

韶关学院

毕业设计

题 目： 基于 Chisel 的精简指令集微处理器设计

学生姓名： 荀阳霖

学 号： 19125042040

二级学院： 信息工程学院

专 业： 物联网工程

班 级： 20 物联网工程 2 班

指导教师姓名及职称： 王娜 讲师

起止时间： 2023 年 9 月 —— 2024 年 6 月

(教务处制)

基于 Chisel 的精简指令集微处理器设计

摘要：针对当前处理器芯片领域面临的高技术壁垒、人才短缺、物联网时代多元化需求以及安全挑战等问题，结合开源指令集架构 RISC-V 的兴起，提出并实现了一款基于 Chisel 语言的精简指令集微处理器设计，具备模块化、可扩展的架构特点，适用于物联网应用的多样性和安全性要求。Chisel 语言的采用显著提升了设计效率与代码质量，降低了芯片设计门槛，体现了敏捷开发在芯片设计领域的优势。通过严谨的验证流程与仿真分析，证明了所设计处理器的正确性和有效性。为物联网领域提供了定制化、低成本、高能效的处理器解决方案，同时为我国芯片设计人才培养与核心技术掌握提供了实践范例。

关键词： RISC-V 指令集；Chisel 语言；敏捷开发；芯片设计；片上系统

Chisel implementation of a RISC microprocessor

ABSTRACT: In response to the high technological barriers, talent shortages, diverse demands of the Internet-of-Things (IoT) era, and security challenges confronting the processor chip domain, this work proposes and realizes a modular and extensible Reduced Instruction Set Computer (RISC) microprocessor design based on the Chisel language, tailored to accommodate the variety and security requisites of IoT applications. The adoption of Chisel significantly enhances design efficiency and code quality, lowering the entry barrier for chip design while exemplifying the benefits of agile development within the realm of semiconductor engineering. Through rigorous verification processes and simulation analyses, the correctness and efficacy of the designed processor are substantiated. This effort furnishes the IoT sector with a customizable, cost-effective, and energy-efficient processor solution, concurrently providing a practical exemplar for cultivating talent in chip design and mastering core technologies within our nation.

Key words: RISC-V instruction set; Chisel; Agile development; Chip design; System on chip

目 录

中文摘要	I
ABSTRACT	II
目 录	III
绪论	1
1.1 研究工作的背景与意义	1
1.2 国内外研究现状	2
1.3 一生一芯计划	3
处理器架构与敏捷开发	4
2.1 指令集架构	4
2.1.1 复杂指令集与精简指令集	4
2.2 RISC-V 指令集	4
2.2.1 RISC-V 指令集的优势	5
2.3 敏捷开发及应用	6
2.3.1 敏捷开发的意义	6
2.3.2 芯片设计与敏捷开发	7
2.3.3 芯片行业的敏捷开发	7
2.4 Chisel 语言在芯片设计中的优势及其实现原理	8
2.4.1 Chisel 的优点	9
处理器微架构设计与实现	12
3.1 信号握手	13
3.2 取指单元	13
3.2.1 PC 生成器	14
3.2.2 指令访存器	15
3.3 译码单元	16
3.3.1 译码器	17
3.4 执行单元	17

3.5 访存单元	18
3.5.1 数据访存器	19
3.6 写回单元	19
3.7 Chisel 实现	20
3.7.1 顶层模块	20
3.7.2 寄存器组	20
3.7.3 译码单元	21
3.7.4 算数逻辑单元	23
3.7.5 AXI 读分用器	23
 验证	26
4.1 验证流程	26
4.2 硬件验证环境	26
4.2.1 Difftest	27
4.2.2 Verilator	27
4.2.3 自制调试器	27
4.2.4 仿真界面	28
4.3 软件验证环境	29
4.3.1 抽象计算机	29
4.3.2 适配抽象计算机	31
4.3.3 适配存储结构	32
4.4 仿真分析	34
4.4.1 运行测试	34
 原型验证	38
5.1 片上系统	38
5.1.1 外设地址空间	39
5.2 现场可编程门阵列	39
5.3 GPIO 控制器	40
5.3.1 寄存器设计	40
5.3.2 主体代码	40
5.3.3 测试程序	41

5.4 UART 控制器	43
5.4.1 寄存器设计	43
5.4.2 测试程序	43
总结与展望	44
6.1 总结	44
6.2 展望	45
参考文献	46
致 谢	VI
附录	VII

绪论

1.1 研究工作的背景与意义

高质量芯片人才需要掌握计算机整体的软硬件理论知识与全程工程技术。既要在“学中做”，也要在“做中学”^[1]。完成知识的有机融合，打通从数字电路、组成原理、体系结构、操作系统、嵌入式开发的全流程知识点。

世界正在加速转型进入数字经济时代。而做为数字化引擎的处理器芯片，则是一切的基石。虽然信息领域市场巨大，竞争强烈，但十几年来却已经形成了“赢家通吃”的场面。高精尖技术与人才长期被把握在少数西方大公司的壁垒之中。

随着中美贸易环境的变化和技术封锁措施，一些中国企业和高校被列入美国政府的‘实体清单’，这对中国芯片技术研发和人才培养带来了显著影响。目前，我国的计算机专业人才培养面临着较大的结构问题。顶层应用开发者过多，而底层软硬件研发人员缺乏。特别是芯片设计人才严重不足。2022 年间我国集成电路产业人才缺口达到 3.5 万^[2]。这与当前芯片设计门槛过高，导致中国大学无法开展芯片相关教学与研究密切相关^[3]。

晶体管性能高速提升的时代即将结束，半导体行业即将进入后摩尔时代。提升芯片性能与能效的压力逐渐转移到了架构设计师与电路设计师上^[4]。对领域特定架构 (DSA) 的需求日渐增加，产业界急需一种更加快速、灵活的研发方法和基础架构。

RISC-V(第五代精简指令集)是一款开源且免费的精简指令集架构。自从该指令集架构发布以来，在国内外吸引了许多关注。中国工程院院士倪光南表示，RISC-V 具有模块化、可扩展、易定制的优势，以及不受垄断制约、供应链安全容易保障的优势，基于 RISC-V 架构的 DSA 服务器是中国的机遇^[5,6]。

如今，已有来自 70 多个国家的 3900 多个实体加入了 RISC-V 联盟，其中高级成员有超过 40% 来自中国。国内外越来越多的公司、组织和开发人员正在支持和贡献 RISC-V。包括芯片设计商、软件开发商、工具提供商以及三星电子、西部数据和英伟达等行业巨头。开放的 RISC-V 架构鼓励了全球范围内的协作和技术创新，降低了进入壁垒，极大地促进学术界和创业公司在芯片设计领域的创新活动。其开源免费、模块化、容易定制

等特点，为我国芯片设计的发展带来了新的机遇。有助于我国掌握核心关键技术的同时开放地对待国际交流与合作。

物联网时代的到来对处理器芯片提出了更多的需求。RISC-V 的模块化设计允许制造商根据特定应用场景进行裁剪或扩展，设计出高度优化的芯片解决方案，降低功耗、提升性能并缩小芯片面积，进而降低成本。从而满足物联网行业中多样化、碎片化的市场需求。同时，鉴于物联网设备数量庞大且分布广泛，其安全问题愈发凸显。RISC-V 因其透明的架构设计，有利于定制深度的防御策略，确保从硬件层面强化安全特性。诸如“港华芯”^[7]这样的 RISC-V 物联网安全芯片的成功案例表明，RISC-V 能够有效应对物联网安全挑战，并在能源、智慧城市等多个领域实现关键突破。

1.2 国内外研究现状

RISC-V 指令集具有简洁，现代的特点，自 2010 年推出以来，受到了国内外的广泛关注。很多高校和企业都开始基于 RISC-V 指令集设计设计了各式的处理器核。

在国际上，伯克利大学基于 RISC-V 指令集，使用 Chisel 语言构建来开源的 SoC 生成器，Rocket Chip^[8]。由伯克利大学的研究团队于 2012 年推出。该生成器支持生成有序执行内核（Rocket）与乱序执行内核（Boom），还支持指令集扩展，协处理器等功能。用户通过配置文件描述设计后，Rocket Chip 生成器可以自动挑选所需的模块组合到设计中。

Tenstorrent 公司通过组合大量精简的 RISC-V 核，使用片上网络技术，实现根据负载动态开关，在降低功耗的同时支持大规模的 AI 加速计算。

在国内，睿思芯科仅用了 7 个月的时间^[9]，就完成了一款基于 RISC-V 的 64 位可编程终端 AI 芯片。这相比传统芯片设计效率提高了大概三倍。

中科院香山团队在 2021 年成功设计并流片了第一版雁栖湖架构处理器。在 28nm 的工艺节点下达到 1.3GHz 的频率。而后在 2023 年成功设计并流片了第二版南湖架构处理器，在 14nm 工艺节点下频率达到 2GHz。香山处理器的目标是成为面向世界的体系结构创新开源平台，基础能力、设施、流程的建立是香山处理器长期高质量发展的关键，香山团队设计了丰富的基础工具支撑起了这一复杂度量级的敏捷验证流程。

中国科学院软件研究所发布基于 RISC-V 的开源笔记本电脑“如意 BOOK”，搭载 C910 处理器，在 openEuler 操作系统上流畅运行钉钉、Libre Office 等大型办公软件。

阿里巴巴旗下公司平头哥的玄铁 RISC-V 处理器核，5 年间发布 3 个系列 9 款产品，覆盖高性能、高能效、低功耗等不同场景，在 AI、5G 通信、自动驾驶、金融等领域展开广泛应用创新，出货已超 40 亿颗。

算能公司于 2023 年推出的基于 RISC-V 处理器 SG2042，主频 2GHz，拥有 64 个核心，达到高性能处理器梯队。

兆易创新生产的 GD32V 系列微控制器（MCU）也使用了 RISC-V 核心，广泛在国内嵌入式领域使用。

1.3 一生一芯计划

为应对我国芯片人才培养中遇到的设计门槛高、课程间关联弱、实践不足、学习坡度陡等挑战。缓解我国芯片被“卡脖子”等形式。秉承“科研为国分忧，创新与民造福的理念。循着“设计一种新的教学体系，降低芯片设计门槛，让学生能设计自己的芯片并成功流片”的思路。在中国科学院-先进计算机系统研究中心联与中国科学院大学的大力支持下。由孙凝晖院士部署，包云岗团队牵头设计并组织了“一生一芯”计划。

计划的目标是突破传统课程的边界，构建一套软硬件协同，理论和实践并发，打通前后端的全链条处理器芯片教学流程。让学生理解从体系结构设计、开发，到集成验证、综合、物理设计，最终生成可流片的版图的全过程。籍此提高我国处理器芯片设计的人才培养质量，培养更多紧缺的芯片人才。“一生一芯”计划不限学校、年级、专业，只要学生对处理器芯片设计感兴趣，都能报名参加，并资助完成学习计划的在校生免费流片。通过开放式的教育，打破教育资源壁垒，加速培养我国处理器芯片紧缺的人才。

处理器架构与敏捷开发

2.1 指令集架构

2.1.1 复杂指令集与精简指令集

指令集发明于 1960 年代，当时 IBM 为了解决该公司不同计算机产品线的程序无法通用的情况。推出了完全独立于硬件架构的指令集架构（ISA）这一概念。

在当时，程序员通常直接使用汇编语言进行开发。市场普遍认为指令集应当更加丰富，每条指令本身应该能实现更多功能。因此，当时一些高级语言中常见的概念，如函数调用、循环、数组访问等都有直接对应的机器指令。这使得早期的指令集均为复杂指令集（CISC）。

1980 年开始，随着存储器成本大幅下降、指令数目持续增加、高级语言的普及，CISC 的一些缺陷开始暴露。指令利用率低、控制电路趋于复杂、研发周期过长、验证难度增加等^[10] 催生了对于精简指令集（RISC）的研究。

与 CISC 不同，RISC 强调每条指令的功能单一化。通过指令的组合来完成复杂操作。由于指令复杂程度下降，编译器更容易进行代码生成。同时节约的电路面积可以用于缓存或流水线等高级结构，使得 RISC 获得了优秀的能效比。

2.2 RISC-V 指令集

2010 年，David Patterson 等人出于教学目的，在仔细评估了市面上各类指令集后，发现它们都存在设计上的不足^[11]。决定重头开始，吸取指令集架构领域 25 年期间的经验教训。设计了一款新的开源指令集架构——RISC-V。

RISC-V 指令编码格式如下

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1	funct3		rd		opcode					R-type
	imm[11:0]			rs1	funct3		rd		opcode					I-type
imm[11:5]		rs2		rs1	funct3		imm[4:0]		opcode					S-type
imm[12:10:5]		rs2		rs1	funct3	imm[4:1 11]		opcode						B-type
	imm[31:12]						rd		opcode					U-type
	imm[20:10:1 11 19:12]						rd		opcode					J-type

图 2.1 RISC-V 指令格式

RISC-V 指令编码尽可能将含义相同的数据放在指令的同一位置，简化了指令译码。如：指令存在目标寄存器字段时，其一定在指令的[11,7]位。指令存在立即数时，立即数的符号位一定在指令的[31]位从而使符号扩展与指令译码并行展开。当立即数在指令中的位置被其他字段切断时，指令编码的设计使得切断后数位的排布在不同指令间尽可能重叠。允许工程师使用更少的多路选择器实现立即数的提取，简化了电路设计。

