PowerPC Figure – PPC 入门与优化

By Skywind（2007）
http://www.joynb.net/blog/

背景介绍

PowerPC 于 1991 年 IBM/MOTO/APPLE 研制，大量应用于服务器（AIX / AS400 系列及苹果系列服务器），家用游戏机（PS3, Wii, XBOX, GameCube），以及嵌入式（仅次于 Arm/x86 排第三）。PowerPC 核心在于开放系统软件标准，其应用范围仅次于 x86，是除去 x86 外最值得开发者了解的体系。

不需要写出非常高效的代码，但要了解基本效率原则；不需要大规模开发 PPC 程序，但需要时能写几段、调试时能看懂哪里错了。本文将从对比 x86 入手，引入 RISC 及 PowerPC 体系概念，向读者介绍该体系指令集，常用优化方法和交叉编译环境及模拟器的搭建等内容。

PowerPC 基础知识

1990 年 IBM 时任总裁 Kuehler 说服了摩托罗拉公司和苹果公司与 IBM 公司共同参与制订 PowerPC 体系结构。为了让 AS/400 也成为其中一员，1991 / 1992 年罗彻斯特实验室开始为 AS/400 扩充并制订 PowerPC 的 64 位结构。

----《罗彻斯特城堡》

大部分 CPU 指令集都可以分为：数据读写、数值计算、流程控制与设备管理四个部分，其中设备管理不属于介绍范围。开放系统软件标准在于硬件/软件只要符合该标准都能在 PowerPC 下运行，也就是说先今有大量 CPU 虽然实现不一，但是他们在标准上都支持了 PowerPC 体系，使得开发与接口更为方便。

PPC 使用 RISC（精简指令集），指令字长都是 32bit，一条 Intel 指令往往可以由多条 PPC 指令组合表示。Endian 一般都是可调的，默认使用 BE（Big Endian），同时 PPC 没有栈，也就是说应用程序需要自己实现相关操作。

常用术语介绍

POWERPC	字 长	简 称	X86
BYTE	8 BITS	B	BYTE
HALF WORD	16 BITS	H	WORD

WORD	32 BITS	W	DWORD
DWORD	64 BITS	D	QWORD

PowerPC 其他名称	X86 其他名称
Branch	JUMP
CR+SPR	EFLAGS

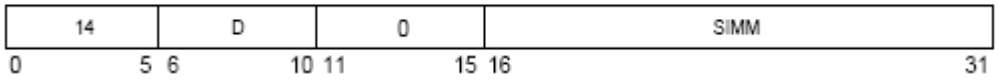
常用寄存器

类 别	说 明	补 充
通用寄存器 GPRs	R0 - R31 各 32bit PPC64 中是 64bit	相当于 EAX / EBX / ECX
条件寄存器 CR	CR 32bit 被分为 8 段 每段 4 位：LT,GT,EQ, SO	CR0-CR7 各 4 位组成 CR 相关运算需指明 CRn
连接寄存器 LR	用作记录跳转地址 32bit PPC64 中是 64bit	常用作记录子程序返回的地址 子程序调用指令在跳转前会改写
特殊寄存器 XER	记录溢出和进位标志	作为 CR 的补充
计数器 CTR	32bit (PPC64 中是 64bit)	相当于 ECX 用途
浮点寄存器 FPRs	FPR0 - FPR31 各 64bit	PPC32/PPC64 的浮点都是 64 位
浮点状态 FPSCR	浮点运算类型和异常等	同时可设置浮点异常捕获掩码

问题 1：如何加载 32 位立即数？

在 PPC 下如何加载 32 位的立即数呢？RISC 下 PPC 的每条指令都是 4 个字节定长。除去指令与寄存器参数编码，只有剩下 16bit 的长度用来描述立即数，比如立即数加载指令 LI：

LI rD, SIMM



立即数 SIMM 字段仅 16 位，如何表示 32 位？

答案：只有分两次载如，使用 LIS（立即数载入并左移）和 ADDI（立即数加法）分两次加载。因此 32bit 的立即数加载需要分两次完成：

```
LIS R3, 0x1122          加载并左移 16 位
ADDI R3, R3, 0x3344     再加上低 16 位
```

