论文及学生实验报告精选

1、论文精选

序号	论文
1	刘卫东，张悠慧，向勇，王生原，李山山，面向系统能力培养的计算机专业课程体系建设实践，中国大学教学，2014（8）：p48-52
2	刘亚楠，刘卫东，张小平，李山山，THINPAD 教学计算机实验平台设计，实验技术与管理，2012（11）：p115-118
3	李山山等，计算机组成原理课程实验教学的调查与研究，计算机教育，2012（22）：p127-129
4	郑纬民、张悠慧，面向“计算机使用者”的计算机体系结构教程，计算机教育，2012（11）：p90-93
5	李山山等，面向系统能力的计算机组成原理实验实施，计算机教育，2014（15）:p107-110
6	李山山等，美国计算机硬件系列课程与实验的调研报告，计算机教育，2010(15)：p16-20
7	李山山等，uC/OS 内核在基于 FPGA 的 CPU 上的移植，实验技术与管理，2010（4）：p87-90
8	全成斌，李山山，美国计算机专业课程体系的调研报告，计算机教育，2010（15）：p7-15
9	王生原，董 渊，张素琴，“编译原理”课程实验项目介绍，计算机教育，2009(23)：p12-14
10	Li Shan-shan, etc, the exploration for computer system capacity training in experimental teaching, 2015 IEEE Frontiers in Education Conference, FIE 2015, October 21, 2015 - October 24, 2015, El Paso, USA.
11	Zhang, Yuxiang; Chen, Yu ; Ma, Xiaojian ; Tang, Yuhan ; Niu, Yilin ; Li, Shanshan ; Liu, Weidong, Remote FPGA lab platform for computer system curriculum, ACM International Conference Proceeding Series, v Part F127754, May 12, 2017, Proceedings of the ACM Turing 50th Celebration Conference - China, ACM TUR-C 2017

2、学生实验报告精选

面向系统能力培养的计算机专业课程体系建设实践

刘卫东 张悠慧 向 勇 王生原 李山山

摘 要：系统能力培养是提高计算机专业本科教学质量和水平的一个重要方向。本文提出了系统能力培养的基本概念和内涵，介绍了清华大学计算机科学与技术系在计算机系统能力培养方面所做的工作和实践效果，讨论了课程体系建设和改革的关键内容，给出了一种加强计算机系统能力培养的教学方案。

关键词：系统能力；课程体系；计算机专业

2006年，国务院发布《国家中长期科学和技术发展规划纲要（2006－2020年）》，将“核高基”列入国家重大科技专项。“核高基”是核心电子器件、高端通用芯片及基础软件产品的简称，其被列入国家重大科技专项，一方面体现了它们在国民经济发展中的重大作用，另一方面也折射出我国在这一领域内与世界先进水平存在较大差距。人才培养是科技进步的基础，反思我们在该领域基础知识教学中的不足，改进我们的培养手段和方法，培养出具备创新能力的高素质人才，是值得教育界探讨的话题。

“核高基”是计算机系统的核心和基础。尽管我国在计算机教学领域取得了长足的进步，但主要体现在大规模的应用型人才的养成，没有规模化培养出能够深入理解并掌握计算机系统核心、具备引导行业发展能力的人才。因此，建设新的计算机专业课程体系，着重培养具备计算机系统能力的高素质人才是我们的任务。

计算机系统能力是指能自觉运用系统观，理解计算机系统的整体性、关联性、层次性、动态性和开放性，并用系统化方法，掌握计算机硬软件协同工作及相互作用机制的能力。系统能力包括系统分析能力、系统设计能力和系统验证能力三个方面。系统分析能力就是给定系统结构和输入，分析系统输出的能力；系统设计能力就是给定系统输入和输出，综合出系统结构的能力；系统验证能力就是给定系统结构，验证系统结构与功能符合的能力。三个方面相辅相成，共同构成计算机专业本科毕业生的基本能力和专业素养。

刘卫东，清华大学计算机科学与技术系教授。

一、课程体系现状分析

计算机专业课程体系中，设置了大量与计算机系统相关的课程，如数字逻辑电路、汇编语言程序设计、计算机组成原理、计算机系统结构、编译原理、操作系统、微计算机接口、嵌入式系统等。这些课程的总体目标，是建立起完整的计算机软硬件系统的知识结构，在课程体系中占有重要的地位。

然而，大量的系统类课程的学习，并没有给学生带来计算机系统能力的全面提升。同学们普遍反映，对于计算机系统还处于“只见树木，不见森林”的状态，对于计算机各个子系统有比较深入的了解，但对各子系统之间的相互作用的关系则了解不深，也不清楚各子系统间的衔接机制。

深入剖析这一现象，我们认为，现有的计算机专业课程教学中，普遍存在以下几个方面的问题。

（1）教学内容上，各门课程独立规划、独立教学，造成了知识体系中知识点冗余和衔接关系脱节。一方面，每门课程强调自身知识体系的完整性和系统性，造成一些知识重复讲述；另一方

面，各课程之间知识点缺乏前后衔接和有效整合，难以形成完整的计算机系统知识体系。

（2）教学方法上，各课程采取分析式教学方法较多，突出系统原理的讲解，而限于条件的不足，缺乏对基本完整计算机系统较为全面的说明。造成的结果是学生掌握了基本概念，但难以转化为设计完整计算机系统的基本能力。

（3）实验手段上，基本侧重于对原理的简单验证，而缺乏对复杂系统的综合设计实践。虽然小规模实验可以达到让学生基本理解掌握系统运行原理和初步具备系统开发能力的目的，但由于缺乏足够的工程工作量，使得复杂系统中存在的较为深刻的问题难以暴露。因而，学生虽然经过了训练，但却因训练强度不足，不能对系统有较为深刻的认识和完成具有工程规模的系统级开发，甚至是对计算机系统有“盲人摸象”的感觉。

计算机系统本身具有整体性、关联性、层次性、动态性和开放性等特点，但由于其复杂性，如果课程体系中各课程之间缺乏有效沟通和相互协作机制，就会造成学生系统能力培养和训练上的不足。因此，对课程体系改革势在必行。

二、面向系统能力培养的课程体系建设

为了弥补计算机系统专业课程教学中存在的这一不足，加强专业课程中的系统能力培养，结合计算机类专业教学指导委员会计算机系统能力培养的相关要求，有必要对计算机专业课程体系进行改革。其目标确定为：将计算机系统类课程在教学理念上统一贯彻“注重系统、强调实验、培养能力”三个方面；将教学内容进行统一规划，为同学们构建完整系统的知识体系和知识结构；并从工程教育的角度，探索一体化设计课程体系，以及原理性与工程性结合、分析式与综合式互动的教学方法；实验手段上注重教学载体和实验平台的统一，最终实现学生能够基于一个指令集系统，自主设计一台功能计算机、一个操作系统核心、一个编译系统的教学目标，切实使学生全面掌握计算机系统的知识点，并能融会贯通，培养计算机系统的设计能力。

为达成改革目标，我们在教学内容规划、实验体系设计、实验平台开发等方面开展了一系列的工作。

1. 统一规划教学内容和教学方法

根据课程体系改革目标，我们首先选择计算机系统类课程中的数字逻辑电路、汇编语言程序设计、计算机组成原理、操作系统、编译原理 5 门核心课程，组建计算机系统类课程群。以 ACM 发布的计算机专业课程知识体系为蓝本，对照我系课程体系中的相关教学内容，查找各课程知识点和教学内容的不足，各课程进行补充和完善，注意各课程教学内容间的衔接。具体调整内容如下表所示。

课组内容调整与优化表

课程名称	主要调整/优化内容
数字逻辑电路	使用可编程芯片的入门级以上到部件级以下的设计实验（包括多路选择器、基本组合电路/时序电路以及综合实验），以及相应的 EDA 工具训练，为处理器实验奠定部件设计与工具应用基础
汇编语言程序设计	（1）强化了汇编语言作为处理器软硬件接口、作为计算机系统结构规格的内容讲解 （2）增加了 MIPS32 指令集部分，以及相应的中断/异常处理、虚存管理等内容（包括其在汇编层面的表示） （3）增加了典型 C 代码在汇编层面的表示以及反汇编等内容
计算机组成原理	（1）MIPS32 指令系统设计及分析 （2）简单计算机系统设计和实现，该计算机系统的 CPU 至少能够支持 MIPS 指令集的一个子集，具体实现可以是多周期或者是指令流水方式，CPU 必须能够支持中断，包括软中断和硬中断，完成中断请求、中断响应和中断服务及返回的全过程 （3）虚拟存储管理中 TLB 的作用及组成

操作系统	（1）以 X86 和 MIPS32 指令集/处理器作为操作系统的目标平台 （2）强调操作系统原理与实验的结合，要求在实验中完成一个简化但必须是在真实硬件上能工作的小操作系统，并实现操作系统的核心算法，从而加深对原理的理解 （3）强调各个实验的核心算法形成一个有机整体，后面的实验会用到前面实验的代码，并最终形成一个完整的小操作系统
编译原理	（1）以 MIPS32 指令集/处理器作为编译器的目标平台 （2）强调编译技术与计算机系统整机的关联，从编译程序能发挥的作用及它在整个计算机系统中的位置等角度，将计算机系统整机概念贯穿起来，充分理解编译程序/系统与其他系统类程序或工具之间的联系 （3）强调编译技术与计算机系统使用的关联，充分理解通过编译优化可以使计算机系统发挥更有效的作用（与操作系统的角度不同），也要理解编译技术在这方面的局限性

在理顺教学内容的基础上，各课程以完成基本计算机系统设计和实现为教学目标，改进教学方法。在加强原理性知识讲解的同时，强化工程化实现方法的训练，力求学生在系统原理和工程实现方法两方面均有收获。

2. 统一规划课程实验体系

计算机系统能力培养中，实践占有很大的比重，是学生运用所学的原理知识，解决实际计算机系统设计问题的过程，更是检验教学效果的重要手段。然而，计算机系统是一个复杂的巨系统，要让学生在有限的时间下完成教学和实践内容，需要我们精心设计教学实验体系，围绕教学改革目标设置各课程的阶段子目标和相应的实验内容，完成模块设计和实现后，再通过综合实验来最终集成，形成一个完整的计算机系统设计和实现。

根据这一思路，我们在各相关课程中调整原有的实验体系和实验内容。将课程实验作为最终综合实验的模块或基础，既能巩固课程中学习的原理性知识，又能作为整体综合实验的模块使用，实现实验体系的递进化。

我们以操作系统的实验体系为例介绍其设计思想。在清华大学计算机科学与技术系操作系统课程中，安排了 8 个教学实验，通过精心安排和组织，8 个实验由基础到全面，最终构成了一个基本完整的教学操作系统。

（1）实验 0（操作系统实验环境和工具）的目的在于了解和熟悉操作系统实验的编译方法和流程、基于硬件模拟器的操作系统内核调试方法以及 OS 启动前方的基本功能，并复习了 C 语言和汇编语言，为基于 ucore 的后续实验打下实验方法上的基础。

（2）实验 1（系统软件启动过程）的目的是实现 bootloader 加载和运行操作系统软件的工作过程，从而理解启动 bootloader 的过程、bootloader 的文件组成、ucore OS 的启动过程、中断处理机制、通过串口/并口/CGA 输出字符的方法。让同学能够运用多种工具来开始操作系统功能。

（3）实验 2（物理内存管理）与实验 3（虚拟内存管理）的目的是了解系统如何管理内存，深入理解软硬件的分页模式，页表的建立和使用方法，缺页中断的处理等基本处理方法与管理方法。需要实现缺页异常处理和页替换算法，支持虚拟存储管理，为应用提供大于物理内存的虚拟地址空间。二者在实验内容上形成循序渐进的关系。

（4）实验 4（内核线程管理）、实验 5（用户进程管理）、实验 6（调度器）、实验 7（同步互斥）帮助学生了解操作系统中内核线程创建和执行的过程、了解上下文切换是如何具体实现的，并进一步了解操作系统中用户进程的实现以及操作系统的调度管理机制，实现基于管程的条件变量机制。四者在实验内容上形成循序渐进的关系。

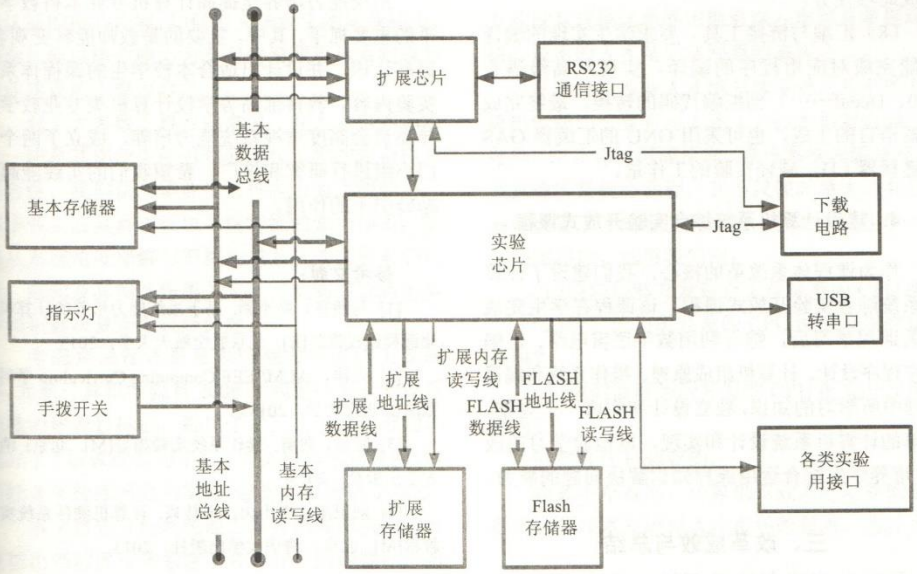
（5）实验 8（文件系统）要求学生了解基本的文件系统调用的实现方法，掌握基于索引节点的文件系统和虚拟文件系统的具体实现，能够完

成读文件操作和基于文件系统的执行程序机制，扩展基本文件系统，以支持目录、软链接等，扩展缓存机制，提高缓存的效率和性能。

3．建设统一的教学实验支撑平台

为配合统一规划后的教学内容和相关教学实验教学，并为综合实验打好基础，统一各课程教学实验支撑平台十分重要。教学实验支撑平台应能完成各课程规划和调整后的教学实验，至少应包括硬件系统平台及开发调试工具、指令级模拟器、汇编器、编译器、教学操作系统模拟器及调试工具等，还应包括上位机和实验开发板的通信程序等。所有平台应支持一个简单、规范、基本完整的统一的指令系统。为此，我们开发完成了支持 50 条左右 MIPS32 指令的硬件计算机系统 THINPAD，作为硬件开发平台。同时，为它配套开发了指令系统模拟器、汇编器、编译器、数据通信程序、终端程序等一系列软件系统和调试工具，并且完成了教学操作系统 Ucore 的移植，使其能在该硬件平台上运行。另外，开发了编译系统，使编译生成的代码能在操作系统调用下运行，基本完成了教学实验支撑平台的建设，并在实际教学中得到了检验。

（1）硬件系统平台组成。计算机系统设计需要给学生提供基本的硬件平台。考虑到硬件设计的主流技术和教学基本要求，系统设计主要采用可编程硬件实现技术。因此，整个硬件平台以大规模可编程逻辑器件为中心，通过总线连接 SRAM 存储器和 Flash 存储器，再配合以外围各种接口。下图是硬件平台的基本组成。



计算机系统综合设计硬件平台组成图

（2）MIPS 指令系统模拟器。指令系统模拟器可帮助学生完成简单的机器语言程序的软件模拟和调试，避免直接在不可靠的硬件平台上调试带来的困难。指令系统模拟器可采用通用的 MIPS 指令系统模拟器，也可以有针对性地开发。

（3）MIPS 指令系统汇编器。汇编器可完成汇编语言到机器语言的转换，并帮助学生理解程序。给学生提供一个方便的、有针对性的汇编器，对完成实验是有帮助的。

（4）仿真终端程序。完整的计算机系统需要配置一定的外部设备。最常用、最简单的外部设备就是计算机终端。因此，配置一个具备一定功

能的仿真终端程序，可以帮助学生提高调试效率，降低硬件实现的难度。

（5）数据通信程序。硬件调试所需要的程序和数据应装入到硬件平台的非电易失性存储器中，需要为实验提供一个通信程序完成这一功能。

（6）监控程序。在完成能运行操作系统的计算机硬件系统实现之前，可以采取运行一个相对简单但具备一定功能的监控程序作为检验初步硬件系统实现的手段，也可以作为计算机组成课程的阶段性成果。

（7）GCC 编译器。由于建议的操作系统由 C 语言编写，故需要有一个能将操作系统编译到指定的指令系统的编译器。由学生自行设计和实现能编译操作系统的编译器规模太大，难度也很高，故建议使用 GCC 编译器并对其进行一些调整后完成此项任务。

（8）汇编与链接工具。要求学生实现的编译器能完成对应用程序的编译，实现从高级语言（C0、Decaf……）到汇编代码的转换，最终完成机器语言的生成，也可采用 GNU 的汇编器 GAS 和链接器 LD，减轻实验的工作量。

4. 建设计算机系统综合实验开放式课程

作为课程体系改革的核心，我们建设了计算机系统综合实验开放式课程。该课程在学生完成相关课程学习后，综合利用数字逻辑电路、汇编语言程序设计、计算机组成原理、操作系统和编译原理中所学习的知识，独立设计和完成一个完整、简单的计算机系统设计和实现，以检验学习的成果，培养学生综合运用课程知识解决问题的能力。

三、改革成效与总结

课程体系各项改革措施已在清华大学计算机科学与技术系本科教学中逐步实施。教学实验平台已基本开发完成，并已在课程实验中使用；各课程的教学内容、实验体系基本调整到位；计算机系统综合实验课程虽尚未全面开设，但已以兴趣小组的方式在小范围进行了三轮实验，已经具备全面开设课程的基础，即将在本学年夏季学期面向全系学生开设。

同学们普遍反映，改革后各门课程之间教学内容衔接良好，教学实验体系完整，对理解教学内容有很好的帮助。各课程统一了教学实验支撑平台，也使同学实验更为方便，减少了对熟悉实验平台的负担。参加过计算机系统综合实验的同学则觉得收获更大，更加深刻领会了计算机系统运行的原理和内部实现的机制，真正设计实现了人生的第一台计算机，给了他们一个充分展示自己能力的舞台。

系统能力培养是提高计算机专业本科教学水平的重要抓手，其中，重要的是教师能转变观念，提高认识，并设计出适合本校学生的课程体系和实验内容。教育部高等学校计算机类专业教学指导委员会高度重视系统能力培养，成立了两个专门小组进行研究和推广，希望我们的实践能起到抛砖引玉的作用。

参考文献：

[1] 马殿富，高小鹏. 基于系统能力培养的计算机专业课程建设报告[R]. 北京航空航天大学，2013.

[2] 张铭，ACM/IEEE Computing Curriculum 学科规范[R]. 北京大学，2013.

[3] 陈渝，向勇. 操作系统实验指导[M]. 北京：清华大学出版社，2013.

[4] 刘卫东，李山山，宋佳兴. 计算机硬件系统实验教程[M]. 北京：清华大学出版社，2013.

[责任编辑：余大品]

THINPAD 教学计算机实验平台设计

刘亚楠，刘卫东，张小平，李山山

（清华大学 计算机科学与技术系，北京 10084）

摘　要：计算机硬件课程在计算机专业课程体系中占有重要的位置，但现有的实验平台无法满足各高校的实验教学需求。该文通过对国内计算机硬件实验平台现状的分析，针对存在的问题，采用主流的硬件设计技术，实现了一个功能更全面、运行更稳定、性能更强大、使用更方便、成本更低廉的实验平台。该平台通用性强，灵活性高，接口丰富，具有良好的运行效率和交互性，支持多门硬件课程的实验需求，必将为提高教学质量起到积极的作用。

关键词：实验平台设计；计算机硬件实验；EDA；FPGA；THINPAD

中图分类号：G434　　文献标志码：A　　文章编号：1002-4956(2012)11-0115-04

Design of THINPAD experimental platform of teaching computer

Liu Yanan, Liu Weidong, Zhang Xiaoping, Li Shanshan

(Department of Computer Science & Technology, Tsinghua University, Beijing 10084, China)

Abstract: Computer hardware courses in computer course system occupy an important position, but the existing experimental platforms cannot satisfy the requirement of experimental teaching of all colleges and universities. Through the analysis of the current status about domestic computer hardware experimental platforms, this article, in view of the existing problems, describes a way using mainstream hardware design techniques to achieve a platform, which has strong versatility, high flexibility, abundant interface, with a good operational efficiency and interactivity, supporting the experimental needs of a number of hardware courses. This platform will play an active role to improve the quality of teaching.

Key words: design of experimental platform; computer hardware experiments; EDA; FPGA; THINPAD

目前，国内计算机硬件实验平台主要分为 2 种：一种是利用计算机某些组成部分的分离元件构成的实验平台，来完成各种简单模型的验证性实验[1]；另一种是配合相应的软件工具和硬件载体，以 EDA 技术为基础开展实验的平台，这种平台在国内已逐渐成为主流。现阶段高校使用的实验平台基本上是各实验室自行研制开发或通过高价购买实验开发板为主，这 2 种情况都有不足之处，亟待改进。自行研制开发的实验平台总的来说成本可控，技术配套全面，但是由于器材更新换代较快，突显出这类实验平台的不足：一是器件普遍性能不高，容量小，老化损坏严重，维护比较困难；二是平台集成度低，空间利用率差；三是扩展能力和兼容性差，设计有缺陷，运行稳定性不高；四是配套实验教学内容陈旧，影响教学效果[2]。有些院校由于技术力量单薄，会选择购买开发板，这样既可以省去开发时间，又可以得到技术支持。但是购买开发板也存在问题：一是设备设计复杂，成本过高；二是管理上技术受限，配套不齐全，设备调试困难大；三是缺乏开发技术，平台兼容性差，不易扩展[3-4]。

THINPAD(TsingHua mINi PAD)教学计算机系统是我们为了弥补这些不足而设计的一个新型计算机硬件实验平台，它采用 Xilinx 公司的 FPGA 芯片作为核心硬件，配置 2 路独立的总线系统，连接 SRAM 静态存储器作为内存，并配备多种外部接口，如串口、USB、FLASH、VGA、PS2 等，再配合微型开关、指示灯、数码管等调试装置，构成了一个完整的计算机硬件实验平台。同时，我们还为它设计了配套丰富的软件系统，如模拟器、编译器、汇编器及监控程序等，满足计算机系统实验的需要。在此基础上，设计和实现了验证性、设计性和综合性 3 种层次的实验实例。整个实

收稿日期：2012-04-23

作者简介：刘亚楠(1983—)，男，吉林梅河口，在读硕士研究生，研究方向：网络安全

E-mail: liuyanan09@qq.com

通信作者：刘卫东(1968—)，男，江西分宜，博士，副教授，研究方向：计算机网络及应用、网络服务质量、工作流等。

验平台系统硬件、软件配置完整、实验内容全面、实验技术先进,覆盖了计算机组成原理、嵌入式系统、数字逻辑、计算机接口等多门硬件课程的知识点,并能为操作系统、编译原理等软件课程提供硬件支持。

1 实验平台简介

1.1 实验平台整体结构

THINPAD 硬件实验平台以 Xilinx 公司的 SPARTAN-Ⅲ系列的 XC3S1200E 芯片为核心,配合其他外围芯片和器件,通过编写不同的实验逻辑,实现完成多种不同功能的实验系统。实验平台结构如图 1 所示。

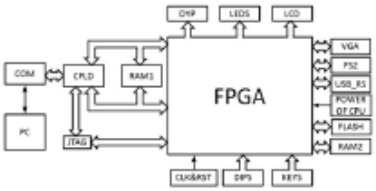


图 1 THINPAD 硬件实验平台结构图

THINPAD 硬件实验平台的平面实物图如图 2 所示。其中:

(1) FPGA 是整个平台的核心,可由实验者写入代码、进行实验。平台使用 Xilinx 公司的 SPARTAN-Ⅲ系列的 XC3S1200E 芯片,1 200K 系统门,2 168 个 CLB,19 512 个逻辑单元,采用 FG320 的封装形式[5];

(2) CPLD 通过其实现的 UART,用于串口通信控制。本平台上使用的 CPLD 是由 Xilinx 生产的 XC95144XL 芯片,此款芯片使用 3.3 V 电压,适用于高性能、低电压的通信和计算机系统,它由 8 个 54 V18 功能模块组成,可提供 3 200 个 5 ns 延迟可用门[6];

(3) RAM1 和 RAM2 是 2 个独立的存储芯片,二者具有不同的数据总线和地址总线,可以用来分别存储实验测试程序以及数据,也可以统一编址存储测试程序和数据;

(4) 手拨开关由 2 组开关组成,共 16 位,微动开关 4 个,用于向 FPGA 接入数据;

(5) 指示灯用于观察总线数据和 RAM1 的控制信号;

(6) LEDS 用于信号观察和检测,实验过程中可以将需要检测的信号进入 LEDS 上,共 16 位;

(7) RS232 通信接口用于通信;

(8) USB 转串口将实验芯片的串行口转换为 USB 接口;

(9) Flash 存储器存储实验芯片所需的程序和数据;

(10) 各类实验用接口,如 VGA、PS2 等,用于辅助进行各种硬件实验;

(11) 晶振为平台提供时钟频率,实验者可以根据实验具体要求进行分频使用,共 2 个。

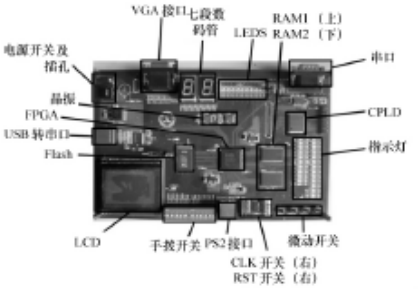


图 2 THINPAD 硬件实验平台的平面实物图

1.2 指令系统设计与分析

实验平台使用的是类似于 MIPS16e 标准的 THCO MIPS 指令系统。MIPS 是比较流行的 RISC 处理器之一,全称为"无内部互锁流水级的微处理器"(microprocessor without interlocked pipeline stages),80 年代初起源于斯坦福大学,其使用的 MIPS 指令系统为 32 位指令,格式统一,易于流水[7]。相关技术人员为了扩展 MIPS 指令系统的功能又制定了 MIPS16e 标准,它是 MIPS 指令面向应用的扩展,可降低应用程序所需的存储容量,因此嵌入式设计人员能够降低成本,提供更高的代码密度。MIPS16e 标准的指令为 16 bit,并兼容 MIPS32 和 MIPS64 指令集[8]。

THCO MIPS 指令系统是为 THINPAD 硬件实验平台设计的指令系统,它遵照 MIPS16e 指令系统的格式,选择了其中的 43 条指令组成,基本覆盖了除一些特殊指令(如乘除法)外的全部指令。另外,为降低系统设计难度,指令系统只支持对存储器的字进行访问,去除了按字节、半字访问等指令。THCOMIPS 指令系统功能齐全,寻址方式简单,实现难度较低,十分适合作为教学计算机指令系统使用。整个指令系统的指令按操作类型共分为 4 种:R 型、I 型、B 型和 J 型。R 型指令从寄存器堆中读取源操作数,结果写回寄存器堆;I 型指令使用一个(4 bit 或 5 bit 或 8 bit 或 11 bit)立即数作为一个源操作数;B 型指令使用一个立即数作为跳转的目标地址;J 型指令使用寄存器的值作为跳转的目标地址[9]。

2 实验平台设计与实现

2.1 模拟器

模拟器是在实验过程中可以通过模拟本实验平台的硬件标准，支持前面介绍的 THCO MIPS 指令系统，用户通过模拟器熟悉指令在实验平台上的运行过程，同时可以借助模拟器测试开发汇编器和监控程序。模拟器实现了汇编、反汇编、查看和设置寄存器值、单步或连续运行以及对断点的操作等功能。模拟器的结构简单，代码公开，方便学生研究代码，不断对其功能进行完善和扩充。模拟器运行的实例见图 3。

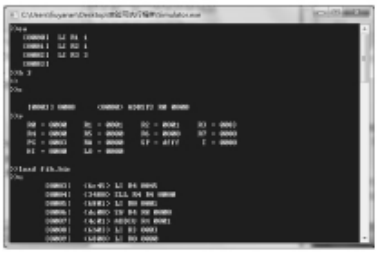


图 3 模拟器运行界面图

2.2 汇编器

汇编器支持所有的指令，使用 3 遍扫描的方式，通过 lex 词法分析生成器生成词法分析程序，语法分析生成器 yacc 生成语法分析。在编译词法分析器和语法分析器时使用的是 linux 下 g++ 编程，在 Windows 模式下用 MinGW 支持。

2.3 监控程序和仿真终端

使用汇编语言写的监控程序，其主要功能是检测 CPU 实现的正确性，有效地支持操作系统的运行，支持汇编、反汇编、连续运行、查看内存、查看寄存器、中断功能等命令。仿真终端是实现与 THINPAD 硬件实验平台及模拟器进行通信的工具，可以方便地验证监控程序的正确性。通过仿真终端运行监控程序的实例见图 4。

2.4 Flash 与 RAM 读写控制端

为了能够实现对 FlashH 和 RAM 内部数据的读写，方便监控程序的存储，实现了 Flash 与 RAM 的读写控制端。可以利用控制端载入二进制或专用格式文件到软件中的数据区，进而将软件中的数据写入到 Flash 或 RAM 中，涉及的单元会被自动擦除，读取数据时将读取 Flash 或 RAM 中的数据到软件的数据区，并进一步导出数据文件。

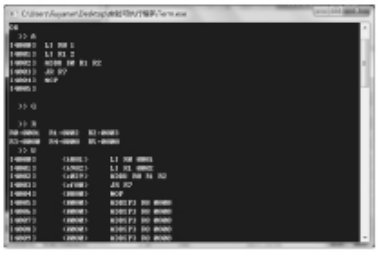


图 4 监控程序运行界面图

3 实验设计

实验内容主要涉及计算机组成原理，包括三部分：验证性实验、设计性实验和综合性实验[10]。

3.1 验证性实验

验证性实验主要包括：指令系统实验、监控程序扩展实验、数据通路实验、存储器实验、串行接口实验、VGA 接口实验、键盘实验、FLASH 实验、控制器实验等[11]，主要是通过此类实验使得学生能够尽快地掌握实验工具、硬件语言和熟悉指令系统及实验平台，适合在教学中结合理论内容适时安排，增进课程的趣味性，使得原本枯燥乏味的书本理论更易接受。

3.2 设计性实验

设计性实验主要包括：单周期 CPU 实验、多周期 CPU 实验、指令流水 CPU 实验、Cache 实验等，此类实验主要是在验证性实验的基础上，让学生自己动手设计、实现具备一定功能的器件，过程中需要综合运用基础理论知识和技巧，具有一定的挑战性，能够巩固和提高学生对理论知识的掌握程度，充分理解 CPU 的运行方式，牢固建立计算机整机概念[12]。实验内容适合中、高水平的院校进行教学应用，这样既不会因为实验难度过大使学生感到有压力，也不会因为难度低、没有挑战性，失去学习的兴趣。

3.3 综合性实验

综合性实验主要包括：支持指令流水的计算机系统设计与实现，实现多道程序、高速缓冲存储器、双机通信、中断、层次储存管理系统等扩展功能。此类实验在设计支持指令流水的计算机系统的基础上系统地扩展了实现功能，要运用多方面的学科知识，有一定的难度，适合水平较高的院校选择。

3.4 部分实验介绍

3.4.1 数据通路实验

数据通路实验即 ALU 设计实验，其主要是根据数据通路，用 VHDL 语言实现一个简单的 ALU，并在

教学实验板上验证实现的 ALU 的功能。本实验通过设计一个简单的 ALU 帮助学生更好地理解数据通路和 ALU 的工作原理，并通过该实验使学生熟悉 VHDL 硬件描述语言，为接下来的实验打好基础；算术逻辑部件的主要功能是对二进制数据进行定点算术运算、逻辑运算和各种移位操作，算术运算包括定点加减乘除运算，逻辑运算主要有逻辑与、逻辑或、逻辑异或和逻辑非操作。ALU 通常有 2 个数据输入端 A 和 B，一个数据输出端 Y 以及标志位等。本实验通过实现一个状态机，并根据状态机状态的变化实现不同的运算，将结果和标志位呈现出来。实验中的 ALU 可以实现基本的算术运算、逻辑运算、移位运算等，要求 ALU 为 16 bit，算术元算时数据用补码表示。功能如表 1 所示。

表 1 ALU 实现的基本运算

操作码	功能	描述
ADD	A+B	加法
SUB	A−B	减法
AND	A and B	与
OR	A or B	或
XOR	A xor B	与或
NOT	not A	取非
SLL	A sll B	逻辑左移 B 位
SLA	A sla B	算术左移 B 位
ROL	A rol B	循环左移 B 位

主要实验步骤：(1)在 Xilinx 中用 VHDL 编写一个 ALU，可以用 2 个状态机分别表示输入状态和运算操作状态。要求是先后输入操作码和 2 个操作数，然后得出结果和标志位。为了便于实现，我们可以均输入 2 个操作数而不关心单操作数的情况，因为我们只对有效操作数操作。(2)在 THINPAD 教学板上演示时，reset 和时钟均用手动开关或按钮，便于演示。操作码和操作数在开关 SW0—SW15 上输入。为便于观察和调试，每次 ALU 得到操作码和操作数，最好可以在 LED 上显示一下。最后的运算结果在 L0—L15 上显示，标志位可自行选择显示方法。展示的方法是多样的，大家可以自行设计展示的方式。

3.4.2 控制器实验

控制器实验即 THCO MIPS 指令控制器实验。实验主要是编写 VHDL 程序，实现具有 7 条 MIPS 指令功能的控制器，7 条指令分别为：ADDU、SUBU、BEQZ、JR、XOR、LW、SW。本实验通过用 VHDL 硬件描述语言设计指令控制器，帮助学生学习和掌握 THCO MIPS 指令系统的指令功能和指令格式，并掌握硬件描述语言的书写规范。实验在 Window 环境下，使用 Xilinx ISE 12.3 进行程序编写，通过数据线烧入 THINPAD 教学实验板，并需要将各个指令周期的控制信号(共 13 个控制信号)作为输出进行展示。

主要实验步骤：(1)定义输入信号：指令、时钟、重置，以及进行展示的必要控制信号；(2)定义 13 个需输出的控制信号；(3)定义状态机，并在执行过程中控制器的不同周期，对控制信号赋值；(4)设置自定义的展示方式。

4 结束语

计算机硬件课程教学需要有高水平、高质量的教学计算机系统的支持。THINPAD 教学计算机系统设计上充分考虑到现有平台的不足，采用主流的硬件设计技术，可由学生完成 1 台硬软件配置齐全的计算机系统，满足教学的要求。它配套的实验内容丰富，适应不同难度需求的课程，紧贴理论教学，其开放的实验环境，利于实验内容更新，不但能够使学生掌握基本的实验原理，而且能够培养学生的动手能力和创造力，加深对工程设计的理解。该平台通用性强，灵活性高，接口丰富、具有良好的运行效率和交互性，支持多门硬件课程的实验需求，必将为提高教学质量起到积极的作用。

参考文献(References)

[1] 方怡晴. 基于 FPGA 技术的"计算机组成原理"课程的实验教学[J]. 实验室研究与探索，2004，23(5)：14-18.

[2] 王诚，宋佳兴. 教学计算机系统的设计和实现[J]. 计算机工程与应用，2005(12)：213-216.

[3] 李山山，全成斌. 计算机组成原理课程实验教学的调查与研究[J]. 计算机教育，2010(22)：127-129，137.

[4] 白中英. 计算机组织与体系结构[M]. 北京：清华大学出版社，2008.

[5] Xilinx. Spartan-3E FPGA Family：Complete Data Sheet Product Specification[EB/OL]. [2012-01-22]. http://www.xilinx.com/support/documentation/data-sheets/ds312.pdf.

[6] Xilinx. XC95144XL：High Performance CPLD Preliminary Product Specification. [EB/OL]. [2012-01-22]. http://html.datasheet5.com/a/vd/da/avddadsk/preview.html.

[7] John L. Hennessy，David A. Patterson Computer Architecture：a quantitative approach[M]. 北京：China Machine Press，2002.

[8] 百度百科. MIPS[EB/OL]. (2012-05-15). [2012-05-16]. http://baike.baidu.com/view/120375.htm.

[9] 王诚，刘卫东，宋佳兴. 计算机组成与设计[M]. 3 版. 北京：清华大学出版社，2008.

[10] 王诚，刘卫东，宋佳兴. 计算机组成与设计实验指导[M]. 3 版. 北京：清华大学出版社，2008.

[11] 袁春风. 计算机组成与系统结构[M]. 北京：清华大学出版社，2010.

[12] 陈勇. 计算机组成原理实验平台优化[D]. 北京：清华大学，2009.

文章编号：1672-5913(2010)22-0127-03　　中图分类号：G642　　文献标识码：A

计算机组成原理课程实验教学的调查与研究

李山山，全成斌
(清华大学 计算机实验教学中心，北京 100084)

摘　要：针对国内计算机组成原理课程实验教学落后的实际情况，在分析多个高校实验教学的基础上，结合清华大学多年实验教学经验，从课程设置、实验设置和实验管理及评价几个方面，阐述课程实验教学现状，指出使用可编程芯片进行计算机原型系统设计实验已经成为大势所趋。

关键词：计算机组成原理；实验教学；可编程器件；实验评价

计算机组成原理课程是计算机学科的一门专业基础课，主要内容包括计算机构成及其各个部分如何协调工作[1]。在整个计算机专业课程体系中，计算机组成原理是起着承上启下的作用[2-3]，它以数字逻辑课程为基础，而自身又是计算机系统结构、编译原理、操作系统等课程的基础。同时计算机组成原理又是一门与实践结合很紧密的课程，课程实验一直是教学中的一个重点，各高校也很重视，在实验上投入了大量的精力。

2009年11月在南京召开了“计算机组成与结构课程群”的实验教学研讨会，会上讨论了国内实验教学的进展和不足，本文立足于此次会议，结合各校的实验教学环节，以计算机组成原理课程为例，对实验教学进行研究。

1　课程设置

目前，国内大多数高校都将计算机组成原理作为第一门专业课程安排在数字逻辑课之后，主要内容包括：计算机系统的基本概念、指令系统、处理器组成(运算器、控制器等)、存储系统、输入输出系统、流水线技术等[1,4]。组成原理一般会安排在大二下学期甚至大三上学期，这样就不可避免地造成与其他专业课程同时开课，使得学生在没有掌握计算机组成之前就开始更高层次的专业课学习，这样无法体现计算机组成原理的专业基础课作用。

为了解决这些问题，一些学校在课程设置上学习了国外大学的做法，开设了一门计算机入门性质的课程，如清华大学和中国科技大学开设了计算机系统导论课程，课程系统地介绍了计算机专业的一些入门知识：最底层的器件→逻辑门电路→微结构→指令集结构→程序→算法→问题域。这样，学生对计算机有了概括性的基础知识，这样就可以避免课程安排的问题了，同时，教师在计算机组成原理课上就可以更加深入地介绍计算机的组成和工作原理了。

各校一般都在计算机组成原理课程中安排试验[4-5]，课程的总课时中有专门的实验课时，让学生在学习理论课的同时完成实验，这样做的好处是让学生能够将理论学习和实验操作同时进行，加深对知识的理解，但是由于进度安排的问题，综合性的大实验(如处理器设计)只能被安排在学期后段，学生需要短时间内投入大量精力才能完成。对于一些无法单独在组成原理课程中实现的更大规模的课程设计型实验，需要学生掌握系统结构、编译原理、操作系统等课程的知识，也需要更多的实验课时，为此，一些学校开设了专门的计算机综合实践课程，如东南大学的计算机系统综合课程设计、中国科技大学的计算机系统原型设计等，这些课程综合了计算机学科多方面的知识，以计算机组成原理为实验基础，进一步拓展了实验的领域。

作者简介：李山山(1979-)，男，工程师，硕士，研究方向为计算机系统结构、计算机实验系统；全成斌(1972-)，男，高级工程师，博士，研究方向为计算机系统结构、计算机实验系统。

中国科技大学华夏班在课程设置上参考了国外大学的一些方案，面向计算机系统结构学科发展前沿，强调前瞻性、先进性和实践性，探索出了计算机组成课程群课程设置(见表1)的新方向。

从课程设置中我们可以看出实验在总课时和总学分中所占的比重很大，几乎占到了二分之一，并且计算机系统原型设计是一门实验课程，分为A和B两个部分，A为CPU设计，B为系统软件设计，二者结合起来就是一个完整的计算机系统原型。清华大学在课程设置上也与其类似，只是没有专门的实验课程，内容也简化很多，主要着重于计算机组成原理的相关内容。

表1 华夏班计算机组成课程群课程设置

课程类型	课程名称	学时(理论/实验)	学分(理论/实验)	学期
导论课	计算机系统导论	60/60	3/2	大一(下)
基础课	数字系统设计	40/90	2/3	大二(下)
基础课	计算机组成原理与设计	60/60	3/2	大三(上)
基础课	计算机体系结构	60/40	3/1	大四(上)
综合课	计算机系统原型设计	0/120	0/3	大三(下)

2 实验设置

目前国内的计算机组成原理课程实验都已经逐渐向处理器设计这一方向靠拢，差别只在于实验的方式和难度。表2是参加此次会议的几个学校课程实验设置情况。

表2 计算机组成原理课程实验

学校	实验内容	实验平台
清华大学[1]	部件实验(运算器，控制器)； 内存、串口访问； 课程设计—16位类MIPS指令的五级流水线CPU，能够在上面运行一套监控程序，实现对实验系统的操作，实验3个人一组。	TEC2008(主芯片为FPGA)。
南京大学	功能部件设计(ALU、GRS等)； 单周期CPU、多周期CPU(22条不同类别指令)。 大作业(2~4人/组)。五级流水线CPU。	Altera DE2/70开发板。
国防科技大学	设计和实现一款16位CPU。	清华科教TEC-CA(主芯片为FPGA)。
中国科技大学	流水线设计+Cache+存储控制器。	FPGA实验板。
北京航空航天大学	存储器和运算器综合实验； 微程序控制器实验； 8位CISC CPU设计； 课程设计——32位MIPS-C CPU设计。	杭州康欣GW48(主芯片为FPGA)。
东南大学	硬件部件实验； 课程设计——32位MIPS指令CPU设计。	自己开发的FPGA板。

从表2可以看出，计算机组成原理课程的实验已经由以前的验证性部件实验逐渐过渡到处理器设计及计算机系统搭建这一层次上，具体体现在以下几个方面：

1) 使用可编程逻辑器件作为实验平台[2]，这样能够大大提高实验的灵活性和可操作性，根据学生能力的不同安排不同层次和难度的实验，充分发挥学生的主观能动性，在实验内容和形式上不断创新，同时也激发了学生的兴趣，实验样式也不再呆板和单调。但是，这样需要有更加完善的实验评价机制，做到公平和公正；还需要学生掌握硬件描述语言和相应的EDA工具软件，这些就需要对课程内容进行适当的调整或者得到先修课程的支持。

2) 指令集基本上都是MIPS或者类MIPS的，其好处是指令系统成熟，格式规整，有很好的技术和文档支持。使用这类的指令系统，学生能够更好的掌握和理解，设计出来的处理器结构也更加规范，而且有很多相应的设计文档和实例可供参考；其次，如果想要进行更高层次的实验内容，就需要相应的编译器等工具的支持，MIPS指令系统在这一点有很大的优势，

有了这些工具开发难度能够大大降低。因此目前来看采用MIPS指令系统是一个很好的方案。不过这样也有一些缺点，采用统一的指令系统限制了学生在指令系统设计上的灵活性，使得设计出来的处理器过于类似，过多的设计资料也使得学生可以更加容易偷懒，使得实验效果降低。

3) 基本上将流水线等知识应用到实验之中[5]。由于各个学校都在不同程度的推进计算机组成原理课程改革，普遍将流水线、高速缓存等内容加入了教学计划中，实验中也相应的加入了这些内容；同时MIPS指令系统能够很好的支持流水线的设计，现有的资料和教材大多也是围绕着流水线处理器设计展开的，因此流水线处理器的设计已经成为了各个学校实验的基本内容。在清华的计算机组成原理实验中并没有规定一定要实现流水线，要求学生完成多周期或者流水线处理器的设计，仅过几轮实验教学，学生普遍选择了流水线处理器的设计，因为多周期处理器的设计并不比流水线处理器设计简单很多，而且相应的设计资料较少。不过有一点是值得商榷的，就是为了组成原理实验有更好的显示度，在实验中加入了不少其他课程的内容，比如编译、操作系统等内容。这些内容安排在单独的综合实验课程中还可以，放到组成原理课程实验中就有些喧宾夺主了，学生会投入太多的精力在这些内容上，组成原理实验还是应该以理解计算机组成及工作原理为目的，不需要完成其他课程的内容，只有少数能力较强的同学在完成了基本内容后，才值得鼓励去做这些事情。

4) 实验规模较大，需要多个同学分工协作来完成。在以往的计算机组成原理实验中大多数是以验证性的实验为主，学生往往可以独立完成，但是处理器设计这一类的实验单靠个人完成对学生的压力太大，这就需要学生组成一个驼队来完成实验，这样不仅能够减轻学生的工作量，还可以培养他们团队协作的能力。一个团队规模控制在2～3人比较合适，人数太多会造成有人懈怠，达不到实验目的。

3　实验管理和评价

由于计算机组成原理实验内容和形式的更新，具体的实验管理方式和评价机制也有了很大的改动。以前是以实验室为主的实验模式，学生根据实验室安排的实验内容和时间来完成规定的实验；现在则是实验室根据学生的实验进度和需求，提供相应的实验支持，包括设备、场地以及人员等。

以清华大学的组成原理课程实验为例，前两个验证性的实验安排在实验室统一完成，帮助学生熟悉软件工具和实验设备，然后再安排课程大实验。大实验过程中会将实验设备发放给学生，让学生能够在宿舍进行实验，同时实验室保证一定的开放时间，方便学生来实验室做实验。在整个过程中安排三次集中的实验课程，实行小班教学，目的是能够更好的掌握学生目前的实验情况，控制实验进度和解决学生遇到的一些实际问题。在整个实验过程中，实验室的主要作用就是后勤保障和监督进度，协助和督促学生完成实验。

由于实验内容和形式的变化，实验已经不能简单的通过检查实验数据来评定一个实验完成的程度，需要从多个方面进行评价。对于我们的大实验，首先会提供一套标准的测试程序，通过这些程序来检查实验结果是否正确；然后学生需要针对自己的处理器提供自测程序来体现自己设计的处理器的特点，这些测试都是需要教师或助教现场检查的，检查的同时会询问他们在设计及实现的过程中是否独立完成以及各自的分工，以便确定是否存在抄袭现象、工作量分配是否合理。对于完成较好或者有所创新的小组给与加分奖励[7]，并鼓励其在实验总结课上展示自己的成果。将这些汇总然后结合实验报告及平时实验情况，就能够给出一个比较全面公平的实验评价结果。

在这种实验管理模式和评价机制下，能够很好的提高同学的积极性和对实验的整体把握程度，教师也能够掌控好实验进度和学生掌握情况，达到很好的实验效果，不过这需要教师和助教通力合作，所花费的精力也比较多。

4　结语

随着实验技术的不断进步，计算机组成原理课程实验在内容和形式上已经发生了很大的变化，各个高校都有着自己的发展思路，但是大的方向是一致的，总体说来就是实验已经由验证型实验过渡到设计型实验，内容也变成了在可编程芯片上进行处理器设计，进而形成一个简单的计算机系统，可以说是计算机组成原理课程实验已经由验证计算机各部分功能逐渐过渡到设计及搭建计算机系统这一层面上。

(下转 137 页)

参考文献：

[1] 中国就业培训技术指导中心编. 职业概论[M]. 北京: 中国劳动社会保障出版社, 2009: 254-255.

[2] 石令明. 校企合作框架下的高职生产性实训研究[EB/OL]. [2010-04-15]. http://gjsf.lzzy.net/content.aspx?id=115483494680.

[3] 丁金昌. 校内生产性实训基地建设的探索[R]. 北京: 全国高职高专校长联席会议, 2008.

[4] 马广. "引企入校"建设生产性实训基地的实践[J]. 职业技术教育, 2008(32): 60-61.

Training School-enterprise Cooperation and Productive Logistics Information Technology Professional Core Competencies of Students in Vocational

MI Zhi-qiang, DENG Zi-yun, JIAN Xiang-hui

(Hunan Vocational College of Modern Logistics, Changsha 410131, China)

Abstract: Logistics information technology, core competencies of professional analysis of vocational students to explore logistics information technology professional and productive school-enterprise cooperation model, and train school-enterprise cooperation and productive logistics information technology professional core competencies students to study the initial implementation effect of an analysis, thinking ahead to development, and vocational colleges for the same productive practice teaching experience provided a useful reference.

Key words: vocational education; production of school-enterprise cooperation; logistics information technology, professional core competenciesy

(编辑：白杰)

(上接 129 页)

参考文献：

[1] 王诚, 刘卫东, 宋佳兴. 计算机组成与设计[M]. 北京: 清华大学出版社, 2008: 6-7.

[2] 罗克露, 谭华, 单立平. 计算机组成原理实验改革探索[J]. 实验科学与技术, 2004(3): 57-59.

[3] 郝秉华. 结合 EDA 的计算机组成原理实践教学探究[J]. 内蒙古科技与经济, 2009(11): 103-104.

[4] 叶雪琴, 唐建宇, 熊成. 基于 EDA 的计算机硬件课程实践教学的研究[J]. 计算机教育, 2007(7): 90-93.

[5] David A. Patterson, John L. Hennessy. 计算机组成与设计: 硬件软件接口[M]. 北京: 机械工业出版社, 2006: 368-383.

[6] 王诚, 刘卫东, 宋佳兴. 计算机组成与设计实验指导[M]. 北京: 清华大学出版社, 2008: 12-48.

[7] 马明涛. 计算机组成原理课程的实践教学方法初探[J]. 山西财经大学学报, 2009(11): 21.

Investigation and Research on the Experiment Teaching of Computer Organization Course

LI Shan-shan, QUAN Cheng-bin

(Lab for Computer Education, Tsinghua University, Beijing 100084, China)

Abstract: The experiment of the Computer Organization can not catch up with the requirement for experiment teaching. This paper investigates the computer organization experiment of several colleges and universities, and discusses the current status on course setting, experiment setting, experiment management and experiment evaluation of the Computer Organization, points out that the computer prototype experiment on programmable device is the trend of the experiment teaching on computer organization course.

Key words: Computer Organization; experiment teaching; programmable device; experiment evaluation

(编辑：彭远红)

✦教育与教学研究

文章编号：1672-5913(2012)11-0090-04　　中图分类号：G642

面向“计算机使用者”的计算机体系结构课程

郑纬民，张悠慧
（清华大学 计算机科学与技术系，北京 100084）

摘　要：目前国内高校开设计算机体系结构课程的总体思路通常是面向“计算机设计者”来组织课程内容，但是随着各个学科交叉方向的发展，这一思路无法满足不同专业学生的需求。文章介绍清华大学开设的面向“计算机使用者”的计算机体系结构课程，强调从程序员以及软件运行的角度来学习计算机体系结构，讲解其基本组成与工作原理，分析计算机体系结构对系统性能的影响，并且比较其与传统课程的不同之处。

关键词：计算机体系结构；汇编语言；虚存

1　背景

计算机体系结构课程是国内外很多大学的本科生骨干课程。该课程将软件和硬件理论结合讲述，覆盖了计算机组成、体系结构、微体系结构(Micro-architecture)、编程等多方面内容。它对于学生理解计算机内部结构，更好地掌握硬件设计和软件优化技能，从而建立关于计算机整机系统的完整概念是很有裨益的，另外，它可以为学生进一步学习计算机操作系统、编译器和网络互联打下基础。

目前，国内高校开设计算机体系结构课程的总体思路多是面向“计算机设计制造者”来组织课程内容，内容与教材上很多都是参考 John L. Hennessy 与 David A. Patterson 的 *Computer Architecture*[1] 或者其他组织结构上近似的教材。但随着各个学科交叉方向的发展以及软件工程专业的完备与独立，以“计算机设计者”的身份作为课程出发点难以满足各方面的不同需求，因为对于大部分学生而言，学习微体系结构并不是为了设计处理器，而是为了了解处理器的内部结构与工作原理后能写出性能更优的程序；学习操作系统也不是为了设计操作系统，而是了解操作系统后能正确地使用系统调用；学习汇编语言不是为了写汇编语言程序，而是为了了解自己用高级语言写的程序编译后会成为什么样子，从而帮助自己调试/优化程序。所以，我们需要在原先的课程体系中引入新的、交叉的内容，使其能够满足更广泛的需求。

对此，笔者认为要根据授课对象的不同开设多种计算机体系结构课程：一种仍是传统的面对系统工程师的课程，从计算机构建角度出发，讲解微处理器体系结构、存储层次、多核/多机系统组织与互连等内容，侧重计算机内部结构与硬件相关的知识和理论；另一种则面对更为广泛的“计算机使用者”（包括软件工程师等），强调从程序员以及软件运行的角度来看计算机体系结构，讲解其基本组成及其工作原理，分析计算机组成对系统性能的影响。

郑纬民教授

存在的对编程“知其然而不知其所以然”的现象。这一现象往往表现为学生的编程能力有差异，而且在系统知识方面，包括与程序相关的运行环境、操作系统接口、计算机的指令/数据标识与内存布局(Memory Layout)、微体系结构等方面的知识均较为欠缺。这对于学生进一步掌握编程技巧/优化技术，深入学习相关课程是不利的。

笔者在清华大学计算机系、软件学院等本科生教学体系内，参照上述思路，开设了不同的“计算机体系结构”课程，在下文中将较为详细地介绍“面向计算机使用者”的课程教学纲要、内容组织以及实验等，并比对它与传统课程的差别。

2 课程内容与组织

根据《中国计算机科学技术百科全书》的定义[2]，计算机体系结构包括计算机系统的物理或者硬件结构、各部分组成的属性以及这些部分的相互联系。狭义的计算机体系结构指的是从系统软件开发人员角度看到的计算机系统的功能行为和概念结构；广义的范围则涵盖更多，根据ACM相关课程[3]的定义，其包含数据的机器表示(Machine level representation of data)、指令集与程序的机器表示(Assembly level machine organization)、存储层次结构(Memory system organization and architecture)、系统接口与通信(Interfacing and communication)、微体系结构、多核/多机系统、性能优化以及虚拟化等内容。

因此，该课程的内容组织仍然需要与传统的课程一样，基本涵盖上述广义范围内的多数内容，但教学目的与内容组织思路不同。我们借鉴了卡内基·梅隆大学的相关课程[4]，强调从程序员以及软件运行的角度去看计算机系统，讲解其基本组成及其工作原理，分析计算机组成对系统性能的影响；更为注重程序的数据、指令在计算机系统的表示、存储与运行流程，从应用与高级语言开始自上而下的组织相关概念与内容。课程的参考书选用了与上述课程[4]配套的《深入理解计算机系统》[5]。

课程大纲以程序运行对硬件系统的需求为中心，涵盖了以下内容：1)信息的表示和处理、整表示与运算。2)程序的机器表示：汇编指令种类与格式、汇编指令与C语言等。3)微体系结构：微处理器基础知识、多周期处理器、流水线处理器。4)程序优化：处理器缓存基础、多种类型缓存组织、存储层次结构与代码优化、处理器结构无关的优化、处理器结构相关的优化。5)系统调用：异常处理流程与API接口、进程层次结构、进程间通信等。6)系统虚存：程序内存布局/内存分配/操作系统虚存管理与工作流程/微体系结构的TLB组织等。7)系统I/O：UNIX I/O接口、网络编程、并发程序等。

配套实验为处理器设计大实验，要求学生独立设计CPU系统结构，包括指令系统、数据表示方式、寻址方式、寄存器结构、存储系统和流水线结构等；要求学生采用一种硬件描述语言和大容量FPGA器件来实现自己设计的CPU系统结构，并掌握CPU芯片的调试和测试方法。

3 与传统课程的比较

上述大纲与传统的体系结构课程是不同的，目的在于回答以下问题：哪些与计算机系统有关的知识可以帮助程序员写出更好的程序（正确性和高性能）。

因此，一个明显的特点是C语言的语法、接口、运行流程等纵贯整个课程，将程序表示、运行过程、微体系结构、系统接口等内容串起来，使得学生能够明白到底什么是“程序”与“程序

张悠慧副教授

运行”，而不仅仅是“编码”。

下面分别从机器语言、虚存、微体系结构与程序优化等角度来说明即便是类似的内容，在“面向使用者”与“面向设计者”的课程中也是不同的。

1）机器语言。

课程直接采用主流的X86指令集作为入门，而不是原先体系结构课程中的MIPS指令集，主要是对设计实验性CPU，MIPS更加简单易上手，而X86从编程来看更加实用。

当然，X86指令集过于复杂，在讲解微体系结构时不好掌握难易尺度（与基于MIPS相比）。一方面，我们直接采用了教材《深入理解计算机系统》及其课程材料，使用简化的X86指令集Y86及其微体系结构，这样既保持了一定的X86指令集的特点，又使其对于教学更加实用；同时，我们设计了针对Y86的课程实验，具体内容为在Y86流水线处理器中增加指令缓存与数据缓存功能模拟，研究修改Y86模拟器源代码，加入指令缓存与数据缓存功能。这两个缓存要求分开实现，每一个缓存的基本参数可以在模拟器编译/或者运行前配置，参数包括：(1) 缓存大小；(2) 缓存能够支持直接映射、2-way、4-way模式；(3) 缓存的line长度可以设定，8byte、16byte、32byte；(4) 缓存miss时的访问内存延时可以设定为100、200、300个周期；(5) 采用write-through / read-allocation策略；(6) 当缓存命中时，认为其不会引起流水线stall；(7) 如果缺失，则引起流水线stall，stall的周期数即为设定的访问内存延时。

虽然汇编作为一门实用语言现在已不是主流语言了，但是它作为一种体系结构设计思想的体现是无可替代的。因此在课程中我们逐步弱化汇编作为一门语言的内容，突出汇编承接上层应用、高级语言与下层处理器结构的特点，融入汇编与C语言、与性能优化等交叉内容。比如，在“程序的机器表示”中引入了典型的C语言代码块编译为汇编后的识别/解释，使用汇编直观解释运行时栈/递归的变化，不同时代编译器产生的同一段C代码的不同汇编语言的比较与解释（可以联系微体系结构）等内容。

2）虚存。

从计算机的专业课程角度来看，“虚存”这一概念在多门课程里均有体现，如操作系统相关课程会涉及软件虚存管理、进程地址空间隔离、内存分配、缺页中断、mmap接口等内容；而在传统的“计算机体系结构”课中着重介绍处理器内部相关微体系结构的内容，尤其是TLB实现、工作流程以及部分中断的处理流程。仅从软件和硬件角度都不能完整地介绍虚存的必要性与工作流程，必须结合起来讨论。因此，在“面向使用者”的课程中，我们将微体系结构层面的TLB和较为宏观级别的虚存管理串联起来，使得学生们能够将各个层面的相关内容贯通起来，更加清楚地掌握其工作流程（见图1）。

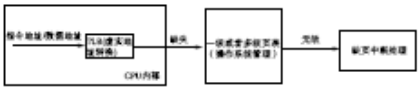


图1 虚实地址转换流程

3）程序优化与体系结构。

① 缓存优化。

在“面向设计者”的课程中，这部分内容注重于缓存自身的结构设计、优化设计与电路设计，包括结构层面的各种策略的概念、特点的掌握与比较。

新的课程内容在强调Cache的空间/时间局部性原理以及基本组织、工作流程的基础上，将提升面向程序员的Cache相关代码优化技巧的重要性，分为以下几个方面。

● 将高层C代码运行过程中的一些访存行为（比如进行矩阵运算时的访存）与底层缓存的具体查找、插入、替换等贯穿起来，使得缓存命中/缺失对于高层代码的影响显得更为直观。图2是三个功能相同的矩阵乘函数，其访存次序的不同对于缓存命中率以及运行性能有直接影响，通过这个例子可以直观地联系底层缓存的替换过程来说明相关优化的必要性。

● ICache的优化：精简代码执行路径，简化调用关系，减少冗余代码等。

图 2　矩阵乘 [3][4][5]

● DCache 的优化：通过一个具体的实例来说明数据间可能存在的缓存冲突，比如，某个程序会非常频繁地使用 K 段空间的数据，而这 K 段空间的地址的低位又非常接近，那么它们就会使用相同的 cache set。如果 K 大于 cache 的路数(way)，那么 cache 的缺失会非常频繁，因为每一段空间的数据会不停地将同一 cache set 内的其他数据替换出去。

可以通过优化来改善数据的空间局部性和时间局部性来减少数据失效，基本方法为：数据合并；内外循环交换、循环融合；分块；减小程序等。

② 指令级并行优化与微体系结构。

课程在处理器微体系结构方面要求学生掌握多周期处理器与流水线处理器的简单工作原理，而对更进一步的指令级并行执行结构的说明则更多地结合软件优化来展开。

一个具体的例子就参考了文献 [5] 中连续乘的例子。同样的一组连续乘运算，采用了下面这一运算次序会导致其在流水线内的乘法器上被强制串行计算。

((((((((((1 * x0) * x1) * x2) * x3) * x4) * x5) * x6) * x7) * x8) * x9) * x10) * x11) .

而调整次序后的 ((((((1 * x0) * x2) * x4) * x6) * x8) * x10) *((((((1 * x1) * x3) * x5) * x7) * x9) * x11) 则可能获得近两倍的性能提升。

主要原因在于现代通用处理器往往具备下列功能：

● 流水线化的乘法部件；

● 多个乘法部件；

● 指令发射部件能够发现一定的指令并行性。

这就使得第二种写法的连续乘能够充分利用这些硬件特征。具体执行的流程图对比如图 3 所示，其中方框表示乘法器。

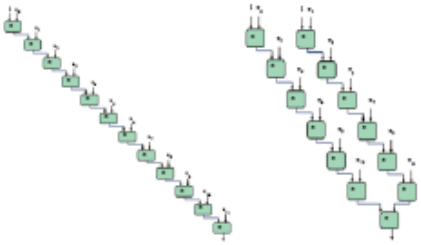


图 3　连续乘的处理器执行 [5] 对比

此外，在微体系结构教学中我们也强调流水线结构对性能的影响，尤其是条件跳转指令的不利影响，从而介绍设计 conditional mov 指令的必要性（当然其也有局限），并可以与上文提到的“不同时代编译器产生的同一段 C 代码的不同汇编语言的比较与解释”联系起来，加深印象。

4　结语

从学生反馈来看，这门课不但传授了计算机体系结构的相关知识，更为重要的是课程将应用、编程、系统软硬件等层面的内容贯穿起来，帮助学生建立计算机系统的整体思维，这对本科生了解什么是计算机、它是如何工作的，进而对其产生兴趣是非常有利的。

参考文献：

[1] Hennessy John L, Patterson David A. Computer Architecture: A Quantitative Approach[M]. 5th ed. San Francisco:Elsevier, 2012.

[2] 张效祥. 计算机科学技术百科全书[M]. 北京: 清华大学出版社, 2005: 21.

[3] The Joint Task Force on Computing Curricula Association for Computing Machinery IEEE-Computer Society. Computer Science Curricula 2013 (draft)[EB/OL]. [2012-04-15]. http://www.sigart.org/CS2013-EAAI2011panel-RequestForFeedback.pdf.

[4] Bryant Randal E, O'Hallaron David R. Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e)[EB/OL]. [2012-04-15]. http://csapp.cs.cmu.edu/public/instructors.html.

[5] Bryant Randal E, O'Hallaron David R. Computer Systems: A Programmer's Perspective[M]. 龚奕利, 雷迎春, 译. 北京: 机械工业出版社, 2010.

[6] Bryant Randal E. Code Optimization: Machine-independent Optimizations[EB/OL]. [2012-04-15]. http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f04/lectures/class10.pdf.pp.48-51.

（编辑：彭远红）

文章编号：1672-5913(2014)15-0107-04　　中图分类号：G642

面向系统能力的计算机组成原理实验实施

李山山，刘卫东

（清华大学 计算机科学与技术系，北京 100084）

摘　要：系统能力培养是计算机专业学生培养的核心要求，因此要在理论教学和实验教学两方面重视学生的系统设计和应用能力。作为计算机专业的核心基础课，计算机组成原理在实验教学中要从系统的角度开展实验。文章介绍清华大学计算机组成原理课程的实验体系。

关键词：系统能力培养；计算机组成原理；实验教学；实验实施

0　引　言

李山山

计算机组成原理是计算机专业的核心基础课程[1]，在课程体系中占有重要位置，起到承上启下的作用。教师以程序设计语言及数字逻辑电路为先导课程[2]，重点解析计算机硬件系统的基本组成、运行原理和协同工作机制，分析计算机组成对系统性能的影响，阐述计算机系统的基本设计方法，帮助学生建立计算机整机系统的概念[3]，为学习系统结构、操作系统等课程的学习提供扎实基础。计算机组成原理虽然是一门偏重硬件的课程，但却是很多系统和软件课程的基础，它的实践性很强，需要学生在实践过程中充分理解和掌握计算机的工作原理和具体结构。

1　面向系统能力的实验教学

为了适应新时代的计算机专业人才需求，计算机专业要求培养学生的系统设计和系统应用能力，使学生能够掌握计算机系统的基本工作原理并理解计算机软硬件系统的相互作用关系。为了满足系统能力培养的需求，计算机专业本科教学课程体系中核心课程的教学需要进行调整，让课程之间的联系更紧密，衔接更顺畅[4]。各课程的教学实验也要围绕着系统能力培养展开，使课程实验内容能够自然衔接，形成一个整体的实验教学体系，因此各课程的实验教学要能够承接前导课程实验的内容，并为后续课程进行铺垫和准备[5]。

对于计算机组成原理的实验教学来说，该课程是专业核心课程，同时是第一门具有计算机专业特色的硬件基础课，因此课程的教学实验设计是一项具有挑战性的工作。首先，实验内容既要有分离的又要有系统的[6]，不但要有能够体现计算机各部件的基本硬件组成和内部运行原理的基础实验，而且要有能体现计算机内部协同工作机制尤其是软硬件协同工作的系统实验，甚至还需要一些能结合编译系统、操作系统等课程内容的综合性实验供学有余力的学生选做。另外，在实验方法上，既要满足硬件系列课程基础性的要求，又要采用当前主流的硬件系统设计和实现方法，以提高学生的硬件系统设计和调试能力。实验平台设计要注重体现系统观点，提供软硬件系统框架，以承载实验内容并满足实验方法上的要求。

刘卫东

第一作者简介：李山山，男，工程师，研究方向为计算机实验教学、计算机体系结构和物联网，lishanshan@tsinghua.edu.cn。

为了满足上述要求，我们结合课程的特点，对计算机组成原理的教学实验进行改革，设计开发包括系统层次实验内容、硬件实验设备、软件实验工具在内的一整套实验方案。经过近 5 年的实验教学，我们通过在实际教学过程中不断改革创新，形成一个可实施的系统层次基本实验教学体系。

2 实验体系及系统构成

计算机组成原理实验体系的设计参考计算机层次结构，实验更贴近真实的计算机系统，也满足系统的概念。实验教学不仅要设计出计算机处理器的主要硬件逻辑，还要设计出一个具有相对完整系统结构的计算机系统，即一台能够运行的教学计算机，我们称之为 THINPAD 教学计算机系统。对照计算机的层次结构，THINPAD 教学计算机系统层次如图 1 所示。

图 1 THINPAD 教学计算机系统层次

从图 1 可以看出，THINPAD 教学计算机系统具有相对完整的层次结构，它以硬件电路为基础，具有相对完备的指令系统，向上支持监控程序作为软件控制系统，再向上层次支持汇编语言程序，从而形成了一个独立的计算机系统，同时 THINPAD 还有一系列的工具软件支持实验的开展和进行。

THINPAD 教学计算机系统的硬件电路板如图 2 所示，采用目前流行的大规模可编程芯片 FPGA 为主要的实验芯片，存储器使用大容量的 SRAM 和 Flash 存储芯片，数据总线采用指令和数据独立的存储结构，再加上丰富的外围接口，为计算机组成原理课程提供更加灵活方便的实验方式[7]。

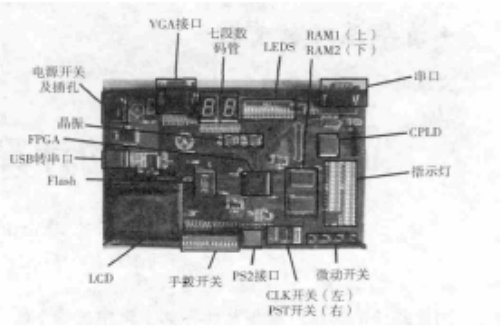


图 2 硬件电路板

指令系统是计算机系统的灵魂和核心，THINPAD 教学计算机的指令系统主要满足教学的需要，对指令系统的完备性、兼容性方面要求不是很高。从教学角度考虑，指令系统应基本完备，满足运行简单监控程序的需要；指令格式和寻址方式应尽量简单，可在硬件上简洁实现。根据这些特点和要求，实验系统借用 MIPS16e 指令系统中部分指令的格式和功能进行必要的调整，形成 THINPAD 教学机指令系统，见表 1。

表 1 指令系统

ADDIU	ADDIU3	ADDSP3	ADDSP
ADDU	AND	B	BEQZ
BNEZ	BTEQZ	BTNEZ	CMP
CMPI	INT	JALR	JR
JRRA	LI	LW	LW_SP
MFIH	MFPC	MOVE	MTIH
MTSP	NEG	NOT	NOP
OR	SLL	SLLV	SLT
SLTI	SLTU	SLTUI	SRA
SRAV	SRL	SRLV	SUBU
SW	SW_RS	SW_SP	XOR

44 条指令中，监控程序的实现使用其中的 25 条，它们是 ADDIU、ADDIU3、ADDSP、ADDU、AND、B、BEQZ、BNEZ、BTEQZ、CMP、JR、LI、LW、LW_SP、MFIH、MFPC、MTIH、MTSP、NOP、OR、SLL、SRA、SUBU、SW、SW_SP，这些指令构成的集合称为基本指令集，缺少其中的任何一条指令都将导致监控程序无法运行。

有了指令系统就可以据此设计出软件系统，THINPAD 最主要的软件是监控程序，运行在硬

件平台上，相当于一个简易的操作系统，可接收用户输入的命令并完成相应功能，如运行用户程序、将用户输入的汇编语句翻译成机器指令并保存在教学机存储器中、反汇编机器指令为汇编语句、显示内存中内容等，有了监控程序就可以进一步开发应用软件。

由于指令系统中没有设计专门的输入 / 输出指令，而是将输入 / 输出功能由访存指令 LW 和 SW 通过指定的端口地址实现，因此教学计算机监控程序将存储空间和输入 / 输出端口统一编址，形成统一的地址空间并将其划分为系统程序区、用户程序区、系统数据区及用户数据区 4 部分，另外设置一些地址作为对外部设备访问的接口。监控程序划分的地址空间段见表 2。

表 2　监控程序划分的地址空间段

功能区	地址段	说明
系统程序区	0x0000-0x3FFF	存放监控程序
用户程序区	0x4000-0x7FFF	存放用户程序
系统数据区	0x8000-0xBEFF	监控程序使用的数据区
Com1 数据端口 / 命令端口	0xBF00-0xBF01	串口的端口
Com2 数据端口 / 命令端口	0xBF02-0xBF03	第 2 个串口的端口
预留给其他接口	0xBF04-0xBF0F	保留
系统堆栈区	0xBF10-0xBFFF	用于系统堆栈
用户数据区	0xC000-0xFFFF	用户程序使用的数据区

为了配合实验开展，THINPAD 教学计算机系统同时配备必要的软件工具，包括以下几方面。

1）模拟器 simulator。

运行在 PC 机中，主要功能是模拟教学计算机指令系统编写汇编程序的执行情况。在教学实验中，可用来熟悉教学机指令系统、作为汇编语言程序的实验平台等，也可在硬件调试过程中作为参照系统使用。

2）汇编器 assembler。

虽然监控程序可完成一些汇编语句的汇编工作，但是监控程序功能还比较薄弱，比较适合一些功能简单、需要语句少的汇编程序编辑和汇编工作。一些功能复杂、规模比较大的汇编程序还需要通过 PC 机进行编辑，然后直接汇编成教学机机器语言程序，再放置到教学机硬件上调试运行。

3）终端程序 term。

终端程序在 PC 机上通过串口和教学机连接，这样 PC 机可作为教学机的输入 / 输出设备。

4）数据通讯程序 FlashAndRam。

教学计算机的许多实验需要在其存储器中预先加载程序或者数据，数据通讯程序可以完成 PC 机和教学机存储器之间的数据交换。

有了以上这些系统软件，就可以构成基本完备的实验计算机软件体系，也可以有效地帮助学生开展系统层次的实验。

3　实验实施方案

实验教学的具体实施不可能一蹴而就，一开始进行计算机系统设计实验需要由浅入深、由部件到系统，逐步引导学生设计完成自己的计算机系统。在具体的实验教学开展过程中，我们设计验证性、设计性和综合性 3 个层次的实验内容，实验的数量不一定要很多，但每个实验都为后面的系统实验做准备。

经过多年的摸索，我们探索出一个行之有效的实验实施方案。首先，推动先导课程数字逻辑课程的实验教学，在该课程中安排学生接触硬件设计语言和可编程器件的内容，掌握硬件设计的基本方法，从而为组成原理课程实验做好准备和铺垫。在组成原理课程中，指令系统是学生在课程中最先接触的内容，同时指令系统也是整个硬件设计的基础，需要让学生充分掌握。在该部分我们安排一个验证性实验，让学生编写汇编程序并在模拟器中运行和调试，主要目的是让学生熟悉教学计算机 THINPAD 的指令系统，包括指令功能、指令格式和寻址方式等，同时了解教学计算机模拟器以及监控程序的功能并熟练使用这些软件。在该实验中教师也要求学生阅读监控程序的源代码，为后期在实际硬件上运行调试做准备。

随着课程的进行，当讲解算术逻辑单元 ALU 时，我们会安排第一个硬件实验内容——设计和实现 ALU 部件。教师通过该实验让学生掌握 ALU 的基本设计方法和数据传送通路，熟悉 THINPAD 硬件实验系统。该实验是一个设计性实验，设计好的 ALU 会被当作后续系统实验中处理器的一个重要组成部分。

当讲解存储器时，教师会安排内存储器系统实验。该实验的主要内容是设计一个状态机和内存读写逻辑，完成对 THINPAD 实验板上存储器 RAM 的访问。通过该实验，学生可以熟悉内存储器的配置以及与总线的连接方式，掌握教学机内存的访问时序和方法并理解总线数据传输的基本原理。在实验过程中我们使用数据通讯程序 FlashAndRam 协助开展实验。

讲解输入/输出时安排串行口访问实验。该实验的主要内容是实现教学机和 PC 机上串口的通信，学生可以通过该实验掌握计算机输入/输出系统的设计。在具体实验过程中，我们发现存储和输入/输出是系统实验的一个难点，有了这两个实验，后续的系统实验中有关指令/数据访存以及串口通讯的部分就没有问题，而且剩下的硬件设计内容都与外部器件无关，只需要在可编程芯片内部设计即可。

上述实验都完成后，课程就进入计算机系统设计实验，教师要求学生在 THINPAD 教学计算机硬件平台上设计并实现一个完整的计算机系统，使其能运行教学计算机监控程序，内存地址分配应满足监控程序使用内存的要求。实验中使用数据通信程序 FlashAndRam 将监控程序的二进制代码直接装入到内存 RAM 中，然后通过 PC 机上的终端软件与教学计算机通讯并控制程序运行。该实验是一个综合性的系统级实验，学生通过该实验能够更全面、更系统地掌握计算机系统知识。

在实验时间安排上，前面的部件实验都安排成单次实验，占用一次实验课时，学生也可以在课下完成实验。对于系统综合实验，教师会在理论课的课时中安排３次专门的实验课，采用小班上课的形式控制实验进度。实验课的主要内容是讲解当前进度下会遇到的问题并进行实验答疑。

在课程实验安排中并没有安排传统的控制器部件实验，这主要是因为控制器的设计与处理器息息相关，很难将其分离出来，而且在系统实验中会实现控制器，考虑到课程实验时间安排，也没有设置专门的控制器实验。在实验过程中，实验教学主要强调计算机的硬件设计，尤其是处理器的设计，而不关注操作系统和应用程序的设计，这些都是用来验证设计的计算机硬件能否正确工作，因此监控程序等软件都是提供好的，包括源代码，学生只需要熟悉这些软件的结构和使用方法，为硬件设计提供支持。同时，通过学习这些软件，学生为后续的操作系统和编译原理课程实验做好准备。

对于更高层次的计算机系统实验，目前我们已经在该实验系统上将操作系统和编译原理的课程实验融合进来，形成一个完整的计算机系统实验体系，同时专门安排这样的实验课程。

4 结 语

计算机组成原理实验是本科计算机硬件实验中的一个重点和难点，如何让学生在系统层次进行计算机组成原理实验以及如何提高学生的系统能力是我们一直关注的问题。通过多年努力，我们探索出一套行之有效的实验方案，设计出面向计算机系统的实验内容，取得良好效果。实验中还有一些需要进一步改进的地方，包括如何更加合理地调整实验进度和难度，增加监控程序功能如 debug 功能；进一步完善一些硬件的调试工具，以便更好地支持实验；适当引入操作系统内容如 TLB 如何在硬件实验中使用等。

参考文献：
[1] 唐朔飞，刘旭东，王诚．“计算机组成原理”课程教学实施方案[J]．中国大学教学，2011(11)：42-45.
[2] 丁柏秀，王文涛，吕晓丽．“计算机组成原理”教学内容及教学方法探讨[J]．长春理工大学学报，2012(1)：196-197.
[3] 袁春风，张泽生，蔡晓燕．计算机组成原理课程实践教学探索[J]．计算机教育，2011(17)：110-114.
[4] 王志英，周兴社，袁春风．计算机专业学生系统能力培养和系统课程体系设置研究[J]．计算机教育，2013(9)：1-6.
[5] 杨新凯．面向计算机系统能力培养的数字逻辑课程教学改革探讨[J]．教育教学论坛，2014(6)：148-149.
[6] 张磊，郑榕，田军峰．计算机组成原理理论实验教学无缝结合的新方法[J]．实验室研究与探索，2013(5)：168-172.
[7] 刘亚楠，刘卫东，张小平．THINPAD教学计算机实验平台设计[J]．实验技术与管理，2012(11)：115-118.

