

版本历史

文档更新记录		文档名:	Lab07-1_CPU 初始化程序完善	
		版本号	V0.1	
		创建人:	计算机体系结构研讨课教学组	
		创建日期:	2017-12-04	
更新历史				
序号	更新日期	更新人	版本号	更新内容
1	2017/12/05	邢金璋	V0.1	初版。

文档信息反馈: xingjinzhang@loongson.cn

1 实验七第一阶段 CPU 初始化程序完善

在学习并尝试本章节前，你需要具有以下环境和能力：

- (1) 较为熟练使用 Vivado 工具。
- (2) 一定的汇编编程能力。

通过本章节的学习，你将获得：

- (1) TLB、CACHE 的结构，及其初始化原理和方法。
- (2) 串口的工作原理和波特率计算方法。
- (3) PMON 的简单使用。

在本章节的学习过程中，你可能需要查阅：

- (1) 认真阅读任务书。
- (2) See mips run 书籍。
- (3) MIPS 官方手册。

在开展本次实验前，请确认自己知道以下知识：

1.1 实验目的

1. 理解计算机系统启动时的初始化过程
2. 理解 TLB、CACHE 的结构，掌握其初始化的原理和方法
3. 了解串口的工作原理，掌握波特率的设置方法
4. 认识 PMON 并掌握简单的命令
5. 进一步提高 MIPS 汇编编程的能力
6. 进一步提高软件的调试经验

1.2 实验设备

1. 装有 Xilinx Vivado、MIPS 交叉编译环境的计算机一台。
2. 龙芯体系结构教学实验箱（Artix-7）一套。

1.3 实验任务

补充完 PMON 源代码中被删除的 Cache 初始化、TLB 初始化、串口初始化部分，编译后在一个已完成的 SoC 设计上正常启动并成功装载和启动 Linux 内核。

要求刚下载完 bit 文件后，PMON 运行时打印的信息无“TLB init Error!!!”和“cache init Error!!!”。且能正确装载并启动 Linux 内核。

1.4 实验环境

参考文档“A10_SoC_up 使用环境说明”。

1.5 实验检查

Lab7 第一阶段实验在 2017 年 12 月 12 日进行检查。现场为上板检查。

上板检查，要求现场下载 bit 文件。只看刚下载完的 PMON 运行情况。之后演示加载并启动内核的方法。

1.6 实验提交

Lab7 第一阶段实验作品提交截止日期是 2017 年 12 月 12 日 18:00。

(1) 纸质档提交

提交方式：课上现场提交，每组都必须要有。

截止时间：2017 年 12 月 12 日 18:00。

提交内容：纸质档 lab7-1 实验报告。

实验报告模板参考“A06_实验报告模板_v0.2”。

(2) 电子档提交

提交方式：打包上传到 Sep 课程网站 lab7-1 作业下，每组都必须要有。

截止时间：2017 年 12 月 12 日 18:00。

提交内容：电子档为一压缩包。

第一阶段提交目录层次如下（请将其中的“**组号**”，替换为本组组号）。

lab7-1_ 组号 /	目录，lab7-1 作品。
lab7-1_ 组号 .pdf/	Lab7-1 实验报告，实验报告模板参考“A06_实验报告模板 v0.2
--start.S	完善初始化程序后的源码

1.7 实验说明

本次实验基于我们提供的一个完整的 SoC，该 SoC 中集成了一个完备的带有 TLB 和 Cache 的双发射 32 位 CPU(以下简称 CPU232)，并且集成了 SPI flash 控制器、串口(UART)控制器、网卡(MAC)控制器和内存(DDR3)控制等。

本次实验内容为完善 PMON(类似于 BIOS 程序)，PMON 是作为软件代码需要编译后烧写到 SPI flash 芯片中。而硬件部分是直接提供该 SoC 使用 Vivado 生成的 bit 流文件，可直接烧写到开发板上，运行 SPI flash 上的 PMON 即可。故本次实验只涉及到 MIPS 汇编编程，且无法通过 Xsim 仿真进行调试，只能使用软件调试手段。

1.7.1 PMON 说明

在下载 pmon_lab7.tar.gz 后，请在 Linux 环境里解压，且不能再与 windows 的共享目录里解压。因为 pmon_lab7 里存在符号链接，windows 的文件系统格式不支持符号链接。

PMON 目录比较复杂。但在本次实验中主要只关注 **pmon_lab7/zloader.ls1b/** 和 **pmon_lab7/Targets/LS1B/ls1b/**两个目录，其中前者为执行编译的目录，后者为本次实验需要修改的启动文件 start.S 的目录。

