温州大學
WENZHOU UNIVERSITY

本科毕业设计 (论文)

基于 RISC-V 的处理器仿真设计

学　　院: 计算机与人工智能学院

班　　级: 21 网工 1 班

专　　业: 网络工程

学　　号: 21211835111

姓　　名: 卢宇鑫

指导教师: 金可仲

指导单位: 计算机与人工智能学院

完成时间: 2025 年 4 月 16 日

摘要

本文提出了一种基于 RISC-V 架构的处理器仿真设计方法，专注于 RV32E 指令集的高效实现。研究从处理器核心的内部结构出发，系统地介绍了取指单元、译码单元、计算单元、访存单元和写回单元的设计细节。通过合理连接这些关键单元，构建了一个简单的单周期处理器架构。在此基础上，引入握手信号以优化时序逻辑，进一步将单周期处理器升级为多周期处理器，显著提升了处理器的时序灵活性与性能表现。

进一步地，本文采用 ARM 的 AXI4 总线协议，实现了处理器与外部设备的高效连接，并成功将其集成到片上系统（SoC）中。为了进一步优化处理器性能，处理器引入了流水线技术和高速缓存技术，显著提升了处理器的并行处理能力和内存访问速度，使其能够更好地满足现代计算任务的需求。

在验证环节，本文详细探讨了处理器的模拟仿真过程。通过使用开源工具 Verilator 将 RTL 代码编译为 C++ 程序并进行仿真运行，确保了仿真的高效性和灵活性。同时，研究使用一种高效的差分测试方法（Differential testing），通过与行为模拟器的状态比对，实现了测试的高效性和准确性，从而确保了设计的高效性和可靠性。

综上所述，本文的研究为 RISC-V 架构处理器的设计与验证提供了一种系统化、高效化的方法，具有重要的理论意义和实践价值，为后续的处理器设计与优化工作奠定了坚实的基础。

关键词： RISC-V；体系结构；性能优化；设计方法

Abstract

This paper presents a processor simulation design method based on the RISC-V architecture, focusing on the efficient implementation of the RV32E instruction set. Starting from the internal structure of the processor core, the study systematically introduces the design details of the fetch unit, decode unit, execution unit, memory access unit, and write-back unit. By properly connecting these key units, a simple single-cycle processor architecture is constructed. On this basis, handshake signals are introduced to optimize timing logic, further upgrading the single-cycle processor to a multi-cycle processor, which significantly enhances the processor's timing flexibility and performance.

Furthermore, this paper employs ARM's AXI4 bus protocol to achieve efficient connections between the processor and external devices, successfully integrating it into a System on Chip (SoC). To further optimize the processor's performance, pipeline and high-speed cache technologies are introduced, which significantly improve the processor's parallel processing capabilities and memory access speed, enabling it to better meet the needs of modern computing tasks.

In the verification section, this paper thoroughly explores the simulation process of the processor. By using the open-source tool Verilator to compile RTL code into C++ programs and run simulations, the efficiency and flexibility of the simulation are ensured. Additionally, an efficient differential testing method is employed, comparing the states with those of a behavioral simulator to achieve high efficiency and accuracy in testing, thereby ensuring the efficiency and reliability of the design. In summary, this study provides a systematic and efficient method for the design and verification of RISC-V architecture processors, which holds significant theoretical and practical value and lays a solid foundation for subsequent processor design and optimization work.

Key Words: RISC-V; Architecture; Performance Optimization; Design Approach

目录

图目录

表目录

第 1 章 绪论

1.1 研究背景

芯片被誉为现代工业的掌上明珠，是信息时代的基石。自 1959 年世界上第一颗芯片诞生以来[1]，芯片技术以惊人的速度发展，推动了从个人计算机到智能手机、从数据中心到物联网设备的全面革新。半导体行业也因此成为当今世界最具战略意义和经济价值的行业之一。经过半个多世纪的发展，全球处理器架构市场逐渐形成了两大主流阵营：面向高性能计算的 X86 架构和面向嵌入式系统的 ARM 架构。然而，随着技术的不断演进和应用场景的多样化，现有的体系结构逐渐暴露出诸多问题：

1) **复杂指令集架构（CISC）的效率问题**：以 X86 为代表的复杂指令集架构虽然功能强大，但其指令集冗长且复杂，导致指令执行效率较低。为了实现复杂的指令功能，处理器需要集成更多的晶体管和电路，这不仅增加了芯片的设计难度和制造成本，还显著提高了功耗和发热量，限制了其在低功耗场景中的应用。

2) **闭源与授权限制**：X86 和 ARM 架构均属于闭源架构，其核心技术和指令集受到严格的版权保护。使用这些架构需要获得相应公司的授权许可，这不仅增加了研发成本，还限制了中小企业和新兴市场的准入，阻碍了技术的普及和创新。

3) **市场垄断与供应商锁定**：由于 X86 和 ARM 架构在各自领域的主导地位，相关技术的使用高度依赖于特定供应商的支持。这种市场垄断和供应商锁定的局面导致技术更新缓慢，用户选择受限，进一步抑制了行业的发展活力。

为了解决现有体系结构存在的问题，并顺应现代计算机体系结构设计的发展趋势，RISC-V（开源精简指令集架构）应运而生。RISC-V 以其开源、免费、开放和自由的特性，迅速成为全球学术界和工业界关注的焦点。任何个人或组织都可以自由使用、修改和分发 RISC-V 的设计，这为处理器架构的创新和普及提供了前所未有的机会。

RISC-V 的起源可以追溯到 20 世纪 80 年代初，当时加州大学伯克利分校的 David Patterson 教授和斯坦福大学的 John Hennessy 教授分别提出了精简指令集计算（RISC）理念[2]。RISC 的核心思想是通过简化指令集，使处理器设计更加高效、易于实现和优化。基于这一理念，伯克利分校开发了 RISC-I 和 RISC-II 原型机，成功验证了 RISC 架构的可行性和优越性。2010 年，加州大学伯克利分校的 Krste Asanović 教授及其团队启动了 RISC-V 项目，旨在设计一种全新的、开放的指令集架构，以满足现代计算的多样化需求[3]。2014 年，伯克利团队发布了 RISC-V 的初始规范，包括 32 位和 64 位的基本指令集[4]，这一规范的发布标志着 RISC-V 正式进入公众视野。

2015 年，RISC-V 基金会（RISC-V Foundation）正式成立，吸引了包括谷歌、英特尔、英伟达等全球顶尖科技公司以及众多学术机构和研究组织的加入。2017 年，首批基于 RISC-V 的商用芯片发布，展现了 RISC-V 在实际应用中的巨大潜力。2020 年，为了避免潜在的政治和法律风险，RISC-V 基金会迁至瑞士，并更名为 RISC-V International，进一步提升了 RISC-V 的国际化水平和开放性。

如今，RISC-V 已成为全球范围内最具活力的开源指令集架构之一，其应用领域涵盖

嵌入式系统、物联网、高性能计算、人工智能、数据中心等多个领域。RISC-V 的崛起不仅为处理器架构的设计和实现提供了新的思路，也为全球半导体行业注入了新的活力，推动了技术的民主化和创新生态的繁荣发展。

更重要的是，在中国的芯片产业中，主流 CPU 架构一直受西方国家的制约，整体的处理器行业生态与国际先进水平依旧存在不小的差距，RISC-V 架构的出现为中国开源芯片的发展提供了新的机遇。2021 年，开源首次被写入《中华人民共和国国民经济和社会发展第十四个五年规划和 2035 年远景目标纲要》[5]，全国各界都投入到对 RISC-V 开源生态的建设，对 RISC-V 的支持大幅度上升。例如，由多家 RISC-V 领域重点企业、研究机构、行业协会发起成立的 “中国 RISC-V 产业联盟”；由网信办、中科院等多个国家部委支持成立的 “中国开放指令生态 RISC-V 联盟”；由加州伯克利大学和清华大学合作发起成立的 “RISC-V 国际开源实验室”；由中科院、多家行业龙头企业和顶尖科研单位发起成立的 “北京开源芯片研究院” 等。以上例子说明在国家的政策支持下，各行各业的积极参与下，中国的 RISC-V 开源生态环境正在以不可阻挡的速度发展。

1.2 国内外研究现状

近年来，RISC-V 技术以其开源、模块化和可扩展的特性，在全球范围内引发了广泛关注，并取得了令人瞩目的研究成果。从高性能计算到嵌入式系统，从学术研究到商业应用，RISC-V 正在迅速崛起，成为处理器架构领域的重要力量。

2019 年，西部数据公司（Western Digital）推出了 SweRV 核心，这是一款高性能的 RISC-V 处理器核心，专为数据中心和存储应用设计，SweRV 的发布不仅展示了 RISC-V 在高性能计算领域的潜力，还标志着 RISC-V 从学术研究向工业应用的重大跨越[6]。2021 年，阿里巴巴旗下的平头哥半导体发布了玄铁 907 处理器，这是一款基于 RISC-V 架构的高性能处理器，已成功授权给多家企业使用，进一步推动了 RISC-V 的商业化进程[7]。2024 年，中国科学院计算技术研究所推出了 “香山” 昆明湖架构 V2，这是一款开源的高性能 RISC-V 处理器，其卓越的性能和创新的设计再次证明了 RISC-V 在技术创新上的巨大潜力[8]。

在系统级集成方面，Vedran Dakić 等人提出了一种异构 RISC-V SoC 设计，该设计集成了高性能的乱序核心、高能效的顺序核心以及专用加速器，充分展现了 RISC-V 在灵活性和可扩展性方面的优势[9]。此外，Koch 等人开发了针对 RISC-V 的 FPGA 框架 FABulous，为 RISC-V 处理器的快速原型设计和验证提供了高效的工具支持[10]。在低功耗设计方面，邓等人设计了一款超低功耗 RISC-V 流水线结构处理器，适用于物联网等低功耗场景[11]。在设计方法学方面，钟等人提出了一种软硬件联合验证的设计方法，通过结合 Verilator 软件仿真和 FPGA 硬件验证，显著提高了 RISC-V 处理器的开发效率和可靠性[12]。在总线设计方面，郝等人采用 ARM 公司提出的 AHB 总线协议，成功实现了系统总线的设计，不仅保持了处理器的高性能，还显著减小了芯片的流片面积[13]。针对浮点数运算的挑战，潘等人提出了一种优化的浮点运算单元（FPU）设计，通过改进算法和硬件结构，显著提升了浮点数运算的效率和精度[14]。

1.3 技术对比

在现代处理器架构中，RISC（精简指令集计算）和 CISC（复杂指令集计算）是两种主要的设计理念。以 RISC-V、X86 和 ARM 为例，RISC-V 作为 RISC 架构的代表，展现出显著的优势，尤其是在开源性、模块化设计、可扩展性、功耗和性能等方面。

1) 在开源性方面，RISC-V 的完全开源特性使其在灵活性和可扩展性方面远超 X86 和 ARM。这种开源性不仅降低了开发成本，还允许开发者根据具体需求进行定制和优化，从而推动了创新和多样化应用的开发。相比之下，X86 和 ARM 均为非开源架构，需要授权才能使用，这在一定程度上限制了其在特定领域的应用。

2) 在处理器设计方面，RISC-V 的模块化设计和高可扩展性是其显著优势。它允许开发者根据不同的应用场景灵活选择和扩展指令集，从而更好地适应从嵌入式系统到高性能计算的多样化需求。而 X86 架构由于其复杂指令集和非模块化设计，在可扩展性方面表现较差，难以满足新兴应用的快速变化需求。

3) 在功耗方面，RISC-V 的设计理念使其在低功耗场景中表现出色。其简洁的指令集和高效的执行效率使得处理器能够在较低的功耗下运行，这对于移动设备和物联网应用尤为重要。相比之下，X86 架构由于其复杂的设计和较高的功耗，在移动和嵌入式领域存在明显劣势，尽管其在高性能计算中仍具有一定的优势。

