NaiveMIPS 设计文档

第一届全国大学生计算机系统能力培养大赛
复赛提交

张宇翔　　王邈　　刘家昌

September 21, 2017

Table 1: 修订历史

时间	作者	内容更改
2015.11.18	张宇翔	CPU 及外设文档
2016.1.2	王邈	增加 TLB、GPU、性能测试程序部分
2016.1.15	张宇翔	系统软件说明
2017.8.10	张宇翔	CPU 设计更新，增加新的 SoC 设计
2017.8.17	刘家昌	Cache 设计，LCD 控制器设计
2017.8.19	张宇翔	U-boot 描述
2017.8.20	王邈	Linux 描述
2017.9.20	张宇翔	增加 VGA 控制器、PS/2 控制器说明
2017.9.21	张宇翔	增加图形加速器、I/O 控制寄存器说明

Contents

Chapter 1

文档说明

1.1 编写目的

本文档旨在描述 NaiveMIPS 处理器及其 SoC 的功能特点和内部设计。阅读本文档可以帮助开发人员了解本处理器的工作原理、对其进行功能扩展，或者集成进其他系统。文档假设阅读者已经熟悉 MIPS 体系结构，不对于 MIPS 本身作过多的阐述。文档中对于 MIPS 相关功能特征表述不清楚之处，以参考文献中列出的 MISP 规范为准。

1.2 术语定义

本文档中出现的术语缩写定义如下：

AHB 高级高性能总线

ALU 算数逻辑单元

APB 高级外设总线

AXI 高级可扩展接口

Cache 高速缓存

CAM 内容寻址存储器

CP 协处理器

CPU 中央处理器

DDR 双倍数据速率

Flash 快闪存储器

FPGA 现场可编程逻辑门阵列

GPIO 通用输入输出口

I/O 输入输出

ISA 指令集架构

JTAG 联合测试行动小组

LCD 液晶显示屏

MAC 介质访问控制器

MIPS 无内部互锁流水线微处理器

MMU 内存管理单元

Phy 物理层

RAM 随机访问存储器

ROM 只读存储器

SDRAM 同步动态随机访问存储器

SoC 片上系统

SPI 串行外设总线

SRAM 静态随机访问存储器

TLB 翻译后备缓冲区

Chapter 2

设计概述

2.1 MIPS 指令系统

2.1.1 设计特点

本项目的核心需求是设计一个部分兼容于 MIPS32 Release1 体系结构的 CPU，目标是在 CPU 的通用性和复杂性之前取得较好的平衡。因此该 CPU 实现了 MIPS32R1 规范中最常用的指令，以及运行操作系统所必要的 CP0 寄存器和异常。

处理器采用了经典的 5 级流水线设计，并支持 TLB 和一级指令、数据 Cache，从而提高系统的运行效率。TLB 设计为 16 路全相联结构，字段设计与 MIPS32R1 规范一致。一级 Cache 为直接映射结构，行数和大小可配置。

通过移植并运行 Linux Kernel、U-Boot、uCore 等较复杂的软件项目，我们在一定程度上验证了本处理器设计的通用性和正确性。

2.1.2 MIPS ISA

在本项目中，根据项目要求，CPU 支持的指令集为 MIPS32 指令集的子集，该指令集包含一下几种指令类型:

- 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

项目需求给出的指令集有 57 条指令，我们处于通用性考虑实现了更多的指令，覆盖 MIPS32 Release1 规范中所有 GCC 可能自动生成的指令（不启用 FPU 情况下），从而使得大部分的 MIPS 程序能够正确在 CPU 上正确执行。完整的指令集实现列表在附录 6.1中列出，指令的编码和含义严格遵守 MIPS32R1 规范。

CPU 中的 32 个通用寄存器，PC 寄存器，LO、HI 寄存器，按照 MIPS32 规范实现。在访存方面，支持各种尺寸的访存指令及 4 条非对齐访存指令，字节序固定为小端序。

从性能角度出发，CPU 设计为 5 级流水。CPU 中的控制逻辑配合数据通路设计，解决数据冲突、控制冲突和结构冲突，尽量减少流水线暂停的情况发生。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

考虑到实际系统中常常使用 DRAM 作为主存储器，其随机访问延迟较大，我们在 CPU 设计中包含了可选的一级 Cache 支持。在涉及到 DRAM 的系统中，可以引入 Cache 模块，降低平均访存时间。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。

L1 采用直接映射机制，访存的物理地址映射到固定的 Cache 行上。参数化的设计使得 L1 Cache 行的长度和数量均可在综合时灵活配置。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。L1 对外采用 AHB 总线接口，支持突发传输，提高总线利用率。

2.2 特权态资源

2.2.1 异常处理

为支持操作系统运行，CPU 支持异常处理。异常包括由于程序运行错误、TLB 缺失造成的异常，已经外部硬件信号触发的中断。由于采用流水线设计，要求必须支持精确异常处理。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。

本处理器支持的异常有：

1. **Int** 外部中断及系统定时器中断
2. **AdEL** 执行加载操作时地址非法
3. **AdES** 执行存储指令时地址非法
4. **TLBL** 执行加载操作时发生 TLB 缺失或无效
5. **TLBS** 执行存储指令时发生 TLB 缺失或无效
6. **Sys** 执行 SYSCALL 指令
7. **Bp** 执行 BREAK 指令
8. **RI** 解码时发现无效指令
9. **CpU** 在用户态使用特权指令
10. **Ov** 整数计算溢出

为方便系统程序开发，本处理器扩展实现了 MIPS32 Release2 描述的 CP0 的 **EBase** 寄存器，其复位值为 0x80000000。根据 MIPS 规范，异常处理程序基址采用如下方法计算。

CP0 $\text{Status}_{\text{BEV}}$	异常基址	备注
1	0xBFC00200	复位状态
0	CP0 Ebase	

不同的异常类型有不同的入口偏移量，在基址的基础上加上偏移量可计算得出异常的入口地址。偏移量如表所示：

异常类型	条件	偏移量	备注
中断	CP0 Cause_{IV}=0	+0x180	复位状态
	CP0 Cause_{IV}=1	+0x200	
TLB 缺失	CP0 $\text{Status}_{\text{EXL}}$=0	+0x0	
	CP0 $\text{Status}_{\text{EXL}}$=1	+0x180	
TLB 无效及其它异常		+0x180	

作为示例，当系统复位后，中断的入口地址为 0xBFC00380。

已实现的异常的处理流程与 MIPS32R1 规范一致。

中断还将受到一些 CP0 寄存器的影响。本处理器支持 6 个硬件中断，它们分别受到 CP0 中 $\text{Status}_{\text{IM7}}$ .. $\text{Status}_{\text{IM2}}$ 位控制，对应位为 1 时中断启用。另外支持两个软件中断，受到 CP0 中 $\text{Status}_{\text{IM1}}$ .. $\text{Status}_{\text{IM0}}$ 位控制，对应位为 1 时中断启用。

所有的中断可以被 CP0 的 $\text{Status}_{\text{IE}}$ 字段屏蔽，当其设为 0 时所有中断无效。此外，中断仅在处理器正常状态下（$\text{Status}_{\text{ERL}}$=0 且 $\text{Status}_{\text{EXL}}$=0）可以发生。

2.2.2 内存管理

处理器中的内存管理模块将把程序中使用的虚拟地址映射成物理地址送至总线接口。本处理器包含 TLB，可以实现动态内存映射，处于灵活性考量，是否启用 TLB 可以在综合时由参数配置。

MIPS32 规范中将虚拟地址划分为多个区间：

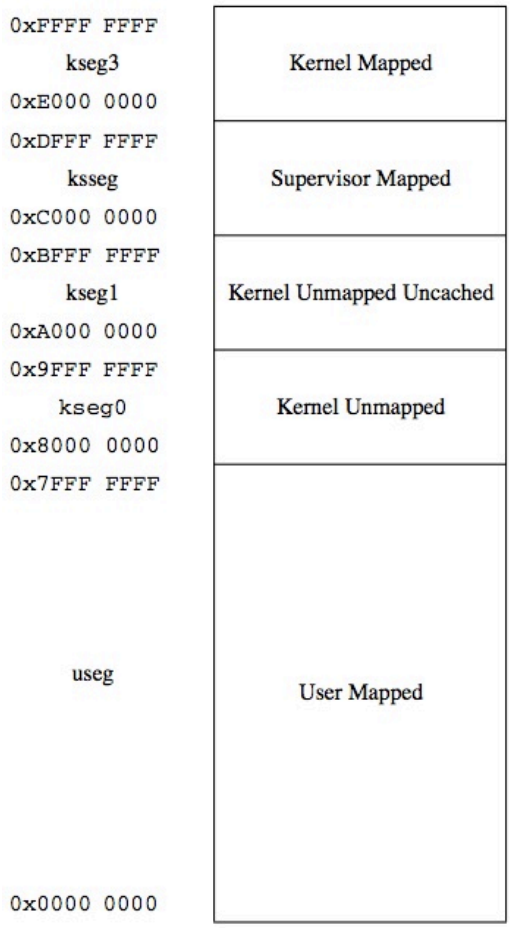

Figure 2.1: 虚拟地址段 [3]

各个区间向物理地址的映射规则如表 2.1所示

Table 2.1: 虚拟地址映射

虚拟地址段	映射方法（启用 TLB）	映射方法（不启用 TLB）
kuseg	由 TLB 映射	物理地址 = 虚拟地址
kseg0	物理地址 = 虚拟地址 - 0x80000000	物理地址 = 虚拟地址 - 0x80000000
kseg1	物理地址 = 虚拟地址 - 0xA0000000	物理地址 = 虚拟地址 - 0xA0000000
ksseg	由 TLB 映射	物理地址 = 虚拟地址
kseg3	由 TLB 映射	物理地址 = 虚拟地址

当采用 TLB 映射内存时，虚拟地址将会与 TLB 中的条目比较，发生匹配后可以得出物理地址。每一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
  - 全局标志（G）
  - 进程号（ASID）
  - 虚拟页编号（VPN）
- 物理翻译段
  - 物理页帧号（PFN）
  - 合法位（V）
  - 修改位（D）
  - Cache 控制位（C）

处理器设计时，物理地址和虚拟地址均固定为 32 位，内存页大小为 4KB。故地址的低 12 位作为页内偏移，不做转换，仅高 20 位由 TLB 转换。TLB 共有 16 项，紧邻的奇偶页存储在一项中，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。

本处理器亦实现了 TLBP 指令用于探测 TLB 条目。

2.2.3 CP0

本处理器对于 CP0 寄存器的实现，除了异常、TLB 相关的寄存器之外，还有系统定时器功能及提供处理器信息的若干寄存器。

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，系统定时器的计数值和匹配值分别由 CP0 寄存器 Count 和 Compare 保存。在本项目中定时器中断固定连接至 6 个硬件中断输入的最高位。

为了向操作系统提供必要的硬件信息，本处理器实现了 PRId、Config、Config1 寄存器，字段定义参考规范。

CP0 中实现的寄存器及其字段的完整信息在附录 6.2 中列出。

Chapter 3

详细描述

3.1 开发测试平台

3.1.1 硬件平台

NaiveMIPS 处理器经过真实硬件平台上运行验证，验证过的平台包括 DE2i 开发板、系统能力培养大赛实验箱及 Thinpad 教学实验板等。根据不同硬件平台上存储器、外设的配置不同，处理器周边的总线和接口控制器需进行相应的调整，但处理器本身代码不用修改。

