软工计原联合项目

测试文档

NonExist 组
张钰晖，杨一滨，周正平

目录

Chapter 1

文档说明

1.1 背景

本项目的顶级需求为在 Thinpad 开发板上运行 ucore 操作系统。其衍生需求为设计一个基于 MIPS 32 架构的 CPU，以实现 ucore 所需的 46 条指令、精确异常与外设调度。

本项目是计算机组成原理和软件工程课程的联合实验，项目需求方为计算机组成原理与软件工程课程。

本项目的承担方为 NonExist 小组，包括计 55 班张钰晖、计 55 班杨一滨、计 54 班周正平 3 位成员。

1.2 编写目的

本需求文档的编写目的如下：

1. **需求分析：** 明确软件（ucore）对硬件（CPU）的具体需求，及需完成的功能
2. **系统概述：** 对系统总体框架进行清晰完整的描述
3. **学习目标：** 明确开发所需技术，订立学习目标

其目标读者包括 NonExist 小组的开发者、成品 CPU 的使用者、项目的评估者。

1.3 需求概览

首先给出本项目的需求分析图：

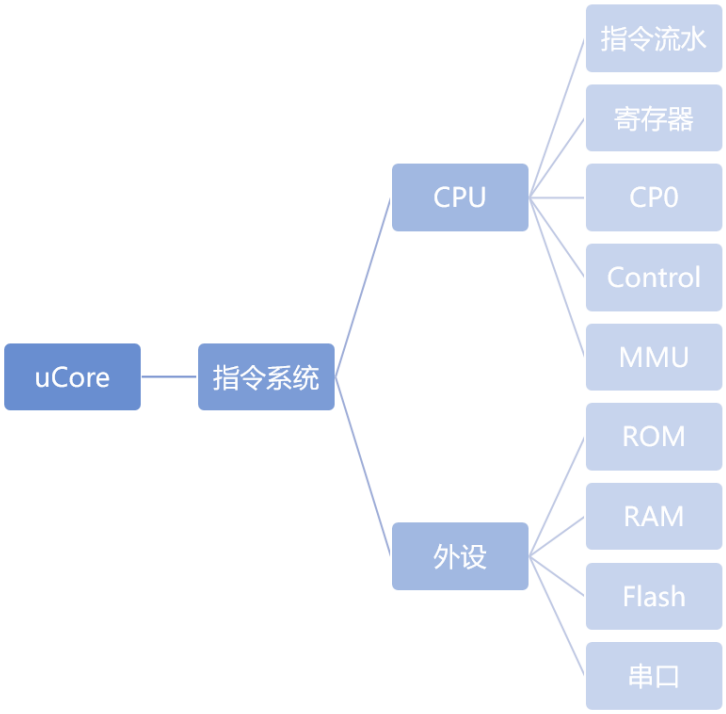


图 1.1: 需求总览

可以看出，本项目的需求可以作如下分层：

1. **ucore**：操作系统，编译后由大量 MIPS 指令组成。需要硬件实现 MIPS 指令系统的一个子集。
2. **指令系统**：是 MIPS 指令系统的一个子集，需要硬件提供 CPU（核心调度）与外部设备（组织集成）。
3. **CPU 与外部设备**：是硬件最底层的实现，由大量基础模块组成。各个基础模块由开发板提供的 FPGA 或外围芯片实现。

本文档也因而遵从自顶向下的分析流程，其组织结构与需求分析的层次相似。

1.4 定义

以下列出此项目涉及的部分英文简称及其含义（详细解释会在后文给出）：

英文简称	英文全称	含义
MIPS	Microprocessor without interlocked piped stages	一种典型 RISC 指令集，本实验 CPU 的基本架构
CPU	Central Processing Unit	中央处理器，负责硬件的核心调度
ALU	Arithmetic Logic Unit	算术逻辑单元
HI/LO	High/Low Register	寄存器，用于存储乘法运算结果 (64 位) 的高、低各 32 位
MMU	Memory Management Unit	内存管理单元，用于地址映射
TLB	Translation Lookaside Buffer	快表，用于通过缓存加速 MMU 地址映射查表速度
VA	Virtual Address	虚拟地址
PA	Physical Address	物理地址
ROM	Read-Only Memory	只读存储器，用于存储 BootLoader
RAM	Random Access Memory	随机存储器，用于实现内存
FLASH	Flash Memory	快闪存储器，用于实现硬盘
VGA	Video Graphics Array	视频图形阵列，一种显示器接口标准
HDMI	High Definition Multimedia Interface	高清晰度多媒体接口，一种显示器接口标准

Chapter 2

开发环境

这一部分对本项目涉及的开发环境进行说明。
本章分为如下几个部分：

1. **硬件端开发环境**：对实验开发板及其配套 EDA 工具进行说明
2. **软件端开发环境**：对 ucore 操作系统所需的编译流程、仿真方法、运行平台进行说明

2.1 硬件端开发环境

硬件端开发环境，主要包括实验开发板及配套的 EDA 开发工具。以下进行总结：

1. **Thinpad 开发板 (Late 2017 版)**

元件	型号
FPGA	Xilinx Artix-7 xc7a100tfgg676-2L
RAM	IS61WV102416ALL 32 位 2 块各 4MB
FLASH	JS28F640J3D75 8MB
HDMI	TFP410

2. **EDA 开发工具**

软件	运行平台
Vivado 2017.3	Windows 10 64bit
串口调试精灵	Windows 10 64bit
二进制显示软件	Windows 10 64bit

3. **云端硬件调试平台 ThinpadCloud**

通过该在线平台，可以远程对开发板进行操控，而无需随身携带，极大地方便了前期调试。其网址如下：

平台	网址
ThinpadCloud	http://thinpad.dynv6.net:8000

2.2 软件端开发环境

软件端开发环境，主要指 ucore 操作系统所需的编译、仿真、运行环境。以下进行总结：

1. **编译**

工具	版本号	平台	主要功能
mips-sde-elf-gcc	4.6.3	Ubuntu 16.04 x64	编译 ucore

编译应按照如下流程进行：

(a) **下载 ucore-thumips 源代码**

这里我们需下载 ucore-thumips, 是 ucore 操作系统的 MIPS 移植版本。从其 GitHub 仓库[1]下载即可，不再赘述。

(b) **安装 mips-sde-elf-gcc**

i. 从 Sourcery CodeBench 官网[2]下载文件 mips-{版本号}-mips-sde-elf.bin

ii. 执行`./mips-{版本号}-mips-sde-elf.bin` 以执行默认安装向导，这一步骤之后应完成环境变量的配置。可输入 `mips-sde-elf-gcc --version` 进行验证。

iii. 如环境变量配置不成功,应手动在 ~/.bashrc 中加入`exportPATH="{path/to/mipsgcc}/bin:$PATH"` 并重启终端。

(c) **修改 Makefile**

需修改的变量	修改后的值	说明
GCCPREFIX	GCCPREFIX:=mips-sde-elf-	用于指定编译工具
ON_FPGA	y/n	y 表示为 FPGA 编译，用于硬件上实际运行 n 表示为 qemu 模拟器编译，用于仿真调试
USER_APPLIST	sh, ls, cat, ...	指定 ucore 可支持的应用程序列表 当 ON_FPGA=y 时，默认仅支持 sh 命令， 后期调试通过后可增加其他命令

[1] https://github.com/chyh1990/ucore-thumips

[2] https://sourcery.mentor.com/GNUToolchain/release2189

(d) **make**

输入 `make` 命令以编译。

2. **仿真**

工具	平台	主要功能
qemu-thumips	Ubuntu 16.04 x64	1. 通过修改 thumips_insn.txt，确定指令子集能否支持 ucore 2. 对比观察 ucore 在开发板和在模拟器上的运行输出，推断错误原因

