

# 第十二讲

---

## 第五讲：指令系统与MIPS汇编

### 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

### 二. MIPS指令系统

### 三. MIPS汇编语言（自学）

1. 概述
2. MIPS汇编指令和存储格式
3. MIPS汇编程序

## 1.1 指令系统概述

### ❖ 指令系统的基本问题

- 操作类型：应该提供哪些（多少）操作？
  - 用LD/ST/INC/BRN已经足够编写任何计算程序，但不实用，程序太长。
- 操作对象：如何表示？可以表示多少？
  - 大多数是双值运算（如 $A \leftarrow B + C$ ）
  - 存在单值运算（如 $A \leftarrow \sim B$ ）
- 指令格式：如何将这些内容编码成一致的格式？
  - 指令长度、字段、编码等问题

## 1.1 指令系统概述

### ❖ 机器指令的要素

- **操作码 (Operation Code)**: 指明进行的何种操作
- **源操作数地址 (Source Operand Reference)**: 参加操作的操作数的地址，可能有多个。
- **目的操作数地址 (Destination Operand Reference)**: 保存操作结果的地址。
- **下条指令的地址 (Next Instruction Reference)**: 指明下一条要运行的指令的位置。程序中的指令，大多数情况下是按顺序依次执行的，因此，绝大多数指令中并不显式指明下一条指令的地址，也就是说，指令格式中可以包含这部分信息。只有少数指令需要显示指明下一条指令的地址。

# 1.1 指令系统概述

## ❖ 指令类型

- ▶ 数据传输指令：寄存器与存储器之间，寄存器之间传递数据；
- ▶ 算术/逻辑运算指令：寄存器（或存储器）中整型数或逻辑型数据的运算操作，以及处理浮点数运算的浮点运算指令；
- ▶ 程序控制指令：控制程序执行顺序，条件转移或跳转，子程序调用和返回等；
- ▶ 其它指令：多用户多任务系统中的特权指令；复位/暂停/空操作等指令。

# 1.1 指令系统概述

## ❖ 操作数的类型

- 数值（无符号、定点、浮点）
- 逻辑型数、字符
- 地址（操作数地址、指令地址）

## ❖ 操作数的位置

- 存储器（存储器地址）
- 寄存器（寄存器地址）
- 输入输出端口（输入输出端口地址）

## ❖ 操作数的存储方式

- 大端（**big-endian**）次序：最高有效字节存储在地址最小位置
- 小端（**little-endian**）次序：最低有效字节存储在地址最小位置

例： `Int a; //0x12345678`

地址	值
a+0	12
a+1	34
a+2	56
a+3	78

大端次序

地址	值
a+0	78
a+1	56
a+2	34
a+3	12

小端次序

## 1.1 指令系统概述

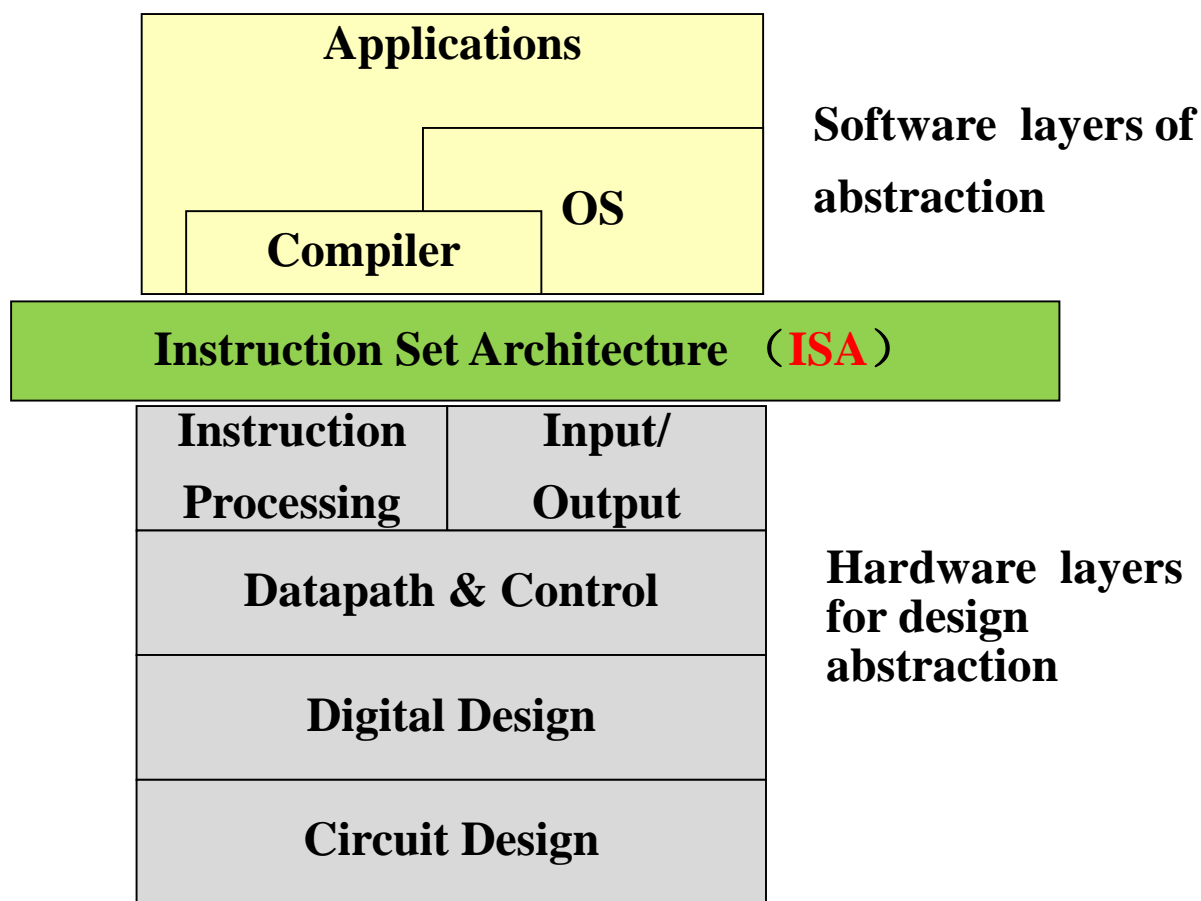
### ❖ 从指令执行周期看指令涉及的内容



# 1.1 指令系统概述

## ❖ 指令集系统结构(ISA)

➤ 机器语言编程者的视角，机器内部结构和行为能力的指令级抽象





## 1.1 指令系统概述

### ❖ 指令集系统架构（ISA）种类

➤ 大部分ISA都可归类为通用寄存器系统结构

➤ Register-Memory式ISA（如 X86）

- 多种指令可以访问内存；
- 存在寄存器操作数和内存操作数直接运行的指令；

➤ Register-Register (Load-Store) 式ISA（如 MIPS）

- 只有装载（LOAD）和存储（STORE）指令可以访问内存
- 运算指令操作数全部为寄存器操作数；

### ❖ Load-Store是ISA的一种趋势

## 1.1 指令系统概述

---

### ❖ 通用寄存器的优势

- 寄存器比存储器快
- 寄存器便于编译器使用
- 寄存器可以保存变量
- 减少存储器访问，提高速度
- 提高代码密度，寄存器地址比存储器地址短

## 第五讲：指令系统与MIPS汇编

### 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

### 二. MIPS指令系统

### 三. MIPS汇编语言

1. 概述
2. MIPS汇编指令和存储格式
3. MIPS汇编程序

## 1.2 指令格式

❖ **机器指令：** 计算机硬件可以执行的表示一种基本操作的二进制代码

- 指令格式：操作码 + 操作数（操作数地址）
- 操作码：指明指令的操作性质
- 操作数（地址）：指明操作数的位置（或操作数本身）



❖ **指令的表示**

- 机器表示：二进制代码
- 符号化表示：助记符，如： MOV AX, BX

## 1.2 指令格式

### ❖ 操作数地址的数目

- 三地址:  $Des \leftarrow (Sur1) \text{ OP } (Sur2)$

OP	Des Add	Sur1 Add	Sur2 Add
----	---------	----------	----------

- 双地址:  $Des \leftarrow (Sur) \text{ OP } (Des)$

OP	Des Add	Sur Add
----	---------	---------

- 单地址: 累加器作为其中一个操作数的双操作数型,  
或单操作数型

OP	Add
----	-----

- 零地址: 隐含操作数型, 或无操作数型

OP
----

## 1.2 指令格式

### ❖ 操作码结构

- 固定长度操作码：操作码长度（占二进制位数）固定不变。
  - 硬件设计简单
  - 指令译码时间开销较小
  - 指令空间效率较低
- 可变长度操作码：操作码长度随指令地址数目的不同而不同。
  - 硬件设计相对复杂
  - 指令译码时间开销较大
  - 指令空间利用率较高

### ❖ 指令长度

- 定长指令系统，如**MIPS**指令
- 变长指令系统：一般为字节的整数倍，如**80X86**指令

## 第五讲：指令系统与MIPS汇编

### 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

### 二. MIPS指令系统

### 三. MIPS汇编语言

1. 概述
2. MIPS汇编指令和存储格式
3. MIPS汇编程序

## 1.3 寻址方式

### ❖ 形式地址与有效地址

- 形式地址：指令中直接给出的操作数的地址编码
- 有效地址：操作数在内存中的地址，可由形式地址和寻址方式计算得到
- 寻址方式：根据形式地址，计算出操作数有效地址的方法（算法）

### ❖ 常用寻址方式

- 立即寻址
- 寄存器直接寻址
- 寄存器间接寻址
- 基址寻址 / 变址寻址
- 相对寻址：基址寻址的特例，程序计数器PC作为基址寄存器
- 堆栈寻址



## 1.3 寻址方式

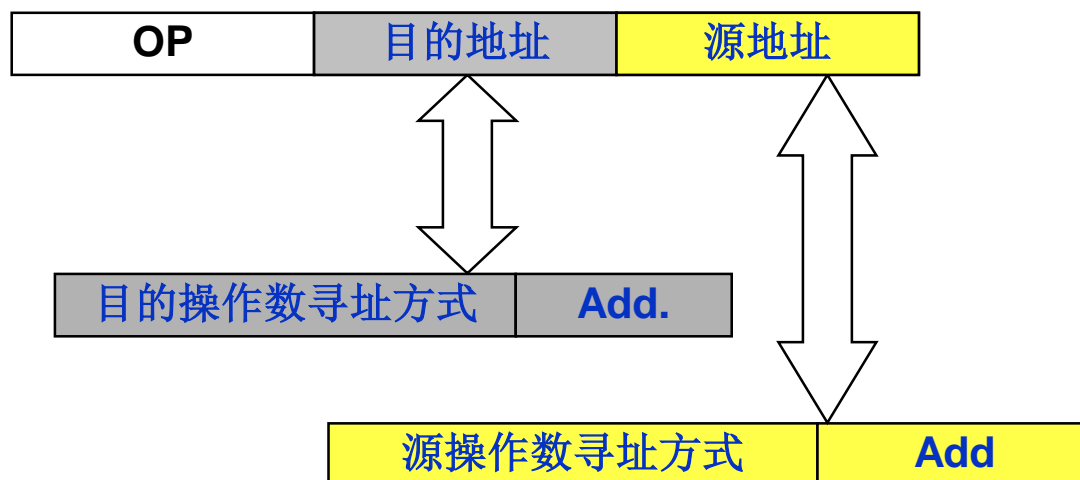
### ❖ 寻址方式的确定

➤ 在操作码中给定寻址方式:

- 如**MIPS**指令，指令中仅有一个主(虚)存地址的，且指令中仅有一二种寻址方式。**Load/store**型机器指令属于这种情况。

➤ 指令中专门的寻址方式位

- 如**X86**指令，指令中有多个操作数，且寻址方式各不相同，需要各自说明寻址方式。



## 1.3 寻址方式

### ❖ 指令代码和寻址描述中有关缩写的约定

- **OP**: 操作码
- **Des**: 目的操作数地址
- **Sur**: 源操作数地址
- **A或Add**: 形式地址（内存地址）
- **Mod**: 寻址方式
- **Rn**: 通用寄存器
- **Rx**: 变址寄存器
- **Rb**: 基址寄存器
- **SP**: 堆栈指针（寄存器）
- **EA**: 有效地址
- **Data**: 操作数
- **Operand**: 操作数
- **(Rn)**: 寄存器Rn的内容（值）
- **Mem[A]**: 内存地址为A的单元的内容
- **Imme. Data**: 立即数
- **XXH**: 16进制数XX

## 1.3 寻址方式

### ❖ 立即寻址

- 操作数直接在指令代码中给出。



↑  
源操作数

### ❖ 说明

- 立即寻址只能作为双操作数指令的源操作数。
- **Operand = Imme. Data**
- 例: **MOV AX, 1000H** (80X86指令,  $AX \leftarrow 1000H$ )  
**addi \$s1, \$s2, 100** (MIPS指令,  $\$s1 \leftarrow \$s2 + 100$ )

### ❖ 思考

- 立即寻址的操作数在什么地方, 存储器 or 寄存器?
- 立即数的地址?

