

北京理工大学

本科生毕业设计(论文)

基于 Chisel 语言实现的 RISC-V 处理器

Chisel Implementation of RISC-V Processor

学 院:	计算机学院
专 业:	物联网工程
学生姓名:	贺 清
学 号:	1120161774
指导教师:	陆 慧 梅

2020 年 6 月 10 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：2020年6月10日

关于使用授权的声明

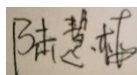
本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名：



日期：2020年6月10日

指导老师签名：



日期：2020年6月10日

基于 Chisel 语言实现的 RISC-V 处理器

摘 要

随着时代的发展，硬件的实现功能从很简单算术变为各式各样的功能，硬件设计所花费的代码量也急剧升高。硬件功能的多样化和智能化使得用 Verilog 语言来描述硬件电路所要花费的时间和空间变的更大，造成开发困难和维护困难。是否存在一种语言能够使得硬件开发更加方便呢？Chisel (Constructing Hardware In a Scala Embedded Language) 语言是由加州大学伯克利分校开发，是一种基于 Scala 的开源硬件构造语言，具有高度参数化等优点。Chisel 的特点能够有效的解决现在硬件开发时间和空间花费大的问题。本文基于 RISC-V 和 Chisel 语言设计实现一个处理器，在完成预定任务之后与同功能的 Verilog 语言处理器进行性能比较。

本文在介绍了 RISC-V 指令集架构和 Chisel 语言等相关理论基础上，针对 Chisel 高度参数化、构造简单等特点，将原有的 Verilog 语言处理器的部分模块替换成由 Chisel 语言编写的模块。本设计实现参考了一款开源的 RISC-V 架构的 Verilog 语言处理器。通过使用 Chisel 语言将模块中各种信号进行灵活的链接。研究设计了大部分原有的处理器模块，只保留了原先 Verilog 语言处理器中各个阶级传递的模块。然后研究对比了两种语言处理器在设计思路、编写难易度、模块设计和维护代码上的优劣，并给出相应分析。最后基于原有的综合实验系统，将 Chisel 实现的处理器替换进去进行指令功能测试，并对系统中 SDRAM 传输问题进行了改进。

关键词：Chisel; Verilog; RISC-V; 优劣对比; SDRAM

Chisel Implementation of RISC-V Processor

Abstract

With the development of the times, the function of hardware has changed from simple arithmetic to various functions, and the amount of code spent on hardware design has increased dramatically. With the diversification and intelligence of hardware functions, the time and space needed to describe the hardware circuit with Verilog language becomes larger, which makes the development and maintenance difficult. Is there a language that makes hardware development easier? Chisel language is developed by the University of California, Berkeley. It is an open-source hardware construction language based on Scala, with the advantages of high parameterization. The features of chisel can effectively solve the problem of time and space consumption in hardware development. This paper designs and implements a CPU based on RISC-V and Chisel, and compares it with Verilog language CPU of the same function.

In this paper, based on the introduction of RISC-V ISA and Chisel language and other related theories, aiming at the characteristics of Chisel, such as high parameterization and simple construction, some modules of the original Verilog CPU are replaced by Chisel language modules. This design and implementation refer to an open source RISC-V architecture Verilog language CPU, through the use of the Chisel language to link various signals in the module better. Most of the original CPU modules are researched and designed, only the modules passed by different classes in Verilog CPU are retained. Then the paper studies and compares the advantages and disadvantages of the two language CPUs in terms of design idea, writing difficulty, module design and maintenance code, and gives the corresponding analysis. Finally, based on the original integrated experimental system, the CPU implemented by Chisel is replaced for instruction function test, and the SDRAM transmission problem in the system is improved.

Keywords: Chisel; Verilog; RISC-V; Comparison; SDRAM

目 录

摘 要	I
Abstract	II
第 1 章 绪论	1
1.1 Chisel 背景和研究意义	1
1.2 国外研究现状	2
1.3 本文研究内容	3
1.4 论文组织结构	4
第 2 章 相关理论与技术基础	5
2.1 FPGA 简介	5
2.2 综合实验系统	6
2.3 本章小结	8
第 3 章 用 Chisel 语言实现 RISC-V 处理器	9
3.1 处理器的总体介绍	9
3.1.1 处理器功能	9
3.1.2 指令集	10
3.2 部分模块设计实现	12
3.2.1 译码模块 Decode	12
3.2.2 执行模块 Execute	14
3.2.3 状态控制寄存器 CSR	15
3.2.4 内存管理单元 MMU	20
3.3 结果分析	20
3.3.1 测试代码	20
3.3.2 仿真与实际结果	22
3.4 本章小结	24
第 4 章 综合实验系统的改进	25
4.1 SDRAM	25
4.2 系统中 SDRAM 交互转换桥的实现	26
4.3 系统中与 SDRAM 的交互问题及改进	29
4.3.1 存在的问题	29
4.3.2 转换桥的改进	31
4.4 本章小结	34
第 5 章 Chisel 与 Verilog 对比	35

5.1 前期操作对比	35
5.2 模块设计思路对比	36
5.2.1 译码模块对比	36
5.2.2 各阶段数据传输模块对比	38
5.3 Chisel 实现与 Verilog 实现的优劣.....	40
5.4 本章小结	42
结 论	44
参考文献	46
致 谢	47

第 1 章 绪论

1.1 Chisel 背景和研究意义

随着当今世界的发展和快速增长的处理器设计，厂家想要突破原有的处理器设计瓶颈变得越来越困难。想要突破这样的瓶颈必须在设计上进行优化或者更换更好的算法。而在硬件设计这方面，研究者和开发者最先接触到的语言大概就是 Verilog，无数的基于 Verilog 语言的硬件被设计和开发。它是一种硬件描述语言，用于算法级、门级到开关级的抽象层次设计建模。但随着它的使用部分研究者认为该语言并不能完全良好的表达开发者的意图，部分设计过于繁琐复杂，而其他的硬件设计语言也有此类问题，是否存在一种语言能更好的表达开发者的意图，并减少工作量，提高开发效率。

基于以上原因，加州大学伯克利分校的一群研究者们开发出了一种基于 Scala 语言的硬件设计语言 Chisel (Constructing Hardware In a Scala Embedded Language)。为什么 Chisel 是基于 Scala 开发的而不是 Verilog？在今天众多的编程语言中，Java 往往是开发者们的首选语言，市面上基于 Java 语言开发的软件程序也多的数不过来，而 Java 之父 James Gosling 曾赞美过 Scala，认为该语言为除去 Java 后的首选语言。Scala 是一门基于 JVM (Java Virtual Machine) 运行的语言，与 Java 兼容性好，能相互调用。Scala 相比 Java 并不逊色，可能会更加优秀，毕竟开发者的意图是创造一门比 Java 更加优秀、高效、好用的语言。Scala 是一门面向对象的函数式语言，它的特点正如名字一样是可自由伸缩的特性，既能裁剪已有的类，也能扩展自己想要的库；可以简简单单开发设计一个简单的脚本，也可以开发出一个复杂庞大的软件系统。开发者们可以便利的使用 Scala 的库来发开新的东西，比如设计一门新的语言。

鉴于 Scala 如此优秀的特性，研究者们开发出了 Chisel 语言。作为开发者团队的成员的 Krste Asanovic 教授认为选择 Scala 是因为它的特性能够更好的描述电路，可以方便的将它转换为 Verilog 语言；再者它操作符即方法、柯里化、纯粹的面向对象、强大的模式匹配、便捷的泛型编写、特质混入、函数式编程等特性使得 Scala 开发语言更加的方便快捷。使用 FIRRTL[1] 能更加方便的将 Chisel 转换为中间文件 FIRRTL，进而转换为 Verilog 语言文件。

Chisel 的设计解决了 Verilog 语言的不足。首先引入了 Verilog 语言没有的面向对象的特性，使设计更加理想；其次，减少了不必要的语法问题，利用 Scala 中便利的设定将复杂的重复过程简单化。原先 Verilog 是用于电路验证的，因此存在许多不可综合的语法。而在 Chisel 到 Verilog 的转换过程中不会采用不可综合语法，在设计的时候就能排除不可综合的设计。最后利用 Scala 的特性能够迅速的改变电路结构，完成设计与修改，能够更好的维护修改庞大 SoC 系统。

1.2 国外研究现状

Chisel 最先是由加州大学伯克利分校的几十名成员开发完成的硬件构造语言。他们仅仅花费了 1 年的时候就完成了从 RISC-V 指令集设计到 Rocket 芯片的成功流片。

最为大家熟知的 RISC-V 架构的 Chisel 语言开源 SoC 生成器为 Rocket Chip，它包含了由 Core、Cache 以及互连等构成的模块库，以此为基础构建一个完整的 SoC，并且可以生成可综合的 RTL 代码。

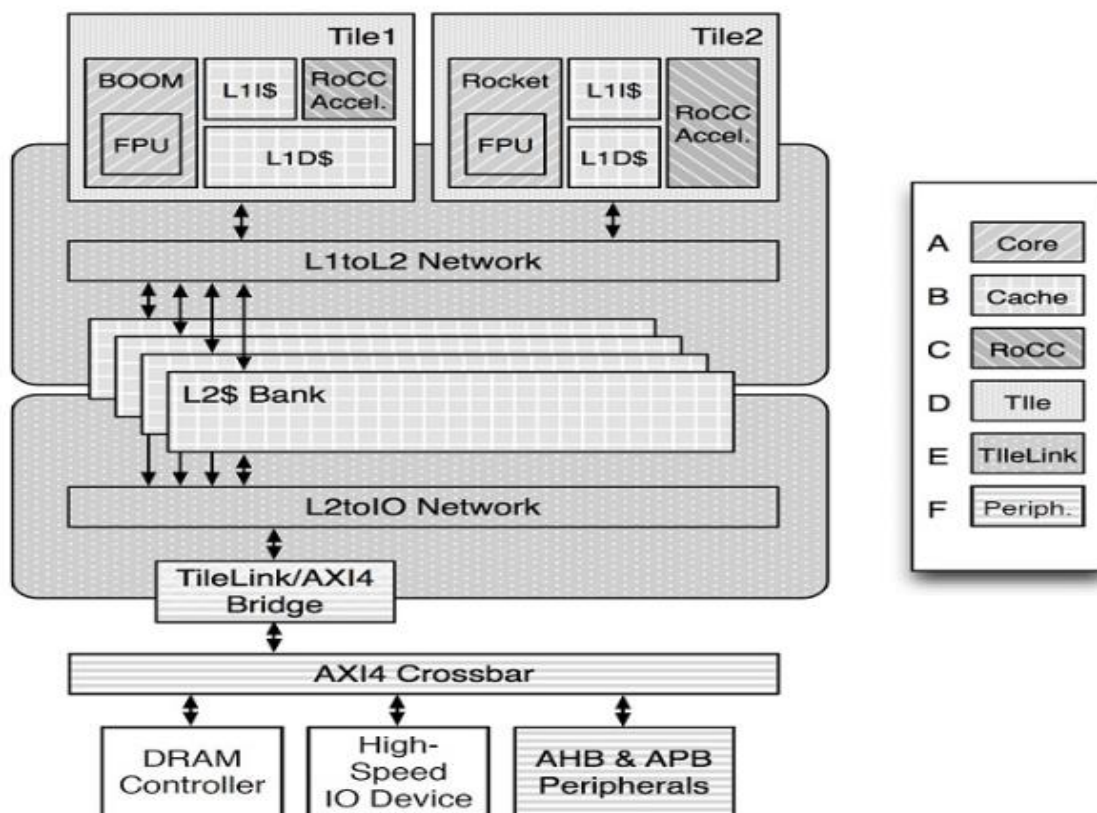


图 1-1 Rocket Chip 生成器包含: A) Core 生成器 B) Cache 缓存生成器 C) RoCC 协处理器生成器 D) Tile 块生成器 E) TileLink 生成器 F) 外设

Rocket Chip 生成器的构造如图 1-1 所示。Rocket Chip 主要有两个用途，第一个是它可以用来生成 RISC-V 的 RTL 实现，该实现具有顺序执行的流水线、浮点运算单元、多级缓存、虚拟内存和其他模块；另一个用途是把它作为基础的函数库来使用，可以在其他的实现中复用 Rocket Chip 中的部分模块，比如 BOOM(伯克利另一个 RISC-V 乱序执行 superscalar 实现)[1]将 Rocket Chip 当函数库使用，复用了其中的 core 和 cache 功能。Rocket Chip 中还包含了可以帮助开发者用来开发的帮助工具和硬件模块生成器，使其可以作为可复用的函数库来使用，使得开发更加的方便快捷。

还有其他很多利用 Chisel 实现的 RISC-V 架构处理器，比如 BOOM，RV64 乱序处理器；BottleRocket，RV32IMC 微处理器。还有很多使用 Verilog 实现的 RISC-V 架构的处理器，如 PicoRV32，Clifford Wolf 设计的 RV32 处理器，是本实验所使用的综合实验系统中原有的 SoC 系统，该系统在后面的章节简易介绍。

1.3 本文研究内容

本文研究实现基于 Chisel 语言的 RISC-V 架构处理器，处理器主要功能包括：正常五级流水，RV32IMA 指令集(指令编码长度为 32 位，32 位地址空间，支持 32 个通用寄存器)，支持 RISC-V 定义的中断和异常，支持 MMU。具体研究内容和工作如下：

(1)研究设计基于使用 Chisel 语言的 RISC-V 架构处理器，参考开源的同 RISC-V 架构的 Verilog 语言处理器进行设计，完整实现原有个各种功能。设计中保留了原有的 Verilog 语言处理器中的各个阶段数据传输控制的模块。最后将完整的处理器移植进实验系统中进行测试。

