MASTER THESIS XIDIAN UNIVERSITY

西安电子科技大学

硕士学位论文

基于RISC-V指令集的宇航用浮点单元容错研究与实现

作者姓名 梁宗南

指导教师姓名、职称 杨孟飞 教授

申请学位类别 工学硕士

学校代码 10701　　学　号 20031211393
分 类 号 TP316　　密　级 公开

西安电子科技大学

硕士学位论文

基于 RISC-V 指令集的宇航用浮点单元容错研究与实现

作者姓名：梁宗南

一级学科：计算机科学与技术

二级学科（研究方向）：

学位类别：工学硕士

指导教师姓名、职称：杨孟飞　教授

学　　院：计算机科学与技术学院

提交日期：2023 年 3 月

Research and Implementation of Floating Point Unit Fault-Tolerant for Aerospace Based on RISC-V

A thesis submitted to

XIDIAN UNIVERSITY

in partial fulfillment of the requirements

for the degree of Master

in Computer Science and Technology

By

Liang Zongnan

Supervisor: Yang Mengfei Title: Professor

March 2023

摘要

现代宇航技术快速发展，对宇航处理器的处理性能、可靠性等方面提出了越来越高的要求。宇航处理器在恶劣的环境中运行，暴露在空间辐射、极端温度等环境中会使系统发生故障。相比于整数单元，处理器中的浮点单元具有较大的硅面积，受到粒子影响的概率更大，更容易出现故障。目前对于浮点寄存器堆采用了多种方式进行加固，主要基于空间冗余和信息冗余进行，纠错操作会中断处理器的执行流程，且未考虑到近邻单元的连续多比特错误的问题。在浮点运算单元的加固方面，使用双模冗余方法对运算过程中的错误进行检测，这种容错方式带来了较大的面积开销。

本文首先对IEEE 754浮点数标准和RISC-V浮点扩展指令集进行研究，对浮点运算单元中进行容错的必要性和可行性进行分析，将浮点数运算过程根据数据路径划分为两个阶段：浮点寄存器堆读写、浮点数算术运算。对浮点寄存器堆的访问间隔进行统计，浮点寄存器堆中存在长时间未被改写的寄存器，只对寄存器的输出进行纠正时，会存在错误累积问题。此外，对浮点寄存器堆的错误敏感性进行评估，实验结果表明，尾数的低位中出现的错误对整体运算结果的影响较小，符号、阶码、尾数的高位中出现的错误对整体运算结果的影响较大，这一现象是由于浮点数运算过程中的对阶操作和舍入操作导致的，因此可以对浮点寄存器的不同位置采用不同的容错策略。

针对浮点寄存器堆，提出了一种基于 Cache 的寄存器堆结构，可以在寄存器堆的访问空闲期间进行检错和纠正，避免了错误的累积，相比于其他的寄存器堆结构，这种寄存器堆资源开销更少。此外，校验器基于交叠编码技术进行设计，可以检测到寄存器中近邻连续多比特位的翻转错误。

针对浮点运算单元，使用乘积码对操作数进行编码，并对编码后的操作数进行运算，通过特定的解码器对运算过程中的单比特错误进行检测。对除三电路进行改进，提出了一种基于查找表的无减法器的快速电路，面积减少了39.9%，延时减少了28.7%。基于上述研究成果，对 RISC-V 指令集浮点运算单元中的浮点加法器进行了容错设计，相比于直接使用双模冗余进行加固，面积减少了48%。

本文针对宇航用浮点单元的容错设计方法进行研究，对浮点单元进行加固设计，实现基于 RISC-V 指令集的宇航用高可靠浮点单元，对我国下一代宇航用处理器的研制具有一定的参考意义。

关 键 词：RISC-V，浮点单元，寄存器堆，容错计算，可靠性

ABSTRACT

The rapid development of modern aerospace technology has put forward increasingly high requirements for the processing performance, reliability, and other aspects of aerospace processors. Aerospace processors operate in harsh environments, exposuring to space radiation, extreme temperatures, and other environments can cause system failures. Compared to integer units, floating-point units in processors have a larger silicon area, are more likely to be affected by particles, and are more prone to malfunctions. At present, multiple methods have been used to reinforce floating-point register file, mainly based on spatial redundancy and information redundancy. The fault tolerance ability is limited, and error correction operations will interrupt the execution process of the processor, without considering the problem of continuous multi bit errors in neighboring units. In terms of the reinforcement of floating-point computing units, the dual module redundancy method is used to detect errors during the operation process, which brings significant area overhead.

In this paper, we first study the IEEE 754 floating-point standard and RISC-V floating-point extended instruction set, analyze the necessity and feasibility of fault tolerance in the floating-point unit, and divide the floating-point operation process into two stages according to the data path, which are floating-point register file read/write and floating-point arithmetic operation. Statistics are conducted on the access intervals of the floating-point register file. There are registers in the floating-point register file that have not been rewritten for a long time. When only correcting the output of the registers, there will be an error accumulation problem. In addition, the fault sensitivity of the floating-point register file is evaluated. The experimental results show that the errors in the low bits of the mantissa have a small impact on the overall operation result, while the errors in the sign bit, exponent, and the high bits of mantissa have a large impact on the overall operation result. This phenomenon is caused by the opposite order operation and rounding operation in the process of floating-point operation, Therefore, different fault-tolerant strategies can be adopted for different positions of floating-point registers.

For floating-point register file, this paper proposes a register file structure based on Cache, which can detect and correct errors during idle access of the register file, avoiding error accumulation. Compared with other register file structures, this has less resource overhead.

In addition, the checker is designed based on the interleved technology, which can detect the flipping error of adjacent consecutive multi bit in the register.

For the floating-point operation unit, the product code is used to encode the operands, and the encoded operands are operated. A specific decoder is used to detect single bit errors during the operation. The circuit used to divide by three is improved, and a fast circuit without subtracter based on look-up table is proposed. The area is reduced by 39.9%, and the delay is reduced by 28.7%. Based on the above research results, a fault-tolerant design for the floating-point adder in the floating-point unit with RISC-V instruction set is performed, which reduced the area by 48% compared to directly using dual module redundancy for reinforcement.

In this paper, the fault-tolerant design method of the floating-point unit for aerospace is studied, the floating-point unit is hardened, and the highly reliable floating-point unit for aerospace based on RISC-V instruction set is realized, which has certain reference significance for the development of the next generation of aerospace processor in China.

Keywords: RISC-V, Floating Point Unit, Register File, Fault-Tolerant Computing, Reliability

插图索引

表格索引

缩略语对照表

缩略语	英文全称	中文对照
EDC	Error Detection Code	错误检错码
ECC	Error Correction Code	错误纠错码
EDAC	Error Detection And Correction	错误纠错与检错
SEC	Single Error Correction	纠一位错
SEC-DED	Single Error Correction-Double Error Detection	纠一位错、检两位错
FPGA	Field Programmable Gate Array	现场可编程门阵列
ASIC	Application Specific Integrated Circuit	专用集成电路
SRAM	Static Random Access Memory	静态随机存储器
I-SRAM	Instruction Static Random Access Memory	指令 SRAM
D-SRAM	Data Static Random Access Memory	数据 SRAM
CPU	Central Processing Unit	中央处理器
FPU	Floating Point Unit	浮点单元
TMR	Triple Modular Redundancy	三模冗余
DMR	Dual Modular Redundancy	双模冗余
UART	Universal Asynchronous Receiver/Transmitter	通用非同步收发器
MUX	Multiplexer	多路选择器
CLB	Configurable Logic Block	可配置逻辑块
LUT	Look-up Table	查找表
FF	Flip-flop	触发器
RPC	Reduced Precision Checking	降低精度检查

目 录

第一章 绪论

1.1 研究背景与意义

纵观宇航用处理器的发展历程，欧洲主要基于 SPARC 架构进行设计[1,2]，美国则是基于 PowerPC 架构进行设计[3,4]，SPARC 架构和 PowerPC 架构是宇航应用领域的主导架构。但是，近些年来 SPARC 架构和 PowerPC 架构的发展缓慢，许多商业公司更是停止了对 SPARC 处理器的发展与维护，促使宇航用处理器需要转向具有更大商业市场的架构，对此，许多国家的航天机构开始转向设计基于 RISC-V 架构的宇航用处理器[5]。

SPARC 结构由于其架构稳定、体系标准开放的特点，我国目前研制的宇航用处理器都是基于 SPARC 架构进行设计的。SPARC 架构具有可扩展的特性，这一特性体现在处理器具有大量的寄存器，可以根据需求决定需要使用的实际寄存器的数量，以平衡电路成本和程序调用时上下文切换时产生的负担。SPARC 架构中的寄存器通过寄存器窗口来管理，寄存器窗口使得简单的软件程序在调用时无需对寄存器进行保存，提高了执行效率，但是对于操作系统的设计，寄存器窗口所带来的优势并不明显，原因在于，操作系统在设计时不仅需要考虑窗口溢出问题，并且在系统上下文切换时需要大量的线程栈来对寄存器进行保存，同时对这些寄存器的容错设计也需要消耗大量的资源。

RISC-V 是一种新兴的开源精简指令集架构，由加州大学伯克利分校在 2010 年首次发布并使用[6]。RISC-V 顺应现代计算机系统设计需求和现有的体系结构发展趋势，由于其推出时间较晚，规避了多年以来计算机架构发展过程中暴露的问题，去除了对历史遗留问题的兼容和对旧技术的依赖。RISC-V 架构由于其开放的特性，没有非技术性问题的干预，符合我国宇航用处理器自主发展的基本要求，成为我国下一代宇航用处理器架构的首选。RISC-V 吸收了其他体系结构的优点，并且采用模块化设计，支持指令集的扩展，允许开发者根据资源消耗、安全性、实时性等不同需求，基于扩展指令集进行专业化的系统设计与开发，从而实现系统高度可定制化[7]。

现代宇航技术快速发展，对星载计算机的控制系统、航电系统等关键电子系统的处理性能、可靠性等方面提出了越来越高的要求。这些系统在恶劣的环境中运行，暴露在空间辐射、极端温度等环境中会使系统发生短暂或者永久性的故障[8]。宇航高可靠处理器是星载计算机系统中的重要部件之一，其可靠性关系到整个星载计算机系统的稳定。近些年来，随着半导体制造工艺的不断提升，器件尺寸减少、供电电压降低，大大地降低了器件的噪声容限，导致辐射敏感度不断增加，在这些工艺下制造的器件

受到高能粒子所带来的影响更为严重[9-12]，在设计阶段进行容错设计的必要性变得越来越重要。

浮点数相比整型数，数值可表示范围更大，广泛应用于科学计算、导航控制、图像处理等高性能计算领域。随着空间任务的复杂化，对处理器的性能提出了更高的要求，许多宇航处理器都内置了浮点单元。空间环境中的各种高能粒子的影响会导致处理器出现故障，浮点单元具有较大的硅面积，受到高能粒子的影响的概率更大，因此更容易出现故障[13]。

本文针对宇航用浮点单元的容错设计方法进行研究，对浮点单元进行加固设计，提高浮点寄存器堆的容错能力，减少纠错操作对处理器产生的影响，减少对浮点运算单元进行容错设计产生的资源开销，实现基于 RISC-V 指令集的宇航用高可靠浮点单元，为我国下一代宇航用处理器的设计提供了理论方法和工程实践经验。

1.2 国内外研究现状

目前，已有多种宇航用高可靠处理器成功应用于不同种类的航天器上。面对未来宇航用处理器的需求，选择架构稳定、体系标准开放的架构进行宇航用高可靠处理器的设计成为我国宇航用处理器设计的首选。

AT697F[1]内部集成了一个符合 IEEE 754 标准的浮点单元，是一款基于 SPARC V8 架构设计的高性能、低功耗的 32 位宇航用处理器芯片。AT697F 采用三模冗余进行加固设计，三模冗余基于三个具有不同偏斜的时钟，可以有效减少单粒子翻转故障带来的影响。

GR740[2]是一款采用 32 位 LEON4 内核、具有 7 级流水线的 SPARC V8E 架构宇航用处理器，对通用寄存器堆和浮点寄存器堆等片上块存储设备进行加固处理，对单粒子翻转故障具有一定的容错能力。

北京控制工程研究所设计的 SoC2008 处理器[14]基于 SPARC V8 架构，内置支持单、双精度浮点数据类型的浮点单元，实现了所有 SPARC V8 浮点指令。SoC2008 对寄存器堆采用 EDAC（Error Detection And Correction）进行保护，可以检测两位错、纠正一位错。对于运算单元，SoC2008 提供了多重容错策略在处理器运行期间进行选择，包括三模冗余、奇偶校验、BCH 校验，对错误进行的纠正会增加指令执行的时钟周期。

北京微电子研究所设计的 BM3823 处理器[15]内置一个浮点单元，可以处理单精度和双精度浮点数，浮点数格式和指令集符合 ANSI/IEEE 754-1985 标准。BM3823 采用三模冗余和 EDAC 技术进行容错设计，对浮点运算单元中的寄存器采用三模冗余进行加固，对浮点寄存器堆采用 Hamming 码进行编码。当检测到浮点寄存器堆中的

可纠正错误时，流水线将被暂停，纠正后的数据会按照读出的地址写回到寄存器堆中，然后重启流水线，从暂停位置重新运行程序，对于不可纠正的错误，采用陷阱方式进行处理。

综上所述，目前已经在轨应用的宇航用高可靠处理器通常使用 EDAC 对寄存器堆进行加固，对运算部件进行三模冗余，采用抗辐照单元库在后端设计时进行综合。对寄存器堆中的错误的检查和纠正操作只在读寄存器时进行，因此当某个寄存器长时间不被访问时，该寄存器中出现的错误将不会被处理。此外，对错误的纠正会引起流水线断流，这将对处理器的性能产生一定影响。随着半导体工艺的提升，近邻单元之间的耦合进一步加强，单次高能粒子事件引起近邻连续位翻转的概率逐步升高，连续的多比特错误的情况应该予以重视。

近些年来，RISC-V 架构由于其开放、模块化的特点，可实现高度定制化，逐渐应用于宇航用高可靠处理器的设计中。在 RISC-V 架构中，浮点单元（Floating Point Unit）的结构如图 1.1 所示，由浮点运算单元（FPOU，Floating-point Operation Unit）和浮点寄存器堆（FPRF，Floating-point Register File）组成。浮点运算的执行过程为从浮点寄存器堆中读取操作数，在浮点运算单元中对操作数进行算术运算，最后将运算的结果写回浮点寄存器堆中。浮点运算单元中的寄存器（REG，Register）用于对操作数和运算结果进行暂存，由于浮点运算逻辑复杂，通常无法在一个时钟周期内完成运算操作，因此浮点运算过程（OP 阶段）会被划分多个子阶段，每个时钟周期执行一个子阶段，每个子阶段的执行结果也会被暂时存放在寄存器中。

针对浮点单元的容错设计从两个方面进行，即浮点运算单元容错和浮点寄存器堆容错，后文将详细介绍这两个方面国内外的研究现状。

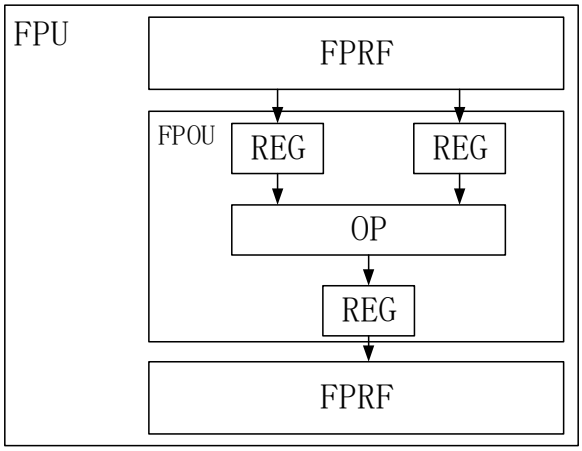

图1.1 浮点单元结构图

1.2.1 浮点运算单元容错

对浮点运算单元进行容错设计，最简单的方式就是采用复制的方式，这种方式也被称为双模冗余（DMR，Dual Modular Redundancy），通过对并行传输的两个数据进行比较就可以知道数据路径是否出错，这种方式可以应用于任何给定的设计，只需要复制一条数据路径并添加一个比较器就可以实现，但是这种方式会产生额外 100%的面积开销。三模冗余（TMR，Triple Modular Redundancy）也是一种常用且相对简单的加固方式，Gallagher 等人[16]对浮点除法器使用了三模冗余进行加固。相比双模冗余，三模冗余可以直接对错误进行纠正，不会造成流水线断流，因此在时间开销上具有优势，但是三模冗余增加了额外 200%的逻辑资源消耗，并且还会影响系统的关键路径，带来额外的路径延时。采用三模冗余加固的处理器很难运行在较高的时钟频率上，在商业处理器上很少使用，但是对于宇航用处理器，为了提高稳定性而牺牲一部分性能是值得的。

直接使用多模冗余对浮点运算单元进行加固会带来大量的面积开销，对此，Eibl[17]等人和 Seetharam[18]等人进行了折中考量，通过减少用于检查的运算电路的操作数的位宽来减少电路面积，提出了 RPC（Reduced Precision Checking）方法，分别对浮点加法单元和浮点乘法单元进行了容错设计。该方法与双模冗余的方式相似，对被保护的运算单元进行复制，得到备份的运算单元，通过校验器对两个运算单元的运算结果进行比对，但是不同的是，RPC 方法中备份的运算单元仅实现了对浮点数的符号位、阶码、尾数的高位的运算，因此在提高容错能力的同时有效降低了资源开销。由于备份的运算单元的操作数只对尾数的高位进行运算，因此不能检测到尾数低位中的错误。此外，被截断的操作数会由于丢失了一些数据，直接运算后的结果会有误差，为了解决这一问题，这些研究中设计了修正电路，来修正由于操作数低位被截断造成的运算误差。

浮点运算单元中除了运算逻辑，还包含控制逻辑。Maniatakos 等人[19]在实验中发现，在以 IEEE 754 浮点数标准设计的浮点运算单元中，控制逻辑中的错误会导致广泛的数据路径损坏，并以很高的概率影响指数部分，因此指数监控可用于检测浮点运算单元控制逻辑中的错误。对此，Maniatakos 等人提出了一种非侵入式并发错误检测方法，通过对指数部分进行监视以保护浮点运算单元的控制逻辑，基于这种方法设计的单元可以检测到 93%以上的瞬时错误和 98.1%以上可以影响到指数的错误，并且产生的面积开销要低于三模冗余方式，仅增加 5.8%。

对于算术运算电路的加固，还可以采用算术检错码进行编码，通过解码可以检测算术运算电路在执行过程中是否出错。常用的算术检错码有乘积码和剩余码，但是二者在浮点运算中的应用受限，原因是浮点运算过程中需要对操作数进行移位，移位操作会破坏算术检错码的编码结构，并且浮点运算的一个显著的特点是需要进行舍入操

作，舍入操作的存在会进一步破坏算术检错码的编码结构。

综上所述，目前针对浮点运算单元的加固主要基于多模冗余进行，并且为了减少多模冗余带来的面积开销，一些研究工作中对用于检测的运算电路中的数据位宽进行缩减，在检错能力和资源消耗方面进行了折中考量。

1.2.2 寄存器堆容错

处理器中的寄存器堆分为通用寄存器堆和浮点寄存器堆，二者在存储结构和读写操作上基本是一致的，具有很大的相似性。目前国内外针对寄存器堆容错技术开展了较为深入的研究，主要从空间冗余和信息冗余两个方面进行研究。

对寄存器堆使用三模冗余可实现对多位错误的屏蔽，但是对寄存器堆中所有的寄存器都进行三模冗余可能会导致过多的面积开销和功耗开销，并且一些寄存器对处理器的可靠性的影响较小，因此没有必要对寄存器堆中的全部寄存器采用三模冗余。Liang 等人[20]通过分析不同软件程序对寄存器堆的使用情况，设计了一种有效的策略，可以根据寄存器的脆弱性对其进行评级，并基于该策略提出了一种新的寄存器堆容错方法，即 Partial-TMR，该方法选择性地保护更脆弱的寄存器免受多位错误的影响。相比于未经加固处理的浮点寄存器堆，使用这种方法加固后的系统在容错能力上提升了 5.17%，虽然纠错能力略低于全三模冗余的，但是相比于全三模冗余减少了 71.6%的面积开销和 64.9%的功耗开销。

三模冗余是一种常用的空间冗余方法，电路结构简单且易于实现，但是带来了巨大的面积开销与功耗开销。相比之下，信息冗余通过增加少量冗余位来对原始数据进行保护，面积开销较少。Hamming 码是最常用的信息冗余方法，可以实现单比特错误的检测与纠正，也称为 SEC 码（Single Error Correction Code）。添加额外的 1 位全局校验，Hamming 码可以实现检测两位错误、纠正一位错误，即 SEC-DED 码（Single Error Correction-Double Error Detection Code）。ParShield[21]使用 Hamming 码来保护寄存器堆，通过 Hamming 码可以实现对寄存器堆中的错误进行检测与纠正，同时为了减少面积开销与功耗开销，选择性地保护更易收攻击的寄存器。与 ParShield 类似，Santos 等人[22]同样使用 Hamming 码来对寄存器堆进行保护，并且 Hamming 码还被应用于对 PC 寄存器的保护。由于寄存器堆处于系统的关键路径上，同时解码器电路较为复杂，对寄存器堆的容错设计会增加路径延时。对此，Reviriego 等人[23]对基于 SEC-DED 的 Hamming 码进行改进，只对数据位中出现的错误进行纠正，对校验位中的错误则不进行处理。改进后的编码器和解码器分别减少了 20.6%和 14.4%的面积开销，在延时方面，改进后的解码器减少了 8.9%。

近些年来，基于 SRAM 的 FPGA 上实现的软处理器由于其灵活性和易于集成的特点，越来越多地被用于宇航应用领域。Ramos 等人[24]根据 FPGA 的特点，提出了一

种适用于 FPGA 平台的软核处理器寄存器堆的加固方法。该方法通过将软处理器的寄存器堆映射到片上的分布式 RAM 中来实现备份，通过添加奇偶校验来对寄存器堆中的错误进行检测，支持同时对寄存器堆中的最多两个寄存器进行读取操作，当读取的寄存器中出现错误时，切换逻辑会交换读地址信号，从而实现对错误的屏蔽。在 FPGA 资源使用上，这种方法所需的错误屏蔽开销远低于传统的三模冗余。但是这种方法只能检测寄存器中的单比特错误，对于多比特同时出现错误的情况则无法处理。并且由于是对错误进行屏蔽，因此无法对错误进行真正的纠正，寄存器堆中的错误累积问题仍无法得到解决。