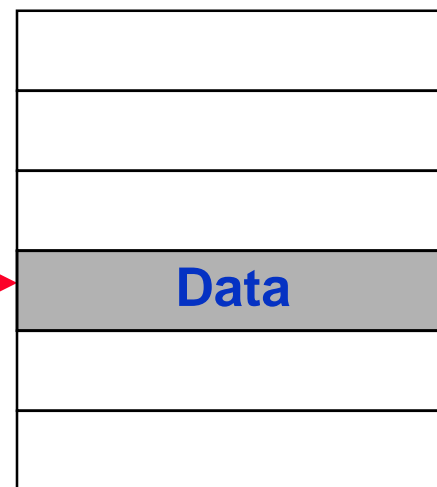
## 1.3 寻址方式

### ❖ 寄存器直接寻址

- 操作数在寄存器中，指令地址字段给出寄存器的地址（编码）
- $EA = Rn$ ,  $Operand = (Rn)$
- 例：MOV [BX], AX （80X86指令）  
add \$s1,\$s2,\$s3 （MIPS指令， $\$s1 \leftarrow \$s2 + \$s3$ ）

OP	Des	Mod	Rn
----	-----	-----	----

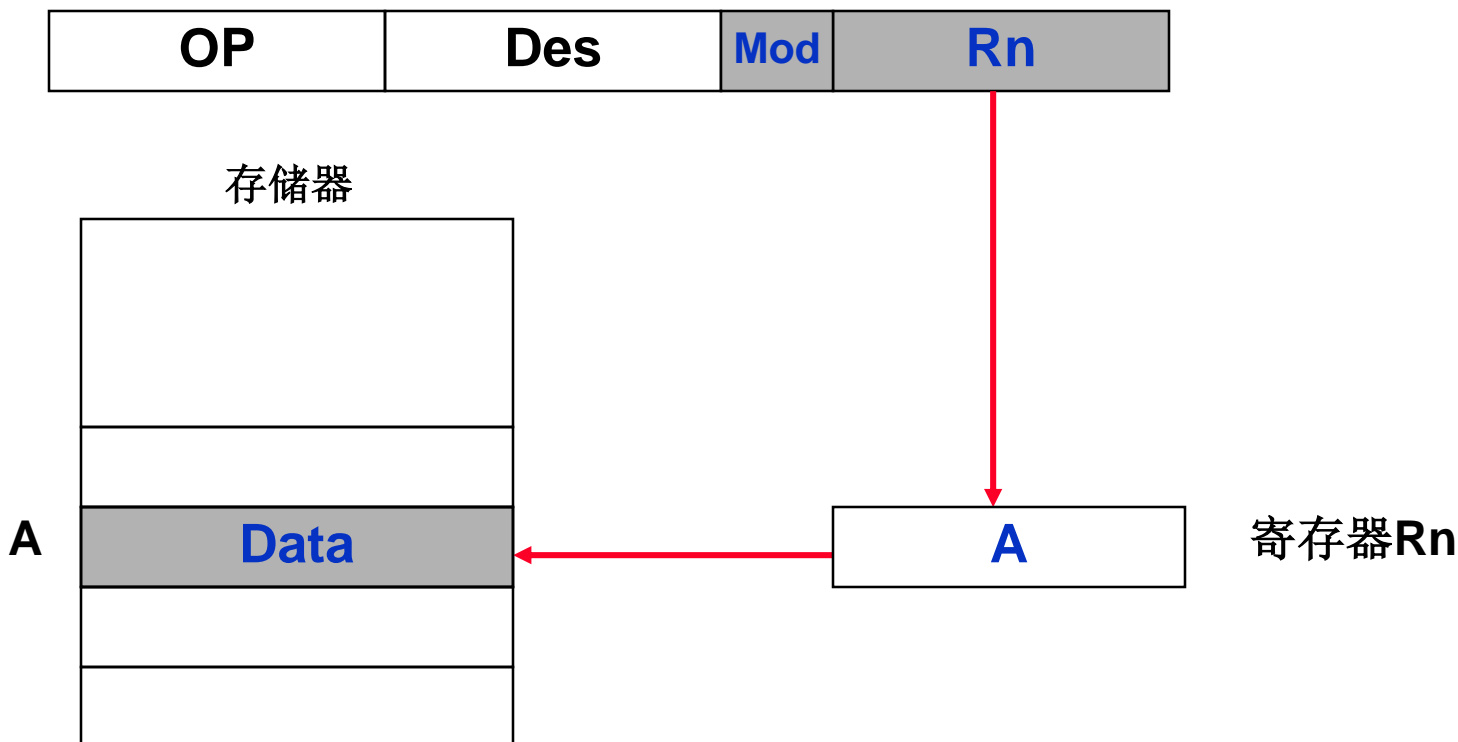
通用寄存器组GR



## 1.3 寻址方式

### ❖ 寄存器间接寻址

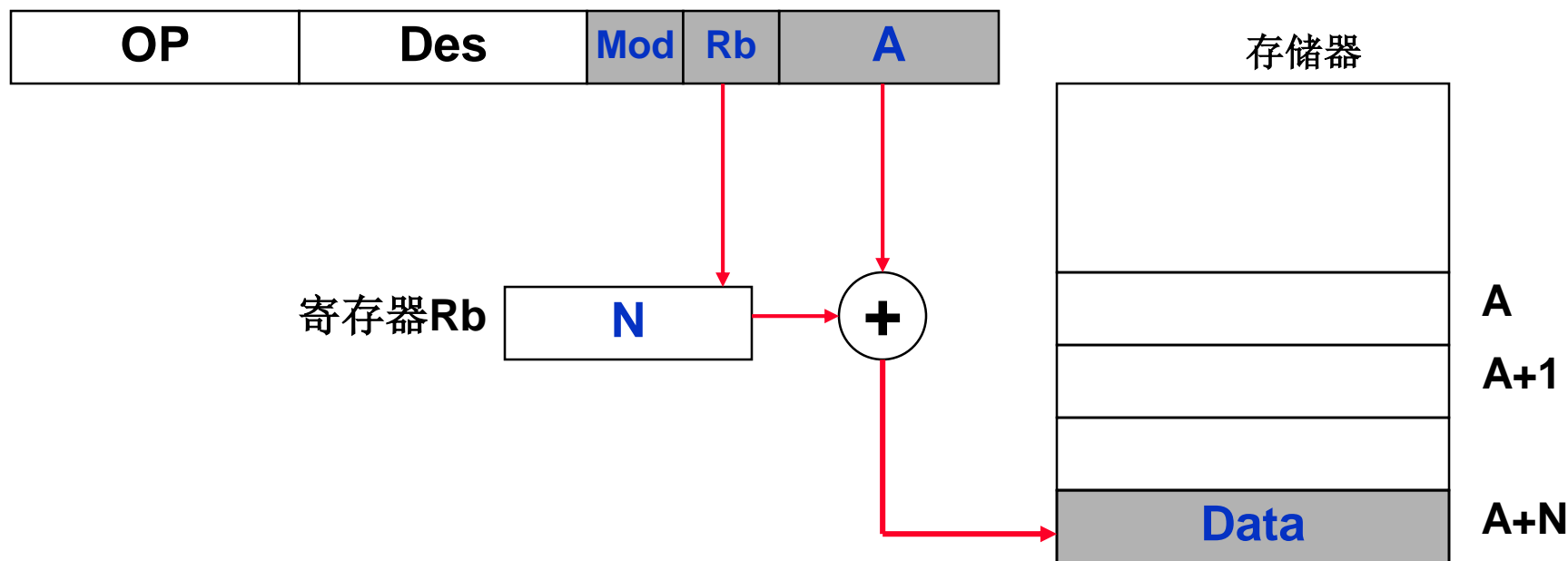
- 操作数在存储器中，指令地址字段中给出的寄存器的内容是操作数在存储器中的地址。
- $EA = (Rn)$ ,  $Operand = Mem[(Rn)]$
- 例: **MOV AX, [BX]** (80X86指令,  $AX \leftarrow Mem[(BX)]$ )



## 1.3 寻址方式

### ❖ 基址寻址

- 操作数在存储器中，指令地址字段给出一基址寄存器和一形式地址，基址寄存器的内容与形式地址之和是操作数的内存地址。
- $EA = (Rb) + A$ ,  $Operand = Mem[(Rb) + A]$
- 例: **MOV AX, 1000H[BX]** (80X86指令,  $AX \leftarrow Mem[(Bx) + 1000]$ )  
**lw \$s1, 100(\$s2)** (MIPS指令,  $\$s1 \leftarrow Mem[\$s2 + 100]$ )

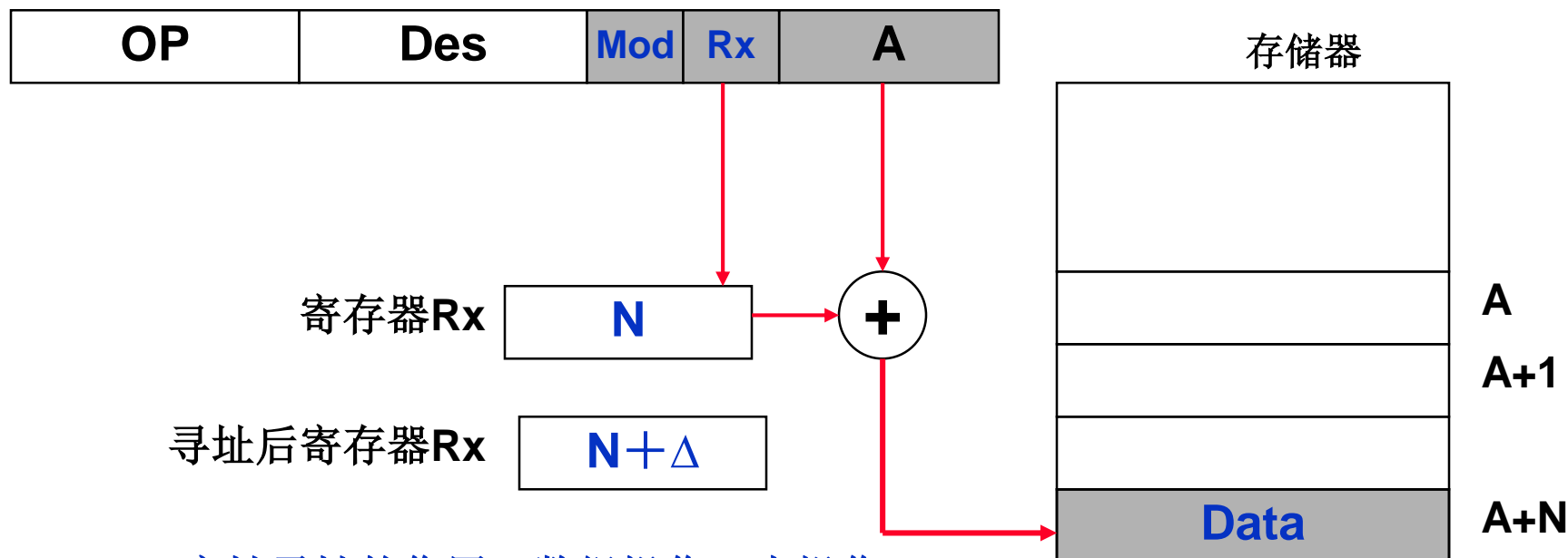


基址寻址的作用：较短的形式地址长度可以实现较大的存储空间的寻址。

## 1.3 寻址方式

### ❖ 变址寻址

- 操作数在存储器中，指令地址字段给出一变址寄存器和一形式地址，变址寄存器的内容与形式地址之和是操作数的内存地址。
- $EA = (Rx) + A$ ,  $Operand = Mem[(Rx) + A]$
- 有的系统中，变址寻址完成后，变址寄存器的内容将自动进行调整。  
 $Rx \leftarrow (Rx) + \Delta$  (操作数Data的字节数)
- 例: `MOV AX, 1000H[DI]` (80X86指令)

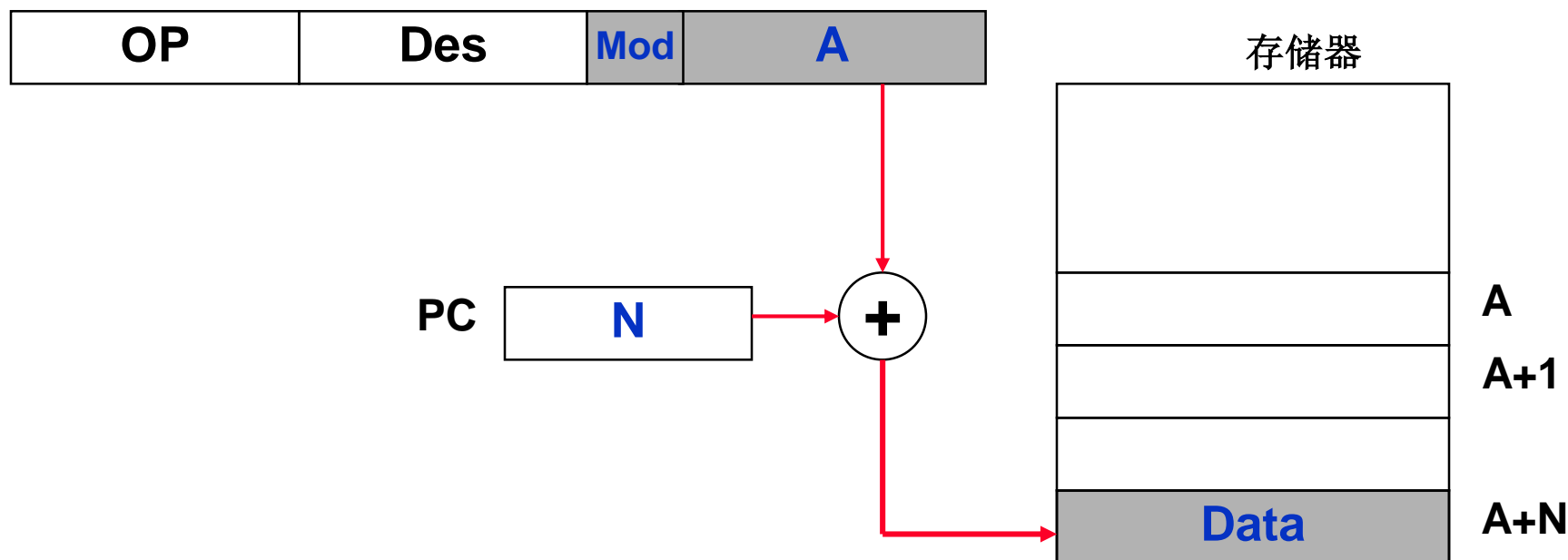


变址寻址的作用：数组操作，串操作

## 1.3 寻址方式

### ❖ 相对寻址

- 基址寻址的特例，由程序计数器**PC**作为基址寄存器，指令中给出的形式地址作为位移量，二者之和是操作数的内存地址。
- $EA = (PC) + A$ ,  $Operand = Mem[(PC) + A]$
- 例：JNE A (80X86指令)  
    beq \$s1, \$s2, 100 (MIPS指令)

