# 计算机组成原理笔记

陈鸿峥

2019.01 *

## 目录



*Build 20190106

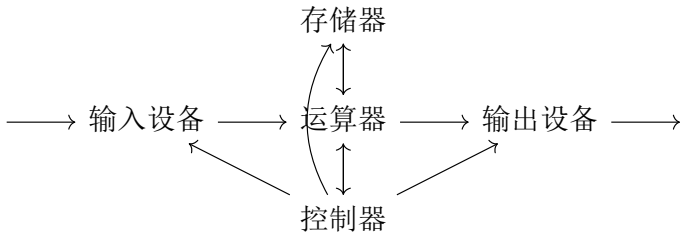
本课程使用的教材为David A.Patterson (UCB), John L.Hennessy (Stanford), 《计算机组成与设计（硬件软件接口）》.

# 1 计算机系统概述

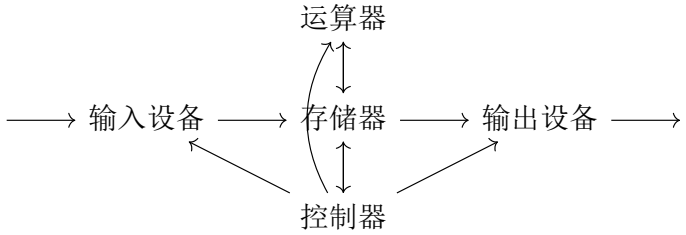
## 1.1 计算模型

- 图灵机(1936)

- 冯诺依曼体系结构(1945)[1] — 存储程序原理（**运算器**为中心）
  计算机采用**二进制**表示机器指令和数据，按照程序指令**顺序**执行




而现在由于计算不是瓶颈，存储访问成为了瓶颈，故现代微机以**存储器**为中心




[运算器、控制器]（CPU）、存储器为计算机的核心，合称主机；外围设备，简称外设，指除主机外的其他设备，包括IO设备、外存等

计算机中的信息仍用二进制表示的原因：由物理器件性能决定

- 二进制只有两种状态，容易找到具有2个稳定状态并且状态转换容易控制的物理器件（数字电路）
- 二进制编码运算规则简单
- 二进制的0、1与二值逻辑一致，容易实现逻辑运算

## 1.2 计算机的发展历程

按发展历程可分为：电子管、晶体管、集成电路、（超）大规模集成电路四代计算机

重大历史事件如下

| | | | |
|---|---|---|---|
| 1904 | 弗莱明(Fleming) | 二极管 | |
| 1907 | 德福雷斯特(De Forest) | 三极管 | |
| 1938 | 香农(Shannon) | 布尔代数与二值电子器件（继电器） | 奠定数字电路基石 |
| 1946 | | 第一台通用计算机ENIAC | 十进制 |
| 1947 | 布莱顿(Brattain)<br>巴丁(Bardeen) | 点接触晶体管 | |
| 1949 | 肖克利(Shockley) | 结型晶体管(1949) | 1956诺贝尔奖 |
| 1950 | | 二进制和存储程序EDVAC | 实现冯诺依曼设想（组合进步） |
| 1958 | Jack Kilby | 集成电路 | 2000诺贝尔奖 |
| 1965 | Moore | 摩尔定律 | 在价格不变的情况下，每18个月芯片上晶体管数目翻倍，性能也提升一倍 |
| 1971 | Intel | 第一款微处理器4004 | 10$\mu$m |

[1]非冯诺依曼体系结构：并行计算、量子计算、生物计算

### 1.2.1 单处理器(1971-2002)

性能提升主要手段

- 提升工作主频：KHz增长至GHz（生产工艺进步，流水线级数增加）
- 指令级并行(ILP)

**命题 1** (安迪-比尔定律). *Andy gives, Bill takes away. 安迪是原Intel CEO，比尔是原微软CEO，硬件厂商靠软件开发商用光自己提供的硬件资源得以生存*

但遇到频率墙和功耗墙

$$\text{功耗(power)} \propto 1/2 \times \text{CMOS电容} \times \text{电压}^2 \times \text{转换(01)频率}$$

2004年，Intel放弃4GHz Pentium4芯片开发，因无法解决散热问题，通过加快主频提升处理器性能的路走到尽头

### 1.2.2 多核处理器(2005-)

采用多核处理器不过是将硬件的问题丢到软件[2]

**定理 1** (阿姆达尔(Amdahl)定律).

$$\text{改进后的执行时间} = \text{受改进影响部分的执行时间}/\text{改进提高的倍数} + \text{不受影响的执行时间}$$

$$S_A = \frac{1}{s + (1-s)/N},$$

对计算机系统的某个部分采用并行优化措施后所获得的计算机性能的提高是有上限的，上限由串行部分所占的比例决定

**定理 2** (古斯塔夫森(Gustafson)定律).

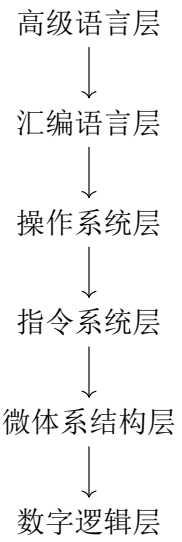
$$S_G = (s' + p' \times N)/(s' + p') = N + (1-N) \times s',$$

其中，$s'$和$p'$为程序串行部分与可并行化部分在并行系统上执行的时间占总时间的比例，$N$为处理器数量，简便起见设总时间$s' + p' = 1$

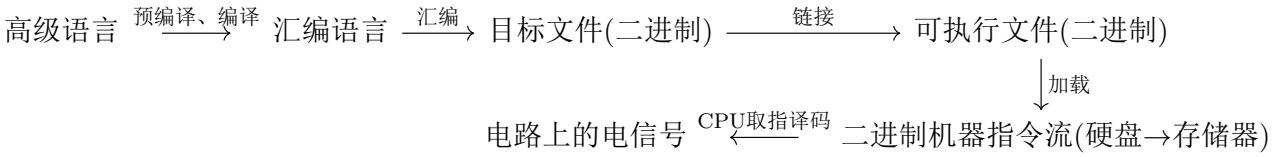
打破Amdahl定律**问题规模不变**的假设，任何足够大的任务都可以被有效地并行化，只要问题规模可扩展，并行所带来的加速比就可以扩展

[2] “向多核的转变并不是因为我们在软件或体系结构技术上取得了中大突破而带来的。相反，这种转变是当单处理器体系结构发展遇到了难以克服的巨大障碍时，我们被迫作出的一种选择。” —Kurt Keutzer (UCB), *The Landscape of Parallel Computing Research: A View from Berkeley*

## 1.3 计算机系统的层次结构




程序编译运行过程：




计算机内部工作过程：逐条执行加载到内存中的二进制机器指令流的过程

指令执行分为两个阶段，周期性重复性进行：

- 取指阶段：CPU从内存中读取指令，程序计数器(PC)保存要被要被取出的**下一条**指令的地址，除非遇跳转指令，否则都加一个增量[3]
- 执行阶段：对取出的指令译码后执行

软件系统可分为系统软件和应用软件

## 1.4 计算机结构的八个想法

1. 摩尔(Moore)定律：集成电路资源每18 – 24个月翻倍
2. 抽象(abstraction)：简化设计
3. 加速常用操作(Make common case fast)：见定理1
4. 并行(parallelism)
5. 流水线(pipelining)
6. 预测(prediction)
7. 内存等级制(hierarchy)
8. 冗余实现可靠性(redundancy)：检测故障及解决

[3]程序计数器（Program Counter）是一个实际存在的寄存器吗？ - Belleve的回答 - 知乎 https://www.zhihu.com/question/22609253/answer/21965180 PC每次增加**一条指令的长度/寻址粒度**，在MIPS中一条指令长4字节，寻址粒度1字节，故每次PC加4；而x86体系指令长度不定，每次增加量会变化

## 1.5 基本指标

表示计算机通信带宽时

| KB(yte) | MB | GB | TB | PB | EB | ZB |
|---|---|---|---|---|---|---|
| $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{15}$ | $10^{18}$ | $10^{21}$ |

表示计算机存储二进制时

| KiB(yte) | MiB | GiB | TiB |
|---|---|---|---|
| $2^{10}$ | $2^{20}$ | $2^{30}$ | $2^{40}$ |

- 位(bit/b)：计算机处理、存储、传输信息的最小单位
- 字节(Byte/B) 1 Byte = 8 bit：现代计算机主存按字节编制，字节是最小可寻址单位
- 字(Word)：表示被处理信息的单位，用来度量数据类型的宽度[4]

一台32位的电脑，一个字等于4个字节，字长为32位；若字长为16位，则一个字等于2字节.
4字节相当于8位16进制编码

## 1.6 性能评价

CPU主频：对同一型号计算机，主频越高，完成指令一个执行步骤时间越短

$$\text{计算机的性能(Performance)} = 1/\text{执行时间(Execution time)}$$

按照单位（量纲）进行换算即可

$$\begin{aligned}\text{CPU执行时间(s)} &= \text{执行程序所需CPU时钟周期(cyc)} \times \text{时钟周期s/cyc)} \\ &= \text{指令数目(ins)} \times \text{CPI(cyc/ins)} \times \text{时钟周期(s/cyc)}\end{aligned}$$

程序性能对执行事件的影响：

| | 指令数 | CPI | 时钟周期 |
|---|---|---|---|
| 算法、编程语言、编译器 | × | × | |
| 指令集 | × | × | × |
| 计算机组成 | | × | × |
| 实现技术 | | | × |

体系结构=指令集体系结构（功能定义与设计）+计算机组成（考虑用什么材料）
举例来说：

- 指令集(ISA)考虑：是否提供乘法指令
- 组成(Organization)考虑：如何实现乘法指令（专门乘法器还是加法器+移位器）
- 实现技术(Technology)考虑：如何布线、用什么材料和工艺

[4]字长是指CPU中**数据通路的宽度**，等于CPU内部总线的宽度或运算器的位数或通用寄存器的宽度；字和字长的宽度可以一样，也可以不同，通常是字节的整数倍

# 2 指令系统

IS处在软件和硬件的交界面上，能同时被硬件设计者和系统程序员看到

从硬件设计者角度看

- IS为CPU提供功能需求
- IS设计目标：易于硬件逻辑设计

从系统程序员角度看

- 通过IS使用硬件资源
- IS设计目标：易于编写编译器

IS设计的好坏决定了计算机的性能和成本

## 2.1 概述

| 复杂指令集计算机 | 精简指令集计算机 |
| --- | --- |
| CISC, Complex Instruction Set Computer | RISC, Reduce Instruction Set Computer |
| 出现较早，大而全 | 小而精 |
| 指令周期长，专用寄存器，微程序控制，难编译优化生成高效目标代码，效率低(二八定律，简单指令使用频率高) | 指令周期短，**大量通用寄存器**，组合逻辑电路控制，优化编译系统，简单寻址方式 |
| 变长指令字 | 定长指令字 |
| 借鉴思想 | 现在多用 |
| x86 | ARM, MIPS, SPARC |

## 2.2 指令格式

指令一般由**操作码**和**地址码**（包括操作数和寻址方式）决定

### 2.2.1 操作码设计

- 关注程序代码长度时：变长指令字、变长操作码
- 关注性能时：定长指令字、定长操作码

扩展操作码：使用频率高的指令用短的操作码，频率低的指令用长的操作码

- 零地址指令：空操作、停机、堆栈
- 一地址指令：取反、取负、累加器
- 二地址指令（最常用）：分别存放双目运算中两个源操作数地址，并将其中一个地址作为结果地址
- 三地址指令（RISC）：双目运算中两个源操作数地址和一个结果地址

**例 1.** *36位长的指令系统，设计一个扩展操作码，使之能表示以下指令*

- *7条两个14位地址和一个5位地址的指令*

- *600*条一个*14*位地址和一个*5*位地址的指令
- *100*条无地址指令

**分析.** 各指令的范围如下

- 三地址指令：$36 = 3(op) + 14(ad1) + 14(ad2) + 5(ad3)$
  $000 \sim 110\,((0)_{10} \sim (6)_{10})$共*7*条，111为扩展码
- 二地址指令：$36 = 3 + 14(op) + 14(ad1) + 5(ad2)$
  $00,0000,0000,0000 \sim 00,0010,0101,0111\,((0)_{10} \sim (599)_{10})$共*600*条，$111,00,0010,0101,1000$为拓展码
- 零地址指令：$36 = 3 + 14(op) + 12 + 7$
  最后*7*位$000,0000 \sim 110,0011\,((0)_{10} \sim (99)_{10})$共*100*条

### 2.2.2 寻址方式

通常特指操作数寻址（对应的是指令寻址，PC增值和跳转），目的是扩大访存范围，提高访问数据的灵活性和有效性

| | | |
|---|---|---|
| 立即数寻址 | 直接给出操作数本身，无需访存快速，操作数大小受地址字段长度限制，大量使用 | MOV AX, 1000H |
| 存储器直接寻址 | 操作数在存储器中，直接给出操作数在存储器中的地址，寻址空间受指令地址字段长度限制，较少使用 | MOV AX, [1000H] |
| 存储器间接寻址 | 存储器中的内容是操作数的地址，需二次寻址 | |
| 寄存器直接寻址 | 直接给出寄存器编号，无需访存速度快，地址范围有限，可用通用寄存器较少，使用最多，提高性能常用手段 | MOV AX, BX |
| 寄存器间接寻址 | 寄存器中的内容是操作数的地址，二次寻址 | MOV AX, [BX] |
| 相对寻址（偏移） | 相对**当前指令(PC)**位移量为A的单元，跳转指令 | EA=(PC)+A |
| 基址寻址（偏移） | 相对基址(B)位移量为A的单元，**OS页面（重定位）**，面向系统，程序逻辑空间与存储器物理空间的无关性 | EA=(B)+A |
| 变址寻址（偏移） | 相对形式地址A（数组基址）位移量为(I)的单元，X为数组元素大小，面向用户 | EA=(I)+A, I=(I)±X |
| 堆栈寻址 | 从寄存器到堆栈或反过来，指令短 | EA=栈顶(SP) |
| 复合寻址 | 间接寻址+相对/变址寻址 | 间接相对EA=(PC)+A, 相对间接EA=((PC)+A) |

