第二章

指令：计算机的语言

概要

- 指令集简介
- 硬件操作—算术运算
- 操作数
- 数的表示
- 指令表示
- 逻辑操作
- 决策指令
- 硬件与过程
- 人机交互
- 寻址
- 同步
- 程序执行

指令集简介—指令集

- 一个计算机的指令表
- 不同的计算机架构具有不同的指令集。
  - 大部分的地方都相似。
- 早期计算机具有非常简单的指令集。
  - 简便实现
- 许多现代计算机也是具有简单指令集

指令集简介—MIPS 指令集

- 本书使用MIPS作为例子
- 斯坦福大学MIPS被MIPS科技公司商业化
- 在嵌入式芯片市场占有相当大的市场份额
  - 应用在包括用户电子，网络/存储设备，相机，打印机等等
- 许多现代的ISA的范例
  - ARMv7/X86/ARMv8
  - 实验使用WinMIPS64
  - 参见图2-1、封2

硬件操作—算术运算

- 加减运算,三个操作数
  - 两个源操作数和一个目的操作数

```
add a, b, c   # a gets b + c
```

- 所有的算术运算具有这个形式。
- 设计原则1：简单源于规整（Simplicity favours regularity）
  - 规范化使得实现简单，简单使得高性能变得低成本。

硬件操作—算术示例

- 例题 C code:

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

- 编译器产生 MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

操作数—寄存器操作

- MIPS 有 32 × 32-bit 寄存器文件
  - 用于频繁读取数据
  - 序号为0-31
  - 32位的数据称为“word”

寄存器名字	*寄存器编号*	*寄存器功能*
$zero	***$0***	*恒等于零*
$at	***$1***	*被汇编器保留，用于处理大的常数*
$v0 – $v1	***$2-$3***	*存放函数返回值*
$a0 – $a3	***$4-$7***	*传递函数参数*
$t0 – $t7	***$8-$15***	*存放临时变量*
$s0 – $s7	***$16-$23***	*存放需要保存的临时值*
$t8 – $t9	***$24-$25***	*额外的存放临时变量*
$k0 – $k1	***$26-$27***	*用于操作系统内核*
$gp	***$28***	*指向全局变量的指针*
$sp	***$29***	*指向栈顶的指针*
$fp	***$30***	*指向栈帧的指针*
$ra	***$31***	*返回地址，用于函数调用*

操作数—寄存器操作

- 算术运算指令只使用寄存器操作
- *设计原则 2:* Smaller is faster
  - c.f. 主存: 几十个亿的地址

操作数—寄存器操作示例

- C code:

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

  - f, …, j in $s0, …, $s4

- 编译后的 MIPS code（对比第5页的g/h/i/j）：

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

操作数—内存操作

- 主存用于组合数据
  - 数组，结构体，动态数据
- 实现算术运算时
  - 从内存往寄存器装载数据
  - 从寄存器取数据保存到内存
- 内存是以byte表示的
  - 每个地址都是以8位的地址符表示的
- words是在内存中对齐的
  - 地址必须是4的倍数
- MIPS 使用大端方式（Big-Endian）
  - 非常重要的字节至少用一个word来表示。

Little-Endian: least-significant byte at least address

高字节存储在低地址中
低字节存储在高地址中

高字节存储在高地址中
低字节存储在低地址中

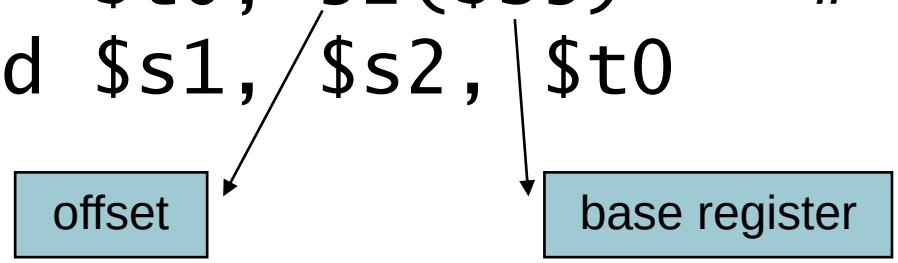
操作数—内存操作示例1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - 下标 8 对应字节偏移量 32
    - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```



操作数—内存操作示例2

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, 数组A起始地址在 $s3

- 编译产生的 MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

操作数—寄存器 vs.内存

- 操作内存数据需要加载和存入操作
  - 需要执行更多指令
- 寄存器比内存要更快地存取数据
- 编译器需要尽可能多的使用寄存器变量
  - 仅将不怎么频繁使用的变量放入内存。
  - 寄存器优化（尽可能多地使用）是非常重要的。

操作数—立即数操作

- 常量/立即数在一个指令中表明

```
addi $s3, $s3, 4
```

- 没有立即数的减法指令，只能用负数相加

```
addi $s2, $s1, -1
```

- *设计原则 3:* Make the common case fast
  - 小的常量是很常见的
  - 立即数操作可以避免使用装载指令

操作数—常量0

- MIPS 寄存器 0 ($zero) 是表示常量0
  - 不能重写寄存器0
- 对相同的操作使用
  - E.g., 寄存器之间移动

```
add $t2, $s1, $zero
```

数的表示—无符号二进制整数

- 给一个n位数

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围: 0 到 $+2^n - 1$
- 举例

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

$= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

$= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

使用32位

0 to +4,294,967,295

数的表示—2进制补码有符号整数

- 给一个n位数

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- 范围: -2^{n-1} to $+2^{n-1} - 1$
- 举例

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

$= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

使用32位

–2,147,483,648 to +2,147,483,647

数的表示— 2进制补码有符号整数

- 第31位是符号位
  - 1 表示负数
  - 0 表示非负数
- $-(-2^{n-1})$ 不能够被表示
- 非负数具有相同的非负的二进制补码的表示形式。
- 一些特殊数
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - 最小负整数: 1000 0000 … 0000 → -2^{31}
  - 最大正整数: 0111 1111 … 1111 → $+2^{31}-1$

数的表示—有符号反码

- 补码和加 1
  - 补码是表示为 1 → 0, 0 → 1

    正数 X，–X的反码为：$\bar{X}$

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- 举例: 反码 +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

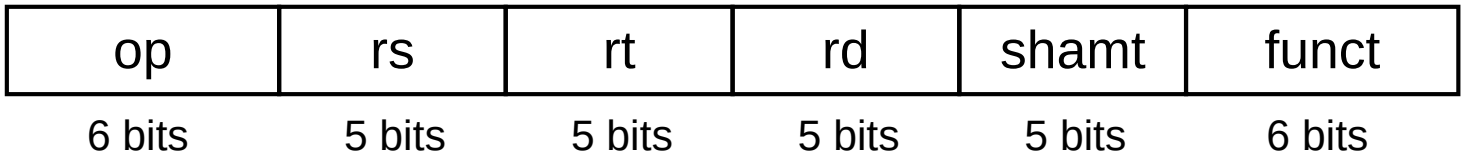
数的表示—符号扩展

- 使用更多位来表示一个数字
  - 数值保持不变
- 在MIPS指令集中
  - addi: 扩展立即数
  - lb, lh: 扩展到取字节/半字
  - beq, bne: 扩展位移
- 重复符号位在左边
  - c.f. 无符号值：扩展位是0
- 举例: 8-位 to 16-位
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

指令表示—综述

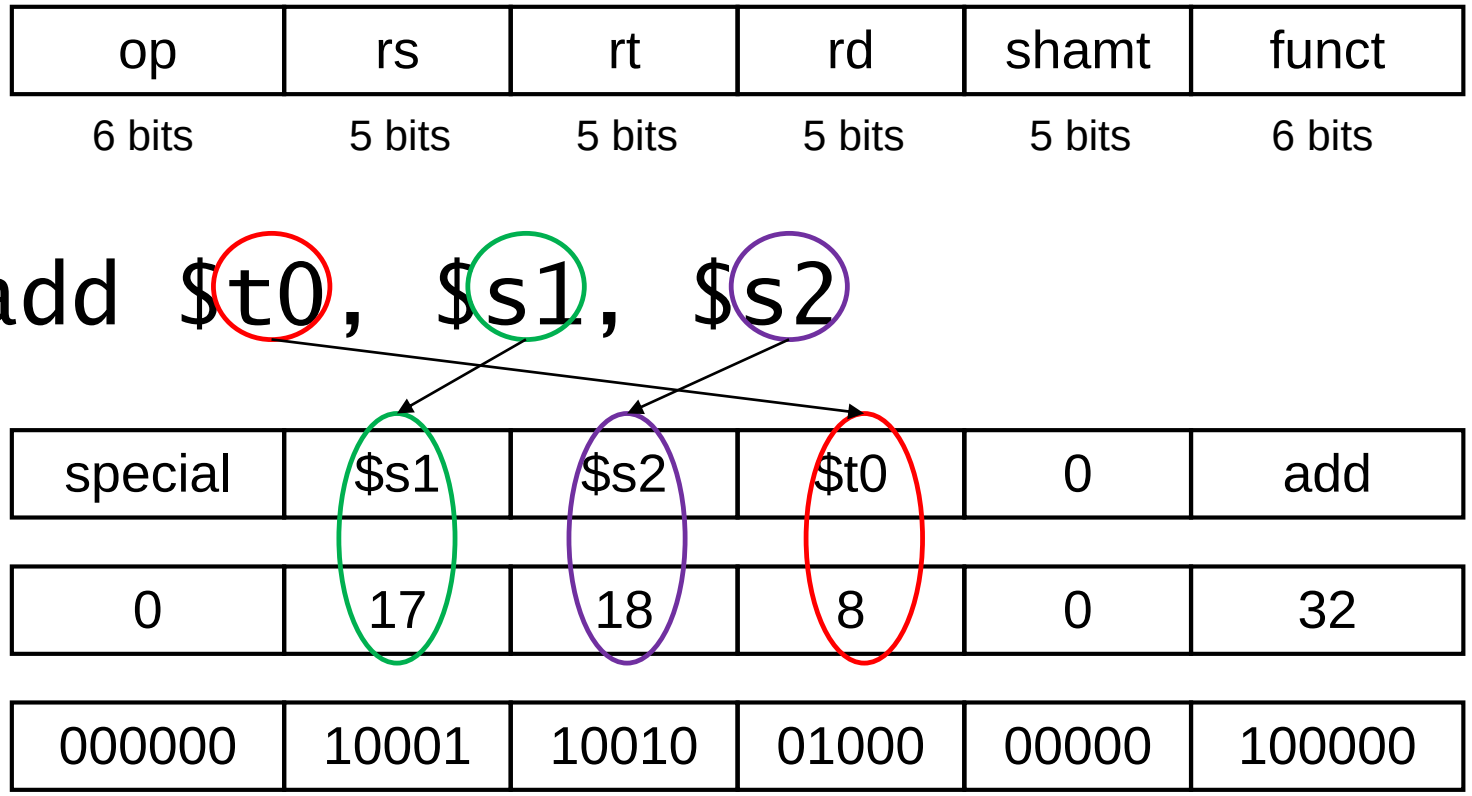
- 指令用二进制编码
  - 称为机器代码
- MIPS 指令
  - 每条指令编码为 32位指令字
  - 用较小的数值编码为操作代码（opcode）、寄存号
  - 规整化
  - 寄存器编号
    - $t0 – $t7 是 reg’s 8 – 15
    - $t8 – $t9 是 reg’s 24 – 25
    - $s0 – $s7 是 reg’s 16 – 23
    - 其他将在用到时介绍

指令表示— MIPS R-型指令



- 运算指令：算术/逻辑
- 指令字段
  - op: 操作码(opcode)
  - rs: 第一个源寄存器编号
  - rt: 第二个源寄存器编号
  - rd: destination 目的寄存器编号
  - shamt: 移位位数(00000 表示不移位)
  - funct: 功能码(扩展操作码extends opcode)