(2) 研究同功能下 Verilog 语言处理器和 Chisel 语言处理器的优劣，对比将从初步认识语言、模块设计思路和代码维护难易度等方面进行，从仿真波形方面进行对比分析。

(3)将处理器替换进综合实验系统之后，改进实验系统中存在 SDRAM 读写问题，使实验系统在实际环境中的运行能够更加的稳定。

1.4 论文组织结构

Chisel 语言实现处理器的各个模块是本文研究的重点。论文共有六章，每个章节安排如下所示：

第 1 章 绪论，讨论了本文的研究目的，从为什么选择 Chisel 语言、国外研究发展两个方面阐述，明确了研究内容和结构安排。

第 2 章 相关理论与技术基础，对实现系统所需的 FPGA 平台、综合实验系统进行详细阐述。

第 3 章 具体的 Chisel 语言处理器的实现过程，总述处理器功能实现，介绍指令集所包含的指令，对典型模块进行设计分析，讲解设计思路，最后通过仿真测试和上板测试来对设计进行功能验证。

第 4 章 两种语言实现的处理器的对比，从前期接触到中期设计测试到后期维护修改三个方面进行对比。

第 5 章 基于综合实验系统中 SDRAM 读写问题的改进，综合实验系统由于更改了 Soc 框架导致了总线与 SDRAM 数据交互发生了问题，从总线转换桥中的状态设置和时钟设置入手进行改进，使能够花费更少的时间稳定 SDRAM 读写。

总结与展望，对全文内容进行归纳并提出未来的发展目标。

第 2 章 相关理论与技术基础

基于 Chisel 语言设计的 RISC-V 架构处理器的最终目的是要在现实环境中能够进行计算任务和程序的运行。本章对实验所需要的实验环境开发板和综合实验系统进行介绍。

2.1 FPGA 简介

FPGA (Field Programmable Gate Array) 是在 PAL、GAL 等可编程器件的基础上进一步发展的产物。它是作为专用集成电路 (ASIC) 领域中的一种半定制电路而出现的, 既解决了定制电路的不足, 又克服了原有可编程器件门电路数有限的缺点。

与传统的芯片设计进行对比, FPGA 芯片优点在于:

- (1) FPGA 由逻辑单元、乘法器等资源组成, 可实现乘法器、寄存器等硬件电路。
- (2) FPGA 可使用 Verilog 来设计, 从门电路到 FIR 电路。
- (3) FPGA 可以无限制重新编程, 加载新方案时间极短, 减少硬件开销。
- (4) FPGA 工作频率可以通过修改设计来改变, 来满足不同的需求。

FPGA 芯片的缺点如下:

- (1) FPGA 所有功能依靠硬件实现, 无法实现分支条件跳转。
- (2) FPGA 只能定点运算

本次实验使用的是小脚丫 STEP-CYC10 的基于 Intel Cyclone10 设计的 FPGA 开发板, 芯片型号是 10CL016YU256C8G。另外, 板卡上集成了 USB Blaster 编程器、SDRAM、FLASH 等多种外设。板上预留了 PCIE 子卡插座, 可方便进行扩展。其板载资源如下:

表 2-1 板载资源

资源种类	数量	资源种类	数量
LE 资源	16000	可扩展 STEP-PCIE 接口	1 个
片上存储空间	504Kbit	集成 USB-Blaste 编写器	1 个
DSP blocks	56 个	SDRAM	64Mbit
PLL	4 路	Flash	64Mbit
Micro USB 接口	2 路	三轴加速度计 ADXL345	1 个

数码管	4 位	USB 转 Uart 桥接芯片 CP2012	1 个
RGB 三色 LED	2 个	12M 与 50M 双路时钟源	1 个
5 向按键	1 路	LED	8 路

2.2 综合实验系统

本实验所使用的实验环境由清华大学提供。

SoC，全称 System-on-a-Chip，是将一些关键组件集成在一块的系统或芯片。使用 SoC 系统可以很好的减少空间开销，使空间利用率的到提高，以方便提供更多的空间给其他模块。

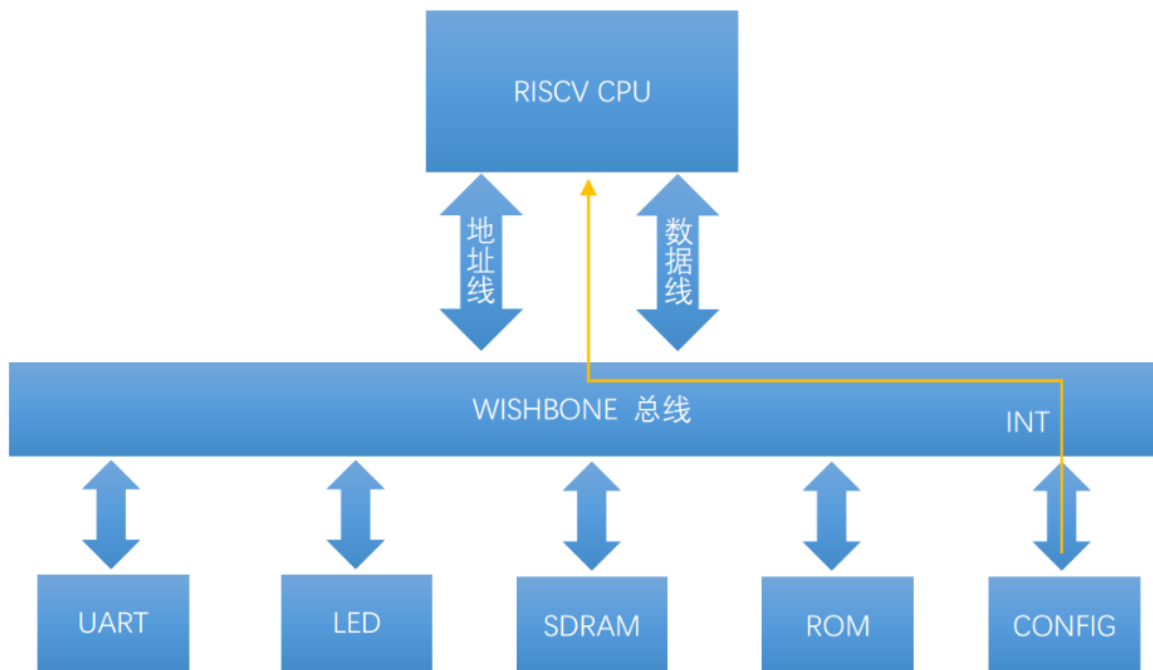


图 2-1 综合实验系统 SoC 总体框架

本次使用的综合实验系统的 SoC 框架如图 2-1 所示。RISC-V 架构处理器通过 Wishbone 协议总线与其他设备进行数据交互。而 SDRAM 和 Wishbone 交互需要进行数据处理，需要将 32 位数据与 16 位数据进行转换，此部分存在转换正确问题，此问题的改进将在第五章综合试验系统的改进部分阐述。

CONFIG 模块是为了兼容 BBL，该模块中保存了一些硬件信息提供 BBL 查询设置，另外根据 BBL 要求，timer 与 cmp 的地址也是通过内存地址访问的，这里也一并归于此模块中。当 timer 达到 cmp 的数值时会触发一个定时器中断，直接送往处理器。

ROM 模块是一个封装好的 IP 核，是保存处理器运行指令的只读存储器。该 IP 核的配置大小为 64Kb，实际使用大小为 48Kb，即程序大小必须在 48Kb 以下，或者 32Kb 以下为最优，超出该大小将无法正常使用。可以利用 Quartus 程序反复修改 ROM 中的数据来完成重新烧录新程序。

RAM 模块，随机存储模块，是处理器中进行数据交换的存储器，可以随时读写，临时存储操作系统或程序数据。该模块可以通过开关调用宏来开启或关闭，所以图中并未画出。

SDRAM 模块，同步动态随机存取内存模块，同步是指存储器工作需要同步时钟，内部命令的发送与数据的传输都以它为基准，所有行动都基于一个时钟；动态是指存储阵列需要不断地刷新电容电量以保证数据不丢失；随机存取是指数据不是线性依次存取，而是随机地址进行读/写。本实验系统采用两种 SDRAM 来实现此模块，一种为手把手写 CPU 中提供的开源 SDRAM 程序，另一种是使用 qsys 的 SDRAM 的 IP 核。第一种 SDRAM 使用空间有限，无法存储太大的数据；第二种 IP 核式 SDRAM 在 Wishbone 总线到 Avalon 总线的转换中存在问题，读出的数据存在错误的乱码，此问题的修改在第五章详细说明。

UART 模块，串口模块，通过接受或者发送数据来与上位机进行交互。此模块设置为停止位 1 位，无校验位，波特率 115200，在仿真的时候需要在接受端模拟一个串口。

LED 模块，简单的开发板 LED 控制模块，通过变化 LED 的二进制码来进行简易的 LED 亮灭控制，通过 LED 的亮灭传播简易的信息。

综合实验系统的设计者构建了这基于 RISC-V 架构的实验平台，并对后续 CPU 的替换进行了便捷性设计，使替换处理器只需修改关于处理器的 Verilog 语言文件，整体框架并不需要改变。之前系统中处理器为一种集成 IP 核中的一部分，如果要替换处理器需要更改 IP 核设计，复杂且繁琐，并不利于实验。后续系统维护者对操作系统方面的功能进行了修改完善，修改了 BBL 后移植了 32 位 rcore 操作系统，实验

系统运行操作系统的功能进行了验证，使得实验系统更加稳定。本文后续实验基于该综合试验系统，使用的系统功能仅验证设计实现的处理器的正确性。

2.3 本章小结

小脚丫 STEP-CYC10 系 FPGA 开发板能很好作为本次研究实验的搭载系统，只需使用 Intel 的 Quartus 软件就能轻松将程序烧录至开发板中。清华大学提供的综合试验系统能完美的契合本次设计的处理器，为处理器的移植测试提供了良好的环境。

第 3 章 用 Chisel 语言实现 RISC-V 处理器

Chisel 语言是使用率比 Verilog 语言低的一种优秀硬件构造语言，随着越来越多的高校对 Chisel 在硬件上的研究，它的特点就越来越明显。本章从处理器的典型模块的设计出发，阐述 Chisel 在处理器设计上的优势与对原有处理器的改进。

3.1 处理器的总体介绍

3.1.1 处理器功能

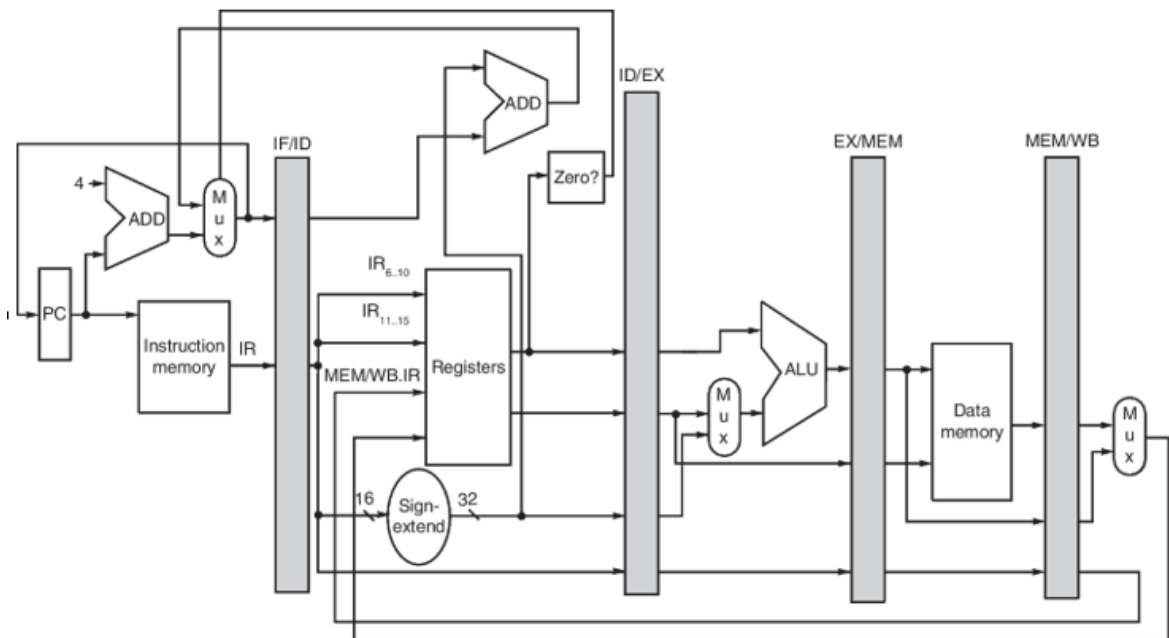


图 3-1 处理器的五级流水架构[12]

本实验设计的处理器的架构图如图 3-1 所示，采用的传统的五级流水架构，支持 M 态和 S 态特权级，还具有 MMU 和 TLB，无 CACHE 设计，实现 RISC-V 中规定的异常、自陷和中断处理。

本实验设计的处理器的五级流水线如图 3-1 所示，取指阶段根据 PC 模块的输出值在指令存储器中寻找对应地址指令，找到指令后将指令传送给译码阶段；译码阶段接受指令，根据预设的指令特征码解析指令，并拆分指令中的有效信息，然后将指令

