

第四届全国大学生计算机系统能力培养大赛（龙芯杯）

---

# EasterMIPS

## 项目设计报告

---

易璐 谢云龙 徐逸辰

北京邮电大学

这道题我真的做不队

{yilu, makiras, linyxus}@bupt.edu.cn

2020 年 8 月

# 目录

<b>第一部分 概述</b>	<b>3</b>
1.1 项目概述 . . . . .	3
1.1.1 设计框架 . . . . .	3
1.1.2 系统概述 . . . . .	3
1.1.3 外设概述 . . . . .	3
<b>第二部分 CPU</b>	<b>4</b>
2.1 流水线结构 . . . . .	4
2.1.1 总体架构 . . . . .	4
2.1.2 取指级 . . . . .	4
2.1.3 译码/发射级 . . . . .	5
2.1.4 执行阶段 . . . . .	5
2.1.5 访存阶段 . . . . .	5
2.1.6 写回阶段 . . . . .	6
2.1.7 级间数据前递方案 . . . . .	6
2.1.8 级间控制信号 . . . . .	6
2.2 指令集 . . . . .	6
2.3 协处理器 0 . . . . .	7
2.4 内存管理 . . . . .	7
2.5 中断和异常 . . . . .	8
2.5.1 中断 . . . . .	8
2.5.2 异常 . . . . .	8
2.6 缓存设计 . . . . .	8
2.6.1 指令缓存 (I-Cache) . . . . .	8
2.6.2 数据缓存 (D-Cache) . . . . .	9
2.6.3 写回缓存 (Victim Cache) . . . . .	10
2.7 CPU 外部接口 . . . . .	10
<b>第三部分 系统与外设</b>	<b>12</b>
3.1 系统移植 . . . . .	12
3.1.1 Pmon . . . . .	12
3.1.2 Ucore . . . . .	13
3.1.3 Linux . . . . .	14
3.2 外设添加 . . . . .	15

目录	2
<b>第四部分 开发支持工具</b>	<b>16</b>
4.1 自动化构建及测试 . . . . .	16
4.2 疫情期间远程协作 . . . . .	17
<b>第五部分 思考与展望</b>	<b>18</b>
5.1 CPU 性能 . . . . .	18
5.1.1 IPC 分析 . . . . .	18
5.1.2 频率分析 . . . . .	19
5.2 Cache 的设计与开发 . . . . .	19
5.2.1 Cache 的不足与优化 . . . . .	19
5.2.2 Cache 的开发 . . . . .	20
5.3 系统调试及外设 . . . . .	20
<b>第六部分 附录</b>	<b>22</b>
6.1 参考文献 . . . . .	22
6.2 致谢 . . . . .	22

# 第一部分 概述

## 1.1 项目概述

本项目是在“龙芯杯”大赛方提供的 FPGA 实验平台 (FPGA 为 Artix-7 XC7A200T) 上实现一个片上系统 (SOC)。其中，CPU 基于 MIPS 32 Rev 1 指令集架构，包含指令缓存和数据缓存；外设包含 SPI Flash,GPIO,MAC,DDR3,NAND, 串口及 LCD 屏幕；能够启动并使用 PMON 操作系统、uCore 操作系统，能够启动 Linux 至用户态初始化。功能方面能够通过大赛方提供的功能测试、性能测试、系统测试；性能方面频率达 90MHz，每时钟周期指令数为龙芯 GS132 的 27.044 倍。

### 1.1.1 设计框架

EasterMIPS 中的 CPU 采用顺序双发射五级流水线架构，实现了 MIPS32 Rev 1 的 78 条指令，18 个 CP0 寄存器，8 个中断，9 种例外，8 项转换检测缓冲区，固定页大小为 4KB。CPU 对外的访存通信通过四个接口，分别是 Uncached 属性指令接口、Cached 属性指令接口、Uncached 属性数据接口、Cached 属性数据接口。四个接口通过 AXI3 协议，经过 AXI Crossbar 整合成为一个接口与外设交互。

EasterMIPS 实现了指令缓存 (I-Cache) 与数据缓存 (D-Cache)，响应 CPU 的取指与访存请求。I-Cache 与 D-Cache 大小均为 16K，在连续命中时，能够实现不间断地流水返回数据。I-Cache 与 D-Cache 分别引出一个 AXI 接口。D-Cache 能够缓冲 CPU 的写请求，并且实现了一个写回缓存 (Victim Cache)，兼具了缓存与写回队列的功能。

### 1.1.2 系统概述

EasterMips 完整支持 PMON 和 Ucore 启动运行所需要的指令集，并且支持这两个系统的全部异常处理。可以完全的运行这两个系统，并执行应用程序。

对于 Linux，通过添加 CFLAGS 构建选项限制分支及修改内核代码去除自陷指令，完成了内核部分的的指令支持。同时使用 Buildroot 工具链完成了 Ramdisk 的裁剪构建，去除了原 ramdisk 中不支持的指令。该 ramdisk 在官方固件上完成了测试，支持简单的性能监控及登录鉴权。但受限于时间，EasterMips 目前仅能完成内核初始化过程，无法进入用户态。

### 1.1.3 外设概述

EasterMips 支持 SPI Flash 板载引导芯片，GPIO，MAC 网络控制器，DDR3 内存控制器，NAND Flash 板载储存，Urat 串口通信 (RS232)，LCD 显示。其中 LCD 屏幕在硬件系统初始化时会显示 RGB565 格式的 16 位图像，在系统中经由 CONFREG 进行显示控制。

# 第二部分 CPU

## 2.1 流水线结构

### 2.1.1 总体架构

CPU 采用五级流水顺序双发射架构，分别为取指、译码/发射、执行、访存、写回五级，在较多情况下可以同时发射两条相邻的指令；创新性使用译码级判断数据前递方案，执行级得到前递数据的方法来解决数据冲突。

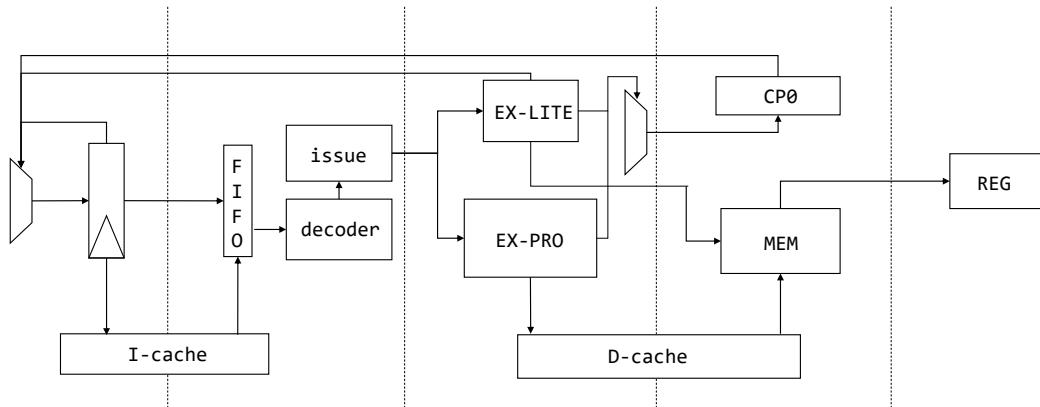


图 2.1: 流水线结构

### 2.1.2 取指级

取指阶段根据内存管理单元（MMU）给出的 cached 属性向指令缓存发起取指请求或直接向 RAM 发起取指请求，每次最多可返回两条指令，并被送入指令 FIFO 中等待译码阶段取出并发射。

