



天津大学

数字逻辑与数字系统

# 5

# 指令集体系结构

Instruction Set Architecture

## ■ 计算机体系结构

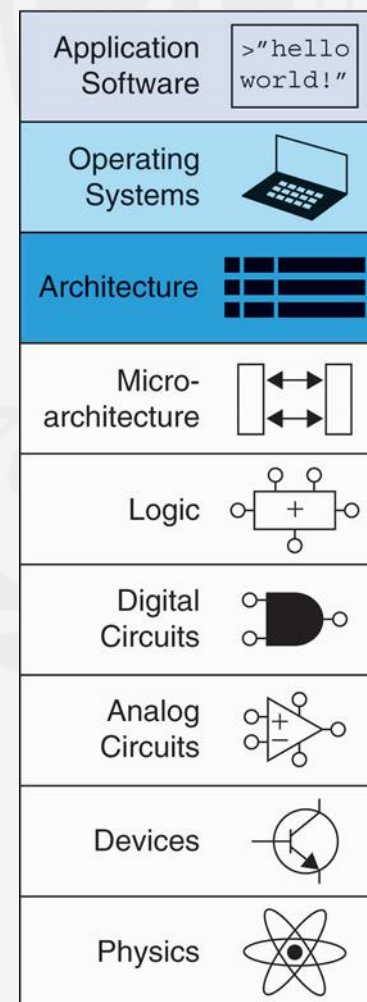
程序员所能见到的计算机，由助记符（汇编语言）和操作数（寄存器和存储器）来定义。

如：x86、MIPS、SPARC、PowerPC等

## ■ 微体系结构

体系结构的硬件实现方式

相同体系结构可能具有不同的微体系结构







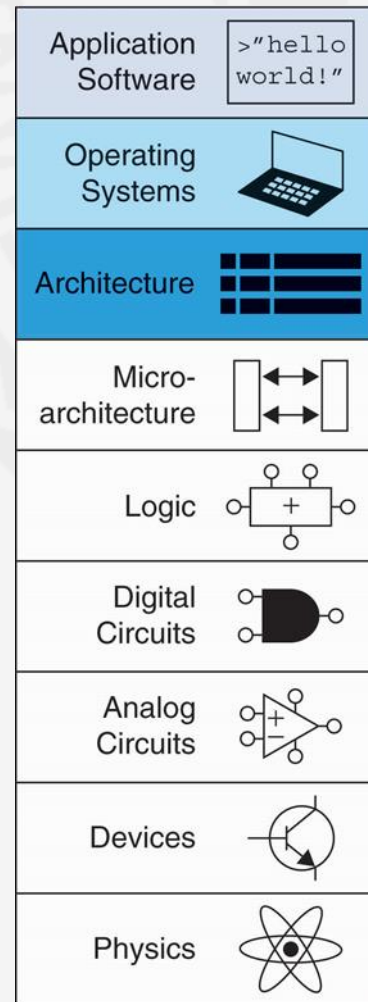
# 本章内容

Topic

## □ 冯·诺依曼体系结构

## □ 汇编语言

## □ 机器语言





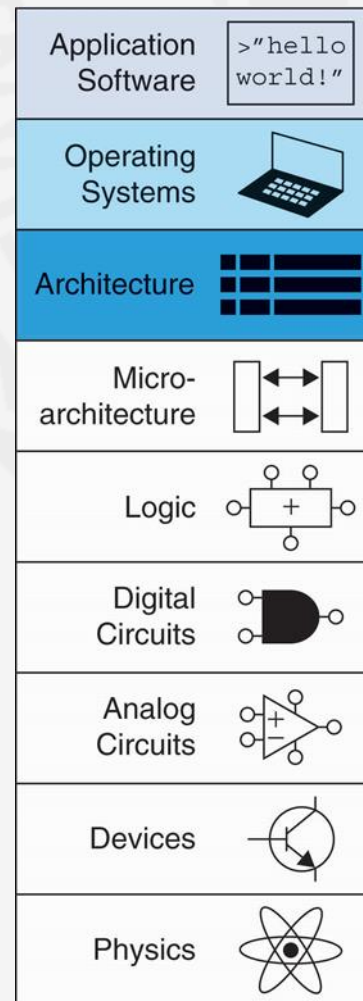
# 本章内容

Topic

## 冯·诺依曼体系结构

## 汇编语言

## 机器语言





# 冯·诺依曼体系结构

Von Neumann Computer

电子计算机的问世，奠基人是英国科学家图灵（Alan Turing）和美籍匈牙利科学家冯·诺依曼（John Von Neumann）。图灵的贡献是建立了图灵机的理论模型，奠定了人工智能的基础。而冯·诺依曼则是首先提出了计算机体系结构的设想。





