

北京理工大学

本科生毕业设计(论文)

基于 RISC-V 和 rCore 的计算机系统实验设计

Experimental design of computer system based on Rust and
rCore

学 院:	信息与电子学院
专 业:	电子信息工程
学生姓名:	范志鹏
学 号:	1120161371
指导教师:	白霞

20XX 年 月 日

原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名：

日期：

年

月

日

关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名：

日期：

年

月

日

指导老师签名：

日期：

年

月

日

基于 RISC-V 和 rCore 的计算机系统实验设计

摘 要

本论文的主要内容是研究在资源有限的硬件平台上如何进行计算机硬件和软件的协同设计,并探索如何将新型的语言 Rust 和架构 RISC-V 应用到这个开发过程中。从硬件和软件两个方面入手,将专业课程的内容进行衔接。

在硬件方面,通过在 RISC-V 软核的基础上添加串口、SDRAM 等多种外设,搭建了一个功能齐全的 SoC。该 SoC 不仅能在仿真环境下正确运行,更能通过 CYC10-FPGA 开发板在真实环境下对已搭建的 SoC 进行功能测试。

在软件方面,综合应用汇编、C 语言、Rust 语言对 BBL, rCore 操作系统进行移植,最终实现了软件模拟 TLB 替换、格式化输出、中断处理、ELF 解析、虚拟内存管理、线程切换、用户进程等功能。

从总体上看,成功地在自己搭建的 RISC-V 架构的 SOC 硬件平台上移植了基于 Rust 语言的 rCore 操作系统。

关键词: Rust; RISC-V; 操作系统; CPU; BBL

Experimental design of computer system based on Rust and rCore

Abstract

The main content of this paper is to study how to co-design computer hardware and software on a hardware platform with limited resources, and explore how to apply the new language Rust and architecture RISC-V to this development process. Starting from the two aspects of hardware and software, the content of the professional courses is connected.

In terms of hardware, by adding slots, SDRAM and other peripherals on the basis of RISC-V soft core, a fully functional SoC is built. The development board performs functional test on the built SoC under the real environment.

In terms of software, comprehensive application assembly, C language, Rust language transplantation of BBL, rCore operating system, and finally realized software simulation TLB replacement, formatted output, interrupt processing, ELF analysis, virtual memory management, thread switching, user Process and other functions.

Key Words: Rust; RISC-V; Operating System; CPU; BBL

目 录

摘 要	I
Abstract	II
第 1 章 绪论	1
1.1 研究目标与研究意义	1
1.1.1 研究目标	1
1.1.2 研究意义	1
1.2 研究现状	错误!未定义书签。
1.3 论文结构	1
第 2 章 研究现状与技术分析	3
2.1 RISC-V 软核.....	4
2.1.1 picorv32	4
2.1.2 openriscv ^[9]	4
2.2 Rust 语言.....	7
2.2.1 包管理器.....	7
2.2.2 支持多种语言调用	7
2.2.4 函数式编程	7
2.2.5 强大的宏系统	8
2.2.6 生命周期	8
2.2.7 并发的安全性	9
2.3 BBL	9
2.4 rCore 操作系统.....	10
2.5 本章小结	10
第 3 章 环境搭建与工具介绍	11
3.1 FPGA 硬件平台.....	11
3.1.1 STEP-CYC10 FPGA 开发板.....	11
3.2 Verilog 综合与仿真工具.....	13
3.2.1 Quartus	13

3.2.2 Modelsim	13
3.3 RISC-V 交叉编译链	13
3.3.1 交叉编译	13
3.3.2 RISC-V 交叉编译链说明	14
3.3.3 RISC-V 交叉编译链安装	15
3.3.4 编译链测试	16
3.4 Rust 编译链	16
3.4.1 Rust 编译链安装	16
3.4.2 Rust 编译链测试	17
3.5 本章小结	18
第4章 硬件平台搭建	19
4.1 SoC 模块组成与设计	19
4.1.1 封装 openriscv	20
4.1.2 ROM 控制器	20
4.1.3 SDRAM	21
4.1.4 总线转换桥	23
4.1.5 UART 控制器	24
4.1.6 CONFIG	26
4.2 SOC 硬件平台功能验证	26
4.2.1 testbench 搭建	27
4.2.2 测试用例	27
4.2.3 添加测例	28
4.3 本章小结	31
第5章 软件分析移植	32
5.1 软件分析移植	32
5.1.1 压缩原理介绍	32
5.1.2 解压缩代码适配	33
5.2 BBL 分析与移植	34
5.2.1 BBL 解析	34
5.2.2 BBL 修改	36

5.3 操作系统分析与移植	42
5.3.1 最小化内核	43
5.3.2 中断管理	45
5.3.3 页表管理	48
5.3.4 线程切换	51
5.3.5 用户进程	54
5.4 本章小结	57
第6章 实验结果与展望	58
6.1 RISC-V 指令测试.....	58
6.2 rCore 与用户程序.....	58
6.3 未来展望	错误!未定义书签。
总结	63
参考文献	64
附 录	65
致 谢	66

第 1 章 绪论

1.1 研究目标与研究意义

1.1.1 研究目标

本工程的研究目标是 1. 在 CYC10 FPGA 开发平台上搭建起基于 RISC-V CPU 的 SOC，验证系统功能的正确性。2. 在此基础上修改 bootloader 和操作系统来适配本系统，使得在硬件系统上可以运行起操作系统。3. 在硬件资源有限的情况下，探索用软件模拟，已达到相同功能的可行性。4. 使用新语言 Rust，新架构 RISC-V 来构建本项目，发掘使用这些新技术带来的优势。

1.1.2 研究意义

工程的研究意义有两方面，首先是可以将本科所学到的一系列知识有效地串联起来，以实验为主线，加深学生对计算机软硬件理解。在硬件开发方面，能够应用本科期间所学的课程如《数字电路基础》、《微机原理与接口技术》等课程知识；在软件开发方面，可以应用《C++程序设计》、《数据结构》、《嵌入式系统》等课程知识，这样通过这个实验系统就可以将本科期间所学的知识进行融会贯通。

其次，在现实生活中的很多领域如物联网，嵌入式等，由于成本考量或实际尺寸等，软硬件开发经常面临着资源不足的问题。在本工程中的一个重点就是探究高效利用 FPGA 开发板的资源，并将有些功能如 TLB 替换，除法等移动到软件部分模拟实现。这样虽然速度稍微变慢，但是在有限资源的情况下保证了整个系统的功能正确性，这是十分有现实意义与应用场景的。

1.2 论文结构

本论文在第一章中首先介绍 RISC-V, Rust 语言和 FPGA 应用的基本知识和本实验的研究目标研究意义。第二章介绍与本论文相关的技术和现有成果，这部分主要包括 RISC-V 软核的分析，Rust 语言的优势及原因、BBL 简介和 rCore 操作系统简单说明。第三章中，介绍本实验的平台搭建和工具链安装，这为后文的实验奠定了基础。第四章主要介绍实验中的 SOC 硬件平台的设计与搭建过程以及相关测试。第五章主

要介绍软件部分包括解压缩模块、BBL 和 rCore 操作系统，展示了整个软件系统的搭建流程。第六章则展示整个实验的结果，并对未来的工作给出展望。

第 2 章 研究现状与技术分析

2.1 RISC-V 架构

RISC-V 是一个通用的指令集架构，它能适应包括从最袖珍的嵌入式控制器，到最快的高性能计算机等各种规模的处理器，现如今已经得到了广泛应用，该架构具有如下特点^[4]：

1. 开源

RISC-V 是一个开源的指令集架构，这与几乎所有的旧架构都不相同，开源保证了这个架构的稳定性，即不会受到某个公司的影响。RISC-V 基金会的目标就是维护 RISC-V 的稳定性，就像 Linux 之于操作系统一样。

2. 模块化指令集

此外，传统的计算机体系结构的方法是增量 ISA，这样新处理器不仅必须实现新的 ISA 拓展，还必须实现过去的所有拓展，这是为了保证向后的二进制兼容性。但这样会给新体系的设计增加不必要的负担。

而 RISC-V 则采用的是模块化设计，它的核心是一个名为 RV32I 的基础 ISA，运行一个完整的软件栈。RV32I 是固定的，永远不会改变。这为编译器编写者，操作系统开发人员和汇编语言程序员提供了稳定的目标。模块化来源于可选的标准扩展，根据应用程序的需要，硬件可以包含或不包含这些扩展。这种模块化特性使得 RISC-V 具有了袖珍化、低能耗的特点，而这对于嵌入式应用可能至关重要。RISC-V 编译器得知当前硬件包含哪些扩展后，便可以生成当前硬件条件下的最佳代码。惯例是把代表扩展的字母附加到指令集名称之后作为指示。例如，RV32IMFD 将乘法(RV32M)，单精度浮点(RV32F)和双精度浮点(RV32D)的扩展添加到了基础指令集(RV32I)中。

3. 简洁

在 x86, arm 等体系结构中，都会有较为复杂的指令，在一条指令中执行多次加载，和寄存器读写，并且同时还执行分支语句，而在 RISC-V 中则没有这样的指令。因为复杂指令给硬件设计者带来了不小的困难。此外简单的处理器对嵌入式应用程序也是有益的，因为它更容易预测执行的时间。

4. 程序大小

程序越小，存储它所需的芯片面积就越小。更小的程序还能减少指令缓存的未命中问题，从而节省了功耗，也提高了性能。短的代码长度是 ISA 架构师的目标之一。x86-32 ISA 的指令可以短至 1 字节，也可以长达 15 字节。虽然逻辑上 x86 的变字节长度的指令要更短一些，但是实验表明，在 32 位架构下，x86 的程序大小最大，而 RISC-V 相比 x86 而言会减少 9% 的空间大小。

2.2 RISC-V 软核

2.2.1 picorv32

在[5]中，使用的是以 picorv32 为基础的第三方 CPU 进行开发，picorv32 是经过精心设计和裁剪的小型 CPU，它可以支持 IMC 三种模块的指令集。

具体来说，picorv32 有如下特点：1. 逻辑资源占用少，一般而言在 Xilinx7 系列的架构中，只需要 750-2000 个 LUT 就可以满足设计。2. CPU 主频高，在 Xilinx7 系列上，最高可以运行到 250MHz – 450MHz。3. 支持不同总线，picorv32 可以支持的总线协议包括 SRAM，WISHBONE 和 AXI4，方便开发者灵活选择。4. 提供中断控制，这里 picorv32 通过自定义的指令来实现。5. 对于协处理器，picorv32 也有相应的支持。

在[5]中，朱威浦基于 picorv32 进行了实验，虽然取得了一定进展，但是由于 picorv32 处理器使用的是自定义指令来支持特权级架构，因而在运行操作系统，移植 bootloader 的过程中遇到了比较大的挑战。此外由于该软核功能简单，不支持 TLB 功能，所以操作系统上关于虚拟内存的相关实验也无法得以开展。

2.2.2 openriscv^[9]

该软核 openriscv 相比于 picorv32 来说虽然没有广泛的使用，但是它的特权级是按照手册实现的，并且支持大部分的 M 态、S 态、U 态下的特权级寄存器。这样它的健壮性更好，在移植的过程中，就不需要考虑自定义指令这些问题。

另外该软核支持 TLB，对于指令和数据分别有 16 项 TLB，这样从硬件层次上就保证了操作系统可以拥有更强大的功能，如虚拟内存的支持等等。

该软核封装为 Wishbone 总线，所以开发过程中只需要添加相应 Wishbone 总线

的外设就可以。

不过此软核也存在如下缺点，首先是相比 `picorv32` 来讲，它占的资源更大，多达上万个逻辑资源，并且时钟频率最大只能有 30MHz；其次更为棘手的是，`openriscv` 并不是一个很成熟的 CPU，在 2017 年发布后就停止更新了，所以需要对照新的指令规范进行重新修改，才能正常运行。

考虑到论文的目标对于性能的要求不是很高，更多的是功能的验证，所以这里采用 `openriscv` 软核作为 CPU 进行开发。

2.3 总线协议

总线是计算机各种功能部件之间传送信息的公共通信干线，它是由导线组成的传输线束，按照计算机所传输的信息种类，计算机的总线可以划分为数据总线、地址总线和控制总线，分别用来传输数据、数据地址和控制信号。下面对本工程中所采用的两种总线进行介绍。

2.3.1 Wishbone 总线简介

Wishbone 总线(32 位)在主机与从机之间的交互信号如下表所示：

表 4-3 WISHBONE 总线信号说明

信号名称	位宽	作用
CLK	1	时钟输入
ADDR_O	32	地址线
DATA_O	32	数据线(输出)
DATA_I	32	数据线(输入)
WE_O	1	写使能
SEL_O	4	选通
STB_O	1	使能
ACK_I	1	确认

CYC_O	1	使能
-------	---	----

WISHBONE 协议最关键的信号是 CYC、STB 与 ACK，CYC 和 STB 同时拉高时表示请求开始，在整个过程中，保持高电平，一直等到 slave 响应 ACK 拉高后的下一周期，CYC，STB 和 ACK 拉低，至此一个请求结束。

2.3.2 Avalon 总线介绍

Avalon 总线(16 位)在主机与从机之间的交互信号如下表所示：

表 4-4 AVALON 总线信号说明

信号名称	位宽	作用
CLK	1	时钟输入
ADDRESS	22	地址线
READDATA	16	数据线(输入)
WRITEDATA	16	数据线(输出)
BYTEENABLE	4	使能
CHIPSELECT	1	片选
WRITE	1	写使能
READ	1	读使能
WAITREQUEST	1	等待
READVALID	1	读确认

对于 Avalon 协议而言,关键的几个信号是 READ、WRITE、WAIT 和 READVALID。当 READ/WRITE 拉高代表读或写的请求，但是与 Wishbone 不同的是，这个请求一只保持到 WAIT 变低，在 WAIT 为高时，从机处理请求，对于写请求来说，只需等待 WAIT 变为低电平就可以,而对于读请求来说还需要等待 READVALID 变为高电平，才表明总线交互结束。

2.4 Rust 语言

Rust 是由 Mozilla 主导开发的通用、编译型编程语言。它具有“安全，并发，使用”的特点，支持函数式，并发式，过程式以及面向对象的编程风格。Rust 的语法与 C++相似，但它在维持高性能的情况下能保证内存安全。

Rust 是编译型强类型语言，不是 JavaScript、Python 这种动态类型的语言，也不是 Java、c#这种需要虚拟机的语言，也不是像 Java、Golang 这样需要 GC 的语言。Rust 的运行效率与 C++相同甚至更好。

在过去，如果要写高性能，高实时性，或者接近底层的大型程序，几乎只能选择 C、C++。出于开发效率的考量，应用软件很少用汇编开发，使用纯 C 的也很少，而操作系统等底层软件更多的使用 C 语言，并伴随少量的汇编。C++从 1983 年诞生以来，从 C with classes，到 C with template，再到 C++11，编程范式几经改动，有许多需要向后兼容的负担，而 Rust 没有历史包袱，所以从整体上看，其设计上比 c++更精炼。

2.3.1 包管理器

Rust 在开发阶段砍掉标准库 std 中的许多重型功能，并建设了 Rust 包管理软件 cargo，包托管网站 crates.io，包文档的托管网站 docs.rs，其中的许多包的质量和 std 一样高，使用也很方便。Rust 语言的标准库种类很多如 hashmap、统一 api 的多线程、网络功能等。Rust 还允许程序不使用 std 库，只使用平台无关的 core 库，以进行嵌入式和操作系统开发，也允许手动选择内存分配器（molloc）。

