



课程内容简介

2022年秋

教学团队

□ 主讲教师

- 刘卫东 教授 liuwd@tsinghua.edu.cn
- 陈 康 研究员 chenkang@tsinghua.edu.cn
- 陆游游 副教授 luyouyou@tsinghua.edu.cn

□ 交流方式

- 网络课堂 <http://learn.tsinghua.edu.cn>

上课地点、交流地点和时间

□ 刘卫东老师

- 上课地点：六教6A117
- 办公室：东主楼9区-409
- 答疑时间：周五下午，4:00pm-5:00pm

□ 陈康老师

- 上课地点：六教6A215
- 办公室：FIT 3-107
- 答疑时间：周一下午，2:00pm-4:00pm

□ 陆游游老师

- 上课地点：三教1202
- 办公室：东主楼8-210
- 答疑时间：周二下午，4:00pm-5:00pm

□ 餐叙

教学团队

□ 李山山 实验员

□ 助教

- 高一川 (gaoyc20@mails.tsinghua.edu.cn)
- 林家桢 (linjz20@mails.tsinghua.edu.cn)
- 康鸿博 (khb20@mails.tsinghua.edu.cn)
- 黄嘉良 (huangjl22@mails.tsinghua.edu.cn)
- 杨倚天 (yangyiti22@mails.tsinghua.edu.cn)
- 刘子昂 (liuza22@mails.tsinghua.edu.cn)
- 刘泓尊 (liuhz22@mails.tsinghua.edu.cn)
- 崔轶锴 (cuiyk19@mails.tsinghua.edu.cn)
- 王拓为 (wtw18@mails.tsinghua.edu.cn)
- 丁韶峰 (dsf19@mails.tsinghua.edu.cn)

计算机组成原理

□ 学分：4

□ 学时：64+32

□ 先修课程

- 数字逻辑，高级语言程序设计，汇编语言程序设计
(计算机系统概论)

□ 后续课程

- 操作系统，系统结构
- 编译原理

硬件系列课程

□ 计算机体体系结构 (Architecture)

- 对程序员精确描述计算机硬件的功能
- 对硬件工程师的最“抽象”的设计需求

□ 计算机组成原理 (Organization)

- 计算机体体系结构的逻辑实现
- 计算机硬件功能的集成
- 计算机硬件性能评价
- 计算机硬件优化

□ 数字电路 (Digital Logic)

- 计算机组成的物理实现
- 组成部件

主要教学内容

□ 计算机的层次结构

- 学习计算机组成原理的基本方法

□ 计算机如何执行程序

- 本课程要解决的基本问题

□ 运算器的功能、组成和运行原理

- 程序功能是如何实现的

□ 控制器的功能、组成和运行原理

- 程序是如何执行的？
- 怎样执行得更快一些

□ 存储器及层次存储系统

□ 输入/输出设备和总线

学习目标

□ 了解计算机的硬件组成

- 五大组成部件
- 其它专业课程的基础

□ 掌握计算机的运行原理

- 计算机怎样执行机器语言程序
- 计算机层次之间的交互关系

□ 设计能力

- 抽象、分层、流水、并行/串行
- 提高编程能力

□ 培养计算机系统能力

培养计算机系统能力

□ 什么是计算机系统能力？

- 系统观：整体性、关联性、层次性、动态性、开放性
- 系统方法：软件硬件协同及相互作用，层次结构

□ 如何培养计算机系统能力？

- 围绕目标：构建计算机系统
- 多课联动：课程间的衔接
- 课程实验设计：注意系统的设计和实现

□ 怎样检验是否具备计算机系统能力？

- 设计和实现“自己”的计算机系统
 - 自己的计算机硬件，自己的操作系统，自己的编译器，自己的路由器

组成原理学习目的

- 掌握单 CPU 计算机的完整硬件组成
 - 基本工作原理
 - 内部运行机制
 - 建立完整计算机系统概念
- 了解计算机系统的新技术
- 达到能独立设计一台完整计算机的水平
 - 硬件、软件齐全
 - 功能基本完整
- 知识和能力两个方面都得到提高

教学环节和学习方法

- 课堂讲授
 - PPT中需要独立阅读的知识内容
 - 阅读参考资料
 - 课后复习
 - 思考
 - 习题
 - 完成实验及报告
 - 讨论和总结
 - 考试
- 博学
 - 审问
 - 慎思
 - 明辨
 - 笃行

评分标准

□ 书面作业与小实验

- 作业或小实验缺2次（含），作业成绩为0
- 发现抄袭现象，作业成绩为0
- 若作业成绩为0，则考试无效
- 网上按时提交各个部分作业，迟交酌情扣分

□ 实验和报告

- 实验报告可按照要求，提交电子版

□ 考试

□ 总成绩评定（最终为等级制成绩）

- If 考试成绩 \geq 全年级考试成绩的平均值/2
Then 总评成绩 = 考试成绩*40% + 实验成绩*50% + 作业成绩*10%
Else 总评成绩 = 考试成绩
- 根据总评成绩，评定等级成绩

实验

- 实验1：监控程序，熟悉RISC-V汇编语言编程
- 实验2：计数器实验，熟悉 Vivado 开发软件和工程模板，编写简单的时序逻辑和组合逻辑
- 实验3：运算器实验（ALU+寄存器堆），熟悉ALU与寄存器堆的实现方法，适应硬件编程思维
- 实验4：总线Slave实验（SRAM控制器），理解 Wishbone 总线的基本概念，熟悉 SRAM 时序
- 实验5：总线Master实验（内存串口），熟悉 Wishbone 总线 Master 的实现，理解总线概念
- 实验6：处理器实验，，实现简单的处理器设计
- 综合实验：流水线处理器计算机系统实验（选做）
 - 基本要求：实现基本的流水线的处理器，驱动串口和静态内存，可以运行基本版监控程序
 - 扩展要求：尽量消除指令之间的冲突，进行性能分析和比较，扩展功能（中断），扩展功能（虚拟内存，应用程序、编译器）

实验

- 实验指导书
 - 在线指导书：<https://lab.cs.tsinghua.edu.cn/cod-lab-docs-2022/>
- 实验通过在线实验平台提交
 - 在线实验平台：<https://lab.cs.tsinghua.edu.cn/thinpad/>
 - 账号和密码由个人info账号统一认证登录
 - 代码托管：<https://git.tsinghua.edu.cn/>
 - 账号和密码由个人info账号统一认证登录
- 每次实验之前需通过评测后，方可进行实验
 - 需认真完成评测题目

实验评分标准

- 前6个实验要求所有同学个人独立完成
 - 满分70分：个人独立完成实验1,2,3,4,5,6
- 计算机系统实验（综合实验）鼓励同学们去做，自由组合（一般为三人），原则上按组给成绩
- 计算机系统实验（综合实验）评分参考
 - 80分：完成个人实验+流水线处理器运行监控程序的基础版本
 - 80分-100分：鼓励额外功能性能特点，包括且不限于中断、虚拟地址、外设驱动、ucore执行、特色应用、指令多发射、SIMD指令、动态分支预测、缓存等
 - 100分：能够运行ucore或同等水平

Honor Code



考试

- 闭卷考试
- 一张A4纸，正反面都可以
 - 可手写，可打印
 - 写上姓名、学号
 - 连同答卷一起上交
- 四个考试专用章，现场盖章

教学要求

□ 课堂纪律

- 按时上课，不迟到，不旷课
- 认真听讲，积极思考
- 不带食品到教室

□ 诚信要求

- 独立完成作业，不得抄袭
- 分组独立完成实验及实验报告
- 考试不作弊

教材和参考书

□ 教材

- Computer Organization & Design: the Hardware/Software Interface, 5th, RISC-V Edition, 机械工业出版社
- 在线实验教程

□ 参考书目

- 《计算机组成—结构化方法》 刘卫东 宋佳兴译, 人民邮电出版社
- 《计算机系统实验》 刘卫东 等, 高等教育出版社

实验安排

- RISC-V系统实验
- 实验时间与往年相同（参考教学日历，仅供参考，会适时调整）
- 实验提交源代码，在线编译和测试
- 尽快熟悉实验环境，理解实验内容

实验的截止日期

- 9月13日~9月20日，实验1
 - 9月23日~9月30日，实验2
 - 9月30日~10月14日，实验3
 - 10月14日~10月28日，实验4&实验5
 - 10月28日~11月15日，实验6
 - 11月15日~12月9日，综合实验
-
- 请各位同学务必仔细阅读实验指导书，如果有疑问向助教以及主讲教师尽快提出
 - 实验一定要尽早进行规划
 - 注意：如果数字逻辑内容不熟悉的同学，请务必提前自学数字逻辑部分内容

谢谢



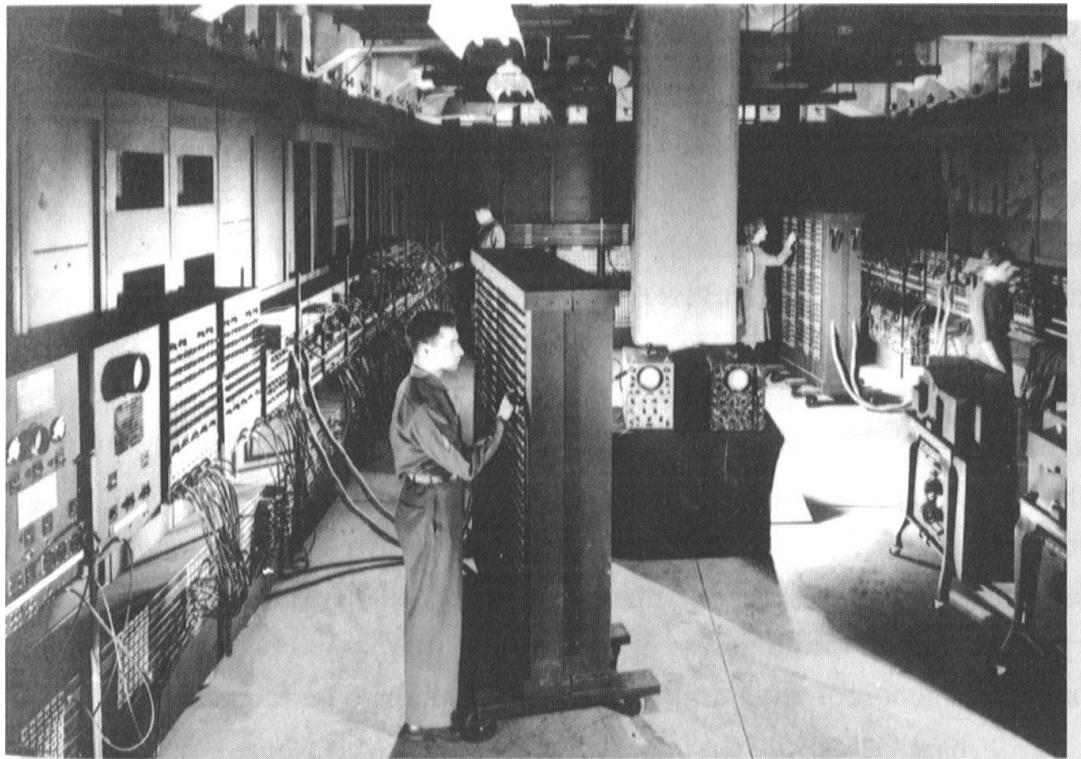
计算机系统概述

2022年秋

计算机是什么

- 一种高速运行的电子设备
 - 用于进行数据的算术或者逻辑运算
 - 可接受输入信息
 - 根据用户要求对信息进行加工
 - 输出结果
-
- A calculating machine, esp. an automatic electronic device for performing mathematic or logical operations; freq. with defining word prefixed, as analogue, digital, electronic computer.
 - –Oxford English Dictionary

计算机基本组成



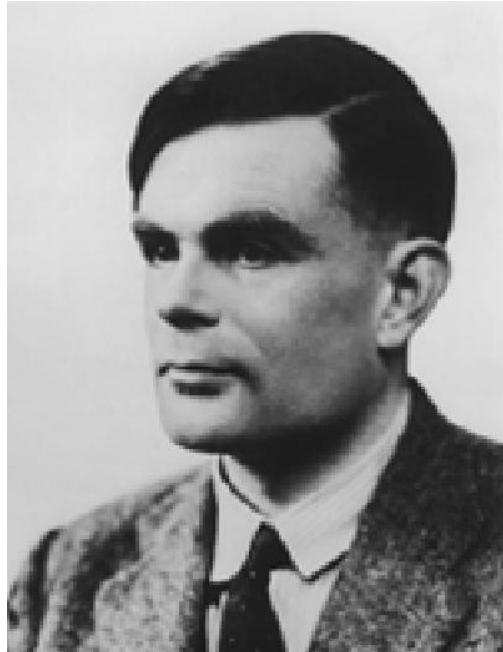
□ 计算机理论基础诞生超过60年

- Turing
- Shannon
- Von Neumann

□ 组成计算机的关键部件也没有大的改变

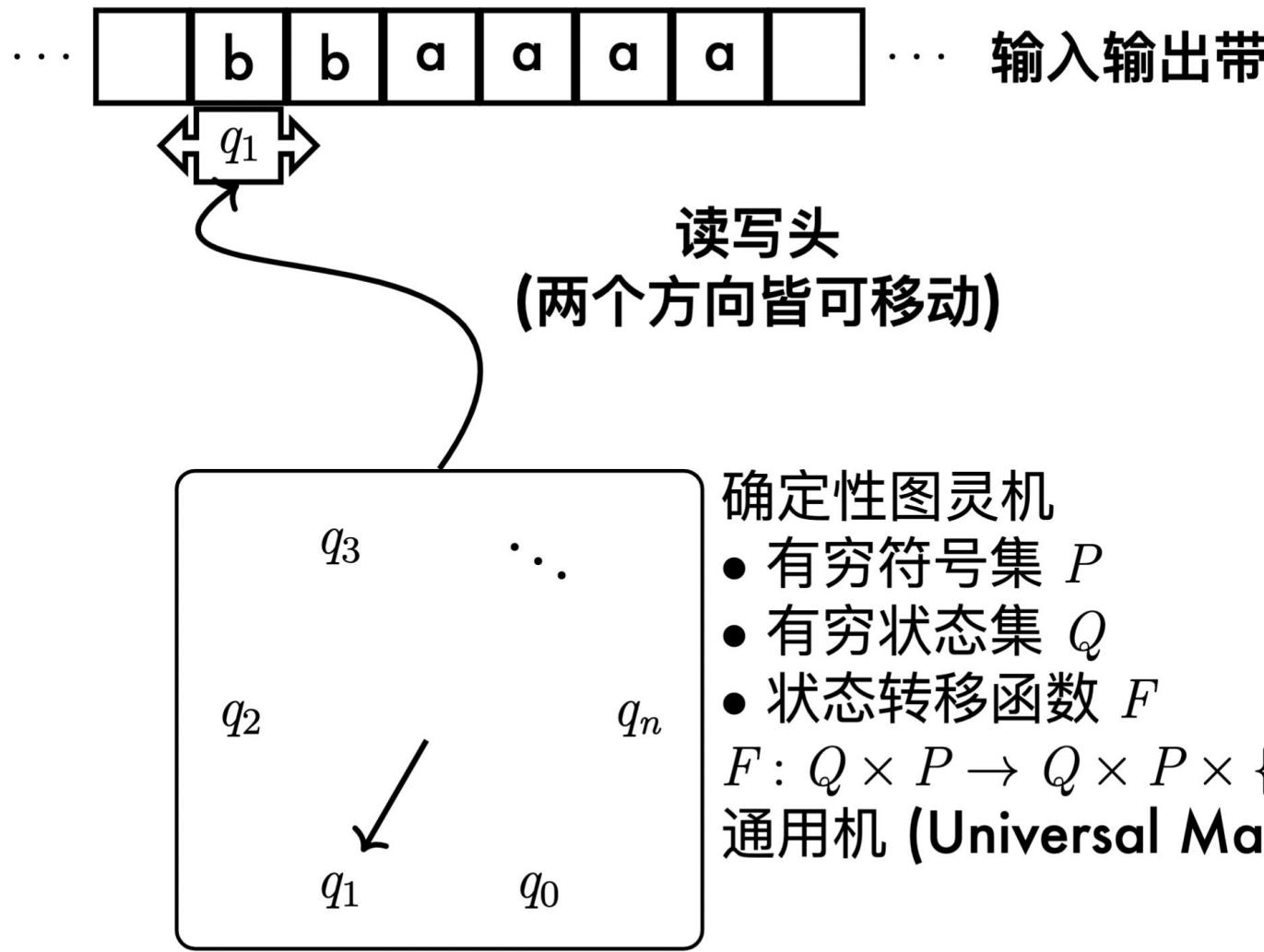
- CPU
 - Data Path
 - Controller
- Memory
- IO

图灵机



- 1937年，Alan Turing提出一种“通用”计算机的概念，它可以执行任何一个描述好的程序（算法），实现需要的功能，形成了“可计算性”概念的基础。
- 存储程序的思想，使计算机从专用走向通用。正是这一创新，开创了计算机的新时代。
- 50年代，Turing提出了“智能”计算机的概念。

图灵机 (Turing Machine)



确定性图灵机

- 有穷符号集 P
- 有穷状态集 Q
- 状态转移函数 F

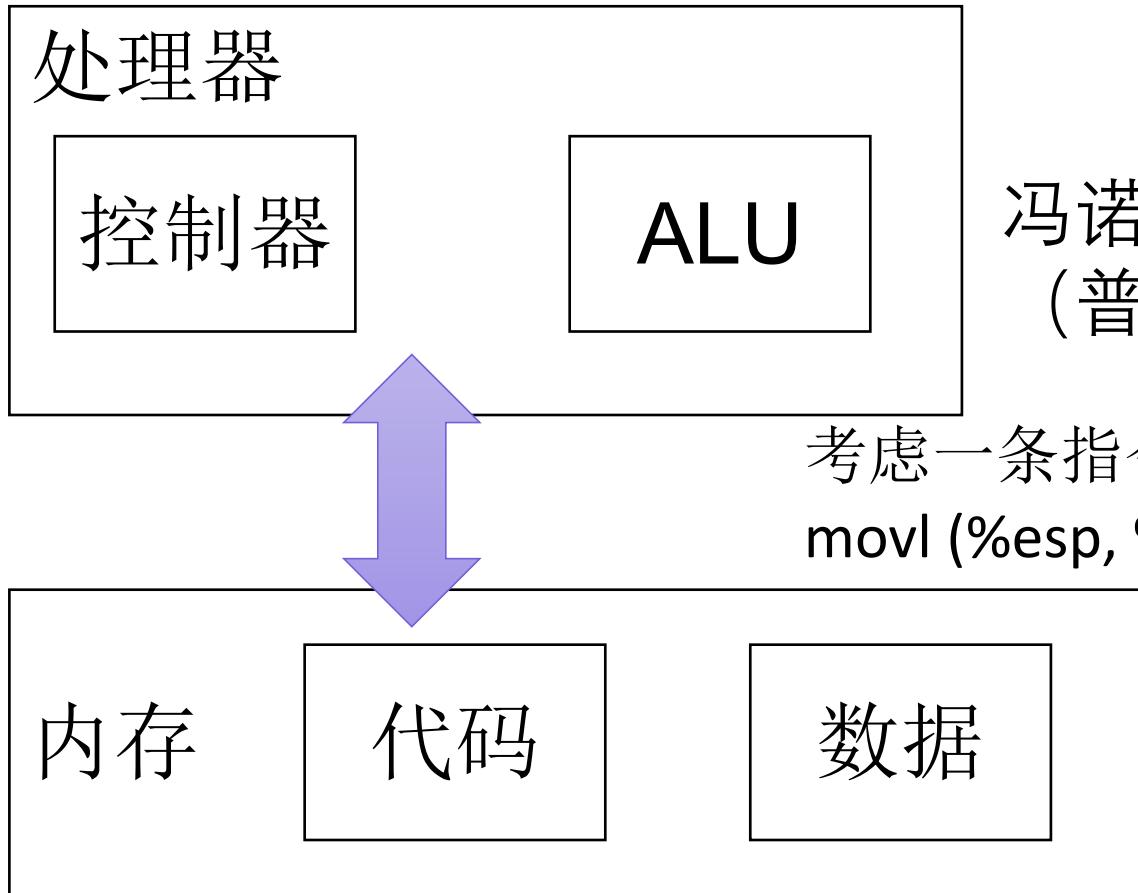
$$F : Q \times P \rightarrow Q \times P \times \{L, 0, R\}$$

通用机 (**Universal Machine**)

图灵机特点

- 通用计算机：确定了现代计算机的理论基础。
- 存储程序计算机：问题的求解由程序或过程给出，程序和过程可以通过语言描述。
- 有限速度：计算机执行程序的时间是有限的。
- 有限空间：计算机程序的存放空间和数据存放空间也是有限的。
- 奠定了现代计算机的理论基础。

Von Neumann 计算机



存储程序、二进制、体系结构

Von Neumann 计算机本质特征

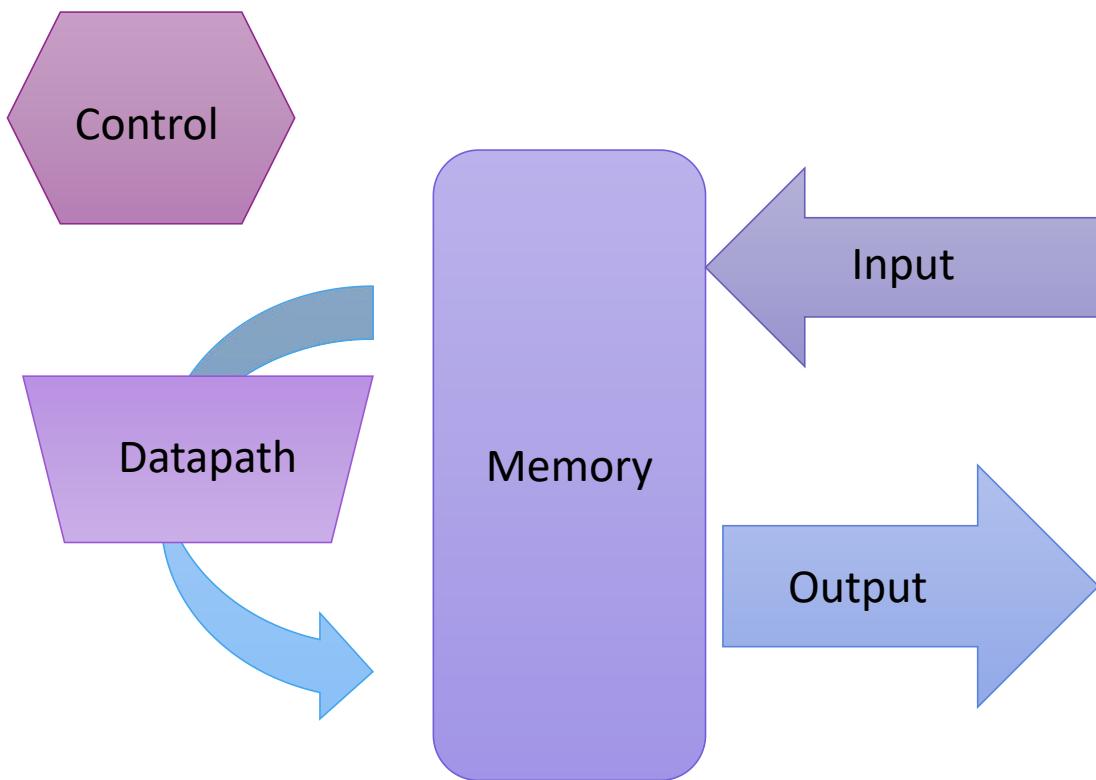
□ 存储程序

- 指令被存储在内存中
- 内存是指令和数据的统一的存储
 - 对一个存储值的解释取决于控制信号

□ 顺序指令处理

- 每次处理一条指令（获取、执行、完成）
- 程序计数器（指令指针）指向当前指令
- 除了控制转移指令外，程序计数器是按顺序推进的

计算机运行机制



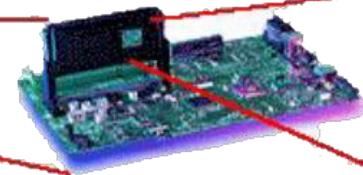
- **Datapath:** 完成算术和逻辑运算，通常包括其中的寄存器。
- **Control:** CPU的组成部分，它根据程序指令来指挥 datapath, memory以及I/O 运行，共同完成程序功能。
- **Memory:** 存放运行时程序及其所需要的数据的场所。
- **Input:** 信息进入计算机的设备，如键盘、鼠标等。
- **Output:** 将计算结果展示给用户的设备，如显示器、磁盘、打印机、喇叭等。

计算机剖析

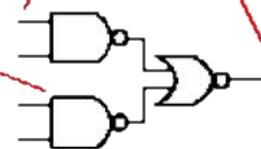
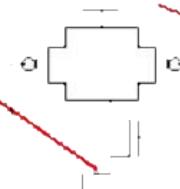
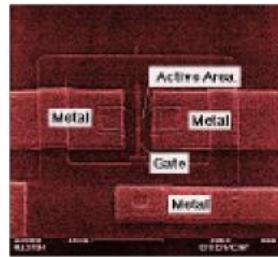
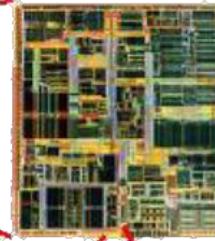
计算机系统



计算机结构



计算机组成与实现



生产制造

电路设计

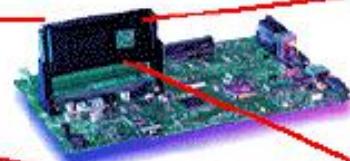
逻辑设计

计算机语言与计算机层次

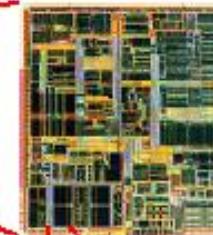
计算机系统



计算机结构



计算机组成和实现



```
while (event = getnext()) {  
    /* process event */  
    switch (event->type) {  
        case BUTTONUP:  
            win = event->W;  
            if (!win) break;  
            do_button (win);  
            break;  
        case BUTTONDOWN:  
            ...  
    }  
    ...
```

```
jal _getnext  
ori $a0,$0,0  
lw $t0,8($v0)  
lw $t0,12($t0)  
beq $t0,0,0x401834  
li $t1,4  
beq $t0,$t1,0x4018a0
```

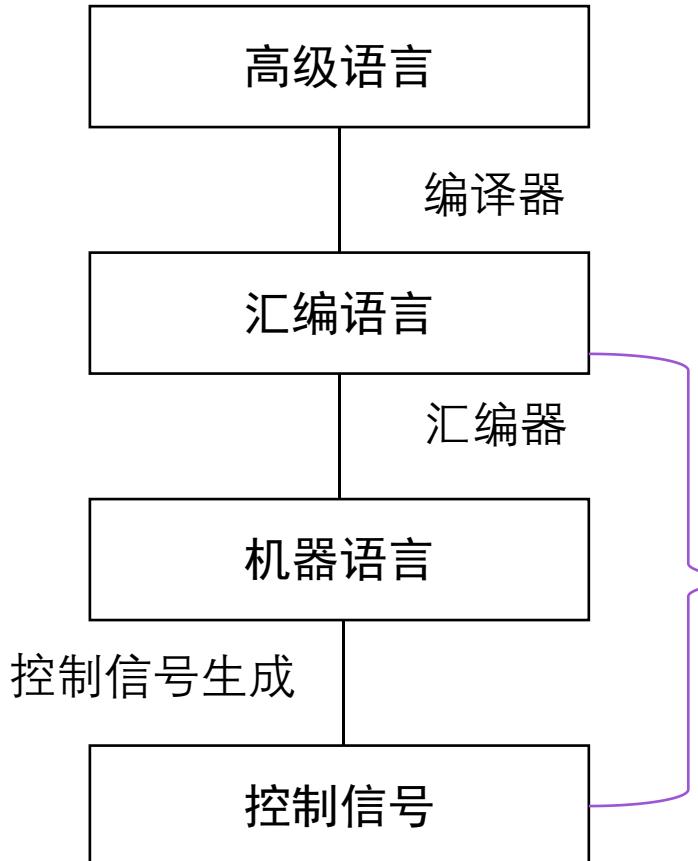
0x0c004841
0x00000000
0x34040000
0x8c480008
0x00000000
0x8d08000c
0x10001834
0x00000000
0x24090004
0x11090002
...

高级语言

汇编语言

机器语言

计算机的层次结构



$\text{temp} = \text{v}[k];$
 $\text{v}[k] = \text{v}[k+1];$
 $\text{v}[k+1] = \text{temp};$

lw t0, t2, 0
lw t1, t2, 4
sw t1, t2, 0
sw t0, t2, 4

计算机组成

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

如何使用计算机

□ 单用户、独程序

- ENIAC，仅运行一个程序，运行其它程序时要进行硬件编程

□ 单用户、单程序

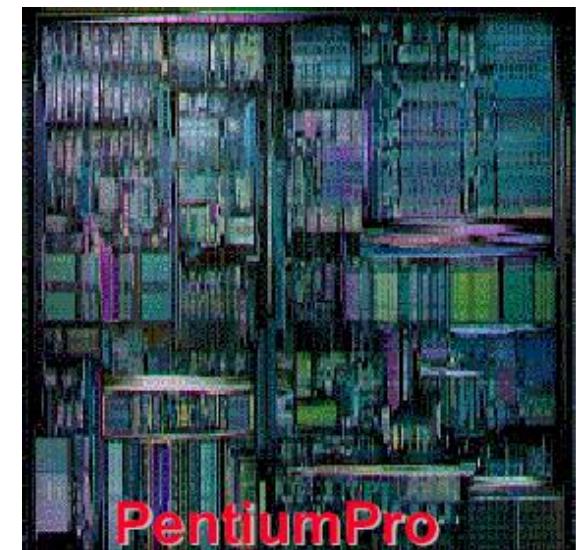
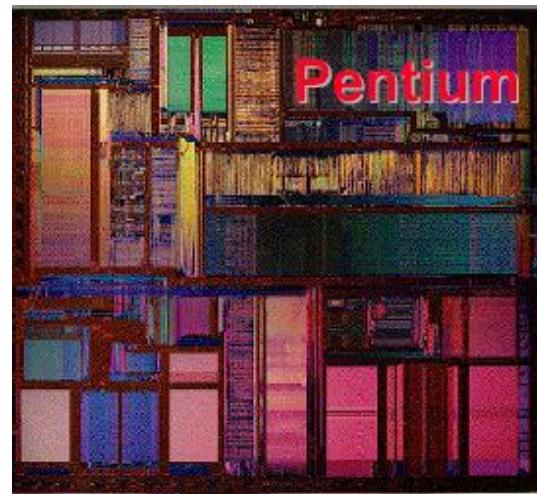
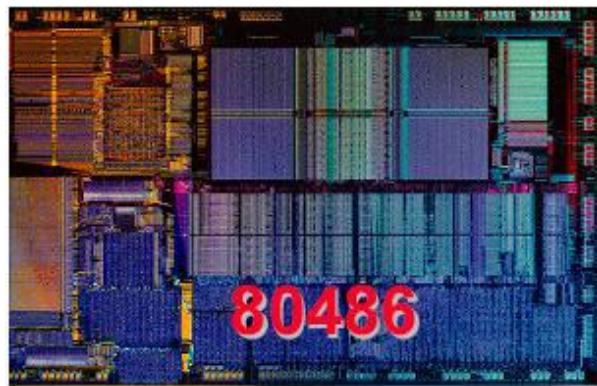
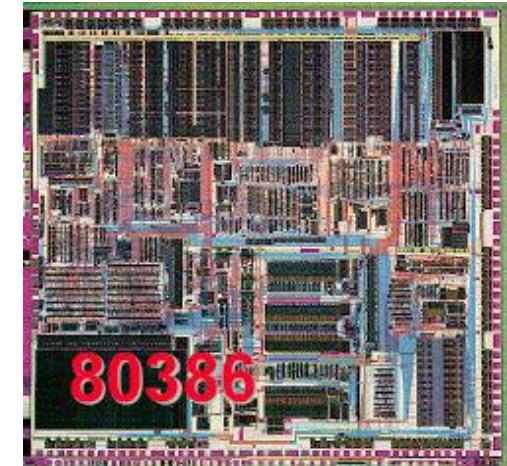
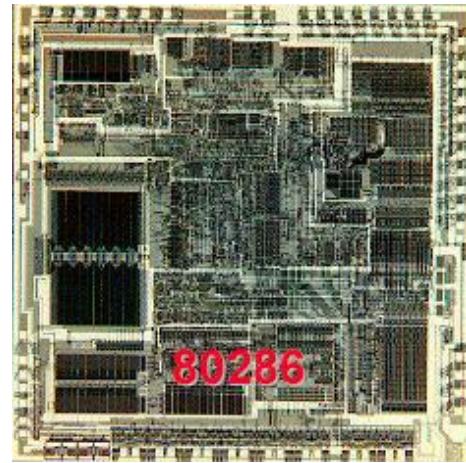
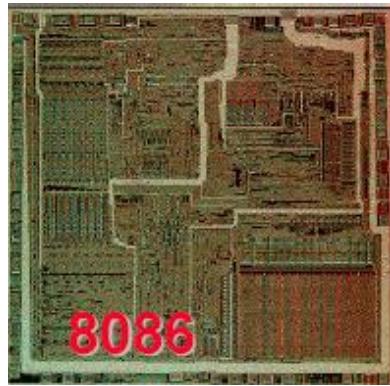
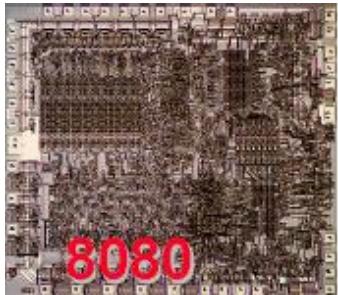
□ 多用户、单程序

- 分时，操作系统进行简单管理（共享设备驱动等）
- 1960年，IBM709FMS

□ 多用户、多程序

- 时间片进一步划分
- 1969年，UNIX，通用操作系统

应用的普及



学习与思考方法

- 1. Abstraction(Layers of Representation/Interpretation)
 - 2. Moore's Law
 - 3. Principle of Locality/Memory Hierarchy
 - 4. Parallelism (Pipeline)
 - 5. Performance Measurement & Improvement
 - 6. Dependability via Redundancy
-
- 计算机组装，体系结构会经常随着底层技术和上层应用的变化而变化，新出来的概念繁多，需要从中总结出规律性的东西
 - 学习组成原理的时候，需要经常思考上面的想法是如何落实到实际的计算机组装中的
 - 在课程的不同部分会体现上面的某一点或者几点

小结

□ 学习方法

- 博学 审问 慎思 明辨 笃行

□ 计算机组成原理

- 单CPU计算机完整的硬件系统的基本原理与内部运行机制

□ 计算机的层次结构

- 理解计算机系统和结构的钥匙

□ 计算机的发展历史

- 以史为鉴，可知兴替，可明得失。

谢谢



计算机指令系统

2022年秋

内容概要

- 计算机程序及分类
- 指令系统基本知识
- RISC-V指令系统简介
- THINPAD RV32指令系统
- THINPAD指令模拟器

计算机程序

- Computer programs(also software programs, or just programs) are instructions for a computer. A computer requires programs to function, and a computer program does nothing unless its instructions are executed by a central processor. Computer programs are either executable programs or the source code from which executable programs are derived (e.g., compiled).

- 程序员和计算机硬件之间交互的语言
- 高级语言
- 汇编语言
- 机器语言

Von Newmann结构计算机

□ 存储程序计算机

- 程序由指令构成
- 程序功能通过指令序列描述
- 指令序列在存储器中顺序存放

□ 顺序执行指令

- PC指向需要执行的指令
- 从存储器中读出指令执行
- 读取完成之后，PC自增，指向下一条指令

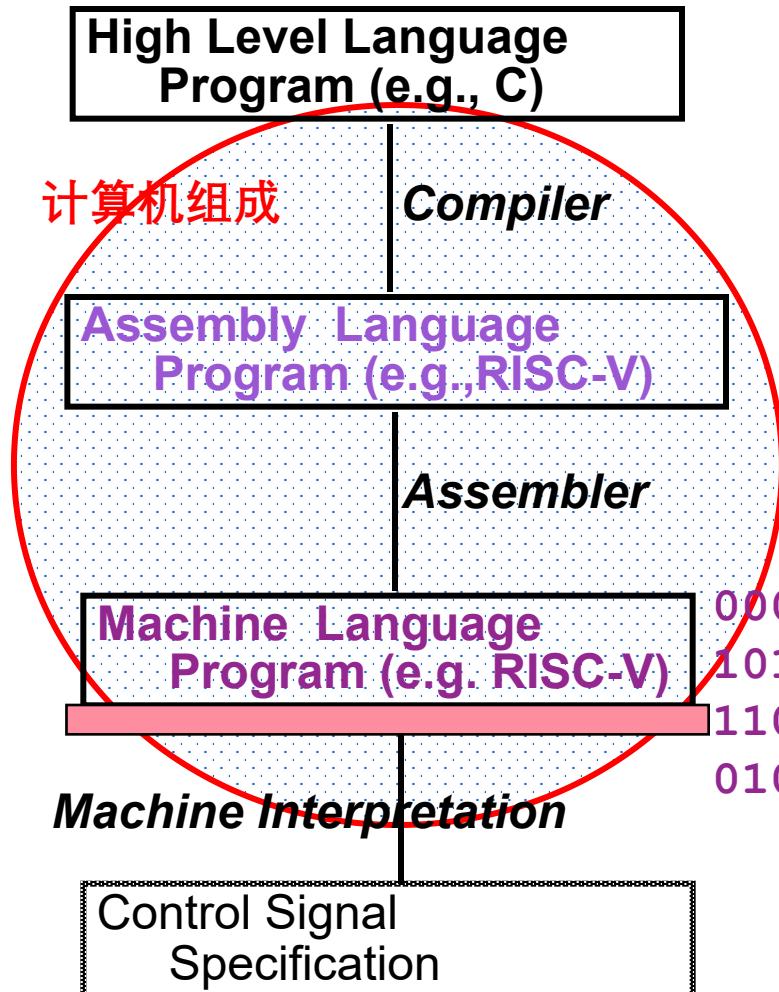
高级语言

- 高级语言又称算法语言，它的实现思路，不是过分地“靠拢”计算机硬件的指令系统，而是着重面向解决实际问题所用的算法，瞄准的是如何使程序设计人员能够方便地写出处理问题和解题过程的程序，力争使程序设计工作的效率更高。
- 用高级语言设计出来的程序，需要经过编译程序先翻译成机器语言程序，才能在计算机的硬件系统上予以执行，个别的选用解释执行方案。
- 高级语言的程序通用性强，在不同型号的计算机之间更容易移植。对高级语言进行编译、汇编后得到机器语言在计算机上运行。

汇编语言以及机器语言

- 汇编语言是对计算机机器语言进行符号化处理的结果,再增加一些为方便程序设计而实现的扩展功能。
- 在汇编语言中,可以用英文单词或其缩写替代二进制的指令代码,更容易记忆和理解;还可以选用英文单词来表示程序中的数据(常量、变量和语句标号),使程序员不必亲自为这些数据分配存储单元,而是留给汇编程序去处理,达到基本可用标准。
- 若在此基础上,能够在支持程序的不同结构特性(如循环和重复执行结构,子程序所用哑变元替换为真实参数)等方面提供必要的支持,使该汇编语言的实用程度更高。
- 汇编程序要经过汇编器翻译成机器语言后方可运行
- 机器语言是计算机硬件能直接识别和运行的指令的集合,是二进制码组成的指令,用机器语言设计程序基本不可行。
- 程序的最小单元是指令,同时,指令也是计算机硬件执行程序的最小单位。

Levels of Representation



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

lw t0, t2, 0
lw t1, t2, 4
sw t1, t2, 0
sw t0, t2, 4

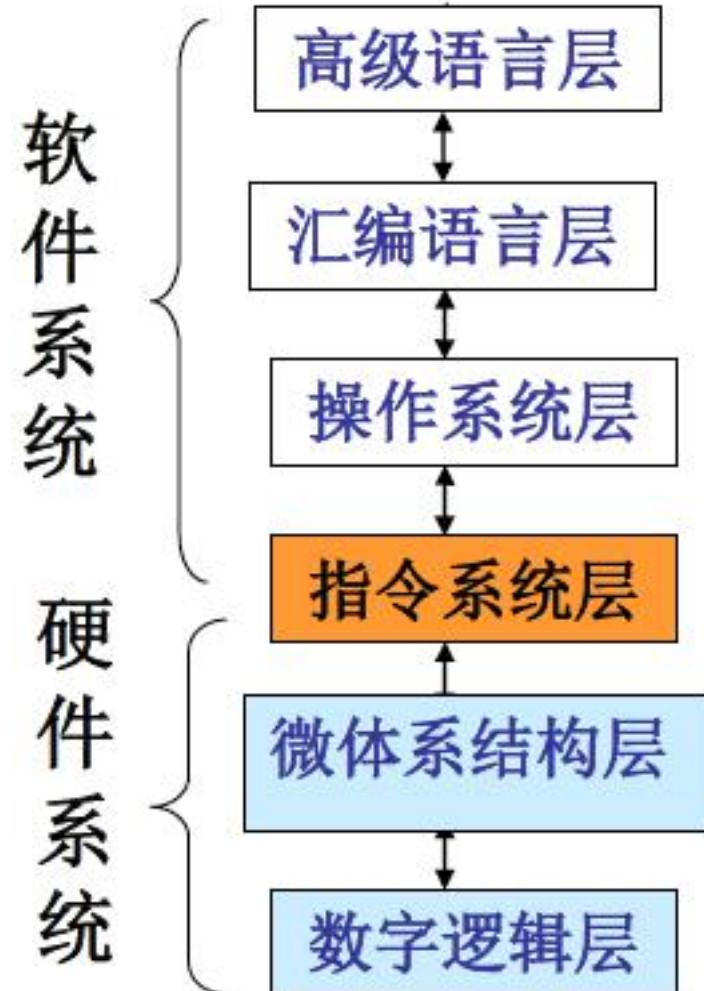
0000	1001	1100	0110	1010	1111	0101	1000
1010	1111	0101	1000	0000	1001	1100	0110
1100	0110	1010	1111	0101	1000	0000	1001
0101	1000	0000	1001	1100	0110	1010	1111

指令和指令系统

- 计算机系统由硬件和软件两大部分组成。硬件指由中央处理器、存储器以及外围设备等组成 的实际装置。软件是为了使用计算机而编写的各种系统的和用户的程序，程序由一个序列的计算机指令组成。
- 指令是计算机运行的最小的功能单元，是指挥计算机硬件运行的命令，是由多个二进制位组成的位串，是计算机硬件可以直接识别和执行的信息体。指令中应指明指令所完成的操作，并明确操作对象。
- 一台计算机提供的全部指令构成该计算机的指令系统。指令用于程序设计人员告知计算机执行一个最基本运算、处理功能，多条指令可以组成一个程序，完成一项预期的任务。

指令系统地位

- 可以从6个层次分析和看待计算机系统的基本组成。
- 指令系统层处在硬件系统和软件系统之间，是硬、软件之间的接口部分，对两部分都有重要影响。
- 硬件系统用于实现每条指令的功能，解决指令之间的衔接关系；
- 软件由按一定规则组织起来的许多条指令组成，完成一定的数据运算或者事务处理功能。
- 指令系统优劣是一个计算机系统是否成功的关键因素。



计算机系统的层次结构

指令的功能分类

□ 数据运算指令

- 算数运算，逻辑运算

□ 数据传输指令

- 内存/寄存器，寄存器/寄存器

□ 控制指令

- 无条件跳转，条件跳转，子程序的支持（调用和返回）

□ 输入输出指令

- 与输入输出端口的数据传输（输入输出模型如何）

□ 其它指令

- 停机、开/关中断、空操作、特权指令、设置条件码

指令格式

□ 指令格式：操作码，操作数地址的二进制分配方案



- 操作码：指令的操作功能
- 操作数地址：操作数存放的地址，或者操作数本身

□ 指令字：完整的一条指令的二进制表示

□ 指令字长：指令字中二进制代码的位数

- 机器字长：计算机能够直接处理的二进制数据的位数
- 指令字长（字节倍数）：0.5, 1, 2,个机器字长
- 定长指令字结构 变长指令字结构
- 定长操作码 扩展操作码

寻址方式

- 如何找寄存器？
- 如何找立即数？
- 如何找内存地址？
 - 直接给出内存地址
 - 通过寄存器进行偏移找出地址
- 如何找输入输出的端口地址？

寻址方式

- 寻址方式（又称编址方式）指的是确定本条指令的操作数地址及下一条要执行的指令地址的方法。
- 不同的计算机系统,使用数目和功能不同的寻址方式，其实现的复杂程度和运行性能各不相同。有的计算机寻址方式较少，而有些计算机采用多种寻址方式。
- 通常需要在指令中为每一个操作数专设一个地址字段，用来表示数据的来源或去向的地址。在指令中给出的操作数（或指令）的地址被称为形式地址，使用形式地址信息并按一定规则计算出来或读操作得到的一个数值才是数据（或指令）的实际地址。在指令的操作数地址字段，可能要指出：
 - ①运算器中的累加器的编号或专用寄存器名称（编号）
 - ②输入/输出指令中用到的I/O 设备的入出端口地址
 - ③内存储器的一个存储单元（或一I/O设备）的地址

评价计算机性能的指标

□ 吞吐率

- 单位时间内完成的任务数量

□ 响应时间

- 完成任务的时间

□ 衡量性能的指标

- MIPS
- CPI
- CPU Time
- CPU Clock

□ 综合测试程序

指令系统分类

- 复杂指令集（CISC：Complex Instruction Set Computing）
 - 特点是指令数目多而复杂，每条指令字长并不相等，电脑必须加以判读，并为此付出了性能的代价。
 - 例子：x86指令集
- 精简指令集（RISC：Reduced Instruction Set Computing）
 - 特点是对指令数目和寻址方式都做了精简，指令长度规整，长度相同（压缩指令除外），使其实现更容易，指令并行执行程度更好，编译器的效率更高。
 - 例子：MIPS指令集，RISC-V指令集，ARM指令集
- 超长指令字（VLIW：Very Long Instruction Word）
 - 将简短而长度统一的精简指令组合出超长指令，每次运行一条超长指令，等于并发运行多条短指令。
 - 例子：Intel安腾指令集

RISC-V 指令系统

- RV32I - 基础整数指令、
 - 特权指令、
 - RV32F – 单精度浮点指令、
 - RV32D – 双精度浮点指令、
 - 扩展指令
- RV32
 - 面向32位处理器的指令系统
- RV64
 - 64位字长的RISC-V指令集

ThinPAD RISC-V指令系统

□ 采用与RV32I兼容的指令格式

- 32位固定字长
- 操作码位置及长度固定
- 寻址方式简单

□ 供设计有19+3+6+1条指令

- 19条是基础指令，可以用于执行基本的监控程序（不包括异常与中断，不包括虚拟地址）
- 3条是运行RV64监控程序需要的指令
- 6条是支持优先级，支持中断与异常的指令
- 1条是支持虚拟地址的指令（没有TLB的话可实现为Nop）
- 可根据需要进行扩展

□ 作为本课程教学实验的指令系统

RISC-V 指令格式（32位）

- 所有的指令都是32位字长，有 6 种指令格式：寄存器型，立即数型，存储型，分支指令、跳转指令和大立即数

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB / B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
UJ / J 型	Imm[20,10:1,11,19:12]			rd	opcode	
U 型	Imm[31:12]			rd	opcode	

ThinPAD RV32指令系统概况

所需要实现的基础指令19条，后续只统计这19条的情况

监控程序基础版本	ADD, ADDI, AND, ANDI, AUIPC, BEQ, BNE, JAL, JALR, LB, LUI, LW, OR, ORI, SB, SLLI, SRLI, SW, XOR
监控程序支持中断版本	基础版本所有指令+CSRRC, CSRRS, CSRRW, EBREAK, ECALL, MRET
监控程序支持TLB版本	上一行所有指令+SFENCE. VMA

ThinPAD RV32指令 (R型)

R型

funct7	rs2	rs1	funct3	rd	opcode
--------	-----	-----	--------	----	--------

0000000

rs2

rs1

000

rd

0110011

add

$$\text{reg}[rd] = \text{reg}[rs1] + \text{reg}[rs2]$$
$$PC = PC + 4$$

0000000

rs2

rs1

111

rd

0110011

and

$$\text{reg}[rd] = \text{reg}[rs1] \& \text{reg}[rs2]$$
$$PC = PC + 4$$

0000000

rs2

rs1

110

rd

0110011

or

$$\text{reg}[rd] = \text{reg}[rs1] | \text{reg}[rs2]$$
$$PC = PC + 4$$

0000000

rs2

rs1

100

rd

0110011

xor

$$\text{reg}[rd] = \text{reg}[rs1] ^ \text{reg}[rs2]$$
$$PC = PC + 4$$

ThinPAD RV32指令 (I型-1)

