电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER THESIS

论文题目 基于 RISC-V 的 SoC 设计及其 RTOS 移植

学科专业 微电子学与固体电子学

学　　号 201721030201

作者姓名 邓紫珊

指导教师 翟亚红　副教授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 RISC-V 的 SoC 设计及其 RTOS 移植

邓紫珊

指导教师 翟亚红 副教授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 学科专业 微电子学与固体电子学

提交论文日期 2020 年 5 月 15 日 论文答辩日期 2020 年 5 月 28 日

学位授予单位和日期 电子科技大学 2020 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

RISC-V-BASED SOC DESIGN AND ITS RTOS MIGRATION

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Discipline: **Microelectronics & Solid-State Electronics**

Author: **Zishan Deng**

Supervisor: **A/Prof. Yahong Zhai**

School: **School of Electronic Science and Engineering**

摘　要

由高级语言编写的程序，通过相应的某种编译器，根据相应的指令集架构，可以被翻译成能被相应的处理器实现（Implementation）所识别的指令。如今我们所能见到的处理器基本上都采用了 x86 与 ARM 这两种指令集架构，漫长的发展过程使这两种架构足够成熟，也使得它们难以避免地出现许多问题。不少设计者对 x86 与 ARM 架构那过于复杂的指令集、昂贵的商业授权和难以获取的源码颇有微词，在这样的情况下，全新的指令集架构 RISC-V 逐渐吸引了设计者们的注意。RISC-V 提供了免费开源、开发周期较短的处理器实现方案。面对国外芯片的生态和专利壁垒，RISC-V 有望成为我国自主研制处理器芯片的一个极好的选择。

本次设计主要通过对 RISC-V 官方提供的参考处理器实现项目 Rocket Chip 的研究，构建了基于 RISC-V 的 SoC，首先对基于 Rocket Chip 的 SoC 的前端设计进行了研究，基于 0.13μm 工艺，通过逻辑综合和后端物理设计完成了 SoC 的物理实现，然后由 Rocket Chip 生成的软件模拟器初步对 SoC 的功能进行仿真，基于 Xilinx ARTY A7 开发板，将构建得到的 SoC 用 FPGA 实现，并对其进行原型验证，最后在基于 FPGA 的 RISC-V 平台上运行了 FreeRTOS，实现了 RISC-V SoC 的操作系统移植。本次设计主要完成了以下工作：

1) 对 Rocket Chip 进行了研究。分析研究了 Rocket Chip 项目的架构，搭建了 RISC-V 交叉编译工具链，借由 Rocket Chip 项目生成基于 RISC-V 的 SoC，利用软件模拟器和一个简单的测试程序对其进行软件模拟，初步验证了其功能。

2) 对构建得到的 SoC 的后端物理设计进行了研究。利用 Design Compiler 进行逻辑综合，将设计从 RTL 代码转换为门级网表，并通过了时序检查和形式验证，然后利用 IC Compiler 工具完成设计的后端物理设计，并通过了时序检查、物理验证和形式验证，得到最终的设计版图。

3) 对构建得到的 SoC 的 FPGA 实现进行了研究。利用 Vivado 工具建立工程并对其进行综合，利用 Xilinx ARTY A7 开发板将其用 FPGA 实现，并进行原型验证。

4) 考虑到嵌入式开发常需要实时操作系统来提高开发效率，对 RISC-V 平台上的操作系统移植进行了研究。基于 FreeRTOS 项目的源码，编写了一个实例，利用 Xilinx ARTY A7 开发板实现了基于 FPGA 的 RISC-V 平台上的 FreeRTOS 移植。

关键词：RISC-V，Rocket Chip，FPGA，物理设计，操作系统移植

ABSTRACT

According to a corresponding instruction set architecture, a program written in a high-level programming language can be translated into instructions by a corresponding compiler, and the instuctions can be recognized by a corresponding processor. Nowadays processors are mainly based on two instruction set architectures, x86 and ARM. The long development history has made these two architectures mature enough, but also brought some unavoidable problems. Many designers have complained about the overly complicated instruction set, expensive commercial license, and source code that cannot be modified or even visible of the x86 and ARM architecture. Under the circumstances, a new instruction set architecture, RISC-V gradually attracts designers' attention. RISC-V provides a free and open source processor implementation with a relatively short development period. Facing the software ecosystem and patent barriers of foreign chips, RISC-V is expected to become an excellent choice for China to independently develop processor chips.

The paper is mainly based on the research of the reference RISC-V processor, Rocket Chip, provided by the research team of RISC-V, and implement a RISC-V-based SoC. Firstly the paper studies the front-end design of the Rocket Chip-based SoC. The physical implementation of the SoC is accomplished through logic synthesis and back-end physical design. Then the function of the SoC is initially simulated by the generated software simulator. Based on the Xilinx ARTY A7 development board, the obtained SoC is implemented with FPGA and passes the prototype verification. Finally FreeRTOS is migrated on the FPGA-based RISC-V platform. The paper can be summarized as follows:

1) The Rocket Chip Generator project is studied. Analyzing the architecture of the Rocket Chip project, we builds the RISC-V gnu toolchain and obtain a RISC-V-based SoC, and then utilize the generated software simulator and a simple test program to perform functional verification.

2) The physical design flow of the generated SoC is studied. Utilizing Design Compiler for logic synthesis, we convert the design from RTL to gate-level netlist, and timing check and formal verification is passed. And then we use the IC Compiler tool to complete the physical implementation of the design, and timing check, physical

verification, and formal verification is passed, and we obtain the final design layout.

3) The FPGA implementation of the obtained SoC is studied. Using Vivado to build and synthesize the project, we utilize Xilinx ARTY A7 development board to implement the synthesized design with FPGA, and perform the prototype verification.

4) Considering that embedded software development often requires real-time operating systems to improve development efficiency, the operating system migration on the RISC-V platform is studied. Based on the source code of the FreeRTOS project, we build a demo and run it in the FPGA-based RISC-V platform.

Keywords: RISC-V, Rocket Chip, FPGA, physical design, OS migration

目 录

第一章 绪 论

1.1 研究工作的背景与意义

我们都知道，计算机只能识别和执行机器指令，也就是数字“1”和“0”组成的二进制代码，而现在大家都使用便于理解的高级语言，比如 C、JAVA 等，来编写程序，从而方便地实现我们想要的功能。那么对人类来说可读性好的语言是如何翻译成机器语言，让机器可以理解相应的操作的呢？这就需要设计所谓的指令集，基于指令集去构建相应的指令集架构（Instruction Set Architecture，ISA）。由高级语言编写的程序，通过相应的某种编译器，根据相应的指令集架构，可以被翻译成能被机器识别的指令。而机器如何执行指令，一条指令如何在硬件上实现它所表述的操作，这就是所谓的微处理器体系架构。具体地从物理上实现某种指令有多种方式，也就是说，一种指令集架构可以对应多种微处理器结构。

最初的指令集的提出是基于当初的计算机性能不足、需要提高运算效率的需求，指令集的出现大大提高了设计者们的开发效率，也是出于这个原因，人们希望一条指令可以控制计算机处理尽量多的操作，这就使得指令变得越来越复杂，指令集的体量随之大幅增长。然而，复杂的指令越来越多，其使用率实则远低于简单的指令，过于全面的指令集反而成了一种负担。出于这种新的需求，人们设计了新的指令集，只保留了原来的指令集中被反复使用的简单指令，指令集被大大的简化了。为了区分这两种指令集，之前相对复杂的指令集被称为 CISC（Complex Instruction Set Computer，复杂指令集计算机），后来精简后的指令集被称为 RISC（Reduced Instruction Set Computer，精简指令集计算机）[1]。从硬件的角度上，为了实现相应的指令集，处理器的设计也基于这两个体系各自向前发展。

发展到今天，常见的处理器基本上被 x86 与 ARM 这两种指令集架构所占据。我们所使用的个人电脑绝大多数都是基于 x86 架构的，而 x86 架构其实归属于 CISC。Intel 公司在 1978 年推出了第一个基于 x86 指令集架构的 CPU“8086”[2]。1981 年，IBM 公司推出了基于 Intel 的 x86 处理器的个人电脑[3]。20 世纪 80 年代后，基于 Intel 的处理器的个人电脑上普遍安装微软的 Windows 操作系统，二者几乎组成了固定搭配，硬件设计与软件开发齐头并进，迅速占据了绝大部分市场份额，如今，个人电脑的用户已经习惯了这个搭配，也习惯了四十年来发展起来的完善的生态环境[4]。但多年以来的发展也使得 x86 架构越来越复杂和冗余，为了保持兼容性，设计者们研发新的处理器时，不仅要实现现有的 ISA，还要去实现所有的过往的 ISA 扩展，哪怕那已经过时甚至是错误的。为此，基于 CISC 的 x86 处理器也会借鉴和引入一些 RISC 的思想，从而对自身做出改进[5]。

ARM 架构则广泛应用于移动通讯领域，目前我们所使用的手机基本都是基于 ARM 架构。ARM 指令集架构属于 RISC 体系，ARM 公司于 1985 年推出了第一款 ARM 处理器内核[6]。ARM 选择提供 IP 核，授权给其他芯片公司去生产处理器芯片的开放模式，逐渐形成基于 ARM 架构的生态圈，占据了服务器、嵌入式和移动终端的市场。谷歌推出的安卓系统，苹果、高通、三星公司推出的处理器，基本都是基于 ARM 指令集架构来实现的。

漫长的发展过程使 x86 和 ARM 架构成为非常成熟的架构，也使得它们难以避免地出现许多问题，不论是其指令集的复杂和冗余，抑或是昂贵的商业授权，都对设计者造成了极大的限制，更何况 x86 处理器和 ARM 处理器的源码是限制修改而且难以获取的。对免费和开放的处理器架构的需求，使各种开源架构进入了设计者们的视线。

Sun Microsystems 公司于 1986 年推出了 SPARC 处理器，SPARC 架构完全对外开放，成为经典的 RISC 处理器体系架构之一[7]。SPARC 处理器曾一度统治过服务器和工作站市场，但没能对 x86 处理器和 Windows 系统这个搭配挑战成功，后来逐渐消失在人们的视野当中。

开源社区 Opencores 基于 RISC 指令集架构推出了面向嵌入式系统的 OpenRISC 处理器，其源代码开放，且提供了一套完整的开发工具链，许多商业公司和研究机构都将 OpenRISC 体系架构引入了自己的开发和研究中去[8,9]。但是作为一个社区推动的项目，OpenRISC 有些缺乏管理，各种文档的跟进和更新不够及时，再加上 GPL（GNU General Public License，GNU 通用公共许可证）协议下开源的项目无法做到商业化，都在一定程度上制约了 OpenRISC 的发展。

后来，全新的指令集架构 RISC-V 于 2010 年诞生在加州大学伯克利分校，Krste Asanović 教授等开发人员最初推出 RISC-V 架构其实是为了开展课程和研究，之后这个优秀的指令集架构渐渐吸引了外部人员的注意[10-12]。2015 年，RISC-V 基金会启动，旨在推动基于 RISC-V 架构的软件设计和硬件开发的协同创新[12]。2018 年 RISC-V 基金会宣布与 Linux 基金会合作，得到了 Linux 这个庞大的开源软件生态的支持[13]。发展到今天，RISC-V 基金会的三百多个成员遍布全球各地，阿里巴巴、谷歌、高通、三星、西部数据等等都是其中的成员[12]。

RISC-V 为软件开发者提供了免费的指令集规范，硬件设计者根据规范去实现满足所需的处理器，这种通过开放标准和开放源代码来让广大设计者参与处理器创新的方式极大地降低了处理器设计的门槛，有望在新兴的 AI 与 IoT 领域中对 ARM 的统治地位形成挑战，也为中国在芯片设计领域打破国外技术垄断提供了很好的机会。

1.2 RISC-V 处理器国内外研究历史与现状

基于 RISC-V 指令集架构既可以去设计开源的处理器，也支持商业模式，开发周期也相对短暂，无论是对中小型公司和个人设计者，还是对知名的芯片公司和互联网企业，都可以作为开发产品的好选择。

RISC-V 指令集架构的官方处理器实现项目名为 Rocket Chip，一个完全开源的 RISC-V 处理器生成器，由加州大学伯克利分校的 RISC-V 研发团队推出，也是最符合规范的硬件实现，源代码托管在 GitHub 平台上[14,15]。Rocket Chip 是一个开源的项目，使用开源许可证 BSD（Berkeley Software Distribution，伯克利软件套件），用 Chisel（Constructing Hardware In a Scala Embedded Language，嵌入 Scala 的硬件描述语言）编写，可以用来生成 Rocket 内核和 BOOM 内核，对它的详细介绍可见第二章。该团队还推出了基于 Rocket Chip 的开源 SoC 项目 freedom，其源代码同样托管在 GitHub 平台上[16]。本次设计主要参考这两个项目进行。

谷歌公司非官方地推出了一个开源 RISC-V 产品，BottleRocket，源代码托管在谷歌的 GitHub 官网上[17]。BottleRocket 是一个基于 Rocket Chip 的自定义的微处理器结构，实现了 RV32IMC 指令集架构，也就是基础指令集 RV32I 和扩展指令集 RV32M 和 RV32C。BottleRocket 希望用尽量简单的设计实现 RISC-V，只使用了 Rocket Chip 的几个关键部件，包括指令译码器和 CSR（Control Status Register，控制状态寄存器）。

谷歌公司还发起了开源项目 OpenTitan，将管理权交由 lowRISC，通过与诸多院校和公司的合作，旨在基于 zero-riscy 内核设计通用的芯片加密模块，苏黎世联邦理工学院、捷德集团、新唐科技和西部数据都是该项目的成员[18]。

lowRISC 是一个由剑桥大学的工程师团队组成的非营利性组织，推出了基于 Rocket 内核的 64 位开源 SoC 平台 lowRISC chip，实现了 RV32IM 指令集架构，正在计划增加对苏黎世联邦理工学院开发的 64 位内核 Ariane 的支持[19]。

苏黎世联邦理工学院与波罗尼亚大学一同推出开源 PULP（Parallel Ultra-Low-Power）平台，使用项目团队开发的 32 位 RISC-V 处理器内核 CV32E40P 或者 Ibex，或者 64 位的 RISC-V 处理器内核 Ariane 作为主核。CV32E40P 本是基于 OpenRISC 架构的 OR10N 处理器核心的一部分，后来被 PULP 团队以 RI5CY 为名改成了一个 RISC-V 内核进行使用和维护。CV32E40P 实现了 RV32IM[F]C 指令集架构，是一个有序执行的 RISC-V 处理器内核，包含 4 级流水线。2020 年 2 月被贡献给 OpenHW，现托管在 OpenHW 的 GitHub 官网上[20]。Ibex 由 zero-riscy 内核发展而来，已被贡献给 lowRISC，是针对超低功耗和面积开发的，拥有 2 级流水线，实现了 RV32IMC，现托管在 lowRISC 的 GitHub 官网上[21]。Ariane 是一个

单发的有序执行的 RISC-V 处理器内核，拥有 6 级流水线，实现了 64 位 RISC-V 指令集的所有子集和扩展，现托管在 PULP 项目的 GitHub 官网上[22]。

2015 年，加州大学伯克利分校的 RISC-V 研发团队成立了商业公司 SiFive，推出了可由客户自定义的 RISC-V IP 核，以及同样支持定制的 SoC 平台，包括针对低功耗需求的 Freedom Aware 平台，面向 IoT 领域的 Freedom Everywhere 平台，针对高性能需求的 Freedom Unleashed 平台，以及面向 AI 领域的 Freedom Revolution 平台[23]。

晶心科技推出了一系列嵌入式处理器和 SoC 平台，将 RISC-V 吸纳进自己的 AndeStar™ V5 指令集架构，推出了 AndesCore™ 产品线，包括 N25F，NX25/NX25F，A25 和 AX25 处理器内核，推出了 FreeStart AE250、Standard AE250 和 AE350 等 SoC 平台，建立了一套完善的软件和硬件开发环境[24]。

越来越多的国家也开始对 RISC-V 重视起来，RISC-V 本身就是在美国国防高级研究计划局（Defense Advanced Research Projects Agency，DARPA）资助的项目中出现的，印度的几个自研处理器的政府项目都转向了 RISC-V 架构[25]。在中国，RISC-V 也引起了设计者们的极大兴趣。

2018 年 11 月，中国开放指令生态联盟（China RISC-V Alliance，CRAV）成立，旨在基于 RISC-V 建立起中国的开源芯片生态，多个研究机构、企业、院校和个人设计者都是其成员[26]。

蜂鸟 E203 是中国第一个开源 RISC-V 处理器内核，其源代码托管在 GitHub 平台上，为国内 RISC-V 处理器的研究者和院校提供了极好的实践案例，推动了国内 RISC-V 生态的发展[27]。

后来，蜂鸟 E203 的开发者胡振波创立了芯来科技，推出了 N100、N200、N300、N600、N900 等多个系列的 RISC-V 处理器，为多家企业提供了定制 RISC-V 处理器 IP 核。芯来科技与兆易创新合作，为其定制了 RISC-V 处理器内核 Bumblebee，兆易创新基于此推出了 GD32VF103 系列的量产通用微处理器[28]。

阿里平头哥也致力于将 RISC-V 架构应用于自研的玄铁系列处理器上，推出了 64 位 RISC-V 处理器内核 C910 和兼容 RISC-V 的 E902，在 AIoT 领域的生态中占据了重要位置[29]。

此外，华米科技推出了针对可穿戴设备的基于 RISC-V 的 AI 处理器黄山 1 号，于 2019 年 8 月宣布量产，并应用于商业产品 Amazfit 米动健康手环上[30]。

上海交通大学与瓶钵信息科技合作研发了“蓬莱”安全架构，一个可应用于 RISC-V 平台的开源 TEE（Trusted Execution Environment，可信执行环境）系统，同时也丰富了 RISC-V 的指令集扩展，现托管在 GitHub 平台上[31]。

中国科学院计算技术研究所开展了标签化 RISC-V 项目，对 RISC-V 架构进行了探索和研究[32]。

众所周知，芯片产业是投入极高、回报极慢的领域，而 RISC-V 提供了免费开源、开发周期较短的解决方案。面对国外芯片的生态和专利壁垒，RISC-V 有望成为我国自主研制处理器芯片的一个极好的选择，或许能为国产芯片市场的繁荣提供机遇。

表 1-1 部分国内外现有的 RISC-V 处理器和 SoC 平台

名称	发布者	处理器内核	商业模式
Rocket Chip	加州大学伯克利分校	Rocket	开源
BottleRocket	谷歌	Rocket	开源
lowRISC chip	lowRISC	RV32IM	开源
PULP	苏黎世联邦理工学院	CV32E40P，Ibex，Ariane	开源
FE310-G002	SiFive	E31	可授权
FreeStart AE250、Standard AE250	晶心科技	N22	可授权
AE350	晶心科技	N25F, D25F, A25, A25MP, NX25F, AX25, AX25MP	可授权
HBird-E200-SOC	胡振波	蜂鸟 E203	开源
GD32VF103	兆易创新	芯来科技 Bumblebee	封闭
玄铁 C910	阿里平头哥	RV64GCV	可授权
玄铁 E902	阿里平头哥	RV32EMC/IMC/EC	可授权
黄山 1 号	华米科技	-	封闭

