龙芯 2E 体系结构概述

currentversion: 0.2

comcat <jiankemeng@gmail.com>

Sun Wah Hi-Tech System Software Ltd.

主要内容

内存访问方式

指令内部编码

过程调用约定

指令集概述

细节问题 (32, endian,aligned)

“体系结构”的界定

我们主要从软件设计者的角度来讨论，主要关注处理器所使用的指令集结构、寄存器约定以及高速缓存等。

指令集是处理器硬件与软件的**接口**。指令集定下来，软硬件之间的沟通桥梁就有了，软件用之操控硬件；硬件如何实现之，则有较大的自由度。

在处理器的设计中，指令集的设计比较困难，很难在短时间内做出一个成熟的指令集。目前工业上使用的处理器的指令集都是经过很长时间的积累才完善起来的。

龙芯的策略是站在前人积累的基础上往前走。（兼容 MIPS III）

内存访问方式

龙芯计算指令

先看一个简单的例子：

使用龙芯 (godson) 的指令完成运算： t0 = t1 + t2

```
add   t0, t1, t2        # 寄存器 t0 的内容为  t1 + t2
```

逻辑运算：

```
or   t0, t1, t2        # t0 = t1 | t2
```

可以看到龙芯的算术逻辑运算需要 3 个操作数，且只能是寄存器，不能直接作用于存储器，这一点很重要。

如果要操作的目标数据位于存储器中，则：

龙芯下访问内存数据

必须先使用访存指令 lw 将数据取到寄存器中，然后再操作

如：要完成 t0 = t1 + D[4] 的运算，则：

数组 D 位于存储器中，假设其首地址已经置于 t2，数组元素为 int 型，则：

```
lw      t3, 16(t2)      #t3 中的内容为 b[4]
add     t0, t1, t3      #t0 = t1 + t3
```

龙芯的体系结构下，访问存储器必须通过 lw (load word) 读取或者 用 sw (store word) 写入： sw t0, 0(t2)

x86 下访存方式

注意与 x86 的区别， x86 算术逻辑指令可以直接对位于内存中的数据进行操作，而不需要先将其取到寄存器中：（ebx 置数组首地址）

```
add     16(%ebx), %eax    #eax = eax + [ebx+16]
or      %eax, 4(%ebx)     #[ebx+4] |= eax
```

在此基础上， x86 尚有多种扩展寻址方式：

```
add   (%ebx,%ecx),%eax          #eax += [ebx+ecx]
add   (%ebx,%ecx,0x2),%eax      #eax += [ebx+ecx*2]
sub   -8(%ebx,%ecx,0x4),%eax    #eax -= [ebx+ecx*4-8]
```

龙芯是没有这些扩展的，龙芯的访存方式是很简单的。

A. 小结

x86 具有功能强大的寻址方式，一个操作数可以位于存储器中，通常称为 register/memory 系统结构。

龙芯呢，寻址方式简单，要求操作数总在寄存器中，这种精简的结构，通常可以称为 load/store 系统结构。

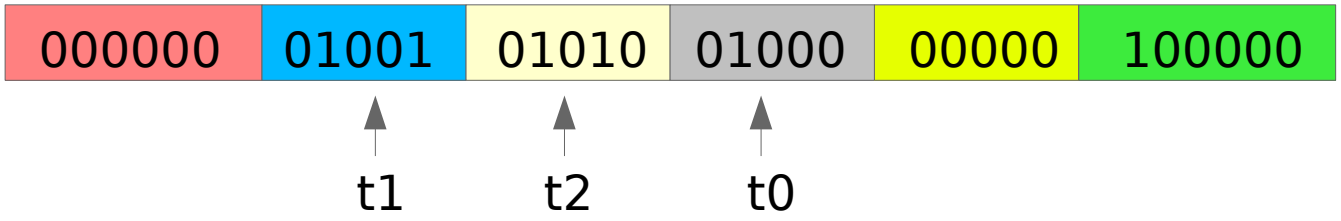
指令内部编码

指令的内部编码（机器码）

就以编码 add 指令为例，先看龙芯的。

龙芯下， add 就一种寻址方式，编码的话就一种形式：

```
add   t0, t1, t2   <⟷>   add   $8, $9, $10
```



处理器会根据第一个字段（操作码）与最有一个字段（函数码），确定其为加法运算，第二字段为加法操作的第一个源操作数寄存器号 (9=t1)，第三个字段表示加法操作的另一个源操作数寄存器号 (10=t2)，第四个字段中目的寄存器号 (8=t0)

