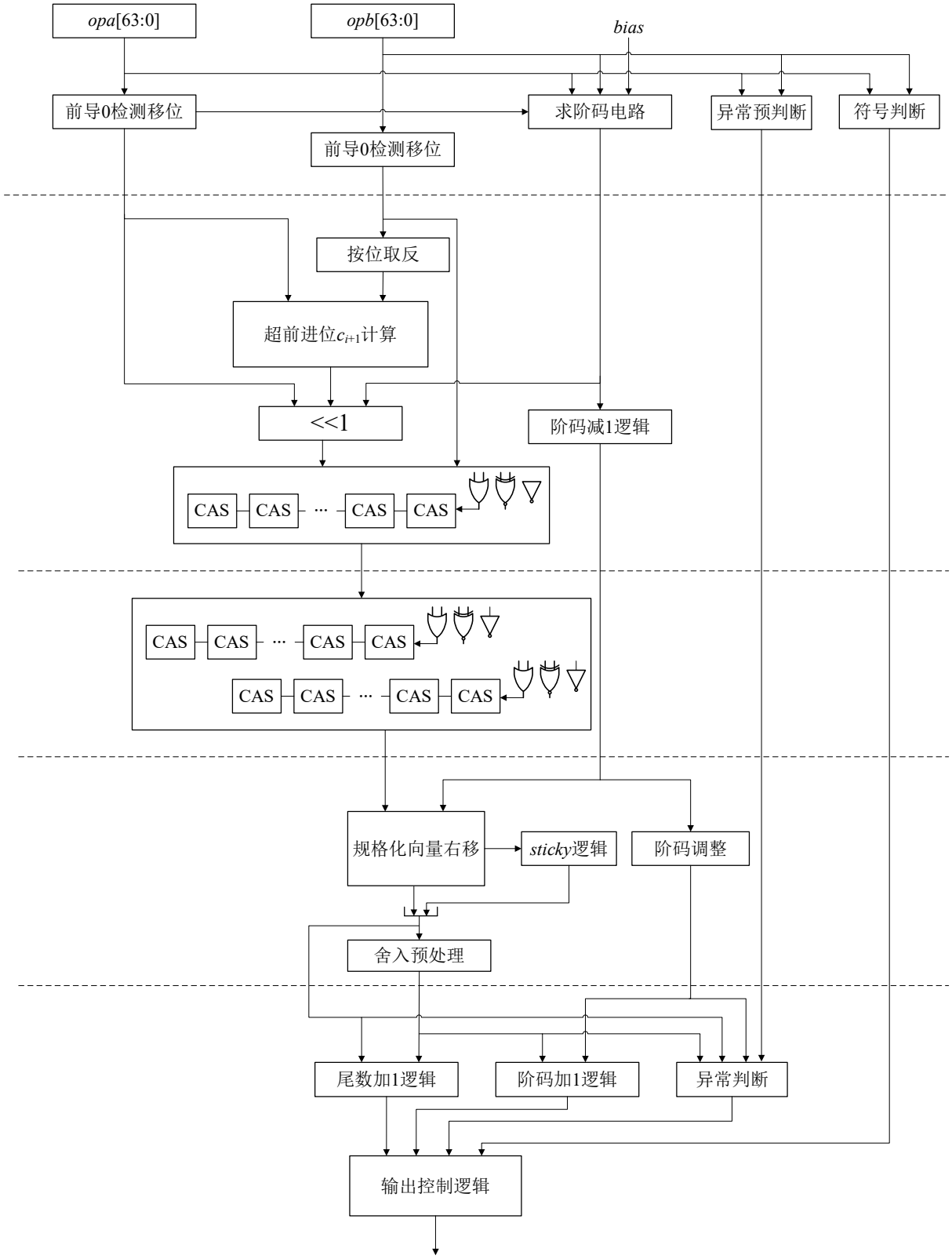

图 4-15 浮点除法与开平方模块整体架构

第一级根据所用的尾数除法和开平方算法对数据进行预处理，本课题采用不恢复余数算法。首先对尾数进行前导 0 检测，实现方法见式(3-9)和式(3-10)，采用前导 0 边检测边移位的方法。计算出尾数中第一个 1 的位置，并将这个 1 左移到最高位，使得尾数的隐藏位为 1，实际上即对尾数进行了规格化。这样做可直接确定尾数进行除法或开平方后所得到的结果的小数点的位置，由于规格化后的尾数均位于区间[1,2)，那么当执行除法时，结果位于区间(0.5,2)，而执行开方时位于区间[1,2)，从而确定不恢复余数算法阵列第一级最高位 CAS 单元计算得到的 c_{i+1} 即为结果的隐藏位，而小数点位于隐藏位的后面，之后各级最高位 CAS 单元依次求出结果的小数部分各个位。如果没有对尾数进行预移位，将难以判断结果的小数点的位置，对尾数进行预处理是有必要的。对于浮点除法和开平方的求阶逻辑本课题将使用同一套硬件资源实现以减少开销，具体逻辑在 4.4.2 中进行详述。结果浮点数的符号位也在该级得到，浮点除法取为两个源浮点数符号位的异或，开平方则取 0，因为结果一定为正数，并进行异常预判断。

第二级对前级预处理好的尾数进行细调，假设预处理后的尾数分别为 f_a 和 f_b，判断是否需要将 f_a 再左移一位。如前所述，除法后的结果位于(0.5,2)，这会导致在规格化时应判断是否需要左移一位，而一般更希望只执行右移（此时只需关心临时阶码是否小于等于 0），从而简化逻辑以优化面积。在实现上通过 f_a-f_b 的符号位来判断尾数的大小关系，当 $f_a<f_b$ 时，结果将位于区间[0.5,1)，那么将 f_a 左移一个比特后再进行尾数除法，反之无需调整。这样处理后商一定位于区间[1,2)，从而保证了规格化前尾数的隐藏位一定为 1。只需要通过符号位即可判断 f_a 和 f_b 的大小关系，利用 f_a 和 f_b 的各比特 f_{ai} 和 f_{bi}，通过超前进位逻辑可算得最高位输出的进位值。如果其为 0，那么 f_a-f_b 的符号位为 1，此时 f_a 小于 f_b，反之则求得 f_a 大于等于 f_b。每比特进位值 c_{i+1} 由公式(4-4)计算：

$$c_{i+1} = f_{ai}\overline{f_{bi}} + c_i\left(f_{ai} + \overline{f_{bi}}\right) \tag{4-4}$$

f_a 减去 f_b 相当于计算 f_a 加上 f_b 按位取反后的值再加 1，硬件实现上对 f_{bi} 取反，而 c_0 取为 1。对于浮点数开平方，为使被开平方尾数的指数为偶数，如果是规格化数开方，对于阶码是偶数的情况，需要在第二级对移位后的尾数也左移一个比特；对于非规格化数开方，如果第一级所得到的前导 0 的数量为奇数，也做相应的左移处理。且为了减少流水线第三级的迭代周期数，在该级会完成 CAS 阵列第一级的运算，得到第一个余数后传至流水线第三级。

第三级执行尾数除法和开平方，由于不恢复余数算法阵列的资源消耗是非常大的，对于双精度浮点数来说，不恢复余数算法阵列的级数有 54 级，如此长的组合逻辑路径

会大大降低模块可正常工作的最高频率，因此采用迭代的方式实现这个阵列，硬件上使用 CAS 阵列的二级结构。通过迭代的方式，将阵列输出与输入相连，最大程度地对 CAS 进行复用，每一次迭代完成 CAS 阵列的两级运算。由于不同精度的尾数除法或开平方的结果位宽不同，而每迭代一个周期产生两比特的结果，这也是导致不同精度的浮点除法和浮点开平方指令执行周期不同的根本原因。

第四级完成尾数的规格化操作，由于对尾数进行了预处理，此时尾数的最高位一定是 1，无需再考虑左移。区别于浮点数加减法，该模块使用向量右移器，当前级所求得的临时阶码 e_{tmp} 为负值或者 0 时，需要对尾数进行右移操作，右移位数 *rshiftnum* 可由公式(4-5)表示：

$$rshiftnum=\begin{cases} e_{tmp}-1, & e_{tmp}\le 0 \\ 0, & e_{tmp}>0 \end{cases} \tag{4-5}$$

其中 e_{tmp} 为前级算得的临时阶码，右移器的结构与图 4-8 类似，但低位每有一个比特被右移出时，需要使用 *sticky* 逻辑将被移出的这个比特与当前的最低位进行或运算后存入 *GRS* 的最低位 *S*，而不像左移器将移出的位丢弃。

第五级依照RISC-V协议完成舍入和异常处理，判断尾数和阶码是否执行加1操作，实现方法与前述的模块类似，在此不过多赘述。

对于浮点除法 *opa*/*opb*，需考虑以下情况：（1）只要 *opa* 不为∞也不为 NaN，那么当 *opb* 为∞时，结果一定为 0；（2）只要 *opa* 不为 0 且不为 NaN，那么当 *opb* 为 0 时，结果一定是∞；（3）0/0 和∞/∞均为无效操作，结果为 qNaN；（4）只要有一个操作数为 NaN，结果即为 qNaN。其中（3）会将无效操作异常指示信号拉高，而（4）只有在源浮点数有任意一个为 sNaN 的情况下才会将其拉高。

对于浮点开平方运算，需要考虑以下情况：（1）+∞开平方的结果也为+∞；（2）对负数进行开平方是无效操作，结果为 qNaN；（3）对 NaN 进行开平方结果为 qNaN。其中（2）拉高无效操作异常指示信号，（3）只有在源浮点数为 sNaN 时将其拉高。