指令表示— R-format Example



$$00000010001100100100000000100000_2 = 02324020_{16}$$

指令表示— 十六进制

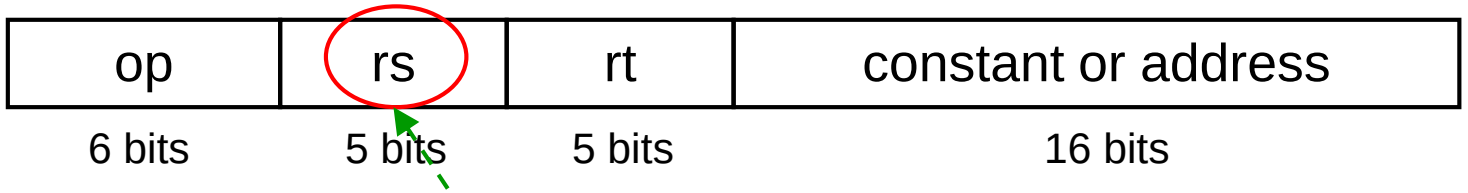
- 基底是 16
  - 二进制串的压缩表示（使用16进制的原因）
  - 四位二进制组成一个16进制数

0	***0000***	***4***	***0100***	***8***	***1000***	***c***	***1100***
1	***0001***	***5***	***0101***	***9***	***1001***	***d***	***1101***
2	***0010***	***6***	***0110***	***a***	***1010***	***e***	***1110***
3	***0011***	***7***	***0111***	***b***	***1011***	***f***	***1111***

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

指令表示— MIPS I-型 指令

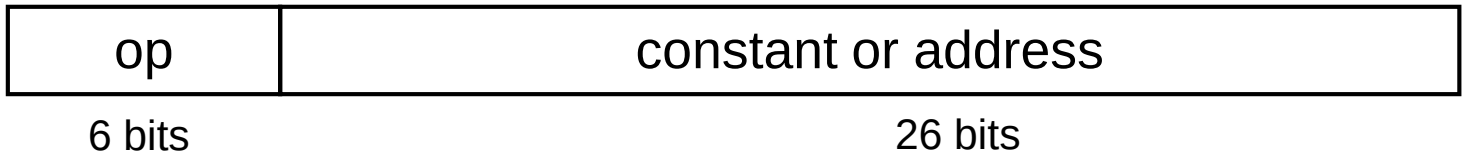
对应R型指令的rd/shamt/funct字段



- 立即数算术和读数/存数指令（操作数地址）
  - rt: 目的或源寄存器编号
  - 常量的取值范围: -2^{15} to $+2^{15} - 1$
  - Address: 偏移加上在rs中的基址
- *设计原则3:* 优秀的设计需要适当的折中方案
  - 不同的类型指令采用不同的解码方式，但是都是32位的统一指令长度
  - 尽可能保持格式相似，允许不同

指令表示— MIPS J-型 指令

对应R型指令的rd/shamt/funct字段



- 跳转指令（操作数地址）
  - 常量的取值范围: -2^{26} to $+2^{26} - 1$
  - Address可以由寄存器给出

指令表示— 汇总

MIPS指令（add/sub/addi/lw/sw）格式汇总-1

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	
add immediate	I	8_{ten}	reg	reg				constant
lw (load word)	I	35_{ten}	reg	reg				address
sw (store word)	I	43_{ten}	reg	reg				address

指令表示— 汇总

MIPS指令（add/sub/addi/lw/sw）格式汇总（含指令示例）

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

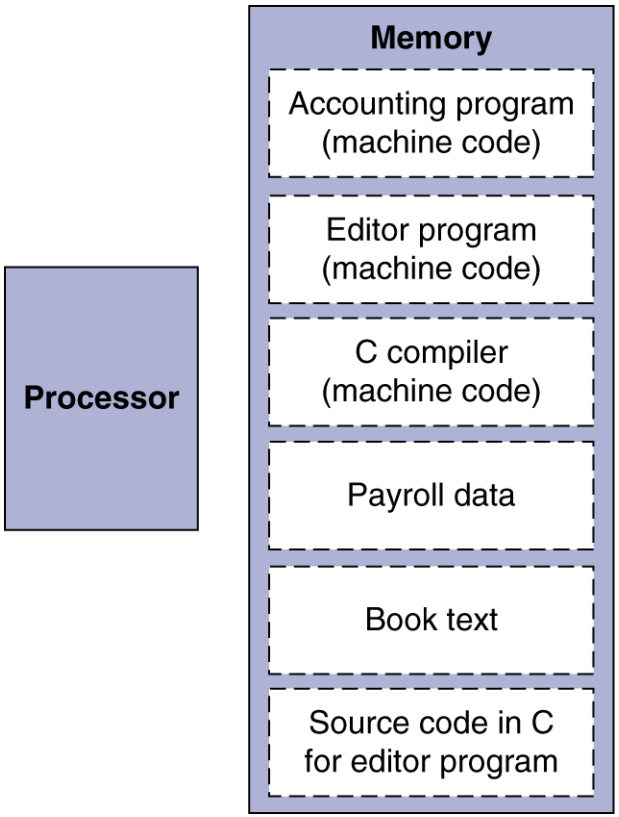
例题 P56

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

```
1. sub $t0, $t1, $t2
2. add $t2, $t0, $t1
3. sub $t2, $t1, $t0
4. sub $t2, $t0, $t1
```

指令表示— 计算机中存储

The BIG Picture



- 指令/数据都被表示为二进制
- 指令和数据被存放在内存中
- 程序能在程序中处理
  - e.g., 编译器，链接器, …
- 二进制通用性允许编译程序在不同的计算平台上运行
  - 标准化的ISAs

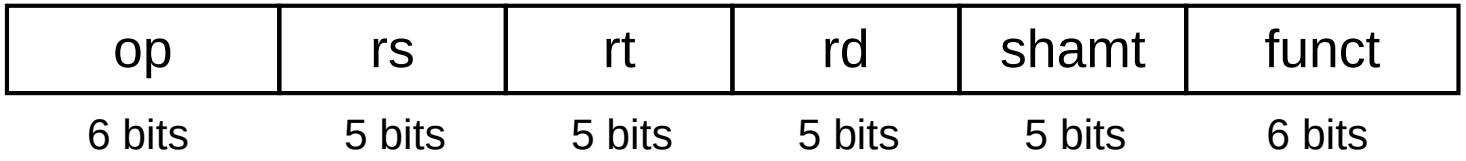
逻辑操作—综述

- 对位进行处理的指令

逻辑操作	C	Java	MIPS
左移	<<	<<	sll
右移	>>	>>>	srl
按位与	&	&	and, andi
按位或	|	|	or, ori
按位取反	~	~	nor

- 用于对字中的若干“位”打包和拆包的操作

逻辑操作—移位操作



- shamt: 移多少位
- 逻辑左移sll
  - 左移空位填0
  - 逻辑左移i位相当于乘 2^i
- 逻辑右移srl
  - 逻辑右移空位填0
  - 逻辑右移i位相当于除 2^i (仅对无符号数)

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

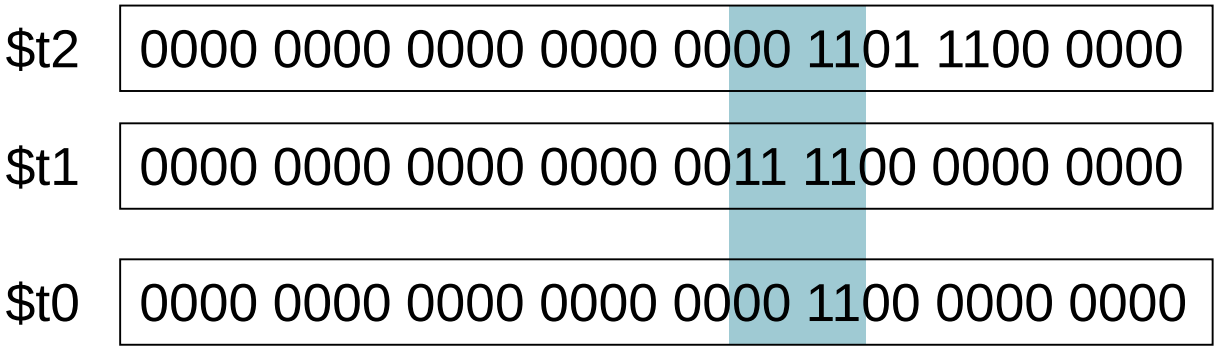
逻辑操作—例子

- 逻辑运算举例
- `and $t0, $t1, $t2`
- `or  $t0, $t1, $t2`
- `nor $t0, $t1, $zero`

逻辑操作—与操作

- 可用于一个字中的掩码操作
  - Select 选择某些位，其他位清零

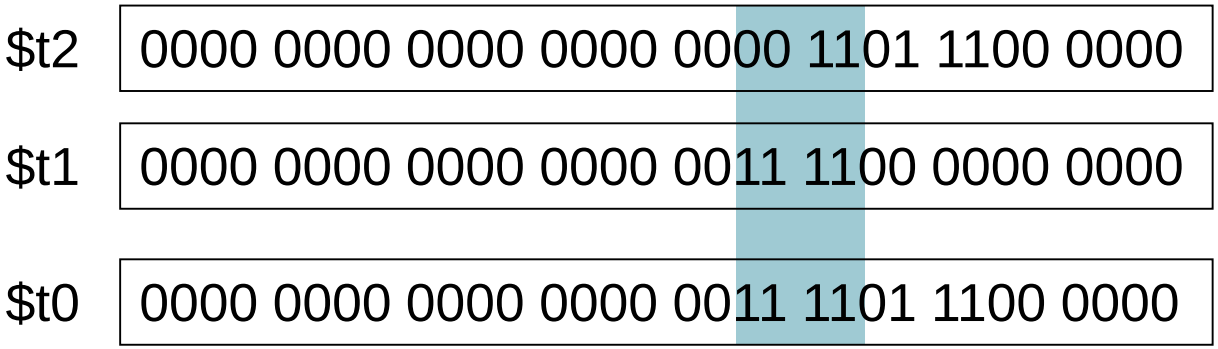
```
and $t0, $t1, $t2
```



逻辑操作—或操作

- 用于把包含字中的一些位置1，其他不变

```
or $t0, $t1, $t2
```



逻辑操作—按位取反操作

- 用于改变字中的一些位
  - 0 变成 1, 1 变成 0
- MIPS 3-操作数指令 NOR
  - a NOR b == NOT (a OR b) “或非”

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000	0000	0000	0000	0011	1100	0000	0000
$t0	1111	1111	1111	1111	1100	0011	1111	1111

决策指令—综述

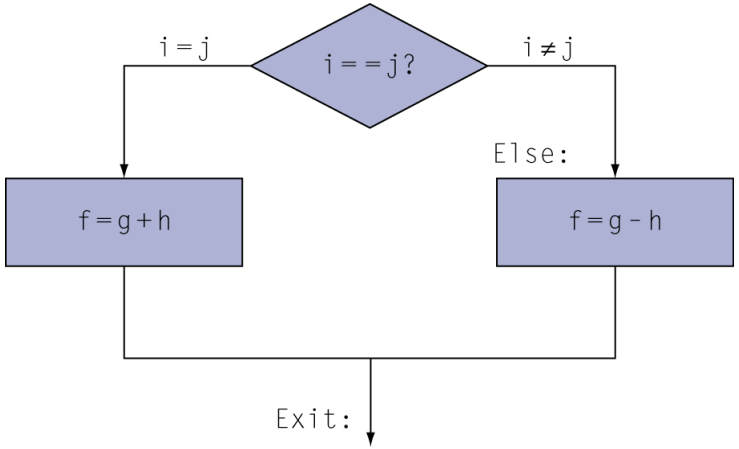
- 如果条件为真，跳转到被标签的指令执行
  - 否则，继续执行
- `beq rs, rt, L1`
  - if (rs == rt) 转到标签为L1的指令执行;
- `bne rs, rt, L1`
  - if (rs != rt)转到标签为L1的指令执行;
- `j L1`
  - 无条件跳转到标签为L1的指令执行

决策指令—编译IF语句

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Assembler calculates addresses

决策指令—编译循环语句

- C code:

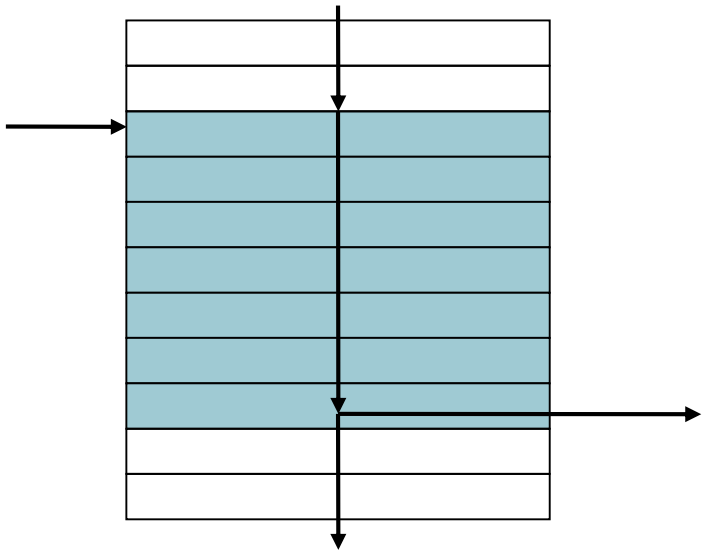
```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

决策指令—基本块

- A 一个基本块就是一个指令序列，其中
  - 内部没有跳出的指令（结束指令除外）
  - 也没有被跳转到的指令(开始指令除外）



- 编译器标识基本块用于优化
- 高级处理机能够加速基本块的执行

决策指令—更多的条件操作

- Set如果条件为真置1，否则置0
- `slt rd, rs, rt` 小于则置位
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- beq, bne可以和其他指令结合使用

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

?

