

北京理工大学

本科生毕业设计(论文)

基于 Rust 语言和 RISC-V 平台的操作系统
设计和改进

Operating System Design and Improvements based on Rust and
RISC-V Platform

学 院： 计算机学院

专 业： 物联网工程

学生姓名： 朱威浦

学 号： 1120151869

指导教师： 陆慧梅

2019 年 5 月 24 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 Rust 语言和 RISC-V 平台的操作系统 设计和改进

摘要

本文介绍了如今最新的 RISC-V 指令集架构和 Rust 语言的特点和优势，以实验作为主线，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 `picorv32` 的基础上添加串口、数码管等多种外设，搭建了一个功能齐全的 SoC。该 SoC 不仅能在仿真环境下运行，更能通过 STEP-MAX10 FPGA 开发板在真实环境下对搭建的 SoC 进行功能测试。在软件方面，为进一步体现硬件平台的可用性，实验将 rCore 教学操作系统裁剪为嵌入式系统，综合应用汇编、C 语言、Rust 语言对其进行移植，最终实现格式化输出、中断处理、ELF 解析等功能，并顺利运行于硬件平台上。本实验说明在诸如 STEP-MAX10 系列资源有限的开发板上进行系统的开发具有一定可行性。

关键词：RISC-V；Rust；`picorv32`；STEP MAX10；嵌入式系统

Operating System Design and Improvements based on Rust and RISC-V Platform

Abstract

This paper introduces the features and advantages of the newest RISC-V instruction set architecture and Rust language, connecting the professional courses with experiments as the main line from the aspects of hardware and software. In terms of hardware, a fully functional SoC is built by adding a variety of peripherals such as serial port and segment on the basis of RISC-V soft core ——picorv32. The SoC can not only run in the simulation environment, but also test the functions of the SoC built in the STEP-MAX10 FPGA board in the real environment. In terms of software, in order to further reflect the usability of the hardware platform, the experiment tailors the rCore teaching operating system into an embedded system and transplants it with Assembly language, C language and Rust language comprehensively to realize functions such as formatted output, interrupt handle and ELF programme analysis, and finally runs smoothly on the hardware platform. The experiment shows that it is feasible to develop the system in resource - limited development boards like STEP-MAX10 series.

Key Words: RISC-V; Rust; picorv32; STEP MAX10; embedded system

目 录

摘要.....	I
Abstract	II
第 1 章 绪论	1
1.1 背景	1
1.1.1 RISC-V 的发展	1
1.1.2 Rust 语言介绍	1
1.1.3 FPGA 的应用	1
1.2 研究目标与研究意义	2
1.2.1 研究目标.....	2
1.2.2 研究意义.....	2
1.3 论文结构	2
第 2 章 研究现状与技术分析	4
2.1 RISC-V 软核 picorv32	4
2.1.1 picorv32 概况	4
2.1.2 picorv32 的性能分析	4
2.1.3 picorv32 SRAM 接口	7
2.1.4 picorv32 AXI-Lite 接口	8
2.1.5 picorv32 PCPI 接口	8
2.2 Rust 语言的特点与优势	9
2.2.1 Rust 语言的优势分析.....	9
2.2.2 Rust 语言与 C++语言的比较	10
2.3 rCore 教学操作系统	11
2.4 本章小结	11
第 3 章 环境搭建与工具介绍	12
3.1 实验硬件平台	12
3.1.1 STEP FPGA 开发板介绍	12
3.1.2 开发环境搭建平台	13

3.2 Verilog 综合与仿真工具	13
3.2.1 Quartus.....	13
3.2.2 iverilog.....	14
3.2.3 gtkwave.....	14
3.3 RISC-V 交叉编译工具链.....	14
3.3.1 交叉编译的难点分析	14
3.3.2 RISC-V 交叉编译工具链介绍	15
3.3.3 RISC-V 交叉编译工具链的安装	16
3.3.4 交叉编译工具链的测试	18
3.4 Rust 工具链的安装.....	19
3.5 本章小结	21
第 4 章 基于 picorv32 软核的 SoC 硬件平台搭建	22
4.1 SoC 模块组成与接口设计	22
4.1.1 SoC 模块设计	22
4.1.2 串口设备的实现	22
4.1.3 地址空间的设计与分配	23
4.2 picorv32 IP 核的封装	23
4.3 picorv32 CPU 参数的配置.....	25
4.3.1 指令集模块的配置.....	25
4.3.2 中断支持的配置	25
4.4 添加外设	26
4.4.1 ROM	26
4.4.2 RAM	27
4.4.3 串口	27
4.4.4 数码管	28
4.4.5 GPIO	28
4.4.6 PLL	28
4.5 设备连接与地址分配	28
4.6 顶层模块的设计	28

4.7 引脚与约束文件	29
4.8 编译下板	29
4.9 资源使用情况分析	29
4.10 本章小结	29
第 5 章 SoC 硬件平台功能的测试	31
5.1 外设基本功能的测试	31
5.1.1 ROM 功能的测试	31
5.1.2 UART 功能的测试	31
5.2 RISC-V 官方测试的移植	32
5.3 Intel Hex 格式文件的生成方式	33
5.4 调试的方法	34
5.4.1 仿真环境下的调试	34
5.4.2 真实环境下的调试	35
5.5 本章小结	36
第 6 章 引导程序的设计	37
6.1 picorv32 的特权极架构	37
6.1.1 CSR 寄存器	37
6.1.2 特权极指令	38
6.1.3 自定义指令的使用	39
6.1.4 中断信号	40
6.2 引导程序功能分析	40
6.3 引导程序运行流程	41
6.4 中断处理程序	42
6.4.1 中断处理函数	43
6.5 串口驱动程序	44
6.6 SBI 设计	46
6.7 本章小结	47
第 7 章 rCore 操作系统的分析与移植	48
7.1 移植 rCore 的可行性分析	48

7.2 最小化内核	48
7.2.1 新建工程.....	49
7.2.2 更改编译的目标架构	49
7.2.3 去除标准库的引用	51
7.2.4 堆栈设置.....	52
7.2.5 操作系统 main 函数	53
7.3 SBI 函数封装	53
7.3.1 系统调用库的创建.....	53
7.3.2 库导入	55
7.4 格式化输出	55
7.4.1 IO 模块	55
7.4.2 Write trait 的实现.....	55
7.4.3 println! 宏	57
7.5 ELF 解析	58
7.6 用户程序的编写	58
7.7 程序的链接与加载	59
7.7.1 引导程序的内存分配	59
7.7.2 操作系统的链接	60
7.7.3 用户程序的链接	61
7.7.4 地址空间布局	62
7.8 本章小结	62
第 8 章 实验成果与展望	63
8.1 实验成果展示	63
8.2 未来展望	65
结 论	67
参考文献	68
附 录	70
附录 A.....	70
附录 B.....	71

附录 C.....	73
致 谢.....	75

第1章 绪论

1.1 背景

1.1.1 RISC-V 的发展

RISC-V 是一种全新、简单、免费的开源指令集架构，于 2010 年发源于加州大学伯克利分校。自发布以来，RISC-V 凭借其优雅简洁、高效模块化的特性越来越受到学术界和工业界的重视和欢迎。2015 年，RISC-V 基金会成立，并开始接管 RISC-V 标准指令和架构的维护工作。RISC-V 基金会由 235 名成员组成，旨在建立第一个开放、协作的软件和硬件创新者社区，为创新提供动力^[1]。如今，RISC-V 生态链正在不断发展和完善，未来将具备更大的潜能和价值。

1.1.2 Rust 语言介绍

Rust 是一门新兴的系统编程语言，最初由 Mozilla 研究院的 Graydon Hoare 设计创造，致力于成为优雅解决高并发和高安全性系统问题的编程语言^[2]。Rust 是可以支持函数式、命令式以及泛型等编程范式的多范式语言，其在语法风格和 C++ 类似，但是能够在保证性能的同时提供更好的内存安全^[3]。

Rust 语言属于静态编译式语言，支持静态编译和动态编译，通过所有权和生命周期机制，使得使用 Rust 语言编写的程序能够避免空指针、内存越界、段错误、数据竞争等一系列问题。Rust 编译器采用 LLVM 作为后端，通过使用 Rust 语言编写并完成自举，说明其实现具有充分的完备性。

1.1.3 FPGA 的应用

FPGA（Field-Programmable Gate Array），即现场可编程门阵列。开发人员通过硬件描述语言所描述的硬件设计，经过 EDA 软件的综合与布局，可以烧录至 FPGA 中对其进行快速地验证，是现代 IC 设计验证的主流技术。

本实验通过将自行设计的硬件 SoC 综合烧录至 STEP-MAX FPGA 开发板中，以搭建基本的硬件系统并对其进行验证。

1.2 研究目标与研究意义

1.2.1 研究目标

本实验的研究目标为在 STEP-MAX10 FPGA 开发板中搭建基于 RISC-V 软核 picorv32 的硬件系统，通过添加各种外部设备控制器，实现对多种外部设备的控制以充分利用板上的硬件资源。在硬件系统搭建完成后，通过裁剪和移植 rCore 操作系统，将其编译链接为兼容 RISC-V 处理器和特定 SoC 的系统，进一步验证硬件平台的可用性。

1.2.2 研究意义

本实验的研究意义在于能够综合应用一系列课程知识，以实验作为主线，将各课程内容进行衔接，提升对计算机体系结构的认知能力以及系统工程能力。在硬件开发方面，能够应用《数字电子技术》、《计算机硬件系统设计》、《计算机组成原理》、《计算机体系结构》、《微机接口技术》课程基础知识，针对如今最为前沿的 RISC-V 架构进行学习，在 FPGA 中搭建出自行设计的硬件系统。在软件开发方面，综合应用《操作系统》、《编译原理与设计》课程知识，对现有硬件系统进行组织，并实现对 rCore 操作系统的移植。

如今随着 FPGA 开发板不断增加的硬件资源，其价格的增长成为阻碍学生自行学习硬件系统知识的一大阻力。本实验通过在 STEP MAX10 系列开发板中进行硬件系统实验，目的在于探究在资源受限的情况上，能够将硬件系统搭建至何种水平，能否胜任大学课程对于学生能力培养的基本要求。

1.3 论文结构

本论文第一章介绍了 RISC-V 和 Rust 语言的背景知识，阐述 FPGA 的应用方式及本实验的研究目标。第二章介绍与本论文相关的技术和现有成果，

包括 `picorv32` 的分析，Rust 语言的优势及 `rCore` 的研究。第三章介绍本实验所依赖的硬件环境、RISC-V 和 Rust 工具链的安装及使用方法。第四章介绍本实验 SoC 硬件平台的设计和搭建过程。第五章则说明对硬件平台测试和调试的手段。第六章介绍 `picorv32` 的特权级架构，展示了引导程序的设计和运行过程，以及链接操作系统和用户程序的方式。第七章说明 `rCore` 操作系统移植的可行性，展示了操作系统从无到有，从最小化内核到格式化输出再到解析用户程序的构建过程。第八章则对实验结果做出展示，并对未来的工作给出的展望和建议。

第2章 研究现状与技术分析

2.1 RISC-V 软核 picorv32

2.1.1 picorv32 概况

picorv32 是由著名 IC 工程师 Clifford 开发设计的 CPU，其主要专注点在于小体积，高频率与低功率^[4]。该项目源码目前开源于 GitHub，用户可遵项 ISC 开源协议对其进行使用。

picorv32 是一个用纯 Verilog 语言实现的 RISC-V 软核，其优点是体积小，占用资源少，运行频率高，支持多种模块的 RISC-V 指令集，并且能够通过预留接口对其功能进行拓展。

picorv32 访存接口包括 SRAM 接口，AXI-Lite 接口，WISHBONE 接口等。所有接口 CPU 均为以原生访存接口 CPU 作为核心，再增加相应接口转换模块生成。使用者可以通过配置参数实现对 CPU 部件和功能的选择。

2.1.2 picorv32 的性能分析

（1）CPU 时钟周期数

picorv32 中执行每条指令所需的时钟周期数接近 4。表 2-1 为具体每条指令执行所需的周期数^[4]。

表 2-1 picorv32 指令 CPI

Instruction	CPI	CPI(SP)
direct jump (jal)	3	3
ALU reg + immediate	3	3
ALU reg + reg	3	4
branch (not taken)	3	4
memory load	5	5
memory store	5	6
branch (taken)	5	6

indirect jump (jalr)	6	6
shift operations	4~14	4~15
MUL	40	40
MULH[SU U]	72	72
DIV[U]/REM	72	72

CPI---执行每条指令的周期数，开启双口读寄存器。

CPI(SP)---执行每条指令的周期数，单口读寄存器。

picorv32 在项目自搭建的嵌入式 SoC 系统中运行 Dhrystone benchmark 基准测试程序结果为 0.516 DMIPS/MHz (908 Dhrystones/Second/MHz), Dhrystone benchmark 基准测试程序平均 CPI 为 4.100^[4]。

（2）CPU 运行频率

通过在 Xilinx 系列 FPGA 芯片上布局布线，分析得到 picorv32 所能够达到的最高频率如表 2-2 所示^[4]。

表 2-2 picorv32 运行频率

Device	Device	Speedgrade	Clock Period (Freq.)
Xilinx Kintex-7T	xc7k70t-fbg676-2	-2	2.4 ns (416 MHz)
Xilinx Kintex-7T	xc7k70t-fbg676-3	-3	2.2 ns (454 MHz)
Xilinx Virtex-7T	xc7v585t-ffg1761-2	-2	2.3 ns (434 MHz)
Xilinx Virtex-7T	xc7v585t-ffg1761-3	-3	2.2 ns (454 MHz)
Xilinx Kintex UltraScale	xcku035-fbva676-2-e	-2	2.0 ns (500 MHz)
Xilinx Kintex	xcku035-fbva676-3-e	-3	1.8 ns (555 MHz)

UltraScale			
Xilinx Virtex UltraScale	xcvu065-ffvc1517-2-e	-2	2.1 ns (476 MHz)
Xilinx Virtex UltraScale	xcvu065-ffvc1517-3-e	-3	2.0 ns (500 MHz)
Xilinx Kintex UltraScale+	xcku3p-ffva676-2-e	-2	1.4 ns (714 MHz)
Xilinx Kintex UltraScale+	xcku3p-ffva676-3-e	-3	1.3 ns (769 MHz)
Xilinx Virtex UltraScale+	xcvu3p-ffvc1517-2-e	-2	1.5 ns (666 MHz)
Xilinx Virtex UltraScale+	xcvu3p-ffvc1517-3-e	-3	1.4 ns (714 MHz)

（3）CPU 体积与占用资源

通过在 Xilinx 7-Series FPGA 上布局布线得到 picorv3 所消耗的资源如表 2-3 所示^[4]。

表 2-3 picorv32 占用资源

Core Variant	Slice LUTs	LUTs as Memory	Slice Registers
picorv32 (small)	716	48	442
picorv32 (regular)	917	48	583
picorv32 (large)	2019	88	1085

- picorv32 (small)：不包含 counter 指令、two-stage shifts、mem_radata

锁存和未定义指令异常、地址对齐异常配置的 `picorv32`。

- `picorv32 (regular)` : `picorv32` 的默认配置，即所有的 `parameter` 参数使用默认值。

- `picorv32 (large)` : 包含 `PCPI`、`IRQ`、`MUL`、`DIV`、`COMPRESSED_ISA` 的 `picorv32`。

2.1.3 `picorv32` SRAM 接口

(1) SRAM 接口信号定义如表 2-4 所示^[4]。

表 2-4 `picorv32`SRAM 接口信号

方向	宽度	名称
output	1	<code>mem_valid</code>
output	1	<code>mem_instr</code>
input	1	<code>mem_ready</code>
output	32	<code>mem_addr</code>
output	32	<code>mem_wdata</code>
output	4	<code>mem_wstrb</code>
input	32	<code>mem_rdata</code>

(2) SRAM 接口时序

SRAM 接口时序为 CPU 通过 `mem_valid` 启动数据传输。`mem_valid` 信号一直保持高直到接收方将 `mem_ready` 信号置高，其余信号输出在 `mem_valid` 为高期间保持稳定。如果是指令访存，则将 `mem_instr` 置高。

在读取数据时，`mem_wstrb` 值为 0，`mem_wdata` 未使用。访存地址为 `mem_addr`，当 `mem_ready` 置高时，`mem_rdata` 的数据有效。CPU 可在给出 `mem_valid` 的在同一周期内，读取返回的数据。