4.4.2 求阶码电路的设计与优化

在流水线前两级对尾数进行了预处理，阶码也应做相应的处理，总结浮点数除法和开平方以及不恢复余数算法的规律，将不同浮点操作的求阶码电路使用同一套硬件资源实现，用最少的加法器来实现多种操作的阶码求值。

以双精度浮点数为例，其他精度下的运算也是类似的。在进行浮点数除法时，假设被除数的阶码为 e_a，尾数为 f_a，除数的阶码为 e_b，尾数为 f_b，且在流水线第一级得到 f_a

和 f_b 的前导 0 数量分别为 z_a 和 z_b，那么临时阶码 e_2 可由公式(4-6)和公式(4-7)得到，其中需要产生中间值 e_1：

$$e_1 = \{2'b0, e_a\} - \{2'b0, e_b\} + bias \tag{4-6}$$

$$e_2 = e_1 - z_a + z_b \tag{4-7}$$

其中对于双精度浮点数，*bias* 为 1023，考虑到加法可能产生一个比特的溢出，且减法可能产生负数结果，因此阶码的高位需补两个 0。可以发现，如果使用二输入的进位传播加法器 CPA 共需要四个。进位传播加法器是一种非常消耗硬件资源的逻辑，为减小面积在进行浮点数开平方时复用这些加法器。

对于浮点数开平方，考虑规格化数开平方，假设被开方数阶码为 e_a，尾数为 f_a。若阶码为奇数，减去偏置值后实际指数 e_a-1023 是偶数，可直接开平方；而如果阶码是偶数，则实际指数是奇数，将尾数左移一个比特，阶码相应地减去 1。综上得到临时阶码 e_2 的求解可由公式(4-8)表示：

$$e_2 = \begin{cases} \dfrac{e_a + 1}{2} + \dfrac{bias - 1}{2}, & e_a\text{为奇数} \\ \dfrac{e_a}{2} + \dfrac{bias - 1}{2}, & e_a\text{为偶数} \end{cases} \tag{4-8}$$

总结 e_a 为奇数和偶数两种情况，为便于硬件实现，统一由公式(4-9)表示：

$$e_2 = e_a >> 1 + e_a \%2 + \frac{bias - 1}{2} \tag{4-9}$$

可以发现，对于规格化数开平方，需要两个二输入的进位传播加法器，而且将求阶通过上式计算在硬件上是很容易实现的，对于双精度浮点数，e_a 是一个 11 比特的数值，上式在硬件上可由公式(4-10)表示：

$$e_2 = e_a[10:1] + e_a[0] + \frac{bias - 1}{2} \tag{4-10}$$

e_a 右移出去的最低一个比特不丢弃，而是加回去，从而实现阶码为偶数和奇数两种情况求阶电路的统一。规格化尾数的前导 0 数量 z_a 为 0，而对于非规格化数则为非 0，阶码计算时需要引入 z_a，且在预处理时需要将尾数进行左移以使隐藏位为 1。类似地，求阶逻辑映射到硬件上可由公式(4-11)表示：

$$e_2 = -z_a[10:1] - z_a[0] + e_a[0] + \frac{bias - 1}{2} \tag{4-11}$$

在浮点系统中，非规格化数阶码的 *bias* 比规格化数小 1，本课题在进行硬件设计时将它们的 *bias* 统一，将非规格化数的阶码以 1 进行处理，即 e_a[0]为 1。通过以上对浮点

数除法和浮点数开平方的归纳，将它们的求阶电路统一成使用两个三输入的加法器来实现，将以上的求阶公式统一为由公式(4-12)和公式(4-13)表示：

$$e_1 = Mux(e_a[10:0], e_a[10:1]) + Mux(e_a[0], \overline{e_b[10:0]}) + Mux\left(bias, \frac{bias-1}{2}\right) \quad (4\text{-}12)$$

$$e_2 = e_1 + \overline{Mux(z_a[10:0], z_a[10:1])} + Mux(z_b, 0) \quad (4\text{-}13)$$

其中 *Mux* 表示选择逻辑，根据执行的运算从括号内选出所需的项，此处未考虑阶码高位添加的两比特 0，最终求得的阶码为 e_2。e_1 的计算与 LZD 并行且通常快于 LZD，与前导 0 数量 *z* 无关的项均可于 e_1 进行计算，从而简化 e_2 的选择逻辑，缩短时序路径。且利用进位保存加法器 CSA 以减少加法器的输入项，由公式(3-16)和公式(3-17)可知，CSA 同样可以处理如 *A+B-C* 的运算，对负值项进行按位取反即可，并可省去减法运算时求补码所需的加 1 电路。对于三输入加法以一级 3-2CSA 和一级二输入的 CPA 实现，从而减少 CPA 的使用。CSA 各个比特是独立计算的，相比 CPA 速度更快。注意到上式并没有包含 $z_a[0]$，如果 CSA 执行 *A+B-C*，那么 *carry* 的最低位取 1，考虑到应减去 $z_a[0]$，只需要在进行开平方操作时，如果 $z_a[0]$为 1 时，则将 *carry* 减 1，即 *carry* 的最低位取 0。

通过本文的方法将浮点除法和开平方的求阶电路统一，以较小的面积实现快速的求阶码电路。

4.4.3 尾数除法与开平方的 SIMD 优化

如第三章所述，与其他算法对比，不恢复余数算法具有占用资源少和处理速度较快的优点，且尾数的除法和开平方运算可复用同一个 CAS 阵列，本课题采用该算法进行尾数除法和开平方模块的设计。

结果寄存器变量 reg_q 的位宽取决于结果的位宽，考虑 *GRS*，为了减少迭代的次数，只用 *GS* 位来完成舍入。模块迭代时只需迭代至 *G* 位，当求出 *G* 位后剩余余数非零，则代表此时存在冗余数据，*S* 位取为 1。对于双精度浮点数，加上隐藏位，结果为 55 比特。如 4.1 所述，源浮点数的尾数经过预处理后，尾数除法与开平方结果的最高位一定为 1，因此 reg_q 只需保存 54 比特的结果，单精度结果仍保存为 26 比特。半精度情况下保存三个 13 比特结果，另外的结果则保存为 12 比特，具体原理与双精度情况类似。其数据结构如图 4-16 所示，f_d、f_s 和 f_h 分别表示双精度、单精度和半精度浮点数的尾数。f_{s0} 与 f_{h1} 低位对齐，f_{d0}、f_{s1} 和 f_{h3} 低位对齐，这样存储便于后续进行规格化右移。且 CAS 在运算时需要从 reg_q 对应位置获取 *p* 值以进行加减控制，这种数据结构使得在不同精度下同一个 CAS 单元在计算时可以使用 reg_q 同一位置的数据作为 *p* 进行计算，简化控制逻辑。

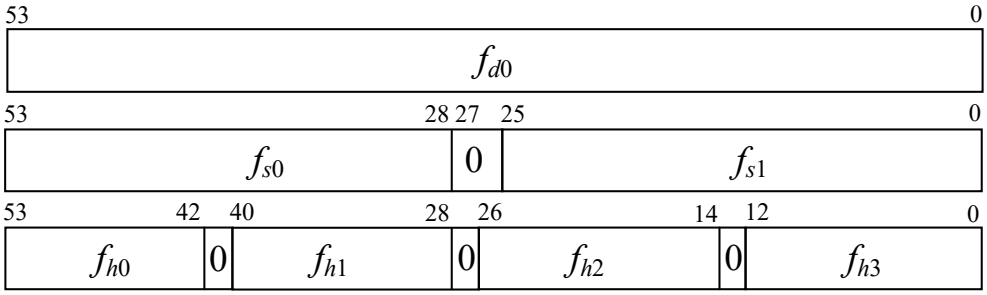

图 4-16 reg_q 数据结构

注意到图 4-16 中单精度情况下相邻尾数间隔一个 0，实际上每个尾数的最高位只有在最后一轮迭代结束后才会被移入 1，而在迭代过程中始终为 0，因此也可视为相邻尾数间间隔两个 0。这使得 CAS 阵列在计算尾数开平方时，两个 0 的位置可以用以计算由公式(3-25)表示的最低两个比特逻辑。且在不同精度下，同一位置的 CAS 单元将使用 reg_q 的同一比特数据，减少选择逻辑的使用以减小面积。

尾数寄存器变量 reg_f 在执行除法时存储除数的值，且在迭代过程中保持不变；执行开平方时则存储被开平方数，且由于单周期完成 CAS 阵列的两级运算，每周期应左移四个比特。对于双精度运算，执行除法时，除数在最高位补上符号位 0 后位宽为 54 比特；执行开平方时，考虑流水线第二级在细调时可能将被开平方数左移一个比特，被开方数位宽也为 54 比特。对于单精度则存储两个 25 比特的数，为匹配位宽在低位补 0，对于除法和开平方操作，在低位补 0 并不会影响结果，同理对于半精度则存储四个 12 比特的数。由此确定 reg_f 位宽为 54 比特，数据结构与图 4-11 类似，采用高位对齐。

对于余数寄存器变量 reg_r，由于 CAS 阵列执行开平方时所需的余数位宽与结果位宽相同，那么 reg_r 应取 55 比特。但由于在迭代过程中结果的最高位为常量 0，该位并不影响 CAS 阵列的计算结果，因此 reg_r 可取为 54 比特，位宽仍与 reg_q 相同。而执行除法时当级使用上级 CAS 输出余数的低 53 比特数据并在低位补一个 0，综上 reg_r 的位宽确定为 54 比特。