指令的内部编码（机器码）

x86 下 add 的编码就要复杂的多了，因为它有多种寻址方式，每种寻址方式都有一种编码格式，至少如下的编码格式：

register1 to register2: 0000 000w : 11 reg1 reg2

register2 to register1: 0000 001w : 11 reg1 reg2

memory to register: 0000 001w : mod reg r/m

register to memory: 0000 000w : mod reg r/m

w:	1 位，用来表示操作是字节 (0) 还是双字 (1)
reg1,reg2:	3 位，所用寄存器的编码
mod:	2 位，用来选择内存的寻址模式
reg:	3 位，所用寄存器的编码
r/m:	3 位，与 mod 合用，决定寻址模式

指令的内部编码（机器码）

来看一个实际的例子，分别在龙芯和 x86 的平台上，用 gcc 把下面这个函数编译成机器码，看看他们的差别：

```
int add(int x, int y)
{
    return x + y;
}
```

指令的内部编码（机器码）

x86 下该函数的编码为：

```
08048354 <add>:
 8048354:   55                  push   %ebp
 8048355:   89 e5               mov    %esp,%ebp
 8048357:   8b 45 0c            mov    0xc(%ebp),%eax
 804835a:   03 45 08            add    0x8(%ebp),%eax
 804835d:   5d                  pop    %ebp
 804835e:   c3                  ret
```

可以看到， x86 下指令的机器码，长短不一，这个能有效减少指令所占用的存储空间。

x86 的指令长度为 1~17 个字节

指令的内部编码（机器码）

龙芯下该函数的编码为（未优化）：

```
00400650 <add>:
  400650:   27bdfff8        addiu   sp,sp,-8
  400654:   afbe0000        sw      s8,0(sp)
  400658:   03a0f021        move    s8,sp
  40065c:   afc40008        sw      a0,8(s8)
  400660:   afc5000c        sw      a1,12(s8)
  400664:   8fc30008        lw      v1,8(s8)
  400668:   8fc2000c        lw      v0,12(s8)
  40066c:   00000000        nop
  400670:   00621021        addu    v0,v1,v0
  400674:   03c0e821        move    sp,s8
  400678:   8fbe0000        lw      s8,0(sp)
  40067c:   27bd0008        addiu   sp,sp,8
  400680:   03e00008        jr      ra
  400684:   00000000        nop
```

龙芯则所有的指令长度一样，都是 32 位

B. 小结

x86 高效的编码方式，使得其能有效的减少代码长度，这个在早期存储空间受限时，意义重大。但高效往往带来复杂，一条 add 指令有十几种编码，这种多样性往往带来硬件实现上的困难。

龙芯呢，指令编码简单，所有整数指令只有 3 种格式，所有格式中，寄存器字段都在相同的位置，规则性很强，非常容易理解、记忆，硬件实现起来也比较容易。但极强的规则性往往要牺牲掉一些效率，所有指令的等长，就会浪费一些空间。

好的设计需要合适的折衷

过程调用约定

从一个例子开始

我们分别在 x86 和龙芯平台上编译以下代码：

```
int add(int x, int y)
{
   return x + y;
}

int main()
{
   add(2, 3);
}
```

