中国科学技术大学

本科毕业论文

基于龙芯架构 32 位精简版指令集的超标量处理器设计

作者姓名：	马子睿
学　　号：	PB20111623
专　　业：	计算机科学与技术
导师姓名：	卢建良 实验师，叶笑春 研究员
完成时间：	2024 年 5 月 30 日

摘　要

本论文主要探讨了超标量处理器设计以及其中预译码技术的创新及其在性能优化中的应用。首先，本文介绍了超标量处理器的总体设计，包括流水线结构、指令调度等关键部分，并对其中关键的技术点进行了较为详细的解读。其次，本文重点分析了跳转分支指令对于超深流水线的影响和分支预测的局限性。随后，针对现有问题，本文提出了一种新颖的预译码技术，旨在克服传统分支处理方案中存在的瓶颈，提高处理器的时序性能和执行效率。

新提出的预译码技术主要包括以下创新点：首先，本方法致力于在不引入过多前端额外流水级的情况下，以尽可能精简的时序完成分支指令的修正。其次，本方法不局限于预译码单元的优化，而是充分结合了前端多级流水线的特点，实现了时序优化的协同工作。最后，本文通过深入研究商用处理器的设计理念和实现细节，我们将其中适用于本文预译码技术的部分移植到了本文的设计中，并对其进行了必要的修改和优化，以适应超标量处理器的特定架构和需求。

最后，本文介绍了自行开发的仿真环境，并在其中对提出的预译码技术进行了验证和评估。实验结果表明，新提出的预译码技术相比传统方案在吞吐率和性能效率上均取得了显著提升，验证了其在超标量处理器中的有效性和可行性，同时也验证了自行开发的仿真环境对超标量处理器调试具有一定的助助力。因此，本文的研究为超标量处理器的性能优化提供了重要的理论基础和实践指导，具有一定的学术和应用价值。

关键词：超标量；预译码；性能探究

ABSTRACT

This paper primarily explores the design of superscalar processors and the innovation of predecoding technology within them, along with its application in performance optimization. Firstly, an overview of the overall design of superscalar processors is provided, including key components such as pipeline structure and instruction scheduling, with detailed explanations of crucial technical aspects. Secondly, the paper analyzes the impact of branch instructions on deeply pipelined processors and the limitations of branch prediction. Subsequently, addressing these existing issues, a novel predecoding technology is proposed aimed at overcoming bottlenecks in traditional branch handling schemes, enhancing processor timing performance, and execution efficiency.

The newly proposed predecoding technology primarily introduces the following innovations: Firstly, the method endeavors to correct branch instructions with minimal additional frontend pipeline stages, aiming for a streamlined timing approach. Secondly, the approach does not limit itself to predecoding unit optimization but fully integrates with the characteristics of the frontend multistage pipeline, achieving collaborative timing optimization. Lastly, by delving into the design principles and implementation details of commercial processors, applicable portions for predecoding technology in this paper are transplanted into its design, subsequently modified and optimized to suit the specific architecture and requirements of superscalar processors.

Finally, the paper introduces a self-developed simulation environment, wherein the proposed predecoding technology is validated and evaluated. Experimental results demonstrate significant improvements in throughput and performance efficiency compared to traditional approaches, validating its effectiveness and feasibility in superscalar processors. Additionally, the developed simulation environment's assistance in debugging superscalar processors is affirmed. Hence, this research provides a significant theoretical foundation and practical guidance for performance optimization of superscalar processors, offering academic and practical value.

Key Words: superscalar; predecoding; performance exploration

目 录

第一章 绪论 …… 4
第一节 研究背景 …… 4
第二节 超标量处理器概述 …… 4
第三节 超标量处理器的优势 …… 5
第四节 超标量处理器研究现状 …… 5
第五节 本文主要内容 …… 6

第二章 LA32R 超标量处理器设计 …… 8
第一节 流水线概览 …… 8
第二节 前端基础流水线 …… 9
一、预取指阶段 …… 9
二、取指阶段 …… 9
三、预译码阶段 …… 9
四、译码阶段 …… 9
五、重命名阶段 …… 10
第三节 后端基础流水线 …… 11
一、后端功能模块概述 …… 11
二、发射与唤醒 …… 11
三、读寄存器堆 …… 12
四、算数执行 …… 12
五、乘法与除法 …… 12
六、存储器访问 …… 13
七、写回阶段 …… 14
八、提交阶段 …… 14
第四节 特权系统的支持 …… 15
一、控制状态寄存器 …… 15
二、异常处理 …… 15
三、中断处理 …… 16
四、访存单元控制指令 …… 16

第三章　仿真开发环境 LSIM 设计 …… 17
第一节　开发背景 …… 17
一、仿真环境对于处理器开发的意义 …… 17
二、处理器仿真环境现状 …… 17
第二节　基于 Verilator 的 LSIM 仿真框架概览 …… 18
一、Verilator 简介 …… 18
二、仿真框架概览 …… 18
第三节　处理器监管分析技术 …… 19
一、Difftest …… 19
二、性能统计 …… 21

第四章　预译码系统的创新与优化 …… 22
第一节　预译码系统概述 …… 22
一、预译码技术的主要思想 …… 22
二、预译码技术面临的问题 …… 22
第二节　跳转指令码立即数替换 …… 22
一、跳转地址计算的流水级分配问题 …… 22
二、指令码替换思想在实现层次的问题 …… 23
三、ICache 适配设计 …… 24
四、预译码器的设计 …… 24
五、指令码恢复 …… 25

第五章　基于 LSIM 的仿真功能与性能探究 …… 26
第一节　仿真 Soc 的搭建 …… 26
第二节　测试程序构建 …… 27
一、交叉编译工具链 …… 27
二、应用程序基础设施搭建 …… 27
三、测试程序通用编译流程 …… 27
第三节　处理器功能仿真验证 …… 28
一、Chiplab 测试程序 …… 28
二、南京大学 ICS 课程测试程序 …… 28

第四节　处理器性能探究 …… 29
一、乱序多发射的性能测试 …… 30
二、预译码的性能测试 …… 31
第五节　本章小结 …… 32

第六章　片上系统时序优化探索 …… 33
第一节　SOC 的搭建 …… 33
第二节　基于 Vivado 的时序优化探索 …… 34
一、交叠技术 …… 34
二、跳转指令立即数替换技术 …… 36
第三节　电路综合实现结果 …… 36
一、SOC 资源综合实现结果 …… 36
二、SOC 时序综合实现结果 …… 37
第四节　本章小结 …… 38

第七章　成果总结与展望 …… 39
第一节　成果总结 …… 39
第二节　应用与推广 …… 39
一、开发者社区开源工作 …… 39
二、中国科大《计算机系统综合实验》课程 …… 40
三、中国科大《计算机组成原理》课程 …… 40
四、“一生一芯”学员“龙芯杯”参赛教程 …… 40
第三节　未来展望 …… 40

参考文献 …… 42
致谢 …… 44

第一章　绪论

第一节　研究背景

随着云计算、人工智能和物联网等技术的迅猛发展，当今社会的个体与组织每天都会产生巨大的数据。据统计，到 2025 年全球数据总量将达到 175 ZB，而如何使用处理器对这些数据进行高效、精确的处理与分析就成为了现代计算科学的重要研究课题。自 20 世纪 50 年代诞生以来，传统的处理器设计虽然能自动化地正确处理数据，但随着单个处理器核心的性能逐渐达到物理限制，传统处理器设计也逐渐到达其性能上限。因此，研究人员开始探索如何通过增加并行执行的指令数量来提高处理器性能，而超标量处理器因其能够同时执行多条指令而备受关注。

传统的超标量处理器设计面临着诸多挑战，包括指令调度、资源冲突、数据依赖性等问题。为了克服这些挑战，研究人员提出了许多创新的设计技术，如动态调度、推测执行、分支预测、乱序执行等。这些技术的引入使得超标量处理器能够更好地挖掘指令间潜在的并行性能，从而使得处理器的处理能力产生了巨大的突破。时至今日，超标量设计依然是通用处理器提高性能的重要手段，以之为核心的设计方法日新月异，使之成为了计算机系统结构研究的一个热点话题。

第二节　超标量处理器概述

超标量处理器是一种高性能计算机处理器架构,旨在实现指令级并行性(ILP)的最大化。其设计理念是通过同时执行多条指令来提高处理器的性能，而不仅仅是增加时钟频率或增加处理器核心数量。相比传统的顺序执行处理器，超标量处理器具有更高的并行执行能力，可以在硬件层次动态调度指令，检测其相关性，并行执行多条指令，以充分利用硬件资源[1]。

超标量处理器的主要组件包括:

1. **取指单元**：负责从内存中读取指令，并将其送入指令解码单元。

2. **指令解码单元**：将取指单元读取到的指令进行解码，确定其操作类型和操作数。

3. **执行单元**：由多个执行管道组成，每个管道可以独立执行不同的指令操作，如算术运算、逻辑运算和内存访问等。

4. **寄存器重命名单元**：解决由于指令并行执行引起的寄存器冲突问题，确保指令在执行过程中使用正确的寄存器值。

5. **提交单元**：负责确保指令按程序顺序提交结果，从而维护程序的正确性。

超标量处理器通过这些组件的协同工作，可以显著提高计算机系统的处理性能。然而，其设计和实现也更为复杂，需要解决许多技术挑战，如数据依赖、控制依赖和资源冲突等问题。尽管如此，超标量技术在现代高性能处理器中依然得到了广泛应用，推动了计算机体系结构的发展。

第三节　超标量处理器的优势

超标量处理器相对于传统的处理器架构具有许多优势[2]，主要包括：

1. **更高的性能和效率**：超标量处理器能够同时执行多条指令，充分利用处理器的硬件资源，从而提高了处理器的性能和效率。通过动态指令调度、乱序执行等技术，超标量处理器可以最大化地利用指令级并行性，实现更高的指令吞吐量。

2. **更好的资源利用**：超标量处理器具有多个功能单元，能够并行执行不同类型的指令，充分利用处理器的硬件资源，提高了资源利用率。这使得超标量处理器能够更有效地处理各种计算和数据处理任务。

3. **更灵活的指令调度**：超标量处理器采用动态指令调度技术，可以根据当前的指令依赖关系和可用资源来动态调整指令的执行顺序，从而最大化地利用指令级并行性。这使得超标量处理器能够更灵活地适应不同类型的计算负载。

4. **更高的吞吐量**：由于超标量处理器能够同时执行多条指令，因此具有更高的指令吞吐量。这使得超标量处理器能够更快地完成计算任务，提高了系统的整体性能。

5. **更好的可扩展性**：超标量处理器的设计可以很容易地进行扩展，通过增加功能单元或者优化指令调度算法等方式来提高性能。这使得超标量处理器能够适应不断增长的计算需求，具有较好的可扩展性和适应性。

第四节　超标量处理器研究现状

超标量处理器技术自从 20 世纪末以来一直是计算机架构领域的重要发展方向。这种处理器通过并行地执行多条指令来提高性能，能够在一个时钟周期内启

动多个指令序列，显著提升了执行效率。

目前，超标量技术的研究与发展主要集中在几个关键领域：

1. **指令级并行度（ILP）的提升**：研究人员持续探索如何提高处理器的指令级并行度，这包括改善指令调度算法，增加执行单元，以及优化指令流水线。通过这些方法，处理器可以更有效地处理指令依赖和执行指令重排序。

2. **分支预测技术**：分支预测是超标量处理器性能提升中的关键技术之一。近年来，分支预测器的设计已经变得越来越精细和复杂，包括使用机器学习技术来预测代码中的分支决策，从而减少由于分支误预测引起的性能损失。

3. **能耗管理**：随着处理器核心数量的增加，能耗管理成为超标量处理器设计的一个重要考虑因素。研究者们在设计时不仅需要关注性能提升，还需考虑到能效比。这包括开发动态电压调节技术、时钟门控技术以及更有效的散热解决方案[3]。

4. **多核与众核架构**：为了进一步提升计算能力，现代超标量处理器越来越多地采用多核和众核架构。这需要在核心间实现有效的通信和同步机制，同时优化软件来充分利用硬件资源[4]。

5. **处理器微架构的创新**：例如，引入了模块化的设计、异构计算元素（集成 CPU 与 GPU 等）和专用加速器[5]（如 AI 加速器），这些都是近年来超标量处理器领域的研究热点。

随着技术的发展，超标量处理器将继续适应新的计算需求，例如高性能计算、大数据处理和人工智能等领域的应用。处理器设计师需在提升计算能力和优化能效之间找到平衡，同时还需不断创新以适应不断变化的技术环境。

