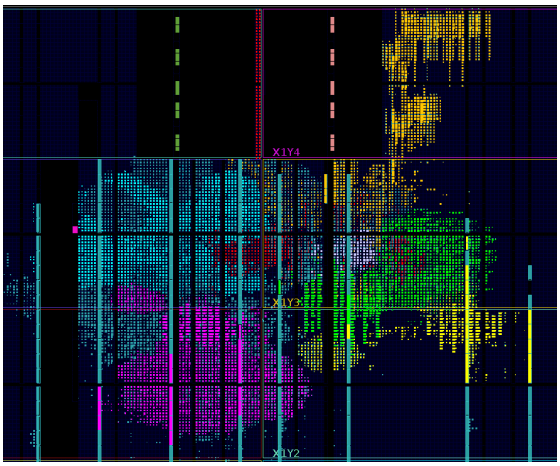
“龙芯杯”第三届全国大学生计算机系统能力培养大赛

清华大学“编程是一件很危险的事情”队[1]

NonTrivialMIPS 项目
决赛设计报告



陈晟祺
harry-chen@outlook.com

周聿浩
miskcoo@gmail.com

刘晓义
circuitcoder0@gmail.com

陈嘉杰
jiegec@qq.com

2019 年 8 月

[1]封面图为 CPU 综合后的资源占用情况，不同颜色代表不同组件

目录

第一部分　概述

1.1　项目背景

本项目的成果是在龙芯提供的 FPGA 实验平台上设计并实现基于 MIPS 32 的 CPU，并使用实验板上的周边硬件，成为一个片上系统（SOC）。其能够支持标准 MIPS 32 Rev 1 指令集的一个较完整子集和 MIPS 32 Rev 2 指令集的部分功能，能够运行功能、性能测试，并能够运行 u-boot 引导器、uCore 操作系统、Linux 操作系统等。

1.2　项目概览

本项目计划设计和实现的部分主要包括：CPU、SoC 设计、Bootloader、系统软件移植、自动化测试。项目使用的硬件语言为 SystemVerilog 2005。下面为各个部分的概览。

1.2.1　CPU

CPU 的设计包含指令集、流水线结构（微架构）、内存管理单元、异常处理机制、协处理器以及其他增强功能。

指令集 本项目的 CPU 实现的指令是 MIPS 32 Rev 1 指令集的一个较完整子集，包括了所有的算术逻辑指令、控制流指令和大部分特权指令（不包括与缓存有关的），覆盖了大赛要求需要的所有 57 条指令。MIPS 32 Rev 2 中的部分指令（如 CP0 中的 ebase 寄存器）由于被操作系统需要，也包含在实现中。

流水线结构 本项目实现了具有 **10 段动态流水线**结构的**超标量（双发射）**处理器：根据流水线上处理指令的不同，以及当前指令与之前指令的数据/访存相关的不同，流水线各个阶段的操作也将不同。大致来说，流水线的各个阶段可以分为取指、译码、发射、执行、访存/延迟执行和写回。其中取指和访存各有 3 个阶段，发射有 2 个阶段。由于访存延迟过长，访存相关的暂停对性能影响较大，我们允许某些指令的执行延迟到访存阶段，即允许在译码阶段操作数未准备好的时候就发射指令，在其后操作数准备好时再执行。为了更好地支持超标量，我们的取指单元较为独立，获取的指令会放入指令 FIFO，同时其有一个分支预测器。取指单元在分支预测成功的情况下保证生成的指令流就是执行所需的指令流，这处理了延迟槽等机制。取指单元一个周期至多会放入 FIFO 三条指令。

内存管理单元 本项目实现了内存管理单元（MMU）以进行从虚拟地址到物理地址的映射，本项目的内存划分遵循 MIPS 32 标准，将使用转换检测缓冲区（TLB）以加速页表的查询，并对所有外设实现内存映射 IO（MMIO）。

异常处理机制 本项目完整支持 MIPS 32 Rev 1 的异常和中断机制，正确处理同步和异步异常，支持硬件和软件中断，并实现精确异常。

协处理器 本项目实现了 MIPS 32 Rev 1 中为 CP0 处理器规定的几乎所有指令和寄存器，以正确运行操作系统。

缓存系统 本项目实现了指令和数据缓存以及 MIPS 32 Rev 1 中要求的缓存控制指令，加快取指和访存。

增强功能 本项目实现了一种可扩展的专用计算模块，它可以完成各类特殊的计算需求，例如密码学算法中的 AES、SHA、MD5 等。各个不同的计算单元通过统一的接口同 CPU 连接，计算单元通过其自身的寄存器与 CPU 交互。各类计算单元仅需要提供一个读写其自身寄存器的接口就可以直接接入 CPU，具体和该模块的交互由软件通过 MFC2 和 MTC2 来完成。

1.2.2 SoC 设计

为了验证 CPU 的设计，并且使用实验箱上提供的硬件进行更多的功能演示，我们以 NonTrivialMIPS CPU 为核心搭建了一个 SoC (System-on-Chip)，具有以下模块：

CPU NonTrivialMIPS CPU （包含 Cache）

DRAM 支持 使用板载 DDR3 SDRAM 作为主存

额外存储 板载 128KB BootROM 和 64KB RAM，用于初级引导程序

Flash 读写 支持软件通过 SPI 协议读取 FPGA 配置 Flash、读写扩展 Flash

串口 16550 兼容的串口控制器，可调节波特率

以太网 使用 Xilinx IP 构建的以太网控制器，实现 100Mbps 通信

GPIO 使用龙芯 confreg 组件，软件控制 LED、数码管，读取开关、按键等状态

图像输出 实现标准的 VGA Framebuffer，并实现了 DMA 硬件加速绘制；支持使用外置 NT35510 LCD 绘制图像

PS/2 支持 PS/2 键盘/鼠标输入

USB 使用板载 USB PHY 提供 USB 2.0 Full Speed 控制器支持，支持人体工程学设备（键鼠）、大容量存储（U 盘）等不同类型的设备

1.2.3 Bootloader

Bootloader 用于引导操作系统，本项目中运行的 Bootloader 分两个阶段，分别是自行编写的 TrivialBootloader 和移植的 U-Boot。前者是被固化在 Bootrom 中的程序，使用 C++ 边下，需要支持从 Flash、SRAM、串口等多途径启动，提供任意地址转储等功能，负责基本的异常处理，并支持内存和外设的检查。而 U-Boot 是被 TrivialBootloader 加载的较复杂的引导程序，支持网络启动、Flash 启动、性能测试等高级功能，需要对源代码进行平台相关移植。

1.2.4　操作系统移植

我们在 SoC 上成功运行了 `uCore-thumips` 操作系统，并通过了其中的一系列测试。同时，我们将最新的 `Linux 5.2.8` 操作系统移植到实验板上，并成功运行；其能够使用 SoC 上的所有外设，包括串口、以太网、USB、图形输出等。我们在 Linux 上成功运行了大量用户态程序，包括 GNU 命令行工具、Xorg Server、Python 解释器等。

1.2.5　自动化测试与部署

本项目还使用了基于 GitLab CI 的自动化综合、测试、部署系统，包括以下的功能：

- 项目需求、设计等文档的自动编译
- 基于事先撰写的 testbench 自动对 CPU、外设和整个板上系统运行 RTL 仿真
- 自动调用 Vivado 生成 Bitstream 文件，并缓存可复用的中间结果
- 所有相应软件的自动编译，使用 QEMU 对操作系统进行测试
- 使用在线实验平台 SDK，在真实环境中运行性能、功能测试和操作系统，并提取数据进行分析和报告

1.3　名词解释

表1.1中是本项目中可能用到的一些名词缩写及它们的解释，以后本项目相关的文档中将不加解释地使用这些缩写。

表 1.1: 名词缩写和解释

缩写	全称	含义
MIPS	Microprocessor without Interlocked Pipeline Stages	无内部互锁流水级的微处理器
CPU	Central Processing Unit	中央处理器
FPU	Floating Point Unit	浮点处理器
CP0/1	Co-Processor 0/1	协处理器 0/1
ALU	Arithmetic Logic Unit	算术逻辑单元
MMU	Memory Management Unit	内存管理单元
TLB	Translation Lookaside Buffer	旁路快表缓冲
PA/VA	Physical/Virtual Address	物理/虚拟地址
ROM	Read Only Memory	只读存储器
(D)RAM	(Dynamic) Random Access Memory	（动态）随机访问存储器
UART	Universal Asynchronous Receiver-Transmitter	通用异步接收器-发射器
GPIO	General-Purpose Input/Output	通用目的输入/输出
MMIO	Memory Mapped Input/Output	内存映射输入/输出
USB	Universal Serial Bus	通用串行总线
SOC	System On a Chip	片上系统

1.4　开发平台

1.4.1　硬件平台

本项目使用的硬件平台为龙芯 FPGA 实验箱，其主要部件为 Xilinx 的 Artix 7 系列 FPGA，型号为 xc7a200t，包含外部器件：

DRAM 128MB DDR3 SDRAM

NOR Flash (CFG) 16MB

NOR Flash (SPI) 32MB

VGA 接口 RGB 输出，各 4 bit

以太网接口 DM9161 100Mbps Ethernet PHY

其他接口 标准 RS232 串口、PS/2 接口、USB PHY

显示屏 NT35510 LCD，分辨率 800 × 480

GPIO 数码管 × 8，LED × 26，拨码开关 × 8，按键 × 19

1.4.2　软件平台

本项目使用 GitLab-CI 进行自动化集成和测试，借助 Docker 保证运行结果可复现。

开发 IDE Xilinx Vivado 2018.3 Web HL Edition

CI 系统 Ubuntu 18.04.1

编译器套件 cross-mipsel-linux-gnu-binutils 2.32-1, cross-mipsel-linux-gnu-gcc 9.1.0-1 (AUR)

1.5　参考资料

本项目的设计、开发过程需要参考包括且不限于下面列出的书籍、文献和资料：

- 计算机组成与设计：硬件/软件接口. David A.Patterson
- *See MIPS Run Linux.* Dominic Sweetman
- 自己动手写 *CPU.* 雷思磊
- *MIPS® Architecture For Programmers I, II, III.* Imagination Technologies LTD.
- *Vivado* 使用误区与进阶. Ally Zhou
- *NaiveMIPS* 设计文档. 张宇翔，王邈，刘家昌
- *32-bits MIPS CPU* 设计文档. 谢磊，李北辰
- 各外设使用手册. 相关厂商

第二部分　CPU

2.1　流水线结构

2.1.1　总体架构

CPU 采用双发射顺序执行，共有 10 级流水。整体的流水线结构如图 2.1 所示。其中为了能够得到较高的频率，指令和数据 Cache 均采用 3 级流水。为了解决访存周期过长而导致的访存暂停过多，我们动态地判断指令的执行时间，允许指令在操作数没有准备好时就发射，并且在操作数准备好时再执行。流水线各个阶段的功能会根据指令类型和数据相关情况动态变化。

各个阶段的功能大致如下：

- **取指 1** 计算当前 PC，发送请求给指令 Cache。
- **取指 2** 指令 Cache 读取数据。
- **取指 3** 对指令进行简单译码，进行简单的分支解析，将指令包放入 FIFO。
- **译码/发射** 从 FIFO 获取指令包，进行译码并且决定是否发射。
- **读操作数** 从寄存器堆以及数据旁路读取操作数。
- **执行** 执行指令。读取存储指令的数据。
- **访存 1/延迟读操作数** 向数据 Cache 发送访存请求。同时处理异常，对于延迟执行的指令进行操作数读取。
- **访存 2/延迟执行** 数据 Cache 计算数据。延迟执行的指令在此时执行。
- **访存 3** 读取访存结果。
- **写回** 写回数据到寄存器堆。

