**Cortex-R4加固总结报告**

**2023年6月**

目 录

[第1章 Cortex-R4 SEU敏感性分析 1](#_Toc136804475)

[1.1 Cortex-R4处理器分析 1](#_Toc136804476)

[1.1.1 Cortex-R4内核结构 1](#_Toc136804477)

[1.1.2 Cortex-R4流水线结构 2](#_Toc136804478)

[1.1.3 Cache结构 4](#_Toc136804479)

[1.2 最小系统SoC搭建 4](#_Toc136804480)

[1.2.1 SoC硬件平台结构 4](#_Toc136804481)

[1.2.2 SoC软件平台结构 6](#_Toc136804482)

[1.3 功能验证 11](#_Toc136804483)

[1.4 软错误敏感性分析 11](#_Toc136804484)

[1.4.1 搭建仿真故障注入平台 11](#_Toc136804485)

[1.4.2 软错误敏感性分析结果 14](#_Toc136804486)

[1.4.2.1 流水线分析结果 14](#_Toc136804487)

[1.4.2.2 Cache分析结果 18](#_Toc136804488)

[第2章 流水线与寄存器堆SEU容错设计 19](#_Toc136804489)

[2.1 基于Lockstep的加固方案 19](#_Toc136804490)

[2.2 流水线Lockstep加固设计 22](#_Toc136804491)

[2.2.1 恢复设计整体结构 22](#_Toc136804492)

[2.2.2 硬件部分恢复错误设计 24](#_Toc136804493)

[2.2.3 软件部分恢复错误设计 26](#_Toc136804494)

[2.3 自恢复三模冗余加固设计 27](#_Toc136804495)

[第3章 Cache SEU容错设计 30](#_Toc136804496)

[3.1 加固仿真验证 30](#_Toc136804497)

[第4章 容错效果评估与开销对比 31](#_Toc136804498)

[4.1 流水线与寄存器堆容错效果分析 31](#_Toc136804499)

[4.1.1 故障注入结果及分析 31](#_Toc136804500)

[4.1.2 资源利用与错误恢复时间分析 32](#_Toc136804501)

[4.2 Cache容错效果分析 35](#_Toc136804502)

# Cortex-R4 SEU敏感性分析

## Cortex-R4处理器分析

### Cortex-R4内核结构

Cortex-R4处理器内核具有指令预取和指令队列、分支预测等结构，按结构可分为预取指单元(Prefetch Unit, PFU)、数据运算单元(Data Processing Unit, DPU)、存取单元(Load/Store Unit, LSU)以及L1存储系统接口(包括L1级Cache接口，与ATCM、B0TCM、B1TCM的接口)、L2存储系统接口(包括L2级AXI Slave和AXI Master接口)，其内核结构如图1‑1所示。

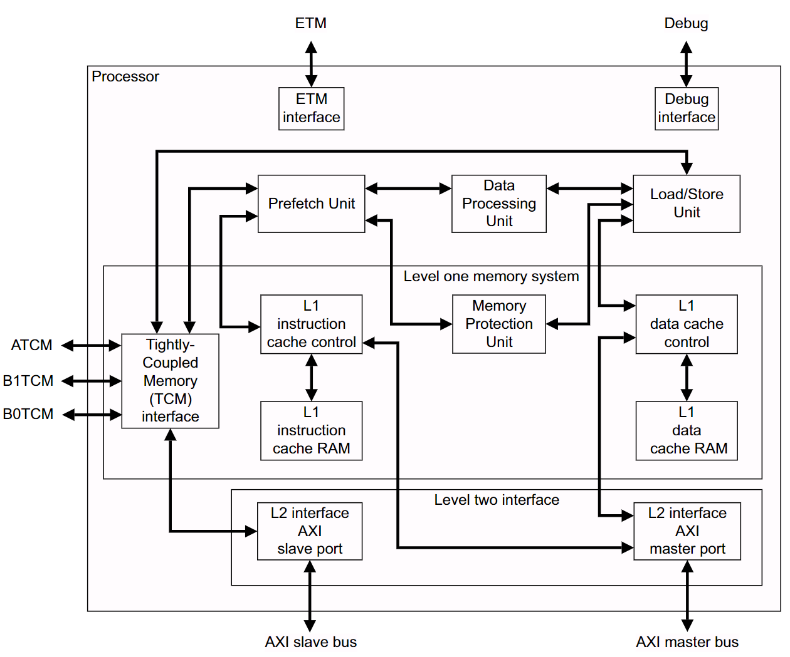


图1‑1 Cortex-R4内核结构框图

预取指单元(PFU)负责从指令存储器中获取64位宽的指令，从中提取指令(包括ARM和Thumb-2指令)，并向数据运算单元(DPU)提供指令流，同时具有分支预测器。PFU设计用于和L1 Cache与TCM交互数据，其中典型的访问时间是两个时钟周期。PFU内部包括两级预取(Fetch)和一级预译码(Pre-decoding)，在预译码之后会将指令推入指令队列(IQ, instruction queue)。指令队列(IQ)是一个FIFO，一次可读写一条或两条指令，IQ最多可存储八个预译码后的指令，是PFU和DPU交互数据的重要部件，但IQ并不属于流水线阶段。PFU中有两个预译码组件，因此可以一次将64位数据预译码并将指令写入指令队列中。通过这种方式，整个流水线能够持续每个周期吞吐两条指令，既不会死等数据也不会因等待流水线而停顿。此外，在预译码阶段会进行分支预测，PFU有两种预测器，Taken-ness预测和目标地址预测。Taken-ness预测适用于目标地址与PC有固定偏移量的分支预测，而目标地址预测则用于函数返回。此外，PFU包含一个三输入的64位预取缓冲区(pre-fetch buffer)，只要预取缓冲区中有空间，PFU就能持续获取下一个64位指令。

数据运算单元(DPU)负责指令译码、发射、执行等内容。指令会经过译码阶段进行完全的译码，得到立即数等数据。之后会进入发射(issue)阶段，此阶段控制信号会根据指令类别的不同发往不同的通路，即存取、数据处理、浮点运算。流水线上都会有ex1(execute1)、ex2(execute2)、wr(writeback)三个流水线阶段，但具体通路间是独立的，可以达到同时执行不同指令的目的，即超标量双发射执行。在最终向寄存器堆写入数据时，是通过一个选择器来选择写入的数据来自哪个通路。

存取单元(LSU)负责所有加载(load)和存储(store)指令操作，分别与DPU、TCM和存储系统有接口。LSU模块只负责所有和存储相关的指令，因此减少了流水线停顿的时间。

### Cortex-R4流水线结构

Cortex-R4是32位超标量RISC处理器，具有8级流水线，如果启用了浮点运算单元(FPU)则会增加一个流水线周期用于浮点结果的写回，Cortex-R4的流水线结构如图1‑2所示。下面对流水线各个阶段进行介绍。

预取指阶段：在8级流水线中有两个阶段都是预取指令的阶段，PFU从指令存储器中获取64位宽的指令数据，并存储在缓冲区中，并将指令送往预译码阶段。

预译码阶段：对指令进行简单的译码，区分出指令是属于ARM指令集还是Thumb指令集，同时在此阶段还会进行分支预测，以减少分支指令所带来的时间消耗。

译码阶段：对指令进行全译码，获得立即数等所需数据。

发射阶段：在发射阶段将流水线根据不同的指令分为不同的路径通路。

执行和写回阶段：分为四个并行的路径和浮点运算路径，由于本课题使用的Cortex-R4内核并未启用浮点运算模块，因此分析其他四个并行路径通路。Load/Store路径处理所有的内存访问，分为三个流水线周期，最后一个周期会更新寄存器堆。Load/Store路径在指令发射阶段就会生成地址，以平衡其他流水线路径，这样可以减少数据冒险的延迟。Mac(Multiply-accumulates)路径负责处理乘法累加，正常来说共三个流水线周期，但如果相邻的下一条指令要使用Mac结果，那么会多增加一个周期用于加载。ALU路径则负责算术和逻辑运算，包括预移位和基本的ALU运算。Divider路径则负责除法运算，使用Radix-4算法，在单个流水线阶段典型的32位除法大约需要6个周期。