在写入数据时，`mem_wstrb` 为字节选择信号，`mem_rdata` 不使用。CPU 将 `mem_wdata` 的数据写入地址 `mem_addr`。写入成功后，内存将 `mem_ready` 置高。

2.1.4 picorv32 AXI-Lite 接口

picorv32_axi_adapter 模块定义于 picorv32.v 文件中，遵循 AMBA AXI-Lite 接口信号定义和使用标准，能够将 picorv32 原生 SRAM 访存接口转换为 AXI-Lite 接口。picorv32.v 文件将 picorv32 模块与 picorv32_axi_adapter 模块封装为 picorv32_axi 模块。

2.1.5 picorv32 PCPI 接口

PCPI 接口为 picorv32 的拓展接口。出于对小体积的追求，picorv32 尽可能缩减了 CPU 的功能，同时通过提供拓展接口的形式使得用户能自行添加新功能。picorv32 使用 Pico 协处理器接口(PCPI) 来实现用户自定义指令（分支指令类型除外）。

PCPI 接口信号定义如表 2-5 所示^[4]。

表 2-5 picorv32 PCPI 接口信号

方向	宽度	名称
output	1	pcpi_valid
output	32	pcpi_insn
output	32	pcpi_rs1
output	32	pcpi_rs2
input	1	pcpi_wr
input	32	pcpi_rd
input	1	pcpi_wait
input	1	pcpi_ready

当遇到 CPU 内部不支持的指令时，若 ENABEL_PCPI 参数设置为 1，则开启 Pico 协处理器模式。CPU 将 pcpi_valid 信号置高，同时将该非法指令通过 pcpi_insn 输出，CPU 将在内部对指令进行译码，并读取 rs1 和 rs2 寄存器的值输出至 pcpi_rs1 和 pcpi_rs2 上。当协处理器执行指令完成时，将 pcpi_ready 信号置高，若需写入 rd 寄存器，则将结果写入 pcpi_rd，并将 pcpi_wr 信号置高，然后 CPU 将解析指令的 rd 字段，并将 pcpi_rd 的值从写

入对应的寄存器。当协处理器在 16 个时钟周期内没有给出 `pcpi_ready` 信号时，将触发非法指令异常，CPU 将跳转至相应的中断处理程序。若协处理器需要多于 16 个时钟周期，则应提前将 `pcpi_wait` 信号置高，以避免引发非法指令异常。

2.2 Rust 语言的特点与优势

Rust 语言是一个全新开发的高级语言。设计人员在开发过程中，借鉴过去近 30 年的语言理论研究和实际软件工程的经验进行设计。为了追求“更好的设计和实现”的目标，设计者们大胆舍弃历史包袱，不为向前兼容而放弃更好的设计或实现。

2.2.1 Rust 语言的优势分析

（1）Rust 语言是接近底层的语言，具有接近实时且高性能的优点。在高级语言中，若需实现接近实时的性能，便不能存在垃圾回收机制。典型的例子为 `java`、`python` 等语言因为垃圾回收机制导致其运行时的延迟。

（2）Rust 语言是多范式编程语言，能够为开发人员提供一系列基础的组件。此外，Rust 还拥有强大的管理系统 `Cargo` 以及中心化的库管理 `crates.io`，更加提升包管理的方便性。C 语言是最贴近底层的高级语言，能够轻易达到高性能的目标。但是，C 语言是纯粹的过程式编程语言，开发者在编写程序时缺少数据结构基础设施，将严重地影响开发效率。

（3）Rust 语言具有足够的抽象能力。从开发效率和可读可维护性的层面上来说，高级语言必须具有足够的抽象能力，并希望抽象不带来额外的运行时开销。Rust 语言中的 `Generics` 和 `Trait` 特性便体现了零开销抽象的设计原则。

（4）Rust 语言具有所有权和生命周期。所有权系统是 Rust 最独特的功能，其令 Rust 无需垃圾回收即可保障内存安全^[5]。Rust 程序中的每一个值都有一个被称为其所有者的变量，每个值有且只有一个所有者，当所有者即变量离开作用域，该值将被丢弃。每一个引用和指针都有一个生命周期，对象不允许在同一作用范围内有两个可变引用。编译器在编译时将对所有权和生命

周期进行检测，凡是违背基本原则的引用都会被禁止被使用。稳健的大程序则意味着程序要在进生产环境之前尽量消灭错误，这就意味着高级语言需要具备一个静态类型，最好同时是强类型的语言，使得编译器能够尽早发现问题。Rust 编译器严格的类型检查以及对所有权机制的设计真正实现了 Rust 官方所声称的"prevents nearly all segfaults, and guarantees thread safety"特性。

2.2.2 Rust 语言与 C++语言的比较

C++语言是在项目编程领域中应用最为广泛的语言，其应用场景多为编写接近实时高性能，稳健，并有足够开发效率的大程序。从整体上看，C++语言是一个高性能的静态强类型多范式语言。但在具备以上优点的同时，C++语言也存在这许多的问题。C++语言开发的目标之一是与 C 语言兼容，为了避免 Python 2 和 Python 3 的故事，甚至是 IA64 和 x86_64 的错误，设计人员不得不保证 C++一直保持向前兼容。这是由于历史原因与设计人员当时抉择所看重的东西所导致的^[6]。

Rust 语言除具备 C++语言具备的优势外，其所有权和生命周期机制保证了内存的安全性。举例说明此问题，C++中的 String 类 `c_str()`方法用以将 String 类型转换为 char 型数组。`c_str()`函数返回一个指向正规 C 字符串的指针，内容与 String 相同，但是 `c_str()`返回的是仅是一个临时指针，若未使用 `strcpy()`函数将其拷贝而是直接使用，会导致悬垂指针的严重后果。但是这种问题在 Rust 语言中能够被编译器识别并无法编译通过。正是源于这一特性，使得原本在 C++里无法使用的优化做法变得具有可行性。比如若要将字符串分割成不同部分，并在不同的地方使用，当项目较大时，为了保障安全性，通常的做法是把分割的结果拷贝一份单独处理，这样便可避免在处理分割的字符串的时候原本的字符串已经不存在，即悬垂指针的问题。但是当我们使用 Rust 编写程序，便可舍弃拷贝的环节，直接使用原来的字符串而不用担心出现野指针的情况，生命周期的设定可以让编译器完成这一繁琐的检查，如果有任何的字符串在处理分割结果之前被使用，编译器将会检查出此种错误。

由于 Rust 没有垃圾回收的控制，且其实现接近底层。在达到同样安全性

的前提下，Rust 并不会比 C++ 慢。Rust 拥有足够多的语言特性来保证开发的效率，并且比 C++ 语言吸收了更多的现代优秀的语言特性。与 C++ 一致的零抽象消耗。杀手级的所有权和生命周期机制，加上现代语言的类型系统。总之，在语言层面上来说，Rust 语言无疑是比 C++ 优秀的一个高性能静态强类型多范式语言^[6]。

2.3 rCore 教学操作系统

rCore 是基于清华大学教学操作系统 uCore 操作系统的 Rust 移植版本，致力于使用现代编程语言，以提升操作系统的开发体验和质量，并进一步探索未来操作系统的设计实现方式^[7]。目前 rCore 具有基本的内存管理、进程管理、文件系统功能，支持多种目标机器架构，如 x86_64、RISC-V32/64、AArch64 指令集，能够在 QEMU、Labeled-RISCV、K210 开发板、树莓派 3B+ 等多种平台上运行。

本毕设实验将参考 rCore 操作系统的开发流程对其进行移植适配。

2.4 本章小结

本章首先对 picorv32 软核进行了详细的介绍，从 CPU 的性能、功耗、资源占用再到 CPU 对外的接口以及拓展实现的方式。然后对 Rust 语言进行介绍，分析其优势并以 C++ 语言作为对比说明了 Rust 语言在安全方面的优点。最后介绍了 rCore 教学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 实验硬件平台

3.1.1 STEP FPGA 开发板介绍

STEP-FPGA 开发板，即小脚丫系列 FPGA 开发板，是由国内硬件平台设计开发公司——思得普信息科技有限公司发布的小体积、高性价比开发板。本实验主要使用 STEP-MAX10 系列 FPGA 开发板作为硬件开发平台。STEP-MAX10 是基于 Altera 公司芯片开发的 FPGA 开发板，板卡集成下载器，使用 MicroUSB 数据线可完成开发板的供电与下载^[8]。其核心器件为 Altera 10M08SAM153，板载资源如表 3-1 所示^[8]。

表 3-1 STEP-MAX 开发板板载资源

LE 资源	8000 个	RGB LED	2 路
用户闪存	172KB	用户 LED	8 路
Block Memory	378Kbit	拨码开关	4 路
PLL	2 路	按键开关	4 路
硬件乘法器	24 路	用户可扩展 IO	36 个
GPIO 接口	112 个	Micro USB 接口	1 路
供电电压	3.3.V	数码管	2 位

STEP-MAX10 FPGA 开发板内部原理如图 3-1 所示^[8]。

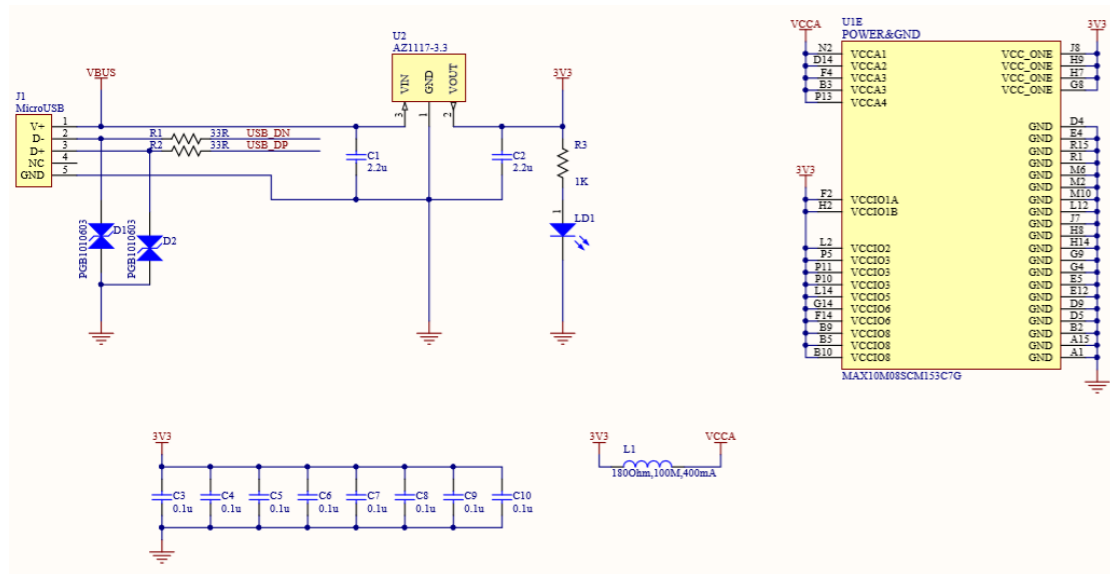


图 3-1 STEP-MAX 内部原理图

实际效果如图 3-2 所示^[8]。

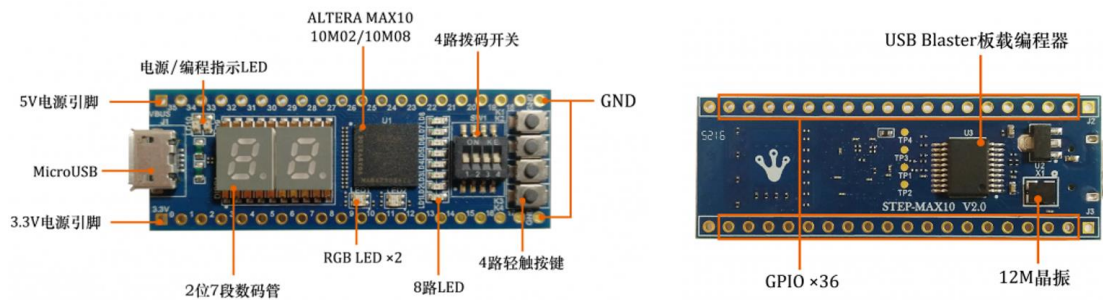


图 3-2 STEP-MAX10 实际效果图

3.1.2 开发环境搭建平台

开发环境搭建系统为 Ubuntu16.04 和 Windows10 操作系统。

3.2 Verilog 综合与仿真工具

3.2.1 Quartus

Quartus 为 Intel 公司发布的针对 Altera FPGA 芯片进行综合实现的 EDA 开发软件。本实验采用 Quartus Lite 18.1 版本，并安装在 Windows10 操作系

统中。

3.2.2 iverilog

iverilog 为可通过 Linux 终端运行的 Verilog 模拟和综合工具。虽然可直接使用 apt 安装，但由于实验中需使用到与该源软件不兼容的 Verilog 特性，故需要自行下载 iverilog 源代码编译安装。

3.2.3 gtkwave

gtkwave 是一个使用 GTK 的 WAV 文件波形察看工具，支持 Verilog VCD/EVCD 文件格式。通过在 Ubuntu16.04 终端中输入如下命令安装

```
$ sudo apt-get install gtkwave
```

3.3 RISC-V 交叉编译工具链

3.3.1 交叉编译的难点分析

编译器运行的计算机环境称为 host，运行的新程序所使用的计算机称为 target。当 host 和 target 是相同类型的计算机时，编译方式是本机编译。当 host 和 target 不同时，编译方式是交叉编译。当前大多数机器编译主要针对 x86 架构和硬件平台，因此多为本地编译。

交叉编译的难点主要体现在如下两个方面^[9]。

（1）机器特性不同

字大小：字大小决定相关数据类型的位宽，如指针、整型数据等。

大小端：分为大端和小端，决定数据和地址的对应关系。

对齐方式：不同平台读取数据的方式不同，有些只能读取或写入对齐地址中的数据，否则将会出现访存异常。因此编译器会通过补齐结构的方式来对齐变量。

默认符号类型：不同平台对于某些数据类型默认是有符号还是无符号有不同的规定，如“char”数据类型，解决方法是提供一个编译器参数，如“-funsigned-char”，以强制默认值为已知值。

（2）编译时的主机环境与目标环境不同

库支持：程序在交叉编译时，动态链接的程序必须在编译时访问相应的共享库。目标系统中的共享库需要添加到跨编译工具链中，以便程序可以针对它们进行链接。对于特定平台下运行的程序，难以找到能够契合其硬件的库支持，因此需要根据现有库进行改写或自行编写简易库支持。

由于实验所编写的软件将会运行于自行搭建的平台，即 **bare-metal** 的环境中。若直接使用本地编译的方式，程序将无法适配平台并正常运行，故无法使用本地编译环境对程序进行直接的编译。为此，在正式开始软件开发之前，需要为本地机器安装交叉编译环境，使编译出来的程序能够运行于另一种体系结构的不同目标平台之下。

3.3.2 RISC-V 交叉编译工具链介绍

riscv-gnu-toolchain：RISC-V 编译工具链，包括将源程序编译、汇编、链接为可执行文件的一系列工具。

riscv-gcc：GCC 为 **gnu compiler collection** 的缩写，即由 GNU 开发的一系列编译器集合，支持将多种高级语言编译为可执行文件的工具。**riscv-gcc** 是将高级语言程序或者 RISC-V 汇编程序编译为 RISC-V 架构 ELF 可执行程序 of 的编译器。

实验中所用到的工具如表 3-2 所示。

表 3-2 riscv-gnu 工具链程序

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 交叉编译工具链的安装

RISC-V 交叉编译工具链由 RISC-V 基金会进行维护，其源码托管于 GitHub 平台，通过下载源代码在 Ubuntu 中进行本地编译即可使用。

执行如下命令安装本地编译所需依赖项^[10]

```
$ 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
```

（1）下载 GNU 编译工具链源代码

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

（2）指定编译参数

```
$ cd riscv-gnu-toolchain
```

```
$ ./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=ilp32d
```

其中，configure 脚本的各参数可选项如下。

`--prefix=` 指定安装路径，若不指定则默认安装在当前路径。

`--with-arch=` 指定安装模块，`rv32` 代表 32 位交叉编译工具链、`rv64` 代表 64 位交叉编译工具链。RISC-V 的子模块包括 I、M、A、D、F、C，其中 G 模块为包含 I、M、A、D、F 的集合，若不指定则默认安装全部模块。

`--with-abi=` 指定 abi，RISC-V 的 abi 调用模式参考 3.3.4 节，若不指定则默认安装全部模式。

（3）编译工具链源码

```
$ make -j$(nproc)
```

执行 `make` 命令安装 `newlib` 作为 C 语言函数库的交叉编译工具链。

执行 `make linux` 命令安装 `glibc` 作为 C 语言运行库的交叉编译工具链。

安装完成后可在指定的路径中找到工具链的可执行文件。将该路径下的 `bin` 目录路径加入至 `PATH`，以便在 `Terminal` 中直接通过工具链名称执行编译工具链中的程序。

查看该 `bin` 目录下的程序，可以看到程序均具有相同的前缀。不同工具链前缀不同，其区别如下：

`riscv32-unknown-elf-gcc`：针对于 RISC-V32 架构的编译器，使用的 C 运行库为 `newlib`。

`riscv64-unknown-elf-gcc`：针对于 RISC-V64 架构的编译器，使用的 C 运行库为 `newlib`。

`riscv32-unknown-linux-gnu-gcc` 针对于 RISC-V32 架构的编译器，使用的 C 运行库为 Linux 中的标准 `glibc`。

（4）下载 `spike` 和 `riscv-pk` 源码^[11]

```
$ git submodule update --init --recursive
```

（5）导出 RISC-V 环境变量变量

```
$ export RISC_V=riscv 安装路径
```

（6）执行脚本编译

```
$ ./build-rv32ima.sh
```

安装完成后，终端出现提示信息 "RISC-V Toolchain installation completed!"