```
if (i>j) f = i;
else f = j;
```

决策指令—分支指令设计

- beq、bne、slt 实现所有判断
- 为什么没有blt, bge, 等指令?
- 硬件执行<, ≥, ... 比 =, ≠慢
  - 指令中结合分支指令包含更多工作，需要更慢的时钟
  - 所有的指令都受到影响!
- slt与beq/bne结合实现跳转
- case/switch：转移表（jump table）

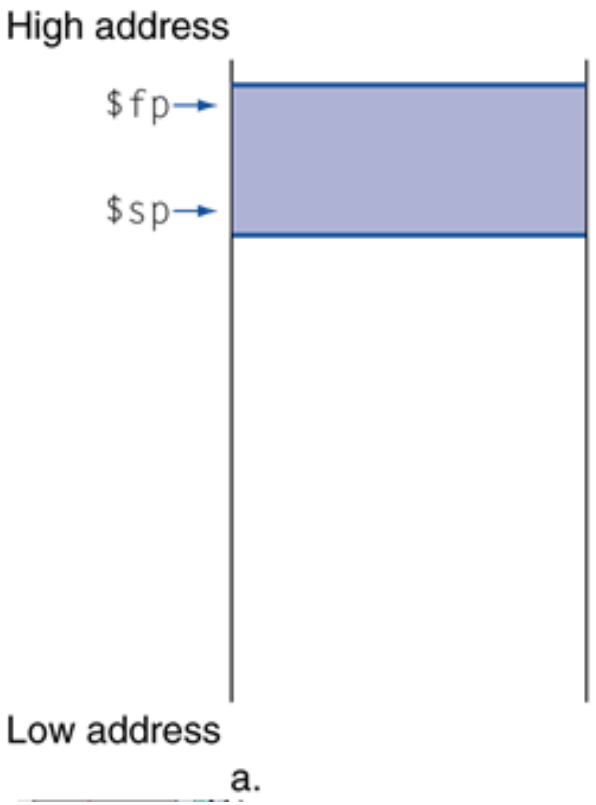
决策指令—有符号数vs.无符号数

- 有符号数比较: `slt`, `slti`
- 无符号数比较: `sltu`, `sltui`
- `slt  $t0, $s0, $s1 #if($s0<$s1)` $t0 = 1
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001

  - `slt  $t0, $s0, $s1  # signed,`–1 < +1
  - `sltu $t0, $s0, $s1  # unsigned,` 4,294,967,295 > 1 ⇒ $t0 = 0
  - `sltui $t0, $s0, 2 # unsigned,` 4,294,967,295 > 2 ⇒ $t0 = 0

硬件与过程—简述

- 原则
  1. 保留现场：地址和资源
  2. 获得信息：地址和资源
  3. 参数传递



stack 堆栈

$fp 帧指针

$sp 栈指针

$gp 全局指针

硬件与过程—简述

- 遵循步骤
  1. 将参数放在过程可以访问的寄存器里
  2. 将控制权转移给过程
  3. 获得过程所需要的存储资源
  4. 执行过程的操作
  5. 将结果放在调用程序可以访问到的寄存器
  6. 将控制权返回到调用点

硬件与过程— Register Usage

- $a0 – $a3: 传递参数(reg's 4 – 7)
- $v0, $v1: 返回结果值(reg's 2 and 3)
- $t0 – $t9: 临时寄存器(reg's 8 – 15，24-25)
  - 可以被调用者改写
- $s0 – $s7: 保存参数(reg's 16 – 23)
  - 必须被调用者（过程）保存和恢复
- $gp: 静态数据的全局指针寄存器 (reg 28)
- $sp: 堆栈指针寄存器(reg 29)
- $fp: 帧指针寄存器-保存过程帧的第一个字(reg 30)
- $ra: 返回地址寄存器 (reg 31)

硬件与过程—过程调用指令

- 过程调用：跳转和链接

  `jal ProcedureLabel`

  - 下一条指令的地址在寄存器 $ra 中（自动）
  - 跳转到目标地址
- 过程返回:寄存器跳转

  `jr $ra`

  - 复制 $ra 到程序计数器
  - 也可以用于运算后跳转
    - e.g., case/switch 语句

硬件与过程—Example-LEAF

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - 参数 g, …, j 在 $a0, …, $a3中
  - f in $s0 (因此, 需要存储$s0 到堆栈)
  - 结果在$v0

硬件与过程—非嵌套过程的例子

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

嵌套过程-非叶过程NON-LEAF

- 过程调用其它过程
- 对于嵌套调用，调用者需要存储到堆栈的信息：
  - 它的返回地址
  - 调用后还需要用的任何参数寄存器和临时寄存器
- 调用后返回，寄存器会从堆栈中恢复

Non-Leaf Procedure Example

- C code:

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

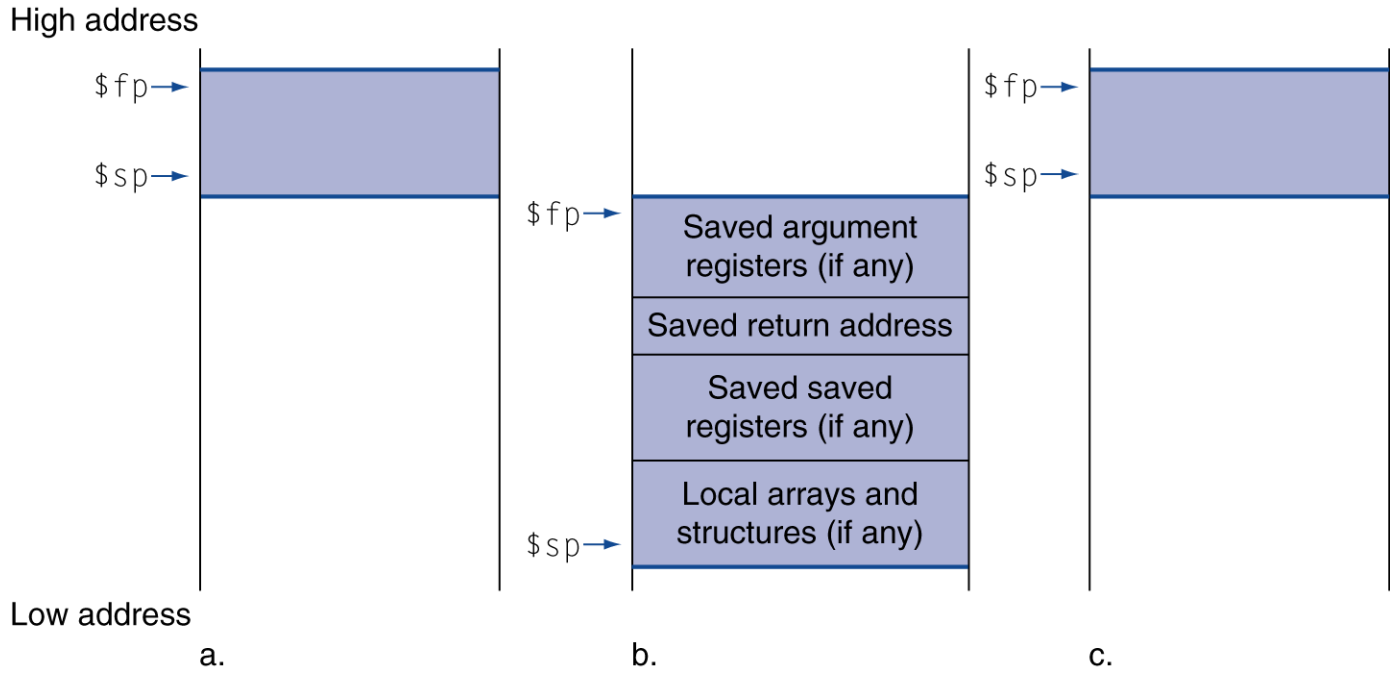
  - 参数 n 放在 $a0中
  - 结果放在 $v0

Non-Leaf Procedure Example

- MIPS code:

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```

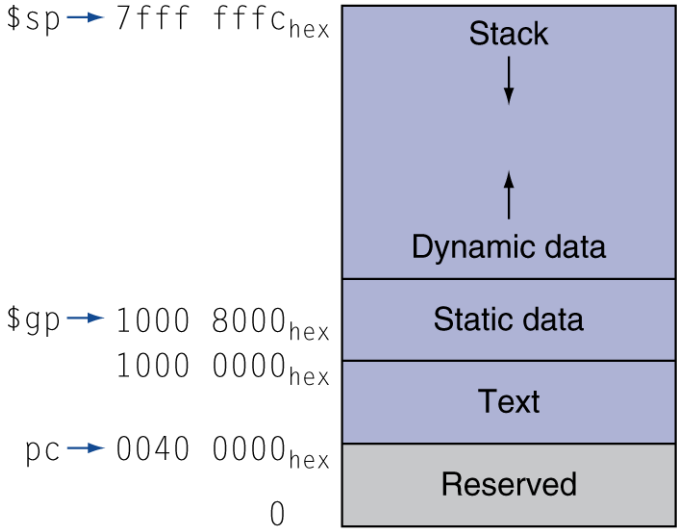
硬件与过程—堆栈中的局部数据



- 局部数据由调用者分配
  - e.g., C 自动分配变量
- 过程帧(活动记录)
  - 被一些编译器使用控制堆栈存储

硬件与过程—内存布局

- 正文：程序代码
- 静态数据：全局变量
  - e.g., C语言静态变量，常数数组和字符串
  - $gp 初始化地址，允许段内的 ±偏移
- 动态数据：堆
- E.g., malloc 函数C, new函数 Java
- 堆栈：自动存储



2^8→256
2^10→1K
2^20→1M

MIPS寄存器约定

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

人机交互—字符数据

- 字节编码的字符集
- ASCII: 128 个
  - 95 graphic, 33 control
  - Latin-1: 256 个
    - ASCII, +96 more graphic characters
- Unicode: 32位字符集
  - Used in Java, C++ wide characters, ...
  - 宽字符世界上大多数字母表
  - UTF-8, UTF-16: 可变长度编码

人机交互—字节/半字操作

- 使用位操作
- MIPS字节/半字 读取/存储
- 字符串处理是较常用的方式

lb rt, offset(rs)　　lh rt, offset(rs)

  - Sign extend to 32 bits in rt

lbu rt, offset(rs)　　lhu rt, offset(rs)

  - Zero extend to 32 bits in rt

sb rt, offset(rs)　　sh rt, offset(rs)

  - 存储恰恰是字节或半子的形式

人机交互—字符串拷贝举例

- C code (naïve):
  - 非终止字符串Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!= '\0' )
    i += 1;
}
```

  - 假定x, y 基地址在 $a0, $a1中;
  - i 在 $s0 中

人机交互—字符串拷贝

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

32-bit 立即数操作

- `lui rt, constant` #将立即数放到rt高16位
- 大部分常数都比较小，16位表示足够
- 完整32位结合 `ori` 指令

```
lui $s0, 61
```