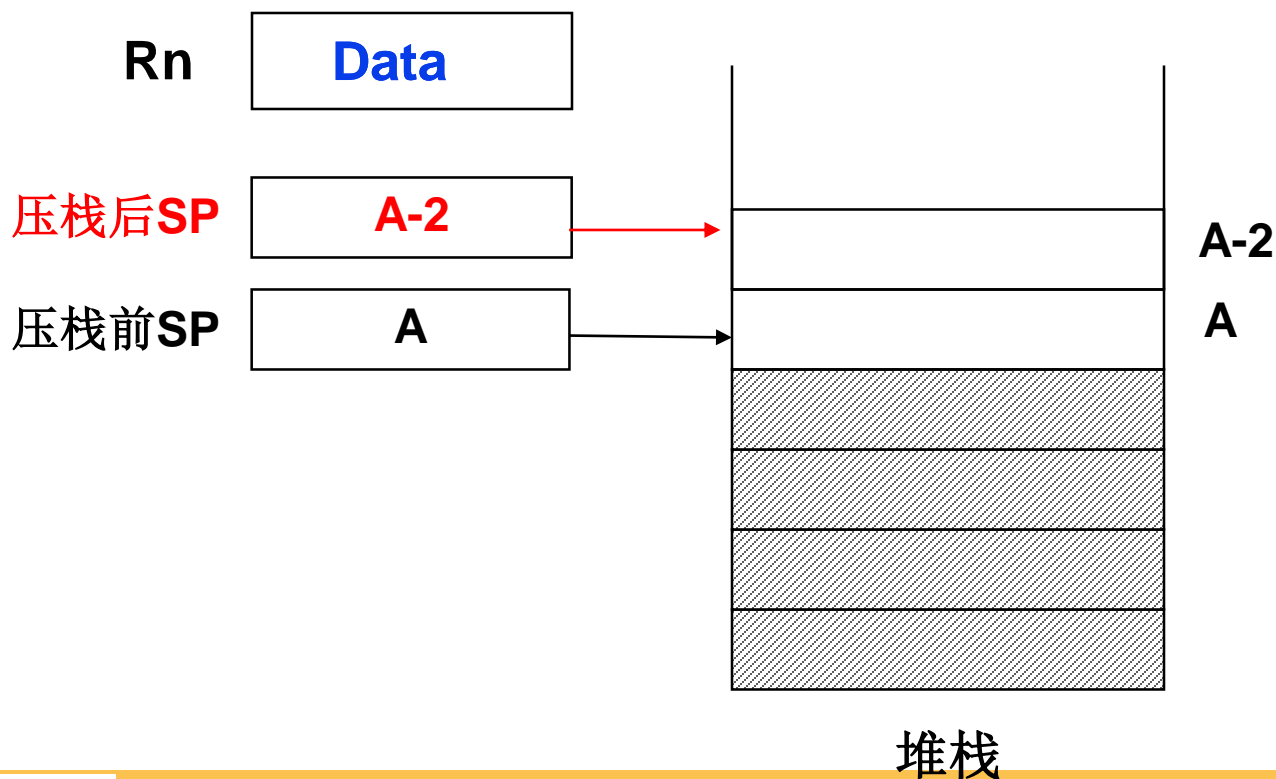




## 1.3 寻址方式

### ❖ 堆栈寻址

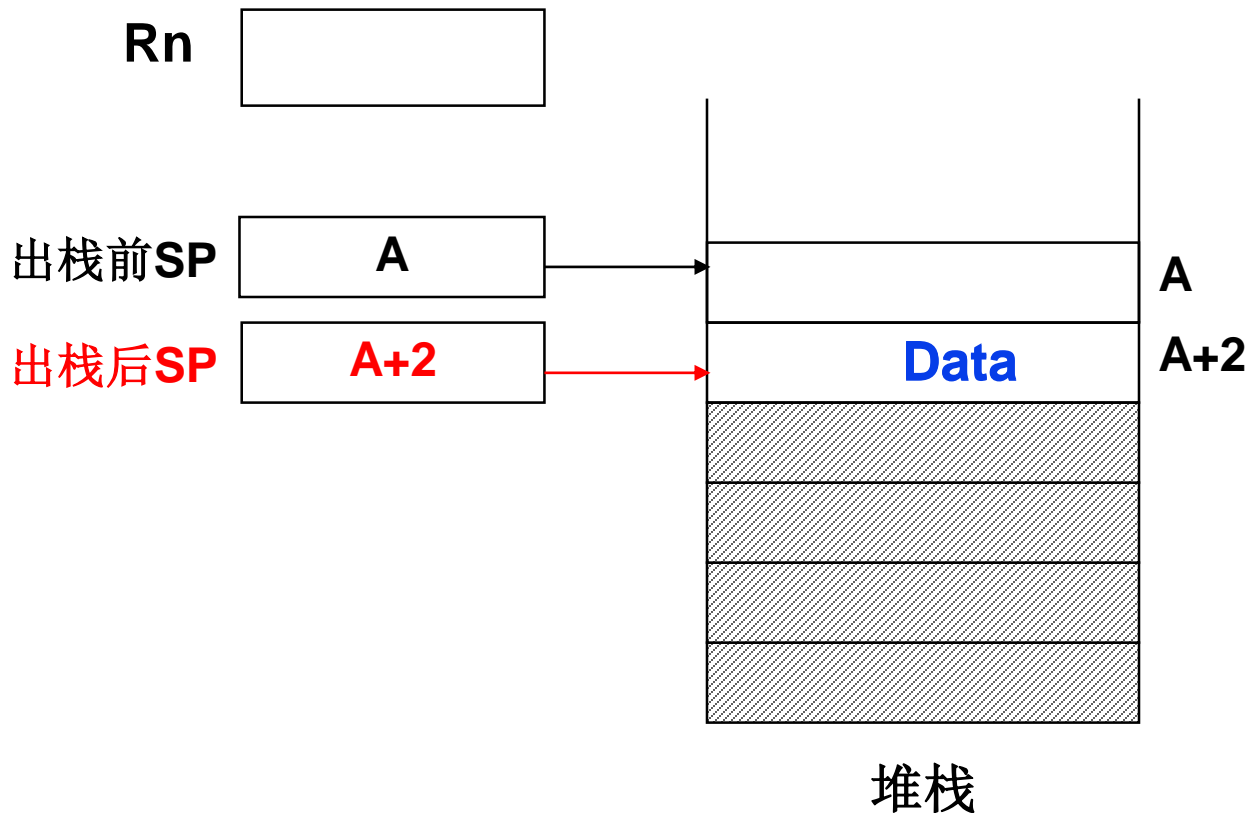
- 堆栈的结构：一段内存区域。
- 堆栈指针(SP)：是一个特殊寄存器部件，指向栈顶
- 压栈操作：**PUSH Rn**，假定寄存器Rn为16位寄存器  
 $(SP) \leftarrow (Rn), SP \leftarrow (SP)-2$



## 1.3 寻址方式

### ❖ 堆栈寻址

- 出栈操作: **POP Rn**, 假定寄存器Rn为16位寄存器  
 $SP \leftarrow (SP) + 2, Rn \leftarrow ((SP))$



## 1.3 寻址方式 - 小结

### ❖ 指令操作数的寻址方式

➤ 形式地址、有效地址、寻址

寻址方式	操作数存放地	操作数有效地址EA	操作数的值 Operand	访问操作数所需访问次数
立即寻址	指令中	指令中的立即数字段	Imme. Data	0
寄存器直接寻址	寄存器	Rn	(Rn)	0
寄存器间接寻址	存储器	(Rn)	((Rn))	1
基址寻址	存储器	(Rb)+A	((Rb)+A)	1
变址寻址	存储器	(Rx)+A	((Rx)+A)	1
相对寻址	存储器	(PC)+A	((PC)+A)	1
堆栈寻址	存储器	(SP)	((SP))	1

# 第五讲：指令系统与MIPS汇编

## 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

## 二. **MIPS**指令系统

## 三. MIPS汇编语言

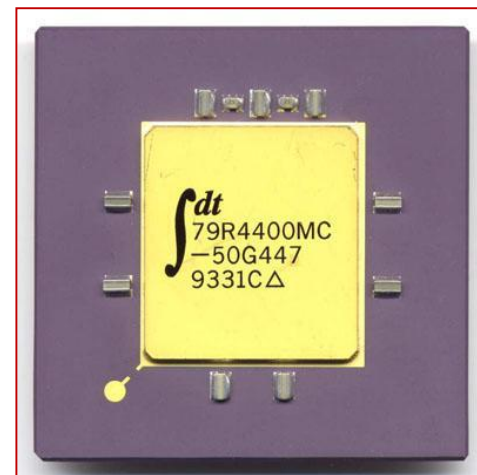
1. 概述
2. MIPS汇编指令和存储格式
3. MIPS汇编程序

## 2.2 MIPS 指令系统

### ❖ MIPS R系列CPU简介

- RISC (Reduced Instruction set Computer, 精简指令集计算机, RISC) 微处理器
- MIPS (Microprocessor without interlocked piped stages, 无内部互锁流水级的微处理器),
- 最早在80年代初由Stanford大学Patterson教授领导的研究小组研制出来, MIPS公司的R系列就是在此基础上开发的RISC微处理器。
- 1986年, 推出R2000 (32位)
- 1988年, 推出R3000 (32位)
- 1991年, 推出R4000 (64位)
- 1994年, 推出R8000 (64位)
- 1996年, 推出R10000
- 1997年, 推出R20000
- 通用指令体系MIPS I、MIPS II、MIPS III、MIPS IV到MIPS V, 嵌入式指令体系MIPS16、MIPS32到MIPS64, 发展已经十分成熟。在设计理念上MIPS强调软硬件协同提高性能, 同时简化硬件设计。

MIPS  
TECHNOLOGIES

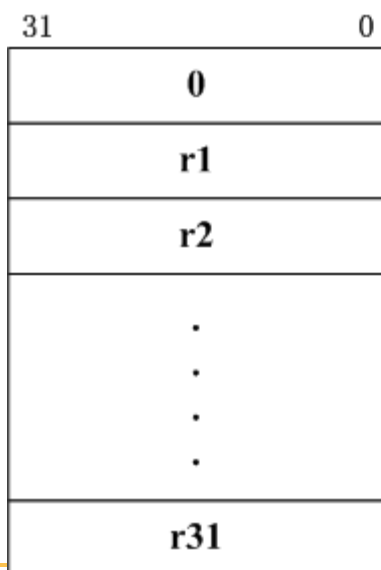


## 2.2 MIPS 指令系统

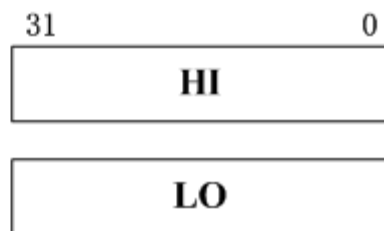
### ❖ MIPS R2000/R3000 寄存器结构

- 32位虚拟地址空间
- $32 \times 32$  bit GPRs, \$0~\$31
- $32 \times 32$  bit FPRs, \$f0~\$f31
- HI, LO, PC（相应的指令mfhi/mthi,mflo/mtlo来实现HI和LO寄存器与通用寄存器之间的数据交换）

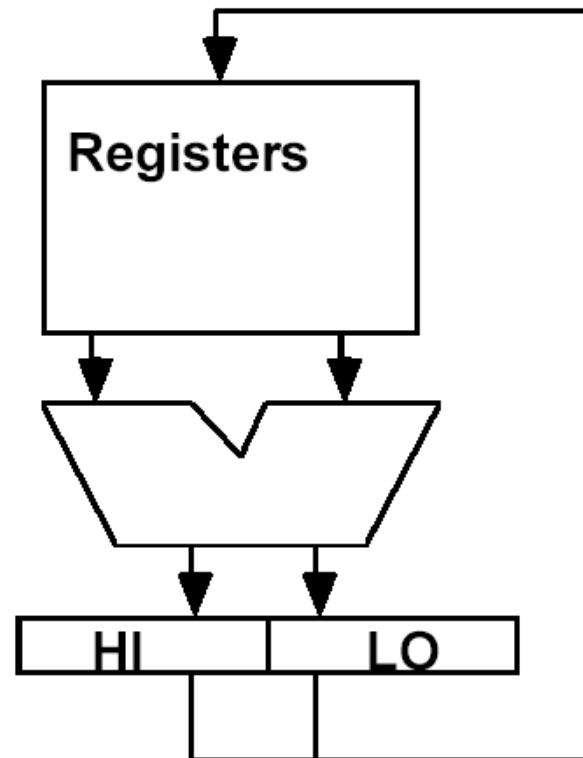
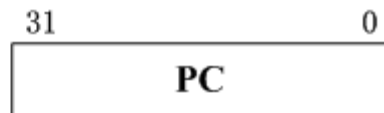
通用寄存器



乘/除寄存器



程序计数器



## 2.2 MIPS 指令系统

### ❖ MIPS 寄存器使用的约定

Name	Reg. Num	Usage
zero	0	constant value =0(恒为0)
at	1	reserved for assembler(为汇编程序保留)
v0 – v1	2 – 3	values for results(过程调用返回值)
a0 – a3	4 – 7	Arguments(过程调用参数)
t0 – t7	8 – 15	Temporaries(临时变量)
s0 – s7	16 – 23	Saved(保存)
t8 – t9	24 – 25	more temporaries(其他临时变量)
k0 – k1	26 – 27	reserved for kernel(为OS保留)
gp	28	global pointer(全局指针)
sp / s8	29	stack pointer (栈指针)
fp	30	frame pointer (帧指针)
ra	31	return address (过程调用返回地址)

Registers are referenced either by number—\$0...\$31, or by name —\$t0,\$s1...\$ra.

## 2.2 MIPS 指令系统

### ❖ MIPS指令格式

- MIPS只有3种指令格式，32位固定长度指令格式
  - R (Register) 类型指令：两个寄存器操作数计算，结果送寄存器
  - I (Immediate) 类型指令：使用1个16位立即数作操作数；
  - J (Jump) 类型指令：跳转指令，26位跳转地址
- 最多3地址指令：add \$t0, \$s1, \$s2 ( $\$t0 \leftarrow \$s1 + \$s2$ )
- 对于Load/Store指令，单一寻址模式：Base + Displacement
- 没有间接寻址
- 16位立即数
- 简单转移条件（与0比较，或者比较两个寄存器是否相等）
- 无条件码



## 2.2 MIPS 指令系统

### ❖ MIPS 指令格式

- Op: 6 bits, Opcode
- Rs: 5 bits, The first register source operand
- Rt: 5 bits, The second register source operand
- Rd: 5 bits, The register destination operand
- Shamt: 5 bits, Shift amount ( shift instruction)
- Func: 6 bits, function code ( another Opcode)

	6	5	5	5	5	6
R类型	Op	Rs	Rt	Rd	Shamt	Func
I类型	Op	Rs	Rt	16 bit Address or Immediate		
J类型	Op	26 bit Address ( for Jump Instruction)				

## 2.2 MIPS 指令系统

### ❖ MIPS寻址方式

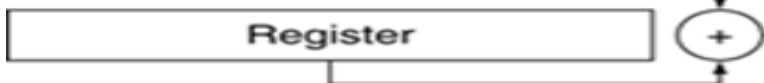
#### 1. 立即寻址



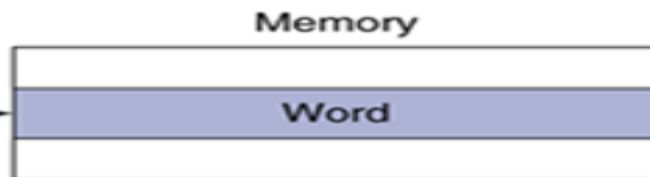
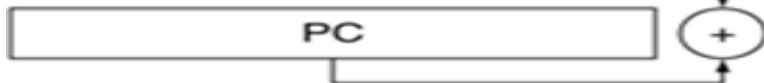
#### 2. 寄存器寻址



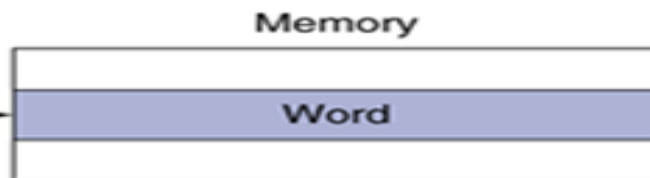
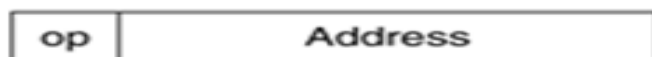
#### 3. 基址寻址



#### 4. PC相对寻址



#### 5. 伪直接寻址



## 2.2 MIPS 指令系统 —— 指令类型

### ❖ Load/Store（取数/存储）指令

- I类型指令，存储器与通用寄存器之间传送数据
- 支持唯一的存储器寻址方式：**Base+Index**
- 取数指令：**LB**（取字节）、**LBU**（取不带符号字节）、**LH**（取半字）、**LHU**（取不带符号的半字）、**LW**（取字）、**LWL**、**LWR**
- 存储指令：**SB**（存字节）、**SH**（存半字）、**SW**（存字）、**SWL**、**SWR**