指令 FIFO 长度为 16，采用双指针环形 FIFO 结构，双指针分别为读指针和写指针。当读指针和写指针指向同一个区域，则当前 FIFO 为空；当写指针加一为读指针，则当前 FIFO 为满。当清空流水线、发生分支，将读指针指向写指针。

### 2.1.3 译码/发射级

该阶段首先从指令 FIFO 中取出两条指令进行译码，同时会将这两条指令需要读取的寄存器编号送入寄存器文件，寄存器文件将返回数据前递方案（详细见 2.2.1 数据前递方案）和寄存器文件中对应的寄存器值。此处定义非乘除法的算术运算、逻辑运算指令为简单运算指令；定义数据相关为相邻两条指令的写后读数据相关。

两条流水线分别定义为 PRO 流水线和 LITE 流水线，PRO 流水线实现了除了分支指令功能外的所有指令共功能；LITE 流水线实现了分支指令功能和简单运算指令功能。

发射规则如下：

1. 分支和延迟槽始终一起发射，发射方法为将分支发射到 LITE 流水线，将延迟槽发射到 PRO 流水线；
2. 若两条指令中任一条是简单运算指令，且无数据相关，所需寄存器值可在执行阶段获得，则可同时发射两条，发射方法为将其中的简单运算指令发射到 LITE 流水线，将另一条发射到 PRO 流水线；
3. 若只能从 FIFO 读出一条指令，或两条指令中均非简单运算指令，或两条指令存在数据相关，或第二条指令所需寄存器值不能在执行阶段获得，仅发射第一条，发射方法为将第一条指令发射到 PRO 流水线；
4. 若只能从 FIFO 读出零条指令，或第一条指令所需寄存器值不能在执行阶段获得，发射零条。

为了方便发射逻辑判断，译码得到的指令操作码采用“三位指令类型 + 五位类型内编号”格式，发射时判断前三位即可得知对应的指令类型。

发射时将有一位标志位 reverse 表示发射顺序，当 reverse 为 1，代表 LITE 流水线中的指令是 PRO 流水线中的指令的前一条；反之，代表 PRO 流水线中的指令是 LITE 流水线中的指令的前一条。该标志位将随流水线流水至最后一级。

### 2.1.4 执行阶段

执行阶段对所有非访存指令进行计算并得到结果，其中乘法指令和除法执行使用 Xilinx IP 核实现，分别在 4 个周期和 34 个周期后返回，执行乘除法指令时会暂停流水线直至计算结果完成。

执行阶段对访存指令进行数据准备，包括访存地址和写入数据的计算等，将虚地址发送给内存管理单元，得到访存的 cached 属性。

执行阶段对异常进行判断，并仲裁两条流水线的异常情况，最多向异常处理模块提交一个异常和一个 CP0 寄存器写请求。

执行阶段提交分支请求，由于分支仅在 LITE 流水线中处理，所以此时不需要仲裁。

### 2.1.5 访存阶段

访存阶段根据内存管理单元 (MMU) 给出的 cached 属性向数据缓存发起访存请求或者直接向 RAM 发起访存请求。若不能在下一拍完成访存请求，则需要暂停流水线，等待数据返回。

同时在本级处理中断和异常。

### 2.1.6 写回阶段

写回阶段从获得加载类指令的写回结果和其他所有指令的写回结果，发起寄存器写请求。

### 2.1.7 级间数据前递方案

本项目创新性地使用译码阶段、执行阶段相配合的方案进行数据前递。译码阶段获取前递方案，执行阶段根据前递方案获取寄存器的值。

译码阶段在从 FIFO 取出两条指令后向寄存器文件模块发送指令中的 rs, rt 寄存器编号，寄存器文件根据当前执行阶段和访存阶段的指令的写寄存器情况确认前递方案。每个寄存器的前递方案采用 4 位表示，分别代表使用执行阶段 PRO 流水线的结果、使用执行阶段 LITE 流水线的结果、使用访存阶段 PRO 流水线的结果、使用访存阶段 LITE 流水线的结果。同时对于每个寄存器使用一位有效位表示前递方案是否有效，如果存在访存相关的数据前递，则有效位置为无效。

发射方案和前递方案有关，当一指令某一寄存器的前递方案无效且需要读该寄存器时，需要等待直至该指令的前递方案有效才能发射该指令。

执行阶段根据前递方案，在访存阶段的结果、写回阶段的结果、寄存器文件的数据中做出选择。由于流水线前进一级，前递方案的 4 位此时分别对应使用访存阶段 PRO 流水线的结果、使用访存阶段 LITE 流水线的结果、使用写回阶段 PRO 流水线的结果、使用写回阶段 LITE 流水线的结果，如果四位均无效，则选择寄存器文件中的数据。

### 2.1.8 级间控制信号

取指、译码/发射、执行、访存四级之间均有清空 (flush)、暂停 (stall) 两个控制信号；访存和写回之间有清空 (flush) 信号。由统一的控制模块接受各级流水的暂停请求和例外请求，继而对各级发出控制信号。

## 2.2 指令集

CPU 在大赛要求的 57 条指令基础之上，增加了部分指令以启动操作系统。实现的所有指令如下：

- **算术运算指令** ADD, ADDU, SUB, SUBU, ADDI, ADDIU, SLT, SLTU, SLTI, SLTIU, MUL, MULT, MULTU, DIV, DIVU, MADD, MADDU, MSUB, MSUBU, CLO, CLZ
- **逻辑运算指令** AND, OR, XOR, ANDI, ORI, XORI, NOR, LUI
- **移位指令** SLL, SRL, SRA, SLLV, SRLV, SRAV
- **访存指令** SB, SH, SW, SWL, SWR, LB, LBU, LH, LHU, LW, LWL, LWR
- **分支跳转指令** BEQ, BNE, BGEZ, BGTZ, BLEZ, BLTZ, BGEZAL, BLTZAL, J, JAL, JR, JALR
- **数据移动指令** MFHI, MFLO, MTHI, MTLO, MOVZ, MOVN
- **特权指令** CACHE(实现为空), SYSCALL, BREAK, TLBR, TLBWR, TLBWI, TLBP, ERET, MTC0, MFC0, PREF (实现为空) , SYNC (实现为空) , WAIT (实现为空)