仿真应按如下流程进行：

(a) **下载 qemu-thumips 源代码**

这里我们需下载 qemu-thumips，是 qemu 模拟器的 MIPS 移植版本。从其 GitHub 仓库[3]下载即可，不再赘述。

(b) **配置**

输入`./configure -target-list=mipsel-softmmu` 命令即可。

(c) **make**

输入 `make` 命令以编译。

需注意，这一步依具体平台可能出现链接错误。笔者出现的链接错误位于 qemu-timer.o，最终通过修改 Makefile 与 Makefile.target 解决：

```
# Makefile, line 37
LIBS+=-lz -lrt -lm $(LIBS_TOOLS)

# Makefile.target, line 37
LIBS+=-lz -lrt -lm
```

3. **运行**

ucore 经过编译后，应将生成的 obj/ucore-kernel-initrd (注意根目录下 flash.img 是一个软链接文件，链接指向 ucore-kernel-initrd) 文件写入 FLASH 中。详见 “ucore 需求分析” 的 “Boot 阶段” 一节。

[3]https://github.com/chyh1990/QEMU-thumips

Chapter 3

ucore 需求分析

正如文档说明部分所述，“运行 ucore 操作系统” 为本项目的顶级需求。因此，本章将对 ucore 对硬件与指令系统的需求进行详细的分析。

本章分为以下几部分：

1. **总述**：简要叙述 ucore 操作系统运行成功的标准
2. **Boot 过程**：操作系统的启动与初始化流程
3. **指令系统**：为运行 ucore，CPU 应支持的指令集合
4. **地址映射**：包括 ucore 的内存管理与外设调度，叙述其对 MMU 的需求
5. **异常处理**：描述 ucore 的异常处理机制，及硬件需实现的所有异常

操作系统逻辑复杂，每部分内容不尽相同，但除总述外，在介绍流程上都遵循以下主体框架：

1. **相关文件**：列举 ucore 与这部分紧密相关的文件，便于读者查阅
2. **原理说明**：说明 ucore 在这部分的实现机制或流程
3. **需求分析**：根据以上分析，明确此部分对硬件的需求

3.1 总述

本次实验的最终目的为成功在 Thinpad 开发板上运行 ucore 操作系统。所谓 “运行成功”，需要通过如下 2 个标准衡量：

1. **系统正常启动**：ucore 操作系统能正常 Boot、初始化并最终输出信息 “user sh is running!!!”，中途不发生系统 panic (操统发生致命错误且无法修复的情况)：
2. **能正常运行各个应用程序**：能在硬件上正确运行 ucore 可支持的所有应用程序，需要输出正确。

   需注意，所有应用程序均为 *ucore* 在软件端实现。它们的根本作用在于测试硬件实现的正确性，硬件无需关心其实现机制。只要正确实现 *ucore* 操作系统对硬件的需求，便应当可以运行所有的应用程序。此外，如果需进行扩展，也可在此添加自己使用 *C* 语言或汇编语言编写的应用程序。

所有应用程序清单如下，它们定义在 ucore 的 user/目录：

应用程序	功能（期望输出）
ls	打印文件系统信息
pwd	打印当前完整路径
cat	打印一个文件的内容
sh	打印一条语句 “user sh is running!!!”
forktest	连续调用 fork() 函数，新建一系列线程，并打印信息
yield	连续调用 yield() 函数，要求进程重新调度，并打印信息
hello	打印一条语句 “Hello world!!.” 并显示当前进程编号
faultreadkernel	在用户态访问内核态地址，结果是系统 panic
faultread	在用户态访问非法内存地址 0x0，结果是系统 panic
badarg	先调用 fork() 再调用 yield()，打印进程调度信息
pgdir	打印页表信息
exit	先调用 fork() 再连续调用 yield()，打印进程调度与退出信息
sleep	每隔一定时间打印一条信息

3.2 Boot 阶段

Boot 阶段为操作系统的启动阶段。

3.2.1 相关文件

文件	简介
boot/bootasm.S	BootLoader 的 MIPS 汇编代码，需编译后存入 ROM
init/init.c	操作系统入口

3.2.2 Boot 过程

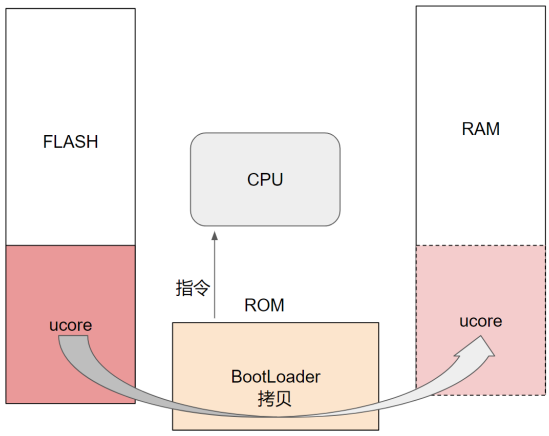


图 3.1: BootLoader 将 ucore 从 FLASH 拷贝至 RAM 中

由于 FLASH 断电不消失的特性，ucore 需要存放在 FLASH 中；而系统加载完毕后，ucore 应该在内存（RAM）中。因此，需要一个 Boot 程序，负责在硬件启动时将 ucore 从 FLASH 导入至 RAM。

CPU 初始化时，置 PC 为 Boot 程序首地址，从 ROM 开始处（VA 0xBFC00000）运行 Boot 程序。每次从 FLASH 读出数据再写入 RAM，直至拷贝完成。

BootLoader 拷贝完成后，将 PC 跳转至 VA 0x80000000，进入操作系统入口。此时 ucore 进行初始化并输出调试信息。

3.2.3 需求分析

由上述分析可知，Boot 过程的需求包括以下几点：

1. **ROM**：使用 FPGA 实现 ROM，并设置硬件初始化时的 PC 值为 ROM 的虚拟地址（见“地址映射”一节）
2. **BootLoader**：将 BootLoader 编译后写入 ROM
3. **ucore**：将 ucore 编译后写入 FLASH

3.3 指令系统

3.3.1 相关文件

ucore 的所有文件

3.3.2 需求分析

ucore 编译完成后，便会被转化成一条条 MIPS 指令。通过进行 qemu 仿真可知，ucore 运行共需要 45 条指令，包括如下几部分：

1. **算术逻辑指令**：共 22 条，包括加减乘、与或非、移位等指令
2. **分支跳转指令**：共 10 条，包括分支（B）与跳转（J）指令
3. **访存指令**：共 5 条，包括读取（L）与写入（S）指令
4. **移动指令**：共 4 条，包括对 HILO 寄存器的读写指令
5. **陷入指令**：共 1 条，包括 SYSCALL
6. **特权指令**：共 5 条，包括对 CP0 的访问、异常返回及 TLB 异常时使用的指令

通过这 45 条指令，ucore 实现了作为操作系统的调度功能。
该部分的需求为在硬件上实现这 45 条指令。详细分析见“指令系统需求分析”一章。

3.4 地址映射

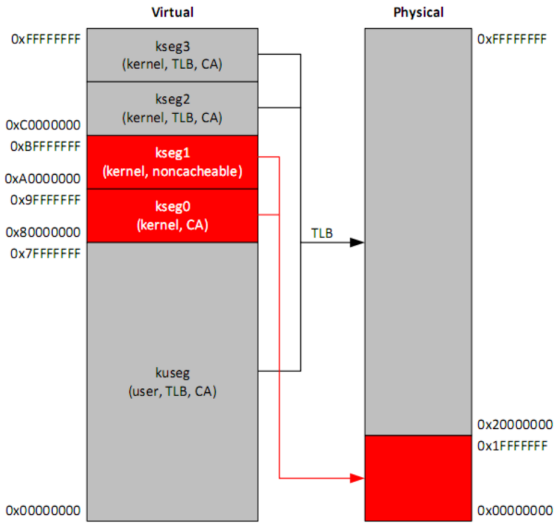


图 3.2: MIPS 标准地址映射

地址映射在硬件上由 MMU（Memory Manage Unit）完成，其最重要的意义体现在 2 方面：

1. **内存管理（面向软件）**：使进程在寻址时可以超过物理内存大小，且多个进程的地址空间互不干扰
2. **外设调度（面向硬件）**：面向 CPU 统一各个外设的访问接口，使之均可使用访存指令进行读写