综上所述，目前针对寄存器堆的加固主要从空间冗余和信息冗余两个方面进行，寄存器堆位于关键路径上，为了减少对处理器运行的影响，出现错误时需要被及时纠正。双模冗余不具备纠错能力，而三模冗余会产生较多的面积开销，因此提出了一些基于三模冗余的动态加固方法。对寄存器堆使用信息冗余进行加固虽然可以对出现的错误进行纠正，但是纠错能力有限。另外，随着半导体制造工艺的不断提升，器件尺寸减少、供电电压降低，大大地降低了器件的噪声容限，导致辐射敏感度不断增加，寄存器堆等存储器的近邻单元之间的相互影响逐渐明显，这些问题都需要在对寄存器堆进行容错设计时考虑。

1.3 本文研究内容

本文的研究目标为完成基于 RISC-V 指令集的浮点单元的容错设计，具体的研究内容包含以下两个方面：

1. 对于浮点寄存器堆，针对目前浮点寄存器堆容错设计中出现的问题，设计一种高可靠的寄存器堆结构。这种寄存器堆结构易于实现，具有较低的访问延时，不会对系统关键路径产生严重影响，使用系统中的数据 Cache 来存放寄存器堆中的数据的备份，减少备份所带来的资源开销。纠错操作可以自动进行，解决寄存器堆中错误累积的问题。在校验器的设计上，使用了交叠编码技术，通过将连续多比特位拆分到不同的校验分组中进行校验，提高浮点寄存器堆对近邻连续位翻转错误的容错能力；
2. 对于浮点运算单元，从加固操作的可行性、加固后的容错能力提升等方面进行研究，分析浮点运算过程，针对浮点运算过程中不同阶段的特点，采用不同的容错方法。通过将算术检错码容错技术与浮点运算舍入操作进行结合，提高浮点运算的容错能力，并且不会影响浮点数的运算结果。对于算术检测码无法保护的电路，采用其他方式进行容错，从而在提升浮点运算单元容错能力的同时，减少因容错产生的额外的资源开销。优化乘积码解码器，提出

了一种基于查找表的快速除三电路，可以在单个时钟周期得到运算结果，并且产生较小的面积开销。

1.4 论文结构

针对上述研究内容，本文的组织结构如下：

第一章：绪论。本章首先介绍论文的研究背景与意义，其次调研了浮点单元容错和寄存器堆容错的国内外研究现状，最后介绍本文的研究目标与研究内容，以及文章的组织结构。

第二章：相关理论概述。本章将分别介绍 IEEE 754 浮点数标准、RISC-V 浮点扩展指令集、容错技术等相关理论内容，并对现有的容错技术进行分析和总结，为后文针对浮点寄存器堆和浮点运算单元进行容错设计奠定基础。

第三章：浮点寄存器堆容错研究。本章将介绍浮点寄存器堆的容错技术的研究与设计方法，首先介绍本文提出的基于 Cache 的寄存器堆容错结构，改进校验器使其可以完成对近邻连续多比特翻转错误的检测，其次设计测试平台对加固后的寄存器堆进行错误注入，最后通过多个关键指标的横向对比验证了性能和可靠性上的优势，并对不足进行分析。

第四章：浮点运算单元容错研究。本章将介绍浮点运算单元的容错技术的研究与设计方法，首先对浮点运算单元的可靠性进行分析，总结对浮点运算单元使用乘积码进行加固所面临的问题，然后介绍改进后的除三电路结构和加固后的浮点加法器的结构，最后设计测试平台对加固后的浮点加法器进行功能验证。

第五章：总结与展望。本章首先对本文的研究工作进行了总结，介绍了论文的创新点，最后对未来的研究进行了展望。

第二章　相关理论概述

浮点单元是完成浮点数运算的处理单元，广泛应用于科学计算、数据处理等领域。本章将对 IEEE 754 浮点数标准、RISC-V 浮点扩展指令集、容错技术分别进行介绍。

2.1 浮点数标准

浮点是一种数的表示方法，一个数被表示成尾数部分和指数部分，其数值为尾数乘以基数的指数幂。与定点数相比，浮点数可以在某个固定长度的存储空间中表示定点数无法表示的更大范围的数。浮点计算是指浮点数参与的计算，浮点计算伴随着因为无法精确表示而进行的近似和舍入操作。

目前应用最为广泛的浮点数标准为 IEEE 754 标准[25]，多款处理器中均使用此标准来设计浮点单元[14-15]。IEEE 754 中定义了单精度浮点数、双精度浮点数、半精度浮点数等多种浮点数格式，以支持不同的应用场景。宇航用处理器[14-15]支持单精度浮点数和双精度浮点数，格式如图 2.1 所示。

- 单精度浮点数：数据位宽为 32 位，包含 1 位符号位，8 位阶码和 23 位尾数。
- 双精度浮点数：数据位宽为 64 位，包含 1 位符号位，11 位阶码和 52 位尾数。

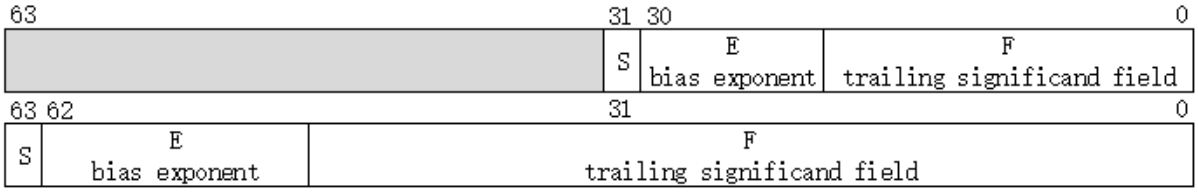

图2.1 IEEE 754 标准浮点数格式

其中，尾数包含整数部分和小数部分，存储时只存储小数部分，整数部分则通过阶码来隐含表示，整数部分的位宽为 1 位。

- 对于规格化小数，整数部分数值为 1，表示形式如下：

$$\pm 1.F \times 2^{E-bias}$$

其中，对于单精度浮点数，$bias = 127$；对于双精度浮点数，$bias = 1023$。

- 对于非规格化小数，整数部分数值为 0，表示形式如下：

$$\pm 0.F \times 2^{-(e_{max}-1)}$$

其中，对于单精度浮点数，$e_{max} = 127$；对于双精度浮点数，$e_{max} = 1023$。

针对单精度浮点数和双精度浮点数，其可表示的数值范围如图 2.2 所示。可以发现，浮点数越靠近 0，相邻两个浮点数间隔越小。

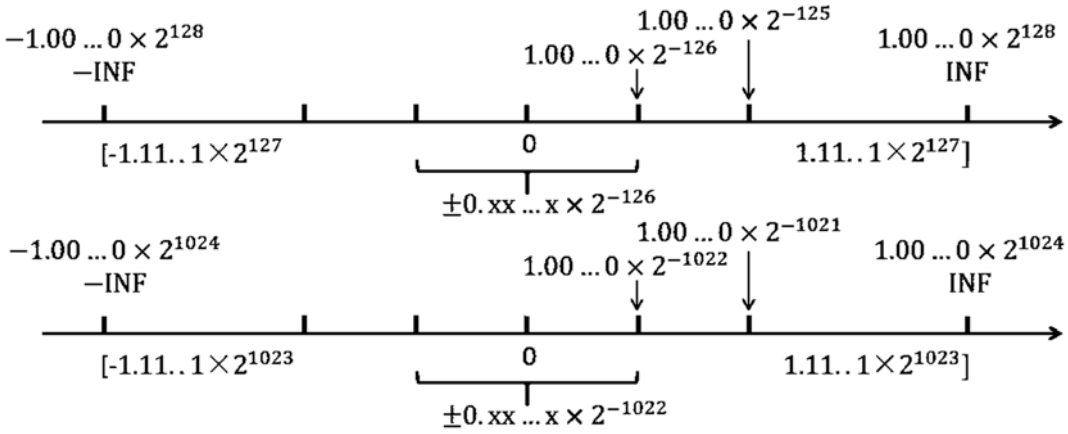

图2.2 IEEE 754 标准浮点数数值范围

IEEE 754 标准中定义了三种特殊数，如表 2.1 所示：0、Inf 和 NaN。其中，Inf 表示“无穷大”，根据符号位的不同分为 Inf 和-Inf；NaN 为“非数”，根据尾数最高位的不同分为 sNaN 和 qNaN，qNaN 用于表示未定义的算术运算结果，例如除零运算的结果，sNaN 用于标识未初始化的值，可以用来捕捉异常。

表2.1 IEEE 754 标准浮点数类型

类型	指数部分（带偏移值）	小数部分
0	0	0
非规格化小数	0	(0,1)
规格化小数	$1\sim2^{w}-2$	[1,2)
Inf	$2^{w}-1$	0
NaN	$2^{w}-1$	非 0

注：w 为阶码位的位宽，单精度浮点数 w=8，双精度浮点数 w=11。

2.2 浮点指令集

RISC-V 是一种新兴的开源精简指令集架构，由加州大学伯克利分校在 2010 年首次发布[6]。RISC-V 建立在现有的体系结构长期发展所暴露出的种种问题之上，顺应现代信息系统设计需求和体系结构发展趋势。目前，RISC-V 提供了三个浮点扩展指令集，兼容 IEEE 754-2008 算术标准，分别是单精度浮点扩展（F 扩展）、双精度浮点

扩展（D 扩展）、四精度浮点扩展（Q 扩展）[7]。在 32 位 RISC-V 指令集架构中，RV32F 扩展指令包含 26 条指令，RV32D 扩展指令包含 26 条指令[6]。RV32FD 扩展指令格式如图 2.3 所示，指令位宽为 32 位。图 2.3 中的红色标识的比特位用于区分 RV32F 和 RV32D 指令，同时也可以看到 RISC-V 的指令格式较为整齐，极大的简化了指令译码器的设计。

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
FADD.S	0000000		fs2	fs1	rm	fd	1010011
FSUB.S	0000100		fs2	fs1	rm	fd	1010011
FMAX.S	0010100		fs2	fs1	001	fd	1010011
FMIN.S	0010100		fs2	fs1	000	fd	1010011
FEQ.S	1010000		fs2	fs1	010	fd	1010011
FLE.S	1010000		fs2	fs1	000	fd	1010011
FLT.S	1010000		fs2	fs1	001	fd	1010011
FCLASS.S	1110000		00000	fs1	001	fd	1010011
FSGNJ.S	0010000		fs2	fs1	000	fd	1010011
FSGNJN.S	0010000		fs2	fs1	001	fd	1010011
FSGNJX.S	0010000		fs2	fs1	010	fd	1010011
FMV.W.X	1111000		00000	fs1	000	fd	1010011
FMV.X.W	1110000		00000	fs1	000	fd	1010011
FCVT.S.W	1101000		00000	fs1	rm	fd	1010011
FCVT.S.WU	1101000		00001	fs1	rm	fd	1010011
FCVT.W.S	1100000		00000	fs1	rm	fd	1010011
FCVT.WU.S	1100000		00001	fs1	rm	fd	1010011
FMUL.S	0001000		fs2	fs1	rm	fd	1010011
FMADD.S	fs3	00	fs2	fs1	rm	fd	1000011
FMSUB.S	fs3	00	fs2	fs1	rm	fd	1000111
FNMADD.S	fs3	00	fs2	fs1	rm	fd	1001111
FNMSUB.S	fs3	00	fs2	fs1	rm	fd	1001011
FDIV.S	0001100		fs2	fs1	rm	fd	1010011
FSQRT.S	0101100		0000	fs1	rm	fd	1010011
FLW	imm12[11:0]			fs1	010	fd	0000111
FSW	imm12[11:5]		fs2	fs1	010	imm12[4:0]	0100111

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
FADD.D	0000001		fs2	fs1	rm	fd	1010011
FSUB.D	0000101		fs2	fs1	rm	fd	1010011
FMAX.D	0010101		fs2	fs1	001	fd	1010011
FMIN.D	0010101		fs2	fs1	000	fd	1010011
FEQ.D	1010001		fs2	fs1	010	fd	1010011
FLE.D	1010001		fs2	fs1	000	fd	1010011
FLT.D	1010001		fs2	fs1	001	fd	1010011
FCLASS.D	1110001		00000	fs1	001	fd	1010011
FSGNJ.D	0010001		fs2	fs1	000	fd	1010011
FSGNJN.D	0010001		fs2	fs1	001	fd	1010011
FSGNJX.D	0010001		fs2	fs1	010	fd	1010011
FCVT.D.W	1101001		00000	fs1	rm	fd	1010011
FCVT.D.WU	1101001		00001	fs1	rm	fd	1010011
FCVT.W.D	1100001		00000	fs1	rm	fd	1010011
FCVT.WU.D	1100001		00001	fs1	rm	fd	1010011
FCVT.D.S	0100001		00000	fs1	rm	fd	1010011
FCVT.S.D	0100000		00000	fs1	rm	fd	1010011
FMUL.D	0001001		fs2	fs1	rm	fd	1010011
FMADD.D	fs3	01	fs2	fs1	rm	fd	1000011
FMSUB.D	fs3	01	fs2	fs1	rm	fd	1000111
FNMADD.D	fs3	01	fs2	fs1	rm	fd	1001111
FNMSUB.D	fs3	01	fs2	fs1	rm	fd	1001011
FDIV.D	0001101		fs2	fs1	rm	fd	1010011
FSQRT.D	0101101		0000	fs1	rm	fd	1010011
FLD	imm12[11:0]			fs1	011	fd	0000111
FSD	imm12[11:5]		fs2	fs1	011	imm12[4:0]	0100111

图2.3 RV32FD 扩展指令指令格式

如表 2.2 所示，RV32FD 扩展指令集指令可以划分为算术运算指令、比较指令、访存指令、转换指令、符号注入指令、分类指令。其中，访存指令、符号注入指令、分类指令不需要进行算术运算，除访存指令外，其他指令均可在一个时钟周期内完成；算术运算指令、比较指令、转换指令需要进行算术运算，通常需要多个时钟周期。在需要算术运算的指令中，乘法、乘加、乘减指令需要进行乘法运算，除法指令、开方指令通过除法运算实现，其他指令则需要通过加法器来完成运算操作。浮点单元中完成算数运算的运算单元包含浮点加法器、浮点乘法器、浮点除法器，对于特殊浮点数之间的运算，由于运算结果是固定的，因此可以通过选择器来输出运算结果。

RISC-V 在指令的设计上遵循着简洁的设计原则，仅保留了最基本的指令，复杂的操作可以通过多条简单指令来实现，例如，通过符号注入指令 FSGNJ.S/FSGNJ.D 即可以完成浮点寄存器之间数据的转移，因此浮点寄存器之间的数据转移操作可以通过上述指令进行。对于一些浮点数算法，例如矩阵乘法，乘法操作之后会立即执行一次加法操作，如果使用两条指令，整个运算过程中将涉及到两次舍入（乘法操作和加法操作中各一次）。RV32FD 扩展指令集提供乘加、乘减指令，使用一条指令融合乘法和加/减法过程，整个运算过程中只有一次舍入操作，相比于使用两条独立指令，可以减少两次舍入操作所带来的运算误差。

表2.2 RV32FD 扩展指令集指令

类别	RV32F 扩展指令集	RV32D 扩展指令集
算术运算指令	FADD.S、FSUB.S FMUL.S、FDIV.S、FSQRT.S FMADD.S、FMSUB.S FNMADD.S、FNMSUB.S	FADD.D、FSUB.D FMUL.D、FDIV.D、FSQRT.D FMADD.D、FMSUB.D FNMADD.D、FNMSUB.D
	FMAX.S、FMIN.S	FMAX.D、FMIN.D
比较指令	FEQ.S、FLT.S、FLE.S	FEQ.D、FLT.D、FLE.D
访存指令	FLW、FSW	FLD、FSD
转换指令	FCVT.W.S、FCVT.WU.S FCVT.S.W、FCVT.S.WU	FCVT.W.D、FCVT.WU.D FCVT.D.W、FCVT.D.WU
		FCVT.D.S、FCVT.S.D
	FMV.X.W、FMV.W.X	
符号注入指令	FSGNJ.S、FSGNJN.S、 FSGNJX.S	FSGNJ.D、FSGNJN.D、 FSGNJX.D
分类指令	FCLASS.S	FCLASS.D

2.3 容错技术

目前常用的容错技术的主要形式是进行冗余设计，分为三种类型：时间冗余、空间冗余和信息冗余。时间冗余重复执行指令操作，指令的重复会增加指令执行的周期。空间冗余使用多个相同的硬件电路，并在这些电路中同时执行相应操作，通过动态故障检测机构（例如表决器）来保证系统的正确输出，硬件电路的重复将导致功耗和面积的增加。信息冗余添加额外的编码数据，例如 EDC（Error Detection Code）和 ECC（Error Correction Code），面积开销和功耗低于空间冗余，由于添加了额外的编码和解码逻辑，可能会出现电路时序问题。以下是一些经典和常用的容错方法。

2.3.1 三模冗余

三模冗余是一种空间冗余的加固方法，该结构由三个冗余并行模块和一个表决器组成[26]。这三个模块通过表决系统将多数结果作为正确结果进行输出，即三取二。除非其中两个模块的相同位置出现相同的错误，否则错误都可以被屏蔽，从而保证系统的正确输出。Velazco 等人[27]统计了一些 SRAM 存储器的位翻转情况，观察到只出现少数多比特位翻转的情况，因此三模冗余中两个模块同时发生错误的概率非常小，在两个模块的同一位置同时发生错误的概率更小。三模冗余的结构如图 2.4 所示。

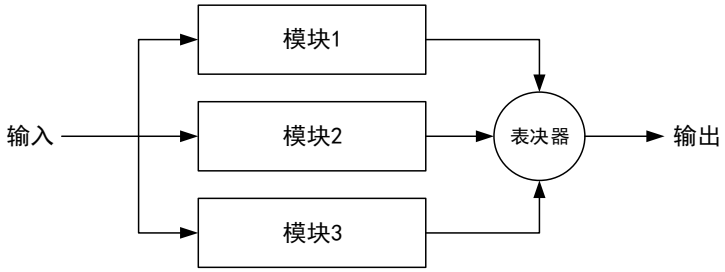

图2.4 三模冗余模型

三模冗余是一种最常用的冗余加固方法，三模冗余后的系统可以对内部的错误进行屏蔽，通过三选二表决器输出正确数据，但是不会对出错的模块中的错误进行纠正。这种冗余方式可以应用于组合逻辑电路，也可以对时序电路中的触发器进行三模冗余，对触发器的输出进行结果表决，触发器可以使用三个相位偏移的时钟，可以减少单粒子瞬态故障带来的影响。

表决器也容易出现位翻转的情况，对此，Katz 等人[28]提出了一种带有反馈机制的三表决器的三模冗余方法。此外，三模冗余对于占用空间小的内存结构更有效，不适用于占用空间较大的存储结构[29]。

2.3.2 双模冗余

双模冗余可以对组合逻辑和时序逻辑中的错误进行检测，基于时间冗余的技术常用于检测组合逻辑中的瞬时故障，而空间冗余有助于识别时序逻辑中的单粒子翻转错误。采用时间冗余时，其目标是利用粒子撞击时产生的瞬时脉冲特征来比较两个不同时刻的输出信号，组合逻辑在两个不同的时刻锁存，比较器可以检测出瞬时脉冲的出现。采用空间冗余时，双模冗余可以对时序逻辑中的单粒子瞬态错误和时序逻辑中的翻转错误进行检测，其模型如图 2.5 所示。

与三模冗余相比，双模冗余消耗的资源少，但是只能对错误进行检测，无法对出现的错误进行纠正或者屏蔽，并且在使用时间冗余方法对组合逻辑电路进行加固时，需要对单粒子瞬态的持续时间进行考虑。

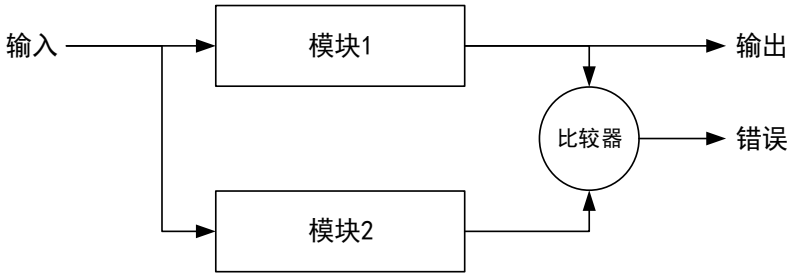

图2.5 双模冗余模型

2.3.3 奇偶校验

奇偶校验是一种基于 EDC 编码的错误检测方法[30-31]。奇偶校验有两种类型，即偶校验和奇校验，如图 2.6 所示。奇偶校验对数据位中的数据进行校验，生成校验码，偶校验确保数据位中的“1”的个数为偶数，奇校验确保“1”的个数为奇数。如果翻转奇数个数据位，校验值将改变，表明传输过程中出现错误。当出现偶数个位翻转时，该检测机制将失效。由于奇偶校验是一种错误检测码，无法确定哪些位被翻转，因此，一旦检测到错误，必须丢弃接收的数据，重新运行传输。

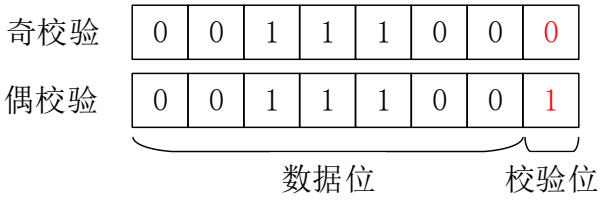

图2.6 奇校验和偶校验模型

2.3.4 Hamming 码

Hamming 码是最简单的信息冗余编码，可用于检测和纠正数据中产生的位错误[32]。由于 Hamming 码的简单性，它们被广泛用于存储单元的加固[33]。Hamming 码[34]利用奇偶校验位的概念，在数据位之后添加一些位来验证数据。通过使用特殊设置的校验位，Hamming 码不仅可以验证数据的有效性，还可以在数据出错时指示错误的位置并纠正。Hamming 码编码需要满足以下关系：$2^k \geq m+k+1$，其中 m 为原码字的位数，k 是码字中检测位的位数，m+k 为码字中的全部位数。Hamming(m+k,m)编码和解码矩阵如图 2.7 和图 2.8 所示，以 Hamming(12,8)为例，其中校验位据附加在编码后数据的末尾。

$$G=\begin{bmatrix}1&0&0&0&0&0&0&0&1&1&0&0\\0&1&0&0&0&0&0&0&1&0&1&1\\0&0&1&0&0&0&0&0&1&0&1&0\\0&0&0&1&0&0&0&0&1&0&0&1\\0&0&0&0&1&0&0&0&0&1&1&1\\0&0&0&0&0&1&0&0&0&1&1&0\\0&0&0&0&0&0&1&0&0&1&0&1\\0&0&0&0&0&0&0&1&0&0&1&1\end{bmatrix}$$