## 2.3 协处理器 0

CPU 实现了 MIPS 32 Rev 1 规范中协处理器 0 中的大部分寄存器，同时为了启动操作系统，实现了 MIPS 32 Rev 2 规范中的 EBase 寄存器。所有寄存器如下，名称摘录自参考资料 3：

- Index Register (CP0 Register 0, Select 0)
- Random Register (CP0 Register 1, Select 0)
- EntryLo0, EntryLo1 (CP0 Registers 2 and 3, Select 0)
- Context Register (CP0 Register 4, Select 0)
- PageMask Register (CP0 Register 5, Select 0)
- Wired Register (CP0 Register 6, Select 0)
- BadVAddr Register (CP0 Register 8, Select 0)
- Count Register (CP0 Register 9, Select 0)
- EntryHi Register (CP0 Register 10, Select 0)
- Compare Register (CP0 Register 11, Select 0)
- Status Register (CP Register 12, Select 0)
- Cause Register (CP0 Register 13, Select 0)
- Exception Program Counter (CP0 Register 14, Select 0)
- Processor Identification (CP0 Register 15, Select 0)
- EBase Register (CP0 Register 15, Select 1)
- Configuration Register (CP0 Register 16, Select 0)
- Configuration Register 1 (CP0 Register 16, Select 1)

## 2.4 内存管理

CPU 使用内存管理单元 (MMU) 来进行虚地址到实地址的转换。对于 kseg0 和 kseg1 两段进行直接映射，kuseg、kseg2、kesg3 通过转换检测缓冲区 (TLB) 转换，当转换检测缓冲区 (TLB) 发生缺失时，触发例外，由软件调入新的 TLB 项消除例外。CPU 共有 8 项 TLB，为了保证良好时序，设计 ibuffer 和 dbuffer，分别对应指令和数据访存最近使用到的 TLB 项。当虚地址在 kseg0、kseg1 或在 buffer 中，可以在执行级当拍返回实地址；否则，将当拍返回流水线暂停信号，下一拍查找 8 项 TLB，返回实地址或者 TLB 例外。

## 2.5 中断和异常

### 2.5.1 中断

CPU 中支持 2 个软件中断 (SW0 SW1)、6 个硬件中断 (HW0 HW5) 和 1 个计时器中断。其中计时器中断复用 HW5 硬件中断。

### 2.5.2 异常

CPU 实现了 MIPS 规范的精确异常，在访存级统一提交异常。CPU 中实现的异常如下，名称摘录自参考资料 3：

- TLB Refill Exception
- TLB Invalid Exception
- TLB Modified Exception
- Integer Overflow Exception
- System Call Exception
- Breakpoint Exception
- Reserved Instruction Exception
- Coprocessor Unusable Exception
- Address Error Exception

## 2.6 缓存设计

CPU 实现了指令缓存和数据缓存，分别响应 CPU 取指与访存模块的请求。并且在与 AXI 的通信中都采用了 Wrap 模式，能够做到优先返回请求的字。

### 2.6.1 指令缓存 (I-Cache)

指令缓存能够响应双发 CPU 的取指请求。CPU 在请求缓存时给出一个地址，I-Cache 会尽量尝试返回从这一地址开始的连续两条指令。在给出请求的当拍，I-Cache 将返回给 CPU 两条指令的 valid 信号，指示返回的指令条数，然后在一拍给出对应的指令数据。

I-Cache 的参数配置为 2 路、每路 256 行、每行 8 字。I-Cache 能够返回已写入缓存的指令以及与 AXI 通信过程中已经获取到的指令。当连续命中时，I-Cache 可以不间断地、流水地返回指令。在发生命中时，会选中需要换出的行，并发起 AXI 请求从内存中读取缺失的行。在读完这一行的全部数据之后，再用一拍写回。

在 I-Cache 的工作过程中，除了写回的那一拍之外，都能立即返回在 Cache 中的、或者已经从 AXI 接口读取到的指令。I-Cache 使用 AXI Block Memory Generator 定制的 Naive RAM 作为数据存储单元。

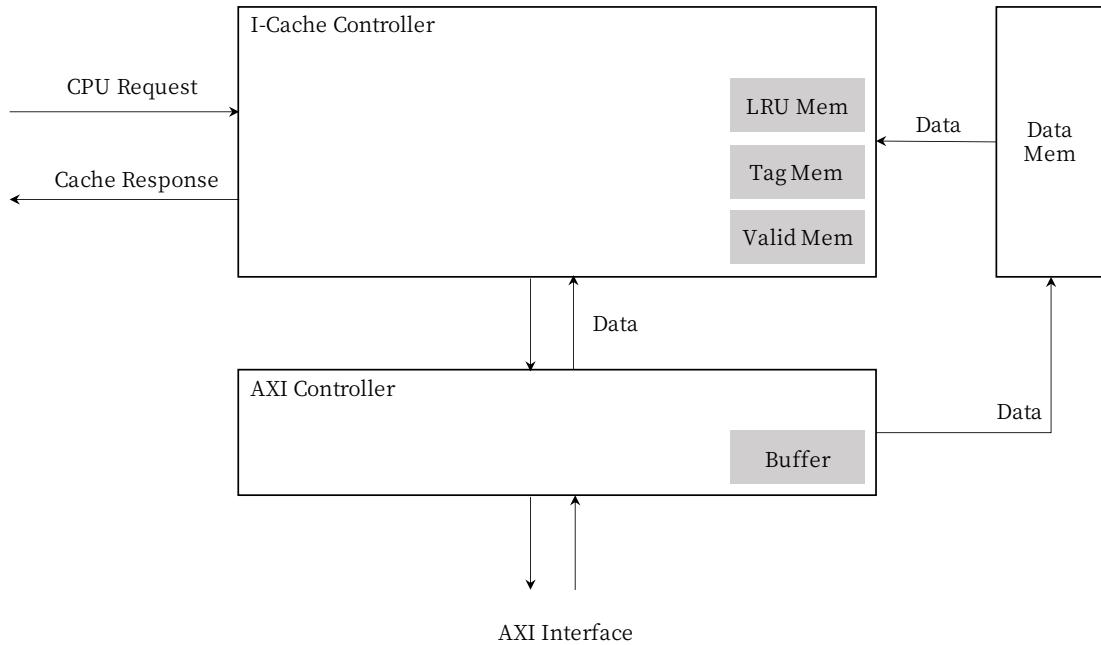


图 2.2: I-Cache 结构

### 2.6.2 数据缓存 (D-Cache)

数据缓存能够响应 CPU 的访存请求。对于 CPU 发来的请求，D-Cache 能够在当拍返回 wait 信号告知 CPU 是否能完成请求，若是读请求且无需 wait，则会在下一拍给出读取的数据。

