

# Computer Organization and Design

---

中山 大 学  
计算机学院

郭雪梅

Email: [guoxuem@mail.sysu.edu.cn](mailto:guoxuem@mail.sysu.edu.cn)

Computer Architecture =  
Instruction Set Architecture  
+ Machine Organization

Read: Chapter 2.1-2.3, 2.5-2.7



# Computer Organization and Design

---

中山 大 学  
计算机学院

郭雪梅

Email: guoxuem@mail.sysu.edu.cn

Computer Architecture =  
Instruction Set Architecture  
+ Machine Organization

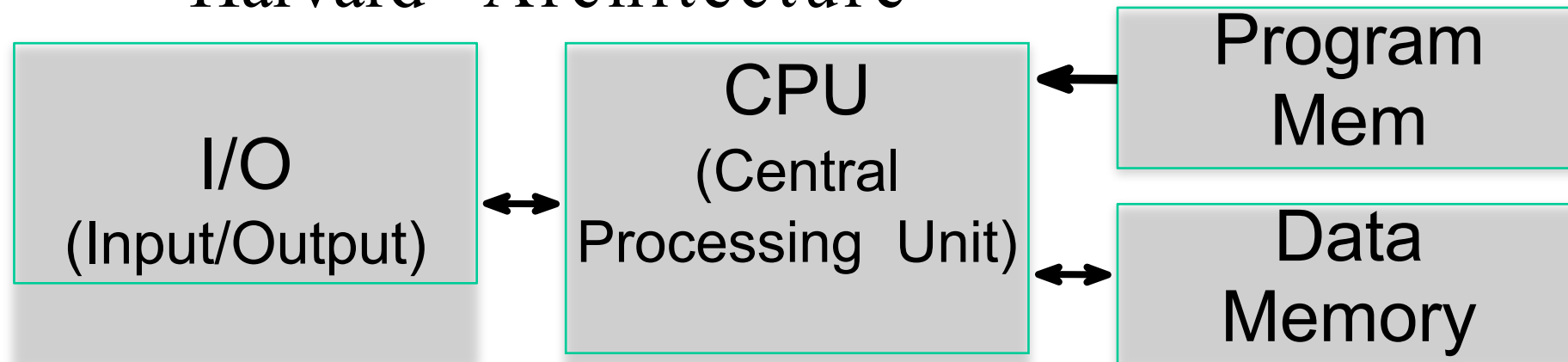
Read: Chapter 2.1-2.3, 2.5-2.7



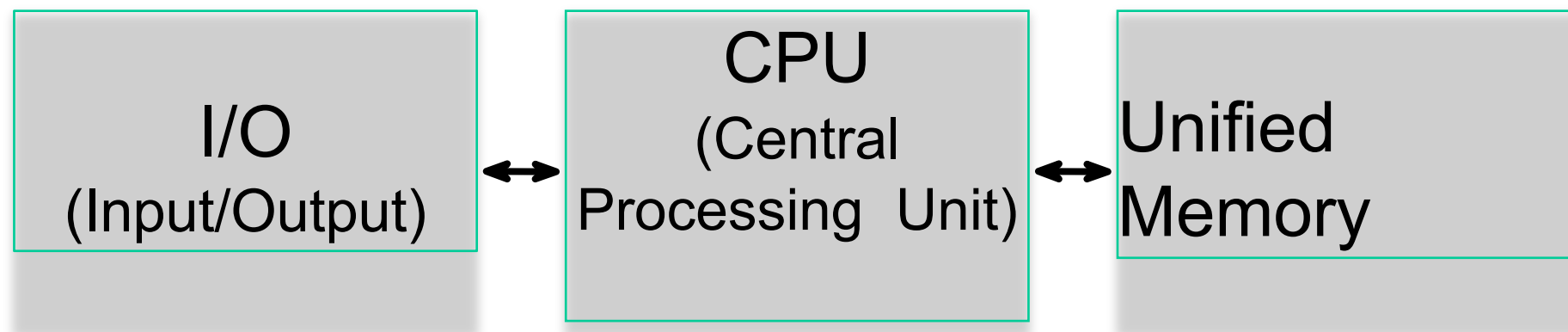
# A Bit of History

“数据”和“指令”是否混合使用的争论导致两种常见的计算机体系结构

## “Harvard” Architecture



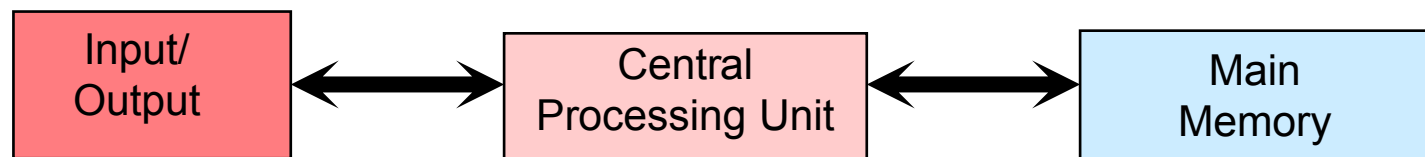
## “Von Neumann” Architecture



# 通用计算机

## \* 冯诺依曼体系结构

- 已经探索了许多通用计算机的架构模型
- 今天的大多数计算机都基于约翰·冯·诺依曼在 1940 年代后期提出的模型
- 它的主要组成部分是：



中央处理单元 (CPU)：取指、解释和执行一组指定的操作，称为指令。

内存：存储 $N$ 个字，每个 $W$ 位，其中 $W$ 是一个固定的架构参数， $N$ 可以扩展以满足需要。

I/O：与外界通信的设备。

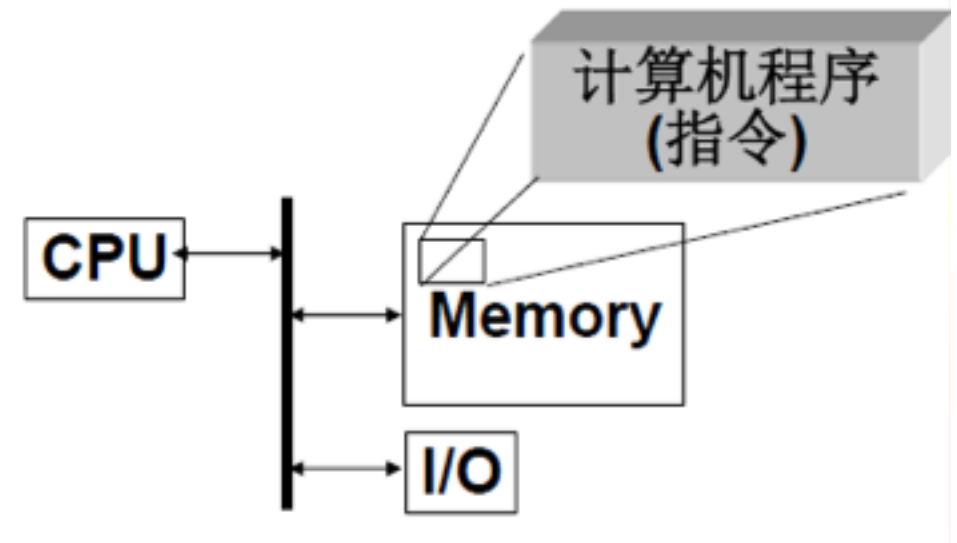
# 冯诺依曼体系结构

- 也称为存储程序计算机（指令存于内存中），有两个关键属性：
- 存储程序
  - 指令在内存中以线性阵列存储
  - 指令和数据统一编址
- 指令顺序执行
  - 一次执行一条指令 (取指, 执行, 完成)
  - 程序计数器(指令指针) 指向当前指令.
  - 程序计数器顺序递增，除非遇到转移指令

# 指令系统体系结构ISA

## Princeton (Von Neumann) 系统结构

- 数据和指令存放在统一存储器中  
(“存储程序计算机”)  
(“stored program computer”)
- 程序当作数据
- 存储系统的利用 (Storage utilization)
- 单一的存储器接口



## Harvard 系统结构

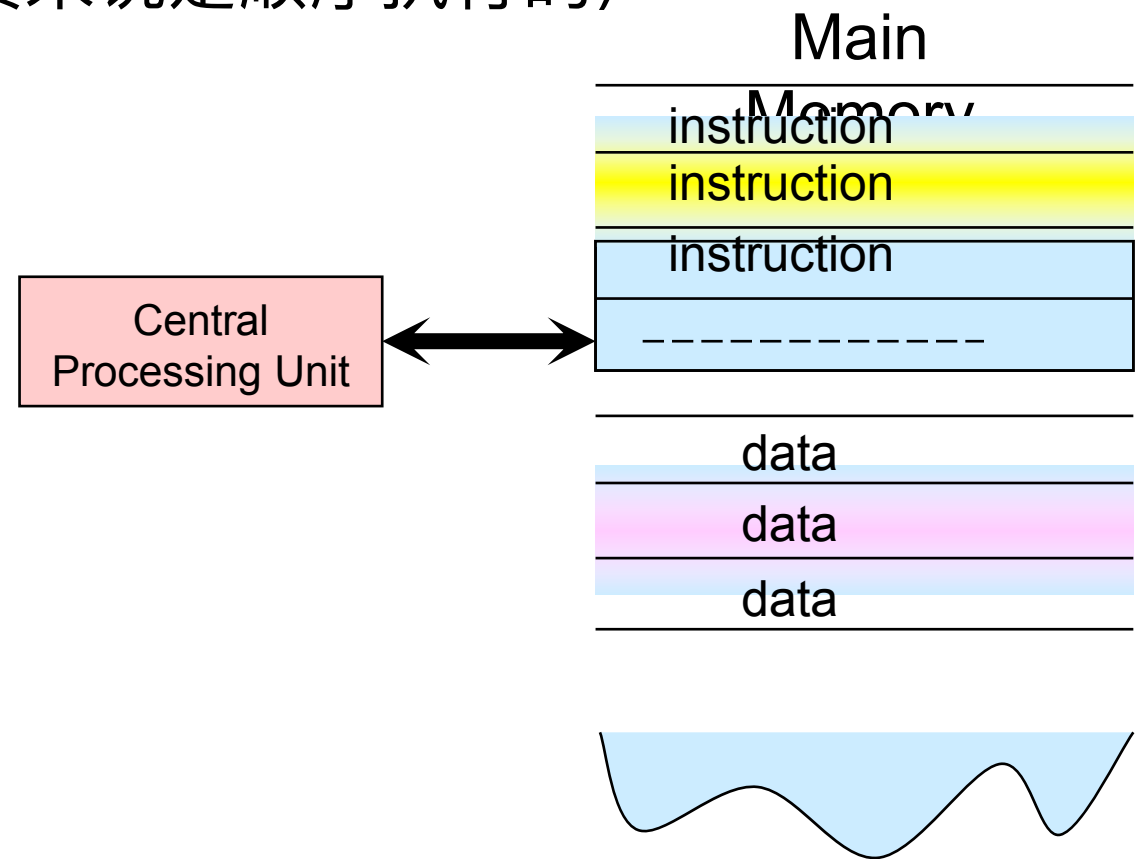
- 数据 & 指令  
存放在不同的存储器中
- 在某些高性能实现中  
具有优势

# 存储程序计算机

- 冯诺依曼架构模型:

- 指令和数据存于统一的存储器中(“main memory”)
- 所有指令顺序执行 (或至少对程序员来说是顺序执行的)

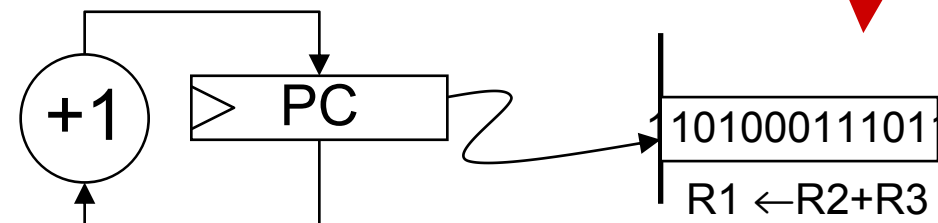
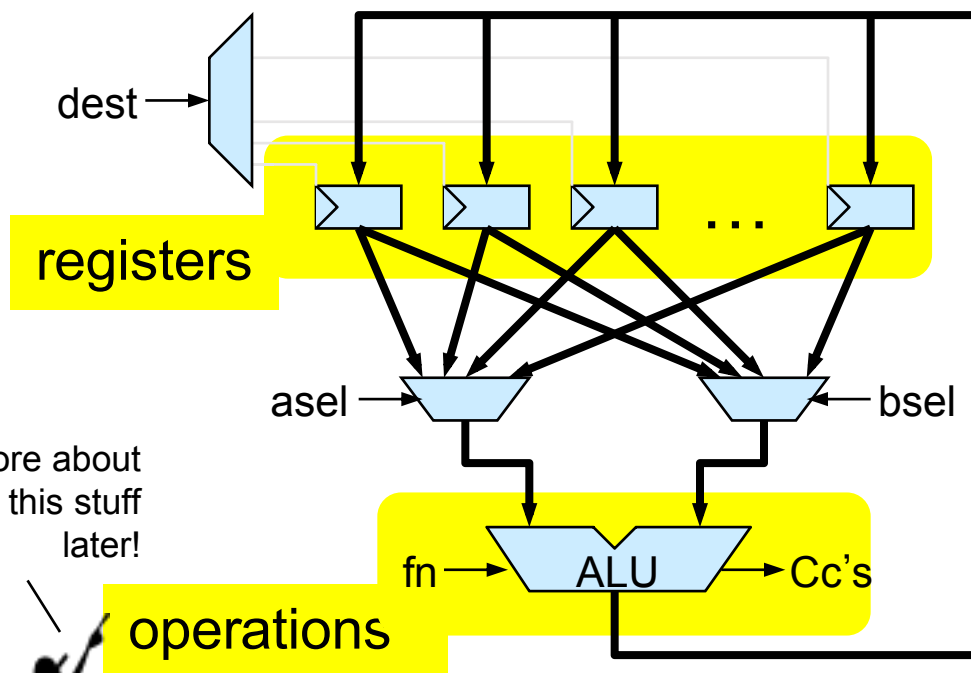
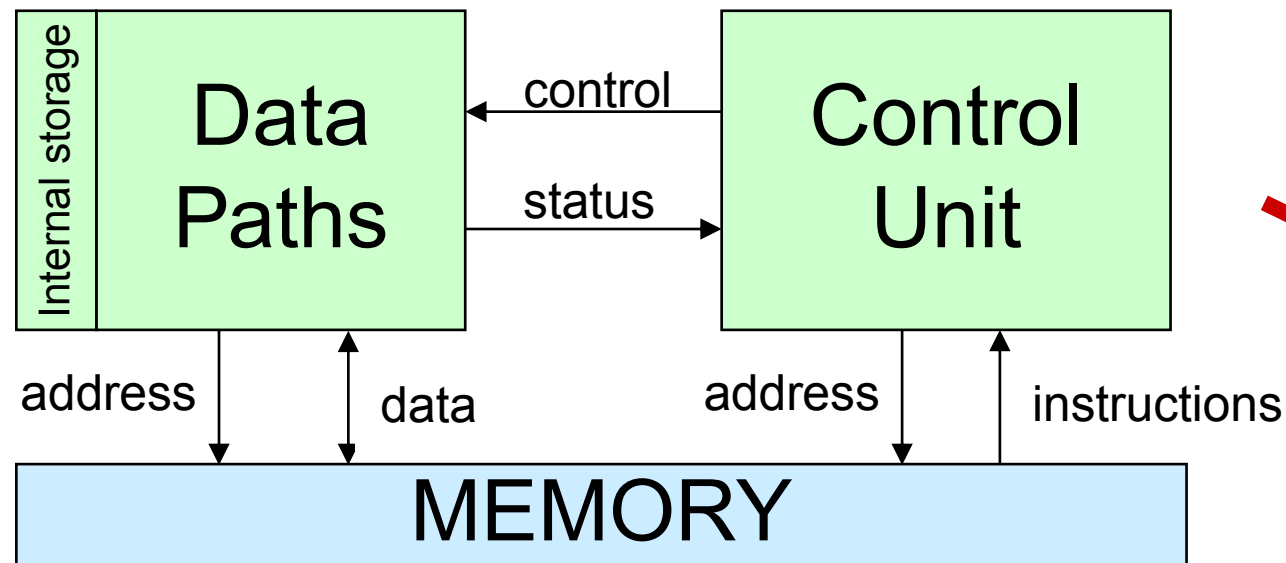
Key idea: 内存不仅保存有数据, 还有组成程序的指令.



- \* CPU 从内存取指并执行 ...