0000 0000 0011 1101 | 0000 0000 0000 0000

```
ori $s0, $s0, 2304
```

0000 0000 0111 1101 | 0000 1001 0000 0000

寻址—地址模式总结

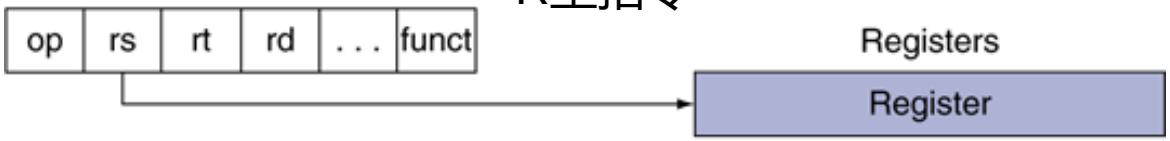
1. Immediate addressing

I型指令，立即数

addi $s1, $s2, 4

2. Register addressing

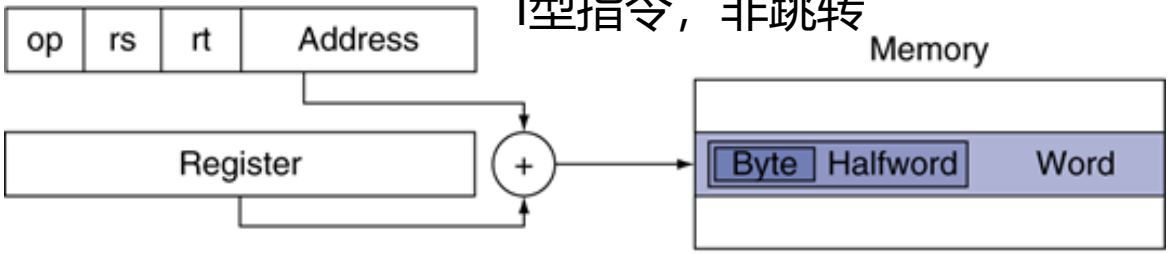
R型指令



add $s1, $s2, $s3

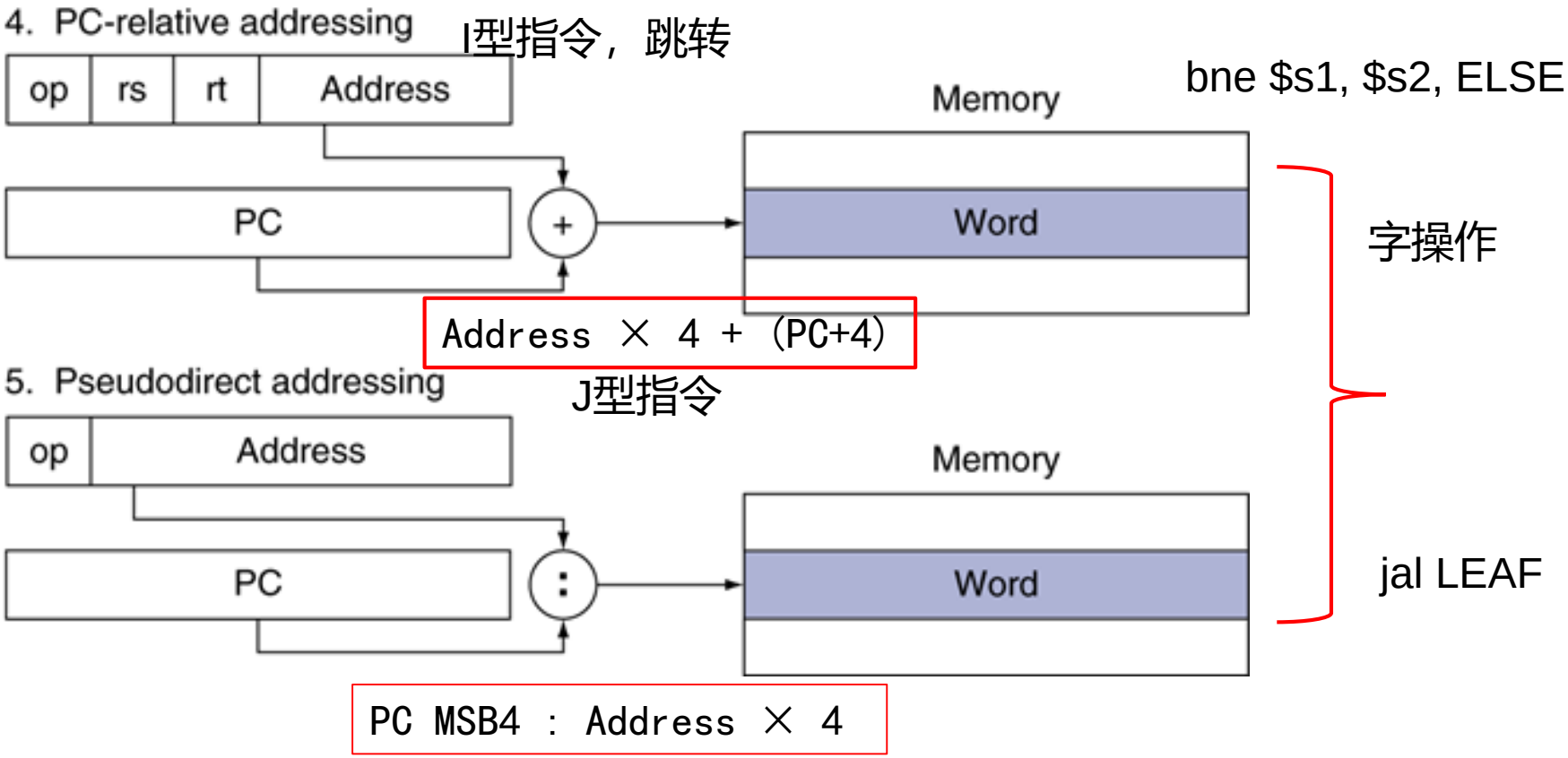
3. Base addressing

I型指令，非跳转



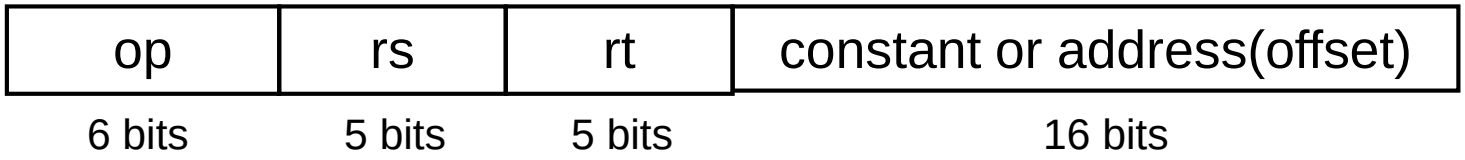
lw $s1, 40($s2)

寻址—地址模式总结


4. PC-relative addressing
I型指令，跳转
bne $s1, $s2, ELSE
op
rs
rt
Address
Memory
PC
+
Word
字操作
Address × 4 + (PC+4)
5. Pseudodirect addressing
J型指令
op
Address
Memory
PC
:
Word
jal LEAF
PC MSB4 : Address × 4

寻址—分支地址

- 分支指令说明
- 操作码，两个寄存器，两个地址
- 大多数跳转目标离跳出的位置较近
- 向前或向后



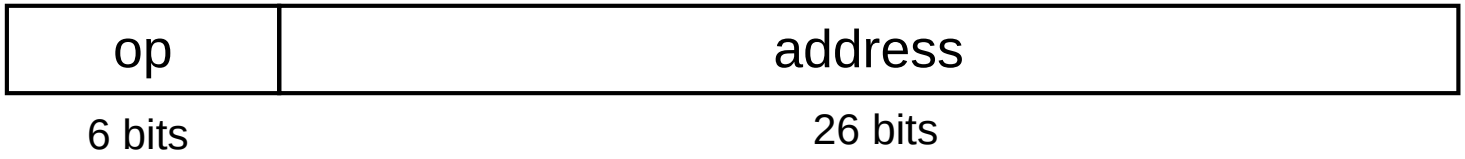
- PC相对寻址
  - 目标地址 = （PC+4）+ offset × 4
  - 此时PC的增加量是4的倍数

下一条指令

+/- 128K

寻址—跳转地址

- 跳转（j和jal）的目标地址可以在代码段的任何位置
  - 指令除op外，指令其它字段都是地址



- 直接跳转到地址
- Target address = $PC_{31\ldots28}$: (address × 4)

4位

28位

256M地址块

寻址—目标地址举例

- 早期例子的循环代码
  - 设循环的起始地址是80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

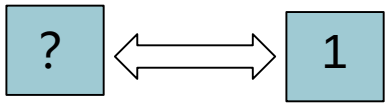
寻址—远程分支

- 如果跳转对象地址太大无法用16位的偏移表示，汇编将重写代码
- 【把短跳转（2^{16} 范围）变成长跳转（2^{26} 范围）】
- Example

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

同步—并行与指令：同步

- 处理器共享存储器同一区域
- P1 写，P2 读
- （任务1写的结果是任务2要读取得值）如果P1和P2不同步，将发生数据竞争
- 结果由访问次序决定
- 依赖硬件提供同步指令
- 原子读/写内存操作
- 在读和写之间，不再允许对该空间的其他操作
- 可以是单一的指令
- 例如寄存器和内存之间的原子交换
- 或者指令的原子配对



0：√

1：×

信号量控制

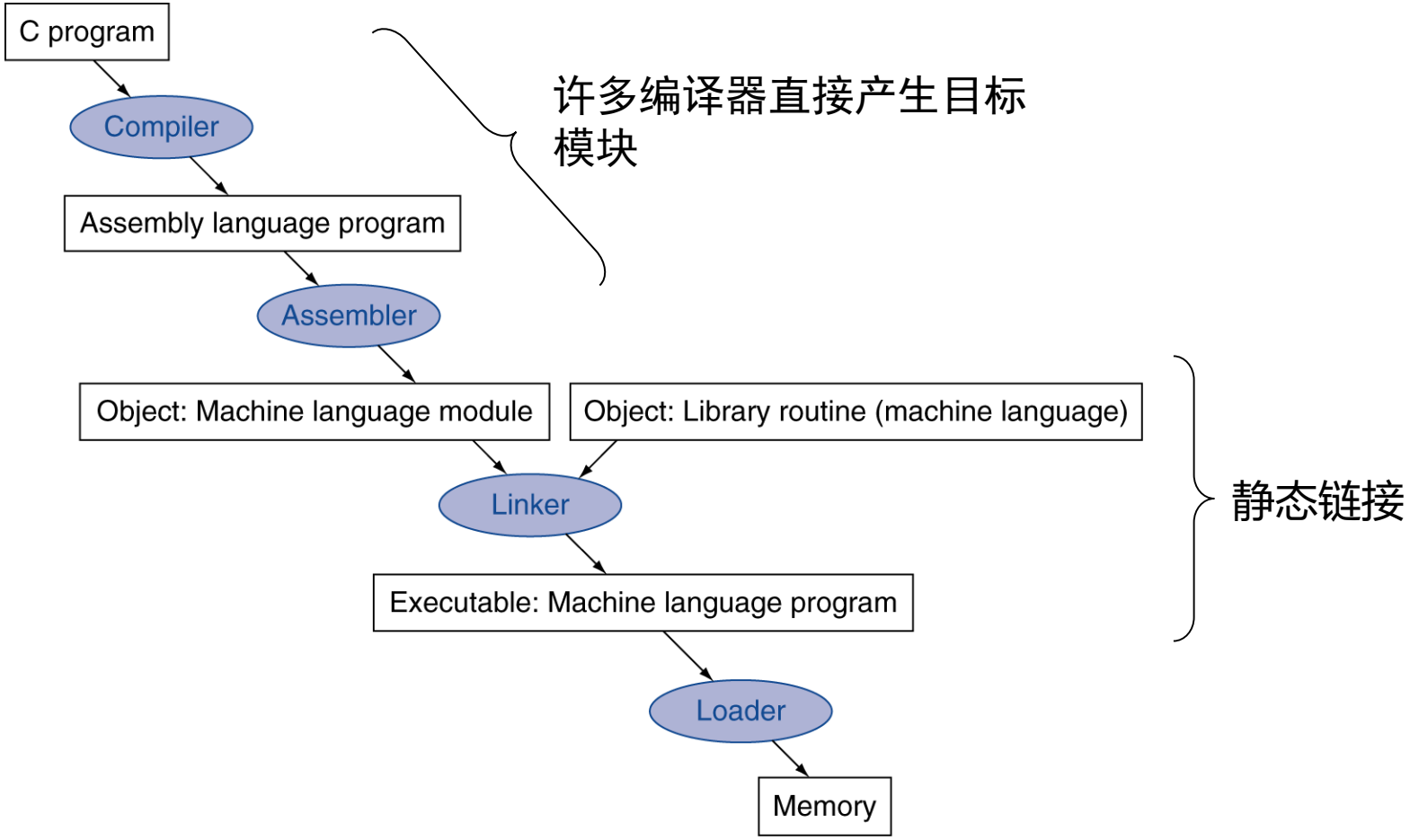
同步— MIPS中的同步

- 链接取数(Load linked)
- ll rt, offset(rs)
- 条件存数(Store conditional)
- sc rt, offset(rs)
  - 如果ll指令没改变该地址内容则成功，rt返回1；
  - 如果被改变了，则失败，rt返回0；
  - 例如：atomic swap（检测和设置锁变量）