(1) 编译 PMON

进入 **pmon_lab7/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。

如果需要删除该目录下编译生成的文件，请执行“**make clean**”。

(2) 修改启动文件 start.S

本次实验完善 PMON 初始化程序，就是对 start.S 进行完善。

进入 **pmon_lab7/Targets/LS1B/ls1b/**目录，即可看到 start.S 文件，为一个汇编程序文件。

其中在第 671 行，需要添加串口控制器设置波特率的代码，如下。

```
663 LEAF(initserial)
664     la v0, COM1_BASE_ADDR
665 1:
666     li v1, FIFO_ENABLE|FIFO_RCV_RST|FIFO_XMT_RST|FIFO_TRIGGER_4
667     sb v1, NSREG(NS16550_FIFO) (v0)
668     #####
669     ##TODO: CODE 1: set baud rate
670     #####
671     nop
672     #####
673     ##TODO: CODE 1: end
674     #####
675     li v1, CFCR_8BITS
676     sb v1, NSREG(NS16550_CFCR) (v0)
677     li v1, MCR_DTR|MCR_RTS
678     sb v1, NSREG(NS16550_MCR) (v0)
679     li v1, 0x0
680     sb v1, NSREG(NS16550_IER) (v0)
681
682     j ra
683     nop
684 END(initserial)
```

在第 755 行，需要添加 cache 初始化的代码，如下。注意第 761 行的“**jr ra**”用于返回调用 cache_init 的母函数中。

```
750 cache_init:
751 #if 1
752 #####
753 ##TODO: CODE 2: cache init
754 #####
755     nop
756     #####
757     ##TODO: CODE 2: end
758     #####
759 #endif
760 cache_init_finish:
761     jr ra
762     nop
```

在第 485 行，需要添加 TLB 初始化的代码，如下。

```
480 LEAF(CPU_TLBInit)
481 #if 1
482 #####
483 ##TODO: CODE 3: TLB init
484 #####
485     nop
486     #####
487     ##TODO: CODE 3: end
488     #####
```

```

489 #endif
490 TLBInit_finish:
491     jr    ra
492     nop
493 END(CPU_TLBInit)

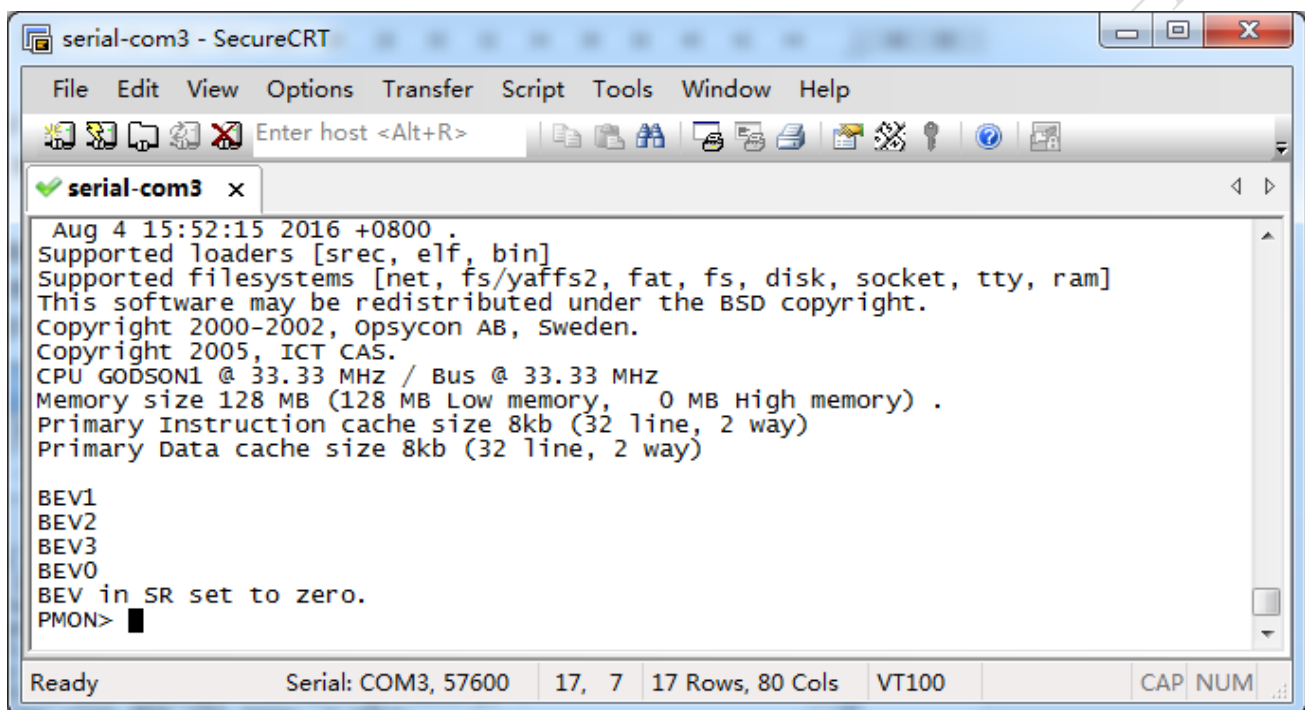
```

修改完 start.S，需要编译时，请回到 **pmmon_lab7/zloader.ls1b/**目录进行编译，不需要 **make clean**，可以直接运行 **./g** 编译。