1.3 本文的主要内容与结构安排

本次设计主要通过对 RISC-V 官方提供的参考处理器实现项目 Rocket Chip 的研究，构建了基于 RISC-V 的 SoC，首先对基于 Rocket Chip 的 SoC 的前端设计进行了研究，基于代工厂提供的 0.13μm 工艺库，通过逻辑综合和后端物理设计完成了 SoC 的物理实现，然后由生成的软件模拟器初步对 SoC 的功能进行仿真，基于 Xilinx ARTY A7 开发板，将构建得到的 SoC 用 FPGA 实现，并对其进行原型验证，最后在基于 FPGA 的 RISC-V 平台上运行了 FreeRTOS，实现了 RISC-V SoC 的操

作系统移植。

本文的章节结构安排如下：

第一章：简要介绍了课题背景与研究意义，以及 RISC-V 处理器的国内外研究历史与现状。

第二章：首先简要介绍了 Rocket Chip 项目，对其进行了基本的了解，然后分析研究了 Rocket Chip 项目的子模块，了解了其架构，接下来搭建了 RISC-V 交叉编译工具链，并借由一个简单的 C 程序对其进行了测试，最后简要介绍了硬件描述语言 Chisel 以及它的使用。

第三章：首先对基于 Rocket Chip 的 RISC-V SoC 前端设计进行了研究，生成了相应的 RTL，然后利用 Design Compiler 进行逻辑综合，将我们的设计从 RTL 代码转换为门级网表，并通过时序检查和形式验证，最后主要利用 IC Compiler 工具通过数据准备、布图规划、布局、时钟树综合和布线等步骤完成设计的物理实现，并通过时序检查、物理验证和形式验证，得到最终的设计版图。

第四章：首先利用之前生成的软件模拟器借由测试程序对 SoC 的功能进行验证，然后利用 Vivado 工具对其进行综合，利用 Xilinx ARTY A7 开发板将其用 FPGA 实现，并进行原型验证。

第五章：首先简要介绍了实时操作系统和 FreeRTOS，然后基于 FreeRTOS 项目的源码，编写了一个实例，利用 Xilinx ARTY A7 开发板实现了 FreeRTOS 在基于 FPGA 的 RISC-V 平台上的移植。

第六章：对本次设计进行了总结和后续工作的展望。

第二章 Rocket Chip 项目概述

2.1 Rocket Chip 项目介绍

Rocket Chip 是由加州大学伯克利分校开发的，它不是一个单一的 SoC 设计实例，而是一个设计生成器，能够从更高级别的源码中生成多个设计实例[14]。Rocket Chip 有非常灵活的参数化设计，便于我们根据特定的应用场景对其进行定制，我们可以通过仅仅更改一个配置，就得到大小迥异的 SoC，可以是嵌入式微处理器，也可以是多核服务器芯片。Rocket Chip 支持以指令集扩展、协处理器或完全独立的新核的形式来组成定制加速器，利用混合的方式提高开发效率，方便设计者们的使用。

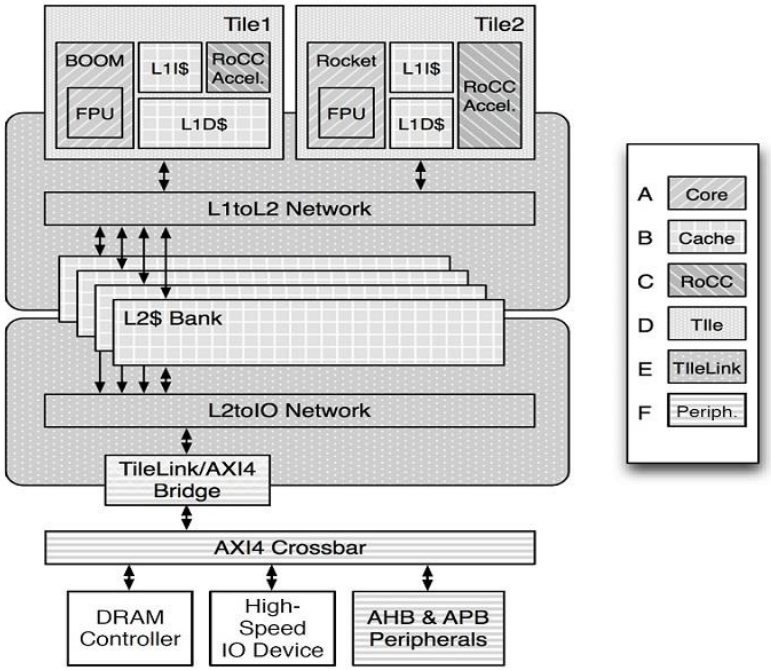


图 2-1 Rocket Chip 生成器的构成[14]

如图 2-1 所示，Rocket Chip 生成器主要由以下生成器和接口构成[14]：

A) Core 生成器：包括标量处理器内核 Rocket 的生成器和乱序执行的超标量处理器内核 BOOM 的生成器，这两种生成器均含有一个可选的 FPU、可配置的功能单元流水线和可定制的分支预测器；

B) Cache 生成器：包括一组缓存和 TLB（Translation Lookaside Buffer，页表缓存）生成器，其大小、连接和替换方法都是可以配置的；

C) RoCC 生成器：Rocket 协处理器接口（Rocket Custom Coprocessor，RoCC）

是一种协处理器模板，其参数可以基于特定应用修改；

D) Tile 生成器：缓存一致性的 tile 生成器模板，其中可以配置处理器内核、加速器和缓存的数目和类型；

E) TileLink 生成器：包括缓存一致性端口模块和缓存控制器的生成器，可配置的选项包括 tile 的数量、一致性的策略、共享的备份存储器和底层物理网络的实现；

F) 外设：兼容 AMBA 总线和很多转换器和控制器的生成器，包括 Z-scale 处理器。

标量处理器内核 Rocket 是基于 RV32G 和 RV64G 指令集架构的五级顺序执行标量处理器内核生成器，包含一个支持分页虚拟内存的 MMU、一个非阻塞的数据缓存和一个支持分支预测的前端。分支预测是可配置的，由 BTB（Branch Target Buffer，分支目标缓冲区）、BHT（Branch History Table，分支历史表）和 RAS（Return Address Stack，返回地址堆栈）实现。Rocket 利用了同样是由加州大学伯克利分校团队提出的基于 Chisel 的浮点单元来实现浮点数。Rocket 支持 RISC-V 的机器级、管理级和用户级的特权架构扩展，可以修改的参数包括一些 ISA 扩展、浮点运算流水线的数量以及缓存和 TLB 的大小。Rocket 也可以作为一个处理器组件库来使用，一些为 Rocket 开发的模块可以为其他设计所用，包括功能单元、缓存、TLB、页表遍历和特权架构实现。

Rocket Chip 是基于 RISC-V ISA 开发的，而 RISC-V 本身也是由加州大学伯克利分校开发的指令集架构，在 RISC-V 基金会的管理之下，是一种开源的扩展性极强的拥有广阔前景的指令集架构。RISC-V 指令集架构可以广泛使用于各种处理器内核，可以应用于高性能的乱序执行的设计，也可以应用于小型的嵌入式的处理器。Rocket Chip 使用 RISC-V 作为其指令集，可以避免出现受商业许可证限制的情况。

RISC-V 本身是一种灵活的模块化的架构，它最基本的特征是所有 RISC-V 处理器核心都必须使用的整数指令子集，可由字母 I 表示，同时也为其他的可选的扩展指令子集留有足够的操作码空间，可选的指令子集中已因其规范性而得到标准化的模块有：乘除法指令，可由字母 M 表示；原子（Atomics）指令，可由字母 A 表示；单精度浮点指令，可由字母 F 表示；双精度浮点指令，可由字母 D 表示；这些比较具有代表性的模块可以组合起来，也就是 IMAFD，组合作为一种通用的标量指令集，可由字母 G 表示；RISC-V 的地址编码模式可以是 32 位、64 位或者 128 位，还有一种压缩的扩展指令子集，可由字母 C 表示，使用的是 16 位的指令编码模式，来减小编码的长度[10]。

RISC-V32I 整数指令子集的指令主要可分为以下几类[10,33]：

1) 立即数赋值指令：它包括两条指令，分别是 LUI 和 AUIPC，两者的指令编码格式都是 U 类，立即数可直接作为指令的操作数输入。LUI 用于构建 32 位常数，将立即数放到目的寄存器 rd 的高 20 位，将 rd 的低 12 位填 0。AUIPC 用于构建 32 位偏移量，将 20 位立即数与 12 位的 0 组合起来，再将其加到 pc 上，得到的地址放入 rd 中。

2) 控制转移指令：可分为两种类型，无条件跳转和条件跳转，两者都可以被用于控制程序中的跳转，分支延迟槽（Branch Delay Slot）在体系结构中不可见。无条件跳转指令的寻址方式全部都是采用 pc 相对寻址，这有助于执行与位置无关的指令。所有条件跳转指令使用 SB 类指令格式，12 位二进制立即数对 2 字节倍数的带符号偏移量进行了编码，并且被加到当前 pc 上以产生目标地址。

3) 访存指令：它包含三种比特宽度的加载（Load）和存储（Store）指令，分别是：机器字(w)、半字(H)和字节(B)，只有 Load 和 Store 指令能够对存储器进行访问。Load 和 Store 的指令编码格式分别为 I 类和 S 类。Load 指令对存储器进行访问，把存储器中的数据传输到寄存器 rd 中。Store 指令对存储器进行写入操作，把寄存器 rs 中的值传输到存储器中。

4) 移位操作指令：移位操作包括算术右移（SRA）、逻辑右移（SRL）和逻辑左移（SLL），被移位的操作数存储在寄存器 rs1 中，移位次数存放在寄存器 rs2 的低 5 位中。

5) 逻辑操作指令：包含 AND、OR 和 XOR 指令，将两操作数按位进行 AND、OR 和 XOR 操作，并且把结果写入至目的寄存器 rd 中。

6) 运算指令：ADD 和 SUB 分别执行加法和减法，SLT 执行有符号数比较，SLTU 执行无符号数的比较。

7) 系统调用指令：系统指令用于访问那些可能需要特权访问的系统功能，以 I 类指令格式编码。系统指令可以分为两类：一类是原子性读-修改-写控制和状态寄存器（CSR）的指令，另一类是其他特权指令。

RISC-V32M 乘除法指令子集主要包括乘法、除法和取余指令[10,33]：

1) 乘法指令：MUL 指令执行一个 XLEN 位数乘 XLEN 位数的乘法，并将结果的低 XLEN 位写入 rd 中。MULH 执行有符号数乘有符号数的乘法，MULHU 执行无符号数乘无符号数的乘法，MULHSU 执行有符号数乘无符号数的乘法，都是将运算结果的高 XLEN 位返回。

2) 除法指令：DIV 执行有符号的 XLEN 位整数除以 XLEN 位整数的除法运算操作，DIVU 执行无符号的 XLEN 位整数除以 XLEN 位整数的除法运算操作。

3) 取余指令：REM 执行有符号的 XLEN 位整数除以 XLEN 位整数的取余操作，REMU 指令执行无符号的 XLEN 位整数除以 XLEN 位整数的取余操作。

Rocket 标量内核还支持一些可选的 A、F、B 指令集扩展。

指令流水线由所有细化的操作阶段串联而成，通常将流水线中细化的操作阶段称为段[34]。Rocket 是一个拥有 5 级流水线的标量单发射顺序处理器内核，指令执行过程可划分为指令读取、指令解码、执行、访问内存和写回等不同阶段[33]：

1) 指令读取（Instruction Fetch）：从存储器中读取指令的这一过程叫做指令读取。Rocket 处理器内核的指令读取阶段主要由 ibuf 部件来实现，Scala 源码具体的实现在 ibuf.scala 中。

2) 指令解码（Instruction Decode）：翻译从存储器中取出的指令的这一过程称为指令解码。经指令解码阶段后可得到指令所需要的寄存器索引以及操作数，能够使用寄存器索引从通用寄存器堆（Register File，Regfile）中将所需操作数读出。指令解码阶段主要对文件 ibuf.scala 中提供的指令进行解码，即可解析出控制线的信息和一些其他的信息，存放到 id_ctrl（控制寄存器）中。Decode 的解码逻辑在 decode.scala 中，映射关系在 idecode.scala 中。解码阶段还能够读取寄存器中的值，供执行阶段使用，但这个值并不一定正确，需要经过旁路等逻辑才是正确的值。

3) 指令执行（Instruction Execute）：指令经解码阶段之后，已得知所需要进行的计算类型，而且也已经从通用寄存器堆中读取出了要完成指令所需的操作数，随后便开始进入指令执行阶段。指令的执行阶段是真正对指令进行运算操作的过程，不同的指令会有不同的需求，最终决定需要使用的部件类型。在执行阶段中，最常见的是作为实施具体运算操作的硬件功能单元——算术逻辑运算部件（Arithmetic Logical Unit，ALU），该部件的源码在 alu.scala 中。访存指令会在执行阶段准备好 Cache 的 request 值。

4) 访问内存（MEM）：访存阶段是指对存储器进行访问从而将数据读出，或者写回的过程。访存部件是 dmem（连接一级数据缓存），Scala 源码包含在 DCache.scala 文件中。访存阶段还会计算分支指令和跳转指令的目的地址。

5) 写回（Write-Back）：把指令执行完的运算结果写入寄存器堆的这一过程称为写回阶段。若指令为普通的运算指令，那么计算结果的值源于执行阶段的计算结果；若指令为存储器读指令，那么计算结果源于访存阶段对存储器进行访问从而读出的数据。在 Rocket 处理器内核中，写回阶段执行包含加法器、乘法器、访存和 ROCC 等需要写回的操作，每个周期只能写回一个寄存器，存在优先级问题。ROCC 的指令发送阶段也在写回阶段，因为写回阶段才能确定好发送给 ROCC 的寄存器值。

如今，RISC-V 已经拥有庞大的软件生态，包括：GCC 编译器和 LLVM 编译器，它们所支持的 binutils 工具集和 glibc 库，Linux 操作系统和 FreeBSD 操作系统的接口，通过 Linux 基金会旗下的 Yocto 组织所提供的 poky 系统开发的一系列软件，以及 QEMU 模拟器和 Spike 模拟器[11,33]。

Rocket Chip 本身是基于 Chisel 硬件描述语言开发的，Chisel 利用 Scala 嵌入式语言来描述硬件，可以直接对可综合的电路进行描述。相对于高级别的软件语言，Chisel 更接近于像 Verilog 这种传统的硬件描述语言，但 Chisel 使得 Scala 这种编程语言可以用来生成电路，对电路进行指向功能和对象的描述。Chisel 还拥有支持结构化数据的丰富的类型系统，可以推测线宽，拥有高层次的状态机描述，可以进行批量线操作。我们可以用 Chisel 生成可综合的 Verilog 代码，将其读入 FPGA 和 ASIC 设计工具。Chisel 还可以生成快速的周期精确的 C++模拟器，与 Verilog 仿真器功能一致，但速度明显更快，可以用来对 Rocket Chip 项目生成的 SoC 进行仿真。

Rocket Chip 生成器基于 Chisel 构建了一个 RISC-V 平台，Rocket Chip 生成器包含许多参数化的芯片构建库，可以生成自定义的 SoC。Rocket Chip 将连接不同的库的生成器的接口标准化，我们可以仅仅通过修改配置文件就将设计中的许多组件替换掉，不需要去对硬件的源代码作出修改。我们可以只测试某个生成器的输出，也可以对整个设计做测试，测试也同样是参数化的，保证测试的覆盖范围足够大。

2.2 Rocket Chip 项目的子模块

Rocket Chip 是一个用来生成可综合的 RTL 代码的 SoC 生成器，它是一个开源的项目，基于 Chisel 硬件描述语言，从复杂的生成器库里，将 core、cache 和 interconnect 等部件组合成为一个完整的 SoC，相当于是把所有的碎片设计粘合在一起形成一个完整的设计。Rocket Chip 提供了使用开源的 RISC-V 指令集架构的通用处理器内核，提供了一种顺序执行的处理器内核 Rocket 的生成器，和一种乱序执行的处理器内核 BOOM 的生成器[24]。

Rocket Chip 生成器是 RISC-V 托管在 Github 上面的官方项目。Rocket Chip 代码仓库中包含许多由 Git 子模块指向的子仓库，其中包含我们生成和测试 SoC 所用的工具。Rocket Chip 代码仓库中还包含生成 RTL 所需的代码，Rocket Chip 会调用 Chisel 编译器来产生 RTL，生成一个完整的 SoC。

Git 子模块指的是将其他的 Git 仓库作为本仓库的一个子目录，子模块会随其他 Git 仓库的更新而更新，本仓库无需去维护这些子模块。Rocket Chip 代码仓库

所追踪的 Git 子模块包括：

1) chisel3：Rocket Chip 利用 Chisel 来生成 RTL 代码，Chisel 现已进展到 Chisel3 版本，详见文献[35]；

2) firrtl：即 Flexible Internal Representation for RTL，从 Chisel 到 RTL 的中间语言表示形式，Chisel3 编译器先生成 firrtl 表示，再进一步生成 Verilog 代码、C 代码等最终形式，详见文献[36]；

3) hardfloat：hardfloat 代码仓库包含了用 Chisel 实现的浮点单元，支持一个可选且符合 IEEE 754-2008 标准的 floating point 实例，它可以执行单精度和双精度浮点运算，用于乘加混合运算、整数与浮点数之间的转换以及不同精度的浮点数之间的转换，详见文献[37]；

4) rocket-tools：支持 Rocket Chip 的一些 RISC-V 工具，包括：ISA 模拟器，Spike；ISA 测试集，riscv-tests；模拟器所支持的所有 RISC-V 指令操作码，riscv-opcodes；RISC-V 代理内核和引导加载程序，riscv-pk；详见文献[38]；

5) torture：Rocket Chip 利用 torture 测试产生器来生成和执行有约束但随机的命令流，以用于设计的核心和非核心部分的压力测试，详见文献[39]。

Rocket Chip 的代码本身是分解在很多 Scala 包（Scala Package）中的，都在 src/main/scala 目录下，有些是用于生成器配置的 Scala 程序，有些则包含 Chisel 编写的 RTL 生成器。Scala package 中主要包括：

1) amba：Rocket Chip 利用该 RTL 包使用 diplomacy 生成 AMBA 总线，包括 AXI4、AHB-lite 和 APB；

2) config：Rocket Chip 利用这个程序包来提供 Scala 接口，通过动态变化的参数库来配置生成器；

3) coreplex：Rocket Chip 利用这个 RTL 包将其他包里的各种组件组合起来生成一个完整的 coreplex，包括 Rocket 内核块、系统总线网络、缓存一致性端口、调试设备、中断处理程序、外设、时钟 crosser 以及从 TileLink 到外部总线如 AXI 或者 AHB 的转换器；

4) devices：Rocket Chip 利用该 RTL 包实现外设，包括调试模块和各种 TL slave；

5) diplomacy：生成参数化的总线所需的参数协商机制，Rocket Chip 中的 TileLink 利用 Diplomacy 来确定不同级别的总线间的互连；

6) groundtest：用于 Rocket Chip 硬件测试的 RTL 包，生成随机的内存访问流，对非核心的内存层次进行压力测试；

7) jtag：Rocket Chip 利用这个 RTL 包实现 JTAG 接口；

8) regmapper：Rocket Chip 利用该程序包来生成具有标准接口的从设备，以访问其内存映射寄存器；

9) rocket：该 RTL 包生成了顺序执行的流水线的处理器内核 Rocket，以及 L1 指令和数据缓存，用来例化内存中的内核并将其连接到外部；

10) tile：这个 RTL 包中的组件可以与内核组合构成 tile，比如 FPU 和加速器；

11) tilelink：该 RTL 包利用 Diplomacy 实现 TileLink 总线协议，包含很多适配器和协议转换器；

12) unittest：这个程序包包含有生成各个模块的可综合的硬件测试器的框架；

13) util：Rocket Chip 利用该程序包提供其他包里共用的 Scala 和 Chisel 构造。

当设计者试图将 RTL 移植到不同的工艺节点上时，在不同的性能、功耗和面积的约束条件下，SoC 生成器有助于快速调整设计，很容易改变设计上的缓存大小和流水线的级数，甚至是应对不同应用的处理器和架构。