### ❖ 运算指令

- R类型指令 和 I类型指令
- 算术运算：**add**, **addu**, **addi**, **addiu**, **sub**, **subu**, **mul**, **mulu**, **div**, **divu**, **mfhi**, **mflo**等
- 逻辑运算：**and**, **andi**, **or**, **ori**, **xor**, **xori**, **nor**等
- 移位指令：**sll**, **srl**, **sra**, **sllv**, **srlv**, **srav**等

## 2.2 MIPS 指令系统 —— 指令类型

### ❖ 跳转和转移指令：控制程序执行顺序

- 跳转指令：J类型指令（26位绝对转向地址）或R类型指令（32位的寄存器地址）
- 转移指令：I类型指令，PC-relative寻址方式，相对程序计数器的16位位移量（立即数）。
- 跳转：J、JAL、JR、JALR
- 转移：BEQ（相等转移）、BNE（不等转移）、BLEZ（小于或等于0转移）、BGTZ（大于0转移）、BLTZ（小于0转移）、BLTZAL、BGEZAL

### ❖ 特殊指令

- R类型指令
- 系统调用SYSCALL
- 断点BREAK

## 2.2 MIPS 指令系统

### ❖ R-Type指令编码示例

➤ 指令: **add \$t0, \$s1, \$s2 ;  $t0 \leftarrow (s1) + (s2)$**

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
指令格式	Op	Rs	Rt	Rd	Shamt	Func
指令编码	00000	10001	10010	01000	00000	10000

$$Rd \leftarrow (Rs) + (Rt)$$

- Op = 000000 (表示R-Type)
- Func = 100000 (表示add)
- Rs = 10001 (表示s1, \$17)
- Rt = 10010 (表示s2, \$18)
- Rd = 01000 (表示t0, \$8)
- Shamt=00000 (表示没有移位)

## 2.2 MIPS 指令系统 —— 指令示例

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
add	add \$1,\$2,\$3	$\$1 \leftarrow \$2 + \$3$	3 operation
subtract	sub \$1,\$2,\$3	$\$1 \leftarrow \$2 - \$3$	3 operation
add immediate	addi \$1,\$2,100	$\$1 \leftarrow \$2 + 100$	+ constant
multiply	mult \$2,\$3	$Hi,Lo \leftarrow \$2 \times \$3$	64-bit signed product
divide	div \$2,\$3	$Lo \leftarrow \$2 \div \$3$ $Hi \leftarrow \$2 \bmod \$3$	Lo = quotient Hi = remainder
move from Hi	mfhi \$1	$\$1 \leftarrow Hi$	Get a copy of Hi
move from Lo	mflo \$1	$\$1 \leftarrow Lo$	Get a copy of Lo
and	and \$1,\$2,\$3	$\$1 \leftarrow \$2 \& \$3$	Logical AND
or	or \$1,\$2,\$3	$\$1 \leftarrow \$2   \$3$	Logical OR
store	sw \$3,\$4(\$5)	$Mem(\$5+\$4) \leftarrow \$3$	Store Word
load	lw \$1,\$2(\$3)	$\$1 \leftarrow Mem(\$3+\$2)$	Load word
jump and link	jal 1000	$\$31 = PC + 4$ Go to 1000	Procedure call
jump register	jr \$31	Go to \$31	procedure return
set on less than	slt \$1,\$2,\$3	if ( $\$2 < \$3$ ) then $\$1 = 1$ else $\$1 = 0$	

## 2.2 MIPS 指令系统 —— SWAP: MIPS过程示例

swap:

```
addi  $sp,$sp, -12    ; Make room on stack for 3 registers
sw     $31, 8($sp)     ; Save return address
sw     $s2, 4($sp)     ; Save registers on stack
sw     $s3, 0($sp)
```

....



```
sll     $s2, $a1, 2      ; mulitply k by 4
addu    $s2, $s2, $a0    ; address of v[k]
lw      $t0, 0($s2)      ; load v[k]
lw      $s3, 4($s2)      ; load v[k+1]
sw      $s3, 0($s2)      ; store v[k+1] into v[k]
sw      $t0, 4($s2)      ; store old v[k] into v[k+1]
```

```
lw      $s3, 0($sp)      ; Restored registers from stack
lw      $s2, 4($sp)
lw      $31, 8($sp)      ; Restore return address
addi    $sp,$sp, 12      ; restore top of stack
jr      $31              ; return to place that called swap
```



## 第五讲：指令系统与MIPS汇编

### 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

### 二. MIPS指令系统

### 三. MIPS汇编语言（自学）

1. 概述
2. MIPS汇编指令和存储格式
3. MIPS汇编程序



# CPU和指令集

- ❖ 执行指令是**CPU**的主要工作
- ❖ 不同的**CPU**有不同的指令集
  - 指令集架构 *Instruction Set Architecture (ISA)*.
  - Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...
- ❖ 精简指令集(**RISC**)的哲学
- ❖ **MIPS** – 最早一家生产出商用 **RISC** 架构的半导体公司
  - **MIPS** 简单、优雅，不被细节所累
  - **MIPS** 在嵌入式中广泛应用, **x86** 很少应用到嵌入式市场, 它更多的是应用到**PC**上



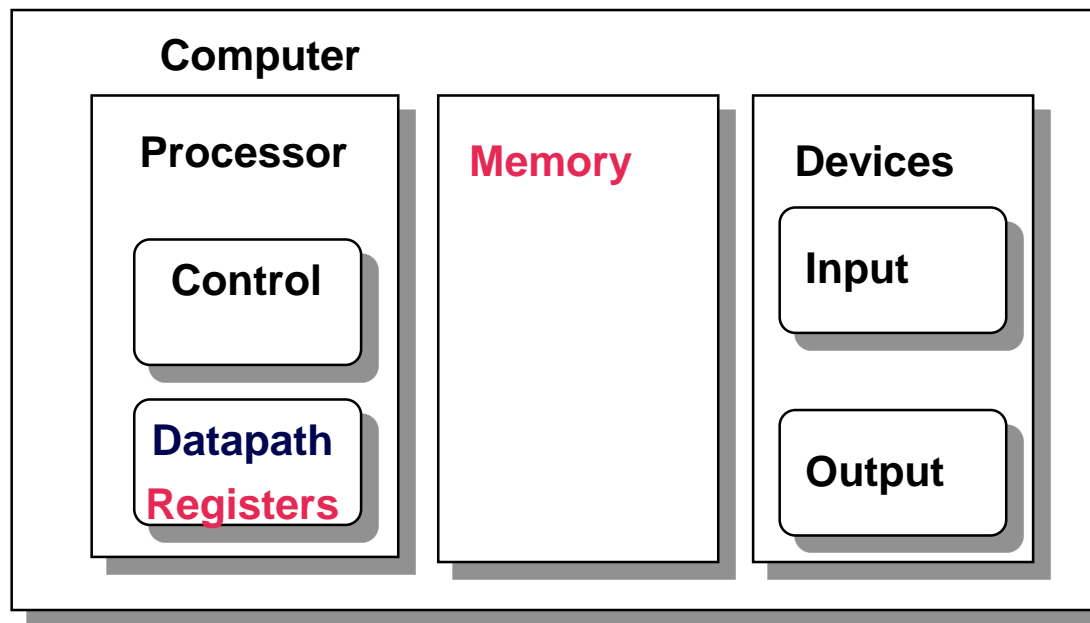
Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