*

(X)代表X地址/寄存器内的内容，如((X))代表寄存器间接寻址

寻址方式的确定

- 操作码中给定寻址方式：MIPS
- 专门寻址方式：x86（0-1字节）
- 指令总数取决于操作码位数
- 寄存器决定了编码位数
- 地址寄存器(MAR)的位数取决于主存地址空间大小
- 数据寄存器(MDR)取决于机器字长

注意看是按**字**编址还是按**字节**编址
但从80年代开始，几乎所有机器都采用字节编址(byte addressing)

## 2.3 数据表示

### 2.3.1 进制(system)

$$\text{进位计数制} = \text{基数} + \text{位权}$$

二进制(binary)、八进制(octonary)、十进制(decimal)、十六进制(hexadecimal)

- 十进制转二进制：整数部分除以2取余，小数部分乘2取整
- 二进制转八进制：从整数最低位开始，三位三位统计
- 二进制转十六进制：从整数最低位开始，四位四位统计

### 2.3.2 符号数

1. 二进制（真值→机器数）
    - 符号数值(sign-magnitude)形式（原码）：首位0为正数，1为负数，将符号位一起考虑有以下表示
    $$A = \begin{cases} A & A \in [0, 2^{n-1}) \\ 2^{n-1} - A & A \in (-2^{n-1}, 0] \end{cases}$$
    - 反码(1's complement)$\sim A$：除符号位不变，其他位取反；同理小数
    $$\sim A = \begin{cases} A & A \in [0, 2^{n-1}) \\ (2^n - 1) + A & A \in (-2^{n-1}, 0] \end{cases}$$
    **分析.** *反码是全1的补数*
    $$\sim A = (2^n - 1) - A = (11\ldots1)_2 - A_2$$
    *即在*$\bmod 2^n - 1$*意义下的运算*
    - 补码(2's complement)$[A]_c$：反码+1，按照原来十进制转二进制方法即可得对应有符号十进制

数，由于没有正负0，故表示的数多了一位，补码的补码为原码；同理小数

$$[A]_c = \begin{cases} A & A \in [0, 2^{n-1}) \\ 2^n + A & A \in [-2^{n-1}, 0) \end{cases}$$

**分析.** 补码的设计非常关键，理解补码的由来对于后面的四则运算有着很大帮助。之所以要有补码，是因为希望能做到**减去一个数等于加上某个数**，而这在模$2^n$的意义下即可实现。
那么就有

$$[A]_c = 2^n - A = ((2^n - 1) - A) + 1 = \sim A + 1$$

即在$\bmod 2^n$意义下的运算，以4位二进制为例

$$(5)_{10} = (0101)_2 \implies (5)_c = 2^4 - 5 = 11 = (10000)_2 - (0101)_2 = (1011)_2$$

由$[A]_c$求$[-A]_c$要连同符号位一起取反加1

- 移码(bias)$[A]_b$：补码的符号位取反，引入目的是保证浮点数的机器零

$$[A]_b = A + 2^{n-1}, A \in (-2^{n-1}, 2^{n-1})$$

**分析.** 相当于把正数移到负数的部分，负数移到正数的部分
注意区别移码的定义$(2^{n-1})$和具体浮点数阶码$(2^{n-1} - 1)$的实施[5]

2. 十进制：
   - ASCII码
   - BCD码：四位表示一位十进制数

### 2.3.3 小数表示

- 定点数，首位符号位
  - 定点整数：小数点固定在最低位右边，$0 \le |x| \le 2^n - 1$
  - 定点小数：小数点固定在**数值部分**最高位的左边，$0 \le |x| \le 1 - 2^{-n}$
- 浮点数(IEEE 754)

| 符号S,1 | 阶码E,8,移码 | 尾数F,23,原码 |
|---|---|---|

规格化数，即令小数点前面必为1，隐含表示
移码偏置常数为127（单精度）、1023（双精度），作用为简化比较

$$(-1)^S \times 1.F \times 2^{E-127}$$

[5]原因可见https://blog.angularindepth.com/the-mechanics-behind-exponent-bias-in-floating-point-9b3185083528

**例 2.**

$$1\ 0110\ 1001\ 0001 = 1.0110\ 1001\ 0001 \times 2^{(12)_{10}}$$

指数：$12 + 127 = 139 \rightarrow 1000\ 1011$

尾数：011 0100 1000 1000 0000 0000 左对齐，因为有小数点

| 符号$S$ | 指数$E(exponent)$ | 尾数$F(mantissa)$ |
|---|---|---|
| 0 | 1000 1011 | 011 0100 1000 1000 0000 0000 |
| *1*位 | *8*位 | *23*位 |

特殊值表示

| 阶码（移码） | 尾数 | 数据类型 |
|---|---|---|
| $1 \sim 254$ | 任何值 | 规格化数 |
| 0 | 0 | 0 |
| 0 | 非零数 | 非规格化数 |
| 255 | 0 | $+\infty/-\infty$ |
| 255 | 非零数 | NAN, Not A Number |

单精度可表示范围$[10^{-38}, 10^{+38}]$，双精度$[10^{-308}, 10^{+308}]$

### 2.3.4 C语言数据类型

C语言中数据类型大小以字节为单位

| 声明 | 数据长度(32位机，Byte) |
|---|---|
| char | 1 |
| short | 2 |
| int | 4 |
| long | 4 |
| float | 4 |
| double | 8 |

## 2.4 数据存储

- 大端方式(Big Endian)：最高有效位(MSB)所在地址为数的地址，MIPS，Photoshop、JPEG
- 小端方式(Little Endian)：最低有效位(LSB)所在地址为数的地址，x86，GIF、RTF

字节交换：大端小端互换

数据边界对齐：减少访存次数，按字地址对齐（4的倍数，二进制后两位为0）

## 2.5 数据纠错

冗余校验思想，增添校验位

- 奇偶校验码：$P = b_{n-1} \oplus b_{n-2} \oplus \cdots \oplus b_0 \oplus 1$与结果的$P'$再取异或，为1则奇数位错
  只能发现奇数位出错，且不具有纠错能力
- 海明码
- 循环码

## 2.6 MIPS指令系统

所有指令都是32位宽，按字地址对齐 注：J-Type中伪直接寻址含义：PC高4位拼上26位直接目标地

表 1: 三种指令格式

| | | | |
|---|---|---|---|
| R-Type | 用于寄存器 sub rd,rs,rt | 寄存器寻址 | $32 = 6(op) + 5(rs) + 5(rt) + 5(rd) + 5(shamt) + 6(funct)$ |
| I-Type | 运算指令： ori rt,rs,imm16<br>存储指令： lw rt,rs,imm16<br>条件分支： beq rs,rt,imm16 | 立即数、基址(PC)寻址 | $32 = 6(op) + 5(rs) + 5(rt) + 16(\text{immediate})$ |
| J-Type | 无条件跳转 j target | 按字对齐的伪直接寻址 | $32 = 6(op) + 26(\text{target address})$ |

址，最后添2个0（相当于乘4）为32位目标地址
指令字段含义

- 操作码(op)
- 第一个源操作数寄存器(rs)，5位32个
- 第二个源操作数寄存器(rt)
- 结果寄存器(rd)
- 移位指令的位移量(shamt)

操作码的不同编码定义了不同的含义，若操作码相同时，再用不同功能码区分

## 2.7 MIPS语法表

三个特殊寄存器：HI、LO、PC
存储器数据指定：

- 31个32位通用寄存器(GPR)，零寄存器$r_0 = 0$
- 32个32位浮点寄存器($f_0 - f_{31}$)
- 3个特殊寄存器：HI、LO、PC
- 32位机→可访问空间 $2^{32} bytes = 4GB$（按字节编址）
- 大端方式
- 只可通过load/store指令访问存储器（不像x86算术运算也可直接访存）
- 访存地址通过一个32位寄存器内容加16位偏移量（有符号补码）得到
- 数据按边界对齐

函数调用：

- 若过程调用参数多余4个，返回值超过2个，则需要保存到寄存器的栈区中

表 2: 通用寄存器

| 寄存器 | 名称 | 用途 |
|---|---|---|
| $0 | $zero | 常量0 |
| $1 | $at | 保留给汇编器 |
| $2-$3 | $v0-$v1 | 函数调用返回值 |
| $4-$7 | $a0-$a3 | 函数调用参数 |
| $8-$15 | $t0-$t7 | 临时变量 |
| $16-$23 | $s0-$s7 | 保存(saved) |
| $24-$25 | $t8-$t9 | 其他临时变量 |
| $26-$27 | $k0-$k1 | 为OS保留 |
| $28 | $gp | 全局指针(Global Pointer) |
| $29 | $sp | 堆栈指针(Stack Pointer) |
| $30 | $fp | 帧指针(Frame Pointer) |
| $31 | $ra | 函数调用返回地址(return address) |

- 上方是调用程序压栈(pushed by caller)，保存在调用程序帧中，返回地址上方连续区域
- 被调程序使用(used by callee)，相对于帧指针(%ebp)的访问
- 栈向下增长至堆，堆向上增长至栈

表 3: MIPS指令

| | | | |
|---|---|---|---|
| 算术 | 加法 | add $s1,$s2,$s3 | $s1=$s2+$s3 |
| | 立即数加法 | addi $s1,$s2,20 | $s1=$s2+20 |
| | 乘法 | mult $s2,$s3 | HI,LO=$s2 $s3 |
| | 除法 | div $s2,$s3 | LO=$s2/$s3,HI=$s2 mod $s3 |
| | 从高位移出 | mfhi $s1 | $s1=HI |
| | 从低位移出 | mflo $s1 | $s1=LO |
| 逻辑 | 与 | and $s1,$s2,$s3 | $s1=$s2&$s3 |
| | 立即与 | andi $s1,$s2,20 | $s1=$s2&20 |
| | 左移 | sll $s1,$s2,20 | $s1=$s2<<20 |
| 传输 | 存字 | sw $s1,500($s2) | ($s2+500)=$s1 |
| | 存半字 | sh $s1,500($s2) | ($s2+500)=$s2 |
| | 存位 | sb $s1,500($s2) | ($s2+500)=$s3 |
| | 读字 | lw $s1,500($s2) | $s1=($s2+500) |
| 条件分支 | 分支等于 | beq $s1,$s2,25 | if ($s1==$s2) goto PC+4+100 |
| | 分支不等 | bne $s1,$s2,25 | if ($s1!=$s2) goto PC+4+100 |
| | 分支小于 | slt $s1,$s2,$s3 | if ($s2<$s3) $s1=1;else $s1=0 |
| 跳转 | 跳转 | j 10000 | goto 10000 |
| | switch/函数返回 | jr $ra | goto $ra |
| | 函数调用 | jal 10000 | $ra=PC+4;goto 10000 |

# 3 计算机的运算

## 3.1 位运算

位运算针对二进制数，逻辑运算针对表达式的值

| | | |
|---|---|---|
| 无符号数 | 逻辑左移 | 高位移出，低位补0 |
| | 逻辑右移 | 低位移出，高位补0 |
| 有符号数 | 算术左移 | 高位移出，低位补0 |
| | 算术右移 | 低位移出，高位补**符号位** |

移位符号<<和>>不区分算术还是逻辑移位，只由参与运算的数值决定

大于机器所能表示的最大正数称为上溢，小于机器所能表示的最小负数称为下溢

算术左移溢出判断：若移出的位不等于新的符号位，即$CF \oplus SF = 1$，则溢出

### 3.1.1 位扩展和位截断

- 位扩展：如`float`变`double`；数据存入寄存器时也要扩展，`lb`
  - 无符号数：0扩展，即前面补0
  - 有符号整数：符号扩展，即前面补符号
- 位截断：如`double`变`int`；强行丢弃长数的高位，可能溢出或数据不正确

### 3.1.2 标志位

| | | |
|---|---|---|
| NF(SF) | negative | 符号 |
| OF(VF) | overflow | 溢出 |
| CF | carry | 进借位 |
| ZF | zero | 零 |

- 进/借位CF：**无符号数**运算结果是否超出范围，即使超出结果仍对
- 溢出OF：**有符号数**运算结果是否超出范围，若超出则结果不对

## 3.2 加减法

| 原码 | 补码 | 移码 |
|---|---|---|
| 符号与数值单独运算 | 符号与数值一起运算 | 符号与数值一起运算 |
| 同号相加进位溢出 | 变形两位补码<br>01正溢出，10负溢出 | 两加数和和数符号都相同则溢出 |
| $A \pm B = A \pm B$ | $[A+B]_c = [A]_c + [B]_c$<br>$[A-B]_c = [A]_c + [-B]_c$ | $[A]_b + [B]_b = [A+B]_c$<br>$[A]_b - [B]_b = [A]_b + [-[B]_b]_c = [A-B]_c$ |
| 浮点数尾数 | 定点数 | 浮点数阶码 |

注意进位和溢出的区别，在模$2^n$意义下，加负数进位相当于回到原点；而只有同号运算才可能溢出

如$1111 - 1101 = [01111]_c + [10011]_c = [100010]_c$，进位但不溢出（补码往前往后模特性，只有**补码**有）

而$-1111 - 1101 = [10001]_c + [10011]_c = [100100]_c$，进位且溢出

浮点数加减

1. 对阶

- 小阶向大阶对齐，阶小尾数右移阶差$\Delta E = |E_X - E_Y|$
- 注意要将隐含的1移到小数部分，空出位补0
- 移出的低位保留到特定的附加位上