第五节　本文主要内容

本文旨在介绍笔者开发的一款具有一定创新性的超标量处理器，该处理器在设计中融合了多项优化和创新点，并具有独特的处理器调试环境特色。本文将着重介绍该超标量处理器的创新和优化点，以及针对调试环境所做的改进。

首先，本文将详细介绍该超标量处理器的创新点。笔者充分考虑了指令级并行性的最大化，采用了创新的技术和方法来提高处理器的性能和效率。其中包括动态指令调度算法的优化，乱序执行技术的改进，以及多功能单元的设计等方面。

其次，本文将重点介绍该超标量处理器的优化点。在设计过程中，笔者针对

处理器的各个方面进行了深入的优化，包括逻辑设计和微架构设计等。通过优化处理器的硬件资源利用和指令调度算法，使处理器整体性能获得了较大的提升。

最后，本文将介绍该超标量处理器的处理器调试环境特色。为了提高处理器的调试效率和可靠性，笔者设计了一套独特的处理器调试环境，使得开发人员能够更加方便地对处理器进行调试和性能分析，

第二章　LA32R 超标量处理器设计

第一节　流水线概览

本项目的超标量处理器基于 Tomasulo 旁路算法[6] 156-162，采用 11-12 级流水、四发射、乱序执行、顺序提交的方式进行构建。该处理器支持龙芯架构 32 位精简版（LA32R）指令集除 IBAR 和 DBAR 外的全部指令和 16 种处理器异常[7] 61-62。

按照指令流动的方式，该处理器的流水级可划分为：

- 前端：预取指、取指、预译码、译码、重命名
- 后端：发射、读寄存器堆、执行（地址转换、访存）、写回、提交

在前端中，指令保持程序中原有顺序流动，并在重命名阶段被打乱送入对应的功能单元发射队列。后端使用重排序缓存维护指令原有顺序，并通过乱序多发射的思想尽可能挖掘指令间的并行性。

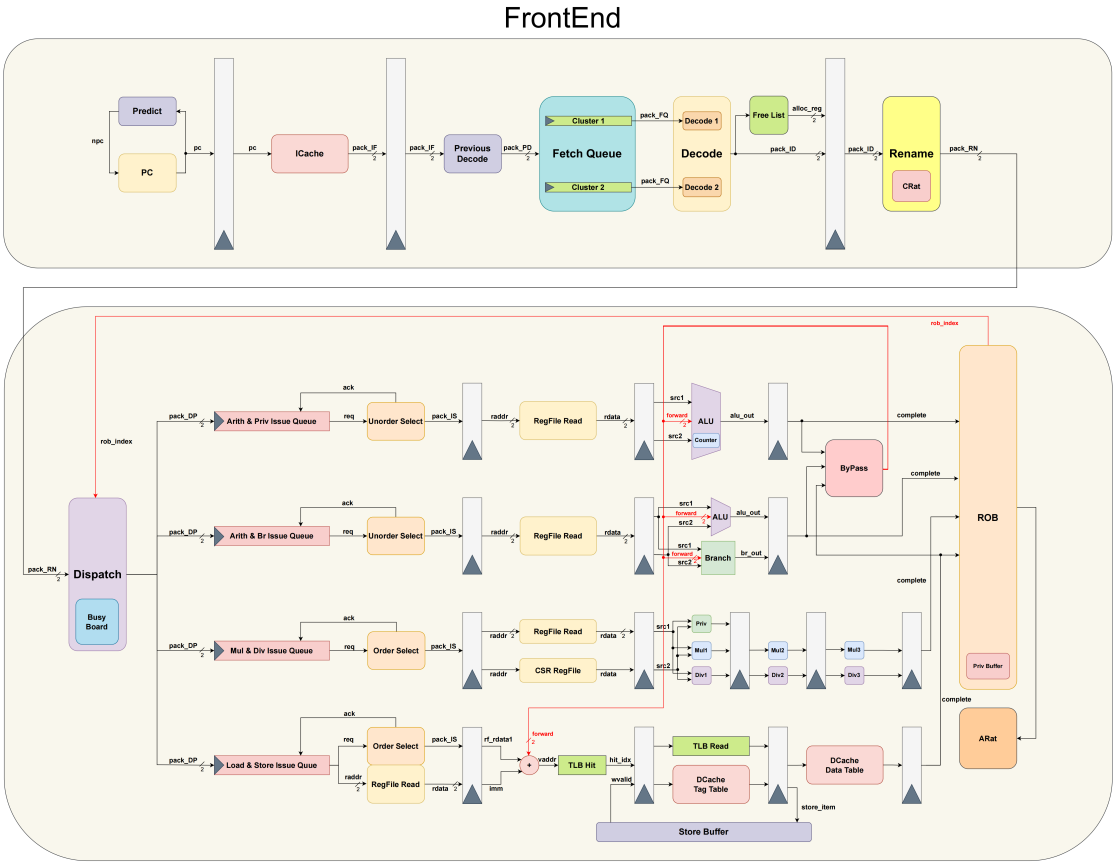

图 2.1　流水线架构简图

第二节　前端基础流水线

该处理器采取双取指的实现方式，前端的每一个流水级在每个周期内至多运输两条指令。下面对各个流水级进行介绍。

一、预取指阶段

预取指阶段由程序计数器（PC）、取指地址转换单元（Inst-TLB）和分支预测器构成。该阶段产生两个顺序指令地址送入流水线，并使用这两个地址对分支指令的方向和目标地址进行预测。分支预测是基于局部历史的 2bit 感知机预测，并配备了返回地址栈（RAS），用以对间接跳转进行预测。

若 PC 当前对齐 4 字节但不对齐 8 字节，虽然在大多数情况下指令高速缓存仍然可以给出两条指令，但为减少分支预测器的逻辑延迟和复杂度，在当前的设计中，这种情况下处理器会认为仅有一条指令是有效的。

二、取指阶段

取指阶段主要由指令高速缓存（ICache）构成。由于地址转换已经在预取指阶段完成，故 ICache 的访问是基于物理地址的。若 ICache 命中，则会给出地址相邻的两条指令，否则流水线的前两个阶段将陷入阻塞，直至缺失的缓存块从内存中读出。

三、预译码阶段

为了应对分支预测器的容量和替换问题，在指令进入流水线后，预译码阶段可以进行针对跳转的初步译码和进一步预测，从而对分支预测进行一定程度的补救。经过预译码的指令会被送入取指队列，该队列由交叠[8] 133-134 实现的伪双端口 FIFO 实现，每个周期可以送入 0-2 条指令，并发射出 0 或 2 条指令。

四、译码阶段

译码阶段由译码器和空闲物理寄存器列表构成。其中，译码器借助 Chisel 中的 BitPat 自动生成组合逻辑，负责对从取指队列中发射出的至多 2 条指令进行译码，获得其寄存器编号、ALU 操作数、访存类型、跳转类型等信号。空闲物理寄存器列表为需要写寄存器堆的指令分配空闲物理寄存器，并从提交阶段获取待废除的物理寄存器。若空闲物理寄存器列表为空，则流水线需要阻塞前 3 个阶

段，直至后端有待废除的物理寄存器提交至此。

五、重命名阶段

1. 重命名系统

在重命名系统中，寄存器重命名采用了统一物理寄存器堆重命名[8] 201-203 的方式实现。其中，重命名映射表利用逻辑寄存器的编号和每一个存储单元分别比较，获取到对应的物理寄存器编号及其当前是否滞留在发射队列的信息。同时，对于需要目的寄存器的指令，重命名系统还会将分配的物理寄存器的对应表项的有效位置位，并标记其为“滞留于发射队列”的状态。由于每个周期至多有两条指令需要重命名，因此可能出现写后读 (RAW) 和写后写 (WAW) 相关，重命名系统需要根据较旧指令的重命名情况，对较新指令读出的物理源寄存器编号进行修缮，并在出现 WAW 相关时将较新的目的逻辑寄存器写入映射表。

每条指令不仅需要读出两个源寄存器对应的物理寄存器编号，还需要读出目的寄存器曾经对应的物理寄存器编号，这个编号作为待废除的物理寄存器编号，将会在提交阶段提交给空闲物理寄存器列表，从而实现回收。

由于重命名阶段处在推断状态，因此在提交阶段，处理器维护了另一个正确的映射有效位表，当分支预测失败时，重命名阶段的有效位表会被设置为提交阶段的有效位表，从而实现了映射关系的恢复。

2. 分派器

分派器是连接处理器前端和后端的重要元件。这个单元会根据译码阶段给出的发射队列选择信号，为每一个发射队列提供如下信息：

- 当前周期需要进入该发射队列的指令掩码
- 当前周期需要进入该发射队列的指令处于这该指令组的哪个位置

对于功能相同的发射队列，分派器会比较当前这些发射队列中存在的指令总数，将指令送入总数较少的队列，以减少前端流水线的停顿。

3. 指令顺序的维护

在这个阶段，每一条指令会从重排序缓存（ROB）中获取 1 个顺序编号。这个编号将会伴随这条指令进入后端，引导其将完成信息写入正确的 ROB 表项，直至其完成执行并顺序提交。若当前周期重排序缓存已满，则流水线需要阻塞前 4 个阶段，直至有指令提交。

第三节　后端基础流水线

一、后端功能模块概述

处理器的后端负责对指令规定的功能进行执行。在本项目的乱序多发射后端中，共有 4 个独立的功能单元，分别是：

- 算数逻辑运算单元（第一运算单元）：处理算数逻辑指令和计数器读取指令
- 算术逻辑 & 分支跳转单元（第二运算单元）：处理算数逻辑指令和分支跳转指令
- 乘除 & 特权处理单元：处理乘除法和特权指令
- 访存处理单元：处理器加载、存储和同步指令

二、发射与唤醒

处理器的发射队列采用压缩队列实现，其中第一、第二运算单元的发射队列为乱序发射队列，而为了保证特权指令的顺序和强序非缓存[7]7 内存访问的实现，乘除和访存单元的发射队列为顺序发射队列。

1. 发射

发射操作由发射队列和选择模块协同完成。这两个模块间使用 Request-Acknowledge 协议进行通讯。发射队列的每个表项包括：

- 指令包：包含当前指令在后端所需的全部信号
- 寄存器唤醒标记：表示当前寄存器是否被前序指令唤醒
- 寄存器由加载指令唤醒标记：表示当前寄存器是否由加载指令唤醒

顺序发射队列仅会在寄存器已经就绪时，使最旧的指令向选择模块发送发射请求。若后续流水线无阻塞，则选择模块会响应这个请求，并将指令发射到后续流水线中。而在乱序发射队列中，每一个源操作数准备好的有效表项都会向选择模块发送发射请求，选择模块会使用优先编码器，响应其中最旧指令的请求——这是因为最旧的指令往往处于相关链顶端，发射后可以唤醒更多指令，尽可能降低处理器内部的指令相关性。

2. 唤醒

发射队列中的每一个表项都会将自身的 2 个源寄存器编号与唤醒总线上的 4 个寄存器编号进行比较，若相等则将唤醒标记置位。在处理器中，有两处唤醒操作较为重要：

（1）推测唤醒

由于加载指令往往处于相关链的顶端，因此访存处理单元采取了推测唤醒的方式以加快其唤醒效率。在访存的第一个周期，该指令便开始唤醒其他指令。若下一个周期发现该指令发生了高速缓存缺失，则被当前指令唤醒的所有指令应当在缺失处理完成前滞留在发射队列中，否则正常发射即可。

（2）相关指令连续发射

算数逻辑指令占据了程序绝大部分比重，因此这些指令若能够在相关时连续发射，就能够对处理器性能有较大提升，故该处理器中算术逻辑队列的发射与唤醒在同一个周期内完成。

三、读寄存器堆

为了适应四个发射队列的独立读取和写入，寄存器堆共有 8 个读端口和 4 个写端口，每一个读端口都配备了写优先。由于重命名系统的存在，后端不可能存在两条目的寄存器相同的指令，因此写优先至多命中一个写数据，故流水线可以将四个写入数据前递到每一个读端口上，并使用独热多选器对写优先数据进行选择。

四、算数执行

该处理器共有两个 ALU，支持 LA32R 指令集规定的全部 11 种运算。其中第一运算单元还承担了读取恒定频率计数器值[7]22 的功能，第二运算单元的 ALU 还承担了为 BL 和 JIRL 指令计算 PC+4 的功能。

五、乘法与除法

1. 华莱士树流水乘法

处理器的乘法单元采用了 3 级流水的华莱士树流水乘法，使用 33 位扩展乘法[9]144 来实现 LA32R 指令集中全部乘法指令。具体如下：

- 第一级：2 位 Booth 编码，使用乘数将被乘数编码，将部分和的数量由 32 削减至 16；
- 第二级：8 层 66 位保留进位加法器树，每层将部分和削减三分之一，直至剩余两个部分和；
- 第三级：64 位全加器，将最后两个部分和相加，得到最后结果。