3.3.4 交叉编译工具链的测试

（1）以 Hello World 程序为例，创建 HelloWorld.c 文件，其内容如下。

```
#include<stdio.io>

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

通过如下命令编译文件。

```
$ riscv32-unknown-elf-gcc -march=rv32i -mabi -mcmmodel=medlow
```

```
HelloWorld.c -o HelloWorld
```

（2）编译参数含义如下^[12]

-march

功能：指定编译的目标架构模块。

类型：

32 位交叉编译工具链支持如下目标架构 rv32gc (g = i[mafd])

64 位交叉编译工具链支持如下目标架构 rv32(64) gc (g = i[mafd])

-mabi

功能：指定编译的目标架构 abi。

类型：

32 位交叉编译工具链支持以下 abi。

ilp32: int, long, 指针是 32 bits。GPRs 和堆栈用于参数传递。

ilp32f: int, long, 指针是 32bits。GPRs、32 位 FPRs 和堆栈用于参数传递。

ilp32d: int, long, 指针是 32bits。GPRs、64 位 FPRs 和堆栈用于参数传递。

64 位交叉编译工具链在 32 位的基础上增加以下 abi。

lp64: long, 指针是 64bits。GPRs 和堆栈用于参数传递。

lp64f: long, 指针是 64bits。GPRs、32 位 FPRs 和堆栈用于参数传递。

lp64d: long, 指针是 64bits。GPRs、64 位 FPRs 和堆栈用于参数传递。

-mcmmodel

功能：指定目标代码模型。目标代码模型指示对符号的约束，编译器可以利用这些约束来生成更高效的代码。

类型：

medlow：程序及其静态定义的符号必须位于一个 2GB 地址范围内，介于绝对地址 -2 GB 和 + 2GB 之间。lui 和 addi 共同使用生成地址。

medany：程序及其静态定义的符号必须位于单个 4GB 地址范围内。

auipc 和 addi 共同使用生成地址。

若不指定以上参数，编译器将使用默认值

虽然本地机器不能够直接运行该 RISC-V 程序，但通过 RISC-V 仿真模拟器 spike 和代理内核 pk，即可在本地机器上运行 riscv 程序。

执行如下命令运行 HelloWorld 程序

```
$ spike pk HelloWorld
```

程序运行将执行并在终端输出信息 “Hello World!”。

3.4 Rust 工具链的安装

rustup 工具是管理安装 Rust 官方版本工具链的二进制程序，可用于配置基于目录的 Rust 工具链。

Cargo 是 Rust 用于构建系统和进行包管理的工具，具有构建项目，下载依赖，编译源码三大功能。

通过在 Ubuntu 终端执行如下命令即可安装^[5]。

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

选择自定义安装，并使用 nightly 版本，软件将默认安装到 \$HOME/.cargo/bin 目录。

安装完成后，输入如下命令查看版本。

```
$ rustc --version
```

出现如下带 nightly 后缀的版本即为安装成功。

rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)

使用 nightly 版本的原因分析。Rust 语言有三种发布版本：nightly、beta、stable。nightly 为每天发布的新版本特性，其功能不稳定、且可能在未来时间内被移除。beta 为测试版本，定期从 nightly 版本中合并产生。stable 为稳定版本，定期从 beta 版本中合并产生。因此，nightly 版本代表当前 Rust 开发功能的前沿，具有很多 stable 版本无法使用的特性，而在编写操作系统等无需依赖标准库，且需要和汇编、C 语言结合的底层软件时，使用 nightly 版本可以方便使用更多 Rust 语言功能和特性。

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，默认创建项目的主函数内容如下。

```
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

本章内容为对系统环境搭建、工具使用的介绍。首先对硬件开发板 STEP-MAX10 的资源进行说明，并给出开发板的硬件原理图和实物图。其次对 Verilog 语言综合工具进行了介绍，因为开发板芯片为 Altera 系列，所以用来搭建系统的 EDA 软件为 Quartus。最后，介绍 RISC-V 交叉编译工具链和 Rust 工具链的安装方式，各工具的功能及使用方法。

第4章 基于 picorv32 软核的 SoC 硬件平台搭建

4.1 SoC 模块组成与接口设计

4.1.1 SoC 模块设计

SoC 总体硬件设计模块如图 4-1 所示。

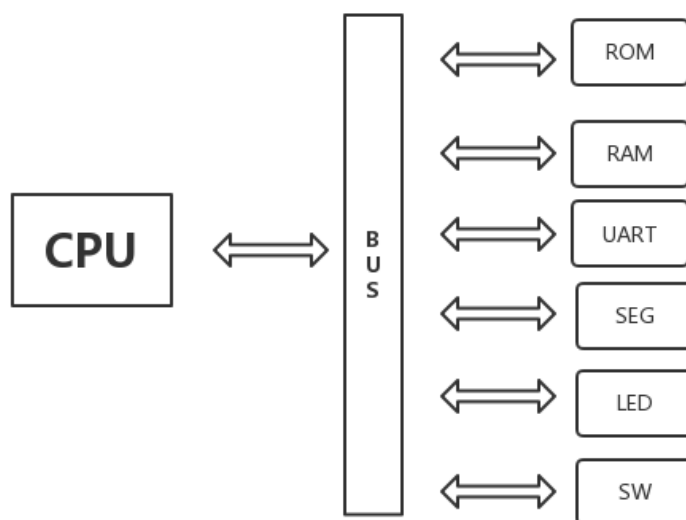


图 4-1 SoC 总体硬件设计模块

CPU 采用的是封装为 AXI-Lite 接口 IP 核的 picorv32，其余外部设备控制器均采用 Quartus 自带的 Avolon 接口 IP 核实现。通过 Quartus 中的工具 Platform Designer 将各部件进行连接，便可自动生成各设备接口转换与连接的电路。

4.1.2 串口设备的实现

基于 3.1.1 节对 STEP-MAX10 系列 FPGA 开发板资源的分析可知，SoC 现有可直接使用的外设为 LED、数码管、拨码开关和按键开关。此时，开发板仍缺少可直接与上位机进行信号传递的外部设备，无法实现交互功能。在众多的通信外设中，较为简单实用的设备为 UART。为此，本次实验利用开发板中的 36 个用户可拓展 IO 口实现简易的 UART 设备接口，即选取 2 个 IO 口分别作为 rx 和 tx 信号接口，再加上板载 5V VBus 供电与 GND 接口，通过杜邦线与 3.3V TTL 转 USB 设备连接转换后，便可实现串口功能。

串口设备实际效果如图 4-2 所示。

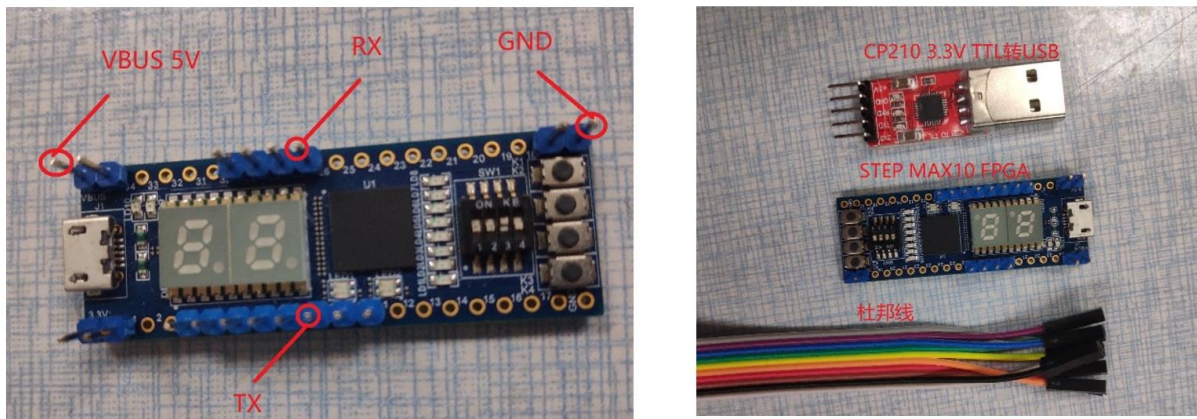


图 4-2 串口设备实际效果

4.1.3 地址空间的设计与分配

基于 picorv32 搭建的 SoC 硬件平台地址空间分配如表 4-1 所示。

表 4-1 SoC 硬件平台地址空间

外设	地址分配	大小
ROM	0x0000_0000~0x0000_ffff	44 KB
RAM	0x0001_0000~0x00010_fff	4 KB
UART	0x0200_0000~0x0200_001f	32 B
LED	0x0300_0000~0x0300_000f	16 B
Segment	0x0400_0000~0x0400_000f	16 B
SW	0x0500_0000~0x0500_000f	16 B

4.2 picorv32 IP 核的封装

Quartus 软件为用户提供了系统集成工具 Platform Designer (Qsys)，用以自动生成互连逻辑来连接 IP 核和子系统。为方便使用 Platform designer 进行系统构建，并进一步拓展各部件的功能与实用性，可选择将 picorv32 封装为 IP 核，用以直接与 SoC 其余部件连接。Platform Designer 官方 IP 核所定义的接口为 Avalon 接口，但同时也能够兼容 AXI，AXI-Lite，AXI-Stream 等一系列总线协议，使用以上总线接口的 IP 核皆可在 Platform Designer 中直接

相互连接，由工具对部件各接口进行转换连接。

使用 Platform Designer 中的 new component 功能建立新的 AXI-Lite IP 核。一般情况下，Platform Designer 并不能直接认识所有信号定义，也无法将 AXI-Lite 总线中的信号与其余信号进行分类。根据 Quartus Platform Degrigner 用户手册^[13]可知，Platform Degrigner 可识别具有特定前缀的信号，并将其进行归类。各类别前缀对应的接口类型如图 4-3 所示。

Interface Prefix	Interface Type
asi	Avalon-ST sink (input)
aso	Avalon-ST source (output)
avm	Avalon-MM master
avs	Avalon-MM slave
axm	AXI master
axs	AXI slave
apm	APB master
aps	APB slave
coe	Conduit
csi	Clock Sink (input)
cso	Clock Source (output)
inr	Interrupt receiver
ins	Interrupt sender
ncm	Nios II custom instruction master
ncs	Nios II custom instruction slave
rsi	Reset sink (input)
continued...	

图 4-3 接口类型前缀

实验中主要使用的接口类型为 AXI master，Conduit，Clock Sink，Interrupt receiver，Reset sink。可见，接口类型中并未对 AXI-Lite 接口前缀进行定义，所以使用 AXI master 前缀作为替代，也可实现信号分类功能，但仍需手动更改识别的接口类型并对齐接口中的每个信号。

为了让 Platform Degrigner 能够直接识别接口类型和自动分类，需要在 picorv32-axi 模块的基础上进行封装，将其接口名称按照 Platform Designer 对 AXI 类型接口的命名规范进行命名，其次可以去除 picorv-axi 中一些不必要的接口，仅留下时钟，复位，中断，AXI-Lite 接口，其实现在 picorv32_axi_wrapper.v 文件中。

在 new component 中添加 picorv32_axi_wrapper.v 和 picorv32.v 文件。此

时需要将 `picorv32_axi_wrapper` 作为 Top-level File，添加完成后综合文件。

选择 `reset` 的 `signal type` 设置为 `reset_n`，保证复位信号为低电平有效。若系统无法自动对齐信号，需要自行选择接口和信号，如图 4-4 所示。

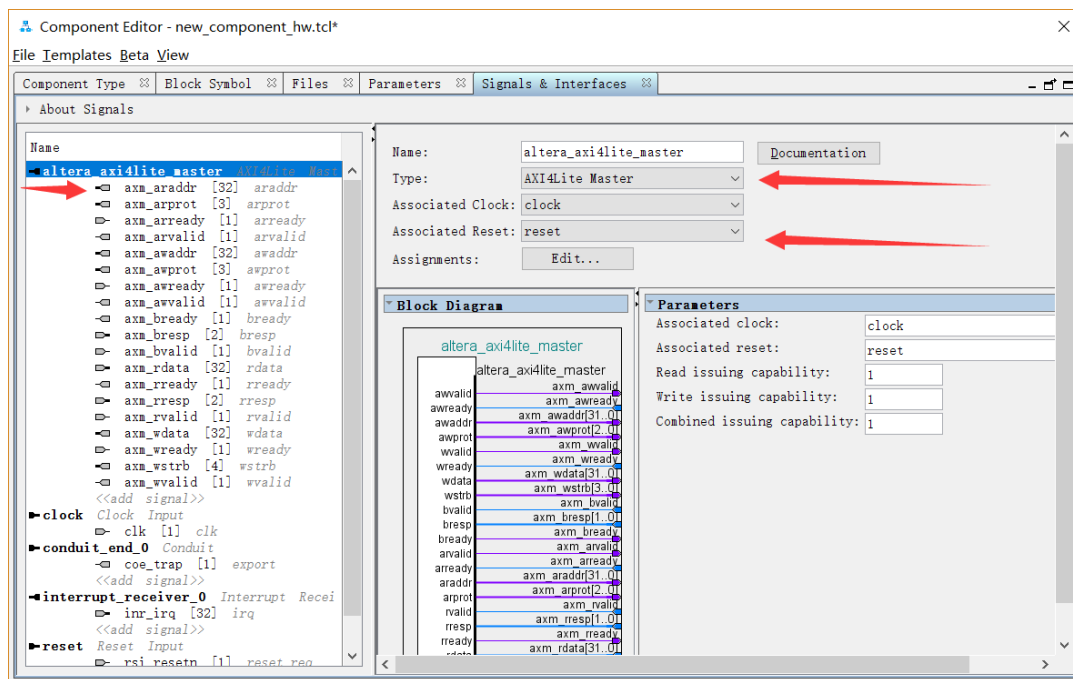


图 4-4 reset 接口信号的设置

4.3 picorv32 CPU 参数的配置

picorv32 可通过多个 `parameter` 参数配置其功能^[4]。

4.3.1 指令集模块的配置

picorv32 实现 RISC-V IMC 三个模块的指令，其中 I 模块为标准配置。

M 模块配置方式如下。

- (1) `ENABLE_MUL` 默认值为 0，需使用乘法指令时则将该参数设置为 1。
- (2) `ENABLE_DIV` 默认值为 0，需使用除法指令时则将该参数设置为 1。