2. 尾数加减
3. 结果规格化（左移或右移）
4. 11入，01舍，10强制结果为偶数，00结果不变
5. 判断结果正确性，是否上下溢出（当浮点数的阶码小于机器表示的最小阶码时，则下溢，将其当为0处理）

浮点数乘除

1. 阶码相加减，判溢出
2. 尾数相乘除
3. 规格化，判溢出
4. 舍入
5. 确定符号位

C不考虑溢出的异常处理

- unsigned整型溢出：模运算
- signed整型溢出：undefined behavior
- MIPS上的C编译器会选用无符号的算术运算指令，如`addu`、`addiu`、`subu`

串行/行波进位加法器：考虑当前进位或者结合$C_{i-1}$的进位

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$
$$C_i = (A_i \oplus B_i)C_{i-1} + A_iB_i$$

并行加法器：

- 若$A_iB_i = 1$，则$C_{\text{out}} = 1$与$C_{\text{in}}$无关（进位生成）：$G_i = A_iB_i$
- 若$A_i + B_i = 1$，则$C_{\text{out}} = 1$与$C_{\text{in}}$相同（进位传递）：$P_i = A_i \oplus B_i$
- 传递进位：$C_{i-1}P_i$

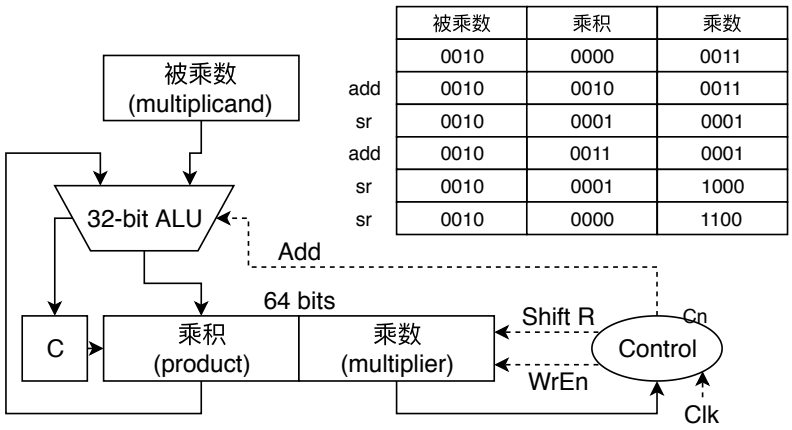
组内并行，组间串或并行(4/8位一组)

- 1个SN74181构成一个4位先行进位ALU
- 4个SN74181串行构成一个16位单级先行进位ALU
- 4个SN74181与1个SN74182(BCLA,成组先行进位芯片)构成16位两级先行进位ALU

## 3.3 乘法

### 3.3.1 原码

等同于乘数(multiplier)和部分积一起右移，或者乘数右移被乘数左移循环计数器$C_n$：循环次数31 符号位异或，数值部分绝对值相乘最多$n$次$n+1$位加法与$n$次移位构成




| | 被乘数 | 乘积 | 乘数 |
|---|---|---|---|
| | 0010 | 0000 | 0011 |
| add | 0010 | 0010 | 0011 |
| sr | 0010 | 0001 | 0001 |
| add | 0010 | 0011 | 0001 |
| sr | 0010 | 0001 | 1000 |
| sr | 0010 | 0000 | 1100 |

有符号数：符号位异或，数值部分绝对值相乘

### 3.3.2 补码

Booth乘法被乘数A符号任意，乘数B为非负数

- $A \geq 0$：与原码相同，符号位参与运算
- $A < 0$：A采用变形补码，$[A \cdot B]_c = [A]_c \cdot [B]_c$

乘数为负数

$$[A \cdot B]_c = [A]_c \cdot ([B]_c)_{\text{尾}} + [-A]_c$$

综合来说乘数为负数

$$[A \cdot B]_c = [A]_c \cdot ([B]_c)_{\text{尾}} + [-A]_c \cdot B_0$$

$[P_{i+1}]_c = 2^{-1}\{[P_i]_c + (B_{n-i+1} - B_{n-i})[A]_c\}$　　当前乘数寄存器最后两位

$[P_{n+1}]_c = [P_n]_c + (B_1 - B_0)[A]_c$

| $(B_n, B_{n+1})$ | $[P_{i+1}]_c$ | op |
|---|---|---|
| 0 0 | $2^{-1}[P_i]_c$ | $\to 1$ |
| 0 1 | $2^{-1}([P_i]_c + [A]_c)$ | $+, \to 1$ |
| 1 0 | $2^{-1}([P_i]_c + [-A]_c)$ | $-, \to 1$ |
| 1 1 | $2^{-1}[P_i]_c$ | $\to 1$ |

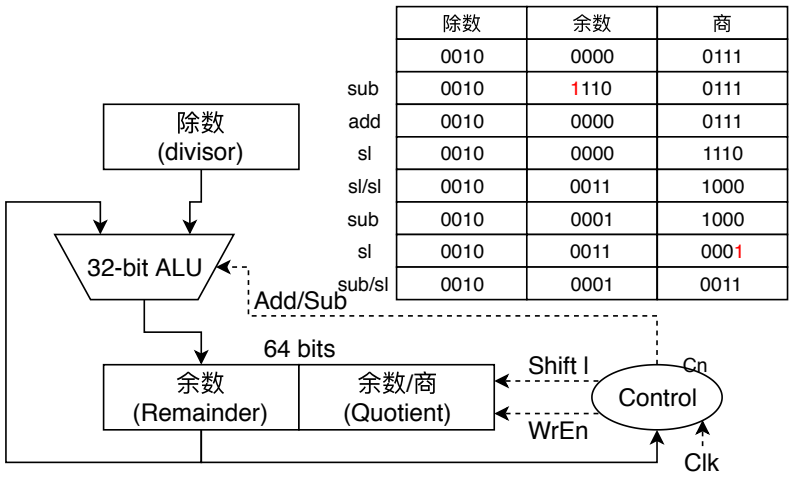
- 被乘数A和部分积P都取变形补码，乘数取一位符号位，参与运算
- 乘数末尾增设附加位$B_{n+1}$，初始值为0
- 按补码移位规则（符号右移）
- 共$n+1$次操作（看乘数几位，不包符号），第$n+1$步部分积不移位

**例 3.** 定点小数$A = -0.011$，$B = 0.101$，求$[A \cdot B]_c$

**分析.**

$$[A]_c = 11.101 \qquad [-A]_c = 00.011$$

$$[B]_c = 0.101 \qquad [P_0]_c = 00.000$$




| | 除数 | 余数 | 商 |
|---|---|---|---|
| | 0010 | 0000 | 0111 |
| sub | 0010 | 1110 | 0111 |
| add | 0010 | 0000 | 0111 |
| sl | 0010 | 0000 | 1110 |
| sl/sl | 0010 | 0011 | 1000 |
| sub | 0010 | 0001 | 1000 |
| sl | 0010 | 0011 | 0001 |
| sub/sl | 0010 | 0001 | 0011 |

| | 被乘数 | 部分积 | 乘数 |
|---|---|---|---|
| 初始化 | *11.101* | *00.000* | *0.101(0)* |
| 加-$A$ | *11.101* | *00.011* | *0.101(0)* |
| 右移 | *11.101* | *00.001* | *10.10(1)* |
| 加$A$ | *11.101* | *11.110* | *10.10(1)* |
| 右移（符号位） | *11.101* | *11.111* | *010.1(0)* |
| 加-$A$ | *11.101* | *00.010* | *010.1(0)* |
| 右移 | *11.101* | *00.001* | *0010.(1)* |
| 加$A$ | *11.101* | *11.110* | *0010.(1)* |
| 右移（仅乘数） | *11.101* | *11.110* | *[0]001(0).* |

故$[A \cdot B]_c = 1.110001$（注意乘数首位为符号位）

* $(-1) \cdot (-1)$是**定点小数**补码乘法唯一移出的情况

伪加器(Carry Save Adder, CSA)：将进位在本级加法器中保存，留待以后计算

三个$n$位数的和，需将伪加和$S_{p_i}$与左移一位的伪加进位$C_{p_i}$相加求得，普通并行加法器(CPA)

柱形乘法器，采用多级CSA和一级CPA构成

## 3.4 除法

### 3.4.1 原码

恢复余数除法

- A-B
- 若余数小于0，A+B
- 左移

加减交替法/不恢复余数法

- 符号位单独处理
- 被除数（余数）设置双符号位，便于判断溢出

- 余数为**正**/负，商为1/0，余数和商寄存器同步左移，左移后的余数**减**去/加上除数的绝对值得到新余数
- 重复$n+1$步（$n$位尾数，1位符号位）
- 最后一步余数不左移
- 若最后一步余数为负（假余数），则需加$|B|$得到正确的余数

#### 3.4.2 补码

Booth除法：先上商，后加减

- 余数符号和除数符号上商，商左移
  - 同号，商1，余数左移，减去除数
  - 异号，商0，余数左移，加上除数
- 最后一步上商，余数不变
- 商的符号取反
- 恢复余数和修正商
  - 除法除尽时，余数寄存器全0
    - 除数为正，商不必修正
    - 除数为负，所得商加$2^{-n}$
  - 除法除不尽时
    - 被除数与除数同号，且余数与除数异号，恢复余数：$[R_n]_c+[B]_c$
    - 被除数与除数异号，且余数与除数同号，恢复余数：$[R_n]_c+[-B]_c$
    - 商为证，商的反码与补码相同，不必修正
    - 商为负，商的反码在末位加1，即加$2^{-n}$

# 4 处理器

## 4.1 处理器概述

计算机五大组成部分：[控制器+数据通路（运算器）]处理器、存储器、输入、输出

- 操作元件：组合逻辑电路，所有操作元件都必须从状态元件接受输入，并并将输出写入状态元件
- 状态元件：时序逻辑电路，只有状态元件可以存储信息

**定义 1** (寄存器组(Register File))**.** 包含

1. 两个读端口（组合逻辑）：*busA*和*busB*读入地址，经过一个取数时间*(AccessTime)*后，两条线有效
2. 一个写端口（时序逻辑）：写使能为*1*且时钟边沿到达

要输入信号在寄存器的输出端才有效

一个时钟周期就是一个节拍

PC和指令寄存器的位数分别取决于存储器的容量和指令字长

## 4.2 数据通路的建立

CPU执行指令主要分为两个阶段

1. 取指阶段（公共操作）：
   - 取指令：经过一个clk-to-Q(门闩延迟)，PC得到新值，经access time后得到当前指令
   - $PC \leftarrow PC + \Delta = PC + 4$
   - 译码
2. 执行阶段：
   - 主存地址运算
   - 取操作数
   - 算术逻辑运算
   - 存结果
   - 判断检测异常事件
   - 若有异常，则自动切换到异常处理程序
   - 检测是否有中断请求，有则转中断处理