执行所需信号传递到执行阶段；执行阶段根据指令执行不同的功能，最终将结果传递给存储阶段；如果指令不为存储器相关，则将结果传递给写回阶段，反之根据指令信息进行存储器操作；写回阶段根据指令不同将指令执行产生的结果传递到不同的阶段。如果存在需要暂停流水线的指令，根据指令的不同暂停各阶段数据传递，等待指令的执行完成并取消暂停信号。

处理器顶层文件的数据传输采用的是 Wishbone 总线协议，指令传输和数据传输分开。总线工作方式为，当 STB 选通信号有效时，即为高电平时，视为主设备请求发起一次总线操作；然后总线周期信号 CYC 有效，表示总线正在使用；传输过程中主设备发送地址、数据选择、写使能信号给从设备，当从设备通过握手接受传输操作后，才能进行数据传输，从设备将数据返回给主设备。

3.1.2 指令集

本实验处理器所使用的指令集为 RISC-V 中的 RV32I 基础指令集及其 M(基本乘法扩展集)、A(基本原子操作集)两种扩展指令集。RV32I 指令集为基础的 32 位整数指令集，32 位地址空间，寄存器为 32 位，其中包含整数的基本运算，读取和存储的基本指令。扩展指令集 M 为标准整数乘法扩展集，其中包含了整数寄存器中的乘法指令，实现获得两数相乘的结果或者除法产生的余数和商。扩展指令集 A 为标准操作原子扩展集，其中包括对存储器的原子读、写、修改和处理器间的同步的指令[12]。

RV32IMA 指令集的指令具有 32 位固定长度，并且需要 32 位地址对齐，支持指令变长扩展，扩展长度为 16 倍数，需要 16 位地址对齐[12]。32 位指令最低 2 位必为“11”，当所有位为 0 或 1 时为非法指令。RISC-V 默认采用小端存储，即低地址存放低字节，高地址存放高字节，非标准变种中可支持大端或双端，本实验设计所采用的是小端存储。

RV32I 中有 31 个寄存器 X1-X31，用来保存整数数值，其中 X0 为常数 0 不变，还有一个 XPC 用来存放当前指令地址。

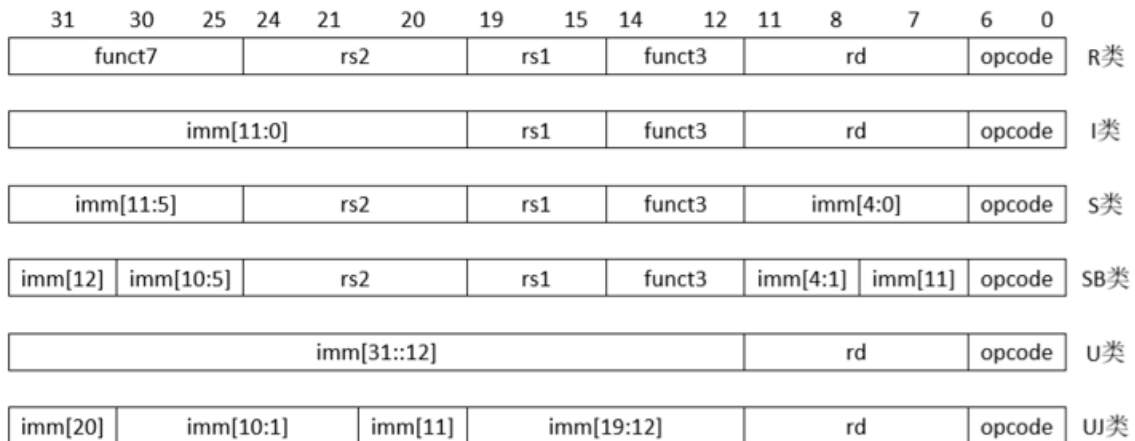


图 3-2 基本指令格式

本次实验中的指令格式如图 3-2 所示，R/I/S/U 类为核心指令格式，都是 32 位固定长度指令。SB/UI 为基于立即数处理的变种指令。

R 类指令包含 ADD/SLT/SLTU/AND/OR/XOR/SLL/SRL/SUB/SRA 等，包含两个操作数寄存器地址和一个目的寄存器地址，funct7/3 和 opcode 用来区分具体指令。

I 类指令包含 ADDI/SLTI (U)/ANDI/ORI/XORI/Load 等，包含一个 I 型立即数和一个操作数，funct3 和 opcode 用来区分具体指令。

S 类指令包含 Store 指令，将 rs2 中的数据存储在存储器中，根据指令的不同存储不同的位数。

U 类指令主要用于 LUI/AUIPC，LUI 用于构建 32 位常数，AUIPC 用于构建一个作用于 PC 上的偏移量。

无条件转移指令采用 UI 类指令格式，rd 为目标地址，立即数为偏移量。条件转移指令采用 SB 类指令格式，立即数为偏移量，rs1/2 为判断操作数，根据指令不同进行不同条件判断，然后决定是否跳转。

RV32M 扩展指令集向 RV32I 中添加了整数乘法和除法指令，其中包含 MUL/MULH/MULHSU/MULHU/DIV/DIVU/REM/REMU。除法指令中 DIV/DIVU 将商放入目标寄存器中，REM/REMU 将余数放入目标寄存器中。乘法指令中 MUL 用于获得 32 位乘法中 64 位结果的低 32 位，高 32 位根据是否需要符号由 MULH/MULHU 完成，如果操作数一个为有符号另一个为无符号则使用 MULHSU 来获得高 32 位。所以乘法需要两条指令来获得一个完整的结果。

RV32A 向 RV32I 中添加了原子操作指令，其中包括 LR.W/SC.W/AMOSWAP.W/AMOADD.W/AMOXOR.W/AMOAND.W/AMOOR.W/AMOMIN.W/AMOMAX.W/AMOMINU.W/AMOMAXU.W。AMO 指令对内存中的操作数执行一个原子操作，并将目标寄存器设定为操作前的值。加载保留和条件存储保证了它们两条指令之间的操作的原子性。加载保留读取一个内存字，存入目标寄存器中，并留下这个字的保留记录。而如果条件存储的目标地址上存在保留记录，它就把字存入这个地址。如果存入成功，它向目标寄存器中写入 0，否则写入一个非 0 的错误代码[12]。

3.2 部分模块设计实现

3.2.1 译码模块 Decode

译码模块是处理器中解析指令的核心模块，在该模块中将 32 位指令根据操作码 opcode 区别为不同的指令类型，然后根据图 3.2 中的指令类型进一步获取区分指令的特征码 funct，确认指令后开始设置信号，计算立即数和设置寄存器，最后将数据传输给执行阶段进行指令的执行。

```
def BEQ          = BitPat("b????????????????000?????1100011")
def BNE          = BitPat("b????????????????001?????1100011")
def BLT          = BitPat("b????????????????100?????1100011")
def BGE          = BitPat("b????????????????101?????1100011")
def BLTU         = BitPat("b????????????????110?????1100011")
def BGEU         = BitPat("b????????????????111?????1100011")
...
...
...
def CSRRWI       = BitPat("b????????????????101?????1110011")
def CSRRSI       = BitPat("b????????????????110?????1110011")
def CSRRCI       = BitPat("b????????????????111?????1110011")
def SLLI_RV32    = BitPat("b0000000????????????001?????0010011")
```

图 3-3 指令编码设置

首先明确一条指令需要清楚指令的特征，指令的特征包含在 opcode 和 funct 中，如图 3-3 中非“？”字所示，所有指令的特征都是独一无二的，一条指令只存在一种特征码。模块开始根据指令特征设定好所有的指令码为后面进行指令识别提供帮助。

了解指令特征之后就需要将指令中包含的有效信息，即图 3-3 中的 BitPat 码中“？”部分进行切割解读。Chisel 中的列表操作能够完美的解决指令与信号分配的问题。首先将各个指令所要用到的信号量集合起来形成一个指令信号集合，如图 3-4 所示。图中第一行为指令运算相关，根据指令所需要进行的计算运算修改，无则设为默认；第二行为寄存器读写相关，根据指令对寄存器读写的要求进行修改；第三行为立即数相关，根据立即数操作指令提供所需类型的立即数；第四行为存储器地址，是 Load/Store 指令相关的访问地址；第五行为跳转与延迟相关，根据跳转指令设定前两位信号，如果指令需要暂停流水线则设定后两种信号；第六行为中断异常特权级相关，根据系统指令的类型进行设定；第七行为 CSR 模块相关，如果指令存在对 CSR 模块的读写操作，则根据所需类型设定此类信号。

```
def default = List(d.EXE_NOP_OP, d.EXE_RES_NOP, d.Y, // aluop, alusel, ins
                  d.WriteDisable, d.ReadDisable, d.ReadDisable, // wreg, reg1, reg2
                  d.ZeroWord, // imm
                  d.ZeroWord, // mem_addr
                  d.ZeroWord, d.X, d.NotInDelaySlot, d.X, // jaddr, jflag, delay, step
                  d.X, d.X, d.X, d.X, d.X, // sys, bre, sret, mret, fence
                  d.ReadDisable, d.CSRWriteDisable) // csr_read, csr_we
```

图 3-4 指令信号集合列表

图 3-4 所展示的就是本实验中确认指令后所要设定的全部信号操作。以指令 ADDI 为例子。ADDI 是一个立即数加法指令，实现的功能为将源操作数和立即数相加，结果存到目标寄存器。当输入的指令为 ADDI 时，图 3-4 中的第一行需要更改计算器操作为 ADD 操作，设置 alusel 为算术运算 ARITHMETIC，这是一个正常指令所以 ins 非法位置为否，需要打开寄存器写操作来存储结果，打开操作数 1 的开关，配置立即数为 I 型立即数。因为是个加法指令，所以跳转延迟、CSR 等信号不需操作。最后 ADDI 的指令信号设定如图 3-5 所示。

```

ins.ADDI ->
List(d.EXE_ADD_OP, d.EXE_RES_ARITHMETIC, d.X,
    d.WriteEnable, d.ReadEnable, d.ReadDisable,
    imm_i_type,
    d.ZeroWord,
    d.ZeroWord, d.X, d.NotInDelaySlot, d.X,
    d.X, d.X, d.X, d.X, d.X,
    d.ReadDisable, d.CSRWriteDisable),

```

图 3-5 ADDI 指令的设定

其他指令的设置方法和 ADDI 的设定方法完全一样，只需要更改期中的参数值即可变为不同指令的设定，方便快捷，方便扩展。设定完成后只需要将设定量按照顺序赋值给输入输出的信号即可完成指令的解码工作。

在解码中使用 List 的好处在于能够快速添加、删除或修改指令，添加删除只需要在特征码定义和列表中进行操作，而修改只需找到对应指令修改参数值即可，方便各个指令调试后的修改，也方便阅读。

而 Verilog 中指令译码的设计是首先根据指令类型的不同分为立即数指令、跳转指令、Load 类指令、Store 类指令、系统调用指令、CSR 类指令、原子操作指令和其他指令；再根据指令第二特征码的不同细分为具体指令，然后对指令设定其信号量。这种方法的处理过程繁琐，需要细分指令类型，不如 Chisel 中指令特征码的整体设计方便。

3.2.2 执行模块 Execute

当完成上一小节的指令译码过程后，就需要根据译码的结果完成指令所希望的工作，而这部分工作由执行阶段执行模块进行调度分配。本实现的执行部分分为五个部分：逻辑运算、位移运算、CSR 运算、基本算数运算和乘除法运算。

基本算数运算主要是进行加减运算。逻辑运算主要进行操作数位操作运算，包括 OR、AND、XOR 等，位运算结束后返回结果值到目标寄存器中存储。位移运算主要是完成译码阶段发送来存在位移操作数操作的运算，比如 SLL、SRL、SRA 等操作，位移运算结束后返回结果值到目标寄存器中存储。乘法运算主要是根据指令的不同返回两个 32 位数相乘后的高低位部分。除法运算较为复杂，需要设定相关除法的操作，比如除法器空闲、是否完成上一次除法和是否有符号的设定，然后将这部分信号传递给单独的除法模块，花费较长的时钟周期后根据指令的不同返回商或者余。这四部分

运算结合整理为运算模块 ALU，如图 3-1 中 ALU 部分，可以根据指令增加新的运算操作。

除了进行运算外，该执行模块还需要将部分数据处理后传送给存储器模块，让存储器获得存储格式、地址和数据。当译码层传送 CSR 相关读写信号时，执行模块需要将 CSR 使能信号和 CSR 寄存器地址传递给 CSR 模块，等待 CSR 返回读取数据，根据指令的不同对数据进行处理后输出给其他模块。如图 3-6 所示。