图2.7 Hamming(12,8)编码矩阵

$$H=\begin{bmatrix}1&1&1&1&0&0&0&0&1&0&0&0\\1&0&0&0&1&1&1&0&0&1&0&0\\0&1&1&0&1&1&0&1&0&0&1&0\\0&1&0&1&1&0&1&1&0&0&0&1\end{bmatrix}$$

图2.8 Hamming(12,8)解码矩阵

此外，常见的信息冗余方法有 RM 码[35]（Reed-Muller 码）、RS 码[36]（Reed-Solomon 码）、BCH 码[37]（Bose-Chaudhuri-Hocquenghem 码）等。与 Hamming 码相比，RM、RS 和 BCH 码具有更复杂的逻辑和更高的纠错和检测能力，因此更可靠，但其编码和解码逻辑更复杂，一般用于对内存等低速存储设备进行加固。

2.4 本章小结

本章介绍了浮点数在现代计算机中的表示方法，详细介绍了 IEEE 754 浮点数标准下单精度浮点数、双精度浮点数的表示形式、数值范围、特殊数，并对特殊数的使用场景进行了说明。

其次，介绍了新兴的 RISC-V 架构中的浮点数扩展指令集 RV32FD 中的浮点指令，对浮点指令进行分类、分析，将浮点指令根据是否需要进行运算划分为需要进行算术运算操作的浮点指令和不需要进行算术运算操作指令。对于浮点数算术运算操作，由于其具有较长的关键路径，很难在单个时钟周期内完成，因此现代计算机中对于浮点数运算操作的实现，采用切分多个时钟周期的方式进行实现。

最后对典型容错技术进行介绍，三模冗余是一种常用的空间冗余方法，优点是结构简单，通过表决器来保证系统的正确输出。双模冗余可以对组合逻辑和时序逻辑中的错误进行检测，基于时间冗余的技术常用于检测组合逻辑中的瞬时故障，而空间冗余有助于识别时序逻辑中的单粒子翻转错误。检测码和纠错码是通过信息冗余的方式来对系统进行保护，相比于三模冗余等空间冗余方式，资源开销小，但是编码电路、解码电路会对系统的关键路径产生一定影响。

第三章 浮点寄存器堆容错研究

浮点寄存器堆（FPRF，Floating-point Register File）用于存储浮点运算的操作数和运算结果，针对浮点寄存器堆的容错设计，本文提出了一种基于 Cache 结构的寄存器堆容错结构，即 C-DMR，使用双模冗余对寄存器堆中的数据进行备份。与传统双模冗余设计不同，C-DMR 中的寄存器堆与其备份是异构的，寄存器堆数据的备份存放在数据 Cache 中，目前针对 Cache 已经有较为成熟的容错方案[38]，因此备份在 Cache 中的寄存器堆数据可以认为是正确的（备份在 Cache 中的数据中的错误可以通过 Cache 的容错机制得到纠正），当寄存器堆中出现错误时，自动从 Cache 中读取备份数据对寄存器进行数据恢复。在检错与纠错策略的设计上，每次读寄存器时会对全部寄存器进行检查，产生纠错序列，依次对出错的寄存器进行数据恢复。校验算法使用分段交叠编码对寄存器中的数据进行分组校验，从而可以检测数据中的近邻连续多比特翻转错误。

本章的最后介绍了针对基于 Cache 结构的寄存器堆方法进行的实验的结果，从可靠性、路径延时、资源开销、对 Cache 性能的影响等方面对实验结果进行了分析。

3.1 浮点寄存器可靠性分析

在计算机中，处理器的运行速度远快于存储器的访问速度，处理器运算时的中间结果被暂时保存在寄存器中，只有在加载操作数和运算完成后才会对存储器进行访问，这样可以避免由于多次读写存储器所带来的延时。但是这种策略也带来一些弊端，在空间环境中，寄存器堆中的数据与存储器中的数据一样，同样容易受到高能粒子的影响，为了减少粒子翻转带来的影响，必须对寄存器堆进行加固处理。由于寄存器的作用是加速计算机的运行，寄存器堆往往位于系统的关键路径上，因此对其进行加固处理会增加关键路径的长度，增加了数据传输的延时，严重阻碍了处理器的运行速度的提高。在寄存器堆的加固设计上，需要在保证容错能力的前提下减少容错带来的面积开销和路径延时的增加。

目前国内外研究学者对寄存器堆的加固大多采用 Hamming 编码进行容错[21-23]，只对寄存器输出的数据进行纠正，纠正后的正确数据不会写回到寄存器中，同时为了加速 Hamming 解码器的解码过程，不会对出现在校验码中的错误进行纠正（使用 Hamming 码进行编码后，编码后的数据由有效数据和校验码两部分组成），这种容错策略存在一些不足，数据驻留寄存器期间在不同位置上先后发生两次翻转错误时，第一次出现的错误可以被检测到并纠正，但由于纠正后的数据没有写回到寄存器中，因

此错误仍会驻留在寄存器中，导致第二次出现错误时无法进行纠正。在 3.1.1 节中将使用标准测试集对浮点寄存器的访问间隔进行统计，对浮点寄存器的访问间隔进行定量分析。

另外，现代处理器中的浮点单元主要是采用 IEEE 754 标准进行设计，浮点数由符号位、阶码、尾数三部分组成，浮点数的表示如公式 3-1 所示，其中 sign 表示符号位，expn 表示带偏移的阶码，bias 表示阶码的偏移，frac 表示尾数的小数部分，不包含整数位。

$$(-1)^{sign} \times (0 \text{ or } 1).\text{frac} \times 2^{expn-bias} \tag{3-1}$$

浮点运算涉及到舍入操作，例如在进行浮点加法运算时，两个具有相同阶码的浮点数才能对尾数进行运算，因此需要调整数值较小的浮点数的阶码值，同时对其尾数进行右移，右移时会将尾数的低位移动到舍入区，舍入区中的数据通过或运算进行融合，当错误出现在此区域中时，或运算会屏蔽掉这些错误，从而使得错误不会对运算结果产生影响。另外，浮点运算的舍入操作模式也可以对运算结果进行修正，但是这种修正具有一定的局限性，只对特定位置上出现的错误有效，并且还与浮点运算时所选择的舍入模式有关。在 3.1.2 节中将对浮点数不同区间出现单比特错误时的影响进行实验分析。

3.1.1 浮点寄存器堆访问间隔统计

与通用寄存器作用相似，浮点寄存器用于存放浮点运算时的操作数和运算结果，目前的研究工作中对于程序运行期间浮点寄存器内数据的保持时间还没有进行过相应的统计。浮点单元是一个算术运算部件，理论上浮点寄存器堆不会存在类似 R9 寄存器（ARM 架构中用于存放数据段基地址，操作系统在加载程序时进行设置）和 GP 寄存器（RISC-V 架构中用于实现对内存的单指令访问）这样的需要对数据进行长时间保持的寄存器，但是这种假设需要通过实验来进行定量分析。本节将通过实验对 RISC-V 架构处理器中的浮点寄存器写后读间隔进行统计，分析浮点寄存器内数据的保持时间。

测试平台基于 T-Head 的 E906 内核[39]进行搭建，支持 RV32IMACFD 指令集，测试平台的系统结构如图 3.1 所示：AHB_ICM 与 HB1 组成 AHB 总线矩阵，I-SRAM 和 D-SRAM 通过 AHB-Lite 总线（IAHB-Lite 和 DAHB-Lite）与内核进行连接，分别存放运行的软件测试程序和数据。GPIO 和 UART 外设通过 H2P1 桥设备连接到内核的 SAHB-Lite 总线上。测试程序的执行状态通过 UART 与监视模块进行交互，浮点寄存器堆的访问操作则直接由监视模块进行统计与分析。

软件测试程序使用 MiBench[40]的 basicmath 测试集，该测试集支持对浮点数加、减、乘、除、开方运算的测试。

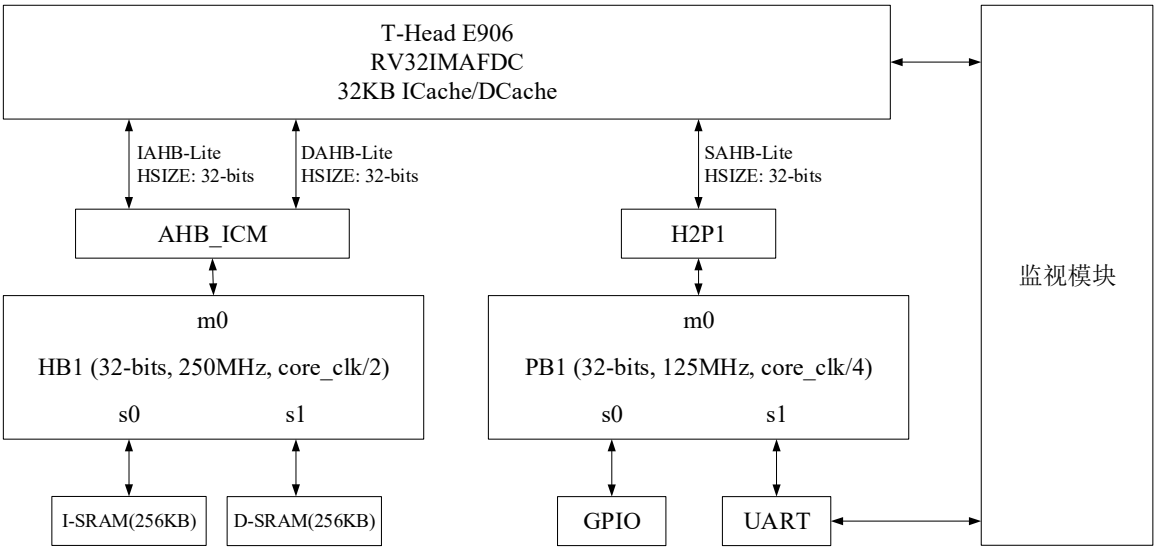

图3.1 测试平台结构（基于 E906 内核）

浮点寄存器写后读间隔统计程序逻辑流程图如图 3.2 所示，统计程序对 E906 中的浮点寄存器的访问操作进行监视，RISC-V 处理器中不同的指令对浮点寄存器进行的操作可分为读操作、写操作、空闲操作，其中只有读操作和写操作才会对浮点寄存器进行访问。统计程序中申请了两个数组：wfreg[32]和 rfreg[32]，分别用于记录每个浮点寄存器距离上一次写入和读取所经历的时钟数，数组下标为寄存器序号。当浮点寄存器被读取时，对应寄存器的 wfreg 和 rfreg 的值会被打印输出，然后进行复位，具体执行过程如下：

1. 初始化 wfreg 和 rfreg 数组；
2. 加载测试程序并开始运行；
3. 判断测试程序是否执行完成；
   a) 执行结束，执行步骤 6；
   b) 执行未结束，执行步骤 4；
4. 判断当前指令是否对浮点寄存器进行访问（即是否对浮点寄存器进行读写操作）；
   a) 对浮点寄存器进行写入操作时，对应的 wfreg 复位；
   b) 对浮点寄存器进行读取操作时，对应的 rfreg 复位，复位前先对 rfreg 和 wfreg 进行打印；
5. 执行步骤 3；
6. 结束。

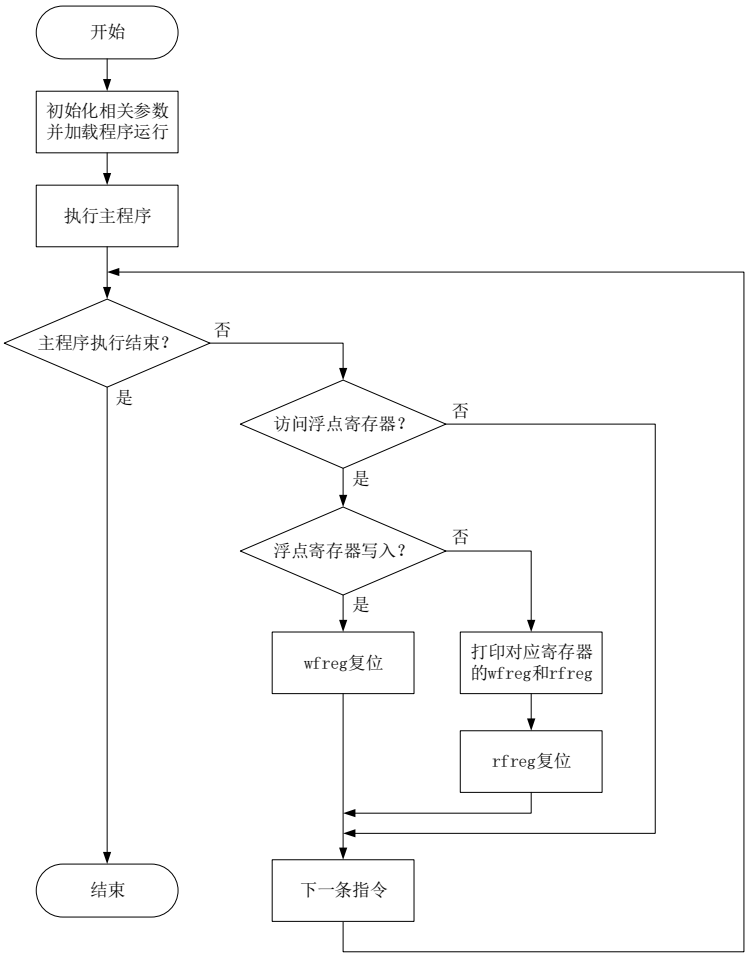

图3.2 浮点寄存器写后读间隔统计程序执行流程

软件测试程序的编译器使用 GCC[41]，这是一个开源的编程语言编译器，支持多种优化等级，优化等级的说明如表 3.1 所示。

表3.1 GCC 编译器优化等级

优化等级	说明
-O0	不进行优化，调试时可产生预期效果
-O1	减少代码大小和执行时间，不执行任何需要大量编译时间的优化
-O2、-O3	进一步优化，生成的代码性能提升，需要消耗大量的编译时间
-Os	在-O2 的基础上减少了生成的代码的大小

浮点寄存器的平均写后读间隔统计结果如图 3.3 所示，按照时钟 Tick 进行了分组。为方便观测，根据处理器的内核时钟频率进行了单位换算，处理器内核时钟为 500MHz，换算关系如表 3.2 所示。

表3.2 时钟换算关系（内核时钟 500MHz）

内核时钟	Tick	时间
500MHz	100	200ns
	1000	2us
	10000	20us
	10^8	200ms

从图 3.3 中的实验结果可以看到，优化等级 O2 和 O3 下的浮点寄存器的平均写后读间隔的结果基本一致，这是因为优化等级 O3 包含了 O2 的全部优化选项，仅额外增加了与循环控制相关的优化选项，这些选项与浮点寄存器的优化的相关性很小，因此没有对测试结果产生较大影响。

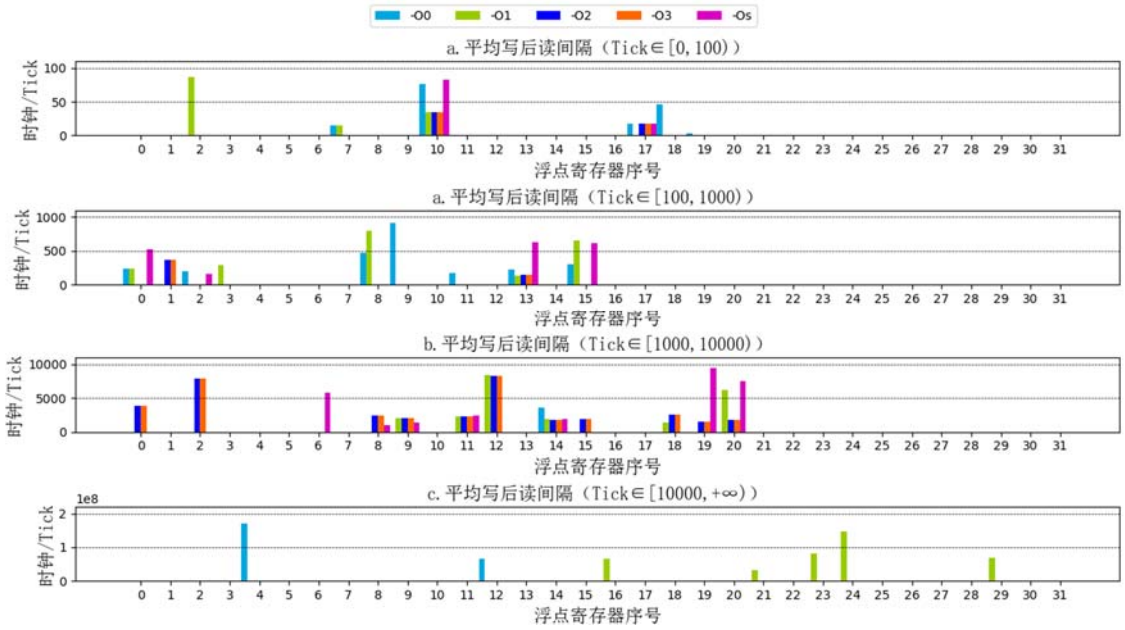

图3.3 浮点寄存器的平均写后读间隔统计

如图 3.3 中所展示的，处理器内核时钟 500MHz，在 MiBench 测试集下，不同优化等级下的浮点寄存器的平均写后读间隔基本保持在 20us 以内（Tick＜10000），特别地，优化等级 O0 和优化等级 O1 下的部分浮点寄存器的平均写后读间隔超过了 200ms（Tick>10^8）。优化等级 O2 和优化等级 O3 的结果分布基本是一致的，相比其他优化等级，使用了更多的浮点寄存器，这是因为这两个优化等级为了加速运算，会将操作数提前加载到浮点寄存器中。优化等级 O0 和优化等级 O1 下浮点寄存器的平均写后读间隔在 2us 以内（Tick＜1000）的频次更高，这是因为这两个优化等级会频繁地进行存储器访问操作，每次运算前都会从存储器中加载数据到浮点寄存器中，浮

点寄存器会被频繁的写入。从实验结果中可以看到，增加优化等级的级别会增加浮点寄存器的平均写后读间隔，并且浮点寄存器的平均写后读间隔时间较长，在数据驻留浮点寄存器期间，空间中的高能粒子足以造成数据位的翻转。

以上实验是基于 MiBench 测试集进行的，运算较为密集并且完全由编译器进行优化，浮点寄存器的平均写后读间隔没有超过 400ms（处理器内核工作在 500MHz），但是在实际应用中，浮点寄存器的平均写后读间隔可能会远远超过这个数值，如表 3.3 所示，定义了一个向量点乘的函数 calc，其功能是完成 1024 个浮点数的点乘运算，实际上进行运算的元素数量可以通过函数的参数来确定，在表 3.3 所示的例子中，运算的元素的数量为 1024。在编译器的 O2 优化下，从函数 calc 的汇编代码可以看到，浮点寄存器 fa0 中的数据只会被读取，浮点寄存器 fa0 中的数据在函数 calc 被调用前写入，在函数 calc 的执行期间不会更新其中的数据，浮点寄存器 fa0 的写后读间隔与向量 Array 中的元素个数成正比。

表3.3 浮点寄存器数据保持时间分析

源代码	汇编代码
1 float Array[1024], Num; 2 calc(Array, Num);	<calc>: 1 lui a5,0xa 2 addi a5,a5,-960 3 add a5,a5,a0 4 flw fa5,0(a0) 5 addi a0,a0,4 6 fmul.s fa5,fa5,fa0 7 fsw fa5,-4(a0) 8 bne a0,a5,532 <calc+0x8> 9 ret

在嵌入式应用中，由于嵌入式处理器的主频较低、资源有限，较高的编译优化可以减少指令数量、减少访存延时，因此可以减少最终得到的可执行文件的大小，同时指令数量的减少、访存延时的降低可以提高嵌入式处理器的执行效率。很多程序中还会出现由工程人员手动优化的情况，这种情况下可能会进一步增加寄存器的平均写后读间隔。

3.1.2 浮点数故障敏感性评估

浮点数不同区间的故障敏感性评估实验使用 MiBench 测试集进行测试，错误注入方式为每次进行浮点运算时向其中一个操作数中的随机位置注入 1 位错误，测试结果如图 3.4 所示。图 3.4 中的横轴表示浮点数的不同区间，例如，[0-7]表示浮点数的 bit0~bit7 区间。误差计算公式如公式 3-2 所示，根据错误注入后运算结果与标准结果（未进行错误注入时的运算结果）的误差大小将测试结果划分为 equal、0~0.0001、0.0001~1、error。

$$\varepsilon = \left| \frac{注入故障后的结果 - 标准结果}{标准结果} \right| \tag{3-2}$$

其中，误差小于 10^{-7} 记为 equal，误差大于 1 记为 error，其中只有 equal 分组中的运算结果在错误注入后是正确的。

从图 3.4 中可以看到，在[0-31]区间中进行错误注入无法得到正确结果，正确率为 0，完全错误（error，即误差大于 1）的几率为 35.3%。在[0-7]区间中进行单比特错误注入时仍有 49.2%的正确率，这是因为这部分区间位于尾数的低位，浮点数在运算过程中对尾数的移位操作、舍入操作可以将此区间中出现的错误屏蔽，但这要求两个操作数的大小关系可以保证在阶码对齐阶段可以将出错的区间移动到舍入区，具有一定的局限性，这一局限性从[0-15]区间的实验结果中可以得到体现，该区间内进行错误注入造成正确率下降了 17.6%。相比之下，在[24-31]区间内注错造成了 70.7%的错误率，这是因为这部分区间包含了浮点数的符号位和阶码的高位，出现错误时对运算结果的影响较大。