（编辑：宋文婷）

文章编号：1672-5913(2010)15-0016-05 中图分类号：G642 文献标识码：B

美国计算机硬件系列课程与实验的调研报告

李山山，全成斌
(清华大学 计算机实验教学中心，北京 100084)

摘 要：在调研美国中部有代表性大学的硬件系列课程后，整理出这些学校相关课程的课程设置、实验内容和实验管理等内容，举例介绍了实验的具体安排。通过分析这些学校的硬件系列课程和实验，找出教学和实验方面的不足，提高课程教学质量和实验水平。

关键词：计算机教学；硬件系列课程；实验安排；实验管理

赴美国中西部大学考察学习归来，我们重点学习了各层次大学计算机科学(CS)和计算机工程(CE/ECE)本科专业课程设置、实验教学体系、实验环境建设以及实验内容开发的情况，本文重点阐述计算机硬件系列课程的设置、实验安排、实验管理等方面的调研情况。

1 课程设置

美国大部分高校将计算机学科分成了科学与工程两个部分，分别称为 CS(Computer Science)和 ECE(Electrical and Computer Engineering)，各自侧重的方面也不同，在硬件系列课程中体现得尤为明显。CS 注重讲清计算机的硬件组成和工作原理，实验也主要以软件模拟实验为主；而 ECE 注重讲解计算机的组成和构建方式，实验也会在实际的硬件电路上进行。也有一些学校将 CS 和 ECE 二者结合为一个系，一般称作 EECS 或 ECS。

在课程设置上，各校一般会安排一门入门课程，将计算的相关内容简要介绍一下，其中包括计算机组成原理的基础内容，然后在后续选修课程中再深入介绍组成原理和系统结构的内容。各个学校在硬件系列课程的设置中不会把数字逻辑、计算机组成原理和系统结构分得很清晰，课程设置也不尽相同，不过总的介绍内容都是类似的。下面以 UIUC 的 CS、Purdue 的 ECE 和西北大学的 EECS 为例详细介绍课程设置及内容，如表 1、表 2 和表 3 所示。

表 1 UIUC CS 硬件系列课程[1]

课 程	主要内容
CS 231: Computer Architecture I (计算机体系结构I)	从门电路开始，内容包括组合逻辑和时序逻辑、算术逻辑单元、存储器和控制器。
CS 232: Computer Architecture II (计算机体系结构II)	组成与系统结构的基本内容，包括汇编语言、指令集系统、数据表示、子程序、输入输出、加载和链接、流水线、高速缓存、性能分析等。(http://www.cs.uiuc.edu/class/fa09/cs232/)
CS 433: Computer System Organization (计算机系统组成)	计算机系统结构深入课程，包括计算机系统分析与设计、各部分之间的管理、性能分析、指令系统设计、流水线、向量机、存储系统。(http://www.cs.uiuc.edu/class/sp09/cs433/)

作者简介：李山山(1979-)，男，工程师、硕士，研究方向为计算机系统结构、计算机实验系统；全成斌(1972-)，男，高级工程师、博士，研究方向为计算机系统结构、计算机实验系统。

表 2　Purdue ECE 硬件系列课程[2]

课　　程	主要内容
ECE 26600 - Digital Logic Design (数字逻辑设计)	介绍逻辑设计，着重于设计和实现。内容包括布尔代数、逻辑优化、电路特性、组合逻辑与时序逻辑电路分析、时钟电路、状态机及应用。
ECE 26700 - Digital Logic Design Laboratory (数字逻辑设计实验)	是ECE26600课程对应的实验课程。
ECE 36500 - Introduction to the Design of Digital Computers (数字计算机设计导论)	介绍计算机系统的硬件组成，包括指令集选择、算术逻辑单元、硬连线和微程序控制器、内存组织、IO接口设计。
ECE 43700 - Computer Design and Prototyping (计算机原型机设计)	介绍计算机组成和设计，包括指令集选择、算术逻辑单元设计、数据通路设计、控制策略、流水线、存储系统、IO接口设计。(http://cobweb.ecn.purdue.edu/~ece437l/)

表 3　西北大学 EECS 硬件系列课程[3]

课　　程	主要内容
EECS 203 - Introduction to Computer Engineering (计算机工程导论)	简要介绍计算机工程设计，包括布尔代数、逻辑门电路、组合逻辑电路的设计和化简、译码器/多路选择器/加法器、时序电路和触发器、汇编语言。(http://www.eecs.northwestern.edu/academics/course/eecs_203/)
EECS 303 - Advanced Digital Logic Design (数字逻辑设计)	介绍数字逻辑设计、组合逻辑和实习逻辑电路分析、算术逻辑单元、EDA工具、使用VHDL进行模拟和仿真。(http://www.eecs.northwestern.edu/academics/course/eecs_303/)
EECS 361 - Computer Architecture I (计算机体系结构I)	从整体上理解和设计计算机系统、指令系统、体系结构设计、数据通路设计、控制器设计、单周期/多周期/流水线处理器的设计、冲突检测和数据旁路、存储系统及高速缓存和虚拟内存、外围设备和接口。(http://www.eecs.northwestern.edu/academics/course/eecs_361/)
EECS 362 - Computer Architecture Project (计算机体系结构实验)	团队合作完成一个全指令集的支持流水线的处理器的详细设计，包括各个部件、数据通路、控制电路各部分的设计、使用VHDL语言和软件工具进行模拟和仿真。(http://www.eecs.northwestern.edu/academics/course/eecs_362/)

从以上课程内容可以看出，这些学校的硬件系列课程虽然名称略有不同，但是主要内容是类似的，都是按照数字逻辑、组成原理、系统结构这一条主线逐步深入的。不过 CS 偏重理论方面，着重讲清硬件是如何工作的，构架是什么样的以及如何进行性能分析；而 ECE 则注重应用，着重关注硬件是如何组成的，各部分之间的关系以及如何设计。这一点在实验上体现得十分明显，下一节将重点进行讨论。

2　实验内容

在各个学校的实验安排中，CS 一般很少安排实际的硬件实验，而主要通过软件模拟的方式进行，或者安排软件性能分析实验，要求只要掌握硬件的工作原理就可以了，不需要接触实际的硬件；而 ECE 大部分将实验安排在实际的硬件电路上进行，让学生真正设计和实现一个硬件电路。

下面分别以 UIUC 的 CS、Purdue 的 ECE 和西北大学的 EECS 为例介绍具体实验内容，如表 4、表 5 和表 6 所示。

从表 4 可以看出，UIUC 的 CS 专业在硬件系列课程实验方面都是进行软件实验，不涉及实际的硬件电路。数字逻辑使用模拟软件进行电路的仿真，组成原理和系统结构则是进行了软件层次的实验，并没有采用处理器设计这一类实验。

从表 5 可以看出，Purdue ECE 的实验大多是在实际硬件上进行的，数字逻辑采用 TTL 电路进行实验，组成原理和系统结构实验既采用了软件模拟，也采用了基于 FPGA 进行处理器设计的实际硬件实验。

表 4 UIUC CS 硬件系列课程实验

课 程	主要实验
CS 231: Computer Architecture I (计算机体系结构I)	使用模拟软件Logic Works进行电路的模拟，实验内容主要有门电路、组合电路、时序电路、加法减法器、存储器等。
CS 232: Computer Architecture II (计算机体系结构II)	基于汇编的实验，分成5个Machine Project，使用MIPS2的指令集，在一个模拟系统上进行实验。
CS 433: Computer System Organization (计算机系统组成)	分析不同的处理器构架，完成一个报告。

表 5 Purdue ECE 硬件系列课程实验

课 程	主要实验
ECE 26700 - Digital Logic Design Laboratory (数字逻辑设计实验)	使用TTL电路进行实验。
ECE 36500 - Introduction To The Design Of Digital Computers (数字计算机设计导论)	使用软件模拟进行实验。
ECE 43700 - Computer Design And Prototyping (计算机原型机设计)	使用VHDL语言设计一个双核处理器，在FPGA上实现。

表 6 西北大学 EECS 硬件系列课程实验

课 程	主要实验
EECS 203 - Introduction to Computer Engineering (计算机工程导论)	使用面包板和TTL器件，实验内容如下： 1)验证两个简单电路。 2)设计一个简单的组合逻辑电路。 3)设计一个复杂的组合逻辑电路。 4)设计一个算术运算器。 5)汇编语言编程，控制单片机进行简单操作。 6)汇编语言编程，实现对一个机器人的控制。
EECS 303 - Advanced Digital Logic Design (数字逻辑设计)	使用ModelSim等软件进行仿真实验，使用VHDL语言，实验内容主要是软件使用，并实现一个状态机。
EECS 361 - Computer Architecture I (计算机体系结构I)	一个整学期的大实验，使用mentor的图形工具，设计一个单周期处理器，完成模拟。
EECS 362 - Computer Architecture Project (计算机体系结构实验)	一个整学期的大实验，使用VHDL设计一个流水线处理器，完成模拟。

从表 6 可以看出，西北大学的数字逻辑实验采用面包板进行实际电路上的实验，组成原理和系统结构则是采用软件模拟的方案进行处理器设计的实验，如果再深入一步，就是使用 FPGA 等可编程器件下载进行实际硬件实验了。

从以上这 3 个学校的硬件系列课实验内容来看，数字逻辑课程都是进行逻辑电路的实验，区别在于是采用软件模拟的方式还是使用实际电路进行实验，CS 偏向使用软件模拟，ECE 则更偏向在实际硬件上进行实验。组成原理和系统结构的实验差别就很大了，CS 偏重上层软件的实验，如汇编实验或者进行性能分析，而 ECE 更注重处理器设计，完成一个具有一定功能和结构的处理器并在硬件上实现。这样安排实验内容都是基于教学的需求，体现出了学科方向的不同要求。

3 实验安排

由于时间仓促，我们在此次行程中没有详细了解每门课程的具体实验内容，只是重点调研了几门课程。下面以 Purdue 的 ECE 43700 课程为例介绍实验的具体安排。

ECE43700 实验使用的教学语言是 VHDL，使用的教学软件有 ModelSim(VHDL 设计和软件仿真工具软件，ModelSim 公司产品)和 Quartus II(硬件开发和仿真工具软件，Altera 公司产品)，教学硬件是 Altera DE2 开发及教学电路板(包含 FPGA 芯片，Altera 公司产品)，辅助硬件是 Tektronix TLA 715 logic analyzer 逻辑分析仪(Tektronix 公司产品，操作系统 Windows 2000)。实验设备如图 1 和图 2 所示：

图 1　DE2 开发板

图 2　Tek 逻辑分析仪

具体实验安排如下：

1) 第 1 周：寄存器文件设计。设计 32×32 的寄存器文件，双读单写；5~32 位址解码器，读写控制。

2) 第 2 周：ALU 算术逻辑单元设计。设计实现 8 种逻辑算术功能的算术逻辑单元，8 种逻辑算术功能为 SLL、SRL、ADD、SUB、AND、NOR、OR、XOR。

3) 第 3、4 周：单周期处理器设计。集成前两个实验成果，设计单周期处理器，实现大部分 MIPS 指令集(I 类、R 类、J 类)软、硬件正确运行。

4) 第 5、6、7 周：流水线处理器设计。在前 3 个实验的基础上，设计 5 级流水线处理器，实现大部分 MIPS 指令集软硬件正确运行，控制单元，分支预测，转发和延迟功能等。

5) 第 8、9 周：高速缓存设计(包括指令缓存和数据缓存)。5 级流水线处理器集成缓存，实现 MIPS 大部分指令集软、硬件正确运行。

6) 第 10、11、12 周：双核处理器设计。设计缓存联控单元(使用有限状态机实现 MSI 算法)，控制双核 5 级流水线处理器，双指令缓存，双数据缓存，单一内存，实现 MIPS 大部分指令集软、硬件正确运行。

ECE43700 是大四的 1 门系统结构课程实验，实验内容为使用硬件设计语言设计和实现 1 个双核处理器，并在可编程器件 FPGA 上下载运行。在具体的实验安排上，实验被划分成若干个具体步骤，从部件设计到单周期处理器设计，再到流水线处理器设计，最后引入高速缓存，直至完成双核处理器的设计。整个过程中实验内容逐步深入，要求也十分明确，既便于学生循序渐进地开展实验，也便于控制进度。在实验工作量方面，学生需要完成单周期、流水线和双核三种处理器的设计，工作量十分饱满，能很好地提高学生硬件设计能力和实际工程能力。在实验环境和支撑方面，实验室提供了强大的实验开发平台，并配备了先进的实验辅助测量仪器，为学生提供了很好的实验保障，使实验能够顺利完成。

4　实验管理

在实验的具体管理方面，各个学校都投入了很多的人力和物力，以保障课程实验能够顺利开展。通过对各学校实验管理的调研，我们认为虽然他们各有特色，但也有以下几个共同点：

1) 同实验安排一样，为了控制实验进度，将整个实验细分成若干个步骤，并设置相应的检查点，规定各检查点应该完成的任务，学生需要在检查点之前完成相应的任务，教师或助教会根据学生完成情况安

排实验辅导等，这样就能够很好地控制整个实验的进度，实现目标化管理。在检查过程中，教师一般会安排学生作实验报告，讲解自己的进度、遇到的困难以及解决方案等，锻炼学生的表达能力。

2) 设置专门的实验课程或课程中安排专门的实验课时，实验课一般实行小班教学，这样教师可以充分了解每个学生的实验情况，及时解决他们遇到的具体问题。同时，教师还在课外安排相应的答疑时间，并且充分利用网络资源，通过 E-mail 或网上答疑等为学生提供帮助。

3) 由于实验的工作量比较大，学校会设置较多助教，尤其是综合实验，一般 20 名学生就会安排 1 名助教。对助教的要求也比较严格，他们不仅要批改作业，辅导实验，还要抽出时间在专门的答疑教室答疑，通过网络解答学生的问题，需要助教投入更多的时间和精力。

4) 有专门的实验室管理人员协助教师进行实验，保证实验时间。一般实验室的开放时间比较长，方便学生随时来作实验，尤其是软件模拟的实验，基本上保证机房 24 小时开放。对于硬件实验，大部分学校安排的自由实验课时都比较长，学生可以利用课余时间到实验室进行实验。

总结起来，各学校在实验管理上都做出了比较大的投入。虽然实验的工作量较大，但采取有效的管理机制，加之人员的配合，保证了实验能够顺利进行，学生能够完成实验任务，取得良好的教学效果。

5 小结

通过对美国中部 UIUC、伊利诺伊理工、芝加哥大学、西北大学和普度大学这 5 所大学硬件系列课程的调研，我们整理出他们的课程设置及实验内容，了解了他们具体的实验安排和管理办法。

对比国内的硬件系列课程，在课程设置和实验内容上已经逐渐接近我们考察学校的水平。但由于实验课时不足，造成实验任务较小，学生没有得到充分锻炼。在实验管理方面，国内投入的实验教学力量比较少，尤其是助教数量偏少，实验管理没有达到较高水平。在实验设备和环境上，国内各高校都已经开始使用硬件设计语言和可编程芯片进行实验，实验平台已经达到国外高校水平，但由于经费不足，因此实验辅助仪器还比较少。

通过此次考察，我们找到了国内硬件系列课程和实验与美国几所大学的异同点，以此来把握这些课程和实验的主流方向，了解自身的不足，不断改进教学和实验，切实提高教学质量和实验层次。

参考文献：

[1] UIUC CS. 2009 UIUC 计算机专业课程设置[EB/OL]. [2010-06-30]. http://courses.illinois.edu/cis/2009/fall/catalog/CS/.

[2] Purdue ECE. 课程设置[EB/OL]. [2010-06-30]. http://esa-oas-prod-wl.itap.purdue.edu/prod/bzwsrch.p_search_catalog?subject=ECE.

[3] Northwestern EECS. EECS 课程设置[EB/OL]. [2010-06-30]. http://www.eecs.northwestern.edu/academics/course/.

Investigation and Research on the Computer Hardware Series Course and Experiment in USA

LI Shan-shan, QUAN Cheng-bin

(Computer Education and Experiment Center, Tsinghua University, Beijing 100084, China)

Abstract: After the investigation and research on the computer hardware series course and experiment of typical colleges in the midland of USA, we summarize the course setting, experiment contents and experiment arrangement of the corresponding courses in these colleges. After illustrating the experiment organization using a detailed course experiment, we analyze these courses to find out the lack in teaching and experiment in order to improve the teaching quality and experiment level.

Key words: computer education; hardware series course; experiment organization; experiment arrangement

ISSN 1002-4956
CN11-2034/T
实验技术与管理
Experimental Technology and Management
第27卷 第4期 2010年4月
Vol. 27 No. 4 Apr. 2010

计算机技术应用

uC/OS-Ⅱ内核在基于FPGA的CPU上的移植

李山山[1]，李耀辉[1]，刘敬晗[2]，汤志忠[1]

(1. 清华大学 计算机教学实验中心，北京 100084；2. 清华大学 基础训练中心，北京 100084)

摘 要：介绍了基于FPGA(field programmable gate arry)的CPU MiniArm的指令系统、体系结构以及具体实现方法，讨论了将uC/OS-Ⅱ内核移植到MiniArm平台的关键技术，并详细描述了移植的具体实现过程，介绍了移植测试的方法并分析了测试结果，最后总结了移植操作系统到基于FPGA的CPU上的一般方法。

关键词：FPGA；CPU；uC/OS-Ⅱ；移植

中图分类号：TP311 **文献标志码**：B **文章编号**：1002-4956(2010)04-0087-04

Transplanting uC/OS-Ⅱ kernel to FPGA-based CPU MiniArm

LI Shanshan[1], LI Yaoqiang[1], LIU Jinghan[2], TANG Zhizhong[1]

(1. Laboratory for Computer Education, Tsinghua University, Beijing 100084, China; 2. The Fundamental Industrial Training Center, Tsinghua University, Beijing 100084, China)

Abstract: This article introduces FPGA-based CPU called MiniArm, mainly on its instruction set, structure and detailed implementation; discusses the key technology of transplanting uC/OS-Ⅱ kernel to MiniArm, and the detailed process of the implement is introduced; analyzes the test method and the result of the transplanting; and finally summarizes the general methods of transplanting operating system to FPGA-based CPU.

Key words: FPGA; CPU; uC/OS-Ⅱ; transplantation

近年来，基于FPGA(field programmable gate arry)的CPU设计得到了越来越广泛的重视和应用，但对一个完整计算机系统来说，单有硬件CPU是远远不足的，运行于该CPU之上的软件系统是不可或缺的，而操作系统是其中最关键的系统软件。实际应用中，采用移植手段获得可运行于基于FPGA的CPU上的操作系统，所花费的代价最小。本文简介了笔者开发的基于FPGA的CPU MiniArm的体系结构，并以之为移植目标，以uC/OS-Ⅱ为移植对象来详细讨论移植的过程，同时通过硬件仿真给出移植测试结果，最后分析了移植中要注意的一些问题，总结了移植的一般方法。

1 MiniArm CPU介绍

MiniArm是笔者开发的基于FPGA的32位CPU[1]，使用的FPGA芯片为Cyclon II EP2C20Q240C8。该CPU是一个5级流水线的结构；内部实现了独立的指令和数据Cache；实现了Arm7TDMI指令集[2]的一个子集，支持批量数据传送指令等复杂指令；支持USR，IRQ，SVC，SYS 4种运行模式。

1.1 指令系统

MiniArm实现的指令系统是根据uC/OS-Ⅱ内核代码分析后得到的最简指令集，该指令集可以支持uC/OS-Ⅱ内核运行所需的必要指令，支持Arm 7的所有寻址方式[3]，支持条件代码。具体指令如表1所示。

表1 MiniArm指令系统

指令类型	条数	指令名称
数据传送指令	1	mov
数据处理指令	8	and，sub等
转移指令	2	bl，bx
加载/存储指令	6	ldr，ldrb，ldrh等
批量数据传送指令	2	ldm，stm
状态寄存器指令	2	mrs，msr
空指令	1	nop
操作数移位操作	3	lsl，lsr，asr

收稿日期：2009-04-22
基金项目：国家自然科学基金资助(60173010)
作者简介：李山山(1979—)，男，山东省五莲县人，硕士，工程师，研究方向：计算机系统结构、计算机教学。

1.2 CPU总体结构

体系结构是根据经典的5级流水设计[4]的,支持Cache和指令的分支预测,总体结构如图1所示。从图中可以看出,MiniArm有如下特征:经典5级流水、支持分支预测、哈佛结构的缓存、支持指令乱序完成、采用状态机来协助流水线处理Arm7TDMI指令集中的复杂指令。

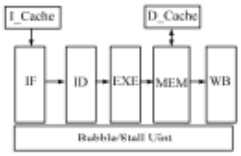


图1 MiniArm总体结构图

1.3 具体内部结构实现

1.3.1 流水段的主要功能

(1) 取指段IF。用来取出下一条指令,包括指令寄存器IP和指令缓存I-Cache。

(2) 译码段ID。用来根据指令生成各控制信号,并从寄存器中取出操作数。寄存器堆为19个32位的寄存器。

(3) 执行段EXE。执行算数和逻辑运算,运算器ALU支持8种基本运算,生成结果和产生符号位及状态标志位。

(4) 访存段MEM。用来读写数据缓存。

(5) 流水线写回段WB。用来改写寄存器的值。

1.3.2 气泡/阻塞控制单元Bubble/Stall Unit

这个部件是流水线控制模块,用于决定各个流水线寄存器的行为:通过、保持或清零,同时决定IP是否改变。具体工作方式如下:

(1) 当指令Cache缺失时,保持IP不变,当有跳转发生时,保持正在译码阶段的跳转指令,否则向译码阶段塞气泡,后面的指令继续运行;

(2) 当数据Cache缺失时,锁住整个流水线;

(3) 当成功跳转时,将刚刚取出的那条指令作废;

(4) 当处在执行阶段的ldr类指令与处在译码阶段的指令发生写后读相关时,IP和译码阶段的指令锁住,后面的指令向前执行,向执行阶段塞一个气泡。

(5) 当处在译码阶段的指令与处在访存阶段的ldr类指令发生写后读相关时,锁住前3级流水段,向访存阶段塞气泡。

1.3.3 指令/数据缓存I-Cache/D-Cache

指令Cache采用状态机模型,包括下面4个状态:ST_RESET0(初始的初始化状态),ST_RESET(正常的初始化状态),ST_NORMAL(通常的状态),ST_REPLACING(该状态下完成缺失Cache块的载入)。

数据Cache采用状态机模型,包括下面几个状态:ST_RESET0(初始的初始化状态),ST_RESET(正常的初始化状态),ST_NORMAL(通常的状态),ST_WRITEBACK1(将Cache数据写回主存的第1周期),ST_WRITEBACK2(将Cache数据写回主存的第2周期),ST_REPLACING(该状态下完成缺失Cache块的载入)。

2 移植uC/OS-II

2.1 uC/OS-II简介及可行性分析

uC/OS-II是一个代码完全公开的、微型的、完全占先式的实时操作系统,具有很强的可移植性。它具有完全可剥夺型的实时内核,其核心工作原理是让最高优先级的就绪任务处于运行状态,具有多任务的特点,可以管理64个任务,其中56个任务分配给用户;具有内核可裁减性,可确定性的特点,并提供很多系统服务,比如信号量、互斥信号量、消息队列、内存的分配和释放等[5]。

它的源码完全公开,可以很方便地移植到Arm系列CPU上。uC/OS-II在windows环境下就有优秀的编译器Arm Development Suite和调试器AXD与之相配,具有可固化性,uC/OS-Ⅱ是为嵌入式应用而设计的,这就意味着,只要有固化手段就可以方便地下载到Rom或者FPGA的片上内存里去;系统架构简单,纯内核编译之后大小只有15kB,可以存储在FPGA的片上内存里。基于以上几点考虑,选择uC/OS-II作为移植的系统,本文的uC/OS-II初始版本是可以运行在Intel XScale PX255嵌入式系统[6-7]上的版本。

2.2 裁减uC/OS-II内核

移植中采用Arm Developer Suite v1.2编译器(ADS),集成环境是CodeWarrior;调试器是ARM eXtended Debugger(AXD)。在ADS环境下移植uC/OS-II可直接用AXD进行软件仿真,用硬件仿真软件ModelSim进行硬件层次的仿真,然后再进行硬件实际调试运行。

uC/OS-II内核可以分为处理器无关代码、处理器相关代码以及应用相关代码3个部分。移植中只需修改与处理器相关部分的文件,下面简要介绍一下主要修改的部分:

(1) 裁减驱动函数。uC/OS-II中有不少函数是设备驱动程序,而MiniArm没有实现这些设备,可以裁减掉该部分设备驱动程序的调用[6]。

(2) CPU运行模式调整。MiniArm只实现了USR,SVC,SYS,IRQ等模式的寄存器,所以应该在main.s中裁减UND,ABT,FIQ等模式及其堆栈的初

始化。

(3) 内存地址分配设置。因为MiniArm没有地址转换器,所以必须重新设置uC/OS-II的内存分配,使得它和MiniArm的内存接口相一致。在main.inc里重新设置Rom,Ram和堆栈的大小以及起始地址,减小全局变量的最大个数:

```
ROM_Base    EQU 0x00000000 ;ROM
ROM_Size    EQU 0x00004000 ;RAM
RAM_Base    EQU 0x00004000 ;全局变量空间
Globe_Variable_Size EQU 15 *1024
STACK_SIZE EQU 1024 *1
RAM_Source_Base EQU
RAM_Base + Globe_Variable_Size + STACK_SIZE
STACK_LOCATION EQU RAM_Source_Base
```

初始栈顶STACK_LOCATION = RAM_Base + Globe_Variable_Size + STACK_SIZE = 0x8000,而SVC态的堆栈栈顶为STACK_LOCATION - 2 * STACK_SIZE = 0x7800

重新设置Ram的起始地址:

```
ER_data_plus_bss 0x00004000
```

3 验证及其结果分析

验证过程的主要指导思想是测试操作系统是否能够正确地启动并调度到测试程序,并且运行过程中内部寄存器内容正确。下面是移植测试实验以及对实验结果的分析。

3.1 死循环任务测试

(1) 测试程序。uC/OS-II是基于优先级调度的抢占式的实时操作系统,为了方便测试,在CEntry.c中编写一个最简单的测试程序(死循环),以测试操作系统是否能够正确地启动并调度到测试程序。

```
void task2(void *pd){
    for(;;) {}
}
```

在main函数中建立该任务的线程,然后调用OSStart函数启动操作系统,如果程序运行正确,则操作系统最后会进入到task2的死循环;而且寄存器r9应该是RAM的起始地址0x4000。由于只有一个死循环的任务,所以SVC态堆栈中只有main函数的返回值,应该为0x77f8,其他寄存器的值为0。

用ADS编译,生成uCOS_II_DeadCircle.bin文件,大小为15k。在AXD中,选择CPU为ARM7TDMI,模拟运行uCOS_II_DeadCircle.bin。程序可以运行到任务Task2的死循环。此时cpsr为0x00000013,运行在svc态。R13_svc是0x77F8.

(2) ModelSim硬件仿真。在Modelsim运行MiniArm并加载进行仿真,得到如图2所示。

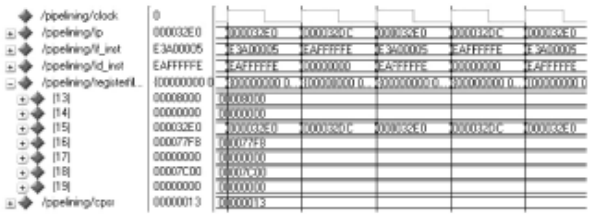


图2 死循环任务测试仿真波形

图中R13,R14,R16,R17,R18,R19分别对应R13_usr,R14_usr,R13_svc,R14_svc,R13_irq,R14_irq。

当IP指向0x32dc的时候,取下一条指令[0xeafffffe],即b task2,此时由于发生跳转,ID段冒泡,执行空指令;当IP指向0x32e0的时候,取下一条指令[0xe3a00005],此时ID段执行指令[0xeafffffe]。程序进入b task2的死循环中。

对比软件模拟和硬件仿真的结果,两者最后都进入task2的死循环中,CPSR都是0x00000013,运行在svc态,对应的svc态的通用寄存器的值也相等。所以,ModelSim仿真中Task2创建成功,运行正常。

由此向前分析前面的代码,比较模拟和仿真的结果,可以得到OStaskstkInit,OSTaskCreate,OSStart等函数工作正常,uC/OS-II内核在MiniArm上运行正确。

3.2 调度任务测试

3.2.1 测试程序

创建2个任务,它们相互调度。如果系统正常运行,将会在2个任务之间轮流运行。

```
void YouTask(void *pd){
    for(;;){
```

```
        if(timecount == 0) OSTaskResume(0);
        timecount --;
    }
}
void MyTask(void *pd){
    OSTaskCreate(YouTask,(void *)0,&YouTaskStk
[TASK_STK_SIZE - 1],2);
    for(;;){
        if(timecount == 3)
            OSTaskSuspend(OS_PRIO_SELF);
        timecount ++;
    };
}
```

编译内核,启动 AXD,可以观察到任务 YouTask 和 MyTask 相互调度。

3.2.2 ModelSim 硬件仿真

在 Modelsim 运行 MiniArm 并加载 uCOS_II_Dispatcher.hex 进行仿真,结果如图 3 所示。

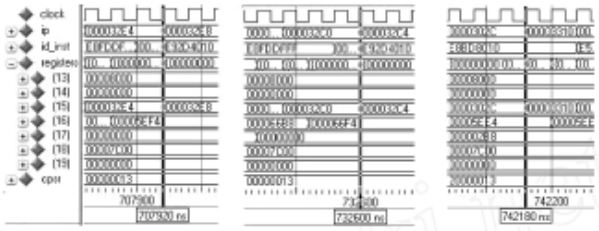


图 3 调度任务测试仿真波形

MiniArm 在 707920ns 第 1 次运行到 MyTask,然后继续并发生调度,在 732600ns 第 1 次进入 YouTask,继续再次发生调度,在 742180ns 回到 MyTask。从此,这 2 个任务轮流运行。整个运行结果与理论结果,模拟运行结果都完全一致。

比较模拟和仿真的结果,可以得到 OSTaskSuspend,OSTaskResume,OS_Sched 等函数工作正常,内核在 MiniArm 上运行正确,移植基本成功。

3.3 硬件测试

仿真通过后,使用了 Altera 公司的 FPGA 芯片 EP2C20 作为测试芯片,将 MiniArm 下载到 FPGA 芯片中,将 uC/OS-II 装载到 RAM 中,形成一个简单的硬件系统,配合外围电路进行测试,测试结果与仿真结果相同。

4 后续工作

为了更好地验证移植的正确性,加入简单的文件系统;另外,可在 MiniArm 上加入简单的外围设备,如串口、输入输出设备等。更重要的工作是虽然目前的移植的 uC/OS-II 已经可以在 FPGA 芯片上运行,但是需要做进一步的工作,完成一块开发板,使之成为一个完整的嵌入式系统。

5 结束语

本文介绍了基于 FPGA 的 CPU MiniArm 的指令系统、体系结构和实现方法,并介绍了 uC/os-II 移植到 MiniArm 的全过程,以及测试方法及结果。对 CPU 的设计和嵌入式实时操作系统的移植有一定的参考价值。总结来说,应该尽量选择通用指令集的一个子集作为 CPU 的指令集,以充分利用现有编译器,方便移植操作系统;接着,结合 CPU 的硬件资源,选择合适的操作系统,使得两者的特性相配;然后,对移植对象中与硬件相关的代码进行修改,加入测试程序,编译内核;最后用仿真软件对 CPU 和操作系统内核自身进行硬件仿真并在 FPGA 芯片上进行测试。

参考文献 (References):

[1] 李山山,汤志忠,周继群. 基于 FPGA 的开放式教学 CPU 的设计与测试系统[J]. 计算机工程与应用,2005(14):98-100.
[2] 杜春雷. ARM 体系结构与编程[M]. 北京:清华大学出版社,2003.
[3] 李佳. ARM 系列处理器应用技术完全手册[M]. 北京:人民邮电出版社,2006.
[4] John L. Hennessy, David A. Patterson. 计算机体系结构—量化研究方法[M]. 4 版. 白跃彬,译. 北京:电子工业出版社,2007.
[5] Labrosse JJ. uC/OS-II--源码公开的实时嵌入式操作系统[M]. 邵贝贝,译. 北京:北京航空航天大学出版社,2003.
[6] 陈章龙,唐志强,徐时亮. 嵌入式技术与系统—Intel XScale 结构开发[M]. 北京:北京航空航天大学出版社,2004.
[7] Raj Kamil. 嵌入式系统结构——体系结构、编程与设计[M]. 陈曙晖,译. 北京:清华大学出版社,2005.

文章编号：1672-5913(2010)15-0007-09 中图分类号：G642 文献标识码：B

美国计算机专业课程体系的调研报告

全成斌，李山山
(清华大学 计算机实验教学中心，北京 100084)

摘 要：在调研了美国中部有代表性的大学后，整理出这些学校计算机专业本科生课程体系建设的共同理念，描述了分层次的课程体系，重点阐述了核心课程的设置情况。通过分析具有代表性的学校的课程体系，以掌握本科教学和实验教学的主流方向，提高教学质量和实验管理能力。

关键词：计算机专业；课程体系；核心课程；教学

与 UIUC 教师合影(左一为赵有健老师，右一为本文作者全成斌)

为了切实提高计算机学科实验教学水平，在教育部的支持下，国家级实验教学示范中心计算机学科组考察了美国中部多所院校，重点学习了各层次大学计算机科学(CS)和计算机工程(CE/ECE)本科专业的课程设置、实验教学体系、实验环境建设以及实验内容开发情况，本文重点报告计算机专业课程体系设置的调研情况。

1 课程设置理念

每所大学都有自身的特点，但是通过总结不难发现，美国主要高校在计算机专业课程设置的理念上却大同小异，即以学生为本、兴趣培养为基础，在宽厚的理论和实践功底上，将学生培养成为计算机特定方向上的专业人才。

在课程体系建设时，各所大学首先坚持以学生为本。美国高校有公立和私立之分，以排名情况为参考，在相同层次的学校中，私立学校的本科教学质量较高，公立学校的本科学生数量很多，在教学质量上就略逊一筹。但是在课程建设时，各学校都是充分考虑为学生服务的，而私立学校学费高，考虑得就更周到一些。从低年级到高年级，课程由浅入深、层次清晰。在大多数学校，学生入学时可以多专业任选，每个专业都会将本学科的特点和作用以最浅显的方式展现给学生。换句话说，如果学生选择了计算机学科，就意味着他是对这个学科有浓厚兴趣的，整个培养课程体系都是来满足其兴趣要求的。当然，课程设置会充分考虑学生的能力，学分要求不是很高，课程的数量尤其是必修课程

数量不多，核心课程最多不超过 10 门，更加强调专业基础[1-2]，这一点将在核心课程设置部分详细阐述。

其次，在强调宽厚基础的同时，课程设置注重理论与实践相结合。在低年级课程中，几乎所有大学都开设了交流与写作课程(详细内容参见附录 1)，几乎所有课程中都有各种形式的实验，实验大多是以项目(Project)来设立的，每门课的项目数量不多，内容由简入难，使学生将所学知识逐步深入运用起来。理论课程与实验课程是紧密结合的，学校根据课程的特点合理分配理论和实验课时，让学生通过实践彻底领会理论的实质和内涵。

大学是培养人才的摇篮，不同层次的院校对自身所培养的人才有不同的定位。在课程体系建立时，各所学校都依据自身情况设定不同的人才培养方向。学生有较大的深入学习和发展空间，为成为有用人才打下坚实基础。广度和深度的学科方向设定情况将在下文详细说明。

2 课程体系

各大学基本上按层次划分课程，同时采用了和我们相似的必修课\限选课\任选课方法。但美国必修课的数量大大压缩，主要核心课程一般控制在 10 门以内，其他课程被分成若干个方向(Track 或 Breadth 和 Depth)，学生根据需要选择 1～2 个方向来学习。

综合各学校课程体系，我们基本能够得到类似图 1 所示的课程层次结构图。

其中的基础课程主要是交流(Communication)与表达(Presentation)或者类似课程。广度(Breadth)和深度(Depth)课程是联系紧密的，是深入学习的不同方向，深度课程是在广度课程基础上的深化和加强，图 1 所列方向以西北大学的课程体系为主(各学校方向设定方式和具体课程设置情况参见附录 2)。其他学校的方向略有不同，本文主要关注课程体系设置的方式及其核心课程设定，这里不一一列举各所学校的详细方向设定，核心课程的设定情况参见下节。

3 核心课程

各个学校在核心课程上很相似，数量也不多。有些学校的计算机专业和电子计算机工程的核心课程差别较大(例如 UIUC)，我们就分别列举出来，如果差别不很大，我们就选择更具有代表性的设置方式，具体课程如表 1 所示。

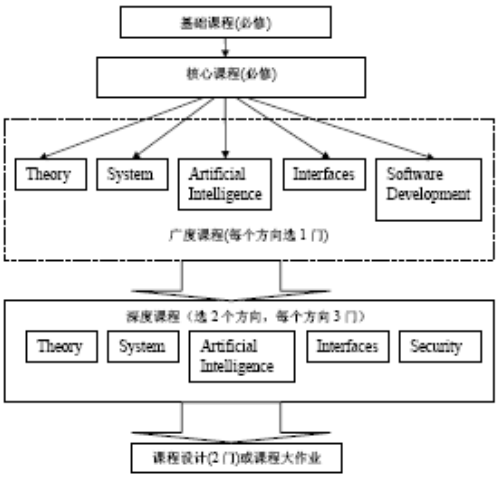


图 1 课程体系结构图

表 1 核心课程列表

学 校	课 程
UIUC(CS)[1-3]	CS 125 Intro to Computer Science CS 173 Discrete Structures CS 210 Ethical and Professional Issues in CS CS 225 Data Structure and Software Principles CS 231 Computer Architecture I CS 232 Computer Architecture II CS 241 System Programming CS 242 Programming Studio CS 373 (was CS 273) Theory of Computation
UIUC(ECE)	ECE 110 - Introduction to Electrical & Computer Engineering ECE 190 - Intro to Computing Systems ECE 210 - Analog Signal Processing ECE 290 - Computer Engineering I ECE 329 - Introduction to Electromagnetic Fields ECE 385 - Digital Systems Laboratory ECE 391 - Computer Systems Engineering ECE 411 - Computer Organization & Design ECE 440 - Solid State Electronic Devices CS 225 - Data Structure & Software Principles
普度大学[3]	CS 18000 Programming I CS 18200 Foundations of Computer Science CS 24000 Programming in C CS 25000 Computer Architecture CS 25100 Data Structures CS 35200 Compilers: Principles and Practice CS 35400 Operating Systems CS 38100 Introduction to the Analysis of Algorithms
西北大学[4-5]	EECS 101 – An Introduction to Computer Science For Everyone EECS 111 – Fundamentals of Computer Programming I EECS 211 – Fundamentals of Computer Programming II EECS 213 – Introduction to Computer Systems EECS 310 – Discrete Mathematics EECS 311 – Introduction to Data Structures
IIT	CS100 Introduction to the Profession CS115 Object-Oriented Programming I CS116 Object-Oriented Programming II CS330 Discrete Structures CS331 Data Structure and Algorithms CS350 Computer Organization /Assembly Language Programming CS351 System Programming CS430 Induction to Algorithm CS440 Programming Language/Translators CS485 Computer in Society
芝加哥大学[3-5]	CMSC 15100(Introduction to Computer Science I) or 16100(Honors Introduction to Computer Science I), and CMSC 15200(Introduction to Computer Science II) or 16200(Honors Introduction to Computer Science II) CMSC 15300(Foundations of Software) CMSC 15400(Introduction to Computer Systems) CMSC 27100(Discrete Mathematics) CMSC 27200(Theory of Algorithms)

综合各学校核心课程设置情况不难发现，目前国内的很多核心课程，在这些学校已经变成了学生的限选或选修课，例如编译原理、操作系统、人工智能和网络原理等，作为可选的方向变成了方向性的课组，学生可以根据爱好选择一类同方向的课程。核心课程的设置具有以下特点：

1) 包括一门计算机入门课程，例如 UIUC 的 CS125、西北大学的 EECS 101、普度大学的 CS 18200、IIT 的 CS100、芝加哥大学的 CMSC 15100 或 16100。这些课程整体地而不是深入到某一方向介绍计算机各个领域的知识，让学生了解并产生兴趣，以便他们喜欢计算机专业，以后能够根据自己的爱好选择方向。

2) 包括一门必需的数学基础课程，专业必需的数学课程设定了离散数学，其他如随机数学、图论等，也作为特定学科方向的方向性课组选择开设。

3) 包括至少一门编程课。这类课程一般安排在较早的学期，让学生尽早掌握编程技巧，提高用计算机解决实际问题的能力。目前来看，第 1 门编程课多

数是 C 语言，也有个别学校一开始使用 Pathon，后来转换成 C，或者直接用 Java 作为入门编程语言。

4) 各个学校无一例外将数据结构作为一门核心课，有些学校的 CS 会在数据结构课上安排一些算法内容。

5) 计算机组成和体系结构也被安排进了核心课，虽然各个学校的课程名称不尽相同。该课主要介绍计算机的组成，阐述它是如何工作的，CS 偏向介绍系统如何工作、能做什么，而 ECE 偏向结构组成，这些在实验上也有充分体现。我们将在下一篇文章中专门讨论，这里不赘述。

在课程体系设立上，各个学校可谓大同小异，但是“小异”之处大多体现着学校特点。在核心课程设定中，有 3 所学校将算法类课程设为核心课程，也有个别学校将操作系统设为核心课程。

4 反思

通过对美国中部 UIUC、伊利诺伊理工、芝加哥大学、西北大学和普度大学这 5 所大学的调研，我们整理出了这 5 所学校本科生课程体系建设的共同理念，描述了分层次的课程体系，重点阐述了核心课程的设置情况。相比之下，目前国内计算机专业的课程体系设立的核心课程数量过多，导致学生压力大，不能集中精力深入学习，影响教学质量。此外，我们缺少对学生学习兴趣的引导，入门课程的学科基础普及教学没有得到足够重视，在对本科生在学科方向上的引导和开放学生的自由选择空间方面也有欠缺，这些问题都值得从事计算机专业教学的同行们深思。

参考文献：

[1] UIUC 计算机系. 2009 UIUC 计算机专业课程设置[EB/OL]. [2010-02-20]. http://courses.illinois.edu/cis/2009/fall/catalog/CS/.

[2] UIUC 计算机系. UIUC 计算机本科课程设置[EB/OL]. [2010-02-20]. http://cs.illinois.edu/undergraduates/academics.

[3] Purdue 计算机系. Purdue's Undergraduate Curriculum[EB/OL]. [2010-02-20]. http://www.cs.purdue.edu/academic_programs/undergraduate/curriculum/bachelor/.

[4] Northwestern EECS 系. EECS Courses Introduction[EB/OL]. [2010-02-20]. http://www.eecs.northwestern.edu/academics/course/.

[5] Northwestern EECS 系. EECS213 Intro to Computer Systems[EB/OL]. [2010-02-20]. http://www.cs.northwestern.edu/~pdinda/icsclass/.

[6] Northwestern EECS 系. EECS 340 Intro to Networking[EB/OL]. [2010-02-20]. http://www.cs.northwestern.edu/~ychen/classes/cs340-w08/.

[7] Northwestern EECS 系. EECS 343 Intro to OS[EB/OL]. [2010-02-20]. http://www.aqualab.cs.northwestern.edu/classes/eecs-343-f09/.

[8] Northwestern EECS 系. EECS311 Data Structures[EB/OL]. [2010-02-20]. http://www.cs.northwestern.edu/academics/courses/311/.

[9] Northwestern EECS 系. EECS 350 Intro to Computer Security[EB/OL]. [2010-02-20]. http://www.cs.northwestern.edu/~ychen/classes/cs350-w07/.

[10] 芝加哥大学计算机系. UChicago 计算机专业课程设置[EB/OL]. [2010-02-20]. http://www.cs.uchicago.edu/maps.

Survey of Computer Major Curriculum System of USA

QUAN Cheng-bin, LI Shan-shan
(Computer Education and Experiment Center, Tsinghua University, Beijing 100084, China)

Abstract: We describe the computer major curriculum system and the curriculum construction idea after the survey of typical colleges in the midland of USA. According to the analysis of the curriculum system, we want to find out the main direction on teaching and experiment in order to improve the teaching quality and experiment level.

Key words: computer major; curriculum system; key course; teaching

附录 1：写作和演讲课程

除了核心课程是必修外，大多数学校安排了演讲或写作类课程为必修课，通过这些课程培养学生的交流和写作能力，并在以后的课程中使用。表 1 是这 5 所大学的演讲或写作类课程。

表 1 5 所大学的演讲或写作类课程[1-4][10]

学 校	课 程
UIUC(CS)	Composition I Advanced Composition - can be completed by choosing one of the following options - see technical track requirements below for details: CS 427 and CS 429; CS 492 and CS 493; CS 499
UIUC(ECE)	RHET 105 - Principles of Composition
西北大学	Communication Skills
普度大学	ENGL 106, 108 Tech Writing/Presenting or Tech Writing Tech Presenting
IIT	COM421 or COM428 Technical Communication or Verbal and Visual Communication

附录 2：课程方向

在选修课程上，各个学校根据自身情况将课程分成了不同方向(Track)。UIUC 计算机专业(CS)具有代表性，而西北大学刚刚调整好课程体系，整个课程层次明晰，因此这里主要介绍这两所学校的课程方向。

1 UIUC(CS)选修课程

UIUC(CS)的选修课程分为如下三个方向：

1) CS Track。

2) Computational Science and Engineering (CSE) Track。

3) Math Track。

1.1 CS Track(CS 方向)

表 2 是 UIUC 的 CS 方向的课程设置及相应学时。

其中在 Specialization 中，根据不同方向又细分为以下几个方向课程，学生必须选择其中 1 个方向的课程[1-2]：

表 2 UIUC 的 CS 方向课程设置及学时[1-2]

学 时	课程设置
3	CS 357 (was CS 257) Numerical Methods I
3	CS 421 Programming Languages and Compilers
3	CS 473 Algorithms
6	Specialization (select one from the list below)
6	Two additional CS 400-level courses numbered 410-489 or 498
3～6	One of the following thesis/project options: CS 499 Senior Thesis, CS 492 Senior Project in CS I, and either CS 493 Senior Project in CS II ACP, or CS 494 Senior Project in CS III, CS 427 Software Engineering I, and either CS 428 Software Engineering II or CS 429 Software Engineering II, ACP (Note: CS 429 is identical to CS 428 with an additional writing component. Likewise, CS 493 is identical to CS 494 with an additional writing component.)

1) Systems。

(1) CS 423 Operating Systems Design。

(2) CS 431 Embedded Systems or CS 433 Computer System Organization。

2) Databases (任选 2 门)。

(1) CS 410 Intro to Text Info Systems (3 hours)。

(2) CS 411 Database Systems。

(3) CS 412 Intro to Data Mining (3 hours)。

3) Graphics (任选 2 门)

(1) CS 414 Multimedia Systems。

(2) CS 418 Computer Graphics I。

(3) CS 419 Production Computer Graphics。

4) Human-Computer Interaction。

(1) CS 465 User Interface Design (required)。

(2) 再任选 1 门。

① CS 498 Special Topics in CS, section KK Social Computing。

② CS 498 Special Topics in CS, section KK Social Visualization。

5) Languages。

(1) CS 422 Programming Language Design。

(2) CS 426 Compiler Construction。

6) Artificial Intelligence (任选 2 门)。

(1) CS 440 Artificial Intelligence。

(2) CS 443 Introduction to Robotics。

(3) CS 446 Machine Learning。

(4) CS 412 Intro to Data Mining。

7) Security (任选 2 门)。

(1) CS 461 Computer Security I。

(2) CS 463 Computer Security II。

(3) CS 460 Security Laboratory。

8) Networking (任选 2 门)。

(1) CS 438 Communication Networks(required)。

(2) CS 425 Distributed Systems or CS 439 Wireless Networks。

1.2 CSE Track(CSE 方向)

表 3 是 UIUC 的 CSE 方向的课程设置及相应学时。

表 3 UIUC 的 CSE 方向课程设置相应学时[1-2]

学 时	课程设置
3	CS 421 Programming Languages and Compilers
3	CS 473 Algorithms
3	Math 441 Differential Equations
3	CS 357 Numerical Methods I
3	CS 457 Numerical Methods II or CS 450 Numerical Analysis
9	Scientific Concentration (See list below)
3	CS 499 Senior Thesis

与 CS 类似，学生在专业方向(Scientific Concentration)中要选择 1 门课程，内容如下[1-2]：

1) Aerospace Engineering: AE 201 and 252, plus either AE 311 and 312, or AE 352 and 353。

2) Applied Mathematics:Any three of MATH 442, 446, 481, 488, or 489。

3) Astronomy:ASTR 210 plus any two of ASTR 350, 404, 405, 406, or 414。

4) Atmospheric Sciences:ATMS 300 plus any two of ATMS 401, 402, 403, or 410。

5) Biology:IB 150 and MCB 150 plus either IB 204 or MCB 250。

6) Biomedical Instrumentation:ECE 210, 414, and 415。

7) Biomolecular Engineering:Any three of CHBE 471, 472, 473, 474。

8) Chemical Engineering:CHBE 221, 321, and 421。

9) Chemistry: CHEM 104/105, 222 and 223, and 232。

10) Control:GE 320 plus any two of GE 420, GE 424, or ECE 486。

11) Electrical Engineering:ECE 329 plus any two of ECE 440, 441, 442, 450, or 452。

12) Engineering Mechanics:TAM 210 or 211, TAM 212, and any one of TAM 251, 335, or 470。

13) Environmental Engineering:CEE 330 plus any two of CEE 434, 437, 442, 443, or 444。

14) Genetics:ANSC 340, 441, and 446 or 447; or MCB 250, 418, and 421。

15) Geology:GEOL 107, 108, and any one of GEOL 401, 411, 432, 440, or 452。

16) Manufacturing Engineering:MFGE 310 plus any two of MFGE 420, 430, or 450。

17) Materials Science:MSE 280 plus any two of MSE 304, 401, 402, 420, or 450。

18) Mechanical Engineering:Any three of ME 300, 310, 320, 330, 340, 370, or 371。

19) Modeling and Simulation:Any three of ANSC 448, ECE 475, GEOG 468, MSE 482, or MSE 485。

20) Neuroscience:MCB 414 or PSYC 404 plus any two of MCB 412, 415, 416, 417, 419, PSYC 414, 415。

21) Nuclear Engineering:NPRE 247 plus any two of NPRE 402, 412, or 455。

22) Operations Research:IE 310 and any two of IE 410/CS 481, IE 411, IE 412, IE 413/CS 482。

23) Optimization:Any three of ECE 490, IE 411, MATH 482, or MATH 484。

24) Physics:PHYS 325 plus any two of PHYS 326, 402, 427, 435, 436, 460, or 485。

25) Plasma Engineering:ECE 329 or PHYS 435 plus NPRE 421 and 429。

26) Psychology:Any three of PSYC 210, 224, 248, 321, 358, 414。

27) Radiological Engineering:NPRE 446 plus any two of NPRE 435, 441, or 447。

28) Robotics:GE 320, 421, and 422。

29) Signal and Image Processing:ECE 210 plus any two of ECE 280, 410, 418, or 480。

30) Statistics:STAT 410 plus any two of STAT 420, 424, 425, 426, 428, or 429。

31) Structural Engineering:CEE 360 plus any two of CEE 470, 471, or 472。

1.3 Math Track (Math 方向)

表 4 是 UIUC 的 Math 方向的课程设置及相应学时。

表 4　UIUC 的 Math 方向课程设置及学时[1-2]

学　时	课程设置
3	CS 421 Programming Languages and Compilers
3	CS 473 Algorithms
3	MATH 441 Differential Equations
6/3	CS 357 Numerical Methods I and CS 457 Numerical Methods II, both, or CS 450 Numerical Analysis
3	CS 475 Formal Models of Computation OR one of the following: MATH 413, MATH 414, MATH 417, MATH 432, MATH 453, MATH 482
9	Three additional 400-level Math courses
3	CS 499 Senior Thesis

2　西北大学课程层次

西北大学将课程分为 5 个部分，分别是基础课程(Background)、核心课程(Core)、广度课程(Breadth)、深度课程(Depth)和课程设计(Project)。

2.1　基础课程

基础课程为基本要求的课程，主要是数学基础课等。有以下几个方面的课程[4]：

1) Continuous Mathematics。

2) Probability and Statistics。

3) Physical and Life Sciences。

4) Social Sciences and the Humanities。

5) Communication Skills。

对于从未编过程序的学生，需要选择下列课程中的 1 门：

1) EECS 110 (C) – An Introduction to Programming for Non-majors using the C。

2) EECS 110 (Python)–An Introduction to Programming for Non-majors using the Python。

2.2　核心课程

以下课程每个计算机专业学生必修[4]：

1) EECS 101 – An Introduction to Computer Science For Everyone。

2) EECS 111 – Fundamentals of Computer Programming I。

3) EECS 211 – Fundamentals of Computer Programming II。

4) EECS 213 – Introduction to Computer Systems。

5) EECS 310 – Discrete Mathematics。

6) EECS 311 – Introduction to Data Structures。

2.3　广度课程

这类课程包含了计算机专业的各个方向，学生应大体知道，不需要深入链接，因此他们可以在每个方向上选择 1 门课程[4]：

2.3.1　Theory

1) EECS 336 – Design and Analysis of Algorithms。

2) MATH 374 – Theory of Computability and Turing Machines。

3) EECS 328 – Numerical Methods。

4) EECS 335 – Introduction to the Theory of Computation。

5) EECS 356 – Formal Specification and Verification。

2.3.2 Systems

1) EECS 322 – Compiler Construction。

2) EECS 339 – Introduction to Databases。

3) EECS 340 – Introduction to Networking。

4) EECS 343 – Operating Systems。

5) EECS 303 – Digital Logic Design。

6) EECS 345 – Distributed Systems。

7) EECS 346 – Microprocessor Systems Design。

8) EECS 350 – Introduction to Security。

9) EECS 358 – Parallel Systems。

10) EECS 361 – Computer Architecture。

11) EECS 397 – Real-time Systems。

12) EECS 440 – Advanced Networking。

13) EECS 441 – Resource Virtualization。

14) EECS 442 – Dynamic Behavior of Applications, Hosts, and Networks。

15) EECS 443 – Advanced Operating Systems。

16) EECS 450 – Internet Security。

17) EECS 464 – Advanced Databases。

2.3.3 Artificial Intelligence

1) EECS 325 – AI Programming。

2) EECS 337 – Semantic Information Processing。

3) EECS 344 – Design of Computer Problem Solvers。

4) EECS 348 – Introduction to AI。

5) EECS 349 – Machine Learning。

6) EECS 360 – Models with Multi-agent Languages。

7) EECS 395/495 – AI For Interactive Entertainment。

8) EECS 395/495 – Knowledge Representation。

9) EECS 395/495 – Behavior-based Robotics。

2.3.4 Interfaces

1) EECS 330 – Human-Computer Interaction。

2) EECS 351 – Introduction to Computer Graphics。

3) EECS 352 – Machine Perception of Music。

4) EECS 370 – Computer Game Design。

5) EECS 332 – Digital Image Analysis。

6) EECS 395 – Intermediate Computer Graphics。

7) EECS 395 – Advanced Computer Graphics。

8) EECS 395/495 – Computer Animation。

9) EECS 395/495 – Graphics and Perception。

10) EECS 395/495 – Image-based Modeling and Rendering。

11) EECS 395/495 – Human-centered Product Design。

2.3.5 Software Development

1) EECS 338 – Practicum in Intelligent Information Systems。

2) EECS 394-1,2 – Software Project Management。

3) EECS 395 – RTFM courses。

2.4 深度课程

学生同时应该在两个方向深入学习，每个方向至少选择 3 门课程：

2.4.1 Theory

1) EECS 336 – Design and Analysis of Algorithms。

2) MATH 308 – Graph Theory *(added for 2008-09)*。

3) MATH 374 – Theory of Computability and Turing Machines。

4) EECS 395/495 – Current Topics in Algorithms。

5) EECS 457 – Advanced Algorithms。

6) EECS 328 – Numerical Methods。

7) EECS 335 – Introduction to the Theory of Computation。

8) EECS 356 – Formal Specification and Verification。

9) EECS 357 – Introduction to VLSI CAD。

10) EECS 395/495 – Algorithms for Bioinformatics。

11) EECS 395/495 – Algorithmic Research for e-Commerce。

12) EECS 459 – VLSI Algorithmics。

13) EECS 399 – Independent Study。

2.4.2 Systems

1) EECS 322 – Compiler Construction。

2) EECS 339 – Introduction to Databases。

3) EECS 340 – Introduction to Networking。

4) EECS 343 – Operating Systems。

5) EECS 361 – Computer Architecture。

6) EECS 345 – Distributed Systems。

7) EECS 350 – Introduction to Security。

8) EECS 354 – Network Penetration & Security。

9) EECS 358 – Parallel Systems。

10) EECS 395 – Appropriate Selected Topics - with advisor approval。

11) EECS 440 – Advanced Networking。

12) EECS 441 – Resource Virtualization。

13) EECS 442 – Dynamic Behavior of Applications, Hosts, and Networks。

14) EECS 443 – Advanced Operating Systems。

15) EECS 450 – Internet Security。

16) EECS 464 – Advanced Databases。

17) EECS 399 – Independent Study。

2.4.3 Artificial Intelligence

1) EECS 325 – AI Programming。

2) EECS 337 – Semantic Information Processing。

3) EECS 344 – Design of Computer Problem Solvers。

4) EECS 348 – Introduction to AI。

5) EECS 349 – Machine Learning。

6) EECS 360 – Models with Multi-agent Languages。

7) EECS 395/495 – AI For Interactive Entertainment。

8) EECS 395/495 – Knowledge Representation。

9) EECS 395/495 – Behavior-based Robotics。

10) EECS 399 – Independent Study。

2.4.4 Interfaces

1) EECS 330 – Human-Computer Interaction。

2) EECS 351 – Introduction to Computer Graphics。

3) EECS 352 – Machine Perception of Music。

4) EECS 370 – Computer Game Design。

5) EECS 332 – Digital Image Analysis。

6) EECS 395 – Intermediate Computer Graphics。

7) EECS 395 – Advanced Computer Graphics。

8) EECS 395/495 – Computer Animation。

9) EECS 395/495 – Graphics and Perception。

10) EECS 395/495 – Image-based Modeling and Rendering。

11) EECS 395/495 – Human-centered Product Design。

12) EECS 399 – Independent Study。

2.4.5 Security

1) EECS 350 – Introduction to Computer Security。

2) EECS 450 – Internet Security。

3) EECS 510-4 Computer Security and Information Assurance。

4) EECS 322 – Compiler Construction。

5) EECS 339 – Introduction to Database Systems。

6) EECS 340 – Introduction to Networking。

7) EECS 343 – Operating Systems。

8) EECS 345 – Distributed Systems。

9) EECS 354 – Network Penetration and Security。

10) EECS 395 – Appropriate Selected Topics - with advisor approval。

11) EECS 440 – Advanced Networks。

12) EECS 441 – Resource Virtualization。

13) EECS 443 – Advanced Operating Systems。

14) EECS 399 – Independent Study。

2.5 课程设计

学生应该至少选择两个学期的课程设计类课程，主要内容如下：

1) 一个两学期的 EECS 399 Project。

2) 两个一学期的 EECS 399 Projects。

3) 一个 EECS 399 Project 和一门课程设计。

4) 两门课程设计。

文章编号：1672-5913(2009)23-0012-03

“编译原理”课程实验项目介绍

王生原，董　渊，张素琴

(清华大学 计算机科学与技术系，北京 100084)

摘　要：在“编译原理”课程的教学中，实验项目是十分关键的部分。Decaf/Mind 项目是近几年清华大学计算机系本科生“编译原理”课程的主体实验项目，在该项目中，学生在实验框架基础上，针对一个简单面向对象语言的实现开展 4～5 个阶段的编程实验，对理解和巩固理论知识以及提高软件系统的开发能力有较大帮助。本文就 Decaf/Mind 项目的背景、内容以及实施情况进行简要介绍。

关键词：编译原理；课程实验；Decaf/Mind 项目

中图分类号：G642　　**文献标识码**：A

在清华大学计算机系本科生“编译原理”课程的教学中，Decaf/Mind 课程实验项目从 1998 级开始，至现在的 2007 级，经历了 10 届学生。1998～2002 年，该项目是选作的(但分值较高的)，自 2003 级之后成为必做的课程实验项目。本文首先介绍 Decaf/Mind 项目的背景，然后根据目前的情况(2006～2007 级)，对该实验项目的内容以及实施情况进行简要介绍。

1　Decaf/Mind 项目的背景

2001 年，我们引进了 Stanford 课程 CS143(Compilers, http://www.stanford.edu/class/cs143/, CS143, Stanford University)的课程实验框架(其原始框架由 Julie Zelenski 设计)。该实验框架设计实现一种简单面向对象语言 Decaf 的编译器，因此我们称之为 Decaf 项目。

Decaf 是一种强类型的、单继承的简单面向对象语言，是一种较为流行的教学语言，曾经在 Stanford、MIT、University of Tennessee、Brown、Texas A&M、Southern Adventist 等多所大学的相关课程中使用。

在 1998 级本科生的“编译原理”课程(2001 年秋季学期)中，我们首次采用了 Decaf 项目，并根据需要对实验框架进行了一定的调整，包括适应 Windows 平台、增加目标代码在 X86 的执行以及对源语言进行适当的改动等。比如在 2002 级，我们对该项目进行一定的简化之后，称之为 TOOL 项目。

从 2003 级的课程之后，我们对原始的 Decaf 项目实验框架进行了 3 次实质性改动。

在 2003～2004 级的 Decaf 项目中，我们将原先实验框架的开发语言由 C++改为 Java。

在计 50 班(2005 级“姚”班)的“编译原理”课程中，我们参考了 U.C.Berkeley 课程 CS164(Programming Languages and Compilers, http://inst.eecs.berkeley.edu/～cs164/archives. html, CS143, University of California at Berkeley)的 COOL 课程项目以及 Cornell 大学课程 CS412(Introduction to Compilers, http://www.cs.cornell.edu/ courses/cs412/2003sp/ CS412/413, Cornell University)中所采用的 Iota 项目，将实验框架由原来的单遍组织改为多遍组织，我们称之为 Mind(Mind is not decaf)项目，并称源语言为 Mind 语言。

由于计 50 班的编译课程安排在 Java 程序设计课程之前，所以首次 Mind 项目的开发语言为 C++。随后，在 2005

级其他班的课程中，我们又将开发语言由 C++改回 Java。

从 2006 级开始，实验框架没有发生大的变动，只是对其进行微调或进行适当简化。下面是对 2006～2007 级教学情况的介绍。

2 课程实验项目的内容

Decaf/Mind 项目的实验框架是设计实现 Decaf/Mind 语言的编译器，该编译器的工作原理如图 1 所示。

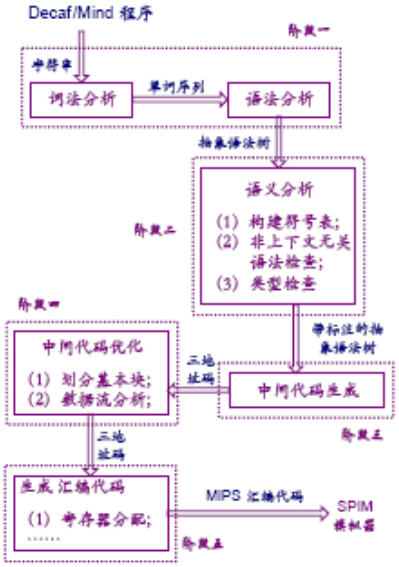


图 1 Decaf/Mind 编译器工作原理

我们将实验分成如下 5 个阶段：

阶段 1：词法和语法分析。借助 Lex 和 Yacc 实现词法和语法分析，一遍扫描后直接产生一种高级中间表示(实验指定的抽象语法树 AST)。使用的 Lex 和 Yacc 版本分别为 Flex(Jflex, http://jflex.de/) 和 BYACC/J(BYACC/J, http://byaccj.sourceforge.net/)。原始的 Decaf 项目采用 Yacc 实现主要编译阶段，实验的 3 个阶段(词法、语法分析阶段，语义分析阶段，生成三地址中间代码阶段)是一个单遍的过程(注：从三地址码到 MIPS 汇编代码的翻译由实验框架给定，没有安排阶段实验)。对于这个单遍的实验框架，前几届学生感觉调试的难度较大，有一些语言特征难以实现，但它可以和理论课学习中语法制导翻译的部分相呼应。修改框架后，依赖于 Yacc 工具的部分大大简化，生成抽象语法树 AST 形式后，符号表的建立和静态语义的检查工作只需使用树遍历算法就可实现，这符合现实中编译系统的实际情况。这种框架不能与语法制导翻译的理论直接呼应，但可用于间接指导。

阶段 2：语义分析。遍历抽象语法树构造符号表、实现静态语义检查(非上下文无关语法检查以及类型检查等)，产生带标注的抽象语法树。在这一阶段中，我们把语义分析分为对抽象语法树 AST 的两趟扫描进行：第一趟扫描时建立符号表的信息，并检测符号声明冲突以及跟声明有关的符号引用问题(例如 A 继承于 B，但是 B 没有定义的情况)；第二趟扫描时检查所有的语句以及表达式的参数数据类型。通过这一阶段的实验工作，学生可以熟练掌握 Visitor 设计模式的使用。

阶段 3：中间代码生成。将带标注的抽象语法树(decorated AST)所表示的输入程序翻译成适合后期处理的另一种中间表示方式，即三地址码 TAC，并在合适的地方加入诸如检查数组访问越界、数组大小非法等运行时错误的内容。通过一阶段的实验工作，学生可以掌握常见语言成分的中间代码翻译方法，并且可以对过程调用约定、面向对象机制的实现方法、存储布局等内容有切实的了解。

阶段 4：中间代码优化。根据教学计划，目前的实验框架只要求基于 TAC 实现一些简单的数据流分析，没有包含中间代码优化算法的内容。在 2005～2007 级的实验中，只要求学生实现活跃变量分析，既包括以基本块为单位的分析，也包括以基本块内单个语句为单位的分析。

阶段 5：目标代码生成。实验框架包括汇编指令选择、寄存器分配和栈帧管理，实验内容可以设计为对这些部分进行改进。考虑到学生负担问题，目前我们没有安排这一阶段的实验任务。

完成这些阶段后，即可产生适合实际 MIPS 机器的汇编代码，可以利用由美国 Wisconsin 大学所开发的 MIPS R2000/R3000 模拟器 SPIM(MIPS SPIM, University of Wisconsin-Madison, http://pages.cs.wisc.edu/~larus/spim.html)来运行这些汇编代码。

3 课程实验项目的实施

Decaf/Mind 项目一般在“编译原理”课程学期的第 4 或第 5 周开始，历时 8 周(遇特殊情况顺延)，共 4 个阶段，各阶段 2 周。为鼓励学生积极进取，学生可以对实验工作自行扩展，自行扩展部分在第 4 阶段截止日的随后两周内完成提交。

2006 级每一阶段的满分成绩为 10 分，实验成绩满分 40 分。2007 级实验成绩满分 35 分，各阶段的分布情况是：第 1～3 阶段为 9 分，第 4 阶段为 8 分。自行扩展部分在 4 个阶段的评分完成后统一评分，最高 5 分，将直接加入总评成绩。

各阶段评分的依据包括程序部分和实验报告部分。程序部分主要看输出结果与标准输出的一致程度，占每阶段成绩的 80%；报告部分主要看对提交的作业报告的描述，例如是否清楚说明了自己的工作内容等。每阶段截止提交 2 周后，该阶段成绩将被公布。如有抄袭，将取消阶段成绩，并给予警告。每名学生共有 2 天的晚交额度，每超过 1 天在总成绩中扣 2 分，公布评阅结果时会同时提醒每名学生剩下的晚交额度。

对自行扩展部分的评价是综合考虑创新性、实用性、合理性、难度、工作量等因素进行的。我们要求该部分选题一定是在已有实验框架基础上进行的有意义的改进工作，通常需要与教师或助教沟通后方可确定选题。教师可以对该部分的选题进行必要的引导，以避免一些意义不大的重复性工作。比如引导学生开展如下选题：函数式风格语句的实现、例外处理支持、垃圾回收机制、新的代码生成机制(必要时增加新的低级表示)、有意义的优化算法(必要时增加新的中间表示)的实现等。

4 结语

本文简要介绍了清华大学计算机系本科生“编译原理”课程 Decaf/Mind 课程实验项目的基本情况，包括该项目的背景、内容以及实施情况。

Decaf/Mind 课程实验项目实施以来受到计算机系学生的重视。对大多数学生而言，这个实验项目是入学以来遇到的第一个有一定规模的完整的程序设计项目(“编译原理”课在大三上开设)，又由于本课程在计算机系课程体系中具有重要地位，加之项目本身的魅力，学生兴致较高，综合能力有明显提高。

“编译原理”课程实验项目的设计和实施是一项较为复杂的系统工程，期待国内同仁对本文所介绍的实验项目提出宝贵的意见。

致谢：Decaf/Mind 课程实验项目经过多届学生的实践日趋成熟。我们要特别感谢杨俊峰(Stanford 助教)、张迎辉(1999～2000 级助教)、毛雁华(2000～2001 级助教)、刘天淼(2001 级助教)、唐硕(2002 级助教)、梁英毅(2003～2005 级助教)、张铎(2005～2007 级助教)、蒋波等学生的倾情奉献。还有许多老师和学生对该项目作出了贡献，但由于失去统计，没能一一列出他们的名字，这里一并表达感谢。

Introduction to the Course Lab-Project for Principles and Practice of Compiler Construction

WANG Sheng-yuan, DONG Yuan, ZHANG Su-qin
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084,China)

Abstract: The lab-project is very important in the course Principles and Practice of Compiler Construction. The Decaf/Mind project is the major lab-project of this course for the undergraduates in Department of Computer Science and Technology in Tsinghua University. In the project, students will come through 4～5 phases of coding experience to implement a simple object-oriented language on the basis of the project framework. The project is helpful for both the theoretical study and the practical training in developing software system. The background, content and arrangement of the Decaf/Mind project are briefly introduced in the paper.

Key words: principles and practice of compiler construction; course lab-project; Decaf/Mind project

The Exploration for Computer System Capacity Training in Experimental Teaching

Li Shan-shan, Quan Cheng-bin, Chen Yong-qiang
Department of Computer Science and Technology
Tsinghua University
Beijing, China
lishanshan,quancb,chenyongqiang@tsinghua.edu.cn

***Abstract*—Computer system capability is an essential capability for the computer professional undergraduate students, and the system capacity training becomes an important target for the computer professional education. Computer system design experiment is the key method to achieve this target. This paper introduces the exploration in training computer system capability of the Dept. CS&T Tsinghua University, which establish a series of course experiments to form a computer system experiment. This paper focuses on the experimental teaching, from the course contents adjustment to course experiments integration, to achieve an experimental teaching system for system capacity training. And this paper presents some key points for the computer system experiment, and gives a sample experiment embodiment of the computer system experiment for the computer system capability training.**

Keywords—Computer System Capacity; Experimental Teaching

I. INTRODUCTION

For the computer professional students, the systematic view of the computer system becomes more and more important. In their eyes, the computer should be a completed organic whole which can work coordinately, not just the software or hardware part working separately.