I型

imm[11:0]	rs1	funct3	rd	opcode
-----------	-----	--------	----	--------

imm[11:0]	rs1	000	rd	0010011	addi
-----------	-----	-----	----	---------	------

$$\begin{aligned} \text{reg}[rd] &= \text{reg}[rs1] + \text{sign_ext}(imm[11:0]) \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

imm[11:0]	rs1	111	rd	0010011	andi
-----------	-----	-----	----	---------	------

$$\begin{aligned} \text{reg}[rd] &= \text{reg}[rs1] \& \text{ sign_ext}(imm[11:0]) \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

imm[11:0]	rs1	110	rd	0010011	ori
-----------	-----	-----	----	---------	-----

$$\begin{aligned} \text{reg}[rd] &= \text{reg}[rs1] | \text{sign_ext}(imm[11:0]) \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

0000000	shamt	rs1	001	rd	0010011	slli
---------	-------	-----	-----	----	---------	------

$$\begin{aligned} \text{shamt} &= \text{inst}[24:20] \\ \text{reg}[rd] &= \text{reg}[rs1] << \text{shamt} \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

0000000	shamt	rs1	101	rd	0010011	srli
---------	-------	-----	-----	----	---------	------

$$\begin{aligned} \text{shamt} &= \text{inst}[24:20] \\ \text{reg}[rd] &= \text{reg}[rs1] >> \text{shamt} \\ \text{PC} &= \text{PC} + 4 \end{aligned}$$

ThinPAD RV32指令 (I型-2)

I型

imm[11:0]	rs1	funct3	rd	opcode
-----------	-----	--------	----	--------

imm[11:0]	rs1	000	rd	1100111	jalr
-----------	-----	-----	----	---------	------

$$\text{Imml} = \text{sign_ext}(\text{imm}[11:0])$$

$$\text{Reg}[rd] \leftarrow \text{pc} + 4$$

$$\text{Pc} \leftarrow \{\text{reg}[rs1] + \text{imml}\}[31:1], 1'b0\}$$

imm[11:0]	rs1	000	rd	0000011	lb
-----------	-----	-----	----	---------	----

$$\text{reg}[rd] \leq \text{sign_ext}(\text{mem}[\text{reg}[rs1] + \text{imml}][7:0])$$

$$\text{pc} = \text{pc} + 4$$

imm[11:0]	rs1	010	rd	0000011	lw
-----------	-----	-----	----	---------	----

$$\text{reg}[rd] \leftarrow \text{mem}[\text{reg}[rs1] + \text{imml}]$$

$$\text{pc} = \text{pc} + 4$$

ThinPAD RV32指令 (S型)

S型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
----	-----------	-----	-----	--------	----------	--------

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
-----------	-----	-----	-----	----------	---------	----

mem[reg[rs1] + sign_ext(imm[11:0])] <= reg[rs2] [7:0] byte store
pc = pc + 4

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
-----------	-----	-----	-----	----------	---------	----

mem[reg[rs1] + sign_ext(imm[11:0])] <= reg[rs2]
pc = pc + 4

ThinPAD RV32指令 (SB/B型)

SB / B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
----------	--------------	-----	-----	--------	-------------	--------

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	beq
--------------	-----	-----	-----	---------------	---------	-----

branch = (reg[rs1] == reg[rs2])
immB = sign_ext({imm[12:1],1'b0})
pc <= branch ? pc + immB : pc + 4

imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	bne
--------------	-----	-----	-----	---------------	---------	-----

branch = ~(reg[rs1] == reg[rs2])
immB = sign_ext({imm[12:1],1'b0})
pc <= branch ? pc + immB : pc + 4

ThinPAD RV32指令 (UJ/J型)

UJ / J 型

Imm[20,10:1,11,19:12]	rd	opcode
-----------------------	----	--------

imm[20 10:1 11 19:12]	rd	1101111	jal
-----------------------------	----	---------	-----

immJ = sign_ext(imm[20:1], 1'b0)

reg[rd] <- pc + 4

pc <- pc+immJ

ThinPAD RV32指令 (U型)

U型

Imm[31:12]	rd	opcode
------------	----	--------

Imm[31:12]

rd

opcode

auipc

$$\text{immU} = \text{inst}[31:12]$$

$$\text{Reg}[rd] \leq \{\text{immU}, 000000000000\} + \text{pc}$$

$$\text{Pc} = \text{pc} + 4$$

imm[31:12]

rd

0110111

lui

$$\text{immU} = \text{inst}[31:12]$$

$$\text{reg}[rd] \leq \{\text{immU}, 000000000000\}$$

$$\text{pc} = \text{pc} + 4$$

程序举例(sum.s)

```
int sum()
{
    int sum = 0;
    for (int i = 1; i < 11; i++)
        sum += i;
    return sum
}
```

-Og版本

sum:

```
    li    a5, 1
    li    a0, 0
.L2:
    li    a4, 10
    bgt   a5, a4, .L4
    add   a0, a0, a5
    addi  a5, a5, 1
    j     .L2
.L4:
    ret
```

-O2版本

sum:

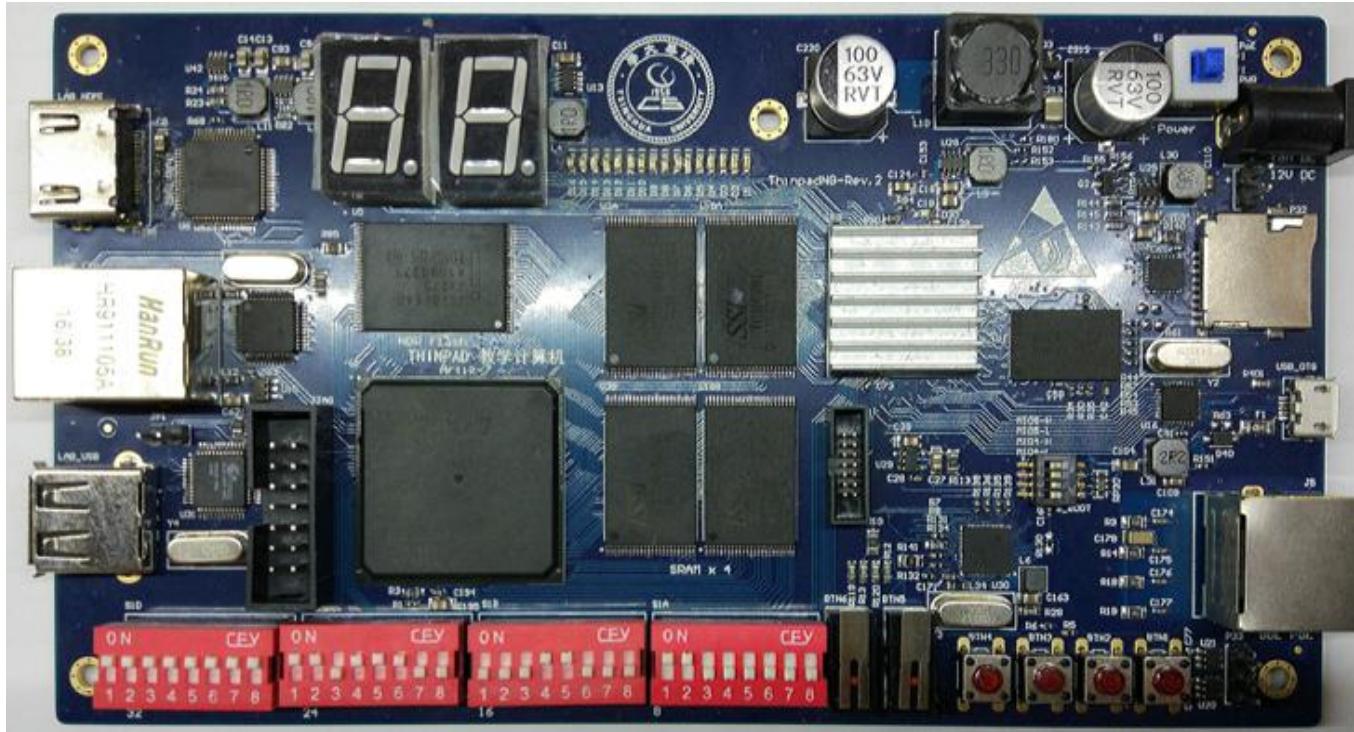
```
    li    a0, 55
    ret
```

程序举例(fib.s)

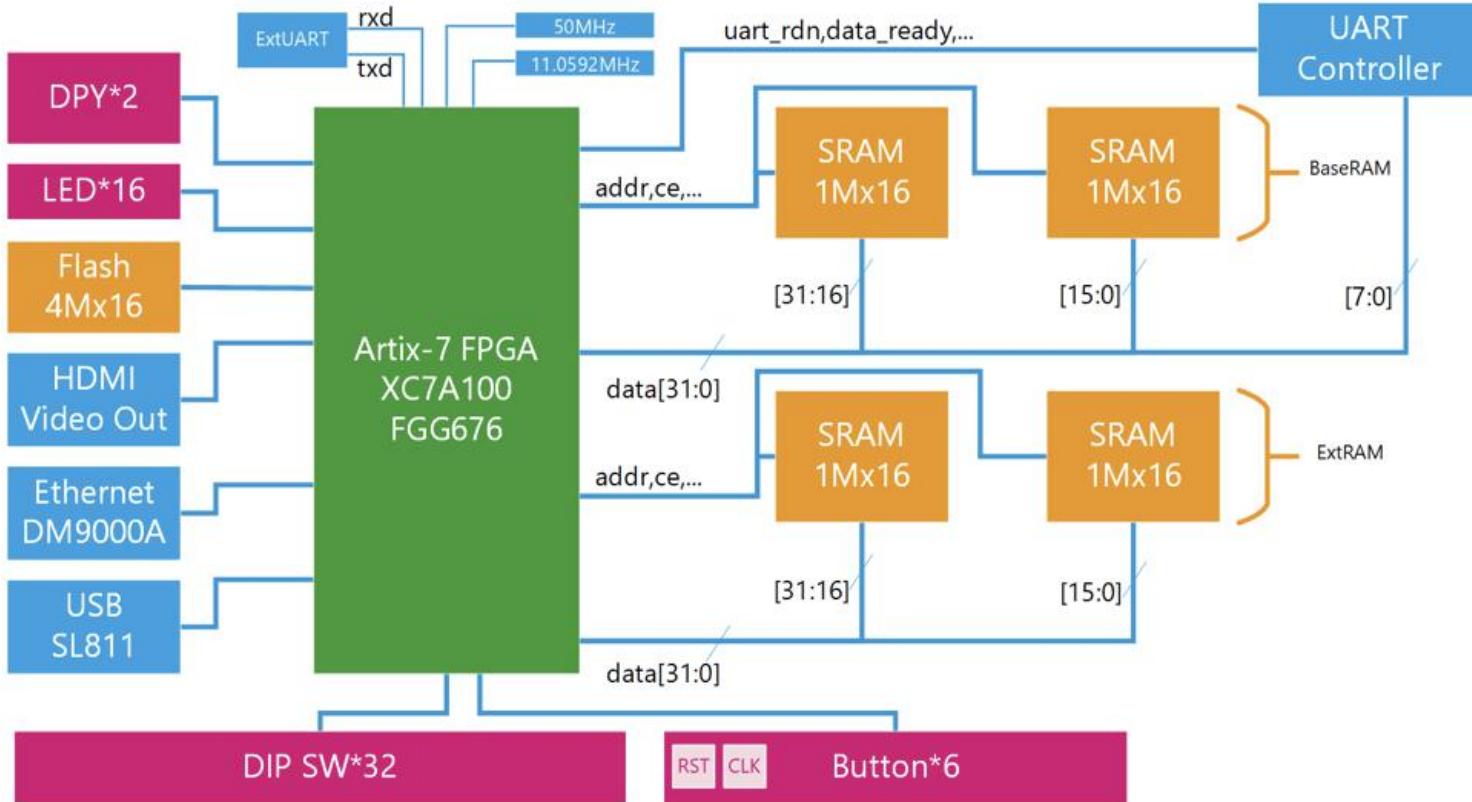
```
int fibo[10];                                .align 2
void fib()                                     .globl fib
{                                                 .type fib, @function
    int i;                                         fib:
    fibo[0]=1; fibo[1] =1;
    for ( i=2 ; i<10; i++)
        fibo[i]= fibo[i-1] + fibo[i-2];
}                                                 lui  a5,%hi(fibo)          # high value of fibo in a5
                                                addi a4,a5,%lo(fibo)  # set a4 = address of fibo
                                                li   a3,1              # a3 = 1
                                                addi a5,a5,%lo(fibo)  # set a5 = address of fibo
                                                sw   a3,0(a4)          # fibo[0] = 1
                                                sw   a3,4(a4)          # fibo[1] = 1
                                                addi a2,a5,32         # a2 = address of fibo[8]
                                                li   a4,1              # a4 = 1
                                                j    .L3               # goto L3
.L5:                                            lw   a4,4(a5)          # a4 = a5[1]
                                                lw   a3,0(a5)          # a3 = a5[0]
.L3:                                            add  a4,a4,a3          # a4 = a4 + a3
                                                sw   a4,8(a5)          # a5[2] = a4
                                                addi a5,a5,4
# a5 += 4 (points to the next entry)
                                                bne a5,a2,.L5          # not finished? goto L5
                                                ret
```

ThinPAD的硬件组成

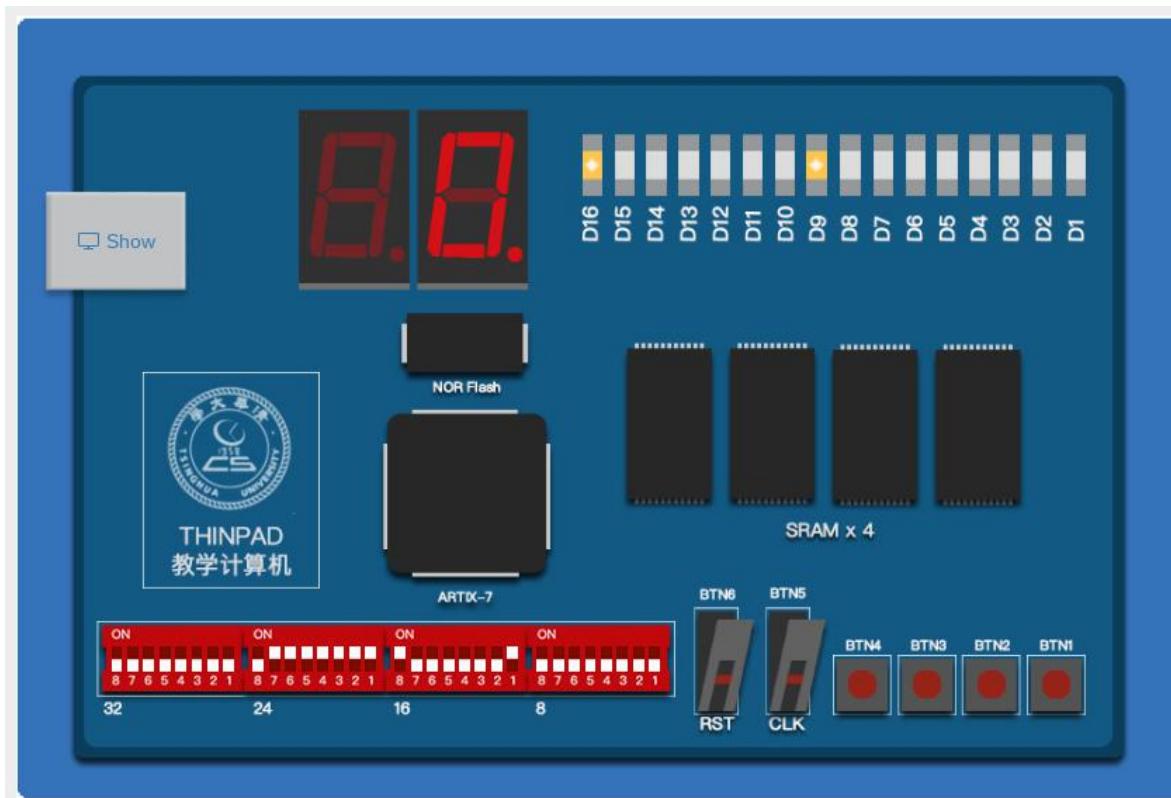
- 主要有一块FPGA，四块SRAM，串口以及其它的一些外围接口电路组成



硬件平台的内部电路构成



网络控制界面



监控程序的地址空间划分

虚地址区间	说明
0x80000000-0x800FFFFF	Kernel代码空间
0x80100000-0x803FFFFF	用户代码空间
0x80400000-0x807EFFFF	用户数据空间
0x807F0000-0x807FFFFF	Kernel数据空间
0x10000000-0x10000008	串口数据及状态

串口寄存器位定义

地址	位	说明
0x10000000 (数据寄存器)	[7:0]	串口数据, 读、写地址 分别表示串口接收、发送 一个字节
0x10000005 (状态寄存器)	[5]	状态位, 只读, 为1时 表示串口空闲, 可发送数据
0x10000005 (状态寄存器)	[0]	状态位, 只读, 为1时 表示串口收到数据

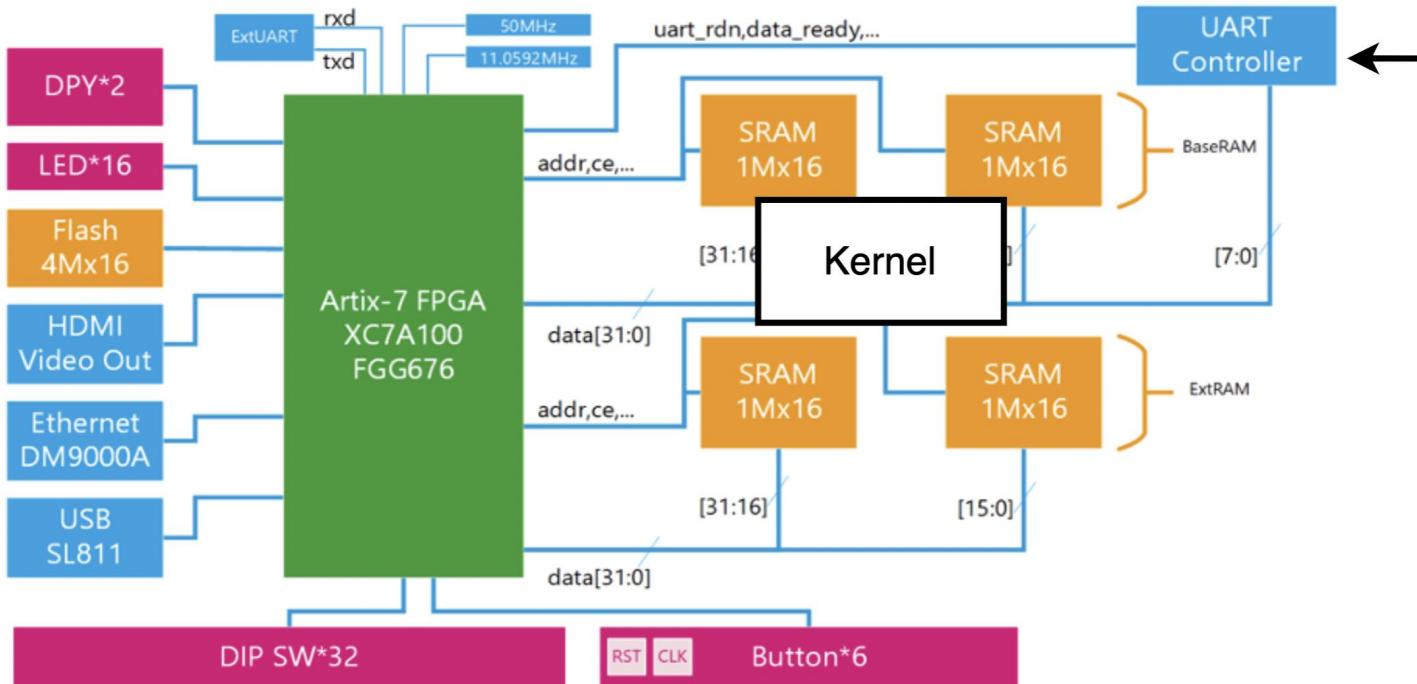
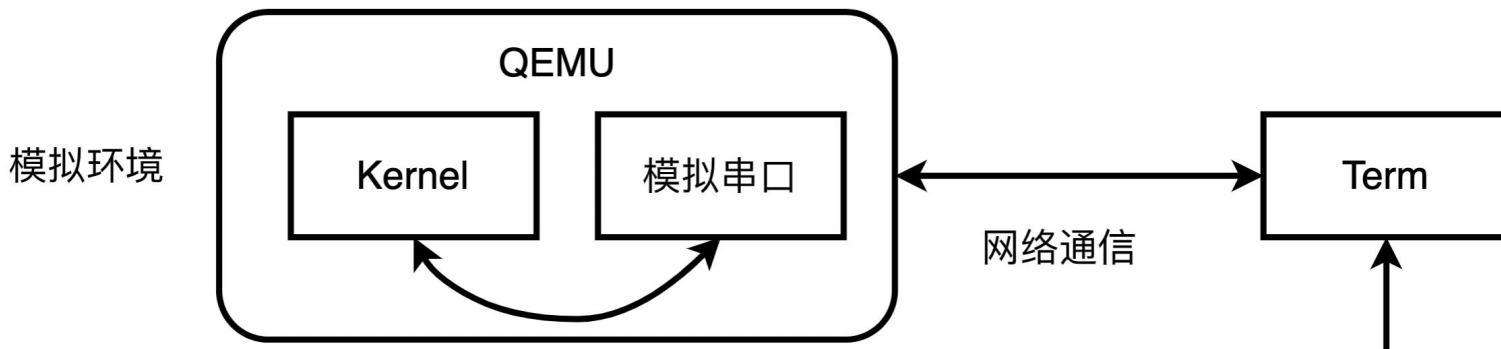
MMIO: Memory Mapped IO

- 输入输出可以通过特殊的端口进行指令操作（例如x86下面的in/out）
- 或者，将输入输出端口直接映射到内存单元，对于特殊内存地址的读写就完成了对于外设的读写
- 外设通过外设寄存器与主机进行信息交互，外设寄存器就表现为内存的单元，或者端口上的数据读写单元

模拟器

- 模拟器可以模拟ThinPAD硬件的执行，通过QEMU来模拟RISC-V处理器
- 监控程序可以直接运行在模拟器上
- 模拟器可以运行在Windows, Linux和Mac三个平台
- 不建议在Mac下面运行，因为后续的实验环境Vivado只能在Linux和Windows下面执行
 - 在Mac下面建议安装虚拟机，使用VirtualBox就行

模拟器与真实环境



程序设计举例（写入串口）

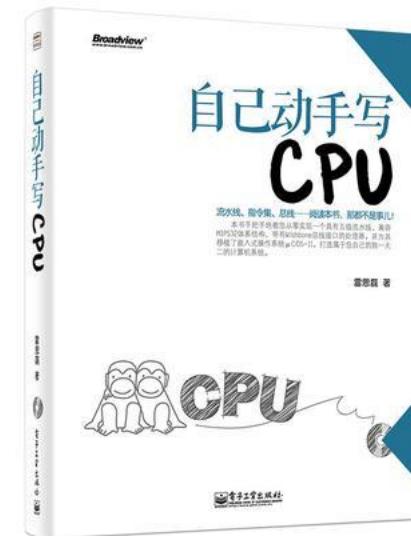
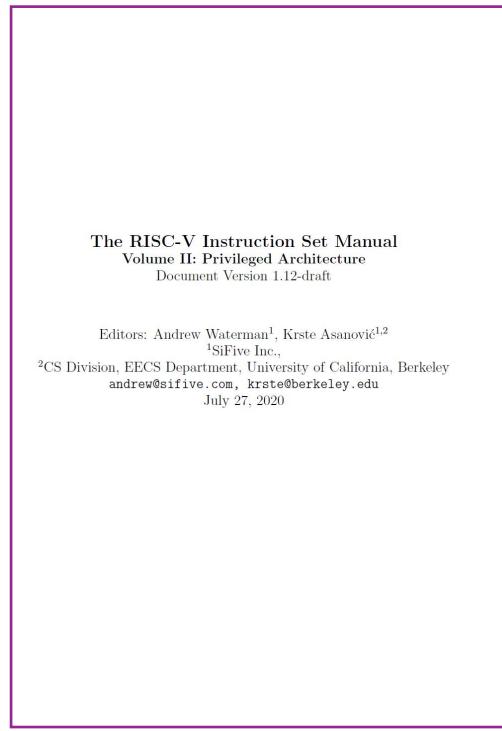
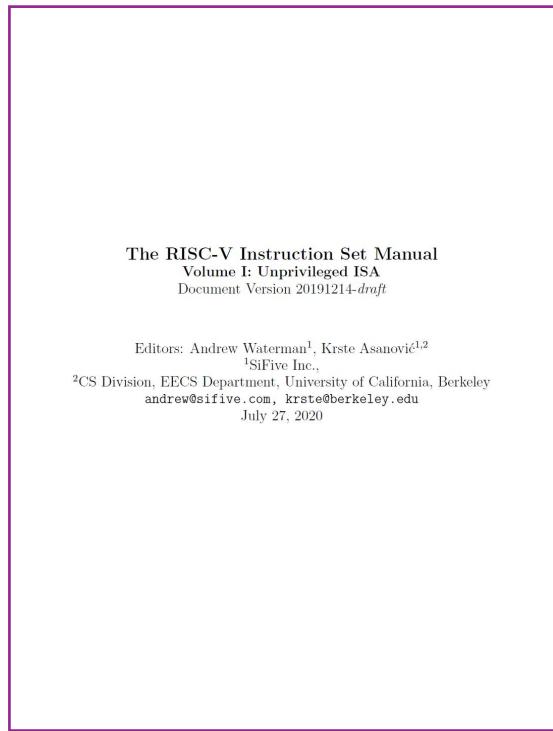
```
WRITE_SERIAL:                                // 写串口：将a0的低八位写入串口
    li t0, COM1
.TESTW:
    lb t1, %lo(COM_LSR_OFFSET)(t0)      // 查看串口状态
    andi t1, t1, COM_LSR_THRE           // 截取写状态位
    bne t1, zero, .WSERIAL            // 状态位非零可写进入写
    j .TESTW                          // 检测验证，忙等待
.WSERIAL:
    sb a0, %lo(COM_THR_OFFSET)(t0)    // 写入寄存器a0中的值
    jr ra
```

程序设计举例（从串口读出）

```
READ_SERIAL:                                // 读串口：将读到的数据写入a0低八位
    li t0, COM1
.TESTR:
    lb t1, %lo(COM_LSR_OFFSET)(t0)
    andi t1, t1, COM_LSR_DR                // 截取读状态位
    bne t1, zero, .RSERIAL                 // 状态位非零可读进入读
    j .TESTR                               // 检测验证
.RSERIAL:
    lb a0, %lo(COM_RBR_OFFSET)(t0)
    jr ra
```

实验参考书

- 《自己动手写CPU》 详细给出了如何使用Verilog写一个MIPS CPU的例子，可以作为实验的基础
- RISC-V指令手册，用户层编程手册，系统编程手册



实验提示

- 监控程序在kern下， 命令行程序在term下
- 务必仔细阅读监控程序下的README.md， 这个文件非常重要， 务必仔细阅读， 包含了监控程序的指示， 以及需要实现的指令和格式
- 按照试验指导材料运行监控程序， 编写汇编代码， 熟悉监控程序

实验环境说明

- 所有的实验都可以云上进行，而不需要直接使用实验板子，本年度的实验不发实验板子
- 网络实验能够支持运行监控程序，ucore等，可以完成基本实验
- 后期做扩展实验的小组，有需要的提出申请，每一个小组可以借出一套实验装置，期末之前归还

小结

□ 计算机程序语言

- 高级语言
- 汇编语言
- 机器语言

□ 指令系统

- 硬件/软件接口
- 指令功能/指令格式

□ ThinPAD RISC-V指令系统

谢谢



数字逻辑 组合逻辑与时序逻辑

2022年秋

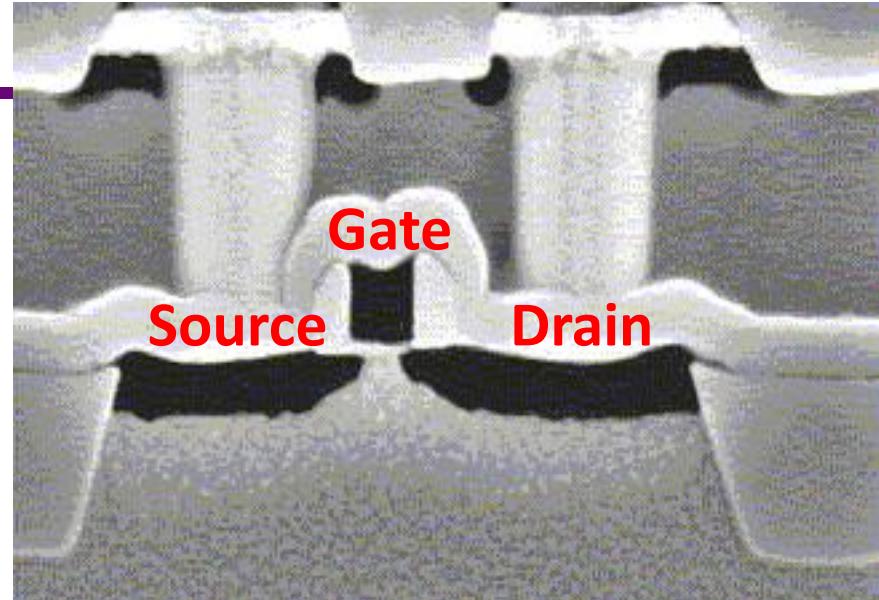
内容概要

- 门电路
 - 组合逻辑
 - 时序逻辑
-
- 大部分内容来自于Onur Mutlu教授（ETH）的上课内容

MOS晶体管

□ 通过组合

- 导体（金属 Metal）
- 绝缘体（氧化物 Oxide）
- 半导体



□ 得到一个晶体管 (MOS)。

- Gate: 栅极, Source: 源极, Drain: 漏极

□ 可以把许多这样的东西结合起来，实现简单的逻辑门

□ 组成原理所依赖的最底层器件

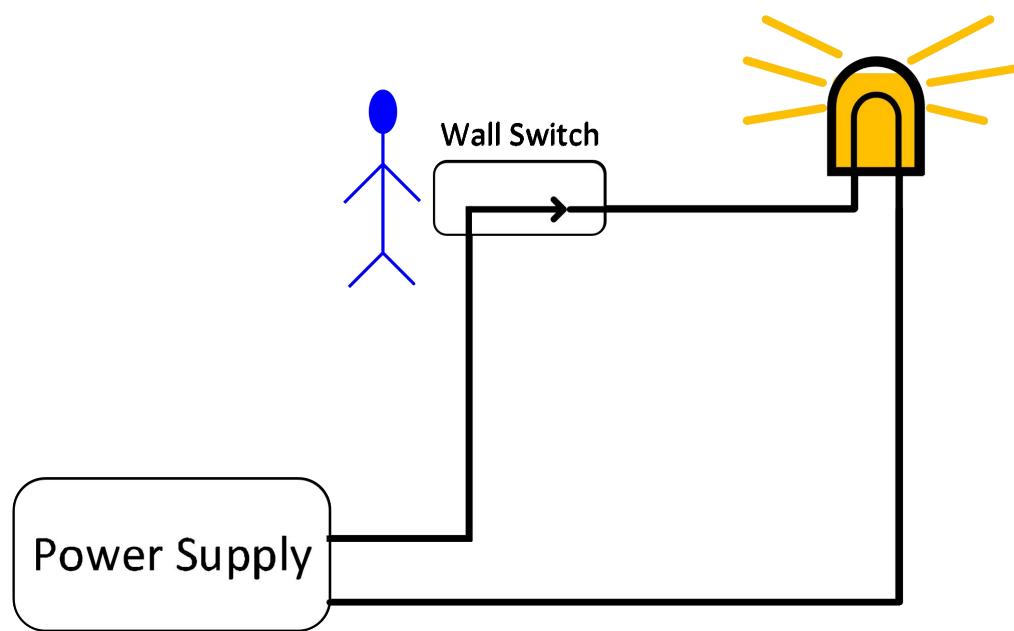
两种MOS管类型

- 有两种类型的MOS管: p-type, n-type

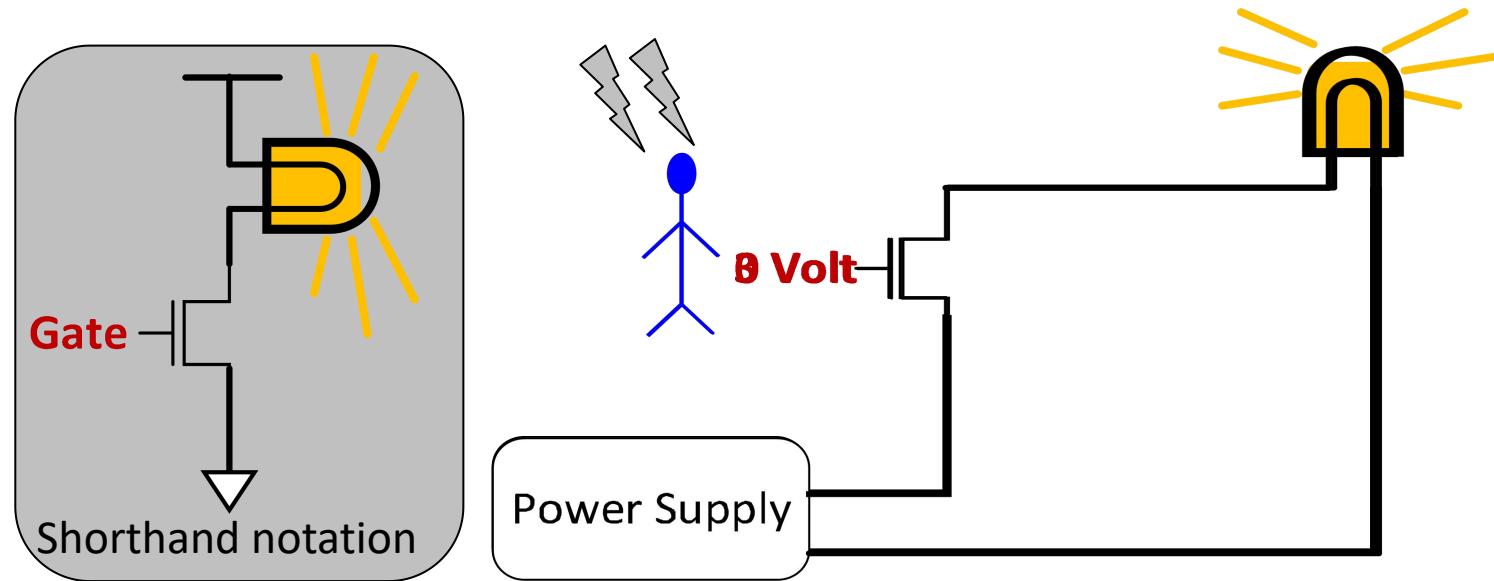


- n-type: Gate极加高电压则导通source（接电源正极）和drain（接电源负极）。无电压则不导通。
- p-type: 相反。Gate极加0电压，则导通source和drain。加高电压则不导通。

开关电路



使用MOS管的开关电路

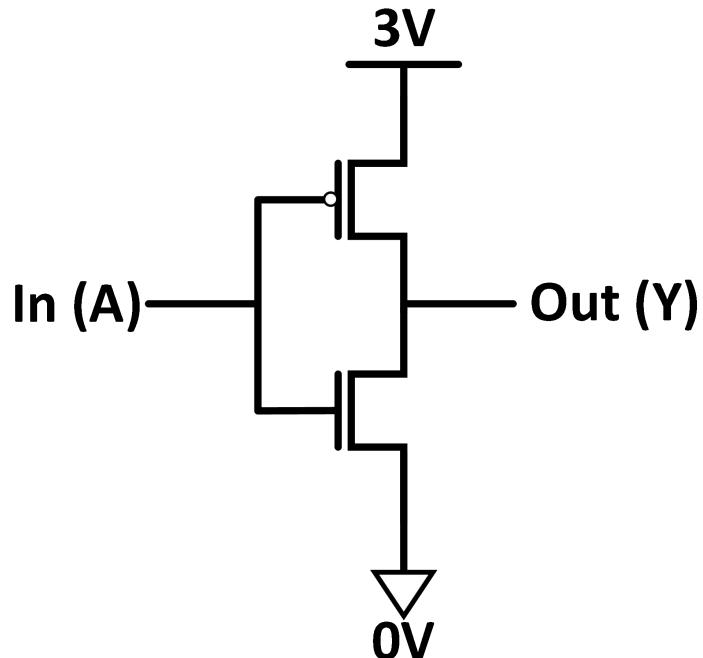


CMOS管

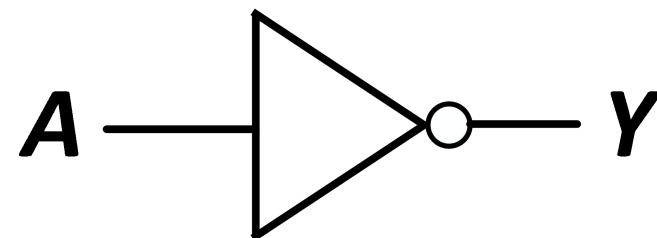
- 现代计算机同时使用n型和p型晶体管，即互补式 Complementary MOS (CMOS) 技术

$$\text{nMOS} + \text{pMOS} = \text{CMOS}$$

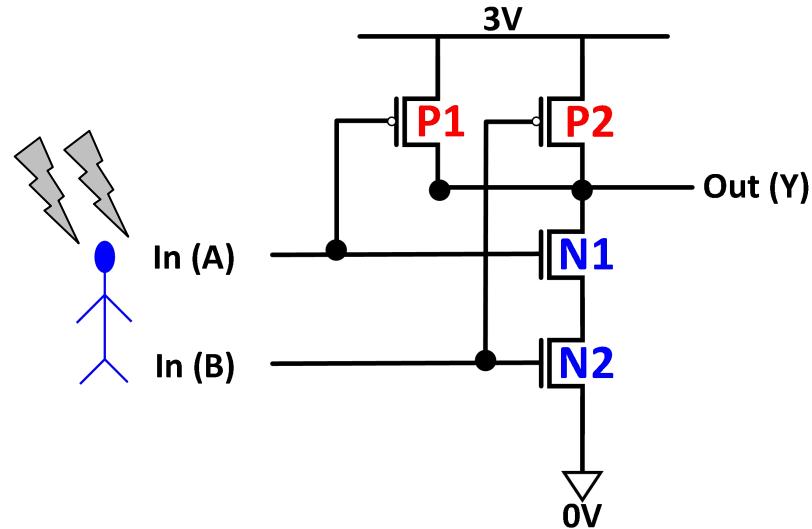
- 下面的电路起到什么作用？（这是个非门）



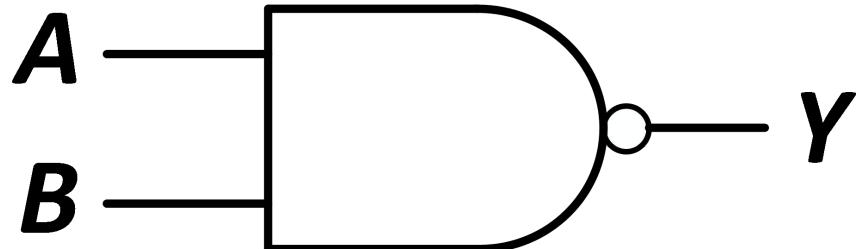
p型晶体管擅长拉高电压
n型晶体管擅长拉低电压



NAND门



$$Y = \overline{A \cdot B} = \overline{AB}$$

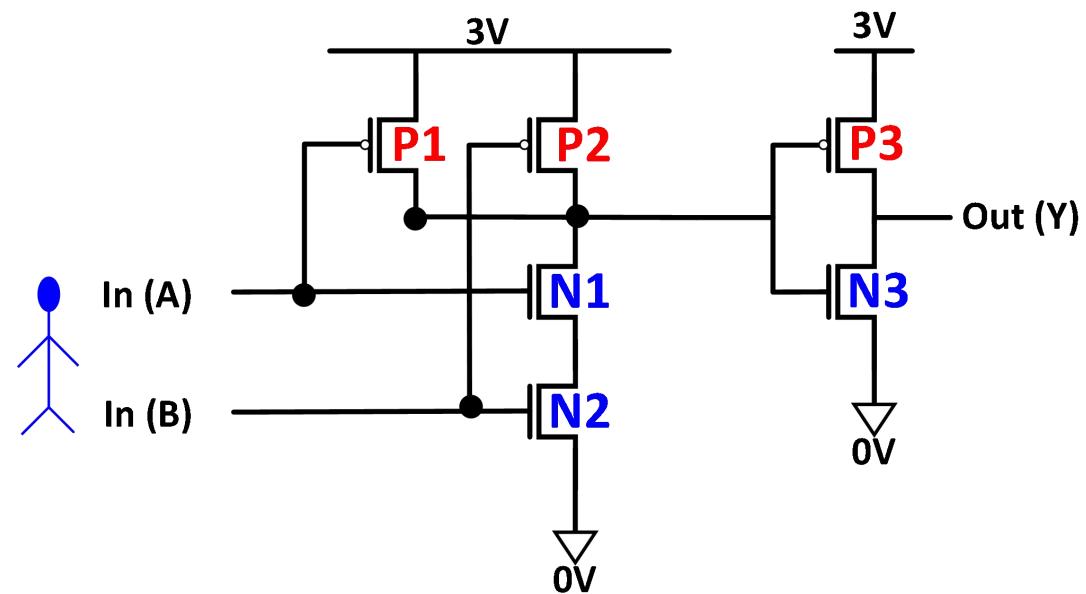
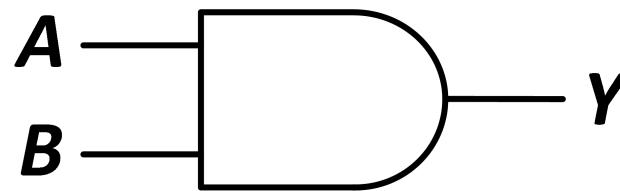


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

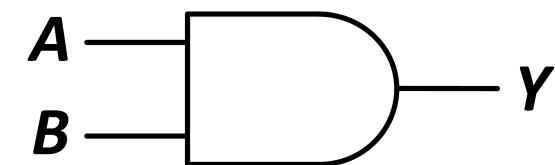
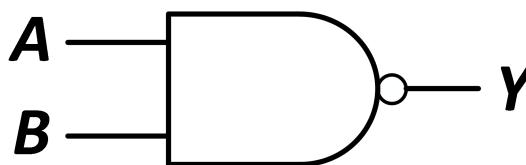
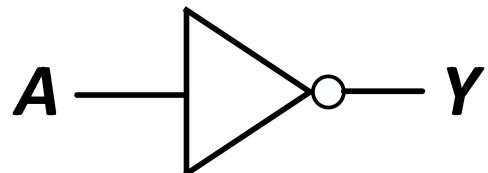
AND门

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$$Y = A \cdot B = AB$$



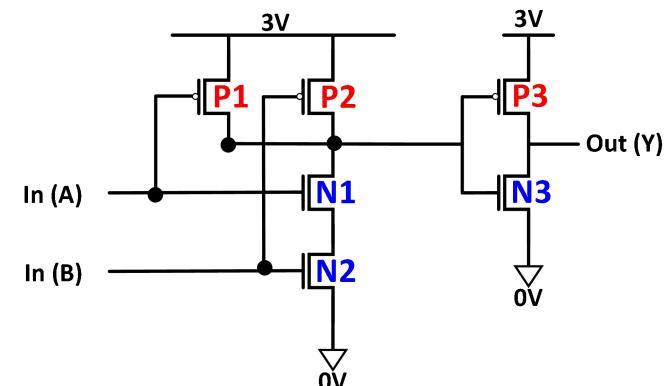
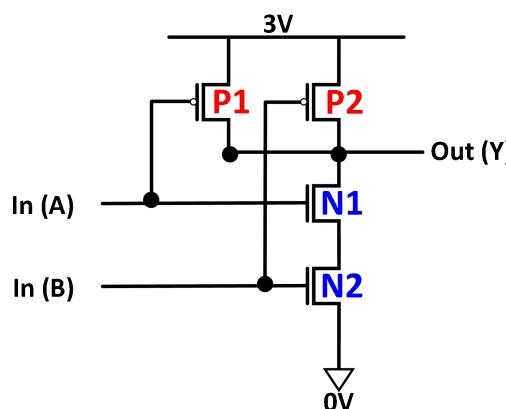
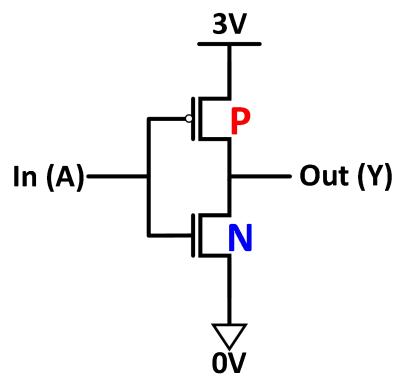
CMOS NOT, NAND, AND Gates (组合逻辑)



A	Y
0	1
1	0

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



三输入NAND

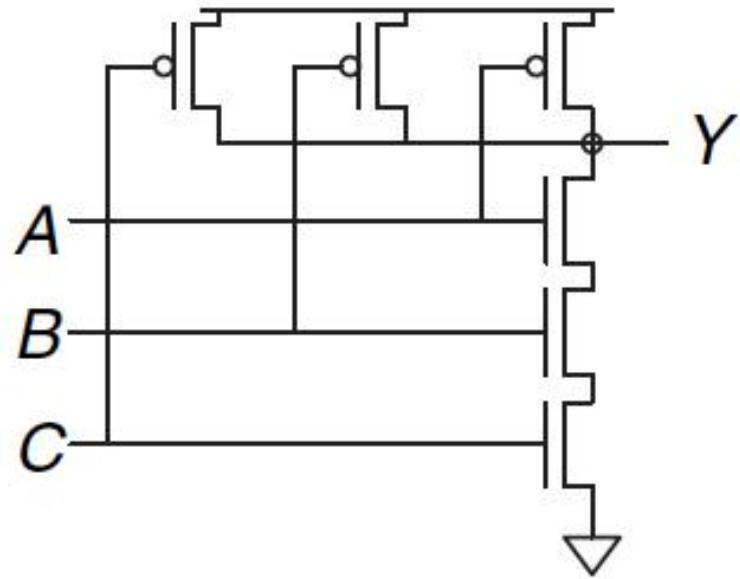


Figure 1.35 Three-input NAND
gate schematic

一般CMOS门结构 (1)

□ 用于构建任何反转逻辑门的一般形式，如：NOT, NAND, 或NOR

- 网络可以由串联或并联的晶体管组成。
- 当晶体管并联时，如果其中一个晶体管处于导通状态，网络就会接通。
- 当晶体管串联时，只有当所有晶体管都接通时，网络才会接通。

p型晶体管用来拉高电压
n型晶体管用来拉低电压
形成稳定输出

一般CMOS门结构（2）

□ 在任何时候都应该有一个网络处于开启状态，另一个网络处于关闭状态。

- 都导通的话就短路了，很可能是错误的操作
- 如果两个网络同时关闭，则输出是浮动的未定义。

使用这个结构的原因

- MOS晶体管不是完美的开关
 - pMOS晶体管能很好地输出1，但不能很好地输出0
 - nMOS晶体管能很好地输出0，但不能很好地输出1。
-
- pMOS晶体管擅长 "拉高 "输出端
 - nMOS晶体管擅长"拉低 "输出端

