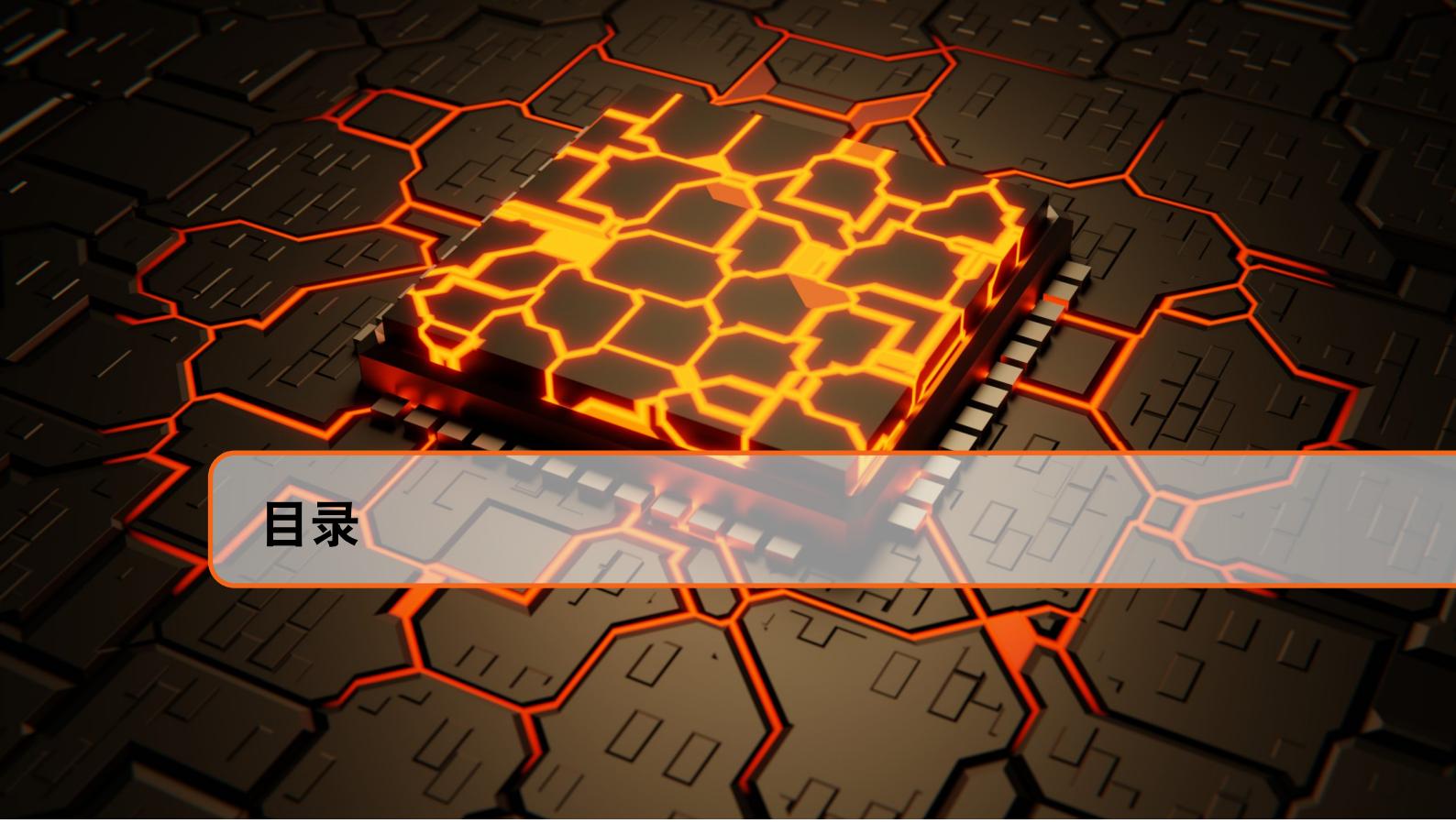


# ZenCove 项目决赛设计报告

第六届“龙芯杯”全国大学生计算机系统能力培养大赛

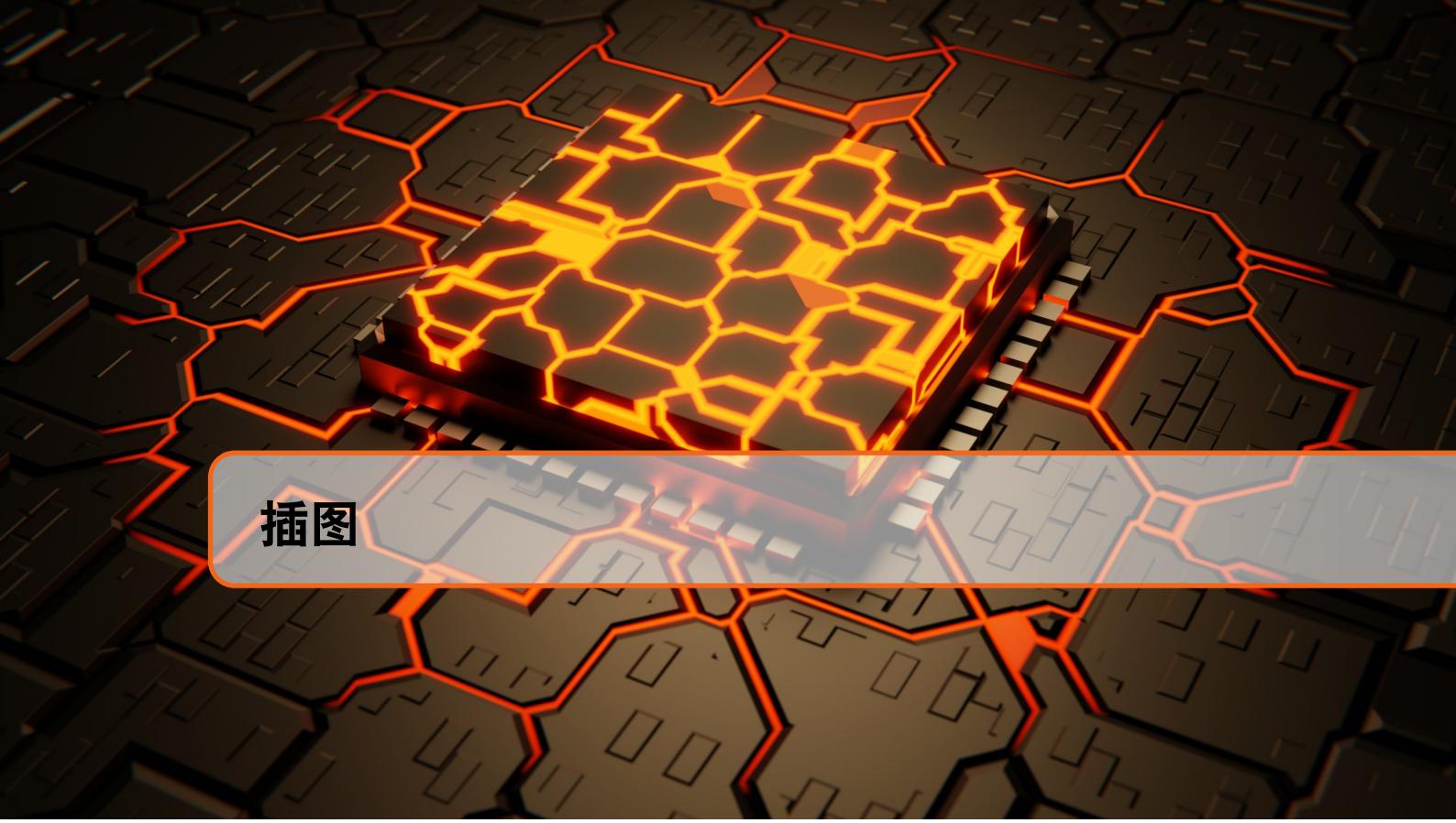
崔轶锴 张为 王拓为



# 目录

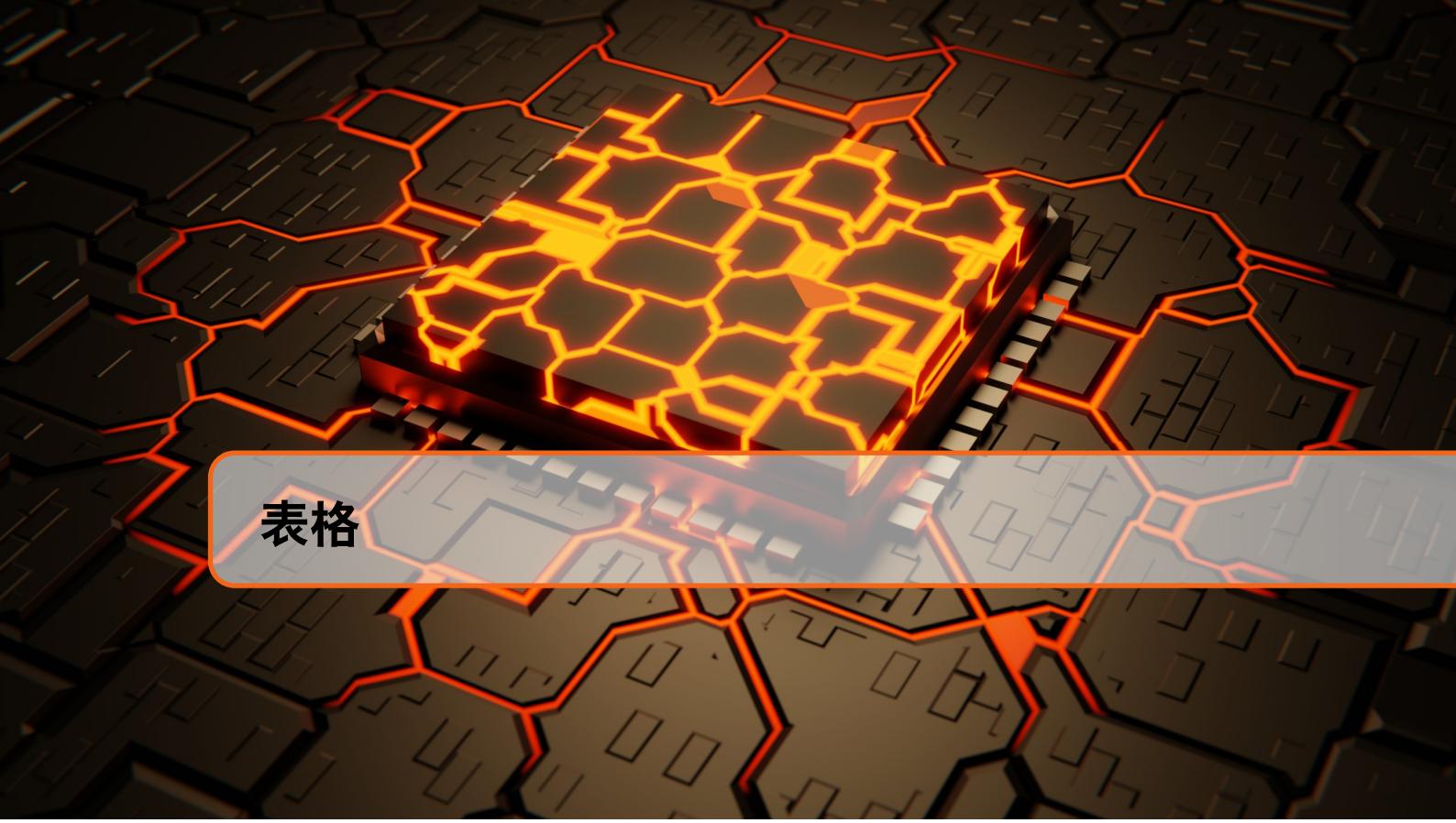
<b>1</b>	<b>项目概述</b>	<b>6</b>
<b>1.1</b>	<b>项目背景</b>	<b>6</b>
<b>1.2</b>	<b>项目概述</b>	<b>6</b>
1.2.1	硬件语言	6
1.2.2	设计模式	6
1.2.3	CPU 架构	7
1.2.4	SoC 设计	7
1.2.5	系统软件	7
<b>1.3</b>	<b>开发平台</b>	<b>7</b>
1.3.1	硬件平台	7
1.3.2	软件平台	8
<b>1.4</b>	<b>参考资料</b>	<b>8</b>
<b>2</b>	<b>CPU 架构</b>	<b>9</b>
<b>2.1</b>	<b>基本架构</b>	<b>9</b>
2.1.1	前端	10
2.1.2	后端	11
2.1.3	指令提交	11
<b>2.2</b>	<b>指令支持</b>	<b>12</b>
<b>2.3</b>	<b>协处理器 0</b>	<b>12</b>
<b>2.4</b>	<b>内存管理</b>	<b>13</b>

2.5	<b>异常中断</b>	13