0(s1) → t4
并行执行时能否成功？

```
try: add $t0,$zero,1 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

程序执行—Translation & Startup



程序执行—汇编伪指令

- 大多数汇编指令和机器指令是一对一的
- 特殊的是伪指令
- 伪指令：汇编指令的变种
- `move $t0, $t1` → `add $t0, $zero, $t1`

  `blt $t0, $t1, L` → `slt $at, $t0, $t1`
  `bne $at, $zero, L`

$at (register 1)：汇编程序的临时寄存器
（使用汇编语言时候的硬件额外的开销

程序执行—生成目标模块

- 汇编器（或编译器）把程序翻译成机器语言
- 提供从部分构建完整程序的信息
  - 目标文件头：描述目标文件其他部分的大小和位置
  - 正文段：翻译后的指令，包含机器语言代码
  - 静态数据段：包含在程序生命周期内分配的数据
  - 重定位信息，标记了一些程序加载进内存时依赖于绝对地址的指令和数据
  - 符号表，全局定义和外部引用
  - 调试信息：用于关联源文件

程序执行—链接目标模块

- 产生一个可执行的映像
  1. 合并段（代码和数据数据库象征性放入内存）
  2. 决定数据和指令标签的地址
  3. 修补引用（内部和外部引用）

- 可以留下依靠重定位程序修复的部分
  - 但虚拟内存，不需要做这些
  - 虚拟内存空间，程序必须以绝对地址装入

程序执行—加载程序

- 把待执行的程序从硬盘的镜像文件读入内存

1. 读取可执行文件头来确定正文段和数据段的大小
2. 为正文和数据创建一个足够大的地址空间
3. 把指令和初始数据拷贝到内存或者设置页表项，使它们可用
4. 把主程序的参数复制到栈顶
5. 初始化寄存器（包括堆栈指针$sp，帧指针$fp，全局指针$gp ）
6. 跳转到启动进程
   - 复制参数到寄存器并调用主函数main
   - 主函数返回时，通过系统调用exit终止程序

程序执行—动态链接库

- 调用时，只是连接或装入库文件
  - 过程代码重定位；
  - 避免所有程序中出现的链接库；但是这些库的信息是一次性代入内存，占用内存空间。只是在用到的时候才链接该库；
  - 自动装入最新的编译器中的版本的动态库。

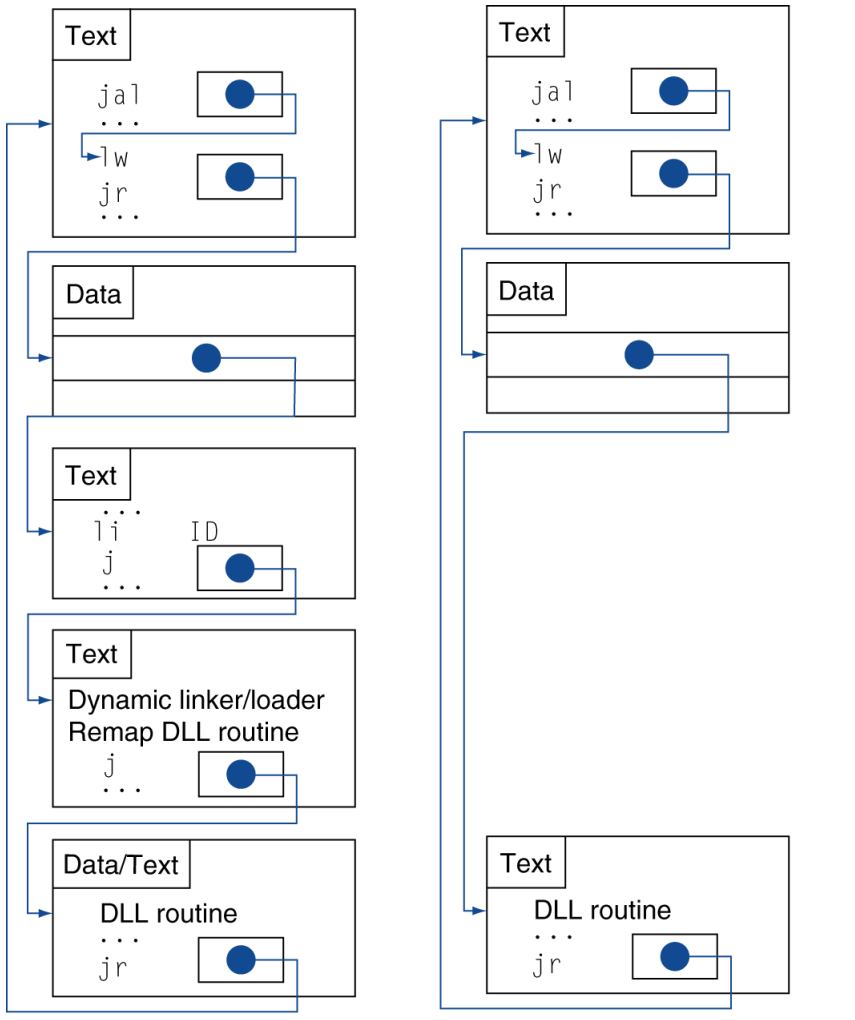
程序执行—晚过程连接

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

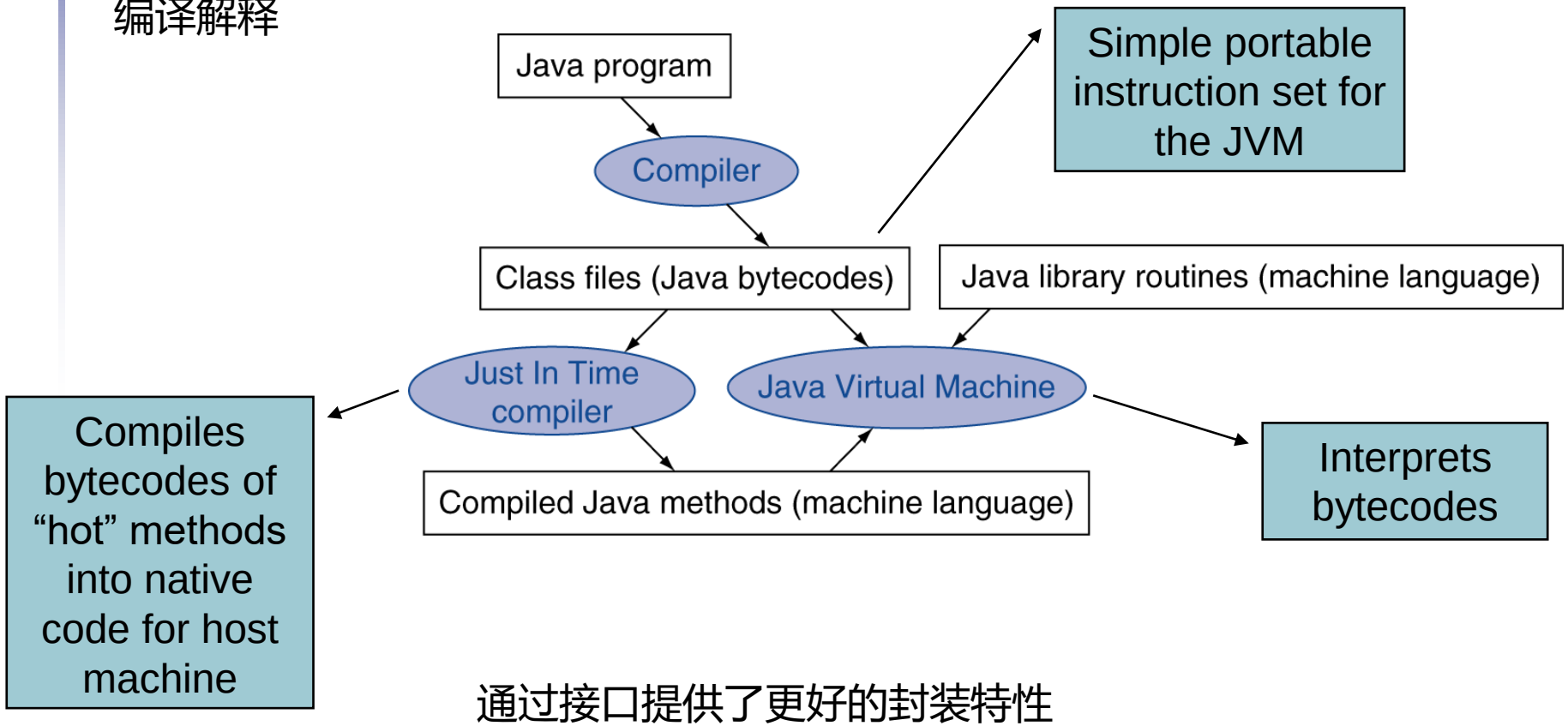


a. First call to DLL routine

b. Subsequent calls to DLL routine

程序执行—启动一个Java程序

编译解释



通过接口提供了更好的封装特性

C Sort Example

- 使用汇编指令的冒泡排序
- （交换内存中两个位置所存的值）
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

Swap过程

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

C 中排序过程

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

过程主体

```
        move $s2, $a0           # save $a0 into $s2
        move $s3, $a1           # save $a1 into $s3
        move $s0, $zero         # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1       # j = i - 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2        # $t1 = j * 4
        add  $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # $t3 = v[j]
        lw   $t4, 4($t2)        # $t4 = v[j + 1]
        slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
        move $a0, $s2           # 1st param of swap is v (old $a0)
        move $a1, $s1           # 2nd param of swap is j
        jal  swap               # call swap procedure
        addi $s1, $s1, -1       # j -= 1
        j    for2tst            # jump to test of inner loop
exit2:  addi $s0, $s0, 1        # i += 1
        j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

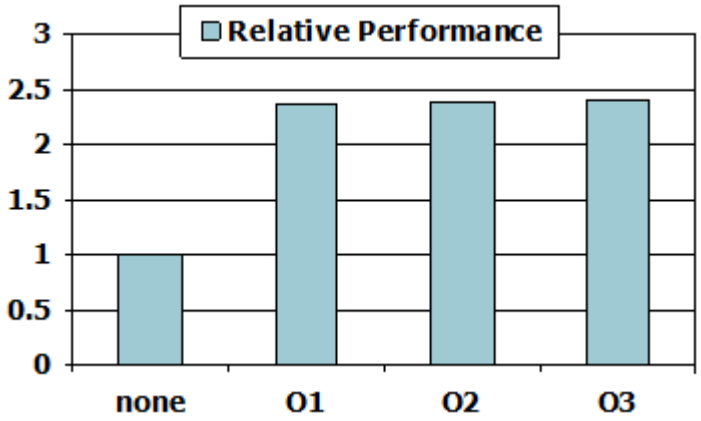
完整过程

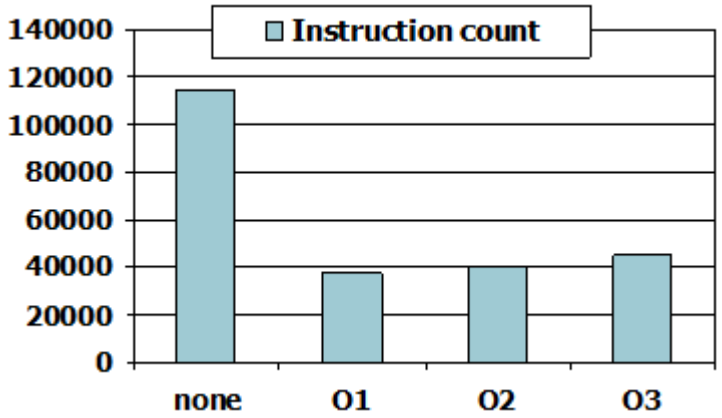
```
sort:    addi $sp,$sp, -20      # make room on stack for 5 registers
         sw $ra, 16($sp)        # save $ra on stack
         sw $s3,12($sp)         # save $s3 on stack
         sw $s2, 8($sp)         # save $s2 on stack
         sw $s1, 4($sp)         # save $s1 on stack
         sw $s0, 0($sp)         # save $s0 on stack
         …                      # procedure body
         …
         exit1: lw $s0, 0($sp)  # restore $s0 from stack
         lw $s1, 4($sp)         # restore $s1 from stack
         lw $s2, 8($sp)         # restore $s2 from stack
         lw $s3,12($sp)         # restore $s3 from stack
         lw $ra,16($sp)         # restore $ra from stack
         addi $sp,$sp, 20       # restore stack pointer
         jr $ra                 # return to calling routine
```