D-Cache 能够命中缓存的数据以及在 AXI 读取中已经获得的数据。当连续命中时，D-Cache 能够不间断地、流水地完成访存请求。无论是读请求还是写请求，只要命中，D-Cache 都能够连续完成请求。

当发生 miss 时，D-Cache 将选中需要换出的行，若选中的行需要写回 (dirty 标志位有效)，D-Cache 会在下一拍从缓存数据中读出这一行，并在下一拍将这一行的数据与地址提交给 Victim Cache，若 Victim Cache 的 FIFO 没有被充满，则一拍内即可完成提交；同时，D-Cache 会向 Victim Cache 发起查询，检查缺失的行是否在其内，若有，则在一拍内读回，若没有，则发起 AXI 请求，从 AXI 接口读取缺失的行。当读取到这一行的所有数据后，在一拍内写回。

与 I-Cache 类似，除了写回的那一拍，只要发起的请求命中缓存，D-Cache 都能在一拍内完成请求，并且不间断地完成连续命中的请求。

除此以外，D-Cache 能够缓存一次发生 miss 的写请求。具体地，当 D-Cache 得到了 CPU 发来的一个写请求，发生 miss，且此时 AXI 控制逻辑空闲时，D-Cache 不会阻塞流水线，而是直接拉低 wait 信号，缓存这个请求，开始 AXI 读，在读到需要的字之后再完成请求。这样的设计能够在单独的写请求发生 miss 时，减少对流水线的阻塞，优化整体性能。

D-Cache 的参数配置为 2 路、每路 256 行、每行 8 字。此外，D-Cache 由 Chisel3 开发完成，支持相联度、行数、行大小可配置。

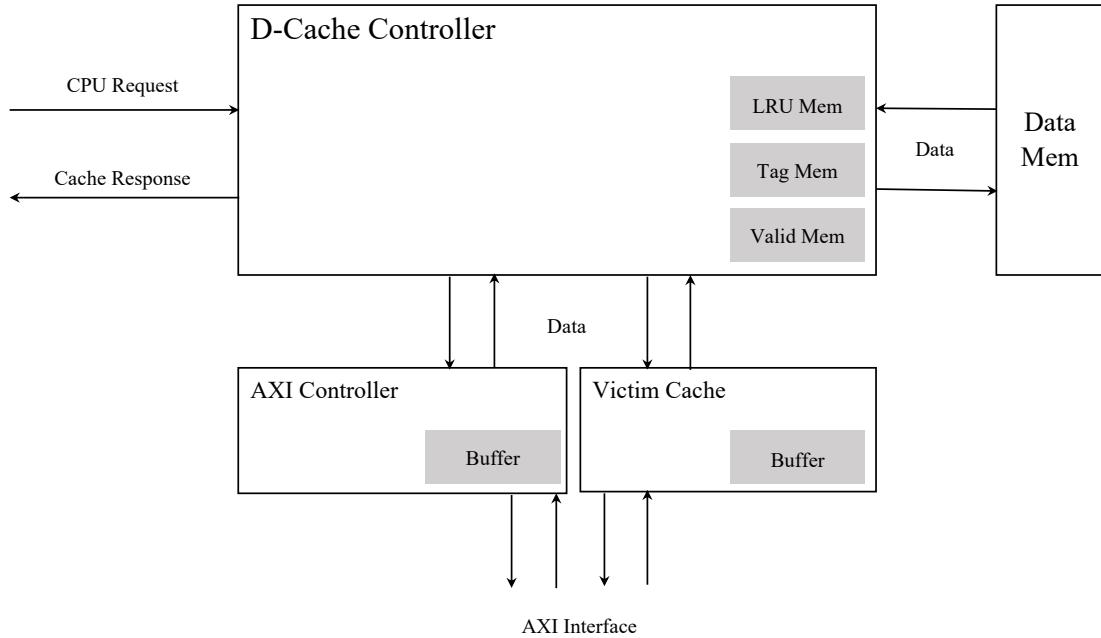


图 2.3: D-Cache 结构

### 2.6.3 写回缓存 (Victim Cache)

Victim Cache 能够处理 D-Cache 提交的写回请求。它的存在为 Cache 性能带来了两方面的提升，(1) D-Cache 无需自行处理某一行的写回，无需等待写回完成就能进行新的 AXI 读取，减少了 D-Cache 的返回时延；(2) 当 D-Cache 发生 miss 时，若缺失的行为刚刚换出写回的行，则可以在一拍内从 Victim Cache 中获取，大大降低了在这种情况下的延迟。

Victim Cache 本质是一个 FIFO。当收到 D-Cache 发起的访存请求时，将该请求写入 FIFO。AXI 控制逻辑会按序完成 FIFO 中的写请求。只要 FIFO 没有被充满，Victim Cache 都能立即响应 D-Cache 的请求。

值得注意的是，在 FIFO 中的每一行都有一个 valid 位，当写入请求时，valid 位被置为有效。当完成请求时，并不会清除 valid 位。而对于 D-Cache 的查询请求，所有 valid 位有效的行都能被命中。因而在 FIFO 中的数据，在完成写回之后，依然能够提供给 D-Cache。这意味着在访存范围较大且前后反复访存导致写回频繁发生时，只要写回的行在重新命中时仍然在 FIFO 存储空间中未被覆盖，就能在一拍内返回给 D-Cache，大大优化了性能。

然而，这也带来了一个问题：若同一行先后两次被提交进 FIFO，则 FIFO 的存储空间中可能包含同一行的新旧两个版本。判断版本的新旧会带来较为复杂的逻辑，因而在我们的实现中，转而保证 FIFO 存储空间中任意时刻只可能有同一行的一个副本有效，且为最新的副本（如果有）。实现方法是在处理 D-Cache 的写回请求时，判断写回地址是否在 FIFO 存储空间中且有效，若有效则将其无效再将新请求的数据写入 FIFO。

## 2.7 CPU 外部接口

CPU 的整体结构如图 2.4 中所示。CPU 共引出了 4 个独立的 AXI 接口，分别是不经过缓存的取指、访存 (Un-cached Inst and Un-cached Data) 与经过缓存的取指、访存 (I-Cache and D-Cache)。由于提供 SOC 要求 CPU 仅仅引出一个 AXI 接口，因而使用 Xilinx IP 定

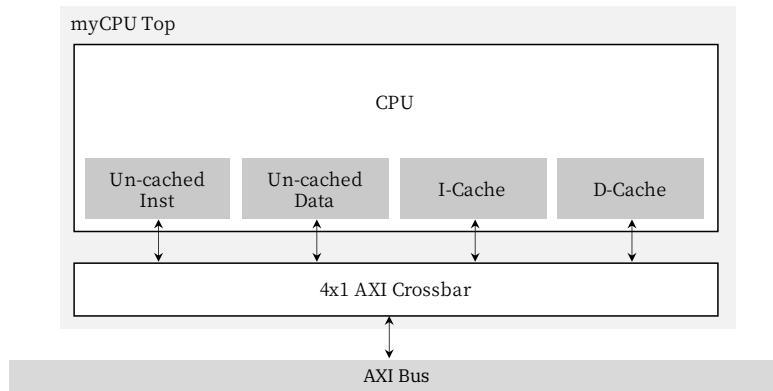


图 2.4: CPU 外部接口

制一个 AXI Crossbar，包含 4 个 Slave Interface，一个 Master Interface，将四个 AXI 接口整合为一个引出到 SOC。