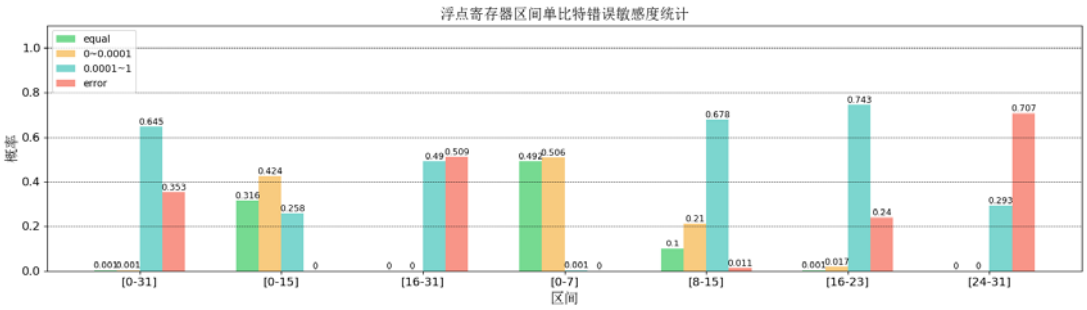

图3.4 浮点寄存器区间单比特错误敏感度统计

3.2 浮点寄存器容错设计

上一节中从浮点寄存器堆数据的保持时间和浮点运算中单比特错误对浮点数不同区域的影响进行了分析。

从浮点寄存器访问间隔的统计结果中可以看到，多数浮点寄存器的平均写后读间隔小于 20us，在优化等级 O0 和优化等级 O1 下，部分浮点寄存器的平均写后读间隔超过了 200ms。操作数会在浮点寄存器中保持较长时间，在数据保持期间，空间中的高能粒子会导致浮点寄存器堆出现错误，因此需要对浮点寄存器堆进行容错设计。另外，随着半导体制造工艺的不断提升，器件尺寸减少、供电电压降低，大大地降低了器件的噪声容限，导致辐射敏感度不断增加，寄存器堆等存储器的近邻单元之间的相互影响逐渐明显，而现有的针对寄存器堆的加固方法无法实现对近邻连续多比特错误的检测。

从浮点数故障敏感性评估实验可以发现，浮点数本身具有一定的容错能力，这是由于浮点数的格式、特有的运算过程所带来的，从图 3.4 的实验结果可以看到在浮点数尾数的低位中出现的单比特错误对最终的运算结果产生的影响较小，这种影响取决于具体的浮点运算以及操作数之间的大小关系，依赖浮点数本身的特性来规避错误所产生影响具有一定的局限性，例如当浮点加法运算的两个操作数阶码相同，且尾数加法没有产生溢出时，运算过程中不会有舍入操作发生，此时错误将导致运算结果出现较大误差。

基于上述的实验与研究，本文提出一种基于 Cache 的寄存器堆结构，使用双模冗余对寄存器堆中的数据进行备份。与传统双模冗余设计不同，在本文所提出的寄存器容错结构中，寄存器堆与其备份是异构的，寄存器堆数据的备份存放在系统中的数据 Cache 中，Cache 通过自身的容错机制与内存数据进行交互，以确保存储在 Cache 中的数据的正确性，因此备份在 Cache 中的寄存器堆数据可以认为是正确的，当寄存器中的错误被检测到，将自动从 Cache 中读取备份数据对寄存器进行数据恢复，进而实现对寄存器中错误的纠正。在检错与纠错策略的设计上，每次读寄存器时会对全部寄存器进行检查，产生纠错序列，依次对出错的寄存器进行数据恢复。校验算法使用分段交叠编码（即 interleved 技术[42]）的方式对寄存器中的数据进行分组校验，从而可以检测数据中的近邻连续多比特翻转错误。相比于其他的寄存器堆加固结构，本文提出的这种寄存器堆中检错和纠错的代价更小，检错和纠错是自动进行的，可以有效减少寄存器中的错误累积。

本节将对基于 Cache 结构的寄存器堆和基于交叠编码技术的校验器结构进行介绍。

3.2.1 基于 Cache 结构的寄存器堆

图 3.5 展示了对基于 Cache 结构的寄存器堆的读写操作访问流程，本文所设计的寄存器堆容错结构可以应用于通用寄存器堆和浮点寄存器堆，下面将以浮点寄存器堆为例对工作流程进行说明。

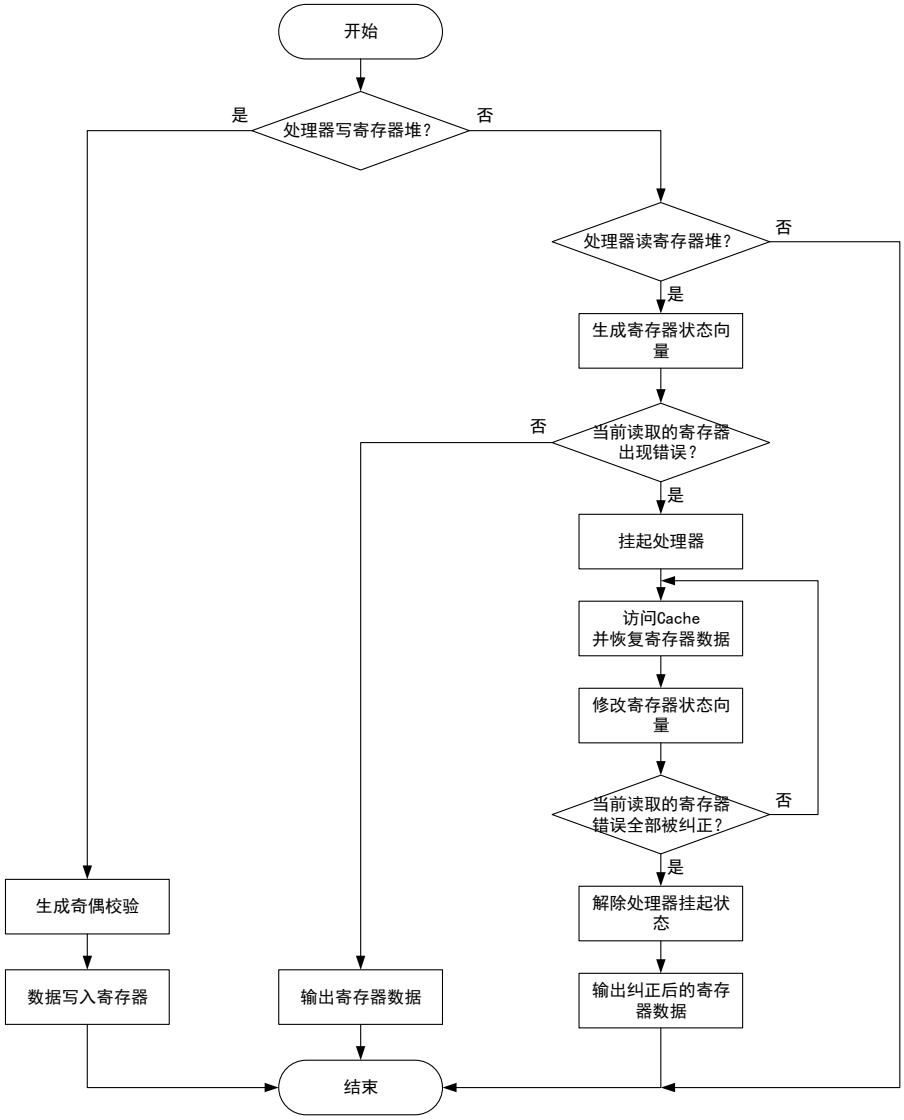

图3.5 基于 Cache 结构的浮点寄存器堆的读写操作访问流程

首先，判断处理器对浮点寄存器进行的操作的类型：

1. 如果是写操作，则生成校验（以奇偶校验为例），然后将待写入的数据和生成的校验值一起保存；
2. 如果是读操作，则进行检错操作，获取所有浮点寄存器的状态，并根据不同浮点寄存器状态进行相应操作；
   a) 当前操作读取的寄存器中出现错误时，挂起处理器，屏蔽其他寄存器中的错误，即优先对当前读取的寄存器的错误进行纠正。由于处理器同时可以对多个浮点寄存器进行读取操作，而每个时钟周期最多只能对一个寄存器中的错误进行纠正，因此当读取的多个寄存器中都出现错误时，会根据寄存器序号进行排序，优先对寄存器序号较小的寄存器中的错误进行纠正；

b) 当前操作读取的寄存器中未出现错误时，此时纠正操作与读取操作不会互相影响；

i. 如果其他寄存器中出现错误时，会根据寄存器序号进行排序，优先对寄存器序号较小的寄存器中的错误进行纠正；

ii. 如果其他寄存器中未出现错误时，则不会进行纠错操作。

3. 当寄存器堆未被处理器访问，且寄存器堆中的错误未被全部纠正，则根据寄存器状态向量逐个对出错的寄存器进行数据恢复。

1. 寄存器堆容错结构

寄存器堆容错结构如图 3.6 所示，由三部分组成，分别是寄存器堆、Cache、旁路逻辑，寄存器堆支持同时读取两个寄存器或对一个寄存器进行数据写入。寄存器堆由控制器、地址译码器、寄存器阵列组成，用于存储数据、对数据进行编码和校验。旁路逻辑由比较器和多路选择器 MUX 组成，用于对数据输出进行加速和产生处理器挂起信号 hold。Cache 用于存放寄存器数据的备份，位于系统的数据 Cache 中，是系统中的已有部件。

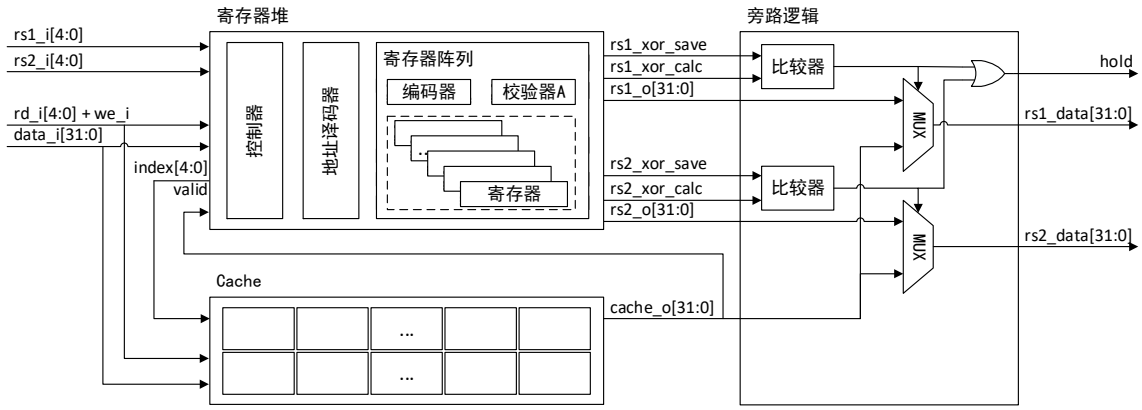

图3.6 寄存器堆容错结构

图 3.6 中的寄存器堆与正常的寄存器堆在功能上基本一致，可以实现对数据的存储。此外，本文中设计的寄存器堆在数据写入时生成校验值并保存，在读取相应寄存器时生成对应寄存器的校验值，与数据、写入时生成并保存的校验值一起输出到旁路逻辑中。

寄存器堆输出的数据有两个来源，分别是寄存器阵列和 Cache，旁路逻辑用于对数据输出进行选择，选择信号是一个单比特的二值信号，来自于寄存器写入时产生并保存的校验值与读取时生成的校验的比较结果。同时，旁路逻辑还会产生处理器挂起信号 hold，该信号用于通知系统进入挂起状态，挂起信号 hold 由两个比较器输出的结果经过或运算得到，即 rs1_i 和 rs2_i 对应的寄存器中有一个数据校验出错，就会产

生 hold 信号。

寄存器堆容错结构中的信号分为内部信号和外部信号，外部信号用于与处理器的其他部件进行交互，内部信号仅用于在寄存器堆内部进行信息传输，对外部不可见，信号的详细说明如表 3.4 所示（表中“-”表示内部信号，不区分方向）。

表3.4 寄存器堆容错结构信号列表

名称	位宽/bit	方向	信号类型	说明
rs1_i	5	输入	外部信号	寄存器读地址 1
rs2_i	5	输入	外部信号	寄存器读地址 2
rd_i	5	输入	外部信号	寄存器写地址
we_i	1	输入	外部信号	寄存器写使能
data_i	32	输入	外部信号	寄存器写数据
index	5	-	内部信号	Cache 访问地址
valid	1	-	内部信号	Cache 写有效
cache_o	32	-	内部信号	Cache 返回数据
rs1_xor_save	1	-	内部信号	保存的校验值 1
rs1_xor_calc	1	-	内部信号	计算的校验值 1
rs1_o	32	-	内部信号	寄存器数据 1
rs2_xor_save	1	-	内部信号	保存的校验值 2
rs2_xor_calc	1	-	内部信号	计算的校验值 2
rs2_o	32	-	内部信号	寄存器数据 2
hold	1	输出	外部信号	处理器挂起信号
rs1_data	32	输出	外部信号	寄存器读数据 1
rs2_data	32	输出	外部信号	寄存器读数据 2

图 3.6 中的 Cache 为数据 Cache，其内部由 SRAM 搭建而成，与寄存器堆相比，Cache 的容量更大，不易实现多端口的同时读取，因此每个周期只能完成一笔数据的读取和写入。本文中将寄存器堆的备份数据存放到数据 Cache 中，占用了 Cache 中的部分缓存行，在读写逻辑上没有进行较大的修改，每个时钟周期只能从 Cache 中读取一个寄存器的数据备份，寄存器堆支持同时读取两个寄存器，因此在最坏情况下，即读取的两个寄存器中都出现了错误，此时 hold 信号将持续两个时钟周期。一般情况下，检错和纠错操作是自动进行的，即当出错的寄存器不是要读取的寄存器时，寄存

器的数据输出和出错寄存器的错误纠正可以并行进行，不会相互影响。

2. 寄存器堆结构

寄存器堆的功能为保存写入的数据、生成并保存校验值、检查寄存器状态，其结构如图 3.7 所示，由控制器、地址译码器、寄存器阵列组成。其中，控制器内部包含预处理器、校验器 B、多路选择器 MUX，控制器控制对寄存器阵列的访问，并完成对寄存器中的错误的纠正操作，当纠正操作与读写操作发生访问冲突时，控制器会对当前进行的操作进行排序，以保证数据访问的正确性。地址译码器负责将寄存器的地址信号 rd_i/rs1_i/rs2_i 译码成寄存器的选择信号 rd_sel/rs1_sel/rs2_sel，本质上是一个 5-to-32 的译码器，实现将 5 比特的二进制编码译码成 32 比特的 one-hot 码，译码后的信号连接到每个寄存器的使能端。寄存器阵列中对写入的数据进行校验，产生校验值，并对数据和校验值进行存储，在读取时对寄存器状态进行检查，产生寄存器状态向量 check_value，寄存器状态向量 check_value 是一个 32 比特的向量，每一位对应一个寄存器的状态（即是否出现可被检测到的错误）。

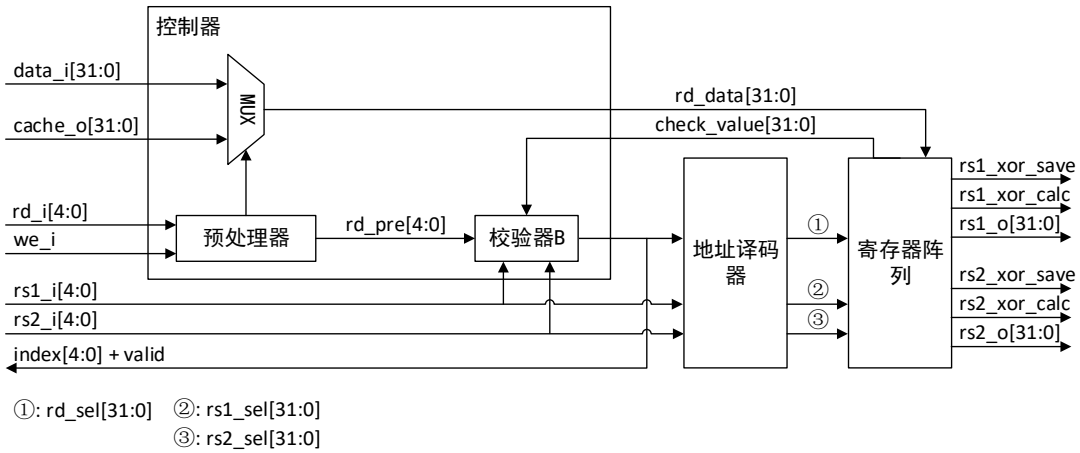

图3.7 寄存器堆结构

需要说明的是，we_i 和 valid 信号用于表示对寄存器的写操作是否有效，在浮点寄存器堆的设计时，这两个信号是必要的，但是在通用寄存器堆设计时，这两个信号可以不使用，这是因为在 RISC-V 架构中，通用寄存器 0 通过硬连线的方式连接到 GND，其数据恒为 0，写入通用寄存器 0 的任何数据都不会生效，在设计通用寄存器堆时，“rd_i=0”可以用来代替 we_i。

寄存器堆中的信号分为内部信号和外部信号，外部信号用于与寄存器堆容错结构中的其他部件进行交互，内部信号仅用于在寄存器堆内部进行信息传输，对外部不可见，信号的详细说明如表 3.5 所示（表中“-”表示内部信号，不区分方向）。

表3.5 寄存器堆信号列表

名称	位宽/bit	方向	信号类型	说明
rs1_i	5	输入	外部信号	寄存器读地址 1
rs2_i	5	输入	外部信号	寄存器读地址 2
rd_i	5	输入	外部信号	寄存器写地址
we_i	1	输入	外部信号	寄存器写使能
data_i	32	输入	外部信号	寄存器写数据
index	5	输出	外部信号	Cache 访问地址
valid	1	输出	外部信号	Cache 写有效
cache_o	32	输入	外部信号	Cache 返回数据
rs1_xor_save	1	输出	外部信号	保存的校验值 1
rs1_xor_calc	1	输出	外部信号	计算的校验值 1
rs1_o	32	输出	外部信号	寄存器数据 1
rs2_xor_save	1	输出	外部信号	保存的校验值 2
rs2_xor_calc	1	输出	外部信号	计算的校验值 2
rs2_o	32	输出	外部信号	寄存器数据 2
rd_pre	5	-	内部信号	预处理后的寄存器写地址
rd_data	32	-	内部信号	预处理后的寄存器写数据
check_value	32	-	内部信号	寄存器状态向量
rd_sel	32	-	内部信号	寄存器写选择信号
rs1_sel	32	-	内部信号	寄存器读选择信号 1
rs2_sel	32	-	内部信号	寄存器读选择信号 2

控制器设计是整个寄存器堆容错结构中的核心内容，对寄存器中出现的错误的纠正操作是偶发的，纠正操作也会对寄存器中的数据进行修改，因此当写入操作和纠正操作修改同一个寄存器时，此时将导致错误。控制器完成对纠正操作和写入操作的调度，以确保寄存器中的数据的正确性。另外，由于纠正操作会对寄存器中的数据进行修改，因此会出现与写入操作时相似的写后读冲突，这部分内容在寄存器堆的设计时已经被考虑到。下面将详细介绍控制器的设计细节。

如图 3.7 所示，控制器的数据输入来自 data_i 和 cache_o，其中 data_i 为处理器的写回数据，cache_o 为寄存器数据在 Cache 中的备份，data_i 和 cache_o 通过一个多

路选择器传递到 rd_data，多路选择的选择信号来自预处理器，可以认为每个时钟周期都会发生一次寄存器的写入操作，但数据的来源不同，预处理器完成对最终写入寄存器中的数据来源的控制。

预处理器对处理器发出的寄存器写使能信号 we_i 进行检查，判断处理器当前操作是否为处理器发起的写寄存器操作。预处理器的输出 rd_pre 来自于 rd_i，并对多路选择器进行控制，实现对最终写入寄存器中的数据来源的选择，多路选择器的输出结果如表 3.6 所示。

表3.6 多路选择器真值表

预处理器输入	多路选择器输出
we_i＝0	rd_data＝cache_o
we_i＝1	rd_data＝data_i

校验器 B 完成对寄存器中可被检测到的错误的检测和纠正操作，其中纠正操作可以看作是由寄存器堆内部发起的写寄存器操作（区别于与处理器发起的写寄存器操作，该操作为写入操作），并且当处理器对寄存器进行写入操作时，校验器 B 不工作，这样设置的原因是当写入操作发生时，意味着寄存器中的旧值将被刷新，旧值中的错误也会随着寄存器的刷新而被丢弃，因此写入操作具有最高的优先级，对同一个寄存器同时进行写入操作和纠正操作，此时的纠正操作无效。此外，校验器 B 对寄存器阵列输出的寄存器状态向量 check_value 进行检查，状态向量 check_value 存放每个寄存器当前时刻的状态（即是否出现可被检测到的错误），并且通过 rs1_i 和 rs2_i 进一步判断当前正在读取的寄存器中是否出现错误。当多个寄存器中都出错时，校验器会对出现错误的寄存器进行排序，按照寄存器序号增序的方式依此对出错的寄存器进行纠错，每个时钟周期仅完成对一个寄存器的纠错操作。

下面对不同情况进行分别说明：

1. 处理器写寄存器

   ——校验器 B 不工作，即此时不进行检错和纠错操作；

2. 处理器读寄存器

   a) rs1_i/rs2_i 对应的寄存器中两个寄存器都出错

      ——校验器 B 屏蔽除 rs1_i/rs2_i 外其他寄存器中的错误，并按照寄存器序号递增的顺序依次对 rs1_i/rs2_i 中出错的寄存器中的错误进行纠正，此时处理器会进入挂起状态，持续 2 个时钟周期；

   b) rs1_i/rs2_i 对应的寄存器中仅有一个寄存器出错

——校验器 B 屏蔽除 rs1_i/rs2_i 外其他寄存器中的错误，并对 rs1_i/rs2_i 中出错的寄存器中的错误进行纠正，此时处理器会进入挂起状态，持续 1 个时钟周期；

c) rs1_i/rs2_i 对应的寄存器中均未出错，其他寄存器中有多个寄存器出现错误

——校验器 B 按照寄存器序号递增的顺序依次对出错的寄存器中的错误进行纠正，并且此时寄存器数据的数据的输出和纠错操作是互不干扰的，即处理器不会进入挂起状态；

d) rs1_i/rs2_i 对应的寄存器中均未出错，其他寄存器中均未出错

——校验器 B 不工作，即此时不进行检错和纠错操作；

3. 处理器不操作寄存器

——与处理器读寄存器情况下的操作相同。

3. 寄存器阵列结构

寄存器阵列的结构如图 3.8 所示，由编码器、校验器 A、寄存器组成，编码器和校验器 A 的校验算法相同，当对校验算法进行修改时，需要保证二者使用的校验算法一致。编码器用于对写入的数据进行编码，生成校验值，校验值与写入数据一起存储到寄存器中。校验器 A 对寄存器进行检查，生成寄存器状态向量 check_value，状态向量 check_value 存放每个寄存器当前时刻的状态（即是否出现可被检测到的错误），处理器每次读寄存器时会对全部寄存器中的数据进行校验，与写入时生成并保存的校验值进行比较，比较的结果即为状态向量。