# 冯·诺依曼体系结构

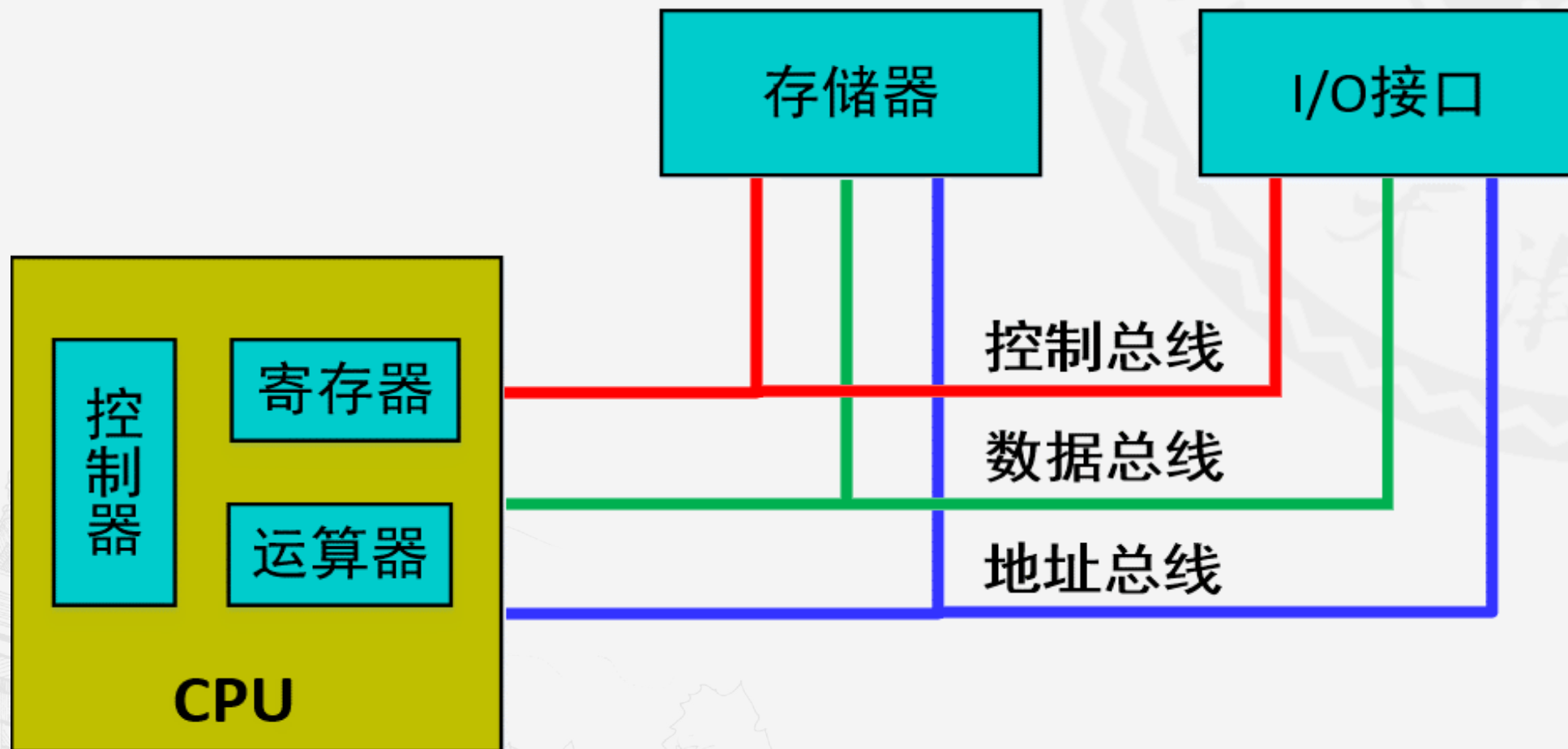
Von Neumann Computer

- ① 采用二进制形式表示数据和指令。指令由操作码和地址码组成。
- ② 将程序和数据存放在存储器中，计算机在工作时从存储器取出指令加以执行，自动完成计算任务。这就是“存储程序”和“程序控制”（简称存储程序控制）的概念。
- ③ 指令的执行是顺序的，一般情况下，程序按照指令在存储器中存放的顺序执行，程序分支由转移指令实现。
- ④ 计算机由**存储器、运算器、控制器、输入设备和输出设备**五大基本部件组成，并规定了这五个部分的基本功能。



# 冯·诺依曼体系结构

Von Neumann Computer







## 冯·诺依曼结构五大部件

- **运算器（ALU）**：计算机中直接完成各种运算（算术运算、逻辑运算）的部件
- **控制器（Controller）**：发出控制命令、控制机器各部件自动、协调工作的部件
- **存储器（Memory）**：用来保存和记录原始数据、程序和运算结果的部件
- **输入设备（Input device）**：用来往计算机中输送程序、数据的装置
- **输出设备（Output Device）**：将计算结果输送出来的装置



## 一些基本概念

- **中央处理器（ Central Processing Unit, CPU ）**：运算器和控制器
- **总线（ Bus ）**：传送信息的公共通道  
  
系统总线包括：**地址总线**、**数据总线**和**控制总线**
- **寄存器文件（ Register Files ）**：CPU内部用来存放数据的一些小型存储区域  
  
用来暂时存放参与运算的数据和运算结果
- **指令（ Instruction ）**：给机器下达的完成一项基本操作的指令
- **程序（ Program ）**：完成一项任务所需的并按照一定顺序排列起来的指令集合



