重庆大学课程设计报告

课程设计题目:	MIPS SOC 设计与性能优化
学　　院:	计算机学院
专 业 班 级:	计算机科学与技术 01 班
年　　级:	2019
学　　生:	袁福焱　江焰丰　李果　李雅雯
学　　号:	20174260　20174261　20174325　20174341
完 成 时 间:	2019 年　12 月　30 日
成　　绩:	90
指 导 教 师:	钟将

重庆大学教务处制

项目	分值	优秀 $100 > x \geq 90$	良好 $90 > x \geq 70$	中等 $80 > x \geq 70$	及格 $70 > x \geq 60$	不及格 $x < 60$	评分
		参考标准					
学习态度	15	学习态度认真，科学作风严谨，严格保证设计时间并按任务书中规定的进度开展各项工作	学习态度比较认真，科学作风良好，能按期圆满完成任务书规定的任务	学习态度尚好，遵守组织纪律，基本保证设计时间，按期完成各项工作	学习态度尚可，能遵守组织纪律，能按期完成任务	学习马虎，纪律涣散，工作作风不严谨，不能保证设计时间和进度	
技术水平与实际能力	25	设计合理、理论分析与计算正确，实验数据准确，有很强的实际动手能力、经济分析能力和计算机应用能力，文献查阅能力强、引用合理、调查调研非常合理、可信	设计合理、理论分析与计算正确，实验数据比较准确，有较强的实际动手能力、经济分析能力和计算机应用能力，文献引用、调查调研比较合理、可信	设计合理，理论分析与计算基本正确，实验数据比较准确，有一定的实际动手能力，主要文献引用、调查调研比较可信	设计基本合理，理论分析与计算无大错，实验数据无大错	设计不合理，理论分析与计算有原则错误，实验数据不可靠，实际动手能力差，文献引用、调查调研有较大的问题	
创新	10	有重大改进或独特见解，有一定实用价值	有较大改进或新颖的见解，实用性尚可	有一定改进或新的见解	有一定见解	观念陈旧	
论文(计算书、图纸)撰写质量	50	结构严谨，逻辑性强，层次清晰，语言准确，文字流畅，完全符合规范化要求，书写工整或用计算机打印成文；图纸非常工整、清晰	结构合理，符合逻辑，文章层次分明，语言准确，文字流畅，符合规范化要求，书写工整或用计算机打印成文；图纸工整、清晰	结构合理，层次较为分明，文理通顺，基本达到规范化要求，书写比较工整；图纸比较工整、清晰	结构基本合理，逻辑基本清楚，文字尚通顺，勉强达到规范化要求；图纸比较工整	内容空泛，结构混乱，文字表达不清，错别字较多，达不到规范化要求；图纸不工整或不清晰	

指导教师评定成绩：

指导教师签名：

MIPS SOC 设计报告

袁福焱、江焰丰、李果、李雅雯

1 设计简介

本次我们采用的设计是基于哈佛结构的 32 位五级流水线的处理器。该设计实现了大赛用 MIPS 基准指令集规范的所有内容，包含所有非浮点 MIPS I 指令和 MIPS32 中的 ERET 指令，CP0 协处理器，支持中断和系统调用。

2 设计详细思路

2.1 CPU

2.1.1 流水线 CPU 概述

基于性能和复杂度地综合考虑，我们采用五级流水线的形式来设计 CPU，即将整个流水线过程分为取指 (Fetch)，译码 (Decode)，执行 (Execute)，访存 (Memory)，回写 (Writeback) 五个阶段。在 Fetch 阶段，通过 PC 传出的地址从指令存储器中取出对应的指令，并将其送入 Decode 阶段，同时确定下一条指令的地址。在 Decode 阶段，将对指令进行译码，同时从通用寄存器中得到需要的寄存器的值，如果需要用到立即数，则对立即数进行符号扩展或无符号扩展，如果需要用到 HILO 寄存器，也在该阶段进行相应的处理。同时也在 Decode 阶段进行跳转指令的判断，如果需要跳转，则将跳转后的新指令送进 PC 中。译码后的指令进入 Excute 阶段，此阶段进行的是实际的算术逻辑运算。而运算的结果将被送至 Memory 阶段。在 Memory 阶段，除了对内存的访问，同时还负责进行对异常的判断和处理。紧接着运算结果或读取的内存的值将被送至最后一个阶段，即 Writeback 阶段，在此阶段它们将被写入寄存器中。CP0 中相关处理也在该阶段进行。

在 CPU 中，针对数据冲突，采取数据前推和暂停两种策略，即如果后级有数据尚未写入某寄存器中，而当前有需引入该寄存器的值，则提前采用需要的后级值，如果后级有数据需要从数据存储器中得到后再写入，而当前却又需要该存储器的值，则暂停流水线一个周期。针对分支跳转指令造成的控制冲突则采用延迟槽解决。

在 CPU 中，实现了精确异常处理，即发现异常时并不当时立刻处理，而是统一推至 Memory 阶段依据一定优先级进行依次处理。如果在 Memory 阶段发现先前的阶段或 Memory 阶段出现异常，则清空流水线，跳转至异常处理程序地址，同时将记录当前地址和写回数据，待异常处理结束再跳转回来。

2.1.2 CPU 核心接口说明

Name	Width	Direction	Discreption
clk	1	Input	时钟信号
rst	1	Input	复位信号,高有效
int	5...0	Input	中断信号
stall_by_sram	1	Input	缺失导致的暂停信号
inst_sram_en	1	Output	指令存储器使能信号
inst_sram_wen	3...0	Output	指令存储器写使能信号,
inst_sram_addr	31...0	Output	指令读取的地址
inst_sram_wdata	31...0	Output	写入指令存储器的数据,恒为 0
inst_sram_rdata	31...0	Input	从指令存储器读出的数据
data_sram_en	1	Output	数据存储器使能信号
data_sram_wen	3...0	Output	数据存储器写使能信号
data_sram_addr	31...0	Output	数据存储的地址
data_sram_wdata	31...0	Output	写入数据存储器的数据
data_sram_rdata	31...0	Input	从数据存储器中读出的地址