2.3.2 支持多种语言调用

Rust 设计上无 GC、VM，这让其很容易调用 C 代码，但 Rust 更进一步，实现了 Rust 与 C、C++的高效双向调用。许多语言只能自己调用 C 库，并且有效率上的问题，但 Rust 可以和 C 无缝交流，既可以在 Rust 项目中使用 C 库，也能在 C 项目中使用 Rust 库。此外 Rust 也能和 python, JavaScript, Ruby 进行模块调用。

2.3.4 函数式编程

Rust 广泛的使用了枚举类型，Rust 的枚举与 C 不同，更像是来自函数式编程语

言的代数数据类型。最常见的两个枚举就是 `Option` 和 `Result`:

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

`Option` 解决了其他程序语言中由 `NULL` 引发的内存错误, 在 `Rust` 中, 使用 `Option` 的方法提取到 `Some` 中的数据 `T`, 并根据需要在遇到 `None` 时崩溃 `unwrap()`、返回默认值 `unwrap_or(T)`

`Result` 则代表了另外一种常见情况: 异常处理。`Rust` 不使用 `try`, `catch` 这种处理方式, 而是选择使用返回 `Result` 枚举, 如果返回的是 `Ok`, 那么就 `Ok`, 如果是 `Error`, 那就需要处理错误, 或者将错误再次返回——效果和 `try` 相同。

```
enum Result<T> {  
    Ok(T),  
    Err(Error),  
}
```

2.3.5 强大的宏系统

`Rust` 具有强大的宏系统。`C` 语言的宏是基于文本替换的, 很容易出现替换问题, 而 `Rust` 的宏是基于语法单元进行展开的。宏可以自动生成代码, 从而达到可以实现接受任意多个参数的函数, 如下面寻找最小值的这个函数所体现:

```
macro_rules! find_min {  
    ($x:expr) => ($x);  
    ($x:expr, $($y:expr),+) => (  
        std::cmp::min($x, find_min!($($y),+))  
    )  
}
```

2.3.6 生命周期

`Rustc` 禁止在 `unsafe` 外使用裸指针, 也禁止产生空指针。`Rust` 使用一套简单的概念, 让 `Rustc` 编译器维护了类型的生命周期的问题。首先, `Rust` 中的变量只有标识符

的作用，不包含值，只有将资源绑定到标识符上变量才可以直接使用。例如 `let x:i32=100`,就是将 100 这个 i32 的值绑定到 x 这个标识符上,如果我们只声明 `let x:i32`,再试着打印 x, 编译器就会报错, 而许多语言会使用默认值甚至是内存里的随机值。Rust 的作用域为大括号级别的, 如果局部变量离开作用域, 会导致其绑定的资源销毁 (drop), 这是编译器自动插入并执行的, 类似 C++ 的析构, 但我们不需要手动编写 drop 函数, 这是通过所有权系统实现的。一个资源, 无论是栈上的还是堆中的, 都只能绑定到一个变量, 即这个变量拥有了这个资源的所有权。只有拥有资源所有权的变量离开作用域了, 资源才会被销毁。一个资源对应一个变量, Rustc 在编译时就可以确定所有资源的生命周期, 即从其创建到最后一个拥有它的变量离开作用域, 这套系统也被称为 RAII (Resource Acquisition Is Initialization, 资源获取就是初始化), 如果我们在生命周期外使用资源就会在编译时报错, 而 C++ 则是运行时[6]。

这种生命周期的使用将一些隐藏的错误前推至编译阶段, 相比于运行时发现错误, 大大减少了纠错调试的时间。

2.3.7 并发的安全性

C 语言/C++ 发明时 CPU 还都是单核心, 而现在 CPU 核心数越来越多, 再加上超线程技术, 应用程序势必要走向多线程, 然而竞争的存在使得使用 C++ 编写程序很容易出现 bug。Rust 通过 Send+Sync 这两个 trait 处理多线程问题。这两个 trait 没有定义任何方法, Send 表示为如果 `T: Send`, 将 T move 到另一个线程, 能保证所有权能安全的转移到新的线程并与原线程解耦, 不会使原线程 use after free 或可以同时访问同一块内存等内存安全问题。Sync 表示为如果 `T: Sync`, 将 &T send 到另一个线程, 不会出现竞争问题。

2.5 BBL

一个操作系统的运行之前是 bootloader 的工作, 通过这段程序, 系统就可以完成硬件的初始化, 从而将系统的软硬件配合起来这样进入操作系统后就可以正常地执行, 通常 bootloader 是一个严重依赖于硬件的环境, 特别是对于小型的嵌入式系统来说, 就更需要根据硬件不同进行定制, 从而适配整个系统。

具体到本工程而言, 所采用的是由加州伯克利大学开发的 RISC-V bootloader,

这个 `bootloader` 小巧，易于理解，方便用户修改以适配各自的硬件平台。在 `BBL` 中，完成的工作包括清空寄存器，设置 `CSR` 特殊寄存器，将一些 `S` 态中断异常重定向到 `M` 态，另外 `BBL` 还完成了内存信息的获取初始化，中断初始化，页表初始化以及最后的加载 `OS` 的过程。

实际上，`bbl` 虽然是一个比较完善的 `bootloader`，但是具体到本工程而言，还需要进行进一步地设置，而不是直接拿来就可以用的，尤其是本工程中需要部分硬件实现交由软件模拟，所以对于 `bbl` 和硬件交互密切的工作要进行一定修改。

2.6 rCore 操作系统

近些年，有很多非常好的使用 `Rust` 语言搭建操作系统的尝试，下面将对本工程中采用的 `rCore` 进行介绍。

`rCore` 是清华大学教学操作系统 `ucore` 的 `Rust` 移植版本，使用现代编程语言，以提升开发操作系统过程中的体验和质量，并充分利用 `Rust` 语言，减少开发过程中调试寻找 `bug` 的时间，并进一步探索未来操作系统的设计实现方式。

目前 `rCore` 具有中断支持，内存管理，虚拟内存，线程切换，用户进程，文件系统等功能，支持多种目标指令架构，对于 `RISC-V32` 来说也是支持的，而且能够在 `qemu` 模拟器环境下运行，这对开发调试是非常有帮助的。

鉴于 `rCore` 功能完善且整体系统不是十分复杂，所以选择移植 `rCore` 到本 `CPU` 上，从而完成相关实验。

2.7 本章小结

本章首先对两种不同的 `RISC-V` 软核进行对比和分析，从功能完备的角度上最终选择 `openrisce` 作为开发的第三方 `CPU` 软核。然后介绍了 `Rust` 系统编程语言的优势及其安全特点。最后介绍了 `BBL` 和 `rCore` 操作系统的基本开发情况。

第 3 章 环境搭建与工具介绍

3.1 FPGA 硬件平台

3.1.1 STEP-CYC10 FPGA 开发板

本工程中使用的 FPGA 开发平台是由思得普信息科技有限公司生产的小脚丫系列开发板，这类开发板具有小体积、高性价比的优势。在本实验中，主要基于 CYC10 系列 FPGA 开发板，小脚丫 STEP-CYC10 是一款基于 Intel Cyclone10 设计的 FPGA 开发板，芯片型号是 10CL016YU256C8G。另外，板卡上集成了 USB Blaster 编程器、SDRAM、FLASH 等多种外设。板上预留了 PCIE 子卡插座，可方便进行扩展具体资源如下表所示^[7]：

表 3-1 STEP-CYC10 开发板资源

资源种类	数量	资源种类	数量
LE 资源	16000 个	PCIE 接口	1 个
片上 ROM	505Kb	USB 擦写器	1 个
DSP	56 个	SDRAM	8MB
PLL	4 路	FLASH	8MB
数码管	4 个	三轴加速器	1 个
三色 LED	2 个	时钟源	2 个
5 向按键	1 个	LED	8 路

熟悉 STEP-CYC10 FPGA 开发板硬件原理图也是开发流程中不可缺少的一环，其中比较关键的部分如下图所示^[7]：

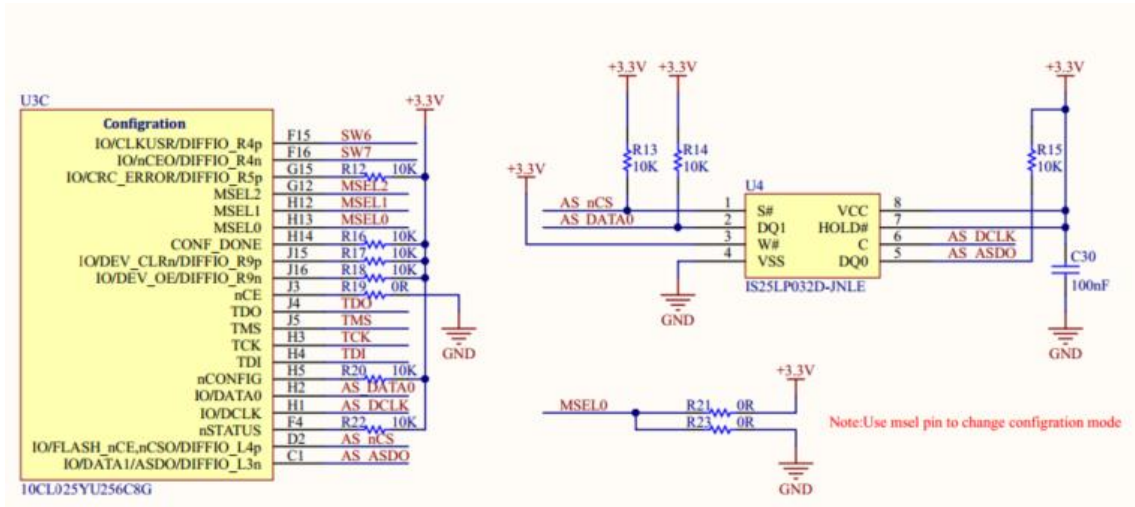


图 3-1 FLASH 相关的硬件原理图

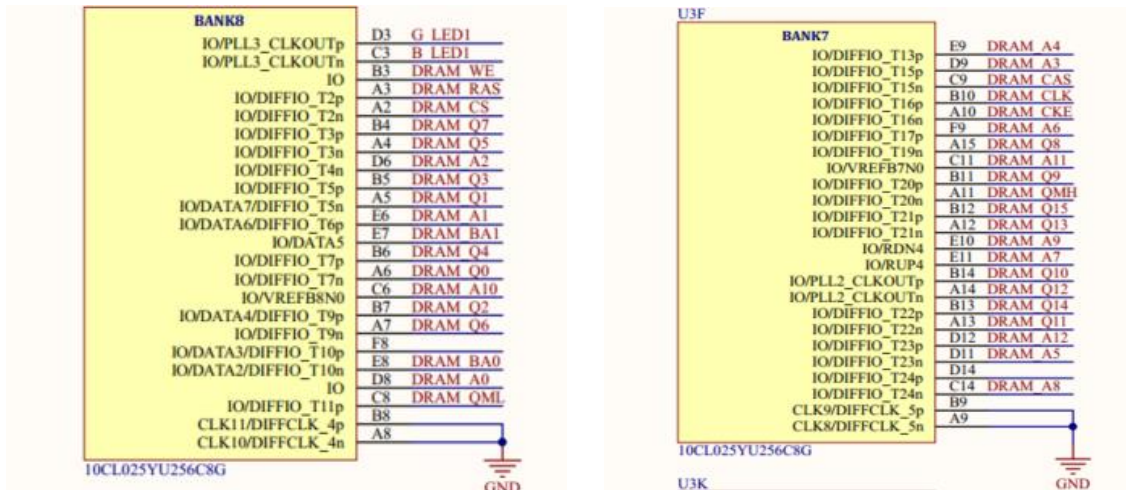


图 3-2 SDRAM 相关的硬件原理图

最终实物的效果如下图 3-3 所示^[7]:

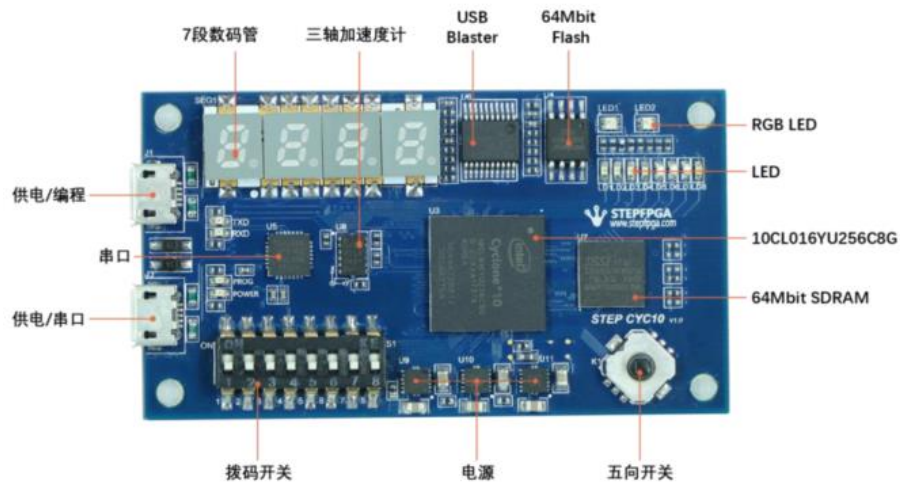


图 3-3 STEP-CYC10 实物图

3.2 Verilog 综合与仿真工具

3.2.1 Quartus

Quartus 是一款可编程逻辑的设计环境，在这个软件中用户可以通过编写 Verilog 语言进行综合最终将 bit 文件下载到开发板中实现相应的功能。

Quartus 中还有一个非常有用的硬件调试工具叫做 Logical Analyzer，它与 ISE 中的 Chipscope 十分相似，都是利用额外的资源来实时观察信号，相当于用硬件模拟了多路示波器，利用这个工具就可以实时地抓取信号，对调试有很大的帮助。

3.2.2 Modelsim

在此过程中，可能还需要用到 Modelsim 对 CPU 的功能进行仿真，这样可以确保下板的过程更加顺利。Modelsim 是一款优秀的 HDL 语言仿真软件，它能提供友好的仿真环境，是业界唯一的单内核支持 VHDL 和 Verilog 混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk 技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是 FPGA/ASIC 设计的首选仿真软件。

3.3 RISC-V 交叉编译链

3.3.1 交叉编译

在开发嵌入式系统时，通常都采用交叉开发（Cross Developing）的模式，即：开发系统是建立在硬件资源丰富的 PC 机或工作站上，通常称其为宿主机（host），应用程序的编辑，编译，链接等过程都是在 Host 上完成的；而应用程序的最终运行平台却是和 Host 有很大差别的嵌入式设备，通常称其为目标机（Target）；调试在二者之间联机交互进行^[8]。这里二者的差别主要是指：

- 1) 硬件上差别很大。通常 CPU 的类型都不同。例如本工程中 Host 的 CPU 为 Intel i5-7200U 芯片，而 Target 为 RISC-V 芯片；

- 2) 软件环境的差异。在 Host 上有通用 OS 的支持如 Windows 或 Linux 等，而

Target 一般为裸机，没有任何的软件资源，或者运行专用的嵌入式操作系统。通常情况下，现有的目标机 OS 是用于对嵌入式应用的支撑而不是用于目标机的开发环境平台。