x86 下

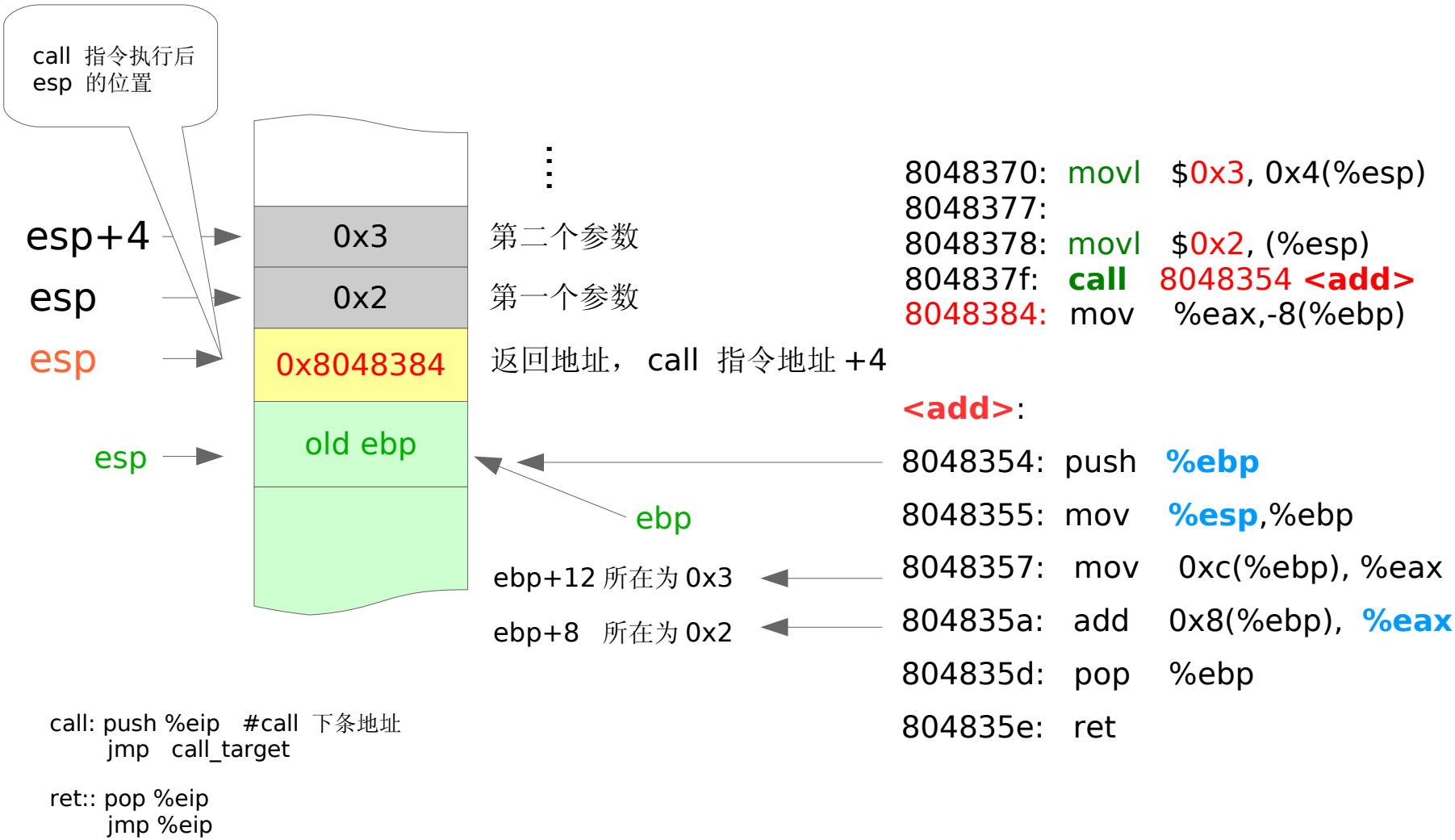
编译后，反汇编：

```
08048354 <add>:
 8048354:   55                      push   %ebp
 8048355:   89 e5                   mov    %esp,%ebp
 8048357:   8b 45 0c                mov    0xc(%ebp), %eax
 804835a:   03 45 08                add    0x8(%ebp),  %eax
 804835d:   5d                      pop    %ebp
 804835e:   c3                      ret

0804835f <main>:
 ......
 804836d:   83 ec 18                        sub    $0x18, %esp

 8048370:   c7 44 24 04 03 00 00            movl   $0x3, 0x4(%esp)
 8048377:   00
 8048378:   c7 04 24 02 00 00 00            movl   $0x2, (%esp)
 804837f:   e8 d0 ff ff ff                  call   8048354 <add>          #call add(1,2)
 8048384:   89 45 f8                        mov    %eax,0xfffffff8(%ebp)  #a = add(1,2)
 8048387:   83 c4 18                        add    $0x18,%esp
 .....
```

x86 下过程调用



call: push %eip #call 下条地址
jmp call_target

ret:: pop %eip
jmp %eip

过程调用一般过程

参数传递，要将参数放在子过程能够访问到的地方（x86 用栈传递）

将控制转移到子过程（x86 之 call 指令）

执行子过程

返回值传递，子过程将结果置于主过程能够访问到的地方（x86 将返回值置于 eax ）

将控制返回到初始位置（x86 之 ret 指令）

龙芯对过程调用的支持

寄存器支持：

a0~a3： 4 个参数寄存器，用来传递参数， 5 个参数的话，第 5 个用栈传递

v0~v1： 2 个返回值寄存器，用于返回结果值

ra： 1 个返回地址寄存器，用于返回到初始位置

指令支持：

龙芯使用 jal 来支持过程的控制转移， jal 全称为 jump and link，完成的操作为：在跳转到某个地址的同时，将下一条指令的地址保存于寄存器 ra 中。

控制返回的话，因为子过程可以通过 ra 获得返回的地址，则子过程在完成任务后，直接 jr ra，就完成返回了。其中 jr (jump register)，完成的操作为：跳转到寄存器所指示的地址处执行。

龙芯下整个过程

重新回到前面的 C 程序，编译后，反汇编：

```
00400670 <add>:
  400670:   03e00008    jr   ra                  # ra 置返回地址
  400674:   00851021    addu    v0,a0,a1         # v0 置返回值；该指令位于分支延迟槽

00400678 <main>:
  ...
  400684:   27bdffe0    addiu    sp,sp,-32       # 栈上分配 32 字节的空间
  400688:   afbf0018    sw   ra,24(sp)           # 保存 main 的返回地址于 [sp+24] 处
  40068c:   afbc0010    sw   gp,16(sp)           # 保存 gp 值于 [sp+16] 处
  400690:   8f99805c    lw   t9,-32676(gp)       # [gp - 32676] 处的值就是 400670
  400694:   24040002    li   a0,2
  400698:   0320f809    jalr    t9               # 将下条指令地址置入 ra，并跳转到 t9 处执行
  40069c:   24050003    li   a1,3
  ...
```