2.1.3 CPU 结构

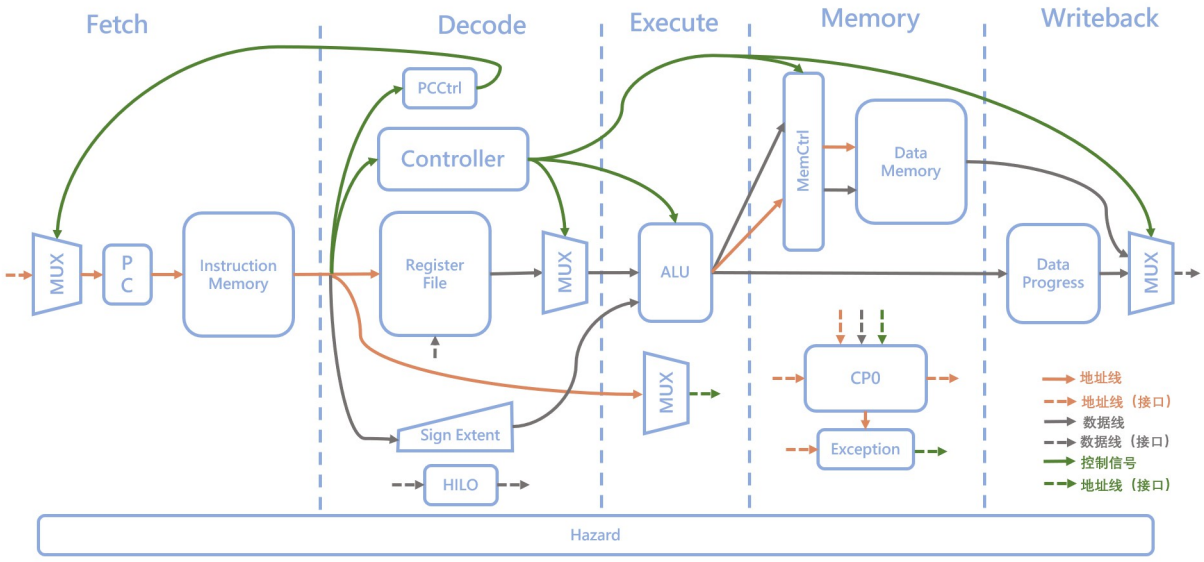

图 1: CPU 结构图

2.1.4 Fetch 阶段分析

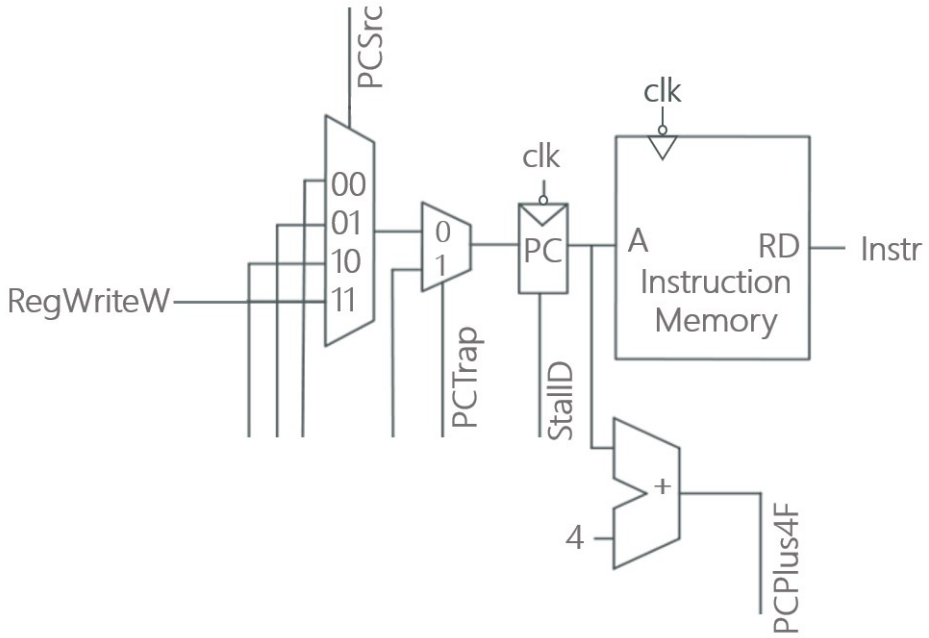

图 2: Fetch 阶段线路图

如上图 (2) 所示,Fetch 阶段由一个 PC 寄存器组成,该寄存器一般储存了读取的指令的地址,并每个周期都自增 4 后,一边将自增 4 后的地址传回,一边将此地址传向 Decode 阶段,当遇到跳转和异常处理指令时,则 PC 中存放跳转后指令地址或异常处理程序入口地址。

在 Fetch 阶段,同样存在着一个四选一多路选择器和一个二选一多路选择器,四选一多路选择器的控制信号 PCSrcD 决定下一条指令是顺序读取指令还是进行分支或者跳转指令。二选一多路选择器的控制信号 PCTrap 决定下一条指令是否跳转到异常处理程序的入口。

Fetch 阶段接口说明

Name	Width	Direction	Discreption
clk	1	Input	时钟信号
rst	1	Input	复位信号,高有效
pc_next	31...0	Input	存放着下一条要执行的指令地址
PC	31...0	Output	存放此时执行指令地址的寄存器
PCSrcD	1...0	Input	是否设定 pc_next 为分支或跳转指令地址
PCTrap	1	Input	是否设定 pc_next 为异常处理地址

四选一多路选择器接口说明

Name	PCSrcD	Width	Direction	Discreption
pc_plus4F	00	31...0	Input	顺序读取指令的地址
pc_branchD	01			分支指令的目的地址
pc_jumpD	10			立即数跳转的目的地址
pc_control_a	11			寄存器跳转的目的目的地址
p_next			Output	下一条执行的目的地址