因此，基于上述交叉开发模式的开发系统被称为嵌入式交叉开发系统，它一般主要包括：

- 1) 交叉编译工具：在宿主机上，能够将一个源程序编译生成一个可执行程序软件，这个可执行程序能够在目标机上执行。
- 2) 交叉调试工具：在宿主机上能对目标机上的可执行程序进行源码或汇编级调试的软件

3.3.2 RISC-V 交叉编译链说明

在本工程用需要用到的具体的编译链就是 RISC-V 的交叉编译链，因为 Host 是 x86 平台的，所以需要在 Host 上生成可以编译 RISC-V 的编译链，具体用到的工具如下表所示：

表 3-2 gnu 工具链

名称	功能
as	GNU 汇编器，将汇编源代码，编译为机器代码
ld	GNU 链接器，将多个目标文件，链接为可执行文件
ar	用于创建、修改库
addr2line	将 ELF 特定指令地址定位至源文件行号
objcopy	用于从 ELF 文件中提取或翻译特定内容，可用于不同格式二进制文件的转换。
objdump	用于查看 ELF 文件信息，反汇编可执行程序为汇编文件
readelf	显示 ELF 文件的信息
strip	用于将可执行文件中的部分信息去除，如 debug 信息，可在不影响程序正常运行的前提下减小可执行文件的大小以节约空间
riscv-gdb	用于调试 RISC-V 程序的调试工具

riscv-glibc	Linux 系统下的 RISC-V 架构 GNU C 函数库
riscv-newlib	面向嵌入式系统的 C 语言库
riscv-qemu	RISC-V 架构的机器仿真器和虚拟化器, 可在其模拟的硬件机器上运行操作系统
riscv-isa-sim	spike 模拟器是 RISC-V ISA 的仿真器, 功能类似于 qemu
riscv-pk	运行于本地机器上的代理内核, 是一个轻量级的应用程序执行环境, 可直接运行于 spike 模拟器之中, 然后加载运行静态链接的 RISC-V ELF 程序
bbl	Berkeley Boot Loader, 此引导程序可用于在真实环境下加载操作系统内核
riscv-test	RISC-V 官方的测试程序

3.3.3 RISC-V 交叉编译链安装

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-
dev libgmp-dev gawk build-
essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
$ cd riscv-gnu-toolchain
$ ./configure --prefix=/opt/riscv --with-arch=rv32ima --with-abi=ilp32
$ make -j4
```

在默认情况下, 如果不使用 `configure` 进行配置, 则所有的文件或被放置在 `/usr/local/bin` 目录下, 为了方便管理可以选择参数 `—prefix` 指定路径。

由于工程中对应的 RISC-V CPU 是一个 32 位的支持 IMA 三个模块的, 所以在配置 `—with-arch` 过程中需要额外注意, 因为 RISC-V 有很多其他模块比如压缩模块 C, 配置这个模块虽然会减少储存代码所需的容量, 但对大大增大系统设计的复杂度, 所以此时并不对 C 进行配置。

参数 `—with-abi` 的目的主要是依据硬件平台是否支持浮点单元进行相应的配置, 因为本工程中的 CPU 不支持硬件浮点, 所以在选择这个参数的过程中需要配置成软浮点模式即 `ilp32`, 如果 CPU 有 FPU 的话, 则设置成 `ilp32d` 就可以。

设置 `configure` 还有很多其他的配置，在 `riscv-gnu-toolchain` 目录下运行

```
./configure -h
```

通过这条语句，可以看到更多关于配置编译链的说明帮助。

在进行完上述步骤后还需要进行环境变量的添加，在 `~/.bashrc` 中添加 `PATH=/opt/riscv/bin:$PATH` 之后键入

```
$ source ~/.bashrc
```

至此编译链安装完毕。

3.3.4 编译链测试

接下来在任何一个地方命令行输入 `riscv32` 后输入两次 `Tab`，应该会有自动补全成 `riscv32-unknown-elf` 并显示若干编译链工具。

在安装好 `Qemu` 的前提下，生成 `qemu-riscv32`，然后编写一个简单的测试程序 `hello.c`

```
#include<stdio.h>

int main() {
    printf("helloworld!\n");
    return 0;
}
```

然后执行如下命令：

```
$ riscv32-unknown-elf-gcc -o hello hello.c
$ qemu-riscv32 hello
```

最终可以看到屏幕上打印出 `helloworld` 的字样，至此证明编译链安装成功。

3.4 Rust 编译链

3.4.1 Rust 编译链安装

有了上一节中的 `riscv32` 下的 `gcc` 还是不够的，针对于 `Rust` 语言还需要有额外的编译器，这就是 `rustc`，通过如下安装命令：

```
$ curl -sSf https://static.rust-lang.org/rustup.sh | sh
$ source $HOME/.cargo/env
```

```
$ rustup update
```

此时键入 `rustc -version` 会得到类似于 `rustc x.y.z (abcabcabc yyyy-mm-dd)` 的信息，这时自带的包管理器 `cargo` 也会安装完毕。

3.4.2 Rust 编译链测试

对 Rust 工具链进行测试，使用 `cargo` 构建项目，通过如下命令构建 `helloworld` 可执行文件项目。

```
$ Cargo new helloworld
```

其创建的目录结构如下：

```
$ cd hello_world
$ tree
├── Cargo.toml
└── src
    └── main.rs
1 directory, 2 files
```

`Cargo.toml` 文件为整个项目的信息，包括名称、使用的外部库依赖等信息。

```
[package]
name = "hello_world"
version = "0.0.1"
authors = ["Your Name <you@example.com>"]
```

`src/main.rs` 文件为项目主函数，项目将从此处开始运行，默认创建项目的 `main` 函数内容如下：

```
fn main() {
    println!("Hello, world!");
}
```

通过如下命令可编译运行项目。

```
$ cargo run
Compiling hello_world v0.0.1
Running `target/debug/hello_world`
Hello, world!
```


3.5 本章小结

在本章中，首先介绍了 FPGA 硬件开发平台的参数，展示了原理图，对开发板资源做了相关介绍。在此之后，对开发过程中用到的硬件语言集成开发工具 Quartus 以及仿真软件 Modelsim 进行了介绍。最后针对 RISC-V 交叉编译链和 Rust 编译链进行了如何安装和测试的说明，这为后续实验奠定了基础。

第 4 章 基于 RISC-V 的 SOC 搭建

4.1 SoC 模块组成与设计

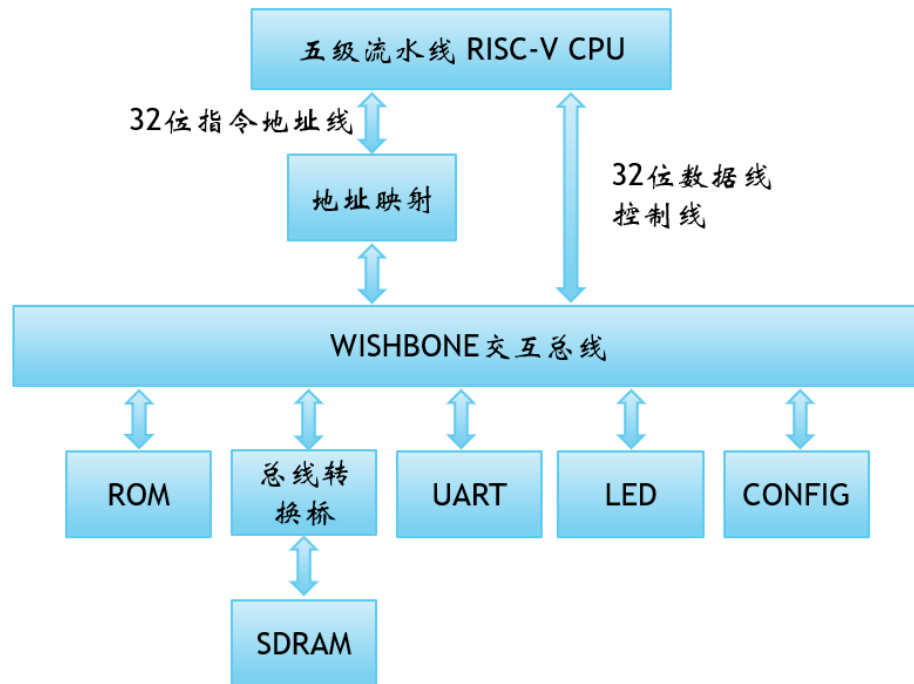


图 4-1 SOC 框架示意图

上图展示了整个片上系统的框架，首先是最基础的 RISC-V CPU，对应于本工程中的就是 openriscv，该 CPU 通过地址转换后封装为 Wishbone 总线形式，然后与其他外设交互访问。

一些必须的外设包括 ROM、SDRAM、UART，LED 等，在工程中将他们封装为 Wishbone 总线的形式后就可以接入到整个系统中。

这里有一个比较特别的模块叫 CONFIG，CONFIG 模块对应于 verilog 中 config_string 模块，这个模块存在的目的是为了兼容 BBL，其中保存了一些硬件信息供 BBL 查询设置，另外根据 BBL 要求，timer 与 cmp 的地址也是通过内存地址访问的，这里也一并归于此模块中。当 timer 达到 cmp 的数值时会触发一个定时器中断，直接送往 CPU。

4.1.1 封装 openriscv

本工程中采用 openriscv 原生支持 Wishbone 总线，所以无须进行大的修改，但是对于地址映射而言，每个硬件平台都是不同的，这里的设置如下表所示：

表 4-1 SOC 硬件平台地址映射

外设	地址分布	大小
ROM	0x0001_0000~0x0001_c000	48KB
SDRAM	0x8000_0000~0x8080_0000	8 MB
UART	0x0200_0000~0x0200_0020	32 B
LED	0x3000_0000~0x0300_0010	16 B
CONFIG	0x0000_1000~0x0000_0100	256 B
	0x4000_0000~0x4000_0010	16 B
	0x4000_0000~0x4000_0004	4B

该部分可以在 ./wishbone_cycl0/phy_bus_addr_conv.v 中找到对应的 Verilog 语句及宏定义，只需修改其中的数值即可。举个例子，如想修改 RAM 的地址分配，只需要修改以下两个宏即可分别代表长度和起始地址，其余不需更改。

```
`define RAM_PHYSICAL_ADDR_BEGIN          34'h08000_0000
`define RAM_PHYSICAL_ADDR_LEN            34'h00080_0000
```

4.1.2 ROM 控制器

一个需要注意的事项是，这里的 ROM 里面调用了已经封装好的 IP 核，这个 IP 核的配置为 深度=16384，宽度=32bits，这样算起来一共有 64KB，与之前的 48KB 不符。这是因为 IP 核深度只能配置为 16384/8192，即 64KB/32KB，没有中间选项，所以只好如此，但并不影响结果，只要保证真正用到的 rom 不超过 48KB 即可。或者配置为 8192 也可以，这样程序限制在小于 32KB。

有了这个 IP 核之后，我们还需要把它封装成 Wishbone 总线，这里 ROM 的例化方式如下：