2.3 RISC-V 交叉编译工具链的搭建及测试

我们一般使用高级语言，如 C 语言，在计算机上编写代码，要使我们编写的代码能被计算机所识别，一般需要经历四个环节：预处理、编译、汇编和链接。我们编写的 C 程序经过预处理过程，头文件中的内容会被添加到预处理后的文件中，宏定义会替换掉程序中的相应内容，根据标识符，条件编译中不必要的指令会被删去，程序中的注释的内容也会被删去。编译过程则是对预处理后的文件做语法的检查和分析，再将其转换成汇编代码。相对于高级语言，汇编语言是更底层的面向硬件的语言，利用助记符对硬件进行操作，但对计算机来说，它其实只能识别和执行机器指令，也就是数字“1”和“0”组成的二进制代码，所以我们还需要通过汇编过程将汇编代码转换成机器码。链接过程将程序中调用的各个函数和汇编后的文件链接到一起，生成一个可执行文件，去在计算机上执行。这个过程中的每个环节都有对应的工具软件，也就是所谓的工具链。将 C 程序转换为可执行文件的这个过程，称为本地编译。

而如果我们在 x86 计算机上编写的 C 程序，最终要在其他架构的平台上运行，比如 RISC-V 平台，两种架构的指令集并不相同，不可兼容，我们就需要工具在将 C 程序转换为可执行文件的过程中，针对 RISC-V 指令集架构生成相应的二进制代码，这就是所谓的交叉编译[40]。

本次设计需要将 x86 平台上编写的 C 程序转换为可以在 RISC-V 平台上运行的可执行文件，需要在 Linux 环境中安装 RISC-V 交叉编译工具链。RISC-V 官方就提供了 RISC-V 交叉编译所需要的一系列工具，包括比较通用的 ELF/Newlib 工

具链和相对更加复杂的 Linux-ELF/glibc 工具链。

Newlib 是一个旨在嵌入式系统上使用的 C 函数库，是由很多免费的库组成的集合[41]。Newlib 对用户提供且只提供源代码，已推出了 3.0.0 版本，其源码目前由 Red Hat 公司在维护。

glibc 是 GNU 在 1988 年推出的一个 C 函数库，现在常见的 GNU 系统、GNU/Linux 系统以及其他基于 Linux 内核的操作系统基本都使用了 glibc，目前已推出 2.31 版本[42]。

考虑到 glibc 过于庞大，包含了许多我们不需要的函数，在本次设计中，我们选择使用基于 Newlib 的工具链。安装好交叉编译工具链后，我们还需要在 Linux 系统的 PATH 环境变量中添加交叉编译工具链所在的路径，至此，我们本次所使用的 RISC-V 交叉编译工具链基本搭建完成。

接下来，我们可以通过 Spike 模拟器、交叉编译工具和 riscv-pk，借由一个简单的 C 程序，来测试我们所搭建的 RISC-V 交叉编译工具链是否在正常工作。Spike 是一个 RISC-V ISA 模拟器，实现了若干个 RISC-V 线程的功能模型，支持 RISC-V 的多种指令集模块。riscv-pk（RISC-V Proxy Kernel）是一个支持静态链接的 RISC-V ELF 二进制文件的小型的执行程序的系统环境，在有限的 I/O 功能下实现了有限的 RISC-V 内核，使我们可以在代理内核中运行 RISC-V 程序。

我们编写了一段简单的累加求和代码如下，保存为 test.c：

```
#include <stdio.h>
int main() {
    int i, sum;
    for (i = 1; i <= 100; i++) {sum += i;}
    printf("1 + 2 + 3 + … + 100 = %d\n", sum);
    return 0;
}
```

利用 RISC-V 交叉编译工具 riscv64-unknown-elf-gcc 编译该程序，生成对应的可执行文件 test，再利用 Spike 执行该文件，输出结果如图 2-2 所示，显示了正确的计算结果，所搭建的 RISC-V 交叉编译工具链可以正常工作。

```
[riscv@localhost ~/test]$spike pk test
1 + 2 + 3 + … + 100 = 5050
```

图 2-2 交叉编译工具链测试

至此，RISC-V 交叉编译工具链搭建顺利完成。

2.4 Chisel 概述

正如前文所述，Rocket Chip 是基于 Chisel 语言编写的一种硬件生成器，那么 Chisel 具体是什么，又是如何编写出生成器的呢？接下来将对 Chisel 进行一番系统的介绍。

Chisel 是一种硬件描述语言，如同它的全称，Chisel 是加州大学伯克利分校 RISC-V 团队基于编程语言 Scala 开发的，在 Scala 中增加了硬件构造原语，使设计者们可以利用编程语言去编写复杂的参数化硬件生成器，从而生成可综合的 Verilog 代码[10]。生成器这种方式使得硬件设计可以在一定意义上如同软件开发一般，拥有一套可以复用的硬件模块库，比如 Chisel 标准库中的 FIFO 队列和仲裁器模块，这使得硬件语言拥有类似于编程语言的抽象层次，同时又具备低层次语言直接面向硬件的特征，可以大大提高设计者们的生产力。

要理解 Chisel 语言编写的项目，首先我们要研究 Scala 的使用，那么 Scala 又是什么呢？Scala 是一种高级编程语言，以简洁的方式将面向对象的编程（object-oriented programming，OOP）与函数式编程（functional programming，FP）结合，很适合用来做领域特定语言（Domain-Specific Language，DSL），可以被组合或者塑造成新的语言[43]。支持 JAVA 库，Scala 代码编译后得到.class 文件，可在 JVM（Java Virtual Machine）上运行。

以简单的 Scala 使用为例，将之前那段累加求和代码修改如下：

```
class test(max: Int) {
   var sum = 0
   var i = 1
   for (i <- 1 to max) {sum += i}
   println(s"sum is $sum")
}
```

以上代码中，我们定义了一个名为 test 的类（Class），Scala 中的类是一个定义了变量、常量、方法或函数的对特定对象的抽象。用 test 这个类创建实例时，需要带一个名为 max 的整数参数。test 这个类在大括号括起来的代码块中，定义了类的数据和操作。然后利用关键字 var 依次声明了变量 sum 和 i，并分别赋初值为 0 和 1。

接下来我们用一个 for 循环来完成累加操作，Scala 中的 for 语句与其他编程语言中的没有大的区别，我们在数字区间 1 至 100 内进行遍历，累加求和的值赋给变量 sum。最后利用 Scala 的 println 语句将变量 sum 的值打印至标准输出，Scala 的 println 语句其实就相当于打印时自动换行的 C 语言中的 print 语句。

定义好类后，我们用关键词 new 将 test 类实例化，这样就可以去访问类中的变量和操作。参数 max 的值为 100，如图 2-3 所示，可以看到 1 至 100 累加求和的结果显示在标准输出，为 5050。

```
scala> var x = new test(100)
sum is 5050
x: test = test@1e8ab90f
```

图 2-3 Scala 类的实例化

我们也可以通过函数（Function）来实现以上的功能，比如：

```
def accu(max: Int) : Int = {
  var sum:Int = 0
  for (i <- 1 to max) {sum += i}
  return sum
}
```

若我们将以上函数定义在类中，那么它会被称为方法（Method），大部分时候我们并不会去区分这两者。我们利用关键字 def 定义了一个函数 accu，该函数需要一个整数参数 max，并会最终返回一个整数的值。函数中，我们首先声明了一个整数变量 sum，并赋初值为 0，然后利用 for 循环从 1 至 100 进行累加求和，最后返回变量 sum 的值。Scala 中认为一个代码块的最后一行是其返回值，我们的函数 accu 的返回值是 sum，当然，我们在代码块中也可以不使用返回值。

定义好函数后，我们可以直接调用它，给定参数值 max 为 100，如图 2-4 所示，可以看到返回整数值 5050。

```
scala> accu(100)
res18: Int = 5050
```

图 2-4 Scala 函数的定义和调用

了解了 Scala 最基本的语法之后，理解 Chisel 会变得容易一些。那么 Chisel 又是如何利用 Scala 来生成硬件的呢？

首先，我们需要了解，Chisel 生成硬件电路需要 firrtl 这个中间表示，firrtl 相当于是一个硬件编译器。Chisel 利用 Scala 编写了一系列对输入电路进行转换、验证、优化和输出的代码，存放在 firrtl 的 Git 库中，这些代码链接到一起并写出最终的电路后就构成了 firrtl 编译器，将 Chisel 生成的电路最终转换为数字电路。firrtl 支持设计者去自定义电路的转换。

Chisel 作为一种硬件描述语言，它最强大的地方在于可以编写生成器

（Generator），并在设计中反复的使用它。如下所示，以 Rocket Chip 项目中的一个分频器模块为例[15]：

```
class Pow2ClockDivider(pow2: Int) extends Module {
  val io = IO(new Bundle {
    val clock_out = Output(Clock())
  })
  if (pow2 == 0) {
    io.clock_out := clock
  } else {
    val dividers = Seq.fill(pow2) { Module(new ClockDivider2) }
    dividers.init.zip(dividers.tail).map { case (last, next) =>
      next.io.clk_in := last.io.clk_out
    }
    dividers.head.io.clk_in := clock
    io.clock_out := dividers.last.io.clk_out
  }
}
```

这段代码定义了一个名为 Pow2ClockDivider 模块（Module），这其实与 Verilog 语言中的 Module 有些相似。Scala 使用关键字 extends 来对其它类进行继承，Module 类可以称为父类，Pow2ClockDivider 类可以称为子类，子类可以继承父类的全部成员变量和方法，并可以进行扩展。Module 类是 Chisel 自带的，用户所定义的全部硬件模块都要先去继承 Module 类。该硬件模块需要一个整数参数 pow2，表示要对时钟做 2^{pow2} 次方分频。模块中，首先要声明端口，这需要实例化一个在其他文件中定义的 Bundle 类，声明 clock_out 为时钟输出端口，所有的这些端口声明赋给常量 io。然后在生成器内部描述所有的电路，对电路节点进行连接。先利用 if 语句，判断是否要进行分频，若分频参数 pow2 为 0，对时钟做 2^0 分频，也就是不进行分频，直接将时钟与输出相连。否则，首先例化 pow2 个二分频器，赋给集合 dividers，Chisel 常常利用 Scala 的集合 Seq 来构造和传递参数，然后利用 Scala Map（映射）对集合中的每个分频器调用代码块 next.io.clk_in := last.io.clk_out 进行两两首尾相连。最后将时钟与集合 dividers 中的第一个分频器的输入端相连，将集合 dividers 中的最后一个分频器的输出端与 Pow2ClockDivider 模块的输出端相连。

这段代码其实并不仅仅描述了一个模块，它描述了所有的 2^{pow2} 时钟分频器模块，只要改变参数 pow2，就可以实现不同的分频器，所有我们可以将

Pow2ClockDivider 称为生成器。而 Pow2ClockDivider 其实是个 Scala 类，我们可以对它进行实例化来调用相应参数的分频器，这就是 Chisel 所说的参数化。参数化的设计可以通过参数传递方便地调用各个模块，提高硬件开发效率和准确性。

Rocket Chip 项目就可以借由参数的传递去自定义我们想要的处理器配置并生成相应的 RTL。比如，在 rocket-chip/src/main/scala/system/Configs.scala 文件中，定义了多组配置，基于 Configs.scala 中的配置，来生成 ExampleRocketSystem.scala 中所定义的 SoC。由于 Rocket Chip 将处理器的每一种功能都进行了参数化，我们可以在 Configs.scala 文件中定义新的 config 类，参考已有的 config 类，修改其中的参数，构成我们需要的功能，再将我们需要的功能组合起来，形成我们想要的配置，比如，Configs.scala 文件中提供的基本配置 BaseConfig 中包含一个 16KB 的 2 路组相连 L1 指令缓存，而我们想要得到一个 32KB 的 4 路组相连 L1 指令缓存，我们就需要将 L1ICacheWays 中的 nWays 参数改为 4。我们也可以直接就已有的配置更改其功能的组合，比如，Configs.scala 文件中提供的可选配置 DefaultSmallConfig 通过 extends 关键字继承了基本配置 BaseConfig 的所有参数，只是增加了一个 WithNSmallCores 的调用。

2.5 本章小结

本章主要研究了 Rocket Chip 项目的架构。首先，我们简要介绍了 Rocket Chip 项目，对其有了基本的了解。然后，我们研究了 Rocket Chip 项目的子模块，了解了其架构。接下来，为了能够在 RISC-V 平台上运行 C 程序，我们搭建了 RISC-V 交叉编译工具链，并借由一个简单的 C 程序对其进行了测试。最后，为了更好的理解 Rocket Chip 项目，我们对 Chisel 硬件描述语言的使用进行了简单的介绍。

第三章 基于 RISC-V 的 SoC 的前后端实现

3.1 基于 Rocket Chip 的 SoC 的前端设计研究

基于 Rocket Chip 项目原有的配置，对其进行扩展，生成一个 SoC。定义了一个新的类 System，继承原来的 RocketSubsystem 类，并对其进行扩展，增加 MaskROM、Debug、UART、SPI、GPIO、PWM、I2C 等外设，配置好各个外设的参数。创建一个模块 Platform，将其作为 SoC 的顶层，例化之前定义的 System，并在设计的输入输出中增加外设 JTAG、GPIO、SPI 的引脚，增加 JTAG 调试接口的复位引脚，声明为 Bool 型输入端口，增加一个用于调试的复位信号端口，声明为 Bool 型输出。在 Platform 模块的代码块中建立 GPIO 管脚复用，将 GPIO 复用成 UART、PWM、SPI 和 I2C。由此生成的 SoC 基本架构如图 3-1 所示。

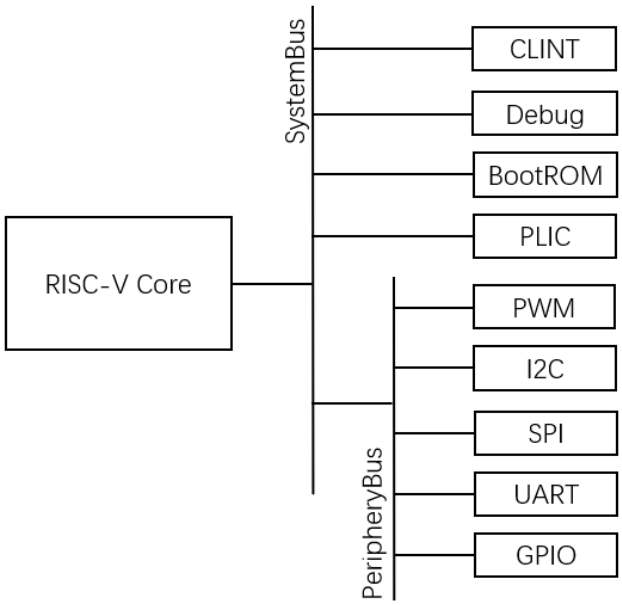


图 3-1 基于 Rocket Chip 的 SoC 基本架构示意图

项目调用的 MaskROM 模块挂接在 SystemBus 上，位于内存映射地址 0x10000-0x11FFF 上，其中包含一个 2K*32 的 BootRom 模块。Bootrom 的行为级模型由项目提供的 rocket-chip/scripts/vlsi_rom_gen 脚本生成。考虑到要将项目下载到开发板上，在 bootrom/xip 目录下有一个汇编文件 xip.S，其中写有一个直接跳转至地址 0x20400000 的程序，也就是直接跳转至 Arty A7 开发板的 SPI Flash 的地址 0x00400000。先利用编译工具 riscv64-unknown-elf-gcc 将 xip.S 转换为可执行文件 xip.elf，再利用复制工具 riscv64-unknown-elf-objcopy 将 xip.elf 转换为二进制文件

xip.bin，将它转换为 xip.hex，将 xip.hex 文件写入 BootRom 作为引导加载程序。复位向量 global_reset_vector 指向 MaskROM 的地址，使 SoC 系统复位后从 BootROM 启动。

项目调用 Debug 调试模块的方式是在顶层配置中增加一个 JTAG DTM（Debug Transport Module），并引出 JTAG 接口连到系统的输入端，也可以直接在系统顶层利用参数 DebugModuleParams 声明 DMI（Debug Module Interface，调试模块接口）的调用，使系统可以由外界调试信号通过 JTAG 接口进行测试和调试。Debug 调试模块挂接在 SystemBus 上，外界信号通过 JTAG 接口输入，在 DTM 中转换成同步信号，再通过调试模块与处理器核心交互，得到调试结果并输出。Jtag_TCK 与系统主时钟是异步时钟，这一点在编写设计约束时需要注意。

项目调用了 2 个 UART 模块挂接在 PeripheryBus 上，作为与外界通信的一种方式。UART 模块通过信号接收接口 UARTRx 接收外界信号 rxd，接收到的数据流转换为并行数据存入 rxfifo，再由处理器核心处理。要发送的数据则先写入 txfifo，再由信号发送接口 UARTTx 转换成串行数据流 txd 进行输出。包含一个波特率产生器模块 div。项目所调用的 UART 外设支持 8 个数据位、无奇偶校验位、1 个起始位、1 或 2 个停止位格式的数据收发，不支持硬件流控制，不支持其他调制解调器控制信号，不支持同步串行数据传输，这一点在利用串口工具与其通信时需要注意。

项目调用 GPIO 模块作为默认的普通的输入输出引脚，但部分 GPIO 引脚也可以作为其他外设的输入输出复用。GPIO 可以通过软件来操作其中的寄存器，也可以直接从硬件底层操作寄存器。模块提供了通过寄存器从硬件底层驱动 GPIO 引脚的 IOF 功能，利用*GPIOPort 转换器将 UART、SPI、PWM、I2C 的输出转换为 IOF，引脚复用时可直接由硬件驱动。GPIO 引脚不受硬件控制时则仍由软件控制。

此外，项目调用了 2 个 SPI 模块用来与外界进行同步通信，还调用了一个 QSPI 专门与片外的 SPIFlash 通信。由时钟产生器 sckdiv 产生串行时钟 SPI_sck。传输开始时数据被写入 txfifo 中，传输结束后从设备的回应被写入 rxfifo 中。项目调用了 I2C 模块，通过时钟线 scl 与数据线 sda 来进行数据传输。项目还调用 PWM 模块用来输出各种类型的波形，以及生成内部计时器中断。

sbt 是用 Scala 编写的可用于编译 Scala 或 Java 项目的一个工程建立工具，项目利用 sbt 工具编译项目的 Scala 代码，生成.fir 文件，通过 firrtl 工具将.fir 转换为 Verilog 代码。