组合逻辑的延迟

- 延迟：从输入端变化开始，到输出端稳定输出更新之后的值所经历的时间
- 串联的速度比并联的速度慢
 - 线路上的电阻更大

功耗

□ 动态功耗

■ $C * V^2 * f$

■ C = 电路的电容（导线和门）

■ V = 电源电压

■ f = 电容充电频率

□ 静态功耗

■ $V * I_{leakage}$

■ 电源电压*漏电流

□ 能耗

■ Power * Time

逻辑电路

- 一个逻辑电路由以下部分组成。
 - 输入
 - 输出
- 功能说明（描述输入和输出之间的关系）
- 时序规范（描述输入变化和输出响应之间的延迟）

逻辑电路的类型

□ 组合逻辑

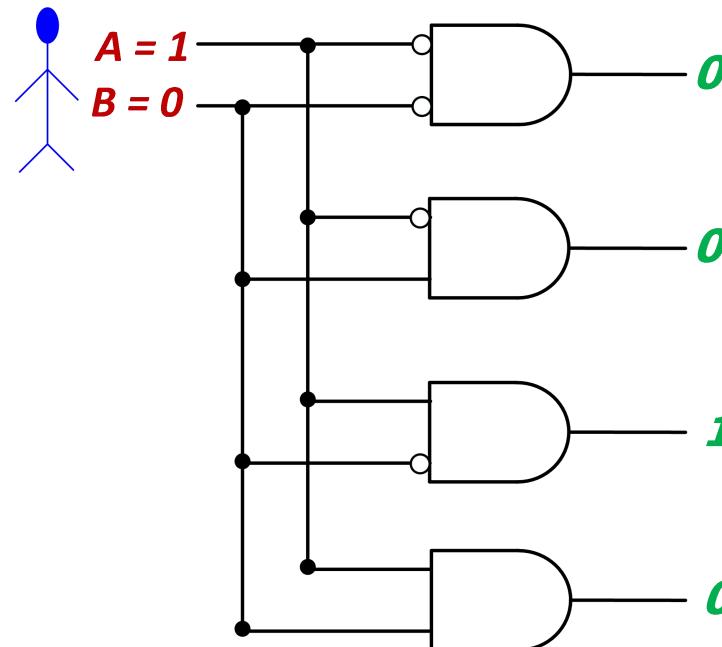
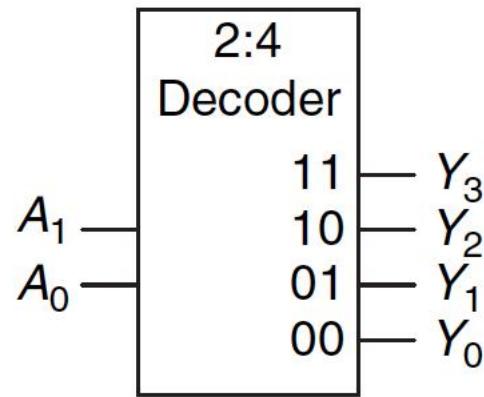
- 无记忆
- 输出严格依赖于现在应用于电路的输入值的组合

□ 时序逻辑

- 可以“存储”数据值，历史值
- 输出由之前（历史值）和当前的输入值决定

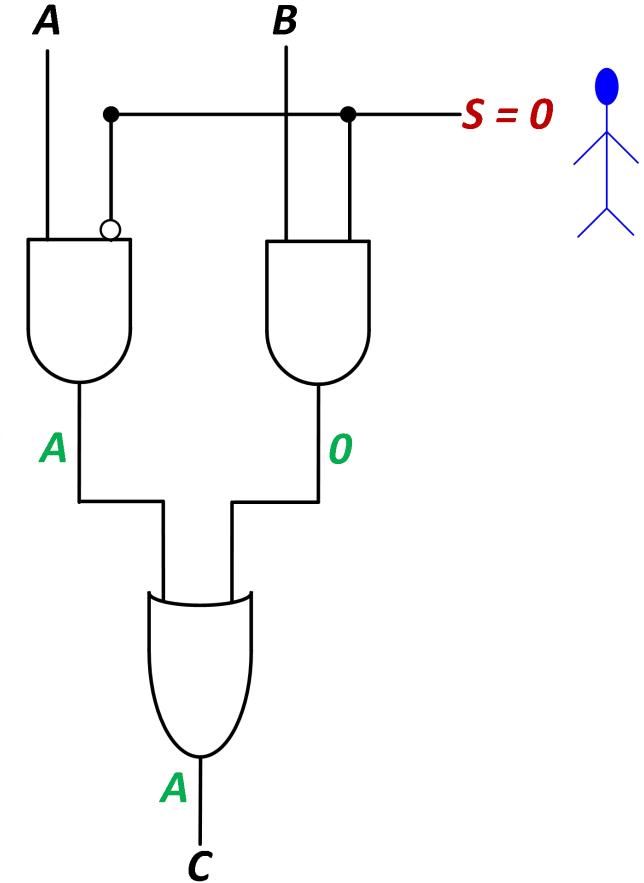
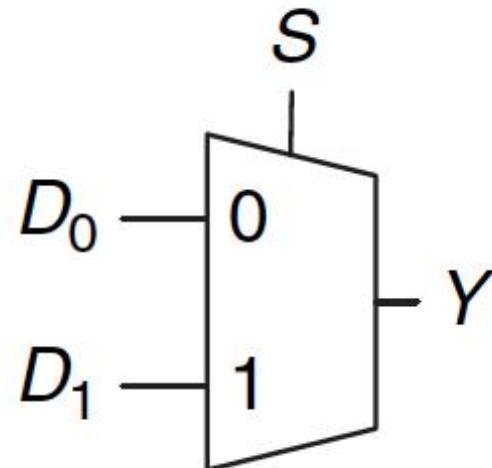
解码器

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

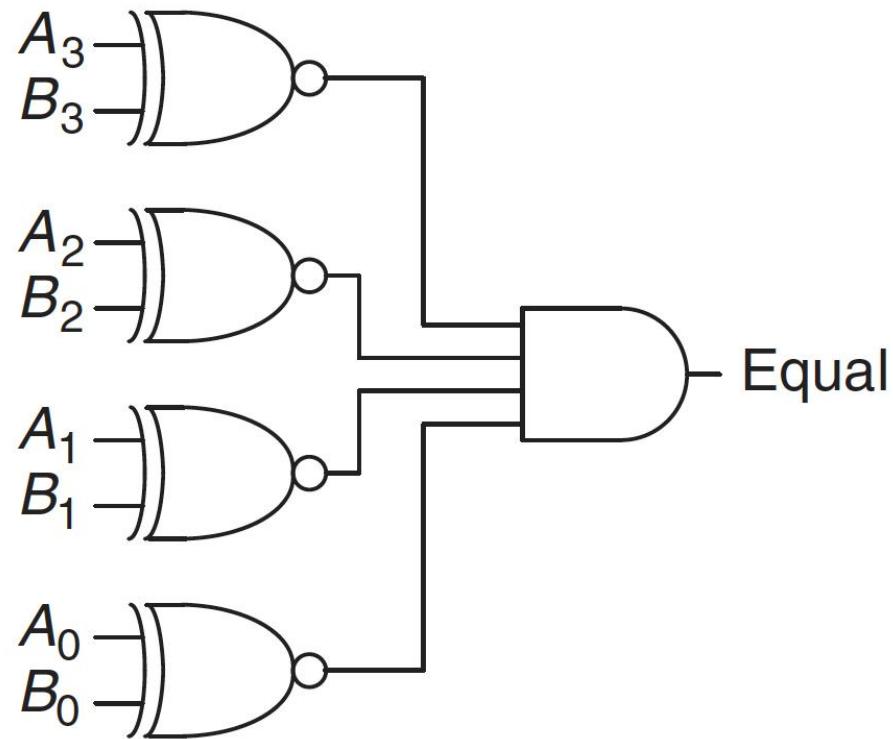
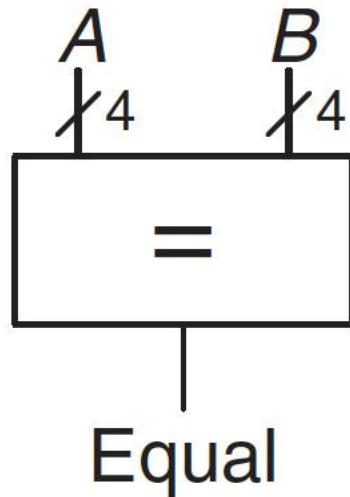


多路选择器

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

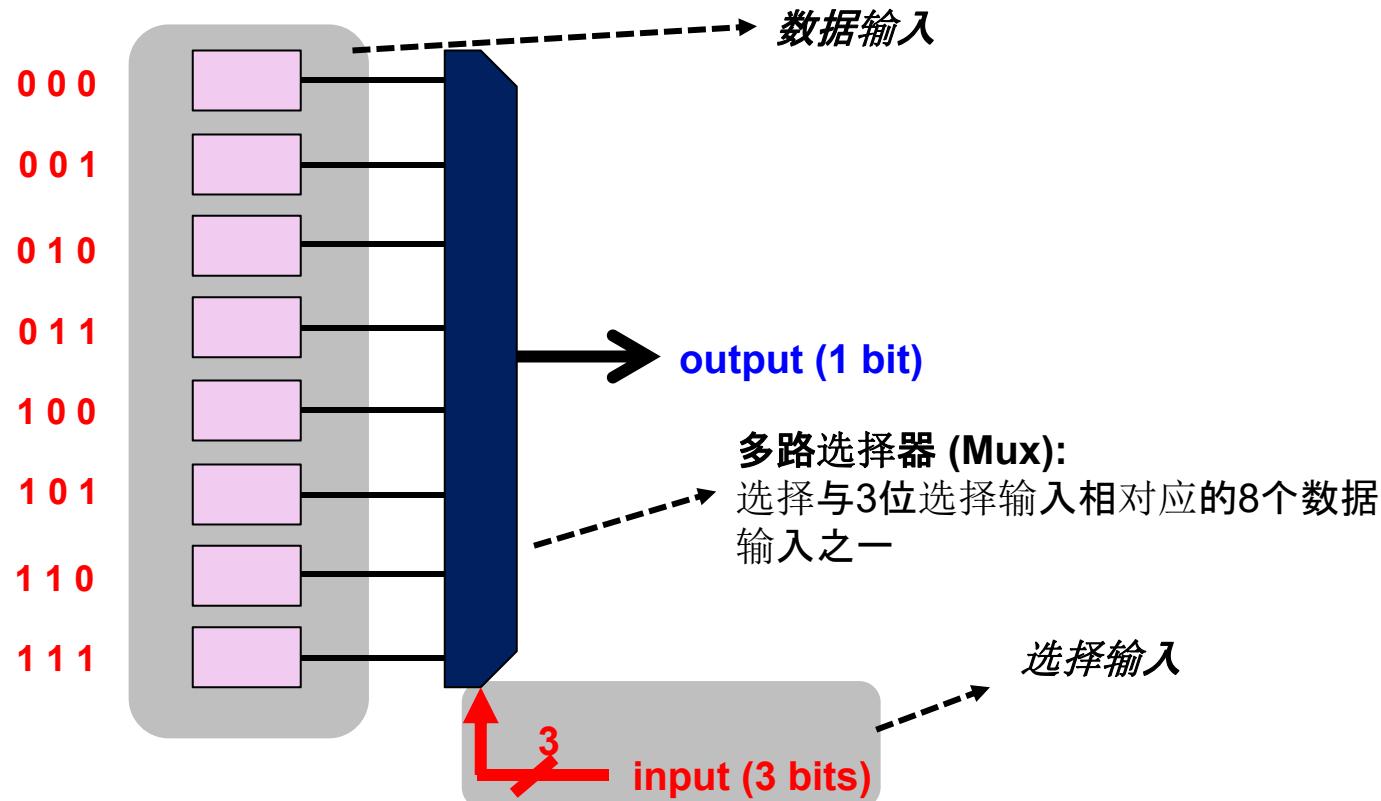


比较器



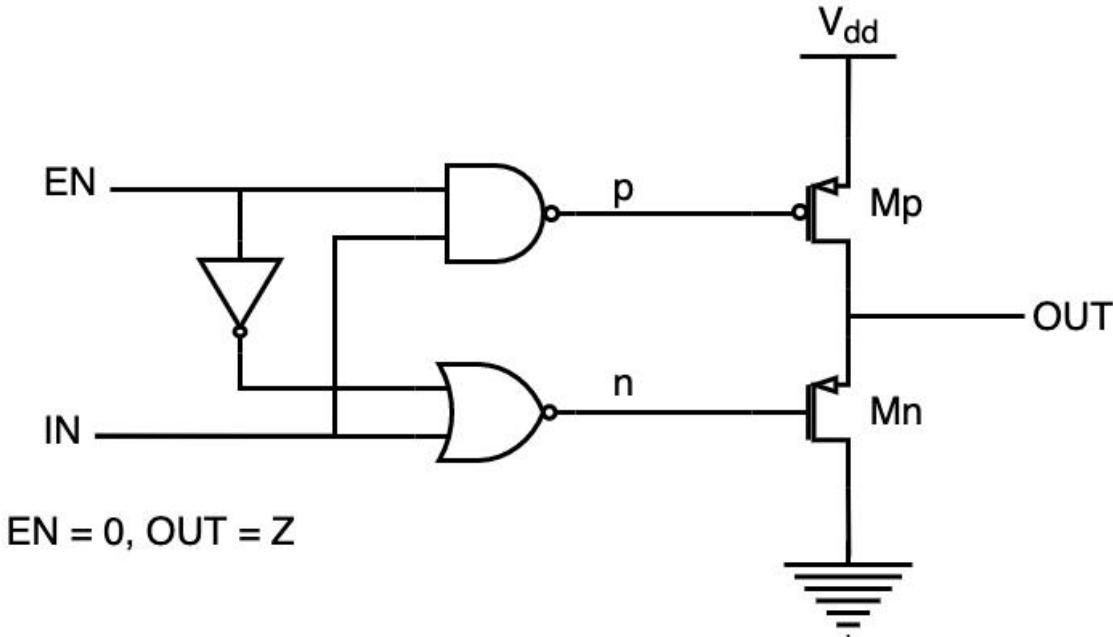
查找表LUT

□ 3-bit input LUT (3-LUT)



3-LUT可以实现任何3位输入逻辑函数

三态缓冲门



EN = 1, IN = 1, p = 0, Mp open, n = 0, Mn close, Out = 1

EN = 1, IN = 0, p = 1, Mp close, n = 1, Mn open, Out = 0

Out = IN

Tristate
Buffer

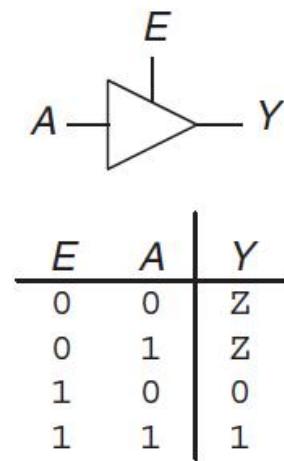
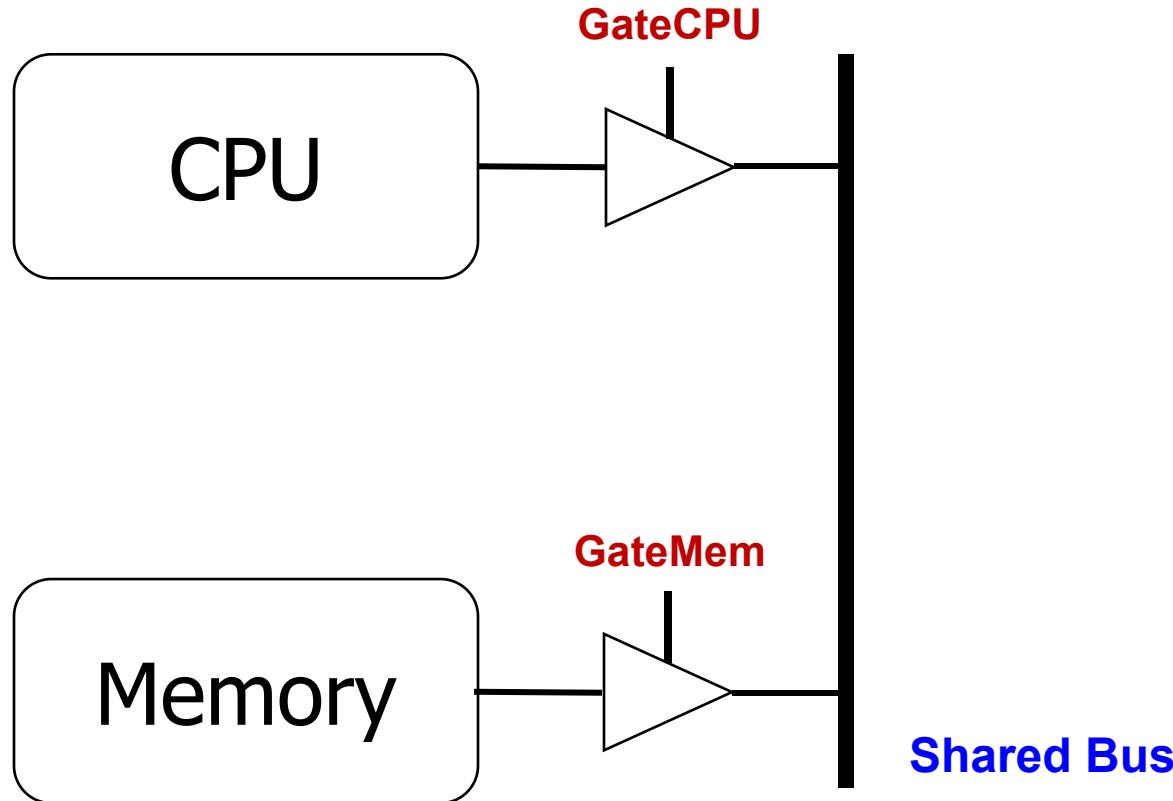


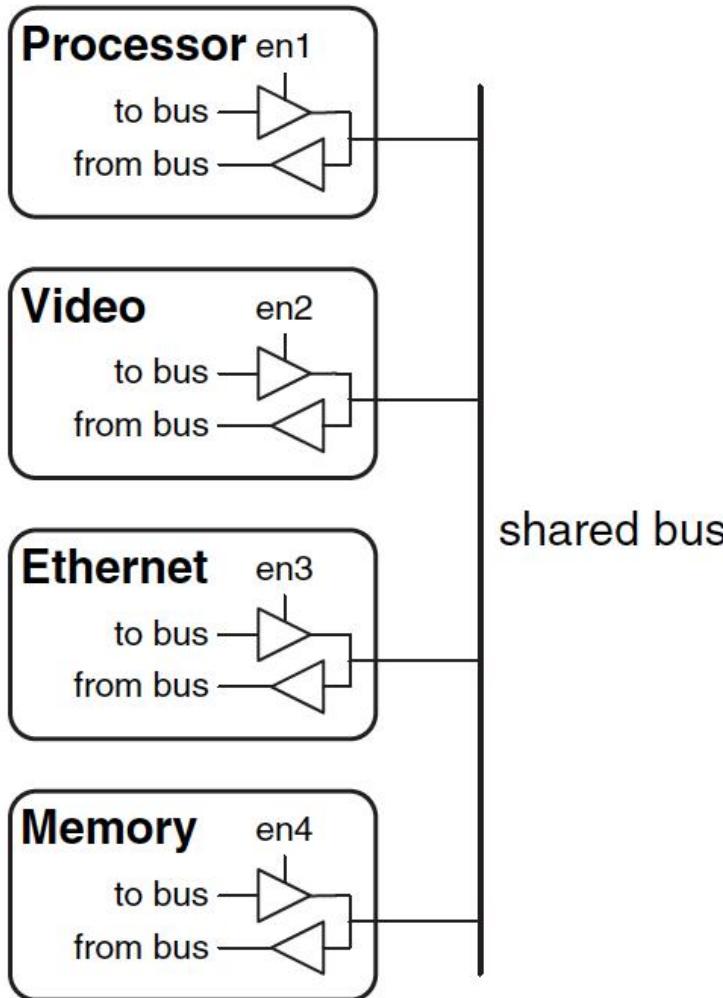
Figure 2.40 Tristate buffer

三态缓冲门的应用

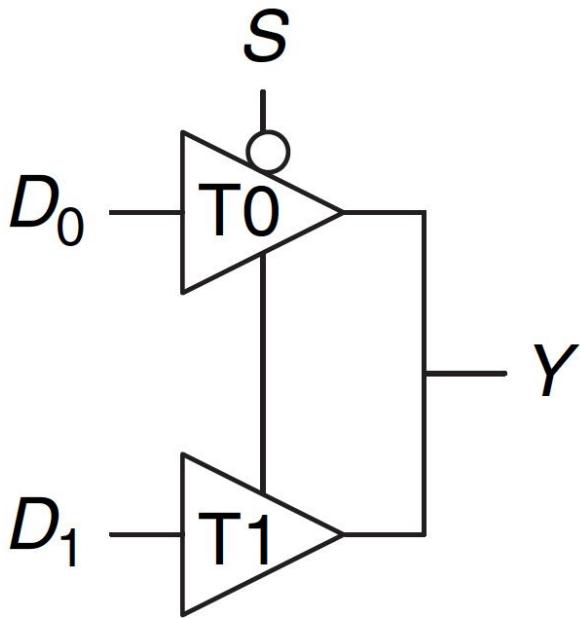
- 想象一条连接CPU和内存的导线
- 在任何时候，CPU或内存其中之一可以在导线上放置一个值，但不能同时放置。
- 使用两个三态逻辑缓冲器：一个由CPU驱动，另一个由内存驱动；并确保在任何时候最多只有一个被启用。



三态缓冲门与共享总线



使用三态缓冲门的多路选择器



$$Y = D_0 \bar{S} + D_1 S$$

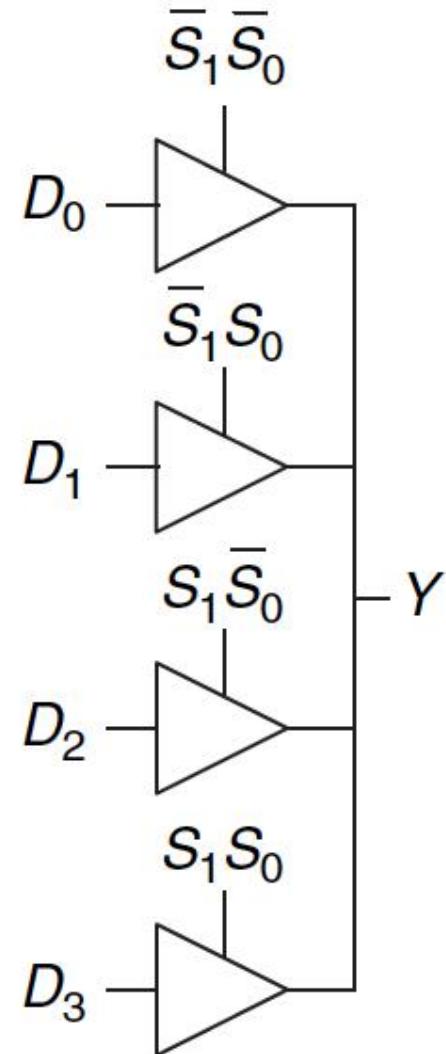
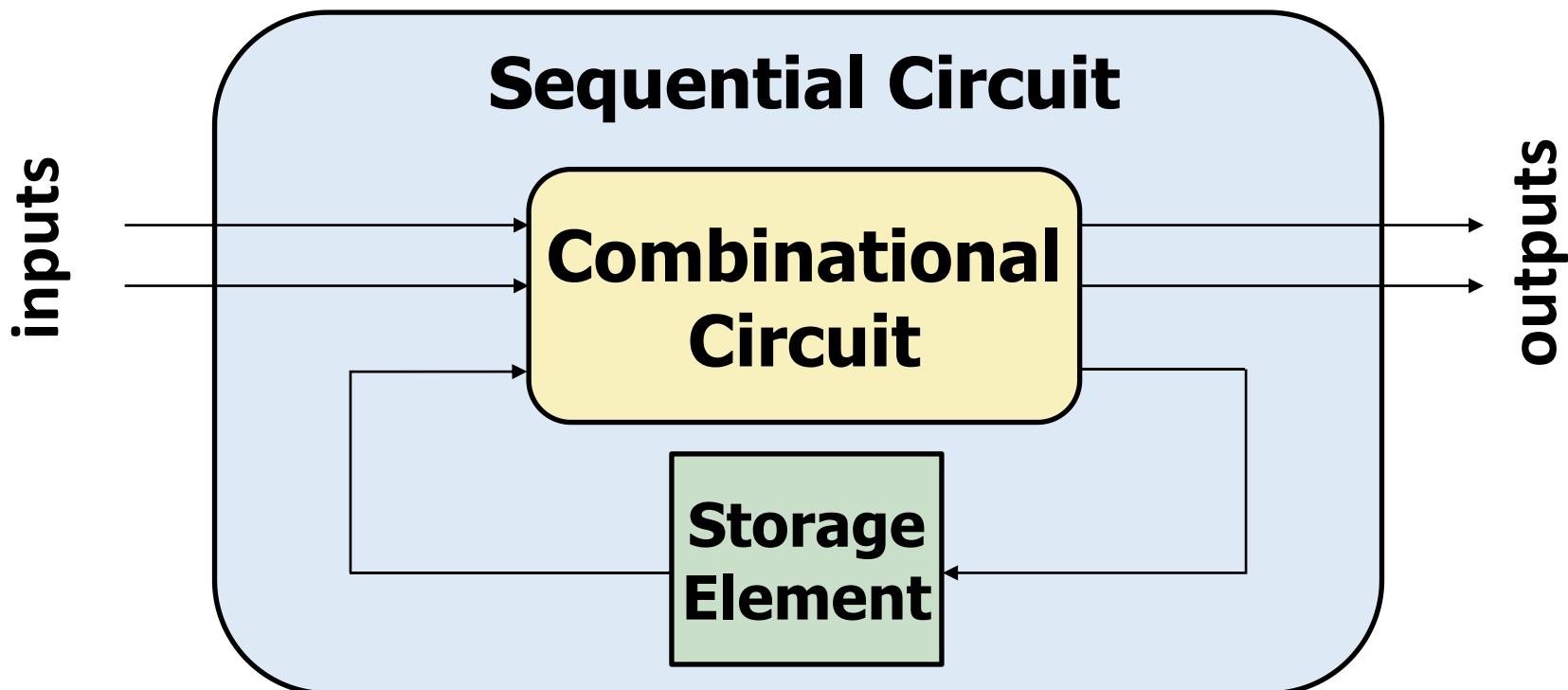


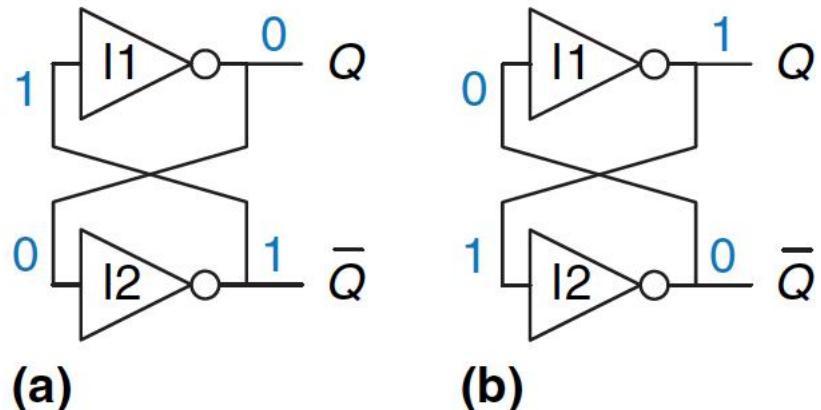
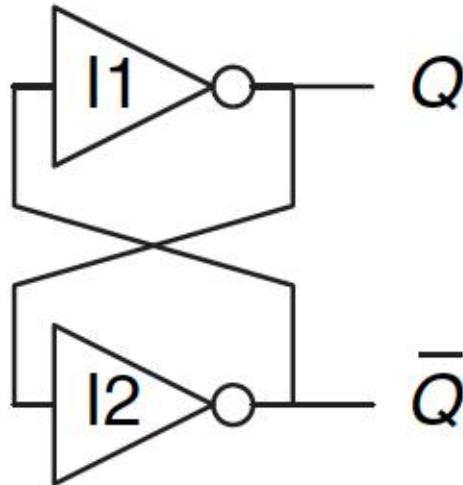
Figure 2.56 Multiplexer using tristate buffers

时序逻辑电路

- 组合逻辑电路的输出只取决于当前的输入
- 希望电路产生的输出取决于当前和过去的输入值，要求电路具有记忆功能（存储信息）
- 如何设计一个存储信息的电路？



基本记忆单元



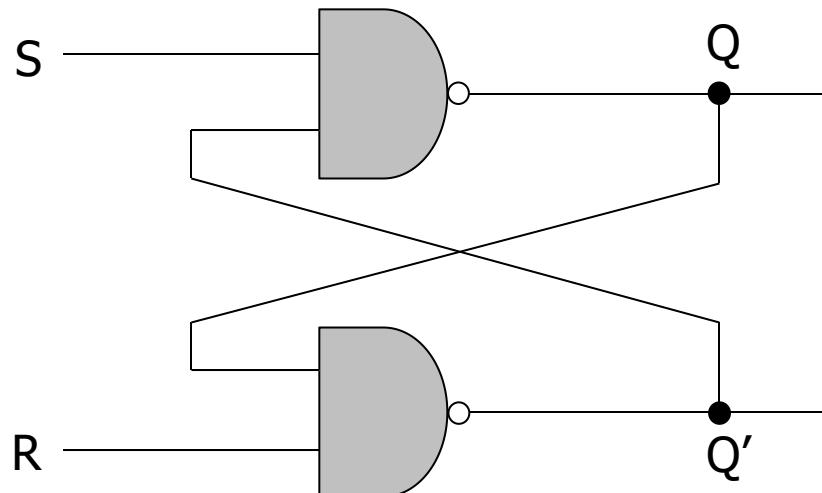
- 交叉耦合反相器 Cross-Coupled Inverters
- 有两个稳定状态。 $Q=1$ 或 $Q=0$ 。
- 有第三个 "亚稳态" 状态， 两个输出都在0和1之间振荡
- 还需要一种机制去设置Q

RS-Latch (锁存器)

□ 交叉耦合的NAND门

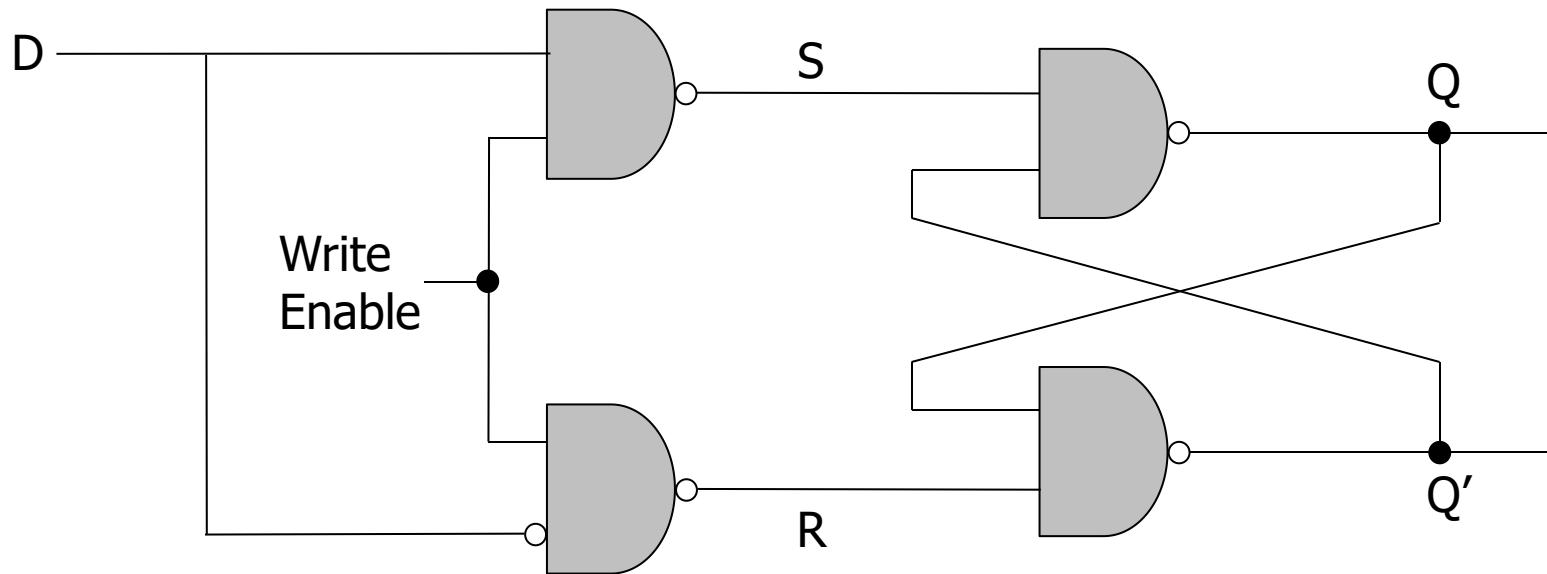
- 数据存储在Q（相反值为 Q' ）。
- S和R是控制输入
 - 在静态（空闲）状态下，S和R都保持为1
 - S（设置）：驱动S为0（保持R为1），将Q改为1
 - R（复位）：驱动R为0（保持S为1），将Q变为0

□ S和R不应该同时为0



Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

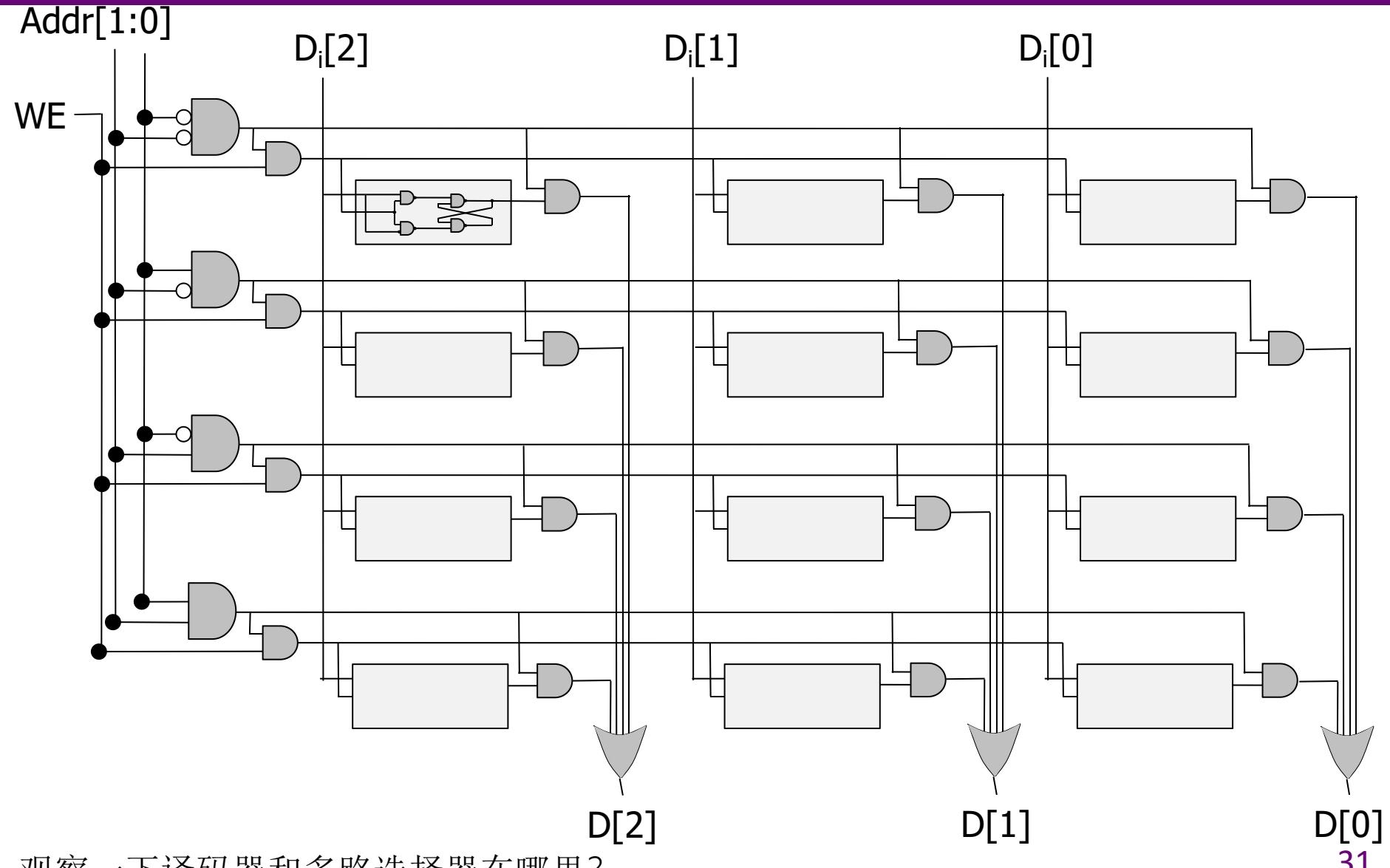
D-Latch



- 当写入使能 (WE)
被设置为1时，Q取D
的值
- S和R不可能同时为0

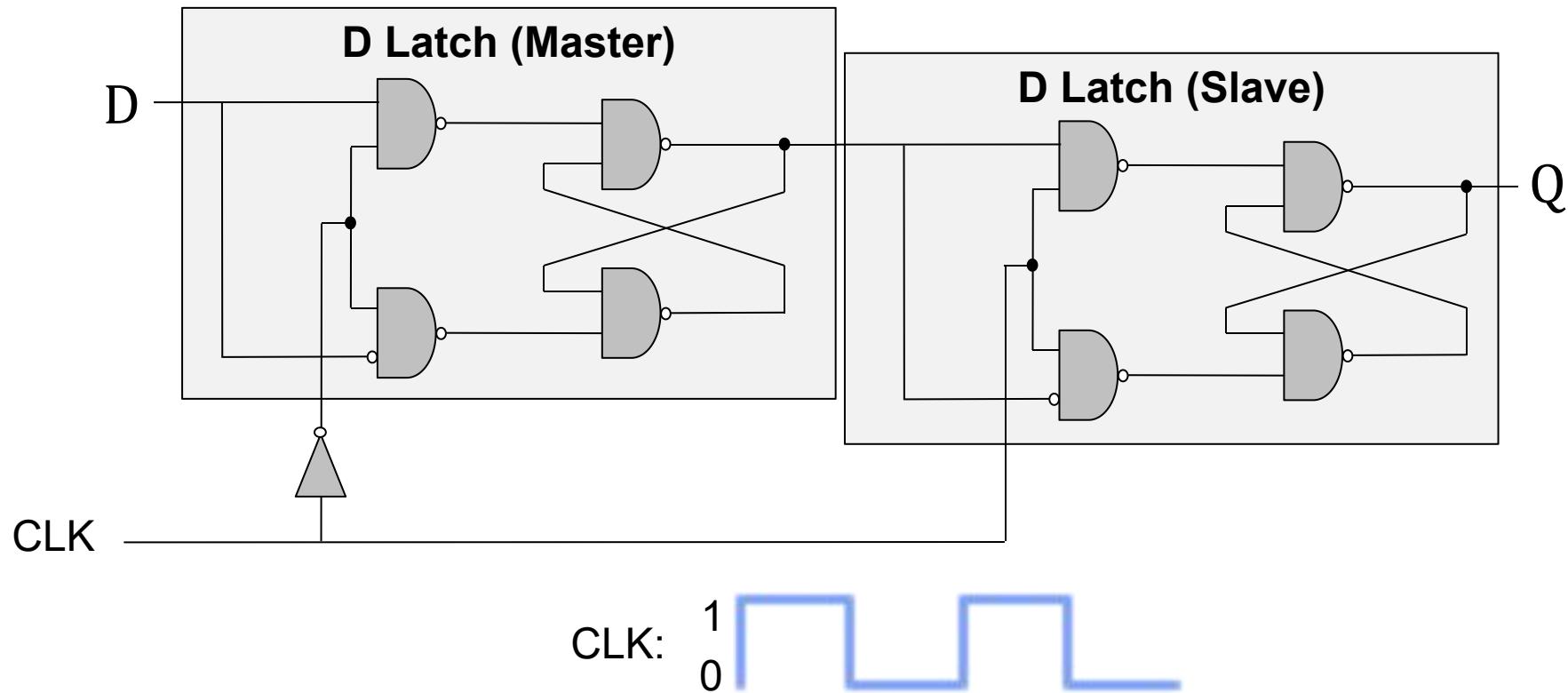
Input		Output
WE	D	Q
0	0	Q_{prev}
0	1	Q_{prev}
1	0	0
1	1	1

简单的内存组织



观察一下译码器和多路选择器在哪里？

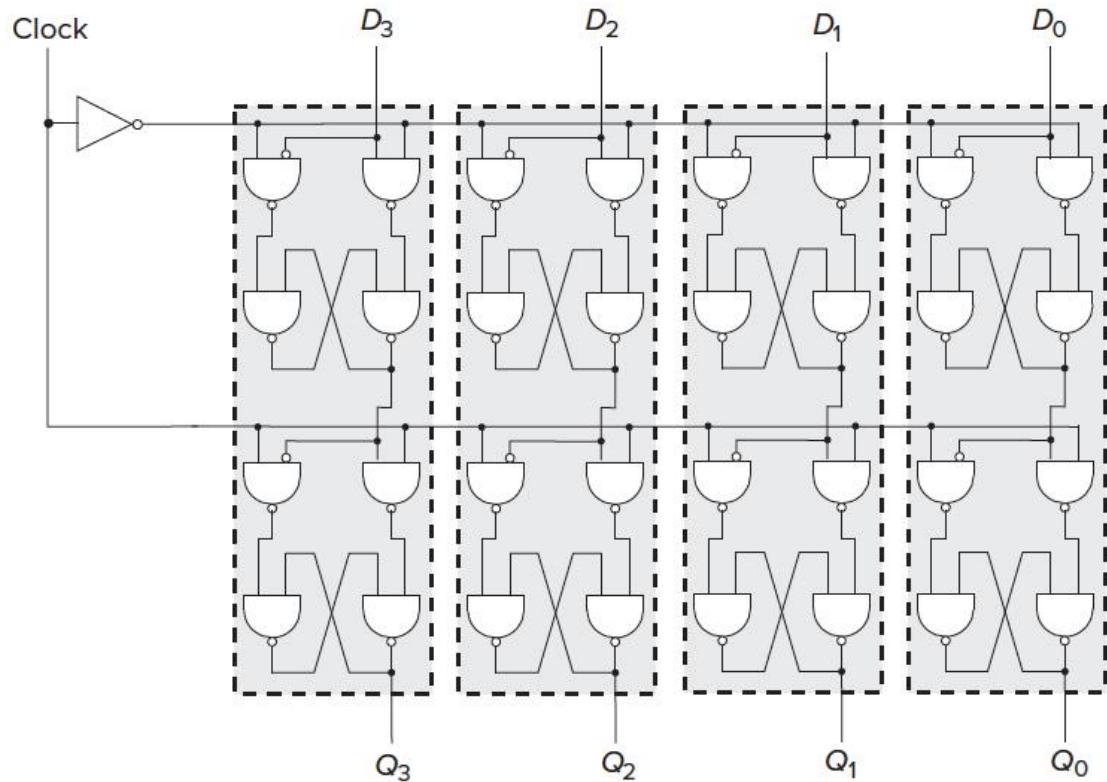
D-FlipFlop (触发器)



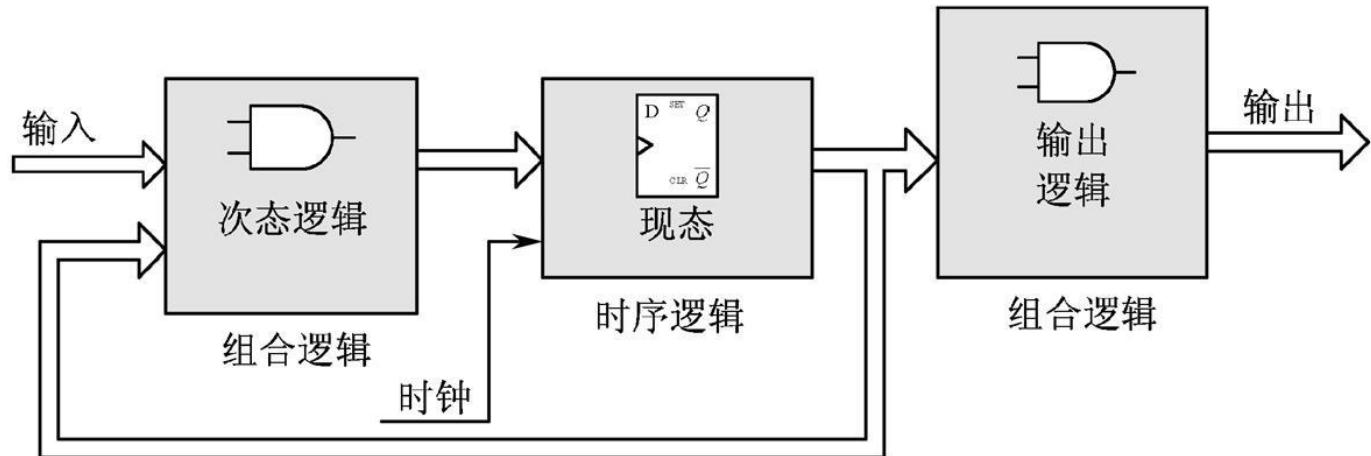
- 当时钟为低电平时， master将D传播到slave的输入端（Q不变）。
- 当时钟为高电平时， slave锁住D（Q存储D）。
- 在时钟的上升沿（时钟从0->1）， Q被分配D。
- 特性： 1) 数据在时钟边沿发生变化， 2) 数据在整个时钟周期可用

基于触发器的寄存器

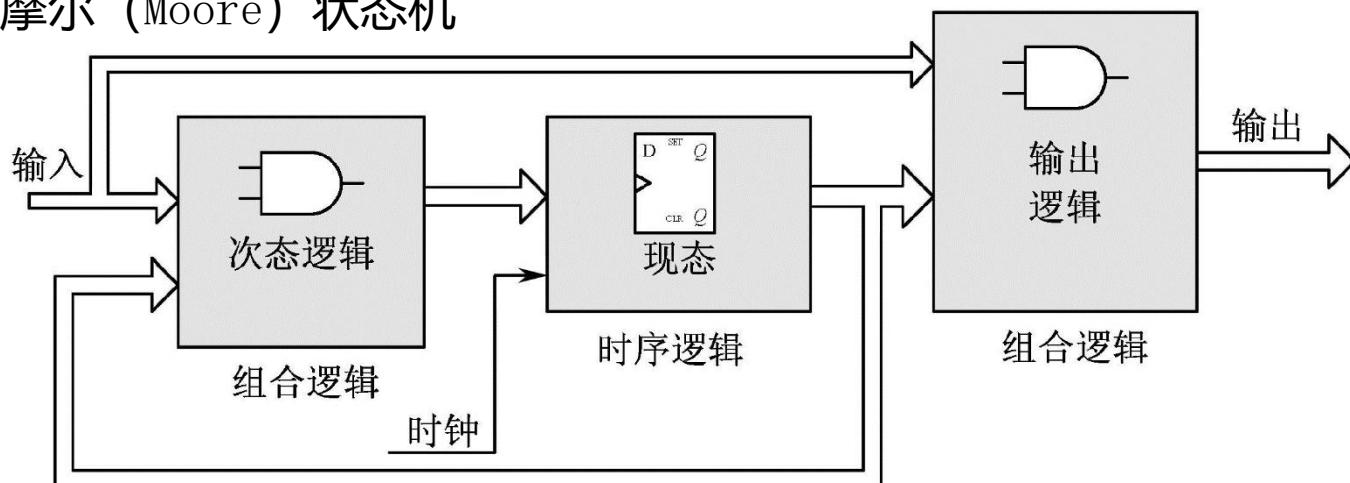
□ 一组D触发器



时序逻辑的有限状态机设计



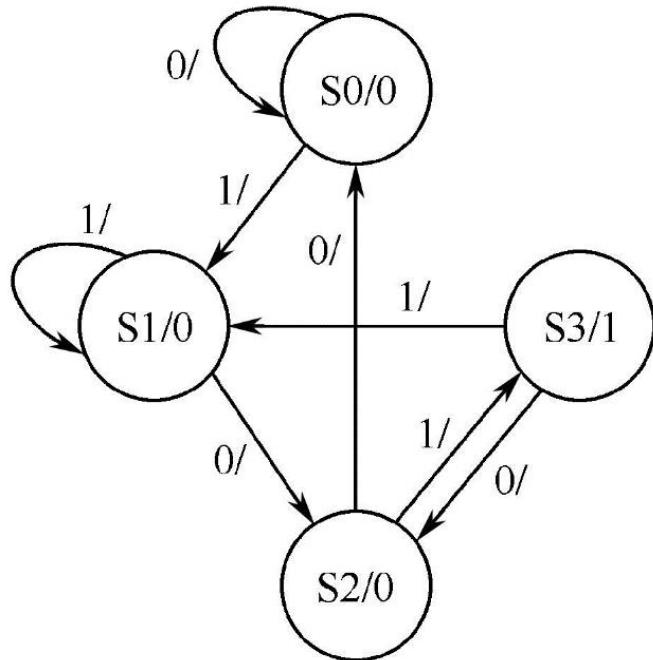
摩尔型状态机 (Moore)：输出只和当前状态有关而与输入无关，则称为摩尔 (Moore) 状态机