两条指令后，R3 完成对 0x11223344 的加载

特性：不一样的子程序调用

```
f1:         子程序入口
    blr     返回（跳转到 LR 地址）
start:
    bl f1   调用 f1（跳转并保存地址到 LR）
    li r1, 1    设置 r1 = 1
    li r3, 1    设置 r3 = 1
    sc      系统调用：结束程序
```

PPC 使用了 LR 寄存器（Link Register）来完成：在 bl 指令跳转前，下条指令（li r1,1）的地址会被保存到 LR 而执行到 f1 中的 blr 时，系统会跳转到 LR 所表示的地址，完成返回。

数据读写指令

说明	POWERPC	说 明	X86 参考

加载	LBZ R3, 0(R2)
	LHZ R3, 10(R2)
	LWZ R3, 32(R2)
		读字节 R3<-(R2)
		读半字 R3<-(R2+10)
		读单字 R3<-(R2+32)
			MOV AL,[EBX]
			MOV AX,[EBX+10]
			MOV EAX,[EBX+32]

保存	STB R3, 0(R2)
	STH R3, 10(R2)
	STW R3, 32(R2)
		写字节 R3->(R2)
		写半字 R3->(R2+10)
		写单字 R3->(R2+32)
			MOV [EBX], AL
			MOV [EBX+10], AX
			MOV [EBX+32], EAX

赋值	LIS R3, 0, 1122h ADDI R3, R3, 34h	R3 = 1122h << 16 R3 = R3 + 34h	MOV EAX,11220034h
转移	OR R3, R10, R10 (别名 MR R3, R10)	R3 = R10 or R10 R3 = R10	MOV EAX, ECX

注意：LBZ R3, 0(R2)与 LHZ R3,10(R2)并不全等同于 MOV AL,[EBX]和 MOV AX,[EBX+10]。前者将字节和半字加载到 R3 时顺便清空了高位，而后两条指令加载数据到 EAX 并不会清空高位。

第一个程序：Hello World !!

把下面的程序保存成 hello.s，并交叉编译：

```
# powerpc-eabi-as -gstabs hello.s -o hello.o
# powerpc-eabi-ld hello.o -o hello
```

```
.global _start                  /* 请将本程序保存成 hello.s */
.data                           /* 后面将讲解如何在虚拟机中调试 */
msg: .asciz "Hello, PowerPC World !!\n"
len = . - msg
.text                           /* 代码部分开始 */
_start:
li %r0, 4                       /* r0 = 4    */
li %r3, 1                       /* r3 = 1    */
```

```
lis %r4, msg@ha          /* r4 = msg(high) << 16 */
addi %r4, %r4, msg@l     /* r4 = r4 + msg(low)   */
li %r5, len              /* r5 = len */
sc                       /* system call (print) */
li %r0, 1                /* r0 = 1 */
li %r3, 1                /* r3 = 1 */
sc                       /* system call (exit) */
```

完成交叉编译后用 qemu 模拟器执行：
qemu-ppc hello
Hello, PowerPC World !!

关于如何在 x86 环境下交叉编译与调试，详细见第三部分的的“PowerPC 编译调试”。

特殊寄存器操作

类 别	指 令	说 明
连接寄存器	mflr rD	rD <- LR （等同于 mfspr rD, 8 ）
	mtlr rD	LR <- rD （等同于 mtspr 8, rD ）
时间寄存器	mftb rD, TBR	读取 PowerPC 内置的计时器到寄存器
条件寄存器	mfcr rD	rD <- CR
	mtcr rD	CR <- rD
CR 运算	crand crbD, crbA, crbB	crbD <- (crbA) AND (crbB)
	cror crbD, crbA, crbB	crbD <- (crbA) AND (crbB)
	creqv crbD, crbA, crbB	crbD <- (crbA) == (crbB)
状态寄存器	mfmsr rD	rD <- Machine State Register
	mtmsr rD	MSR <- rD

问题 2：没有栈仅靠 LR 如何递归？

```
f1:
    mflr r2              保存 LR 中记录的地址到 r2
    stw r2, -8(r1)       记录 r2 的数值到 MEM[r1-8]处
    addi r1, r1, -60     r1 后移 60 个字节，完成进栈操作
    ….
    addi r1, r1, 60      r1 前移 60 个字节，准备出栈
    lwz r2, -8(r1)       读出老的 LR 值到 r2
    mtfr r2              将 r2 的内容复制到 LR
    blr                  返回（跳转到 LR 地址）
start:
    ….
```

- bl f1 调用 f1（跳转并保存地址到 LR）
- ….