- CPU 是 H/W 的接口或说明器
- 对于这个说明器程序是简单的数据
- 主存: 单一可扩展的资源池
  - 对数据和程序大小有约束

# 冯诺依曼计算机剖析



- 指令是二进制编码
- 程序计数器或 PC: 下一条指令执行地址
- 翻译指令为控制信号, 实现数据通路的逻辑



# 指令原理

程序员和硬件之间的协议

- ▶ 定义系统的可见状态, 定义状态如何响应指令而更改

对于程序员来说: ISA是程序如何执行的模型

对于硬件设计者来说: ISA是正确执行程序的方式

- ▶ 有了稳定的ISA, 软件不在乎硬件在引擎盖下是什么样子

- ▶ 硬件实现可能会发生巨大变化

- ▶ 只要硬件实现相同的ISA, 所有先前的软件都仍然能够运行

- ▶ 示例: x86 ISA已经跨越了许多芯片; 指令已添加了很多, 但用于先前芯片的SW仍能运行

ISA规范: 指令集的二进制编码。

# Architecture vs.Implementation

**Architecture (体系结构)**:定义计算机系统对程序和数据集的响应

- ▶ 对于程序员来讲是计算机系统的可见元素

**实现Implementation (microarchitecture)**:定义计算机怎样做

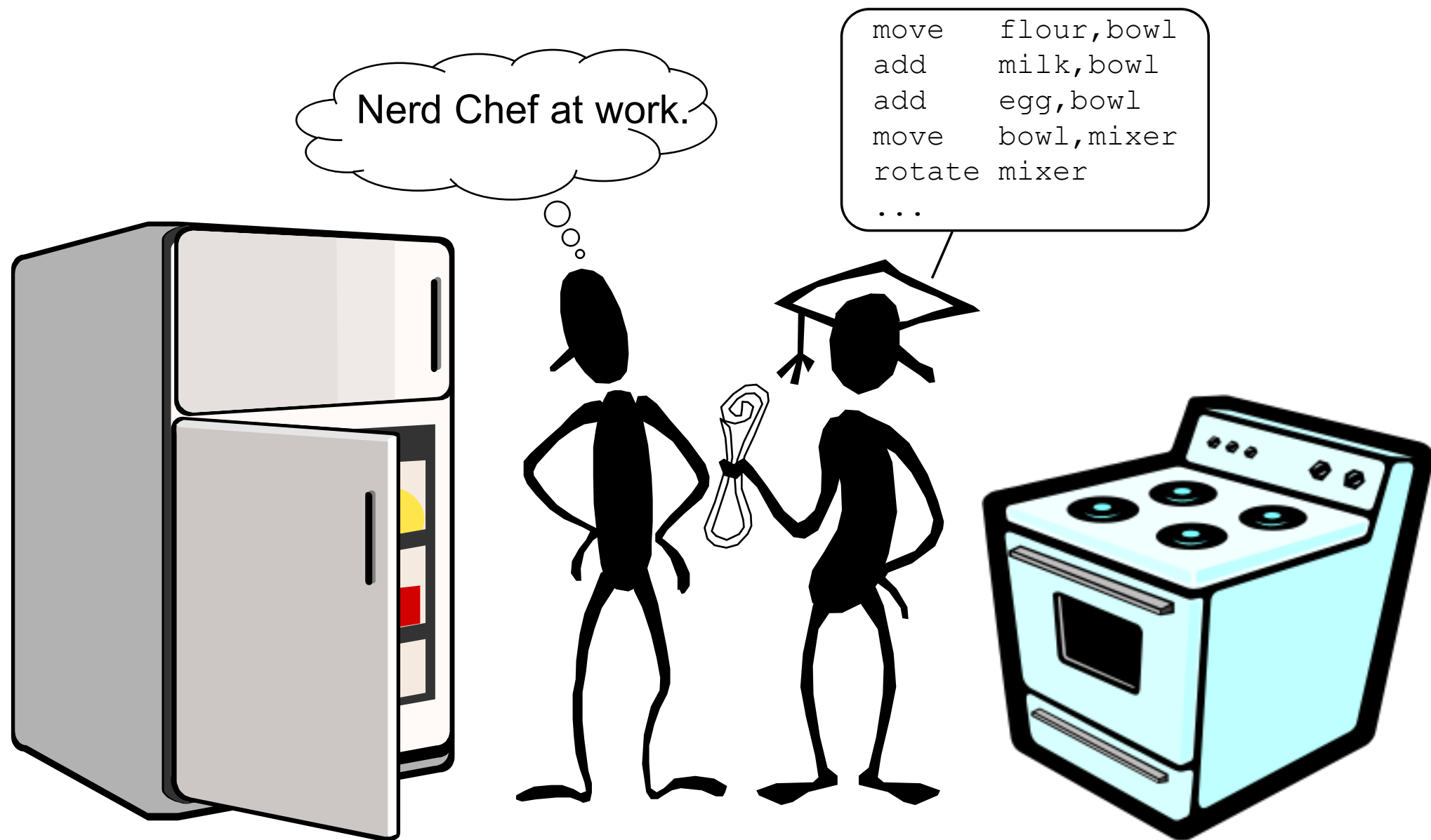
- ▶ 要完成的步骤顺序
- ▶ 操作。执行每个操作的时间。
- ▶ 隐藏的“记账”功能。

*如果体系结构 (Architecture)发生变化, 某些程序可能不能再运行或返回相同的答案。*

*如果实现 (Implementation ) 发生变化, 一些程序可能运行得更快/更慢/更好, 但答案不会改变。*

# Concocting an Instruction Set

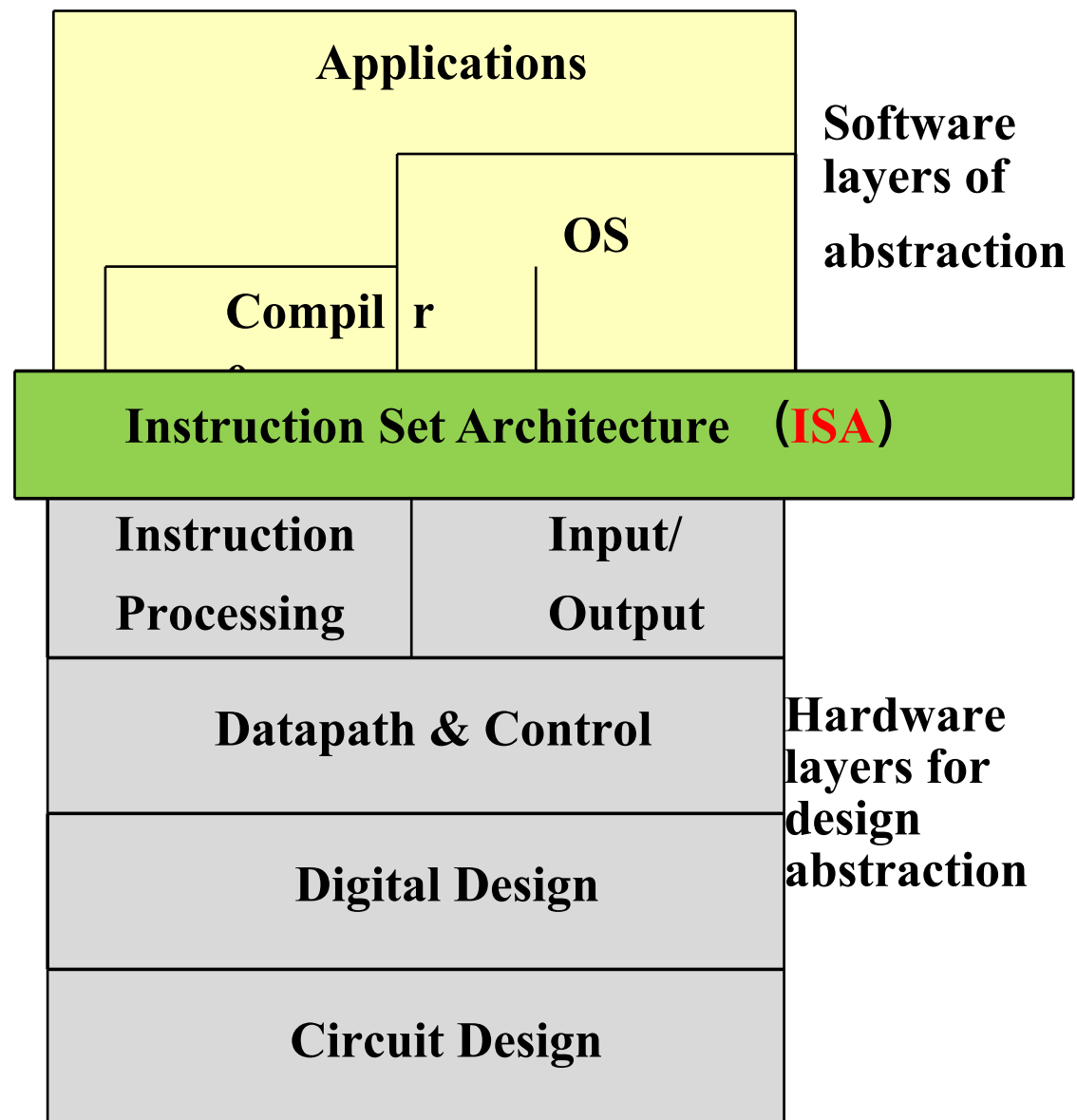
## 编制指令集



Read: Chapter 2.1-2.3, 2.5-2.7

# 指令系统概述

- ❖ **指令(Instruction):** 计算机硬件可以执行的表示一种基本操作的二进制代码
- ❖ **指令集(Instruction Set):** 计算机中所有指令的集合
- ❖ **指令集系统结构(Instruction Set Architecture, ISA):**  
计算机硬件与底层软件之间的界面，是机器语言程序员所需了解的计算机的属性，如：指令集、数据类型及表示、寄存器组织、存储器的组织和寻址方式、I/O结构、中断机制、保护机制等



# Instruction Set Architecture (ISA)

- 指令编码引发了有趣的选择...
  - 权衡取舍:性能、紧凑性、可编程性
  - 一致性. 不同的指令应该
    - 指令大小一样吗?
    - 花同等时间执行吗?
      - 趋势: 一致性. 提供简单、速度、流水线.
  - 复杂度. 不同种类的指令有多少? 什么级别的操作?
    - 对特定软件操作的支持级别: 数组索引、过程调用、“多项式评估”等
    - “精简指令集计算机”(RISC) 理念: 指令简单, 速度优化
    - 工程与艺术的结合...
- 试验 (通过模拟) 是我们做出选择的最佳方法!

Our representative example: the **MIPS** architecture!

# Assembly Language Instructions

## 汇编语言表示的指令

### 机器语言

- ISA 指令集体系结构要使得硬件构建、编译容易实现，同时最大化性能，最小化造价
- 存储程序概念
  - 指令存在内存中
- 我们的目标: MIPS ISA
  - 类似于自 1980 年代以来开发的其他 ISA
  - 使用于Broadcom, Cisco, NEC, Nintendo, Sony, ...

**设计目标：最大化性能、最小化成本、缩短设计时间（上市时间）、最小化内存空间（嵌入式系统）、最小化功耗（移动系统）**

# 指令系统概述 —— 基本问题

## ❖ 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

# 指令系统概述 ---- 指令的要素

## ❖ 机器指令的要素

### ➤ 操作码(Operation Code):

指明进行的何种操作 (如 ADD, MOV, I/O)

### ➤ 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, 一般指令是按顺序依次执行的, 所以绝大多数指令中并不显式的指明下一条指令的地址, 也就是说, 指令格式中并不包含这部分信息。只有少数指令需要显式指明下一条指令的地址。



# 指令系统概述 —— 指令类型

- ❖ **数据传送指令：**寄存器与存储器之间，寄存器之间传递数据
  - 取数、存数、传送、交换、设置/清除指令（Mov, Store, Load, Set等）
  - 串操作指令（MOVSB, MOVSW）
  - I/O指令：IN, OUT
  - 堆栈指令：PUSH, POP
- ❖ **算术/逻辑运算指令**
  - 算术运算指令：定点数、浮点数、十进制数的加减乘除运算
  - 逻辑运算指令：与/或/非/异或等逻辑运算（And, Or, Not, Xor等）
  - 移位指令：算术移位，逻辑移位，循环移位
  - 向量运算指令：对整个向量或矩阵求和、求积等运算
- ❖ **程序控制类指令**
  - 转移指令：无条件转移指令，有条件转移指令
  - 循环控制指令（LOOP）
  - 子程序调用与返回指令（CALL, RET）
  - 程序中断指令及返回（INT, IRET）
- ❖ **其它指令：**多用户多任务系统中的特权指令；复位/暂停/空操作等指令等等

# 指令系统概述——指令类型

## ■ 80X86使用最多的10条指令

◦ Rank instruction Integer Average Percent total executed

1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

◦ Simple instructions dominate instruction frequency

# 指令系统概述——操作数

## ❖ 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

# 指令系统概述 ——ISA种类

## 指令系统体系结构（ISA）的基本种类

### ❖ ISA分类可考虑的主要因素：

- 操作数的存储位置和方法
- 显式表示的操作数个数
- 操作数的类型和大小
- 操作数的寻址方式
- 指令集所提供的操作类型

### ❖ 根据操作数的存储位置和方法，ISA可分为：

- 堆栈类（Stack）
- 累加器类（Accumulator）
- 通用寄存器类（General Purpose Register）

# 指令系统概述 ——ISA种类

## ❖ 指令系统体系结构 (ISA) 的基本种类

### ➤ 堆栈类 (Stack)

0 Address:  $\text{AddTos (Top Of Stack)} \leftarrow \text{Tos} + \text{Next}$

### ➤ 累加器类 (Accumulator, 1 Register )

1 Address :      Add A               $\text{Acc} \leftarrow \text{Acc} + \text{Mem}[A]$

1+x Address:   Addx A             $\text{Acc} \leftarrow \text{Acc} + \text{Mem}[A+x]$

### ➤ 通用寄存器类 (General Purpose Register)

- (装入Load - 存储Store式)

#### Register-Register式

3 Address :   Add Ra Rb Rc

#### 装入Load - 存储Store式

Load Ra Rb   Store Ra Rb

#### •Register-Memory式

2 Address:   Add Ra B

#### •Memory-Memory式

3 Address:   Add A B C

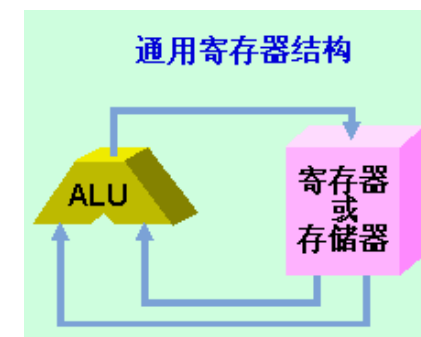
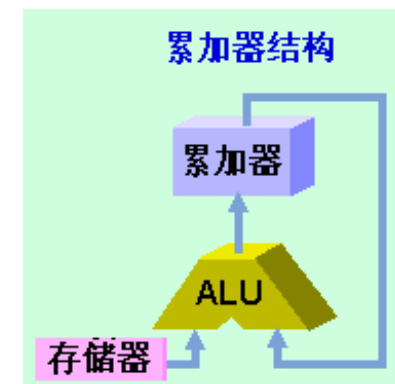
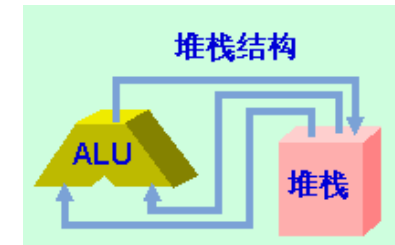
$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$

$\text{Ra} \leftarrow \text{Mem}[\text{Rb}]$

$\text{Mem}[\text{Rb}] \leftarrow \text{Ra}$

$\text{Ra} \leftarrow \text{Ra} + \text{Mem}(\text{B})$

$\text{Mem}(\text{A}) \leftarrow \text{Mem}(\text{B}) + \text{Mem}(\text{C})$



# CISC与RISC

## ❖ 指令系统优化设计的两种相反的方向

➤ 增强指令功能：CISC (Complex Instruction Set Computer)，即复杂指令系统计算机

- 思路：把一些原来由软件实现的、常用的功能改用硬件指令来实现
- 特点：格式复杂，寻址方式复杂，指令种类多
- 实例：X86指令系统

➤ 简化指令功能：RISC (Reduced Instruction Set Computer)，即精简指令系统计算机

- 思路：只将使用频度高且简单的功能用硬件指令来实现，不常使用的或复杂的功能由软件实现
- 特点：格式简单，指令长度和操作码长度固定；简单寻址方式，大部分指令使用寄存器直接寻址
- 实例：MIPS 指令系统

# CISC与RISC

## ❖ CISC的背景

- ➤ 计算机硬件成本不断下降，软件开发成本不断提高
  - 在指令系统中增加更多的、更复杂的指令，以提高操作系统的效率；
  - 并尽量缩短指令系统与高级语言的语义差别，以便于高级语言的编译。
- ➤ 程序的兼容性
  - 为保持程序的兼容性，同一系列计算机的新机器和高档机的指令系统只能扩充而不能缩减。

# CISC与RISC

## ❖ CISC指令系统的特点

- 指令系统复杂庞大（一般数百条指令）；
- 寻址方式多，指令格式多，指令字长不固定；
- 可访存指令不受限制；
- 各种指令使用频率相差很大；
- 各种指令执行时间相差也很大；
- 大多数采用微程序控制器。