## 指令集

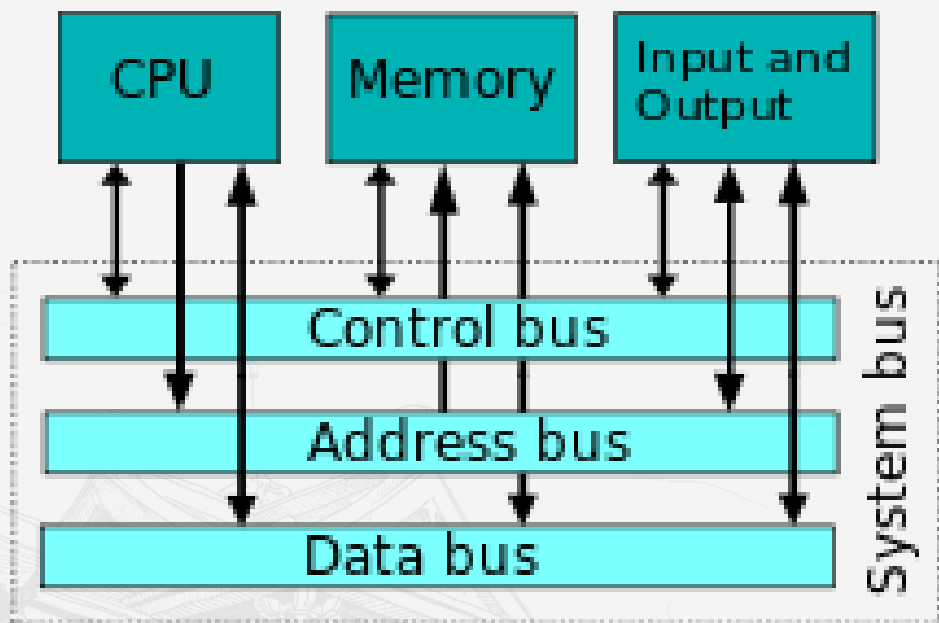
- CPU厂商给属于同一系列的CPU产品定的一个规范，是区分不同类型CPU的重要标识
- x86、x86-64、ARM、MIPS、PowerPC、SPARC
- 不同处理器架构通常具有不同的指令集 (instruction sets)