# 计算机系统的组成结构

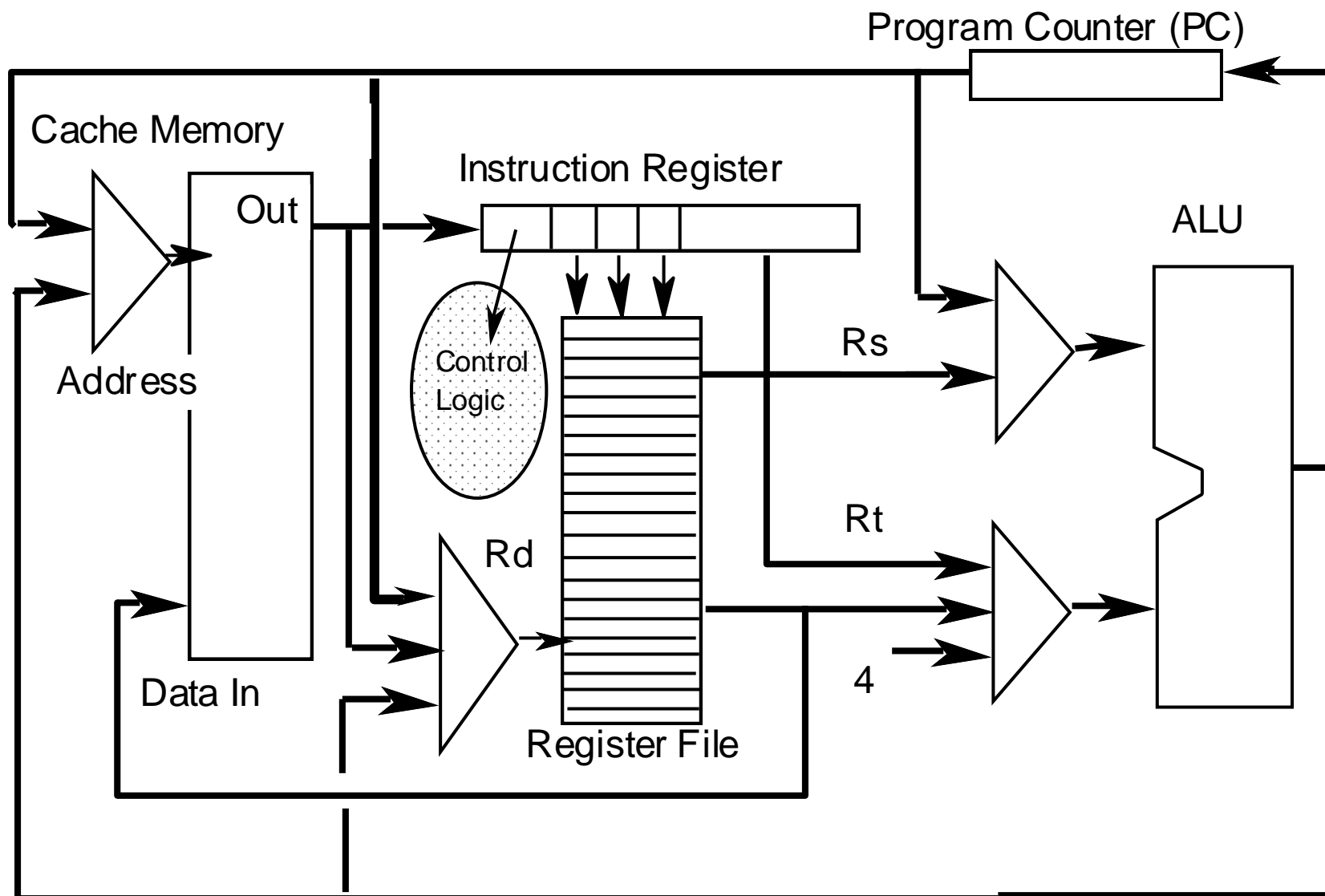
## ❖ 计算机系统的组成

- 控制
- 运算
- 存储
- 输入/输出

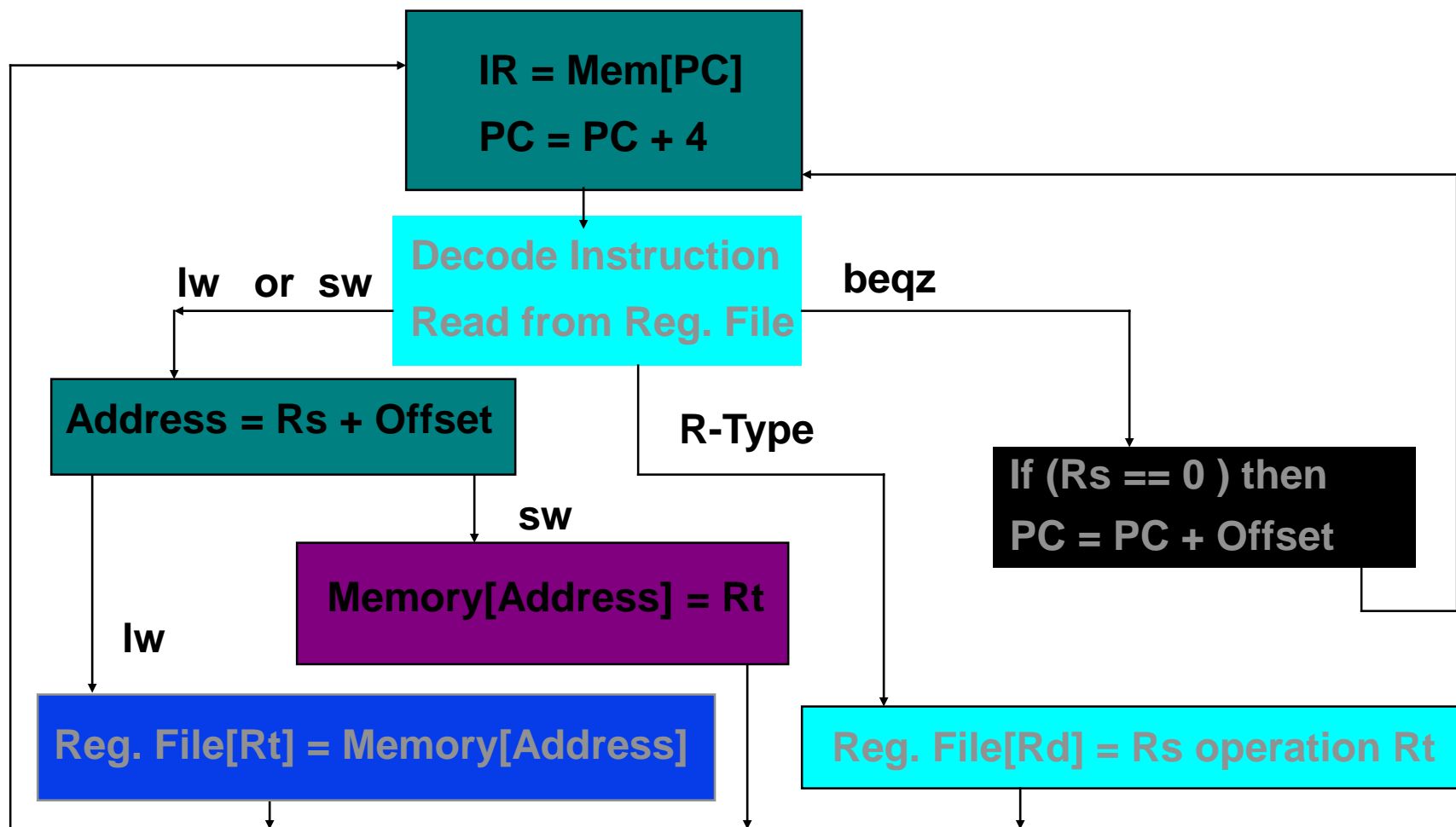
## ❖ 寄存器是数据通路的一部分



# 数据通路



# 寄存器传送的控制逻辑



# 内存布局

❖ **Text:** 程序代码段

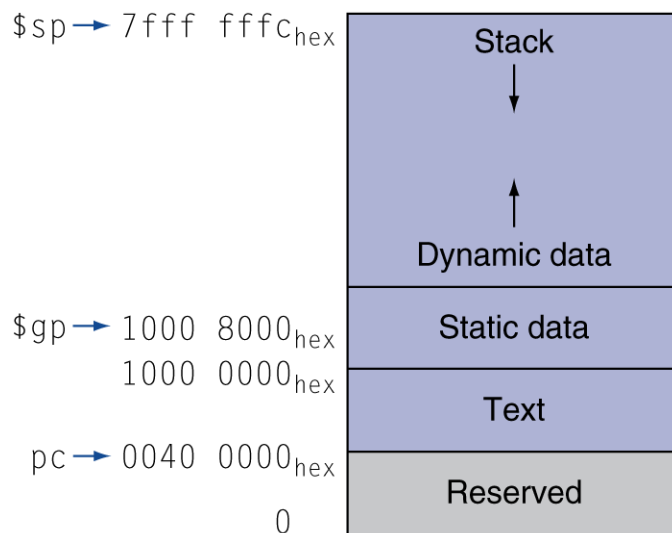
❖ **Static data:** 全局变量

- 例如, C语言中的静态变量, 常数数组和串
- **\$gp** 寄存器初始地址±偏移量寻址本段内存

❖ **Dynamic data:** 堆

- 例如, C中的**malloc**, Java中的**new**

❖ **Stack:** 栈, 自动存储区



# 寻址模式回顾

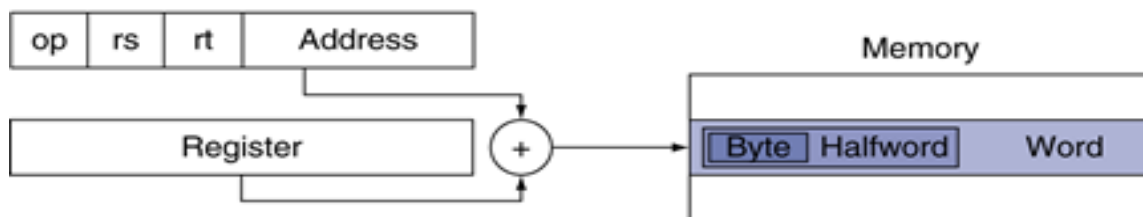
## 1. 立即寻址



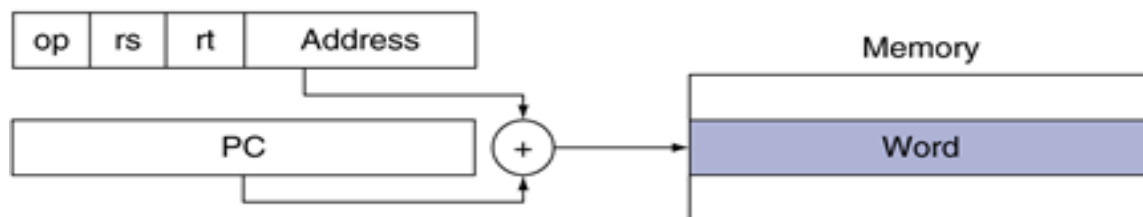
## 2. 寄存器寻址



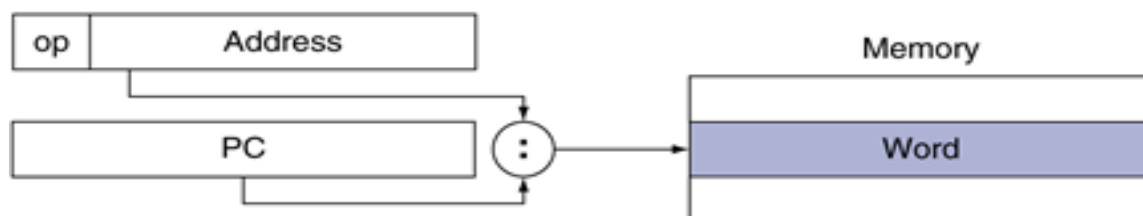
## 3. 基址寻址



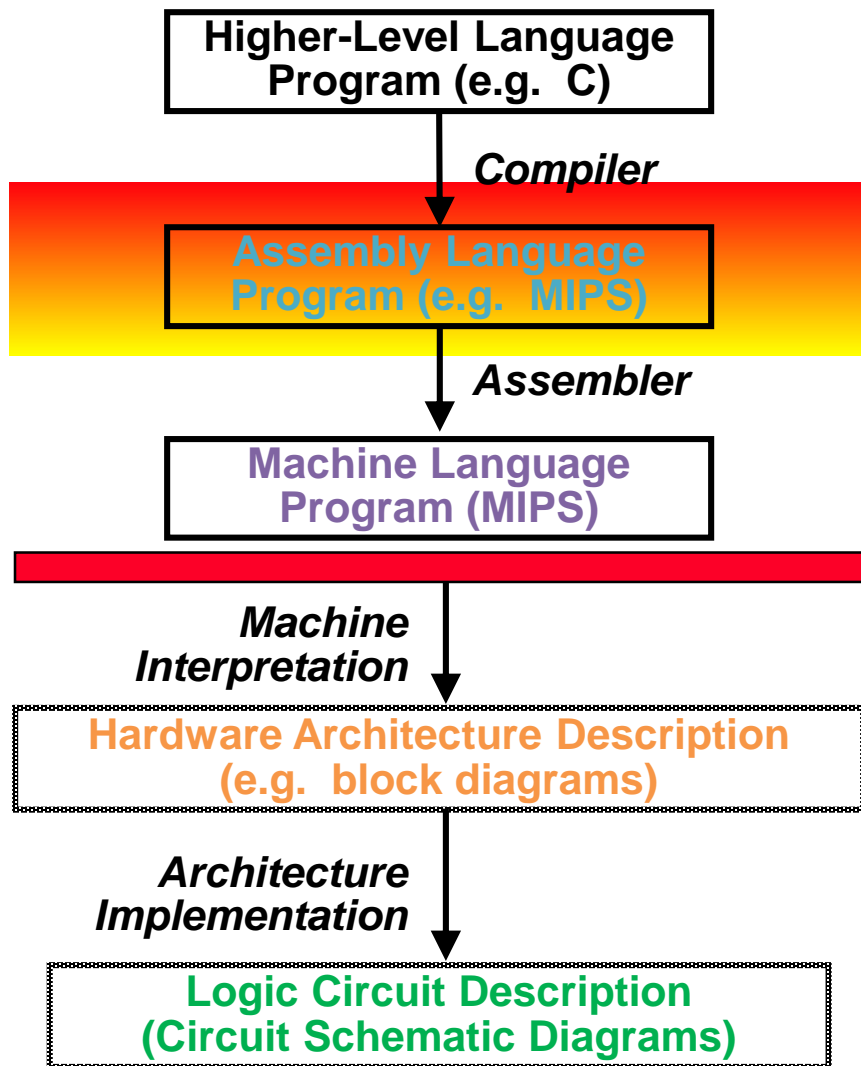
## 4. PC相对寻址



## 5. 伪直接寻址



# 汇编语言在层次结构中的位置

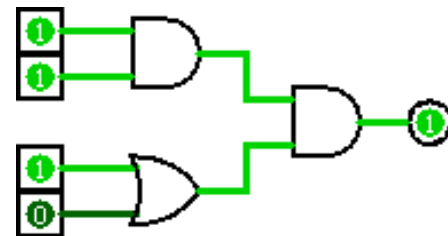
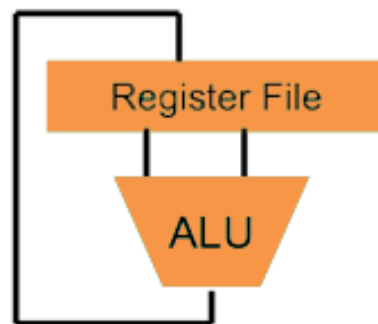


```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
Lw  $t0, 0($2)  
Lw  $t1, 4($2)  
Sw  $t1, 0($2)  
Sw  $t0, 4($2)
```

更紧凑、更高效的代码  
有时比编译优化更有效

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



# MIPS 汇编语言程序示例

<u>Label</u>	<u>Op-Code</u>	<u>Dest.</u>	<u>S1,</u>	<u>S2</u>	<u>Comments</u>
	move	\$a0,	\$0		# \$a0 = 0
	li	\$t0,	99		# \$t0 = 99
loop:					
	add	\$a0,	\$a0, \$t0		# \$a0 = \$a0 + \$t0
	addi	\$t0,	\$t0, -1		# \$t0 = \$t0 - 1
	bnez	\$t0,	loop		# if (\$t0 != zero) branch to loop
	li	\$v0,	1		# Print the value in \$a0
	syscall				
	li	\$v0,	10		# Terminate Program Run
	syscall				



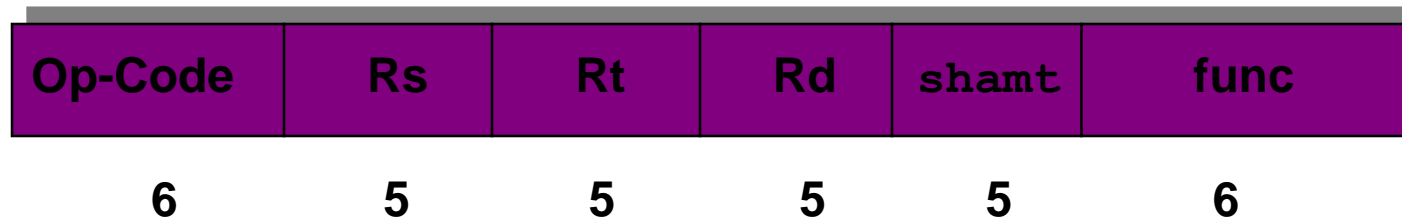
# MIPS指令集

---

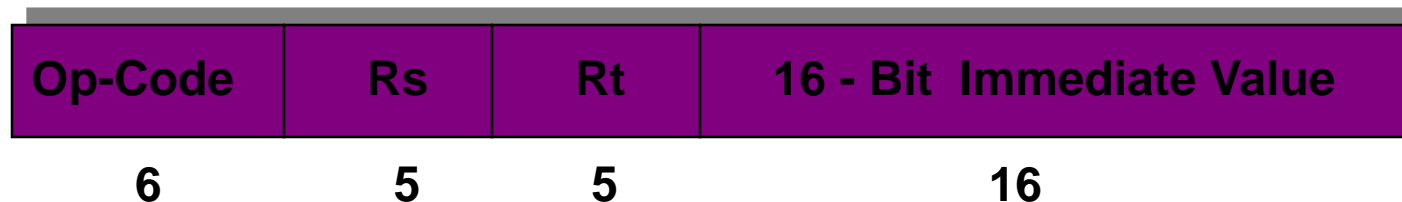
- ❖ 算术、逻辑和移位指令
- ❖ 存/取指令
- ❖ 条件分支指令
- ❖ 函数调用指令

# MIPS指令字格式

## ❖ R- Format ( Register )



## ❖ I- Format ( Immediate )



## ❖ J- Format ( Jump )



## 伪指令

---

❖取地址	<b>la     \$s0, table</b>
❖取立即数	<b>li     \$v0, 10</b>
❖移动	<b>move \$t8, \$sp</b>
❖乘	<b>mul   \$t2, \$a0, \$a1</b>
❖除	<b>div   \$s1, \$v1, \$t7</b>
❖求余	<b>rem   \$s2, \$v1, \$t7</b>
❖取反	<b>neg   \$s0, \$s0</b>

## 寄存器命名规范约定

**\$0** : 常量0

**\$v0** : 函数返回值

**\$a0** : 函数传递参数

**\$t0** : 临时变量寄存器

**\$s0** : 保存寄存器

**\$sp** : 栈指针

**\$ra** : 返回地址

# 第五讲：指令系统与MIPS汇编

## 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

## 二. 典型指令系统介绍

1. 8086/8088指令系统
2. MIPS指令系统
3. CISC与RISC

## 三. MIPS汇编语言

1. 概述
2. **MIPS**汇编指令和指令字
3. MIPS汇编程序

# MIPS指令语法、变量和注释

## ❖ 指令语法

操作符, 目标, 源1, 源2

1) 操作名称 (“操作符”) ; 2) 操作结果 (“目标”)

3) 1st 操作数 (“源1”) ; 4) 2nd 操作数 (“源2”)

➤ 语法是固定的, 通过约定好的统一的规则使硬件实现更简单

## ❖ 汇编语言中, 每一条语句就是执行指令集中的一条简单指令

## ❖ 为保持硬件的简单, 汇编语言不使用变量, 操作数都是寄存器(registers)

➤ 在汇编语言里, 寄存器没有数值类型 (注意与C等高级程序语言的不同);

➤ 寄存器是直接由硬件实现的, 速度很快!

➤ 寄存器是直接由硬件上实现的, 固定的数目, 资源受限!