x86 寄存器使用约定

x86 共有 8 个 GPR： eax, ebx, ecx, edx, esi, edi, ebp, esp

体系结构层面上，对寄存器的使用有一些约定：

eax 常用来存放函数的返回值

ebp 用来做过程调用的帧指针

esp 用来做过程调用的栈指针（push,pop 操作时，会自动修改之）

字符串操作时 (movs, lods, stos)， esi 用来存放源地址、

edi 目的地址等等，这些操作也会自动修改 esi, edi 的值。

GCC 对龙芯寄存器的约定

龙芯总共有 32 个通用寄存器，编号为： 0~31

Register	Name	Usage
$0	zero	常量 0
$1	at	用作汇编器的暂时变量
$2-$3	**v0 - v1**	函数调用返回值（非浮点）
$4-$7	**a0 - a3**	函数调用时前 **4** 个参数（非浮点）
$8-$15	t0 - t7	暂时变量（子函数调用时无需保存和恢复）
$16-$23	s0 - s7	寄存器变量（子函数使用前须保存，返回之前须恢复）
$24-$25	t8 - t9	暂时变量（子函数调用时无需保存和恢复）
$26-$27	k0 - k1	通常被中断或异常处理程序使用，用于保存一些系统参数
$28	gp	全局指针;一些系统维护这个指针来更方便的存取 static 和 extern 变量
$29	**sp**	堆栈指针
$30	**fp**	一般用来做帧指针
$31	**ra**	返回地址

注：gcc 中，用 t9 保存子过程起始地址

小结

龙芯使用 4 个专用的寄存器传递参数，第 4 个以后的参数使用栈传递，对大多数过程而言超过 4 个参数的很少，因为访问内存的代价要远远的高于访问寄存器，这个在一定程度上提高了效率。

x86 因为 GPR 比较少，用专用的寄存器来传递参数，对其而言是太奢侈了。

指令集概述

x86 指令集

x86 指令集包括整数指令、浮点指令、多媒体指令等。其中整数指令可以分为 4 类：数据传送指令，算术和逻辑运算指令，控制流指令，字符串操作指令。

数据传送： mov, push, pop

算术和逻辑运算： add, sub, inc, dec, and, or, shl, shr, cmp

控制流： jmp, jz, jnz, call, ret

字符串操作： movs, lods, stos

龙芯指令集概要

龙芯 2E 的指令集由 MIPS III 加上自定义的指令组成。可以分为：访存指令、计算指令、跳转和分支指令以及协处理器指令（用于操控协处理器，完成内存管理、异常处理、浮点运算、多媒体运算等）。取每类典型指令，陈述如下：

访存指令： lb, sb, lh, sh, lw, sw, ld, sd (byte, half-word, word, double-word)

计算指令： add, sub, mult, div, and, or, sll, srl, slt...

跳转分支指令： j, jr, jal, beq, bne...

协处理器指令： tlbwr, cache, mtc0, mfc1...