2.1.5 Decode 阶段分析

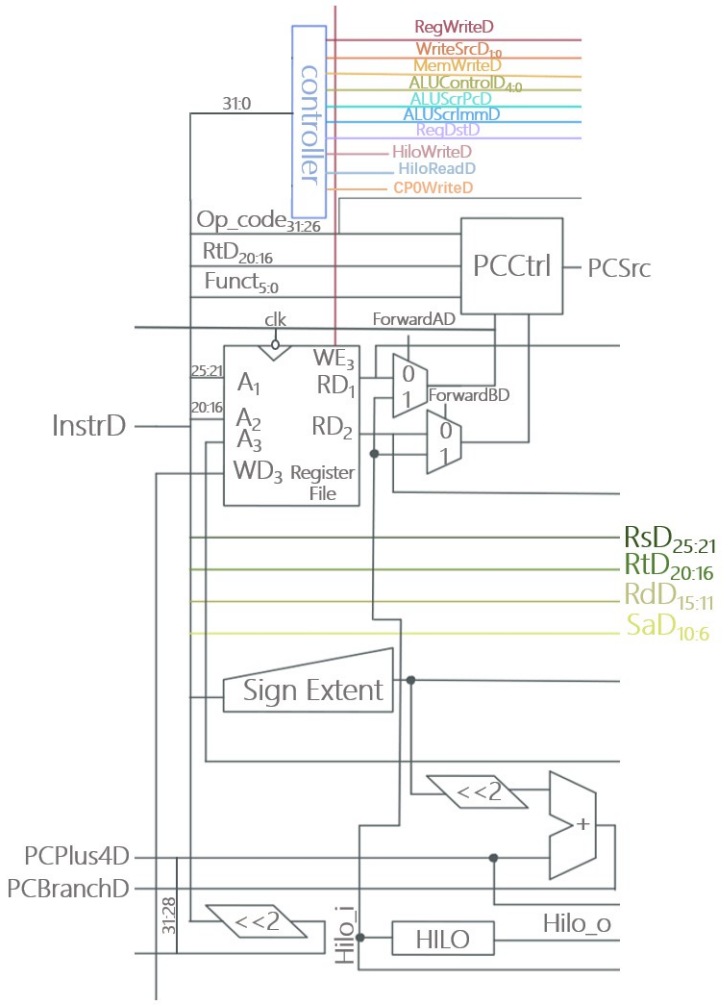

图 3: Decode 阶段线路图

如图 (3) 所示,Decode 阶段包括了通用寄存器访问、译码、分支判断以及 Hilo 寄存器相关部分的工作。

在 Decode 阶段中,译码的工作由 Controller 调用 main_decoder 和 alu_decoder 完成。main_decoder 负责翻译出指令类型及操作,alu_decoder 则负责译码得出需要的具体的算术逻辑运算。根据 main_decoder 译码后的地址送入通用寄存器 (Register File) 中取出相应寄存器的值,再将这些值送至 Excete 阶段。如果存在数据冲突,即当前需要用到后级还未来得及写回寄存器的值时,则直接采用该值而不是暂停流水线等待值的写回。此部

分均为组合逻辑,Controller 中的信号通过触发器一级一级依次传递,在线路图中同一个信号在不同级中末尾字母不同,为书写简略,在表中省略。

Decode 阶段接口说明

Name	Width	Direction	Discreption
InstrD	31...0	Input	指令输入
Op_code	5...0	Input	指令 InstrD 的 31...26 位
Funct	5...0	Input	指令 InstrD 的 5...0 位
PCPlus4D	31...0	Input	自增 4 后的地址
RsD	4...0	Output	指令中寄存器 s 的地址,没有则为 0
RtD	4...0	Output	指令中寄存器 t 的地址,没有则为 0
RdD	4...0	Output	指令中寄存器 d 的地址,没有则为 0
sign_immD	31...0	Output	符号或无符号扩展后的立即数,没有则为 0
PCBranchD	31...0	Output	分支指令地址

Controllor 接口说明

Name	Width	Direction	Discreption
InstrD	31...0	Input	指令输入
main_decoder	12...0	Output	解码后的指令,取值见宏定义 defines*
alu_decoder	4...0	Output	解码后的算术运算类型,见宏定义 aludefines*

Controllor 信号分解说明

Name	Offset	Width	Discreption
RegWrite	0	1	通用寄存器写信号
RegDst	2...1	1...0	Writeback 阶段写回的寄存器的控制信号
AluSrcPC	3	1	是否将 PCPlus4 作为第一操作数的控制信号
AluSrcImm	4	1	是否将立即数作为第二操作数的控制信号
WriteSrc	6...5	1...0	Writeback 阶段写回的数据的控制信号
HiloRead	7	1	Hilo 寄存器读信号
HiloWrite	8	1	Hilo 寄存器写信号
Branch	9	1	分支指令控制信号
UnsignExtend	10	1	是否进行无符号扩展控制信号
Jump	11	1	跳转指令控制信号
CP0Write	12	1	CP0 写信号
AluControl		4...0	决定 Alu 具体采用的算术逻辑运算的控制信号

分支跳转判断在 PCCtrl 中进行,如果发现是分支指令,则通过 PCCtrl 判断分支条件并输出对应的分支指令目的地址和 PCSrc 控制信号;如果发现是跳转指令,同理在此判断跳

转指令类型,并输出对应的跳转指令目的地址和 PCSrc 控制信号。

Hilo 寄存器相关操作也在 Decode 阶段进行,即需要进行乘除法时。

Hilo 接口说明

Name	Width	Direction	Discreption
clk	1	Input	时钟信号
rst	1	Input	复位信号,高有效
we	1	Input	写使能,高有效
hilo_i	63...0	Input	Hilo 寄存器输入
hilo_o	63...0	Output	Hilo 寄存器输出

2.1.6 Excute 阶段分析

在 Excute 阶段,由 alu 负责实际的算术逻辑运算。alu 接收从 Decode 阶段传来的 alu_control 确定指令需要的算术逻辑运算,将结果推至 Memory 阶段。同时,如果前级需要运算结果,则立即前推回去以供需要。

传入 alu 的有第一操作数和第二操作数,在 Excute 阶段,还有两个三选一多路选择器和两个二选一多路选择器来决定传入的是来自通用寄存器的数据还是后级前推回的数据,二选一多路选择器决定传入的是三选一多路选择器传进的数据还是将 PCPlus4 中的地址作为操作数传入,来解决部分指令需要采用 PC 中地址指计算的问题。同时传入 alu 中的还有 hilo 寄存器中的数据,来应对需要多周期运算的除法相关操作,以及 sa,即指令 InstrD 的 10...6 位,在部分移位指令中使用。

alu 中进行的绝大多数运算,都可以在一个周期内完成,但也难免会存在需要多周期进行的运算,如除法。本 CPU 中采用的除法器是经过我们改进的版本。

Execute 阶段接口说明

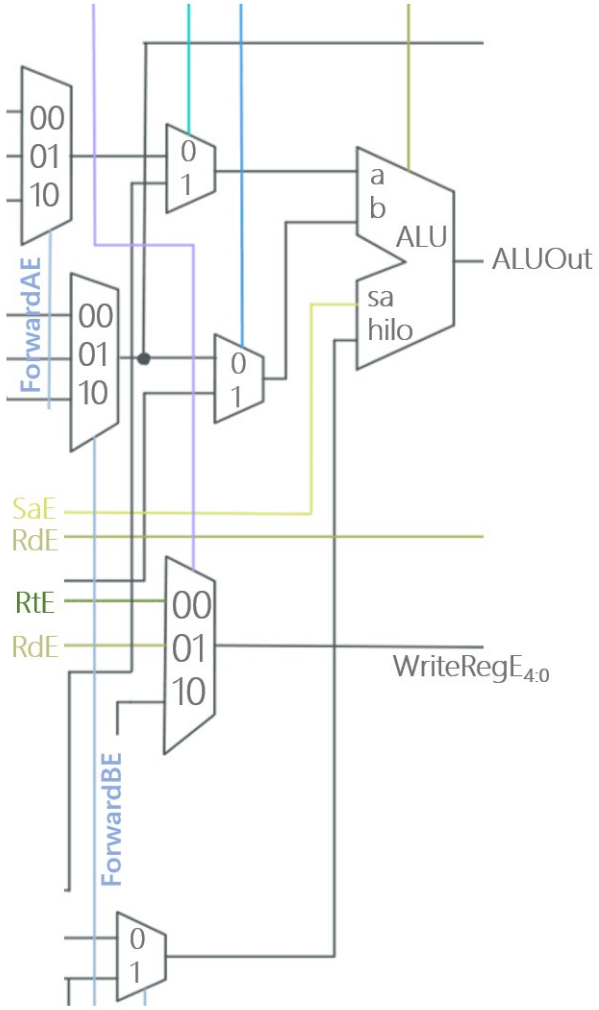

图 4: Execute 阶段线路图

Name	Width	Direction	Discreption
clk	1	Input	时钟信号
rst	1	Input	复位信号,高有效
a	31...0	Input	算术逻辑运算第一操作数
b	31...0	Input	算术逻辑运算第二操作数
hilo	63...0	Input	hilo 寄存器数据
sa	4...0	Input	某些移位指令中指定的立即数位移量
alu_control	4...0	Input	决定进行的算术逻辑运算方式的控制信号
y	63...0	Output	算术逻辑运算的结果
overflow	1	Output	整形溢出例外

