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：

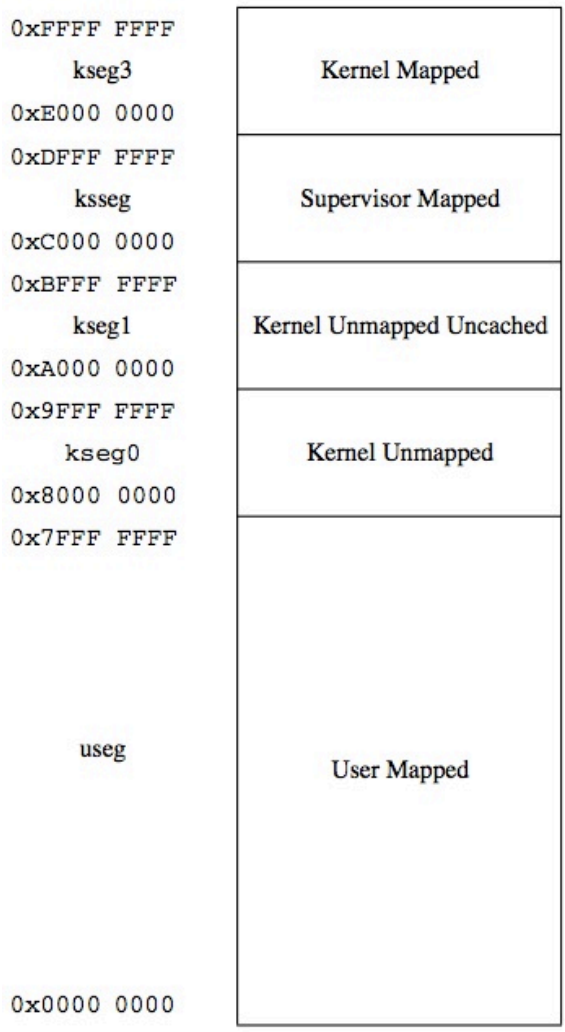

Figure 2.1: Virtual Address Space

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：

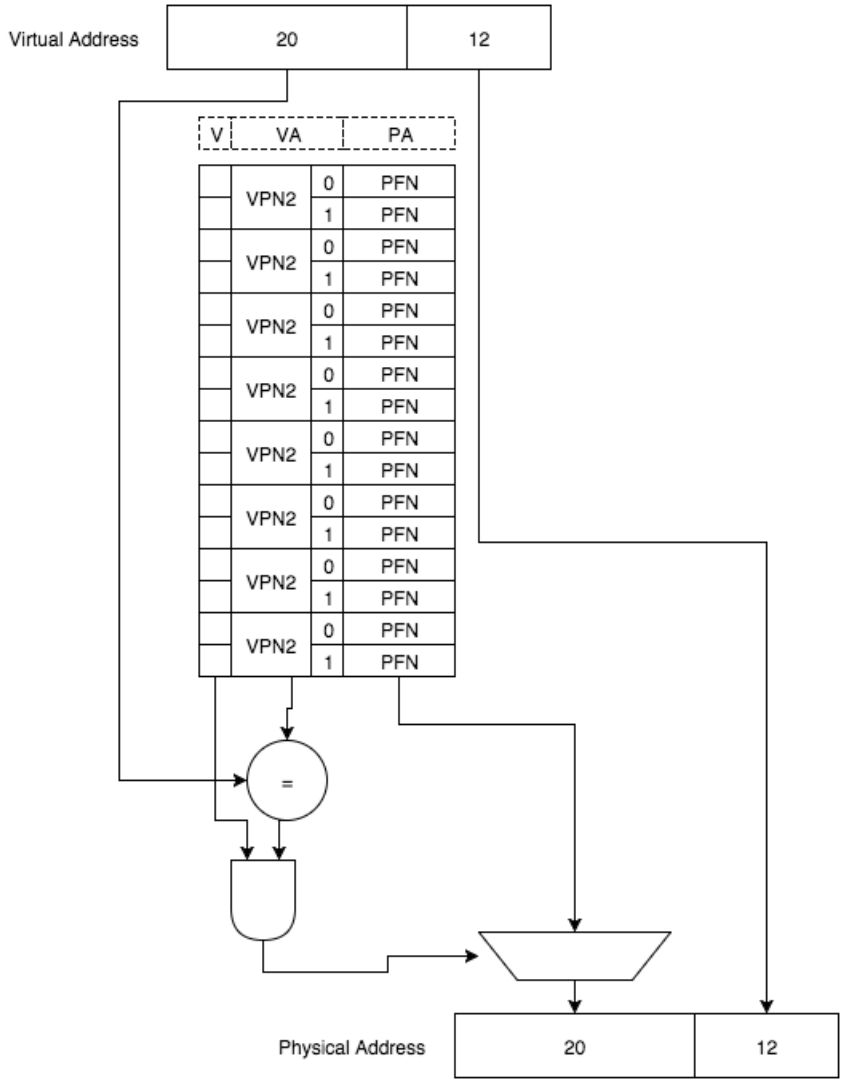

Figure 3.1: TLB 转换流程

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

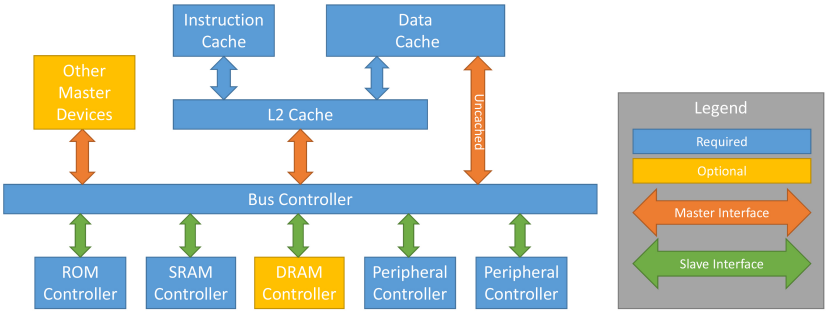

Figure 3.2: Memory System Architecture

