

二、MIPS体系结构

○ 指令集体系结构类型：寄存器——寄存器型

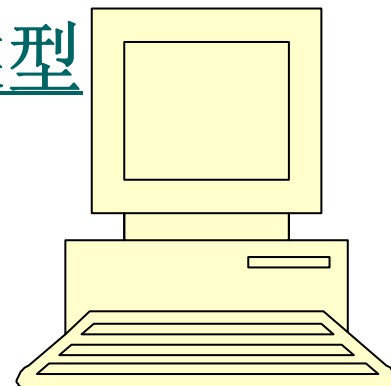
(1) 寄存器的特点

(2) 整数乘法单元和寄存器

(3) 寻址方式

(4) 存储器和寄存器中的数据类型

(5) 流水线冒险



二、MIPS体系结构——（1）寄存器

- MIPS 包含32个通用寄存器
- 硬件没有强制性的指定寄存器使用规则，但是在实际使用中，这些寄存器的用法都遵循一系列约定
- 寄存器约定用法引入了一系列的寄存器约定名。在使用寄存器的时候，要尽量用这些约定名或助记符，而不直接引用寄存器编号

| 31 | 16 | 15 | 0 |
|-------|----|----|---|
| \$ 0 | | | |
| \$ 1 | | | |
| \$ 2 | | | |
| \$ 3 | | | |
| \$ 4 | | | |
| | | | |
| \$ 27 | | | |
| \$ 28 | | | |
| \$ 29 | | | |
| \$ 30 | | | |
| \$ 31 | | | |

| | |
|--|--|
| | |
|--|--|

 hi

| | |
|--|--|
| | |
|--|--|

 lo

| 编号 | 助记符 | 用法 |
|-------|---------------|--|
| 0 | <u>zero</u> | 永远为0 |
| 1 | <u>at</u> | 用做汇编器的暂时变量 |
| 2-3 | <u>v0, v1</u> | 子函数调用返回结果 |
| 4-7 | <u>a0-a3</u> | 子函数调用的参数 |
| 8-15 | <u>t0-t7</u> | 暂时变量，子函数使用时不需要保存与恢复 |
| 24-25 | <u>t8-t9</u> | |
| 16-23 | <u>s0-s7</u> | 子函数寄存器变量。在返回之前子函数必须保存和恢复使用过的变量，从而调用函数知道这些寄存器的值没有变化 |
| 26,27 | <u>k0,k1</u> | 通常被中断或异常处理程序使用作为保存一些系统参数 |
| 28 | <u>gp</u> | 全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量 |
| 29 | <u>sp</u> | 堆栈指针 |
| 30 | <u>s8/fp</u> | 第9个寄存器变量。/ 框架指针 |
| 31 | <u>ra</u> | 子函数的返回地址 |

MIPS 通用寄存器的使用

○ 两个特殊寄存器:

➤ **\$0**: 不管你存放什么值, 其返回值永远是零。

```
ori $s6,$0,0x1000
```

```
beq $0,$0, fact
```

➤ **\$31**: 永远存放着正常函数调用指令(jal)的返回地

主程序:

...

```
addi $a0, $0, 3
```

```
jal A
```

```
move $s1, $a0 #伪指令
```

...

过程**A**:

...

...

```
jr $ra
```


MIPS 通用寄存器的使用

- **\$ at:** 这个寄存器由编译器生成的复合指令使用。
 - 当必须明确地使用它(如存入数据或从异常处理中恢复寄存器值)时，有一个汇编directive可以阻止编译器隐式地使用它(这样会有一些汇编宏指令不能用)。

MIPS 通用寄存器的使用

- **\$ v0, \$ v1:** 用来存放一个子程序(函数)的非浮点运算的结果或返回值。如果这两个寄存器不够存放需要返回的值，编译器将会通过内存来完成。

```
...  
    li  $ v0, 1           #返回1  
    j   L1                #跳转到代码并返回  
...  
L1:                          #结果保存到寄存器$ v0  
    sw  $ ra ,20 ($ sp)    #保存寄存器$ ra  
    sw  $fp , 16 ($ sp)    #保存寄存器$ fp  
...
```


MIPS 通用寄存器的使用

- **\$ a0-a3**: 用来传递子函数调用时前4个非浮点参数。

sw \$a0 , 0 (\$ fp) #保存参数

lw \$ v0 , 0 (\$ fp) #装入a0

MIPS 通用寄存器的使用

\$ t0-t9: 依照约定，一个子函数可以不用保存并随便的使用这些寄存器。

- 在作表达式计算时，这些寄存器是非常好的暂时变量。
- 注意：当调用一个子函数时，这些寄存器中的值有可能被子函数破坏掉。

MIPS 通用寄存器的使用

- **\$ s0-s8:** 依照约定
 - 子函数必须保证当函数返回时这些寄存器的内容必须恢复到函数调用以前的值,
 - 或者在子函数里不用这些寄存器或把它们保存在堆栈上并在函数退出时恢复。
- 这种约定使得这些寄存器非常适合作为寄存器变量、或存放一些在函数调用期间必须保存的原来的值。

move \$s1,\$0 (add \$s1, \$0, \$0)

move \$s2,\$s1 (add \$s2, \$0, \$s1)

addiu \$s3,\$0,1

lw \$s4,0(\$a0)

MIPS 通用寄存器的使用

- **\$ k0, k1:** 被OS的异常或中断处理程序使用。被使用后将不会恢复原来的值。因此它们很少在别的地方被使用。

MIPS 通用寄存器的使用

- **\$ gp:**

- 如果存在一个全局指针，它将指向运行时决定的静态数据(static data)区域的一个位置。这意味着，利用gp作基指针，在gp指针32K左右的数据存取，系统只需要一条指令就可完成。

| Op | Rs | Rt | Address |
|----|----|----|---------|
|----|----|----|---------|

6位

5位

5位

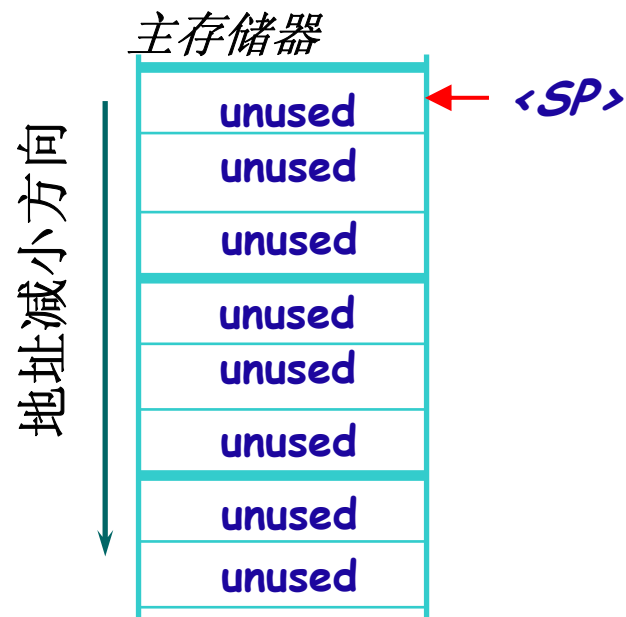
16位

MIPS 通用寄存器的使用

- 如果没有全局指针，存取一个静态数据区域的值需要两条指令：
 - 一条是获取有编译器和loader决定好的32位的地址常量。
 - 另外一条是对数据的真正存取。
- 为了使用**\$ gp**，编译器在编译时刻必须知道一个数据是否在**\$ gp**的64K范围之内。
- 并不是所有的编译和运行系统支持gp的使用。

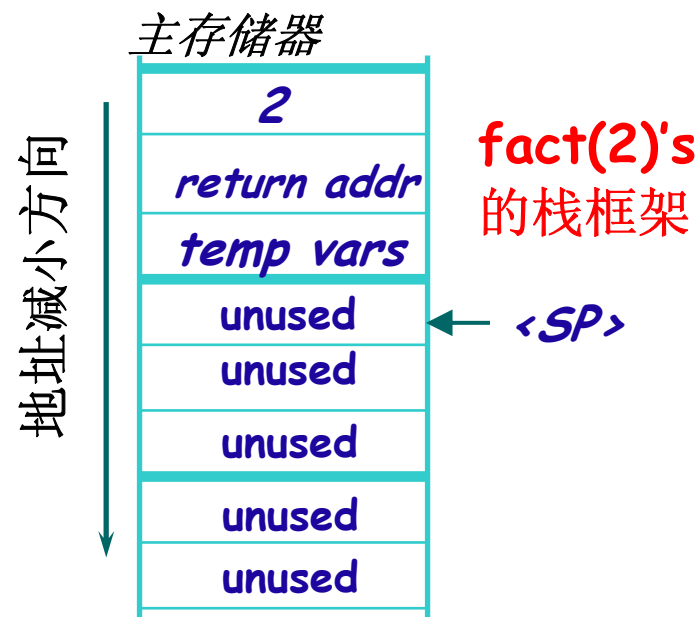
MIPS 通用寄存器的使用

- **\$ sp**: 堆栈指针的上下需要显式的通过指令来实现。因此MIPS通常只在子函数进入和退出的时刻才调整堆栈的指针。这通过被调用的子函数来实现。
- **SP**通常被调整到这个被调用的子函数需要的堆栈的最低的地方，从而编译器可以通过相对於sp的偏移量来存取堆栈上的堆栈变量。



MIPS 通用寄存器的使用

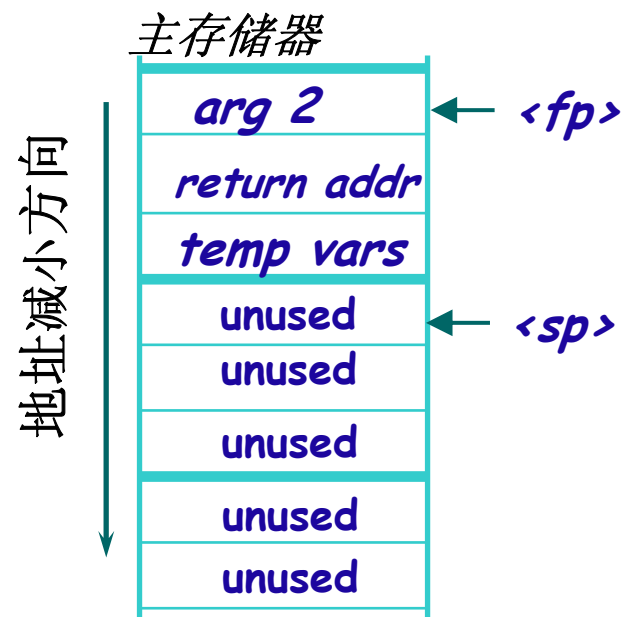
- **\$ fp**: 另外的约定名是s8。
- **fp**作为框架指针可以被过程用来记录堆栈的情况，在一个过程中变量相对于框架指针的偏移量是不变的。一些编程语言显示的支持这一点。
- 汇编程序员经常会利用fp的这个用法。C语言的库函数 `alloca()` 就是利用了fp来动态调整堆栈的。



MIPS 通用寄存器的使用

注意：如果堆栈的底部在编译时刻不能被决定，你就不能通过**\$ sp**来存取堆栈变量，因此**\$ fp**被初始化为一个相对于该函数堆栈的一个常量的位置。

这种用法对其他函数可以是不可见的。



MIPS 通用寄存器的使用

- **\$ ra**: 当调用任何一个子函数时，返回地址存放在ra寄存器中，因此通常一个子程序的最后一个指令是：

jr ra.