2.6	<b>缓存设计</b>	13
2.7	<b>分支预测</b>	14
2.7.1	Gselect 分支方向预测器	14
2.7.2	调用返回预测器	14
2.8	<b>微架构优化</b>	15
2.8.1	推测唤醒	15
2.8.2	流水线前传	15
2.9	<b>外部接口</b>	15
<b>3</b>	<b>SoC 设计</b>	<b>16</b>
3.1	<b>总体结构</b>	16
3.2	<b>地址映射</b>	17
3.3	<b>中断连接</b>	18
3.4	<b>驱动说明</b>	19
3.4.1	串口控制器	19
3.4.2	以太网口控制器	19
3.4.3	Flash	19
3.4.4	PS2 接口	19
3.4.5	GPIO 控制	19
3.4.6	LCD 屏	19
3.4.7	TFT-VGA 控制器	20
<b>4</b>	<b>系统软件</b>	<b>21</b>
4.1	<b>监控程序</b>	21
4.2	<b>引导程序</b>	21
4.2.1	一级引导程序 Bootloader	21
4.2.2	二级引导程序 U-Boot	21
4.3	<b>操作系统程序</b>	22
4.3.1	uCore-thumips	22
4.3.2	Linux	22
	<b>Appendices</b>	<b>23</b>
<b>A</b>	<b>声明与致谢</b>	<b>23</b>
A.1	<b>版权声明</b>	23
A.2	<b>致谢</b>	23