In Computer Science Curricula 2013(CS2013), the System-level perspective becomes one of the expected characteristics of the computer science graduates. It requires the graduates of a computer science program think at multiple levels of detail and abstraction. This understanding should transcend the implementation details of the various components to encompass an appreciation for the structure of computer systems and the processes involved in their construction and analysis. They need to recognize the context in which a computer system may function, including its interactions with people and the physical world.[1]

In China, the Guidance Committee of Computer Professional Education of Ministry of Education also requires to enhance the system ability training in computer education and practice. It listed four main abilities of the computer professional, which are computational-thinking ability, algorithm design and analysis ability, program design and implementation ability and system ability. The system ability includes system cognition, system design, system application and system implementation.[2]

In order to meet these requirement, computer professional students are required to have the capabilities of the system design and system application. These capacities requires the students to master basic computer system knowledge, understand the interaction between computer hardware and software systems. It is just the computer system capacity training requirements.

The computer system capacity is a comprehensive ability, which includes the use of the system level view consciously, understanding the integrity, relevance, hierarchy, dynamic and opening of the computer system, and also requires the students be able to use a systematic approach to master the collaborate of the computer hardware and software, understand the mechanism of interaction between them[3].

In order to meet the needs of computer system capacity training, the core courses of the computer science undergraduate curriculum system should be adjusted to make the correlation between the course content closer, and let the courses connections smoother. Similarly, the experimental teaching of these courses should be carried out around the computer system capacity training. It requires the course experiments connected seamlessly, that is the course experiment should undertake the content of the previous course experiment and at the same time prepare for the subsequent course experiment. Thus all these experiments will eventually form an integral experimental teaching system.

II. EXPERIMENTAL TEACHING IN SYSTEM

The experiment or practice training is an important method to help the student to understand the knowledge of the courses. In the CS2013 the Project Experience is listed as the characteristics of the computer science graduates. It mentions that all graduates of computer science programs should have been involved in at least one substantial project. Such projects should challenge students by being integrative, requiring evaluation of potential solutions, and requiring work on a larger scale than typical course projects. Students should have opportunities to develop their interpersonal communication skills as part of their project experience[1]. The experiment of computer system is a good project for this requirement, which need the students to complete by team work and is an integration practice of several courses in experimental teaching .

The current computer curriculum system includes a large number of computer systems related courses, such as Digital Logic Circuits, Assembly Language, Computer Organization, Computer Architecture, Compile Theory, Operating Systems, Embedded Systems, etc. Each of these courses has its own separated experiment contents, but these experiments are not good or enough for helping students to build systematic knowledge of the computer hardware and software systems. Through these courses learning and experiments practice, the students have in-depth understanding of the subsystem introduced in these courses, but they have not such understanding on the interaction between the various subsystems of other courses or on how these subsystems integrated to one computer system. That is the students have not really grasp the computer knowledge in systematic level.

The main reasons for this situation we think are the following two aspects:

(1) In the teaching process, these courses are planned and carried out independently. And these courses emphasize the integrity and systematic for their own self-knowledge system, thus it causes the redundancy of the knowledge, and meanwhile it also leads to insufficiency of the interconnection content between the consequent courses, so it is difficult to form a complete computer system knowledge system for the students.

(2) In the experiment process, these courses focus on the study of their own content, so the experiment is lack of the continuity of the leading course experiments, as well as the supports for the subsequent course experiments. And most of the experiments are mainly the simple verification of the principle, lack of the practice for complex integrated system design.

Computer experiment teaching is an integrated process, which need high correlation between the courses. The experiments need overall planning from the systematic level, in order to effectively develop and training students' ability in computer system capabilities[4]. Therefore it is demanded to adjust the experiment teaching system.

First of all, on the teaching principle, the courses should focus on how to improve the level of the system capability training. Based on the idea of "Focus system, Emphasizing experiments, Building capacity", the related courses should adjust the courses' content to fit for the system level teaching.

Second, experiment teaching content need to be planned unified, in order to set up a complete knowledge system on systematic level for the students, from the underlying hardware to the operating system and the compiler, to form a complete experimental teaching system.

Furthermore, it is needed to establish a unified teaching experiment platform, thus each course experiment will be on the same implementation carrier, which is helpful for the coherence in these experiments, and also convenient to carry out the experiments.

Finally, the target of the experiments is that the students can design and implement an educational computer system, which is based on a certain instruction set, and on it they can run an operating system kernel and implement a compiler for this system. The computer hardware, OS and compiler are all implemented by the students themselves. Through the practice on this system experiment, the students will comprehend the computer on systematic view and the system capability will be improved greatly.

III. COMPUTER SYSTEM EXPERIMENT

In order to make up the deficiencies of the students system capacity in the teaching of computer professional courses, the Guidance Committee of Computer Professional Education suggests several universities to attempt the system experiment project, in order to explore an approach on the integration of course experiments using the methods of the engineering education, to form a systematic experiment of the computer professional. The computer departments of Tsinghua University, Beihang University, Zhejiang University, etc all joined this project, and have made some progress in the exploration process.

In Tsinghua university, the Computer Experiment Teaching center is the National Demonstration Center[5], in order to achieve the objectives of the system experiment, we made following changes in the system related courses and experiments: adjust and optimize the courses content, integrate the experiments, design the experimental system and develop the experiment platform. In the past three years, we continuously improve our system experiment, and gradually formed a prototype of the experimental teaching system for the goal of system capacity training.

A. Planning Experimental Teaching Unified

Based on the computer professional curriculum system, we analyzed the knowledge points of each courses related to the system experiment, and listed the overlapped teaching content in each courses, found back the connection content missed before between the consequence courses. In the teaching process, these courses are planned unified and cooperated with each other.

The computer system experiment mainly involves Digital Logic Circuit, Assembly Language, Computer Organization, Operating System, Compiler Theory, these five courses. In order to meet the requirements of the computer system experiment, the experiments of these courses were analyzed and adjusted, focusing on the interconnections of the contents between the course experiments. The detail adjustment contents is shown in Table I.

These adjustment of the entire experiments is intended to link up the experimental teaching contents in each courses to form a complete computer system experiment. This system experiment will strengthen the training of computer system capability.

B. System Experiment Dazign

The computer system experiment is a complex system level experiment. The students need to design and implement a complete computer system. They should complete several stages in each course, and in the Computer System Design

TABLE I. ADJUSTMENT AND OPTIMIZATION IN EXPERIMENTS

Course Name	adjustment or optimization content
Digital Logic Circuit	1. Strengthen the experiments of using programmable logic device, including gate level to component level design 2. Emphasis on the experiment of the computer component. 3. Learn how to use the corresponding EDA tool for complex system design.
Assembly Language	1. Introduce the assembly language as the interface and connection of the hardware and software system. 2. Add MIPS instruction set experiments, including interrupt and exception handling, virtual memory management, etc. 3. Add the experiment of typical C code expressed in assembly language level and the disassembly experiment.
Computer Organization	1. Design and analysis of instruction system based the MIPS instruction set 2. The design and implementation of a simple computer systems, in which the CPU at least supports the subset of the designed instruction set with Multi-cycle or pipelined. 3. The designed CPU should support interrupts, including soft interrupt and hardware interrupt. 4. Add the TLB experiment of the virtual memory management.
Operating System	1. The designed instruction set / processor in Computer Organization course is the target platform of the operating system. 2. Strengthen the combination of operating system theory and experiment. Require to complete a simplified mini OS that can be able to work on the real hardware platform. 3. Emphasize that the core algorithms of each experiment can form an organic whole. The experiment will use its previous experiments code, and eventually form a complete small operating system.
Compiler Theory	1. The designed instruction set / processor is the target platform of the compiler. 2. Implement a compiler that supports the experimental platform used in previous courses. 3. Add the experiment of optimization for the specific experimental system.

course they will eventually integrated previous experiments to form a complete computer system. So, the relevant courses should adjust the experimental system and experimental content, treat their own experiment as the module or base of the final comprehensive system experiment.

Assembly Language and Digital Logic Circuit is the experimental foundation courses, providing an experimental basis for other courses; the experiments of Computer Organization, Operating System and Compiler Theory will become the parts of computer systems. The experiments of Computer Organization will implement the basic hardware, and Operating System experiments will be the software parts. The following are the experiments of these two courses.

The experiments of Computer Organization will implement the hardware part of the system experiment, with three-level experiment contents: verification level, design level and comprehensive level experiment content. The final experiment is to design a computer hardware which can run an assembly program called monitor program that can manage the hardware resources of the experiment platform.

Lab 1 (Instruction Set Lab). This is a verification experiment, writing the assembler language programs in the simulator using the instruction set for the designed CPU. This experiment is in order to let the students be familiar with the instruction set and understand the function of the simulator and monitor program.

Lab 2 (Component Lab). Design and implement the ALU. Let the students learn the basic ALU design methods and data path, and to be familiar with the hardware experiment platform. The designed ALU will be used as important component of the CPU in the following experiments.

Lab 3 (Memory& IO Lab). Design a state machine to access the memory and IO on the platform. It is a design level experiment, which will help the students to comprehend the memory access timing and learn how the data exchanged on the bus.

Lab 4 (Hardware System Lab). Design and implement a complete computer hardware system on the platform, and the monitoring program can run upon the designed hardware CPU core. A data transfer program will load the binary code of the monitor program directly into the memory, and then the designed computer will run the monitor program, which will communicate with PC. This experiment is a comprehensive experiment that allows students to learn how to design an underlying hardware of the education computer, and this computer should support the OS in the Operating System course.

Operating system is the basic software in the computer system, and it is tightly integrated with the hardware[6]. In order to support the hardware designed in Computer Organization, Operation System course's experiments were based on ucore OS and transplanted to the new platform[7].

Lab 0 (experimental operating system environments and tools).It aims to let the students understand and become familiar with the tools and process in the whole course experiments, including kernel debugging, simulator, etc.

Lab 1 (startup process). It will implement the bootloader for loading and running the operating system, in order to understand the process of starting the bootloader, bootloader's files, the boot process of the ucore OS, and interrupt handling mechanism.

Lab 2 (physical memory management) & Lab 3 (virtual memory management). These labs will help the student to understand how the system manages the memory.

Lab 4 (kernel thread management)& Lab 5 (user process management)& Lab6 (scheduler)& Lab 7 (synchronous mutex). These labs will help the students understand the process of the kernel thread creation and execution, and understand how to implement the context switching.

Lab 8 (File System). It requires the students to understand the file system and its implementation technique.

After these labs, the students will grasp a simple operation system of ucore in the simulator, and in following course they will run it on their designed hardware.

C. Experiment Platform

The unified planned system experiment needs a unified platform to support the experiment deployment. This platform need to support each course experiments, and can support the comprehensive computer system experiment as well. The platform includes not only the actual hardware circuit board, but also the software tools, which contains the debugging tools, simulator, assembler, compiler and other tools. All of the platform should support a simple, standardized instruction set. So we choose about 50 MIPS like instructions as our instruction set, and designed the circuit board, named THINPAD(TsingHua mINi PAD)[8], as the hardware carrier. We also developed some software tools, including system simulator, assembler, compiler, terminal program and etc. And we have run the teaching operating system ucore on this platform. In addition, the corresponding compiler was developed. The program generated by this compiler can be run in the ucore. Thus the system experiment platform was established and has been used in the course experiment teaching.

1) Hardware platform.

The figure 1 is the composition of the hardware platform. The circuit board uses the programmable logic device as the experiment carrier. This programmable device is a FPGA, which is the experiment chip that will act as the CPU of the experiment computer system. The students need to design and implement a CPU using the EDA tools, and configure the design into the experiment chip. The platform provides two separate memory sections(Base Memory and Extend memory), act as the instruction memory and data memory. The flash is used to store the operation system and RS232 interface is the input/output of the experiment computer. And there are several peripherals to help debugging, such as LEDs, switches. these devices like SRAM memory, Flash memory and other peripherals are all connected to the experiment chips via several buses.

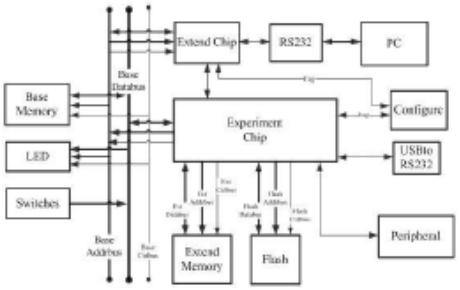


Fig. 1. Example of a figure caption. *(figure caption)*

2) Software tools

For the system experiment and the hardware platform, we developed several software tools to help the students completing the experiment. The following is the main tools:

a) Instruction simulator. This simulator can help students complete software simulation and debugging. The students can use this simulator debug their assembly program, avoiding directly debug on the hardware platform.

b) Assembler. This assembler can convert assembly language to binary code of the hardware platform. This tool can help the student to write some simple assembly program for their designed computer system.

c) Data communication program. This program can load/store the data or program into the memory/Flash on the board.

d) Monitor program. Before running the operating system on the hardware, this program is a relatively complex program which can be a preliminary test on the hardware system implementation. It can work as a mini operating system to manage the platform's resources and testing programs.

e) GCC compiler. As the operating system is written in C, the GCC compiler can compile the operating system to the instruction set. Of course, this GCC compiler was modified for experiment platform, and it only used for the operating system (ucore) compilation.

f) Hardware simulator: This simulator is a hardware simulator which is specifically designed for our experiment CPU. In this simulator, the student can get the signal value of the designed hardware.[9]

D. Computer System Design Course

In order to integrate the experiments of system related courses, we add a new course in the curriculum called Computer System Design course, which is an experimental opening course. This course has been selected as the Challenging Courses of Tsinghua University.

After completion of previous courses, the students will use the knowledge learned in Digital Logic Circuit, Assembly Language, Computer Organization, Operating System and Compiler Theory, etc. to design and implement an integral simple computer system independently in this course. It will improve the students' problem solving skills and the computer system capacity.

E. Experiment Process and Evaluation

The system experiment involves several courses' experiment. In each course, the students need to finish a part of experiment which will be prepared for the next course experiment. As the system experiment is very complex, so it need the students to complete in team work. They will keep working together in the whole system experiment. which will improve their communication and collaboration skills.

The final experiment evaluation contain two aspects:

One is on-site checking, the teams need to show their experiment to the teacher or TA, that is running their experiment computer on the platform. The teacher or TA will check whether they have implemented the basic function required and validate whether their computer can run the operation system (ucore) and the testing programs successfully. And if they have completed some additional functions, they will get some extra points on their scores. At last the teacher or TA will give an evaluation result of their experiment.

The other is presentation, each team will give a presentation to other teams, and these teams will give a evaluation result for this presentation. Because the students doing the experiment together, they will know each other more clearly. And the presentation will improve their express ability.

Thus, Combined these two evaluation result, our evaluation will be relatively fair.

IV. Achievements

The computer system experiment has gradually carried out in the undergraduate teaching of the Department of Computer Science& Technologies, Tsinghua University. The teaching experimental platform has been used in the course experiments, and got good results. Some students has completed the system experiment: the ucore has run on their designed CPU, and some applications can be compiled by their own compiler and running successfully in the system. After complete the real first computer system, the students have an enormous sense of achievement. Figure 2 is a computer system implemented in the experiment course, which is running a slide shown program.

Through the computer system experiment, the students generally reflect that they understand the computer system more deeply and this experiment is good for their comprehending the knowledge in these corresponding courses. All courses use the unified experiment platform brings more convenient for the students experiments. The computer system experiment gave them a platform to use their knowledge and skills, and it helps them get more profound understanding of the mechanisms and principles of computer system.

Fig. 2. Computer system implemented in experiment

V. Summary

Computer system capacity training is an important target for computer undergraduate teaching, in which experimental teach plays an important role. Dept. CS&T Tsinghua University explored several years on it and established an experimental system, which has been deployed and got some achievement in the teaching process. Through this experimental system, the students' computer system capability has effectively improved.

Acknowledgment

I would like to recognize LiuWei-dong for guiding us on the experiments of Computer Organization, Xiang Yong for guiding us on the experiment of Operation System, Wang Sheng-yuan for guiding us on the experiment of Compiler Theory, Liu Cong, Wang Ning-ping, Jia Kai for implement the system experiment.

References

[1] The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, Computer Science Curricula 2013, http://www.acm.org/education/CS2013-final-report.pdf.

[2] Guidance Committee of Computer Professional Education of Ministry of Education, Computer science and technology professional ability cultivation of colleges and universities, Beijing, Mechanical Industry Press, 2010.

[3] Wang Z.Y. and Zhou X.S, Research on Systematic Ability for Computer Professional Students and Curriculum, Computer Education, 2013.9, pp1-6.

[4] Yuan C.F. and Wang S, University Computer Education Focus on System Concept Training, China University Teaching 2013.12, pp41-46.

[5] Li. P and Jin. Q, Reform and Implementation of Experimental Teaching for Demonstration Center, Proceedings of the 2012 Second International Conference on Business Computing and Global Informatization ,pp887 – 890, 2012.

[6] Ma H.B. and Ke J, Reform and Practice of Experimental Teaching of Computer Hardware, Research and Exploration in Laboratory, 2013.10, pp360-362.

[7] Chen Y. and Xiang Y, Guide for Operating System Experiment, Beijing, Tsinghua University Press, 2013.

[8] Liu Y.N. and Liu W.D, Design of THINPAD Educational Computer Platform, Experimental Technology and Management, 2012.11, pp115-118.

[9] Michael Black, A hardware simulator for teaching CPU design, Annual Joint Conference Integrating Technology into Computer Science Education , p380-380. 2012.

Remote FPGA Lab Platform for Computer System Curriculum

Yuxiang Zhang
Tsinghua Univ.
zhangyuxiang13@mails.tsinghua.edu.cn

Yu Chen
Tsinghua Univ.
chenyu15@mails.tsinghua.edu.cn

Xiaojian Ma
Tsinghua Univ.
maxj14@mails.tsinghua.edu.cn

Yuhan Tang
Tsinghua Univ.
yh-ta14@mails.tsinghua.edu.cn

Yilin Niu
Tsinghua Univ.
niuyl14@mails.tsinghua.edu.cn

Shanshan Li
Tsinghua Univ.
liss02@mails.tsinghua.edu.cn

Weidong Liu
Tsinghua Univ.
liuwd@mail.tsinghua.edu.cn

ABSTRACT

This paper presents a MOOC-ready online FPGA laboratory platform which targets computer system experiments. Goal of design is to provide user with highly approximate experience and results as offline experiments. Rich functions are implemented by utilizing SoC FPGA as the controller of lab board. The design details and effects are discussed in this paper.

CCS CONCEPTS

• Social and professional topics → Computing education programs; • Hardware → Reconfigurable logic and FPGAs; *Logic circuits*; • Computer systems organization → *Architectures*;

KEYWORDS

FPGA; MOOC; Remote Lab; Digital Circuit

1 INTRODUCTION

The computer system is one of the key curriculum for computer science major students. Courses such as Digital Circuit, Computer Organization, Compiler and Operating System introduce computer system in different aspects. To deepen the understanding of the computer architecture, experiments are usually necessary in these courses. For comprehensive and systematic understanding of computer system, the knowledge units introduced in all of these courses should be covered in the experiments.[7]

In Computer Organization course, students are required to build some function units with digital circuits,including ALU (Arithmetic Logical Unit), serial port, memory controller, and simple processors. At this stage, the processor is capable of running a monitor program.

Light-weight operating system can be ported to this processor as the training of Operating System course. Finally, students may build a self-contained computer system which implements a specific ISA (e.g. MIPS32) based on prior works. The light-weight operating system running on the computer verifies the correctness of implementation. Students may also modify the compiler to support their computer system.

To support the experiments mentioned above, platform must provide programmable logic and essential I/O functions. The FPGA based development boards are the common choice. However, the emerging ways of remote education like MOOC(Massive Open Online Course) makes it hard for students to do experiments with boards in hand. The experiment have to be done online. Furthermore, MOOC requires more resources since the students enrolled can be much more than offline classes.

In this paper we represent a remote FPGA experiment platform optimized for experiments of computer system curriculum. We have designed an FPGA lab board and software, which supports both online and offline experiments. Figure 1 gives the remote user interface and photo of lab board. The goal is that experience of doing experiment online should approximate to offline experiment, and the result of online experiment should be reproducible offline. In online scenario, a number of lab boards are installed in the server room, connected to server via network. Students operate a visualized board in web browser, all of the operations are executed on assigned board.

2 RELATED WORK

Some online FPGA lab designs have been proposed in previous works. In [1], by connecting I/O ports of FPGA development board to parallel port of PC, the concept of remote FPGA lab was tested. To display the state of real hardware, Webcam was used in [6] and [2]. [6] utilized the Altera's In System Memory Content Editor with logic inside FPGA to implement virtual switches. [2] used low-cost MCU as the an adapter between FPGA and PC. Only basic user I/O control provided in implementations above, they are not suitable for complicated experiments. In [3] a laboratory system with visualized view was presented. It was based on NI lab equipment and LabView software. In [4] and [5] the system-specific logic in FPGA cooperated with external USB controller to transfer data between user logic and servers. To run on this platform, user's logic have to be integrated to given wrapper code

ACM TUR-C '17, May 12-14, 2017, Shanghai, China

DOI: http://dx.doi.org/10.1145/3063955.3063958

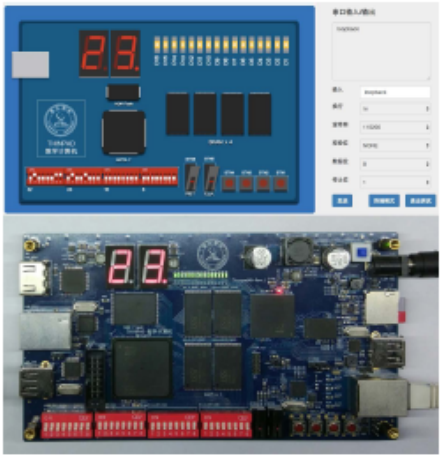


Figure 1: Web UI and lab board

before compilation. Communication through USB also limited the scalability of system, because one server may work with few boards considering the bandwidth of USB host controller.

Another approach of online digital circuit laboratory is building a platform based on simulation. [8] presented a web-based electronic circuit simulation system, in which the SPICE simulation engine is chosen. A major issue of using such simulation system for experiments of computer system curriculum is that, the performance of cycle accurate simulator is not enough to support a complicated design(e.g. MIPS32 CPU). Simulation of a processor running operating system is almost not possible. Besides, the experiment results of simulation may not be as accurate as the results produced by real hardware.

3 TYPICAL EXPERIMENTS

This section lists some typical experiments related to computer system curriculum. Platform design should meet the requirements of the following experiments.

3.1 Digital Circuit Basics

As a tutorial of digital system design, basic input, output and state machine design methodologies are introduced in this experiment. For example, the task can be building a counter with segment display. Additionally, the counter can be controlled by switches, then switch debouncing logic is required.

Another example is ALU(Arithmetic Logic Unit) implementation. ALU designing requires understanding of combinational arithmetic circuits. In this experiment, the ALU designed inside the FPGA takes input from switches, and displays results on LED.

3.2 RAM Access

Memory is one of the main parts of computer system. Memory reading and writing are the basic operations of the processor. In the RAM access experiment, students have to design a memory controller, which generates the signals meeting the timing of SRAM. If data read from RAM matches they written, the implementation is considered correct.

3.3 Serial Port Communication

Serial port communication is a basis of follow-up experiments. Students need to make communication with other device(e.g. PC) by reading/writing registers of serial port controller chips. Besides the communication itself, the techniques of bus arbitration are also important. Because serial controller chip shares bus with memory, so bus contention have to be properly handled by hardware logic.

3.4 MIPS CPU Implementation

Implementing a MIPS processor can be a harder task, the theory of computer organization is applied to the practice. Students should implement a processor, which compatible to a given simplified MIPS ISA. A monitor program is also provided to verify the processor implementation.

As a challenge task, additional functions such as TLB and exception are required to be implemented. This task requires deeper understanding of the processor architecture and implementation techniques. With these essential functions, an operating system can be ported and runs on the processor.

4 LABORATORY PLATFORM DESIGN

4.1 Overview

The proposed platform consists of a number of lab boards and at least one server, shown in Figure 2. All of the user interactivity and laboratory management jobs are processed on the server, while the user-implemented FPGA logic runs lab boards. Each user will be assigned to one lab board after logged in. When users exit the system, the board assigned will be recycled and ready for next user.

Each lab board consists of two subsystems: The laboratory subsystem and controller subsystem. All resources of the laboratory subsystem are open to the user assigned to this lab board, which means the FPGA is fully programmable for user, nothing is reserved for remote control. The controller subsystem monitors the laboratory subsystem, and exchanges data between laboratory subsystem and servers.

For example, if users turn on a switch on remote UI, the server will send this request to controller subsystem, which sets the logic level of corresponding switch of laboratory subsystem to high. Then, the user-programmed logic in laboratory subsystem may drive an LED high, this level change will be captured by controller subsystem, which sends the event to server, resulting in the corresponding LED highlighted on remote UI.

4.2 Laboratory Subsystem

4.2.1 On-board Resources. Laboratory subsystem features a Xilinx Artix-7 series FPGA, with 101,440 logic cells, which is capable of running most experiments related to computer architecture

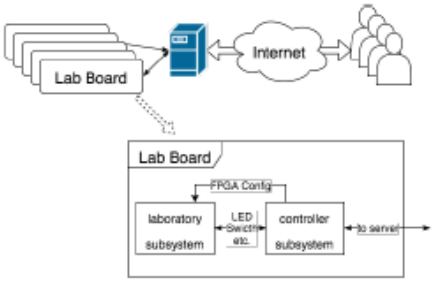


Figure 2: Remote laboratory platform architecture

curriculum. Large logic capacity makes it possible for students to implement more advanced functions such as Caches.

Four 2 MB 16-bit SRAM are organized as two independent units, with 1 Mega × 32-bit in each unit. The address, data and control signals of two units are separately routed to FPGA. This memory schema is useful for some simple architecture, where instructions and data are stored in two different memory spaces. Because instructions and data can be simultaneously accessed, structural hazards can be avoided. Users may also merge two memory units into a single 8 MB memory unit with glue logic. By using all 8 MB memory on board, running most embedded operating systems is possible.

The parallel bus signals between FPAG and SRAM are routed to controller subsystem, hence the controller is able to monitor the memory access transactions on bus.

An 8 MB 16-bit NOR Flash memory is presented as non-volatile storage. User may use this storage to implement file system. This is also useful for storing OS image. Loading image form Flash is faster than serial port, which saves time while debugging hardware logic.

Simple I/O components are also provided for users. 32-bit DIP switch, 2 push-button with hardware debouncing and 4 push-button without debouncing are directly connected to FPGA, acting as the user input. Two 7-seg display and 16 LEDs are connected to FPGA as observable signal output.

Several wires connecting lab FPGA and controller subsystem directly are reserved for extension functions. More peripherals (e.g. PS/2 keyboard) can be emulated by controller with these wires.

Additionally, a parallel-to-DVI encoder chip on board makes it possible for user logic to generate graphical output in offline scenario.

4.2.2 FPGA In-system Configuration. One of the essential functions of remote FPGA lab is to configure the FPGA remotely. The common way to configure FPGA through JTAG requires Xilinx proprietary tools, making it hard to be integrated into third party system.

Our platform utilizes the "Slave Serial Configuration" mode of Xilinx FPGA. As shown in Figure 3, only one INIT_B signal and two serial data serials are required to configure the FPGA. An additional DONE signal can be connected to monitor the status of configuration.

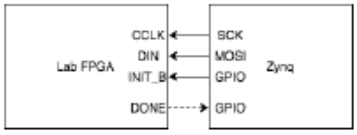


Figure 3: Slave configuration mode connections

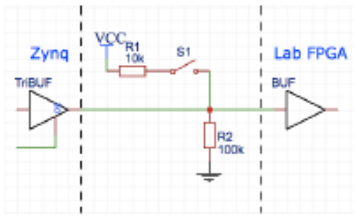


Figure 4: Switch and FPGA connection circuit

The bitstream file (*.bit) uploaded by user is firstly parsed by libxbf [1] to extract the configuration data. Then Zynq sends a low-level pulse on INIT_B pin to put the lab FPGA into configuration state, the configuration data can be transmitted to lab FPGA through 2-wire SPI connections. With SPI clock running up to 50 MHz, the whole configuration process finishes in less than 1 second.

4.3 Controller Subsystem

4.3.1 Controller Core. The controller subsystem is constructed of a Xilinx Zynq-7 SoC FPGA, which integrated an ARM Cortex-A9 processor into the FPGA fabric. FPGA and ARM CPU share the DDR memory.

Considering that some functions (e.g. LED sampling and switch control) require high I/O capacity, and some (e.g. bus analyzer) requires real-time processing, the FPGA is a suitable choice. However, network communication with server requires software involvement. By implementing hardware interface in FPGA, communication software in CPU, this design takes advantages of both FPGA and CPU.

The connection between CPU and server is established on 1000M Ethernet. The Ethernet port on board is design to be PoE-compatible. With the support of powering the board over Ethernet cable, massive deployment is simplified.

4.3.2 I/O Sampling and Control. All user I/O signals for lab FPGA, such as LED and switch connections, are also routed to Zynq for remote display and control. The GPIO core implemented in programmable logic of Zynq handles the I/O sampling and control. The sample rate of LED is over 100Hz, enough for human observation. To avoid the collision of different logic level set by physical switch and Zynq, the circuit design described by Figure 4 is used.

[1] https://github.com/wkozak/libxbf

In the offline use scenario, the pin of Zynq is set to high-impedance mode, then physical switch works. While the pin of Zynq is set to drive mode in online scenario, then remote control overrides the state of physical switch.

For several physical switch without hardware debouncing, the switch bouncing is emulated by logic in Zynq, to ensure that experiment results online matches offline ones.

4.3.3 Memory Bus Analyzer. The memory bus analyzer function is designed to help students debug their memory access timing. Like most logic analyzer, this function is able to record the transactions on memory bus and display them as waveform. With the help of analyzer, common problems like insufficient address setup time can be easily figured out. Wrong control flow implementation can also be located by checking the memory access sequence.

Since all memory singles between SRAM and lab FPGA are also connected to programmable logic part of Zynq, all memory accesses can by monitored by controller. When memory bus acquisition started, the logic implemented in Zynq waits for $\overline{WR}$ and $\overline{RD}$ assertion, then stores the address, data and timing information to internal FIFO for buffering. Finally the data in FIFO is written to DDR memory, and transferred to server for display. The acquisition can be started and stopped manually, or stops automatically when running out of internal storage.

4.3.4 Serial Port Controller Emulation. To support the serial port experiment, this platform emulated a 8250 UART compatible serial port controller. It is a hardware logic implemented in programmable logic instead of a real chip. Data received from serial port is wrapped and transferred to server via network, while user input from server is unwrapped and written to data FIFO of serial port controller. Because of the requirement that serial controller shares bus with memory, the address and data port of the controller can reuse the pins of memory bus analyzer, which saves I/O pins of Zynq.

4.3.5 Static Memory Controller. Static memory controller is a part of the programmable logic in controller subsystem. This memory controller is able to read and write the SRAM of lab FPGA through interconnects on board. It only works when lab FPGA is unconfigured, otherwise user logic may also access SRAM and cause contention. The basic idea to integrate such a memory controller is to help assess the work of students. After design of students writes data into SRAM, teacher may put lab FPGA into unconfigured state then check the content in SRAM using this function. SRAM read and write function is exposed on web UI like serial communication function.

4.3.6 Control Software. In the ARM side of Zynq SoC, an embedded Linux controls all of the hardware modules in programmable logic such as I/O sampling and serial port. The main procedure of controlling software is an event loop, which handles events from server and programmable logic. Corresponding actions are taken when receiving an event.

Serial port communication function was implemented with the Boost[2] Asio framework. Data sent by lab FPGA triggers pre-registered callback functions, which then sends the data to server.

[2]http://www.boost.org/

Figure 5: Different images used to emulate the physical state of buttons, DIP switches and LEDs

The Linux system boots from a SD card connected to Zynq for now. However, considering the convenience of software upgrade in massive deployment, the design can be changed that, only bootloader stored on board, while root file system mounts from network(i.e. the server).

4.4 Remote User Interface

4.4.1 Server Architecture. In our platform, server is the center of the whole system. It manages all of the lab boards by assigning free boards to authenticated users and recycling resources from disconnected users.

The server program is built over Node.js, an event-driven asynchronous I/O framework, which is capable of handling large number of concurrent requests. The major role of server is to exchange data between boards and browsers. Because board I/O state (e.g. LED) changes and serial port communication produce very small amounts of data, the server may theoretically handle hundreds of online users at the same time. However, this number is usually limited by the number of physical boards deployed.

Both server-to-browser and server-to-board date transfers are based on Socket.IO [3] , which is a cross-platform bidirectional communication library written in Javascripts. A native C++ implementation of Socket.IO is also available, and is used in Zynq on board. For the server-to-browser data transfers, modern Websocket protocol is used to achieve lower latency than traditional long polling method. Data transferred over Socket.IO is serialized to JSON(JavaScript Object Notation) instead of raw binary data, preventing compatibility issues among various system environment.

We choose MongoDB [4] as the database, storing user information and metadata. Currently we have logged user operations in an experiment. These data may be analyzed to optimize the platform design.

4.4.2 Visualized Operation. The visualized board is shown in the main panel, with three kinds of components: LED (including segment display), DIP switch and button. LED state changes between normal and highlighted to represent the actual hardware state changes. One clicking on a DIP switch toggles its state between 0 and 1. Pressing and holding on button sets its state to 1, while releasing sets its state back to 0. Different images are used to emulate the physical state of switches.

[3]http://socket.io/
[4]https://www.mongodb.com/

The web UI is written in pure HTML, CSS and Javascripts without dependency of obsolete Flash or Applet technologies, achieving quick page response and high browser compatibility.

4.4.3 Serial Port. Two methods of serial communications are provided on UI, simple serial input box and terminal emulator. For simple communication with serial port, user enters text on input text box and click send button, while reply from board shows on receiving text area. This function is usually used by students who want to test the hardware design. They may send some commands to hardware and get replies of hardware.

In operating system experiments, an unix-like shell is commonly required. To support the interactive shell in OS experiments, our platform provides a terminal emulator based on xterm.js [5] library. It behaves just like a normal terminal emulator in unix-like system.

The parameters of serial port such as baudrate, stop bits and parity check are adjustable with dropdown menu before opening the port. In terminal emulator, the automatic newline character conversion between "\r", "\n" and "\r \n" can be manually selected by users, making it convenient for users to work with all kinds of target operating systems inside the board.

An serial port API is also provided for user-developed application to communicate with program inside board. The API is useful for user-customized functions. As an example, a user implemented file transfer tool could work with this API to transfer file to board from computer.

4.4.4 Bus Signal Waveform Display. As the front-end of memory bus analyzer, a timing diagram display with WavaDrom [6] is implemented referring BeagleLogic [7]. After the signal acquisition, the singles of memory bus are displayed on the page with time bar and measurement tools. Users may visually find out the timing problem by comparing the actual timing value with timing characteristics specified by memory chip datasheet.

5 DESIGN FEATURES

Design of our platform is optimized for massive deployment to support the MOOC. With PoE function presented, only an Ethernet cable is needed for one board, no more accessories are required.

All signal sampling and controls are non-intrusive, no modification of user design is needed for remote experiment. The results of online experiments would be the same as the offline ones.

The difficulty of debugging timing problem on memory bus is very common among students. Because of the lack of efficient debug tools, they may spend lots of time figuring out where the problem is. To solve this issue, our platform offers the memory bus analyzer function. The tool helps user check the timing visually, and locate the problem when monitor program runs abnormally.

SRAM read and write functions assist in laboratory assessment. Checking content in memory chips becomes really easy now.

6 DEMONSTRATION

In this section we verify the capabilities of remote lab by running an actual work from students. It is an implementation of 5-stage pipeline MIPS32 processor and its peripherals, which is capable of running uCore operating system and modified Linux kernel. This design utilized 7922K LUT, 7000K FF, and 16 BRAM , which is less than 20% of the available resources of the lab FPGA on our platform.

The steps are shown in Figure 6. Firstly, user logs in and uploads the .bit file to the platform. Then clicking on reset button let processor begin executing the bootloader stored in the block RAM of FPGA. The LED indicates that the bootloader runs. Run the host program of bootloader, which loads image of OS into the SRAM via serial port, and jumps to entry of OS. Later, logs of boot process are printed in the terminal window. After the boot process, user can now interacts with OS in terminal emulator.

(a) Upload bitstream (b) OS image loading

(c) OS boot process (d) Interactive terminal

Figure 6: Demonstration of remote experiment steps

Memory bus analyzer can be activated at any time while the user-deigned logic running. Then we manually stopped the acquisition after a while. The sampled bus access sequence is displayed in a list. The corresponding signal timing is drawn as waveform when clinking on a specific memory access cycle.

7 CONCLUSION AND FUTURE WORK

We have presented a remote FPGA lab platform designed for computer system curriculum experiments. The platform make it possible for MOOC students to do course experiments with real hardware online. It also helps students debug the hardware logic design issues.

In our future work, the online HDL editing, simulation and compilation functions will be integrated, making the platform more convenient. The debugging function would be enhanced with the supporting of user-defined triggers and internal signal probes. Example of typical experiments will be provided to help beginners quickly understand the usage of the platform.

[5] http://xtermjs.org/
[6] http://wavedrom.com/
[7] http://beaglelogic.github.io/

计算机组成原理大实验
实验报告

计 24 杜　鹃 2012011354
计 22 黄　杰 2012011272
计 25 矣晓沅 2012011364

2012 年 12 月 20 日

目录

一、 实验目标

- 基于 THINPAD 教学计算机，设计：
- 基于 MIPS16 指令集的流水线 CPU
- 使用基本存储、扩展存储、 Flash、 IO 设备
- 能够运行 kernel、监控程序、 汇编测试程序

二、 指令集任务

	序号	指令	序号	指令
基本指令集	1	ADDIU	14	LW_SP
	2	ADDIU3	15	MFIH
	3	ADDSP	16	MFPC
	4	ADDU	17	MTIH
	5	AND	18	MTSP
	6	B	19	NOP
	7	BEQZ	20	OR
	8	BNEZ	21	SLL
	9	BTEQZ	22	SRA
	10	CMP	23	SUBU
	11	JR	24	SW
	12	LI	25	SW_SP
	13	LW		
扩展指令集	26	NOT	29	CMPI
	27	SLLV	30	SLTI
	28	ADDSP3		

三、 实验结果

3.1 实验成果参数

•CPU 主频为 6.25MHz（12.5MHz 有时会出一些问题，所以只能二分之，6.25MHz 是稳定频率）

• RAM 频率为 25MHz

• 正常运行 kernel 内核程序，正常运行所有 project1 程序。

• VGA 分辨率为 640*480， VGA（显存）运行频率 25

• 测试程序的运行时间：

测试程序编号	测试程序说明	测试结果（ms）
1	性能标定，这段程序一般没有数据冲突和结构冲突，可作为性能标定	28.006
2	运算数据冲突的效率测试，假设正确处理了数据冲突，	22.004

	有数据冲突的地方不再加 NOP。	
3	假设正确处理了延迟槽，行为与模拟器一样，延迟槽里可能填充语句。	20.013
4	访存数据冲突性能测试	32.018
5	读写指令存储器测试	16.019

3.2 实验成果简列

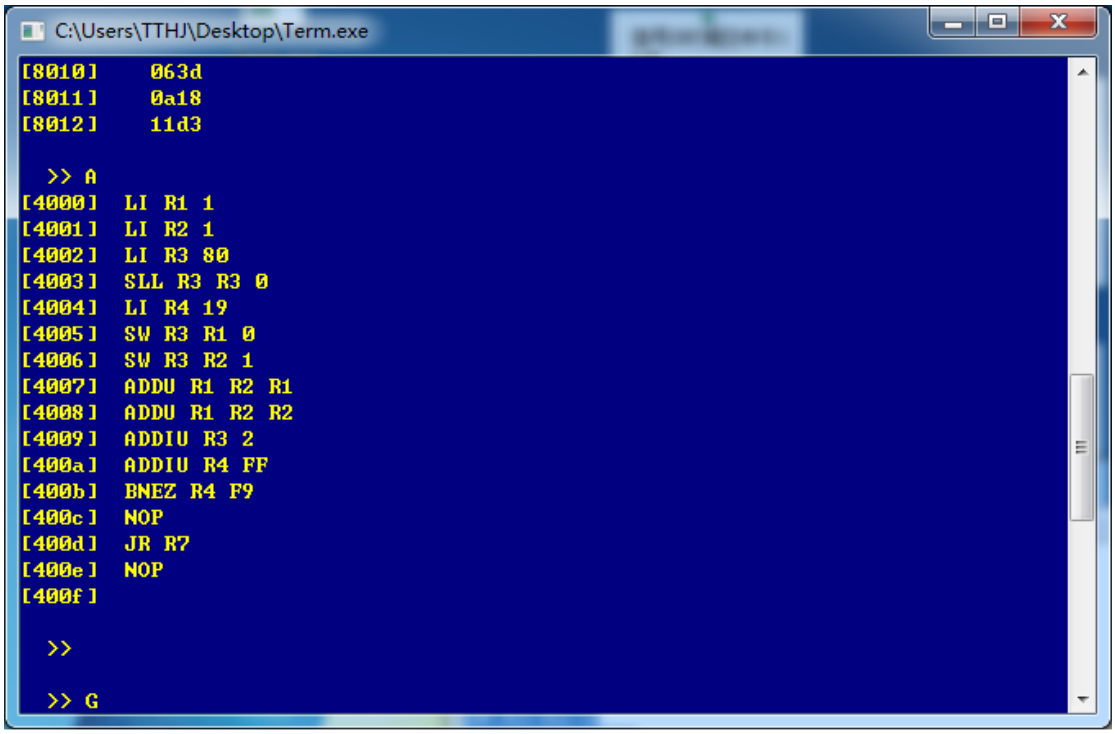
- 清晰的模块分工
- 数据旁路元件
- 冒险检测单元
- FLASH 自启动
- 使用地址映射，统一管理外围 I/O 设备
- 延时槽
- 串口通信
- VGA
  - VGA 等宽 ASCII 字符集显示
  - VGA 双端 FIFO 显存
  - VGA 显示制定字符内容

3.3 实验成果图片展示

- **基本功能实现**

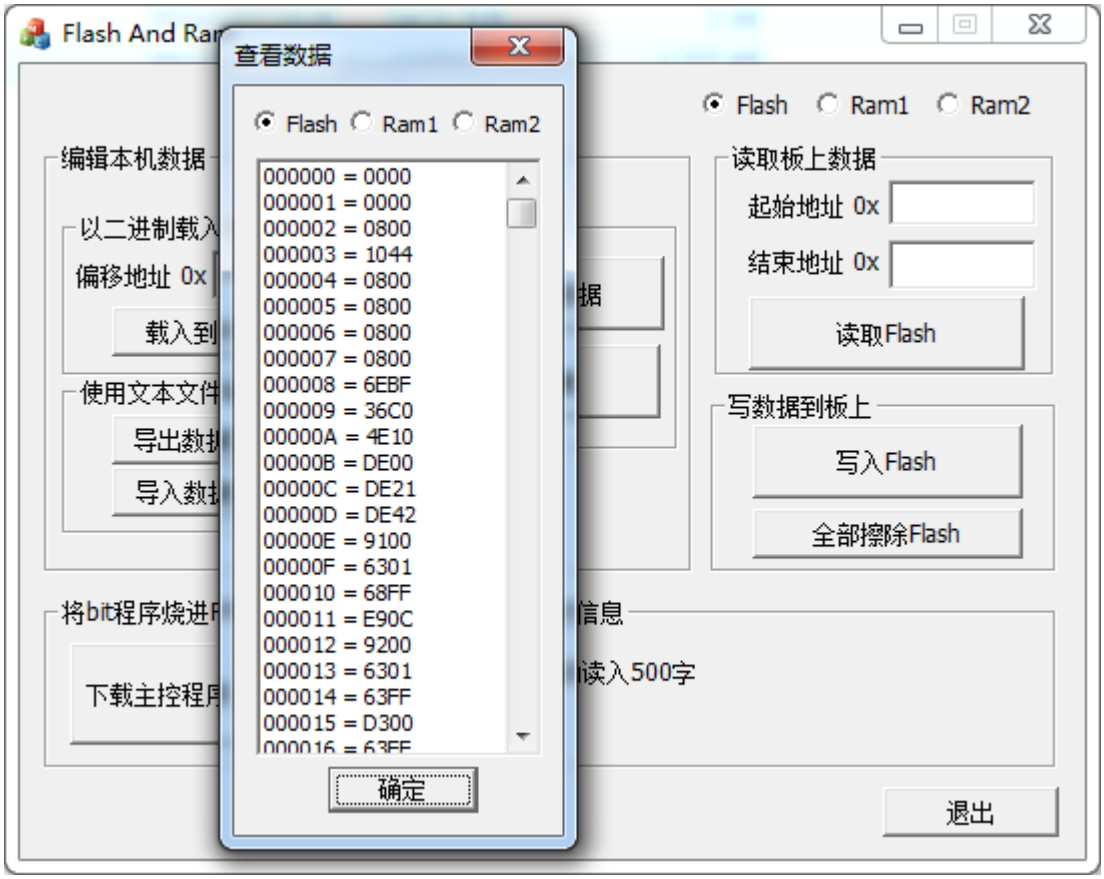
CPU 能够稳定运行 kernel，与 Term 进行正确地串口连接和数据通信。

可以在 CPU 上运行用户程序，如下图是求 fibnacci 数列的代码。运行后可以可以正确得到对应数列的数值。



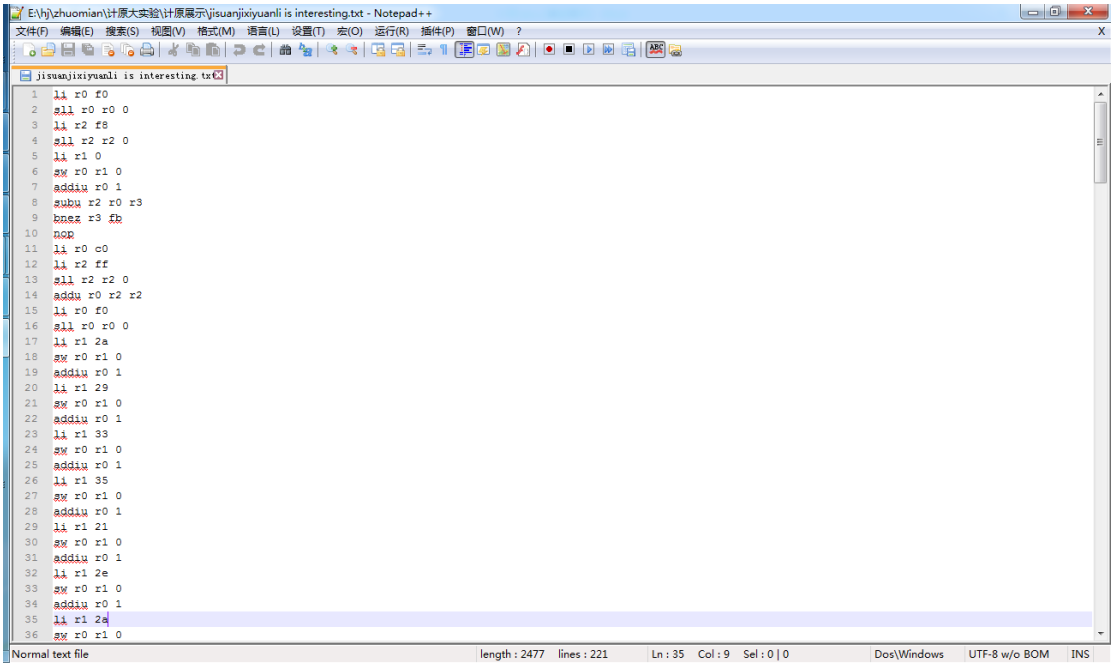
- **实现 flash 自启动**

预先将 kernel 烧入 flash 中。断电后 flash 中内容依旧保存，在下次使用时 CPU 自动将 flash 中的 kernel 载入到 ram 中，实现自启动。



- **VGA 展示**

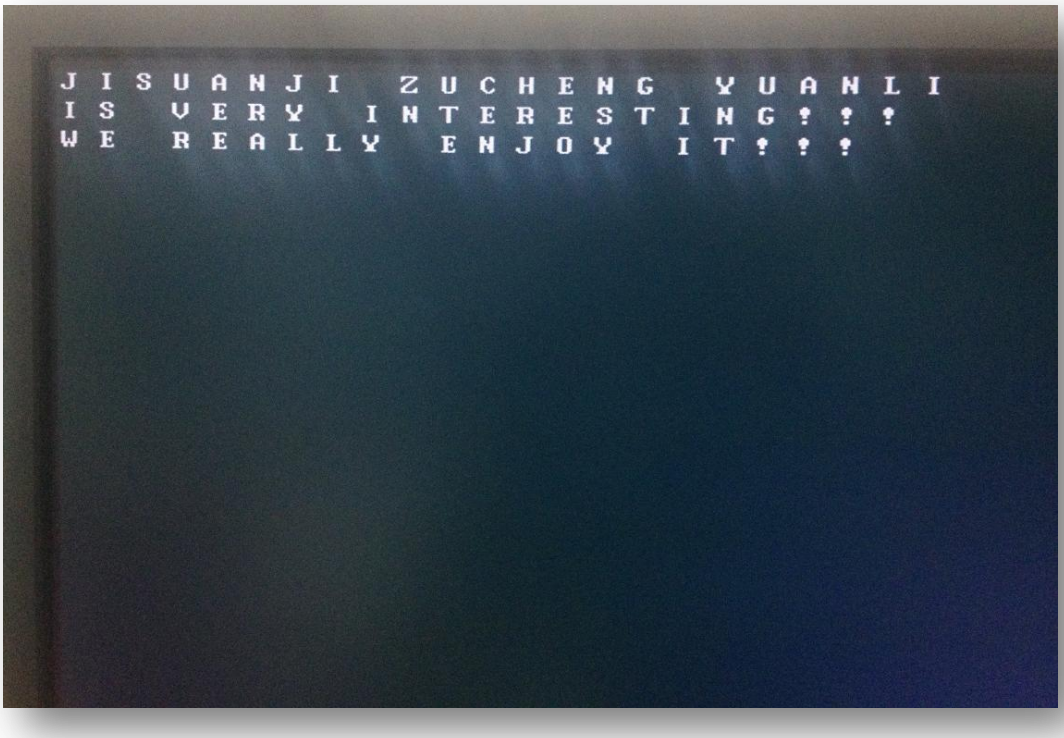
将 ram 中的 0F00 到 0FFF 的地址与显存绑定，在其上的写内存操作直接反映在 VGA 屏幕上。在这里我们测试了一个简单的写入显存的程序：



最终得到 VGA 屏幕显示如下图。

计算机系组成原理 is very interesting!!!

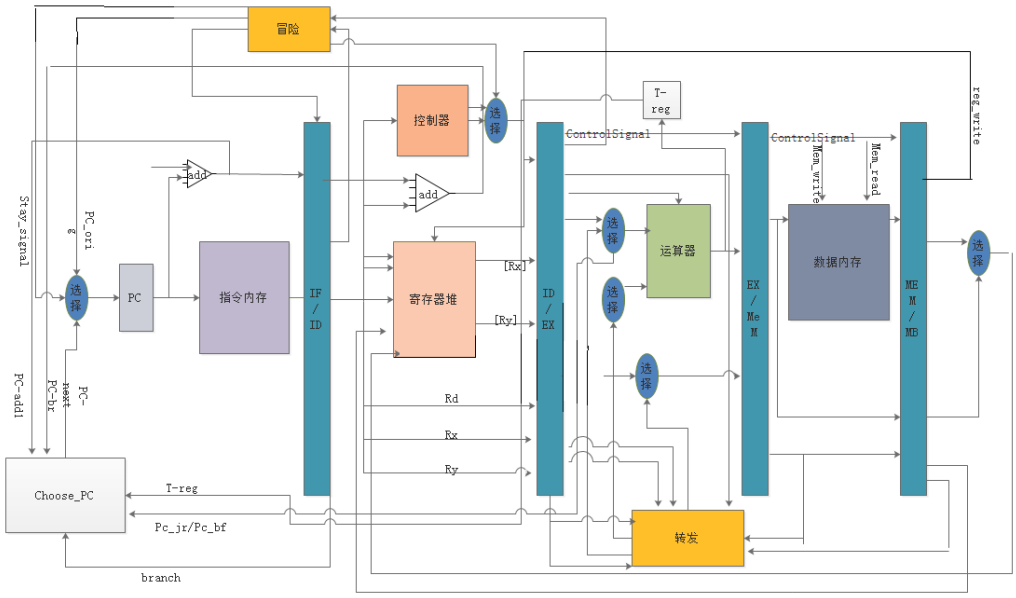
We really enjoy it!!!



四、 架构设计

4.1 数据通路：

我们以课本上介绍的 32 位 MIPS 指令集的流水 CPU 的数据通路为基础，结合我们所要实现的 THCO MIPS 指令集的具体要求，设计出 16 位流水 CPU 的数据通路，经过多次修改和调整，最终数据通路效果图如下：



4.2 结构设计：

将所有需要的工具模块定义为元件，在 cpu.vhd 中例化所有元件，并且按照数据通路的设计连接接口即可。采用这样的思想便于程序的设计和分块实现，组员之间能够更好的协调合作。同时也便利了后期调试与代码修改。

4.3 时序控制：

在 get_pc,IF_ID,ID_EXE,EXE_MEM,MEM_WB，这些阶段寄存器的敏感信号为时钟信号，根据 clk 的 cpu 时钟进行操作。每次时钟发生上升沿跳变就进行一次操作，将上一个阶段的结果传递到下一个阶段中去。mem 模块接收一个频率更高的 mem 时钟，操作内存读写的过程。其他的元件敏感的输入是对应需要的指令、信号等值，这些值发生变化输出的值就对应变化。这种组合逻辑与时序逻辑共同使用的方式，使我们对每一步中流水线中各个元件进行的操作有了更好的理解和把握，便于我们

的代码实现和调试。

五、 模块设计简述

5.1 PC 指令控制——Choose_pc 与 if_stay

在 IF 阶段需要从指令内存中得到指令，这里最为关键的一步是确定此时 PC 的值。Choose_Pc 元件实现了传入上一条指令的信息，能够根据上一条指令的不同情况，选择跳转到下一条指令的 pc 的位置。这里要处理 B、J 等多种指令情况。同时在插入气泡的情况下，我们需要将 PC 恢复到上一个 PC 的值，这里用 if_stay 元件实现。

5.2 译码器——Decode

我们实现的译码器主要作用是解析出指令中的寄存器地址，以便于下一步从寄存器堆中取出数据传递给控制器。此译码器功能比较简单，输入为完整的指令，输出为两个寄存器编号。

5.3 寄存器堆——Regfile

我们的指令一共会用到 11 个寄存器：8 个普通寄存器(0000 – 0100) + sp(1000) + ih(1001) + T 寄存器。其中，为了跳转指令的正确运行，我们将 T 寄存器单独分离出来（详细说明见后面章节），其他寄存器都保存为 16 位的信号数组。Regfile 模块为 Decode 模块的下一级，输入寄存器编号可以输出相应的寄存器中的数据传递给控制器；寄存器堆的写功能由 RegWrite 控制信号控制，在时钟上升沿期间如果 RegWrite 控制信号为 1 则将传入的数据写入到相应的寄存器中。

5.4 控制器模块——ID

指令被取出后，ID 模块对其进行细致解析，主要解析内容为：a. 传递给运算器 ALU 的操作运算码和两个操作数 b.指令用到的寄存器编号，两个读寄存器编号 Rx、Ry(用于数据旁路的需求检验)，以及一个写寄存器编号； c. 控制信号，包括内存读写信号、寄存器写信号，以及判断写入地址是内存还是寄存器的判断信号； d. 用于执行跳转指令的信号和相对应的 pc 值，当解析出的指令是跳转指令时需要传递 PC 值给 Choose_PC。其中，为了实现方便和便于调试，我们对 16 位的指令进行了重新编码，将其映射到 0-29 这 30 个整数，处理模块为 Mapper（见下一部分），在 ID 中会对 Mapper 进行元件例化进行映射。

5.5 指令映射——Mapper

指令为 16 位，不方便进行判断和代码编写，为了简化编写和调试过程，我们将指令进行了处理，将 30 条指令映射到 0-29 这 30 个整数范围上。映射表如下：

指令	编码	指令	编码	指令	编码
NOP	0	SW	10	LW_SP	20
ADDIU3	1	ADDU	11	MFIH	21
AND	2	SUBU	12	MFPC	22
CMP	3	ADDIU	13	MTIH	23
LW	4	ADDSP3	14	SLTI	24
NOT	5	BEQZ	15	SW_SP	25
OR	6	BNEZ	16	MTSP	26
SLL	7	CMPI	17	ADDSP	27
SLLV	8	JR	18	BTEQZ	28
SRA	9	LI	19	B	29

5.6 段寄存器模块

设计实验通路的时候，我们一开始认为段寄存器的功能只是起到连接各处理模块的作用，所以可以不用进行特别的模块实现，后来经过我们的讨论，发现我们之前的思路是有漏洞的，主要从以下两个角度考虑：1）流水结构的一大特点就是同时有 5 个指令运行，段寄存器可以保证将每条指令的运行情况进行正确保存，比如各个阶段的 PC 值； 2）为了保证一些操作的读写顺序能够正确进行，段寄存器很好地保证了整个流水 CPU 的时序逻辑。每个段寄存器设计为时序逻辑部件，只有当时钟到来的时候才进行更新和读取，这样保证了整个系统的稳定。我们用的段寄存器有：if_id，id_exe，alu_exe,mem_wb， 一共四个，每两个阶段之间有一个阶段寄存器。

5.7 数据旁路模块——Passer

流水处理中会遇到数据冲突。每条指令是在 WB 阶段才进行寄存器的写回，如此则可能该条指令尚未执行到 WB 阶段(尚未写回)，下面的指令就需要用到该寄存器更新后的值。由此需要建立数据旁路，把 EXE_MEM 或者 MEM_WB 阶段寄存器中的值转发到相应的地方。PASSER 模块负责判断数据冲突，并根据数据冲突与否以及冲突的类型，生成选择控制信号，传给 EXE 阶段的三路选择器，由三路选择器进行操作数的选择。

5.8 冒险检测模块——ConflictCheck

数据冲突可以通过旁路转发得到，但是也有确实无法拿到的数据以及跳转指令的控制冲突，冒险检测模块主要针对这两类冒险。

第一类是专门针对 LW 指令。LW 的指令必须要 MEM 阶段结束之后才可以获取，若是下一条指令就需要用此值就会发生冲突。因为此时，第二条指令(如 ADDIU)在 EXE 阶段，ALU 需要用寄存器中的值，但第一条指令(LW 指令)还在 MEM 阶段，访存尚未结束。这时候就必须把第二条指令暂停一个周期。

第二类是针对三类特殊的跳转指令：BEQZ、BNEZ 和 JR。按照 MIPS 汇编的编程要求，需要在每个分支跳转语句之后加一个 nop 语句。我们在 ID 阶段就处理分支跳

转语句，获取分支跳转的 PC 值，这样 nop 之后的语句就能够取到正确的指令。但是 BEQZ、BNEZ 和 JR 三条语句，分支跳转 PC 的获取需要用到通用寄存器的值，类似 LW，这些值最快要在语句进行到 EXE 阶段才能通过数据旁路转发得到，因此也需要暂停一个周期。因此，针对 BEQZ、BNEZ 和 JR 三条语句，实质上是做了两次暂停，一是软件上加入的 nop，二是硬件上的强行暂停。

该冒险检测单元的主要作用就是检测以上两种冲突并生成暂停信号，将暂停信号传给 ID 阶段相应部件，由 ID 阶段将控制信号清零，达到暂停一个周期的效果。

5.9 T 寄存器——TReg

仔细研读指令集后，我们发现 BTEQZ 这条指令，PC 的生成并不需要用到通用寄存器(无条件跳转指令 B 也是如此)，因此我们将 BTWQZ(以及 B)指令的处理从 EXE 阶段提前到了 ID 阶段，这样就可以保证在软件上手动加了一个 nop 之后，下一条指令一定能够拿到正确的 PC，减少周期的暂停从而提高运行的效率。为此，我们将 T 寄存器从通用寄存器堆中分离出来，并单独处理。

5.10 Flash 自启动模块——Flash

此次试验中，我们加入了 Flash 自启动的功能。因为 FPGA 是断电即失的，我们将监控程序烧如 Flash，每次开机启动时，CPU 会先将 Flash 中存储的监控程序指令逐一读取并写入到 RAM2 中。如此，一来更接近真正的计算机的启动过程，二来方便了调试。

Flash 模块负责根据输入的 Flash 地址，读入一条指令并输入。该模块只是负责一条指令的取出，具体的 Boot 流程以及写入 RAM 是在 Flash 的上层元件，MemoryController 里处理的。

5.11 内存读写模块——MemoryController

该模块负责内存的访问，包括指令的读取与写入、数据的读取与写入。在实际的操作过程中，内存相关的部分还涉及串口的访问以及我们做的 VGA 的扩展。由于不同模块由不同同学负责，设计、代码编写及模块调试时我们分开进行，但最终实现，为了便于整体调试和调用，我们将内存读写、串口操作、VGA 控制都整合到了一个元件例化中。在 MemoryController 部分，只叙述内存读写相关的内容，串口通信与 VGA 详见下文相关部分。

5.12 串口通信模块——MemoryController_serial

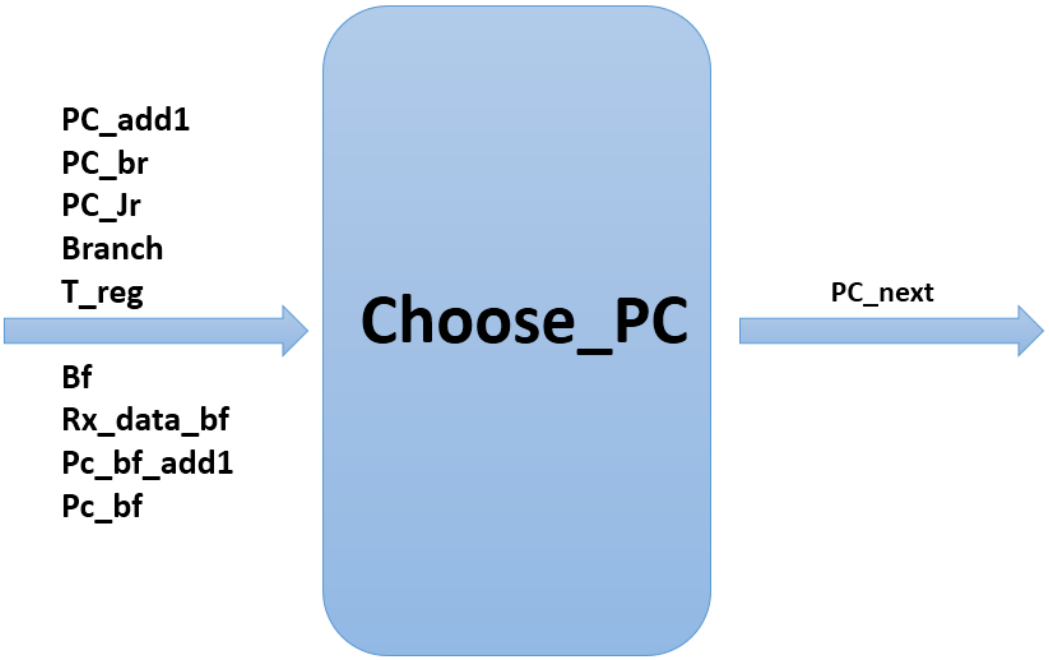
串口通信代码与内存访问代码在同一个模块中，在这里将 BF00 与串口做了一个映射。即访问内存中的 BF00 位置，视为对串口的读写访问。对串口的读写操作方式在前一次的实验中已经有了充分地学习与认识。在串口调试过程中，使用了串口调试精灵作为调试工具。

5.13 VGA 模块——VGA_play

设置 0F00-0FFF 内存位置为显存，往显存中写入的字符能够显示在显示屏幕上。一共能在屏幕中显示横向 40 个，纵向 30 个字符。在内存读写中检测是否为写显存，这部分在 MemoryController 中实现。同时另外例化一个元件实现 VGA 的具体显示功能。

六、 主要模块实现详述

6.1 PC 指令控制——Choose_pc 与 if_stay



上图中的输入输出信号说明：

信号名称	信 号 类型	信号含义
Pc_add1	输入	不跳转的下一条 pc 值
PC_jr	输入	如果是 J 指令跳转情况 pc 值
Pc_br	输入	如果是 B 或 BTEQZ 跳转情况 pc 值
Branch	输入	判断为 b 指令还是 J 指令还是 BTEQZ 或是非跳转指令
T_reg	输入	T 寄存器的值
Bf	输入	判断是否为 BNEZ 或者 BEQZ 的信号
Rx_data_bf	输入	BNEZ 与 BEQZ 中判断的寄存器的值
Pc_bf_add1	输入	如果 BNEZ 与 BEQZ 不跳转的下一条 pc 值
Pc_bf	输入	如果 BNEZ 与 BEQZ 跳转 PC 的下一条的值
Pc_next	输出	输出的下一条 PC 的值

由于下一条 PC 有多种情况，我们区分不同的跳转情况情况，有如下几种：

1）最基本的情况是不跳转，pc=pc+1

2）如果跳转指令是 B 指令，跳转位置为 pc+立即数，在 ID 阶段即可得到立即数的值，从 ID 阶段将信号和值传递到这里。

3）如果跳转指令是 Jr 指令，跳转位置为指定寄存器的值，需要考虑其他指令对该寄存器的修改，所以需要经过旁路解决冲突，同时正确的寄存器的值晚一个周期才能到达，在这里需要加入一个气泡。

4）如果跳转指令是 BTEQZ 指令，跳转位置为 pc+立即数，同时需要 T 寄存器的值，因为 T 寄存器在 ALU 阶段实时可以得到，此时取的 T 值不会产生冲突，不需要使用旁路处理。

5）如果跳转指令是 BNEZ 或 BEQZ 指令，跳转位置为 pc+立即数，需要对指定寄存器的值的判断，所以需要经过旁路解决冲突，正确的寄存器的值晚一个周期才能到达。在这里需要加入一个气泡。

实现代码：

```
entity choose_pc is
    port(
        pc_add1:in std_logic_vector(15 downto 0);
        pc_jr :in std_logic_vector(15 downto 0);
        pc_br : in std_logic_vector(15 downto 0);
        rx_data_bf:in std_logic_vector(15 downto 0);
        T_reg : in std_logic;
        bf    : in std_logic_vector(1 downto 0);
        branch : in std_logic_vector(1 downto 0);
        pc_bf:in std_logic_vector(15 downto 0);
        pc_bf_add1:in std_logic_vector(15 downto 0);
        pc_next : out std_logic_vector(15 downto 0)
    );
end choose_pc;
```

其中 bf 与 branch 为控制信号，而 pc_add1,pc_jr,pc_br 等是对应指令要跳转到的位置。Pc_next 是唯一的输出作为下一条指令的 pc 位置，传给 IF-ID 阶段寄存器。

```
case bf is
    when "01"=>--beqz
        if (rx_data_bf=x"0000") then
            t_pc_next:=pc_bf;
        else
            t_pc_next:=pc_bf_add1;
        end if;
    when "10"=>--bnez
        if (rx_data_bf/=x"0000") then
            t_pc_next:=pc_bf;
        else
            t_pc_next:=pc_bf_add1;
        end if;
    when "00"=>
        case branch is

            when "01"=>--jr
                t_pc_next:=pc_jr;
            when "10"=>--btegz
                if (T_reg='0') then
                    t_pc_next:=pc_br;
                else
                    t_pc_next:=pc_add1;
                end if;
            when "11"=>--b
                t_pc_next:=pc_br;
            when others=>
                t_pc_next:=pc_add1;
        end case;


    when others=>null;
end case;
```

If_stay 代码实现：

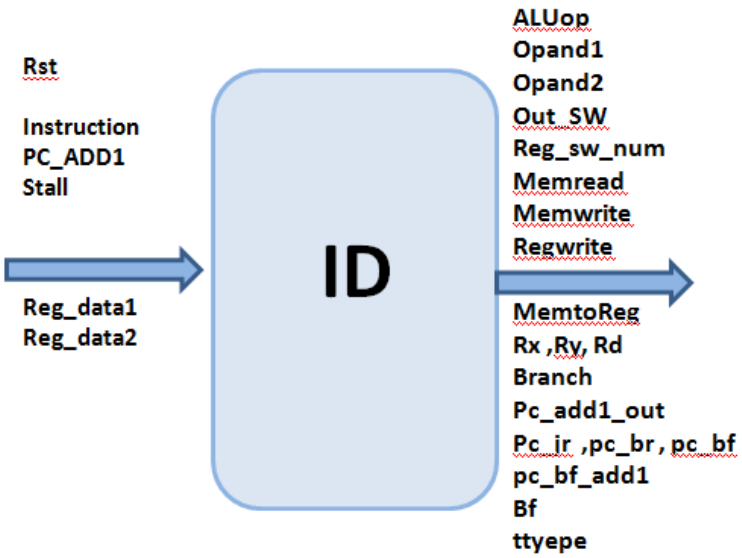
```
entity if_stay is
port(
    pc_in : in std_logic_vector(15 downto 0);
    pc_stay:in std_logic;
    pc_orig:in std_logic_vector(15 downto 0);
    pc_out : out std_logic_vector(15 downto 0)
);
end if_stay;

architecture Behavioral of if_stay is
    shared variable t_pc : std_logic_vector(15 downto 0);
begin

process (pc_stay,pc_in,pc_orig)
    begin
        if (pc_stay='1') then
            t_pc:=pc_orig;
        else
            t_pc:=pc_in;
        end if;
        pc_out <=t_pc;
    end process;
end Behavioral;
```

Pc_orig 是上一条 pc 的值，根据传入的信号判断是否需要将 pc 恢复到上一条 pc 的情况。

6.2 控制器模块 —— ID



上图中的输入输出信号含义如下：

信号名称	信号类型	信号含义
Instruction	输入	需要解析的完整指令
PC_add1	输入	已经计算好的 PC 加 1 的新 PC 值
Stall	输入	中断信号，需要插入 nop 时会用到
Reg_data1、2	输入	由寄存器堆输入的数据，根据指令传递给 Opand 信号
Aluop	输出	传递到运算器的操作码
Opand1/2	输出	传递到运算器的两个操作数
Out_sw	输出	SW 指令特殊用到的传出数据
Reg_sw_num	输出	SW 指令用到的寄存器编号
Memread、Memwrite	输出	内存读写信号
regwrite	输出	寄存器写信号
Memtoreg	输出	写入到寄存器还是内存的判断信号
Rx 、ry	输出	指令需要读取的寄存器编号，用于判断数据冲突需求
Ry	输出	指令需要修改的寄存器编号
Branch	输出	判断跳转指令内容的信号
Pc_add1_out	输出	传递到下一层的 PC 加 1 的新 PC 值
Pc_jr,pc_br Pc_bf,pc_bf_add1	输出	跳转指令和 beqz, bnez 对应的新 PC 值
Bf	输出	判断是 BEQZ 还是 BNEQZ 的信号
ttype	输出	判断用到 T 寄存器指令的内容

控制器模块的输入输出如上图。指令被取出后，ID 模块对其进行细致解析，主要解析内容为：

a. 传递给运算器 ALU 的操作运算码和两个操作数 ；其中，操作码为 3 位的信号，其含义如下：

操作码	含义	操作码	含义
000	ADD	100	NOT
001	SUB	101	SLL
010	AND	110	SRA
011	OR	111	NO

b. 指令用到的寄存器编号，两个读寄存器编号 Rx、Ry 以及一个写寄存器编号；其中，前两项用于数据旁路的需求检验，如果这条指令用到上一条指令写入的寄存器，则由数据旁路进行数据传递。如果指令中读取的寄存器数目小于 2，则将不用的寄存器输出信号每一位置为 1，如

```
case op_link is
when 13 =>--ADDIU
   Opand1 <= Reg_data1;
   Opand2 <= imm8;
   Ry<="1111";
   Rx<=reg1;
   Rd <= reg1;
   tmp_op := ALU_ADD;
```

c. 控制信号，包括内存读写信号、寄存器写信号，以及判断写入地址是内存还是寄存器的判断信号； 信号默认值为 0，对于某一些指令需要进行设置，如以下代码：

```
when 4 | 20 => --LW | LW_SP
   tmp_memread := '1';
   tmp_regwrite := '1';
   tmp_memtoreg :='1';
--MemWrite
when 25|10 => --SW_SP|SW
   tmp_memwrite := '1';
--RegWrite
when 13|1|14|11|12|2 |19|21|22|5|6|7|8|9|27|23|26=>
--ADDIU|ADDIU3|ADDSP3|ADDU|AND|LI|MFIH|NOT|OR|SLL|SLLV|SRA|SUBU|addsp|mtsp
   tmp_regwrite := '1';
   tmp_memtoreg :='0';
```

d. 用于执行跳转指令的信号和相对应的 pc 值，当解析出的指令是跳转指令时需要传递 PC 值给 Choose_PC。在这里用到的指令有:

1）beqz, bnez（这两条的判断信号由 bf 传出）;

```
case op_link is
   when 15=>--beqz
      pc_bf<=pc_add1+imm8;
      pc_bf_add1 <=pc_add1;
      bf <="01";
   when 16=>--bnez
      pc_bf<=pc_add1+imm8;
      pc_bf_add1<=pc_add1;
      bf<="10";
```