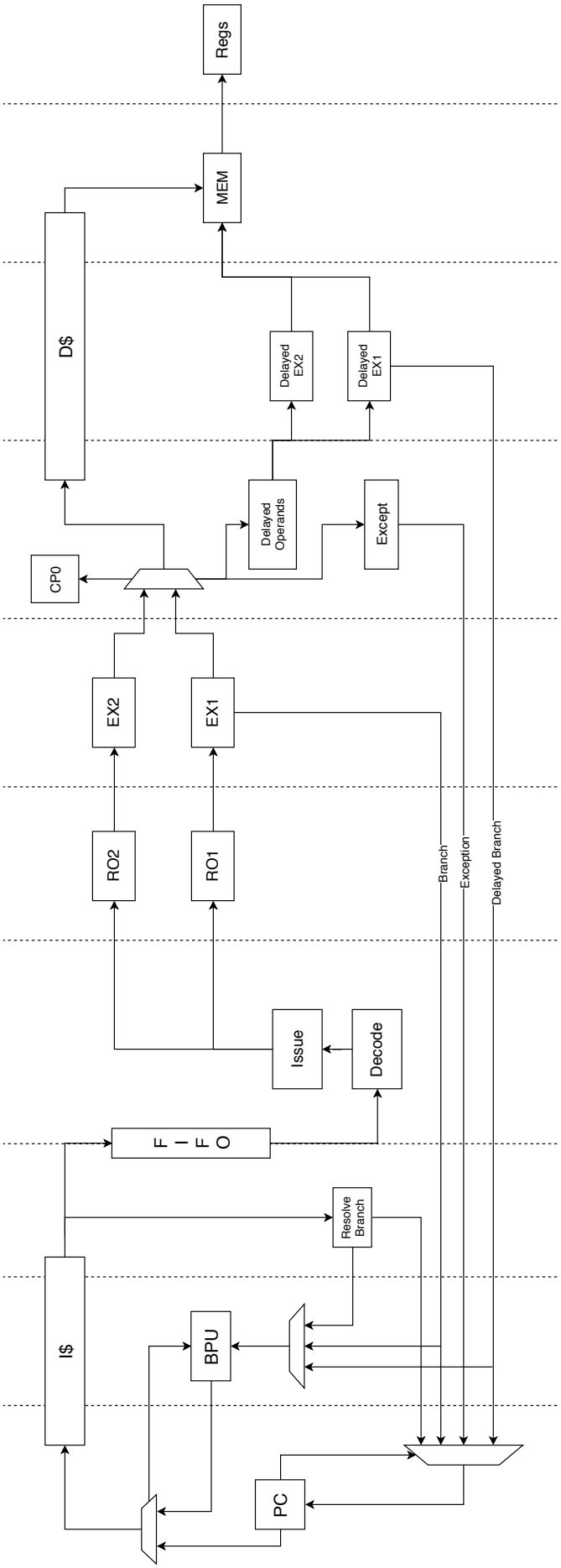


图 2.1: CPU 流水线结构

2.1.2 取指阶段

取指阶段共有 3 级流水，最后得到的指令会放入指令 FIFO。在此 3 阶段中同时会进行分支预测，同时能够保证在分支命中的情况下放入 FIFO 的指令流就是所需要执行的指令流。分支预测失败会刷新指令 FIFO，该信息由执行单元生成。

分支预测采用传统的 2 比特分支预测，BTB 和 BHT 采用 BRAM 实现，延迟为一个周期。

取指的第一个阶段，会计算取指的地址并且发送给 Cache，同时计算下一个 PC 的值。

取指的第二个阶段，会对下一个阶段需要放入 FIFO 的指令包的一部分内容进行计算。

取指的第三个阶段，会对得到的指令进行简单的译码，判断是否为分支同时反馈给分支预测和 PC 生成器。如果将非分支指令预测为分支，那么及时进行流水线刷新。此阶段还会将指令包放入指令 FIFO，至多会有 3 条指令被放入（即第二条指令是分支，并且延迟槽和分支在同一个 Cache 行中）。

当前周期所需要取指的地址通过当前 PC 和 BTB/BHT 得到的分支信息得到，下一个周期的 PC 受到当前 PC，分支预测信息，指令 FIFO 是否满，异常请求和执行阶段所解析的分支信息控制。如果指令 FIFO 满则会丢弃当前数据并且重新在对应为止取指。

为了支持多发射流水线，指令 Cache 要求地址按照 8 字节对齐，每次会返回 64 位的数据，另外如果随后的 64 位数据仍然在 Cache 行内，也会将其返回，这是为了更快地处理延迟槽。由于 MIPS 具有延迟槽机制，我们在分支预测跳转之后还需要将延迟槽也放入指令 FIFO。为了尽量获取更多的指令，如果指令 Cache 返回的 64 或者 128 位数据中有延迟槽，那么当前就可以直接取分支目标的指令，否则需要取延迟槽指令。

2.1.3 译码和发射阶段

译码和发射均在流水线的第 4 个阶段。译码器会从指令 FIFO 中获取对应信息，并且进行译码。

发射阶段较为复杂，其共分为如下几种情况

1. 由于我们的流水线设计中仅有一个乘法器和除法器，乘除指令一个周期内最多只能发射一条。另外，为了设计简便，CP0 相关指令一个周期内也最多只能发射一条。除此之外，由于只有一个访存单元，访存指令在一个周期最多也只发射一条。

2. 如果一个发射包内两条指令存在数据相关，那么只发射第一条指令。

3. 分支指令必须和延迟槽一起发射，如果延迟槽没有取到则需要暂停，如果分支指令是发射包第二条指令则不能发射。

由于访存结果在流水线第 9 个阶段才能得到，如果在译码阶段直接读操作数则访存相关需要 3 个周期的暂停。为了更好地处理访存相关，我们进行部分指令的延迟执行。具体来说，如果指令在译码阶段后不会产生异常（例如 AND，XOR，ADDU 等），我们可以将这类指令的操作数读取延迟到访存的第一阶段，执行延迟到访存的第二阶段。这样访存相关的暂停最好可以缩短到一个周期。更进一步，我们还可以将分支指令也推迟到这个阶段。但是由于分支有可能预测失败，我们需要能够撤销其后的指令。幸运的是，访存请求的提交和异常的触发都是在访存第一阶段，而分支指令的解析是在访存第二阶段，并且其延迟槽不会有异常。这样恰好可以在解析到分支预测失败时刷新流水线，使得其后的指令不会改变处理器状态。

当然，如果指令不能延迟执行，那么仍然需要暂停到操作数均可用为止。此时的访存相关还要考虑正在延迟执行的指令而不仅仅是访存指令。

总的来说，在发射阶段需要暂停的情况大致如下：

1. 指令无法延迟执行，并且操作数没有准备好。这包含访存指令数据没有准备好，以及延迟执行的指令结果没有准备好。
2. 指令可以延迟执行，但是在执行阶段有访存指令。这是真正的无法消除的访存相关。
3. 分支指令但是延迟槽还不在 FIFO 中。
4. 执行阶段正在执行特权指令。这是由于 CP0 的设计原因。
5. 流水线上存在非 MFC0 和 MTC0 的特权指令，这是为了解决可能的 CP0 冒险。

2.1.4 读操作数阶段

指令所需要的两个操作数在第 5 个阶段从寄存器堆和数据旁路读取。对于延迟执行的指令，此时读取的操作数可能是无效的，会在之后再次读取。对于存储指令，存储的数据可能是无效的，会在执行阶段再次读取。

2.1.5 执行阶段

执行阶段是流水线的第 6 个阶段。此阶段会生成指令的结果，同时计算异常请求，计算访存地址。

对于内存写指令（如 SW、SC 等）会在此阶段读取需要写的数据。对于多周期指令，在计算完前会生成一个暂停信号。流水线仅有一个乘法和除法单元。

2.1.6 访存阶段

访存阶段是流水线的第 7, 8, 9 阶段，在第 7 阶段将访存请求发送给数据 Cache，在第 9 个阶段可以得到访存结果。

该阶段和延迟执行阶段是重叠的。

2.1.7 延迟执行阶段

大部分指令都是在译码阶段读取操作数，在执行阶段计算结果。如果指令之间存在访存相关，并且指令是分支或者不会在执行阶段出现异常，那么其可以在流水线第 7 阶段读取操作数，其后一个阶段计算结果。如果是分支指令则会生成分支结果反馈给分支预测器。

该阶段和访存阶段是重叠的。

2.1.8 写回阶段

此阶段写回寄存器请求，是流水线的最后一个阶段。

2.2 指令集

下方按照功能划分列举了 CPU 所支持的 MIPS 指令，各条指令的具体编码以及功能在 MIPS 文档中有详细的描述。

- **自陷指令** TGE, TEGU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI
- **分支指令** BLTZ, BGEZ, BLTZAL, BGEZAL, BEQ, BNE, BLEZ, BGTZ, JR, JALR, J, JAL
- **逻辑指令** AND, OR, XOR, ANDI, ORI, XORI, NOR, SLL, SRL, SRA, SLLV, SRLV, SRAV
- **算术指令** ADD, ADDU, SUB, SUBU, ADDI, ADDIU, MUL, MULT, MULTU, DIV, DIVU, MADD, MADDU, MSUB, MSUBU, CLO, CLZ
- **访存指令** SB, SH, SW, SWL, SWR, LB, LH, LWL, LWR, LW, LBU, LHU, LL, SC
- **特权指令** CACHE, SYSCALL, BREAK, TLBR, TLBWI, TLBWR, TLBP, ERET, MTC0, MFC0
- **条件移动指令** SLT, SLTU, SLTI, SLTIU, MOVN, MOVZ
- **无条件移动指令** LUI, SLT, SLTU, MFHI, MFLO, MTHI, MTLO

2.3 协处理器 0

CP0 是 MIPS 规范中必要的一个协处理器，它提供了操作系统所必须的功能抽象，例如异常处理、内存管理和资源访问控制等。

在 CP0 中有多个 32 位寄存器，各个寄存器均通过 MTC0 和 MFC0 读写。另外，诸如 TLBWI、TLBWR 和 TLBP 等特权指令还有异常的发生也有可能会影响其值。

表2.1中列出了必须实现的 CP0 寄存器。

表 2.1: 必要的 CP0 寄存器

编号	名称	功能
8	BadVAddr	最近发生的与地址相关的异常所对应的地址
9	Count	计数器
11	Compare	计时中断控制器
12	Status	处理器状态及控制
13	Cause	上一次异常的原因
14	EPC	上一次异常发生的地址
15	PRId	处理器版本和标识符
16	Config0	处理器配置
30	ErrorEPC	上一次异常发生的地址

为了实现 TLB MMU 的功能，还需要表2.2中所列出的寄存器。

表 2.2: MMU 所需要的 CP0 寄存器

编号	名称	功能
0	Index	TLB 数组的索引
1	Random	随机数
2	EntryLo0	TLB 项的低位
3	EntryLo1	TLB 项的低位
4	Context	指向内存中页表入口的指针
5	PageMask	控制 TLB 的虚拟页大小
6	Wired	控制 TLB 中固定的页数
10	EntryHi	TLB 项的高位

同时，为了支持自定义异常向量，还需要额外实现一个 MIPS 32 Rev 2 中的 EBase 寄存器。

2.4 中断和异常

2.4.1 中断

MIPS 的中断一共有 8 个，从 0 开始编号。其中 0 号中断和 1 号中断是软件中断，由软件设置 Cause 寄存器中的对应位来触发。其余 6 个中断为硬件中断，由外部硬件触发。在实现中，由 Count/Compare 寄存器组合而成的定时中断的中断号为 7。