2.2.1 RISC-V 指令集的优势

- 基础加扩展式的 ISA：RISC-V 架构以模块化的方式组合在一起，用户可以根据需求选择不同模块以满足不同场景的应用。

表 2.1 RISC-V 指令集模块（节选）

RISC-V 指令集	名称	指令数	描述
基础指令集	RV32I	47	基本整数指令，寻址空间 32 位，必须实现
	RV32E	47	指令同上，寄存器减少为 16 个，用于嵌入式领域
	RV64I	59	整数指令，寻址空间位 64 位，包含 RV32I
	RV128I	71	整数指令，寻址空间位 128 位，包含 RV64I
扩展指令集	M	8	整数乘除法
	A	11	原子存储指令
	F	26	32 位单精度浮点指令
	D	26	64 位双精度浮点指令
	C	46	高密度指令，长度为正常指令的一半

- 紧凑的代码体积：智能物联网产品对成本和存储空间通常较为敏感，RISC-V 提供统一的指令压缩扩展，可以在同样的大小下存储更多指令，并且不对成本产生显著影响。
- 开放：传统 ISA 高额的授权费和严格的使用限制对于学术界和初创企业来说是巨大的负担。而 RISC-V 是开源架构，其规范可公开获取，无需支付专利费或许可费，有助于减少行业依赖于少数专有指令集架构的风险，促进公平竞争和市场活力。
- 简洁：RISC-V 在设计之初参考了大量现有 ISA 的经验与缺点，设计更加现代化，基础指令只有 40 余条。避免了对某一特定微架构的过度优化，允许研究者在不被各类复杂的兼容性需求约束的情况下，自由探索各类微架构设计。

- 灵活:RISC-V 允许自定义和扩展指令集架构,使芯片设计师能够根据业务需求量身定制架构。这种灵活性使 RISC-V 在物联网设备到高性能计算系统等广泛应用领域都具有吸引力。
- 便于教育和研究:RISC-V 作为简洁且现代的开源架构,使其成为体系结构教学与研究的绝佳平台。许多大学和学术机构已采用 RISC-V 进行教学,并开发创新的硬件和软件解决方案。
- 完整的软件支持: RISC-V 提供完整的软件堆栈, 编译器工具链以及继承开发环境和操作系统支持。

2.3 敏捷开发及应用

敏捷开发是一种现代软件开发方法论,它倡导迭代、增量开发,强调灵活性和响应变化的能力。这种方法论源于对传统瀑布模型的反思与改进,后者往往在一个阶段结束后才转入下一个阶段,各个阶段间耦合度较高,且不易适应需求变更。进而发展为项目延期,客户需求无法满足,甚至项目失败。

在敏捷开发中,项目被分解为一系列短周期(称为冲刺或迭代),每个周期专注于交付可用的产品增量。团队在整个开发过程中持续集成、测试和交付,保证软件质量。敏捷开发的核心价值观和原则体现在《敏捷宣言》^[12]中,包括重视个体和交互胜过流程和工具、可工作的软件高于详尽的文档、与客户协作并积极响应变化、以及持续交付并欢迎更改需求。

2.3.1 敏捷开发的意义

与传统互联网时代相比,智能物联网时代(AIoT)的设备成本功耗受限,专用性强。传统上追求性能和通用的计算芯片不再适用。需要根据细分领域的需求,软硬件件协同优化,深度定制领域专用处理器(XPU)。以满足特定场景对于芯片的成本,性能与能效需求。^[13]

这将使不同 AIoT 设备对芯片的需求差异化、专一化、碎片化。如果继续沿用传统的瀑布式开发流程,过长的芯片研发和上市时间显然无法满足海量 AIoT 设备的快速定制需要。

2.3.2 芯片设计与敏捷开发

传统的芯片设计流程包含一个固定的“代码冻结”阶段，在此阶段之后，代码修订受到严格限制。这就导致了后端物理设计团队的反馈不能及时有效地帮助前端架构团队改进设计。

而使用敏捷思想指导开发的芯片，在完成每一个功能点的编写后，前后端团队都会一起进行测试与验证。为快速改进设计提供了可能。

敏捷开发可大幅降低芯片的设计周期，并在市场需求变化时快速响应。通过敏捷开发，伯克利大学的研究团队成功在五年内开发并生产了 11 种不同的处理器芯片^[14]。

2.3.3 芯片行业的敏捷开发

在芯片设计领域中，敏捷开发意味着在设计初期就利用高级语言快速构建模拟器原型，软件的灵活性允许工程师快速尝试各种设计方案，尽早获得反馈，并迅速迭代设计。设计成熟后，敏捷开发流程允许团队在较短时间内将设计迁移到 FPGA 上进行验证和调试，FPGA 的高速和高准确性允许工程师在这一步运行完整的操作系统与测试程序。同时与客户紧密合作，确保芯片设计能满足客户需求。设计过程中，设计完成后，可以通过电子设计自动化（EDA）工具对芯片电路的面积、功耗和时序进行细致的分析。整个过程中，前端架构和后端物理团队同步协作，共同参与每次迭代的验证和优化，从而显著缩短了设计周期，提高了产品质量。

当芯片电路设计敲定后，就可以与晶圆厂对接，准备进行投片生产。

整个设计流程如 图 2.2 所示。其前三步都可以通过敏捷开发大幅提高效率与质量。



图 2.2 芯片开发流程

2.4 Chisel 语言在芯片设计中的优势及其实现原理

目前，最流行的两种硬件定义语言（HDL）是 Verilog 和 VHDL^[15]。硬件工程师通过 HDL 描述硬件电路的具体行为与结构。Verilog 与 VHDL 最初都是为硬件仿真而创建的，后来才被用于综合。这些语言缺乏抽象能力，使得组件难以在项目之间重复使用。

Chisel^[16] 是一种基于高级编程语言 Scala^[17] 的新型硬件构造语言（HCL）。由伯克利大学的研究团队于 2012 年推出。

Chisel 在 Scala 中提供了各类硬件原语的抽象，使得工程师者可以用 Scala 类型描述电路接口，以 Scala 函数操作电路组件。这种元编程可实现表现力强、可靠、类型安全的电路生成器，从而提高逻辑设计的效率和稳健性。而基于生成器的设计促进了模块在项目之间的重用。

需要说明的是，虽然 Chisel 具有许多传统硬件描述语言不具备的高级特性，但其还是一门硬件构造语言，而不是高层次综合。Chisel 代码在运行后会生成对应硬件的形式化描述，以寄存器传输语言灵活中间码（Flexible Intermediate Representation for RTL, FIRRTL）形式记录^[18]。

FIRRTL 随后被输入硬件编译框架(Hardware Compiler Framework, HCF) CIRCT^[19]。经过多次递降变换(lowering transformations)^[20], 逐级去除高层次的抽象, 最终输出优化后的底层 RTL 代码(如 Verilog)。使用 HCF 的好处之一是其内置了大量的优化器, 可进行常量传播、共用表达式合并、不可达代码消除或向专用集成电路(ASIC)与现场可编程门阵列(FPGA)提供针对性的优化^[21]。这些特点使工程师能够更专注于逻辑功能的设计, 从而显著提升设计效率与代码质量。

2.4.1 Chisel 的优点

开发高效

Chisel 语言的高效主要体现在四个方面:

- 信号整体连接: Chisel 丰富的类型系统和连接功能。允许工程师以组件化的方式声明和连接多个信号线, 并进行整体连接。从而简化了总线接口修改的工作流程。避免了在使用传统 HDL 开发时, 修改总线需要手动更新多个模块的端口而产生潜在错误。
- 元编程: Chisel 可以抽象出多份相似模块的共用部分, 通过使用参数化的硬件模板, 工程师可以创建出可复用的硬件库。有效减少了代码冗余。SystemVerilog 虽然提供类似的功能, 但仅用于验证^[22], 属于不可综合代码。
- 面向对象: Chisel 中可以将常用的电路组件封装成类, 并通过继承与派生的方式定义常用变体, 便于代码重用, 减少冗余。藉此, Chisel 内置了大量的预定义组件供工程师使用。以描述一个 32 位带使能端带复位的寄存器为例, 代码 2.1, 代码 2.2 表明, Chisel 实现不仅更加简短, 并且明确表示了这个寄存器的行为。而 Verilog 实现需要工程师阅读 Always 块的内容才能确定该寄存器的行为。

```
val delay1_r = RegEnable(io.in, 0.U(32.W), io.enable)
io.out := delay1_r
```

代码 2.1 Chisel 实现

```

reg [31:0] delay1_r;
assign io_out = delay1_r;
always @(posedge clock) begin
    if (reset) begin
        delay1_r <= 32'h0;
    end else if (io_enable) begin
        delay1_r <= io_in;
    end
end

```

代码 2.2 Verilog 实现

- 函数式: Chisel 的函数式特性将允许工程师使用 map 或 zip 算子实现电路级联的批处理操作, 可以简洁地描述数据流。如图 2.3 所示, 级联的算子操作可以直接对应生成后的电路。与 Verilog 的数据流建模相比, 工程师不再需要关心运算符优先级。

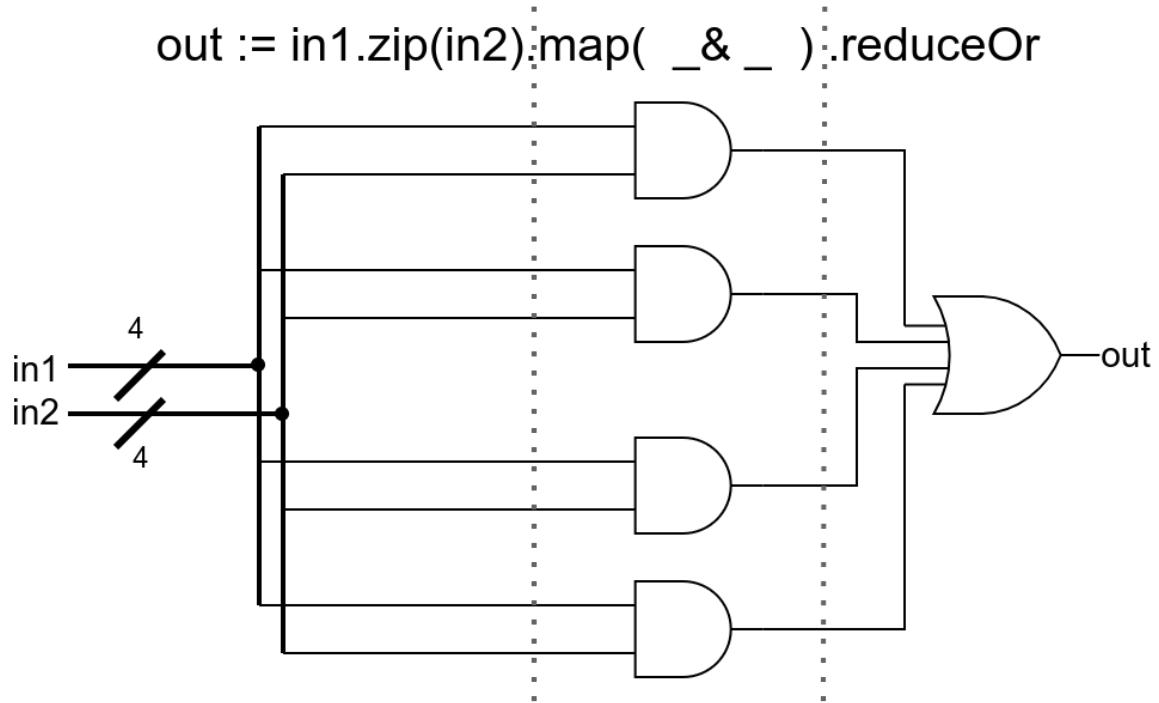


图 2.3 函数式操作与电路设计紧密对应

代码质量更优

余子濠等人 (2019) 的研究^[23] 指出。在 FPGA 上, 与经验丰富的专职工程师使用 Verilog 相比, 经训练的本科生使用 Chisel 的高级特性开发同样功能一致的缓存模块。无论能效、频率或资源占用, 均优于 Verilog 实现, 且代码可维护性更优。

表 2.2 资源占用与性能对比

类型	Verilog	Chisel
最高频率/MHz	135	154
功耗/W	0.77	0.74
LUT 逻辑	5676	2594
LUT RAM	1796	1492
触发器	4266	747
代码长度	618	155

在开发速度方面，专职工程师使用 Verilog，消耗至少 6 周完成开发。而另一位有 Chisel 经验的本科生则在 3 天内完成了所有工作，而且代码长度显著短于前者。

而在芯片整体设计方面，吕治宽（2022）的研究^[24]表明，使用两种语言开发同一微结构的 RISC-V 处理器。Verilog 的时间消耗是 Chisel 的 3 倍。

处理器微架构设计与实现

本文将首先实现一个单发射，简单流水线，顺序执行的基本 RV32E 扩展处理器芯片微架构。

为方便拓展，本微架构流水级之间使用异步定时方式通信，通过握手信号控制数据是否流动。每个流水级的行为只取决于自身和下游模块的状态，流水级均可以独立工作。通过这种设计，避免了对全局控制器的需求，进而简化添加指令和流水级的难度。也为探索乱序执行做好了准备。