C 模块配置方式如下。

`COMPRESS_ISA` 默认值为 0，需使用压缩指令集时则将该参数设置为 1。

4.3.2 中断支持的配置

- (1) `ENABLE_IRQ` 默认值为 0，需支持中断处理时则将该参数设置为 1。

（2）`ENABLE_IRQ_QREGS` 默认值为 1，当不需要使用 `q0~q3` 寄存器时则应将该参数设置为 0。若关闭 `IRQ_QREGS`，则在处理中断时，根据 RISC-V abi 的规定，一般的 C 语言程序编译时将不会使用到 `global pointer` 寄存器和 `thread pointer` 寄存器，故可将返回地址存储在 `x3 (gp)`，中断掩码存储在 `x4 (tp)` 之中。当 `ENABLE_IRQ` 的值为 0 时，`q0~q3` 寄存器也无法使用。

（3）`ENABLE_IRQ_TIMER` 默认值为 1，当不需要时钟中断时则应将该参数设置为 0。当 `ENABLE_IRQ` 的值为 0 时，时钟中断也无法使用。

（4）`MASKED_IRQ` 默认值为 `32'h 0000_0000`，其值对应需要屏蔽的 IRQ 掩码，需要和 `irq_mask` 寄存器中的值做“&”操作决定最终 IRQ 的屏蔽掩码。

（5）`LATCHED_IRQ` 默认值为 `32'h ffff_ffff`，当对应位为 1 时，表明与该位对应的 `irq` 外部输入信号将被锁存直至该中断被触发。

（6）`PROGADDR_RESET` 默认值为 `32'h 0000_0000`，其值为 CPU 启动时所执行的初始地址。

（7）`PROGADDR_IRQ` 默认值为 `32'h 0000_0010`，其值为 CPU 在发生中断时需跳转的地址。

（8）`STACKADDR` 默认值为 `32'h ffff_ffff`，其值与默认值不同时，将被设置为堆栈寄存器 `x2 (sp)` 初始地址。RISC-V 函数调用转换要求 `sp` 寄存器的值必须为 16 字节对齐。

（9）`ENABLE_PCPI` 默认值为 0，当需要支持外部协处理器时则应将该参数设置为 1。

4.4 添加外设

4.4.1 ROM

ROM 模块用于存储运行程序，直接使用 ROM IP 核即可。位数选择 32bits，大小选择 10240 words，共 40KB。主要参数为无需锁存输出值，并使用 Intel Hex 格式文件对 ROM IP 核进行初始化。

4.4.2 RAM

RAM 模块作为数据的存储和堆栈区域，直接使用 RAM IP 核即可。位数选择 32bits，大小选择 1024 words，共 4KB。主要参数为不锁存数据，加入 byteenable，当 CPU 同时读写同一地址时，返回新的数据，无需初始化。

4.4.3 串口

串口使用 RS-232 IP 核实现。主要参数为波特率 115200，去掉 Fixed baud rate 的选项。根据 Quartus IP 核用户手册^[14]可知，UART RS-232 IP 核内部寄存器分配如图 4-5 所示。

Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					(1 2)	(1 2)	Receive Data						
1	txdata	WO	Reserved					(1 2)	(1 2)	Transmit Data						
2	status (11)	RW	Reserved	eo p	cts	dct s		e	rrd y	trd y	tm t	toe	ro e	br k	fe	pe
3	control	RW	Reserved	ieo p	rts	idc ts	trb k	ie	irr dy	itr dy	it mt	ito e	iro e	ibr k	ife	ipe
4	divisor (13)	RW	Baud Rate Divisor													
5	endof- packet (13)	RW	Reserved					(1 2)	(1 2)	End-of-Packet Value						

图 4-5 RS-232 IP 核内部寄存器

波特率的计算方法如式 4-1^[14]。

$$\text{Divisor} = \frac{\text{clock frequency}}{\text{baud rate}} - 1 \quad (4-1)$$

根据项目经验，式 4-1 并非在任何情况下都成立，该验证过程详见 5.4 节。

若 PLL IP 核 input clock 为 12Mhz，output clock 为 50Mhz，baud rate 为 115200，则

$$\text{Divisor} = \frac{12\text{Mhz}}{115200} = 104$$

此时需要在程序中再次设置 Divisor 的值，而不是使用 IP 根据默认公式计算的 Divisor 值才能够让串口正常工作。

若 pll 时钟 IP 核 input clock 为 50Mhz，output clock 为 50Mhz，baud rate 为 115200，则

$$\text{Divisor} = \frac{50\text{Mhz}}{115200} = 434$$

此时式 4-1 成立，编写程序时无需额外写 Divisor，串口即可按照预先设定的波特率正常工作。

4.4.4 数码管

数码管没有可直接使用的 IP 核，需使用 PIO IP 核输出数码管显示的二进制数值，然后使用 Verilog 编写 Segment 驱动模块，读取 PIO 输出的二进制数值，并相应地将其译码为数码管显示所需的电平。

4.4.5 GPIO

GPIO 模块用于控制包括 LED，按键开关，拨码开关等设备，可直接选择 PIO IP 核进行控制。LED 的参数选择为输出 8 位。按键开关和拨码开关的参数选择输入 8 位。

4.4.6 PLL

PLL IP 核将输入时钟分频为所需的频率，此实验中输入频率为板载晶振频率 12Mhz，选择输出的频率为 50Mhz。

4.5 设备连接与地址分配

使用 Platform Designer -> System Content 连接各部件，按照 4.3.2 节的设计更改外设地址，更改外设名称，并添加对外接口。点击 finish，并生成 qsys 文件。

4.6 顶层模块的设计

将 Platform Designer 中生成 pico_axi.qsys 文件加入项目工程，并在顶层模块中对该文件进行例化，其例化示例在 pico_axi_inst.v 文件中，同时加入对数码管译码模块 Segment 的例化。

4.7 引脚与约束文件

对项目进行综合后便需对引脚进行配置，根据 STEP-MAX10 硬件手册^[8]可知，数码管为共阴极数码管，LED 为低电平点亮，各设备的引脚分配均可由手册查询，将 rx 和 tx 接入任意的 GPIO 引脚即可。

增加时钟约束，新建 Synnopsys Design Constraint File，写入 clock 源时钟约束，其中 12Mhz 时钟的周期为 83.33ns。

4.8 编译下板

编译前对启动方式进行配置，STEP MAX10 开发板的 Configuration Mode 需选择带 “With Memory Initialization” 的启动方式。

对项目进行全编译，编译完成后，选择 sof 格式文件下板。用杜邦线将 STEP-MAX10 开发板和 3.3.V TTL 转 USB 设备接口相连，并将 TTL 转 USB 设备连接至电脑。打开串口助手，设置波特率为 115200，即可看到程序运行的打印信息。

4.9 资源使用情况分析

在项目完成全编译后，查看 Quartus 中的 Flow Statue 状态栏，可得到 SoC 对 FPGA 内部资源的使用情况如表 4-2 所示。

表 4-2 FPGA 资源的使用情况

Family	MAX 10
Device	10M08SAM153I7G
Total logic elements	3,857 / 8,064 (48%)
Total pins	34 / 112 (30%)
Total memory bits	264,448 / 387,072 (68%)
UFM blocks	0 / 1 (0%)
ADC blocks	0 / 1 (0%)

4.10 本章小结

本章内容为介绍 SoC 硬件平台的搭建过程。首先介绍 SoC 的功能设计，从

整体上对 SoC 模块，地址空间分配进行说明。接着介绍串口硬件设备的实现方式。然后介绍在 Platform Designer 中封装 picorv32 为 IP 核的方法以及 picorv32 的各模块功能的参数配置。通过使用 IP 核实现各个外部设备控制器并在 Platform Designer 中直接对各设备进行连接，然后生成连接的基本电路。然后，为 Platform Designer 中生成的系统编写顶层模块，同时添加数码管译码模块，分配引脚和加入时钟约束后，全编译下板运行。最后给出了实验所搭建的硬件系统占用的 FPGA 资源情况。

第5章 SoC 硬件平台功能的测试

5.1 外设基本功能的测试

5.1.1 ROM 功能的测试

本实验使用 ROM IP 核作为 SoC 运行程序的存储部件。在项目中使用该 IP 核之前，需要确保 ROM IP 核能够在 STEP MAX10 开发板中正常工作，同时也需了解掌握程序的存放格式、初始化方法和读取方式。为此，在搭建基于 picorv32 的 SoC 之前，需额外搭建一个用于验证 ROM IP 核的项目，并对其功能进行测试。

新建项目 `rom_test`，该项目的主要功能为读取当前 ROM IP 核的指令内容显示于数码管，同时将字地址显示于 LED 灯。由于仅有两个数码管，最多可显示两个 16 进制数字即 1 个字节，所以使数码管将指令低字节显示出来。每按动一次复位键，使输入 ROM IP 核的字地址初始化为 0。每按动一次字地址自增键，则将字地址自增 1。由于项目使用组合逻辑实现数码管译码功能，ROM IP 核内部亦采用组合逻辑实现数据读取功能，所以下板后每当按动一次字地址自增按键，则代表指令字地址值的 LED 灯二进制数加 1，同时数码管显示 ROM 中该地址对应的指令字最低字节。

使用自行编写的 Hex 格式对 ROM IP 核进行初始化，则在项目下板后即可通过比对 Hex 文件内容与数码管显示内容判断当前 ROM IP 核是否正常工作。

5.1.2 UART 功能的测试

本实验采用 RS-232 IP 核实现串口功能，并与上位机进行交互。在搭建完成 SoC 后，首先对 UART 功能进行测试，即编写运行于 SoC 的裸机程序，使其能够通过串口收发数据。首先编写简单的串口测试程序，主要功能为设置 UART 的 Divisor 数值，然后读取 divisor 并将其显示至数码管，循环往串口写入同一字符，同时判断 status 寄存器，是否从上位机接收到数据，若检

测到数据则将该字符回显至终端。与此同时，也可使 LED 在一定时间内不断变化，以判断下板后程序是否一直在运行。

5.2 RISC-V 官方测试的移植

为了确保 SoC 搭建的正确性，尤其是 CPU 功能的正确性，需要使用 RISC-V 官网测试用例对 SoC 功能进行验证。

当前存在两种方式对 RISC-V 官方测试程序进行移植，第一种方式为直接下载 RISC-V 官方测试用例托管在 GitHub 仓库中的 `riscv-test` 项目，但此时需要自行搭建执行测试用例的代码环境。第二种方式为使用 `picorv32` 仓库中 `firmware` 作为执行测试用例的代码环境，但需要根据平台差异对 `firmware` 进行移植适配。通过对工作量的权衡比较，选择采用第二种方式进行移植——以 `firmware` 目录下的测试程序为基准，对自行搭建的 SoC 进行适配。

（1）更改外设地址，主要对串口设备的地址进行修改。根据 4.1 节对 `firmware` 所运行的 SoC 平台的描述可知，`firmware` 程序的地址空间为 64KB 的 RAM，串口地址为 `0x1000_0000`。将程序中所有使用到串口设备的地址更换为系统中的串口基址 `0x0200_0000`。包括 `start.S` 中输出“DONE”信息部分代码的串口地址，`tests/riscvtest.h` 文件中输出 TEST FUNC NAME 处的串口地址，RVTEST PASS 宏和 RVTEST FAIL 宏往串口输出“OK”，“ERROR”信息处的串口地址。除更改串口地址外，还需在每次写串口之前加入判断 `status` 寄存器 `trdy` 位的代码。

（2）由于 STEP-MAX10 开发板中的内存容量较小，无法将编译出来的 `firmware` 程序完全放置在 ROM 之中，故仍需对 `firmware` 程序作出适当的剪裁。首先去掉 `firmware` 目录下的所有 C 语言文件，即去除 `print` 功能，去除查找梅森素数的 C 语言程序 `sieve.c`，去除对中断，异常进行处理的 `irq.c`，去除 `multest.c`，`stats.c`，同时在 `Makefile` 中将 `firmware` 目录下 C 文件生成的目标文件链接至 ELF 文件的命令去除。其次，需要在 `start.S` 中去除调用这部分功能的代码（将文件开头 `ENABLE` 的宏注释）。

（3）对链接脚本 `sections.lds` 文件进行修改。原程序中该链接脚本直接将所有程序段放置在 `0x0000_0000 ~ 0x0000_c000` 的地址空间内。根据 SoC 地址

空间分布，需将程序的可执行代码段和只读数据放置在 0x00000000 ~ 0x0000afff 处的 ROM 处，将数据段放置在 0x0001_0000 ~ 0x0001_0fff 处的 RAM 段。

（4）加载数据段对 RAM 进行初始化。由于该程序直接运行于硬件系统之上，并没有加载程序将程序的数据段加载至 RAM 中，而 RAM 又无法同 ROM 一样使用 Hex 格式文件对其进行初始化。此时，需要在链接脚本中使用 AT 命令，将数据段的 LMA 设置在 ROM 地址空间中。然后在 Start.S 中使用汇编代码通过循环形式将数据段复制至 RAM 中进行初始化。

（5）将测试程序分为小模块。使用 Makefile 对程序进行编译，此时仍可能出现如下错误 “segmentation fault”，提示信息为程序过大，无法完全放入 ROM 空间中。说明无法一次性运行所有官方测试用例，此时可分四个批次分别运行 TEST 中的代码。通过更改根目录下的 Makefile 文件，将 tests 目录下的目标文件部分链接至 firmware 可执行文件中。

5.3 Intel Hex 格式文件的生成方式

Intel Hex 文件可用于初始化 ROM IP 核，文件格式详见附录 A，其本质为可执行程序代码段提取后加入地址、校验码并按照给定格式进行排列的文件。

若要将运行于 SoC 的源程序变为 Hex 文件，首先需使用 RISC-V 工具链将汇编源程序和 C 语言等高级语言程序编译链接为 ELF 可执行文件。

（1）编译

对于汇编文件，使用如下命令将其编译为目标文件。

```
$ riscv32-unknown-elf-gcc -c 汇编文件名 -o 输出文件名 -march=rv32im
```

对于 C 语言文件，使用如下命令将其编译为目标文件。

```
$ riscv32-unknown-elf-gcc -c 输入文件名 -o 输出文件名 -march=rv32im  
-ffreestanding -nostdlib
```

（2）链接

将所有目标文件同链接脚本编译为 ELF 文件，命令如下。

```
$ riscv32-unknown-elf-gcc -c 输入文件名 -o 输出文件名 -march=rv32im
```

-ffreestanding -nostdlib -Wl,-Bstatic,-T,链接脚本文件 -lgcc

（3）提取代码段

通过 RISC-V 工具链中的 `objdump` 提取 ELF 文件中的代码段数据，命令如下。

`$ riscv32-unknown-elf-objcopy -O binary -j .text ELF 文件 输出文件`

参数 `-O` 用以指定输出文件的格式，`binary` 表示输出文件为二进制文件。`objcopy` 中自带的 BFD 库中支持 Intel Hex 格式，可通过 `-O` 参数指定，格式名称为 `ihex`。但根据实际使用情况，当直接使用 `ihex` 格式时，输出文件的数据为小端序，若用于初始化以 32bits 字为单位的 ROM IP 核，则从 ROM IP 核中读取出来的数据的大小端则会相反。此外，其地址字段以字节为单位进行编址，若 ROM IP 指定为 32bits 的，则用于初始化的 `hex` 格式地址也应按字编址，而不是按字节编址。

因此需要自行编写软件将 `bin` 文件转换为 `hex` 格式，本实验通过编写两个程序对 `bin` 文件进行转换，其中 `bin2coe.py` 程序能将 `bin` 文件读入，然后转换为以 32 位指令为一行的 `data`，并加入 `Start code`，`byte count`，`address`，`record type`，命名为 `coe` 文件（不是标准 `coe`）。`coe2hex.c` 程序用以将 `coe` 文件读入，然后计算并加入 `checksum`。

5.4 调试的方法

在实际环境中，不管是硬件平台还是软件的功能实现都不可能一蹴而就。当程序运行时没有达到预期要求，便须对其进行调试。现将本实验过程中，对串口功能的调试方法介绍如下。

5.4.1 仿真环境下的调试

根据 5.1.2 节串口的测试程序的功能设计可知，程序若正常运行，其预期效果应为数码管显示此时 RS-232 IP 核内部 `Divisor` 寄存器的值，LED 灯循环发生变化，串口不断向上位机发送数据，上位机设置相应的波特率后可接收到预定的字符串。若程序下板后，无法根据指定的波特率接收到数据或者接收到的数据为乱码，则说明 RS-232 IP 核的配置出现问题。

首先在仿真环境下对该功能进行验证。

（1）编写 testbench 给 SoC 各接口激励信号。在 testbench 文件中设置输入时钟为 12Mhz，即与 STEP-MAX10 开发板上的晶振频率相同。