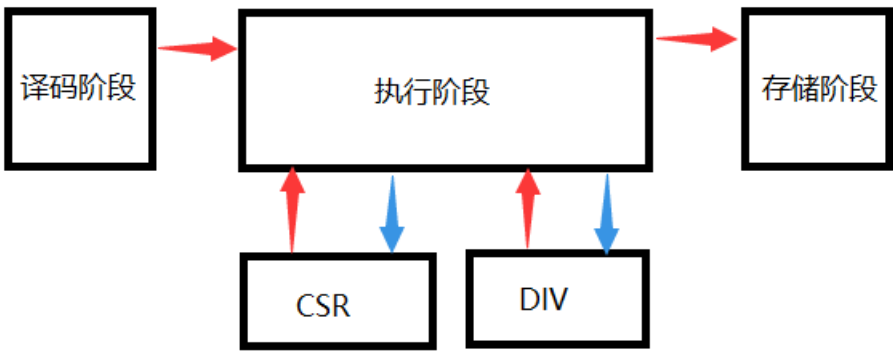


图 3-6 执行模块数据传输图

3. 2. 3 状态控制寄存器 CSR

CSR 状态控制寄存器是 RISC-V 架构定义的一个配置处理器和记录运行状态的寄存器。该寄存器是内部寄存器，使用的其自己的地址编码空间，与存储器无关。

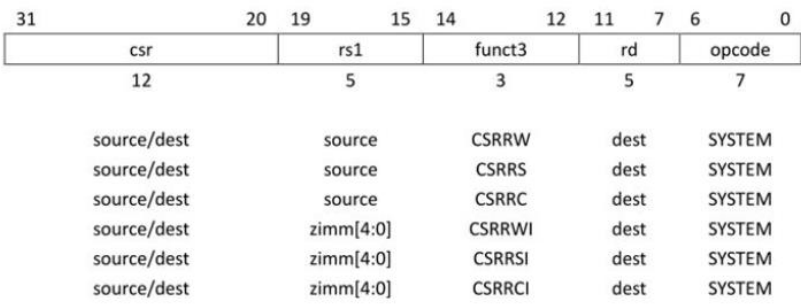


图 3-7 CSR 指令格式

CSR 指令及其格式如图 3-7 所示。

CSR_{RRW} 用来原子性的交换 CSR 和寄存器中的数值。指令读取 CSR 的值写入 rd 寄存器中，将 rs1 的值写入 CSR 中，完成值交换。

CSR_{RS} 读取 CSR 中的值，根据 rs1 中被置 1 的位置来确定 CSR 相同可写位为 1，然后将转换后的值存入 rd 寄存器中。

CSR_{RC} 与 CSR_{RS} 相反，读取 CSR 中的值，根据 rs1 中被置 0 的位置来确定 CSR 相同可写位为 0，然后将转换后的值存入 rd 寄存器中。

后三条指令功能与前面三条对应相同，只是将 rs1 的值替换为 Z 型立即数然后进行相关操作。

CSR 寄存器通过上述指令，修改其自身的值来更新处理器的运行状态。

该模块除了设计描述记录处理器状态，还设计了机器级和管理员级两个特权级中对中断异常的处理。

级别	编码	名字	缩写
0	00	用户/应用程序	U
1	01	管理员	S
2	10	Hypervisor	H
3	11	机器	M

图 3-8 RISC-V 特权级

机器级是最高的特权级，也是 RISC-V 中必须存在的特权级。机器级可以访问机器实现，而用户级和管理员级被用于程序运行和操作系统运行，如图 3-8 所示。

在 ISA 中 SYSTEM 型操作码主要用来编码所有的特权级指令，根据图 3-4 中第六行 SRET 和 MRET 设定管理员级和机器级。这类指令可以分为两类，一是原子性读-修改-写控制和状态寄存器的指令，另一种是其他特权指令。

地址	特权	名字	描述
管理员自陷 Setup			
0x100	SRW	sstatus	管理员状态寄存器
0x101	SRW	stvec	管理员自陷处理函数基地址
0x104	SRW	sie	管理员中断使能寄存器
0x121	SRW	stimecmp	墙钟（Wall-clock）定时器比较值
管理员定时器			
0xD01	SRO	stime	管理员墙钟时间寄存器
0xD81	SRO	stimeh	stime 的高 32 位，仅 RV32
管理员自陷处理			
0x140	SRW	sscratch	管理员自陷处理函数 Scratch 寄存器
0x141	SRW	sepc	管理员异常程序计数器（exception program counter）
0x142	SRO	scause	管理员自陷原因（trap cause）
0x143	SRO	sbadaddr	管理员坏地址（bad address）
0x144	SRW	sip	管理员挂起的中断（interrupt pending）

图 3-9 S 态 CSR 地址分配

地址	特权	名字	描述
机器信息寄存器			
0xF00	MRO	mcpuid	CPU 描述
0xF01	MRO	mimpid	Vendor ID 和版本号
0xF10	MRO	mhartid	硬件线程 ID
机器自陷 Setup			
0x300	MRW	mstatus	机器状态寄存器
0x301	MRW	mtvec	机器自陷处理函数基地址
0x302	MRW	mtdeleg	机器自陷转移（delegation）寄存器
0x304	MRW	mie	机器中断使能寄存器
0x321	MRW	mtimecmp	机器墙钟（Wall-clock）定时器比较值
机器定时器和计数器			
0x701	MRW	mtime	机器墙钟时间寄存器
0x741	MRW	mtimeh	mtime 的高 32 位，仅 RV32
机器自陷处理			
0x340	MRW	mscratch	机器自陷处理函数 Scratch 寄存器
0x341	MRW	mepc	机器异常程序计数器（exception program counter）
0x342	MRW	mcause	机器自陷原因（trap cause）
0x343	MRW	mbadaddr	机器坏地址（bad address）
0x344	MRW	mip	机器挂起的中断（interrupt pending）

图 3-10 M 态 CSR 地址分配

特权级的实现是根据特权级中寄存器设置来实现的。图 3-9 与 3-10 给出了 S 态和 M 态下各个 CSR 地址的分配，在该分配下分别对不同特权级下的寄存器进行设置。

首先从最高特权级机器级（M 态）的各种寄存器进行介绍。

机器状态寄存器 mstatus 是一个 32 位可读写寄存器，其设定如图 3-11 所示。

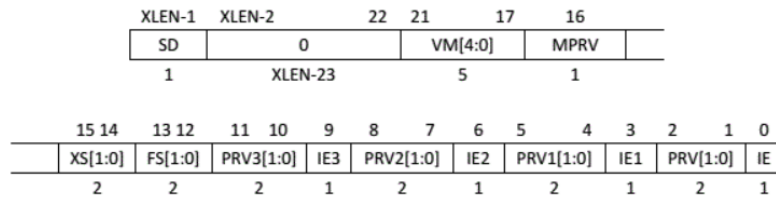


图 3-11 机器状态寄存器 mstatus

本实验中主要需要对 VM[4:0] 字段进行编写，模式为 Mbare，无翻译和保护，采用机器物理地址；Sv32，采用 32 位虚拟寻址页面，用于支持操作系统。

机器自陷转移寄存器 mtdeleg，是一个可读写 32 位寄存器，包含一个只读的零，用来自陷定向机器模式，是必须实现的寄存器。

机器中断寄存器 mip 和 mie，是两个可读写 32 位寄存器，mip 寄存器中存储挂起中断的信息，mie 寄存器中存储中断使能。

机器定时器寄存器 mtime，当 mtime 的数值达到一定值得时候将触发一次定时器中断，此寄存器用来稳定时钟。

机器异常程序计数器 mepc，一个 32 位可读写寄存器，当发生自陷时，mepc 存储发生异常的指令的虚拟地址。

机器原因寄存器 mcause，用来保存中断或异常标识代码，记录的中断或异常如图 3-12 所示。

中断 (Interrupt)	异常代码 (Exception Code)	描述
0	0	指令地址未对齐
0	1	指令访问失效 (fault)
0	2	非法指令
0	3	断点
0	4	Load 地址未对齐
0	5	Load 访问失效 (fault)
0	6	Store/AMO 地址未对齐
0	7	Store/AMO 访问失效 (fault)
0	8	从 U-mode 进行环境调用
0	9	从 S-mode 进行环境调用
0	10	从 H-mode 进行环境调用
0	11	从 M-mode 进行环境调用
0	≥12	保留
1	0	软件中断
1	1	定时器中断
1	≥2	保留

图 3-12 中断或异常 mcause 的值

机器级地址寄存器 `mbadaddr`，当发生取地址异常、取值访问异常、L/S 地址异常、L/S 访问异常时，该寄存器存入失效的地址，此时 `mepc` 存储指令起始部分地址。

管理员级相关寄存器的设定和机器级的寄存器设定相同，是机器级设定的子集。此外管理员级还需对页表进行管理。管理员页表基址寄存器 `sptbr` 用来保存当前根页表的管理员物理地址。

在 Sv32 模式中时，管理员模式是工作在 32 位分页的虚拟存储器系统中的，该系统能够支持如 Unix 操作系统的运行。Sv32 实现支持了一个 32 位的虚拟地址空间，该空间被分割成 4KB 的页面。一个虚拟地址被分为虚拟页号和页内偏移。实验中通过设置 PTE 的值来维护页的读写操作，PTE 格式如图 3-13 所示。

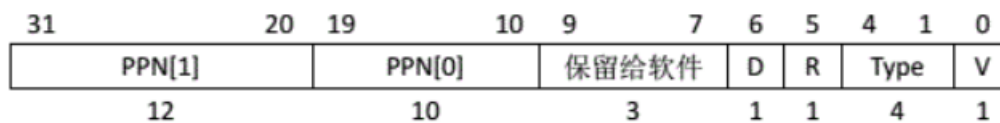


图 3-13 Sv 页表项

V 位表示 PTE 是否有效，为 0 时 PTE 可被软件任意使用；Type 是当 V 不为 0 时对 PTE 的定义，是指向下一个页表还是指向叶子。PTE 的更新对于其他写 PTE 操作来说必须是原子的。

了解了各特权级中各种寄存器的设定后，开始对中断处理过程进行介绍。

当 CSR 寄存器接收到状态更新信号的时候，根据输入的数据更新各个二进制位上的数字来更新状态。以中断的处理为例，当寄存器 `mie` 位置为 1 时，表示存在一个中断使能，开始进行中断处理；根据特权级的不同更新的寄存器也不同，管理员级的中断使能寄存器为 `sie`。假设为机器级中断，中断为指令异常，则需要判断是机器级或是管理员级的软件中断，还是机器级或是管理员级的时间中断，又或者是机器级的额外指令中断。根据判断的不同，更新 CSR 中对应中断码并返回相关错误地址给 PC 模块进入中断处理。如果是管理员级发生的中断，则只需要更新 CSR 中与管理员级相关的寄存器值即可，相关操作和机器级相同。

3.2.4 内存管理单元 MMU

内存管理单元，MMU，是负责处理器中内存访问请求的模块。该模块功能包括虚实地址的转换、内存保护和高速缓存的控制。本实验中主要实现虚实地址的转换。

TLB, Translation Lookaside Buffer, 简称快表, 存储了当前最有可能被访问到的页表项。本实验 MMU 模块中使用了 TLB 表, TLB 的数据由 CSR 进行设定。TLB 由标识和数据组成, 标识存储的是虚拟地址的一部分, 数据存放的是物理页号等信息。TLB 映射分为三种: 全关联映射、直接映射和分组关联映射, 本次使用的为直接映射, 即一个虚地址对应 TLB 中的一项。

当处理器要访问虚拟地址的时候, 首先需要计算 TLB 是否命中问题, 如果 Mbare 模式位为 1, 则无页表选择, 永不命中; 如果模式位 Sv32, 则有 32 位虚拟页表, 根据虚拟地址的高 20 位去 TLB 中查找, 如果没有对应表, 则 TLB 未命中, 设置相应 TLB_update 位为 1, 提醒 CSR 进行更新。如果命中则直接从 TLB 表中获得物理地址然后返回 CSR 进行访问。

```
.elsewhen(io.vir_i === d.CSR_mstatus_vm_Sv32){
  hit(i) := ((io.vir_addr_i & tlb_mask(i)) === (tlb_vpn(i) & tlb_mask(i)))
```

图 3-14 计算 TLB 是否命中

如果命中则将对对应虚拟地址中的物理地址输出, 如图 3-15 所示。

```
when(tlb_exception === false.B){
  io.phy_addr_o := phy_addr(io.hit_index_o)
}
```

图 3-15 TLB 命中后输出物理地址

3.3 结果分析

3.3.1 测试代码

指令测试代码参考结合了最新的 RISC-V 测例, 需要注意的是实验系统之前使用的第三方软核 Picorv32 并不能完善的支持特权级架构, 并且存在一些作者自定义的

指令，本实验实现的处理器并不存在此问题，如果需要验证原有的和现在的不同，需要修改测试集。

对于普通的用户级指令，可以参考 Picorv32 中用到的方法，将每条指令一个个测试然后输出是否匹配的结果。测试指令集默认使用的集合为 RV32IMA 指令，即能够测试本实验实现所有的指令。如果需要单独测试其中一个指令集，在 make 编译时需带上 mode 参数，如 make mode=rv32ui。

```
TEST(lui)
TEST(auipc)
TEST(j)
TEST(jal)
TEST(jalr)
...
TEST(div)
TEST(divu)
TEST(rem)
TEST(remu)
```

图 3-16 测试用例

对于 SDRAM 的测试用例，只是简单的使用了一种函数，如图 3-17 所示。函数测试 SDRAM 的输入输出是否正常，防止 SDRAM 正常其他指令不对的情况。

```
1  #input a3 change:a0,a1,a2,a3,a4,t3
2  print_uint32:
```