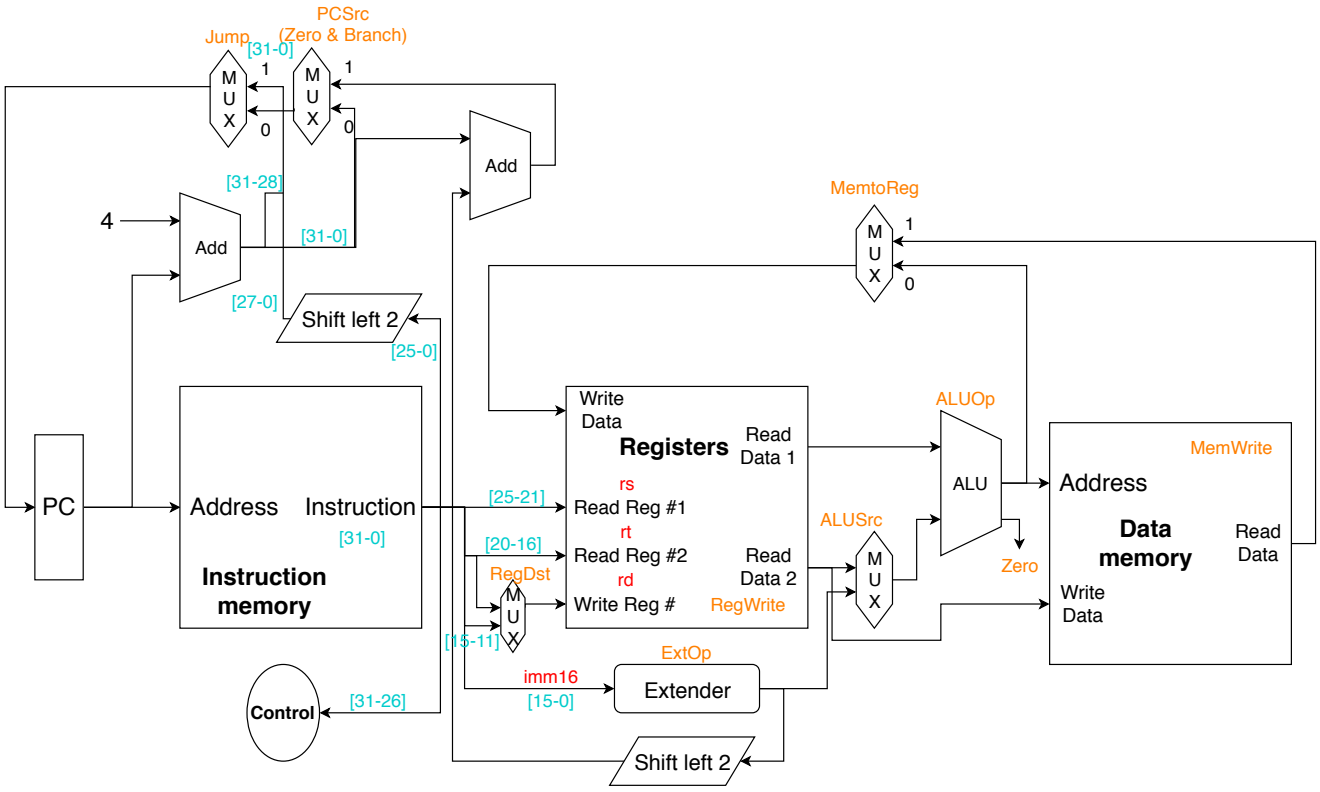



图 1: MIPS基本数据通路

MIPS中三类基本指令：R-type、I-type、J-type
七条指令

1. 加减 `add/sub rd,rs,rt`

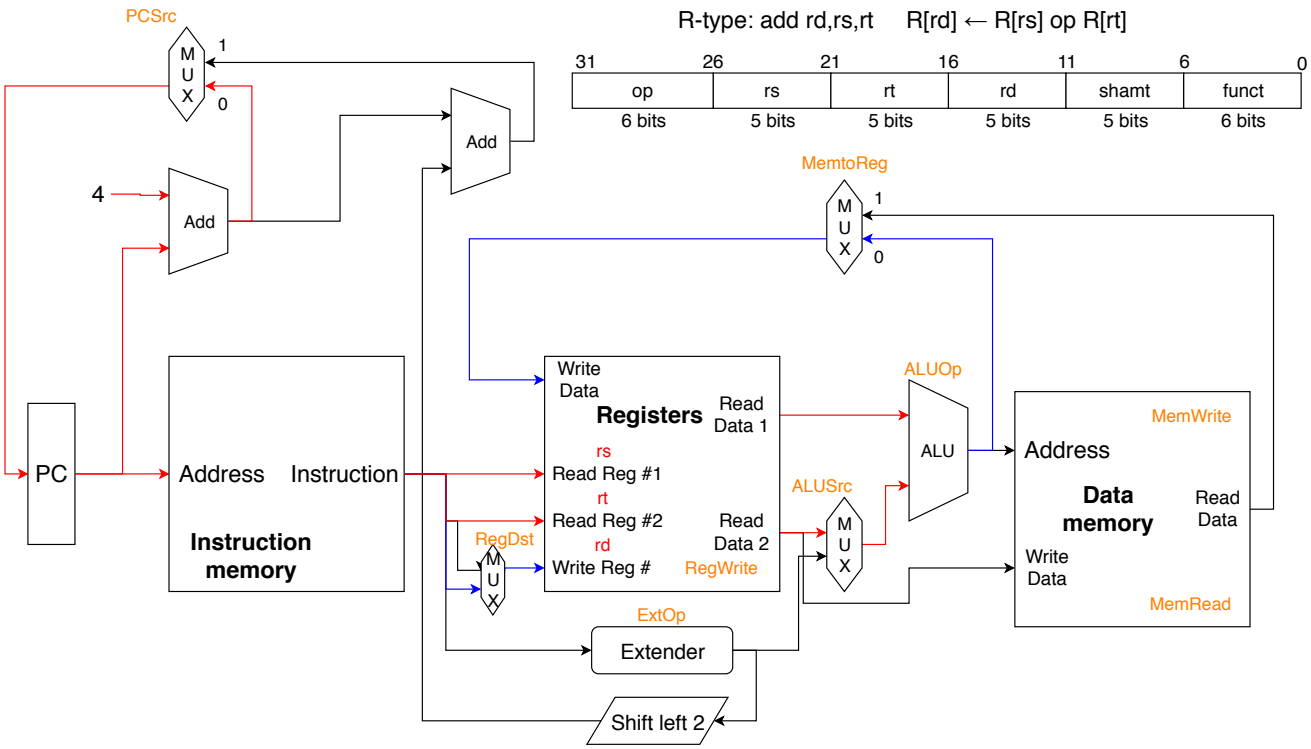



图 2: add/sub通路

2. 或立即数 `ori rt,rs,imm16`：零扩展 立即数需要零扩展(ZeroExt)为32位
3. 存 `lw rt,rs,imm16`：符号扩展
4. 取 `sw rt,rs,imm16`：符号扩展
5. 分支 `beq rs,rt,imm16`：PC只需30位，因每次加4
6. 跳转 `j target`：PC+4的高四位串接26位立即数然后左移2位

## 4.3 多周期

五个阶段：取指(IF)、译码(ID)、执行(EXE)、访存(MEM)、写回(WB)

- 每个周期都在下个时钟到来时结束（此时存储元件被更新）
- 取指结束时PC+4开始写入PC，下个周期时，PC已被更新，置IRWr=0
- ALU空闲可用来投机计算转移地址
- 有限状态机：通过组合逻辑硬连线(PLA)实现
  下一状态时当前状态和操作码的函数
- 微程序：用ROM存放微程序实现