系统能力培养大赛实验箱

该平台由龙芯公司设计制造，主要技术参数如下：

组件	数量	型号/参数
FPGA	1	Xilinx Artix-7 XC7A200T
DRAM	1	128MB DDR3 SDRAM
SRAM	1	128K × 8bit
NAND Flash	1	128MB
NOR Flash	1	1MB SPI
串口	1	
以太网控制器	1	DM9161 100M Ethernet Phy
VGA 接口	1	4bit × 3ch
PS/2 接口	1	
LCD	1	800 × 480
数码管	8	
LED	24+2	
拨码开关	8	
按钮开关	19	
晶振	1	100M

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

处理器核心代码不使用 FPGA 厂商的 IP 核，故兼容 Altera 和 Xilinx 两个平台。平台相关的代码均位于 xilinx/ 和 altera/ 两个目录中，共享的代码位于 src/ 目录下。

对于 Altera 环境，即 DE2i 开发板，使用的编译环境是 Quartus 15.1，仿真环境为 Modelsim-Altera。

对于 Xilinx 环境，即 Thinpad，使用的编译和仿真环境是 Vivado 2017.2。

处理器开发语言选择 Verilog HDL，部分 Testbench 用了 SystemVerilog 语言编写。

3.2 CPU

3.2.1 概述

NaiveMIPS 的 CPU 设计为 5 级流水，分别为取指 (IF)、译码 (ID)、执行 (EX)、访存 (MM) 和写回 (WB)。在取指阶段，PC 寄存器中指令地址经过 MMU 翻译为物理地址，送至指令总线接口。从总线返回的指令送入译码阶段，对指令内容进行解码，同时从寄存器堆中取出需要的通用寄存器值。译码阶段同时进行转移类指令的判断，如果发生转移则将新的地址送至 PC。解码后的指令变为内部代码，同寄存器值一并送至执行阶段，实际的算术逻辑运算在执行阶段发生。运算结果送至访存阶段，如当前是读写内存的指令，则在此阶段访问数据总线。此阶段中 MMU 将负责数据访存地址的翻译。此外，访存阶段还进行异常的判断和处理。运算结果或读取内存的值接下来送至写回阶段，它们在此阶段写入寄存器中。

CPU 中存在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。发现异常时将有相关逻辑关闭数据访存和后级的写回，确保出现异常的指令不改写内存和寄存器。

3.2.2 CPU 核心接口

CPU 模块的顶层设计在 **HDL/src/cpu/naive_mips.v** 文件中描述，该设计文件清晰地描述了 5 级流水线各个阶段之间的连接关系，属于各个阶段的信号命名用阶段的名称作为前缀来区分。

顶层模块对外的接口即为 CPU 核心接口。如果使用 Cache，则 CPU 核心接口连接至 Cache 模块，否则直接连接到总线互联上。核心接口采用指令与数据总线分离式设计，即哈佛结构，访问内存的冲突将由外部的总线互联仲裁。

接口信号描述如下。

Name	Width	Direction	Description
rst_n	1	In	CPU 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（恒为全 1）
ibus_read	3..0	Out	指令总线读使能
ibus_write	1	Out	指令总线写使能（未使用，恒为 0）
ibus_wrdata	31..0	Out	指令总线写数据（未使用，恒为 0）
ibus_rddata	31..0	In	指令总线读数据
ibus_stall	1	In	指令总线暂停请求
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	数据总线暂停请求
dbus_uncached_read	1	Out	数据总线读使能（对于标记为不缓存的内存地址）
dbus_uncached_write	1	Out	数据总线写使能（对于标记为不缓存的内存地址）
dbus_rddata_uncached	31..0	In	数据总线读数据（对于标记为不缓存的内存地址）
dbus_uncached_stall	1	In	数据总线暂停请求（对于标记为不缓存的内存地址）
dbus_dcache_inv_wb	1	Out	DCache 命中时写回控制信号
dbus_icache_inv	1	Out	ICache 命中时失效控制信号
dbus_iv_stall	1	In	Cache 写回或失效操作等待
hardware_int_in	4..0	In	外部设备中断信号输入（异步）

对于一次访存请求（指令、数据相同），CPU 置 read 或者 write 信号为 1，同时给出 address 及 dataenable，对于写操作还同时给出 wrdata。若从设备忙，则在同一周期置 stall 信号为 1，此时 CPU 将等待，保持控制信号不变，直到 stall 变为 0。当一个周期内 write 为 1 且 stall 为 0，表明一次写入成功；当一个周期内 read 为 1 且 stall 为 0，表明一次读取成功，读数据 rddata 在下一周期返回。

在启用 Cache 时，数据总线对于不可缓存的内存地址（如外设寄存器）访问时，将置 uncached_read 和 uncached_write 信号，而不是 read 或者 write 信号，它们可以绕过 cache 连接至总线互联，读数据从 rddata_uncached 接口返回，其余信号与有 cache 的接口共享。

当启用 Cache 时，CPU 对于 cache 控制指令还会给出几个两个控制信号。分别用于 DCache 的“命中时写回”和 ICache 的“命中时失效”操作，对应的地址由数据总线 address 信号给出。Cache 对于控制操作需要多个周期完成时，同样置 stall 信号使处理器等待。

3.2.3 CPU 结构

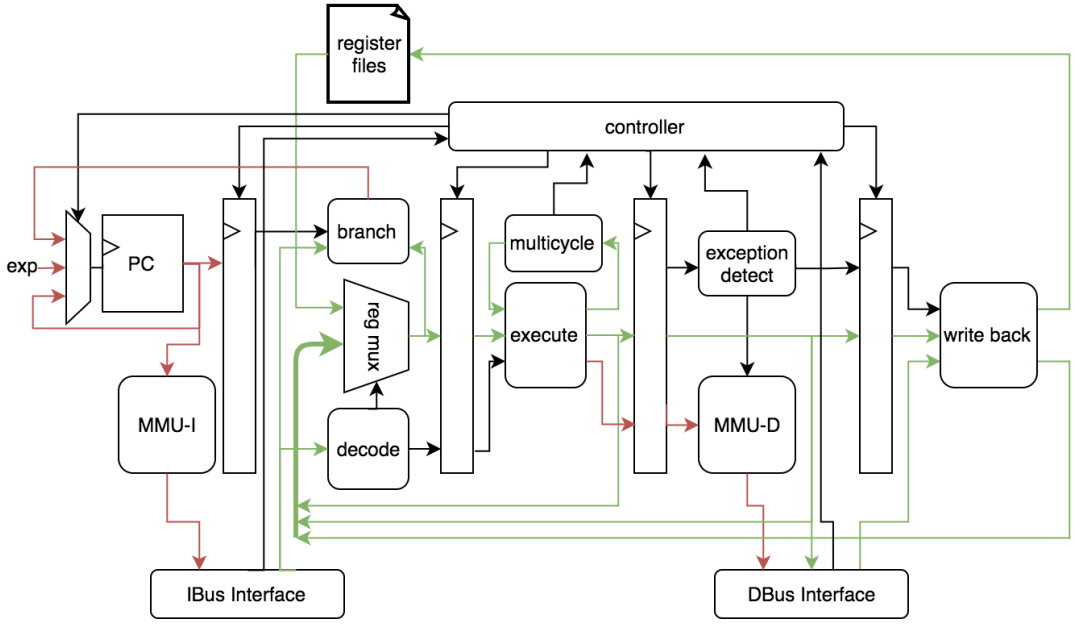

Figure 3.1: CPU 数据流图

CPU 的数据流图如 3.1 所示，其中地址信号用红色标记，数据用绿色标记。

如数据流图所示，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。流水线的数据冲突采用前推的方法解决，在译码阶段有多路选择器选择所使用的寄存器值来自哪个阶段。值得注意的是，并非所有数据冲突都能这样解决。对于加载类指令，其后紧邻的两条指令如果使用了读内存的结果，则不可能在译码阶段就获得结果，此时将由控制模块暂停流水线，待访存结果进入写回阶段后，再继续后续指令。

控制模块位于流水线外侧，控制流水线运行。其控制能力包括暂停流水线的任意阶段，和清空（复位）流水线中的数据。暂停流水线的原因包括来自指令、数据总线接口的暂停请求，执行阶段需要多周期的运算（如除法）未完成，无法靠数据前推来解决的数据冲突等。所有暂停流水线的情况由表 3.2.3 所描述。表中的条件由上至下依次判断，如果满足则暂停特定的阶段，均不满足则所有阶段正常运行。

条件	PC	IF-ID	ID-EX	EX-MM	MM-WB
数据总线暂停请求	暂停	暂停	暂停	暂停	暂停
多周期计算指令未完成	暂停	暂停	暂停	暂停	
执行阶段为 MFC0，且访存阶段为 MTC0 指令	暂停	暂停	暂停	暂停	
执行阶段为加载类指令，且目标寄存器与译码阶段访问寄存器相同	暂停	暂停	暂停		
访存阶段为加载类指令，且目标寄存器与译码阶段访问寄存器相同	暂停	暂停	暂停		
指令总线暂停请求	暂停	暂停	暂停		

异常检测模块会对指令执行过程中发送异常进行集中检测。指令在取指阶段、译码阶段和执行

阶段发现异常时不会立即处理，而是产生异常标记，随着流水线传递。到达访存阶段时，由一组合逻辑对所有标记进行检测，发现异常后，按照规范要求把异常信息写入 CP0 寄存器，并给控制模块传递异常信号，同时阻止当前的访存操作。控制模块中在收到异常信号后，立即对流水线各阶段进行清空操作，此时异常之前的指令已经完成写回，而之后的指令被清空，从而达到精确异常处理的要求。接下来，异常入口地址被写入 PC 寄存器，流水恢复工作，开始运行异常处理程序。

对于异常处理程序返回指令 ERET，由于需要的操作与异常处理相似，故在处理器内部也作为一种异常来处理，不过其对应的异常入口为 CP0 的 EPC 寄存器。

3.2.4 取指阶段分析

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，指令总线返回的数据即为要执行的指令，该指令被送入译码阶段。指令总线接口逻辑中包含一个地址暂存寄存器，当总线收到暂定请求时，物理地址被写入暂存寄存器中并保持不变，直到总线事务完成。这保证了一个总线事务的完整性，保证地址不会由于流水线其它阶段的操作（如写 TLB）而发生变化。

在指令地址转换为物理地址过程中，可能遇到 TLB 缺失，或者地址非对齐等异常，此时不对异常立即处理，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。遇到异常时也不访问总线取值，而是传给后级空操作指令 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	同步复位，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 或 PC 自增才生效
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

PC 真值表

rst_n	enable	is_exception	is_branch	PC 变化
L	X	X	X	PC ← 0xBFC00000
H	L	X	X	PC ← PC
H	H	H	X	PC ← exception_new_pc
H	H	L	H	PC ← branch_address
H	H	L	L	PC ← PC+4

3.2.5 译码阶段分析

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_taken	1	Out	是否发生分支
branch_address	31..0	Out	分支目的地址
return_address	31..0	Out	返回地址，用于写入 $31 寄存器
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 执行阶段分析

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。对于乘法指令，原则上可以在单周期完成，但考虑到延迟较大会影响主频，改为两周期实现。

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，

[1] http://opencores.org/project,divider

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。

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	2..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
cp0_sel	2..0	Out	CP0 Select 值
we_cp0	1	Out	CP0 写使能
we_tlb	1	Out	TLB 写入使能
tlb_probe	1	Out	TLB 探测使能
syscall	1	Out	是否为 SYSCALL 指令
break_inst	1	Out	是否为 BREAK 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 访存阶段分析

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 内部计算时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 节。此外，检测到地址非对齐异常时，模块将输出数据地址异常指示信号。