注意：本次实验的用于表示 cp0 寄存器的助记符是 **COP_0_*** 类型的。具体对应关系请查看 **pmmon_lab7/sys/arch/mips/include/cpu.h**。

(3) PMON 命令

如果 PMON 是完善的，则将编译后的 **gzrom.bin** 烧入 flash 芯片，并将 **archlab_lab7.bit** 下载到开发板上。连接上串口，打开串口软件，设置好波特率。则可以在串口窗口中看到 PMON 运行信息，运行成功后则会进入 PMON 提示符，此时可以输入 PMON 命令。



```

serial-com3 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+R>
serial-com3 x
Aug 4 15:52:15 2016 +0800 .
Supported loaders [srec, elf, bin]
Supported filesystems [net, fs/yaffs2, fat, fs, disk, socket, tty, ram]
This software may be redistributed under the BSD copyright.
Copyright 2000-2002, Opsycon AB, Sweden.
Copyright 2005, ICT CAS.
CPU GODSON1 @ 33.33 MHz / Bus @ 33.33 MHz
Memory size 128 MB (128 MB Low memory, 0 MB High memory) .
Primary Instruction cache size 8kb (32 line, 2 way)
Primary Data cache size 8kb (32 line, 2 way)

BEV1
BEV2
BEV3
BEV0
BEV in SR set to zero.
PMON>

```

比如，本实验 SoC 中具有 MAC 控制器，PMON 中也有 MAC 驱动，则我们输入命令 **ifconfig dmfe0 10.90.50.44** 则可以给开发板上的网卡配置 IP 为 **10.90.50.44**（具体需配置的 IP 请查阅同网段的电脑 IP），假设同网段的电脑 IP 为 **10.90.50.43**，则可以继续输入命令 **ping 10.90.50.43** 用于查看网络是否成功接入。Linux 在 ping 网络是会一直发 ping 包，可以 **Ctrl+C** 取消 ping。运行结果如下：

```

PMON> ifconfig dmfe0 10.90.50.44
rx ring 70acee0
tx ring 70acf60
DE4X5_BMR= fe000000
DE4X5_TPD= 0
DE4X5_RRBA= 70acee0
DE4X5_TRBA= 70acf60
DE4X5_STS= f0660004
DE4X5_OMR= 32002242
TX error status2 = 0x00000000
After setup
DE4X5_BMR= fe000000
DE4X5_TPD= 0

```

```

DE4X5_RRBA= 70acee0
DE4X5_TRBA= 70acf60
DE4X5_STS= f0660004
DE4X5_OMR= 32002242
PMON> ping 10.90.50.43
PING 10.90.50.43 (10.90.50.43): 56 data bytes
64 bytes from 10.90.50.43: icmp_seq=0 ttl=64 time=3.708 ms
64 bytes from 10.90.50.43: icmp_seq=1 ttl=64 time=2.331 ms
64 bytes from 10.90.50.43: icmp_seq=2 ttl=64 time=2.235 ms

--- 10.90.50.43 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 2.235/2.750/3.708 ms
PMON>

```

网络配置好了，在本实验中需要通过网络下载 Linux 内核，需要先搭建好 tftp 服务器，假设搭建的 tftp 服务器 IP 为 10.90.50.43，将要下载 Linux 内核（vmlinux，已包含在 lab7 实验包中）放到 tftp 服务器的根目录下，输入命令“**load tftp://10.90.50.43/vmlinux**”即可 load 内核进入 FPGA 上的内存。

```

PMON> load tftp://10.90.50.43/vmlinux
Loading file: tftp://10.90.50.43/vmlinux (elf)
0x80200000/5350380 + 0x8071a3ec/169220(z) + 6926 syms\
Entry address is 802041f0
PMON> g console=ttyS0,115200 rdinit=sbin/init

```

上表中显示 load 成功了，输入命令“**g console=ttyS0,baudrate rdinit=sbin/init**”即可运行该内核，命令中 **baudrate** 需为数字，即为串口控制器设置的波特率，设置不对时，串口显示字符为乱码。

```

.....
mount: mounting n on /proc/bus/usb failed: No such file or directory
mdev: /sys/class: No such file or directory
login
Godson2@[/]>ls
bin  etc  lib  mnt  root  sys  usr
dev  hello.c linuxrc proc  sbin  tmp
Godson2@[/]>cd root/
Godson2@[~]>vi 1.txt
Godson2@[~]>ls
1.txt
Godson2@[~]>cat 1.txt
hello,world!
Godson2@[~]>

```

当运行 Linux 内核成功后，会出现“**Godson2@[~]>**”提示符，可以使用常用的 Linux 命令，如上表。

PMON 中还有可以通过命令实现在线烧写 flash，命令为使用命令“**load -r -f bfc00000 tftp://10.90.50.43/gzrom.bin**”。同样需先搭建 tftp 服务器，并将要烧写到 flash 中的 bin 文件拷贝到 tftp 服务器根目录下，并为开发板网卡配置好 IP，运行结果如下。

```

.....
BEV in SR set to zero.
PMON> ifconfig dmfe0 10.90.50.44
rx ring 70acee0
tx ring 70acf60
DE4X5_BMR= fe000000