- 子函数如果还要调用其他的子函数，必须保存ra的值，通常通过堆栈。

```
.text
main:
subu   $sp , $sp,32
      #堆栈帧有32字节长
sw     $ra , 20 ($ sp )
      #保存返回地址
sw     $fp, 16 ($ sp )
      #保存旧帧指针
addu   $fp, $ sp, 28
      #设置帧指针
```


MIPS 通用寄存器的使用

- **\$31**: 永远存放着正常函数调用指令(jal)的返回地址。
 - 请注意call-by-registe的jalr指令可以使用任何寄存器来存放其返回地址。
 - 当然，如不用\$31，看起来程序会有点古怪。
- 其他方面，所有的寄存器都是一样的。可以被用在任何一个指令中

MIPS 通用寄存器的使用

- **MIPS**里没有状态码。CPU状态寄存器或内部都不包含任何用户程序计算的结果状态信息。
- **hi**和**lo**是与乘法运算器相关的两个寄存器大小的用来存放结果的地方。
 - 它们并不是通用寄存器，除了用在乘除法之外，也不能有做其他用途。
 - MIPS里定义了一些指令可以往hi和lo里存入任何值。

MIPS 通用寄存器的使用

- 浮点运算协处理器(浮点加速器, FPA), 如果存在的话, 有32个浮点寄存器。按汇编语言的简单约定讲, 是从**\$f0**到**\$f31**。
- 实际上, 对于MIPS I和MIPS II的机器, 只有16个偶数号的寄存器可以用来做数学计算。当然, 它们可以既用来做单精度(32位)和双精度(64位)。当你做一个双精度的运算时, 寄存器\$f1存放\$f0的余数。奇数号的寄存器只用来作为寄存器与FPA之间的数据传送。
- MIPS III CPU有32个FP寄存器。但是为了保持软件与过去的兼容性, 最好不要用奇数号的寄存器。

X86寄存器结构

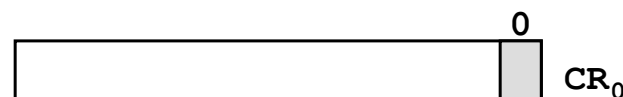
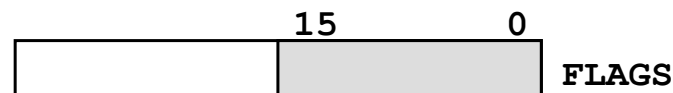
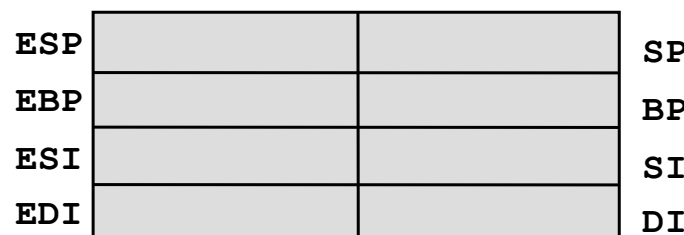
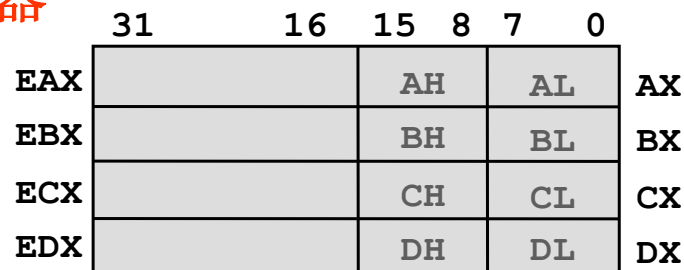
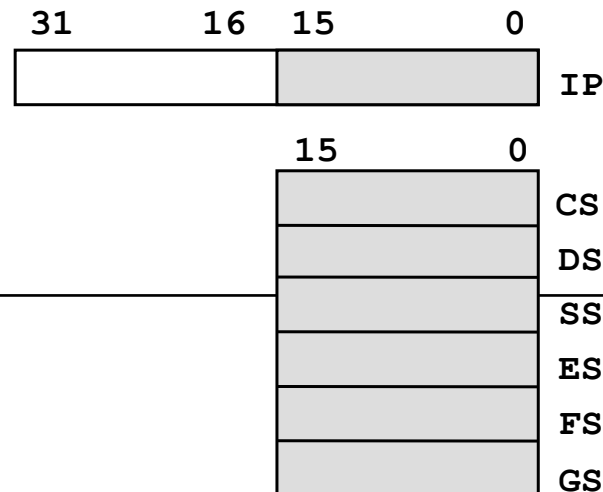
寄存器

- 指令指针寄存器
- 段寄存器
- 标志寄存器
- 控制寄存器0(CR0)
- 通用寄存器

专用寄存器

○ **CR0**的第**0**位**PE**为保护允许位,切换实模式和保护模式

- PE=0, 选择实模式
- PE=1, 选择保护模式



三、MIPS体系结构

——（2）整数乘法单元和寄存器

- MIPS体系结构认为乘法非常重要，应该用硬件实现乘法指令，这在RISC CPU中并不常见。
- 乘法结果寄存器是互锁的：只有在整数乘法运算完成，得到完整的结果后，才能读取结果寄存器。

思路：牺牲速度以换取执行简单和节省芯片空间

三、MIPS体系结构——（3）寻址方式

- 只有加载或存储指令可以访问存储器
- 存储器寻址方式：
 - 基址-偏移寻址：存储单元的地址是某个寄存器与指令中的偏移量之和
- MIPS只有三种指令格式：

| | | | | | |
|--------|----|----|----|-------|-------|
| opcode | rs | rt | rd | shamt | funct |
|--------|----|----|----|-------|-------|

| | | | |
|--------|----|----|--------|
| opcode | rs | rt | offset |
|--------|----|----|--------|

| | |
|--------|---------|
| opcode | address |
|--------|---------|

二、MIPS体系结构

——（4）存储器和寄存器中的数据类型

- MIPS CPU的一次操作可读出或写入1~8个字节的数据

| C名称 | MIPS 名称 | Size/字节 | 汇编助记符 |
|-------------|----------|---------|-------|
| long long | dword | 8 | “d” |
| int / long* | word | 4 | “w” |
| short | halfword | 2 | “h” |
| char | byte | 1 | “b” |

*MIPS编译器提供了64位指针，它把long解释成64位数据，总之long不应该小于int

三、MIPS指令与汇编

对于一条汇编语言指令来说，有两个问题要解决

- 要指出进行什么操作
- 要指出大多数指令涉及的**操作数**和**操作结果**放在何处

1. 指令格式

2. 寻址方式

3. 指令系统

MIPS: 操作数

| 名称 | 实例 | 注释 |
|----------------------|--|---|
| 32个寄存器 | \$s0, ...\$s7 \$t0,...\$t7 | 数据的快速定位，算术运算指令的操作数必须在寄存器中 |
| 2 ³⁰ 个存储字 | 存储器[1] ... 存储器[2 ³⁰] | MIPS只能使用数据传送指令访问。 MIPS中使用字寻址，相邻数据字的字地址相差4。 |

常数？

MIPS: 操作数

○ 寄存器为什么只有**32**个？

- 因为电信号传输距离越远，传输时间就越长，寄存器太多将会延长时钟周期。

○ 存储器对齐限制：

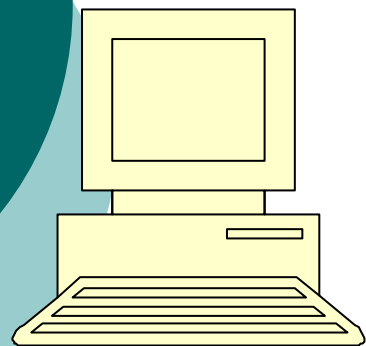
- MIPS中字的地址必须是4的倍数

| | |
|----------------|------------|
| 存储器: 12 | 100 |
| 8 | 10 |
| 4 | 101 |
| 0 | 1 |

地址

数据

MIPS指令与汇编



所有的**MIPS**指令都是**32**位长

1. 指令格式

2. 寻址方式

3. 指令系统

✓ R-format: 所有其他

✓ I-format: 用于有立即数的指令,

lw, sw, beq, bne

(不包含移位指令)

✓ J-format: 无条件跳转 j, 并连接jal

MIPS指令格式

(1) R-型 指令

- 一条32位的MIPS R型指令按下表bit数划分为6个字段： $6 + 5 + 5 + 5 + 5 + 6 = 32\text{bit}$

| | | | | | |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |
|---|---|---|---|---|---|

- 各段含义如下：

| opcode | rs | rt | rd | shamt | funct |
|--------|-------------|-------------|--------------------|-------|-------|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 操作码 | 第1个源 操作数 | 第2个源 操作数 | 目标寄 存器 (放结果) | 偏移量 | 函数码 |

MIPS指令格式

(1) R-Format 例子

- MIPS R型指令:

`add $8, $9, $10`

10进制表示:

| | | | | | |
|---|---|----|---|---|----|
| 0 | 9 | 10 | 8 | 0 | 32 |
|---|---|----|---|---|----|

二进制表示的话:

| | | | | | |
|--------|-------|-------|-------|-------|--------|
| 000000 | 01001 | 01010 | 01000 | 00000 | 100000 |
|--------|-------|-------|-------|-------|--------|

hex

16进制: 012A 4020_{hex}

10进制: 19,546,144_{ten}

MIPS指令格式

(2) I-型 指令

- 一条32位的MIPS I型指令按下表bit数划分为4个字段：6 + 5 + 5 + 16 = 32bit

| | | | |
|---|---|---|----|
| 6 | 5 | 5 | 16 |
|---|---|---|----|

- 各段含义如下：

| opcode | rs | rt | address |
|--------|--------------------|--------------------|-------------------|
| ↓ | ↓ | ↓ | ↓ |
| 操作码 | 第1个源 操作数 寄存器 | 目标寄 存器(放 结果) | 地址相 对基址 偏移量 |

MIPS指令格式

(2) I-型 指令

- **I-型 指令**：装入/存储指令、分支指令和立即数运算指令的格式
 - 数据装入： $Rt = Mem[Rs + Address]$
 - 数据存储： $Mem[Rs + Address] = Rt$

| Op | Rs | Rt | Address |
|----|----|----|---------|
|----|----|----|---------|

6位

5位

5位

16位

例1： 装入/存储指令

`lw $s1, 100($s2)`

`8E510064`

MIPS指令格式

(2) I-型 指令

○ 例2：分支指令

- $\text{if}(\text{Rs} < \text{relation} > \text{Rt}) \text{ goto } (\text{PC}+4)+\text{Address}$

| | | | |
|----|----|----|---------|
| Op | Rs | Rt | Address |
|----|----|----|---------|

6位

5位

5位

16位

- 分支指令采用的寻址方式为**PC相对寻址**——分支目标的地址是PC+4与指令中的偏移量之和

MIPS指令格式

(2) I-Format 例子(1/2)

例3: `addi $21, $22, -50`

`opcode` = 8 (具体是什么操作)

`rs` = 22 (操作数寄存器)

`rt` = 21 (目的寄存器)

`immediate` = -50 (by default, this is decimal)

请写出指令翻译成机器码。

MIPS指令格式

(2) I-Format 例子 (2/2)

例3: `addi $21, $22, -50`

十进制指令格式:

| | | | |
|---|----|----|-----|
| 8 | 22 | 21 | -50 |
|---|----|----|-----|

二进制指令格式:

| | | | |
|--------|-------|-------|------------------|
| 001000 | 10110 | 10101 | 1111111111001110 |
|--------|-------|-------|------------------|

十六进制表示: `22D5 FFCEhex`

十进制表示: `584,449,998ten`

MIPS指令格式

(3) J-Format 指令 (1/2)

○ J型指令格式:



○ 每部分的名字是:



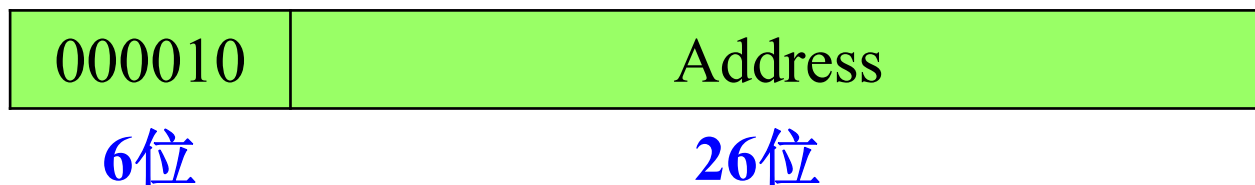
○ 关键点:

- ✓ 必须让 opcode 部分与R和I型指令一致,以便于电路理解执行
- ✓ 其他字段都节省出来给跳转的目的地址用以表示很大的跳转范围.

MIPS指令格式

(3) J-Format 指令

○ 例：跳转指令的格式



- 跳转指令采用伪直接寻址——跳转地址为指令中的26位常数与PC中的高位拼接得到

例： j 10000 08002710

0000 1000 0000 0000 0010 0111 0001 0000

MIPS指令格式

(3) J-Format 指令

○ 总之：

- 新的PC = { PC[31..28], 目标地址, 00 }

○ 一定要明白上面的每一项是从哪里来的！

○ **Note:** { , , } 表示合并

{ 4 bits , 26 bits , 2 bits } = 32 bit address

- { 1010, 11111111111111111111111111111111, 00 } = 1010111111111111111111111111111100

- Note: 书上用 ||, Verilog里用{ , , }

? 关于MIPS指令格式的思考

○ 指令格式为什么要分三类呢?

- 所有指令长度相同，都是32位
- 要让每一条指令刚好合适，充分发挥作用
-

提示：和ALU及指令译码电路设计有关

小测试

哪条指令可以用10进制的 35_{ten} 表示?

A. add \$0, \$0, \$0

| | | | | | |
|--------|----|----|----|-------|-------|
| opcode | rs | rt | rd | shamt | funct |
|--------|----|----|----|-------|-------|

B. subu \$s0,\$s0,\$s0

| | | | | | |
|--------|----|----|----|-------|-------|
| opcode | rs | rt | rd | shamt | funct |
|--------|----|----|----|-------|-------|

C. lw \$0, 0(\$0)

| | | | | | |
|--------|----|----|--------|--|--|
| opcode | rs | rt | offset | | |
|--------|----|----|--------|--|--|

D. addi \$0, \$0, 35

| | | | | | |
|--------|----|----|-----------|--|--|
| opcode | rs | rt | immediate | | |
|--------|----|----|-----------|--|--|

E. subu \$0, \$0, \$0

| | | | | | |
|--------|----|----|----|-------|-------|
| opcode | rs | rt | rd | shamt | funct |
|--------|----|----|----|-------|-------|

指令怎么能用数字表示呢!