4) 在性能方面，RISC-V 通过高效的指令执行和流水线设计，能够实现高性能处理。虽然 X86 和 ARM 在某些应用场景中也能提供高性能，但 RISC-V 的高效设计使其在功耗和性能的平衡上更具优势。此外，RISC-V 的开源性和模块化设计进一步增强了其在性能优化方面的潜力。

5) 在指令数量方面，与 X86 架构的指令集相比，RISC-V 的指令集格式更为简洁且高效。RISC-V 仅定义了六种指令格式，且每条指令长度固定为 32 位或 64 位，这极大地降低了指令译码时的复杂度和开销。相比之下，X86 架构由于其 CISC 的特性，指令长度不定，每次取指需按照最大指令字长读取，并在译码阶段进行分割，这无疑增加了指令处理的复杂度。此外，RV32I 的基础指令数量仅为 47 条，即使加上乘除法扩展（RV32M），指令总数也不超过 60 条。如图1-1所示，X86 架构在发布初期有 80 条指令，到 2015 年，其指令数量已增长至 1338 条，增加了 16 倍。

综上所述，RISC-V 作为 RISC 架构的代表，在开源性、模块化设计、可扩展性、功耗和性能等方面展现出显著的优势，使其在现代处理器架构中具有广阔的应用前景。

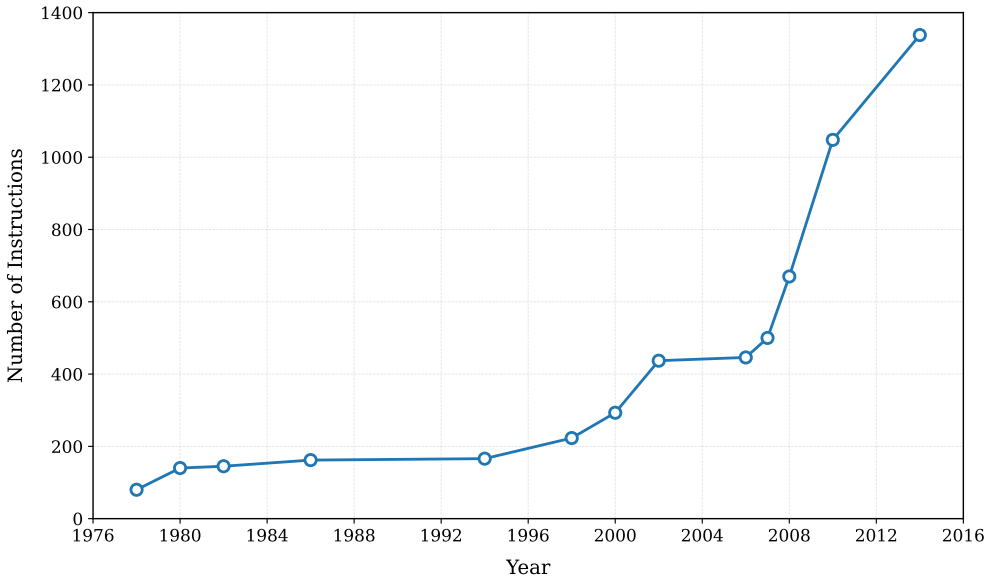


图 1-1 x86 指令集发展历程（1978-2014）

表 1-1 指令集对比

特性	RISC-V	X86	ARM
类型	RISC	CISC	RISC
开源性	开源	闭源	闭源
模块化	支持	不支持	不支持
扩展性	高	低	低
功耗	低	高	低
性能	高	高	高

1.4 本章小结

本章从研究背景、国内外研究现状以及技术对比三个方面对 RISC-V 架构进行了全面介绍。首先，通过对现有处理器架构（如 X86 和 ARM）的分析，指出了复杂指令集架构（CISC）的效率问题、闭源与授权限制以及市场垄断与供应商锁定等问题。这些问题的存在促使了开源精简指令集架构（RISC-V）的诞生。RISC-V 以其开源、免费、开放和自由的特性，迅速成为全球学术界和工业界关注的焦点，为处理器架构的创新和普及提供了前所未有的机会。

在国内外研究现状方面，近年来 RISC-V 技术取得了显著的进展。从嵌入式系统到高性能计算，从学术研究到商业应用，RISC-V 正在迅速崛起，成为处理器架构领域的重要力量。众多企业和研究机构纷纷推出了基于 RISC-V 的处理器和相关技术，进一步推动了 RISC-V 的商业化进程和技术创新。

在技术对比部分，通过对 RISC-V、X86 和 ARM 三种架构的详细对比，展示了 RISC-V 在开源性、模块化设计、可扩展性、功耗和性能等方面的优势。RISC-V 的完全开源特性降低了开发成本，其模块化设计和高可扩展性使其能够灵活适应多样化的应用场景。在功耗

方面，RISC-V 的设计理念使其在低功耗场景中表现出色，而其高效的指令执行和流水线设计则使其在性能上具有显著优势。

总的来说，RISC-V 具有极高的发展潜力，且逐渐成为 X86 和 ARM 双足鼎立之后的“第三极”。

第 2 章 RISC-V 处理器介绍

2.1 RISC-V 处理器指令集

2.1.1 指令集格式

RISC-V 基础指令集（Base ISA）作为 RISC-V 架构的核心，为处理器提供了基本的指令集框架。如表2-1和图2-1所示，RV32I 包含了六种基本指令类型：

1) **R 型**：两个操作数来自寄存器，用于寄存器间操作；
2) **I 型**：两个操作数来自立即数和寄存器，用于实现立即数和取数操作；
3) **S 型**：两个操作数来自寄存器，一个操作数来自立即数，用于实现访存操作；
4) **B 型**：两个操作数来自寄存器，一个操作数来自立即数，用于实现条件分支操作；
5) **U 型**：一个操作数来自立即数，用于实现长立即数操作；
6) **J 型**：一个操作数来自立即数，用于实现无条件跳转操作。

从指令格式就可以看出 RISC-V 指令集最大的优点——精简。首先，指令的长度都为 32 位，同一种类型的指令格式单一，大幅度减小了译码器的开销以及实现难度。其次，R 型指令提供了三个寄存器，这对于需要三个操作数的指令不需要额外的访存，避免了访存带来的开销。然后，寄存器的编码都在指令的固定位置（rs1 和 rs2），在译码之前就可以先读取寄存器的值读。最后，立即数的符号位被编码至指令的最高位，所以，立即数的符号扩展操作可以与译码操作并行处理。

表 2-1 RISC-V 指令格式介绍

类型	用途	示例指令
R 型（寄存器-寄存器操作）	算术和逻辑运算	add x1, x2, x3
I 型（立即数操作）	加载、立即数操作和跳转	addi x1, x2, 10
S 型（存储操作）	数据从寄存器存储到内存	sw x1, 12(x2)
B 型（条件分支）	条件分支跳转	beq x1, x2, label
U 型（高位立即数操作）	加载高位立即数	lui x1, 0x12345
J 型（无条件跳转）	函数调用或长跳转	jal x1, label

2.1.2 通用寄存器

如图2-2表示，RISC-V 有 32 个寄存器，特殊的是，x0 寄存器硬连线为 0，可替代约 15% 的指令操作，而 X86 需要显式 XOR 清零。为了提升处理器的性能，数据应该尽量存储在寄存器中，但是频繁的恢复和保存寄存器需要不断地访问内存，会带来不小的开销。为了避免这种情况，RISC-V 的处理方案是设置临时寄存器（t0-6）和保存寄存器（s0-11）。临时寄存器的值不需要保存至内存，而保存寄存器的值需要保存至内存。

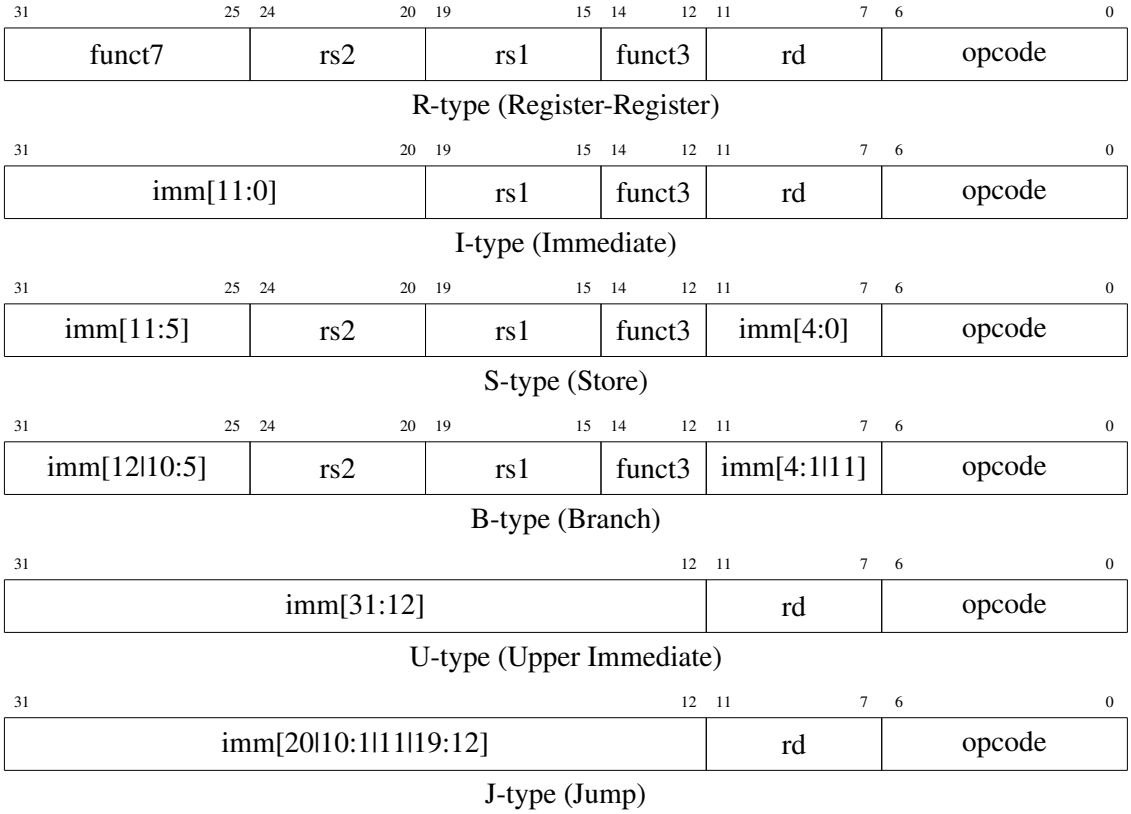


图 2-1 RV32I 指令集格式

相比 X86 的 16 个寄存器，RISC-V 架构的寄存器数量多一倍，适当地提升寄存器的数量，处理器可以充分地调度更多的寄存器，以至于加快程序编译和运行的速度。但是，寄存器并不是越多越好，由于其制造工艺和其特殊性导致了寄存器的成本昂贵，所以更多的寄存器会导致更高的成本和硬件复杂度，指令集也需要更多的比特位来对寄存器进行编码，一定程度上压缩了其余字段的编码空间，会提升编码和译码的复杂度。

2.1.3 指令集扩展

RISC-V 指令集的另一个显著特点是模块化设计。RV32I 作为基础 ISA，虽然只定义了 47 条指令，但是足以支持运行基本的软件，其稳定性为开发者提供了可靠的指令集基础。模块化设计允许开发者根据具体需求选择特定的扩展，例如：

1) **RV32M**：支持乘除法扩展，从基本指令集分离出来的一个单独标准，需要设计对应的乘除法单元，适用于嵌入式系统、低功耗微控制器、高效处理整数运算的场景；