The L1 Cache is organized as follows

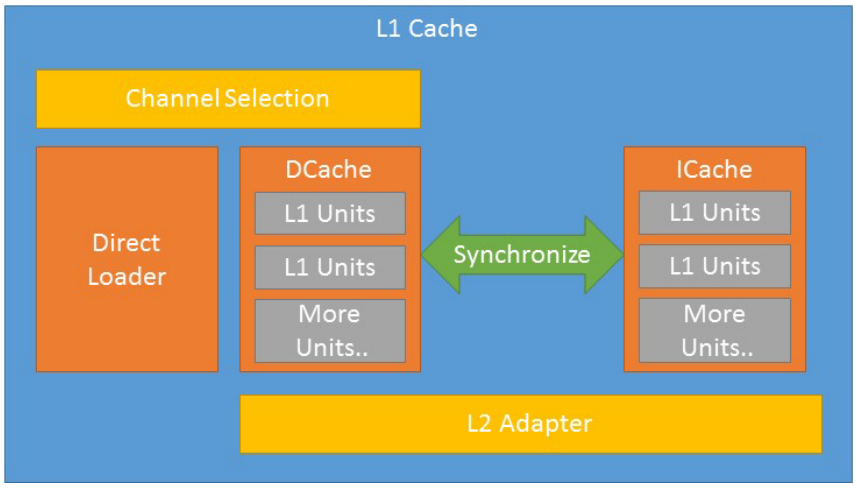

Figure 3.3: L1 Cache System Diagram

The L2 Cache is organized as follows

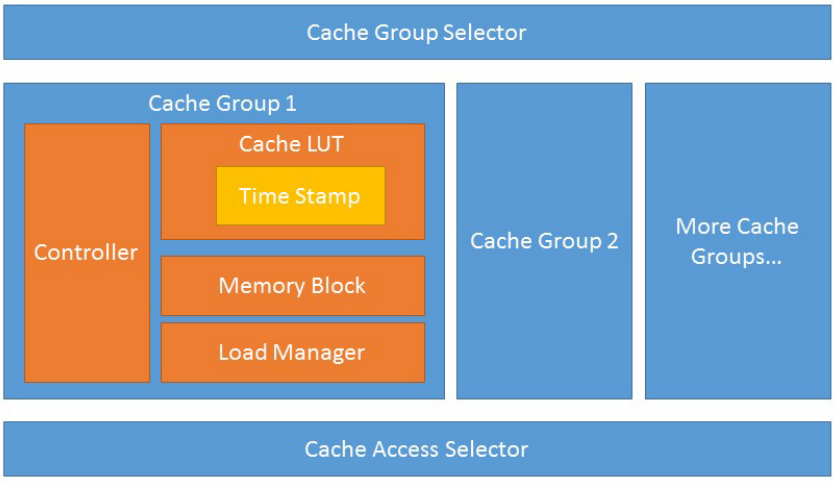

Figure 3.4: L2 Cache System Diagram

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

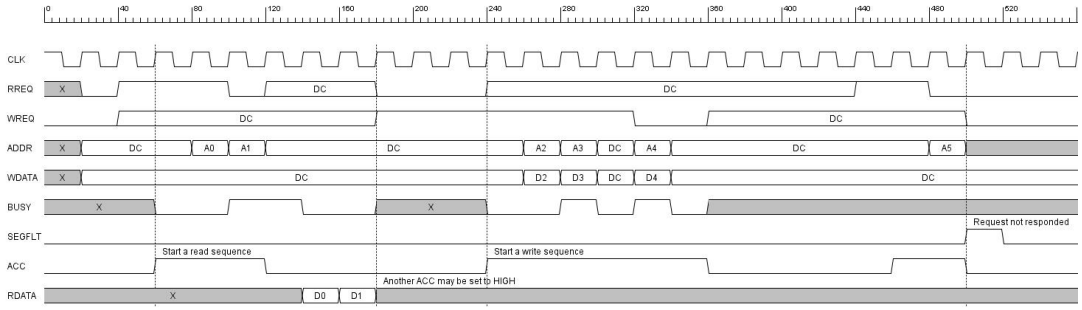

Figure 3.5: Master Device Timing Diagram

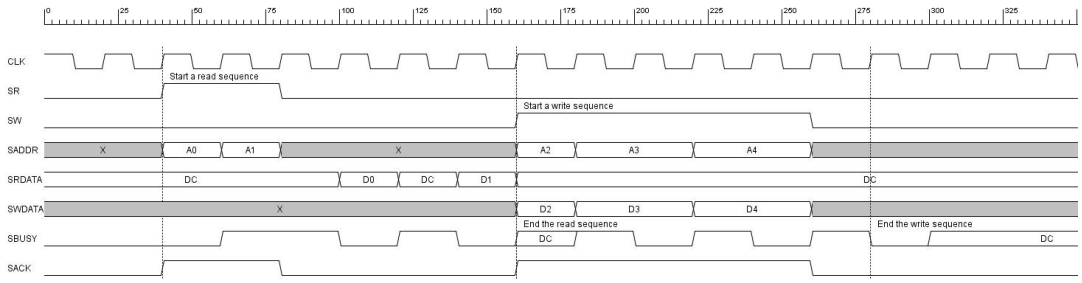