基于不恢复余数算法，尾数除法与开平方模块的核心为 CAS 阵列，且每一级的 CAS 单元数量为 54 个。为满足向量运算的需求，基于“硬件隔离”的方法，本课题对阵列的 CAS 单元进行合理的结构划分。为方便阐述，此处以位宽为 10 比特的 CAS 阵列为例进行阐述，该结构可完成一个 10 比特的尾数除法或开平方，也可实现两个 4 比特的尾数除法或开平方，注意此处的尾数已经过流水线第一级和第二级的预处理。

假设对 10 比特尾数 $h_n=h_{n9}h_{n8}h_{n7}h_{n6}h_{n5}h_{n4}h_{n3}h_{n2}h_{n1}h_{n0}$ 开平方，图 4-17 展示了对 10 比特尾数开平方时 CAS 阵列的的运算过程，假设该级执行加法，分别从 reg_r 和 reg_q 取得

相应数据。最低两个比特取{1,1}与 h_{n9} 和 h_{n8} 相加，如果执行减法，则应取{0,1}与 h_{n9} 和 h_{n8} 相减，这两个比特加法不通过 CAS 单元实现，而通过 3.4.2 的优化方法进行实现，具体实现见式(3-26)、式(3-27)和式(3-28)，每次迭代后将 h_n 左移两个比特。图 4-18 则展示了实现两个 4 比特尾数开平方时的运算过程，假设对两个 4 比特尾数 $h_n=h_{n3}h_{n2}h_{n1}h_{n0}$ 和 $h_m=h_{m3}h_{m2}h_{m1}h_{m0}$ 开平方，h_m 的开平方运算通过高 6 个 CAS 单元实现，而 h_n 则使用低 4 个 CAS 单元，最低两个比特加法另外计算，每次迭代后将 h_m 和 h_n 左移两个比特。

	reg_r[9]	reg_r[8]	reg_r[7]	reg_r[6]	reg_r[5]	reg_r[4]	reg_r[3]	reg_r[2]	reg_r[1]	reg_r[0]	h_{n9}	h_{n8}
+	reg_q[9]	reg_q[8]	reg_q[7]	reg_q[6]	reg_q[5]	reg_q[4]	reg_q[3]	reg_q[2]	reg_q[1]	reg_q[0]	1	1

图 4-17 执行 10 比特尾数开平方的 CAS 输入

	reg_r[9]	reg_r[8]	reg_r[7]	reg_r[6]	h_{m3}	h_{m2}	reg_r[3]	reg_r[2]	reg_r[1]	reg_r[0]	h_{n3}	h_{n2}
+	reg_q[9]	reg_q[8]	reg_q[7]	reg_q[6]	1	1	reg_q[3]	reg_q[2]	reg_q[1]	reg_q[0]	1	1

图 4-18 执行两个 4 比特尾数开平方的 CAS 输入

图 4-19 展示了执行 10 比特尾数除法时的 CAS 阵列的运算过程，假设 10 比特被除数为 $h_n=h_{n9}h_{n8}h_{n7}h_{n6}h_{n5}h_{n4}h_{n3}h_{n2}h_{n1}h_{n0}$，除数为 $h'_n=h'_{n9}h'_{n8}h'_{n7}h'_{n6}h'_{n5}h'_{n4}h'_{n3}h'_{n2}h'_{n1}h'_{n0}$，运算时需要使用上一级 CAS 阵列保存的 reg_r，第一级 CAS 阵列则使用 h_n 替代 reg_r 的数据，最低两比特加法“x”表示执行除法时无需使用。图 4-20 则展示了向量运算两个 4 比特尾数除法时的运算过程，假设两个被除数分别为 $h_n=h_{n3}h_{n2}h_{n1}h_{n0}$ 和 $h_m=h_{m3}h_mh_{m1}h_{m0}$，除数分别是 $h'_n=h'_{n3}h'_{n2}h'_{n1}h'_{n0}$ 和 $h'_m=h'_{m3}h'_mh'_{m1}h'_{m0}$。为和开平方匹配，$h_m$ 和 h'_m 的除法使用 CAS 阵列的高 6 个 CAS 单元实现，基于图 4-16 的数据结构，低位补两个 0，这不会影响最终的结果。h_n 和 h'_n 相除则使用低 4 个 CAS 单元实现，最低两比特加法无需使用。

	reg_r[9]	reg_r[8]	reg_r[7]	reg_r[6]	reg_r[5]	reg_r[4]	reg_r[3]	reg_r[2]	reg_r[1]	reg_r[0]	x	x
+	h'_{n9}	h'_{n8}	h'_{n7}	h'_{n6}	h'_{n5}	h'_{n4}	h'_{n3}	h'_{n2}	h'_{n1}	h'_{n0}	x	x

图 4-19 执行 10 比特尾数除法的 CAS 输入

	reg_r[9]	reg_r[8]	reg_r[7]	reg_r[6]	reg_r[5]	reg_r[4]	reg_r[3]	reg_r[2]	reg_r[1]	reg_r[0]	x	x
+	h'_{m3}	h'_{m2}	h'_{m1}	h'_{m0}	0	0	h'_{n3}	h'_{n2}	h'_{n1}	h'_{n0}	x	x

图 4-20 执行两个 4 比特尾数除法的 CAS 输入

基于以上讨论，CAS 阵列单级有 54 个 CAS 单元，而开平方所需的最低两比特加法通过逻辑门实现，最终尾数除法与开平方阵列的单级结构如图 4-21 所示。x 和 y 的具体数值根据所执行的操作和精度进行选择，本课题使用的阵列由两级图 4-21 的结构组成。在迭代时向量运算会把每组阵列最高位 CAS 输出的 c_{i+1} 保存到 reg_q 中，第一级 CAS 单

元进行加减控制所需的 p 值从 reg_q 对应位置获取即可，第二级 CAS 单元的 p 则从第一级 CAS 单元输出的 c_{i+1} 获取。CAS 单元的输入进位 c_i 根据当前执行的精度选择输入 p 值或是上一个 CAS 单元输出的 c_{i+1}。

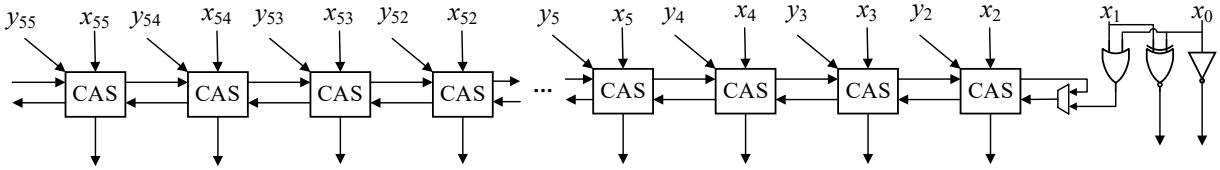

图 4-21 单级 CAS 阵列结构图

通过本节方法，实现多种精度融合的尾数向量除法与开平方硬件结构，本课题采用的结构在减小面积上有很大的优势，只使用一套 CAS 资源实现半精度、单精度和双精度浮点数的尾数向量除法和开平方运算。

4.4.4 流水线阻塞

尾数除法与开平方模块的 CAS 阵列需要使用两级图 4-21 的结构进行多周期迭代才能得到最终结果，在此期间输出 *busy* 为 1 以指示无法对流水线第二级的数据进行接收，如果此时有新的待处理数据将产生资源冲突，应对流水线进行阻塞。浮点除法与开平方模块顶层配置 *valid* 和 *ready*，结合尾数除法模块输出的 *busy* 来控制流水线阻塞。在与图 4-1 的输入控制逻辑相连的一端，*valid* 作为输入指示外部传入的源浮点数是否有效，而 *ready* 作为输出指示浮点除法与开平方模块能否接收该源浮点数，定义为 $valid_0$ 和 $ready_0$。在与输出控制逻辑相连的一端，*valid* 则作为输出指示运算器输出的结果浮点数是否有效，而 *ready* 作为输入指示外部模块能否接收该结果，定义为 $valid_1$ 和 $ready_1$。

当 $valid_0$ 和 $ready_0$ 均为高电平时，代表 $valid_0$ 和 $ready_0$ 握手成功，此时浮点除法与开平方模块成功接收外部传入的源浮点数，并将数据随着流水线往下一级传播。如果尾数除法与开方模块输出的 *busy* 为 1，第二级将会被阻塞。而当第二级无有效数据或 *busy* 为 0 时，第一级仍可接收新的浮点数，那么 $ready_0$ 作为输出信号会被置位高电平，反之则将前两级进行阻塞，并拉低 $ready_0$。

运算器每完成一次浮点除法或开平方运算便拉高 $valid_1$ 并输出有效结果，如果此时 $ready_1$ 为高电平，那么 $valid_1$ 和 $ready_1$ 握手成功，浮点除法与开平方模块输出的结果可被外部接收。当 $ready_1$ 为 1，或 $valid_1$ 为 0 时，后两级不会受到阻塞，流水线保持正向传递。如果此时尾数除法与开平方模块已计算出有效结果，则传递结果有效信号至第四级，反之第三级与第四级间将保持阻塞。

通过这种方式，最大程度地减少阻塞所占用的周期数以保持尽量高的吞吐率。吞吐率即单位时间内流水线所能执行完成并输出结果的指令条数，假设执行 n 次浮点除法或开平方操作，尾数除法模块所迭代的周期数为 T，那么本课题所设计的浮点数除法与开平方模块的指令吞吐率 $rate$ 可由公式(4-14)表示：

$$rate = \frac{n}{T + 4 + (n-1) \times T} \tag{4-14}$$

当 n 趋于无穷大时，不同精度的浮点除法或浮点开平方运算所需的指令周期数和吞吐率如表 4-1 所示，其中吞吐率包括非向量模式和向量模式，同样时间向量模式可以并行处理更多的数据。