2) **RV32A**：支持原子指令扩展，对共享内存的数据进行操作的一种方式，能够保证多线程并发执行的一致性，适用于多核处理器、操作系统内核、实时系统的场景；

3) **RV32F**：支持单精度浮点扩展，新增了浮点寄存器，支持单精度浮点的传输、比较、转换、分类，适用于图形处理、传感器数据处理、轻量级机器学习推理的场景；

4) **RV32D**：支持双精度浮点扩展，将浮点寄存器扩展至 64 位，新增了支持双精度浮点数相关的运算，适用于科学计算、高精度工程仿真、复杂模型训练的场景；

表 2-2 RISC-V 寄存器介绍

寄存器	名称	功能
x0	zero	始终为 0
x1	ra	返回地址
x2	sp	栈指针
x3	gp	全局指针
x4	tp	线程指针
x5	t0	临时寄存器/链接寄存器
x6-7	t1-2	临时寄存器
x8	fp / s0	帧指针/保存寄存器
x9	s1	保存寄存器
x10-11	a0-1	函数参数/返回值
x12-17	a2-7	函数参数
x18-27	s2-11	保存寄存器
x28-31	t3-6	临时寄存器

5) **RV32V**: 支持向量扩展，用于数据并行执行功能，该扩展引入了新的向量寄存器和操作指令，该扩展使 RISC-V 对于数据处理产生了质的提升，同时也保持了指令的简洁度。

这种灵活的扩展机制使得 RISC-V 能够适应从嵌入式系统到高性能计算的多样化应用场景，如：物联网设备只需 RV32M + RV32F，而科学计算需要 RV32F + RV32D。这种设计理念为芯片设计者、软件开发者和终端用户带来了多方面的好处。

2.1.4 特权指令集

RISC-V 特权指令集（Privileged Instruction Set）定义了处理器在系统级操作中的行为规范，包括中断处理、内存管理、特权模式切换等核心功能。它是操作系统、监控程序（Hypervisor）和底层固件开发的基础。如2-3所示，RISC-V 目前有以下 4 种特权模式：U 模式（User, 用户模式），S 模式（Supervisor, 管理模式），H 模式（Hypervisor，监视模式），M 模式（Machine,M 模式）。RISC-V 通过 CSR（控制状态寄存器）来更改权限模式。

表 2-3 RISC-V 特权模式

等级	CSR 编码	模式
0	00	U
1	01	S
2	10	H
3	11	M

1) **U 模式**：用户模式是最低级的特权模式，只支持最低级别的权限操作，运行应用程序，禁止直接访问硬件或特权指令；

2) **S 模式**：管理模式拥有次高特权级，可以操作计算机中的敏感资源，用于管理虚拟内存、进程调度和系统调用；

3) **H 模式**：监视模式拥有比 S 模式更高的特权级，可以用于管理跨机器的资源，如管理多个虚拟机；

4) **M 模式**：机器模式拥有最高的特权级，可以执行任何机器操作，能访问所有物理资源（如定时器、中断控制器、PMP）的模式。

2.2 RISC-V 汇编

由于 C 程序无法直接运行在计算机上，所以需要将源代码翻译成为机器语言，才能在计算机上运行，该过程涉及编译、汇编、链接、载入，图2-2展示了整个翻译过程。

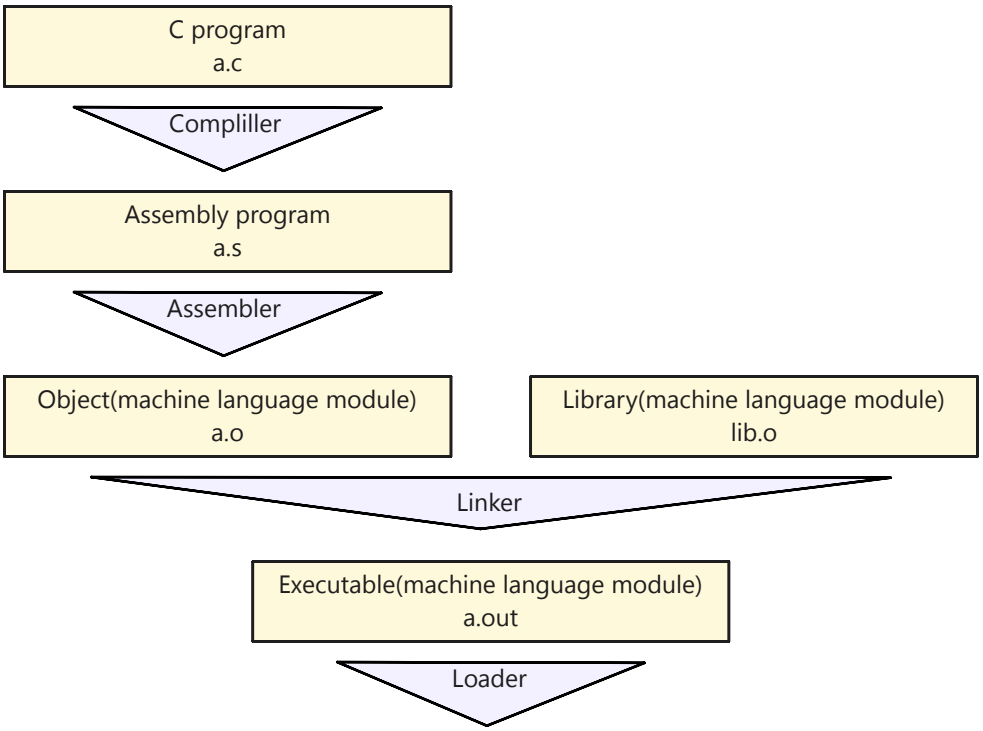


图 2-2 C 源代码的翻译过程

2.2.1 编译器

编译器是一类程序，用于将高级语言转换为机器语言指令集。如表2-4所示，RISC-V 编译器支持多种 ABI（应用程序二进制接口），具体使用的接口取决于处理器所支持的扩展（例如 F、D 扩展）。其中，ilp32 表示在 C 语言中 int、long 和 pointer 均为 32 位。后缀则表示浮点参数的传递方式：ilp32 表示浮点参数通过整数寄存器传递；ilp32f 表示单精度浮点参数通过浮点寄存器传递；ilp32d 表示双精度浮点参数通过浮点寄存器传递。

因此，编译 RV32I 指令集的代码时，ABI 选项必须为 ilp32（GCC 参数为：-march=rv32i -mabi=ilp32）。然而，在 RISC-V 的约定中，浮点指令的函数调用不一定需要使用浮点寄存器。这意味着，当编译支持 RV32IFD 指令集的代码时，ABI 选项可以是 ilp32、ilp32f 或 ilp32d 中的任意一种。这种灵活性使得开发者能够根据具体的应用场景和性能需求选择最合适的 ABI 配置。

表 2-4 ABI 常见名称

ABI 名称	数据类型	参数传递
ilp32	整型（32 位）	浮点参数在整数寄存器中传递
ilp32f	单精度浮点（32 位）	单精度浮点参数在浮点寄存器中传递
ilp32d	双精度浮点（64 位）	双精度浮点参数在浮点寄存器中传递

在经典的 C 语言 Hello Word 程序编译中，C 程序2-3通过编译器编译后得到汇编程序2-4。

```c
#include <stdio.h>
int main() {
  printf ("Hello World\n");
  return 0;
}
```

图 2-3 C 语言 Hello World 源代码（hello.c）

```
.text                       # 指示符：进入代码节
.align 2                    # 指示符：将代码按 2^2 字节对齐
.globl main                 # 指示符：声明全局符号 main
main:                       # main 的开始符号
    addi sp,sp,-16          # 分配栈帧
    sw   ra,12(sp)          # 保存返回地址
    lui  a0,%hi(string1)    # 计算 string1 的地址
    addi a0,a0,%lo(string1)
    call printf             # 调用 printf 函数
    lw   ra,12(sp)          # 恢复返回地址
    addi sp,sp,16           # 释放栈帧
    li   a0,0               # 装入返回值 0
    ret                     # 返回
    .section .rodata        # 指示符：进入只读数据节
    .balign 4               # 指示符：将数据按 4 字节对齐
string1:                    # 第一个字符串符号
    .string "Hello World\n" # 指示符：以空字符结尾的字符串
```

图 2-4 Hello world 汇编程序（hello.s）

2.2.2 汇编器

汇编器是将汇编程序翻译成为计算机能够运行的机器语言的一类程序。其在生成目标代码的过程中会替换一些“伪指令”，“伪指令”是基于配置常规指令实现的，“伪指令”编译器开发者和汇编程序员十分有用。为了简化汇编代码，RISC-V 的汇编器有 60 种伪指令。比如：ret 是 RISC-V 中最常见的伪指令，实现从子过程返回的功能，但实际的指令是 jalr x0, x1, 0。当出现伪指令时，汇编器会将其替换为实际的指令。x0 寄存器的存在为许多伪指令提供了方便的操作，如 j、ret、beqz 等指令，这在很大程度上化简了 RISC-V 指令集，表2-5提供了依赖 x0 的伪指令和与其对应的实际指令。

表 2-5 与零寄存器相关的 RISC-V 伪指令（续）

伪指令	实际指令	含义
nop	addi x0, x0, 0	空操作
neg rd, rs	sub rd, x0, rs	取负
negw rd, rs	subw rd, x0, rs	取负字
snez rd, rs	sltu rd, x0, rs	不等于零时置位
sltz rd, rs	slt rd, rs, x0	小于零时置位
sgtz rd, rs	slt rd, x0, rs	大于零时置位
beqz rs, offset	beq rs, x0, offset	等于零时分支
bnez rs, offset	bne rs, x0, offset	不等于零时分支
blez rs, offset	bge x0, rs, offset	小于等于零时分支
bgez rs, offset	bge rs, x0, offset	大于等于零时分支
bltz rs, offset	blt rs, x0, offset	小于零时分支
bgtz rs, offset	blt x0, rs, offset	大于零时分支
j offset	jal x0, offset	跳转
jr rs	jalr x0, rs, 0	寄存器跳转
ret	jalr x0, x1, 0	从子过程返回
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	尾调用
rdinstret[h] rd	csrrs rd, instret[h], x0	读指令计数器
rdcycle[h] rd	csrrs rd, cycle[h], x0	读周期计数器
rdtime[h] rd	csrrs rd, time[h], x0	读实时时钟
csrr rd, csr	csrrs rd, csr, x0	CSR 读
csrw csr, rs	csrrw x0, csr, rs	CSR 写
csrs csr, rs	csrrs x0, csr, rs	CSR 置位

续下页

表 2-5 与零寄存器相关的 RISC-V 伪指令（续）

伪指令	实际指令	含义
csrc csr, rs	csrrc x0, csr, rs	CSR 清位
csrwi csr, imm	csrrwi x0, csr, imm	CSR 写立即数
csrsi csr, imm	csrrsi x0, csr, imm	CSR 置位立即数
csrci csr, imm	csrrci x0, csr, imm	CSR 清位立即数
frcsr rd	csrrs rd, fcsr, x0	读浮点 CSR
fscsr rs	csrrw x0, fcsr, rs	写浮点 CSR
frrm rd	csrrs rd, frm, x0	读舍入模式
fsrm rs	csrrw x0, frm, rs	写舍入模式
frflags rd	csrrs rd, fflags, x0	读异常标志
fsflags rs	csrrw x0, fflags, rs	写异常标志

将 Hello Word 汇编程序通过汇编器译器翻译后，得到机器语言的 Hello World 程序2-5。

```
00000000 <main>:
   0: ff010113  addi    sp,sp,-16
   4: 00112623 sw       ra,12(sp)
   8: 00000537 lui      a0,0x0
   c: 00050513 mv       a0,a0
  10: 00000097 auipc    ra,0x0
  14: 000080e7  jalr    ra
  18: 00c12083 lw       ra,12(sp)
  1c: 01010113 addi     sp,sp,16
  20: 00000513 li       a0,0
  24: 00008067 ret
```