– 数字和寄存器名的对应关系:

0: \$0, 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

– 操作码opcode和其他字段(if necessary)

add: opcode = 0, funct = 32

addi: opcode = 8

subu: opcode = 0, funct = 35

lw: opcode = 35

MIPS寻址方式

1. 指令格式

2. 寻址方式

3. 指令系统

✓ 寄存器寻址

✓ 立即数寻址

✓ 基址或偏移寻址

✓ PC相对寻址

✓ 伪直接寻址

MIPS 寻址方式

(1) 寄存器寻址

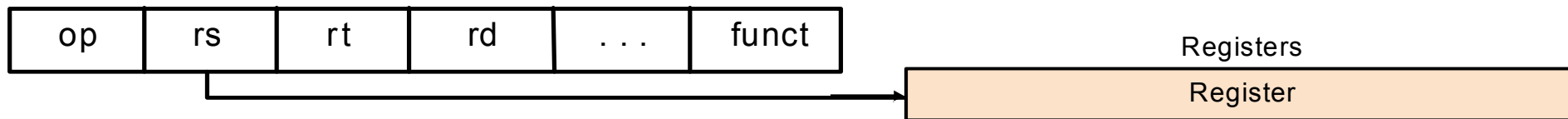
例：MIPS算术运算指令的操作数必须从32个32位寄存器中选取

`add $ t0, $s1, $s2` # 寄存器\$ t0的内容为 $g+h$

`add $ t1, $s3, $s4` # 寄存器\$ t1的内容为 $i+j$

`sub $ s0, $t0, $t1` # 寄存器\$ s0的内容为 $(g+h)-(i+j)$

Register addressing

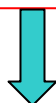


MIPS 寻址方式

(2) 立即数寻址

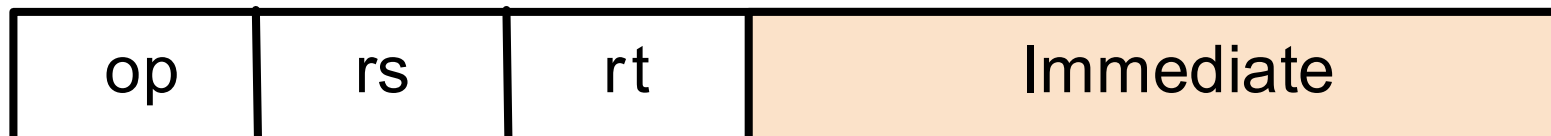
- 以常数作为操作数，无须访问存储器就可以使用常数。
- 因为常数操作数频繁出现，所以在算术指令中加入常数字段，比从存储器中读取常数快得多。

Lw \$ t0, AddrConstant4(\$ zero) # \$ t0 = 常数4
add \$ sp, \$sp, \$t0 # \$ sp = \$ sp + \$t0 (\$ t0 == 4)



addi \$ sp, \$sp, 4 # \$ sp = \$ sp + 4

Immediate addressing



MIPS 寻址方式

(2) 立即数寻址

- 怎样把一个**32**位长的常数装入寄存器**\$ s0**中？

0000 0000 0011 1101 | 0000 1001 0000 0000

61

2304

Load Upper Immediate

Lui \$ s0, 61 # \$ s0= 0000 0000 0011 1101 0000 0000 0000 0000
addi \$ s0, \$s0, 2340 # \$ s0= 0000 0000 0011 1101 0000 1001 0000 0000

拆散和重装大常数由汇编程序来完成

ori \$s5,\$0,0x1000

sll \$s5,\$s5,16 # \$s5=A[]=0x10000000,

Immediate addressing

| | | | |
|----|----|----|-----------|
| op | rs | rt | Immediate |
|----|----|----|-----------|

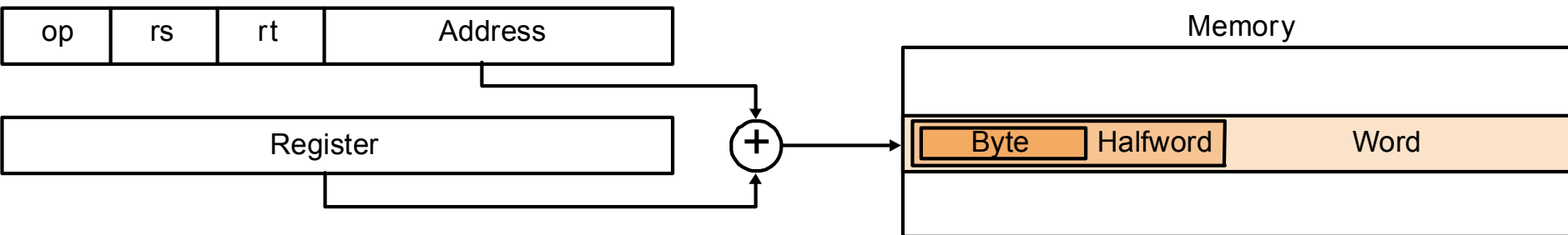
MIPS 寻址方式

(3) 基址或偏移寻址

- 操作数在存储器中，且存储器地址是某寄存器与指令中某常量的和。

Lw \$ t0, 8 (\$ s0) **#\$ s0**中装的是存储器中的地址

3. Base addressing

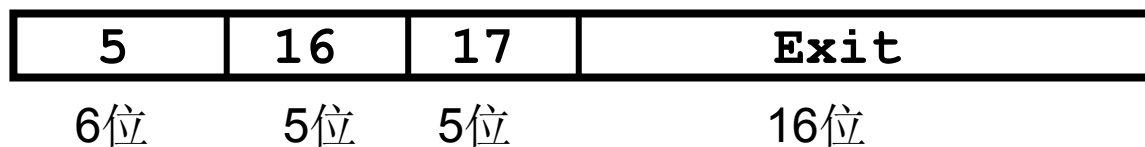


MIPS 寻址方式

(4) PC相对寻址

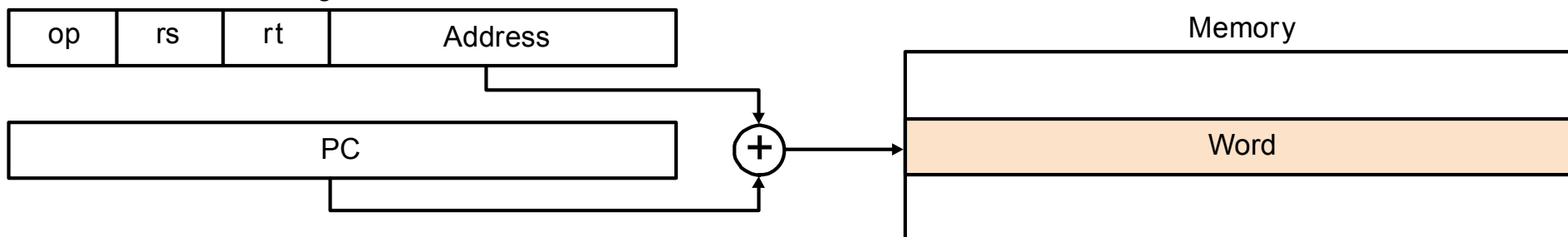
例：条件分支指令

bne \$s0, \$s1, Exit #如果\$s0不等于\$s1，则跳转到Exit



程序计数器=PC寄存器+分支地址

4. PC-relative addressing



MIPS 寻址方式

(4) PC相对寻址

问题1: 为什么选PC寄存器?

因为几乎所有的条件分支指令都是跳转到附近的地址。

问题2: 如何处理16位无法表达的远距离分支?

插入一个无条件跳转到分支目标地址的指令, 把分支指令中的条件变反以决定是否跳过该指令。

Bne \$s0, \$s1, L2



$2^{15} < |L2-PC| < 2^{27}$

```
beq  $s0, $s1, L1  
j    L2
```

L1:

.....

L2:

.....

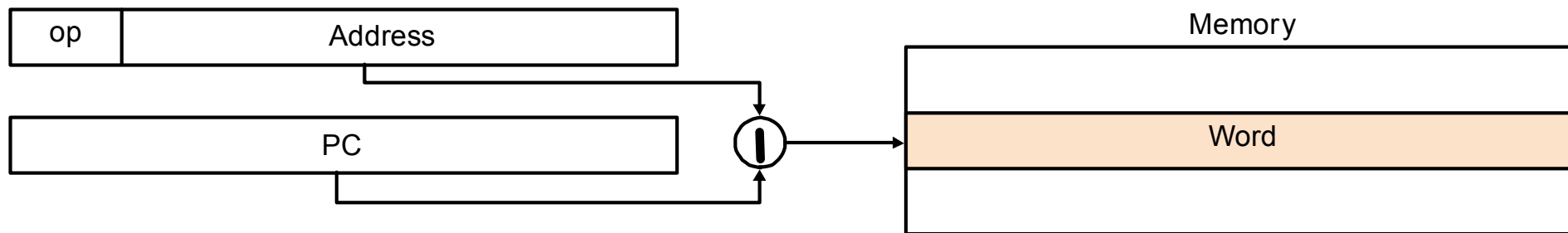
MIPS 寻址方式

(5) 伪直接寻址

跳转地址 = PC中原高4位 | 指令中的26位 | 00

j 10000 #跳转到PC | 10000

5. Pseudodirect addressing



问题1: PC高位限制了跳转的范围，如何跳出高4位限制的范围？

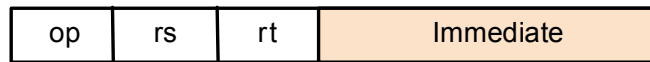
问题2: 目的地址比当前的PC地址小怎么办？

MIPS寻址方式

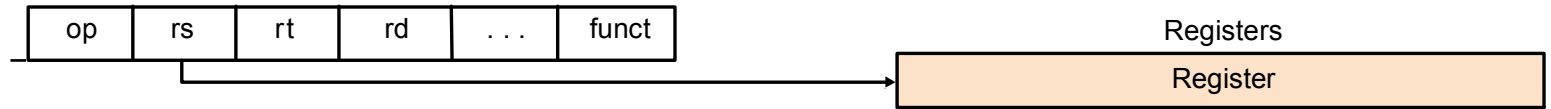
- MIPS采用存取体系结构：只有**取**和**存储**指令实现存取。
- 机器本身只提供一种存储器寻址方式： $c(rx)$
立即数**c** + 寄存器**rx**的值
- 虚拟机则为**取**和**存储**指令提供了几种寻址方式：

| 格式 | 地址计算 |
|-----------------------|-----------------|
| (register) | 寄存器内容 |
| imm | 立即数 |
| Imm(register) | 立即数+寄存器内容 |
| label | lable地址 |
| lable ± imm | lable地址 ± 立即数 |
| Lable ± imm(register) | lable地址 ± 寄存器内容 |

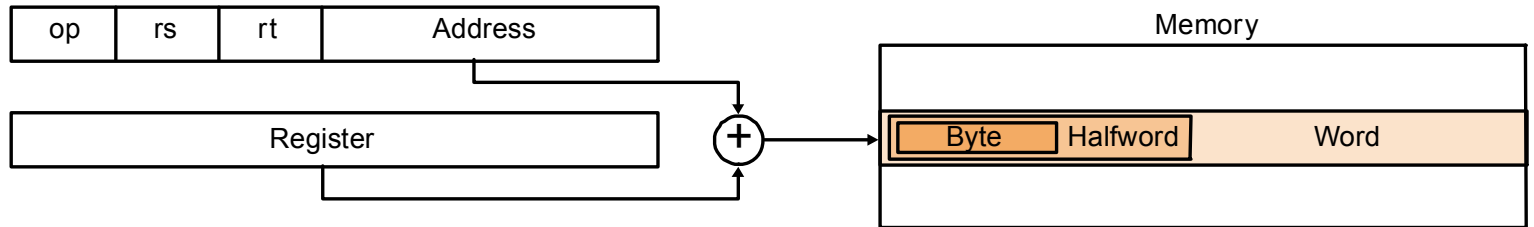
1. Immediate addressing



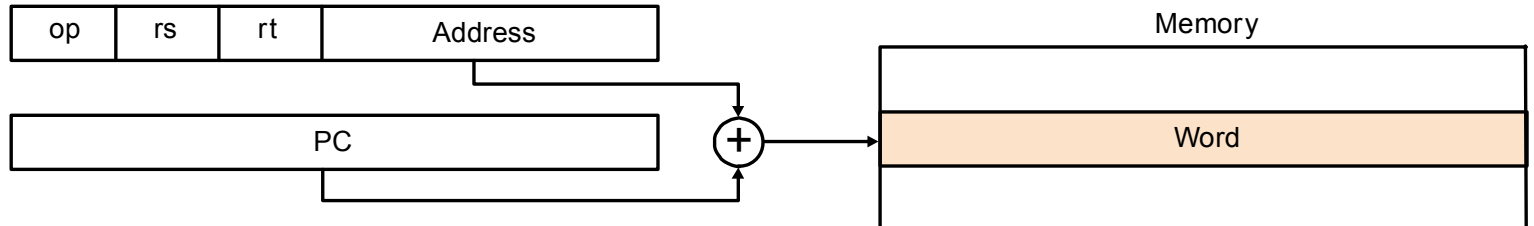
2. Register addressing



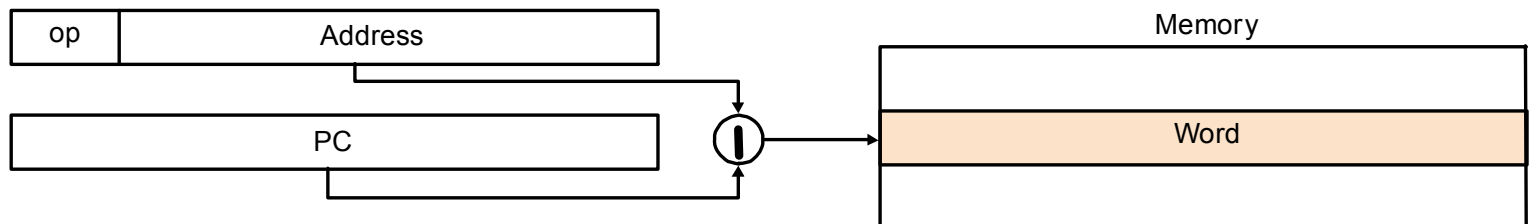
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



MIPS
寻址
方式

MIPS 指令系统

1. 指令格式

2. 寻址方式

3. 指令系统

(1) 数据传送类:

(2) 算术/逻辑运算类:

(3) 控制类:

(4) 输入输出类:

MIPS指令系统:

1、数据传送指令

指令 (存储器访问) 说明

sw \$3, 500(\$4) Store word

sh \$3, 502(\$2) Store half

sb \$2, 41(\$3) Store byte

lw \$1, 30(\$2) Load word

lh \$1, 40(\$3) Load halfword

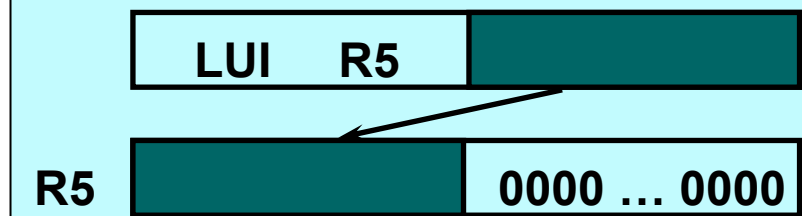
lhu \$1, 40(\$3) Load halfword unsigned

lb \$1, 40(\$3) Load byte

lbu \$1, 40(\$3) Load byte unsigned

lui \$1, 40 Load Upper Immediate

(16 bits shifted left by 16)



Q: Why need lui?