图 3-17 SDRAM 测试

3.3.2 仿真与实际结果

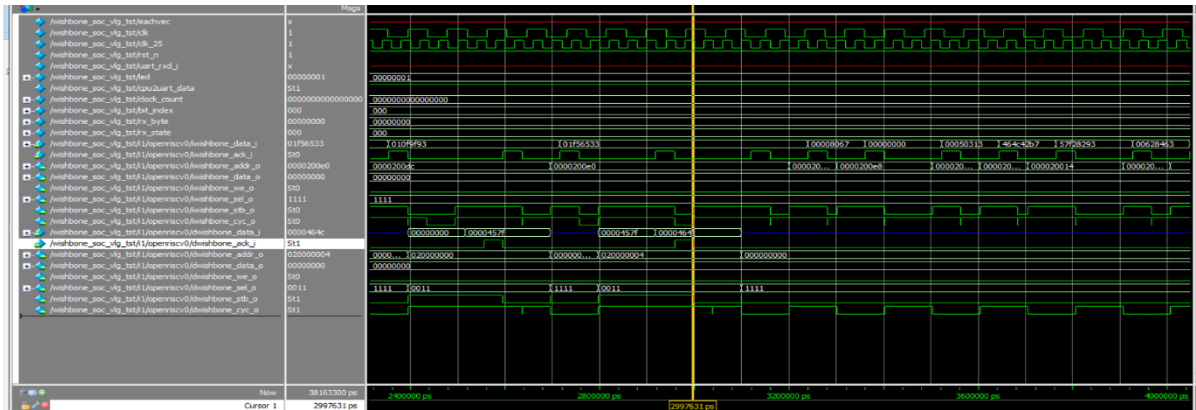


图 3-18 替换处理器前 OS 的仿真测试

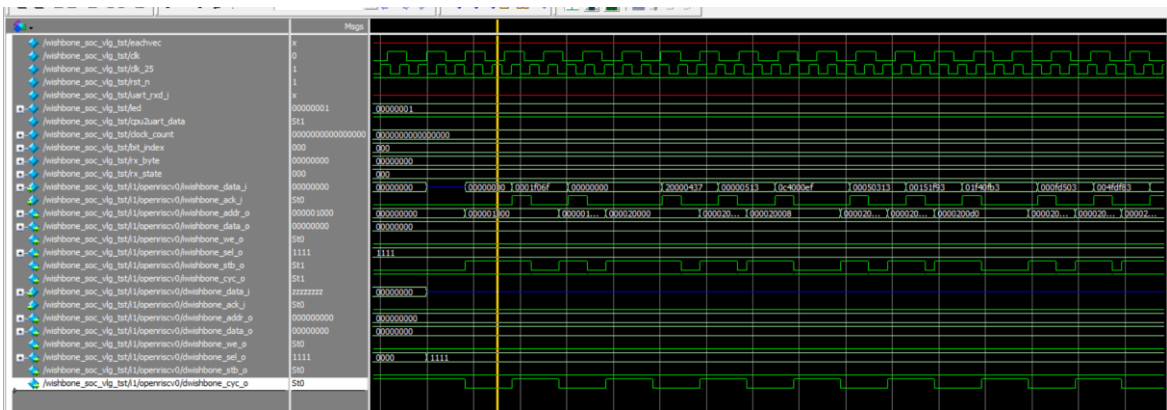


图 3-19 替换处理器后 OS 仿真测试

从图 3-18 和 3-19 可以看出，本实验设计实现的处理器在功能仿真上并没有问题，与原有的处理器计算出相同的结果。上板指令测试也完全正确，结果图如 3-20 和图 3-21 所示。



```
COM10 - PuTTY
lui..OK
auipc..OK
j..OK
jal..OK
jalr..OK
beq..OK
bne..OK
blt..OK
bge..OK
bltu..OK
bgeu..OK
lb..OK
lh..OK
lw..OK
lbu..OK
lhu..OK
sb..OK
sh..OK
sw..OK
addi..OK
slti..OK
xori..OK
ori..OK
```

图 3-20 上板指令测试(上)



```
COM10 - PuTTY
sra..OK
or..OK
and..OK
mulh..OK
mulhsu..OK
mulhu..OK
mul..OK
div..OK
divu..OK
rem..OK
remu..OK
simple..OK
sdram..OK
amoadd_w..OK
amoand_w..OK
amomaxu_w..OK
amomax_w..OK
amominu_w..OK
amomin_w..OK
amoor_w..OK
amoswap_w..OK
amoxor_w..OK
```

图 3-21 上板指令测试(下)

3.4 本章小结

本章主要介绍了实验所实现的处理器的总体功能，并从一些重要模块，如：译码模块、执行模块、CSR 和 MMU 对设计思路进行了讲解。最后使用修改后的开源测试代码对实现的处理器进行了功能测试，测试效果与预期符合，功能完全正确。测试效果图见本章倒数第二小节。

第 4 章 综合实验系统的改进

SDRAM 是一种优于普通 RAM 的存储系统，能够动态存储数据。在本实验所使用的综合实验系统中使用了一种 SDRAM 的 IP 核，但是该 IP 核在工作上存在问题，需要对其进行修改。本章对系统与 SDRAM 的数据转换桥进行了修改，重写了逻辑，同步了时序，虽然 SDRAM 工作不正常问题未能解决，但是当 SDRAM 持续工作一段时间后能达到正常输入输出的状态。

4.1 SDRAM

SDRAM，全称 Synchronous Dynamic Random Access Memory，同步动态随机存储器。同步是指存在一个 SDRAM 工作时钟，内部指令的发送与数据的传输都以该时钟为基准。动态指存储阵列需要不间断的刷新来保证数据不丢失。随机是指数据的存储不是顺序线性存储的，而是根据地址自由的进行存取。RAM，全称 Random-Access Memory，随机存取存储器，只有随机存储的能力。SDRAM 相比 RAM 来讲更加的方便快捷。RAM 这种存储结构在掉电时数据会丢失，上电时会重刷，数据也会丢失，而 SDRAM 内部是用电容进行数据存储，当断电后数据不会立马消失，而是缓慢消失，重新上电则会重新给电容充电延长数据保存。由于 SDRAM 的随机存储，需要利用地址等信息才能正确读取里面存储的数据，并不能根据一个地址和数据大小来顺序读取数据。

实验系统中使用的 SDRAM 的 IP 核配置如下图所示。

Use	Con...	Name	Description	Export	Clock	Base
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> clk_0	Clock Source			
	<input type="checkbox"/>	clk_in	Clock Input	clk	exported clk_0	
	<input type="checkbox"/>	clk_in_reset	Reset Input	reset		
	<input type="checkbox"/>	clk	Clock Output	Double-click to		
	<input type="checkbox"/>	clk_reset	Reset Output	Double-click to		
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> new_sdram_controller_0	SDRAM Controller Intel FPGA IP			
	<input type="checkbox"/>	clk	Clock Input	Double-click to	clk_0 [clk]	
	<input type="checkbox"/>	reset	Reset Input	Double-click to		
	<input type="checkbox"/>	s1	Avalon Memory Mapped Slave	avalon_sdram		
	<input type="checkbox"/>	wire	Conduit	sdram		

图 4-1 SDRAM IP 核配置

IP 核采用的是 Avalon 总线协议，与实验系统使用的 Wishbone 总线协议不同。

4.2 系统中 SDRAM 交互转换桥的实现

根据上一小节所展示的 IP 核配置可以看出，此 SDRAM 使用的是 Avalon 总线协议，是一种较为简单的片内总线。SDRAM 是从设备，使用的 slave 接口。Avalon 总线所有外设接口与 Avalon 总线时钟同步，不需要很复杂的握手协议。所有的信号都是统一的，要么全为高电平有效，要么全为低电平有效。

两种总线的示意图如下图所示，左侧为 Wishbone 总线，右侧为 Avalon 总线。

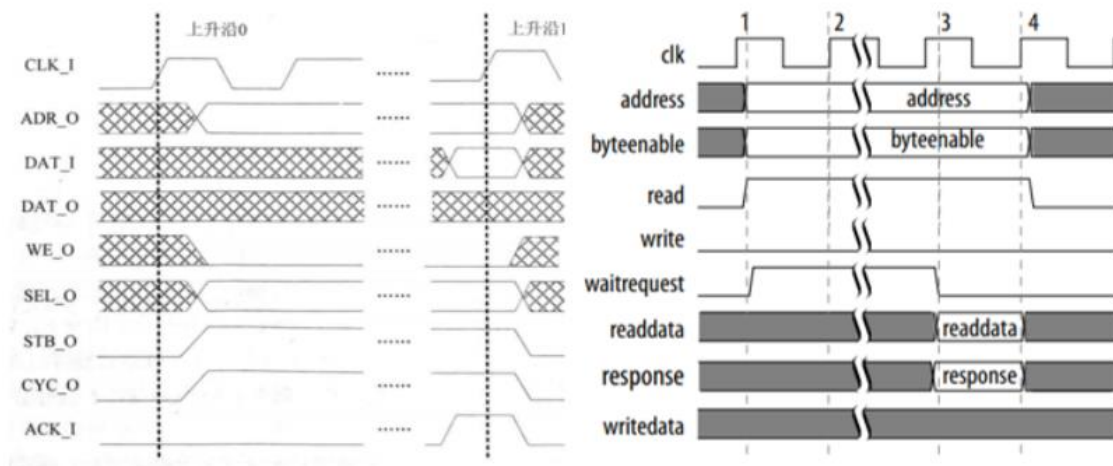


图 4-2 Wishbone 总线和 Avalon 总线示意图

Wishbone 总线协议中最关键的信号是 CYC、STB 与 ACK，CYC 为总线周期信号，表示一个主设备正在占用总线；STB 为选通信号，表示主设备发起一次总线操作；ACK 为主从设备交互结束信号。CYC 和 STB 同时拉高时表示主从设备交互开始，在整个过程中保持高电平，一直等到从设备响应 ACK 拉高后的下一周期，CYC、STB 和 ACK 拉低，至此一个交互结束。对于 Avalon 总线协议而言，关键的几个信号是 READ、WRITE、WAIT 和 READVALID。当 READ/WRITE 拉高代表读或写的请求，但是与 WISHBONE 不同的是，这个请求一直保持到 WAIT 变低，在 WAIT 为高时，从机处理请求，对于写请求来说，只需等待 WAIT 变为低电平就可以，而对于读请求来说还需要等待 READVALID 变为高电平，才表明总线交互结束。在时序上，由于 Avalon 总线协议是 SDRAM 所使用的协议，Wishbone 总线协议是整体 SoC 架构所使用的协议，所以 Avalon 总线上的时钟频率要比 Wishbone 总线上的时钟频率高。

Wishbone 总线到 Avalon 总线的转换需要根据信号的类型来进行转换。将 Wishbone 信号转换为 Avalon 信号需要将 Wishbone 总线中位数选择 SEL 与 Avalon 总线中 byteenable 信号相对应，由于 Wishbone 总线为 32 位，Avalon 总线为 16 位，所以 SEL 低两位和高两位为对应不同的 16 位数据。根据 SEL 高低的不同，需要更改地址，Avalon 总线中地址最低位置 0 表示低 16 位数据，置 1 表示高 16 位数据。因为 Avalon 总线地址长度为 22，所以地址有效位为 Wishbone 总线地址的第 2 到 22 位二进制码，最低位为高低标志码。

两种总线的相关信号转换完毕后，就必须对 SDRAM 内部读写信号进行设定以读写正确的数据。SDRAM 的读写信号如图 4-3 和图 4-4 所示。

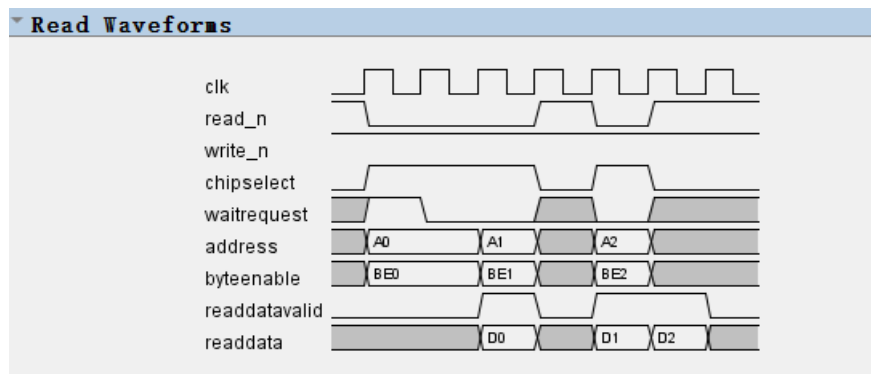


图 4-3 SDRAM 读信号波形

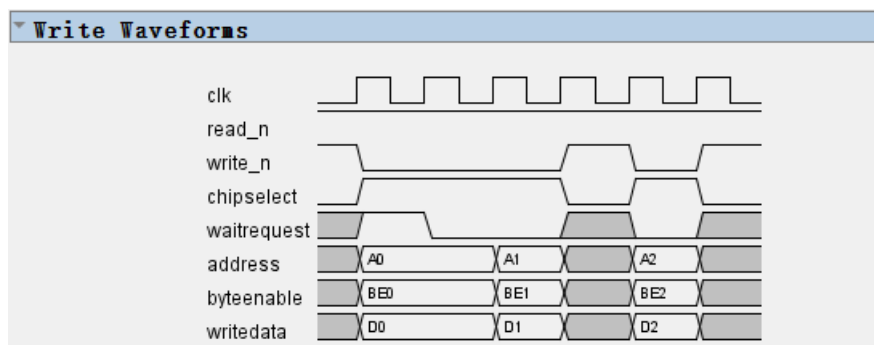


图 4-4 SDRAM 写信号波形

根据图 4-3 所示的读信号波形可以得知，当低电平有效的 read_n 为低电平时，视为一次读请求，SDRAM 进行一次读操作。此时如果片选信号 chipselect 为高电平则此次请求不被忽略。要读取 SDRAM 的数据需要对应的地址 address 和数据位数选

择信号 `byteenable`。当等待请求 `waitrequest` 无效的时候，表示 SDRAM 空闲，响应操作，当读允许信号 `readdatavalid` 有效时，开始根据输入的地址和位数信号读取相应位置的数据，读取数据完成后输入下一组数据信息。若读允许相关信号未发生改变则再次进行读，直到读不允许。