图1‑2 Cortex-R4流水线结构图

### Cache结构

## 最小系统SoC搭建

### SoC硬件平台结构

最小系统SoC结构如图1‑3所示，包括了AXI 总线互连、APB总线、ROM、RAM等必要组件。



图1‑3 最小系统SoC结构图

Cortex-R4内核中很多组件和功能是可配置的，这是通过设置宏定义来实现的。由于很多功能是ARM所特有的技术，例如紧耦合存储器(TCM)，对这些组件进行抗辐射加固设计并没有普遍性和通用性。同时还有一些功能对于组成可运行的最小系统没有太大帮助，因此对Cortex-R4内核进行相关配置，如表1‑1所示。

表1‑1 Cortex-R4内核配置

|  |  |  |
| --- | --- | --- |
| 配置选项 | 数值 | 含义 |
| ICACHE\_NONE | 1 | 具有奇偶校验功能的指令Cache |
| DCACHE\_NONE | 1 | 具有奇偶校验功能的数据Cache |
| NO\_A\_TCM\_INF | 1 | 不包含ATCM端口 |
| ATCM\_NONE | 1 | 不包含ATCM内存 |
| NO\_B0\_TCM\_INF | 1 | 不包含B0TCM端口 |
| NO\_B1\_TCM\_INF | 1 | 不包含B1TCM端口 |
| BTCM\_NONE | 1 | 不包含BTCM内存 |
| TCM\_HI\_INIT\_ADDR | 20’h4000 | TCM默认地址 |
| NO\_IE | 1 | 不支持大端(big-endian)指令 |
| NO\_FPU | 1 | 不包含浮点运算单元(FPU) |
| MPU\_REGIONS | 12 | 包含12个内存管理单元(Memory Protection Unit, MPU)区域 |
| BREAK\_POINTS | 8 | 共8个断点数量 |
| WATCH\_POINTS | 8 | 共8个监视点数量 |
| PERPH\_REV\_AND | 4’h0 | ECO修复版本号 |
| NO\_SLAVE | 1 | 不包含AXI Slave端口 |

TCM内存是ARM独有技术，虽然可以带来性能的提升，但对流水线加固设计并不具有通用性，因此未启用ATCM、B0TCM、B1TCM等组件。AXI Slave端口是用于和直接内存访问设备(Direct Memory Access, DMA)之间的通信，因此也未启用。在MPU设置上共启用了12个内存管理单元区域，可以对不同的区域设置不同的访问特性(包括是否支持特权访问等)和内存属性(如可缓存等)，以此来对内存进行保护。

Cortex-R4处理器采用了哈佛结构，即指令存储器和数据存储器是相互分开独立的，因此最小系统SoC需要两个片上存储器，ROM用于存储指令代码，RAM用于存储数据。

CRG(Clock Reset Generator, 时钟复位产生模块)是SoC设计中的关键部分，负责整个芯片的时钟和复位的产生，在常见的单片机(Microcontroller Unit, MCU)芯片设计中也是关键的组件。通常CRG模块在时钟方面会进行时钟门控、时钟分频和时钟切换等工作，在复位方面会进行同步释放、复位控制等工作。将芯片重要的时钟和复位控制放在一个模块有利于时序的调整和后端流程的进行，同时MCU芯片对低功耗有较高要求，CRG模块可以对时钟进行关断，降低功耗，可以利用工艺库中的ICG(Integrated Clock Gating, 集成时钟门控)模块进行时钟门控，避免手动写的HDL代码产生毛刺(glitch)等问题。最小系统SoC中的CRG模块只包含时钟门控和复位同步释放功能。

Cortex-R4使用的是AXI-3(Advanced eXtensible Interface)64位高速总线，主机和从机间通过总线进行通信。通常各主机和从机间会有总线互联矩阵(Bus Matrix)，主要负责主机仲裁、从机地址译码等。最小系统SoC未包含其他主机设备，因此只包含地址译码、状态机控制等内容。常见的低速外设一般挂在APB(Advanced Peripheral Bus)32位总线上，在和AXI总线交互时需要AXI-APB的总线桥。

最小系统SoC包含了常见的通用外设UART和Timer等，挂在APB总线上。UART(Universal Asynchronous Receiver/Transmitter, 通用异步收发传输器)是常见的异步串口设备，MCU中经常用于调试信息的打印串口。Timer定时器是计时功能，是MCU不可缺少的必要外设。最小系统SoC的UART和Timer外设均为ARM的CMSDK(Cortex-M System Design Kit, Cortex-M系统设计工具包)开源外设。

### SoC软件平台结构

SoC不止需要硬件平台，还需要可以运行程序的软件平台。在嵌入式开发中，程序代码通过编译工具进行编译链接后得到可执行文件，之后在仿真时加载到指令存储器中。

嵌入式开发常用的编译器是GCC(GNU Compiler Collection)，而且有针对裸机ARM处理器的gcc-arm-none-eabi交叉编译工具。编译器负责将C/C++语言和汇编语言的程序翻译成处理器上可执行的机器码文件。GCC是一个强大的工具集合，它由多个组件组成，包括预处理器、编译器、汇编器、链接器等。但在ARM处理器中，ARM公司开发了一款编译工具ARMCC(ARM C/C++ Compiler)，在编译时对ARM指令集等有相关优化，也是ARM处理器主流的编译器。此外，ARMCC还集成在Keil以及ARM DS IDE里面，因此在嵌入式开发过程中，当使用Keil软件或ARM DS IDE开发时，默认的内嵌编译器就是ARMCC。

由于Cortex-R4处理器是ARM处理器中典型且已成功商用的一款，而且为了和嵌入式开发相匹配，选择ARMCC作为编译工具。

SoC软件平台各部分和功能如表1‑2所示，主要包括启动文件、链接脚本文件、printf重定向文件、Cache设置文件、C程序的头文件等。

表1‑2 SoC软件平台各部分功能表

|  |  |
| --- | --- |
| 文件名称 | 功能 |
| Init.s | 启动文件 |
| InitCache.s | Cache使能 |
| scatter.scat | 分散加载文件 |
| retarget.c | printf重定向 |
| startup.c | 进入main函数之前的额外启动工作 |
| common.c | 共用的函数声明、函数原型、宏定义等 |
| common.h | common.c的头文件 |
| sim\_stdout.c | 仿真printf重定向具体实现 |
| sim\_stdout.h | sim\_stdout.c的头文件 |

首先分别对各文件进行编译，汇编文件“Init.s”、“InitCache.s”使用命令“armasm”进行编译，C程序文件“startup.c”等使用命令“armcc”进行编译。编译后得到相同名称的目标代码文件，目标代码需要通过链接才可转化为可执行的映像文件，链接通过使用命令“armlink”来完成。在链接过程中，需要使用分散加载文件“scatter.scat”，代码如下：

ROM\_LOAD 0x0

{

INIT 0x0

{

\*.o (RESET, +First)

\* (InRoot$$Sections)