```

```

DE4X5_TPD= 0
DE4X5_RRBA= 70acee0
DE4X5_TRBA= 70acf60
DE4X5_STS= f0660004
DE4X5_OMR= 32002242
TX error status2 = 0x00000000
After setup
DE4X5_BMR= fe000000
DE4X5_TPD= 0
DE4X5_RRBA= 70acee0
DE4X5_TRBA= 70acf60
DE4X5_STS= f0660004
DE4X5_OMR= 32002242
PMON> load -r -f bfc00000 tftp://10.90.50.43/gzrom.bin
Loading file: tftp://10.90.50.43/gzrom.bin (bin)
|
Loaded 299024 bytes

Programming flash 83000000:49010 into bfc00000
Verifying FLASH. No Errors found.
PMON>

```

需要注意的时，由于我们的运行的 PMON 是烧写到开发板上可插拔的 flash 芯片中的，在线烧写的 bin 文件也是烧写到该 flash 芯片中，也就覆盖了原有的 PMON，但在线烧写完成后，FPGA 上依然可以运行 PMON，因为 PMON 是拷贝到内存中去执行的。这时复位开发板，就可以运行新烧写到 flash 芯片中的软件程序。

(4) PMON 在线烧写

在本节第(3)点中说明了可以运行 PMON 在线烧写 flash，本次实验由于需要多次修改 PMON，但线上烧写比较麻烦，因而可以使用 PMON 在线烧写 PMON。

首先，需要确保串口初始化部分已调通，否则运行 PMON，主机的串口界面不是没反应就是乱码。

其次 TLB 和 Cache 初始化代码就算有错误，PMON 也可以正确运行且能成功加载 Linux 内核并启动，这时因为 FPGA 上所有 ram 自己有初始值(即有确定的高或低电压)，故不用初始化也可以正确读到一个确定的值，使得 CPU 运行不会出错。而在实际芯片上，不初始化，其电压就不确定，故其值就是不确定，存在 CPU 运行出错的风险。

基于 FPGA 上 TLB 和 Cache 不用初始化就能用，故我们在调试 TLB 和 Cache 初始化代码时，每次修改完 PMON，可以使用在线烧写 PMON，大大便于实验的进行。

但既然不初始化 TLB 和 Cache 就能正确运行，那本次实验如何检测初始化的是否正确？PMON 里会自行检测 TLB 和 Cache 是否正确初始化了，如果 TLB 或 Cache 有一个初始化有误，则会打印 **Error** 提示信息。有一个初始化正确了，则会打印 **OK** 提示信息。如下表所示，就是 TLB 初始化正确，但 Cache 初始化错误。

```

.....
TLB init OK!!!

cache init Error!!!

ERROR!!!

ERROR!!!

ERROR!!!

```


!!!!!!!!!!!!!!!

.....

无论 TLB 和 Cache 是否初始化正确，PMON 都是可以正确运行的。故发现初始化有错后，修改 PMON 编译后，拷贝到 tftp 服务器根目录下，使用 laod 命令在线烧写即可。

本次实验的结果正确的评判，就是运行 PMON 打印的为如下信息：

.....
TLB init OK!!!

cache init OK!!!
.....

1.7.2 软件调试手段

本次实验可以使用的调试手段只有软件手段，且只有加打印的方法。start.S 中封装好了两类打印函数 PRINTSTR()和 hexserial，使用方法可以参考 269 行到 273 行的打印出 cp0 寄存器值的代码，如下。

```
269    PRINTSTR(" CONFIG=")
270    mfc0    a0, COP_0_CONFIG
271    bal hexserial
272    nop
273    PRINTSTR("\r\n")
```

可以看到 PRINTSTR()会直接打印括号里的字符串。而 hexserial 需要使用 a0 进行传参，并将 a0 里的值转化为 16 进制进行打印出来，比如，假设 COP_0_CONFIG 值为全 0，则上表的打印结果应为“CONFIG=00000000”。

在使用两类打印函数时需要注意，这两个函数也是使用汇编编写的，里面使用了 a0、a1、a2、a3、v0、v1、ra 寄存器，因而打印函数会对这 7 个通用寄存器进行修改，你要确保这种修改不会对调用该打印函数处的代码造成影响。特别是 Cache 和 TLB 初始化处是使用“jr ra”进行返回的，如果你在初始化代码中使用了打印函数，则必定会修改 ra 寄存器，初始化完成后则无法正确返回了。**如果使用了打印函数，请自行解决该问题。**

所有的打印函数都是通过串口输出到屏幕上的。故在使用打印函数前，必须先设置到串口波特率使得串口正常工作。

1.7.3 串口初始化

串口初始化即是对串口控制器里的寄存器进行赋值，使得串口可以正常工作。本次实验的串口初始化部分只要求完成波特率的设置。

(1) 内部寄存器

本次实验提供的 SOC 中串口控制器很好地兼容国际工业标准半导体设备 16550A，内部有 8 个控制寄存器，每个寄存器宽度为 8bit。设置波特率涉及到的寄存器如下：