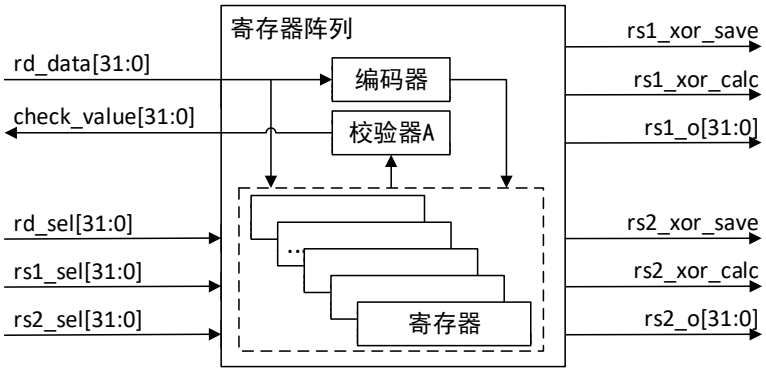

图3.8 寄存器阵列结构

如表 3.7 所示，寄存器阵列中的信号用于寄存器阵列与寄存器堆的其他部件进行数据交互。

表3.7 寄存器阵列信号列表

名称	位宽/bit	方向	信号类型	说明
rd_sel	32	输入	外部信号	寄存器写选择信号
rs1_sel	32	输入	外部信号	寄存器读选择信号 1
rs2_sel	32	输入	外部信号	寄存器读选择信号 2
rd_data	32	输入	外部信号	预处理后的寄存器写数据
check_value	32	输出	外部信号	寄存器状态向量
rs1_xor_save	1	输出	外部信号	保存的校验值 1
rs1_xor_calc	1	输出	外部信号	计算的校验值 1
rs1_o	32	输出	外部信号	寄存器数据 1
rs2_xor_save	1	输出	外部信号	保存的校验值 2
rs2_xor_calc	1	输出	外部信号	计算的校验值 2
rs2_o	32	输出	外部信号	寄存器数据 2

3.2.2 基于交叠编码技术的校验器

随着半导体工艺的提升，寄存器堆等存储器的近邻单元之间的相互影响逐渐明显，同时为了降低功耗而降低供电电压也导致近邻单元之间的耦合进一步加强，单次高能粒子事件引起近邻连续位翻转的概率逐步升高。现有的寄存器堆加固方法不具备处理连续多位的突发错误的能力，例如 Hamming 码方法，本文中使用交叠编码技术来处理近邻连续多位的错误。

交叠编码技术可以处理连续多位的突发错误，不能完全检测非连续的多位错误。本文中只考虑近邻多位连续突发错误，对于非连续的多位错误则不作考虑，主要原因如下：

1. 非近邻的多位翻转的发生概率很低，即由多粒子引起的位翻转故障发生的概率很低[27]；
2. 近邻连续多位翻转实际是由单个粒子引起的，其本质仍是单粒子翻转的故障；
3. 交叠编码技术对部分非连续位的翻转由一定的容错能力，通过调整交叠编码的间隔可以提高对非连续位的翻转的检测。

本文设计了一种基于交叠编码技术的校验器，用于替换 3.2.1 节设计的基于 Cache 结构的寄存器堆中的校验器 A。使用交叠编码技术的校验器加固的寄存器堆可以检测近邻连续位翻转的故障，并且与现有方法中使用的校验算法相比，交叠编码技术将数据划分为多个分组，每个组计算得到一个校验值，各组之间可以并行工作，数据路径

延时较小。

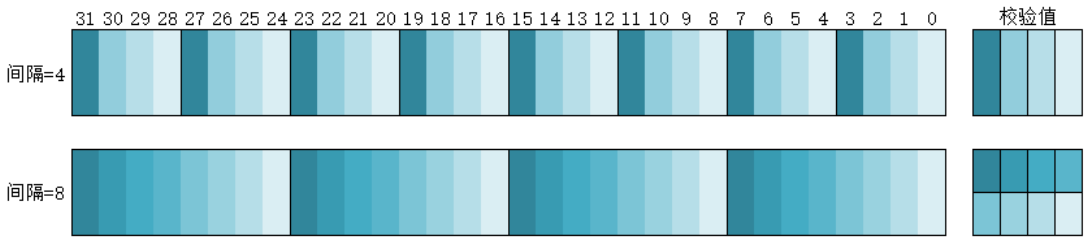

图3.9 基于交叠编码技术的校验器

图 3.9 展示了基于交叠编码技术的校验器，以单精度浮点数为例，有效数据宽度为 32 比特。按照不同的间隔对有效数据进行分组，每个校验分组内单独进行校验，得到该分组的校验值，交叠间隔为 4 时得到 4 个分组，交叠间隔为 8 时得到 8 个分组。从图 3.9 中可以看到，同一个校验组内的数据在物理空间上分布不连续，因此相邻连续多比特错误会被分割到不同的校验组中，这种分配策略提高了系统的检错能力。同时，为了最大程度的发挥基于交叠编码技术的校验器的检错能力，不同分组的检测结果最终通过或门进行输出，因此只要有一个分组检测出错误，则此寄存器中的数据即被判定为出错。

在各分组的校验算法的选择上，本文选择奇偶校验，因为奇偶校验可以用极低的成本实现对错误的检测。奇偶校验在硬件实现上较为方便，通过异或运算就可以实现奇偶校验，异或运算在物理上对应异或门，逻辑结构简单。交叠编码技术将数据划分为相互独立的分组，每个分组可以单独进行运算，通过这种并行执行的方式可以缩短路径延时。

本文选择交叠间隔为 4 和交叠间隔为 8 进行实验分析，后文将分别对这两种编码情况进行说明。

1. 8 位交叠编码奇偶校验

以单精度浮点数作为实验对象，首先对数据进行分组，分组的规则为对寄存器的位序号进行模 8 运算，运算结果相同的位为同一个分组，生成的校验位的位数需要 8 位。在 8 位交叠编码奇偶校验中，连续的 8 比特错误被分割到 8 个分组中，每个分组中均可以检测到错误。极端条件下，当连续 16 比特错误发生时，每个分组中被分配 2 比特错误，此时错误无法被检测到。

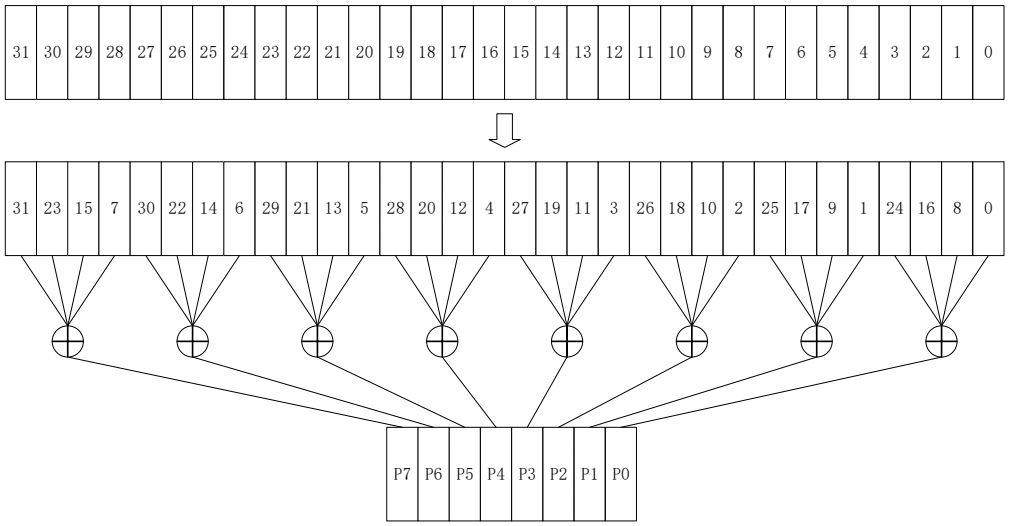

图3.10 校验器编解码过程（交叠间隔为 8）

生成校验的过程如图 3.10 所示，对寄存器的位序号进行编号，由于使用单精度浮点数进行实验，寄存器数据的位宽为 32 比特。

1. 计算 r＝index MOD 8，index∈[0,31]；
2. 将 r 相同的 index 划分到同一组，记为 $Group_r$，r∈[0,7]；
3. 对同一 Group 内的数据位进行按位异或，得到校验值 P_r，r∈[0,7]，校验值的计算可以并行进行，生成校验和检验时的逻辑电路的层数小，通过使用树形结构，逻辑电路的层数为 2 层。

2. 4 位交叠编码奇偶校验

对单精度浮点数进行操作，分组规则为对位序号进行模 4 运算，运算结果相同的位为同一个分组，生成的校验位的位数需要 4 位。

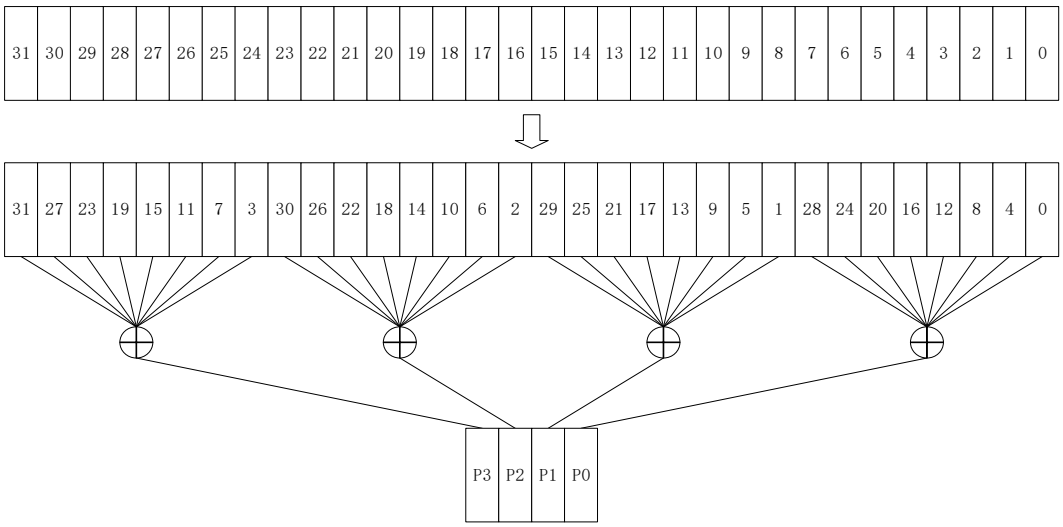

图3.11 校验器编解码过程（交叠间隔为 4）

生成校验的过程如图 3.11 所示，对寄存器的位序号进行编号，由于使用单精度浮点数进行实验，寄存器数据的位宽为 32 比特。

1. 计算 r＝index MOD 4，index∈[0,31]；
2. 将 r 相同的 index 划分到同一组，记为 $Group_r$，r∈[0,3]；
3. 对同一 Group 内的数据位进行按位异或，得到校验值 P_r，r∈[0,3]校验值的计算可以并行进行，生成校验和检验时的逻辑电路的层数小，通过使用树形结构，逻辑电路的层数为 3 层。

3.3 实验与验证

3.3.1 测试平台设计

本文基于 Cache 结构对寄存器堆进行容错设计，加固后的寄存器堆结构如图 3.12 所示，并且使用交叠编码技术对图 3.8 中的校验器 A 进行改进，增强了对寄存器堆近邻连续位翻转错误的容错能力。

为验证所设计的容错寄存器堆的可靠性，本文设计了如图 3.12 所示的测试平台，测试平台基于 Vivado 和 Synopsys VCS 进行搭建，由控制台、随机数生成器、加固后的模块、未加固的模块、终端/文件五部分构成。

1. 随机数生成器生成随机的测试向量，测试向量为值的范围为[0,31]的随机整数，用于模拟错误注入时错误的注入位置；
2. 控制台的执行过程划分为四个阶段：输入、驱动、监视、输出；
   a) 输入阶段从随机数生成器中获取测试向量；
   b) 驱动阶段首先发起寄存器写入操作，将未注入错误的寄存器数据写入到图 3.12 中的两个被测模块中（加固后的模块、未加固的模块），然后使用输入阶段获取到的测试向量来确定注入位置，进而进行错误注入，错误注入的模型有以下三种方式：
      i. 恒 1 注入，即将指定位固定为“1”；
      ii. 恒 0 注入，即将指定位固定为“0”；
      iii. 翻转注入，即将指定位的数据翻转，“0”翻转为“1”，“1”翻转为“0”。
   c) 监视阶段发起寄存器读取操作，从图 3.12 中的两个被测模块中读取数据，并与原始数据进行比较；
   d) 输出阶段将比较结果输出到显示终端或文件中，除比较结果外，还可以有选择地输出其他调试信息。
3. 终端/文件用于对测试结果进行显示和保存，通过使用模拟器软件提供的系统函数实现。

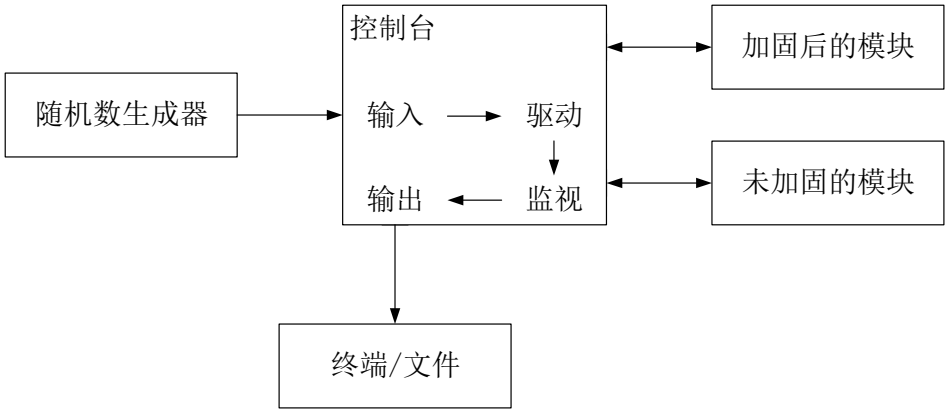

图3.12 浮点寄存器堆测试平台结构

3.3.2 实验结果

使用图 3.12 所示的测试平台对几种采用不同加固方案进行容错设计的寄存器堆进行错误注入实验，以验证本文中所设计的浮点寄存器堆容错结构的可靠性。并且在 Xilinx 的 FPGA 平台和 SMIC 的 180nm 工艺库下分别进行综合实验，从路径延时和资源开销方面进行对比分析。

本文中所设计的寄存器堆容错结构将寄存器的备份存放到处理器中的数据 Cache 中，这种加固策略占用了数据 Cache 中的部分 cacheline。理论上，这种容错设计会影响数据 Cache 的性能，但是处理器中的寄存器的数量有限（RISC-V 架构中具有 32 个通用寄存器和 32 个浮点寄存器，位宽分别为 32 比特和 64 比特），因此其对数据 Cache 的性能的影响也是有限的，对此本文进行了实验，以评估文中所提出的寄存器堆容错结构对数据 Cache 性能的影响。

1. 可靠性

表 3.8 展示了对几种采用不同加固方案进行容错设计的寄存器堆进行错误注入实验的实验结果，分别进行连续 1 比特、连续 4 比特、连续 8 比特错误注入，通过随机数生成器产生随机的注入位置，每次注入的错误个数恒定（例如在连续 4 比特错误注入的测试条件下，每次均有 4 个位出现错误）。

在本实验中对比测试的几种采用不同加固方案中，Hamming 码方法可以实现对错误的检测和纠正，但是部分其他方法只能进行错误的检测，如三模冗余，因此在本实验中只对不同方法的错误检测能力进行统计，即使部分方法可以实现对错误的纠正，但这个方面的能力不会被统计。

表 3.8 中“×”表示在给定错误注入条件下无法检测到错误，“√”表示可以检测到错误。采用三模冗余加固的寄存器堆可以检测到给定的所有错误类型，这是因为连续的错误只会被注入到三个之中的其中一个，因此通过表决器可以将错误屏蔽。采用

Hamming 码加固的浮点寄存器堆无法检测到连续 4 比特错误和连续 8 比特错误，这是因为使用的编码方式只能检测两位错、纠正一位错。Mirror Backup 方法[24]同样无法检测到连续 4 比特错误和连续 8 比特错误，因为其使用奇偶校验，只能检测到奇数个数的错误。C-DMR 使用了基于交叠编码技术的校验器，对于错误个数不等于交叠间隔的偶数倍的错误均能检测到。

从寄存器堆错误注入实验结果中可以看到，由于使用了交叠编码技术，基于 Cache 的寄存器堆结构对连续多比特错误有较好的容错能力，可以应对寄存器堆中出现的近邻连续多比特翻转错误。

表3.8 寄存器堆错误注入实验结果

	连续 1bit 错误	连续 4bit 错误	连续 8bit 错误
未加固	×	×	×
三模冗余	√	√	√
Hamming	√	×	×
Mirror Backup	√	×	×
C-DMR，交叠间隔为 4	√	√	×
C-DMR，交叠间隔为 8	√	√	√

2. 路径延时

对于路径延时的实验，本文首先使用 Xilinx 公司的 FPGA 对几种采用不同加固方案进行容错设计的寄存器堆进行综合，FPGA 型号为 XC7A35TFGG484-1，实验结果如表 3.9 所示，实验中统计了读操作的路径延时。从实验结果中可以看到，采用不同加固方案加固的寄存器堆的读操作的路径延时的差异很小，这是由于 FPGA 自身的结构特点所导致的，Xilinx 公司的 FPGA 中包含了许多可配置逻辑块（CLB，Configurable Logic Block），每个 CLB 中又包含了多个查找表（LUT，Look-up Table）和触发器（FF，Flip-flop），LUT 是查找表，内部存放逻辑运算的真值表，通过查表的方式完成逻辑运算，逻辑运算的输入对应于 LUT 的输入信号，逻辑门的输出对应于 LUT 的输出信号，LUT 的这一特点导致无论逻辑运算的复杂与否，其路径延时都是一样的。在 FPGA 实验中，路径延时的差异主要来自于线网延时，线网延时是由布线关系带来的，不同的方案所需要的逻辑资源不同，布线关系也就不同，逻辑面积越大，布线距离越长，延时越大。

本文还使用 SMIC 的 180nm 工艺库进行了综合，实验结果如表 3.9 所示。从表 3.9 中可以看到，进行三模冗余后的寄存器堆的延时基本没有增加，这是因为三模冗

余结构简单，关键路径仅增加了表决器对应的电路，这部分影响很小。使用 Hamming 码方法加固的寄存器堆相比于未加固的寄存器堆，增加了 1.01ns 的延时，延时的增加是由于增加的解码电路所导致的。Mirror Backup 方法[24]和 C-DMR 方法增加的延时最多，这是因为这两种方法使用了奇偶校验进行错误检测，编码和解码电路的电路产生了较多的路径延时，并且这两种方式都对寄存器堆进行备份操作，备份带来了面积的增加，从而导致路径长度增加。同样使用 C-DMR 结构，相比于交叠间隔为 4，交叠间隔为 8 在延时增加方面减少了 0.11ns，原因是交叠间隔为 8 中的编码器和解码器的逻辑门层数更少，路径更短。

从寄存器堆路径延时的实验中可以看到，基于 Cache 的寄存器堆结构由于加强了检错能力，并且可以对寄存器堆中的数据进行刷新，导致其在 SMIC 180nm 工艺库中综合后的路径延时几乎与 Mirror Backup 方法的路径延时相同，大于三模冗余方案和 Hamming 码方案。在 FPGA 中进行综合，几种加固方案所带来的路径延时的差别很小，这是 FPGA 的特点所导致的，因此对 FPGA 中实现的处理器软核中的寄存器堆进行加固时，基于 Cache 的寄存器堆结构可以显著提升容错能力，同时增加较少的路径延时。

表3.9 寄存器堆路径延时

	FPGA		SMIC 180nm
	逻辑器件延时/ns	线网延时/ns	延时/ns
未加固	4.083	4.311	1.31
三模冗余	4.111	4.585	1.31
Hamming	4.083	4.795	2.32
Mirror Backup	4.207	5.422	5.00
C-DMR，交叠间隔为 4	4.083	4.795	4.92
C-DMR，交叠间隔为 8	4.331	5.337	4.81

3. 资源开销

表 3.10 展示了几种采用不同加固方案进行容错设计的浮点寄存器堆资源开销的测试结果，分别在 Xilinx 的 FPGA 平台上和 SMIC 的 180nm 工艺库下进行综合。在 FPGA 资源占用上，三模冗余方法使用了 3 倍的 FF 资源，但 LUT 资源只有 2.93 倍，这是因为实验中只对每个浮点寄存器进行了三模冗余，对于地址译码部分没有进行冗余，并且 FPGA 中的 LUT 是一个可编程的逻辑单元，LUT 在表示逻辑运算时，简单的逻辑运算无法用完 LUT 的全部资源，因此在进行三模冗余时，综合软件的优化操

作使 LUT 得到更为充分的使用，因此最终 LUT 数量上并未增加 2 倍。Mirror Backup 方法[24]完整复制了地址译码电路，增加了 1.14 倍 LUT，额外增加的 LUT 是由于交换地址信号的电路产生的。Hamming 码方法中增加了编码器和解码器电路，增加了 1.24 倍的 LUT。相比于其他方法，C-DMR 方法增加的 LUT 最少，增加的这部分 LUT 用于错误的检测、错误的纠正、数据 Cache 的读取和写入，增加的 FF 用于存放寄存器写入时的校验值，不用于对寄存器数据的备份，备份的数据存放在数据 Cache 中，这是处理器中已有部件，因此没有对其进行统计。