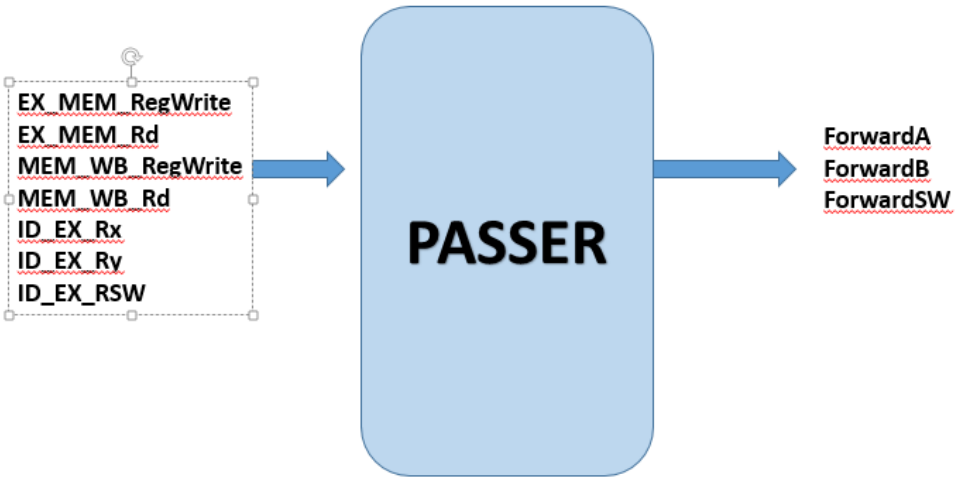
2）B，JR，BTEQZ（这三条的判断信号由 branch 传出）

```
case op_link is
   when 29=>--B
      branch<="11";
      pc_br <=pc_add1 + imm11;
   when 18=>--JR
      branch<="01";
      pc_jr <=Reg_data1;
   when 28=>--BtEQZ
      branch<="10";
      pc_br <=pc_add1+imm8;
```

3）用到 T 寄存器的指令输出 Ttype 传递给 Treg 进行单独处理；

```
case op_link is
   when 3|17=> --CMP CMPI
      ttype<="01";
   when  24=> --SLTI
      ttype<="10";
```

6.3 数据旁路模块——PASSER



数据旁路模块的输入与输出如上图。数据冲突可分为两种类型，一种是 EXE 冲突，即后续指令需要用的寄存器值是该条指令在 EXE 阶段生成的，一般为 ALU 算术运算的结果，在 EXE_MEM 或者 MEM_WB 两个阶段寄存器中都能取到；另一种是 MEM 冲突，即后续指令需要用的寄存器值是该条指令在 MEM 阶段生成的，一般为内存读取的结果，可在 MEM_WB 阶段寄存器中取到。

上图中的输入输出各信号含义如下：

信号名称	信 号 类 型	信号含义
EX_MEM_RegWrite	输入	EXE_MEM 阶段寄存器中的寄存器写信号
EX_MEM_Rd	输入	EXE_MEM 阶段寄存器中的目的寄存器编号
MEM_WB_RegWrite	输入	MEM_WB 阶段寄存器中的寄存器写信号
MEM_WB_Rd	输入	MEM_WB 阶段寄存器中的目的寄存器编号
ID_EX_Rx	输入	ID_EXE 阶段寄存器中的第一源寄存器编号
ID_EX_Ry	输入	ID_EXE 阶段寄存器中的第二源寄存器编号
ID_EX_RSW	输入	ID_EXE 阶段寄存器中针对 SW 指令，储存写入值的寄存器编号
ForwardA	输出	ALU 操作数 A 的选择信号
ForwardB	输出	ALU 操作数 B 的选择信号
ForwardSW	输出	SW 指令写入值的选择信号

两类冒险的判断条件如下——

EXE 类冒险：

```
--检测EXE冒险
if ( EX_MEM_RegWrite = '1'
   AND (EX_MEX_Rd = ID_EX_Rx) ) then
      temp_ForwardA := "10";
end if;

if ( EX_MEM_RegWrite = '1'
   AND (EX_MEX_Rd = ID_EX_Ry) ) then
      temp_ForwardB := "10";
end if;

if ( EX_MEM_RegWrite = '1'
   AND (EX_MEX_Rd = ID_EX_RSW) ) then
      temp_ForwardSW := "10";
end if;
```

由于具体数据是在 EXE 阶段 ALU 才真正需要，所以数据旁路转发的数据直接输入到 ALU 阶段的三路选择器。ALU 阶段共有两个三路选择器(A 和 B)，分别对 ALU 的两个操作数进行选择。进行数据冲突判断时，该条指令正执行到 EXE 阶段，所以判断条件为，上一条指令需要写寄存器(把 ALU 的计算结果写回寄存器堆)，且写入的寄存器编号和该条指令的某个源寄存器编号相同，则可认为有数据冲突，生成相应的转发信号。

MEM 类冒险：

```
--检测MEM冒险
if ( MEM_WB_RegWrite = '1'
   AND NOT( (EX_MEM_RegWrite = '1')  AND (EX_MEX_Rd = ID_EX_Rx))
   AND (MEM_WB_Rd = ID_EX_Rx) ) then
      temp_ForwardA := "01";
end if;

if ( MEM_WB_RegWrite = '1'
   AND NOT( EX_MEM_RegWrite = '1'  AND EX_MEX_Rd = ID_EX_Ry)
   AND (MEM_WB_Rd = ID_EX_Ry) ) then
      temp_ForwardB := "01";
end if;

if ( MEM_WB_RegWrite = '1'
   AND NOT( EX_MEM_RegWrite = '1'  AND EX_MEX_Rd = ID_EX_RSW)
   AND (MEM_WB_Rd = ID_EX_RSW) ) then
      temp_ForwardSW := "01";
end if;
```

产生 MEM 类冒险有两种情况，一是本条指令 ALU 需要用到的寄存器值是上一条指令读取内存得到的，则该值必须在上一条指令完成 MEM 阶段后才能从 MEM_WB 阶段寄存器中得到。另一种情况是，本条指令需要用到的寄存器值是上上条指令 ALU 计算的结果，当本条指令进行到 EXE 阶段时，上上条指令进行到 WB 阶段但是并未完成写回，所以需要转发 MEM_WB 阶段寄存器的值。

数据冲突还值得注意的是，对于连续更新某个寄存器值的情况需要特别考虑。例如，对于如下三条指令：

LI R1 1

LI R1 2

ADDIU3 R3 R1 3

执行第三条指令时，应该拿到的 R1 中的值应该是 2 而非 1，即需要从 EXE_MEM 阶段寄存器中转发，而非从 MEM_WB 阶段寄存器中转发。可以看到，我们上述 MEM 冒险的判断中，相比 EXE 冒险多了一个判断条件，其含义是：如上此条指令和上上条指令发生数据冲突，但是和上一条指令没有数据冲突，才进行 MEM_WB 阶段寄存器的转发。这是因为我们在 PASSER 单元里，先检测 EXE 冒险，再检测 MEM 冒险，两种检测不互斥。这样，针对上述三条指令，则会先生成 EXE 冒险的选择信号，再去检测 MEM 冒险，虽然也检测到了 MEM 冒险，但是不符合“如上此条指令和上上条指令发生数据冲突，但是和上一条指令没有数据冲突”的条件，所以 EXE 冒险选择信号不会被 MEM 冒险选择信号覆盖，所以第三条指令能正确拿到最新的 R1 值（即 EXE_MEM 阶段寄存器中的值）。

另外，相比标准的数据旁路处理，我们增加了一个输入信号 ID_EX_RSW 和一个输出信号 ForwardSW，这是专门针对 SW 指令处理的转发。

SW 指令的指令功能和指令格式如下：

SW rx ry immediate

MEM[R[x]+Sign_extend(immediate)] <- R[y]

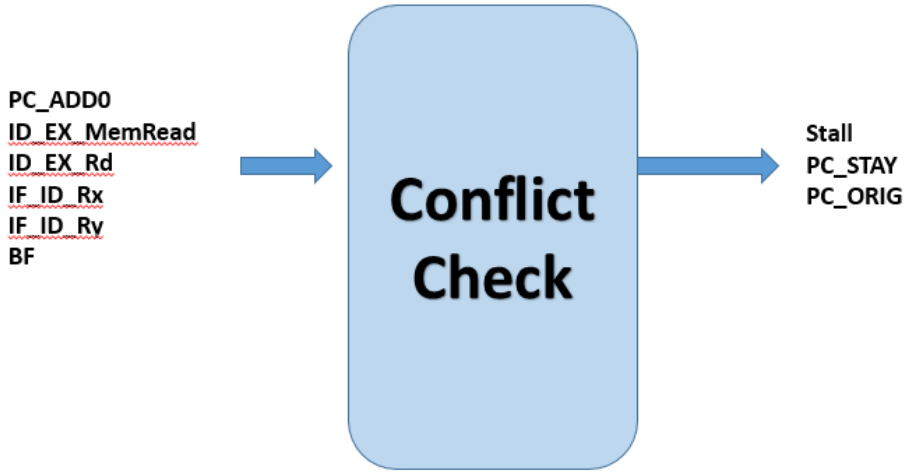
需要在 EXE 阶段用 ALU 计算写入的地址，ALU 的两个操作数分别为 Rx 寄存器的值和立即数。但是要写入的值 Ry 也可能发生数据冲突，而该值不是 ALU 的两个操作数，因为需要为其设置单独的三路选择器和转发信号。

ForwardA、ForwardB 和 ForwardSW 用两位信号表示，其值和对应的含义如下表：

转发信号	转发值所在的源	含义
Forward=00	ID_EXE 阶段寄存器	操作数是正常访问寄存器堆得到的
Forward=10	EXE_MEM 阶段寄存器	操作数由 EXE_MEM 阶段寄存器转发得到
Forward=01	MEM_WB 阶段寄存器	操作数由 MEM_WB 阶段寄存器转发得到

另外需要说明的是，按照标准的 MIPS 实现，0 号寄存器的值始终是 0，所以对 0 号寄存器的操作不应该纳入数据冲突的考虑范围，也就是数据转发时，需要判断冲突的寄存器是否是 0 号寄存器，如果是，则不必转发，因为其值始终是 0。但是参考了监控程序的代码，其中并未考虑 0 寄存器，所以我们在实现时也就没有将其纳入考虑范围。

6.4 冒险检测模块——ConflictCheck



冒险检测模块的输入输出如上图，其中各信号的含义如下表：

信号名称	信号含义
PC_ADD0	上一条指令的 PC 加一后的值
ID_EX_MemRead	ID_EXE 阶段寄存器中的内存读信号
ID_EX_Rd	ID_EXE 阶段寄存器中的目的寄存器编号
IF_ID_Rx	IF_ID 阶段寄存器中的第一源寄存器编号
IF_ID_Ry	IF_ID 阶段寄存器中的第二源寄存器编号
BF	跳转指令的类型
Stall	周期暂停信号
PC_STAY	PC 保持信号
PC_ORIG	即输入的 PC_ADD0 的值

周期暂停信号生成的判断如下，一共有两种情况：

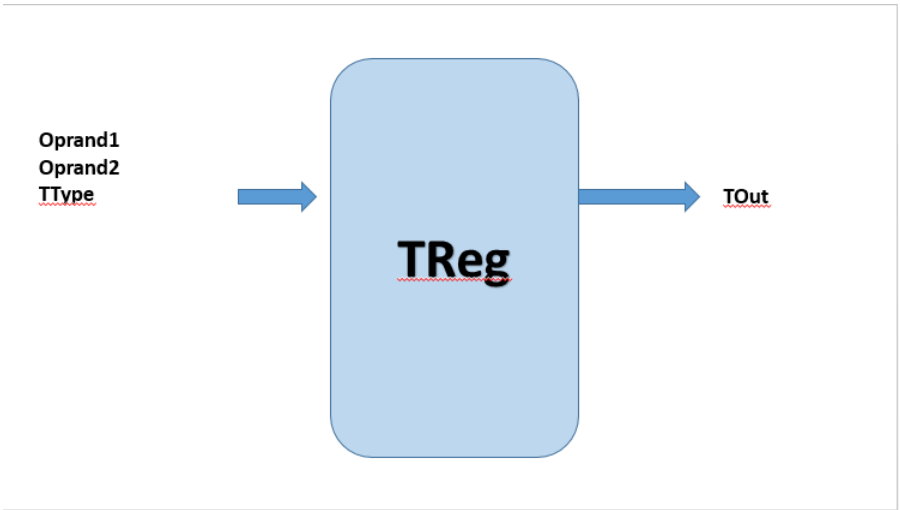
一是，第一条指令是读内存指令(通过 ID_EX_MemRead 信号判断)，第一条指令的目的寄存器编号与第二条指令的其中一个源寄存器编号相同时，发出暂停指令。此时第二条指令正在 ID 阶段，一旦检测到 Stall 信号，就立即将所有控制信号清零，

这样此条指令虽然跑完了，但什么也没做，相当于跑了一条 nop，也就是暂停了一个周期。但是为了达到暂停的效果，三条指令必须把第二条指令从头跑一遍，但是按照正常的流程，每个周期 PC 都会自动加一，为了确保 PC 正确，冒险检测单元还负责暂存第二条指令的 PC，生成 Stall 信号的同时，把老的 PC 和 PC 保持信号一起传给 PC 多路选择器，让 IF 取到正确的 PC。

二是，当第一条指令是 BEQZ、BNEZ、JR 其中之一时，必须暂停一个周期，这由 BF(Branch Flag)来判断，在 ID 阶段译码得到本条指令的类型后，由译码器传入 BF 给冒险检测单元，当 BF 为 01(BEQZ)/10(BNEZ)/11(JR)时，进行暂停。

这样处理可以确保控制冲突得到正确处理，但是还有一个小问题，以上判断逻辑需要从 IF_ID 阶段寄存器就拿到两个源寄存器的编号，但该步骤应该在 ID 阶段处理。好在获取 Rx 和 Ry 花费的时间远小于一个周期，所以我们在 IF 阶段也进行了 Rx，Ry 的获取。

6.5 T 寄存器——TReg

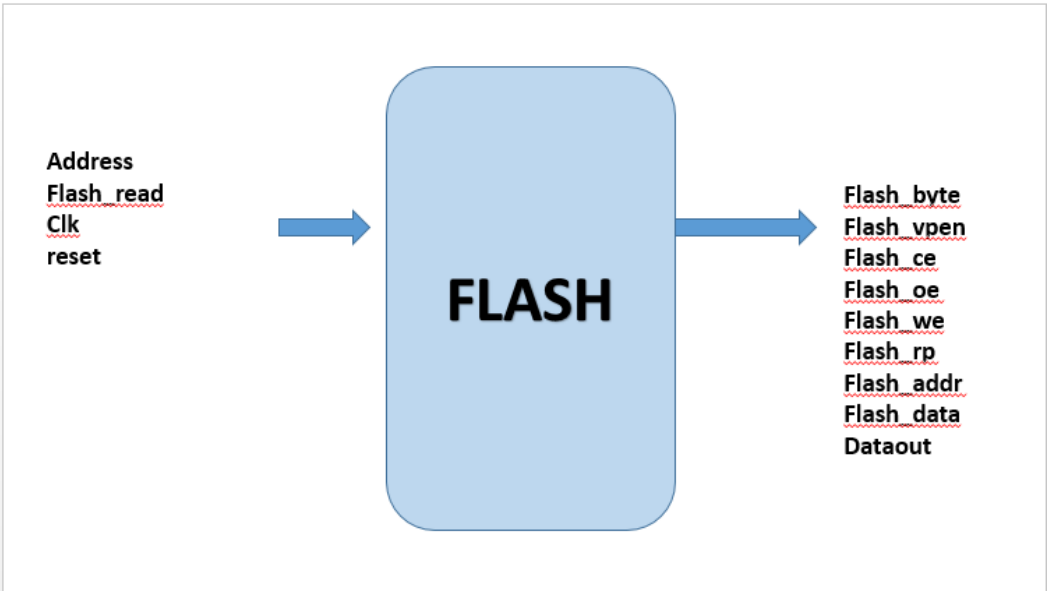


T 寄存器的输入输出如上图所示，各信号解释如下表：

信号名	信号含义
Oprand1	第一操作数
Oprand1	第二操作数
TType	对 T 寄存器操作的类型，0 为 CMP，1 为 SLTI
Tout	T 寄存器的值

TReg 模块处理负责 T 寄存器值的存储之外，还要负责 T 寄存器值的计算。因为需要在 ID 阶段就获取到正确的 T 值，此时上一条指令在 EXE 阶段，因此在 TReg 模块里进行 T 值的计算，并且采用组合逻辑。传入的操作数一旦发生变化，T 的值就立即改变，以此确保 ID 阶段结束时一定能取到正确的 T 值。我们的指令集里设计到 T 寄存器写的指令有 CMP 和 SLTI 两条，根据 TType 判断是哪两条指令，再根据操作数修改 T 值，内部采用一个变量来保存，接到 Tout 输出出去。

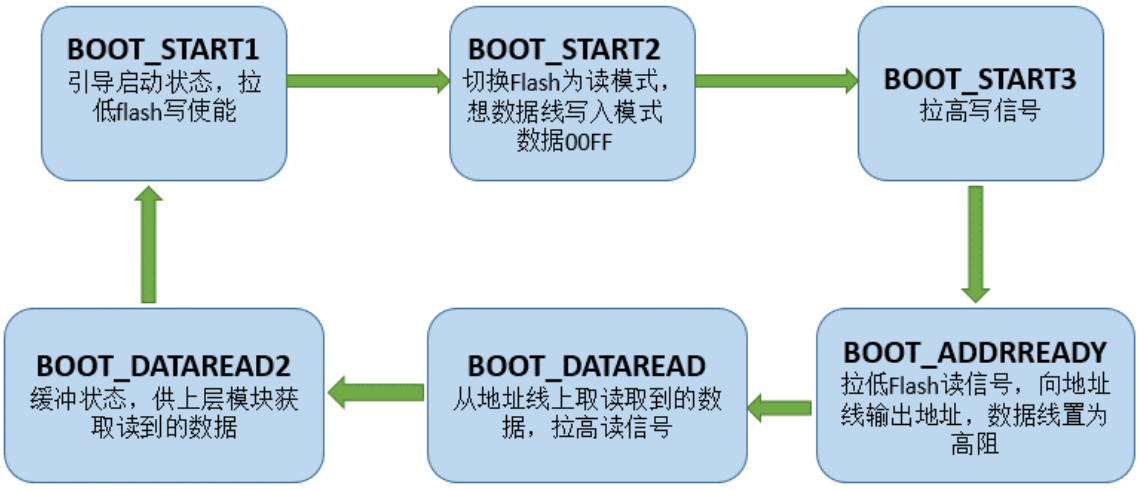
6.6 Flash 自启动模块——Flash



模块示意图如上，其中各信号的含义如下：

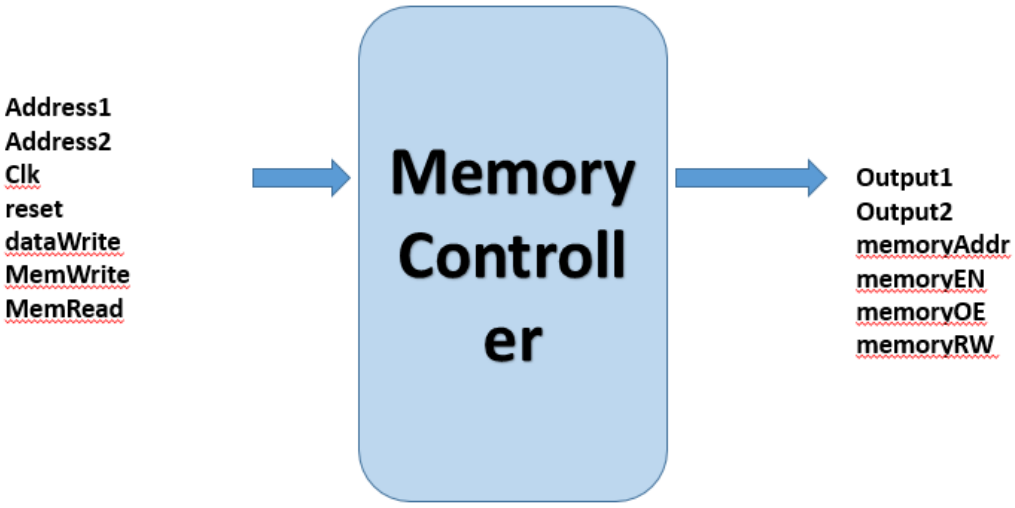
信号名	信号含义
Address	要读取数据的 Flash 地址，有上层模块传入
Flash_read	Flash 读信号，此信号为 0 时读 flash,否则什么也不做，由上层模块传入
Clk	时钟信号
Reset	复位信号
Flash_byte	Flash 操作模式，置为 1，即字模式
Flash_vpen	Flash 写保护，置为 1
Flash_ce	Flash 使能信号，置为 0
Flash_oe	Flash 读使能
Flash_we	Flash 写使能
Flash_rp	Flash 工作模式，置为 1
Flash_addr	输出到 flash 上的地址
Flash_data	Flash 返回的数据
Dataout	传出到上层模块的数据(Flash 读取的内容)

完成一次 Flash 读取需要经历六个状态(步骤)，分别为 BOOT_START1、BOOT_START2 、 BOOT_START3 、 BOOT_ADDRREADY 、 BOOT_DATAREAD 、



BOOT_DATAREAD2，状态机及各状态的含义如下图所示：

6.7 内存读写模块——MemoryController



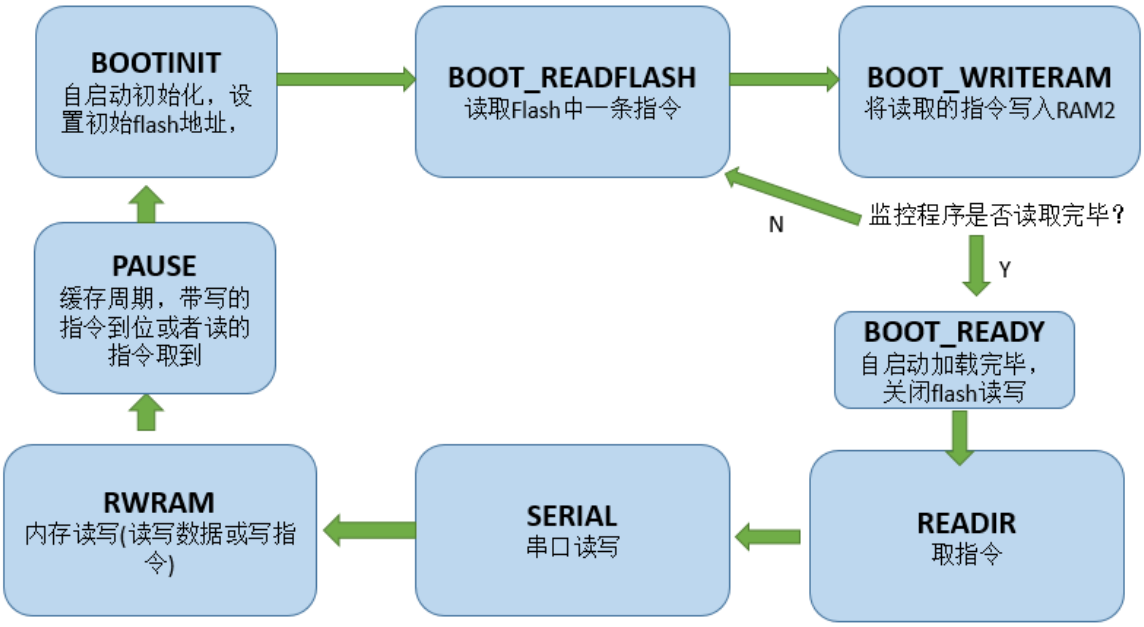
上为模块示意图(只涉及内存读写部分)，其中各信号作用如下表所示：

信号名称	信号含义
Address1	指令地址
Address2	数据地址
Clk	时钟信号
Reset	复位信号
dataWrite	要写的数据(或要写的指令)
MemWrite	内存写信号
MemRead	内存读信号
Output1	读取的指令
Output2	读取的数据
memoryAddr	传到 RAM 上的地址
memoryEN	RAM 片选使能
memoryOE	RAM 读使能
memoryRW	RAM 写使能

按照 MIPS 流水的思想，为了解决结构冲突，应该将数据和指令分开存储，即用 RAM1 存指令用 RAM2 存数据。但是实际操作过程中，我们发现基于教学机的平台并无法解决结构冲突，因为需要从终端 Term 写指令。即使将 RAM1 用于存放指令，RAM2 用于存放数据，由于每个周期都必须取指令,若是同时写指令则会造成 RAM1 总线冲突。同时，用户的程序也不能放在 RAM2，因为这样会和数据发生结构冲突。在加上 RAM1 和串口公用数据总线，也可能造成取指和向终端输出的是发生总线冲突。

基于以上考虑，我们将程序和数据都统一存放在 RAM2 当中，RAM1 的总线专门用于串口通信。

那么如何解决指令读写与数据读写的总线冲突呢？为此，我们将每一次内存相关的访问(包括指令读写、数据读写、串口通信)划分为四个状态，分别是取值、串口读写、数据读写以及一个缓冲状态。每一次访问该模块(不论需要完成什么功能)，都需要经过这四个状态，同时再由 MemRead、MemWrite 等信号来控制是否真的需要进行相应的操作，若需要，则操作，不需要的话就可以直接跳入下一个状态。同时，把 Flash 自启动的模块例化并在该模块中使用，一旦启动，先进行 Flash 加载监



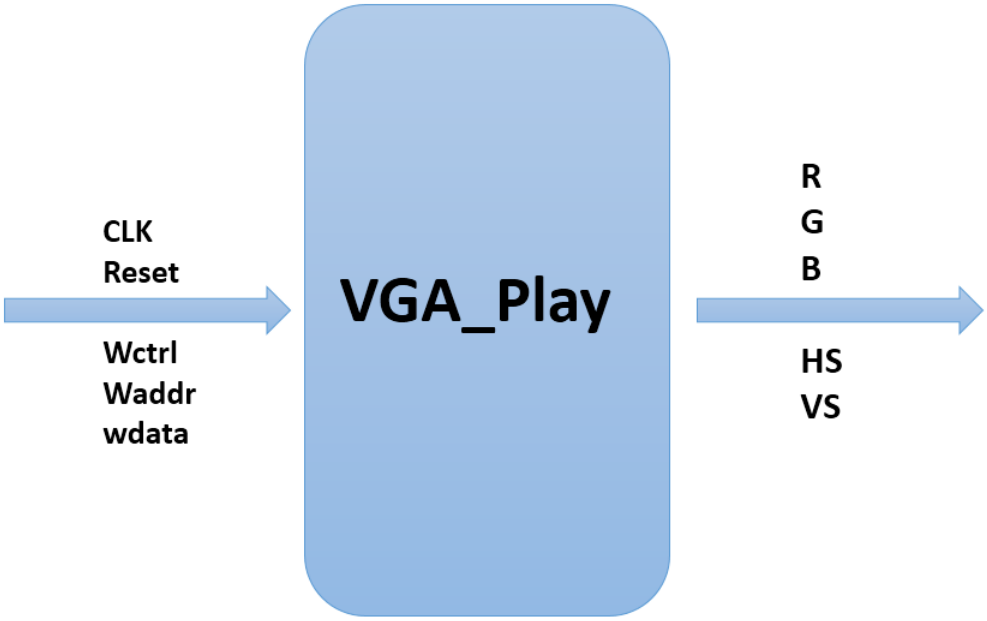
控程序，待加载完成，再进入常规 CPU 循环。整个模块的状态机如下图：

Flash 自启动加载完毕后，cpu 控制的正常内存访问就只包含四个状态。按照如上方法可以解决结构冲突，但是缺点就是，cpu 流水周期长度必须是单项访存操作的四倍。我们将 25M 的时钟加到内存访问部分，则 cpu 的主频只有其四分之一，即 6.25M。经测试，6.25M 主频能够确保无误，把 50M 时钟加到内存访问部分，cpu 主频提高到 12.5M 时，有时不能稳定输出。

6.8 串口通信模块——MemoryController_serial

状态机结构设置与 MemoryController 一致，在 SERIAL 中进行串口读写的准备。在 RWRAM 中判断这次内存读写是否 WieBF00 即是否为串口读写，如果为串口读写就将已经准备好的数据发出，或接受将要传送来的数据。

6.9 VGA 显示模块——VGA_play



输入输出信号：

信号名称	信号含义
Clk	时钟信号-50M
Reset	复位信号
R	RGB 颜色值
G	RGB 颜色值
B	RGB 颜色值
HS	场同步信号
VS	场同步信号
Wctrl	显存写信号
Waddr	显存写地址
Wdata	显存写数据

利用 Xlilinx 的 IPCore 中含有的字模和显存代码，外部设计了一个 VGA_play 的例化加以封装。可以接受 11 位的地址（表示 FIFO 显存的地址）、8 位的数据（表示该位置显示的字符）和 1 位的写使能作为输入，用以确定写入显存的位置和数据。

字模(char_mem):每个可以显示 ASCII 字符得到一个 10*15 的点阵，对其进行零扩展变成 16*16 的存储空间，所以共需要 15*16*95 的存储空间。

显存（fifo_mem）：根据字模的设计，整个 VGA 中每个字符以 16*16 的点阵为单位排布，所以整个 VGA 屏幕横向可以放置 40 个字符，纵向可以放置 30 个字符。对整个显存进行零扩展，使用 64*32*8 作为显存的空间。

零扩展：使用零扩展浪费了一定的存储空间，但是在程序中就可以用向量的拼接、截取来代替乘除运算了。更加快捷方便的计算了显存与字模的地址。

实现代码：

```
-- store char
ram: char_mem port map(clka => clk, addra => char_addr, douta => pr);

-- display cache
cache: fifo_mem port map(
    -- a for write
    clka => clk,
    -- enable, 1 is write signal
    wea => wctrl,
    addra => waddr,
    dina => wdata,
    -- b for read
    clkb => clk,
    addrb => caddr,
    doutb => char
);

-- cache addr 5 + 6 = 11
caddr <= vector_y(8 downto 4) & vector_x(9 downto 4);

-- char acess addr 7 + 4 + 4 = 15
-- last 2 control the display(x, y)
-- first char control which char
char_addr <=  char(6 downto 0) & vector_y(3 downto 0) & vector_x(3 downto 0);
```

代码说明：

其中 Vector_x（0-639）和 Vector_y（0-479）是扫描 VGA 横轴纵轴的变量。通过截取 x 的前 6 位与 y 的前 5 位，确定显存的位置。然后从显存中获取对应位置的字符信息，得到对应字符 16*16 点阵的信息。通过截取坐标后 4 位得到想要控制的 16*16 的点阵的点位置，判断是否点亮。在实现过程中，零扩展带来的无需乘除直接拼接截取的好处显示无疑。

七、遇到的问题和解决

(1) 在取指阶段与内存读写阶段都会对内存有读写操作，但是一片内存不可能同时进行读与写两种操作，会产生冲突。

解决：开始时我们尝试充分利用 RAM1 和 RAM2 两片内存来规避这个问题。认为这样就可以将取指的内存操作与内存读写阶段的内存读写操作独立开来。但是在实现过程中发现，指令内存并不是只读的，也存在我们对指令内存的写操作。这些写入操作必然需要在内存读写阶段完成，所以区分 RAM1 和 RAM2 也不能解决这个冲突。最终我们用统一的 MemoryContorller 来处理这个问题，其实质就是在状态机的不同阶段分别进行读写操作，牺牲了一定的时间解决了冲突的问题。

(2) 在进行取指阶段调试的时候发现取指错误，将指令分别输出到 VGA 与 LED 灯发现同一条指令，VGA 显示与 LED 灯显示结果不以一致。

解决：开始时认为是 VGA 模块存在错误，但是反复调试后发现 VGA 模块单独工作正常。难以解释其与 LED 灯的不同。在反复观察代码后，依旧没有发现逻辑上的错误。花费了大量时间后我们怀疑是对信号的直接赋值导致了信号的不稳定，引起了错误。将传入信号先赋值给一个变量，再赋值给输出信号，就解决了这个问题。硬件语言更加需要我们用一种规范化的形式来编写，避免不必要的错误。

(3) 运行指令时，每次遇到 SW 指令，在其附近会带来错误。

解决：反复观察分析原因之后发现，因为 SW 指令将一个寄存器的值写入到了内存中取，内存位置是另一个寄存器加上一个立即数。我们设计的程序将寄存器与立即数作为 ALU 的两个操作数，而将这个要写入的寄存器直接传递到内存读写阶段，没有进行旁路的处理。这样导致了这个寄存器的值不是最新的值，产生了错误。在分析出错误之后，增加了一个新的旁路信号处理就解决了这个问题。

(4) 在修改完 PC 选择代码后，另一同学调试内存读写阶段，总存在读写有误的情况。

解决：对内存读写阶段反复确认以后发现并没有问题，但难以解释运行 kernel 总是不正确的情况。经过一天反复的讨论和检查后才发现，修改完 PC 代码的同学与另一同学代码传递过程中版本出现了问题。这个问题提醒了我们版本控制的重要性，特别在多人合作的项目中，一个好的版本控制可以节约很多的时间。

(5) 写回指令异常，总是无法正确写会寄存器。

解决：利用 VGA 输出了几乎所有的中间信号，最终发现在 MEM_WB 阶段寄存器向下一阶段传递信号中，寄存器写回信号总是 0，不随指令改变而改变。重新仔细检查了这段代码后发现，MEM_WB 传出的信号名称为 reg_write_mem_wb，而下一个阶段接受的是 regwrite_mem_wb。之前没有发现的原因是在写完代码后调试编译过程中，对于缺失的变量直接在定义中补充，没有仔细思考其中的关联性。

八、 心得体会

8.1 我们的问题

分析遇到的问题，就会发现有一些问题是设计上的问题，但是在开始时候粗略的了解往往只能得到”大概是这样“，”大概是对的吧“的结论，之后当真正调试代码过程中发现了错误，才逐渐发现了设计过程中一个又一个的漏洞。这些漏洞的填补往往花费比写更多的时间,有些漏洞甚至需要对整个模块的架构进行大的修改。随着截止日期的临近，这样大的修改总是让人感觉深深的绝望。

另外还有很多花费了很多时间的问题，是由一个又一个看似小问题带来的。例如变量名拼写错误、程序版本控制错误等，因为编译时间的冗长与调试的困难，这些不容易被发现的小问题往往让我们觉得十分头疼。但其实这些问题一开始就可以通过更好的代码习惯，更好的版本控制习惯来规避的，希望在以后的学习中能够减少这样的问题。

还有一些难以解释的问题，有时候重新烧一变就好了，有时候重启软件就解决了，有时候调整代码串无关紧要的顺序就解决了，我想区别于软件的相对稳定性，这就是硬件的魅力所在吧：）

8.2 我们的经验

我们在调试过程中让我们感到非常有帮助的是 VGA 工具的利用，因为硬件的特殊性，了解运行情况如果只用板子的 LED 灯输出可读性是非常差的，而且需要修改的代码和编译时间都很长，这样会大大降低调试效率。考虑到这一点，我们在基础代码的分工开始就考虑就将 VGA 的显示工作考虑进来了，不得不说这是非常明智的一个选择。

另外，分工与合作的调度的合理性也很重要。我们的主要流程是：一起设计数据通

路，分开编写代码，一起合代码和调试。其中，一起设计数据通路这一点很关键，组里每一位同学对整个数据通路有清楚的认识对于后期的代码合并和调试非常重要。我们在后面发现数据通路存在问题的时候，只需要说明问题，所有成员可以很快地明白问题所在然后修改自己部分的代码，效率很高。

最后，我们想强调课本的重要性。我们在之前的课程学习中主要的学习参考资料是老师给的课件，对课本的重视程度不够。在实验过程中，遇到问题的时候发现都可以从课本上找到细致的解释。所以，刘卫东老师要求我们课下认真学习课本还是有道理的啊！毕竟长者的经验：）

8.3 收获与感想

整体下来，造计算机的过程真是让人又恨又爱，悲喜交加。怎么找也找不到的 bug，临近 ddl 发现数据通路的漏洞，刷夜的辛劳和凛冽寒风中从 FIT 到宿舍的归程…. 我们在造机的过程中真是无时无刻不在承受着身体和心理的考验。但是同时！代码跑成功时的欢呼，每一个问题解决后的豁然开朗，每一个知识漏洞在调 bug 的过程中逐一补全，这些收获和快乐又是那么实实在在，让人觉得之前一切痛苦都得到了完美的诠释。

另外，我们的意志和团队精神也得到了很大的提升。我们组的 3 个成员平日关系极好，又因同在辩论队形成了高度的默契感。遇到问题的时候，我们整个团队都调动高度的乐观态度和积极性去面对，中间还喊出了“调不出来我们就把板子吃下去！”的口号，每个人都用自己的乐观感染着别人，慢慢长夜，因为有相互的鼓励而不觉得辛苦！

辛苦而充实，痛并快乐着，这就是我们对整个造机过程的总体感想吧！

九、感谢

在此特别感谢矣晓沅的妈妈，每次都半夜结束后辛苦已经睡下的阿姨再爬起来….
阿姨还经常给我们提供食物和热水，给我们强大的后勤支持！

PS：附一张认真工作图：）

计算机组成原理大实验 实验报告

309 组

2012011302 陈伯乐

2012011318 何蔚然

2012011324 党唯真

目录

一、数据通路展示及讲解

数据通路图经过数次重构，最终版本见下面完整图示。

由图所示，四个灰色竖长条分别为 IF/ID、ID/EX、EX/MEM 和 MEM/WB 四个级间寄存器，其隔开的五个区域则分别为 IF、ID、EX、MEM 和 WB，其详细职责及设计思想将在后文进行阐述。

以下将选取六条具体指令作为代表，对数据通路的工作流程进行解释。

1) ADDIU

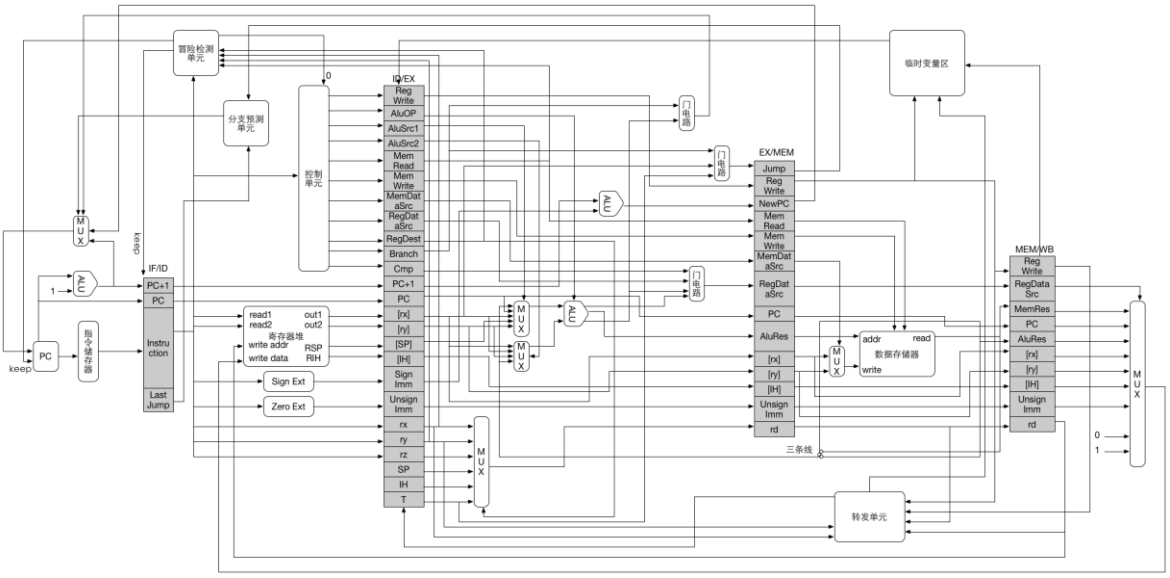
2) BEQZ

3) LW

4) SW

5) SLTU

6) NOP



处理每条指令的时候，第一步均为利用指令储存器，根据 PC 值取出指令，并存储在 IF/ID 级间寄存器中，同时会传递 PC 和 PC+1 的值。LastJump 用于分支预测单元，将在后文进行解释说明。以下每条指令均默认已进行完毕上述步骤。

由于冒险检测单元、分支预测单元、转发单元等工作模式较为固定，下文解释指令时也将不特定对这些单元的工作流程进行描述，只关注于最主要的工作流程。

1) ADDIU

ID 级：

Instruction 中存储的指令传输到控制单元，控制单元识别出该指令后，由于不涉及内存读写，故对 MemRead 和 MemWrite 赋值为 0；该指令没有跳转操作和比较操作，Branch 和 Cmp 分别赋值为 0 和 00，表示无作用；最后写回寄存器的值由 ALU 产生，所以控制多路选择器返回信息选择的 RegDataSrc 赋值为 000，表示选择 AluRes；同样道理，最后写回到寄存器 Rx 中，故 RegDest 赋值为 000，表示 Rx；而所使用的 ALU 运算方法为加法，故赋值 AluOP 为 000，表示加法；而 AluSrc1 和 AluSrc2 则是指定 EX 级中 ALU 前两个多路选择器所应选择的信号。

而同时，立即数 Immediate 经过 Sign Ext 和 Zero Ext 两个操作单元后，分别得到符号扩展和零扩展后的数值，分别存入 Sign Imm 和 Unsign Imm 两个级间寄存器中。这里，Unsign Imm 当中的值是我们想要的。

至于寄存器堆，我们将对 read1 和 read2 分别赋值，使其向 ID/EX 级间寄存器堆的[rx]、[ry]、[SP]和[IH]分别赋值为相应寄存器的当前取值。

EX 级：

根据 AluSrc1 和 AluSrc2 的结果，ALU 前第一个多路选择器选择 Sign Imm 作为输出，第二个多路选择器选择 Rx 作为输出。而 ALU 根据 AluOP 选择加法对两个多路选择器的输出进行运算，结果放入 AluRes，同时生成 Zero 和 Sign 两个信号，分别表示结果是否为 0 以及其符号位。其中，只有 AluRes 被写入 EX/MEM 级间寄存器堆中，其余两个信号 Zero 和 Sign 则用于分支预测单元和协助冒险检测单元。

同时，下方的多路选择器将根据 RegDest 选择 rx 作为最后写回寄存器堆的寄存器，并将这一信息赋值给 EX/MEM 中的 rd。

MEM 级：

根据 MemRead 和 MemWrite，数据存储器判断本指令不需要对内

存进行读写，故可以忽略 addr 和 write 的输入，输出 MemRes 约定为 0。

其余寄存器按数据通路图示原样传输到 MEM/WB 级间寄存器堆中。

WB 级：

根据 RegDataSrc，WB 级的多路选择器选择 AluRes 作为输出，传递到 ID 级寄存器堆的 write data 中，表示指令运算结果。同时，保存写回寄存器信息的 rd 也传递到 ID 级寄存器堆的 write addr 中，表示写入寄存器地址，即指定寄存器为 rx。

寄存器堆收到信号后将运算结果保存到寄存器 Rx 后，该指令的所有工序都已完成。

2) BEQZ

ID 级：

Instruction 中存储的指令传输到控制单元，控制单元识别出该指令后，由于不涉及内存读写，故对 MemRead 和 MemWrite 赋值为 0；由于需要判断 R[x]是否为 0，若 R[x]=0，那么需要更新 PC 值为 PC+Sign_extend(immediate)，所以 Branch 赋值为 10，表示判 0 操作；由于不是 Compare 类指令，所以 Cmp 赋值为 00，表示无作用；最后只需要更新 PC 值，而不需要让寄存器堆写入新的值，所以控制多路选择器返回信息选择的 RegDataSrc 赋值为 1111，表示无用信号；同样道理，我们给 RegDest 赋值为 111，表示没有对应的目标寄存器；而所使用的 ALU 运算方法为加法，故赋值 AluOP 为 000，表示加法；而 AluSrc1 和 AluSrc2 则是指定 EX 级中 ALU 前两个多路选择器所应选择的信号。

同时，立即数 Immediate 经过 Sign Ext 和 Zero Ext 两个操作单元后，分别得到符号扩展和零扩展后的数值，分别存入 Sign Imm 和 Unsign Imm 两个级间寄存器中。这里，Sign Imm 当中的值是我们想要的。

至于寄存器堆，read1 和 read2 的信号值将无关紧要，因为该指

令不需要往后传递任何目标寄存器。

EX 级：

根据 AluSrc1 和 AluSrc2 的结果，ALU 前第一个多路选择器选择 PC 作为输出，第二个多路选择器选择 0 作为输出。而 ALU 根据 AluOP 选择加法对两个多路选择器的输出进行运算，结果放入 AluRes，同时生成 Zero 和 Sign 两个信号，分别表示结果是否为 0 以及其符号位，该指令中我们只关心 Zero 的取值，Sign 所属的门电路输出信号将无关紧要。

然后最上方逻辑门电路中，取 Branch 和 Zero 作为输入，如果 R[x]确实等于 0，则 Zero 为高电位。配合 Branch，给 PC 的多路选择器发出一个信号，该信号将和 Jump 合作以决定新的 PC 值到底是跳转前的 PC 还是跳转后的 PC。而后，下方的门电路取 Branch、R[x]和 T 作为输入，这其中只有 Branch 和 R[x]为有效信号，计算本次跳转是否要被执行，并赋值到 Jump 中——如果当真要跳转，则 Jump 被赋值为高电位；否则如果这次跳转不执行，则 Jump 被赋值为低电位。然后 Jump 将发送到 PC 的多路选择器中，以在下一个周期决定是否给 PC 赋值为新的 PC 值。

同时，下方的多路选择器将根据 RegDest 选择 rx 作为最后写回寄存器堆的寄存器，并将这一信息赋值给 EX/MEM 中的 rd。但该指令中，rd 值没有意义，将在下一周期内被当做 0 传递，即无作用。

新的 PC 值将由上方的 ALU，取 PC+1 和 Sign Imm 作为输入，对两者做有符号加法，其运算结果即为新的 PC 值。这个结果将被赋值到 EX/MEM 级间寄存器堆的 NewPC 寄存器中，并在下一个周期发送到 PC 的多路选择器中作为其中一个待选值。

MEM 级：

根据 MemRead 和 MemWrite，数据存储器判断本指令不需要对内存进行读写，故可以忽略 addr 和 write 的输入，输出 MemRes 约定为 0。

其余寄存器按数据通路图示原样传输到 MEM/WB 级间寄存器堆中。

但同样在该指令中没有意义，都以 0 作为输出。

WB 级：

根据 RegDataSrc，WB 级的多路选择器选择 AluRes 作为输出，传递到 ID 级寄存器堆的 write data 中，表示指令运算结果。同时，rd 也将如上一条指令一样被传递。但由于该指令不需要这两个信息，故他们都被约定为 0，即表示无作用的低电位。这样即便它们被传递，也不会对整个系统的数据造成任何改变。

PC 的多路选择器这时有 PC+1 和 NewPC 两个备选输入值，以及分支预测单元和上述门电路的选择输入。这时将根据计算结果判断是否跳转。若跳转，则 NewPC 被选择输出，否则输出 PC+1。然后 PC 的值将被这个输出所改变（当然，如果冒险检测单元对其输出保持信号，则还要进一步推导）。

3) LW

ID 级：

Instruction 中存储的指令传输到控制单元，控制单元识别出该指令后，由于需要读内存，所以 MemRead 被置为高电位；而此指令不需要写内存，所以 MemWrite 为 0；同样，由于不包含跳转操作，所以 Branch 为 0；也不是 Compare 类指令，所以 Cmp 赋值为 00，表示无作用；最后需要对 R[x]+Sign_extend(immediate)这个地址进行取值，所以 RegDataSrc 置为 001，表示对 ALU 的计算结果当作地址取值；而取出来的值要赋给 R[y]，所以 RegDest 赋值为 001，表示 Ry；而所使用的 ALU 运算方法为加法，故赋值 AluOP 为 000，表示加法；而 AluSrc1 和 AluSrc2 则是指定 EX 级中 ALU 前两个多路选择器所应选择的信号。

同时，立即数 Immediate 经过 Sign Ext 和 Zero Ext 两个操作单元后，分别得到符号扩展和零扩展后的数值，分别存入 Sign Imm 和 Unsign Imm 两个级间寄存器中。这里，Sign Imm 当中的值是我们想要的。

至于寄存器堆，我们赋值 read1 和 read2 使其能取出 Rx 的值，但对 Ry 当前的值我们并不关心。

EX 级：

根据 AluSrc1 和 AluSrc2 的结果，ALU 前第一个多路选择器选择 Sign Imm 作为输出，第二个多路选择器选择 Rx 作为输出。而 ALU 根据 AluOP 选择加法对两个多路选择器的输出进行运算，结果放入 AluRes。本指令中我们不关心 Zero 和 Sign 两个信号。

同时，下方的多路选择器将根据 RegDest 选择 ry 作为最后写回寄存器堆的寄存器，并将这一信息赋值给 EX/MEM 中的 rd。

MEM 级：

根据 MemRead 的高电位和 AluRes，数据存储器取出对应内存地址的值，并输出到 MemRes 中。

WB 级：

根据 RegDataSrc，多路选择器选择 MemRes 作为输出，传递到 ID 级的寄存器堆中，配合 rd，指定寄存器堆的 Ry 作为写入寄存器。

这一周期把数据写入到 Ry 后，该指令所有流程都已完成。

4) SW

ID 级：

Instruction 中存储的指令传输到控制单元，控制单元识别出该指令后，由于需要写内存，所以 MemWrite 被置为高电位；而此指令不需要读内存，所以 MemRead 为 0；同样，由于不包含跳转操作，所以 Branch 为 0；也不是 Compare 类指令，所以 Cmp 赋值为 00，表示无作用;因为需要取 R[y]这个地址的值,所以 RegDataSrc 置为 101；而取出来的值要赋给 R[y]，所以 RegDest 赋值为 001，表示 Ry；而所使用的 ALU 运算方法为加法，故赋值 AluOP 为 000，表示加法；而 AluSrc1 和 AluSrc2 则是指定 EX 级中 ALU 前两个多路选择器所应选择的信号。

同时，立即数 Immediate 经过 Sign Ext 和 Zero Ext 两个操作单

元后，分别得到符号扩展和零扩展后的数值，分别存入 Sign Imm 和 Unsign Imm 两个级间寄存器中。这里，Sign Imm 当中的值是我们想要的。

至于寄存器堆，我们赋值 read1 和 read2 使得 Rx 和 Ry 被输出。

EX 级：

根据 AluSrc1 和 AluSrc2 的结果，ALU 前第一个多路选择器选择 Sign Imm 作为输出，第二个多路选择器选择 Rx 作为输出。而 ALU 根据 AluOP 选择加法对两个多路选择器的输出进行运算，结果放入 AluRes。本指令中我们不关心 Zero 和 Sign 两个信号。

该指令中 rd 无作用，传输低电位即可。

MEM 级：

把 AluRes 作为 addr 传入数据存储器中，多路选择器根据 MemDataSrc 选择 Ry 作为写入数据，而数据存储器根据 MemWrite 的高电位选择向 Rx+Sign Imm 这个地址写入 Ry 的内容。

至此该指令完成。

WB 级：

本指令中该级无作用。

5) SLTU

ID 级：

Instruction 中存储的指令传输到控制单元，控制单元识别出该指令后，由于不需要读写内存，所以 MemWrite 和 MemRead 被置为低电位；同样，由于不包含跳转操作，所以 Branch 为 0；需要对 Rx-Ry 判断大小，所以是 Compare 类指令，Cmp 赋值为 01，表示判正；最后赋值给 T 非 0 即 1，所以 RegDataSrc 置为 111；而取出来的值要赋给 T，所以 RegDest 赋值为 100，表示 T；而所使用的 ALU 运算方法为减法，故赋值 AluOP 为 001，表示减法；而 AluSrc1 和 AluSrc2 则是指定 EX 级中 ALU 前两个多路选择器所应选择的信号。

对寄存器堆的 read1 和 read2 赋值，使其输出 Rx 和 Ry 到 ID/EX

级间寄存器堆中。

EX 级：

根据 AluSrc1 和 AluSrc2 的结果，ALU 前第一个多路选择器选择 Ry 作为输出，第二个多路选择器选择 Rx 作为输出。而 ALU 根据 AluOP 选择减法对两个多路选择器的输出进行运算，结果放入 AluRes，同时生成 Zero 和 Sign 信号。中间的门电路根据 Sign 和 Cmp 选择到底输出 1 还是 0，并把选择信息传递到下一级的 RegDataSrc。

Rd 通过多路选择器选择 T 并传递到下一级中。

MEM 级：

不需要对内存进行读写，故数据存储器在本级无作用。

本级只需要将上一级寄存器信息传递到下一级即可。

WB 级：

根据 RegDataSrc，多路选择器选择传输 0 还是 1 作为输出，传递到 ID 级的寄存器堆中作为内容。同时 rd 指定寄存器 T 并传递到寄存器堆。

等寄存器堆把内容写入到寄存器 T 后，该指令执行完毕。

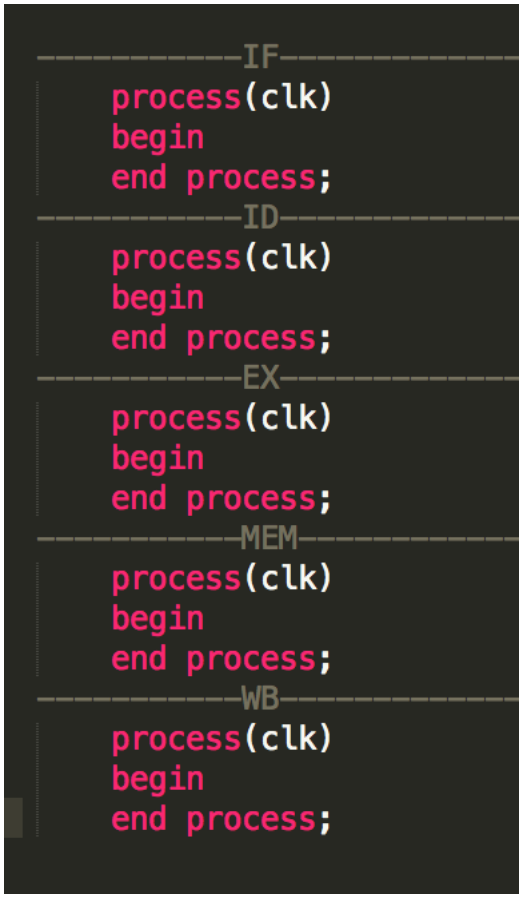
6) NOP

该指令在需要跳转或有别的需求时使用，基本原理很简单，即在所有功能模块中都传递低电位，并约定这样为无作用。这时 ALU 仍会正常工作，但考虑到 0+0=0，最后只会往 0 号寄存器写入 0，这样对整个系统的正常工作无影响。

二、五级流水线 CPU 整体设计思路

我们要以流水线方式实现一个面向指定 16 位指令集的处理器。依照 MIPS 经典五级流水结构,程序的总体框架应分为读取指令(IF 阶段)、指令译码(ID 阶段)、ALU(EX 阶段)、读写内存(MEM 阶段)以及写回寄存器(WB 阶段)。五级流水按照相同并且对齐的时钟周期进行彼此独立的工作,可以用五个 process 监控同

一个时钟信号以达到独立并且同步工作的效果。因此程序的整体结构应为下图所示：



五级流水的工作周期相同，所以 CPU 的周期就由正常工作所需时间最长的读写内存 MEM 阶段决定，MEM 阶段可能进行一次读操作和写操作，这个过程需要监控 5 个时钟上升沿，因此我们每一级流水的工作周期为 5 个时钟周期。也就是当我们的 CPU 接入 50MHZ 的时钟时，CPU 可以获得 10MHZ 的主频。

同 MIPS 经典五级流水结构相似，处理器需要流水寄存器暂存指令和数据的值。我们设置了 IF/ID、ID/EX、EX/MEM、MEM/WB 五个流水线寄存器用于保存指令的值和控制信号。程序结构如下图所示：

```
------------------------I_F/ID--------------------------------
signal IF_ID_PC           : std_logic_vector(15 downto 0);
signal IF_ID_PC_1         : std_logic_vector(15 downto 0);
signal IF_ID_instruction  : std_logic_vector(15 downto 0);
...
------------------------ID/EX---------------------------------
signal ID_EX_PC_1         : std_logic_vector(15 downto 0);
signal ID_EX_PC           : std_logic_vector(15 downto 0);
signal ID_EX_rx           : std_logic_vector(15 downto 0);
...
------------------------EX/MEM--------------------------------
signal EX_MEM_NEWPC       : std_logic_vector(15 downto 0);
signal EX_MEM_PC          : std_logic_vector(15 downto 0);
signal EX_MEM_alures      : std_logic_vector(15 downto 0);
...
------------------------MEM/WB--------------------------------
signal MEM_WB_memres      : std_logic_vector(15 downto 0);
signal MEM_WB_PC          : std_logic_vector(15 downto 0);
signal MEM_WB_alures      : std_logic_vector(15 downto 0);
...
```

每一级流水阶段在自己的第一个时钟周期从自己与之前流水级之间的流水寄存器中取得正常工作所需的数据与信号，在自己的第五个时钟周期向自己与下一级流水之间的流水寄存器中写入下一级流水阶段所需的数据和信号。这样可以保证在不考虑冲突的情况下各流水级之间传递的指令和数据是完全正确的。

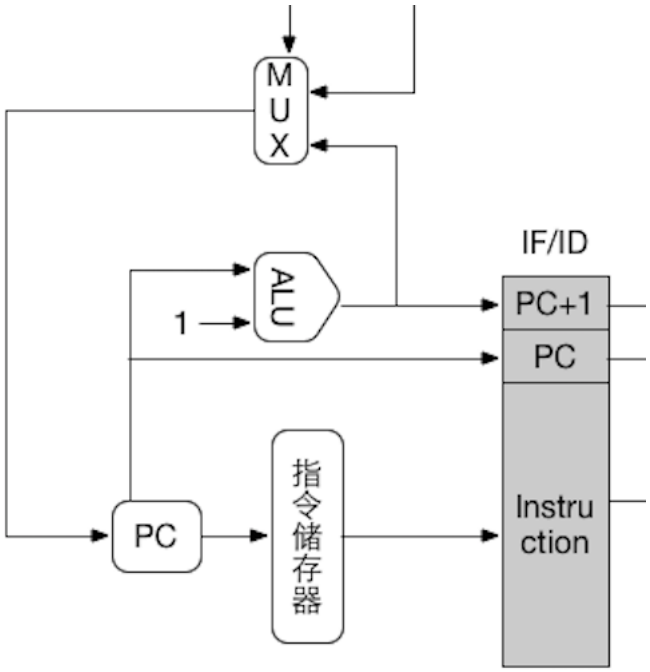
考虑到五级流水结构不可避免的数据冒险和控制冒险，要解决冲突我们需要增加转发单元和冒险检测单元来分别解决上面提到的这两类冒险。为了使程序结构更加清晰，便于理解和调试。除了上述提到的部分，我们将 ALU 模块、控制单元模块、访存模块、分支预测模块用元件例化的方式实现。这些模块都会在之后的报告中详细说明。

另外在整个五级流水 CPU 的程序实现中，五级流水阶段和访存模块是时序逻辑单元，转发单元、冒险检测单元、ALU 单元、控制单元和分支预测单元是组合逻辑单元。

分级具体说明：

1、取指令阶段（IF）：

该阶段的工作流程如下图所示：



在第一个周期里PC得到下一条指令的地址，如果上一条指令不是跳转指令，那么 PC = PC + 1，即顺序执行下一条指令。如果上一条指令是跳转指令，IF流水级会根据上次跳转的成功与否选择顺序执行指令或是执行跳转至后的指令。可见下图程序说明：

```
if ID_EX_branch = "00" then
    IF_PC <= IF_ID_PC_1;
    extend_addr(15 downto 0) := IF_ID_PC_1;
else
    if last_jump = '0' then
        pre_pc := IF_ID_PC_1;
    else pre_pc := ID_NEWPC;
    end if;
    IF_PC <= pre_pc; extend_addr(15 downto 0) := pre_pc;
end if;
```

第一、二、三周期的三个时钟上升沿内从指令存储器中读出指令的内容。

第五个周期将 PC、下一条指令的地址以及该指令信息写入 IF/ID 流水线寄存器中。如果该指令应该正常执行（hold 信号为 0），则将本阶段的信息直接写入流水线寄存器中，否则将 NOP 写入 IF/ID 流水线寄存器中，并将下一条指令的地址置为正确的地址。可见下图程序说明：

```
if hold = '0' then
    IF_ID_PC <= IF_PC;
    IF_ID_PC_1 <= IF_PC_1;
    IF_ID_instruction <= IF_instruction;
else
    IF_ID_instruction <= "0000000000000000";
    IF_ID_PC <= IF_PC;
    if EX_MEM_jump /= last_jump then
        if EX_MEM_jump = '1' then
            IF_ID_PC_1 <= EX_MEM_NEWPC;
        end if;
    else IF_ID_PC_1 <= IF_PC;
    end if;
end if;
```

以上过程中涉及到冒险检测单元和分支预测单元的部分将在下文详细说明。

2、指令译码阶段（ID）：

该阶段为数据准备和控制信号生成阶段。

控制信号生成：

将已经从指令存储器中取出的 16 位指令作为敏感信号传入控制单元，这 16 位二进制数包含了一条指令的所有信息，因此可以生成不同指令所需的所有控制信号。该级流水得到控制信号以后，在自己的第五个周期将控制信号写入 ID/EX 流水线寄存器中保存，供下一级流水使用。

数据准备：

通过对 30 条指令的 16 位二进制数的分析，我们得到 16 位二进制码每几位可能代表的含义，如下图所示：

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">操作码</td><td colspan="3">rx</td><td colspan="3">ry</td><td colspan="3">rz</td><td></td><td></td></tr>
<tr><td colspan="8">操作码</td><td colspan="8">立即数</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5">立即数</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3">立即数</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5">操作码</td></tr>
</table>

15~11 位一定是操作码；

10~8　位可能是 rx 寄存器，可能是操作码；

7~5　位可能是 ry 寄存器；

4~2　位可能是 rz 寄存器；

7~0　位可能是立即数；

4~0　位可能是立即数，可能是操作码；

4~2　位可能是立即数。

在此阶段控制单元可能还没有产生控制信号通知本阶段 16 位指令的每一位代表什么含义，因此可以将所有数据事先准备好，供下一级流水进行选择使用。无论本条指令是否需要 rx，ry 或者 rz 寄存器，本阶段都会根据指令的 10~2 位得到三个寄存器中的值备用。同理，无论本条指令是否需要立即数，都会根据 16 位指令得到立即数的值备用。需要注意的是，不同的需要用到立即数的指令是对 16 位二进制指令的不同部分进行扩展得到的，因此需要根据指令内容选择 7~0 位、4~0 位或者 4~2 位的二进制位进行扩展得到立即数。

在 ID 流水级的第三个时钟周期可以得到所有的准备数据，并在第五个时钟周期将这些数据写入 ID/EX 流水线寄存器保持并供下一级流水使用。

3、执行阶段（EX）：

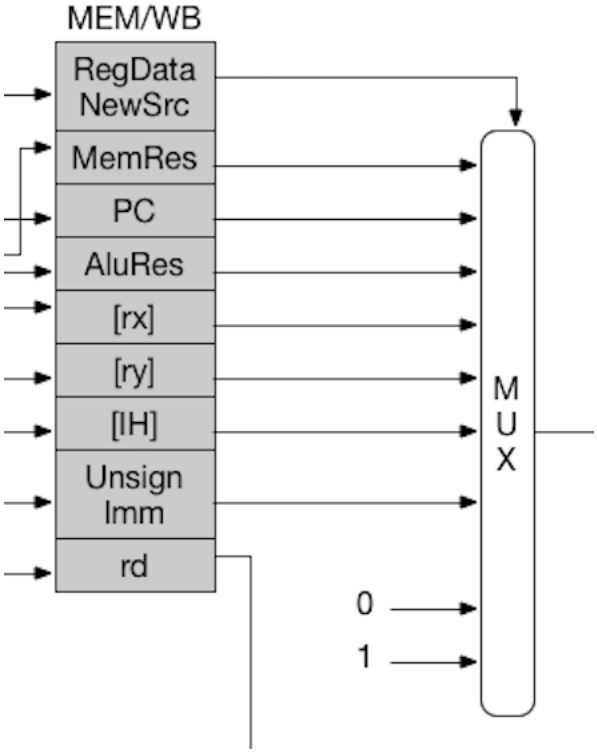
本阶段是指令的执行阶段。在本阶段的第一个时钟周期，该级流水会在转发单元的控制下，得到所需寄存器的值。

在接下来的三个时钟周期里，本级流水会根据如下信号计算出本级和下一级需要用到的数据和信号。

```
signal EX_regdatasrc  : std_logic_vector(3 downto 0);
signal EX_regdest     : std_logic_vector(2 downto 0);
signal EX_branch      : std_logic_vector(1 downto 0);
signal EX_jr          : std_logic;
signal EX_cmp         : std_logic_vector(1 downto 0);
signal EX_aluop       : std_logic_vector(2 downto 0);
signal EX_alusrc1     : std_logic_vector(2 downto 0);
signal EX_alusrc2     : std_logic_vector(1 downto 0);
signal EX_regwrite    : std_logic:= '0';
```

1. 根据三位信号 EX_redest 得到写回寄存器的变编号，即从 R0~R7，SP，IH 和 T 中选择写回寄存器。

2. 根据三位信号 EX_alusrc1 得到 ALU 的第一个输入值，即从 rx，ry，SP，PC 中选择一个作为 ALU 的第一个输入。根据二位信号 EX_alusrc2 得到 ALU 的第二个输入值，即从 ry，符号扩展立即数，零扩展立即数中选择一个作为 ALU 的第二个输入。根据三位信号 EX_aluop 作为 ALU 的控制信号，即控制 ALU 进行哪一种操作。并将 ALU 的输出作为 ALU_result 保存下来。然后可以得到该 result 是否为零的信号 EX_zero 和是否为负数的信号 EX_sign。
3. 计算新的指令地址，所有指令跳转的 B 类指令都是将 PC+1+符号扩展立即数作为下一条指令的新地址。只有 JR 指令是将 rx 的值作为下一条指令的新地址。因此根据一位信号 EX_jr 作为控制信号产生跳转的新地址。
4. 根据二位信号 EX_branch 和 rx 或者 T 的值生成控制信号 EX_jump 控制第一级流水下一条指令是否应该发生跳转。B 指令和 JR 指令需要无条件跳转，对应 EX_branch=“11”。BNEZ 指令只有在 rx 不为零时进行跳转，对应 EX_branch=“01”。BEQZ 指令和 BTEQZ 指令分别在 rx 为零和 T 为零时进行跳转，对应 EX_branch=“10”。
5. 根据四位指令 EX_cmp 和一位信号 EX_zero 以及 EX_sign 进一步完善四位控制信号 EX_regdatasrc。如下图所示 EX_regdatasrc 的作用为从 MEM/WB 流水线寄存器中选择数值写回到 rd 代表的寄存器中。在第三季流水之前，若 rd 代表 T 寄存器，EX_regdatasrc 只能控制将 0 或者 1 写回 T 寄存器，并不能明确区分。需要判断计算结果为零或为负来确定 T 的写回值。CMP 指令和 CMPI 指令要求结果为零时选择 0 写回，否则选择 1 写回，对应 EX_cmp=“11”。SLTU 指令要求结果为正时选择 0 写回，否则选择 1 写回，对应 EX_cmp=“01”。通过以上操作可以讲原来代表写回 0or1 的 EX_regdatasrc 控制信号区分开来。



4、内存读写阶段（MEM）：

流水线第四级为 MEM 级，主要负责内存的读写和串口的访问。由于涉及到内存和串口，该级是 CPU 效率的瓶颈，这一步需要 5 个时钟周期，其它各级因此也需要 5 个时钟周期来完成。

该级首先进行的是取指令操作，从 RAM2 中取出指令（具体方法见后面 RAM 读取部分），在这一步取指是由于后面的操作中还有可能会对 RAM2 进行操作，因此放在一起，从准备取指到取指完成需要 3 个时钟周期。由于取指在第四级完成，所以 CPU 刚运行时前三条指令为空指令被浪费，但由于需要执行的指令较多，因此这点损失可以忽略不计。

在完成取指的同时，MEM 级开始准备进行数据读写或串口访问，只有当控制信号 memread 或 memwrite 为 1 时这一部分才会被执行。具体执行什么操作取决于地址值，按照实验要求，地址 0xBF00 作为串口地址，若向该地址进行读写则不向内存中读写而向串口进行读写；地址 0xBF01 作为串口情况地址，若向该地址读则返回串口情况，串口情况用两位进行表示，高位表示允许读操作(data_ready)，低位表示允许写操作(tbre and tsre)；其他地址进行正常的读

写操作，由地址决定读写对象是 RAM1 还是 RAM2。读操作需要 3 个周期，写操作需要 2 个周期，由于这一步开始和之前的取指结束在同一时钟周期里，因此该段需要至少 5 个时钟周期。

5、写回阶段（WB）

流水线的最后一级主要负责将前面的结果写回寄存器堆，MEM/WB 段间寄存器保存了之前运算和读取内存、串口等的结果，执行该步时首先根据控制信号 WB_regdatasrc 决定写往寄存器的数据内容，共 9 中可能值，接下来若 WB_regwrite 为 1，则说明需要写回寄存器，根据控制信号 WB_rd 判断目的寄存器并将刚才得到的数据写回寄存器；否则说明该指令不会修改寄存器的值，不对寄存器堆进行任何操作。

在写回部分的设计中有一点是需要注意的，这一级需要访问寄存器堆，而流水线第二级同样涉及到寄存器堆的访问，因此这时在同一周期中寄存器堆会被访问两次，这时二者之间就有了访问时序问题。由于存在读后写的冒险情况，必须保证先将数据写回寄存器堆，再从寄存器堆中读取数据，以保证从寄存器中读取到正确的内容。在此次实验中，对寄存器堆的修改发生在一个 CPU 周期的第 2 个时钟周期，而读取寄存器发生在 CPU 周期的第 3 个时钟周期，符合避免冒险的设计。

三、部件设计

1、运算器 ALU

ALU 是 CPU 的重要组成部分，大多数指令的执行实际上就是依靠的 ALU。由于所有指令都是一元或二元运算，因此 ALU 设计为 2 输入，同时运算器要负责所有可能的运算，因此还要增加一个输入来控制运算，即操作码。在本次实验中，共需要使用 8 种运算：

运算	操作码	使用该操作的指令
加法	000	ADDIU、BEQZ、LW、SW 等
减法	001	SUBU、CMPI、SLTU 等
与	111	AND

或	010	OR
非	101	NOT
左移	011	SLL
逻辑右移	110	SRA
算数右移	100	SRL

与之前小实验中设计的 ALU 不同的是，本次试验中使用的 ALU 并不是时序电路而是逻辑电路，只要操作数或操作码中的任意输入发生变化，ALU 的输出值就会发生变化。这是由于实验中并没有这样的要求，将 ALU 设计成时序的反而增加了设计的难度（比如时钟周期的同步等）。

需要注意的是，虽然部分运算（加减法）有符号运算和无符号运算之分，但在 ALU 中它们都是一样的。因为它们运算的算法是一样的，只是最后对结果的采取了不同的理解。

2、控制信号

控制信号主要由控制单元生成，决定了流水线各级的操作内容。在本次试验中使用了 12 种控制信号，现分别说明如下：

1、regdatasrc

说明写回寄存器的值的内容，在 WB 级中使用，设为 4 位信号，共有 10 种取值。

信号取值	信号含义	指令
0000	取 ALU 的运算结果	ADDIU、OR 等
0001	取读内存结果	LW、LW_SP
0010	取 PC 寄存器值	MFPC
0011	取 IH 寄存器值	MFIH
0100	取 rx 寄存器值	MTIH、MTSP
0101	取 ry 寄存器值	MOVE
0110	取立即数值	LI
0111	取 0 或 1	CMP、CMPI
1111	无操作	B、NOP 等

上表是由控制器产生的信号含义，这时信号 0111 表示取 0 或 1 是由于此时尚不知道到底是取 0 还是取 1，在经过 ALU 运算之后会根据运算结果对信号值进行修改：0111 表示取 0,1000 表示取 1.

2、regdest

指明目的寄存器，在 WB 级使用，设为 3 位信号，有 7 中取值

信号取值	信号含义	指令
000	向 rx 写回	ADDIU、OR 等
001	向 ry 写回	ADDIU3、LI
010	向 rz 写回	ADDU、SUBU
011	向 SP 写回	MTSP
100	向 T 写回	CMP、CPMI 等
101	向 IH 写回	MTIH
111	无操作	B、SW 等

3、memread

1 位信号，说明是否需要读内存，在 MEM 级使用

信号取值	信号含义	指令
1	读取内存	LI、LI_SP
0	不读取内存	其他

4、memwrite

1 位信号，说明是否需要写内存，在 MEM 级使用

信号取值	信号含义	指令
0	不写内存	SW、SW_SP
1	写内存	其他

注意所有指令中不存在同时对内存进行读和写的指令，因此 memread 和 memwrite 不可能同时为 1.