图 2-5 机器语言的 Hello world 程序（hello.o）

2.2.3 链接器

链接器是一种程序，它将一个或多个目标文件或库函数链接成一个可执行目标文件。这种机制的优势在于，如果只修改了一个文件，就无需重新编译所有源代码，从而提高了开发效率。在 Unix 系统中，链接器通常处理后缀为.o 的目标文件，并生成后缀为.a 的库文件；而在 MS-DOS 系统中，链接器的输入文件后缀为.OBJ 或.LIB，输出文件后缀为.EXE。

每个目标文件除了包含指令外，还包含一张符号表。这张符号表记录了程序中所有需要在链接过程中确定地址的符号，包括数据符号和代码符号。由于 32 位指令的长度限制，一个 32 位地址无法直接嵌入到指令中，因此链接器通常需要为每个符号调整两条 RV32I

指令以容纳完整的地址信息。图2-6展示了图2-5中的目标文件经过链接后生成的 a.out 文件。

链接器在工作时还会检查程序的 ABI 是否与所有链接库匹配。尽管编译器支持多种 ABI 和 ISA 扩展的组合，但实际在机器上可能只安装了特定组合的库。因此，一个常见问题是在未安装兼容库的情况下尝试链接程序。在这种情况下，链接器通常不会输出详细的诊断信息，而是简单地尝试进行链接，随后提示不兼容。这种情况通常发生在交叉编译环境中，即在一台计算机上为另一台计算机编译程序时。为了避免这种问题，确保所有参与链接的库与目标平台的 ABI 兼容是非常重要的。

```
000101b0 <main>:
    101b0: ff010113  addi    sp,sp,-16
    101b4: 00112623 sw       ra,12(sp)
    101b8: 00021537 lui      a0,0x21
    101bc: a1050513 addi     a0,a0,-1520 # 20a10 <string1>
    101c0: 288000ef  jal     ra,10450 <printf>
    101c4: 00c12083 lw       ra,12(sp)
    101c8: 01010113 addi     sp,sp,16
    101cc: 00000513 li       a0,0
    101d0: 00008067 ret
```

图 2-6 链接后的 Hello world 程序（在 Unix 系统中改名为 a.out）

2.2.4 加载器

图2-6展示了一个典型的可执行文件，该文件存储在计算机的存储设备中，由操作系统加载并执行。当运行一个程序时，加载器会将其加载到内存中，并跳转到程序的起始地址。在现代操作系统中，加载器负责将程序从存储设备加载到内存，并启动程序的执行。

动态链接程序的加载过程较为复杂，对于动态链接程序，操作系统并不会直接启动程序，而是先启动动态链接器。动态链接器负责加载程序及其所需的动态链接库，处理所有的首次外部函数调用，将函数加载到内存，并修改程序，使其指向正确的函数地址。这种机制使得程序可以共享库函数，减少了内存占用，并提高了系统的灵活性和效率。

2.3 本章小结

本章全面介绍了 RISC-V ISA 的基础内容。首先，深入探讨了指令集的基本格式，阐述了其精简和高效的设计理念。接着，详细讲解了通用寄存器的架构和功能，强调了其在数据存储和运算中的关键作用。此外，对指令扩展进行了深入的介绍，展示了几种经典的扩展指令集，通过模块化设计增强了处理器的功能。特权指令集作为系统级操作的核心，也在本章中得到了详细的阐述，包括了四种特权模式和其对于处理器的管理功能。

在编译过程方面，本章系统地介绍了从源代码到可执行程序的整个流程。首先，编译器将高级语言转换为汇编语言，这一过程涉及到语法分析、语义检查和代码优化。随后，汇编器将汇编代码转换为机器语言，生成目标文件。链接器则负责将多个目标文件和库函

数整合成一个可执行目标文件，解决了符号引用和地址分配的问题。最后，加载器在程序运行时将其加载到内存中，并确保其正确执行。

第 3 章　RISC-V 处理器设计

本章将深入探讨微架构和指令集架构（ISA）对处理器性能的影响，并结合计算机科学与电子信息领域的知识，采用 RISC-V 开源指令集架构，设计一款单周期微处理器。该设计基于 RV32I 基础指令集，旨在提供一个高效且灵活的处理器架构。

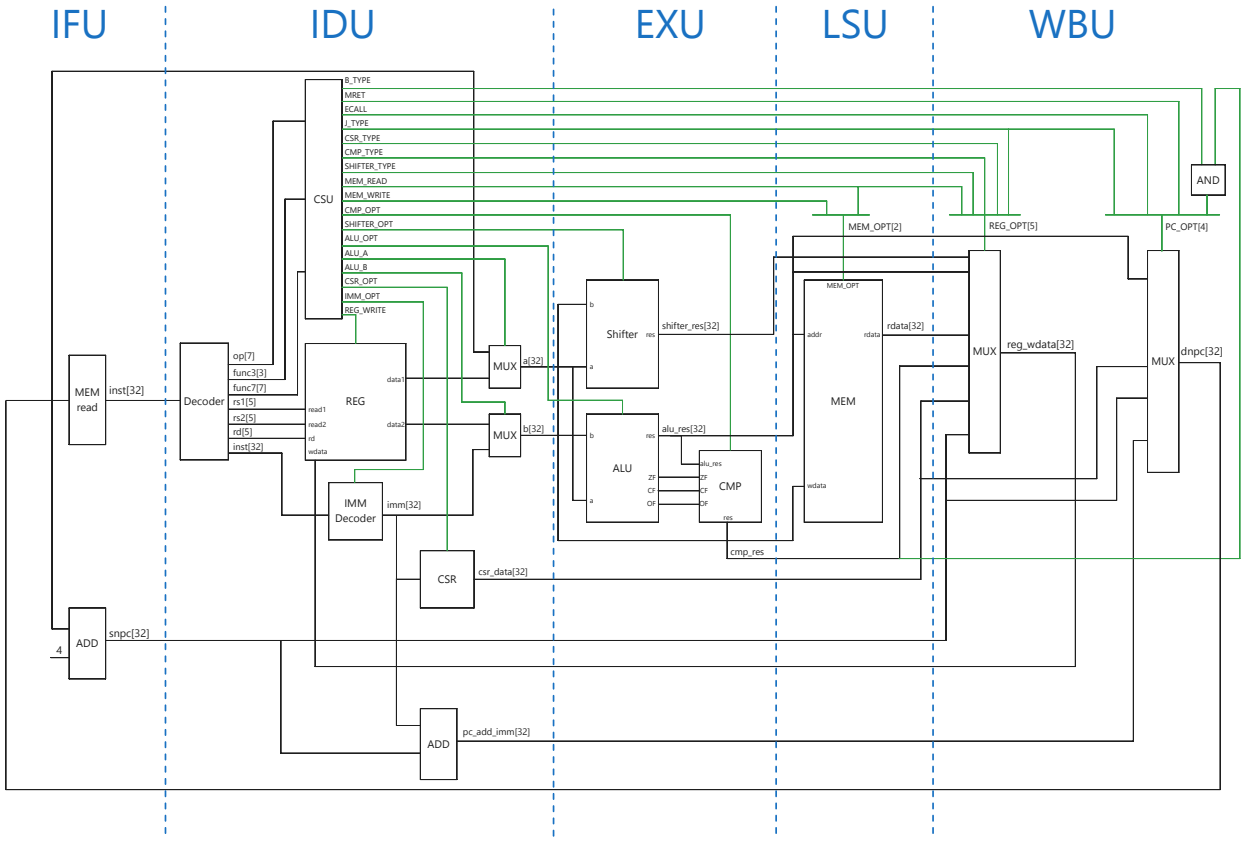


图 3-1 处理器整体架构图

微处理器的整体架构如图3-1所示，主要由五个关键单元组成：IFU（取指单元）、IDU（译码单元）、EXU（执行单元）、LSU（访存单元）和 WBU（写回单元）。以下是各单元的详细数据通路过程：

1) **取指阶段**：取指单元根据 PC 从内存中读取指令，并将指令传递给 IDU。这一过程确保了指令流的连续性，为后续的处理步骤提供了基础。

2) **译码阶段**：IDU 负责将指令译码为多条控制信号。在这一阶段，寄存器的值被读取，同时进行立即数的符号扩展。译码后的控制信号被发送到 EXU、LSU 和 WBU，以协调后续的执行和数据处理。

3) **执行阶段**：EXU 包含三个主要单元：加法器、移位器和比较器。这些单元根据来自 IDU 的控制信号执行算术和逻辑运算。运算结果随后被传递到 EXU 和 WBU，以支持进一步的数据处理和存储。

4) **访存阶段**：LSU 根据来自 IDU 的控制信号和来自 EXU 的运算结果，执行存储器的读写操作。读取的数据被发送到 WBU，以便后续的存储。

5) **写回阶段**：WBU 根据前面单元的信号，将处理结果写回到相应的寄存器中。这一阶段确保了数据的最终存储和状态的更新。

通过这种模块化设计，微处理器能够高效地处理指令流，实现数据的快速通路。这种设计不仅展示了 RISC-V 架构的灵活性和高效性，还提供了一个深入了解计算机系统设计的机会。这种基于单周期设计的微处理器，为后续的多周期、流水线优化提供了十分便捷的基础。

3.1 取指单元（IFU）

取指单元（Instruction Fetch Unit）是单周期 RISC-V 处理器数据通路的起始单元，其核心功能是在每个时钟周期内从指令存储器中读取当前指令并计算下一条指令地址。该单元由程序计数器（PC）、指令存储器（IMEM）和地址计算逻辑构成：PC 寄存器在每个时钟上升沿输出指令地址至 IMEM，IMEM 返回对应的 32 位机器指令；同时，地址计算逻辑通过专用加法器生成 PC+4 地址（默认顺序执行）或分支目标地址（需与后续写回单元配合），并通过多路选择器确定下一周期 PC 值。由于单周期架构在一个时钟周期内完成全部指令操作，取指单元不包含流水线缓冲机制，其指令存储器的访问时间直接决定了处理器最高工作频率。该单元的输出端与译码单元直接相连，形成” 取指-译码-执行” 的完整单周期数据通路。

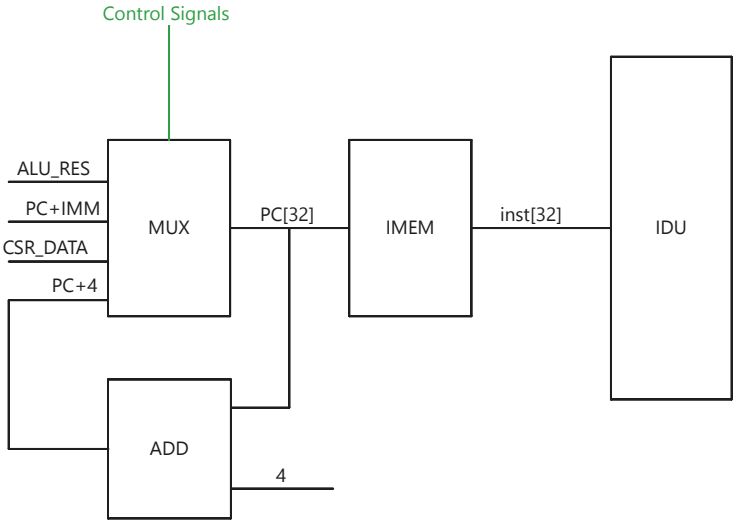


图 3-2 取指单元设计图

如图3-2所示，程序计数器的选择有以下四种情况：

1) **ALU+RE**：这种选择用于实现寄存器跳转，将程序计数器设置为加法器运算结果所在的地址。这种机制允许程序根据寄存器中的值动态调整执行流程，从而实现复杂的控制结构；