如果该中断满足触发条件则会触发异常并且进入中断处理程序。中断的触发要求如下条件全部满足：

1. `Status` 寄存器中对应的中断被打开；
2. 全局中断使能，即 `Status.IE` 为 1；
3. 当前不在异常处理程序中，即 `Status.EXL` 和 `Status.ERL` 为 0。

2.4.2 异常

MIPS 的异常是“精确异常”，也就是在异常发生前的指令都会执行完毕，异常发生之后的指令不会继续执行。在异常发生时，CPU 会跳转到对应的异常向量处执行异常处理代码并设置 CP0 中对应的寄存器记录异常的原因和一些额外的信息，同时还会进入 Kernel Mode。处理异常代码的异常向量由“基地址 + 偏移”来决定，偏移是根据异常来确定的，基地址是由 CP0 的 EBase 寄存器决定。

在我们的实现中，在流水的各个阶段产生了异常，不会马上触发，而是被记录下来，顺着流水直到 MEM 阶段完成后再进行触发。同时，还会考虑双发射两条流水线，优先触发第一条流水线的异常。特别地，异常返回指令 ERET 被实现为一种特殊的异常，保留到 MEM 阶段触发。

需要支持的异常在表2.3中列出。

表 2.3: 主要支持的异常

异常简称	异常说明	异常简称	异常说明
Int	中断	Sys	系统调用
Mod	TLB 修改异常	Bp	断点
TLBL	TLB Load 异常	RI	保留指令
TLBS	TLB Store 异常	CpU	协处理器不可用
AdEL	地址 Load 异常	Ov	算术溢出
AdES	地址 Store 异常	Tr	自陷异常

2.5 内存管理

MIPS 为操作系统的内存管理提供了较为简单的支持，虚拟地址通过 MMU 转换为物理地址。MIPS 标准对虚拟地址和物理地址的映射如表2.4所示。

表 2.4: 虚拟地址空间

段	虚拟地址	权限	物理地址
kseg3/ksseg	0xC000000-0xFFFFFFFF	Kernel	由 TLB 转换
kseg1	0xA0000000-0xBFFFFFFF	Kernel	0x00000000-0x1FFFFFFF
kseg0	0x80000000-0x9FFFFFFF	Kernel	0x00000000-0x1FFFFFFF
useg	0x00000000-0x7FFFFFFF	User	由 TLB 转换

具体的地址转换由 TLB 来完成，TLB 可以认为是在 CPU 内部的地址转换表的高速缓存。具体的内容需要由操作系统来进行填充。如果在 TLB 中没有找到对应虚拟地址则会触发一个 TLB miss 异常，操作系统对该异常处理，并且将对应转换表填入 TLB 中的某一项来完成对地址的转换。我们一共实现了 16 个 TLB 项。

2.6 增强功能

我们实现了一种可扩展的专用计算模块，它可以完成各类特殊的计算需求，例如密码学算法中的 AES、SHA、MD5 等。各个不同的计算单元通过统一的接口同 CPU 连接，计算单元通过其自身的寄存器与 CPU 交互。各类计算单元仅需要提供一个读写其自身寄存器的接口就可以直接接入 CPU，具体和该模块的交互由软件通过 MFC2 和 MTC2 来完成。

CPU 通过 MTC2 和 MFC2 两条指令来访问各个功能部件，其指令格式如图 2.2 所示。

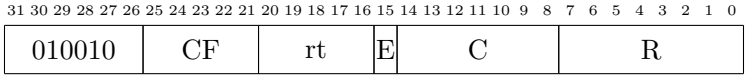


图 2.2: CPU 扩展指令（CP2 协处理器）格式

MTC2 用于将 CPU 寄存器的值移入计算单元的寄存器中，MFC2 用于读取计算单元寄存器值。MTC2 对应的 CF 为 00100，MFC2 对应的 CF 为 00000。其中 rt 表示 CPU 寄存器的编号，E 表示传输的数据的端序是否与 CPU 端序不同。C 表示功能部件的 ID，R 表示计算单元的寄存器编号。

作为示例，我们利用开源的 AES 模块[1]实现了 AES128 和 AES256 的硬件计算单元，其有 8 个密钥寄存器，4 个输入寄存器，4 个输出寄存器，它们分别存放计算 AES 所需的数据。同时，还有一个控制寄存器和一个状态寄存器。控制寄存器用于告诉计算模块数据是否准备好，是否开始计算以及密钥长度等信息。状态寄存器用于获取当前模块是否可用以及计算是否完成。

2.7 缓存设计

CPU 提供指令缓存和数据缓存，分别处理取指和访存请求，并实现了最小化的 CACHE 控制指令。所有的缓存均为虚拟索引物理标签（VIPT）。

2.7.1 指令缓存

指令缓存提供了取指支持，大小和相连度可配置，采用三级流水，其第一个流水段接受请求，并且读取对应的 TAG 和数据。第二个流水段判断是否命中，第三个流水段给出数据。如果不命中，在第三流水段通过 AXI 和外设通信读入。由于采用双发射执行，指令缓存每次读取返回两条指令，并且如果下一条指令也在缓存中的话，也一起取出。

MIPS 要求 CACHE 控制指令必须实现清除特定地址或者索引位置的数据缓存。由于清除每一路对应索引位置的缓存包含了清除特定地址，所以我们将这两个请求都实现为清除了特定索引位置的所有缓存。

2.7.2 数据缓存

数据缓存提供了访存支持，包括经过缓存和不经过缓存两个类型的访存。根据 MIPS 标准要求，对于不同段的访存，数据缓存行为不同。

- **kseg3/ksseg** 按 TLB 判断是否经过缓存
- **kseg1** 不经过缓存
- **kseg0** 可以由 CP0 $\text{Config}_{\text{K0}}$ 寄存器控制，MIPS 标准中这一行为的实现可选
- **useg** 按 TLB 判断是否经过缓存

在 NonTrivialMIPS CPU 中，数据缓存使用延迟写回的实现，写失效时进行写分配，包含写合并操作，大小和相连度可配置。当访存位置的数据在缓存或者写回 FIFO 中时，可以保证在一个周期内完成读写，避免流水线暂停。使用延迟写回可以同时利用 AXI 总线上的读写通道。

数据缓存共包含三级流水，结构如图 2.3 所示。每一路数据缓存中包含两个 RAM 存储，分别包含每个索引位置的标签和数据。为了能够快速读取，标签存储使用 Xilinx Parameterized Macros 生成的 LUT RAM，可以实现在当前周期读出数据。数据 RAM 使用 Xilinx Parameterized Macros 生成的 Block RAM。

数据缓存还包含一个 FIFO，作为延迟写回的队列。为了实现写合并，FIFO 提供一个额外的随机访问接口，可以使用 FIFO 中某一行的地址完成对于 FIFO 的随机读写。

对于读请求，数据缓存的工作流程如下。

[1] https://github.com/secworks/aes

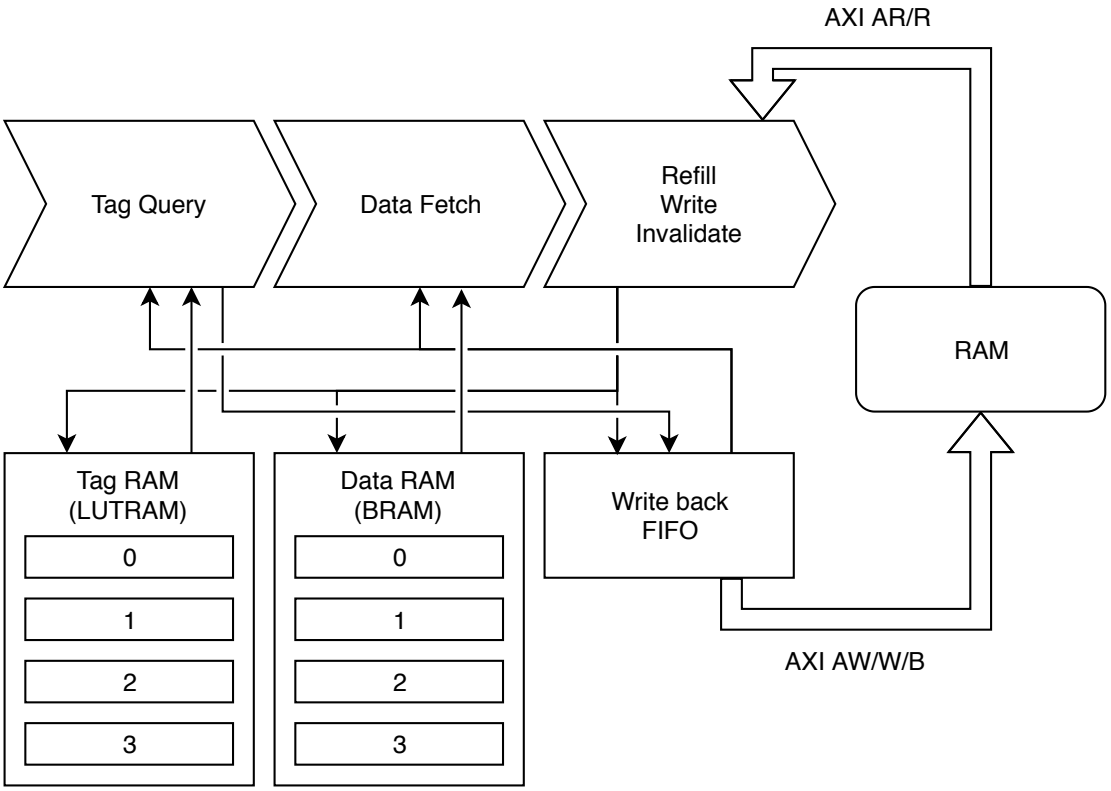


图 2.3: 数据缓存结构

- **第一阶段**完成对于标签的查询，写回 FIFO 的查询，计算得到缓存命中。
- **第二阶段**从数据 RAM 中取得对应索引位置的数据，计算读出的数据。
- **第三阶段**给出数据。如果缓存缺失，通过 AXI 和外设通信，更新缓存，并将覆写位置的脏数据送入 FIFO 中。

对于写请求，数据缓存的工作流程如下。

- **第一阶段**完成对于标签的查询，计算得到缓存命中，完成对于 FIFO 中的写合并操作。
- **第二阶段**得到数据 RAM 对应索引位置的数据，根据 byteenable 计算得出需要写入数据 RAM 的数据。
- **第三阶段**完成对标签、数据 RAM 的写入。如果缓存缺失，则进行写分配，并将覆写位置的脏数据送入 FIFO 中。

MIPS 要求 CACHE 控制指令必须实现清除、写回特定地址或者索引位置的数据缓存。由于清除也包含写回操作，所以和指令缓存相同，我们实现了清除每一路缓存的特定索引位置，并等待脏数据写回完成。

2.8 外部接口

CPU 整体的结构如图 2.4 所示，NonTrivialMIPS CPU 本身暴露三个 AXI 4 Master 接口，分别进行指令读取、数据存取和不过缓存的数据存取。Cache 控制器与 CPU 之间使

用自定义的总线信号进行握手和数据传输，以避免在内部使用 AXI 协议带来的不必要延时；而对外，则将必要的访存转换为符合 AXI 规范的信号，与 SoC 进行交互。

由于预赛 `myCPU` 目录中的 CPU 顶层模块对外只允许暴露一个 AXI 3 Master 接口，我们使用 Xilinx 的 AXI Crossbar IP 将其转换为三个 AXI 4 Slave 接口以适应 CPU 的需要。为了不使得指令预取阻塞访存，crossbar 上的仲裁顺序为 Passthrough > Data Cache > Instruction Cache。在 CPU 直接连接到我们设计的 SoC 时，则不使用这一 crossbar，而是将三个 AXI 4 Master 接口直接连接到 Interconnect 上。