在项目主目录下启动 sbt，以 firrtl 目录作为 sbt 的基础目录，使它得以自行去寻找项目源码目录以及 firrtl/src/main/scala 目录中的所有 scala 源文件和

firrtl/src/main/resources 中的文件，还需要在 firrtl/utils/bin 目录下建立一个用于缓存数据的 jar 包，在项目目录和 chisel3 目录下分别建立 lib 目录并将刚建立的 jar 包拷贝到刚刚建立的两个 lib 目录下作为依赖，便于 sbt 寻找。然后使用 sbt 的 run 命令去执行之前在配置文件中定义的类中的方法，也就是按照我们之前的配置来指定生成器生成相应的 firrtl，存入.fir 文件。利用 firrtl 提供的 Driver 方法启动编译器，利用 Java 将 firrtl 转换为 Verilog 代码，得到所构建的 SoC 的 RTL。

如图 3-2 所示，项目由 Chisel 源码可以生成 C++代码，使用 C++编译工具编译并运行它，会得到一个周期精确的软件模拟器，速度比 Synopsys 的 VCS 仿真工具快得多，而且不需要 license，生成的软件模拟器甚至可以查看波形。项目由 Chisel 源码可以生成为 FPGA 优化的 Verilog 代码，我们可以将 SoC 映射在 FPGA 上。项目还可以生成可用于 ASIC 实现的 Verilog 代码，基于代工厂所提供的标准 ASIC 库，配合 EDA 工具，从而得到 GDS 版图数据。

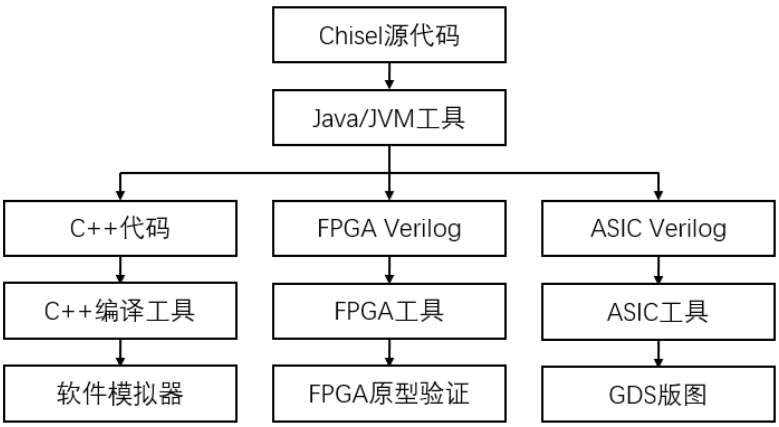


图 3-2 基于 Chisel 源码的项目前后端流程示意图

3.2 基于 Rocket Chip 的 SoC 的逻辑综合

逻辑综合是 ASIC 芯片设计过程中的一个重要步骤，我们在这一步将数字电路的 RTL 级（Register Transfer Level，寄存器传输级）描述转化为可用于自动布局布线的门级网表。逻辑综合主要可以分为三个过程，即：翻译，由逻辑综合工具将数字电路的 RTL 级描述转化为布尔函数表达式；优化，由逻辑综合工具根据我们提供的设计要求对电路的布尔函数表达式做一定的逻辑重组和优化；映射，由逻辑综合工具从目标工艺库中找到合适的逻辑单元来替换掉优化后的设计中所有的逻辑单元，得到我们所需的门级网表[44]。

3.2.1 设计约束

在本次设计中，我们利用 Design Compiler 工具，基于代工厂的 0.13μm 工艺库，对之前得到的基于 Rocket Chip 的 RISC-V SoC 的 RTL 代码进行逻辑综合。

首先，我们需要设置 Design Compiler 工具进行逻辑综合时所使用的库文件，主要包括：目标库（Target Library），也就是逻辑综合时所使用的特定工艺的标准单元的库文件，其中包含所有逻辑单元的功能描述、输入输出、管脚电容、相应的时序和功耗信息等；链接库（Link Library），也就是逻辑综合进行综合时所调用的标准单元的库文件和除了标准单元以外的其他元件的所有库文件，比如我们设计中调用的存储单元。我们使用 Memory Compiler 工具生成存储单元的时序库和物理库文件，去替换设计中的行为级模型。考虑到希望 Design Compiler 综合结果能与 IC Compiler 布局布线有更好的一致性，我们使用的是 Design Compiler 的 topographical 模式，此模式下会在综合时就对设计进行虚拟布线，这样估算出来的延迟会比使用线负载模型更接近真正布局布线时的情况，所以接下来我们还需要设置好所有单元的物理库文件，并提供寄生参数文件。

我们需要设置逻辑综合过程是在何种操作环境（Operating Condition）下进行。操作环境主要包括时序库和 RC Corner。本次设计所使用的工艺下一般取四种时序库，表示工艺、电压和温度不同时的器件延迟，如表 3-1 所示。逻辑综合过程一般只取 Slow 这个时序库，使所有单元都工作在最慢的条件下，看最终能否满足设计要求。

表 3-1 四种时序库

时序库 / PVT	Slow	Typical	Fast（0℃）	Fast（-40℃）
工艺（Process）	1	1	1	1
电压（Voltage）	1.08V	1.20V	1.32V	1.32V
温度（Temperature）	125℃	25℃	0℃	-40℃

本次设计所使用的代工厂的 0.13μm 工艺提供三种 RC Corner（cmin，typ，cmax）的寄生参数库文件，用来计算设计中的互连线延迟。类似的，逻辑综合过程一般只取 cmax 这种 RC Corner，考虑互连线延迟最大的情况，看最终能否满足设计要求。

至于一些其他的设计约束，我们一般将其写在 SDC（Synopsys Design Constraints）文件中，并在开始逻辑综合之前直接读入该文件。SDC 文件中一般包括以下几种设计约束：

1) 时序约束，主要是与时钟有关的约束，包括：时钟的定义，比如，在本次设计中，我们需要一个频率为 100MHz 的时钟，故需创建一个时钟 clock，并将其周期设置为 10ns，还有一个与其异步的 JTAG 时钟 jtck，同样对其进行创建，将其周期设置为 100ns，以及另一个异步时钟 qspi_sck，将其周期设置为 20ns，将异步时钟约束为不同的时钟组，使工具不对它们之间的时序路径进行检查；时钟的转换时间（Clock Transition），一般指时钟信号的电压从 10%的峰值上升至 90%的峰值所需的时间，或者电压从 90%的峰值下降至 10%的峰值所需的时间，这个时间越大，意味着设计的速度越慢，功耗越大，我们将其设置为 0.4ns，以限制时钟的转换时间的大小；时钟的不确定性（Clock Uncertainty），在真实的电路工作中，时钟信号会发生抖动（jitter），相应的时钟周期会产生变化，我们将其设置为 0.4ns 来模拟真实的时钟；

2) 设计规则约束，主要包括最大转换时间（Max Transition）、最大扇出（Max Fanout）和最大电容（Max Capacitance）约束等，工艺的时序库中对每个单元都有其默认的设计规则约束，但有些时候时序库的设计规则约束会显得过于乐观，我们通常可以人为地对整个设计施加额外的合理的设计规则约束，以满足本次设计的时序、功耗和面积要求，我们将整个设计的最大扇出设置为 64，最大转换时间设置为 0.5ns，最大电容按默认约束；

3) 环境约束，主要包括输入端口转换时间（Input Transition）、输出端口负载（Output Load）、输入延迟（Input Delay）和输出延迟（Output Delay）等，我们其实并不知道设计的外部会是怎样的环境，我们将输入端口转换时间设置为 0.6ns，输出端口电容负载设置为 3pF，输入延迟和输出延迟暂设置为 0，设置这些环境约束，可以使工具在之后的时序分析中，得以计算出每一条相关的时序路径的精确延时，以保证设计的时序收敛。

在逻辑综合阶段，我们还将时钟都设置为理想时钟网络，避免工具根据以上的设计规则约束对时钟网络进行优化，这是我们在后端物理设计的时钟树综合一步再单独去专门考虑的。此外，我们将最大面积约束为 0，希望工具使用最大的优化力度，优化过后的设计面积越小越好。

3.2.2 综合结果

综合工具进行逻辑综合过后，我们需要产生相应的报告并仔细检查其中是否存在一些问题。首先，我们使用 check_mv_design 命令产生相应的结果报告。该条命令其实主要应用于多电压设计中用来检查是否存在违反多电压规则的错误或警告。在本次设计中，我们主要使用该条命令来检查时序库与物理库之间是否存在

不匹配，这会导致物理设计中电源连接出现问题。根据 check_mv_design 检查结果报告，如图 3-3 所示，可以看出，并没有任何的错误和警告。

```
----------------------------------------------------------------------------
       Clock Gating Style Checks
----------------------------------------------------------------------------
No clock gating style defined yet.

----------------------------------------------------------------------------
                        Target Library Subset Checks
----------------------------------------------------------------------------
No Errors/Warnings Found.

----------------------------------------------------------------------------
                             Power Domain Checks
----------------------------------------------------------------------------

----------------------------------------------------------------------------
                         Cell Operating Condition Checks
----------------------------------------------------------------------------
No Errors/Warnings Found.

----------------------------------------------------------------------------
           Power Domain and Operating Condition Consistency Checks
----------------------------------------------------------------------------
No Errors/Warnings Found.
----------------------------------------------------------------------------
       Power/Ground Pin Connection Checks
----------------------------------------------------------------------------
No Errors/Warnings Found.

----------------------------------------------------------------------------
       Supply Operating Voltage Checks
----------------------------------------------------------------------------
No Errors/Warnings Found.
```

图 3-3 check_mv_design 检查结果报告

接下来便是我们在后端设计中最关注的时序收敛问题，根据图 3-4 所示的时序报告，我们可以看出，WNS（Worst Negative Slack）为 0，也就是说，在最差的一条时序路径上的建立时间也是收敛的，TNS（Total Negative Slack）为 0，时序违例的路径数为 0，以此判断，综合后的设计没有时序问题。至于保持时间的违例，逻辑综合阶段不予考虑。

```
----------------------------------------------------------------------

Design  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0

Design (Hold)  WNS: 0.21  TNS: 3.07  Number of Violating Paths: 43

----------------------------------------------------------------------
```

图 3-4 逻辑综合后的时序结果

我们还产生了面积报告，如图 3-5 所示，整个设计的器件总面积约为 2256577.783898μm^2。考虑到本次设计所使用的 0.13μm 的工艺下的 2 输入与非门

器件的面积约为 6.8 μm^2，本次设计等效门数约为 331850。

```
Area
-----------------------------------
Combinational Area:   320253.713959
Noncombinational Area:
                      994577.823845
Buf/Inv Area:          77601.897864
Total Buffer Area:          7224.13
Total Inverter Area:       17715.76
Macro/Black Box Area: 941746.246094
Net Area:                  0.000000
Net XLength        :     1593327.62
Net YLength        :     1582175.25
-----------------------------------
Cell Area:            2256577.783898
Design Area:          2256577.783898
Net Length         :      3175503.00
```

图 3-5 逻辑综合面积报告

此外，根据图 3-6 所示的功耗报告，可以看到，本次设计综合后的内部功耗约为 41.7144mW，开关功耗约为 0.4741mW，泄漏功耗约为 0.58904mW，总功耗约为 42.7775mW。

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             4.7334           0.2924        1.5365e+07            5.0411  (  11.78%)
memory            18.2966           0.1515        4.2210e+08           18.8703  (  44.11%)
black_box          0.0000           0.0000            0.0000            0.0000  (   0.00%)
clock_network  2.1937e-05        3.3846e-04          707.4980        3.6110e-04  (   0.00%)
register          18.6761        4.8911e-03        8.3926e+07           18.7649  (  43.87%)
sequential     2.3107e-04        4.5943e-07        8.8372e+06        9.0688e-03  (   0.02%)
combinational  7.9738e-03        2.4994e-02        5.8813e+07        9.1781e-02  (   0.21%)
--------------------------------------------------------------------------------------------------
Total             41.7144 mW         0.4741 mW     5.8904e+08 pW        42.7775 mW
```

图 3-6 逻辑综合后功耗报告

形式验证技术作为一种利用数学上穷举的方法将两个设计做等效性检查的方法，省去了大量传统验证技术开发测试向量的时间。现在可用于形式验证的工具主要有 Synopsys 公司旗下的 Formality 工具和 Cadence 公司旗下的 LEC 工具。在本次设计中，我们使用 Formality 工具，来对逻辑综合前的 RTL 设计与综合后产生的门级网表进行形式验证。如图 3-7 所示，验证成功，表明我们的设计在综合前后的逻辑功能并未发生改变。

```
********************************* Verification Results *********************************
Verification SUCCEEDED
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
   ATTENTION: RTL interpretation messages were produced during link
              of reference design.
              Verification results may disagree with a logic simulator.
--------------------------------------------------------------------------------
 Reference design: r:/WORK/ROCKET
 Implementation design: i:/WORK/ROCKET
 12518 Passing compare points
--------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop    BBNet      Cut     Port      DFF      LAT    TOTAL
--------------------------------------------------------------------------------------------
Passing (equivalent)         815       0       61        0       69    11571        1    12518
Failing (not equivalent)       0       0        0        0        0        0        0        0
********************************************************************************************
```

图 3-7 形式验证结果报告

3.3 基于 Rocket Chip 的 SoC 的后端物理设计

在本次设计中，我们主要利用 Synopsys 公司旗下的 IC Compiler 工具来完成芯片的后端物理设计部分。数字电路的后端物理设计流程基本如图 3-8 所示，主要包括数据准备、布图规划（Floorplan）、布局布线（Place&Route）、时钟树综合（Clock Tree Synthesis，CTS）、签核（Signoff）与物理验证，而这整个过程中，我们要始终注意设计的时序是否收敛。

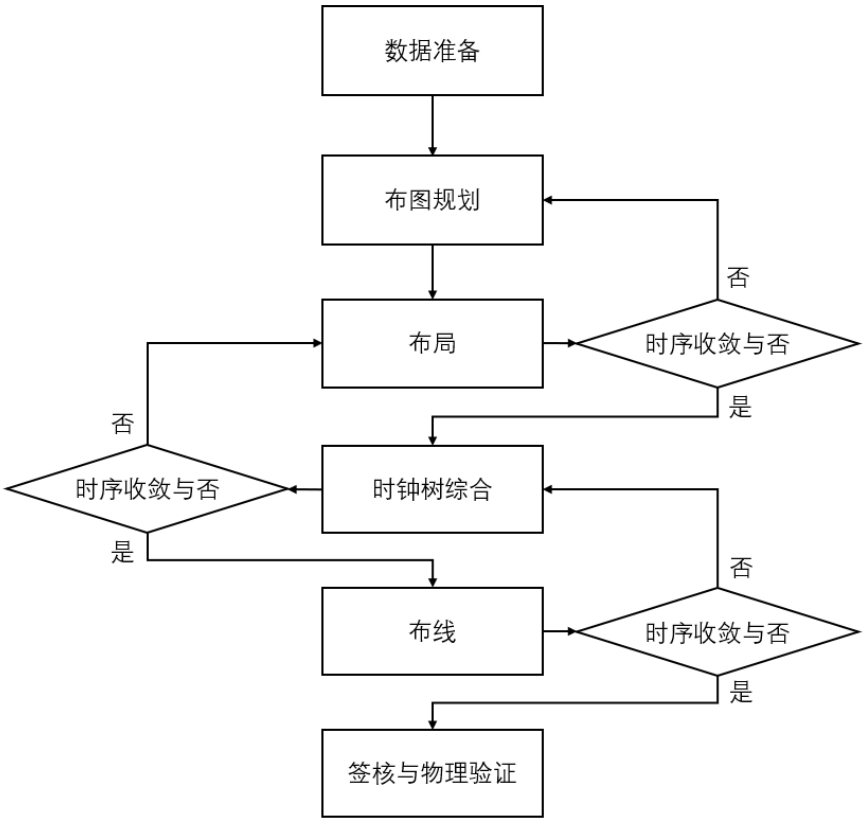


图 3-8 数字电路的后端物理设计流程

3.3.1 数据准备

首先，我们要准备好 IC Compiler 工具所需的文件，主要包括逻辑综合产生的门级网表文件、相应的 SDC 文件、本次设计所使用的 0.13μm 工艺下所有的时序库和物理库文件、工艺文件、寄生参数文件以及工具本身所需的一些环境配置。由于我们之前使用了 Design Compiler 的 topographical 模式进行逻辑综合，此时需要做的各项设置与逻辑综合前所作的设计约束比较相似，此处不予赘述。

3.3.2 布图规划

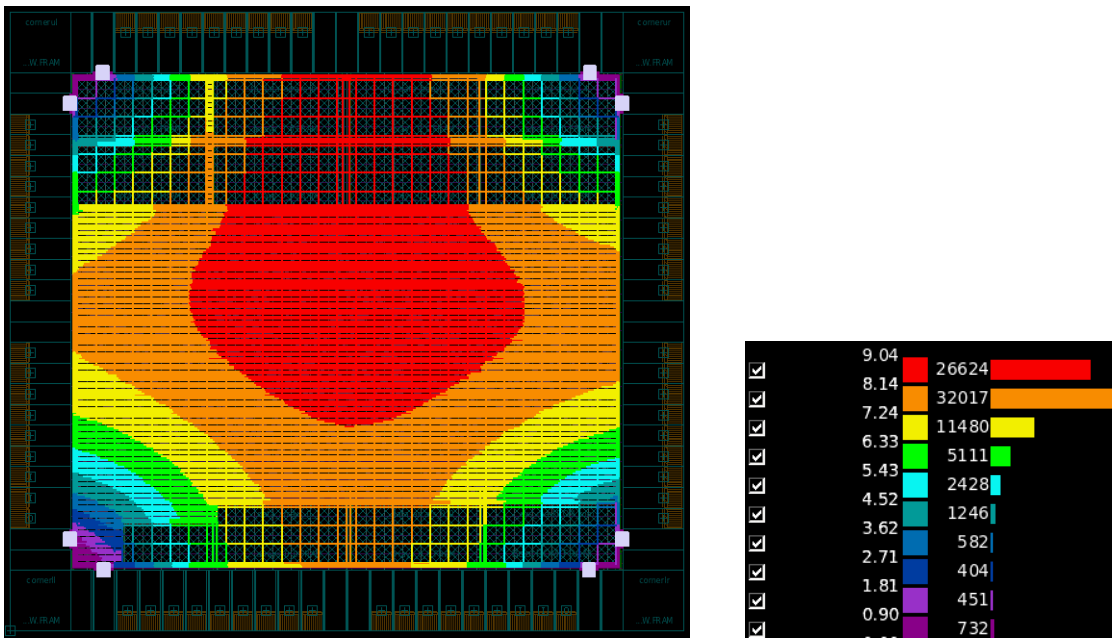
完成以上的数据准备，将其读入 IC Compiler 工具并检查无误后，我们就可以开始进行布图规划了。布图规划主要包括定义设计模块的大小形状、宏单元的位置摆放以及电源规划等，本次设计的模块规模不大，采用展平式设计，无需考虑内部子模块与端口以及内部子模块之间的层次关系。经过估算，规划了 1744μm*1567μm 的 core 区域，利用率约为 0.6。考虑到 ROM 和 SRAM 存储单元的 pin 均在纵向上，将它们摆放在 core 的上下两边附近，尽量不去影响之后内部标准单元的布局。

接下来需要进行电源规划，给我们的设计提供一个稳定的供电网络。本次设计使用工艺库中的电源 IO Pad PVDD1W 和 PVSS1W 为 core 内部供电，使用 PVDD2W 和 PVSS2W 为 IO 供电，将 8 组 PVDD1W 和 PVSS1W 以及 4 组 PVDD2W 和 PVSS2W 比较均匀地放置在 pad ring 的四条边上。考虑到布线资源使用率和电迁移的问题，电源布线通常会使用高层金属，本次设计采用的工艺有八层金属，故使用 METAL7 和 METAL8 来走电源条线，一般用中间金属层走电源环线，故使用 METAL3 和 METAL4 在供电 IO Pad 与 core 之间的区域走电源环线。考虑到 ROM 和 SRAM 存储单元内部已存在位于 METAL3 和 METAL4 的电源环线，不再额外为其增加电源环。然后我们需要将电源条线连接至电源环，并布 power rail，使所有单元的电源 pin 都连接到 followpin 上，整个电源网络都连接起来。接下来我们可以利用 verify_pg_nets 命令检查电源线的连接，如图 3-9 所示，电源线和地线均没有出现连接错误。