这一部分将深入解析 ucore 的地址映射标准，从而明确对硬件 MMU（Memory Manage Unit）的需求。

3.4.1 相关文件

首先列出 ucore 定义地址映射的关键文件：

文件	关键变量	值	简介
boot/bootasm.S	FLASH_START	0xBE000000	FLASH 的起始虚拟地址
	FLASH_SIZE	0x01000000	FLASH 的地址大小
kern/mm/memlayout.h	KMEMSIZE	1M	内存总大小
kern/include/mips_vm.h	MIPS_KUSEG	0x00000000	kuseg 段起始地址
	MIPS_KSEG0	0x80000000	kseg0 段起始地址
	MIPS_KSEG1	0xa0000000	kseg1 段起始地址
	MIPS_KSEG2	0xc0000000	kseg2 段起始地址
include/thumips.h	COM1	0xBFD003F8	串口虚拟地址

3.4.2 内存管理

内存管理需求主要面向运行在操统之上的应用软件。首先，应用程序的大小不应受到物理内存的限制，例如开发板上的 RAM 总大小仅有 8M，但对于 32 位机器而言，可用虚拟地址空间可达 $2^{32} = 4G$；其次，各应用程序的地址空间应各自独立。

页表 操统因而需维护页表（Page Table），以实现从虚拟地址到物理地址的转换。具体而言，每当程序访问内存时，需进行 2 次访存：

1. **查询页表**：读取内存中的页表，查找对应的物理地址
2. **获取数据**：访问该物理地址，获取数据

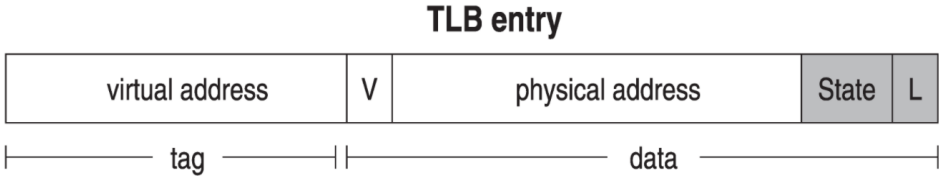


图 3.3: TLB 表项

TLB 由于内存访问具有局部性，如能将页表最近被访问的一部分以 CPU 内部的逻辑单元存储，则访存效率可获得极大提高。TLB（Translation Lookaside Buffer）就是这样一种高速缓存。具体而言，在加入 TLB 之后，程序访存流程如下：

1. **查询 TLB**：根据虚拟地址的高 20 位（32 减去页位数 12），查找 TLB 中的表项

2. **TLB HIT**：如 TLB 中有此表项，则直接其对应的物理地址

3. **TLB MISS**：如 TLB 中无此表项，则再进一步访问内存查询页表，获取物理地址之后，将其作为一个新表项写入 TLB

3.4.3 外设调度

在 CPU 核心之外，运转着 ROM、RAM（相当于内存）、FLASH（相当于硬盘）、串口（相当于标准输出）、VGA（相当于显示屏）等多个外部设备。它们的用途、接口、访问时序各不相同。

然而，对于操作系统与 CPU 而言，它们的访问接口是统一的：均使用 L/S 型指令实现收/发数据。这个过程中需要进行地址映射。

RAM 除操统文件中的定义、“内存管理” 小节的说明外，考虑到开发板上提供了 2 块大小为 4M 的 RAM（Base RAM 与 Ext RAM），可将 KMEMSIZE 改为 8M，从而提供更大的内存空间。

FLASH、串口 见操统文件中的定义。

ROM 在 Boot 阶段，CPU 复位时 PC 位于 VA 0xBFC00000 处，因而该虚拟地址应映射到 ROM。此外，考虑 BootLoader 指令较少，为 ROM 分配 1KB 的地址空间即可。

VGA 此外，项目计划实现拓展功能 VGA，其屏幕大小为 800x600，故而需要 468.75KB 的地址空间用于显存。姑且在 kseg1 段中分配一段开始于 VA 0xBA000000、大小为 512KB 的地址空间留作显存。

3.4.4 需求分析

综上所述，MMU 需要将 VA 的 kuseg、kseg2 段通过 TLB 映射到内存（RAM），在 kseg0 段通过抹去最高位直接得到 RAM 中的 PA，在 kseg1 段将部分地址映射到除 RAM 外的各个外设。

以下给出最终地址映射方案：

段（权限）	虚拟地址	映射目标（物理地址）	备注
kuseg（用户态）	0x00000000 - 0x7FFFFFFF	RAM（通过查询 TLB 动态确定）	用户程序
kseg0（内核态）	0x80000000 - 0x807FFFFF	RAM（0x00000000 - 0x007FFFFF）	开机时存放操作系统
kseg1（内核态）	0xBE000000 - 0xBEFFFFFF	FLASH	关机时存放操作系统
	0xBFC00000 - 0xBFC00FFF	ROM	存放 BootLoader
	0xBFD003F8 - 0xBFD003FC	串口	串口数据/串口状态
	0xBA000000 - 0xBA080000	VGA	显存，用于显示图像
kseg2（内核态）	0xC0000000 - 0xFFFFFFFF	RAM（通过查询 TLB 动态确定）	

3.5 异常处理

在 MIPS32 架构中，有一些事件会打断程序的正常执行流程。一些由**外部事件**触发，如串口有数据待读入，称为中断；另一些则由 CPU **内部指令**引起，如算术溢出、系统调用等。这些事件统称为异常。

以下对异常处理流程进行归纳，并总结 ucore 涉及的所有异常，从而明确对硬件的需求。

3.5.1 相关文件

以下列出 ucore 异常处理的部分关键文件：

文件	关键函数	简介
kern/trap/vector.S	___exception__vector	异常处理向量，汇集各入口
kern/trap/exception.S	ramExcHandle__tlbmiss	TLB 异常处理入口
	ramExcHandle__general	通用异常处理入口
	common__exception	保存现场、调用操统异常处理代码、恢复现场
kern/trap/trap.c	mips__trap	操统异常处理代码
	trap__dispatch	被 mips__trap 调用，分类处理各种异常

3.5.2 异常处理流程

硬件端（CPU）

1. **异常响应**：硬件检测异常，并将异常原因、类型等存入 CP0 相关寄存器中
2. **异常处理**：将 PC 跳转到 CP0 Ebase 寄存器所指示的操统异常处理入口地址，并禁用异常检测
3. **异常返回**：执行 ERET 指令，跳转回被异常打断的指令，重新使能异常检测

软件端（ucore）

1. **异常响应**：保存现场（通用寄存器等）至内存中
2. **异常处理**：从异常处理入口处开始，根据异常类型，执行异常处理代码
3. **异常返回**：从内存中恢复现场，并使用 ERET 指令返回

3.5.3 需求分析

根据上述分析，只需针对 ucore 能够处理的那些异常，在硬件上加以实现，即可满足异常处理需求。

阅读 trap.c：trap__dispatch 函数可知 ucore 处理的所有异常类型。它们通过 CP0 Cause 寄存器的 ExcCode（异常号）字段加以区分：

中断 中断由 CP0 Cause 寄存器的 IP 标志位（中断号）进一步区分，包括如下 2 种：

异常名（异常号）	中断名（中断号）	硬件触发条件	ucore 处理流程
Interrupt（0）	时钟中断（7）	CP0 Compare 与 Count 寄存器的值相等	进程调度后重启时钟
	串口中断（4）	串行接口处有数据待写入	读串口并写入 stdin

TLB MISS 在访存 load 或 store 时 TLB 中无匹配表项时产生，包括如下 2 种异常：

异常名（异常号）	硬件触发条件	ucore 处理流程
TLBL（2）	访存 load 时 TLB 中无匹配表项	首先根据 CP0 BadVAddr 寄存器确定缺失地址，然后通过设置 CP0 相关寄存器构造新的 TLB 表项，最后使用 tlbwr 指令随机选择一个位置，重填该项
TLBS（3）	访存 store 时 TLB 中无匹配表项	同上

系统调用 系统调用由 Syscall 序号进一步区分，其类型在操作系统中定义，硬件无需关心其具体实现：

异常名（异常号）	硬件触发条件	ucore 处理流程
Syscall（8）	执行指令 SYSCALL	根据 Syscall 序号，调用 syscall/syscall.c 中相应的处理代码