\* (+RO)

}

DATA 0x40000000 0x10000

{

\* (+RW, +ZI)

}

ARM\_LIB\_STACKHEAP 0x40010000 EMPTY 0x10000

{}

}

分散加载文件“scatter.scat”主要用于提供ARM链接器生成映像文件时的相关信息。分散加载文件分了三个内存空间，“INIT”指定启动代码等程序代码的空间，“DATA”指定已经初始化的静态变量和未初始化的静态变量等数据空间，“ARM\_LIB\_STACKHEAP”指定堆栈空间的地址和大小。

Cortex-R4处理器复位后默认从地址0x0开始取指，因此“INIT”段的起始地址是0x0。因为SoC的RAM地址在0x40000000，因此“DATA”段的起始地址和RAM地址一致。

在经过链接后会得到可执行的ELF文件，但为了能在HDL仿真时使用，需要使用命令“fromelf”将其转化为二进制文件。最后将生成的二进制文件通过Verilog的关键字readmemh在仿真开始时初始化指令存储器，这样就可以在仿真时运行所需要的程序，代码如下所示。

|  |
| --- |
| initial begin : loadbin\_block  $readmemh(“main.ini”,dut.isram.mem);  end |

为了可以自动化进行程序的编译链接和仿真的运行，使用Makefile管理整个过程。只需在终端输入命令“make”就可以完成软件的编译、链接以及HDL仿真的过程。

在处理器开始运行后，整个软件平台运行过程如图1‑4所示。程序开始运行后，Cortex-R4从地址0x0开始取指令，0x0地址就是保存中断向量表的地方，会首先执行复位中断处理函数“Reset\_Handler”，在中断处理函数里会进行初始化寄存器堆，设置六个特权模式和用户模式的堆栈指针地址和大小。完成初始化的工作之后，将链接脚本的链接符号导入到MPU对应的区域中，同时在“Init.s”文件中还定义了各特权模式的处理弱函数(WEAK)，这样在ARMCC工具链接时会首先链接定义为非WEAK的同名函数，如果找不到则链接WEAK函数，这样的定义可以使得在C程序里直接使用各特权模式的处理函数。

在完成各种初始化工作之后，程序应该会跳转到主函数，但在此之前还需要完成Cache使能工作。C语言中可以使用“$Sub$$main”在主函数前插入一段新代码，Cache使能的工作就在这里进行。使用“$Sub$$main”的方式去进行使能Cache等其他设置工作的原因是需要等待“Init.s”里的全部初始化任务完成，包括数据的搬运等工作。

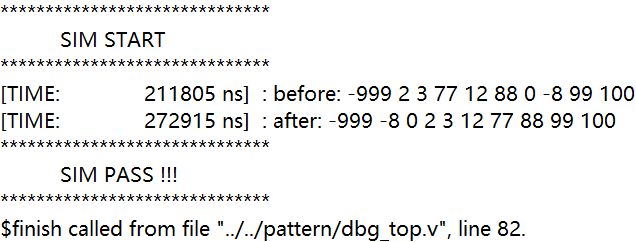
在主函数中是用户的代码空间，可能会使用到共用的函数声明、函数原型、宏定义等内容，或者是printf重定义的头文件，直到整个程序运行结束。



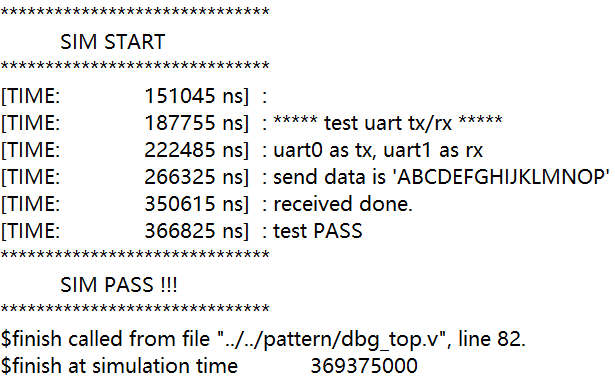
图1‑4 软件平台运行过程

## 功能验证

对最小系统SoC运行冒泡排序算法程序和基本UART传输测试，以确保SoC的功能正确性。结果正确，如图1‑5所示，说明搭建的SoC功能正确。



a) 冒泡排序测试结果



b) UART测试结果

图1‑5 功能验证结果

## 软错误敏感性分析

### 搭建仿真故障注入平台

无论是评估系统的可靠性还是评估抗辐射设计的加固效果，都需要有正确的测试平台，但在设计的不同时期进行评估测试的成本也不同，在设计初期利用仿真的形式进行故障注入模拟是常用的低成本测试方法。故障注入是人工将故障注入至设计中，并观察在故障注入前后系统的运行情况，可以得到故障对系统的影响以及加固设计的效果。基于仿真的故障注入方法可以较容易控制故障的数量和发生频率，可以更快的得到评估结果。

基于此，使用Perl语言开发了仿真故障注入平台，该平台如图1‑6所示。该平台主要模拟常见的单粒子效应发生的情况，重点是时序电路发生错误的情况。单粒子效应对时序电路产生的影响主要是可能会导致寄存器保存的数据发生翻转，直到寄存器再次被写入新数据，而在这一段时间内寄存器错误的数据可能会传至整个系统，从而导致系统发生错误乃至整体失效。



图1‑6 仿真故障注入平台

仿真故障注入平台通过获取用户输入的设计相关信息以及故障注入相关参数，就可以找出设计中所有寄存器并对其进行仿真故障注入。平台具有两种注入模式，分别为单寄存器注入和模块注入，两种故障注入的模式可以使得用户根据不同需求而选择。单寄存器注入是在一次的程序运行过程中，只对一个寄存器进行故障注入。模块注入则是在一次的程序运行过程中，对一个模块内所有寄存器进行随机故障注入。平台首先会获取用户输入参数，例如故障注入的模式、要故障注入的模块、每个寄存器故障注入的次数、程序运行的次数、故障注入的开始结束时间等，在获得这些必要的参数后，平台在每次运行仿真前都会生成随机的testbench，之后调用外部的系统命令运行仿真，当运行次数达到用户输入的参数时停止，最后分析每次仿真的结果统计软错误率情况并生成报告。

仿真故障注入是通过在testbench中随机生成多个initial块语句，如图1‑7所示。在Verilog语法中，force和release关键字可以对信号进行强制赋值和解除强制赋值。每个initial块是对一个寄存器进行强制数据翻转，而翻转发生的时间是随机产生的。同时针对多位宽的寄存器，会随机选取连续的部分比特进行故障注入，这也是模拟单粒子效应对多比特寄存器的影响。