2.1.7 Memory 阶段分析

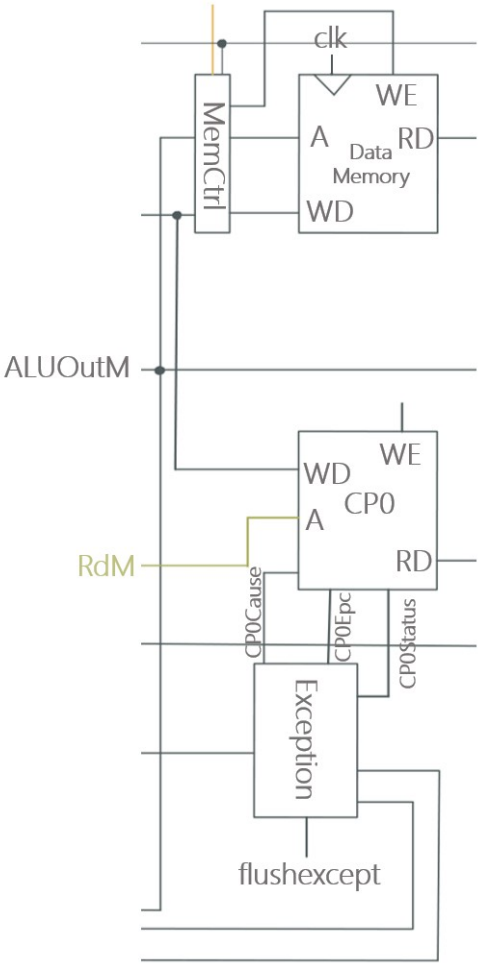

图 5: Memory 阶段线路图

在 Memory 阶段，负责存储，加载指令读写内存，异常检验，即如果存在异常，则刷新流水线并跳至异常处理程序入口，以及 CP0 中的相关处理。

Memory 阶段中，专门设立了 MemCtrl 模块来判断执行的指令是否需要对内存进行读写操作，并在判断后输出数据，地址以及相应的使能信号。MemCtrl 中只涉及组合逻辑。

异常检测模块 Exception 接收一个八位的信号 Exc 来判断是否存在异常并依据优先级决定异常处理顺序。同时，异常检测模块也接收 CP0 中传来的相关数据和地址，根据需要传出异常处理程序入口地址，相关控制信号以及刷新流水线信号 flashexcept。

MemCtrl 接口说明

Name	Width	Direction	Discreption
write_data	31...0	Input	处理前的前级要写入内存的数据
mem_en	1	Output	读写使能信号
memsel	3...0	Output	控制写的字节的使能信号
final_addr	31...0	Output	前级要写入的数据的写地址
final_wdata	31...0	Output	处理后的要写入内存的数据

在写入数据的时候,根据指令的不同,写入的数据也会有所差别。例如,SB 指令是写入最低位的字节,SH 指令则是写入低半字。因此,在 MemCtrl 中,我们通过 Mem_en 和 memsel 来决定读写的字节。

内存写入数据说明

Mem_en	memsel[3]	memsel[2]	memsel[1]	memsel[0]	写入的字节
0	x	x	x	x	不能进行读写
1	0	0	0	1	最低字节
	0	0	1	0	次低字节
	0	1	0	0	次高字节
	1	0	0	0	最高字节
	0	0	1	1	低半字
	1	1	0	0	高半字
	1	1	1	1	全字

CP0 接口说明

Name	Width	Direction	Discreption
clk	1	Input	时钟信号
rst	1	Input	复位信号,高有效
we_i	1	Input	CP0 写使能
waddr_i	4...0	Input	写入 CP0 的数据的地址
raddr_i	4...0	Input	读入 CP0 的数据的地址
data_i	31...0	Input	传入 CP0 的数据
excepttype_i	31...0	Input	异常类型
current_inst_addr_i	31..0	Input	当前指令的地址
is_in_delaysolt_i	31..0	Input	是否由延迟槽指令导致异常
bad_addr_i	31..0	Input	导致异常的非对齐地址
data_o	31..0	Output	从 CP0 中取出的数据
count_o	31..0	Output	处理器计数周期
cpmpare_o	31..0	Output	定事中断控制
status_o	31..0	Output	处理器状态和控制寄存器
cause_o	31..0	Output	处理器状态控制
epc_o	31..0	Output	异常指令的 pc 值
config_o	31..0	Output	配置寄存器,用于设置 CPU 参数
prid_o	31..0	Output	
badvaddr	31..0	Output	记录最近一次地址相关异常的地址
timer_int_o	31..0	Output	

Exception 接口说明

Name	Width	Direction	Discreption
int_hard	5...0	Input	异常是否为硬件中断
ri	1	Input	保留指令异常
break	1	Input	断点例外
syscall	1	Input	系统调用
overflow	1	Input	整型溢出例外
addrErrorSw	1	Input	地址错例外 (写数据)
addrErrorLw	1	Input	地址错例外 (读数据)
pcError	1	Input	地址错例外 (取指令)
eretM	1	Input	特权指令
pcM	31...0	Input	PC 值
alu_outM	31...0	Input	算术逻辑运算的结果
except_type	31...0	Output	异常类型
flush_except	1	Output	刷新流水线的使能信号
pc_except	31...0	Output	异常指令的 PC 值
pc_trap	1	Output	是否跳入异常指令处理入口的控制信号
badvaddrM	31...0	Output	记录最近一次地址相关异常的地址

2.1.8 Writeback 阶段分析

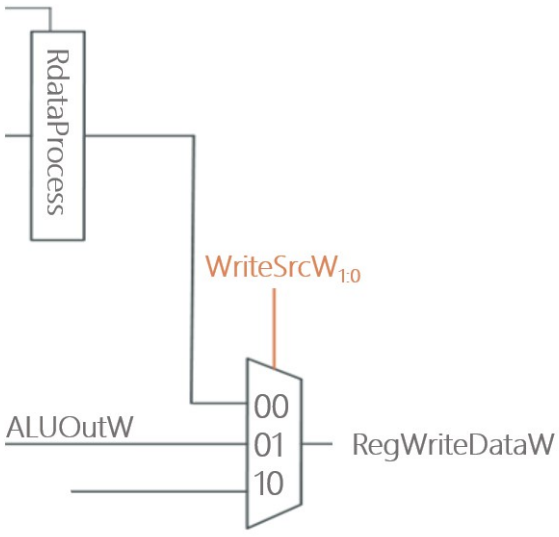

图 6: Writeback 阶段线路图

Writeback 阶段负责将数据写回寄存器中。读取的数据在此处进入 Rdataprocess 中进行处理，跟 Memory 阶段中对写入数据的处理类似，在该模块中，针对不同的指令需求，读出需要的字，半字或字节。最终通过控制信号 $\text{WriteSrc}_{1:0}$ 来决定写回的数据为 Alu 的算术逻辑运算结果，读入的数据还是从 CP0 中取出的数据。