表 4-1 浮点除法与开平方运算吞吐率表

精度	指令周期数	非向量模式吞吐率	向量模式吞吐率
半精度	10	1/6	2/3
单精度	16	1/12	1/6
双精度	31	1/27	1/27

4.5 本章小结

本章对硬件设计部分进行阐述，首先介绍运算器的整体架构，之后基于第二、三章理论，着重对各个浮点运算模块进行设计。研究重点在于提升浮点运算模块的速度和减小面积，通过对运算模块合理规划流水线以提高工作的最高频率，并对运算逻辑进行优化，包括移位逻辑、负结果修正逻辑和求阶逻辑等的优化。为满足向量运算的需求对各尾数处理模块进行 SIMD 优化，包括尾数加减法模块、尾数乘法模块和尾数除法与开平方模块，以及尾数移位模块。基于“硬件隔离”的思想实现多种精度融合的尾数向量运算结构，只使用一套硬件资源来实现向量化运算以减小面积。

第五章 系统仿真与验证

本章对本课题所编写好的设计代码进行测试，其中包括功能验证、逻辑综合、形式验证和 FPGA 板级验证等过程，确保所设计的浮点运算器满足设计目标。

5.1 浮点运算器的功能验证

5.1.1 验证平台设计

为了保证本课题所设计的模块符合浮点协议规范，本课题对所设计的浮点运算器进行仿真以完成功能验证。本文基于 System Verilog 语言搭建验证平台（testbench），基于软硬协同的验证思想，实现的浮点运算器作为待测试硬件模块放于验证平台中，软件部分则均由 C 语言进行编写。

验证平台环境如图 5-1 所示，设定随机数种子，通过随机数生成为系统提供测试激励，激励信号输入到待测运算器和 C 语言编写的 RISC-V 浮点算法模型中，其中 C 算法模型已经过充分验证，在 testbench 中作为标准参考模型使用。软件从待测运算器获得运算结果和异常指示信号后与 C 算法模型得到的结果进行比较。如果比较不通过则将待测运算器和 C 算法模型的输出均打印出来以便进行对比调试，*error_test_num* 记录运算错误的测试激励数量。每完成一个激励的测试便将 *finished_test_num* 加 1，直至已测试激励的数量达到设定的最大值或者运算错误的次数达到最大值时仿真停止。

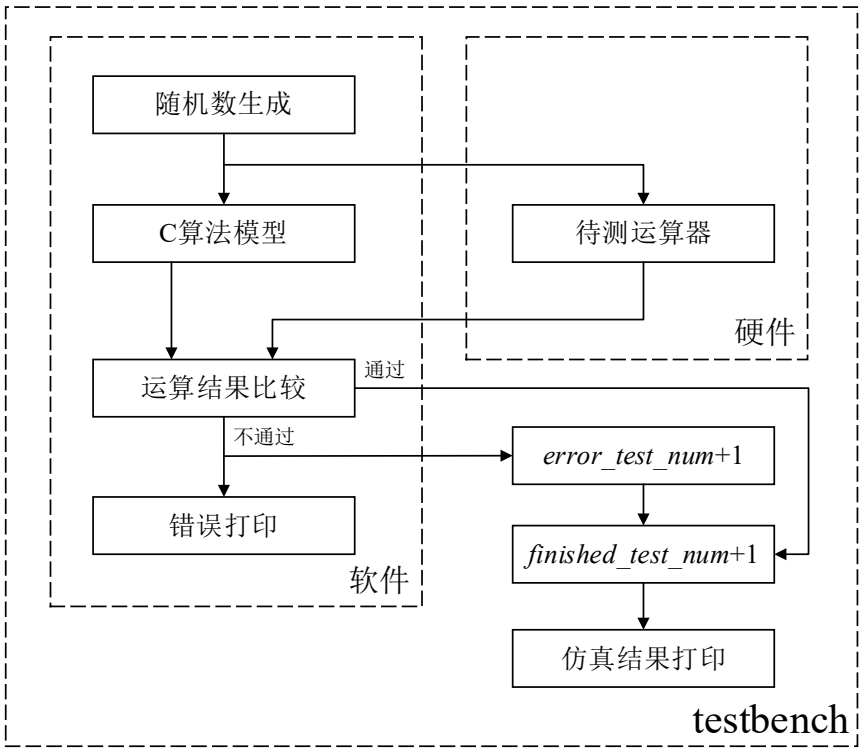

图 5-1 验证环境框图

5.1.2 浮点运算功能验证结果

为加快浮点运算测试流程，本设计分模块进行功能验证，基于图 5-1 设计的测试平台环境，对浮点加减法与格式转换运算、浮点乘加运算和浮点除法与开平方运算进行了大规模的功能验证。验证过程使用 ChiselStage 类的 execute 方法将设计好的 Chisel 代码转换为 Verilog 文件，之后利用 VCS 工具完成编译仿真工作，此过程会生成仿真波形文件，如果出现错误情况，使用 Verdi 工具查看波形以定位到错误位置并进行 debug。

图 5-2 为浮点加减法向量运算的测试波形，*faddcvt_enable* 为 1 使能浮点加减法与格式转换模块，*io_fp_format* 为 1 选择单精度模式，*io_rm* 为 1 即舍入模式为 RTZ，*io_fcmd* 为 0 代表执行浮点数加减法运算，*io_is_vec* 为 1 表示向量运算，执行 48808003+4068fafb 和 3fffc000+b3ffc000。当两个浮点数的符号位相同时执行浮点数加法，反之则执行减法，因此前者为浮点数加法运算，后者为浮点数减法运算。经过三级寄存器打拍后在第四个周期输出运算结果 48808077 和 3fffbfff，输出异常信号为 01，即存在不精确异常。

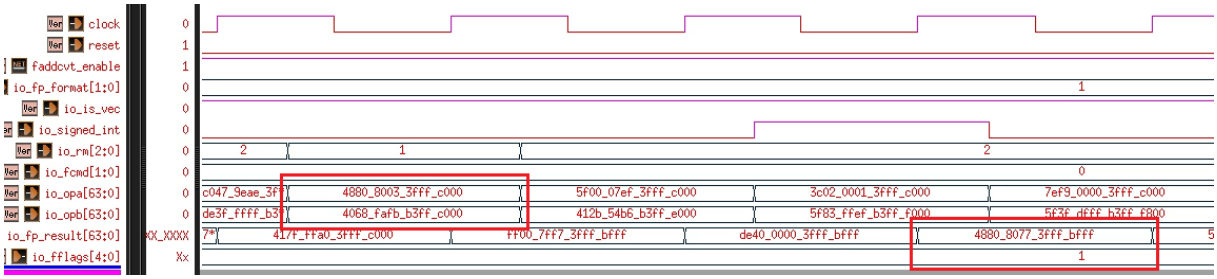

图 5-2 浮点加减法运算仿真波形图

图 5-3 为有符号整数转浮点数向量运算的测试波形，*faddcvt_enable* 为 1 使能浮点加减法与格式转换模块，*io_fp_format* 为 0 选择半精度模式，*io_rm* 为 2 即舍入模式为 RDN，*io_fcmd* 为 1 执行整数转浮点数运算，*io_signed_int* 为 1 执行有符号整数转换，io_is_vec 为 1 表示向量运算，将 136f、f80f、cff0 和 f80f 转换为浮点数。经三级寄存器打拍后在第四个周期输出结果 6cdb、e7f1、f202 和 e7f1，输出异常信号为 01，存在不精确异常。

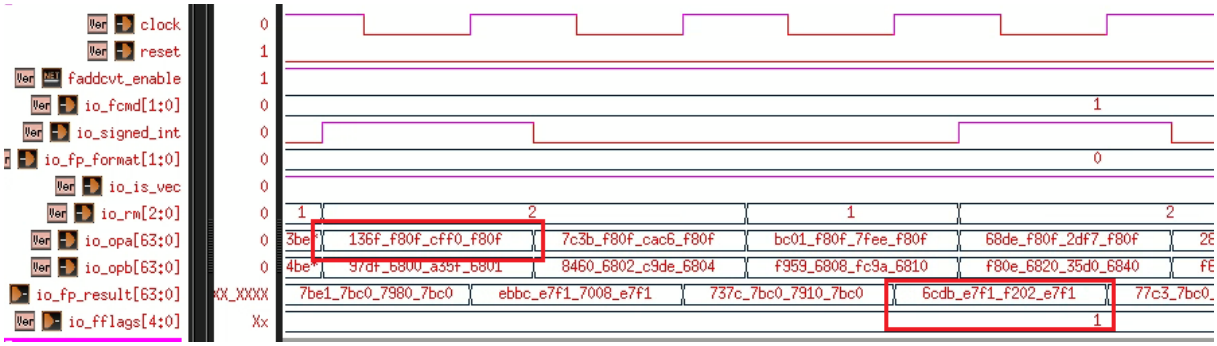

图 5-3 有符号整数转浮点数运算仿真波形图

图 5-4 为浮点数转有符号整数向量运算的测试波形，*faddcvt_enable* 为 1 使能浮点加减法与格式转换模块，*io_fp_format* 为 2 选择双精度模式，此时 *io_is_vec* 信号不影响结果，*io_rm* 为 4 即舍入模式为 RMM，*io_fcmd* 为 2 代表执行浮点数转整数运算，*io_signed_int* 为 1 代表结果为有符号整数，该过程将浮点数 c34f7fffffffffd 转换为有符号整数。经过三级寄存器打拍后在第四个周期输出运算结果 ffc1000000000006，输出异常信号为 00，没有异常发生。