2. 对数加速移位除法

除法和取余运算本质相同，都类似于竖式除法的方式进行多周期移位相减。但是当被除数较小时，竖式除法是没有必要使用 32 个周期完成运算的。因此我们可以使用如下方法对除法进行速度优化：

- 第一个周期，取被除数的对数，即寻找最高位的 1 处于第几位，记录在计数器中，并将被除数左移至最高非零位处于第 31 位；
- 除法开始后，只需要运行被除数的对数个周期，即可完成运算。

六、存储器访问

本项目基于存储器访问做出了较多优化。考虑到时序问题，访存流水线的设计与其余流水线有较大差别，也引入了一些特殊的部件。下面一一进行介绍：

1. 发射队列提前读取寄存器堆

由于访存队列是压缩的且是顺序发送的，因此寄存器堆的读地址只会来自发射队列的第 0 个表项，是相对固定的，故发射队列和寄存器堆之间没有段间寄存器堆，指令发射与读取寄存器堆是同步进行的，这样可以使得访存流水线的流水级数得到压缩。

2. 地址计算与 TLB 命中判断

TLB 全相连结构的时间延迟，使其前后都不可有任何其他的组合逻辑。若存在组合逻辑，则需要对 TLB 读取进行切割，因此和取指不同，访存的 TLB 采用了两级切割。在第一个流水级中访存单元计算出虚拟地址，并和 TLB 的每一个表项进行命中判断，将命中信息独热码送入段间寄存器。在这个流水级，虚拟地址也被送入数据高速缓存（DCache），根据虚拟 Index 物理 Tag 原理[9]227，这个周期可以使用虚拟地址中的 Index 部分开始查找 Tag 和 Data 表所在的块式存储器（BRAM）。

3. TLB 读取与标签比较

段间寄存器中的 TLB 命中独热码可以作为地址，对 TLB 表项进行索引。索引出的结果将送入数据高速缓存，和读出的 Tag 进行比较，并将命中信息送入下一段间寄存器。

此阶段的访存请求都是推测的，写操作不能在此时产生效果，故这个阶段需要将写请求写入一个写缓冲（Store Buffer）中，当存储指令被提交时，该指令才会对高速缓存发起写操作。在执行加载指令时，读请求也要同步查找写缓冲，查找方式如下：

- 对当前请求地址对应的 4 个字节分别进行全相连查找，将命中信息锁存一个周期；
- 利用命中信息读取对应的写缓冲数据，当有多项命中时，根据尾指针的位置找到最近写入的数据。

4. 高速缓存缺失判断与数据重组

为优化时序，高速缓存在获取到命中信息后，并不会立刻使用状态机进行缺失判断，而是锁存一个周期后，送入高速缓存状态机。

为了使得 LA32R 中的强序非缓存访问能够按照顺序确定地执行，所有的非可缓存访问都会滞留在这个阶段，直至这个访问被确认为是最旧的一条指令，访存流水线才会继续流动。

对于加载指令，其数据可能来自于高速缓存，也可能来自于写缓冲，因此在这个阶段需要根据写缓冲的命中情况，对数据进行拼接。

七、写回阶段

写回阶段将会对物理寄存器堆发起写操作，并标记重排序缓存对应表项的完成状态。为了加速运算指令的执行，在这个流水级内，第一、二运算和访存流水线中架设了旁路网络：第一、第二运算流水线可以相互前递数据，使得二者间的相关指令可以连续发射；同时，第一、第二运算和访存流水线可以将数据前递回访存流水线的地址计算阶段，进一步压缩了访存的流水线级数。

八、提交阶段

处理器的提交阶段负责维护处理器状态，主要包括重排序缓存（ROB）和状态恢复元件（ARAT）。

ROB 采用伪双端口交叠 FIFO 的形式实现，每个周期最多可以将 ROB 头部的两条指令提交。在基础架构中，为了减少分支预测器的写入端口数，头部两条指令提交的前提是：执行完成且靠前的指令不为跳转指令。如果最头部的指令是跳转指令，那么当前周期只能提交一条指令

ARAT 中维护了寄存器映射表的有效位，可在分支预测失败时恢复到重命名单元中。同时，ARAT 还维护了分支预测器返回地址栈的栈顶指针，在分支预测失败时，也会恢复到分支预测器中，从而恢复预测器中程序执行的栈结构记录。

第四节　特权系统的支持

本项目处理器支持龙芯架构 32 位精简版指令集所规定的全部特权架构，本节将会从特权架构所必需的几个部分进行介绍。

一、控制状态寄存器

1. CSR 写指令对指令流的影响

处理器可以对控制状态寄存器（CSR）进行读写。值得注意的是，除了读写，在触发异常、触发中断时 CSR 的值也会发生改变，这意味着控制状态寄存器的值变化可能会影响指令流（例如在写入 CRMD 寄存器后立刻执行特权不合法的特权指令）。考虑到 CSR 读写指令执行频率并不高，因此对 CSR 写指令采用执行后冲刷流水线的思路来实现。

2. CSR 写指令的写入时机

对 CSR 进行重命名的代价是比较大的，在没有重命名的情况下，写回段不能直接对 CSR 进行写入，而是应当在该指令提交后再进行写入。同时，TLB 查找需要一些 CSR 的信息，为优化地址转换的时序，当 CSR 被写入时，处理器会多阻塞一个周期，用以将 CSR 信息锁存一个周期后送入 TLB。

3. ROB 对 CSR 写指令的支持

CSR 写指令由乘除模块进行处理，这个模块是顺序发射的，因此处理器在 ROB 中维护了一个特权缓冲区，CSR 写地址、写数据等信息都会在执行阶段写入这个缓冲区，提交时由这个缓存对 CSR 发起写操作。由于特权写指令一定会冲刷全部流水线，故在冲刷流水线前，缓冲区仅仅会记录最旧的 CSR 写请求，后续写请求将会被缓冲区无视。

二、异常处理

LA32R 指令集规定了若干异常，根据触发地点不同可以分为：

- 前端：PIF，PPI，ADEF，SYS，BRK，INE，IPE，TLBR
- 后端：PIL，PIS，PME，PPI，ADEM，ALE，TLBR

可以注意到，后端的异常只会在访存流水线中产生，因此可以采用如下方法进行处理：

- 在译码阶段之前产生的所有异常，都会将产生异常的指令替换为一条 mul $r0, $r0, $r0 指令，并在重命名阶段将异常写入重排序缓存；

- 在译码阶段产生的异常，直接将其目标队列序号设置为乘除法队列，并将运算设置为乘法，并在重命名阶段将异常写入重排序缓存；
- 由于只有加载和存储指令可能在后端触发异常，而能够在后端触发异常的加载和存储指令在前端一定没有触发异常（否则这条指令将被被替换为 mul $r0, $r0, $r0 并送入乘除法队列）。因此，访存流水线会在写回段将后端产生的异常写入重排序缓存。并将这条指令附带的全部写入控制信号设置为无效。

三、中断处理

LA32R 规定，ESTAT 寄存器在每个周期都会对中断源进行采样[7]53。一旦其侦测到存在异常信号，就会将中断向量立刻保存到 ROB 的缓冲区里，当下一条指令提交时，这条指令会被打上中断标记，并触发这个中断，将指令流重定向到中断向量表起始地址。

四、访存单元控制指令

1. TLB 读写与查找
   - TLBRD：执行阶段读出对应的 TLB 表项，写入 ROB 的缓冲区中，提交时同步到 CSR 中。
   - TLBWR/TLBFILL：提交阶段将 CSR 中的值写入 TLB 中，TLBFILL 使用了计数器的低 4 位进行随机化处理。
   - TLBSRCH：执行阶段查找 TLB，找到命中的表项索引，写入 ROB 的缓冲区中，提交时同步到 CSR 中。
   - INVTLB：执行阶段将操作数写入 ROB 的缓冲区中，提交时对 TLB 进行对应的无效化处理。
2. Cache 操作

CACOP 指令涉及地址转换，因此需要送入访存流水线。CACOP 必须保证：在执行之前所有的 Store 操作都已经完成，因此，CACOP 会滞留在发射队列中，直到这条指令被确认为最旧的指令才会发射。

对于 DCache 而言，CACOP 可以视为一次普通的访存，而对于 ICache 而言，这条指令打断了正常取指的节奏，因此当 CACOP 提交后，流水线依然需要冲刷一次来恢复到正常的取指节奏。

第三章　仿真开发环境 LSIM 设计

第一节　开发背景

一、仿真环境对于处理器开发的意义

处理器的开发对时效性、严谨性都有着较高的要求，而对处理器的调试工作也与软件程序开发有很大的区别。由于处理器开发规模大、综合性强，从多如牛毛的信号中提取、分析出关键问题，并用尽可能小的成本对其进行修改，成为了影响调试效率的关键因素。一个较为完善的仿真环境，可以从以下几个方面助力于处理器的开发：

- 降低成本和风险：传统的硬件原型开发需要大量的时间和资源，并且一旦发现问题，修复起来成本高昂。仿真环境能够在软件层面模拟硬件行为，因此可以大幅降低硬件原型开发的成本和风险。
- 加速开发周期：仿真环境允许工程师在软件模拟中快速测试各种设计方案，包括新的架构、指令集、优化算法等[10]。这样可以大幅缩短从概念到产品的时间，使得处理器能够更快地投入市场。
- 简化调试和优化流程：仿真环境提供了丰富的调试工具和性能分析功能，使得工程师可以快速定位和解决设计中的问题，并对性能进行精细调优。相比于在实际硬件上进行调试和优化，仿真环境更加灵活和高效。

二、处理器仿真环境现状

目前，各大处理器开发企业、科研院所都拥有自己的完备的处理器调试环境，但对于开发处理器的个体或小团体而言，这些调试环境是无法获取到的。因此，我们只能从开发者社区等来源获取到基于高校基础教学的仿真环境源代码。但对于超标量处理器的设计而言，需要监测、分析的信号和逻辑有显著的个体化趋势，目前几乎无法获取到能普适各种设计的仿真环境；同时，超标量处理器的开发重在性能分析，如何通过处理器信号使软件自动生成具有针对性的运行报告也成为了一个重要问题[11]。因此，笔者希望能够开发一套适配于超标量处理器的仿真环境，在兼顾高性能、高稳定性的同时，能够针对性地进行调试和性能统计，并尽可能简化架构，便于开发者进行修改。

第二节　基于 Verilator 的 LSIM 仿真框架概览

一、Verilator 简介

Verilator 是一种开源的硬件描述语言 (HDL) 仿真工具，主要用于对 Verilog 和 SystemVerilog 代码进行快速仿真[12]。它是一个基于 C++ 的仿真器，能够将硬件描述语言代码转换成高速、高性能的 C++ 模拟器，从而实现对硬件设计的快速验证和调试。

相较于继承的开发环境，Verilator 不仅仅允许我们单纯运行仿真，还可以在仿真的每个周期内进行额外的计算，从而分析处理器当前状态是否符合我们的预期。

二、仿真框架概览

LSIM 基于 Verilator 进行开发，程序结构如下：

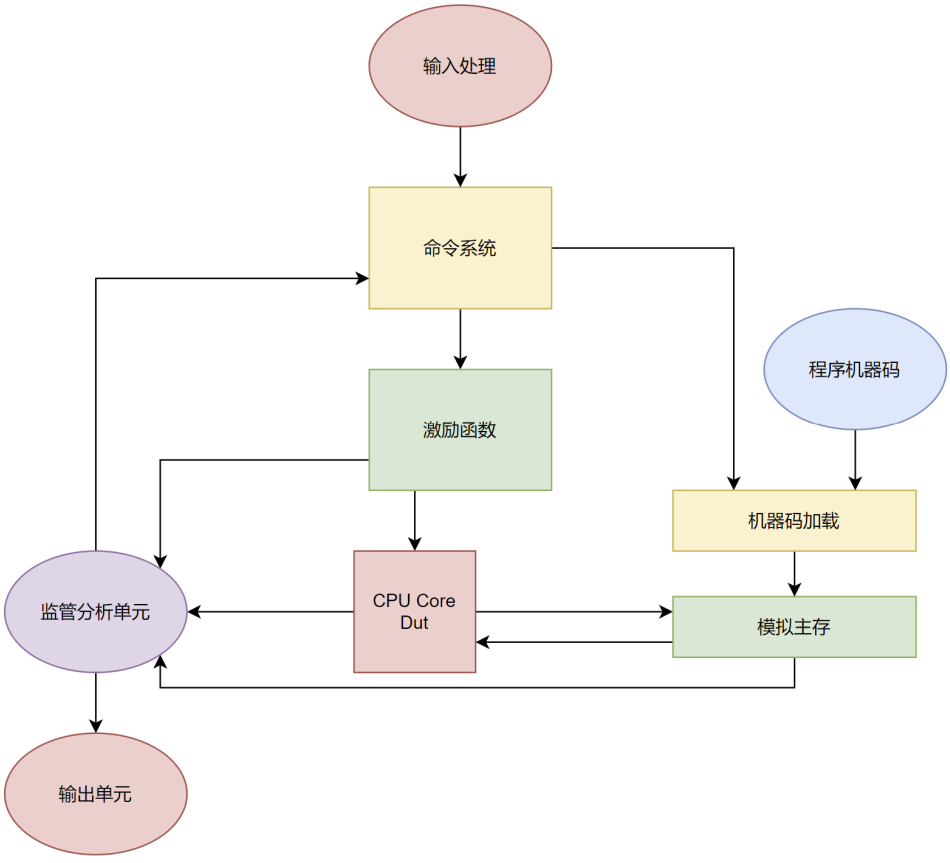

图 3.1　仿真框架简图

1. 命令系统

LSIM 配备了一个基础的人机交互命令系统，仿照 GDB 的各类命令，用户可以向程序输入多种调试命令，从而实现单步执行、连续执行、查看处理器状态等多种调试基本功能。