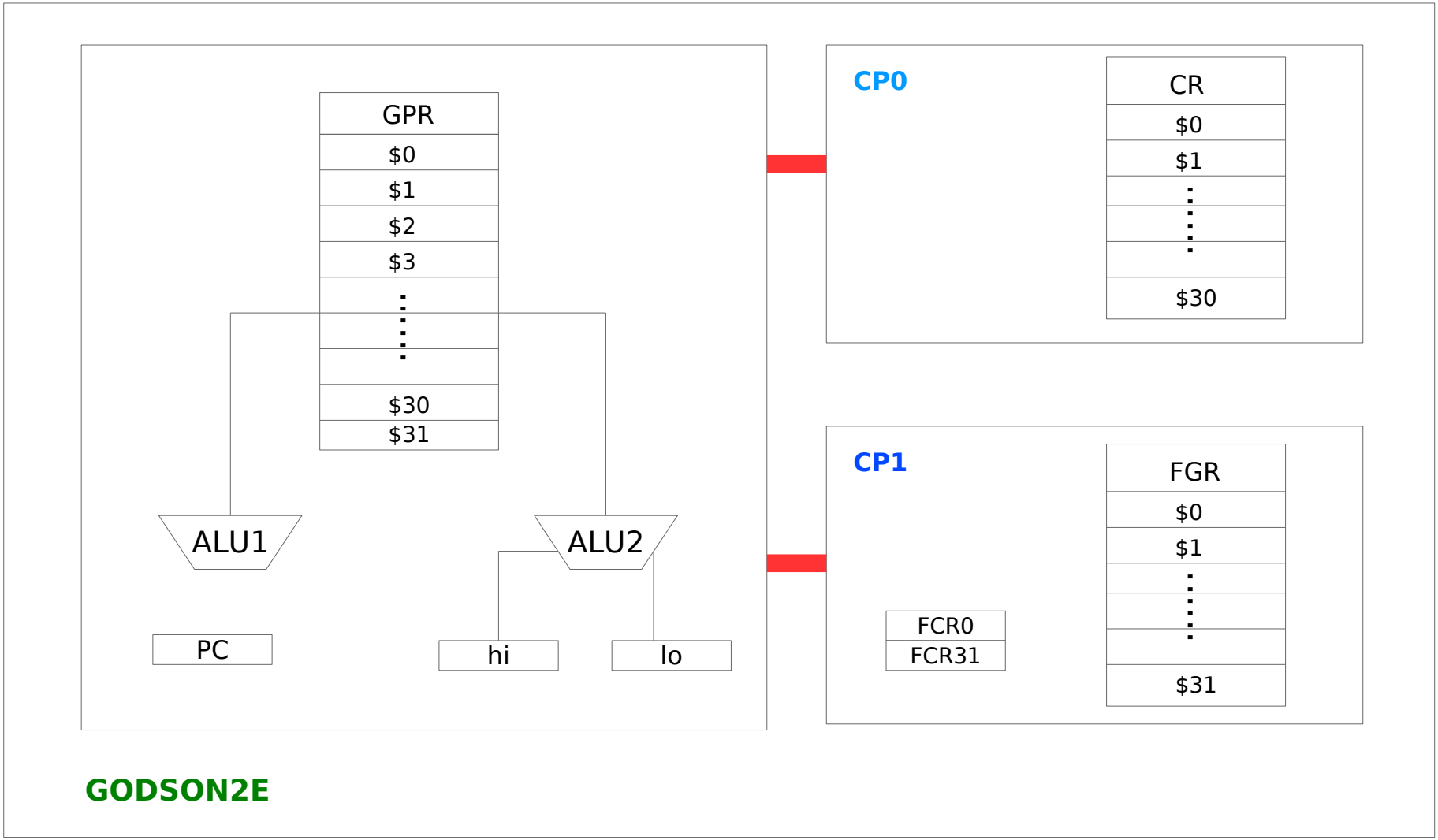
龙芯协处理器指令

龙芯协处理器指令完成协处理器内部的操作。龙芯 2E 处理器内部有两个协处理器，分别为 0 号协处理器 (CP0) 和 1 号协处理器 (CP1) 。

CP0 是系统协处理器，主要用于管理内存和处理异常。作用于其上的指令有：用于和通用寄存器交换数据的 mfc0, mtc0 ，用于 TLB 管理的 tlbwr, tlbr ，用于高速缓存管理的 cache 以及用于异常返回的 eret 等。其有自己的专用寄存器（ CR，控制寄存器）。

CP1 是龙芯的浮点协处理器，用来执行龙芯的浮点指令、自定义的多媒体指令和龙芯扩展的定点指令。这些指令都在浮点寄存器 (FGR) 上操作。如果让 CP1 中的浮点寄存器 (FGR) 与通用寄存器 (GPR) 交换数据也要专用的数据交换指令： mfc1, mtc1 。 CP1 亦有 2 个控制寄存器 (FCR) ，分别为 FCR0, FCR31 ，需使用 cfc1, ctc1 与通用寄存器 (GPR) 交换数据

程序员眼中的龙芯


GPR
$0
$1
$2
$3
$30
$31
ALU1
ALU2
PC
hi
lo
CP0
CR
$0
$1
$30
CP1
FGR
$0
$1
$31
FCR0
FCR31
GODSON2E

龙芯指令集特点

指令总数较 x86 要少

没有 push， pop 等栈操作指令

没有 movs, lods 等字符串操作指令

对过程调用的支持方面，没有 ret 指令，jal/jalr 对应于 call

总而言之，龙芯的指令集较 x86 精简的多，便于理解、记忆，是个相对容易实现的指令集。

总结

从前面的分析，可以看到龙芯具有简单的寻址模式、精简的指令集、固定长度的指令编码、大量的通用寄存器是个典型的RISC （Reduced Instruction Set Computer）系统结构。这种结构的原则是：通过指令流水线获得高性能，易于用硬件实现，与高度优化的编译器的集合提高性能。

相比较而言，x86 是个典型的 CISC （Complex Instruction Set Computer）结构，特点也很明显：功能强大的寻址模式和指令、高效的指令编码、较少的寄存器。其设计原则为：简化编译器，减少执行代码的长度。

细节问题

总述

龙芯处理器主要包括三个系列。龙芯 1 号系列主要面向嵌入式应用，龙芯 2 号超标量处理器系列主要面向桌面应用，龙芯 3 号多核处理器系列主要面向服务器和高性能机应用。