而为了优化系统性能，采用了 D-Cache > I-Cache = Un-cached Data = Un-cached Inst 的优先级配置，也即 D-Cache 访存优先级最高，其余三个进行公平竞争 (round-robin)，来保证数据访存不会被其他请求阻塞。

## 第三部分 系统与外设

### 3.1 系统移植

受限于实力及时间，目前 EasterMips 仅成功进行了 PMON for LS1B，Ucore 的正常启动。对于 Linux，只完成了内核态的初始化流程，在 ramdisk\_execute\_command 执行 /sbin/init 进入用户态时，会因不明原因发生 Panic。外设方面，我们进行了 NT35510 LCD 屏幕的初始化加载及 linux 驱动移植，但受限于进度，我们暂时无法测试移植驱动的工作情况。以下固件的编译均于如下环境下进行：

1. 龙芯提供的 gcc-4.3-ls232(mipsel)
  2. Ubuntu 18.04 64bit

同时,由于 EasterMips 暂未支持 MIPSr1 中的部分指令,故编译时使用`-mno-branch-likely` 去除 `branch-likely` 指令,同时将 `cache` 等指令实现为空。

### 3.1.1 Pmon

在进行 Pmon 移植时, Github 的编译指导带来了关键帮助:帮助定位目标设备为 LS\_1B, 避免了因目标设备选择错误导致的诸多问题。其次, 在移植过程中存在 `gzrom` 反汇编内容主要为压缩数据, 进而无法查看指令的困难。经过查找编译过程的中间文件, 我们发现未压缩的 PMON 位于 Targets/LS1B/complie/ls1b 文件夹下, 通过对该文件进行反汇编, 调试得以正常进行。在调试中, EasterMips 处理了如下两个较为典型的问题:

**badblocks** 在初期移植过程中，启动扫描储存器时经常出现异常多的 badblocks。在当时因未影响进行后续调试，故并没有研究该问题。而在后续对 icache 的重构中，该问题时隐时现。经过推断，该问题与可能与取指过程有关，但由于未能抓取明确波形，我们仅将其视为 icache 出错的警告信号。并通过该现象多次将指出错的隐患迅速解决。

(a) 截图 1

(b) 截图 2

图 3.1: PMON 网络调试

**网络无连接** 在移植完成后的一次时序优化中，由于忽略了数据前递的部分条件导致 PMON 的网络无法正常加载。具体表现为 swl 与 swr 同时执行时，会丢失前者的数据。该问题的调试过程非常有趣：在调试中首先通过 Linux 的 tcpdump 应用监听局域网可以发现，配置网络时 PMON 会发出一个无法识别的数据包，这说明写入寄存器配置的过程应当是执行无误的。其次，观察发现该数据包每 32 位的后 16 位都为全 0。

考虑到其格式非常类似 ARP 数据包，我们将其与正常工作时进行了比对。如图 3.1 中所示。有理由假定  $[0x0, 0x10]$  是由 CPU 写入的，出现了异常。而  $0xc0a80168$  则是配置的 192.168.1.104 地址 16 进制数据，不经过该段程序。故我们针对性地对此类问题进行了查找，并解决了问题。

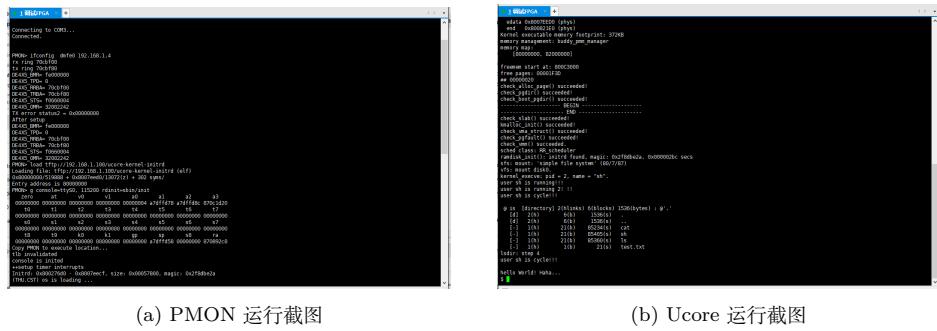


图 3.2: PMON 与 Ucore 的运行截图

### 3.1.2 Ucore

EasterMips 在进行 Ucore 移植的过程中，主要帮助 CPU 部分进行 CP0 相关寄存器的实现及修改，并标准化诸如初始值和默认赋值、操作等实现。在进行移植的过程中，主要出现过如下问题：

**启动引导异常** 在初次遇到问题时，我们直接使用了官方提供的 ucore 而非自己编译的内核进行引导。在执行时触发了 PMON 的异常，直接跳转到 .v200\_msg 段。经过反汇编发现官方给出的 ucore 存在一个直接跳转，与我们后续自行编译的跳转不符。我们猜测这是由于面向 Rom 与面向 Ram 构建的不同，重新构建后程序正常进入启动流程。同时在启动后续过程中，还存在系统会跳转到 PMON 的异常处理程序的异常行为。检查波形后我们发现这是我们没有正确实现 CP0-Ebase 寄存器导致的。

**内核错误中断** 在启动初期，`++interrupt timer` 之后系统会直接陷入 trap，我们在此抓取了波形，并定位到系统在此时不应受理中断请求，但是没有被屏蔽掉。该问题耗费我们了很多时间用来 debug 硬件错误，直到多次检查源码后注意到 `kern\trap\trap.c` 文件中的 `interrupt_handle()` 在处理中断时没有考虑到 CP0\_Status 寄存器，该问题导致了系统总是产生错误中断并打断启动流程。该问题提醒我们处理故障时不能仅考虑硬件问题，也应考虑系统驱动的问题。修改代码后系统正常运行到下一步。

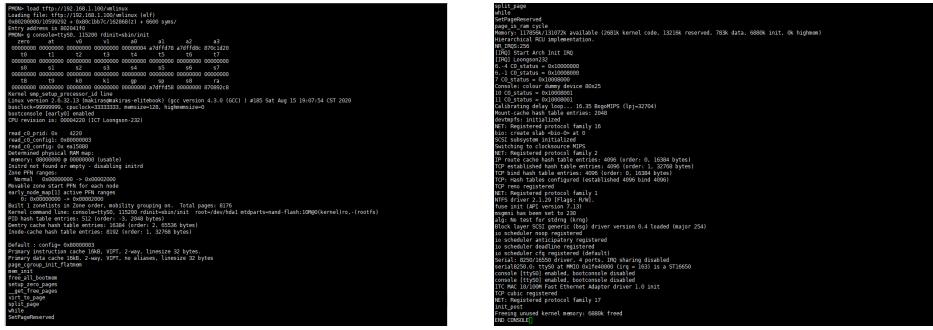


图 3.3: Linux 运行截图 (仅完成内核态初始化)