Figure 3.6: Slave Device Timing Diagram

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

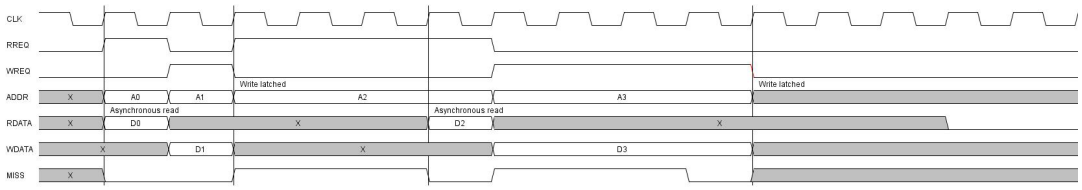

Figure 3.7: DCache Service Interface

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

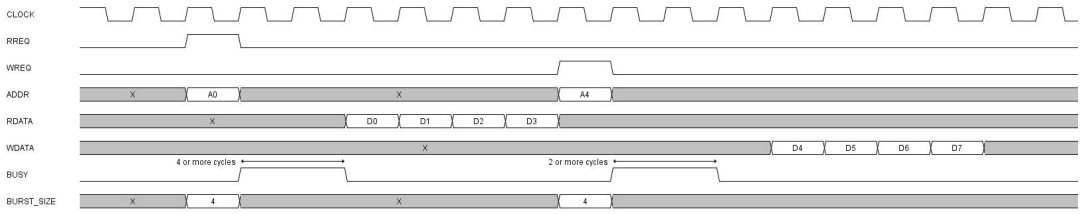

Figure 3.8: L2Cache Service Interface

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