5、branch

判断是否为B型指令，如果是B型指令，可能需要修改新PC值。对于不同的B型指令会产生不同的信号来说明跳转条件，在EX阶段使用，设为2位信号，共4中取值

信号取值	信号含义	指令
01	判断不为0跳转	BNEQZ
10	判断为0跳转	BEQZ、BTEQZ
11	无条件跳转	B、JR
00	无操作，非B型指令	其他

6、cmp

与branch信号类似，判断是否是比较型指令，若是则给出设置T的条件，在EX级使用，设为2位信号，共3中取值

信号取值	信号含义	指令
01	判断是否非负	SLTU
11	判断是否为0	CMP、CMPI
00	无操作，非比较型指令	其他

7、aluop

说明ALU的操作符，在EX级使用，控制本指令的运算类型，设为3位信号，共8种取值

信号取值	信号含义	指令
000	加法	ADDIU、BEQZ、LW、SW等
001	减法	SUBU、CMPI、SLTU等
111	与	AND
010	或	OR
101	非	NOT
011	左移	SLL

110	逻辑右移	SRA
100	算数右移	SRL

部分指令不需要 ALU 参与，如 MOVE 指令，但仍将该步的操作码设为 000，只要后面不用到它就不会产生任何影响，反而减少了一位信号。

8、alusrc1

说明 ALU 的第一个操作数，在 EX 级使用，设为 3 位信号，共 5 种取值

信号取值	信号含义	指令
000	取 rx 值	ADDIU、CMPI 等
001	取 ry 值	SLL、NOT 等
010	取 SP 值	LW_SP、SW_SP
011	取 PC 值	BEQZ、BTEQZ 等
111	无操作，取 0	MFPC、NOP 等

9、alusrc2

说明 ALU 的第二个操作数，在 EX 级使用，设为 2 位信号，共 4 种取值

信号取值	信号含义	指令
00	取符号扩展的立即数	ADDIU、BEQZ 等
01	取 ry 值	CMP、SUBU 等
10	取无符号扩展的立即数	SRL、SLL 等
11	无操作，取 0	MFPC、NOP 等

10、memdatasrc

若有写内存操作，该信号指明写内存的地址，设为 1 为信号，在 MEM 级使用，若该指令不写内存，则默认置为 0，此时仍不会写内存，因为 memwrite 信号的值为 0。

信号取值	信号含义	指令
0	取 ry 的值为写地址	SW 等

1	取 rx 的值为写地址	SW_SP

11、regwrite

指明是否需要将数据写回寄存器堆，在 WB 级使用，设为 1 位信号

信号取值	信号含义	指令
0	取 ry 的值为写地址	BEQZ、SW 等
1	取 rx 的值为写地址	ADDIU、CMP 等

12、jr

判断是否为 jr 指令，在 ID、EX 级使用，设为 1 位信号

信号取值	信号含义	指令
1	是 jr 指令	JR
0	不是 jr 指令	其他

3、控制单元

控制单元的主要作用是由指令产生对应的控制信号。仔细观察各指令可知，对大多数指令，指令前 5 位可以决定指令类型，由此即可产生所有的控制信号，有少数信号前五位相同（例如 and、cmp 等），此时需要依靠后面的一些字段进行判断，一旦判断出指令类型，便可以由下表产生对应的指令，各类控制信号的含义见前面控制信号设计部分。

指令	RegDataSrc	RegDest	MemRead	MemWrite	Branch	Cmp	AluOP	AluSrc1	AluSrc2	MemDataSrc	regwrite	jr
ADDIU	0000	000	0	0	00	00	000	00	00	0	1	0
ADDIU3	0000	001	0	0	00	00	000	00	00	0	1	0
ADD	0000	011	0	0	00	0	00	10	00	0	1	0

SP						0	0					
ADDU	0000	010	0	0	00	00	000	00	01	0	1	0
AND	0000	000	0	0	00	00	111	00	01	0	1	0
B	1111	111	0	0	11	00	000	11	00	0	0	0
BEQZ	1111	111	0	0	10	00	000	11	00	0	0	0
BNEZ	1111	111	0	0	01	00	000	11	00	0	0	0
BTEQZ	1111	111	0	0	10	00	000	11	00	0	0	0
CMP	0111	100	0	0	00	11	001	00	01	0	1	0
JR	1111	111	0	0	11	00	000	00	11	0	0	1
LI	0110	000	0	0	00	00	000	00	11	0	1	0
LW	0001	001	1	0	00	00	000	00	00	0	1	0
LW_SP	0001	000	1	0	00	00	000	10	00	0	1	0
MFIH	0011	000	0	0	00	00	000	00	11	0	1	0
MFPC	0010	000	0	0	00	00	000	00	11	0	1	0
MTIH	0100	101	0	0	00	00	000	00	11	0	1	0

MTSP	0100	011	0	0	00	00	000	00	11	0	1	0
NOP	1111	111	0	0	00	00	000	00	11	0	0	0
OR	0000	000	0	0	00	00	010	00	01	0	1	0
SLL	0000	000	0	0	00	00	011	01	10	0	1	0
SRA	0000	000	0	0	00	00	100	01	10	0	1	0
SUBU	0000	010	0	0	00	00	001	00	01	0	1	0
SW	1111	111	0	1	00	00	000	00	00	0	0	0
SW_SP	1111	111	0	1	00	00	000	10	00	1	0	0
CMPI	0111	100	0	0	00	11	001	00	00	0	1	0
MOVE	0101	000	0	0	00	00	000	00	11	0	1	0
NOT	0000	000	0	0	00	00	101	01	11	0	1	0
SRL	0000	000	0	0	00	00	110	01	10	0	1	0
SLTU	0111	100	0	0	00	01	001	00	01	0	1	0

与 ALU 一样，控制单元也是逻辑电路组成的，输出随着输入的指令变化而变化，与时钟无关。

4、冒险检测单元

这一单元用来解决结构冲突。结构冲突是由前一条指令读取数据存储器而后一条指令访问上一条指令写回寄存器的值而引起的。比如说：

LW R1 0x0004

ADDIU R1 R1 0x0002

第二条指令需要在EX阶段第一个时钟周期读取R1的值，但是第一条指令会在MEM阶段第五个时钟周期得到应该写回R1中的数据存储器中的数值。因此第二条指令无论如何也不能通过转发的方式得到正确R1的结果，所以需要在第一条指令和第二条指令之间插入一条NOP指令，即空指令来使第二条指令等待一级流水的工作时间再重新载入以获得正确的R1值。冒险检测单元根据IF阶段读入的后一条指令和ID阶段生成的控制信号memread（是否读取数据存储器）来检测是否发生了结构冲突，如果发生了结构冲突，冒险检测单元可以在IF阶段的第四个时钟周期生成一个控制信号hold，并将其置为1，代表发生了数据冲突。IF阶段在第五个时钟周期监控hold的值，若hold为零，则指令正常向下一级流水传递。若hold为1，则将IF/ID流水线寄存器中的指令置零，并将PC+1（即下一条指令置为当前PC），使得下一条指令依旧是这条没有运行的指令。

5、分支预测单元：

这一单元用来解决控制冲突以及对指令地址进行预判。当前一条指令是B类指令或者JR时，程序可能会发生跳转。由于延时槽的存在，跳转指令的下一条指令一定会执行，这样冲突会出现在第三条指令上。只有当第一条指令（跳转指令）进行到EX阶段第五个时钟周期时才能确定是否跳转，但此时第三条指令已经将指令从指令存储器中取出来了，这导致第三条指令是错误的指令。朴素的处理方式为遇到跳转指令则强行在第二、第三条指令之间插入一条NOP指令，即强制第三条指令等待一级流水周期以消灭冲突。但这样最少等待一个周期，最多等待两个周期的做法效率并不客观，因此我们引入分支预测的处理方式。

因为大多数同样的跳转会进行多次，因此我们根据上次跳转的结果来预判这一次是否进行跳转。具体实现过程见如下代码段：

图一：每当一条新指令读入，都检查之前的第二条指令是不是跳转指令。

如果不是的话，直接读取下一条指令。如果是的话根据上一次跳转结果进行预判，如果上一次跳转成功，那么这一次也预判跳转成功，否则直接读取下一条指令。

图二：当跳转指令在 EX 阶段结束时得到了是否应该跳转的信号，与上次跳转结果进行对比。如果两者一致，则说明这次跳转预测正确，流水正常进行。否则预测失败，清空该条指令并重新读取该指令。置信号量 hold = ‘1’；

图三：在第三条指令 IF 阶段的第五个时钟周期，即向流水线寄存器写入数据时检查 hold 是否为一（跳转预判错误）。如果预判错误，则将指令清空，插入一条 NOP 语句。并同时将下一条指令的地址置为正确的地址，供下一条指令使用。

```
if ID_EX_branch = "00" then
    IF_PC <= IF_ID_PC_1;
    extend_addr(15 downto 0) := IF_ID_PC_1;
else
    if last_jump = '0' then
        pre_pc := IF_ID_PC_1;
    else pre_pc := ID_NEWPC;
    end if;
    IF_PC <= pre_pc; extend_addr(15 downto 0) := pre_pc;
end if;
```

```
if EX_MEM_branch /= "00" then
    if EX_MEM_jump /= last_jump then
        hold <= '1';
    end if;
else
```

```
if hold = '0' then
    IF_ID_PC <= IF_PC;
    IF_ID_PC_1 <= IF_PC_1;
    IF_ID_instruction <= IF_instruction;
else
    IF_ID_instruction <= "0000000000000000";
    IF_ID_PC <= IF_PC;
    if EX_MEM_jump /= last_jump then
        if EX_MEM_jump = '1' then
            IF_ID_PC_1 <= EX_MEM_NEWPC;
        end if;
    else IF_ID_PC_1 <= IF_PC;
    end if;
end if;
last_jump := EX_MEM_jump;
```

6、转发单元

在 MIPS 经典的五级流水线设计中,要考虑解决各种冲突的问题。最常见的一类冲突就是数据冲突,比如说:

LI R1 0x0004

ADDIU R1 R1 0x0002

第二条指令需要读取 R1 的值,但是第一条指令的 R1 还没有写回。但此时可以发现,R1 的值可以在 ALU 结束之后立即转发给 ALU。类似的,判断条 件可以如此写出:

1 EX_MEM_RegDst == ID_EX_AluSrc1

2 EX_MEM_RegDst == ID_EX_AluSrc2

3 MEM_WB_RegDst == ID_EX_AluSrc1

4 MEM_WB_RegDst == ID_EX_AluSrc2

只要上述 4 个条件任意一个成立,就将 Alu 的两个操作数更换。这里使用了数据旁路技术(也称转发,forwarding)。即 ALU 在读取操作数 AluSrc1 和 AluSrc2 的同时,也会读取前两条指令的写回寄存器编号和当前指令的源寄存器编号,如果上述判断成立,则将相关的值送给 ALU 的输入,用以代替原来的输入。

但我们的针对 16 位指令系统设计的 CPU 有所不同,某些指令可能直接将一个寄存器的值直接写入另一个寄存器, 例如:

MOVE：R[x] <-- R[y]

MTSP： SP <-- R[x]

因此转发单元只给 ALU 的输入提供信号显然是无法完成正常功能的。自然的想法为转发单元作为整个 ID/EX 流水线寄存器的控制信号提供者，不过这样处理的话转发单元的输入信号和输出信号就变的非常冗余。因此我们考虑用自己设计的纠错单元和临时寄存器堆来代替转发单元。

纠错单元和简单来说就是记录 R0~R7、SP、IH、T 这些寄存器中哪一个或者两个目前已经被更改，但是由于流水线的设计问题，目前还没有被写回寄存器堆中，导致 ID/EX 流水线寄存器中的值并不正确。而临时寄存器堆则记录了不正确的寄存器的正确取值。因此 EX 阶段在第一周期从 ID/EX 流水线寄存器取值时需要有信号通知它要取的值是正确的还是错误的。如果不正确则从临时寄存器堆中取得正确的值。这里的设计借鉴了 cache 的思想，计算机不同的核从内存中取值时也涉及到类似的数据冲突，并应用广播的方式协调解决。对于我们的 CPU 具体实现见下图代码：

```
----------------------临时寄存器堆----------------------------------
signal MEM_WB_tmp         : std_logic_vector(15 downto 0);
signal EX_MEM_tmp         : std_logic_vector(15 downto 0);
signal MEM_WB_tmp2        : std_logic;
signal EX_MEM_tmp2        : std_logic;
```

6、串口与内存访问

串口访问：

串口是 CPU 与 PC 进行数据交互的接口。实验中串口地址设为 0xBF00，也就是说向该地址进行读写操作时操作对象不是内存而是串行接口。串行接口访问的操作方式如下

1、读串口：接收串口上发送来的数据，该操作分配了 3 个时钟周期：第一个周期进行初始化操作，将 rdn 置为 1，将数据线设为高阻，做好接收数据的准备；第二个周期将 rdn 置为 0，开始进行数据读取；第三个周期读数完成，保存从串口上读到的数据并写入 MEM/WB 段间寄存器。

2、写串口：将数据发送至串口上，该操作分配了 3 个时钟周期：第一个周

期进行初始化操作，将 wrn 值 1，同时由 EX/MEM 段间寄存器准备数据线，第二个周期将 wrn 置为 0，锁存将要发送的数据；第三个周期再将 wrn 置为 1，将数据发送出去。

除此之外还有报告串口情况的操作，该操作为读地址 0xBF01，将返回两位结果，分别表示允许读和允许写，读状态位是 data_ready 的值，该值表明有一个待读的数在串口上，允许进行读操作数；写状态是(tbre and tsre)，该值表示串口空闲，可以进行写操作。若进行串口情况报告，则该步不进行串口的访问。

ram 读写：

内存用于存放程序指令和数据，本次实验中使用两片 SRAM 作为内存，将 RAM1 用作数据内存，将 RAM2 用作指令内存。RAM 访问的操作方式如下：

1、读 RAM：从 RAM 中的指定地址读取数据，需要 2 个时钟周期：第一个周期进行初始化，拉低 OE 使能并保持 WE 使能高电平，由 EX/MEM 段间寄存器准备好地址线，同时将数据线设为高阻，开始进行读数；第二个周期完成读数，将数据线上的读数结果保存下来并将 WE 使能拉高。

2、写 RAM：向 RAM 中的指定地址写数据，需要 3 个周期：第一个周期进行初始化，拉高 OE 使能和 WE 使能，由 EX/MEM 段间寄存器准备好数据和地址线；第二个周期拉低 WE 使能并保持 OE 使能高电平，开始写数据；第三个周期完成写数据并将 WE 使能拉高

无论是哪种 RAM 访问操作，都需要保证总使能 EN 为低电平以保证读写的正常进行，同时还有一点需要注意，RAM1 和串口的数据线是共用的，也就是说二者可能互相影响，新词在操作时需要格外注意：对 RAM1 进行操作时，要将 wrn 和 rdn 都置为 1,；对串口进行操作时，要将 RAM1 总使能拉高。

四、实验代码

见上传文件中的 CPU 文件夹

五、实验结果

本组最终完成了 5 级流水 CPU，主频约为 2MHz，下图为测试程序 1 和 5 的测试结果

```
  >> A
[4000]  LI R5 FF
[4001]  NOP
[4002]  NOP
[4003]  NOP
[4004]  SLL R5 R5 0
[4005]  NOP
[4006]  NOP
[4007]  NOP
[4008]  ADDIU R5 82
[4009]  LI R4 60
[400a]  NOP
[400b]  NOP
[400c]  NOP
[400d]  ADDIU R4 1
[400e]  LI R0 0
[400f]  LI R1 1
[4010]  LI R2 2
[4011]  BNEZ R4 FB
[4012]  NOP
[4013]  ADDIU R5 1
[4014]  NOP
[4015]  NOP
[4016]  NOP
[4017]  BNEZ R5 F1
[4018]  NOP
[4019]  JR R7
[401a]  NOP
[401b]

  >> G
running time : 70.028 s
```

```
  >> A
[4000]  LI R2 FF
[4001]  LI R3 55
[4002]  SLL R3 R3 0
[4003]  LI R5 FF
[4004]  SLL R5 R5 0
[4005]  ADDIU R5 83
[4006]  LI R4 61
[4007]  SW R3 R4 1
[4008]  BNEZ R4 FE
[4009]  ADDIU R4 1
[400a]  BNEZ R5 FB
[400b]  ADDIU R5 1
[400c]  JR R7
[400d]  NOP
[400e]

  >> G
running time : 40.014 s
```

六、实验感想与收获

1、这个实验让我们亲自设计了 CPU，在实验过程中必须弄清楚 CPU 的组成原理，了解每一部分的作用及其设计方式。通过完成本次实验，我们对 CPU 的结构有了十分深刻的认识，相信在很长一段时间内，这部分内容都会令我们印象深刻。

2、本次实验属于硬件编程，使用的是之前没怎么认真使用过的 VHDL 语言，使用起来相当不熟练，而且硬件编程与软件编程有不少区别，例如变量赋值等问题。在实验中我们也遇到了许多这方面问题，通过查询资料和与同学讨论才得到解决办法，这也让我们深刻的了解到硬件编程的困难。

大实验 实验报告

计 31（104 组）

陈冲 赖涵光 蔡文静

目录

一、实验目的

（1）加深对计算机系统知识的理解；

（2）进一步理解和掌握流水线结构计算机各部件组成及内部工作原理；

（3）掌握计算机外部输入输出的设计；

（4）培养硬件设计和调试的能力。

二、完成情况概述

本次大实验通过编写 VHDL 代码实现了支持指令流水的计算机系统。使用 10M 时钟，实现延迟槽。最终成果：可以正常运行监控程序；正常运行五条扩展指令，运算结果正确；正常运行测试样例，实现了数据冲突、结构冲突、控制冲突的解决，同时，运行速度较快；完成了 INT 中断与单步分支预测两项扩展功能；代码精简，算上注释，仅用不到 900 行代码就实现了上述所有功能。

同时，为了方便管理三人编写的代码，也为了修改后重新使用此前的代码，我们还使用了 git 工具进行版本控制工作，这对我们的调试工作有很大帮助。

总的来说，本组设计实现的 CPU 以较高的质量完成了实验的所有要求，性能稳定，并实现了一些扩展。

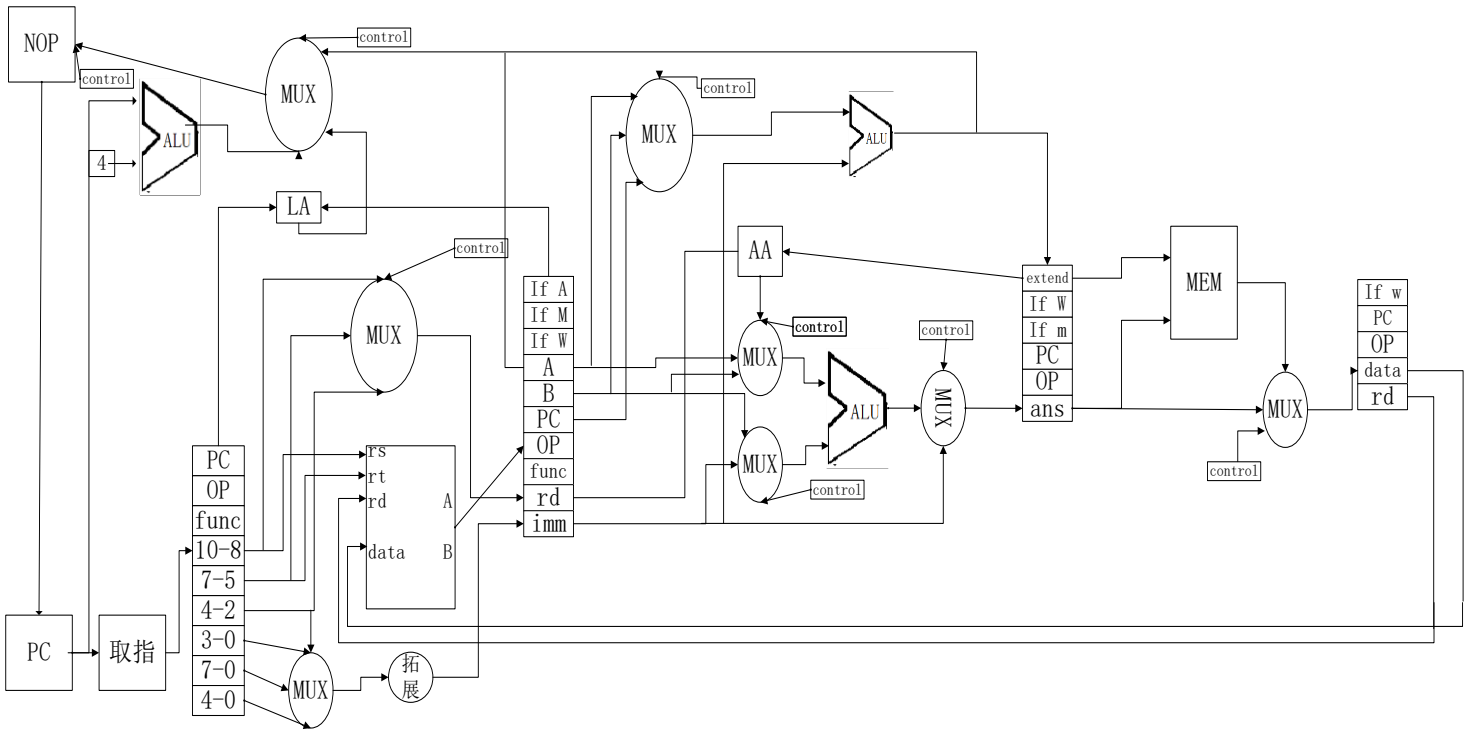
三、指令系统与数据通路

指令系统：

本次 CPU 大实验实现了给定的 25 条基本指令与 5 条扩展指令（SLLV、SRAV、CMPI、MOVE、SLTU），指令列表如下：

0	00001				NOP
1	00010				B
2	00100				BEQZ
3	00101				BNEZ
4	00110			00	SLL
5	00110			11	SRA
6	01000				ADDIU
7	01001				ADDIU
8	01100	000			BTEQZ
9	01100	011			ADDSP
10	01100	100			MTSP
11	01101				LI
12	01110				CMPI
13	01111				MOVE
14	10010				LW_SP
15	10011				LW
16	11010				SW_SP
17	11011				SW
18	11100			01	ADDU
19	11100			11	SUBU
20	11101		000	00	JR
21	11101		010	00	MFPC
22	11101		001	00	SLLV
23	11101		011	00	AND
24	11101		011	01	OR
25	11101		010	10	CMP
26	11101		001	11	SRAV
27	11110			00	MFIH
28	11110			01	MTIH
29	11101		000	11	SLTU

数据通路：



本组设计的数据通路如上图所示。在数据通路中，我们不仅明确了五级流水各个阶段的过程，同时还写出了各个阶段需要的所有阶段寄存器，这对我们实验初期的工作有很大帮助。同时，在设计数据通路时，我们已经尝试进行冲突的解决，例如，图中就标出了 AA 型数据冲突和 LA 型数据冲突的解决方式，具体的冲突解决方法下面将详细讲述。

四、冲突解决方法

1.数据冲突

AA 型数据冲突：例如连续两个 ADDU 指令，我们使用数据旁路来解决。在 EXE 的第 0 阶段，判断是否需要从数据旁路中获得操作寄存器的值，如下图。

```
if id_exe_a_exe = '1' then
        exe_a := id_exe_a_from_exe;
elsif id_exe_a_me = '1' then
        exe_a := id_exe_a_from_me;
else
        exe_a := id_exe_a;
end if;
if id_exe_b_exe = '1' then
        exe_b := id_exe_b_from_exe;
elsif id_exe_b_me = '1' then
        exe_b := id_exe_b_from_me;
else
        exe_b := id_exe_b;
end if;
```

在 EXE 的第 4 阶段，判断上一条指令的结果寄存器和该条指令的操作寄存器相同，且没有访存，则使用数据旁路，如下图。

```
if exe_wb = '1' and id_op /= 0 then
        if exe_rz = id_rx then
                id_exe_a_exe <= '1';
                id_exe_a_from_exe <= exe_ans;
        end if;
        if exe_rz = id_ry then
                id_exe_b_exe <= '1';
                id_exe_b_from_exe <= exe_ans;
        end if;
end if;
```

LA 型数据冲突：例如 LW 指令后紧跟一条 ADDU 指令，我们使用插入 NOP 的方法来解决，如下图。

```
if exe_lw = '1' and id_op /= 0 then
        if exe_rz = id_rx or exe_rz = id_ry then
                pre_pc_by_exe <= '1';
                pre_pc_from_exe <= exe_pc;
                if_id_op_nop_exe <= '1';
                id_exe_op_nop_exe <= '1';
        end if;
end if;
```

2.结构冲突

通过分层把取指和 MEM 访存分割在不同阶段，以此来避免结构冲突。

3.控制冲突

B 型控制冲突：使用分支预测的方法，在扩展中会详细描述。

J 型控制冲突：使用插入 NOP 的方法来实现，如下图。

```
when 20 =>     --          JR
        id_exe_wb <= '0';
        pre_pc_by_id <= '1';
        get_regbin( "0" & id_ins(10 downto 8), pre_pc_from_id);
        if_id_op_nop_id <= '1';
        if id_ins(10 downto 8)= "110" and int_jr ='1'    then
                int_jr <= '0';
                int_recover <= '1';
        end if;
```

五、性能测试运行情况

1.性能标定

```
[4008]  ADDIU R5 82
[4009]  LI R4 60
[400a]  NOP
[400b]  NOP
[400c]  NOP
[400d]  ADDIU R4 1
[400e]  LI R0 0
[400f]  LI R1 1
[4010]  LI R2 2
[4011]  BNEZ R4 FB
[4012]  NOP
[4013]  ADDIU R5 1
[4014]  NOP
[4015]  NOP
[4016]  NOP
[4017]  BNEZ R5 F1
[4018]  NOP
[4019]  JR R7
[401a]  NOP
[401b]

  >> G
running time : 15.027 s
```

运行时间 15.027 秒

2.运算数据冲突的效率测试

```
[4005]  ADDU R1 R1 R2
[4006]  ADDU R2 R1 R3
[4007]  SUBU R3 R2 R2
[4008]  CMP R1 R2
[4009]  ADDU R2 R3 R2
[400a]  BEQZ R4 3
[400b]  ADDIU R4 1
[400c]  BTEQZ F8
[400d]  NOP
[400e]  ADDIU R5 1
[400f]  BNEZ R5 F5
[4010]  NOP
[4011]  JR R7
[4012]  NOP
[4013]

  >> G
running time : 27.565 s
```

运行时间 27.565 秒

3.控制指令冲突测试

```
  >> A
[4000]  LI R1 1
[4001]  LI R5 FF
[4002]  SLL R5 R5 0
[4003]  ADDIU R5 83
[4004]  LI R4 60
[4005]  CMP R4 R1
[4006]  BTEQZ 3
[4007]  ADDIU R4 1
[4008]  BNEZ R4 FD
[4009]  CMP R4 R1
[400a]  BNEZ R5 F9
[400b]  ADDIU R5 1
[400c]  JR R7
[400d]  NOP
[400e]

  >> G
running time : 15.027 s
```

运行时间 15.027 秒

4.访存数据冲突性能测试

```
  >> A
[4000]  LI R2 FF
[4001]  LI R3 C0
[4002]  SLL R3 R3 0
[4003]  LI R5 FF
[4004]  SLL R5 R5 0
[4005]  ADDIU R5 83
[4006]  LI R1 61
[4007]  SW R3 R1 2
[4008]  LW R3 R4 2
[4009]  SW R3 R4 1
[400a]  LW R3 R1 1
[400b]  BNEZ R4 FB
[400c]  ADDIU R1 1
[400d]  BNEZ R5 F8
[400e]  ADDIU R5 1
[400f]  JR R7
[4010]  NOP
[4011]

  >> G
running time : 20.025 s
```

运行时间 20.025 秒

5.读写指令存储器测试

```
  >> A
[4000]  LI R2 FF
[4001]  LI R3 55
[4002]  SLL R3 R3 0
[4003]  LI R5 FF
[4004]  SLL R5 R5 0
[4005]  ADDIU R5 83
[4006]  LI R4 61
[4007]  SW R3 R4 1
[4008]  BNEZ R4 FE
[4009]  ADDIU R4 1
[400a]  BNEZ R5 FB
[400b]  ADDIU R5 1
[400c]  JR R7
[400d]  NOP
[400e]

  >> G
running time : 7.528 s
```

运行时间 7.528 秒

六、扩展指令运行情况

针对五条扩展指令，我们自己编写了测试程序对其进行测试，运行后通过 R 命令查看寄存器的值，结果全部正确。

1. SLLV

```
  >> A
[4000]  LI R1 2
[4001]  SLL R2 R1 0
[4002]  SLL R3 R1 1
[4003]  LI R4 1
[4004]  SLLV R1 R4
[4005]  JR R7
[4006]

  >> G
running time : 0.028 s

  >> R
R0=0000  R1=0002  R2=0200
R3=0004  R4=0004  R5=8007
```

2. SRAV

```
  >> A
[4000]  LI R1 2
[4001]  SLL R1 R1 0
[4002]  SRA R2 R1 0
[4003]  SRA R3 R2 1
[4004]  LI R4 1
[4005]  SRAV R3 R4
[4006]  JR R7
[4007]

  >> G
running time : 0.028 s

  >> R
R0=0000  R1=0200  R2=0002
R3=0001  R4=0000  R5=8007
```

3. CMPI

```
[4000]  LI R1 1
[4001]  CMPI R1 1
[4002]  BTEQZ 3
[4003]  LI R3 3
[4004]  LI R4 1
[4005]  JR R7
[4006]

  >> G
running time : 0.028 s

  >> R
R0=0000  R1=0001  R2=0002
R3=0003  R4=0000  R5=8007
```

CMPI 涉及对 T 寄存器的赋值，因此紧跟一条 BTEQZ——根据 T 寄存器的跳转指令，来检验是否正确对 T 寄存器进行赋值。由于使用了延迟槽，因此 BTEQZ 的下一条指令“LI　R3　3”依然会执行，之后正常跳转，即“LI　R4　1”不会执行。

4. MOVE

5. SLTU

SLTU 涉及对 T 寄存器的赋值，因此紧跟一条 BTEQZ 来验证正确性，方法同 CMPI。

七、扩展

1.单步分支预测

实现方法：

记录上次 B 型指令（不包含无条件跳转）是否跳转，以此来预测下一 B 型指令的目标地址。

首先在 ID 阶段译码出是一条 B 型指令，则根据 predict 的值预测是继续执行还是跳转。

```
if predict = '1' then
        pre_pc_by_id <= '1';
        pre_pc_from_id <= id_pc +sign_extend8( id_ins(7 downto 0) );
```

在 EXE 阶段，已经可以得到计算的结果，此时就可以判断是否预测错误。以 BEQZ 为例，如下图，exe_a="0000000000000000" and predict='0'表示应该跳转但没有跳转；exe_a/="0000000000000000" and predict='1'表示不该跳转却跳转了，需要在后面进行补救。exe_br 为‘1’表示预测错误，同时计算出正确的地址 exe_pc2。

```
when 2 | 8 =>    --    BEQZ
        if (exe_a = "0000000000000000" and predict = '0' ) or (exe_a /= "0000000000000000" and predict = '1') then
                exe_br := '1';
                exe_pc2 := exe_pc + exe_imm;
        end if;
```

如果预测成功则什么都不做，如果预测错误则进行补救：若是应该跳转但没有跳转，则执行这一跳转；若是不该跳转却跳转了，则让 pc 加一得到正确的 pc。纠正的方法是插入 NOP。同时将 predict 取反保证正确性。

```
if exe_br = '1' then
        pre_pc_by_exe <= '1';
        if predict = '0' then
                pre_pc_from_exe <= exe_pc2;
        else
                pre_pc_from_exe <= exe_pc + 1 ;
        end if;
        predict <= not predict ;
        if_id_op_nop_exe <= '1';
        --id_exe_op_nop_exe <= '1';
end if;
```

2.INT 指令中断

实现方法：

通过阅读监控程序，知道了监控程序中处理中断的方法。首先进行保存现场的工作，如下图，

```
signal  int_r0          : std_logic_vector( 15 downto 0) := "0000000000000000";
signal  int_r1          : std_logic_vector( 15 downto 0) := "0000000000000000";
signal  int_imm         : std_logic_vector( 15 downto 0) := "0000000000000000";
signal  int_pc          : std_logic_vector( 15 downto 0) := "0000000000000000";
signal  int_jr          : std_logic := '0';
signal  int_recover     : std_logic := '0';
```

int_r0、int_r1、int_imm、int_pc 备份当前的 r0，r1，中断号，当前 PC，int_jr 表示是否跳转，int_recover 表示是否恢复。其次，如下图，

```
when "11111" =>
        if_id_op <= 30;
        pre_pc <= "0000000000000101";
```

在 IF 阶段把 INT 指令编号为 30。再次，如下图，

```
when 30 =>
        id_exe_wb <= '0';
        int_jr <='1';
        int_imm <= "000000000000" & id_ins(3 downto 0);
        int_pc<= id_pc;
```

在 ID 阶段，当检测到 INT 指令时，实行跳转，把 int_jr 设为 1，并备份 PC。然后，在 WB 阶段，如下图，

```
if me_wb_op = 30 then
        int_r0 <= r0;
        int_r1 <= r1;
        sp <= "1011111100010000";
        r0 <= int_imm;
        r1 <= int_pc;
end if;
```

当检测到 INT 指令时，对 R0、R1 进行备份，令 SP=BF10，将中断号和当前 PC 存入 R0，R1。

根据监控程序，当指令运行到“JR R6”时返回原 PC，因此，在 ID 阶段处理 JR 指令过程中，当检测到“JR R6”且 int_jr=1 时，如下图，

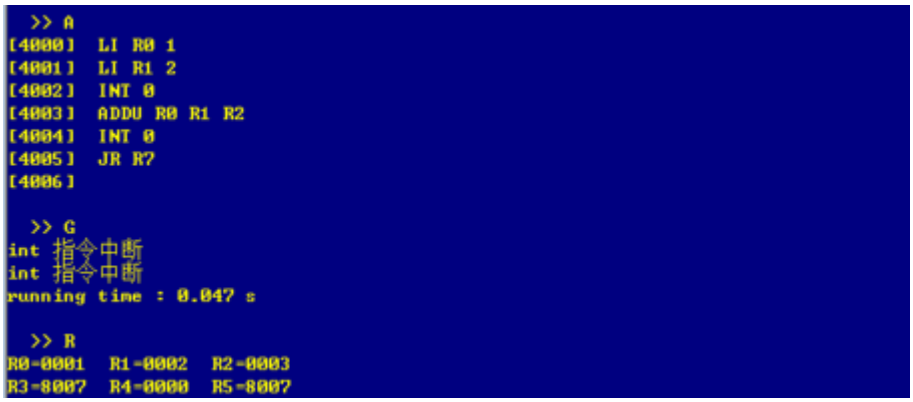
```
if id_ins(10 downto 8)= "110" and int_jr ='1'    then
        int_jr <= '0';
        int_recover <= '1';
end if;
```

设 int_recover=1，int_jr=0。最后，在 WB 第 4 阶段，若检测到

int_recover=1 则把原先保存的 R0、R1 恢复，如下图。

```
if int_recover ='1' then
        r0 <= int_r0;
        r1 <= int_r1;
end if;
```

运行结果：



八、遇到的问题及解决

1.处理中断时，当恢复现场时，R3 的值没有恢复，通过读监控程序对于中断部分的代码（如下图）

```
LW_SP R7 0x0000

ADDSP 0x0001
ADDSP 0x0001
NOP
MTIH R3;
JR R6
LW_SP R3 0x00FF

NOP
```

发现对 R3 的处理位于“JR　R6”之后，这样 R3 没有恢复这一问题也就得到了解释。

2.处理 INT 指令中断时，我们发现如果中断号不为 0，则不会输

出“int 指令中断”这行字，而是输出 ASCII 码值为中断号的符号。因此，我们查阅了 Term 程序的 C++源代码，如下图，

```
switch(ch)
{
    case 0x0f:
        return;
    case 0x10:
        cout<<"外部中断"<<endl;
        break;
    case 0x20:
        cout<<"时钟中断"<<endl;
        break;
    case 0x00:
        cout<<"int 指令中断"<<endl;
        break;
    default:
        printf("%c",ch);
        break;
}
break;
```

这表明只有当地址为 0000 时才会输出该行字，且地址为 000F 时是直接 return 的。但是，根据监控程序，如下图，

```
;提示终端，进入中断处理
LI R3 0x000F
```

000F 时进入中断处理，这与 Term 程序有些矛盾。

我们猜测，应当将监控程序改为 LI R3 0X0000，即 0000 时进入中断处理，而在“int 指令中断”的下一行输出中断号，且 Term 程序中的 printf(“%c”,ch)应该改为 printf(“%c”,ch+48)，输出真正的中断号。

九、实验分工

陈冲：负责 IF 阶段、ID 阶段代码，主要冲突的解决，扩展的实现

赖涵光：负责数据通路及阶段寄存器的设计，EXE 阶段代码，测试代码的编写

蔡文静：负责旁路设计，MEM 阶段、WB 阶段代码

调试由三人共同完成。

十、实验心得

此前就听过学长对计原大实验的各种评价——工作量大，难度高，调试费时，需要熬很多夜，“灵魂升华”，等等。但当自己真正面对这一挑战时，其实并没有想那么多，只是想要尽快地上手。

一开始是难度重重的，真正自己需要设计 CPU 时才发现对之前的课程内容，包括五级流水的过程和细节、冲突的解决等，都理解不够透彻，我们只得再翻出课堂的 ppt 仔细研读。数据通路及阶段寄存器都设计完成后，代码的编写还是比较顺畅的。

但调试阶段又是一个极其考验细心和耐心的过程。由于硬件描述语言的特殊性，调试起来比较麻烦，而且耗费时间长，因此我们必须十分细心全面地编写代码、查找 bug，甚至到了检查的前一天，我们还查出 SLL 指令的一个 bug。而陈冲同学在检查前两天由于调试不断出错，一度要放弃实现中断功能，但最终耐心还是战胜了一切。

通过三周的大实验，在课程上，我们对于计算机的组成和 CPU 的工作原理有了更加深入的理解，也能够更加熟练地掌握硬件描述语言。在其他方面，我们体会到了团队的重要性，享受到了团队合作的快乐。同时，这次大实验也是对我们身心的一次大考验，最终我们战

胜了调试过程中产生的焦躁情绪、熬夜带来的困顿、其他科作业的步步紧逼，高质量完成了此次大实验。

十一、对本课程的建议

1.建议延长大实验的时间，分阶段完成，而不要仅限于三周。

2.前期可以采取适当的方式加深学生对课堂核心内容的理解，避免大实验开始时手忙脚乱。

最后，衷心感谢刘卫东老师、李山山老师和各位助教对我们的悉心指导，让我们能够圆满完成此次大实验，特别是刘卫东老师在小班实验课上对我们组提出的表扬和批评，让我们受益匪浅。

NaiveMIPS Requirements and Design

王邈　　赵晟佳　　张宇翔

January 18, 2016

Table 1: Resvision History

时间	作者	内容更改
2015.10.23	赵晟佳	初始版本
2015.11.18	张宇翔	CPU 及外设文档
2015.12.23	赵晟佳	UMA 更新
2015.12.31	张宇翔	NaiveMIPS++ 说明更新
2016.1.2	王邈	增加 GPU、性能测试程序部分

Contents

Chapter 1

Introduction

1.1 项目背景

本项目为清华大学计算机科学与技术系计算机组成原理课程、软件工程课程、操作系统课程、编译原理课程联合实验。其目标是设计一颗部分兼容于 MIPS32 体系结构的 CPU，在 FPGA 硬件平台上验证，并能够运行 ucore 操作系统，和 Decaf 语言编写的应用程序。项目开发目的是，让学生在自主完成计算机底层的组件的开发的过程中，深入理解计算机系统原理，锻炼系统层面开发的能力，体会项目管理和协作的方法。

1.2 需求方代表

- 计算机组成原理课程：刘卫东老师
- 软件工程课程：白晓颖老师
- 操作系统课程：向勇老师
- 编译原理课程：王生原老师

1.3 术语定义

本文档中出现的术语缩写定义如下：

术语	定义
MIPS	无内部互锁流水线微处理器
CPU	中央处理器
Cache	高速缓存
ALU	算数逻辑单元
MMU	内存管理单元
TLB	翻译后备缓冲区
RAM	随机访问存储器
SRAM	静态随机访问存储器

SRAM	同步静态随机访问存储器
GPIO	通用输入输出口
ROM	只读存储器
JTAG	联合测试行动小组
Flash	快闪存储器
FPGA	现场可编程逻辑门阵列
CP	协处理器
API	应用程序接口
GDB	GNU 调试器
GCC	GNU 编译器套件

Chapter 2

Requirements

项目需求分别从 3 门课程实验要求中产生，并在分析基础上综合得出。

2.1 MIPS32S 指令系统

2.1.1 简述

本项目的核心需求是设计部分兼容于 MIPS32 体系结构的 CPU，该部分是《计算机组成原理》课程的实验要求，也是项目其余部分的基础。CPU 在系统时钟的驱动下，在一个至多个周期内解释执行一条指令，并按指令操作数据，而后继续获取执行下一条指令，如此往复，达到运行计算机程序的目的。

2.1.2 MIPS ISA

在本项目中，根据课程要求，CPU 支持的指令集为 MIPS32 指令集的子集，该指令集包含的指令类型如下:

- 加载、存储指令，如：LB、LW、SB、SW
- 简单算数运算指令，如：ADDI、SUB
- 逻辑运算类指令，如：ANDI、ORI、XOR、SLL、SRA
- 乘除法相关指令，如：MUL、MADD、DIV、MFHI、MTLO
- 分支与跳转指令，如：J、JAL、JR、BEQ、BGEZ
- 条件移动指令，如：MOVZ、MOVN
- 异常相关指令，如：SYSCALL、ERET
- 系统控制指令，如：MFC0、MTC0、TLBWI、CACHE

原课程要求给出的指令集有 48 条指令，覆盖 GCC 4.6 编译器编译 uCore 操作系统后产生的所有指令。但我们出于兼容性和可扩展性考虑，将需求改为实现 69 条指令，覆盖 MIPS32 Release1 规范中，除了浮点运算、非对齐访存之外，所有 GCC 可能自动生成的指令。从而使得我们增加 uCore 代码和启用优化选项后，仍然能够正确在 CPU 上正确执行。

完整的指令集在附录 4.1中列出。

CPU 中的 32 个通用寄存器，PC 寄存器，LO、HI 寄存器，应当按照 MIPS32 规范实现。

作为 CPU 中的核心运算部件——ALU 的算术与逻辑运算规则，参照 MIPS32 规范中对于指令的行为描述实现。

从性能角度出发，我们确定 CPU 设计为 5 级流水。CPU 中的控制器配合数据通路设计，解决数据冲突、控制冲突和结构冲突。所有分支指令后，均存在延迟槽，可用于编译时的指令重排序优化。

2.1.3 Cache

作为课程提高要求之一，CPU 还将包含两级 Cache。其中一级 Cache（以下简称 L1）分为指令和数据 Cache 两部分，分别用于指令和数据访存的加速。二级 Cache（以下简称 L2）为指令和数据共享 Cache。

L1 能够保证在命中时提供 0 周期的访问延时，从而避免流水线在取指和（数据）访存阶段暂停。L2 经由总线，从 RAM 中加载数据，或向 RAM 写入数据时，支持突发传输，提高总线利用率。

2.1.4 异常处理

作为课程要求之一，CPU 支持异常处理。需要支持的异常类型，为 MIPS32 规范的子集。其中包括，由外部硬件信号触发的特殊异常——中断异常。

由于采用流水线设计，要求 CPU 支持精确异常处理。即 CPU 能准确记录发生异常的指令位置（包括位于延迟槽中的指令），并确保异常发生之前的指令均完全执行，且发生异常的指令及之后的指令取消执行。

异常处理流程参照 MIPS32 规范。

关于对异常支持的需求，详细描述见 uCore 需求分析中 2.2.4一节。

2.1.5 CP0

CP0 用于管理硬件，其中包含多个特殊功能寄存器，来配置各项功能。需要实现特殊功能寄存器包括如下几个方面：

- 异常处理：实现一些寄存器，用于保存发生异常时的一些信息，以供异常处理程序使用
- 虚存管理：实现用于配置 TLB 功能的寄存器，与 TLB 配合完成内存管理
- 功能设置：包括系统定时器、中断使能等功能配置

根据 uCore 需求分析 2.2.6，要实现的寄存器和字段的详细解释在附录 4.2中列出。注意，CP0 实现并不完全符合 MIPS32 规范，因为某些规范要求必须实现的字段并未实现。

2.1.6 MMU 支持

MIPS32 规范中将虚拟地址划分为多个区间：

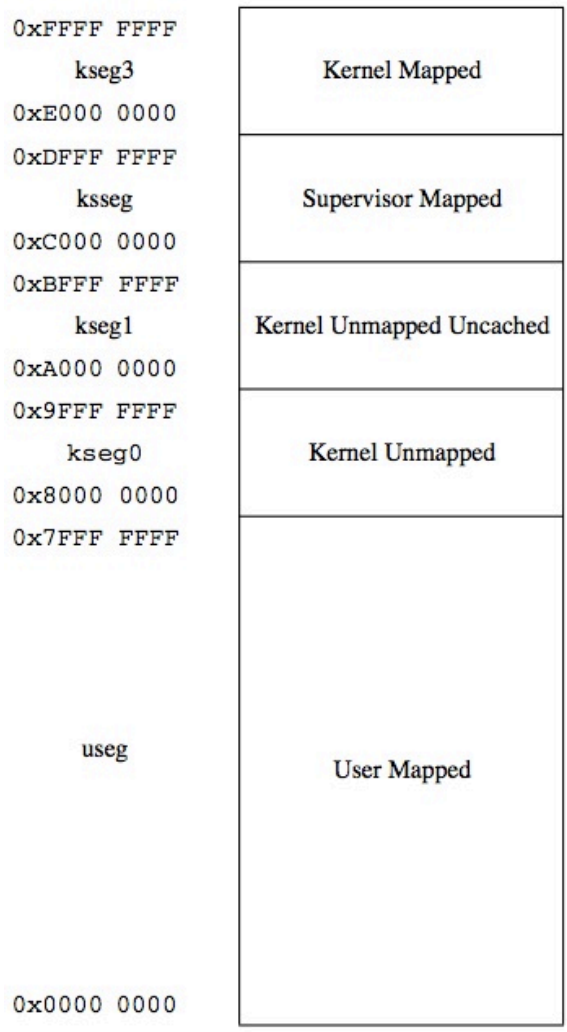


Figure 2.1: Virtual Address Space

其中 kseg3、ksseg 和 kuseg 使用 TLB 映射至物理地址，其余区间直接映射至物理地址。

直接映射

使用直接映射方式时，将 32 位虚拟地址的高 3 位清零后即可得到对应的物理地址。

TLB

TLB 是一种全关联的用于转换虚拟地址的结构，uCore 依赖 TLB 实现应用程序的虚拟地址至 RAM 上物理地址的映射。TLB 分为指令和数据两部分，分别用于取指和（数据）访存时地址转换操作，两者共享相同的转换条目。

TLB 的结构中有若干条目。每一条条目中由两个有机结合的部件构成。这两个部件分别是分别是比较段和物理翻译段。本项目中，需要实现的部件有：

- 比较段
  - 虚拟页编号（VPN）
- 物理翻译段
  - 物理页帧号（PFN）
  - 合法位（V）

为配合 TLB 工作，CP0 中需要实现相应的字段，异常处理中也需要支持 TLB 缺失异常。详细分析见 uCore 需求分析 2.2.5。

2.1.7 外设支持

串口控制器

串口控制器的用途是使得项目中的 CPU 上运行的程序，能够与通过串口相连的计算机交换数据。

串口控制器作为总线上的外设，用于处理兼容 RS-232 标准的串行数据通讯。控制器需支持特定的串行数据参数配置，即 115200 波特率、8 比特数据、1 比特停止位、无奇偶校验位，不实现硬件流控功能。

对内部，该控制器包含一个总线从设备接口，一路中断信号输出。从设备上包括 3 个特殊功能寄存器，其功能描述如下：

- 状态寄存器：包含接收非空标志位、发送空标志位
- 发送数据寄存器：用于存储当前要发送的字节
- 接收数据寄存器：用于存储最新收到的字节

其中状态寄存器可读写，发送数据寄存器只写，接收数据寄存器只读。因此发送数据寄存器和接收数据寄存器可以共享一个总线地址。

行为上，发送空标志位有效时，向发送数据寄存器的一次写操作，可以触发一个字节的串行数据发送过程。而当串口接收一个字节后，接收非空标志位自动置位，从接收数据寄存器可以读出收到的数据，同时接收非空标志位自动复位。

RAM 控制器

在硬件平台上，SRAM 芯片与 FPGA 相连，用于存储运行时的指令和数据。由于 SRAM 接口时序与内部总线不兼容，因此需要 RAM 控制器作协议转换。

RAM 控制器作为总线上的外设，接受来自 CPU 的读写请求，并对 SRAM 芯片产生相应的控制信号时序。该控制器将 RAM 空间直接映射到总线上的一段地址空间，CPU 可以直接在该地址空间上读写 SRAM。

对于无 Cache 的系统，RAM 控制器还需要将外部 SRAM 虚拟成一个双端口 RAM，即可以在一个总线周期同时完成两个不相关的读写请求，从而避免 CPU 流水中取指和（数据）访存的结构冲突。因此这种情况下 RAM 控制器的工作频率，需要是总线频率的 2 倍以上，利用时分复用实现双端口特性。

Flash 控制器

Flash 芯片位于硬件平台上，作为一种非易失存储介质，在掉电时保存程序和数据。由于 Flash 接口时序与内部总线不兼容，因此需要 Flash 控制器作协议转换。

Flash 控制器作为总线上的外设，接受来自 CPU 的读写请求，并对 Flash 芯片产生相应的控制信号时序。该控制器将 Flash 空间直接映射到总线上的一段地址空间，CPU 可以直接在该地址空间上读写 Flash。

由于 Flash 的访问周期较长，一次读操作可能占用多个总线周期，因此 Flash 控制器可以发出从设备忙信号，指示 CPU 需要等待一个至多个周期，直到 Flash 芯片准备好。

BootROM

BootROM 是一个片内的 ROM，其中固化了引导代码。在上电复位时，CPU 中的指令地址将指向 BootROM 所在地址空间，执行 BootROM 中的引导代码，由引导代码完成各种引导操作（如从串口引导、从 Flash 引导等）。引导代码的功能描述详见 NaiveBootloader 的设计说明章节 3.10。

2.2 uCore 操作系统

2.2.1 简述

《软件工程》实验核心要求是在自己设计的 MIPS 处理器上运行 uCore 操作系统。因此在处理器设计过程中需要对 uCore 依赖的处理器功能进行分析，并实现这些功能从而支持 uCore 系统运行。

2.2.2 uCore 说明

课程提供已经移植到 MIPS32 平台的 uCore 操作系统及其源代码。提供的 uCore 有两个版本，陈宇恒移植的版本[1]（以下简称 cyh 版），和 armcpu 小组在前者基础上修改的版本[2]（以下简称 armcpu 版）。

Known Issue

cyh 版的 uCore 中存在一些 bug，这些 bug 在前人的实验中已经部分被发现并得到的修复，也有一些在我们开发过程中被发现，这里一并列出：

- **kern/trap/vectors.S** 文件中的中断向量表，即 **___exception__vector** 可能由于汇编器优化导致顺序、位置出错
- **kern/trap/trap.c** 文件中的异常处理函数 **trap__dispatch** 中对非法指令即 EX_RI 异常的处理未判断是否来自内核态，导致错误的用户态程序可能导致 kernel panic
- **kern/driver/console.c** 文件中串口中断处理函数 **serial__int__handler** 中判断了 COM_IIR 寄存器的值，确定是否有中断发生。但该寄存器仅存在于 QEMU 模拟器中，因此应当增加条件编译宏定义，不编译进 FPGA 版本
- **kern/driver/ramdisk.c** 文件中 **check__initrd** 函数在输出 magic 值时，将 _initrd_begin 直接转为 int* 并访存，将导致编译器产生非对齐访存指令，可以通过修改程序来避免

上述问题的修复补丁已经通过 pull request 合并到 cyh 版本的 master 分支，问题得到解决。

2.2.3 启动过程

uCore 由固化在 FPGA 内部 ROM 中的引导程序，从 Flash 拷贝到 RAM 中运行。操作系统入口位于 **kern/init/entry.S** 中的 **kernel__entry** 函数。这部分代码由汇编编写。在准备完成 C 的运行环境后，进入 **kern/init/init.c** 中的 **kern__init** 函数，开始内核初始化流程。

操作系统初始化过程中，与硬件相关的主要步骤依次为 TLB 初始化、中断控制器初始化、串口控制器初始化、系统定时器中断初始化。CPU 需要正确地支持这些初始化操作。

2.2.4 MIPS 异常依赖

在操作系统运行过程中，时钟中断、外设中断和 TLB 缺失等异常会时常发生，异常处理程序入口有预先放置的代码用于处理异常。uCore 操作系统依赖一些必要的异常才能正常工作，下面将分析 uCore 对异常的需求。

所有异常的入口都位于 **kern/trap/vectors.S** 文件中，标签 **___exception__vector** 指向的代码地址——即异常向量表基址——将在内核入口（**kern/init/entry.S** 中的 **kernel__entry** 函数）中写入 CP0 的 **EBase** 寄存器。而 **___exception__vector** 中的多条跳转指令及空指令，构成了异常向量表。与 MIPS 规范相同，向量表中 +0 偏移处为 TLB 相关异常，+0x180 字节偏移处为其它异常的入口。发送异常时，MIPS 处理器应当根据发生异常的类型，跳转至 EBase+ 偏移地址处，开始执行异常处理程序。

[1] https://github.com/chyh1990/ucore-thumips
[2] https://git.net9.org/armcpu-devteam/armcpu/tree/master/ucore

异常返回操作由 **kern/trap/exception.S** 中的 ERET 指令完成。

异常发生和返回时均会涉及 CP0 中相关字段的数值更新，其流程与 MIPS32 规范一致，这里不再赘述。

异常处理程序进行必要的保存现场操作后，进入 **kern/trap/trap.c** 文件中 **trap_dispatch** 函数，其中判断了发生异常的类型。分析这段代码可知，uCore 实际用到的异常有：

1. **EX_IRQ** 外部中断及系统定时器中断
2. **EX_TLBL** 执行加载指令时发生 TLB 缺失
3. **EX_TLBS** 执行存储指令时发生 TLB 缺失
4. **EX_RI** 解码时发现无效指令
5. **EX_SYS** 执行 SYSCALL 指令
6. **EX_ADEL** 执行加载指令时地址非对齐
7. **EX_ADES** 执行存储指令时地址非对齐

这些异常是 CPU 必须支持的。其中 **EX_IRQ** 异常由各个中断信号触发，它们是：

1. 系统定时器中断：在系统定时器计数匹配时触发，用于操作系统调度。中断号 7
2. 串口中断：在串口收到数据时触发。中断号 4

另外，所有的中断可以被 CP0 的 **Status** 寄存器中 **IE** 字段屏蔽，当其设为 0 时所有中断无效。uCore 会在内核初始化的末尾处，调用 **kern/driver/intr.c** 中的 **intr_enable** 函数将 **IE** 置 1，以启用中断。

2.2.5 TLB 依赖

uCore 利用 TLB 配合内存中的数据结构实现虚存管理。当某次 TLB 缺失触发异常后，**kern/trap/trap.c** 文件中的 **handle_tlbmiss** 函数将被调用，该函数查询内核数据结构，正常情况下将查到的虚地址与物理地址映射关系写入 TLB。实际 TLB 写入发生在 **kern/include/thumips_tlb.h** 文件中的 **tlb_refill** 函数内。

从地址的计算过程可以看出，uCore 中物理地址和虚拟地址均为 32 位，内存页大小均为 4KB。故地址的低 12 位作为页内偏移，不做转换，高 20 位由 TLB 转换。TLB 共有 16 项，其中相邻两项作为一组，每次必须同时写入。TLB 写入操作需要依赖 CP0 中的 Index、EntryLo0、EntryLo1、EntryHi 寄存器来传递参数，最终通过 TLBWI 特权指令完成两条 TLB 记录的写入。参数格式如下表所示：

Item	VA[31..12]		PA[31..12]	Valid
Index.Index*2	EntryHi.VPN2	0	EntryLo0.PFN	EntryLo0.V
Index.Index*2+1	EntryHi.VPN2	1	EntryLo1.PFN	EntryLo1.V

2.2.6 CP0 依赖

uCore 对于 CP0 的依赖，除了上文已经阐述的异常、TLB 相关控制寄存器之外，还有系统定时器功能。

系统定时器是一个独立于 CPU 运行的计数器，该计数器在每个 CPU 时钟周期加 1，并在与预设的匹配值相等时触发定时器中断。uCore 利用该中断获得 CPU 控制权，进行进程的调度。

系统定时器的计数值和匹配值分别由寄存器 **Count** 和 **Compare** 保存，uCore 中相应的配置在 **kern/driver/clock.c** 文件中。分析代码可知，在每次发生定时器中断时，uCore 会将匹配值设定为当前计数值 +TIMER0_INTERVAL，从而使得中断周期性发生，同时，中断时还将运行等待的任务以及对时间戳变量加 1。

综上所述，CP0 中需要实现的寄存器和字段的完整信息在附录 4.2中列出。

2.2.7 串口

uCore 使用了串口控制器用于日志输出和用户交互。串口控制器驱动代码在 **kern/driver/console.c** 中，其中宏定义 COM1 为 0xbfd003f8，即串口控制器的数据寄存器所在的地址，而状态寄存器地址为 0xbfd003f8+4。

串口控制器的状态寄存器第 0 位为发送空标志位，第 1 位为接收非空标志位。

串口控制器的行为描述，见 2.1.7小节。

2.3 On-Chip Debugger

2.3.1 Overview

作为《软件工程》的课程项目要求之一，NaiveMIPS 需要实现硬件的片上调试模块，及其配套的上位机调试工具。调试工具能够实现指令级别的单步运行、断点，寄存器和内存查看等功能。

为了达到这些功能需求，我们将整个调试工具拆分为 3 部分，分别为片上调试模块、上位机调试协议代理和 GDB 调试器。

2.3.2 On-Chip Module

片上调试模块一方面通过硬件信号监控和控制 CPU，另一方面通过某种专用的通信接口（如串口、JTAG）与上位机通信。与 CPU 的信号连接包括流水线清空、暂停信号，当前指令地址，寄存器读取控制信号、总线（内存）读取控制信号等。在上位机的指挥下，片上调试模块发送或接收这些信号，从而达到控制 CPU 运行和读取各种信息的目的。

2.3.3 Debugger Agent

调试协议代理程序工作在上位机上，一方面通过专用接口与片上的调试模块通信，另一方面通过 TCP 与 GDB 通信。该程序的主要工作是将 GDB Remote Protocol 中的指令翻译成简单的控制指令发送给硬件，并将硬件返回的数据封装成 GDB 要求的格式。这样 GDB 就能与片上的模块通信了。

2.3.4 GDB

GDB 是一个支持多种平台的开源调试工具，提供强大的指令级和源码级调试功能。GDB 可以用 GDB Remote Protocol 与远端的软硬件建立连接，从而调试远端运行的程序。GDB 完全可以满足课程对于调试器的要求，且有极大的实用价值。因此我们选择 GDB 作为调试工具，并通过实现其规定的协议来使 GDB 支持 NaiveMIPS。

2.4 Memory System

2.4.1 Introduction

内存系统应当为 NaiveMIPS core 提供软件透明的内存访问功能，同时，通过两级 cache 提高性能。同时，内存系统应当按照 MIPS 内存管理标准，提供 cached 地址区域和直连地址区域。

内存系统还应当支持多从总线以及恰当的地址转换方式，以及对于冲突访问的仲裁功能

2.4.2 ICache

ICache 是 CPU 可以直接访问的取指 cache，如果 CPU 访问了一个存储在 cache 中的地址，那么所读写的数据应当在本周期内异步地提供或者写入，否则应当将 MISS 信号置高，直到所需的数据已经加载到 cache 中。ICache 有 512B 的存储，组织方式为 16 个 cache line，每个 cache line 有 32B，固定映射

2.4.3 DCache

DCache 是 CPU 可以直接访问的数据 cache，如果 CPU 访问了一个存储在 cache 中的地址，那么所读写的数据应当在本周期内异步地提供或者写入，否则应当将 MISS 信号置高，直到所需的数据已经加载到 cache 中。当对 DCache 进行了写入，应当将写入结果同步到 ICache 中。DCache 有 512B 的存储，组织方式为 16 个 cache line，每个 cache line 有 32B，固定映射

2.4.4 L2Cache

L2Cache 是大容量，但具有较高的访问延迟的存储器。L2Cache 应当支持 DCache 和 ICache 的同时访问，并提供同时访问的仲裁功能。如果 DCache 和 ICache 发出了读取或写入一个 cache line 的请求后，如果该 cache line 在 L2Cache 中，应当应许该请求，并进行相应的操作，否则应当将 BUSY 信号置高，向总线发出读写请求，并去出相应的 cache line

L2Cache 采用 16 相联的方式组织，每个地址可以映射到 16 个 cache line 中。每次需要加载一个 cache line 时，如果 16 个 cache line 有一个空位，则使用空闲的位置，否则去除最近最远访问的 cache line。当对 cache line 的某个地址写入后，还需将此 cache line 标记为 dirty，如果需要去除该 cache line，需要先将修改写回到内存中

L2Cache 有 128KB，每个 cache line 有 64B

2.4.5 Bus Controller

总线控制器是一个多主多从的总线，需要至少支持 4 个主设备和 8 个从设备。总线协议需要支持

- 主设备发出读写请求，总线仲裁，在总线有空闲周期时选择优先级高的设备，批准其请求
- 主设备请求得到准许后，发出一个或者多个读或写请求，但不可以读写交替
- 从设备监听总线地址，当总线地址与自身的有效地址匹配时，发出匹配的 ACK 信号，并开始读取总线上的读或写命令
- 从设备在规定的时间完成操作，如果读写不能按时完成，需要将 BUSY 信号置高，此时总线通知主设备暂停，直至从设备可以继续响应请求

2.5 Decaf

2.5.1 简述

NaiveMIPS 项目的 Decaf 部分需求源于《编译原理》课程拓展实验，其核心需求是构建一个 Decaf 语言的交叉编译工具链，将 Decaf 源文件编译为 MIPS 架构的 uCore 操作系统上的用户态可执行文件。

2.5.2 已有资源

Decaf 部分的开发在已有的资源上开展，它们包括：

1. MIPS Decaf 编译器主体，包括完整的前后端实现

2. MIPS GCC 工具链，包括 C 编译器、汇编器、链接器等

3. MIPS 平台上的 uCore 操作系统

2.5.3 MIPS 指令

MIPS Decaf 编译器已经能够实现将 Decaf 源文件编译为 MIPS 汇编代码，且汇编代码符合 GNU 汇编器输入格式，能够被转为二进制指令。编译器的后端指令生成代码位于 **decaf/backend/Mips.java** 文件中，从中可以找到所有编译器可能产生的汇编指令及伪指令。分析发现，这些指令（或展开后的伪指令）均已在 CPU 支持的指令集中（见附录 4.1），因此编译器产生的指令汇编后可以直接在 NaiveMIPS CPU 上执行。

2.5.4 运行时库函数

Decaf 语言中包含一些语法元素（如 Print、new）需要运行时库函数支持。编译器中规定了必要的运行时库函数接口，接口描述位于 **decaf/machdesc/Intrinsic.java** 文件中。当前编译器中不包含库函数的实现，因此需要我们自行实现。

这些库函数分为如下几类：

- **内存分配** _Alloc
- **字符串比较** _StringEqual
- **数据输入** _ReadLine _ReadInteger
- **数据输出** _PrintInt _PrintString _PrintBool
- **程序退出** _Halt

库函数功能详细解释见设计说明 3.11.1小节。

2.5.5 uCore 接口

uCore 系统对于应用程序提供了一些系统 API，这些 API 与 C 标准库类似，实现在 uCore 代码目录的 **user/libs/**子目录中。其中可能被 Decaf 程序使用到的 API 有：

- **read** 从文件（标准输入）读取一个或多个字节
- **printf** 格式化输出
- **strcmp** 字符串比较
- **exit** 退出当前进程

Decaf 运行时库将基于这些 API 实现，从而实现 Decaf 程序在 uCore 上运行。

Chapter 3

Design

3.1 Overview

NaiveMIPS 的整体硬件设计为一个 System-on-Chip，即在一块 FPGA 芯片上实现 CPU 和各种外设、接口等组件。从资源消耗的角度的出发，我们将设计分为两个不同的版本，即带有 Cache 及完整总线支持的版本（后文简称 NaiveMIPS++），和一个简化总线且不带有 Cache 的版本（后文简称 Basic 版本）。

Basic 版可以在 Thinpad 实验平台上运行，而 NaiveMIPS++ 由于逻辑资源占用超过硬件容量，只能在逻辑资源更加丰富的 DE2i 开发板上运行。后文介绍中，对于两个版本有区别的地方将单独说明。

项目中必要的软件部分有 NaiveBootloader 和 uCore 操作系统两部分。前者位于片内的 BootROM 中，在上电复位后运行，用于加载和引导 uCore 操作系统，也可兼作 Flash 编程工具和硬件测试工具。uCore 操作系统一般存放在 Flash 芯片中，由 NaiveBootloader 加载至内存，然后在内存中运行。操作系统引导完成后，可以运行用户态程序，并且有命令行界面与用户交互。

作为附加要求，项目中还包括 NaiveDebugger 调试器，是一个由软件硬件协同构成的指令级调试工具，可以与 PC 联机调试片内程序，实现单步运行、断点、数据观察等功能。此外，我们还完善了 Decaf 编译工具链，使得用 Decaf 语言编写的程序可以编译后在 NaiveMIPS 的 uCore 环境下，作为用户态应用程序运行。

3.1.1 硬件平台

本系统将在真实硬件平台上运行验证，验证平台分为 Thinpad 和 DE2i 两种。

Thinpad

该平台由计算机原理课程实验室提供，技术参数如下：

组件	数量	型号/参数
FPGA	1	Xilinx Spartan-6 XC6SLX100
SRAM	4	4 片总共 2M × 32bits
Flash	1	4M × 16bits
CPLD	1	Xilinx XC95144XL
串口	3	

数码管	2	
LED	16	
PS/2 接口	1	
拨码开关	32	
按钮开关	4	
晶振	2	11.0592M、50M
以太网控制器	1	DM9000A Fast Ethernet Controller
VGA 接口	1	3bits DAC / Channel
USB-OTG 控制器	1	ISP1362

DE2i

该平台由 Terasic 公司设计制造，为 CPU+FPGA 架构，我们只使用了其中 FPGA 部分，其技术参数如下:

组件	数量	型号/参数
FPGA	1	Altera CycloneIV EP4CGX150DF31C8
SSRAM	4	两片总共 1M × 32bits
Flash	1	32M × 16bits
串口	1	
数码管	8	
LED	18	
拨码开关	18	
按钮开关	4	
晶振	1	50M

3.1.2 开发环境

由于需要兼容两种硬件平台，我们同时维护了 Altera 和 Xilinx 开发环境的工程文件，它们分别位于 **HDL/altera/NaiveMIPS.qsf** 和 **HDL/xilinx/NaiveMIPS/NaiveMIPS.xise**。对于两者用到的的 IP Core 接口都进行了封装，使其它部分的代码能够兼容两个平台。

对于 Altera 环境，即 DE2i 开发板，使用的编译环境是 Quartus 13.1。

对于 Xilinx 环境，即 Thinpad，使用的编译环境是 ISE 14.7。

硬件部分开发语言选择 Verilog HDL，部分 Testbench 用了 SystemVerilog 语言编写。

3.2 CPU

3.2.1 Overview

NaiveMIPS 的 CPU 设计为 5 级流水，实现 69 条指令，完整指令集在附录 4.1中列出。

CPU 中的数据冲突采用数据前推的方法解决，即如果发现后级存在尚未写入某个寄存器的数据，而当前又引用了那个寄存器的值时，就直接使用来自后级的值。分支指令造成的控制冲突通过延迟槽解决，所有分支指令后，均存在延迟槽，可用在编译时通过指令重排序充分利用。

CPU 实现了精确异常处理，实现方法是对于各阶段产生的异常均不立即处理，而是随流水线一直推至访存阶段。在访存阶段统一检查之前的阶段，以及访存本身有无异常发生，如果有异常则发出信号给控制器，控制器会清空流水线，并设置 PC 为异常处理入口。

Basic 和 NaiveMIPS++ 两者在 CPU 设计上的主要不同，在于总线访问部分。Basic 版本直接将指令和数据总线对外暴露；而 NaiveMIPS++ 将指令和数据总线分别与 ICache 和 DCache 相连，对外暴露统一的外设总线。另外，片上调试器模块暂未移植到 NaiveMIPS++。

CPU 内部使用的宏定义在 **HDL/src/cpu/defs.v** 文件中，后文中提到宏定义，如不特殊说明，均指该文件中的定义。

3.2.2 接口

CPU 模块的顶层在 **HDL/src/cpu/naive_mips.v** 文件中描述，该模块的接口为 CPU 的边界。

Basic 版本

Name	Width	Direction	Description
rst_n	1	In	CPU 异步复位信号，低有效
clk	1	In	CPU 主时钟
ibus_address	1	Out	指令总线地址线
ibus_byteenable	3..0	Out	指令总线字节使能
ibus_read	3..0	Out	指令总线读使能
ibus_write	1	Out	指令总线写使能（预留，暂未使用，恒为 0）
ibus_wrdata	31..0	Out	指令总线写数据（预留，暂未使用，恒为 0）
ibus_rddata	31..0	In	指令总线读数据
dbus_address	31..0	Out	数据总线地址线
dbus_byteenable	3..0	Out	数据总线字节使能
dbus_read	1	Out	数据总线读使能
dbus_write	1	Out	数据总线写使能
dbus_wrdata	31..0	Out	数据总线写数据
dbus_rddata	31..0	In	数据总线读数据
dbus_stall	1	In	数据总线设备暂停请求信号
hardware_int_in	4..0	In	外部设备中断信号输入
debugger_uart_clk	1	In	片上调试器专用串口，串口时钟
debugger_uart_txd	1	Out	片上调试器专用串口，发送端
debugger_uart_rxd	1	In	片上调试器专用串口，接收端

Basic 版本 CPU 采用指令与数据总线分离式设计，对于访问内存的冲突，在内存控制器中解决。

接口描述中总线是 Basic 总线，相关信号详细说明参见 3.3一节，片上调试器相关信号参见 3.9一节。

NaiveMIPS++

Name	Width	Direction	Description
rst_n	1	In	CPU 异步复位信号，低有效
clk	1	In	CPU 主时钟
bus_address	31..0	Out	总线地址线
bus_read	1	Out	总线读使能
bus_write	1	Out	总线写使能
bus_wrdata	31..0	Out	总线写数据
bus_rddata	31..0	In	总线读数据
bus_stall	1	In	总线设备暂停请求信号
bus_ack	1	In	总线设备应答信号
hardware_int_in	4..0	In	外部设备中断信号输入

由于 CPU 中引入 cache，对外总线接口合并为一个。其总线类型是 UMA 总线，相关信号时序说明参见 3.4.2一节中 Slave Device 接口描述。

3.2.3 Datapath

数据流图 TBD

流水线由取指、译码、执行、访存、写回 5 个阶段构成，每个阶段之间存在一级触发器。在每个 CPU 时钟周期，各个阶段输出的数据同时通过触发器进入下一阶段。

流水线外侧有独立的控制模块，用于控制流水线运行。其控制能力包括暂停流水线的一段，和清空流水线中的数据。

3.2.4 Stage-IF

取指阶段为 5 级流水线的第一个阶段，其功能是从存储器中取出下一条待执行的指令。相关代码位于 **HDL/src/cpu/stage_if/**目录中。

该阶段中包含一个 PC 寄存器，存储下一条指令的地址。PC 在正常情况下每个周期自动加 4，指向后一条指令，在遇到跳转或异常时，将被设定为某个特定地址（如跳转目的地址、异常处理入口）。

PC 寄存器输出的地址经过 MMU 转换后送到指令总线的地址线上，指令总线在同一周期（0 周期延迟）返回的数据即为要执行的指令，该指令被送入译码阶段。在 Basic 版本中指令总线直接暴露给 CPU 外部，而 NaiveMIPS++ 中，指令总线与 ICache 相连。如果发生 ICache 缺失，控制器将产生信号暂停整个流水线。

在指令地址转换为物理地址过程中，可能遇到 TLB 缺失，此时不对 TLB 异常立即处理，而是将异常状态标志向后传递，直到传递至访存阶段后再一并处理。而在发生异常后传给后级的指令为空操作 NOP。

PC 寄存器复位后的值为 0xbfc00000，即片内 BootROM 所在虚拟地址。

PC 寄存器接口

Name	Width	Direction	Description
pc_reg	31..0	Out	PC 寄存器值输出
rst_n	1	In	异步复位，低有效
clk	1	In	时钟信号
enable	1	In	使能信号，只有使能有效时 is_branch 和 PC 计数功能才生效
branch_address	31..0	In	分支跳转的目的地址
is_branch	1	In	是否设定 PC 为分支地址
exception_new_pc	31..0	In	异常处理的目的地址
is_exception	1	In	是否设定 PC 为异常处理地址
debug_new_pc	31..0	In	调试器目的地址
is_debug	1	In	是否设定 PC 为调试器目的地址
debug_reset	1	In	同步复位，调试器用

PC 真值表

debug_reset	is_debug	is_exception	enable	is_branch	PC
H	X	X	X	X	PC ← Initial
L	H	X	X	X	PC ← debug_new_pc
L	L	H	X	X	PC ← exception_new_pc
L	L	L	L	X	PC 值不变
L	L	L	H	L	PC ← PC+4
L	L	L	H	H	PC ← branch_address

3.2.5 Stage-ID

译码阶段负责指令解码、通用寄存器访问、分支判断等工作。相关代码位于 **HDL/src/cpu/stage_id/**目录中。

MIPS32 指令分为 I、J、R 3 种类型，其译码工作分别在 **id_i**、**id_j**、**id_r** 模块中进行。3 种指令译码结果在 **id** 模块中汇总输出。

通用寄存器访问也在译码阶段完成。根据指令解码结果，待访问的寄存器的地址输出到寄存器堆中，寄存器堆返回的数据送入执行阶段。为解决流水线数据冲突，寄存器的值还可能通过数据前推的方式从后级的输出中直接获取，其多路选择器在 **reg_val_mux** 模块中实现。模块中依次判断执行、访存、写回阶段要改写的寄存器地址与当前需要读的寄存器是否相同，如果地址相同且的确是写寄存器操作，就直接输出该阶段要写入寄存器的值。如果没有找到任何一个匹配的，就输出寄存器堆中的值。

分支判断在 **branch** 模块中实现。模块如果发现指令是分支类型，则根据指令解码的结果，计算分支条件，如果满足条件则输出分支使能信号给 PC 寄存器。

译码阶段模块全部为组合逻辑。

id 模块接口

Name	Width	Direction	Description
op	7..0	Out	解码后的指令，取值见宏定义 ‘OP_*
op_type	1..0	Out	指令类型（I、J、R），取值见宏定义 ‘OPTYPE_*
reg_s	4..0	Out	指令中寄存器 s 的地址，没有则为 0
reg_t	4..0	Out	指令中寄存器 t 的地址，没有则为 0
reg_d	4..0	Out	指令中寄存器 d 的地址，没有则为 0
immediate	16..0	Out	I 类指令中包含的立即数，如果不是 I 类指令则为 0
flag_unsigned	1	Out	指令是否为无符号型，即带有 u 后缀
inst	31..0	In	指令输入
pc_value	31..0	In	指令所在的地址，未用

reg_val_mux 模块接口

Name	Width	Direction	Description
value_o	31..0	Out	寄存器值选取结果
reg_addr	4..0	In	要访问的寄存器地址
value_from_regs	31..0	In	来自寄存器堆的值
addr_from_ex	4..0	In	执行阶段要写的寄存器的地址，不写则为 0
value_from_ex	31..0	In	执行阶段要写的寄存器的数据，不写则为 0
access_op_from_ex	1..0	In	执行阶段输出的访存操作，见宏定义 ‘ACCESS_OP_*
addr_from_mm	4..0	In	访存阶段要写的寄存器的地址，不写则为 0
value_from_mm	31..0	In	访存阶段要写的寄存器的数据，不写则为 0
access_op_from_mm	1..0	In	访存阶段的访存操作，见宏定义 ‘ACCESS_OP_*
addr_from_wb	4..0	In	写回阶段要写的寄存器的地址，不写则为 0
value_from_wb	31..0	In	写回阶段要写的寄存器的数据，不写则为 0
write_enable_from_wb	1	In	写回阶段寄存器写使能

branch 模块接口

Name	Width	Direction	Description
is_branch	1	Out	是否存在有效的分支指令
branch_address	31..0	Out	分支目的地址
return_address	31..0	Out	返回地址 Link
inst	31..0	In	指令输入
pc_value	31..0	In	指令所在地址（用于计算目的地址）
reg_s_value	31..0	In	s 寄存器值（用于条件分支）
reg_t_value	31..0	In	t 寄存器值（用于条件分支）