# CISC与RISC

## ❖ RISC的背景

### ➤ 80-20规律

- 典型程序中80%的语句仅仅使用处理机中20%的指令，且这些指令都属于简单指令，如：取数、加、转移等。
- 付出巨大代价添加的复杂指令仅有20%的使用概率

### ➤ VLSI时代

- VLSI，即超大规模集成电路 (Very Large Scale Integrated circuits)；
- 复杂的指令系统需要复杂的控制器，占用较多的芯片面积，它的设计、验证、实现都变得更加困难。

# CISC与RISC

## ❖ RISC技术

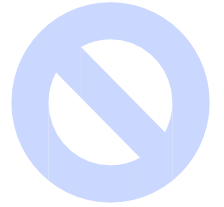
- 把使用频率为80%的、在指令系统中仅占20%的简单指令保留下来，消除剩下80%的复杂指令，复杂功能用子程序实现
- 不用微程序控制，采用简单的硬连线控制，控制器极大简化，加上优化编译配合硬件的改进，使系统的速度大大提高
- 短周期时间、单周期执行指令（指令执行在一个机器周期内完成）
- Load（取）/Store（存）结构，取数（存储器→寄存器）、存数（寄存器→存储器）
- 大寄存器堆，寄存器数量较多
- 哈佛（Harvard）总线结构，指令Cache、数据Cache，双总线动态访问机构
- 高效的流水线结构、延迟转移、重叠寄存器窗口技术等

# CISC与RISC

## ❖ RISC的指令系统的特点

- 处理器通用寄存器数量较多；
- 由使用频率较高的简单指令构成；
- 简单固定格式的指令系统；
- 指令格式种类少，寻址方式种类少；
- 访问内存仅限Load/Store指令，其他操作针对寄存器；
- 指令采用流水技术。

# CISC与RISC



## ❖ RISC与CISC性能对比

- CPI:Cycles per Instruction, 执行一条指令的平均周期数
- RISC比CISC机器的CPI 要小
- CISC一般用微码技术, 一条指令往往要用好几个周期才能实现, 复杂指令所需的周期数则更多, CISC机器CPI 一般为4-6
- RISC一般指令一个周期内完成, 即CPI=1, 但LOAD、STORE等指令要略长些, 因此, RISC计算机的CPI约大于1

## ❖ RISC与CISC技术的融合

- 随着芯片集成度和硬件速度的增大, RISC系统也越来越复杂
- CISC也吸收了很多RISC的设计思想
  - 如: Intel 80486比80286更加注重常用指令的执行效率, 减少常用指令执行所需的周期数。

# MIPS Architecture

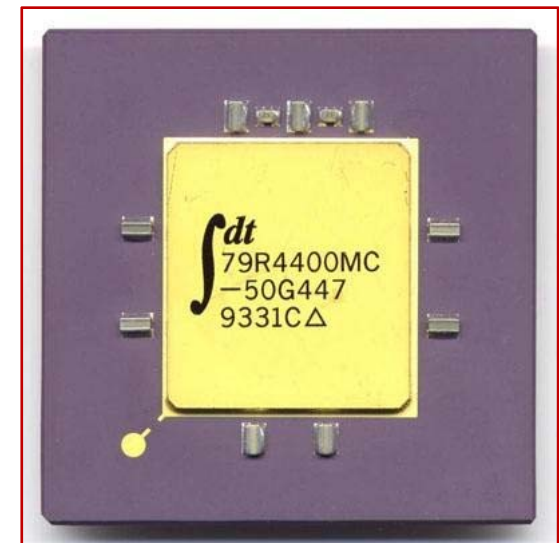
- MIPS: 建立了第一代商用RISC架构之一的半导体公司 (1984-2013, 被Imagination Technologies收购)
- 为什么选择 MIPS 而不是Intel x86 (或 ARM)?
  - MIPS简单、优雅； 避免陷入繁杂的细节
  - MIPS（曾经）广泛用于嵌入式应用系统，例如消费电子产品和网络路由器；x86 很少用于嵌入式，而嵌入式计算机比 PC 多得多。

# 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

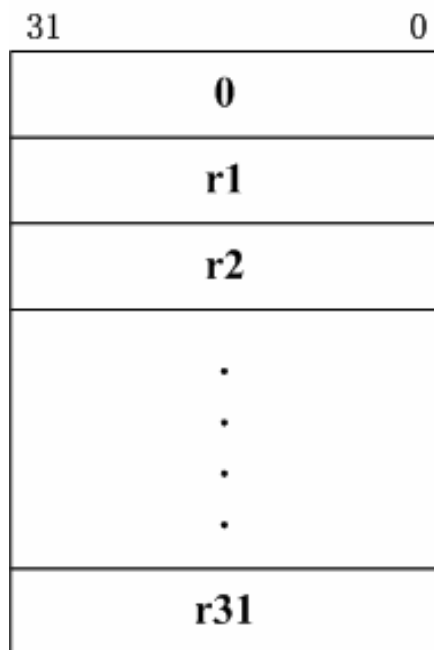


# MIPS 指令系统

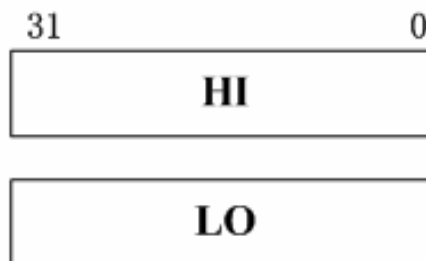
## ❖ MIPS R2000/R3000 寄存器结构

- 32位虚拟地址空间
- 32×32 bit GPRs, \$0~\$31
- 32×32 bit FPRs, \$f0~\$f31
- HI, LO, PC

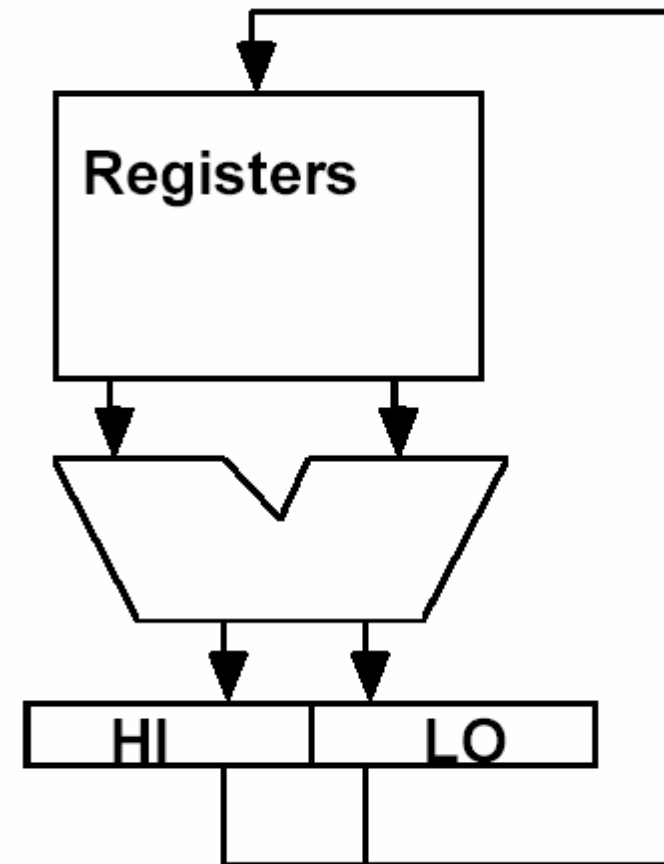
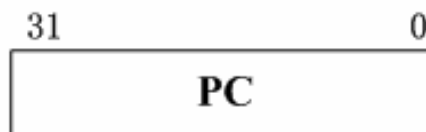
通用寄存器



乘/除寄存器



程序计数器



# MIPS 指令系统

## ❖ MIPS寄存器使用的约定

汇编变量：寄存器

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
...		(callee can clobber)
15	t7	
16	s0	callee saves
...		(caller can clobber)
23	s7	
24	t8	temporary (caller saves)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

汇编操作数是称为寄存器 (registers) 的对象，由于寄存器直接在硬件中，它们非常快



# Names of MIPS Registers

- MIPS 有32个，位宽也是32的寄存器，编号 0 到 31
- 每个寄存器都可以通过编号或名称来引用
- 用编号引用：
  - \$0, \$1, \$2, ... \$30, \$31
- 名称：
  - \$16 - \$23 → \$s0 - \$s7 (像C一样，可以存变量)
  - \$8 - \$15 → \$t0 - \$t7 (一般用来存临时变量)
- 通常，使用名称使代码更具可读性

# C, Java Variables vs. Registers

- 在 C (和大部分高级语言中):

- 需事先声明变量类型

- Example:

```
int fahr, celsius;  
char a, b, c, d, e;
```

- 每个变量只能表示它声明的类型的值（例如，不能混合和匹配 int 和 char 变量）

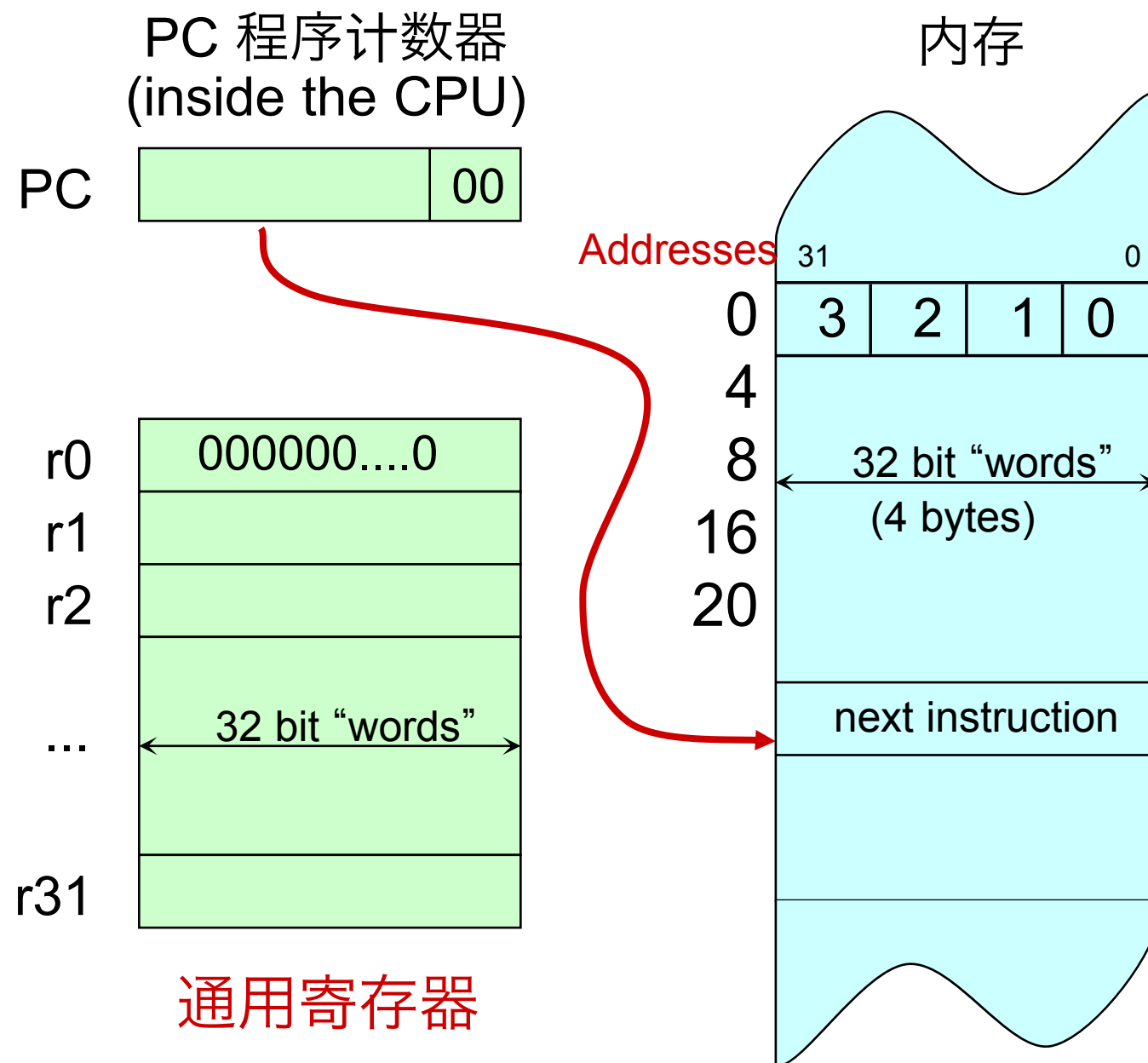
- 在汇编语言中:

- 寄存器没有类型;

- 操作 (Operation) 决定寄存器中的变量类型。

# MIPS Programming Model

具有代表性的简单 RISC 机器



我们使用MIPS子集, MIPS-32 核心指令集

Fetch/Execute loop:

- fetch  $\text{Mem}[\text{PC}]$
- $\text{PC} = \text{PC} + 4^\dagger$
- execute fetched instruction (may change PC!)
- repeat!