从 SMIC 的 180nm 工艺库下综合的结果看，C-DMR 方法下增加的面积是除了 Hamming 码方法外最少的（在 SMIC 180nm 工艺库中进行综合实验时，C-DMR 方法中的 Cache 所占用的资源也被统计，因此实验结果包含了 Cache 的资源开销）。需要特别说明的是，Mirror Backup 方法增加了 1.49 倍的面积开销，几乎与三模冗余方法所增加的面积开销相同，这是因为 Mirror Backup 本身是针对基于 SRAM 结构的 FPGA 设计的，因此在 ASIC 电路中不具有优势。

综上所述，在资源开销方面，本文提出的基于 Cache 的寄存器堆结构所消耗的查找表和触发器数量均小于其他容错方案，特别地，当交叠间隔设置为 8 时，消耗的触发器数量要略微多于 Hamming 码方法。在 SMIC 180nm 工艺库中综合时，对 C-DMR 的实验结果中包含了 Cache 所占用的资源，但最终的面积开销仍小于三模冗余和 Mirror Backup 方法。

表3.10 寄存器堆资源开销

	FPGA				SMIC 180nm	
	查找表/LUTs		触发器/FFs		面积/um*um	
未加固	2864	1.00x	992	1.00x	115113.40	1.00x
三模冗余	8400	2.93x	2976	3.00x	297952.31	2.59x
Hamming	6702	2.34x	1178	1.19x	237172.32	2.06x
Mirror Backup	6407	2.24x	2048	2.06x	298607.61	2.59x
C-DMR，交叠间隔为 4	4537	1.58x	1120	1.13x	266737.37	2.32x
C-DMR，交叠间隔为 8	4683	1.64x	1248	1.26x	277518.23	2.41x

4. 性能

Cache 性能实验在 T-Head 的 E906 SmartL 平台上进行，其硬件结构如图 3.1 所示，数据 Cache 采用两路组相联设计，cacheline 的大小是可配置的，支持 2KiB、4KiB、8KiB、16KiB 和 32KiB，cacheline 大小为 32 字节。为了避免对 Cache 产生较大的影

响，C-DMR 占用了 12 组 cacheline，每组中仅占用 way1 来对寄存器数据进行备份。

表3.11 不同容量 Cache 中 cacheline 占用比例

Cache大小	cacheline数量	被占用cacheline数量	被占用比例
2KiB	64	12	37.5%
4KiB	128	12	18.75%
8KiB	256	12	9.38%
16KiB	512	12	4.69%
32KiB	1024	12	2.35%

表 3.11 展示了不同 Cache 容量下 C-DMR 占用的 cacheline 比例。随着 Cache 容量的增加，C-DMR 占用的 cacheline 比例逐渐降低，本文选择 2KiB 的 Cache 作为实验对象，已获得最坏情况下 C-DMR 结构对 Cache 性能的影响情况。

本文评估了使用 C-DMR 结构加固寄存器堆对数据 Cache 的性能的影响，使用 coremark 基准测试进行评估，在时钟配置上，处理器内核时钟、Cache 访问时钟、内存访问时钟之间的比值关系为 1:2:4，测试结果如图 3.13 所示。

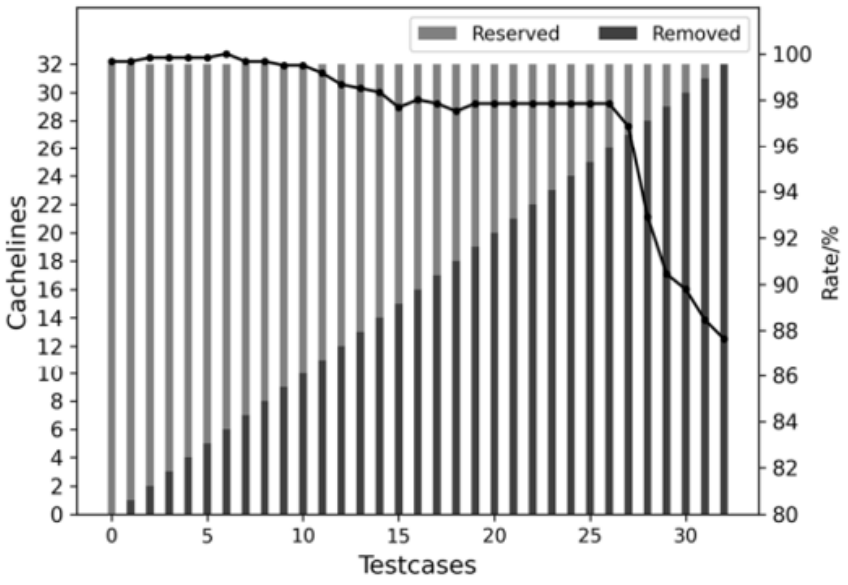

图3.13 Cache 性能测试（2KiB）

实验结果表明，占用数据 Cache 中的 12 个 cacheline 对 Cache 的性能的影响很小（小于 2%），这是因为在微控制器中，内核设备和存储设备之间的运行速率相差不大，Cache 的主要作用是减少内核对片上总线的占用。微控制器中的低速设备一般为

I/O 设备，I/O 设备的访问往往被设置成不可缓存，即不需要经过 Cache，以避免缓存一致性的影响。此外，微控制器的存储器空间分配可以通过链接脚本来完成，因此在必要时，可以通过软件和硬件协调将数据访问使用的 cacheline 与 C-DMR 占用的 cacheline 分隔开，从而减少由于 cacheline 被占用而造成的性能损失。当超过 25 个 cacheline 被占用时，Cache 性能将从 98%显著下降到 88%，造成这种现象的原因是，随着被占用的 cacheline 越来越多，Cache 中 cacheline 替换更容易被触发，频繁进行替换将导致系统性能下降。然而，由于本文的方法只需要占用最多 12 个 cacheline，从实验结果来看，所导致的性能下降非常小。

3.4 本章小结

本文使用标准的测试集对浮点寄存器堆的访问间隔进行统计，浮点寄存器中数据的保持时间可以达到 10000 个时钟周期，此期间可能会受到空间中高能粒子的影响而产生翻转错误，对此，本文提出了一种基于 Cache 结构的寄存器堆结构，即 C-DMR，其中校验器使用基于交叠编码技术进行设计，可以有效应对寄存器堆中的近邻连续多比特错误，具有较强的容错能力。在 FPGA 平台和 SMIC 180nm 工艺库中分别对几种采用不同加固方案进行容错设计的寄存器堆进行综合实验，从实验结果中可以看到，C-DMR 方法在路径延时和资源开销上均具有优势。

C-DMR 结构中使用数据 Cache 来对寄存器的备份数据进行保存，该结构占用了数据 Cache 中的部分 cacheline，因此 Cache 中的 cacheline 不能全部用于对内存数据访问的加速。本文进行了相应实验，以评估对 Cache 性能的影响，实验结果表明，在微控制器中使用 C-DMR 结构对寄存器堆进行加固时，所导致的性能下降非常小（小于 2%）。

第四章　浮点运算单元容错研究

浮点运算单元（FPOU，Floating-point Operation Unit）中对浮点数进行浮点运算操作，与单周期的整数运算不同，浮点运算一般需要经历多个时钟周期才能完成，并且由于浮点运算的复杂性，浮点单元所占用的面积更大，对其进行容错设计会产生较大的面积开销。

针对浮点运算单元的容错设计，本文采用乘积码和双模冗余结合的方式对其进行加固，使用乘积码对移位操作后的浮点运算的操作数进行编码，然后对编码后的操作数进行运算，运算后的结果经过解码得到最终的结果，解码的同时还会输出错误信号，指示运算过程中是否出现错误，对于乘积码方法无法应用的电路则采用双模冗余进行加固。在解码电路的设计方面，本文提出了一种基于查找表的快速除法电路，可以使解码操作在单个时钟周期内完成，同时具有较小的面积开销，并且针对不同的应用场景，提出了串联方式级联和并联方式级联两种结构，可以根据不同的需求进行选择，从而在路径延时和资源开销上达到均衡。

本章的最后介绍了对浮点加法器进行的容错设计方法和实验的结果，从可靠性、路径延时、资源开销等方面对实验结果进行了分析。

4.1 浮点运算单元可靠性分析

集成电路在极端恶劣的运行环境中运行或者长时间运行，可能会发生偏离指定或正确功能的行为，例如，或门出现永久故障，输出保持为“0”，此时如果输入“1”，那么输出就与预期不相符。集成电路出现故障的原因是多方面的，例如，外部干扰可能会导致逻辑门产生瞬时故障，在出现这种情况的几个时钟周期内，逻辑门的输出可能会偏离预期结果，这种干扰在空间环境中更为严重。此外，集成电路的制造缺陷、过热等因素也可能造成导线的断裂或与另一根导线短路。因此，确保数字系统在发生故障的情况下正确运行是十分重要的，浮点运算单元作为处理器中重要的运算单元，需要实现可靠计算。

通过冗余的方式可以实现对算术运算单元的保护，这种方式的优点是结构简单、易于实现，但是算术运算单元的重复增加了面积开销，并且，浮点运算单元由于其自身的复杂性，相比整数运算单元具有较大的电路面积，对浮点运算单元直接采用复制方式进行容错的代价较大。

使用算术检错码同样可以实现对运算过程中的出现的错误进行检测，该方式不会产生较大的面积开销[43]。算术检错编码允许直接对编码后的操作数进行算术操作，操

作数经过编码后会影响数据的位宽，需要对运算电路的位宽进行增加。乘积码是一种常用的算术检错编码，编码后的数据被直接送到算术运算单元中进行运算，与编码前的数据的运算过程一致，使用乘积码编码后数据的位宽的增加有限，因此可以使用比多重复制方式低得多的硬件开销来保护算术运算单元免受电路故障的影响。但是，由于浮点运算的特点，目前在浮点运算单元进行基于乘积码的容错设计仍面临许多问题，影响了乘积码容错方法在浮点运算单元中的应用：

1. 乘积码的编码和解码电路中涉及到了乘法和除法运算，乘法器和除法器会产生较多的面积开销，增加路径延时；
2. 浮点运算过程中的尾数移位、舍入会影响乘积码解码器对错误的检测，导致乘积码解码器误检；
3. 浮点运算的结果需要进行规格化，操作数经过编码会导致已有的尾数前导“1”预测器的输出不准确。

对于问题 2，移位操作会对编码的数据进行部分丢弃，这将导致数据无法被正确解码，目前尚无有效的解决方案，但是可以将操作数的乘积码编码操作移动到尾数移位操作之后进行，并对编码电路之前的电路则采用双模冗余进行加固。

对于问题 1 和问题 3，本文将进行分析并提出相应的解决方案，从而使得乘积码可以在浮点运算单元中进行应用。

4.2 浮点运算单元容错设计

对于浮点运算单元，本文使用乘积码对其进行容错设计，并且对乘积码方法无法应用到的电路采用双模冗余方法进行加固。

本节首先介绍乘积码容错的原理，使用校验模数为 3 的乘积码对操作数进行编码，然后介绍本文所提出的乘积码解码器结构（即快速除三电路），最后以浮点加法器为例对其进行加固设计。

4.2.1 乘积码容错原理

乘积码[43]也被称为 AN 编码或者 AN 码，将一个数 N 表示为乘积 AN，其中 A 为校验模数，是一个常量。要确定一个 AN 编码的操作数是否有效，需要检查这个操作数关于 A 的整除性，即是否可以被 A 整除。现代计算机中的操作数以二进制形式进行存储和表示，因此当出现单比特错误时，操作数中引入的误差为 2^n，其中 n∈非负整数，即出现错误的位置要么都是“0”到“1”的翻转、要么都是“1”到“0”的翻转，这种错误也被称为单向错误，是数字电路实现中的一类重要错误。校验模数为奇数时即可实现对这种单比特错误的检测。乘积码的编码和解码涉及到乘法和除法运算，

为了使这些编码简单实用，应确保校验模数的乘法和除法操作简单。

一类低成本乘积码的校验模数的形式为：$A=2^a-1$，其中 a∈正整数，这类乘积码的乘法操作可以通过移位和减法实现，具体操作如公式 4-1 所示。

$$AN = (2^a - 1) \times N = 2^a \times N - N \tag{4-1}$$

乘积码是一种不可分编码，在这种编码方式中，原始数据与用于检查的冗余数据被混杂在一起，因此无法从 AN 码上直接得到原始数值 N，获取原始数值 N 需要进行解码，即除以校验模数。本文所使用的低成本乘积码的校验模数中 a=2，即 A=3，可以检测所有的单比特错误。编码操作过程较为简单，操作数 N 在进行运算前，需要先乘三，在实际电路中，可以简化为将 N 左移两位，然后减去 N，从而将乘法操作转化为减法操作，极大地缩减了编码电路的延时。对于解码器的设计，需要将运算得到的数进行除三运算，除三运算则复杂得多，现代处理器中的除法器是基于迭代减法或者迭代乘法设计的，除法运算过程中需要使用减法器和乘法器，并且完成一次除法运算需要多个周期，因此需要对除三电路进行优化，使其可以在一个周期内完成除三运算，并且除三运算能够同时得到商和余数，余数不为 0 时表示运算过程中出错，商为最终的运算结果。本文在 4.2.2 节中将介绍除三运算的优化电路，提出一种基于查找表的快速除三电路，以满足上述要求。

4.2.2 快速除三电路设计

本文所提出的一种基于查找表的快速除三电路，可以实现在单个时钟周期内完成操作数的除三运算，如图 4.1 所示，输入为位宽为 N 比特的被除数，输出为位宽为 N 比特的商和位宽为 2 比特的余数，电路内部由若干基本单元（基本除三电路单元）进行组合，本节将对其结构设计进行介绍。

图4.1 快速除三电路结构

1. 基本除三电路

基本除三电路是本文提出的基于查找表的快速除三电路中的基本单元，使用查找表实现 4 比特数的除三运算，运算得到 2 比特的商（通过对被除数进行限制，运算得到的商的最高 2 位恒为 0，因此可以将最高 2 位截断）和 2 比特的余数。

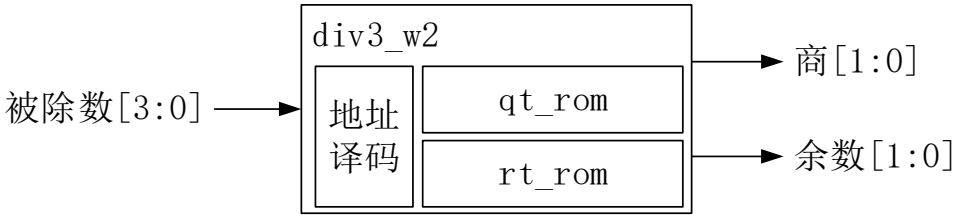

图4.2 基本除三电路 div3_w2 模块结构

如图 4.2 所示，基本除三电路 div3_w2 模块完成 4 比特数的除三运算操作，由地址译码器、qt_rom、rt_rom 组成，其中地址译码器的功能为对 4 比特的被除数进行译码，译码后的信号用作 qt_rom 和 rt_rom 的地址信号，qt_rom 和 rt_rom 分别存放 2 比特的商和 2 比特的余数，qt_rom 和 rt_rom 存放的表项如表 4.1 所示，均为二进制形式，通过查找表可以完成复杂的逻辑运算，对 qt_rom 和 rt_rom 进行寻址得到除三运算的商和余数。

表4.1 基本除三电路 div3_w2 模块真值表

地址	qt_rom 表项	rt_rom 表项
0000	00	00
0001	00	01
0010	00	10
0011	01	00
0100	01	01
0101	01	10
0110	10	00
0111	10	01
1000	10	10
1001	11	00
1010	11	01
1011	11	10
11xx	×	×

在 qt_rom 和 rt_rom 的表项中，对 11xx（即 1100、1101、1110、1111）不进行寻址，这是因为，当被除数为 11xx 时，此时得到的商为 1xx，位宽为 3 比特，这会造成商溢出。此外，快速除三电路由若干基本除三电路组合而成，组合的方式为前一级电路输出的余数作为后一级电路输入的被除数的最高两位，而除三运算的余数只有三个

有效值：00、01、10，因此除第一级的基本除三电路外，其他的基本除三电路输入的被除数的最高两位不会为 11，因此这种设计是安全的。

2. 8 比特数除三电路

通过对若干基本除三电路进行组合，可以得到多比特数除三运算的商和余数。如图 4.3 所示，8 比特数的除三运算模块 div3_w8 使用 4 个 div3_w2 模块进行串联，输入的被除数为 10 比特，与基本除三电路的要求相似，输入的被除数中的最高两位的有效取值为 00、01、10，这两位用于接收上一级除三运算所产生的余数。div3_w8 模块输出 8 比特的商（通过对被除数进行限制，运算得到的商的最高 2 位恒为 0，因此可以将最高 2 位截断）和 2 比特的余数，div3_w8 模块可以通过级联方式构造多比特数的除三运算电路。

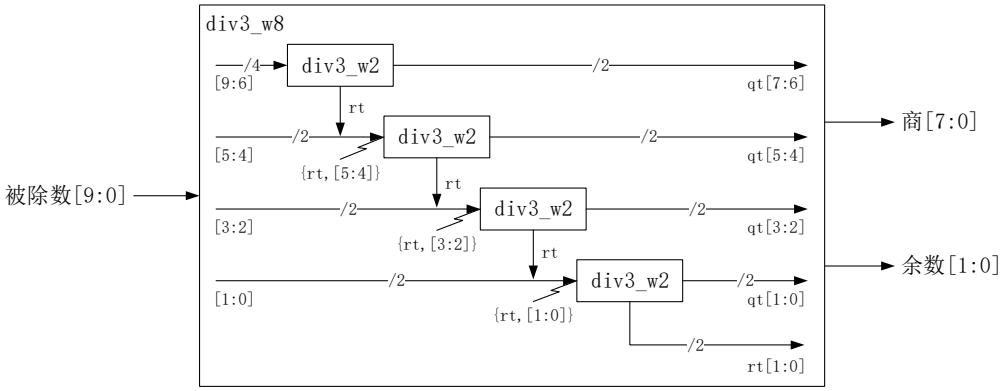

图4.3 8 比特数除三电路 div3_w8 模块结构

在图 4.3 中，div3_w8 模块中的每个 div3_w2 模块得到 2 比特的商和 2 比特的余数，高位被除数通过 div3_w2 模块得到余数，余数与低位被除数结合，构成新的 4 比特数传入下一级 div3_w2 模块中。在连接方式的设计上，4 个 div3_w2 模块通过串联级联的方式进行连接，因此后级 div3_w2 模块需要等待前级 div3_w2 模块输出余数后才能继续运算。

3. 多比特数除三电路

多比特数（位宽大于 8 比特）除三电路可以直接通过将基本除三电路进行串联得到，在串联连接之后，每一级电路只能实现对被除数中的 2 个比特位进行运算，灵活性较差。本文选择先使用基本除三电路构造 8 比特数除三电路，然后使用基本除三电路和 8 比特数除三电路来组合得到多比特数除三电路，这种模块化设计可以极大地方便后期对电路进行优化。

多比特数除三电路的构造方法为，首先将被除数的高位填充“0”，使得填充后的

被除数的位宽为 8 的整数倍。被除数的高位填充“0”后会导致运算的位宽增加，这部分除三后商和余数均为“0”，因此对最终的运算结果没有影响，但是会导致无效的运算，增加电路面积消耗和路径延时，因此可以根据实际情况选择 div3_w2 模块和 div3_w8 模块来组合，尽量避免数据填充。

下面以单精度浮点数中使用的多比特数除三电路为例，对其结构设计进行介绍，数据位宽为 26 比特数（单精度浮点数的尾数部分位宽为 23 比特，扩展整数位后的扩展尾数为 24 比特，26 比特为原始扩展尾数乘 3 后的数值的位宽），根据构造方式的不同，分为以串联方式级联的 26 比特数除三电路，和以并联方式级联的 26 比特数除三电路。

图 4.4 展示了以串联方式级联的 26 比特数除三电路，由 3 个 div3_w8 模块和 1 个 div3_w2 模块构成，从 26 比特被除数的高位开始，每 8 比特传入一个 div3_w8 模块中，其中第一级的 div3_w8 模块的被除数的最高两位的值为 00，之后的 div3_w8 模块的被除数的最高两位的值来自于前一级 div3_w8 模块的 qt 输出，即前一级电路输出的余数。26 比特被除数的最低两位传入一个 div3_w2 模块中，得到最终的 2 比特余数。3 个 div3_w8 模块和 1 个 div3_w2 模块输出的商 qt 通过拼接得到 26 比特的商。由于 div3_w8 模块内部由 4 个 div3_w2 模块串联构成，因此以串联方式级联 26 比特数除三电路实际上是由 13 个 div3_w2 模块串联构成，这种串联方式构造的多比特数除三电路的面积小，但由于后级 div3_w2 模块需要等待前一级 div3_w2 模块的运算结果，因此具有较大的延时，当被除数的位宽较大时，以串联方式级联的多比特数除三电路可能造成时序违例的问题。

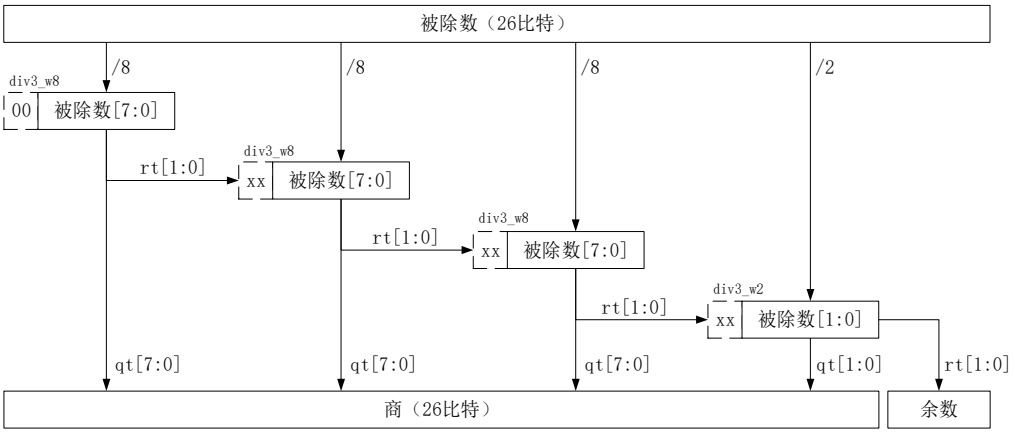

图4.4 多比特数除三电路 div3 模块结构（串联方式级联）

图 4.5 展示的是以并联方式级联的 26 比特数除三电路，由 7 个 div3_w8 模块、1 个 div3_w2 模块和 2 个选择器构成。与以串联方式级联的 26 比特数除三电路不同，以并联方式级联的 26 比特数除三电路中后级模块无需等待前一级模块的运算结果，