实际机器的寄存器组和存储器情况：

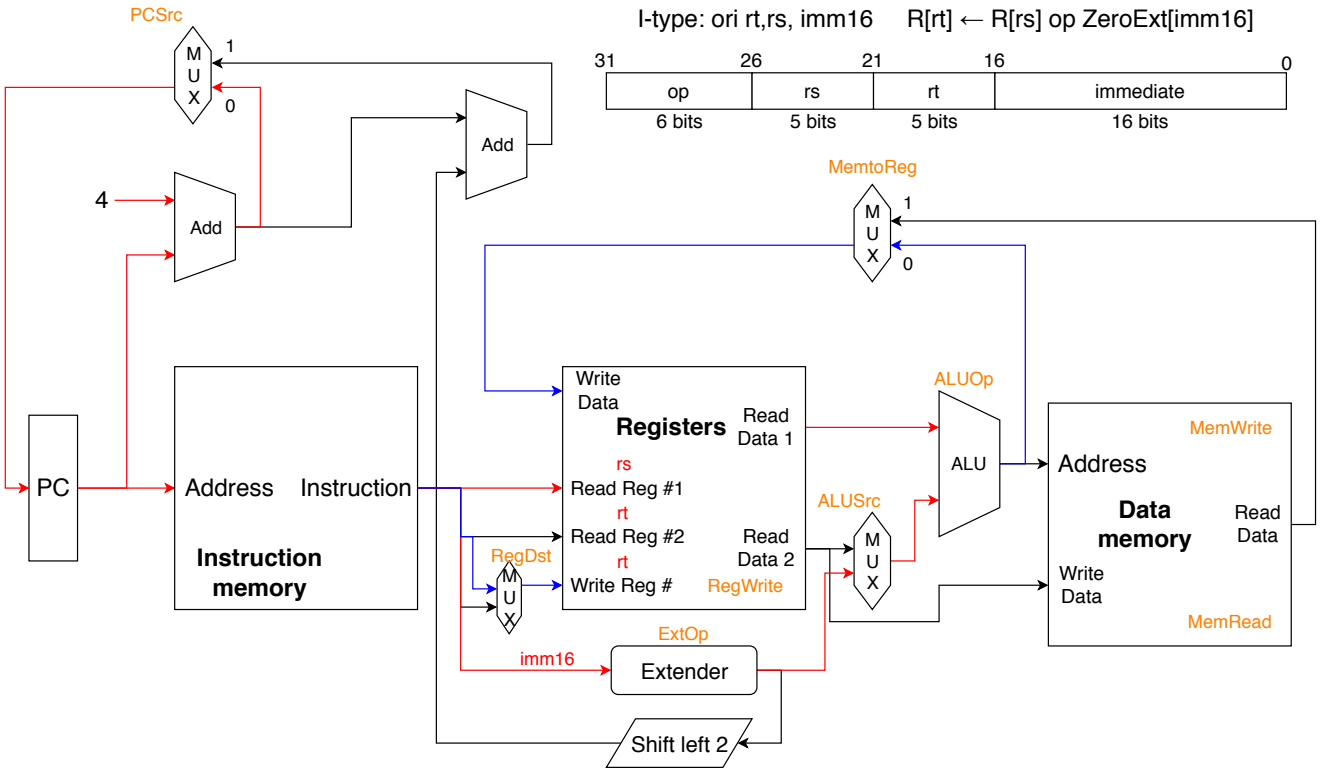



图 3: ori通路

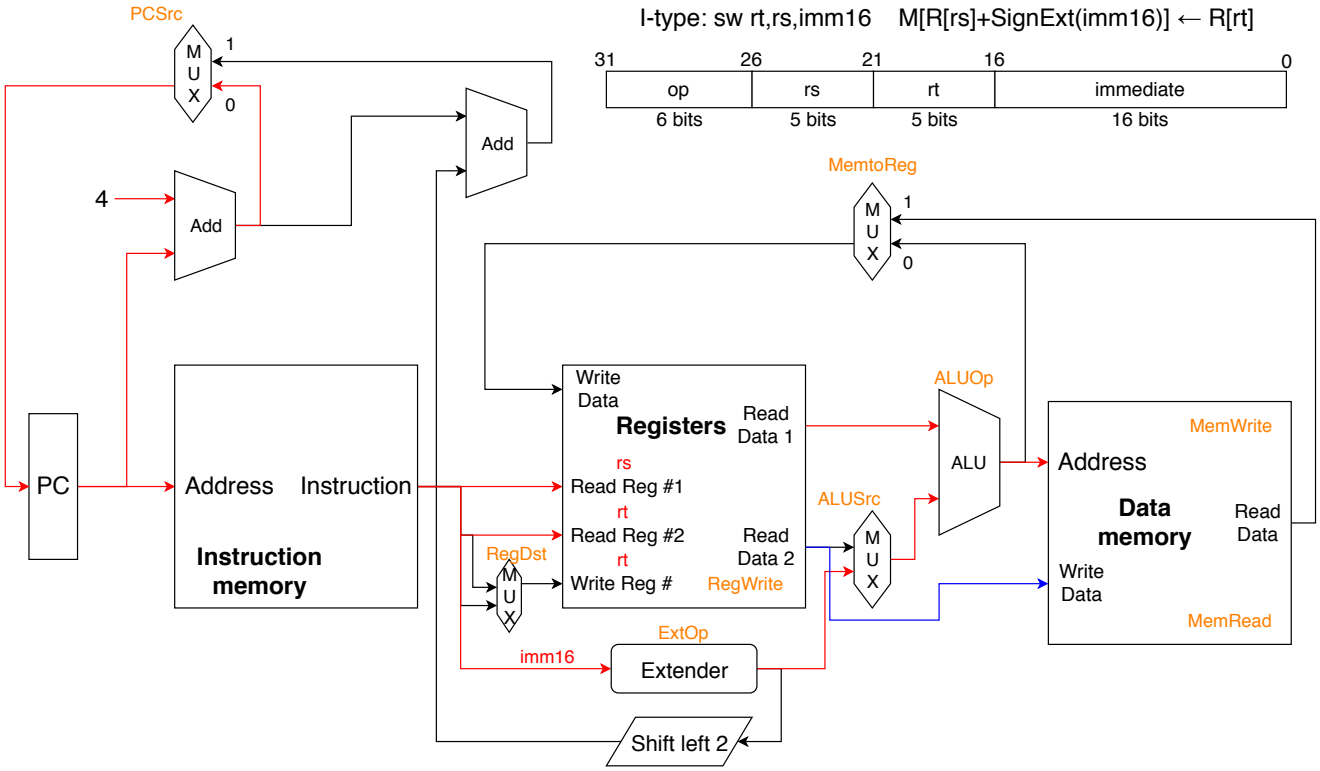



图 4: sw通路

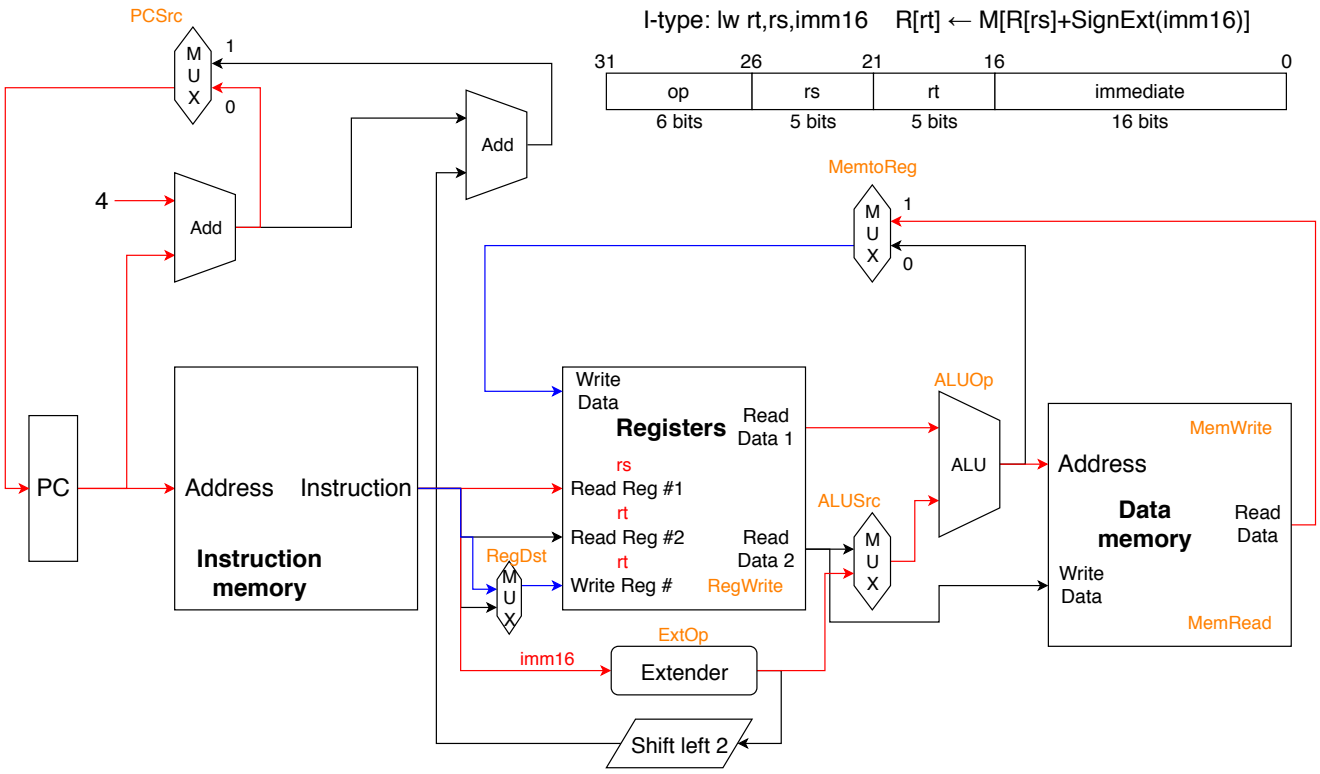



图 5: lw通路

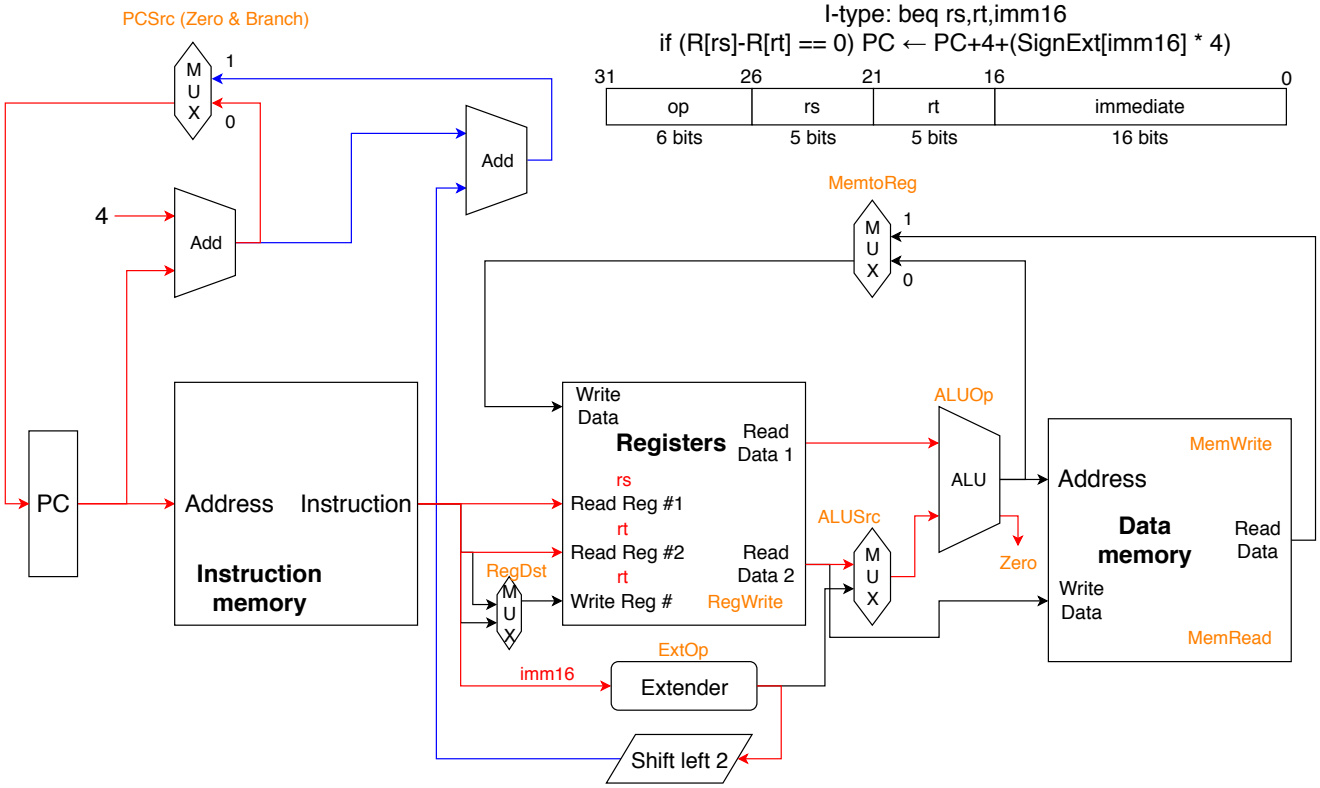



图 6: beq通路

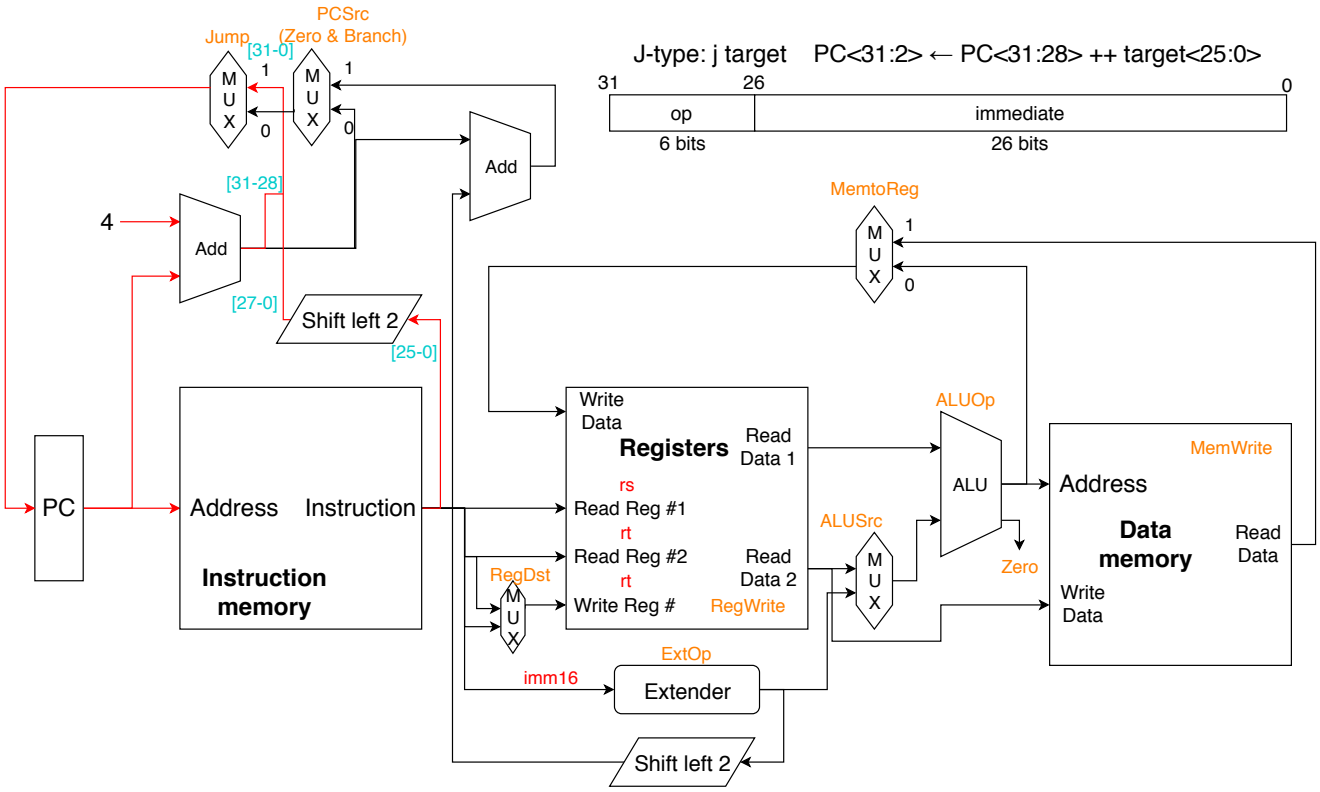



图 7: jump通路

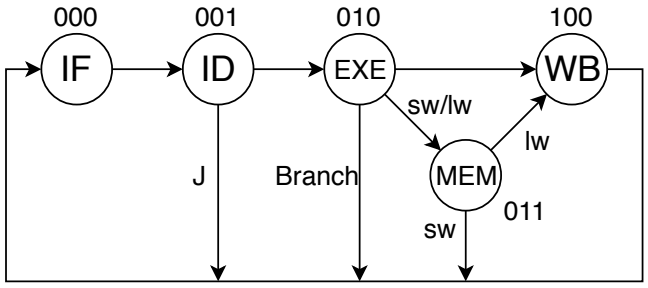



图 8: 多周期CPU状态转移表

- 寄存器有时钟输入，存储器可能无时钟输入
- 写操作不是由时钟边缘触发，而是一个组合电路：Din稳定，写使能为1，经过Write Access时间，才进行写入；会存在竞争问题

## 4.4 流水线

### 4.4.1 概述

提升工作主频：
减少每个流水级执行时间→减少每个流水级的任务量→任务再分解→增加流水线级数
副作用：

- 寄存器开销(overhead)：收益下降
- 非均匀延迟(Nonuniform delays)：吞吐率受限于最慢栈的时间，但很难将ALU和存储器划分成更小的栈

单个任务执行时间没有缩短，但是总的吞吐率增加了
时钟周期等于最长阶段花费时间$t$，N条指令执行时间$(5+N-1)\times t$
利于流水线执行的指令集

- 指令长度一致：简化取指和指令译码
- 指令格式少，且源寄存器位置相同：利于在指令未知时预取操作数
- 只有load/store指令才能访问存储器，利于减少操作步骤，规整流水线
- 数据和指令在内存中**对齐**存放，利于减少访存次数和流水线规整

### 4.4.2 冒险(Harzard)

- 结构冒险/资源冲突：一个功能部件同时被多条指令使用产生，如Load和R-type同时要写回
  - 通过加空操作(NOP)延迟写操作（每条指令都有五个阶段）
  - 设置多个部件（比如多个端口、寄存器读写口分开），避免冲突
- 控制冒险/分支冒险/转移冒险：在jump/beq之前已有几条指令被取出
  - 阻塞、NOP
  - 分支预测
    - 静态：总是预测条件不满足，或加启发式规则
    - 动态：根据历史情况(Branch History Table, BHT)进行调整（微型强化学习）
  - 指令静态调度：编译优化指令顺序，实现分支延迟
- 数据冒险/数据相关：写后读
  - 转发(forwarding/bypassing)：将数据从流水段寄存器中直接渠道ALU的输入端，如果在Data Memory读出则无法转发(Load-use数据冒险)
  - 阻塞(stall)：插入Bubble或插入NOP

– 静态指令调度：编译优化指令顺序，拉大具有数据冒险指令的距离，减少流水线可能产生的停顿（可以解决load-use）；也即先把后面无关的操作调到前面来执行；或者说利用闲置资源先干后面的事情（乱序执行）

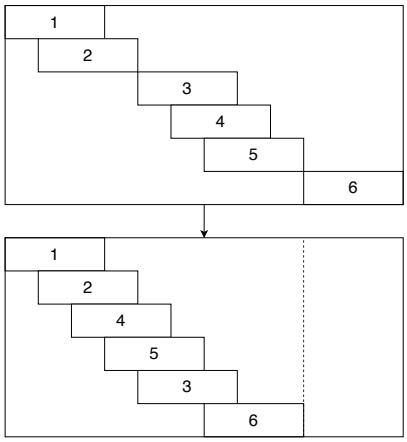



图 9: 编译器调度

#### 4.4.3 指令级并行

指令之间的相关性

- 结构相关：同时存取相同寄存器或存储器
- 数据相关：RAW、WAR、WAW（注意中英文顺序区别）
- 控制相关

指令级并行(ILP)技术

- 静态多发射：在执行前，由编译器帮助封装多条指令并处理冒险
  - 发射包：可以在给定时钟周期内发射多条指令
  - 发射槽：在给定时钟周期内能够发射指令的位置
  - 超长指令字(VLIW)：一类可以同时启动多个操作的指令集，一条指令来实现多个操作的并行执行
- 动态多发射：在运行时，由处理器发射多条指令并处理冒险
  - 超标量(superscalar)：一种高级流水线技术，可以使每个周期处理器能执行的指令数超过一条试图在一个周期取出多条指令并行执行，通过在处理器中内置多条流水线来同时执行多个处理。本质是以空间换取时间，在不同流水线中不相关地执行多条指令。允许指令以不同于原程序顺序的次序执行
  - 动态流水线调度：对指令进行重新排序以避免阻塞的硬件支持

线程级并行(TLP)技术

- 对于MIMD（多指令流多数据流），每个处理器执行自己的指令流

- 为了加大程序执行的并行力度，将程序划分为多个单一控制的执行流，每个执行流称之为一个线程。同一个进程中的不同线程共享数据空间，拥有自己的执行堆栈和程序计数器
- 提高系统整体的吞吐量

主要技术

- 超线程(Hyper-Threading)技术
  - Intel Xeon(2002)
  - 允许物理上单个处理器采用共享执行资源的方法同时执行两个或更多的分离代码流(线程)，又称**软件多线程**
  - 把单物理处理器模拟成2个或多个逻辑处理器，每个逻辑处理器都有独立的IA-32架构，即拥有自己的通用寄存器、段寄存器、控制寄存器、调试寄存器等
  - 减少CPU的闲置时间，提高系统的资源利用率
  - 逻辑处理器共享的资源包括执行引擎和系统总线接口
  - 举例：两个线程整型&浮点、运算&I/O
- 多核(Multi-Core)技术—空间并行
  - Intel Core (2006)
  - 通过在一个物理封装中集成多个分离的完整执行核来提供**硬件多线程**能力
  - 每个完整的执行核拥有独立的指令集、执行单元，即不仅有自己的AS，还拥有自己的执行引擎，总线接口与L2 Cache等
  - 本质是硬件的冗余，让不同处理器并发执行不同任务
  - 效率与性能提升要比HT技术高得多