3.2.6 Stage-EX

执行阶段完成实际的算术与逻辑运算，相关代码位于 **HDL/src/cpu/stage_ex/**目录中。

大部分运算指令都可以单周期完成，在 **ex** 模块中实现，表现为一个组合逻辑；而某些运算（如除法）需要多周期完成，因而需要维护一个状态机，在运算过程中保持 stall 信号有效，指示控制器暂停流水线的前级，直到运算完成，状态机在 **multi_cycle** 模块中实现。

算术与逻辑运算规则，参照 MIPS32 规范中对于指令的行为描述完整实现。除法运算器使用来自 OpenCores 网站的一个开源 IP 核[1]。它是一个可配置的参数化除法器，支持无符号除以无符号整数，我们将其配置为 64 位被除数和 32 位除数模式（因为它要求被除数位宽是除数的两倍），但是被除数的高 32 为填 0。对于有符号除法，先将输入的补码取绝对值，再对运算结果修正为有符号的结果。

由于一条指令最多进行一种写操作（写寄存器、写内存、写 CP0 等），所有要写的数据通过一个 data_o 信号输出，并由 mem_access_op 信号指定是哪一种写操作。对于非按字访存的情况，由 mem_access_sz 信号指定访存的长度。

ex 模块还会输出指示特权指令的 is_priv_inst 信号，以及指示有符号运算溢出的 overflow 信号，以便访存阶段的异常检查模块产生相应的异常。

[1] http://opencores.org/project,divider

ex 模块接口

Name	Width	Direction	Description
clk	1	In	时钟
rst_n	1	In	异步复位，低有效
mem_access_op	1..0	Out	访存操作类型，见宏定义 ‘ACCESS_OP_*
mem_access_sz	1..0	Out	访存长度，见宏定义 ‘ACCESS_SZ_*
data_o	31..0	Out	要写入内存或寄存器的数据
mem_addr	31..0	Out	要写入的内存地址
reg_addr	4..0	Out	要写入的寄存器地址
overflow	1	Out	有符号运算溢出
stall	1	Out	多周期运算，流水线暂停信号输出
exception_flush	1	In	异常发生，复位多周期指令状态机
op	7..0	In	解码后的指令，取值见宏定义 ‘OP_*
op_type	1..0	In	指令类型（I、J、R），取值见宏定义 ‘OPTYPE_*
address	31..0	In	写入的返回地址（仅限于分支 Link 指令）
reg_s	4..0	In	s 寄存器地址
reg_t	4..0	In	t 寄存器地址
reg_d	4..0	In	d 寄存器地址
reg_s_value	31..0	In	s 寄存器值
reg_t_value	31..0	In	t 寄存器值
immediate	15..0	In	立即数值（仅对 I 类指令有效）
flag_unsigned	1	In	指令为无符号类型
reg_hilo_value	63..0	In	当前 HI、LO 寄存器的值
reg_hilo_o	63..0	Out	HI、LO 要写入的值
we_hilo	1	Out	HI、LO 写使能
cp0_rd_addr	4..0	Out	CP0 读取地址
reg_cp0_value	31..0	In	从 CP0 寄存器读出的值
cp0_wr_addr	4..0	Out	CP0 要写入地址
we_cp0	1	Out	CP0 写使能
we_tlb	1	Out	TLB 写入使能
syscall	1	Out	是否为 syscall 指令
eret	1	Out	是否为 eret 指令
is_priv_inst	1	Out	是否为特权指令

3.2.7 Stage-MM

访存阶段负责存储、加载指令读写内存,以及异常检查,相关代码位于 **HDL/src/cpu/stage_mm/**目录中。

访存模块在检测到前级模块送来加载操作指示后，通过数据总线读内存，并将得到的数据送至写回阶段；在检测到上级模块送来存储操作指示后，通过数据总线写内存。

由于总线上总是按照 32 位对齐访问的，在访存指令中又存在半字（16 位）、字节（8 位）访存的情况，因此需要根据不同的访存方式和地址偏移，设置总线的字节使能信号，并在写内存时将数据放于数据总线正确的位置上，读内存时从数据总线正确的位置获得数据。

例如，当按使用 LB 指令访问地址 0x03 处的数据时，应当将总线地址设为 0x00，并从数据线的 [31..24] 位上获取数据。

在 CPU 内部的地址均为虚拟地址，数据总线上使用的是物理地址，因此本阶段还会使用 MMU 对地址进行转换，转换方式参见 3.2.9。另外，检测到地址非对齐的访存时，模块将输出对齐异常指示信号。

在 Basic 版本中数据总线直接暴露给 CPU 外部，而 NaiveMIPS++ 中，数据总线与 DCache 相连。如果发生 DCache 缺失或者从设备请求等待，控制器将产生信号暂停流水线访存及之前的阶段。

异常检查模块对于来自前级各类异常信号进行检查，发现异常后输出信号给控制器，清除流水线，开始异常处理流程，同时输出信号给 CP0，记录异常的相关信息。

访存阶段模块全部为组合逻辑。

mm 模块接口

Name	Width	Direction	Description
mem_access_op	1..0	In	访存操作类型，见宏定义 ‘ACCESS_OP_*
mem_access_sz	1..0	In	访存长度，见宏定义 ‘ACCESS_SZ_*
data_i	31..0	In	前级给出要写入寄存器或内存的数据
reg_addr_i	4..0	In	前级给出要写入寄存器的地址
addr_i	31..0	In	前级给出要写入内存的地址
flag_unsigned	1	In	是否为无符号扩展
exception_flush	31..0	In	指示异常发生，未用
mem_address	31..0	Out	数据总线的地址
mem_data_o	31..0	Out	数据总线写数据
mem_data_i	31..0	In	数据总线读数据
mem_rd	1	Out	数据总线读使能
mem_wr	1	Out	数据总线写使能
mem_byte_en	3..0	Out	数据总线字节使能
alignment_err	1	Out	非对齐访存指示
data_o	31..0	Out	要写入寄存器的数据

exception 模块接口

Name	Width	Direction	Description
flush	1	Out	流水线清空请求
cp0_wr_exp	1	Out	CP0 异常相关字段写使能
cp0_clean_exl	1	Out	CP0 EXL 字段清零请求
exp_epc	31..0	Out	CP0 EPC 字段值
exp_code	4..0	Out	CP0 CODE 字段值
exp_bad_vaddr	31..0	Out	CP0 BadV 字段值
exception_new_pc	31..0	Out	异常处理入口地址
iaddr_exp_miss	1	In	指令地址 TLB 缺失
daddr_exp_miss	1	In	数据地址 TLB 缺失
iaddr_exp_illegal	1	In	指令地址非对齐
daddr_exp_illegal	1	In	数据地址非对齐
data_we	1	In	访存为写操作
invalid_inst	1	In	无效指令异常
syscall	1	In	Syscall 异常
eret	1	In	Eret 伪异常
pc_value	31..0	In	当前指令地址
mem_access_vaddr	31..0	In	当前访存虚拟地址
in_delayslot	1	In	当前指令位于延迟槽
overflow	1	In	符号运算溢出
hardware_int	5..0	In	外部硬件中断
software_int	1..0	In	CP0 软中断
allow_int	1	In	全局中断使能
ebase_in	19..0	In	异常入口基址
epc_in	31..0	In	CP0 EPC 字段值
restrict_priv_inst	1	In	非法使用特权指令

3.2.8 Stage-WB

写回阶段负责最终将数据写入寄存器中，相关代码位于 **HDL/src/cpu/stage_wb/**目录中。

该阶段只有一个 **wb** 模块，模块根据前级输入产生一个写使能信号，将值写入寄存器堆中。

wb 模块接口

Name	Width	Direction	Description
reg_we	1	Out	寄存器写使能
mem_access_op	1..0	In	访存操作类型，见宏定义 ‘ACCESS_OP_*
mem_access_sz	1..0	In	访存长度，见宏定义 ‘ACCESS_SZ_*
data_i	1	In	待写入的数据
reg_addr_i	1	In	待写入的寄存器地址

3.2.9 MMU

MIPS 中虚拟地址到物理地址的转换由 MMU 完成，相关代码位于 **HDL/src/cpu/mmu/**目录中。

参照 2.1.6中对虚拟地址段的描述，某些虚拟地址可以通过直接映射的方式转成物理地址，该转换在 **mem_map** 模块中完成。而其余需要用 TLB 转换的段则由 TLB 查表转换，TLB 实现在 **tlbConverter** 模块中。

TLB 为 16 项全相连接结构，每项均包含有效位、物理地址和虚拟地址。转换时硬件实现虚拟地址与 TLB 中有效的条目逐条比较，找到虚拟地址匹配的条目后，返回其表示的物理地址。流程如下：

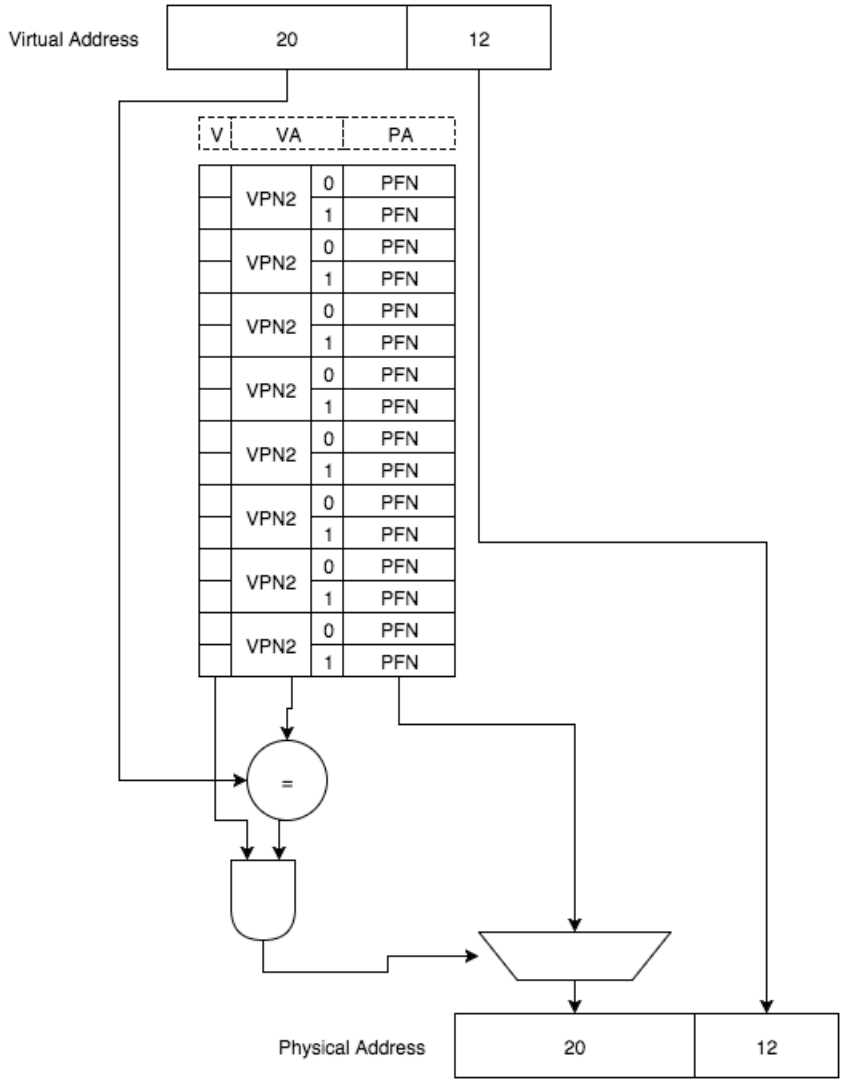


Figure 3.1: TLB 转换流程

3.3 Basic 总线

Basic 总线是一个纯组合逻辑的地址映射模块，用于将各个外设与 CPU 相连，并根据地址空间选择不同的外设。其实现位于 **HDL/src/bus/**文件夹中，分为 **ibus** 和 **dbus** 两个模块，对应指令总线和数据总线的内存映射。

3.3.1 指令总线映射

外设	起始物理地址	长度
RAM	0x00000000	0x800000
BootROM	0x1fc00000	0x100000

3.3.2 数据总线映射

外设	起始物理地址	长度
RAM	0x00000000	0x800000
VGA 控制器	0x1b000000	0x60000
Flash	0x1e000000	0x1000000
串口控制器	0x1fd003f0	0x10
GPIO	0x1fd00400	0x100
精确计时器	0x1fd00500	0x10

3.3.3 NaiveMIPS++

在 NaiveMIPS++ 中，由于 Cache 的引入，解决了结构冲突问题，我们不再单独区分指令和数据总线，指令和数据访存共享一条总线。地址空间映射方式为上述两种映射合并得到。新引入的 uma 总线控制器说明见 3.4.2一节。

3.4 Uniform Memory Access System

3.4.1 System Architecture

Introduction

The UMA system (Uniform Memory Access System) is a bus protocol and related devices designed for NaiveMIPS core. The memory system assigns addressing schemes for every storage and peripheral device. Combined with caching, all memory devices can be treated uniformly by an abstract R/W interface with high performance.

Functions of this system include caching for instruction and data memory, asynchronous memory controllers for peripherals, SRAM and DRAM controllers, and a bus system that supports up to 4 master devices and 8 slave devices

System Diagram

The system contains the following high level components

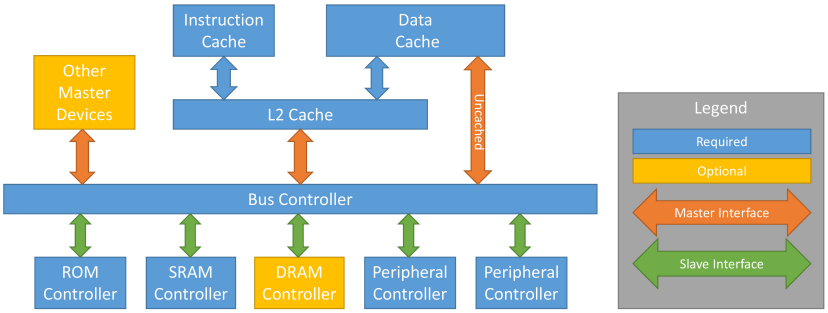


Figure 3.2: Memory System Architecture

The L1 Cache is organized as follows

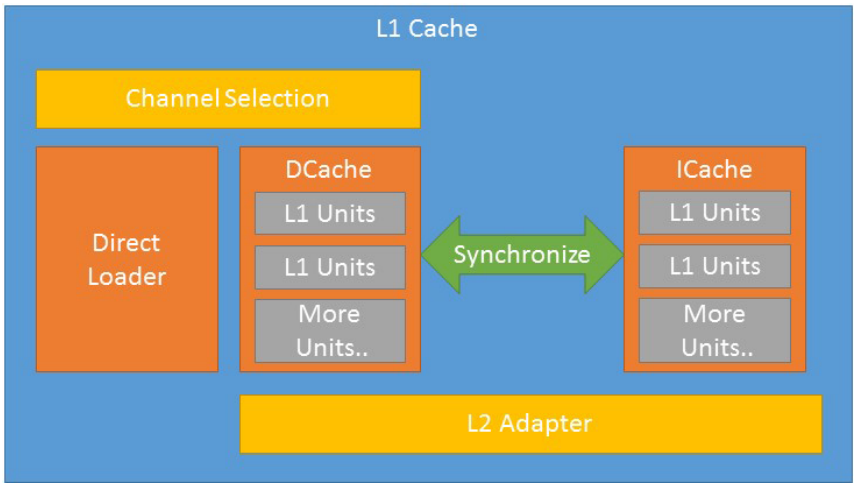


Figure 3.3: L1 Cache System Diagram

The L2 Cache is organized as follows

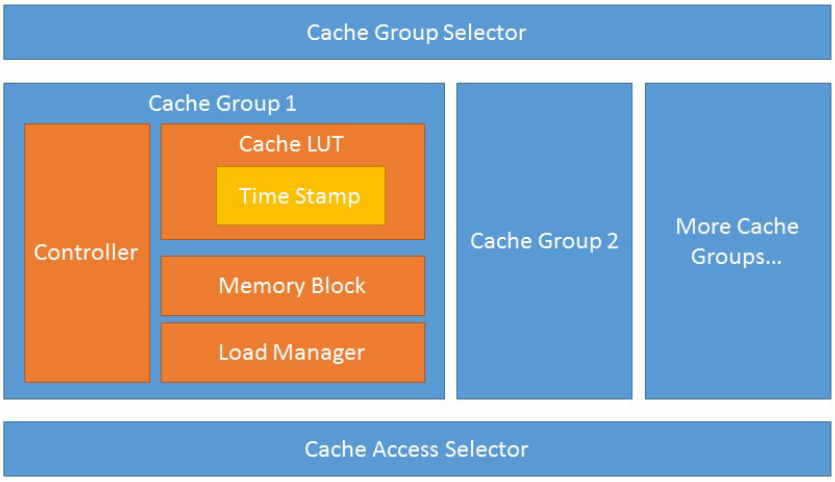


Figure 3.4: L2 Cache System Diagram

3.4.2 Bus Controller

Introduction

The Bus controller consists of a set of master interfaces and a set of slave interfaces. A master device may be connected to a master interface and issue R/W requests to slave devices. A slave device may be connected to a slave interface and respond to R/W requests issued by masters. Both types of devices must conform to the protocol described in this section for correct communication.

Interface

1. **Master Interface**

   At most four master devices can be connected to the bus to issue R/W requests to slave devices. Below is the interface for a single master device

Name	Width	Direction	Description
ADDR	31:0	W	Address to R/W
RDATA	31:0	R	Data requested from slave
WDATA	31:0	W	Data written to the slave
RREQ	1	W	Issue a read request to the slave
WREQ	1	W	Issue a write request to the slave
ACC	1	R	The request to R/W has been accepted
BUSY	1	R	Requested data currently unavailable

Table 3.3: Master Device Interface

2. **Slave Interface**

   At most eight slave devices can be connected to the bus to respond to master requests. Below is the interface for a single slave device

Name	Width	Direction	Description
SADDR	31:0	R	Address to R/W
SWDATA	31:0	W	Data written to slave
SRDATA	31:0	R	Date returned by slave
SR	1	R	A read request is issued to the slave
SW	1	R	A write request is issued to the slave
SBUSY	1	W	Slave busy
SACK	1	W	The address matches the slave device

Table 3.4: Slave Device Interface

3. **Fault Interface**

Name	Width	Direction	Description
SEG_FAULT	1	R	Indicate occurrence of an error
SEG_REASON	2:0	W	Trigger of the segmentation fault
SEG_ADDR	31:0	R	Address that triggered the fault

Details

1. **Master Device**

   When the master device wants to use the bus, it must first set RREQ or WREQ to HIGH, corresponding to Read/Write requests. If both are set to high, WREQ will be ignored, and the bus responds to RREQ.

   If the bus accepts the request, it will set ACC to HIGH. The master should issue the corresponding request by setting appropriate values for ADDR and WDATA after the rising edge when ACC is HIGH. Any R/W action performed will take effect on the rising edge of the next clock cycle. On the cycle of the last R/W, the master must set RREQ/WREQ to LOW. The R/W action will take effect on the next rising edge, but any further action will

be ignored, and the bus will accept requests from other devices, or accept a different kind of request. Request once issued should not be droped without at least making one transfer.

During a read request, because a slave device may not be able to provide data instantly, BUSY may be set to HIGH to indicate current unavailability of the requested data. BUSY is set to LOW each time a valid data item appears on DATAR, and this data should be registered on the rising edge of the next clock cycle. During a write request, if the slave cannot accept any more data, BUSY is set to HIGH to indicate this. Any write by the master on subsequent rising edges are ignored by the slave, until the edge BUSY has been reset to LOW.

Every rising edge when ACC is high corresponds to a valid data R/W request. However, if the master do not want to issue a request on a cycle, it must set HOLD to high. No request will be registered on the rising edge during which HOLD is HIGH.

The master device must access one and only one slave device during each request. A segmentation fault is trigger if no slave device responds to the memory address, or if the slave device that responds to the memory address changed. This is represented by having SEGFLT set to HIGH. The debug register SEGADDR stores the address that triggered the previous SEGFLT.

2. **Slave Device**

   A slave device must listen to the bus at all times, and perform the requested R/W action.

   When SR is set to HIGH, a read request is issued on the rising edge of each clock cycle. If the address corresponds to the address of the slave, the slave must respond by providing the required data. If the slave device cannot provide this data in the clock cycle after the rising edge, it must set SBUSY to HIGH until the cycle valid data appears on SDATAR. The data must hold until the rising edge of the next clock cycle, and must preserve the order of the requests. On the other hand, if he address do not correspond to the address of the slave, the slave must set all bits of SDATAR, SBUSY and SACK to LOW.

   When SW is set to HIGH, a write request is issued on the rising edge of each clock cycle. The slave must perform this write. It is the slave's responsibility to guarantee that this write must take effect immediately and any read issued on the next rising edge shall reflect this change. If the slave cannot process any more writes, it should set SBUSY to HIGH to indicate this fact, after which it is free to ignore any request issued on the bus. The bus controller guarantees that SR and SW are not both set to HIGH on the same clock cycle. Furthermore, the controller guarantees that a write request will only be issued on the clock cycle after the slave has provided data for the most recent read request.

   The slave must set SACK to HIGH for one clock cycle for any request it processes. Failure to do so may result in segmentation fault (SEGFLT) on the master device. If the request is a read, the slave must also set SIDLE to LOW until the cycle the obtained data appears.

3. **Error Handling**

   An error will occur during an abnormal access. SEG_FAULT is set to HIGH, SEG_ADDR to the R/W address that triggered the fault, and SEG_REASON is set to one of the following

Value	Reason
0	No fault
1	No slave device responded to request
2	Slave device drop response during request
3	Master timing error
4	Time out

Timing Diagram

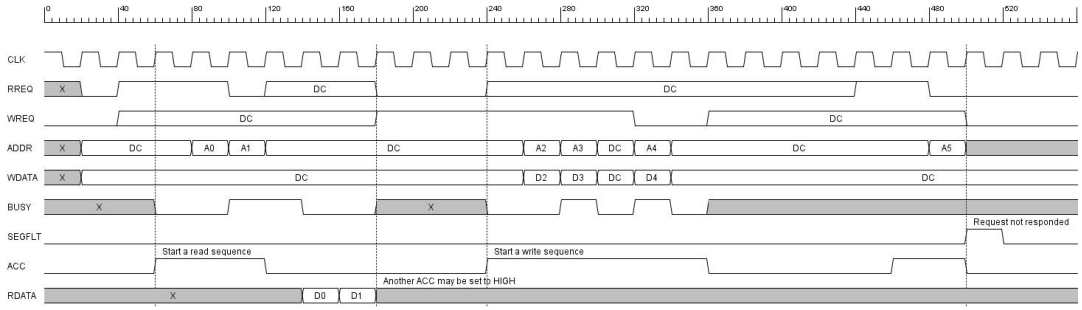


Figure 3.5: Master Device Timing Diagram

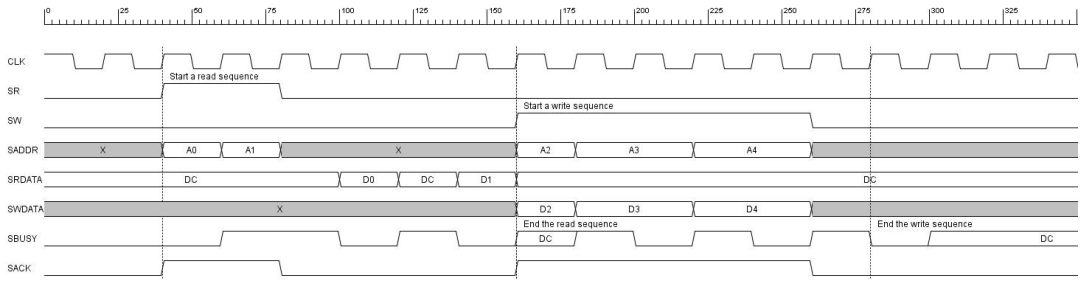


Figure 3.6: Slave Device Timing Diagram

3.4.3 Internal Memory

Introduction

These are simple RAM memory devices that are implemented by the M9K blocks within Altera FPGAs. Each device contains 32kB of memory in a relative address range of 0x0000-0x7FFF. Access is 4-byte aligned, as the last two bits are discarded.

The device can have a user specified base address on the bus by setting the parameter BASE_ADDR. By default the base address is zero.

Interface

This device implements standard bus interface as shown in table 3.4

3.5 Cache

3.5.1 ICache

Introduction

The ICache is an unassociative cache with 512B of memory capacity. The cache is organized as follows:

- The cache contains 16 sets
- Each set contains 1 cache line
- Each cache line contains 8 words
- Each word is four bytes

This organization leads to the following address partitioning scheme

Offset	31..9	8..5	4..2	1..0
Purpose	Tag Address	Set Address	Word Address	Word Offset

Interface

1. Service Interface

Name	Width	Direction	Description
ADDR	31..0	W	Address to R/W
RDATA	31..0	R	Data read from cache
RREQ	1	W	Issue a read request
MISS	1	R	The requested address not in the cache

2. Data Fetch Interface

   This device can be directly connected with L2Cache to R/W data.

Details

To read from an address set RREQ to HIGH and ADDR to the corresponding address, if data item is in L1 cache, RDATA will be set to the fetched data asynchronously with low latency, otherwise, MISS is set to HIGH asynchronously. Under this situation, a fetch from L2 cache will be performed, and may take tens of cycles. When the data has been retrieved, MISS will be set to LOW after a rising edge, and RDATA will produce the correct data item.

This device automatically maintains consistency with DCache. This is achieved by implementing the following strategies

1. If the requested address is also in DCache, then fetch the data from DCache instead
2. If a cache line is both in ICache and DCache, while DCache is dirty and initiates a write back. The cache line in ICache is also invalidated

Timing Diagram

This is exactly the same as Figure 3.7, removing the unused signals

3.5.2 DCache

Introduction

The DCache is an unassociative cache with 512B of memory capacity. The cache is organized as follows:

- The cache contains 16 sets
- Each set contains 1 cache line
- Each cache line contains 8 words
- Each word is four bytes

This organization leads to the following address partitioning scheme

Offset	31..9	8..5	4..2	1..0
Purpose	Tag Address	Set Address	Word Address	Word Offset

Interface

1. Service Interface

Name	Width	Direction	Description
ADDR	31..0	W	Address to R/W
RDATA	31..0	R	Data read from cache
WDATA	31..0	W	Date written to cache
WMASK	3..0	W	Write mask
RREQ	1	W	Issue a read request
WREQ	1	W	Issue a write request
MISS	1	R	The requested address not in the cache

2. Data Fetch Interface

   This device can be directly connected with L2Cache to R/W data.

Details

To read from an address set RREQ to HIGH and ADDR to the corresponding address, if data item is in L1 cache, RDATA will be set to the fetched data asynchronously with low latency, otherwise, MISS is set to HIGH asynchronously. Under this situation, a fetch from L2 cache will be performed, and may take tens of cycles. When the data has been retrieved, MISS will be set to LOW after a rising edge, and RDATA will produce the correct data item.

To write to an address, set WREQ to HIGH, ADDR to the corresponding address, and WDATA to written data. If data item is in L1 cache, MISS is set to LOW asynchronously, and the write will register on the next rising edge. Upon a cache miss, MISS is set to HIGH asynchronously.

All subsequent operations to device will be ignored until the cycle MISS is set to LOW. MISS will stay HIGH for a minimum of 8 clock cycles. To write only a part of a word, set WMASK to HIGH for the byte we would like to write. Bytes corresponding to bits where WMASK is LOW will stay unchanged

On either case, if MISS is set to HIGH, the desired operation has not been performed, and RREQ/WREQ, ADDR, WDATA must hold their values until MISS is set to LOW, and for one more cycle in which the desired operation is performed.

Timing Diagram

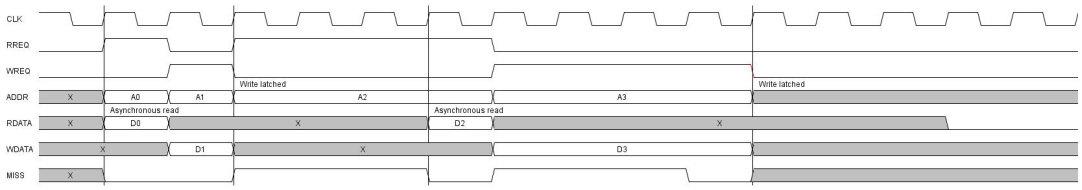


Figure 3.7: DCache Service Interface

3.5.3 Direct Loader

Introduction

This is a device with compatible service interface with DCache, but performs no caching. All read and write operations are directly launched on the bus

Interface

1. Service Interface

   This is identical to DCache service interface as in Section 1

2. Bus Interface

   This device implements standard bus master interface

3.5.4 L2Cache

Introduction

The L2Cache is a 16-way set associative cache with 32KB of memory capacity. The cache is organized as follows:

- The cache contains 32 sets
- Each set contains 16 cache lines
- Each cache line contains 16 words
- Each word is four bytes

This organization leads to the following address partitioning scheme

Offset	31..11	10..6	5..2	1..0
Purpose	Tag Address	Set Address	Word Address	Word Offset

When a read or write selects a location not present in the cache, the entire cache line will be fetched from memory. When a cache set is full, the entry that was least recently visited shall be eliminated by a write-back.

Interface

1. Service Interface

   These interfaces form the services provided by the cache device. Master devices can use these ports to performed the desired operations.

Name	Width	Direction	Description
RREQ	1	W	Issue a read request
WREQ	1	W	Issue a write request
ADDR	31..0	W	Address to R/W
BURST_SIZE	3..0	W	The number of words we would like to R/W
RDATA	31..0	R	Data read from cache
WDATA	31..0	W	Date written to cache
BUSY	1	R	The requested operation needs to wait

2. Bus Interface

   This device implements standard Master Device interface as in table 3.3

Details

To read a block of memory, set RREQ to HIGH, ADDR to the physical address to read, and BURST_SIZE to the number of words to fetch. All words should belong to the same cache line, otherwise a fault is generated. On the rising edge this request will be registered and BUSY is set to HIGH for a minimum of four cycles (even for a cache hit), the requested data appears on RDATA on the first cycle busy transits to LOW, and subsequent data appears on consecutive uninterrupted cycles. Note that whether a cache miss or not occurred is opaque to the user, but in general a cache miss results in a long busy period.

To write a block of memory, set WREQ to HIGH, ADDR to the physical address to write, and BURST_SIZE to the number of words to write. All words should belong to the same cache line, otherwise a fault is generated. On the rising edge this request will be registered, and BUSY is set to HIGH for a minimum of two cycles (even for a cache hit), the written data should be presented on WDATA after the rising edge for which BUSY is LOW, and subsequent data should be presented in consecutive uninterrupted cycles.

A new request cannot be issued if a previous request has not been finished. For a read, this means that the last requested data item has not appeared on RDATA, and for a write, the last written data has not been latched on WDATA

Timing Diagram

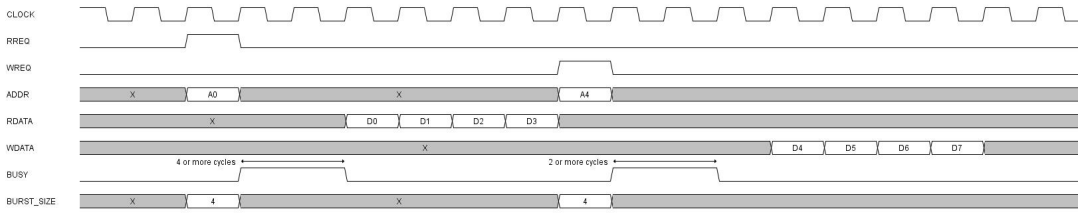


Figure 3.8: L2Cache Service Interface

3.5.5 L2 Adapter

Introduction

This device allows two devices (ICache and DCache) to access the L2Cache interface without interference. The two devices can perform R/W as if they are the unique user of the L2Cache interface.

Interface

1. Service Interface

   The interface below is offered in two copies so that both devices can use the L2 Cache interface as if they are the sole user

Name	Width	Direction	Description
RREQ	1	W	Issue a read request
WREQ	1	W	Issue a write request
ADDR	31..0	W	Address to R/W
BURST_SIZE	3..0	W	The number of words we would like to R/W
RDATA	31..0	R	Data read from cache
WDATA	31..0	W	Date written to cache
BUSY	1	R	The requested operation needs to wait

2. L2 Cache Interface

   This device implements standard L2 Cache interface and can be directly connected with L2Cache.

3.5.6 Cache Group

Introduction

A Cache Group is a sub-device of L2Cache. This is a fully associative cache block, with 16 cache lines. Each cache line can map to any tag address.

Interface

1. Service Interface

Name	Width	Direction	Description
ADDR	31..0	R	Address to R/W
RREQ	1	R	Request a burst read
WREQ	1	R	Request a burst write
BURST_SIZE	2:0	R	Request a burst size of up to 8 words
RDATA	31..0	W	Data read from the cache
WDATA	31..0	R	Data written to the cache
BUSY	1	W	The operation must wait

2. Bus Interface

   This device implements standard Master Device interface as in table 3.3. Furthermore, this device guarantees that when no request is active, all bus output pins are set to LOW. This allows connecting all devices by an OR.

Details

To read a block of memory, set RREQ to HIGH, ADDR to the physical address to read, and BURST_SIZE to the number of words to fetch. All words should belong to the same cache line, otherwise a fault is generated. On the rising edge this request will be registered and BUSY is set to HIGH for a minimum of four cycles (even for a cache hit), the requested data appears on RDATA on the first cycle busy transits to LOW, and subsequent data appears on consecutive uninterrupted cycles. Note that whether a cache miss or not occurred is opaque to the user, but in general a cache miss results in a long busy period.

To write a block of memory, set WREQ to HIGH, ADDR to the physical address to write, and BURST_SIZE to the number of words to write. All words should belong to the same cache line, otherwise a fault is generated. On the rising edge this request will be registered, and BUSY is set to HIGH for a minimum of two cycles (even for a cache hit), the written data should be presented on WDATA after the rising edge for which BUSY is LOW, and subsequent data should be presented in consecutive uninterrupted cycles.

A new request cannot be issued if a previous request has not been finished. For a read, this means that the last requested data item has not appeared on RDATA, and for a write, the last written data has not been latched on WDATA

Timing Diagram

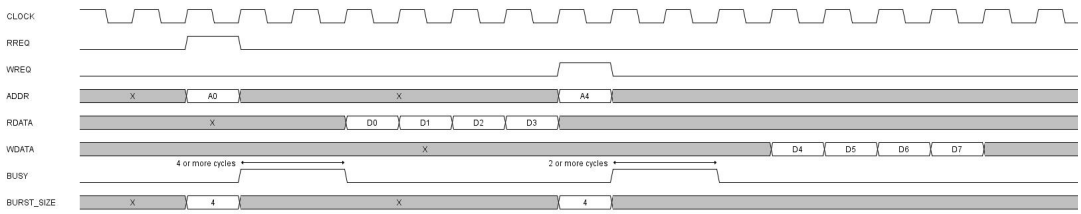


Figure 3.9: Cache Group Service Interface

3.5.7 Load Manager

Introduction

This module is a component of L2Cache that performs read and write of entire cache lines between L2Cache memory and bus. Each cache line is assumed to contain 8 4-byte words, so every read and write request results in a burst R/W of 8 consecutive words.

Interface

1. Service Interface

Name	Width	Direction	Description
WTRIG	1	R	Write the content of cache line into memory
RTRIG	1	R	Read the content of cache line from memory
PADDR	31..0	R	Physical address to R/W
LADDR	7..0	R	Cache line address to R/W
FINISH	1	W	The requested operation is complete
FAULT	1	W	A fault occurred

2. Cache Access Interface

   These pins access the cache memory in the cache group

Name	Width	Direction	Description
WREQ	1	W	Issue a write request
RADDR	7..0	W	Address to read
WADDR	7..0	W	Address to write
RDATA	31..0	R	Data read from cache
WDATA	31..0	W	Data written to cache

3. Bus Interface

   This device implements standard Master Device interface as in table 3.3

Details

To initiate a write back/read request, set WTRIG/RTRIG to HIGH, and PADDR and LADDR to their correct values. The request is latched on the next rising edge. After WTRIG/RTRIG is issued, read/write control of cache memory must be given to LoadManager starting from the cycle WTRIG/RTRIG is registered. For a write request, the LoadManager will read out contents from the cache memory, and access the bus to write to their correct memory locations. After the last data is acknowledged by the bus, FINISH is set to HIGH for one clock cycle, new requests can be issued starting the next cycle. For a read request, the LoadManager will read contents from bus and write them to the cache memory. On the cycle the last data item is latched to WDATA, FINISH is set to HIGH for one clock cycle, and new requests can be issued starting the next cycle.

This device assumes that cache memory has a read with two cycle latency, that is, the data requested is produced upon the second rising edges after the cycle the request is issued.

Timing Diagram

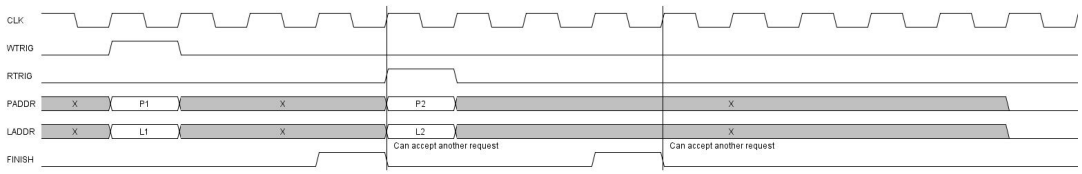


Figure 3.10: Load Manager Service Interface

3.5.8 Cache LUT

Introduction

This is a sub-device of L2Cache that manages the cache mapping table

Interface

Name	Width	Direction	Description
EN	1	R	Perform a query on TAG_ADDR
TAG_ADDR	18..0	R	The tag of the address location
GROUP_ADDR	3..0	W	The cache line corresponding to the tag
CACHE_HIT	1	W	The tag location is in the cache
REPLACED_LINE	1	W	The cache line that should be replaced

Details

When the cache group would like to access a memory location, it should set EN to HIGH and TAG_ADDR to the correct tag value. If the cache line is in the cache memory, the address of the cache line is returned by GROUP_ADDR, and CACHE_HIT is set to HIGH on the next clock cycle. Otherwise, CACHE_HIT is set to LOW on the next clock cycle, and GROUP_ADDR is set to the line that must be replaced after two cycles (See timing diagram). For either case, a new query should only be issued after the cycle required data appears on GROUP_ADDR.

Timing Diagram

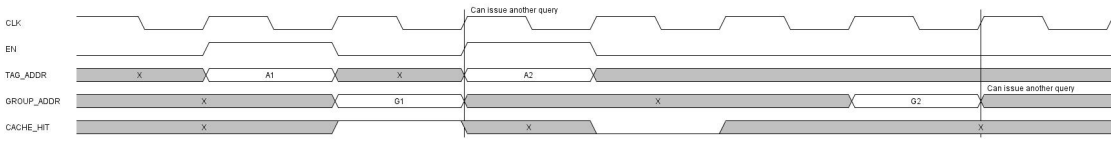


Figure 3.11: Device Timing Diagram

3.5.9 Time Stamp Manager

Introduction

This is a module that records the usage history of the cache lines, and outputs the least recently used cache line by a fast comparison tree algorithm

Interface

Name	Width	Direction	Description
EN	1	W	A cache line is accessed
ACCESS	3..0	W	Index of accessed cache line
OLDEST	3..0	R	Least recently accessed line

Details

When EN is set to HIGH, the cache line indexed by ACCESS will have its time stamp cleared, while other cache lines will have their time stamp incremented

OLDEST outputs the index with greatest time stamp value, regardless of the EN. However, after EN is set to HIGH, and time stamp values are altered, it takes three clock cycles before this change is reflected.

Timing Diagram

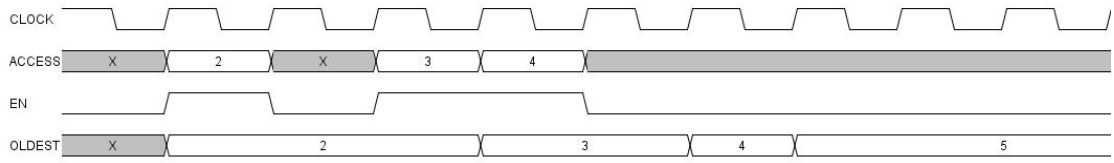


Figure 3.12: Device Timing Diagram

3.6 外设

3.6.1 SRAM 控制器

SRAM 控制器仅用于 Basic 版设计中，其设计目的是解决指令总线和数据总线需要同时访问内存造成的冲突。解决方法是使将内存访问的频率提高，在一个 CPU 周期下完成两次访存操作，两次访存对总线透明。控制器实现位于 **HDL/src/sram/**文件夹中。

SRAM 控制器需要一个频率为总线时钟 4 倍，且相位对齐的时钟，用于驱动控制器中的状态机，产生内存控制信号。我们把该时钟称作 RAM 时钟。在 RAM 时钟与总线时钟对齐的上升沿后，总线控制信号与地址锁存到 SRAM 控制器上。在 RAM 时钟下一个上升沿时，指令总线的地址和控制信号锁存到 SRAM 芯片上。在 RAM 时钟下一个上升沿时，指令从 RAM 芯片上锁存至控制器中，同时数据地址和控制信号锁存到 SRAM 芯片上。在 RAM 时钟下一个上升沿时，数据从 SRAM 芯片上锁存至控制器中。之后紧接着又是总线时钟的下一个上升沿，重复该过程。

由于 Thinpad 实验平台上没有将 SRAM 的字节使能信号引入 FPGA，导致一次写入必须是 32 位的，无法直接实现半字、字节的写入。对此，我们设计了一个转换模块 **bytes_conv**，当其发现是非全字写操作时，产生一个总线周期的暂停请求，使 CPU 暂停一个周期。在这个周期，模块自行产生一个读请求给 SRAM 控制器，读取要写入部分所在的全字，并将写修改应用到读取的全字上，在下一周期即可全字写入 SRAM。

3.6.2 SSRAM 控制器

NaiveMIPS++ 只能在 DE2i 平台上运行，而 DE2i 上使用的是 SSRAM，因此必须实现一个 SSRAM 控制器，将 SSRAM 连接到总线上，代码位于 **HDL/src/sram/ssram_ctl.v** 文件中。

SSRAM 可以近似等效为 SRAM 的地址和数据信号上各增加一级触发器，这样一来所有信号均与时钟同步，解决组合逻辑容易出现的一些问题，提高了运行时钟频率。带来的麻烦是读数据增加了两个周期的延迟，即地址输入后，等待两个时钟周期才能得到数据，但由于读取过程是流水的，因此不影响带宽。

根据 3.4.2节对 slave 的时序说明，对于一次 brust 读操作，只要让 SBUSY 信号在开始时有效一个周期，之后失效，即可让地址和数据相差两个周期，这样 SSRAM 的输出就能直接送给总线了；而对于写操作，不需要延迟，每个周期写入一个字，SBUSY 保持为失效状态即可。

3.6.3 Flash 控制器

Flash 控制器用于连接 Thinpad 板上的 Flash 芯片与数据总线,代码位于 **HDL/src/flash/flash_top.v** 文件中。

Flash 芯片接口时序与 SRAM 类似，但不同之处在于它的总线接口 Cycle Time 较长，速度较慢。因此，CPU 每次访问 Flash 都需要等待一个至多个周期，且保持总线上的信号不变。我们在 Flash 控制器中加入一个状态机，用于产生从设备等待请求信号，CPU 收到该信号后会暂停等待。状态机对于每次 Flash 访问请求，都产生固定周期的等待请求信号，从而满足 Flash 时序要求。

等待周期数量可以由模块中的 **FLASH_BUS_CYCLE** 参数指定，该参数应根据总线频率和 Flash 芯片参数确定，保证等待的时间长于 Flash 的 Cycle Time 即可。

值得注意的一点是，Flash 芯片数据线只有 16 位，因此在读写 Flash 时，32 位数据的高 16 位实际上是无效的，这需要在软件中加以处理。

3.6.4 串口控制器

串口控制器用于产生 RS-232 串行通信信号时序，代码位于 **HDL/src/uart/**文件夹中。控制器共有 3 个模块，**uart_top** 与总线连接，实现各控制寄存器，**uart_tx** 产生串行信号，用于数据发送，**uart_rx** 接收串行信号。

串口控制器工作在 115200 波特率下，工作模式为 8 数据位，1 停止位，无校验位。控制器的接收模块采用 3 倍过采样方式，尽可能过滤信号中存在的毛刺，提高正确率。

寄存器说明

Table 3.5: Data Register offset: 0x8

31..8	7	6	5	4	3	2	1	0
	r/w							
Reserved	Data							

- **Data** 写入操作开始一个字节发送，读取操作读出收到的一个字节数据

Table 3.6: Status Register offset: 0xc

31..2	1	0
	r	r
Reserved	RXNE	TXE

- **RXNE** 接收非空标志位，表示当前有收到后尚未读出的数据
- **TXE** 发送为空标志位，表示当前发送寄存器为空，可以发送

uart_tx 信号说明

Name	Width	Direction	Description
clk_bus	1	In	总线时钟
clk_uart	1	In	串行信号时钟 11.0592M
rst_n	1	In	异步复位，低有效
txd	1	Out	串行信号输出
idle	1	Out	发送状态机空闲，与总线时钟同步
tx_request	1	In	发送起始请求，与总线时钟同步
data	7..0	In	待发送数据，与总线时钟同步

uart_rx 信号说明

Name	Width	Direction	Description
clk_bus	1	In	总线时钟
clk_uart	1	In	串行信号时钟 11.0592M
rst_n	1	In	异步复位，低有效
rxd_in	1	In	串行信号输入
clear	1	In	清除接收数据，准备下次接收，与总线时钟同步
data_available	1	Out	接收数据寄存器非空，与总线时钟同步
data	7..0	Out	收到的数据，与总线时钟同步

3.6.5 GPIO

GPIO 控制器提供一个总线至普通 I/O 口的接口，代码位于 **HDL/src/gpio/gpio_top.v** 文件。

控制器支持 64 个 I/O 口，分为两组，GPIOA、GPIOB。在硬件上两组 I/O 连接到 LED、数码管和拨码开关上。每个 I/O 都可以配置为输入或输出模式，输入模式下 CPU 可以通过控制器获得某个引脚的电平状态（如获得开关通断），输出模式下 CPU 可以通过控制器设定某个引脚的电平状态（如控制 LED 点亮）。

寄存器说明

Table 3.7: GPIOA Data Register offset: 0x0

31..0
r/w
Data

Table 3.8: GPIOA Direction Register offset: 0x4

31..0
w
Direction

Table 3.9: GPIOB Data Register offset: 0x8

31..0
r/w
Data

Table 3.10: GPIOB Direction Register offset: 0xc

31..0
w
Direction

- **Direction** I/O 方向配置，每位对应一个引脚，1 表示输出，0 表示输入
- **Data** I/O 数据寄存器，读操作得到引脚输入电平状态，写操作设定引脚输出电平

3.6.6 精确计时器

精确计时器用于提供一个可靠的，走时不依赖于 CPU 指令执行过程和主频的计时参考源。其代码位于 **HDL/src/ticker/ticker.v**。

计时器计数时钟由 50M 输入时钟经过 PLL 变频后得到，固定为 1KHz，与 CPU 主频无关。

寄存器说明

Table 3.11: Ticker Register offset: 0x0

31..0
r
Ticker

- **Ticker** 自复位起到读取该寄存器止经过的毫秒数

3.6.7 VGA 控制器

VGA 控制器用于驱动 VGA 接口，产生 800×600 @ 72Hz 的黑白视频信号。同时，本控制器具有可调的循环偏移输出功能，便于在控制台输出滚动时绘制文字。该模块的代码位于 **HDL/src/gpu/gpu.v**

寄存器说明

Table 3.12: GPU Register offset: i ($0 \leq i < 60000, i\&3 = 0$)

31..0
w
Pxl[8*i+31]...Pxl[8*i+0]

Table 3.13: GPU Register offset: 0x50000

31..0
w
Offset

- **Pxl** 像素寄存器，当某一位为 0 时，该位所代表的像素为黑色，反之为白色。
- **Offset** 同步偏移寄存器。当该寄存器不为 0 时，屏幕原点提前 $Offset \times 32$ 个像素被绘制。

3.7 SoC 顶层设计

3.7.1 Basic

Basic 版本的 SoC 顶层设计文件为 **HDL/src/soc_toplevel.v**，系统框图如下图所示：

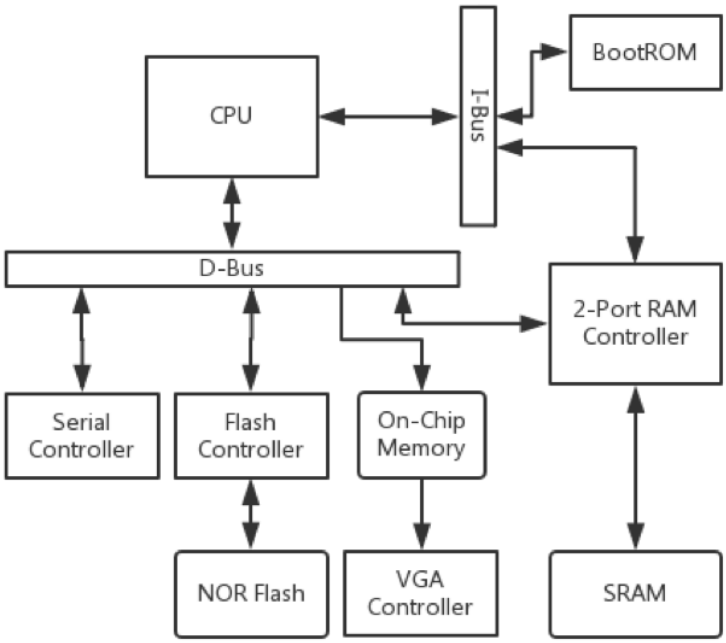


Figure 3.13: 系统框图（Basic 版本）

从中可以看出指令与数据总线分离，指令只能从 RAM 和 BootROM 中获取。其它外设只能通过数据总线访问。

3.7.2 NaiveMIPS++

增加 Cache 后的 NaiveMIPS++ 顶层文件是 **HDL/src/soc_toplevel_cache.v**。CPU 的指令和数据总线与 Cache 相连，L2 Cache 再通过 uma 总线与其它外设连接。外设总线只有一条，双端口的 SRAM 控制器不再被使用，所有外设都可以通过数据或指令总线访问。系统框图如 3.2所示。

3.8 uCore

NaiveMIPS 的 uCore 是在 cyh 版本操作系统（参见 2.2.2）上作了一些修改得到的。主要改进是使其符合硬件设计、优化性能，并增加一些额外功能。

3.8.1 开发环境

uCore 主要开发工具为 GCC 交叉编译工具链。我们使用的是自行编译的 GCC，版本号为 5.2.0，其配置选项如下：

```
../configure --target=mips-sde-elf --disable-nls --enable-languages=c
--disable-multilib --without-headers --disable-shared --without-newlib
--disable-libgomp -disable-libssp --disable-threads --disable-libgcc
```

此外编译 uCore 还需要 binutils 工具包，以提供汇编器、连接器等，使用的版本为 2.25.1，配置选项：

```
./configure --disable-werror --target=mips-sde-elf
```

3.8.2 编译选项

由于 NaiveMIPS 支持分支指令后的延迟槽和精确异常处理，为了提高程序运行性能，我们启用了编译时的指令重排（即去掉），以充分利用流水性能。

从运行的性能考虑，我们启动了所有 C 代码编译时的优化，优化等级为 O2。由于较为全面地实现了 MIPS32 指令集，在启用优化后产生的所有指令仍在已经实现的指令范围内。

最终对 uCore 顶层 Makefile 中的 CFLAGS 修改后变为：

```
CFLAGS :=  -fno-builtin -nostdlib  -nostdinc -mno-float -g  -EL -G0 -O2 -Wa,-O0
```

3.8.3 内存管理

cyh 版 uCore 在修改 TLB 时使用的是随机替换指令 TLBWR，考虑到随机实现较困难，同时也为了给软件更大的扩展余地（如更复杂的替换策略），CPU 中实现的是替换指定条目的 TLBWI 指令。由此我们对 uCore 作了相应的修改。在文件 **kern/include/thumips_tlb.h** 的 **tlb_refill** 函数末尾，将 **tlb_replace_random** 调用换成了 **write_one_tlb**。**write_one_tlb** 的 index 参数即为需要替换的 TLB 条目，目前选用随机替换策略，将 index 设置为 0 至 7 中一个随机的整数。

内存容量方面，由于我们在硬件上将 baseRAM 和 extRAM 合并起来使用，共有 8M Bytes 的物理内存。因此将 **kern/mm/memlayout.h** 中针对 MACH_FPGA 的 **KMEMSIZE** 宏定义修改为 `(8 << 20)`。

3.8.4 精确计时器驱动

为了便于用软件测定系统启动后的时间，避免通过 Tick 数测定时间带来相关偏差，我们在 FPGA 上通过 PLL 实现了一个精确的硬件定时器，并配以相应的内核驱动。为了方便用户态程序读取该定时器，我们接管了 `sys_gettime` 系统调用，从而使用户程序可以获得毫秒级的精确定时。

3.8.5 性能测试程序

为了方便对比不同 CPU 实现的运算等方面的性能差异，我们编制了 CPU 整数运算性能测定程序 `Mpack`。该程序可以测定单位时间内执行的整数运算次数。本程序反复计算随机给定的两个矩（方）阵的乘积，测定计算完成时间，利用给定矩阵规模推知运算次数，将该次数除以完成时间，从而达到测定单位时间内执行的整数运算次数的目的。

3.9 NaiveDebugger

3.9.1 On-Chip Module

片上调试模块集成在 CPU 中，一方面通过硬件信号监控和控制 CPU，另一方面通过某种专用的通信接口（如串口、JTAG）与上位机通信，其实现位于 **HDL/src/cpu/debugger/**文件夹中。片上模块包含两个模块，一个是与 CPU 直接相连的调试其状态机 **dbg_ctl**，另一个是于上位机 Agent 通信的协议接口 **dbg_uart**。

dbg_ctl 模块在访存监视指令，如果发现指令地址与预设的断点相同，着发出信号清除流水线，设置 PC 为这条指令，同时暂停流水线，等待上位机指令。此时上位机中通过查询状态发现已经断下，可以发送查看数据、查看寄存器等各种指令。当需要恢复执行时，撤销断点，恢复流水线，则 CPU 从被断下的指令开始继续执行。

在本项目中上位机 Agent 通信选择了一个独立的串口，用于传输调试器的信息，与串口外设无关。串口上的传输协议非常简单，上位机发送 1 字节命令，和 4 字节参数（可选，由命令决定）。随后，下位机发送 4 字节结果作为应答。命令宏定义在 **dbg_ctl** 模块中，描述如下：

Name	Description	Command Code	Argument
CMD_STOP	CPU 暂停	0x1	
CMD_CONT	CPU 继续	0x2	
CMD_EN_BP	启用断点	0x3	
CMD_DIS_BP	禁用断点	0x4	
CMD_SET_BP	设置断点地址	0x85	断点地址
CMD_READ_REG	读通用寄存器	0x86	寄存器地址
CMD_READ_CP0	读 CP0 寄存器	0x87	CP0 地址
CMD_READ_HI	读 HI 寄存器	0x8	
CMD_READ_LO	读 LO 寄存器	0x9	
CMD_READ_PC	读 PC 寄存器	0xa	
CMD_RESET	复位 CPU	0xb	
CMD_READ_IMEM	读指令内存	0x8c	内存地址
CMD_STEP	一次单步运行	0x0d	
CMD_QUERY	查询状态	0x0e	

其中读内存指令只能在 CPU 停止状态下使用。

dbg_ctl 信号描述如下：

Name	Width	Direction	Description
clk	31..0	In	CPU 时钟
rst_n	31..0	In	异步复位，低有效
inst_pc_value	31..0	In	当前指令地址
inst_in_delayslot	31..0	In	当前指令位于延迟槽
main_reg_addr	31..0	Out	通用寄存器地址
main_reg_value	31..0	In	通用寄存器值输入
cp0_reg_addr	31..0	Out	CP0 寄存器地址
cp0_reg_value	31..0	In	CP0 寄存器值输入
hilo_reg_value	63..0	In	HI、LO 寄存器值输入
pc_reg_value	31..0	In	PC 寄存器值输入
pc_reset	31..0	Out	PC 复位
debug_stall	31..0	Out	暂停流水线
flush	31..0	Out	清空流水线，设置新的 PC
new_pc_value	31..0	Out	新的 PC 值
debugger_mem_read	1	Out	访存读使能
debugger_mem_addr	31..0	Out	访存地址
debugger_mem_data	31..0	In	访存数据输入
host_cmd	7..0	In	命令代码，与 dbg_uart 相连
host_param	31..0	In	命令参数，与 dbg_uart 相连
host_cmd_en	1	In	命令有效，与 dbg_uart 相连
host_result	31..0	Out	返回结果，与 dbg_uart 相连

3.9.2 Debugger Agent

调试协议代理程序工作在上位机上，一方面通过专用接口与片上的调试模块通信，另一方面通过 TCP 与 GDB 通信，其实现位于 **naive-debugger/gdbserver/gdb-server.c** 文件中。该文件修改至 ST-LINK 项目[2]中的 GDB Server 程序。

程序的主要流程是监听 GDB TCP 端口，等待 GDB 连接上后，进入主循环。在主循环中，对于收到的 GDB 指令，翻译成与片上调试器通信的协议，并将结果翻译回 GDB 要求的文本格式。与 GDB 的通信协议为 GDB Remote Protocol，其描述可以在手册[3]中找到。

3.10 NaiveBootloader

NaiveBootloader 是一个引导程序，固化在 BootROM 中。CPU 复位后，PC 寄存器将指向 BootROM 所在的地址空间，即 NaiveBootloader 的入口地址，因此 NaiveBootloader 是 CPU 启动后最先执行的程序。

NaiveBootloader 支持串口通信，也就是说可以在上位机发送命令给 NaiveBootloader。利用该程序，我们除了可以引导 uCore 系统外，还可以做大量的调试操作，例如从串口把目标程序加载进 RAM 并执行等，避免了每次把目标程序写入 Flash 中。

[2]https://github.com/texane/stlink
[3]https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

3.10.1 Bootloader 固件

NaiveBootloader 的固件部分（即固化在 BootROM 中的程序）用汇编语言编写，文件位于 **asm_program/boot.s**。

之所以使用汇编编写，是因为这样可以完全控制程序行为，比如使其不使用 RAM。这样的好处是，进行初期硬件调试时，可以用它来测试内存控制器是否工作正常（将一段数据通过串口写入内存，在通过串口读出，在电脑上比较）。技术内存控制器不正常工作，Bootloader 固件也能正确工作。

固件启动时，会首先初始化 GPIO，通过 GPIO 读取拨码开关 0 的状态，如果开关为 1 时就调用 uCore 的 bootloader，从 Flash 中加载 uCore 并运行。而当开关为 0 时就进入调试模式，此时可以在电脑用上位机程序控制 bootloader 进行各种操作。

bootloader 固件可在 asm_program 目录用 `make` 命令编译，编译完后会产生 boot.mif 和 boot.coe 文件，分别对应 Altera 和 Xilinx FPGA 的内部存储器初始化文件。

3.10.2 上位机程序

上位机程序用 Python 编写，位于 **HDL/utility/serial_load.py**。直接运行将输出使用说明：

```
Usage: ./serial_load.py <options>
  NaiveBootloader host program.
  Options are:

    -h --help          Display this information
    -s <device>
    --serial <device>  Specify serial port
    -b <baud>
    --baud <baud>      Specify serial baudrate
    -t <test>
    --test <test>      Run a test
        uart           UART loopback test
        ram            RAM read/write test
        flash          Flash access test
    -l <elf_file>      Load ELF to RAM and run
    --bin <address>    Load binary file, specify load address
    -g <address>
    --run <address>    Jump to <address> and run
    -p <bin_file>      Program file to Flash
    -r <bin_file>      Read from Flash to file
    --size <size>      Read only <size> bytes
    --term             Start a terminal after loading
```

这里列举几种常见的用法：

串口读写测试 `serial_load.py -s <dev> --test uart`

内存读写测试 `serial_load.py -s <dev> --test ram`

Flash 访问测试 `serial_load.py -s <dev> --test flash`

Flash 写入 `serial_load.py -s <dev> -p <file>`

加载 **ELF** 文件至内存并运行 `serial_load.py -s <dev> -l <file> --term`

3.11 Decaf

3.11.1 Runtime Library

结构

本运行时库由“decafIo”和“decafCall”两部分组成，其中，“decafCall”用汇编书写，用以实现Decaf 程序运行时库函数，以供 Decaf 程序调用；“decafIo”用 C 语言书写，用以实现相关库函数的具体逻辑。本运行时库编译后，将两部分合二为一，生成 `libdecaf.a`，链接 Decaf 程序时，只需添加 `-ldecaf` 即可将本运行库链接进来。

本运行库依赖于 ucore 标准用户态运行时库“libuser”，为保证链接成功，需添加 `-luser`。

ABI

依照标准 [7] 的规定，在函数调用过程中，调用函数（caller）和被调用函数（callee）应至少满足如下调用标准（仅摘录部分相关要点，调用过程局限于传递（传入或传出）32 位整数（或相当类型，如指针等）的情形）：

1. 调用函数应当将返回地址保存在 `ra` 寄存器，被调用函数在执行完毕后应跳转回该地址。
2. 被调用函数在跳转回返回地址时，应当保证 `sp`、`fp`、`s0-s7` 等寄存器与进入函数之前一致。
3. 被调用函数应当将返回值保存于寄存器 `v0`，调用函数应当在这一寄存器读取返回值。
4. 调用函数应当在栈帧（stack frame）中保留大小相当于被调用函数参数大小的空间；如果被调用函数的参数少于四个，则应至少保留四个相应的参数占用的空间（16 字节）；而被调用函数可以向上述空间中写入数据。
5. 调用函数应当在寄存器 `a0-a3` 中保存第 0 ～ 3 个参数的值，调用函数应当在内存地址 `memory[%sp + 4*i]` 的位置处保存第 i 个参数的值（$i \geq 4$）；而被调用函数应该在上述位置获得传入参数。

而通过阅读给定的编译器输出的汇编代码，我们发现，此编译器产生的函数调用代码，违背了上述第四条、第五条，取而代之的是：

4. 调用函数在栈帧中保留大小相当于被调用函数参数个数加一个单位的大小的空间；被调用函数可以向上述空间内写入数据。
5. 调用函数在内存地址 `memory[%sp + 4 * (i+1)]` 中保存第 i 个参数的值；被调用函数在上述位置获得传入参数。

对比上述约定，我们不难发现，如果 Decaf 程序直接调用“decafIo”中的相应 C 语言编写的函数，则会出现如下问题：

- 被调用的 C 语言函数无法从寄存器 `a0-a3` 中读取前四个参数
- 当被调用的 C 语言函数的参数个数少于四个时（几乎全部函数），被调用的函数有可能写入内存位置 `mem[%sp + 4*i]`$(0 \leq i < 4)$，从而破坏调用者的栈帧结构。

因此，我们用汇编语言编写了“decafCall”，作为 Decaf 程序和“decafIo”库的兼容层。“decafCall”实现了下述功能：

1. 重新分配栈空间；
2. 从原栈帧读入参数到寄存器中；
3. 保存返回地址并加载新的返回地址；

4. 执行标准 C 函数调用，调用相应的“decafIo”中的库函数;

5. 恢复原栈帧、返回地址;

6. 返回 Decaf 程序。

库函数实现

函数实现位于 **libdecaf/decafIo.c** 文件中，下面给出各个库函数的设计方法。

__Alloc
分配内存。由于 uCore 不支持用户态动态内存分配，因此声明了一个（进程中）全局的静态内存池，每次分配内存请求都从该内存池中取出指定长度的空间。

__StringEqual
比较两个字符串。直接使用 uCore 提供的 strcmp 函数比较字符串，strcmp 返回 0 则说明相等，返回 true，否则返回 false。

__ReadLine
读取一行字符串。利用 read 函数逐字节从 stdin 读取字符保存到缓冲区，直到读取到换行符时停止，将缓冲区返回。由于 uCore 的控制台无回显，每读取到一个字符还须将其打印出来，使得用户可见回显。

__ReadInteger
读取一个整数。利用 read 函数逐字节从 stdin 读取字符，首先跳过负号、数字以外的字符，然后判断是否为负号，之后逐个读入数字，并转为整数。此外，还须消耗掉整数之后的空白字符直到换行符，以免对可能紧跟的 _ReadLine 调用造成影响。由于 uCore 的控制台无回显，每读取到一个字符还须将其打印出来，使得用户可见回显。

__PrintInt
打印一个整数。直接用 printf 函数实现。

__PrintString
打印一个字符串。直接用 printf 函数实现。

__PrintBool
打印一个布尔值。根据值转为”true” 或”false”，用 printf 函数打印。

__Halt
结束程序。调用 uCore 的 exit 函数，结束当前进程。

3.11.2 编译器修改

在开发中，我们发现 Decaf 编译器的两点不足，并进行了修改。

Decaf 在生成汇编代码时，没有对函数入口符号标明类型，导致生成的目标文件中缺少函数符号的类型信息，不利于调试时产生反汇编代码。因此在 **backend/Mips.java** 文件 `emitProlog` 函数开头插入如下代码来产生符号类型说明。

```
emit(null, ".type " + entryLabel.name + ", @function", null);
```

由于 Decaf 的 main 函数无返回值，因此其退出时返回值寄存器中的值是随机的。而非 0 的返回值可能被误当作程序未正确退出。因此我们修改了 **translate/Translater.java** 中 **createFuncty** 函数，将主函数在汇编代码中符号名称改为了 ___decaf_main，从而可在运行时库中将 ___decaf_main 封装成符合 C 规范（int 返回值）的 main 函数，正确返回 0，避免了不可预测的返回值。

3.11.3 编译流程

整体编译流程如下图所示:

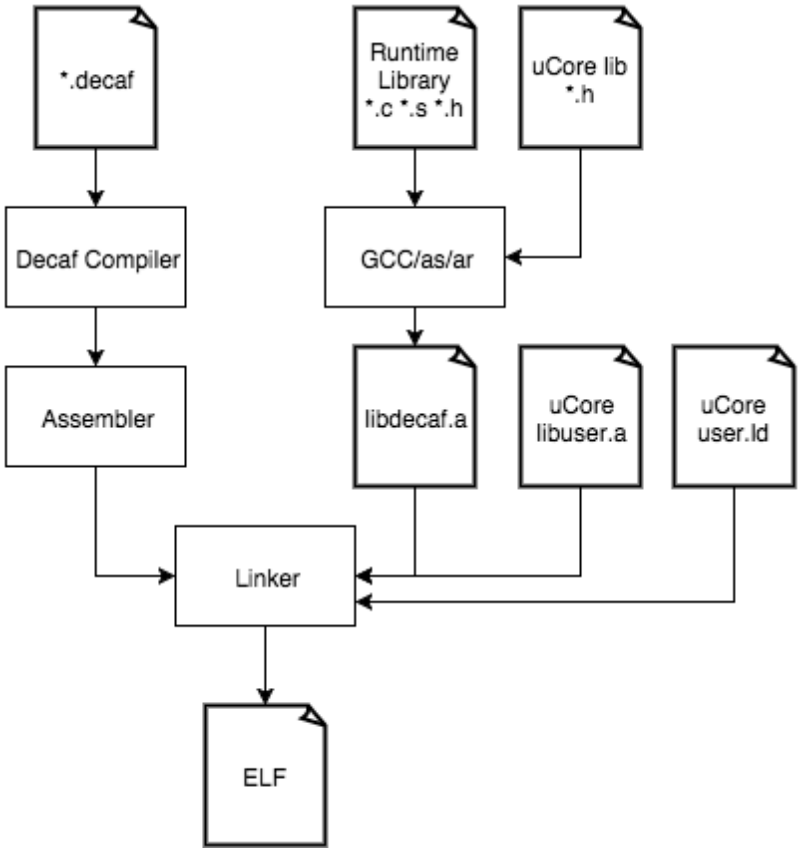


Figure 3.14: Compilation Procedure

编译流程分为两部分，一部分是运行时库的编译，一部分是 Decaf 应用程序的编译过程。由于 uCore 不支持动态链接库，整个过程都只有静态链接。

其中 libdecaf.a 为编译后的运行时静态库，只需在 Decaf 安装时编译一次，之后在编译 Decaf 应用程序是不必重新编译。运行时库由于用汇编和 C 编写，直接使用 GCC 工具链编译。

Decaf 应用程序的编译过程，要经历 Decaf 编译器和汇编器两个阶段后得到目标文件，并在 uCore 的链接脚本控制下，与运行时库、uCore 用户程序库链接，得到最终的可执行文件。

Chapter 4

Appendix

4.1 NaiveMIPS 指令集

处理器支持的全部指令如下：

Mnemonic	Instruction
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
SB	Store Byte
SH	Store Halfword
SW	Store Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ANDI	And Immediate
LUI	Load Upper Immediate
ORI	Or Immediate
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
XORI	Exclusive Or Immediate
ADD	Add Word
ADDU	Add Unsigned Word
AND	And

NOR	Nor
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word
XOR	Exclusive Or
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
NOR	Nor
OR	Or
XOR	Exclusive Or
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word
MADDU	Multiply and Add Word Unsigned
MFHI	Move From HI
MFLO	Move From LO
MSUB	Multiply and Subtract Word
MSUBU	Multiply and Subtract Word Unsigned
MTHI	Move To HI
MTLO	Move To LO
MUL	Multiply Word to Register
MULT	Multiply Word
MULTU	Multiply Unsigned Word
J	Jump
JAL	Jump and Link

JALR	Jump and Link Register
JR	Jump Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero
BLTZAL	Branch on Less Than Zero and Link
BEQL	Branch on Equal Likely
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely
BGEZL	Branch on Greater Than or Equal to Zero Likely
BGTZL	Branch on Greater Than Zero Likely
BLEZL	Branch on Less Than or Equal to Zero Likely
BLTZALL	Branch on Less Than Zero and Link Likely
BLTZL	Branch on Less Than Zero Likely
BNEL	Branch on Not Equal Likely
MOVF	Move Conditional on Floating Point False
MOVN	Move Conditional on Not Zero
MOVT	Move Conditional on Floating Point True
MOVZ	Move Conditional on Zero
SYSCALL	System Call
ERET	Return from Exception
MTC0	Move To Coprocessor 0
MFC0	Move From Coprocessor 0
CACHE	Perform the cache operation
TLBWI	Write a TLB entry indexed by the Index register

4.2 CP0

Register 0 *Index* TLB 表入口索引

Fieleds	Bits	Description	R/W	Reset State
Reserved	31..4	TLB index. Software writes this field to provide the index to the TLB entry referenced by the TLBR and TLBWI instructions.	R/W	Undefined
Index	3..0			

Register 2 *EntryLo0* 偶数虚拟页入口的低位地址

Register 3 *EntryLo1* 奇数虚拟页入口的低位地址

Fieleds	Bits	Description	R/W	Reset State
Reserved	31..26	Page Frame Number. Corresponds to bits[31..12] of the physical address.	R/W	Undefined
PFN	25..6			
Reserved	5..2	Valid bit, indicating that the TLB entry, and thus the virtual page mapping are valid. If this bit is a one, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a TLB Invalid exception.	R/W	Undefined
V	1			
Reserved	0			

Register 8 *BadVAddr* 记录异常的虚拟地址

Fieleds	Bits	Description	R/W	Reset State
BadVAddr	31..0	Bad virtual address	R	Undefined

Register 9 *Count* 系统定时器计数值

Fieleds	Bits	Description	R/W	Reset State
Count	31..0	Interval counter	R/W	Undefined

Register 10 *EntryHi* TLB 入口高位地址

Fieleds	Bits	Description	R/W	Reset State
VPN2	31..13	VA[31..13] of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write.	R/W	Undefined
Reserved	12..0			

Register 11 *Compare* 系统定时器比较匹配值

Fieleds	Bits	Description	R/W	Reset State
Compare	31..0	Interval count compare value	R/W	Undefined

Register 12 *Status* 中断控制、系统状态、工作模式等配置

Fieleds	Bits	Description	R/W	Reset State
Reserved	31..5			
UM	4	If Supervisor Mode is not implemented, this bit denotes the base operating mode of the processor. The encoding of this bit is: 0 Base mode is Kernel Mode; 1 Base mode is User Mode.	R/W	Undefined
R0	3	If Supervisor Mode is not implemented, this bit is reserved. This bit must be ignored on write and read as zero.	R	0
Reserved	2			
EXL	1	Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception are taken.	R/W	Undefined
IE	0	Interrupt Enable: Acts as the master enable for software and hardware interrupts	R/W	Undefined

Register 13 ***Cause*** 记录异常原因

Fieleds	Bits	Description	R/W	Reset State
Reserved	31..16			
IP[7:2]	15..10	Indicates an external interrupt is pending: 15 (Hardware interrupt 5, timer or performance counter interrupt), 14 (Hardware interrupt 4), 13 (Hardware interrupt 3), 12 (Hardware interrupt 2), 11 (Hardware interrupt 1), 10 (Hardware interrupt 0)	R	Undefined
IP[1:0]	9..8	Controls the request for software interrupts: 9 (Request software interrupt 1), 8 (Request software interrupt 0)	R/W	Undefined
ExcCode	6..2	Exception code	R	Undefined
Reserved	1..0			

Register 14 ***EPC*** 异常恢复后执行代码所在的地址

Fieleds	Bits	Description	R/W	Reset State
EPC	31..0	Exception Program Counter	R/W	Undefined

Register 15 *EBase* 异常处理程序入口

Fieleds	Bits	Description	R/W	Reset State
1	31	This bit is ignored on write and returns one on read.	R	1
0	30	This bit is ignored on write and returns zero on read.	R	0
Exception Base	29..12	In conjunction with bits 31..30, this field specifies the base address of the exception vectors.	R/W	0

References

[1] MIPS32 Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture

[2] MIPS32 Architecture For Programmers Volume II: The MIPS32™ Instruction Set

[3] MIPS32 Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture

[4] 计算机系统实验准备

[5] 计算机系统综合设计与实现——CP0 中断 MMU

[6] 基于简化版 MIPS32 指令集 CPU 的 ucore 教学操作系统移植

[7] SYSTEM V APPLICATION BINARY INTERFACE MIPS RISC Processor Supplement 3rd Edition

NaiveMIPS Test Document

January 18, 2016

Table 1: Resvision History

时间	作者	内容更改
2015.11.3	赵晟佳	初始版本
2015.12.15	张宇翔	增加更多测试程序
2016.1.1	张宇翔	添加性能测试结果

Contents

1 CPU

本章节描述了 NaiveMIPS 项目 CPU 部分的测试方法。

1.1 CPU 自动化测试

为了在开发中能够及时发现问题，我们基于 SystemVerilog 语言开发了 CPU 自动仿真测试程序。其核心思想是自动把某段 testcase 代码加载进内存，然后使调至代码入口运行。testcase 会针对性地运行一些待测的指令，并将指令的结果尽量反映为寄存器值的变化。而 Testbench 将监测寄存器变化，一旦发现某个寄存器的值改变，就与 testcase 程序对应的答案文件相比较，如果发现不同，则说明 CPU 执行有误，将报告错误并暂停仿真。

这种测试方法将 testcase 本身与测试工具分离，使得 testcase 的开发变得容易。并且测试过程是自动化的，如果没有发现错误，则完全不需要人工干预，使得测试过程变得高效。

1.1.1 Testbench

testbench 文件位于 **HDL/testbench/basic_test/test_cpu.sv**，Modelsim 仿真工程位于 **HDL/testbench/model**
打开工程后，启动 test_cpu 程序即可开始测试。tesetcase 将从 **HDL/testbench/testcase** 目录加载。

仿真过程中会输出每次寄存器改变的情况，全部测试完成后将输出成功提示。

```
# PC=80001040
# $27=00000000
# correct
# PC=80001044
# PC=80001048
# PC=8000104c
# Pseudo Exception: ERET
# PC=80001050
# PC=00000704
# PC=00000708
# PC=0000070c
# PC=80000208
# PC=8000020c
# $16=dead0000
# correct
# PC=80000210
# PC=8000020c
# PC=80000210
# $8=0000face
# correct
# Unit test succeeded!
# Break in Module test_cpu at :
```

Figure 1: Testbench

1.1.2 Testcase

testcase 由一组同名的 “.mem” 文件和 “.ans” 文件组成。“.mem” 文件是指令的十六进制代码，每行一条指令，该文件可在 **HDL/testbench/testcase/**目录由 make 命令从 “.s” 结尾的汇编代码编译产生。

“.ans” 是 tesetcase 的答案文件，通常由手工编写。其格式是每行描述一次寄存器值改变，格式为 “<reg> <value> “。支持的 <reg> 有通用寄存器 $0 $31，以及 hi、lo。<value> 为 16 进制表示的数字。举例来说，描述 $3 寄存器变为 0xdeadbeef 可以写作 “$3 deadbeef”。所有的寄存器改变必须顺序列出，中间不得省略。对于除法指令一次性改变 hi、lo 寄存器的情况，应该先列出 hi 的改变。

注意，Testbench 只能检测到寄存器变化后才能与答案比较。因此如果某条指令设置了寄存器，但刚好和原来值一样，将不能被 Testbench 检测到。在编写 testcase 时应当注意这一点。

1.2 基础指令测试

HDL/testbench/testcase/中对于基础指令的测试位于如下文件中：

- **inst_move.s** 条件移动类数据测试
- **inst_alu.s** 算数逻辑运算指令测试（不含除法）
- **inst_div.s** 除法指令测试
- **inst_branch.s** 分支指令测试
- **inst_div.s** 除法指令测试
- **inst_jump.s** 跳转指令测试
- **inst_mem.s** 加载存储指令测试
- **inst_endian.s** 加载存储字节序测试

1.3 异常与 MMU 测试

HDL/testbench/testcase/中对于异常与 MMU 的测试位于如下文件中：

- **overflow_exp.s** 算数溢出异常测试
- **mem_exp.s** 访存异常测试
- **timer_int.s** 系统定时器中断测试
- **tlb.s** TLB 测试
- **inst_syscalls** syscall 指令测试

1.4 TLB 独立测试

考虑到 TLB 复杂性，单独为 TLB 编写了 testbench，可以对各个信号做更加全面的检查。文件位于 **HDL/testbench/tlb_test/tlb_test.sv**。可以直接在 modelsim 中运行，不依赖外部文件。

2 Memory System & cache

This section describes testing performed on the memory system that includes L1 cache, L2 cache, bus controller and a few simple peripheral devices

2.1 Full System

1. Full Functionality Test: This test combines all components in the memory system. 8 Internal Memory devices are connected on the bus as slave device. 500k random reads and writes are performed on the ICache and DCache respectively. The read address are selected with high probability as the recently written addresses. If all content read match content written, the test passes. Otherwise the program stops where a mismatch occurs

    - Source: Main/MemorySystem_stimulus.v
    - Status: Pass by automated verification
    - This test requires 100M units of time to run

2.2 Bus Controller

1. Basic Protocol Test

   This test simulates several master and slave devices to test for correctness of the basic protocol the bus implements

   - : BusController/BusController_stimulus.v
   - Status: Pass by inspection of generated signal waveform

2. Memory Access Test

   This test connects eight Internal Memory devices on the bus, four masters are simulated to access the memory devices. The masters issue random reads and writes in burst of four words. The test will pass if the content returned to the masters are what they most recently wrote. If the test fails, it will stop where the incorrect read occurred. The test will also stop in failure if a segmentation fault occurred.