串口控制器内部寄存器	偏移量	描述
UART_TLL	0x0	分频锁存器低 8 位，仅当 UART_LCR[7]为 1 时，偏移为 0x0 的寄存器才用作分频锁存器低位。
UART_TLH	0x1	分频锁存器高 8 位，仅当 UART_LCR[7]为 1 时，偏移 0x1 的寄存器才用作分频锁存器高位。
UART_LCR	0x3	线路控制器，仅当最高位为 1 时，偏移为 0x0、0x1 的寄存器才用作分频锁存器。

UART_TLL 寄存器详细描述如下：

名称：分频锁存器低位（需先设置 UART_LCR[7]为 1，才能用作分频锁存器低位）				
位宽：[7:0]				
复位值：0x00				
位域	位域名称	位宽	访问	描述
7:0	LSB	8	RW	存放时钟分频数的低 8 位。

UART_TLH 寄存器详细描述如下：

名称：分频锁存器高位（需先设置 UART_LCR[7]为 1，才能用作分频锁存器高位）				
位宽：[7:0]				
复位值：0x00				
位域	位域名称	位宽	访问	描述
7:0	MSB	8	RW	存放分频锁存器高 8 位。

UART_LCR 寄存器详细描述如下：

名称：线路控制器				
位宽：[7:0]				
复位值：0x03				
位域	位域名称	位宽	访问	描述
7	DLAB	1	RW	分频锁存器访问位。 '1' - 偏移 0x0, 0x1 的寄存器用作分频锁存器 '0' - 偏移 0x0, 0x1 的寄存器用作其他寄存器
6:0	-	7	RW	设置波特率时不需要用到，写 0 即可。

实验中的串口控制器的基址为 0xbfe40000，相应对 UART_TLH 寄存器的访问即对地址 0xbfe40001 的 load/store。由于这些寄存器均为 1 字节，故对其 load/store 需使用字节 load/store，即 LB/SB。

从上表中也可以看到 0xbfe40000 和 0xbfe40001 对应的寄存器在 UART_LCR[7]不同时，其意义也不一样，只有当 UART_LCR[7]为 1 时，才能组合为分频锁存器。

所谓分频锁存器即为时钟分频数的存放寄存器，显然 16bit 数 {UART_TLH, UART_TLL} 即为串口控制器对 CPU 时钟的分频。实验中 SOC 中 CPU 接的时钟频率为 33Mhz。

CPU 时钟频率记为 CPU_CLK ，串口控制器时钟频率记为 $UART_CLK$ ，则 $UART_CLK = CPU_CLK \div \{UART_TLH, UART_TLL\}$ 。

(2) 波特率计算

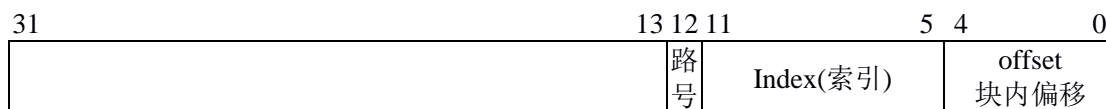
所谓波特率，是指串口通信时的速率，即单位时间是传输的波特数，一般单位为 bps(Bauds per second)，即每秒波特数。一般串口常见波特率有：9600, 14400, 19200, 38400, 57600, 115200, 230400, 380400, 460800, 921600 等。

波特率计算公式为： $UART_CLK \div 16$ 。注意被除数为串口的时钟频率，串口的时钟频率是由 CPU 的时钟频率分频得到的。

1.7.4 Cache 初始化

(1) Cache 结构

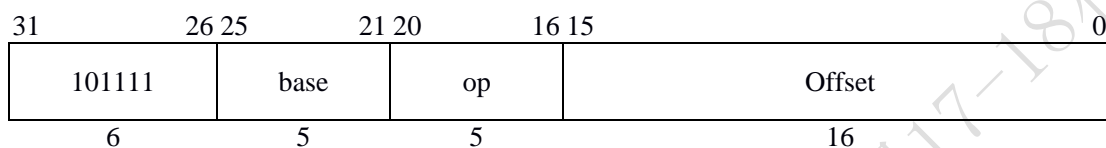
CPU232 中 Cache 模块只实现了 1 级指令 cache 和 1 级数据 cache，指令和数据 cache 结构相同，均为 2 路组相连，每路 128 个 Cache 块(即 Cache 行)，每个 Cache 块里 32 字节。因而 Cache 使用地址[31:0]中的[11:5]进行索引。由于实现页大小为 4KB，因而物理地址和虚拟地址的[11:0]是一样的，故用于索引的[11:5]直接使用虚拟地址即可。剩余的高 20 位([31:12])包含在 TAG 域中，高 20 位在虚拟地址和物理地址中就一样了，因而 TAG 域中存放的则是物理地址的高 20 位。地址形式如下：