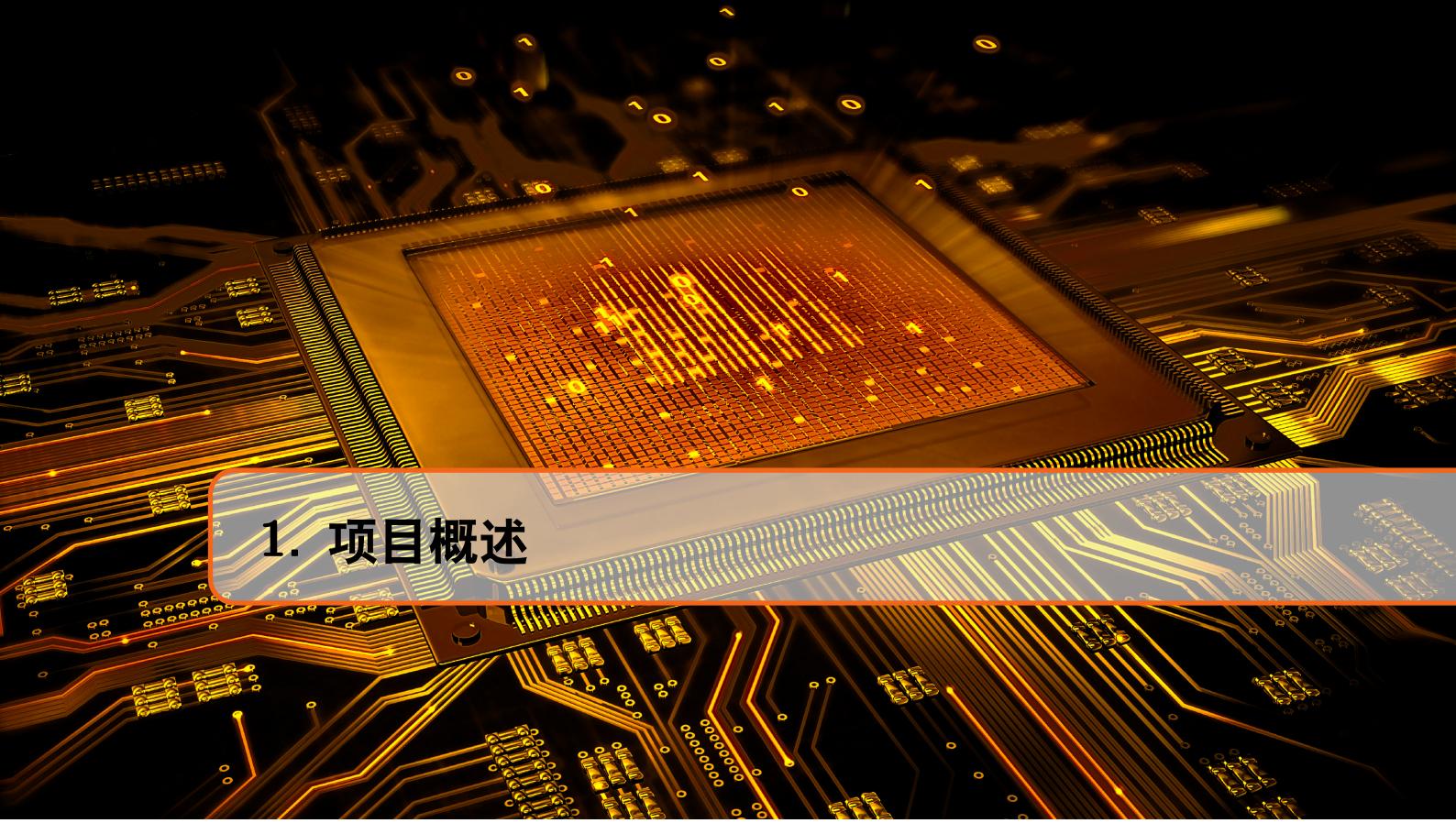
插图

2.1 ZenCove 整体架构 .....	10
3.1 ZenCove SoC 整体结构 .....	17



## 表格

3.1 ZenCove SoC 地址映射表	18
3.2 ZenCove 中断配置	18



## 1. 项目概述

### 1.1 项目背景

本项目是第六届“龙芯杯”全国大学生计算机系统能力培养大赛（NSCSCC 2022）的参赛作品。

项目实现了基于 MIPS32 指令集的乱序多发射 CPU——ZenCove，并以比赛提供的 FPGA 实验平台为基础，利用周边硬件，实现了完整的 CPU 微架构。经测试，可以通过全部的功能测试、性能测试，并运行监控程序、TrivialBootloader 引导程序、U-Boot 引导程序、uCore 操作系统和 Linux 操作系统等。

### 1.2 项目概述

#### 1.2.1 硬件语言

本项目使用基于 Scala 构建的硬件描述语言 SpinalHDL 编写。相较于传统硬件描述语言，SpinalHDL 具有更强大的表达能力、抽象能力和面向对象能力，可以降低硬件设计和实现的难度，提升开发效率。

#### 1.2.2 设计模式

本项目借鉴了开源 RISC-V CPU 项目 Vexriscv 的设计模式，将包括 ALU、寄存器文件、PC 组件等在内的几乎全部功能模块作为插件（Plugins）在流水线中插入或移除。得益于此，各部分功能代码不再分散在各个文件中，而是位于单独的插件文件中，实现了多个代码文件的解耦。这既使得 CPU 具有了高度的可配置性，也为协同开发提供了便利。

### 1.2.3 CPU 架构

本项目采用了乱序多发射的基本架构，在标准配置下为 4 发射。其中，流水线结构可以大致分为前端取指译码和后端执行提交两部分。具体地，前端部分包括取指令 1、取指令 2、指令译码、寄存器重命名和指令派发 5 个阶段，后端部分包括发射、读寄存器、执行和写回 4 个阶段。

本项目支持基于 TLB 的虚拟地址转换，并实现了指令和数据缓存以加快取指和访存。在指令支持方面，基于 MIPS32 Release 1 指令集，我们实现了单核处理器运行 Linux 所需的 92 条指令，支持规范中必须实现的 23 个 CP0 寄存器和 12 种异常类型，实现了精确异常。

### 1.2.4 SoC 设计

为了验证 CPU 设计和实现的正确性，同时更充分地利用实验硬件平台上提供的外设资源进行功能演示，我们围绕 ZenCove CPU 搭建了一个 SoC(System-on-Chip)，包含以下模块：

- CPU: ZenCove CPU (包含 Cache)
- DRAM: 板载 128MiB DDR3 SDRAM 作为主存
- 片内 RAM: 板载 128KB BootROM 和 64KB RAM 用于存放和运行一级引导程序
- Flash: 用于存放二级引导程序
- 串口: 16550 兼容的串口控制器
- 以太网: 使用 Xilinx IP 构建的以太网控制器，实线 10Mbps 通信
- GPIO: 拨码开关，按钮键盘，数码管，led 等
- VGA: 支持在 VGA 上进行图像输出
- PS2: 支持 PS2 键盘输入
- LCD: 支持在 LCD 屏上进行图像输出

### 1.2.5 系统软件

为了进一步验证 CPU 设计和实现的正确性，我们尝试并成功运行了如下系统软件：

- 清华大学 MIPS 32 位版本监控程序
- NonTrivialMIPS 项目的 TrivialBootloader
- U-Boot 引导程序 (v2022.04)
- 清华大学 uCore-mips 操作系统
- Linux 操作系统 (v5.19-rc4)

## 1.3 开发平台

### 1.3.1 硬件平台

本项目使用的硬件平台是比赛提供的龙芯体系结构教学实验箱 (Artix-7)，其核心是一块基于 FPGA 芯片的嵌入式系统开发板，型号为 XC7A200T-FBG676。此外，平台包含了 DDR3、SRAM、NAND、Flash 等丰富的外设资源。

### 1.3.2 软件平台

本项目的开发共分为两部分。

第一部分是基于 mill 0.10 版本构建的 Scala 项目，在 VS Code/JetBrain IntelliJ IDEA 中进行开发，用于生成 Verilog 代码

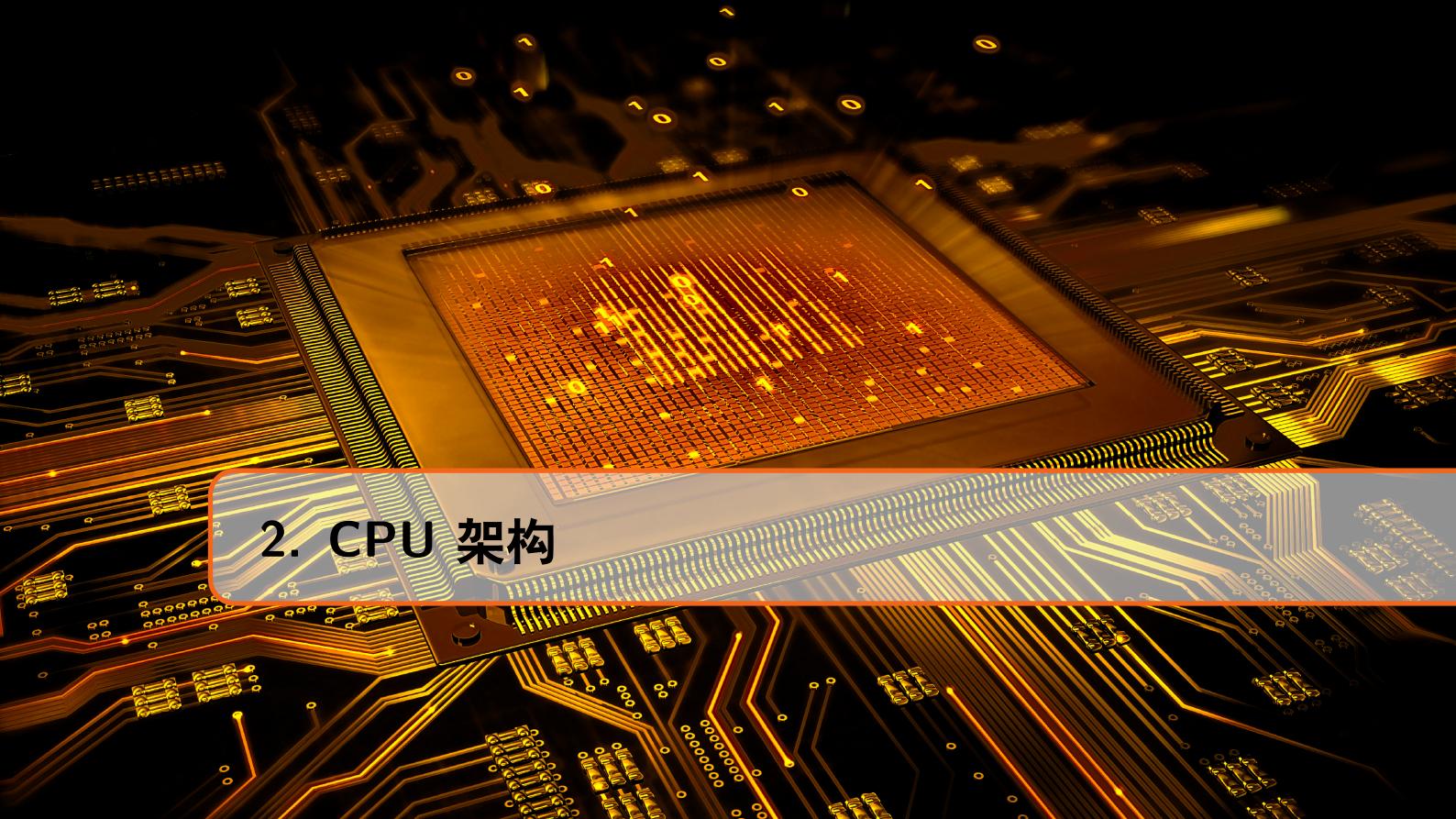
第二部分使用 Xilinx Vivado 2019.2 Web HL Edition 作为开发 IDE 平台。