```
ip_rom ip_rom0 (
```

```
.address      (wishbone_addr_i[15:2]),
.clock        (clk),
.q            (wishbone_data_o)
);
```

这个 rom 是一个很简单的逻辑，给定输入地址，在下一个周期返回输出数据，所以这里所需的工作就是延时一个周期发送 CYC 和 STB 确认信号就可以，之后对于 RAM 来说也是如此的。

4.1.3 SDRAM

SDRAM 的芯片型号为 IS45S16400J-7BLA2，该型号 SDRAM 芯片可用时钟频率为 200、166、143、133MHz，本文选取工作时钟频率为 200MHz，事实上在 150MHz 条件下也可运行。存储空间大小为 64Mb，数据总线宽度为 16 位，该 SDRAM 由 4 个 bank 组成，芯片总容量的计算方式为：

$$1\text{Mbit} * 16\text{bit} * 4\text{bank} = 67108864\text{bits} = 64\text{Mbit},$$

每个 bank 的组成：

$$4096\text{ rows} * 256\text{ columns} * 16\text{ bits},$$

即以 12 行 8 列的结构排列。

控制 SDRAM 正常工作的几个重要参数分别为：tRCD、CL、tWR、tRP。tRCD 代表发送读写命令与行有效命令的时间间隔，广义的 tRCD 以时钟周期为单位，如 tRCD=2，代表延迟周期为两个时钟周期，具体到确切时间，则需根据具体时钟频率而定。CL 代表从 CAS 与读取命令发出到第一个数据输出的时间，CL 单位与 tRCD 一样为时钟周期，具体耗时由时钟频率决定。tWR 是为了保证数据的可靠写入，留下的足够的写入以及校正时间。最后，tRP 表示在发出预充电命令到允许发送 RAS 行有效命令打开新的工作行的间隔时间，在 200MHz 的工作频率下，根据芯片手册，几个重要参数的数值分别为：

表 4-2 SDRAM 参数

参数	数值
CL(CAS)	3
t_rcd	20ns

t _{ac}	5.4ns
t _{wr}	14ns

若使用 SDRAM，需使用 SDRAM 控制器作为中介连接 CPU 与外设 SDRAM，本文使用 Quartus 中 SDRAM controller IP 核作为硬件系统的 SDRAM 控制器，上述参数和其余参数均在 SDRAM 控制器中设定。如下图所示：

Memory Profile		Timing
CAS latency cycles::	<input type="radio"/> 1 <input type="radio"/> 2 <input checked="" type="radio"/> 3	
Initialization refresh cycles:	2	
Issue one refresh command every:	15.625	us
Delay after powerup, before initialization:	100.0	us
Duration of refresh command (t _{rfc}):	70.0	ns
Duration of precharge command (t _{rp}):	20.0	ns
ACTIVE to READ or WRITE delay (t _{rcd}):	20.0	ns
Access time (t _{ac}):	5.4	ns
Write recovery time (t _{wr} , no auto precharge):	14.0	ns

图 4-2 qsys 中实际 SDRAM 参数设置

CPU 的总线协议是 Wishbone，而 SDRAM IP 核的总线协议是 Avalon，所以需要在两者之间进行转换。下一节介绍转换桥

4.1.4 总线转换桥

总线转换桥的功能是将 CPU 的 Wishbone 总线转换为何 SDRAM 控制器适配的 Avalon 总线，其二是由于 SDRAM 控制器的位宽是 16 位的，所以还需要进行 32 位到 16 位的位宽转换，其三是 CPU 主频和 SDRAM 控制器主频不匹配，需要进行相应控制，从而使得整体上功能没有问题。

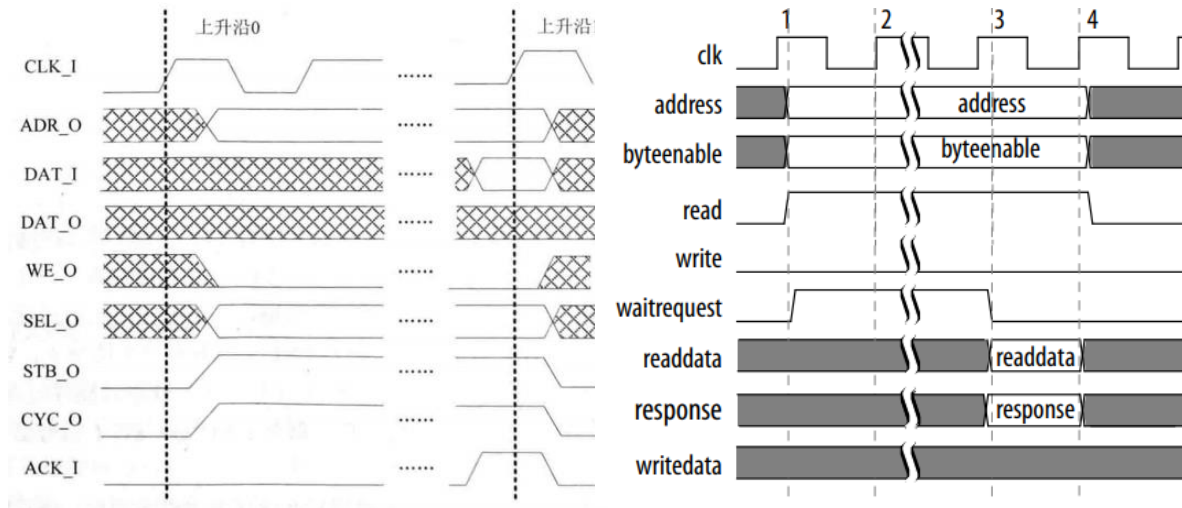


图 4-3 两种总线协议握手时序图

这里采用有限状态机的思路设计转换桥的过程，整体思想就是根据握手信号的变化进入到不同的状态，然后在这个状态中根据另外一些信号再改变到新的状态，循环往复。

总得说来需要这样几个状态：

(1)IDLE: 这个状态代表一开始初始化，所有关于握手信号都要置低，以免进行不必要的请求

(2)写请求: 这个状态是当 Wishbone 总线收到 CPU 发送来的写使能信号时进入的状态，此时转换桥发送给 Avalon 一个信号。

(3)写等待: 这个状态是上一个状态马上进入到这个状态，表明转换桥在等待从机的确认信号

(4)完成: 这个状态表明请求已经被从机响应，整个请求已经完成了。

(5)、(6)读请求与读等待同理

另外在实现中，由于需要从 32 位转换到 16 位，所以每个读写状态又分为高位低位，所以一共有 10 个状态，初始 1 个，完成 1 个，写和读各占 4 个。

最终的状态转换图如下：

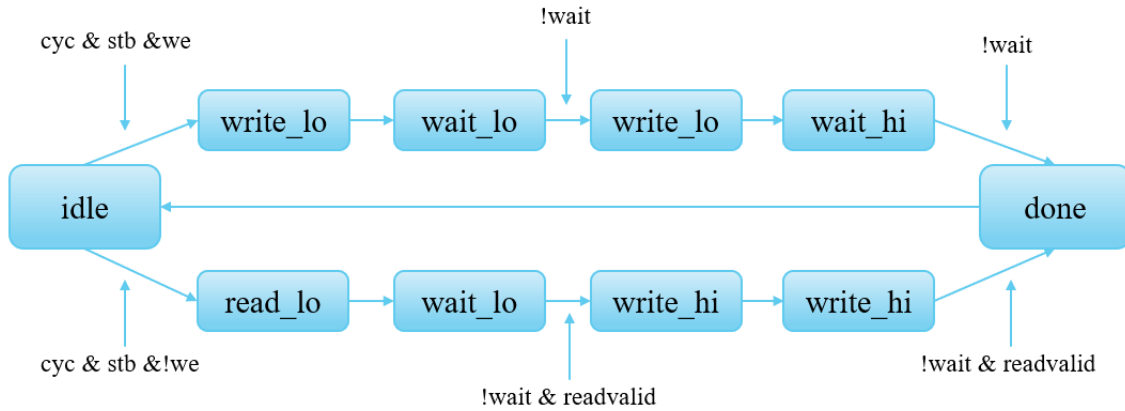


图 4-4 总线转化桥的状态转换图

4.1.5 UART 控制器

4.1.5.1 UART 简介

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter，通常称为 UART）是一种异步收发传输器，是电脑硬件的一部分，将数据透过串列通讯和平行通讯间作传输转换。UART 通常用在与其他通讯接口（如 EIA RS-232）的连接上。

串口在嵌入式系统当中是一类重要的数据通信接口，其本质功能是作为 CPU 和串行设备间的编码转换器。当数据从 CPU 经过串行端口发送出去时，字节数据转换为串行的位；在接收数据时，串行的位被转换为字节数据。应用程序要使用串口进行通信，必须在使用之前向操作系统提出资源申请要求（打开串口），通信完成后必须释放资源（关闭串口）。典型地，串口用于 ASCII 码字符的传输。通信使用 3 根线完成：（1）地线，（2）发送数据线，（3）接收数据线。串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。对于两个进行通行的端口，这些参数必须匹配：波特率是一个衡量通信速度的参数，它表示每秒钟传送的 bit 的个数；数据位是衡量通信中实际数据位的参数，当计算机发送一个信息包，标准的值是 5, 7 和 8 位。如何设置取决于你的需求；停止位用于表示单个包的最后一位，典型的值为 1, 1.5 和 2 位，停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会；奇偶校验位是串口通信中一种简单的检错方式，有四种检错方式——偶、奇、高和低，也可以没有校验位^[10]。

总的来说串口具有以下几个重要参数:

(1) 波特率: 串口异步通讯中由于没有时钟信号, 所以通讯双方需要约定好波特率, 即每个码元的长度, 以便对信号进行解码。常见的波特率有 4800、9600、115200 等。

(2) 起始位: 表明数据开始, 起始信号用逻辑 0 表示。

(3) 停止位: 停止信号由 0.5、1、1.5 或 2 个逻辑 1 的数据位表示, 需要收发双方提前约定好。

(4) 数据: 传输的数据内容, 其长度一般被约定为 5、6、7 或 8 位长。

(5) 校验位: 由于在通讯过程中易受到外部干扰导致传输数据出现偏差, 所以在有效数据之后加上校验位解决。校验方法有奇校验 (odd)、偶校验 (even)、0 校验 (space)、1 校验 (mark) 及无校验 (noparity)。奇校验要求有效数据和校验位中“1”的个数为奇数, 偶校验刚好相反, 要求有效数据和校验位的“1”数量为偶数。

4.1.5.2 UART 控制器设计

在前人的工作中, 已经有了关于 UART 控制器的实现, 不过这种实现是基于第三方 IP 核实现的、由于第三方 IP 核是一个兼容 UART16550 的协议, 所以它拥有较为复杂的逻辑, 如果直接使用, 在本工程中就显得有些过于臃肿了。因而在本工程中重写了关于 UART 的控制, 精简了相关逻辑。

在本工程中, 将 UART 控制器的控制设置为一个 FSM, 并有如下几个状态:

(1) idle

idle 表明现在串口控制器出于空闲的时候, 也就是此时 CPU 并没有给控制器发送请求。

(2) req

在 req 阶段, uart 控制器收到了来自 Wishbone 总线的请求, 根据设定好的波特率开始收发数据。具体实现是每次先用寄存器把收到的需要发送的字符缓存起来, 然后每隔一个波特率时间就将这个寄存器的最后一个 bit 发送出去:

```
if(send_bitcnt == 0) begin
    state <= StateOk;
end else if(send_divcnt >= CfgDivder) begin
```



```
send_pattern <= {1'b1, send_pattern[9:1]};  
send_bitcnt <= send_bitcnt - 1;  
send_divcnt <= 0;  
end else begin  
send_divcnt = send_divcnt + 1;  
end
```

（3）Ok

Ok 阶段是代表这个请求结束了，此时会发送 ACK 相应，然后自动转到 req。

4.1.6 CONFIG

CONFIG 实际上是配合软件 BBL 设置的一个存储硬件信息的逻辑，具体说来，它是通过将指令硬编码成实现的来实现。而这些指令的有整个平台的硬件信息比如 ram 地址大小，uart 地址等等，在第 5 章介绍 BBL 的时候还会再详细讨论。

4.2 SOC 硬件平台功能验证

搭建起这个 SOC 平台之后，就是对其进行验证。具体来验证又分两步走，第一步是在仿真环境下用 Modelsim 进行初步的验证，在这一步中如果遇到错误可以通过波形就行纠错，或者通过一些辅助打印语句观察一些寄存器的值。一般而言，仿真出错分为以下两类：

(1) 波形出错：从波形图直接观察，而不需要分析电路设计的功能，就能判断错，比如波形信号中的“X”，或者高阻，或者一直为零。

(2) 逻辑出错：波形直接观察正常，但是电路执行结果不符合预期，属于逻辑出错，比如某些寄存器的值等等。

波形出错想对而言是更容易查出来的，而逻辑出错则需要重新认真审视一下电路的逻辑，必要时使用逻辑分析仪帮助调试。

4.2.1 testbench 搭建

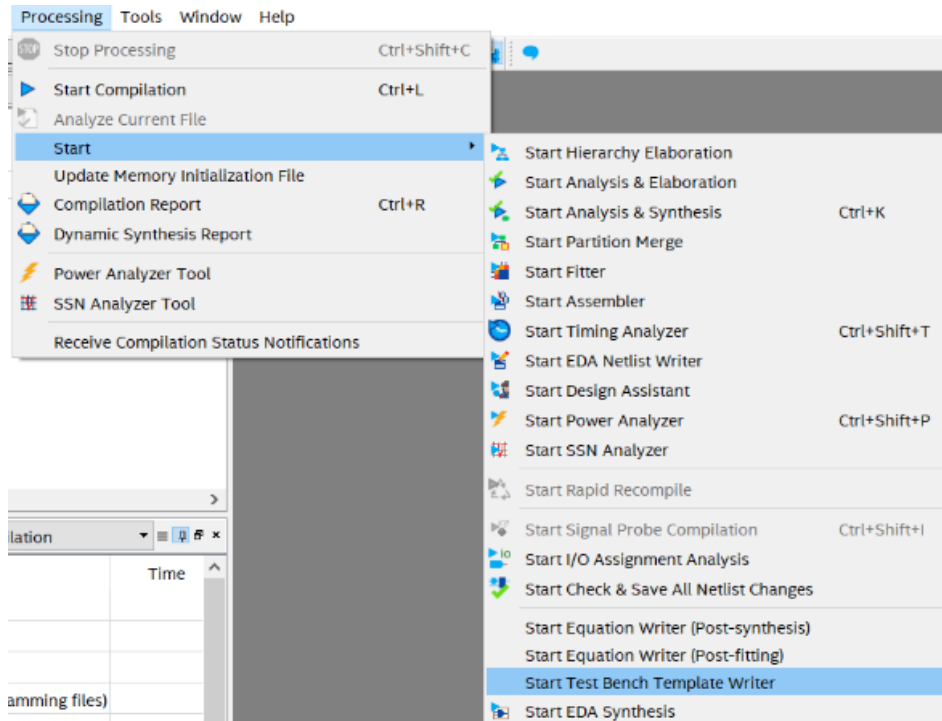


图 4-5 testbench搭建示意图

在 Quartus 中可以通过如上图的步骤生成一个 testbench，在添加一些时钟激励之后，还需要进行一个模拟串口的搭建。

这是由于从 SOC 出来的 RX，TX 这些关于 UART 的数据在仿真环境下并没有相应的接收设置，因为在 testbench 中只有时钟 CLK 和 RST 的激励，所以如果想测试 UART 是否正确，还需要用软件模拟一个真实串口才可以。

真实的串口一般满足 UART16550 协议，但是由于 SOC 的 UART 控制器只是完成了最基本的功能，所以只需要搭建一个和 SOC 中串口控制器相似的模拟串口即可。这部分的设计大致上和 UART 一样，一个不同点是，接收的时候需要选取采样点，在本工程中采用一个周期一半时刻的采样值。

4.2.2 测试用例

只有完善的测试用例才能检测是系统是否完全正确，这里采取的RISC-V官方的指令集进行测试。在测试之前还需要完成一些代码移植工作，具体内容如下：

(1)更改外设地址，主要是针对串口和RAM的地址更改。在表4-1中介绍了如何分配地址空间，对于UART工程中是将0x0200_0000分配给了这个地址，所以设计到串

口打印相关的功能时，需要把所有原先不匹配的地址修改一下。针对RAM地址则是具体体现在链接脚本上：

```
ENTRY(_start)
MEMORY
{
    ROM (rx) : ORIGIN = 0x00010000, LENGTH = 0xc000    /* ROM 48 KB */
    RAM (xrw): ORIGIN = 0x80000000, LENGTH = 0x800000 /* RAM 8 MB */
}
```

链接脚本是测试的一个关键的组成部分，其中 ROM 的大小要和 SOC 中设置的(见表 4-1)保持一致才可以。实际上 ROM 大小还必须必所生成的程序要大，这样才能保证 ROM 可以装的下这些程序。

生成的.hex 文件中的最后一行是 00000001FF 代表，而前一行代表最后的 ROM 的数据，对于 Intel 的 hex 格式文件，第 3-6 位代表从第一个地址开始的第多少个字，这里若 ROM 大小是 48KB，则这个数字应小于 48KB(dec)=0xC000(hex)，如果地址空间不够则需要将不同测例拆分开。

(2)完善测例与 makefile

对于原子指令的测试，可以直接加入到 start.S 中，并且通过#ifdef , #endif 这样的方式控制编译链接哪些测试。在测试 RV32UI 指令的时候，其余的指令就不会被测试，这样保证了在空间小的 ROM 下依然可以进行测试运行。

makefile 中包含了如何生成目标文件的脚本代码，由于需要将多个指令合并都到一起进行测试，所以针对原先的 makefile 需要进行修改。在 makefile 中添加了一个 mode 变量，默认是 rv32ui，还有 rv32ua 以及其余的特权级的指令可以选择。

4.2.3 添加测例

4.2.3.1 测例解析

在测试遇到问题的时刻，有时需要添加一些自定义的测例，由于官方测例较为晦涩，通过宏进行了层层封装，所以需要先将测例内容搞清楚再着手编写。

下面先以 addi.S 为例进行说明

```
#include "riscv_test.h"
```

```
#include "test_macros.h"

RVTEST_RV32U
RVTEST_CODE_BEGIN
```

打开 addi.S 文件后，看到在真正的测例宏前有如下几行，其中 `riscv_test` 包含了一些测试初始化，打印 PAST, FAIL 的宏，而 `test_macros.h` 则包含了不同指令测试的宏。具体之后还会有例子。

下面 `RVTEST_RV32U` 代表这是 32 位的测例，因为 `riscv-test` 只有在 64 位下才有真正的源代码，32 位只是借用了 64 位的测例，并通过宏的形式进行少量修改，因为要测试的是 32 位指令集，所以要有这行。

`RVTEST_CODE_BEGIN` 是来自于 `riscv_test.h`

```
#define RVTEST_CODE_BEGIN \
    .text; \
    .global TEST_FUNC_NAME; \
    .global TEST_FUNC_RET; \
    TEST_FUNC_NAME: \
        li a0, 0x00ff; \
    .delay_pr: \
        addi a0, a0, -1; \
        bne a0, zero, .delay_pr; \
        lui a0, %hi(.test_name); \
        addi a0, a0, %lo(.test_name); \
        lui a2, 0x02000000>>12; \
    ...
```

`TEST_FUNC_NAME` 这里就是代表测试 `addi` 由 `start.S` 调用

`.delay_pr` 是一个延时，原先可能是 `0xffff` 或者一个更大的数，但是在仿真下回消耗很大不必要的时间，这里调小了点，这部分是打印功能测试的名字，对于本例是 `addi..`，之后就进入了真正的测例。

这里是 `addi.S` 的第一条代码：

```
TEST_IMM_OP( 2, addi, 0x00000000, 0x00000000, 0x000 );
```

TEST_IMM_OP 是一个宏，定义如下：