2) **PC+IMM**：这种选择用于实现 PC 相对跳转，将程序计数器设置为当前程序计数器与立即数的和所在的地址。这种机制允许程序在代码中进行相对位置的跳转，常用于条件分支和循环结构；

3) **CSR_DATA**：这种选择用于实现异常中断跳转，将程序计数器设置为控制状态寄存器值所在的地址。这种机制在处理异常和中断时非常关键，它允许程序发生异常和中断时跳转到预定义的处理程序，以及中断处理结束时返回程序；

4) **PC+4**：这种选择用于实现静态跳转，将程序计数器设置为当前程序计数器与 4 的和所在的地址。这是程序正常执行时的默认行为，用于顺序执行下一条指令。

这些 PC 选择机制共同确保了程序的灵活执行和高效控制流管理，为处理器在各种复杂场景下的稳定运行提供了基础。

3.2 译码单元（IDU）

译码单元（Instruction Decode Unit）负责将指令中携带的信息提取出来，这些信息被称为控制信号，后续将被传递给执行单元、访存单元和写回单元，用于进行结果选择。译码器的实现相对简单，如图3-3所示，主要由译码器（Decoder）和控制信号单元（CSU）组成。

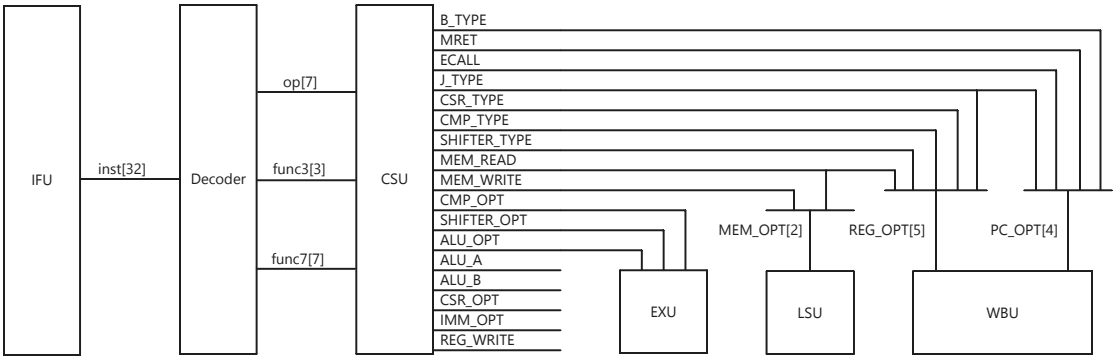


图 3-3 译码单元设计图

3.2.1 译码器

译码器（Decoder）的任务是对接收到的指令进行初始译码，生成操作码（op）、功能码（func3 和 func7）、源寄存器（rs1 和 rs2）、目标寄存器（rd）以及立即数（imm）等关键信息。这些信息是后续处理的基础，其接口描述如表3-1所示。

3.2.2 译码器

控制信号单元（CSU）则负责将操作码（op）、功能码（func3 和 func7）转换为具体的控制信号。这些控制信号将指导处理器的各个模块如何协作以正确执行指令。CSU 生成的控制信号包括 ALU 操作类型、数据通路选择信号等，其接口描述如表3-2所示。

表 3-1 译码器接口描述表

信号	位宽	描述
op	7	标识指令类型
func3	3	进一步细化指令具体操作
func7	7	扩展指令的功能
rs1	5	读寄存器 1
rs2	5	读寄存器 2
rd	5	写寄存器

表 3-2 控制信号单元接口描述表

信号	位宽	描述	下游
REG_WRITE	1	是否将写入寄存器	IDU
IMM_OPT	6	立即数的扩展方式	IDU
CSR_OPT	2	选择 CSR 的值。	IDU
ALU_B	1	选择操作数 B	EXU
ALU_A	1	选择操作数 A	EXU
ALU_OPT	3	加法器的运算类型	EXU
CMP_OPT	4	比较器的运算类型	EXU
SHIFTER_OPT	3	移位器的运算类型	EXU
MEM_OPT	2	存储器的读写类型	LSU
REG_OPT	5	写入寄存器堆的值	WBU
PC_OPT	4	PC 的更新方式	WBU

3.3 执行单元（EXU）

执行单元（Execution Unit）是处理器的核心组件之一，负责对操作数 A 和操作数 B 进行算术和逻辑运算，并将计算结果传递给访存单元和写回单元，以支持对存储器、寄存器和程序计数器的写操作。如图3-4所示，执行单元主要由加法器（ALU）、比较器（CMP）和移位器（Shifter）组成。

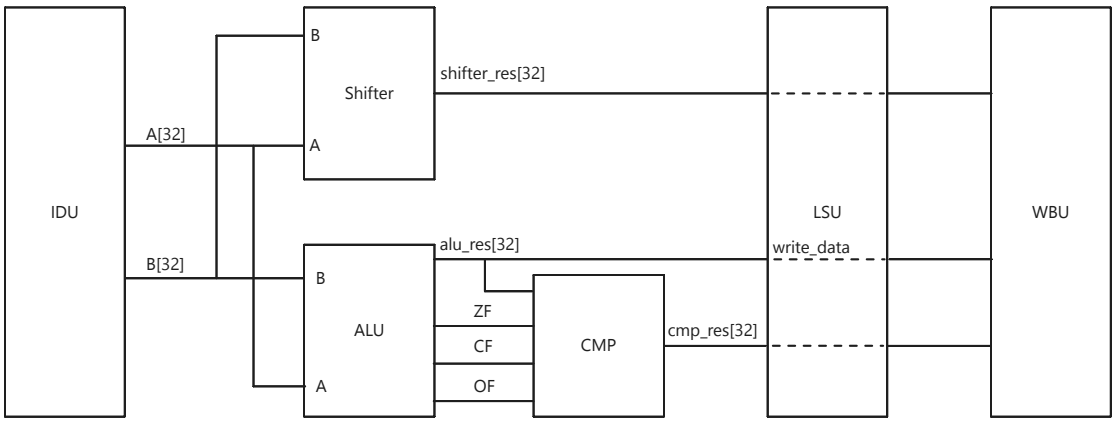


图 3-4 执行单元设计图

3.3.1 加法器

加法器（ALU）是处理器执行单元的核心组件，用于执行基本的算术和逻辑运算。它能够高效地处理加法、减法，以及逻辑运算（如与、或、非等），确保数据的快速处理和准确计算。如图3-5所示，该加法器设计采用经典的架构。输入信号有：两个 32 位的补码操作数，支持有符号数的运算；一个 1 位控制信号，用于决定执行加法或减法运算（控制信号为 1，执行减法运算）。输出信号有：一个 32 位的运算结果，表示算术或逻辑运算的输出；一个 1 位的零标志符 (ZF)，用于指示运算结果是否为零；一个 1 位的进位标志符 (CF)，用于指示运算过程中是否发生进位；一个 1 位的溢出标志符 (OF)，用于检测有符号数运算中的溢出情况。

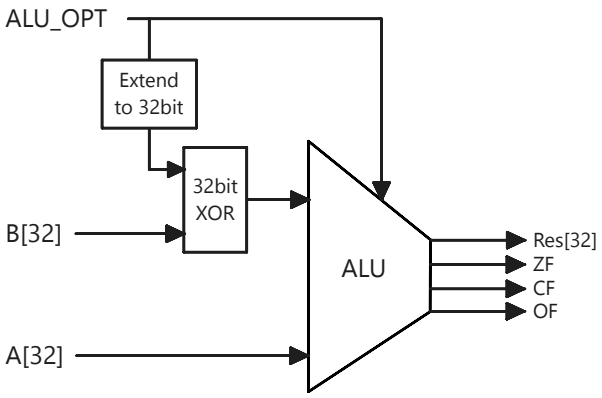


图 3-5 加法器设计图

标志符在计算机系统中起着至关重要的作用，它们是处理器在执行指令后生成的信号，用于指示运算结果的特定状态。此外标志符还是比较器的输入，对于比较运算的判断十分重要。

1) **零标志符（ZF）**：指示运算结果是否为零。如果 ZF 为 1，处理器可以判断某个操作的结果为零。ZF 的运算参考公式3-1，其中 Res_i 表示运算结果的第 i 位。

$$ZF = \neg(Res_{31} \vee Res_{30} \vee \cdots \vee Res_0) \tag{3-1}$$

2) **进位标志符（CF）**：指示无符号数加法中的溢出或减法中的借位。如果 CF 为 1，说明无符号运算发生溢出。CF 的运算参考公式3-2，其中 C_i 表示第 i 位的进位信息。

$$CF = C_{32} \oplus C_0 \tag{3-2}$$

3) **溢出标志符（OF）**：指示有符号数运算中的溢出。如果 OF 为 1，说明有符号运算发生溢出。OF 的运算参考公式3-3。

$$OF = C_{32} \oplus C_{31} \tag{3-3}$$

除了最基本的加减法运算之外，加法器还应该支持逻辑运算和移位。将 ALU_OPT 扩充至 3 位，从而能够支持更多的运算类型，扩充后的接口描述如表3-3所示。

表 3-3 ALU_OPT 接口描述表

ALU_OPT	功能	操作
000	加法	$A + B$
001	减法	$A - B$
010	取反	$\neg A$
011	与	$A \wedge B$
100	或	$A \vee B$
101	异或	$A \oplus B$

3.3.2 比较器

比较器（CMP）是处理器执行单元中的关键器件，用于执行条件比较操作。它能够高效地判断两个操作数之间的关系，为条件分支指令提供依据。输入信号包括来自加法器的四个标志位（SF、ZF、CF 和 OF），这些标志位用于计算比较结果。此外，比较器还接收一个 4 位的控制信号（CMP_OPT），用于决定执行何种比较运算。CMP_OPT 的接口描述如表3-4所示。

3.3.3 移位器

移位器（Shifter）是处理器执行单元中的重要模块，用于执行数据的移位操作。它能够高效地处理多种类型的移位运算，包括逻辑左移、逻辑右移和算术右移，这些操作在数

表 3-4 CMP_OPT 接口描述表

CMP_OPT	功能	逻辑
0001 或 1001	==	$ZF = 1$
0010 或 1010	!=	$ZF = 0$
0011	<（有符号）	$SF \oplus OF$
0100	<=（有符号）	$(SF \oplus OF) \vee (ZF = 1)$
0101	>（有符号）	$\neg(SF \oplus OF)$
0110	>=（有符号）	$\neg(SF \oplus OF) \vee (ZF = 1)$
1011	<（无符号）	$\neg CF$
1100	<=（无符号）	$\neg CF \vee (ZF = 1)$
1101	>（无符号）	$CF \wedge (ZF = 0)$
1110	>=（无符号）	CF

据处理和数值计算中非常常见。输入信号包括一个 32 位的被移数与 32 位的移数，这两个操作数用于计算移位结果。此外，移位器还会接收一个 2 位的控制信号（SHIFTER_OPT），用于决定执行何种移位运算。SHIFTER_OPT 的接口描述表如3-5所示。

1) **逻辑左移（ZF）**：将操作数的各位向左移动指定的位数，空出的低位填充零。

2) **进位标志符（CF）**：将操作数的各位向右移动指定的位数，空出的高位填充零。

3) **溢出标志符（OF）**：将操作数的各位向右移动指定的位数，空出的高位填充操作数的符号位（最高位），以保持有符号数的符号不变。

表 3-5 SHIFTER_OPT 接口描述表

SHIFTER_OPT	功能
00 或 01	逻辑左移
10	逻辑右移
11	算术右移

3.4 访存单元（LSU）

访存单元（Load Store Unit）是处理器与存储器交互的关键模块，其设计严格遵循加载-存储（Load-Store）架构原则，负责对存储器进行读写操作。如图3-6所示，写操作的数据来源于寄存器，写地址来源于执行单元，而读取的数据则被发送到写回单元。