其他异常 此类异常无需在硬件上实现。因为它们在 ucore 中的实现均为简单报错，如发生在用户态，则用户进程退出；如发生在内核态，则系统 panic。总而言之，即使硬件实现了这些异常，亦无法使 ucore 在这些情况下正常运行。为完整起见，亦在此列出：

异常名（异常号）	硬件触发条件	ucore 处理流程
ADEL（4）	访存 load 时地址未对齐	简单报错
ADES（5）	访存 store 时地址未对齐	简单报错
RI（10）	无效指令	简单报错
CPU（11）	协处理器不可用	简单报错
OV（12）	算术溢出	简单报错

Chapter 4

指令系统需求分析

指令系统是硬件（CPU）对 ucore 唯一透明可见的接口，也因而成为在“运行 ucore”之下的第二级需求。

在对 ucore 进行需求分析后可知，在指令系统这一层面，一共有 46 条需要实现的指令。各条指令的详细格式见附录。

本章包括如下几个部分：

1. **概述**：对指令流水线的简要介绍
2. **算术逻辑指令**：共 22 条，包括加减乘、与或非、移位等指令
3. **分支跳转指令**：共 10 条，包括分支（B）与跳转（J）指令
4. **访存指令**：共 5 条，包括读取（L）与写入（S）指令
5. **移动指令**：共 4 条，包括对 HI/LO 寄存器的读写指令
6. **陷入指令**：共 1 条，包括 SYSCALL
7. **特权指令**：共 5 条，包括对 CP0 的访问、异常返回及 TLB 异常时使用的指令
8. **总结**：以上指令对硬件需求的扼要总结

本章除概述与总结外，每节遵从以下介绍流程：

1. **功能**：简要描述此类指令的功能
2. **硬件需求**：为实现此类指令，对硬件的结构和功能需求
3. **异常**：此类指令可能触发的异常，或对异常处理的影响

4.1 概述

本次项目计划实现指令流水 CPU。概括说来，每条指令被分为以下 5 个阶段，每个阶段由一个硬件单元执行：

名称	代号	功能
取指	IF	从指令存储器中读取指令
译码	ID	指令译码，同时读取寄存器
执行	EX	执行操作，或计算地址
访存	MEM	进行访存操作
回写	WB	将计算结果写入寄存器

以下文档内容中的“流水线”指的是这 5 个硬件单元构成的整体。它负责指令的执行，同时集成了对周边硬件单元的调度工作，在 CPU 中居于核心位置。

对流水线结构的详细分析见“CPU 需求分析”一章。

4.2 算术逻辑指令

算术逻辑指令共 22 条，包括加减乘、与或非、移位等，总结如下：

4.2.1 功能

指令	运算	功能
ADDIU、ADDU	A + B	无符号数加法
SUBU	A - B	无符号数减法
MULT	A × B	乘法
SLT、SLTI、SLTIU、SLTU	A < B	符号数比较、无符号数比较
AND、ANDI	A and B	与
OR、ORI	A or B	或
NOR	A nor B	或非
XOR、XORI	A xor B	异或
SLL、SLLV	A sll B	逻辑左移
SRL、SRLV	A srl B	逻辑右移
SRA、SRAV	A sra B	算术右移
LUI	A = imm || 0^{16}	加载立即数至寄存器

4.2.2 硬件需求

这部分对硬件的需求主要体现在 ALU 上，它位于流水线的第 3 部分（EX）。具体地，ALU 需接受 2 个 32 位整数及相关控制信号作为输入，并以 1 个 32 位整数作为输出。控制信号决定了 ALU 执行的运算；此外，运算结果应在 WB 阶段被写入寄存器。

特别地，乘法运算的结果是一个 64 位整数，因此需要 HI/LO 寄存器分别用于存储乘积的高、低 32 位，而不能存入通用寄存器。

算术逻辑运算指令对硬件的需求总结如下：

1. **流水线**：能对算术运算指令进行解码识别。能读写通用寄存器。能计算 2 个 32 位整数作指定运算的结果，并产生一个 32 位整数作为输出。
2. **HI/LO**：当运算类型为乘法时，可以保存 64 位的乘法运算结果。

4.2.3 异常

无。

4.3 分支跳转指令

分支跳转指令共 10 条，包括分支（Branch）与跳转（Jump）2 类指令，总结如下：

4.3.1 功能

指令	功能
BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ	分支指令，用于有条件跳转。 条件包括 $==, !=, \geq, >, \leq, <$。
J、JR JAL、JALR	跳转指令，用于无条件跳转； 同上，但会使用寄存器（常为 R[31]）预先保存此前的 PC 值

4.3.2 硬件需求

这部分对硬件的需求主要集中于流水线的设计。此外，由于延迟槽指令的存在，异常处理需增加一些特殊的判断逻辑，详细分析见“异常”部分。

跳转指令对硬件的需求主要集中于流水线，总结如下：

1. **流水线**：能对分支跳转指令进行解码识别。能读写通用寄存器。此外能改变指令的执行顺序并判断状态：
   - **PC**：可以在每个周期取指时，根据“是否需要跳转”这一控制信号，对 PC 的下一次取指做出判断：如需跳转，则将 PC 赋值为跳转目标地址；否则 PC 照常自增 4。
   - **延迟槽**：流水线中各个硬件单元可以确定当前正在执行的指令是否在延迟槽中。更具体的分析见“异常部分”。

4.3.3 异常

分支跳转指令本身不会产生异常。但由于分支跳转指令引入了延迟槽指令，这会对异常处理带来额外的判断逻辑。

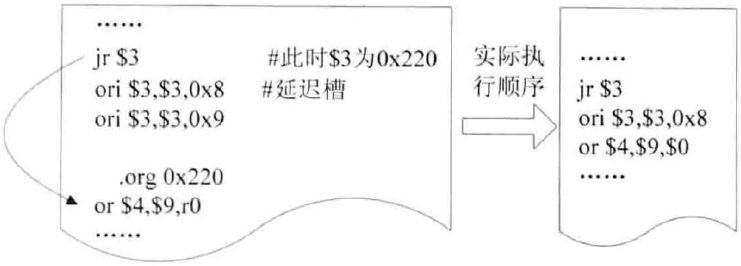


图 4.1: 延迟槽指令

延迟槽指令 我们规定分支指令后面的指令位置为分支延迟槽，其中的指令称为延迟指令。由于在计算分支目标地址时，延迟指令已经进入流水线，因而无论跳转发生与否，它都必然被执行。

异常处理相关 具体地，在异常处理时，需将异常发生时的指令地址（PC 值）存入 CP0 的 EPC 寄存器中，以便异常处理结束后返回该地址继续执行。然而如果触发异常的指令为延迟槽指令，则需要将 EPC 寄存器的值置为 PC-4，而非 PC 的当前值。这是因为在延迟槽指令之前可能已经发生了跳转，因而其后需继续执行的指令序列应位于跳转的目标地址，而非延迟槽指令之后。

因此，流水线中的各个硬件单元需能够确定当前指令是否位于延迟槽，以便在发生异常时可以向 EPC 寄存器中保存正确的地址。

4.4 访存指令

访存指令共 5 条，包括读取（Load）与写入（Store）2 类指令，总结如下：

4.4.1 功能

指令	功能
LB、LBU	读内存中的一个字节（8 位）
LW	读内存中的一个字（32 位）
SB	写内存中第一个字节（8 位）
SW	写内存中的一个字（32 位）

4.4.2 硬件需求

这一部分对硬件的需求包括 3 大部分。首先，流水线需要对访存指令进行译码，如果是读，则需要读取相应的寄存器；否则为写，需要将结果写入寄存器中；其次，需要在流水线框架之上集成其他外

设，包括在 ucore 部分已经有所介绍的 ROM、RAM、FLASH、串口、VGA 等；最后，为了在指令向 CPU 提供的虚拟地址与外设的物理地址之间实现衔接，硬件需要通过 MMU 与 TLB 进行地址映射。