```
#define TEST_IMM_OP( testnum, inst, result, val1, imm ) \  
    TEST_CASE( testnum, x3, result, \  
        li x1, val1; \  
        inst x3, x1, SEXT_IMM(imm); \  
    )
```

宏的声明表示接受 testnum,inst,result,val,imm 几个参数并调用 TEST_CASE 进行测试，TEST_CASE 定义如下：

```
#define TEST_CASE( testnum, testreg, correctval, code... ) \  
test_ ## testnum: \  
    code; \  
    li x29, correctval; \  
    li TESTNUM, testnum; \  
    bne testreg, x29, fail;
```

追踪到 TEST_CASE，一上来是一个声明第一个 test，test_ ## testnum 将会被展开成 test_1 test_2 的形式，之后 code 则是通过 TEST_IMM_OP 传进来的，这里是一个可变参量，所以可以有多条语句。之后将比对运算结果是否是正确的即 testreg 的数值是否和 correctval 相等，如果不相等就跳转到失败，打印"FAIL"然后返回。

将宏 TEST_IMM_OP 对于本例进行展开就是

```
li x1, 0x00000000,  
addi x3, x1, SEXT_IMM(0)  
li x29, 0  
li TESTNUM, 2  
bne x3, x29, fail
```

即验证 $0 + 0$ 的结果是否等于 0。

4.2.3.1 添加测例 SDRAM.S

这里由于官方对内存测试并不完整，于是可以依据现有测例进行补充添加：

```
li a0, 0x80000000
```

```
li a1, 0x80800000
test:
TEST_ST_OP( 2, lw, sw, 0x12345678, 0, a0 );
addi a0,a0,4
sub a2,a1,a0
bneq test
...
```

这里完成的是对 SDRAM 全域的读写测试，从 0x8000_0000 一直到 0x8080_0000，每次都向 a0 的地址写入 0x12345678 并对 a0 的地址进行读出，如果相同就测试下一个字直到结束。

这样利用原有测例中的宏辅助开展进行新的测试实验，节省了代码，减少了撰写测例时引入新的 bug，简化了测试与调试的流程。

4.2.4 测试难点——转换桥

4.3 本章小结

本章内容主要介绍 SOC 硬件平台的搭建及测试过程。首先介绍了 SOC 的整体框架，各部分组成，及地址分配。之后分别介绍了 ROM 控制器，SDRAM 的 IP 核，WISHBONE 与 AVALON 之间转换桥与 UART 控制器。将这些控制器使用总线交互组合起来就构成了 SOC 系统，最后给出了对当前 SOC 环境的测试用例及说明。

第5章 软件系统分析移植

对于软件系统而言，在工程中首先移植了基于 LZ77 和 Huffman 编码的解压缩程序，通过移植解压缩程序使得整个工程文件容量减少至原来的十分之一，这部分内容对应于 5.1 节。在解压缩工作之后，由 BBL 引导加载操作系统加载至内存相应位置，在 5.2 节将着重介绍 BBL 相关的移植工作。在 BBL 完成操作系统引导后，系统控制权交给 rCore 操作系统，在 5.3 节中对 rCore 操作系统展开相应的介绍。

整个软件系统整体框架示意图如下所示：

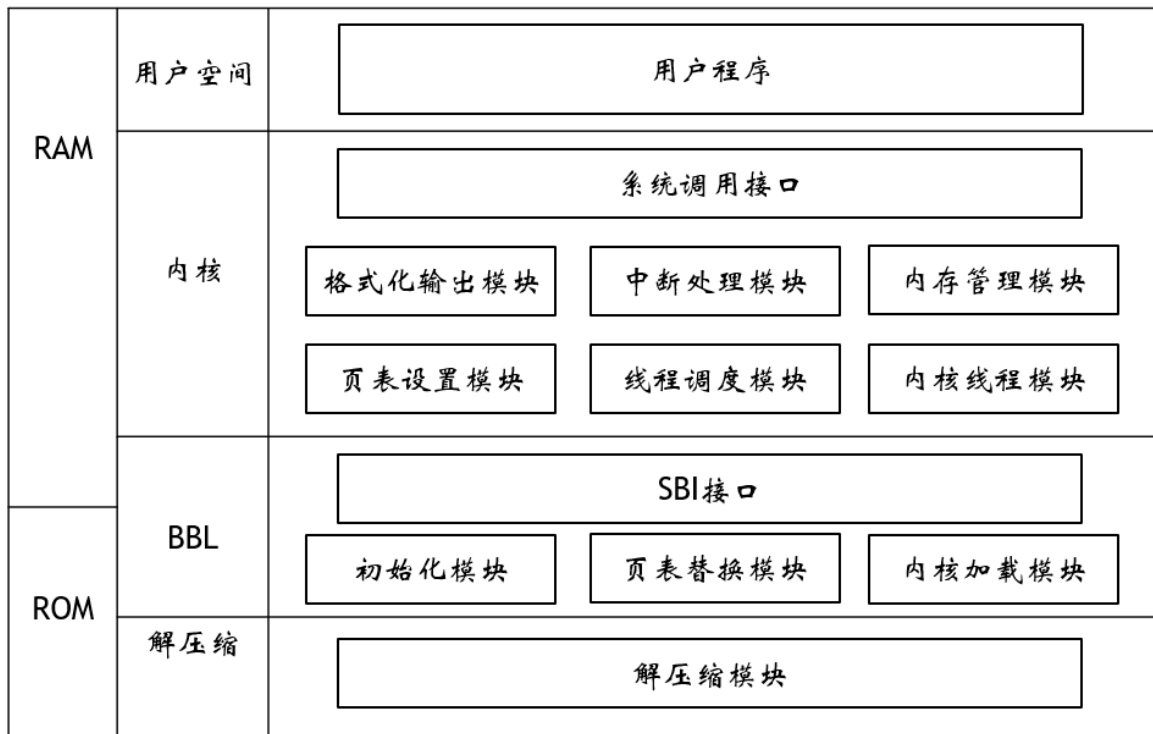


图 5-1 软件系统整体框架示意图

5.1 解压缩适配

5.1.1 压缩原理介绍

为了最大限度的应用有限的存储空间，我们将操作系统镜像进行压缩后加载到 rom，通过解压缩程序将 rom 中的镜像解压至 sdram 中执行。受到存储空间的限制，并且考虑到开发环境的因素，压缩镜像格式我们选取 gzip 格式，且为了节省空间，我们选取一个应用于嵌入式，并且可对数组按字节进行解压的解压方式：tinf，tinf

是一个小型的解压缩库，专为 deflate 压缩数据格式提供解压缩算法。我们可调用其解压缩函数直接对由 rustos 压缩镜像转换成的数组进行解压缩，并解压到指定地址，以达到最大限度应用有限 rom 空间的目的。在这个压缩过程中，使用到了两种算法，LZ77 和 Huffman 编码。

(1)LZ77 算法

通过使用编码器或者解码器中已经出现过的相应匹配数据信息替换当前数据从而实现压缩功能。这个匹配信息使用称为长度-距离对的一对数据进行编码，它等同于“每个给定长度的字符都等于后面特定距离字符位置上的未压缩数据流。”（“距离”有时也称作“偏移”。）

编码器和解码器都必须保存一定数量的最近的数据，如最近 2 KB、4 KB 或者 32 KB 的数据。保存这些数据的结构叫作滑动窗口，因为这样所以 LZ77 有时也称作滑动窗口压缩。编码器需要保存这个数据查找匹配数据，解码器保存这个数据解释编码器所指代的匹配数据。

(2)Huffman 编码

霍夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为 0 层，叶结点到根结点的路径长度为叶结点的层数）。树的路径长度是从树根到每一结点的路径长度之和，记为 $WPL = (W_1 * L_1 + W_2 * L_2 + W_3 * L_3 + \dots + W_n * L_n)$ ，N 个权值 W_i ($i=1,2,\dots,n$) 构成一棵有 N 个叶结点的二叉树，相应的叶结点的路径长度为 L_i ($i=1,2,\dots,n$)。可以证明霍夫曼树的 WPL 是最小的。得到保存在压缩文件中的每个符号的出现次数的信息。根据每个符号的出现次数，建立 Huffman 树，得到每个符号的 Huffman 编码。将压缩文件中的每个 Huffman 编码替换成它对应的符号，并输出。

5.1.2 解压缩代码适配

对于解压缩代码部分无需进行更多的修改，只需要在硬件和软件添加相应的处理就可以。对于硬件而言，新建一个为 decompressor 的 ROM 控制块，大小为 4KB，这里只需按照一般 ROM 的控制器处理就可以，然后在总线转换的时候加入 decompressor 的地址映射。

<code>`define DECOMPRESSOR_PHYSICAL_ADDR_BEGIN 34'h00003_0000</code>
--


```
`define DECOMPRESSOR_PHYSICAL_ADDR_LEN    34'h00000_1000 // 4KB
```

对于软件部分而言，使用 tinfl 的 C 语言文件进行压缩，在压缩之前还需清空 bss 段，将在片上的 ROM 段拷贝到 SDRAM 上，设置堆栈指针，最后才轮到解压缩功能。

5.2 BBL 分析与移植

5.2.1 BBL 解析

Berkeley Boot Loader (BBL) 是 M 态的程序，可以引导我们移植的 rCore 以及操作系统。其基本上可以认为是硬件/软件的接口，无论是对于操作系统的移植，还是对于 RISC-V 的硬件设计，都是同等的重要的。下面我们简要介绍 BBL 所完成的功能。

5.2.1.1 引导部分

在运行 BBL 之前，首先应将 BBL 置于内存 0x8000_0000 之后的位置，在真实硬件中，CPU 加电后执行 0x00001000 处的首条指令，通过 auipc 跳转到 0x80000000 开始执行 BBL 的启动代码。0x8000_0000 处对应的是 ./machine/mentry.S 中的一条跳转到 do_reset 指令。之后跳转到 do_reset 时首先进行的是寄存器清零，置 mscratch 为 0。接下来将 mtvec 设置为 trap_vector 的地址，并进行检测。设置 sp 为 binary 最后的位置(页对齐)，跳转到函数 init_first_hart。

随后，它会清空所有的寄存器，置 mscratch 为零。设置中断向量 mtvec，设置 mstatus.VM 为 VM_SV32 虚拟页式存储。随后将 mideleg, medeleg 寄存器设置重定向到 S 态的中断和异常。然后它会读取 parse_config_string，获取硬件相关的信息，包括内存大小，外部中断，时钟特权寄存器 mtime, mtimecmp 的内存映射位置，等等。之后，它会设置内存，加载 elf 格式的 S 态 OS，并设置页表，使其线性映射到 elf 中设定的虚拟地址。最后通过 mret 返回到 OS 入口处。

5.2.1.2 中断、异常部分

Berkeley Boot Loader 的中断、异常入口在 ./machine/mentry.S 的

trap_vector 位置。根据中断、异常原因，它会调用不同的处理函数。例如，如果发生了时钟中断，它会调用 timer_interrupt 函数，如果发生了 S 态的系统调用，它会进一步调用 mcall_trap 函数等等。通过异常， bbl 可以代替硬件模拟一些指令。当硬件检测到没有实现的非法指令时，会陷入 BBL 的异常。根据异常原因， BBL 会进一步调用 illegal_insn_trap 函数。此函数会模拟一部分没有实现的指令，并返回到中断处。通过这样的设置， BBL 不仅可以模拟浮点运算，对后文中提到的 TLB 也可模拟实现。

5.2.1.3 SBI 接口

SBI，即 Supervisor Binary Interface。为实现对上层系统提供调用接口的功能，引导程序通过 ecall 软中断的形式提供系统调用接口。通过中断处理函数没有预留参数作为系统调用号， 执行相应的如串口打印的功能。

SBI 呈现为一组函数，它的实现在 ./machine/sbi_entry.S 中，OS 只能获得头文件 ./machine/sbi.h 和对应的函数地址 ./machine/sbi.S。

```
// sbie_entry.S
.align 4
li a7, MCALL_CONSOLE_PUTCHAR
ecall
mret
```

这组函数实现的就是调用 ecall，而 a7 则作为中断向量号进行识别，处理之后就执行 mret 返回原来的代码位置。

为了实现这样的设计，需要操作系统在设置页表时单独为 SBI 设置一页，物理地址为对应的 SBI 函数入口。

```
// ./bbl/bbl.c
extern char _sbi_end;
uintptr_t num_sbi_pages = ((uintptr_t)&_sbi_end - DRAM_BASE - 1) / RISCVP_GSIZE
+ 1;
assert(num_sbi_pages <= (1 << RISCVP_GLEVEL_BITS));
for (uintptr_t i = 0; i < num_sbi_pages; i++) {
```

```

uintptr_t idx = (1 << RISCVP_GLEVEL_BITS) - num_sbi_pages + i;
sbi_pt[idx] = pte_create((DRAM_BASE / RISCVP_PGSIZE) + i, PTE_G | PTE_R | PTE_X);
}

```

5.2.2 BBL 修改

5.2.2.1 降低初始映射粒度

在 BBL 中，进行了粗粒度的页表映射，由于这部分页表映射的粒度为 4MB 对齐的，这样 BBL 占用的地址就至少是 4MB 空间，但实际上 BBL 仅用了几十 KB，为了减少内存空间的浪费，工程中将映射的粒度由 4MB 减少到 4KB，4KB 也是 RISC-V 架构支持的最小的页表单元。

在 ./bbl/bbl.c 中需要将首个空闲的物理地址对齐成 4KB：

```

- first_free_paddr = ROUNDUP(first_free_paddr, MEGAPAGE_SIZE);
+ first_free_paddr = ROUNDUP(first_free_paddr, RISCVP_PGSIZE);

```

这样相比原来最多损失的内存空间大小是 4KB，减少了 1024 倍，其次还需要在页表初始化进行修改。首先是下面所示的更改，在获取内存大小的时候此时只需要对齐到 4KB 就可以而不能是 4MB，这样可以获取到准确的哪些内存可用。

```

+ mem_size = MIN(mem_size, highest_va - info.first_user_vaddr) & -RISCVP_PGSIZE;
- mem_size = MIN(mem_size, highest_va - info.first_user_vaddr) & -MEGAPAGE_SIZE;

```

此外在初始化页表的时候需要进行一些改动，主要是下面这片代码所示的那样。首先对每个页进行映射的时候粒度是 4KB，所以 for 循环中每次虚拟地址和物理地址都要加 4KB 而不是 4MB，其次在每次设置页表项即 middle_pt 那一行，需要调用的是 pte_create 而非 ptd_create 表明这是一个 4KB 的页表映射而不是一个 4MB 的巨页也行社，此外对于偏移量 ll_idx 也需要设置成 4KB 对其的形式。

```

for (uintptr_t vaddr = info.first_user_vaddr, paddr = vaddr + info.load_offset, end = info.first_vaddr_after_user; paddr < DRAM_BASE + mem_size; vaddr += RISCVP_PGSIZE, paddr += RISCVP_PGSIZE) {
    int ll_shift = RISCVP_PGSHIFT;

```

```

size_t ll_idx = (info.first_user_vaddr >> ll_shift) & ((1 << RISCV_PGLEVEL
_BITS)-1);

ll_idx += ((vaddr - info.first_user_vaddr) >> ll_shift);

middle_pt[ll_idx] = pte_create(paddr >> RISCV_PGSHIFT, PTE_G | PTE_R |
PTE_W | PTE_X);
}

```

5.2.2.2 TLB 缺失异常处理

由于硬件资源有限并考虑到硬件实现的复杂度，大部分 TLB 的替换都是由软件完成的，硬件只维护一个基本的读写功能，如当 TLB 满的时候需要写入新的页表项的时候硬件只负责读取和写入某一个项，而究竟这个项的内容是什么，选择哪个项进行替换这都是交由软件维护的，这部分对于虚拟内存的支持是不可或缺的。