异常检查模块对于本阶段内发生异常，以及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，触发异常处理流程，同时输出信号给 CP0，记录异常的相关信息。异常检查模块会根据不同异常类型和 CP0 中相关字段的值，计算异常处理入口的地址，该地址被控制器模块送入 PC 寄存器。

中断作为一种特殊的异常，也在此阶段判断，但为了保证异步发生的中断不打乱流水线运行，本处理器设计时把中断信号嵌入流水线寄存器中，从上一阶段送入访存阶段，从而保证流水线暂停时中断跟着暂停。

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	2..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
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_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

exception 模块接口

Name	Width	Direction	Description
flush	1	Out	总异常发生指示
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 字段要写入的值
exp_code	4..0	Out	CP0 CODE 字段要写入的值
exp_asid	7..0	Out	CP0 ASID 字段要写入的值
exp_asid_we	1	Out	CP0 ASID 字段写使能
exp_bad_vaddr	31..0	Out	CP0 BadV 字段值
exp_badv_we	1	Out	CP0 BadV 字段写使能
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_invalid	1	In	指令地址 TLB 无效
daddr_exp_invalid	1	In	数据地址 TLB 无效
iaddr_exp_illegal	1	In	指令地址非对齐
daddr_exp_illegal	1	In	数据地址非对齐
daddr_exp_dirty	1	In	数据地址 TLB 脏标志
if_exl	1	In	指令访存时处于异常态
mm_exl	1	In	数据访存时处于异常态
if_asid	7..0	In	指令访存时的 ASID
mm_asid	7..0	In	数据访存时的 ASID
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
break_inst	1	In	Bp 异常
eret	1	In	Eret 伪异常
restrict_priv_inst	1	In	非法使用特权指令
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
interrupt_flags	7..0	In	软硬件中断标志
allow_int	1	In	当前状态中断允许发生
ebase_in	19..0	In	CP0 EBase 寄存器的值
epc_in	31..0	In	CP0 EPC 寄存器值

3.2.8 写回阶段分析

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。对于来自数据总线的信号，需要根据地址中的字节偏移量等信息，把读入的数据存储到寄存器合适的位置上。

wb 模块接口

Name	Width	Direction	Description
reg_we	1	Out	寄存器写使能
data_o	31..0	Out	写入寄存器的数据
mem_access_op	1..0	In	访存操作类型，见宏定义 ACCESS_OP_*
mem_access_sz	2..0	In	访存长度，见宏定义 ACCESS_SZ_*
data_i	31..0	In	待写入的数据，来自上一阶段
mem_data_i	31..0	In	来自数据总线的读数据
addr_i	31..0	In	数据总线的地址
exception_det	1	In	异常发生指示
flag_unsigned	1	In	访存指令为无符号
reg_addr_i	4..0	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 2.2.2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，其余地址根据是否启用 TLB，选择是用 TLB 转换还是直接映射。TLB 查表转换实现在 **tlbConverter** 模块中，该本模块本质上是一个 CAM，对于地址、ASID 进行比较，并输出匹配的表项。

TLB 为 16 项全相连接结构，每项均包含有效位、脏位、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。CPU 设计为 4K 固定页大小，所以虚拟地址的低 12 位直接成为物理地址的低 12 位，而高 20 位用于查表。

由于指令和数据访存可能同时需要 MMU 转换地址，MMU 中的模块实际上存在两个实例，分别为指令和数据访存服务，在修改时它们同步写入，从而保证内容一致。

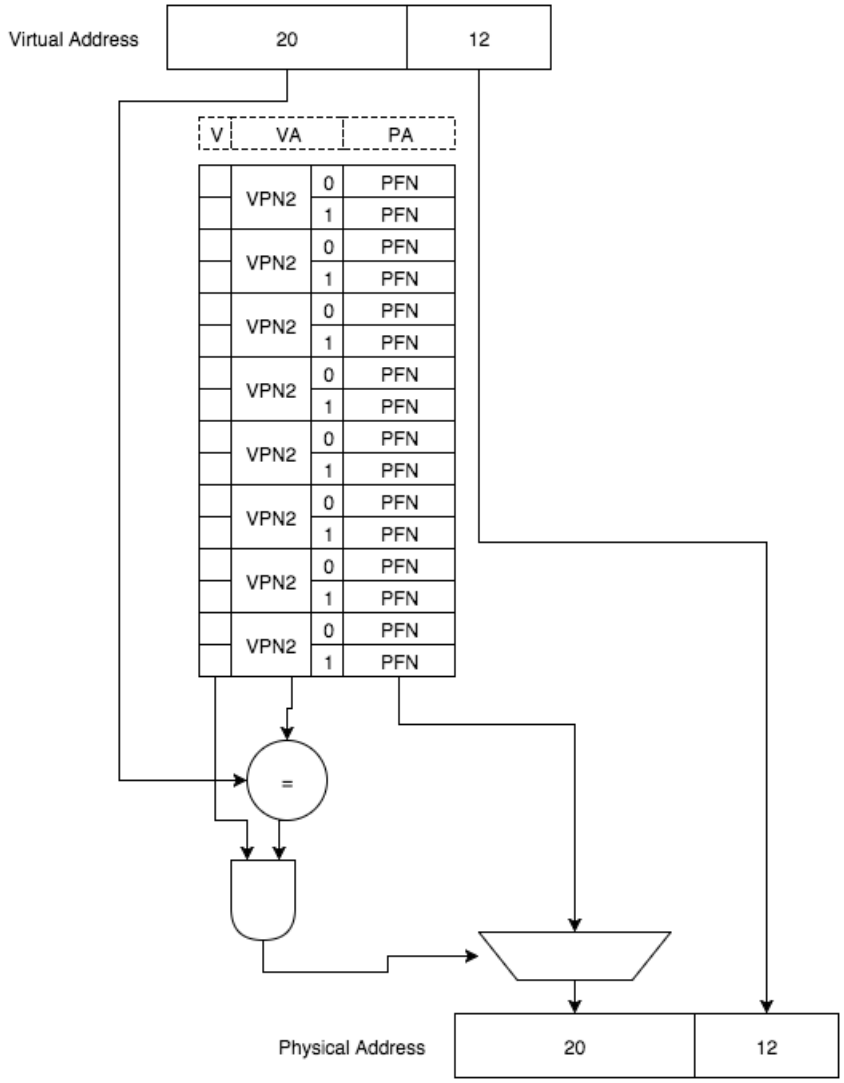

Figure 3.2: TLB 对于地址的转换流程

3.3 Cache

3.3.1 设计目标

设计 cache 的目的是为了提高 CPU 的性能，因此在满足硬件资源的前提下，使得 cache 的容量越大，设计的性能越高，则是所需的设计目标。

对于 cache 是否实现为一个的问题，基于之前的经验：将数据缓存（D-cache）与指令缓存（I-cache）分开实现为两个 cache，可以减少同一个 cache 行的竞争；也可以减少竞争条件，使得实现更加便捷。之前版本的 cache 由于实现为一个 cache，性能没有显著提高，并且复杂度显著提升了。

对于 cache 是否透明的问题：非透明的 cache 可以使得更加鲁棒。另外，CPU 需要能够在流水线的取指阶段使用指令缓存来获得下一条指令，同时也需要能够在随后的阶段对指令缓存进行「命中时失效（hit invalidate）」操作。CPU 需要能够根据指令，对于数据缓存进行「命中时写回（hit writeback）」操作。

对于 cache 是否写通（write through）模式，或是写回（writeback）模式：CPU 可能向 cache 发出非对齐访存，但必然在一个字内。但 cache 到内存总线的请求，必须是对齐的。因此，前期调研参考的 OpenRISC 实现方案并不可行，主要原因在于其一级缓存为写通模式，并且使用一个简单的队列写回队列（store buffer）来加快写回操作。由于非对齐访存模式会使得写通后，写回队列

的操作亦为非对齐访存，或是每次写操作需要先进行一次从内存的读操作，十分缓慢，这一模式是不可行的。

对于 cache 的下标（index）与标签（tag）使用物理地址或是虚拟地址的问题：使用虚拟地址虽然能够缩短周期内组合逻辑延时，但会导致高速缓存重影问题；使用虚拟地址作为下标，物理地址作为标签，可以同时完成 cache 行的索引和 TLB 转换，随后快速进行比对校验，但需要软件层面进行保护，复杂度也有所提升；使用物理地址作为下标和标签，则可能导致单周期内组合逻辑必须先通过 TLB，后通过 cache，并在这一周期内返回。由于已有代码的时延瓶颈不在于 TLB，这一查找可以很快完成，使用物理地址可以有效暂时避免高速缓存重影问题。过早优化可能导致问题，因此，在未来工作中如果有必要，可以再对这一方面进行优化。

对于单个 cache 行的大小和行数：由于 DDR3 单次爆发（burst）传输可以支持 64 字节，这一传输大小可以获得最高的效率。因此，在设计上希望一个 cache 行的读出/写回内存操作，能在 DDR3 的单次爆发传输过程中完成。单个 cache 行应当为 64 字节，或者说 16 个字。而在这一行大小的约束下，行数必须为二的幂次，且在线路时延允许的情况下，越大 cache 的性能越好。

对于 cache 的接口：在 cache 与原有 CPU 之间应当使用「总线设计-CPU 到 Cache」的 byte enable 的总线设计。在 cache 外应当使用「总线设计-Cache 到桥」的 AHB-lite 设计。其设计考虑在前文已经叙述。

对于 cache 的时序特点：系统设计中，最复杂的组合逻辑不在 CPU 到 cache 上，对这一部分的流水线化暂时不能使得每一周期花费的时间变短，从而时钟频率加快。相对的，如果没能合理地流水线化，由于 cache 的所有操作至少都需要两个周期，CPU 的效率反而会降低。流水线化也会带来很大的复杂度。因此，cache 需要在周期开始前，利用组合逻辑返回命中的读操作。但鉴于写操作需要至少一个时钟沿，命中的写操作可以使用一个周期。读出/写回内存的时序则依赖于总线的状态。

根据上文的分析，cache 需要设计为满足以下性质：

1. 指令缓存（I-cache）与数据缓存（D-cache）应当分别实现。
2. 实现一个对 CPU 不透明的典型 MIPS32 cache 机制，提供 cache 相关指令，即：
   (a) 指令缓存（I-cache）应当有独立的地址总线与控制线用于命中时失效（hit invalidate）。
   (b) 数据缓存（D-cache）应当有控制线用于命中时失效（hit writeback）。
3. Cache 需要是写回（writeback）模式的，以支持非对齐访存并转换为对齐访存。
4. 下标（index）与标签（tag）均使用物理地址。
5. 单个 cache 行应当为 16 字，行数必须为二的幂次，且在线路时延允许的情况下尽可能大。
6. 使用 byte enable 的总线设计与 CPU 连接，使用 AHB-lite 与桥通信。
7. Cache 的命中读应由组合逻辑在一个周期内完成；命中写在一个周期后完成。

3.3.2 指令缓存（I-cache）

接口设计

1. 全局信号

名称	信号源	描述
nrst	重置控制器	重置。唯一的**低**活跃信号，当信号值为**低**时，将会重置整个 I-cache 系统，包括清空 cache 内容，初始化状态机状态，中断一切正在进行的传输操作。这意味着 cache 期望与自己通信的所有系统也都由这一重置信号控制。
clk	系统时钟源	时钟。这一时钟同时控制了与 CPU 的总线传输，以及与 AHB-lite 的总线传输。所有的传输时序均在**上升**沿完成。