$^\dagger$  MIPS 使用字节内存地址. 指令是 32位的, 占4个字节 (一个字) 地址, 指令之间差4个字节地址。

# MIPS Register File 寄存器堆

- 运算指令的操作数必须使用寄存器，即数据通路中的寄存器堆中的寄存器

- 32个位宽32的寄存器

- 两个读端口
- 一个写端口

## □ 寄存器

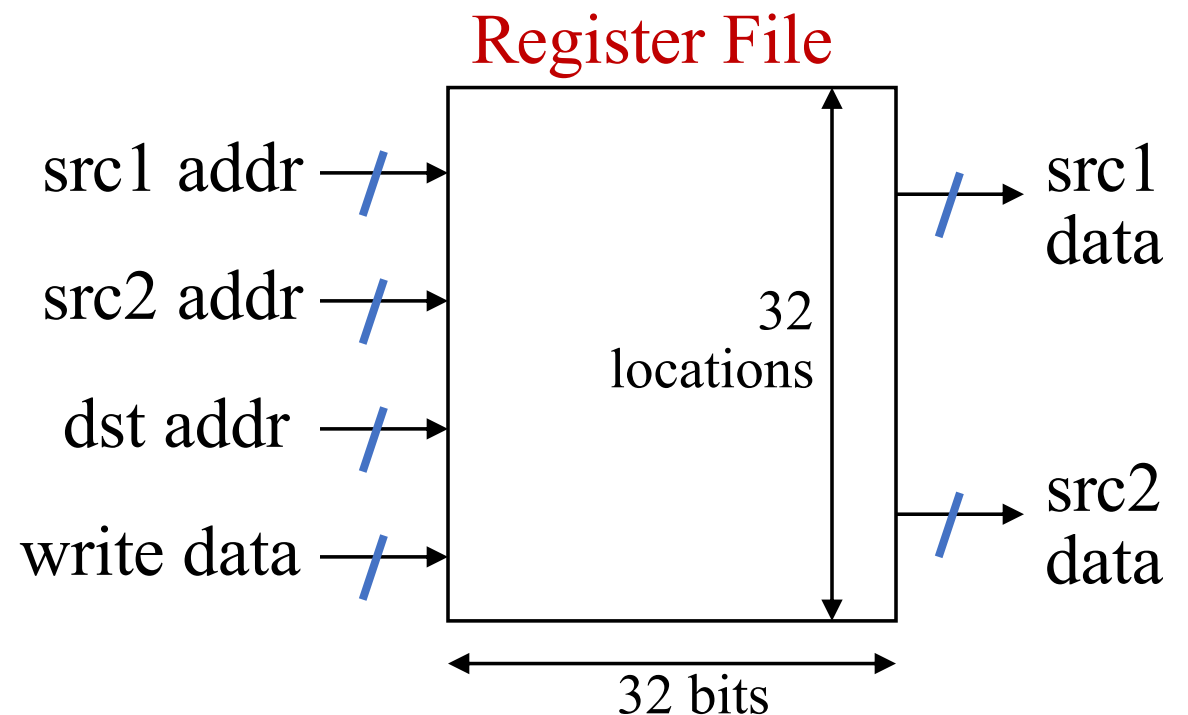
快

易于编译器使用

改善代码密度

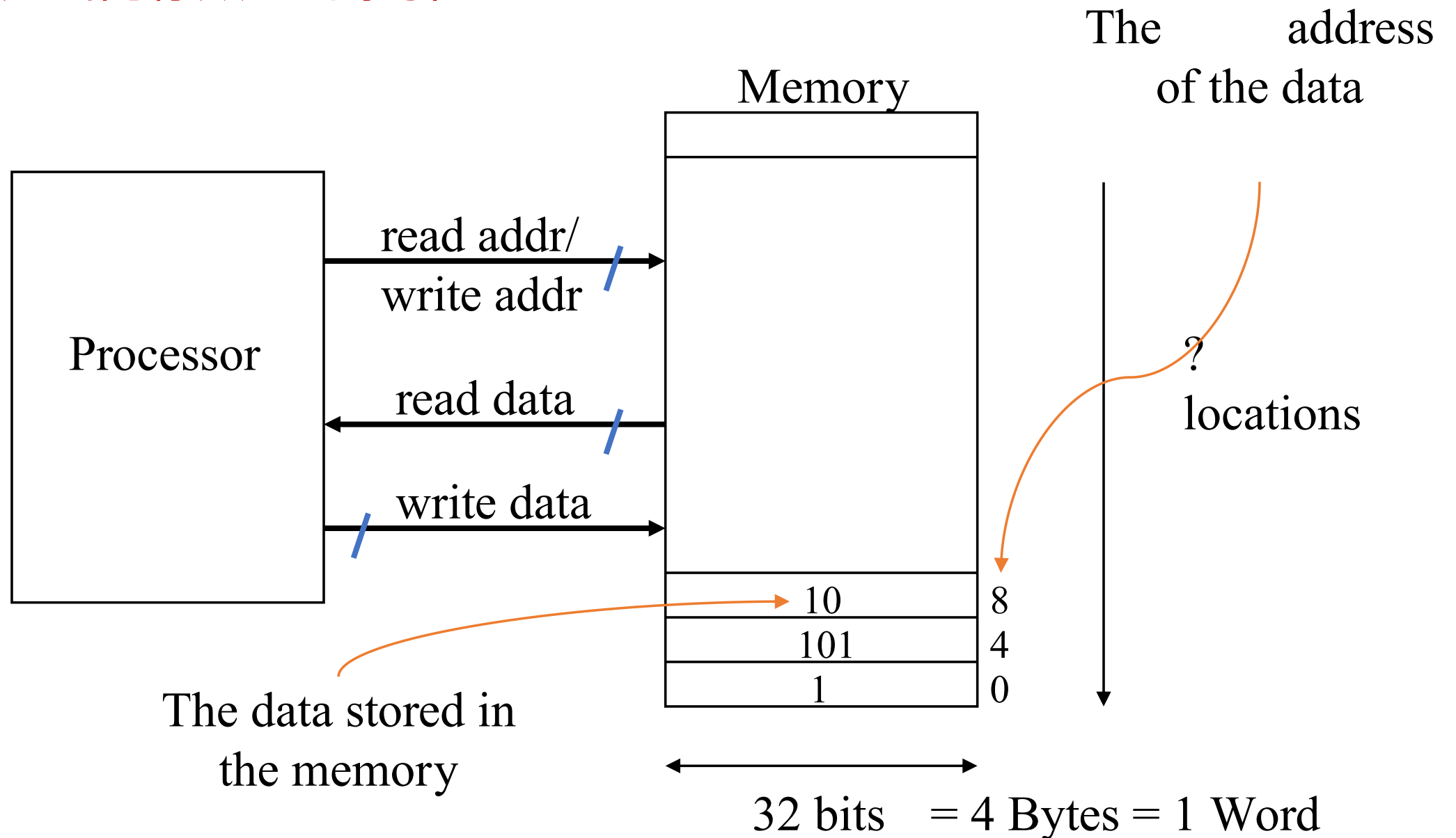
- 与内存地址相比，寄存器命名位数少

## □ 寄存器地址用\$ 表示



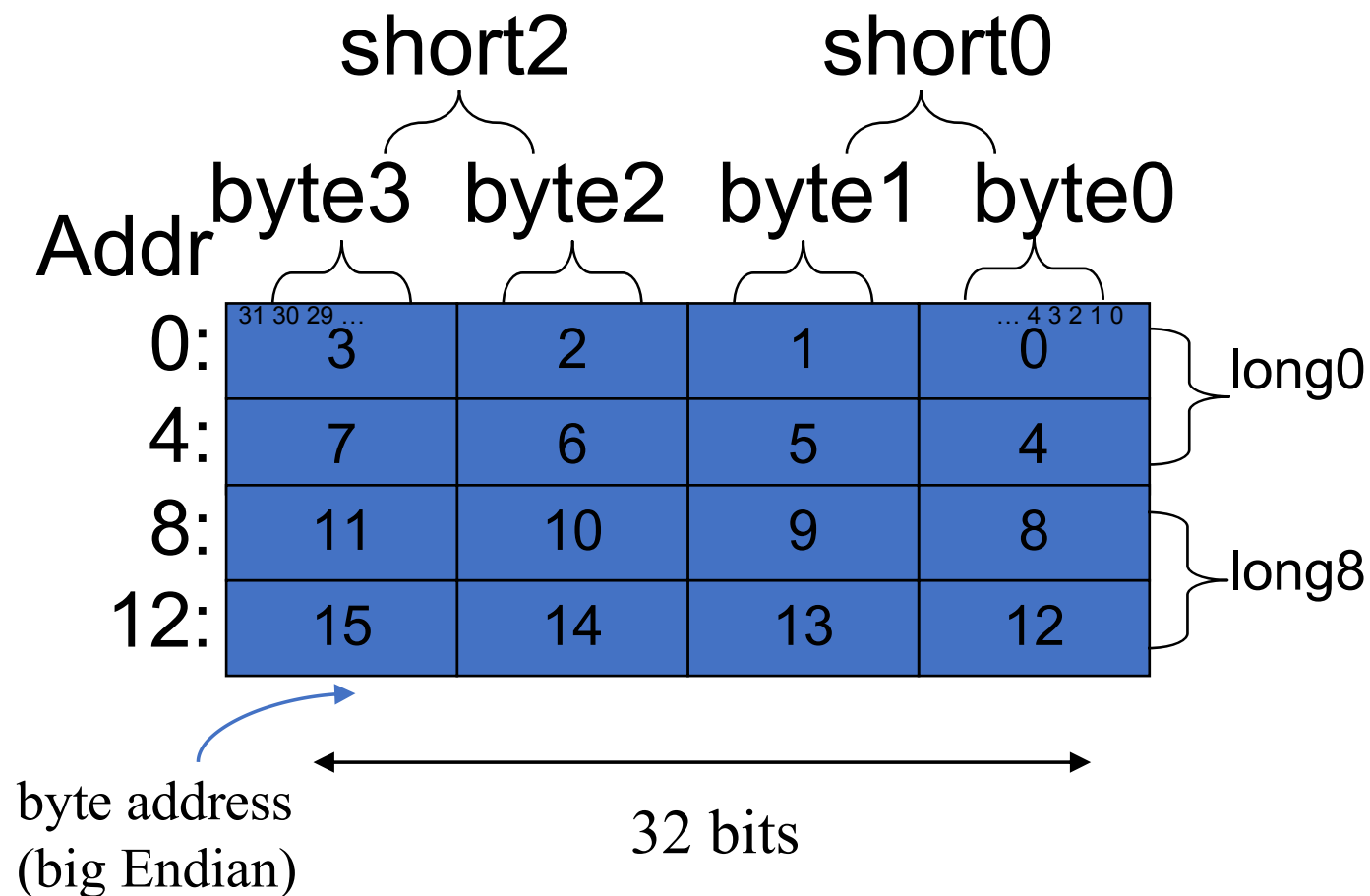
# 处理器 – 内存互连

- 内存是一个大的一维数组
- 地址充当内存数组的索引



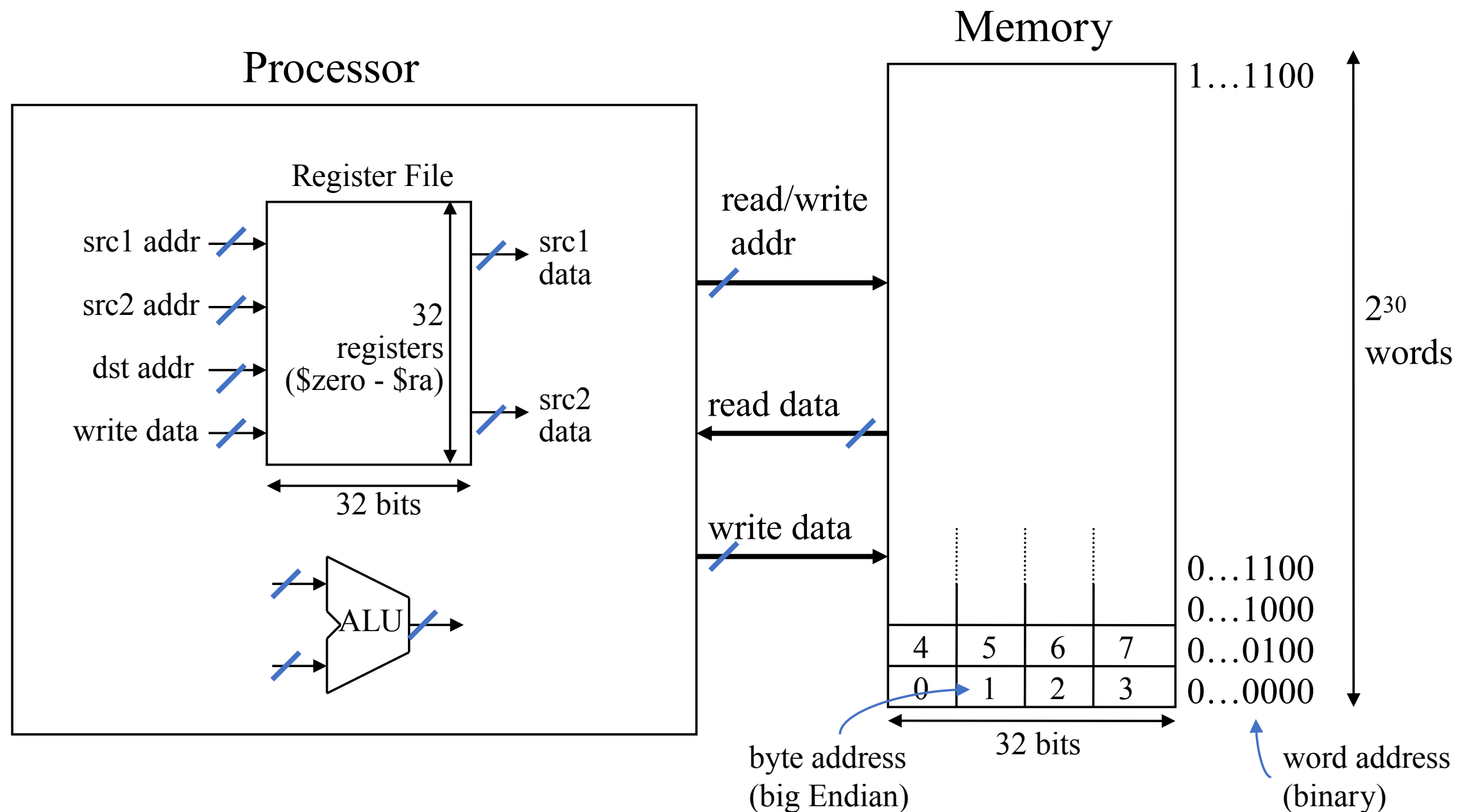
# MIPS 内存单元

- 内存字节编址
- 但是不同块大小怎样访问
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)



# MIPS Organization

- ❑ 算术运算指令 – to/from the register file
- ❑ 存取指令 - to/from memory

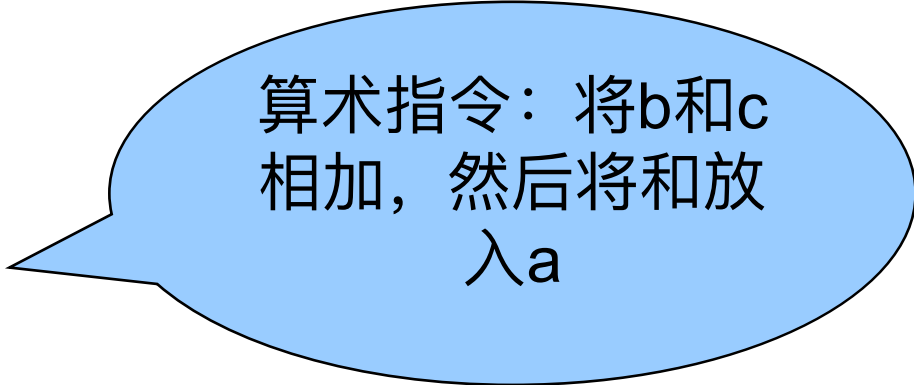


# 计算机硬件的操作

- 每台计算机都必须能够执行算术运算.

example:

**ADD** a , b , c



算术指令：将b和c  
相加，然后将和放  
入a

1. a,b,c ——操作数
2. 在计算机进行算术运算时，它们必须分配到寄存器中.
3. 大部分计算机都使用字节地址。

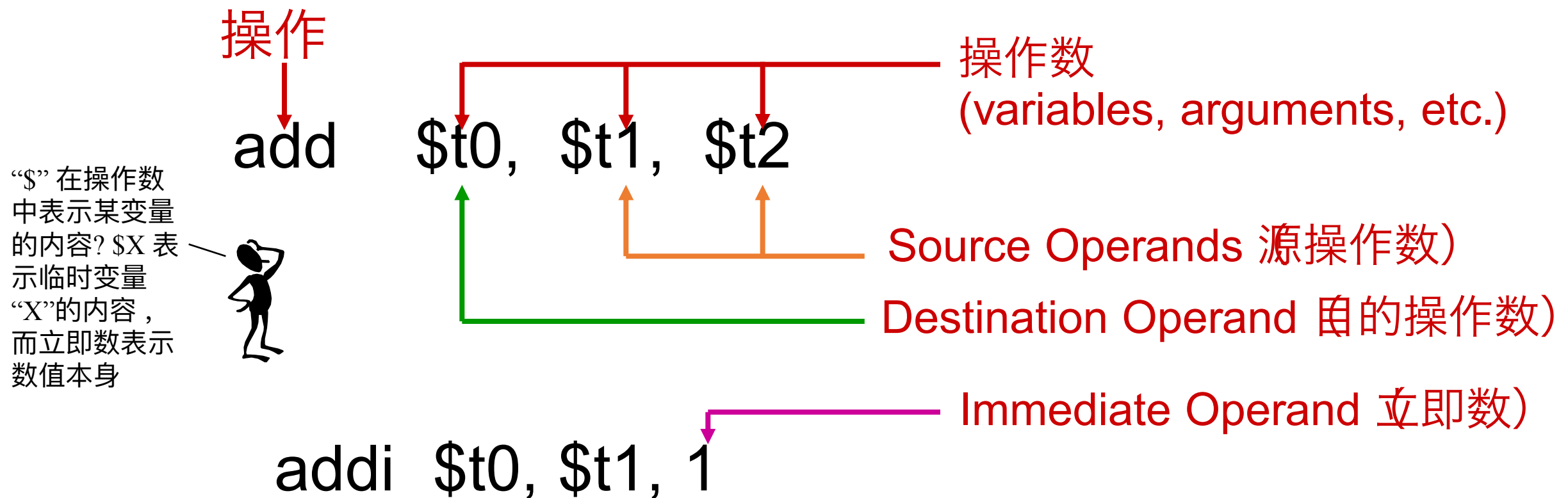


# Anatomy of an Instruction

## 指令剖析

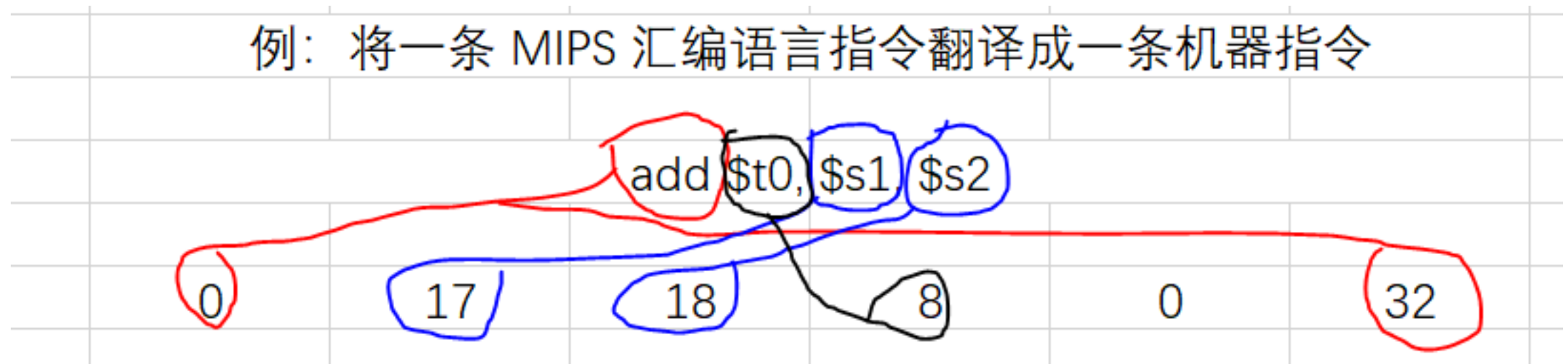
- 计算机执行一组称为指令的原始操作
  - 指令规定操作和操作数（完成操作的变量）
  - 操作数类型: 立即数, 源操作数, 目的操作数