## ❖ MIPS有32个寄存器, 每一个寄存器的宽度为32bits (字)

➤ 从0到31给32个寄存器编号, 除了有编号外, 每个寄存器还有自己的名字

➤ 按照编号的引用方式: \$0, \$1, \$2, ... \$30, \$31

➤ 约定: 每个寄存器都取一个名字以便写代码时更方便

\$16 - \$23 ➔ \$s0 - \$s7, (与C语言中的变量对应)

\$8 - \$15, \$24-\$25 ➔ \$t0 - \$t7, \$8-\$9, (与临时变量对应)

## ❖ 通常情况下使用名字来指定寄存器, 以增加代码的可读性

➤ 另一个增加代码可读性的方法: 注释, # 被用来做MIPS的注释

# MIPS汇编中的算术、逻辑和移位运算

## ❖ MIPS 整数加/减法

➤ 加法: `add $s0,$s1,$s2` (in MIPS), 相当于: `a = b + c` (in C)

➤ 减法: `sub $s3,$s4,$s5` (in MIPS), 相当于: `d = e - f` (in C)

➤ C 语言中一条语句 `a=b+c+d-e`, 需拆成多条汇编指令

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

➤ 另一条C语言的语句 `f=(g+h)-(i+j)`, 使用临时变量寄存器

```
add $t0,$s1,$s2 # temp = g + h
add $t1,$s3,$s4 # temp = i + j
sub $s0,$t0,$t1 # f=(g+h)-(i+j)
```

## ❖ 寄存器 Zero和立即数

➤ 定义寄存器 `zero` (`$0` or `$zero`), 表示数字0

```
add $s0,$s1,$zero (in MIPS); f = g (in C)
```

➤ 立即数是数值常量, 针对立即数设置专门指令

例如, 立即数加: `addi $s0,$s1,10` (in MIPS); `f=g+10` (in C)

➤ 语法与`add`指令类似, 除了最后一个参数用数值代替了寄存器

➤ MIPS中没有立即数的减法, 用立即数加实现

```
addi $s0,$s1,-10 (in MIPS); f=g-10 (in C)
```

# MIPS汇编中的算术、逻辑和移位运算

## ❖ 算术运算中的溢出

- 发生溢出是由于计算机有限的数值表示引起的
- 有些语言会自动检测出异常(Ada), 有些不会(C)

## ❖ MIPS有2种加减运算指令，每一种指令又有两种数值方式

- 下面的 可以检测出溢出异常

- add (add)
- add immediate (addi)
- subtract (sub)

- 下面的 不会检测出溢出异常

- add unsigned (addu)
- add immediate unsigned (addiu)
- subtract unsigned (subu)

- 编译器会自动挑选合适的运算指令类型

- MIPS中的C编译器会使用  
addu, addiu, subu

不检查溢出异常



# MIPS汇编中的算术、逻辑和移位运算

## ❖ 位操作

- 把寄存器中的值拆开来，看成是32个单独的1位二进制数值
- 逻辑和移位操作

## ❖ 逻辑操作

- 两种基本的逻辑操作符

**AND:** 当两个数都为1时输出1； **OR:** 至少有一个为1时输出1；按位与、按位或

- 逻辑指令的语法

操作符（指令名） 结果寄存器，操作数1（寄存器），操作数2（寄存器/立即数）

and, or: 操作数2是寄存器； andi, ori: 操作数2是立即数

## ❖ 移位操作

- 将一个字的所有位向左或向右移动一定的位数

- 移位指令语法

操作符（指令名） 结果寄存器，操作数1（寄存器），移位量（<32的常量/寄存器）

- MIPS移位指令

**sll** (逻辑左移): 左移并且补0，位移量为立即数；

**srl** (逻辑右移): 右移并且补0，位移量为立即数；

**sra** (算术右移): 右移并且在空位做符号扩展填充，位移量为立即数；

**sllv, srlv, srav**, 移位量存储在寄存器中，处理方式与立即数位移量类似

# MIPS汇编中的数据存取

- ❖ **MIPS** 算术指令只能操作寄存器, 不能直接操作内存
- ❖ **数据存取指令**在内存与寄存器之间传输数据
- ❖ **内存到寄存器: Load**
  - 要指定访问内存的具体地址, 需要两个数据值  
指向内存某地址的指针和数字偏移量, 内存地址通常由这两个值相加得到
  - **Load** 指令语法  
操作码, 寄存器, 数值偏移量(寄存器)
  - **Load**指令操作码: **lw**  
例子:**lw \$t0, 12(\$s0)**; **\$s0** 称为基址寄存器, **12** 称为偏移量
- ❖ **寄存器到内存: Store**
  - 将寄存器中的数值写回内存中去
  - **Store**与**Load**指令的语法格式是完全一样的  
操作码, 寄存器, 数值偏移量(寄存器)
  - **Store**指令操作码: **sw**  
例子:**sw \$t0, 12(\$s0)**
  - “**Store** 进内存”

# MIPS汇编中的数据存取

## ❖ 指针 vs. 值

- 一个寄存器中可以存储**32-bit**的数值
  - **(signed) int, unsigned int, pointer** (内存地址), 或其它类型
  - 要确保指令操作的数值类型是有意义的

## ❖ 寻址: 字节 vs. 字

- 内存中的每一个字都有一个地址, 类似于一个数组的索引
- 现代计算机按字节编址, **32-bit (4 bytes)** 字地址按 **4**递增

■ **Memory[0], Memory[4], Memory[8], ...**

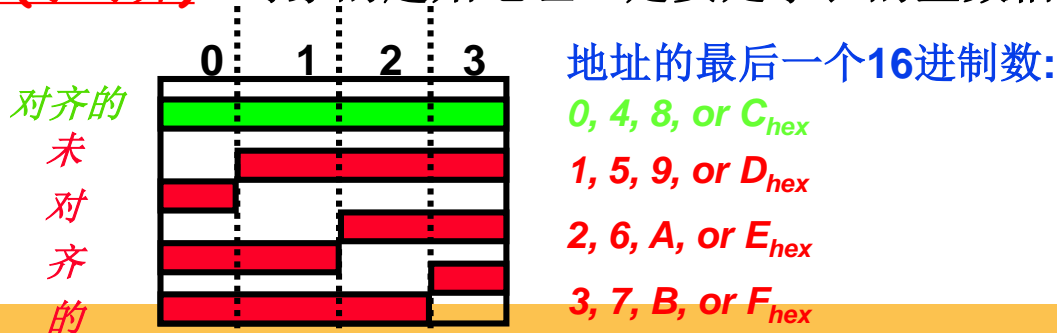
例子:  $g = h + A[5]$ ;  $g: \$s1, h: \$s2, \$s3: A$ 的基地址

`lw $t0, 20($s3) # $t0 gets A[5]`

`add $s1, $s2, $t0 # $s1 = h + A[5]`

## ❖ 字对齐

- 无论是 **lw** 还是 **sw**, 基址寄存器的值与偏移量的和始终应该是**4的倍数**
- **Alignment(字对齐)**: 对象的起始地址一定要是字长的整数倍



# MIPS汇编中的数据存取

## ❖ 寄存器vs. 内存

- 变量数比寄存器数多怎么办?
- 编译器会将最经常使用的变量保留在寄存器中, 不常使用的放在内存中

## ❖ 字节的存/取

- 除了需要在内存与寄存器之间按字传送 (**lw**, **sw**)外, **MIPS** 还有按字节传送的指令。读字节: **lb**; 写字节: **sb**
- 与**lw**, **sw**格式相同

例如: **lb \$s0, 3(\$s1)**

把内存中的某个地址(“3” + *s1* 中的地址值) 所存储的数值拷贝到*s0*的**低地址字节**上.

那么**32**位中的其余**24**位填充什么呢?

**lb**: 使用位扩展(或称为符号扩展)填充剩余**24**位

有些情况下我们不想使用位扩展(如**char**类型)

**MIPS** 不使用位扩展的指令: **lbu**

# MIPS汇编中的分支和循环

❖ 要实现一个真正的计算机,除了操作数据还需要代码能够做判断、决定和转跳...

➤ C 和 MIPS都提供 labels标签 用来支持 “goto”

C: 使用goto是非常可怕的; MIPS: 必须有goto!

❖ MIPS 判断指令（分支）

➤ 条件分支

▪ `beq register1, register2, L1;`

相当于C当中: `if (register1==register2) goto L1`

▪ `bne register1, register2, L1`

相当于C当中: `if (register1!=register2) goto L1`

➤ 无条件分支

▪ `J label;` 跳转到标签label所在的代码, 不需要满足任何条件

相当于C当中: `goto label`

▪ 从技术的角度来说, 等同于 `beq $0,$0,label`

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

```
f: $s0 ; g: $s1; h: $s2; i: $s3; j: $s4
```

```
beq $s3,$s4,True # branch i==j
sub $s0,$s1,$s2  # f=g-h(false)
j    Fin         # goto Fin
True:add $s0,$s1,$s2 # f=g+h (true)
```

Fin:

# MIPS汇编中的分支和循环

❖ 考察C中循环的例子; 假定A[] 是整型数组

do {  
重写成

<code>g = g + A[i];</code>	<code>Loop: g = g + A[i];</code>
<code>i = i + j;</code>	<code>i = i + j;</code>
<code>} while (i != h);</code>	<code>if (i != h) goto Loop;</code>

假定g, h, i, j和A[]的基址分别对应\$s1, \$s2, \$s3, \$s4, \$s5  
对应的MIPS 代码:

```
Loop: sll $t1,$s3,2    #$t1= 4*i
      add $t1,$t1,$s5  #$t1=addr A
      lw  $t1,0($t1)   #$t1=A[i]
      add $s1,$s1,$t1  #g=g+A[i]
      add $s3,$s3,$s4  #i=i+j
      bne $s3,$s2,Loop# goto Loop if i!=h
```

❖ C中有3种循环结构

- While; do... while; for
- 每一种都可以用另外两种中的任一种等价表达, 上面的例子同样适用于 while 和 for 循环

# MIPS汇编中的分支和循环

## ❖ MIPS 不等式指令

➤ 语法: `slt reg1, reg2, reg3`; 含义: “Set on Less Than”

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

➤ 如何表达: `if (g < h) goto Less`;

假定 `g:$s0, h:$s1`

```
slt $t0, $s0, $s1 # $t0 = 1 if g < h
bne $t0, $0, Less # goto Less if $t0 != 0
```

➤ `bne` 和 `beq` 通常在 `slt` 指令后用寄存器0来做分支判断

使用 `slt` ➔ `bne` 指令对表示 `if (... < ...) goto ...`

如何表达 `>`, `≥` ? 使用 `slt` ➔ `beq` 指令对表示 `if (... ≥ ...) goto ...`

## ❖ 不等式中的立即数和无符号数

➤ `slt` 指令的立即数版本: `slti`

➤ 无符号数不等式指令: `sltu, sltiu`

## 例子: C语言 Switch 语句

❖ 根据k的取值从4个选项中选择1个，C代码如下：

```
switch (k) {  
    case 0: f=i+j; break; /* k=0 */  
    case 1: f=g+h; break; /* k=1 */  
    case 2: f=g-h; break; /* k=2 */  
    case 3: f=i-j; break; /* k=3 */  
}
```

❖ 先简化成if-else语句链

```
if(k==0) f=i+j;  
else if(k==1) f=g+h;  
else if(k==2) f=g-h;  
else if(k==3) f=i-j;
```

变量映射: f:\$s0, g:\$s1, h:\$s2, i:\$s3, j:\$s4, k:\$s5

❖ MIPS 代码如下

```
        bne $s5,$0,L1      # branch k!=0  
        add $s0,$s3,$s4    #k==0 so f=i+j  
        j Exit             # end of case so Exit  
L1:     addi $t0,$s5,-1     # $t0=k-1  
        bne $t0,$0,L2      # branch k!=1  
        add $s0,$s1,$s2    #k==1 so f=g+h  
        j Exit             # end of case so Exit  
L2:     addi $t0,$s5,-2     # $t0=k-2  
        bne $t0,$0,L3      # branch k!=2  
        sub $s0,$s1,$s2    #k==2 so f=g-h  
        j Exit             # end of case so Exit  
L3:     addi $t0,$s5,-3     # $t0=k-3  
        bne $t0,$0,Exit     # branch k!=3  
        sub $s0,$s3,$s4    #k==3 so f=i-j  
Exit:
```



# 第五讲：指令系统与MIPS汇编

## 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

## 二. MIPS指令系统

## 三. MIPS汇编语言

1. 概述
2. **MIPS**汇编指令和指令字
3. MIPS汇编程序

# 函数

```
main() {  
    int i,j,k,m;  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}  
  
/* really dumb mult function */  
int mult (int mcand, int mlier){  
    int product;  
    product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier -1; }  
    return product;  
}
```

哪些信息是编译器和程序员需要  
踪和记录的？

什么指令可以实现这样的功能？

❖ 用寄存器来记录函数调用信息

❖ 寄存器规范

➤ 返回地址

\$ra

➤ 参数

\$a0, \$a1, \$a2, \$a3

➤ 返回值

\$v0, \$v1

➤ 局部变量

\$s0, \$s1, ... , \$s7

❖ 还会用到栈！

## 支持函数功能的指令

### ❖ 可同时执行转跳和存储返回地址的指令

➤ `jal, (jump and link)`

➤ 用 `jal` 使得函数调用更快，同时不需要了解代码读入内存的地址细节

### ❖ `jal` 与 `j` 的语法相同

`jal label`

### ❖ `jal` 执行步骤其实应该是 ‘`laj`’ (`link and jump`)

➤ 第一步 (`link`): 将下一条指令地址存入 `$ra` (为什么下一条?)

➤ 第二步 (`jump`): 向给定的 `label` 转跳

### ❖ 寄存器转跳指令 `jr`，可转跳至寄存器中存储的地址

`jr register`

➤ 在函数调用中非常有用

`jal` 指令将返回地址存储在寄存器 (`$ra`) 中

`jr $ra` 跳回该地址

## 嵌套调用

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

- ❖ **sumSquare**被调用，而**sumSquare**又调用**mult**
  - **\$ra**中存储着 **sumSquare** 的返回地址，但是会在调用 **mult**时重写
  - 需要在调用**mult**之前存储 **sumSquare** 返回地址
- ❖ 需要在**\$ra**之外存储相关信息！
- ❖ 当一个 **C** 程序运行时，有**3**块重要的内存区域被分配
  - 静态区 (**static**)：存储静态变量，一旦程序声明，直到程序执行结束才会清除，比如**C**程序的全局变量
  - 堆 (**heap**)：动态声明的变量
  - 栈 (**stack**)：程序执行过程中使用的空间，可用来存储寄存器值



# 使用栈

- ❖ 寄存器\$sp始终指向栈空间最后被使用的位置——栈指针
- ❖ 使用栈的时候，对该指针减去需要的空间量，并向该空间填写信息
- ❖ 刚才的C例子

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

```

sumSquare:

“push”

```
addi $sp,$sp,-8    # space on stack  
sw $ra, 4($sp)     # save ret addr  
sw $a1, 0($sp)     # save y
```

```
add $a1,$a0,$zero  # prep args  
jal mult           # call mult
```

“pop”

```
lw $a1, 0($sp)     # restore y  
add $v0,$v0,$a1    # mult()+y  
lw $ra, 4($sp)     # get ret addr  
addi $sp,$sp,8     # restore stack  
jr $ra
```

mult: ...

# 调用规则

## ❖ 调用的步骤

- 将需要保存的值压入栈
- 如果需要的话, 指定参数
- `jal` 调用
- 从栈中恢复相关的值

## ❖ 调用过程中的规则

- 通过 `jal` 指令调用, 使用一个 `jr $ra` 指令返回
- 最多可接受4个入口参数, `$a0, $a1, $a2, $a3`
- 返回值通常在 `$v0` 中(如果需要, 可以使用 `$v1`)
- 必须遵守寄存器使用规范 (即使是在那些只有你自己调用的函数中)!