龙芯 2E 处理器是 64 位、兼容 MIPS III 指令集的通用 RISC 处理器。

龙芯的商品名称为：LOONSON

龙芯的内部开发代号为：Godson

32 位兼容模式

龙芯 2E 为 64 位处理器，可以以 64 位为单位，对信息进行处理。其 32 个通用寄存器，都是 64 位的。

龙芯 2E 亦可运行于 32 位模式下，处理 32 位数据时，只要将编号为 63~32 位（31~0 置 32 位数据）的高位符号扩展即可：

`0x70b0 55aa` 进入寄存器则扩展为： `0x0000 0000 70b0 55aa`

`0x80b0 55aa` 最高位为 1，则扩展为： `0xffff ffff 80b0 55aa`

注意用 lbu, lhu, lwu 读取无符号整数时，因为并没有符号位，所以高 32 位是扩展为 0 的

处理器字节顺序

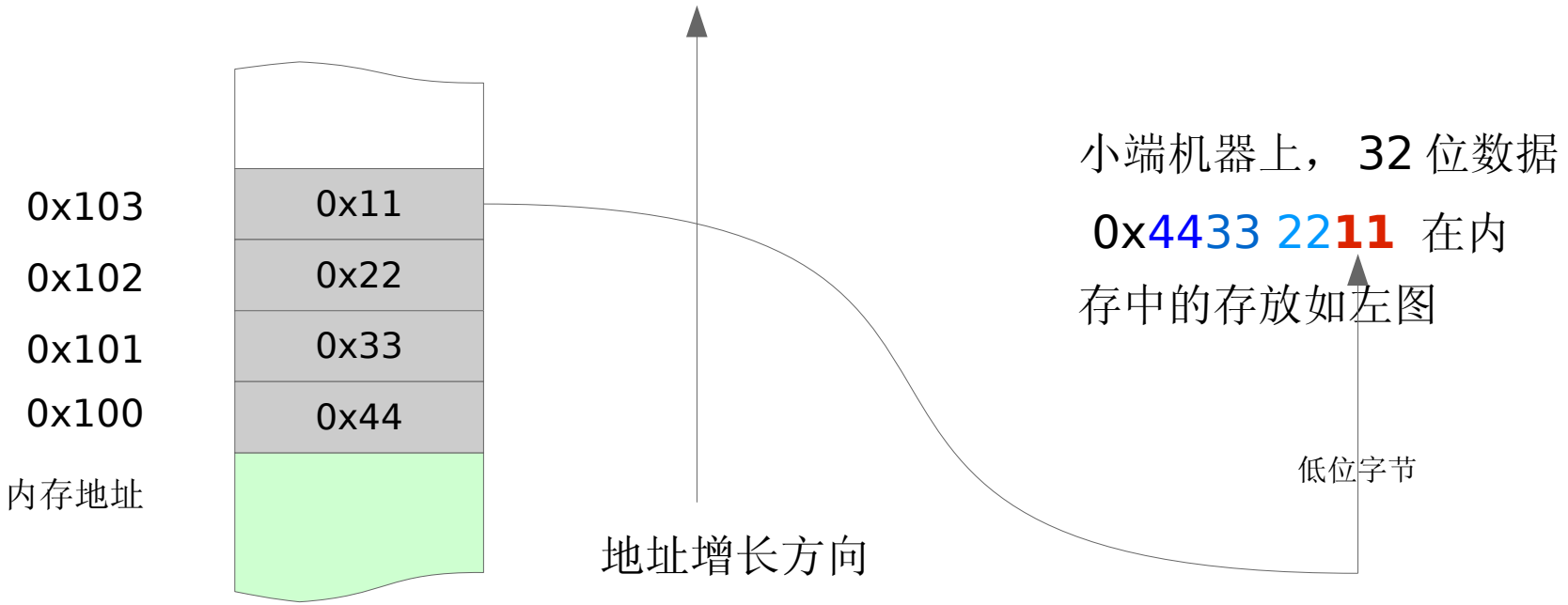
处理器需要对数据单元进行编号，因此最小编号的字节是位于最左或最右的字节，习惯称为字节顺序。

通常人类阅读、书写规则为：左高右低。如我们程序中写为 0x4433 2211，则最低位在最右边，第一个字节为 0x11，往左依次递增。CPU 的内部寄存器亦可理解为以这种格式存放数据。