# 冯·诺依曼体系结构

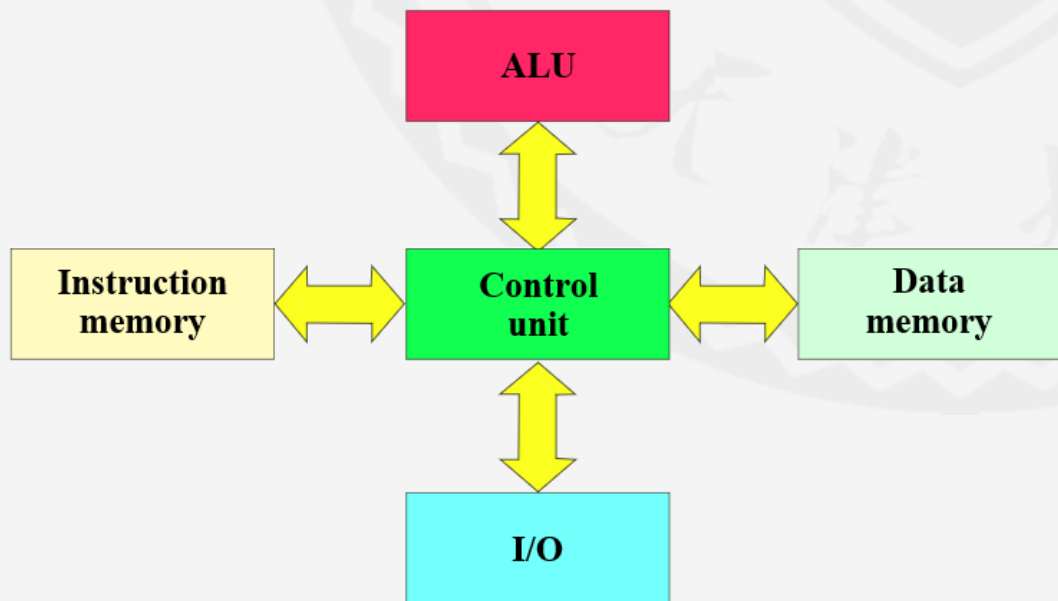
Von Neumann Computer

## 冯·诺依曼结构



程序指令和数据统一存储

## 哈佛结构



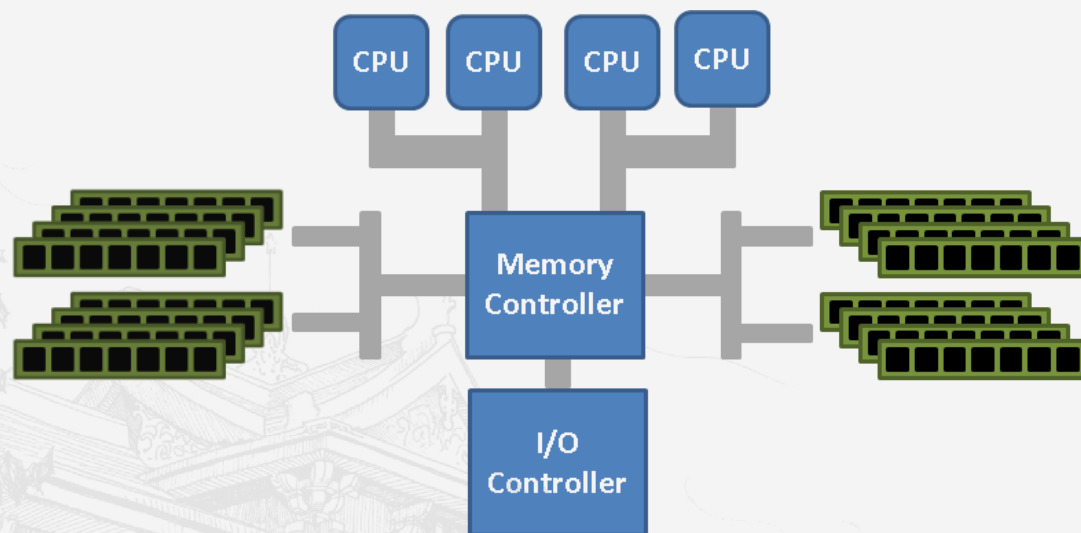
程序指令存储和数据存储分开



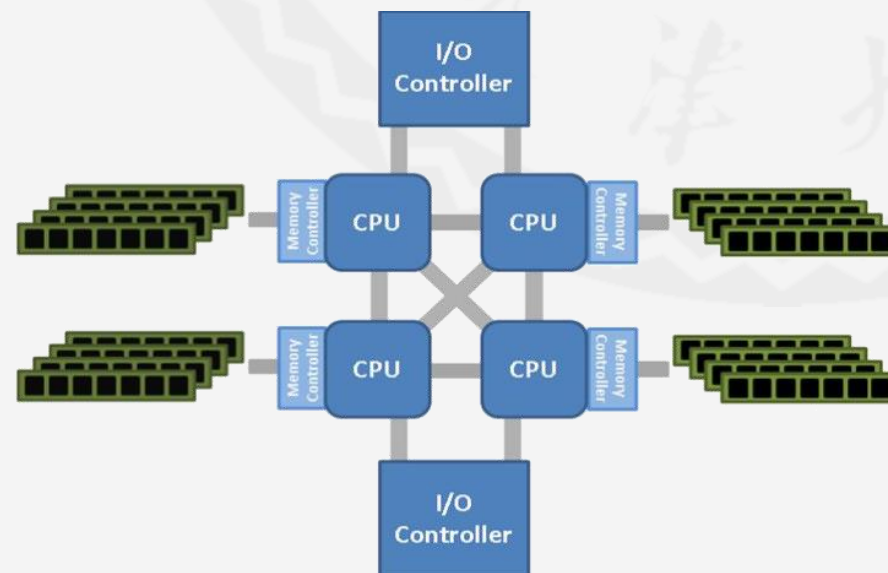


## 并行体系结构

### Uniform Memory Access (UMA)



### Non-Uniform Memory Access (NUMA)





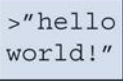

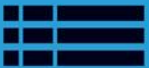
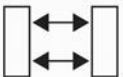
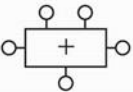
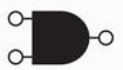
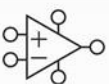

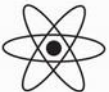
# 本章内容

Topic

□ 冯·诺依曼体系结构

□ 汇编语言

□ 机器语言

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

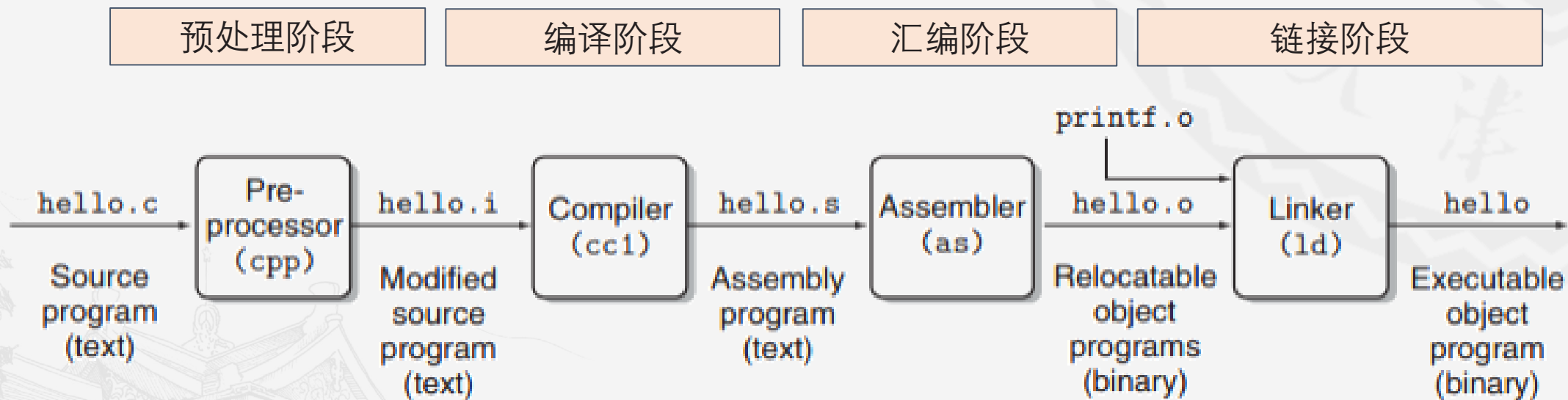


## 什么是汇编语言？

- 机器语言：由二进制0和1组成，是可以被CPU直接执行的指令
- 汇编语言：机器语言的助记符，是一种低级语言
- 通常，不同指令集的处理器所支持的汇编语言不相同
- 汇编语言不具备跨平台的能力



## 编译系统







## MIPS架构

- 由John Hennessy 和同事于1981在斯坦福大学提出
- 应用于许多商业系统：Silicon Graphics, Nintendo和Cisco等
- 2002年，中国科学院计算所开始研发**龙芯**处理器，采用MIPS架构
- 2009年，龙芯得到MIPS公司的正式授权





## MIPS体系架构设计准则

- ① 简单设计有助于规整化
- ② 加速常用功能
- ③ 越小越快
- ④ 好的设计需要好的折中



## 指令：加法

### C 语言

```
a = b + c;
```

### MIPS 汇编

```
add a, b, c
```

- **add**: 助记符，指明当前指令的具体功能
- **b, c**: 源操作数，待运算的数据或数据的存放位置
- **a**: 目标操作数，运算结果存放的位置