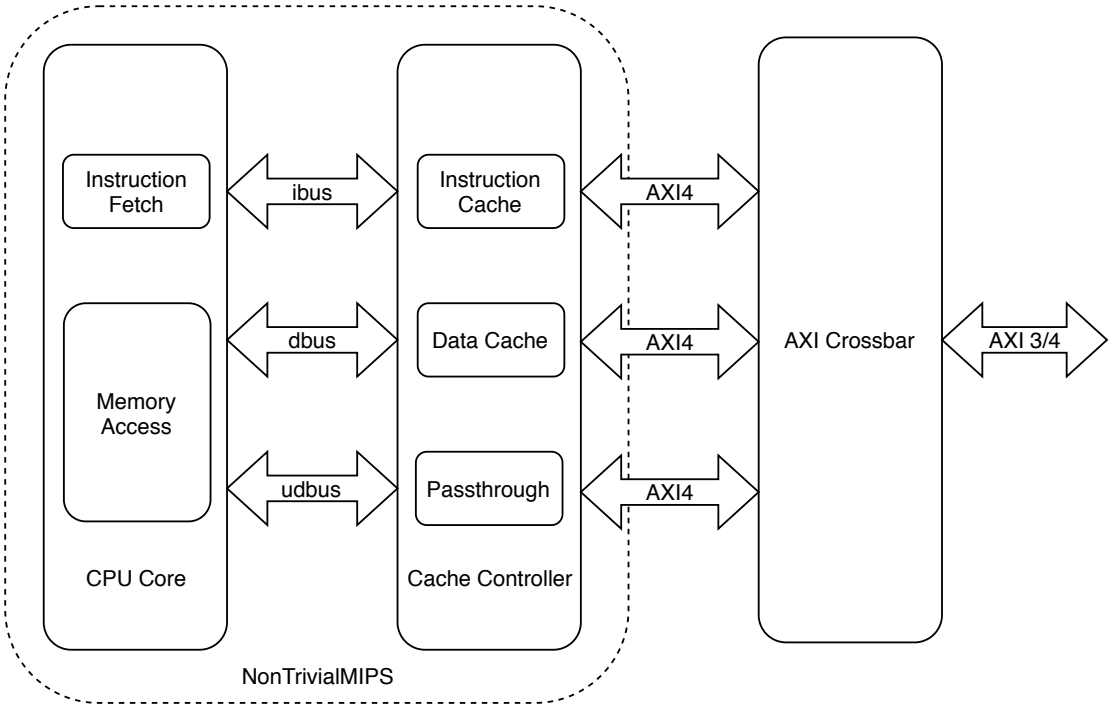


图 2.4: CPU 整体接口架构

第三部分　SoC 设计

3.1　概况

为了更好地利用实验板提供的硬件资源，对 NonTrivialMIPS 进行功能演示，我们搭建了一个较为完整的 SoC，能够在大赛提供的实验箱上运行。

3.1.1　SoC 结构

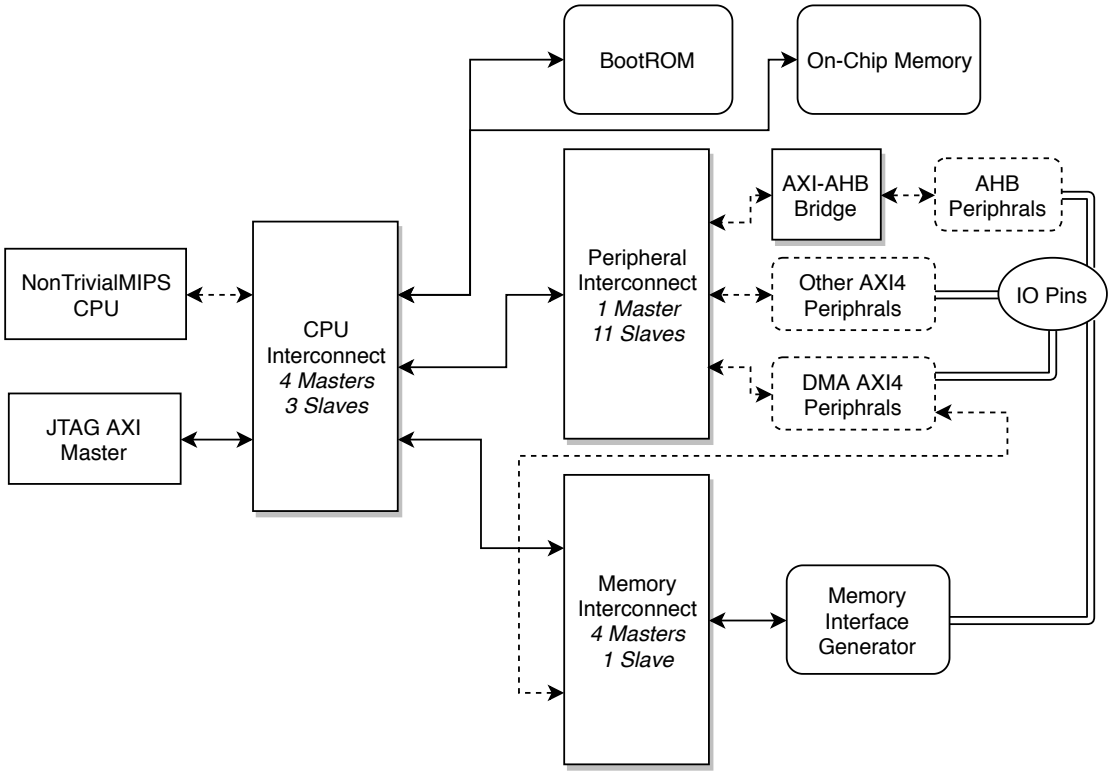


图 3.1: SoC 结构

SoC 由 Vivao 的 Block Design 设计功能组件，项目文件为 `vivado/NonTrivialMIPS.xpr`，总体结构如图 3.1 所示。其中各个组件之间均使用标准的 AXI4 总线协议进行连接，并使用了多个 AXI Interconnect。图中无阴影的矩形表示 AXI Master，带阴影的矩形表示 AXI 总线互联（Interconnect）或适配器（Bridge），圆角矩形表示外设。图中的所有实线表示其为具体设备（及其连接关系），虚线表示由多个具体设备构成的一类设备（及其连接关系），双线表示到 FPGA 片外的连接。

我们将 NonTrivialMIPS CPU 的三个 AXI 接口（指令 Cache、数据 Cache、无 Cache

直通）直接连接到 CPU 互联模块上，不再需要预赛时 CPU 内部的互联模块。这一互联模块上还连接了 Xilinx 提供的 JTAG AXI Master IP，可以绕过 CPU 直接与设备进行通信，方便进行调试。这一互联模块直接连接了 BootROM 与片上 RAM（OCM）两个使用 FPGA 的 Block RAM 资源生成的存储，以及外设互联、内存互联模块。这一设计使得 TrivialBootloader 等程序可以无需运行在内存中，而是在 OCM 中分配堆栈资源。这样，它就可以在内存的任意位置加载操作系统，而不用担心自身代码或数据被覆盖；并且当系统发生不可逆转或者没有捕获的异常/错误时，其依旧可以获取系统控制权，输出必要的调试信息后安全地停止工作。

外设互联模块上连接了除 DDR 内存控制器外所有的外设，其中由于部分外设使用 AHB 协议，我们额外连接了一个 AXI 到 AHB 协议的转换桥。

内存互联模块只有一个从设备，即 Xilinx 的 Mmemory Generator Interface IP 模块。在正确配置后，其能自动对板载的 DDR3 SDRAM 进行训练等操作，并将自己暴露为一片连续的存储空间供用户使用，隐藏了所有实现细节。除 CPU 互联模块外，内存互联的主设备还包含所有有 DMA 需求的外设，包括 VGA 控制器、FrameBuffer 读写控制器等。这样的连接能够让硬件在空闲时直接对内存进行读写，而无需 CPU 干预，大大加速了图形渲染、内存修改等过程。

3.1.2　地址分配

表 3.1 列举了各个设备的物理地址空间分配，其对于总线上的所有 Master 设备都是一致的。

表 3.1: 外设物理地址分配

名称	起始地址	结束地址	有效大小	类型
DDR 控制器	0x00000000	0x07FFFFFF	128 MB	存储
OCM 控制器	0x08000000	0x0800FFFF	64 KB	存储
CFG Flash 控制器（内存映射）	0x1A000000	0x1AFFFFFF	16 MB	存储
以太网控制器	0x1C000000	0x1C00FFFF	64 KB	寄存器
VGA 控制器	0x1C010000	0x1C01FFFF	64 KB	寄存器
PS/2 控制器	0x1C020000	0x1C020FFF	4 KB	寄存器
LCD 控制器	0x1C030000	0x1C030FFF	4 KB	寄存器
SPI Flash 控制器	0x1C040000	0x1C040FFF	4 KB	寄存器
USB 控制器	0x1C050000	0x1C050FFF	4 KB	寄存器
Framebuffer Reader 控制器	0x1C060000	0x1C06FFFF	64 KB	寄存器
Framebuffer Writer 控制器	0x1C070000	0x1C07FFFF	64 KB	寄存器
外部中断控制器	0x1D000000	0x1D00FFFF	64 KB	寄存器
BootROM 控制器	0x1FC00000	0x1FC1FFFF	128 KB	存储
串口控制器	0x1FD02000	0x1FD03FFF	8 KB	寄存器
GPIO 控制器	0x1FF00000	0x1FF0FFFF	64 KB	寄存器

3.1.3　中断连接

一些外设需要通过中断告知 CPU 自身的状态而完成 IO 操作。MIPS 规范中 CPU 至多支持 5 个外部中断（编号为 2 到 6），并且 NonTrivialMIPS 只支持电平触发的中断。因此，我们使用了 Xilinx Interrupt Controller IP 来对类型不同或者超出数量限制的中断进行统一转换和管理。表 3.2 描述了这些中断的连接关系。

表 3.2: 中断连接关系

名称	中断类型	接收方	中断编号
串口控制器	高电平	CPU	2
PS/2 控制器	高电平	CPU	3
AXI 中断控制器	高电平	CPU	6
以太网控制器	上升沿	AXI 中断控制器	0
SPI Flash 控制器	上升沿	AXI 中断控制器	1
CFG Flash 控制器	上升沿	AXI 中断控制器	2
USB 控制器	高电平	AXI 中断控制器	3

3.2　硬件支持

FPGA 板载了较多的物理接口和硬件资源，为了充分利用，达到较好的展示效果，我们均对它们进行了适配。

3.2.1　串口控制器

板载串口为 RS-232 接口，我们使用 Xilinx 提供的 AXI UART 16550 IP 实现了一个标准的 NS16550 UART 控制器。其支持比较完整的串口协议，包括可变波特率、流量控制和读写缓冲区。在软件使用串口前，需要首先对多个配置寄存器进行初始化，配置波特率、中断等不同的属性，方可正常与 PC 进行通信。串口控制器有一个中断，指示收到了数据，它被直接连接到 CPU 的第一个硬件中断上。

3.2.2　USB 控制器

实验箱上的 USB PHY 为 Microchip USB 3500 ，支持 Host/Device/OTG 三种模式下的 USB 2.0 协议。它提供的是 UTMI+ Level 3.0 的接口，我们从实验箱的原理图中找到了对应引脚在 FPGA 上的绑定，并添加到约束文件中。由于 USB 3500 仅实现了物理层的收发逻辑，我们以开源的 UltraEmbedded USB 1.1 Host Controller IP 为基础，进行了大量的修改以满足使用需求。具体工作将工作频率调整为 USB 2.0 Full Speed 指定的 60MHz、调整 IFS 逻辑、添加输出延迟约束等等。此外，我们也需要移植驱动到 U-Boot 和 Linux 中，这将在下面的系统软件部分中详述。