但一个数据单元在内存中的存放方式却有两种模式。

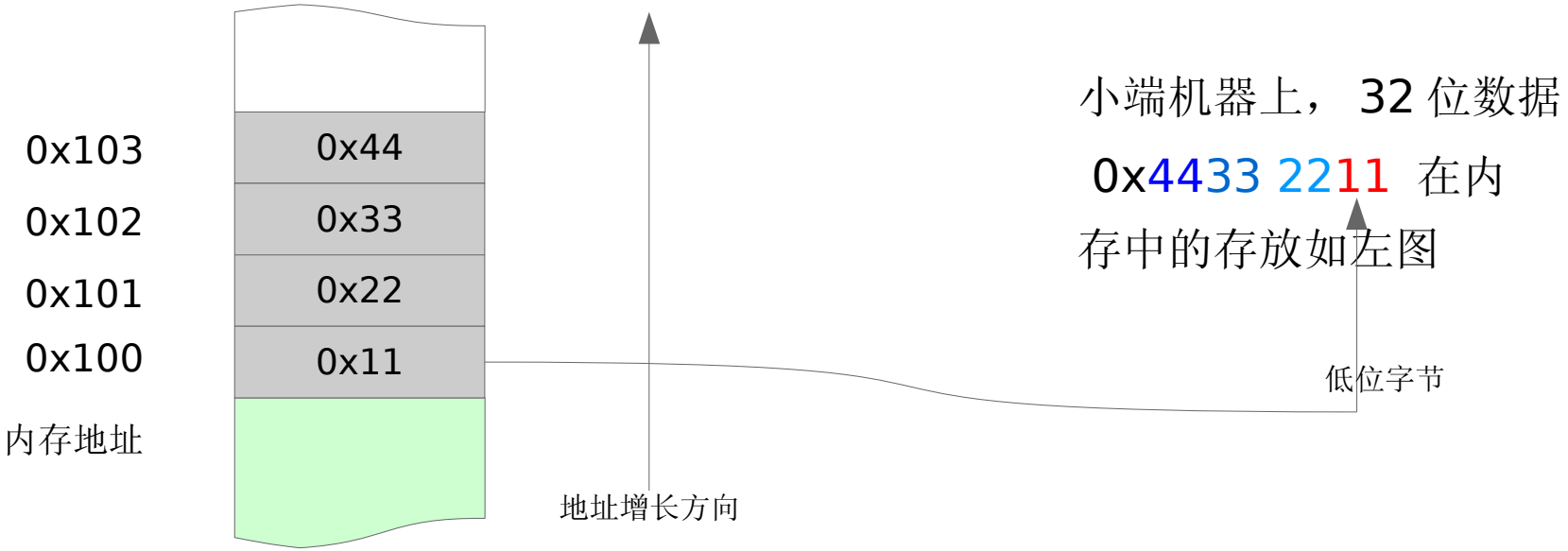
大端模式

大端模式 (Big endian)：低位字节位于存储器的高地址处的存放方式



小端模式

小端模式 (Little endian)：低位字节位于存储器的低地址处的存放方式。



龙芯字节顺序

多数 MIPS 兼容的处理器 2 种模式都支持，龙芯只实现了小端模式。

字节顺序主要在处理器与存储器交换数据时，表现突出。看一下龙芯的实际过程：

龙芯的访存指令有：以字节为单位的 lb， sb；以半字为单位的 lh (load halfword)， sh (store halfword)；以字为单位 (32bit) 的 lw， sw；以及以双字为单位的 ld， sd；基本覆盖了 C 语言的基本数据类型。

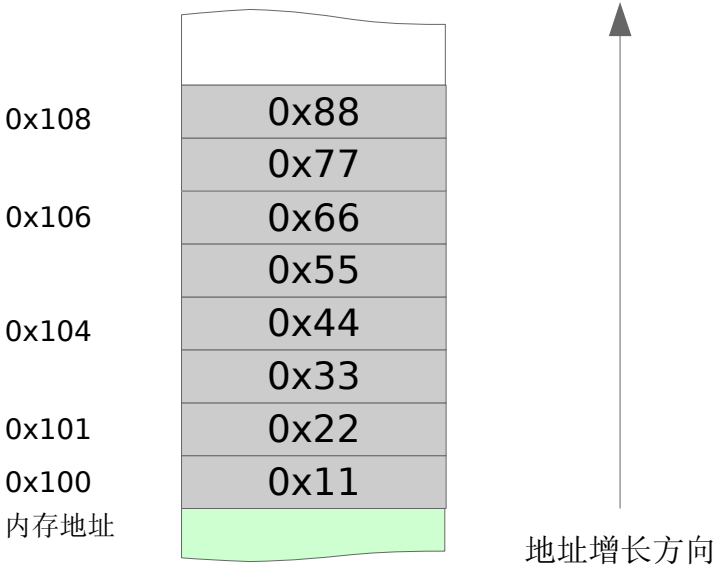
```
t0  值为 0x100

lh  t1, 0(t0)      #t1=0x2211
lw  t2, 0(t0)      #t2=0x44332211
ld  t3, 0(t0)      #t3=0x8877665544332211
```