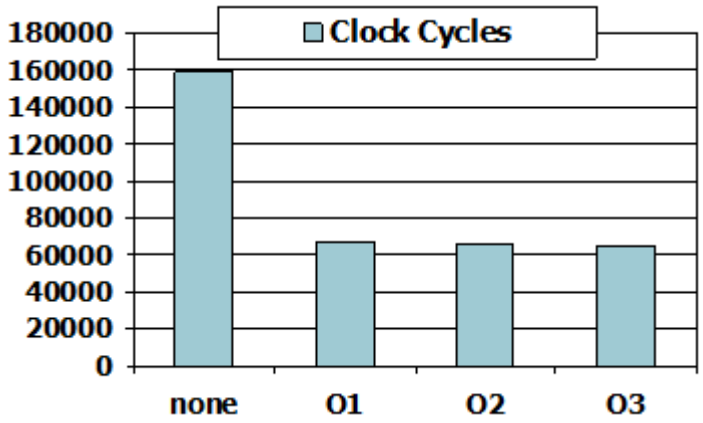
编译优化的影响

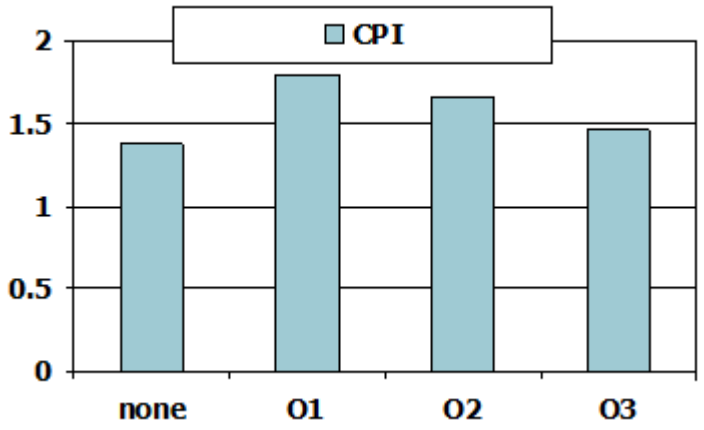
1、编译时间
2、指令数；
3、时钟周期
4、CPI

Compiled with gcc for Pentium 4 under Linux

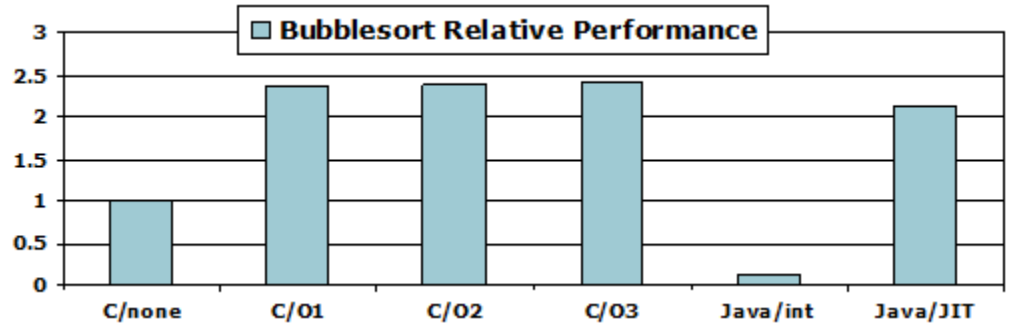


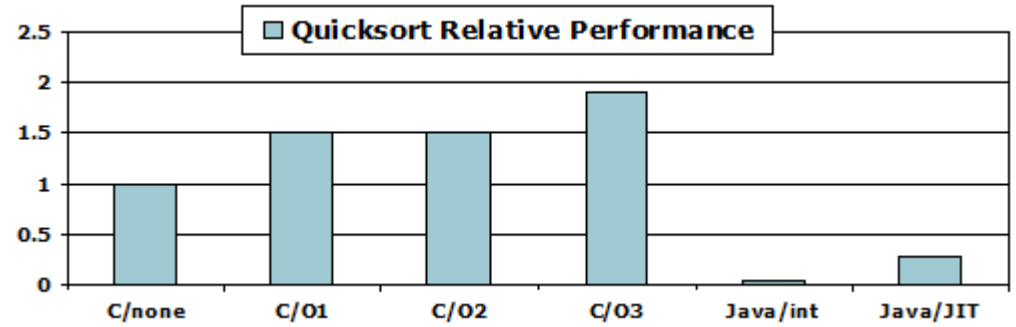


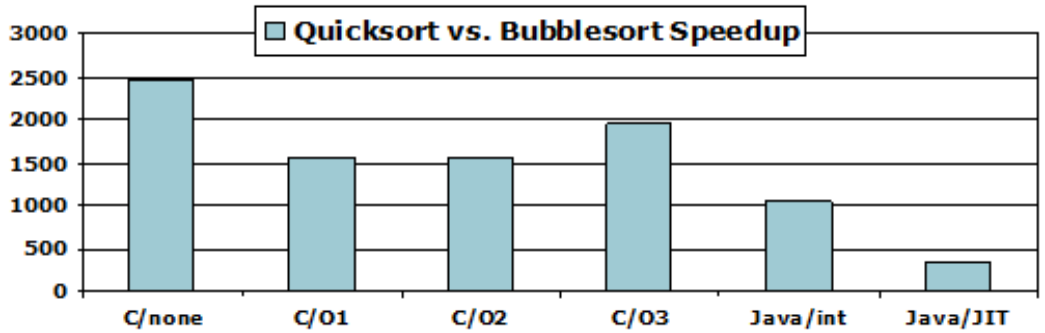




Effect of Language and Algorithm







经验教训

- 单独用指令数和CPI不能很好的描述性能
- 编译器优化对算法敏感
- 对比 Java虚拟机解释Java/JIT编译代码已经足够快了
  - 在一些情况下要对编译完C进行优化
- 没有什么能恢复一个垃圾的算法！

**数组和指针

- 数组下标计算涉及
  - 元素的下标乘以元素的大小
  - 加上数组的首地址
- 指针直接对应于内存地址
  - 可以避免复杂的索引

```
Lw $t0, 0($t1)
sw
```

**例子:数组和指针实现clearing

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 =
                        #   &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 =
                        #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto
loop1
```

```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 =
                        #   &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 =
                        #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

**数组和指针的比较

- 乘指令长度的操作变成移位操作
- 数组需要移到内环
  - 下标计算用i的自增操作
  - 相对于而后者 增加指针
- 编译器可以获得和手动使用指针相同的效果
  - 引导变量删除
  - 可以使程序更清晰，更安全

**ARM & MIPS Similarities

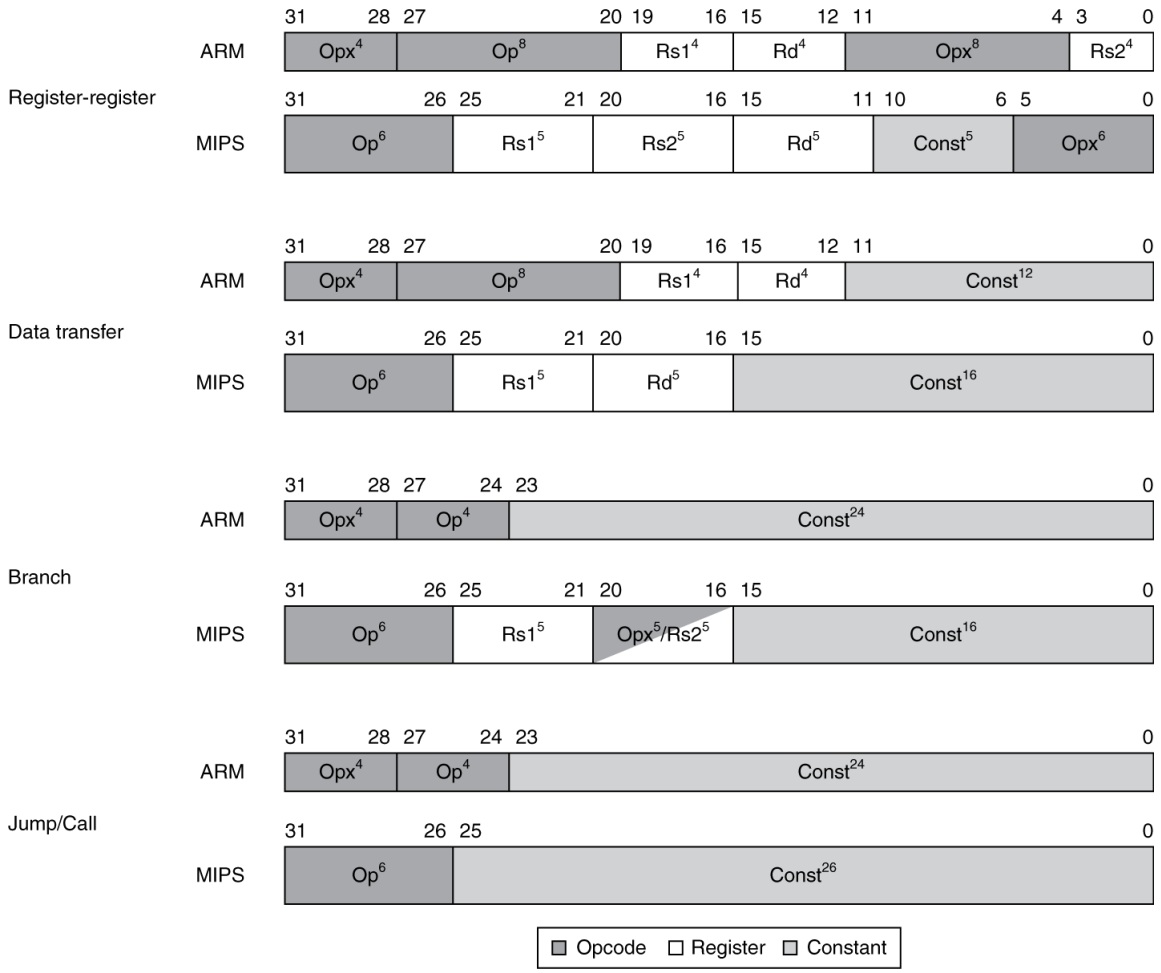
- ARM: 最流行的嵌入式处理机
- 基本指令集和MIPS类似

	ARM	MIPS
发布时间	1985	1985
指令大小	32 bits	32 bits
寻址空间	32-bit flat	32-bit flat
数据对齐	Aligned	Aligned
数据寻址方式	9	3
寄存器	15 × 32-bit	31 × 32-bit
Input/output	存储器映射	存储器映射

**ARM中的比较和分支

- 使用条件码表示一个算术/逻辑指令的结果
  - Negative（负）, zero（零）, carry（进位）, overflow（溢出）
  - 比较指令，设置条件码不保留结果
- 每条指令都有一个可选的执行条件
  - 指令字的高4位：条件值
  - 可以取代仅为了跳过一条指令的分支指令