```
entry_label:
    addi $sp, $sp, -framesize
    sw $ra, framesize-4($sp)    # save $ra
    save other regs if need be
```

## 一个函数的基本结构

*Body ... (call other functions...)*

```
restore other regs if need be
lw $ra, framesize-4($sp)    # restore $ra
addi $sp, $sp, framesize
jr $ra
```

# MIPS 寄存器分配

寄存器 0	\$0	\$zero
汇编器预留	\$1	\$at
返回值	\$2-\$3	\$v0-\$v1
参数	\$4-\$7	\$a0-\$a3
临时	\$8-\$15	\$t0-\$t7
保存	\$16-\$23	\$s0-\$s7
临时	\$24-\$25	\$t8-\$t9
内核占用	\$26-\$27	\$k0-\$k1
全局指针	\$28	\$gp
栈指针	\$29	\$sp
帧指针	\$30	\$fp
返回地址	\$31	\$ra

- **\$at**: 编译器随时可能使用，最好不要用
- **\$k0-\$k1**: 操作系统随时会使用，最好不要用
- **\$gp, \$fp**: 可以不用理会

## 寄存器规范

- ❖ **寄存器规范**: 一套规则——在执行了一个函数调用(**jal**)后, 哪些寄存器的值要保证不变, 哪些可能已经变了
- ❖ **保存寄存器**
  - **\$0**: 不能改变, 永远是0
  - **\$s0-\$s7**: 如果被修改了需要恢复。如果被调用函数由于各种原因改变了这些值, 它必须在返回之前将这些寄存器的原始值恢复
  - **\$sp**: 如果被修改了需要恢复。栈指针在**jal** 执行之前和之后必须是指向的同一个地址, 不然调用函数就无法从栈上正常恢复数据了
  - **HINT** – 所有保存寄存器都以 **S** 开头!
- ❖ **易变寄存器**
  - **\$ra**: 会改变。**jal** 会自动更改这个寄存器值, 调用函数需要将其值保存在栈上
  - **\$v0-\$v1**: 会改变。始终保存最新的返回值
  - **\$a0-\$a3**: 会改变。调用函数如果在调用完成后还要用到这些寄存器中的值, 就要在调用前将这些值保存在自己的栈空间内
  - **\$t0-\$t9**: 会改变。任何函数在任何时候都可以更新这些寄存器中的值, 调用函数如果在调用完成后还要用到这些寄存器中的值, 就要在调用前将这些值保存在自己的栈空间内
- ❖ 调用和被调用都只需要保存他们各自使用的临时和保存寄存器值, 并非所有的寄存器都要保存



# 存储程序概念

## ❖ 冯诺依曼计算机建立在两个大原则之上

- 指令与数值的表示形式一模一样
- 全部程序可以被存储在内存中，像数据一样被读写

## ❖ 简化计算机系统的软/硬件

- 用于数据操作的内存技术完全适用于指令操作

## ❖ 导致的结果

### ➤ 编址

- 所有存储在内存中的东西都有一个地址，分支与跳转语句的执行正是基于此
- 对地址的随意使用会导致很难查找的bug
- 有一个寄存器始终保存正在执行的指令地址：“Program Counter” (**PC**)，从根本上说就是一个指向内存的指针

### ➤ 二进制代码兼容性

- 程序以二进制的形式给出，程序与特定的指令集绑定
- 新机器想要运行旧程序（“二进制代码”）时，必须将程序按照新的指令集进行编译
- 导致“向后兼容”的指令集不断进化

# 作为指令的数字

- ❖ 现在我们处理的所有的数据都是按字来分配的 (32位字长)
  - 每个寄存器是一个字
  - **lw**, **sw** 每次只能访问内存中的一个字
- ❖ 如何来表示指令呢?
  - 计算机只认识1和0, 所以 “**add \$t0,\$0,\$0**” 对计算机来说没有意义
  - **MIPS** 追求简单: 数据是按字存放的, 指令也按字存放吧!
- ❖ 一个字有32位, 我们把一个字分成几个“字段”(“**fields**”)
  - 每个“字段”用来提供指令的一部分信息
- ❖ 可以定义不同的分配“字段”的方法, **MIPS** 基于简单原则, 定义了以下3种指令格式的基本类型
  - **I-format**(立即数格式)  
当指令中有立即数的时候使用, 包括**lw**、**sw** (偏移量是立即数) 以及分支语句 (**beq** and **bne**)。 (但是这种格式**不包含“移位”指令**)
  - **J-format**(跳转指令格式)  
**j**, **jal**
  - **R-format**(寄存器格式)  
适用于其他的指令

# R-Format 指令

❖ 以位为单位定义各个“字段”的大小

Opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
------------	--------	--------	--------	-----------	-----------

每个字段都被看成是**5-bit (0-31)** 或**6-bit (0-63)** 的无符号整数, 而不是一个**32-bit**整数的一部分

- opcode: 与其他字段结合决定指令(等于0时代表所有R-Format指令)
- funct: 与opcode组合起来, 决定该条指令名(操作符)
- rs (**Source Register**): 通常指定存放第一个操作数的寄存器
- rt (**Target Register**): 通常指定存放第二个操作数的寄存器
- rd (**Destination Register**): 通常指定存放计算结果的寄存器
- shamt: 这个字段中存储执行移位运算时要移的位数(该字段在不进行移位操作的指令中通常会置0)

注意3个寄存器字段:

- 每个寄存器字段是**5-bit**, 可以用它来完整的表示出**0-31**之间的所有无符号整数, 这样每一个寄存器字段中的数值就是对应的**32**个寄存器中的一个

# R-Format 指令的例子

## ❖ MIPS 指令

add \$8, \$9, \$10

opcode = 0

funct = 32

rd = 8 (目标结果)

rs = 9 (第一 **操作数**)

rt = 10 (第二 **操作数**)

shamt = 0 (非移位指令)

每个字段的  
十进制表示

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

每个字段的二进制表示

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

hex

十六进制表示:

012A 4020<sub>16</sub>

十进制表示:

19,546,144<sub>10</sub>

称为 机器语言指令

# I-Format 指令

## ❖ 带立即数的指令

### I-Format 指令

opcode (6)	rs (5)	rt (5)	immediate (16)
------------	--------	--------	----------------

只有一个字段与R-format不同, **opcode** 还在原来的位置不变

➤ **opcode**: 因为有了 **funct** 字段, **opcode** 在 **I-format** 指令中可以唯一确定一条指令 (**R-format** 用2个6-bit的字段而非一个12-bit字段来确定一条指令的原因: 为了与其他指令格式保持一致)

➤ **rs**: 表示 **唯一** 的操作数寄存器(如果有的话)

➤ **rt**: 存储计算结果的寄存器(**target register**)

➤ 立即数字段

- ✓ **addi, slti, sltiu**, 立即数通过**位扩展(符号扩展)**的方式扩成32 位
- ✓ **16 bits** ➔ 可以表示出 $2^{16}$ 个不同的整数值
- ✓ 这么大的立即数在处理一些特别的指令(如**lw** 或 **sw**)时已经足够了, 即使用**slti**指令, 在大多数情况下也是没有问题的

# I-Format 指令的例子

## ❖ MIPS 指令

`addi     $21, $22, -50`

`opcode = 8`

`rs = 22` (保存操作数的寄存器)

`rt = 21` (目标寄存器, 存储结果值用)

`immediate = -50` (默认为十进制)

十进制表示

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

二进制表示

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

十六进制表示:     `0x22D5 FFCE16`

十进制表示:        `584,449,99810`

# I-Format 指令的问题

## ❖ 立即数太大怎么办？

- 当需要的立即数在其字段内可以表示的时候，`addi`, `lw`, `sw` 和 `slti` 指令执行时都没有问题
- 但是如果太大，字段无法表示怎么办？在使用任何一个I-Format指令时，我们都必须考虑：如果立即数是一个32-bit的数值该怎么办？

## ❖ 解决方案

- 使用软件技巧 + 新的指令
- 不改变现有指令：只要加入一条新指令来帮忙

## ❖ 新指令： `lui register, immediate`

- Load Upper Immediate
- 将一个16-bit的立即数存入寄存器的高16位，将寄存器的低16位全部置0

这样就没  
问题啦

例子: `addi $t0, $t0, 0xABABCD`  
改为: `lui $at, 0xABAB`  
`ori $at, $at, 0xCDCD`  
`add $t0, $t0, $at`

立即数太大，  
这条指令根本  
放不进去

- 每条I-format指令只有16-bit用来存放立即数

## 使用I-Format的分支语句: 程序计数器相对寻址

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

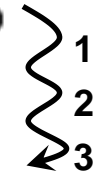
- opcode 指明指令是 beq 或 bne
- rs 和 rt 指明要比较的两个寄存器
- 立即数字段?
  - 立即数只有16 bits, PC (程序计数器) 有32-bit 的指向内存的指针; 立即数无法表示出完整的内存地址
- 通常使用的if-else, while, for等分支语句, 一般循环体都较小
- 函数调用与无条件跳转指令都会用到跳转指令(j and jal), 不是分支指令
- 结论:多数情况下, 分支语句转跳时, PC 的变化值都相差不大
- 在32-bit 指令格式中执行分支语句的解决方案: PC-相对寻址
  - 将16-bit立即数使用补码表示, 在需要分支的时候与PC相加
  - 可以分支到 $PC \pm 2^{15}$  字节的地方
  - 还能不能更好?
- 注意: 指令的起始地址都一定要是4的倍数(字对齐)
  - 和PC相加的立即数也应该是4的倍数! 其实可以分支到 $PC \pm 2^{15}$  指令字了(或者说  $PC \pm 2^{17}$  字节)
  - $PC = (PC + 4) + (immediate * 4)$



## 分支的例子

### ❖ MIPS 代码

```
Loop: beq      $9, $0, End
      add      $8, $8, $10
      addi     $9, $9, -1
      j        Loop
End:
```



**beq** 分支指令是I-Format格式的

opcode = 4

rs = 9 (第一操作数)

rt = 0 (第二操作数)

immediate = ???

立即数字段:

要和**PC**相加(或相减)的指令数, 是从分支语句的**下一条**指令算起的

在这条 **beq** 的分支中, **immediate = 3**

分支指令的十进制表示

4	9	0	3
---	---	---	---

分支指令的二进制表示

000100	01001	00000	000000000000000011
--------	-------	-------	--------------------

## J-Format 指令

- ❖ 在分支语句中, 假定不会分支到太远的地方, 所以可以指明PC的 **变化值**
- ❖ 对于一般的跳转指令 (j 和 jal), 是有可能跳到内存中 **任意** 一个地方的
  - 理想情况下, 可以直接给出一个 **32-bit** 的内存地址, 告诉要跳到哪里

### J-Format 指令

opcode (6)	target address (26)
------------	---------------------

- 保持 opcode 字段与 R-format 及 I-format 一样, 维护一致性原则
- 把其他所有字段都加到一起, 使能表示的地址尽量大: **26bits**
- 利用字对齐, 可以表示出 **32-bit** 地址的 **28 bits**
- 剩下的最高4位根据定义, 直接从PC取

$$\text{New PC} = \{ \text{PC}[31..28], \text{target address}, 00 \}$$

- 如果确实需要一个 **32-bit** 地址, 就把它放进寄存器, 使用 jr 指令

# 第五讲：指令系统与MIPS汇编

## 一. 指令格式

1. 指令系统概述
2. 指令格式
3. 寻址方式

## 二. MIPS指令系统

## 三. MIPS汇编语言

1. 概述
2. MIPS汇编指令和指令字
3. **MIPS汇编程序**

# 汇编语言语句

## ❖ MIPS汇编中的3类语句

1. 可执行指令
  - 为处理器生成在运行时执行的机器码，告诉处理器该做什么
2. 伪指令和宏
  - 由汇编程序翻译成真正的指令，简化编程人员的工作
3. 汇编伪指令
  - 当翻译代码时为汇编程序提供信息，用来定义段、分配内存变量等
  - 不可执行：汇编伪指令不是指令集的一部分

## ❖ 通常一个语句一行

## ❖ 汇编语言指令格式

[标签:]      操作符      [操作数]      [#注释]

### ➤ 标签: (可选)

- 标记内存地址，必须跟冒号；通常在数据和代码段出现

### ➤ 操作符

- 定义操作 (比如 `add`, `sub`, 等)

### ➤ 操作数

- 指明操作需要的数据；可以是寄存器，内存变量或常数
- 大多数指令有3个操作数

### ➤ #注释

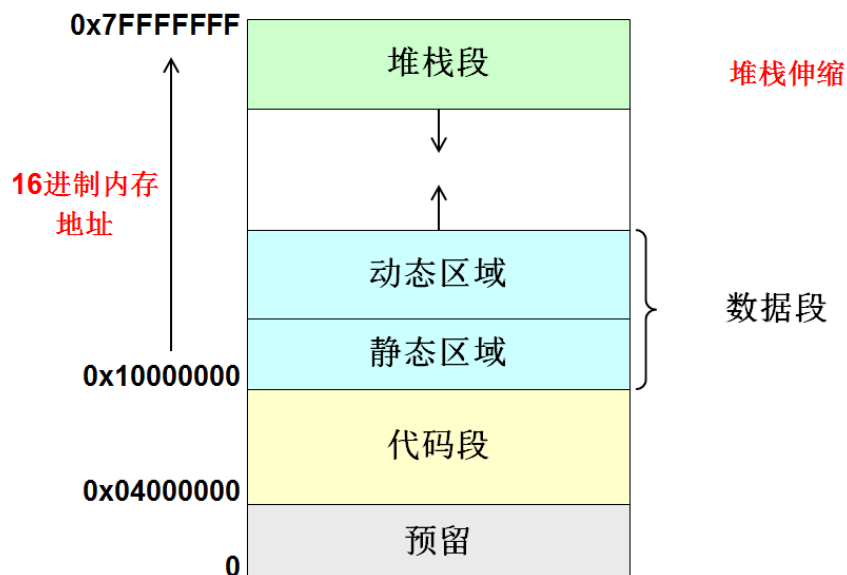
- 由 ‘#’ 开头在1行内结束；非常重要！

# 程序模板

## ❖ .DATA, .TEXT, 和 .GLOBL 伪指令

- **.DATA** 伪指令，定义程序的**数据段**，程序变量需要在该伪指令下定义
  - 汇编程序会分配和初始化变量的存储空间
- **.TEXT** 伪指令，定义程序的**代码段**
- **.GLOBL** 伪指令，声明一个符号为**全局的**，可被其它文件引用
  - 用该伪指令声明一个程序的 *main* 过程