两部分共同使用 GitLab-CI 进行自动化集成。

## 1.4 参考资料

本项目的设计和开发过程中参考和借鉴了包括但不限于以下资料：

- 超标量处理器设计. 姚永斌
- The Berkeley Out-of-Order Machine (BOOM)
  - 文档: <https://docs.boom-core.org/en/latest/sections/intro-overview/boom.html>
  - 仓库: <https://github.com/riscv-boom/riscv-boom>
- SpinalHDL 文档: <https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html>
- Vexriscv 仓库: <https://github.com/SpinalHDL/VexRiscv>
- NonTrivial-MIPS 仓库: <https://github.com/trivialmips/nontrivial-mips>
- LLCL-MIPS 仓库: <https://github.com/huang-j1/LLCL-MIPS>
- MIPS® Architecture For Programmers I, II, III. Imagination Technologies LTD.
- 各外设使用手册. 相关厂商



## 2. CPU 架构

### 2.1 基本架构

ZenCove 采用乱序多发射的基本架构，发射数量为可配置的参数。在标准配置下为 4 发射。乱序多发射流水线大致可以分为前端流水线与后端流水线，以重排序缓存 (ROB) 和发射队列为分隔。为了满足精确异常的要求，在前端与后端执行完指令后，由 ROB 异步地将指令提交。

CPU 的整体架构如下图所示：

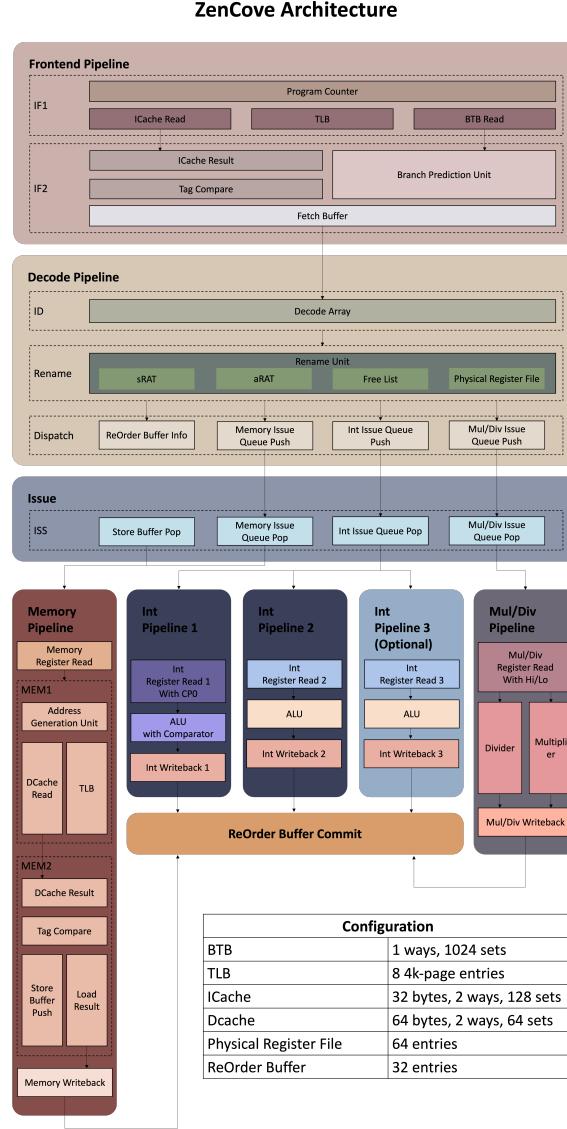


图 2.1: ZenCove 整体架构

### 2.1.1 前端

前端流水线共 5 级，包括两级取指令、指令译码、寄存器重命名、指令派发。

**取指令 1** 这一阶段进行 PC 的 TLB 查询与地址翻译，并进行 I-Cache 查询的第一阶段。分支预测在此阶段查询分支目标缓存 (BTB) 与预测历史表 (PHT)。

**取指令 2** 这一阶段进行 I-Cache 查询的第二阶段，并对 Cache 缺失的情况进行填充。分支预测在此阶段形成预测结果，预测目标 PC。这一阶段之后将指令装入取指令缓冲 (Fetch buffer)，这样可以平衡取指令速度与指令派发速度，也将取指令速度“削峰填谷”，减少气泡与暂停的出现。

**指令译码** 指令译码阶段取出取指令缓冲中的若干条指令（宽度为可配置参数），进行译码，翻译成微码 (Micro op,  $\mu$ op)。由于 MIPS 为 RISC 架构，我们保证每条指令只生成一个  $\mu$ op。

**寄存器重命名** ZenCove 采用显式重命名，即只采用物理寄存器重命名逻辑寄存器，将寄存器映射存储在重命名映射表 (Rename alias table, RAT) 中。由于精确异常回滚的需要，重命名映射表需要存储推测性的和架构性的两份。采用 FIFO 管理的空闲寄存器列表 (Free list) 管理可以被分配的物理寄存器，在此阶段给需要写寄存器的指令分配物理寄存器。若需要分配而 Free list 已空，则需要暂停此阶段。

**指令派发** 此阶段根据指令类型将指令装入不同的发射队列。ROB 的装入可以在此阶段，也可以在寄存器重命名阶段。

### 2.1.2 后端

后端流水线数量等于发射宽度。在默认配置下，有三条不完全对称的整数流水线，一条乘除法流水线，一条访存流水线，组成 5 发射。所有后端流水线都由发射、读寄存器、执行、写回 4 个主要阶段组成，其中访存流水线的执行由于 D-Cache 访问的需要扩展成了两个阶段。

**发射** ZenCove 采用分布式的发射队列，有整数发射队列、乘除法发射队列、访存发射队列 3 个发射队列。整数发射队列有至多 3 条整数流水线（第 3 条为可选）共同取指令发射，乘除法与访存各自面向一条流水线。整数发射采用纯乱序的方式，三条流水线共同选择避免重复发射；乘除法与访存目前均采用纯顺序的方式，因此它们都是 FIFO 队列，但是不同发射队列的指令之间仍然构成了乱序。整数流水线在此阶段唤醒整数发射队列中的指令，使得在前传下整数指令写后读 (RAW) 相关可以背靠背执行。

**读寄存器** 此阶段读物理寄存器堆，并进行寄存器的前传。整数流水线在此阶段唤醒与它 RAW 相关的后续指令，使得所有指令均可以间隔一个气泡读到整数指令的结果。

**执行** 每一条整数流水线均包括相同的算术逻辑单元 (ALU)，而仅有一条整数流水线包括比较器，读协处理器 0 (CP0) 的功能。乘除法流水线在此阶段计算乘法、除法，以及处理 Hi/Lo 寄存器的读写。访存流水线的执行第一阶段进行 TLB 查询与地址翻译，并进行 D-Cache 查询的第一阶段。第二阶段进行 D-Cache 查询的第二阶段，并对 Cache 读缺失的情况进行填充。同时对于 Store 指令和 Uncached 地址段的 Load 指令，由于执行会对处理器状态造成不可恢复的改变，装入 **Store buffer** 等待提交后执行。从 Store buffer 发射的指令，在此阶段进行 D-Cache 的写，进行写缺失填充，或对 Uncached 情况进行直接设备访问。