指令在计算机内部是以若干个或高或低的电信号的序列表示的, 并且形式上和数的表示相同。



# 计算机中指令的表示

- **字段 (field)**：机器指令可以分为若干字段。如下例，



- **指令格式 (instruction format)**：二进制数字段组成的指令表示的形式。从位的数目上来看，MIPS 指令占 32 位，与数据字的位数相等。为了遵循简单源于规整的原则，所有的 MIPS 指令都是 32 位长。
- **机器语言 (machine code)**：在计算机系统中用于交流的二进制表示形式。其指令序列叫做**机器码 (machine code)**。

# MIPS 指令系统

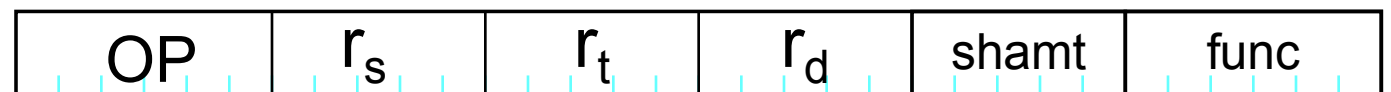
## ❖ MIPS指令格式

- 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 ( $t0 = s1 + s2$ )
  - 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

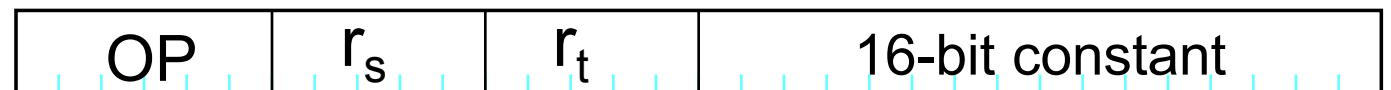
# MIPS Instruction Formats

- MIPS 所有指令都是 32-bit word
- 每条指令由不同“字段”组成:
  - 一个“OPCODE”操作码 6位
    - 规定执行什么操作(fewer than 64)
  - 3个5位操作数字段
    - 规定使用哪个寄存器 (one of 32) 源/目的
  - 嵌入常数
    - 叫做“literals” or “immediates”立即数
    - 不同操作下, 有16-bits, 5-bits or 26-bits 几种情况
    - 有时作为有符号数, 有时作为无符号数
- 有三种指令格式:

- **R-type**, 3 register operands (2 sources, destination)



- **I-type**, 2 register operands, 16-bit constant



- **J-type**, no register operands, 26-bit constant



# R-Format Instructions

- 分别定义以下位数的“字段”： $6 + 5 + 5 + 5 + 5 + 6 = 32$
- 为了简化，每个字段有个名字：

6	5	5	5	5	6
---	---	---	---	---	---

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

- **重要提示：**在这些幻灯片和书中，每个字段都被视为 5 位或 6 位无符号整数，而不是 32 位整数的一部分
- 结论：5 位字段可以表示 0-31 的任意数字，而 6 位字段可以表示 0-63 的任意数字

# R-Format Instructions

- 字段的含义:

- rs (Source Register源寄存器): 通常用于指定第一个操作数的寄存器
- rt (Target Register 目标寄存器): 通常用于指定第二个操作数的寄存器

Operand 操作数 (注意名称具有误导性)

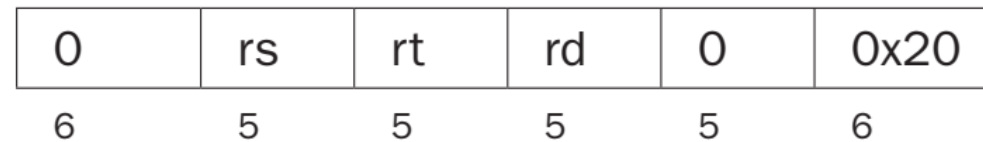
- rd (Destination Register目的寄存器): 通常用于指定计算结果存放寄存器

- 后面的字段:

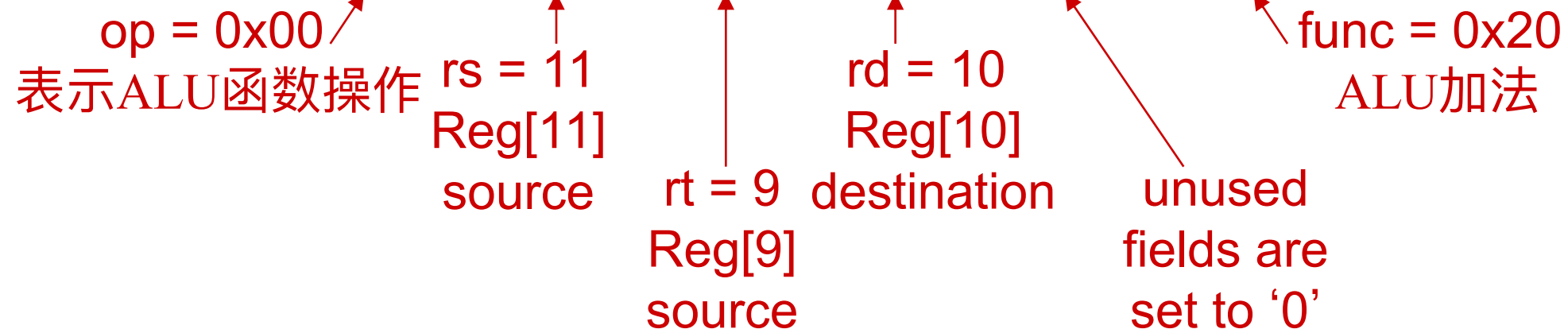
- shamt: 此字段包含移位指令的移位数量。将一个 32 位字移位超过 31 是没有意义, 所以这个字段只有 5 位 (它可以表示数字 0-31)
- 除移位指令外, 该字段均设置为 0

# MIPS ALU Operations

## 示例编码操作: ADD 指令



R-type: 0000000010110100101010000001000000



add \$10, \$11, \$9 (“汇编语言表示”)

MIPS 汇编语言的约定是首先指定目标操作数,  
然后是源操作数.

add rd, rs, rt:

$\text{Reg}[\text{rd}] = \text{Reg}[\text{rs}] + \text{Reg}[\text{rt}]$

“将 rs 的内容与 rt 的内容相加,  
将结果存储在 rd”中

相类似的其它ALU 操作:

算术运算: add, sub, addu, subu

比较: slt, sltu

逻辑: and, or, xor, nor

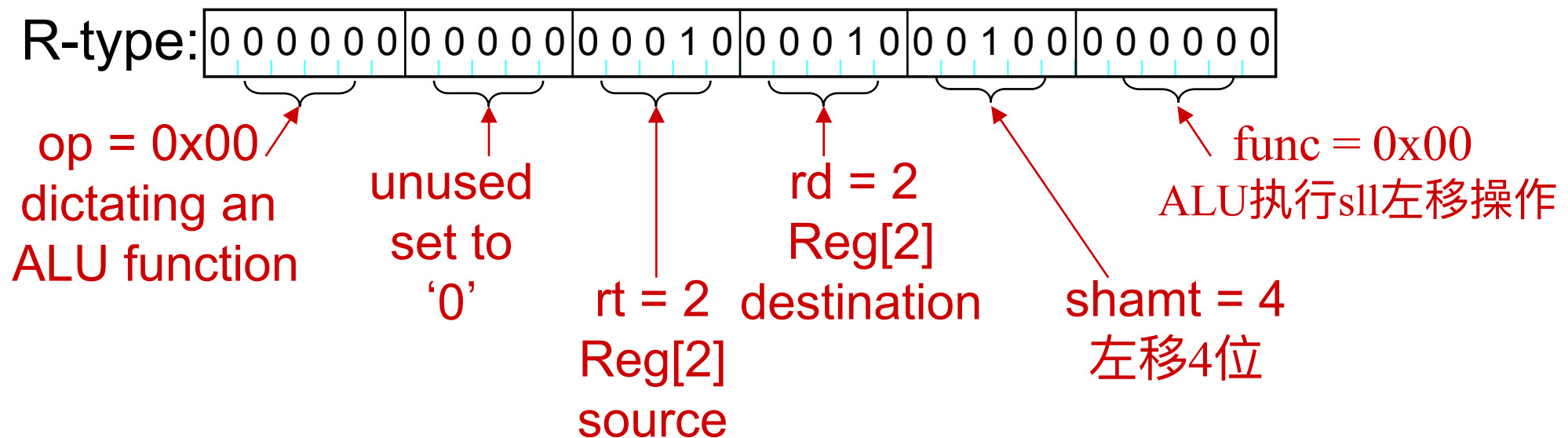
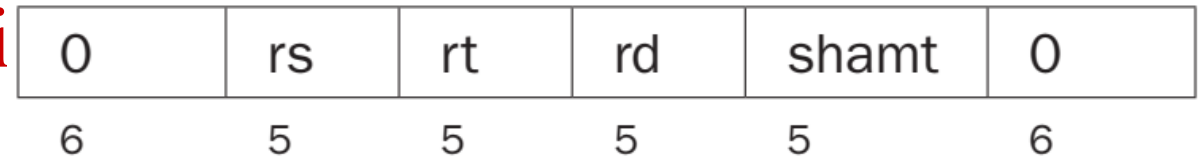
Shift移位: sll, srl, sra, sllv, srav, srlv

对寄存器内  
容的引用以  
“\$”为前缀,  
以将它们与  
常量或内存  
地址区分开  
来



# MIPS Shift Operations

SHIFT LOGICAL LEFT instruction  
左移指令



Assembly: `sll $2, $2, 4`

`sll rd, rt, shamt:`

$\text{Reg}[\text{rd}] = \text{Reg}[\text{rt}] \ll \text{shamt}$

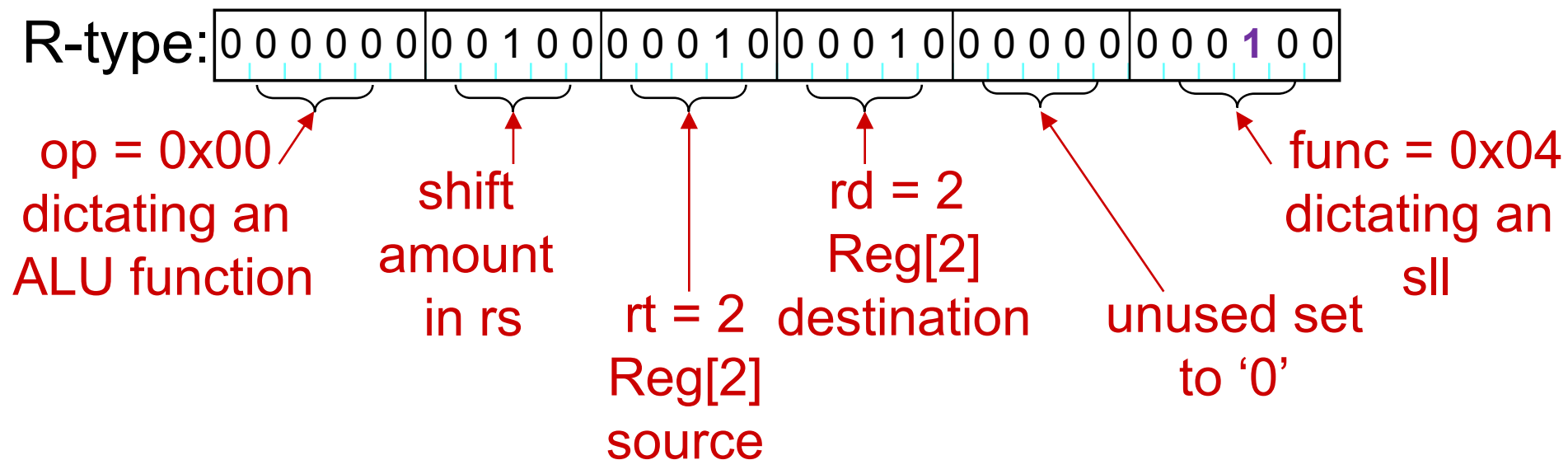
“*rt*内容左移 *shamt*; 结果存于*rd*中”

How are shifts useful?

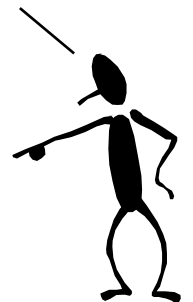


# MIPS Shift Operations

SLLV (SLL Variable), 左移位数存在寄存器RS中



这是 MIPS 的特殊语法，在此 ALU 指令中，rt 操作数在 rs 操作数之前。通常情况是相反的



移位量不在指令中，  
而是在寄存器中

Assembly: `sllv $2, $2, $8`

`sllv rd, rt, rs:`

$\text{Reg}[\text{rd}] = \text{Reg}[\text{rt}] \ll \text{Reg}[\text{rs}]$

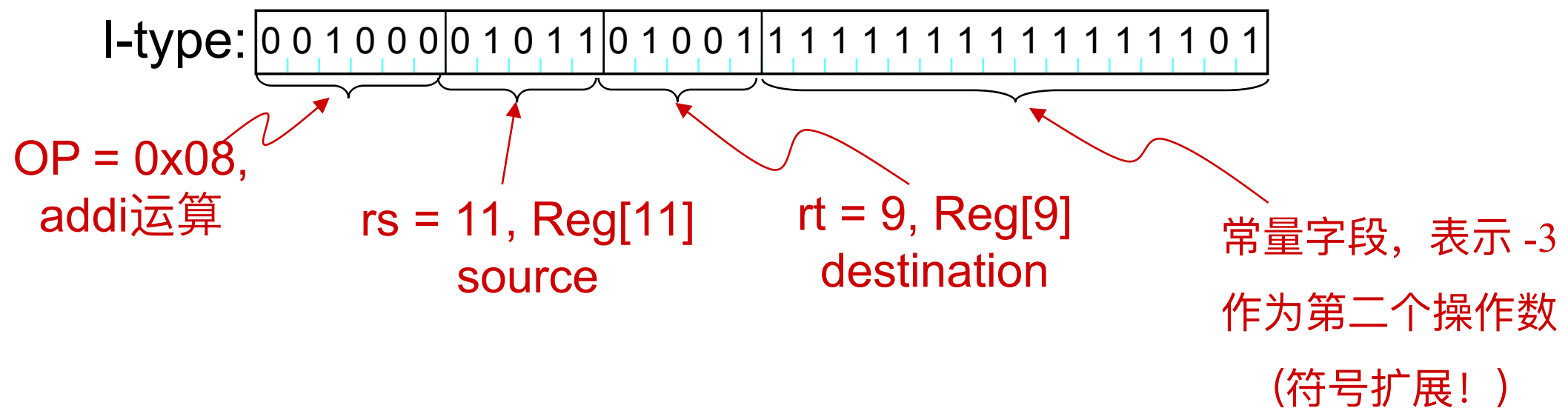
0	rs	rt	rd	0	4
6	5	5	5	5	6

“将 rt 的内容左移 rs 的内容；将结果存储在 rd 中”

# MIPS ALU Operations with Immediate

addi 指令: 寄存器内容与有符号数加法:

8	rs	rt	imm
6	5	5	16



汇编表示: `addi $9, $11, -3`

`addi rt, rs, imm:`

$\text{Reg}[rt] = \text{Reg}[rs] + \text{sxt}(imm)$

“将rs的内容与常数相加; 将结果存储在rt中”

其它类似ALU指令:

算术运算: `addi`, `addiu`

比较: `slti`, `sltiu`

逻辑: `andi`, `ori`, `xori`, `lui`

算术运算和比较运算要对立即数进行符号扩展, 但逻辑运算不需要符号扩展。



# 为什么指令里放常数? (立即数)

- constants常数/immediates立即数 有用吗?
  - 小的常数在程序中使用几率 (50% 操作数)
    - C编译器 (gcc) ALU 中 52% 的操作数是常数
    - 电路仿真中 (spice) 69% 涉及常数
    - e.g.,  $B = B + 1$ ;  $C = W \ \& \ 0x00ff$ ;  $A = B + 0$ ;
- 示例:

**addi        \$29, \$29, 4**

8	rs	rt	imm
6	5	5	16

**slti        \$8, \$18, 10**

0xa	rs	rt	imm
6	5	5	16

**andi        \$29, \$29, 6**

0xc	rs	rt	imm
6	5	5	16

**ori        \$29, \$29, 4**

0xd	rs	rt	imm
6	5	5	16

# MIPS 编程示例 (片段)

- 假设计算下列表达式:

$$f = (g + h) - (i + j)$$

- 变量f, g, h, i, 和 j 分别放在寄存器\$16, \$17, \$18, \$19, 以及 \$20中, MIPS 汇编代码怎样写?

```
add $8, $17, $18      # (g + h)
add $9, $19, $20      # (i + j)
sub $16, $8, $9       # f = (g + h) - (i + j)
```

- 回答问题:

- 这些变量是如何放在寄存器中的?
- 答: 我们需要更多的指令来解决, 可以把数据从内存取到寄存器, 并从寄存器存储到内存中的指令。

# MIPS 寄存器使用约定

- MIPS 一些特定用途的寄存器
  - \$0 寄存器 硬件设计为常数零

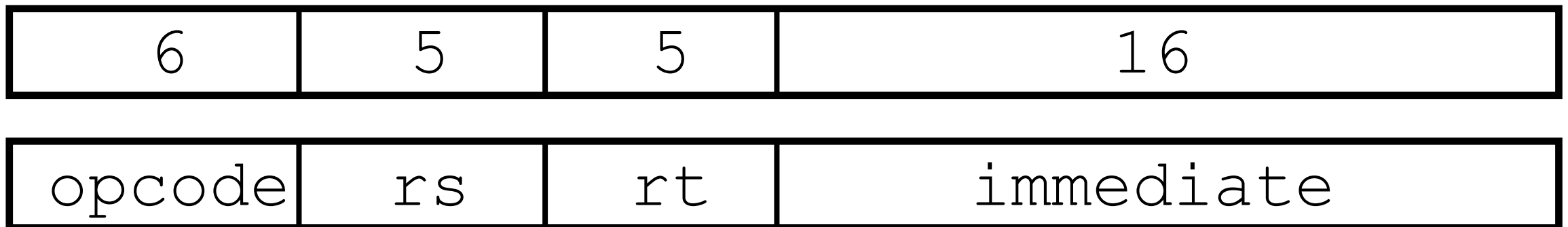
Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

# I-Format Instructions

## 立即数-格式指令

- 立即数指令怎么样?
  - 5位的字段最大可表示的数为31: 立即数的值用几位数?
  - 理想化, MIPS 只有一种指令格式(for simplicity): 我们需要折中, 兼顾不同情况。
- 定义与 R 格式部分一致的新指令格式:
  - 首先注意, 如果指令有立即数, 那么它最多使用 2 个寄存器, R指令用3个寄存器。

定义以下位数的“字段”:  $6 + 5 + 5 + 16 = 32$  位



**关键概念:** 只有一个字段与 R 格式不一致。最重要的是, 操作码仍然在同一个位置!

# Working with Constants

## 指令中使用常量

- 立即数指令允许在指令中指定常量

- 举例

- add 2000 to register \$5

- ```
addi $5, $5, 2000
```

- subtract 60 from register \$5

- ```
addi $5, $5, -60
```

- ... no **subi** instruction!