2. 机器码加载与模拟主存

为了实现冯·诺依曼架构的处理器系统[13]，我们采用软件来模拟实现主存。在仿真运行前，程序会根据主函数的参数获取到机器码的路径，并将对应的机器码复制到模拟主存的对应位置。模拟主存支持 SRAM 接口与 AXI4 总线接口，可以对其进行读写操作。

3. 激励函数

激励函数基于 Verilator，可以通过修改时钟、复位等处理器输入接口的值，并调用 Verilator 的 API 对电路状态进行计算。除此之外，激励函数也会驱动监管分析单元，对处理器的各项指标进行实时监测。

4. 监管分析单元

监管分析单元是整个软件框架的核心部分。这个部分每个周期会提取超标量处理器内部状态，经过一系列较为复杂的分析运算，将指令执行结果反馈给用户和命令系统。同时，这个单元也负责生成波形、记录处理器运行踪迹，当仿真框架需要暂停或用户发现问题时，可以及时将这些时间序列反馈给用户，便于用户快速锁定问题。

除此之外，该模块具有较好的可拓展性，用户可以根据自己的需求，在仿真框架中加入需要统计的数据（例如 IPC、stall 周期数等），在运行结束后自动生成一份运行报告。当需要大批量运行不同测试时，这些运行报告将帮助用户快速验证硬件算法的可行性和性能。

第三节　处理器监管分析技术

一、Difftest

Difftest 技术是在龙芯教育开源调试平台 Chiplab[14] 中提到了一种技术，旨在将处理器每一步的执行结果与“一个完全正确的处理器”进行对比，从而实现出错立刻停顿并输出错误信息，能够帮助开发者快速定位到错误点。

一般情况下，“完全正确的处理器”往往是通过软件模拟器实现的，这个模拟器一般是作为库的形式提供给主程序。每当处理器有一条指令提交，仿真代码

就会调用模拟器提供的 API，获得模拟器的执行指令后的“处理器状态”，并与我们设计的处理器的状态进行一一对比。一旦存在不同，则程序立刻停止仿真并输出对应的问题和调试信息。

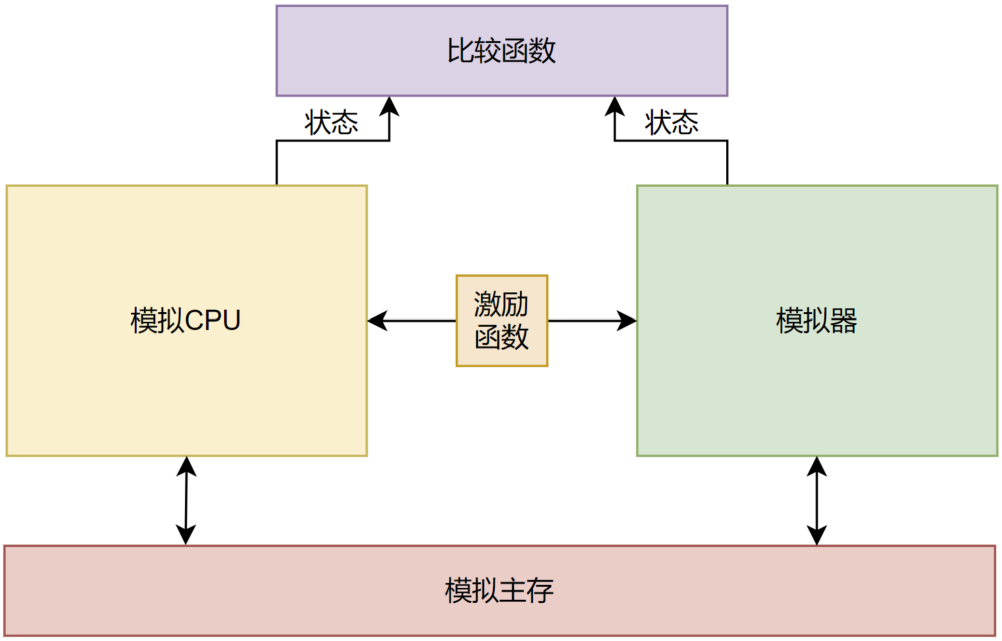

图 3.2　Difftest 原理图

传统的 Difftest 需要实时获取处理器状态[①]，这个过程使用了 Verilator 提供的 DPI-C 机制，将 verilog 代码中描述的某些变量直接传到仿真环境中。事实证明，这种实现方式较为耗时，涉及了大量数据的拷贝操作，不利于大规模仿真的进行。此外，由于 Chisel 不支持 DPI-C 机制，这导致 Chisel 开发者不得不在关键模块使用 BlackBox，并内嵌 verilog 代码进行仿真环境的适配——这显然和 Chisel “敏捷开发”的宗旨并不相符。

相较 Chiplab 基于开源项目 Nemu[15] 开发的 Difftest 框架，本项目中的 Difftest 框架采用了“软件维护处理器状态”的思路，通过将每个周期将寄存器堆写数据、写使能、写地址拉到顶层模块，并在仿真环境中用数组模拟寄存器堆的方法，极大减少了数据搬运的开销。除此以外，本项目不再使用 DPI-C 机制对信号进行读取，而是只读取顶层模块输出信号，增加了对 Chisel 生成的 verilog 代码的调试支持。

[①]这里的“状态”比较狭义，指的是处理器中 32 个通用寄存器、所有控制状态寄存器和 PC 值的集合

二、性能统计

超标量处理器的性能调优非常依赖于对各个缓存满、空周期数的统计，因为缓存的满、空往往会导致流水线的停顿或空转，造成一定程度上的性能损失。因此，我们在仿真框架中声明了一个 Statistic 统计类，这个统计类可以自定义成员与方法，每个周期这个统计对象都会根据处理器的输出信号更新自己的状态，从而实现对处理器各项指标的统计。当前性能统计已经支持的功能包括：

1. 分支预测正确率统计：支持 BL、B、JIRL 和条件分支指令的正确率分别统计
2. 流水线提交率：统计每个周期平均提交的指令数，是衡量 IPC 的直接指标
3. ROB 满率：ROB 满的周期数占总周期数的百分比
4. 重命名空闲列表空率：重命名空闲列表空的周期数占总周期数的百分比
5. 高速缓存命中率：访存命中高速缓存的比率
6. 发射队列发射率：各个发射队列平均每个周期发出的指令数

我们希望通过参数调整，能够让处理器各个部分减少停顿的时间，尽量让各个元件的停顿率相近，以求让流水线能够更顺畅地流动。这样一来，一方面可以提高 IPC，另一方面也能减少硬件资源的开销。当我们可以对这些指标进行统计后，我们就可以对超标量处理器的运行状态有一个整体的认知。例如，当 ROB 满率较低而重命名空闲列表总为空时，我们就可以分析得到，供重命名的物理寄存器太少，这时就可以加入一些物理寄存器，让指令尽量少地堵塞在这个位置。

第四章　预译码系统的创新与优化

第一节　预译码系统概述

一、预译码技术的主要思想

预译码技术是流水线前端的一种技术，旨在尽可能在前端修正一些由于分支预测失败导致的无谓性能损失[16]。由于乱序多发射处理器的流水级一般比较深，在执行阶段或提交阶段进行分支纠正的惩罚往往非常巨大[17]，一条分支指令预测失败可能会冲刷掉 10-20 条指令——这对分支预测的正确率要求也往往更高。但随着现代程序逐渐向大体量、远跳转的方向发展，分支预测器必须拥有更大的容量，但这对电路面积而言往往也是不现实的。

因此，我们考虑到一种折中的方案：当获取到指令的指令码后直接对其初步译码，对能够在前端判断出跳转方向和计算出跳转地址的指令，根据其预测结果直接进行修正。如此一来，一部分跳转指令的错误预测就可以很快得到修正，减少了处理器后端修正跳转预测带来的流水线冲刷惩罚。

二、预译码技术面临的问题

预译码技术在理论上是比较容易实现的，但在实际设计应用中也面临了许多问题。其一，预译码需要通过 32 位加法计算跳转目标地址，这对时序本不宽裕的 next pc 信号生成逻辑无疑是雪上加霜；其二，预译码强调及时性，如果我们采用寄存器来改善其时序，其对于 IPC 的提升也会大打折扣；其三，预译码对指令纠正的程度需要进行考量：如果需要对跳转地址预测错误进行纠正，那么 32 位比较运算的时序也不容忽视，如果只对跳转方向进行纠正，那么纠正效果往往也不尽如人意。因此，在设计预译码模块时，我们必须“牵一发而动全身”，将前端几个模块协同起来考虑，综合效率和时序两个方面进行设计。

第二节　跳转指令码立即数替换

一、跳转地址计算的流水级分配问题

在龙芯架构 32 位精简版指令集中，直接跳转的目标地址是通过 PC 与立即数的和来计算的。这个 32 位加法在执行单元并不会造成太大的延迟，但在预译

码单元，由于源操作数源自一个译码器，结果还需要参与 next pc 来源的选择，因此它的计算对电路面积和延迟都有比较大的影响。

指令从 Cache 读出到预译码给出结果的全部逻辑延迟大致可以表示为

$$T_{bram} + T_{pdecode} + T_{add32} \tag{4.1}$$

其中，T_{bram} 是 Block Memory 的读取延迟，$T_{pdecode}$ 是预译码逻辑的延迟，T_{add32} 是 32 位加法器的延迟。Block Memory 的延迟已经很大，因此，我们必须要将 Block Memory 取出的结果锁存一个周期。为保证预译码修正及时性，预译码和地址计算最好要在同一个周期完成，且后者依赖于前者的结果。因此这里需要有两种对组合逻辑优化的思路：

- 将分支方向纠正的逻辑和单纯判断分支类型的逻辑分离开，后者延迟应当更小，可以减少之后计算地址的前置延迟；
- 使用特殊的手段将加法计算进行强度削减，以减少其延迟。

第一种方法是比较容易实现的，只需要对所有分支指令的编码特点进行总结，减少其分析的组合电路复杂度即可。第二种方法则需要多级流水线配合，这里我们将重点介绍第二种方法的设计——指令码替换。

二、指令码替换思想在实现层次的问题

在高通公司 2016 年的 Falkor CPU[18] 的发布会上，该公司提出，当指令高速缓存将数据从主存取回时，可以对这些数据进行一次译码，并利用缺失虚地址和指令码中的立即数计算出跳转地址，并使用跳转地址替代掉原来的立即数。该公司并未详细说明实现的方法，但很显然这个说法对于我们而言存在两个挑战：

1. 高速缓存存储问题

目标跳转地址共有 32 位，即便通过压缩，也很难压缩到龙芯架构 32 位精简版指令集中分支指令的 16 位里。因此，我们无法存储跳转指令目标地址的所有位。

2. 高速缓存访问形式问题

该公司的这款 CPU 采用了 Virtual Tag Virtual Index[19] 的访问形式，所有的访问都是基于虚地址的。而笔者设计的处理器采用了 Physical Tag Virtual Index 的访问形式，这就意味着，如果一块物理地址对应了一条跳转指令，且处理器修改了地址映射，并且随后用映射到该物理地址的新的虚拟地址访问了这条跳转指令，那么替换掉原立即数的地址将会是基于旧虚拟地址计算的结果，这会导致