米利型状态机 (Mealy)：输出不仅和当前状态有关而且和输入有关，则称为米利 (Mealy) 状态机

使用有限状态机检测101序列-1

使用时序逻辑（寄存器）记录状态

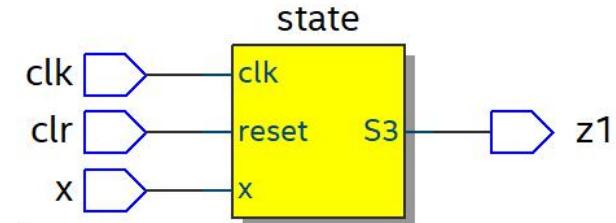


```
module ttt(clk,clr,x,z1);
  input clk,clr,x;
  output reg z1;
  reg[1:0] state,next_state;
  parameter S0=2'b00,S1=2'b01,S2=2'b11,
  S3=2'b10;
```

模块接口

现态逻辑

```
always_ff @ (posedge clk, posedge clr)
begin
  if(clr)
    state<=S0;
  else
    state<=next_state;
end
```



使用有限状态机检测101序列-1

次态逻辑

```
always_comb
begin
    case(state)
        S0:
            begin
                if(x)  next_state<=S1;
                else   next_state<=S0;
            end
        S1:
            begin
                if(x)  next_state<=S1;
                else   next_state<=S2;
            end
        S2:
            begin
                if(x)  next_state<=S3;
                else   next_state<=S0;
            end
        S3:
            begin
                if(x)  next_state<=S1;
                else   next_state<=S2;
            end
        default:
            next_state<=S0;
    endcase
end
```

输出逻辑

```
always_comb
begin
    case(state)
        S3: z1=1'b1;
        default: z1=1'b0;
    endcase
end
```

摩尔型状态机

同步电路，异步电路

□ 同步电路

- 时钟源只有一个，所有触发器连接相同的时钟源
- 触发器的状态与时钟变化同步
- 有利于静态时序分析
- 强耦合关系，不利于面积优化和低功耗优化
- 存在时钟偏斜问题（与时钟源距离不同）

□ 异步电路

- 没有统一的时钟（可以有多个时钟），有的时钟同源不同相，有的时钟不同源
- 很难对电路进行静态时序分析
- 弱耦合关系，设计灵活，比同步功耗低

□ 只考虑同步电路

小结

□ MOS管门电路特性

□ 组合逻辑

- 功能
- 延迟
- 功耗

□ 时序逻辑

□ 延迟

谢谢



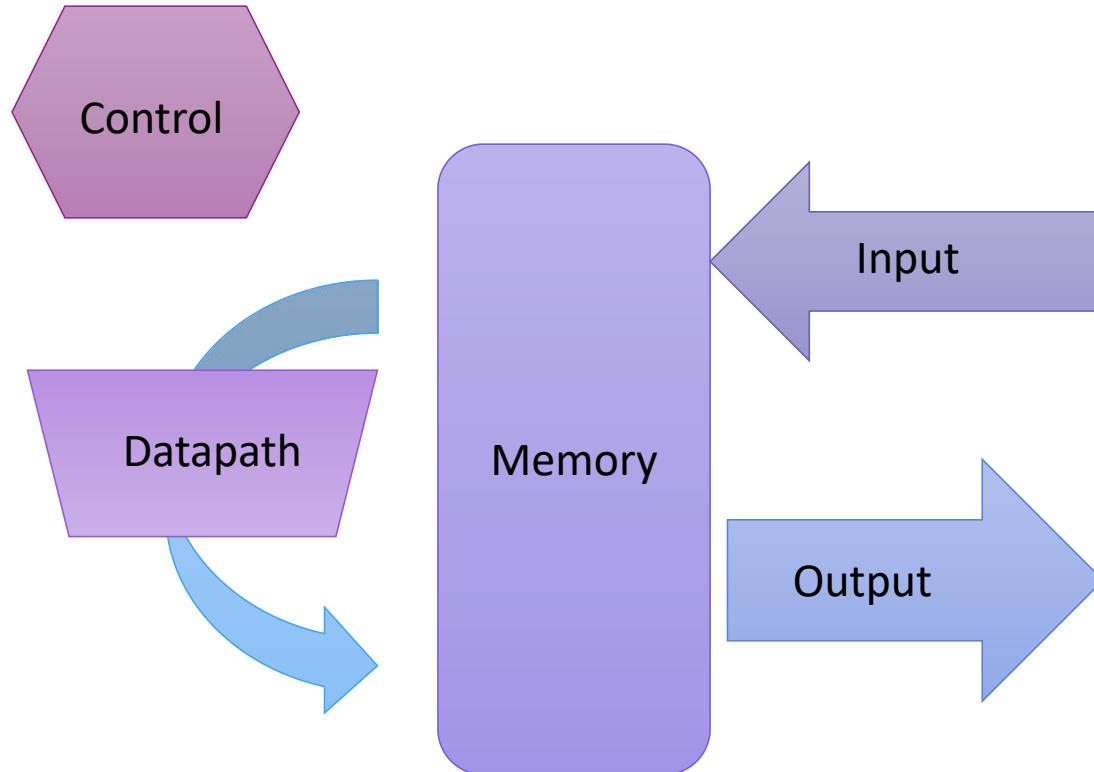
算术运算及电路实现

2022年秋

内容提要

- 运算器功能
- 用于实现运算功能的基础逻辑电路
- ALU设计
- 算术运算的实现

计算机运行机制



Datapath:
完成算术和逻辑运算，通常包括其中的寄存器

运算器基本功能

□ 完成算术、逻辑运算

- + - × ÷ ∧ ∨ ¬

□ 得到运算结果的状态

- C Z V S

□ 取得操作数

- 寄存器组、数据总线

□ 输出、存放运算结果

- 寄存器组、数据总线

□ 暂存运算的中间结果

- Q寄存器，移位寄存器

□ 由控制器产生的控制信号驱动

运算器的基本逻辑电路

□ 逻辑门电路

- 完成逻辑运算

□ 加法器

- 完成加法运算

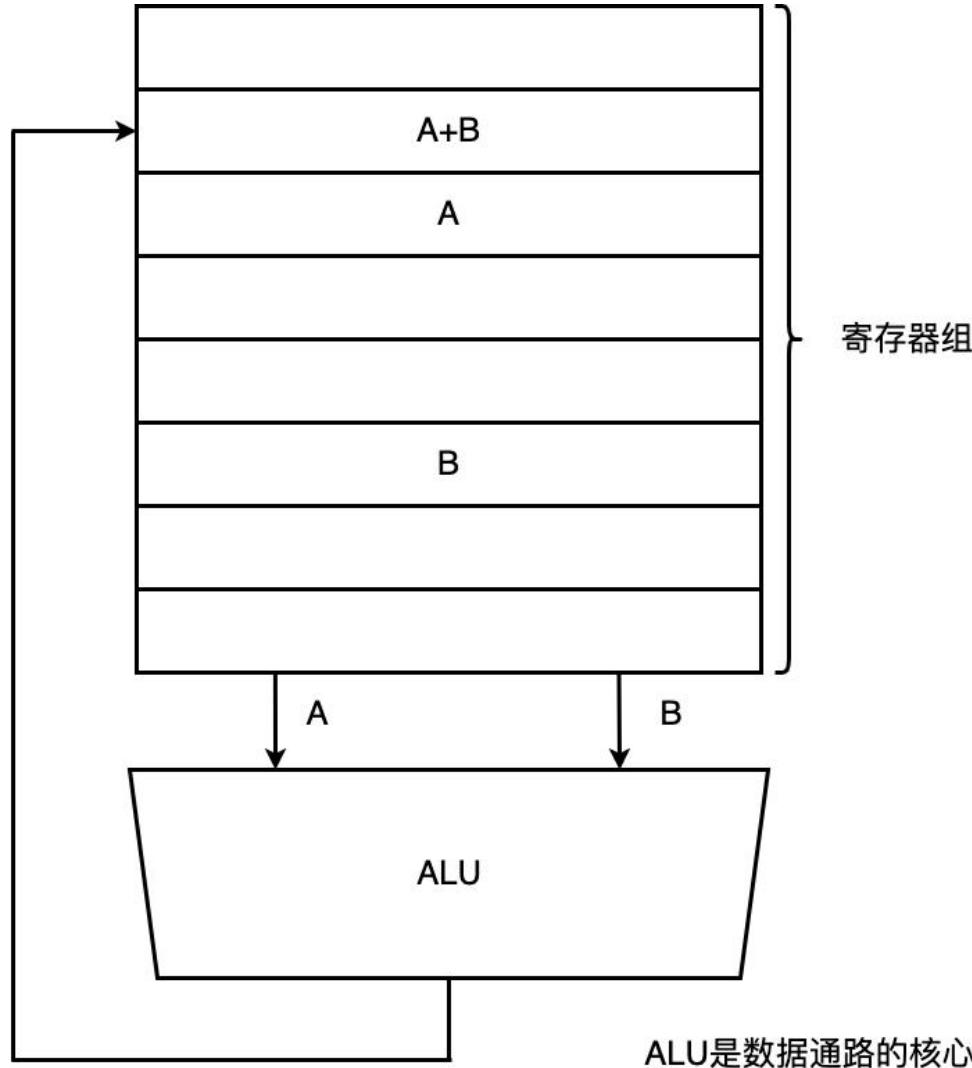
□ 触发器

- 保存数据

□ 多路选择器、移位器

- 选择、连通

数据通路 (Datapath)



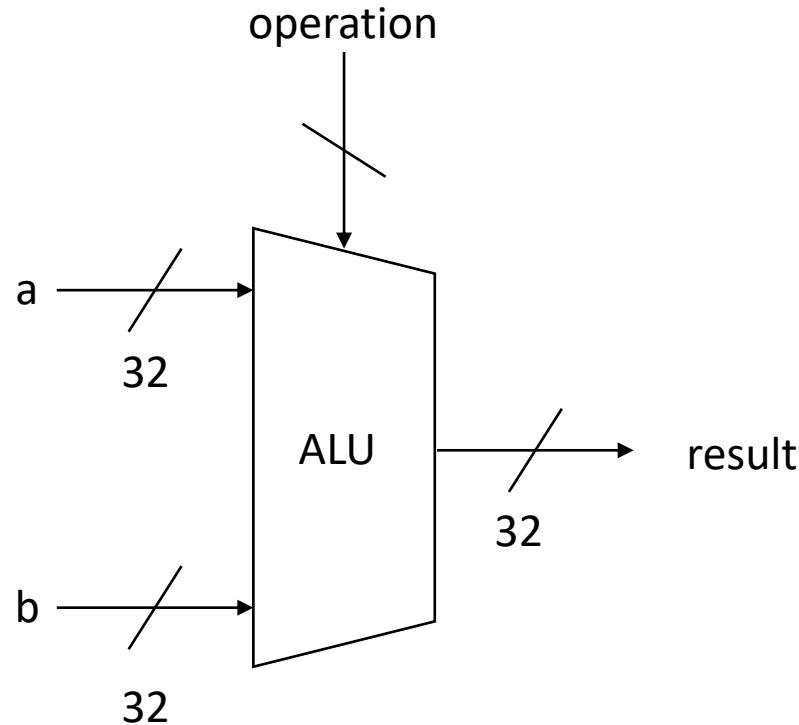
ALU功能和设计

□ 功能

- 对操作数A、B完成算术逻辑运算
- ADD, AND, OR

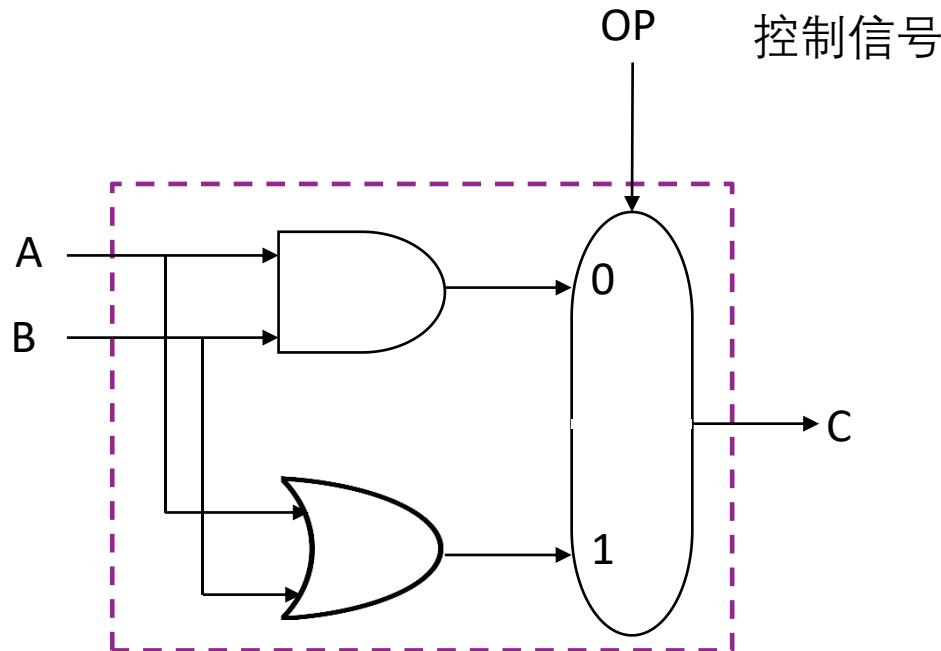
□ 设计

- 算术运算
 - 加法器
- 逻辑运算
 - 与门、或门



1位ALU逻辑运算实现

- 直接用逻辑门实现与和或的功能
- 多路选择器，通过OP控制信号输出结果



控制信号

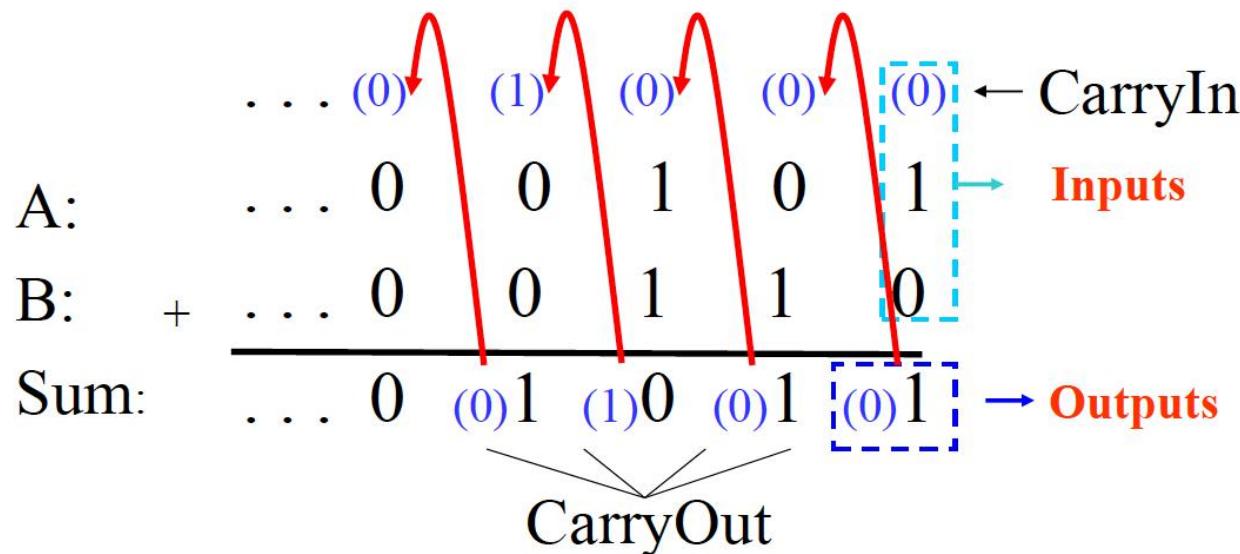
功能表

OP	C
0	A AND B
1	A OR B

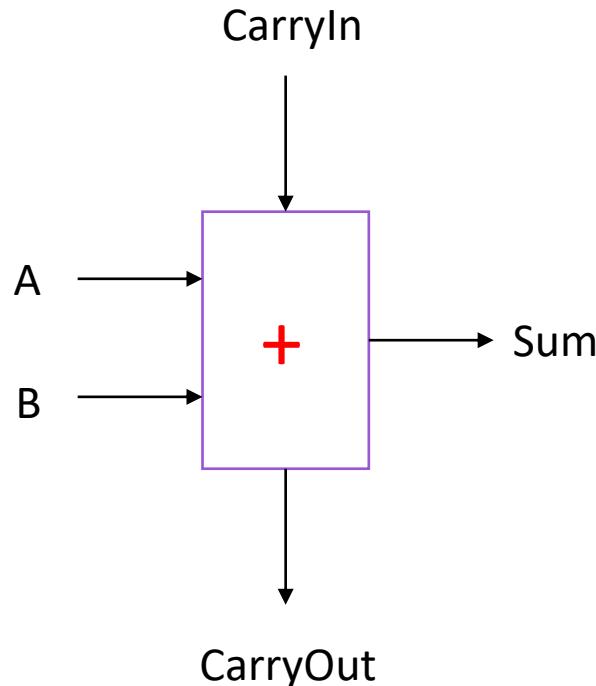
1位ALU加法运算实现

□ 1位的加法：

- 3个输入信号： $A_i, B_i, \text{CarryIn}_i$
- 2个输出信号： $\text{Sum}_i, \text{CarryOut}_i$
 - $\text{CarryIn}_{i+1} = \text{CarryOut}_i$



1位全加器的设计与实现



A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

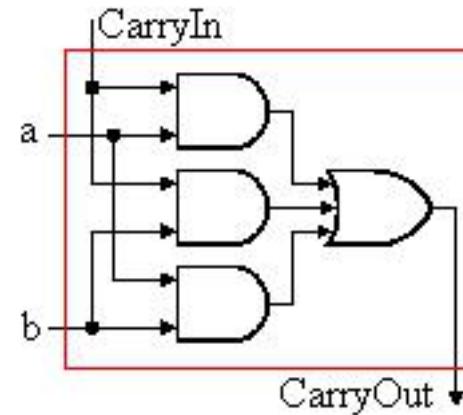
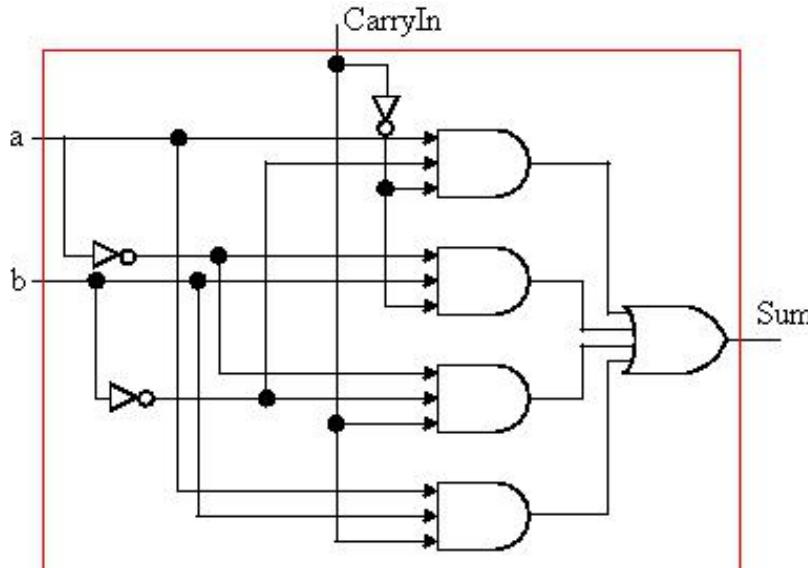
依据真值表可以获得其逻辑表达式，并通过组合逻辑实现

$$\begin{aligned} \text{CarryOut} &= (\neg A * B * \text{CarryIn}) + (A * \neg B * \text{CarryIn}) + (A * B * \neg \text{CarryIn}) + (A * B * \text{CarryIn}) \\ &= (B * \text{CarryIn}) + (A * \text{CarryIn}) + (A * B) \end{aligned}$$

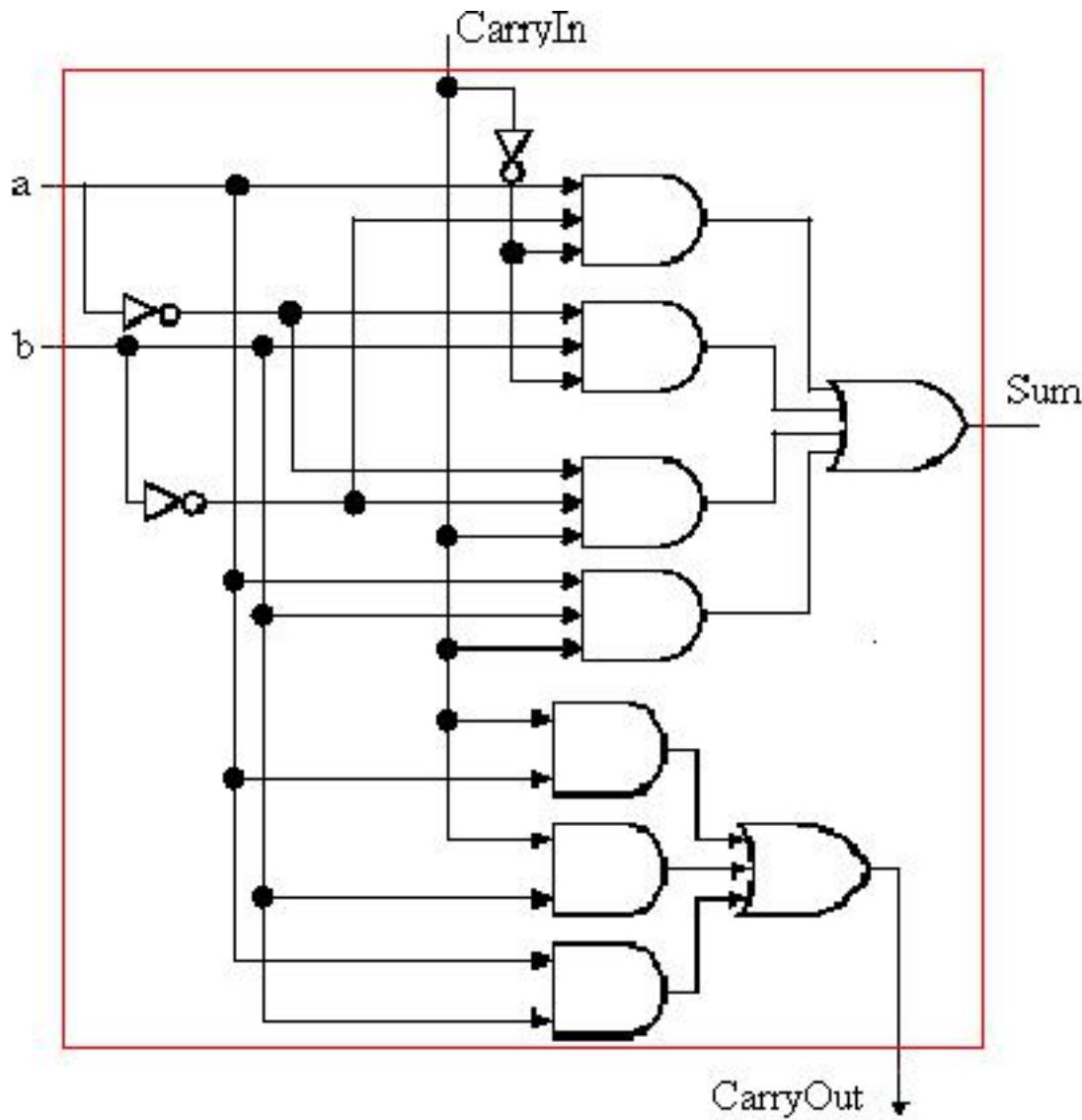
$$\text{Sum} = (\neg A * \neg B * \text{CarryIn}) + (\neg A * B * \neg \text{CarryIn}) + (A * \neg B * \neg \text{CarryIn}) + (A * B * \text{CarryIn})$$

全加器设计与实现

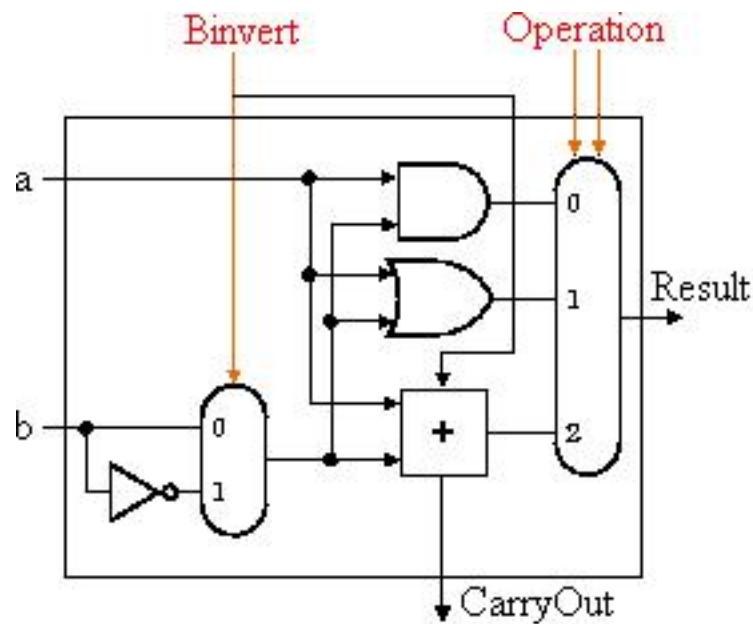
- 用逻辑门实现加法，求Sum
- 用逻辑门求CarryOut
- 将所有相同的输入连接在一起



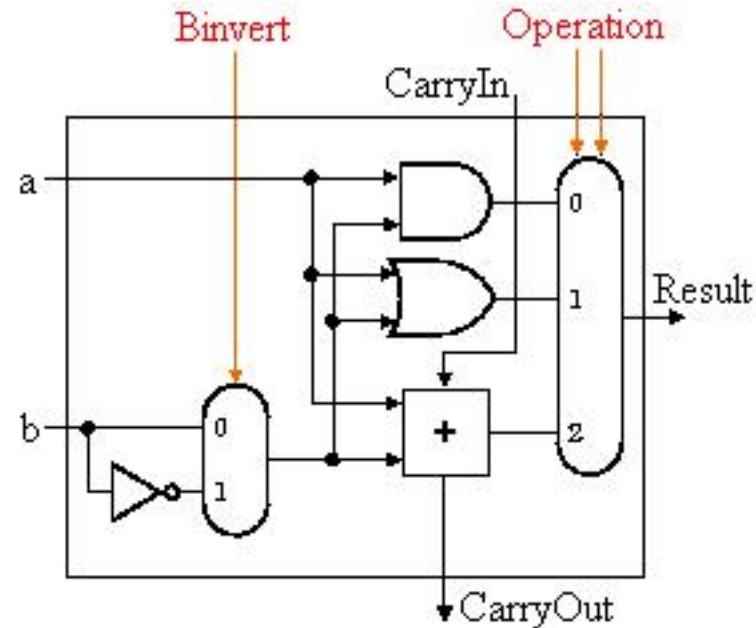
全加器



1位的ALU



最低位



其它位

1位ALU的设计过程

- 确定ALU的功能
- 确定ALU的输入参数
- 根据功能要求得到真值表，获得逻辑表达式
- 依据逻辑表达式实现逻辑电路
- 如何实现4位的ALU呢？

4位ALU实现方法

□ 思路1：

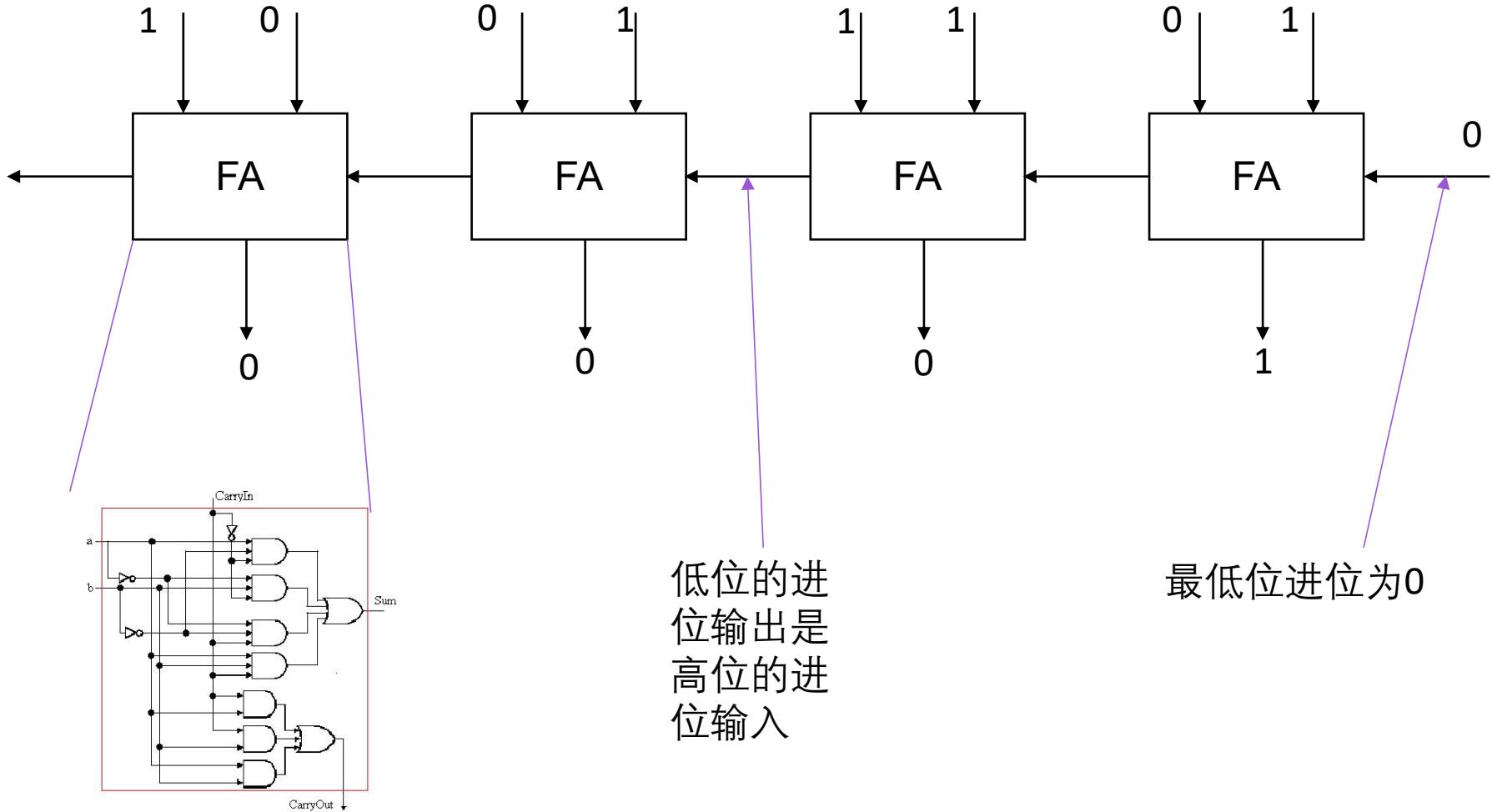
- 同1位ALU设计，写真值表，逻辑表达式，通过逻辑电路实现

□ 思路2：

- 使用1位ALU串联起来，得到4位ALU

$$\begin{array}{r} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{0} & \textcolor{blue}{0} \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline & \textcolor{red}{1} & 0 & 0 & 1 \end{array}$$

4位ALU设计



超前进位生成

□ 如何能提前得到Cout?

□ 显然

- 当a=b=0时, Cout=0
- 当a=b=1时, Cout=1
- 当a=1, b=0或a=0, b=1时候,
Cout=Cin

$$C_1 = a_1 b_1 + (a_1 + b_1) C_0 = G_1 + P_1 C_0$$

$$C_2 = a_2 b_2 + (a_2 + b_2) C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_3 = a_3 b_3 + (a_3 + b_3) C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$P_1 C_0$$

$$C_4 = a_4 b_4 + (a_4 + b_4) C_3 = \dots$$

$$P_i = a_i + b_i$$

$$G_i = a_i * b_i$$

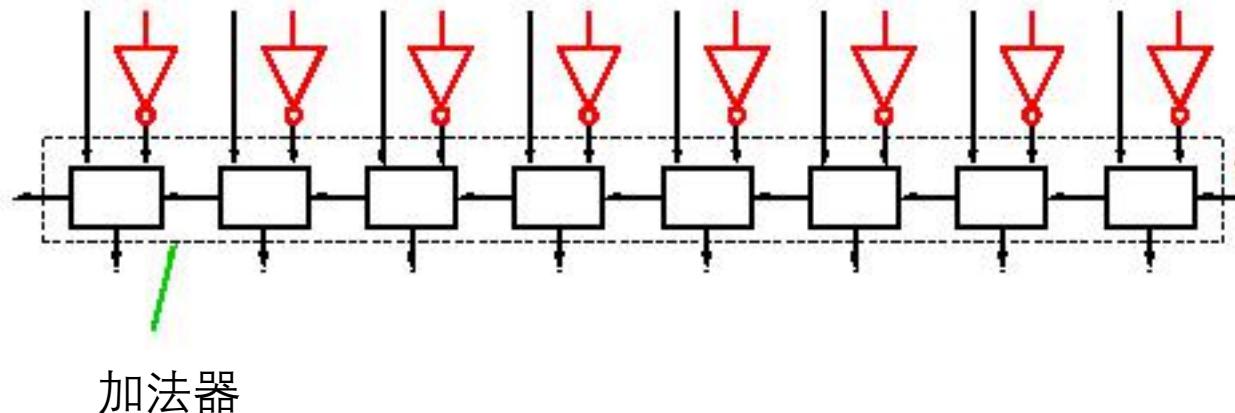
通过单独的
进位电路，
可以同时得
到计算结果
和进位

其它的结果标志

- $Z = (F1=0)^*(F2=0)^*(F3=0)^*(F4=0)$
- $S = \text{最高位}$
- $OV = \neg F1 * \neg F2 * S + F1 * F2 * \neg S$

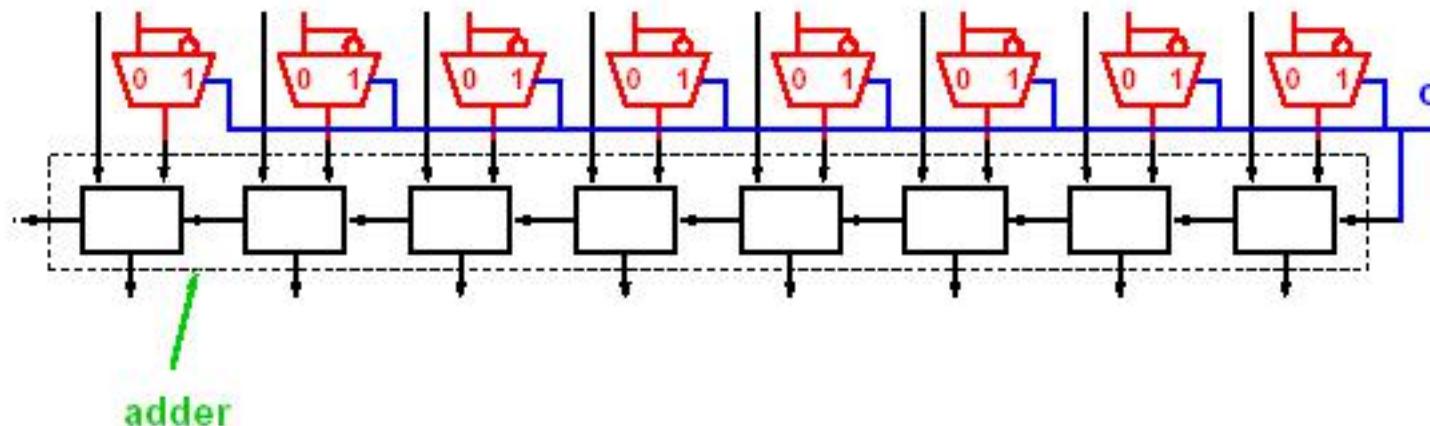
补码的减法

- 根据运算规则：
- $[a-b]_{\text{补}} = [a]_{\text{补}} + [-b]_{\text{补}}$
- $[-b]_{\text{补}}$ 的补码为：将 $[b]_{\text{补}}$ 的各位取反，并加1
- 用加法器实现减法



将加法和减法组合

- 给定控制命令C=0做加法， C=1做减法
- 可以使用选择器来实现



原码乘法

□ 从一个简单的例子开始

$$\begin{array}{r} & 1 & 0 & 0 & 0 \\ \times & 1 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array}$$

二进制乘法算法描述

□ 基本算法

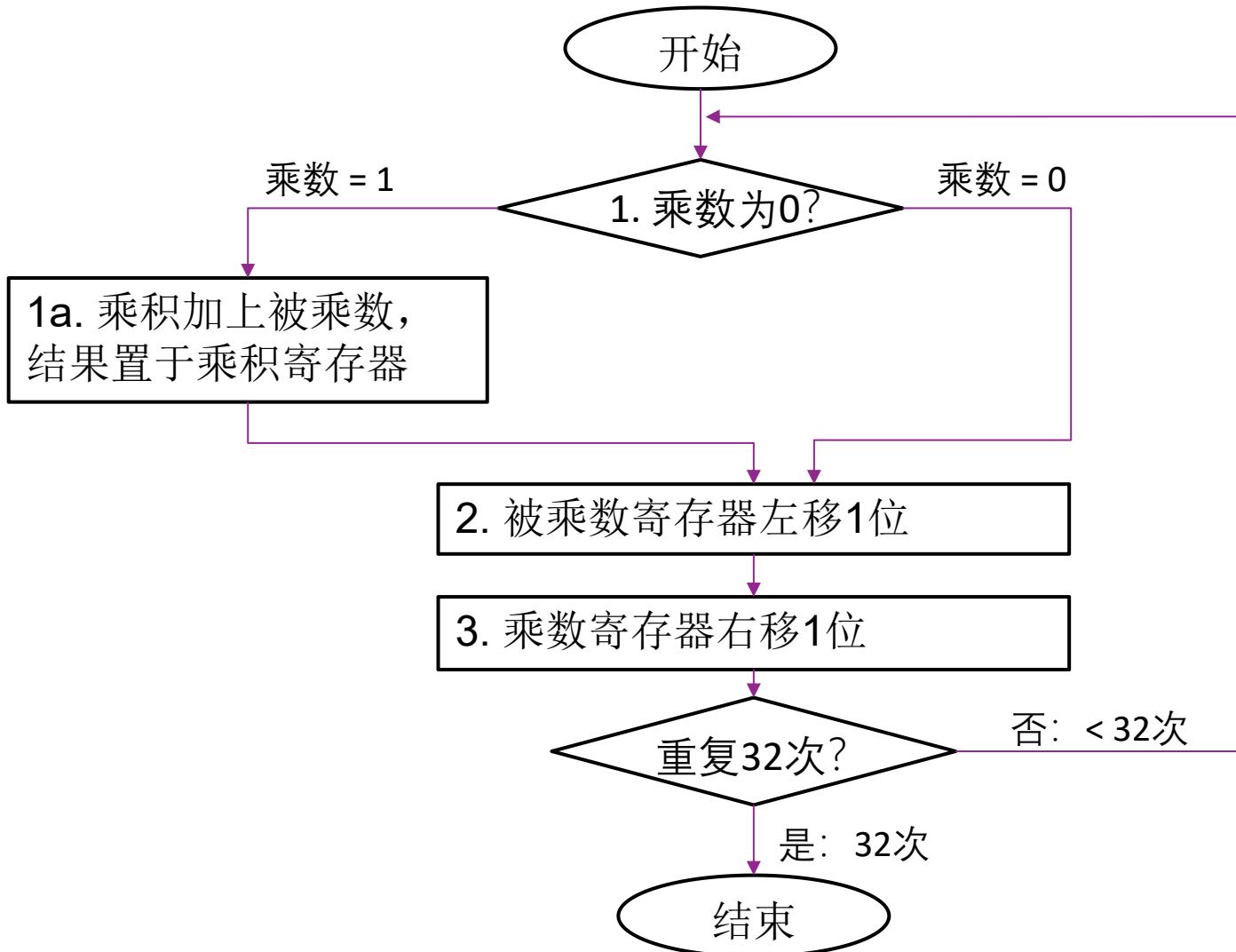
- 若乘数的当前位==1， 将被乘数和部分积求和
- 若乘数的当前为==0，则跳过
- 将部分积移位
- 所有位都乘完后， 部分积即为最终结果

□ N位乘数*M位被乘数 → N+M位的积

□ 乘法显然比加法更加复杂

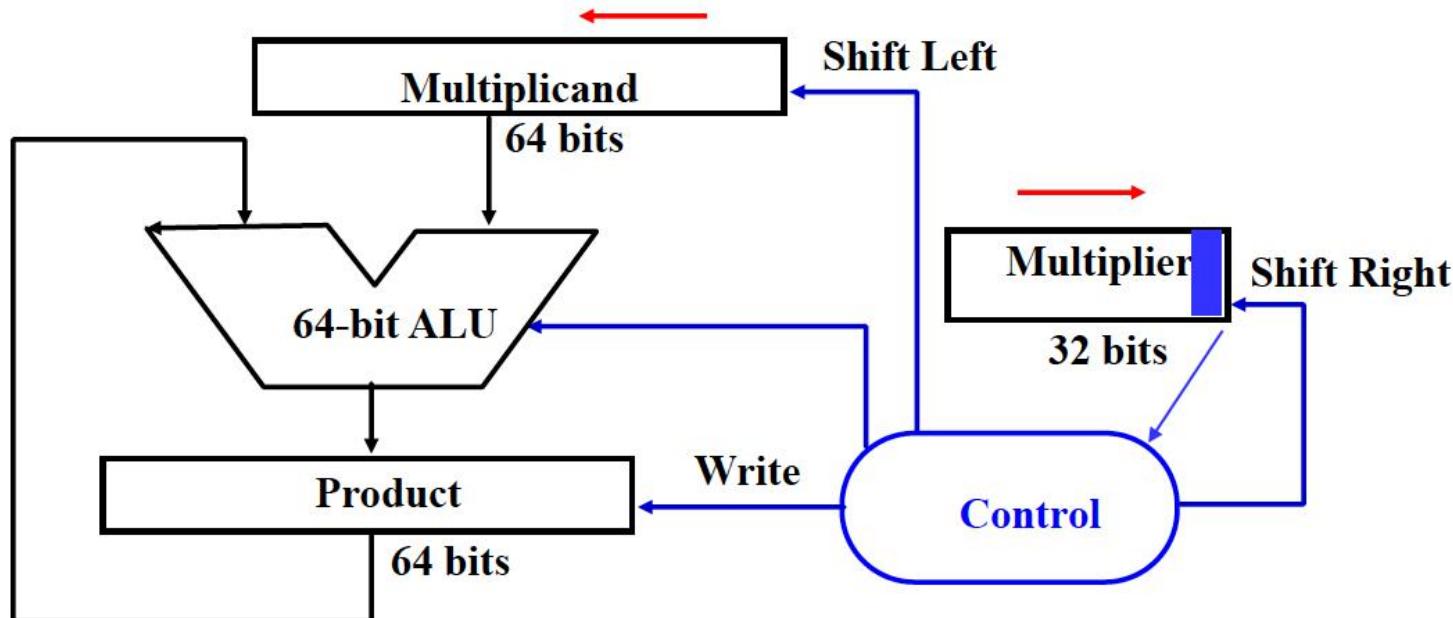
- 但是要比10进制乘法要简单

乘法算法 (1)



原码乘法的实现（一）

- 64-位被乘数寄存器，64-位ALU，64-位部分积寄存器，32-位乘数寄存器



$$\text{Multiplier} = \text{datapath} + \text{control}$$

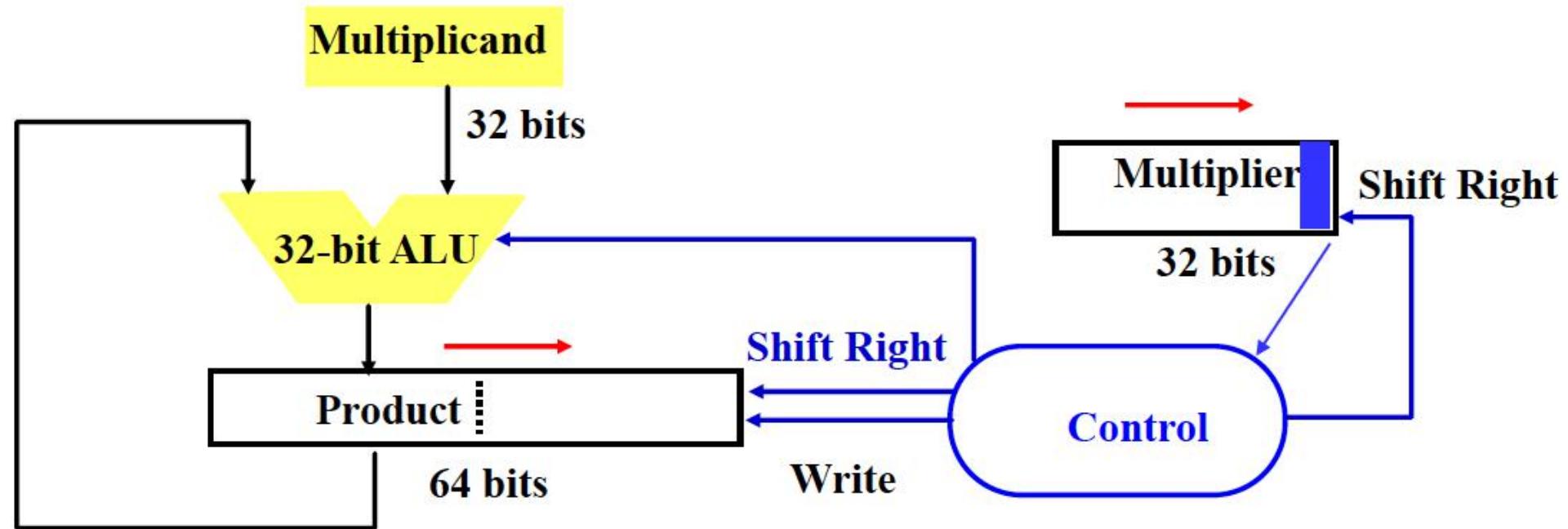
不足

□ 实现（一）的不足：

- 被乘数的一半存储的只是0，浪费存储空间
- 每次加法实际上只有一半的位有效，浪费了计算能力

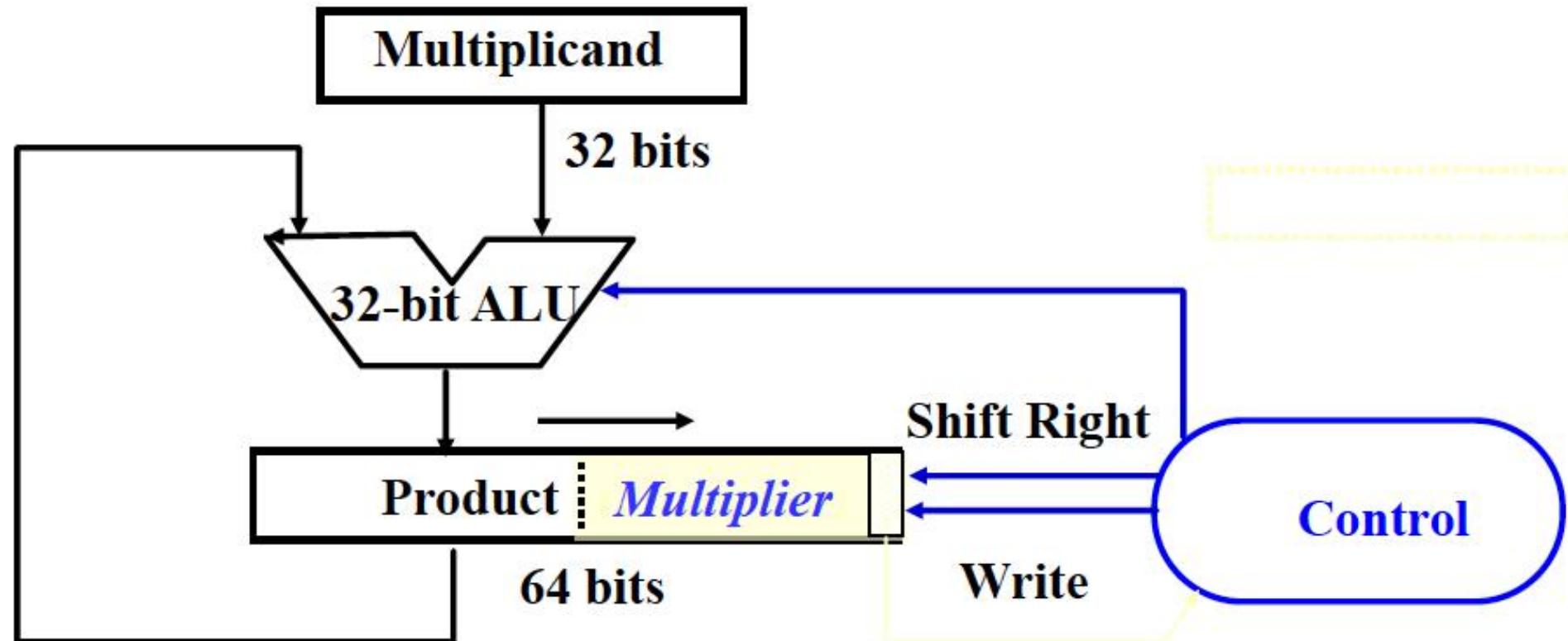
原码的乘法实现（二）

- 32位被乘数寄存器，32位ALU，64位部分积寄存器



原码乘法实现（三）

- 32位被乘数寄存器，32位ALU，64位部分积寄存器（0-位乘数寄存器）



实现（三）的优点

- 实现（二）解决了对加法器位数的浪费
- 需要注意的是，乘数寄存器也存在浪费的情况
 - 把已经完成的乘数位移出，移入的是0
 - 解决这个浪费，可以把乘数和部分积低位结合起来

补码乘法

□ 方案一：

- 将补码转换为原码绝对值，进行原码的正数乘法
- 依据以下原则得到符号位，并转换回补码表示
 - 同号为正
 - 异号为负

□ 方案二：补码直接乘

- 布斯算法

布斯算法的推导过程

□ 布斯算法的原理

- 虽然乘法是加法的重复，但也可以将它理解成加法和减法的组合。

□ 例如：十进制乘法

$$\square 6 \times 99 = 6 \times 100 - 6 \times 1 = 600 - 6 = 594$$

布斯算法

□ 二进制举例

$$0111 \times 0011 = ?$$

$$\begin{array}{r} & 0 & 1 & 1 & 1 \\ \times & 0 & 0 & 1 & 1 \\ \hline & 0 & 1 & 1 & 1 \\ & 0 & 1 & 1 & 1 \\ & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

我们也可以把它看成是下面计算过程：

$$0 - 7 * 1 + 7 * 4 = 0 - 7 + 28 = 21$$

补码乘法运算

若 $[x]_{\text{补}} = x_{n-1}x_{n-2} \cdots x_1x_0$, 则:

$$x = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

同理: $[y]_{\text{补}} = y_{n-1}y_{n-2} \cdots y_1y_0$, 则:

$$y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} y_i 2^i$$

补码乘法运算

$$\begin{aligned}[x \times y]_{\text{补}} &= [x]_{\text{补}} \times y \\ &= [x]_{\text{补}} \times (-2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} y_i 2^i) \\ &= [x]_{\text{补}} \times [2^{n-1}(y_{n-2} - y_{n-1}) + 2^{n-2}(y_{n-3} - y_{n-2}) + \cdots + 2^0(y_{-1} - y_0)] \\ &= [x]_{\text{补}} \times \sum_{i=0}^{n-1} 2^i (y_{i-1} - y_i), \text{ 其中: } y_{-1} = 0\end{aligned}$$

- 可直接用补码进行乘法运算
- 根据乘数相邻两位的不同组合，确定是 $+[x]_{\text{补}}$ 或 $-[x]_{\text{补}}$

补码乘法运算

用 Y 的值乘 $[X]_{\text{补}}$, 达到 $[X]_{\text{补}} \times [Y]_{\text{补}}$, 求出 $[X * Y]_{\text{补}}$, 不必区分符号与数值位。乘数最低一位之后要补初值为 0 的一位附加线路, 并且每次乘运算需要看附加位和最低位两位取值的不同情况决定如何计算部分积, 其规则是:

- 00: +0
- 01: +被乘数
- 10: -被乘数
- 11: +0

举例： $2 \times (-5)$

步骤	操作	部分积	附加位
0	初始值	0000011011	0
1	-X	1111011011	0
	右移	1111101101	1
2	右移	1111110110	1
3	+X	0000110110	1
	右移	0000011011	0
4	-X	1111011011	0
	右移	1111101101	1
5	右移	1111110110	1

$$(1111110110)_2 = -10$$

乘法运算：小结

- 与加法比较，需要使用更多的硬件来实现，也更复杂
- 若使用简单的方法来实现，则需要多个计算周期
- 仅仅介绍了乘法运算的一些“皮毛”：有许多提升和优化的空间

除法运算

- 在计算机内实现除运算时，存在与乘法运算类似的几个问题：加法器与寄存器的配合，被除数位数更长，商要一位一位地计算出来等。这可以用左移余数得到解决，且被除数的低位部分可以与最终的商合用同一个寄存器，余数与上商同时左移。
- 除法可以用原码或补码计算，都比较方便，也有一次求多位商的快速除法方案，还可以用快速乘法器完成快速除法运算。

原码一位除运算

$$[Y/X]_{\text{原}} = (X_s \oplus Y_s) (|Y| / |X|)$$

- 原码一位除是指用原码表示的数相除，求出原码表示的商。除操作的过程中，每次求出一位商。
- 恢复余数法：被除数-除数，若结果 $>=0$ ，则上商1，移位；若结果 <0 ，则商0，恢复余数后，再移位；
- 求下一位商；
- 但计算机内从来不用这种办法，而是直接用求得的负余数求下一位商。

原码一位除运算

$$[X / Y]_{\text{原}} = (X \oplus Y) (|X| / |Y|)$$

例如： $X = 0.1011$ $Y = -0.1101$

$$\begin{array}{r} 0.1101 \\ 0.1101 \sqrt{0.10110} \\ \underline{-01101} \\ 10010 \\ \underline{01101} \\ 10100 \\ \underline{1101} \\ 0111 \end{array}$$

被除数（余数）	商	
00 1011	00000	初态
01 0110	00000	第1次
01 0010	0001	第2次
00 1010	0011	第3次
01 0100	0110	第4次
00 0111	01101	第5次

X 和 Y 符号异或为负
最终商原码表示为：

1 1101
余数为： $0.0111 \cdot 2^{-4}$

加减交替除法原理证明

1. 若第*i*-1次求商，减运算的余数为 $+ R_{i-1}$ ，商1，余数左移1位得 $2R_{i-1}$ 。

2. 则下一步第*i* 次求商 $R_i = 2R_{i-1} - Y$, 若 $R_i \geq 0$, ...