- logically AND \$5 with 0x8723 and put the result in \$7

- ```
andi $7, $5, 0x8723
```

- put the number 1234 in \$10

- ```
addi $10, $0, 1234
```

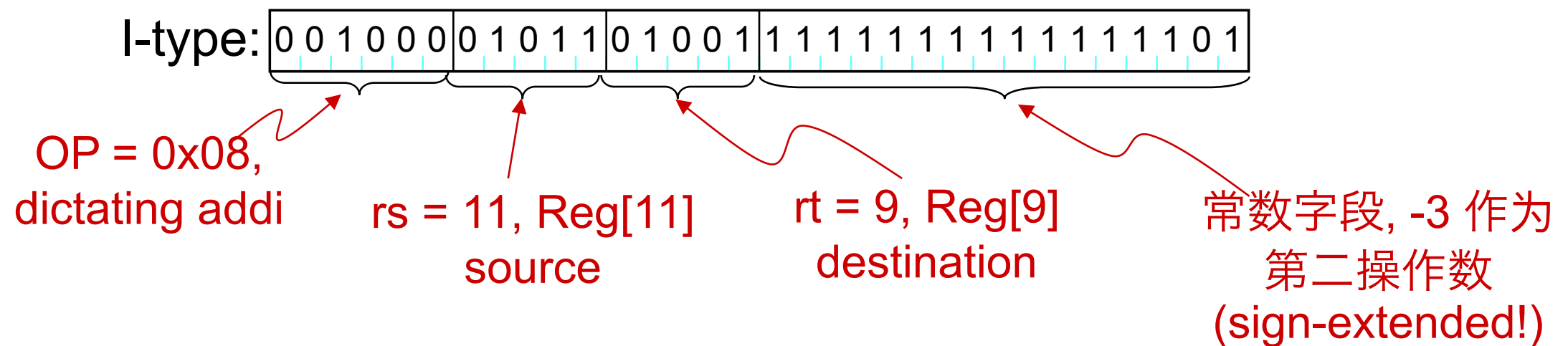
- 但是...

- 这些常数限定在16位!

- 可表示范围: 有符号数[-32768...32767], 或无符号数 [0...65535]

# ADDI指令回顾

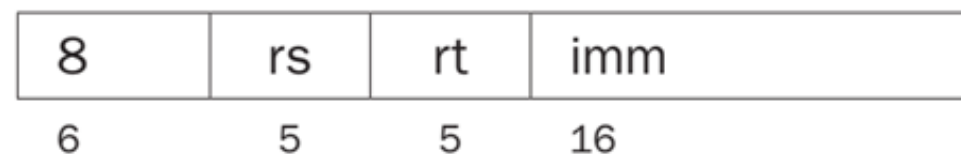
addi 指令: 寄存器内容与有符号常数相加:



汇编表示: `addi $9, $11, -3`

`addi rt, rs, imm:`

$\text{Reg}[rt] = \text{Reg}[rs] + \text{sxt}(imm)$

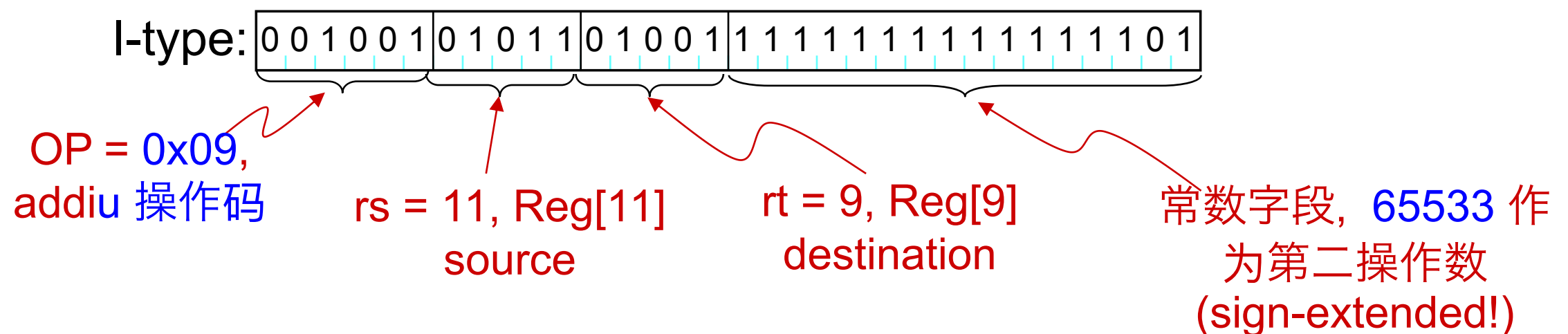


“寄存器rs内容与有符号常数相加;  
结果存于rt中”



# ADDIU: Signed vs. Unsigned Constants

addiu 指令:寄存器内容与符号常数相加:

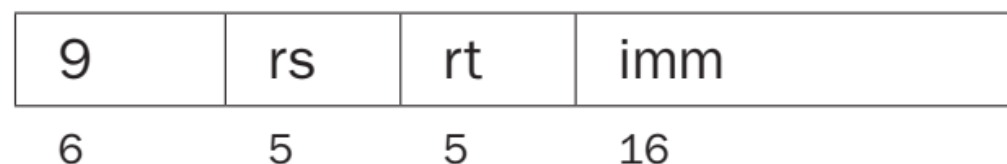


汇编表示: addiu \$9, \$11, 65533

addiu rt, rs, imm:

$\text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + \text{sxt}(\text{imm})$

“寄存器rs内容与符号常数相加;  
结果存于rt中”



逻辑操作指令的常数，是  
“unsigned”，零扩展，  
总是无符号数。

**ADDI、ADDIU：**

**加立即数，区别在于是否检测溢出。无符号加不检测溢出。**

**ADD、ADDU：加寄存器，区别在于是否检测溢出。无符号加不检测溢出。**

即，在忽略溢出的前提下，ADDI与ADDIU等价，ADD与ADDU等价。对于CPU来说，**有符号或者无符号都不重要**。他们的运算都是一样的，不管有没有符号位，都是从最低位加，进位，一直到最高位，**区别在于是否具有溢出检测**。

# 更大的常数怎么处理？

- Problem: 更大的常数（大于16位表示）怎样工作？
  - Example: 把32-bit 数值**0x5678ABCD** 存入 \$5, 怎么处理？
  - CLASS: 你怎样做？

- 解一:

- 把高位的16位 (0x5678) 先存入 \$5
  - 然后左移\$5 内容16 位, (0x5678 0000)
  - 再 与低16位相加“add” (0x5678 0000 + 0xABCD)

```
addi $5, $0, 0x5678
```

```
sll  $5, $5, 16
```

```
addi $5, $5, 0xABCD
```

- 小问题:

- **addi** 处理常数为有符号数, 会造成错误
  - 相加时要用**ori** 逻辑指令替代。

# 更大的常数怎么处理续？

- 观察：前面两步经常遇到！
  - 用一条新加的指令来完成
  - 前面两步(**addi + sll**) 组合为

**lui**

“装入高16位立即数”

16-bit 立即数放入寄存器的前半部分

- 举例：32-bit 0x5678ABCD 数值放入 \$5中

**lui \$5, 0x5678**

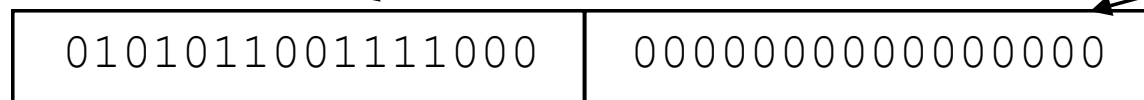
**ori \$5, \$5, 0xABCD**

# 更大的常数怎么处理续？

- 更仔细地观察:

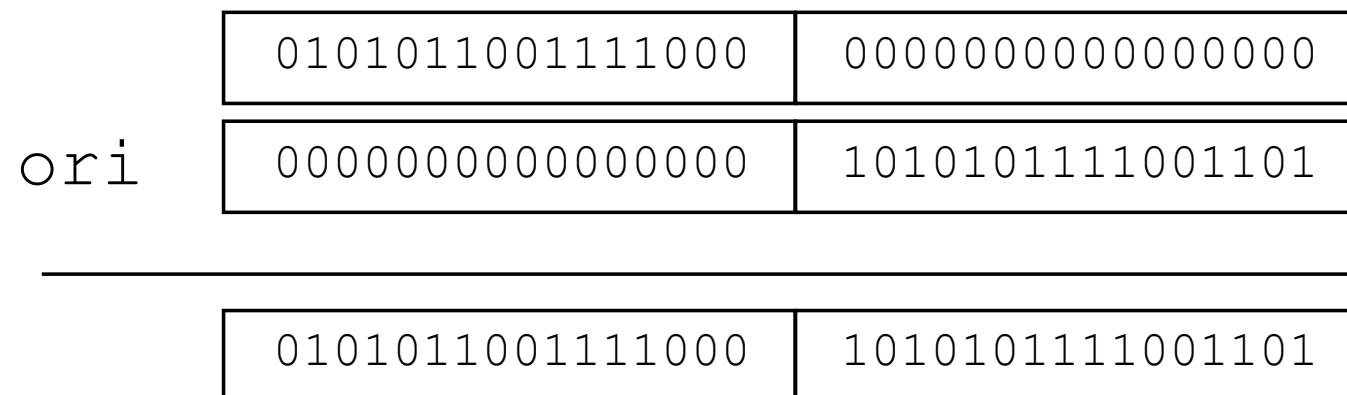
- “装入前半部分立即数”

lui \$5, 0x5678 // 0101 0110 0111 1000 in binary



- 低16位正确写入

ori \$5, \$5, 0xABCD // 1010 1011 1100 1101



提醒：在 MIPS 中，立即数逻辑指令（ANDI、ORI、XORI）不会对它们的常量操作数进行符号扩展



# Accessing Memory

## 内存访问

- MIPS 是一种“加载存储”架构
  - 算数运算单元ALU 的所有指令的操作数都在寄存器或是立即数中
  - 不能与存在存储器中的数值直接运算
    - 必须首先将值从内存中装入寄存器(called LOAD)
    - 计算结果存回内存中(called STORE)

# MIPS Load 装入指令



- Load 指令也是 I-type 指令

`lw rt, imm(rs)`

含义:  $\text{Reg}[\text{rt}] = \text{Mem}[\text{Reg}[\text{rs}] + \text{sign-ext}(\text{imm})]$

缩写: `lw rt, imm` for `lw rt, imm($0)`

- 做如下操作:
  - 取出寄存器 \$rs 的值
  - 与立即数（有符号数）相加
  - 相加结果作为内存地址索引
  - 相应地址中取出的值存入寄存器 \$rt 中

# MIPS Store 写入内存指令



- Store 指令也是I-type指令

`sw rt, imm(rs)`

含义:  $\text{Mem}[\text{Reg}[\text{rs}] + \text{sign-ext}(\text{imm})] = \text{Reg}[\text{rt}]$

缩写: `sw rt, imm` for `sw rt, imm($0)`

- 做如下操作:
  - 取出寄存器\$rs的值
  - 与立即数（有符号数）相加
  - 相加结果作为内存地址索引
  - 读出寄存器\$rt 的内容写入到内存此地址单元中



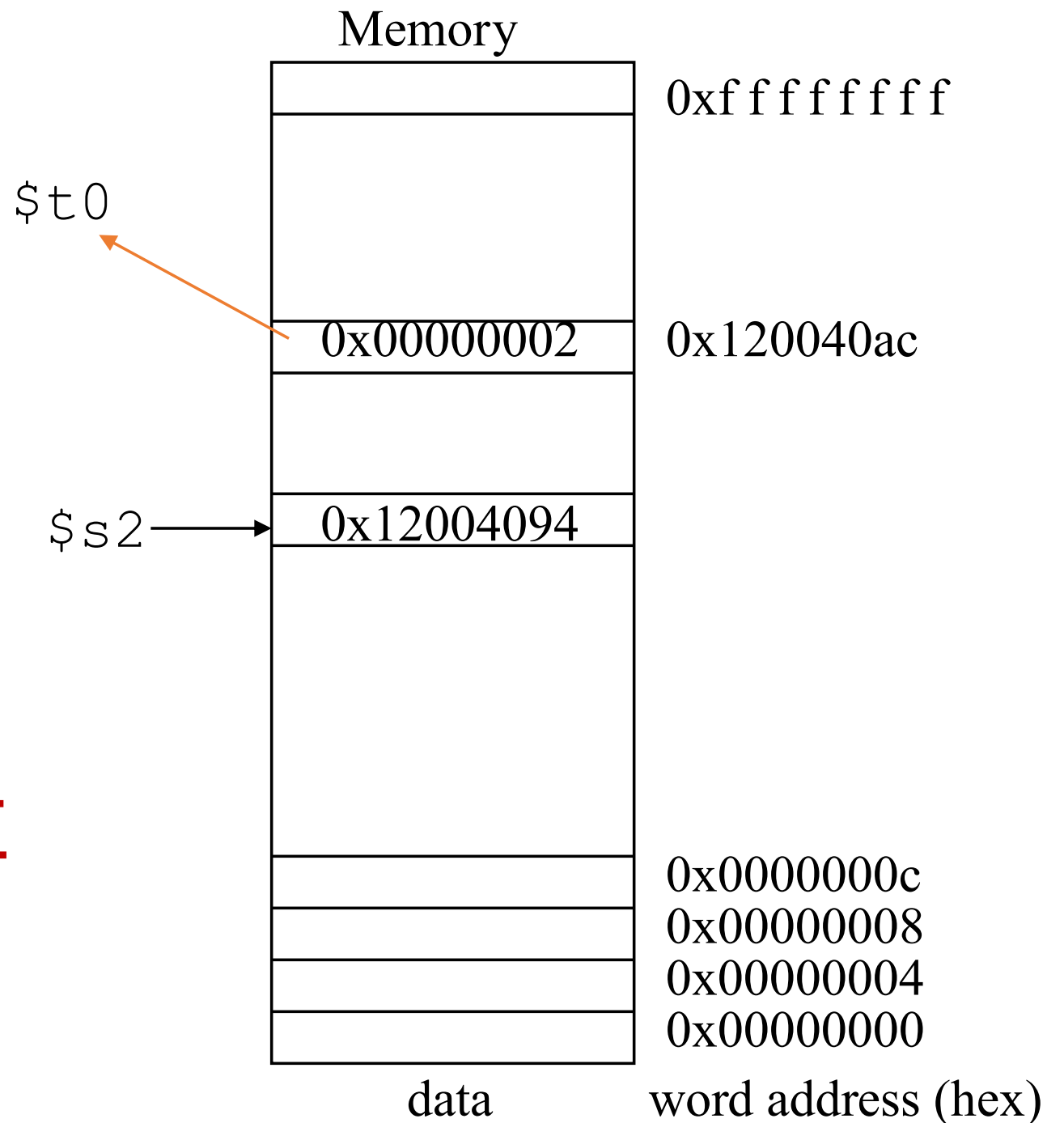
# Memory Address Location

- 举例说明: `lw $t0, 24($s2)`

$$24_{10} + \$s2 =$$

$$\begin{array}{r} \dots 1001\ 0100 \\ + \dots 0001\ 1000 \\ \hline \dots 1010\ 1100 = \\ 0x120040ac \end{array}$$