```

trap_table:
    .word bad_trap
+ .word tlb_i_miss_trap
    .word illegal_insn_trap
    .word bad_trap
    .word misaligned_load_trap
+ .word tlb_r_miss_trap
    .word misaligned_store_trap
+ .word tlb_w_miss_trap

```

这个 trap_table 是异常向量表的意思，之前在这部分代码块中，所有带“+”的部分，都是 bad_trap，意思是这个异常是不应该发生的，而我们此时对于 TLB 满的时候或是读取指令缺失或是数据读写缺失都分别新增加一个 trap，也就是说此时硬件或报一个异常，然后根据相应的偏移值进行相应的处理如 tlb_i_miss_trap

对于 tlb_i_miss_trap 而言,他对应的函数位于 ./machine/emulation.c 中:

```

void tlb_i_miss_trap(uintptr_t* regs, uintptr_t mcause, uintptr_t mepc) {
    tlb_miss_trap(regs, mcause, mepc, 1, 0, 0);
}

```

具体这个函数的细节在后文会提到，目前只需清楚这个 `trap_table` 是存储的各种 `trap` 的处理程序的地址就可以。由于这些 `trap` 都是在 `m` 态被触发的，结合 RISC-V 的架构，此时有一个叫 `mtvec` 的特权寄存器扮演了重要角色，在处理器 `m` 态下发生异常时，硬件会根据 `mtvec` 跳转到相应的地址，那么在 `bb1` 中 `mtvec` 被设置的过程，实际上就在 `./machine/mentry.S` 中：

```
la t0, trap_vector
csw mtvec, t0
```

通过这里可以看到，`bb1` 把 `trap_vector` 赋值给了 `mtvec`，而 `trap_vector` 也在这个文件中，对应代码如下：

```
trap_vector:
    csrrw sp, mscratch, sp
    beqzsp, .Ltrap_from_machine_mode #这里也是跳转到到.Lhandle_trap_in_machine_mode
    STORE a0, 10*REGBYTES(sp)
    STORE a1, 11*REGBYTES(sp)

    csrr a1, mcause
    bgez a1, .Lhandle_trap_in_machine_mode

    # This is an interrupt. Discard the mcause MSB and decode the rest.
    sll a1, a1, 1

    # Is it a machine timer interrupt?
    li a0, IRQ_M_TIMER * 2
    bne a0, a1, 1f
    li a1, TIMER_INTERRUPT_VECTOR
    j .Lhandle_trap_in_machine_mode
```

在这段程序可以看到，根据中断的不同类型，最终都会跳转到 `.Lhandle_trap_in_machine_mode` 中去。

```

.Lhandle_trap_in_machine_mode:
# Preserve the registers.  Compute the address of the trap handler.
# more store...
l:auipc t0, %pcrel_hi(trap_table)  # t0 <- %hi(trap_table)
    STORE t1, 6*REGBYTES(sp)
    sll t1, a1, 2                    # t1 <- mcause << 2
    STORE t2, 7*REGBYTES(sp)
    add t1, t0, t1                  # t1 <- %hi(trap_table)[mcause]
    STORE s0, 8*REGBYTES(sp)
    LWU t1, %pcrel_lo(1b)(t1) # t1 <- trap_table[mcause] #GOT 表 indirect addressing
    STORE s1, 9*REGBYTES(sp)
    # more store ...
    jalr t1 # 跳转到 t1 对应的地址
    # restore ...

```

可以看到根据 `mcause` 选择相应的 `trap_table` 的偏移量(即对应哪个 `trap` 处理程序), `t1` 最终就指向了对应的处理程序的地址, 最终一个 `jalr` 就跳转过去了。

整个 TLB 的流程从宏观上看如下: 当操作系统在 S 态发生一个 `tlb_i_miss` 的时候, 会抛出一个 `strap` 异常, `strap` 由于 `medeleg` 的设置对应位是 0, 所以交给了 M 态处理, 处理的函数就是 `trap_vector`, 根据 `mcause` 里面对应的 `trap`, 软件就会知道这个是一个 `tlb_i_miss`, 进行一些跳转前的保护寄存器的工作后, 就跳转到这个 `trap_table` 里面对应的 `tlb_i_miss` 的地址上去执行了, 执行完毕后, 就恢复寄存器最后执行 `mret` 返回异常之前的地址。

5.2.2.3 TLB 替换的细节

无论是指令缺失还是数据缺失最终都会引到 `tlb_miss_trap` 中, 只不过属性值不太一样而已, `tlb_miss_trap` 是控制 `tlb` 的核心, 这个函数的原型如下:

```
void tlb_miss_trap(uintptr_t* regs, uintptr_t mcause, uintptr_t mepc, int ex, int rd, int wt)
```

第一个参数 `regs`, 含义是寄存器的地址, 而 `mcause`, `mepc` 就是 `csr` 中的数值, 需要注意的是, 这里的 `regs`, `mcause` 都已经被实实在在地存储在内存中某个地方, 而

不是硬件中地某个 LUT,FF，因为如果只是存储在硬件中，软件就无法访问了。

由于 mcause 和 mepc 都是存在硬件上的某个寄存器，所以需要将他们从硬件转移到软件上，这部分其实是在刚刚的 .Lhandle_trap_in_machine_mode 中完成了：

```
...
csrr a1, mcause
...
STORE s1, 9*REGBYTES(sp)
mv a0, sp                # a0 <- regs
STORE a2, 12*REGBYTES(sp)
csrr a2, mepc            # a2 <- mepc
...
```

实际上 regs(所对应的堆栈 sp)和 mcause, mepc 已经被保存到 a0, a1, a2 上了，根据 cdecl 调用规则和 riscv 的寄存器调用规则可以知道，当调用这个 tlb_miss_trap 函数的时候，a2, a1, a1(从右到左)会被依次压栈，然后 tlb_miss_trap 进入这个函数的时候就会依次 pop 出来使用了。

在 tlb_miss_trap 函数中 首先获取 mstatus 中 vm 的数值以方便知道是用的 RV_32 页表还是其他形式的 TLB，然后根据 __riscv_xlen 的数值,判断是 32 位还是 64 位的系统，从而获取相应的页表基地址。在本工程中采用的是 VM_32，此时设置 level=2, ptesize=4, vplen=10,意思是两级页表，每个页表 4 字节，tlb 中虚地址的长度是 10。

```
uintptr_t mstatus = read_csr(mstatus);
uint32_t vm = (EXTRACT_FIELD(mstatus, MSTATUS_VM));

#if __riscv_xlen == 32
    uint32_t p = 32;
    uintptr_t a = ((read_csr(sptbr)) & ((1 << 22) - 1)) * RISCVPGSIZE;
#else
    ...
#endif

switch(vm) {
```

```

case VM_SV32: levels = 2; ptesize = 4; vpnlen = 10; break;

    ...

}

```

当第一次进入下面的循环时, `a` 代表上文中对应的 `root_page_table` 找到的 `pte`, 而 `pte_p` 就代表这个虚拟地址对应的一级页表的地址, 然后 `pte` 就是一级页表(或者叫巨页), 页目录项的值了。进行一些检查之后, 如果当前页表的内容是指向下一级的(`XWR` 均为 0), 那么更新 `a` 位当前 `pte` 对应的页表项基址:

```

for (i = levels - 1; i --) {
p -= vpnlen; // p = 32 - 10 = 22
// 之前 mask = 0
mask = ~((~mask) >> vpnlen); //这行之后 mask = 1111_1111_1000_00...._0000
uintptr_t vpn = ((va >> p) & ((1 << vpnlen) - 1)); //vpn = va[31:22]
uintptr_t *pte_p = (uintptr_t *) (a + vpn * ptesize); // a = root_page_table pte 相当于
是 root 的偏移量
uintptr_t pte = *pte_p;

```

此时已经获取到要页表项的值, 如果是 `tlbmiss` 的情况, 则直接根据对应的 `index` 更新 `tlb` 页表项的数值即可, 否则只需要更新虚拟地址对应的物理地址的表项:

```

if(((uintptr_t)read_csr(0x7c0)) >> (__riscv_xlen - 1)) {
    uintptr_t index_old = read_csr(0x7c0);
    uintptr_t va_old = read_csr(0x7c1);
    uintptr_t mask_old = read_csr(0x7c2);
    uintptr_t pte_old = read_csr(0x7c3);
    uintptr_t *pte_p_old = (uintptr_t *)read_csr(0x7c4);
    *pte_p = pte;
    write_csr(0x7c3, pte);

    write_csr(0x7c0, index_old);
    assert(read_csr(0x7c0) == (index_old << 1) >> 1);
    return;
}

```



```
else {  
    write_csr(0x7c0, index);  
    write_csr(0x7c1, va & mask);  
    write_csr(0x7c2, mask);  
    write_csr(0x7c3, pte);  
    write_csr(0x7c4, pte_p);  
    index += 1;  
    return;  
}
```

这部分是通过添加一些用户自定义指令实现的，新增了 5 个特权级指令，如下所示：

表 5-1 新增特权级指令

指令地址	对应硬件宏	含义
0x7c0	CSR_mtlbindex	index 位
0x7c1	CSR_mtlbvpn	虚拟地址位
0x7c2	CSR_mtlbmask	对齐的数值
0x7c3	CSR_mtlbpte	该 TLB 对应的物理地址
0x7c4	CSR_mtlbptevaddr	pte 对应的虚拟地址

这部分在硬件中也需要添加相应的支持，主要包括对于这些新加入的 CSR 寄存器的读写支持；对于 index 位，由于 TLB 是 16 页表项，所以 index 的范围是 4'b0000 ~ 4'b1111。但 index 位宽是 32 位，在实现中还利用 index 最高位宽表明是否需要更新，也就是当找到了这个 tlb 表项，但是由于访问权限不对的时候，这个 index 最高位就是 1，进入了上文的 if 块，而如果 TLB 根本没有命中，那么最高位就没有置 1，也就是进入了上文的 else 块。

5.3 操作系统分析与移植

这部分将分析操作系统 rCore 及相关的移植过程，更详细的步骤已通过 gitbook 的形式发布在 <https://moon548834.github.io/cyc10-rcore-tutorial/> 上。

5.3.1 最小化内核

5.3.1.1 移除标准库依赖

操作系统的本质是一个独立可执行的程序，这决定了操作系统在编写过程中不能依赖于某些平台的函数库，所以首先要做的是去除标准库的引用，对于 Rust 而言创建完 cargo 后加入：

```
#![no_std]
#![no_main]
```

其中 no_std 代表不链接标准库，而 no_main 表示不以 main 作为 Rust 的默认入口点。

5.3.1.2 添加对 panic 的支持

如果没有了 std 库，就需自己实现 panic 功能，这代表操作系统在出现错误时的需要进行维护的一些工作

```
use core::panic::PanicInfo;

// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

这样程序在 panic 后就会在 loop 中一直循环下去而并不返回，这里的“!”代表这个函数不会返回，在这段代码中，使用到了核心库 core，这与标准库 std 并不一样，它不需要操作系统支持，可以直接调用。

5.3.1.3 编写链接脚本

```
OUTPUT_ARCH(riscv)

ENTRY(_start)

BASE_ADDRESS = 0x80200000;

SECTIONS
```

```
{  
    . = BASE_ADDRESS;  
    start = .;  
    .text : {  
        stext = .;  
        *(.text.entry)  
        *(.text.text.*)  
        . = ALIGN(4K);  
        etext = .;  
    }  
    ....  
}
```

首先使用 `OUTPUT_ARCH` 指定架构，`ENTRY` 指定整个可执行文件的入口点是 `_start`，所以 `_start` 函数是整个操作系统第一条语句执行的地方。

链接脚本整体写在 `SECTION` 中，这里面有形如像代码段中 `text` 所展示的每一个小的段，以 `.text` 为例代表了这个段开始的地址是 `stext`，这里面包括了任何以后缀为 `.text.entry` 和 `.text.text` 的内容，`etext` 就是 `text` 段的结尾，并以 `4K` 对齐。

这个链接脚本从 `BASE_ADDRESS` 开始向下放置各个段，依次是 `.text`, `.rodata`, `.data`, `.stack`, `.bss`。并像 `text` 段中那样记录了他们的开始和结尾地址。

5.3.1.4 重写 `_start` 入口

```
.section .text.entry  
.globl _start  
_start:  
    la sp, bootstacktop  
    call rust_main
```

这里使用 `.globl` 代表是一个全局地址，这使得链接脚本可以找到这个 `_start`，设置完堆栈指针后，使用 `call` 指令进入到 `rust_main` 这个函数这意味着内核运行环境设置完毕，正式进入内核。

5.3.1.5 封装 SBI

BBL 实际上不仅起到了 `bootloader` 的作用，还提供了一些服务供我们在编写内核的时候使用，这层接口就被称作 `SBI(Supervisor Binary Interface)`。是 `S` 模式下内核与 `M` 模式下内核环境之间的标准接口。这部分最核心的代码如下：

```
fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
    let ret;
    unsafe {
        asm!("ecall"
            : "{x10}" (ret)
            : "{x10}" (arg0), "{x11}" (arg1), "{x12}" (arg2), "{x17}" (which)
            : "memory"
            : "volatile");
    }
    ret
}

pub fn console_putchar(ch: usize) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}

...
```

这代表调用 `ecall` 指令，把函数返回值给 `x10` 寄存器，将参数分别传入到 `x10,x11,x12,x17` 寄存器，`which` 是一个比较重要的参数，`ecall` 根据 `which` 指明对应的 `sbi` 服务，这样底层的 `BBL` 就可以根据这个值执行相应的服务。在操作系统运行中，执行某个打印的功能时，调用 `console_putchar` 这个函数，这个函数进一步调用 `ecall`，根据 `BBL` 已经设置好的中断服务程序，就完成了打印字符的功能。

5.3.2 中断管理

5.3.2.1 栈帧

在中断发生那一刻，程序的运行状态或称之为上下文环境需要保存在一些寄存器

种，硬件的工作只有设置中断原因，中断地址，然后根据 `stvec` 直接跳转到中断处理函数，这时除了硬件的工作外，软件也需要将当前任务的寄存器保存在栈上，这样等中断处理程序结束之后就可以从栈上恢复中断前的运行环境。

```
pub struct TrapFrame {  
    pub x: [usize; 32], // General registers  
    pub sstatus: Sstatus, // Supervisor Status Register  
    pub sepc: usize, // Supervisor exception program counter  
    pub stval: usize, // Supervisor trap value  
    pub scause: Scause, // Scause register: record the cause of exception/interrupt/trap  
}
```

在操作系统中，将所有的寄存器和一些必要的特权级寄存器保存下来，这些内容称之为栈帧，当中断发生的时候，需要将这些寄存器依次入栈，而中断结束的时候再将他们依次出栈。

5.3.2.2 上下文环境的保存与恢复

保存上下文环境具体是指将整个栈帧保存在内核栈上，如果现在操作系统就处在内核态，他们现在的栈顶指针 `sp` 就是指向了内核栈的地址。但如果现在运行的任务是用户态的，那么中断时还需要完成从用户栈切换到内核栈。