3.2.3　以太网控制器

USB

实验箱上提供了一个 10/100 Mbps 的以太网 PHY（物理接口），引出了以太网协议中规定的标准的 MII 和 MDIO 接口，用于数据传输和物理芯片配置。我们使用了 Xilinx 提供的 AXI Ethernet Lite IP 实现以太网控制器，在 U-Boot 和 Linux 中都有对应的驱动。这一控制器不支持 DMA，因此实际的数据传输无法达到 100 Mbps 全速工作。事实上，经过测试，速度最高可达到 2MB/s 左右，已经足够；考虑到内存带宽分配问题，我们没有使用完整的 Xilinx Ethernet MAC 和 DMA IP。由于该 IP 仅提供上升沿触发的中断，我们使用中断控制器来转换中断类型。

3.2.4 Flash 控制器

实验室上共有两片 SPI NOR Flash 芯片，其中一片固化连接至 FPGA 配置专用硬件逻辑，另一片是普通 SPI I/O 引脚，可以插拔更换 Flash 芯片。我们都使用 Xilinx 提供的 AXI Quad SPI IP 来进行控制。

对于配置 Flash（即 CFG Flash），其主要作用是存储 FPGA 的 bitstream，在上电时自动对 FPGA 进行配置。其总容量为 16 MB，而 bitstream 只会占用 6 到 7 MB 的空间，因此我们在剩余空间中存储了 U-Boot 和 Linux Kernel 的 ELF 格式文件。为了方便地使用配置 Flash，通过 IP 的 XIP（Execute In Place）特性，我们可以将它映射为一块只读的内存空间，从而 TrivialBootloader 可以直接从中加载 U-Boot，U-Boot 也能加载 Linux 内核。

另一块可插拔的 Flash（称为 SPI Flash）被我们用作通用的存储，因此 XIP 的只读模式不能满足需求。我们将其暴露为一个标准的 SPI 控制器，由软件来进行相应的管理。U-Boot 与 Linux 都有 SPI 协议和 Flash 读写、MTD 设备支持。通过测试，我们能够正确地对 Flash 进行擦除、读取、写入等操作。同时，为了 IO 更高效，我们还将 IP 提供的中断引出，通过中断控制器连接到 CPU。

3.2.5 LCD 控制器

实验板板载了一块 NT35510 LCD 屏幕，分辨率为 800×480。它事实上是一个通用的异步类 SRAM 设备，有一个指令寄存器和一个数据寄存器，可以通过信号控制，分别进行读写。我们使用了 Verilog 实现的 NT35510 控制器（感谢张宇翔同学早先的移植），将两个寄存器分别映射到两个地址，通过 APB 协议（转换后为 AXI 协议）即可控制屏幕。控制器只是实现了到屏幕的通信，其本身的控制逻辑比较复杂，我们会在下面的驱动移植部分中较为详细地叙述。

3.2.6 VGA 控制器

实验板提供了标准的 VGA 接口，我们使用 Xilinx 提供的 Thin Film Transistor Controller IP 来实现图象的输出。它事实上提供了一个标准的 Framebuffer，可以通过控制 AXI 信号写入起始地址等配置，而在被启动后，通过 DMA 的方法不断从内存中读取图像数据，转换为符合 VGA 时序的信号输出。其输出的图像分辨率固定为 640×480，24 位真彩色。由于提供的 DAC 实际深度小于 24 位，我们将较低位舍弃，对图像质量没有明显影响。

由于 Framebuffer 将被 Linux 作为控制台和主视频输出使用，而使用过程中将频繁发生像素的拷贝（如窗口移动、终端滚屏等），因此我们添加了 Xilinx 的 Framebuffer Reader 和 Framebuffer Writer 这两个 IP，用 DMA 的方式完成内存的拷贝。在两者的 AXI Stream 协议之间，我们添加了一个简单的 RTL 模块，可以将途径其的像素数据替换为常数或者异

或一个常数（这也是 Linux 经常进行的任务，用于绘制一个矩形或者阴影等）。这一硬件组件可用于加速 Framebuffer 的绘制，也需要 Linux 中相应软件驱动的配合，具体将在下面章节中说明。

3.2.7　PS/2 控制器

实验板载一个 PS/2 物理接口，可以用于连接较老的鼠标/键盘等设备。我们使用 Altera 提供的 University Program PS/2 IP 对其进行控制，在 Linux 中也有对应的驱动支持。由于其为 APB 总线，同样需要 Bridge 进行转换；它的中断信号被直接连接到 CPU 的 3 号中断引脚。

3.2.8　GPIO 控制器

实验班上还有较多的 GPIO 设备，包括拨码开关、案件、数码管、单双色 LED 等。由于龙芯提供的 `confreg` 模块已经较好地对这些外设进行了统一控制和管理，还提供了额外的计时器、虚拟串口等功能，并包装为 AXI 接口。考虑到没有必要进行重复的实现，我们直接使用了这一模块。需要注意的是，由于其硬编码了一些参数（包括一些信号宽度），需要进行一些修复，才能直接使用在 Block Design 中。

第四部分　引导程序（Bootloader）

4.1　第一阶段：TrivialBootloader

作为板上系统的一部分，BootROM 中包含了系统每次上电或重置时都会首先执行的代码，起始物理地址为 `0x1FC00000`。由于这部分程序是固化在 FPGA 中的，为了节约有限的板载存储，Bootrom 中的代码不能太大。因此，有必要撰写一个较小的 Bootloader 来进行初步的系统初始化和加载工作，将其命名为 TrivialBootloader。

4.1.1　汇编部分

作为程序入口，在跳转到高级语言编写的代码之前，需要先使用汇编语言设置一些基本参数，如栈基址（`sp`）寄存器和异常处理入口等。具体地，汇编部分具有如下功能：

- 系统的全局初始化，设置 `sp`，`gp` 寄存器，跳转到实际代码入口
- C++ 代码退出后的清理与提示
- 启用异常处理并设置异常向量

其中实际代码入口为 `_main` 符号，`sp`，`gp` 寄存器被分别设置到 `_stack`，`_gp` 符号。而异常处理会跳转到 `_exception_handler` 符号。这些符号需要在链接时被填充。

4.1.2　高级语言部分

高级语言部分也分为两个 C 与 C++ 两个子部分。C 编写的部分主要用于提供 `_main` 函数和 `_exception_handler` 函数。前者具有打印欢迎信息、清零 BSS 段、执行 `main` 函数、打印返回值等功能，后者负责处理异常，从 CP0 中收集异常原因、异常地址等信息，以友好的方式打印出来。

C++ 部分是 Bootloader 的主体，也也是真正的 `main` 函数实现位置，其具有的功能包括：

内存检测 通过不同块大小随机读写检测内存硬件与实现是否存在问题

ELF 启动 支持从非易失存储（Flash）中读取合法的 ELF 文件头，正确地将其复制到内存的相应位置并跳转

直接启动 支持直接跳转到内存（`0x8000000`）、片上内存（`0x8800000`）入口点启动，便于系统移植时的调试工作

串口旁加载 支持从串口直接向内存中加载数据和指令，并跳转到指定的位置启动

内存转储 支持将指定的内存区域的数据转储到串口输出

异常处理 正确处理各种操作异常、非法情况（如没有选择启动模式、内存检测失败、要复制到内存的数据覆盖了 Bootloader 本身的代码），在发生异常时通过串口、LED 等多种途径给出友好可读的提示

考虑到指令和数据 Cache 的存在可能会引发数据不一致，TrivialBootloader 在拷贝代码前将通过 CP0 `Config0` 寄存器中相应位关闭 `kseg0` 段的全部缓存，并在跳转到外部代码前再打开缓存。通过这一简单的处理，能够保证在执行前代码已经被全部写入到相应存储器中。

作为功能举例，在引导 Flash 中操作系统的 ELF 文件时，Bootloader 输出为（没有启用内存检查）：

```
*****TrivialMIPS Bare Metal System*****
Compilation time: 19:37:09 Jul 25 2019
=====Entering TrivialBootloader=====
Bootloader used memory: from 0x88000000 to 0x88010000
Mode: Boot
Device: SPI Flash
Valid ELF file found, will now copy to RAM.
Copying 24 bytes from offset 0x19480 to address 0x80019400
Copying 2159744 bytes from offset 0x80 to address 0x80000000
Booting from address 0x80000000...
=====Exiting TrivialBootloader=====
```

4.1.3　链接脚本

由于 Bootloader 可能被放置在不同位置，此时其本身可用的内存空间（作为栈）以及加载基址是不同的。因此，我们编写了如下的链接脚本。可以看到，我们将内存划分为高低两片，根据代码生成的位置选择不同的区域放置栈代码段和数据段。此外，可用内存范围、BSS 段范围、需要检查的内存范围等都是由链接器提供的符号。其可以通过编译时定义不同的常量进行预处理，从而为高级语言提供关于平台的信息，灵活地适应多种要求。

附录 B.1 中包含了当前使用的链接脚本。

4.2　第二阶段：U-Boot

4.2.1　背景

U-Boot 是一个启动引导程序，常见于嵌入式系统中，用于引导 Linux 等操作系统。通过运行 U-Boot 引导程序，可以支持从 Flash、U 盘、网络等来源加载 uCore、Linux 系统镜像到内存并进行引导。由于 U-Boot 本身有较强的命令行功能和交互能力，它也可以作为一个硬件测试与演示的工具。在本系统的设计中，U-Boot 将作为二级引导程序，放置在 Flash 中，被 TrivialBootloader 所加载。

4.2.2　硬件需求

U-Boot 对 CPU 的功能要求较低，它不使用 MIPS 的中断和 TLB 机制，因此硬件可以不需要实现这些机制。对于其它的异常，仅仅在程序运行不正常时才会发生，如果假定程序能正常运行，对异常处理也没有要求。

作为功能丰富的引导程序，其将用到 SPI Flash、网络等外设。其中，网络控制器的正常工作是至关重要的，否则片上系统将失去从外部加载系统的功能。

4.2.3　编译方法

U-Boot 可以直接使用1.4.2节中给出的编译器套件进行编译和调试。具体地，只需要从 `https://github.com/Harry-Chen/u-boot-trivialmips` 下载源代码，并运行下列命令：

```
make CROSS_COMPILE=mipsel-linux-gnu- nontrivialmips_thinpad_defconfig
make CROSS_COMPILE=mipsel-linux-gnu-
mipsel-linux-gnu-strip u-boot
```

执行完后，即可生成最终的 ELF 可执行文件 `u-boot`，将其写入配置 Flash 的 bitstream 之后的位置，并正确配置 TrivialBootloader 中的 Flash 镜像起始地址，即可被自动加载。

4.2.4　移植内容

U-Boot 与 Linux 内核源码的组织架构类似，也都采用了 `DTS`（设备树源码）来描述设备，因此移植方法也比较类似。主要的移植工作主要分为两部分，一部分是添加 CPU 相关的 SoC 支持，一部分是添加板级支持。

由于之前的类似项目已经有了完成度较高的工作[1]，本项目在其基础上进一步进行修改。主要的工作有：

- 更新到 U-Boot v2019.7 版本
- 启用串口控制器驱动（包括系统启动早期和后期两个部分）
- 添加定时器读取功能，准确反映运行时间
- 编写 DTS 格式设备描述以准确反映 SoC 片上设备资源，加载相应驱动（包括网络、Flash 等）
- 利用 `bootmenu` 命令，增加启动菜单，提供引导至 uCore/Linux 操作系统、通过 DHCP/TFTP 协议网络引导、进入 U-Boot 控制台等选项
- 调整 U-Boot 编译选项，将 Flash 管理、网络配置等工具内嵌于镜像中