注意 **offset** 偏差值可 **正**  
可 **负**



# MIPS Memory Addresses

- **lw** 和 **sw** 指令读或写的内容是 32-bit 一个字，内存中占4个字节地址
  - 因此, 地址计算必须是4的倍数
    - $\text{Reg}[\text{rs}] + \text{sign-ext}(\text{imm})$  二进制数的后两位必须是“00”
  - 否则: 运行时会出现异常
- 这类指令也有针对字节访问的指令
  - **lb** (load byte)
  - **sb** (store byte)
    - 工作方式相同, 但它们的地址不必是 4 的倍数

# 编程存储惯例

编译时分配内存地址

- 数据和变量放在内存中
- 操作数放在寄存器中
- 计算的临时结果也放在寄存器中

1000:	n
1004:	r
1008:	x
100C:	y
1010:	

编译器翻译成

```
lw      $t0, 0x1008($0)
addiu   $t0, $t0, 37
sw      $t0, 0x100C($0)
```

或更友好地,

x= **【0x1008】**

y= **【0x100C】**

```
lw      $t0, x
addiu   $t0, $t0, 37
sw      $t0, y
```

rs 缺省为0寄存器Reg[0] (0)

```
int x, y;
y = x + 37;
```

编译方式: 加载、计算、存储

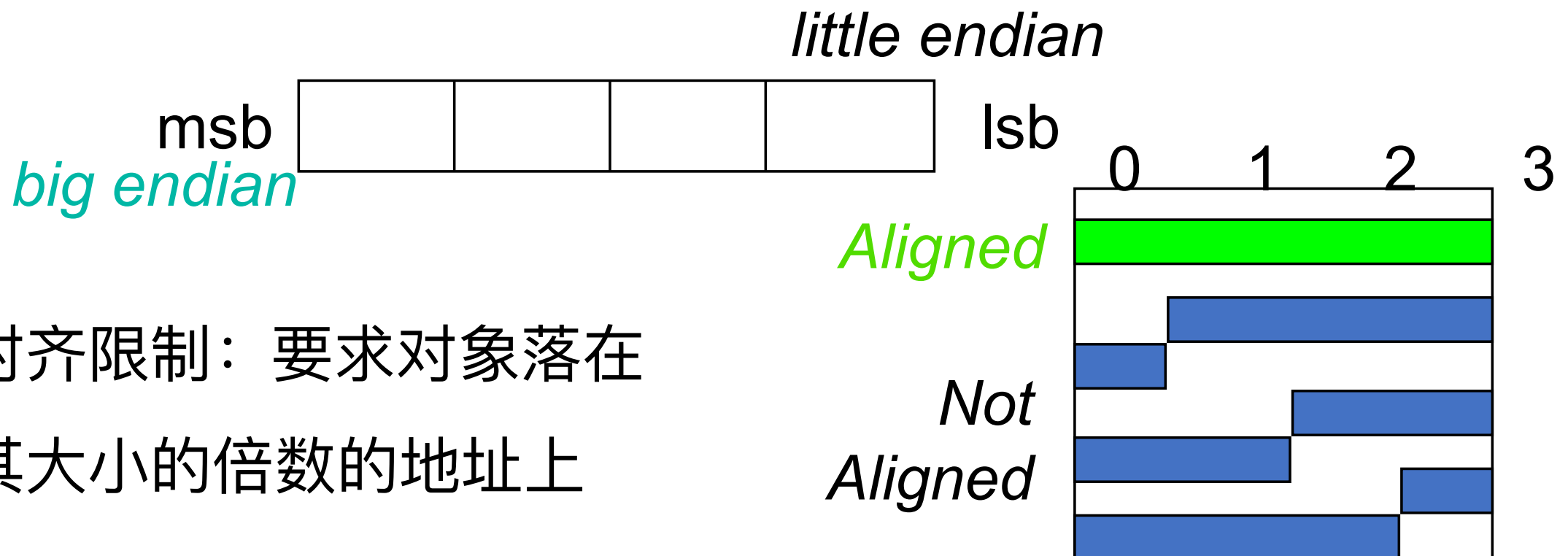
# 字节编址

- 由于大部分 ISAs 都支持字节(8 bits)存储, 以字节编址
- 因此, 内存字地址必须乘以4 (对齐约束)
- **Big Endian (大端存储)**: 存储字的最左边字节地址作为字地址

IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian (小端存储)**: 存储字的最右边字节地址作为字地址

Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



对齐限制: 要求对象落在  
其大小的倍数的地址上

# 内存读和写入字节

- MIPS 提供有内存字节操作指令

lbu/lb \$t0, 1(\$s3) #load byte from memory

sb \$t0, 6(\$s3) #store byte to memory

op	rs	rt	16 bit number
----	----	----	---------------

## □ 内存8位写入或读出的是什么？

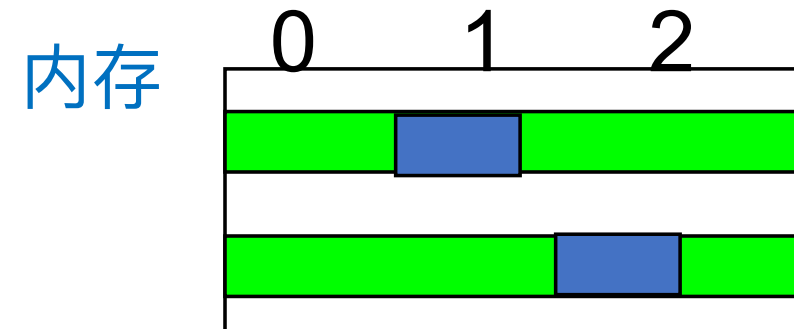
- | Load指令将内存中读出的字节放在目标寄存器的最右边 8 位

- | 寄存器中的其它位怎么办？LOAD指令对其它位进行0扩展。

- | store 指令从寄存器中的最右边8位取出一个字节，把它写入内存字节处

- 内存字的其它字节不变

寄存器 0000...0000



寄存器

# 举例说明

- 给定如下代码，执行完的结果是什么？

```
add $s3, $zero, $zero
```

```
lbu $t0, 1($s3)
```

```
sb $t0, 6($s3)
```

Memory	
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F 12 9 F F F	4
0x 0 0 9 0 1 2 A 0	0
Data	字地址(Decimal)

- \$t0寄存器的左边值？

\$t0 = 0x00000090

- 内存字有变化吗，变成什么？

mem(4) = 0xFFFF90FF

- 如果机器是小端存储little Endian，结果是什么？

\$t0 = 0x00000012

mem(4) = 0xFF12FFFF

# Loading and Storing 半字操作

- MIPS 也提供了内存半字操作

lh/lhu \$t0, 1(\$s3) #load half word from memory

sh \$t0, 6(\$s3) #store half word to memory

op	rs	rt	16 bit number
----	----	----	---------------

## □ 16 bits 内存取和写入内存

- | load 半字 从内存取出16位半字，放在目标寄存器的最右边，高位0扩展
- | store 半字，从寄存器的右边取出16位，写入内存半字，占两个地址
  - 内存字的其它部分不变

# Lb与lbu, lh与lhu

指令lb 和 lh 装入的数据进行如下扩展:

- lbu, lhu 是零扩展
- lb, lh 是符号扩展



# MIPS Branch Instructions

## 条件跳转指令

MIPS 分支指令提供了一种有条件地将 PC 更改到某个附近位置的方法...



Op=4

`beq rs, rt, label # Branch if equal`

```
if (REG[RS] == REG[RT])  
{  
    PC = PC + 4 + 4*offset;  
}
```

Op=5

`bne rs, rt, label # Branch if not equal`

```
if (REG[RS] != REG[RT])  
{  
    PC = PC + 4 + 4*offset;  
}
```



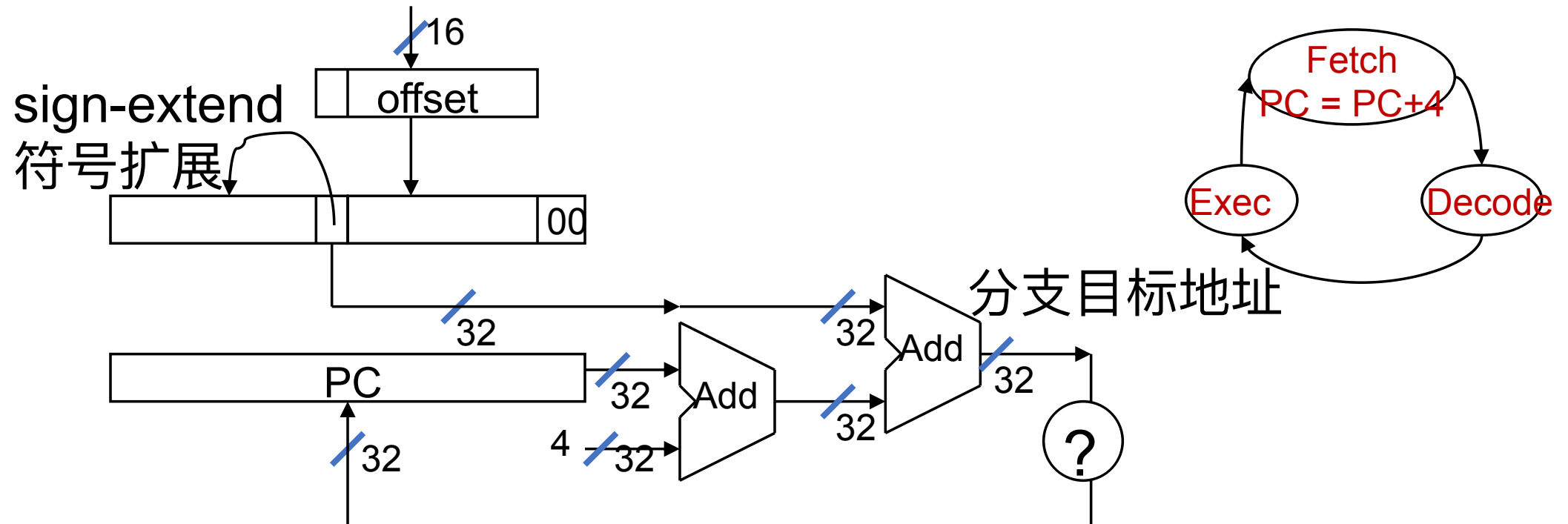
注意内存地址必须是4的倍数, 目标地址也必须是4的倍数

注意: 跳转目标规定的是相对于它的下一条指令的偏移 (默认取下一条指令). 汇编器计算出相对于目标地址 (通常用label表示) 的偏移量, 偏移量常数字段16位限定了跳转范围.

# 拆解 Branch 目标地址

- 更新后的 PC ( $PC+4$ ) 内容与变换后的32位偏移量相加，16 位分支偏移量通过以下公式转换为 32 位的值
  - 16-bit 偏移量后面加两个0，然后符号扩展为32-bit.
- 如果分支条件为真，则在下一个取周期之前将结果写入 PC

从分支指令的低 16 位得到



# Offset Tradeoffs

- 为什么不直接存偏移量的低16位，这样就不用加低阶的两个零了，造成混乱，...
- 这样将造成跳转距离缩短到离目标地址  $-2^{13}$  到  $+2^{13}-1$  的范围。
- 而加两个零的额外硬件成本非常低，并且对时钟周期时间没有影响

# 分支指令举例

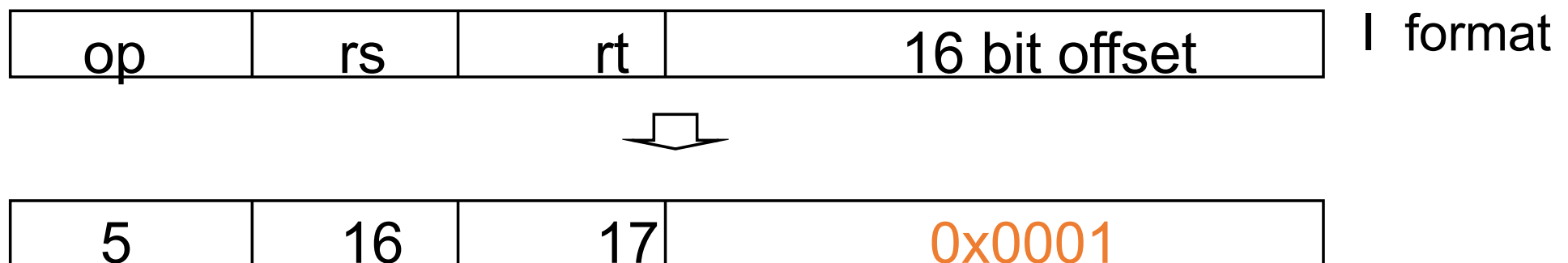
- 汇编代码

bne \$s0, \$s1, Lbl1

add \$s3, \$s0, \$s1

Lbl1: ...

- bne指令的格式:



## □ 记住

在取指bne后, PC 更新为下一条指令地址 ( $PC = PC + 4$ ).

偏移量 (加上 2 个低位零) 符号扩展后与 (更新的) PC相加, 得到目标地址

# 编译 C 语言 if 指令到MIPS汇编 (1/2)

- C语言

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

- 变量映射:

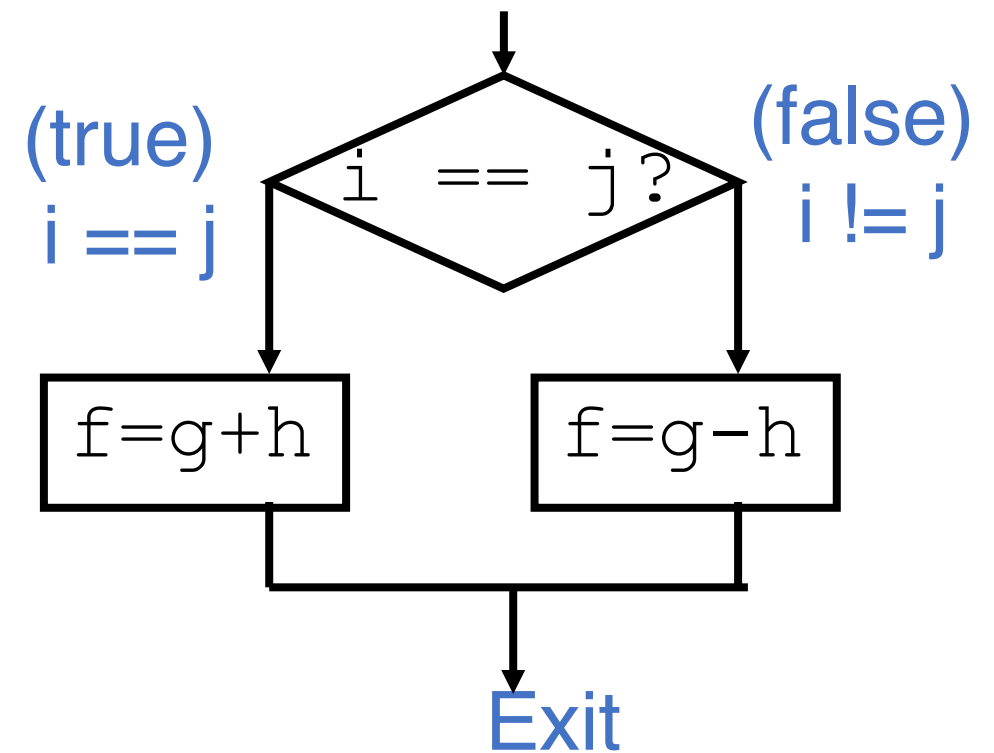
f: \$s0

g: \$s1

h: \$s2

i: \$s3

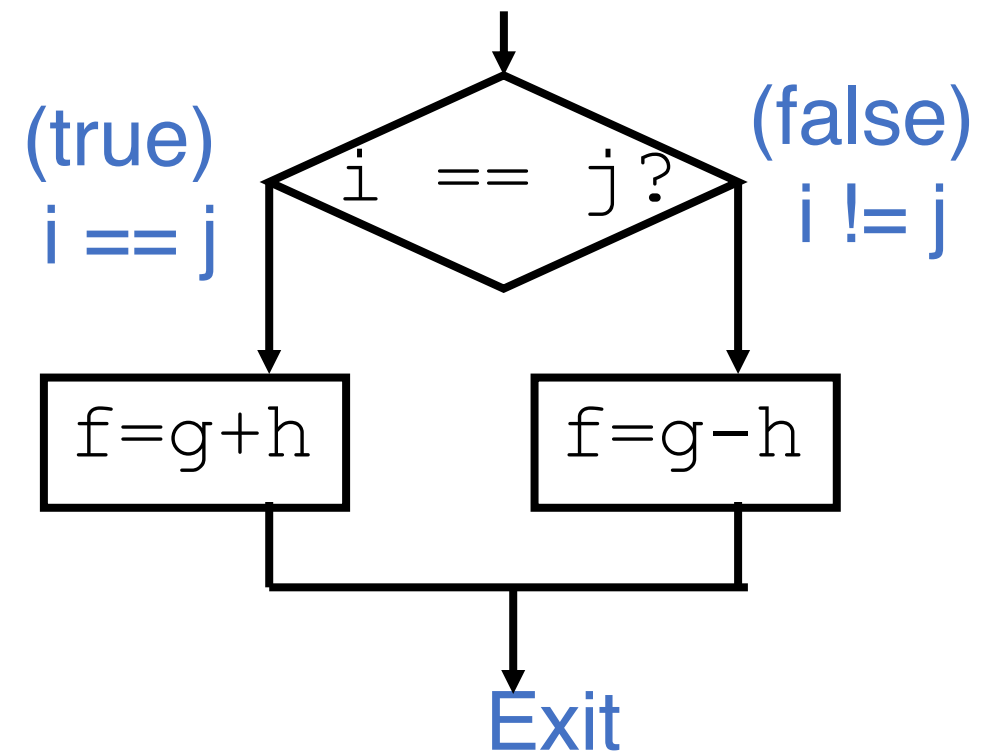
j: \$s4



# 编译 C 语言 if 指令到MIPS汇编(2/2)

## • C手动编译

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



## •编译成MIPS 代码:

```
0x00400020 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:
```



4	16	17	0x0002
---	----	----	--------

注意: 编译器自动为目标地址生成 标签labels.