MIPS指令系统:

2、算术\逻辑指令

- 每条指令有且仅有3个操作数，且只执行一个操作
 - Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
 - AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
 - SLL, SRL, SRA, SLLV, SRLV, SRAV

操作数个数固定所需的硬件——

简单

MIPS指令系统:

2、算术\逻辑指令

○ R格式——ALU指令的格式

- Rs、Rt、Rd为寄存器编号
- Funct字段为指令功能代码

| 操作码 | 第一源 操作数 | 第二源 操作数 | 目标源 操作数 | 功能码 | |
|--------|------------|------------|------------|-------|-------|
| 000000 | Rs | Rt | Rd | 00000 | Funct |
| 6位 | 5位 | 5位 | 5位 | 5位 | 6位 |

例: add \$s1, \$s2, \$s3

02538820

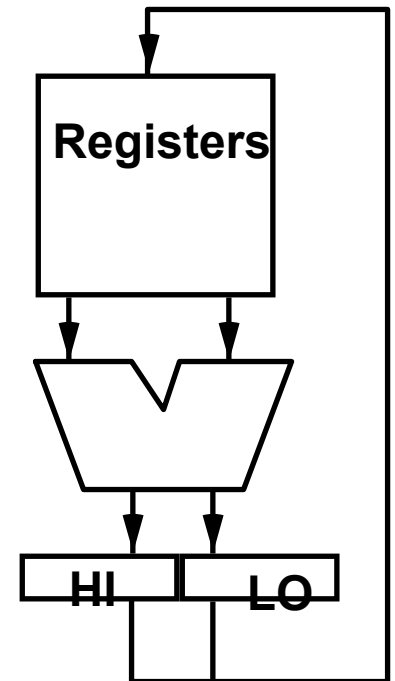
000000 10010 10011 10001 00000 100000

MIPS指令系统: 2、算术\逻辑指令

乘法 / 除法multiply/divide

Start multiply, divide

- MULT rs, rt
- MULTU rs, rt
- DIV rs, rt
- DIVU rs, rt
- Move result from multiply, divide
 - MFHI rd
 - MFLO rd
- Move to HI or LO
 - MTHI rd
 - MTLO rd



Q: Why not Third field for destination?

MIPS指令系统:

2、算术\逻辑指令

| 指令 | 例子 | 含义 |
|---------------|------------------|--|
| add | add \$1,\$2,\$3 | $\$1 = \$2 + \$3$ 3 operands; <u>exception possible</u> |
| subtract | sub \$1,\$2,\$3 | $\$1 = \$2 - \$3$ 3 operands; <u>exception possible</u> |
| add immediate | addi \$1,\$2,100 | $\$1 = \$2 + 100$ + constant; <u>exception possible</u> |
| add unsigned | addu \$1,\$2,\$3 | $\$1 = \$2 + \$3$ 3 operands; <u>no exceptions</u> |

MIPS指令系统:

2、算术\逻辑指令

| 指令 | 例子 | 含义 |
|-------------------|-------------------|---|
| subtract unsigned | subu \$1,\$2,\$3 | $\$1 = \$2 - \$3$ 3 operands; <u>no exceptions</u> |
| add imm. unsign. | addiu \$1,\$2,100 | $\$1 = \$2 + 100 + \text{constant};$ <u>no exceptions</u> |
| multiply | mult \$2,\$3 | $\text{Hi, Lo} = \$2 \times \3 64-bit signed product |
| multiply unsigned | multu \$2,\$3 | $\text{Hi, Lo} = \$2 \times \3 64-bit unsigned product |

MIPS指令系统:

2、算术\逻辑指令

| 指令 | 例子 | 含义 |
|-----------------|--------------|--|
| divide | div \$2,\$3 | Lo = $\$2 \div \3 , Lo = quotient, Hi = remainder, Hi = $\$2 \bmod \3 |
| divide unsigned | divu \$2,\$3 | Lo = $\$2 \div \3 , Unsigned quotient & remainder Hi = $\$2 \bmod \3 |
| Move from Hi | mfhi \$1 | $\$1 = \text{Hi}$ Used to get copy of Hi |
| Move from Lo | mflo \$1 | $\$1 = \text{Lo}$ Used to get copy of Lo |

MIPS 算术指令

| 指令 | 例子 | 含义 | 说明 |
|-------------------|-------------------|---|---------------------------------------|
| add | add \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; <u>exception possible</u> |
| subtract | sub \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; <u>exception possible</u> |
| add immediate | addi \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; <u>exception possible</u> |
| add unsigned | addu \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; <u>no exceptions</u> |
| subtract unsigned | subu \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; <u>no exceptions</u> |
| add imm. unsign. | addiu \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; <u>no exceptions</u> |
| multiply | mult \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit signed product |
| multiply unsigned | multu \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit unsigned product |
| divide | div \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Lo = quotient, Hi = remainder |
| divide unsigned | divu \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Unsigned quotient & remainder |
| Move from Hi | mfhi \$1 | $\$1 = \text{Hi}$ | Used to get copy of Hi |
| Move from Lo | mflo \$1 | $\$1 = \text{Lo}$ | Used to get copy of Lo |

Q: Which add for address arithmetic? Which add for integers?

Q: Which add for address arithmetic?

- 无符号整数一般都是用来表示存储器地址，溢出即使发生，也大多数被忽略掉了，因此 **MIPS** 采用了两种不同的算术运算指令来分别处理
 - (1) `addu`, `addiu`, `subu` 在发生溢出的时候不产生异常，
 - (2) `add`, `addi`, `sub` 在发生溢出时产生异常

究竟何时发生溢出？

加法：两正数相加结果为负，两负数相加结果为正

减法：正数减负数结果为负，负数减正数结果为正

MIPS在检测到溢出发生时会产生一个异常，造成溢出指令的地址被存到一个特定寄存器中

乘法： **multu Hi**不为0， **mult Hi**各位不等于**Lo**符号位

MIPS乘/除法指令都忽略了溢出的情况，程序必须自己判断得到的积/商是否超出了**32**位寄存器能表示的范围，还必须自己处理除零操作

MIPS指令系统:

2、算术\ **逻辑** 指令

| 指令 | 例子 | 含义 |
|------------|------------------------|---|
| and | and \$1,\$2,\$3 | $\\$1 = \\$2 \ \& \ \\3 3 reg. operands; Logical AND |
| or | or \$1,\$2,\$3 | $\\$1 = \\$2 \ \ \\3 3 reg. operands; Logical OR |
| xor | xor \$1,\$2,\$3 | $\\$1 = \\$2 \ \wedge \ \\3 3 reg. operands; Logical XOR |
| nor | nor \$1,\$2,\$3 | $\\$1 = \sim(\\$2 \ \ \\$3)$ 3 reg. operands; Logical NOR |

MIPS指令系统:

2、算术\逻辑指令

指令

例子

含义

and immediate

andi \$1,\$2,10

$\$1 = \$2 \& 10$

Logical AND reg, constant

or immediate

ori \$1,\$2,10

$\$1 = \$2 | 10$

Logical OR reg, constant

xor immediate

xori \$1, \$2,10

$\$1 = \sim \$2 \& \sim 10$

Logical XOR reg, constant

MIPS指令系统:

2、算术\逻辑指令

| 指令 | 例子 | 含义 |
|---------------------|------------------|---|
| shift left logical | sll \$1,\$2,10 | $\$1 = \$2 \ll 10$ Shift left by constant |
| shift right logical | srl \$1,\$2,10 | $\$1 = \$2 \gg 10$ Shift right by constant |
| shift right arithm. | sra \$1,\$2,10 | $\$1 = \$2 \gg 10$ Shift right (sign extend) |
| shift left logical | sllv \$1,\$2,\$3 | $\$1 = \$2 \ll \$3$ Shift left by variable |
| shift right logical | srlv \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ Shift right by variable |
| shift right arithm. | srav \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ Shift right arith. by variable |

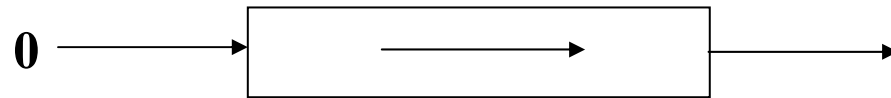
MIPS指令系统:

2、算术\逻辑指令

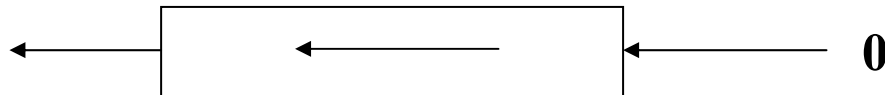
○ 移位指令



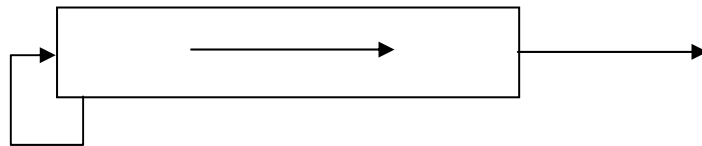
SLL 逻辑左移
(Shift Left Logical)



SHR 逻辑右移
(Shift Right Logical)



SLA 算术左移
(Shift Left Arithmetic)



SAR 算术右移
(Shift Right Arithmetic)

MIPS指令系统:

2、算术\逻辑指令

| 指令 | 例子 | 含义 | 说明 |
|---------------------|------------------|----------------------------|--------------------------------|
| and | and \$1,\$2,\$3 | $\$1 = \$2 \& \$3$ | 3 reg. operands; Logical AND |
| or | or \$1,\$2,\$3 | $\$1 = \$2 \mid \$3$ | 3 reg. operands; Logical OR |
| xor | xor \$1,\$2,\$3 | $\$1 = \$2 \wedge \$3$ | 3 reg. operands; Logical XOR |
| nor | nor \$1,\$2,\$3 | $\$1 = \sim(\$2 \mid \$3)$ | 3 reg. operands; Logical NOR |
| and immediate | andi \$1,\$2,10 | $\$1 = \$2 \& 10$ | Logical AND reg, constant |
| or immediate | ori \$1,\$2,10 | $\$1 = \$2 \mid 10$ | Logical OR reg, constant |
| xor immediate | xori \$1, \$2,10 | $\$1 = \sim\$2 \& \sim 10$ | Logical XOR reg, constant |
| shift left logical | sll \$1,\$2,10 | $\$1 = \$2 \ll 10$ | Shift left by constant |
| shift right logical | srl \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right by constant |
| shift right arithm. | sra \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right (sign extend) |
| shift left logical | sllv \$1,\$2,\$3 | $\$1 = \$2 \ll \$3$ | Shift left by variable |
| shift right logical | srlv \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ | Shift right by variable |
| shift right arithm. | srav \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ | Shift right arith. by variable |

Q: Can some multiply by 2^i ? Divide by 2^i ? Invert?

MIPS什么时候做符号位扩展?

- 符号位扩展:

Examples of sign extending 8 bits to 16 bits:

00001010 \Rightarrow 00000000 00001010

10001100 \Rightarrow 11111111 10001100

- 什么时候立即数被符号位扩展?

- 算术指令 总是将立即数做符号位扩展 即便指令是无符号的!
- 逻辑指令 (andi, ori通常处理无符号数) 不对立即数做符号位扩展 (They are zero extended)
- Load/Store address computations **always sign extend immediates**

MIPS什么时候做符号位扩展?