[1] `https://github.com/z4yx/u-boot-naivemips/`

第五部分　系统软件

5.1　uCore-thumips 操作系统

5.1.1　系统概述

uCore-thumips[1] 是针对简化后的 MIPS 32 实现：MIPS32S 平台的 uCore 移植版本。该项目针对 MIPS32S 平台实现了对应的 Bootloader、初始化流程、异常处理、内存管理和上下文切换流程。相比标准的 MIPS 32，MIPS32S 缺少部分指令且不支持延迟槽。针对这些不同，uCore-thumips 对 uCore 操作系统的编译选项进行了相应的修改，并提供了额外的库函数实现缺失的指令（如 `divu`）的功能。

5.1.2　编译方法

在非 mipsel 平台编译、调试 uCore-thumips 需要使用面向 mipsel 架构的交叉编译、调试工具链，所需工具主要包括 binutils、gcc 和 gdb。

Debian 系统下，`gcc-mipsel-linux-gnu` 和 `binutils-mipsel-linux-gnu` 软件包分别提供了预编译的目标平台为 mipsel 的 binutils 和 gcc。其它操作系统的工具链可参考 LinuxMIPS 项目文档[2]自行编译。此外 Sourcery CodeBench Lite[3] 提供了预编译的 mipsel 工具链。

交叉编译时，指定 `CROSS_COMPILE` 环境变量或修改 Makefile 中 `CROSS_COMPILE` 变量为所使用的交叉编译器，即可使用 make 进行编译。编译后得到镜像 `ucore-kernel-initrd` 和 `boot/loader.bin` 分别为系统内核 ELF 和 Bootloader。

进行移植时，需针对片上系统对 Makefile 中相应配置进行修改，包括延迟槽、浮点模块等编译选项、为用户 App 预留存储大小等。

5.1.3　系统分析

启动流程

uCore-thumips 的引导、启动流程主要分为 Bootloader 加载系统、初始化 C 环境、初始化系统三个步骤。

uCore-thumips 提供了简易的 Bootloader `boot/bootasm.S`，该程序从 Flash（默认为地址 0xBE000000）读取合法的 ELF 文件头，将其复制到内存的相应位置并跳转。

Bootloader 加载系统后将跳转至 `kern/init/entry.S` 中的 `kernel_entry` 过程。在此过程中，系统将重置 CP0 中异常相关寄存器、设置 TLB 相关异常向量；同时，正确设置 `sp`，

[1] `https://github.com/z4yx/ucore-thumips`

[2] `https://www.linux-mips.org/wiki/Toolchains`

[3] `https://sourcery.mentor.com/GNUToolchain/release2189`

gp，清空 bss 以满足 C 程序运行要求，之后跳转至 kern/init/init.c 中的 kern_init 函数。kern_init 函数将完成中断控制、控制台、异常、内存管理、进程管理等系统功能的初始化。

进行移植时，需将 Bootloader 替换为针对 NonTrivialMIPS 片上系统自行实现的 TrivialBootloader 或 U-Boot，针对平台对中断控制、控制台等功能的初始化过程进行相应修改。

内存管理

MIPS32 使用软件进行 TLB 缺失处理，当发生 TLB 缺失时会触发 TLB Refill 异常。uCore-thumips 已经实现了 TLB Refill 异常的处理。发生 TLB 缺失时，系统会首先检查页表判断是否为缺页，若为缺页调用 do_pgfault 进行处理，否则检查权限后填充 TLB 表项。

异常处理

异常处理程序通过访问 CP0 中的 Cause 寄存器获取异常信息，同时需要正确设置 Status 寄存器中的某些位。用户态和特权态切换时，uCore 内核使用 trapframe 结构存储程序运行状况。uCore-thumips 已实现和 CP0 中寄存器的交互及 trapframe 的保存。

5.1.4　移植内容

编译选项

针对 NonTrivialMIPS 平台，我们对原版 ucore-thumips 的 Makefile 进行了如下修改。由于 NonTrivialMIPS 平台实现了 CP1 浮点运算协处理器，我们关闭了编译选项中软浮点数的开关；由于 CPU 支持延迟槽，我们允许编译器在延迟槽中生成代码。

外设配置

为使 uCore 正确操作外设，我们对外设相关常量进行配置。相关常量集中在内核源码中的头文件 kern/include/thumips.h，修改的常量包括串口、等设备对应的内存地址，以及串口、时钟等设备的 IRQ 号。

内存管理

我们修改了 uCore 的 TLB 替换策略，轮流选择 TLB 项进行替换，使用 tlbwi 指令。对 TLB 进行 reset 时，也将 TLB 的项数改为实际的项数。

另外，在移植过程中，我们发现了 uCore 存在的一个问题。在进程内存管理组件 pmm.c 的内存拷贝函数 copy_range 中，如果内存不足导致分配的 npage 为空，原本的实现会直接导致内核崩溃。根据系统 API 语义，此时实际应该返回 -E_NO_MEM。按此修复后，系统内核不会在应用程序申请较多内存导致内存不足时崩溃，而是会正确地杀死用户态程序。

5.1.5　新增内容

ulib 字符输入库函数

ulib 中只实现了字符输出的库函数，没有实现输入的相关函数。为了使用户态程序可以方便地与用户交互，我们为 ulib 增加了 fgetch、getchar 以及 readint 三个库函数。

`fgetch` 从指定文件读取一个字符，`getchar` 从标准输入读入一个字符，`readint` 从标准输入读入一个十进制整数。

外设通信与系统调用

外设所映射到的内存地址在内核态无法访问，为此需在内核中增加访问外设的系统调用。我们加入了 `sys_pread` 和 `sys_pwrite` 两个通用的外设访问系统调用。其中 `sys_pread` 从外设读取数据，传入的参数以此为外设序号、读取数据的目的地址、读取数据的长度；`sys_pwrite` 将数据写入外设，传入的参数为外设序号、写入数据的来源地址、写入数据的长度。在 ulib 中，我们也加入了对这两个系统调用的封装，便于用户态程序使用。我们额外在 ulib 中加入了对硬件计时器访问的封装 `int check_timer(uint32_t* time)` 与 `int set_timer(uint32_t time)`。

Mandelbrot 集绘制演示程序

Mandelbrot 集指是的 $z_0 = 0,\ z_{n+1} = z_n^2 + c$ 收敛的复数 c 的集合。我们在 uCore 用户态实现了绘制 Mandelbrot 集的演示程序，用以验证 NonTrivialMIPS 平台的浮点计算及图像输出能力。可证明，若 c 属于 Mandelbrot 集，则 $|z_n| \leq 2$。在实际计算时，若经过 `maxIteration` 次迭代仍有 $|z_n| \leq 2$，则近似认为 c 属于 Mandelbrot 集。对于平面上的每个点 c，将使得 $|z_n| > 2$ 的最小的 n 映射到可显示的 256 种颜色之一，即可绘制 Mandelbrot 集的图像。

利用 ulib 中新增的字符输入库函数，该演示程序可与用户交互，用户通过按键进行图像的平移与缩放。

视频播放演示程序

我们实现了一个简单的视频播放演示程序，可以从 Flash 中读取视频文件，输出到帧缓冲区中显示。我们首先对图像进行二值化，用 1 个 bit 表示一个像素，之后使用基于 LZ77 的压缩算法对二值化后的视频进行压缩。压缩算法的原理为，维护一个固定长度的窗口，从窗口中找到接下来输入中的最长匹配，用最长匹配在窗口中的位置、长度以及下一个字节表示这一最长匹配，最后将最长匹配加入窗口。伪代码如下：

```
triplets = []
window = Queue(maxlen=WIN_SIZE)
i = 0
while i < len(data):
    for j in range(i+WIN_SIZE, i, -1):
        if data[i: j] in window:
            offset = window.index_of(data[i: j])
            length = j - i
            break
    triplets.append((offset, length, data[j]))
    window.push_back(data[i:j])
    i += length
```

解压过程是压缩过程的逆过程，这里不再叙述。实际实现时，三元组中的每个元素都用 8 位固定长度整数存储，`WIN_SIZE` 取 255。最终压缩后，视频体积为原二值化视频的 4%，长度为 30s、帧率 15fps、分辨率为 800*600 的黑白视频体积为 1MB。

幻灯片播放程序

在视频播放的基础上，我们编写了幻灯片播放程序。与视频播放的不同之处在于，为实现前后翻页，幻灯片的每一页都分别进行压缩，并在文件头部依次以 8 位无符号数保存总页数及各页数据长度。另外，幻灯片的色彩为 4 位灰度，使用 2 个二进制位表示一个像素。

5.2　Decaf 教学编程语言

5.2.1　概述

Decaf 是编译原理课程使用的实验性编程语言，编译原理课程提供了 Java 编写的 Decaf 编译器，能将 Decaf 源文件编译为 MIPS32 平台汇编。我们对操作系统、编译器进行了一些修改，使得 Decaf 程序可以在 uCore 的用户态正确运行。由于 NonTrivialMIPS 实现了完整的 MIPS32 Rev I 指令集，我们无需对编译器后端指令生成部分进行修改，主要工作集中于适配 Decaf 标准库。

最终，我们在 NonTrivialMIPS 平台上成功运行了 `blackjack, math, nqueens, fibonacci` 四个 Decaf 应用程序。

5.2.2　Decaf 标准库移植

Decaf 中涉及到与操作系统交互的部分为 8 个标准库函数，定义与功能详见表5.1。

表 5.1: Decaf 标准库函数

名称	功能
Allocate	分配内存，如果失败则自动退出程序
ReadLine	读取一行字符串 (最大 63 个字符)
ReadInt	读取一个整数
StringEqual	比较两个字符串
PrintInt	打印一个整数
PrintString	打印一个字符串
PrintBool	打印一个布尔值
Halt	结束程序

为使得 Decaf 程序正确运行，需要为 uCore 系统构建 Decaf 标准库。我们选择将 libdecaf[4] 移植到 uCore 平台。在 libdecaf 中，我们使用 C 语言，通过调用 ulib 中的相关函数，实现 Decaf 程序的输入、输出、退出功能，具体层次结构见图5.1。

Allocate 在 ulib 中没有对应的实现。由于 Decaf 只分配内存而不释放，我们在 libdecaf 的中增加了一个静态数组作为 Decaf 程序的堆，调用 Allocate 时，首先将长度进行四字节对齐，然后返回数组相应元素的地址。

[4] `https://github.com/z4yx/libdecaf`

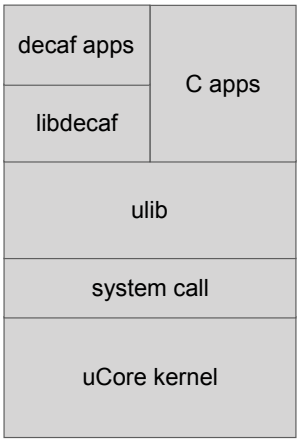


图 5.1: Decaf 标准库实现层次结构

5.2.3　与平台约定衔接

Decaf 进行函数调用时，会将参数从右到左依次压入栈中，而在 uCore 系统中前四个参数分别保存在 a0 到 a3 寄存器中，其余参数压入栈中。因此，为了使得 Decaf 程序能够调用 C 编写的 libdecaf 中的函数，需对 Decaf 的标准库函数调用进行封装。我们使用汇编语言编写标准库函数调用的封装（`decafCall.S`），在调用时将参数从栈中取出，保存在对应的寄存器中，然后调用 libdecaf 的对应函数，以完成 ABI 的转换。以 `PrintString` 函数为例，核心代码如下所示。

uCore 的用户态程序正常退出时应返回（在 v0 寄存器中保存）0，而 Decaf 程序的主函数没有返回值。为使其正确返回 0，我们修改了 Decaf 编译器的后端，将 Decaf 程序主函数的名称修改为 `__decaf_main`。同时，在 libdecaf 中加入 main 函数，调用 `__decaf_main` 并返回 0。