在逻辑上，为了方便优化，本微架构分为以下几个基础单元：取指单元，译码单元，执行单元，访存单元，写回单元。处理器核的设计方案如图 3.1 所示。

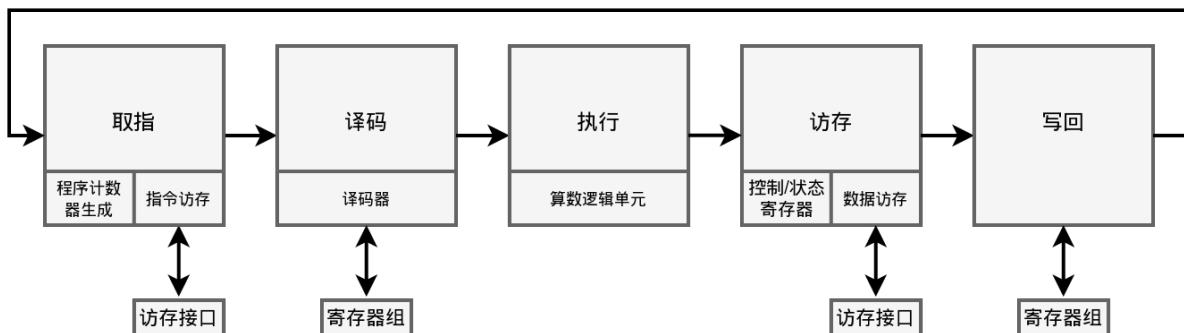


图 3.1 处理器核设计

片上系统的设计如图 3.2 所示。

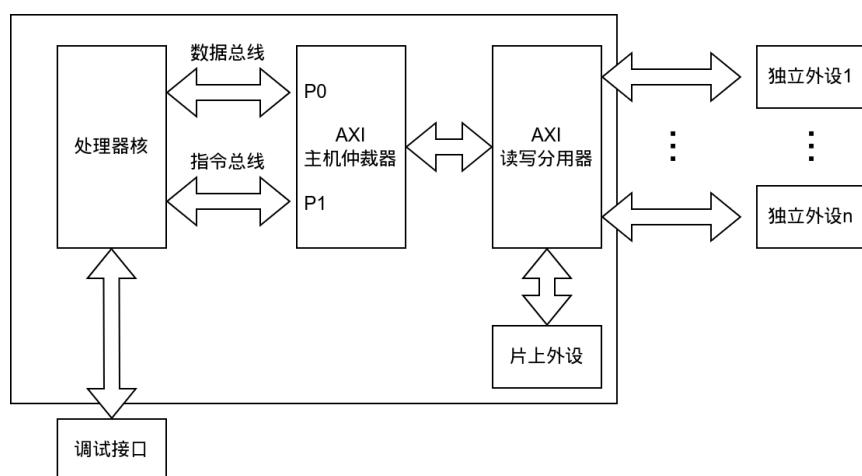


图 3.2 片上系统设计

3.1 信号握手

如图 3.3 所示，每个基础单元均可以根据自身情况产生“准备/有效 (Ready / Valid) ”握手信号。Valid 标识模块当前的数据 (Data) 是否有效，Ready 标识本模块是否能接收 Data 输入。

模块间遵循表 3.1 的语义进行传输握手。此外，为防止数据丢失，当模块表示当前数据有效后，除非数据成功传输，否则不能撤销有效信号。本级有效而下级忙的情况称为发生数据反压 (Back pressure)

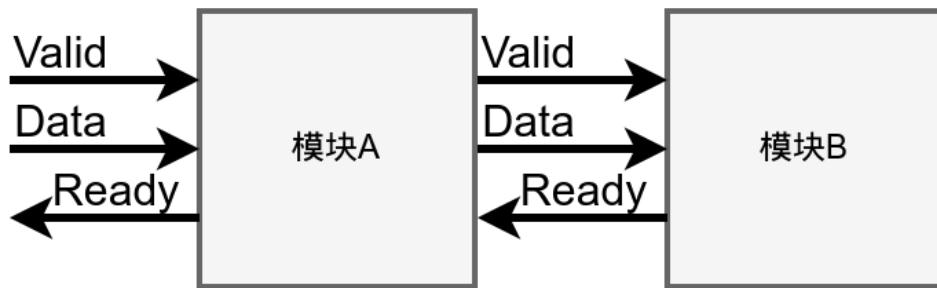


图 3.3 Ready / Valid 握手信号

表 3.1 握手信号行为说明

下级 Ready	本级 Valid	数据传输行为
有效	有效	数据成功传输至下级模块
有效	无效	当前数据无效 下级模块需等待
无效	有效	下级模块忙 本级需保持 Data 并等待
无效	无效	不发生数据传输

3.2 取指单元

取指单元的总体架构如图 3.4。

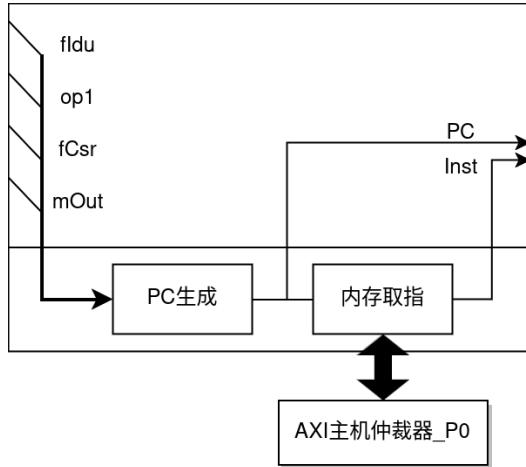


图 3.4 取指单元总体架构

取指单元根据上一条指令的译码、执行结果修改程序计数器 (PC)，并取出下一条指令。整个取值单元分成 PC 生成器及指令访存器。

3.2.1 PC 生成器

PC 生成器将根据当前处理器的状态以及指令来确定下一执行周期的程序计数器的值。目前，程序计数器生成器支持以下行为：

1. 顺序执行
2. 普通跳转（不切换特权级）
3. 暂停执行
4. 自陷指令——陷入
5. 自陷指令——返回

RISCV 架构支持比较两个通用寄存器的值并根据比较结果进行分支跳转 (B 系列指令)。无条件跳转 (J 系指令)，也支持在异常发生时进入异常处理程序并返回。PC 寄存器的结构如图 3.5 所示，其行为如表 3.2 所示。

表 3.2 PC 生成表

类型	add1	add2	next_pc	PC 寄存器
顺序执行	pc	4	sum_next_pc	允许写入
无条件跳转	pc	立即数	sum_next_pc	允许写入
条件跳转	pc	据结果选择	sum_next_pc	允许写入
异常陷入	无关	无关	mtVec	允许写入
异常返回	无关	无关	mePC	允许写入
反压暂停	无关	无关	无关	禁止写入

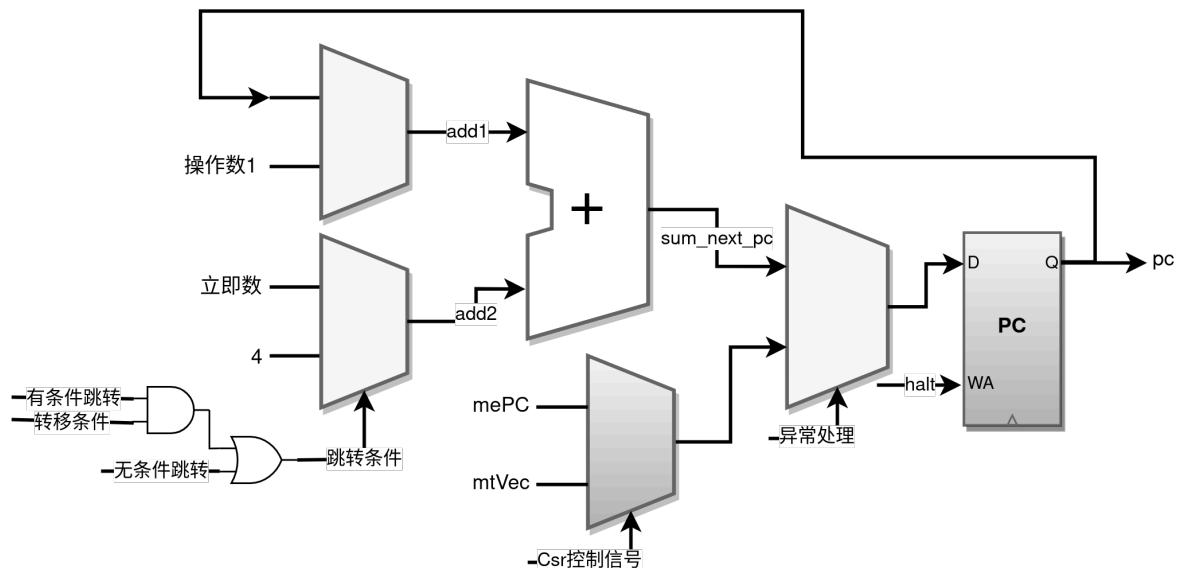


图 3.5 PC 生成器结构

3.2.2 指令访存器

指令访存器的任务是根据当前 PC，向处理器核内的 AXI^[25] 主机仲裁单元发起访存请求。其结构如图 3.6 所示。在复位后，指令访存控制器在 idle 状态等待有效的访存事务发生。收到访存请求后，按照表 3.3 的行为对取值器的输出端口进行操作。需要注意的是，该状态机的输出均来自寄存器。

由于 AXI 协议规定在读事务完成时不再保持数据有效信号。在读取完成后，指令执行中的若干周期，指令有效信号 将由状态机提供

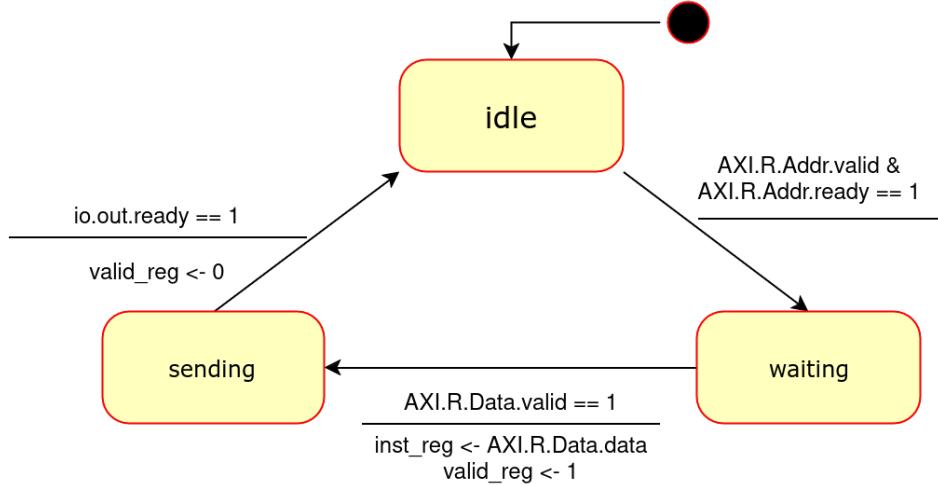


图 3.6 取指状态机

表 3.3 取指单元状态机转移说明

状态名	说明	离开条件
idle	当前未在访问指令存储器	存储器 AXI 读握手成功
waiting	读请求已发出。等待指令存储器回复	存储器回复数据
sending	向 CPU 下游部件提供指令码	下游部件汇报准备完成

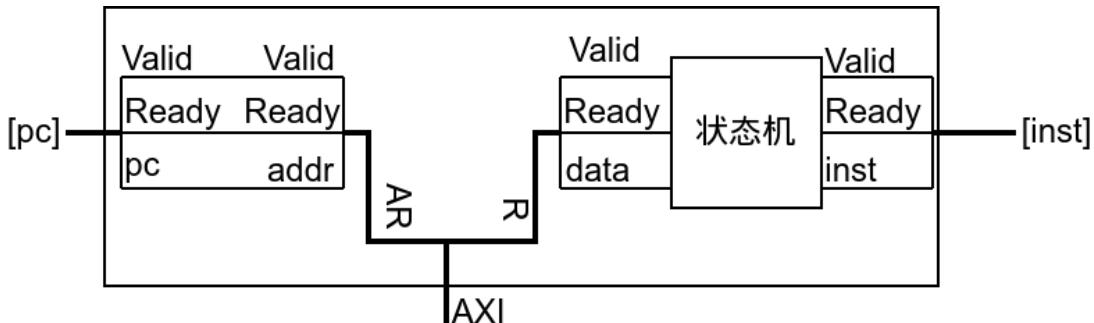


图 3.7 指令访存器结构

3.3 译码单元

译码单元的任务是根据当前指令，通过译码器产生各类控制信号；从寄存器组中读出数据。

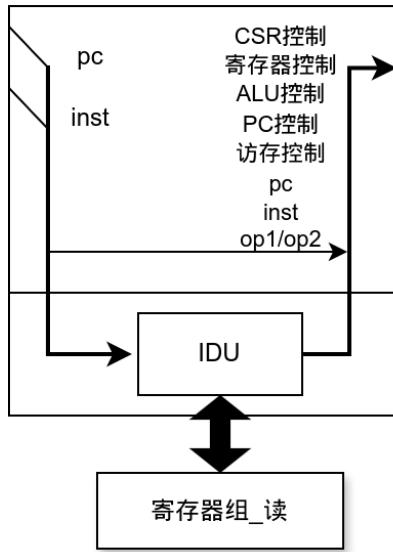


图 3.8 取指单元总体架构

3.3.1 译码器

译码器主要功能如下：

- 访问寄存器：如表 2.1 所示，RISC-V 指令集中，对于需要访问寄存器的指令，其寄存器编号均在指令的固定位置。为简化电路，译码器被设计为不对当前指令类型进行判断，固定将 24~20 与 19~15 段的内容视为寄存器编号，进行寄存器访问。
- 立即数产生与符号扩展：如表 2.1 所示，RISC-V 指令集中，对于具有立即数的指令，无论立即数长度，其符号位固定位 31 位。为简化电路，符号扩展被设计为固定使用 31 位做为符号进行扩展，各下级模块由解码结果决定是否使用。在后续设计中，将根据时序报告决定是否将本部件移动到执行单元。
- 控制信号生成：根据指令类型和具体指令操作。为下级模块提供控制信号。