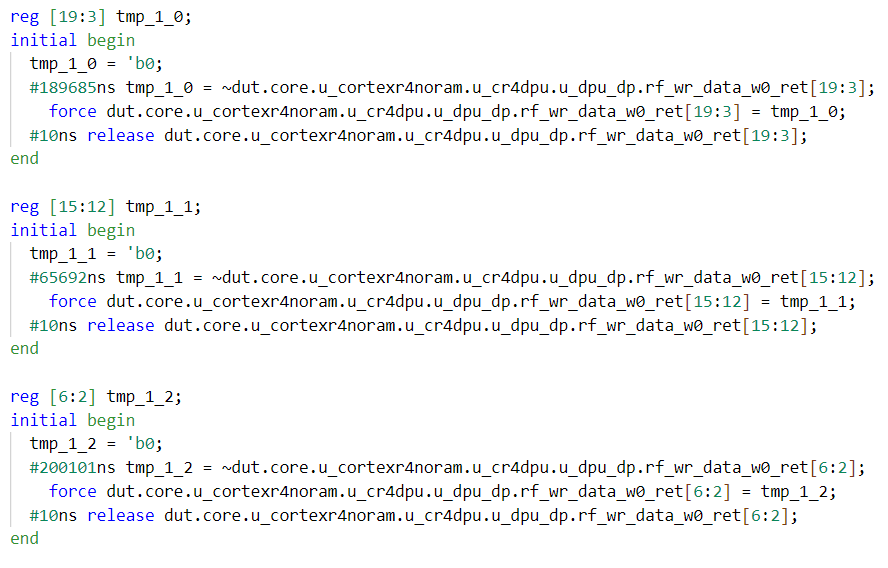


图1‑7 生成的随机故障注入代码示例

仿真故障注入平台具体子函数的功能说明如表1‑3所示，可以看出，为了使得仿真故障注入平台具有通用性以及更好的移植性，在设计中很多地方都使用到了递归函数。譬如，对于一个RTL电路设计而言，模块例化的层次关系是未知的，用户是不知道例化最深层次的是哪个模块；譬如，在Verilog宏定义代码块中，关键字`ifdef与`endif之间还有可能包括另一个`ifdef，而宏定义代码块中究竟包含了多少个宏定义代码块，这也是未知的。但仿真故障注入平台使用了递归思想，很好的解决了这一问题，也使得该仿真故障注入平台具有很好的通用性，不仅仅只适用于特定的设计。

表1‑3 仿真故障注入平台子函数功能表

|  |  |
| --- | --- |
| 子函数名称 | 函数功能 |
| find\_include\_macro | 查找`include文件中的宏定义 |
| clear\_macro | 查找最小的宏定义代码块 |
| find\_macro | 递归查找已定义的宏定义代码块 |
| find\_module | 查找module，避免一个文件多个模块同时存在 |
| find\_inst | 递归查找设计的所有例化层次 |
| replace\_param | 替换参数为实际数值 |
| replace\_param\_in\_param | 递归查找参数中包含其他参数 |
| find\_signals | 查找出所有寄存器，并按单比特和多比特分类 |
| find\_always | 查找时序always代码块中的寄存器 |
| generate\_tb | 生成随机故障注入的testbench |
| print\_report | 生成详细的软错误率报告 |

因此，相较于其他故障注入工具而言，本仿真故障注入平台具有以下优势：

1. 通用性。通过使用递归思想，可以完整的梳理出设计中模块的例化层次关系，同时保留已定义的宏定义代码部分而删除未定义的宏定义代码片段，这样既可以不受无关宏定义相关影响又能保留用户定义的宏定义内容。这些功能使得仿真故障注入平台具有很好的通用性，可以适用于其他代码规范的设计。
2. 易用性。平台不需要指定特定的仿真工具，也不需要去改变很多原有的仿真环境，也无需更改RTL代码的位置。
3. 自动化运行。平台只需用户输入必要的参数，就可以自动地统计各模块的软错误率并生成详细的软错误率报告，完成整个故障注入的过程。

### 软错误敏感性分析结果

### 流水线分析结果

使用开发的仿真故障注入平台对Cortex-R4处理器在未加固的情况下进行故障注入，从而分析其软错误敏感性。其中，共进行了两个不同的测试程序，一个是冒泡排序算法，一个是UART基本传输程序，这两个均可以反应处理器大多数工作时的状态。冒泡排序算法的基本思想是通过比较相邻元素的大小，并根据需要交换它们的位置，以逐步将最大（或最小）的元素冒泡到合适的位置。重复执行这一过程，直到所有元素按照从小到大（或从大到小）的顺序排列好。冒泡排序算法可以反映处理器的软错误敏感性，这是因为在冒泡排序的执行过程中，涉及到了处理器常用的指令操作，如比较、跳转和存储指令等。而UART基本传输可以涉及到处理器控制相关外设的场景且UART是常用的串口。完整的故障注入参数如表1‑4所示。

表1‑4 故障注入参数

|  |  |  |
| --- | --- | --- |
| 测试程序 | 参数名称 | 参数内容 |
| uart\_test | 单次程序运行次数 | 50 |
| 故障注入区间 | 45000ns至350000ns |
| 故障注入位置 | Cortex-R4内核所有寄存器，共1778个 |
| 故障注入次数 | 单个寄存器故障注入5次 |
| 其他信息 | 单核，无Cache，无TCM |
| bubble\_sort | 单次程序运行次数 | 50 |
| 故障注入区间 | 87000ns至150000ns |
| 故障注入位置 | Cortex-R4内核所有寄存器，共1778个 |
| 故障注入次数 | 单个寄存器故障注入5次 |
| 其他信息 | 单核，64K I-Cache/D-Cache，无TCM |

经过仿真故障注入平台后可以得到各模块的软错误率。对得到的数据进行处理，以DPU、PFU等主要模块进行划分，就可以得到如图1‑8所示的结果，表示Cortex-R4处理器各模块由于模块的软错误而导致系统出错的平均错误率。

由结果可以分析得出，软错误率最高的模块为cr4caches\_axim，此模块主要负责内核和存储系统的数据交互，包括取值、加载和存储指令等，因此会有较高的错误率。而cr4mpu模块负责内存的管理，所以此模块下UART测试程序会比冒泡程序有更高的错误率。图1‑8中各模块的平均软错误率是将软错误率报告中属于该模块的所有测试结果进行平均计算，只能反应此模块平均的软错误率，而包含的小模块有可能会达到100%的软错误率，这些模块更需要在后续加固设计中给予特别注意。

图1‑8 Cortex-R4内核各模块平均系统错误率

DPU模块是Cortex-R4处理器流水线的核心部分，大部分流水线阶段均在此模块，其内部模块的平均系统错误率如图1‑9所示。

图1‑9 DPU内部模块平均系统错误率

由图1‑9可知，DPU内部有些子模块的平均错误率高达100%，最高的几个模块均是和处理器状态息息相关的，例如u\_cpsr是保存处理器程序状态寄存器CPSR的模块，并且内部还包括了各模式的SPSR寄存器，因此对流水线至关重要，u\_dpu\_regbank是Cortex-R4处理器寄存器堆所在模块，寄存器堆是处理器实时保存数据计算结果的寄存器，对程序也是至关重要的。其他子模块以及各子模块内部包括的模块统计软错误率方法和分析与此相似，在此不再赘述。

为了可以得到流水线更加具体的敏感性情况，对单个寄存器的错误率情况进行深入的分析，因此需要对寄存器进行单独故障注入。以u\_dpu\_ctl模块为例，其是流水线控制信号的所在模块，使用仿真故障注入平台对u\_dpu\_ctl模块的所有寄存器进行单独的故障注入，其注入时的参数与表1‑4一致，结果如图1‑10所示。

图1‑10 u\_dpu\_ctl模块内寄存器软错误率

由图1‑10可以看到，对于不同的寄存器，其发生错误而导致处理器系统出错的可能性是不同的。有些寄存器对流水线运行控制的重要性不高，且每个时钟周期都会重新写入，这些即使发生了错误也对整体影响不大，而有些寄存器只要发生了错误就会立刻传递至整个流水线，从而导致系统出错。例如寄存器instr\_wfi\_iss的错误率高达100%，几乎只要出错就会影响整个进程，因为这个寄存器是控制WFI指令相关的命令，WFI是在等待中断时事件，因此本不该等待中断却因为寄存器出错而导致处理器一直在等待中断，进程被卡死。对于这些不同软错误敏感性的寄存器后续要采取针对性且合适的加固方法。