（2）若使用 UART RS-232 IP 核搭建环境，则该 IP 核在仿真环境下默认的 Divisor 大小为 4。因为需要与下板环境相同，所以需要额外加入在程序中设置 Divisor 数值的代码。

（3）在 testbench 中模拟上位机给 rx 接口发送数据，每一位符号的持续时间为 $\text{divisor} \times \text{cycle}$ 。

（4）运行后，首先观察 tx 和 led 是否有变化，波形是否与预期的效果相同。若不相同，则检查此时程序运行的状态。如果程序运行至预定的地址范围之外或者陷入循环，则说明是程序的逻辑错误。

（5）使用 objdump 工具反编译 ELF 文件或者使用 addr2line 工具查看相应地址对应的源代码，以排除程序的问题。

（6）若程序能够正常运行，但是相应接口却没有输出信号或者信号值不对，则根据问题可以定位相应的硬件设备问题，比如 tx 的宽度与 rx 接收的宽度不相符，则是 Divisor 寄存器的问题。

5.4.2 真实环境下的调试

在仿真环境下程序能够正常运行并得到预期效果，下板后在真实环境下却未出现预期效果，则需进一步对问题进行排查。

（1）如果程序下板后 LED 正常运行，且数码管显示的分频系数数值为按照手册公式计算的数值，但是上位机按照设置波特率接收数据时却显示为乱码，此时可以定位问题在于串口波特率。

（2）通过示波器查看引脚 tx 的信号波形。若 SoC 中运行的程序在相同时间间隔内往 tx 引脚发送同样的数据，则示波器能够明显接收到稳定的信号波形。通过 Trigger 设置低电平触发，则运行后若有数据发送至 tx，则示波器将捕捉此次发送的数据。

（3）tx 引脚发送的高电平为 3.3v，使用示波器对每一位符号的发送位宽进行测量，若 tx 每位信号位宽度为 Δx ，则串口发送的波特率计算公式为式 5-1。

$$\text{baund rate} = \frac{1}{\Delta x} (5 - 1)$$

通过计算，可以得知此时串口实际输出的波特率与设定的波特率是否相符，然后对 RS-232 IP 核参数做进一步调整。

（4）根据式 4-1 的计算公式可知，波特率的值与 Divisor 还有时钟频率有关。在使用示波器查看串口实际输出波特率后，也可将其余信号引出至板外然后使用示波器查看，如时钟接入预留 IO 口。数码管的显示内容为 16 进制的 Divisor。此时，即可得知波特率计算公式中的三个未知量数值。使用 matlab 可推导出波特率计算公式，然后同 Quartus IP 核用户手册中给出的公式进行比较。

（6）本次实验中对 RS-232 IP 核调试得出的结论如 4.4.3 节所述。

（7）在真实环境中也可利用 Quartus 中的工具 Signal Trap Logic Analyzer，下板后捕捉 SoC 内部的信号进行调试。

5.5 本章小结

本章内容介绍了 SoC 硬件平台的测试方法。首先介绍了外设测试程序的功能。然后以 picorv32 仓库中的 firmware 目录作为基础，移植 RISC-V 官方测试程序，通过更改外设地址，修改链接脚本等实现对程序的装载执行。接着介绍了 Intel Hex 格式文件的生成方式，用以初始化 ROM 从而加载需要运行的程序。最后，以串口设备的调试过程为例介绍了此次实验用到的调试工具和方法。

第6章 引导程序的设计

6.1 picorv32 的特权极架构

picorv32 的特权级架构并未参照 RISC-V Privilege ISA Specification 中的规范进行实现。为了使用尽可能少的硬件资源，开发者通过实现自定义的指令集和 CSR 寄存器集对中断进行处理。

6.1.1 CSR 寄存器

CSR 寄存器即控制状态寄存器，用于配置或记录 CPU 的运行状态。picorv32 中定义的 CSR 寄存器如表 6-1 所示^[4]。

表 6-1 picorv32 CSR 寄存器

CSR	功能
q0	存储中断或异常返回地址，对应 RISC-V 标准 CSR 中的 epc 寄存器
q1 (irq_pending)	存储当前准备处理的中断号，当某位置为 1 时，表明与该位下标对应的中断触发，对应 RISC-V 标准 CSR 中的 ip 寄存器
q2	作为临时寄存器保存中间值，对应 RISC-V 标准 CSR 中的 scratch
q3	作为临时寄存器保存中间值，对应 RISC-V 标准 CSR 中的 scratch
irq_mask	中断屏蔽掩码，当某位置 1 时，表明与该位下标对应的中断将被屏蔽，不对其进行处理
timer	时钟计数器，每个时钟周期自减 1，当其值变为 0 后触发时钟中断。若需重新触发，需要重新赋值
counter_cycle	周期计数器，每个时钟周期自增 1
counter_insr	指令计数器，每执行一条指令自增 1

6.1.2 特权极指令

picorv32 通过自定义指令方式对 CSR 寄存器进行访问或对中断进行处理^[4]。

（1）getq rd, qs

编码方式：0000000 0----- 000XX --- XXXXX 0001011

f7 rs2 qs f3 rd opcode

功能：该指令将通用寄存器中的值赋给 q_n 寄存器。

示例：getq x5, q2

（2）setq qd, rs

编码方式：0000001 0----- XXXXX --- 000XX 0001011

f7 rs2 rs f3 qd opcode

功能：该指令将 q_n 寄存器中的值赋给通用寄存器。

示例：setq q2, x5

（3）retirq

编码方式：0000010 0----- 00000 --- 00000 0001011

f7 rs2 rs f3 rd opcode

功能：该指令使程序从中断处理中返回，即将 q_0 寄存器中的值赋给 PC 寄存器，同时重新开启中断。

示例：retirq

（4）maskirq

编码方式：0000011 0----- XXXXX --- XXXXX 0001011

f7 rs2 rs f3 rd opcode

功能：该指令将 rs 寄存器中的值写入 irq_mask 寄存器，同时将 irq_mask 的旧值返回至 rd 寄存器。

示例：maskirq x1, x2

（5）waitirq

编码方式：0000100 0----- 00000 --- XXXXX 0001011

f7 rs2 rs f3 rd opcode

功能：CPU 暂停执行等待中断被挂起，当有中断触发时才会使 CPU

继续运行，`irq_pending` 的值将被写入 `rd` 寄存器。

示例：`waitirq x1`

（6）timer

编码方式：0000101 ----- XXXXX --- XXXXX 0001011

f7 rs2 rs f3 rd opcode

功能：设置 `timer` 寄存器的值，将 `rs` 的值赋给 `timer`，同时将 `timer` 的旧值赋给 `rd`。

示例：`timer x1, x2`

6.1.3 自定义指令的使用

由于自定义类型指令无法通过 GNU 汇编器 `as` 进行编译，故需要通过宏定义方式对自定义类型指令进行编译。各自定义指令的宏定义如表 6-2 所示 [4]。

表 6-2 自定义指令的宏定义

```
#define r_type_insn(_f7, _rs2, _rs1, _f3, _rd, _opc) \
.word (((_f7) << 25) | ((_rs2) << 20) | ((_rs1) << 15) | ((_f3) << 12) | ((_rd) << 7) | ((_opc) << 0))

#define picorv32_getq_insn(_rd, _qs) \
r_type_insn(0b0000000, 0, regnum_ ## _qs, 0b100, regnum_ ## _rd, 0b0001011)

#define picorv32_setq_insn(_qd, _rs) \
r_type_insn(0b0000001, 0, regnum_ ## _rs, 0b010, regnum_ ## _qd, 0b0001011)

#define picorv32_retirq_insn() \
r_type_insn(0b0000010, 0, 0, 0b000, 0, 0b0001011)

#define picorv32_maskirq_insn(_rd, _rs) \
r_type_insn(0b0000011, 0, regnum_ ## _rs, 0b110, regnum_ ## _rd, 0b0001011)

#define picorv32_waitirq_insn(_rd) \
r_type_insn(0b0000100, 0, 0, 0b100, regnum_ ## _rd, 0b0001011)

#define picorv32_timer_insn(_rd, _rs) \
r_type_insn(0b0000101, 0, regnum_ ## _rs, 0b110, regnum_ ## _rd, 0b0001011)
```


6.1.4 中断信号

picorv32 CPU 实现简易的内部中断控制器，该控制器接口如表 6-3 所示。

表 6-3 picorv32 中断控制器接口

名称	方向	位宽	功能
irq	input	32	接入外部中断源
eoi	output	32	当进入 CPU 进入中断处理时，若将此次需处理的中断号所对应的位置高，直到中断处理结束
trap	output	1	当 CPU 进入中断处理状态时，该位置为高

picorv32 可处理 32 种中断，irq 接口输入的每一位对应一种类型的中断。其中 0-2 号中断能够被 CPU 内部所触发，其中断控制类型如表 6-4 所示^[4]。

表 6-4 picorv32 中断类型

IRQ	Interrupt Source
0	Timer Interrupt
1	EBREAK/ECALL or Illegal Instruction
2	BUS Error (Unalign Memory Access)

其余中断可由用户自定义。CPU 一次可处理多个中断，CPU 复位后所有中断将被默认关闭，timer 寄存器也被默认清理。若在 1~2 号中断被屏蔽的情况下出现非法指令或者访存地址错误，将导致 CPU 停止运行。CPU 在进入中断处理的过程中将自动关闭中断，直到执行 irqret 指令后自动开启中断。发生中断时，处理器需要跳转的地址为固定值，可通过参数 PROGADDR_IRQ 配置，其默认值为 0x10，一旦配置成功便无法再此更改，除非重新综合布局 SoC。

6.2 引导程序功能分析

一般的引导程序功能为对硬件进行基本的初始化，解析并加载操作系统

内核至内存指定位置后跳转执行操作系统。由于本实验中引导程序的设计受如下两个基本限制：

- （1）硬件存储资源局促，操作系统无法由 ROM 重新加载至 RAM 中执行。
- （2）picorv32 CPU 并未按照 RISC-V Privilege Specification 中对于特权极架构标准的要求进行实现。

为此引导程序按照如下方案进行设计：

- （1）简化解析加载 ELF 格式操作系统内核的代码实现，将编译完成的操作系统内核的不同程序段转换为二进制文件并直接加载至引导程序特定位置，在编译引导程序时再一同编译。
- （2）提供 sbi 接口（Supervisor Binary Interface）和中断处理。

引导程序功能模块结构如图 6-1 所示。

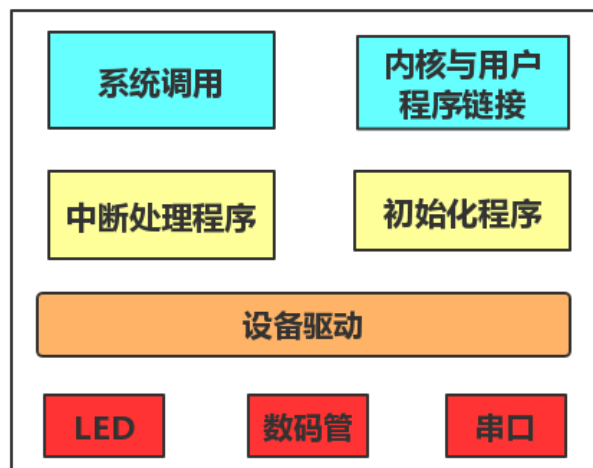


图 6-1 引导程序功能模块结构

6.3 引导程序运行流程

（1）引导程序代码段作为 CPU 执行的第一段代码，其起始地址将被放置在 CPU 启动地址处，即 0x0000_0000。由于 CPU 的中断跳转地址设置在 0x10 处，引导程序从 0x10 处开始实现中断处理功能。本实验中直接使用无条件跳转指令跳过中断处理程序入口，转入初始化代码。

（2）初始化代码需要将 32 个通用寄存器初值设置为零。其次，加载程序的数据段和 bss 段至内存中作为内存初始化数据，bss 数据段初值全部赋为零。

（3）设置串口设备的分频系数。

（4）初始化引导程序的堆栈寄存器 `sp`、全局寄存器 `gp` 和线程寄存器 `tp`。

其中，堆栈寄存器的值设置为 RAM 物理地址空间的地址最大值，由于堆栈的增长顺序为自高地址向低地址生长，故可有效避免堆栈溢出。

（5）引导程序打印“BOOT”信息，表明设备初始化完成。

（6）引导程序调用“`picorv32_maskirq_insn`”自定义指令设置 `irq_mask` 寄存器的值，等同于开启中断，并通过 `ecall` 指令触发异常，进而检验当前中断是否已成功开启。

（7）引导程序调用“`picorv32_timer_insn`”自定义指令，设置时钟计数器的值为 10000。

（8）引导程序进入循环点亮 LED 灯，每次循环均调用

“`picorv32_waitirq_insn`”自定义指令等待时钟中断触发后再继续执行，循环 8 次后跳转至操作系统代码段开始执行。

（9）通过“`.incbin`”伪指令将操作系统代码段直接加载到标号为 `text_payload_start` 处，同时也将该地址设置为引导程序代码段偏移 0x1000 的地址。

（10）通过“`.incbin`”伪指令将用户可执行程序 ELF 文件放置在 `elf_payload_start` 标号处。

（11）引导程序数据段将预留 32 个字大小空间作为中断处理程序保存寄存器现场的堆栈，其开始地址为 `irq_regs` 标号。

（12）数据段预留 128 个字大小空间作为中断处理函数堆栈。

6.4 中断处理程序

（1）中断处理程序入口从地址为 0x10 处开始^[4]。由于启用了 `q0~q3`，所以可使用 `q2~q3` 作为 `scratch` 来存储 `x1`、`x2` 寄存器的值，故通过 `picorv32_setq_insn` 自定义指令将 `x1` 和 `x2` 的值设置到 `q2` 和 `q3` 之中。

（2）中断处理程序将 `irq_regs` 赋给 `x1` 寄存器，`irq_regs` 为在 RAM 中分配给中断处理程序保存寄存器的地址，用以存储引发中断时各通用寄存器的值，并将中断返回地址存储在偏移量为 0 的地址处，其余寄存器按照下标存储在

$n*4$ 偏移地址处。

（3）中断处理程序将 `irq_stack` 赋给 `sp` 寄存器，`irq_stack` 为在 RAM 中分配给中断处理程序的堆栈地址。

（4）中断处理程序将 `irq_regs` 和 `q1` 寄存器的值作为参数，并通过 `a0`，`a1` 寄存器传入，调用中断处理函数 `irq` 解析中断类型并做出相应处理。

（5）从中断处理函数 `irq` 返回后，中断处理程序将恢复中断现场，重新设置时钟计数器。

（6）调用 `picorv32_retirq_insn` 自定义指令返回原程序继续执行。

因为 CPU 进入异常模式时自动屏蔽中断，即不允许嵌套中断，故无需额外加入代码开关中断。

6.4.1 中断处理函数

中断处理函数使用 C 语言编写，其声明形式在 `firmware.h` 头文件中，实现在 `irq.c` 文件中。为避免跨平台导致的数据长度冲突，在头文件中加入对 `stdint.h` 的引用，则在编写程序时可直接使用 C99 标准库所定义的数据类型，而不使用 `gnu C` 标准库定义的数据类型。

（1）中断处理函数声明

中断处理函数声明形式如下。

```
uint32_t * irq(uint32_t *regs, uint32_t irqs)
```

可见，`irq` 函数需要传递两个参数，第一个参数为指向存储中断现场通用寄存器的内存指针，第二个参数为中断类型掩码。由 6.3.3 节可知，这两个参数分别通过 `a0` 和 `a1` 寄存器传入。中断处理函数返回值为指向存储通用寄存器的内存指针。

（2）判断中断类型

根据 6.1.4 节，系统中预设的中断信号类型共有 3 类，分别对应中断掩码中下标为 0~2 的位，其中 0 号位为时钟中断，1 号位为异常指令 `EBREAK`、`ECALL` 和非法指令，2 号位为访存地址不对齐异常。`irq` 处理函数首先需要根据传入的 `irq` 掩码判断出具体的中断类型号，即依次比较判断相应位的值。

（3）时钟中断

对于 1 号时钟中断，系统设计的处理方式是通过全局静态变量存储时钟中断触发的次数。每次触发时钟中断，便打印 “[TIMER-IRQ] + 时钟中断次数”字符串至终端。