   - Source: BusController/BusController_mem_stimulus.v
   - Status: Pass by automated verification
   - Requires 100K units of time to finish

3. L2 Cache Test(See Section 2.7)

2.3 Internal Memory

1. Basic Function Test

   This test simulates bus requests on the memory device

   - Source: InternalMemory/InternalMemory_stimulus.v
   - Status: Pass by inspection of generated signal waveform

2. Memory Access Test (See Section 2.2)

3. L2Cache Test(See Section 2.7)

2.4 ICache

1. Full Functionality Test (see Section 2.1)

   This test makes sure that the consistency issue appears and is checked. The read requests issued to ICache is an address recently written to DCache with high probability.

2.5 DCache

1. Basic Function Test This test connects the DCache, L2Cache, bus and 8 InternalMemory devices as slaves. 100k random reads and writes are performed. The reads and writes cover all valid address ranges randomly. The read results are checked against written data. If they do not match, the program halts at where a mismatch occurs.

   - Source: L1Cache/DCache_stimulus.v
   - Status: Pass by automated verification and white box waveform inspection
   - Requires 100M units of time to run

2. Full Functionality Test (see Section 2.1)

2.6 Direct Loader

1. Basic Function Test This test is identical to the test on DCache as in Section 2.5. Set the macro 'TEST_DIRECT', and the test bench performs 100k random R/W and matches the read results with what was written

   - Source: L1Cache/DCache_stimulus.v
   - Status: Pass by automated verification and inspection of waveform

2.7 L2Cache

1. Basic Function Test This test connects the cache group onto the bus, and connects 8 InternalMemory devices as slaves. 100k random reads and writes are performed in 1 - 16 word bursts. The reads and writes cover all valid address ranges randomly. The read results are checked against written data. If they do not match, the program halts at where a mismatch occurs.

   This test is activated by disabling the macro TEST_CACHE_GROUP in the simulation file. Otherwise, the alternative test in section 2.9

   - Source: L2Cache/CacheGroup_stimulus.v
   - Status: Pass by automated verification
   - Requires 10M units of time to run

2. Full Functionality Test (see Section 2.1)

2.8 L2 Adapter

1. Full Functionality Test (see Section 2.1)

2.9 Cache Group

1. Basic Function Test This test connects the cache group onto the bus, and connects 8 InternalMemory devices as slaves. 100k random reads and writes are performed in 1 - 16 word bursts. The reads and writes cover all valid address ranges randomly. The read results are checked against written data. If they do not match, the program halts at where a mismatch occurs.

   This test is activated by defining the macro TEST_CACHE_GROUP in the simulation file. Otherwise, the alternative test in section 2.7

   - Source: L2Cache/CacheGroup_stimulus.v
   - Status: Pass by automated verification
   - Requires 10M units of time to run

2. Full Functionality Test (see Section 2.1)

2.10 Load Manager

1. Basic Function Test

   This test simulates control, memory and bus behavior to check if the device behaves as expected.

   - File: L2Cache/LoadManager_stimulus.v
   - Status: Pass by inspection of generated waveform

2.11 Cache LUT

1. Basic Function Test This test consists of randomly generated operations on the cache. 100k queries are performed. Each query has 0.1 probability to be an access to a location not in the cache, and 0.9 probability to a location in the cache. For the first type, the testbench checks that a cache miss is generated, and the suggested replacement has oldest time stamp. For the second type, the testbench checks that a cache hit is generated, and the correct cache line address is produced.

   - Source: L2Cache/CacheEntry_stimulus.v
   - Status: Pass by automated verification
   - Requires at least 1M units of time to finish

2.12 Time Stamp Manager

1. Basic Function Test

   This test consists of randomly generated input and expected output. If the actual output matches the expected output, the program terminates successfully. Otherwise, it will be stopped at the location where a mismatch occurs.

   - Source: L2Cache/TimeStamp_stimulus.v
   - Status: Pass by automated verification and inspection
   - The test requires 50K units of time to run

3 Peripheral

3.1 串口控制器单元测试

串口控制器单元测试位于 **HDL/testbench/uart_test/uart_test.sv**。被测试对象范围包括 **uart_tx**、**uart_rx**、**uart_top** 模块。

可以直接在 modelsim 工程中运行。

3.2 VGA 控制器单元测试

VGA 控制器单元测试位于 **HDL/testbench/gpu_test/gpu_test.v**。被测试对象是 **gpu** 模块。

可以直接在 modelsim 工程中运行。

4 SoC

SoC 测试为系统级整体测试，包括仿真测试和硬件测试。

4.1 SoC 仿真

SoC 仿真位于 **HDL/testbench/toplevel/test.v** 文件中，可以直接在 modelsim 工程中运行。

该文件构造了一个板级的仿真环境，实例化了 SRAM 和 Flash 的仿真模型，使得整个项目运行可以非常接近真实硬件环境。SRAM 模型使用 Alliance Memory 公司提供的 AS7C34098A 芯片仿真模型。该芯片是 256K × 16 的 SRAM，容量比真实硬件要小，但速度指标接近。Flash 模型使用 Spansion 公司提供的 S29Gl064N01 芯片仿真模型，容量与真实硬件相同。

仿真脚本默认会从 **flash_preload.mem** 文件中加载 Flash 内容初始化文件，从 **ram_preload.mem.*** 文件中加载 RAM 内容初始化文件。初始化文件格式为 modelsim 的 $readmemh 函数支持的格式。可以在下面提到的测试程序目录中编译生成。

4.2 测试程序

汇编测试程序位于 **asm_program** 目录中，这些程序编译后既可以加载到硬件上运行，可以在上述 SoC 仿真环境中运行。

现有测试程序：

- **gpio_test.s** GPIO 控制器测试，读取拨码开关状态并显示在 LED 上
- **leds.s** LED 流水灯程序
- **mem_test.s** 内存读写测试程序
- **uart_test.s** 串口回环测试程序
- **boot.s** NaiveBootloader 固件程序（包含 UART、RAM 测试功能，见 5）
- **flash_test.s** Flash 芯片测试程序，将擦除 Flash

5 Test with NaiveBootloader

NaiveBootloader 是一个引导程序，固化在 BootROM 中。该程序支持串口通信，也就是说可以在上位机发送命令给 Bootloader。利用该程序，我们除了可以引导 uCore 系统外，还可以做大量的测试操作。且测试结果直接在上位机上展示，便于需找故障点。

NaiveBootloader 使用方法见设计文档中相关章节描述。

6 uCore

最终项目验证基于 uCore 操作系统运行测试。要求操作系统正常启动，进入命令提示符。之后依次运行 uCore 自带的用户态测试程序。

6.1 测试结果

除 waitkill 命令外，别的测试程序都成功运行并退出。waitkill 命令处于无响应状态，无法知道结果。由于在 QEMU 中运行 waitkill 命令得到同样结果，我们认为是 uCore 自身的 bug 导致程序失败，与项目无关。

7 Performance

调整 CPU 运行频率使其满足时序约束，并能够稳定运行 uCore，最终得到最高运行频率。

- **Basic 版本（无 Cache）** CPU 主频 10MHz
- **NaiveMIPS++（有 Cache）** CPU 主频 58MHz

2016/11

实验报告

支持指令流水的计算机系统设计与实现

计 42　李则言　2014011292

计 42　杨松涛　2014011316

计 42　李晓涵　2014011297

目录

一、实验介绍

1.1 实验目标

（1）加深对计算机系统知识的理解。

（2）进一步理解和掌握流水线结构计算机各部件组成及内部工作原理。

（3）掌握计算机外部输入输出的设计。

（4）培养硬件设计和调试能力。

1.2 指令集

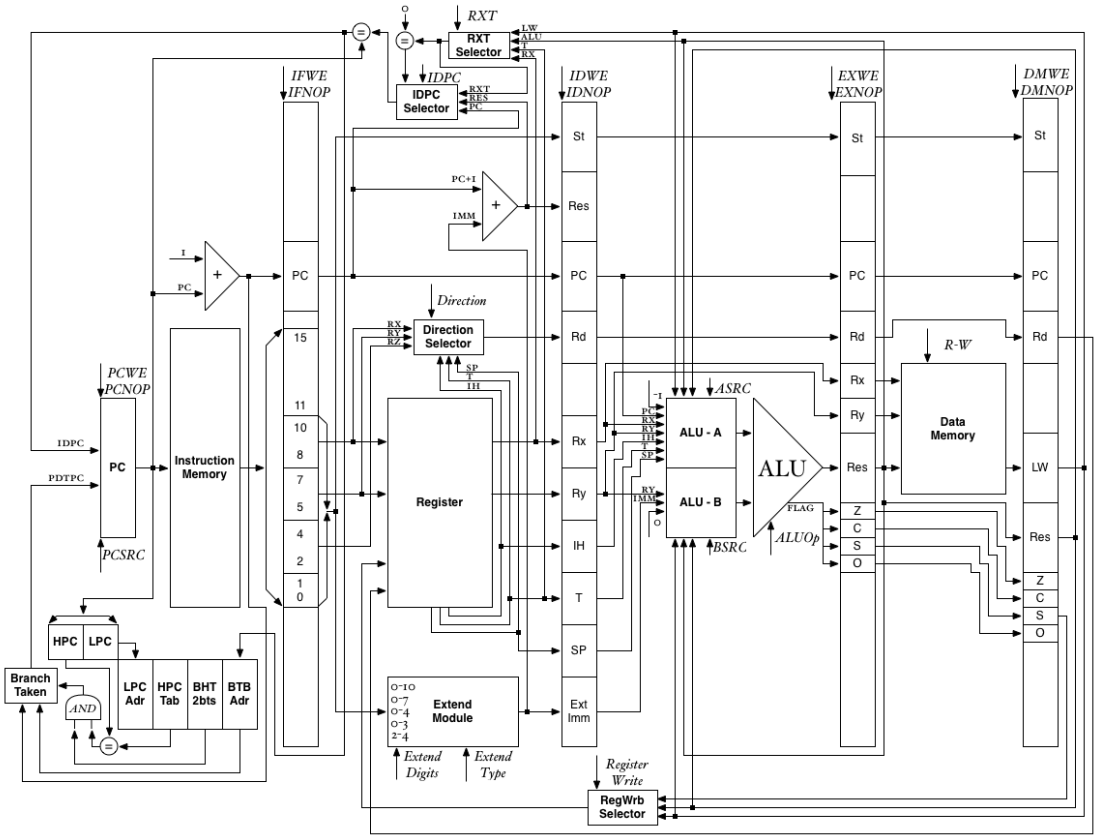
	序号	指令		序号	指令
基本指令	1	ADDIU	基本指令	16	MFPC
	2	ADDIU3		17	MTIH
	3	ADDSP		18	MTSP
	4	ADDU		19	NOP
	5	AND		20	OR
	6	B		21	SLL
	7	BEQZ		22	SRA
	8	BNEZ		23	SUBU
	9	BTEQZ		24	SW
	10	CMP		25	SW_SP
	11	JR	扩展指令	26	MOVE
	12	LI		27	SLLV
	13	LW		28	ADDSP3
	14	LW_SP		29	CMPI
	15	MFIH		30	SLTI

1.3 实验成果

实验实现的 CPU 主频 22.5M，能够正确运行监控程序和测试程序。无延迟槽。支持程序从 Flash 自启动，支持 VGA 与 PS2 键盘。

二、实验设计

2.1 数据通路



数据通路是 CPU 设计的灵魂。像许多标准流水线 CPU 一样，我们将一条指令设计成取指、译码、运行、访存、写回 5 个周期，另有一个控制器单元负责在译码段生成某指令的所有控制信号，并将后续用到的信号保存在段寄存器中。

为了解决数据冲突，增加了数据旁路。

在实际实现中，为了平衡译码段的工作量，将立即数扩展模块转移到了取指段进行。由于时间有限，BTB 模块没有正确实现。此外还在实际实现中增加了一些数据旁路等，图上没有体现。

2.2 控制信号

一个指令所需的所有控制信号的生成在译码段（ID 段）实现，在后续周期才用到的那些，通过段寄存器向后传递。控制信号在生成的时候，综合考虑了各个段寄存器的情况，需要使用数据旁路等。关于数据冲突的判断也是根据控制信号的生成来实现的。详细内容请见“模块划分与实现”的 ControlUnit 部分的介绍。

2.3 数据冲突的处理

数据冲突指的是在 ID 段读取寄存器的值的时候不能得到正确的值。这是由于一条指令的之前的几条指令修改了同一个寄存器的值，但是在完成写回之前之后的指令就需要使用寄存器的值。我们总是在读取寄存器的时候去考虑数据冲突的处理。按照标准的 5 段流水线结构，读取寄存器总是在一条指令的 ID 段，于是我们会在 ID 段生成全部的关于数据冲突处理的信号。

数据冲突的解决方式有两种：**一是插入气泡。**在 ID 段控制器就会决定这条指令时候会由于数据冲突而插入气泡。插入气泡其实就是在这条发生数据冲突的指令之前在硬件上插入一条 NOP 指令。因此我们的实现方式是在下一个时钟周期到来的时候，将 PC 寄存器和 IF 段寄存器写使能关闭，同时将 ID 段寄存器写入 NOP 指令对应的控制信号。这样这条指令就被替换成了 NOP，它本身以及下一条指令又分别重新执行了 ID，IF 段。**二是数据旁路。**虽然寄存器堆中还没有正确的寄存器值，但是可能在某个段寄存器会保存正确的值。这个时候让使用者直接从段寄存器去取就可以拿到正确的值。

在一条指令的 ID 段执行的时候：它的上一条指令处于 EXE 段，对应的控制指令在 ID_RF 中；它的上上条指令处于 MEM 段，对应的控制指令在 EXE_RF 中；再往上一条指令就在 WB 段了。所以判断数据冲突需要向前去查看 3 条指令的写回地址。也就是说需要去查看 ID_RF，EXE_RF，MEM_RF 中的值。总的来说，需要使用寄存器的值的地方有以下几个：**ID 段**，计算跳转的目标地址，以及决定是否跳转。这个时候上上上条指令都尚在 WB 段，所以必须向前查看 3 条指令。**EXE 段**，ALU 的 A，B 操作数。这个时候上上条指令在 WB 段，需要查看两条指令。**MEM 段**，要写入内存的数据。只需要查看上一条指令即可。

一条指令如果要改写寄存器，数据来源只可能有两个：ALU 的计算结果，在 EXE 段即可给出；MEM 访存的结果，在 MEM 段给出。只有 LW 和 LW_SP 指令会访存，因此这两条指令需要特殊对待。

所以，什么时候需要插入气泡？只有当需要寄存器的值的同时正确的值还没有被计算出来的时候才需要。那么有以下几种情况：

1. 这条指令是跳转指令，而且上一条指令的目标寄存器和这条指令的某个源寄存器相同。因为跳转指令在 ID 就需要得到正确的寄存器值，而上一条指令还在 EXE 段。只有 EXE 段结束才能保证得到正确的值。

2. 这条指令是跳转指令，而且上上条指令的目标寄存器和这条指令的某个源寄存器相同，而且上上条指令是 LW 或者 LW_SP。这个时候上上条指令在 MEM 段，需要等 MEM 段结束之后才能保证拿到正确的值。

3. 这条指令不是跳转指令，而且上一条指令的目标寄存器和这条指令的某个源寄存器相同，而且上一条指令是 LW 或者 LW_SP。当这条指令到达 EXE 段，需要使用寄存器值的时候，上一条指令的 MEM 段还未结束，所以没有正确的值。

数据旁路用以下的规则去判断，这个时候已经排除了需要插入气泡的情况：

1. 对于跳转指令在 ID 计算使用的操作数。如果上上条指令的目标寄存器和源寄存器的地址相同且不是读取内存指令。源操作数使用 EXE_RF 中保存的 ALU 的计算结果；如果上上上条指令的目标寄存器和源寄存器的地址相同，那么使用 MEM_RF 中保存的 ALU 计算结果或者 IO 模块的结果。

2. 对于在 EXE 段 ALU 使用的操作数。如果上一条指令的目标寄存器和源寄存器的地址相同且不是读取内存指令。源操作数使用 EXE_RF 中保存的 ALU 的计算结果；如果上上条指令的目标寄存器和源寄存器的地址相同，那么使用 MEM_RF 中保存的 ALU 计算结果或者 IO 模块的结果。

3. 对于在 MEM 段，要写入内存的数据。如果上一条指令的目标寄存器和源寄存器的地址相同，那么使用 MEM_RF 中保存的 ALU 计算结果或者 IO 模块的结果。

三、模块划分与实现

3.1 顶层模块：NaiveCPU

NaiveCPU 是本次 CPU 设计的顶层模块，它的输入输出接口是直接与硬件管脚绑定的，负责整个 CPU 的 IO 等操作，如 Ram 的控制信号和数据地址信息，flash 的控制信号和数据地址信息，串口的控制信号和数据地址信息，VGA 和 PS2 的控制信号和数据地址信息，时钟信息输入和调试信息如 LED 灯的输出等。

这个模块中集合了其他所有功能模块，保存了众多寄存器信号、控制信号等，实现各模块的沟通调度。

3.2 功能模块：ClockModule Digit7Light ExtendModule ALU

这几个功能模块负责实现一些基本的功能，较为简单，如下：

ClockModule	实现时钟分频等，为各个模块提供时钟。
Digit7Light	实现数字译码，便于输出。
ExtendModule	实现立即数扩展。
ALU	实现运算功能。

3.3 控制信号生成：ControlUnit

控制信号的生成在整个 CPU 运转中起着至关重要的作用，这个模块是核心模块之一。这个模块的输入主要来自于指令、PC、段寄存器等，产生的控制信号及其说明如下：

使用周期	控制信号	含义
IF	ExDigitsOp	立即数扩展中，立即数的来源，即立即数在指令中是那些位。
	ExSignOp	立即数扩展是 0 扩展还是 1 扩展。
	INT	是否是中断指令。
ID	DirOp	目的寄存器的选择信号。

	RXTOp	跳转指令时，根据指令来判断要进行比较的目标来源，选择信号，在 IDPCRXT 部件中会用到并解释。
	IDPCOp	跳转指令时，根据指令来选择要跳转的目标，选择信号，在 IDPCRXT 部件中会用到并解释。
	RegWrbOp	写回寄存器的选择信号。
EXE	AluOp	ALU 操作类型的选择，如加法、减法、与操作、或操作等。
	AMuxOp	ALU 中 A 操作数的选择信号。
	BMuxOp	ALU 中 B 操作数的选择信号。
MEM	SWSrc	写回操作时，写回的数据来源的选择信号，选择是 RX 还是 RY。
	SWMuxOp	写回操作时，写回的数据来源的选择信号。
	RamRWOp	Ram 是读还是写的控制信号。
	MemEN	是否需要访存
段寄存器使能信号	IF_RFOp	IF 段寄存器的使能信号，两位，包含读写使能、清空信号。
	ID_RFOp	ID 段寄存器的使能信号，两位，包含读写使能、清空信号。
	EXE_RFOp	EXE 段寄存器的使能信号，两位，包含读写使能、清空信号。
	MEM_RFOp	MEM 段寄存器的使能信号，两位，包含读写使能、清空信号。
	PC_RFOp	PC 段寄存器的选择信号，选择 PC 是取预测 PC 还是跳转 PC 等。
	PC_RFWE	PC 寄存器的写使能信号。

3.4 寄存器堆：Registers

保存寄存器的值，实现寄存器的读写。

值得一提的是，寄存器的读写是没有冲突的，只需判断一个需要读出的是否正好是所写回的值。

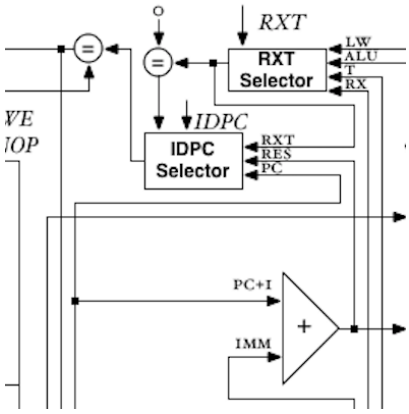
3.5 段寄存器：IF_RF　ID_RF　EXE_RF　MEM_RF

段寄存器负责一些数据和信号在各流水段之间的传输，段寄存器的读写使能和清空信号也担负着流水线暂停、插入空指令等重任。

3.6 PC 相关：PC_RF　IF_PCAdder　ID_PCAdder　IDPCRXT

PC_RF 即 PC 寄存器，保存最近一次的 PC 值；IF_PCAdder 负责在 IF 段进行 PC+1 的操作；在跳转指令时，ID_PCAdder 负责在 ID 段将 PC 与立即数相加，得到一个跳转地址，传给后续模块进行处理。

IDPCRXT 是一个比较重要的部件，实现的是数据通路中 ID 段的这一部分功能：



可以看出包括两部分。一部分是根据指令来判断要进行比较的目标来源，可能是 T 寄存器、RX 寄存器、ALU 的运算结果（旁路取得）或者访存得到的 LW 结果（旁路取得），拿到这个来源之后与 0 判等，再根据判等结果和指令，判断跳转目标是 PC（不跳转）、ID_PCAdder 的运算结果、寄存器中取得的值。

3.7 选择器：AMux BMux DirectionModule RegWrbModule

AMux 和 BMux 是 ALU 操作数的选择器，选择数据来源让 ALU 进行运算。

DirectionModule 是目标寄存器的选择器，根据从控制单元拿到的控制信号来选择目标寄存器。

RegWrbModule 是写回段写回数据的选择器，有可能是 ALU 的运算结果或者 flag 的值、MEM 段 LW 的结果。

3.8 IO 单元：MemUart

MemUart 模块集中完成 CPU 的 IO 操作，包括访存、串口访问、flash 访问，其中内存的读写在 IF 段和 MEM 段都要使用。我们将 Ram1 作为数据存储区，Ram2 作为指令存储区。

MemUart 模块有两个状态机，使用的是 45MHz 的时钟。流水段寄存器使用的是这个时钟的 2 分频，因此主频是 22.5MHz。通过这两个状态，IO 模块可以完成 MEM 和 IF 所有的 IO 操作。也就是说，如果出现 MEM 段访问指令内存的情况，可以在一个 CPU 时钟周期内既完成 MEM 段的操作，又完成 IF 段的操作，不需要暂停流水线。

使用两个状态完成两次内存读写，或者一次串口读写加一次内存读是比较困难的。在这一模块的设计上，我们花了比较大的精力，设计出了非常精巧的时序控制逻辑。以下分情况讲述我们的设计思路：

MEM 段访问串口

因为我们将 RAM2 作为指令内存，所以访问串口和 IF 段的取值是没有冲突的。那么这其实是一种比较简单的情况，因为访问串口和取指可以同时进行。

但是考虑到 MEM 段访问 RAM2 的情况，IF 的取值永远只利用第二个状态。

1. 在第一个时钟周期，RAM2 不使能。
2. 在第二个时钟上升沿之后，
    - 将 RAM2 的控制信号赋为读状态(EN<=1，OE<=0，WE<=1)。
    - 地址给成 PC。

- 数据总线赋高阻。
- 将数据总线输出链接到 IO 模块的 INS 输出端口。

那么当内存中读出数据的时候，数据总线上就是指令的内容，IO 模块将其输出到段寄存器。当下一个时钟到来的时候，指令就被保存到了段寄存器中。这就完成了取值。

串口和 RAM2 是完全独立的，互不影响:

在访问串口的时候，RAM1 始终不使能。同时将数据总线 1 始终链接到 IO 模块的访存结果输出上。

1. 读串口

WRN 总是置 1。

在第一个时钟上升沿的时候，将 RDN 置 1，数据总线置高阻。这是为读取串口做准备。在第二个时钟到来时，如果 dataready 为 1，就将 rdn 置为 0。之后 CPLD 就会将数据从缓冲区发送到数据总线 1 上。下一个时候到来的时候，段寄存器就会保存下来。

2. 写串口

RDN 总是置 1。

在第一个周期将 WRN 置为 1。第二个周期，如果 tbre 且 tsre 都为 1，那么将 WRN 置为 0 同时将要写入的数据放到数据总线上。

3. 获取串口状态

这是最简单的情况。直接将 dataready 放到输出的第 1 位，将 tbre&&tsre 放到输出的第 0 位。

MEM 段访问 RAM1

因为 RAM1 和 RAM2 互不干扰，所以这种情况也比较简单。

访问 RAM1 的时候 WRN 和 RDN 始终为 1，以关闭串口。

1. 写 RAM1

在第一个时钟周期，将 RAM1 使能，但是不读也不写(WE<=1，OE<=1)。在第二个时钟周期，将地址和数据放到总线上，同时将 WE 置为 0。

2. 读 RAM1

在第二个时钟到来的时候将 RAM1 置为读状态(EN<=0，WE<=1，OE<=0)，同时给出地址，数据总线赋高阻，同时将数据总线连到 IO 模块的输出上。

MEM 段访问 RAM2

1. 读 RAM2

控制信号和之前介绍的取值给出的控制信号是一致的，而且两个状态控制信号相同。

有一个问题在于第二个状态的时候，数据总线 2 上的值已经是指令而不是 MEM 访存的结果。因此此时 IO 模块的访存输出应该和一个寄存器而不是直接和数据总线 2 相连。这个寄存器在第二个时钟到来的时候保存下来数据总线 2 的值。

2. 写 RAM2

控制信号的写 RAM1 相同，但是要在第一个时钟到来的时候就给出。

有一个问题在于内存要写入的数据必须要经过一个选择器，这个选择器的输入来自段寄存器。因此在第一个时钟到来的时候还不能知道具体要写入的数据。所以给数据总线赋写入数据要在第一个时钟周期，而不是第一个时钟的上升沿。在第一个时钟到来后不久，正确的数据就会到达。在第二个时钟到来之前，还有足够的时间将其写入内存。也正因此之前写 RAM1 才将控制信号放到了第二个状态给出。

在访存过程中我们还遇到了一些问题，由于 IO 的模块的第一个时钟上升沿和 EXE 段寄存器，PC 寄存器的时钟上升沿是同时的。因此这个时候 EXE 段寄存器中的控制信号和数据还是上一条指令的，PC 寄存器中还是上一个周期的。这就会导致错误。不过，由于我们取值总是在第二个状态，那个时候 PC 就有了正确的值了，所以现在只需要关注 EXE 段寄存器。我们通过一个数据旁路解决这个问题。因为 MEM 段访存的地址永远都是来自 ALU 的计算结果，控制信号全部都是来自 ID 段寄存器，因此我们将这些数据直接连到 IO 模块。在第一个时钟上升沿的时候，使用这些数据。在其他时候都正常使用段寄存器的数据。

3.9 DCM_SP 部件

单纯使用 VHDL 代码只能实现对时钟的简单整数分频，而不能很容易地实现任意分频和倍频。在实际调试的过程中，我们发现接入 CPU 的时钟为 50MHz 部分情况下会出错，但是如果使用 2 分频或者 3 分频又显得太慢。因此我们需要对时钟能进行更精确地控制。

DCM_SP 是 Xilinx 自带的 IPCore 组件。通过这个组件，我们可以调用芯片上的锁相环部件，实现精确的时钟控制。

在工程中添加新的源文件的时候，选择 IP Core。调用锁相环的部件包括 PLL，DCM，DCM_SP 等，每个芯片的支持是不同的。本次实验使用的 Spartan3E1200 芯片只支持 DCM_SP。在 DCM_SP 的设置中需要选择 CLKFX 输出，并且选择输出频率。在代码中，将 DCM_SP 部件作为一个 Component 调用即可。

值得注意的是，由于锁相环实现的限制。接入锁相环的输入的外部时钟输入将只能接入锁相环(而不能在从这个输入接到别的部件)。这似乎是为保证时钟的信号有足够的强度。为了使用未分频的原始输入信号，我们只能从 DCM 的输出中链接 CLK0 输出端口。

通过使用 DCM_SP 部件，我们的 CPU 接入的时钟为 45MHz。这是保证正确运行前提下的最大值。

四、扩展功能

本节介绍所实现的扩展功能，包括 VGA 屏幕显示，PS2 键盘输入，和 Flash 自启动。

其中 VGA 和 PS2 部分源码参见 VGAController.vhd，Flash 部分源码参见 MemUart.vhd。

4.1 VGA 屏幕显示

本次实验中，VGA 部分实现了一块 640 * 480 的屏幕，共做了 4 种显示模式（可进行切换），分为“单张图片”、“指令画图”、“固定寄存器”和“简易 Vim”。

这里部分模式中使用了 Xilinx 的 IPCore 中 BlockMemory 模块，定时从其

中取得数据，再在 25 MHz 时钟下送到 VGA 处。通过改变 BlockMemory 的种类和接口，实现了不同显示模式。

4.1.1 单张图片

```
component lxh_mem is
    Port ( clkA : in std_logic;
           AddrA : in std_logic_vector(12 downto 0);
           DoutA : out std_logic_vector(15 downto 0));
end component;
```

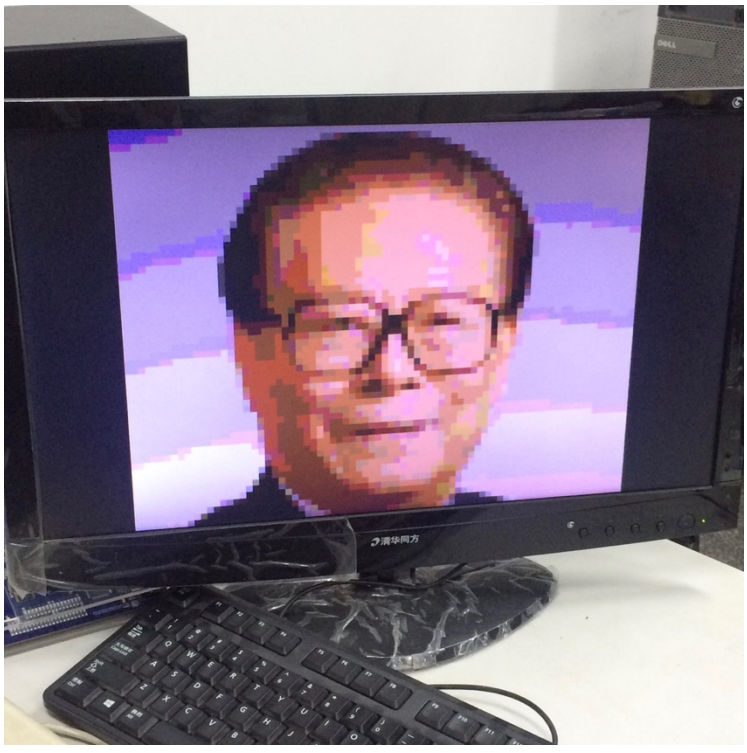
BlockMemory 设置为 Single Port ROM 形式，通过预先使用脚本生成的 .coe 文件进行初始化。ROM 单个存储向量长为 16，其中第 10~8 位、6~4 位、2~0 位分别存储 RGB 三种颜色的三位长数值，其他位无效。由于存储空间有限，图片像素点定为 8 * 8，全图分辨率为 80 * 60，ROM 空间使用 128 * 64 = 8192 个向量。

4.1.2 指令画图

```
component zz_mem is
    Port ( wea : in std_logic_vector(0 downto 0);
           AddrA : in std_logic_vector(12 downto 0);
           DinA : in std_logic_vector(15 downto 0);
           clkA : in std_logic;
           AddrB : in std_logic_vector(12 downto 0);
           DoutB : out std_logic_vector(15 downto 0);
           clkB : in std_logic);
end component;
```

BlockMemory 设置为 Single Dual Port RAM 形式，存储方案和“单张图片”模式相同，但增加了输入接口，可以对指定地址的数据（即屏幕上特定位置的 8 * 8 的像素点的颜色）进行设置。

在 MemUart 内存/串口读写模块中，监测对内存上指定地址空间（0xE000~0xFFFF）的写命令，如发现便将写入数据送至 VGAController 模块，同时设置写开关，将此数据写入 RAM 显存中。



4.1.3 固定寄存器

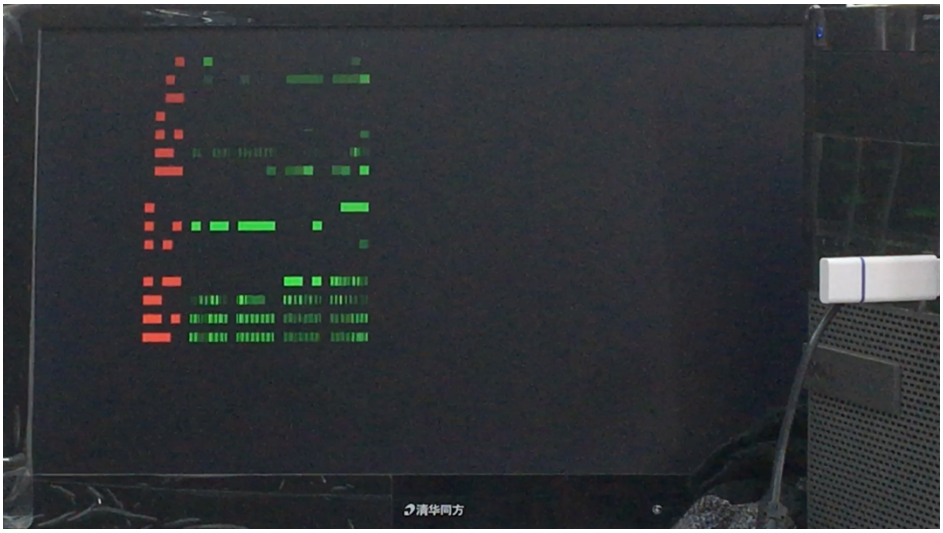
```
constant half : std_logic_vector(2 downto 0) := "001";

impure function print_register(input: std_logic_vector(15 downto 0))
return std_logic_vector is
    variable tempG : std_logic_vector(2 downto 0) := (others => '0');
begin
    ...
    return tempG;
end print_register;
```

这一模式是最早做出来的（甚至先于第一次代码完全编译通过），目的是显示寄存器的值，尽可能方便调试过程。由于此时尚未开始使用 IPCore，因此布局全部靠手动画图（大量条件控制）来实现。

每一寄存器左边的红色方块是二进制序号的标示，便于查看。右侧的 4 * 4 共计 16 个方块显示寄存器的值，左边为高位。具体颜色设置上，置为 1 的值使用 111（满色，亮而醒目），置为 0 的值使用 001（仍然有淡淡的虚影，便于占位和美观），无关区域使用 000 置黑色。寄存器部分的显示提了函数出来，在后续调试过程中只需简单设置前方标号，再直接调用函数显示寄存器的值即可。

这一显示方案和常规的 4 位 16 进制数字表示相比，在调试过程中有较为显著的优势。4 个数字位置相同，在接入晶振时钟时难以看清低位的变化，而将每一位的位置分开，既可以直接按二进制位进行观察（如指令操作码为 5 位，此种表示不需要换算），也可以直观看到变量的改变，对程序运行状态有良好的把握。



4.1.4 简易 Vim

```
component char_mem is
    Port ( clkA : in std_logic;
           AddrA : in std_logic_vector(10 downto 0);
           DoutA : out std_logic_vector(7 downto 0));
end component;

component fifo_mem is
    Port ( wea : in std_logic_vector(0 downto 0);
           AddrA : in std_logic_vector(11 downto 0);
           DinA : in std_logic_vector(7 downto 0);
           clkA : in std_logic;
           AddrB : in std_logic_vector(11 downto 0);
           DoutB : out std_logic_vector(7 downto 0);
           clkB : in std_logic);
end component;

caddr_origin <= y(8 downto 4) & x(9 downto 3);
char_addr <= char(6 downto 0) & y(3 downto 0);
```

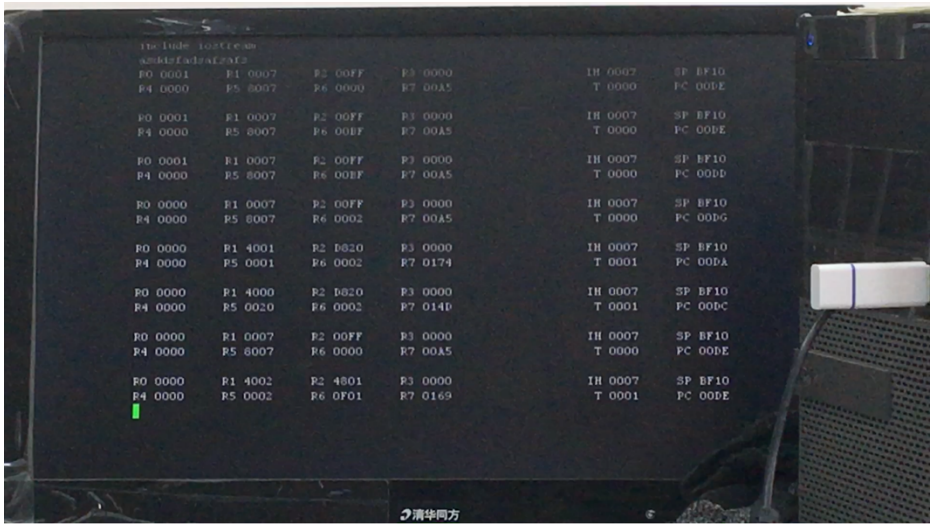
Vim 模式实现了一个具备多种功能的文本编辑器。屏幕被分为 80 * 30 个 8 * 16 的方阵，每个方阵上可以放置一个 8 * 16 点阵的字模。通过拼接纵向扫描坐标`y`低 4 位之上和横向扫描坐标`x`低 3 位之上的地址，可以定位到一个特定的方阵，共计 128 * 32 = 4096 个方阵。显存将 BlockMemory 设置为 Single Dual Port RAM 模式，单个存储向量长为 8，其中低 7 位标示 128 位的 ASCII 码。

字模部分将 BlockMemory 设置为 Single Port ROM 形式，通过预先设计好的 .coe 文件进行初始化。ROM 单个存储向量长为 8，代表一个字模的一行。将 ASCII 码 128 个字符的 7 位二进制序号和 16 行的 4 位二进制序号（VGA 纵向扫描坐标`y`的低 4 位）拼接为一个 11 位的地址，从字模库中进行查找。

```
if pr(CONV_INTEGER(not x(2 downto 0)))='1' then
    tempR <= (others => '1');
    tempG <= (others => '1');
    tempB <= (others => '1');
else
    ...
```

`pr`作为查找到的结果，通过检查横向扫描坐标`x`低位处的值（因字模文件顺序问题，需进行翻转以实现`8-x`），即可得知该点处是否应进行显示。

关于 Vim 的输入及逻辑控制，将在下一节 PS2 部分进行阐述。



4.2 PS2 键盘

本次实验中，PS2 部分实现了对一块具有 PS2 接口的键盘按键的识别。出于对连续按键、交错按键等按键情况种类繁多、识别逻辑过于复杂的考虑，最后统一只利用状态机识别按键松开的瞬时动作，因而不识别连续按键。经过多人打字实验，当 WPM 达到一定程度时，由于各人习惯（常规键盘识别的是按下去的瞬时）可能会出现错误，但总体而言，偏差并不多。况且由于是显存实现，修改也较为方便。

另外，考虑到 PS2 信号不稳定，特地做了频率限制，每 0.02 秒只允许识别一个按键。经试验，效果良好。

4.2.1 显示模式切换

实验中将 ctrl 和 alt 作为模式切换按键，分别执行“下翻一页”和“上翻一页”的功能，可以将屏幕在上一节所述 VGA 的四种模式间做切换。

4.2.2 常规按键识别

实验中对键盘上主键盘区的几乎所有按键都做了识别（除 backspace 外）。控制器在逻辑上维护了一个“当前输入位置指针”指向某个 80 * 30 中的字点阵，并在显示屏上维护了一个“下一输入位置的光标（绿色）”，二者进行了位置上的

绑定。每当输入合法有效的字符，便会将当前字符的 ASCII 码写入显存上的指定位置（由当前输入位置控制），并右移输入位置（和光标），而 VGA 则会不断扫描显存上各地址 ASCII 数值，送至字模模块取得点阵内容进行显示。

4.2.3 特殊按键逻辑识别

- Shift & Caps Lock

当按下 Shift（L，R） 或 Caps Lock 按键，光标变为红色，标示进入大写锁定状态。此状态下一切有上标的按键都会识别为上标，例如`a`识别为`A`，`3`识别为`#`，`[`识别为`{`等。直到再次按下左或右 Shift 或 Caps Lock，光标恢复为绿色，大写锁定状态解除。

- Arrows & Home & End & Page Up & Page Down & Tab & Enter

无论是否为大写锁定状态，按下四个方向键都会将光标向对应方向移动一格。特别地，左向键在行初会上移至上一行末，右向键在行末会移至下一行初，并且在第一行初或最后一行末会移至最后一行末或第一行初。

Home 和 End 键会使光标移动至本行初或本行末，Page Up 会将光标向上移动四行，若中途遇第一行则再移动至最后一行；Page Down 则向下，其他同理。

当按下 Tab 键，光标会向右侧移动四个格子，在跨行情形上与右方向键一致。特别地，在大写锁定状态下的 Tab 会识别为 Shift Tab，向左侧移动。

当按下 Enter 键，光标会移动至下一行初。在最后一行按下 Enter 键，会移动至第一行初。

- Windows

我们特别设计了 Windows 的特殊功能——可以即时地在下两行输出所有寄存

器的值，随后光标被移动至再下一行初。当程序运行起来后，连续按下 Windows 键可以看到寄存器的值在不断变动。

- Delete

考虑到文字的可读性和输入错误的可能，我们设计了删除模式。

在非大写锁定状态下按下 Delete 按键，Vim 会认为进入行删除模式。此时若再次按下 Delete，则光标所在行被删除（全部以空白替代），光标移动至本行初；而任何其他按键都会退出行删除模式，并执行其效果。

特别地，在大写锁定状态下按下 Delete 按键，Vim 会认为进入页删除模式，并对第二个 Delete 执行页清空操作，光标移动至第一行初。同理，任何其他按键都会退出页删除模式，并执行效果。

- Esc: Command Mode

在上述模式（称为插入模式）下按下 Esc 按键，光标变为黄色，Vim 切换为命令模式。此时包括 Shift、Delete 等在内的大部分输入编辑命令无效，光标控制按键则正常。在此状态下，支持 Vim 的移动命令方式：按下一个数字，然后对 H，J，K，L 四个按键执行所按按键次数，如`3 H`会将光标一次性向左移动 3 个格子（遇页内边界会在边界停止，不会出现跨行等情况）。出于实现方式的考虑，暂且仅支持一位数字重复命令。

4.3 Flash 自启动

Flash 自启动部分原本想与当前的顶层部件分开，再加一个顶层部件来控制 boot 和运行过程，但是由于 Ram2 数据 inout 类型的限制，只能整合在 MemUart 部件中。具体实现也较为简单，在 MemUart 部件中增加一个判断是否 boot 结束的量，在每次时钟沿到来的时候都进行判断，如果在 boot 阶段则进行 Flash 读写或指令写入，否则执行过程访存。

Flash 的读取用状态机实现，分为以下几个阶段：

1. Flash 的数据总线置 0X00FF，WE 置 0。
2. WE 置 1，这样向 Flash 送了一个 00FF 数据，表示我需要读取。
3. OE 置 0，数据总线置高阻，准备读取。
4. OE 置 1，数据已传送到数据总线，可以接收。

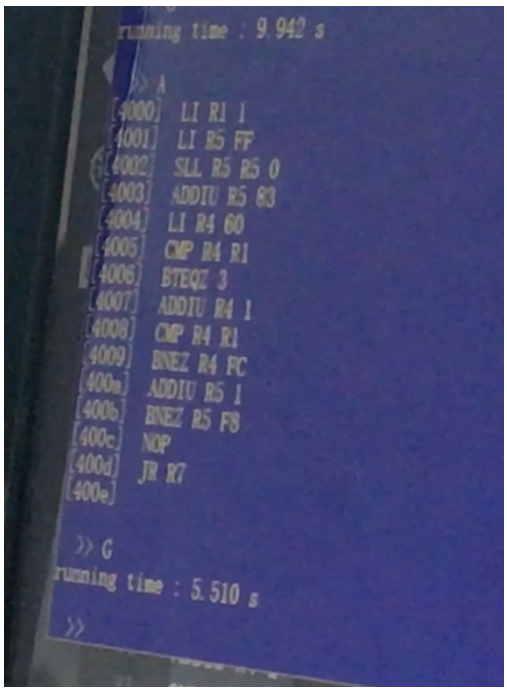
每次程序启动前都会将 Flash 中的指令放到 Ram2 中，完成自启动。

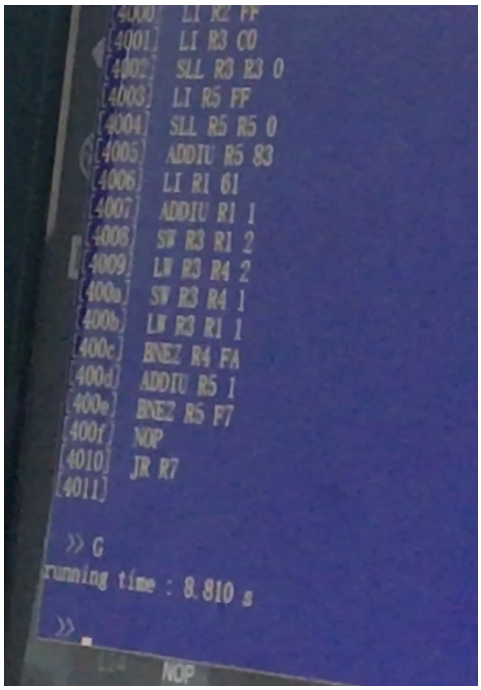
五、性能测试

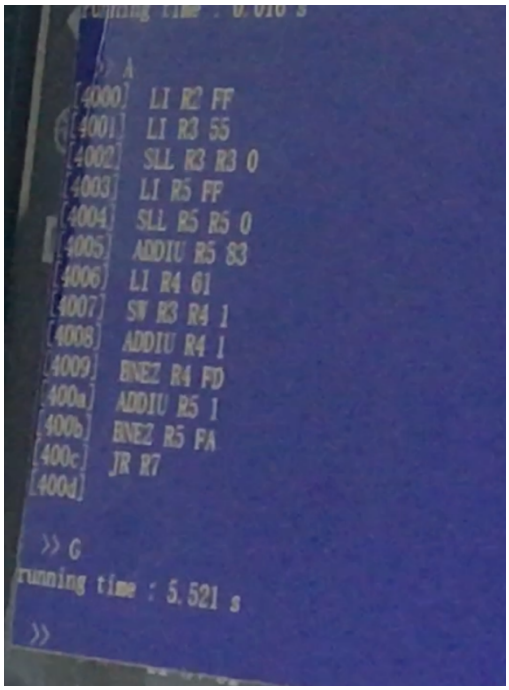
几个测试程序的运行情况：

测试程序 1	6.639s
测试程序 2	9.942s
测试程序 3	5.510s
测试程序 4	8.810s
测试程序 5	5.521s

运行时照片（运行时忘记截图，图片不太清晰）：







六、实验总结

经过历届学长学姐的安利、学生节节目的渲染，“造计算机”早已成为贵系的一个标签，对于这样一个非常富有挑战性的工作，我们既感到很有压力，又充满了激动和期待。

虽然这个项目整体有难度，但是还是有很多可以选择和发挥的余地的。我们从一开始就决定要做流水线，虽然这可能比多周期要困难一些，当时也设想了一些扩展，比如 BTB 动态预测、VGA、Flash 自启动、中断等，但是最后只完成了 VGA 和 Flash 自启动，虽然有一些遗憾，但是这个过程中也收获了很多知识。在这个过程中可以体会到，大家都是愿意去接受挑战的，不管结果如何，相信这种勇气都是一种宝贵的财富。

完成的过程中，确实经历了很多困难和沮丧，有时候花费一上午的时间只是配好一个环境，调了一整天的才调好内存和串口，用一个晚上完成的重构突然想起前一版本的 bug 在哪儿然后 reset --hard……虽然痛苦，但是其实都是必经的阶段吧。也还是有些好运气的，比如 ControlUnit 没怎么出错就都跑通了，Flash 移植到 MemUart 模块一下就打出了 OK……人品守恒，诚不我欺。

对于我们来说，除了知识上的收获，这项大作业也锻炼了我们团队合作和协

调的能力；对于整个计算机系来说，造计算机很大地增加了我们对专业的认同感——从我们刷屏的朋友圈就看得出来。

听说大家觉得造计算机越来越简单了，不能体现贵系神课的水平了，老师要提高难度了。然而我们反正是造完了。希望学弟学妹们能造出更快更好的计算机：）

附：

项目 GitHub 地址：https://github.com/lizeyan/THINPAD

需求文档

金涛 何纬捷 李品农

January 2018

Contents

1 文档说明

1.1 背景

本项目旨在设计一个基于MIPS32的CPU及其周边硬件，能够支持标准MIPS 32 指令集的一个子集，最终运行ucore操作系统。本项目是计算机组成原理和软件工程的联合实验，项目需求方为计算机组成原理课程。承担此项目的是404NotFound小组。

1.2 项目概览

本项目需要设计实现的部分包括CPU和外设控制，采用VHDL和Verilog两种语言编写。

1.2.1 CPU

CPU的总体设计主要包括指令系统、基本数据通路、流水线结构、MMU、异常与中断处理。

1.2.2 外设

外设控制部分主要包括SRAM读写控制、Flash读控制、串口通信、DVI控制。

1.3 项目目标

能够通过引导程序将ucore操作系统从Flash 加载到RAM中，并顺利运行，接受用户输入，执行操作系统相应功能。

1.4 开发环境

1.4.1 硬件环境

使用提供的Thinpad开发板：搭载Xilinx Artix-7系列FPGA：XC7A100T，两组SRAM内存，每组4MB容量、32位数据线以及8MB NOR Flash闪存。外围接口包括：SL811 USB，DM9000网卡，TFP410 DVI图像输出。

1.4.2 软件环境

EDA工具使用Xilinx Vivado 2017.2以上版本。

2 ucore操作系统解析

2.1 简介

ucore 是一款类似Unix 的教学操作系统，本项目使用的是ucore 的MIPS移植版，称为ucore-thumips。上述移植版的GitHub项目中含有一份简要文档（以下简称为简要文档），以下内容也部分参考了该文档。

2.2 Boot阶段

Boot 阶段是操作系统启动阶段，在硬件进行初始化后，引导程BootLoader从硬盘中加载操作系统到内存，然后将控制权交由操作系统，操作系统初始化相关状态，随后进行进程调度。整个阶段可以分为3 部分：硬件初始化阶段，BootLoader阶段，操作系统初始化阶段。

2.2.1 硬件初始化阶段

硬件接收到Reset信号后，根据MIPS标准，将进行如下初始化：

1.初始化CP0寄存器：如Status 寄存器。

2.初始化TLB：TLB中有一位表示是否可以匹配的“隐藏”位，这一阶段需要将之置为禁止。

3. PC 初始化：取值地址将从VA 0xBFC00000开始。硬件完成初始化后，将从VA 0xBFC00000 开始执行指令，进入BootLoader 阶段。

2.2.2 Bootloader阶段

BootLoader 被存储在FPGA 的ROM中,由于编译后的BootLoader不足1KB，为方便起见，MMU 将VA 0xBFC00000-0xBFC00FFF 映射到ROM。

在BootLoader 中，定义了Flash 的起始地址FLASH_START为0xBE000000，且Flash 大小FLASH_SIZE 为16MB，所以MMU 需要将VA 0xBE000000-0xBEFFFFF映射到Flash。

在Bootloader中，认为Flash 的数据线为16 位，通过两次访存取出32 位长的指令，因而在控制读取Flash时，应使用16位读取模式并将高16位补0。

BootLoader需要将操作系统从Flash拷贝到内存（也就是SRAM）中，根MIPS标准，操作系统应被拷贝到kseg0(VA 0x80000000-0x9FFFFFFF，对应PA0x00000000-0x1FFFFFFF，512MB 大小)，但是我们的SRAM 只有8MB大小，这显然是不够的。注意到kern/mm/memlayout.h 文件定义了KMEMSIZE常量，其指明了内存大小为1MB，因而MMU只需要将VA0x80000000-0x800FFFFF映射到PA 0x00000000-0x000FFFFF 即可。

在BootLoader 完成拷贝操作系统的任务后，会跳到VA 0x80000000，进而执行操作系统初始化过程。

2.2.3 操作系统初始化阶段

这个阶段中，首先进入entry.S 的kernel_entry过程，该过程进行CP0_CAUSE、CP0_STATUS、 CP0_EBASE 寄存器的初始化。然后进入init.c 的kern_init过程，该过程依次进行下列初始化：

1. 将所有TLB 表项无效化。

2. 中断初始化：将CP0_STATUS 的IM 段置为全0，禁用所有中断。

3. 串口初始化：将CP0_STATUS 的IM 段中对应串口的位置为1，打开串口中断。

4.时钟中断初始化：读取CP0_COUNT，加偏移量后存入CP0_COMPARE。并将CP0_STATUS的IM段中对应时钟的位置为1，打开时钟中断。

随后输出内核调试信息，进行高层初始化，高层初始化与硬件关系不大，包括：物理地址管理器初始化、虚拟地址管理器初始化、进程调度初始化、IDE和文件系统初始化。最后将CP0_STATUS 的IE 位置为1，所有中断生效，进入进程调度状态。

2.3 异常处理

2.3.1 异常处理流程

1.硬件准备好异常相关寄存器，跳到异常处理入口，异常处理向量定义在trap/vector.S 中

2. 异常处理入口定义在trap/exception.S 中，操作系统首先保存异常现场状态，根据异常的不同调用相应的处理代码，如果异常类型不被支持，则陷入panic 状态。

3.异常处理结束后，进入异常返回过程，恢复保存的异常现场状态，继续执行被中断的程序。

2.3.2 异常类型

1. **中断(Int)**
(a)**时钟中断**：调用timer_list 中待调用的进程，而后重启时钟。
(b)**串口中断**：发生串口中断的时机为有数据在串口等待写入，中断发生后系统会读取串口准备好的数据，写入标准输入。

2. **TLB 缺失异常**：该类异常有两个，分别是load 时异常TLBL 和store 时异常TLBS，二者处理方式一致，均从CP0 的BadVAddr 寄存器取得缺失地址，重填TLB 表项。

3.**系统调用Sys**：根据相应的系统调用类型，取得相关参数，进行系统调用。ucore 支持的系统调用见下表。

4. 地址不对齐异常（load 时异常AdEL 和store时异常AdES），非法（保留）指令异常(RI)，协处理器不可用异常(CpU)，算术溢出异常(Ov)：此类异常如果发生在内核态，则系统陷入panic 状态；如果发生在用户态，则用户进程退出。

Syscall 序号	Syscall 说明
0	sys_exit
1	sys_fork
2	sys_wait
3	sys_exec
4	sys_yield
5	sys_kill
6	sys_getpid
7	sys_putc
8	sys_pgdir
9	sys_gettime
10	sys_sleep
11	sys_open
12	sys_close
13	sys_read
14	sys_write
15	sys_seek
16	sys_fstat
17	sys_fsync
18	sys_getcwd
19	sys_getdirentry
20	sys_dup

Figure 1: 系统调用类型

3 基础部件

3.1 ALU

ALU 全称为算术逻辑单元（Arithmetic Logic Unit），是能实现多种算术运算和逻辑运算的组合逻辑电路，也是CPU 中的核心组成部分。例如，存储访问指令用ALU计算目标数据的地址，算术逻辑指令用ALU 执行运算，而分支跳转指令用ALU进行比较。根据ucore操作系统的实际需求，我们的ALU 只需要进行基本的整数算术逻辑运算即可，不需要支持浮点数运算。附录所示ucore操作系统的47 条基本指令中，包含了8 条算术指令、6 条分支指令、8 条逻辑指令和6 条移位指令。从这些指令出发，ALU 运算需求可以整理为下表。

操作码	功能	描述	操作码	功能	描述
ADD	$A+B$	加法	SLL	A sll B	逻辑左移 B 位
SUB	$A-B$	减法	SRL	A srl B	逻辑右移 B 位
AND	A and B	位与	SRA	A sra B	算术右移 B 位
OR	A or B	位或	EQU	$A=B$	判断是否 A 等于 B
XOR	A xor B	位异或	SLT	$A<B$	判断是否 A 小于 B
NOT	not A	位非			

Figure 2: ALU运算功能表

具体实现需求中，ALU 接受两个32位整数与控制信号作为输入，以一个32位整数作为输出。很显然，ALU 根据不同的控制信号执行不同的操作。这些控制信号来源于指令当中的特定字段，在流水线结构的译码阶段生成。如果是算术逻辑指令，ALU 的结果将写回寄存器；如果是存取指令，ALU 的结果可作为读写寄存器的地址；如果是分支指令，ALU 的结果会被用于决定下一条指令地址。

3.2 乘法器

MIPS32 指令集的乘法指令与其余算术逻辑指令略有差异。它接受两个32位整数为输入，以64位整数保存乘法运算结果，存储在HILO 寄存器内。其中，HI 寄存器保存结果的高32 位，LO 寄存器保存结果的低32 位。这里运算结果的位数和存储位置发生了变化，因此我们需要将乘法器视为独立的运算单元，而不再是ALU 的一部分。由于ucore 操作系统中并未涉及到MADD、MADDU、MSUB、MSUBU 等乘法与加减法混合指令，我们的乘法器只需要一个时钟周期就可以完成计算，从而避免了流水线调度的问题。

乘法器在具体实现中，主要用于执行MULT、MULTU 两种指令，输入两个32 位整数，将计算结果保存在HILO 寄存器中。HILO 寄存器的读写操作，可以通过MFLO、MFHI、MTLO 和MTHI 四种指令完成。

3.3 寄存器组

MIPS32 CPU 有32 个通用目的寄存器，被命名为$0 至$31。各个寄存器的功能及汇编程序中的使用约定如下表所示：

在采用MIPS32 架构的硬件系统中，寄存器组内包括上述的32 个通用目的寄存器，并且每个寄存器的位宽均为32 位。寄存器内的值可以在一个时钟周期内的任意时间读出，但只能在时钟上升沿处更改。MIPS32 指令集中的R 型指令（如算术逻辑指令、条件移动指令等）最多涉及三个不同寄存器的访问，并且规定从两个寄存器中读取数据，并将结果写入最后一个寄存器中。因此，寄存器组模块需要接收全局的时钟信号，支持对任意两个寄存器同时进行的读操作，和在时钟边沿处对某一个寄存器进行的写操作。

此外我们注意到，并非所有指令都需要将结果写入寄存器，而且对0 号寄存器的写入操作不符合寄存器功能要求，这意味着我们还需要为寄存器组添加必要的写使能信号，以规避潜在风险。

寄存器号	寄存器名	功能
$0	$zero$	常量 0
$1	a_t	保留给汇编器
$2-$3	$v_0 - v_1$	函数调用返回值
$4-$7	$a_0 - a_3$	函数调用参数
$8-$15	$t_0 - t_7$	调用者保存寄存器
$16-$23	$s_0 - s_7$	被调用者保存寄存器
$24-$25	$t_8 - t_9$	调用者保存寄存器
$26-$27	$k_0 - k_1$	保留给异常处理使用
$28	gp	全局指针（Global Pointer）
$29	sp	堆栈指针（Stack Pointer）
$30	fp	帧指针（Frame Pointer）
$31	ra	返回地址（Return Address）

Figure 3: MIPS32 CPU 寄存器名称及功能表

3.4 CP0

实验用操作系统ucore 没有实现浮点运算，因此我们不需要实现CP1 和CP3。CP2 没有特殊功能，同样不用实现。只有CP0 是唯一不可选的协处理器，由于它涉及中断处理、提供可选配置、观察并控制系统缓存或时钟、地址转换等关键功能，我们必须加以足够重视。MIPS32 架构定义的协处理器CP0 所负责的主要工作如下。

配置CPU 工作状态：通过读写一个或一些内部寄存器，改变一些很根本的CPU 特性，如大端表示和小端表示的转换。

高速缓存控制：用来控制、读、写缓存。

异常控制：异常的检测和处理由CP0 中的一些寄存器进行定义和控制。

存储管理单元控制：对系统存储区域进行控制、管理和分配，主要涉及对MMU、TLB 的操作。

其它：集成一些提供额外功能的模块，如时钟、计数器、奇偶校验器等。

CP0 包含32 个寄存器，每个寄存器位数为32 位。操作系统ucore 所涉及的协处理器CP0 寄存器共有11 个，这些寄存器的编号、名称和功能已在下表中列出。为了实现对CP0 的控制功能，我们需要使用MTC0、MFC0 两条协处理器访问指令。在MIPS32 指令集架构中，MTC0 指令实现修改CP0 中的寄存器，MFC0 指令实现读取CP0 中的寄存器，具体格式可参考附录中的ucore 指令格式列表。异常发生时，CPU 中的异常处理模块访问CP0 寄存器，将异常原因、指令地址、访存错误地址、处理器状态等信息写入对应的寄存器。

寄存器号	寄存器名	功能
0	Index	TLB 阵列的入口索引
2	EntryLo0	偶数虚拟页入口地址的低 32 位部分
3	EntryLo1	奇数虚拟页入口地址的低 32 位部分
8	BadVAddr	记录最近一次存储发生异常时的虚拟地址
9	Count	与 Compare 寄存器组成片内计时器，两者相等时发出时钟中断信号
10	EntryHi	TLB 入口地址的高 32 位部分
11	Compare	与 Count 寄存器组成片内计时器，两者相等时发出时钟中断信号
12	Status	处理器状态和控制寄存器，决定 CPU 特权等级和中断使能等
13	Cause	保存最近一次异常原因
14	EPC	保存最近一次异常时的程序计数器
15	Ebase	保存异常处理程序的入口地址

Figure 4: CP0 寄存器功能表（部分）

3.4.1 Count 寄存器

计数用寄存器，计数频率一般与CPU 时钟频率相同，当计数到达32 位无符号数上限时从0 重新开始计数。可读写。

3.4.2 Compare 寄存器

与Count 寄存器一起完成定时中断功能。当Count 寄存器中的数值与Compare 寄存器中的值一样时，产生定时中断。该中断一直保持到有数据被写入Compare 寄存器为止。可读写。

3.4.3 Status 寄存器

用来控制处理器的操作模式、中断使能及诊断状态。其字段如下表所示。可读写。

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
标志名	CU3-CU0				RP	R	RE	0		BEV	TS	SR	NMI	0		
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
标志名	IM7-IM0								R			UM	R	ERL	EXL	IE

Figure 5: Status 寄存器字段表

上表中表示为R 的字段是保留字段，下面介绍其中比较重要的非保留字段。

CU3-CU0：表示协处理器是否可用（Coprocessor Usability），分别控制协处理器CP3 到CP0。由于只有协处理器CP0，故可设置为4’b0001。

BEV：表示是否使用启动异常向量（Bootstrap Exception Vector）。

TS：表示是否关闭TLB（TLB Shutdown）。

SR：表示是否为软重启（Soft Reset），为1 表示重启异常是由软重启造成的。

NMI：表示是否是不可屏蔽中断（Non-Maskable Interrupt），为1 表示重启异常是由不可屏蔽中断造成的。

IM7-IM0：表示是否屏蔽相应的中断。MIPS 处理器可以有8 个中断源，对应IM 字段的8位，其中6 个中断源是处理器外部硬件中断，另外2 个是软件中断。

UM：表示是否为用户模式（User Mode），为1 表示处理器运行在内核模式，为0 表示处理器运行在用户模式。

EXL：表示是否处于异常级（Exception Level），当异常发生时，本字段置1。

IE：表示是否使能中断（Interrupt Enable），这是全局中断使能标志位。为1 表示中断使能，为0 表示中断禁止。

3.4.4 Cause 寄存器

主要记录最近一次异常发生的原因，也控制软件中断请求。其字段如下表所示。除了IP[1:0]、IV 和WP 字段，其余字段都是只读的。

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
标志名	BD	R	CE		DC	PCI	0		IV	WP	0					
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
标志名	IP[7:2]						IP[1:0]		0	ExcCode					0	

Figure 6: Cause 寄存器字段表

上表中表示为R 的字段是保留字段，下面介绍其中比较重要的非保留字段。

BD：当发生异常的指令处于分支延迟槽（Branch DelaySlot）时，该字段置1。

IP[7:2]：中断挂起（Interrupt Pending）字段，相应位用来指明外部硬件中断是否发生。

IP[1:0]：中断挂起字段，对应软件中断。

ExcCode：用来记录发生了哪些异常。操作系统ucore 涉及到的异常种类如下表所示。

ExcCode	助记符	描述
0	Int	中断
1	Mod	TLB 修改异常或保留
2	TLBL	TLB 加载异常、取指异常或保留
3	TLBS	TLB 存储异常或保留
4	AdEL	加载或取指过程中，地址错误异常
5	AdES	存储过程中，地址错误异常
8	Sys	系统调用指令 SYSCALL
10	RI	执行未定义指令引起的异常
11	CpU	协处理器不可用异常
12	Ov	整数溢出异常

Figure 7: ExcCode 的编码及含义表（部分）

3.4.5 EPC 寄存器

异常程序计数器（Exception Program Counter, EPC），用来存储异常返回地址，一般相当于发生异常的指令的地址。如果发生异常的指令位于延迟槽中，则EPC 存储的是前一条转移指令的地址。可读写。

3.5 MMU

MMU 是Memory Management Unit 的缩写，中文名是内存管理单元，它是CPU 中用来管理虚拟存储器、物理存储器的控制模块，同时也负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权，多用户多进程操作系统。我们知道，计算机的虚拟地址空间大小由CPU 位数所决定，对于一个32 位的CPU，地址范围为0-0xFFFFFFFF，总大小为4G。而实际物理地址往往只是虚拟地址空间的子集，因此在具备MMU 的计算机上，虚拟地址将通过MMU 模块转化为物理地址。

MIPS32 架构的MMU 主要实现虚拟内存映射的功能，以读写控制信号和虚拟地址为输入，实现对应物理地址数据的存储和访问。MIPS 标准的内存映射方案如下表所示：

段	虚拟地址（VA）	权限	物理地址（PA）
kuseg	0x00000000 - 0x7FFFFFFF	用户态	TLB
kseg0	0x80000000 - 0x9FFFFFFF	内核态	0x00000000 - 0x1FFFFFFF
kseg1	0xA0000000 - 0xBFFFFFFF	内核态	0x00000000 - 0x1FFFFFFF
kseg2	0xC0000000 - 0xFFFFFFFF	内核态	TLB

Figure 8: MIPS 标准虚拟地址映射方案

在本次实验中，整体的内存映射方案如上表所示，由于开发板上的SRAM 不能覆盖kseg0 和kseg1 段的物理地址空间，因此该段空间实际有效的地址范围是：VA 0x80000000 - 0x800FFFFF(对应PA 0x0000000-0x000FFFFF)。

此外，kseg1 段的部分地址将不被映射到物理内存地址，而是映射到特殊设备。

VA 0xBE000000-0xBEFFFFFF 被映射到Flash。

VA 0xBFC00000-0xBFC00FFF 被映射到ROM。

VA 0xBFD003F8-0xBFD003FC 被映射到串口。

3.6 TLB

旁路快表缓冲（Translation Lookaside Buffer, TLB），简称快表，可以理解为一种地址变换高速缓存。

在上一个部分中，计算机内部虚拟地址与实际物理地址之间的对应关系通常由一种特殊的数据结构——页表（Page Table）进行储存。因此，虚拟内存到物理内存的转换可以通过查表操作进行。由于页表存放在主存中，程序每次访问内存至少需要两次：一次访存获取物理地址，第二次访存才获得数据。提高访存性能的关键在于依靠页表的访问局部性。当一个转换的虚拟页号被使用时，它可能在不久的将来再次被使用到。TLB 就是符合这些要求的一种高速缓存，内存管理硬件使用它来改善虚拟地址到物理地址的转换速度。

CPU 访问虚拟地址时，会首先根据虚拟地址的高20 位在TLB 中查找。如果快表中没有相应的表项，则触发页表缺失异常（TLB miss），需要通过访问内存中的页表计算出相应的物理地址。与此同时，物理地址被存放在一个TLB 表项中，以后对同一线性地址的访问，直接从TLB 表项中获取物理地址即可，称为页表命中（TLB hit）。

结合操作系统ucore 及THINPAD 教学实验平台的实际情况，我们最终确定将TLB 条目数设置为16。虚拟地址的高20 位作为虚页号，用于在TLB 中查找对应的页表项，返回实际物理地址。

3.7 异常及中断处理

MIP32 架构中的异常包括中断（Interrupt）、陷阱（Trap）、系统调用（System Call）以及其它任何可以打断程序正常执行流程的操作。在ucore异常处理一节和CP0 一节的ExcCode 表中，我们已经列举了操作系统ucore 涉及的异常，而下表则进一步按照优先级对MIPS CPU 所需处理异常类型进行了排序。

优先级	异常	描述
1	Reset	硬件复位
2	Soft Reset	发生致命错误后对系统进行软复位
5	NMI	不可屏蔽的中断
7	Interrupt	检测到 8 个中断之一
11	AdEL	取指地址对齐异常
12	TLB Refill	指令 TLB 缺失
13	TLB Invalid	指令 TLB 无效
16	Sys	执行系统调用指令 SYSCALL
16	RI	无效指令
16	Ov	算术操作指令 ADD、ADDI、SUB 运算溢出
19	AdEL	加载数据的地址未对齐
19	AdES	存储数据的地址未对齐
20	TLB Refill	数据 TLB 缺失
21	TLB Invalid	数据 TLB 无效
22	TLB Mod	对不可写的 TLB 进行了写操作

Figure 9: MIPS32 异常类型及优先级表（部分）

检测到异常发生之后，CPU 会执行一系列指令以处理异常。MIPS32 架构的异常处理过程如下：

1. 检测CP0 中Status 寄存器的EXL 字段。如果EXL 为0，将异常原因保存到CP0 协处理器Cause 寄存器的ExcCode 字段。

2. 检查发生异常的指令是否在延迟槽中，如果在则设置EPC 寄存器的值为指令地址减4，Cause 寄存器BD 字段为1；否则设置EPC 寄存器的值为指令地址，Cause 寄存器BD 字段为0。