### Cache分析结果

# 流水线与寄存器堆SEU容错设计

本章将详细描述整个Cortex-R4流水线加固设计的具体实现。整个方案的设计是基于Lockstep技术来实现容错功能，通过对已存在的两种Lockstep技术的分析，针对Cortex-R4流水线进行Lockstep的加固实现方法。Lockstep实现方法选择软硬件结合方式，在硬件方面对比处理器相关信号，同时需要及时保存处理器状态，当发生错误时停止保存处理器状态，产生中断。在软件方面执行恢复错误的程序，并恢复相关的PC值和处理器状态等。

最后，针对影响处理器状态和立刻影响流水线的寄存器，采用可自恢复的三模冗余结构，以确保此类寄存器不会影响处理器的正常运行。

## 基于Lockstep的加固方案

目前常见的Lockstep处理器架构有两种，如图2‑1所示。图2‑1 a)为共用存储系统结构，两个处理器以主从关系工作并且共享相同的输入和内存，主处理器可以访问外部模块和存储器，而从处理器不能访问。图2‑1 b)为独立的存储系统结构，主从处理器都有自己的独立内存，并且都可以访问外部模块。对比模块用于实时比较两个处理器接口的输出数据，当比较结果不一致时会产生中断或其他方式交由系统处理错误。两种结构都有优点，共用存储结构面积更小而且不需要解决共享部分的访问冲突，而独立存储结构还可以检测内存错误。

a) 共用存储系统结构 b) 独立存储系统架构

图2‑1 两种常见的Lockstep架构

除此之外，Lockstep结构也有可能不止两个处理器，但更多核的Lockstep结构也与此基本一致。不同的Lockstep结构都会通过对比相应的信号来判断两个处理器运行状态是否一致，而对比信号不一致后如何恢复软错误则是不同，恢复功能也是各种不同结构的核心功能点。目前商用或研究的Lockstep恢复功能主要有两种，基于软件检查点的恢复逻辑和全硬件Lockstep技术，下面简要分析这两种方法并比较各自优缺点。

基于检查点的Lockstep结构会在软件层面上将应用程序划分多个执行块(execution block)，每个执行块之间都有一个检查点(Verification Point, VP)，此外检查点在程序开头和结束还会存在。基于检查点的Lockstep运行的应用程序过程如图2‑2 b)所示，与普通程序流程即图2‑2 a)相比，增加的每个检查点会比较内核输出的信号以及处理器内部重要的状态寄存器，只有等待检查点通过后才可执行下一个代码块，同时会将此刻的处理器状态保存。如果检查点未通过，那么会产生新中断来执行程序回滚，将处理器恢复至上一个检查点并继续执行。



a) 普通程序流程 b) 带有检查点的程序流程

图2‑2 基于检查点的Lockstep架构应用程序运行过程

全硬件实现的Lockstep架构会实时保存处理器的状态，例如有文献提出了使用全硬件的Lockstep功率边缘计算处理器容错技术，在硬件中实时保存处理器的正确节点状态。全硬件的Lockstep架构当检测到发生错误时，两个处理器会复位或产生中断，然后恢复至上一个自动保存的正确节点重新执行，全硬件Lockstep架构的每个周期都可以看作类似软件的检查点。

两种实现方式有各自的优缺点，基于检查点的Lockstep技术实现方式相对简单明了，但需要更改软件程序以增加检查点，而且检查点的个数和分布在不同实现方式中差别很大。同时，基于检查点的Lockstep结构通常只会比较处理器输出的信号，对于使得处理器无法再响应的错误没有恢复办法。全硬件的Lockstep技术在性能和恢复时间方面具有明显的优势，并且和其他Lockstep的实现方式相比不需要增加过多的面积开销。但全硬件实现Lockstep技术需要完整全面地了解整个处理器内核的结构，因为要更改处理器内核的结构所以对技术人员的设计水平也有很高要求。同时全硬件实现的Lockstep技术不具有通用性，不同处理器内部结构的差别很大，较复杂的处理器结构实现全硬件的Lockstep技术难度不亚于重新设计处理器，也因此增加了大量内核更改后的验证工作。

同时，流水线中会存在影响处理器状态和立刻影响流水线的寄存器，如果这些寄存器发生了错误，Lockstep技术也无法保证及时正确地恢复错误，为了避免这些寄存器对流水线的影响，采用三模冗余或带有自恢复结构的三模冗余加固方法。基于以上分析，对不同流水线寄存器采取响应加固方法，方案如图2‑3所示。



图2‑3 Cortex-R4处理器流水线加固方案

因为大部分流水线寄存器均可以通过Lockstep方法进行恢复，下面针对流水线Lockstep设计方法进行说明。

首先，主(Master)处理器和从(Slave)处理器运行相同的指令，但Slave处理器会延迟一个周期运行，也就是说，Slave处理器内部所有信号的状态都应该和Master处理器前一个周期的状态相同。

其次，在两个处理器的比较信号选择上，不仅要对比处理器输出的信号，还要对比处理器内部重要的寄存器，包括程序状态寄存器、寄存器堆等。这样可以保证更加及时的检测到错误，以免处理器进入hang状态，即挂死，如果处理器因为发生错误而导致hang状态，也就是通常所说的死机，这个时候处理器已经无法响应任何指令，更不用谈执行恢复逻辑的可能，此时只有硬件复位之后再可恢复。

然后，在错误恢复逻辑上，选择软硬件结合的方式去实现。硬件需要及时保存处理器状态，当发生错误时停止保存处理器状态，同时产生FIQ中断，当进入中断处理函数中，会开始执行恢复错误的程序。软错误的恢复工作大部分都是由软件完成，这样既可以做到灵活实现，也可以避免过多的更改处理器内核结构。软件恢复程序需要做到快速而简洁，避免占用过多的时间。

最后，当恢复完成时，会退出中断恢复程序，同时PC会跳转到发生错误的指令处，重新执行发生错误的指令。在恢复完成时会同时恢复寄存器堆值，以确保和发生错误前的状态一致。

## 流水线Lockstep加固设计

### 恢复设计整体结构

为了使得流水线和寄存器堆加固设计在消耗资源较少的情况下有良好效果，首先要分析流水线中易受到软错误影响的结构。在微处理器流水线中，执行单元（例如加法器和乘法器）通常比保持处理器状态或流水线中停顿点的结构更不容易受到软错误的影响。这是因为两个原因，首先，执行单元主要由逻辑电路组成，它们具有更高级别的逻辑、电气和锁存窗口屏蔽。相比之下，保存架构状态或停顿点的结构由不具有许多这些掩蔽属性的状态位组成。第二个原因在于执行单元是指令的执行部分，不会让指令停顿，指令暴露的时间也少于其他流水线结构。例如预取指单元，通常处理器设计中取值部分都会包含缓存用于提升取值的能力，这些缓存和译码后的缓存可以看作指令的停顿，它们的暴露时间窗口要大于执行单元。因此，对于处理器流水线来说，具有停滞状态的结构是软错误保护的首要单元。

进一步对保持处理器状态和流水线停顿点结构进行分析，保持处理器状态的寄存器是流水线发生错误后能否恢复成功的关键，而流水线的停顿点结构要取决于所在位置，流水线发生错误恢复时总是会清空推测结果，例如分支预测器，假若错误发生在分支预测器又恰巧此条分支指令预测失败，那么分支预测器本身就会清空，会对软错误产生窗口屏蔽效果。