```
Checking [VSS]:
        There are no floating shapes
        All the pins are connected.
        No errors are found.
Checking [VDD]:
        There are no floating shapes
        All the pins are connected.
        No errors are found.
Checked 2 nets, 0 have Errors
```

图 3-9 检查电源线的连接

此时我们可以分析一下设计的 IR drop。在集成电路芯片上，不同器件的电压并不是一个理想的固定的值，在不同的芯片位置和不同的时间，电源网络上的电压会与供电 IO 给 core 提供的电压有所不同，而随折半导体制程越来越先进，器件的特征尺寸越来越小，器件所需的电压越来越小，而金属互连线的宽度越来越窄，金属互连线的电阻值随之增大，离供电单元越远的器件越容易产生电压降，速度随之变慢，器件的性能也变差了。利用 analyze_fp_rail 命令，以 PVDD1W 和 PVSS1W 为供电 IO Pad，让工具去估计每个器件实际的电压情况。分析结果如图 3-10(a)所示，虽然看起来大半部分飘红，但根据图 3-10(b)所示，最大的电压降为 9.04mV，远低于我们设置的供电电压 1.32V 的 10%，以此，我们认为本次设计的性能受 IR drop 影响不大，电源规划较为合理。

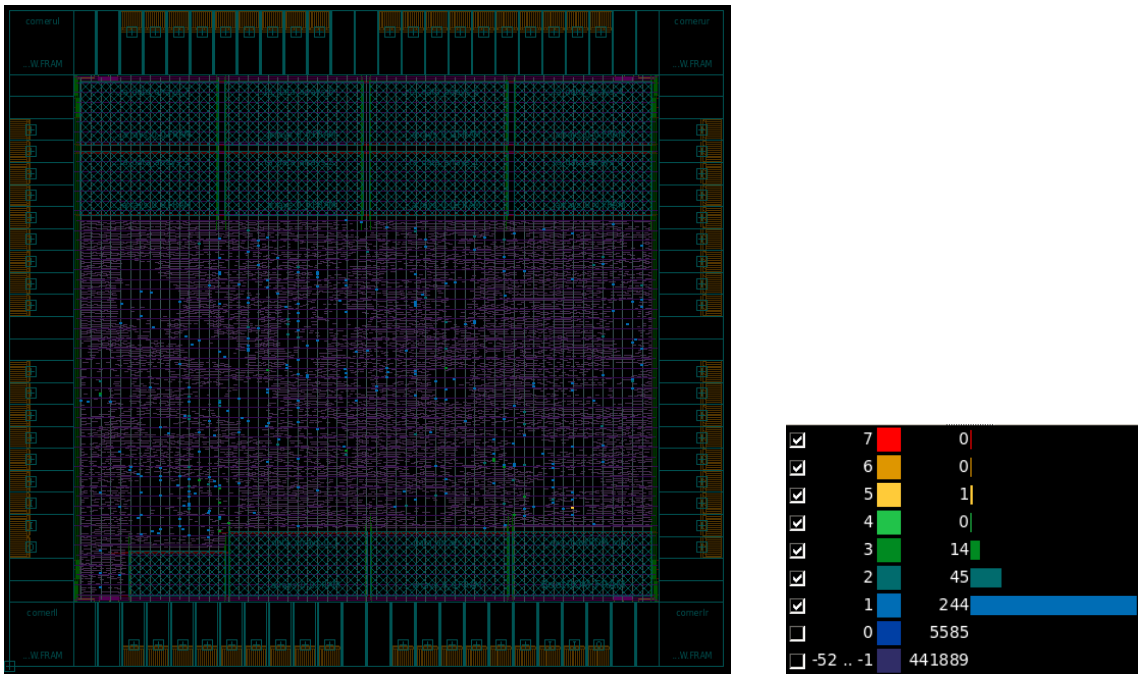


(a) (b)

图 3-10 IR drop 情况。(a)IR drop 分布；(b)不同区域 IR drop 的值，以及同等 IR drop 的区域的数量和颜色表示

完成电源网络的设计后，我们可以做一次 floorplan 阶段的简单布局，通过观察其拥塞（Congestion），大概评估一下本次布图规划的质量，看是否可以往之后的步骤进行。工具检查拥塞情况时，会先做一次全局布线，根据此次布线的结果，判断设计中的每一个 GRC（Global Routing Cell）的四条边上实际需要走的绕线与其分配到的绕线资源相差多少，从而判断是否溢出（Overflow），若溢出太多，可以认为本次布图规划并不尽如人意，在最终进行布线时可能会很难布通，此时就需要对布图规划进行改善。工具将设计划分为数十万个 GRC，然后据此计算 GRC

的每条边上实际会通过多少互连线，而每条边上占据的布线通道（Track）有多少，两值相减就得到 Overflow，溢出越多，Congestion Map 中 GRC 的那条边越标红，可以明显地看出拥塞集中在哪些地方。如图 3-11 所示，我们的设计中没有明显存在拥塞的地方。



(a) (b)

图 3-11 拥塞情况示意图。(a) 拥塞情况示意图；(b) 不同区域 overflow 的值，以及存在相同 overflow 值的区域的数量和颜色表示

查看文字报告如图 3-12 所示，我们的布图规划中没有 GRC 存在 overflow，不存在拥塞情况，可以继续进行布局。

```
Information: Reporting global route congestion data from Milkyway...

Both Dirs: Overflow = 0 Max = 0 (0 GRCs) GRCs = 0 (0.00%)
H routing: Overflow = 0 Max = 0 (0 GRCs) GRCs = 0  (0.00%)
V routing: Overflow = 0 Max = 0 (0 GRCs) GRCs = 0  (0.00%)
```

图 3-12 拥塞报告

本次布图规划的结果如图 3-13 所示，算上周围的 IO Pad，floorplan 的大小为 2173.86μm×1998.25μm。

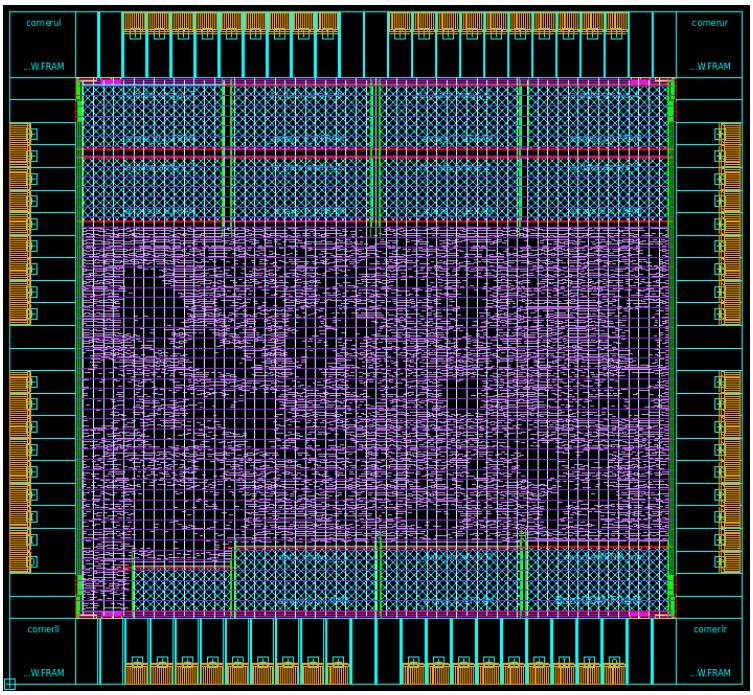


图 3-13 floorplan 结果

3.3.3 布局

实现本次设计的 floorplan 之后，我们需要做布局（Placement），也就是将所有的标准单元都适当地摆放在 core 区域中的行通道（Row）上。工具在进行布局时，首先会在不考虑标准单元会否重叠以及准确地摆在 row 上的情况下，快速地完成一个粗略的布局，接下来在此基础上分析一些高扇出的单元，并按设计规则约束修复它们，然后再进行一些逻辑优化，还有时序、拥塞和功耗的优化，最后做细致地布局，对标准单元做小范围内的移动，最终可以确定所有的标准单元均有合法的布局。布局时，时序分析只需考虑器件延时和互连线延迟最差的情况。布局完成后，我们可以生成相应的时序报告，如图 3-14 所示，建立时间的 WNS 和 TNS 均为 0，没有存在时序违例的路径。在布局这一步，我们无需对保持时间进行优化，但是可以分析结果，以作参考。

```
----------------------------------------------------------------------

Scenario: func_wc_cmax   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Design  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0


Scenario: func_wc_cmax  (Hold)  WNS: 0.22  TNS: 5.31  Number of Violating Paths: 26
Design (Hold)  WNS: 0.22  TNS: 5.31  Number of Violating Paths: 26

----------------------------------------------------------------------
```

图 3-14 布局后的时序结果

在布局后，需要在设计中插入电压钳位单元 tie cell，使得普通器件的引脚不会直接与 VDD 和 VSS 相连，而是通过 TIEHI 与 VDD 相连，通过 TIELO 与 VSS 相连，对器件作 ESD 防护，同时也将普通的信号与电源信号区分开来，方便之后做 LVS 验证。

3.3.4 时钟树综合

接下来，我们需要对设计做时钟树综合。时钟树综合的目的是使时钟的根节点到所有叶单元的时钟端口的长度尽量相等且尽量短，使时钟分析的结果尽可能地接近理想情况。根据表 3-1 所示的四种时序库和三个 RC corner，我们也相应地创建四种典型的 scenario，也就是 func_wc_cmax（Slow+cmax）、func_tt_ctyp（Typical+typ）、func_ml_cmin（Fast（0℃）+cmin）和 func_lt_cmin（Fast（-40℃）+cmin），由工具在这四个 scenario 下分别对设计做时序分析。若设计的时序可以在这四种 scenario 下皆收敛，则在其他工作环境中基本也不会出现时序问题了。时钟树综合后的结构如图 3-15 所示。

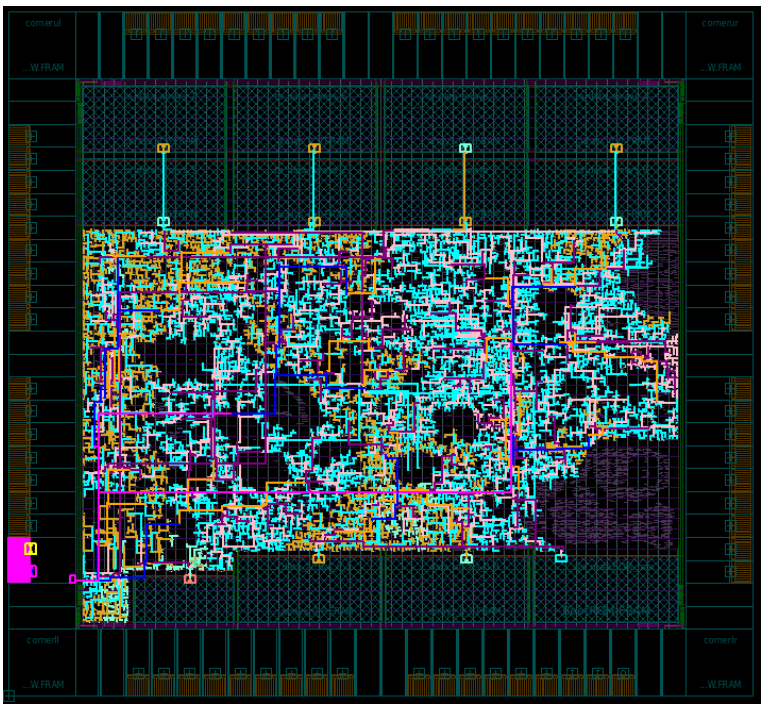


图 3-15 时钟树综合后的时钟树结构

时钟树优化后，产生时序报告，各个时钟在各个 scenario 下的时钟偏差（Clock Skew）如表 3-2 所示。由于时钟 qspi_sck 下没有时序逻辑单元，故也不存在时序路径，不考虑其时钟偏差。

表 3-2 各个时钟在各个 scenario 下的时钟偏差（单位：ns）

scenario / clock	func_wc_cmax	func_tt_ctyp	func_ml_cmin	func_lt_cmin
clock	0.1112	0.1190	0.0786	0.0777
jtck	0.0613	0.0840	0.0976	0.0967

此时在 func_wc_cmax、func_tt_ctyp 和 func_ml_cmin scenario 下仍存在不超过 150ps 的最大转换时间的时序违例，在 clock 时钟的 func_wc_cmax scenario 下存在 30ps 的建立时间违例，在 clock 和 jtck 时钟的 func_lt_cmin scenario 下均存在一定的保持时间时序违例，在 clock 时钟的 func_ml_cmin scenario 下也存在一定的保持时间时序违例，可以再对时钟树做几次增量优化，尽量修复这些问题，其中，少许的保持时间时序违例可以暂时忽略，留到布线后再去解决。时钟树优化和布线后产生的时序报告结果如图 3-16 所示，所有 scenario 下均没有建立时间时序违例，一条路径存在保持时间时序违例，slack 约为 9ps，预计可以在布线后得到解决，此时暂且忽略，先向后进行。

```
------------------------------------------------------------------

Scenario: func_wc_cmax   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Scenario: func_lt_cmin   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Scenario: func_ml_cmin   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Scenario: func_tt_ctyp   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Design  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0


Scenario: func_wc_cmax  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Scenario: func_lt_cmin  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 1
Scenario: func_ml_cmin  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Scenario: func_tt_ctyp  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
Design (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 1

------------------------------------------------------------------
```

图 3-16 时钟树布线后的时序结果

3.3.5 布线

时钟树综合后，我们的设计进入布线阶段，由工具自动对信号线布线。工具会先对整个设计做全局布线，然后计算出尽量少地产生拥塞的方法，将布线分配到各个 GRC，放置到布线通道上，接下来做详细布线，工具将设计中的互连线连接起来并连到器件的 pin 上，过程中不断分区修复产生的 DRC，最后进行最终的优化。考虑到天线效应的影响，布线前应对工具设置好天线规则，使工具自动在布线时在需要的位置插入天线二极管，解决天线效应的问题。对布线后的设计进行增量优化后，最终产生时序报告结果如图 3-17 所示，可以看到布线后的设计在所有 scenario 下均没有建立时间时序违例，也没有保持时间时序违例。

```
----------------------------------------------------------------

Scenario: func_wc_cmax   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_lt_cmin   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_ml_cmin   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_tt_ctyp   WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Design  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)


Scenario: func_wc_cmax  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_lt_cmin  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_ml_cmin  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Scenario: func_tt_ctyp  (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)
Design (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0  (with Crosstalk delta delays)

----------------------------------------------------------------
```

图 3-17 布线后的时序结果

可以用 IC Compiler 大致检查一下布线后的设计有无 DRC，检查报告如图 3-18 所示，可初步认为布线后的设计并无 DRC 和天线效应违例。

```
Verify Summary:

Total number of nets = 56668, of which 0 are not extracted
Total number of open nets = 0, of which 0 are frozen
Total number of excluded ports = 0 ports of 0 unplaced cells connected to 0 nets
                                 0 ports without pins of 0 cells connected to 0 nets
                                 0 ports of 0 cover cells connected to 0 non-pg nets
Total number of DRCs = 0
Total number of antenna violations = 0
Total number of voltage-area violations = no voltage-areas defined
Total number of tie to rail violations = not checked
Total number of tie to rail directly violations = not checked
```

图 3-18 布线后 DRC 检查报告

我们还使用 IC Compiler 产生了 LVS 检查报告，如图 3-19 所示。工具在设计中查得 20 个浮空的端口，经确认，这些端口均为寄存器的 QN 端，我们认为这些端口浮空没有问题，可忽略该项错误。可初步认为布线后的设计没有 LVS 错误。

```
** Total Floating ports are 20.
** Total Floating Nets are 0.
** Total SHORT Nets are 0.
** Total OPEN Nets are 0.
** Total Electrical Equivalent Error are 0.
** Total Must Joint Error are 0.
```

图 3-19 布线后 LVS 检查报告

电源端口和电源线的连接情况报告如图 3-20 所示，布线后的设计中不存在浮空的电源物理连线，没有未连接的电源端口。

```
Checking [VSS]:
        There are no floating shapes
        All the pins are connected.
        No errors are found.
Checking [VDD]:
        There are no floating shapes
        All the pins are connected.
        No errors are found.
Checked 2 nets, 0 have Errors
```

图 3-20 布线后电源连接情况报告

3.3.6 签核与物理验证

最后，我们在设计中插入标准单元填充单元（Filler Cell），以填补标准单元之间的空隙，保持整个设计有源区的连贯性，以满足 DRC 的要求。然后我们就可以导出设计的 GDS 版图文件，并将其导入 Calibre 工具进行详细的 DRC 和 LVS 检查。考虑到天线效应，我们需利用 Calibre 工具和代工厂提供的详尽的天线规则来对设计中是否存在天线效应问题做出检查，如图 3-22 所示。物理验证通过以后，利用 StarRC 工具提取设计的 RC 参数，生成 spef 文件并与门级网表文件一同导入 PrimeTime 工具，经过几轮 ECO 迭代，最终通过签核时序分析，以此时的设计作为最终版本。由于在整个后端设计的过程中有对设计做出许多优化，我们还需生成一个门级网表，利用 Formality 工具来与逻辑综合后的门级网表进行比对，验证结果如图 3-23 所示，表明我们的设计在进行物理设计前后的逻辑功能没有发生改变。

```
                    OVERALL COMPARISON RESULTS


        #       ###################       _   _
       #        #                 #       *   *
     #   #      #     CORRECT     #         |
      # #       #                 #       \___/
       #        ###################


 Warning:  Ambiguity points were found and resolved arbitrarily.