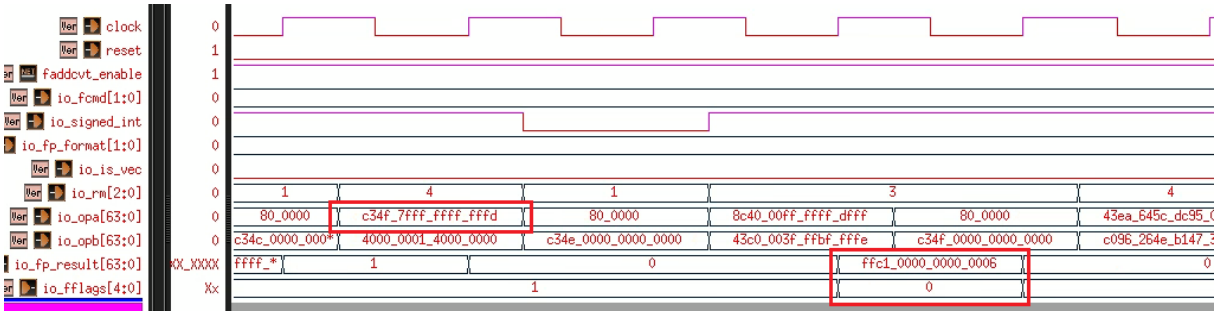

图 5-4 浮点数转有符号整数运算仿真波形图

图 5-5 为无符号整数转浮点数向量运算的测试波形，*faddcvt_enable* 为 1 使能浮点加减法与格式转换模块，*io_fp_format* 为 2 选择双精度模式，*io_rm* 为 4 即舍入模式为 RTZ，*io_fcmd* 为 1 代表执行整数转浮点数运算，*io_signed_int* 为 0 代表对无符号整数进行转换，该过程将无符号整数 3fc8a4d95375dae5 转换为浮点数。经过三级寄存器打拍后在第四个周期输出运算结果 43cfe4526ca9baed，输出异常信号为 01，存在不精确异常。

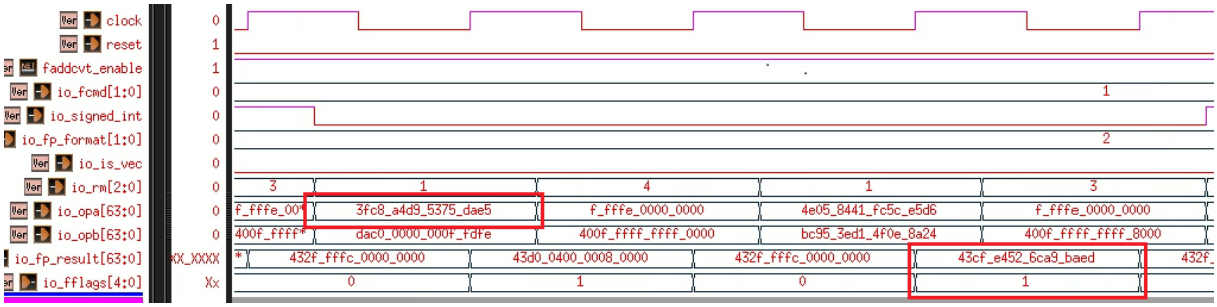

图 5-5 无符号整数转浮点数运算仿真波形图

图 5-6 为浮点数转无符号整数向量运算的测试波形，*faddcvt_enable* 为 1 使能浮点加减法与格式转换模块，*io_fp_format* 为 1 选择单精度模式，*io_rm* 为 3 即舍入模式为 RUP，*io_fcmd* 为 2 代表执行浮点数转整数运算，*io_signed_int* 为 0 代表结果为无符号整数，*io_is_vec* 为 1 表示向量运算，将浮点数 5e7807fe 和 3f000400 转换为无符号整数。经过三级寄存器打拍后在第四个周期输出运算结果 ffffffff 和 00000001，输出异常信号为 11，发生无效操作异常和不精确异常。其中浮点数 5e7807fe 转换为无符号整数为无效操作，由于超出了无符号整数的可表示范围，结果输出 32 位数所能表示的最大的无符号整数

ffffffff。

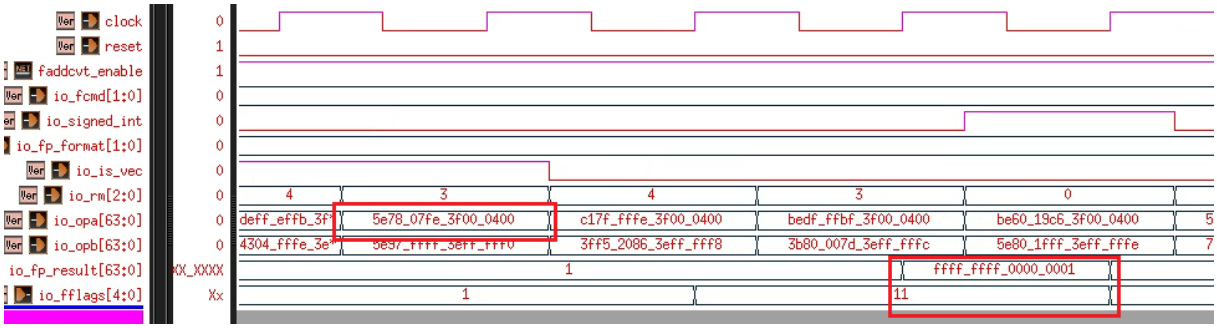

图 5-6 浮点数转无符号整数运算仿真波形图

图 5-7 为浮点乘加、乘减向量运算的测试波形，*fmaer_enable* 为 1 使能浮点乘加模块，*io_fp_format* 为 0 选择单精度模式，*io_rm* 为 4 即舍入模式为 RMM，*io_is_vec* 为 1 表示向量运算，执行 03c0×6bf0+3c0f、6805×cfde+95ee、03c0×6bf0+3c1f 和 6b07×5fca+805f。当执行 *opb*×*opb*+*opc* 时，此时浮点数 *opa*、*opb* 和 *opc* 的符号位异或结果为 0 时执行浮点乘加，反之执行浮点数乘减，因此最后一项执行浮点乘减，其他为浮点乘加。经过三级寄存器打拍后在第四个周期输出运算结果 3cfd、fbe8、3d0d 和 7c00，输出异常信号为 05，发生不精确异常和上溢异常。出现上溢异常是由于 6b07×5fca+805f 超出了浮点数可表示的最大值（不包括∞），由于此时舍入模式取 RMM，且结果为正值，依照 RISC-V 协议输出+∞，即 7c00。

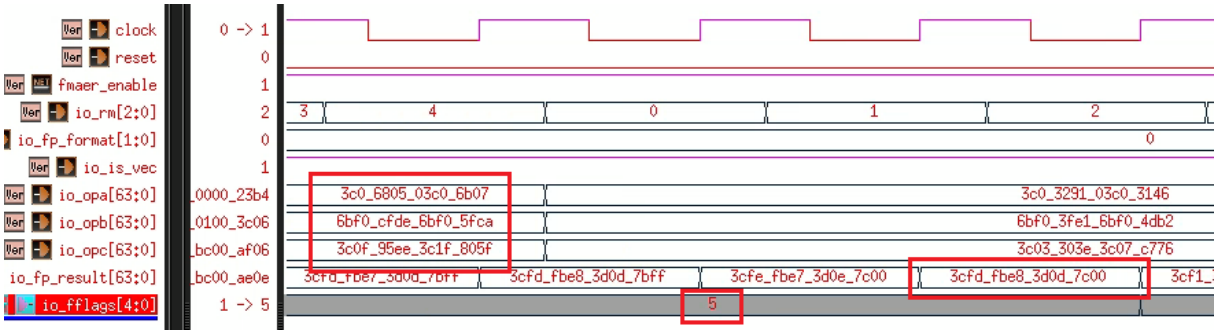

图 5-7 浮点乘加、乘减运算仿真波形图

图 5-8 为浮点除法向量运算的测试波形，*fdivsqrt_enable* 为 1 使能浮点除法与开平方模块，*io_fp_format* 为 0 选择半精度模式，*io_rm* 为 4 即舍入模式为 RMM，*io_is_fdiv* 为 1 表示执行除法，*io_is_vec* 为 1 表示向量运算，执行 34ff/36ff、9db5/78ff、34ff/377f 和 c80f/4f7f。*io_in_valid* 和 *io_in_ready* 均为高电平时开始运算，在第 10 个周期运算完成，拉高 *io_out_valid*，输出运算结果 39b7、8002、3955 和 b455，输出异常信号为 03，即存在不精确异常和下溢异常，下溢异常是由于浮点除法 9db5/78ff 的结果为非规格化数且结果为非精确值。