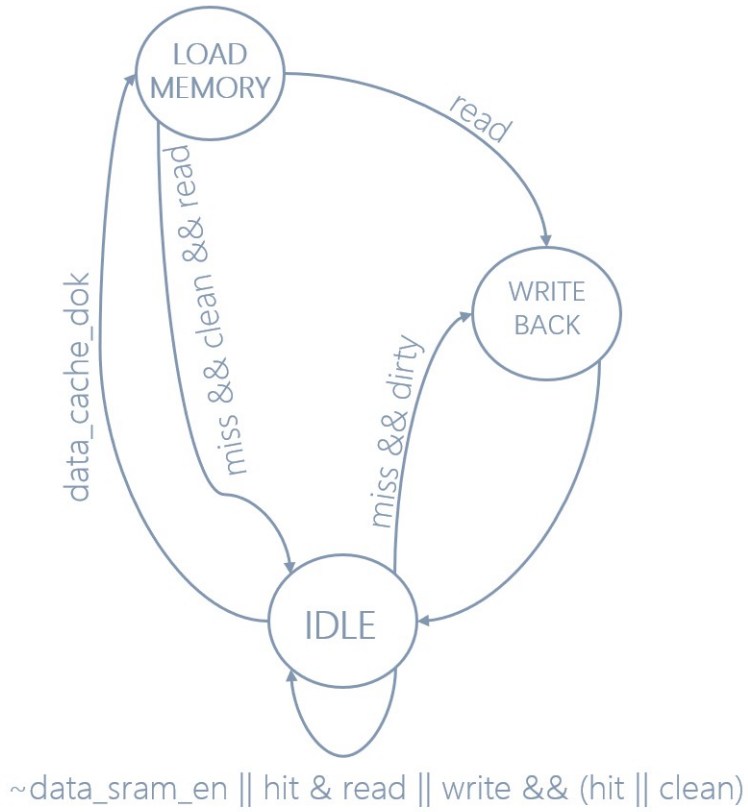

图 7: D_Cache 状态转移图

Writeback 阶段接口说明

Name	Width	Direction	Discreption
read_data	31...0	Input	原始的读入数据
addr	31...0	Input	读入数据的地址
final_rdata	31...0	Input	处理后的读入数据

2.2 Cache

如图 (7) 所示,Cache 模块由 i_cache(指令 cache) 与 d_cache(数据 cache) 组成.cache 使用 vivado 内部 IP 核 Dram(Dsitribute Ram) 实现。cache 主要包含三部分,分别为 Valid, Tag, Data. 其中 d_cache 还增加了 dirty 位,用于缺失处理;cache 内部还包括了 cache 控制部分和 hit 比对部分。当地址的 Tag 与 DTag 相同地址的 Tag 一致且 DValid 相同地址为 1 时,Cache 命中,否则对内存进行操作。

由于采用的是写回方式,每次写操作仅对 cache 写入,当发生读缺失时,首先通过 dirty 位判断当前数据块是否修改,如果 dirty 位为 1,则将当前数据亏啊写回到内存中,然后从内村中读取目标数据块写入到 cache 中,如果 dirty 位为 0,则直接将目标数据块从内存中读取到 cache 中。对于写操作,同样判断当前数据块位是否为 1,如果为 1,则将当前数据块写入到内存中,然后将目标数据写入到内存中。

3 Soc

3.1 SoC up 结构

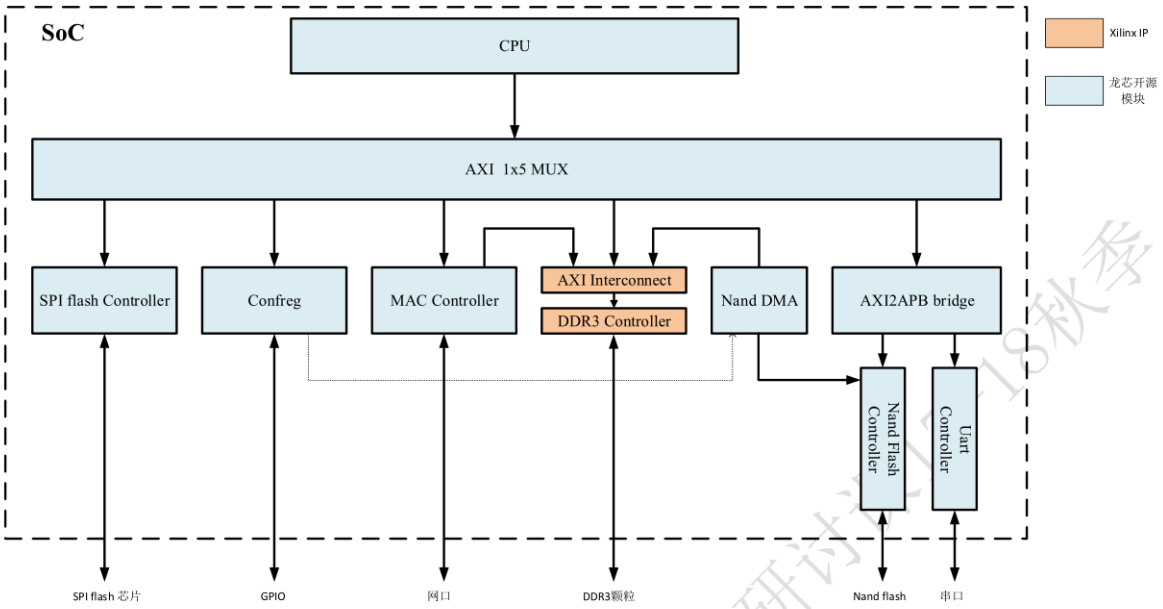

图 8: SoC_up.png

图片引用自 A10_SoC_up 使用环境说明.pdf

4 操作系统引导程序——PMON

由于在 CPU 上运行出操作系统之前,需要有一个引导程序。而龙芯平台提供了一个名为 pmon 的系统,可作为引导。但在运行 pmon 前需要先编译出可在 cpu 上运行的指令。实验的环境是基于 ubuntu 的,在 ubuntu 下解压 pmon_archlab.tar.gz,其中包括了所有的 pmon 的文件及其代码。

4.1 交叉工具链

下载 gcc-4.3-ls232.tar.gz 包。

下载后解压会得到一个 opt 的文件:

[abc@www]$ sudo tar -zxvf gcc-4.3-ls232.tar.gz

[abc@www]$ cd opt

[abc@www]$ sudo cp -r gcc-4.3-ls232 /opt/

[abc@www]$ echo “export PATH=/opt/gcc-4.3-ls232/bin:$PATH”» /.bashrc

[abc@www]$ source /.bashrc

对于 64 位系统,还要安装 lsb-core:

[abc@www]$ sudo apt-get install lsb-core

另外还需要的安装包:

[abc@www]$ sudo apt-get install lib32z1

完成上述工作后如果可以输入 mipsel-linux-gcc -v 命令,如果可以正确查看版本号则说明配置正确。

4.2 pmon 编译与反编译

进入 pmon_archlab 文件配置环境。

需要的环境:

+ makedepend(sudo apt-get install xutils-dev)

+ ncurses-devel(sudo apt-get install libncurses5-dev)

+ bison(sudo apt-get install bison)

+ flex(sudo apt-get install flex)

环境安装完成之后。进入到 tools/pmoncfg/中执行 make 生成 pmoncfg 可执行文件将生成的 pmoncfg 的路径添加到 PATH 中。

然后进入/zloader.ls1b/目录,执行./g。即可完成 pmon 的编译。编译后会得到一个 gzrom.bin 文件,该文件为需要烧写到 flash 里的二进制文件。

同时编译会到的 gzrom 文件,该文件为 elf 文件,可以使用命令 mipsel-linux-objdump 对其进行反汇编。(mipsel-linux-objdump -S gzrom)

如果需要删除该目录下编译生成的文件,请执行"make clean"。下图为 pmon 二进制文件编译的执行流程。

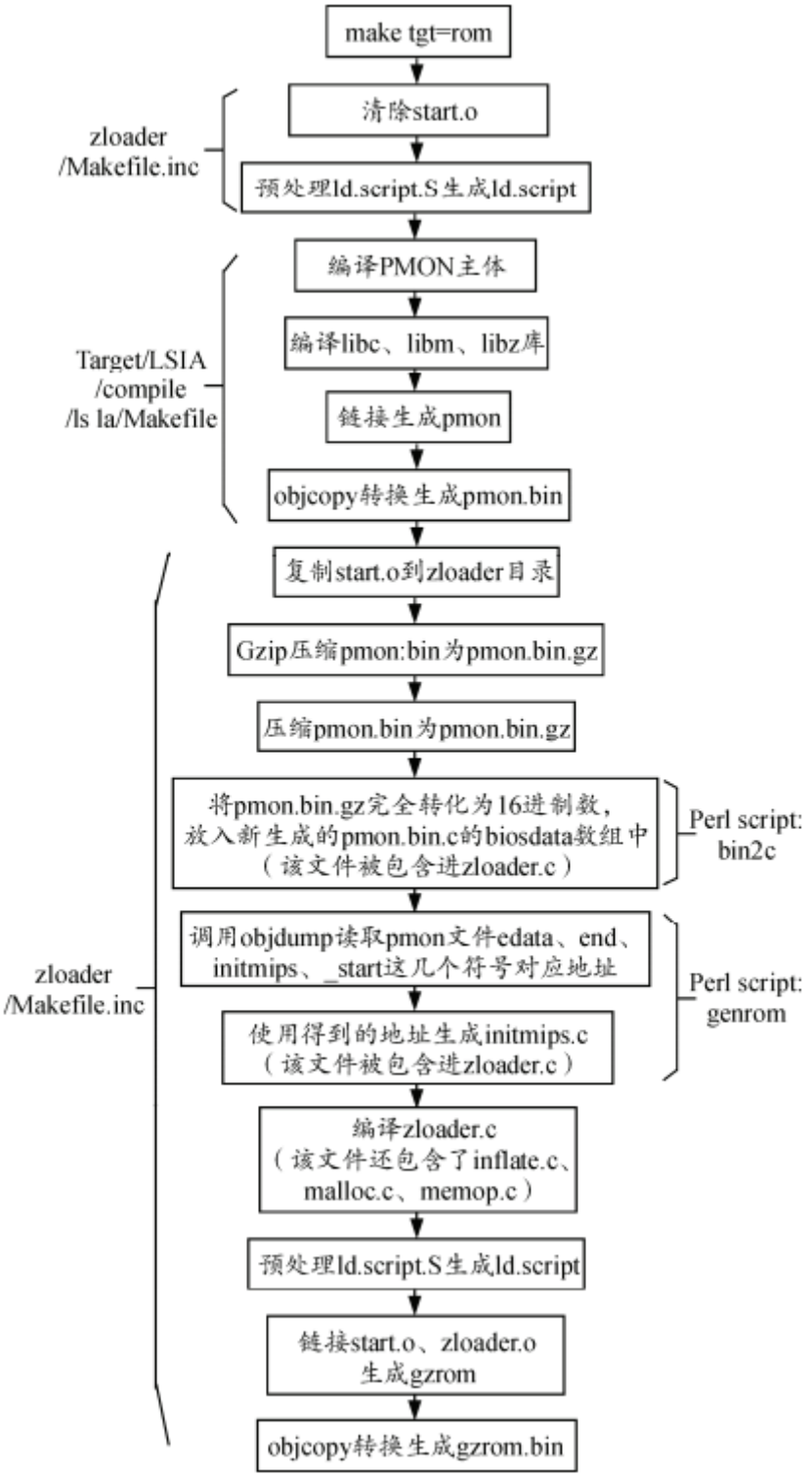

图 9: pmon 二进制文件编译的执行流程图

4.3 pmon 启动过程

在启动 pmon 运行在 CPU 上前,需要了解 pmon 的启动流程。按照龙芯的架构要求,0xA0000000 0xBFFFFFFF(kseg1 段)是唯一在系统重启时能够正常工作的内存映射空间,并且地址为 0Xbfc00000 为重新启动时的入口向量,即 CPU 启动后第一条执行语句

的所在地址。对于 kseg1 段的地址,通过将最高 3 位清零的方法,映射到响应的物理地址。

启动地址的映射关系,完全由硬件实现。在这之后,就需要靠 start.S 所包含的源码来处理接下来的初始化工作,然后跳转到 C 代码的入口继续执行。启动过程如下图。