### 4.5 异常处理

- 内部异常(Exception)：CPU发生的意外事件或特殊事件
  - 硬故障中断：电源掉电、硬件线路故障等；机器将**终止**，调出中断程序重启操作系统
  - 程序性中断：执行某条指令时发生的异常
    - 故障：执行指令引起的异常，如溢出、缺页、访问超时
    - 自陷：预先安排的事件，如单步跟踪、系统调用等（自愿中断），处理完回到下条指令!
- 外部中断：CPU外发生的特殊事件，外界发送中断请求信号，如打印机缺纸、外设准备好、DMA传输结束等

检测到异常时，处理器需进行以下操作

- 关中断：使处理器处于禁止中断状态
- 保护断点和程序状态：堆栈或特定寄存器
- 识别异常事件

– 软件识别：非向量中断(MIPS)
  EPC存放断点（异常处理后返回到的指令地址），总是存优先级最高的一个
  设置异常状态寄存器(Cause)，操作系统用一个异常处理程序，按优先级顺序查询异常状态寄存器，识别出异常事件，转入相应的中断服务程序执行
– 硬件识别：向量中断(80x86)
  用专门的硬件查询电路按优先级顺序识别异常，得到“中断类型号”，到中断向量表中读取对应的中断服务程序的入口地址
  中断向量地址=中断类型号×4

## 4.6 控制器

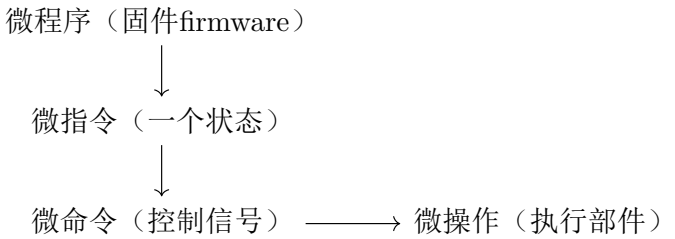
现在常用折中方案：简单指令用硬连线，复杂指令用微控制

### 4.6.1 硬连线控制器

组合逻辑/有限状态机

- 程序计数器PC
- 指令寄存器IR
- 节拍发生器Timing：有限状态机，确定指令执行顺序
- 控制信号产生电路CU

### 4.6.2 微程序控制器




微程序设计

- 将传统的程序设计方法运用到控制器的逻辑设计中
- 用规整的存储逻辑代替不规则的硬接线逻辑来实现计算机控制器功能的技术

Wilkes模型(1951)：输入指令寄存器IR中的操作码和机器状态标志；输出微操作控制信号
控制器处理一条指令的工作过程，就是启动这条指令在控制存储器中所对应的微程序，一条一条地顺序执行微指令的过程
微程序控制器工作过程

1. IR中操作码经微程序顺序控制逻辑$\mu$C变换为该条指令微程序入口的微地址码
2. 访问地址部件FCMAR选择微地址码，作为当前控制存储器的访问地址
3. 根据地址从控制存储器读出一条微指令存入微指令寄存器中，其控制信号字段表示了当前运算器、存储器、控制器及FCMAR所需要执行的所有微操作

4. 重复(2)、(3)，用微指令寄存器中地址码作为当前微地址码，直到一条指令对应的微程序执行完毕；接着执行一段微程序取下一条指令存放在IR中，然后返回(1)

- 程序计数器PC
- 指令寄存器IR
- 微指令下地址形成部件
- 控制存储器（只读，存放微程序）和微指令存储器

微指令格式：$\mu$OP、$\mu$Add（下条微指令地址）、常数（可选）
微指令编码方式：

- 水平型微指令（面向控制逻辑）：相容微命令尽量多安排在一条微指令中，最大限度并行
  - 直接控制编码：不需译码，每个微命令用一位信息表示，易并行，但编码空间利用率低（即有多少个控制信号就多少位）
  - 字段直接编码（显式编码）：将微操作划分为若干小字段，每个字段单独编码，需译码
    - 相容微操作：可同时进行，划分在不同字段
    - 互斥微操作：不能同时进行，划分在同一字段
  - 字段间接编码（隐式编码）：某些参与编码的微命令不能由一个控制字段直接定义，需要两个或两个以上的控制字段来定义
- 垂直型微指令（面向算法）：采用短格式，一条微指令只能控制一个微操作，速度慢
  - 最短编码法：将所有微命令统一编码每条微指令只包含一个微命令，每次只产生一个微操作，通过译码器产生微操作控制信号；微程序长，大量译码电路，速度慢

微指令地址产生方法：

- 顺序转移（计数器）法：下条微指令地址隐含在微程序计数器$\mu$PC中
- 断定（下址字段）法：当前微指令中显式指定下条微指令地址

# 5 存储器的层次结构

## 5.1 概述

存储器按照存储方式可分为以下几种：

1. 随机访问存储器(RAM)
   存储器任意单元可随时访问且访问所需时间相同
   - 静态(SRAM)：cache
     **触发器**（寄存器也是）；只要加电源，信息就能一直保持；集成度低，引脚多，速度快
     一般由以下几个部分组成：
     - 存储阵列（存储体）
     - 译码器电路

- 控制电路：$\overline{CE}$(Chip Enable), $\overline{WE}$(Write Enable), $\overline{OE}$(Output Enable)（后两个通常可合并）
- 数据缓冲电路

- 动态(DRAM)：主存
  **电容**；每隔一段时间必须刷新；现在一般用DDR3 SDRAM(Double Data-Rate Synchronous)

  - 分散式/异步刷新：在最大刷新间隔前将所有行刷新一遍
  - 集中刷新
  - 透明式刷新：每访存一次刷新一次

2. 只读存储器(ROM)：BIOS(Basic Input Output System)
   正常工作时只读，能随机读出，不能随机写入
   - MROM：掩模式，只读
   - PROM：一次性编程
   - EPROM/EEPROM：多次改写，后者为电可擦除可编程存储器
3. 相联存储器(Content Addressed Memory, CAM)：**快表(TLB)**
   按**内容**检索到存储位置进行读写
4. 直接存取存储器(DAS)：磁盘
   可以直接定位到要读写的数据块，存取时间的长短与数据所在位置有关
5. 顺序存储器(SAS)：磁带、电荷耦合器件CCD、VCD
   数据按顺序从存储载体的始端读出或写入，存取时间的长短与数据所在位置有关

基本术语：

- 记忆单元/存储位元/位元(Cell)：具有两种稳态的能够表示二进制数0和1的物理器件
- 存储单元/编址单位(Addressing Unit)：存储器中具有**相同地址**的那些位构成一个存储单元，又称为一个编址单位
- 存储体/存储矩阵/存储阵列(Bank)：所有存储单元构成一个存储阵列

存储位元 → 存储单元 → 存储矩阵 → 存储芯片(译码、驱动、读/写电路) → 存储模块(内存条) → 存储器

按照信息的可保存性分为：

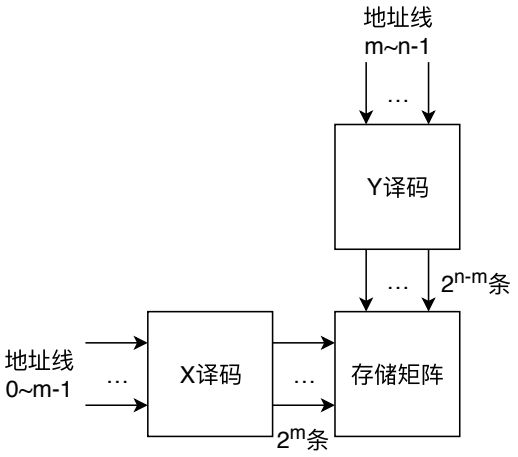
- 断电后数据是否丢失
  - 挥发性(volatile)/易失存储器：SRAM、DRAM
  - 非挥发性/非易失存储器(NVM)：ROM、磁盘、闪存
- 读出后是否保存数据
  - 破坏性存储器（读出原信息被破坏，需重写）：DRAM
  - 非破坏性存储器：SRAM

地址译码两种方式

1. 线选法（一位地址译码）
   SRAM，只在单方向译码，同时读出一条字线上的所有位，$k$位地址对应$2^k$地址驱动线
2. 位片式（二维双译码）
   $k$位地址对应$2^{\frac{k}{2}}+2^{\frac{k}{2}}$条地址驱动线
   若行列地址线复用，则可表示$2^{2k}$个地址

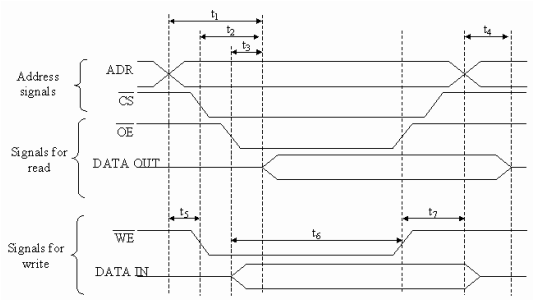
**例 4.** 某一*SRAM*芯片，容量为*1024×8*位，除电源和接地端外，该芯片数据与地址引脚的最小数目为？

**分析.** $1024B=2^{10}B$，故需*10*条地址线，*8*位则*8*条数据线，外加*1*条片选信号和*1*条读写控制线，共*20*条




存储器读写的时序：横线在中间为高阻，斜线为任意值，空白为有效

- SRAM：行列地址同时送
  地址有效→片选有效→数据有效→片选无效→地址无效




- DRAM：地址复用
  行地址有效→行地址选通→列地址、数据有效→列地址选通→数据输入→全部无效
- 读出时间$T_r$：从发出地址读命令到将数据读出来所需时间
- 存取周期$T_s$：连续两次读出两个主存单元所需的最小时间间隔（如DRAM要重写再生），$T_s<T_r$

多个请求同时读写存储器

- 双端口存储器（时间并行）：同时读写不同存储单元没有问题

- 多模块交叉存储器（空间并行）：每个体都有自己的MAR、MDR和读写电路，可独立并行工作
  - 连续编址（高位交叉）：用主存地址码高位区分体号，低位表示模块内地址；存储地址连续的数据落在同一存储体内，易发生访存冲突（同时访问一个存储体），并行可能性小
  - 交叉编址（低位交叉）：以存储体个数（质数）为模交叉编址，把连续几个地址保存到不同存储器中；CPU同时送出m个地址，高位作为存储器地址，低位负责选择数据；由存储器分时使用数据总线进行信息传递（流水线）

主存空间的划分

1. ROM区：存放系统程序、标准子程序
2. RAM区：存放用户程序

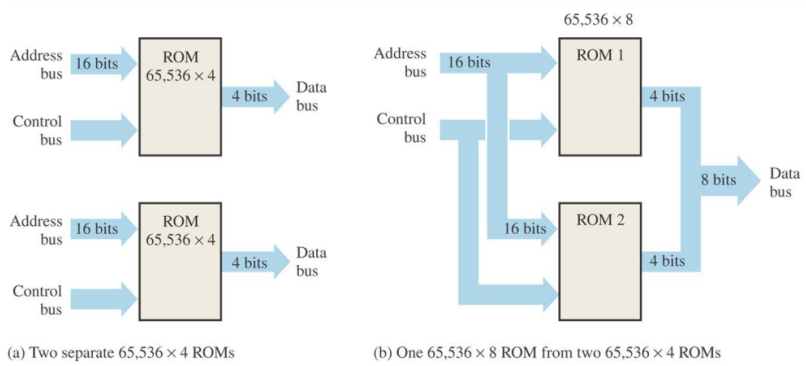
带宽/数据传输率：每秒从主存中读出的二进制数据的数量

## 5.2 存储容量扩展

多少K就要多少根地址线，比如64KiB=$2^1$6B就需要$A_0 \sim A_{15}$地址线
地址线、片选信号$\overline{CS}$、读写信号$\overline{WE}$、电源线、地线

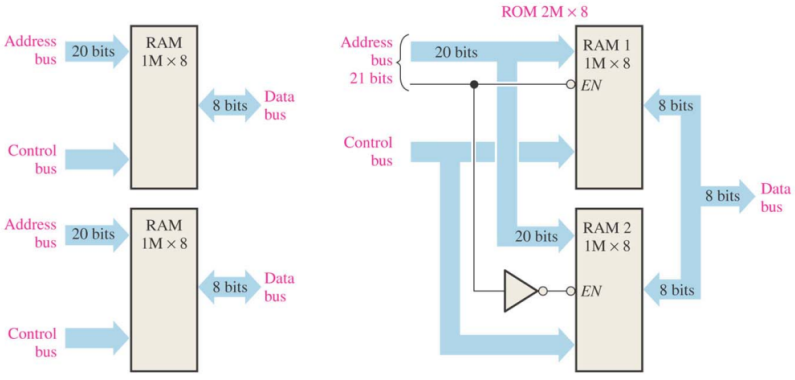
### 5.2.1 位扩展

- 存储芯片($mk \times n$位/片)构成存储器($mk \times N$位)，需要$\lceil N/n \rceil$片
- 字数不变（存储单元个数不变），位数扩展（字长加长）
- 地址线及读写控制线相连接（同时读入），数据线单独引出，不需外部译码器/片选信号




### 5.2.2 字扩展

- 存储芯片($mk \times n$位/片)构成存储器($Mk \times n$位)，需要$\lceil M/m \rceil$片
- 位数不变（字长不变），扩充容量（存储单元个数增加）
- 地址线、读写控制线对应相接，但需要加片选信号（与外部译码器相连）




(a) Individual memories each store 1,048,576 8-bit words

(b) Memories expanded to form a 2M × 8 RAM requiring a 21-bit address bus

### 5.2.3 字位扩展

**例 5.** 用$1K\times 4b$的$DRAM$芯片（$64\times 64$结构）构成$16K\times 16b$的存储器

**分析.** 1. 所需芯片数目：$32KB/0.5KB=64$

2. 位扩展$16/4=4$个，字扩展$16/1=16$组
3. 总共需$14$条地址线，因$16K=2^{14}$
4. 其中字扩展需要$4$个片选信号$16 = 2^4$，$4$条线
5. 片内寻址$10$条地址线
6. 刷新周期$64T_{refresh} < T_{max}$