访存指令对硬件的主要需求可以总结如下：

1. **流水线**：能对访存指令进行解码识别。能读写通用寄存器。能通过 MMU 以统一的接口访问外设。能执行 TLB 异常处理流程。
2. **外设**：能在流水线的框架之上集成外设，实现其读写操作。
3. **MMU**：能将指令提供的虚拟地址映射到对应的外设，实现地址转换。
4. **CP0**：能提供寄存器记录可能的异常信息。

4.4.3 异常

在 ucore 需求分析部分得出的异常列表中，访存指令可能触发其中的 TLB MISS 异常、地址未对齐异常。其中，TLB MISS 异常为硬件需要实现的异常。

4.5 移动指令

移动指令共 4 条，主要包括对 HI/LO 寄存器的读写指令。总结如下：

4.5.1 功能

指令	功能
MFHI、MFLO	读 HI/LO 寄存器
MTHI、MTLO	写 HI/LO 寄存器

4.5.2 硬件需求

这一部分对硬件的需求集中于对 HI/LO 寄存器的访问。具体地，流水线需要在识别出这些指令的类型后，实现与 HI/LO 寄存器的读写交互。HI/LO 寄存器则需要实现相应的访问接口以支持读写。

总结如下：

1. **流水线**：能对移动指令进行解码识别。能读写通用寄存器。能访问 HI/LO 寄存器的读写接口。
2. **HI/LO**：能提供接口支持读写操作。

4.5.3 异常

无。

4.6 陷入指令

陷入指令共包括 1 条指令 SYSCALL。

4.6.1 功能

指令	功能
SYSCALL	系统调用指令，操统执行此指令时，陷入内核态并调用相应异常处理代码

4.6.2 硬件需求

这一部分对硬件的需求主要集中于异常处理：此条指令会触发 SYSCALL 异常，因而硬件需要首先在流水线阶段识别该指令，然后依通用异常处理流程将异常原因等写入 CP0 相关寄存器中。

SYSCALL 指令对硬件的主要需求可以总结如下：

1. **流水线**：能对 SYSCALL 指令进行解码识别。能执行通用异常处理流程。
2. **CP0**：能提供寄存器记录异常信息。

4.6.3 异常

触发 SYSCALL 异常。

4.7 特权指令

特权指令共 5 条，包括对 CP0 的访问、TLB 的维护、异常的返回等功能，总结如下：

4.7.1 功能

指令	功能
MFC0、MTC0	读写协处理器 CP0
TLBWI、TLBWR	写 TLB 表项
ERET	异常返回：取消异常标志并返回受害指令地址

4.7.2 硬件需求

这一部分对硬件的需求较而杂。首先，这类指令之所以称为特权指令，是因为它们只能在内核态被调用，其功能多与异常处理、TLB 维护（多用于 TLB 异常处理）等相关。其次，无论是 TLB 的维护，还是异常的处理，都需要流水线、MMU、CP0 等多个硬件单元的协同配合。

这里仅简要总结这部分指令对硬件的需求：

1. **流水线**：能对各特权指令进行解码识别。能读写通用寄存器。能执行通用异常、TLB 异常处理流程。
2. **MMU**：能写入 TLB 表项。
3. **CP0**：能提供寄存器记录可能的异常信息。

4.7.3 异常

无。

4.8 总结

总体来说，指令系统对硬件的需求可以统一总结如下：

所属部分	硬件单元	需求
CPU	**流水线**	**取指（IF）**：能每周期取得下一条指令的地址。 **译码（ID）**：能对不同的指令解码识别。能读写通用寄存器。 **执行（EX）**：能实现 ALU 提供的运算功能。能从 CP0、HI/LO 读出数据。 **访存（MEM）**：能通过 MMU 提供的接口对外部设备进行读写访问。 **回写（WB）**：能将数据写入通用寄存器或 CP0、HI/LO 寄存器。 各个部分能协同实现通用异常、TLB 异常处理流程。
	寄存器堆	能提供 32 个通用寄存器用于实现指令中操作数的传递。能提供读写接口。
	HI/LO 寄存器	能保存乘法运算的结果（64 位整数）。能提供读写接口。
	CP0	能提供不同寄存器以记录异常信息的不同字段。能提供读写接口。
	MMU	能实现 ucore 所需的地址映射功能。能实现 TLB 的表项写入操作。
外设	**外设**	能以黑盒方式使用外设，将其集成到 MMU 之下，提供读写接口。

Chapter 5

CPU 需求分析

紧承上一章对指令系统的需求分析，我们将其所依赖的硬件系统分为 2 大部分：

1. **CPU**：硬件系统的核心。需作为白盒实现其全部控制逻辑。
2. **外设**：由 CPU 进行调度。需以黑盒方式集成至 CPU 之下。

本章主要对 CPU 部分进行需求分析。其中，流水线承担着指令运行和调度的功能，可谓是 CPU 的心脏。许多“血管”从中延伸出来，将数据输送到 CP0、MMU 等多个模块，犹如人体的五脏六腑。当然，正如心脏需要心房心室，CPU 还需要通用寄存器、HI/LO 寄存器等用于数据的暂存。

以上的比喻描绘了一个极其粗略的 CPU 结构蓝图。

本章分为如下几个部分：

1. **指令流水**：详细介绍五级流水线架构，及其可能遇到的冒险问题
2. **寄存器**：包括 32 个通用寄存器与 HI/LO 寄存器
3. **CP0**：协处理器 CP0
4. **Control**：用于控制流水线的运行
5. **MMU**：内存管理单元，用于地址映射

每部分介绍其标准解读或是功能说明。

5.1 指令流水

流水线承担着指令运行和调度的功能，在时钟脉冲的作用下周而复始地执行着一条条指令，可谓是 CPU 的心脏。

何为五级流水线？具体来说，体现在以下 2 个方面：

1. **每条指令被拆分为 5 个步骤**：共包括 5 个硬件单元，每个单元负责其中的一个环节。

名称	代号	功能
取指	IF	从指令存储器中读取指令
译码	ID	指令译码，同时读取寄存器
执行	EX	执行操作，或计算地址
访存	MEM	进行访存操作
回写	WB	将计算结果写入寄存器

2. **5 个硬件单元并行执行**：在任何一个时钟周期内，上述的 5 个硬件单元分别在处理第 $n+5, n+4, n+3, n+2, n+1$ 条指令的第 1, 2, 3, 4, 5 个阶段（如下图所示）：

图 5.1: 流水线结构

硬件单元的并行执行同时有其优点和缺点：

- **优点**：并行结构加速指令的执行。设每条指令每个步骤的平均执行时间为 $\triangle t$，则对于一个多周期 CPU（无流水线结构）而言，n 条指令所需时间为 $5n\triangle t$；而对于流水线结构而言，n 条指令仅需 $(n+4)\triangle t$。可以看出，当指令数目很大时，流水线结构可带来约 4 倍左右的性能提升。
- **缺点**：并行结构同样带来了冒险问题。具体地，有 2 种情形由此产生：
  - **数据冒险**：例如后一条指令的 ID 阶段需读取前一条指令在 WB 阶段才写入的寄存器值，但这在时序上矛盾。
  - **控制冒险**：例如跳转指令在 ID 阶段才能确定目标地址，然而此时其下一条指令（延迟槽指令）已进入流水线的 IF 阶段。

总体来说，指令流水线需每周期并行执行 5 条指令，能实现暂停清空等控制逻辑，并且能作为系统的核心，在其上集成各个外设单元。此外，还需设法解决其中可能涉及的冒险问题。

5.2 寄存器

如果将流水线比喻成 CPU 的心脏，那么寄存器便好像其中贮存血液的心房，储存着 CPU 的临时数据。

这里说的“寄存器”，既包括 32 个通用寄存器，还包括 HI/LO 寄存器用于存储乘法运算的结果。寄存器使用 FPGA 上的逻辑单元实现，它们被贡献出来用于硬件系统的数据存储，因而也成为了 CPU 的珍贵资源。