- Multiply/Divide have no immediate operands however:
 - “unsigned” \Rightarrow treat operands as unsigned
- The data loaded by the instructions lb and lh are extended as follows (“unsigned” \Rightarrow don't extend):
 - lbu, lhu are zero extended
 - lb, lh are sign extended

Q: 既然已经没有了立即数参与运算了，为什么还需要**addu**这个指令?(无符号)

MIPS什么时候做符号位扩展？

- **addiu**在执行前也要将指令中的立即数进行符号位扩展，因为：虽然**U**代表无符号数，但是**addiu**事实上都被看作一条不会发生溢出的**add**指令，因此常用它来加上一个负的立即数。
- 同理：**addu**，**sltiu**，**sltu**

3、控制类指令

○ 比较与分支

- BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
- BNE rs, rt, offset <>

○ 与零比较并分支

- BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
- BGTZ rs, offset >
- BLTZ <
- BGEZ >=
- BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
- BGEZAL >=0

- 如果要实现其他比较功能，则需要两个以上的指令
- 基本上都是与零比较（这个速度快）

MIPS指令系统:

3、控制类指令

跳转分支和比较指令

| <i>Instruction</i> | <i>Example</i> | <i>Meaning</i> |
|---------------------|---------------------|---|
| branch on equal | beq \$1, \$2, 100 | if (\$1 == \$2) go to PC+4+100 <i>Equal test; PC relative branch</i> |
| branch on not eq. | bne \$1, \$2, 100 | if (\$1 != \$2) go to PC+4+100 <i>Not equal test; PC relative</i> |
| set on less than | slt \$1, \$2, \$3 | if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; 2' s comp.</i> |
| set less than imm. | slti \$1, \$2, 100 | if (\$2 < 100) \$1=1; else \$1=0 <i>Compare < constant; 2' s comp.</i> |
| set less than uns. | sltu \$1, \$2, \$3 | if (\$2 < \$3) \$1=1; else \$1=0 <i>Compare less than; natural numbers</i> |
| set l. t. imm. uns. | sltiu \$1, \$2, 100 | if (\$2 < 100) \$1=1; else \$1=0 <i>Compare < constant; natural numbers</i> |
| jump | j 10000 | go to 10000 <i>Jump to target address</i> |
| jump register | jr \$31 | go to \$31 <i>For switch, procedure return</i> |
| jump and link | jal 10000 | \$31 = PC + 4; go to 10000 <i>For procedure call</i> |

MIPS指令系统:

3、控制类指令

跳转分支和比较指令

| <i>Instruction</i> | <i>Example</i> | <i>Meaning</i> |
|--------------------|------------------------|---|
| branch on equal | beq \$1,\$2,100 | if (\$1==\$2) go to PC+4+100 <i>Equal test; PC relative branch</i> |
| branch on not eq. | bne \$1,\$2,100 | if (\$1!= \$2) go to PC+4+100 <i>Not equal test; PC relative</i> |
| jump | j 10000 | go to 10000 <i>Jump to target address</i> |
| jump register | jr \$31 | go to \$31 <i>For switch, procedure return</i> |
| jump and link | jal 10000 | \$31 = PC + 4; go to 10000 <i>For procedure call</i> |

3、控制类指令

跳转分支和比较指令

| <i>Instruction</i> | <i>Example</i> | <i>Meaning</i> |
|---------------------|--|----------------|
| set on less than | <code>slt \$1, \$2, \$3</code> if ($\$2 < \3) $\$1=1$; else $\$1=0$ <i>Compare less than; 2's comp.</i> | |
| set less than imm. | <code>slti \$1, \$2, 100</code> if ($\$2 < 100$) $\$1=1$; else $\$1=0$ <i>Compare < constant; 2's comp.</i> | |
| set less than uns. | <code>sltu \$1, \$2, \$3</code> if ($\$2 < \3) $\$1=1$; else $\$1=0$ <i>Compare less than; natural numbers</i> | |
| set l. t. imm. uns. | <code>sltiu \$1, \$2, 100</code> if ($\$2 < 100$) $\$1=1$; else $\$1=0$ <i>Compare < constant; natural numbers</i> | |

有符号与无符号比较运算

\$1 = 0...00 0000 0000 0000 0001_{two}

\$2 = 0...00 0000 0000 0000 0010_{two}

\$3 = 1...11 1111 1111 1111 1111_{two}

○ After executing these instructions:

slt \$4, \$2, \$1 ; if (\$2 < \$1) \$4=1; else \$4=0

slt \$5, \$3, \$1 ; if (\$3 < \$1) \$5=1; else \$5=0

sltu \$6, \$2, \$1 ; if (\$2 < \$1) \$6=1; else \$6=0

sltu \$7, \$3, \$1 ; if (\$3 < \$1) \$7=1; else \$7=0

○ What are values of registers \$4 - \$7? Why?

\$4 = ; \$5 = ; \$6 = ; \$7 = ;

小结:

MIPS 编译器寄存器使用约定

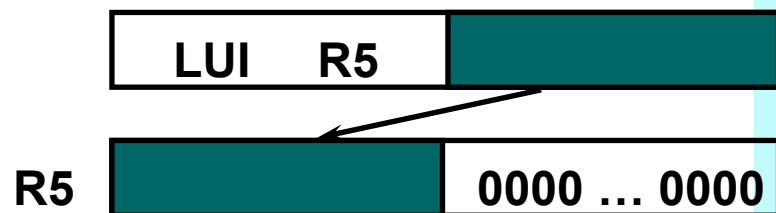
| 名称 | 寄存器号 | 用途 | 在调用时是否保留 |
|-------------|-------|-----------|----------|
| \$zero | 0 | 常数 0 | n/a |
| \$at | 1 | 为汇编程序保留 | |
| \$v0-\$v1 | 2-3 | 结果值和表达式求值 | no |
| \$a0-\$a3 | 4-7 | 参数 | no |
| \$t0-\$t7 | 8-15 | 临时变量 | no |
| \$s0-\$s7 | 16-23 | 保存 | yes |
| \$t18-\$t19 | 24-25 | 其他临时变量 | no |
| \$k0-\$k1 | 26-27 | 为操作系统保留 | |
| \$gp | 28 | 全局指针 | yes |
| \$sp | 29 | 栈指针 | yes |
| \$fp | 30 | 框架指针 | yes |
| \$ra | 31 | 返回地址 | yes |

MIPS指令系统:

1、数据传送指令

指令 (存储器访问) 说明

| | | |
|-----|---------------|--|
| sw | \$3, 500(\$4) | Store word |
| sh | \$3, 502(\$2) | Store half |
| sb | \$2, 41(\$3) | Store byte |
| lw | \$1, 30(\$2) | Load word |
| lh | \$1, 40(\$3) | Load halfword |
| lhu | \$1, 40(\$3) | Load halfword unsigned |
| lb | \$1, 40(\$3) | Load byte |
| lbu | \$1, 40(\$3) | Load byte unsigned |
| lui | \$1, 40 | Load Upper Immediate (16 bits shifted left by 16) |



Q: Why need lui?

MIPS 算术指令

| 指令 | 例子 | 含义 | 说明 |
|-------------------|-------------------|---|--|
| add | add \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; exception possible |
| subtract | sub \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; exception possible |
| add immediate | addi \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; exception possible |
| add unsigned | addu \$1,\$2,\$3 | $\$1 = \$2 + \$3$ | 3 operands; no exceptions |
| subtract unsigned | subu \$1,\$2,\$3 | $\$1 = \$2 - \$3$ | 3 operands; no exceptions |
| add imm. unsign. | addiu \$1,\$2,100 | $\$1 = \$2 + 100$ | + constant; no exceptions |
| multiply | mult \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit signed product |
| multiply unsigned | multu \$2,\$3 | Hi, Lo = $\$2 \times \3 | 64-bit unsigned product |
| divide | div \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Lo = quotient, Hi = remainder |
| divide unsigned | divu \$2,\$3 | Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3 | Unsigned quotient & remainder |
| Move from Hi | mfhi \$1 | $\$1 = \text{Hi}$ | Used to get copy of Hi |
| Move from Lo | mflo \$1 | $\$1 = \text{Lo}$ | Used to get copy of Lo |

Q: Which add for address arithmetic? Which add for integers?

MIPS指令系统:


2、算术\逻辑指令

| 指令 | 例子 | 含义 | 说明 |
|---------------------|------------------|----------------------------|--------------------------------|
| and | and \$1,\$2,\$3 | $\$1 = \$2 \& \$3$ | 3 reg. operands; Logical AND |
| or | or \$1,\$2,\$3 | $\$1 = \$2 \$3$ | 3 reg. operands; Logical OR |
| xor | xor \$1,\$2,\$3 | $\$1 = \$2 \wedge \$3$ | 3 reg. operands; Logical XOR |
| nor | nor \$1,\$2,\$3 | $\$1 = \sim(\$2 \$3)$ | 3 reg. operands; Logical NOR |
| and immediate | andi \$1,\$2,10 | $\$1 = \$2 \& 10$ | Logical AND reg, constant |
| or immediate | ori \$1,\$2,10 | $\$1 = \$2 10$ | Logical OR reg, constant |
| xor immediate | xori \$1, \$2,10 | $\$1 = \sim\$2 \& \sim 10$ | Logical XOR reg, constant |
| shift left logical | sll \$1,\$2,10 | $\$1 = \$2 \ll 10$ | Shift left by constant |
| shift right logical | srl \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right by constant |
| shift right arithm. | sra \$1,\$2,10 | $\$1 = \$2 \gg 10$ | Shift right (sign extend) |
| shift left logical | sllv \$1,\$2,\$3 | $\$1 = \$2 \ll \$3$ | Shift left by variable |
| shift right logical | srlv \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ | Shift right by variable |
| shift right arithm. | srav \$1,\$2,\$3 | $\$1 = \$2 \gg \$3$ | Shift right arith. by variable |

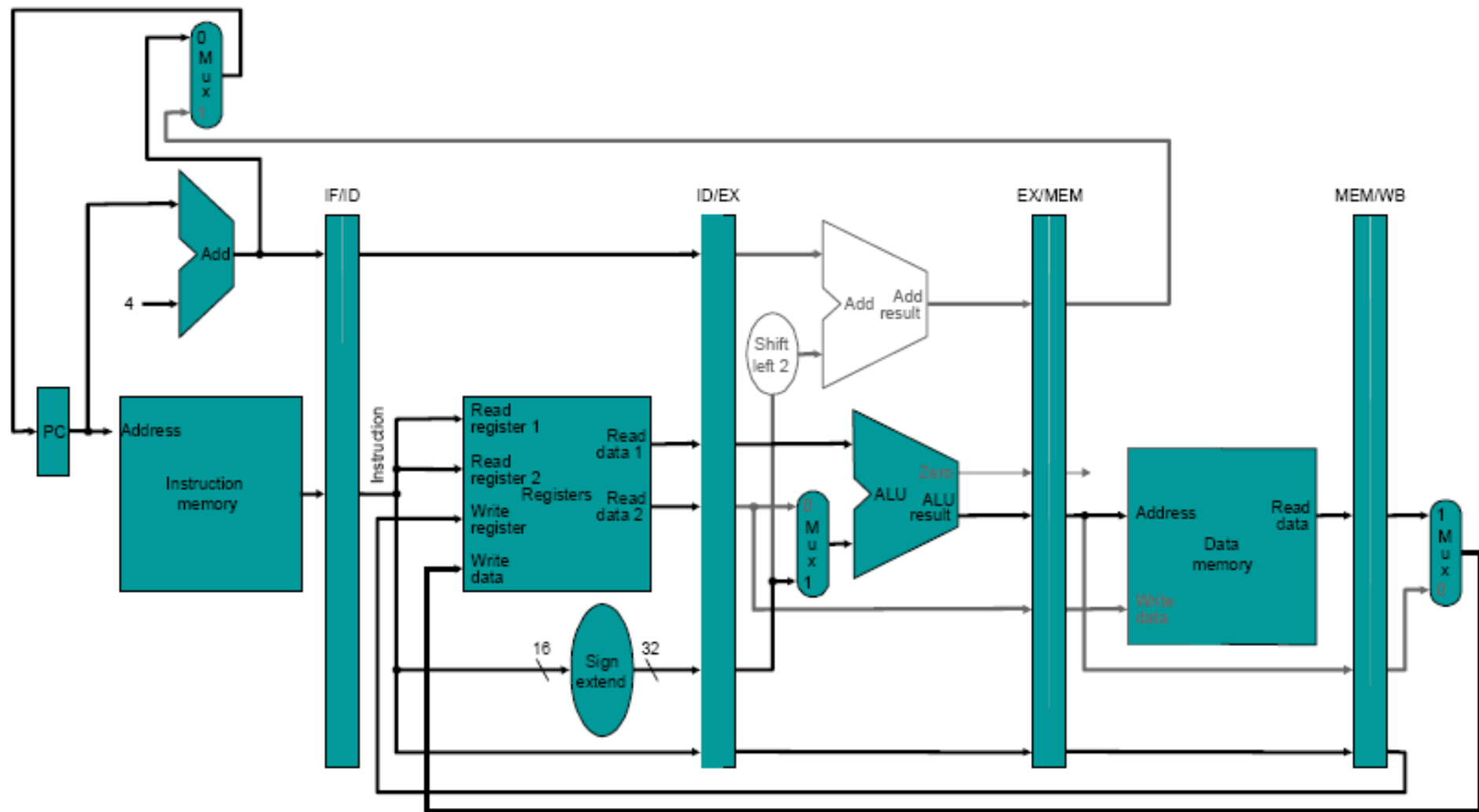
3、控制类指令

跳转分支和比较指令

| <i>Instruction</i> | <i>Example</i> | <i>Meaning</i> |
|---------------------|---------------------|---|
| branch on equal | beq \$1, \$2, 100 | if ($\$1 == \2) go to PC+4+100 <i>Equal test; PC relative branch</i> |
| branch on not eq. | bne \$1, \$2, 100 | if ($\$1 \neq \2) go to PC+4+100 <i>Not equal test; PC relative</i> |
| set on less than | slt \$1, \$2, \$3 | if ($\$2 < \3) \$1=1; else \$1=0 <i>Compare less than; 2' s comp.</i> |
| set less than imm. | slti \$1, \$2, 100 | if ($\$2 < 100$) \$1=1; else \$1=0 <i>Compare < constant; 2' s comp.</i> |
| set less than uns. | sltu \$1, \$2, \$3 | if ($\$2 < \3) \$1=1; else \$1=0 <i>Compare less than; natural numbers</i> |
| set l. t. imm. uns. | sltiu \$1, \$2, 100 | if ($\$2 < 100$) \$1=1; else \$1=0 <i>Compare < constant; natural numbers</i> |
| jump | j 10000 | go to 10000 <i>Jump to target address</i> |
| jump register | jr \$31 | go to \$31 <i>For switch, procedure return</i> |
| jump and link | jal 10000 | \$31 = PC + 4; go to 10000 <i>For procedure call</i> |