为了降低访问延迟并提高性能，现代处理器中的 LSU 通常采用多种优化技术，如高速缓存（Cache）、预取（Prefetching）和多端口存储器。这些技术能够显著减少存储器访问延迟，提高数据传输效率，从而提升处理器的主频和 IPC（每周期指令数）。然而，在本章中，仅探讨单周期处理器的设计，因此访存单元采用最简单的存取方案。关于高速缓存优

化的详细讨论将在第 4 章中进行，届时将介绍如何通过缓存技术进一步提升访存单元的性能。

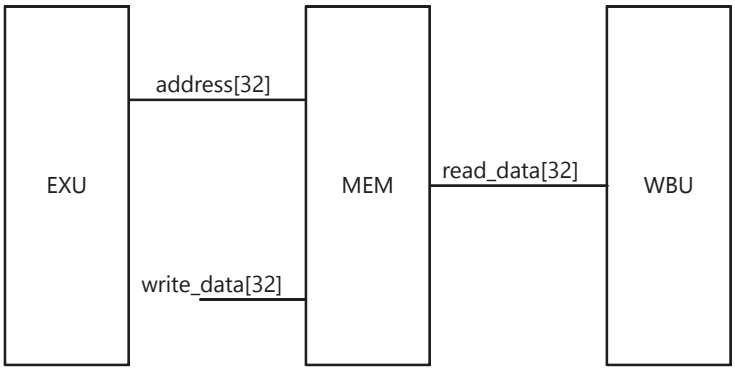


图 3-6 访存单元设计图

3.5 写回单元（WBU）

写回单元（Write Back Unit，WBU）是微处理器五个核心单元中的最后一个单元，其主要职责是将处理结果写入寄存器堆，并更新程序计数器。WBU 确保了运算结果的最终存储和程序流程的正确推进，其设计的高效性和可靠性直接影响处理器的整体性能和稳定性。

写回寄存器的值由控制信号 REG_OPT 确定，该控制信号的位宽为 5 位，由译码单元输出。更新程序计数器的值由控制信号 PC_OPT 确定，该控制信号的位宽为 4 位，由译码单元和执行单元共同输出。表3-6和表3-7是控制信号 REG_OPT 和 PC_OPT 的详细描述。

表 3-6 REG_OPT 接口描述表

REG_OPT	写入寄存器堆的值
00000	加法器结果
00001	PC+4
00010	CSR 的值
00100	比较器结果
01000	访存数据
10000	移位器结果

表 3-7 PC_OPT 接口描述表

PC_OPT	更新 PC 的值
0000	PC+4
0001	PC+IMM
0010	CSR 的值（MRET）
0100	CSR 的值（ECALL）
1000	加法器结果

3.6 本章小结

本章着重阐述了 RISC-V 单周期处理器的设计方法，采用 RV32I 指令集，该指令集支持 RISC-V 架构的基本功能。文中依次详细介绍了处理器五个核心单元的设计，分别为指令取指单元（IFU）、指令译码单元（IDU）、执行单元（EXU）、访存单元（LSU）和写回单元（WBU）。IFU 负责依据程序计数器（PC）从存储器中读取指令，并将其传递至 IDU；IDU 承担将指令译码为多种控制信号的任务，为后续处理单元提供必要的支持；EXU 包含加法器、比较器和移位器，用于执行算术和逻辑运算，并将结果传递给后续单元；LSU 实现存储器的读写操作；WBU 则负责更新寄存器堆和程序计数器（PC）。这种模块化设计明确划分了各单元的职责，不仅降低了设计的复杂度，还便于后续的技术优化，例如总线架构、高速缓存策略和片上系统集成等。

第 4 章 处理器相关技术设计

第 3 章实现了 RISC-V 单周期处理器的设计，其中每条指令均在一个时钟周期内完成。然而，实际上，存储器的存取操作无法在一个周期内完成，即使采用最先进的 SRAM 工艺，也至少需要一个周期的存取延迟[15]。因此，单周期处理器在实际中无法直接流片生产。

本章将在单周期处理器的基础上进行扩展，通过引入 AXI4 总线协议将其改造为多周期处理器。在此基础上，将在存储优化部分引入高速缓存技术，以提升访存效率；在并行优化部分引入流水线技术，以增强处理器的并行处理能力。

4.1 总线

计算机中的各个模块并非孤立运作，不同模块间的数据交换是系统正常运行的关键。例如，指令取指单元（IFU）与指令译码单元（IDU）之间、中央处理器（CPU）与内存控制器之间，均需遵循特定的通信协议以实现高效的数据传输。从广义上讲，总线可被视为一种通信系统，其核心功能在于协调不同模块间的数据流动。本节将专注于狭义上的总线设计，即模块间通信协议的制定。这一协议不仅定义了数据传输的格式和时序，确保数据的完整性和可靠性。通过精心设计的通信协议，可以显著提升系统的整体性能和可扩展性。

4.1.1 内部总线

在处理器内部，各单元间需相互通信。通常，主动发起通信的模块称为主设备（master），而响应通信的模块称为从设备（slave）。然而，主设备无法保证每个周期都能向从设备发送消息（例如，IFU 无法在每个周期内取到指令），从设备也无法确保每个周期都能处理主设备的消息（例如，EXU 无法在一个周期内完成运算）。这种不确定性体现了异步通信的特性，即两个模块之间的通信时机无法预先确定。

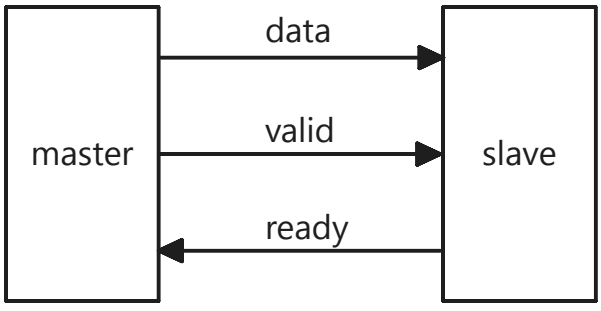


图 4-1 模块间通信的握手协议

为确保异步通信中数据传输的准确性，需设计一种握手协议。如图4-1所示，该协议包含三种信号：data 是主设备向从设备发送的数据；valid：表示发送的数据有效；ready 表示从设备准备好接收数据。通信仅在 valid 和 ready 信号同时有效时发生。这种机制不仅实现了异步通信，还增强了通信协议的灵活性，确保了数据传输的可靠性和高效性。

关于通信逻辑的设计，握手双方需依据握手信号的不同情况进入相应状态并采取操作。为此，需设计一个有限状态自动机，其状态转移如图4-2所示：

1) master 初始处于空闲状态 idle，此时将 valid 信号置为无效。

- 如果无需发送消息，则持续保持 idle 状态。
- 若需发送消息，则将 valid 信号置为有效，并转入 wait_ready 状态。

2) master 处于 wait_ready 状态时，需检测来自 slave 的 ready 信号。

- 若 ready 信号无效，则继续停留在 wait_ready 状态。
- 一旦 ready 信号有效，表明握手成功，随即向 slave 发送消息，并返回 idle 状态。

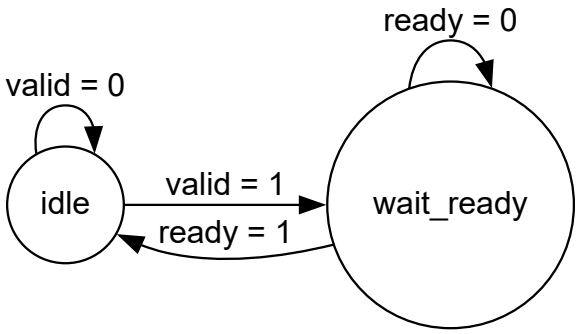


图 4-2 master 的有限状态自动机设计

在单周期处理器的基础上，如图4-3所示，通过在有数据交换的单元模块之间引入握手信号，可以有效管理模块间的通信。对于单周期处理器，每个周期内上游模块发送的消息都被视为有效，而下游模块始终处于就绪状态以接收新消息。这种设计假设所有模块都能在一个周期内完成其任务，这在实际应用中可能并不现实。

相比之下，多周期处理器采用更为灵活的通信机制。在多周期处理器中，当上游模块空闲时，它不会发送有效消息，而下游模块在忙碌时也不会接收新消息。这种机制允许模块根据自身处理速度进行通信，避免了不必要的等待和资源浪费。特别地，IFU 会在收到 WBU 的完成信号后，才取下一条指令，确保了数据传输的准确性和处理器操作的有序性。这种基于握手信号的通信协议，不仅提高了处理器的灵活性和效率，还为处理复杂操作提供了必要的支持。

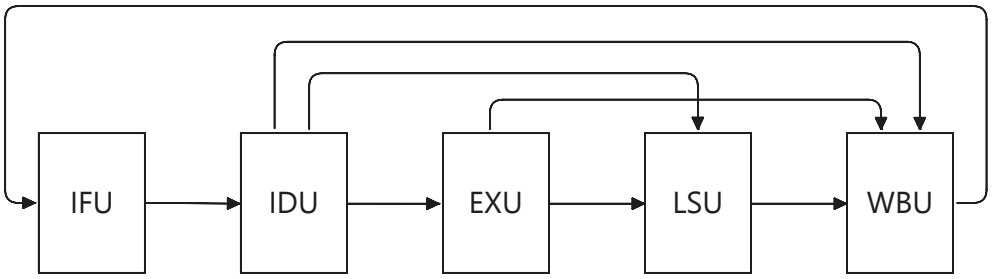


图 4-3 多周期处理器结构图

如图4-4所示，集中式控制处理器通常显得较为累赘，尤其是在模块数量和复杂度不断

增加的情况下。这种控制方式需要一个全局控制器来收集所有模块的状态信息，并据此来协调各个模块的下一步操作。可以将集中控制器视为一个大型的有限状态自动机，它通过一个全局的状态机来管理各个模块的状态，从而控制模块间的通信并决定下一个状态。然而，这种集中式的方法在可扩展性方面存在明显不足，因为随着系统规模的扩大，控制器的设计复杂度会呈指数级增长，需要考虑每个模块不同状态的组合，这使得统一决策变得异常困难。

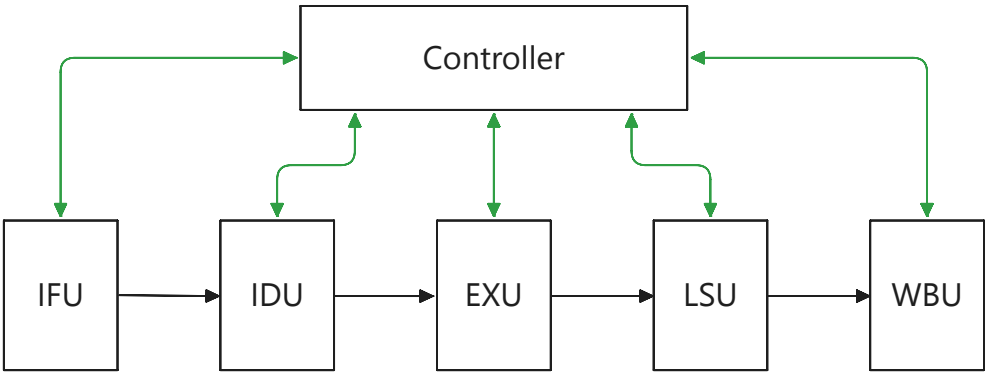


图 4-4 集中控制处理器结构图

所以，基于握手的分布式控制提供了一种更为灵活和可扩展的解决方案。在这种方法中，每个模块的行为仅取决于自身的状态以及下游模块的状态。这种局部通信机制允许各个模块相对独立地运作，从而简化了整体设计。分布式控制不仅提高了系统的可扩展性，还使得处理器结构的修改和扩展变得更加容易，因为每个模块可以独立地进行调整和优化，而不会对整个系统造成过多的影响。这种模块化的设计理念，使得处理器能够更好地适应不断变化的需求和日益增长的复杂性。