### 3.1.3 Linux

在进行 Linux 移植的过程中，我们遇到了诸多困难。查阅参考资料后我们的移植解决了下列问题。

**指令集裁剪** 初期调试 `ucore` 的过程中由于我们还未实现 `tlbwr` 指令，我们选择利用软件进行模拟。而在调试 Linux 时由于 `build_tlb_refill_handler` 函数的存在，软件模拟存在不可靠和难以调试的问题，故我们补充实现了该指令。通过在 `arch/mips/include/asm/longson-ls1x/cpu-feature-overrides.h` 中定义 `cpu_has_llsc/ejtag/fpu` 等选项，我们屏蔽了系统调用 `LL/SC` 指令，减少了一定的工作量。对于自陷指令 `TNE`, `BREAK` 等，我们选择在`.../asm/bug.h` 文件中删除相关的指令，进而绕过该问题。然而在实际操作中，该问题增加了手动 Debug 的难度，有许多信息需要手动设置 `printk`。每次调试都需要重新加载，而 PMON 仅能以 260K 左右的速度加载 10M 左右的内核，这十分消耗时间。如果实现该指令，极有可能减少大量调试时间。

**BuildRoot 制作 Ramdisk 文件** 由于系统自带的 `ramdisk.cpio` 文件未提供源代码，其中含有许多我们没有实现的指令，我们需要修改 `ramdisk` 文件。考虑到通过反汇编修改难度过高，故最终决定利用 `buildroot` 工具制作我们自己的 `ramdisk` 文件。虽然由于时间问题在 EasterCPU 上没有成功进行初始化，但至少已在 `SOC_UP_33M` 上确认实现了正常运行，提供了简单的登录认证和性能监控。在制作过程中，我们在最顶层 `makefile` 文件中始终找不到 `CFLAGS` 等编译参数的配置，通过全局搜索，我们才发现该参数实际处在 `packages` 目录下。添加编译选项后，我们确认该方法成功解决了 `ramdisk` 中含有不受支持指令的问题。

**外部配置问题** 在进行调试时，`EasterMips` 启动过程在 `Calibrating Delay Loop` 受阻，查阅资料我们得知这是利用定时器中断估计 CPU 性能的函数，进而定位问题是由于时钟中断信号所致。但由于前期在调试 `soc_up` 时误将 `IP7` 中断有效信号误解，将除其之外的整体中断信号取反。最终导致在 `ucore` 在启动过程的最后无法正常输入指令，而 Linux 会卡在串口初始化。该错误操作导致的后续问题花费我们大量的时间去检查。但也因此发现了开启了内核的 `no reconfig serial` 默认选项后，串口波特率并不会改变，即官方提供的内核启动参数中，`115200` 并不是必选项。在调试末期，`EasterMips` 还存在 Linux 初始化时的 TLB 异常循环问题。经过引出 `k0,k1` 寄存器的信号读取，我们发现在读取页表的偶数页时，访存并载入 `CP0_EntryLo0` 的数据始终为 0。该问题检查的检查耗时两天，直到检

查.config 文件时，我们发现系统设置的页大小为 16KB，这可能是中途编译时重新读取了 ls232\_defconfig，而未 make clean 所致的问题。更改为 4KB 后 EasterMips 正常进入下一步 ramdisk 初始化读取。此类问题都是容易出问题的外部配置问题，在调试内核及 CPU 代码时很容易导致定位问题错误，浪费大量时间导致进度滞后。这也从侧面说明了代码习惯及行为记录的重要性。

### 3.2 外设添加

**LCD 屏幕** 我们仅在大赛官方提供的 soc\_up\_33M 上添加了 NT35510 的显示支持，通过将图片载入为.coe 文件，我们令屏幕成功初始化。其控制接口实例位于控制寄存器模块内。后续的 Linux 内核驱动移植因无法确认有效性，故不再进行更多描述。

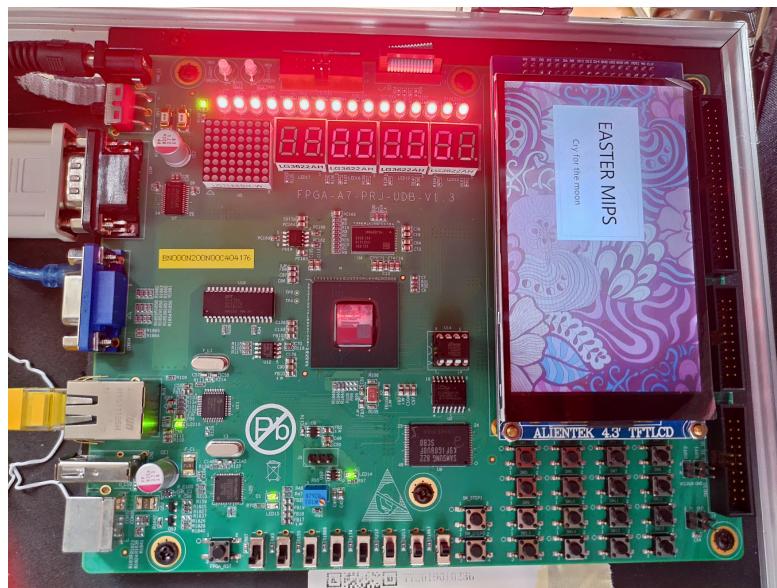


图 3.4: LCD 屏幕显示

**中断号** EasterMips 支持 6 个外部中断信号输入，其中最高位 IP7 外接中断为空，内部由时钟中断复用。

硬件中断名称	来源
Cause <sub>IP7</sub>	GND: 时钟中断复用，无设备接入
Cause <sub>IP6</sub>	dma_int: NAND DMA 的中断请求
Cause <sub>IP5</sub>	nand_int: NAND FLASH 的中断请求
Cause <sub>IP4</sub>	spi: SPI FLASH 的中断请求
Cause <sub>IP3</sub>	uart0_int: 串口的中断请求
Cause <sub>IP2</sub>	mac_int: 网口的中断请求

表 3.1: 中断

# 第四部分 开发支持工具

在疫情期间，由于团队成员异地协作难度较高，经由指导老师的帮助，我们完成了如下系统的构建。提升了协作效率。

## 4.1 自动化构建及测试

得益于周老师提供的一台 10 核 32G 内存 150G 硬盘的服务器，及北邮人团队的校内 Git 服务器，参考并重写清华 Nontrivial-mips 提供的自动构建脚本，通过在服务器上对 vivado 进行 docker 封装，我们成功建立校园环境内的自动构建服务器。该服务器可以完成如下功能：