## 指令：减法

### C 语言

```
a = b - c;
```

### MIPS 汇编

```
sub a, b, c
```

- **sub**: 助记符，指明当前指令的具体功能
- **b, c**: 源操作数
- **a**: 目标操作数
- 仅仅是助记符有变化





## 设计准则1：简单设计有助于规整化

- 指令格式前后一致
- 操作数格式一致
- 易于在硬件中编码和处理
- 更复杂的高级语言代码可以转化为多条指令

### C 语言

```
a = b + c - d;
```

### MIPS 汇编

```
add t, b, c    # t = b + c  
sub a, t, d    # a = t - d
```



## 设计准则2：加速常用功能

- MIPS 仅包含简单和常用的指令
- 译码和执行指令的硬件实现起来更加简单、运算速度更快
- 更复杂但不常见的操作由多条简单指令组合完成
- MIPS是**精简指令集（RISC）**计算机，指令集规模小，仅包含相对较少的简单指令
- **复杂指令集（CISC）**计算机，指令集规模大，硬件实现复杂，指令执行效率受到影响
- x86是复杂指令集



## 操作数

- 汇编指令中的操作数分为三种类型
  - 寄存器 (Register)
  - 存储器 (Memory)
  - 立即数 (常数) (Immediate)



## 操作数：寄存器

- MIPS有32个32位寄存器
- 寄存器的访问速度比存储器快得多
- MIPS被称为32位体系结构
  - 指令的操作数为32位





## 设计准则3：越小越快

- MIPS 仅包含非常少量的寄存器

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	Function return values
\$a0-\$a3	4-7	Function arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	Function return address



## 操作数：寄存器（续）

- 在MIPS汇编中，对于寄存器
  - 名称前都有 \$ 符号
  - 例如：\$0
- 在设计时，每个寄存器都有其特殊的用途，例如：
  - \$0 用户存放常量0
  - 保存寄存器 \$s0 - \$s7，用于存放变量
  - 临时寄存器 \$t0 - \$t9，用于存放大型计算的中间值



## 例：操作数为寄存器的指令

### C 语言

```
a = b - c;
```

### MIPS 汇编

```
# $s0 = a, $s1 = b, $s2 = c
```

```
sub $s0, $s1, $s2
```



## 操作数：存储器

- 如果程序中使用的数据过多，32个寄存器并不能完全满足数据存储的要求
- 更多的数据将会存储在存储器中
- 存储器容量更大，但是速度慢（相对于寄存器）
- 常用的变量一般保存在寄存器中



## 字寻址存储器

- 数据字为32位，每个数据字都有一个唯一的地址
- 实际上，MIPS是针对字节进行寻址的（稍后讨论）

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	字 3
00000002	0 1 E E 2 8 4 2	字 2
00000001	F 2 F 1 A C 0 7	字 1
00000000	A B C D E F 7 8	字 0





## 读字寻址存储器

- 存储器读操作被称为“加载”

- 助记符: **lw** (load word)

- 格式:

**lw** **\$s0**, **5(\$t1)**

- 地址计算: 基地址 ( $\$t1$ ) 加上偏移量 (5), 地址为 ( $\$t1+5$ )

- 操作结果: 寄存器 ( $\$s0$ ) 存储了存储器中地址为 ( $\$t1+5$ ) 位置的数据

- 任意寄存器都可以用于存放内存基地址



## 例：存储器读数据

■ 从 1 地址读入一个字，并存入 `$s3` 寄存器

■ 地址计算：  $(\$0+1) = 1$

■ 结果：寄存器 `$3` 的值为 `0xF2F1AC07`

■ 汇编指令

```
lw    $s3, 1($0)
```

字地址	数据	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	字 3
00000002	0 1 E E 2 8 4 2	字 2
00000001	F 2 F 1 A C 0 7	字 1
00000000	A B C D E F 7 8	字 0



## 写字寻址存储器

- 存储器写操作被称为“存储”
- 助记符: **sw** (**store word**)
- 例: 将寄存器 **\$t4** 中的数据存储至存储器中地址为 **7** 的位置
  - 地址计算: 基地址 (**\$0**), 偏移量 **0x7**, 地址为  $(\$0 + 0x7) = 7$
- 汇编指令

```
sw $t4, 0x7($0)
```