若 $R_i < 0$, 商0

恢复余数为正且左移1位得 $2(R_i + Y)$

3. 则再下一步第*i+1*次求商 $R_{i+1} = 2(R_i + Y) - Y$
 $= 2R_i + Y$

□公式表明, 若上次减运算结果为负, 可直接左移, 本次用 $+Y$ 求余即可; 减运算结果为正, 用 $-Y$ 求余

加减交替除法

被除数(余数)	商	
0 0 1 0 1 1	0 0 0 0 0	开始情形
+) 1 1 0 0 1 1		-Y
—————	0 0 0 0 0	<0, 商0
1 1 1 1 0 0	0 0 0 0 0	左移1位
+) 0 0 1 1 0 1		+Y
—————	0 0 0 0 1	>0, 商1
0 1 0 0 1 0	0 0 0 1 0	左移1位
+) 1 1 0 0 1 1		-Y
—————	0 0 0 1 1	>0, 商1
0 0 1 0 1 0	0 0 1 1 0	左移1位
+) 1 1 0 0 1 1		-Y
—————	0 0 0 1 1 0	<0, 商0
1 1 1 1 0 1	0 1 1 0 0	左移1位
+) 0 0 1 1 0 1		+Y
—————	0 1 1 0 1	>0, 商1

补码除法运算

□ 补码除法与原码除法很类似，差别仅在于：

- 被除数与除数为补码表示，
- 直接用补码除,求出反码商，
- 再修正为近似的补码商.

□ 实现中, 求第一位商要判 2 数符号的同异, 同号, 作减法运算, 异号, 则作加运算;

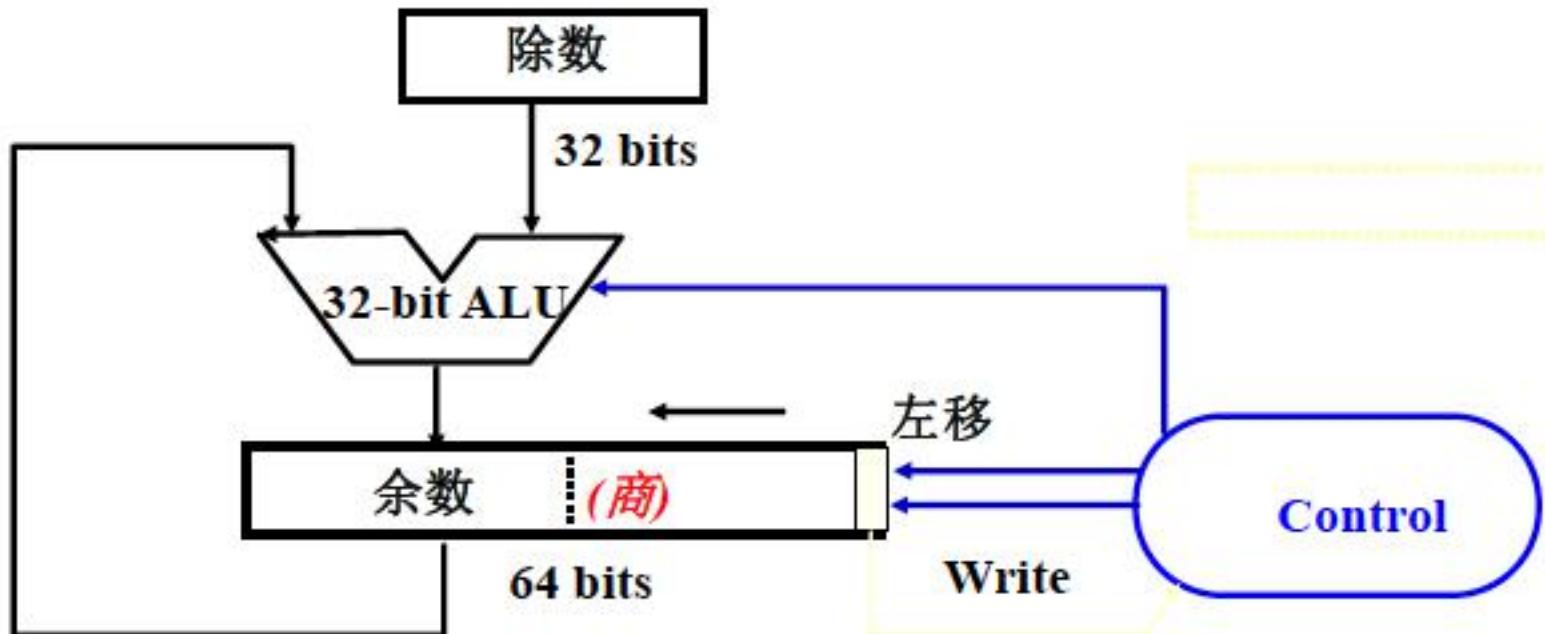
□ 上商, 余数与除数同号, 商 1, 作减求下位商,

□ 余数与除数异号, 商 0, 作加求下位商;

□ 商的修正：多求一位后舍入，或最低位恒置 1

除法的实现

- 32-位除数寄存器，32 -位ALU, 64-位余数（被除数）寄存器



小结

- ALU的基本功能：算术、逻辑运算
- 1位ALU：最基本的功能：加法、与、或
- 位数扩展：快速进位
- 功能扩展：减法、乘法、除法

谢谢



实验预备课

2022年秋

实验预备课

- 可编程逻辑器件介绍
- 硬件编程方法与原则
- 硬件编程流程

可编程逻辑器件设计

- 可编程器件简介
- 设计原则
- 设计流程

可编程器件简介

□ 概述

- PLD是电子设计领域中最具活力和发展前途的一项技术。
- PLD能做什么呢？可以毫不夸张的讲，PLD能完成任何数字器件的功能，上至高性能CPU,下至简单的位片电路，都可以用PLD来实现。
- 目前有多家公司生产CPLD/FPGA，主要有：
ALTERA(Intel)，XILINX，Lattice，Actel。



可编程器件

□ FPGA

- Field Programmable Gate Array 现场可编程门阵列
- FPGA基于SRAM的架构，集成度高，以LE（包括查找表、触发器及其他）为基本单元，有内嵌Memory、DSP等，支持IO标准丰富。

查找表

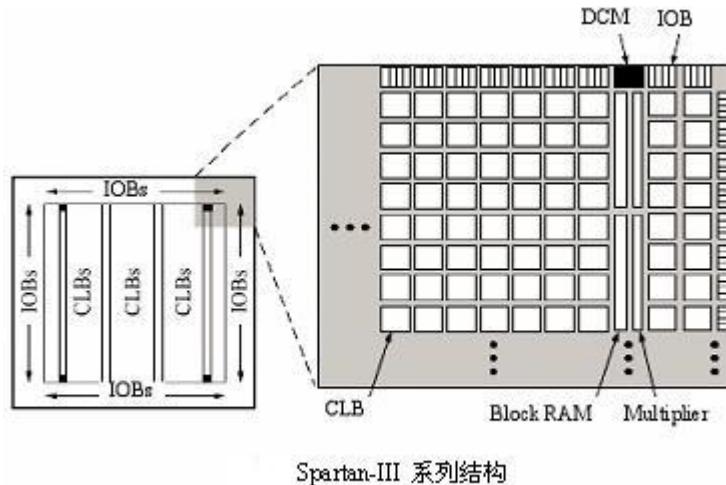
□ 基于查找表 (Look-Up-Table)的原理与结构:

- 采用这种结构的PLD芯片如altera的ACEX,APEX系列, xilinx的Spartan,Virtex系列等。
- 查找表 (Look-Up-Table)简称为LUT, LUT本质上就是一个RAM
- 工作原理
 - 当用户通过原理图或HDL语言描述逻辑电路
 - 软件会自动计算逻辑电路的所有可能的结果，并把结果事先写入RAM
 - 每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。

4输入与门

XilinxFPGA主要部件

- 可编程输入输出单元(IOB)
- 可编程逻辑块(CLB)
- 时钟管理模块(DCM)
- 片内RAM(BRAM)
- 布线资源
- 内嵌功能单元
- 内嵌硬核



Xilinx公司产品概述

□ FPGA

- Virtex 系列
- Spartan器件系列



□ CPLD

- XC9500系列
- CoolRunner系列



□ 其他

- 配置器件SPROM（S系列 P系列）
- IP核

典型的应用领域

□ 数据采集

- 逻辑接口
- 电平接口
 - 电平不同
 - 单端到差分

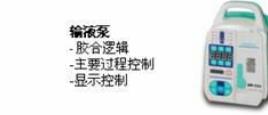


□ 数字信号处理

□ IC设计验证

□ 其他类，消费，医疗，工业控制

□ 数字信号的基本上都可以用PLD实现

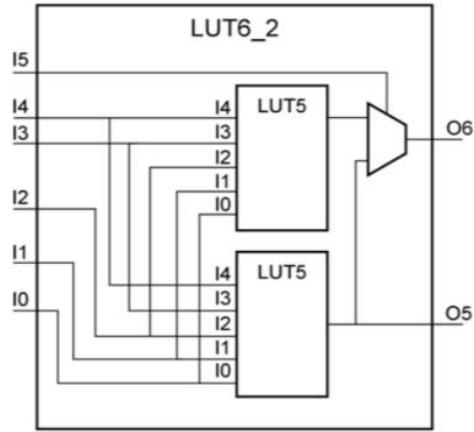


发展趋势

- 高密度，大容量，高速度
- 低成本，低电压，微功耗，微封装
- 基于IP的设计方法
 - FPGA厂家
 - 开源硬件组织
- 动态可重构
 - 通信系统
 - 重构计算机

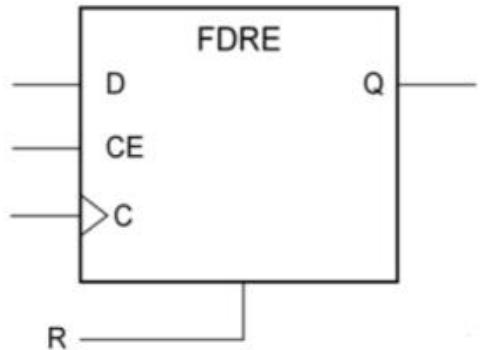
FPGA的片内资源

LUT结构



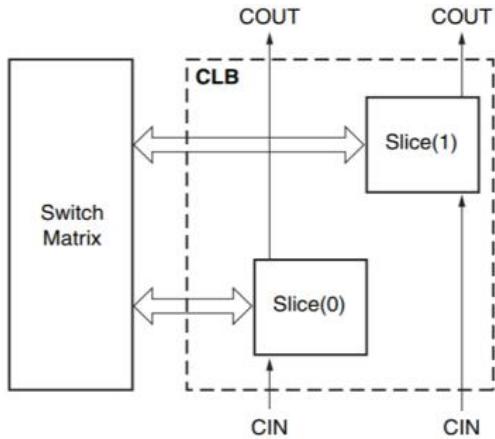
- 5输入， 2输出
- 6输入， 1输出

FF结构（寄存器）

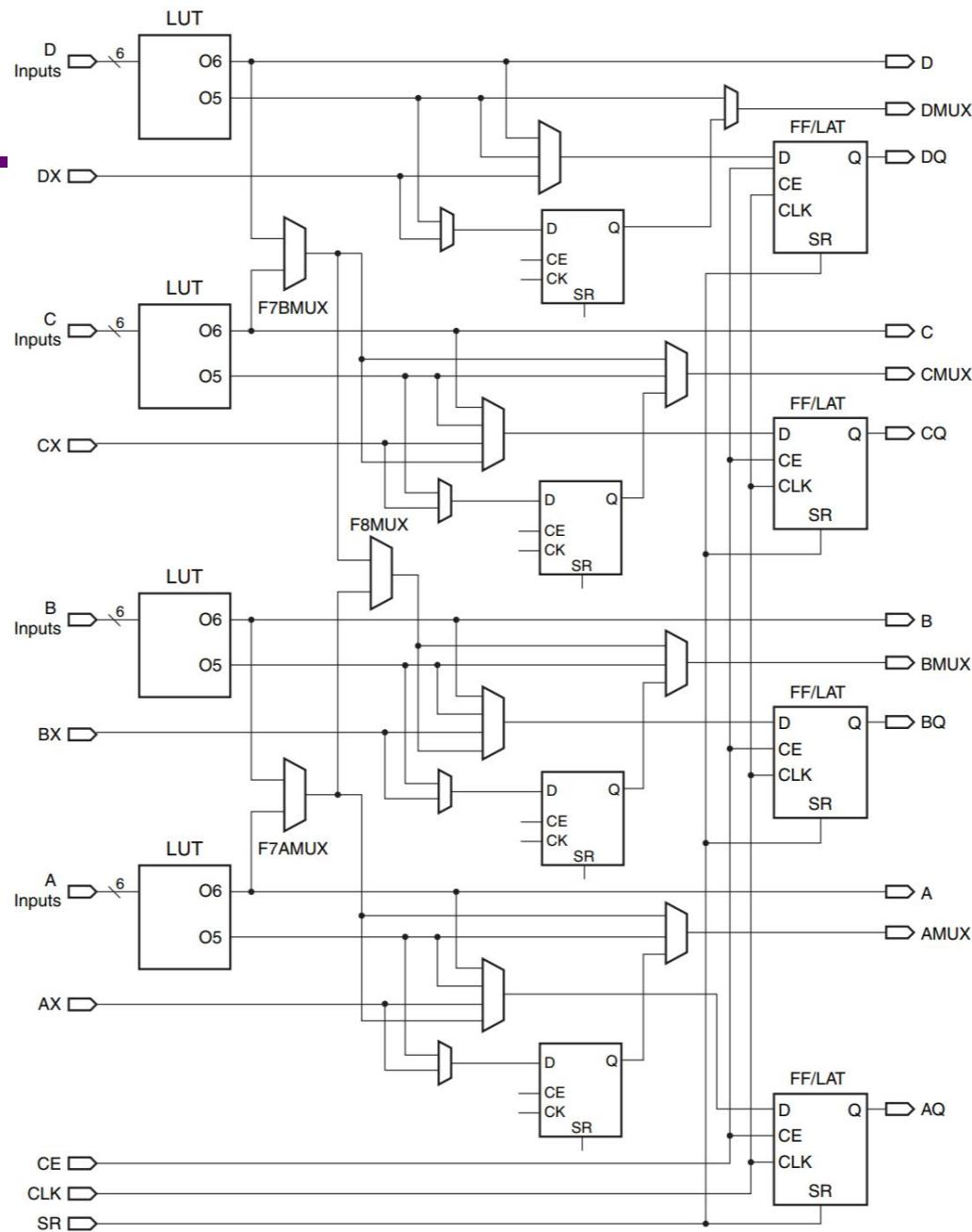


- D是数据输入
- CE是使能信号
- C是时钟信号
- Q是输出
- R是复位信号

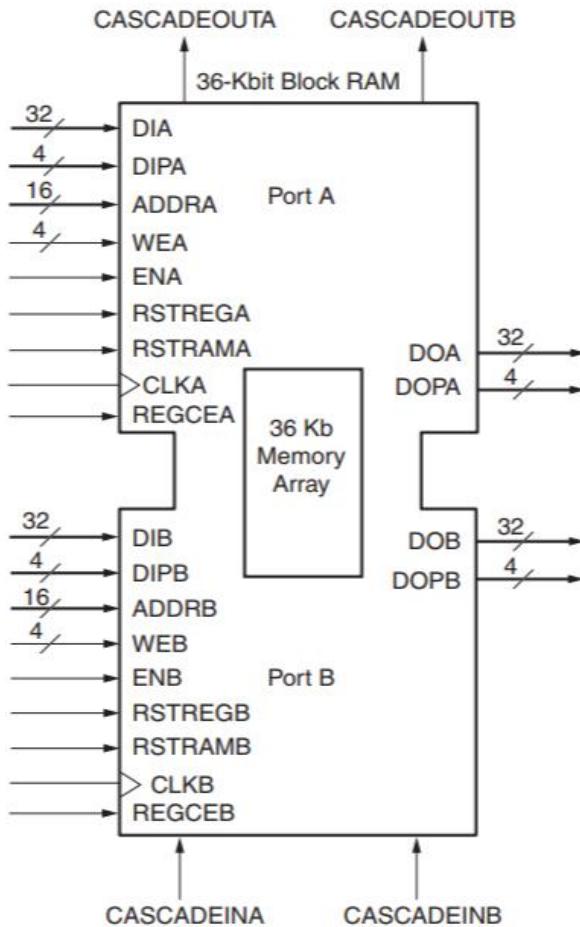
逻辑片SLICE



每两个Slice被组织成一个可配置逻辑块 (CLB) , 最后大量的CLB之间再由开关阵列 连接起来。这样的架构使得FPGA片上的LUT、FF可以自由地连接，形成一个更大规模、可配置的逻辑电路

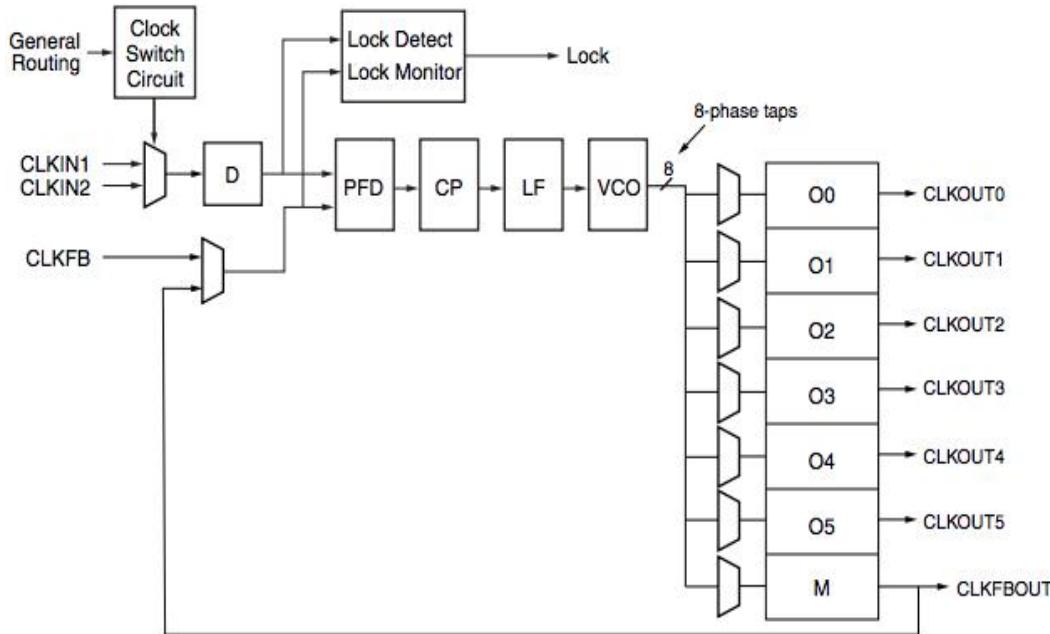


嵌入存储器



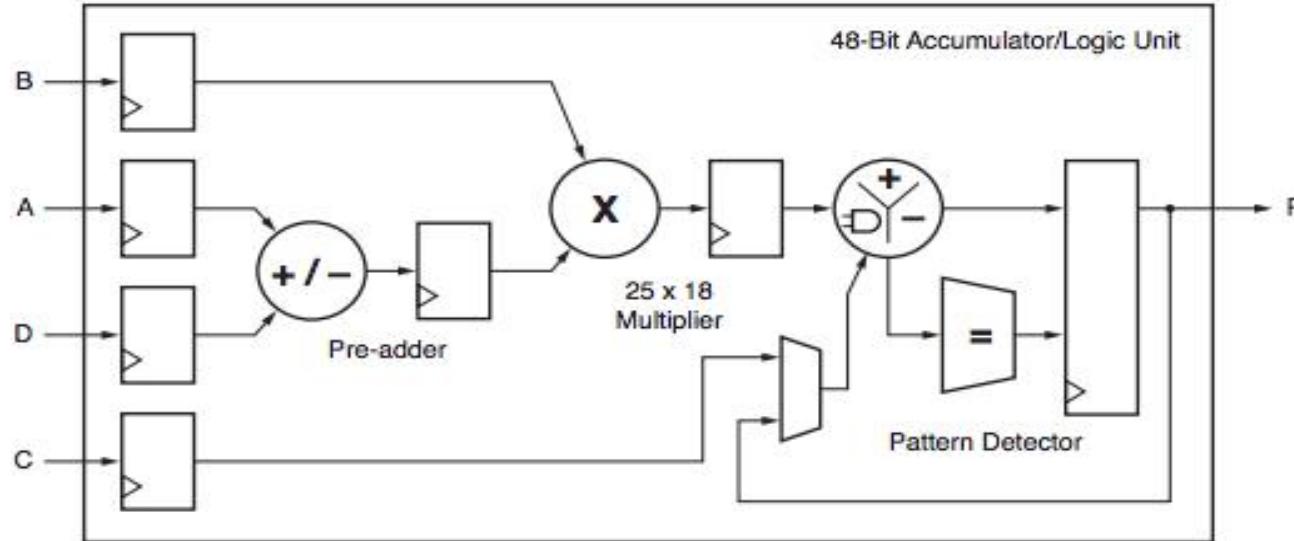
- Block RAM(BRAM)本质上是RAM，是FPGA内部的存储器
- 共有135个BRAM，总的BRAM容量为4860Kb
- 使用Block Memory Generator来使用BRAM

时钟管理单元



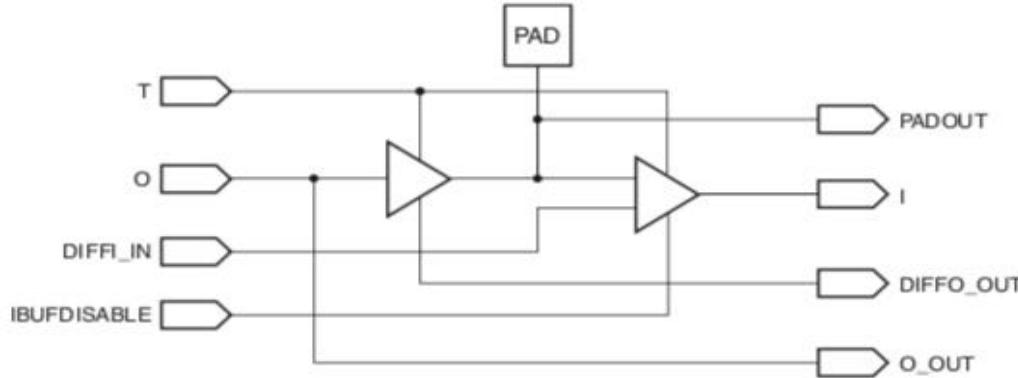
- 可以对输入的时钟信号进行分频、倍频，从而得到用户需要的时钟
- 一个时钟合成器中可以设置多个分频比例，从而得到多个不同频率的时钟

数字信号处理单元



- 包含一个单周期硬件乘法器和一个加法器（也可以认为是累加器）
- 可以用来实现乘法指令操作

IO单元



- I/O口支持输入、输出信号，输出信号还受到3态门控制（可以使得引脚变为高阻态）
- 由EDA工具根据用户编写的逻辑代码中顶层信号的类型input、output和inout来自动选择

硬件设计的方法与原则

一些硬件设计原则

- 面积和速度的平衡与互换
- 功耗考虑
- 硬件原则
- 系统原则
- 同步设计原则

面积和速度的平衡与互换

- 面积和速度是数字系统设计考虑的两个重要指标，FPGA作为快速原型设计和系统验证的方法，首先就要考虑到这两个因素直接的平衡问题；
- 面积指某个FPGA设计综合之后占用的系统资源数，一般用占用的逻辑单元数量及IO接口数量来衡量，这一指标综合软件一般都能给出；
- 更小的面积通常代表更低的成本。

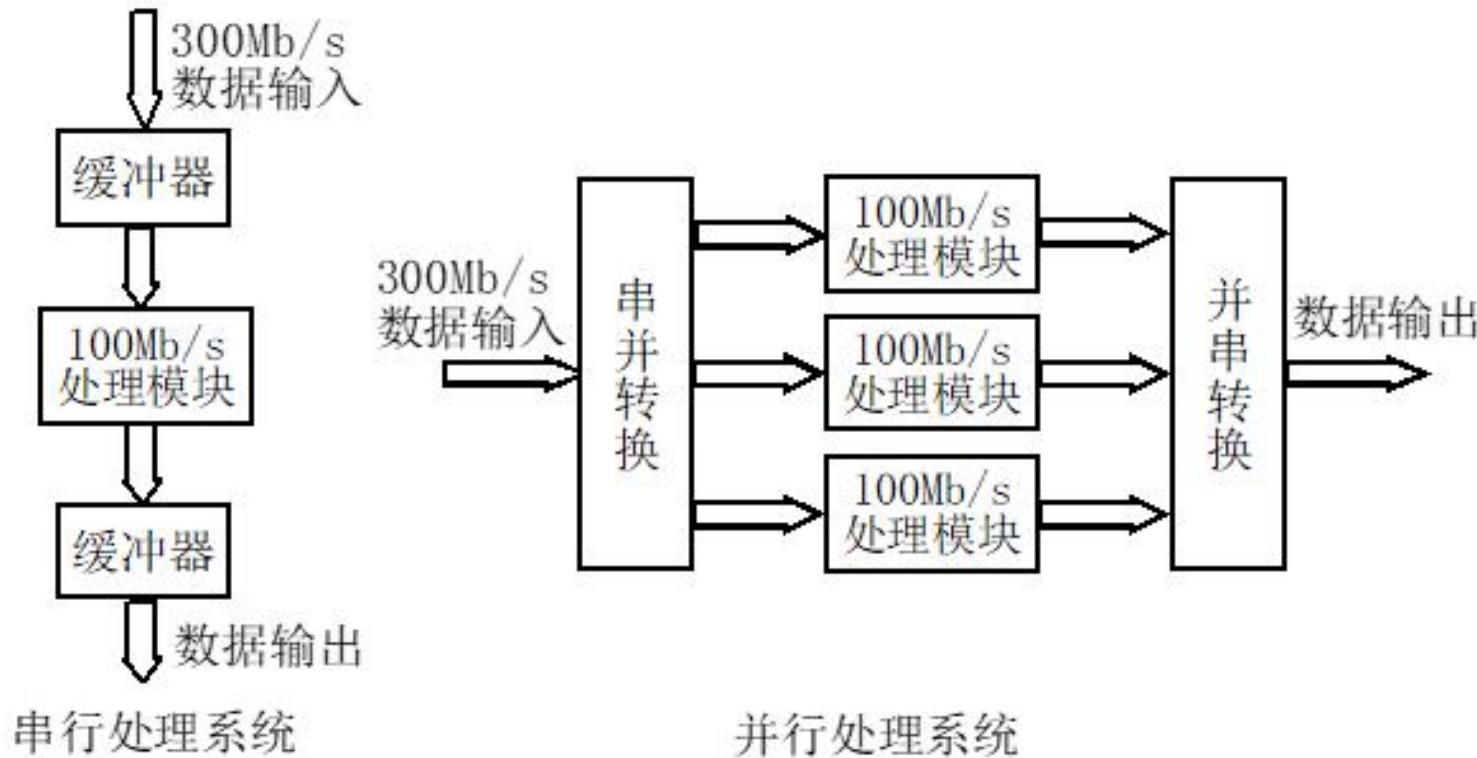
面积和速度的平衡与互换

- 简单说，速度通常指系统工作的频率，高频率常常代表高速度；
- 实际上，进行速度优化不仅仅是简单提高频率，而是要仔细考虑系统各个模块在各种工作状态下的时序要求；
- 另外可以通过并行操作提高速度；
- 在一定的工艺条件下，面积和速度常常是一对矛盾，因此需要考虑面积和速度的转换问题。

面积和速度的平衡与互换

- 将原本复用的模块进行复制，变为并行操作的模块，以牺牲面积来换取速度；
- 很多被复用的模块都是具有逻辑承接或时间先后关系的，无法直接并行化；
- 需要修改硬件设计，重新对模块做规划。

面积和速度的平衡与互换，例子



速度换面积

- 和利用面积换速度正好相反，把并行模块进行复用，以节省面积；
- 逻辑上更为简单，理论上，只要用足够的存储器，总能把并行的功能相同的模块进行复用；
- 但是要优化好存储器的管理，节省存储器，工作也并不简单。

功耗考虑

□ 主要考虑动态功耗：

- 动态功耗和逻辑翻转变化的频率成正比；
- 设计时要考慮尽可能不要让所有的单元同时翻转，避免引起过大的功耗；
- 例如，对于状态机的状态分配，之所以采用Gray码，另一个重要原因就是为了降低功耗；

硬件原则

- 用HDL描述硬件进行数字系统，首先应该考虑的是硬件的实现；
- 程序的可读性，必须以硬件实现为前提；
- HDL描述的是硬件的结构和各个模块之间的连接关系，在设计前应对硬件本身有清晰的了解，然后用适当的HDL进行描述；
- 注意避免软件思维

系统原则

- 数字系统设计应该从宏观和系统全局的角度进行考虑；
 - 例如对于模块等的复用和合理组织所得到的效果远比对于小部分代码的反复推敲大；

系统原则

□设计前应对所用FPGA的底层硬件资源有所了解：

- 底层可编程硬件单元
 - Xilinx的为Slice
 - Altera的为LE
- Block RAM单元
- 布线资源
- 可配置IO单元
- 时钟资源

同步设计原则

- 在目前的条件下，采用异步电路设计并不理想，而现在的FPGA芯片都是为同步电路设计优化的；
- 单纯从IC设计角度看，同步电路比异步电路更加消耗资源；

同步设计原则

- 从资源考虑，关键要优化两种资源的比例；
- 另外，同步时序电路具有有没有毛刺、信号稳定等优点；
- 同步时序电路中延时的产生；
- 同步时序电路中输入的同步；

设计流程

- 设计、输入和综合
- 原理图
- 硬件描述语言
- 设计实现
- 生成比特流文件
- 设计验证
- 门级仿真
- 下载

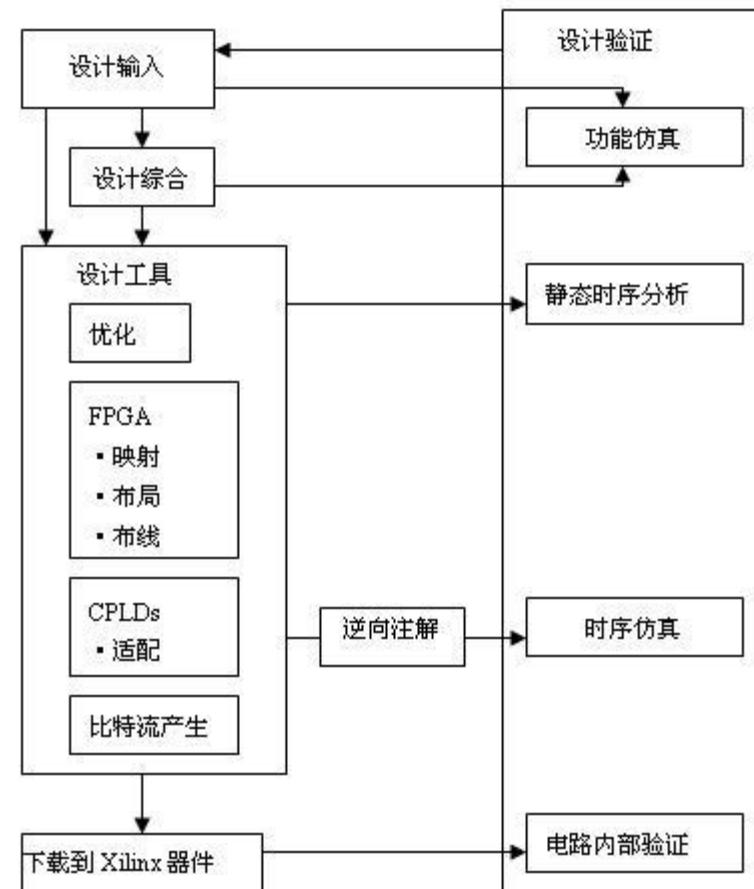


图 1.1 Xilinx 标准的设计流程

设计、输入和综合

- 方案论证、系统设计和FPGA芯片选择等准备工作
- 输入是将系统或电路以某种形式输入给EDA工具的过程
- 创建设计后可以进行功能仿真，也称为前仿真，是在编译之前对用户所设计的电路进行逻辑功能验证
- 综合就是将较高级抽象层次的描述转化成较低层次的描述

层次化设计对原理图和HDL都很重要

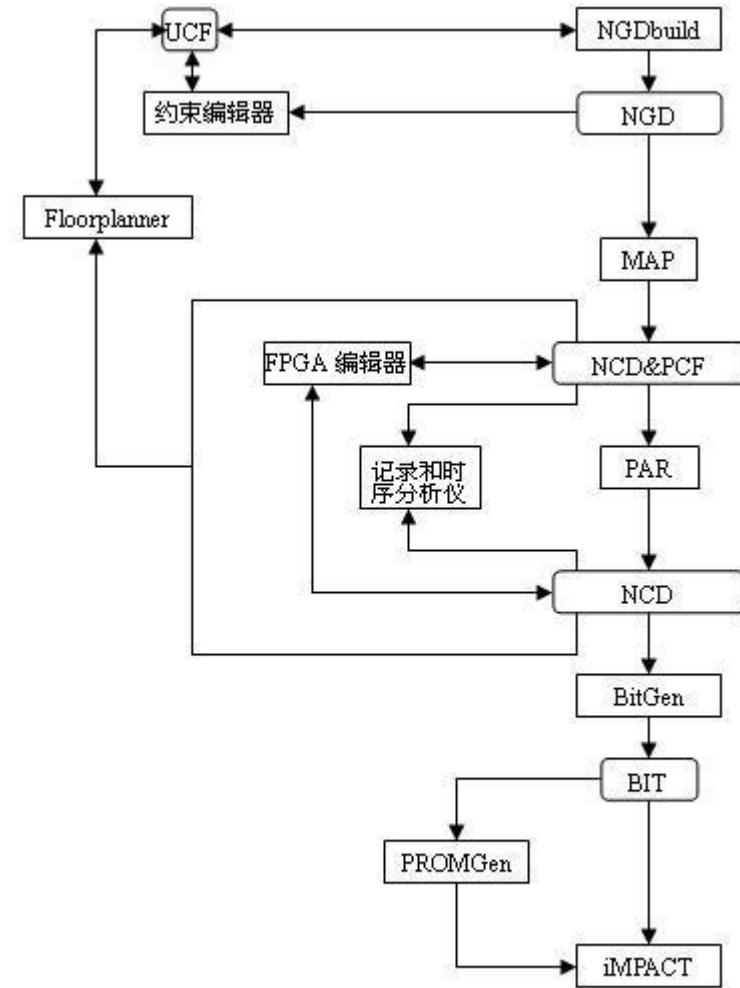
- 可以将设计概念化
- 将设计结构化
- 使调试设计更容易
- 使设计的不同部分的不同输入设计方法（原理图，HDL，本地编辑）能更容易结合
- 使更容易的更新设计，其中包括设计，实现，以及在设计过程中验证个别元件
- 减少优化时间
- 便于并行设计

约束

- 如果想要对设计中的时间参数或者布局参数进行约束，设计者可以指定映射、块布局、以及时间规范
- 映射约束
 - 指定逻辑块如何映射到可配置的逻辑块
- 模块布局
 - 块布局可限制在指定位置，可以是多个位置中的其中一个，或者是一个位置区域。
- 时序规范
 - 可以指定设计中路径的时间要求。

设计实现

从逻辑设计文件映射或适配到指定的器件开始，到物理设计布线成功并生成比特流文件时结束



设计仿真

□ 综合后仿真

- 把综合生成的标准延时文件反标注到综合仿真模型中去，可估计门延时带来的影响。

□ 时序仿真与验证

- 也称后仿真，是指将布局布线的延时信息反标注到设计网表中来检测有无时序违规板级仿真与验证

□ 板级仿真

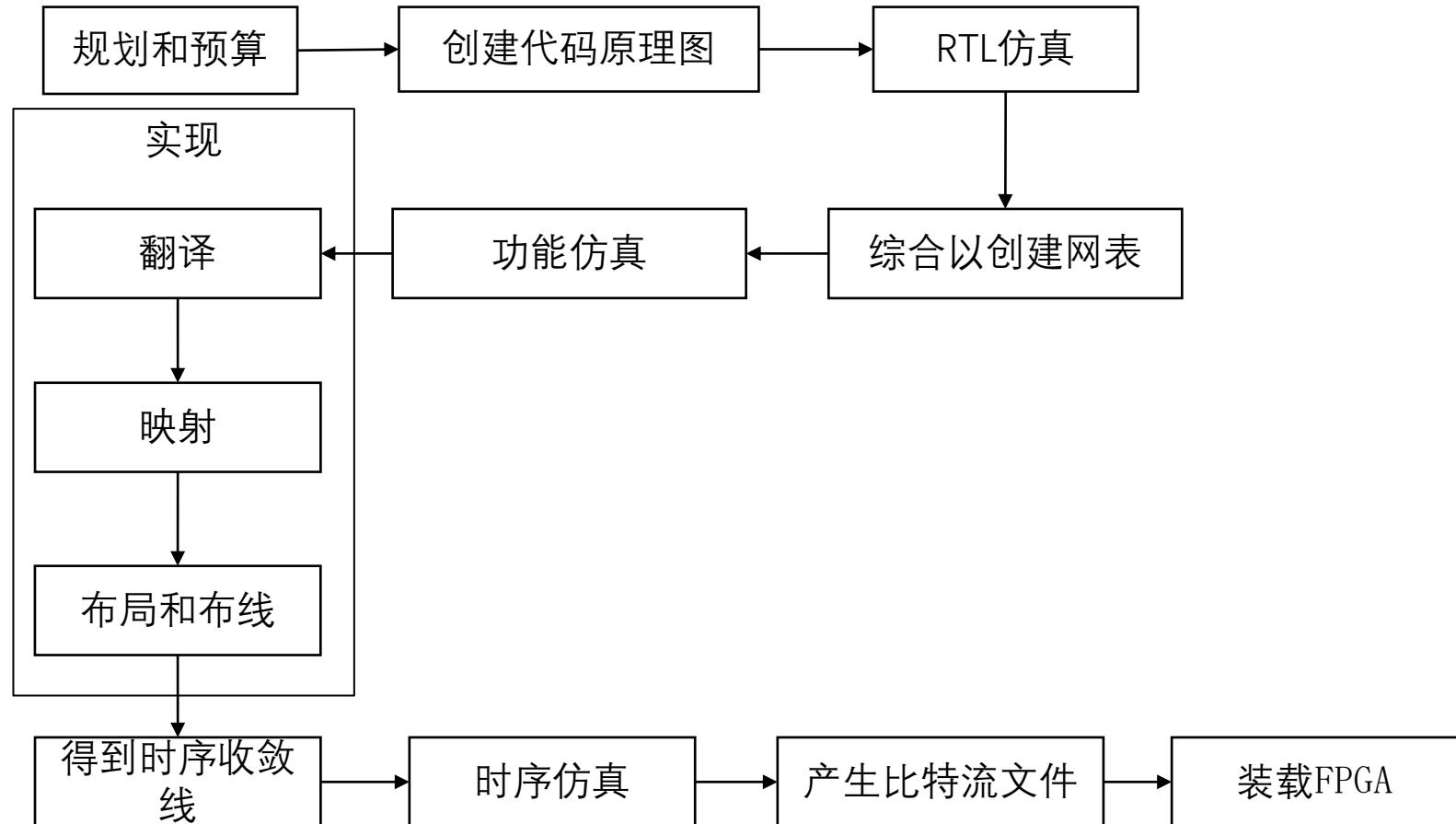
- 主要应用于高速电路设计中，对高速系统的信号完整性、电磁干扰等特征进行分析，一般都以第三方工具进行仿真和验证。