# MIPS Jumps

## 绝对跳转

- MIPS **Branch** 分支指令的跳转范围被限制在离分支指令大约 $\pm 32K$  条指令 ( $\pm 128K$  字节)的范围.
- 跳的更远: 无条件跳转 **jump** 指令
- 指令:

**j label**                    # jump to label ( $PC = PC[31-28] \parallel CONST[25:0]*4$ )

**jal label**   # jump to label and **store PC+4 in \$31**

**jr \$t0**                    # jump to address specified by register's contents

**jalr \$t0, \$ra**            # jump to address specified by first register's contents

and **store PC+4 in second register**

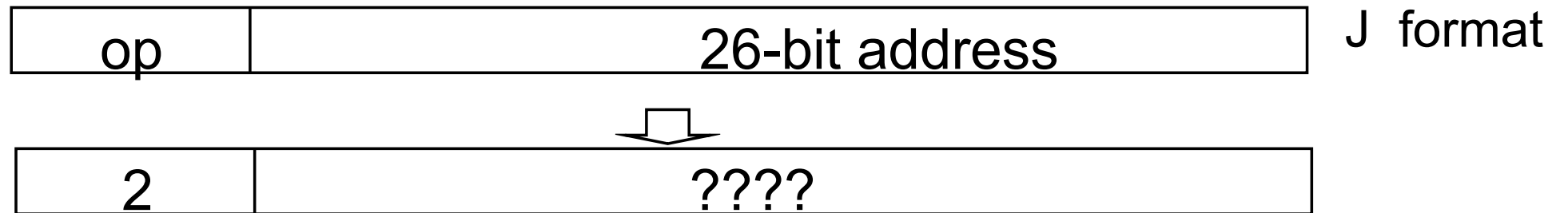
- Formats:
- |                         |        |                 |   |       |   |          |
|-------------------------|--------|-----------------|---|-------|---|----------|
| • J-type: used for j    | OP = 2 | 26-bit constant |   |       |   |          |
| • J-type: used for jal  | OP = 3 | 26-bit constant |   |       |   |          |
| • R-type, used for jr   | OP = 0 | $r_s$           | 0 | 0     | 0 | func = 8 |
| • R-type, used for jalr | OP = 0 | $r_s$           | 0 | $r_d$ | 0 | func = 9 |

# Jumps 指令

- Instruction:

j Lbl                      #go to Lbl

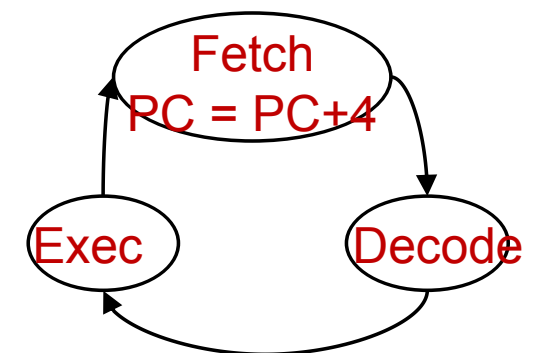
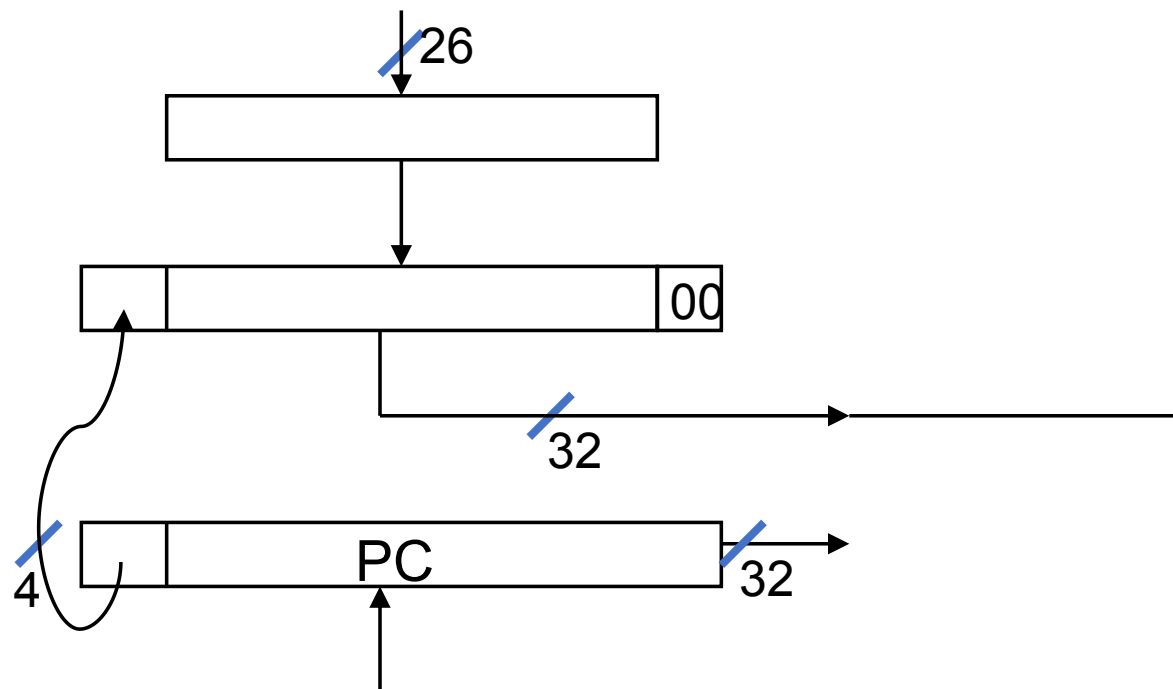
- 格式:



- jump 的目标地址怎样规定?

绝对地址的形成

- 指令中低26位, 后面加两个零
- 把PC+4的高4位与上面的28位合并为32位绝对地址





# 汇编 Branches and Jumps

- 为以下代码序列汇编 MIPS 机器代码（十进制即可）。假设 beq 指令的地址是 0x00400020hex

0x00400020	beq \$s0, \$s1, Else	4	16	17	0x0002
	add \$s3, \$s0, \$s1				
0x00400028	j Exit	2	0000 0100 0 ... 0 0011 00 <sub>2</sub>		
Else:	sub \$s3, \$s0, \$s1				
Exit:	...				

0x00400020	4	16	17	2	
0x00400024	0	16	17	19	0 0x20
0x00400028	2	0000 0100 0 ... 0 0011 00 <sub>2</sub>			
		jmp dst = (0x0) 0x040003 00 <sub>2</sub> (00 <sub>2</sub> )			
		= 0x00400030			
0x0040002c	0	16	17	19	0 0x22
0x00400030	...				

# 函数调用跳转指令的使用

C ... sum(a,b) ; ... /\* a,b:\$s0,\$s1 \*/  
}  
int sum(int x, int y) {  
 return x+y;  
}

---

address

M 1000 add \$a0,\$s0,\$zero # x = a  
1004 add \$a1,\$s1,\$zero # y = b  
1008 addi \$ra,\$zero,1016 # \$ra=1016  
P 1012 j sum # jump to sum  
S 1016 ...  
2000 sum: add \$v0,\$a0,\$a1  
2004 jr \$ra # return instruction

•问: 这里为什么用jr 而不是用j?

- 答: sum 函数调用可能发生在任何地方, 不是返回固定地址. 调用子程序sum 必须能够返回到调用的位置.

# Jal 子程序调用跳转指令

- **jal语法** (jump and link) is same as for j (jump):

jal      label

- jal应该被称为 laj , “链接和跳转”:

第 1 步 (link链接): 将下一条指令的地址保存到 \$ra (为什么是下一条指令? 为什么不是当前一条?)

第 2 步 (jump跳转): 跳转到给定的标签

- **跳转并保存返回地址指令: jump and link (jal)**

- 没用jal指令之前:

```
1008 addi $ra,$zero,1016 #$ra=1016  
1012 j sum                #goto sum
```

- **After:**

```
1008 jal sum # $ra=1012,goto sum  
1012 ...
```

为什么需要jal指令? 使得更快: 函数调用非常通用.

# jr 和 jal 指令

- **jr 语法**(jump register):

**jr register**

- **jr 指令跳转的地址在寄存器中.**
- **在函数调用中非常有用:**
  - **jal** 将下一条指令的地址保存到 \$ra
  - **jr \$ra** jumps back to that address
- 不是跳到指定标签处, jr 指令指定的寄存器中保存有跳转地址.
- 对于函数调用非常有用:
  - jal 存返回地址到寄存器 (\$ra)
  - jr \$ra 返回到那个地址

# Branching Far Away

- 如果 branch 跳转目的地址远于它16位能表示的范围怎么办?

□ 汇编器来救援 - 它插入一个无条件跳转到分支目标并反转条件

**beq     \$s0, \$s1, L1**

**becomes**

**bne     \$s0, \$s1, L2**

**j        L1**

**L2:**

# Multiply and Divide

## 乘除指令

- 比 加/减 指令略微复杂一些
  - 乘 multiply: 积的位数成倍增加!
    - 如果 A, B 是 32-bit 长,  $A * B$  多少位?
  - 除 divide: 整数 A 除以 B , 有两个结果!
    - 商 quotient 和 余数 remainder
- 解决方案: 增加了两个特殊用途的寄存器
  - “Hi” and “Lo”

# Multiply 乘法指令

- MULT instruction
  - **mult rs, rt**
  - 含义：将寄存器 \$rs 和 \$rt 的内容相乘，并将（64 位结果）存储在—对特殊寄存器中{hi, lo}
  - $$\text{hi:lo} = \$rs * \$rt$$
  - 高 32 bits 存于 hi 中, 低 32 bits 存于 lo 中
- 要访问结果，使用两个新指令
  - mfhi**: move from hi
    - $$\text{mfhi rd}$$
      - move the 32-bit half result from hi to \$rd
  - mflo**: move from lo
    - $$\text{mflo rd}$$
      - move the 32-bit half result from lo to \$rd

For example:

**mult \$9,\$8**

**mflo \$9**

**mfhi \$8**

# 乘法指令格式

- `mult rs, rt`



- `mflo rd`, 比如 `mflo $9`



- `mfhi rd`, 比如 `mfhi $8`





# Divide 除法指令

- DIV instruction

- **div rs, rt**



- 含义: 将寄存器 \$rs 的内容除以 \$rt, 并将商存储在 lo 中, 余数存储在 hi 中

**lo** = \$rs / \$rt

**hi** = \$rs % \$rt

- 访问结果, use **mfhi** and **mflo**

- 注意: 有两条对应的无符号数乘法和除法指令

- **multu**



- **divu**



## Now 程序举例: 阶乘 Factorial...

### Synopsis (in C):

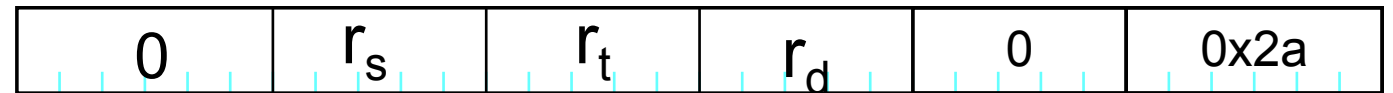
- Input in n, output in ans
- r1, r2 used for temporaries
- assume n is small

```
int n, ans, r1, r2;  
r1 = 1;  
r2 = n;  
while (r2 != 0) {  
    r1 = r1 * r2;  
    r2 = r2 - 1;  
}  
ans = r1;
```

### MIPS code, 汇编程序:

```
n:      .word123  
ans:    .word0  
  
...  
addi$t0, $0, 1# t0 = 1  
lw$t1, n# t1 = n  
loop:   beq$t1, $0, done# while (t1 != 0)  
        mult$t0, $t1# hi:lo = t0 * t1  
        mflo$t0# t0 = t0 * t1  
        addi$t1, $t1, -1# t1 = t1 - 1  
        jloop# Always loop back  
done:   sw$t0, ans# ans = r1
```

# 比较指令: **slt, slti**



- **slt** = set-if-less-than 小于则置一

- **slt rd, rs, rt**

$\$rd = (\$rs < \$rt)$  // “1” if true and “0” if false

- **slti** = set-if-less-than-immediate 小于立即数则置一

- **slti rt, rs, imm**

$\$rt = (\$rs < \text{sign-ext}(\text{imm}))$



- also unsigned flavors 对应两条无符号数比较指令

- **sltu**



**sltiu**



# Logical Instructions 逻辑操作指令

- Boolean operations 布尔操作: 所有32位按位操作
  - AND, OR, NOR, XOR
  - **and, andi**
  - **or, ori**
  - **nor**           // Note: There is no **nori**! Why?
  - **xor, xori**
- 举例:
  - **and \$1, \$2, \$3**  
$$\$1 = \$2 \& \$3$$
  - **xori \$1, \$2, 0xFF12**  
$$\$1 = \$2 \wedge 0x0000FF12$$
  - 更详细内容读课本!

# Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
	subtract	0 & 22	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$
	add immediate	8	addi \$s1, \$s2, 4	$\$s1 = \$s2 + 4$
	shift left logical	0 & 00	sll \$s1, \$s2, 4	$\$s1 = \$s2 \ll 4$
	shift right logical	0 & 02	srl \$s1, \$s2, 4	$\$s1 = \$s2 \gg 4$ (fill with zeros)
	shift right arithmetic	0 & 03	sra \$s1, \$s2, 4	$\$s1 = \$s2 \gg 4$ (fill with sign bit)
	and	0 & 24	and \$s1, \$s2, \$s3	$\$s1 = \$s2 \& \$s3$
	or	0 & 25	or \$s1, \$s2, \$s3	$\$s1 = \$s2 \mid \$s3$
	nor	0 & 27	nor \$s1, \$s2, \$s3	$\$s1 = \text{not } (\$s2 \mid \$s3)$
	and immediate	c	and \$s1, \$s2, ff00	$\$s1 = \$s2 \& 0\text{xff}00$
	or immediate	d	or \$s1, \$s2, ff00	$\$s1 = \$s2 \mid 0\text{xff}00$
	load upper immediate	f	lui \$s1, 0xffff	$\$s1 = 0\text{xffff}0000$

# Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw \$s1, 100(\$s2)	\$s1 = Memory(\$s2+100)
	store word	2b	sw \$s1, 100(\$s2)	Memory(\$s2+100) = \$s1
	load byte	20	lb \$s1, 101(\$s2)	\$s1 = Memory(\$s2+101)
	store byte	28	sb \$s1, 101(\$s2)	Memory(\$s2+101) = \$s1
	load half	21	lh \$s1, 101(\$s2)	\$s1 = Memory(\$s2+102)
	store half	29	sh \$s1, 101(\$s2)	Memory(\$s2+102) = \$s1
Cond. branch (I & R format)	br on equal	4	beq \$s1, \$s2, L	if (\$s1==\$s2) go to L
	br on not equal	5	bne \$s1, \$s2, L	if (\$s1!=\$s2) go to L
	set on less than immediate	a	slti \$s1, \$s2, 100	if (\$s2<100) \$s1=1; else \$s1=0
	set on less than	0 & 2a	slt \$s1, \$s2, \$s3	if (\$s2<\$s3) \$s1=1; else \$s1=0
Uncond. jump	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr \$t1	go to \$t1
	jump and link	3	jal 2500	go to 10000; \$ra=PC+4

# Summary

- 我们课中所讲是 MIPS 指令集的一个子集
  - 有时叫做“miniMIPS”
  - 所有指令都是 32-bit
  - 3 种基本指令格式
    - R-type - Mostly 2 source and 1 destination register
    - I-type - 1-source, a small (16-bit) constant, and a destination register
    - J-type - A large (26-bit) constant used for jumps
  - Load/Store architecture
  - 31 个通用寄存器, 0 寄存器, 硬件设置为 0, 31 个寄存器中, 有几个特殊用途寄存器.
- ISA 设计需要权衡折中, 通常与以下有关
  - 历史、艺术、工程
  - 基准测试结果