```
# Title:                               Filename:
# Author:                             Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
. . .
##### Code segment #####
.text
.globl main
main:                                # main program entry
. . .
li $v0, 10                          # Exit program
syscall
```



# 数据定义

## ❖ 为变量的存储划分内存

- 可能会有选择的为数据分配名字（标签）

## ❖ 语法

[名字:] 伪指令 初始值 [, 初始值] ...

var1: .WORD 10

- 所有的初始值在内存中以二进制数据存储

## ❖ 数据伪指令

- .BYTE 伪指令，以8位字节存储数值表
- .HALF 伪指令，以16位（半字长）存储数值表
- .WORD 伪指令，以32位（一个字长）存储数值表
- .WORD w:n 伪指令，将32位数值 *w* 存入 *n* 个边界对齐的连续的字中
- .FLOAT 伪指令，以单精度浮点数存储数值表
- .DOUBLE 伪指令，以双精度浮点数存储数值表

## ❖ 字符串伪指令

- .ASCII 伪指令，为一个ASCII字符串分配字节序列
- .ASCIIZ 伪指令，与.ASCII 伪指令类似，但字符串以NULL结尾
- .SPACE n 伪指令，为数据段中 *n* 个未初始化的字节分配空间
- 字符串中的特殊字符（按照 C 语言的约定），“新行: \n ， Tab:\t， 引用: \"

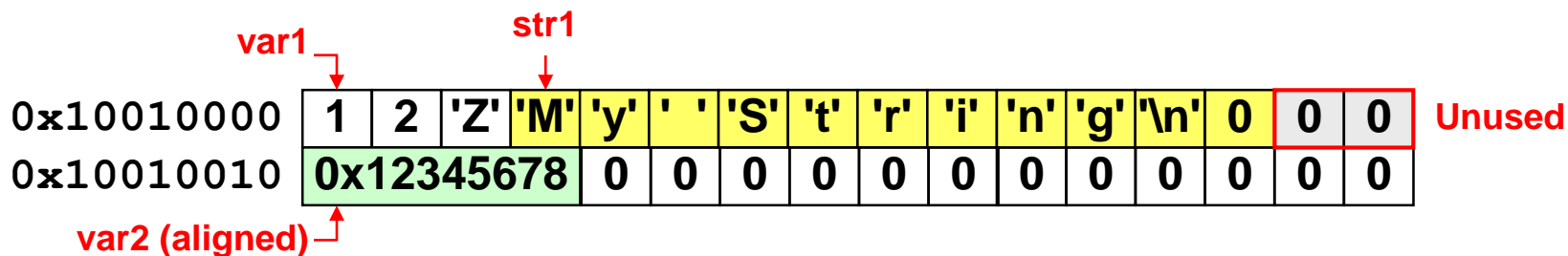
# 数据定义的例子

```
.DATA
var1:  .BYTE    1, 2, 'Z'
str1:  .ASCIIZ  "My String\n"
var2:  .WORD    0x12345678
```

如果初始值超过了值域上界，  
汇编程序会报错

- ❖ 汇编程序为标签（变量）构建符号表
  - 为数据段的每一个标签计算地址

标签	地址
var1	0x10010000
str1	0x10010003
var2	0x10010010



# 内存对齐和字节序

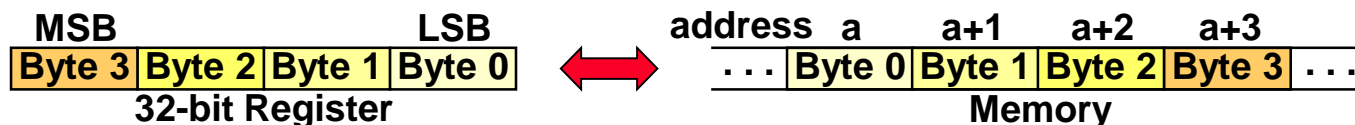
## ❖ 对齐: 地址是空间大小的整数倍

- 字的地址是**4**的整数倍
  - 地址的**2**位最低有效位必须是 **00**
- 半字的地址是 **2** 的整数倍

## ❖ **.ALIGN n** 伪指令, 对下一个定义的数据做 $2^n$ 字节对齐

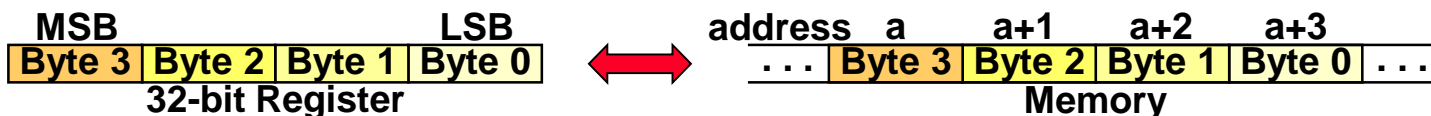
## ❖ 字节序和端

- 处理器对一个字内的字节排序有两种方法
- **小端字节排序**
  - 内存地址 = **最低有效字节**的地址, 例子: Intel IA-32, Alpha



## ➤ **大端字节排序**

- 内存地址 = **最高有效字节**的地址, 例子: SPARC, PA-RISC



- **MIPS** 可以操作以上两种字节序



# 系统调用

- ❖ 程序通过系统调用实现输入/输出
- ❖ MIPS 提供一条特殊的 **syscall** 指令，从操作系统获取服务
  - 使用 **syscall** 系统服务
    - 从 **\$v0** 寄存器中读取服务数
    - 从 **\$a0, \$a** 等寄存器中读取参数值（如果有）
    - 发送 **syscall** 指令
    - 从结果寄存器中取回返回值（如果有）

Service	\$v0	Arguments / Result
Print Integer	1	\$a0 = integer value to print
Print Float	2	\$f12 = float value to print
Print Double	3	\$f12 = double value to print
Print String	4	\$a0 = address of null-terminated string
Read Integer	5	\$v0 = integer read
Read Float	6	\$f0 = float read
Read Double	7	\$f0 = double read
Read String	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	\$a0 = character to print
Read Char	12	\$a0 = character read

# 过程

## ❖ swap 过程 (C程序)

➤ 翻译成 MIPS 汇编语言

## ❖ 调用swap过程: swap(a,10)

➤ 将数组a的地址和10作为参数传递

➤ 调用swap过程, 保存返回地址  
**\$31 = \$ra**

➤ 执行swap过程

➤ 返回对返回地址的控制

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

参数:

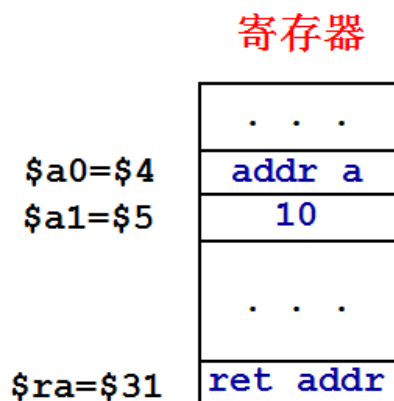
\$a0 = v[] 的地址

\$a1 = k,

返回地址在 \$ra 中

swap:

```
sll $t0,$a1,2    # $t0=k*4
add $t0,$t0,$a0  # $t0=v+k*4
lw  $t1,0($t0)   # $t1=v[k]
lw  $t2,4($t0)   # $t2=v[k+1]
sw  $t2,0($t0)   # v[k]=$t2
sw  $t1,4($t0)   # v[k+1]=$t1
jr  $ra          # return
```



调用

```
la    $a0, a
li    $a1, 10
jal   swap
# return here
...
```

swap:

```
sll $t0,$a1,2
add $t0,$t0,$a0
lw  $t1,0($t0)
lw  $t2,4($t0)
sw  $t2,0($t0)
sw  $t1,4($t0)
jr  $ra
```

# 过程的指令

## ❖ JAL (Jump-and-Link) : 调用指令

- 寄存器  $\$ra = \$31$  被 JAL 用来保存返回地址 ( $\$ra = PC+4$ )
- 通过伪直接寻址转跳

## ❖ JR (Jump Register) : 返回指令

- 跳转到在寄存器Rs ( $PC = Rs$ )中存储的地址所在指令

## ❖ JALR (Jump-and-Link Register)

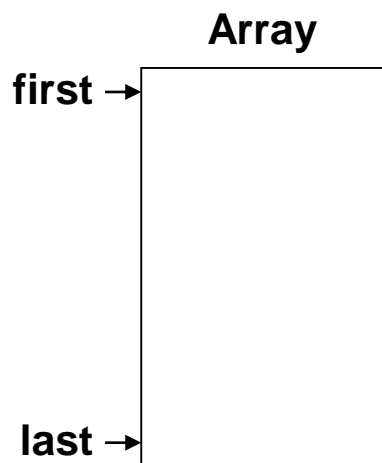
- 在 $Rd = PC+4$ 中存储返回地址,
- 跳转到在寄存器Rs ( $PC = Rs$ )中存储的地址所在过程
- 用于调用方法(地址仅在运行时可知)

Instruction		Meaning	Format					
jal	label	$\$31=PC+4, \text{ jump}$	$op^6 = 3$	$imm^{26}$				
jr	Rs	$PC = Rs$	$op^6 = 0$	$rs^5$	0	0	0	8
jalr	Rd, Rs	$Rd=PC+4, PC=Rs$	$op^6 = 0$	$rs^5$	0	$rd^5$	0	9

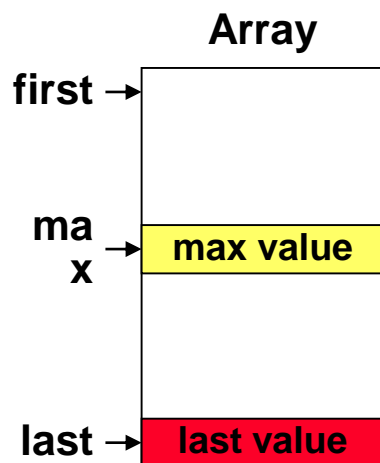
# 参数传递

- ❖ 汇编语言中的参数传递比高级语言中复杂
  - 将所有需要的参数放置在一个可访问的存储区域
  - 然后调用过程
- ❖ 会用到两种类型的存储区域
  - 寄存器: 使用通用寄存器 (**寄存器方法**); 内存: 使用栈 (**栈方法**)
- ❖ 参数传递的两种常用机制
  - 值传递: 传递参数**值**; 引用传递: 传递参数的**地址**
  - 按照约定, 参数传递通过寄存器实现
    - $\$a0 = \$4 \dots \$a3 = \$7$  用来做**参数传递**
    - $\$v0 = \$2 \dots \$v1 = \$3$  用来表示**结果数据**
  - 其它的参数/结果可以放在栈中
- ❖ 运行时栈用于
  - 不适合使用寄存器时用来存储变量/数据结构
  - 过程调用中**保存和恢复寄存器**
  - 实现递归
- ❖ 运行时栈通过软件规范实现
  - **栈指针**  $\$sp = \$29$  (指向栈顶); **帧指针**  $\$fp = \$30$  (指向过程帧)

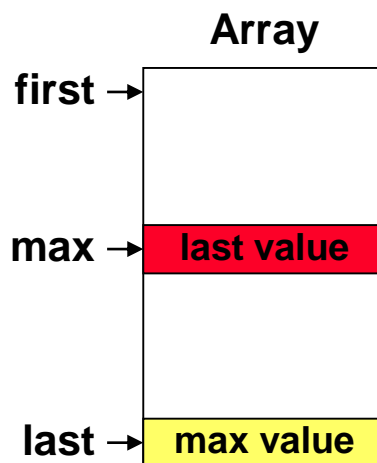
# [例] 选择排序



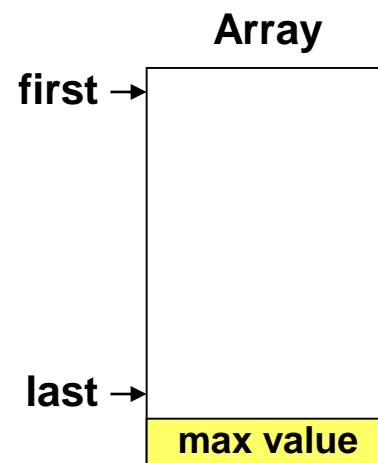
Unsorted



Locate  
Max

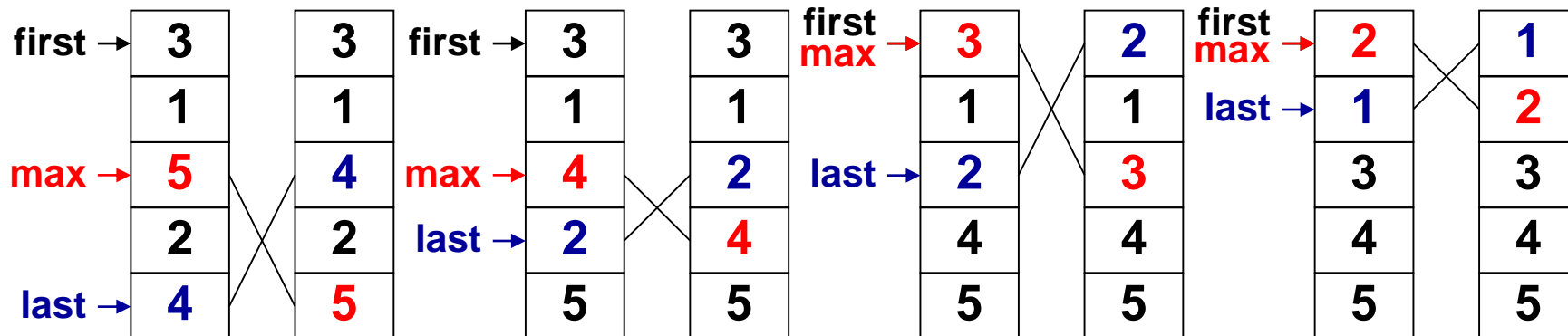


Swap Max  
with Last



Decrement  
Last

❖ 例子



# 选择排序过程

```
# Objective: Sort array using selection sort algorithm
#     Input: $a0 = pointer to first, $a1 = pointer to last
#     Output: array is sorted in place
#####
sort: addiu $sp, $sp, -4      # allocate one word on stack
      sw     $ra, 0($sp)     # save return address on stack
top:  jal    max             # call max procedure
      lw     $t0, 0($a1)     # $t0 = last value
      sw     $t0, 0($v0)     # swap last and max values
      sw     $v1, 0($a1)
      addiu  $a1, $a1, -4    # decrement pointer to last
      bne    $a0, $a1, top   # more elements to sort
      lw     $ra, 0($sp)     # pop return address
      addiu  $sp, $sp, 4
      jr     $ra            # return to caller
```

# 找最大值过程

```
# Objective: Find the address and value of maximum element
#   Input: $a0 = pointer to first, $a1 = pointer to last
#   Output: $v0 = pointer to max, $v1 = value of max
#####
max:  move    $v0, $a0          # max pointer = first pointer
      lw     $v1, 0($v0)       # $v1 = first value
      beq    $a0, $a1, ret     # if (first == last) return
      move   $t0, $a0         # $t0 = array pointer
loop: addi   $t0, $t0, 4       # point to next array element
      lw     $t1, 0($t0)       # $t1 = value of A[i]
      ble    $t1, $v1, skip    # if (A[i] <= max) then skip
      move   $v0, $t0         # found new maximum
      move   $v1, $t1
skip: bne    $t0, $a1, loop    # loop back if more elements
ret:  jr     $ra
```

## 递归过程示例

```
int fact(int n) { if (n<2) return 1; else return (n*fact(n-1)); }
```

```
fact:  slti      $t0,$a0,2      # (n<2)?
        beq      $t0,$0,else    # if false branch to else
        li       $v0,1         # $v0 = 1
        jr       $ra           # return to caller
else:   addiu     $sp,$sp,-8     # allocate 2 words on stack
        sw       $a0,4($sp)    # save argument n
        sw       $ra,0($sp)    # save return address
        addiu     $a0,$a0,-1    # argument = n-1
        jal      fact          # call fact(n-1)
        lw       $a0,4($sp)    # restore argument
        lw       $ra,0($sp)    # restore return address
        mul      $v0,$a0,$v0    # $v0 = n*fact(n-1)
        addi     $sp,$sp,8     # free stack frame
        jr       $ra           # return to caller
```