处理器计算不出正确的跳转地址。

三、ICache 适配设计

在 ICache 中，我们需要为每一个字添加一个 2bit 预译码信息，预译码信息编码含义如下：

- 00：跳转地址高位[①]与该指令虚拟 PC 的高位相同
- 01：跳转地址高位比该指令虚拟 PC 的高位大 1
- 11：跳转地址高位比该指令虚拟 PC 的高位小 1
- 10：保留编码

我们使用了 AXI4 总线结构的突发传输技术，在指令从主存取到 ICache 的过程中，指令会逐条回到 ICache。每有一条指令回到 ICache，ICache 就会使用一个预译码器识别其跳转类型，并根据虚拟地址计算出其跳转目标地址，替换掉原本的立即数后存入 ICache 的存储器。当流水线从 ICache 请求指令时，指令对应的预译码信息也会送出到流水线中。

由于地址映射变化会导致分支地址变化，一旦处理器的地址映射发生变化，高速缓存的所有表项就应当被无效。不过，由于地址映射变化是低频率事件，对处理器整体性能的影响也并不大。

四、预译码器的设计

LA32R 使用了类似 IP 编址的前缀码编码方式[7]，而所有跳转指令的操作码都很短，通过 [31:26] 位即可识别出其跳转类型，这使得预译码不需要关注过多位就可以完成跳转方向的判断。在预译码器中，所有指令会被划分为四类：

- 必定跳转指令：无条件跳转（B）、分支链接（BL）
- 可能跳转指令：有条件跳转
- 不支持的跳转指令：跳转链接指令（JIRL）
- 非跳转指令：其他指令

之所以 JIRL 会被列为不支持的跳转指令，是因为其跳转地址在预译码阶段无法计算，所以它必须听信分支预测器中返回地址栈给出的结果。而对于其他三类指令，预译码器会根据其指令码和立即数进行判断，给出其跳转方向。下图给出了在立即数替换的情况下，预译码器的设计：

[①]这里的“地址高位”和具体指令类型有关。LA32R 中，无条件分支和分支链接指令的“高位”指的是 [31:28] 位，而有条件跳转的“高位”指的是 [31:18] 位。

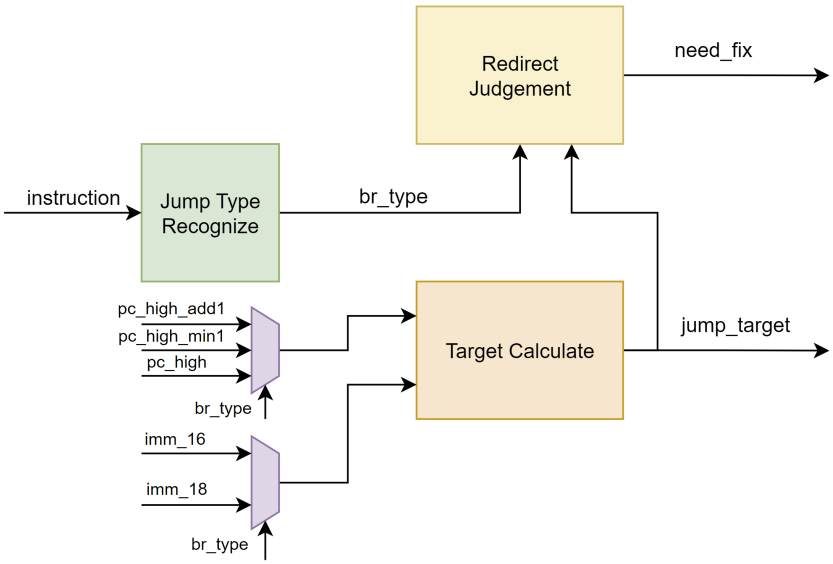

图 4.1　预译码器设计

我们的跳转方向逻辑纠正如下：

1. 必定跳转指令

这两条指令一定会引起指令流的跳转，因此当分支预测器没有预测其跳转，或者跳转地址预测错误时，预译码器都会发起纠正，纠正的跳转地址会在取指阶段传来的高位 PC+1、高位 PC、高位 PC-1 中选取一个，再拼接上指令中的立即数。

2. 条件分支指令

条件分支指令的跳转方向依赖于具体寄存器的值，因此在预译码阶段无法进行完备的判断。因此，若分支预测器确实给出了条件分支指令的预测结果，那么预译码器就不会再对其进行纠正。但若分支预测没有对其进行有效预测，那么预译码器会做出静态预测。

3. 非跳转指令

由于自修改程序的存在，当分支预测器看到一条跳转指令后，若将其修改为非跳转指令，则之后分支预测器还是会给出跳转结果，而并非所有指令都会进入后端的第二运算单元进行分支判断，故这种情况会导致处理器运行错误。预译码器可以在预译码阶段将这种错误纠正，避免错误的指令流进入后续流水线。

五、指令码恢复

为了让后端缓存减小面积，并尽量减少后端流水线的结构和功能，预译码器会对指令编码进行恢复，即将替换到指令中的立即数再替换为原有指令中的立即数。

第五章　基于 LSIM 的仿真功能与性能探究

在之前的章节中，我们已经介绍了超标量处理器、处理器仿真开发环境 LSIM 的设计与实现和预译码技术的设计与实现。在本章中，我们将基于自主开发的 LSIM 仿真环境，对处理器的功能进行仿真，并探究预译码技术对处理器性能的影响。

第一节　仿真 Soc 的搭建

为了验证处理器的正确性，我们使用 LSIM 和第二章所描述的处理器搭建了一个用于仿真的片上系统。这个片上系统的主存和外围设备均由 LSIM 模拟实现。处理器通过 AXI4 总线与内存单元进行通信，并通过 MMIO 进行外围设备访问。整体 Soc 示意图如下：

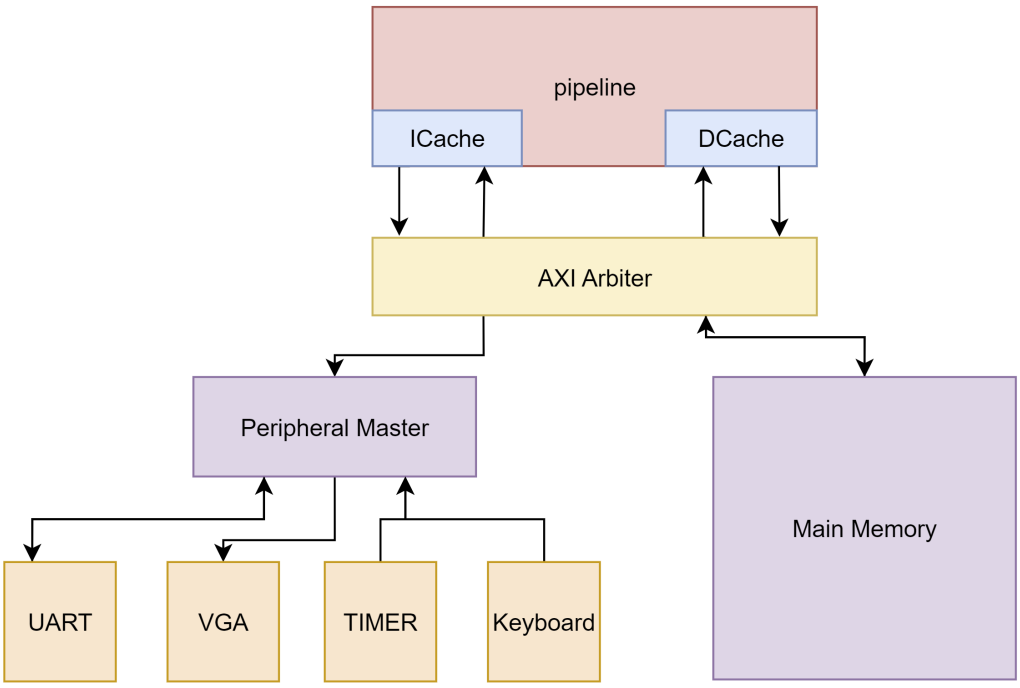

图 5.1　仿真 Soc 原理图

ICache 和 DCache 使用类似 SRAM 的访存方式向 AXI 仲裁桥发起访存请求，AXI 仲裁桥收到请求后，会对两个请求进行仲裁，然后将双 Cache 的请求转换为 AXI 总线信号，并向仿真环境中的模拟内存单元发起访问。内存单元收到请求后，会根据请求的地址，选择访问外设或是主存，并返回访存结果。LSIM 支持的外设在处理器内的地址映射如下：

表 5.1　仿真 Soc 内存地址映射表

外设	地址范围
UART	0xa00003f8 - 0xa00003ff
KEYBOARD	0xa0000060 - 0xa0000067
TIMER	0xa0000040 - 0xa000005f
FB	0xa1000000 - 0xa1008fff
VGACTL	0xa0000100 - 0xa000010f

第二节　测试程序构建

为了验证处理器的功能和性能，我们需要构建一系列的测试程序。虽然开发者社区中有完整的 LA32R 编译工具[20]，但由于其链接库对于我们而言并不透明，编译出的可执行程序的很多细节我们并不清楚，调试过程会给我们造成很多困难。因此，我们决定自己搭建一个简单的编译环境，以便于我们更好地理解程序的运行过程。

一、交叉编译工具链

为了方便测试，我们使用了 LA32R-GCC 8.3.0 交叉编译工具链作为编译工具。该开源编译工具发布在 gitee 上，适配了龙芯架构的新世界 2.0ABI[21]。

二、应用程序基础设施搭建

除此以外，由于 GCC 对于程序代码的布局和链接有自己的规定，而这些规定有时候并不能很好地适配到我们的处理器上，因此我们需要自己为应用程序搭建一个简单的运行时环境，支持外设访问、引导程序等功能。主要功能代码如下：

- **start.S**：程序入口，设置栈指针等。
- **uart.c**：UART 外设驱动，提供了 putc 等函数。
- **vga.c**：FrameBuffer 外设驱动，提供了矩形绘制等函数。
- **kbd.c**：键盘外设驱动，提供了读取键盘输入的函数。
- **rtc.c**：定时器外设驱动，提供了获取当前时间的函数。

三、测试程序通用编译流程

借助交叉编译工具链和应用程序基础设施，我们就可以使用一个通用编译框架对任意的 C 程序和 LA32R 汇编程序进行编译，以得到。编译流程如下：

- **编译基础设施**：使用交叉编译工具编译所有基础设施，并生成链接库。如

果无需更新基础设施，那么这个步骤只会执行一次。

- **编译测试程序**：收集所有需要编译的 C 程序和汇编程序，使用交叉编译工具链编译 C 程序，生成对象文件。
- **链接基础设施**：将基础设施的链接库取代 GCC 默认的链接库，生成可执行文件。
- **生成二进制文件**：使用 objcopy 工具将可执行文件转换为二进制文件，以便于 LSIM 加载。
- **生成反汇编文件**：使用 objdump 工具将可执行文件转换为反汇编文件，以便于查看程序的运行过程。

第三节　处理器功能仿真验证

为了更加全面地验证处理器功能的正确性，我们分别从龙芯教育开源调试平台 Chiplab 提供的测试程序、南京大学 ICS 课程提供的测试程序等多个仓库进行了测试程序移植，并对处理器进行了测试。

借助 LSIM 的 Difftest 功能，我们可以很容易知道处理器在运行过程中是否出现了错误：若处理器运行途中的状态与模拟器运行的状态不一致，Difftest 会立刻停止仿真，并输出错误信息和“ABORT”信息；而若处理器顺利执行完所有指令，Difftest 会输出“HIT GOOD TRAP”信息。接下来我们将以 Chiplab 和南京大学 ICS 课程提供的测试程序为例，分析测试程序的测试重点，并展示仿真结果。

一、Chiplab 测试程序

Chiplab 提供的测试程序由汇编语言编写，每一个测试都对应了一种指令或一种异常中断。这些测试程序可以很好地验证处理器的基本功能，且测试层层递进，大体上只会使用已经测试过的指令构建新的测试用例。我们使用了 Chiplab 提供的测试程序对处理器进行了测试，LSIM 反馈结果如表 5.2所示。从结果可知，我们的处理器可以较好地支持 LA32R 指令集的基本功能。