2. 服务信号

名称	源	描述
dbus_read	CPU	读指令。以上文描述的总线协议请求读操作。
dbus_rdaddr	CPU	读地址。读指令请求读的指令内容位置。
dbus_rddata	Cache	读数据。Cache 返回的读指令对应的地址指令数据。
dbus_rdstall	Cache	读拖延。当 Cache 需要从主存获得数据时，扩展等待周期。
dbus_hitinvalidate	CPU	命中时失效指令。以上文描述的总线协议请求命中时失效。
dbus_ivaddr	CPU	命中时失效地址。需要被命中时失效的缓存行地址。
dbus_ivstall	Cache	命中时失效拖延。当 Cache 需要将行标记为失效时，扩展一个周期。

3. 数据获取信号

名称	源	描述
AHB_haddr	Cache	AHB-lite 协议中定义的地址。32 位的系统地址总线，对应读取请求的位置。
AHB_hburst	Cache	AHB-lite 协议中定义的传输的类型，在 cache 中恒定为 16 节拍的定长递增（incrementing）爆发（burst）传输。
AHB_hprot	Cache	AHB-lite 协议中定义的总线权限信息，在当前项目中没有实际应用。因此恒定为 `4'b0011`。
AHB_hrdata	桥	AHB-lite 协议中定义的读取数据线。宽度为 32 位。
AHB_hready_in	Cache	AHB-lite 协议中定义的指示当前的传输完成状况，在 cache 中，被直接连接到 AHB_hready_out。
AHB_hready_out	桥	AHB-lite 协议中定义的当前的传输完成状况指示。如果为**低**，则 cache 对当前字的读取操作扩充一个周期。
AHB_hresp	桥	AHB-lite 协议中定义的错误指示。为**低**时，指示发生了一个错误。
AHB_hsize	Cache	AHB-lite 协议中定义的传输大小。在 cache 中输出恒定为字操作，若要获得更高速的传输，可以增大数据线宽并改变这一输出。
AHB_htrans	Cache	AHB-lite 协议中定义的当前的传输状态，在 cache 中可能依赖当前读写的状况，指示为空闲（IDLE）、非序列的（NONSEQ）和序列的（SEQ）。当 cache 未命中时，状态将由空闲变为非序列，开启一次 cache 行的读取工作；并在开始第二个字的读取时变为序列的。
AHB_hwdata	Cache	AHB-lite 协议中定义的写入数据线。宽度为 32 位。
AHB_hwrite	Cache	AHB-lite 协议中定义的传输的方向。当**高**时，操作为写。而时，操作为读。在爆发（burst）传输的过程中保持恒定。在当前 cache 中恒为**低**。
AHB_sel	Cache	AHB-lite 协议中定义的片选线。在传输开始时会被置**高**。

储存结构

指令缓存（I-cache）是一个 4096 字节大小的直接映射一级缓存，缓存结构如下：

- 指令缓存共有 64 个组。
- 每个组各有一个 cache 行。
- 每个缓存行各有 16 个字。

- 每个字有 4 个字节。

因此，地址的功能分区如下：

偏移量	31..12	11..6	5..2	1..0
用途	行标签	组地址	字地址	字偏移，被直接丢弃

其中，组地址用于寻找缓存行，字地址是缓存行内的字为单位的偏移量。当请求地址的行标签与缓存内储存的行标签相等时，缓存命中。字偏移由于一定是对齐到字，被直接丢弃。

缓存内，共实现 64 个缓存行，每一缓存行储存了储存的行的标签，行数据以及行有效位。具体实现参见缓存行（3.3.4）部分。

状态机

指令缓存（I-cache）分为两部分，一部分由组合逻辑构建，用于从 cache 中直接读取数据——在 cache 没有在进行内存读取操作时，若指令命中缓存，状态机不会发生变化，拖延线不会改变，指令数据将会直接返回；另一部分由时序逻辑构建，以状态机的形式构建，用于从支持突发传输的 AHB-lite 接口中读写数据，以及对 cache 行进行写入操作。共有四个状态如下：

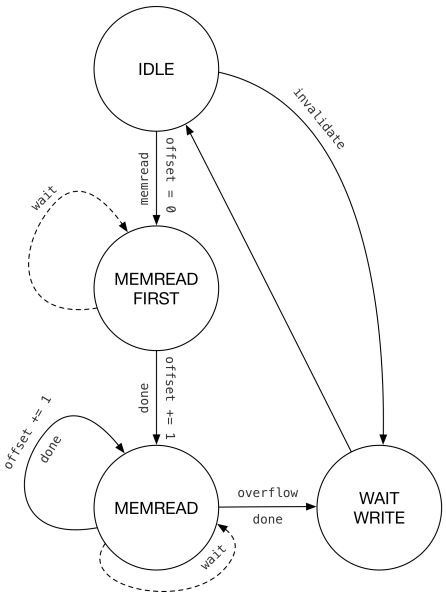

Figure 3.3: ICache 状态机

其中 `IDLE` 是唯一一个能够响应读命中的状态，代表 cache 元件没有在进行 cache 行写入工作或是交互。`MEMREAD_FIRST` 是读取第一个内存字的状态。`MEMREAD` 是读取后续内存字的状态。`WAIT_WRITE` 是对 cacheline 的请求刚刚发出，还未得到满足的状态。

当 cache 元件在 `IDLE` 状态下，接收到一个命中时无效请求时，会设定 cache 行的输入，在下一个周期写入 cache 的寄存器，使得对应行失效。随后切换到 `WAIT_WRITE` 等待 cache 寄存器写入。

当 cache 元件在 `IDLE` 状态下，接收到一个读请求时，会立刻使用组合逻辑电路判断请求的地址是否已经在 cache 内。若存在，则在同一周期内给出读取结果。若未命中或失效，则 `stall` 信号立刻拉高，在下一个上升沿启动内存读取时序逻辑，并保持读取请求的挂起。AHB-lite 第一个周期 `AHB_htrans` 将会是 `NONSEQ`，设定 `AHB_sel`，根据请求的位置设定读取的内存地址。首先读取第一个 offset = 0。对应的等待状态为 `MEMREAD_FIRST`。

当请求同时发生时，命中时无效请求优先于读请求，且在非 `IDLE` 状态下，cache 元件不响应任何请求。当同时命中时无效请求与读请求发生时，将会首先进行无效化后返回 `IDLE`，随后由于不命中或失效，从 `IDLE` 状态进入内存读取状态。

在 MEMREAD_FIRST 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。读取请求发送完成时，AHB_htrans 变为 SEQ，offset 递增。切换到等待状态 MEMREAD 的同时，设定 cache 行的输入，在下一个周期写入 cache 的寄存器。

在 MEMREAD 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。读取请求发送完成时，offset 递增，同时设定 cache 行的输入，在下一个周期写入 cache 的寄存器。在提交完最后一个字的读取请求后，AHB_htrans 变为 IDLE。如果整个 cache 行均已读取完毕，则设定 cache 行有效，并切换到 WAIT_WRITE 等待最后一个 cache 寄存器写入。

测试

在 Vivado 中利用模拟出来的 AS7C34098A RAM 设备和一个 ahb_slave，对 I-cache 进行测试。

具体流程为随机读与 invalidate 1 MByte 内存中的地址，在读与 invalidate 之前，直接修改 AS7C34098A 中的数据。进行 1000000 次随机操作，检查随机读出的数据是否与随机写入 AS7C34098A 中的数据相符。

I-cache 通过了这一测试。

3.3.3 数据缓存（D-cache）

接口设计

1. 全局信号

名称	信号源	描述
nrst	重置控制器	重置。唯一的**低**活跃信号，当信号值为**低**时，将会重置整个 D-cache 系统，包括清空 cache 内容，初始化状态机状态，中断一切正在进行的传输操作。这意味着 cache 期望与自己通信的所有系统也都由这一重置信号控制。
clk	系统时钟源	时钟。这一时钟同时控制了与 CPU 的总线传输，以及与 AHB-lite 的总线传输。所有的传输时序均在**上升**沿完成。

2. 服务信号

名称	源	描述
dbus_addr	CPU	地址。读指令请求读的内容位置，或是写指令请求写的内容位置，或是命中时失效指令请求被命中时失效的缓存行地址，或是命中时写回请求被命中时写回的缓存行地址。
dbus_byteenable	CPU	字节启用位。对于写指令有效，仅有这一数据线中指定的，对应字对齐地址上的字节/字节集会被 wrdata 中对应的字节/字节集替代。
dbus_read	CPU	读指令。以上文描述的总线协议请求读操作。
dbus_write	CPU	写指令。以上文描述的总线协议请求写操作。
dbus_hitwriteback	CPU	命中时写回指令。以上文描述的总线协议请求命中时写回。
dbus_hitinvalidate	CPU	命中时失效指令。以上文描述的总线协议请求命中时失效。
dbus_wrdata	CPU	写数据。当写指令生效时，cache 将这一内存数据根据 byteenable 的指示写入对应的地址的内容数据。
dbus_rddata	Cache	读数据。当读指令生效时，cache 返回对应的地址的内存数据。
dbus_stall	Cache	拖延。当 Cache 需要从主存获得数据，或者向主存写入数据时，扩展等待周期。

3. 数据获取信号

名称	源	描述
AHB_haddr	Cache	AHB-lite 协议中定义的地址。32 位的系统地址总线，对应读取或者写入请求的位置。
AHB_hburst	Cache	AHB-lite 协议中定义的传输的类型，在 cache 中恒定为 16 节拍的定长递增（incrementing）爆发（burst）传输。
AHB_hprot	Cache	AHB-lite 协议中定义的总线权限信息，在当前项目中没有实际应用。因此恒定为 `4'b0011`。
AHB_hrdata	桥	AHB-lite 协议中定义的读取数据线。宽度为 32 位。
AHB_hready_in	Cache	AHB-lite 协议中定义的指示当前的传输完成状况，在 cache 中，被直接连接到 AHB_hready_out。
AHB_hready_out	桥	AHB-lite 协议中定义的当前的传输完成状况指示。如果为**低**，则 cache 对当前字的读取或者写入操作扩充一个周期。
AHB_hresp	桥	AHB-lite 协议中定义的错误指示。为**低**时，指示发生了一个错误。
AHB_hsize	Cache	AHB-lite 协议中定义的传输大小。在 cache 中输出恒定为字操作，若要获得更高速的传输，可以增大数据线宽并改变这一输出。
AHB_htrans	Cache	AHB-lite 协议中定义的当前的传输状态，在 cache 中可能依赖当前读写的状况，指示为空闲（IDLE）、非序列的（NONSEQ）和序列的（SEQ）。当 cache 需要写回，或者未命中时，状态将由空闲变为非序列，开启一次 cache 行的写回或者读取工作；并在开始第二个字的写回或者读取时变为序列的。
AHB_hwdata	Cache	AHB-lite 协议中定义的写入数据线。宽度为 32 位。
AHB_hwrite	Cache	AHB-lite 协议中定义的传输的方向。当**高**时，操作为写。而时，操作为读。在爆发（burst）传输的过程中保持恒定。在写回内存时为**高**，执行写回操作。在进行内存读取时为**低**，执行读取操作。
AHB_sel	Cache	AHB-lite 协议中定义的片选线。在传输开始时会被置**高**。