虽然 PPC 没有直接提供栈相关指令（PUSH/POP/CALL/RET），应用程序却常用 R1 来模拟栈指针，实现多层调用时对 LR 的记录与恢复。

数值计算指令

说明	POWERPC	说 明

基础	ADD R3, R4, R5
	SUBF R3, R4, R5
	MULLW R3, R4, R5
	DIVW R3, R4, R5
		R3 = R4 + R5
		R3 = (Not R4) + R5 + 1 = R5 – R4
		R3 = R4 x R5
		R3 = R4 / R5

标志	ADD. R3, R4, R5 SUBF. R3, R4, R5 MULLW. R3, R4, R5 DIVW. R3, R4, R5	注意指令下标“.”，代表影响条件寄存器 RC0：LT, GT, EQ, SO（小于，大于，等于，益出）的状态。
逻辑	AND R3, R4, R5 OR R3, R4, R5 NOR R3, R4, R5 XOR R3, R4, R5	R3 = R4 AND R5 R3 = R4 OR R5 R3 = Not (R4 OR R5) R3 = R4 XOR R5

特性：RISC 的“加载/存储”体系

RISC 决定了 PowerPC 使用加载/存储体系，即所有计算都是在寄存器中完成，而不是在主存中。除去加载/存储指令，所有操作都是针对寄存器的（少部分立即数），执行消耗周期相同且无须访问主存。

CISC 体系（如 x86）几乎所有操作都可对内存、寄存器或两者同时进行操作。传统上，处理器被设计成适应更加复杂的指令。

RISC 是基于“最简单的计算机指令是最经常被执行的”这一研究基础。用简单指令的组合来执行复杂的指令。这样处理器的时间安排能以较简单和快速运算为基础，能在给定时钟速度下执行较多指令。

现代的 CISC 处理器将自己的指令转换成了内部使用 RISC 格式，以实现更高的效率。

PowerPC 流程控制

IBM 公司 70 年代首次开发 RISC，但知道多年以后才应用到 IBM 公司的系统中。尽管第一款 IBM RISC 处理器早在 1986 年就被应用到 IBM PC-RT 中，但直道 IBM 1990 年推出 RS/6000 服务器时，该技术才开始受到重视。

-- 《罗彻斯特城堡》

条件寄存器

CR（Condition Register）一共 32 位，从低位到高位被分成 CR0-CR7 八段，每段四位。每个四位的 CRn 从低到高分别是：LT（小于标志）、GT（大于）、EQ（等于）和 SO（溢出）比较指令或条件跳转指令均可指明具体操作哪个 CRn，由此可以同时判断多个条件。整数计算默认更改 CR0，浮点数计算默认更改 CR1。

举例：求绝对值

```
_ABS:
    cmpwi %r3, 0        /* 参数 R3 与 0 比较  */
    bgt greater_than    /* 如大于就跳转  */
    neg %r3, %r3        /* 取负值：  R3 = Not(R3) + 1 */
greater_than:
    blr                 /* 返回 （从 LR 取出地址并跳转） */
_start:
    li %r3, 123         /* 加载立即数 123 */
    bl _ABS             /* 调用 _ABS （跳转前记录地址到 LR）*/
    li %r0, 1
    li %r3, 1
    sc                  /* 系统调用：结束程序 */
```

比较：cmpw rA, rB （比较有符号），cmpwi rA, IMM（立即数比较），cmpwl rA, rB（无符号）。
跳转：blt addr （小于跳转），bgt addr（大于跳转），beq addr（等于跳转）
类似：bne（不等），ble（小于等于），bge（大于等于）