二、南京大学 ICS 课程测试程序

南京大学 ICS 课程测试程序被称为 functest，是由 C 语言开发的多个测试程序的集合。该系列程序不再针对单个指令进行测试，而是借助一些特定的程序功

表 5.2　Chiplab 测试程序测试结果

测试程序	结果	测试程序	结果	测试程序	结果
n1_lu12i_w	PASS	n2_add_w	PASS	n3_addi_w	PASS
n4_sub_w	PASS	n5_slt	PASS	n6_sltu	PASS
n7_and	PASS	n8_or	PASS	n9_xor	PASS
n10_nor	PASS	n11_slli_w	PASS	n12_srli_w	PASS
n13_srai_w	PASS	n14_ld_w	PASS	n15_st_w	PASS
n16_beq	PASS	n17_bne	PASS	n18_bl	PASS
n19_jirl	PASS	n20_b	PASS	n21_pcaddu12i	PASS
n22_slti	PASS	n23_sltui	PASS	n24_andi	PASS
n25_ori	PASS	n26_xori	PASS	n27_sll_w	PASS
n28_sra_w	PASS	n29_srl_w	PASS	n30_div_w	PASS
n31_div_wu	PASS	n32_mul_w	PASS	n33_mulh_w	PASS
n34_mulh_wu	PASS	n35_mod_w	PASS	n36_mod_wu	PASS
n37_blt	PASS	n38_bge	PASS	n39_bltu	PASS
n40_bgeu	PASS	n41_ld_b	PASS	n42_ld_h	PASS
n43_ld_bu	PASS	n44_ld_hu	PASS	n45_st_b	PASS
n46_st_h	PASS	n47_syscall_ex	PASS	n48_brk_ex	PASS
n49_ti_ex	PASS	n50_ine_ex	PASS	n51_soft_int_ex	PASS
n52_adef_ex	PASS	n53_ale_ld_w_ex	PASS	n54_ale_ld_h_ex	PASS
n55_ale_ld_hu_ex	PASS	n56_ale_st_h_ex	PASS	n57_ale_st_w_ex	PASS
n58_rdcnt	PASS	n59_tlbrd_tlbwr	PASS	n60_tlbfill	PASS
n61_tlbsrch	PASS	n62_invtlb_0x0	PASS	n63_invtlb_0x1	PASS
n64_invtlb_0x2	PASS	n65_invtlb_0x3	PASS	n66_invtlb_0x4	PASS
n67_invtlb_0x5	PASS	n68_invtlb_0x6	PASS	n69_invtlb_inv_op	PASS
n70_tlb_4MB	PASS	n71_tlb_ex	PASS	n72_dmw	PASS
n73_icacop_op0	PASS	n74_dcacop_op0	PASS	n75_icacop_op1	PASS
n76_dcacop_op1	PASS	n77_icacop_op2	PASS	n78_dcacop_op2	PASS
n79_cache_writeback	PASS	n80_ti_ex_idle	PASS	n81_atomic_ins	PASS

能，对处理器的整体功能进行测试。同时，由于 functest 需要借助交叉编译器进行编译，它也能更好的测试在真实编译环境下处理器是否还能正常工作。我们使用了 functest 对处理器进行了测试，LSIM 反馈结果如表 5.3所示。测试结果表明，在基于高级语言的测试程序中，我们的处理器也能够较为稳定地工作。

第四节　处理器性能探究

在性能测试部分，我们移植了若干通用的性能测试程序，包括 CoreMark、Dhrystone、FireEye 等[22]。这些程序各有特色，能够很好地测试处理器的各类性能。我们将在下文中展示这些测试程序的测试结果，并将处理器与参考跑分、2023 年“龙芯杯”系统能力大赛特等奖作品进行对比，探索乱序多发射和预译码技术对处理器性能的影响。

表 5.3　南京大学 ICS 课程测试程序测试结果

测试程序	结果	测试程序	结果	测试程序	结果
add-longlong	PASS	add	PASS	bit	PASS
bubble-sort	PASS	crc32	PASS	div	PASS
dummy	PASS	fact	PASS	fib	PASS
goldbach	PASS	hello-str	PASS	if-else	PASS
leap-year	PASS	load-store	PASS	matrix-mul	PASS
max	PASS	mersenne	PASS	min3	PASS
mov-c	PASS	movsx	PASS	mul-longlong	PASS
pascal	PASS	prime	PASS	quick-sort	PASS
recursion	PASS	select-sort	PASS	shift	PASS
shuixianhua	PASS	string	PASS	sub-longlong	PASS
sum	PASS	switch	PASS	to-lower-case	PASS
unalign	PASS	wanshu	PASS		

一、乱序多发射的性能测试

为了测试乱序多发射对于性能的提升，我们从 Chiplab 中移植了性能测试 FireEye。这个性能测试会根据一个基准核给出一个性能分数，并提供直观的对比。不过，由于我们无从得知这个基准核的设计方式，无法探究乱序多发射对于性能的提升，因此我们附加使用 2022“龙芯杯”大赛 Loongarch 挑战赛道亚军 Clap 战队的作品作为对比。测试结果如表 5.4所示。

表 5.4　FireEye 测试结果

Benchmark	Ref.	Clap	Run Time
A0_1-limits01	43.5077	8.59177	8.19723
A1_1-large03	62.2865	29.5317	27.8985
B0_10-test19	508.288	93.0654	90.2945
B2_50-test19	333.118	82.8083	75.4324
B3_50-test19	279.832	83.2205	80.9683
B5_500-test19	554.867	156.22	139.4254
D0_100-test07	286.818	95.9582	92.0052
D0_1000-test10	553.577	246.948	208.9571
D0_1000-test06	119.142	123.142	108.583
F0_1-test8	47.4547	11.4117	9.15732
G0_10-test03	898.276	187.288	179.435
G0_10-test06	587.155	136.769	122.646
H0_40-test03	349.407	84.2216	77.5346
I2_10-test1	167.637	26.3194	20.3536
coremark-0	177.385	21.3837	18.6476
coremark-1	177.932	21.4363	18.2179
dhrystone	197.368	39.6035	36.2342

注：Ref. 为 FireEye 参考核运行时间，Clap 为 Clap 战队处理器执行时间，Run Time 为我们的处理器执行时间

Clap 战队的处理器是一款顺序双发射的处理器，而我们的处理器是一款乱序四发射的处理器。FireEye 测试以规模化和安全性著称，因此其中的许多功能函数是计算密集型的。从测试结果来看，我们的处理器在 FireEye 测试中的性能

表现优于 Clap 战队的处理器。这表明，特别是在计算密集型程序中，乱序多发射技术对于处理器性能的提升是显著的。

二、预译码的性能测试

1. 预译码对分支指令性能的影响

预译码器最主要的功能是缓解分支预测失败对于指令流的影响。在我们的实现中，预译码器可以只对分支预测方向进行纠正（Pre_Dir），也可以同时对分支预测的方向和分支预测地址进行纠正（Pre_All）。我们使用了 CoreMark、Dhrystone、Quick-Sort、Bubble-Sort 四个测试程序，对比了不使用预译码（Pre_No）和使用预译码时的总分支预测正确率（即经过分支预测和预译码修正之后的分支预测正确率），结果如表 5.5 所示。

表 5.5　不同预译码配置下总分支指令预测正确率

测试程序	分支指令	Pre_No	Pre_Dir	Pre_All
coremark	Branch	84.2034%	86.9294%	87.8953%
	JIRL	86.4243%	88.5354%	92.3559%
	BL&B	82.8743%	86.425%	100%
dhrystone	Branch	95.0194%	96.2738%	97.4319%
	JIRL	64.8172%	66.7817%	75.2983%
	BL&B	97.1942%	97.1932%	100%
quick-sort	Branch	82.1049%	85.9861%	86.0724%
	JIRL	90.8741%	92.1312%	94.8052%
	BL&B	88.3132%	90.9381%	100%
bubble-sort	Branch	83.1353%	86.7424%	88.1374%
	JIRL	89.4234%	90.8752%	93.0233%
	BL&B	86.4235%	95.7723%	100%

注：JIRL 只包括作为返回指令使用的 JIRL 指令，不包括作为间接跳转链接的 JIRL 指令

对于分支方向和地址都纠正的预译码而言，一个最显著的特点就是无条件跳转和跳转链接指令的总预测率都可以到达 100%，这是因为它们在前端就可以完全判断出是否跳转，从而预译码器能够完全纠正。由于分支预测中返回地址栈的存在，一旦跳转链接指令的预测正确率提高，那么作为返回指令的 jirl 指令的预测正确率也会提高。

对于有条件跳转指令，由于其跳转方向依赖于具体寄存器的值，因此在预测码阶段无法进行完备的判断。不过，对于分支预测没有给出的指令，基于跳转地址的静态预测可以提高预测正确率，这对于循环体较大的程序有一定好处。

2. IPC 性能测试

作为基于 FPGA 开发的处理器，我们采用几种通用性能测试程序，通过仿真进行 IPC 测试，与 2023 年龙芯杯全国系统能力大赛团体赛道全国特等奖的处理器 NOP 进行比较，结果如表 5.6所示。

表 5.6　预译码模块对处理器 IPC 的影响

测试程序	无预译码	有预译码	NOP
coremark	0.74	0.87	0.87
dhrystone	1.08	1.10	0.89
quick-sort	0.64	0.74	0.68
bubble-sort	0.58	0.65	0.63

注：无预译码所在列为不使用预译码时处理器仿真运行对应测试程序的 IPC，有预译码所在列为使用预译码时处理器仿真运行对应测试程序的 IPC

对于 bubble-sort 和 dhrystone 这一类代码长度较短的测试，由于分支预测可以记录大部分分支指令，故预译码的改进程度并不大。但对于 coremark 这一类代码长度较长的测试，分支预测器会比较频繁地进行替换，分支预测的准确率受到了较大的影响，这时预译码的效果就比较明显地显现出来。

通过对 coremark 的细致研究可以看出，在我们的处理器配置下，预测表有多个被多次替换的项是跳转链接和条件跳转指令相互替换，这时局部预测的计数器会受到一定程度的影响，导致预测结果波动较大。此时预译码器对条件分支的静态预测和跳转链接的修正就发挥了比较重要的作用。

第五节　本章小结

本章中，我们基于自主开发的 LSIM 仿真环境，对处理器的功能进行了仿真，并探究了预译码技术对处理器性能的影响。我们首先介绍了仿真 Soc 的搭建，然后介绍了测试程序的构建和通用编译流程。接着，我们对处理器的功能进行了仿真验证，分别从 Chiplab 和南京大学 ICS 课程提供的测试程序进行了测试。最后，我们探究了乱序多发射和预译码技术对处理器性能的影响，通过 FireEye、CoreMark、Dhrystone 等测试程序的测试，展示了预译码技术对处理器性能的提升。

第六章　片上系统时序优化探索

在本章中，我们将基于 Vivado2023.1，对由我们设计的超标量处理器组成的片上系统进行时序优化探索，讨论重要元件的重要时序优化方法。之后我们将结合之前章节介绍的预译码技术改进手段，探索该创新对于上板时序的影响。

第一节　SOC 的搭建

为了对处理器进行上板验证，我们为处理器核搭建了简单的外围设备，使其能够支持输入输出，从而能够作为一个独立的整体存在于开发板上。整体 SOC 结构如下图所示：

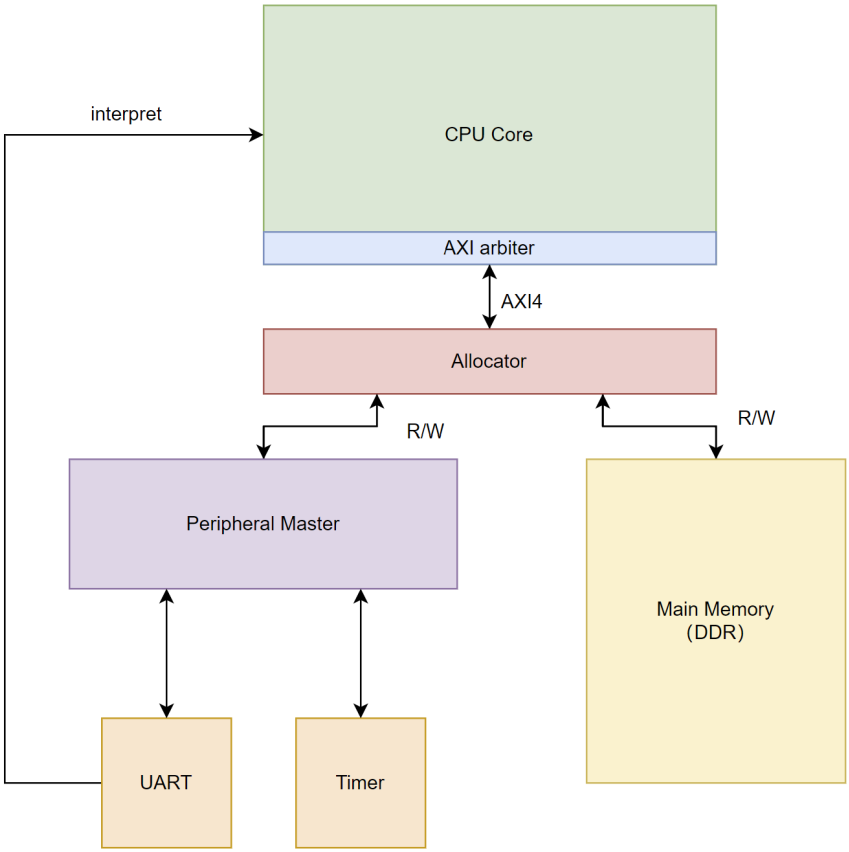

图 6.1　Soc 结构图

该 SOC 使用开发板上的 DDR 作为主存，使用时钟同步连接到访存请求分配器（Allocator）上。在外设方面，片上系统采用 AXI4 总线协议对外设进行访问，支持串口输入输出和实时时钟两个外设。同时，串口输入可以向处理器发送