**指令编码


Register-register
ARM
31 28 27 20 19 16 15 12 11 4 3 0
Opx4 Op8 Rs14 Rd4 Opx8 Rs24
MIPS
31 26 25 21 20 16 15 11 10 6 5 0
Op6 Rs15 Rs25 Rd5 Const5 Opx6
Data transfer
ARM
31 28 27 20 19 16 15 12 11 0
Opx4 Op8 Rs14 Rd4 Const12
MIPS
31 26 25 21 20 16 15 0
Op6 Rs15 Rd5 Const16
Branch
ARM
31 28 27 24 23 0
Opx4 Op4 Const24
MIPS
31 26 25 21 20 16 15 0
Op6 Rs15 Opx5/Rs25 Const16
Jump/Call
ARM
31 28 27 24 23 0
Opx4 Op4 Const24
MIPS
31 26 25 0
Op6 Const26
Opcode
Register
Constant

**The Intel x86 ISA指令集

- 向前兼容的演变过程

研究面向消费者需求!
计算机/微电子（集成电路）行业特点

  - 8080 (1974): 8-bit microprocessor
    - 累加器, 加了3个寄存器组对
  - 8086 (1978): 16-bit extension to 8080
    - 复杂指令集(CISC)
  - 8087 (1980): 浮点协同处理器
    - 添加了浮点指令寄存器堆栈
  - 80286 (1982): 24-bit addresses, MMU
    - 基于段的内存映射和保护
  - 80386 (1985): 32-bit extension (now IA-32)
    - 新的寻址方式和额外的操作
    - 增加了对页的支持并提供了段寻址

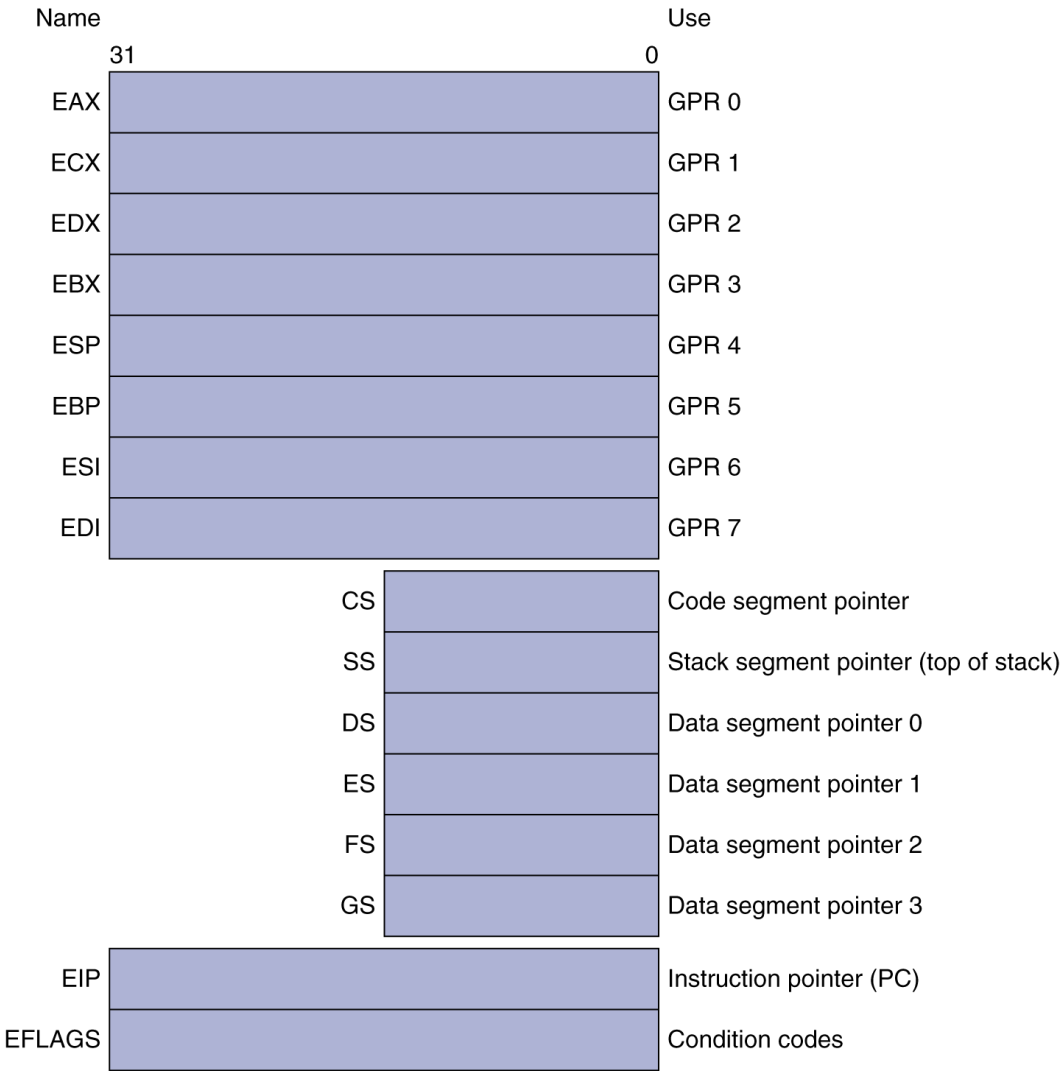
**The Intel x86 ISA

- 进一步发展
  - i486 (1989): pipelined（流水线）, on-chip caches（芯片缓冲） and FPU
    - 兼容其它品牌: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug 臭名昭著的浮点除法错误
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers 关联寄存器
  - Pentium 4 (2001)
    - New microarchitecture 新微架构
    - Added SSE2 instructions

**The Intel x86 ISA

- 更多
  - AMD64 (2003): 把体系结构扩展到64位
  - EM64T – 扩展内存地址到64位(2004)
    - AMD64 adopted by Intel (with refinements)
    - 增加了 SSE3 指令
  - Intel Core (2006)
    - 增加了SSE4 指令, 支持虚拟机
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- 如果interl不扩展兼容性，它的竞争者也会。
  - 技术上的优雅 ≠ 市场成功

**Basic x86 寄存器



**Basic x86 寻址方式

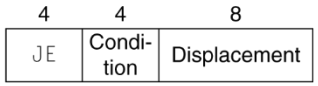
- 每条指令2个操作数

Source/dest operand	*Second source operand*
Register	*Register*
Register	*Immediate*
Register	*Memory*
Memory	*Register*
Memory	*Immediate*

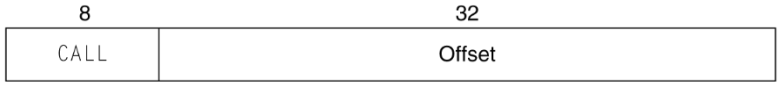
- 内存寻址方式
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

**x86 指令编码

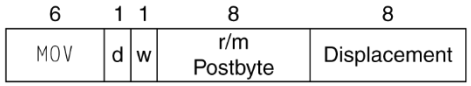
a. JE EIP + displacement



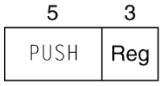
b. CALL



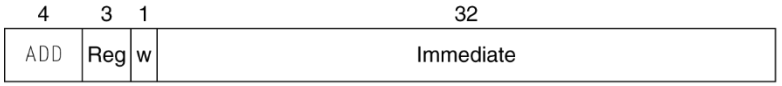
c. MOV EBX, [EDI + 45]



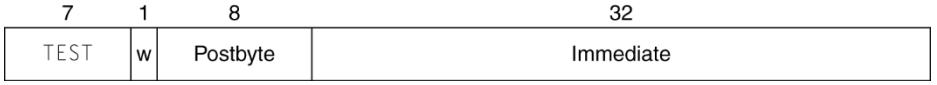
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 可变长度编码
  - 后置字节编码
  - 后置字节修改操作
    - 操作长度, 重复, 锁定, …

**实现 IA-32

- 复杂指令系统的实现更困难
  - 硬件翻译指令到简单的微处理机运行
    - 简单指令：1对1
    - 复杂指令： 1对多
  - 微引擎和 RISC相似
  - 市场份额使其经济可行
- 与RISC性能比较
  - 编译器可以避开复杂的指令

**ARM v8指令

- 迁移到64位, ARM做了个完整的检修
- ARM v8 类似 MIPS
  - 从v7来的变化:
    - 没有条件执行域
    - 立即数是12位的存储空间
    - 多个下降加载/存储
    - PC 不再是一个GPR
    - GPR 集扩展到32
    - 寻址方式为所有字大小工作
    - 除法指令
    - 分支如果等同于/不等同与指令

谬误

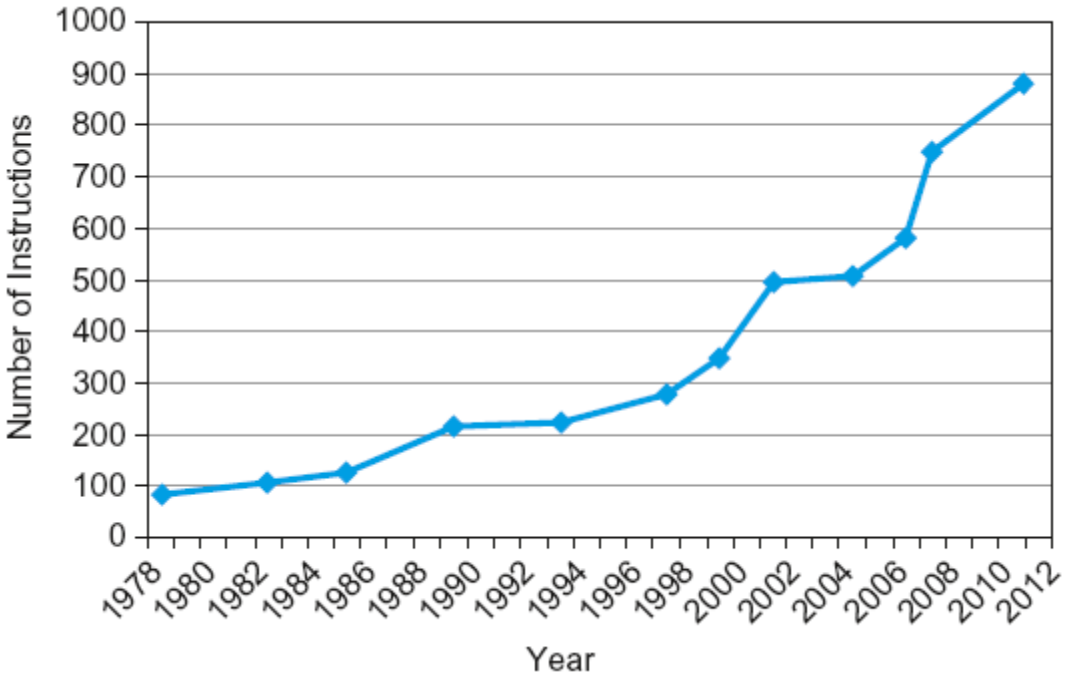
- 更强大的指令⇒ 更高的性能
  - 需求更少的指令
  - 但复杂指令实现困难
    - 可能带慢了所有的指令, 包括简单指令
  - 编译器更擅长从简单指令译码
- 使用汇编程序获得高性能
  - 但现代编译器更适合现代的处理机
  - 更多的代码量 ⇒ 更多的错误，更少的产出

1. 需要更多时钟周期
2.更复杂的译码电路

算法!