【要解释所有控制信号的用途吗？】

3.4 执行单元

执行阶段是微架构数据流的核心，如图 3.9 所示。其任务是根据译码单元产生的控制信号，对操作数进行逻辑运算。

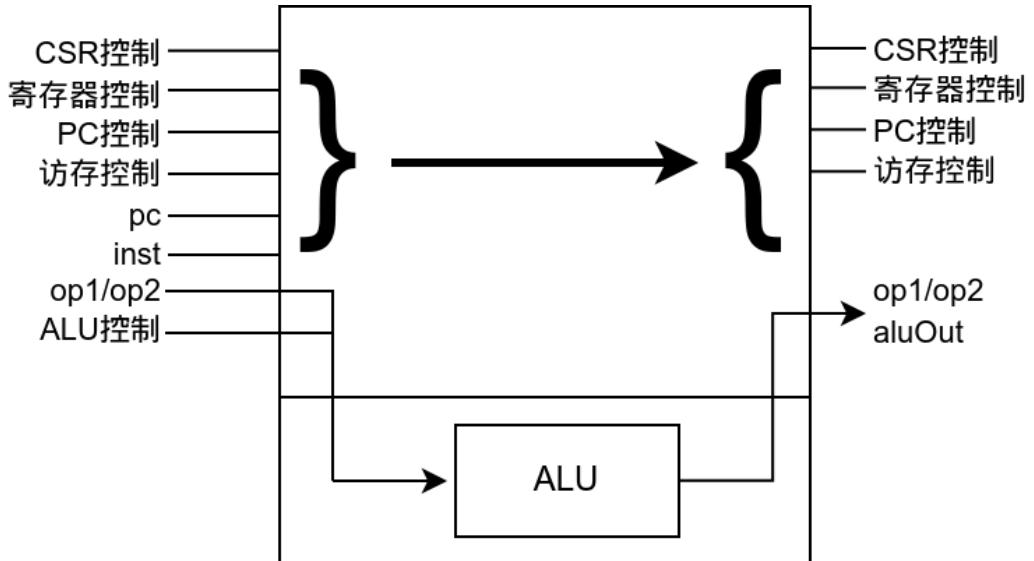


图 3.9 执行单元总体架构

3.5 访存单元

访存单元是为了装载 (LOAD) 与储存 (STORE) 指令设计的。整体结构如图 3.10 所示。为了提高流水线后各阶段的效率，设计时分析了 LOAD/STORE 指令的资源需求，决定将 CSR 的读写合并于本模块内。

根据当前执行的指令不同，模块顶层的 Ready/Valid 生成器会选择旁路或使用来自数据访存器的 Ready/Valid 信号。

在执行访存指令时 AluOut 输出的是目标内存地址。

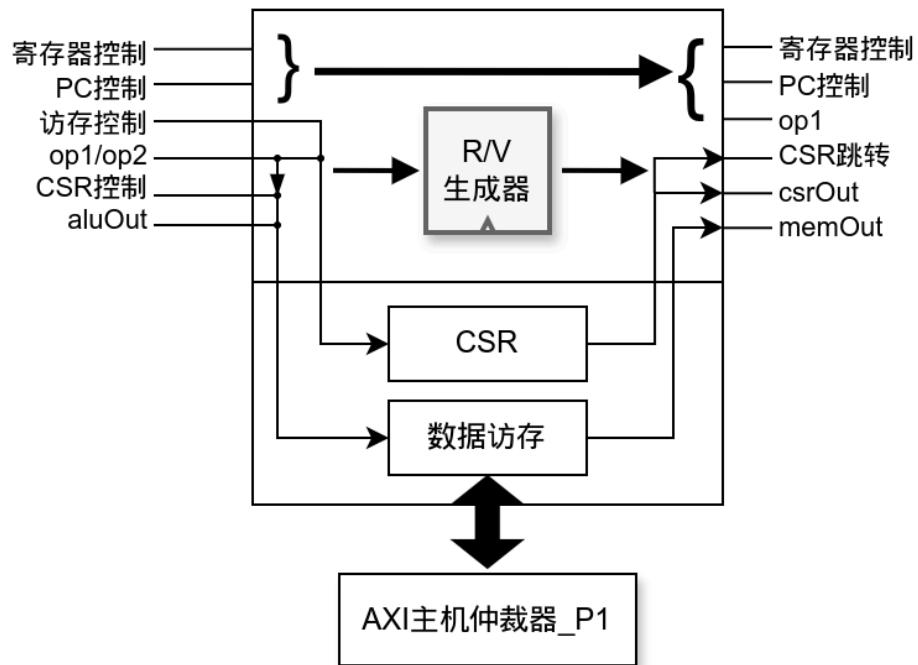


图 3.10 访存单元总体架构

3.5.1 数据访存器

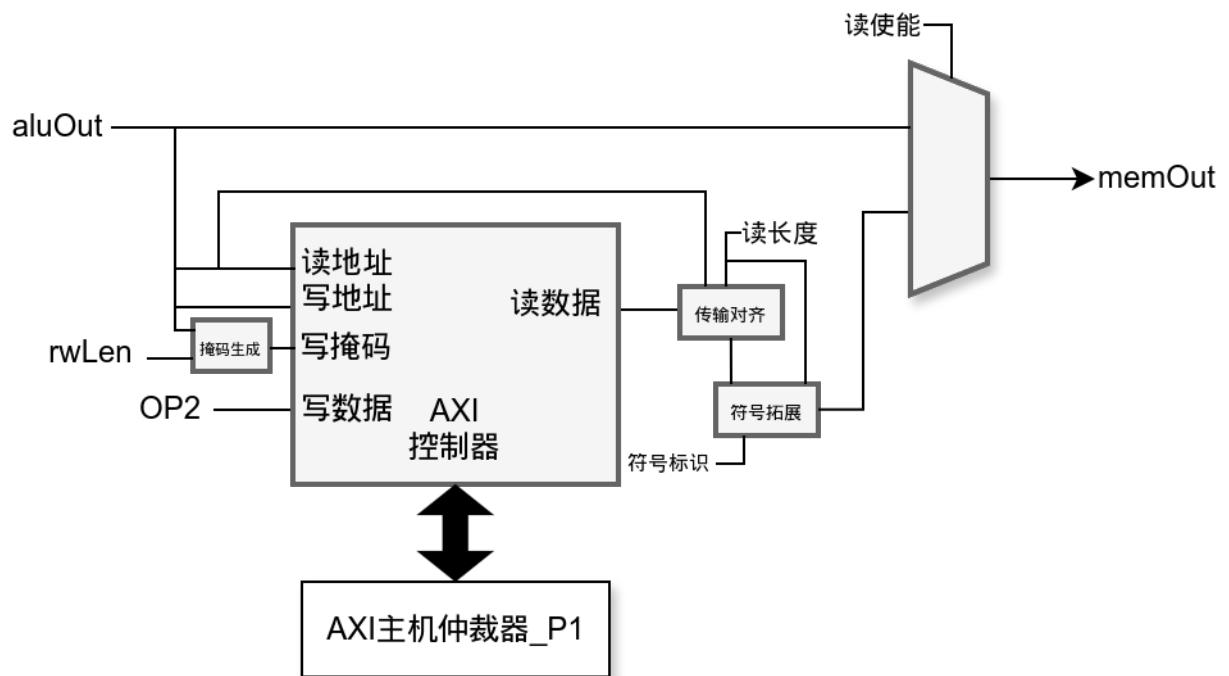


图 3.11 数据访存器架构

3.6 写回单元

写回单元负责将指令运行结果写入寄存器。

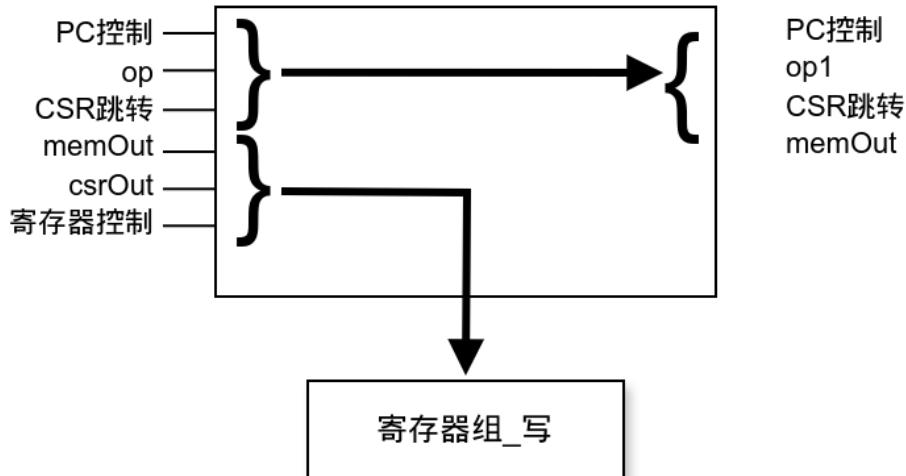


图 3.12 写回单元总体架构

3.7 Chisel 实现

本段节选了部分模块的具体实现代码，并说明 Chisel 对开发带来的便利与好处。

3.7.1 顶层模块

在 Verilog 中，顶层模块通常充斥着大量的模块间连线与冗长的实例化端口连接。人工编写不仅耗费时间，且容易出错。

如 代码 3.1 所示，通过 Chisel 的整体连接，可以使用少量代码完成安全可靠的连接。

```

class Top extends Module {
    val fetch      = Module(new Fetch)
    val decode     = Module(new Decode)
    val execute    = Module(new Execute)
    val memory     = Module(new Memory)
    val writeback  = Module(new Writeback)

    fetch.io.out   <-> decode.io.in
    decode.io.out  <-> execute.io.in
    execute.io.out <-> memory.io.in
    memory.io.out  <-> writeback.io.in
    fetch.io.in    <-> writeback.io.out
}

```

代码 3.1 顶层模块主要代码

3.7.2 寄存器组

由于线束 (Bundle) 的存在，声明寄存器组的 IO 接口变得更加容易，且当 Idu 或 Csr 增加新的控制寄存器组的信号时，不需要改动寄存器组的 IO 接口声明。

```

class Reg extends Module {
    val io = IO(new Bundle {
        val fIdu = Flipped(new IduRegBundle())
        val wValid = Input(Bool())
        val op1 = Output(UInt(32.W))
        val op2 = Output(UInt(32.W))
        val r10 = Output(UInt(32.W)) // for ebreak
        val mem = Input(UInt(32.W))
        val csr = Input(UInt(XLEN.W))
    })

    val regs = RegInit(VecInit(Seq.fill(16)(0.U(32.W))))
    io.r10 := regs(10) // for ebreak

    val wdata = Wire(UInt(XLEN.W))
    wdata := MuxLookup(io.fIdu.W.wdataSel, io.mem) {
        Seq(
            RegWdataSel.csr -> io.csr,
            RegWdataSel.mOut -> io.mem
        )
    }
    when(io.fIdu.W.wen === 1.U & io.wValid) {
        regs(io.fIdu.W.rd(3,0)) := wdata
    }

    io.op1 := regs(io.fIdu.R.rs1(3,0))
    io.op2 := regs(io.fIdu.R.rs2(3,0))

    regs(0) := 0.U
}

```

代码 3.2 REG 模块整体代码

3.7.3 译码单元

译码单元关键部分代码节选如下。根据指令类型的不同，alu 操作码由不同的子电路生成， 对应电路将指令码解析成不同的 bundle。再按照该类型指令的格式从 bundle 中提取所需信息，形成操作码。

```

io.toAlu.aluOp := MuxLookup(opType, AluOpTypes.unknown) {
    Seq(
        Opcodes.I -> opI(io.inst),
        ...
        Opcodes.B -> opB(io.inst),
    )
}
def opB(inst: UInt) = {
    val i = inst.asTypeOf(new Inst_B_bundle)
    val r = WireDefault(AluOpTypes.unknown)
    switch(i.fun3) {
        is("b000".U) { r := AluOpTypes.eq } //beq
        is("b001".U) { r := AluOpTypes.neq } //bne
        ...
        is("b101".U) { r := AluOpTypes.s_ge } //bge
    }
    r
}
def opI(inst: UInt) = {
    val i = inst.asTypeOf(new Inst_I_bundle)
    val r = WireDefault(AluOpTypes.unknown)
    when(opCode === Opcodes.LOAD) {
        switch(i.fun3) {
            is("b000".U) { r := AluOpTypes.add } //lb
            ...
            is("b101".U) { r := AluOpTypes.add } //lhu
        }
    }.elsewhen(opCode === Opcodes.JALR) {
        r := AluOpTypes.op1p4 //jalr
    }.otherwise {
        switch(i.fun3) {
            is("b000".U) { r := AluOpTypes.add } //addi
            is("b010".U) { r := AluOpTypes.s_less } //slti
            ...
            is("b111".U) { r := AluOpTypes.and } //andi
            is("b101".U) {
                switch(inst(31, 26)) { // srli, srai have unique I-type format
                    is("b000000".U) { r := AluOpTypes.sr } //srli
                    is("b010000".U) { r := AluOpTypes.s_sra } //srai
                }}}
        r
    }
}
class Inst_I_bundle extends Bundle {
    val imm11_0 = UInt((31 - 20 + 1).W)
    val rs1      = UInt((19 - 15 + 1).W)
    val fun3     = UInt((14 - 12 + 1).W)
    val rd       = UInt((11 - 7 + 1).W)
    val op       = UInt((6 - 0 + 1).W)
}

```

代码 3.3 译码单元代码节选

3.7.4 算数逻辑单元

算数逻辑模块（ALU）的代码如 代码 3.4 所示。通过使用枚举类型，代码可读性增加，ALU 操作一目了然，代码可读性显著提高。同时 Chisel 可以进行类型检查，防止使用错误的数据类型进行匹配。使用 MuxLookup 则有效提高了代码密度。

```
object AluOpTypes extends ChiselEnum {
    val sub, add, and, xor, neq, ge,
    ...
    less, op1p4, unknown = Value
}
class Alu extends Module {
    ...
    io.out := MuxLookup(io.fIdu.aluOp, 0.U(XLEN.W)) {
        Seq(
            AluOpTypes.sub  -> (op1 - op2),
            AluOpTypes.add  -> (op1 + op2),
            AluOpTypes.neq  -> (op1 /= op2),
            AluOpTypes.less  -> (op1 < op2),
            AluOpTypes.ge   -> (op1 >= op2),
            AluOpTypes.and  -> (op1 & op2),
            AluOpTypes.xor  -> (op1 ^ op2),
            ...
        )
    }
}
```