地址的第 12 位指示了 Cache 的路号，在 cache 指令 Index_Store_Tag 用来初始化 Cache 时会涉及该位。

(2) Cache 指令

MIPS 中 Cache 指令的格式如下：



汇编格式：CACHE op, offset(base)

指令说明：依据 op 做不同的 Cache 操作

指令中的 base 域为寄存器号。

op 域的低两位，即[17:16]用于选择操作哪个 Cache，编码如下。

[17:16],op 低 2 位	操作对象
2'b00	1 级指令 cache
2'b01	1 级数据 cache
2'b10	3 级 cache
2'b11	2 级 cache

op 域的高三位，即[20:18]用于指示操作类似，编码如下：

[20:18],op 高 3 位	名称	解释
3'b000	Index invalidate	依据 index，设置 cache 行无效
3'b001	Index Load Tag	依据 index，读出 cache 行的存放的 Tag 到 cp0 寄存器 TagLo/TagHi,DataLo/DataHi
3'b010	Index Store Tag	依据 index，将 cp0 寄存器 TagLo/TagHi 写入 cache 行的 Tag 域
3'b011	保留	-
3'b100	Hit invalidate	依据地址命中 cache 时，设置行无效，不写回
3'b101	(数据 cache)Hit Writeback invalidate	依据地址命中 cache 时，设置行无效，需写回
	(指令 cache)Fill	从提供的地址取数填充到 cache 中
3'b110	Hit Writeback	命中 cache，写回
3'b111	Fetch and lock	将寻址数据写到 cache 中，并锁住该行。

显然 op 的高 3 位和低 2 位结合，可以对不同的 cache 进行多种操作。详细 cache 的介绍和 cache 指令的描述请查阅《see mips run》的第 4 章，和 mips 手册卷 2 的 3.2 节指令介绍。

(3) 专用 Cache 初始化函数

所谓 Cache 初始化,就是将 Cache 里面的有效位清 0,数据位写 0 或写 1,使得 Cache 进入确定的状态。一般初始化,就是依次向所有的 Cache 行写全 0。而向 Cache 行写的动作可以通过 Cache 指令 **Index Store Tag** 完成。

Index Store Tag 指令是将 **offset(base)**的地址[12:12]位作为路号去选择某一路 cache,并将[11:5]位作为索引去寻址该路的 Cache 行,并将 **TagLo/TagHi** 的值写入到 Cache 行的 Tag 域。

TagLo/TagHi 为 cp0 寄存器,用于存放 Tag 的低位和高位。在我们提供的 CPU 核中,只用到 **TagLo** 即可,因为 **TagLo** 已经足够存放 cache 行的 Tag 域了。关于 **TagLo/TagHi** 的详细描述请查阅 MIPS 手册卷 3 的 9.46, 9.48 小节。

初始化 cache 时,需事先通过 mtc0 向 **TagLo** 写入 0, index 依据 **offset(base)**地址的[11:5]位指示,需每次加 1,当 index 从 0 加到 7'b111_1111 即完成了一路 cache 的所有行的初始化。类似可以完成另一路 cache 的初始化。

相应的,指令和数据 cache 都要如此分别初始化。此时 cache 指令的 op 域高 3 位代表 **Index Store Tag**,低两位则用于指示指令或数据 cache,故 cache 指令的 op 域就有两种情况了:初始化指令 Cache,op 为 0x08(5'b01000);初始化数据 Cache,op 为 0x09(5'b01001)。

比如,现在我们想初始化指令 cache 的第 0 路的第 0 个 cache 行,假定 base 寄存器为 a0,初值为(0x8000_0000),则先向 **TagLo** 写入 0,然后使用指令“cache 0x08, 0x0(a0)”即可;假设想初始化数据 cache 的第 1 路的第 1 个 cache 行,依然假定 a0 为 0x8000_0000,则使用指令“cache 0x09, 0x1000(a0)”即可。

(4) 通用 Cache 初始化函数

对于本次实验,我们明确给出了 CPU232 的 cache 结构,比如,路数、行数、行大小等。因而可以基于这些信息写出该专用的 cache 初始化代码,但如果想要自己所写的 cache 初始化代码具有通用性,则应该假定编程人员并不知道 CPU 里的 cache 的结构,随后去编写 cache 初始化代码。那如何让软件自己确认 CPU 里的 cache 结构呢?