**写回** 此阶段将执行结果分别写入物理寄存器和 ROB。

### 2.1.3 指令提交

ROB 读取未提交的指令，判断指令是否可以提交，按顺序提交至多提交宽度条指令，并对分支预测错误、异常与中断、处理器状态回滚、Store buffer 激活等需要提交时处理的情况进行处理。

为了降低硬件复杂度同时简化特殊情况的处理，我们限制了只有 0 号提交口可以处理特殊的指令（如 Conditional Move, Uncached Load 等）和特殊的情况（如异常，分支预测错误等需要重置处理器状态的情况）。

## 2.2 指令支持

ZenCove 共实现了 92 条 MIPS 指令，基本涵盖了单核 MIPS32 Release 1 处理器除浮点外所需的指令，按照功能划分如下：

**算术指令** ADD, ADDI, ADDIU, ADDU, CLO, CLZ, DIV, DIVU, MADD, MADDU, MSUB, MSUBU, MUL, MULT, MULTU, SUB, SUBU

**逻辑指令** AND, ANDI, NOR, OR, ORI, SLL, SLLV, SRA, SRAV, SRL, SRLV, XOR, XORI

**访存指令** LB, LBU, LH, LHU, LW, LWL, LWR, SB, SH, SW, SWL, SWR

**分支指令** BEQ, BGEZ, BGEZAL, BGTZ, BLEZ, BLTZ, BLTZAL, BNE, J, JAL, JALR, JALR.HB, JR, JR.HB

**自陷指令** TEQ, TEQI, TGE, TGEI, TGEIU, TGEU, TLT, TLTI, TLTIU, TLTU, TNE, TNEI

**移动指令** LUI, MFHI, MFLO, MOVN, MOVZ, MTHI, MTLO, SLT, SLTI, SLTU, SLTIU

**特权指令** BREAK, CACHE, ERET, MFC0, MTC0, SYSCALL, TLBP, TLBR, TLBWI, TLBWR, WAIT

**其他指令** PERF, SYNC

由于 AXI 总线不支持以 3 bytes 为访问粒度进行读写，故 Unaligned Load/Store 指令 (LWL/LWR/SWL/SWR) 不支持以 Uncached 方式进行读写。

## 2.3 协处理器 0

ZenCove 共实现了 23 个 CP0 寄存器以提供对于操作系统所必须的功能支持，按照地址排序如下：

名称	地址	名称	地址
Index	0, 0	IntCtl	12, 1
Random	1, 0	SRSCtl	12, 2
EntryLo0	2, 0	Cause	13, 0
EntryLo1	3, 0	EPC	14, 0
Context	4, 0	PRId	15, 0
PageMask	5, 0	EBase	15, 1
Wired	6, 0	Config	16, 0
BadVaddr	8, 0	Config1	16, 1
Count	9, 0	Config2	16, 2
EntryHi	10, 0	Config3	16, 3
Compare	11, 0	ErrorEPC	30, 0
Status	12, 0		

## 2.4 内存管理

ZenCove 支持基于 TLB 的虚拟地址转换，TLB 项数可配置。依据 MIPS 标准，各对应段的映射关系和访存行为如下：

名称	地址段	User 态可访问？	固定映射？	可 Cache？
kuseg (ERL=1)	[0x0000_0000, 0x7FFF_FFFF]	N/A	是	否
useg/kuseg	[0x0000_0000, 0x7FFF_FFFF]	是	否	根据 TLB
kseg0	[0x8000_0000, 0x9FFF_FFFF]	否	是	K0=1 则是
kseg1	[0xA000_0000, 0xBFFF_FFFF]	否	是	否
sseg	[0xC000_0000, 0xDFFF_FFFF]	否	否	根据 TLB
kseg3	[0xE000_0000, 0xFFFF_FFFF]	否	否	根据 TLB

注意，kseg0 段与 kseg1 段映射到相同的物理地址段。在 SoC 上，这一段部分为 MMIO 寄存器，因此使得一些 kseg0 地址的访问错误地以 Cached 方式访问了 MMIO。虽然程序正常执行不会出现这样的访问，但是乱序处理器的前瞻执行会导致访问随机的虚拟地址。因此我们在实际实现时根据 SoC 的实际地址分配对 kseg0 一部分进行了屏蔽，避免了 MMIO 以 Cached 方式被映射到。

## 2.5 异常中断

ZenCove 共实现了 12 个异常编号、共 17 种异常的处理，按照编号排序如下：

名称	编号
Interrupt	0
TLB modification	1
TLB invalid/refill (Load or Instruction Fetch)	2
TLB invalid/refill (Store)	3
Address error (Load or Instruction Fetch)	4
Address error (Store)	5
Syscall	8
Breakpoint	9
Reserved instruction	10
Coprocessor Unusable	11
Arithmetic Overflow	12
Trap	13

## 2.6 缓存设计

ZenCove 采用可配置组相联缓存，默认配置下 I-Cache 大小 8KB，2 路，Cache line 为 32B。D-Cache 大小 8KB，2 路，Cache line 为 64B。均为两级流水线访问，I-Cache 单

周期默认配置取 16B，满足取出同一 Cache line 至多 4 条指令的要求。Cache 采用 VIPT，无重名。均采用 LRU 作为替换策略，D-Cache 为写回 Cache。

## 2.7 分支预测

分支预测器一共 3 部分，分支目标缓冲 (BTB)，分支方向预测器和基于返回地址栈的调用返回预测器。

在实际实验中我们尝试了两位饱和计数器局部预测器，Gshare、Gselect 全局预测器，以及局部预测和全局预测的混合预测器。由于插件设计模式的便利性，只需要简单的修改就可以切换到不同的分支方向预测器以及与其匹配的分支目标缓冲。最终我们采用了 Gselect 全局预测器，可以较好地平衡分支指令间的冲突和全局历史的长度。

ZenCove 的 BTB 大小和具体参数可配置，默认配置下可存储 1024 项，直接映射，每周期可以任意取连续 4 项，对应连续 4 个 PC 的分支信息。

默认配置下每项存储以下信息：

- tag: 用于判断信息是否与地址匹配
- target: 目标地址
- isCall: 是否为函数调用指令
- isRet: 是否为函数返回指令

### 2.7.1 Gselect 分支方向预测器

默认参数下，Gselect PHT 共有 8192 个表项，每个表项对应 2 位饱和计数器。记录 5 位长度的全局历史，与 PC 的 9 至 2 位拼接查询表项，形成预测结果。

分支错误信息在 ROB 对应指令提交进行更新，在初始化时需要区分条件分支 (branch) 和无条件分支 (jump) 指令。

### 2.7.2 调用返回预测器

ZenCove 的返回地址栈 (RAS) 大小可配置，默认为 8 项。

对于 jr/jalr/jal/bal 等指令，若其写回寄存器为 ra 寄存器，则我们认为其为函数调用指令。若其跳转目标为 ra 寄存器的内容，则我们认为其为调用返回指令。

在前端进行分支预测时，若其遇到对应的函数调用/返回指令，则将预测跳转地址压栈/弹出作为预测跳转地址。为了让后端在提交分支预测错误信息时可以恢复 RAS 的指针信息，我们将该信息同指令的  $\mu$ op 一起传到 ROB 中，在后端提交时进行恢复。

若函数调用层数超过 RAS 项数，则循环覆盖，该操作只会造成特别底层的函数调用返回时产生分支地址的预测错误。

若函数返回弹出 RAS 项时，弹出了被覆盖的项，也只会导致分支地址错误，不会产生更多的副作用。

## 2.8 微架构优化

ZenCove 微架构在设计时，为了进一步提高指令执行效率，加入了推测唤醒和流水线前传两个执行方法。

这两个方法可以有效减少指令在各个执行流水线中冲突时暂停的周期，从而有效降低 CPI。

### 2.8.1 推测唤醒

Load 指令与需要读 load 结果的后续指令的 RAW 依赖是程序执行中常见的依赖，即 load-use 依赖。

由于 Load 指令执行周期数不确定，尤其是在 Cache 缺失时，因此只能在访存流水线的执行 2 (MEM2) 阶段进行唤醒。这导致 load-use 要空 3 个气泡。即使是乱序执行也通常找不到这么多不相关指令，导致 Load 经常导致气泡出现。