□ 芯片编程与调试

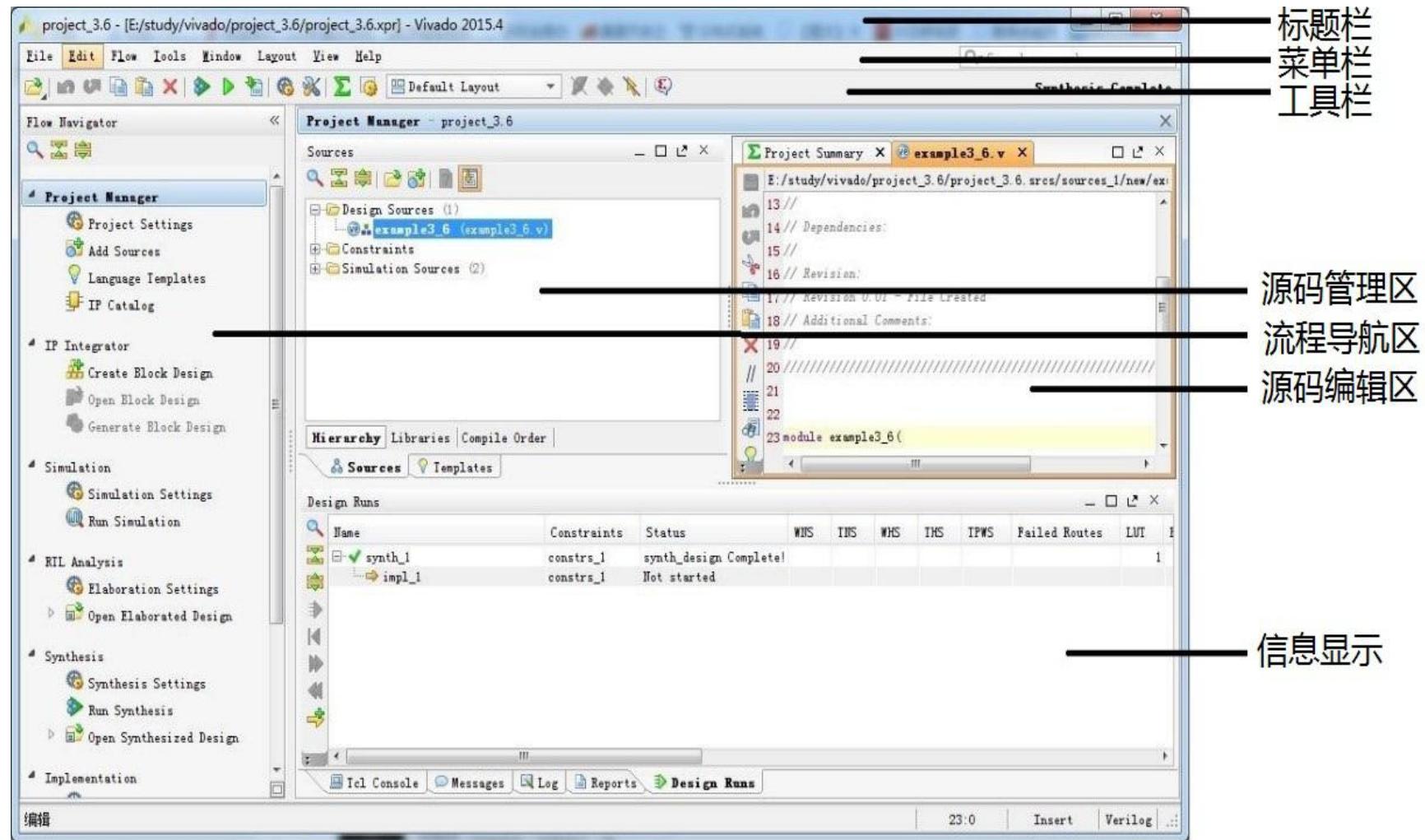
- 将编程数据下载到FPGA芯片中

Vivado进行硬件设计的 流程

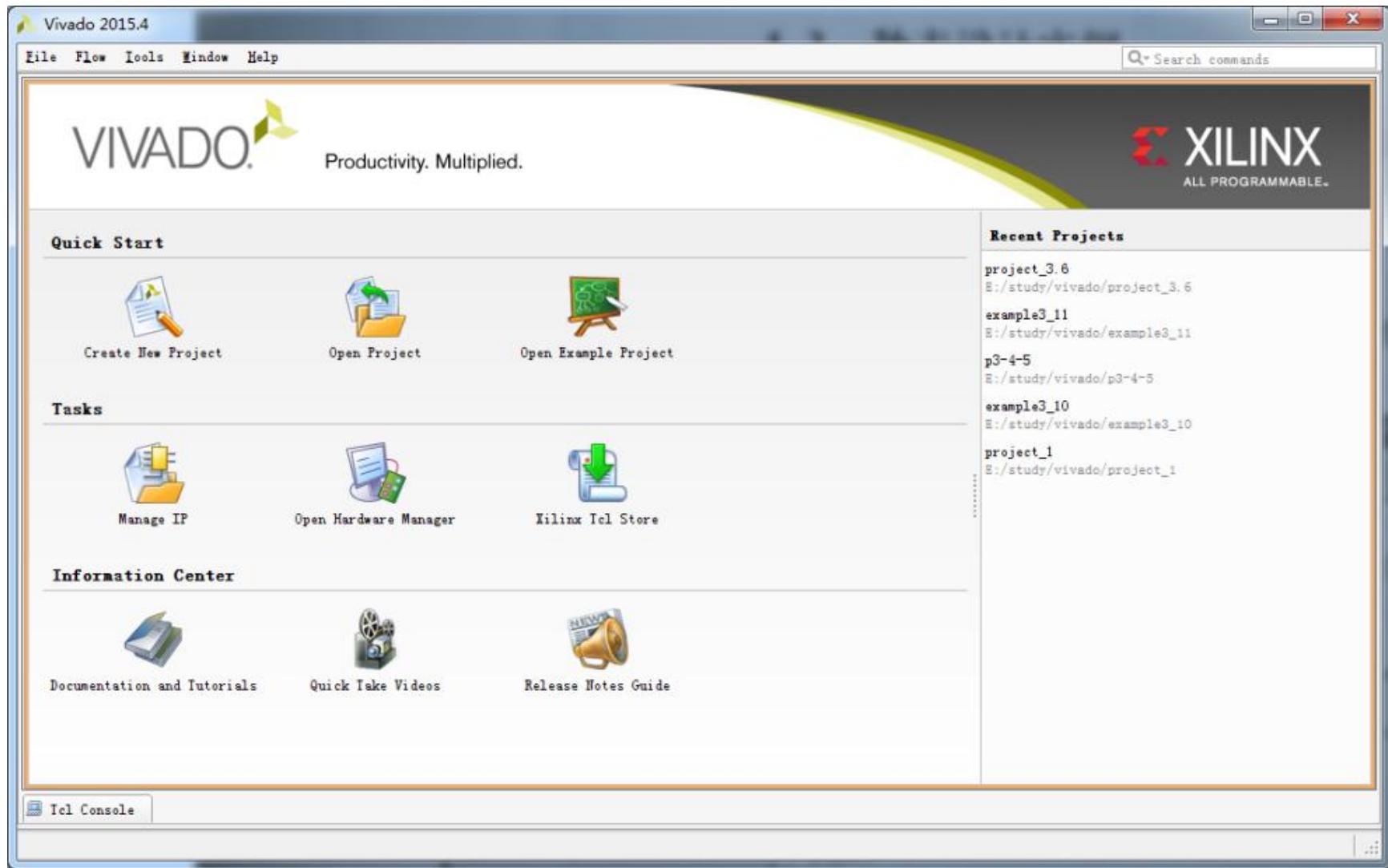
开发流程



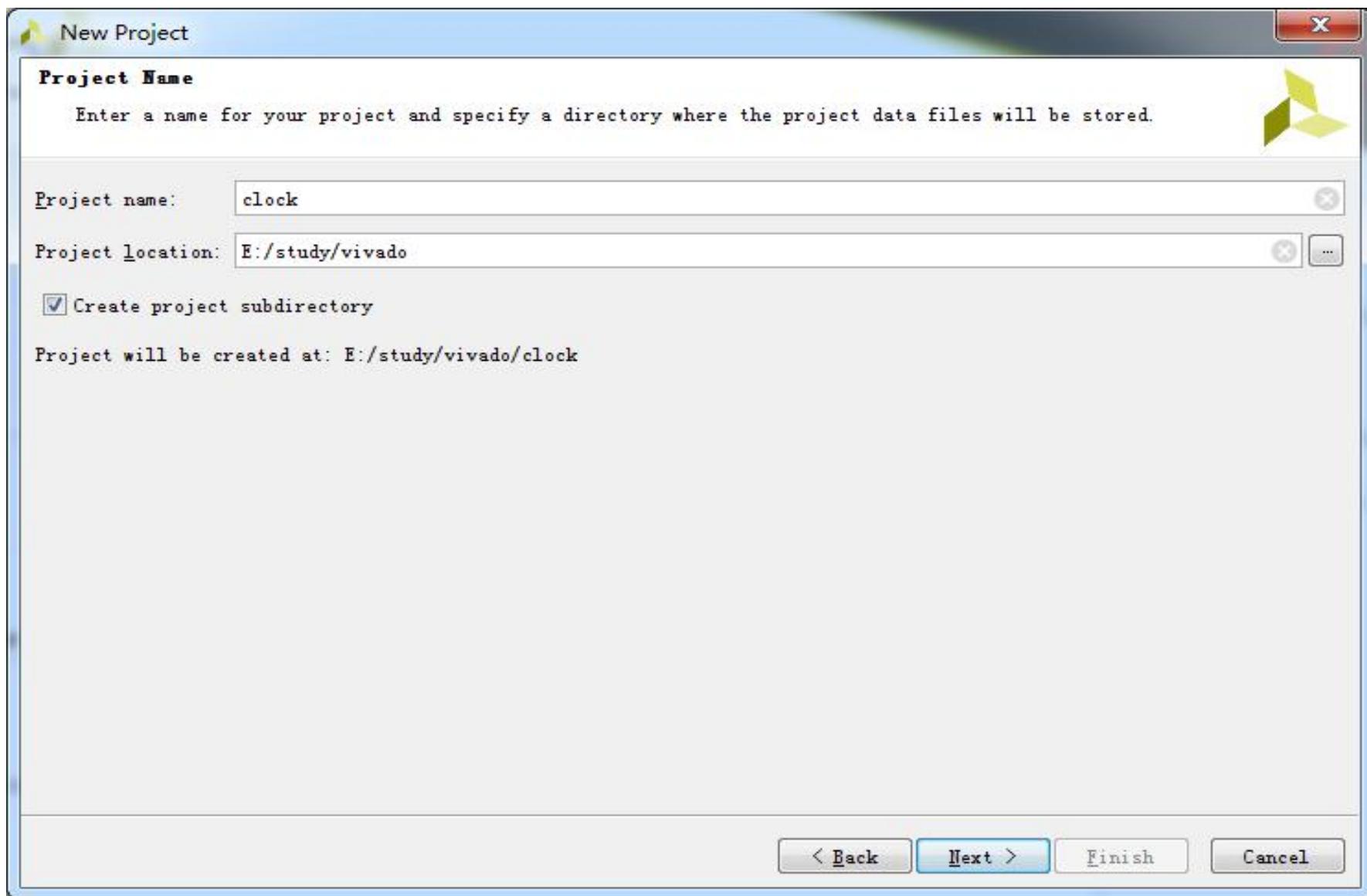
用户界面



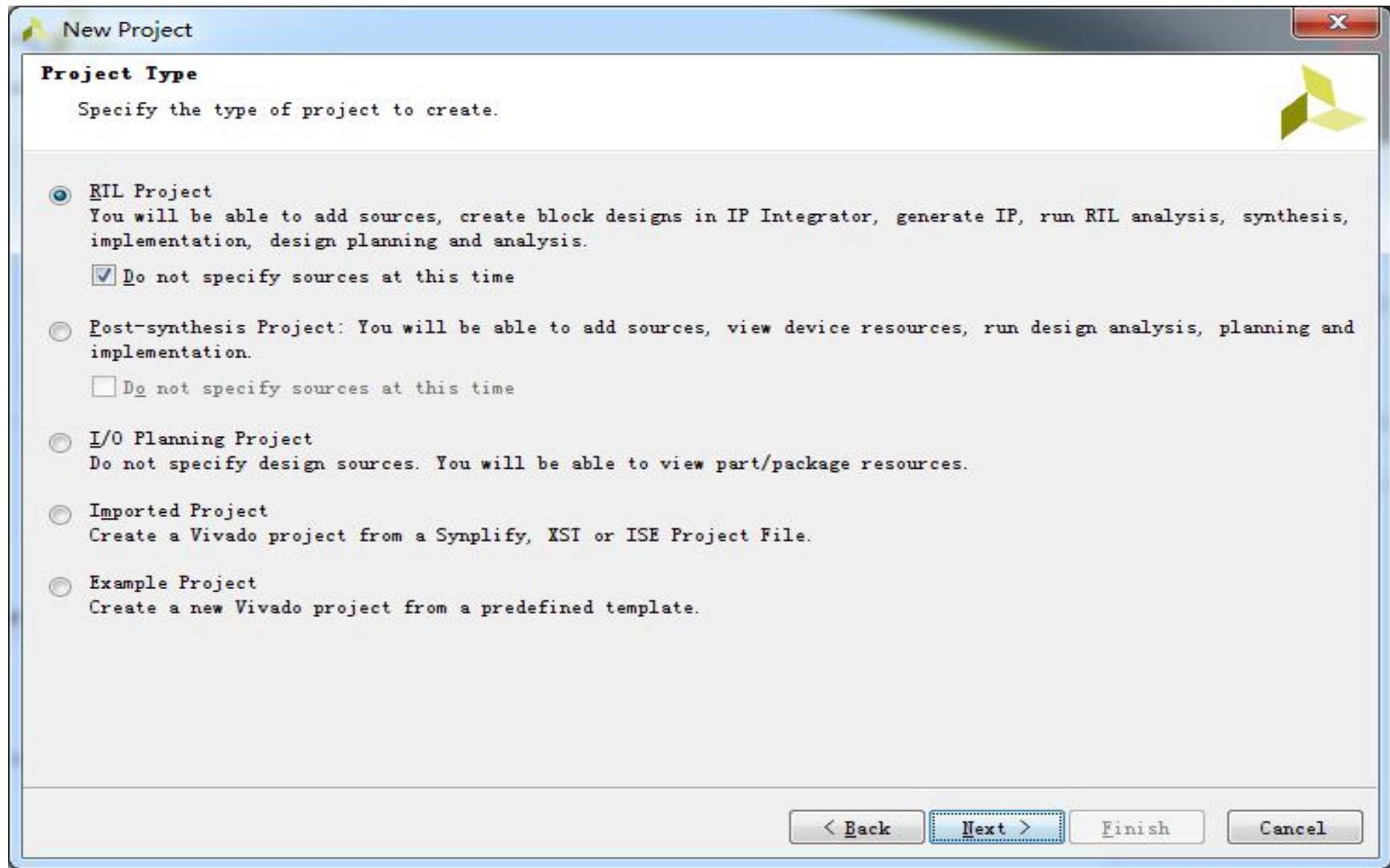
创建空白项目



设定项目名称和位置



设定项目类型



项目设定

New Project

Default Part
Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts Boards

Filter

Product category: All Speed grade: All

Family: All Temp grade: All

Package: All

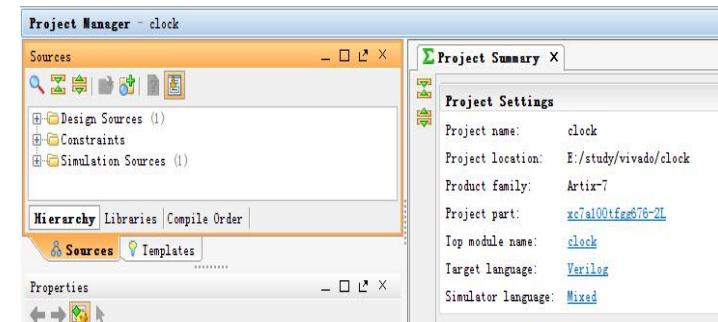
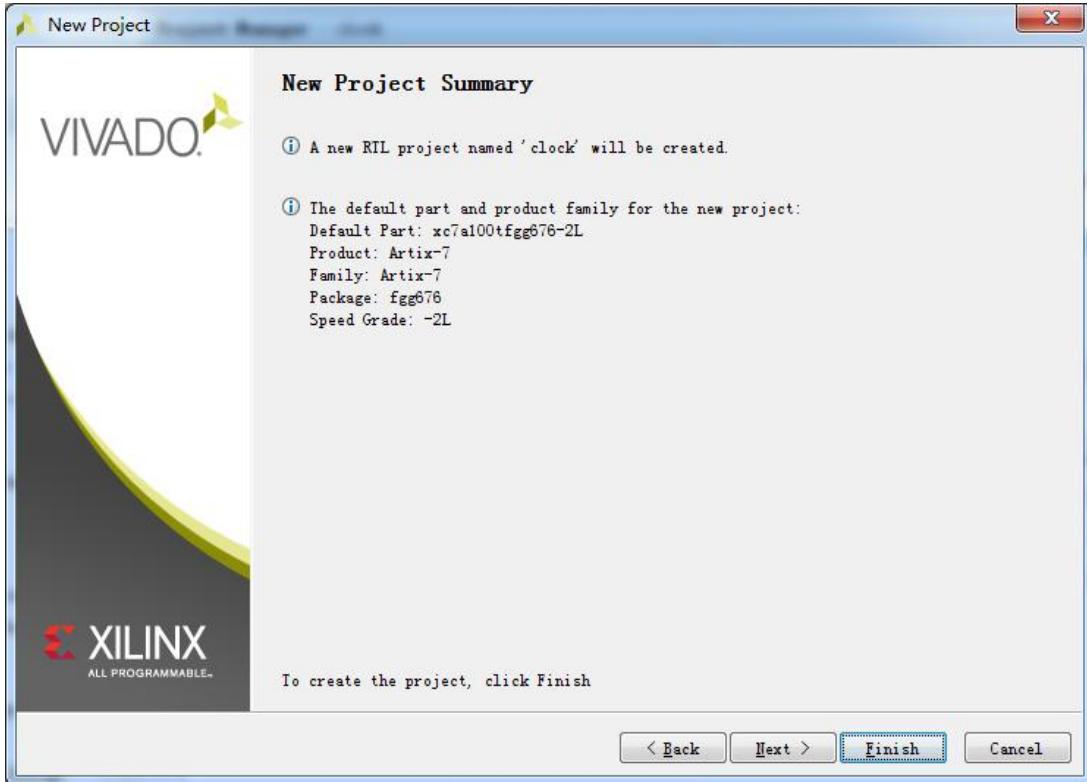
[Reset All Filters](#)

Search: (64 matches)

Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GIPE2 Transceivers	Gb Transceivers	Available IOBs
xc7a75tfgg676-2L	676	105	180	94400	8	8	300
xc7a75t1fgg676-2L	676	105	180	94400	8	8	300
xc7a100tfgg676-2L	676	135	240	126800	8	8	300
xc7a100t1fgg676-2L	676	135	240	126800	8	8	300
xc7a200tfbg676-2L	676	365	740	269200	8	8	400
xc7a200tfbv676-2L	676	365	740	269200	8	8	400
xc7a200t1fbv676-2L	676	365	740	269200	8	8	400
xc7a200t1fbg676-2L	676	365	740	269200	8	8	400
...

< Back [Next >](#) Finish Cancel

项目信息汇总与空白项目

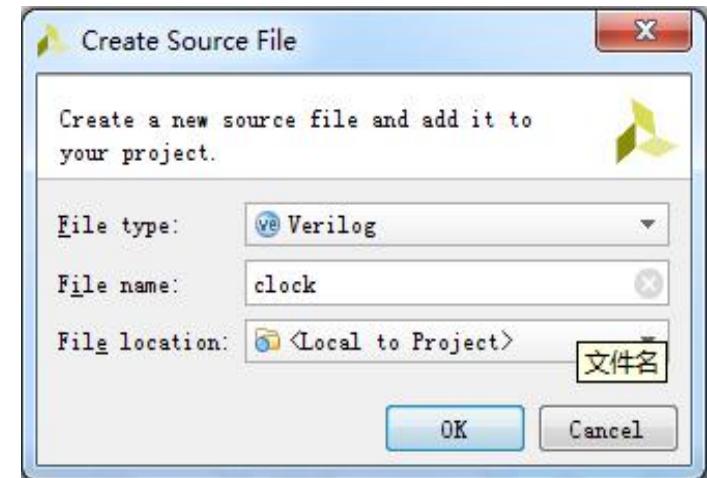
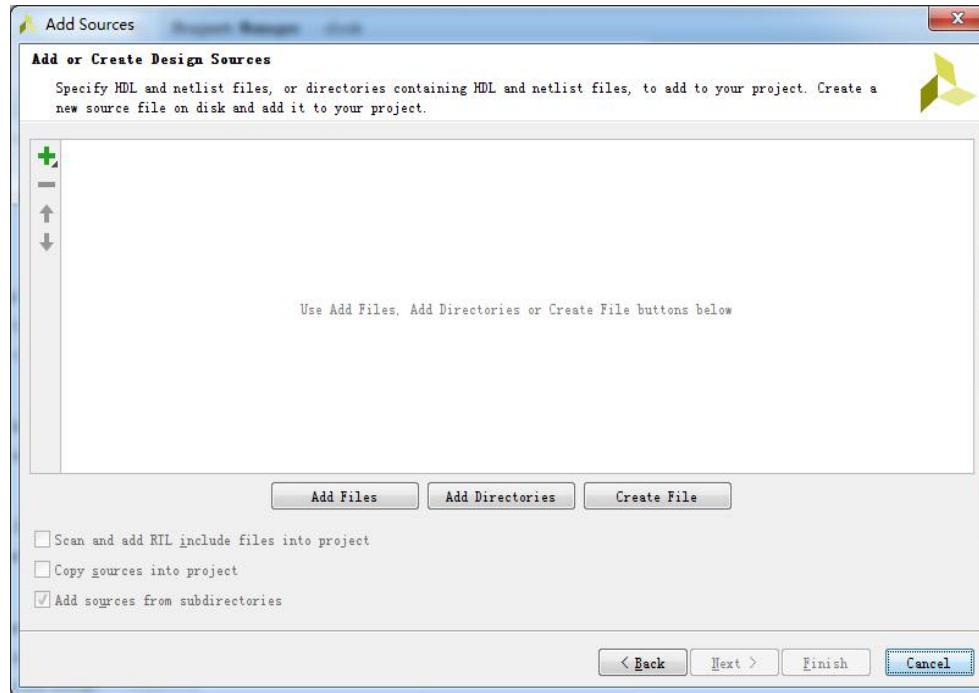


添加源文件

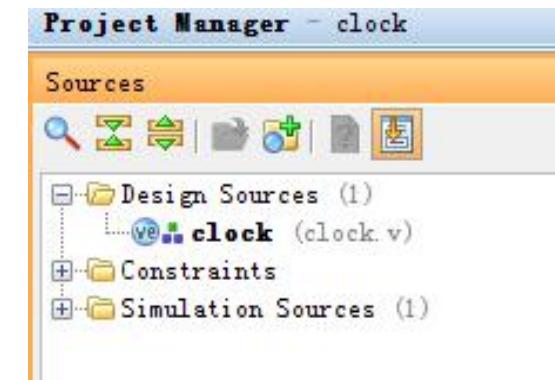
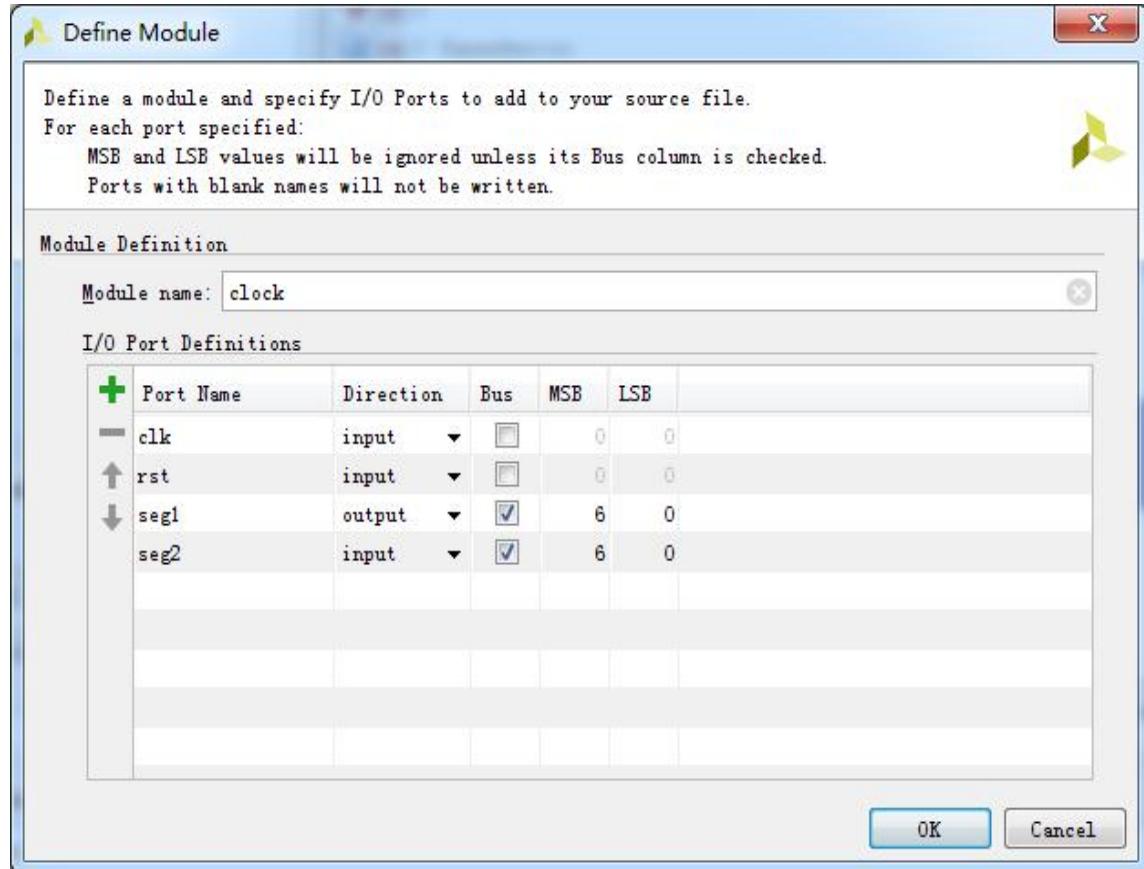
□ 在左边管理区单击Add Sources，选择 New Source，出现创建源文件向导的选择源码类型对话框，在此选择 Add or Create design sources。



添加文件， 创建新文件

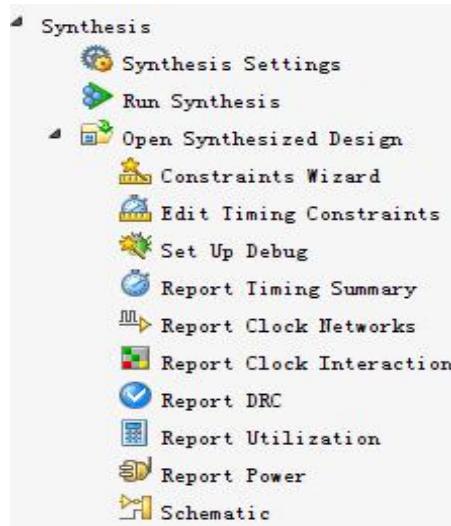


模块定义



新文件添加完成

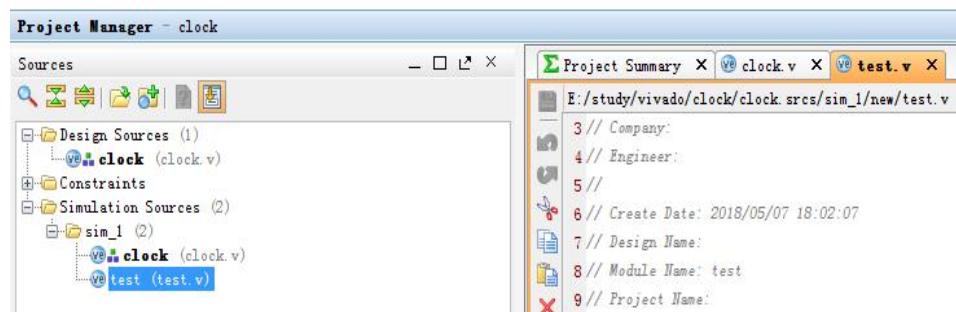
综合与仿真



综合操作

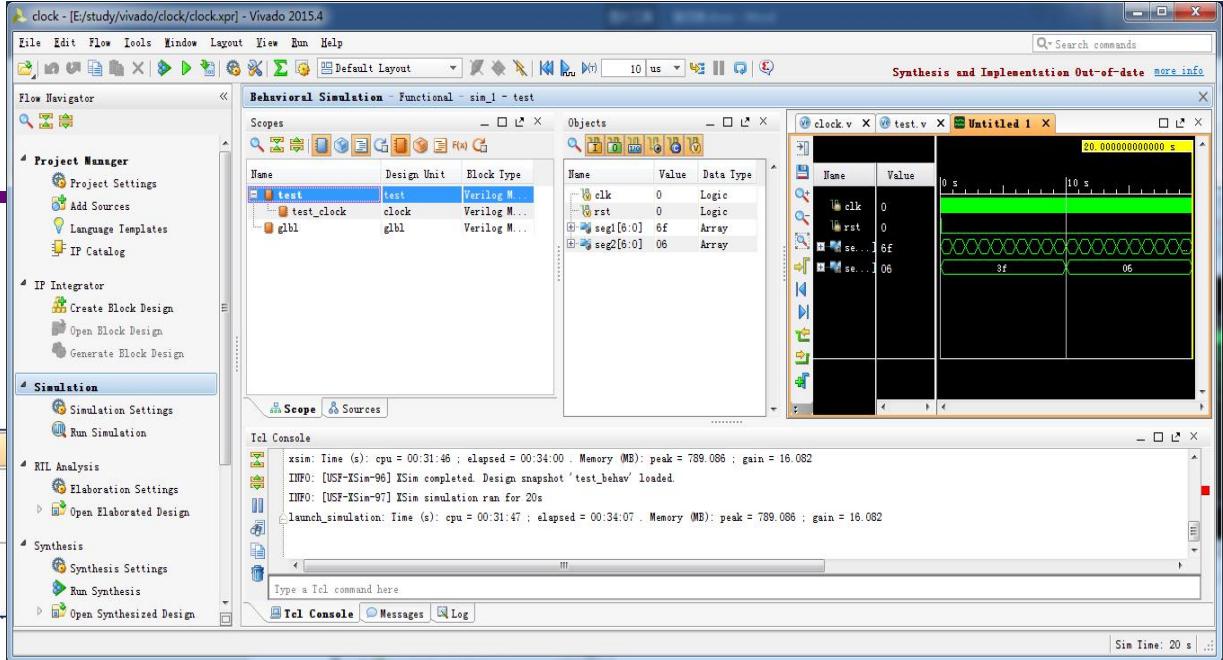
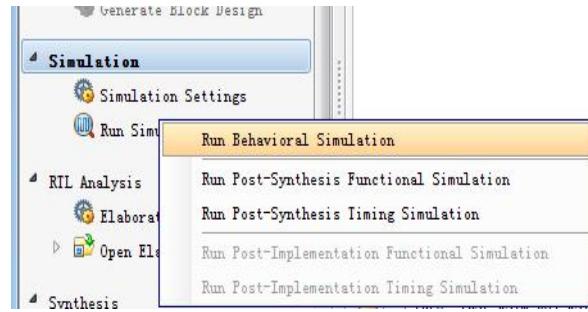


创建仿真代码



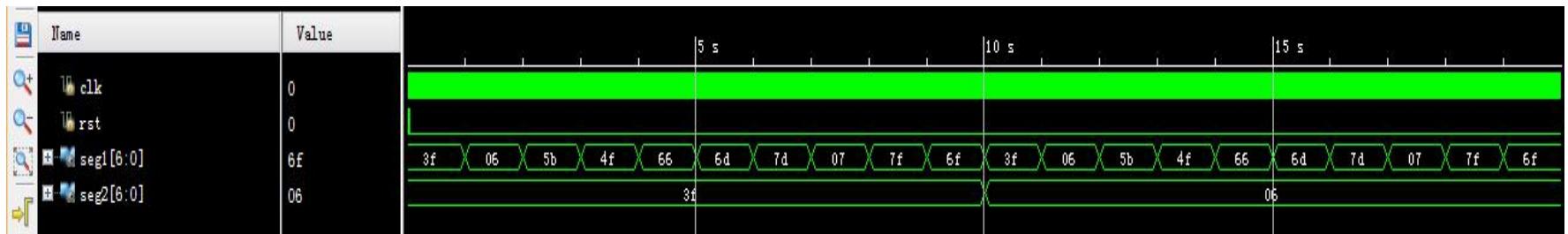
仿真源码创建完成

仿真



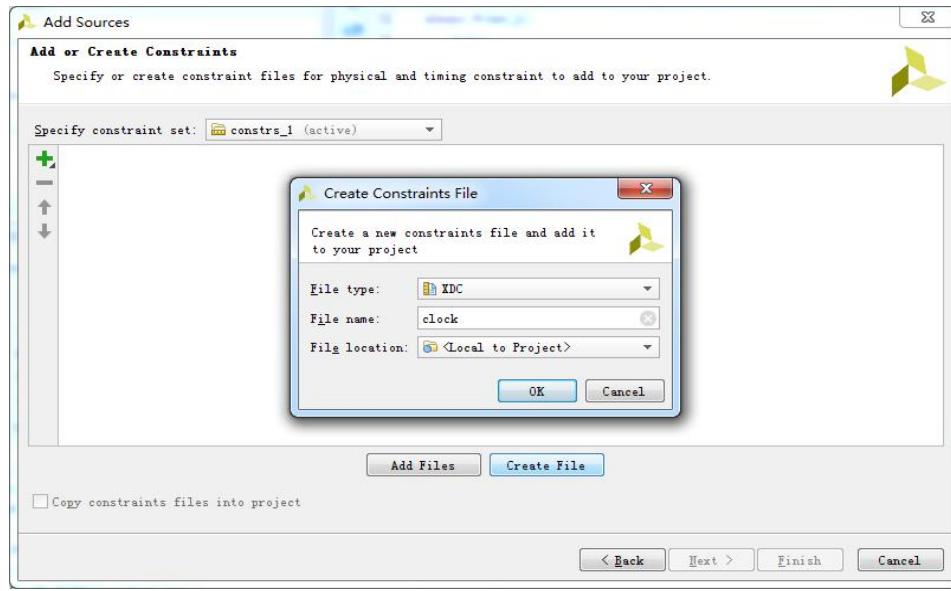
仿真开始

仿真结果窗口



仿真结果

添加约束（样例项目已经有约束）



添加约束文件

The screenshot shows the Vivado IDE's 'Sources' panel and the code editor for the 'clock.xdc' file. The 'Sources' panel displays a tree structure with 'Design Sources' containing a 'clock' item, 'Constraints' containing a 'constrs_1' item which has a 'clock.xdc' target, and 'Simulation Sources' containing a 'sim_1' item with a 'test' item. The code editor shows the XDC file content:

```
E:/study/vivado/clock/clock.srs/constrs_1/new/clock.xdc
1 #Clock
2 set_property -dict {PACKAGE_PIN D18 IOSTANDARD LVCMS33} [get
3 set_property -dict {PACKAGE_PIN F22 IOSTANDARD LVCMS33} [get
4 #DPTO
5 set_property IOSTANDARD LVCMS33 [get_ports seg1[*]]
6 set_property PACKAGE_PIN F15 [get_ports {seg1[2]}]
7 set_property PACKAGE_PIN H15 [get_ports {seg1[3]}]
8 set_property PACKAGE_PIN G15 [get_ports {seg1[4]}]
9 set_property PACKAGE_PIN M16 [get_ports {seg1[1]}]
```

编辑约束文件

实现，生成可配置文件



实现按钮



生成可配置文件 (bit文件)

本地装载

上传设计文件

选择文件 未选择任何文件
请选择.bit设计文件

写入实验FPGA

串口参数设置

串口选择
CPLD串口控制器

▲开发板每次重启后需要重新选择

应用

将板上的 Micro USB 接口连接到电脑，可检测到一个虚拟串口。如果在Windows系统上不能自动识别，可尝试下载驱动文件，并用更新驱动程序向导安装。

Flash与RAM读写

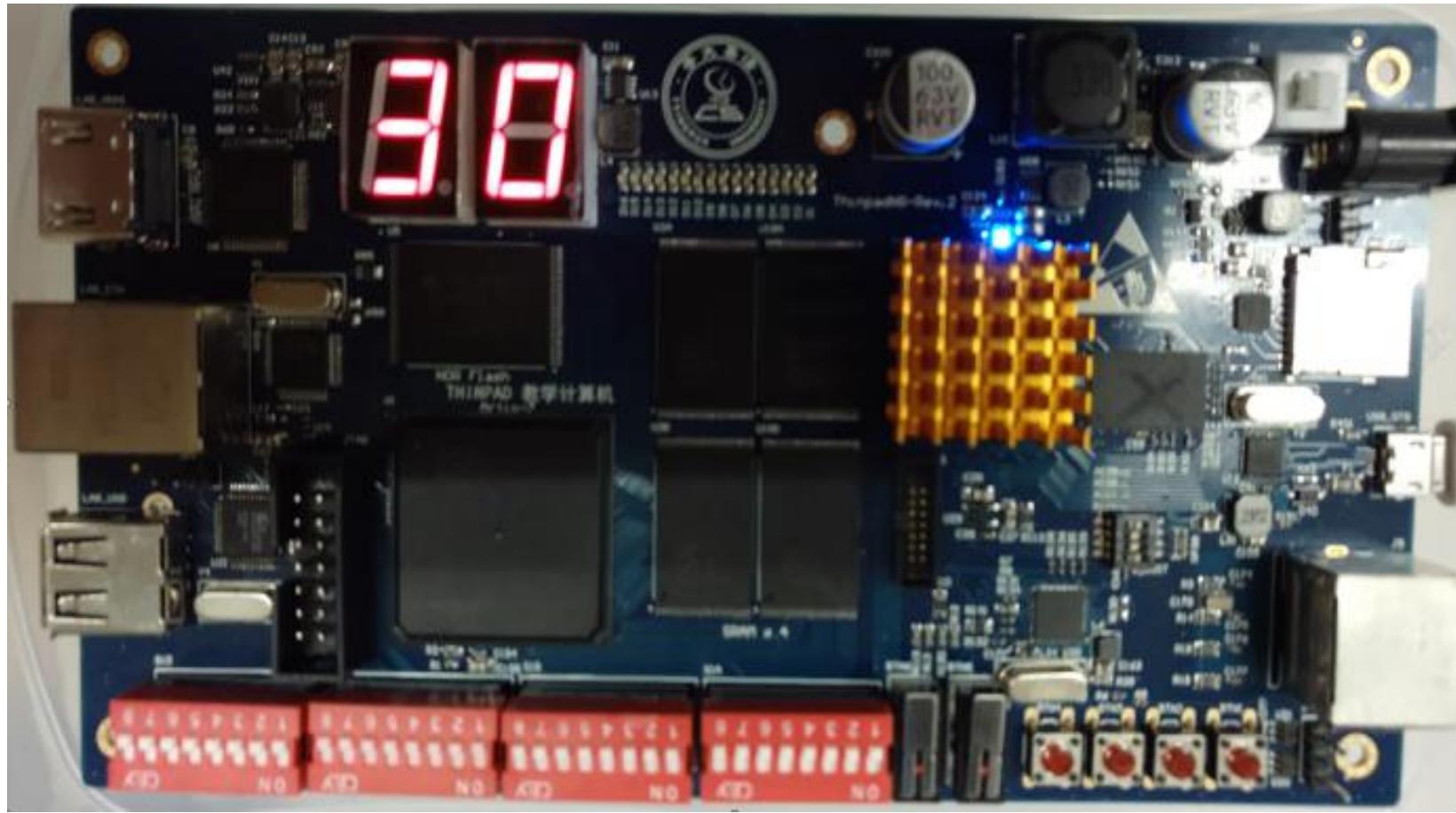
存储选择 Flash 容量 8MB

起始地址 0x (Byte)
起始地址应当按16位对齐（即为偶数）

读取数据 0x 读取长度 (Byte) 读取
读取长度应当按16位对齐（即为偶数）

写入数据 选择文件 未选择任何文件 写入
写入长度为数据文件大小，按16位对齐（即为偶数）

本地执行



远程装载

ThinpadCloud 文件上传 工作区域 admin 登出

Thinpad rev.3

ThinPAD-Cloud 教学计算机硬件平台，硬件版本3。

参考文件：
[工程模板下载（含引脚约束）rev.3](#)

开发板分配 自动分配 ▾

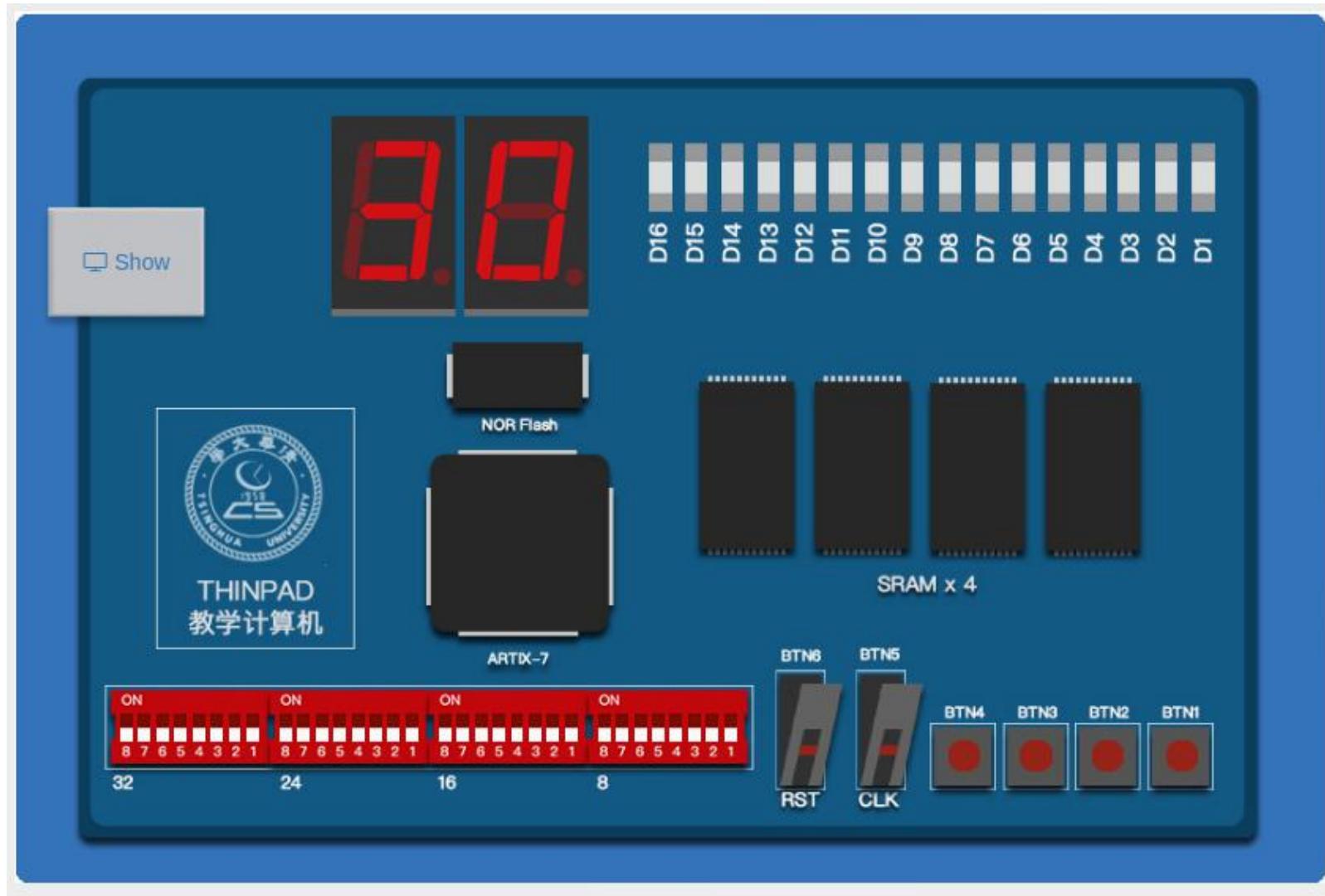
Bitstream [选择文件](#) clock.bit
编译得到的 .bit 文件

文件信息 clock;UserID=0xFFFFFFFF;Version=2018.3

编译时间 2019/06/24 16:51:45

上传并开始

远程执行



参考网址

□ <https://www.fpga4fun.com/>

□ <https://timetoexplore.net/>

□ 上面两个网站有比较复杂的例子和详细的器件说明

□ 没有学过数字逻辑课程的同学，请务必尽快自学这部分相关的内容，注意下面的基本概念

- 注意组合逻辑的特点
- 注意时序逻辑的特点
- 时钟起到什么作用
- 什么是时延

谢谢



硬件描述语言 Verilog and SystemVerilog

2022年秋

内容

- Verilog简介
 - 组合逻辑硬件描述
 - 时序逻辑时序逻辑硬件描述
-
- 大部分内容来自于Onur Mutlu教授（ETH）的上课内容

为什么需要硬件描述语言？

□ 硬件极其复杂，不可能直接用门电路搭接出最终的硬件电路，通过硬件描述语言进行抽象，使得对硬件的描述成为了可能

- 晶体管（门级），连线

□ 硬件描述语言

- 可用于描述复杂的硬件设计
- 可用于行为仿真（包括功能和时序）
- 可用于硬件的综合

□ 硬件描述语言被设计出来完成上述的目的

- 不同的硬件描述语言有很多的相似性（VHDL, Verilog），完成相同的目标
- 语言功能通常可以直接映射，特别是对于常用的子集
- Verilog（SystemVerilog）在工业界常用

硬件设计的准则——层次化设计

<https://techreport.com/review/21987/intel-core-i7-3960x-processor>

□ 按照层次来组织硬件模块

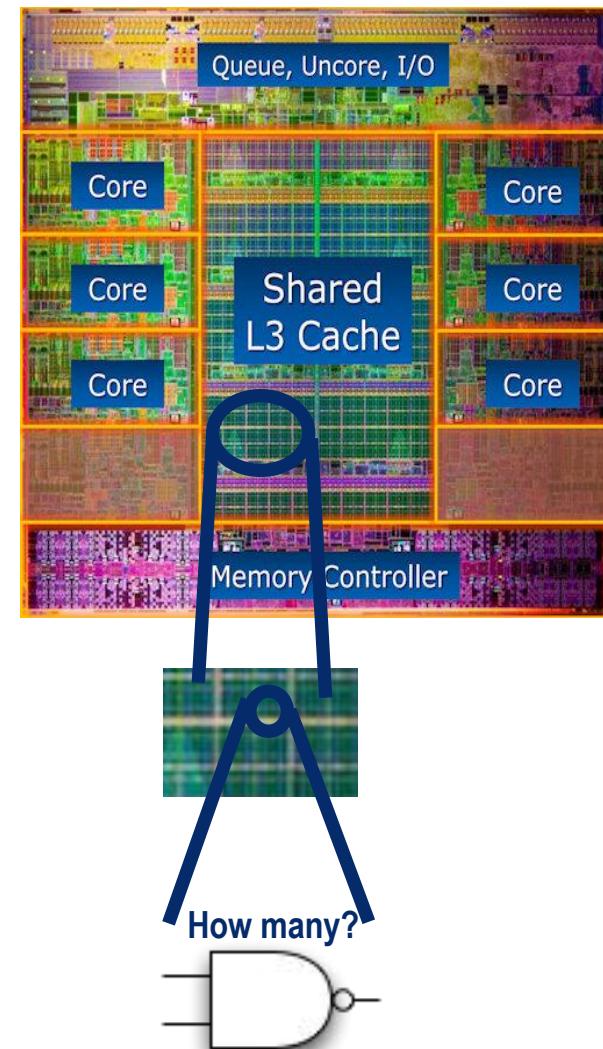
- 底层为“基本原语”门 (AND, OR, ...)
- 通过基本门的原件例化来实现简单模块 (e.g., 例如多路选择器 MUXes)
- 通过简单模块例化来实现复杂模块...