MIPS CPU 里有一个 cp0 寄存器 **config1** 里面存有 cache 的信息,config1 寄存器结构如下:

31	30	25	24	22	21	19	18	16	15	13	12	10	9	7	6	5	4	3	2	1	0
M	MMU Size-1	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP							

MMU Size 为 CPU 里实现的 TLB 大小,从 1~64。在 TLB 初始化时需要用到。

IS 指示一级(L1)指令 cache 的每路的 cache 行的数目,计算公式为: $64 \times 2^{IS} (0 \leq IS \leq 6)$,当 **IS** 等于 7 时表示每路有 32 个 cache 行。

IL 指示一级(L1)指令 cache 的 cache 行的大小,计算公式为: $2 \times 2^{IL} (1 \leq IL \leq 6)$,当 **IL** 为 0 表示没有指令 cache,为 7 用于保留。

IA 指示一级(L1)指令 cache 的相连路数,计算公式为: $IA+1 (0 \leq IA \leq 7)$ 。

DS、DL、DA 用于指示一级(L1)数据 cache 的结构,计算公式同 **IS、IL、IA**。

关于 **config1** 寄存器的详细描述前参与 MIPS 手册卷 3 的 9.31 节。

软件可以通过 mtc0 读出 **config1** 的值,比如,“mtc0 a0, c0_config, 1”。因而我们可以通过移位、或运算等指令计算得到 CPU 里的 cache 结构,从而用于 cache 初始化,使得编写的 cache 初始化函数在 MIPS CPU 上具有通用性。

(5) Cache 初始化测试

当完成 cache 初始化后,我们可以 cache 指令 **Index Load Tag**(该指令显然与 **Index Store Tag** 作用相反)读出某一行 cache,读出的 Tag 值写入在 **TagLo/TagHi**,在 CPU232 中,只用到 **TagLo** 寄存器。

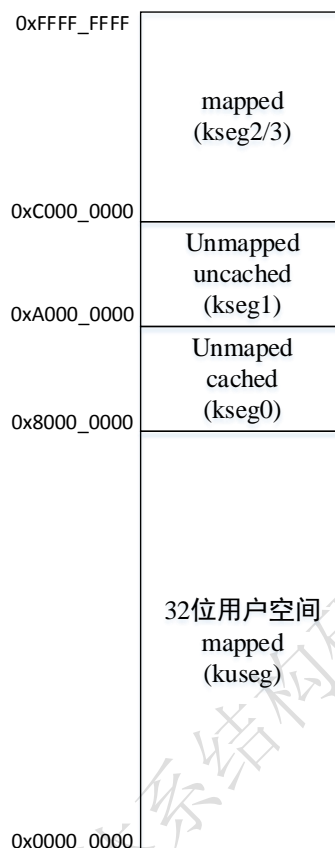
因而在对 Cache 初始化时,先将 **TagLo** 写 0,然后完成 cache 初始化后,将 **TagLo** 写一个非 0 的值。随后通过 **Index Load Tag** 指令读取任意 cache 行的 Tag 到 **TagLo**,然后查看 TagLo 是否为 0,非 0 则表示 cache 初始化失败。(注意 CPU232 中只有数据 Cache 有 **Index Load Tag** 指令,指令 Cache 没有 **Index Load Tag** 指令)

显然当我们初始时需要初始化所有 cache 行,但我们测试时,则没必要测试所有 cache 行,只用测试每路的第 0 个,最后 1 个,和中间任选一个的 cache 行即可。

1.7.5 TLB 初始化

(1) TLB 结构

MIPS 虚拟地址空间的内存映射的 32 位视图如下：



mapped 和 *unmapped* 指示是否需要通过 MMU 进行转换。**kseg0** 和 **kseg1** 是 *unmapped*，他们是永远指向物理地址 **0x0000_0000~0x2000_0000**。而 **kuseg** 和 **kseg2/3** 都是需要经过 MMU 单元进行地址翻译的(从虚拟地址转换为物理地址)。因为处理器刚启动时，MMU 是未设置好的，故我们无法对 **kuseg** 和 **kseg2/3** 进行地址翻译，这也是为什么 MIPS 处理器从 **0xbfc0_0000**(**kseg1** 段)启动。换言之，在我们为设置好 MMU 之前，不能使用地址 **kuseg** 和 **kseg2/3**。

MMU 单元有简单形式的，比如固定映射，**gs132** 即实现的该形式；也有复杂形式，比如基于 TLB 的 MMU，这也是大部分 CPU 都会支持的，**CPU232** 也实现了该形式。故需要对 TLB 进行初始化，才能使得程序正常访问 **kuseg** 和 **kseg2/3**，从而可以启动一个操作系统。

TLB 是基于页翻译的，即将一个虚拟页翻译为一个物理页，页内偏移不变，主要是地址高位的转换。对于 4KB 页，页内偏移占 12 位，故需要转换的为高 20 位。MIPS32 中 TLB 结构如下：

EntryHi		PageMask		EntryLo0		EntryLo1	
VPN2	ASID	PageMask	G	PFN0	C、D、V	PFN1	C、D、V
19bit	8bit	12bit	1bit	20bit	5bit	20bit	5bit

上表中第 1 列的 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 为 **cp0** 寄存器。

第 2 列为一项 TLB 的结构，左侧 **VPN2**(虚拟地址高位)、**ASID**(地址空间标识)和 **PageMask**(页大小屏蔽位)结合在一起用于与虚拟地址进行比较，如果命中，则说明该项 TLB 后指示的 **PFN0**(转换后的物理地址高位)和 **PFN1**(转换后的物理地址高位)即为需要对应的奇偶页的物理地址高位，C、D、V 为标志位。显然 MIPS32 中为一个 TLB 项一次映射奇偶两个页。具体信息请查阅 MIPS 手册卷 III。