*****************************************************************************************************************
                                  CELL  SUMMARY
*****************************************************************************************************************

 Result         Layout                      Source
 -----------    -----------                 --------------
 CORRECT        ROCKET                      ROCKET
```

图 3-21 LVS 结果

Check / Cell	Results
Check ANT_GT_CORE	0
Check ANT_GT_IO	0
Check ANT_CT	0
Check ANT_M1_thin	0
Check ANT_M1_thick	0
Check ANT_V1	0
Check ANT_M2_thin	0
Check ANT_M2_thick	0
Check ANT_V2	0
Check ANT_M3_thin	0
Check ANT_M3_thick	0
Check ANT_V3	0
Check ANT_M4_thin	0
Check ANT_M4_thick	0
Check ANT_V4	0
Check ANT_M5_thin	0
Check ANT_M5_thick	0
Check ANT_V5	0
Check ANT_M6_thin	0
Check ANT_M6_thick	0
Check ANT_V6	0
Check ANT_M7_thin	0
Check ANT_M7_thick	0
Check ANT_TV2	0
Check ANT_TM2_thin	0
Check ANT_TM2_thick	0

图 3-22 天线效应检查

```
********************************** Verification Results **********************************
Verification SUCCEEDED
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
------------------------------------------------------------
 Reference design: r:/WORK/ROCKET
 Implementation design: i:/WORK/ROCKET
 12524 Passing compare points
-------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop    BBNet     Cut     Port      DFF      LAT    TOTAL
-------------------------------------------------------------------------------------------
Passing (equivalent)         815       0       67       0       69    11571        2    12524
Failing (not equivalent)       0       0        0       0        0        0        0        0
*******************************************************************************************
```

图 3-23 形式验证结果报告

最终得到的设计的版图如图 3-24 所示。

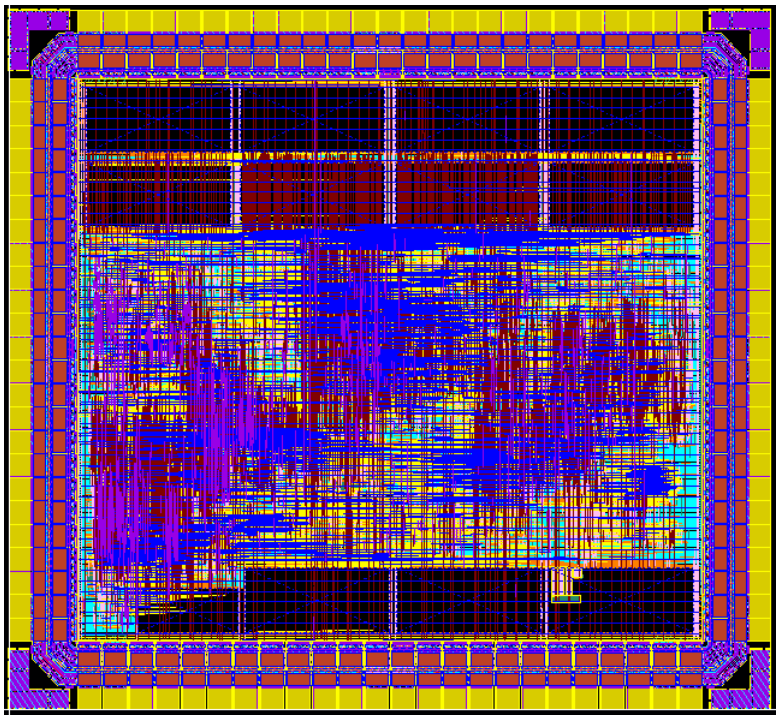


图 3-24 GDS 版图

3.4 本章小结

本章主要对基于 Rocket Chip 的 SoC 的前端设计进行了研究，生成了相应的 RTL，然后通过逻辑综合将其从 RTL 代码转化为门级网表，再通过后端物理设计得到最终的版图实现。首先，我们基于 Rocket Chip 生成了相应 SoC 的 RTL 代码。然后，我们撰写设计约束，基于代工厂的 0.13μm 工艺，利用 Design Compiler 工具，将设计从 RTL 代码转换为门级网表，并通过时序检查和形式验证。最后，我们主要利用 IC Compiler 工具通过数据准备、布图规划、布局、时钟树综合和布线等步骤完成设计的物理实现，并通过时序检查、物理验证和形式验证，得到最终的设计版图。

第四章 基于 RISC-V 的 SoC 平台的验证

4.1 基于 RISC-V 的 SoC 平台的软件模拟

基于 Chisel 的项目可以生成基于 RISC-V 的 SoC 的软件模拟器，比用 VCS 仿真快得多，且没有许可证的限制。定义顶层模块 TestHarness 作为 SoC 系统的测试平台，来进行软件模拟。利用项目生成的 C 模拟器，我们对生成的 SoC 进行了验证。

4.1.1 riscv-tests 测试集

riscv-tests 是 RISC-V 官方提供的一系列 RISC-V 架构下的测试程序，包括 benchmark 基准测试、debug 测试、isa 指令测试、mt 矩阵乘法测试的源文件、编译相关文件和一些配置。RISC-V 提供 riscv-tests 就是希望每个测试程序不论在何种 CPU 使用环境下编译执行，是否使用虚拟内存，单核还是多核，是否使用计时器中断，都能得到相同的正确结果。为了使 riscv-tests 提供的每个测试程序得到最大程度的重用，每个程序被限定为只能使用某个特定 TVM（Test Virtual Machine，测试虚拟机）的功能。TVM 通过以下定义来进行分类：

1) 可以使用的一组寄存器和指令；
2) 可以访问内存的哪些部分；
3) 测试程序开始和结束执行的方式；
4) 测试数据的输入方式；
5) 测试结果的输出方式。

如表 4-1 所示，目前 RISC-V 定义的 TVM 有以下几种：

表 4-1 目前 RISC-V 定义的 TVM

TVM 名称	描述
rv32ui	支持用户级 RV32，仅支持整数指令集
rv32si	支持管理级 RV32，仅支持整数指令集
rv64ui	支持用户级 RV64，仅支持整数指令集
rv64uf	支持用户级 RV64，支持整数和浮点数指令集
rv64uv	支持用户级 RV64，支持整数、浮点数和向量指令集
rv64si	支持管理级 RV64，仅支持整数指令集
rv64sv	支持管理级 RV64，支持整数和向量指令集

也就是说，凡是 rv32ui 的测试程序也必定可以用于 rv32u 的 TVM，可以用于这些基本的用户级 TVM 的测试程序也必定可以用于指令集扩展后的更高级别的处理器，也就是说，RISC-V 提供的这一系列 riscv-tests 就足以应对 RISC-V 开发中的绝大多数情况了。

测试程序写在“.S”汇编文件中，程序一开始便调用了 riscv_test.h 头文件，里面包含了 TVM 所使用的宏，然后指明它所针对的 TVM 名称。测试程序从 RVTEST_CODE_BEGIN 指令处开始执行，至 RVTEST_PASS 或者 RVTEST_CODE_END 结束执行，其间若出现错误，就会跳转至 RVTEST_FAIL。

以 rv64ui 的加法测试程序 riscv-tests/isa/rv64ui/add.S 在单核且禁用虚拟内存的目标环境下的使用为例，我们想要测试一下之前生成的 SoC 的加法性能。通过以下命令对加法测试程序进行仿真，且输出详细的命令执行过程：

./emulator-freechips.rocketchip.system-debug +verbose $RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-add 2>&1 | spike-dasm

得到结果如图 4-1 所示，在第 54910 个周期后测试成功完成。

```
C0:      54626 [1] pc=[0000000080000048] W[r 0=000000008000004c] R[r31=0000000080001043]
 R[r25=0000000000000003] inst=[ff9ff06f] j       pc - 0x8
C0:      54627 [1] pc=[0000000080000040] W[r30=0000000080001040] R[r 0=0000000000000000]
 R[r 0=0000000000000000] inst=[00001f17] auipc   t5, 0x1
C0:      54628 [1] pc=[0000000080000044] W[r 0=0000000080001000] R[r30=0000000080001040]
 R[r 3=0000000000000001] inst=[fc3f2023] sw      gp, -64(t5)
*** PASSED *** Completed after 54910 cycles
```

图 4-1 加法测试的部分测试结果

可以看出，在唯一一个启动的 core 0 核，第 54626 个周期，在 pc=[0x80000048]处，指令“ff9ff06f”也就是“j pc - 0x8”有效，值[0x8000004c]写入寄存器 r0，指令解码时，指令流水线已经读入 r31 和 r25 的值。其他指令周期的指令执行过程也可以类似这样去理解。

加法测试程序 add.S 中包含了很多测试指令，比如：

```
TEST_RR_OP( 2,  add, 0x00000000, 0x00000000, 0x00000000 );
TEST_RR_OP( 3,  add, 0x00000002, 0x00000001, 0x00000001 );
TEST_RR_OP( 4,  add, 0x0000000a, 0x00000003, 0x00000007 );
```

我们可以从 add.S 程序开头调用的 riscv-tests/isa/macros/scalar/test_macros.h 头文件中找到 TEST_RR_OP 的定义，如下所示：

```
#define TEST_RR_OP( testnum, inst, result, val1, val2 ) \
    TEST_CASE( testnum, x14, result, \
```

```
    li    x1, MASK_XLEN(val1); \
    li    x2, MASK_XLEN(val2); \
    inst x14, x1, x2; \
)
```

TEST_RR_OP 命令是对寄存器-寄存器操作指令进行测试的，TEST_RR_OP 的第一个参数是测试次数，第二个参数是指令，第三个参数是指令预期得到的正确结果，第四和第五个参数则是指令的两个操作数。以上述那条 TEST_RR_OP 测试指令为例，若最后两个数值相加后的结果与第三个参数不符，则会跳转至 RVTEST_FAIL，测试失败。

我们可以根据 test_macros.h 头文件中的定义对 add.S 文件中的指令逐条作分析，但汇编语言并不是易于理解的语言，这样分析效率极低。我们可以利用 RISC-V 的交叉编译工具链中的反汇编工具 riscv64-unknown-elf-objdump 来解读测试程序中的内容，执行指令如下：

riscv64-unknown-elf-objdump -d $RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-add

得到反汇编结果如图 4-2 所示，可以从中找到 pc=[0x80000048]对应的指令是“ff9ff06f”，然后跳转至 pc=[0x80000040]，与图 4-1 所示的指令操作一致，与我们上述的测试结果相符。

```
0000000080000040 <write_tohost>:
    80000040:   00001f17                auipc   t5,0x1
    80000044:   fc3f2023                sw      gp,-64(t5) # 80001000 <tohost>
    80000048:   ff9ff06f                j       80000040 <write_tohost>
```

图 4-2 反汇编的部分结果

4.1.2 软件模拟及测试结果

我们可以继续利用第二章为测试 RISC-V 交叉编译链编写的累加求和程序 test.c，通过代理内核 riscv-pk，来对 SoC 进行仿真。将之前由 RISC-V 交叉编译工具 riscv64-unknown-elf-gcc 编译程序 test.c 生成的可执行文件 test 拷贝到 emulator 目录下，启动 SoC 的 C 模拟器，执行测试程序，测试结果如图 4-3 所示，得到了正确的输出。

```
[riscv@localhost emulator]$./emulator-freechips.rocketchip.system-debug pk test
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 34765
1 + 2 + 3 + … + 100 = 5050
```

图 4-3 riscv-pk 上执行程序得到的测试结果

但这种方法需要花较长的时间，我们也可以利用 riscv-tests 测试集的项目文件来编译程序 test.c。在 riscv-tests/benchmarks 目录下新建一个名为“test”的目录，并对 Makefile 脚本文件作出修改，在 bmarks 变量中增加 test 目录，脚本中调用了 riscv64-unknown-elf-gcc 来编译该程序，生成对应的可执行文件 test.riscv。启动 SoC 的 C 模拟器，测试结果如图 4-4 所示，也得到了正确的输出，而仿真过程实则迅速得多。

```
[riscv@localhost emulator]$./emulator-freechips.rocketchip.system-debug test.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_e
nable=1.
Listening on port 46507
1 + 2 + 3 + … + 100 = 5050
```

图 4-4 裸机执行程序得到的测试结果

执行以下命令，在 emulator/output 目录下生成测试的波形文件 test.riscv.vcd，并将其转换成 vpd 格式：

rm -rf output/test.riscv.vpd.vcd && mkfifo output/test.riscv.vpd.vcd

vcd2vpd output/test.riscv.vpd.vcd output/test.riscv.vpd > /dev/null &

./emulator-freechips.rocketchip.system-debug +max-cycles=100000000 +verbose -voutput/test.riscv.vpd.vcd output/test.riscv 3>&1 1>&2 2>&3 | spike-dasm > output/test.riscv.out

可以使用 Synopsys 的 VCS 工具的 DVE 图形界面来观察生成的波形。如图 4-5 所示，我们的累加求和测试程序在 alu 模块中进行计算，程序中利用了一个 for 循环，从 1 开始，累加求和，最初 sum 为 0，加数 i 为 1，然后 sum 与 i 相加，sum 为 1，加数 i 递增为 2，接下来 sum 与 i 相加得到 3，以此类推。

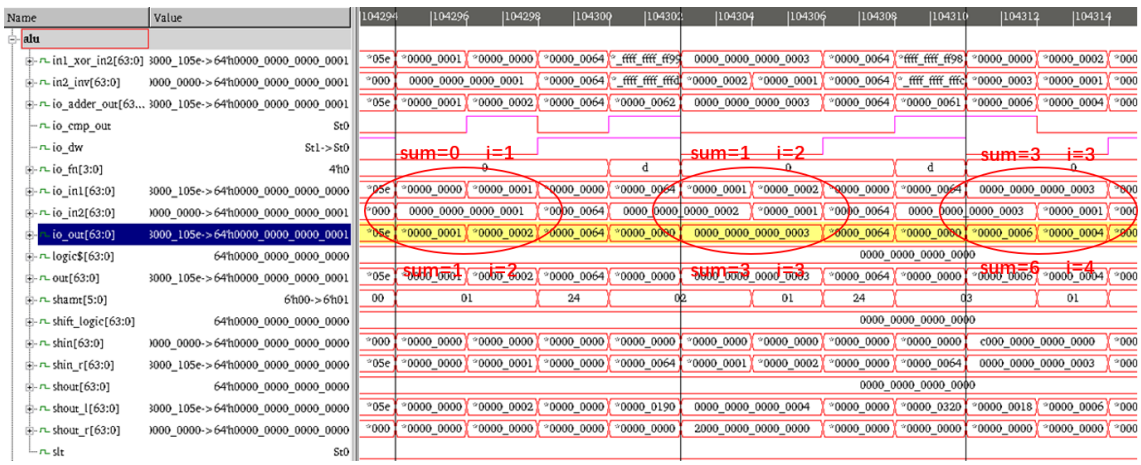


图 4-5 求和开始

加数 i 递增至 100，也就是 64'h0000_0000_0000_0064 时，此时的 sum 值为 4950，也就是 64'h0000_0000_0000_1356，二者相加得到 64'h0000_0000_0000_13ba，也就是 5050，此时加数 i 仍在递增，变为 101，然后判断 101 大于 100，循环结束，最终得到累加求和 sum 的值为 5050，与我们预期的结果相符。

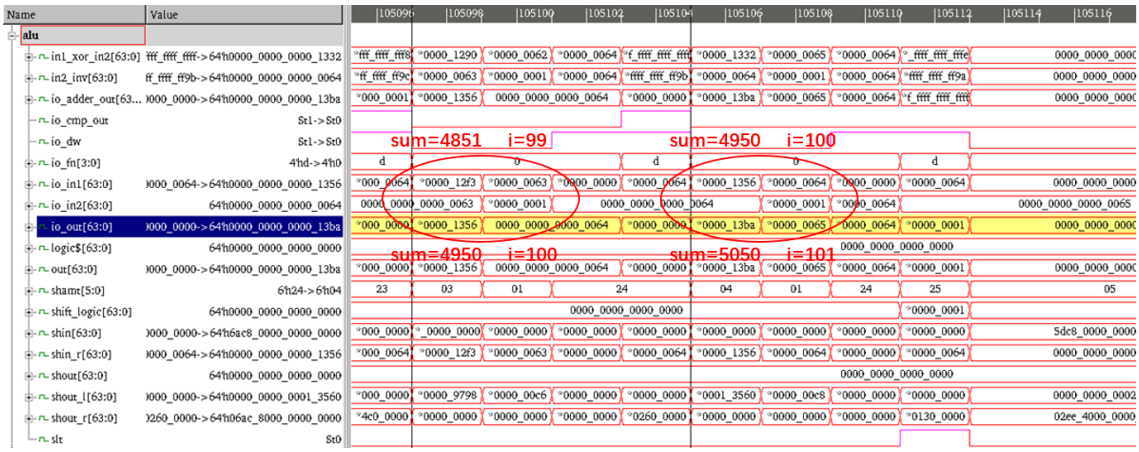


图 4-6 求和结束

4.1.3 用 GDB 调试 RISC-V 程序

我们还可以利用 GDB 来调试模拟器所执行的 RISC-V 程序。GDB 是 GNU 推出的调试工具，可以用 GDB 去查看程序执行过程中的操作。使用模拟器中的 RBB（Remote Bit-Bang）客户端，模拟器就可以与 OpenOCD 交互，OpenOCD 可以与 GDB 交互。

可以使用模拟器执行一个测试程序，同时通过 OpenOCD 和 GDB 对其进行调试。对之前使用的累加求和代码进行些许修改，命名为 prog.c，存入 riscv-tests/benchmarks/prog 目录下。代码修改后如下：

```
volatile int wait = 1;
int sum = 0;
int main() {
    int i;
    sum = 0;
    while (wait);
    for (i = 1; i <= 100; i++) {sum += i;}
    while (!wait);
}
```

仍利用 riscv-tests 的项目文件去编译该程序，在 Makefile 脚本文件的 bmarks

变量中增加 prog 目录，为了支持 GDB 调试，还需给 RISCV_GCC_OPTS 变量添加参数“-g -Og”。编译该程序得到对应的可执行文件 prog.riscv。

启动模拟器，去执行编译后的程序，如图 4-7 所示。

```
[riscv@localhost emulator]$./emulator-freechips.rocketchip.system +jtag_rbb_enable=1 --rbb-port=
9823 prog.riscv
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 9823
Attempting to accept client socket
Accepted successfully.
```

图 4-7 在带有 RBB 的模拟器上执行测试程序

接下来，我们需要启动 OpenOCD，编辑好如下的配置文件，存为 cemulator.cfg：

```
interface remote_bitbang
remote_bitbang_host localhost
remote_bitbang_port 9823
set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5
set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME
gdb_report_data_abort enable
init
halt
```

在一个新的终端以如上配置启动 OpenOCD，如图 4-8 所示。

```
[riscv@localhost emulator]$openocd -f ./cemulator.cfg
Open On-Chip Debugger 0.10.0+dev-00198-g35eed36ff (2019-11-06-16:26)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Warn : Adapter driver 'remote_bitbang' did not declare which transports it allows; assumin
g legacy JTAG-only
Info : only one transport option; autoselect 'jtag'
Info : Initializing remote_bitbang driver
Info : Connecting to localhost:9823
Info : remote_bitbang driver initialized
Info : This adapter doesn't support configurable speed
Info : JTAG tap: riscv.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0
000, ver: 0x0)
Info : datacount=2 progbufsize=16
Info : Disabling abstract command reads from CSRs.
Info : Disabling abstract command writes to CSRs.
Info : Examined RISC-V core; found 1 harts
Info :  hart 0: XLEN=64, misa=0x800000000014112d
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

图 4-8 启动 OpenOCD

再打开一个新的终端，启动 GDB，如图 4-9 所示。

```
[riscv@localhost emulator]$riscv64-unknown-elf-gdb prog.riscv
GNU gdb (GDB) 8.0.50.20170724-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog.riscv...done.
(gdb)
```

图 4-9 启动 GDB

给 GDB 的 remotetimeout 设置一个等待目标响应的时间，然后加载测试程序，一步步调试程序，如图 4-11 所示，程序执行后停留在“while(wait)”处，通过赋值 wait=0 使程序继续执行，最后，变量 sum 的值由“0”变为“5050”，与我们预期的结果一致。

```
(gdb) set remotetimeout 50000
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x0000000000010058 in ?? ()
(gdb) load
Loading section .text.init, size 0x1c2 lma 0x80000000
Loading section .tohost, size 0x48 lma 0x80001000
Loading section .text, size 0x6b8 lma 0x80001048
Loading section .rodata, size 0x158 lma 0x80001700
Loading section .rodata.str1.8, size 0x48 lma 0x80001858
Loading section .sdata, size 0x4 lma 0x800018a0
Start address 0x80000000, load size 2662
Transfer rate: 56 bytes/sec, 443 bytes/write.
```