规定如果在中断前时处于用户态，则 `sscratch` 保存的时内核栈地址，如果中断前是内核态，则 `sscratch` 保存的是 0。

```
.macro SAVE_ALL  
    csrrw sp, sscratch, sp  
    bnez sp, trap_from_user  
trap_from_kernel:  
    csrr sp, sscratch  
trap_from_user:
```

`csrrw` 指令是原子交换，在上面的代码块中，将 `sp` 写入到 `sscratch` 中，并再将 `sscratch` 写进 `sp` 中，实际上就是原子的交换了 `sp` 和 `sscratch` 的值。如果 `sp=0`，那么

表明之前的是内核栈，不用切换，因此不跳转，将 `sscratch` 读回 `sp`。否则就是从用户态进入中断，需要切换栈。

```
trap_from_user:
    addi sp, sp, -36*XLENB
    STORE x1, 1
    STORE x3, 3
    STORE x4, 4
    ...
    STORE x30, 30
    STORE x31, 31
    csrrw s0, sscratch, x0
    # 分别将四个寄存器的值保存在 s1,s2,s3,s4 中
    csrr s1, sstatus
    csrr s2, sepc
    csrr s3, stval
    csrr s4, scause

    STORE s0, 2
    STORE s1, 32
    ...
```

首先需要提前分配栈帧，这里是 36 个 word，然后存放除了 `sp` 和 `x0` 以外的 30 个通用寄存器，因为 `sp` 是堆栈指针，需要特殊处理，而 `x0` 永远为 0 所以不必保存。此时，如果从内核态进入则 `sscratch` 为内核栈地址，否则是用户栈地址。按照规定，再内核态中 `sscratch` 为 0，所以需要把堆栈指针地址保存到内核栈上，以便于恢复。处理完最复杂的 `sp` 指针后，将其余的按照栈帧的顺序依次压栈，至此保存工作完成。恢复过程只需按顺序读出，所有写入的操作变为读即可。

5.3.3 页表管理

5.3.3.1 线段树管理物理内存

线段树是使用一个完全二叉树来存储对应于其每一个区间的数据。该二叉树的每一个结点保存着对应于这一个区间的信息。同时，线段树所使用的二叉树是用一个数组保存的，与堆的实现方式相同。

例如，给定一个长度为 N 的数组 arr (第一个元素的下标为 1)，其所对应的线段树 T 各个结点的含义如下：

1. T 的根结点代表整个数组所在的区间对应的信息，即 $arr[1:N]$ 所对应的信息。
2. T 的每一个叶结点存储对应于输入数组的每一个单个元素构成的区间 $arr[i]$ 所对应的信息，此处 $0 \leq i < N$ 。
3. T 的每一个中间结点存储对应于输入数组某一区间 $arr[i:j]$ 对应的信息，此处 $0 \leq i < j < N$ 。

以根结点为例，根结点代表 $arr[0:N]$ 区间所对应的信息，接着根结点被分为两个子树，分别存储 $arr[0:(N-1)/2]$ 及 $arr[(N-1)/2+1:N]$ 两个子区间对应的信息。也就是说，对于每一个结点，其左右子结点分别存储母结点区间拆分为两半之后各自区间的信息。也就是说对于长度为 N 的输入数组，线段树的高度为 $\log N$ 。

具体到内存管理中，节点上的值是只有两个 0 或 1，表示这个节点对应的区间内是否还有物理空闲页(0 代表空闲，1 代表被占用了)，最小的节点代表 4KB，树的高度每增加 1，则该节点代表的物理内存扩大一倍，如下图所示：

1 [0,32KB)							
2 [0, 16KB]				3 [16KB, 32KB]			
4 [0,8KB)		5 [8KB, 16KB)		6 [16KB, 24KB)		7 [24KB, 32KB)	
8 [0KB, 4KB)	9 [4KB, 8KB)	10 [8KB, 12KB)	11 [12KB, 16KB)	12 [17KB, 20KB)	13 [20KB, 24KB)	14 [24KB, 28KB)	15 [28KB, 32KB)

图 5-1 线段树内存管理示意图

当分配某个页表时，先从根开始查找，如果为 0 表明还有空余则从左右两侧挑选一个为 0 的子节点进行继续查找，直到下一级比想要分配的还要小就停止搜索。这时所在的节点和所有该节点的子节点都被置为 1。恢复的时候是从底向上的逆过程，使用线段树的好处是可以将每次分配所需要的时间减少到 $O(\log n)$ 。

5.3.3.2 从虚拟内存到物理内存

在 S 模式下，提供了一种传统的虚拟内存系统，它将内存分为固定大小的页来进行地址转换和对内存内容的保护。启用分页的时候，大多数地址都是虚拟地址。如果想访问物理内存，那么它们必须被转换为真正的物理地址，这通过遍历一种称为页表的高基数树来实现。页表中的叶节点指示虚拟地址是否已经被映射到了真正的物理页表，如果是，则指示了哪些权限模式和通过哪种类型的访问可以操作这个页[13]。

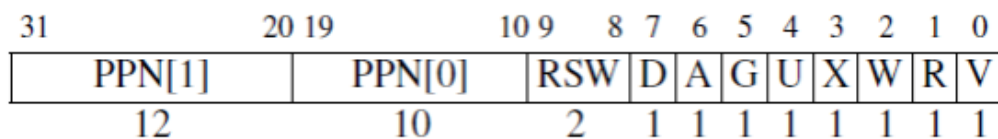


图 5-2 RV32 页表项(PTE)

图 5-2 显示了 Sv32 页表项（page-table entry, PTE）的布局，从左到右分别包含如下所述的域：

V 位决定了该页表项的其余部分是否有效（V = 1 时有效）。若 V = 0，则任何遍历到此页表项的虚址转换操作都会导致页错误。

R、W 和 X 位分别表示此页是否可以读取、写入和执行。如果这三个位都是 0，那么这个页表项是指向下一级页表的指针，否则它是页表树的一个叶节点。

U 位表示该页是否是用户页面。若 U = 0，则 U 模式不能访问此页面，但 S 模式可以。若 U = 1，则 U 模式下能访问这个页面，而 S 模式不能。

G 位表示这个映射是否对所有虚址空间有效，硬件可以用这个信息来提高地址转

换的性能。这一位通常只用于属于操作系统的页面。

A 位表示自从上次 A 位被清除以来，该页面是否被访问过。

D 位表示自从上次清除 D 位以来页面是否被弄脏（例如被写入）。

RSW 域留给操作系统使用，它会被硬件忽略。

PPN 域包含物理页号，这是物理地址的一部分。若这个页表项是一个叶节点，那么 PPN 是转换后物理地址的一部分。否则 PPN 给出下一节页表的地址[13]。

（图 10.10 将 PPN 划分为两个子域，以简化地址转换算法的描述。）

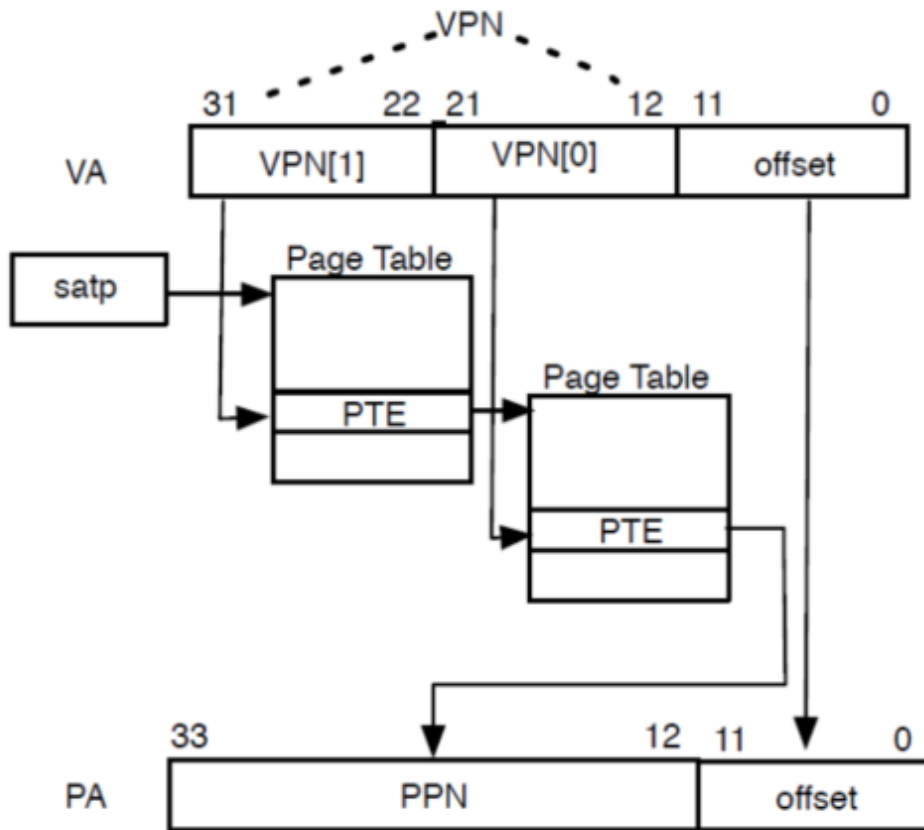


图 5-3 RV32 地址转化过程

当在 satp 寄存器中启用了分页时，S 模式和 U 模式中的虚拟地址会以从根部遍历页表的方式转换为物理地址。图 10.14 描述了这个过程[13]：

1. satp.PPN 给出了一级页表的基址，VA[31:22]给出了一级页号，因此处理器会读取位于地址 $(\text{satp.PPN} \times 4096 + \text{VA}[31:22] \times 4)$ 的页表项。

2. 该 PTE 包含二级页表的基址，VA[21:12]给出了二级页号，因此处理器读取位于地址 $(\text{PTE.PPN} \times 4096 + \text{VA}[21:12] \times 4)$ 的叶节点页表项。

3. 叶节点页表项的 PPN 字段和页内偏移（原始虚址的最低 12 个有效位）组成了最终结果：物理地址就是 $(\text{LeafPTE.PPN} \times 4096 + \text{VA}[11:0])$

5.3.3.3 内核初始映射

由于支持了虚拟内存，所以在启动的时刻需要先进行粗糙的映射，将链接地址设置为 0xC0000000 而物理地址仍然是从 0x80000000 开始，所以获得初始的页表后，需要减去偏移量，然后写入 satp，使能 satp 的最高位，再刷新 tlb 就可以了：

```
lui    t0, %hi(boot_page_table_sv32)
li     t1, 0xC0000000 - 0x80000000
sub    t0, t0, t1
srli   t0, t0, 12
li     t1, 1 << 31
or     t0, t0, t1
csrw   satp, t0
sfence.vma
```

其中 boot_page_table_sv32 是最初始的映射，里面保留了一个从 0xC040_0000 到 0x8040_0000 的巨页映射：

```
boot_page_table_sv32:
    .zero 4 * 514
    .zero 4 * 255
    # 0xC0400000 -> 0x80400000 (4M)
    .word 0x2010002f #DAG XWRV
    .zero 4 * 254
boot_page_table_sv32_top:
```

5.3.4 线程切换

5.3.4.1 线程上下文

线程的执行情况一般包括，CPU 各寄存器状态，尤其是 PC 和堆栈指针 SP，以及线程栈中的内容，这会包括函数中的局部变量。

一个线程不会总是占用 CPU 的资源，因为在执行过程之中，它可能会被切换出去；然后过一段时间又被切换回来，所以为了在切换前后状态能保持一致，就需要把

线程状态的信息保存下来，等到该线程再次被调度的时候恢复。根据 RISC-V 规范，调用者只需保存寄存器 s0 ~ s11，所以需要最终的线程状态信息如下：

```
pub struct ContextContent {  
    pub ra: usize,  
    satp: usize,  
    s: [usize; 12],  
    tf: TrapFrame,  
}
```

其中 ra 代表返回地址，satp 代表页基址，s 代表调用者保护的寄存器，最后的中断帧是用来进行线程初始化。

5.3.4.2 线程切换

```
pub unsafe extern "C" fn switch(&mut self, target: &mut Context) {  
    asm!(include_str!("process/switch.asm") ::: "volatile");  
}
```

这部分需要汇编语言的支持，在调用汇编语言之前，按照函数调用约定，将当前的线程状态地址传递给 a0 寄存器，将目标线程的状态地址传递给 a1。

在这个汇编文件中，和中断上下文的保存恢复一样，也是先将这些原来被切换的线程寄存器参数保存之后，加载新线程的寄存器参数。

5.3.4.3 线程池

一个线程的状态一般有就绪，运行，睡眠，退出几种状态，并且从调度器的角度看，每个线程应该都需要有一个独一无二的 ID 用以区分它和其他线程：

```
pub enum Status {  
    Ready,  
    Running(Tid),  
    Sleeping,  
    Exited(ExitCode),  
}
```

将线程状态与当前的线程进行进一步封装，得到结构体 `ThreadInfo`，而线程池中会包含一个 `ThreadInfo` 的 `vector`，并且还会有一个线程调度的算法 `scheduler`。

```
struct ThreadInfo {  
    status: Status,  
    thread: Option<Box<Thread>>,  
}  
  
pub struct ThreadPool {  
    threads: Vec<Option<ThreadInfo>>,  
    scheduler: Box<dyn Scheduler>,  
}
```

作为一个线程池，需要实现调度相关的一系列操作：

表 5-1 线程池操作

操作函数	含义
New	新建一个线程池，
alloc_tid	为新线程分配一个新的 Tid
Add	添加一个可立即开始运行的线程
Acquire	从线程池中取一个线程开始运行
Retrieve	让当前线程交出 CPU 资源
tick	时钟中断时查看当前所运行线程是否要被切换
exit	退出线程

5.3.4.4 Round Robin 调度算法

时间片轮转调度算法(Round Robin)的基本思想是让每个线程在就绪队列中的等待时间与占用 CPU 的执行时间成正比例。其大致实现是：

1. 将所有的就绪线程按照 FCFS 原则，排成一个就绪队列。
2. 每次调度时将 CPU 分派（dispatch）给队首进程，让其执行一个时间片。
3. 在时钟中断时，统计比较当前线程时间片是否已经用完(tick 函数)。如用完，

则调度器（scheduler）暂停当前进程的执行，将其送到就绪队列的末尾（pop 旧线程），并通过切换执行就绪队列的队首进程（push 新线程）；如没用完，则线程继续使用。

```
pub trait Scheduler {  
    fn push(&mut self, tid: Tid);           //把 Tid 线程放入就绪队列  
    fn pop(&mut self) -> Option<Tid>;      //从就绪队列取出线程  
    fn tick(&mut self) -> bool;           //时钟 tick（代表时间片）处理  
    fn exit(&mut self, tid: Tid);          //线程退出  
}
```

5.3.5 用户进程

5.3.5.1 编写用户程序

与创建内核的操作类似，首先需要建立用户程序工程：

```
$ mkdir usr; cd usr  
$ cargo new rust --bin  
$ rm usr/rust/src/main.rs
```

因为用户程序正常运行时是在用户态执行，而在 U 态下只能通过 `ecall` 指令触发异常来发出系统服务请求，此时 CPU 进入内核态（S 态），操作系统通过中断服务例程收到请求，根据相应的偏移量跳转到指定地址，最后返回 U 态的用户程序。