代码 3.4 ALU 模块 Chisel 代码节选

3.7.5 AXI 读分用器

AXI 读分用器的代码展现了 Chisel 对可变端口与函数式编程的支持。

在 Verilog 中，数目可变端口往往需要大量的 generate 语句块来实现具体操作，工程师容易写出可读性差的代码。Chisel 中，搭配函数式编程，可变数量端口的操作不会引起太多整体代码的变化。有助于工程师编写更加通用化的模块。代码 3.5 展示了本处理器总线部分的 AXI 读分用器实现。

```

object AXIDecoder{
    val maps = Seq(DMap("h8000_0000".U, "h8800_0000".U), // SRAM
                  DMap("hA000_03F8".U, "hA000_03F8".U), // UART
                  DMap("hA000_0048".U, "ha000_004C".U), // RTC
                  )
}
case class DMap(val begin: UInt, val end: UInt){
    def hit(addr: UInt):Bool = { addr >= begin && addr <= end }
}

class AXIReadDecoder(val n: Int) extends Module{
    val io = IO(new Bundle{
        val in = new AXIReadOnly
        val out = Vec(n, Flipped(new AXIReadOnly))
    })
    val decodedOH = Wire(UInt(n.W)) // One hot code
    val lastSelectedOH = RegEnable(decodedOH, 0.U(n.W), io.in.ar.arready)

    decodedOH := AXIDecoder.maps.map(_.hit( io.in.ar.araddr ) &&
                                    io.in.ar.arvalid )

    io.in.ar.arready := (decodedOH & io.out.map(_.ar.arready )).orR

    val lastSelected = OHToUInt(lastSelectedOH)
    io.in.r.exclude(_.rvalid) :<= io.out(lastSelected).r.exclude(_.rvalid)
    io.in.r.rvalid := ( lastSelectedOH &
                        io.out.map(_.r.rvalid)
                    ).orR

    for((out, selected) <- io.out.zip(decodedOH.asBools)){
        out.ar.arvalid := selected
        io.in.ar.exclude(_.arvalid) :>= out.ar.exclude(_.arvalid)
        out.r.rready <> io.in.r.rready
    }
}

```

代码 3.5 AXI 读分用器代码节选

AXI 读分用器 (AXIReadDecoder) 是一种一对多信号分配器件，当输入端请求 AXI 读操作时，根据操作的地址范围，将请求分配到合适的外设中，并记录当前选择。本实现还可以根据实例化时提供的参数 n 决定输出端口数量。

AXIDecoder.maps 按连线顺序保存了各个外设对应的地址。其独立于分用器主体代码外，便于在开发过程中增加新的外设。

decodedOH 使用独热码保存根据当前输入地址选择的输出端口号。它具有 n 位，对应 n 个端口。在计算 *decodedOH* 的代码中，*AXIDecoder.maps.map(...)* 会取出

AXIDecoder.maps 列表中的所有对象，逐一运行括号中指定的函数。该函数会将当前输入地址 araddr 送入列表的各个成员的 hit 函数，hit 函数会根据输入和自身保存的地址范围返回是否命中。各结果分别和 io.in.ar.arvalid 进行与操作，确保分用器只在输入地址有效时进行解码。

按照 AXI 标准，io.in.ar.arready 会向主机汇报所选的外设是否准备完成。其产生逻辑如下：1. 当前输入地址落在某个从机的地址范围内；2. 该从机准备完成；3. 当前输入地址有效；io.out.map(_.ar.arready) 会从所有输出中选出 arready 端口，并将其和 decodedOH 按位与。最终结果进行规约或操作，就可以生成符合语义的 io.in.ar.arready

例如，主机发出了向属于 UART 外设地址的 AXI 读取请求，当前 SRAM，UART 外设空闲，RTC 外设正忙。decodedOH 与 io.in.ar.arready 的生成过程如图 3.13 所示。

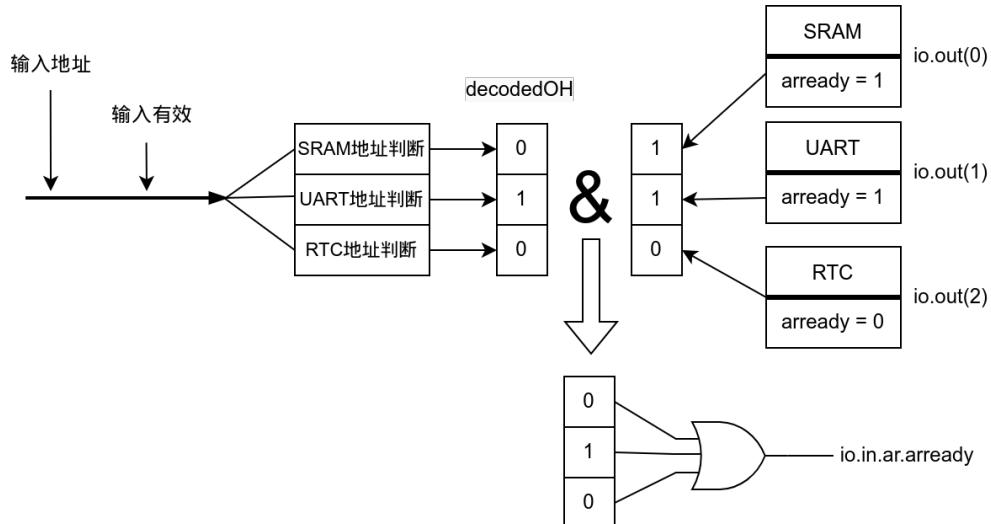


图 3.13 选择 UART 时独热码与有效信号的生成

通过使用 `zip` 算子，输出信号端口使能独热码逐一对与输出端口对象绑定，一起进入循环体。循环体内的代码则会将解码结果派发给各个输出接口所连接的外设。

验证

由于芯片生产费用高昂，所以在正式投片前，需要进行严格的测试，确保功能正确。本文采用软硬件协同的方式对芯片设计进行验证。首先在 Linux 下搭建了测试框架，然后通过实际编译器产生的测试例，使用自动化测试平台批量测试。

4.1 验证流程

验证平台使用 Makefile 脚本，KConfig 脚本，标准参考实现及以测试例组成。整体验证流程如 图 4.1 所示。

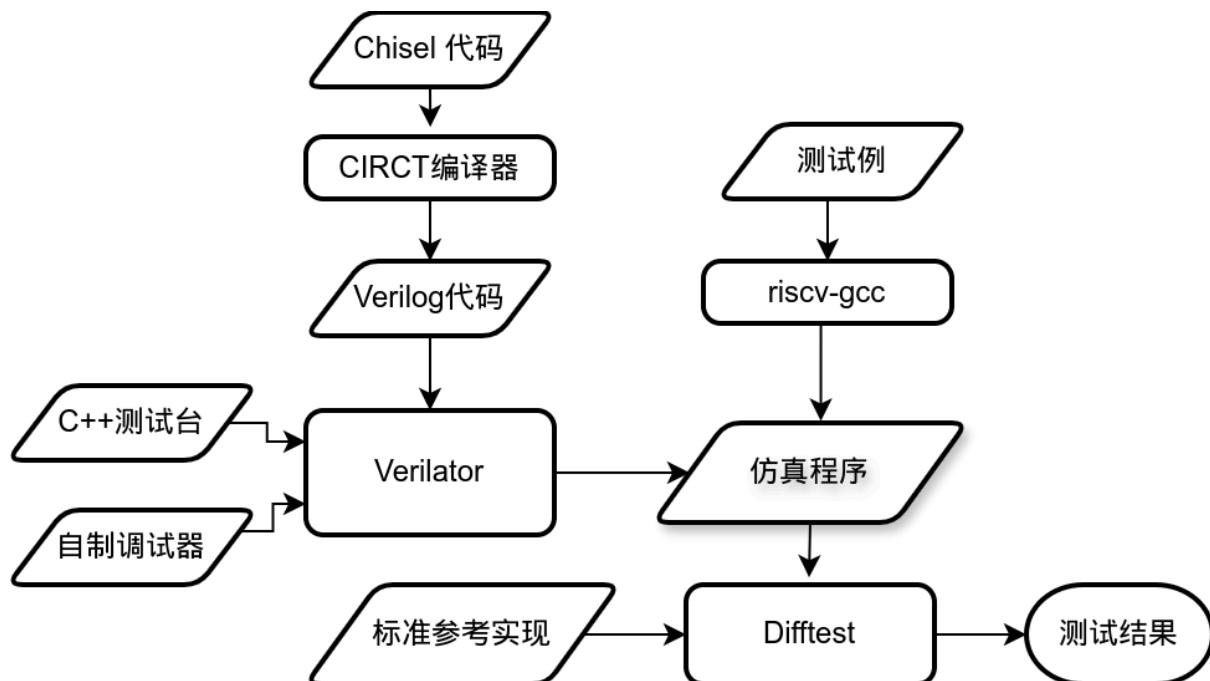


图 4.1 验证平台结构

测试开始前，Chisel 代码经过 CIRCT 编译为 Verilog 代码。并根据处理器芯片信号结构，设计 C++ 测试台提供运行期间所需要的输入信号。自制调试器可以记录调用栈，访存信息，在测试失败时辅助分析。通过 Verilator 将前述代码生成为周期等价的仿真程序。仿真分析可以读入编译好的测试程序，在执行时自动与标准参考实现进行比对。若比对结果没有差异，表明当前设计功能正确。

4.2 硬件验证环境

接下来对验证流程中的主要模块进行讲解

4.2.1 Difftest

Difftest 方法源于软件工程领域, 全称 Differential Testing(差分测试)。其核心思想是: 对于根据同一规范的两种实现, 给定相同的有定义的输入, 它们的行为应当一致^[26]。在测试环境中, 仿真环境每执行一条指令, 都会让标准 RISC-V 模拟器 *Spike* 也执行一遍。通过对比两侧执行后的通用寄存器与控制寄存器即可判断 Chisel 实现是否正确。

4.2.2 Verilator

Verilator 是一款基于周期的开源 Verilog 仿真工具, 可以将 Verilog 代码转化为 C++ 代码, 继而编译成可执行文件, 从而实现 Verilog 代码的仿真。它支持使用 C++ 开发测试台, 并能导出仿真期间的波形文件。

4.2.3 自制调试器

为了最大限度地加速排错, 作者在此期间也制作了一些专用调试工具。

- mtrace: 访存监视单元, 支持监视指定内存区间的读/写活动。使用方法如附录图 A.1 所示。
- 单指令执行: 为验证通用性, 模拟的存储器读写具有一个随机延迟。这导致单条指令的执行周期长度不确定。而传统的单步仿真不能适配这一点。为此, 作者在测试台中设计了按指令为单位的执行方式, 相关代码如附录图 A.2 所示。
- 调用栈分析: 为了确定客户程序在仿真器上的执行情况, 作者设计了调用栈分析模块。在仿真开始前, 读取输入程序的符号表, 确认所有函数的地址。当 CPU 进行跳转时, 根据目标地址和指令形式区分函数调用与函数返回, 生成调用栈。该模块可以识别尾递归。
- 波形分析器插件: 波形产生后, 指令只能以十六进制机器码的形式展示。不便于阅读。作者为波形分析器设计了一个插件。使用反汇编工具将机器码转换为易读的汇编指令。如图 4.3 , 图 4.2 所示。相关代码已发布^[27]。

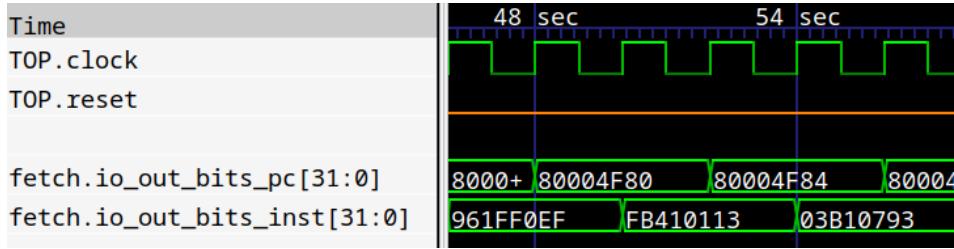


图 4.2 在波形软件中的原始显示

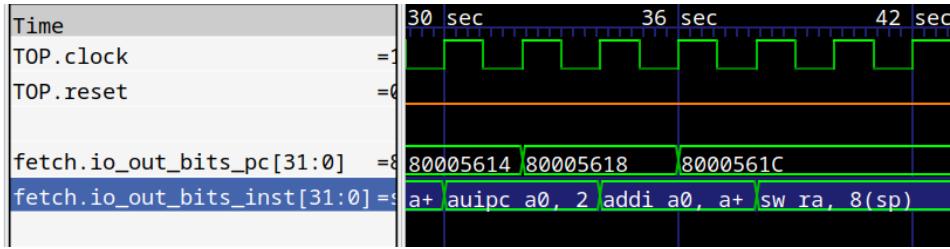


图 4.3 使用插件后可显示对应汇编

- 自动化性能分析：为提高仿真效率，作者设计了专用于性能分析的 Makefile 脚本，可自动将仿真器编译为分析模式，运行测试程序并呈现可读的性能报告。相关代码如附录图 A.3 所示。