数据比较指令

名 称	助记符	语法格式
有符号立即数比较	cmpi	crfD, L, rA, SIMM
有符号数比较	cmp	crfD, L, rA, rB
无符号立即数比较	cmpli	crfD, L, rA, UIMM
无符号数比较	cmpl	crfD, L, rA, rB

有符号立即数字比较	cmpwi	crfD, rA, SIMM (同 cmpi crfD, 0, rA, SIMM)
有符号数字比较	cmpw	crfD, rA, rB (同 cmp crfD, 0, rA, rB)
无符号立即数字比较	cmplwi	crfD, rA, UIMM (同 cmpli crfD, 0, rA, UIMM)
无符号数字比较	cmplw	crfD, rA, rB (同 cmpl crfD, 0, rA, rB)

特性：多条件寄存器

- 判断相同 - 老代码：

```
cmpw r3, r4
beq _branch_1
```

- 判断相同 - 新代码：

```
cmpw cr4, r3, r4
beq cr4, _branch_1
```

可以在比较和跳转命令第一个参数指明所使用的条件寄存器，如果不写的话，默认 CR0。
由此我们可以用更多条件寄存器同时判断若干条件，再用 cand/ cor/ cxor 复合运算。

数值比较 - 有符号

CMP crfD, L, rA, rB

```
a <- EXTS(rA)                  扩展符号到 a (如果无符号比较 cmpl 则直接 a = rA)
b <- EXTS(rB)                  扩展符号到 b (如果无符号比较 cmpl 则直接 b = rB)
If a < b then c = 0b100        设置小于标志
else if a > b then c = 0b010   设置大于标志
else c = 0b001                 设置等于标志
CR[4 * crfD  -  4 * crfD + 3] <- c || XER[SO]        记录 4 位状态
```

举例说明：

```
cmpw rA, rB      比较 rA, rB 的低 32 位结果存 cr0 (同 cmp 0, 0, rA, rB)
cmpd rA, rB      比较 rA, rB 的全 64 位结果存 cr0 (同 cmp 0, 1, rA, rB)
cmpwc r3, rA, rB 比较 rA, rB 的低 32 位结果存 cr3 (同 cmp 3, 0, rA, rB)
```

转移指令

指令 B（branch）是绝对地址无条件跳转，BA 是相对地址无条件跳转，BL 是跳转前将下一条指令的地址记录到 LR（可以用 blr 跳转到 LR 所指地址），BLA 是相对地址跳转，并将下条指令地址记录地址到 LR。

名 称	助记符	语法格式	解释
无条件转移	b (ba bl bla)	目标地址	子程序调用 bl/bla

有条件转移	bc (bca bcl bcla)	BO, BI, 目标地址	-
条件转移 bclrx	bclr (bclrl)	BO, BI (地址取 lr)	常用子程序返回
条件转移 bcctrx	bcctr (bcctrl)	BO, BI (地址取 ctr)	-
跳转到 LR 处	blr	无	同 bclr 0b10100,0

条件跳转中 BI 用来表示具体需要测试的条件寄存器 CR 的位，BO 用来表示测试方式，比如是测试大于/小于/等于还是测试计数器 CTR 的值，故此 blr 等同 bclr 0b10100, 0。

特性：指令的别名

PowerPC 指令助记符有大量别名：
比如 CMPW rA, rB 其实是 CMP 0, 0, rA, rB
比如 BEQ addr 其实是 BC 0xC, 2, addr
比如 BLR 其实是 BCLR 0x14, 0

转移指令如果没有指明条件寄存器，则默认使用 CR0（CR 的 0-3 位）；bca 相对于 bc 或者 ba 相对于 b，他们的指令码都相同，只是 AA 位（是否用绝对地址）为 1 ；bcl 相对于 bc 或者 bl 相对于 b，他们的指令码亦同，仅 LK 位（是否记录地址）为 1。