因此，恢复逻辑需要保持处理器状态，在检测到发生错误后执行恢复逻辑使得处理器可以回退至上一个正确的状态。整个恢复设计分为硬件部分和软件部分，硬件部分主要负责检测是否有软错误发生，保存处理器正确的状态，以及和软件恢复程序交互的接口。软件部分主要负责错误的恢复，将处理器回退至出错前的状态再重新执行。

针对流水线错误恢复设计整体结构如图2‑4所示，两个处理器对比信号选择寄存器堆写入数据、寄存器堆数据、预取指的指令等信号，通过对比得到相应的错误标志信号。在加固设计中，存在一个APB从机，用于和CPU相互通信，在未发生错误时CPU正确的状态信息会保存在APB从机中，一旦错误标志信号被拉高，证明错误恢复开始，进入FIQ中断模式，中断程序中通过读取APB从机的信息来恢复CPU的运行状态。



图2‑4 流水线错误恢复设计整体结构

### 硬件部分恢复错误设计

2.2.1节已经对错误恢复设计整体结构进行了介绍，在硬件部分的恢复错误设计中APB从机的保存的信号如表2‑1所示，保存了处理器核心的状态寄存器。

表2‑1 恢复逻辑中APB从机寄存器表

|  |  |  |
| --- | --- | --- |
| 寄存器名称 | 偏移地址 | 功能描述 |
| fault\_r | 0x0 | 是否发生了错误 |
| pc\_rec\_r | 0x4 | 保存的恢复PC值 |
| sp\_rec\_r | 0x8 | 保存的恢复堆栈指针SP |
| cpsr\_rec\_r | 0xc | 保存的恢复CPSR |

硬件部分恢复设计的其他相关信号如表2‑2所示，包含了恢复过程中重要的一些信号，以及用于恢复寄存器堆数值的信号，这些信号用于恢复流水线和寄存器堆发生软错误。

表2‑2 恢复逻辑其他信号表

|  |  |  |
| --- | --- | --- |
| 信号名称 | 是否为输入输出 | 功能描述 |
| regbank\_wdata\_fault\_flag | 否 | 流水线错误导致寄存器堆写入发生错误 |
| regbank\_fault\_flag | 否 | 寄存器堆发生错误 |
| pd\_ins\_fault\_flag | 否 | 预取指单元指令发生错误 |
| during\_rec\_flag | 否 | 表示正在恢复错误 |
| pc\_instr0\_wr\_i | 是 | 写回阶段的PC值 |
| cpsr\_ret\_ls\_i | 是 | 当前指令时的CPSR |
| ls\_nfiq\_o | 是 | FIQ中断信号 |
| regbank\_en\_ls\_rec\_o | 是 | 寄存器堆恢复使能信号 |
| regbank\_ls\_rec\_o | 是 | 寄存器堆恢复数值信号 |

为了简要说明硬件部分恢复软错误的设计，选择一段简单的汇编代码为例来说明，汇编的部分代码如下：

MOV r1, #0x11

MOV r2, #0x11

MOV r0, #0x22

MOV r1, #0x22

MOV r2, #0x22

汇编代码为依次向r0-r2写入数据“0x11”和“0x22”。通过故障注入平台对“MOV r0,#0x22”指令进行故障注入，使写入寄存器r0的值不是“0x22”，从而发生错误使得恢复逻辑启动。