（4）异常指令中断

对于 2 号异常指令中断，则需要进一步判断引发中断的原因。根据 6.3.3 节可知，存储中断现场通用寄存器的内存空间 `regs` 的 0 地址偏移处存储的值为中断处理程序返回地址，根据对异常的实现可知，对于 `ECALL`、`EBREAK` 一类的触发软中断的异常指令，其返回地址为触发软中断指令的下一条指令地址。故可通过计算得出异常指令地址，其计算方式如下。

```
uint32_t pc = (reg[0] & 1) ? regs[0] - 3 : regs[0] - 4;
```

`pc` 的值即为异常指令地址，并以字为单位对齐。通过访问内存将该地址处的指令读取回来做进一步的判断。通过如下语句读取引发中断的指令，并存入 `instr` 变量中。

```
uint32_t instr = *(uint32_t *)pc;
```

此时根据 RISC-V 指令编码手册，比较 `instr` 的值即可判断出异常指令的类型。若为 `EBRAK` 指令，则 `instr` 的编码为 `0x001000073`。若为 `ECALL` 指令，则 `instr` 的编码为 `0x00000073`。其余情况均可判断该指令为非法指令。

（5）访存地址不对齐中断

对于 3 号访存地址不对齐中断，系统设计的处理方式为打印提示信息：“Bus Error”，然后停机等待重启。

（6）中断调试信息的打印。

由于传入的参数包括对存储通用寄存器的内存地址，可通过打印通用寄存器的值作为调试信息对软件进行调试，在遇到 `ebreak` 指令或者非法指令时将打印所有寄存器的值。通过两个循环语句，系统分别用下标访问存储通用寄存器的数组，并将其值以 16 进制形式打印，4 个寄存器为一行打印。

6.5 串口驱动程序

操作系统通过串口与上位机进行交互，即通过串口向上位机发送数据。上位机接收后，通过串口工具进行显示，反之也可由串口工具向下位机发送

数据，操作系统接收进行处理。串口功能直接通过 Quartus RS-232 IP 核实现，其配置和使用方式详见 4.6.3 节。驱动程序需访问和使用的寄存器包括 rxdata、txdata、status 寄存器，其偏移量分别为 0、1、2，以字为单位。因为串口设备的基址为 0x0200_0000，故分别对三个寄存器的基址的进行宏定义如下。

```
#define INPORT      (*(volatile uint32_t*)0x02000000)    //uart_rx
#define OUTPORT     (*(volatile uint32_t*)0x02000004)    //uart_tx
#define uart_status (*(volatile uint32_t*)0x02000008)
```

由于寄存器以字为单位，故使用 uint32_t 类型的指针指向内存映射的 UART 设备寄存器。volatile 关键字的使用是为了避免编译器对使用该变量值的地方做过多优化，例如在执行检查 status 寄存器中的发送允许位 trdy 时需执行循环等待，如下所示。

```
while(!(uart_status & 0x0040));
```

若未使用 volatile 关键字，则编译器在进行数据流分析时，将认定该循环中没有任何能够改变 uart_status 寄存器的语句，则会将该语句优化为

```
if(!(uart_status & 0x0040)) {
    for(;;);
}
```

显然此时编译器优化导致代码中循环判断 status 寄存器值的功能发生改变。通过 volatile 关键字即可使编译器在使用该变量时必须每次重新读取其值，而不是直接使用保存在寄存器中的备份。

（1）print_char()函数的实现

printf_char()函数的功能为向 txdata 寄存器发送数据，输入参数为需发送的字符，返回值为 0。在发送数据之前，调用 trdy 函数判断当前串口的状态，若为可发送状态，则向 txdata 寄存器发送数据。此外，驱动程序还在 print_char()函数的基础上，实现打印字符串的函数 print_str()，打印十进制数字的函数 print_dec()以及打印十六进制的函数 print_hex()。

（2）getchar()函数的实现

getchar()函数的功能为从 rxdata 寄存器读取数据，无输入参数，返回值

为读取的寄存器数据并将其转换为 `char` 类型。在读取数据之前，调用 `rrdy` 函数判断当前串口的状态，若为可读取状态，则从 `rxdata` 寄存器读取数据。

6.6 SBI 设计

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。由于中断处理函数没有预留参数作为系统调用号，设计将自定义系统调用函数的参数传递规则。系统调用函数的参数传递规则如表 6-5 所示。

表 6-5 系统调用函数的参数

寄存器	功能
a0~a3	系统调用函数参数
a7	系统调用号
a0	系统调用函数返回值

中断处理函数通过读取 `regs` 数组中偏移量为 10~13（a0~a3）的值即可得到调用函数传递的三个参数，通过读取 `regs` 数组中偏移量为 17（a7）的值即可得到系统调用类型。

在中断处理函数中，定义系统调用型号的枚举类型如下。

```
enum sbi_call_t {  
    SBI_CONSOLE_PUTCHAR = 1,  
    SBI_CONSOLE_GETCHAR = 2,  
};
```

系统仅实现两种系统调用。

（1）对于 `SBI_CONSOLE_PUTCHAR` 系统调用，即向 `console` 打印一个字符，其通过将第一个参数 `a0` 传入的值传递给由输入输出驱动提供 `print_char()` 函数实现功能，返回值为 0。

（2）对于 `SBI_CONSOLE_GETCHAR` 系统调用，即向 `console` 打印一个字符，其通过调用输入输出驱动提供的 `getchar()` 函数，对字符进行接收。

6.7 本章小结

本章内容为介绍引导程序的功能和运行流程。首先介绍了 `picorv32` 的特权极实现，由于其与标准的 `RISC-V` 架构实现不同，所以引导程序作为直接与硬件交互的底层软件需对其进行适配。其次，介绍了引导程序在资源限制的情况下对功能的选择，并介绍引导程序的运行流程。最后逐一介绍中断处理、驱动和 `SBI` 的实现方式。

第7章 rCore 操作系统的分析与移植

7.1 移植 rCore 的可行性分析

rCore 操作系统 RISC-V 架构版本所针对的平台为 Sifive 公司发布的 HiFive Unleashed 开发板。若需将 rCore 移植至本实验所搭建的 SoC 硬件平台，则需要对操作系统中硬件支持部分进行更改。

目前硬件平台上所搭建的 SoC 可用的硬件资源包括 378Kbit 的 Block Memory, UART, GPIO, 数码管等。操作系统代码可存储至布局为 ROM 的 Block Memory 资源中，使得系统能够被 CPU 正常加载执行。与此同时，串口设备保证了操作系统基本的输入输出功能，便于用户与系统进行交互。最后，利用布局为 RAM 的 Block Memory 资源保证了操作系统在运行时能够加载数据和设置堆栈等程序运行所必须的资源支持。

考虑到目前所持有的硬件资源较少，尤其是用于存储和加载程序的存储设备的局限性，无法支持将操作系统加载至 RAM 中进行运行，故将操作系统固化至 ROM 中，由 CPU 直接读取存储在 ROM 中的操作系统并直接运行。RAM 资源则完全留作数据加载区域与堆栈分配区域，即硬件资源仅支持将 rCore 移植为嵌入式操作系统。对该嵌入式操作系统的功能设计如下：

- （1）实现格式化基本输入输出。
- （2）实现对中断和异常的处理。
- （3）实现对 ELF 格式可执行用户程序的解析和执行。

7.2 最小化内核

操作系统的本质是一个独立式可执行程序，即裸机程序。这决定了操作系统的编写过程与一般直接在系统中运行的可执行程序不同，其最大的特点是无法使用依赖于某特定平台的函数库。此外，由于编译器没有符合硬件平台的目标结构，因此其编译也必须以特殊方式进行。

7.2.1 新建工程

通过 `cargo` 新建系统工程，`xy_os`。

执行如下命令，默认新建二进制可执行项目

```
$ cargo new xy_os
```

执行如下命令测试项目是否可以正常运行

```
$ cargo run
```

其运行效果如下。

```
z@z-X555LI:~$ cargo new xy_os
    Created binary (application) `xy_os` package
z@z-X555LI:~$ cd xy_os/
z@z-X555LI:~/xy_os$ cargo run
    Compiling xy_os v0.1.0 (/home/z/xy_os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
    Running `target/debug/xy_os`
Hello, world!
```

此时该项目为针对 x86 平台 Linux 操作系统构建的二进制可执行项目。

7.2.2 更改编译的目标架构

为了让编译器编译出符合目标架构的操作系统，需要根据架构特点进行更改。Rust 使用 LLVM 作为其编译器后端，首先查看 `rust` 支持编译的目标机器架构，执行如下命令：

```
$ rustup target list
```

截取部分输出结果如下。

```
z@z-X555LI:~$ rustup target list
mips-unknown-linux-gnu
riscv32imac-unknown-none-elf
riscv32imc-unknown-none-elf
riscv64gc-unknown-none-elf
```

```
riscv64imac-unknown-none-elf
x86_64-unknown-linux-gnu (default)
```

target 各字段的含义分别为 cpu 架构、供应商、操作系统和 ABI。可见，当前 Rust 编译器的默认架构为 x86_64-unknown-linux-gnu，即为 x86_64 的 CPU，未知厂商，Linux 操作系统、gnu ABI 目标架构。

由输出结果可知 Rust 编译器支持 riscv32 架构，但无法保证完全与系统兼容，因此需要使用 json 文件对 LLVM 的目标三元组进行自定义，并在编译项目时通过 --target json 文件名指定^[16]。

在项目中新建文件 xy_ox.json，json 文件的内容如下。

```
{  "llvm-target": "riscv32",
    "data-layout": "e-m:e-p:32:32-i64:64-n32-S128",
    "target-endian": "little",
    "target-pointer-width": "32",
    "target-c-int-width": "32",
    "os": "none",
    "arch": "riscv32",
    "cpu": "generic-rv32",
    "features": "+m",
    "max-atomic-width": "32",
    "linker": "rust-ld",
    "linker-flavor": "ld.lld",
    "pre-link-args": {
        "ld.lld": ["-Tsrc/boot/linker.ld"] },
    "executables": true,
    "panic-strategy": "abort",
    "relocation-model": "static",
    "eliminate-frame-pointer": false
```

选择的 LLVM 目标架构为 riscv32，小端序，int 类型和指针宽度均为 32 位，同时指定自定义链接脚本对操作系统进行链接，链接脚本详见附录 C。

7.2.3 去除标准库的引用

查看项目中 `main.rs` 文件内容如下。

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

该文件默认使用标准库中的 `println!` 宏进行格式化输出。在 7.4.3 节将实现该宏，此时可暂时将该宏的使用去除。

在文件中加入如下语句关闭编译器默认链接标准库的功能^[16]

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

其中，`no_main` 代表不再链接标准库，则程序将不再使用运行时库。
`no_main` 代表不再使用 Rust 默认的函数入口点。

Rust 规定当程序发生异常时需要有相应的函数对其进行处理。标准库对应的函数为 `panic`，当关闭对标准库的使用时，该函数便无法被编译器使用。为避免程序出现错误，需要实现与 `panic` 同名的函数，其实现如下。

```
use core::panic::PanicInfo;  
#[panic_handler]  
fn panic(_info: &PanicInfo) -> ! {  
    loop {}  
}
```

Rust 语言中有不依赖于平台的 `core` 库，该库主要包含基础的 Rust 类型，如 `Result`、`Option` 和迭代器等。`panic` 函数则使用 `core` 库中的错误信息类型 `PanicInfo` 作为参数。

标准的 Rust 程序在对 `panic` 处理时会进行堆栈展开，以析构堆栈中的所有生存变量，达到释放内存的目的。此时将使用到标准库中定义的

‘`eh_personality`’ 语义项^[16]，解决方式为禁用堆栈展开，即修改 `Cargo.toml` 文件，加入如下内容。

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

7.2.4 堆栈设置

由 6.3 节引导程序启动操作系统的方式可知，引导程序并未为操作系统设置堆栈，因此在跳转至操作系统 main 函数运行前，需手动设置堆栈。初始化堆栈的功能可使用汇编指令实现，Rust 语言通过 `global_asm!` 宏可使用内嵌汇编指令，同时可通过 `include_str!` 将特定文件作为字符串包含到目标文件中^[16]。在 `main.rs` 文件中添加如下内容。

```
#![feature(global_asm)]
global_asm!(include_str!("entry.asm"));
```

`#![feature(global_asm)]` 可开启 Rust 编译器对汇编指令的支持。

设置堆栈的 `entry.asm` 内容如下。

```
.section .text
.global _start
_start:
    lui sp, %hi(bootstacktop)
    addi sp, sp, %lo(bootstacktop)
    call main

.section .data
.align 4
.global bootstack
bootstack:
    .space 2048
.global bootstacktop
bootstacktop:
```

在 `data` 数据段中从标号 `bootstack` 到标号 `bootstacktop` 处预留 2KB 空间

作为操作系统的堆栈空间。因为堆栈自地址高处向低处增长，所以将 `bootstacktop` 赋值给 `sp` 寄存器。堆栈初始化完成后，通过 `call` 指令跳转至 `main` 函数。

7.2.5 操作系统 main 函数

在 `main.rs` 文件中实现 `main` 函数如下。

```
#[no_mangle]
pub extern "C" fn main() { }
```

其中 `#[no_mangle]` 属性可以防止编译器对函数名称进行优化。因为由 `entry.asm` 跳转至 `main` 函数遵循的是 C 调用规则，`extern "C"` 关键字使编译器在编译此函数时使用 C 调用规则，否则将默认使用 Rust 调用规则。`pub` 保证该函数名可以被 `entry.asm` 访问。

Rust 编译器在连接时使用 `rust lld`，在禁用标准库后，将出现如下错误：

“rust lld: error: undefined symbol: abort”

这说明 `rust lld` 需要依赖符号 `abort`，因此需手动实现 `abort` 函数作为标号提供给 `rust lld`，其实现如下。

```
#[no_mangle]
pub extern fn abort() {
    panic!("abort!");
}
```

7.3 SBI 函数封装

引导程序向操作系统提供基本输入输出系统调用，操作系统可通过汇编指令 `ecall`，并传入相应参数进行调用。为方便使用，可以通过创建相应的库对系统调用进行封装，操作系统通过导入外部库的方式使用系统调用功能。

7.3.1 系统调用库的创建

输入如下命令创建库项目 `bbl`^[17]。

```
cargo new --lib bbl
```

修改 lib.rs 文件内容为如下。

```
#![no_std]
#![feature(asm)]
pub mod sbi;
```

#![no_std]表明不使用标准库。由于需要使用 ecall 指令进行系统调用，故通过#![feature(asm)]开启内联汇编。然后声明公共子模块 sbi。

新建文件 sbi.rs 实现 sbi 模块，其功能为对系统调用函数进行封装，内容如下。

```
pub fn console_putchar(ch: usize) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}
pub fn console_getchar() -> usize {
    sbi_call(SBI_CONSOLE_GETCHAR, 0, 0, 0)
}
#[inline(always)]
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
}
const SBI_CONSOLE_PUTCHAR: usize = 1;
const SBI_CONSOLE_GETCHAR: usize = 2;
```

7.3.2 库导入

将 bbl 库复制到 xy_os 项目根目录中，更改 Cargo.toml 文件内容。

```
[dependencies]
bbl = {path = "bbl/"}
```

将该库以相对路径方式加入到项目的依赖中。此时，操作系统即可通过调用该库中的 console_putchar 和 console_getchar 函数进行数据输入输出。

7.4 格式化输出

7.4.1 IO 模块

为实现操作系统代码和功能的模块化，在 xy_os 项目中创建一个新的模块用于管理 io。在 main.rs 文件中加入 IO 模块的声明如下

```
pub mod io;
```

在 src 目录下新建模块文件 io.rs，实现函数 putchar 和 puts，用于实现输入输出功能^[16]，其内容如下。

```
use bbl::sbi;

pub fn putchar(ch: char) {
    sbi::console_putchar(ch as u8 as usize);
}

pub fn puts(s: &str) {
    for ch in s.chars() {
        putchar(ch);
    }
}
```

通过使用 bbl 库中 putchar 函数可输出单个字符，然后使用 putchar 函数和循环语句实现 puts 函数以输出字符串。

7.4.2 Write trait 的实现

Rust core 库中带有用于格式化和打印字符串的库 fmt，该库中包含将消

息格式化为数据流所需方法的 trait——Write，其简要信息如下^[18]。

```
pub trait Write {  
    fn write_str(&mut self, s: &str) -> Result;  
    fn write_char(&mut self, c: char) -> Result { ... }  
    fn write_fmt(&mut self, args: Arguments) -> Result { ... }  
}
```