而 Cache 命中时 Load 执行周期时确定的，因此在 MEM1 阶段即可唤醒，那么 Cache 命中时 load-use 只需要空 2 个气泡，通常就可以找到其它不相关指令进行填充。

但在 Cache 缺失时这一唤醒就是错误的，后续例如整数指令就不能及时读到正确的寄存器结果。

因此我们在 Cache 缺失时简单地将所有流水线暂停。考虑到 Cache 缺失应当是小概率事件，并且 Cache 缺失本就不可避免地导致处理器执行暂停，因此总体而言还是能够做到性能提升。

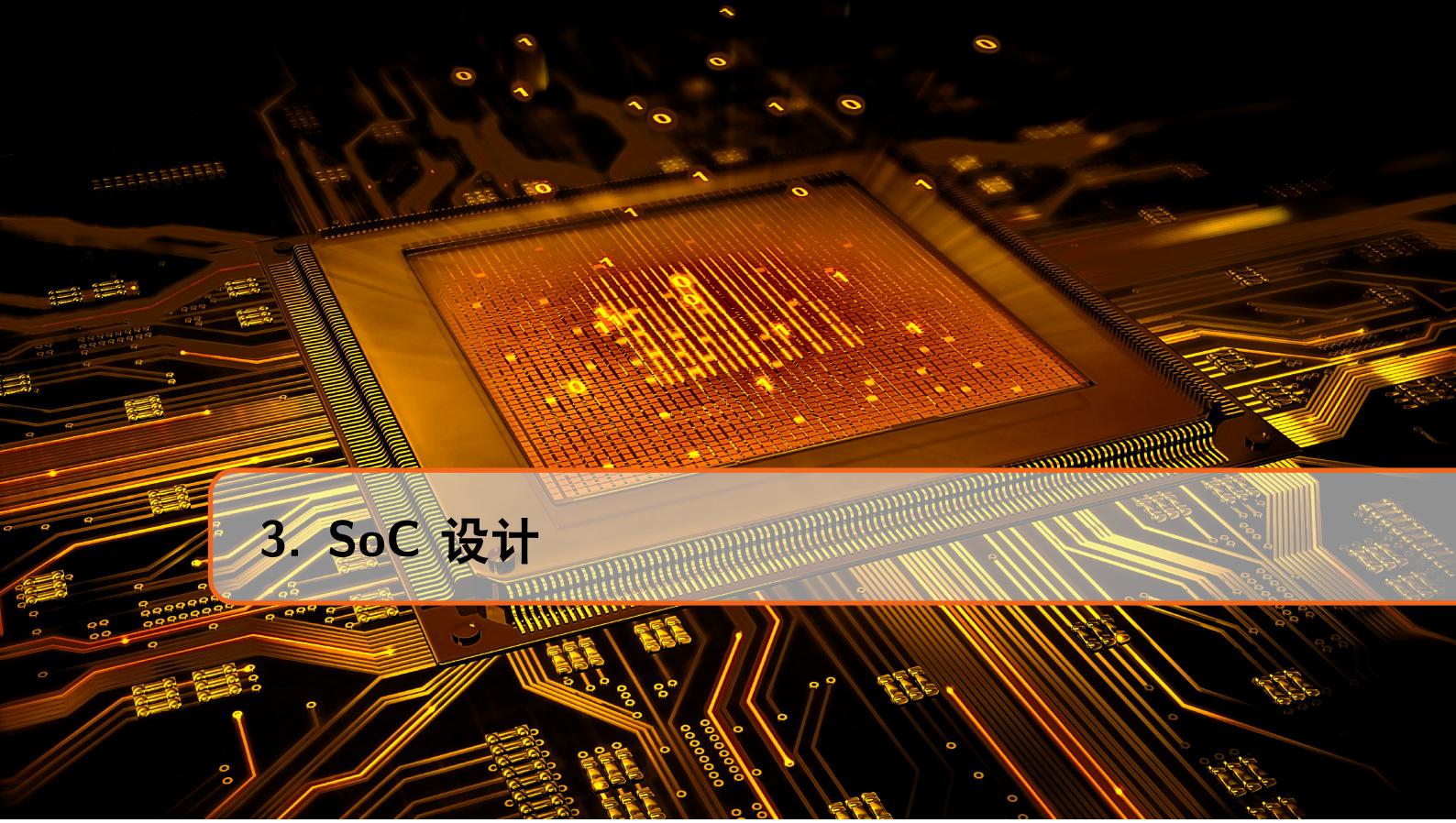
### 2.8.2 流水线前传

由于发射宽度较宽，本处理器没有实现所有流水线之间的前传，但对最常用的整数流水线之间实现了完全的前传。在没有前传的情况下，整数指令与其它指令之间的 RAW 依赖需要空一个气泡，整数流水线的前传使得整数指令之间的 RAW 依赖可以背靠背执行，达到了最高的执行效率。

## 2.9 外部接口

CPU 采用 AXI4 总线与外部相连，流水线直接访问 Instruction bus, Cached data bus, Uncached data bus 三条总线，对于需要暴露一条总线的，采用 AXI crossbar IP 做 3\*1 crossbar。

按照 MIPS 手册，CPU 支持 6 个外部中断。



### 3. SoC 设计

#### 3.1 总体结构

目前，ZenCove 支持的外设如下：

- CPU: ZenCove CPU (包含 Cache)
- DRAM: 板载 128MiB DDR3 SDRAM 作为主存
- 片内 RAM: 板载 128KB BootROM 和 64KB RAM 用于存放和运行一级引导程序
- Flash: 用于存放二级引导程序
- 串口: 16550 兼容的串口控制器
- 以太网: 使用 Xilinx IP 构建的以太网控制器, 实线 10Mbps 通信
- GPIO: 拨码开关, 按钮键盘, 数码管, led 等
- VGA: 支持在 VGA 上进行图像输出
- PS2: 支持 PS2 键盘输入
- LCD: 支持在 LCD 屏上进行图像输出

ZenCove SoC 的整体结构如下图所示：

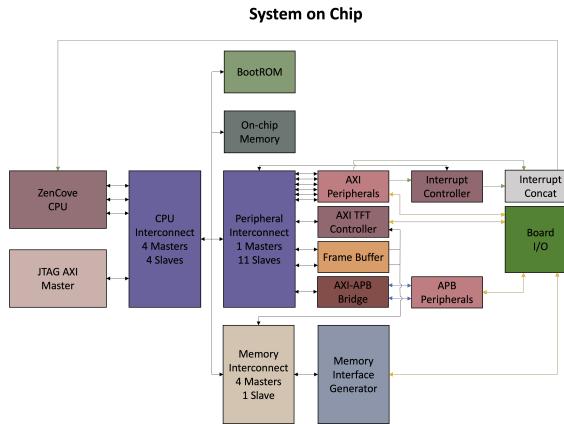


图 3.1: ZenCove SoC 整体结构

SoC 由 Vivado 的 Block Design 搭建。图中深绿色方块为开发板的 IO 接口，紫色方块为 crossbar，其余颜色方块为设备。图中黑色线是 AXI 协议的连线，蓝色线为 APB 协议连接线，黄色线为接口信号线，绿色线为中断信号线。

出于消耗资源量考虑，我们没有将 CPU 的 3 个 AXI 接口（指令总线、数据总线、Uncached 数据总线）和全部设备直接接到同一个 crossbar 上，而是选出两个较常访问，同时平均访问延迟低的设备：Boot ROM 和片上内存，与 CPU 的 AXI 接口通过 crossbar 直接相连。剩余设备都接到了设备用 crossbar 上，与主 crossbar 相连。

片上内存的目的是给我们的 bootloader 提供单独的运行空间，从而在执行系统程序的时候，可以保证 bootloader 不会被覆盖，在运行出错的时候输出相关信息，便于调试。

由于 PS2 和 LCD 屏的驱动使用的是 APB 协议进行交互，因此我们设置了 AXI-APB 转换桥，用于连接两个设备的驱动模块。

同时，由于 TFT (VGA) 需要的内存带宽较高，我们为 TFT 和 Framebuffer 配置了 DMA，让 VGA 在内存总线空闲时可以独立访问 DRAM，从而降低显示画面的延迟。