1. 自动完成功能、性能、TLB 测试的 Simulation 阶段
2. 自动完成 func/perf bitstream 的构建，并上传到平台以供下载。
3. 提供时序检查报告的下载。

通过该自动构建服务器，帮助我们解决了远程协作时代码的有效性问题，减少了 merge 分支时带来的出错可能。

All Pipelines				
状态	流水线	触发者	提交	阶段
已通过	#1231 最新	gitlab CI	timing_final ~ 905e854f [func test][perf test] a little bit ch...	✓
已通过	#1230 最新	gitlab CI	linux_debug ~ 69154a2c [perf test][func test][perf bitstre...	✓ ✓
已通过	#1229	gitlab CI	linux_debug ~ 5f6de48b [func test][perf test] fix ucore boo...	✓
已通过	#1228	gitlab CI	timing_final ~ 26b4ee13 [perf bitstream] Generate bit	✓
已通过	#1227	gitlab CI	timing_final ~ 232bf67f merge change in cache[func test]...	✓ ✓

图 4.1: CI 使用情况

## 4.2 疫情期间远程协作

在疫情期间，团队协作问题给我们的进展带来了很大的困扰。同时团队成员的电脑性能不足也令开发进度受到限制，运行测试时时间长和复现问题困难；还因为系统测试板仅有的一块，在执行后续调试时问题较为繁多。为此，我们通过如下两种方法解决：

1. 通过在校内服务器上进行内网穿透并配置 VNC 远程桌面，进行远程编写。解决本地计算机性能不足和环境不同 (macOS) 的问题，节省调试时间并给本地留出更多的性能。
2. 通过 Teamviewer 进行远程桌面，在调试系统测试的波形问题时，不需要频繁上传下载波形即完成。配合语音通话并进行同屏协作，调试效率大大提升，比单纯靠语音描述更加有效。

整个开发过程中，我们一共进行了 270 余次提交，完成了 194 次有效的自动化构建，节省的总时长大约为 230 小时。

在穿透时，我们选择仅支持 `ssh_public_key`，设置不常用用户名来避免暴力破解，按需开启 VNC 功能，在方便了自身的同时确保校内网络安全。

# 第五部分 思考与展望

## 5.1 CPU 性能

目前，从 IPC 和频率两个维度考察我们的 CPU 性能，IPC 和 GS132 处理器的 IPC 比值为 27.044，频率为 90M。我们深入探究了限制我们的 IPC 和频率的原因。

### 5.1.1 IPC 分析

我们 CPU 设计为顺序双发射，在较多情况下发射两条指令使得我们的 IPC 有较大提升。限制 IPC 的主要是流水线中的暂停。在我们的 CPU 中共有以下几种可能导致流水线的暂停：

1. 等待指令和数据访存完成；
2. 产生分支，流水线需要更新；
3. 数据相关，等待 LOAD 指令的数据结果返回才能发射；
4. 等待乘除法指令执行完成。

通过在 CPU 中增加各级流水线的暂停拍数计数器和暂停原因计数器、查看仿真波形，我们分析以上因素的主次关系和解决方案。

首先，乘除法指令有提升的上限，且乘除法指令无论是在我们的测试集中还是在现实生活生产中，使用较少。可以通过将乘除法指令实现为不暂停流水线直至发起 HILO 访问得到较小的提升。

对于数据相关，为了保证 CPU 的频率，只能在写回寄存器之后再进行前递，如此将导致相邻的访存数据相关指令之间需要暂停两拍。可以通过将部分不会在执行级发生例外的指令延迟至访存级执行。

对于分支，流水线的代价较大，共需要暂停三拍。这是我们的 CPU 设计决定的，在执行阶段产生分支信号后，需要经过取指、指令放入 FIFO、译码并发射三拍，执行级才能恢复非空泡状态。这使得在分支较多且连续执行的指令段较短的程序当中，我们的双发射 CPU 的性能较差。可以通过增加分支预测器，较为明显得改善这一现状。

访存延迟是影响整个处理器性能的关键，这方面主要与 cache 的设计相关。我们在 5.2 节详细展开说明其中的原因。

分支和访存的两个原因，极大程度上造成了我们的 IPC 在十个测试样例中差距较大，IPC 比值最高可达 38.6，但最低仅为 20.8。

### 5.1.2 频率分析

我们的 CPU 频率限制来自于以下几个方面：

1. 双发射由于实现为两条流水线的顺序不定，在写回寄存器和数据前递时判断逻辑较为复杂；
2. D-cache 实现为实 Index 实 Tag，CPU 需要在执行阶段计算虚地址后查找 TLB。

双发射的两条流水线顺序不定起于设计初期考虑的在不占用更多资源的情况下尽可能多发射，对于频率的影响大于设计时的预期。可以折衷为两条流水线顺序固定，两条流水线功能大致相同，对于复杂模块（访存和乘除法）仅实现一个，两条流水线仲裁后使用。

## 5.2 Cache 的设计与开发

### 5.2.1 Cache 的不足与优化

#### Cache 性能优化

我们的 I-Cache 与 D-Cache 都实现为 2 路、每路 256 行、每行 8 字。在连续命中时，可以不间断地流水返回数据。I-Cache 只有在 CPU 的请求既不在 Cache 存储中，也不在 AXI 读缓冲中的时候，或是命中了 Cache 中的数据，但该数据所在行正在进行 AXI 读的时候，又或是这一拍将会把 AXI 读到的数据写近 Cache 存储中的时候会停止流水线。在其他情况下，都能够直接返回数据。D-Cache 只有在 CPU 的请求既没有命中 Cache 存储也没有命中 AXI 读缓冲的时候，或是这一拍正在将 AXI 读到的数据填入某一行中的时候停止流水线。

Cache 的性能有如下优化方式：

- I-Cache 和 D-Cache 在进行 refill 的那一拍无法响应命中了 Cache 行的请求，原因是我们在采用的是单个地址端口的存储器，在 refill 时必须占用地址端口，因而这一拍无法从存储器中读取数据。解决这个问题的方法是采用双地址端口的双端口存储器。
- 无论是 I-Cache 还是 D-Cache，当前都不支持并行的 AXI 读写，也即只能进行一个 AXI 读写事务。这导致在代码发生跳转，且跳转前后位于不同行，且都发生 miss 的情况下，跳转后的指令需要在前一行读完之后，再进行握手，再进行读取，非常影响性能。应当利用好 AXI 总线多个读写事务的功能，同时进行多个读写请求，加快响应时间。这是下一步的改进方向。