储存结构

数据缓存（D-cache）是一个 4096 字节大小的直接映射，写回策略，按写分配的一级缓存，缓存结构如下：

- 指令缓存共有 64 个组。
- 每个组各有一个 cache 行。
- 每个缓存行各有 16 个字。
- 每个字有 4 个字节。

因此，地址的功能分区如下：

偏移量	31..12	11..6	5..2	1..0
用途	行标签	组地址	字地址	字偏移，被直接丢弃

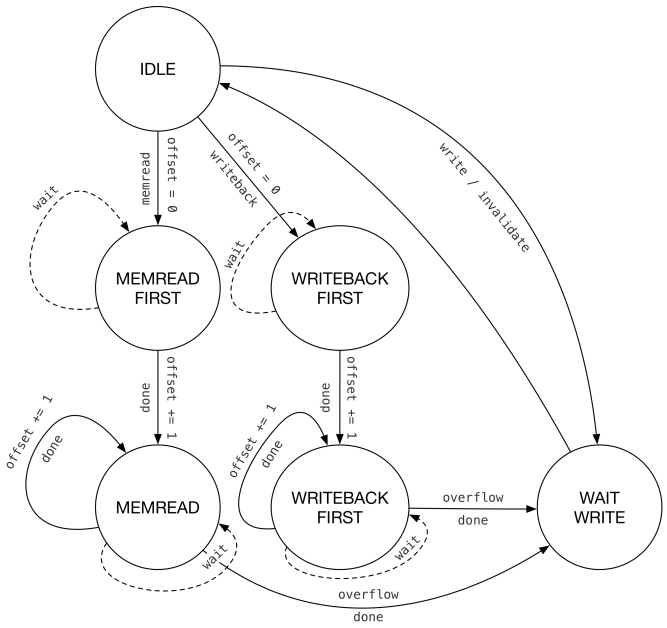

Figure 3.4: DCache 状态机

其中，组地址用于寻找缓存行，字地址是缓存行内的字为单位的偏移量。当请求地址的行标签与缓存内储存的行标签相等时，缓存命中。字偏移由于功能与 byte enable 重复，被直接丢弃。

缓存内，共实现 64 个缓存行，每一缓存行储存了储存的行的标签，行数据，行脏（dirty）位以及行有效位。具体实现参见缓存行（3.3.4）部分。

状态机

数据缓存（D-cache）分为两部分，一部分由组合逻辑构建，用于从 cache 中直接读取数据——在 cache 没有在进行内存读取操作时，若指令命中缓存，状态机不会发生变化，拖延线不会改变，指令数据将会直接返回；另一部分由时序逻辑构建，以状态机的形式构建，用于从支持突发传输的 AHB-lite 接口中读写数据，以及对 cache 行进行写入操作。共有六个状态如下：

其中 `IDLE` 是唯一一个能够响应读命中的状态，代表 cache 元件没有在进行 cache 行写入工作或是交互。`MEMREAD_FIRST` 是读取第一个内存字的状态。`MEMREAD` 是读取后续内存字的状态。`WRITEBACK_FIRST` 是写回第一个内存字的状态。`WRITEBACK` 是写回后续内存字的状态。`WAIT_WRITE` 是对 cacheline 的请求刚刚发出，还未得到满足的状态。

当 cache 元件在 `IDLE` 状态下，接收到一个读请求时，会立刻使用组合逻辑电路判断请求的地址是否已经在 cache 内。若存在，并且没有 cache 行需要写回，没有 cache 行需要读出，则在同一周期内给出读取结果。

当 cache 元件在 `IDLE` 状态下，请求的 cache 行需要写回：例如可能读操作中行未命中，但在对应的行有脏数据需要写回；写操作中行未命中，但进行按写分配的过程中，在对应的行有脏数据需要写回；或者行命中了，指令为写回或者无效，并且有脏数据需要写回。则 `stall` 信号立刻拉高，在下一个上升沿启动内存写回时序逻辑，并保持请求的挂起。AHB-lite 第一个周期 `AHB_htrans` 将会是 `NONSEQ`，设定 `AHB_sel` 以及 `AHB_hwrite`，根据请求的位置设定读取的内存地址。首先写入第一个 offset $= 0$。对应的等待状态为 `WRITEBACK_FIRST`。

当 cache 元件在 `IDLE` 状态下，请求的 cache 行需要读出：例如可能读操作中行未命中；写操作中行未命中，要进行按写分配。则 `stall` 信号立刻拉高，在下一个上升沿启动内存读取时序逻辑，并保持请求的挂起。AHB-lite 第一个周期 `AHB_htrans` 将会是 `NONSEQ`，设定 `AHB_sel`，清除 `AHB_hwrite`，根据请求的位置设定读取的内存地址。首先读取第一个 offset $= 0$。对应的等待状态为 `MEMREAD_FIRST`。

当 cache 元件在 `IDLE` 状态下，接收到一个写请求，并且没有 cache 行需要写回，没有 cache 行

需要读出时，会设定 cache 行的输入，在下一个周期写入 cache 的寄存器，使得对应行的内容被修改。随后切换到 WAIT_WRITE 等待 cache 寄存器写入。

当 cache 元件在 IDLE 状态下，接收到一个命中时无效请求时，并且没有 cache 行需要写回，没有 cache 行需要读出时，会设定 cache 行的输入，在下一个周期写入 cache 的寄存器，使得对应行失效。随后切换到 WAIT_WRITE 等待 cache 寄存器写入。

当任务同时发生时，写回任务优先级最高，随后是内存读取任务，再次是写请求和命中无效请求的处理。这是为了保证脏行在被覆盖前写回，以及写请求能够正确地被按写分配。在非 IDLE 状态下，cache 元件的组合逻辑不响应读命中。

在 MEMREAD_FIRST 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。读取请求发送完成时，AHB_htrans 变为 SEQ，offset 递增。切换到等待状态 MEMREAD 的同时，设定 cache 行的输入，在下一个周期写入 cache 的寄存器。

在 MEMREAD 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。读取请求发送完成时，offset 递增，同时设定 cache 行的输入，在下一个周期写入 cache 的寄存器。在提交完最后一个字的读取请求后，AHB_htrans 变为 IDLE。如果整个 cache 行均已读取完毕，则设定 cache 行有效且不脏，切换到 WAIT_WRITE 等待最后一个 cache 寄存器写入。

在 WRITEBACK_FIRST 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。写入请求发送完成时，AHB_htrans 变为 SEQ，offset 递增。切换到等待状态 MEMREAD 的同时，设定协议定义的的内存写入数据（第一个字）。

在 WRITEBACK 的等待仍然进行的过程中，状态不变，直到 AHB_hready_out 为**高**。写入请求发送完成时，offset 递增，同时设定新的内存写入数据。在提交完最后一个字的读取请求后，AHB_htrans 变为 IDLE。如果整个 cache 行均已写入完毕，则设定 cache 行不脏，并切换到 WAIT_WRITE 等待 cache 寄存器写入。

测试

在 Vivado 中利用模拟出来的 AS7C34098A RAM 设备和一个 ahb_slave，对 D-cache 进行测试。

具体流程为随机读/写/invalidate/writeback 1 MByte 内存中的地址，并维护一个正确的内存数据数组。进行 1000000 次随机操作，检查随机读出的数据是否与正确的内存数据数组中相符。

D-cache 通过了这一测试。

3.3.4 缓存行（Cacheline）

1. 全局信号

名称	信号源	描述
nrst	重置控制器	重置。唯一的**低**活跃信号，当信号值为**低**时，将会重置整个 cache 行系统的数据，行将会立即失效，储存的数据会被清零。
clk	系统时钟源	时钟。传输时序在**上升**沿完成。

2. 服务信号

名称	源	描述
rd_tag	缓存行	标签。当前缓存行中储存的内存的标签，见 I-cache 与 D-cache 的「储存结构」。
rd_off	外部	字地址。请求数据字的字地址，见 I-cache 与 D-cache 的「储存结构」。
rd_data	缓存行	数据。当前缓存行中，请求数据字的字地址上的数据。
rd_dirty	缓存行	脏。**高**当当前缓存行曾被写入，并且尚未写回。
rd_valid	缓存行	有效。**高**当当前缓存行中缓存的内容有效。
wr_write	外部	写入。**高**代表着下一个时钟**上升**沿，下面数据将会被写入缓存行中。
wr_tag	外部	写标签。当写入为**高**，缓存行的标签将会被替代为这个值。
wr_off	外部	写位置。当写入为**高**，标示写入数据字的字地址。
wr_data	外部	写数据。当写入为**高**，写入上述数据字地址上的数据。
wr_byte_enable	外部	字节启用位。对于写指令有效，字中仅有这一数据线中指定的字节/字节集会被 `wr_data` 中对应的字节/字节集替代。
wr_dirty	外部	写脏。当写入为**高**，缓存行的脏属性将会被替代为这个值。当本次写入为内存读取时，应为**低**；为用户写时，为**高**。
wr_valid	外部	写有效。当写入为**高**，缓存行的有效属性将会被替代为这个值。当本次写入为内存读取的完成时，应为**高**；为使失效，或是读取过程中时，为**低**。

缓存行（Cacheline）实现为一个带有 `byte_enable` 功能的寄存器。因此若需要其中个别属性不变化，应当与 `rd_*` 的对应数据相连后，设定 `wr_write`。

`rd_off` 的变化将会在同个周期内反馈到 `rd_data`，得到对应字的内容，由组合逻辑完成。`wr_write` 为**高**时，在下一个时钟上升沿，`wr_off` 上的内容将根据 `wr_byte_enable` 被替换为 `wr_data` 或其一部分；`wr_tag`、`wr_dirty`、`wr_valid` 将会替代原有的属性。

Chapter 4

实验箱上的 SoC 设计

4.1 总体设计

4.1.1 SoC 概况

为了更好地验证 NaiveMIPS CPU 的设计，并配合操作系统完成更多的功能演示，我们基于 NaiveMIPS 搭建了一个 SoC，支持在大赛实验箱上运行。本 SoC 的主要特性有：

- NaiveMIPS CPU，8KB 指令 Cache，16KB 数据 Cache
- 支持板载 DDR3 SDRAM 内存
- 64KB 片上 BootROM，存储一级引导程序
- 16KB 片上静态 RAM，用于引导程序等
- 支持软件读取 FPGA 配置 Flash
- 支持软件读写扩展 SPI Flash
- 16550 兼容串口控制器
- 100M 以太网通信
- GPIO 支持，软件控制 LED，读取开关状态
- VGA 及板载 LCD 屏幕图像输出
- PS/2 键盘输入设备支持
- 使用 u-boot 引导程序，支持 Flash 启动和 tftp 网络启动
- 可运行 uCore、Linux 操作系统

4.1.2 顶层设计

SoC 顶层设计工程为 **HDL/xilinx/NaiveMIPS/PrjVivao.xpr**，顶层采用 Vivado 设计工具提供的 Block Design 设计流程，即图形化的连线方式。SoC 总体框图如下图所示：

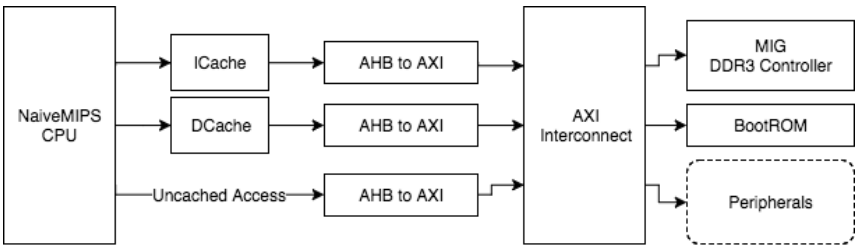

Figure 4.1: SoC 总体框图

考虑到硬件平台上 DDR3 内存的随机访问延迟较大，为了提高性能，必须引入 Cache 机制。因此，我们在 CPU 的指令、数据总线接口后均放置了一级 Cache，再与总线互联模块相连。为了支持不可缓存的访存操作，CPU 上另有一个 uncached 接口，绕过数据 Cache，直连总线互联模块。