中断信号，使处理器可以通过中断处理程序获取到串口的输入数据。

处理器访问内存系统的访问方式、内存地址映射与上一章中提及的仿真 SOC 系统原理没有区别，但值得一提的是，由于我们需要使用硬件决定本次访存访问外设还是主存，所以需要使用访存分配器对 AXI 总线信号进行转接。

第二节　基于 Vivado 的时序优化探索

在上板综合、实现的过程中，我们如果希望提升时钟频率，就必须查看每一次实现的结果中的超时线路，并针对其中提及的问题进行改良优化。在实验的过程中，我们逐步将处理器时钟频率从 60MHz 提升到了 120MHz，本节将会介绍我们在这个过程中关键的时钟频率优化手段。

一、交叠技术

对于超标量处理器设计而言，多读写端口的 FIFO 是十分常见的。如果 FIFO 存在 m 项，并存在 n 个端口，那么就会存在复杂度为 $O(m \times n)$ 的访问电路面积开销。但在很多情况下，如果每次对 FIFO 访问都是稳定从头部读取 n 个数据，那么对于 m 项存在 n 个读端口的 FIFO，我们可以将一个大 FIFO 拆解成 n 个容量为 m/n 的小 FIFO，那么这时同时访问的电路面积就变为 $O(m)$，这种时序优化是非常明显的。下图即为 2 路交叠（interleaving）技术的示意图：

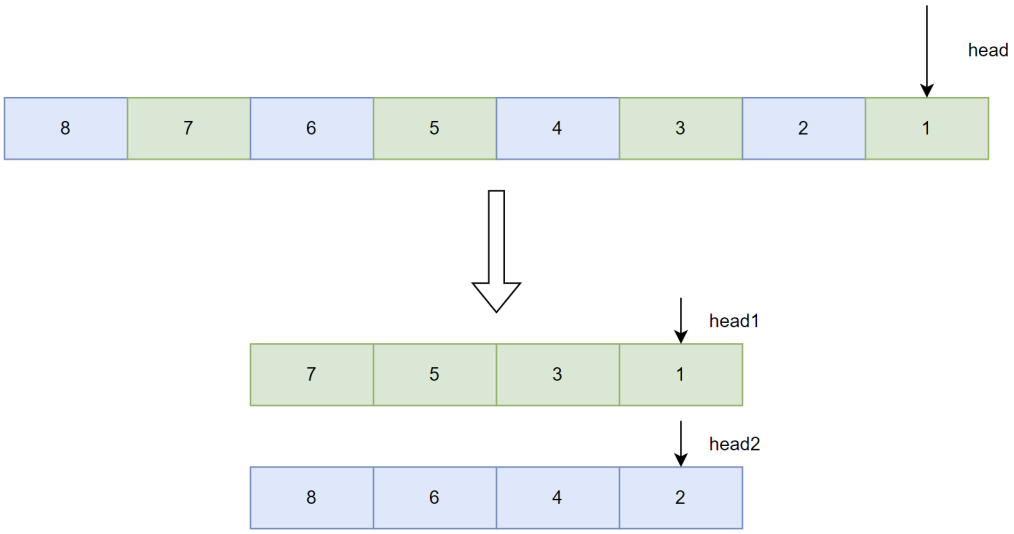

图 6.2　2 路交叠技术示意图

我们的处理器多处采用了交叠技术进行优化，下面我将进行逐步介绍：

1. 取指队列

取指队列负责暂存 ICache 中取出的指令，本质也是一个 FIFO 队列。由于处理器前端每个周期能够处理两条指令，取指队列每个周期都可以从其头部向后续流水线发送两条指令，因此取指队列可以采用交叠技术分割为两个独立的 FIFO 队列，每个队列每个周期都可以向后发送一条指令。

在应用交叠技术前，处理器在 120MHz 的时钟约束下，存在一条从发射队列输出端到译码器，再到空闲寄存器列表输入端的超时路径。而使用了交叠技术后，这条路径满足了时序要求，同时取指队列的查找表资源使用量也从 1368 削减到 1119，削减比率达到了 27.2%。

表 6.1　取指队列时序优化前后资源使用量对比

指标	优化前使用量	优化后使用量
LUT 使用量	1368	1119
FF 使用量	839	845

2. 重排序缓存

重排序缓存（ROB）是处理器中最繁忙的部件之一，每条指令都需要与其进行至少两次交互。在重命名阶段，每条指令都会从 ROB 中获取一个表项编号，且为了减少后端的负载量，每个指令会将一些无需在后端计算的信息（例如 pc、目的物理寄存器堆编号、分支类型等）在这个阶段写入重排序缓存，这无疑会造成极大的电路开销。因此，我们将 ROB 也使用了两路交叠技术进行改进，极大程度上削减了 ROB 部分的面积。

在应用了交叠技术后，ROB 部分的查找表使用量从 6258 削减到 5236，削减比达到了 14.4%。同时，由于 ROB 的面积得到了很大的优化，SOC 整体设计的线延迟也得到了很好的优化，下表为对 ROB 时序优化前，十条总时序裕量最少的通路在优化前后的线延迟对比：

表 6.2　仿真 Soc 内存地址映射表

时序路径	优化前 NetDelay	优化后 NetDelay
PATH1	7.682	6.893
PATH2	7.893	7.021
PATH3	6.231	6.022
PATH4	7.294	5.992
PATH5	5.253	5.214
PATH6	5.976	4.874
PATH7	6.241	5.698
PATH8	7.542	6.983
PATH9	7.392	7.058
PATH10	6.438	5.291

二、跳转指令立即数替换技术

在第四章中，我们介绍了跳转指令立即数替换技术的实现思路，并在第五章中通过仿真验证了预译码技术的可行性。在这一部分，我们将基于全局的思路，对跳转指令立即数替换技术的时序优化效果进行对比。

在进行跳转立即数替换优化前，最大时钟频率为 115.6MHz，其中最长路径是取指——预译码段间寄存器，经过预译码器地址计算和跳转判断后，送入 next pc 生成模块参与生成逻辑，最后送入 pc 寄存器。跳转立即数替换优化后，最大时钟频率提升到 120.0MHz，预译码的 next pc 生成也不再成为关键路径。

表 6.3　跳转立即数替换优化前后时序对比

优化前时钟频率	优化后时钟频率
115.6MHz	120.0MHz

第三节　电路综合实现结果

一、SOC 资源综合实现结果

在 SOC 综合实现的过程中，我们将处理器核与外设连接，形成了一个完整的 SOC 系统。SOC 综合实现的资源占用情况如表 6.3所示，其中包括了 LUT、FF、BRAM、DSP 等资源的使用情况。在资源使用过程中，我们尽可能降低了分布式存储器的使用，将所有大规模存储器都使用了 BRAM 进行替代，以减少面积和时序开销。

Summary

Resource	Utilization	Available	Utilization %
LUT	33020	134600	24.53
LUTRAM	1	46200	0.00
FF	21032	269200	7.81
BRAM	92	365	25.21
IO	18	400	4.50
MMCM	1	10	10.00

图 6.3　SOC 综合实现资源占用情况

二、SOC 时序综合实现结果

图 6.4展示了 SOC 综合实现中的关键路径。这条关键路径从 MMU 中 TLB 的命中逻辑开始，到写缓冲的读缓存逻辑结束，其路径约束为 7.605ns，满足了时序约束。

Delay Type	Incr (ns)	Path ...	Location	Cell...	Cell	Netlist Resources
FDRE (Prop_fdre_C_Q)	(r) 0.379	-0.106	Site: SLICE_X103Y156	Q	d_tlb_entry_r_6_reg (FDRE)	CPU_inst/mmu/tlb/d_tlb_entry_r_6_reg/Q
net (fo=51, routed)	0.863	0.756				CPU_inst/mmu/tlb/d_tlb_entry_r_6
			Site: SLICE_X105Y155	I1	sb_addr_0[28]_i_29 (LUT4)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_29/I1
LUT4 (Prop_lut4_I1_O)	(r) 0.105	0.861	Site: SLICE_X105Y155	O	sb_addr_0[28]_i_29 (LUT4)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_29/O
net (fo=1, routed)	0.237	1.098				CPU_inst/mmu/tlb/sb_addr_0[28]_i_29_n_0
			Site: SLICE_X104Y156	I4	sb_addr_0[28]_i_19 (LUT5)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_19/I4
LUT5 (Prop_lut5_I4_O)	(r) 0.105	1.203	Site: SLICE_X104Y156	O	sb_addr_0[28]_i_19 (LUT5)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_19/O
net (fo=2, routed)	0.685	1.888				CPU_inst/mmu/tlb/sb_addr_0[28]_i_19_n_0
			Site: SLICE_X104Y154	I4	sb_addr_0[28]_i_9 (LUT6)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_9/I4
LUT6 (Prop_lut6_I4_O)	(r) 0.105	1.993	Site: SLICE_X104Y154	O	sb_addr_0[28]_i_9 (LUT6)	CPU_inst/mmu/tlb/sb_addr_0[28]_i_9/O
net (fo=25, routed)	1.073	3.066				CPU_inst/mmu/tlb/_d_tlb_hit_entry_T_5
			Site: SLICE_X110Y152	I2	sb_addr_0[15]_i_2 (LUT6)	CPU_inst/mmu/tlb/sb_addr_0[15]_i_2/I2
LUT6 (Prop_lut6_I2_O)	(r) 0.105	3.171	Site: SLICE_X110Y152	O	sb_addr_0[15]_i_2 (LUT6)	CPU_inst/mmu/tlb/sb_addr_0[15]_i_2/O
net (fo=1, routed)	0.513	3.683				CPU_inst/mmu/tlb/_tlb_io_d_paddr[15]
			Site: SLICE_X110Y144	I4	sb_addr_0[15]_i_1 (LUT5)	CPU_inst/mmu/tlb/sb_addr_0[15]_i_1/I4
LUT5 (Prop_lut5_I4_O)	(r) 0.105	3.788	Site: SLICE_X110Y144	O	sb_addr_0[15]_i_1 (LUT5)	CPU_inst/mmu/tlb/sb_addr_0[15]_i_1/O
net (fo=12, routed)	0.670	4.458				CPU_inst/sb/D[15]
			Site: SLICE_X117Y138	I0	ld_hit_index[3]_i_267 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_267/I0
LUT6 (Prop_lut6_I0_O)	(f) 0.105	4.563	Site: SLICE_X117Y138	O	ld_hit_index[3]_i_267 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_267/O
net (fo=1, routed)	0.507	5.070				CPU_inst/sb/ld_hit_temp_20[13]
			Site: SLICE_X117Y137	I2	ld_hit_index[3]_i_179 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_179/I2
LUT6 (Prop_lut6_I2_O)	(r) 0.105	5.175	Site: SLICE_X117Y137	O	ld_hit_index[3]_i_179 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_179/O
net (fo=1, routed)	0.237	5.412				CPU_inst/sb/ld_hit_index[3]_i_179_n_0
			Site: SLICE_X117Y135	I5	ld_hit_index[3]_i_83 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_83/I5
LUT6 (Prop_lut6_I5_O)	(f) 0.105	5.517	Site: SLICE_X117Y135	O	ld_hit_index[3]_i_83 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_83/O
net (fo=1, routed)	0.454	5.972				CPU_inst/sb/ld_hit_index[3]_i_83_n_0
			Site: SLICE_X117Y135	I5	ld_hit_index[3]_i_25 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_25/I5
LUT6 (Prop_lut6_I5_O)	(r) 0.105	6.077	Site: SLICE_X117Y135	O	ld_hit_index[3]_i_25 (LUT6)	CPU_inst/sb/ld_hit_index[3]_i_25/O
net (fo=4, routed)	0.354	6.430				CPU_inst/sb/ld_hit_temp_2
			Site: SLICE_X116Y136	I0	ld_hit_index_2[3]_i_4 (LUT2)	CPU_inst/sb/ld_hit_index_2[3]_i_4/I0
LUT2 (Prop_lut2_I0_O)	(r) 0.105	6.535	Site: SLICE_X116Y136	O	ld_hit_index_2[3]_i_4 (LUT2)	CPU_inst/sb/ld_hit_index_2[3]_i_4/O
net (fo=3, routed)	0.682	7.217				CPU_inst/sb/ld_hit_2_2
			Site: SLICE_X121Y131	I2	ld_hit_data_2_r_i_1 (LUT6)	CPU_inst/sb/ld_hit_data_2_r_i_1/I2
LUT6 (Prop_lut6_I2_O)	(r) 0.105	7.322	Site: SLICE_X121Y131	O	ld_hit_data_2_r_i_1 (LUT6)	CPU_inst/sb/ld_hit_data_2_r_i_1/O
net (fo=2, routed)	0.283	7.605				CPU_inst/sb/ld_hit_data_2_r_i_1_n_0