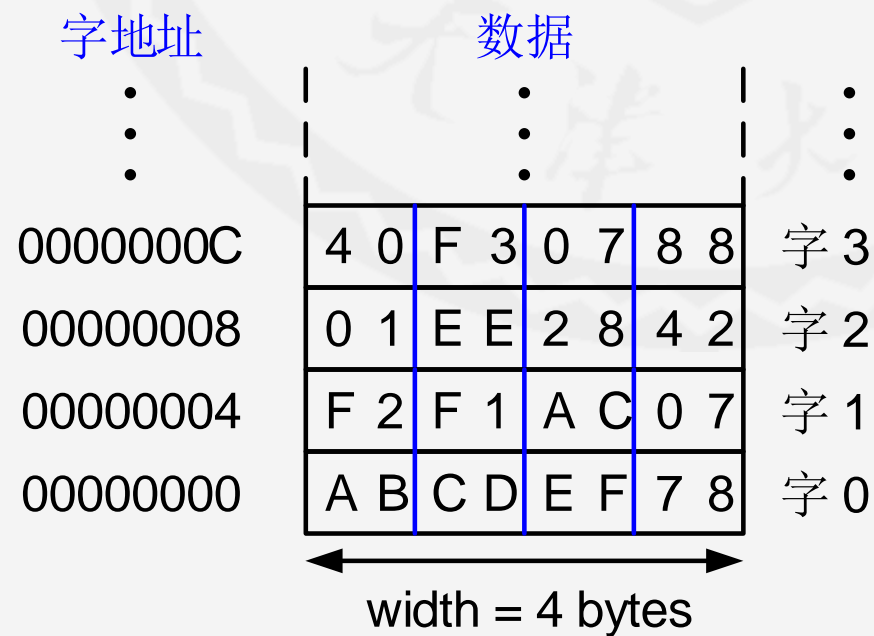
## 字节寻址存储器

■ 每一个字节都有一个唯一的地址

■ 加载和存储单个字节的助记符为：

**lb** (load byte) , **sb** (store byte)

■ 32位字等于4个字节，相邻两个数据字地址相差4





## 字节寻址存储器 (cont.)

- 在字长为32位的字节寻址存储器中，每个数据字的地址都是4的倍数
  - 字2的实际存储器地址为  $2 \times 4 = 8$
  - 字10的实际存储器地址为  $10 \times 4 = 40$  (0x28)
- 实际上，MIPS是基于字节寻址的，而不是基于字寻址的

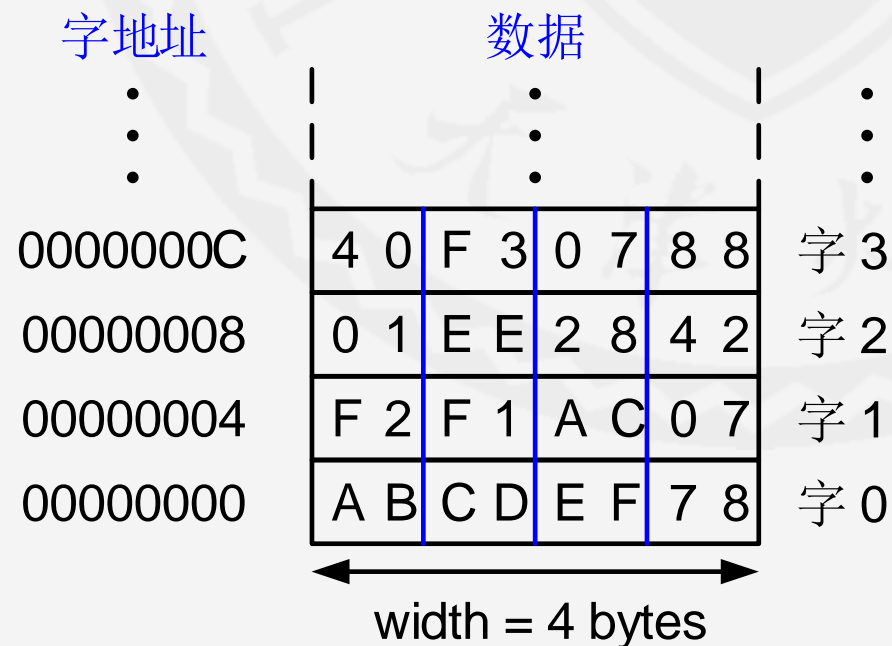




## 读字节寻址存储器

- 例：加载地址 4 的数据字至寄存器 `$s3`
- 在加载后寄存器 `$s3` 存储了数据 `0xF2F1AC07`
- MIPS汇编

```
lw $s3, 4($0)
```