SoC 在设计中使用了多个 Xilinx 提供的 IP 用于外设接口，它们对内均为 AXI 接口，故 SoC 选用的互联总线确定为 AXI 总线。由于 Cache 设计的对外接口为 AHB 接口，在 Cache 和 AXI 总线控制器之间我们使用了 Xilinx 的 AHB-Lite to AXI Bridge IP 进行协议转换。

SoC 的主存储为板载的 DDR3 内存颗粒，在 SoC 内部我们使用 Xilinx 的 MIG IP 实现 DDR 控制器。DDR 控制器再上电后会自动进行 DDR 物理层训练工作，完成后向用户逻辑给出信号。我们将该信号连接至 CPU 复位信号上，从而确保 CPU 运行时 DDR 内存已经可以访问。

当系统复位后，内存中的内容尚未初始化，因而需要一块固定的存储区域存放引导程序。这块区域我们称之为 BootROM，它使用 FPGA 片内的 Block RAM 资源构建，在 FPGA 硬件工程编译时就配置好内容，固化到硬件逻辑中，从而保证系统复位后 CPU 可以读取到启动程序。

AXI 总线互联模块上还连接了多个外设，它们主要实现 SoC 对外部硬件的接口控制，相关细节将在 4.2中描述。

4.1.3 地址映射

SoC 中的地址空间映射由 AXI 互联模块实现，其分配方案如表所示。地址分配考虑与之前的平台兼容等因素，并不完全连续。

Table 4.1: SoC 地址映射

外设名称	接口类型	基址	长度
MIG DDR 控制器	存储空间	0x00000000	128M
OCM 静态内存	存储空间	0x08000000	16K
配置 Flash 控制器	存储空间	0x1A000000	16M
以太网控制器	寄存器	0x1C030000	64K
LCD 屏控制器	寄存器	0x1C060000	4K
PS/2 控制器	寄存器	0x1C062000	4K
中断控制器	寄存器	0x1C080000	4K
VGA 控制器	寄存器	0x1C090000	4K
2D 图形加速器	寄存器	0x1C091000	8K
SPI Flash 控制器	寄存器	0x1E000000	4K
BootROM	存储空间	0x1FC00000	64K
I/O 控制寄存器	寄存器	0x1FD0E000	8K
串口控制器	寄存器	0x1FD02000	8K

4.1.4 中断信号连接

某些外设需要依靠中断机制通知 CPU，从而实现高效的 I/O 操作。MIPS 支持 8 个中断（编号 0 至 7），前两个是软件中断，最后一个保留给 CPU 内部的定时器，实际接到外设上的中断号为 2 至 6。由于 NaiveMIPS 只支持电平类型的中断，而某些外设输出的中断为边沿类型，故 SoC 中引入了一个 AXI 中断控制器，接收边沿类型的中断，输出电平类型的中断到 CPU。SoC 中各个中断的连接关系如表所示。

Table 4.2: SoC 中断信号连接关系

外设名称	中断类型	中断接收方	中断号
以太网控制器	上升沿	AXI 中断控制器	0
SPI Flash 控制器	高电平	CPU	3
串口控制器	高电平	CPU	4
AXI 中断控制器	高电平	CPU	5
PS/2 控制器	高电平	CPU	6

4.2 外设支持

4.2.1 以太网控制器

实验箱平台上板载了以太网 Phy 芯片但没有 MAC，需要在 FPGA 中实现 MAC，从而支持以太网通信。这里我们使用了 Xilinx 提供的简单版本的 10/100M 以太网控制器——AXI Ethernet Lite MAC。出于性能考虑，我们配置其通信模式为全双工，并启用接收、发送缓冲区。

控制器主机侧为 AXI 协议接口，直接连接至总线互联模块。另有一中断信号，类型为边沿触发，无法直接与 CPU 连接，故连接至中断控制器（4.2.2）。

控制器的对外接口为 MII 和 MDIO，其中 MDIO 为两线串行总线用于配置 Phy 芯片，而 MII 为同步并行数据接口，用于传输以太网数据。两者均为以太网协议规定的标准接口，可以直接与 Phy 芯片相连。

MII 接口在 100M 以太网通信时频率达到 25MHz，且需要与随路时钟信号同步，因此有必要添加时序约束。参考 Phy 芯片 DM9161A 数据手册 [9]，可知其发送接口的建立时间、保持时间分别为 12ns、0ns，因此 FPGA 侧 set_output_delay 的 max 和 min 分别为 12ns 和 0ns。Phy 芯片接收端口的时钟上升沿两侧各有至少 10ns 的区间数据有效，因此 FPGA 侧 set_input_delay 的 max 和 min 分别为 40-10 ns 和 10ns。

4.2.2 AXI 中断控制器

NaiveMIPS 只支持电平触发的中断，而有外设的中断类型为边沿类型，因此在 SoC 中我们增加了 Xilinx 提供的 AXI Interrupt Controller 作为扩展的中断控制器。在 Block Design 设计工具中，控制器 IP 核可以自动根据相连的中断源 IP 核获知其中断类型，因而不必手工配置各个输入通道。输出通道由于连接 CPU，需要配置为电平触发模式，且高电平有效。除中断信号线外，该控制器还有一个 AXI 接口，用于控制器内部寄存器访问，它连接至 AXI 总线互联。

4.2.3 NOR Flash 控制器

实验室上共有两片 SPI NOR Flash 芯片，一片连接至 FPGA 配置专用硬件逻辑，一片连接普通 I/O 引脚。我们实例化了两个 Xilinx 提供的 AXI Quad SPI 控制器，分别控制两片 Flash，两个控制器的参数配置有所区别。

板上的 SPI 配置 Flash 主要用于存储比特流，在上电时自动配置 FPGA。但是由于其容量较大，除了比特流之外还剩余了数兆字节的空间，可以用于存储启动引导程序。因此连接到 SPI 配置 Flash 的控制器被设置为工作在 XIP（原地执行）模式，从而使得 CPU 可以直接向 Flash 读出内容。在此模式下控制器为一个透明桥，且只可读取 Flash 内容，不支持写指令。另一方面，由于 SPI 配置 Flash 硬件上连着 FPGA 专用配置引脚上，不能像普通 I/O 引脚那样使用，因此还需要在 IP 核设置中启用“STARTUP Primitive”，才能正确连接 Flash。

板上另一片扩展用的 SPI Flash 我们将其作为通用目的，即可以在操作系统中读写，因此与它相连的控制器被设置为标准 4 线 SPI 控制器。这样软件可以直接向 Flash 芯片发送指令，并收发数据，实现完整的 Flash 读写功能。我们还启用了控制器中的缓冲区，以及中断信号，以提高通信效率。中断信号为电平触发类型，直接连接至 CPU。

两个 SPI Flash 控制器的主机侧数据接口均为 AXI 接口，在 SoC 中连接至总线互联模块。

4.2.4 串口控制器

串口控制器用于产生 RS-232 串行通信信号时序，在 SoC 设计中我们选择了 Xilinx 公司的 AXI UART 16550 核实现这一功能。该控制器可以实现完整的串行协议，支持流控和缓冲区。

串口控制器主机侧为 AXI 总线接口，对外为带有调制解调器信号的全功能串口。其中 AXI 接口在 SoC 中连接至 AXI 总线互联模块，对外接口我们只连接了 txd 和 rxd 信号，分别为串行发送和串行接收，其余信号线未连接。控制器有中断信号，直接连接至 CPU。

串口控制器输入时钟频率参数配置为 100MHz，按照 [4] 中的说明，波特率分频寄存器计算公式为 $\frac{100000000}{16\times Baudrate}$。

4.2.5 I/O 控制寄存器

I/O 控制寄存器提供一个总线至普通 I/O 口的接口，使得程序可以控制简单的输出设备（例如 LED），或读取简单的输入设备（例如开关）的状态，另一方面它还提供了一个定时器，用于 CPU 性能测试。该 IP 是用 Vivado 的 AXI 外设向导创建后，增加 I/O 接口和定时器实现的。

控制器支持多组输入输出，32 位输出信号经过译码后连接至数码管，16 位输出信号直接连接至 LED，8 位输入信号连接到拨码开关上，另有 4 位输出连接至两颗双色 LED。对于输入引脚，CPU 可以通过读取寄存器获得某个引脚的电平状态（即开关通断）；对于输出引脚，CPU 可以通过写寄存器设定某个引脚的电平状态（即控制 LED 点亮）。

控制器中的定时器寄存器是一个可读写的寄存器，读取寄存器可以获得当前计数值，写入寄存器可以设定当前计数值。在没有读写的情况下，计数器会在每个时钟周期自增 1。

寄存器分配由下表描述：

名称	偏移	描述
Timer	0x0000	[31:0] 定时器寄存器
LED	0x1000	[15:0] 16 位 LED 控制
RG0	0x1004	[0] 双色 LED0 绿色控制 [1] 双色 LED0 红色控制
RG1	0x1008	[0] 双色 LED1 绿色控制 [1] 双色 LED1 红色控制
NUM_DATA	0x1010	[31:0] 数码管控制
SW	0x1020	[7:0] 8 位开关输入量

4.2.6 LCD 控制器

LCD 控制器用于驱动板载的 LCD 接口，使得 CPU 可以控制 LCD 显示。该控制器为自己开发的 IP。

LCD 屏幕中包含数据寄存器和指令寄存器两个寄存器，LCD 控制器将两者分别映射到两个地址上，其中偏移量 0 为指令寄存器，偏移量 4 为数据寄存器。CPU 访问这两个地址，就能访问到 LCD 屏幕中的寄存器。LCD 控制器主机侧为 APB 接口，对外为 LCD 屏幕规定的专有接口。控制器信号定义如下：

名称	源	描述
nrst	复位控制器	复位。当信号值为**低**时，将会 LCD 系统。‘ LCD_nrst‘ 将会根据这一个值而进行复位。
clk	系统时钟源	时钟。传输时序在**上升**沿完成。
APB_PADDR	APB 桥	地址。这是 APB 的地址总线，其宽度为 32 位。设备仅检查后三位为 0 或是 4，分别代表支持的两个寄存器。
APB_PSEL	APB 桥	选择。显示器驱动将会从 APB 桥接受选择信号，指示设备被选中用于一次数据传输。
APB_PENABLE	APB 桥	使能。这一信号代表着 APB 传输的第二个以及之后的周期。
APB_PWRITE	APB 桥	方向。当信号为**高**时，进行一次写操作。**低**时进行一次读操作。
APB_PWDATA	APB 桥	写入数据。当 `PWRITE` 为**高**时，这一数据总线应当被要写入的数据填充。数据宽度可以为 32 位。
APB_PREADY	显示器驱动	预备。显示器驱动使用这一接口来扩充一次传输的时钟周期。
APB_PRDATA	显示器驱动	读出数据。显示器驱动将会在此放置读出的数据。数据宽度可以为 32 位。
APB_PSLVERR	显示器驱动	失败。显示器驱动将此与**低**短接，因为显示器的寄存器不会失败。
LCD_nrst	APB 桥	复位。复位 LCD 显示屏，与 `nrst` 直接连接。
LCD_csel	APB 桥	选择。选择显示器，指示一次指令正在进行。
LCD_rs	APB 桥	地址。选择 0（指令）寄存器，或是 1（数据）寄存器。
LCD_wr	APB 桥	写指令。当信号为**低**时，进行一次写操作。
LCD_rd	APB 桥	读指令。当信号为**低**时，进行一次读操作。
LCD_data	输入输出	数据输入输出。在对外接口上表现为 GPIO，在设备内部为输入输出线加一条控制线。