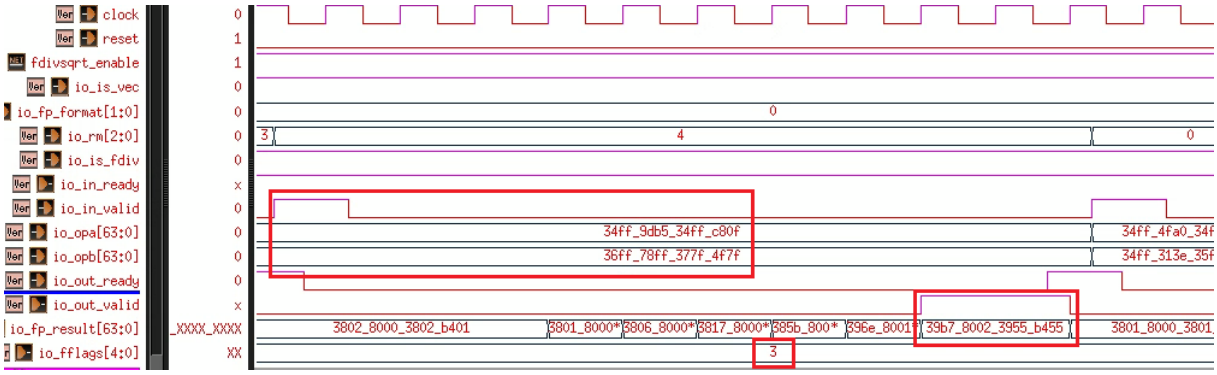

图 5-8 浮点除法运算仿真波形图

图 5-9 为浮点开平方向量运算的测试波形，*fdivsqrt_enable* 为 1 使能浮点除法与开平方模块，*io_fp_format* 为 1 选择单精度模式，*io_rm* 输入为 4 即舍入模式为 RMM，*io_is_fdiv* 为低电平表示执行开平方，*io_is_vec* 为 1 表示向量运算，对 7fffbfff 和 2f9bffff 进行开平方运算。*io_in_valid* 和 *io_in_ready* 均为高电平时开始运算，在第 16 个周期运算完成，拉高 *io_out_valid*，输出运算结果 7fc00000 和 378d4ee4，输出异常信号为 01，即存在不精确异常。注意 7fffbfff 为 NaN，且为 qNaN，根据 RISC-V 协议此时生成协议规定的 qNaN，即 7fc00000，且不会产生无效操作异常信号。

图 5-9 浮点开平方运算仿真波形图

对本文的浮点运算器所支持的浮点运算进行功能验证，大规模随机测试后的结果汇总至表 5-1，无错误情况发生。其中浮点除法与开平方运算测试时输入的测试激励个数相比其他两者的数量要少一些，这是由于运算器执行浮点除法和开平方运算时存在流水线阻塞，执行的周期更长，受限于仿真时间适当减少了测试激励的数量。

表 5-1 浮点运算大规模功能验证结果

浮点运算种类	*finished_test_num*	*error_test_num*
浮点加减法与格式转换	1500000000	0
浮点乘加、乘减	1500000000	0
浮点除法与开平方	500000000	0

随着测试激励数量的增加，系统可靠性也更高，且可以通过更改随机数种子使得每次仿真时生成的测试激励不同，提高测试激励的随机性。由此完成了大规模的功能验证，浮点运算器在功能上基本稳定，符合 IEEE 754-2019 标准和 RISC-V 协议的浮点运算要求，浮点运算器的功能验证通过。

5.2 浮点运算器的逻辑综合

确认功能无误后使用 Design Compiler（DC）工具完成逻辑综合以得到面积数据和时序收敛情况，并跟现有的浮点运算器进行对比。综合阶段工具把 HDL 代码与工艺库中的物理器件相映射以得到电路的物理数据，从工艺库中选取符合条件的硬件单元，实现将 HDL 描述的电路转换为门级网表[53]。综合阶段工具会对设计引入真实的延时信息，从而对各种时序路径进行静态时序分析，以确定设计满足建立时间和保持时间的要求，由此可以得到设计可正常工作的最高频率，且逻辑综合可得到设计的面积数据。

参考比较对象选取目前知名的 RISC-V 浮点运算器，包括 ETHZfpu[26]和 FPnew[27]，以及 Rocket Core[20,21]、Shakti[22]处理器内的浮点运算器，其中 ETHZfpu、Rocket Core 和 Shakti 基于 FPnew 的 parallel 方法实现向量化以便于与本课题设计的向量浮点运算器进行更准确的数据对比。此处不比较顶层控制逻辑的面积，因为其大小与处理器的模块划分相关，例如有的 FPU 顶层实现了指令的译码逻辑或浮点寄存器等功能，功能不一致难以评估。本课题使用 TSMC 7nm 工艺对这些 FPU 进行逻辑综合，对比实现相同浮点运算功能时所需的运算模块的面积，表 5-2 为各个 FPU 中运算模块的面积大小，“—”表示该 FPU 中不包含此模块。

表 5-2 各 FPU 中运算模块面积（μm^2）

Module	FPnew	ETHZfpu	Rocket Core	Shakti	本文
FADD	—	880.92	—	—	—
FCVT	1137.54	778.98	1761.75	946.14	—
FADDCVT	—	—	—	—	1101.00
FMA	4414.35	5227.20	6169.92	8991.94	3639.74
FDIVSQRT	4762.37	3237.12	989.15	4326.30	1223.19
Total	10314.26	10124.22	8920.82	14264.38	5963.93

由表 5-2 可知，本文设计的浮点乘加模块 FMA 优化效果明显，实现的 FMA 面积小

于其他 FPU，减小 18%~60%。本文将浮点加减法和浮点数与整数的转换均实现于浮点加减法与格式转换模块 FADDCVT，其他现有的 FPU 通常将浮点数与整数的转换实现于 FCVT 模块，而浮点加减法实现于 FADD 或 FMA 模块。本文的 FADDCVT 面积小于除 Shakti 外其他运算器的 FADD 和 FCVT 的面积和，减小 3%~38%。虽面积相比 Shakti 稍有增加，但 FADDCVT 相比 Shakti 的 FCVT 多增加一套可执行浮点加减法的运算资源，两者各有优势。而对于浮点除法与开平方模块 FDIVSQRT，本文的实现面积相比除 Rocket Core 外的其他 FPU 减小 62%~74%。Rocket Core 的 FDIVSQRT 面积较小是由于内部尾数除法和开平方模块只实现了一个周期计算出结果的一比特值，而本文可实现一个周期计算出结果的两比特值，虽面积有所增加，但速度和吞吐率更高。

综上由表 5-2 的结果可知，本文所设计的 FPU 在总面积上相比于现有的 FPU 是有一定优势的，表 5-2 中 Total 表示 FPU 的总面积，本文实现的 FPU 总面积相比于其他现有的 FPU 减小 33%~58%，面积的优化效果明显。

为检验面积效率的优化效果，需评估 FPU 的性能与面积的比值。由于 FPU 采用流水线设计，除浮点除法和开平方无法实现每周期输出一次结果，其他浮点运算在数据量趋于无穷大时均可每周期输出一个结果。由图 1-1 可知浮点除法和开平方运算在浮点系统中占比相比其他浮点运算要少得多，当数据量趋于无穷大时，其对 FPU 的平均吞吐率影响较小。为便于对比可认为 FPU 仍每周期输出一个结果，此时可将最高工作频率 f_{max} 等效为系统性能，本课题以最高工作频率与面积的比值来评估面积效率，f_{max} 代表 FPU 处理数据的最高速度。表 5-3 汇总了各 FPU 的最小工作周期 T_{min} 和最高工作频率 f_{max}，并由 $f_{max}/area$ 得到面积效率，$area$ 即为表 5-2 中 Total 的面积。

表 5-3 各 FPU 速度和面积效率对比

	FPnew	ETHZfpu	Rocket Core	Shakti	本文
T_{min}(ns)	0.48	0.52	0.42	0.40	0.38
f_{max}(MHz)	2083.33	1923.08	2380.95	2500.00	2631.58
$f_{max}/area$(MHz/μm^2)	0.20	0.19	0.27	0.18	0.44

由表 5-3 可得本文设计的 FPU 在速度上是有提升的，最高工作频率 f_{max} 高于所有现有 FPU，测得 f_{max} 为 2631.58MHz，相比现有 FPU 提高 5%~37%。

其中现有 FPU 中 Rocket Core 的面积效率最高，由 $f_{max}/area$ 算得为 0.27，而 Shakti 的面积效率最低，算得为 0.18。本文设计的 FPU 的面积效率可达到 0.44，相比现有 FPU

提高 63%~144%。综上，本课题所研究的浮点运算器满足高面积效率的需求。

表 5-4 汇总了各 FPU 的功耗情况，各 FPU 的总功耗均在 10^1mW 量级，处于可接受范围。可见本文尽管采用了高面积效率设计，但并不会引起功耗问题，满足应用要求。

表 5-4 各 FPU 功耗情况（mW）

	FPnew	ETHZfpu	Rocket Core	Shakti	本文
总功耗	27.39	24.99	18.27	49.96	24.70

5.3 浮点运算器的形式验证

逻辑综合时 DC 在对设计进行优化时可能会对电路的结构或者信号名进行修改，为了确保综合得到的门级网表与 HDL 代码功能一致，本课题进行形式验证。工具会将门级网表和 HDL 描述电路中的时序逻辑单元一一匹配，然后将相邻时序单元间的组合逻辑视作一个逻辑锥，之后根据数据传输的方向对这些逻辑锥进行输入[54]。如果门级网表和 HDL 代码中相对应的逻辑锥输出相同，那么由这两个时序逻辑单元和它们间组合逻辑所组成的电路在门级网表和 HDL 代码中功能等价。形式验证可以进一步提高设计的可靠性，具体的验证方法是将电路以端口和时序单元为节点将电路分成多个逻辑锥，逻辑锥内部实际就是节点间的组合逻辑电路，之后对这些逻辑锥的输入和输出进行检测。