- 
-
- MIPS32架构中有32个通用寄存器，虽然硬件没有强制性的指定寄存器使用规则，在实际使用中，这些寄存器的用法都遵循一系列约定。这些约定与硬件确实无关，但如果你想使用别人的代码，编译器和操作系统，你最好是遵循这些约定。
 - 为什么MIPS架构中没有X86中的PC(程序计数)寄存器？
 - **MIPS中程序怎么计数？**

数据通路流水线化



MIPS计算机硬件对过程的支持

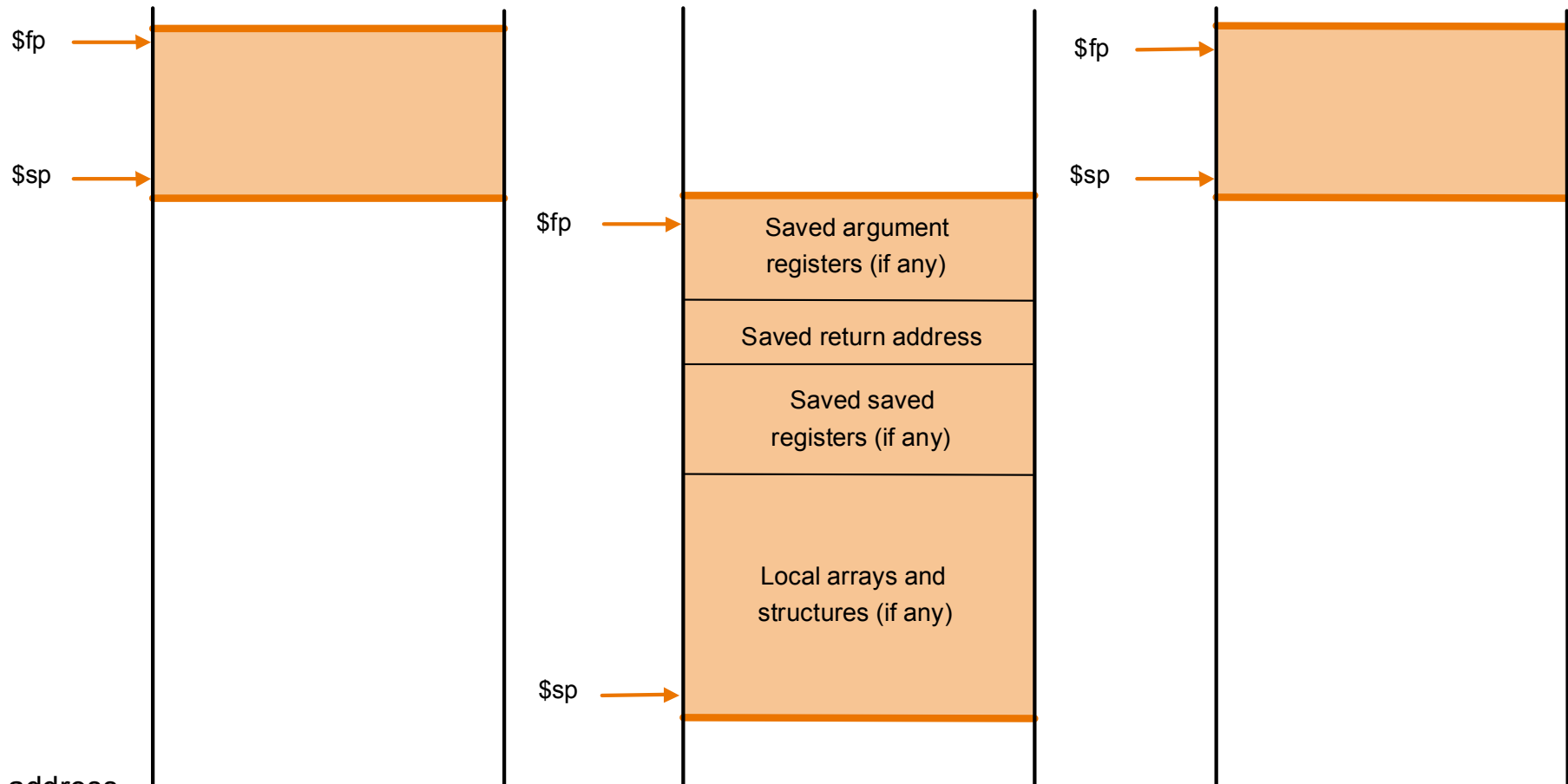
○ 为新数据分配空间

- 利用堆栈存储过程中不适合用寄存器保存的局部变量（如局部数组、或结构）
- **过程框架**：也叫活动记录，是指包含了过程保存的寄存器和局部变量的堆栈段。
- 下图给出了过程调用**之前**、**之中**和**之后**的堆栈状态

MIPS计算机硬件对过程的支持

○ 地址由高到低

High address



a.

b.

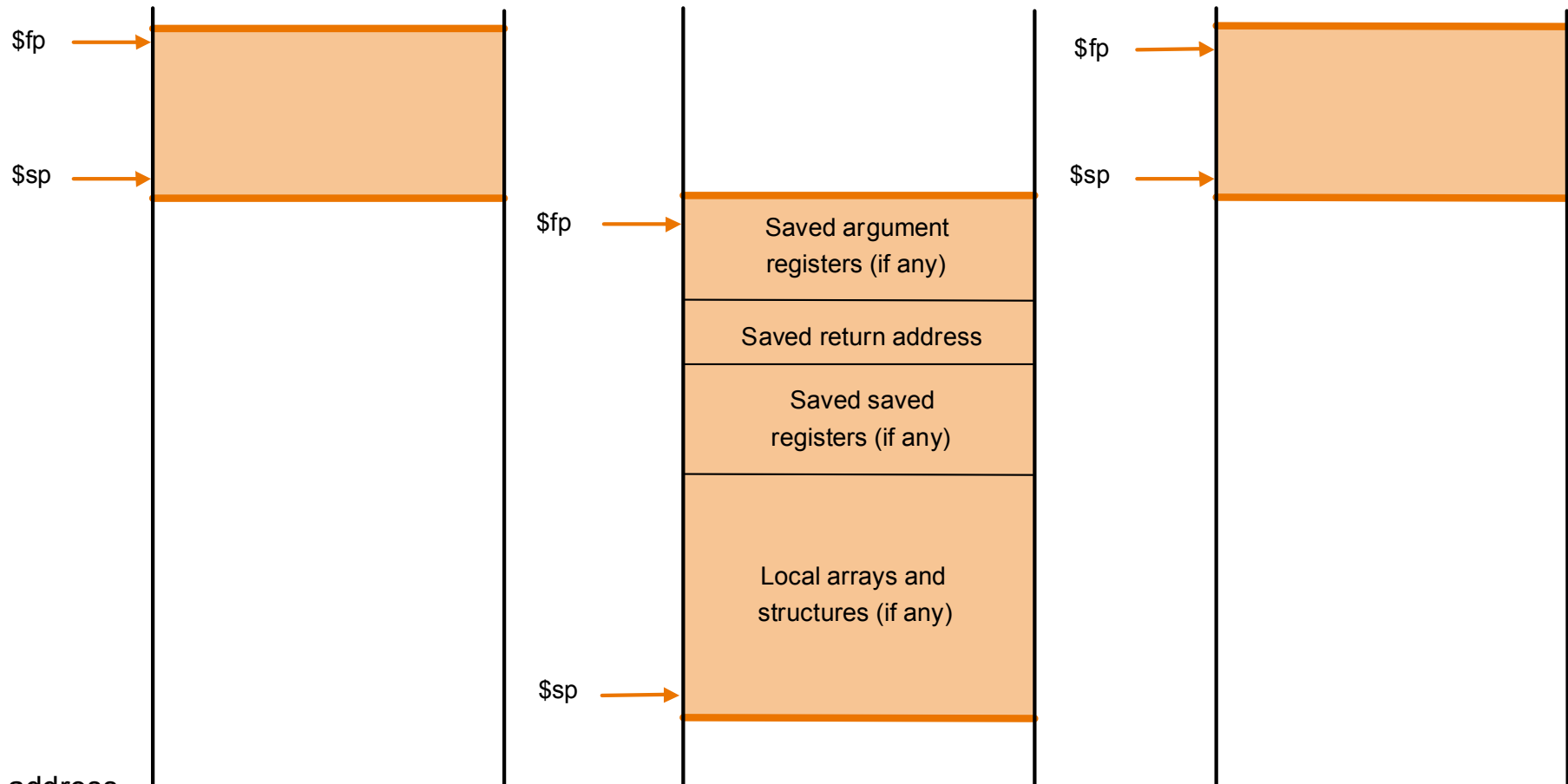
c.

- 框架指针**\$fp**指向框架的第一个字，通常是保存调用的参数寄存器；

MIPS计算机硬件对过程的支持

○ 地址由高到低

High address



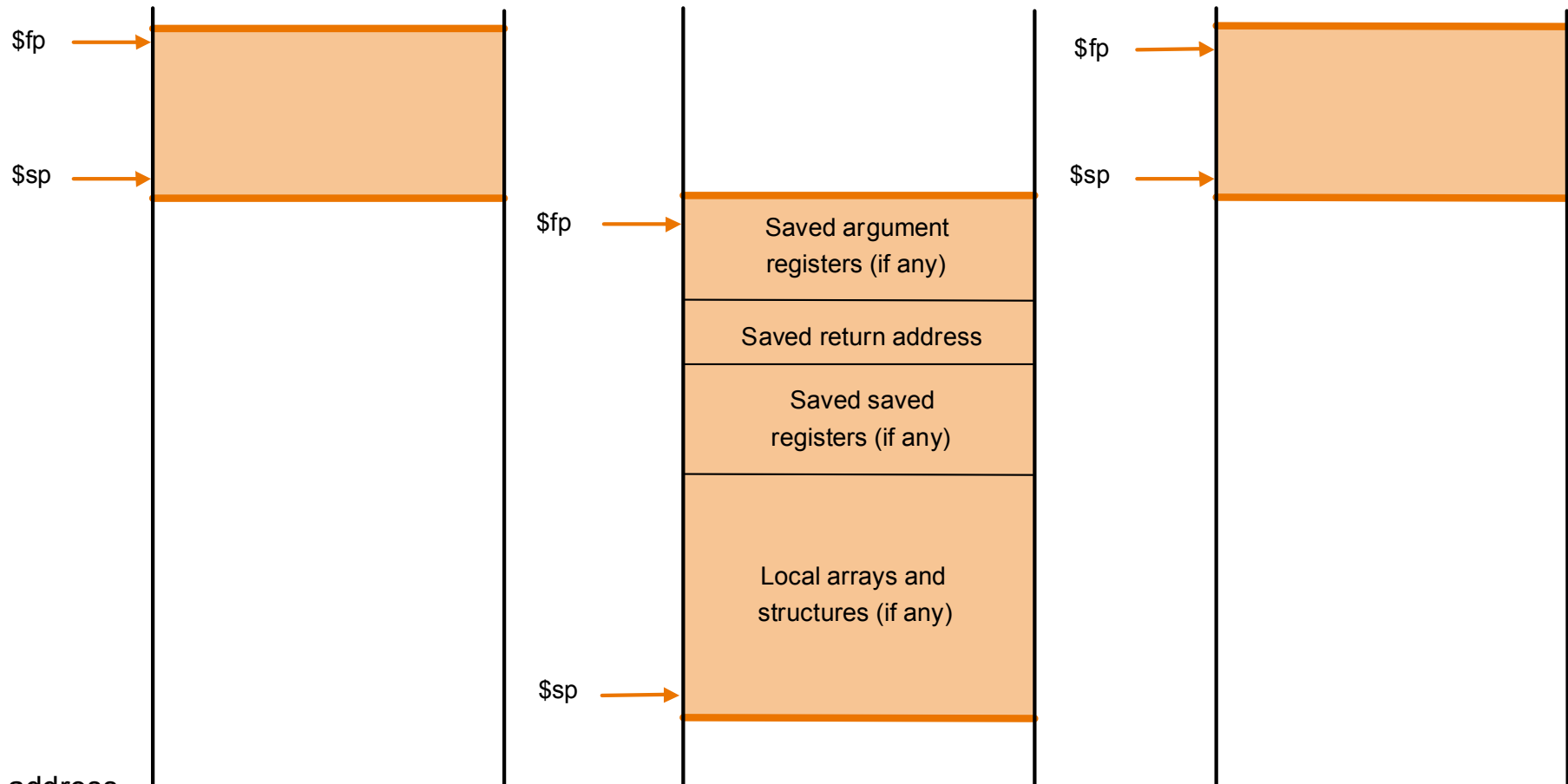
Low address

- a. b. c.
- 栈指针**\$sp**指向栈顶，在程序执行的过程中栈指针有可能改变；

MIPS计算机硬件对过程的支持

○ 地址由高到低

High address



a.

b.