4.2.4 仿真界面

仿真环境支持自动运行或用户手动控制。图 4.4 演示了仿真环境运行时的界面，在本演示中，用户通过 *si 5* 要求仿真环境执行 5 条指令后显示当前的寄存器信息 *info r*。仿真器可以成功显示执行指令的反汇编，并以可读的方式输出当期的寄存器信息。

```
(npc) si 5
0x20000060: ff 5f f0 6f jal    zero, 0x20000054
0x20000054: 00 b5 08 63 beq   a0, a1, 0x20000064
0x20000058: 00 05 20 23 sw    zero, 0(a0)
0x2000005c: 00 45 05 13 addi   a0, a0, 4
0x20000060: ff 5f f0 6f jal    zero, 0x20000054
(npc) info r
x0/$0 0x0          0      x1/ra 0x20000004 536870916
x2/sp 0x0          0      x3/gp 0x0          0
x4/tp 0x0          0      x5/t0 0x0          0
x6/t1 0x0          0      x7/t2 0x0          0
x8/s0 0x0          0      x9/s1 0x0          0
x10/a0 0xf000010  251658256 x11/a1 0xf000404 251659268
x12/a2 0xf000000  251658240 x13/a3 0x0          0
x14/a4 0x0          0      x15/a5 0x0          0
pc    0x20000054  536870996
(npc)
```

图 4.4 仿真界面演示

仿真环境亦可设置断点或直接查看内存，方便开发者观察特定代码区间执行前后的变化。

```
volatile int fib[40] = {1, 1};  
int main() {  
    int i;  
    for(i = 2; i < 40; i++) {  
        fib[i] = fib[i - 1] + fib[i - 2];  
    }  
    return 0;  
}
```

代码 4.1 简单斐波那契数列程序

以代码 4.1 中所示的简单斐波那契数列程序为例，通过反汇编找出循环体内写入 fib 数组的指令后，设置断点 `b 0x200000b8`，即可观察到该写入指令执行前后的内存变化，如图 4.5 所示。

```
[main.cpp:31 welcome] Wave: OFF  
[main.cpp:32 welcome] Build time: 10:40:50, Apr 2 2024  
Welcome to riscv32E-sysSoC!  
For help, type "help"  
(npc) b 0x200000b8  
Watchpoint 0 added  
(npc) c  
Watchpoint 0: $pc==0x200000b8  
Old data = 0  
New data = 1  
(npc) x 4 0xf0000a0  
Examining memory: 0xf0000a0  
0x0f0000a0 0x00000001 0x00000001 0000000000 0000000000  
(npc) si  
0x200000b8: 00 f4 22 23 sw a5, 4($0)  
Watchpoint 0: $pc==0x200000b8  
Old data = 1  
New data = 0  
(npc) x 4 0xf0000a0  
Examining memory: 0xf0000a0  
0x0f0000a0 0x00000001 0x00000001 0x00000002 0000000000  
(npc) █
```

图 4.5 断点调试

4.3 软件验证环境

4.3.1 抽象计算机

鉴于本文所设计的处理器是 RISC-V 架构，而整体开发环境搭建在 X86_64 平台中。因此，为了验证处理器功能，需要使用交叉编译器，生成 RISC-V 的目标代码。

交叉编译指的是在当前编译平台下，编译能运行在其他平台下的程序。本文使用 *riscv64-linux-gnu-gcc* 通过指定 *march=rv32e_zicsr* 约束编译器只生成 RV32E 指令。

由于当前设计的处理器性能较弱，且只能执行 RISC-V 的核心部分，无法运行现代操作系统。本文使用了南京大学提供的 abstract-machine^[28]（抽象计算机简称：AM）框架做为裸机运行时环境。

AM 将计算机功能分为以下层级

- 图灵机：仅限于纯计算任务（TRM）
- 冯诺伊曼机：含有复杂的输入输出扩展（IOE）
- 批处理系统：含有上下文管理能力(CTE)
- 分时多任务：含有虚拟内存管理功能(VME)
- 多处理机：含有多个硬件线程（MPE）

通过实现更加复杂的 CPU 功能，可以开启更多的 AM 层级，进而运行更多更复杂的程序。而当 CPU 功能较弱时，则可以关闭一些 AM 层级，使得系统可以继续运行基础应用程序。CPU 提供硬件功能，AM 提供运行时环境。

客户程序使用 AM 提供的统一接口，面向抽象的计算机体系进行编程，AM 向客户程序屏蔽底层细节。在本架构中，只要完成 AM 对某个环境的适配，为 AM 编写的程序就可以运行在其上。如图 4.6 所示。

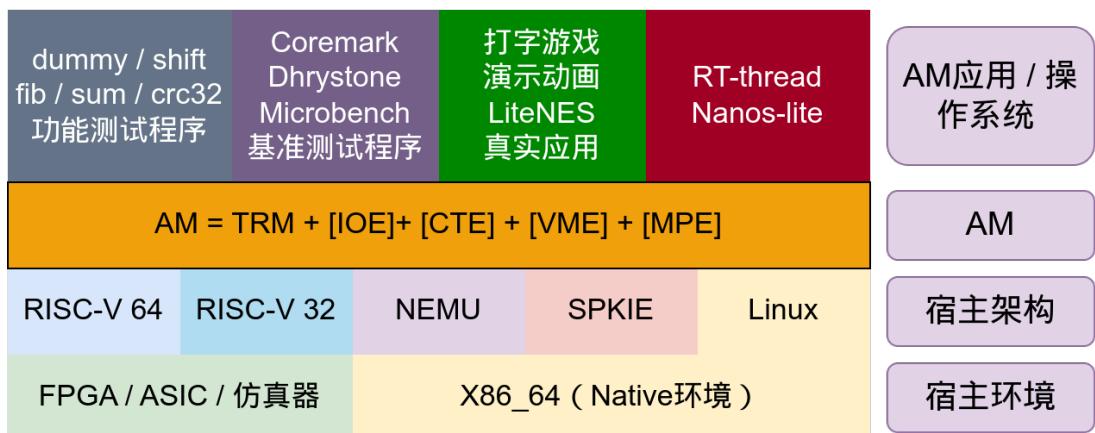


图 4.6 AM 整体层次

4.3.2 适配抽象计算机

通过在硬件中实现某种机制，然后在 AM 中使用该机制填充相应的 API。就可以完成 AM 对该架构的一部分支持。AM 大部分由 C 语言写成。

由于 CPU 复位完成后不具备 C 语言的运行条件，需要使用汇编准备堆栈后才能进入 AM。启动脚本如 代码 4.2 所示。由以下几个步骤组成：

1. 清空帧寄存器
2. 设置栈指针到链接脚本指定的栈起始区
3. 跳转到 _trm_init。进入 C 环境

```
.section entry, "ax"
.globl _start
.type _start, @function

_start:
    mv s0, zero
    la sp, _stack_pointer
    jal _trm_init
```

代码 4.2 start.S

在进入 C 环境后，AM 的需要完成以下操作

1. 实现基本字符输出函数 putch
2. 初始化串口输出
3. 调用客户程序主函数
4. 根据主函数返回值，向硬件发出停机指令。

```

#define DEVICE_BASE 0x10000000
#define SERIAL_PORT (DEVICE_BASE + 0x00000000)

void uart_init(){
    outb(SERIAL_PORT + 3, 0b10000011); // 8N1 with Divisor Latch Access
    outb(SERIAL_PORT + 1, 0x00); // No interrupt
    outb(SERIAL_PORT + 0, 0x21); // Set Divisor
    outb(SERIAL_PORT + 3, 0b00000011); // 8N1 without Divisor Latch Access
}
void putch(char ch) {
    while((inb(SERIAL_PORT + 5) & 0b01000000) == 0){} // busy
    outb(SERIAL_PORT, ch);
}
void halt(int code) {
    asm volatile("mv a0, %0; ebreak" : :"r"(code));
    while (1); // should not reach here
}
void _trm_init() {
    uart_init();
    int ret = main(mainargs);
    halt(ret);
}

```

代码 4.3 trm.c

4.3.3 适配存储结构

由于程序所在的 ROM 在运行时是只读的，因此，为了支持全局变量的写入操作。需要在系统启动时将数据段(data, bss)从 ROM 加载到可读写的 RAM 中。为了实现这一点，需要在链接时标记各个段的加载地址 (LMA) 与运行地址 (VMA)。并编写 bootloader 程序完成初始化时的数据加载操作。

本设计的内存布局如 图 4.7 所示，对应的链接脚本和 bootloader 程序节如下。

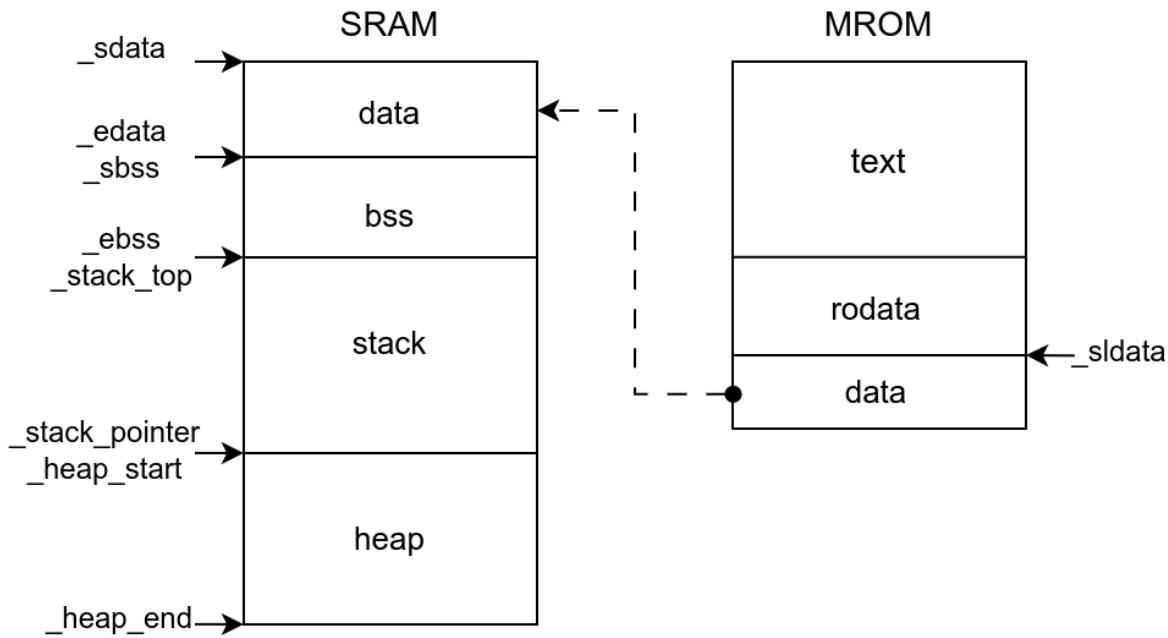


图 4.7 内存布局

```

SECTIONS {
    _sldata = LOADADDR(.data);
    .text : {
        *(entry)
        *(.text*)
    } > MROM
    .rodata : {
        *(.rodata*)
    } > MROM
    .data : ALIGN(4) {
        _sdata = .;
        *(.data*)
        _edata = .;
    } >SRAM AT> MROM
    .bss : {
        _sbss = .;
        *(.bss*)
        _ebss = .;
    } >SRAM
    .heap_stack : {
        _stack_top = ALIGN(4);
        . = _stack_top + 0x1200;
        _stack_pointer = .;
        _heap_start = ALIGN(8);
        _heap_end = ORIGIN(SRAM) + LENGTH(SRAM) - 1;
    } >SRAM
}

```

代码 4.4 链接脚本节选

```

_loader:
    la a0, _sldata
    la a1, _sdata
    la a2, _edata
0:
    beq a1, a2, 1f
    lw a3, 0(a0)
    sw a3, 0(a1)
    addi a0, a0, 4
    addi a1, a1, 4
    j 0b
1:
    la a0, _sbss
    la a1, _ebss
2:
    beq a0, a1, 3f
    sw x0, 0(a0)
    addi a0, a0, 4
    j 2b
3:
    ret

```

代码 4.5 bootloader 节选

4.4 仿真分析

4.4.1 运行测试

基础测试包括基本的算数运算、跳转、加载储存等指令。使用 C 语言编写，测试列表见表 4.1

表 4.1 基本测试程序

测试程序名称	测试内容	测试程序名称	测试内容
add	普通加法	add-longlong	长加法
bit	位操作	bubble-sort	冒泡排序
crc32	循环冗余校验	div	软除法
dummy	空函数跳转	fact	阶乘
fib	斐波那契数列	goldbach	歌德巴赫猜想
hello-str	字符串输出	if-else	分支语句
leap-year	闰年	load-simple	读内存
load-store	读写内存	matrix-mul	矩阵乘法
max	大小比较	mersenne	梅森数
min3	排列	mov-c	写后读
movsx	32 位写后读	mul-longlong	长乘法
pascal	帕斯卡算法	prime	素数
quick-sort	快速排序	recursion	递归
select-sort	选择排序	shift	移位计算
shuixianhua	水仙花数	simple	1+1
string	字符串操作	sub-longlong	长减法
sum	求和	switch	switch 语句
to-lower-case	小写转换	unalign	不对齐访存
wanshu	完数		

由图 4.8 可见，CPU 顺利通过基本测试

```
--- Sim Stopped ---
add-longlong add bit bubble-sort crc32 div dummy fact fib goldbach
rsion select-sort shift shuixianhua simple string sub-longlong su
[ 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-simple] 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!
[ simple] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
coa@laptop ~/syx-workbench/am-kernels/tests/cpu-tests2 <bus>
$
```