我们将 Decaf 程序的编译加入了 uCore 系统的编译过程。在编译用户态程序时，首先检查 `user` 目录下是否存在 Decaf 源文件，若存在则调用编译原理课程提供的 Decaf 编译器将源文件编译为 MIPS 汇编，进而使用汇编器将其转换为目标文件，目标文件与 libdecaf 和 ulib 链接。uCore 中的用户态程序的入口为 ulib 中的 `umain` 函数，这一点通过在链接时指定链接脚本 `user/libs/user.ld` 完成。

5.3　Linux 操作系统

Linux 是最为著名的开源操作系统，有丰富的软硬件支持。我们以 Linux 最新的稳定版本（v5.2.8）作为基线进行移植。

5.3.1　CPU 适配

我们以 Linux 内核内置的 MIPS 4Kc CPU 为基础来实现移植，这是一款标准的 MIPS32 R1 CPU。首先我们需要添加 `TRIVIALMIPS` 这个架构，而后添加对应的架构目录（参见其他架构和 CPU 的目录结构即可）。

由于我们的 CPU 缺少一些功能（如 `Watch` 寄存器和 `PERF` 指令等），我们可以通过两种方式去除 Linux 的依赖。一方面，在 `arch/mips/include/asm/mach-trivialmips/` 目

录下的 cpu-features-overrides.h 中可以通过定义类似 cpu_has_watch 的宏为 0，去除一些特定指令；另一方面，可以通过 arch/mips/trivialmips/Platform 文件来指定编译选项，如使用 -mno-branch-likely 去除 Branch Likely 等未实现的指令。

由于 NonTrivialMIPS 按照规范实现了 TLB 相关的 CP0 寄存器（包括 Index，Wired，EntryHi，EntryLo，Random，PageMask 等）与指令（包括 TLBWI，TLBWR，TLBR 等），因此无需修改 Linux 进程调度修改的代码。此外，Linux 也能够正常从 CP0 的 Config1 寄存器中得到 Cache 的相关属性（如大小、相连度）。对于浮点运算，当 Linux 检测到不存在 CP1 协处理器时，会自动使用软件模拟这些指令。因此，在 NonTrivialMIPS CPU 上运行 Linux 无需进行任何实际的指令简化工作。

5.3.2　板级硬件适配

除 CPU 适配之外，还需要在 Linux 内核代码树中进行下列的改动，以使得其能够在我们设计的 SoC 上工作：

初始化代码 这部分代码均位于 arch/mips/trivialmips 目录下，由多个 C 文件组成。需要完成的工作包括在引导早期阶段（即 early_printk 使用的输出方式）初始化串口控制器并提供一个简单的输出函数（prom_putchar），向内核注册 CPU 的中断控制器和时钟源，初始化设备树供后面的阶段使用。

设备树描述 现代 Linux 使用设备树描述（Device Tree Description）来实现通用的驱动管理与配置，因此我们需要为开发板编写专门的设备树。所有的设备树都位于架构对应的 boot/dts 目录中，我们撰写了 trivialmips/trivialmips_nscscc.dts，其中描述了板载所有设备的信息，包括型号、寄存器地址分配、中断号与连接关系等。

默认内核配置 内核默认配置保存在 arch/mips/configs/ 目录中，我们提供了一个配置文件 trivialmips_nscscc_defconfig，其中启用了实验板必须的驱动，以及调整了运行相关配置。

附录 B.3 中包含了我们当前使用的 DTS 文件，其中描述了各个外设的信息供内核中的驱动程序进行匹配。

此外，我们还将 Linux 在默认图形终端（tty1）上绘制的 Tux 企鹅 Logo 更换为清华大学校徽，以突出我们的移植工作。

5.3.3　驱动移植

Linux 中已经包含了相当多设备的驱动，只需要正确编写 DTS 文件提供信息即可使用。然而，有一些外设的驱动依旧需要移植或者修改。具体地，我们完成了下列的工作：

USB 控制器驱动 USB 控制器的驱动移植自 ultra-embedded 的开源代码，我们也进行了必要的修改以适应硬件的不同。驱动文件位于 drivers/usb/host/ue11-hcd.c，能够将自己注册为 Linux 的标准 USB 2.0 Full Speed Host Controller。经过测试，该控制器驱动可以成功让 Linux 正确识别并操作挂接在 USB 3.0 Hub 下的 HID 设备和 Mass Storage 设备，能够使用键盘在图形终端进行输入、使 xeyes 程序响应鼠标事件，并可以挂载 U 盘上的存储分区并读取数据。

LCD 控制器驱动 板载的 NT35510 LCD 需要进行恰当的初始化以工作，每次写入数据时也需要先发送控制命令。我们从 `https://github.com/z4yx/linux-kernel` 处移植了一个可用的 NT35510 LCD 屏幕驱动，它会在 `/dev` 中创建名为 `nt35510` 的字符设备，只需向其写入 RGB565 格式的原始像素数据即可显示在屏幕上。同时，它还支持 `seek` 操作，可以修改屏幕的某个偏移处的像素数据，而无需全部刷新。

Framebuffer 驱动 内核自带的 `xilinxfb` 驱动能够正常使用，但是重绘等操作均使用内核自带的的 `cfb_copyarea, cfg_fillrect` 等函数，由 CPU 进行数据的拷贝，速度较慢。我们在驱动中重写了这些操作，调用我们在硬件中添加的 DMA 引擎（即 Framebuffer Reader 与 Writer）辅助完成这一工作，以提升控制台重绘的速度。

完成驱动移植后，我们能够在 Framebuffer (VGA) 上绘制一个速度较快的图形终端，并使用 USB 键盘在终端中进行交互。同时，也能够在 LCD 屏幕上绘制任意的图象。

5.3.4　用户态组件

Linux 内核本身并不能提供任何用户态组件，因此我们需要手工编译这一部分。我们选择了著名的嵌入式 Linux 开发工具套件 Buildroot 来协助完成用户程序的构建。构建过程中，我们选择了 `musl` 作为系统的 C/C++ 标准库实现，并使用 `busybox` 来提供大部分的命令行工具。

由于构建的 rootfs 较大，无法使用 initramfs 等方式直接加载，而 Flash 的读写速度也较慢并不灵活，因此我们使用 NFS 协议通过网络挂载系统的根分区。实践证明，在网络稳定的情况下，系统的响应速度并不会受到影响。

大部分 Linux 用户程序在我们的实验板上都可以正常运行，包括 GNU Coreutils、Python 解释器、网络工具（`ip, ping, mtr, wget, nc`）等。由于我们实现了加速的 Framebuffer，因此 Xorg Server 能够正常初始化图形设备，运行 `xscreenserver, xeyes, xterm` 等经典的 `XWindow` 图形程序，并且它们都能正确处理来自 USB 键盘、鼠标的输入。同时，我们也能够基于 Qt 等现代化图形框架正常编写程序进行演示。

5.3.5　示例：软硬件协同设计（AES 加速）

`OpenSSL` 是著名的开源密码学算法库。为了利用第 2.6 节中提到的 CPU 内置的密码学加速单元，我们修改了 `OpenSSL 1.1.1c` 的源代码，采用我们编写的 C 代码替换原有的汇编实现。为了替换 AES 加密算法，共需要修改四个函数，其定义及功能见表 5.2 所示。

表 5.2: OpenSSL AES 实现

名称	功能
`AES_set_encrypt_key`	设置加密密钥，支持 128 位和 256 位两种密钥长度
`AES_encrypt`	对 32 字节的块进行 AES 加密
`AES_set_decrypt_key`	设置解密密钥，支持 128 位和 256 位两种密钥长度
`AES_decrypt`	对 32 字节的块进行 AES 解密

我们在代码中提供了这四个函数的实现，通过在 C 代码中内嵌汇编（见附录 B.2）向 AES 协处理器交换数据，包括写入密钥，设置密钥长度和模式，输入数据并读出数据，按照函数结构进行内存的读写。为了使得加速更加显著，由于加密算法默认采用大端序，而

CPU 运行在小端序，我们在 mfc2 指令中选择一个位指示由硬件进行端序翻转。最终测试结果如表 5.3 所示，最大加速比可达 11 倍左右，提升显著。

表 5.3: OpenSSL AES（采用 `aes-128-cbc` 模式）加解密性能比较

块大小	原吞吐量 (MB/s)	硬件加速后吞吐量（MB/s）
16 Bytes	0.85	5.94
64 Bytes	0.97	8.91
256 Bytes	1.00	10.52
1024 Bytes	1.01	11.02
8192 Bytes	1.01	10.51

由于 `OpenSSL` 提供的密码学算法库 `libcrypto` 被系统中的多个程序（如 `ssh`, `scp`, `wget`）所使用，因此这一修改也能被应用到这些工具软件中，取得同样显著的性能提升。

5.3.6　演示程序：TrivialDashboard

为了充分利用 SoC 实现的功能和优化，我们使用 Qt5 图形框架编写了演示程序 TrivialDashboard，可以监控本机或者远程计算机上 CPU 和网络资源的利用率，可以响应用户的鼠标、键盘输入，并且内置一个虚拟终端。

远程连接通过 SSH 完成，我们选择了 `LibSSH` 作为协议支持库。`LibSSH` 可以选择链接到 OpenSSL 提供的 `libcrypto`，从而利用上一节中提到的 AES 硬件优化。在开启优化后可以看到显著的传输速率提升。

我们使用开源的 `qtermwidget` 项目提供虚拟终端组件，该组件支持鼠标选中、滚动、搜索和在程序内复制粘贴。

在渲染时，程序通过使用 Qt5 提供的 `linuxfb` 驱动，直接跳过 XWindow 和 Xorg Server，在 Framebuffer 层完成用户界面绘制，可以充分利用硬件 DMA 优化，在开启抗锯齿之后仍能达到较高的刷新率。

这一演示程序在 Windows 上开发，大量使用了 C++17 的特性，包括使用 `std::mutex` 和 `std::unique_lock` 进行连接的同步操作。代码可以直接交叉编译之后在 SoC 上运行的 Linux 系统中正常工作。

第六部分　自动化测试

为了保证实现的正确性，本项目进行了自动化的集成、测试与部署。所有的流程都通过 Docker 进行，确保是可完整复现的。

6.1　硬件测试

本项目使用的主要硬件设计语言 SystemVerilog 是一门强大的验证语言，我们用它编写 testbench 来测试硬件模块。主要的测试用例分为有：

CPU 测试 本部分用于测试 CPU 实现指令的正确性。我们对 CPU 的各条指令都编写了相应的测试程序，同时还对各类可能的冲突现象、异常、TLB 的行为编写了对应的测试。测试的过程是通过一个 testbench 虚拟出外部的总线和 RAM 并且接入 CPU，并对 CPU 中的访存动作，包括 WB 阶段对寄存器和的写请求和 MEM 阶段的内存写请求进行监视。每个测试用例对上述动作都会给出响应期待的结果，同时在运行 testbench 时会将监测到的真实的写请求和期望的结果进行对比进而确认程序执行的正确性。整个 CPU 的测试过程是自动化的，通过指定格式的汇编代码即可生成对应的存储文件（`.mem` 文件）、答案文件（`.ans` 文件）；我们编写了用于执行上述动作的 SystemVerilog task，可以直接使用 Vivado 对所有用例依次执行测试，得到比较结果，无需人工介入观察信号，如果发现真实的运行和期望不符会进行报告。所有的测试用例以及解释可见表6.1。除此之外，我们还在 CPU 仿真部分引入了 Verilator 这一基于 C++ 的编译型仿真工具，用于简单的测试，其性能相比解释型工具有数量级上的增强。