例如，被除数[17:10]不需要等待被除数[25:18]的运算结果，该部分电路会分别计算前一级余数为 00、01、10 情况下的结果，等前一级运算完成后再通过选择器进行选择，因此被除数[25:18]、被除数[17:10]、被除数[9:2]同时进行运算，极大的缩减了路径延时，与串联方式级联的 26 比特数除三电路相比，并联级联方式的电路额外使用了 4 个 div3_w8 和 2 个选择器，因此会消耗了更多的面积。使用并联方式级联的 26 比特数除三电路可以实现除法运算的加速，本质是通过将 26 比特除法运算划分为不同的可同时进行的短位宽数的除法，每个短位宽数的除法操作会遍历所有可能的输入情况，本质是一种选择策略。从图 4.5 中可以看到，被除数的高 24 比特的除法可以转化为 3 个 8 比特数的运算操作，可以并行进行，最后只需要依次使用前一级的余数顺序经过一次选择就可以得到 26 比特的商和 2 比特的余数。

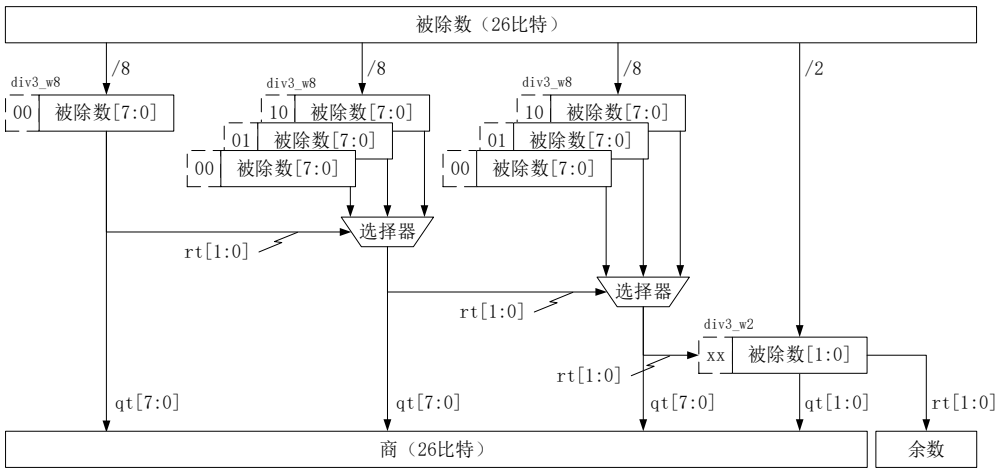

图4.5 多比特数除三电路 div3 模块结构（并联方式级联）

4.2.3 浮点加法器容错设计

计算机中的加法电路可以完成加法运算和减法运算，乘法运算的结果可以通过多次加法运算得到，除法运算则可以通过多次减法运算或乘法运算完成，因此当不考虑执行效率时，可以使用加法电路进行四则运算。

本文以浮点加法器为例进行容错设计，加法器可以完成与浮点数加法和减法相关的指令操作。由于浮点数运算过程中的移位操作会导致被编码的数据无法被正确解码，因此只使用乘积码无法实现对浮点加法器的容错设计，对此，本文使用乘积码和多模冗余结合的方式对浮点加法器进行加固。

1. 浮点加法器

如图 4.6 所示，在计算机五级流水线中，浮点加法器位于 EX 阶段（即执行阶段），由于浮点运算较为复杂，为了满足时序要求，执行阶段需要拆分为多个子阶段，在浮

点加法运算中，执行阶段包含 EX1、EX2、EX3 三个子阶段。

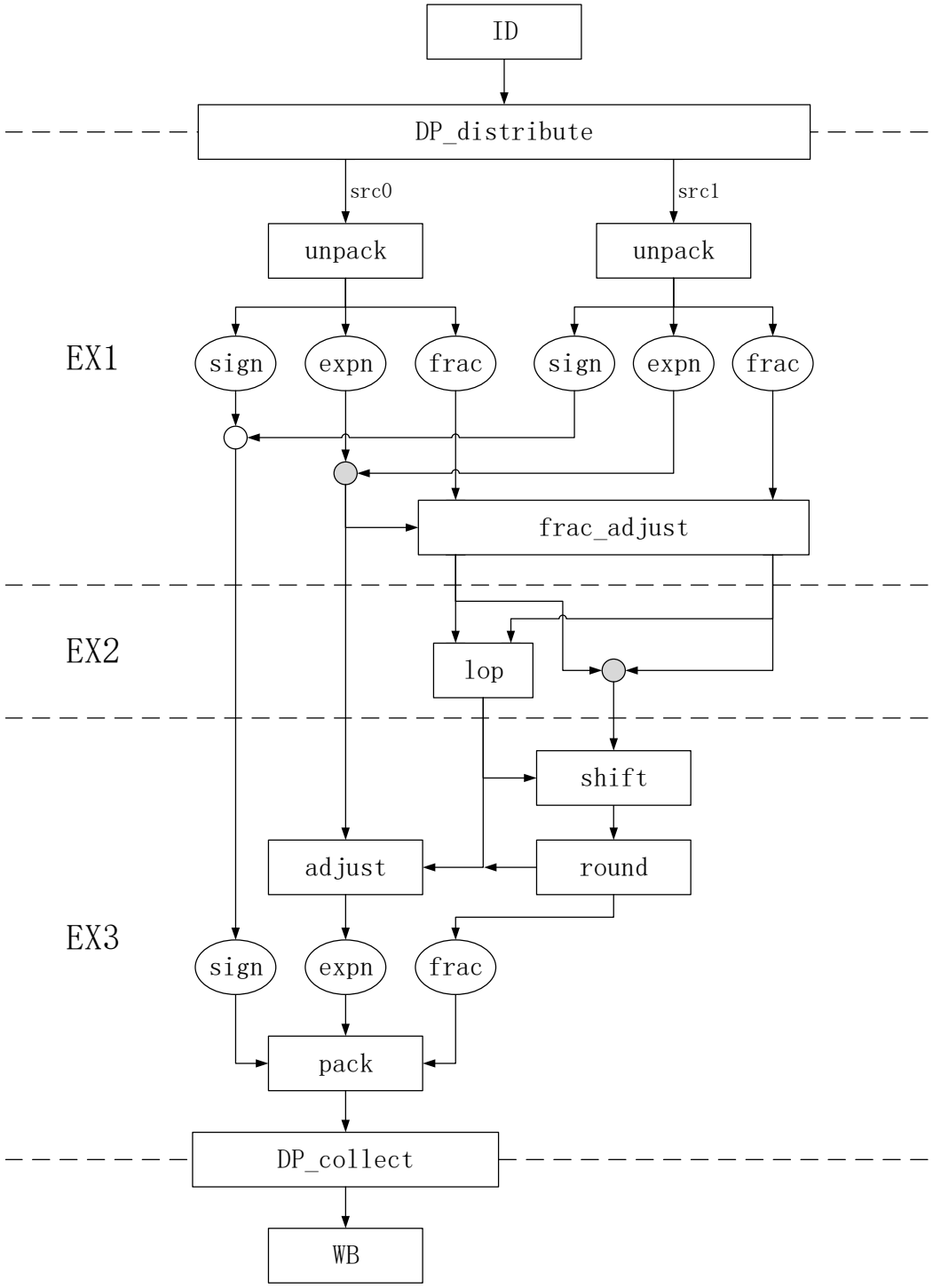

图4.6 浮点加法器流水线结构

ID 阶段（即指令译码阶段）从浮点寄存器堆读取操作数，经过 DP_distribute 模块进行数据分发。

EX1 阶段对操作数进行解包操作，根据 IEEE 754 浮点数标准提取出符号位（sign）、

阶码（expn）、尾数（frac）。符号位单独计算，得到运算结果的符号。阶码进行差运算，结果输入 frac_adjust 模块中用于对尾数的移位操作，对于尾数的运算，则在 EX2 阶段进行。

EX2 阶段进行尾数的运算，对于加法运算，两个尾数相加后最多产生一位进位，因此加法运算后尾数规格化调整较为简单。但是对于减法运算，当两个尾数大小相近时，减法运算将得到一个非规格化数，并且这个数规格化所需要移动的位数是不固定的，如果在减法运算结束后对移动的位数进行判断，会增加运算的时间，通过使用 lop 算法[44]，可以在减法运算的同时对运算结果中的前导“1”的位置进行预测，得到尾数规格化所需要移动的位数。

EX3 阶段对运算后的尾数进行规格化，通过 shift 模块进行，调整后的尾数在 round 模块中进行舍入操作。尾数规格化和舍入过程中会对阶码进行调整，因此 EX2 阶段 lop 模块的输出和 EX3 阶段 round 模块的输出会输入到 adjust 模块中，以完成对阶码的调整。最后，对运算后的符号位、阶码、尾数进行打包，得到符合 IEEE 754 浮点数标准的运算结果，通过 DP_collect 模块进行收集，在 WB 阶段写回到浮点寄存器堆中，完成浮点数运算操作。

2. 浮点加法器容错

浮点加法器容错设计框图如图 4.7 所示，使用乘积码和多模冗余结合的方式对浮点加法器进行加固，不同的子模块采用了不同的加固方法，相邻的执行阶段间数据通过流水线寄存器进行传递，由于数据的保持时间较短，写入的数据在下一个周期会被读走，因此本文没有它们进行加固。输入到 EX1 阶段的 Register 的数据来自浮点寄存器堆，浮点寄存器堆使用本文提出的 C-DMR 结构进行设计，可以检测并纠正由单粒子翻转现象产生的错误，并且对近邻连续多比特翻转错误也有容错能力，因此可以认为输入到浮点加法器中的操作数是可靠的。

在 EX1 阶段，经过流水线寄存器暂存的两个操作数 src0 和 src1 经过两个独立的 unpack 模块进行数据解包，其中，解包后的尾数 frac 的表示形式为“1 位整数位+N 位小数位”，整数位可取值“0”或“1”。符号位的运算电路和阶码的运算电路采用双模冗余的方法进行加固，运算结果保存到流水线寄存器中向下一阶段传递。相比于符号位运算电路和阶码运算电路，尾数的运算电路具有更大的电路面积，直接进行双模冗余会带来巨大的面积开销，因此本文采用乘积码对尾数运算电路进行容错设计，对于尾数运算电路中乘积码无法应用的尾数移位部分，使用双模冗余进行加固。

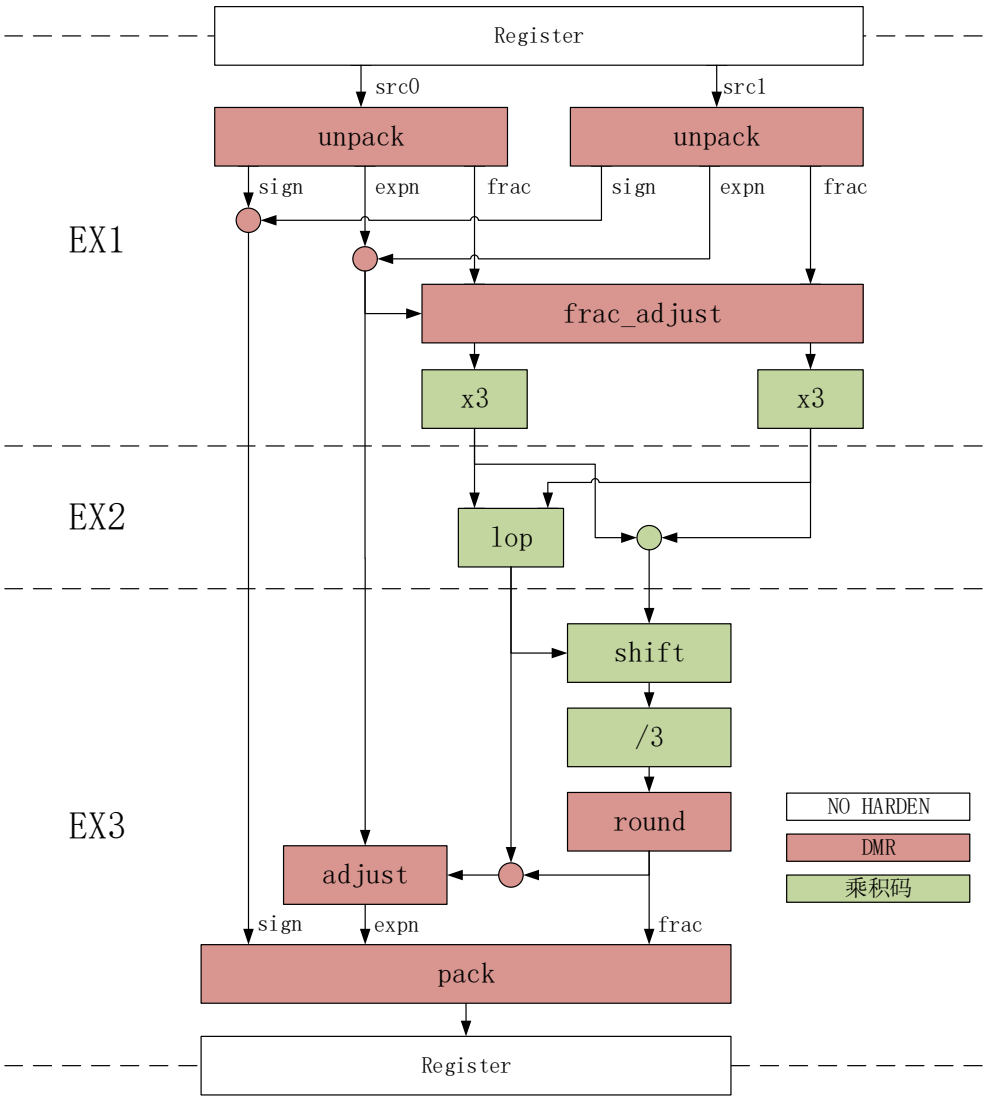

图4.7 浮点加法器容错结构

如图 4.7 所示，尾数的运算包含多个阶段：尾数移位、运算、规格化、舍入。在 EX1 阶段中，frac_adjust 模块中进行尾数的移位操作，根据阶码比较的结果，对数值较小的操作数的尾数进行右移，这部分电路使用双模冗余进行加固。frac_adjust 模块输出两个移位后的尾数，经过编码模块（“x3”模块）对其进行乘积码的编码，编码操作为分别乘以校验模数（校验模数为 3），其中“frac×3”可以表示成“frac×2+frac”的形式，“frac×2”可以通过将 frac 左移 1 位得到，因此“frac×3”只需要一次加法运算即可完成。编码后的两个尾数在 EX2 阶段进行加/减运算，在 EX3 阶段进行尾数运算后的规格化和舍入。

尾数规格化操作由 shift 模块完成，但是根据图 4.7 所示，该模块完成的是尾数运算“3A±3B”的规格化，直接使用 shift 模块的输出去调整阶码值是不正确的，阶码值需要根据尾数运算“A±B”的规格化的结果进行，需要进行修正，具体的操作方法

如表 4.2 所示。由于 lop 模块的预测结果最多只有一位误差[44]，因此 shift 模块输出的尾数结果的可能情况只有 11x.xx…x、10x.xx…x、011.xx…x、010.xx…x，只需要对这四种情况进行修正即可。

表4.2 尾数规格化的修正值

shift 模块输出	解码后输出	阶码修正值
11x.xx…x	01x.xx…x	+1
10x.xx…x	001.xx…x	+0
011.xx…x	001.xx…x	+0
010.xx…x	000.1x…x	-1

EX3 阶段中的解码模块（“/3”模块）通过使用本文改进的快速除三电路，可以实现在一个时钟周期内得到除三运算后的商和余数，其中商为解码后的运算结果，余数用来对运算中的错误进行检测。尾数的舍入操作中需要对尾数进行加“1”操作，可能产生溢出，因此对阶码的调整操作（adjust 模块）放置在舍入操作（round 模块）之后，统一进行一次阶码调整，阶码的调整值有三个来源，分别是：尾数规格化修正值、shift 模块的输出、舍入模块的输出。EX3 阶段最后将符号位、阶码、尾数进行打包，在 WB 阶段写回到浮点寄存器堆。

需要指出的是，EX1 阶段的编码器模块和 EX3 阶段的解码器模块没有进行额外的加固，因此当这两个模块出现故障时，错误将无法被检测到。另外，解码模块仅对 EX2 阶段的尾数的运算结果进行检查，对于 EX2 阶段的 lop 模块则没有配备专门的检查模块，原因是 lop 模块对尾数运算后的结果中的前导“1”的位置的预测，预测仅在两个浮点数相减时才需要，并且当且仅当前导“1”对应的位置的数值从“1”变成“0”才会导致最终的故障，其他情况均可以在尾数规格化时被检测到，故障模型可以通过公式 4-2 进行描述。假设数据结果的位宽为 N 位，并且减法运算后的数据结果中最高位“1”出现在第 i 位中的概率是相等的。

$$p_{失效} = p_{SET} \cdot p_{SUB} \cdot \frac{1}{N} \cdot p_{1\to0} \tag{4-2}$$

其中，

p_{SET}，单粒子瞬态故障发生的概率；

p_{SUB}，减法运算发生的概率；

$p_{1\to0}$，单个比特位状态从“1”变为“0”发生的概率。

4.3 实验与验证

4.3.1 测试平台设计

为验证所加固后的浮点加法器的可靠性，本文设计了如图 4.8 所示的测试平台，测试平台基于 Synopsys VCS 进行搭建，由控制台、激励生成器、激励文件、加固后的模块、未加固的模块、终端/文件六部分构成。

1. 激励生成器生成运算所需的操作数，产生的操作数数值随机且可以覆盖全部的规格化浮点数，操作数保存到激励文件中，激励文件在运行期间被控制台读取；
2. 控制台的执行过程划分为四个阶段：输入、驱动、监视、输出；
   a) 输入阶段从激励文件中获取操作数；
   b) 驱动阶段产生时钟序列，将操作数分别输入到两个模块中，当模块执行到 EX2 阶段时，将尾数运算中随机的一个尾数上的随机的一个比特位固定到“1”，以模拟运算过程中的出错；
   c) 监视阶段对两个模块的运算结果进行监视，对于加固后的模块，还会监视双模冗余比较的结果和解码器的输出；
   d) 输出阶段将比较结果输出到显示终端或文件中，除比较结果外，还可以有选择地输出其他调试信息。
3. 终端/文件用于对测试结果进行显示和保存，通过使用模拟器软件提供的系统函数实现。

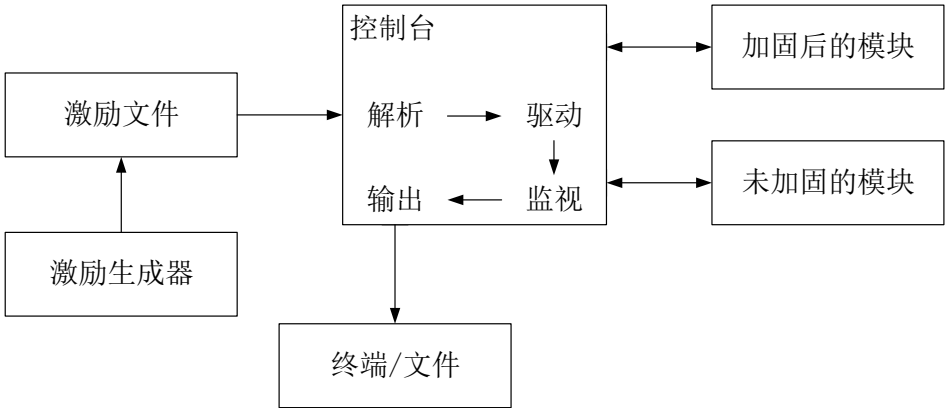

图4.8 浮点运算单元测试平台结构

4.3.2 实验结果

1. 路径延时和面积开销

在 SMIC 180nm 工艺下分别对本文设计的快速除三电路和加固后的浮点加法器电路进行综合，综合后的结果如表 4.3 和表 4.4 所示。

表 4.3 为除三电路在 SMIC 180nm 工艺下的综合结果，从结果中可以看到，未经优化的 26 比特数除三电路的在 SMIC 180nm 工艺下所需的硅面积为 5668.19 um*um，产生的路径延时为 9.76ns，这样的路径延时导致在该工艺下设计的处理器无法稳定运行在 100MHz 的频率下。相比之下，使用本文中提出的串联方式级联得到的 26 比特数除三电路的面积为 3406.23 um*um，路径延时为 6.96ns。使用并联方式级联得到的 26 比特数除三电路由于使用更多的子模块，面积消耗较大，面积为 8222.86 um*um，路径延时为 3.47ns，在 SMIC 180nm 工艺下可以将主频提高到 300MHz。

从表 4.3 中的数据可以看到，使用串联方式级联的 26 比特数除三电路，无论在面积消耗还是路径延时上，均优于未经优化的默认 26 比特数除三电路，并联方式级联的除三电路使用“面积换时间”的策略，路径延时最小，但牺牲了面积。

表4.3 除三电路的面积开销和路径延时

	面积(um*um)	延时(ns)
未经优化（26bit）	5668.19	9.76
串联级联（26bit）	3406.23	6.96
并联级联（26bit）	8222.86	3.47

表 4.4 中展示了对加固前后对浮点加法器电路综合的结果，从综合后的结果中可以看到，加固后的面积增加了 51.98%，路径延时增加了 60.99%，路径延时增加是因为编码操作和解码操作没有在单独的执行阶段中进行，导致数据路径变长，可以通过切分流水线的方式进行优化。相比于对浮点加法器电路直接采用双模冗余进行加固（面积增加大于 100%），本文采用的加固方法的面积开销更少。

表4.4 浮点加法器的面积开销和路径延时

	面积(um*um)		延时(ns)	
加固前	103820.27	-	12.33	-
加固后	157784.46	+51.98%	19.85	+60.99%

2. 可靠性

使用 4.3.1 节中设计的测试平台对加固后的浮点加法器的可靠性进行测试，测试结果如图 4.8 和图 4.9 所示，分别展示了未经加固的浮点加法器和加固后的浮点加法器对相同的操作数的运算结果，为了让浮点加法器中的 lop 模块可以工作，激励生成器生成的测试用例均为浮点数减法运算，图 4.8 和图 4.9 中的每一行为一次浮点运算的测试结果，其中，圆括号前的数据为浮点运算的运算结果，圆括号内的第一个数是测试平台的运算结果，浮点数小数点后均保留 14 位。圆括号内方括号中的数值为“0”表示浮点加法器自身未检测到错误或无错误，不为“0”表示出现错误且错误被检测到。圆括号内的最后一个数为模块的运算结果与测试平台的运算结果的差值。