尽管与硬件实现关系不大，作为对寄存器功能的补充说明，以下列出各个寄存器在软件编写时的使用惯例：

所属部分	寄存器号	寄存器名	简介
通用寄存器	\$0	$zero$	常量 0，不可写入
	\$1	a_t	保留，用于汇编器
	\$2-\$3	$v_0 - v_1$	函数返回值
	\$4-\$7	$a_0 - a_3$	函数参数
	\$8-\$15	$t_0 - t_7$	调用者保存
	\$16-\$23	$s_0 - s_7$	被调用者保存
	\$24-\$25	$t_8 - t_9$	调用者保存
	\$26-\$27	$k_0 - k_1$	保留，用于异常处理
	\$28	gp	全局指针
	\$29	sp	堆栈指针
	\$30	fp	帧指针
	\$31	ra	返回地址
HI/LO 寄存器	-	HI/LO	存储乘法结果

无论是通用寄存器，还是 HI/LO 寄存器，都需要支持 2 种基本操作：读和写。读操作不改变寄存器中的数据，可以在时钟周期的任意时刻进行，表现为组合逻辑；写操作需要修改寄存器中的数据，因而只能在时钟的上升沿进行，表现为时序逻辑。

寄存器的读写由使能信号进行控制。

MIPS32 标准指令集中，涉及寄存器访问的指令（R 型指令）至多同时包括 3 个通用寄存器。其中 2 个为读，1 个为写。因而，对于通用寄存器而言，只需要支持同时以组合逻辑读 2 个寄存器，并支持在时钟上升沿写 1 个寄存器即可。

5.3 CP0

CP0 在 CPU 中居于十分特别的地位。从本质上说，CP0 只不过是一些特殊定义的寄存器的集合，与上面所述的通用寄存器、HI/LO 寄存器没有本质的不同。然而，它又像是一个淋巴结，汇集了大量来自 CPU 其余部分的状态信息。当异常来临，操作系统通过读取其中存储的异常原因等信息来了解 CPU 的状态，正如淋巴结在人体的免疫过程中发挥着重要的指示作用。

以上对 CP0 的比喻是十分粗浅且不甚恰切的，然而可帮助读者建立对 CP0 的直观印象。

具体来说，ucore 需要 CP0 提供的寄存器如下表所示：

寄存器号	寄存器名	功能
0	*Index*	TLB 阵列的入口索引
1	*Random*	随机数发生器，产生 TLB 阵列的随机入口索引
2	*EntryLo*0	TLB 偶数虚页入口地址的低 32 位部分
3	*EntryLo*1	TLB 奇数虚页入口地址的低 32 位部分
8	*BadV Addr*	最近一次发生访存异常的虚拟地址
9	*Count*	与 *Compare* 组成片内计时器，二者中断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* 组成片内计时器，二者中断时发出时钟中断信号
12	*Status*	CPU 当前的多种重要状态，包括异常标志与中断使能等
13	*Cause*	最近一次异常的原因
14	*EPC*	最近一次异常时的 PC 值（如异常发生于延迟槽指令，则为 PC-4）
15	*Ebase*	操作系统异常处理程序的入口地址

可以看出，在本次实验中，CP0 中的寄存器大多可以归纳于以下 3 大类功能：

功能分类	主要寄存器	简要描述
时钟模块	Count、Compare	二者相等时触发时钟中断。操统依此对进程进行调度
异常控制	Status、Cause、EPC、Ebase、BadVAddr	分别保存异常状态、原因、受害指令地址、处理程序地址。特别地，BadVAddr 寄存器用于处理 TLB 相关异常
TLB 维护	Index、Random、EntryLo0、EntryLo1、EntryHi、BadVAddr	分别保存 TLB 阵列的索引、TLB 阵列的随机入口索引、TLB 入口地址低位、TLB 入口地址高位

以下按照功能分类，简要说明各个寄存器需提供硬件实现的部分。

注意：许多寄存器字段需要维护。一些需由硬件确定其应写入的值，而另一些由操统负责决定，硬件只需提供其读写接口即可。这一点在下面的表格中以 *"硬件维护"1* 项予以阐明。

时钟模块

1. **Count 寄存器**：

字段名	bit	描述	硬件维护	软件读写
Count	31:0	一个不停计数的 32 位寄存器，其计数频率一般同 CPU 的时钟频率。当计数达到 32 位无符号数上限时，从 0 开始重新计数。	是	R/W

2. **Compare 寄存器**：

字段名	bit	描述	硬件维护	软件读写
Compare	31:0	当 Count 寄存器中的计数值与 Compare 中的值相等时，产生时钟中断。该中断保持至有数据被写入 Compare 寄存器。	是	R/W

异常控制

1. **Status 寄存器**：其 32 位划分为多个字段，用于记录 CPU 与异常处理相关的多个状态。

   其中，部分字段需由硬件提供实现，简述如下：

字段名	bit	描述	硬件维护	软件读写
CU3-CU0	31:28	对应的协处理器（共 4 个）是否可用。本实验仅实现 CP0，故这几位应保持为常量 4'b0001。	是	R/W
IM7-IM0	15:8	对应的中断源（共 8 个）各自是否被屏蔽。屏蔽时置 0。	否	R/W
EXL	1	是否处于异常级。异常发生时置 1。	是	R/W
IE	0	全局中断只能位。使能时为 1。	否	R/W

2. **Cause 寄存器**：类似 Status，其 32 位划分为多个字段，用于全面地记录最近一次异常的原因。

   其中，部分字段需由硬件提供实现，简述如下：

字段名	bit	描述	硬件维护	软件读写
BD	31	当前发生异常的指令是否位于延迟槽中。位于延迟槽时置 1。	是	R
IV	23	指明中断异常使用一般异常向量（0x180）还是特殊中断向量（0x200）。	否	R/W
WP	22	观测挂起字段。与调试相关。	否	R/W
IP	15:10	对应硬件中断是否发生。	是	R
IP	9:8	对应软件中断是否发生。	是	R/W
ExcCode	6:2	异常号。具体类型见“ucore 需求分析”的“异常处理”一节。	是	R

3. **EPC 寄存器**：

字段名	bit	描述	硬件维护	软件读写
EPC	31:0	用于存储异常返回地址。如触发异常的指令不位于延迟槽中，则存储对应的 PC 值；否则存储 PC - 4。	是	R/W

4. **Ebase 寄存器**：其 32 位划分为多个字段，用于保存操作系统的异常处理入口地址。硬件在发生异常时跳转至此处。

   其中，部分字段需由硬件提供实现，简述如下：

字段名	bit	描述	硬件维护	软件读写
1	31	写入时被忽略，读取时返回 0。	是	R
0	30, 11:10	必须写为 0，读取时返回 0。	是	R
Exception	29:12	操作系统的异常处理入口地址。	否	R/W
CPUNum	9:0	表示 CPU 数量。本工程仅实现单处理器，置为 0 即可。	是	R

5. **BadVAddr 寄存器**：

字段名	bit	描述	硬件维护	软件读写
BadVAddr	31:0	保存最近一次访存异常（TLB MISS）的虚拟地址。	是	R

TLB 维护

1. **Index 寄存器**：

字段名	bit	描述	硬件维护	软件读写
P	31	检测故障。	否	R/W
0	30:6	必须写为 0，读取时返回 0。	是	R
Index	5:0	TLB 入口的索引值	否	R/W

2. **Random 寄存器**：

字段名	bit	描述	硬件维护	软件读写
0	31:6	必须写为 0，读取时返回 0。	是	R
Random	5:0	TLB 随机索引	是	R

3. **EntryLo0、EntryLo1 寄存器**：相当于 TLBWI、TLBWR 指令的接口。其中 EntryLo0 管理偶数页入口，EntryLo1 管理奇数页入口。

字段名	bit	描述	硬件维护	软件读写
0	29:26	必须写为 0，读取时返回 0。	是	R
PFN、CDVG	25:0	物理页帧号及状态信息	否	R/W

4. **EntryHi 寄存器**：包含用于 TLB 读、写和访问操作的虚拟地址匹配信息。

字段名	bit	描述	硬件维护	软件读写
VPN2	31:13	虚拟地址的高 19 位	否	R/W
0	12:8	必须写为 0，读取时返回 0。	是	R
ASID	7:0	地址空间标识符，由操统维护。	否	R/W