形式验证使用 Formality 工具完成，其会对逻辑锥内的组合逻辑的逻辑表达式进行推导以进行等价性验证[55]。验证结果如图 5-10 所示，工具将 71 个 port 和 2670 个 DFF 作为节点进行逻辑锥的划分并进行检测，最终结果显示验证成功。

```
****************************** Verification Results ******************************
Verification SUCCEEDED
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
---------------------------------------------------------
 Reference design: r:/WORK/TOP
 Implementation design: i:/WORK/TOP
 2741 Passing compare points
-------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut     Port     DFF      LAT    TOTAL
-------------------------------------------------------------------------------------
Passing (equivalent)           0       0       0       0       71    2670        0     2741
Failing (not equivalent)       0       0       0       0        0       0        0        0
*************************************************************************************
```

图 5-10 形式验证测试结果

5.4 浮点运算器的板级验证

在芯片设计流程中，将代码下载到 FPGA 进行板级验证是提高设计可靠性的有效方法，本文使用 Nexys Video 开发板，内部为 Artix-7™ 系列芯片。代码通过 Vivado 工具下载至 FPGA 后，计算机通过 RS-232 串口线将待计算数据传输至待测浮点运算器，运

算结束后数据传回计算机，验证时利用串口调试助手将数据打印至显示屏中以便查看。

（1）浮点加减法运算

当两个源浮点数的符号位相同时执行浮点加法，反之执行浮点减法，图 5-11 为浮点加减法向量运算的验证结果，舍入模式取 RNZ，浮点数精度取半精度，执行 b38039ff80fd39ff+4e9abfdfac81bfbf=4e8bbce0ac81bcc0，生成异常指示信号 01，发生不精确异常。

图 5-11 浮点加减法板级验证结果

（2）有符号整数转浮点数

图 5-12 为有符号整数转浮点数向量运算的验证结果，舍入模式取 RTZ，浮点数精度取单精度，将 3da19852007c0000 转换为 4e7686614af80000，生成异常指示信号 01，发生不精确异常。

图 5-12 有符号整数转浮点数板级验证结果

（3）浮点数转有符号整数

图 5-13 为浮点数转有符号整数向量运算的验证结果，舍入模式取 RUP，浮点数精度取单精度，将 7f20001f00000020 转换为 7fffffff00000001，生成异常指示信号 11，发生无效操作异常和不精确异常。其中 7f20001f 转有符号整数为无效操作，由于结果超出了有符号整数可表示的范围，此时根据 RISC-V 协议输出最大的有符号正整数 7fffffff。

图 5-13 浮点数转有符号整数板级验证结果

（4）无符号整数转浮点数

图 5-14 为无符号整数转浮点数向量运算的验证结果，舍入模式取 RMM，浮点数精度取半精度，将 380010089b8312ff 转换为 73006c0278dc6cc0，生成异常指示信号 01，发生不精确异常。

图 5-14 无符号整数转浮点数板级验证结果

（5）浮点数转无符号整数

图 5-15 为浮点数转无符号整数向量运算的验证结果，舍入模式取 RTZ，浮点数精度取双精度，将 43c0000803ffffff 转换为 20001007fffffe00，生成异常指示信号 00，没有发生异常。

图 5-15 浮点数转无符号整数板级验证结果

（6）浮点乘加、乘减运算

当三个源浮点数的符号位异或为 0 时执行浮点乘加，反之执行浮点乘减。图 5-16 为浮点数乘加向量运算的验证结果，此时舍入模式取 RNZ，浮点数精度取半精度，执行 b7f22807786fccfb×37f4befe82ff054e+5bfb186b124ef787=5bf9aac4bea3f787，生成异常指示信号 01，发生不精确异常。

图 5-16 浮点乘加、乘减板级验证结果

（7）浮点除法运算

图 5-17 为浮点除法向量运算的验证结果，舍入模式取 RTZ，浮点数精度取单精度，执行 bfdfffffbf400006/00000000bffefffb=ff8000003ec0c0ca，生成异常指示信号 09，发生除 0 异常和不精确异常。bfdfffff/00000000 会发生除 0 异常，当除数为 0 时，除法结果为无穷大，结果符号位取被除数和除数符号位的异或，结果输出 ff800000，即-∞。

图 5-17 浮点除法板级验证结果

（8）浮点开平方运算

图 5-18 为浮点开平方向量运算的验证结果，舍入模式取 RUP，浮点数精度取双精度，3fefffc001ffffff 开平方得到 3fefffe000f000f0，生成异常指示信号 01，发生不精确异常。

图 5-18 浮点开平方板级验证结果

经验证，本课题设计的浮点运算器功能正确，通过板级验证，能根据五种 RISC-V 舍入模式和三种浮点数精度选择，正确计算浮点运算结果和生成 5 比特异常指示信号。

5.5 本章小结

本章对本课题所设计的适用于 RISC-V 的浮点运算器进行验证。基于软硬协同的思想搭建验证平台，利用随机数生成得到大量测试激励，输入到硬件 FPU 和软件 C 算法模型中，将两者输出的结果进行对比。然后使用 TSMC 7nm 工艺对实现的 FPU 和知名的 RISC-V FPU 进行逻辑综合，评估对速度和面积的优化效果，最高工作频率相比于现有 FPU 提高 5%~37%，面积减小 33%~58%，面积效率提升 63%~144%。之后进行形式验证，并将代码下载到 FPGA 上完成板级验证，完成本课题的验证工作。

总结与展望

1. 全文总结

随着信息化时代的到来，对数据处理的要求越来越高，浮点数相比于定点数精度更高、表示范围更大，更符合信息化场景的需要，但实现起来更为复杂，对浮点运算器进行研究是有所必要的。现有浮点运算器存在支持精度少、难以兼顾速度和面积导致面积效率低、向量化实现困难等问题，本课题着重对这些问题进行优化。

本课题旨在设计适用于 RISC-V 的高面积效率浮点运算器，遵循 IEEE 754-2019 标准，着重优化面积和速度以提高面积效率。基于开源 RISC-V 设计多使用 Chisel 语言，本课题采用 Chisel 进行开发，主要完成以下工作：

（1）本文的浮点运算器可执行 RISC-V 中的浮点加、减、乘、乘加、乘减、除、开平方和浮点数与整数的互相转换运算，并支持 RISC-V 的五种舍入模式和五种异常处理。改善过往研究实现精度单一的问题，支持半精度、单精度和双精度浮点运算，并实现 SIMD 向量运算，可并行执行四个半精度，或两个单精度，或一个双精度浮点运算。

（2）对浮点算法和硬件架构细致分析，以高面积效率的目标选取合适的算法进行硬件实现。合理规划流水线以提高最高工作频率，基于“并行化”思想设计并行移位器以优化浮点加减法中对阶移位的延时。基于前导 0 和后导 0 检测提出一种快速附加位求解算法，将乘加中的负结果修正从关键路径移除，并将修正产生的尾数加 1 和舍入合并处理，减少加 1 电路的使用。另外统一了浮点除法和开平方的求阶逻辑以及优化乘加使用的大位宽加法器以减少加法器开销。

（3）基于 CAS、Booth 算法和不恢复余数算法实现尾数加减法、乘法和除法与开平方电路。着重优化多种精度向量运算导致大面积的问题，在实现尾数运算模块时只使用一套硬件资源来实现多种精度的向量运算。基于“硬件隔离”的思想，提出多种精度融合的尾数向量运算结构，对尾数运算模块进行 SIMD 优化。硬件资源可根据不同位宽尾数的运算要求合理拆分，从而减小面积，并设计向量移位器以实现多种精度的向量移位。

（4）使用 System Verilog 搭建验证平台，生成大量测试激励，将硬件输出结果与 C 参考模型对比以进行功能验证，并使用 TSMC 7nm 工艺对本课题实现的 FPU 和其他现有 FPU 进行综合以进行数据对比，且最后通过形式验证和 FPGA 的板级验证。最终最高工作频率相比现有 FPU 提高了 5%~37%，面积减小 33%~58%，面积效率提高 63%~144%。

2. 工作展望

本课题的设计仍存在一些不足，未来工作主要包括以下几点：

（1）算法层面上查阅更多文献，探寻优化效果更好的浮点算法。

（2）硬件层面上对架构进行更深层的分析，对控制逻辑进一步优化以减少选择器的使用，另外浮点乘加模块的面积对比其他模块的面积仍较大，考虑进一步优化。

（3）功能上考虑支持更多浮点运算，例如不同精度浮点数的转换、不同位宽的浮点数和整数的转换等操作，且可考虑支持四精度的浮点运算。

参考文献

[1] WEI X., YANG H., LI W., et al. A reconfigurable 4-GS/s power-efficient floating-point FFT processor design and implementation based on single-sided binary-tree decomposition[J]. Integration: The VLSI journal, 2019, 66(May): 164-172.

[2] Li J., Zhou X., Wang B., et al. Design of Efficient Floating-Point Convolution Module for Embedded System[J]. Electronics, 2021, 10(4): 467.

[3] 刘厚青. 混合精度浮点的算术部件设计[D]. 安徽: 安徽大学, 2021.

[4] Mueller S. M., Jacobi C., Oh H. J., et al. The Vector Floating-point unit in a Synergistic Processor Element of a CELL Processor[A]. 17th IEEE Symposium on Computer Arithmetic[C]. Los Alamitos: IEEE Computer Society, 2005: 59-67.

[5] Baluni A., Merchant F., Nandy S. K., et al. A Fully Pipelined Modular Multiple Precision Floating-point Multiplier with Vector Support[A]. 2011 International Symposium on Electronic System Design[C]. Los Alamitos: IEEE Computer Society, 2011: 45-50.