根据图 4-4 所示的写信号波形可以得知，当低电平有效的 `write_n` 为低电平时，视为一次写请求，SDRAM 进行一次写操作。此时如果片选信号 `chipselct` 为高电平则此次请求不被忽略。要写 SDRAM 的数据需要对应的地址 `address`、数据位数选择信号 `byteenable` 和要写的数据 `writedata`。当等待请求 `waitrequest` 无效的时候，表示 SDRAM 空闲，响应操作，直接往对应地址写数据。

在实验系统的 `wb32_avalon16` 转换桥中，定义了 10 种状态。分别为：准备状态、写低位等待、写低位、写高位等待、写高位、读低位等待、读低位、读高位等待、读高位、完成。

准备状态中，需要根据请求区分是读还是写操作，确认操作后跳转到对应的低位等待状态。

如果请求为写，则跳转到写低位等待，设定 `write_n` 低电平。如果没有等待请求则跳转到写低位，反之继续等待。在写低位中根据 `avalon` 总线的地址格式将 `wishbone` 地址转换为 `avalon` 地址，并输出 `wishbone` 数据的低 16 位，之后转到写高位等待。如果没有等待请求则跳转到写高位，反之继续等待。写高位输出 `wishbone` 数据的高 16 位。输出完毕后跳转到结束，设置 `write_n` 为高电平，然后返回到准备状态等待下一次请求。

如果请求为读，则跳转到读低位等待，设定 `read_n` 低电平，并且根据地址变换将数据地址和数据位数选择输出。如果没有等待且读允许则跳转到读低位，反之继续等待。在读低位中将得到的数据写给 `wishbone` 数据的低 16 位，然后跳转到读高位等待。在读高位等待中输出需求数据的地址和位数选择信号，如果没有等待且读允许，则跳转到读高位。读高位将读取的数据赋值给 `wishbone` 的高 16 位，此时一个完整的 32 位 `wishbone` 数据被读出。读取完毕后跳转到结束，设置 `read_n` 为高电平，然后返回到准备状态等待下一次请求。

读写状态转换如图 4-5 所示。

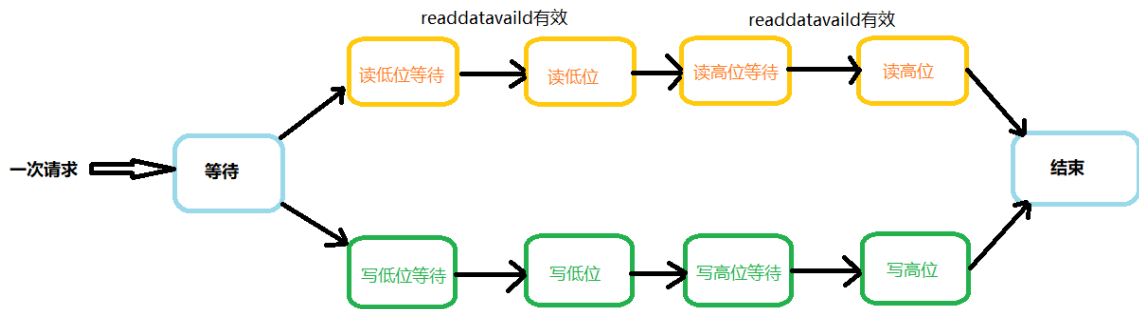


图 4-5 读写状态转换

有了这部分之后，下面解析总线转换桥的编写，参见 wb32_avalon16 代码，这里不具体分析，因为不确定是否完全正确，大致的思路是构建一个状态机，当 Wishbone 总线上有请求时，也就是 CYC 和 STB 都为 1，那么就开始进行转换工作，在开始之前，有一个等待 cnt 的操作，出于担心时序的影响，因为 setup time 小于 0，所以又等待了十几个 Sdram 频率的周期。这里 SDRAM 主频 150MHz，总线 20MHz。如果发现仍然有 CYC 和 STB 都为 1，表明这是一次有效的请求，根据总线中 we 使能情况看是写请求还是读请求，对于读请求和写请求来说，都是通过先低 16 位后高 16 位的操作，对于 Avalon 总线来说，对于写请求等待 waitrequest 信号置于低位就可以进行写操作，而读请求则另外需要等待 readdatavalid 信号置于高位才可以。事实上应该有更快的响应方式，但此处这里采用这种基本的握手规则。

需要注意的是 avalon 总线中关于 read, write 和 byteenable 都是低电平有效的，准确的说应该是对于 qsys 中这个 sdram 是这样规定的。

4.3 系统中与 SDRAM 的交互问题及改进

4.3.1 存在的问题

初步转换桥的设定如上小节介绍所示，运行前设计者的 Quartus 工程文件，将生成的二进制代码烧录至开发板中，通过串口工具观察到 SDRAM 读出了一些错误的数，如图 4-6 所示。该 SDRAM 测试是写数据全为 0010_0000，不会存在写其他数

据，但是读出的数据存在并不是此数的数，有些一眼就能观察出为错误乱码，如图 4-6 中的红框所示。使用其他开源 SDRAM 并不存在此问题，但是存在 SDRAM 实际使用空间过小问题，无法满足系统需要。

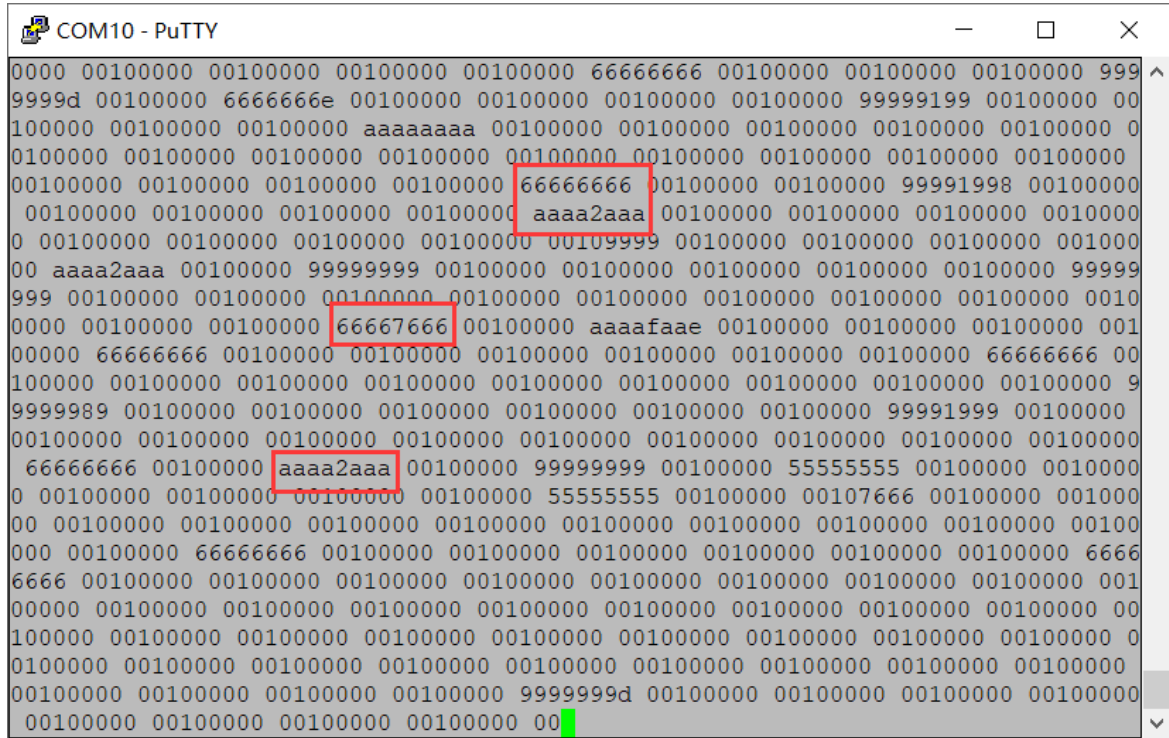


图 4-6 存在错误的 SDRAM 数据读取

针对出现的错误进行分析。在本次试验中出现了一种正确状态，当该非完全正确程序运行到一定时间后，大概 3-5 分钟时，错误数据的出现消失，只会输出正确的 0010_0000 数据。经过对资料的查询，该现象的出现可能为当程序运行一段时间后，开发板的温度提高，随着温度的升高，导致时序降低，进而可能使 SDRAM 时钟和 Wishbone 总线时钟达到一个同步值，此时输入输出达到稳定，输出正确数据。此次初步定为转换桥时序问题。

在本次设计实验之前，该问题已存在实验系统中，根据前人的工作[10]。该转换桥模块实现的是 32 位数据到 16 位数据的转换，由于 SDRAM 地址只有 22 位，所以去掉 32 位地址的高 10 位作为新的存储地址。前实验对 SDRAM 测试步骤为首先进行写一字节立马读一字节形式，结果返回并无错误。然后进行先写四字节再读四字节的形

式，结果返回无错误。最后进行连续大面积写再连续大面积读，此时结果返回产生错误数据。该出错会连续产生十个地址的错误，但不会在固定的地方出现，是随机出现。

结合上述前设计者对 SDRAM 问题的分析，对转换桥的读写进行单独诊断，使用软件 Quartus 中的逻辑分析仪功能对实际的上板读写波形进行检测，使用 chipselec 信号作为触发信号，因为当该信号处于高电平时说明正在发生一次读或写操作。图 4-7 为出现错误数据时各个重要信号的值。从图中可以看出发生错误前读操作读出的数据为正常的 0010_0000h 数据。当发生错误时，高低位读写突然混乱且数据发生异常，原有的读低 16 位数据变为高 16 位数据 0010h，原有的读高 16 位数据变为低 16 位错误数据 AAAAh，然后持续几个周期为错误数据 AAAA_AAAAh，在这之后就为正常的数据的写。根据遇到的问题进行调试，首先假设写正确，将转换桥中的写数据 wishbone_data 替换为一个常数 0000_0010，重新编译烧录后，输出的数据还是错误数据和正确数据混合，读写产生的波形还是和图 4-3、图 4-4 一致。因为写数据被固定，所以发生错误的原因可能出现在转换桥或 SDRAM 上。对此又存在几种可能，一是读时地址错误，导致读取了之前没存储数据的区域，由于没有数据读出一堆乱码；二是写数据的地址错误，导致读写地址不对应，读取不到正确数据。

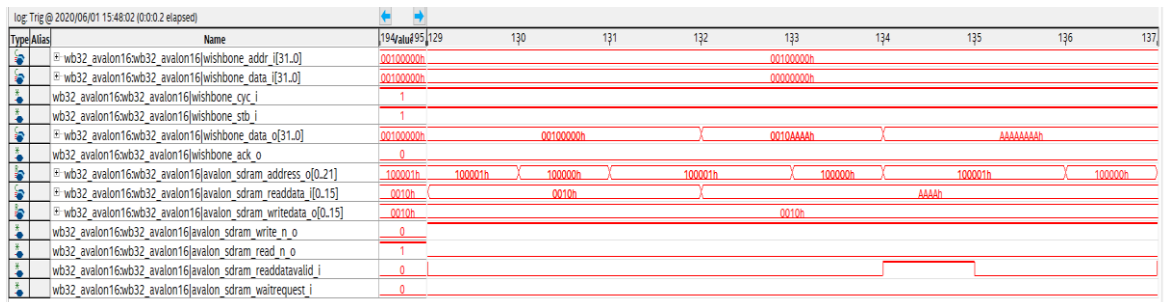


图 4-7 逻辑分析仪捕捉的错误数据波形

综合之前的猜想来看，认为转换桥的时序和读写逻辑、数据地址设定存在问题，需要对这三部分进行修改。

4.3.2 转换桥的改进

首先进行的是对读写逻辑的改进，由于原设计者设计的读写逻辑代码比较混乱，无法正确理清思路，所以需要重新进行编写，保留信号重新设定逻辑。按照 4.2 中

IP 核提供的读写逻辑图，使其符合 4.2 节中的信号设定图。重新设定后的逻辑代码如图 4-8 和 4-9 所示，编写代码为 Verilog。

```

/*****
// 写 SDRAM
// 问题
//
/*****
// if no wait request, start write low byte
state_write_wait_lo: begin
    if(!avalon_sdrām_waitrequest_i) begin
        state <= state_write_byte_lo;

    end
end

// write low byte
state_write_byte_lo: begin
    if(wishbone_sel_r[1:0] != 2'b00) begin
        // write low
        avalon_sdrām_byteenable_n_o <= ~wishbone_sel_r[1:0];
        avalon_sdrām_address_o <= {wishbone_addr_r[21:1], 1'b0};
        avalon_sdrām_writedata_o <= wishbone_data_r[15:0];

        state <= state_write_wait_hi;
    end
end

// if no wait request, start write high byte
state_write_wait_hi: begin
    if(!avalon_sdrām_waitrequest_i) begin
        state <= state_write_byte_hi;
    end
end

// write high byte
state_write_byte_hi: begin
    if(wishbone_sel_r[3:2] != 2'b00) begin // write high
        // write high
        avalon_sdrām_byteenable_n_o <= ~wishbone_sel_r[3:2];
        avalon_sdrām_address_o <= {wishbone_addr_r[21:1], 1'b1};
        avalon_sdrām_writedata_o <= wishbone_data_r[31:16];

        state <= state_done;
    end
end
/*****