助记符	说 明	原 型
beq addr	如果等于则跳转 addr	bc 0b01100, 2, addr （CR0 的 EQ 位==1）
beq cr2, addr	如果 cr2 为等于则跳转 addr	bc 0b01100, 10, addr （CR2 的 EQ 位==1）
bne addr	如果不等于则跳转	bc 0b00100, 2, addr （CR0 的 EQ 位==1）
blt addr	如果小于则跳转	bc 0b01100, 0, addr （CR0 的 LT 位==1）
bgt addr	如果大于则跳转	bc 0b01100, 1, addr （CR0 的 GT 位==1）
ble addr	如果小于等于则跳转	bc 0b00100, 1, addr （CR0 的 GT 位==0）
bge addr	如果大于等于则跳转	bc 0b00100, 0, addr （CR0 的 LT 位==0）
blr	跳转到 LR 寄存器记录的地址	bclr 0b10100, 0, addr（无条件到 LR）

问题 3：如何跳转到 R3 所记录地址
BC, B 等都是用相对地址跳转的。如何实现类似 C 里面的函数指针调用？

答案：需要用到 LR 寄存器：

```
mtlr r3          将 R3 的值保存到 LR
blr              跳转到 LR 所指位置
```

条件转移原理（了解）

BC BO, BI, target_addr (AA=0, LK=0)

m <- 32
If BO[2]=0 then CTR <- CTR - 1 *如果 BO[2]==0 则计数器自减*

```
ctr_ok <- BO[2] | BO[3]                判断计数器条件
cond_ok <- BO[0] | (CR[BI] == BO[1])   判断条件寄存器某位是否符合需求
If ctr_ok & cond_ok then               如果两个条件同时成立则执行跳转
   if AA then NIA <- iea EXTS(BD || 0b00)   如果使用绝对地址
   else NIA <- iea CIA + EXTS(BD || 0b00)   如果使用相对地址
   if LK then LR <- iea CIA + 4             判断是否记录指令地址到LR
```

注：NIA - 新指令地址；CIA - 当前指令地址；EXTS - 扩展正负符号；AA - 是否使用绝对跳转的标志；LK - 是否用 LR 保存下条指令地址（CIA + 4）。

BO 字段常用操作码：

BO=00100　　如果条件成立（CR[BI]==0）则发生跳转
BO=01100　　如果条件不成立（CR[BI]==1）则发生跳转
BO=10100　　直接跳转

问题 4：求绝对值指令原理

下面代码请直接用 CMP/ BC 两条指令实现（提示：参考前面关于 BC/CMP 两条指令原理）

```
cmpw r3, r4
beq _branch_1
```

答案（了解即可）：

```
cmp 0, 0, r3, r4
bc 0b01100, 2, _branch_1
```

其实在实际开发中都是直接书写替代的别名

问题 5：PowerPC 与 x86 的编码区别

PPC 指令系统比 x86/arm 晦涩，同时 RISC 载入常数等指令等要分两次；PPC 大部分指令都是三操作数，而 x86 几乎都是双操作数；PPC 指令比 x86 更细致精准，同样程序 PPC 代码要比 x86 短。

示例：演示递归 – 求阶乘

接下来的程序将通过求阶乘演示递归。之前曾经说过：PPC 没有栈，故而实际递归时需要保存现场与返回地址的工作交给了应用程序，我们一般使用 R1 来模拟栈指针：

```
_factoria:                          /* 求阶乘，输入 R3，返回 R3 */
   mflr %r2
   stw %r2, -8(%r1)
   addi %r1, %r1, -60
_factoria.start:
   cmpwi %cr0, %r3, 1
```

```
    bgt _factoria.n1            /* branch to n1 if r3 > 1 */
    li %r3, 1                   /* return 1 (if r3 <= 1) */
    b _factoria.exit
_factoria.n1:
    stw %r3, 8(%r1)             /* save r3 to stack */
    addi %r3, %r3, -1           /* r3 = r3 - 1 */
    bl _factoria                /* call _factoria */
    lwz %r11, 8(%r1)            /* r11 = [r1 + 8] (old r3) */
    mullw %r3, %r3, %r11        /* r3 = r3 * r11 */
_factoria.exit:
    addi %r1, %r1, 60           /* restore stack point */
    lwz %r2, -8(%r1)            /* resotre LR */
    mtlr %r2
    blr
```