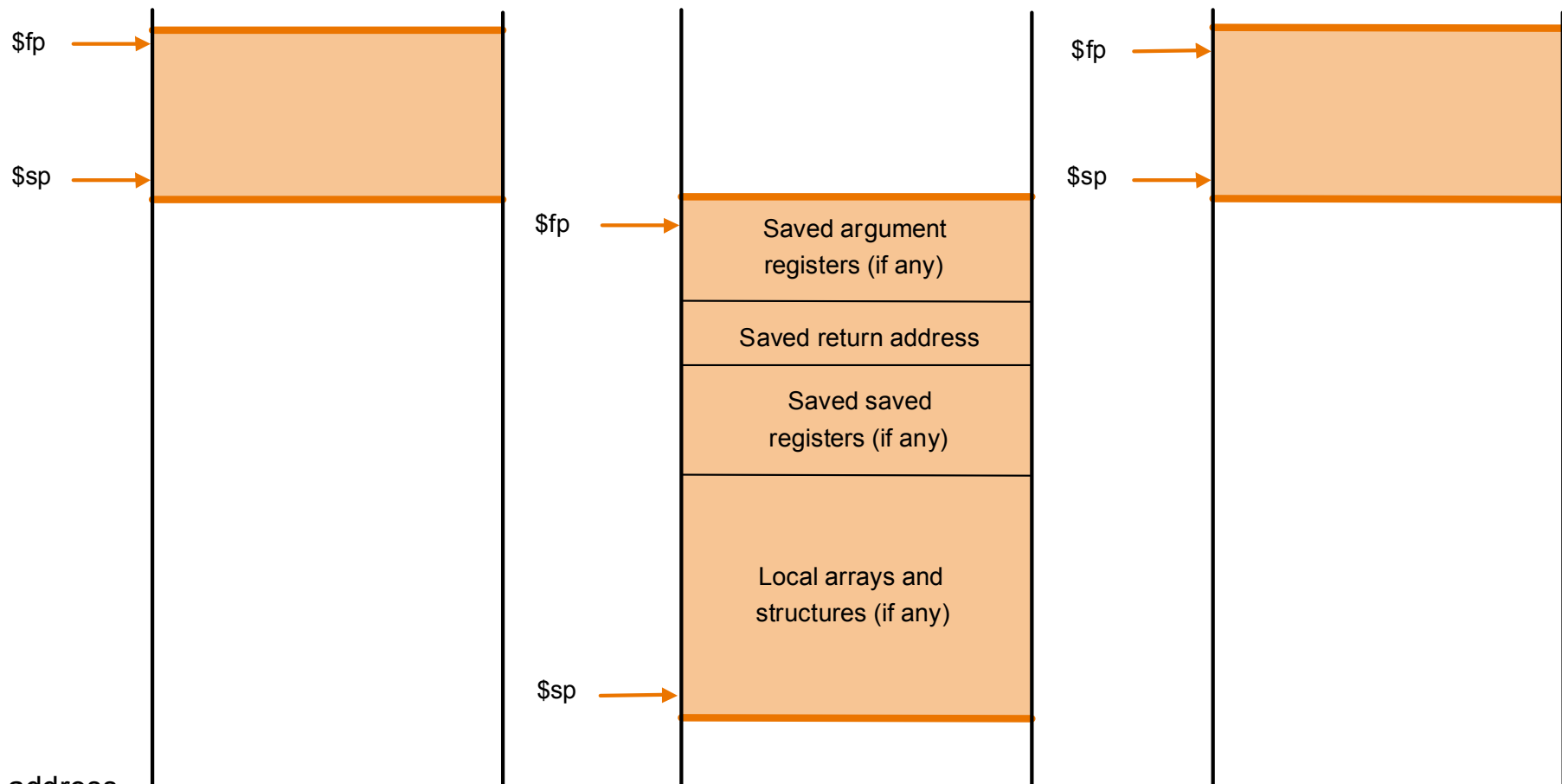
c.

- 因此通过固定的框架指针来访问变量要比用栈指针更简便。

MIPS计算机硬件对过程的支持

○ 地址由高到低

High address



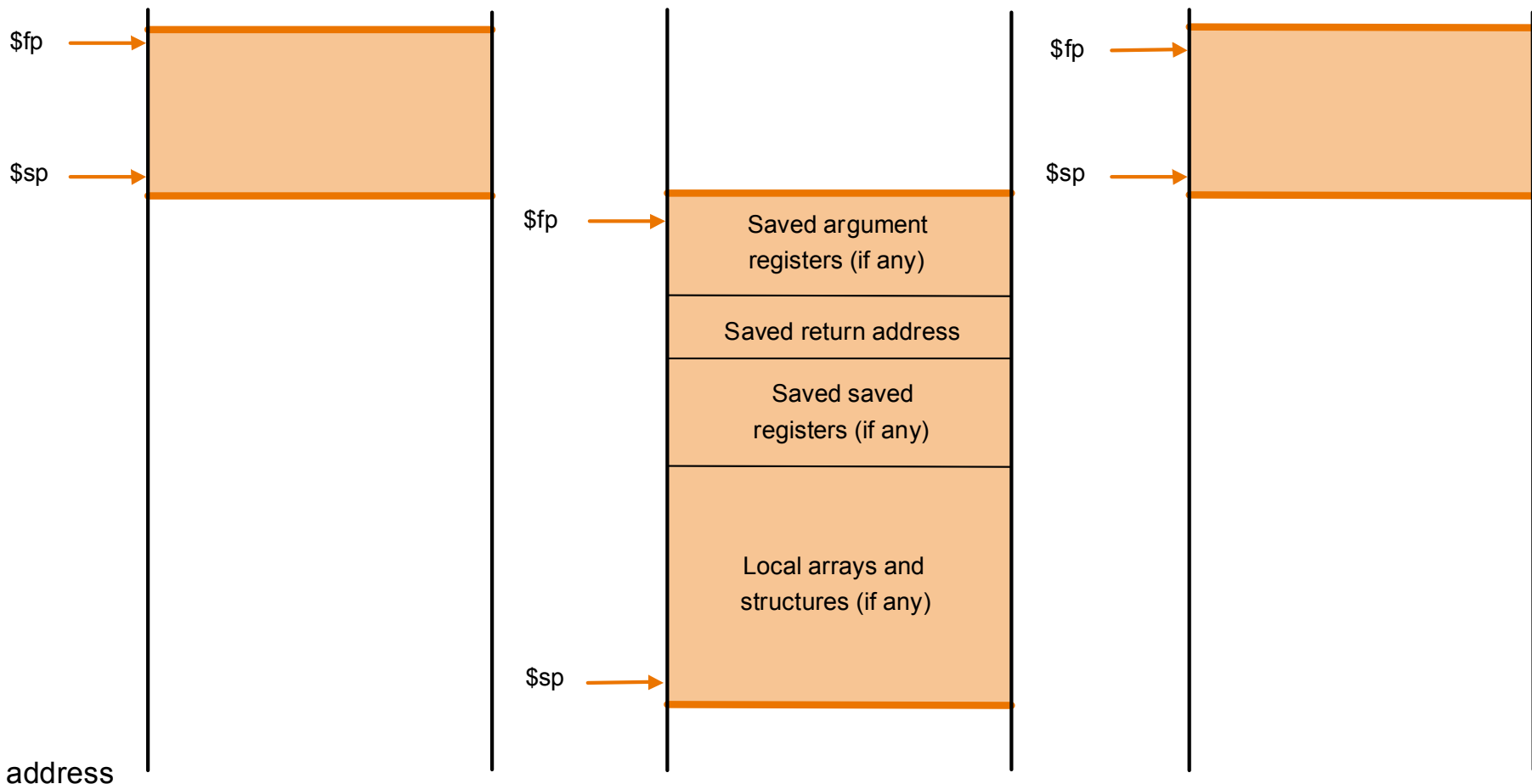
Low address

- 如果一个过程的栈中没有局部变量^a, 编译器将不设置和恢复框架指针^b, 以节省时间^c。

MIPS计算机硬件对过程的支持

○ 地址由高到低

High address



a.

b.

c.

- 当需要框架指针时，以调用时的`$sp`值作为框架指针的初值，调用返回时，根据`$fp`恢复`$sp`值

为新数据分配空间

- 框架指针**\$fp**指向框架的第一个字，通常是保存的参数寄存器；
- 栈指针**\$sp**指向栈顶，在程序执行的过程中栈指针有可能改变；
- 因此通过固定的框架指针来访问变量要比用栈指针更简便。
- 如果一个过程的栈中没有局部变量，编译器将不设置和恢复框架指针，以节省时间。
- 当需要框架指针时，以调用时的**\$sp**值作为框架指针的初值，调用返回时，根据**\$fp**恢复**\$sp**值

\$sp → 7fff ffff hex

MIPS 程序和数据 存储器空间使用约定

- 从顶端开始，对栈指针初始化为 7fffffff，并向下降数据段增长；
- 在底端，程序代码（文本）开始于 00400000；
- 静态数据开始于 10000000；
- 紧接着是由C中malloc进行存储器分配的动态数据，朝堆栈段向上增长

全局指针被设定为易于访问数据的地址，以便使用相对于\$gp的±16位偏移量

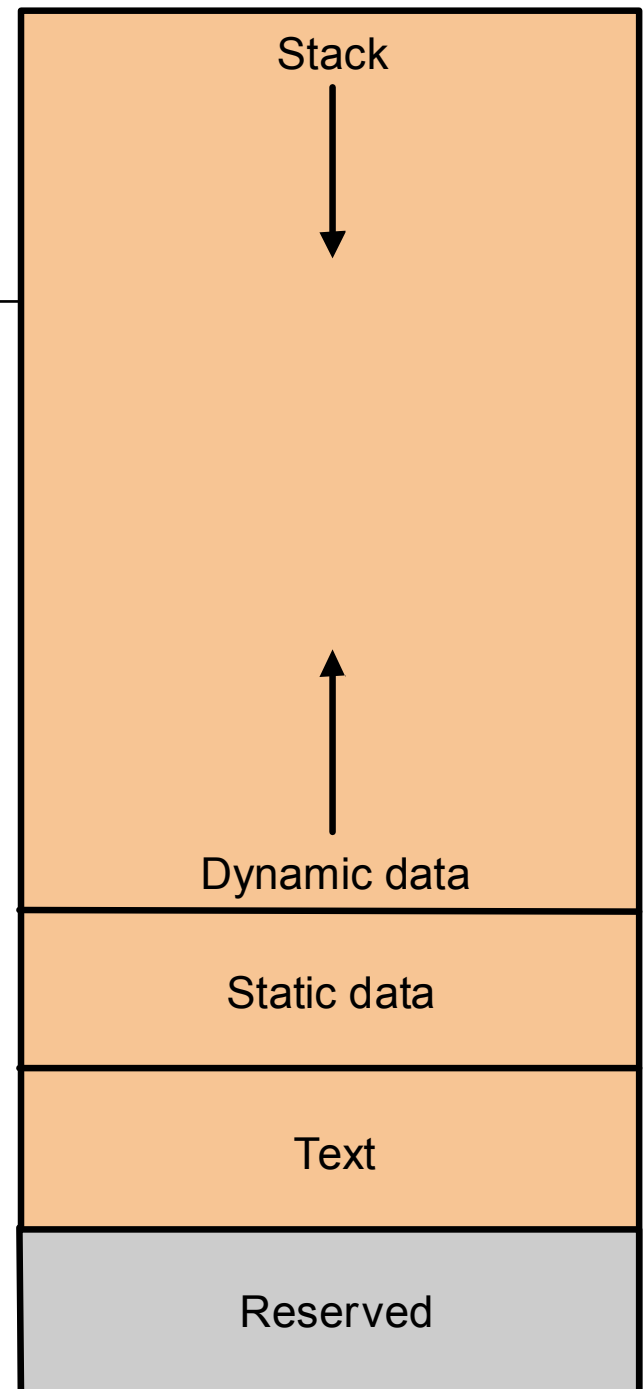
$10000000_{\text{hex}} - 1000ffff_{\text{hex}}$