```

图 4-8 写逻辑代码

```

// 读SDRAM
// 问题
//
//
/*****
// if no wait request, start read low byte
state_read_wait_lo: begin
    avalon_sram_read_n_o <= 1'b1;
    avalon_sram_address_o <= {wishbone_addr_r[21:1],1'b0};
    avalon_sram_byteenable_n_o <= 2'b00;
    if(!avalon_sram_waitrequest_i) begin
        state <= state_read_byte_lo;
    end
end

// read low byte
state_read_byte_lo: begin//
    if(avalon_sram_readdatavalid_i) begin
        wishbone_data_o[31:16] <= avalon_sram_readdata_i;
        //rdata[31:16]
        state <= state_read_wait_hi;
    end
    avalon_sram_read_n_o <= 1'b0;
    //state <= state_read_wait_hi;
end

// if no wait request, start read high byte
state_read_wait_hi: begin
    avalon_sram_read_n_o <= 1'b1;
    avalon_sram_address_o <= {wishbone_addr_r[21:1],1'b1};
    avalon_sram_byteenable_n_o <= 2'b00;
    if(!avalon_sram_waitrequest_i) begin
        state <= state_read_byte_hi;
    end
end

// read high byte
state_read_byte_hi: begin
    if(avalon_sram_readdatavalid_i) begin
        wishbone_data_o[15:0] <= avalon_sram_readdata_i;
        //rdata[15:0]
        state <= state_done;
    end
    avalon_sram_read_n_o <= 1'b0;
    //state <= state_done;
end
end
*****/

```

图 4-9 读逻辑代码

重新设定完代码逻辑后，软件编译烧录至开发板中进行测试。测试结果为还是存在图 4-6 中的问题，但是错误数据的出现次数比为改进前大幅下降。图 4-6 中错误代码的出现率为 14.6%，重写逻辑后错误代码的出现率为 4.3%。此次改进的结果为，代码逻辑的混乱影响了部分数据的读写操作，但并不是主要问题，且达到正确无误码状态也花了三至五分钟，系统并不稳定。

当反复确认信号设定与波形图一致时，开始进行对时序的修改。如果是时序存在问题，那这问题所包含的方面非常广泛，大到或许是整个 SoC 时序存在问题，小到就是转换桥时序出现了差错。由于 SDRAM 时序是比 Wishbone 时序要快很多，SDRAM 的

时序频率高达 200MHz，比处理器的 50MHz 的时序快了很多。为了改进这个问题进行了尝试性修改。由于 wishbone 速度要慢于 avalon 速度，所以如果将更新 wishbone 信号的速度与 avalon 速度同步，即以更高的速度更新信息，以保证当前请求的地址、片选、等待和占有能更快变化。于是按照这种思路，本实验将转换桥中使用 wishbone 时钟的信号设定更改为使用 SDRAM 时钟。这种情况就意味着，现在更新信号数据的速度比以前更快了，更加能够实时的掌握数据的变化。按照此思路改进完代码后编译烧录至开发板中，最后的结果为 SDRAM 的串口输出数据还是存在着错误数据，且错误数据出现率和进行逻辑改进后的相差不大，唯一发生的变化为之前需要花费大概三至五分钟才能稳定输出的过程，现在变更为只需要三十秒到一分钟就能达到稳定状态，稳定速度得到了加快。

由于整个工程项目较大，测试该问题花费时间较多，于是将只保留 SDRAM 相关模块，修改其写输入，使其从控制器中输入数据到 SDRAM 中然后进行读验证。在该最小化测试中，使用上述修改后的转换桥模块进行烧录验证，上板测试时还是出现了非正常数据，但该情况存在时间极短，大概存在了一秒该错误就完全消失，只存在正确的输出数据。通过开发板 RESET 操作后无错误数据出现情况。

最后该综合实验系统的改进只能说是基本完成，找到了修改问题的切入点：时序和逻辑，存在的 SDRAM 读写数据错误问题依旧存在，并未完全解决，只是增加了一种能够在运行一段时间后达到稳定的状态，在此状态下实验系统所进行的操作可以认为是基本正确的，无差错的。

4.4 本章小结

本章主要介绍了实验所使用的综合试验系统中的 SDRAM 以及实验系统与 SDRAM 数据交换产生的读取出错误数据的问题，并针对此问题提出了两种修改方向，并按照 2 种方向进行了修改尝试，虽然结果并没有达到预期的水平，但是存在解决问题的状态，使该实验系统能够正确运行。

第 5 章 Chisel 与 Verilog 对比

任何事物都具有两面性,Chisel 语言在硬件设计上既有它的好处也有它的不足, Verilog 语言也是如此。本章将对两种语言进行比较,从初步认识两种语言到完成一次硬件设计的思路来对比它们的优劣。

5.1 前期操作对比

大部分人初次认识 Verilog 语言的时候,都是从设计一个流水灯程序入手,毕竟 Verilog 是一门硬件设计语言,没有硬件的实际实验环境是没办法学好关于硬件设计的语言的。讲解 Verilog 语言的认识以实现流水灯为例。

用 Verilog 语言设计一个硬件程序,首先需要建立一个模块,模块中可以包含许多的端口或者参数,比如上层决定的参数、输入输出等,在流水灯设计中主要是输出对 LED 灯的控制信号。为了实现流水灯,需要定义一个 led 的输出来控制灯的亮灭,为了实现时间上的控制你还得有有一个板载时钟信号的输入,为了控制流水灯,最简单的控制输入 reset 也可以有,这样就简单的设定了一个流水灯模块的输入输出。接下来需要对模块中怎么实现 LED 的闪烁进行逻辑功能描述。可以根据设计者的喜好,利用计数器控制灯的闪烁频率。现在只需要按照思路设定好 LED 灯的二进制序列就能完美的控制灯。但需要注意的是 reg 型变量和 wire 型变量并不相同, wire 表示直通变化,输入有变化输出立马发生改变; reg 需要有一定的触发,比如时间的改变,当输入发生变化时,随着触发产生输出才会发生变化,常常用在 always 语句中。当流水灯模块设计完成后,流水灯逻辑的设计也就宣布完成,如果想要在实测前了解程序的问题,就还必须设计一下仿真测试。在测试中例化流水灯模块,然后才能在仿真波形中观察到该流水灯模块输入输出波形的变化。当仿真测试成功之后,就可以将程序编译烧录至开发板中了。至此 Verilog 语言的初步认识已经完毕,设计者已经学会了 Verilog 开发中要掌握的模块,输入输出,时序等设计方法。

介绍完 Verilog 语言的初步学习,来看看 Chisel 语言的初步认识是怎样的。首先 Chisel 并不向 Verilog 一样有很多的研究者,学习 Chisel 必须依靠官方提供的手册来一步步建立自己的开发环境,而 Verilog 开发环境的设置网络上有一大堆教程和许许多多公司开发的软件。解决完环境问题后,才开始正式认识 Chisel。理解 Chisel 语言必须要理解 Scala 语言,如果对 Scala 语言了如指掌,那学习 Chisel 语

言就会非常轻松。因为 Chisel 语言是基于 Scala 语言设计出来的，由于 Scala 语言有能更好的嵌入领域专用语言的能力，还有它那强大的库，在开发时就能捕获错误的能力和强大的函数式编程方式的优点，让 Chisel 在继承 Scala 优点的同时，能在硬件构造上更加方便。

初步学习 Chisel 除了从各种语言通用的“hello world”起手外，也可以按照上面提到的 Verilog 语言学习一样从流水灯的设计入手。首先我们需要导入 Chisel 的库，只有这样软件才能识别我们的 Chisel 语法。Chisel 和 Verilog 一样也是设计 Module 的，但不同的是，Verilog 可以在模块中简易的使用 Input 和 Output 来定义输入输出，在 Chisel 中需要先定义一个 Bundle，在其中来定义输入输出。也可以在模块外设定一个输入输出 Bundle，然后在模块中 new 一个该输入输出的实例，该输入输出 Bundle 除了在本模块中使用外，还可以在其他模块中通过引用来创建相同的实例，这一点 Verilog 并做不到。除了输入输出的设定有差别外，Chisel 还可以去掉时钟的声明，因为 Chisel 的输入输出中隐含包含对 CLK 和 RESET 的声明，如果需要额外的时钟才需要进行设定。设定好输入输出后，就可以开始在模块中对逻辑进行描述。Chisel 语言描述流水灯逻辑与 Verilog 语言并没有区别。按照逻辑编写完成后就可以利用软件进行编译，编译后生成的是 Verilog 语言文件，这样就可以使用该 Verilog 语言文件使用上面提到的仿真文件进行测试，最后上板观察效果。

从两者的初步认识可以看出，如果设计者对 Scala 并不了解的话，上手 Chisel 比上手 Verilog 更加的缓慢。但是当学会了 Scala 面向对象的那套思路之后，使用 Chisel 会比 Verilog 更加方便一些。

5.2 模块设计思路对比

两种语言的早期学习除了存在语言语法上学习的差异，其他的方面并不能一眼的看出优劣。本小节将从本实验所设计的 Chisel 语言处理器和原 Verilog 语言处理器的部分模块设计对比方向，对 Chisel 实现和 Verilog 实现进行探讨。

5.2.1 译码模块对比

译码模块的主要作用在 3.2.1 小节中已经详细的说明，这里再做简单的描述。译码模块主要是对接收到的 32 位二进制指令进行解析，区分该条指令是什么指令，需要进行哪些方面的信号设置，需要进行哪些操作，然后将指令上的信息拆分

出来，按照指令所想要达到的目的，将数据传输给下一阶段执行阶段完成指令所想要达到的目的。

以译码模块心脏的代码，即指令的识别，图 5-1 所示的是 Verilog 语言中对 ADDI 指令的解析，从图中可以看出，解析该 ADDI 指令首先需要分别 32 位代码中的低 6 位 opcode 码，由于 ADDI 为立即数加法指令所以识别为 OP_IMM，可是该 OP_IMM 中除了 ADDI 外还有其他立即数操作指令，所以还必须进行区分对比。使用第二特征码 function 码区分为 ADDI 指令。当判断为 ADDI 指令后，就可以对该指令要进行的操作进行设定，将指令中操作数部分、寄存器地址部分剪切下来传递给输出，设定好执行阶段要进行的运算代号加法 ADD，就完成了 ADDI 指令的信号设定。当指令条数增加的时候，begin 到 end 中对信号设定就会改变，以应对不同指令所需要处理的情况。

```
case (opcode)
  `EXE_OP_IMM:
    case (funct3)
      `EXE_ADDI:
        begin
          aluop_o <= `EXE_ADD_OP;
          alusel_o <= `EXE_RES_ARITHMETIC;
          instvalid <= `InstValid;

          wreg_o <= `WriteEnable;

          reg1_read_o <= `ReadEnable;

          imm <= imm_i_type;
        end
    end
end
```

图 5-1 Verilog 语言对 ADDI 指令的译码

而在 Chisel 的实现中，可以简化该部分。如图 3-3、图 3-4 和图 5-2 中所示，创建一种数组 Array，该数组中包含了所有指令的特征码，当输入一条 32 位指令符合数组中 ADDI 指令的特征的时候，定义的列表就会自动设定为 ADDI 信号设定列表，简化了 Verilog 中多次判断的类型，使用列表函数也能更加直观的对指令信号设定的改变进行观察。当产生新的指令的时候，只需要添加该指令的特征码序列，

然后复制粘贴一种指令解析，然后将其中的信号设定设置为新指令所需要的信号设定即可，能够更加方便快捷的增加删除新指令的设置。

```
ins.ADDI ->
  List(d.EXE_ADD_OP, d.EXE_RES_ARITHMETIC, d.X,
    d.WriteEnable, d.ReadEnable, d.ReadDisable,
    imm_i_type,
    d.ZeroWord,
    d.ZeroWord, d.X, d.NotInDelaySlot, d.X,
    d.X, d.X, d.X, d.X, d.X,
    d.ReadDisable, d.CSRWriteDisable),
```

图 5-2 Chisel 实现中对 ADDI 的设定

5.2.2 各阶段数据传输模块对比

设计 Verilog 语言处理器的时候，按照图 3-1 所示的架构进行设定，往往会设计出存在 IF_ID、ID_EX 等这样阶段之间传递上一阶段信号量到下一阶段信号量的模块，该模块中还可能存在控制模块的输入对模块传递之间的限制。

```
else if(stall[3] == `NoStop)
begin
  mem_wd <= ex_wd;
  mem_wreg <= ex_wreg;
  mem_wdata <= ex_wdata;

  mem_aluop <= ex_aluop;
  mem_mem_addr <= ex_mem_addr;
  mem_reg2 <= ex_reg2;

  mem_csr_reg_we <= ex_csr_reg_we;
  mem_csr_reg_addr <= ex_csr_reg_addr;
  mem_csr_reg_data <= ex_csr_reg_data;

  mem_excepttype <= ex_excepttype;
  mem_is_in_delayslot <= ex_is_in_delayslot;
  mem_current_inst_address <= ex_current_inst_address;
  mem_not_stall <= ex_not_stall;

  mem_mem_phy_addr <= ex_mem_phy_addr;
  mem_data_tlb_r_exception <= ex_data_tlb_r_exception;
  mem_data_tlb_w_exception <= ex_data_tlb_w_exception;

  cnt_o <= 2'b00;
  div_started_o <= 1'b0;
```

图 5-3 Verilog 实现的 EX_MEM 的信号传递

如图 5-3 所示的 Verilog 实现的执行阶段到存储阶段的信号传递，当该阶段转换暂停为否的时候传递数据，然后将所有需要的信号进行赋值传递。纵观该模块下来，首先是很长的初始化设定，对各个下一阶段信号量的初值的赋值，然后就是信号的传递，浪费空间的同时也不容易观察信号的连接错误。