#### 实现可配置的 Cache

现在版本的 Cache 中，只有 Chisel3 实现的 D-Cache 能够做到可配置（路数、行数、行大小），而用 SystemVerilog 实现的 I-Cache 与 Cache 顶层模块并不直接支持可配置。不仅如此，由于需要将 Chisel3 生成的 D-Cache 整合到 Cache 顶层模块中，还是需要手动改动许多接线，非常不便。如果能实现所有参数可配置的 Cache，将对性能调优以及在不同情况下的适用性有很大帮助。

事实上，用 Chisel3 完全重写的 Cache 能够实现路数、行数、行大小全部可配置，然而，由于在第 5.2.2 节中提到的原因，这一版本的 Cache 没能成功引导系统。因而在以后的开发中，我们希望能够完全实现一个可以引导系统的全面可配置 Cache。

### 5.2.2 Cache 的开发

**问题** 在本次系统的开发过程中，Cache 的开发处于滞后状态，落后于原定计划近一个多月时间，在七月中下旬才完成开发与调试。不仅如此，由于开发时许多因素考虑欠妥，从七月中下旬到八月上旬这段时间中，为了优化系统性能，保持系统的可维护、可调试性，对 Cache 进行了多次重构。其中包括用 Clash 重写 Cache，将单发 I-Cache 重写为双发，用 Chisel3 重写整个 Cache 模块。然而，由于 Cache 的开发进度原本就已经滞后，留给系统调试、引导系统的时间非常紧张，而对 Cache 频繁、多次、大规模的重构又导致系统、外设测试所用的代码版本与最新版本没有及时同步，再加上重写、改动 Cache 代码后，会给系统带来新的 Bug，需要更多的时间调试系统。最终导致了最终使用的 Cache 版本并非最终版本，I-Cache 采用了第一次重构的，用 SystemVerilog 实现的双发版本，D-Cache 采用了用 Chisel3 重写的较早版本。在后面的 Cache 开发中，为 Cache 实现的诸多优化与功能，都没有使用到最终的 CPU 中。

**反思** 反思这一系列问题与遗憾的原因。Cache 最为 CPU 中非常重要、不可或缺的一部分，理应在开始整体系统调试之前，就完全确定实现方式，敲定代码，完成测试与调试，保证其处于一个稳定、无 Bug 的状态。就算有性能优化与改动，也应当尽量在原有代码的基础上进行，而不是直接进行重构，甚至用另一种工具进行完全的重写。开发 Cache 的工具，Cache 的整体实现方式，应当在开发之前就敲定，而不是在比赛这么有限的时间内进行不断迭代。对 Cache 频繁的迭代与重写，加大了系统测试的调试难度，也最终让我们没有使用最新的，添加了更多优化的 Cache 版本。硬件系统的开发如此，其他领域的开发工作也是同样道理。若时间有限，在实现之前就应当深思熟虑，而不是一开始想着快速实现出一个原型 (prototyping)，然后之后在不断迭代优化——这样的策略只适用于有较充裕的时间，较多的迭代机会的项目，对于龙芯杯这样时间有限的比赛，这种思维并不适用。

**展望：Cache 调试框架** 在 Cache 的开发过程中，我们常常感受到一个 Cache 测试框架的必要性。在龙芯提供的功能测试、性能测试乃至系统测试中，对于 Cache 的测试都是间接的。这带来了两个问题，(1) 对 Cache 测试得不够彻底，不够全面；(2) 在以上测试中若出现了 Cache 的问题，定位起来较为麻烦，性能测试与系统测试尤为如此。因而在开发过程中，我们一直希望能有一个，或是自己开发一个 Cache 测试框架。但由于时间限制，这一想法一直搁置。Cache 作为整个系统中较为重要，也较容易带来问题的一个部分，为它专门开发一个详尽的，全面的测试框架是有必要的。理想中的测试框架应当着重测试 Cache 的正确性与一致性。应当验证其读取的数据是否正确，对于 dirty 的处理是否得当，若实现了写回缓冲，那么能否正确返回写回缓冲中最新的副本。除此以外还有 Cache 指令，在现有的环境中，对 Cache 指令的测试是困难的。除了自行编写测试程序之外，几乎只能在操作系统引导时测试。而 Cache 指令的测试，就算自行编写测试程序，用现有的测试框架也不那么直接，因为 Cache 指令并不直接涉及寄存器的写回。

## 5.3 系统调试及外设

**问题** 在本次系统移植与调试中，尽管有 Git 的帮助，仍然花费了大量的时间在版本同步上。同时许多例如中断信号高低位有效、内核编译的 kernel 配置文件、没有 make clean 导致污染发生等外部的简单配置经常出错，进而让为原本就不充裕的时间就更紧张。同时，软

件与硬件的移植过程中给，并没有提前按设计验证软件是否正常运行，导致问题定位的方向错误，白白浪费大量的时间去 debug 本不存在的硬件问题。在硬件稳定的前提下，系统软件移植过程可以采用快速的迭代的思想。但是在硬件仍未稳定的情况下，快速迭代造成了大量的可运行孤本存在，让后续开发陷入僵局。在前期进行开发时由于时间不紧张，我们还能正常执行稳定的分支维护，到后期 commit 记录过于混乱和跨长时间 merge 的问题给我们很大影响。

**反思** 在移植过程中，尽管我们已经阅读过关于硬件设计应当以稳妥和稳定为主，我们仍然错误的将互联网应用开发的快速迭代思想应用于硬件编程。快速迭代已经变成了我们的一种潜意识中的习惯，这是非常可怕的。在快速迭代的过程中，许多小错误莫名其妙的产生又消失，如果卡住时只能通过比对版本库进行检查。我们认为再次调试系统时，应当使用一个 base commit，在当天内仅在该 commit 上操作，修复，并使用 squash 进行提交合并，保证 commit 信息的有效性，使版本库具有可读性才是最重要的。

**展望** 在系统移植过程中，应当使用 qemu 配置一个系统工作的检测平台。但是受限于时间我们并没有做这个选项。然后后续在这类问题上多花费的时间远超配置所需的时间。所以构建系统的测试平台是有必要的，而通过交互监控可以检查系统的启动过程。同时，我们认为引入一个 EJTAG 来进行数据抓取也能提供很大帮助。可以节省 ILA 抓取的资源数量，查看内存的具体情况，便于在出现问题时进行 debug，节省时间。

# 第六部分 附录

## 6.1 参考文献

MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture.rev3.02

MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set

MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32 Privileged Resource Architecture

Xilinx IP:: AXI Crossbar (2.1) LogiCORE IP Product Guide

## 6.2 致谢

在 EasterMips 的开发过程中，我们获得了来自周峰老师的方向指导和服务器资源。他帮助我们指明了方向并帮助我们解决了疫情导致的合作问题。同时我们还获得了于海鑫学长的诸多指点，他为我们的实现过程提供了巨大的帮助。最后还要感谢北邮人团队于校内提供的 Git 服务器，这极大的方便了我们进行调试。