4.1.2 系统总线

处理器不仅需要建立内部模块间的通信，还需要与外围设备（如存储器、时钟和串口等）进行通信。为了实现这一目标，系统总线扮演着至关重要的角色。本小节将介绍 AXI4（Advanced eXtensible Interface 4）总线协议，该协议是 ARM 的 AMBA（Advanced Microcontroller Bus Architecture）规范的一部分[16]，旨在提供一种灵活、高效的方式来连接处理器、存储器和外设。

AXI4 总线协议以其高效性和灵活性著称，特别适用于复杂系统设计。它通过五条独立的传输通道（读地址通道、写地址通道、读数据通道、写数据通道和写响应通道）实现高效的通信机制，这些通道仅支持单向传输，确保了数据传输的高效性和可靠性。通过这种架构，AXI4 总线能够有效地管理数据流，减少通信延迟，并提高系统的整体性能。其灵活的配置选项使其适用于各种规模和复杂度的系统设计，从而成为现代处理器设计中的首选总线协议之一。简单来说，AXI4 总线是为每种传输数据加上了 valid 和 ready 握手信号。

读地址通道用于从主设备到从设备的读请求地址传输，其信号描述如表4-1所示。

表 4-1 读地址通道信号描述表

信号名	源	描述
ARVALID	主设备	表示主设备已准备好发送读地址信息
ARREADY	从设备	表示从设备已准备好接收读地址信息
ARADDR	主设备	读操作的目标地址
ARLEN	主设备	传输的长度，表示一次读操作的传输次数
ARSIZE	主设备	传输的大小，表示每次传输的数据宽度
ARBURST	主设备	突发类型，描述传输的模式。
ARID	主设备	事务标识符，用于标识不同的读操作

写地址通道用于从主设备到从设备的写请求地址传输，其信号描述如表4-2所示。

表 4-2 写地址通道信号描述表

信号名	源	描述
AWVALID	主设备	表示主设备已准备好发送写地址信息
AWREADY	从设备	表示从设备已准备好接收写地址信息
AWADDR	主设备	写操作的目标地址
AWLEN	主设备	传输的长度，表示一次写操作的传输次数
AWSIZE	主设备	传输的大小，表示每次传输的数据宽度
AWBURST	主设备	突发类型，描述传输的模式。
AWID	主设备	事务标识符，用于标识不同的写操作

读数据通道用于从设备到主设备的读数据响应，其信号描述如表4-3所示。

表 4-3 读数据通道信号描述表

信号名	源	描述
RVALID	从设备	表示从设备已准备好发送读数据信息
RREADY	主设备	表示主设备已准备好接收读数据信息
RDATA	从设备	读操作返回的数据
RRESP	从设备	响应类型，用于描述读操作的状态
RLAST	从设备	表示当前数据是读操作的最后一个数据
RID	从设备	事务标识符，用于标识不同的读操作

写数据通道用于从主设备到从设备的写数据传输，其信号描述如表4-4所示。

表 4-4 写数据通道信号描述表

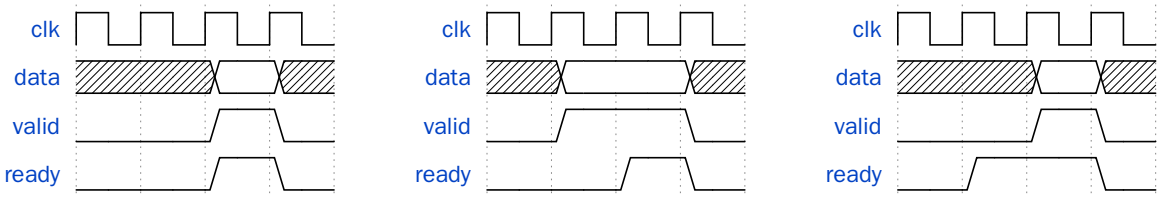
信号名	源	描述
WVALID	主设备	表示主设备已准备好发送写数据信息
WREADY	从设备	表示从设备已准备好接收写数据信息
WDATA	主设备	写操作的数据
WSTRB	主设备	写选通，用于指示数据的哪些字节有效
WLAST	主设备	表示当前数据是写操作的最后一个数据
WID	主设备	事务标识符，用于标识不同的写操作

写响应通道用于从设备到主设备的写操作状态响应，其信号描述如表4-5所示。

表 4-5 写响应通道信号描述表

信号名	源	描述
BVALID	从设备	表示从设备已准备好发送写响应信息
BREADY	主设备	表示主设备已准备好接收写响应信息
BRESP	从设备	写操作的响应类型，用于描述写操作的状态
BID	从设备	事务标识符，用于标识不同的写操作

关于 AXI4 总线的握手机制只有一种（valid 和 ready 信号），若在时钟上升沿检测到 valid 和 ready 信号同时有效，此时消息有效，下一周期 valid 和 ready 信号将置为无效。如图4-5所示，根据 valid 和 ready 信号的先后顺序，将会出现三种时序关系。



(a) ready 和 valid 同时到达　(b) valid 比 ready 先到达　(c) ready 比 valid 先到达

图 4-5 不同情况下的握手时序图

1) **双方同时准备就绪**：若发送方和接收方同时将 valid 和 ready 信号置为有效，数据传输立即开始，完成该次传输任务。

2) **发送方先准备就绪**：若发送方先将 valid 信号置为有效，表示数据或地址及控制信号已准备就绪，发送方需维持这些信号直至接收方将 ready 信号置为有效，从而完成数据传输。

3) **接收方先准备就绪**：若接收方先将 ready 信号置为有效，表示已准备好接收数据，发送方随后需将数据或地址及控制信号准备就绪，并将 valid 信号置为有效，以完成传输。

至此，单周期处理器已成功优化为多周期处理器。内部总线负责协调处理器内部各模块间的通信，而系统总线则负责处理器与外部设备之间的通信。如图4-6所示，该多周期处理器的总线结构通过仲裁器（Arbiter）将指令取指单元（IFU）和访存单元（LSU）连接到 CPU 外部的 AXI4 总线接口。该接口随后连接到信号路由器（XBar），通过 XBar 将信号精确路由至相应的外部设备，从而实现高效的数据传输和设备通信。这种设计确保了处理器内部和外部通信的高效性和灵活性。

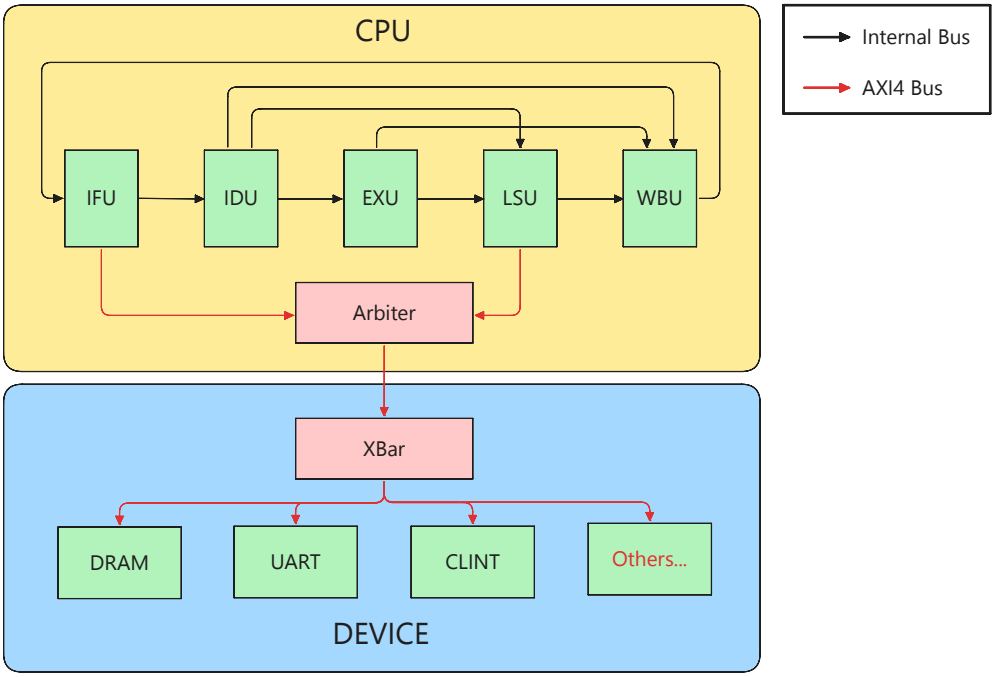


图 4-6 加入总线后的处理器结构图

4.2 系统优化

目前，多周期处理器虽已能够支持基本程序的运行，但其性能表现仍存在明显局限。本节将深入探讨 CPU 的优化策略。在存储优化方面，将引入高速缓存技术，以显著提升数据访问效率，减少存储器访问延迟。在并行优化方面，将采用流水线技术，通过指令重叠执行增强处理器的并行处理能力，从而提高整体性能。这些优化措施旨在解决当前多周期处理器的性能瓶颈，为其在复杂计算任务中的应用奠定基础。

4.2.1 存储优化——高速缓存

针对存储器的层次结构，表4-6展示了不同存储器的性能特点。可以看出，涉及访存的指令通常比其他指令耗时高几个量级。由于存储介质的物理限制，没有一种存储器能够同时满足大容量、高速度和低成本的要求。因此，计算机系统通常集成多种存储器，并通过特定技术将它们有机组织成层次结构，以在整体上实现大容量、高速度和低成本的综合性能。

表 4-6 存储器层次结构

存储器	延迟	成本	容量
寄存器	约 1ns	高	约 1KB
主存（DRAM）	约 10ns	中等	约 10GB
外存（SSD）	约 10us	低	约 1TB
磁盘（HDD）	约 10ms	低	约 10TB
磁带	约 10s	最低	大于 10TB

为了理解存储层次结构的设计原因，需要引入程序局部性的概念：

1) **时间局部性**：一旦访问某个存储单元，短时间内很可能会再次访问它，例如在循环结构中。

2) **空间局部性**：一旦访问某个存储单元，短时间内很可能会访问其相邻单元，例如在数组遍历中。

基于局部性原理，即使慢速存储器的容量很大，程序在一段时间内通常只会访问其中的一小部分数据。因此，可以将这部分数据从慢速存储器预取到快速存储器中，以便后续访问。这就是缓存（cache）的基本思想。为了优化访存效率，可以在寄存器和 DRAM 之间添加一层 cache，在访问 DRAM 之前，先访问 cache：如果所需数据已在 cache 中，则直接访问 Cache 中的数据；如果所需数据不在 cache 中，则先将数据从 DRAM 读入 cache，然后再访问 cache 中的数据。

如图4-7所示，在 IFU 和 Arbiter 之间引入指令缓存（icache），以加速指令的访问；在 LSU 和 Arbiter 之间引入数据缓存（dcache），以加速数据的读写操作。这种设计通过减少对主存的直接访问，显著降低访存延迟，从而提升处理器的整体性能。