而在 Chisel 中有更加方便快捷的实现这一功能的方法，如图 5-4 所示。图中 `ex_mem_io` 类是在 EX 模块代码中声明的将传递到 MEM 阶段的输出信号量，将要传递的信号统一的设定在其中。第二个类中 `ex_in` 利用 `Flipped` 函数将 `ex_mem_io` 输入输出翻转，即 `output` 变 `input` 实现将 EX 输出的变量转换为输入，之后再在模块中利用条件语句等对传递出来的信号进行控制，就可以完成 Verilog 需要花费一个单独模块所要做的工作。

```
class ex_mem_io extends Bundle{
    output...
    ...
}
...
class mem_io extends Bundle{
    val ex_in = Flipped(new ex_mem_io)
    ...
}

class mem extends Module{
    val io = IO(new mem_io)

    when(stall){
        ...
    }
}
```

图 5-4 Chisel 实现的 EX_MEM 的信号传递

Chisel 除了图 5-4 中展示的信号传递外，还有一种方法能够完成，如图 5-5 所示。从图中可以看出 EX 模块中对 EX 输出到 MEM 的输出类的设定并没有改变，变化在 EX 模块中对 MEM 模块的例化。通过简单的“<>”符号完成连接，该符号相当于图 3-1 中的黑色连接线一样将模块中信号量进行连接。按照这种思路所表现的方式

相当于图 5-6 所示，相当于在 EX 模块中例化 MEM 模块然后将数据传送给 MEM 模块。使用这种方式就如同嵌套函数一样 IF 模块中例化 ID 模块，ID 模块中例化 EX 模块，EX 模块中例化 MEM 模块，MEM 模块中例化 WB 模块，层层嵌套传递参数。Chisel 语言只花费了一些函数和符号就完成了 Verilog 语言所要花费很长代码才能实现的功能。

```
class ex_mem_io extends Bundle{
  output...
  ...
}

class ex extends Module{
  val io = IO(new ex_mem_io)

  val mem = Module(new mem)
  when(stall){
    mem.io.xxx <> io.ex_mem_io.xxx
  }
}
```

图 5-5 Chisel 实现的另一种信号传递方式

```
module ex_mem(...);
  mem mem(
    .xxx(xxx)
  )
endmodule
```

图 5-6 例化效果

5.3 Chisel 实现与 Verilog 实现的优劣

根据前面几章和小节对 Chisel 语言与 Verilog 语言优劣的对比，本小节将总结它们的优劣情况。

首先从代码量的方向入手，从该方向观察两者的区别简单直观。经过统计，统计代码为 Chisel 与 Verilog 主要功能代码的量，未包括顶层文件和阶段转换传递模块代码，Verilog 语言所花费的代码量为 4343 行，Chisel 语言所花费的代码量为 3478 行。从统计结果看出，Chisel 语言的代码使用量为 Verilog 语言的 80% 左右，本实验的所设计的 Chisel 语言实现处理器还可以进一步优化代码，如果完成优化，实现相同功能处理器时 Chisel 语言所花费的代码量大概为 Verilog 语言实现花费代码量的 65% 左右。可以看出使用 Chisel 语言开发处理器能够节省硬件空间，简化代码长度。但其中存在一定的问题，如果 Chisel 语言工程代码中缺少对重要部分的注释代码，由于 Chisel 代码的简洁高概括特性，会让后续的维护者花

费大量的时间去理解代码所要表达的功能效果。在这一点上 Verilog 语言可能能更好的让后续维护者理解项目的功能。

调试测试是确认代码有误错误的手段之一。从调试这个角度来看, Chisel 要比 Verilog 好不少。首先如果代码存在语法错误, 编译器能够识别 Chisel 的错误而不能识别 Verilog 的错误, 因为 Chisel 中特有的语法是基于 Chisel 官方给的函数库和 Scala 的函数库, 出现错误能够立刻发现, 降低了错误出现率; 而 Verilog 出现语法错误的话只能在编译的时候才能识别出来, 提高了错误出现率。编写完代码之后就是编译, Chisel 编译所采用的是灵活中间表示 FIRRTL[1], 所以编译过程中需要符合 FIRRTL 的转换规则。使用 FIRRTL 能够快速发现代码中的逻辑错误, 编译器会将错误类型返回, 告知设计者错误出现区域, 便于设计者修改; 而 Verilog 在编译过程中能够检测出的问题就比 Chisel 的 FIRRTL 少很多, 可能一些能造成程序错误的逻辑错误会被忽视, 要排除这种错误可能会花费很多时间。

在编写完毕, 编译通过, 所设计的程序能够正确的运行后, 所要做的就是根据实际反馈修改代码, 对工程进行维护。就工程维护角度的思路来看, 对了解 Chisel 语言和 Verilog 语言的工程师来说, 可能理清前人的设计思路时, Chisel 语言实现的理解过程要比 Verilog 语言实现的复杂一点。如本文所参考的 Verilog 语言实现的处理器为例, 该处理器中各个模块中所实现的部分能够清晰的理解出来, 这一部分是干什么的, 模块的输入输出、阶段信号传递也清楚的表达出来, 能够花费很少的时间了解设计者的设计思路, 知道要修改某部分需要从哪入手。从本文参考的 Chisel 语言设计处理器进行观察, 刚开始是什么都看不懂的, 模块中各种相互调用, 需要从顶层模块开始画一个调用连线图才能够理清, 各个模块调用了其他模块的哪些功能。理解完这层调用思路后要对项目进行维护的话, 就需要根据调用关系进行维护。简单的来说就是理解了 Chisel 语言实现的硬件程序调用过程后, 对相应部分的维护就会轻松很多, 入手难度大, 后期轻松。

可复用性, 又叫可重用性, 指将一部分代码重复利用。使用复用的好处在于能得到较高的生产效率和之后维护花费的成本降低并提高软件质量。实现复用的方法有很多, 简单的为复制粘贴代码, 每块复制代码在工程各个地方, 框架一样内容不同; 复杂到使用队列、栈、链表等操作。本次实验参考的 Verilog 语言实现处理器

所复用的部分非常少，复用部分也简单的使用为复制粘贴操作，对每一块复用部分还需要进行修改和测试。而 Chisel 语言实现从开始就在复用，即要使用 Chisel 语言编写程序就要调用 Chisel 库，而 Chisel 库中又包含了很多便利的函数能让设计者调用使用，减少了自己设计模块来完成相应功能的时间，在这一点上 Verilog 完全不能比。在本次试验 Chisel 语言实现设计中还利用了队列、链表的等结构来提高了复用性，添加修改复用部分只需修改简单的参数就能完成，提高了维护的效率。

总结来说 Verilog 语言实现和 Chisel 语言实现优劣如下。

Verilog 语言实现。

优点：普及率大，使用人数多，可参考的资源较多；入门难度小，易上手。

缺点：并不能完整的实现开发者的意图，需要花费大量的时间进行设计；代码中隐藏的错误无法及时发现，排除错误花费时间长；代码复用率低，维护成本大；开发大型工程需要代码量大，不易于修改调试。

Chisel 语言实现。

优点：存在 Chisel 库，有各种各样的函数设计，复用率高，简化了设计过程，提高了设计效率；代码精简美观；设计中连线简单，能够完好的重现设计者的思路，实现设计者想要实现的意图；在编写和编译时能够发现一些比较隐藏的逻辑错误，减少了排除错误的时间，提高了效率。

缺点：普及率低，使用人数较少，资源参考大多依靠 Chisel 开发者社区提供；语言上手难度高，需要对 Scala 有一定的了解。

5.4 本章小结

本章主要介绍了从初步认识两种语言的优劣到本实验所设计模块思路的不同来对 Chisel 语言实现的处理器与 Verilog 语言实现的处理器的优劣进行了对比。从结果可以看出两种语言都存在它们的优势，但 Chisel 语言的优势要比 Verilog 语言的大，虽然 Chisel 的使用人数和使用率并不高，但 Chisel 语言在实现硬件功能上比 Verilog 语言更加的方便快捷，熟练之后也更加的浅显易懂。

结 论

随着时代的进步，越来越多的高校和研究者们开始注重于Chisel这门硬件构造语言，国内也开始出现Chisel中文社区为大家学习Chisel提供帮助，Chisel得益于它强大的开发库和面向对象的函数式编程设计，使得它的使用率正在提高。本文侧重于从使用Chisel语言设计实现一个与Verilog语言相同功能的处理器，并通过仿真验证和真实上板对处理器的功能进行了测试，所设计的处理器也实现了预期的功能。为了后续的工作，还对搭载处理器的综合实验系统中关于SDRAM的部分进行了改进，减少了错误出现的频率，提高了稳定性，但由于知识的局限性并未完全修复。在此基础上还对两个相同功能处理器的实现过程进行了对比，从中找出两种语言在硬件设计上的优劣。得出的结论为Chisel语言能够更好的开发硬件程序，它在设计和维护上都优于Verilog语言。具体来说，本文的所研究的工作主要有以下几个方面：

1) 基于Chisel语言与RISC-V架构的处理器

设计实现以Chisel语言为工具，RISC-V为架构的CPU是本文的重点。通过对部分模块设计的介绍，来突出用Chisel语言设计所体现出来的方便快捷的特性，使用其强大的复用库能快速构造想要的功能模块。

2) 综合试验系统的改进

综合实验系统是搭载本文设计处理器的一个功能强大的SoC，集合了各种ROM、SDRAM等外部设备。本文主要对综合试验系统中SDRAM的读写问题进行了探讨，从时序错误和逻辑错误两个方面对错误进行分析和改进。由于受时间的限制和作者知识水平的限制，错误并未完全修复，只是降低了错误的出现频率并实现了一种稳定状态，还需要后续对该系统进行完善。

3) 两种语言优劣对比

本文通过对两种语言实现的相同功能处理器的设计分析，对两种语言实现相同功能的难易度进行了对比。本文从入手难易度、模块设计思路与后期维护三个方面对两种语言进行对比，得出了Chisel在硬件程序设计上更加的方便。得益于Chisel官方提供的函数复用库，它能够更快的构建硬件模块，这一点Verilog并做不到。而且后期维护上，Verilog语言巨大量代码不易于维护，而Chisel语言简洁，连线简单，可复用率大，修改部分小，使后期维护的成本比Verilog低。

本文虽然实现了基于Chisel语言和RISC-V架构的处理器，但是设计还不够成熟，其中存在一些设计问题以及代码冗余。语言的对比只是简单的从三中方面进行对比，还可以用更加专业的量化的进行对比。综合试验系统的问题并未完全解决，还需要后续的完善。

参考文献

- [1] Adam Izraelevitz, Jack Koenig, et al. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. [J]10.1109/ICCAD.2017.8203780
- [2] R. Nikhil, “Bluespec System Verilog: efficient, correct RTL from high level specifications,” in Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. [J] MEMOCODE '04. Proceedings, Jun. 2004, pp. 69–70.
- [3] RISC-V Foundation. About the RISC-V Foundation [EB/OL]. <https://riscv.org/risc-v-foundation>. 2020-05-22
- [4] David Patterson. Andrew Waterman. The RISC-V Reader[M]. America:2017-9.
- [5] 吴迪,谢雪炎,吴贺俊.基于 FPGA 的计算机组成原理实验教学探索[J].计算机教育, 2014(18):30-34.
- [6] 李志平,杨西珊,张俊方.基于 FPGA 的计算机系统实验教学平台的设计与实现[J].实验技术与管理, 2009, 026(009):77-80.
- [7] 周宁宁,程春玲.基于 FPGA 技术的计算机组成原理实验系统[J].现代电子技术,2005(01):23-25.
- [8] 朱威浦. 基于 rust 语言和 RISC-V 平台的操作系统设计与改进[D]. .北京: 北京理工大学, 2019.
- [9] 刘雪松. 基于 RISC-V 的计算机系统综合实验设计[D].北京: 首都师范大学, 2020.
- [10] 范志鹏. 基于 Rust 语言和 RISC-V 平台的计算机系统实验设计和改进[D].北京:北京理工大学, 2020.
- [11] 雷凯翔. OpenRISCV[EB/OL]. <https://github.com/shyoshyo/openriscv>
- [12] Risc-V 中文手册 .RISC-V[EB/OL]. <http://crva.io/documents/RISC-V-Reader-Chinese-v2p1.pdf>
- [13] Chisel 维基百科 .CHISEL[EB/OL]. <https://github.com/freechipsproject/chisel3/wiki>
- [14] Chisel 中文社区. CHISEL[EB/OL]. <https://www.chiselchina.com/>
- [15] RISC-V GNU Compiler Toolchain [CP/OL]. <https://github.com/riscv/risc>

致 谢

本次论文是对我本科学习成果的总结，四年的学习化为果实。但由于疫情原因，本次论文的工作大部分时间是在家完成。在论文完成之余，利用该部分对向我提供帮助的老师同学表示感谢。

最感谢的是我的导师陆慧梅老师，感谢陆老师提供的这个选题机会，让我能够将我的学习成果进行展示。从选题到撰写论文的这几个阶段，陆老师都给我提供了宝贵的意见。在实验过程中，对我不足的部分进行了批评并提供了改进意见，对我的疑惑也提供了我所能理解的帮助。

其次感谢清华大学的老师提供的综合试验系统，让我能够有一个平台去验证实现自己的工作，并且也在相关工作上对我进行了指导。

最后感谢我的家人，在疫情期间在我学习时，给我生活上的依靠，在精神上和物质上支持着我。