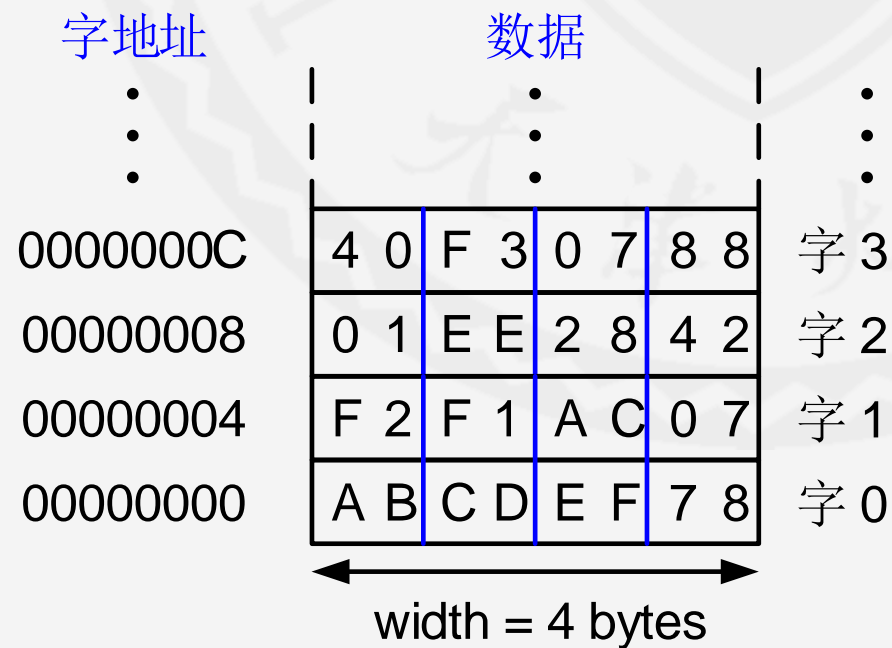


## 写字节寻址存储器

■ 例：存储寄存器 `$t7` 的数据至存储器地址 `0x2c`

■ MIPS汇编

```
sb $t7, 44($0)
```

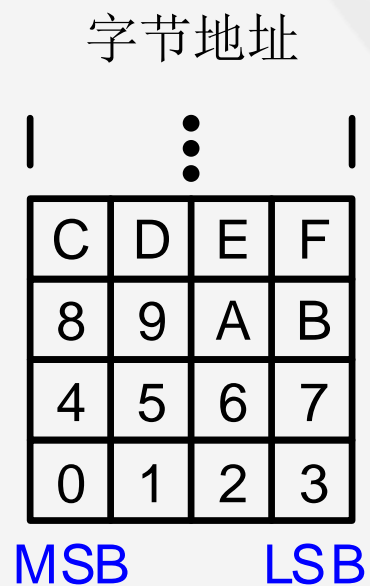




## 字节序

- 多个字节如何存储在一个数据字中？
- 小端（Little-endian）：低地址存放低位数据，高地址存放高位数据
- 大端（Big-endian）：高地址存放低位数据，低地址存放高位数据

大端



小端





## 字节序(cont.)

■ 假设: `$t0` 的初值为 `0x23456789`

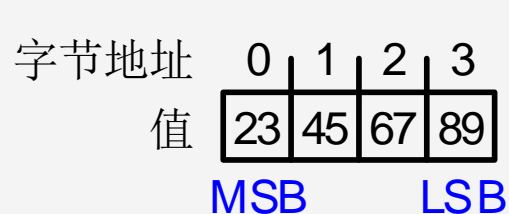
```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

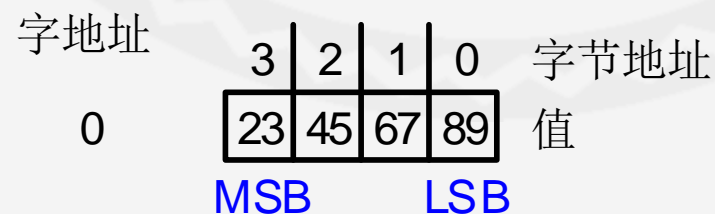
■ 大端: `$s0 = 0x00000045`

■ 小端: `$s0 = 0x00000067`

大端



小端





## 设计准则4：好的设计需要好的折中

- 支持多种指令格式，灵活性更强
  - 3操作数指令： `add`, `sub`
  - 2操作数指令： `lw`, `sw`
  - 1操作数指令（操作数为26位立即数，后面课程中会讲到）
- 指令格式的种类很少
  - 遵循准则1和准则3：简单规整、越小越快





## 操作数：立即数

- 指令 `lw` 和 `sw` 中使用了立即数
- 立即数可以被指令立即访问，不需要通过访问寄存器或存储器得到
- 立即数采用16位补码表示，范围  $[-32768, 32767]$
- 立即数加法指令： `addi`

### C 语言

```
a = a + 4  
b = a - 12
```

### MIPS 汇编

```
# $s0 = a, %s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

- `subi` 是否必要?



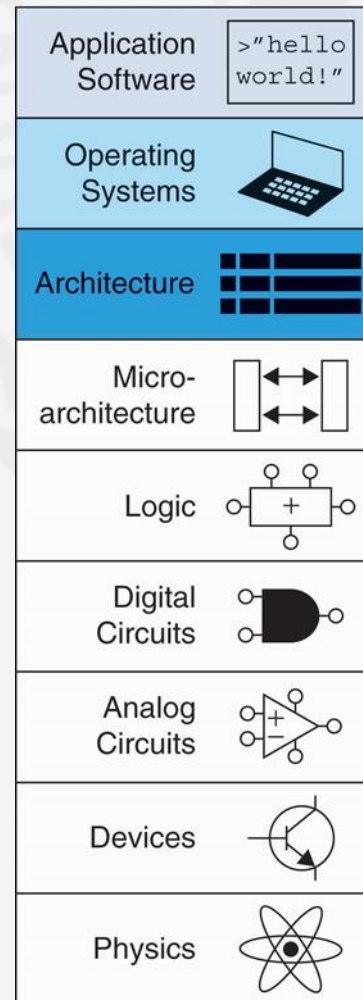
# 本章内容

Topic

□ 冯·诺依曼体系结构

□ 汇编语言

□ 机器语言





## 基本概念

- 指令的二进制表示形式
- 在精简指令集计算机（RISC）中，指令为定长编码
- MIPS指令使用32位编码
  - R（寄存器）类型指令
  - I（立即数）类型指令
  - J（跳转）类型指令