我们还另外添加了 JTAG AXI master，这样可以直接通过 JTAG 访问所有外设，为内存和 IO 相关的调试提供了方便。

## 3.2 地址映射

ZenCove 中各个设备的物理地址分配如表3.1，对于 3 个总线，地址映射相同：

设备	起始地址	结束地址	有效大小	类型
DDR3	0x00000000	0x07FFFFFF	128 MB	存储设备
OCM	0x08000000	0x0800FFFF	64 KB	存储设备
CFG Flash	0x1A000000	0x1AFFFFFF	16 MB	存储设备
Ethernet	0x1C000000	0x1C00FFFF	64 KB	配置寄存器
TFT	0x1C010000	0x1C01FFFF	64 KB	寄存器
PS2	0x1C020000	0x1C020FFF	4 KB	配置寄存器
NT35510	0x1C030000	0x1C030FFF	4 KB	寄存器
SPI Flash	0x1C040000	0x1C040FFF	4 KB	存储设备
Framebuffer Read	0x1C050000	0x1C05FFFF	64 KB	寄存器
Framebuffer Write	0x1C060000	0x1C06FFFF	64 KB	寄存器
Interrupt Controller	0x1D000000	0x1D00FFFF	64 KB	配置寄存器
BootROM	0x1FC00000	0x1FC1FFFF	128 KB	存储设备
Uart	0x1FD02000	0x1FD03FFF	8 KB	配置寄存器
GPIO	0x1FF00000	0x1FF0FFFF	64 KB	配置寄存器

表 3.1: ZenCove SoC 地址映射表

这里由于 AXI 总线最小的地址映射单位为 4KB，因此部分配置寄存器的地址映射大小配置成了 4KB。

同时 Xilinx Uart 16550 需要的地址映射为 10xx，因此给串口控制器配置了 8KB 的空间，而不是最小的 4KB。

### 3.3 中断连接

ZenCove 的中断连接关系如表3.2所示：

设备名称	中断类型	接受方	中断编号
串口	高电平	CPU	2
PS/2	高电平	CPU	3
中断控制器	高电平	CPU	6
以太网	上升沿	中断控制器	0
SPI Flash	上升沿	中断控制器	1
CFG Flash	上升沿	中断控制器	2

表 3.2: ZenCove 中断配置

## 3.4 驱动说明

### 3.4.1 串口控制器

开发板提供的是 RS-232 接口。我们使用 Xilinx UART 16550 进行驱动。其相比 UART lite 支持更多的功能，如可变波特率、读写缓冲等。在监控程序，uCore 和 Linux 中都提供了相关驱动，可以直接使用。

Xilinx UART 16550 控制器有一个中断信号，表示收到数据，由于为电平触发，因此可以直接接到 CPU 的外设中断信号上。后续使用上边沿触发的中断信号需要进行寄存才能够接到 CPU 上进行处理，因此会通过中断控制器，后续中断内容不再详细说明。

### 3.4.2 以太网口控制器

开发板上提供了以太网标准协议中的 MDIO 和 MII 接口。我们使用了 Xilinx Ethernet Lite 进行驱动。Linux 和 U-Boot 中提供了设备驱动，可以直接使用。

该接口在测试时（U-Boot 进行系统镜像拷贝）能够跑到 10Mbps 左右。

### 3.4.3 Flash

开发板上共有两块 Flash，其中一块用于固化设计，保存比特流用。该 Flash 的前 8MB 空间为设计固化预留，因此我们使用了后 8MB 空间，用于存储将要被引导启动的系统，如 uCore, U-Boot 等。这样可以减少上板测试的时间和操作。

另一块 Flash 由于空间较小，我们只将其作为普通设备进行使用。

两个控制器均支持 I/O 中断控制，因此我们将两者的中断信号接到了中断控制器上进行处理。

### 3.4.4 PS2 接口

开发板上提供了一个 PS2 物理接口，可以连接键盘等设备。我们使用了 Altera 提供的 PS2 控制器，将其接入到我们的 SoC 当中。

该控制器使用 APB 协议进行交互，因此我们向 SoC 中接入 AXI-APB 转换桥，从而让其可以与 CPU 正常通信。

该控制器会有输入产生的中断信号，我们将其直接接到了 CPU 上。

### 3.4.5 GPIO 控制

开发板上有许多额外的输入输出设备，包括拨码开关，按钮键盘，数码管，LED 等。我们将龙芯提供的 confreg 模块与 AXI 接口一并打包封装成 IP 核，直接接入到 SoC 中。在封装 confreg 的时候为了适配设备，我们进行了一定程度的修改。

### 3.4.6 LCD 屏

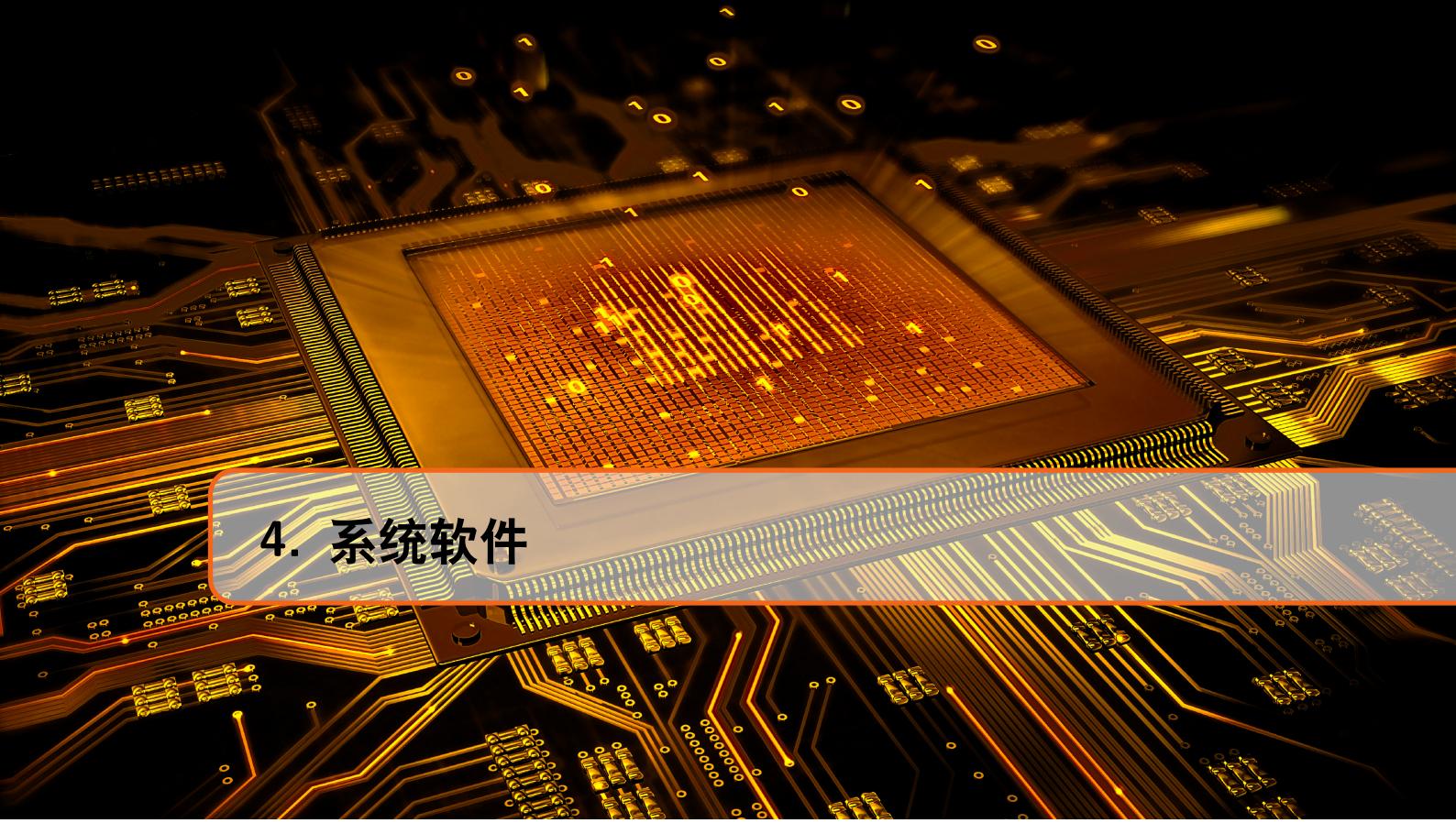
开发板上提供了 NT35510 接口驱动的 480x800 的 LCD 显示屏。我们参考了 NonTrivial-MIPS 提供的 LCD 控制器以及 Linux 上的对应驱动。