□ 层次化设计控制了复杂性

- 类似于编程中的函数/方法的抽象

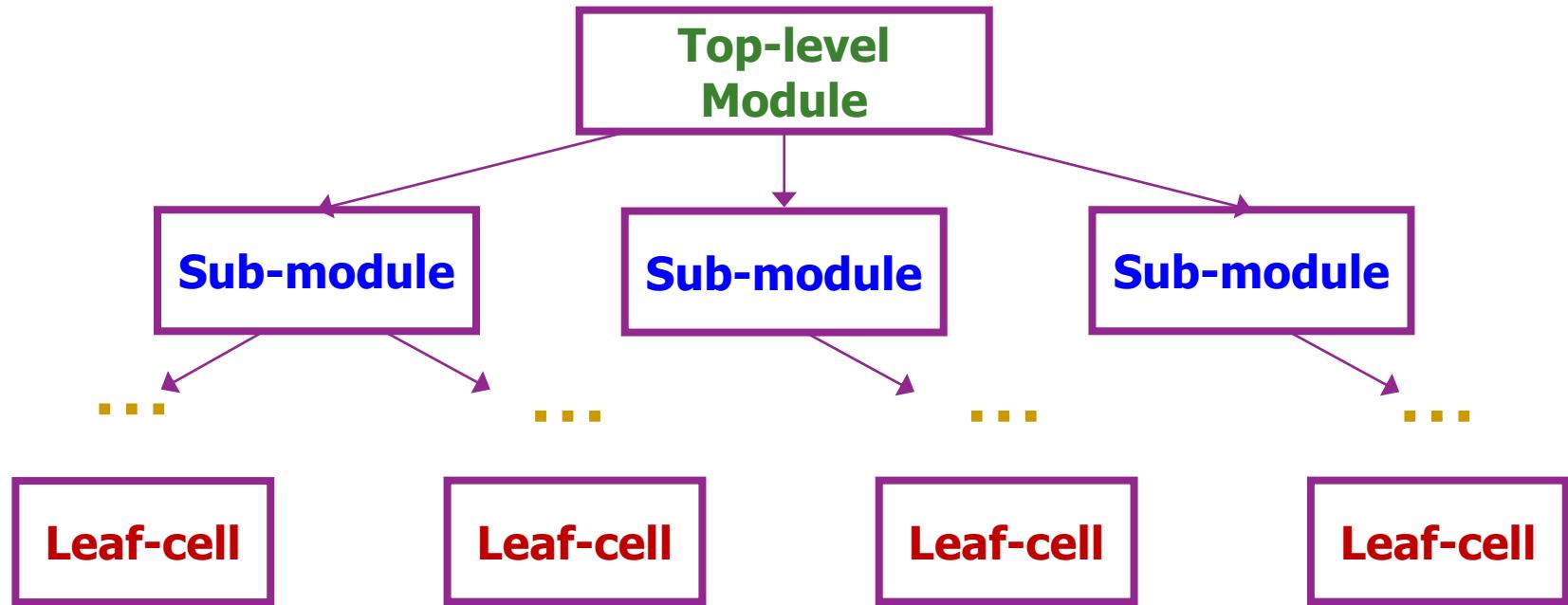
□ 复杂性是一个很大的问题

- Sandy Bridge-E有2.27B个晶体管



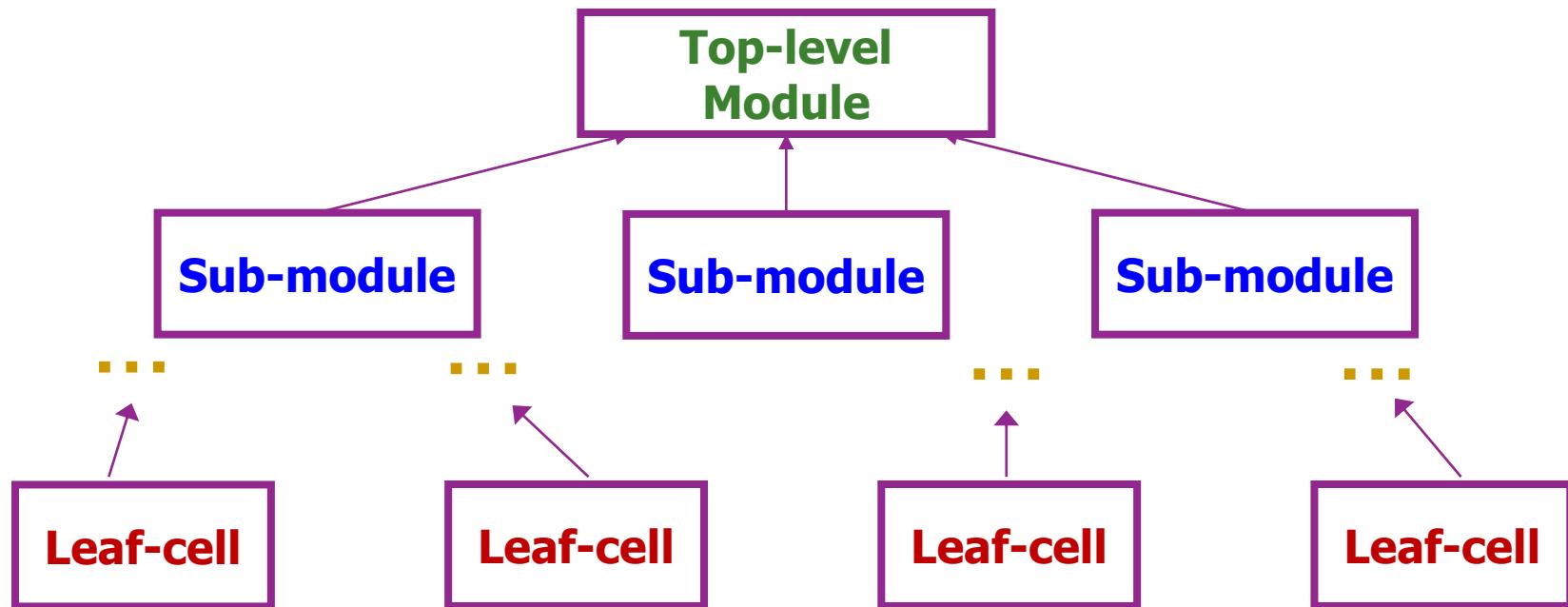
自顶向下的设计方法

- 定义顶层模块，并确定构建顶层模块所需的子模块
- 对子模块进行细分，直到最基本的门级电路(叶子单元 leaf-cell)
 - Leaf cell: 不能进一步分割的电路元件(例如，逻辑门、原始单元库元件等)。



自底向上的设计方法

- 从基本模块开始构建
- 逐步从基本模块向上构造更加复杂的模块
- 直到最终设计完成最上层的顶层模块
- 两种方法不是互斥的，实际设计中需要综合使用



Verilog中定义一个模块

- 模块module是verilog(systemverilog)中的基本编程单元
- 首先是进行模块的定义：
 - 模块的名字
 - 模块端口的名字
 - 模块端口的方向（输入端口还是输出端口）
- 在模块定义的基础上描述模块的功能

example

模块实现

example

```
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

    // here comes the circuit description

endmodule
```

模块的名字

端口的列表
(输入输出端口)

端口的类型

模块的定义

端口接口定义

口下面的代码是等价的

```
module test ( a, b, y );
    input a;
    input b;
    output y;

endmodule
```

```
module test ( input a,
              input b,
              output y );

endmodule
```

端口名字和方向可以放在一起

信号数组

口 多位的输入输出Input/Output (Bus)

- [range_end : range_start]
- 位数: range_end – range_start + 1

口 举例:

```
input  [31:0] a;    // a[31], a[30] .. a[0]
output [15:8] b1;   // b1[15], b1[14] .. b1[8]
output [7:0]  b2;   // b2[7], b2[6] .. b2[0]
input      c;     // single signal
```

- a 代表了一个32位的值 [31:0] a
- 通常使用 [31:0] 比 [0:31]普遍
- 无论如何定义，要保持在程序中是一致的

位操作

```
// You can assign partial buses
wire [15:0] longbus;
wire [7:0] shortbus;
assign shortbus = longbus[12:5];

// Concatenating is by {}
assign y = {a[2],a[1],a[0],a[0]};

// Possible to define multiple copies
assign x = {a[0], a[0], a[0], a[0]};
assign y = { 4{a[0]} };
```

基本的语法

- 区分大小写
- 标识符不可以以数字开始（和c语言一致）
- 空白字符忽略
- 注释

```
// Single line comments start with a //

/* Multiline comments
   are defined like this */
```

Verilog两种基本实现风格

□ 结构描述(门级描述)

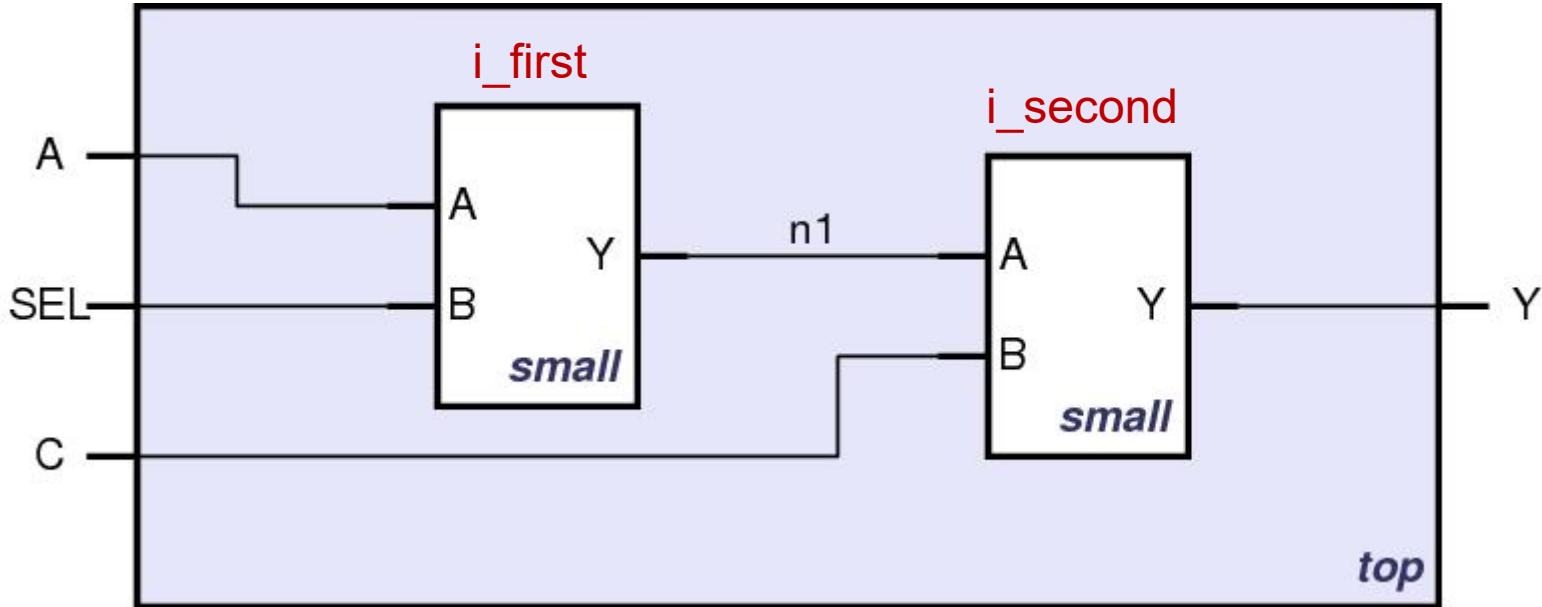
- 更像进行硬件的设计
- 模块主体包含电路的门级描述
- 描述模块是如何相互连接的
- 每个模块包含其它模块（实例）以及这些模块之间的互连关系

□ 行为描述

- 描述模块的行为，而不设计具体的模块
- 包含逻辑和数学运算符
- 抽象水平高于门级描述
 - 相同的行为描述会有多种可能的门级实现方式

□ 实际的系统会同时使用上述的两种实现方式

结构描述：实例化一个模块



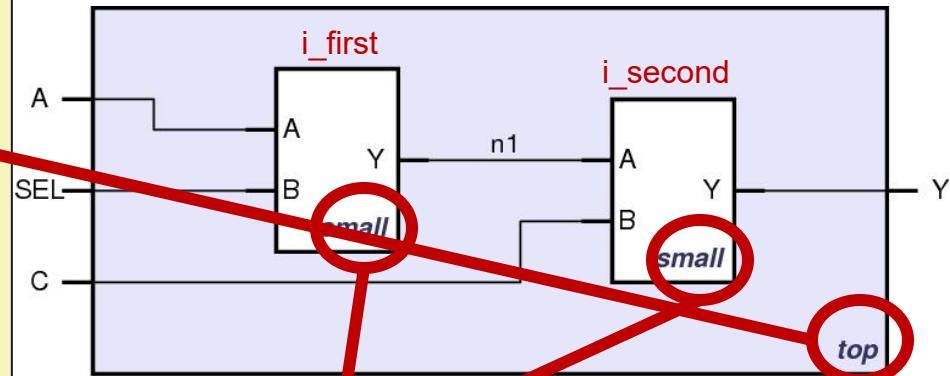
top的模块由两个small模块组成，上图是它们直接的连线关系

结构描述的举例1

模块的定义

```
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    endmodule
```



```
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small

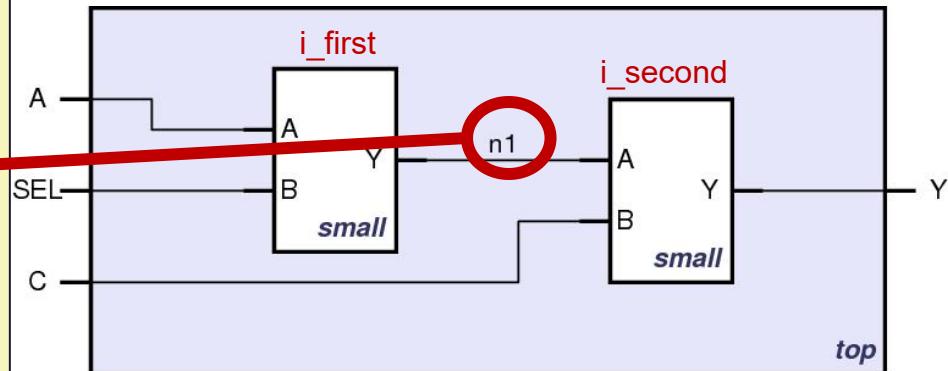
    endmodule
```

结构描述的举例2

口定义连线关系 (模块互联)

```
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;
```

endmodule



```
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small

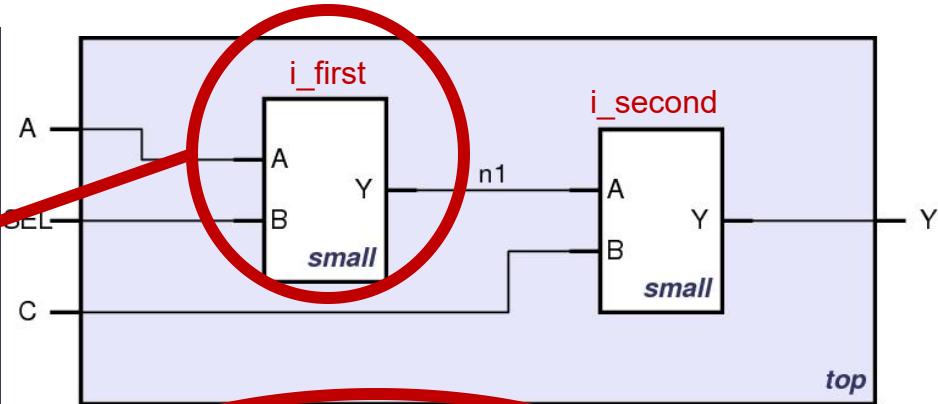
endmodule
```

结构描述的举例3

第一个例化的“small”模块

```
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // instantiate small once
    small i_first (.A(A),
                  .B(SEL),
                  .Y(n1));
endmodule
```



```
module small (A, B, Y);
    input A;
    input B;
    output Y;
    // description of small
endmodule
```

结构描述的举例4

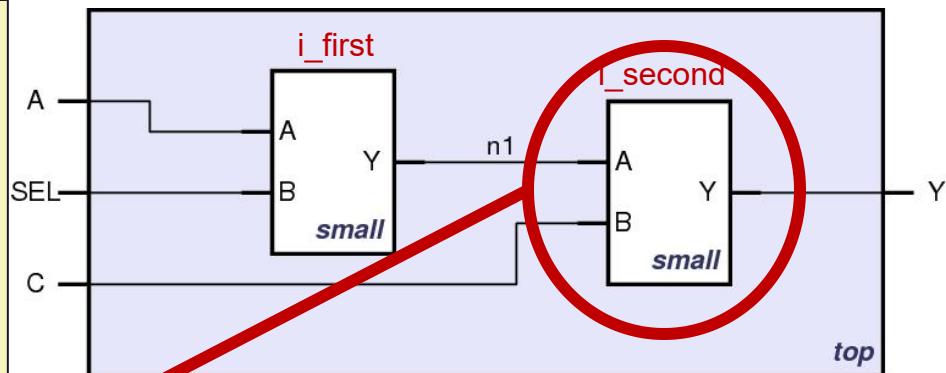
口第二个例化的“small”模块

```
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // instantiate small once
    small i_first (.A(A),
                  .B(SEL),
                  .Y(n1) );

    // instantiate small second time
    small i_second (.A(n1),
                    .B(C),
                    .Y(Y) );

endmodule
```



```
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small

endmodule
```

结构描述的举例5

模块实例化的简短形式

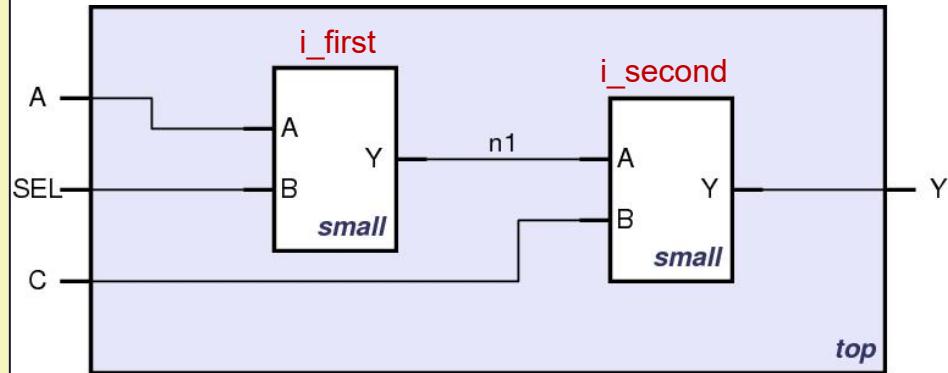
```
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // alternative short form
    small i_first ( A, SEL, n1 );

    /* In short form above,
       pin order very important */

    // safer choice; any pin order
    small i_second ( .B(C),
                      .Y(Y),
                      .A(n1) );

endmodule
```



```
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small

endmodule
```

简短的形式不是好的做法
降低了代码的可维护性

结构描述的举例6

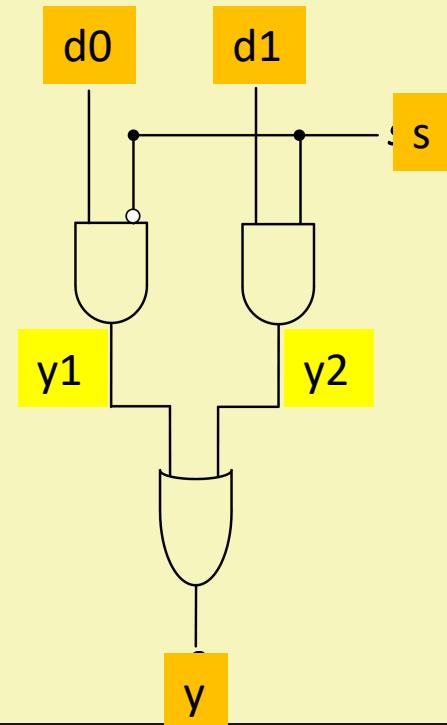
支持基本的逻辑门作为预定义的基本模块

- 这些基本模块像其它模块一样被实例化，它们在Verilog中被预定义，不需要模块定义

```
module mux2(input d0, d1,
             input s,
             output y);
    wire ns, y1, y2;

    not g1 (ns, s);
    and g2 (y1, d0, ns);
    and g3 (y2, d1, s);
    or  g4 (y, y1, y2);

endmodule
```



行为描述

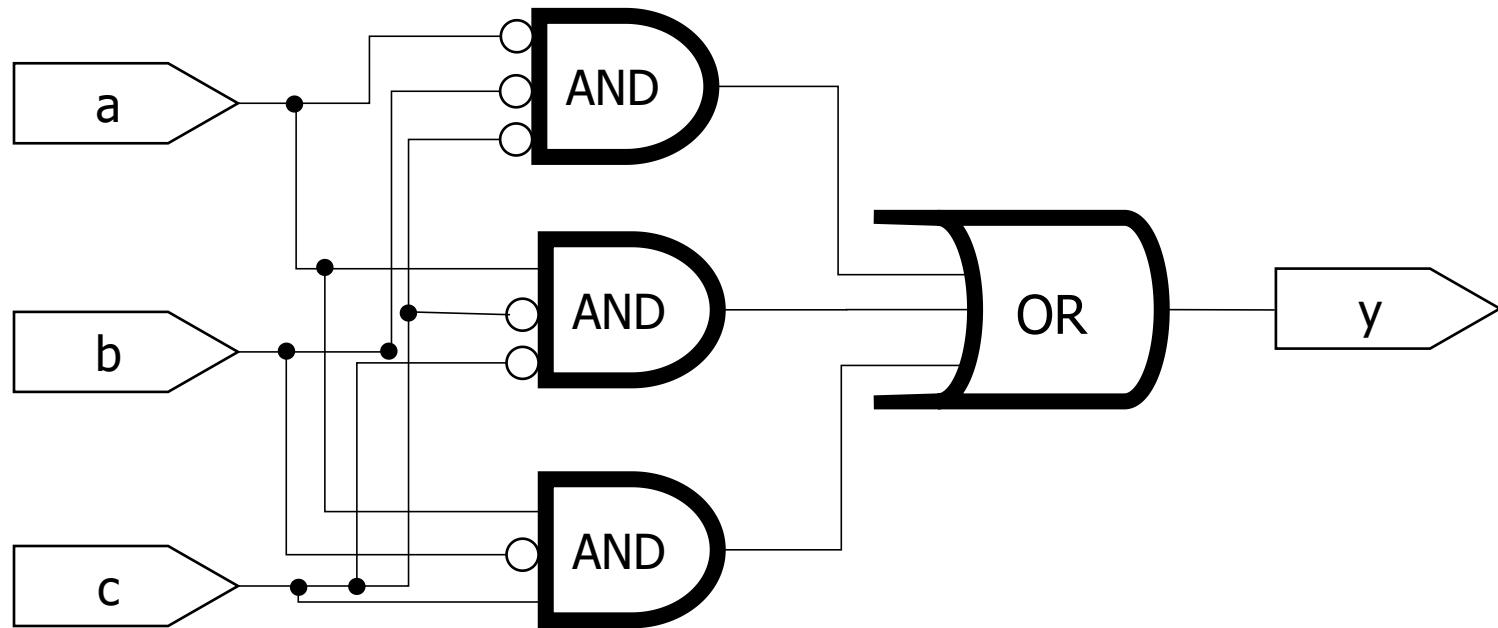
```
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

// here comes the circuit description
assign y = ~a & ~b & ~c |
           a & ~b & ~c |
           a & ~b &   c;

endmodule
```

行为描述：实现结果

行为描述同样定义了硬件电路的模型



行为描述中的位操作

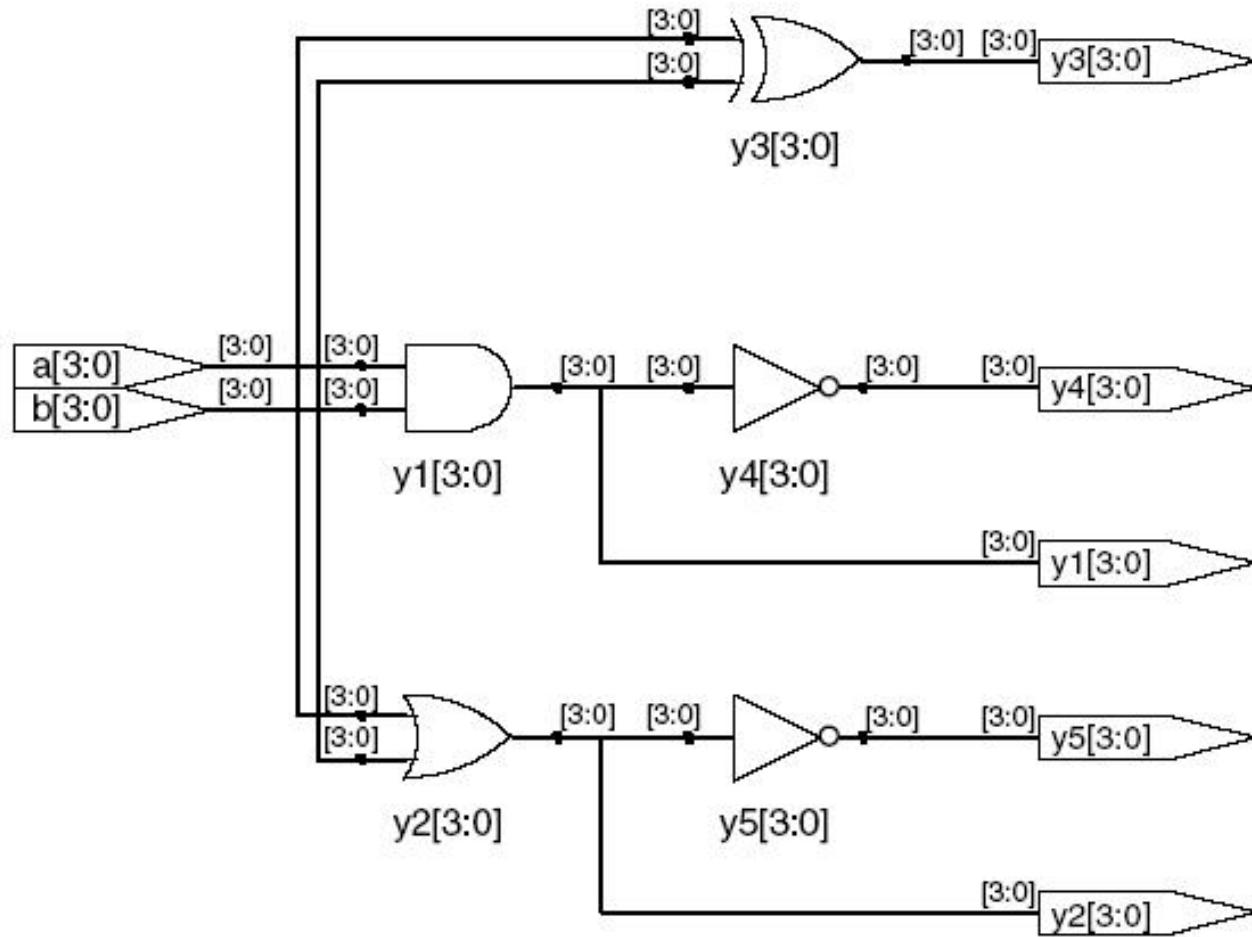
```
module gates(input [3:0] a, b,
              output [3:0] y1, y2, y3, y4, y5);

    /* Five different two-input logic
       gates acting on 4 bit buses */

    assign y1 = a & b;      // AND
    assign y2 = a | b;      // OR
    assign y3 = a ^ b;      // XOR
    assign y4 = ~(a & b);  // NAND
    assign y5 = ~(a | b);  // NOR

endmodule
```

实现结果



行为描述中的归并操作

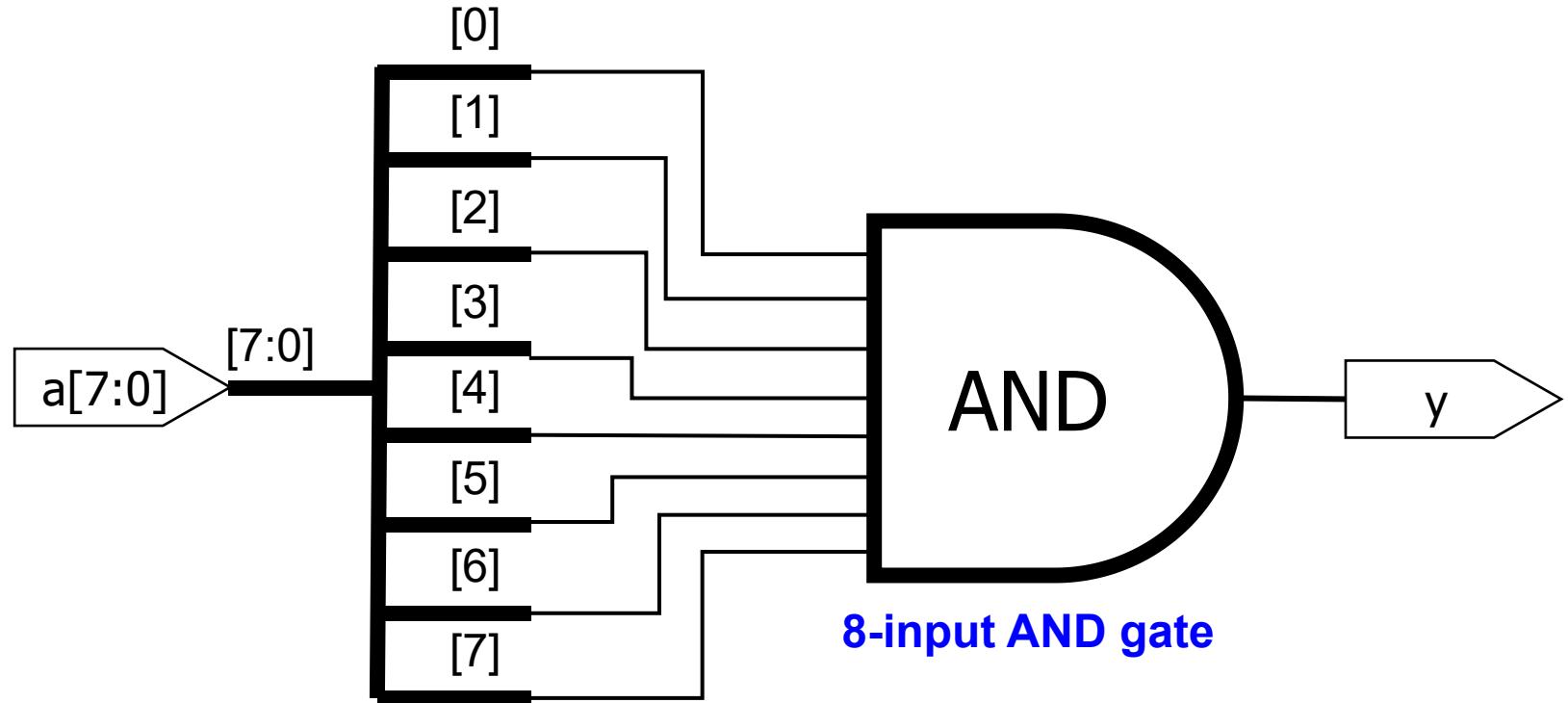
```
module and8(input [7:0] a,
            output          y);

    assign y = &a;

    // &a is much easier to write than
    // assign y = a[7] & a[6] & a[5] & a[4] &
    //           a[3] & a[2] & a[1] & a[0];

endmodule
```

实现结果



行为描述中的条件赋值

```
module mux2(input [3:0] d0, d1,
             input          s,
             output [3:0] y);

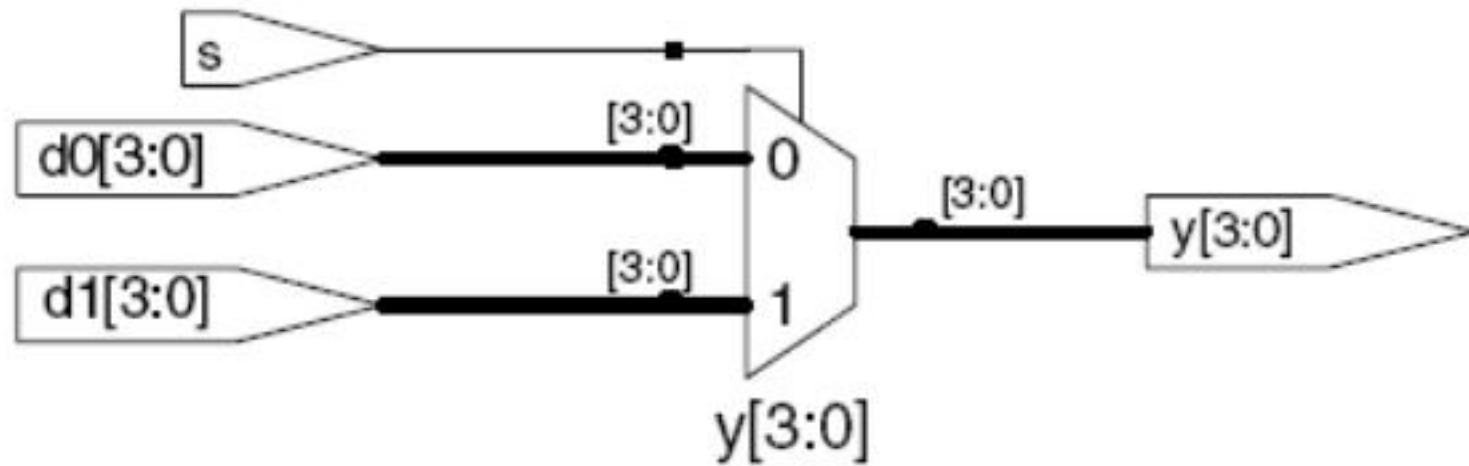
    assign y = s ? d1 : d0;
    // if (s) then y=d1 else y=d0;

endmodule
```

□ ?: 是三元操作符:

- s
- d1
- d0

实现结果



条件赋值2

```
module mux4(input [3:0] d0, d1, d2, d3
            input [1:0] s,
            output [3:0] y);

    assign y = s[1] ? ( s[0] ? d3 : d2)
                  : ( s[0] ? d1 : d0);
    // if (s1) then
    //     if (s0) then y=d3 else y=d2
    // else
    //     if (s0) then y=d1 else y=d0

endmodule
```

条件赋值3

```
module mux4(input [3:0] d0, d1, d2, d3
            input [1:0] s,
            output [3:0] y);

    assign y = (s == 2'b11) ? d3 :
               (s == 2'b10) ? d2 :
               (s == 2'b01) ? d1 :
               d0;
    // if      (s = "11" ) then y= d3
    // else if (s = "10" ) then y= d2
    // else if (s = "01" ) then y= d1
    // else                      y= d0

endmodule
```

操作符优先级

Highest

\sim	NOT
$*, /, \%$	mult, div, mod
$+, -$	add, sub
$<<, >>$	shift
$<<<, >>>$	arithmetic shift
$<, \leq, >, \geq$	comparison
$==, !=$	equal, not equal
$\&, \sim\&$	AND, NAND
$\wedge, \sim\wedge$	XOR, XNOR
$\mid, \sim\mid$	OR, NOR
$:$	ternary operator

Lowest

Verilog中的数字表达

N' Bxx
8' b0000_0001

□ (N) 位数

- 表示将使用多少比特来存储数值

□ (B) Base

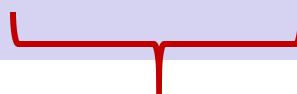
- 基数 b (binary), h (hexadecimal), d (decimal), o (octal)

□ (xx) 数值

- 以基数表示的值
- 也可以有X（无效的，不关心的）和Z（高阻态），作为数值
- 可以使用下划线 _ 来提高可读性

数值举例

Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1010 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001	12'h0	0000 0000 0000

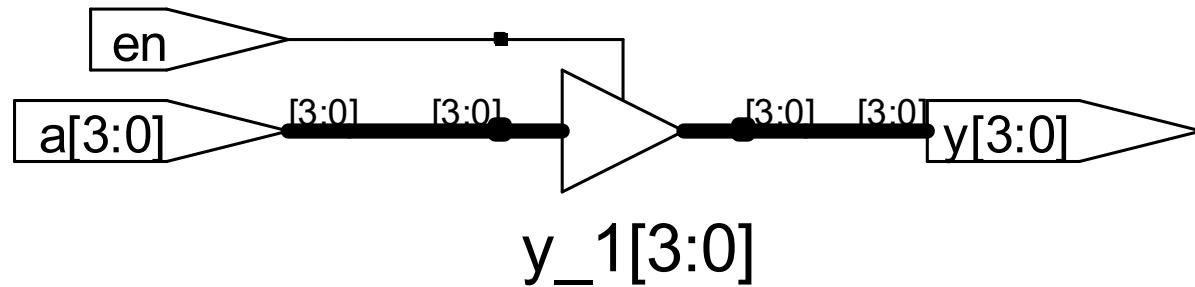

**32 bits
(default)**

高阻态的表达

```
module tristate_buffer(input [3:0] a,
                      input en,
                      output [3:0] y);

  assign y = en ? a : 4'bz;

endmodule
```



高阻态下的真值表

		0	1	Z	X
		0	0	0	0
B		1	0	1	X
	0	0	X	X	X
	1	0	1	X	X
	Z	0	X	X	X
	X	0	X	X	X

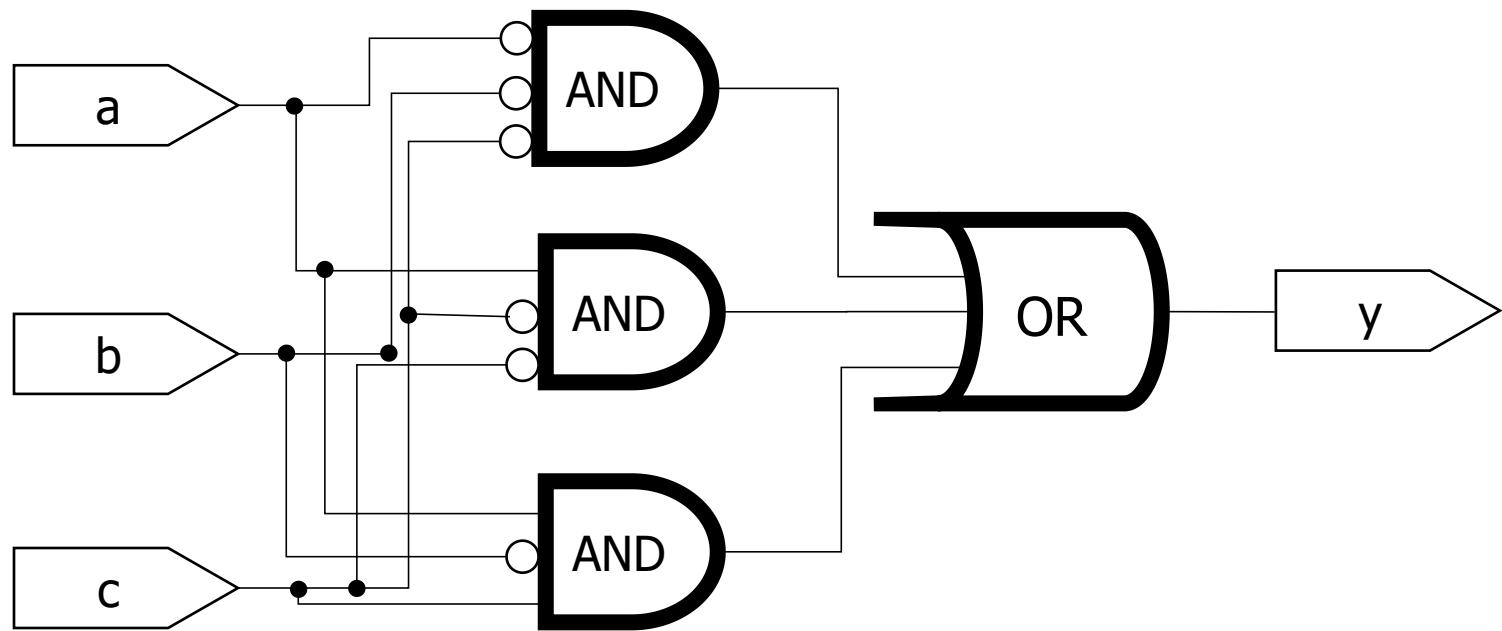
数字逻辑电路中的波形

```
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

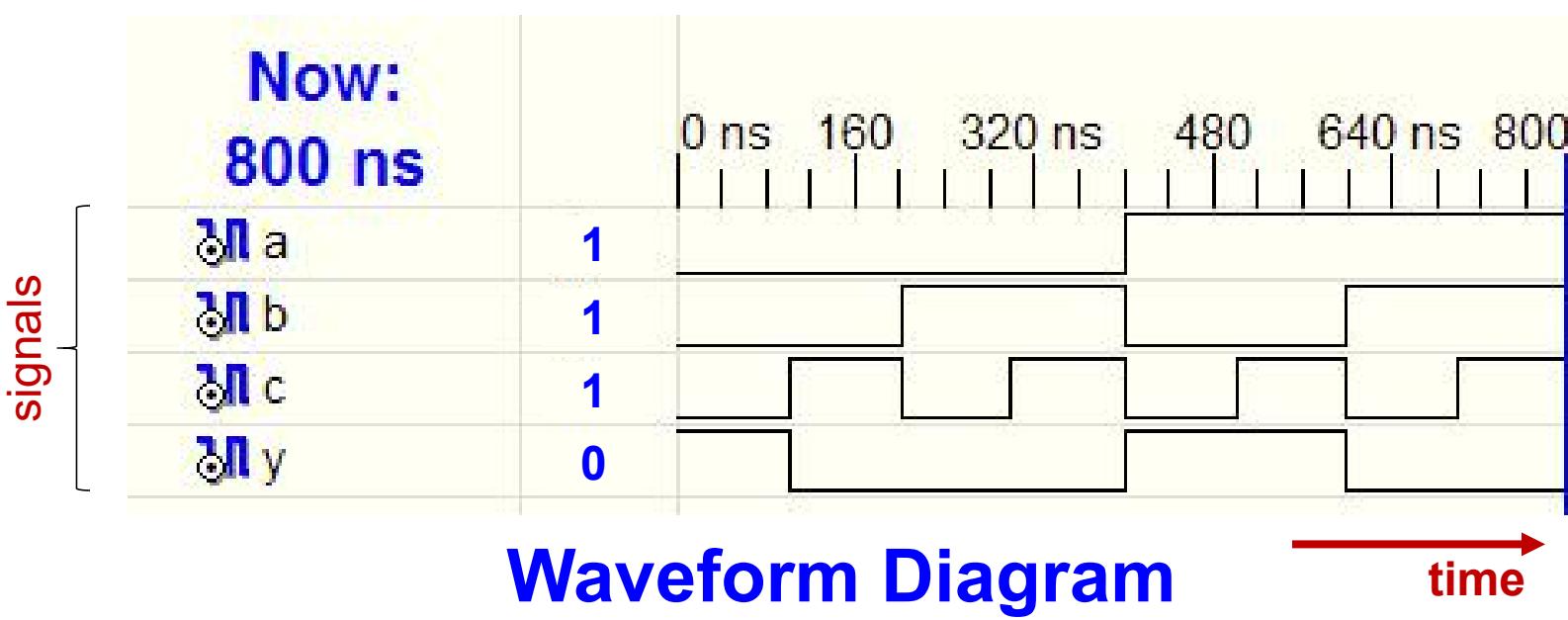
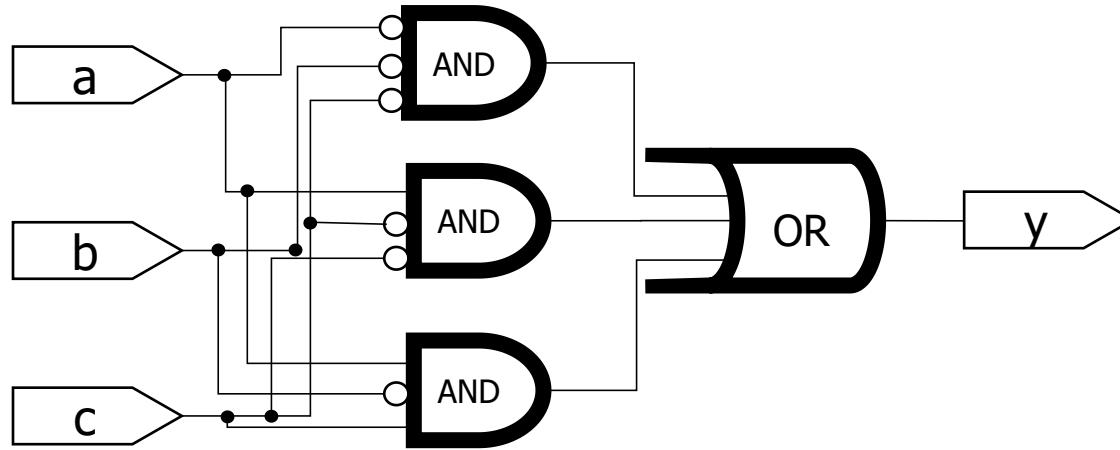
    // here comes the circuit description
    assign y = ~a & ~b & ~c |
                a & ~b & ~c |
                a & ~b &   c;

endmodule
```

综合的结果



仿真波形



硬件编程注意事项

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

- 初学者通常的错误是认为HDL是计算机编程语言，而不是数字硬件的描述语言
- 如果不知道被综合成什么样的硬件，那很有可能就是不对的
- 正确的做法：从组合逻辑，寄存器，有限状态机的角度来考虑系统。可以在写代码之前在纸上绘制模块以及它们之间的连线情况

比较器的多种不同代码实现

□ 可能使用到的模块定义

An XNOR gate

```
module MyXnor (input A, B,  
                output Z);  
  
    assign Z = ~(A ^ B); //not  
    XOR  
  
endmodule
```

An AND gate

```
module MyAnd (input A, B,  
                output Z);  
  
    assign Z = A & B;      // AND  
  
endmodule
```

结构描述实现

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
    MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
    MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
```

逻辑操作符实现

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    assign c01 = c0 & c1;
    assign c23 = c2 & c3;
    assign eq = c01 & c23;

endmodule
```

不使用中间连线信号名字

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    // assign c01 = c0 & c1;
    // assign c23 = c2 & c3;
    // assign eq = c01 & c23;
    assign eq = c0 & c1 & c2 & c3;

endmodule
```

使用信号数组（总线）

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
    wire [3:0] c; // bus definition

    MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0])); // XNOR
    MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1])); // XNOR
    MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2])); // XNOR
    MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3])); // XNOR

    assign eq = &c; // short format

endmodule
```

按位操作运算符

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
    wire [3:0] c; // bus definition

    MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0])); // XNOR
    MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1])); // XNOR
    MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2])); // XNOR
    MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3])); // XNOR

    assign eq = &c; // short format

endmodule
```

最高层的抽象

```
module compare (input [3:0] a, input [3:0] b,  
                output eq);  
  
    // assign c = ~(a ^ b); // XNOR  
  
    // assign eq = &c; // short format  
  
    assign eq = (a == b) ? 1 : 0; // really short  
  
endmodule
```

模块参数

□ 如果需要N位的比较器呢？

```
module mux2
#(parameter width = 8) // name and default value
(input [width-1:0] d0, d1,
 input s,
 output [width-1:0] y);

assign y = s ? d1 : d0;
endmodule
```

参数化模块的例化

```
module mux2
#(parameter width = 8) // name and default value
(input [width-1:0] d0, d1,
 input s,
 output [width-1:0] y);

assign y = s ? d1 : d0;
endmodule
```

```
// If the parameter is not given, the default (8) is assumed
mux2 i_mux (d0, d1, s, out);

// The same module with 12-bit bus width:
mux2 #(12) i_mux_b (d0, d1, s, out);

// A more verbose version:
mux2 #(.width(12)) i_mux_b (.d0(d0), .d1(d1),
                           .s(s), .out(out));
```

延时的指定

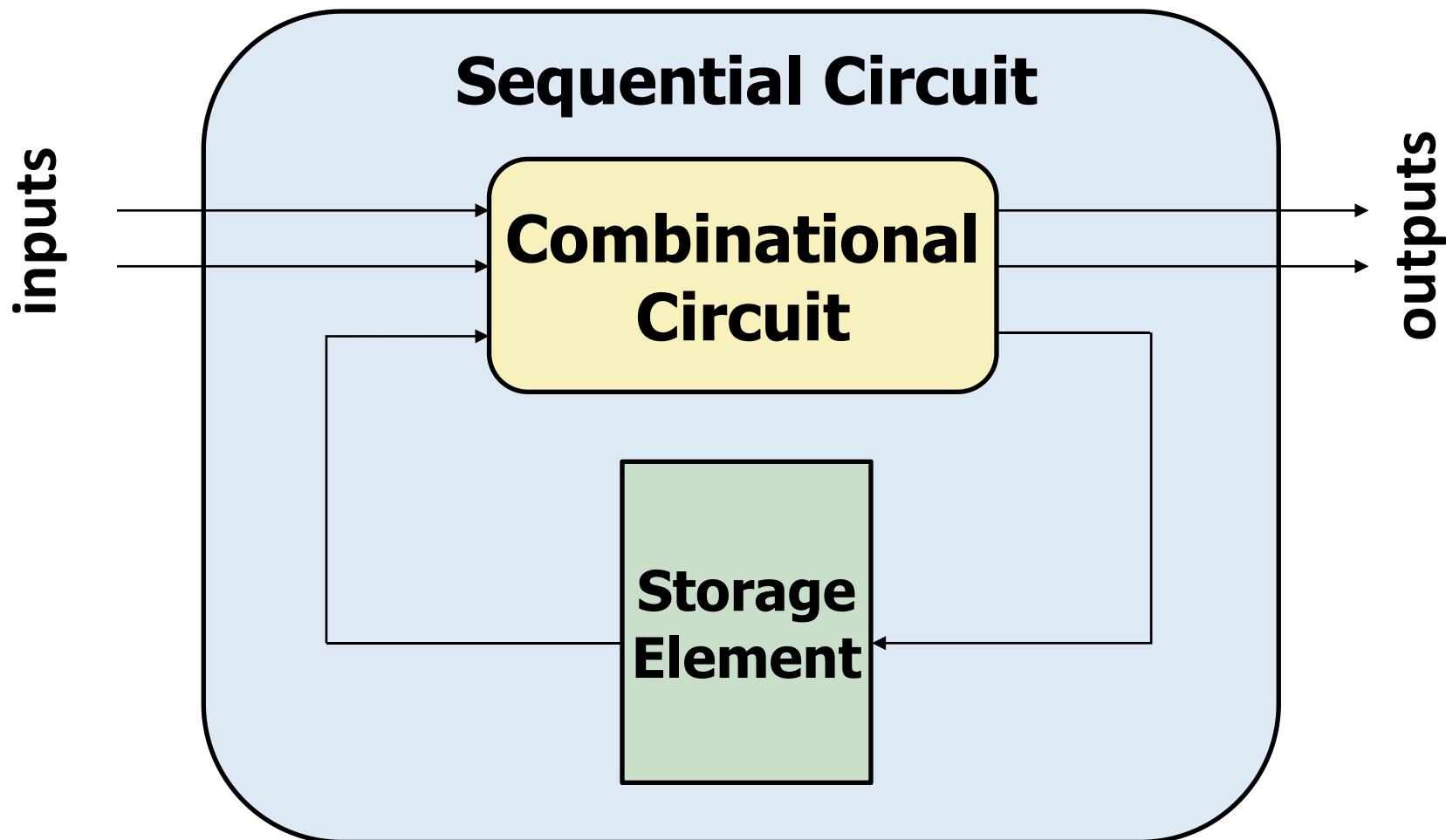
- 在程序中可以指定延时。
- 延时是不可综合的，但是在仿真中非常有用
- 用来仿真硬件的模型
- 1ns代码中时间的单位，1ps仿真的粒度