显示屏接口为异步接口，具体时序关系如下：

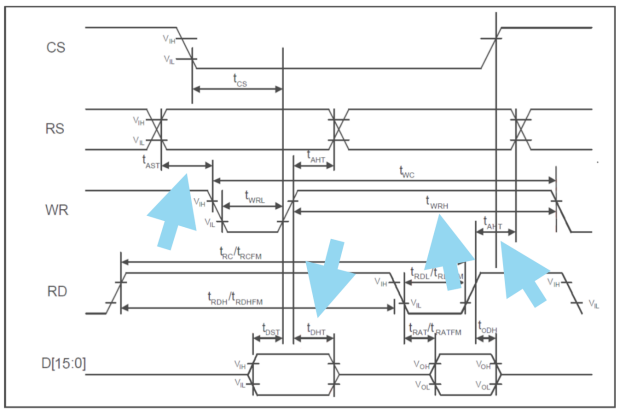

Figure 4.2: LCD 屏幕接口时序

图中标记的是需要特别关注的几个时间要求。

LCD 控制器的主机侧接口经过一个 AXI to APB Bridge 连接至 AXI 总线互联，对外接口直接引出至 FPGA 引脚与 LCD 屏幕相连。

4.2.7 VGA 控制器

VGA 控制器用于将存储在帧缓冲中的图像数据按照一定的时序输出到 VGA 接口上。我们选择 Xilinx 公司的 AXI Thin Film Transistor Controller IP 提供这一功能。该 IP 内部用一个 DMA 不断地从内存中获得图像数据，再按照 VGA 的时序输出。IP 支持固定的分辨率 640x480，8 位色彩深度，我们按照实际硬件接口宽度将像素数据的高位输出。

控制器另有一个 AXI 控制接口，连接到 SoC 总线互联上，所有控制寄存器读写经由该接口进行。操作系统的帧缓冲驱动程序会将分配到的缓冲区地址写入 VGA 控制器的寄存器，并启动控制器传输。

Chapter 5

系统软件

5.1 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。

NaiveBootloader 通常作为一级引导程序，用于启动下一级引导程序（如 U-Boot），亦可直接启动简单的操作系统（如 uCore）。

5.1.1 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）。即使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 时就从 Flash 中加载系统的 elf 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。

5.1.2 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`

CFI Flash 访问测试 `serial_load.py -s <dev> --test flash`

CFI Flash 写入 `serial_load.py -s <dev> -p <file.bin>`

加载 ELF 文件至内存并运行 `serial_load.py -s <dev> -l <file.elf> --term`

5.2 uCore

NaiveMIPS 使用的 uCore 是在原版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。

5.2.1 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 5.3.0，其配置选项如下：

```
../configure --target=mipsel-unknown-linux-gnu --enable-languages=c CC=clang CXX=clang++
```

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 2.26，配置选项：

```
./configure --disable-debug --disable-dependency-tracking  --disable-werror \
--enable-interwork --enable-multilib --enable-64-bit-bfd --enable-targets=all
```

5.2.2 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-fno-delayed-branch），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
```

5.2.3 内存管理

原版 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,由于我们根据物理内存大小,将 **kern/mm/memlayout.h** 中针对 MACH_FPGA 的 **KMEMSIZE** 宏定义修改为实际值。

5.2.4 U-Boot 镜像

为了支持通过 U-Boot 网络启动 uCore 操作系统，我们对 uCore 的编译流程作了一下修改，使之可以直接生成 uImage 格式的镜像。

原版 uCore 的入口点位于物理内存起始位置，由于 uImage 镜像带有 64 字节的头部，因此需要将入口点后移 64 字节。这需要修改 **tools/kernel.ld** 文件中**.text** 节之前的起始地址声明。从“**. = 0x80000000;**”改为“**. = 0x80000040;**”。

另一方面，修改 Makefile，使之可以容易的生成 uImage 格式的 uCore 启动镜像。在 Makefile 文件中新增规则：

```
$(OBJDIR)/ucore.ub: $(OBJDIR)/ucore-kernel-initrd
    $(OBJCOPY) -O binary -v $^ $(OBJDIR)/ucore.bin
    mkimage -A mips -O u-boot -T kernel -C none -a 0x80000040 -e 0x80000040 \
    -n ucore-thumips -d $(OBJDIR)/ucore.bin $@
```

这样就可以支持 `make obj/ucore.ub` 命令来生成 uImage 格式的 uCore 启动镜像 **obj/ucore.ub**。

5.2.5 串口驱动

uCore 使用串口驱动虽然也是 16550 兼容的串口控制器驱动，但是其寄存器大小均为 8 位，相邻寄存器之间的地址相差 1。而 Xilinx 的 16550 控制器的寄存器均为 32 位大小（仅低 8 位有效），相邻寄存器之间的地址相差 4。因此在 **kern/driver/console.c** 中所有串口控制寄存器的偏移量都需要左移两位（即乘 4）。

串口波特率固定设置为 115200，16550 的 DLL 寄存器根据串口控制器手册计算值，参加 4.2.4。

5.3 U-Boot

U-Boot 是一个启动引导程序，常见于嵌入式系统中，用于引导 Linux 等操作系统。在基于 NaiveMIPS 的 SoC 上，运行 U-Boot 引导程序，支持从 Flash、网络等来源加载 Linux、uCore 系统镜像并引导。

在实践中，U-Boot 通常作为二级引导程序，放置在外部存储设备中。CPU 复位后首先执行一级引导程序（NaiveBootloader），一级引导程序将 U-Boot 复制到内存后，跳转到 U-Boot 入口地址，U-Boot 开始运行。

5.3.1 开发环境

U-Boot 主要开发工具为 GCC 交叉编译工具链，包括 GCC 及 Binutls。在开发时我们使用了 Ubuntu 16.04 操作系统提供的软件包“gcc-mipsel-linux-gnu”和“binutils-mipsel-linux-gnu”。

5.3.2 移植过程

我们选用 U-Boot 2017.7 版本作为基础，添加 NaiveMIPS 平台和板级支持代码，源代码来源于 U-Boot 主线[1]。主要的移植步骤如下。

添加 SoC 平台

在源码树的 **arch/mips** 下面是整个 MIPS 平台的支持代码，我们需要向其中添加一款新的芯片，即 NaiveMIPS。这主要涉及到修改或添加几个文件。

arch/mips/Kconfig 添加 MACH_NAIVEMIPS 选项，描述平台的基本特征

arch/mips/Makefile 针对 NaiveMIPS 平台的特定编译选项

arch/mips/mach-naivemips/Kconfig NaiveMIPS 支持的电路板列表

NaiveMIPS 处理器与 MIPS32 Release1 规范最接近，因此我们在 MACH_NAIVEMIPS 选项中 select SUPPORTS_CPU_MIPS32_R1，从而复用 U-Boot 中对于该规范已有的实现代码。

同时由于指令实现上的差异,在编译选项中,针对本平台添加`-mno-branch-likely`和`-mdivide-breaks`，这是为了避免产生未实现的 Branch Likely 和 TEQ 指令。

添加板级描述

在新的 SoC 平台基础上进一步增加使用该平台的电路板描述和外设驱动描述。以大赛实验箱平台为例，该步骤主要涉及到修改或添加几个文件。

arch/mips/mach-naivemips/Kconfig 平台支持的电路板列表中添加 TARGET_NAIVEMIPS_NSCSCC

board/nscscc/Kconfig 实验板的基本信息描述

board/nscscc/FPGA-A7/ 新建实验板的代码目录

board/nscscc/FPGA-A7/Makefile 实验板的代码编译控制文件

board/nscscc/FPGA-A7/naivemips_nscscc.c 实验板初始化代码

arch/mips/dts/naivemips_nscscc.dts 实验板设备树文件

include/configs/naivemips_nscscc.h 实验板配置类宏定义

configs/naivemips_nscscc_defconfig 实验板默认配置

当用户在构建时，通过 menuconfig 工具选择实验板时，**board/nscscc/FPGA-A7/**目录下面相关的代码将被编译，成为板级支持代码，负责电路板相关的初始化工作（如探测内存大小）。由于实验板比较简单，初始化函数基本为留空。

dts 设备树文件和 **include/configs/naivemips_nscscc.h** 头文件中的宏定义，共同提供了电路板具体的硬件信息。这些信息包括外设的型号、地址映射、中断连接及存储空间等。头文件中还包含默认的 U-Boot 环境变量，利用这些环境变量可以实现自动启动等功能，避免每次上电都要求用户手工输入启动命令。

由于 U-Boot 具有高度的可裁剪性，内部大多数功能模块都可以在构建时通过 menuconfig 工具进行选择。我们将最适合本实验板的选项保存到 **configs/naivemips_nscscc_defconfig** 中，便于新用户快速完成构建工作。

[1] https://github.com/u-boot/u-boot

驱动移植

SoC 及板上各类外设都需要驱动程序的支持，U-Boot 主线中已经自带大量驱动程序，但是仍不完善。因此我们需要根据需求移植部分驱动程序。

在 NaiveMIPS 的 SoC 中我们使用了 Xilinx 的 SPI Flash 控制器，其在主线 U-Boot 中的驱动功能较弱。因此我们从 Xilinx 公司维护的 u-boot-xlnx 项目[2]中提取这部分驱动代码，**drivers/spi/xilinx_spi.c**，覆盖主线代码中同名文件。

5.3.3 构建方法

从我们维护的 u-boot-naivemips 项目[3]下载完整源代码。

进入顶层目录，用如下命令选择默认的构建配置选项并构建 U-Boot。

```
make CROSS_COMPILE=mipsel-linux-gnu- naivemips_nscscc_defconfig
make CROSS_COMPILE=mipsel-linux-gnu-
```

在构建成功完成后，顶层目录中会生成 **u-boot** 文件，即为 ELF 格式的 U-Boot 程序，该文件可以由 NaiveBootloader 运行。

5.4 Linux

5.4.1 指令与 CP0 功能依赖精简

我们对内核进行了功能精简，从而使得硬件上可以少实现一些功能。内核提供的 feature 去除机制，在 Linux 内核的 `arch/mips/include/asm/mach-*/` 目录下,可以放置一个 `cpu-feature-overrides.h` 文件,提供平台不支持的功能说明。可以去除的 `feature` 参考 `arch/mips/include/asm/cpu-features.h` 中的宏定义，要去除就 define 为 0。例如 `#define cpu_has_watch 0` 可以去掉内核对 CP0 Watch 寄存器的依赖。

- `ll/sc` 指令去除：`ll/sc` 指令用于实现变量的原子访问，在 NaiveMIPS 中没有实现，故通过 `#define cpu_has_llsc 0` 告诉内核不用 `ll/sc`。在没有 `ll/sc` 的情况下，可以看到 `arch/mips/include/asm/atomic.h` 文件中，采用了开关全局中断的方法来实现原子操作。事实上，不支持 `ll/sc` 指令的处理器还有 mach-ath25，其注释中指出芯片的 `ll/sc` 有问题，不能使用。
- WatchLo 和 WatchHi 寄存器去除：Watch 寄存器用于内存断点实现。在 ptrace 等机制中会用到（参见 `ptrace_set_watch_regs` 函数）。NaiveMIPS 没有实现，故通过 `#define cpu_has_watch 0` 去除。
- `break` 和 `tne` 指令去除:MIPS 架构自己实现了 `BUG` 和 `BUG_ON`（在 `arch/mips/include/asm/bug.h` 文件中），用到了 Trap 机制，即 `break` 和 `tne` 指令，Trap 机制硬件上没有实现。因此直接注释掉指令，使用 generic 的 `BUG()` 实现。
- Branch Likely 类指令去除：在打开 optimize for size（`CONFIG_CC_OPTIMIZE_FOR_SIZE`）后，编译器产生了 Branch Likely 类指令（见 MIPS 规范中的描述），用于优化分支的效率。由于硬件上没有实现，于是在 `arch/mips/Makefile` 中增加一行 `cflags-$(CONFIG_MACH_THINPAD) +=-fno-delayed-branch`，使得编译选项中增加 `-fno-delayed-branch` 避免产生这些指令。

[2]https://github.com/Xilinx/u-boot-xlnx
[3]https://git.net9.org/zhangyx13/u-boot-naivemips

5.4.2 TLB 依赖精简

内核中与 MIPS32 TLB 相关的代码位于 arch/mips/mm/tlb-r4k.c，arch/mips/mm/tlbex.c 文件中，修改其中的逻辑，去除了内核对 Random、Wired 寄存器，以及 TLBWR 指令的依赖。详细修改如下：

此处修改的目的是为了去除内核对于 TLBWR 指令的依赖。TLBWR 意为随机写入一条 TLB 表项。为了去除对该指令的依赖，我们采取了使用软件生成该随机数，使用 TLBW 指令写入 TLB 的做法。

需要修改的地方有两处，一处在 arch/mips/include/asm/mipsregs.h，将 tlb_write_random 换成