图 4.8 基本测试结果

CoreMark^[29] 是一个业界用来衡量嵌入式系统中 CPU 性能的测试程序。它主要测试 CPU 的整数性能，含有列表处理、矩阵操作、状态机和 CRC 等几个测试子项。
本文设计的 CPU 可以正确运行 CoreMark，并在合理的时间内完成测试。

```

***** SIM *****
FTrace load 60 symbols
[log.cpp:13 init_log] Log is written to stdout
[load_img.cpp:27 load_img] Using image file /home/coa/ysyx-workbench/am-kernels/benchmarks/coremark/
build/coremark-riscv32e-ysyxsoc.bin, size = 4808
[cpu.cpp:99 reset] Top Reset Released @10
[cpu.cpp:102 reset] CPU Reset Released @20
[difftest.cpp:38 init_difftest] Difftest is disabled. ref_so_file is ignored
[main.cpp:31 welcome] Wave: OFF
[main.cpp:32 welcome] Build time: 11:43:49, Apr 3 2024
Welcome to riscv32E-ysyxSoC!
Running CoreMark for 1 iterations
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 0
Iterations : 1
Compiler version : GCC13.2.0
seedcrc : 0xE9F5
[0]crclist : 0xE714
[0]crcmatrix : 0x1FD7
[0]crcstate : 0x8E3A
[0]crcfinal : 0xE714
Finised in 0 ms.
=====
CoreMark PASS

--- Got stop request ---
- :0: Verilog $finish
[cpu.cpp:173 cpu_exec] NPC: HIT GOOD TRAP at pc = 0x80002448
[cpu.cpp:115 cpu_statistic] host time spent      = 28 s
[cpu.cpp:117 cpu_statistic] total instructions   = 843206
[cpu.cpp:118 cpu_statistic] total cycles        = 43044277
[cpu.cpp:119 cpu_statistic] total ticks          = 0
[cpu.cpp:122 cpu_statistic] average CPI         = 51.05 c/i
[cpu.cpp:125 cpu_statistic] simulation frequency = 29128 inst/s
[cpu.cpp:127 cpu_statistic] simulation frequency = 1486984 cycle/s
--- Sim Stopped ---
/home/coa/ysyx-workbench/npc/build/ysyxSoCFull -b --elf    28.92s user 0.01s system 99% cpu 28.959 t
otal

```

图 4.9 CoreMark 测试结果

原型验证

5.1 片上系统

为了实现包括输入输出在内的完整功能，需要将单一处理器核封装到 SoC (System on Chip, 片上系统) 中。SoC 不仅仅只包含一个处理器，还有诸多的外围设备，以及连接处理器和外围设备之间的总线。本文使用 ysyxSoC 框架^[30]。其主要结构如图 5.1 所示

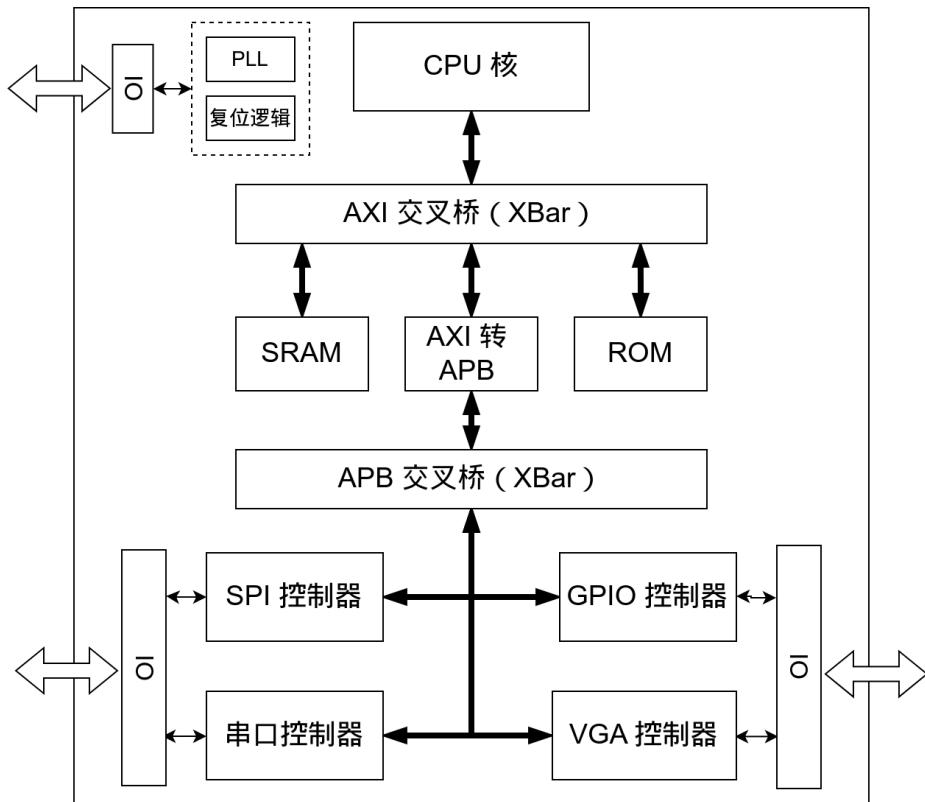


图 5.1 ysyxSoC 架构

SoC 整体架构由 CPU 核，总线，外设接口，复位与时钟四部分组成。

CPU 核通过 AXI 总线与片上外设接口进行数据交互，AXI 交叉桥将根据访存地址将请求转发到不同的外设，直接与 AXI 交叉桥相连接的是高速外设。对于低速外设，AXI 总线通过 AXI 转 APB 桥接器在转换成 APB 协议后连接到 APB 交叉桥。由 APB 总线负责低速外设，这些外设通过片上控制器经 IO 连接到片外外设中。

5.1.1 外设地址空间

目前主要有两种方式来实现CPU核与外设的通信。第一种I/O编址方式是端口映射I/O(PIO), CPU使用专门的I/O指令对设备进行访问,并把设备的地址称作端口号。第二种方式是内存映射IO(MMIO)。外设将自己的设备寄存器做为内存地址提供。当需要访问外设寄存器时, CPU核读取或写入该地址,通过总线桥将指令转发到对应的外设中。

由于PIO把端口号作为I/O指令的一部分。指令集为了兼容已经开发的程序,端口号是只能添加但不能修改的。这意味着,端口映射I/O所能访问的I/O地址空间的大小,在设计I/O指令的那一刻就已经决定下来了。随着设备越来越多,功能也越来越复杂,I/O地址空间有限的端口映射I/O已经逐渐不能满足需求了。因此,目前的主流设计均采用内存映射IO的方式来实现设备交互。

ysyxSoC的MMIO映射如表5.1所示。

表5.1 MMIO映射

设备	地址空间
GPIO	0x1000_2000~0x1000_200f
MROM	0x2000_0000~0x2000_0fff
UART	0x1000_0000~0x1000_0fff
SPI	0x1000_1000~0x1000_1fff
SRAM	0x0f00_0000~0x0f00_1fff
VGA	0x2100_0000~0x211f_ffff

5.2 现场可编程门阵列

现场可编程门阵列(FPGA),是一种半定制的集成电路,其具有高度可编程性,并且接近专用集成电路(ASIC)的特点。其经常用于在流片前的验证阶段中。与微控制器(MCU)不同,FPGA不含固定的逻辑。需要开发者使用硬件描述语言设计电路后才会呈现具体电路的行为。

本文选择了基于Altera EP4CE10的野火征途开发板。其具有1万组逻辑元素,179个可编程IO口。

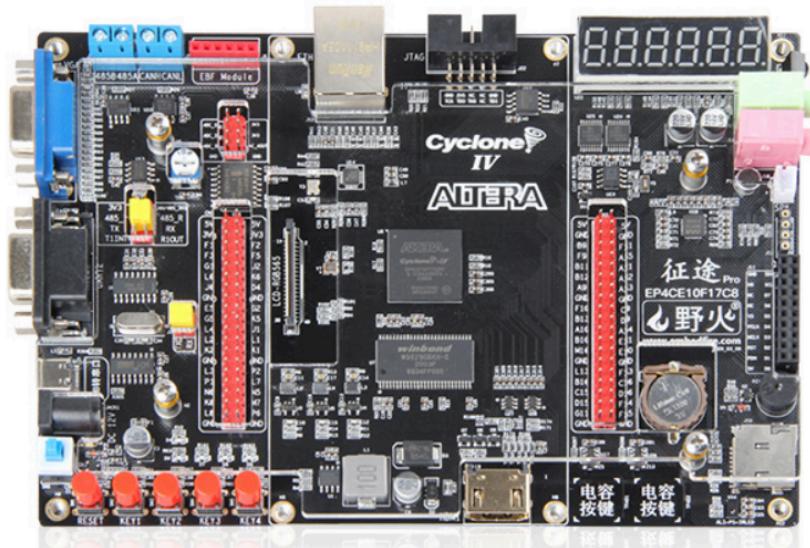


图 5.2 开发板实物图

5.3 GPIO 控制器

5.3.1 寄存器设计

表 5.2 GPIO 控制器寄存器表

名称	地址	宽度	访问	说明
Led	0x0	32	只写	开发板上的 LED 灯
Key	0xC	32	只读	开发板上的按键

5.3.2 主体代码

```

class gpioChisel extends Module {
    val io = IO(new GPIOCtrlIO)
    val addr = io.in.paddr(3,0)
    val led_reg = RegInit("hF".U(4.W))
    val keys_reg = Reg(UInt(6.W))

    io.in.prdata := 0.U
    io.in.pready := 0.U
    io.gpio.led := led_reg

    when(io.in.psel & io.in.pwrite & io.in.penable){
        switch(addr){
            is("h0".U){ led_reg := io.in.pwdata(3,0) }
        }
        io.in.pready := 1.B
    }
    when(io.in.psel & ~ io.in.pwrite){
        when(~ io.in.penable){
            switch(addr){
                is("hC".U){ keys_reg := io.gpio.keys }
            }
        }.otherwise{
            io.in.prdata := keys_reg
            io.in.pready := 1.B
        }
    }
}

```

代码 5.1 GPIO 控制器代码

5.3.3 测试程序

通过 MMIO，CPU 核上运行的程序可以直接控制 LED 灯。通过编写移位程序，可编写基础的流水灯测试程序。

```

_start:
    li a4, 0x10002000
    li a0, 0x00000001
shift:
    srl a1, a0, 4
    slli a0, a0, 1
    or a0, a0, a1
    not a3, a0
    sw a3, 0(a4)
    jal wait
    j shift
wait:
    li t0, 0x7fff
0:
    addi t0, t0, -1
    nop
    bne t0, zero, 0b
    ret

```

代码 5.2 流水灯测试代码

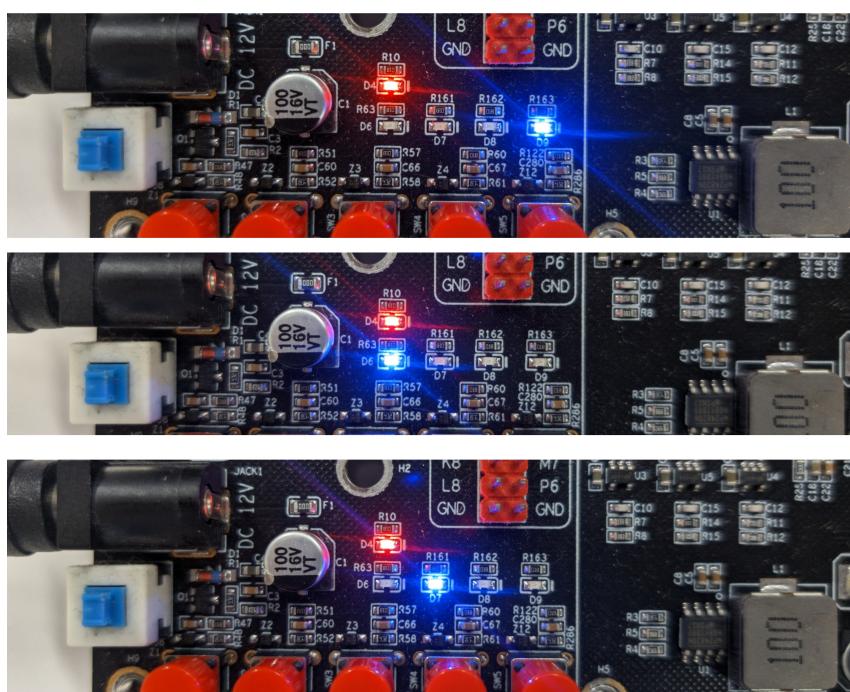


图 5.3 流水灯测试

5.4 UART 控制器

5.4.1 寄存器设计

表 5.3 UART 控制器寄存器表

名称	地址	宽度	访问	说明
发送	0x0	8	只写	发送缓冲区
中断控制	0x1	8	读写	中断使能与屏蔽
串口配置	0x3	8	读写	设置串口的数据格式
线路状态	0x5	8	读写	获取或清除传输状态

5.4.2 测试程序

结合 AM 提供的支持，程序可直接使用 C 语言标准库输出到串口。

```
int printf(const char *fmt, ...){  
    char buffer[1024];  
    va_list ap;  
    va_start(ap, fmt);  
    int ret = vsprintf(buffer, fmt, ap);  
    va_end(ap);  
    putstr(buffer);  
    return ret;  
}  
  
int main(){  
    printf("Yanglin Xun from Shaoguan university\n");  
    printf("1234:%s\n", "abcd");  
}
```

代码 5.3 UART 测试代码节选

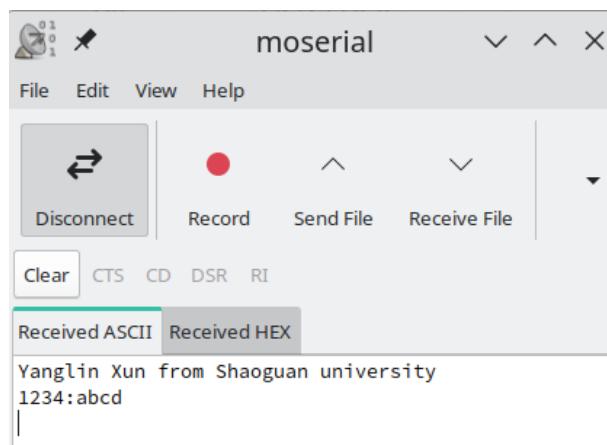


图 5.4 串口助手接收 UART 数据

总结与展望

6.1 总结

本研究成功完成了基于 Chisel 的精简指令集微处理器设计，该处理器具备 RISC-V 指令集的模块化、可扩展特性，适应物联网应用的多样化需求和安全性要求。并通过运行测试程序和跑分程序，证实了所设计处理器的正确性和有效性。