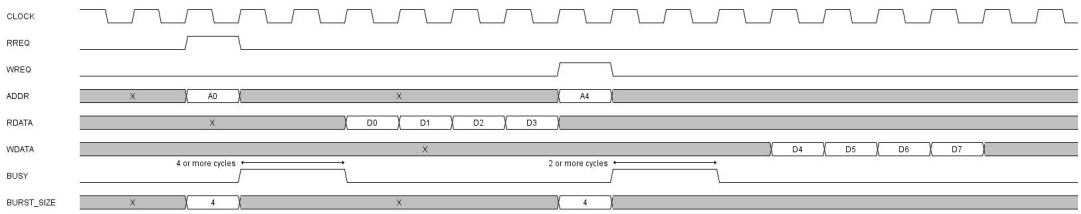

Figure 3.9: Cache Group Service Interface

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

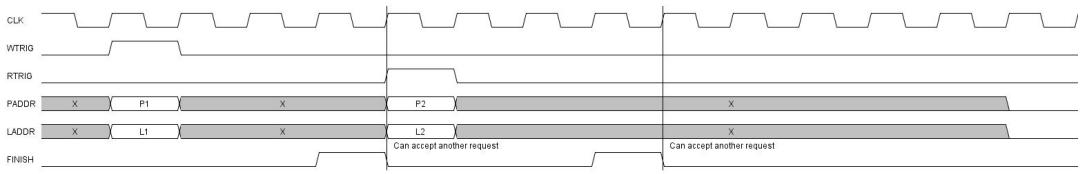

Figure 3.10: Load Manager Service Interface

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

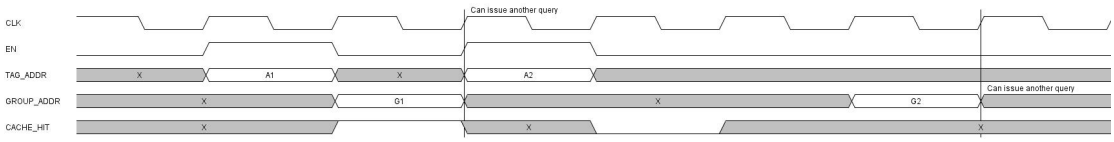

Figure 3.11: Device Timing Diagram

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