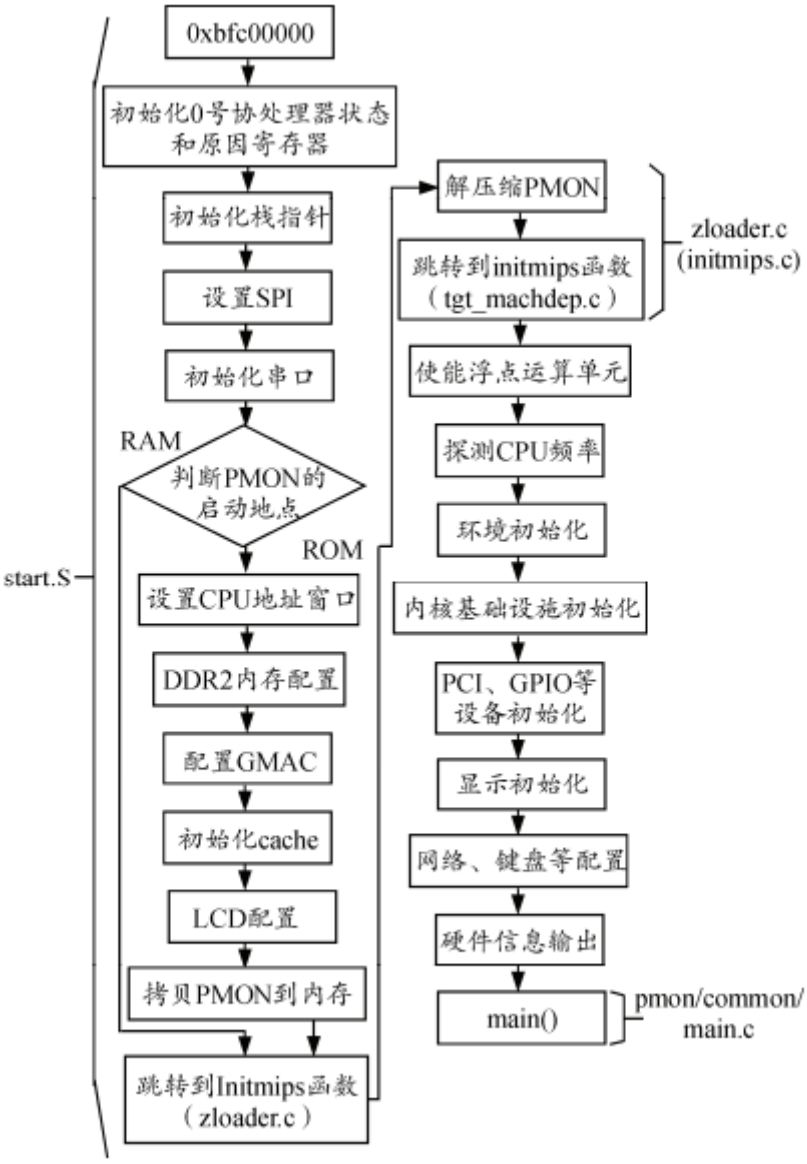

图 10: PMON 启动流程图

从图中可以首先看出:一般情况下,pmon 会直接在 FLASH 中运行,一段时间后才会将自身拷贝到内存中,并且还需要通过 zloader 将压缩过的 pmon 主程序解压。

4.4 上板过程

烧写步骤:

1) flash 芯片正确放置 FPGA 开发板上。

2) FPGA 开发板与电脑连接下载线、串口线。

3) FPGA 板上电,如正常下载 bit 流文件一样下载 programmer_by_uart.bit 至 FPGA 上。

4) 串口软件,波特率选为 230400,使用 xmodem 模式传输 binary 文件。

5) 串口连接正常后根据提示,键盘输入 x 表示开始 xmodem 传输。

6) 串口软件等待传输完成。

建议使用 ECOM。ECOM 传输 xmodem 的进度会弹出一个界面进行显示,可以看到每个传输包的情况。而 SecureCRT 只会在串口界面显示传输百分百。

4.5 bugs

4.5.1 问题一

问题:在初始化串口成功输出第一句输出后,PMON 报异常并卡死

解决过程:在 start.S 添加输出 (使用已定义的 PRINTSTR 宏),发现拷贝 PMON 到内存阶段出错。将拷贝的指令输出,发现与参照的正确输出不同。然后通过 ILA 抓取 CPU 访存阶段的指令与地址,发现地址正确,然而返回的数据不正确。最后发现 CPU 在访问外设数据时 (访问内存与外设采用不同的模块) 的逻辑有错误,而之前的功能测试与性能测试由于访问外设数据很少,因此没有测出来。

4.5.2 问题二

问题:解决上述问题后,PMON 成功进入命令行交互界面。在执行配置 ip 指令 ifconfig 后,屏幕输出一些信息后卡死,且无法返回命令行。

解决过程:在这里我们首先怀疑 PMON 虽然已经进入命令行,但之前的初始化过程可能并未执行正确。于是我们花费了很长时间去阅读 PMON 的 start.S 源码。同时我们也通过抓取波形,发现 PMON 是进入了 start.S 的异常处理程序,且由于 s0 寄存器 (用于重定位,具体可看 pmon 的启动过程) 的错误,在执行 stringserial 输出字符串时,在初始部分会形成一个死循环,且无法输出。此时,我们意识到 PMON 在执行内存拷贝完毕后,虚拟地址便跳转到了起始地址为 0x8000000 的 kseg0 段,在该段内,PMON 执行需经过缓存。而之前 PMON 都在 Kseg1 段执行且不经过缓存。查阅《see mips run》关于异常的一章,我们了解到异常入口点与 SR(BEV) 和 EBase 有关。复位或上电时,由于 cache 还未正确初始化,因此程序只能执行在 kseg1 段 (冷启动异常入口点 0xbfc00000 即在 kseg1 段),而为了提高异常处理的性能,故应该能够在正确初始化 cache 后将异常入口点平移到 kseg0 段。而 SR 的 BEV 位即实现该作用,当 BEV 为 1 时,异常入口点为固定值,当 BEV 为 0 时,异常入口点则由 EBase 寄存器加一个固定偏移产生。了解了相关知识之后,我们添加实现了 EBase 寄存器,问题解决。PMON 可以成功配置 ip,并可以 ping 通局域网内主机。

4.5.3 问题三

问题:配置完 ip 后,用电脑搭建 tftp 服务器,然后通过 PMON 执行 load 指令加载 Linux。结果 PMON 报保留指令异常。

解决过程:为了确定是哪条保留指令,我们研究了一下 PMON 的使用方法。发现 PMON 可以通过 d1, d2, d4 等指令输出内存某地址的内容。(并且后面发现 l 指令可以直接输出具体指令而不仅是 16 进制数值)。最后我们确定是 SYNC 指令未实现。在查阅 MIPS 指令集手册关于 SYNC 的说明后发现,由于我们的 CPU 是单核的经典五级流水结构的

cpu,不存在调节访存指令执行顺序,以及多核同步问题。因此可以直接将 SYNC 实现为 nop 指令。解决问题后,PMON 执行 load 成功。

5 run linux

5.1 linux 启动过程

linux 内核分为两种,分别为经过压缩的 vmlinuz(带解压程序) 和未经过压缩的 vmlinux,我们得到的是未经过压缩的 vmlinux,以下说明以该版本为准。PMON 作为引导程序,在执行 g 命令后,便会跳转到 linux 内核的入口点 (EP, Entry Point),该地址在链接时由 ld 程序写入生成的可执行文件 (ELF 文件)。linux-2.6.32/arch/mips/kernel/head.S 中的 kernel_entry(KE) 函数便处于 EP 处,KE 函数又马上会调用 kernel_entry_setup 与 setup_c0_status_pri 函数,最终调用 init/main.c 中的 start_kernel 函数。此时已经进入了 C 环境,而与架构无关了。之后过程略.....