图 4-10 加载程序

```
(gdb) print wait
$3 = 1
(gdb) print wait=0
$4 = 0
(gdb) c
Continuing.
keep_alive() was not invoked in the 1000ms timelimit. GDB alive packet
 not sent! (2110). Workaround: increase "set remotetimeout" in GDB
^C
Program received signal SIGINT, Interrupt.
0x0000000080001086 in main () at ./test/test.c:8
8       while (!wait);
(gdb) print sum
$5 = 5050
```

图 4-11 调试程序

4.2 基于 RISC-V 的 SoC 平台的 FPGA 原型验证

4.2.1 建立项目工程

定义顶层模块 FPGAChip，在代码块中例化了模块 Platform，并给它的输入输出引脚添加 IOBUF，这就是 FPGA 系统的顶层了，基于此生成可用于 FPGA 实现的 RTL。

启动 Vivado 工具，利用脚本建立我们的项目工程，将之前编译生成的 RTL 文件以及项目调用的模块都添加至项目的搜索路径中，目标板设置为 arty，xc7a35ticsg324-1L。

由于复用了 Xilinx 的 IP，首先创建了 Xilinx 的时钟分频 IP 核 clk_wiz，将其配置为 MMCM，三个输出时钟的频率分别为 8.388MHz、65.000 MHz 和 32.500 MHz，然后将生成的 IP 例化为 mmcm 模块并添加至项目工程中，以主时钟 CLK100MHZ 作为它的输入时钟。再创建一个 Xilinx 的在线调试工具 IP 核 ila，将测量的信号的格式设置为 naive，设置五个信号采样通道，其中两个数据宽度为 32bit，两个为 64bit，还有一个为 97bit，将生成的 IP 例化为 ila 模块并添加至项目工程中。类似的，再创建一个系统重置 IP 核 proc_sys_reset 并将之例化为 reset_sys 模块，使用 mmcm 模块生成的 8.388MHz 的时钟。此外，还调用了 Xilinx 的 PowerOnResetFPGAOnly 上电复位模块来进行 JTAG 的重置，它使用的是 mmcm 模块所生成的 32.500 MHz 的时钟。

然后需要对项目添加相应的时钟、引脚等约束。除了要创建一个 100MHz 的主时钟，将 mmcm 模块输出的三个时钟作为生成时钟，还要定义好 JTAG 时钟 JTCK 以及串行时钟 qspi_sck。值得注意的是，我们创建的 JTAG 时钟与 mmcm 模块的三个输出时钟也分别是异步的，需将其约束为不同的时钟组（Clock Group），使工具不去分析它们之间的时序。

完成对项目的初始化后，接下来利用 Vivado 对其进行综合、布局布线和优化，完成后报告时序、资源占用和设计规则检查的情况。在时序检查中，要求工具分别对以下情况进行检查，确定时序约束编写无误：

1) no_clock：检查是否存在寄存器或锁存器的时钟端未与时钟根节点相连；

2) constant_clock：检查是否存在寄存器或锁存器的时钟端连到了恒定信号上；

3) pulse_width_clock：检查是否存在寄存器或锁存器的时钟端需要进行脉冲宽度检查；

4) unconstrained_internal_endpoints：检查是否存在未施加约束的时序路径；

5) no_input_delay：检查是否存在未指定输入延迟的端口；

6) no_output_delay：检查是否存在未指定输出延迟的端口；

7) multiple_clock：检查是否存在寄存器或锁存器的时钟端连到了多个时钟上；

8) generated_clocks：检查是否存在生成时钟未与时钟源相连；

9) loops：检查是否存在组合逻辑 loop；

10) partial_input_delay：检查是否存在输入端口被指定了输入延迟；

11) partial_output_delay：检查是否存在输出端口被指定了输出延迟；

12) latch_loops：检查是否存在锁存器 loop。

总的时序分析结果如图 4-12 所示，WNS 为正值 12.406ns， WHS（Worst Hold Slack）为正值 0.019ns，WPWS（Worst Pulse Width Slack）为正值 3ns，设计中没有时序违例。

```
------------------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------------------


    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints
    -------      -------  ---------------------  -------------------
     12.406        0.000                      0                20289
    WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints
    -------      -------  ---------------------  -------------------
      0.019        0.000                      0                20289
    WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
    --------     --------  ----------------------  --------------------
       3.000        0.000                       0                  8441


All user specified timing constraints are met.
```

图 4-12 时序报告

各层次的子模块在 FPGA 上的资源占用情况如图 4-13 所示。

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
FPGAChip	(top)	15110	14397	648	65	8058	5	1	2
(FPGAChip)	(top)	10	10	0	0	37	0	0	0
Platform	Platform	15085	14373	648	64	7995	5	1	2
ResetCatchAndSync_d20	ResetCatchAndSync_d20	4	4	0	0	20	0	0	0
spi_dq_0_sync	SynchronizerShiftReg_w1_d3	1	0	0	1	1	0	0	0
spi_dq_1_sync	SynchronizerShiftReg_w1_d3_0	1	0	0	1	1	0	0	0
spi_dq_2_sync	SynchronizerShiftReg_w1_d3_1	1	0	0	1	1	0	0	0
spi_dq_3_sync	SynchronizerShiftReg_w1_d3_2	1	0	0	1	1	0	0	0
sys	System	15077	14369	648	60	7971	5	1	2
fpga_power_on	PowerOnResetFPGAOnly	0	0	0	0	1	0	0	0
ip_mmcm	mmcm	0	0	0	0	0	0	0	0
inst	mmcm_clk_wiz	0	0	0	0	0	0	0	0
ip_reset_sys	reset_sys	15	14	0	1	25	0	0	0
U0	proc_sys_reset	15	14	0	1	25	0	0	0

图 4-13 资源占用报告

设计中存在的设计规则检查违例如图 4-14 所示。其中，有 15 个 BUFC-1 警告，

一部分输入缓冲 IOBUF/IBUF 没有连接任何负载，这是由于设计中确实没有使用它们。我们一项项将其排除，确定这些警告都可以忽略。

Rule	Severity	Description	Violations
BUFC-1	Warning	Input Buffer Connections	15
DPOP-1	Warning	PREG Output pipelining	2
DPOP-2	Warning	MREG Output pipelining	2
PLIO-8	Warning	Placement Constraints Check for IO constraints	1
REQP-1617	Warning	use_IOB_register	1

图 4-14 DRC 报告

最终利用 Vivado 工具编译生成项目的 Bitstream，准备进行 FPGA 实现。

4.2.2 FPGA 原型验证

本次设计中使用 ARTY A7 开发板来进行 FPGA 实现。ARTY A7 是 Xilinx 推出的 Artix-7™ FPGA 系列的开发板，与 Vivado 设计工具完全兼容，便于设计者们的使用。它具备 UART，SPI，I2C 和以太网 MAC 等多种接口，可以满足设计者们对通信接口的众多需求，还包含 12 个 32 位计时器，可以作为计时器来使用。本次设计中使用的 Xilinx Artix-7 XC7A35T 提供了 5200 个 slice，每个 slice 包含 4 个 6 路 LUT 和 8 个 FF，1800 Kbits 的 Block RAM，5 个时钟管理单元，每个时钟管理单元包含 1 个锁相环，1 个混合模式时钟管理器，和 90 个 DSP slice，内部时钟频率超过 450MHz，内嵌了一个 ADC 模块，可以通过 JTAG 和 4 路 SPI Flash 编程[45]。

为了使用 ARTY A7 开发板与主机进行通信，我们需要通过一根 Micro USB 数据线将开发板的 Micro USB 接口与主机的 USB 接口连接起来，以此给开发板供电，还需要一个 JTAG 调试工具，本次设计中使用的是 Olimex ARM-USB-TINY-H，通过一根 USB-A 转 USB-B 数据线将 JTAG 调试器与主机连接起来，通过跳线将 JTAG 调试器与开发板连接起来，以此向开发板上传文件。

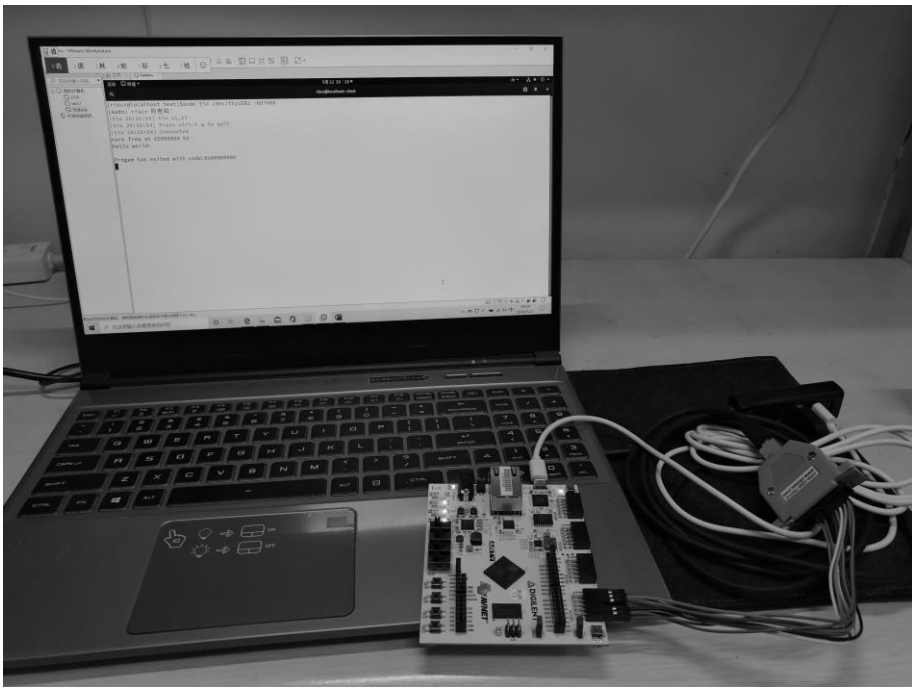


图 4-15 软硬件调试系统示意图

准备好 Arty 开发板后，我们需要一根 USB 线将其与主机连接起来，还需要连接好 JTAG 调试工具用于测试。为了使用 USB 端口，我们需要在 Linux 系统中安装 Xilinx USB 驱动，在 Vivado 安装目录中的 data/xicom/cable_drivers/lin64/install_script/install_drivers 目录下，可以找到可执行文件 install_drivers，在终端中运行它即可安装 Xilinx USB 驱动。为了使用 JTAG 调试工具，我们可以编写 udev 规则，对用到的设备给予用户所在的 plugdev 用户组的权限，将以下规则存入 /etc/udev/rules.d/99- openocd.rules 文件中：

SUBSYSTEM=="usb", ATTR{idVendor}=="15ba", ATTR{idProduct}=="002a", MODE="664", GROUP="plugdev"

SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba", ATTRS{idProduct}=="002a", MODE="664", GROUP="plugdev"

当然，我们也可以用 sudo 去执行命令。

连接好开发板后，我们利用 Linux 系统的命令查看串口设备。首先，所有串口终端的名称都在/dev 目录下，我们可以执行 ll /dev/tty*命令查看这些串口终端的权限，可以看出都需要 root 权限，也就是说，我们对串口的所有操作都需要在 root 权限下进行。然后，可以执行 dmesg | grep ttyS 命令查看串口终端的状态，得到以下信息：

ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 16550A

可以执行 dmesg | grep tty 命令查看系统开机以来串口连接与断开连接的情况，得到以下信息：

usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB0

usb 1-2: FTDI USB Serial Device converter now attached to ttyUSB1

usb 1-2: FTDI USB Serial Device converter now attached to ttyUSB2

ttyUSB*即为 USB 转串口线所使用的端口设备，为了得到更明确的信息，可以执行 ls /dev/serial/by-id 命令查看连接的串口设备，得到以下信息：

usb-15ba_Olimex_OpenOCD_JTAG_ARM-USB-TINY-H_OL0E8E8C-if01-port0 - > ttyUSB0

Olimex ARM-USB-TINY-H 正是本次设计中使用的 JTAG 调试设备，可以看出，它与 ttyUSB0 端口相连。

我们选择使用 tio，一个简单 TTY 终端 I/O 应用程序，来进行串口通信。将 tio 工具的波特率配置为 57600。UART 串口通信一般会选用数字更大的端口，根据之前的信息，也就是将 tio 工具连接至 ttyUSB2 端口设备。

Arty A7 开发板可以通过板载的四路 SPI Flash 芯片上电配置，可以利用 Vivado 工具生成 Bitstream 或者 mcs 文件（Memory Configuration File）。由于 Vivado 的 Lab 版本和 WebPACK 版本都对 ARTY A7 开发板提供了免费的支持，我们使用 Vivado v2017.1 的 WebPACK 版本即可进行闪存编程。

启动 Vivado 工具后，打开我们的项目工程，然后打开 Hardware Manager，选择 Xilinx Artix-7 XC7A35T 开发板，添加之前生成的 mcs 文件，由于设计中的 BootROM 中已写入一个上电即跳转至地址 0x20400000 的程序，也就是 Arty A7 开发板的 SPI Flash 的 0x00400000 字节，mcs 文件中也包含这些内容。将我们的 RISC-V 工程烧录至开发板中。接下来利用 SiFive 托管在 Github 上的开发套件 freedom-e-sdk 编译和上传程序。借由一个最经典的 helloworld 测试程序，执行以下命令，对其进行编译：

sudo make software PROGRAM=hello BOARD=freedom-e300-arty

再将编译后的程序写入开发板的 SPI Flash 中，执行以下命令：

sudo make upload PROGRAM=hello BOARD=freedom-e300-arty

结果如图 4-16 所示，程序的输出“hello world!”通过串口打印在屏幕上，可以看出，RISC-V 程序在基于 FPGA 的 RISC-V 平台上成功运行。

```
[riscv@localhost test]$sudo tio /dev/ttyUSB2 -b57600
[sudo] riscv 的密码：
[tio 20:10:52] tio v1.27
[tio 20:10:52] Press ctrl-t q to quit
[tio 20:10:52] Connected
core freq at 65000000 Hz
hello world!

Progam has exited with code:0x00000000
```

图 4-16 在基于 FPGA 的 RISC-V 平台上运行 RISC-V 程序

4.3 本章小结

本章主要对基于 Rocket Chip 的 SoC 进行了软件模拟和 FPGA 原型验证。首先，利用项目生成的软件模拟器对构建得到的基于 RISC-V 的 SoC 的功能进行初步的验证。然后，利用 Vivado 工具和 Xilinx ARTY A7 开发板，将构建得到的 SoC 用 FPGA 实现，并利用一个测试程序，对设计进行原型验证。

第五章 基于 RISC-V 的 SoC 平台的 RTOS 移植

考虑到嵌入式开发中常需要实时操作系统来提高设计者的开发效率，本章尝试在构建得到的基于 RISC-V 的 SoC 平台上进行实时操作系统移植。

5.1 RTOS 简介

我们都知道什么是操作系统，它是一种支持计算机的基本功能并为计算机上运行的其他程序或应用程序提供服务的计算机程序。操作系统提供的服务使设计者们在编写应用程序时速度更快、方式更简单并且更易于维护。大多数操作系统看上去可以允许多个程序同时执行，这称为多任务处理。而实际上，每个处理器内核只能在特定的时间点运行一个特定的线程，这是由于操作系统的调度程序在确定何时运行哪个程序，并在每个程序之间快速切换，造成了多个程序同时执行的错觉。操作系统的调度程序如何确定何时运行哪个程序，决定了操作系统的类别。比如，在多用户操作系统（如 Unix）中，它所使用的调度程序会确保每个用户获得足够的处理时间，而桌面操作系统（如 Windows）中的调度程序会尽量确保计算机对用户的响应的保持[46]。而实时操作系统（Real Time Operating System，RTOS）中的调度程序则会提供一个具有确定性的执行模式，由于嵌入式系统经常会要求实时性，所以嵌入式系统会尤其注意这一点。实时性要求嵌入式系统必须在严格定义的时间内对特定事件作出响应，只有在可以预测操作系统调度程序的行为的前提下，才能满足实时性的要求。传统的实时调度程序会通过允许用户为每个执行线程分配优先级来实现确定性，然后调度程序会根据优先级来确定接下来要执行哪个线程。现在常见的 RTOS 主要有 Lineo 旗下的μClinux、Micrium 旗下的μC/OS-III、Redhat 旗下的 eCos、Wind River System 旗下的 VxWorksBlackBerry 旗下的 QNX、Apache 旗下的 NuttX，以及由 Richard Barry 所推出的 FreeRTOS 等等[47-53]。本文想要移植的正是 FreeRTOS。

FreeRTOS 与全球领先的芯片公司合作开发了 15 年，它主要是针对微控制器和小型微处理器的实时操作系统，由麻省理工学院免费分发开源许可证，包括其内核以及一系列不断完善的适用于各行各业的库。FreeRTOS 的可靠、开源和便于使用使其成为市场领先的 RTOS 内核，自 2011 年以来，FreeRTOS 在每次 EETimes Embedded Market Survey 中皆列第一[54]。与其他商业化的选择相比，FreeRTOS 主要优势如下：

1) FreeRTOS 为许多不同的体系结构和开发工具提供了一个统一化的独立的

解决方案；

2) SafeRTOS 姐妹项目开展的活动可确保 FreeRTOS 的可靠性；

3) FreeRTOS 功能丰富，甚至仍在持续积极地发展它的功能；

4) FreeRTOS 拥有最小的 ROM、RAM 和内存占用，RTOS 内核的二进制映像的大小一般在 6K 到 12K 字节之间；

5) FreeRTOS 的架构非常简单，RTOS 内核的核心只是三个 C 文件；

6) FreeRTOS 可免费用于商业应用；

7) FreeRTOS 的合作伙伴 WITTENSTEIN 高完整性系统以 OPENRTOS 的形式为其提供了商业许可，专业支持和移植服务；

8) FreeRTOS 可以移植到 SafeRTOS，其中包括针对医疗，汽车和工业领域的认证；

9) FreeRTOS 具备一个庞大的且仍在不断增长的用户基础；

10) FreeRTOS 拥有一个优秀的受监控的非常活跃的免费支持论坛；

11) 如有需要，FreeRTOS 会提供商业支持；

12) FreeRTOS 提供了非常多的文档；

13) FreeRTOS 扩展性好，简单且易于使用；

14) FreeRTOS 为 eCOS、嵌入式 Linux（或 Real Time Linux）乃至 uCLinux 的应用提供了更小更易于使用的实时造作系统替代方案。

RISC-V 指令集体系架构并没有指定由某个特定的 CPU 或者 SoC 来实现，是非常便于设计者去拓展的。同样的，FreeRTOS 的 RISC-V 接口也是可扩展的，FreeRTOS 提供了一个可以用于实现所有 RISC-V 的所共需的寄存器的基本的接口文件，以及实现特定功能和拓展所必需的一组宏，比如一些额外的寄存器。

5.2 基于 Spike 模拟器的 FreeRTOS 移植

先在 Spike 模拟器上尝试 FreeRTOS 的移植。FreeRTOS 官网上可以找到支持 Spike 模拟器的版本，将该例程下载下来，可以看到其目录的结构如下所示：

```
FreeRTOS
    |- Demo
        |- Common
        |- riscv-spike
            |- main.c
    |- Source
```

其中 Source 目录下包含 FreeRTOS 的所有源码，而 Demo 目录下则是针对 Spike

模拟器的示例代码。考虑到要对 FreeRTOS 是否移植成功进行测试，我们在主程序 main.c 中添加了软件定时器，首先声明回调函数如下：

```
static void x_timer_callback( TimerHandle_t x_timer );
```