## 5.3 存储器与CPU的连接

通信方式

- 异步（需握手）
- 同步：CPU和主存由统一时钟信号控制，无需应答

主存空间的划分

- ROM存放系统程序、标准子程序
- RAM存放用户程序

注意74LS138还有三个使能端$G_1, \overline{G_{2A}}, \overline{G_{2B}}$

## 5.4 Cache概述

### 5.4.1 计算机存储层次结构

大容量、高速度、低成本

多级系统的存储容量即为最底层的存储容量，因为是包含关系

CPU与cache之间以字为单位传送，cache与主存之间以块为单位传送

程序访问局部性

- 时间(temporal)局部性：刚被访问过的存储单元很可能不久又被访问，如循环变量

- 空间(spatial)局部性：刚被访问过的存储单元的邻近单元很可能不久被访问，如数组顺序访问

cache对程序员是透明的，即程序员在编写程序时无需了解cache是否存在或如何设置

### 5.4.2 命中与失效

- 命中(hit)：要访问的信息在cache中
- 失效(miss)：不在cache中

平均访问时间

$$\bar{T} = pT_h + (1-p)(T_h + T_m) = T_h + (1-p)T_m$$

影响因素

- cache越大，失效率越低，但成本越高
- block越大，失效率越低；但block在cache中所占比例增加到一定程度，失效率会上升
- cache容量小时，映射方式有影响；容量大时，影响不大

## 5.5 cache与主存的映射

cache存放一个主存块的对应单位为行(line)或块(block)或槽(slock)或项(entry)
位宽即为cache一个块的大小，如传送单位为512B，则cache每个块大小为512B
增大块的大小，以利用空间局部性
频繁的cache替换称为cache抖动

### 5.5.1 直接(direct)映射

cache内存放的内容是主存的一个子集，因此给出一个主存地址，要在cache中找到这个地址，应该将这个32位地址全部用上。末几位用于确定在cache内存储的位置，高位(tag)则用来确定是否为该地址。

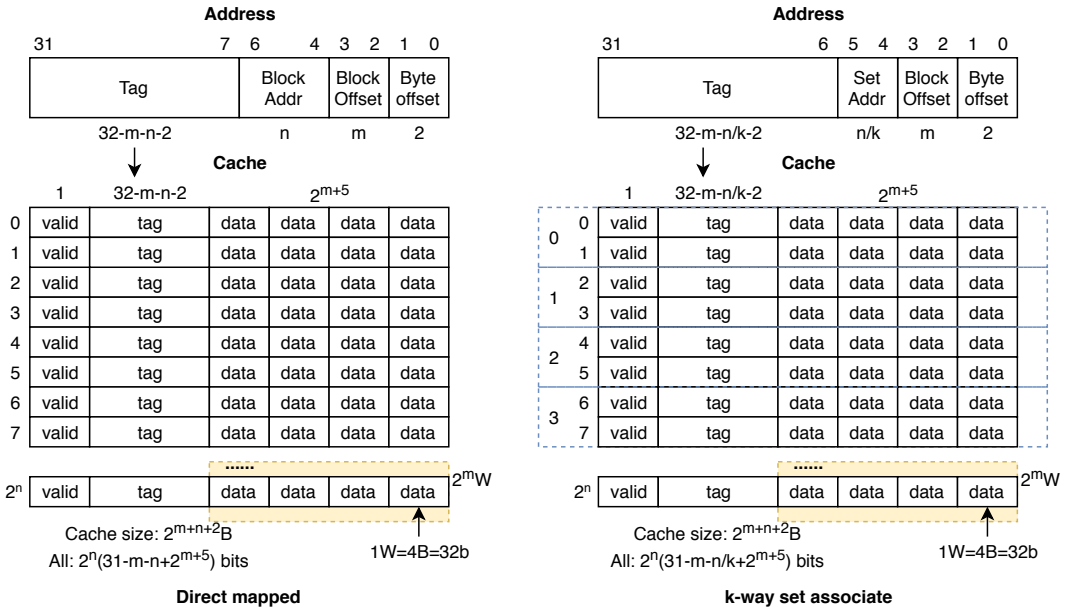
- 注意区别块地址、字地址、字节地址等（一般按字节编址）
- 按字索引则后两位不需要
- 先计算出块地址（因而先模块内数目），再计算块内地址
- $0 \sim n-1$块称为第0区/块群，$n \sim 2n-1$称为第1块群，以此类推

### 5.5.2 全相联(full associative)映射

比较地址要与所有单元比较，线路复杂，成本高

### 5.5.3 组相联(set associative)映射

直接模组的数目。相联度高，缺失率低，但会增加命中时间只要命中了tag则对应的主存块一定在cache里




**例 6.** 某一直接映射的*cache*容量为*8K*字，每块内有*16*个字，主存容量为*512K*字

**分析.**

- 主存有$512K = 2^{19}$个字，$512K/16 = 2^{19}/2^4 = 2^{15}$个块，$512K/8K = 2^6$个区
- *cache*有$8K = 2^{13}$个字，$8K/16 = 2^{13}/2^4 = 2^9$个块
- 主存字地址*19*位，区号*6*位，区内块号*15-6=9*位，块内地址*4*位$(16 = 2^4)$
- *cache*字地址*13*位，块号*9*位，块内*4*位
- 主存第$i$块位于第$\lfloor i/2^9 \rfloor$个区，调入*cache*第$i \bmod 2^9$块

## 5.6 Cache替换算法

- FIFO
- Least Recently Used (LRU)比较好：给每一个cache行设定一个计数器，值越小越经常被使用；不被用则加1
- Random

## 5.7 Cache一致性

- 写命中(hit)
  - 写直达(write through)：同时写cache和主存
    加入写缓冲(write buffer)，先存入写缓冲，当写主存操作结束后再将写缓冲数据释放
  - 写回(write back)：只写cache不写主存，没有同步更新!
    每个cache行置一个脏位(dirty bit)，若cache行中的主存块被修改，则置为1；只有当脏位为1的块从cache中替换出去时才将其写回主存
- 写不命中(miss)
  - 写分配(allocate-on-miss)：更新主存块相应单元，再将该主存块装入cache（空间局部性）

– 写不分配(no-allocate-on-write)：直接写主存，不放回

通常写直达+写分配/写不分配，写回+写分配

## 5.8 多级Cache

一般L1 Cache为分立Cache（数据指令分开放），减少命中时间获得较短时钟周期
L2 Cache为联合Cache，降低缺失率以减少主存缺失损失
Intel Core i7采用三级Cache

| L1 | L2 | L3 |
|---|---|---|
| 32KB I/ 32 KB D | 256KB | 2MB/core |
| 4-way I/8-way D | 8-way | 16-way |
| Pseudo-LRU | Pseudo-LRU | Pseudo-LRU<br>+ordered selection algorithm |

## 5.9 虚拟存储器

产生原因：同时运行更多进程，内存需求增大

1. 允许多个程序有效而安全地共享存储器，消除内存因小而有限的容量给程序设计造成的障碍
   - 可以更有效功效处理器和主存
   - 虚存实现程序地址空间道物理空间的转换，加强了各个程序地址空间之间的保护
2. 允许单用户程序大小超过主存容量，虚存自动管理主存和辅存组成的两级层次结构

### 5.9.1 概述

Cache解决系统速度，虚存解决系统容量
Cache全硬件管理，而虚存由硬件和OS共同管理

- 将内外存统一管理的存储管理机制，按需调页(demand paging)
- 虚存是主存和磁盘的抽象，OS使每个进程看到的存储空间都一致
- 虚存为每个进程提供一个假象，好像每个进程都独占主存，且主存空间极大

分页(paging)基本思想

- 将内存分为固定长且较小的存储块，每个进程也划分为固定长度的程序块
- 程序块(页/page)可装到存储器可用的存储块(页框/page frame)中
- 无需用连续页框来存放一个进程，只需将当前活跃页面调入主存
- 操作系统为每个程序/进程生成一个页表(page table)
- 通过页表实现逻辑地址到物理地址的转换(address mapping)
- 只有进程最后一个零头（内部碎片）不能使用，浪费小

逻辑地址与物理地址区别

- 逻辑地址：程序中指令所用的地址

- 物理地址：存放指令或数据实际内存地址

段式虚拟存储器

- 段是程序本身的属性，且可以有不同长度，可自由调度
- 段本身是程序的逻辑结构所决定的一些独立部分，所以分段对程序员是不透明的（而分页是透明的）
- 但因长度可变，分配主存空间不便，容易产生内存碎片

段页式虚拟存储器

- 程序按模块分段，段内再分页，进入主存仍以**页**为基本单位
- 逻辑地址由段地址、页地址和页内偏移量三个字段构成
- 用段表和页表（每段一个）进行两级定位管理
- 根据段地址到段表中查阅与该段相应的页表指针，再转向页表，然后根据页地址从页表中查到该页在主存中的页框地址，由此访问到页内某数据

### 5.9.2 组织方式

页式虚拟存储器，存在主存中

- 指令给出虚拟地址
- 每个页表记录对应的虚页情况
- valid为0说明缺页(page fault)，代价读磁盘，软件处理：当前指令执行被阻塞，当前进程挂起；缺页处理结束后，回到原指令继续执行
- 当读写操作不符合access right时，发生保护违例
- CPU执行指令时，先由内存管理单元(MMU)将逻辑地址转为物理地址
- 页大小比cache的block大得多，全相联映射
- 写回策略

页表结构

- 每个进程一个页表
- 页表项即为页表中的一项，即物理地址（页号+页内偏移）
- 页表项数由进程大小决定
- 页表在主存的首地址记录在页表基址寄存器中

页面大小不能太小，否则页面个数多，页表太大；也不能太大，一次装入页面时间太长

**例 7.** 某页式虚拟存储器，有*32*位虚拟地址，页大小为*4KB*，每个页表项占*4B*，则操作系统为进程分配的页表最大为多少？

**分析.** 最大页表项数=最多页的数目=$2^{32}/2^{12} = 2^{20}$
最大页表大小=页表项数×每个页表项大小=$2^{20} \times 2^2 B$=*4MB*

**例 8.** 虚拟地址*16*位，物理地址*12*位，页大小为*128B*，*TLB*采用四路组相联，共*16*个页表项

**分析.** 页大小$2^7B$，故7位用来做页内偏移

四路组相联，16个页表项，即4个组，故2位组偏移（TLB索引）

其余6位为TLB标志位，即15 ~ 7位都为虚拟页号

页内偏移不变，其余用来做物理页号

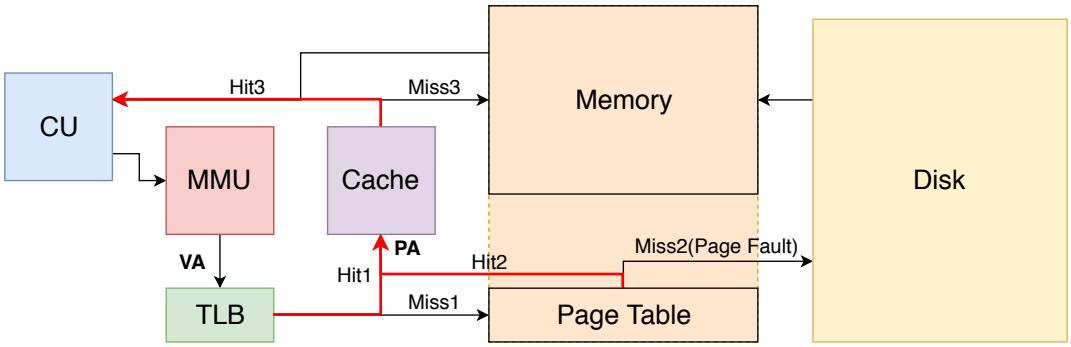
### 5.9.3 快表(TLB)

转换后备缓冲器(Translation-Lookaside Buffer)，存在cache中

每次读写操作都至少带来两次存储器访问，一次访问页表，一次访问所需的数据或指令

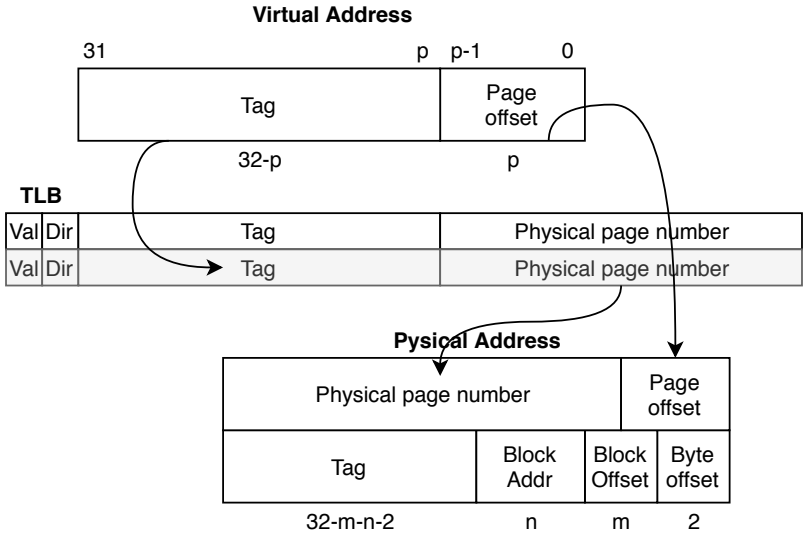
使用cache来存储页表项，它包含了最近使用的那些页表项

不可能页表miss了，快表hit了，因为页表缺失，信息一定不在主存页不在cache




TLB指标

- 多用全相联：命中率高
- 采用随机替换策略：降低替换算法开销
- 写回策略：减少访存次数
- 大小：16 ~ 512项
- 块大小：1 ~ 2项（每个表项4 ~ 8B）
- 命中时间：0.5 ~ 1个时钟周期
- 失靶损失：10 ~ 100个时钟周期
- 命中率：90 ~ 99%