```
static inline void tlb_write_random(void)
 {
/*  __asm__ __volatile__(
        ".set noreorder\n\t"
        "tlbwr\n\t"
        ".set reorder");
*/

  static int random = 0;
  static int numTlbs = -1;
  unsigned int wired;
  if (unlikely(numTlbs == -1)){
    numTlbs = ((read_c0_config1() & MIPS_CONF1_TLBS) >> 25) + 1;
  }
  wired = read_c0_wired();
  if (random < wired){
    random = wired;
  }
  if (random >= numTlbs){
    random = wired;
  }
  write_c0_index(random);
  mtc0_tlbw_hazard();
  tlb_write_indexed();
  random ++;
}
```

一个在 arch/mips/mm/tlbex.c，补充

```
#define C0_WIRED    6, 0
#define C0_RANDOM   1, 0
```

将 build_r4000_tlb_refill_handler() 中对 build_tlb_write_entry(&p, &l, &r, tlb_random) 的调用替换为：

```
//build_tlb_write_entry(&p, &l, &r, tlb_random);
uasm_i_lui(&p, K0, ((int)(&__random) >> 16));
uasm_i_addiu(&p, K0, K0, (int)&__random & 0xffff);
uasm_i_lw(&p, K1, 0, K0);
uasm_i_xori(&p, K1, K1, 1);
uasm_i_sw(&p, K1, 0, K0);
```

```
uasm_i_mfc0(&p, K0, C0_WIRED);
uasm_i_addu(&p, K0, K1, K0);
uasm_i_mtc0(&p, K0, C0_INDEX);
uasm_i_nop(&p);
uasm_i_nop(&p);
build_tlb_write_entry(&p, &l, &r, tlb_indexed);
```

5.4.3 板级设备描述

为了使 Linux 内核知道实际的硬件配置（如总线地址映射），需要实现板级设备描述代码。这些代码分 3 部分：

- 早期初始化代码：位于 `arch/mips/thinpad/init.c`，实现板级的早期初始化，同时提供早期调试信息输出函数 `prom_putchar`。在串口驱动加载之前，`printk` 打印的调试信息最终均通过 `prom_putchar` 函数输出。
- 平台中断控制器代码：位于 `arch/mips/thinpad/intc.c`，通过宏 `IRQCHIP_DECLARE` 将中断控制器的初始化调用挂钩至系统初始化过程。
- 设备树描述:位于 `arch/mips/boot/dts/thinpad/` 目录中,有 `naivemips_de2i.dts`、`naivemips_thinpad.dts`、`naivemips_nscscc.dts` 三个，分别对应三种硬件平台。在该目录下的 `Makefile` 中,会根据内核配置选择将哪一个文件编译进内核。设备树（dts）用于描述硬件连接关系（时钟、地址、中断号）等信息,各个条目含义参考 `Documentation/devicetree/bindings/` 目录中对应各个驱动程序的说明。
- 默认内核配置：位于 `arch/mips/configs/naivemips_nscscc_defconfig` 文件，提供初始内核配置，详见 5.4.4 一节。

5.4.4 使用说明

下载内核源码源码后，进入代码顶层目录，用如下命令生成适用于 NaiveMIPS 的初始内核配置：

```
make ARCH=mips CROSS_COMPILE=mipsel-linux-gnu- naivemips_nscscc_defconfig
```

上述命令将产生 `.config` 内核配置文件。如需编辑配置，使用命令：

```
make ARCH=mips CROSS_COMPILE=mipsel-linux-gnu- menuconfig
```

打开配置界面后，进入第一项 `Machine selection` 菜单后，进入 `Machine type` 可以切换 Thinpad 、DE2i 和 NSCSCC 硬件平台。

配置完成后，开始编译：

```
make ARCH=mips CROSS_COMPILE=mipsel-linux-gnu- uImage -j4
```

其中`-j4` 是指用 4 线程编译，可以根据实际情况调整。

编译成功后将产生 `vmlinux` 文件，即整个内核的 elf 格式可执行文件。该文件可以通过 NaiveBootloader 上位机工具引导内核。同时将产生 `arch/mips/boot/uImage`，该文件可以用于 U-Boot 引导。

Chapter 6

附录

6.1 NaiveMIPS 指令集

本节列出 NaiveMIPS 处理器支持的全部指令（伪指令未列出）。指令描述摘抄至 [2]。

Mnemonic	Instruction
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
AND	And
ANDI	And Immediate
BEQ	Branch on 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
BNE	Branch on Not Equal
BREAK	Breakpoint
CACHE	Perform the cache operation
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word

DIVU	Divide Unsigned Word
ERET	Return from Exception
J	Jump
JAL	Jump and Link
JALR	Jump and Link Register
JR	Jump Register
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LUI	Load Upper Immediate
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
MADD	Multiply and Add Word
MADDU	Multiply and Add Word Unsigned
MFC0	Move From Coprocessor 0
MFHI	Move From HI
MFLO	Move From LO
MOVN	Move Conditional on Not Zero
MOVZ	Move Conditional on Zero
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MTC0	Move To Coprocessor 0
MTHI	Move To HI
MTLO	Move To LO
MUL	Multiply Word to Register
MULT	Multiply Word
MULTU	Multiply Unsigned Word
NOR	Nor
OR	Or
ORI	Or Immediate

SB	Store Byte
SH	Store Halfword
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable
SUB	Subtract Word
SUBU	Subtract Unsigned Word
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right
SYSCALL	System Call
TLBP	Probe TLB for Matching Entry
TLBWI	Write a TLB entry indexed by the Index register
WAIT	Enter Standby Mode
XOR	Exclusive Or
XORI	Exclusive Or Immediate

6.2 CP0

本节列出 NaiveMIPS 处理器支持的 CP0 寄存器字段，描述摘抄至 [3]。

Register 0 *Index* TLB 表入口索引

Fieleds	Bits	Description	R/W	Reset State
P	31	Probe Failure. Hardware writes this bit during execution of the TLBP instruction to indicate whether a TLB match occurred: 0 = A match occurred, and the Index field contains the index of the matching entry, 1 = No match occurred.	R	Undefined
Reserved	30..4			0
Index	3..0	TLB index. Software writes this field to provide the index to the TLB entry referenced by the TLBR and TLBWI instructions. Hardware writes this field with the index of the matching TLB entry during execution of the TLBP instruction. If the TLBP fails to find a match, the contents of this field are UNPREDICTABLE.	R/W	Undefined

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			0
PFN	25..6	Page Frame Number. Corresponds to bits[31..12] of the physical address.	R/W	Undefined
C	5..3	Coherency attribute of the page.	R/W	Undefined
D	2	“Dirty” bit, indicating that the page is writable. If this bit is a one, stores to the page are permitted. If this bit is a zero, stores to the page cause a TLB Modified exception.	R/W	Undefined
V	1	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
G	0	Global bit. On a TLB write, the logical AND of the G bits from both EntryLo0 and EntryLo1 becomes the G bit in the TLB entry. If the TLB entry G bit is a one, ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both EntryLo0 and EntryLo1 reflect the state of the TLB G bit.	R/W	Undefined

Register 4 *Context* 记录页表入口

Fieleds	Bits	Description	R/W	Reset State
PTEBase	31..23	This field is for use by the operating system and is normally written with a value that allows the operating system to use the Context Register as a pointer into the current PTE array in memory.	R/W	Undefined
BadVPN2	22..4	This field is written by hardware on a TLB exception. It contains bits $VA_{31..13}$ of the virtual address that caused the exception.	R	Undefined
0	3..0	Must be written as zero; returns zero on read.	R	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..8			0
ASID	7..0	Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry’ s TLB ASID field.	R/W	Undefined

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
CU	31..28	Controls access to coprocessors 3, 2, 1, and 0, respectively.	R	1
Reserved	31..23			0
BEV	22	Controls the location of exception vectors: 0=Normal, 1=Bootstrap.	R/W	1
Reserved	21..16			0
IM[7:0]	15..8	Interrupt Mask: Controls the enabling of each of the interrupts.	R/W	Undefined
Reserved	9..5			0
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	This bit must be ignored on write and read as zero.	R	0
ERL	2	Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken. 0 = Normal level, 1 = Error level.	R/W	1
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
BD	31	Indicates whether the last exception taken occurred in a branch delay slot.	R	Undefined
Reserved	30..24			0
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector.	R/W	Undefined
Reserved	22..16			0
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			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 Select 0 *PRId* 处理器型号

Fieleds	Bits	Description	R/W	Reset State
Company Options	31..24	Available to the designer or manufacturer of the processor for company-dependent options. The value in this field is not specified by the architecture. If this field is not implemented, it must read as zero.	R	0
Company ID	23..16	Identifies the company that designed or manufactured the processor.	R	1
Processor ID	15..8	Identifies the type of processor.	R	0x80
Revision	7..0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type.	R	0

Register 15 Select 1 *EBase* 异常处理基址

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
Reserved	11..0			0

Register 16 Select 0 *Config* 处理器配置信息 0

Fieleds	Bits	Description	R/W	Reset State
M	31	Denotes that the Config1 register is implemented at a select field value of 1.	R	1
Reserved	30..16		R	0
BE	15	Indicates the endian mode in which the processor is running: 0=Little endian, 1=Big endian.	R	0
Reserved	14..3		R	0
K0	2..0	Kseg0 coherency algorithm.	R/W	Undefined

Register 16 Select 1 *Config1* 处理器配置信息 1

Fieleds	Bits	Description	R/W	Reset State
M	31	This bit is reserved to indicate that a Config2 register is present.	R	0
MMU Size - 1	30..25	Number of entries in the TLB minus one.	R	15
IS	24.22	Icache sets per way.	R	Preset
IL	21.19	Icache line size.	R	Preset
IA	18..16	Icache associativity.	R	Preset
DS	15..13	Dcache sets per way.	R	Preset
DL	12..10	Dcache line size.	R	Preset
DA	9..7	Dcache associativity.	R	Preset
Reserved	6..0			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] https://www.xilinx.com/support/documentation/ip_documentation/axi_uart16550/v2_0/pg143-axi-uart16550.pdf

[5] https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernetlite/v3_0/pg135-axi-ethernetlite.pdf

[6] https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf

[7] https://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf

[8] https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf

[9] http://www.davicom.com.tw/pddocs/DM9161A-DS-F01_101609.pdf

[10] SYSTEM V APPLICATION BINARY INTERFACE MIPS RISC Processor Supplement 3rd Edition