Cache 测试 我们生成不同特征的访存序列（顺序/随机），并捕获真实应用程序的访存序列，将其作为激励传递给 Cache 组件，观察其行为是否与预期一致，本部分用于测试 Cache 的正确性。此部分还使用了一个 AXI RAM 的行为模型，用于替代 Xilinx AXI Block RAM Generator IP 核，并能模拟测试中人工插入的延迟。

同时，我们还为龙芯功能测试、性能测试编写 tcl 脚本并修改 testbench，以在 CI 环境中能够进行自动化测试和结果提取。最后，还可以自动生成上述测试的 bitstream 文件，供直接上板测试。

对于主分支的每一次提交，都需要进行持续集成（CI），步骤包括进行 IP 核的生成与预综合、上述的测试，以及 bitstream 的生成。由于完整仿真速度较慢，通常只运行 CPU 测试部分。

表 6.1: CPU 测试用例（位于 `testbench/cpu/testcases` 目录中）

文件名	测试内容
except/*	异常相关测试
instr/*	指令功能测试
branch/*	分支测试
hazard/*	各类边界情况测试
across_tlb/*	TLB 测试
performance/*	性能测试

6.2　软件测试

在软件方面，本项目计划对编写的所有汇编/C/C++ 代码，移植的 Bootloader、操作系统，以及需要运行的功能测试、性能测试，均编写基于 GitLab CI 的持续集成脚本，保证每个版本都能进行正确的、可重现的编译。

插图索引

表格索引

附录 A　声明与致谢

A.1　版权声明

本项目涉及的代码均在 GitHub 开放，相应的仓库列举如下（名称即为链接）：

TrivialMIPS 初版的 CPU 设计（运行在清华大学 ThinPad 实验板上）

NonTrivialMIPS 最终提交的 CPU 设计（运行在龙芯实验板上）

Software C++ 编写的裸机（Baremetal）程序，包括 TrivialBootloader 等

U-Boot 移植的 U-Boot 引导程序

uCore 移植的 uCore 操作系统

Linux 移植的 Linux 内核

OpenSSL 适配硬件 AES 加速功能的 OpenSSL 程序

TrivialDashboard 使用 Qt 撰写的 Dashboard 演示程序

这些项目均遵循它们特定的开源许可证，某些目录中可能包含受版权保护的内容，使用它们意味着您知晓并愿意承担任何可能的法律责任。

A.2　致谢

本项目开发过程中得到了来自清华大学计算机系张宇翔同学的大力支持，我们在此表示衷心的感谢。此外，我们的指导教师陈康老师、刘卫东老师给我们提出了许多宝贵的建议，唐适之同学、王邈同学、刘家昌同学也向我们提供了帮助，在此一并向他们表示感谢。

附录 B　代码摘录

B.1　TrivialBootloader 使用的链接脚本

```
ENTRY(_start);

MEMORY
{
    BOOTROM  (rx)  : ORIGIN = 0xBFC00000, LENGTH = 128K
    OCM      (rwx) : ORIGIN = 0x88000000, LENGTH = 64K
    RAM      (rwx) : ORIGIN = 0x80000000, LENGTH = 128M
}

#ifdef CODE_INTO_BOOTROM
#define TEXT_AREA BOOTROM
#define DATA_AREA OCM
#else
#define TEXT_AREA OCM
#define DATA_AREA RAM
#endif

SECTIONS
{
    . = ORIGIN(TEXT_AREA);

    _mem_start = ORIGIN(DATA_AREA);
    _mem_end = ORIGIN(DATA_AREA) + LENGTH(DATA_AREA);

#ifdef CODE_INTO_BOOTROM
    _mem_avail_start = ORIGIN(RAM);
    _mem_avail_end = ORIGIN(RAM) + LENGTH(RAM);
#else
    _mem_avail_start = _stack;
    _mem_avail_end = ORIGIN(RAM) + LENGTH(RAM);
#endif
```

```
32
33      .text :
34      {
35          _text = .;
36          *(.text.startup)
37          *(.text*)
38          *(.rodata*)
39          *(.reginfo)
40          *(.init)
41          *(.stub)
42          *(.gnu.warning)
43          *(.MIPS.abiflags)
44          _text_end = .;
45      } > TEXT_AREA
46
47      .data :
48      {
49          _data = .;
50          _stack = _data + LENGTH(DATA_AREA) - 32;
51          *(.data)
52          *(.data*)
53          *(.eh_frame)
54          _gp = ALIGN(16);
55          *(.got.plt) *(.got)
56          *(.sdata)
57          *(.lit8)
58          *(.lit4)
59          _data_end = .;
60      } > DATA_AREA
61
62      .sbss :
63      {
64          *(.sbss)
65          *(.scommon)
66      } > DATA_AREA
67
68      .bss :
69      {
70          _bss = .;
71          *(.dynbss)
72          *(.bss)
73          *(COMMON)
```

```
        _bss_end = .;
    } > DATA_AREA

    /DISCARD/ :
    {
        *(.note*);
        *(.iplt*);
        *(.igot*);
        *(.rel*);
        *(.comment);
    }
}
```

B.2　扩展计算单元的寄存器读写

```
#define write_aes_register_bigendian(addr, variable)        \
  asm volatile(".byte " #addr "\n"                          \
               ".byte 0x81\n"                               \
               ".byte 0b10000000+(%0-0x100)\n"              \
               ".byte 0b01001000\n"                         \
               :                                            \
               : "r"(variable));

#define read_aes_register_bigendian(addr, variable)         \
  asm volatile(".byte " #addr "\n"                          \
               ".byte 0x81\n"                               \
               ".byte 0b00000000+(%0-0x100)\n"              \
               ".byte 0b01001000\n"                         \
               : "=r"(variable));

#define write_aes_register(addr, variable)                  \
  asm volatile(".byte " #addr "\n"                          \
               ".byte 0x01\n"                               \
               ".byte 0b10000000+(%0-0x100)\n"              \
               ".byte 0b01001000\n"                         \
               :                                            \
               : "r"(variable));

#define read_aes_register(addr, variable)                   \
  asm volatile(".byte " #addr "\n"                          \
               ".byte 0x01\n"                               \
```

```
                ".byte 0b00000000+(%0-0x100)\n"             \
                ".byte 0b01001000\n"                        \
                : "=r"(variable));
```

B.3　移植 Linux 使用的 DTS 文件

```
/dts-v1/;

#include <dt-bindings/interrupt-controller/irq.h>

/ {
    compatible = "TrivialMIPS,NSCSCC";
    #address-cells = <1>;
    #size-cells = <1>;

    chosen {
        bootargs = "console=ttyS0,115200n8 console=tty1 rootfstype=squashfs
        ↪  root=/dev/mtdblock1";
    };

    aliases {
            serial0 = &serial0;
        };

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            device_type = "cpu";
            compatible = "mips,4Kc";
            clocks = <&ext>;
            reg = <0>;
        };
    };

    ext: ext {
        compatible = "fixed-clock";
        clock-frequency = <80000000>;
        #clock-cells = <0>;
    };

```

```
35      memory {
36          device_type = "memory";
37          reg = <0x0 0x8000000>;
38      };
39
40      cpuintc: interrupt-controller@0 {
41          #address-cells = <0>;
42          #interrupt-cells = <1>;
43          interrupt-controller;
44          compatible = "mti,cpu-interrupt-controller";
45      };
46
47      soc {
48          compatible = "simple-bus";
49          #address-cells = <1>;
50          #size-cells = <1>;
51          #interrupt-cells = <1>;
52          ranges;
53
54          serial0: serial@1fd02000 {
55              device_type = "serial";
56              compatible = "ns16550a";
57              reg = <0x1fd02000 0x1000>;
58              reg-offset = <0x1000>;
59              reg-io-width = <4>;
60              reg-shift = <2>;
61              current-speed = <115200>;
62              clock-frequency = <100000000>;
63              interrupt-parent = <&cpuintc>;
64              interrupts = <2>;
65          };
66
67          lcd0: lcd@1c030000 {
68              compatible = "lcd,nt35510";
69              reg = <0x1c030000 0x1000>;
70          };
71
72          axi_quad_spi: spi@1c040000 {
73              #address-cells = <1>;
74              #size-cells = <0>;
75              compatible = "xlnx,xps-spi-2.00.a";
76              interrupt-parent = <&axi_intc_0>;
```

```
77              interrupts = <1>;
78              reg = <0x1c040000 0x10000>;
79              xlnx,num-ss-bits = <0x1>;
80              num-cs = <0x1>;
81              fifo-size = <256>;
82
83              flash@0 {
84                  compatible = "mx25l25635f", "jedec,spi-nor";
85                  reg = <0x0>;
86                  spi-max-frequency = <30000000>;
87
88                  spi-rx-bus-width = <4>;
89                  m25p,fast-read;
90
91                  partitions {
92                      compatible = "fixed-partitions";
93                      #address-cells = <1>;
94                      #size-cells = <1>;
95
96                      partition@10000 {
97                          label = "bootenv";
98                          reg = <0x10000 0x10000>;
99                      };
100                     partition@20000 {
101                         label = "rootfs";
102                         reg = <0x20000 0x1fe0000>;
103                     };
104                 };
105             };
106         };
107
108         axi_intc_0: interrupt-controller@1d000000 {
109             #interrupt-cells = <1>;
110             compatible = "xlnx,xps-intc-1.00.a";
111             interrupt-controller;
112             interrupt-parent = <&cpuintc>;
113             interrupts = <6>;
114             reg = <0x1d000000 0x1000>;
115             xlnx,kind-of-intr = <0x15>;
116             xlnx,num-intr-inputs = <0x4>;
117         };
118
```

```
119         axi_ethernetlite: ethernet@1c000000 {
120             compatible = "xlnx,xps-ethernetlite-3.00.a";
121             device_type = "network";
122             mac-address = [19 98 00 01 00 29];
123             phy-handle = <&phy0>;
124             reg = <0x1c000000 0x10000>;
125             xlnx,duplex = <0x1>;
126             xlnx,include-global-buffers = <0x1>;
127             xlnx,include-internal-loopback = <0x0>;
128             xlnx,include-mdio = <0x1>;
129             xlnx,instance = "axi_ethernetlite_inst";
130             xlnx,rx-ping-pong = <0x1>;
131             xlnx,s-axi-id-width = <0x1>;
132             xlnx,tx-ping-pong = <0x1>;
133             xlnx,use-internal = <0x0>;
134             interrupt-parent = <&axi_intc_0>;
135             interrupts = <0>;
136             mdio {
137                 #address-cells = <1>;
138                 #size-cells = <0>;
139                 phy0: phy@1 {
140                     device_type = "ethernet-phy";
141                     reg = <1>;
142                 } ;
143             } ;
144         } ;
145 
146         ps2: ps2@1c020000 {
147             compatible = "altr,ps2-1.0";
148             reg = <0x1c020000 0x1000>;
149             interrupt-parent = <&cpuintc>;
150             interrupts = <3>;
151         };
152 
153         usb: usb@1c050000 {
154             compatible = "ue11-hcd";
155             reg = <0x1c050000 0x1000>;
156             interrupt-parent = <&axi_intc_0>;
157             interrupts = <3>;
158         };
159 
160         axi_tft_0: axi_tft@1c010000 {
```

```
161             compatible = "xlnx,xps-tft-1.00.accl";
162             reg = <0x1c010000 0x1000>, // TFT controller
163                 <0x1c060000 0x1000>, // framebuffer read
164                 <0x1c070000 0x1000>, // framebuffer write
165                 <0x1ff08000 0x4>; // video stream modifier control
166             xlnx,dcr-splb-slave-if = <0x1>;
167             resolution = <640 480>; // actual video size
168             virtual-resolution = <1024 480>; // framebuffersize
169             phys-size = <640 480>; // don't care (physical size of screen)
170         };
171
172     };
173 };
```