执行恢复模块时的波形示意如图2‑5所示，下面进行简单介绍。

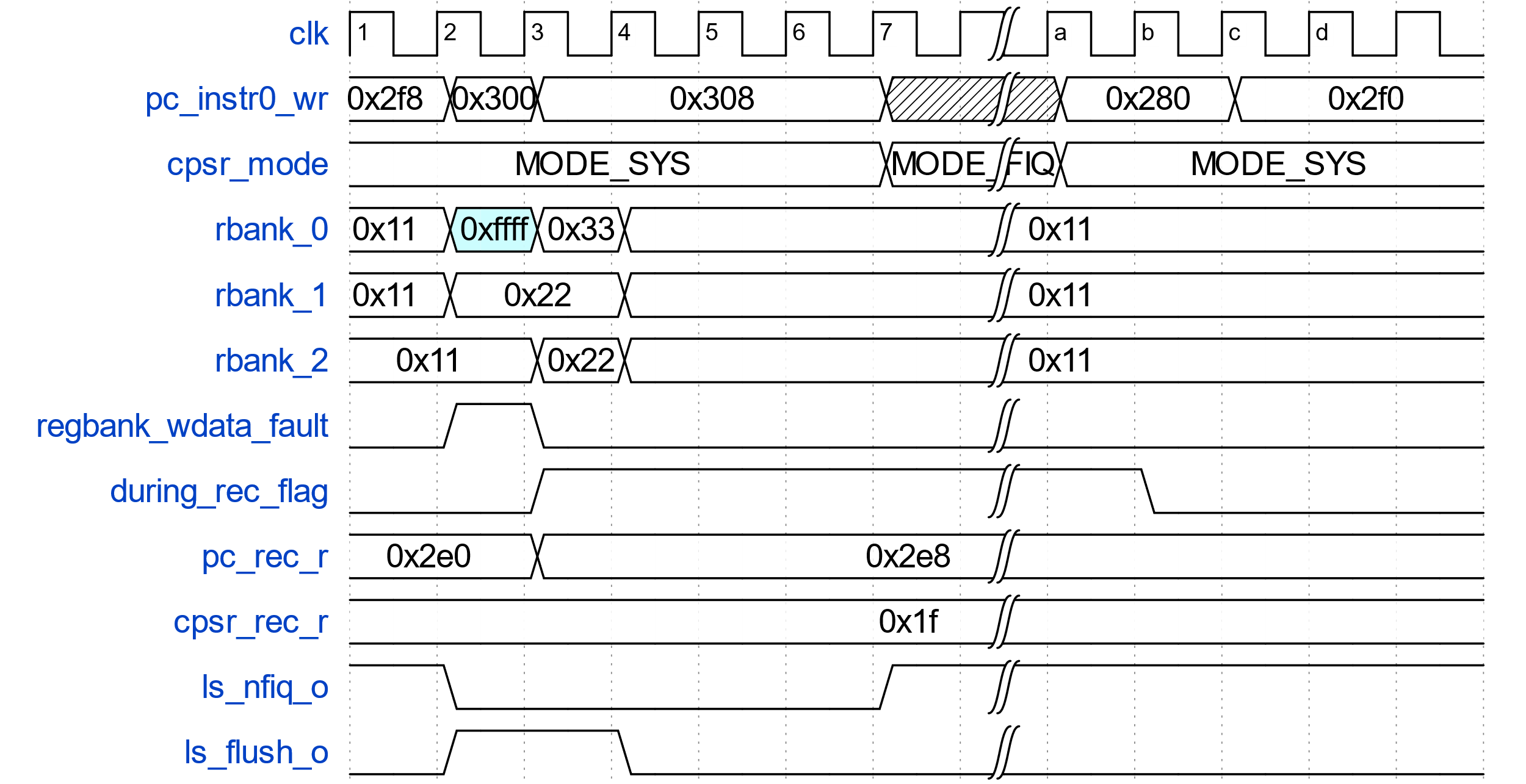


图2‑5 恢复模块波形示意

在2时刻rbank\_2应该是写入0x22，但因为软错误的发生，被错误地写入了0xffff，此时刻检测到了两个处理器写入寄存器堆的wdata数据不一致，因此表示写入寄存器堆错误的信号regbank\_wdata\_fault被拉高，同时会将中断信号ls\_nfiq\_o拉低，直至处理器进入中断程序，处理器进入中断模式时，cpsr\_mode会变为“MODE\_FIQ”。在2时刻时，同时会拉高流水线冲刷信号ls\_flush\_o，以维持几个周期，确保流水线仍在执行的指令不会造成影响。恢复模块同时会保存处理器当前的正确状态，即pc\_rec\_r、cpsr\_rec\_r等寄存器。pc\_rec\_r保存了发生错误的指令的PC值，即错误指令的地址，用于恢复程序完成后PC指向的地址。cpsr\_rec\_r保存了发生错误时处理器的程序状态寄存器，包括处理器在ARM指令集下还是Thumb指令集状态下、处理器的当前模式等各种信息。此外，还会将发生错误时寄存器堆的数据保存下来。在3时刻时，表示正在执行恢复逻辑的信号during\_rec\_flag被拉高，pc\_rec\_r和cpsr\_rec\_r不再改变。虽然在3时刻还会继续执行之前的指令，例如r0-r2还会被写入，但因为后面会将其恢复，因此这时刻无需在意寄存器堆。在4时刻时，寄存器堆被重新写入数据，恢复至发生错误前的状态，即r0-r2重新变为0x11。直到7时刻，处理器进入中断处理函数，在此会进行软错误的恢复。当恢复完成后，在a时刻处理器状态变为用户模式“MODE\_SYS”，在b时刻during\_rec\_flag信号被拉低，表示恢复工作已经完成。处理器在c时刻成功执行之前的指令，即pc\_instr0\_wr变为0x2f0，因为Cortex-R4是64位AXI总线，取值会提前进行，因此和pc\_rec\_r会相差一定数值。也就是说pc\_instr0\_wr为0x2f0但实际执行的是PC为0x2e8处的指令，此时处理器的状态在执行ARM指令集，当处理器运行在Thumb指令集模式下时，pc\_instr0\_wr和实际执行的PC值会相差0x4。因此，恢复模块中还会判断发生错误时处理器是在执行何种指令集，从而选择将PC值减去0x4或0x8，这个对于处理器在执行切换状态的指令时很重要，例如BLX指令，BLX会在跳转的同时切换处理器指令集状态。

### 软件部分恢复错误设计

当恢复模块的硬件部分检测到错误时会产生FIQ中断，从而进入中断处理函数，软件恢复错误设计的汇编代码如下所示：

FIQ\_Init\_Handler PROC

LDR r8, =Lockstep\_Ctrl\_Addr

LDR r9, [r8,#0]

CMP r9, #0

BEQ FIQ\_Normal\_Handler

B FIQ\_Recovery\_Handler

ENDP

FIQ\_Recovery\_Handler PROC

LDR lr, [r8, #4] ; pc

LDR r10, [r8, #0xc] ; spsr

MSR SPSR\_cxsf, r10

SRSFD sp!,#0x11 ; save lr & spsr

RFEFD sp! ; exit FIQ

ENDP

FIQ中断函数共有三个，FIQ\_Init\_Handler为进入中断后首先执行的函数，在此只进行判断是否发生错误，在硬件Lockstep\_ctrl模块里有一个APB从机，读取相应的数据，如果未发生中断则执行FIQ\_Normal\_Handler，即普通的FIQ中断处理函数。当读取的Lockstep\_Ctrl\_Addr处的数据为1时，则表示发生了错误，则会跳转至FIQ\_Recovery\_Handler进行错误恢复。

在FIQ\_Recovery\_Handler函数中，软件恢复逻辑中进行了四个步骤：

1. 恢复PC值。将要恢复的PC值写入FIQ模式的链接寄存器LR中；
2. 恢复CPSR寄存器。使用MSR指令将要恢复的程序状态寄存器CPSR写入FIQ模式的保存程序状态寄存器SPSR中；
3. 保存至FIQ模式的堆栈中。使用SRS指令将LR和SPSR保存到FIQ模式的堆栈；
4. 退出FIQ模式。使用RFE指令从FIQ模式的堆栈中返回。

## 自恢复三模冗余加固设计

流水线中会有一些寄存器会对处理器的状态立即产生影响，例如forceop\_v\_iss是控制PC强制操作的信号，如果此信号发生了错误，那么PC值会立刻发生变化，从而使得两个处理器状态无法同步，也无法去恢复错误。处理器内核中还存在保存或控制处理器状态的寄存器，包括CPSR等，当这些寄存器发生软错误，那么处理器可能会立刻发生状态转变，且很难再恢复。

针对影响处理器状态和立刻影响流水线的寄存器，采用更加稳固的容错技术，以确保此类寄存器不会影响处理器的正常运行。对于寄存器常见和广泛使用的加固方法是三模冗余(TMR)，通过将寄存器复制三份，然后再使用一个多数选择的表决器得到输出结果，如图2‑6所示。



图2‑6 三模冗余原理

这种常见的三模冗余加固方式可以有效地对单个寄存器出错进行屏蔽，通常来说，寄存器本身在电路中会被正常的逻辑更新数据，所以当寄存器错误发生时，后续寄存器会更新数据，从而恢复了冗余寄存器的数据。

但考虑下面一种情况，假若冗余的三个寄存器a、b、c均为单比特寄存器，同时有信号“en”控制三个冗余寄存器的更新，其中寄存器a相关代码示例如下：

|  |
| --- |
| always@(posedge clk or negedge rst\_n)begin  if(~rst\_n)  a <= 1’b0;  else if(en)  a <= in;  end |

但是使能信号“en”的更新频率是要根据整个电路设计功能而定的，假若三个寄存器a、b、c的更新频率并不是很高，当寄存器a发生了错误从而导致数据发生了翻转，此时三模冗余的表决器依然可以得到正确的结果，但假若在使能信号“en”更新之前寄存器b或c也发生了数据翻转，那么三模冗余的表决器得到的结果就是错误的。为了避免此类情况的发生，对三模冗余结构增加自恢复逻辑，使得发生错误的寄存器可以自我恢复，如图2‑7所示。



图2‑7 自恢复三模冗余结构

自恢复的三模冗余在一般结构的基础上增加了自恢复检测逻辑，当有其中一个寄存器因软错误发生而导致数据出错时，发生错误的寄存器的输入会根据检测信号选择正常输入还是表决器的输入，从而达到恢复错误寄存器的目的。

对Cortex-R4内核中所有影响处理器状态和立刻影响流水线的寄存器分为两种，不包含使能控制信号的寄存器，电路的每个时钟周期均会更新数值，无需自恢复，而对于包含使能控制信号的寄存器，采用带有自恢复功能的三模冗余结构。

# Cache SEU容错设计

## 加固仿真验证

# 容错效果评估与开销对比

## 流水线与寄存器堆容错效果分析

在完成流水线加固设计之后，又对加固后的处理器进行了加固仿真验证和功能验证，已确保加固后的处理器整体功能性没有发生改变，证明加固设计并未对处理器基本功能造成过多损失。

在确保加固设计的正确实现后，对加固效果进行评估和分析，分别从软错误率和资源利用等情况进行分析，以评估整体加固方案的加固效果。

### 故障注入结果及分析

使用仿真故障注入平台对加固后的流水线进行随机故障注入，故障注入的参数与1.4.2节使用的相同，使得到的数据更具对比价值。因为DPU模块是Cortex-R4处理器流水线的核心部分，因此重点关注此模块的恢复情况，如表4‑1所示。