5.4 Control

Control 模块用于控制流水线的清空与暂停，还负责在异常发生时确定 PC 跳转的目标地址。若仍将流水线比作心脏，Control 模块就像是控制心脏搏动的神经。

暂停 有时一条指令需多个周期完成，如某些访存指令。这时，为保证流水线上各个硬件单元仍满足时序关系，便需要将流水线暂停相应的周期数。

清空 MIPS 32 标准要求实现精确异常，因此在发生异常的指令之后的指令尽管已经进入流水线，仍需将流水线清空，以确保其不被执行。

异常处理入口地址 当发生异常时，需要将 PC 跳转至操统提供的异常处理入口地址。这一地址需在 Control 模块中赋值为 CP0 Ebase 的值，再由 Control 模块传递至 PC 模块。

5.5 MMU

MMU 为 CPU 与外设打交道的模块。各个外设可以看做实现了读写功能的黑盒子，经 MMU 集成至 CPU 的控制之下。

根据上文 “ucore 需求分析” 的 “地址映射” 一节，MMU 需实现地址映射功能，即将虚拟地址转化为外设的物理地址。其需求细节在那一节中已经充分阐明，此处不再赘述。

Chapter 6

外设需求分析

在上一章中，将本次工程需要实现的硬件系统划分为 CPU、外设 2 大部分，并对其中的 CPU 部分进行了需求分析。本章分析外设部分。

本章分为如下几个部分：

1. **ROM**：只读存储器，用于存储 BootLoader
2. **RAM**：随机存取存储器，用作内存
3. **FLASH**：闪存，用作硬盘
4. **串口**：串行接口，用作通信（标准输入输出）
5. **VGA**：光栅扫描显示器，用作图像显示
6. **总结**：对各个外设的需求进行总结

除总结外，每部分主要介绍其参数、文档信息及功能。

6.1 ROM

ROM 为只读存储器，需提供读支持。

ROM 需利用 FPGA 上的逻辑单元实现，在 Thinpad 开发板上不额外提供元件。其具体需求在“ucore 需求分析”的“Boot 阶段”一节已经详述，此处不再赘言。

6.2 RAM

RAM 在系统中充当内存，需提供读写支持。

Thinpad 开发板上提供的 RAM 包括 BaseRAM（基础内存）与 ExtRAM（扩展内存）2 部分，其共同特点是断电后数据会消失。其大小均为 4MB，每次可读或写 32 位。

在寻址时，BaseRAM 与 ExtRAM 均通过 MMU 统一进行地址映射。

在硬件端，我们需要将 RAM 作为一个黑盒子集成至 CPU。这要求阅读其文档（“使用说明书”），明确这个黑盒子提供的读、写、使能信号等接口。这样，在 CPU 的 MMU 部分便可实现接口的衔接：

接口信号	含义	接口类型
ram_data[31:0]	data，数据总线，位宽 32 位，通过选择信号选择读哪些 byte	inout
ram_addr[19:0]	address，地址总线，位宽 20 位，故 RAM 总大小为 $32 \times 2^{20} = 4MB$	output
ram_be_n[3:0]	select，选择信号，位宽 4 位，选择 32bit=4byte 的哪些位进行读取或写入	output
ram_ce_n	chip enable，第一次下降有效	output
ram_oe_n	output enable，低电平为读入，置 0 表示使能	output
ram_we_n	write enable，低电平为写入，置 0 表示使能	output