根据操作系统的不同，规定了不同的 ABI（应用程序二进制接口），详细定义了栈如何操作，参数如何传递等关键接口规范，开发时需注意查看。

PowerPC 编译调试

交叉编译（在一个平台下编译另一个平台运行的程序）需要一台 Unix 机器或者 Cygwin，下载并重新编译 binutils 即可：

```
# wget http://ftp.gnu.org/gnu/binutils/binutils-2.18.tar.bz2
# tar -jvxf binutils-2.18.tar.bz2
# cd binutils-2.18
# ./configure --target=powerpc-linux-eabi
# make all install
```

模拟器 QEMU 最好在 Linux 环境中使用（才能支持用户模式模拟）
apt-get install qemu （debian 直接安装）

其他平台需要手工编译。所谓用户模式在于不需要模拟整个 PPC 操作系统，而是模拟执行 PPC-Linux 下二进制可执行文件，PPC 程序的系统调用将会转化为本机 Linux 的系统调用。所以我们不需要再在 QEMU 下重新安装一个 Mac OS X 之类的系统：

```
# powerpc-linux-eabi-as -gstabs hello.s -o hello.o
# powerpc-linux-eabi-ld hello.o -o hello
# qemu-ppc ./hello
Hello, PowerPC World !!
#
```

上面是使用第一章中的 hello.s 进行编译，并在虚拟机中运行以后的效果。

PowerPC 指令优化

首先需要认识到 PPC 体系下的 CPU 种类繁多，对具体需要优化的环境需要详细了解。例如流水线的类型如何？以往习惯了 x86 的思维后，我们都以为主频越高越好，流水线越长越好。其实不然。越长的流水线，分支预测失误代价越大，单条指令通过的时间越长。因此如果单算执行一条指令的速度，流水线长 20 的 P4 2.0GHz 速度还没有流水线长 10 的赛扬 1.2GHz 快，而且 Intel 仅仅为了增加并行处理部分指令的机会而增加流水线长度，同时又要保持无法并行时的处理速度，为此只有增加主频，带来功耗的上升，以及分支预测失误的昂贵代价。

CPU 需要根据科学型还是商务型及多媒体型来采取不同的设计优化策略：比如科学型计算机多用小而密集的循环计算，因此普通的分支预测命中率高（90%以上），因为大部分跳转都是向上跳转的循环，而商务型却只有 50%的命中（大部分无规律的逻辑），多媒体型不但计算密集，而且内存吞吐量大。不同应用的 CPU 设计有所不同，优化也不同。

PowerPC 以 AS/400 为例，多为短流水线体系，分支预测失误的代价更少，且主频更低（功耗更小），采用更“聪明”的预测机制，大部分主频很低，但速度惊人。以上流水线设计的两派技术体系争斗了十多年，各有千秋，很多主频比 Intel 低很多的 PowerPC 的芯片，却表现出了更优越的性能，而市场上大部分人只喜欢盲目追求主频，这是一个误区。

1. 指令结对原则

在类 PPC405/440 的系统中，指令被分为下面三类的其中之一，类 405/440 系统能够在单一周期同时执行两条不属于同一种类的指令：

（1） 数据的加载与存储
（2） 任意义下处理：设置 CR 寄存器进行比较，分支，乘除 SPR 寄存器更新
（3） 其他种类操作：非 SPR/CR 寄存器更改，算术与逻辑

如果两条邻接的指令属于同一类别，那么第二条必须等待第一条处理完以后才能被调度，这样做就浪费了时钟周期；而如果邻接的两条指令属于上述不同类别且无结果依赖，那么两条指令能够被同时调度，这样做就能获得比较高的效率。

这与我们 x86 下优化的经验并部相同，在 x86 的流水线中只要无倚赖的代码基本都可以并行运行，比如我们可以并行处理若干无相关的加载或计算，从而在 x86 下达到较高的效率：

```
mov eax, [esi + 10]      三条无依赖加载能并行
mov ebx, [esi + 14]
mov ecx, [esi + 18]
add eax, edx             三条无依赖加法能并行
add ebx, edx
add ecx, edx
```