所以用户态需要和内核态约定系统调用，这里实现两个简单的系统调用：1. 在屏幕输出一个字符，系统调用号为 64；2. 退出用户进程，系统调用号为 97。

约定完成后用户在调用 `syscall` 时，内核就需要根据不同的调用号进行相应处理，用户部分的调用如下：

```
enum SyscallId {  
    Write = 64,  
    Exit = 93,  
}  
  
#[inline(always)]  
fn sys_call(syscall_id: SyscallId, arg0: usize, arg1: usize, arg2: usize, arg3: usize,
```

```

) -> i64 {
    let id = syscall_id as usize;
    let mut ret: i64;
    unsafe {
        asm!(
            "ecall"
            : "{x10}"(ret)
            : "{x17}"(id), "{x10}"(arg0), "{x11}"(arg1), "{x12}"(arg2), "{x13}"(arg
3)
            : "memory"
            : "volatile"

        );
    }
    ret
}

pub fn sys_write(ch: u8) -> i64 {
    sys_call(SyscallId::Write, ch as usize, 0, 0, 0)
}

```

在这个代码块中定义了一个 `enum: SyscallId`, 里面有两个值 `Write = 64`, `Exit = 93`, 这即是刚刚约定的具体实现, 在 `sys_call` 使用了内联汇编, 他的第一个输入参数是系统调用号, 这个值被赋予给了 `id`, 最终 `id` 传递给了寄存器 `x17`, 伴随着 `ecall` 以参数的形式传递到内核态, 以进行相应服务。

在具体使用中以 `Write` 为例, 首先构建一个函数 `sys_write`, 给函数有一个参数 `char` 类型变量, 这个函数调用 `sys_call`, 将 `Write` 的系统调用号传递到第一个参数, 将 `char` 类型变量传递到第二个参数, 最终内核要依据这两个参数完成在屏幕中打印这个变量的动作。

5.3.5.2 内核态增加系统调用支持

在内核中，已经拥有了一个处理一些中断异常的函数，他目前包括了对 S 态下异常中断的处理，定时器中断的处理等等，但是还缺少用户态中断的支持，因此首先需要在总的处理函数中，添加这种用户态中断情况，并设置该情况下，中断服务例程的地址。

```
pub fn rust_trap(tf: &mut TrapFrame) {  
    match tf.scause.cause() {  
        ...  
        Trap::Exception(Exception::UserEnvCall) => syscall(tf),  
        ...  
    }  
}
```

在 rust_trap 中捕捉中断原因加入 UserEnvCall，并设置跳转到 syscall 进行下一步处理。

```
pub fn syscall(id: usize, args: [usize; 3], tf: &mut TrapFrame) -> isize {  
    match id {  
        SYS_WRITE => {  
            print!("{}", args[0] as u8 as char);  
            0  
        },  
        SYS_EXIT => {  
            sys_exit(args[0]);  
            0  
        },  
        _ => {  
            panic!("unknown syscall id {}", id);  
        },  
    }  
}
```

在 `syscall` 函数中，接收到 `id` 和三个寄存器参数，这与用户态中的系统调用是相符的：`"{x17}"(id), "{x10}"(arg0), "{x11}"(arg1), "{x12}"(arg2), "{x13}"(arg3)`，根据 `id` 号不同执行不同的程序，对于 `Write` 函数而言，打印 `args[0]`，也就是传入的字符变量即可。至此简单的系统调用支持与用户态程序添加完毕。

5.4 本章小结

本章针对实验中软件的工作进行了三部分的介绍，第一部分是对解压缩模块的 `LZ77` 及 `Huffman` 编码做了分析；第二部分对实验过程中使用的 `bootloader` 即 `BBL` 做了详细的剖析，着重阐述了使用软件模拟 `TLB` 替换的过程；第三部分对实验中移植的 `rCore` 系统进行了详细分析，分别介绍了最小化内核、中断管理、虚拟内存管理、线程切换、用户进程的功能。

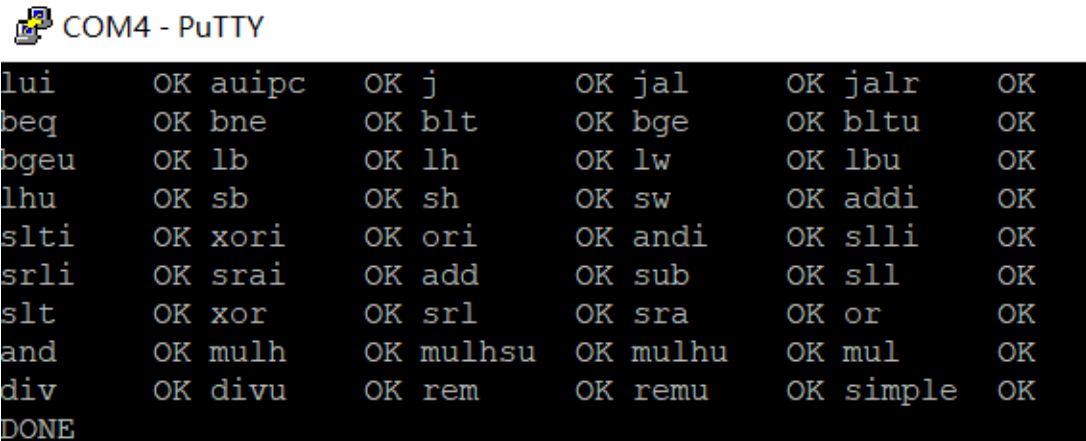
第 6 章 实验结果与展望

6.1 RISC-V 指令测试

在系统构建之前,首先要保证 RISC-V 指令执行政策,这部分程序的预期效果为,通过指令测试后将该指令的名称输出并打印 OK,若测试未通过则打印 ERROR。测试在 Modelsim 仿真环境和 FPGA 开发板的效果分别如图 6-1、图 6-2 所示:

```
# lui      OK auipc   OK j        OK jal      OK jalr     OK
# beq      OK bne     OK blt       OK bge      OK bltu     OK
# bgeu     OK lb      OK lh       OK lw       OK lbu      OK
# lhu      OK sb      OK sh       OK sw       OK addi     OK
# slti     OK xori    OK ori      OK andi     OK slli     OK
# srli     OK srai    OK add      OK sub      OK sll      OK
# slt      OK xor     OK srl      OK sra      OK or       OK
# and      OK mulh    OK mulhsu   OK mulhu    OK mul      OK
# div      OK divu    OK rem      OK remu     OK simple   OK
# DONE
```

图6-1 Modelsim环境下RISC-V指令测试结果



```
COM4 - PuTTY
lui      OK auipc   OK j        OK jal      OK jalr     OK
beq      OK bne     OK blt       OK bge      OK bltu     OK
bgeu     OK lb      OK lh       OK lw       OK lbu      OK
lhu      OK sb      OK sh       OK sw       OK addi     OK
slti     OK xori    OK ori      OK andi     OK slli     OK
srli     OK srai    OK add      OK sub      OK sll      OK
slt      OK xor     OK srl      OK sra      OK or       OK
and      OK mulh    OK mulhsu   OK mulhu    OK mul      OK
div      OK divu    OK rem      OK remu     OK simple   OK
DONE
```

图6-2 FPGA开发板环境下RISC-V指令测试结果

6.2 rCore 与用户程序

6.2.1 各部分功能测试

在完成实验过程中,对各部分进行功能测试是十分必要的,这有助于及时发现

系统的错误，方便问题的定位，下面将介绍几个在搭建操作系统中使用到的测试用例。

6.2.1.1 异常处理及写保护测试

以下代码试图向一个只读的地址空间写入一个数据，预期是系统会报写错误，然后被异常处理函数捕捉，如下图所示：

```
extern "C" {  
    fn srodata();  
}  
unsafe {  
    let ptr = srodata as usize as *mut u8;  
    *ptr = 0xab;  
}
```

```
++++ setup memory!      ++++  
Exception(StorePageFault) @ 0xc0422000 instruction = 0xc0408e1c  
panicked at 'page fault', src/interrupt.rs:36:5
```

图 6-3 异常处理及写保护测试结果

6.2.1.2 物理内存分配测试

以下代码测试物理内存的正确性，首先分配 3 个大小为 4KB 的内存，然后回收第二个分配的内存，然后再分配两个大小为 4KB 的内存。

```
println!("alloc {:x?}", alloc_frame());  
let f = alloc_frame();  
println!("alloc {:x?}", f);  
println!("alloc {:x?}", alloc_frame());  
println!("dealloc {:x?}", f);  
dealloc_frame(f.unwrap());  
println!("alloc {:x?}", alloc_frame());  
println!("alloc {:x?}", alloc_frame());
```

```
alloc Some(Frame(PhysAddr(80477000)))
alloc Some(Frame(PhysAddr(80478000)))
alloc Some(Frame(PhysAddr(80479000)))
dealloc Some(Frame(PhysAddr(80478000)))
alloc Some(Frame(PhysAddr(80478000)))
alloc Some(Frame(PhysAddr(8047a000)))
```

图 6-4 物理内存分配测试

从测试结果可以看到，分配的第二个物理内存 0x80478000~0x804790000 被回收了，然后有被重新分配了，符合实验预期。

6.2.1.3 页表映射测试

由于系统内存位于 0x8000_0000~0x8080_0000，所以直接访问 0xc040_0000 是不正确的，但是由于进行了页表映射，0xc040_0000 被映射到了 0x8040_0000，所以按照预期，访问虚拟地址 0xc0400000 可以得到相应的数据。

```
let ptr = 0xc0400000 as *const u32;
let value = unsafe { ptr.read() };
println!("addr: {:?}, value: {:#x}", ptr, value);
```

进行反汇编可以看到位于 0xc040_0000 的值是 0xc042_0137:

```
Disassembly of section .text:
c0400000 <_start>:
c0400000: c0431137
```

运行系统得到结果如下图所示，符合实验预期：

```
switch satp from 0x80080427 to 0x80080474
++++ setup memory!      ++++
addr: 0xc0400000, value: 0xc0431137
```

图 6-5 页表映射测试

6.2.2 整体功能测试

rCore 预期运行结果为，先打印“RUNNING”，然后设备进行中断初始化，探测可用物理内存，初始化虚拟内存后，切换 satp 页基址项，初始化线程(5 个)，按次序切换，每当切换到一个线程时，打印 10 个当前线程的编号，然后结束当前线程，切换到下一个线程，依次类推到最后一个线程，在 Qemu 和 Modelsim 环境下两者的运行效果分别如图 6-3、图 6-4 所示：

```

rCore_step_by_step os is running!
++++setup interrupt !++++
free physical memory ppn = [0x80474, 0x80540)
switch satp from 0x80080427 to 0x80080474
++++ setup memory!      ++++
it really a executable!
++++ setup process!     ++++

>>>> will switch_to thread 0 in idle_main!
begin of thread 0
0000000000
end of thread 0
thread 0 exited, exit code = 0

<<<< switch_back to idle in idle_main!

>>>> will switch_to thread 1 in idle_main!
begin of thread 1
1111111111
end of thread 1
thread 1 exited, exit code = 0

```

图 6-6 Qemu环境下rCore运行示意图

```

# <<<< switch_back to idle in idle_main!
#
# >>>> will switch_to thread 2 in idle_main!
# begin of thread 2
# 2222222222
# end of thread 2
# thread 2 exited, exit code = 0
#
# <<<< switch_back to idle in idle_main!
#
# >>>> will switch_to thread 3 in idle_main!
# begin of thread 3
# 3333333333
# end of thread 3
# thread 3 exited, exit code = 0

```

图6-7 Modelsim环境下rCore运行示意图

总结

本实验以系统协同设计为目的，分别从硬件和软件两个方面入手，综合运用本科期间学习的知识，探索技术发展趋势，在经过广泛调研和学习的基础上，设计出基于目前最前沿的开源指令集架构——RISC-V 的 CPU 的微型硬件系统，该系统的硬件部分不仅包括一个功能完善且支持多个特权级模式的 CPU，还集成了 ROM、SDRAM 及 UART 的控制器。而软件部分，则通过更改 BBL 和 rCore 操作系统的底层硬件支持，实现了格式化输出，中断处理，虚拟内存管理，线程切换及用户进程的多种功能，总体上实现了对该操作系统的移植。

实验不仅能在开发板环境下可以对系统功能进行测试，并且可以通过仿真工具验证系统的正确性，在实验过程中，综合使用了多种手段对系统所出现的问题和错误进行调试定位，如 Qemu，Modelsim，逻辑分析仪等。

本实验通过重写 UART 控制器，使用软件模拟 TLB 替换以及使用解压缩模块减小文件体积的手段克服了硬件资源的限制。在软件开发过程中，针对 BBL 对 TLB 的支持进行了较大幅度的修改，并且可以在虚拟内存的环境下实现系统的引导，用户程序的转载，并以此展开多个模块的操作系统实验，是一个比较完善的系统模型。

综上所述，本工程通过软硬件协同设计，搭建起了一个功能完整的基于 RISC-V 架构的 SOC 硬件系统，并在 SOC 上成功运行起了以 Rust 语言为基础构建的 rCore 操作题，基本实现了预期目标。

参考文献

- [1] 王月涛. 基于 FPGA 的嵌入式实时操作系统及 TCP/IP 移植[D].电子科技大学,2008.
- [2] 张正威. 基于 AHB 协议的 MIPS 内核总线结构优化及 FPGA 逻辑电路实现[D].西安电子科技大学,2018.
- [3] RISC-V Foundation. About the RISC-V Foundation [EB/OL]. <https://riscv.org/risc-v-foundation>. 2020-05-22
- [4] David Patterson. Andrew Waterman. The RISC-V Reader[M]. America:2017-9.
- [5] Abel Avram. Interview on Rust, a Systems Programming Language Developed by Mozilla[EB/OL]. <https://www.infoq.com/news/2012/08/Interview-Rust>. 2012-08-03/2019-05-22
- [6] 百 度 百 科 . Rust 语 言 [EB/OL]. [https://baike.baidu.com/item/Rust语言/9502634?fr=aladdin#reference-\[3\]-7614357-wrap](https://baike.baidu.com/item/Rust语言/9502634?fr=aladdin#reference-[3]-7614357-wrap). 2020-05-01
- [7] Rust程序设计语言 [EB/OL]. <https://www.rust-lang.org/zh-CN>
- [8] 朱威浦. 基于rust语言和RISC-V平台的操作系统设计与改进[D]. .北京: 北京理工大学, 2019.
- [9] Rust语言介绍[EB/OL]. <https://12101111.github.io/rustyu-yan-jie-shao/>
- [10] 邹楚雄. 交叉编译和交叉调试工具的研究与实现[D].电子科技大学,2006.
- [11] 雷凯翔. OpenRISCV[EB/OL]. <https://github.com/shyoshyo/openriscv>
- [12] 维基百科. UART [EB/OL]. <https://zh.wikipedia.org/wiki/UART>
- [13] 维基百科.LZ77 [EB/OL]. <https://zh.wikipedia.org/wiki/LZ77%E4%B8%8ELZ78>
- [14]

附 录

注：此处无需更改。
阅后删除此文本框。

附录相关内容…

附录是毕业设计（论文）主体的补充项目，为了体现整篇文章的完整性，写入正文又可能有损于论文的条理性、逻辑性和精炼性，这些材料可以写入附录段，但对于每一篇文章并不是必须的。附录依次用大写正体英文字母 A、B、C……编序号，如附录 A、附录 B。阅后删除此段。

附录正文样式与文章正文相同：宋体、小四；行距：22 磅；间距段前段后均为 0 行。阅后删除此段。

致 谢

值此论文完成之际，首先向我的导师……