\$gp → 1000 8000 hex

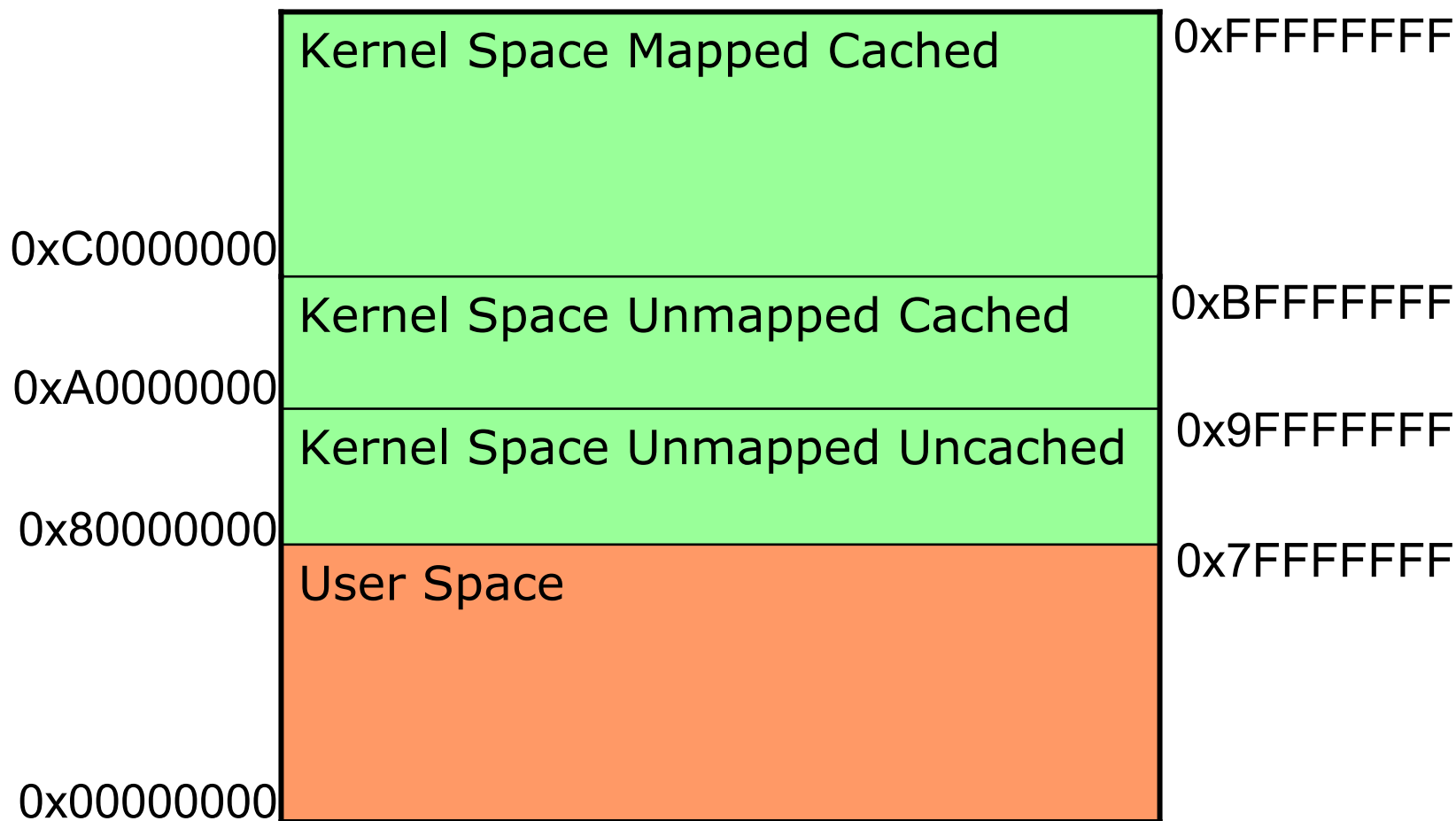
1000 0000 hex

pc → 0040 0000 hex

0



在32位MIPS体系结构下，最多可寻址4GB地址空间。
这4GB空间的分配是怎样的呢？



MIPS Logical Addressing Space

总之: MIPS I之特点

- **32-bit** 固定格式指令 (三种)
- **32个32-bit** 通用寄存器**GPR** (R0 contains zero) and 32个浮点寄存器**FP**(and HI LO)
 - 由软件决定如何使用
- **3元运算指令格式**,包括地址- **reg-reg**等组合.
- 单一的寻址方式(**LD/ST**): 基址加偏移 **base + displacement**
 - 没有间接寻址
- 对**16-bit/32bit** 立即数的支持方式:**LUI**

总之: MIPS I之特点

- 简单的分支条件
 - compare against zero or two registers for $=, \neq$
 - 没有整数条件码
- 延迟分支
 - execute instruction after a branch (or jump) even if the branch is taken

(Compiler can fill a delayed branch with useful work about 50% of the time)

做做看: \$s3=i, \$s4=j, \$s5=@A

```
Loop: addiu  $s4,$s4,1      # j = j + 1
      sll    $t1,$s3,2      # $t1 = 4 * i
      addu   $t1,$t1,$s5    # $t1 = @ A[i]
      lw     $t0,0($t1)     # $t0 = A[i]
      addiu  $s3,$s3,1      # i = i + 1
      slti   $t1,$t0,10     # $t1 = $t0 < 10
      beq    $t1,$0, Loop   # goto Loop
      slti   $t1,$t0, 0     # $t1 = $t0 < 0
      bne    $t1,$0, Loop   # goto Loop
```

```
do j = j + 1
while ( _____ );
```

What C code properly fills in the blank in loop on right?

- 1: `A[i++] >= 10`
- 2: `A[i++] >= 10 | A[i] < 0`
- 3: `A[i] >= 10 || A[i++] < 0`
- 4: `A[i++] >= 10 || A[i] < 0`
- 5: `A[++i] >= 10 && A[i] < 0`
- 6: None of the above

Peer Instruction: \$s3=i, \$s4=j, \$s5=@A

```
Loop: addiu $s4,$s4,1      # j = j + 1
      sll   $t1,$s3,2      # $t1 = 4 * i
      addu  $t1,$t1,$s5    # $t1 = @ A[i]
      lw    $t0,0($t1)     # $t0 = A[i]
      addiu $s3,$s3,1      # i = i + 1
      slti  $t1,$t0,10     # $t1=1  $t0 < 10
      beq   $t1,$0, Loop   # goto Loop if $t1 == 0 ($t0 >= 10)
      slti  $t1,$t0, 0     # $t1=1  $t0 < 0
      bne   $t1,$0, Loop   # goto Loop if $t1 != 0 ($t0 < 0)
```

do j = j + 1
while (_____);

What C code properly fills in the blank in loop on right?

- 1: A[i++] >= 10
- 2: A[i++] >= 10 | A[i] < 0
- 3: A[i++] >= 10 || A[i-1] < 0
- 4: A[i++] >= 10 || A[i] < 0
- 5: A[++i] >= 10 && A[i] < 0

None of the above

Let's Play "Compiler" !

规则:

变量在主存储器中

运算在**ALU**中完成

寄存器中放临时的运算结果或者变量

语句:

$z = (x + y) * (x - y) ;$

lw *\$r1, x(\$0)*

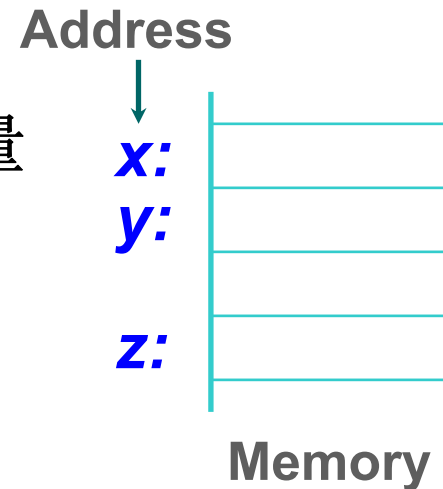
lw *\$r2, y(\$0)*

add *\$r3, \$r1, \$r2*

sub *\$r4, \$r1, \$r2*

mul *\$r5, \$r3, \$r4*

sw *\$r5, z(\$0)*



Statement:

variable = expression ;

code for
expression

→ ***rz***

sw \$rz, var_add(\$0)

expression:

exp1 OP exp2

code for
exp1

→ ***rx***

code for
exp2

→ ***ry***

语句块

Block:

```
{ statement 1 ;  
  statement 2 ;  
  .  
  .  
  statement k ; }
```

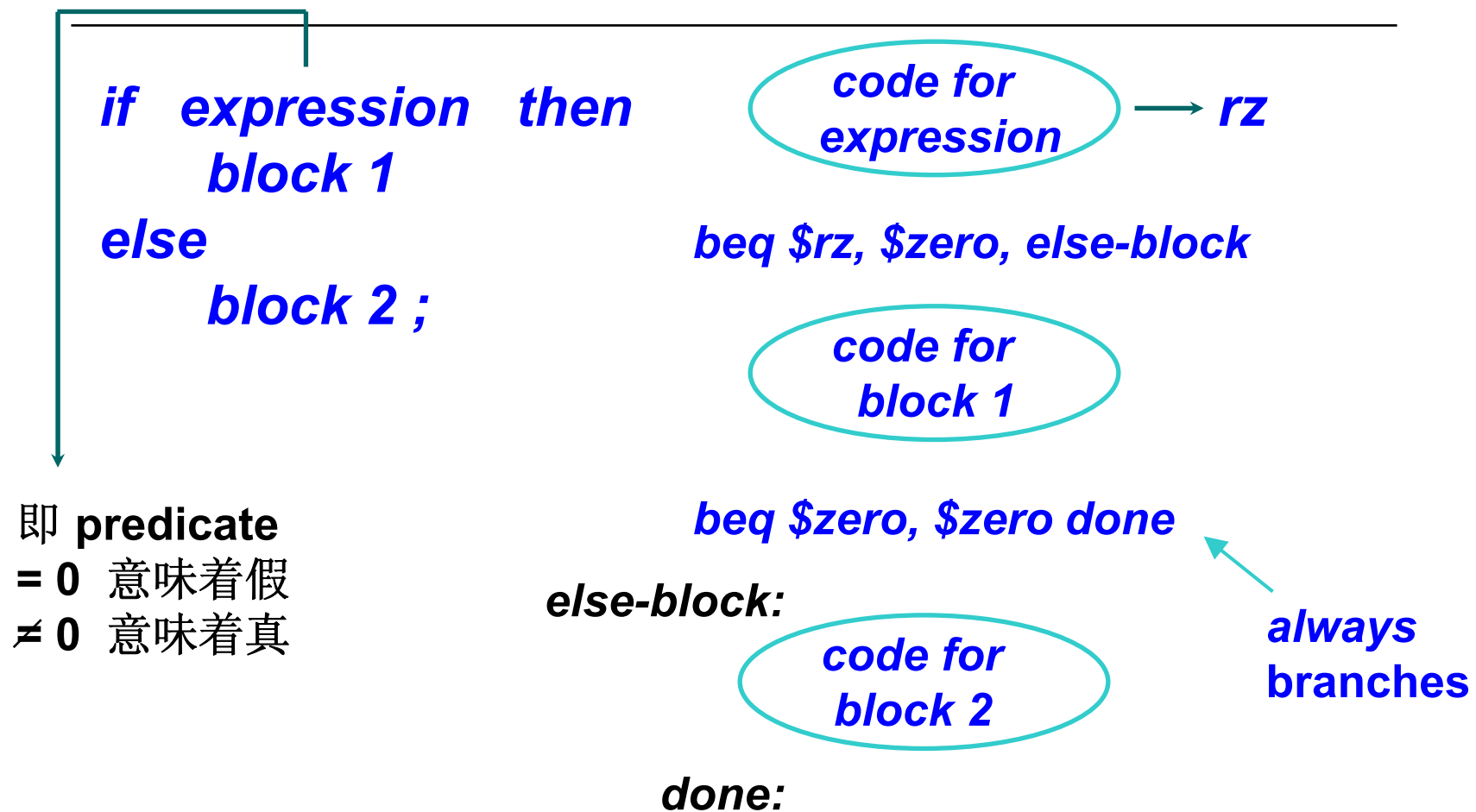
**code for
statement 1**

**code for
statement 2**

**.
.**

**code for
statement k**

条件语句



循环

while 表达式 **do**
代码块;

else-代码块:

表达式的
代码

→ **rz**

beq \$rz, \$zero, done

代码块的
汇编代码

beq \$zero, \$zero, else-block

done:

MIPS汇编语言

- 绝大多数的MIPS汇编语言晦涩难懂，充满了各种寄存器号；
- 可以使用宏处理语言来允许程序员用寄存器名（反映了每个寄存器的习惯用法）代替寄存器号；
- 大多数采用C预处理器，去掉C风格的注释，因此可以在汇编代码里使用这种注释：

```
/* this is a comment*/  
# so is this  
Entrypoint:           # that's a label  
    addu $1, $2, $3    #(register $1=$2+$3)
```

- 注意：目标寄存器在左边

MIPS汇编语言

- MIPS公司的汇编器会合成一些指令
- 严格的编译器应该产生和机器代码一一对应的指令，合成指令是用来帮助程序员的。
 - 一个32位的立即数加载：可以在代码中加载任何数据，汇编器会将其拆开成为两个指令，分别加载在这个数据的前半部分和后半部分
- 从一个内存地址加载：可以加载一个内存变量。汇编器通常会将这个变量地址的高位放入一个暂时的寄存器中，然后将这变量的低位作为一个加载的偏移量。

`li $s1, 0x12345678`

`La $s1, address`

MIPS 通用寄存器的使用

- MIPS体系结构下，程序计数器（**PC**）不是一个寄存器，在一个具有流水线的CPU中，程序计数器的值在一个给定的时刻有多个可选值，但可见的只有一个。

...

jal printf

move \$4, \$6

xxx # return here after call

jal指令的返回地址跟随其后的第二条指令，因为紧跟踪jal指令后面的指令在delay slot(延迟位置)上

- 注意：关于延迟位置的规则是该指令在转移目标(如上述的printf)之前执行。延迟位置指令经常被用来传递函数调用的参数。