谬误

- 向前兼容性 ⇒ 指令集不需改变
  - 但是他们增加了指令数量



x86 instruction set

陷阱

- 连续字的地址不连续
  - 增量是4而不是1!
- 在自动变量的定义过程外，使用指针指向该变量
  - 例如, 使用一个过程中局部数组的指针，从该过程传出
  - 堆栈弹出后，指针变为无效

本章小结

- 设计原则
  1. 简单源于规整
  2. 越小越快
  3. 加速指向常用操作
  4. 优秀的设计需要适宜的复用方案
- 软件和硬件的各个层
  - 编译器, 汇编器, 硬件
- MIPS: 典型的RISC指令集
  - 相对于x86是CISC指令集

本章小结

- 用标准测试程序评测MIPS指令的执行
  - 考虑加速常用的操作
  - 考虑折中的方案

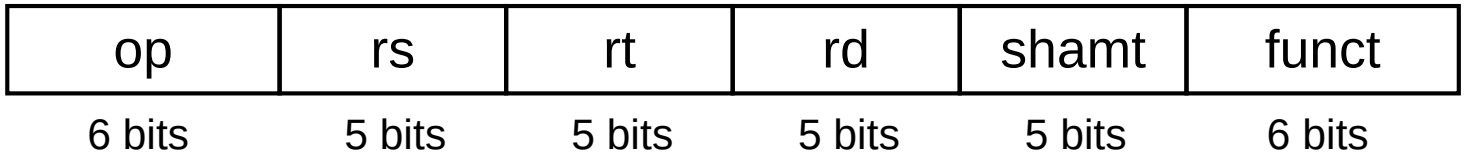
Classic:规定动作
Open：自选动作

指令类	*MIPS 示例*	*SPEC2006 整数*	*SPEC2006 浮点*
Arithmetic	***add, sub, addi***	***16%***	***48%***
Data transfer	***lw, sw, lb, lbu, lh, lhu, sb, lui***	***35%***	***36%***
Logical	***and, or, nor, andi, ori, sll, srl***	***12%***	***4%***
Cond. Branch	***beq, bne, slt, slti, sltiu***	***34%***	***8%***
Jump	***j, jr, jal***	***2%***	***0%***

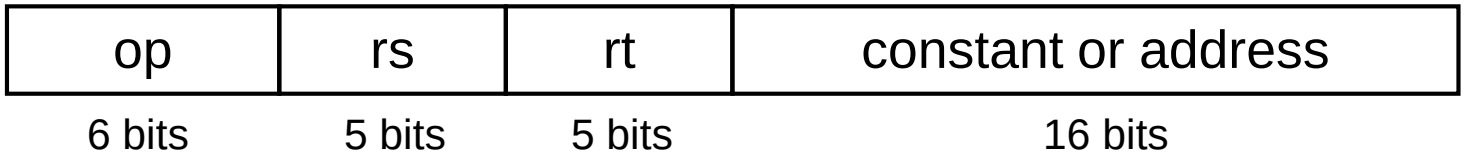
本章小结

寄存器名字	寄存器编号	寄存器功能
$zero	$0	恒等于零
$at	$1	被汇编器保留，用于处理大的常数
$v0 – $v1	$2-$3	存放函数返回值
$a0 – $a3	$4-$7	传递函数参数
$t0 – $t7	$8-$15	存放临时变量
$s0 – $s7	$16-$23	存放需要保存的临时值
$t8 – $t9	$24-$25	额外的存放临时变量
$k0 – $k1	$26-$27	用于操作系统内核
$gp	$28	指向全局变量的指针
$sp	$29	指向栈顶的指针
$fp	$30	指向栈帧的指针
$ra	$31	返回地址，用于函数调用

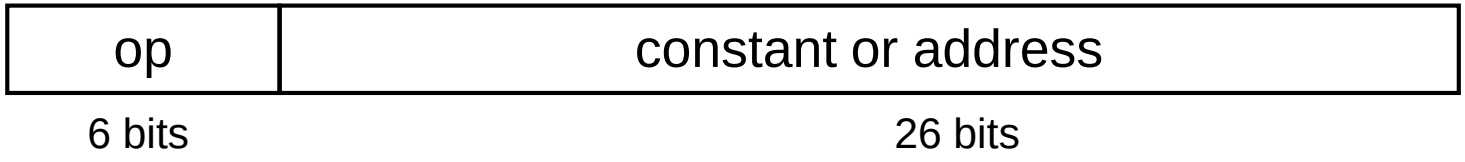
本章小结



R



I



J

简单源于规整（Simplicity favours regularity）
寄存器比内存要更快地存取数据
算术运算指令只使用寄存器操作
指令/数据都被表示为二进制（补码）

Add $t0, $t1, $t2
Lw $s0, 5($s1)

本章小结

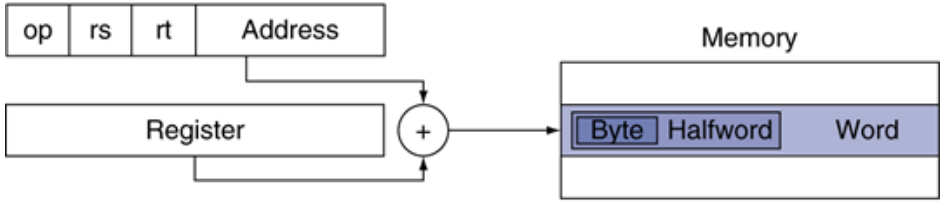
1. Immediate addressing

addi $s1, $s2, 4

2. Register addressing

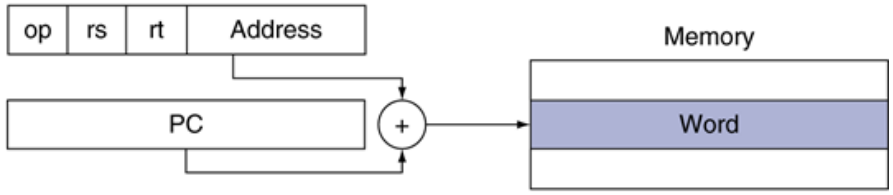
add $s1, $s2, $s3

3. Base addressing



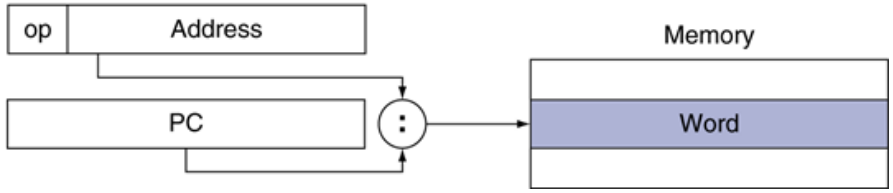
lw $s1, 40($s2)

4. PC-relative addressing



ben $s1, $s2, ELSE

5. Pseudodirect addressing



jal LEAF

本章小结

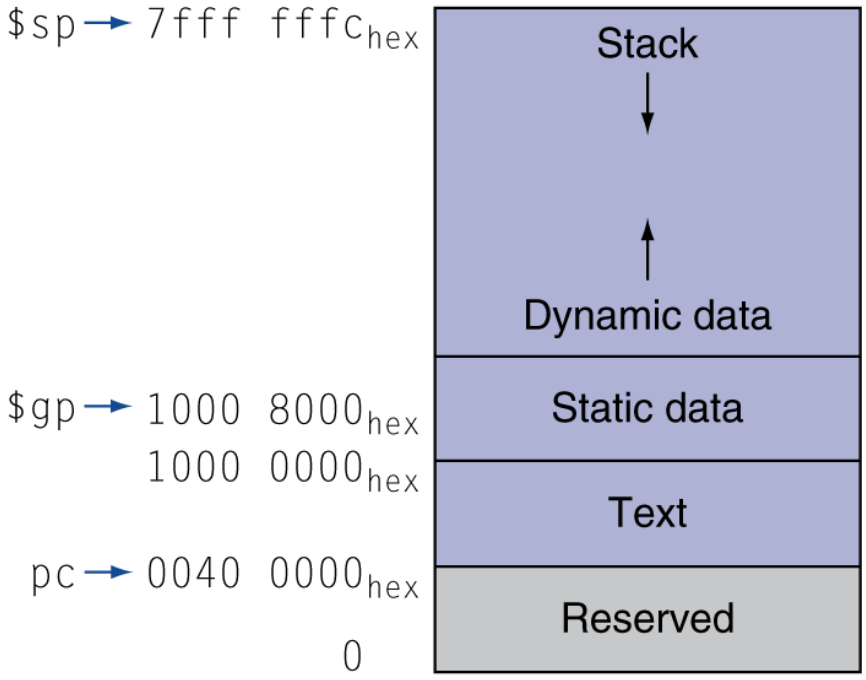
过程

1.保留现场：地址和资源

2.获得信息：地址和资源

3.参数传递

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes



本章小结

编译：把程序翻译成机器语言

序号	名称	作用
1	文件头	描述文件大小和位置
2	正文	指令的机器代码
3	数据	数据
4	重定位信息	依赖绝对地址的指令和数据
5	符号表	全局定义和外部引用
6	调试信息	关联文件

链接：产生一个可执行的映像

序号	名称	作用
1	合并段	代码和数据数据库放入内存
2	地址变换	计算物理地址
3	修补引用	库程序

加载：程序从硬盘的镜像文件读入内存

序号	名称	作用I内容	序号	名称	作用
1	读取文件头	确定文件大小	4	参数复制	把主程序的参数复制到栈顶
2	创建空间	根据指令和数据创建空间	5	寄存器初始化	$sp，$fp，$gp
3	拷贝	指令和初始数据拷贝内存/设置页表项	6	启动进程	启动进程

习题

3个不同处理器 P1、P2、P3 采用相同的指令集。它们的主频和CPI如下表：

问：

（1）哪一个处理器单位时间内执行的指令数最多？

（2）给出2秒内每个处理器可执行的指令数和用的CPU时钟数。

（3）如果希望执行时间缩小30%，但是CPI会增加40%。应该怎么调整主频？

处理器	主频（GHz）	CPI
P1	3	1.5
P2	2.5	1
P3	4	2.2

T=N*CPI*1/f

习题

（1）T_{CPU} = N x CPI x 1/f，单位时间内执行的指令数：N/ T_{CPU}

P1：f_1/CPI_1 = 2 x 10^9

P2：f_2/CPI_2 = 2.5 x 10^9

P3：f_3/CPI_3 = 1.818 x 10^9

（2）2秒内的指令数和CPU时钟数

P1：4 x 10^9，6 x 10^9

P2：5 x 10^9， 5 x 10^9

P3：3.636 x 10^9， 4.4 x 10^9

（3） T_{CPU} *0.7 = N x CPI*1.4 x 1/f，所以主频调整为：

P1：6 x 10^9

P2：5 x 10^9

P3：8 x 10^9

习题

2个处理器 P1、P2 采用相同指令集，执行不同指令的CPI如下：

一个程序里面包含这4种类型指令的比例分别为10%，20%，50%，20%，问：

（1）执行该程序哪一个处理器用的时间多？

（2）执行该程序每个处理器的平均CPI是多少？

处理器	类型1	类型2	类型3	类型4
P1	1	2	3	3
P2	2	2	2	2

(1)

$$T_{CPU} = \sum_{i=1}^{n} (CPI_i \times C_i) \text{ x N x } 1/f$$

T_{P1} = (1x10%+2x20%+3x50%+3x20%) x N x 1/f = 2.6 x N x 1/f

T_{P2} = (2x10%+2x20%+2x50%+2x20%) x N x 1/f = 2 x N x 1/f

(2)

CPI_{P1} = 2.6

CPI_{P2} = 2

习题

给出以下C语句的MIPS代码：B[10] = A[k-i]

寄存器分配如下：

k →S0, i → S1, A → S5, B → S6

习题

给出以下C语句的MIPS代码：B[10] = A[k-i]

寄存器分配如下：

k →S0, i → S1, A → S5, B → S6

答

```
Sub $t0, $s0, $s1
Sll $t0, $t0, 2
Add $t1, $s5, $t0
Lw $t0, 0($t1)
Sw $t0, 40($s6)
```

习题

指令beq $t0,$t1, 100 的地址为 0x8000，如果：

（1）t0=10，t1=11，下一条指令的地址是什么?

（2）t0=10，t1=10，下一条指令的地址是什么?

习题

指令beq $t0,$t1, 100 的地址为 0x8000，如果：

（1）t0=10，t1=11，下一条指令的地址是什么?

（2）t0=10，t1=10，下一条指令的地址是什么?

答：

（1）因为 t0=10 ≠ t1=11，下一条指令的地址为：0x8004。

（2）因为 t0=10 = t1=10，下一条指令的地址为：

0x8004 + (100*4)h

习题

假设当前PC=0x00000000，问：

（1）跳转指令 j 可跳转的地址范围是什么？

（2）分支指令 bne可跳转的地址范围是什么？

习题

假设当前PC=0x00000000，问：

（1）跳转指令 j 可跳转的地址范围是什么?

（2）分支指令 bne可跳转的地址范围是什么?

（1）

26位最大正数为：0x3FFFFFF -> 左移 2 位为： 0xFFFFFFC

所以最大可跳转范围： 0x0FFFFFFC

（2）

16位最大正数为：0x7FFF -> 左移 2 位为： 0x1FFFC

所以最大正向跳转： 0x1FFFC + 4 = 0x20000

16位最小负数为：0x8000 -> 左移 2 位为： 0xE0000

所以最大负向跳转： 0xFFFE0000 + 4 = 0xFFFE0004