两个浮点加法器的运算结果均与测试平台的运算结果进行了比较，从实验结果中可以看到，无论是加固后的浮点加法器的运算结果还是未经加固后的浮点加法器的运算结果，部分测试用例下的运算结果与测试平台的运算结果有微小的差别，造成这个现象的原因是测试平台中进行浮点运算是按照双精度浮点数进行的（从激励文件中读取操作数，转化为双精度浮点数后进行计算），而被测的浮点加法器中进行的是单精度浮点数（从激励文件中读取，然后直接输入到被测的浮点加法器中），并且在显示时保留了 14 位小数，这个精度远超过单精度浮点数所能表示的精度，因此测试结果中出现的微小误差是可以接受的。

```
 71.09006599999999 -  -153.88677100000001 =   224.97683699999999(  224.97683699999999 error [0]     0.00000000000000)
 74.89104300000000 -   632.36321599999997 =  -557.47217300000000( -557.47217300000000 error [0]     0.00000000000000)
-64.25487200000001 -   802.07541200000003 =  -802.33028400000001( -866.33028400000001 error [0]   -64.00000000000000)
 52.46502600000000 -    15.29237700000000 =    33.17264900000001(   37.17264900000000 error [0]     3.99999999999999)
-37.81783400000000 -  -235.16367800000000 =   197.34584400000003(  197.34584400000000 error [0]    -0.00000000000003)
-32.43875400000000 -  -313.77266800000001 =   281.33391400000005(  281.33391399999999 error [0]    -0.00000000000006)
 -7.36250900000000 -  -664.19859799999995 =   656.83608900000002(  656.83608899999990 error [0]    -0.00000000000011)
-24.70051500000000 -  -872.76288199999999 =   784.06236699999999(  848.06236699999999 error [0]    64.00000000000000)
-40.75221800000000 -   605.76113899999996 =  -646.51335699999993( -646.51335699999993 error [0]     0.00000000000000)
 70.21422500000000 -   211.17549099999999 =  -140.96126599999999( -140.96126599999999 error [0]     0.00000000000000)
-43.19432000000000 -  -213.44413399999999 =   170.24981400000001(  170.24981399999999 error [0]    -0.00000000000003)
 45.69002200000000 -  -665.43154500000003 =   647.12156700000003(  711.12156700000003 error [0]    64.00000000000000)
-12.17389600000000 -   943.51132800000005 =  -955.68522399999995( -955.68522400000006 error [0]    -0.00000000000011)
-95.09371899999999 -   936.59134100000006 = -1031.68506000000002(-1031.68506000000002 error [0]     0.00000000000000)
 59.85816000000000 -    13.54330200000000 =    42.31485799999999(   46.31485800000000 error [0]     4.00000000000001)
-39.98116100000000 -   -76.92790599999999 =    36.94674499999999(   36.94674499999999 error [0]     0.00000000000000)
 50.72457000000000 -  -546.07448799999997 =   532.79905799999995(  596.79905799999995 error [0]    64.00000000000000)
-80.91299400000000 -   238.96105000000000 =  -303.87404399999997( -319.87404400000003 error [0]   -16.00000000000006)
-85.25353400000000 -  -216.84074000000001 =   131.58720600000001(  131.58720600000001 error [0]     0.00000000000000)
 69.10592300000000 -   138.52163300000001 =   -69.41571000000000(  -69.41571000000000 error [0]     0.00000000000000)
 -1.63220900000000 -  -877.60215200000005 =   811.96994300000006(  875.96994300000006 error [0]    64.00000000000000)
-90.31431300000000 -  -132.43194900000000 =    42.11763600000000(   42.11763600000000 error [0]     0.00000000000000)
-83.73474600000000 -  -230.68138900000000 =   130.94664299999999(  146.94664299999999 error [0]    16.00000000000000)
 86.18703100000000 -   644.62607000000003 =  -558.43903900000009( -558.43903899999998 error [0]     0.00000000000011)
-34.09200400000000 -   255.30064300000001 =  -289.39264700000001( -289.39264700000001 error [0]     0.00000000000000)
-72.66596400000000 -  -653.34317699999997 =   516.67721300000005(  580.67721299999994 error [0]    63.99999999999989)
-34.55830500000000 -  -720.73328400000003 =   686.17497900000001(  686.17497900000001 error [0]     0.00000000000000)
 98.75237900000000 -  -489.76189599999998 =   588.51427499999988(  588.51427500000000 error [0]     0.00000000000011)
 37.58187400000000 -   914.25961099999995 =  -812.67773699999998( -876.67773699999998 error [0]   -64.00000000000000)
-52.51786000000000 -   185.37176700000001 =  -237.88962699999999( -237.88962700000002 error [0]    -0.00000000000003)
-69.83965499999999 -   820.89594299999999 =  -826.73559799999998( -890.73559799999998 error [0]   -64.00000000000000)
 91.70158900000000 -  -969.10226899999998 =  1060.80385799999976( 1060.80385799999999 error [0]     0.00000000000023)
```

图4.9 未经加固的浮点加法器实验结果

```
 71.09006599999999 -  -153.88677100000001 =   224.97683699999999(  224.97683699999999 error [0]     0.00000000000000)
 74.89104300000000 -   632.36321599999997 =  -557.47217300000000( -557.47217300000000 error [0]     0.00000000000000)
-64.25487200000001 -   802.07541200000003 =  -866.33028400000001( -866.33028400000001 error [0]     0.00000000000000)
 52.46502600000000 -    15.29237700000000 =    35.83931566666666(   37.17264900000000 error [1]     1.33333333333334)
-37.81783400000000 -  -235.16367800000000 =   192.01251066666666(  197.34584400000000 error [1]     5.33333333333334)
-32.43875400000000 -  -313.77266800000001 =   281.33391400000005(  281.33391399999999 error [0]    -0.00000000000006)
 -7.36250900000000 -  -664.19859799999995 =   656.83608899999990(  656.83608899999990 error [0]     0.00000000000000)
-24.70051500000000 -  -872.76288199999999 =   826.72903366666674(  848.06236699999999 error [1]    21.33333333333326)
-40.75221800000000 -   605.76113899999996 =  -646.51335699999993( -646.51335699999993 error [0]     0.00000000000000)
 70.21422500000000 -   211.17549099999999 =  -140.96126599999999( -140.96126599999999 error [0]     0.00000000000000)
-43.19432000000000 -  -213.44413399999999 =   164.91648066666664(  170.24981399999999 error [1]     5.33333333333334)
 45.69002200000000 -  -665.43154500000003 =   689.78823366666666(  711.12156700000003 error [1]    21.33333333333337)
-12.17389600000000 -   943.51132800000005 =  -955.68522399999995( -955.68522400000006 error [0]    -0.00000000000011)
-95.09371899999999 -   936.59134100000006 = -1031.68506000000002(-1031.68506000000002 error [0]     0.00000000000000)
 59.85816000000000 -    13.54330200000000 =    46.31485799999999(   46.31485800000000 error [0]     0.00000000000001)
-39.98116100000000 -   -76.92790599999999 =    34.28007833333333(   36.94674499999999 error [1]     2.66666666666666)
 50.72457000000000 -  -546.07448799999997 =   575.46572466666657(  596.79905799999995 error [1]    21.33333333333337)
-80.91299400000000 -   238.96105000000000 =  -314.54071066666666( -319.87404400000003 error [1]    -5.33333333333337)
-85.25353400000000 -  -216.84074000000001 =   131.58720600000001(  131.58720600000002 error [0]     0.00000000000000)
 69.10592300000000 -   138.52163300000001 =   -64.08237666666668(  -69.41571000000000 error [1]    -5.33333333333333)
 -1.63220900000000 -  -877.60215200000005 =   854.63660966666680(  875.96994300000006 error [1]    21.33333333333326)
-90.31431300000000 -  -132.43194900000000 =    36.78430266666668(   42.11763600000000 error [1]     5.33333333333333)
-83.73474600000000 -  -230.68138900000000 =   141.61330966666668(  146.94664299999999 error [1]     5.33333333333331)
 86.18703100000000 -   644.62607000000003 =  -558.43903899999998( -558.43903899999998 error [0]     0.00000000000000)
-34.09200400000000 -   255.30064300000001 =  -289.39264700000001( -289.39264700000001 error [0]     0.00000000000000)
-72.66596400000000 -  -653.34317699999997 =   559.34387966666668(  580.67721299999994 error [1]    21.33333333333326)
-34.55830500000000 -  -720.73328400000003 =   686.17497900000012(  686.17497900000001 error [0]    -0.00000000000011)
 98.75237900000000 -  -489.76189599999998 =   577.84760833333337(  588.51427500000000 error [1]    10.66666666666663)
 37.58187400000000 -   914.25961099999995 =  -855.34440366666661( -876.67773699999998 error [1]   -21.33333333333337)
-52.51786000000000 -   185.37176700000001 =  -237.88962699999999( -237.88962700000002 error [0]    -0.00000000000003)
-69.83965499999999 -   820.89594299999999 =  -869.40226466666661( -890.73559799999998 error [1]   -21.33333333333337)
 91.70158900000000 -  -969.10226899999998 =  1039.47052466666673( 1060.80385799999999 error [1]    21.33333333333326)
```

图4.10 加固后的浮点加法器实验结果

从图 4.10 中的实验结果中可以看到，加固后的浮点加法器能够检测到尾数运算过程的出现的单比特错误，图中所有运算结果与未注错时的运算结果不相等的测试用例中均输出“error [1]”，表示运算过程出错并且错误被检测到。同时，从图 4.10 中可以看到，部分测试用例虽然被注入了错误，但是运算结果并没有出错，造成这个现象的原因是浮点数本身具有一定的容错能力，注入的错误在浮点数运算过程中被丢弃，因此运算结果没有出错。

综上所述，使用乘积码对浮点加法器进行容错设计，并且对乘积码方法无法应用到的电路采用双模冗余方法进行加固，这种加固方法可以提高浮点加法器的容错能力，可以检测到运算过程中出现的单比特错误。

4.4 本章小结

本文对浮点加法器中的重要模块进行了加固，根据不同子模块的特点采用了不同的加固方法，对符号位和阶码的运算电路采用了双模冗余方法进行加固，对于尾数的运算电路采用了基于乘积码和双模冗余的方式进行加固。在尾数运算电路的编码器和解码器的设计方面，本文优化了电路，编码器中的乘法运算转换为移位运算和加法运算，解码器中的除法运算采用了基于查找表的快速除三电路，可以在一个时钟周期内完成编码和解码操作。由于浮点数尾数运算过程中需要考虑舍入和尾数规格化，本文通过改进 lop 模块和尾数运算模块，将编码后的数据直接进行运算，运算后在解码前进行尾数规格化，并设计了纠正电路对最终结果进行修正。最后在 SMIC 的 180nm

工艺库下对加固后的浮点加法器进行综合，从综合后的结果可以看到，加固后的面积增加了 51.98%，路径延时增加了 60.99%，相比于仅使用双模冗余对整个浮点加法器进行加固，面积减少了将近一半。

第五章 总结与展望

5.1 总结

本文首先对 IEEE 754 浮点数标准和 RISC-V 浮点扩展指令集进行研究，对浮点运算单元中进行容错的必要性和可行性进行分析，将浮点数运算过程根据数据路径划分为两个阶段：浮点寄存器堆读写、浮点数算术运算，并针对这两个阶段分别进行容错设计。本文对浮点寄存器堆的访问间隔进行统计，浮点寄存器堆中存在长时间未被改写的寄存器，只对寄存器的输出进行纠正而不刷新寄存器数据时，会存在错误累积问题。此外，对浮点寄存器堆故障敏感性进行评估，实验结果表明，基于 IEEE 754 标准的浮点数中尾数的低位中出现的错误对整体运算结果的影响较小，符号、阶码、尾数的高位中出现的错误对整体运算结果的影响较大，这一现象是由于浮点数运算过程中的对阶操作和舍入操作导致的，因此可以对浮点寄存器的不同位置采用不同的容错策略，从而在可靠性和资源开销上达到最优。

针对浮点寄存器堆，本文提出了一种基于 Cache 结构的寄存器容错设计方案，使用处理器中已有的数据 Cache 存放寄存器堆中的数据备份，其优势在于复用了处理器中的存储单元，减少了资源消耗。此外，这种方法实现了对寄存器堆中错误数据的刷新，从根本上解决了寄存器堆中错误的累积问题，寄存器堆中的错误会被及时纠正，因此错误不会产生累积，不会出现错误无法被检测的现象。此外，随着半导体工艺的不断发展，空间中单个高能粒子会对存储器近邻单元造成影响，导致多位翻转。针对这种情况，本文设计了一种基于交叠编码技术的校验器，通过对校验间隔的配置，实现对近邻单元翻转错误的容错，从实验结果中可以看到，使用不同交叠间隔的校验器可以有效检测到连续 4 比特和 8 比特的翻转错误，并且通过树形解码，编码器的解码器的电路的延时较小。

针对浮点数运算单元，本文采用双模冗余和乘积码结合的方式对浮点加法器进行容错设计，可以检测运算过程中出现的单比特错误，相比于只使用双模冗余进行加固，面积开销更小。传统的乘积码解码器需要使用除法器完成对操作数的解码，但是除法器的实现较为复杂，需要多个时钟周期才能完成除法操作，无法在算术运算单元中进行应用。本文对除三电路进行改进，提出一种无减法器的快速除三电路，使用查找表来匹配除三运算，使得除三运算可以在单个周期内完成，与未经优化的除三电路相比，在面积消耗和路径延时上均具有优势。

5.2 展望

寄存器堆作为现代处理器架构中的必要部分，其作用是暂存处理器执行过程中的操作数和运算结果，处理器对寄存器堆的访问频繁，并且速度较快，因此无法针对寄存器堆进行复杂的容错设计，限制了寄存器堆容错能力的提高。目前针对 Cache 和内存已经进行了较为全面容错研究，在后续的工作中，将主要对“寄存器堆—Cache—内存”的存储结构进行协同容错研究。

在实际的浮点运算单元中，一般配置有专用的浮点乘法器和浮点除法器，以完成更加快速的浮点数乘法运算和浮点数除法运算。在后续的研究工作中，将对专用的浮点乘法器和浮点除法器的容错方法进行研究。

参考文献

[1] AT697F, Rad-Hard 32 bit SPARC V8 Processor. [Online] Available: https://www.microchip.com [Accessed July 19, 2022].

[2] GR740, Quad-Core LEON4 SPARC V8 Processor. [Online] Available: https://www.gaisler.com/gr740 [Accessed July 19, 2022].

[3] RAD750, Radiation-hardened PowerPC microprocessor. [Online] Available: https://www.baesystems.com [Accessed July 19, 2022].

[4] RAD5545, SpaceVPX multi-core single-board computer. [Online] Available: https://www.baesystems.com [Accessed July 19, 2022].

[5] Andersson J. Development of a NOEL-V RISC-V SoC targeting space applications[C]//2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). IEEE Computer Society, 2020: 66-67.

[6] Waterman A, Lee Y, Patterson D A, et al. The risc-v instruction set manual, volume i: Base user-level isa[J]. EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, 2011, 116.

[7] 刘畅, 武延军, 吴敬征, 等. RISC-V 指令集架构研究综述[J]. 软件学报, 2021, 32(12): 3992-4024.

[8] Yang M, Hua G, Feng Y, et al. Fault tolerance techniques for spacecraft control computers[M]. John Wiley & Sons, 2017.

[9] Johnston A H. Scaling and technology issues for soft error rates[J]. 2000.

[10] O'Bryan M V, LaBel K A, Reed R A, et al. Recent radiation damage and single event effect results for candidate spacecraft electronics[C]//2001 IEEE Radiation Effects Data Workshop. NSREC 2001. Workshop Record. Held in conjunction with IEEE Nuclear and Space Radiation Effects Conference (Cat. No. 01TH8588). IEEE, 2001: 82-99.

[11] O'Bryan M V, LaBel K A, Ladbury R L, et al. Current single event effects and radiation damage results for candidate spacecraft electronics[C]//IEEE Radiation Effects Data Workshop. IEEE, 2002: 82-105.

[12] Dupont E, Nicolaidis M, Rohr P. Embedded robustness IPs for transient-error-free ICs[J]. IEEE Design & Test of Computers, 2002, 19(03): 56-70.

[13] Aranda L A, Wessman N J, Santos L, et al. Analysis of the critical bits of a RISC-V processor implemented in an SRAM-based FPGA for space applications[J]. Electronics, 2020, 9(1): 175.

[14] SoC2008. SoC2008 型 32 位片上系统芯片用户手册[Z]. 北京控制工程研究所. 2013.

[15] BM3823. 300MHz 抗辐照 SPARC CPU 产品使用手册[Z]. 北京微电子技术研究所. 2019.

[16] Gallagher W L, Swartzlander E E. Fault-tolerant Newton-Raphson and Goldschmidt dividers using time shared TMR[J]. IEEE Transactions on Computers, 2000, 49(6): 588-595.

[17] Eibl, Patrick J., Andrew D. Cook, and Daniel J. Sorin. "Reduced precision checking for a floating point adder." 2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. IEEE, 2009.

[18] Seetharam, Kushal, et al. "Applying reduced precision arithmetic to detect errors in floating point multiplication." 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing. IEEE, 2013.

[19] Maniatakos M, Kudva P, Fleischer B M, et al. Low-cost concurrent error detection for floating-point unit (FPU) controllers[J]. IEEE Transactions on Computers, 2012, 62(7): 1376-1388.

[20] Liang X G, Gao Y K, Hua G X. Partial-TMR: A New Method for Protecting Register Files Against Soft Error Based on Lifetime Analysis[J]. Journal of Computer Science and Technology, 2021, 36(5): 1089-1101.

[21] Montesinos P, Liu W, Torrellas J. Using register lifetime predictions to protect register files against soft errors[C]//37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). IEEE, 2007: 286-296.

[22] Santos D A, Luza L M, Zeferino C A, et al. A low-cost fault-tolerant RISC-V processor for space systems[C]//2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS). IEEE, 2020: 1-5.

[23] Reviriego P, Pontarelli S, Maestro J A, et al. A method to construct low delay single error correction codes for protecting data bits only[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2013, 32(3): 479-483.

[24] Ramos A, Ullah A, Reviriego P, et al. Efficient protection of the register file in soft-processors implemented on Xilinx FPGAs[J]. IEEE Transactions on Computers, 2017, 67(2): 299-304.

[25] Microprocessor Standards Committee, 2019. 754-2019-IEEE Standard for Floating-Point Arithmetic.

[26] Carmichael C. Triple module redundancy design techniques for Virtex FPGAs[J]. Xilinx Application Note XAPP197, 2001, 1.

[27] Velazco R, Cheynet P, Ecoffet R. Effects of radiation on digital architectures: one year results from a satellite experiment[C]//Proceedings. XII Symposium on Integrated Circuits and Systems Design (Cat. No. PR00387). IEEE, 1999: 164-169.

[28] Katz R, Wang J J, McCollum J, et al. A SEU-Hard Flip-Flop for Antifuse FPGAs[C]//IEEE NSREC 2001. 2001.

[29] Hentschke R, Marques F, Lima F, et al. Analyzing area and performance penalty of protecting

different digital modules with Hamming code and triple modular redundancy[C]//Proceedings. 15th Symposium on Integrated Circuits and Systems Design. IEEE, 2002: 95-100.

[30] Das A, Touba N A. Layered-ECC: A class of double error correcting codes for high density memory systems[C]//2019 IEEE 37th VLSI Test Symposium (VTS). IEEE, 2019: 1-6.

[31] Anghel L, Alexandrescu D, Nicolaidis M. Evaluation of a soft error tolerance technique based on time and/or space redundancy[C]//Proceedings 13th Symposium on Integrated Circuits and Systems Design (Cat. No. PR00843). IEEE, 2000: 237-242.

[32] Houghton A. The engineer's error coding handbook[M]. Springer Science & Business Media, 2012.

[33] Cui Y, Lou M, Xiao J, et al. Research and implementation of SEC-DED Hamming code algorithm[C]//2013 IEEE International Conference of IEEE Region 10 (TENCON 2013). IEEE, 2013: 1-5.

[34] Hamming R W. Error detecting and error correcting codes[J]. The Bell system technical journal, 1950, 29(2): 147-160.

[35] Wilkerson C, Alameldeen A R, Chishti Z, et al. Reducing cache power with low-cost, multi-bit error-correcting codes[C]//Proceedings of the 37th annual international symposium on Computer architecture. 2010: 83-93.

[36] Reed I S. A class of multiple-error-correcting codes and the decoding scheme[R]. Massachusetts Inst of Tech Lexington Lincoln Lab, 1953.

[37] Muller D E. Application of Boolean algebra to switching circuit design and to error detection[J]. Transactions of the IRE professional group on electronic computers, 1954 (3): 6-12.

[38] Wang X, Gao X, Liang Z, et al. Fault-tolerant architecture for Cache: Summaries, Assessments and Trends[C]//Journal of Physics: Conference Series. IOP Publishing, 2021, 2113(1):012068.

[39] T-head-Semi. opene906. URL, https://github.com/T-headSemi/opene906.git. 2021.

[40] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: A free, commercially representative embedded benchmark suite[C]//Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538). IEEE, 2001: 3-14.

[41] GCC, the GNU Compiler Collection. [Online] Available: https://gcc.gnu.org [Accessed July 26, 2022].

[42] Farbeh H, Delshadtehrani L, Kim H, et al. ECC-United cache: Maximizing efficiency of error detection/correction codes in associative cache memories[J]. IEEE Transactions on Computers, 2020, 70(4): 640-654.

[43] Parhami B. Computer Arithmetic: Algorithms and Hardware Designs, Second Edition. Oxford

University Press, 2010.

[44] Bruguera, Javier D., and Tomas Lang. "Leading-one prediction with concurrent position correction." IEEE Transactions on Computers 48.10 (1999): 1083-1097.

西安电子科技大学
XIDIAN UNIVERSITY

地址：西安市太白南路2号
邮编：710071
网址：www.xidian.edu.cn