图 6.4　SOC 综合实现关键路径

经过时序优化，综合实现后的时序报告如图 6.5、6.6所示，其展示了整体 SOC 综合实现的时序约束，其中片上时钟为 100MHz，经过 PLL 倍频后，接入 SOC 的时钟最大为 120MHz。

Name	Waveform	Period (ns)	Frequency (MHz)
clk	{0.000 5.000}	10.000	100.000
clkfbout_ip_clock	{0.000 25.000}	50.000	20.000
cpu_clk_ip_clock	{0.000 4.167}	8.333	120.000

图 6.5　SOC 综合实现时序报告

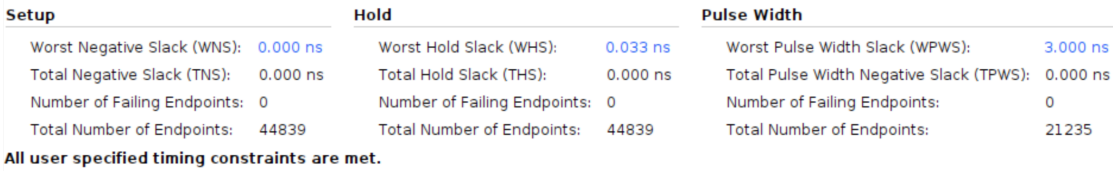

图 6.6　SOC 综合实现时序约束

第四节　本章小结

本章中，我们基于 Vivado2023.1，对由我们设计的超标量处理器组成的片上系统进行时序优化探索，讨论了重要元件的重要时序优化方法。我们首先介绍了 SOC 的搭建，然后逐步介绍了交叠技术和跳转指令立即数替换技术的优化效果。最后，我们展示了 SOC 的资源占用情况和时序综合实现结果。

第七章　成果总结与展望

第一节　成果总结

在本项目中，我们基于龙芯架构 32 位精简版指令集，完全自主地实现了一款乱序四发射的超标量处理器，并为之搭建了较为完备的仿真测试平台，通过仿真、验证、上板等一系列工作，验证了处理器的正确性和高效性。以下是本文做出的工作总结：

- 调研目前主流的超标量处理器设计方法，总结了其优缺点，为我们的设计提供了重要的参考；
- 结合 LA32R 指令集特点，设计了一款四发射的乱序超标量处理器，实现了处理器的大部分功能模块，包括发射、执行、访存等模块，并对其中的关键模块进行了详细的解读；
- 基于国际商用处理器的设计思路，设计了一套适用于我们处理器的预译码技术，很大程度上缓解了处理器的时序问题；
- 搭建了一套完备的处理器仿真测试平台 LSIM，为处理器的功能验证提供了重要的工具支持，并借助当前主流的功能、性能测试程序，验证了处理器的正确性和高效性；
- 借助 Vivado 工具，将处理器的核心部分搭建成了一个 SOC 系统，为处理器的上板验证提供了重要的基础，并通过时序优化，将处理器的时钟频率提升到了 120MHz。

第二节　应用与推广

一、开发者社区开源工作

目前，该项目已经在 Github 开发者社区使用 Apache-2.0 License 开源[23]，为国内外的硬件爱好者、学生、教师提供了一个学习、交流的平台。截至 2024 年 6 月，该项目已经获得了多个 Star，吸引了来自国内外的硬件爱好者的关注，为笔者带来了超过 30 位 Followers，并为国内外的硬件教育提供了一个重要的参考。

二、中国科大《计算机系统综合实验》课程

2023 年秋季学期，本项目中的 LSIM 初版仿真框架第一次应用到了《计算系统综合实验》课程[24]的实验框架中，结束了中国科大硬件课程只能使用 Vivado 波形调试硬件仿真的时代。在课程中，LSIM 框架为同学们提供了 Difftest、指令踪迹等一系列高效调试手段，帮助同学们从零开始搭建起了流水线系统，并运行了简单的操作系统。

三、中国科大《计算机组成原理》课程

2024 年春季学期，基于新版 LSIM 的课程实验仿真框架第一次应用到《计算机组成原理》课程实验中。经过课程组的教师、助教们的共同努力，LSIM 的仿真功能更便捷、更高效，降低了同学们对硬件调试的时间成本。同时，由于调试的便捷，整体实验设计不再被简单模块所桎梏，能够向着更深入的层次设计，带领同学们实现更多更复杂的处理器模块。

LSIM 中用以实现 Difftest 的模拟器也成为了《计算机组成原理》实验中 Loongarch32R 汇编与实时模拟器（LARS）的重要蓝本，同时，LARS 作为国内高校第一款在线模拟平台，为龙芯架构进入高校硬件课程提供了重要的工程条件。

四、“一生一芯”学员“龙芯杯”参赛教程

本项目的超标量处理器设计方法已被中国科学院“一生一芯”项目采纳，成为中国科学院大学 2024 年第八届“龙芯杯”处理器设计大赛参赛队员的重要参考设计。在中国科学院计算技术研究所的支持下，该项目将结合 GEM5 模拟器，为参赛队员们提供设计高性能复杂处理器提供有效的帮助。

第三节　未来展望

从技术角度来看，该项目依然存在非常大的提升空间。在未来，我们希望能从以下几点对项目进行改进和完善：

- 分支预测：目前处理器使用的分支预测比较简单，未来可以结合更先进的分支预测算法，以提升预测正确率；
- 多端口高速缓存：目前数据高速缓存仅支持一个访问端口，而处理器中经常出现读写同时发生的情况，当前设计会在这种情况下损失一定效率，未来可以完善高速缓存内部调度算法，以支持多端口访问；

- 流片：目前的设计距离流片要求还有一定距离，未来我们将继续完善设计，将各部分设计都向流片方向进行努力，力争在未来能够流片成功。

参 考 文 献

[1] SMITH J E, SOHI G S. The microarchitecture of superscalar processors[J/OL]. Proceedings of the IEEE, 1995, 83(12): 1609-1624. DOI: 10.1109/5.476078.

[2] CHATZOPOULOS O, PAPADIMITRIOU G, et al. Towards accurate performance modeling of risc-v designs[J/OL]. CARRV '21, Co-located with ISCA 2021, 2021. DOI: 10.48550/arXiv.2106.09991.

[3] SILVA JUNIOR F C, SILVA I S, JACOBI R P. Evaluating the performance, energy and area tradeoffs of athena in superscalar processors[J/OL]. 2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI), 2021: 1-6. DOI: 10.1109/SBCCI53441.2021.9529979.

[4] 叶笑春, 李文明, 张洋, 等. 高通量众核处理器设计[J/OL]. 数据与计算发展前沿, 2020, 2(1): 70. http://www.jfdc.cnic.cn/CN/abstract/article_34.shtml. DOI: 10.11871/jfdc.issn.2096-742X.2020.01.006.

[5] AUTHORS V. A survey on deep learning hardware accelerators for heterogeneous hpc platforms[A/OL]. 2023.

[6] HENNESSY J L, PATTERSON D A. 计算机体系结构：量化研究方法[M]. 北京: 人民邮电出版社, 2022.

[7] 龙芯中科技术股份有限公司芯片研发部. 龙芯架构 32 位精简版参考手册[S]. 北京: 龙芯中科技术股份有限公司, 2023, v1.03.

[8] 姚永斌. 超标量处理器设计[M]. 北京: 清华大学出版社, 2014.

[9] 汪文祥, 邢金璋. CPU 设计实战[M]. 北京: 机械工业出版社, 2021.

[10] ASAAD S, BELLOFATTO R, BREZZO B, et al. A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation [C/OL]//FPGA 2012. 2012. DOI: 10.1145/2145694.2145725.

[11] SCHNARR E. Fastsim: Memoizing micro-architecture simulation[D/OL]. University of Wisconsin-Madison, 2000. https://pages.cs.wisc.edu/~wwt/fastsim/.

[12] VERIPOOL. Verilator user's guide[M/OL]. Ongoing. https://verilator.org/guide/latest/.

[13] WANG D S. A prototype of quantum von neumann architecture[A/OL]. 2021.

[14] LOONGSONEDU. Chiplab[CP/OL]. Gitee. https://gitee.com/loongson-edu/chi

plab.

[15] PROJECTN N. Nemu (nju emulator)[EB/OL]. Ongoing. https://github.com/NJU-ProjectN/nemu.

[16] MOHAN V, RAJENDRAN G, et al. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills[A/OL]. 2023.

[17] MITTAL S. A survey of techniques for dynamic branch prediction[J/OL]. Concurrency and computation: practice and experience, 2018, 31(1): 3-4. DOI: 10.1002/cpe.4666.

[18] CUTRESS I. Analyzing falkors microarchitecture: A deep dive into qualcomms centriq 2400 for windows server and linux[R/OL]. 2017. https://www.anandtech.com/show/11737/analyzing-falkors-microarchitecture-a-deep-dive-into-qualcomms-centriq-2400-for-windows-server-and-linux.

[19] BALASUBRAMANIAN R, et al. Vespa: Vipt enhancements for superpage accesses[A/OL]. 2017.

[20] CONTRIBUTORS P. Chiplab[EB/OL]. Ongoing. https://gitee.com/OSCPU/ChipLab.

[21] CORPORATION L. Loongarch elf abi specification[M/OL]. Ongoing. https://loongson.github.io/LoongArch-Documentation/loongarch-elf-abi-spec.html.

[22] AUTHORS V. Performance and power analysis for high-performance computation benchmarks[J/OL]. Central European Journal of Computer Science, 2013. DOI: 10.2478/s13537-013-0101-5.

[23] MAZIRUI2001. Loongarch32r pipeline[EB/OL]. Ongoing. https://github.com/MaZirui2001/LA32R-pipeline-scala.

[24] GROUP S. Soc group, university of science and technology of china[EB/OL]. http://soc.ustc.edu.cn/.

致　谢

在我的研究和学习期间，我有幸得到了几位优秀导师和同学的悉心教导和帮助。他们的支持和指导不仅丰富了我的学术知识，更激励了我的科研热情和动力。

首先，我要感谢我的导师，中国科学技术大学的卢建良老师。他深厚的学术功底、严谨的工作态度和敏锐的科学洞察力让我受益匪浅。卢老师不仅在学术研究方面给予了我重要的指导，还在实验操作和项目实施方面提供了无微不至的帮助。他耐心细致的讲解和实用有效的建议，使我在科研道路上不断前行。

其次，我要感谢中国科学院计算技术研究所的叶笑春老师。叶老师在学术研究中展现出的专业素养和创新精神，使我深受启发。叶老师对待科学研究的认真态度和独到见解，深深地影响了我的学术思维方式和研究方法。感谢叶老师在我科研过程中给予的宝贵建议和持续支持。

同时，我要感谢中科睿芯的吴海彬老师。吴老师在科研项目中的指导和帮助，使我在项目设计和实现过程中获得了许多宝贵的经验。他在科学研究中的严谨态度和务实精神，对我有着深远的影响。感谢吴老师一直以来对我的热情帮助和悉心教导。

此外，我要特别感谢中国科学技术大学的张子辰同学和徐航宇同学。在我的项目设计和研究过程中，他们提出了许多宝贵的意见和建议。他们的热心帮助和合作精神，使我们的研究工作更加顺利和高效。感谢他们在我科研道路上的支持和陪伴。

在此，我还要感谢所有在我研究学习期间给予我帮助和支持的朋友、同事和家人。你们的鼓励和支持，是我不断前进的重要动力。

再次衷心感谢所有帮助过我的老师和同学。你们的指导和帮助，使我在科研道路上不断成长和进步。

2024 年 5 月