5.2 bugs

5.2.1 问题一

问题:load 成功后,执行 g 指令执行 linux 失败,报保留指令。

```
8707af70 4448f800 cfc1    t0,$31
PMON> l 0x805dafd8
   cpu_probe+0x18c  8cc2c2e0 lw      v0,-15648(a2)# 0xffffc2e0
   cpu_probe+0x190  2c420001 sltiu   v0,v0,0x1    # 1
   cpu_probe+0x194  00028036 tne     zero,v0
   cpu_probe+0x198  8e030018 lw      v1,24(s0)    # 0x18
   cpu_probe+0x19c  2c630001 sltiu   v1,v1,0x1    # 1
   cpu_probe+0x1a0  00038036 tne     zero,v1
   cpu_probe+0x1a4  3c048072 lui     a0,0x8072    # 32882
   cpu_probe+0x1a8  2490c360 addiu   s0,a0,0xc360 # -15520
   cpu_probe+0x1ac  8e020008 lw      v0,8(s0)     # 0x8
   cpu_probe+0x1b0  30420020 andi    v0,v0,0x20   # 32
more...  break!
PMON> l 0x805db740
   cpu_probe+0x8f4  ae03001c sw      v1,28(s0)    # 0x1c
   cpu_probe+0x8f8  ae020008 sw      v0,8(s0)     # 0x8
   cpu_probe+0x8fc  08176bf6 j       cpu_probe+0x18c # 0x805dafd8
   cpu_probe+0x900  ae050020 sw      a1,32(s0)    # 0x20
   cpu_probe+0x904  2403002b li      v1,0x2b      # 43
   cpu_probe+0x908  3c028056 lui     v0,0x8056    # 32854
   cpu_probe+0x90c  3c068072 lui     a2,0x8072    # 32882
   cpu_probe+0x910  34e4008b ori     a0,a3,0x8b   # 139
   cpu_probe+0x914  ae030018 sw      v1,24(s0)    # 0x18
   cpu_probe+0x918  2442a15c addiu   v0,v0,0xa15c # -24228
more...  break!
PMON>
```

图 11: reserve instruction exception.

解决过程:通过 l 查看内存,发现是 TNE 未实现,增加 TNE 等一系列内陷指令后。又因 cfc1 指令报保留指令异常。因为我们并未实现协处理器 1,即浮点协处理器 (CP1),因此便让其报 cPu 异常 (协处理器不可用异常)。在此期间我们询问了比赛群里的其他队伍。他们认为我们出现该错误的原因是 linux 需要通过软件的方式实现浮点运算,因此在第一次遇到 COP1 指令时需要报 cPu 异常,在之后 linux 便会通过软件模拟浮点运算。可是在我们更改之后,仍没有解决问题,仍然会因为异常返回到 PMON 的命令行,而 g 之后却无任何输出。此问题是在解决一下问题后才一并解决的。

5.2.2 问题二

问题:已知在 linux 启动不久便会调用 start_kernel 函数。我们在 start_kernel 函数的第一句添加一条 printk 语句,可结果仍无任何输出。

解决过程:PMON 的 r 指令作用是输出寄存器的值,我们之前也发现 PMON 的 r 指令无法输出。同时我们通过抓取 cpu 运行 g 后第一条发生的异常,发现 cpu 是在 cpu_probe 函数中产生了一个异常。由于一直没有什么进展,我们仔细翻看了 2018 年国科大写的《myCPU 启动 Linux 指南》,决定把其中我们还未实现的指令,cp0 寄存器全部都先实现,在达到他的所有要求后再看能不能成功。期间我们找到了之前非对齐指令实现的错误,增加了乘累加、ll 和 sc、条件赋值、branchlikely 几类指令,增加了 random、pagemask、wired(都是和 TLB 有关) 寄存器。到此除了 Cache 相关的几条指令我们没有实现外,其他指令和寄存器都实现了。由于没办法使用 printk 函数,我们没办法知道 cpu 到底执行到了哪里。然后我们便想到在 C 的代码中嵌入 mips 汇编的 syscall 指令,让其必定产生异常退出,这样便可以根据 cpu 报的异常是否为 sys 异常判断该指令是否执行。依靠这种“原始”的方式我们最终确定了异常是在 cpu_probe 的哪个地方产生的,并且发现在执行 cpu_probe 调用的一个函数的一个分支时,并没有按照我们想像地执行。以此我们发现我们的 PRID 寄存器的值是错误的。在改成 LOONGSON232 的 0x00004220 后,奇迹般的便输出了 linux 的打印信息。(printk 并不会在调用之后立即打印,这个我们使用实例 cpu 实验过了)。

5.2.3 问题三

问题:在屏幕上出现 linux 的打印信息后,linux 停在了某处.....

```
PMON> load tftp://192.168.100.1/vmlinux
Loading file: tftp://192.168.100.1/vmlinux (elf)
0x80200000/5350380 + 0x8071a3ec/169220(z) + 6926 syms\
Entry address is 802041f0
PMON> g console=ttyS0,115200 rdinit=sbin/init
   zero      at       v0       v1       a0       a1       a2       a3
 00000000 10000000 00000402 00000001 00000003 a7dffd78 a7dffd88 870af5b0
    t0       t1       t2       t3       t4       t5       t6       t7
 00004000 00000003 00000001 805c4fa0 00000001 00000033 00000018 ffffffff
    s0       s1       s2       s3       s4       s5       s6       s7
 80730000 805aff08 00000000 00000000 00000000 00000000 00000000 00000000
    t8       t9       k0       k1       gp       sp       s8       ra
 00000002 00000000 00000000 00000000 805ac000 a7dffd58 00000000 8707ad48
Linux version 2.6.32.13 (xing@xing-zhang) (gcc version 4.3.0 (GCC) ) #61 Fri Oct 21 01:11:12 CST 2016
busclock=99999999, cpuclock=99999999, memsize=128, highmemsize=0
bootconsole [early0] enabled
CPU revision is: 00000000 (ICT Loongson-232)
Determined physical RAM map:
 memory: 00000000 @ 00000000 (usable)
Initrd not found or empty - disabling initrd
```

图 12: The lastest progress.

解决过程:TO BE CONTINUE...

参考文献

[1]https://blog.csdn.net/wangyao199252/article/details/74938761

[2]《A10_SoC_up 使用环境说明》

[3]《Lab07-1_CPU 初始化程序完善》

[4]《See MIPS Run (2nd ed.) [Sweetman 2006-10-31]》

[5]《第二届-myCPU 启动 Linux 指南-国科大》

[6]《基于龙芯 1A 平台的 PMON 源码编译和启动分析》