在修改了一些错误之后，我们成功驱动起了该 LCD 屏幕，具体运行方法如下：

1. 找到一张 480x800 的图片，通过我们提供的 lcd\_gen.py 将其转换为给 lcd 使用的 bin 文件。
2. 将该 bin 文件放到 linux 的文件系统上（如通过 nfs）。
3. 执行如下命令: `cat <bin文件> > /dev/nt35510`。等待写入完成之后就可以在显示屏上看到图片了。

### 3.4.7 TFT-VGA 控制器

开发板上提供了标准的 VGA 接口用于图像输出。我们参考了 NonTrivial-MIPS 的 SoC 进行了 Framebuffer 和 TFT 结构的搭建，并配置了 DMA 从内存中读取图像数据。



## 4. 系统软件

### 4.1 监控程序

监控程序是本次比赛要求运行的系统测试，能够接收用户命令，支持输入汇编指令并运行，查看寄存器及内存状态等功能。由于其覆盖指令较少且指令逻辑较为简单，仅可作为 CPU 功能正确性的初步验证。

### 4.2 引导程序

#### 4.2.1 一级引导程序 Bootloader

嵌入式系统的板载存储通常有限，为了运行诸如 Linux 的所占空间较大的操作系统，需要借助引导程序从 Flash、U 盘、网络等来源加载其镜像文件到内存并进行引导。U-Boot 是目前最为常见的引导程序。

受硬件平台的限制，ZenCove 实现的 SoC 无法直接使用 U-Boot 进行加载，需要使用更低一级的、所占空间更小的引导程序完成系统初始化和加载工作。

基于以往项目，我们选用了 TrivialBootloader 作为一级引导程序。TrivialBootloader 可以直接存放于固化在 FPGA 中的 BootROM 中，在系统每次上电或重置时被首先执行，完成内存的检查、初始化和后续程序的加载工作。

#### 4.2.2 二级引导程序 U-Boot

作为二级引导程序，U-Boot 初始存放在 Flash 中等待被 TrivialBootloader 加载。加载后的 U-Boot 可以从网口通过 tftp 协议加载 Linux 操作系统至指定内存区域，最终完成全部的引导工作。

此外，由于 U-Boot 本身包含了较多的外设驱动并且具备基本的交互功能，也可以作

为硬件测试和演示的工具。

## 4.3 操作系统程序

### 4.3.1 uCore-thumips

uCore-thumips 是针对简化后的 MIPS32 实现的 uCore 移植版本，实现了对应的 Bootloader 和完整的操作系统支持，包括初始化流程、异常处理、内存管理和上下文切换流程，可以更为全面地检验 CPU 设计的正确性。

基于以往项目，我们选用了针对 MIPS32R1 修改后的适配版本，实现了对部分缺失指令的实现和延迟槽的支持。

### 4.3.2 Linux

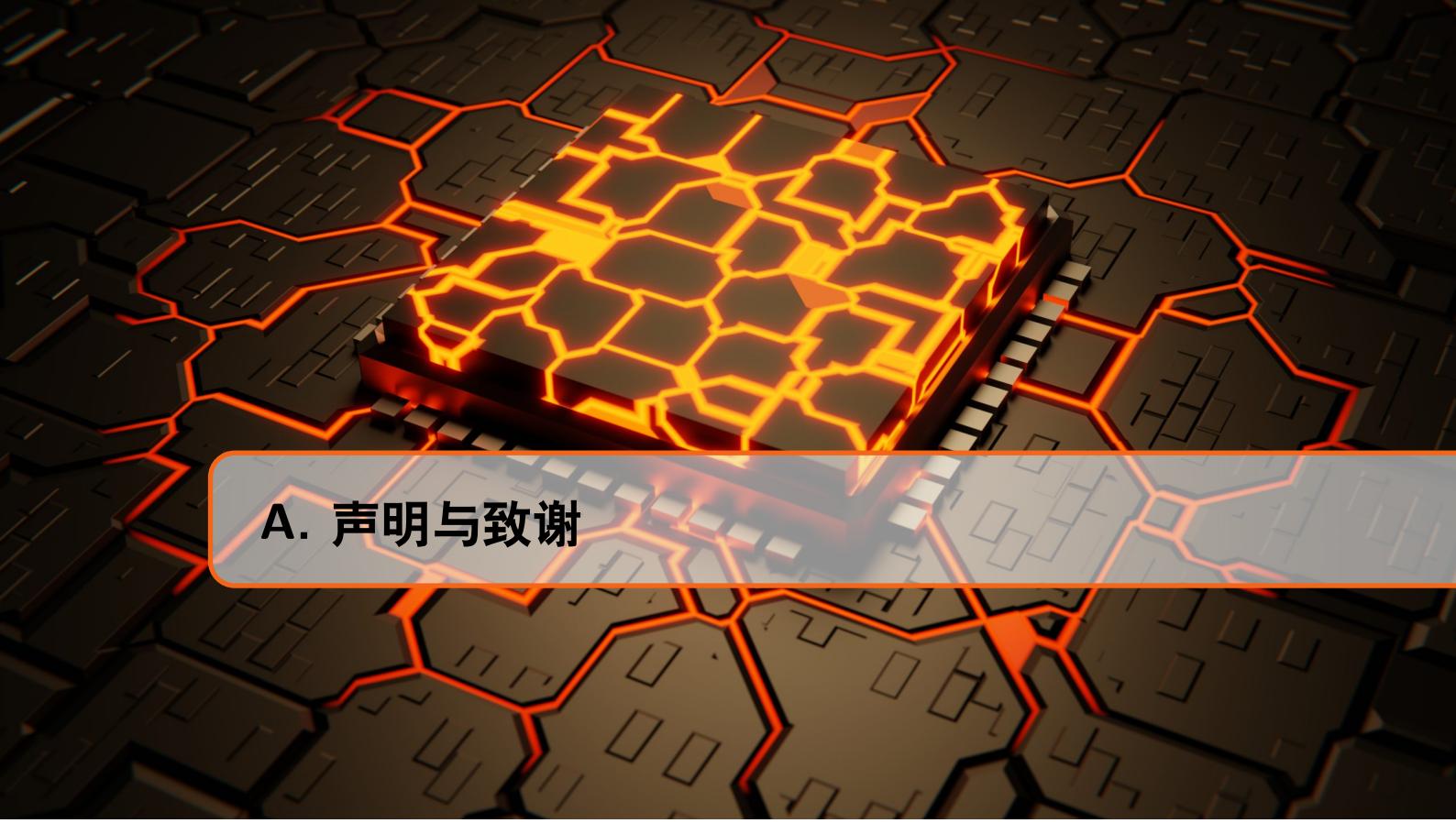
Linux 作为著名的开源系统，提供了丰富的软硬件支持，也是此次比赛中对 CPU 实现正确性的终极考验。在本项目中，我们选择了 Linux 最新发布版本 v5.19-rc4 作为基线进行适配。

除 Linux 内核本身外，我们移植了 VGA 驱动以支持 Framebuffer 和 DMA，同时移植并修复了 LCD 驱动以在 LCD 屏上正确地显示图像。

我们选择了最新稳定版的 Buildroot 进行用户文件系统和用户态程序的构建。构建过程中，我们使用了 musl 作为系统的 C/C++ 标准库实现，并使用 busybox 提供大部分命令行工具。

为了便于演示，我们在文件系统中配置了 micropython 以及一个小游戏 ascii\_invaders，可以直接运行。

值得一提的是，本项目是龙芯杯历史上首次可以完整运行 Linux 的乱序处理器。



## A. 声明与致谢

### A.1 版权声明

本报告著作权归作者所有。您被允许在不作任何修改的情况下重新分发此文档；未经许可，您不得以任何方式复制、引用或演绎其中的任何内容。

### A.2 致谢

本项目的开发过程中得到了许多可贵的帮助。我们要感谢刘卫东、陈康、李山山、陆游游等老师的指导，也要感谢陈嘉杰、高一川、陈晟祺、黄嘉良等学长们的帮助。当然，也要感谢和我们并肩作战的同学们，没有你们，我们不会完成这个项目的设计和开发。