然后在主函数中添加如下内容：

```
TimerHandle_t x_timer = NULL;
x_timer = xTimerCreate( "x_timer", 1000, pdTRUE, (void *) 0, x_timer_callback);
if(x_timer != NULL) xTimerStart(x_timer, 0);
```

回调函数定义如下：

```
static void x_timer_callback( TimerHandle_t x_timer ) {
printf("Time's up!\n");
}
```

然后将修改后的代码编译生成可执行文件 riscv-spike.elf，执行以下命令，利用 Spike 模拟器执行编译后的代码：

```
spike riscv-spike.elf
```

可以看到，每隔一秒会在终端中打印一个“Time’s up!”，如图 5-n 所示，FreeRTOS 移植成功。

```
[riscv@localhost riscv-spike]$spike riscv-spike.elf
Time's up!
Time's up!
Time's up!
Time's up!
Time's up!
Time's up!
Time's up!
Time's up!
```

图 5-1 在 Spike 模拟器上运行 FreeRTOS

在 Spike 模拟器上移植 FreeRTOS 成功后，接下来可以尝试在基于 FPGA 的 RISC-V 平台上移植 FreeRTOS 了。

5.3 基于 FPGA 的 RISC-V 平台的 FreeRTOS 移植

FreeRTOS 项目的源码托管在 GitHub 网站上，我们可以从网站上克隆该项目，该项目的源码中包括了所有的 FreeRTOS 接口文件和应用实例。每个 FreeRTOS 接口均包含三个通用的内核核心组件文件，以及一些针对于某个特定架构处理器或编译器的文件。此次我们所需的文件均包含在 FreeRTOS/Source/Portable/GCC/RISC-V 目录中，该目录的结构如下所示：

```
FreeRTOS / Source / Portable / GCC / RISC-V
        |- chip_specific_extensions
                |- RV32I_CLINT_no_extensions
                        |- freertos_risc_v_chip_specific_extensions.h
        |- port.c
        |- portASM.S
        |- portmacro.h
```

FreeRTOS/Source/portable/GCC/RISC-V/port.c 文件中包含了对当前所有的支持的 RISC-V 芯片通用的代码，该文件伴随一个相应的头文件 portmacro.h，和一个 GCC 所需的汇编器文件 portASM.S，这三个文件主要包含基本的中断和异常相关配置的底层函数声明。freertos_risc_v_chip_specific_extensions.h 头文件专门用于 RISC-V 架构的处理器，使基础的 RTOS 接口文件针对 RISC-V 架构有了更多的拓展。RISC-V 平台有很多种，所以 freertos_risc_v_chip_specific_extensions.h 头文件也有很多个，本次设计中使用了/FreeRTOS/Source/portable/GCC/RISC-V/chip_specific_extensions/RV32I_CLINT_no_extensions 目录下的该文件，需将该路径添加至汇编器的 include 路径中。

我们需对 freertos_risc_v_chip_specific_extensions.h 头文件中的配置进行修改。其他的 RISC-V 平台所使用的定时器实例在本次设计中无法使用，不过我们可以使用 FreeRTOS 的 Windows 接口_WINDOWS_，这个接口可以为我们的测试提供更大的余量。FreeRTOS 项目从 10.3.0 版本开始不再使用 configCLINT_BASE_ADDRESS 配置，而是改为使用 configMTIME_BASE_ADDRESS 和 configMTIMECMP_BASE_ADDRESS 设置，我们同样需对其做出定义，将 configMTIME_BASE_ADDRESS 设置为定时器的基地址，configMTIMECMP_BASE_ADDRESS 设置为定时器比较器的地址。FreeRTOS 调用外部的中断处理程序时，必须提供进入中断时 RISC-V 的 cause 寄存器的值，还需将汇编程序的命令行的 portasmHANDLE_INTERRUPT 选项设置为相应的中断处理程序。FreeRTOS 项目中，在任何函数调用中断服务程序（ISR，Interrupt Service Routine）之前，RISC-V 接口会切换到特定的中断堆栈，中断堆栈所使用的内存地址可以定义在链接脚本中，也可以在 FreeRTOS 项目的接口文件中声明为静态分配数组，我们选择第一种方法，在 flash.lds 文件中添加链接向量__freertos_irq_stack_top，这样调度程序启动前主程序所调用的堆栈可以在调度程序启动后作为中断堆栈被复用。本次设计中无需使用 vPortEndScheduler 函数。由于测试的需要，我们还需注意要使能宏定义 configUSE_TIMERS。

为了判断 FreeRTOS 是否在开发板上的 RISC-V CPU 上移植成功，我们利用 FreeRTOS 的软件定时器编写了例程。我们需要在主程序 main.c 中添加软件定时器，首先声明回调函数如下：

```
static void x_timerCallback( TimerHandle_t x_timer );
```

然后在主函数中添加如下内容：

```
TimerHandle_t x_timer = NULL;
x_timer = xTimerCreate( "x_timer", 1000, pdTRUE, (void *) 0, x_timerCallback);
if(x_timer != NULL) xTimerStart(x_timer, 0);
```

回调函数定义如下：

```
static void x_timerCallback( TimerHandle_t x_timer )
{
static int x=0;
char str[4];
itoa(x,str,10);
write(1,str, sizeof(str)-1);
write(1,"\t",1);
write(1,"Time's up!\n",11);
if(x >= 99)
{x=0;}
else
{x++;}
}
```

我们需依照开发套件中的其它示例程序，使用对应的板级支持包（Board Support Package，BSP），将修改后的 FreeRTOS 项目代码编译生成 elf 文件，并将其上传至 Arty 开发板。利用 tio 工具将主机与开发板连接起来，如图 5-2 所示，每隔一秒，通过串口在屏幕的终端上打印一个“Time's up!”，可以看出，FreeRTOS 在基于 FPGA 的 RISC-V 平台上顺利运行。

```
[riscv@localhost test]$sudo tio /dev/ttyUSB2 -b57600
[tio 20:31:14] tio v1.27
[tio 20:31:14] Press ctrl-t q to quit
[tio 20:31:14] Connected
0        Time's up!
1        Time's up!
2        Time's up!
3        Time's up!
4        Time's up!
5        Time's up!
6        Time's up!
7        Time's up!
8        Time's up!
```

图 5-2 在基于 FPGA 的 RISC-V 平台上运行 FreeRTOS

5.4 本章小结

本章主要在基于 FPGA 的 RISC-V 平台上运行了 FreeRTOS，实现了 RISC-V 的操作系统移植。首先，我们简要了解了实时操作系统和 FreeRTOS。然后，我们基于 Spike 模拟器实现了 FreeRTOS 的移植，并利用测试程序验证了其移植结果的成功。最后，基于 FreeRTOS 项目的源码，编写了一个测试实例，并利用 ARTY 开发板实现了 FreeRTOS 在基于 FPGA 的 RISC-V 平台上的移植。

第六章 全文总结与展望

6.1 全文总结

本次设计通过对RISC-V官方提供的参考处理器实现项目Rocket Chip的研究，构建了基于RISC-V的SoC，首先对基于Rocket Chip的SoC的前端设计进行了研究，基于代工厂的 0.13μm 工艺，通过逻辑综合和后端物理设计完成了 SoC 的物理实现，然后由生成的软件模拟器初步对SoC的功能进行仿真，基于Xilinx ARTY A7开发板，将构建得到的SoC用FPGA实现，并对其进行原型验证，最后在基于FPGA的RISC-V平台上运行了FreeRTOS，实现了RISC-V SoC的操作系统移植。本次设计主要完成了以下工作：

1) 对Rocket Chip进行了研究。分析研究了Rocket Chip项目的架构，搭建了RISC-V交叉编译工具链，借由Rocket Chip项目生成基于RISC-V的SoC，利用软件模拟器和一个简单的测试程序对其进行软件模拟，初步验证了其功能。

2) 对构建得到的SoC的后端物理设计进行了研究。利用Design Compiler进行逻辑综合，将设计从RTL代码转换为门级网表，然后利用IC Compiler工具完成设计的物理实现，并通过了时序检查、物理验证和形式验证，得到最终的设计版图。

3) 对构建得到的SoC的FPGA实现进行了研究。利用Vivado工具建立工程并对其进行综合，利用Xilinx ARTY A7开发板将其用FPGA实现，并进行原型验证。

4) 对RISC-V平台上的操作系统移植进行了研究。基于FreeRTOS项目的源码，编写了一个实例，利用Xilinx ARTY A7开发板实现了FreeRTOS在基于FPGA的RISC-V平台上的移植。

6.2 后续工作展望

由于时间和条件的原因，本次设计中进行后端物理设计时，没有对设计的面积做过多的约束，以后可以针对特定的情况调整所使用的利用率和floorplan形状，尽量压缩设计的面积，找到最优的方案，并且能够在交付代工厂进行流片之后，对实际的芯片进行测试，在对整个设计流程都了解并实践后，更好地理解开源处理器的应用以及更大地发挥其价值。此外，在 RTOS 移植的部分，本次设计只是利用了简单的程序进行了测试，在以后的研究中，希望可以开发一些实际的应用。

致　谢

在论文完稿之际，首先衷心感谢我的导师翟亚红老师在毕业设计的每个阶段提供的悉心的指导与帮助。在翟老师亲切的关怀和细心的督促下，学位论文才得以按预期完成。

特别要感谢王忆文老师在我的学习与生活中给予的帮助与支持。王老师平日里工作繁多，但每次项目研究中遇到难题或者论文写作中遇到瓶颈时，王老师总能适时为我解答疑难，给予指导，一起商量解决的办法。

感谢教研室的同窗们给予我的帮助和鼓励，感谢李南希、杨洲、孙悦、吴浩阳、段一杰、吴径舟、周强、任睿捷、何旭东、贾玲玲、王心弋同学，感谢大家所共同维持的良好的学习与生活氛围，祝愿大家工作顺利。

感谢张博维师姐一直以来对我的关心，祝愿你身体健康，享受生活。

感谢潘涛师弟在项目上对我的帮助，希望你有好的前程。

感谢我的室友陈昱桦、李珍和秦及贺三年以来的朝夕相伴，祝愿你们越来越美。

感谢我的父母，我的家人，感谢你们对我的支持与照顾。

感谢学院，感谢学校，感谢电子科技大学这七年来对我的培养。

参考文献

[1] W. Stallings. Reduced instruction set computer architecture[J]. Proceedings of the IEEE, 1988, 76(1): 38-55

[2] S. P. Morse, W. B. Pohlman, B. W. Ravenel. The Intel 8086 Microprocessor: a 16-bit Evolution of the 8080[J]. Computer, 1978, 11(6): 0-27

[3] 黄绍平. 个人电脑: 走向二十一世纪[J]. 机电新产品导报, 1995(07): 41-42+45.

[4] R. Casadesus-Masanell, D. B. Yoffie. Wintel: Cooperation and Conflict[J]. Management Science, 2007, 53(4): 584-598

[5] 郑飞. 超标量与超流水线混合结构微处理器 Pentium[J]. 微处理机, 1994(4)

[6] 丛欣. ARM 处理器在网络安全领域中的应用[J]. 信息安全与通信保密, 2011(5): 41-42

[7] 施蕾, 刘波, 周凯. 基于 SPARC V8 结构处理器的计算机系统设计[J]. 空间控制技术与应用, 2008(03): 48-52

[8] F. Conti, D. Rossi, A. Pullini, et al. PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision[J]. Journal of Signal Processing Systems, 2015, 84(3): 339-354

[9] 陈曦, 黄毅. 片上系统设计思想与源代码分析(附光盘)[M]. 电子工业出版社, 2008, 9-48

[10] D. A. Patterson, A. Waterman. RISC-V 手册[EB/OL]. http://crva.ict.ac.cn/documents/RISC-V-Reader-Chinese-v2p1.pdf, 2018

[11] 雷思磊. RISC-V 架构的开源处理器及 SoC 研究综述[J]. 单片机与嵌入式系统应用, 2017

[12] RISC-V Foundation. RISC-V History[EB/OL]. https://riscv.org/risc-v-history/, 2020

[13] The Linux Foundation. The Linux Foundation and RISC-V Foundation Announce Joint Collaboration to Enable a New Era of Open Architecture[EB/OL]. https://www.linuxfoundation.org/the-linux-foundation/2018/11/the-linux-foundation-and-risc-v-foundation-announce-joint-collaboration-to-enable-a-new-era-of-open-architecture/, Nov 2018

[14] K. Asanović, R. Avizienis, J. Bachrach, et al. The Rocket Chip Generator[EB/OL]. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf, April 2016

[15] CHIPS Alliance. Rocket Chip Generator[DB/OL]. https://github.com/freechipsproject/rocket-chip, April 2016

[16] SiFive. Freedom[DB/OL]. https://github.com/sifive/freedom, Dec 2019

[17] Google. BottleRocket RV32IMC Core[DB/OL]. https://github.com/google/bottlerocket, Feb 2018

[18] OpenTitan. Introduction to OpenTitan[EB/OL]. https://docs.opentitan.org/, 2020

[19] A. Ramos, J. A. Maestro, P. Reviriego. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection[J]. Microelectronics & Reliability, 2017, 78(nov.): 205-211

[20] OpenHW Group. OpenHW Group CORE-V CV32E40P RISC-V IP[DB/OL]. https://github.com/openhwgroup/cv32e40p, Jan 2020

[21] lowRISC. Ibex RISC-V Core[DB/OL]. https://github.com/lowRISC/ibex, Jan 2020

[22] pulp-platform. Ariane RISC-V CPU[DB/OL]. https://github.com/pulp-platform/ariane, Jan 2020

[23] 邓亚威. 引领芯片定制化革命——SiFive 2018 上海技术研讨会圆满召开[J]. 中国集成电路, 2018(6): 17-18+26

[24] Andes Technology. RISC-V@Andes[EB/OL]. http://www.andestech.com/en/homepage/, 2020

[25] 包云岗. 关于 RISC-V 成为印度国家指令集的一些看法[J]. 中国计算机学会通讯, 2018, 14(1): 38-44

[26] China RISC-V Alliance. 中国开放指令生态（RISC-V）联盟于第五届互联网大会宣布正式成立[EB/OL]. http://crva.ict.ac.cn/?page_id=107, 2018 年 11 月 8 日

[27] 胡振波. 手把手教你设计 CPU——RISC-V 处理器篇[M]. 北京:人民邮电出版社, 2018

[28] Nuclei System Technology. 客户案例: 兆易创新 GD32V 系列 RISC-V 通用 MCU[EB/OL]. https://www.nucleisys.com/product.php?site=lcxp

[29] 梁辰. 阿里平头哥发布首个产品玄铁 910 但这并不是 CPU[EB/OL]. https://baijiahao.baidu.com/s?id=1640026441721109034&wfr=spider&for=pc, 2019 年 7 月 25 日

[30] 消费日报网. 黄山 1 号量产商用，华米科技 AMAZFIT 在健康数据监测有新突破[EB/OL]. https://baijiahao.baidu.com/s?id=1636123472396432687&wfr=spider&for=pc, 2019 年 6 月 12 日

[31] 瓶钵信息科技. 重磅!RISC-V 平台的可信执行环境“蓬莱”正式开源[EB/OL]. https://baijiahao.baidu.com/s?id=1654577121362062020&wfr=spider&for=pc, 2020 年 1 月 2 日

[32] 余子濠, 刘志刚, 李一苇, 等. 芯片敏捷开发实践: 标签化 RISC-V[J]. 计算机研究与发展, 2019, 56(01): 39-52

[33] B. Keller. RISC-V, Spike and the Rocket Core[EB/OL]. CS250 Lab Assignment 2 (Version 091713), Sep 2013

[34] 李广军, 阎波, 水生等. 微处理器系统结构与嵌入式系统设计（第二版）[M]. 北京：电子工业出版社，2011

[35] freechipsproject. Chisel 3: A Modern Hardware Design Language [DB/OL]. https://github.com/

ucb-bar/chisel3, Jan 2020

[36] freechipsproject. Flexible Intermediate Representation for RTL[DB/OL]. https://github.com/ucb-bar/firrtl, Jan 2020

[37] UC Berkeley Architecture Research. Berkeley Hardware Floating-Point Units[DB/OL]. https://github.com/ucb-bar/berkeley-hardfloat, Jan 2020

[38] freechipsproject. rocket-tools[DB/OL]. https://github.com/freechipsproject/rocket-tools, Jan 2020

[39] UC Berkeley Architecture Research. RISC-V Torture Test Generator[DB/OL]. https://github.com/ucb-bar/riscv-torture, Jul 2018

[40] 冯钢, 郑扣根. 基于 GCC 的交叉编译器研究与开发[J]. 计算机工程与设计, 2004, 025(011): 1880-1883

[41] 刘明, 蔡启先, 周兵. 基于 newlib 的通用嵌入式交叉编译工具的构建[J]. 广西科技大学学报, 2010, 21(2): 51-54

[42] 李恺, 翁玉萍. 基于龙芯 2F 的 Glibc 库优化[J]. 电子技术, 2010, 037(010): 27-29

[43] J. Bachrach. Chisel: Constructing Hardware in a Scala Embedded Language[C]. Dac Design Automation Conference. IEEE, 2012

[44] 陈春章, 艾霞, 王国雄. 数字集成电路物理设计[M]. 科学出版社, 2008

[45] Xilinx. All Programmable 7 Series Product Selection Guide[EB/OL]. https://china.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf

[46] 张尧学, 史美林. 计算机操作系统教程-第 2 版[M]. 清华大学出版社, 2000

[47] 杜威, 慕春棣. 基于 uClinux 的触摸屏软硬件设计与关键技术分析[J]. 计算机工程与设计, 2005, 26(4): 914-917

[48] J. J. Labrosse. UC/OS-III The Real-Time Kernel or a High Performance[J]. Microsoft Press, 2009

[49] 刘伟, 綦慧. 基于精密注塑机的嵌入式监控系统研究[J]. 仪器仪表用户, 2005(3): 9-10

[50] R. Peng, X. Zheng. A Multitask Scheduling Algorithm for Vxworks: Design and Task Simulation[C], 2009

[51] 黄水长, 栗盼, 孙胜娟, 等. 基于 NuttX 的多旋翼飞行器控制系统设计[J]. 电子技术应用(03): 59-61, 65

[52] 张龙彪, 张果, 王剑平, 等. 嵌入式操作系统 FreeRTOS 的原理与移植实现[J]. 信息技术, 2012(11)

[53] 何小庆. 嵌入式实时操作系统的现状和未来[J]. 单片机与嵌入式系统应用, 2001, 000(003): 12-13,17

[54] AspenCore. 2017 Embedded Market Survey[EB/OL]. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf, April 2017

攻读硕士学位期间取得的成果

[1] 邓紫珊，王忆文，翟亚红．一种基于 FPGA 的 RISC-V 处理器上的实时操作系统移植方法[P]．中国，发明专利, 202010375612.4, 2020 年 5 月 7 日

[2] Z. Deng, Y. Gao, T. Li. Comparison on driving fatigue related hemodynamics activated by auditory and visual stimulus[C]. Proceedings of SPIE, San Francisco, CA, 2018, 10480-0A

[3] Z. Deng, Y. Gao, T. Li. Low-frequency oscillation amplitude elevation of prefrontal cerebral hemodynamics with driving duration during prolonged driving test[C]. Proceedings of SPIE, San Francisco, CA, 2018, 10481-0Y