3. 设置Status 寄存器的EXL 字段为1，表示进入内核态处理异常，禁止中断。

4. 处理器根据异常种类，转移到相应异常处理例程的入口地址，运行异常处理程序。

5. 处理结束，调用异常返回指令ERET 转到异常发生前的状态。ERET 指令会清除Status 寄存器的EXL 字段，并且将EPC 寄存器的值复制回PC 中。

MIPS CPU 有8 个独立的中断位（在Cause 寄存器中），其中6 个为外部硬件中断，2 个为软件中断。具体实现方面，CPU 能够处理的中断类型如下表所示。遇到中断时，CPU 将跳转至通用的异常处理程序入口。其中中断位4表示串口中断，中断位7表示时钟中断。

4 流水线结构

4.1 流水线概述

流水线是指将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令执行速度。大多数现代处理器选择采用流水线方式执行指令。通常，一条MIPS 指令包含如下五个处理步骤:

1. 从指令存储器中读取指令。

2. 指令译码的同时读取寄存器。MIPS 指令格式允许这两个步骤同时执行。

3. 执行操作或计算地址。

4. 从数据存储器中读取操作数。

5. 将结果写回寄存器。

4.2 数据通路

五级流水线结构的设计，意味着任何一个单时钟周期内，CPU 最多会执行5条指令。因此必须把CPU 数据通路划分为5 个部分，每部分用相对应的指令执行阶段来进行命名:

取指阶段（IF）：从指令存储器读出指令，确定下一条指令地址。

译码阶段（ID）：对取出的指令进行译码，从通用寄存器中读出要使用的寄存器值。如果指令中含有立即数，需要将立即数进行符号扩展或无符号扩展；如果指令是跳转指令，需要给出跳转目标指令地址。

执行阶段（EX）：按照译码阶段给出的操作数和运算类型，进行运算并给出结果。如果是访存类指令，还会计算目标地址。

访存阶段（MEM）：如果是访存类指令，那么此阶段会访问数据存储器，否则只是将执行阶段的运算结果向下传递至回写阶段。同时，此阶段若发生异常，需跳转到异常处理例程入口地址。

回写阶段（WB）：将运算结果保存至目标寄存器。

程序执行过程中，指令与数据由上而下地顺序通过五级流水线。通过增加保存中间数据的寄存器，指令执行过程中可以实现对数据通路的共享。然而数据通路中同样存在一些例外。比如，写回阶段需要把结果写回数据通路中间的寄存器堆中；程序计数器的下一个取值，需在自增的PC 和ID 阶段产生的分支目标之间选择。第一个例外可能导致数据冒险，第二个例外可能导致控制冒险。关于流水线冒险，下一节将给出具体说明和解决方案。

4.3 冒险与解决方案

4.3.1 流水线冒险及其类型

流水线在显著提升程序执行性能的同时，也向我们抛出了额外的问题，其中最主要的是流水线冒险。冒险情况的发生，使得指令序列中下一条指令无法按照设计的时钟周期执行，从而降低流水线的性能。流水线冒险通常可分为以下三种类型:

结构冒险：指令执行过程中，由于多条指令对硬件资源的使用产生冲突而导致的冒险。比如，指令和数据共享同一个存储器，在单时钟周期内，访存操作和取指操作会发生存储器访问冲突，导致结构冒险。

数据冒险：流水线内部的几条指令中，一条指令依赖于先前指令的执行结果。比如当前指令的译码阶段需要读取上一条指令即将写回的寄存器值，这与流水线时间顺序相矛盾。

控制冒险：流水线中的分支跳转指令在ID 阶段才能确定目标地址，但取指过程每时钟周期必需进行，这种为确保取得正确指令而产生的延迟称为控制冒险。

4.4 消除结构冒险

对于相隔两条指令的两条指令，前一条指令的访存阶段与后一条指令的取指阶段可能同时访问存储器而产生结构冒险。采用如下解决办法:

流水线暂停：检测到数据相关时，通过重置流水线寄存器的方式，插入一些空指令周期（或者称作“气泡”），暂停流水线运行。

4.5 消除数据冒险

对于相邻或相隔一条指令存在的数据冒险，MIPS CPU 采用如下两种解决方法。

流水线暂停：例如：LW $1, 0($2) 和ORI $3, $1, 0x0020 指令之间存在数据冒险。当LW 指令处于EX 阶段时，插入暂停周期使得ORI 指令一直处于译码阶段，直至LW 指令将结果写回寄存器$1 后，流水线继续运行。

数据旁路：将计算结果从产生处直接送至其他指令需要的地方，以避免流水线暂停。MIPS CPU将EX 或MEM 的结果传递至ID 部分，解决数据冒险。例如：AND $1, $1, $2 和ADDU $3, $1, $2 指令之间存在数据冒险。ADDU 指令读取寄存器时，AND 指令处于EX 阶段，尚未更新寄存器$1 的值。数据旁路将EX 的计算结果直接传递至ID处，通过添加数据多选器，使得流水线正常运作。

4.6 规避控制冒险

分支转移指令或者其它指令修改了PC 的值，导致已经进入流水线的指令无效，从而引发控制冒险。MIPS CPU 采用分支延迟（Branch Delay Slot）来减小控制冒险的代价。

我们规定分支指令后面的指令位置为分支延迟槽，延迟槽内的指令称为“延迟指令”。延迟指令在计算分支目标地址时已经进入流水线，因此它总是被执行，与跳转发生与否没有关系。为保证延迟槽中只有一条指令，分支判断需要在译码阶段完成，MIPS CPU 会通过额外的计算单元实现这一功能。很显然，并非所有的指令都可以放入延迟槽。部分编译器能够对指令顺序进行调整，在结果不变的情况下使用延迟指令来优化性能。但大多数情况下，延迟指令由程序员依据实际情况设置。

需要注意，如果延迟指令导致异常，那么进入异常处理程序之前，CP0 协处理器的EPC 寄存器应保存之前的分支跳转指令地址，而不是该延迟指令地址。

5 存储器和外围设备

5.1 SRAM

SRAM（Static Random Access Memory），即静态随机存取存储器，是一种具有静止存取功能的内存，不需要刷新电路就可以保存内部数据。SRAM 的优点是速度快，不需要周期性刷新内存数据；缺点是集成度低，掉电后数据会丢失，相同容量下的体积和功耗较大，价格较高。因此SRAM 一般少量用于关键性系统中，以提高运行效率。

THINPAD 教学计算机设置了四片256×1024×16b 的SRAM 作为内存，每两片组合在一起成为一个32位地址线的4MB 内存。其访问过程可通过设计状态机和内存读写逻辑而实现。

5.2 Flash

Flash 存储器又称为闪存，它不仅具备电子可擦除和可编程的功能，还能够快速读取数据。Flash不像SRAM，保存的数据断电后也不会丢失。由于这些特点，它在便携式设备中被大量使用，如手机、平板电脑、U 盘和SD 卡等。

本实验的具体需求中，Flash 存储器被当做计算机的“硬盘”来使用，除存储操作系统ucore 的核心代码外，还可以存储用户程序和数据。操作系统启动之前，写入FPGA 的BootLoader 程序将会把Flash内部的操作系统代码转移至SRAM。因此涉及的操作只有读操作。

5.3 串行接口

THINPAD 教学计算机的FPGA 芯片有两种使用串口的模式，分别是与内存共用总线的CPLD芯片控制模式和直连模式。我们将使用直连模式，自己编写控制代码。串口的功能需求具体表现在：ucore运行时，通过串口向计算机发送调试命令，并接收计算机传回的调试信息，如寄存器状态、内存数据、运行提示信息等。

5.4 DVI图像输出

本实验使用的图像输出接口为每个像素点提供8位的数据宽度，r分量和g分量各占3位，b分量占2位。功能需求为：

固定VGA 显示器的分辨率和屏幕刷新率，输出正确的行场像素数据和同步信号，保持图像稳定。

支持将串口输出的调试信息通过图像输出，并支持全部95 个可显示的ASCII字符。

支持屏幕平滑滚动。

支持通过用户程序切换图像输出的显示模式，使其能从Flash中读取图片并显示。

32位mips设计文档

金涛 何纬捷 李品农

January 2018

Contents

1 文档说明

本文档是404NotFound组32位MIPS32CPU的设计文档，说明了每个模块的作用和设计约束、设计规范等细节。每个模块的描述按三部分进行：

简介：介绍模块的功能和设计思路。

接口和重要的信号：介绍模块的输入输出接口和内部的重要信号。

设计细节和规范：该模块设计时需要注意的细节和需要满足的规范，一些容易产生问题的地方。

2 各模块设计

2.1 Reg

简介 ：模块对应的文件：regfile.vhd
功能：内部有32个MIPS32寄存器。 写寄存器：在时钟上升沿，由信号rst,we,waddr控制写入某个寄存器或者保持不变。 读寄存器：可以有两个寄存器被读取，由组合逻辑实现，信号rst ， raddr1 /raddr2 , re1 /re2 控制读取与否和地址。

regfile

	信号名	来源/去往	意义
输入	clk	电路板	时钟信号
	rst	电路板	重置信号
	we	ID	写寄存器使能
	waddr	ID	写寄存器地址
	wdata	ID	写寄存器数据
	re1	ID	读寄存器1使能
	raddr1	ID	读寄存器1地址
	re2	ID	读寄存器2使能
	raddr2	ID	读寄存器2地址
输出	rdata1	ID	读寄存器1数据
	rdata2	ID	读寄存器2数据

Table 1: regfile的模块说明

设计细节 若写寄存器恰好等于读寄存器的序号，读取的数据应该为写信号准备好，但还未写入寄存器堆的数据。

2.2 HILO

简介 ：模块对应的文件：hilo_reg.vhd
含有两个长度为32位的寄存器，即HI寄存器和LO寄存器。 在时钟上升沿，由rst,we控制是否修改HI,LO寄存器的值。

hilo_reg

	信号名	来源/去往	意义
输入	clk	电路板	时钟信号
	rst	电路板	重置信号，高电平生效
	we	MEM_WB	写使能
	hi_i	MEM_WB	写HI寄存器的值
	lo_i	MEM_WB	写LO寄存器的值
输出	hi_o	EX	读HI寄存器的值
	lo_o	EX	读LO寄存器的值

Table 2: hilo_reg的模块说明

2.3 PC

简介 ：模块对应的文件：pc_reg.vhd
功能：管理指令地址在每个时钟上升沿的变化。在cpu启动时和rst之后的第一个时钟上升沿设置为0xBFC00000，即ROM对应的地址。在没有“暂停”、“跳转”、“清空”的控制信号时，每个上升沿+4。

pc_reg

	信号名	来源/去往	意义
输入	clk	电路板	时钟信号
	rst	电路板	重置信号，高电平生效
	stall	CTRL	暂停信号
	branch_flag_i	ID	是否跳转
	branch_target_address_i	ID	跳转地址
	flush	CTRL	清空信号
	new_pc	CTRL	新的PC值
输出	pc	IF_ID MICS	取指地址
	ce	MICS	取指使能

Table 3: pc_reg的模块说明

设计细节 pc模块要注意的是跳转、暂停、清空之间的处理。 其中一种情况是：跳转指令的下一条指令，延迟槽指令触发了异常。按照MIPS32的规定，延迟槽异常返回地址是上一条指令的地址。MIPS的分支跳转指令流是：分支跳转指令 -> 延时槽指令->目标跳转地址的指令，在中间操作插入了延时槽指令。如果PC在延时槽地址中断后，中断返回时返回延时槽指令地址的话，重新执行的指令流为：延时槽指令 -> (延时槽指令地址 + 4)地址的指令，没有跳转。这样完全不是原来被打断的指令流，为了恢复原来的指令流需要将延时槽前面的跳转指令重新装入流水线。所以在延时槽中断后返回的地址是前面跳转指令的地址。

2.4 IF_ID

简介 ：模块对应的文件：if_id.vhd

功能：每个时钟上升沿，根据控制信号rst,flush,stall三个的优先级和值，把取指的指令清空或者传递到id译码阶段。

if_id

	信号名	来源/去往	意义
输入	clk	电路板	全局时钟信号
	rst	电路板	全局复位信号
	if_pc	PC	取址阶段指令地址
	if_inst	MMU	取址阶段指令
	stall	CTRL	暂停信号
	flush	MEM	流水线清空使能信号
输出	id_pc	ID	译码阶段指令地址
	id_inst	ID	译码阶段指令

Table 4: if_id的模块说明

设计细节 三个控制信号的优先级，rst>flush>stall。

2.5 ID

简介 ：模块对应的文件：id.vhd
功能：流水线的译码阶段。负责识别指令类型和各个参数，读取寄存器，产生流水线的控制信号，处理分支跳转，处理数据通路等。

id

	信号名	来源/去往	意义
输入	rst	电路板	全局复位信号
	pc_i	IF_ID	指令地址
	inst_i	IF_ID	指令内容
	reg1_data_i	REGFILE	寄存器堆读端口1所读数据
	reg2_data_i	REGFILE	寄存器堆读端口2所读数据
	ex_wreg_i	EX	旁路信号，指示当前处于EX阶段指令是否写回寄存器堆
	ex_wd_i	EX	旁路信号，指示当前处于EX阶段指令写回的寄存器地址
	ex_wdata_i	EX	旁路信号，指示当前处于EX阶段指令写回的数据
	ex_aluop_i	EX	旁路信号，指示当前处于EX阶段指令进行的运算子类型
	is_in_delayslot_i	ID_EX	当前指令是否位于延迟槽中
	mem_wreg_i	MEM	旁路信号， 指示当前处于MEM阶段指令是否写回寄存器堆
	mem_wd_i	MEM	旁路信号， 指示当前处于MEM阶段指令写回寄存器地址
	mem_wdata_i	MEM	旁路信号， 指示当前处于MEM阶段指令写回的数据
输出	aluop_o	ID_EX	ALU要进行的运算子类型
	alusel_o	ID_EX	ALU要进行的运算类型
	wd_o	ID_EX	要写入的寄存器地址
	wreg_o	ID_EX	是否要写入寄存器
	stallreq	CTRL	暂停请求
	branch_flag_o	PC	是否发生转移
	branch_target_address_o	PC	转移目标地址
	is_in_delayslot_o	ID_EX	当前指令是否处于延迟槽中
	link_addr_o	ID_EX	转移指令要保存的返回地址
	next_inst_in_delayslot_o	ID_EX	下一条指令是否在延迟槽中
	excepttype_o	ID_EX	译码阶段产生的异常类型
	current_inst_address_o	ID_EX	当前指令地址
	inst_o	ID_EX	当前指令内容

Table 5: id的模块说明

设计细节

2.5.1 分支跳转

我们采用了延迟槽技术，使得跳转指令的下一条指令无论跳转与否必定执行，而跳转指令需要译码阶段判断，此时需要发送是否在延迟槽的信号至下一条指令，即延迟槽指令的取指阶段。

2.5.2 数据旁路EX->ID

具有代表性的例子即：ori指令紧接and指令。 ori的写入寄存器是and读取的寄存器。 ori在EX阶段计算出了写入的数据，在MEM_WB阶段写回寄存器。and在ID阶段读取寄存器，此时ori在EX阶段。 需要and的ID阶段取的数据恰好在EX阶段产生，没有传到寄存器堆里。 由此增加数据通路EX->ID，使得ID阶段可以获得EX阶段计算的寄存器值。 避免了ori指令需要插2个气泡的损失。

2.5.3 数据冒险MEM->ID

典型例子是lw,and。lw在MEM阶段获得写入寄存器的值，而and在ID阶段需要读寄存器的值。 若不插气泡，and在ID阶段时，lw在EX阶段，无法访存获得and需要读的值。因此我们插入一个气泡，同时增加MEM->ID的通路。 数据通路避免了插2个气泡的损失。

2.5.4 异常检测

ID还负责记录译码阶段和之前捕获的所有异常。我们在MEM阶段中统一处理，因此捕获异常后暂时不处理，避免异常处理顺序被打乱。

2.6 ID_EX

简介 ：模块对应的文件：id_ex.vhd
功能：每个时钟上升沿，根据控制信号rst,flush,stall三个的优先级和值，把译码阶段ALU操作数、ALU运算类型、寄存器的值、解析出的访存、当前指令地址、异常缓冲等传递给EX阶段。

id_ex

	信号名	来源/去往	意义
输入	clk	电路板	全局时钟信号
	rst	电路板	全局复位信号
	id_alusel	ID	译码阶段指令要进行的运算类型
	id_aluop	ID	译码阶段指令要进行的运算子类型
	id_reg1	ID	译码阶段指令进行的运算的源操作数1
	id_reg2	ID	译码阶段指令进行的运算的源操作数2
	id_wd	ID	译码阶段指令要写入的寄存器地址
	id_wreg	ID	译码阶段指令是否要写入寄存器
	stall	CTRL	暂停信号
	id_link_address	ID	转移指令保存的返回地址
	id_is_in_delayslot	ID	指令是否在延迟槽中
	next_inst_in_delayslot_i	ID	下一条指令是否在延迟槽中
	flush	CTRL	流水线清空使能
	id_current_inst_address	ID	译码阶段指令地址
	id_excepttype	ID	译码阶段指令异常类型
	id_inst	ID	译码阶段指令内容
输出	ex_link_address	EX	同输入
	ex_is_in_delayslot	EX	同输入
	is_in_delayslot_o	EX	同输入
	ex_current_inst_address	EX	同输入
	ex_excepttype	EX	同输入
	ex_alusel	EX	执行阶段指令要进行的运算类型
	ex_aluop	EX	执行阶段指令要进行的运算子类型
	ex_reg1	EX	执行阶段指令要进行的运算的源操作数1
	ex_reg2	EX	执行阶段指令要进行的运算的源操作数2
	ex_wd	EX	执行阶段指令要写入寄存器的地址
	ex_inst	EX	执行阶段指令内容
	ex_wreg	EX	执行阶段指令是否要写入寄存器

Table 6: id_ex的模块说明

设计细节　由于插入气泡实际是传递到EX阶段的信号延迟，所以ID_EX阶段要控制是否发送信号。 优先级处理，

2.6.1　流水线清空和暂停

如果流水线清空和暂停同时发生，因为流水线暂停解决的是结构冲突，因此流水线暂停优先级高于清空信号。

2.6.2　流水线清空和气泡

清空和气泡同时发生，因为异常在MEM捕获，当前处于IF,ID,EX阶段的指令不应该得到执行，所以清空优先级大于气泡。

2.7　EX

简介　：模块对应的文件：ex.vhd
功能：负责算术、逻辑运算，是ALU的实现模块。它计算出要写回到寄存器堆、CP0寄存器堆、HILO寄存器、mem地址的值，传递给下一阶段。

ex

	信号名	来源/去往	意义
输入	rst	电路板	全局复位信号
	alusel_i	ID_EX	执行阶段要进行运算类型
	aluop_i	ID_EX	执行阶段要进行的运算子类型
	reg1_i	ID_EX	参与运算的源操作数1
	reg2_i	ID_EX	参与运算的源操作数2
	wd_i	ID_EX	指令要写入的寄存器地址
	wreg_i	ID_EX	指令是否要写入寄存器
	inst_i	ID_EX	指令内容
	hi_i	HILO_REG	hi寄存器值
	lo_i	HILO_REG	lo寄存器值
	wb_hi_i	WB	写回阶段指令写hi寄存器的值
	wb_lo_i	WB	写回阶段指令写lo寄存器的值
	wb_whilo_i	WB	写回阶段指令是否要写hilo寄存器
	link_address_i	ID_EX	跳转指令保存的返回地址
	is_in_delayslot_i	ID_EX	当前指令是否在延迟槽中
	current_inst_address_i	ID_EX	下一条指令是否在延迟槽中
	excepttype_i	ID_EX	之前收集的异常信息
	mem_cp0_reg_we	MEM	访存阶段指令是否要写cp0寄存器
	mem_cp0_reg_write_addr	MEM	访存阶段指令写cp0寄存器地址
	mem_cp0_reg_data	MEM	访存阶段指令写cp0寄存器数据
	wb_cp0_reg_we	WB	写回阶段指令是否要写cp0寄存器
	wb_cp0_reg_write_addr	WB	写回阶段指令写cp0寄存器地址
	wb_cp0_reg_data	WB	写回阶段指令写cp0寄存器数据
	cp0_reg_data_i	CP0_REG	读cp0寄存器的值
	mem_hi_i	MEM	访存阶段指令写hi寄存器的值
	mem_lo_i	MEM	访存阶段指令写lo寄存器的值
	mem_whilo_i	MEM	访存阶段指令是否要写hilo寄存器

Table 7: ex的模块说明（1）

ex

	信号名	来源/去往	意义
输出	wd_o	EX_MEM	执行阶段写寄存器地址
	wreg_o	EX_MEM	执行阶段是否要写寄存器
	wdata_o	EX_MEM	执行阶段写的数据
	aluop_o	EX_MEM	执行阶段指令进行的运算子类型
	mem_addr_o	EX_MEM	访存指令对应地址
	reg2_o	EX_MEM	访存指令操作的寄存器原始值
	stallreq	CTRL	暂停请求
	current_inst_address_o	EX_MEM	访存阶段指令地址
	excepttype_o	EX_MEM	执行及之前收集的异常信息
	is_in_delayslot_o	EX_MEM	访存阶段指令是否在延迟槽中
	cp0_reg_read_addr_o	EX_MEM	执行阶段指令读cp0寄存器地址
	cp0_reg_we_o	EX_MEM	执行阶段指令是否要写cp0寄存器
	cp0_reg_write_addr_o	EX_MEM	执行阶段指令写cp0寄存器地址
	cp0_reg_data_o	EX_MEM	读cp0寄存器的值
	hi_o	EX_MEM	访存阶段指令写hi寄存器的值
	lo_o	EX_MEM	访存阶段指令写lo寄存器的值
	whilo_o	EX_MEM	访存阶段指令是否要写hilo寄存器

Table 8: ex的模块说明（2）

设计细节　本阶段使用组合逻辑电路。实现了ALU运算和数据传递的功能。

2.7.1　ALU运算

ALUOP见下图

类别	AluOp	功能
逻辑指令	EXE_OR_OP	逻辑或
逻辑指令	EXE_AND_OP	逻辑与
逻辑指令	EXE_NOR_OP	逻辑或非
逻辑指令	EXE_XOR_OP	逻辑异或
移位操作	EXE_SLL_OP	逻辑左移
移位操作	EXE_SRL_OP	逻辑右移
移位操作	EXE_SRA_OP	算术右移
算术操作	EXE_SLT_OP	符号数比较
算术操作	EXE_SLTU_OP	无符号比较
算术操作	EXE_ADD_OP	溢出检测的加法
算术操作	EXE_ADDU_OP	无溢出检测的加法
算术操作	EXE_ADDI_OP	溢出检测的立即数加
算术操作	EXE_ADDIU_OP	无溢出检测的立即数加
算术操作	EXE_SUB_OP	溢出检测的减法
算术操作	EXE_SUBU_OP	无溢出检测的减法
算术操作	EXE_MUL_OP	乘法
算术操作	EXE_MULT_OP	乘，放于HILO寄存器
其他	EXE_MFHI_OP	取出HI寄存器的值
其他	EXE_MFLO_OP	取出LO寄存器的值
其他	EXE_MFC0_OP	取出CP0寄存器
其他	EXE_MTC0_OP	写入CP0寄存器

Figure 1: aluop类型

2.7.2 其他

当不需要进行ALU运算时，且rst=‘0’时，将ID译码得出的数据传递给MEM阶段。

2.8 EX_MEM

简介 ：模块对应的文件：ex_mem.vhd
功能：在时钟上升沿，将EX的信号传递给MEM模块或其他部分。

ex_mem

	信号名	来源/去往	意义
输入	rst	电路板	全局复位信号
	clk	电路板	全局时钟信号
	ex_wd	EX	执行阶段指令写入寄存器地址
	ex_wreg	EX	执行阶段指令是否要写入寄存器
	ex_wdata	EX	执行阶段指令写入寄存器数据
	stall	CTRL	暂停信号
	flush	CTRL	流水线清空使能
	ex_excepttype	EX	执行阶段产生的异常类型
	ex_is_in_delayslot	EX	执行阶段指令是否在延迟槽中
	ex_current_inst_address	EX	执行阶段指令地址
	ex_hi	EX	执行阶段指令要写入hi的数据
	ex_lo	EX	执行阶段指令要写入hi的数据
	ex_whilo	EX	访存阶段指令是否要写hilo寄存器
	ex_aluop	EX	执行阶段指令的运算子类型
	ex_mem_addr	EX	执行阶段访存指令的目标地址
	ex_reg2	EX	执行阶段指令操作的寄存器初始值
	ex_cp0_reg_we	EX	执行阶段指令是否要写cp0寄存器
	ex_cp0_reg_write_addr	EX	执行阶段指令写cp0寄存器地址
	ex_cp0_reg_data	EX	执行阶段指令写cp0寄存器数据

Table 9: ex_mem的模块说明（1）

ex_mem

	信号名	来源/去往	意义
输出	mem_wd	MEM	访存阶段指令写入寄存器地址
	mem_wreg	MEM	访存阶段指令是否要写入寄存器
	mem_wdata	MEM	访存阶段指令写入寄存器数据
	mem_excepttype	MEM	访存阶段前收集的异常信息
	mem_is_in_delayslot	MEM	访存阶段指令是否在延迟槽中
	mem_current_inst_address	MEM	访存阶段指令地址
	mem_aluop	MEM	访存阶段指令的运算子类型
	mem_mem_addr	MEM	访存阶段指令的目标地址
	mem_reg2	MEM	访存阶段指令操作寄存器的初始值
	mem_hi	MEM	访存阶段指令要写入hi的数据
	mem_lo	MEM	访存阶段指令要写入lo的数据
	mem_whilo	MEM	访存阶段指令是否要写hilo寄存器
	mem_cp0_reg_we	MEM	访存阶段指令是否要写cp0寄存器
	mem_cp0_reg_write_addr	MEM	访存阶段指令写cp0寄存器地址
	mem_cp0_reg_data	MEM	访存阶段指令写cp0寄存器数据

Table 10: ex_mem的模块说明（2）

2.9 MEM

简介 ：模块对应的文件：mem.vhd
功能：该阶段负责访存的操作、保存的异常的处理和信号传递。

mem

	信号名	来源/去往	意义
输入	rst	电路板	重置信号，高电平生效
	wd_i	EX_MEM	写寄存器使能
	wreg_i	EX_MEM	写寄存器号
	wdata_i	EX_MEM	写寄存器数据
	aluop_i	EX_MEM	指令类型
	mem_addr_i	EX_MEM	访存地址
	reg2_i	EX_MEM	寄存器值
	mem_data_i	EX_MEM	写内存数据
	hi_i	EX_MEM	写HI寄存器数据
	lo_i	EX_MEM	写LO寄存器数据
	whilo_i	EX_MEM	写HILO使能
	excepttype_i	EX_MEM	收集的异常
	is_in_delayslot_i	EX_MEM	在延迟槽信号
	current_inst_address_i	EX_MEM	当前指令地址
	excepttype_mc_i	MMU	MMU收集的异常
	cp0_status_i	CP0	STATUS寄存器
	cp0_cause_i	CP0	CAUSE寄存器
	cp0_epc_i	CP0	EPC寄存器
	wb_cp0_reg_we	MEM_WB	写CP0使能
	wb_cp0_reg_write_addr	MEM_WB	写CP0地址
	wb_cp0_reg_data	MEM_WB	写CP0数据
	cp0_reg_we_i	EX_MEM	数据通路-写CP0使能
	cp0_reg_write_addr_i	EX_MEM	数据通路-写CP0地址
	cp0_reg_data_i	EX_MEM	数据通路-写CP0数据
输出	tlb_write	MMU	是否TLBWI指令
	mem_addr_o	MMU	访存地址
	mem_we_o	MMU	写使能
	mem_sel_o	MMU	32位4个字节的使能
	mem_ce_o	MMU	访存使能
	mem_data_o	MMU	写数据
	wd_o	MEM_WB	写寄存器使能
	wreg_o	MEM_WB	写寄存器号
	wdata_o	MEM_WB	写寄存器数据
	hi_o	MEM_WB	写HI寄存器数据
	lo_o	MEM_WB	写LO寄存器数据
	whilo_o	MEM_WB	写HILO使能
	excepttype_o	CP0|CTRL	收集到的异常
	cp0_epc_o	CTRL	EPC值
	is_in_delayslot_o	CP0	是延迟槽指令
	current_inst_address_o	CP0	当前指令地址
	cp0_reg_we_o	CP0	写CP0使能
	cp0_reg_write_addr_o	CP0	写CP0地址
	cp0_reg_data_o	CP0	写CP0数据
	stallreq	CTRL|MICS	MEM暂停信号

Table 11: mem的模块说明

设计细节

2.9.1 访存

MEM模块把访存的控制信号传递给MMU模块，接受来自MMU的输出信号，再传递给下个模块。

2.9.2 异常

MEM模块负责将收集到的异常统一递交给控制模块CTRL。按照优先级处理的异常分别是：中断，MMU的异常（TLBMISS），系统调用，无效指令，陷入，溢出，ERET返回。

2.10 MEM_WB

简介 ：模块对应的文件：mem_wb.vhd
功能：将值写入寄存器。

mem_wb

	信号名	来源/去往	意义
输入	clk	电路板	时钟信号
	rst	电路板	重置信号，高电平生效
	mem_wd	MEM	写寄存器编号
	mem_wreg	MEM	写寄存器号
	mem_wdata	MEM	写数据
	stall	CTRL	暂停信号
	flush	CTRL	清空信号
	mem_hi	MEM	写HI寄存器数据
	mem_lo	MEM	写LO寄存器数据
	mem_whilo	MEM	写HILO控制信号
	mem_cp0_reg_we	MEM	写CP0使能
	mem_cp0_reg_write_addr	MEM	写CP0的地址
	mem_cp0_reg_data	MEM	写CP0的数据
输出	wb_wd	MEM_WB	写寄存器编号
	wb_wreg	REG	写寄存器使能
	wb_wdata	REG	写寄存器数据
	wb_hi	EX|HILO	写HI寄存器数据
	wb_lo	EX|HILO	写LO寄存器数据
	wb_whilo	EX|HILO	写HILO使能
	wb_cp0_reg_we	EX|MEM	写CP0使能
	wb_cp0_reg_write_addr	EX|MEM	写CP0寄存器编号
	wb_cp0_reg_data	EX|MEM	写CP0数据

Table 12: mem_wb的模块说明

设计细节　需要写的寄存器包括：32个基本寄存器、HILO寄存器、CP0寄存器。 需要处理的冲突：flush,stall。

2.11 MMU

简介　：模块对应的文件：mmu.vhd
功能：负责地址映射的最重要的模块，连接了MEM和外设，用一条数据总线传输外设的数据。同时负责了TLBMISS异常的传输。

mmu

	信号名	来源/去往	意义
输入	rst	电路板	重置信号，高电平生效
	writetlb_i	MEM	是否是TLBWI指令，一位
	mem_sel_i	MEM	32位对应的4个字节的使能信号
	we_i	MEM	写使能
	ce_i	MEM	访存使能
	wdata_i	MEM	写数据
	vAddr_i	MEM|PC	虚拟地址
	cp0_cause_i	CP0	cause寄存器
	cp0_status_i	CP0	status寄存器
	cp0_entrylo0_i	CP0	lo0寄存器
	cp0_entrylo1_i	CP0	lo1寄存器
	cp0_entryhi_i	CP0	hi寄存器
	cp0_index_i	CP0	index寄存器
	ram_read_data	SRAM	从sram读的数据
	flash_data	FLASH	从flash读的数据
	transmmit_busy	SERIAL	串口是否在发送
	receive_data_ready	SERIAL	串口是否可接受
	receive_data	SERIAL	从串口接收的数据
	rom_data_i	ROM	从rom接收的数据
输出	tlbBadAddr	MEM	TLB异常访问地址
	invalid_index	MEM	无效的TLB表项号
	data_o	MEM	读取的数据
	ram_en	SRAM	SRAM芯片使能
	ram_wr	SRAM	SRAM读写控制
	ram_be	SRAM	SRAM32位中4个8位的使能
	ram_addr	SRAM	SRAM读写地址
	ram_write_data	SRAM	SRAM写的数据
	exceptiontype_o	MEM	MMU产生的异常信息
	flash_ce	FLASH	FLASH使能
	flash_addr	FLASH	FLASH读地址
	pauseforflash_o	MEM	FLASH暂停流水线
	transmmit_start	SERIAL	串口发送使能
	transmmit_data	SERIAL	串口发送的数据
	receive_data_read_en	SERIAL	串口接收的使能
	rom_addr_o	ROM	ROM的读地址
	dispmem_wen	显存	显存写使能
	dispmem_wdata	显存	显存写入数据

Table 13: mmu的模块说明

设计细节

2.11.1 地址映射

段	虚拟地址（VA）	权限	物理地址（PA）
kuseg	0x00000000 - 0x7FFFFFFF	用户态	TLB
kseg0	0x80000000 - 0x9FFFFFFF	内核态	0x00000000 - 0x1FFFFFFF
kseg1	0xA0000000 - 0xBFFFFFFF	内核态	0x00000000 - 0x1FFFFFFF
kseg2	0xC0000000 - 0xFFFFFFFF	内核态	TLB

Figure 2: 地址映射关系

2.11.2 TLB设计

```
---tlb---
CONSTANT TlbSum : integer := 16;
SUBTYPE TlbItem is integer range 63 downto 0;
CONSTANT TLB_VALID: INTEGER := 63;
SUBTYPE TLB_VPN2 IS INTEGER RANGE 62 DOWNTO 44;
CONSTANT TLB_VALID_1:INTEGER := 43;
CONSTANT TLB_DIRTY_1:INTEGER:=42;
SUBTYPE TLB_PADDR_1 IS INTEGER RANGE 41 DOWNTO 22;
CONSTANT TLB_VALID_2:INTEGER := 21;
CONSTANT TLB_DIRTY_2:INTEGER:=20;
SUBTYPE TLB_PADDR_2 IS INTEGER RANGE 19 DOWNTO 0;
SUBTYPE ASID IS INTEGER RANGE 11 DOWNTO 0;
```

Figure 3: tlb信息

TLB设置了16项，每项64位。
对于一项：63位是该项的有效位；62 downto 44是该项的VPN2；43位是偶数页的有效位；42位是偶数页的脏位；41 downto 22是偶数页的PPN；21位是奇数页的有效位；20位是奇数页的脏位；19 downto 0是奇数页的PPN。

2.12 TLB

简介 ：模块对应的文件：tlb.vhd
功能：对应一个TLB项。实现了输入VADDR输出匹配信号和匹配的物理地址的功能。

tlb

	信号名	来源/去往	意义
输入	Tlbin	MMU	TLB项的值
	vAddr	MMU	需要映射的虚拟地址
输出	pAddr	MMU	映射的物理地址
	valid	MMU	是否匹配到有效的物理页
	match	MMU	虚页号是否匹配

Table 14: tlb的模块说明

设计细节 具体定义见MMU的细节。

2.13 CTRL

简介 ：模块对应的文件：ctrl.vhd 所有流水线的暂停、清空、异常、异常跳转地址都由该模块接收和管理。

ctrl

	信号名	来源/去往	意义
输入	rst	电路板	重置信号，高电平生效
	stallreq_from_id	ID	ID的暂停信号
	stallreq_from_ex	EX	EX的暂停信号
	stallreq_from_mem	MEM	MEM的暂停信号
	ebase_i	CP0	EBASE寄存器的值
	excepttype_i	MEM	MEM收集的异常
	cp0_epc_i	MEM	EPC的值
输出	new_pc	PC	新的PC值
	flush	*	清空信号
	stall	*	暂停信号

Table 15: ctrl的模块说明

*:PC,IF_ID,ID_EX,EX_MEM,MEM_WB

2.14 CP0

简介 ：模块对应的文件：cp0_reg.vhd
功能：管理CP0寄存器堆的输入和输出。

cp0_reg

	信号名	来源/去往	意义
输入	rst	电路板	重置信号，高电平生效
	clk	电路板	时钟信号
	raddr_i	ID	读CP0的地址
	int_i	SOPC	外部中断
	we_i	MEM_WB	写使能
	waddr_i	MEM_WB	写地址
	wdata_i	MEM_WB	写数据
	tlb_addr	MMU	异常TLB的虚地址
	tlb_index	MMU	异常TLB的项号
	excepttype_i	MEM	MEM阶段收集的异常
	current_inst_addr_i	MEM	当前指令地址
	is_in_delayslot_i	MEM	是否延迟槽指令
输出	index_o	MMU	当前INDEX寄存器值
	entrylo0_o	MMU	当前LO0寄存器值
	entrylo1_o	MMU	当前LO1寄存器值
	vgaflash_o	DVI	当前控制DVI显示模式的寄存器值
	badvaddr_o	–	当前BADVADDR寄存器值
	data_o	EX	读取的CP0寄存器值
	count_o	–	当前COUNT寄存器值
	entryhi_o	MMU	当前HI寄存器值
	compare_o	–	当前COMPARE寄存器值
	status_o	MEM|MMU	当前STATUS寄存器值
	cause_o	MEM|MMU	当前CAUSE寄存器值
	epc_o	MEM	当前EPC寄存器值
	ebase_o	CTRL	当前EBASE寄存器值
	timer_int_o	MICS	时钟中断信号

Table 16: cp0_reg的模块说明

设计细节 除了基本的读写CP0功能，还要在异常发生时，负责修改EPC和CAUSE、BADADDR等寄存器的值。

2.15 Flash

简介 ：模块对应的文件：flash.vhd
功能：是CPU 与Flash 芯片的交互控制模块。它接受来自MMU 的访存操作，并按照要求访问Flash 得到数据。由于MIPS CPU 在设计时不涉及Flash 的写和擦除操作，因此FlashControl 只需实现读Flash 操作即可。计算机开始运行时，ROM 存储器内嵌的Bootloader 程序将指挥CPU 将Flash 中的操作系统ucore 转移至SRAM 内，完成启动工作。

flash

	信号名	来源/去往	意义
输入	flash_read	MMU|DVI	读Flash使能
	flash_read_addr	MMU|DVI	读Flash地址
	flash_data	CPU	来自Flash芯片的数据
输出	flash_read_data	MMU|DVI	送往MMU和DVI控制的Flash数据
	flash_a	CPU	直接控制Flash芯片的信号
	flash_rp_n	CPU	直接控制Flash芯片的信号
	flash_vpen	CPU	直接控制Flash芯片的信号
	flash_oe_n	CPU	直接控制Flash芯片的信号
	flash_ce_n	CPU	直接控制Flash芯片的信号
	flash_byte_n	CPU	直接控制Flash芯片的信号
	flash_we_n	CPU	直接控制Flash芯片的信号

Table 17: flash的模块说明

设计细节 FLASH芯片主要参数如下：
大小：4096*16bit，共8MB；
读周期：40ns左右；
写周期：40ns左右；

从以上参数细节可知，Flash 芯片的控制模块地址线为23 位，总大小为8MB。需要注意，Flash芯片内部的编址方式为按字节编址，与SRAM 按字编址的方式存在明显区别，但其地址线为16 位。32 位MIPS CPU 不支持非对齐LW 或SW 指令，也不支持LH 和SH 这样的半字存取指令。为方便起见，我们将Flash 虚拟化为一块16MB 存储芯片，包含8MB 的实际数据与8MB 的零空间，如下图所示：

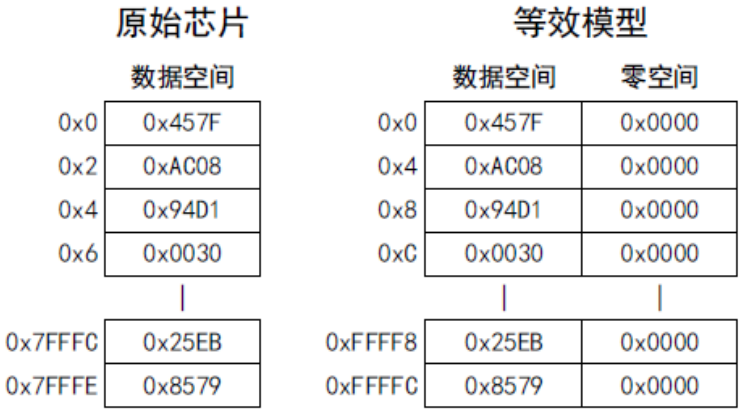


Figure 4: Flash 芯片等效示意图

从0 号地址开始，数据与零空间按照16bit 对齐，交替排列。结合Flash 芯片按字节编址的特性，我们将16bit 数据的存取转化为32bit 数据操作。其中，这32bit 数据的高16 位只起到占位作用，任何时候读取值为0，写入值则被硬件自动忽略。

模块内部使用组合逻辑控制使能信号及数据总线。值得一提的是，在单独测试时，Flash控制模块能在25MHz下正确读取数据。但接入流水线后，由于数据通路延迟的原因，使得实际的Flash读取时间被压缩。因此我们最终将时钟频率调整至12MHz。但我们相信时钟频率还有很大的优化空间，如可以通过流水线暂停的方式使得频率提升1倍。

2.16 Serial

简介 ：模块对应的文件：uart.vhd
功能：是CPU 与串行接口（简称串口）进行交互的模块，属于底层控制模块之一。串口在整个CPU 设计和运行中的作用至关重要，它是CPU 与外界仅有的通信方式。MIPS CPU 的串口模块内置了UART（Universal Asynchronous Receiver Transmitter，通用异步收发传输器），并控制其工作。

uart

	信号名	来源/去往	意义
输入	clk	电路板	CPU主时钟信号
	rst	电路板	复位信号
	transmmit_start	MMU	串口发送使能
	transmmit_data	MMU	串口发送数据
	receive_data_read_en	MMU	串口读取使能
	clk_uart_in	电路板	串口工作时钟
	rxd	电路板	直连串口数据接收线
输出	transmmit_busy	MMU	指示串口是否能发送数据
	receive_data_ready	MMU	串口中断信号
	receive_data	MMU	串口读取数据
	txd	电路板	直连串口数据发送线

Table 18: uart的模块说明

设计细节 串行接口，顾名思义按照串行方式发送和接收数据。发送数据时，UART 内部的波特率发生器产生时钟信号，每时钟周期发送一个比特位；接收数据时，波特率发生器产生波特率的倍频时钟，对收到的信号进行采样以获取数据。RS-232 串行接口一次可以传输8bit 数据，具体形式如下图。

MIPS CPU 所遵循的串口标准如下：
波特率：115200Baud
时钟频率：11.0592MHz
起始位：1bit
数据位：8bit
终止位：1bit
校验位：1bit

实例化UART 时，将波特率和时钟频率作为模块参数传递给接收器和发送器即可。UART 的具体使用方法和源代码参见fpga4fun 网站，地址是http://fpga4fun.com/SerialInterface.html。

串口控制模块本质属于组合逻辑电路，但这并非意味着它不需要寄存器。理论上讲串口读到数据后应该立即引发中断，但在内核状态下，串口中断被屏蔽，无法得到及时响应。如果这段时间内串口收到大量数据而未被读取，则数据会被不断覆盖丢失。为避免此状况，我们设计了串口缓存机制。

串口缓存由先进先出的循环队列实现，首尾指针各由一个寄存器存储。每当串口读取到数据，就会写入队列中，下一时钟上升沿处尾指针自增1；每当串口有数据模式下的LW 操作，就从队头读取数据，下一时钟（与CPU主频相同）上升沿处头指针自增1。产生串口中断的条件即为缓存队列不为空。

必须注意，任何缓存大小都是有限的，无法避免缓存溢出的情况。综合各种可能因素，我们认为将缓存大小设置为16 字节已经足够安全。

2.17 Sram

简介 ：模块对应的文件：sram.vhd
功能：是CPU 与Sram 物理设备的交互控制模块。它接受来自MMU 的数据和访存操作码，按照SRAM 芯片的时序要求正确读写SRAM，并向MMU 返回数据。

sram

	信号名	来源/去往	意义
输入	clk	电路板	CPU主时钟信号
	rst	电路板	复位信号
	ram_en	MMU	RAM使能信号
	ram_wr	MMU	RAM读写操作码
	ram_be	MMU	写RAM时字节选择信号
	ram_addr	MMU	读写RAM地址
	ram_write_data	MMU	写入RAM的数据
	base_ram_data	CPU	来自BaseRam芯片的数据
	ext_ram_data	CPU	来自ExtRam芯片的数据
输出	ram_read_data	MMU	向MMU传送的RAM数据
	base_ram_addr	CPU	直接控制BaseRam芯片的信号
	base_ram_be_n	CPU	直接控制BaseRam芯片的信号
	base_ram_ce_n	CPU	直接控制BaseRam芯片的信号
	base_ram_oe_n	CPU	直接控制BaseRam芯片的信号
	base_ram_we_n	CPU	直接控制BaseRam芯片的信号
	ext_ram_addr	CPU	直接控制ExtRam芯片的信号
	ext_ram_be_n	CPU	直接控制ExtRam芯片的信号
	ext_ram_ce_n	CPU	直接控制ExtRam芯片的信号
	ext_ram_oe_n	CPU	直接控制ExtRam芯片的信号
	ext_ram_we_n	CPU	直接控制ExtRam芯片的信号

Table 19: sram的模块说明

设计细节　SRAM芯片主要参数如下：
大小：1024*16bit，共2MB；
读周期：8ns - 10ns；
写周期：8ns - 10ns；

为支持32 位字长，教学计算机将两片SRAM 组织为一个存储器，该存储器拥有32 位数据线和20 位地址线，总大小为4MB。实际上THINPAD 共带有两个4MB SRAM 存储器，接口标识为BaseRam 和ExtendRam。这样，操作系统允许使用的内存上限是8MB。

需要注意的是，SRAM 写操作应当考虑写保持时间。也就是说，在写使能恢复后，为确保电路稳定，短时间内不能够立即将数据总线赋值为高阻，否则可能导致写入失败。

因此，实现时模块内部使用组合逻辑，但写使能信号由时钟控制。由于使用的时钟占空比为50%，因此写使能在周期后半部分为拉低状态。下一个上升沿到来时，因为数据通路的延迟，使得数据总线晚于写使能被改变，保证了一定的写保持时间。

如上所述，由周期的后半部分进行写操作，理论上时钟频率可以达到50MHZ。但由于流水线数据通路延迟的原因，实际的写操作时间可能不到半个周期。最终我们发现时钟频率小于等于25MHz时，SRAM控制模块能在流水线中稳定运行。

2.18 DVI

简介 ：模块对应的文件：vga.vhd、dispmem.vhd
功能：是控制CPU 与显示器交互的模块。它与显存模块（dispmem.vhd）和DVI 接口相连。 在终端模式下，模块读取显存中存储的ASCII码序列，通过内置的存储字符图像的ROM，使发往串口的调试信息在屏幕上显示。在图片模式下，读取flash特定地址中存储的图片，并将其显示。

vga

	信号名	来源/去往	意义
输入	clk	电路板	25MHz时钟信号
	ascii	显存	显示字符的ASCII码
	rompixel	字符点阵ROM	字符点阵中某个像素的值
	vga_flash_addr	CP0	显示模式下图片的起始地址
	flash_data	FLASH	图片像素数据
输出	hsync	电路板	DVI水平同步信号
	vsync	电路板	DVI垂直同步信号
	memaddr_x	显存	字符坐标X
	memaddr_y	显存	字符坐标Y
	romaddr	字符点阵ROM	字符点阵地址
	vgapixel	电路板	DVI像素数据
	data_enable	电路板	DVI数据使能
	flash_addr	FLASH	Flash读取地址

Table 20: vga的模块说明

设计细节 DVI主要参数如下：

分辨率：800*600；

像素时钟：25MHz；

单像素大小：8bit（R=3bit，G=3bit，B=2bit）；

dispmem模块用于控制显存读写。需要注意的是，该模块仅用于终端显示模式。在该模式下，显存并不需要存储每个像素的8位像素，而是只需要记录每个区域的字符ASCII码，具体像素值从事先例化的字符点阵ROM中二次读取。这样可以显著减少显存的大小。最终每个字符所占大小为16*16个像素，这样整个屏幕同时最多能显示50*37个字符。

屏幕的平滑滚动实现则由循环队列的方式实现。显存大小虽然是固定的，但表示可显示字符首尾的指针却可以改变。因此当总字符超过显存总大小时，通过移动首尾指针即可实现屏幕滚动的效果。

DVI的另一个显示模式是图片模式，该模式能够显示Flash中存储的图片。模式的切换通过我们自己编写的一个用户程序实现。该用户程序通过我们定制的系统调用改变某个特殊CP0寄存器的值，该寄存器中存储的是Flash中存储图片的起始地址，由此告知DVI控制模块使用何种模式显示（地址为0时为终端模式）。

2.19 MICS

简介 ：模块对应的文件：mics.vhd

功能：将除了外设模块（有rom；无serial,flash,sram,vga）的信号连接在一起。

设计细节 vhdl2008语法支持entity work.yourmodule port map();的格式，减少声明模块的重复。

2.20 SOPC

简介 ：模块对应的文件：mmu_sopc.vhd

功能：将时钟、flash、sram、vga、serial和mics模块连接起来。

测试文档

金涛 何纬捷 李品农

January 2018

1 文档说明

本文档是404NotFound组的Mips32 CPU测试文档，包含了整个开发过程中所有测试手段以及结果。文档结构分为模块测试，指令测试，系统测试，性能测试。

2 模块测试

模块测试主要针对流水线的各个模块以及各个外设的单元测试，包括五级流水线搭建的正常工作（每个模块的实现会随着指令的添加而改变，故用指令测试来验证模块实现的正确性），MMU，TLB等。

2.1 五级流水线初步搭建

2.1.1 测试目标

测试流水线框架能否正常工作。

2.1.2 测试过程

运行几条简单的ori指令，观测仿真波形是否正确。

```
ori $1,$0,0x1100        # $1 = $0 | 0x1100 = 0x1100
ori $2,$0,0x0020        # $2 = $0 | 0x0020 = 0x0020
ori $3,$0,0xff00        # $3 = $0 | 0xff00 = 0xff00
ori $4,$0,0xffff        # $4 = $0 | 0xffff = 0xffff
```

2.1.3 测试结果

测试通过，认为流水线搭建正确。

2.2 MMU模块

2.2.1 功能

虚地址映射到实地址。

2.2.2 测试目标

测试MMU的正确映射关系。

2.2.3 测试过程

对于多个直接映射地址进行测试：flash,rom,sram,serial,vga。 先进行仿真测试，此处由于没有外设的仿真，通过MMU输出的控制信号是否正确的标准检验MMU的映射。进行硬件测试则通过写入指定的值，由汇编代码将该地址的值取回寄存器进行对比检验。

硬件测试:
flash测试：使用flash烧写工具写入magic number。将该地址的值取回寄存器进行检测。
rom测试：使用监控程序的bootloader，若正确执行了bootloader的指令，rom即通过测试。
sram测试：将程序段写入sram中，令cpu从sram的启动地址执行指令，发现能成功启动，sram直接映射有效。
serial：使用32位监控程序，由串口进行控制，检测是否能执行。
vga：是否能显示图片。

2.2.4 测试结果

仿真测试通过。
硬件测试:
flash硬件板子不通过测试，无法读出信息。但是在线板子通过测试。确定是硬件故障，暂时跳过flash的使用。
rom测试：监控程序和ucore均正确启动，rom映射正确。
sram测试：监控程序和ucore均正确运行指令，sram映射正确。
serial：使用串口精灵发现存在乱码。使用32位监控程序最初发现串口问题，后定位错误到python脚本中，硬件串口正常。后在在线板子上启动ucore，发现串口也能正常工作。
vga：在线板子能正常显示图片。问题是最初使用vga频率低于25M，在使用硬件显示器时发现无显示或者屏幕闪烁的问题。后修改VGA频率并加入VGA显存缓冲部分，将VGA频率提到25M以上，正常工作。

2.3 TLB模块

2.3.1 功能

缓存虚页实页映射表。

2.3.2 测试目标

测试TLB的正常读写和映射关系的正确性以及TLB相关异常的正常工作。

2.3.3 测试过程及用例

使用补充TLB功能测例进行仿真和硬件测试。具体测试方式为在TLB表为空时对随机地址写随机数，触发TLBW Miss异常，在异常处理程序中填TLB表项。之后再对另一随机地址进行读取操作，触发TLBR Miss异常，填与之前相同的表项，查看读出的数据与之前写入的是否相同。

```
#include <asm/asm.h>
#include <asm/regdef.h>
#include <cpu.h>
#include <machine.h>
#include <ns16550.h>
#include <asm/context.h>
#include <inst_test.h>

LEAF(n100_TLB_test)
    .set noreorder
    lui  a0, 0x1000
    li  v0, 0x10
###test inst
    li t0, 0x00400004
    li t3, 0x00200004
    li t1, 0x1a2b3c4d
    sw t1, 0(t0)
    lw t2, 0(t3)
    nop
    nop
    bne t1, t2, inst_error
###detect exception
    lui s0, 0x0fff
    beq v0, s0, score
    nop
score:
    addiu s3, s3, 1
###output a0|s3
inst_error:
    or t0, a0, s3
    sw t0, 0(s1)
    jr ra
    nop
END(n100_TLB_test)
```

TLBMiss异常处理程序。

```
TLB_ex:
    li t4, 15
```

```
    mtc0 t4, c0_index
    mfc0 t4, c0_badvaddr
    li t5, 0xffffe000
    and t4, t4, t5
    mtc0 t4, c0_entryhi
    li t4, 0x00000002
    mtc0 t4, c0_entrylo0
    li t4, 0x00000042
    mtc0 t4, c0_entrylo1
    tlbwi
    nop
    b ex_ret
    nop
```

2.3.4 测试结果

仿真测试与硬件测试均通过。

2.4 sram模块

2.4.1 功能

计算机的主存部分。

2.4.2 测试目标

测试sram的正确读写和时序要求。

2.4.3 测试过程及用例

使用25MHz时钟驱动，从零地址开始，对于每一对相邻的地址，读取低地址的数据将其写入高地址，正确的结果应为所有地址均为同一数据。

2.4.4 测试结果

测试通过。

2.5 flash模块

2.5.1 功能

计算机的外存部分，用来存储操作系统。

2.5.2 测试目标

测试flash的正确读入和时序要求。

2.5.3 测试过程及用例

使用12MHz时钟驱动，将flash中的数据（8MB）读出写入sram（两块各4MB）。通过读取sram中的数据检验flash读取正确性。

2.5.4 测试结果

测试通过。

2.6 串口模块

2.6.1 功能

对外界的IO通道。

2.6.2 测试目标

测试串口的正确读写和时序要求。

2.6.3 测试过程及用例

串口工作模式为全双工。接收部分通过将接收的数据按顺序写入sram测试，发送部分通过按顺序读取sram的数据并发送测试。

2.6.4 测试结果

测试通过。

2.7 HDMI显示模块

2.7.1 功能

计算机的输出设备。

2.7.2 测试目标

测试显存的正确写入和显示程序的正确工作及时序要求。

2.7.3 测试过程及用例

通过显存的接口循环写入ASCII字符以测试显存是否正确写入以及字符的正常显示和屏幕的平滑滚动。

2.7.4 测试结果

测试通过。

3 指令测试

指令测试是在cpu逐渐完善过程中的阶段测试，每实现一组新的指令就会进行一次指令测试以验证所实现指令和各模块新增修改的正确无误。

3.1 逻辑运算指令

指令包括lui, ori, or, andi, and, xori, xor, nor

3.1.1 测试目标

首先测试数据相关问题的正确解决，然后测试指令的正确实现。

3.1.2 测试用例

数据相关测试用例。

```
ori $1,$0,0x1100        # $1 = $0 | 0x1100 = 0x1100
ori $1,$1,0x0020        # $1 = $1 | 0x0020 = 0x1120
ori $1,$1,0x4400        # $1 = $1 | 0x4400 = 0x5520
ori $1,$1,0x0044        # $1 = $1 | 0x0044 = 0x5564
```

逻辑指令测试用例。

```
lui   $1,0x0101
ori   $1,$1,0x0101
ori   $2,$1,0x1100        # $2 = $1 | 0x1100 = 0x01011101
or    $1,$1,$2            # $1 = $1 | $2 = 0x01011101
andi $3,$1,0x00fe         # $3 = $1 & 0x00fe = 0x00000000
and   $1,$3,$1            # $1 = $3 & $1 = 0x00000000
xori $4,$1,0xff00         # $4 = $1 ^ 0xff00 = 0x0000ff00
xor   $1,$4,$1            # $1 = $4 ^ $1 = 0x0000ff00
nor   $1,$4,$1            # $1 = $4 ~^ $1 = 0xffff00ff    nor is "not
      or"
```

3.1.3 测试结果

观测仿真波形图，与注释中的结果相符，数据相关问题成功解决，指令测试通过测试。

3.2 移位指令

指令包括sll, sllv, sra, srav, srl, srlv

3.2.1 测试目标

测试指令是否正确实现。

3.2.2 测试用例

指令测试用例。

```
lui    $2,0x0404
ori    $2,$2,0x0404
ori    $7,$0,0x7
ori    $5,$0,0x5
ori    $8,$0,0x8
```

```
sll   $2,$2,8    # $2 = 0x40404040 sll 8  = 0x04040400
sllv  $2,$2,$7   # $2 = 0x04040400 sll 7  = 0x02020000
srl   $2,$2,8    # $2 = 0x02020000 srl 8  = 0x00020200
srlv  $2,$2,$5   # $2 = 0x00020200 srl 5  = 0x00001010
nop
sll   $2,$2,19   # $2 = 0x00001010 sll 19 = 0x80800000
sra   $2,$2,16   # $2 = 0x80800000 sra 16 = 0xffff8080
srav  $2,$2,$8   # $2 = 0xffff8080 sra 8  = 0xffffff80
```

3.2.3 测试结果

观测仿真波形图，与注释中的结果相符，测试通过。

3.3 移动指令

指令包括mfhi, mflo, mthi, mtlo。

3.3.1 测试目标

首先测试新的数据相关问题的正确解决，然后测试指令的正确实现。

3.3.2 测试用例

数据相关和指令测试用例。

```
  .org 0x0
  .set noat
  .global _start
_start:
  lui $1,0x0000          # $1 = 0x00000000
  lui $2,0xffff          # $2 = 0xffff0000
  lui $3,0x0505          # $3 = 0x05050000
  lui $4,0x0000          # $4 = 0x00000000

  movz $4,$2,$1          # $4 = 0xffff0000
  movn $4,$3,$1          # $4 = 0xffff0000
  movn $4,$3,$2          # $4 = 0x05050000
  movz $4,$2,$3          # $4 = 0x05050000

  mthi $0                # hi = 0x00000000
  mthi $2                # hi = 0xffff0000
  mthi $3                # hi = 0x05050000
  mfhi $4                # $4 = 0x05050000

  mtlo $3                # li = 0x05050000
  mtlo $2                # li = 0xffff0000
  mtlo $1                # li = 0x00000000
  mflo $4                # $4 = 0x00000000
```

3.3.3 测试结果

观测仿真波形图，与注释中的结果相符，新的数据相关问题成功解决，指令测试通过测试。

3.4 算术运算指令

指令包括addiu, addu, mult, subu, slt, sltu, slti, sltiu。

3.4.1 测试目标

测试以上指令的正确性。

3.4.2 测试用例

指令测试用例。

```
  .org 0x0
  .set noat
  .global _start
_start:
  ######### add\addi\addiu\addu\sub\subu ##########
  ori  $1,$0,0x8000           # $1 = 0x8000
  sll  $1,$1,16               # $1 = 0x80000000
  ori  $1,$1,0x0010           # $1 = 0x80000010

  ori  $2,$0,0x8000           # $2 = 0x8000
  sll  $2,$2,16               # $2 = 0x80000000
  ori  $2,$2,0x0001           # $2 = 0x80000001

  ori  $3,$0,0x0000           # $3 = 0x00000000
  addu $3,$2,$1               # $3 = 0x00000011
  ori  $3,$0,0x0000           # $3 = 0x00000000
  add  $3,$2,$1               # overflow,$3 keep 0x00000000

  sub   $3,$1,$3              # $3 = 0x80000010
  subu  $3,$3,$2              # $3 = 0xF

  addi $3,$3,2                # $3 = 0x11
  ori  $3,$0,0x0000           # $3 = 0x00000000
  addiu $3,$3,0x8000          # $3 = 0xffff8000

  #########     slt\sltu\slti\sltiu     ##########
  or   $1,$0,0xffff           # $1 = 0xffff
  sll  $1,$1,16               # $1 = 0xffff0000
  slt  $2,$1,$0               # $2 = 1
  sltu $2,$1,$0               # $2 = 0
  slti $2,$1,0x8000           # $2 = 1
  sltiu $2,$1,0x8000          # $2 = 1

  #########          clo\clz          ##########
  lui $1,0x0000           # $1 = 0x00000000

  lui $1,0xffff           # $1 = 0xffff0000
  ori $1,$1,0xffff        # $1 = 0xffffffff

  lui $1,0xa100           # $1 = 0xa1000000

  lui $1,0x1100           # $1 = 0x11000000

  ori  $1,$0,0xffff
  sll  $1,$1,16
```

```
46    ori   $1,$1,0xfffb            # $1 = -5
47    ori   $2,$0,6                 # $2 = 6
48    mul   $3,$1,$2                # $3 = -30 = 0xfffffffe2
49
50    mult $1,$2                    # hi = 0xffffffff
51                                  # lo = 0xfffffffe2
52
53    multu $1,$2                   # hi = 0x5
54                                  # lo = 0xfffffffe2
55    nop
56    nop
```

3.4.3 测试结果

观测仿真波形图，与注释中的结果相符，测试通过。

3.5 分支跳转指令

指令包括beq, bgez, bgtz, blez, bltz, bne, j, jal, jalr, jr

3.5.1 测试目标

一方面测试以上指令及延迟槽机制的正确性，另一方面验证ctrl模块的部分功能正确性。

3.5.2 测试用例

指令测试用例。

```
1     .org 0x0
2     .set noat
3     .set noreorder
4     .set nomacro
5     .global _start
6  _start:
7     ori   $1,$0,0x0001    # $1 = 0x1
8     j     0x20
9     ori   $1,$0,0x0002    # $1 = 0x2
10    ori   $1,$0,0x1111
11    ori   $1,$0,0x1100
12
13    .org 0x20
14    ori   $1,$0,0x0003    # $1 = 0x3
15    jal   0x40
16    nop
17    ori   $1,$0,0x0005    # r1 = 0x5
18    ori   $1,$0,0x0006    # r1 = 0x6
19    j     0x60
20    nop
21
22    .org 0x40
23
24    jalr $2,$31
25    or    $1,$2,$0         # $1 = 0x48
```

```
26    ori   $1,$0,0x0009    # $1 = 0x9
27    ori   $1,$0,0x000a    # $1 = 0xa
28    j 0x80
29    nop
30
31    .org 0x60
32    ori   $1,$0,0x0007    # $1 = 0x7
33    jr    $2
34    ori   $1,$0,0x0008    # $1 = 0x8
35    ori   $1,$0,0x1111
36    ori   $1,$0,0x1100
37
38    .org 0x80
39    nop
40
41 _loop:
42    j _loop
43    nop
```

```
1     .org 0x0
2     .set noat
3     .set noreorder
4     .set nomacro
5     .global _start
6  _start:
7     ori   $3,$0,0x8000
8     sll   $3,16               # $3 = 0x80000000
9     ori   $1,$0,0x0001        # $1 = 0x1
10    b     s1
11    ori   $1,$0,0x0002        # $1 = 0x2
12 1:
13    ori   $1,$0,0x1111
14    ori   $1,$0,0x1100
15
16    .org 0x20
17 s1:
18    ori   $1,$0,0x0003        # $1 = 0x3
19    bal   s2                  # actually it's not "bal" since we don'
      t need inst "bal", anyway,  the machine code works well
20    nop
21    ori   $1,$0,0x1100
22    ori   $1,$0,0x1111
23    bne   $1,$0,s3
24    nop
25    ori   $1,$0,0x1100
26    ori   $1,$0,0x1111
27
28    .org 0x50
29 s2:
30    ori   $1,$0,0x0004      # $1 = 0x4
31    beq   $3,$3,s3
32    or    $1,$31,$0         # $1 = 0x2c
33    ori   $1,$0,0x1111
34    ori   $1,$0,0x1100
35 2:
36    ori   $1,$0,0x0007      # $1 = 0x7
37    ori   $1,$0,0x0008      # $1 = 0x8
38    bgtz  $1,s4
```

```
39      ori   $1,$0,0x0009      # $1 = 0x9
40      ori   $1,$0,0x1111
41      ori   $1,$0,0x1100
42
43      .org 0x80
44  s3:
45      ori   $1,$0,0x0005      # $1 = 0x5
46      BGEZ $1,2b
47      ori   $1,$0,0x0006      # $1 = 0x6
48      ori   $1,$0,0x1111
49      ori   $1,$0,0x1100
50
51      .org 0x100
52  s4:
53      ori   $1,$0,0x000a      # $1 = 0xa
54      BGEZAL $3,s3
55      or    $1,$0,$31         # $1 = 0x10c
56      ori   $1,$0,0x000b      # $1 = 0xb
57      ori   $1,$0,0x000c      # $1 = 0xc
58      ori   $1,$0,0x000d      # $1 = 0xd
59      ori   $1,$0,0x000e      # $1 = 0xe
60      bltz $3,s5
61      ori   $1,$0,0x000f      # $1 = 0xf
62      ori   $1,$0,0x1100
63
64
65      .org 0x130
66  s5:
67      ori   $1,$0,0x0010      # $1 = 0x10
68      blez $1,2b
69      ori   $1,$0,0x0011      # $1 = 0x11
70      ori   $1,$0,0x0012      # $1 = 0x12
71      ori   $1,$0,0x0013      # $1 = 0x13
72      bltzal $3,s6
73      or    $1,$0,$31         # $1 = 0x14c
74      ori   $1,$0,0x1100
75
76
77      .org 0x160
78  s6:
79      ori $1,$0,0x0014        # $1 = 0x14
80      nop
81
82
83
84  _loop:
85      j _loop
86      nop
```

3.5.3 测试结果

观测仿真波形图，与注释中的结果相符，ctrl跳转功能与指令测试通过。

3.6 访存指令

指令包括lb, lbu, lw, sb, sw。

3.6.1 测试目标

一方面测试以上指令正确性，另一方面测试流水线暂停以及是否正确解决load相关问题。

3.6.2 测试用例

指令测试用例。

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori   $3,$0,0xeeff
    sb    $3,0x3($0)        # [0x3] = 0xff
    srl   $3,$3,8
    sb    $3,0x2($0)        # [0x2] = 0xee
    ori   $3,$0,0xccdd
    sb    $3,0x1($0)        # [0x1] = 0xdd
    srl   $3,$3,8
    sb    $3,0x0($0)        # [0x0] = 0xcc
    lb    $1,0x3($0)        # $1 = 0xffffffff
    lbu   $1,0x2($0)        # $1 = 0x000000ee
    nop

    ori   $3,$0,0xaabb      # $3 = 0x0000aabb
    sh    $3,0x4($0)        # [0x4] = 0xbb, [0x5] = 0xaa
    lhu   $1,0x4($0)        # $1 = 0x0000aabb
    lh    $1,0x4($0)        # $1 = 0xffffaabb

    ori   $3,$0,0x8899
    sh    $3,0x6($0)        # [0x6] = 0x99, [0x7] = 0x88
    lh    $1,0x6($0)        # $1 = 0xffff8899
    lhu   $1,0x6($0)        # $1 = 0x00008899

    ori   $3,$0,0x4455
    sll   $3,$3,0x10
    ori   $3,$3,0x6677
    sw    $3,0x8($0)        # [0x8] = 0x77, [0x9]= 0x66, [0xa]= 0x55,
     [0xb] = 0x44
    lw    $1,0x8($0)        # $1 = 0x44556677

    lwl   $1, 0x5($0)       # $1 = 0xbb889977
    lwr   $1, 0x8($0)       # $1 = 0xbb889944

    nop
    swr   $1, 0x2($0)       # [0x0] = 0x88, [0x1] = 0x99, [0x2]= 0x44,
      [0x3] = 0xff
    swl   $1, 0x7($0)       # [0x4] = 0xaa, [0x5] = 0xbb, [0x6] = 0x88
     , [0x7] = 0xbb

    lw    $1, 0x0($0)       # $1 = 0x889944ff
    lw    $1, 0x4($0)       # $1 = 0xaabb8844

_loop:
```

```
46    j _loop
47    nop
```

load相关测试用例。

```
1     .org 0x0
2     .set noat
3     .set noreorder
4     .set nomacro
5     .global _start
6  _start:
7     ori $1,$0,0x1234     # $1 = 0x00001234
8     sw  $1,0x0($0)       # [0x0] = 0x00001234
9
10    ori $2,$0,0x1234     # $2 = 0x00001234
11    ori $1,$0,0x0        # $1 = 0x0
12    lw  $1,0x0($0)       # $1 = 0x00001234
13    beq $1,$2,Label
14    nop
15
16    ori $1,$0,0x4567
17    nop
18
19 Label:
20    ori $1,$0,0x89ab     # $1 = 0x000089ab
21    nop
22
23 _loop:
24    j _loop
25    nop
```

3.6.3 测试结果

观测仿真波形图，与注释中的结果相符，load相关问题成功解决，指令测试通过测试。

3.7 协处理器访问指令

指令包括mfc0, mtc0。

3.7.1 测试目标

一方面测试以上指令正确性，另一方面验证cp0模块的正确实现。

3.7.2 测试用例

指令测试用例。

```
1  ori $1,$0,0xf
2  mtc0 $1,$11,0x0  #write to reg Compare, start to count
3  ori $1,$1,0x401
4  mtc0 $1,$12,0x0  #status = 0x401
5  mfc0 $2,$12,0x0  #$2=0x401
```

3.7.3 测试结果

仿真一次通过。

3.8 异常处理相关

指令包括syscall, eret。

3.8.1 测试目标

一方面测试以上指令正确性，另一方面验证ctrl模块部分功能的正确运行。

3.8.2 测试用例

指令测试用例。

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori $1,$0,0x100     # $1 = 0x100
    jr $1
    nop

    .org 0x40
    ori $1,$0,0x8000    # $1 = 0x00008000
    ori $1,$0,0x9000    # $1 = 0x00009000
    mfc0 $1,$14,0x0     # $1 = 0x0000010c
    addi $1,$1,0x4      # $1 = 0x00000110
    mtc0 $1,$14,0x0
    eret
    nop

    .org 0x100
    ori $1,$0,0x1000    # $1 = 0x1000
    sw  $1, 0x0100($0)  # [0x100] = 0x00001000
    mthi $1             # HI = 0x00001000
    syscall
    lw  $1, 0x0100($0)  # $1 = 0x00001000
    mfhi $2             # $2 = 0x00001000
_loop:
    j _loop
    nop
```

时钟中断测试用例。

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori $1,$0,0x100     # $1 = 0x100
    jr $1
```

```
    nop

    .org 0x20
    addi $2,$2,0x1
    mfc0 $1,$11,0x0
    addi $1,$1,100
    mtc0 $1,$11,0x0
    eret
    nop

    .org 0x100
    ori $2,$0,0x0
    ori $1,$0,100
    mtc0 $1,$11,0x0
    lui $1,0x1000
    ori $1,$1,0x401
    mtc0 $1,$12,0x0


_loop:
    j _loop
    nop
```

3.8.3 测试结果

观测波形，与预期结果一致，测试通过。

4 系统测试

系统测试在指令测试之后进行，测例为课程所提供的功能测例，监控程序等。通过全方面的测试验证各模块各指令组合后能否正确工作。同时，在这一阶段的测试中，cpu将会烧入硬件，所以整体的时序性能要求也会成为测试的目标。

4.1 功能测例

选取功能测例中对应于我们所实现指令的40余个基本测例和若干会出现异常的测例，进行仿真测试和硬件测试，硬件测试时使用signal tap查看关键信号波形。

4.1.1 测试目标

更加强力的测试各条指令的正确性，以及配合上sram外设后的时序要求。

4.1.2 测试用例

代码过于冗长，不在此展示。

4.1.3 测试结果

指令SLTI存在错误，检查代码后修复。

sram读写存在问题，改为组合逻辑后修复。

MMU接入后存在信号接反的问题，发现后修复。

访存指令无法在50MHz的频率下正确执行，降频至25MHz后可以正常执行。

4.2 监控程序

4.2.1 测试目标

主要测试串口加入后能否正常工作，其次测试flash接入后能否正常读入。

4.2.2 测试用例

代码过于冗长，不在此展示。

4.2.3 测试结果

flash同样要求cpu降频，降至12Mhz后可以正常读入。

串口模块接入后存在信号对接错误，改正后修复。

提供的term程序存在错位错误，但与cpu无关。

4.3 ucore操作系统

4.3.1 测试目标

最后的综合测试。测试主要通过修改ucore代码和向串口打印信息，配合反汇编代码，进行分析。注：此部分测试由于硬件板子的flash存在无法写入的问题，主要通过在线板子进行，且前期使用跳过bootloader直接写入板子的sram的手段，避免flash的使用。经调试无误后，再使用flash。

4.3.2 测试用例

调试代码冗余较多，并未记录到项目中，不在此展示。

4.3.3 测试结果

1.发现lw指令符号拓展部分产生错误，改正cpu的EX阶段后修复。

2.因为前期使用监控程序，异常入口地址为监控程序的标准，与ucore不一致，改正cpu的CTRL模块修正。实际上，bootloader会将EBASE修改为正确的值，但是前期我们不使用bootloader，所以需要rst使调整EBASE的值。注：此部分不属于mips32的规定。

3.信号传递出现了异常。在MEM阶段存在异常时的访存指令的控制信号被MEM截断了，不流向MMU模块。
此截断导致了异常处理时的访存失败，具体表现为异常时写指令无效。此错误综合了代码调试和硬件调试的两部分分析才被定位。

4.前期调试时不理解时钟中断的作用，注释了时钟中断，导致在用户进程处串口中断触发了也不会唤醒等待标准输入的用户程序。即时间片轮询的方式被注释了。后经队友提醒还原了注释。

5.流水线异常时跳转存在问题。具体问题表现为进入用户程序后下一次跳转回到了核心态。ucore通过syscall进入用户程序，然后在umain.cpp的umain中调用main函数，main的地址由用户程序的main占据。实际上的运行是：进入umain函数再回到核心态。该错误通过在umain中加入一行输出代码解决。

5 性能测试

5.1 测试目标

测试cpu最高工作频率

5.2 测试结果

仿真阶段cpu可以在50Mhz时钟下正常工作，通过仿真的功能测例。

硬件测试阶段，由于sram的读写时序要求，cpu需要降频至25MHz。

硬件测试阶段，由于Flash读时序要求，CPU需要降频至12MHz

最终cpu在12Mhz时钟下通过所有测试