[6] Chen J., Yang C. Optimizing SIMD Parallel Computation with NonConsecutive Array Access in Inline SSE Assembly Language[A]. 2012 Fifth International Conference on Intelligent Computation Technology and Automation[C]. Los Alamitos: IEEE Computer Society, 2012: 254-257.

[7] 胡振波. 手把手教你设计CPU——RISC-V处理器篇[M]. 北京: 人民邮电出版社, 2018: 30-32.

[8] 种丹丹. 基于RISC-V的开源芯片生态发展现状及未来机遇[J]. 中国集成电路, 2021, 30(8): 25-30.

[9] Oberman S. F. Design issues in high performance floating-point arithmetic units[R]. California: Stanford University, 1996.

[10] Boersma M., Kroner M., Layer C., et al. The POWER7 Binary Floating-point unit[A]. 2011 IEEE 20th Symposium on Computer Arithmetic[C]. Los Alamitos: IEEE Computer Society, 2011: 87-91.

[11] Manolopoulos K., Reisis D., Chouliaras V. A. An efficient multiple precision floating-point Multiply-Add Fused unit[J]. Microelectronics Journal, 2016, 49(Mar): 10-18.

[12] Lai J., Seznec A. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs[A]. Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization[C]. Los Alamitos: IEEE Computer Society, 2013: 1-10.

[13]张戈, 齐子初, 胡伟武. 龙芯2号处理器功能部件设计[J]. 计算机研究与发展, 2006, 43(6): 7.

[14]吴瑞阳, 汪文祥, 王焕东, 等. 龙芯GS464E处理器核架构设计[J]. 中国科学: 信息科学, 2015, 45(4): 480-500.

[15]Kathiara J., Leeser M. An Autonomous Vector/Scalar Floating-point Coprocessor for FPGAs[A]. IEEE International Symposium on Field-programmable Custom Computing Machines[C]. Los Alamitos: IEEE Computer Society, 2011: 33-36.

[16]Li K., Mao W., Xie X., et al. Multiple-Precision Floating-Point Dot Product Unit for Efficient Convolution Computation[A]. 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems[C]. Piscataway: IEEE, 2021: 1-4.

[17]潘树朋, 刘有耀, 焦继业, 等. 基于RISC-V浮点指令集FPU的研究与设计[J]. 计算机工程与应用, 2021, 57(3): 7.

[18]754-2019 - IEEE Standard for Floating-Point Arithmetic[S]. USA: IEEE, 2019.

[19]The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213[S]. USA: RISC-V Foundation, 2019.

[20]ASANOVIC K., AVIZIENIS R., BACHRACH J., et al. The rocket chip generator: UCB/EECS-2016-17[R]. Berkeley: University of California, 2016.

[21]Neves E. A., Xavierdesouza S. Exploring Multi-core Design Space: Heracles vs. Rocket Chip Generator[J]. Journal of Computers, 2018, 13(5): 555-563.

[22]Gala N., Menon A., Bodduna R., et al. SHAKTI Processors: An Open-Source Hardware Initiative[A]. 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems[C]. Los Alamitos: IEEE Computer Society, 2016: 7-8.

[23]周毅. 基于RISC-V的PULPino SoC的FPGA原型设计和物理实现[D]. 黑龙江: 哈尔滨工业大学, 2019.

[24]Pullini A., Rossi D., Loi I., et al. Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing[J]. IEEE Journal of Solid-State Circuits, 2019, 54(7): 1970-1981.

[25]Gautschi M., Schiavone P. D., Traber A., et al. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2017, 25(10): 2700-2713.

[26]Gautschi M., Mach S., Ness T. V., et al. pulp-platform fpu. https://github.com/pulp-platform/fpu, 2018.

[27] Mach S., Schuiki F., Zaruba F., et al. FPnew: An Open-Source Multiformat Floating-point unit Architecture for Energy-Proportional Transprecision Computing[J]. IEEE transactions on very large scale integration (VLSI) systems, 2021, 29(4): 774-787.

[28] Xu Y., Yu Z., Tang D., et al. Towards Developing High Performance RISC-V Processors Using Agile Methodology[A]. 2022 55th IEEE/ACM International Symposium on Micro-architecture[C]. Piscataway: IEEE, 2022: 1178-1199.

[29] 韩正飞, 李劲松, 潘红兵, 等. 基于FPGA的浮点向量协处理器设计[J]. 计算机工程, 2012, 38(5): 251-254.

[30] Huang L., Sheng M., Li S., et al. Low-Cost Binary128 Floating-Point FMA Unit Design with SIMD Support[J]. IEEE Transactions on Computers, 2012, 61(5): 745-751.

[31] 凌智强, 谈民, 曾献君. "前导零预测——并行修正"算法中错误的分析和修正[J]. 计算机研究与发展, 2007, 44(z1): 31-34.

[32] Schmookler M. S., Nowka K. J. Leading zero anticipation and detection -- a comparison of methods[A]. Proceedings 15th IEEE Symposium on Computer Arithmetic[C]. Los Alamitos: IEEE Computer Society, 2001: 7-12.

[33] Soderquist P., Leeser M. Area and performance tradeoffs in floating-point divide and square-root implementations[J]. ACM Computing Surveys, 1996, 28(3): 518-564.

[34] Niwal L. S., Hajare S. P. Design of radix 4 divider circuit using SRT algorithm[A]. 2015 International Conference on Communications and Signal Processing[C]. Piscataway: IEEE, 2015: 1107-1110.

[35] 贾志. 浮点除法器的设计与性能改进[D]. 辽宁: 大连理工大学, 2021.

[36] Bansal H., Sharma K. G., Sharma T. Wallace Tree Multiplier Designs: A Performance Comparison Review[J]. Innovative Systems Design & Engineering, 2014, 5(5): 60-67.

[37] Ghasemi M. M., Fathi A., Mousazadeh M., et al. A new high speed and low power decoder/encoder for Radix-4 Booth multiplier[J]. International Journal of Circuit Theory and Applications, 2021, 49(7): 2199-2213.

[38] Yeh W. C., Jen C. W. High-speed Booth encoded parallel multiplier design[J]. IEEE Transactions on Computers, 2000, 49(7): 692-701.

[39] Waters R. S., Swartzlander E. E. A Reduced Complexity Wallace Multiplier Reduction[J]. IEEE Transactions on Computers, 2010, 59(8): 1134-1137.

[40] Jaiswal M. K., Cheung R. C. C., Balakrishnan M., et al. Series Expansion based Efficient Architectures for Double Precision Floating-point Division[J]. Circuits, Systems, and Signal Processing: CSSP, 2014, 33(11): 3499-3526.

[41] Li Y., Chu W. A new non-restoring square root algorithm and its VLSI implementations[A]. Proceedings International Conference on Computer Design: VLSI in Computers and Processors[C]. Los Alamitos: IEEE Computer Society, 2002: 538-544.

[42] Senthilpari C., Kavitha S., Joseph J. Lower delay and area efficient non-restoring array divider by using Shannon based adder technique[A]. 2010 IEEE International Conference on Semiconductor Electronics[C]. Piscataway: IEEE, 2010: 140-144.

[43] 白中英, 戴志涛等. 计算机组成原理（第六版·立体化教材）[M]. 北京: 科学出版社, 2019: 38-42.

[44] 刘志刚, 汪旭东, 郑关东. 基于SRT算法的单精度浮点除法器[J]. 电子技术应用, 2007, 33(10): 4.

[45] 王聪, 宋兆虎, 周武. 改进不恢复余数浮点开方电路的研究与实现[J]. 战术导弹控制技术, 2009, 0(4): 20-22.

[46] 王文广, 曹建, 陈志敏. 改进的不恢复余数的浮点开方算法的研究与FPGA实现[J]. 现代电子技术, 2007, 30(16): 68-71.

[47] Hutchins S., Swartzlander E. A Bfloat16 Fused Multiplier-Adder[A]. 2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference[C]. Piscataway: IEEE, 2020: 52-55.

[48] Kim S. Rutenbar R. A. An Area-Efficient Iterative Single-Precision Floating-Point Multiplier Architecture for FPGA[A]. Proceedings of the 2019 on Great Lakes Symposium on VLSI[C]. New York: Association for Computing Machinery, 2019: 87-92.

[49] Bewick G. W. Fast Multiplication: Algorithms and Implementations[M]. Stanford University, 1994: 16-18.

[50] Gopineedi P. D., Thapliyal H., Srinivas M. B., et al. Novel and Efficient 4: 2 and 5: 2 Compressors with Minimum Number of Transistors Designed for Low-Power Operations[A]. International Conference on Embedded Systems & Applications[C]. Las Vegas: CSREA Press, 2006: 160-168.

[51] Veeramachaneni S., Krishna K., Avinash L., et al. Novel Architectures for High-Speed and Low-Power 3-2, 4-2 and 5-2 Compressors[A]. 20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems[C]. Los Alamitos: IEEE Computer Society, 2007: 324-329.

[52] Savas S., Hertz E., Nordstrom T., et al. Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis[A]. 2017 IEEE Computer Society Annual Symposium on VLSI[C]. Los Alamitos: IEEE Computer Society, 2017: 110-115.

[53] Synopsys. Design Compiler User Guide Version P-2019.03[M]. USA: Synopsys Inc., 2019: 1-6.

[54] Synopsys. Formality User Guide Version T-2022.03[M]. USA: Synopsys Inc., 2022: 21-26.

[55] 沈俊. 浮点运算加速器的设计研究[D]. 浙江: 浙江大学, 2013.