如果龙芯是大端机的话：

```
lh  t1, 0(t0)      t1 则为 0x1122
lw  t2, 0(t0)      t2 则为 0x11223344
```

sh, sw, sd 则为逆向操作。



非对其访问

龙芯下使用访存指令读取或写入数据单元时，目标地址必须是数据单元字节数的整数倍，这个叫做地址对齐。

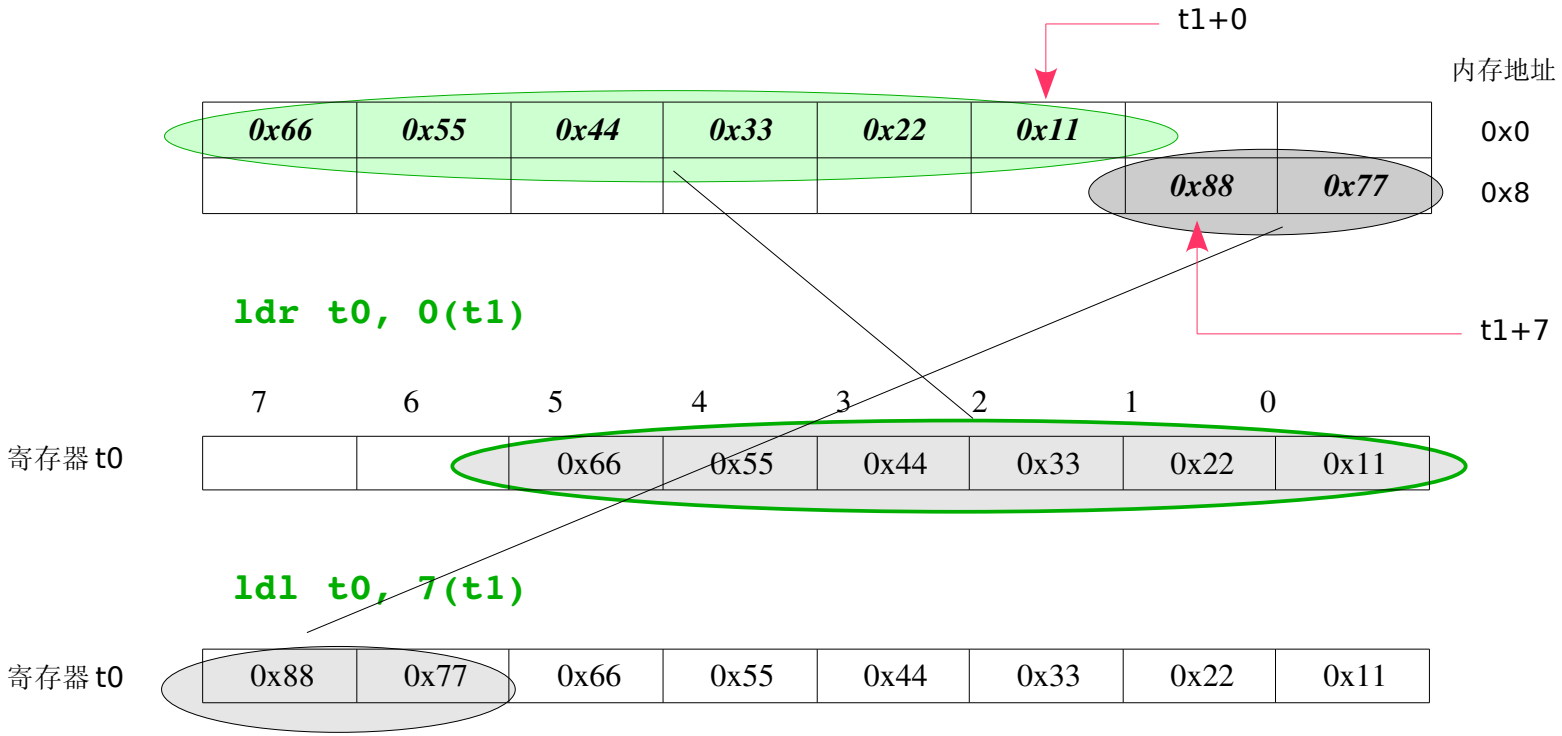
如 lh 读取一个半字时，存储器的地址必须是 2 的整数倍； lw 读取一个字时，存储器的地址必须是 4 的整数倍； sd 写入一个双字时，存储器的地址必须是 8 的整数倍。

x86 能够处理非对齐的访问。而类似龙芯的 RISC CPU 在访问一个非对齐的地址会导致 CPU 陷入异常，结果一般为：系统提示“非法指令”或者“总线错误”。

高级语言中一般不会遇到这种问题，编译器常常会处理好数据类型的对齐。

龙芯下非对其访问的解决

龙芯下可以直接使用宏指令 ulh, ulw, uld 对非对齐地址进行访问。 ulh 会由汇编器展开成 2 个 lb 、一个移位和一个或操作。而 ulw/uld 则由汇编器展开为一条 lwr/ldr (load right) 和一条 lwl/ldl(load left) 。 uld t0, 0(t1) 原理如下：



谢谢！