### 5.9.4 存储保护

- 地址越界：转换到的物理地址不属于可访问范围
- 访问越权：访问操作所拥有的访问权限不符

通过程序重定位进行存储区域保护

- 静态：在装入前将所有地址全部转为物理地址，通过给逻辑地址加界/基准地址实现
- 动态：由硬件地址转换机构实现

## 5.10 并行主存系统

- 均匀存储访问模型(UMA, Uniform Memory Access)：访问任何存储字时间相同
- 非均匀存储访问模型(NUMA)：被共享的存储器在物理上分布于各个处理器

多处理器的cache一致性

- 软件：编译使得共享数据只放在主存，不放入高速缓存
- 硬件：硬件协议维护，关键在于跟踪所有共享数据块的状态
  - 侦听协议(snooping)：侦听到主存中一个单元被修改则将自己cache中单元副本置为无效或更新
  - 目录协议：在主存中设置一个目录表，记录共享数据的所有高速缓存行的位置和状态

# 6 输入输出系统

## 6.1 IO接口

IO接口是主机与IO设备之间数据交换的界面，它屏蔽了IO设备的差异，提供了一致的访问界面。功能如下

1. 数据格式（串并）转换和电平变换
2. 数据缓存（速度不匹配）

IO端口：IO接口中的各类寄存器
寄存器编址

- 统一编址，与主存统一编址，可以用访存指令去访问外设中的存储器
- 独立编址，对IO端口单独编号，需要专门的IO指令

接口芯片有使能端CS

## 6.2 磁盘存储器

磁盘组织：磁道、扇区（访问信息最小单位）、柱面
道(track)密度：垂直于磁道方向上（半径方向）单位长度磁介质所容纳的磁道数
位(bit)密度：单位长度磁道上所能记录的二进制信息位数
面(surface)密度：单位面积上记录的二进制信息位数=道密度×位密度
存取时间$t$=寻道时间$t_s$+旋转等待时间$t_w$+数据传输时间($t_{WR}$)
最大旋转延迟=1/磁盘转速*60s/1min
平均旋转延迟=最大旋转延迟/2（旋转半圈时间）
硬盘容量=柱面×磁头数×每磁道扇区数×每扇区字节数
转速单位为rpm或r/min

**例 9.** 设磁盘平均寻道时间为$10.5ms$，磁盘的转速为$3000$转/分，每个扇区的字节数为$512$，每条磁道的容量为$3072$字节，则磁盘存储器读写一个扇区平均访问时间为？

**分析.** 转速$50$转/分，则每转半圈为$t_w = 0.01s$
数据传输时间（看扇区数据量/一$s$扫过的数据量）$t_{WR} = 512B/(3072B \times 50) = 3.33ms$
故$t = 10.5ms + 10ms + 3.33ms = 23.83ms$

## 6.3 闪存存储器

电可擦写、可编程只读存储器(EEPROM)

- NOR：随机，可以直接按字节访问，主要用于存储程序代码(code)
- NAND：块级IO访问，主要用于存储数据(data)

读快、写慢（块擦）
通过损耗均衡(wear leveling)减少块的磨损
固态盘SSD即为闪存

## 6.4 光存储器

光存储器(Optical Disk Memory, ODM)：采用激光手段对存储介质进行读写操作

- 工作原理：将激光聚焦成极细光束在存储介质上存储信息，根据激光束和反射光的强弱不同，实现信息读写
- 优点：光盘记录密度高，单片存储容量大，非接触式读写，易于更换保管，对环境条件没有苛求

- 缺点：光盘机寻道时间长，可擦写性能不如磁盘快

按记录介质分类

- 形变型光盘：不可逆，如CD-ROM、VCD、DVD-ROM
- 相变型光盘(Phase Change Disc, PCD)：不可逆、可逆
- 磁光型光盘(Magneto Optical Disc, MDO)：可多次读写

蓝光存储密度高

## 6.5 RAID盘阵

计算机系统总体性能的提高很不匹配：处理器和主存性能改进快，但辅存性能改进慢
廉价磁盘冗余阵列(Redundant Array of Inexpensive Disks, RAID)

- 多个廉价磁盘增加容量（类似多体交叉）
- 并行工作提高数据传输速度
- 冗余进行错误恢复，进而提高系统可靠性

1. RAID0：条带化；存储容量、读写速度，但可靠性不够
2. RAID1：镜像盘1+1冗余；有可靠性，但无存储容量、读写速度
3. RAID2：条带化+海明校验码；冗余信息开销大，读性能高，写要同时写数据盘和校验盘，性能低
4. RAID3：奇偶校验法生成单个冗余盘，条区增大为KB，提高吞吐率；适用于大量顺序数据访问的应用场合，如医学图像处理等；某个磁盘损坏时，可以通过其它磁盘重新生成；校验时，须将所有未写数据盘的原数据全部读出，再与写入盘的新数据异或形成新校验数据，并将其写入校验盘
5. RAID4：独立存取技术，使用更大条区，各个盘相互独立访问，并发I/O，共享的校验盘；适合于较小的数据访问，允许并发地发生多个独立访问；多个并发写时，唯一的共享校验盘成为性能瓶颈，应用不广泛；校验时：对小块写，只须将要写数据盘和校验盘的原数据读出，再与写入盘的新数据异或形成新校验数据，并将其写入校验盘
6. RAID5：奇偶校验块分布在各个磁盘中，各个盘相互独立的并发访问；兼顾存储性能、数据可靠性和存储成本，广泛应用于文件和应用服务器、数据库服务器、万维网、邮件服务器等，可以对单盘失效进行恢复
7. RAID6：两个独立的奇偶校验系统，数据可靠性非常高，可以容两块磁盘同时出错；校验开销更大、写性能较差、复杂的控制方式，限制了RAID6的广泛应用
8. RAID0+1：条带化+镜像，单一磁盘坏了就全坏
9. RAID1+0：镜像+条带化，如果一个磁盘坏了，镜像盘重建即可

## 6.6 IO控制方式

### 6.6.1 程序查询(Polling)方式

IO完全由CPU指令控制，数据传输再CPU的寄存器与外设及其接口的数据缓冲寄存器之间进行，IO不直接访问内存

- 查询期间，可以一直不断查询（原地踏步），也可以定时查询（要保证数据不丢失）

- 完全串行工作或部分串行，适用于慢速设备（否则数据会被冲掉）
- 开销极大，CPU完全在等待外设完成

### 6.6.2 程序中断(Interrupt)/中断驱动方式

由外设主动通知CPU，可以处理异常事件

1. 中断请求：当外设准备好时，向CPU发中断请求
2. 中断相应：CPU响应后，中止线性程序执行，转入“中断服务程序”进行输入/输出操作
3. 中断处理：中断服务程序执行完，CPU返回程序断点继续执行，外设和CPU并行工作

区别中断和异常：

- 中断是在一条指令执行结束后开始查询有无中断请求，有的话立即响应，所以中断请求一定是在当前指令执行完时响应中断
- 异常发生在指令执行过程中，所以异常请求不能等到指令执行完才进行异常处理

中断响应的条件

- CPU处于开中断状态
- 在一条指令**执行完**（区别“异常”是在指令执行过程中）
- 至少要有一个未被屏蔽的中断请求

中断响应过程

- 关中断
- 保护断点和程序状态
- 识别中断源

中断判优：在同时出现的若干个中断请求中找出级别最高的，以便进行相应的中断服务

- 软件判优：轮询法
- 硬件判优
  - 串行判优：链式查询
  - 并行判优：独立请求

优先级

- 中断响应的优先级由硬件排队线路决定
- 中断处理优先级由软件设置屏蔽码决定

缺点

- 对IO请求相应慢
- 数据传送速度慢

### 6.6.3 直接存储器(DMA)访问方式

直接存储器存取(Direct Memory Access)：

- 独立于处理器、能在高速外设和主存之间直接传送数据
- 由专门硬件（DMA控制器）控制总线进行传输
- 高速设备（磁盘光盘等），成批数据交换，且单位数据间的时间间隔较短

每次传送数据都要占用一个存储周期

- 采用“请求–响应”方式
  - 每当高速设备准备好数据，就进行一次“DMA请求”，DMA控制器接收到DMA请求后，申请总线使用权
  - DMA控制器的总线使用优先级比CPU高
- 与中断控制方式结合使用
  - DMA传送前，“寻道”“旋转”等操作结束时，通过“中断”告知CPU
  - DMA控制器控制总线传送数据时，CPU执行其他程序
  - DMA传送结束时，要通过“DMA结束中断”告知CPU

CPU对中断请求的相应只能在**每条指令执行完**后，对DMA请求的响应时间可以发生在每个**总线/存取周期结束**时

DMA请求的优先级高于中断请求 DMA数据传送方式（由于可能出现CPU争用现象）

- CPU停止法（成组传送）：CPU脱离总线，停止访问主存；可通过在DMA接口中引入缓冲器减少CPU等待时间
- 周期挪用/窃取法（单字传送）：CPU让出一个总线事务周期
- 交替分时访问法：将存储周期分为两个时间片（透明DMA方式）

IO数据一致性：OS维护

- IO读操作：cache置为无效
- IO写操作：cache flush（刷新），强迫cache中被更新的数据写回内存

### 6.6.4 其他

- 通道方式：I/O模块具有独立I/O处理指令，具备执行专门I/O程序的能力。CPU只需在主存中事先组织好I/O程序，发出相应的I/O命令即可，其对I/O的干预极少
- 处理机方式：I/O模块是一个专门的I/O处理机，拥有独自的存储器和指令集，CPU几乎不参与I/O

## 6.7 串行接口

串行通信是将数据的各个位一位一位地，通过单条**1位**宽的传输线按顺序分时传送

优点：传输距离长，抗干扰强，费用低

波特率：单位时间内传送的二进制数据的位数，以位/秒（b/s）表示，也称为数据位率。它是衡量串行通信速率的重要指标。

收/发时钟直接决定了通信线路上数据传输的速率，对于收/发双方之间数据传输的同步有十分重要的作

用。
信道复用

- 时分多路复用(TDM, Time Division Multiplexing)：将一条物理传输线路按时间分成若干时间片轮换地为多个信号所占用，每个时间片由复用的一个信号占用
- 频分多路复用(FDM, Frequency Division Multiplexing)：利用频率调制原理，将要同时传送的多个信号进行频谱搬移，使它们互不重叠地占据信道频带的不同频率段，然后经发送器从同一信道上同时或不同时地发送出去

异步串行通信协议

- 起始位：标识数据帧的开始。
- 数据位：要传输的实际数据(5－8位)，先发送最低位
- 校验位：提高正确性，通常采用简单的奇偶校验
- 停止位：标识数据帧的结束

# 7 总线

## 7.1 总线概述

总线：在多个部件之间实现互连，用于分时共享方式传输公共信息的一组数据通路
常见IO总线：ISA、USB、PCI、VL-BUS
现行PC机主要系统总线是PCI和ISA
三总线结构：处理机总线、PCI总线、ISA总线
总线周期：通过总线完成一次内存或IO设备读写操作所需时间
总线性能影响因素

- 长度
- 连接元件数目

要在存储器与总线相连部分加三态门，防止随时读取数据（高阻态）
总线的分类

- 内部总线：寄存器、ALU
- 系统总线：CPU、MM、IO控制器
  - 控制线：决定总线功能强弱、适应性（控制IO、读写等）
  - 数据线：决定一次能传送数据的位数/能力
  - 地址线：决定最大寻址空间
- 通信/IO/外部总线：主机、IO设备

* 在系统总线的数据线上，不传输握手信号

总线传送控制/定时方式:总线在双方交换数据的过程中需要时间上配合关系的控制，实质是一种协议或规则

- 同步方式：统一时钟信号，电路简单，适合高速设备；时钟偏移，总线长度不能很长，按最慢的设置（CPU内部总线）
- 异步方式：比同步方式慢，总线频带窄，总线传输周期长，需要握手，用于不同存取速度设备（IO总线）
  - 不互锁：发送接收完就不理对方状态
  - 半互锁：发送方要得到接收方的相应才可进行其他事情
  - 全互锁：来回握手，接收方再得到发送方确认才可进行其他事情
- 半同步方式：wait/ready信号是单向的，不是互锁的；适用于系统工作速度不高，但又包含了许多工作速度差异较大的各类设备的简单系统
- 分离方式：总线读周期分成两个子周期（寻址+数据传送），在两子周期之间，退出总线，从设备准备数据；适用于有很多主模块（如多个处理器或多个DMA设备）的系统

总线主设备

- 总线使用请求信号→总线使用应答信号→使用总线
- 中断请求信号→中断相应信号和中断向量→等待CPU处理
- DMA请求信号→DMA应答信号→数据交换→撤销DMA请求信号

总线操作五个步骤

- 传输请求
- 总线仲裁
- 部件寻址
- 数据传输
- 总线释放

## 7.2 总线设计

高性能（宽通路，分离数据地址线）与低成本（窄通路，复用数据地址线）之间的权衡
复用：不同信号在同一总线上分时传送
总线控制器：对存储空间进行分配、启动等，仲裁哪个主设备获得总线使用权

$$\text{总线带宽（数据传输率）} = \text{总线宽度}/8b \times \text{总线时钟频率}$$

注意区别总线周期（通过总线完成一次完整数据传输所需要的时间）与系统周期
总线的操作可分为传输请求、总线仲裁、部件寻址、数据传输、总线释放

- 主设备：能申请并获得总线控制权的设备
- 从设备：被主设备访问的设备

总线仲裁(arbiter)：总线主设备请求并获得总线控制权的过程

- 分布仲裁：自举裁决、冲突检测
- 集中仲裁：菊花链（串行逐一访问）、独立请求并行判优

仲裁后，获得总线控制权的设备建立“总线忙”信号，用完即撤销

数据传输方式

- 基本：一个地址一个数据
- 成组/猝发(burst)：提高总线数据传输率，一个地址多个数据