在此基础上，又成功实现了 SoC 化和对 FPGA 的支持，开发了 GPIO 和 UART 控制器，详细规定了寄存器设计、主体代码以及测试程序，确保处理器与外设间的协同工作。并完成了板级验证。能实现基本输入输出、硬件线程切换、图灵等价，几倍良好的可拓展性。

借助 Chisel 的抽象化和敏捷开发的特性。快速地完成了处理器的整体设计与 SoC 接入。

主要内容列举如下：

1. 通过阅读大量国内外文献，对 RISC-V 指令集架构的设计与发展现状进行了分析。并对比了其和其他精简指令集架构的优缺点。阅读各类相关的英文文档以确定处理器核的实现与标准文档一致。
2. 介绍了 Chisel 语言，敏捷开发对芯片设计领域的好处。着重介绍了 Chisel 语言如何通过提供丰富硬件原语抽象、支持类型描述电路接口以及使用函数操作电路组件，实现电路生成器的高效、可靠与类型安全性，从而显著提升逻辑设计效率与稳健性。
3. 搭建了软件测试平台，通过 Verilator 与 DiffTest。仔细验证了处理器功能与行为的正确性，通过运行多种测试程序，确保处理器设计可以在各类工作负载下正常运行。
4. 完成了仿真环境到 FPGA 上板验证的迁移。以及对外设模块的仿真测试。学习了仿真环境与 FPGA 的不同，并掌握了基础的 FPGA 开发技能，包括 IP 核调用，添加引脚约束，简单时序约束，片上逻辑分析仪的使用，通过调试器对 FPGA 进行编程及固化。

6.2 展望

虽然本文实现了基于 Chisel 的 RV32E 指令集的处理器设计。搭建了测试平台并完成上板验证，达到了期望的研究效果。但任存在诸多缺陷，还需要进一步的学习、设计与调试。

主要缺陷内容列举如下：

1. 处理器核不支持外部中断，导致所有功能必须通过轮询实现，在某些情况下会浪费处理器计算能力，并导致外部事件不能及时被处理。
2. 处理器核为多周期非流水线设计，单一模块工作时其他模块只能等待。导致从译码单元到回写单元的时序路径极长，不利于提升处理器各模块的利用率与整体时钟频率。
3. 无缓存功能，造成 CPU 每次在存储数据时都必须等待外部存储器响应，降低了运行效率。且不支持 MMU，导致无法运行现代操作系统（如 Linux）。
4. 截至本文初稿完成，SoC 中的 Flash 存储芯片控制器一直没有通过调试。导致上板时只能运行小于 4KB 的程序，极大地限制了能进行的演示。导致上板后只能执行部分测试。造成了上板后的测试数量远少于仿真分析环节的测试。
5. 译码器的设计仍有优化空间。

在今后的工作中，我将继续完成本设计中还未完善的地方，尝试为处理器搭建流水线和 Cache，进一步提升性能。将本设计拓展成真正完善而可用的处理器。并探索如乱序执行，多发射，分支预测等高级体系结构的内容。争取启动 Linux 等实际操作系统。

参考文献

- [1] 徐艳茹, 刘继安, 解壁伟, 等. 科教融合培养关键核心技术人才的理路与机制——OOICCI 芯片人才培养方案解析[J]. 高等工程教育研究, 2023(1): 20–26, 43.
- [2] 雨前顾问, 安谋科技. 2023 年中国大陆集成电路产业人才供需报告[R/OL]. <http://www.by-consulting.cn/home/projects/detail/id/10>.
- [3] 包云岗, 孙凝晖, 张科. 处理器芯片开源设计与敏捷开发方法思考与实践[J]. 中国计算机学会通讯, 2019, 15(10): 42-48.
- [4] HENNESSY J L, PATTERSON D A. A New Golden Age for Computer Architecture[J/OL]. Communications of the ACM, 2019, 62(2): 48-60[2024-03-07]. <https://dl.acm.org/doi/10.1145/3282307>. DOI:10.1145/3282307.
- [5] 澎湃新闻. 倪光南院士: RISC-V 是中国最受欢迎 CPU 架构, 不受垄断制约[EB/OL]. (2023). https://www.thepaper.cn/newsDetail_forward_22141048.
- [6] 快科技. 院士倪光南: 发展基于 RISC-V 架构的 DSA 服务器是中国机遇! 或能对 X86 服务器替代[EB/OL]. (2024). https://www.sohu.com/a/764075353_639898.
- [7] 微五科技. “芯征程”, 芯辉煌! “港华芯”销量突破 100 万片[EB/OL]. (2023) [2024-03-11]. http://www.chinafive.com.cn/site/news_details/143.
- [8] ASANOVIĆ K, AVIZIENIS R, BACHRACH J, 等. The Rocket Chip Generator[R/OL]. (2016-04). <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [9] 石亚琼. 睿思芯科推出基于 RISC-V 的 64 位可编程终端 AI 芯片 Pygmy[EB/OL]. (2018)[2024-03-11]. <https://www.36kr.com/p/1722975698945>.
- [10] PATTERSON D A, DITZEL D R. The case for the reduced instruction set computer[J]. ACM SIGARCH Computer Architecture News, 1980, 8(6): 25-33.
- [11] WATERMAN A S. Design of the RISC-V instruction set architecture[M]. University of California, Berkeley, 2016.
- [12] FOWLER M, HIGHSMITH J, OTHERS. The agile manifesto[J]. Software development, 2001, 9(8): 28-35.

- [13] 包云岗, 常轶松, 韩银和, 等. 处理器芯片敏捷设计方法: 问题与挑战[J]. 计算机研究与发展, 2021, 58(6): 1131-1145.
- [14] LEE Y, WATERMAN A, COOK H, 等. An Agile Approach to Building RISC-V Microprocessors[J/OL]. IEEE Micro, 2016, 36(2): 8-20. DOI:10.1109/MM.2016.11.
- [15] FERDJALLAH M. Introduction to digital systems: modeling, synthesis, and simulation using VHDL[M]. John Wiley & Sons, 2011.
- [16] BACHRACH J, VO H, RICHARDS B, 等 . Chisel: Constructing Hardware in a Scala Embedded Language[C/OL]//Proceedings of the 49th Annual Design Automation Conference. Association for Computing Machinery, 2012: 1216-1225. DOI: 10.1145/2228360.2228584.
- [17] ODERSKY M, ALTHERR P, CREMET V, 等. An overview of the Scala programming language[J/OL]. 2004. <http://infoscience.epfl.ch/record/52656>.
- [18] LI P S, IZRAELEVITZ A M, BACHRACH J. Specification for the FIRRTL Language[R/OL]. (2016-02)[2024-03-01]. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html>.
- [19] ELDRIDGE S, BARUA P, CHAPYZHENKA A, 等 . MLIR as hardware compiler infrastructure[C]//Workshop on Open-Source EDA Technology (WOSET). 2021.
- [20] LATTNER C, AMINI M, BONDHUGULA U, 等. MLIR: Scaling compiler infrastructure for domain specific computation[C]//2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2021: 2-14.
- [21] IZRAELEVITZ A, KOENIG J, LI P, 等. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations[C/OL]//2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD): 卷 0. 2017: 209-216. DOI:10.1109/ICCAD.2017.8203780.
- [22] What SystemVerilog features should be avoided in synthesis?[EB/OL]. <https://stackoverflow.com/questions/20312810/what-systemverilog-features-should-be-avoided-in-synthesis>.

- [23] 余子濠, 刘志刚, 李一苇, 等. 芯片敏捷开发实践:标签化 RISC-V[J]. 计算机研究与发展, 2019, 56(1): 35-48.
- [24] 吕治宽. 基于 verilog 和 chisel 的乱序超标量 RISC-V 处理器设计比较[D]. 2022.
- [25] ARM LIMITED. AMBA AXI and ACE Protocol Specification AXI3, AXI4, AXI5, ACE and ACE5[EB/OL]. (2023). <https://www.arm.com/architecture/system-architectures/amba/amba-specifications>.
- [26] 余子濠, 王华强. DiffTest-一种高效的处理器验证方法[EB/OL]. (2021). <https://www.bilibili.com/video/BV1PU4y1V7X3>.
- [27] 荀阳霖, 翁江杰. gtkwave-filter-process-RISC-V[EB/OL]. (2024). <https://github.com/FurryAcetylCoA/gtkwave-filter-process-RISC-V>.
- [28] 蒋炎岩, 陈璐, 余子濠. abstract-machine[EB/OL]. <https://github.com/NJU-ProjectN/abstract-machine>.
- [29] EEMBC. The Embedded Microprocessor Benchmark Consortium[EB/OL]. <https://www.eembc.org/coremark/>.
- [30] 余子濠, TANG HAOJIN. ysyxsoc[EB/OL]. <https://github.com/OSCPU/ysyxSoC>.

致 谢

回顾在韶关学院的四年时光，收获甚多。不仅是在专业既能方面有了拓展和精进。更是结识了诸多良师益友。

首先要感谢我的导师王娜，她在大一期间就鼓励我及早参加比赛，让我早早地接触到了项目开发。在我的方向陷入困境时，总能给我支持以及新的思考方向。也要感谢信工之星工作室的蒋昌金指导老师，他带领了一批热爱技术的学生，排除诸多困难，一直维护工作室的良好氛围，并不断鞭策我们。

也要感谢中国科学院-先进计算机系统研究中心的包云岗，余子濠，解壁伟老师。他们带领的“一生一芯”计划、开源高性能处理器“香山”项目。为我国处理器人才培养做出了极大的贡献。本文作者是“一生一芯”计划的正式成员。在研究期间得到了该计划的宝贵指导与资助。

感谢 NJU-LUG，提供 NJUThesis Typst 论文模板。

附录

```
coa@laptop ~/ysyx-workbench/am-kernels/tests/cpu-tests <am-tests●>
$ ./Top build/fib-riscv32e-npc.bin
***** SIM *****
[log.cpp:13 init_log] Log is written to stdout
[load_img.cpp:27 load_img] Using image file build/fib-riscv32e-npc.bin, size = 524
[cpu.cpp:89 reset] Reset Released
[difftest.cpp:46 init_difftest] Difftest is disabled. ref_so_file is ignored
[main.cpp:30 welcome] Wave: OFF
[main.cpp:32 welcome] Build time: 16:18:08, Mar 15 2024
Welcome to riscv32E-npc!
For help, type "help"
(npc) mtrace begin 0x800001f0      ← 设置监视区间
(npc) mtrace end 0x800001ff
(npc) mtrace w
(npc) c ← 开始仿真          写入地址    写入内容    指令所在地址
[mtrace.cpp:42 mem_trace_w] Physical Mem write 0x800001f0: 5704e7 @0x80000060
[mtrace.cpp:42 mem_trace_w] Physical Mem write 0x800001f4: 8cccc9 @0x80000060
[mtrace.cpp:42 mem_trace_w] Physical Mem write 0x800001f8: e3d1b0 @0x80000060
[mtrace.cpp:42 mem_trace_w] Physical Mem write 0x800001fc: 1709e79 @0x80000060

--- Got stop request ---
```

图 A.1 mtrace 使用截图

```
static void single_cycle() {
    sim_state.nr_cycle++;

    #ifdef CONFIG_DUMP_WAVE, contextp->timeInc(1));
    dut.clock = 0;
    dut.eval();
    #ifdef CONFIG_DUMP_WAVE, tfp->dump(contextp->time()));

    #ifdef CONFIG_DUMP_WAVE, contextp->timeInc(1));
    dut.clock = 1;
    dut.eval();
    #ifdef CONFIG_DUMP_WAVE, tfp->dump(contextp->time()));
}

static bool single_inst() {
    // 以指令为单位执行，当本函数返回时，被执行指令对电路内寄存器的修改一定是可观测的
    int cycles = 100;
    do {
        single_cycle(); cycles--;
    } while (dut.rootp->Top_DOT_fetch_DOT_pc_DOT_pc_valid != 1 &&
             cycles != 0);
    if (cycles == 0) {
        printf(format: "指令超时 CPU是不是卡住了?\n");
        return false;
    }
    return true;
}
```

图 A.2 单指令执行相关代码

```
profile: prof
prof:
    @echo "Profile Mode"
    $(MAKE) clean_c all PROFLAGS="--prof-cfuncs -CFLAGS -gline-tables-only" > /dev/null
    @if [ -z "$${ARGS}" ]; then \
        echo 'No $$ARGS provided. Using microbench(train)'; \
        $(MAKE) -C ${AM_K}/benchmarks/microbench ARCH=${ARCH} mainargs=train clean image > /dev/null && \
        ARGS=${AM_K}/benchmarks/microbench/build/microbench-${ARCH}.bin && \
        cd ${BUILD_DIR} && ${BIN} -b $$ARGS; \
    else \
        cd ${BUILD_DIR} && ${BIN} -b ${ARGS}; \
    fi

    cd ${BUILD_DIR} && gprof ${BIN} gmon.out > gmon.txt && \
    verilator_profcfunc gmon.txt > vpof.txt && \
    vim +0 vpof.txt && mv ${BIN} ${BIN}.prof
    @rm -r ${BUILD_DIR}/obj_dir/* # Otherwise make sim followed by make prof will seq fault
```

图 A.3 自动性能分析相关代码