第 3 列为 CPU232 中各域的大小，CPU232 中一页为 4KB，页内偏移占 12bit。一项 TLB 同时映射奇偶两个页，故 **VPN2** 为 19bit，而 **PFN0** 和 **PFN1** 为 20bit，与页内偏移结合后正好得到 32bit 的物理地址。也因为页大小为 4KB，故 **PageMask** 为全 0。

CPU232 中有 32 项 TLB，即有 32 项上表中的第 2 列。启动时，TLB 中这些域的值都是未知的，所谓初始化，就是将这些域填成确定的值。

关于 TLB 的详细描述，请参阅《计算机体系结构基础》的第五章和《see mips run》的第 6 章。

(2) TLB 相关 cp0 寄存器

从 TLB 结构中知道，TLB 相关 cp0 寄存器有 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1**，此外本次实验涉及到的还有 **Index**。

EntryHi 结构如下：

31	13 12	8 7	0
VPN2		0	ASID

PageMask 结构如下：

31	25 24	13 12	0
0	Mask	0	

EntryLo0 结构如下：

31 30 29	6 5	0
0	PFN	Flags

此处的 **PFN** 共有 24 位，是因为不同 CPU 中支持的物理地址不一定为 32 位，比如可能为 48 位。但在 CPU232 中就只用到 PFN 的低 20 位。其中 **Flags** 包含 C、D、V、G 等信息。

EntryLo1 结构同 **EntryLo0**。

Index 结构如下：

31	n	n-1	0
0			Index

Index 是用来索引 TLB 项的，故 TLB 有多少项，表中 **n** 有相应的值。在 CPU232 中有 32 项 TLB，故 **n** 为 5。

关于这些 cp0 寄存器的详细描述，请查阅 MIPS 手册卷 3 第 9 章。

(3) TLB 指令

MIPS 指令集中规定了 **TLBWR**、**TLBWI**、**TLBR**、**TLBP** 指令。其中 **TLBWR** 本次实验不用关注。

TLBWR 指令是以 cp0 寄存器 **Random** 为索引，将 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 写入到寻址到的 TLB 项中。显然初始化就是需要使用该指令。

TLBWI 指令是以 cp0 寄存器 **Index** 为索引，将 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 写入到寻址到的 TLB 项中。显然初始化就是需要使用该指令。

TLBR 指令则与 **TLBWI** 相反，是以 **Index** 为索引，读出寻址到的 TLB 项的值写入到 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1**。显然测试 TLB 是否初始化完成也可以使用该指令。

TLBP 指令则是使用 **EntryHi** 里的 **VPN2** 和 **ASID** 去查找整个 TLB，看是否有某一 TLB 项可以翻译该虚拟地址。如果查找到则将该项 TLB 索引号写入到 cp0 寄存器 **Index** 中。显然测试 TLB 是否初始化完成也可以使用该指令。

(4) TLB 初始化

从上一点知道，对 TLB 进行初始化需要使用 **TLBWI** 指令。该指令的用法就是“**TLBWI**”，然后它的默认操作数来自 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1**。为此在使用 **TLBWI** 之前，需要对 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1**、**Index** 赋初值。

EntryHi 中，我们需要赋值 **VPN2** 和 **ASID** 域，其中 **ASID** 域赋任意值都可，但 **VPN2** 建议赋值为 0x40000~0x5ffff 中的某一值，因为该段值对应虚拟内存映射中 **kseg0** 和 **kseg1** 的高 19 位地址，而 **kseg0** 和 **kseg1** 是不会进行 TLB 翻译的，因而访问该段地址也不会去查找 TLB。故在我们完成 TLB 初始化后，确保了无论软件程序访问内存地址，需要进行查找 TLB 进行地址翻译时，都不会命中。在不命中后，软件会自行重新装载 TLB，此时装载的 TLB 内容才是正确的。

PageMask 中，赋值全 0 即可。因为 CPU232 实现为 4KB 页大小。

EntryLo0 和 **EntryLo1** 的 **PFN** 域赋值为任意值均可，但 **flags** 域建议赋值为全 0。

Index 用于索引 TLB，由于 TLB 有 32 项，故 **Index** 赋初值 0，之后从 0 加到 31 即可。

一次 **TLBWI** 指令只能依据 **Index** 初始化一项 TLB。

CPU232 中有 32 项 TLB，故类似 cache 初始化的说明，可以写出专用的 TLB 初始化函数。当然也可以写出 MIPS 处理器通用的 TLB 初始化函数，这时需要软件自行读出 CPU 中 TLB 的项数。

(5) TLB 测试

在完成 TLB 初始化后，可以将 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 赋成别的值，随后使用 **TLBR** 指令读某一 TLB 项(由 **Index** 索引)，确认 **EntryHi**、**PageMask**、**EntryLo0**、**EntryLo1** 里的值是否和初始化 TLB 时写入的值相同。