根据 trait 的特性，仅需为 Write trait 实现 write_str()方法即可使用该 trait 中其余的方法，包括格式化输出方法 write_fmt()。write_str()方法的功能为，将 String Slice 作为参数传入，然后将该 String Slice 输出，返回 Result 类型。仅当成功写入整个 String Slice 时，此方法返回 Ok(()), 在写入所有数据或发生错误之前，此方法不会返回。

在 IO 模块中实现该 fmt trait。首先引入 core::fmt 和 core::Write。

```
use core::fmt::{self, Write};
```

然后定义结构体 StdOut 用以实现 fmt trait^[16]。

```
struct StdOut;  
  
impl fmt::Write for StdOut {  
    fn write_str(&mut self, s: &str) -> fmt::Result {  
        puts(s);  
        Ok()  
    }  
}
```

write_str()方法通过直接调用 IO 模块自实现的 puts 函数输出参数 String Slice。

在为 StdOut 结构体实现 fmt::Write trait 之后，便可使用格式化输出方法 write_fmt()。write_fmt()方法的输入参数为 fmt::Arguments 结构的变量，可通过 core::format_args!宏创建。core::format_args!宏为 Rust core 库用于格式化字符串创建和输出的核心宏，通过为传递的每个参数获取包含 {} 的格式字符串文本来发挥作用。

7.4.3 println! 宏

查看标准库中 `println!` 的实现方式^[19]。

```
[macro_export]
macro_rules! println {
    () => (print!("\n"));
    ($($arg:tt)*) => (print!("{}", format_args!($($arg)*)));
}
```

`println!`宏的实现包含两条匹配规则，匹配完成后的语句均为在 `print!`宏的基础上加入“`\n`”输出。然后通过`#[macro_export]`用以将该宏重导出至根目录。

进一步查看标准库中 `print!` 宏的实现方式。

```
[macro_export]
macro_rules! print {
    ($($arg:tt)*) =>
    ($crate::io::_print(format_args!($($arg)*)));
}
```

`print!`宏的实现为调用 IO 模块中的`_print()`函数。`_print()`函数的参数为`format_args!`宏生成的 `Arguments` 变量。此时，IO 模块仅需实现`_print()`函数即可实现对`println!`宏的移植。

易见`_print()`函数的功能与 `fmt::Write` trait 中的 `write_fmt()`方法完全相同，所以可将其视为对 `write_fmt()`方法的封装。其实现方式如下。

```
pub fn _print(args: fmt::Arguments) {
    StdOut.write_fmt(args).unwrap();
}
```

`_print()`函数直接将其输入参数作为 `write_fmt()`的参数调用该方法。

此时，将标准库中对 `println!` 宏和 `print!` 宏的实现复制到 IO 模块即可使用该宏定义。

7.5 ELF 解析

操作系统设计的基本功能之一为运行 ELF 格式用户程序，由于 RAM 空间的限制，所以将用户程序与操作系统一同固化至 ROM 中，操作系统仅对 ELF 文件进行解析，而不对其进行加载。内核解析出 entry 后便跳转至该用户程序执行。

ELF 解析功能的实现使用外界提供的 xmas-elf 项目^[20]。xmas-elf 项目提供对 ELF 文件结构的解析与提取、转换功能。新建模块 elf 以实现对用户程序的解析功能。

解析 ELF 需提供 ELF 存储地址 `_elf_payload_start` 与 `_elf_payload_end`。由于用户程序通过“.incbin”宏指令嵌入至引导程序中，所以可直接查看引导程序的反编译结果，从而定位 `_elf_payload_start` 与 `_elf_payload_end` 标号的地址。

使用 Rust core 库 slice 模块中的 `from_raw_parts()` 函数截取从 `_elf_payload_start` 内存开始处的整个 ELF 程序，并存储在 `kernel_elf` 变量中，然后使用 `xmas_elf::ElfFile` 结构体将 `kernel_elf` 变量类型强制转换为 `xmas_elf` 库中定义的 `ElfFile` 结构体类型。此时，便可依据 `xmas_elf` 库中的函数对 ELF 文件进行解析。

`header::sanity_check()` 函数的功能为检查 ELF 文件头，从而检查 ELF 文件正确性与完整性。`program_iter()` 函数的功能为遍历 ELF 文件中各段信息。`header.pt2.entry_point()` 函数的功能为取得 ELF 文件头进而解析出程序入口点。

解析过后，操作系统将得到程序入口点地址并存入变量 `kernel_main` 中。为跳转至用户程序开始执行，需要将该地址转换为函数形式。本实验通过 core 库 mem 模块中实现的 `transmute()` 函数将该变量类型转换为 `extern "C" fn` 函数类型。最后调用转换为函数的 `kernel_main` 即可转入用户程序执行。

7.6 用户程序的编写

为验证操作系统解析运行用户程序的功能，需编写基于该系统的用户程

序进行测试。

由于操作系统并未提供编译和库支持，用户程序的编写同操作系统相同，使用 `bare metal` 形式编写。创建用户可执行程序 `hello`。

```
$ cargo new hello
```

新建 `json` 文件，内容与操作系统 `json` 文件相同。

去除标准库链接，使用汇编指令设置堆栈然后跳转至 `main` 函数执行。引入 `bb1` 库支持，新建 `io` 模块作为输入输出模块，同 7.3 节所示。在 `main` 函数中使用 `println!` 宏输出字符串“Hello World from ELF!”

7.7 程序的链接与加载

`picorv32` CPU 并未实现 MMU，即内存管理单元，加之内存资源相当局限，所以操作系统无法实现内存的管理功能。本实验中的所有程序，使用的内存地址均为物理地址，对内存空间的组织分配均手动进行，并在链接脚本中指定。

7.7.1 引导程序的内存分配

操作系统与用户程序最终都将嵌入至引导程序特定的地址标号处，故从本质上看，对引导程序的地址空间分配即为对全局的地址空间进行分配管理。

在引入操作系统与用户程序之前，先对引导程序进行链接，链接脚本内容详见附录 B。在链接脚本中使用“`MEMORY`”命令描述目标平台上内存块的位置与长度，`ROM` 用于存储可执行代码段和只读数据，`RAM` 用于存储数据段和 `bss` 段，其起始地址为硬件平台中 `ROM` 和 `RAM` 的起始地址，大小则为 `ROM` 和 `RAM` 的大小。通过“`SECTIONS`”命令指定编译目标文件的段名。此处，仅需定义 `text`，`data`，`bss` 三个段即可。由于 `RAM` 在开始时无法进行初始化，且引导程序并没有加载程序能够用来加载其数据段。故将使用“`AT`”命令指定 `data` 段的本地加载地址 `LMA` 为 `text` 段中的 `_sidata` 标号。

7.7.2 操作系统的链接

引导程序在进行寄存器和设备的初始化后直接跳转至加载操作系统可执行代码段的标号 `text_payload_start` 处运行。引导程序编译为 ELF 可执行文件后，通过 RISC-V 交叉编译工具链反汇编工具 `objdump` 查看其反汇编代码，指令如下。

```
$ riscv-unknown-elf-objdump -d firmware.elf | less
```

查看 `text_payload_start` 的地址，此地址即为操作系统链接脚本中可执行代码段的起始地址。同理可查看操作系统数据段地址起始地址 `data_payload_start`。

为了方便地址管理，将 `text_payload_start` 与 `data_payload_start` 的地址取整，使用汇编伪指令 `“ .org ”` 为标号起始地址指定偏移量，其中 `text_payload_start` 偏移量为 `0x1000`，`data_payload_start` 偏移量为 `0x300`。此时便可确定操作系统代码段和数据段的链接地址。

在操作系统项目 `xy_os` 工程中新建链接脚本 `linker.ld` 文件，其内容详见附录 C。该脚本 ROM 用于存储可执行代码段和只读数据，RAM 用于存储数据段和 `bss` 段，其起始地址即为所计算的标号偏移量加上 ROM 和 RAM 的起始地址，大小则依据操作系统实际所需分配，可重复调整直至编译成功。

通过 `objcopy` 将操作系统的代码段和数据段提取为 `xy_os.text` 和 `xy_os.data` 两个文件，即

```
$ riscv32-unknown-elf-objcopy -O binary -j .text kernel xy_os.text
```

```
$ riscv32-unknown-elf-objcopy -O binary -j .data kernel xy_os.data
```

将两个文件复制到引导程序项目目录下与引导程序一同编译。

关于为何将操作系统的代码段和数据段分别链接至引导程序不同区域，而不直接链接 ELF 程序的解释。首先将代码段与数据段分开，引导程序则可省去解析 ELF 程序的功能，其次由引导程序直接跳转至内核可执行代码段执行，内核的链接地址也更容易计算给定，反例则可参考 7.7.3 节对用户程序链接地址的计算。最后由于引导程序数据段加载的特殊性，将内核数据段单独链接至引导程序也更便于加载。

7.7.3 用户程序的链接

用户程序直接通过 ELF 格式文件嵌入引导程序 `_elf_payload_start` 标号起始处，通过反汇编引导程序可查看 `_elf_payload_start` 标号起始地址。

此时，由于 `_elf_payload_start` 标号地址与程序可执行代码段起始地址不同，故无法直接将标号地址作为用户程序的链接地址。通过查看引导程序反汇编代码可以定位用户程序用以初始化堆栈的代码，再与用户程序源程序对比，即可确定此时用户程序可执行代码段的起始地址。

但是此时仍然无法将该地址作为用户程序的链接地址，原因是若据此修改用户程序链接脚本，该程序生成的 ELF 文件结构将发生改变，进而导致将用户程序重新嵌入至引导程序相同地址时，可执行代码段的偏移量也会随之改变，则当操作系统在解析完成该用户程序的 `entry` 时，跳转的地址为旧的可执行代码段地址，从而无法正常执行。

解决方法为固定用户程序中链接脚本可执行代码段的位置，并将其取整对齐。同时也固定 `_elf_payload_start` 标号的位置，重新编译。反汇编引导程序代码，查看其可执行代码段地址。此时引导程序中用户程序的可执行代码段地址必然与链接脚本中的给定地址存在偏移。

假设用户程序链接脚本中可执行代码段地址为 $addr_{ld}$ ，引导程序中可执行代码段地址为 $addr_{load}$ ，则其偏移量 $offset$ 的计算方式如式 7-1。

$$offset = addr_{ld} - addr_{load} \quad (7-1)$$

为使两地址偏移量为零，需要在保证用户程序不变的情况下，修改引导程序中 `_elf_payload_start` 标号的地址。引导程序中 `_elf_payload_start` 标号地址的计算方式如式 7-2。

$$addr_{elf} = addr_{elf} + offset \quad (7-2)$$

使用 `.org` 伪指令设置 `_elf_payload_start` 标号的地址值为 $addr_{elf}$ 。

7.7.4 地址空间布局

结合以上对于整个系统地址空间的分配管理可得到布局如图 7-1 所示

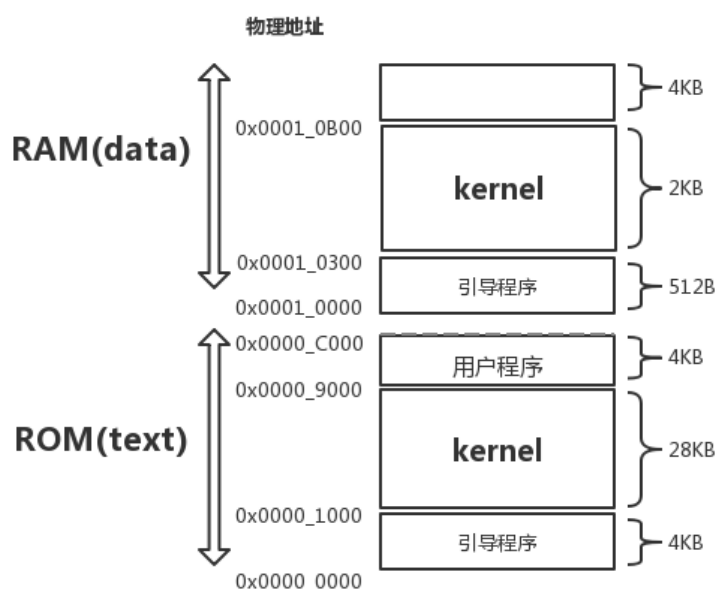


图 7-1 系统地址空间布局

7.8 本章小结

本章内容为介绍操作系统内核的分析与移植过程。首先对在 STEP-MAX 开发板上搭建的 SoC 移植 rCore 的可行性进行了分析，说明系统裁剪后可保留的功能。其次以 rCore 的开发过程文档为模板，介绍了从最小化内核到系统调用再到格式化输出实现的全过程。然后介绍 ELF 程序解析和用户程序的设计。最后介绍引导程序、内核、用户程序的链接加载过程，并给出系统的地址空间布局。

第8章 实验成果与展望

8.1 实验成果展示

（1）串口测试程序。测试程序运行的预期效果为，设备初始化后输出“Booting.....”信息，然后进入轮询程序等待串口输入，若接收到数据，则通过串口将接收的信息输出并打印于串口工具终端。测试效果如图 8-1 所示。



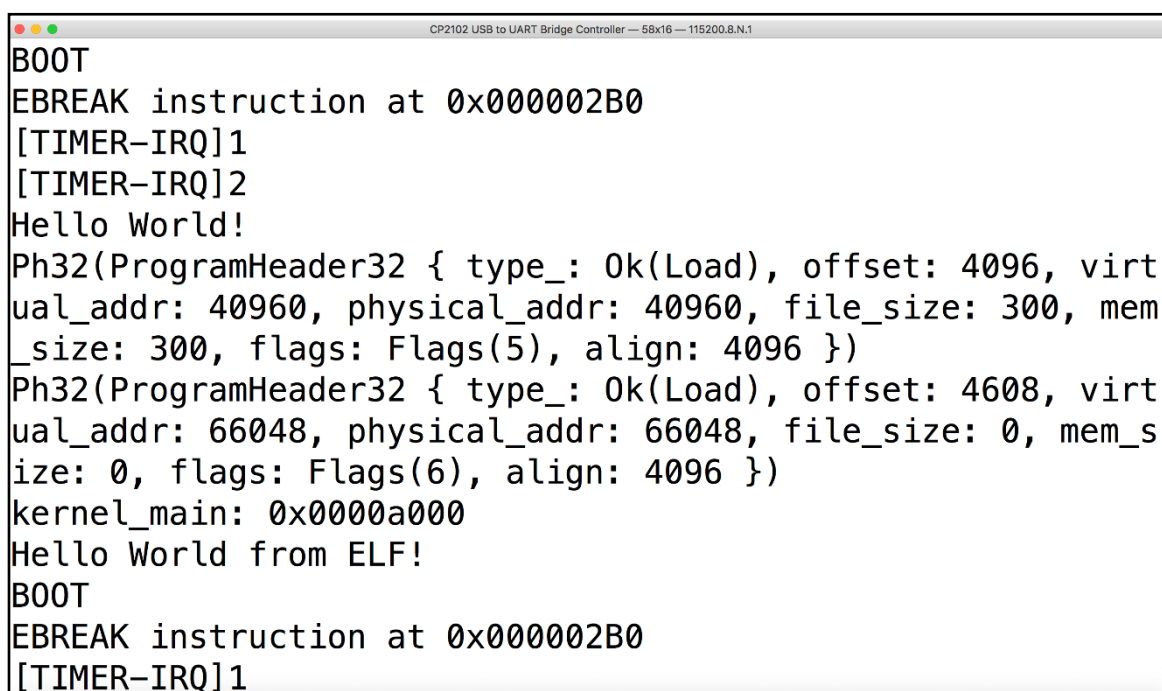
图 8-1 串口测试程序效果

（2）RISC-V 测试程序。程序运行预期效果为，通过指令测试后则将该指令的名称输出并打印“OK”，若测试未通过则打印“ERROR”。执行完测试程序后将执行 EBREAK 指令触发中断。测试效果如图 8-2 所示。

```
lui..OK      bne..OK
auipc..OK    blt..OK
j..OK        bge..OK
jal..OK      bltu..OK
jalr..OK     bgeu..OK
beq..OK      lb..OK
bne..OK      lh..OK
blt..OK      lw..OK
bge..OK      lbu..OK
bltu..OK     lhu..OK
bgeu..OK     sb..OK
lb..OK       sh..OK
lh..OK       sw..OK
lw..OK       addi..OK
lbu..OK      slti..OK
lhu..OK      xori..OK
sb..OK       ori..OK
sh..OK       andi..OK
-----
EBREAK instruction at 0x000006C4
pc 000006C7 x8 00000000 x16 07ECBD6D x24 00000000
x1 00000694 x9 00000000 x17 002E8EB7 x25 00000000
x2 00010000 x10 20000000 x18 00000000 x26 00000000
x3 DEADBEEF x11 075BCD15 x19 000059A8 x27 00000000
x4 DEADBEEF x12 0000004F x20 00000000 x28 00000001
x5 00000DE8 x13 0000004E x21 00000000 x29 7C235965
x6 002E8EB7 x14 00000045 x22 00000000 x30 84A97421
x7 002E8EB7 x15 0000000A x23 00000000 x31 00000000
-----
Number of fast external IRQs counted: 37
Number of slow external IRQs counted: 4
Number of timer IRQs counted: 22
TRAP after 332120 clock cycles
ALL TESTS PASSED.
z@z-X555LI:~/Desktop/picorv32$ █
```