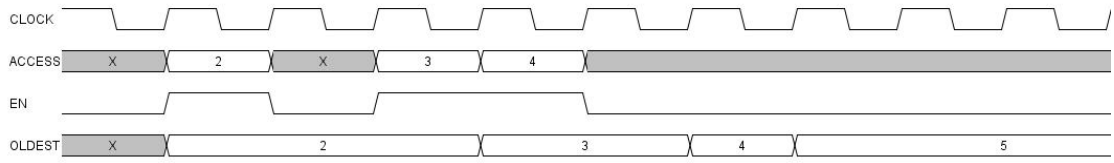

Figure 3.12: Device Timing Diagram

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：

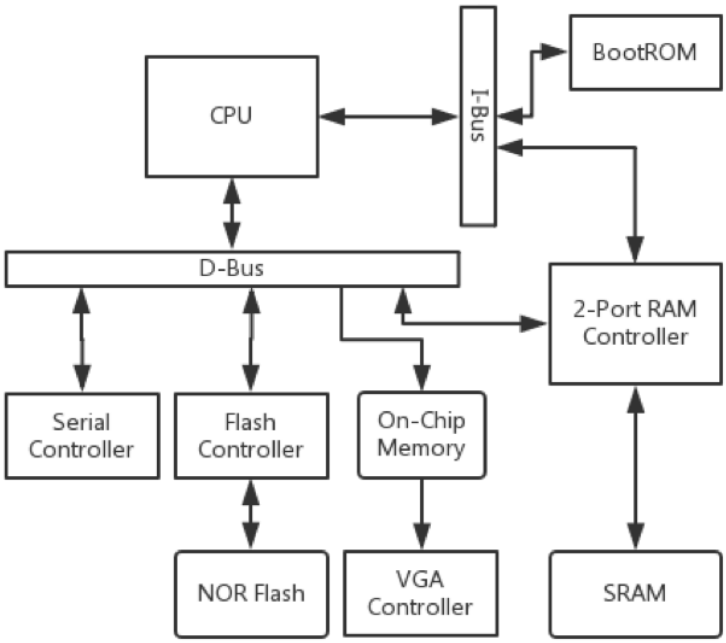

Figure 3.13: 系统框图（Basic 版本）

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:

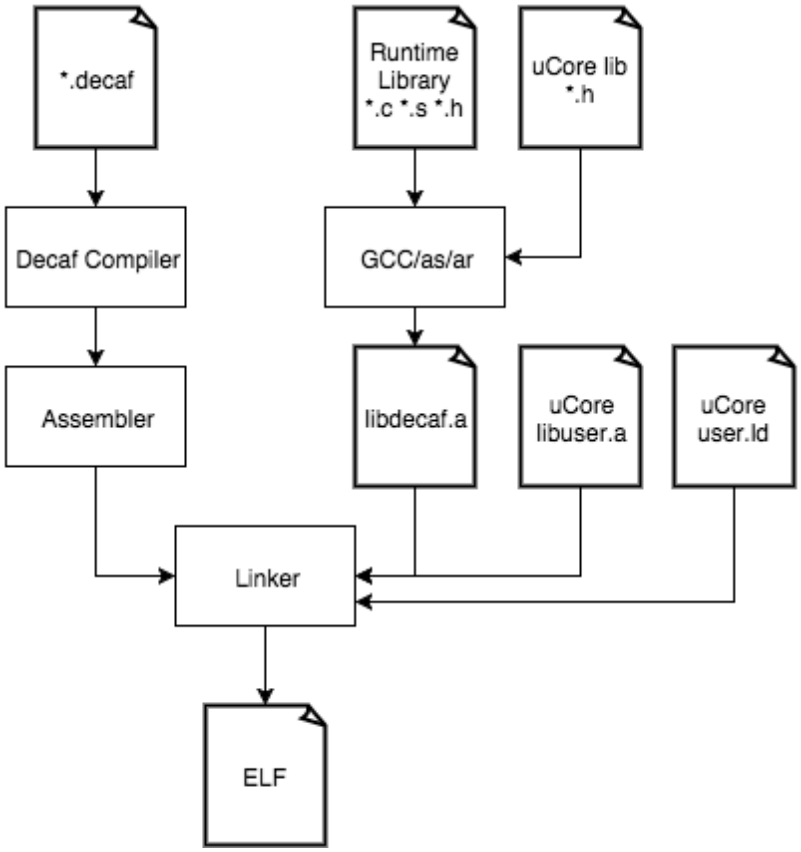

Figure 3.14: Compilation Procedure

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