## R（寄存器）类型指令

### ■ 3操作数:

■ rs、rt: 源寄存器

■ rd: 目标寄存器

### ■ 每个寄存器都有唯一的编号

■ op: 操作码, R类型指令操作码为0

■ funct: 功能码, 特定的R类型操作由funct决定

■ shamt: 仅用于移位操作, 存储移位的位数

## R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



## 举例：R类型指令

### 汇编指令

add \$s0, \$s1, \$s2

sub \$t0, \$t3, \$t5

### 字段值

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

### 机器代码

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

**注意：**操作数的顺序

add rd, rs, rt





## I（立即数）类型指令

### ■ 3个操作数

■ rs, rt: 寄存器操作数

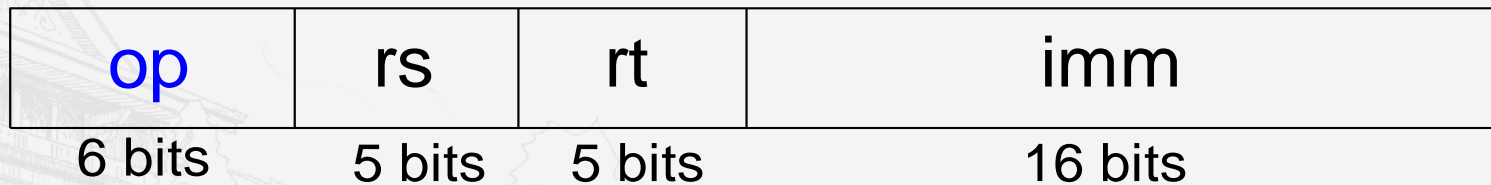
■ imm: 16位补码表示的立即数

### ■ op: 操作码

■ 每个I指令都具有一个对应的op

■ 指令的操作由op字段决定

## I-Type





## 举例：I（立即数）类型指令

### 汇编指令

```
addi $s0, $s1, 5
```

```
addi $t0, $s3, -12
```

```
lw    $t2, 32($0)
```

```
sw    $s1, 4($t1)
```

### 操作数顺序

```
addi rt, rs, imm
```

```
lw    rt, imm(rs)
```

```
sw    rt, imm(rs)
```

### 字段值

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4

6 bits

5 bits

5 bits

16 bits

### 机器代码

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

6 bits

5 bits

5 bits

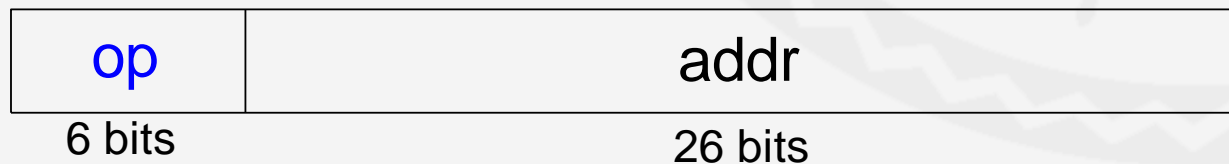
16 bits



## J（跳转）类型指令

- op: 操作码
- 26位的跳转地址
- 跳转指令助记符 j

### J-Type





## 解释机器代码

- 机器代码以操作码开头，根据机器代码后面的部分如何进行解释
- 如果操作码为0
  - 为R类型指令
  - 功能码解释具体的操作类型

机器代码

字段值

汇编指令

(0x2237FFF1)

op	rs	rt	imm
001000	10001	10111	1111 1111 1111 0001
2	2	3	7 F F F 1

op	rs	rt	imm
8	17	23	-15

addi \$s7, \$s1, -15

(0x02F34022)

op	rs	rt	rd	shamt	funct
000000	10111	10011	01000	000000	100010
0	2	F	3	4	0 2 2

op	rs	rt	rd	shamt	funct
0	23	19	8	0	34

sub \$t0, \$s7, \$s3

## 程序的存储

### 汇编指令

lw \$t2, 32(\$0)

add \$s0, \$s1, \$s2

addi \$s0, \$s3, -12

sub \$t0, \$t3, \$t5

### 机器代码

0x8C0A0020

0x02328020

0x2268FFF4

0x016D4022

### 存储程序

地址	指令
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0 ← PC
⋮	⋮

主存

### 程序计数器 (Program Counter, PC)

存储当前正在执行指令在内存中所在的地址



## 常用MIPS汇编指令

### ■ 逻辑运算

■ R类型指令: `and`, `or`, `xor`, `nor`

■ I类型指令: `andi`, `ori`, `xori`, 立即

数为16位补码 (零扩展)

■ 不提供 `not` 指令,  $\text{not } A = A \text{ nor } 0$

■ 不提供 `nori` 指令, 用其他的指令代替

### ■ 移位运算 (R类型指令)

■ `sll`, `srl`, `sra`

例: `sll $t0, $t1, 5`

■ `sllv`, `srlv`, `srav`

例: `sllv $t0, $t1, $t2`





## 常用MIPS汇编指令 (cont.)

### ■ 条件跳转

■ beq 等于时跳转

例: `beq rs, rt, label`

■ bne 不等于时跳转

例: `bne rs, rt, label`

### ■ 无条件跳转

■ j 跳转

例: `j label`

■ 过程调用: `jal, jr`

### ■ j指令跳转目标地址(JTA)的计算

$$JTA = \{ (PC+4)[31:28], \text{addr}, 2'b0 \}$$



## 常用MIPS汇编指令 (cont.)

### ■ 有符号比较运算

■ R类型指令: `slt`

例: `slt rd, rs, rt`

■ I类型指令: `slti` (符号位扩展)

例: `slti rt, rs, imm`

### ■ 无符号比较运算

■ R类型指令: `sltu`

■ I类型指令: `sltiu` (符号位扩展)



## 常用MIPS汇编指令 (cont.)

### ■ 有符号数数据装载 (符号位扩展)

■ 装入字 (32bit) : `lw`

例: `lw rt, imm(rs)`

■ 装入半字 (16bits) : `lh`

例: `lh rt, imm(rs)`

■ 装入字节 (8bits)

例: `lb rt, imm(rs)`

### ■ 无符号数数据装载 (零扩展)

■ 装入半字 (16bits) : `lhu`

例: `lhu rt, imm(rs)`

■ 装入字节 (8bits)

例: `lbu rt, imm(rs)`



## 常用MIPS汇编指令 (cont.)

### ■ 存储数据

■ 存储字 (32bit) : sw

例: `sw rt, imm(rs)`

■ 存储半字 (16bits) : sh

例: `sh rt, imm(rs)`

■ 存储字节 (8bits)

例: `sb rt, imm(rs)`