图 8-2 RISC-V 测试程序效果

(3) rCore 与用户程序。程序运行预期效果为，设备初始化后打印“BOOT”，执行 EBREAK 指令测试中断处理程序是否正常。开启时钟中断，10000 时钟周期触发一次并打印 “[TIMER-IRQ]n”。进入 rCore 后调用 println! 宏打印 “Hello World!”，然后解析用户 ELF 程序，将各 section 信息打印，最后跳转至用户程序运行，打印 “Hello World from ELF!”。程序运行效果如图 8-3 所示。



```
BOOT
EBREAK instruction at 0x000002B0
[TIMER-IRQ]1
[TIMER-IRQ]2
Hello World!
Ph32(ProgramHeader32 { type_: 0k(Load), offset: 4096, virtual_addr: 40960, physical_addr: 40960, file_size: 300, mem_size: 300, flags: Flags(5), align: 4096 })
Ph32(ProgramHeader32 { type_: 0k(Load), offset: 4608, virtual_addr: 66048, physical_addr: 66048, file_size: 0, mem_size: 0, flags: Flags(6), align: 4096 })
kernel_main: 0x0000a000
Hello World from ELF!
BOOT
EBREAK instruction at 0x000002B0
[TIMER-IRQ]1
```

图 8-3 rCore 与用户程序效果

8.2 未来展望

本实验中，由于 picorv32 并未实现标准的 RISC-V 特权极架构，所以在移植 rCore 操作系统的过程中遇到了许多的障碍。由于在搭建 SoC 时，使用了 Quartus 中的 Platform Designer 工具，picorv32 被封装为 AXI-Lite 接口的 IP 核，故若能够找到比 picorv32 更加适合的 CPU，则只需将该 CPU 封装为 Platform Designer 支持的接口 IP 核即可替换 picorv32，其余外设和地址空间均可直接使用。在 4.2.5 节中介绍了 picorv32 的 PCPI 接口，若希望对 picorv32 的功能进行拓展，则可实现一个协处理器并通过 PCPI 接口与 picorv32 连接，比如可通过协处理器方式为 picorv32 实现标准的 RISC-V 特

权极架构。此外，本实验在编写软件过程中遇到的最大问题便是内存空间的不足，picorv32 实现了 C 模块的 RISC-V 指令，若在相同的内存大小中，使用压缩指令集可节省将近一半的内存资源，此处唯一的障碍为开启不定长的指令集后，在取值的访存实现上会遇到些许困难。

结 论

本实验以系统构建为目的，分别从硬件和软件两个方面入手，综合运用各专业课程所学理论知识，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指令集架构——RISC-V CPU 的微型硬件系统，除具备 CPU、存储运行程序的 ROM 和 RAM 外，还具有数码管、LED 和串口等外部设备，为系统与上位机的交互提供支持。实验通过更改 rCore 操作系统的底层硬件支持和裁剪功能，仅保留格式化输出，中断处理，ELF 文件解析的功能，最终实现对操作系统的移植。

实验不仅能在仿真环境下对系统功能进行测试，更能在 STEP-MAX10 开发板的真实环境中对硬件和软件系统的功能进行验证。在实验过程中，使用了多种设备对系统所出现的问题和错误进行调试定位，包括 Modelsim 仿真工具，示波器等，最终得以使系统按照预期效果运行。本实验通过克服硬件资源的限制，对系统各部件进行精心地组织和安排，有效地利用了开发板的硬件资源，包括使用预留用户 IO 接口实现串口功能，划分 Block Memory 实现 ROM 和 RAM 的分配。在软件开发过程中，使用原始的内存管理和组织模式，在物理内存的基础上对引导程序、系统内核、用户程序进行链接装载。虽然软件系统的功能比较粗糙，但也体现了贴近底层的系统软件的运行原理，无疑是一个比较完善的系统模型。总之，本次实验所搭建的硬件和软件系统尽可能地运用各课程所学知识，有效地对课程内容进行衔接，并充分证明诸如 STEP-MAX10 的小型开发板足以具备构建和开发复杂硬件和软件系统的能力。

参考文献

- [1] RISC-V Foundation. About the RISC-V Foundation [EB/OL]. <https://riscv.org/risc-v-foundation>. 2019-05-22
- [2] 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
- [3] 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). 2019-01-17/2019-05-22
- [4] Clifford. PicoRV32 - A Size-Optimized RISC-V CPU [CP/OL]. <https://github.com/cliffordwolf/picorv32>. 2019-03-28/2019-05-22
- [5] Rust 程序设计语言 简体中文版 [EB/OL]. <https://kaisery.github.io/trpl-zh-cn/ch01-01-installation.html>. 2019-05-22
- [6] 卢旺杉. 如何看待 Rust 的应用前景 [EB/OL]. <https://www.zhihu.com/question/30407715/answer/48032883>. 2015-05-27/2019-05-22
- [7] rCore OS 开发文档 [EB/OL]. <https://rcore.gitbook.io/rust-os-docs>. 2019-01-22/2019-05-22
- [8] STEP-MAX10 [S/OL]. <http://www.stepfpga.com/doc/step-max10>. 2019-05-22
- [9] Rob Landley. Introduction to cross-compiling for Linux [EB/OL]. <http://landley.net/writing/docs/cross-compiling.html>. 2019-05-22
- [10] RISC-V GNU Compiler Toolchain [CP/OL]. <https://github.com/riscv/riscv-gnu-toolchain>. 2018-01-03/2019-05-22
- [11] RISC-V Proxy Kernel and Boot Loader [CP/OL]. <https://github.com/riscv/riscv-pk>. 2019-04-14/2019-05-22
- [12] 硅农亚历山大. RISC-V 嵌入式开发入门篇 1: RISC-V GCC 工具链的介绍 [EB/OL]. <https://mp.weixin.qq.com/s/QayXAQPkOcpeEMbOwPa1Ww>. 2018-06-12/2019-05-22
- [13] Intel® Quartus® Prime Standard Edition User Guide [S/OL]. <https://www.intel.com/content/www/us/en/programmable/quartusprime/documentation/default.aspx>

- w.intel.com/content/www/us/en/programmable/documentation/jrw1529444674987.html. 2019-05-14/2019-05-22
- [14] Embedded Peripherals IP User Guide [S/OL]. <https://www.intel.cn/content/www/cn/zh/programmable/documentation/ntt1529445293791.html>. 2019-04-01/2019-05-22
- [15] The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10 Document Version 1.10 [S/OL]. <https://riscv.org/specifications/privileged-isa>. 2019-05-22
- [16] 刘丰源、潘庆霖. 从零开始写 OS [EB/OL]. <https://xy-plus.gitbook.io/rcore-step-by-step>. 2019-05-22
- [17] 王润基等. rCore [CP/OL]. <https://github.com/rcore-os/rCore>. 2019-05-09/2019-05-22
- [18] Function core::fmt::write 1.0.0 [S/OL]. <https://doc.rust-lang.org/core/fmt/fn.write.html>. 2019-05-22
- [19] Philipp Oppermann. Writing an OS in Rust[EB/OL]. <https://os.phil-opp.com>. 2018-02-26/2019-05-22
- [20] xmas-elf [CP/OL]. <https://github.com/nrc/xmas-elf>. 2019-04-24/2019-05-022

附录

附录 A

Intel Hex 格式文件是遵循 Intel Hex 文件格式的 ASCII 文本文件。在 Intel Hex 文件的每一行都包含了一个 Hex 记录。Intel Hex 由任意数量的十六进制记录组成。每个记录包含 5 个域，格式排列如图 A-1 所示。

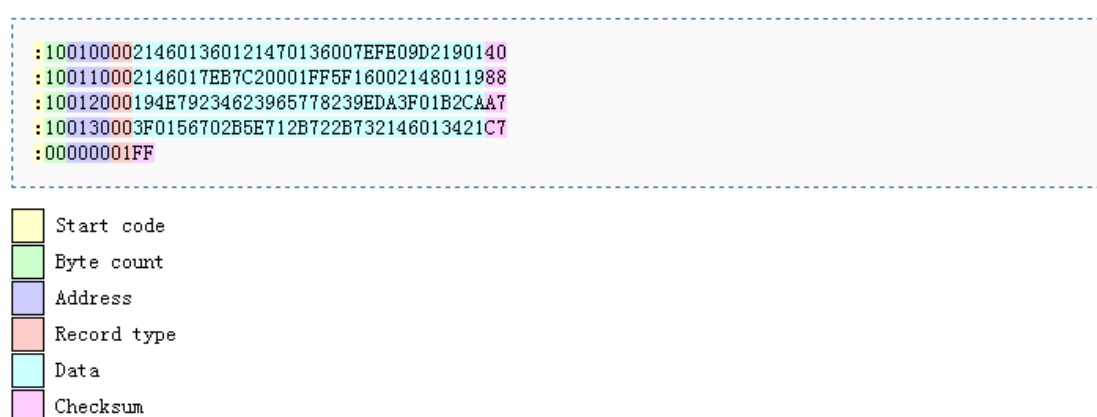


图 A-1

Start Code 起始位，每个 Intel Hex 记录以冒号作为起始位。

Byte count 数据长度域，代表每条记录数据字节的数量。

Address 地址域，代表记录中数据的起始地址

Record type 记录类型域，其值的含义如下。

- | | |
|-------------|-------------|
| 00-数据记录 | 01-文件结束记录 |
| 02-扩展段地址记录 | 03-开始段地址记录 |
| 04-扩展线性地址记录 | 05-开始线性地址记录 |

Data 数据域，一个记录可以有多个数据字节。

Checksum 校验和域，代表记录的校验和。校验和的值为将记录当中所有十六进制编码字节的值相加，以 256 为模进行补足。

附录 B

引导程序链接脚本

```
ENTRY(reset_vec)
MEMORY
{
    ROM (rx)      : ORIGIN = 0x00000000, LENGTH = 0xc000
    RAM (xrw)     : ORIGIN = 0x00010000, LENGTH = 0x1000
}
SECTIONS {
    .text : {
        . = ALIGN(4);
        *(.text) *(.text*) *(.rodata) *(.rodata*)
        *(.srodata) *(.srodata*)
        . = ALIGN(4);
        _etext = .;
        _sidata = _etext;
    } >ROM
    .data : AT ( _sidata )
    {
        . = ALIGN(4);
        _sdata = .;
        _ram_start = .;
        . = ALIGN(4);
        *(.data) *(.data*) *(.sdata) *(.sdata*)
        . = ALIGN(4);
        _edata = .;
    } >RAM
    .bss : {
```



```
. = ALIGN(4);  
_sbss = .;  
*(.bss) *(.bss*) *(.sbss) *(.sbss*)  
*(COMMON)  
. = ALIGN(4);  
_ebss = .;  
} >RAM  
}
```

附录 C

操作系统链接脚本

```
ENTRY(_start)

MEMORY
{
    ROM (rx)      : ORIGIN = 0x00001000, LENGTH = 0x8000
    RAM (xrw)     : ORIGIN = 0x00010300, LENGTH = 0x800
}

SECTIONS {
    .text :
    {
        . = ALIGN(4);
        *(.text) *(.text*) *(.rodata)
        *(.rodata*) *(.srodata) *(.srodata*)
        . = ALIGN(4);
        _etext = .;
        _sdata = _etext;
    } >ROM

    .data : AT ( _sdata )
    {
        . = ALIGN(4);
        _sdata = .;
        _ram_start = .;
        . = ALIGN(4);
        *(.data) *(.data*) *(.sdata) *(.sdata*)
        . = ALIGN(4);
        _edata = .;
    } >RAM
```

```
.bss :
{
    . = ALIGN(4);
    _sbss = .;
    *(.bss) *(.bss*) *(.sbss) *(.sbss*)
    *(COMMON)
    . = ALIGN(4);
    _ebss = .;
} >RAM

.heap :
{
    . = ALIGN(4);
    _heap_start = .;
} >RAM
}
```

致 谢

匆匆四年，转瞬而过。回想入学时，那个一无所知的自己，仿佛 C 语言课上那个没能听懂老师讲 `main` 函数的少年就坐在自己面前发呆感叹。记得大一下学期，被 OJ 支配的恐惧深深笼罩一学期的我跑到导员办公室问他，学计算机能不能不码代码，不搞软件开发？他告诉我说，那你还是尽快转专业吧！我不甘心，难道除了软件，就没有硬件方向可以选择吗？四年来，我一直在迷惘与思考，在历尽数次抉择之后，最终终于逃离了软件开发，走上了硬件，体系结构的道路，然而却没能逃脱码代码的命运。但是，走到今天，回过头来才发现，原先以为的软件开发并不是我当年认为的软件，硬件也不是我想象中的硬件，码代码也不是我想的那种天天面对 OJ 想算法，调 Bug 的码代码，自己也不是那个学 C 语言都觉得费劲的自己。四年，足够一个人学习很多，成长很多。从 C 语言，到 C++，java，python，verilog、Rust。从 x86 到 mips，再到 risc-v。视野越来越宽阔，知识却越学越多。无论如何，当初的疑惑已经慢慢解开，心里也为自己能把计算机搞明白感到开心和舒畅，居然有些小庆幸自己当初选择了计算机这条道路，并且能够坚持下来。

感谢北理工为我提供的平台和机遇，让我这条小鱼在这偌大的池子里畅游无阻！感谢每位辛勤付出的老师，给予我穿越江河湖海的能力和志气！感谢身边的同学和朋友，陪伴我渡过每一个湍流，推促我不断向前！感谢裴明涛老师对我大创工作的指导！感谢李凡老师和实验室的师兄师姐，带我参加物联网挑战赛！感谢向勇、陆慧梅、王娟三位老师，尽心组织龙芯杯比赛，为我打开进入体系结构的大门！感谢龙芯杯的举办，使我在对的时间遇上一群志同道合的人，让我的能力得到突飞猛进的增长！感谢江学谦，耐心专业地为我解答了所有专业知识问题，从他身上我学到了无比多的技能，知道什么是对计算机真正的热爱！感谢向老师，在毕业设计过程中孜孜不倦的指导，他的敬业精神和对挑战的追求深刻的影响着我！感谢陆老师，带我学习养生和哲学之道！感谢清华大学 rCore 开发小组，不仅为我提供了实验素材，更在我每每想要骄傲一下的时候，把我拉回现实的角落！感谢周育聪，给我四年带来许多大脑洞的欢乐！

感谢肖子原，陪我准备保研过程各种材料！感谢李霞学姐和刘雪松学长，带我逛熟清华的各个食堂！最后，感谢我的父母，给我莫大的支持和以我为傲的期望！

现实免不了有遗憾和惋惜。有舍友帮我用 Dev-C++ 轻敲出 Hello World 程序而我却不屑于把屏幕截取保存下来的遗憾。有为了完成小学期代码任务，把身体熬坏留下后遗症的遗憾。有为追求更多技术，没能静下心来好好深入学习一门语言和培养工程能力的遗憾。虽然坎坷不可避免，但我相信，人生，总是不断曲折前进的。努力去追求更好的自己，终有一天，你总会达到你应有的高度！