而这样在大多数类 405/440 的 PPC 下却是有问题的。整数计算属于同一类别，邻接的无依

赖计算指令不能如现代 x86 体系中得到同时运行；载入指令也相同，而整数计算和加载混合却能很好的并行调度：

```
lwz r3, 0(r10)          此加载和下一条加法无依赖，且属不同类别
add r4, r5, r5          加法能和前一条加载并行运行
lwz r6, 4(r10)
add r7, r8, r8
lwz r9, 8(r10)
add r3, r4, r4
```

因此如果我们的潜意识里过分熟悉 x86 优化方式，进而在用 C 开发的时候也会体现出来的话，可以说，这样的 C 代码在 PPC 下很难发挥效果的，即便编译器能优化，也需要给编译器留有优化的余地。

2. 加载依赖原则

当数据从缓存被加载到某寄存器的时候，需要数个周期以后数据才能被其他指令所使用，一条使用到刚被加载数据的指令需在数据被加载后第三个周期才能被调度。故在数据被加载与被使用两条指令之间的数个周期内形成了一段非常有效的优化区间，我们用来放置其他一些指令。加载与处理命令之间能放置的指令数决定于这些指令的种类，决定于他们是否能够结对并行处理，最大能有五条指令的优化空间。

指 令	周 期	说 明
lwz r4,0(r5)	n	加载指令被调度
addi r6,r6,0x20	n	ADDI 能和加载指令在同周期调度
or r7,r8,r9	n + 1	
stw r6,32(r1)	n + 1	
subf r10,r11,r12	n + 2	
srawi r8,r8,4	n + 2	
add r9,r4,r12	n + 3	R4 中的数据在加载命令的 3 周期后可以使用

在加载与使用命令之间能够并行插入的指令数取决于这些指令的混合方式，最少我们也可以插入两条指令进行优化。参考下面的指令，stw 和 lbz 两条处于 n+1 和 n+2 周期的指令不能被结对并行执行，因为他们属于同一类别的指令。

指 令	周 期	说 明
lwz r4,0(r5)	n	加载指令被调度
stw r6,32(r1)	n + 1	存储指令必须等待到 n+1 后才能被调度
lbz r11,20(r1)	n + 2	加载指令必须等待到 n+2 后才能被调度
lwz r9,0(r4)	n + 3	R4 中的数据在加载明了的 3 周期后可以使用

虽然大部分加载指令只有一个目标寄存器，但是需要注意一些带“更新”功能的加载指令，诸如 lwzu 它除了更新目标寄存器外，同时也会更新源寄存器，此时对源寄存器的使用也必须等待该指令被完成执行以后才行。

3. 指令依赖原则

同 x86 类似，有上下文依赖的指令不能同时被调度。在两条指令中，如果第一条指令更新的寄存器会被第二条指令用到，那么这两条指令不能被同时调度，利用这个特性我们将依赖关系的两条指令分开，并且插入至少一条指令完成优化。

比如我们在周期 n 用 add r4,r5,r6 更新了 R4 寄存器，那么就只能到周期 n+1 才能调度到使用 R4 寄存器的 srawi r7,r4,4 指令。在第一条指令的第 n 周期，没有指令能够与之并行执行而造成了浪费，所有正确的方式是在这两条指令之间加入一条无相关的指令，这样便能和 add 指令进行结对而得到同时调度，充分利用了时钟周期。

4. 缓存优化原则

缓存优化的方法基本和 x86 相同，这里再对缓存优化的原理做一下说明，即处理器要使用主存某地址的数据时，需要先将他们加载到缓存，然后才能处理，最后更新回主存去。根据前面的加载依赖原理的阐述，我们知道从缓存加载到寄存器需要三个时钟周期后才可以使用该寄存器，然而如果该地址的数据不在缓存中的话，前面就需要加上更多周期的等待周期，让数据先加载到缓存，最终再经过三个周期的等待后才能使用该数据。

为了降低直接从外存到缓存昂贵代价，现代的处理器都增加了一条预取指令，在 x86 下叫做 prefetch 而在 PowerPC 中叫做 DCBT（Data Cache Block Touch）：