表4‑1 DPU模块加固前后软错误率统计表

|  |  |  |
| --- | --- | --- |
| 模块名称 | 加固前平均软错误率 | 加固后平均软错误率 |
| u\_cpsr | 90.00% | 0.25% |
| u\_dpu\_regbank | 68.00% | 0.00% |
| u\_dpu\_br | 67.00% | 0.63% |
| u\_dpu\_ldst | 56.00% | 1.33% |
| u\_dpu\_de | 54.00% | 0.78% |
| u\_dpu\_cp | 53.33% | 0.00% |
| u\_dpu\_ctl | 44.89% | 0.94% |
| u\_dpu\_dp | 22.33% | 0.73% |
| u\_dpu\_dbg | 8.00% | 0.00% |
| u\_dpu\_etmif | 0.00% | 0.00% |

从表4‑1可以看出，大部分流水线的模块均可以完成恢复错误，说明加固效果符合预期。对于少数部分未恢复成功的，通过查找原因，大多都因为在错误恢复过程中又被强制注入了故障，造成恢复本身就发生了错误，导致恢复失败。其次还有少数恢复失败的原因是因为错误发生导致处理器进入了比FIQ模式特权更高的模式，致使恢复无法进行，这部分的失败需要格外注意。

从表4‑1中还可以看到，寄存器堆u\_dpu\_regbank在随机故障注入的实验中，均恢复成功，说明流水线Lockstep的加固方法对于寄存器堆有很好的加固效果。同时，在分析其他流水线寄存器恢复成功的例子时，也可以看到，如果流水线寄存器是影响计算的，包括运算的数据或者控制写入寄存器堆的使能信号，对于此类寄存器发生错误的情况，流水线Lockstep方法均可以做到完全恢复错误。

但同时也要看到，恢复也会造成有些无必要恢复的情况，因为有些寄存器在不加固情况下，错误率也不会是100%，这是因为可能故障发生的时候该寄存器并没有参与到流水线的进程中，等到参与流水线时寄存器已经被重新写入新的数据。但流水线Lockstep方法选取的对比信号都是流水线阶段上的重点信号，比如寄存器堆是处理器进行运算的核心部分，虽然寄存器堆发生错误时可能并没有使用该数据，但及时恢复寄存器堆对于整个处理器运行都是极有好处的。再比如预取指的指令，虽然可能会在跳转指令预测失败时被清空，但大多数情况下，预取值和预译码的指令都会被使用，比较这个地方也可以更及时地恢复错误。

### 资源利用与错误恢复时间分析

对加固前后的设计进行FPGA实现，通过对比FPGA的LUT逻辑资源和寄存器资源的消耗情况以评估加固前后设计的资源利用情况。分别对只进行流水线Lockstep加固方法和同时使用三模冗余加固的方法进行资源统计，如表4‑2所示。

表4‑2 加固前后资源统计表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 加固方法 | LUT as Logic | | Registers | |
| 资源数 | 与未加固相比比值 | 资源数 | 与未加固相比比值 |
| 未加固 | 29916 | 100.00% | 13041 | 100.00% |
| 流水线Lockstep | 53792 | 179.81% | 24242 | 185.89% |
| 流水线Lockstep+部分三模冗余 | 54619 | 182.57% | 24452 | 187.50% |

从表4‑2可以看出使用流水线Lockstep加固方案与未加固相比，用作逻辑的LUT资源增加了79.81%，寄存器使用资源增加了85.89%。当再加上部分三模冗余加固之后，用作逻辑的LUT资源消耗增加了82.57%，寄存器资源消耗增加了87.50%，可以看到，少数的三模冗余寄存器对资源的消耗并不是很多。流水线Lockstep加固方法资源消耗并没有增加一倍，这是因为综合是对于整个SoC而言的，Lockstep只是针对内核部分逻辑的，因此资源增加在百分之八十左右。对于内核模块也不是单纯的两倍的关系，内核模块u\_cortexr4noram共消耗26538个LUT单元、10313个寄存器，但Lockstep中的Slave处理器内核模块u\_cortexr4noram\_dual只消耗了23867个LUT单元、9207个寄存器，这是因为Slave处理器内核中有些信号并未使用，相关的逻辑也在综合中被优化，同时和内存模块交互数据是由Master处理器来完成的，因此Slave处理器内核中相关的逻辑也会被优化掉。

流水线Lockstep方法与全三模冗余方法的资源消耗对比如表4‑3所示。

表4‑3 不同加固方法资源对比表

|  |  |  |  |
| --- | --- | --- | --- |
| 加固方法 | LUT as Logic | Registers | |
| 流水线Lockstep | 53792 | 24242 |
| 流水线Lockstep+部分三模冗余 | 54619 | 24452 |
| 全三模冗余 | 78286 | 33698 |

从表4‑3中可以看到，相比于全三模冗余，无论是只使用流水线Lockstep加固方法还是结合部分三模冗余加固方法，资源的消耗都小于全三模冗余的加固方法。流水线Lockstep加固方法的LUT逻辑资源消耗大约是全三模冗余加固的68.71%，寄存器资源消耗大约是全三模冗余加固的71.94%。流水线Lockstep加上部分三模冗余的加固方法的LUT逻辑资源消耗大约是全三模冗余加固的69.77%，寄存器资源消耗大约是全三模冗余加固的72.56%。如果将LUT逻辑资源和寄存器资源加在一起当作整体资源，那么流水线Lockstep加固方法的整体资源消耗大约是全三模冗余加固的69.68%，流水线Lockstep加上部分三模冗余的加固方法的整体资源消耗大约是全三模冗余加固的70.61%。这说明两种加固方法在资源消耗上都比全三模冗余占有优势。

接着对加固后的Cortex-R4处理器错误恢复时间进行分析，以冒泡排序算法的完成周期数来估算不同加固方法的错误恢复时间。将加固后随机故障注入的运行周期数进行统计，得到所有仿真运行的平均周期数，之后与未进行故障注入的仿真周期数进行对比。同时，为了验证流水线Lockstep方法的优越性，对Cortex-R4进行了基于软件检查点的Lockstep设计，用于和流水线Lockstep方法进行对比，但基于检查点的Lockstep方法的检查点个数和分布是由人为确定的，这一点会造成不同方法的恢复时间和程序修改的难易度有很大差别，为了更加方便地对比结果，选择在程序开始和结束，以及冒泡排序的开始和结束插入检查点。

对于由于错误而导致处理器进入hang状态从而无法响应指令导致仿真超时的结果，仿真周期数以整个仿真的超时所运行的周期数计算，使用冒泡排序程序作为测试程序，对比结果如表4‑4所示。

表4‑4 不同加固方法运行冒泡排序程序的周期数统计表

|  |  |  |  |
| --- | --- | --- | --- |
| 运行条件 | 运行总次数 | 平均周期数 | 比值 |
| 未加固且未故障注入 | 1 | 26781.00 | 1 |
| 流水线Lockstep | 476 | 27795.18 | 1.04 |
| 流水线Lockstep+部分三模冗余 | 200 | 26922.50 | 1.005 |
| 基于检查点Lockstep | 100 | 36479.06 | 1.36 |

表4‑4中的比值一列为各方法的平均周期数与处理器正常运行程序的周期数的比值，可以量化不同加固方法的平均错误恢复时间。可以看到，流水线Lockstep方法在错误恢复时间上有所增加，而加上对重点寄存器三模冗余之后，错误恢复时间相对于整个冒泡排序程序就可以忽略不计，而基于检查点的Lockstep方法错误恢复时间就较为严重。对于全三模冗余的加固，可以简单认为与未发生故障时的运行完成周期数一致，因此流水线Lockstep的方法错误恢复时间上略低于全三模冗余加固方法。

## Cache容错效果分析