文档中提供的读写时序图如下，CPU 在集成时需注意协调满足：

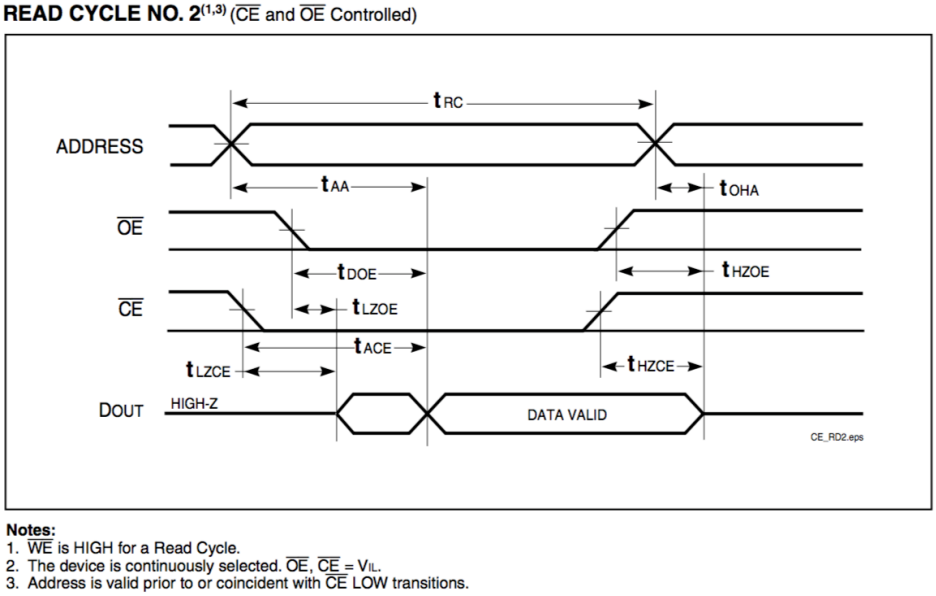


图 6.1: RAM 读时序

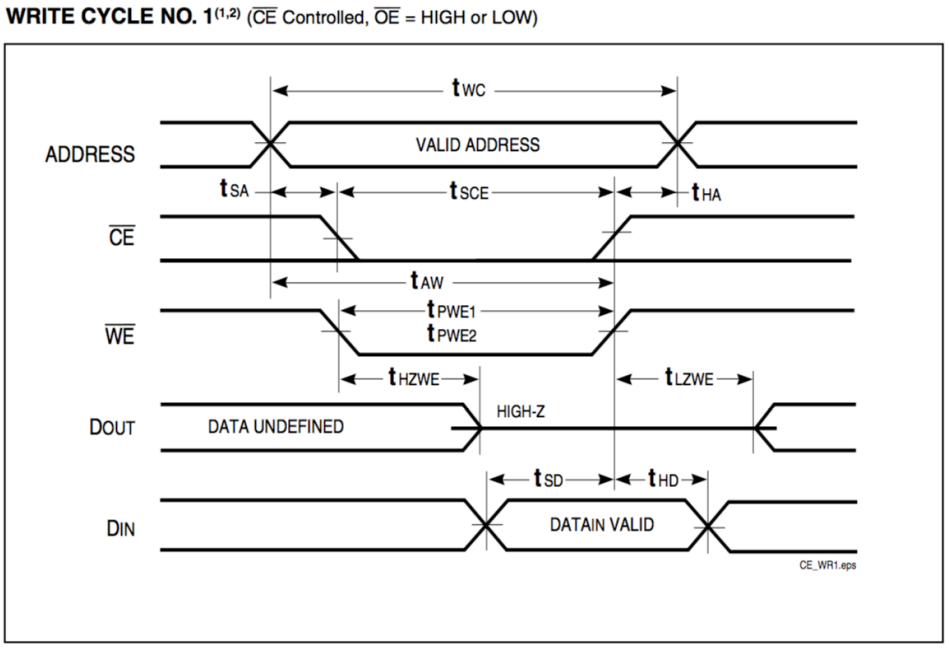


图 6.2: RAM 写时序

6.3 Flash

FLASH 在系统中充当硬盘，用于存储 ucore，仅需提供读支持。

Thinpad 开发板上提供的 FLASH 在掉电后不会丢失数据，其大小为 8MB，采用字编址，每个地址空间存 8bit，每次可选择读 16 位或 8 位。

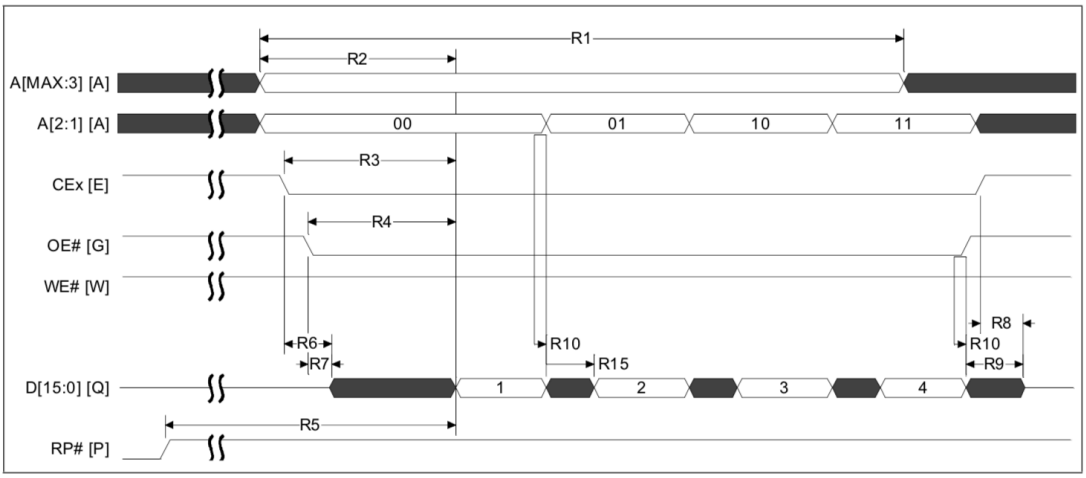
在寻址时，FLASH 通过 MMU 统一进行地址映射。

在硬件端，我们需要将 FLASH 作为一个黑盒子集成至 MMU。这同样要求阅读其文档：

接口信号	含义	接口类型
flash_a[22:0]	address，地址总线，位宽 23 位，故 FLASH 总大小为 $8 \times 2^{23} = 8MB$	output
flash_rp_n	reset，高电平为正常操作，低电平节电模式	output
flash_oe_n	output enable，低电平为读入，置 0 表示使能	output
flash_data[15:0]	data，数据总线，位宽 16 位，一次读 16bit	inout
flash_ce_n	chip enable，第一次下降有效	output
flash_byte_n	byte enable，高电平一次读 16bit，低电平一次读 8bit	output
flash_we_n	write enable，低电平为写入，置 0 表示使能	output
flash_vpen	写保护，置 0 表示使能	output

文档中提供的读时序图如下，CPU 在集成时需注意协调满足：

Figure 12: 4-Word Asynchronous Page Mode Read Waveform



Note: CE_X low is defined as the last edge of CE0, CE1, or CE2 that enables the device. CE_X high is defined at the first edge of CE0, CE1, or CE2 that disables the device.

图 6.3: FLASH 读时序

6.4 串口

串行接口在系统中负责通信，需提供读写支持。ucore 读、写标准输入均通过此处。

Thinpad 开发板上提供的串口通过 UART 芯片进行访问，0xBFD003F8 用作数据发送接收端口，及另 0xBFD003FC 位用于串口状态标记。

寻址时，串口通过 MMU 统一进行地址映射。

在硬件端，我们需要将串口作为一个黑盒子集成至 MMU。不同的是，串口的控制代码（async.v）已经由助教提供，并封装为 2 个模块。其接口如下所示：

接口信号	含义	接口类型
clk	时钟	input
TxD_start	开始标志，开始传输数据前需有 1 周期保持为高电平	input
TxD_data[7:0]	发送至 FPGA 外部的数据，每次可发送 8 位	input
TxD	串行化后，发送至 FPGA 外部的数据，每次发送 1 位	output
TxD_busy	串口状态标志是否繁忙	output

表 6.1: 模块 1：async_transmitter 接口

接口信号	含义	接口类型
clk	时钟	input
RxD	串行化的，来自 FPGA 外部的数据，每次发送 1 位	input
RxD_data_ready	串口是否有准备好的数据待接收	output
RxD_data[7:0]	去串行化后，来自 FPGA 外部的数据，每次可接收 8 位	output
RxD_idle	当一段时间内无接收数据时置 1	output
RxD_endofpacket	当一个数据包被识别时置 1 一个时钟周期	output

表 6.2: 模块 2：async_receiver 接口

网站 http://www.fpga4fun.com/SerialInterface.html 中提供的读写概念图如下：

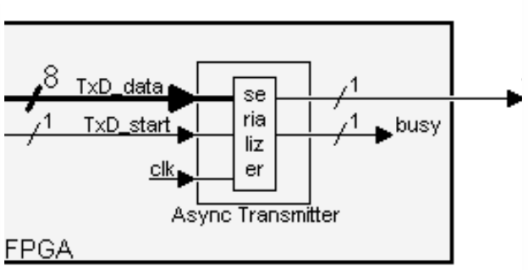


图 6.4: 模块 1：async_transmitter

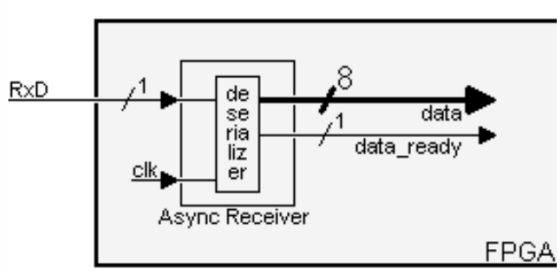


图 6.5: 模块 2：async_receiver

当串口传输数据时，需将其串行化后进行发送，遵循如下步骤：

1. 无数据时，串口发送 idle 信号 (=1)。
2. 每发送 1byte = 8bit 前，串口发送 start 信号 (=0)。
3. 依次发送该字节的 8 位。
4. 每发送 1byte = 8bit 后，串口发送 stop 信号 (=0)。

例如，发送字节 0x55 时，串口发送数据的时序图如下：

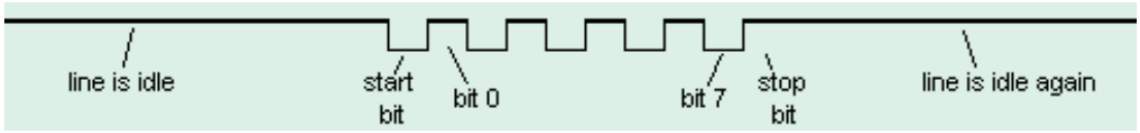


图 6.6: 串口发送字节 0x55

6.5 VGA

VGA 为图像显示设备，需要显存提供读写支持。该设备的具体需求为通过逐行扫描的方式实现图像显示。

Thinpad 开发板提供了 HDMI 接口，使用时可与自己的 HDMI 显示屏进行连接。本次实验所使用的显示屏分辨率为 800×600 像素，屏幕刷新率为 75Hz。在将图像像素编码为 8 位（其中 R/G/B 各 3/3/2 位）后，输出至该引脚，即可实现图像显示功能。

寻址时，显存通过 MMU 统一进行地址映射。

在硬件端，我们需要将 VGA 集成至 MMU。其中逐行扫描屏幕的代码（vga.v）已经由助教提供，我们只需再实现其中的显存模块。vga.v 的接口如下所示：

接口信号	含义	接口类型
clk	时钟	input
hsync	水平同步信号	output
vsync	竖直同步信号	output
hdata	水平像素位置	output
vdata	竖直像素位置	output
data_enable	数据使能信号，使能时置 1	output

表 6.3: vga.v 模块接口

6.6 总结

以下对各个外设的需求进行总结：

外设	功能需求	读写支持
ROM	只读存储器，能用于烧录存储 BootLoader	R
RAM	随机访问存储器，相当于内存，能用于实现运行时数据存储	R/W
FLASH	闪存，相当于硬盘，能用于存储 ucore	R
串口	串行并行转换接口，能实现收发数据通信	R/W
VGA	图像显示接口，能逐行扫描显示图像	R/W

参考文献

[1] 《自己动手写 CPU》雷思磊

[2] 《计算机硬件系统实验教程》刘卫东，李山山，宋佳兴

[3] 《32-bits MIPS CPU 需求文档》谢磊，李北辰

[4] 《32 位 MIPS 需求文档》章彦恺，周恩泽，沈光耀

[5] 《MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set》MIPS Technologies, Inc.
<TODO>: 指令列表