DCBT rA, rB - 将(rA+rB)所表示的地址数据预取到缓存

该指令将提前告诉 CPU 将用到哪块内存，CPU 提前将该内存读入缓存，几个周期以后等到用时就该指令已经在缓存中了。用 dcbt 同 x86 的 prefetch 指令，现代 CPU 的主要瓶颈在主存到缓存之间，高效使用缓存是优化的关键。

下面是一段 x86 下比 memcpy 快 1.6 倍的内存拷贝代码，原因在于对缓存的使用上，先 mm0-mm7 顺序加载，再顺序写入，读入到 mm0 与从 mm0 写入中间间隔 7 条指令，让 CPU 有足够的时间加载，同时使用了预取。

```
loop:
    prefetchnta [esi + 256]      预取 esi + 256 地址的数据
    movq mm0, [esi + 0]          加载 esi + 0 到 mm0
    movq mm1, [esi + 8]
    …
    movq mm7, [esi + 56]
    movntq [edi + 0], mm0        写入 mm0 到 edi + 0
    movntq [edi + 8], mm1        使用穿透缓存方式写入
    …
    movntq [edi + 56], mm7
    add esi, 64                  指针后移 64 字节
```

```
    add edi, 64                 指针后移 64 字节
    dec ecx
    jnz loop                    判断计数器并循环跳转
```

而如果在 PowerPC 下写内存拷贝，我们就不能并列写若干加载指令，因为大部分 PPC 不能并行处理加载，我们需要将加载与存储交叉写：

```
loop:
    dcbt r12, r11               预先加载 (r12+r11) 处内存到缓存
    lwzu r3, 4(r11)             加载内存到 r3 并且移动指针
    lwzu r4, 4(r11)
    lwzu r5, 4(r11)             争取加载指令与写入指令并行运行
    stwu r3, 4(r10)             写入数据从 r3 并且移动指针
    lwzu r6, 4(r11)
    stwu r4, 4(r10)
    lwzu r7, 4(r11)
    stwu r3, 4(r10)             利用多寄存器的特点写下去
    lwzu r8, 4(r11)
    ….
    addi r9, 0, -1              减少计数器
    cmpwi cr4, r9, 0
    blt loop                    计数器未到就跳转
```

该段程序有三处优化，首先是缓存预取，dcbt 在每个循环预先取后面的内容，其次是充分利用 PPC 多寄存器的特点，最后是让加载和保存指令交叉进行充分的发挥并行作用。

如果你所使用的 PowerPC 没有 DCBT 指令的支持，那么我们可以用一些小技巧来达到缓存预取的效果，即将 DCBT 指令替换成一条 lwz 来加载该地址数据到一个无用的寄存器，这种方法称为“硬预取”，在 x86 中也能可以使用该方法来起到缓存预取的作用。

5. PPC 的 AltiVec ™ 指令优化：

在 PowerPC G4 后开始支持 AltiVec ™，这是一套类似 x86 下 MMX+SSE 的 SIMD 指令集，提供 128 位的矢量并行计算（8bit/16bit/32bit 三种计算元）的功能，使多媒体计算平均提高 4-5 倍，而具体的 AltiVec ™ 优化方法，超出本文叙述范围，读者可以自行查找相关资料。

6. 最终优化方法：

开启 C 编译器的汇编输出，在最大优化模式下思考编译器的优化策略。反复阅读对应 CPU 的官方文档，试验、试验、再试验！最终您能写出漂亮高效的 PPC 代码。

参考资料

《PowerPC860 嵌入式系统及应用》，机械工业出版社，陈晓竹，2006
《Linux PowerPC 详解-核心篇》，机械工业出版社，王齐, 1997
《罗彻斯特城堡》，机械工业出版社，IBM，2003
《基于 POWERPC 的嵌入式 LINUX》，北京航空航天大学出版社，漆昭铃, 2004

PowerPC Architecture Book，
http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html

Software optimization techniques for PowerPC 405 and 440，
http://www.ibm.com/developerworks/eserver/library/es-plib1app.html

Unrolling AltiVec Part 1 - Introducing PowerPC SIMD Unit
http://www.ibm.com/developerworks/library/pa-unrollav1/