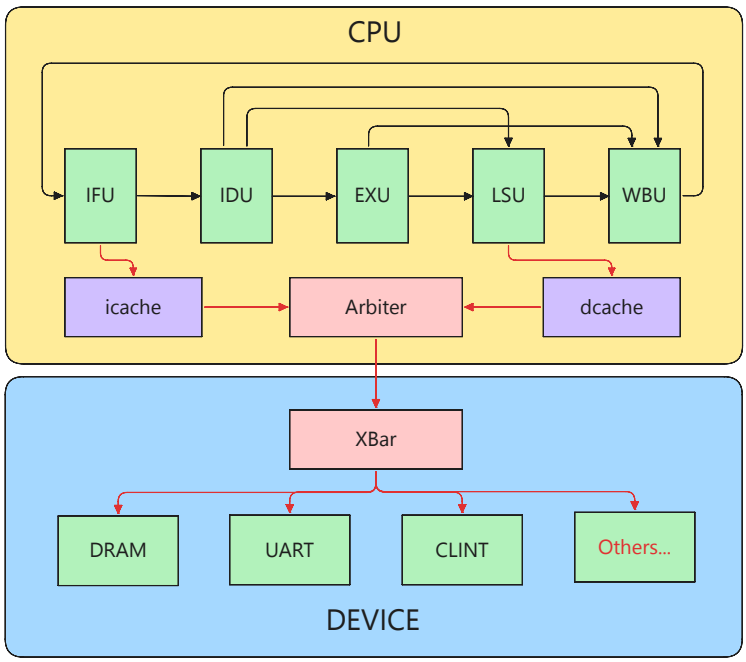


图 4-7 加入 cache 后的处理器结构图

关于主存与缓存地址的映射方式，主要有以下三种，它们对缓存效率和性能的影响各不相同：

1) **全相联映射**：全相联映射允许主存中的任意一个块映射到缓存中的任意一个位置。这种方式提供了最大的灵活性，因为主存中的每个块都可以放置在缓存的任何位置。然而，这种灵活性也带来了较高的查找成本，因为每次访问缓存时都需要检查所有缓存行，以确定是否存在所需的数据。优点是缓存利用率高，能够有效减少缓存缺失，但缺点是硬件实现复杂，查找速度较慢。

2) **直接相联映射**：直接相联映射将主存中的每个块固定映射到缓存中的一个特定位置。具体来说，主存地址通过哈希函数（通常是取模操作）计算出缓存中的唯一位置。这种方式的查找速度非常快，因为每次访问缓存时只需要检查一个位置即可。优点是硬件实现简单，查找速度快，但缺点是缓存利用率较低，缓存缺失率较高。

3) **组相联映射**：组相联映射是前两者的折中方案。它将缓存分为多个组，每个组包含多个缓存行。主存中的每个块可以映射到缓存中的一个特定组，但在组内可以自由选择缓存行。这种方式结合了直接相联映射的快速查找和全相联映射的高利用率，提供了较好的性能和硬件复杂度的平衡。组相联映射在查找速度和缓存利用率之间取得了较好的平衡，是现代缓存设计中最常用的方式。

关于替换算法，由于程序运行一段时间后 cache 的空间会满，所以需要需要某种机制进行替换，主要的替换算法有以下四种：

1) **随机替换（RR，Random Replacement）**：随机选择一个缓存行进行替换。这种方法简单易实现，但性能较差。

2) **先进先出（FIFO，First-In-First-Out）**：替换最早进入缓存的行。这种方法基于时间顺序，简单且易于硬件实现，但可能导致频繁使用的数据被过早替换。

3) **最不经常使用（LFU，Least Frequently Used）**：替换访问次数最少的缓存行。这种方法通过统计每个缓存行的访问频率来决定替换策略，适用于访问模式较为固定的场景。

4) **近期最少使用（LRU，Least Recently Used）**：替换最近一段时间内最少使用的缓存行。这种方法基于时间局部性原理，能够较好地保留频繁访问的数据，是现代缓存设计中最常用的算法之一。

4.2.2 并行优化——流水线

目前，处理器采用多周期设计，即执行一条指令需要多个时钟周期。为了提升指令执行的吞吐量，从而增强计算效率，流水线技术作为一种指令级并行方法被广泛应用。流水线通过将指令的执行过程分解为多个阶段，使得每个阶段可以在不同的时钟周期内并行处理，从而显著提高指令的吞吐量。如图4-8所示，在处理器运行期间，各单元模块并行工作，极大地提升了整体吞吐量。

对于流水线处理器，可以通过让 IFU 持续取指，并在每个模块完成消息处理后立即向下游发送消息，从而实现高效并行处理。具体来说，只需在模块间添加流水线寄存器，并优化模块间通信逻辑，即可实现流水线设计。如图4-9所示，加入流水线技术后的处理器被划分为五个流水段，每个阶段负责特定的指令处理任务，从而实现高效的指令级并行。

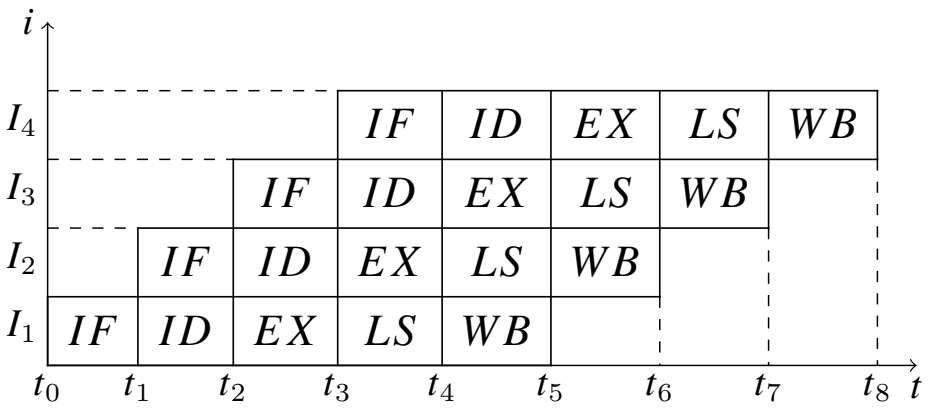


图 4-8 指令流水线时空图

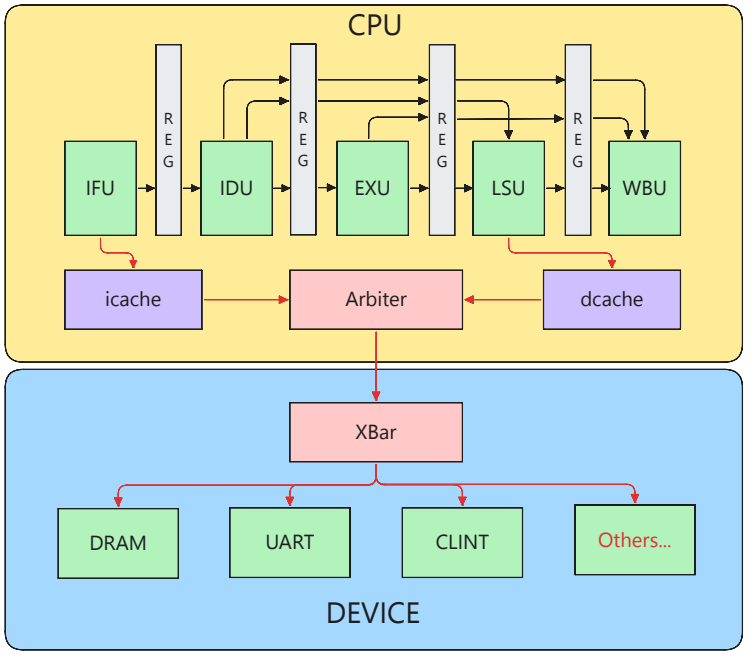


图 4-9 加入流水线技术后的处理器结构图

流水线处理器中存在一些情况，可能导致当前指令无法按预期执行，这种情况称为冒险（Hazard）。若忽视冒险而强行执行指令，将导致 CPU 状态机的转移结果与 ISA 状态机不一致，表现为指令执行结果不符合其语义。因此，在流水线设计中，必须检测并解决冒险问题。冒险主要分为以下三类：

1) **结构冒险**：由于硬件资源不足，导致多个指令同时竞争同一资源。例如，多个指令同时需要访问同一功能单元或总线。可以通过增加硬件资源的副本，以减少竞争；或通过调整指令执行顺序，避免资源冲突。

2) **数据冒险**：由指令间的数据依赖关系引起。例如，后续指令需要使用前条指令尚未生成的结果。可以通过在流水线中插入空操作，确保数据准备好后再执行后续指令；或将后续指令所需的数据直接从上游阶段转发到下游阶段，避免等待。

3) **控制冒险**：由分支指令引起，由于分支方向未确定，处理器无法立即确定后续指令的取指地址。可以通过使用硬件预测器猜测分支方向，提前取指后续指令；或将分支指令后的几条指令延迟执行，确保分支方向确定后再继续。

4.3 本章小结

本章针对处理器在访问存储器和外围设备时存在的延迟问题，引入了总线协议，并采用缓存技术和流水线技术分别优化访存效率和提升并行处理能力。通过这些改进，已完成一款五级流水线的 RISC-V 处理器设计方案。与单周期处理器相比，该方案不仅显著提升了处理器性能，还使其能够在真实环境中实现流片生产，为实际应用奠定了坚实基础。

第 5 章 仿真测试

5.1 verilator 仿真

5.2 软硬件差分测试

第 6 章 总结和展望

6.1 本文总结

6.2 未来展望

参考文献

[1] Faggin F, Hoff M E, Mazor S, et al. The history of the 4004[J]. Ieee Micro, 1996, 16(6):10-20.

[2] Cocke J, Markstein V. The evolution of risc technology at ibm[J]. IBM Journal of research and development, 1990, 34(1):4-11.

[3] 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, 116:1-32.

[4] Asanović K, Patterson D A. Instruction sets should be free: The case for risc-v[J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014.

[5] 黄鑫. “十四五”软件业开源生态加快构建[R]. 经济日报, 2021.

[6] Marena T. Risc-v: high performance embedded swerv™ core microarchitecture, performance and chips alliance[J]. Western Digital Corporation, 2019.

[7] 平头哥发布全新 RISC-V 处理器[J]. 中国集成电路, 2021, 30(06):21.

[8] 王凯帆, 徐易难, 余子濠, 等. 香山开源高性能 RISC-V 处理器设计与实现[J]. 计算机研究与发展, 2023, 60(03):476-493.

[9] Daki0107 V, Mr0161i0107 L, Kuni0107 Z, et al. Evaluating arm and risc-v architectures for high-performance computing with docker and kubernetes[J/OL]. Electronics, 2024, 13(17). https://www.mdpi.com/2079-9292/13/17/3494. DOI: 10.3390/electronics13173494.

[10] Fpga '21: The 2021 acm/sigda international symposium on field-programmable gate arrays[C]. New York, NY, USA: Association for Computing Machinery, 2021.

[11] 邓天传, 胡振波. 一种超低功耗的 RISC-V 处理器流水线结构[J/OL]. 电子技术应用, 2019, 45(06): 50-53. DOI: 10.16157/j.issn.0258-7998.182563.

[12] 钟戴元, 曾庆立, 周佳凯, 等. 基于 RISC-V 处理器的软硬件联合验证平台设计与实现[J]. 信息技术与信息化, 2024(11):23-26.

[13] 郝振和, 焦继业, 李雨倩. 基于 AHB 总线的 RISC-V 微处理器设计与实现[J]. 计算机工程与应用, 2020, 56(20):52-58.

[14] 潘树朋, 刘有耀, 焦继业, 等. 基于 RISC-V 浮点指令集 FPU 的研究与设计[J]. 计算机工程与应用, 2021, 57(03):80-86.

[15] Noguchi H, Okumura S, Iguchi Y, et al. Which is the best dual-port sram in 45-nm process technology? —8t, 10t single end, and 10t differential—[C]//2008 IEEE International Conference on Integrated Circuit Design and Technology and Tutorial. IEEE, 2008: 55-58.

[16] AMBA A. Axi and ace protocol specification[J]. ARM IHI D, 2011, 22(147):118.

致谢

谢谢!