```
'timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a; // output after 9ns

endmodule
```

时序逻辑电路描述



Verilog时序逻辑编程

□ 定义具有记忆单元的模块

- 锁存器latch, 触发器flipflop, 状态机FSM

□ 时序逻辑通过时钟来“触发”

- 锁存器电平触发
- 触发器时钟边沿触发, 我们仅使用边沿触发

□ 时序逻辑编程需要用到

- always过程语句
- posedge/negedge触发条件

always过程块

```
always @ (sensitivity list)
    statement;
```

每当敏感度列表中的事件发生时。
语句就会被执行

触发器（基于触发器的寄存器） 1

```
module flop(input          clk,
             input      [3:0] d,
             output reg [3:0] q);

    always @ (posedge clk)
        q <= d;           // pronounced “q gets d”

endmodule
```

- **posedge** 在时钟上升沿响应（处在敏感列表中）
- **always**过程块内部的语句会在时钟上升沿的时候得到执行
- 时钟上升沿来临时，d被拷贝入q

触发器（基于触发器的寄存器）2

```
module flop(input          clk,
             input [3:0] d,
             output reg [3:0] q);

    always @ (posedge clk)
        q <= d;                                // pronounced “q gets d”

endmodule
```

- **assign** 语句不会在**always**过程块内部使用
- **<=** 非阻塞赋值

触发器（基于触发器的寄存器）3

```
module flop(input          clk,  
            input [3:0] d,  
            output reg [3:0] q);  
  
    always @ (posedge clk)  
        q <= d;           // pronounced “q gets d”  
  
endmodule
```

- 赋值变量需要被声明为reg
- reg这个名字不一定意味着这个值是一个寄存器（可以是，但不一定是）。

赋值语句

- 持续赋值语句 (continuous assignments)
- assign为持续赋值语句，主要用于对wire型变量的赋值。
- 比如：assign c=a&b;
- 在上面的赋值中，a,b,c三个变量皆为wire型变量，a和b信号的任何变化，都将随时反映到c上来

过程赋值语句

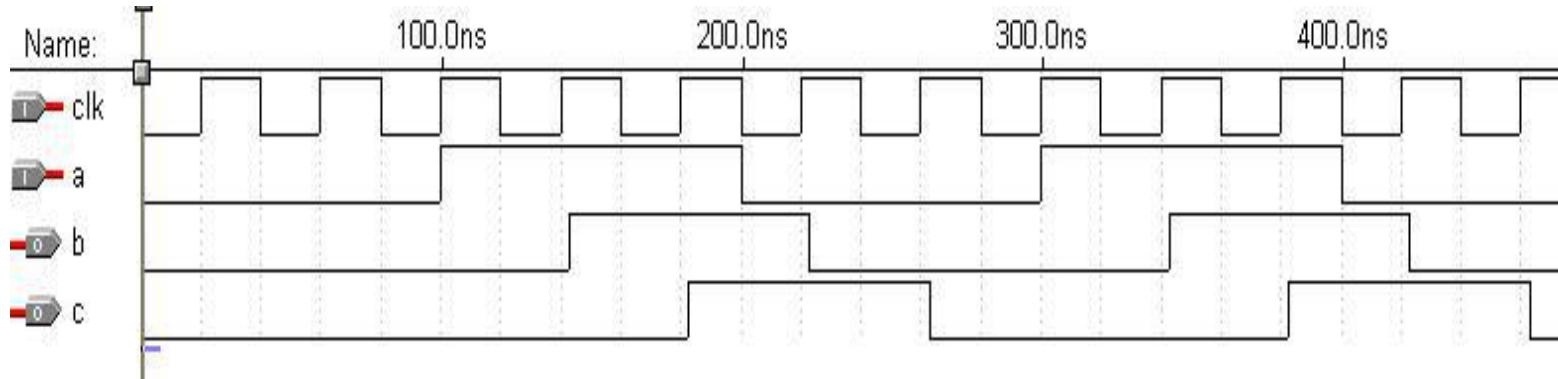
- 过程赋值语句多用于对reg型变量进行赋值。
- (1) 非阻塞 (non_blocking) 赋值方式
- 赋值符号为”<=“，比如： b<=a; 非阻塞赋值在整个过程块结束时才完成赋值操作，即b的值并不是立刻就改变的。
- (2) 阻塞 (blocking) 赋值方式
- 赋值符号为”=”，如： b=a; 阻塞赋值语句在该语句结束时就立即完成赋值操作，即b的值在该条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句不能被执行，仿佛被阻塞了 (blocking) 一样，因此被称为是阻塞赋值方式。

阻塞赋值与非阻塞赋值

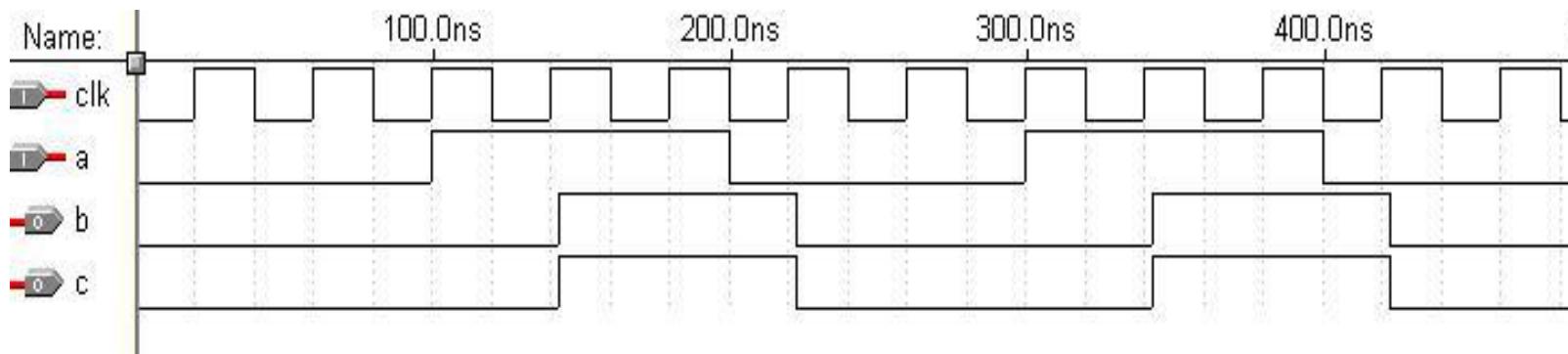
```
module non_block(c,b,a,clk);
output c,b;
input clk,a;
reg c,b;
always @(posedge clk)
begin
  b<=a;
  c<=b;
end
endmodule
```

```
module block(c,b,a,clk);
output c,b;
input clk,a;
reg c,b;
always @(posedge clk)
begin
  b=a;
  c=b;
end
endmodule
```

阻塞赋值与非阻塞赋值的仿真波形

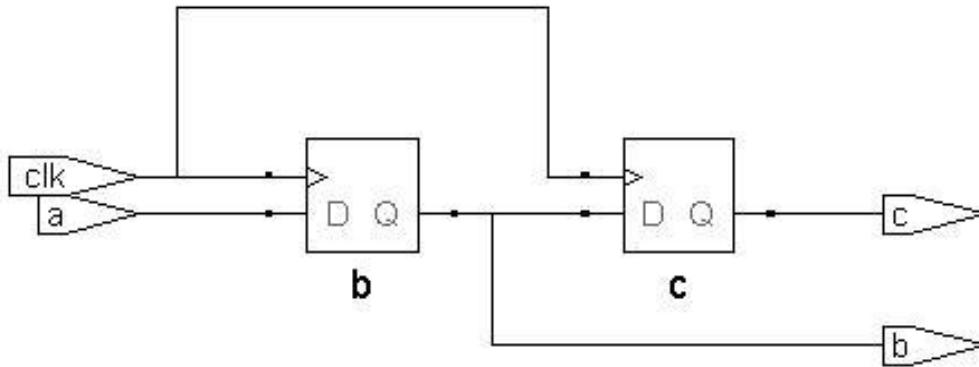


非阻塞赋值仿真波形图

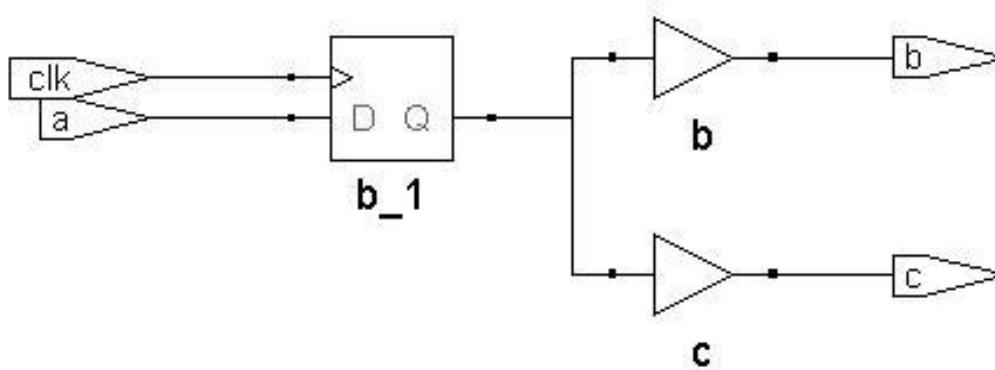


阻塞赋值仿真波形图

阻塞赋值和非阻塞赋值的综合结果



非阻塞赋值综合结果



阻塞赋值综合结果

赋值语句的遵从规则

- 在时序逻辑上使用非阻塞赋值
 - 在组合逻辑上使用阻塞赋值
 - 在组合逻辑上也可以使用连续赋值语句assign
-
- 在always过程块中使用阻塞赋值和非阻塞赋值需要仔细考虑不同赋值的特征

同步复位与异步复位

□ 复位信号用于将硬件初始化到一个已知状态

- 通常在系统启动时激活（上电时）

□ 异步复位

- 复位信号的采样独立于时钟
- 复位信号具有最高的优先权
- 对毛刺敏感，可能会有亚稳态的问题

□ 同步复位

- 复位信号是相对于时钟进行采样的
- 复位应该有足够的长的激活时间，以便在时钟边沿采样成功。
- 完全同步电路的结果

异步重置的触发器1

```
module flop_ar (input          clk,
                  input          reset,
                  input [3:0] d,
                  output reg [3:0] q);

    always @ (posedge clk, negedge reset)
        begin
            if (reset == 0) q <= 0; // when reset
            else           q <= d; // when clk
        end
endmodule
```

- 在这个例子中：有两个事件可以触发这个过程语句
 - 在clk上有一个上升沿
 - 复位时的下降沿

异步重置的触发器2

```
module flop_ar (input          clk,
                 input          reset,
                 input [3:0] d,
                 output reg [3:0] q);

    always @ (posedge clk, negedge reset)
        begin
            if (reset == 0) q <= 0; // when reset
            else           q <= d; // when clk
        end
    endmodule
```

- 首先检查复位：如果复位为0，q被设置为0。
 - 这是一个异步复位，因为复位可以独立于时钟发生（在复位信号的负边沿）
- 如果没有复位，那么常规赋值就会生效

同步重置的触发器1

```
module flop_sr (input          clk,
                  input          reset,
                  input [3:0] d,
                  output reg [3:0] q);

    always @ (posedge clk)
        begin
            if (reset == 0) q <= 0; // when reset
            else           q <= d; // when clk
        end
    endmodule
```

- 该过程只对时钟正边沿敏感
 - 复位只在时钟上升时发生，这是一个同步复位

always过程块

```
module example (input          clk,
                input [3:0] d,
                output reg [3:0] q);

    wire [3:0] normal;           // standard wire
    reg [3:0] special;          // assigned in always

    always @ (posedge clk)
        special <= d;           // first FF array

    assign normal = ~special;   // simple assignment

    always @ (posedge clk)
        q <= normal;           // second FF array
endmodule
```

可以有多个always过程块

不允许在不同的always过程中给相同的信号赋值
为什么?

always过程块的记忆

```
module flop (input          clk,
              input      [3:0] d,
              output reg [3:0] q);

    always @ (posedge clk)
        begin
            q <= d; // when clk rises copy d to q
        end
    endmodule
```

这个过程语句描述的是时钟正边沿时候的动作
如果时钟信号不是正边沿（高电平1，低电平0，或者负边沿）怎么办?
→ 保持不变：这就是过程块记忆实现时序逻辑

不带有记忆效应的always过程块

```
module comb (input           inv,
              input [3:0] data,
              output reg [3:0] result);

    always @ (inv, data)      // trigger with inv, data
        if (inv) result <= ~data; // result is inverted data
        else     result <= data; // result is data

endmodule
```

上述过程块敏感信号是inv和data的电平，穷尽了所有的情况，没有记忆，是组合逻辑过程块

锁存器

```
module latch (input          clk,
              input      [3:0] d,
              output reg [3:0] q);

    always @ (clk, d)
        if (clk) q <= d;           // latch is transparent when
                                    // clock is 1

endmodule
```

这是电平响应，只规定了clk高电平时候的动作
else部分确实，代表这部分q值保持不变，被综合为锁存器

SystemVerilog描述always过程块不同属性

```
module flop (input      clk,
              input      [3:0] d,
              output reg [3:0] q);
    always_ff @ (posedge clk)
        begin
            q <= d;
        end
endmodule
```

```
module latch (input  clk,
               input [3:0] d,
               output reg[3:0] q);
    always_latch @ (clk, d)
        if (clk) q <= d;
endmodule
```

```
module comb (input      inv,
              input      [3:0] data,
              output reg [3:0] result);
    always_comb @ (inv, data) // 没有敏感信号列表
        if (inv) result <= ~data;
        else      result <= data;
endmodule
```

在SystemVerilog中指定过程块的属性可以让系统帮助我们检查，预防出错
在我们的实验中不要出现latch，也不要同时响应时钟的上边沿和下边沿
(FPGA中无此器件)

使用always会不小心生成锁存器

```
wire enable, data;  
reg out_a, out_b;  
  
always @ (*) begin  
    out_a = 1'b0;  
    if(enable) begin  
        out_a = data;  
        out_b = data;  
    end  
end  
No assignment for ~enable
```

```
wire enable, data;  
reg out_a, out_b;  
  
always @ (data) begin  
    out_a = 1'b0;  
    out_b = 1'b0;  
    if(enable) begin  
        out_a = data;  
        out_b = data;  
    end  
end  
Not in the sensitivity list
```

Sequential

Sequential

使用SystemVerilog就会收到检查错误的消息
确定自己实现组合逻辑, 请使用always_comb

case语句

```
module sevensegment (input      [3:0] data,
                      output reg [6:0] segments);

    always @ ( * )          /* * is short for all signals
    always_comb
        case (data)           // case statement
            4'd0: segments = 7'b111_1110; // when data is 0
            4'd1: segments = 7'b011_0000; // when data is 1
            4'd2: segments = 7'b110_1101;
            4'd3: segments = 7'b111_1001;
            4'd4: segments = 7'b011_0011;
            4'd5: segments = 7'b101_1011;
            // etc etc
            default: segments = 7'b000_0000; // required
        endcase

    endmodule
```

casex, casez使用通配符

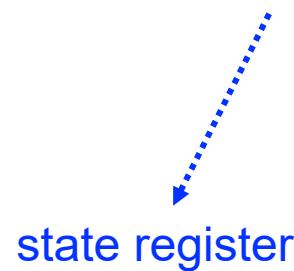
```
always_comb
begin
    {int2, int1, int0} = 3'b000;
    casez (irq)
        3'b1?? : int2 = 1'b1;
        3'b?1? : int1 = 1'b1;
        3'b??1 : int0 = 1'b1;
        default: {int2, int1, int0} = 3'b000;
    endcase
end
```

代码中禁止使用casex, 通配的话用casez就好

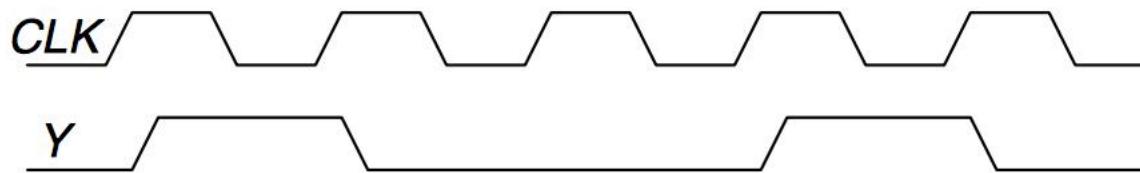
FSM编程

□ 每个有限状态机包含三个部分

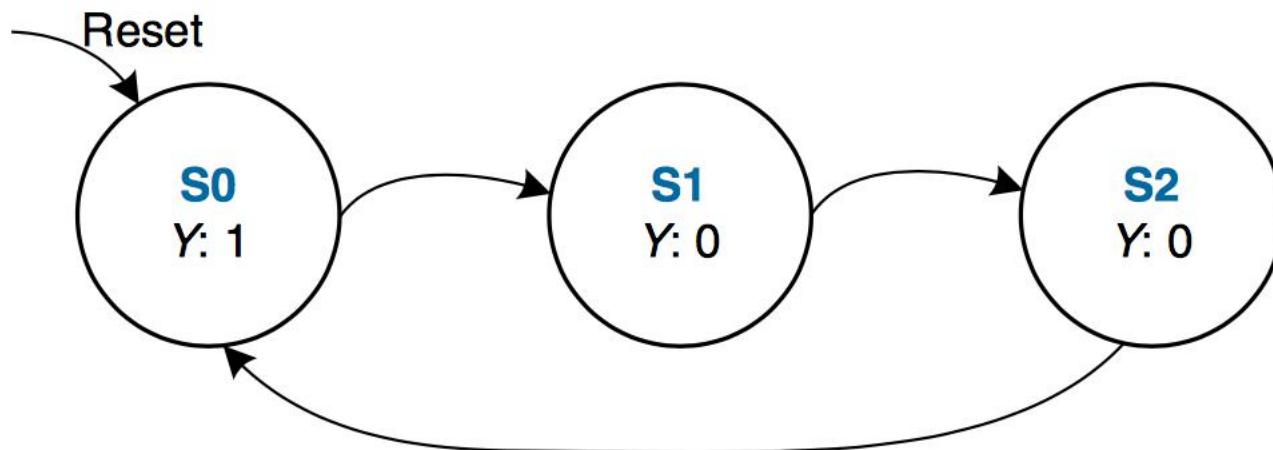
- 现态逻辑（时序逻辑）
- 次态逻辑（组合逻辑）
- 输出逻辑（组合逻辑）



FSM举例：三分频率1



在每3个时钟周期中，输出Y为高电平。换句话说，该输出将时钟的频率除以3。



FSM举例：三分频率2

```
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

    always @ (posedge clk, posedge reset) // state register
        if (reset) state <= S0;
        else         state <= nextstate;

    always @ (*)                                // next state logic
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase

        assign q = (state == S0);                  // output logic
endmodule
```

SystemVerilog

- SystemVerilog 中，一切都可以是 logic
- 不必显式区分 wire 和 reg
 - wire确实是连线，而reg不一定被综合成寄存器
- 在不同的环境下自动被推断为寄存器或组合逻辑
- 可以(几乎在)所有场合代替原有的 wire 和 reg

例子

```
logic [3:0] counter, counter_next;
logic counter_wrap;

// combinational logic
assign counter_wrap = counter == 4'b1111;

// also combinational logic
always_comb begin
    counter_next = counter + 1;
end

// sequential logic
always_ff @(posedge clk) begin
    if (reset) begin
        counter <= 'b0;
    end
    else begin
        counter <= counter + 1;
    end
end
```

例子（错误）

```
logic not_latch, must_be_seq;
```

```
always_comb begin
    if (some_cond) begin
        not_latch = 1'b1;
    end
    // latch (no else) in always_comb -- synthesis error
end
```

```
always_ff @(clk) begin
    must_be_seq <= 1'b1;
    // no trigger edge in always_ff -- synthesis error
end
```

unique 与 priority

- Verilog 的 case 语句语义比较复杂，需要注解提示综合器(参见:full case, parallel case)，并且容易产生 latch。if 也可能会忘记书写分支，导致 latch，或者产生意料之外的多个命中。
- SystemVerilog 中，新增了三个关键词 unique, unique0, priority，可以搭配 case 和 if 使用：
 - unique:只可能命中一项。仿真时如果命中多个或者没有命中将产生警告。
 - unique0:只可能命中至多一项(危险!)。仿真时如果命中多个将产生警告。
 - priority:可能命中多项，此时语句书写顺序作为匹配优先级。仿真时如果没有命中将产生警告。
- 注意：上述几个关键词并不会避免产生 latch，推荐总是将组合逻辑放置在 always_comb 中让综合器进行检查。

unique 与 priority(例子)

□ 注意:如果需要通配符 ?, 请使用 casez, 不要使用 casex。此外, 可综合的代码的 case 中不应当出现 X 和 Z。否则, 可能出现非预期的结果。

```
// 4-bit priority decoder
// p: 4 bits input, d: 2 bits output
always_comb begin
    priority casez (p) begin
        4'b1???: {valid, d} = 3'b111;
        4'b01???: {valid, d} = 3'b110;
        4'b001?: {valid, d} = 3'b101;
        4'b0001: {valid, d} = 3'b100;
        default: {valid, d} = 3'b000;
    end
end
```



谢谢



数据表示及检错纠错

2022年秋

内容概要

- 数据表示的需求
- 逻辑型数据表示
- 字符的表示
- 整数与浮点数
- 检错纠错码

数据的编码与表示

□ 需要在计算机中表示的对象

- 程序、整数、浮点数、字符（串）、逻辑值
- 通过编码表示

□ 表示方式

- 用数字电路的两个状态表示，存放在机器字中
- 由上一层的抽象计算机来识别不同的内容

□ 编码原则

- 少量简单的基本符号
- 一定的规则
- 表示大量复杂的信息
- 计算性能/存储空间

编码表示

- 基本元素
 - 0、1两个基本符号
- 字符
 - 26字符→5位
 - 大/小写+其它符号→7bits
 - 世界上其它语言的文字→16bits (Unicode)
- 无符号整数 (0, 1, ..., 2^{n-1})
- 有符号整数
- 浮点数
- 逻辑值
 - 0→false, 1→true
- 颜色 (RGB)
- 位置/地址/指令
 - 但是: n位只能代表 2^n 个不同的对象

理解对象表示以及在
对象上的允许的操作

逻辑型数据

□ 逻辑型数据

- True, 真
- False, 假

□ 数据表示

- 1
- 0

□ 数据运算

- 与运算
- 或运算
- 非运算
- 异或运算

X	Y	X与Y	X或Y	X的非
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

字符型数据

□ 重要的人机界面

- 由符号组成
- 为每个符号进行编码、由输入/输出设备进行转换
- 一般以字符串的形式在计算机存储器中存放

□ 字符集编码标准

- 主机和设备、主机之间进行信息交换的基础
 - ASCII
 - UNICODE
 - UTF-8

ASCII字符编码

- American Standard Code for Information Interchange
- 采用7位二进制编码， 占用一个字节
- 表示128个西文字符

ASCII码字符集

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	Ø	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	Ø	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	Ø	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.asciitable.com

UNICODE编码

- 它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990年开始研发，1994年正式公布。
- 使用16位表示一个字符，可以表示65536个字符
- 将整个编码空间划分为块，每块为16的整倍数，按块进行分配
- 保留6400个码点供本地化使用
- 依然无法覆盖所有字符

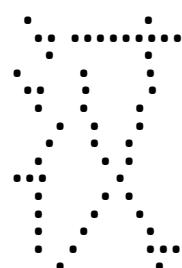
UTF-8编码

字符位数	字节1	字节2	字节3	字节4	字节5	字节6
7	0ddddddd					
11	110dddddd	10dddddd				
16	1110dddd	10dddddd	10dddddd			
21	11110ddd	10dddddd	10dddddd	10dddddd		
26	111110dd	10dddddd	10dddddd	10dddddd	10dddddd	
31	1111110d	10dddddd	10dddddd	10dddddd	10dddddd	10dddddd

- 变长字符编码，提高存储空间利用率
- 字符长度由首字节确定
- 字符首字节外，均以“10”开始，可自同步
- 可扩展性强
- 成为互联网上占统治地位的字符集

点阵字体

- 点阵本质是单色位图
- 如果为0，对应位置没有点，为1显示点
- 文字编码→查找字体文件→找到点阵→显示



	1													1			
		1	1		1	1	1	1	1	1	1	1	1	1	1		
			1											1			
1						1								1			
	1	1					1							1			
			1					1						1			
				1					1					1			
					1					1				1			
						1					1			1			
							1					1		1			
								1					1	1			
									1					1			
										1					1		
											1				1		
												1					
													1	1	1		
														1			

矢量字体 (TrueType)

Bitmap TrueType



- 一个字可以用多条曲线来表示，每条曲线保存其关键点
- 显示字的时候，取出这些关键点，采用平滑的曲线将这些关键点连接起来，并填充闭合空间以显示
- 需要放大或者缩小的时候，按照比例改变关键点的相对位置即可

数值型数据表示

□ 定点数

- 小数点位置固定
- 整数
- 定点小数

□ 浮点数

- 小数点位置浮动

数值范围和数据精度

□ 数值范围

- 数值范围是指一种类型的数据所能表示的最大值和最小值

□ 数据精度

- 通常指实数所能给出的有效数字位数；对浮点数来说，精度不够会造成误差，误差大量积累会出问题

□ 机内处理

- 数值范围和数据精度概念不同。在计算机中，它们的值与用多少个二进制位表示某种类型的数据，以及怎么对这些位进行编码有关

整数

- 原码，反码，补码
- 符号扩展
- 大端，小端
- 加法，减法，乘法，除法

原码，反码，补码表示小结

□ 正数的原码、反码、补码表示均相同，

- 符号位为0，数值位同数的真值。

□ 零的原码和反码均有2个编码，补码只1个码

□ 负数的原码、反码、补码表示均不同

- 符号位为1，数值位：原码为数的绝对值

- 反码为每一位均取反码

- 补码为反码再在最低位+1

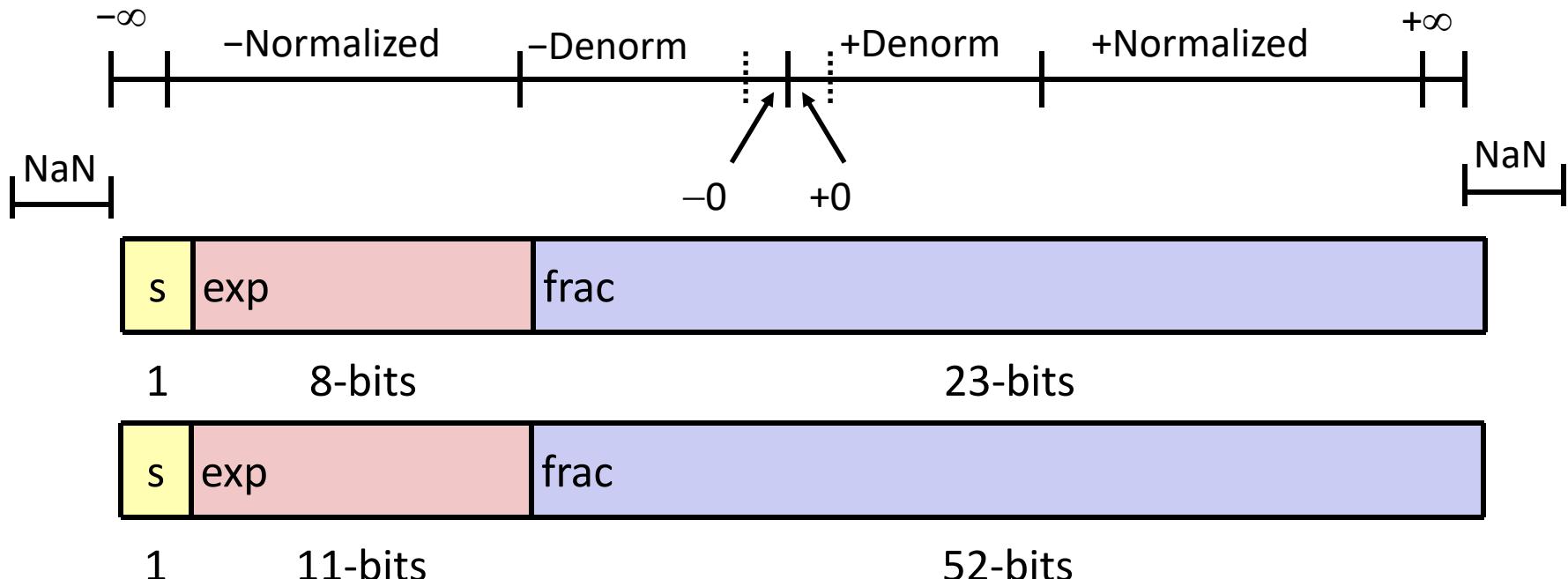
- 只有一个负数的原码与补码是相同的：1100 0000 0000
0000 0000 0000 0000 0000（想想为什么）

□ 由 $[X]_{\text{补}}$ 求 $[-X]_{\text{补}}$ ：每一位取反后再在最低位+1

浮点数

$$(-1)^s M \cdot 2^E$$

$$\text{Bias} = 2^{k-1} - 1$$



□ 单精度, 双精度

□ 规格化数($\text{exp} \neq 0, \text{exp} \neq 111\dots 1$)，非规格化数 ($\text{exp}=0$) , 0, 无穷 ($\text{exp}=111\dots 1, \text{frac}=000\dots 0$), NaN($\text{exp}=111\dots 1, \text{frac} \neq 000\dots 0$)

$v = (-1)^s M \cdot 2^E$
$E = \text{exp} - \text{Bias}$
$M = 1.\text{frac}$

$v = (-1)^s M \cdot 2^E$
$E = 1 - \text{Bias}$
$M = 0.\text{frac}$

单精度:
 $k=8, \text{Bias}=127$

双精度:
 $k=11, \text{Bias}=1023$

浮点数算数运算

- 浮点数加减法运算
- $X = M_x \times 2^{E_x}$, $Y = M_y \times 2^{E_y}$
- (1) 对阶操作, 求阶差: $E = E_x - E_y$,
 - 使阶码小的数的尾数右移 $|E|$ 位
 - 其阶码取大的阶码值;
- (2) 尾数加减
- (3) 规格化处理
- (4) 舍入操作, 可能带来又一次规格化
- (5) 判结果的正确性, 即检查阶码上下溢出

浮点数加运算举例

□ $X=2^{+010} \times 0.1101100, Y=2^{+100} \times (-0.1010110)$

□ 写出X、Y的正确的浮点数表示：

- 阶码用4位移码 尾数用8位原码
- (含符号位) (含符号位)
- 浮
- 浮

□ 为运算方便，尾数的符号位写在数值位之前：

- 浮
- 浮

浮点数加运算举例

□ $X = 2^{+010} \times 0.1101100, Y = 2^{+100} \times (-0.1010110)$

□ (1) 计算阶差 (移码计算) :

■ $E = E_X - E_Y = E_X + (-E_Y) = 1010 + 0100 = 0110$

■ 注意: 阶码计算结果的符号位在此变了一次反, 为-2 的移码, 是X的阶码值小, 使其取Y 的阶码值1100 (即+4); 因此, 相应地修改 $[M_X]_{原} = 0001101100$ (即右移2 位)
(右移出的00被保存到保护位中)

□ (2) 尾数求和: 此处是原码加法, 符号不相同, 绝对值大的减小的, 结果符号取决于绝对值大的数

$$\begin{array}{r} 11010110 \\ - 00011011\text{ 00} \\ \hline 10111011\text{ 00} \end{array}$$

浮点数加运算举例

□ (3) 规格化处理:

- 相加结果,数值的最高位为0,应执行1次左移操作,
- 故得 $[M_{X+Y}]_{原} = 1\ 1110110$, 阶码减1得1 011 (为+3)

□ (4) 舍入处理: 舍入位是0, 按0舍1入规则, 得到最终结果: 1 1110110

□ (5) 检查溢出否: 和的阶码为1011, 不溢出

- 计算后的 $[X+Y]_{浮} = 1\ 1011\ 1110110$
- 即数的实际值为: $2^3 \times (-0.1110110)$

浮点数乘除法

口 算法：

- 阶码加减： 乘法： E_x+E_y , 除法 E_x-E_y
- 对尾数进行乘除法，求得结果
- 规格化
- 舍入，可能再次进行规格化
- 进行溢出检查（阶码）

浮点数运算

□ 浮点数的加减法

- 移码的减法运算
- 无符号数运算

□ 浮点数的乘除法

- 移码的加减运算（注意溢出）

□ 浮点数尾数运算

- 原码运算

浮点运算的特点

□ 浮点数的加法不满足结合律

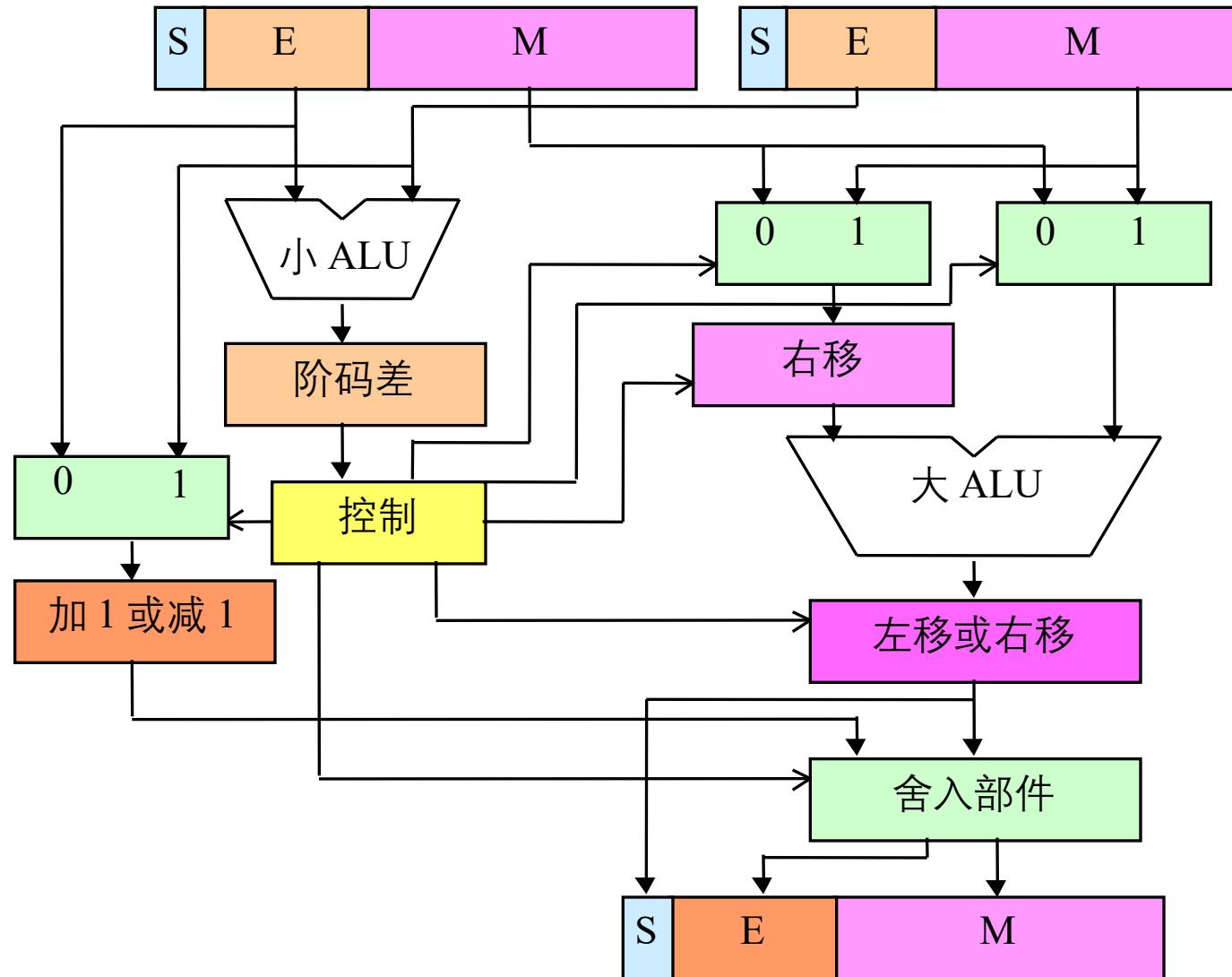
- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
- $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + (1.5 \times 10^{38}) = 0.0$
- $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = (0.0) + 1.0 = 1.0$

□ 浮点数加法不可结合

□ 浮点数的相等比较：小心！只是近似

□ `for(i=0;i!=10;i+=0.1)`

浮点运算部件



浮点运算器举例-Intel 80287



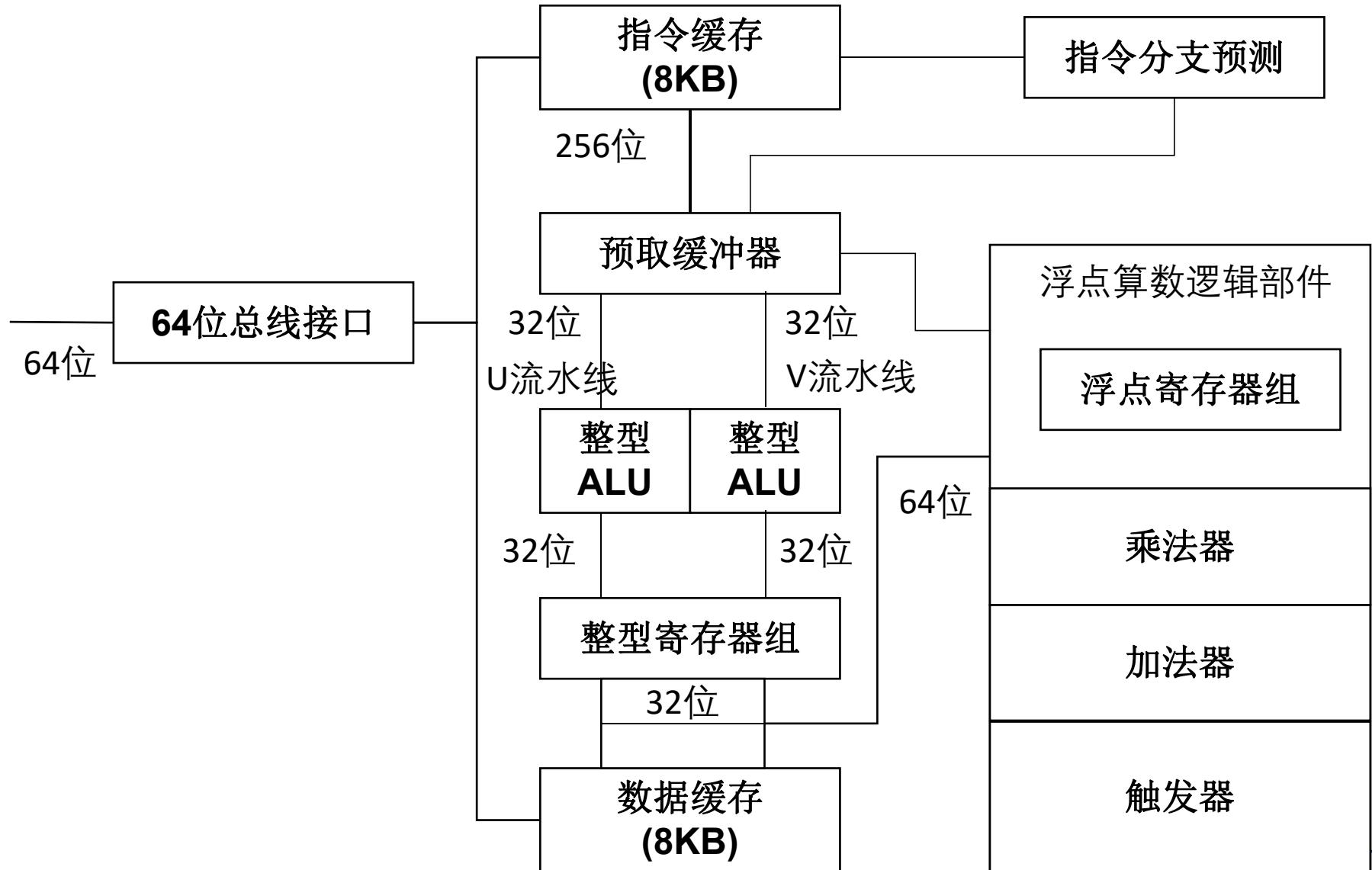
浮点运算部件以协处理器方式和CPU 连接，有独立的控制逻辑；

8个 80位 浮点数寄存器，精度更高，采用堆栈结构并进行了扩展；

支持 3大类共 7 种数据，支持约 60 条指令；

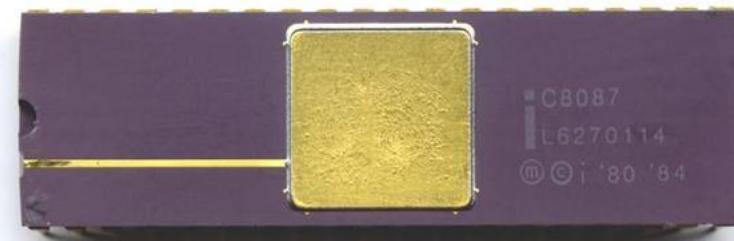
在后来的奔腾机中有重大改进。

Pentium结构简图



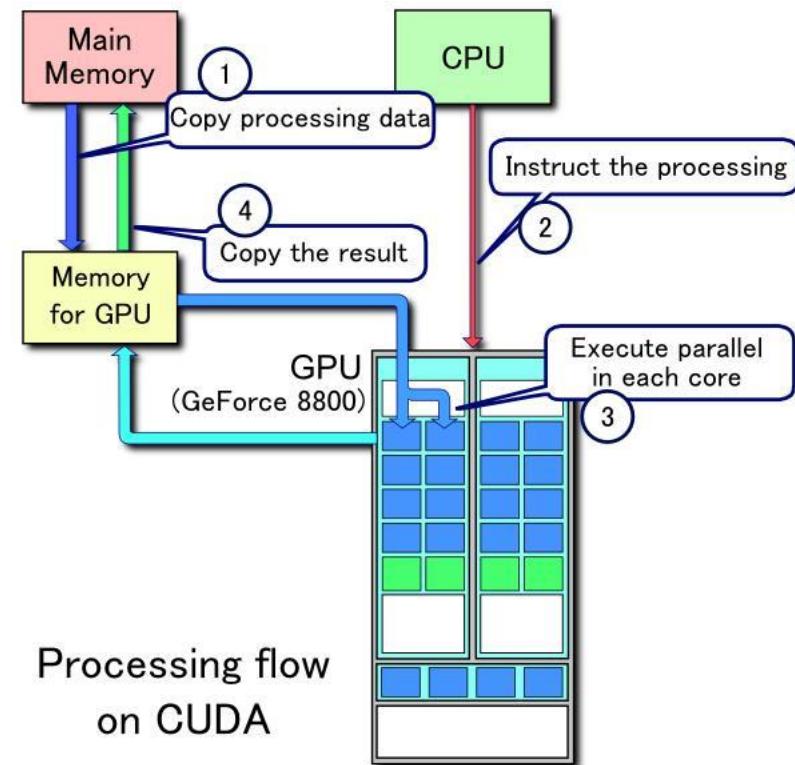
现代处理器对浮点数运算的支持

- Intel设计了独立于8086和8088处理器外的8087数学辅助处理器
- Intel在80486DX处理器核心内首次集成了浮点运算单元
- 在现代处理器中，浮点计算功能会通过SIMD (Single Instruction Multiple Data, 单指令多数据流) 的技术实现并行计算能力
- MMX, SSE1~4, AVX1~2~512, FMA



现代的高速浮点计算单元

- GPGPU, General Purpose Graphic Processing Unit
- CUDA
- OpenCL
- CPU中的浮点运算单元是为了更高精度浮点运算设计的。Intel-AVX 指令集处理512位扩展数据
- GPU中的处理器都是为高度并行计算而设计的，在Nvidia以及AMD 图形处理器上支持的数据精度大多是单精度和双精度浮点计算（FP32和FP64），甚至随着机器学习，深度学习，神经网络的流行，最新的图形处理器甚至支持了半精度浮点运算（FP16）。



检错纠错码

- 数据或编码在存储、传输等过程中可能出错（甚至可能丢失）
- 如何判断已经出错
 - 比较：与所有正确的编码进行比较
 - 特征：检验是否存在某些特征（预先放置到编码中）
- 发现错误之后能否自动纠正？

- 计算机种的数据如何进行检错？

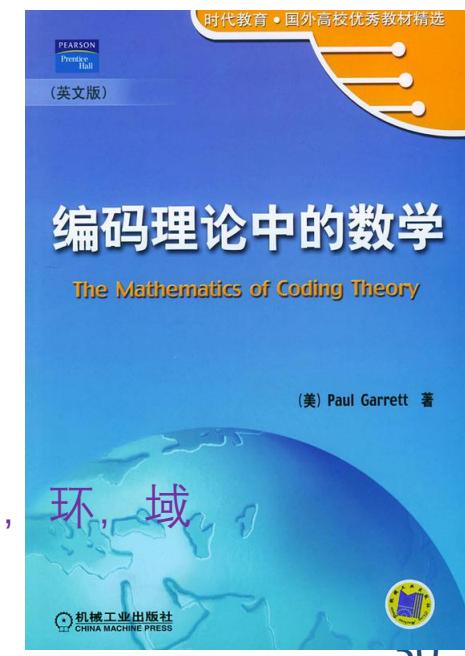
- 纠错？

- 纠删码（Erasure Code）

- 丢失之后恢复（存储）

深入了解
基础：

- 代数结构：群，
- 线性代数



检错纠错码

- 使编码具有某种特征，通过检查这种特征是否存在来判断编码是否正确
- 出错时，如果还能指出是哪位出错，则可以纠正错误
 - 编码
 - 检查
 - 出错后纠正

码距

□ 码距（最小码距）的概念：是指任意两个合法码之间至少有几个二进制位不相同。

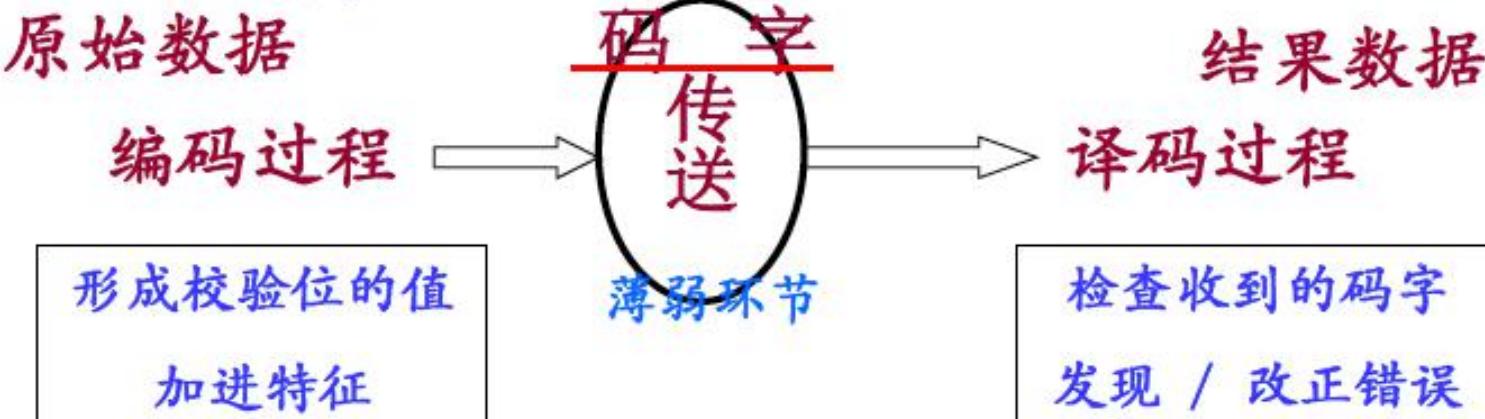
- 仅有有一位不同的编码是无纠错能力的。例如用4位二进制表示16种状态，则16种编码都用到了，此时码距为1。任意一个编码状态的四位码中的一位或者几位出错，都会变成另外一个合法码。这种编码无检错能力。
- 若用4个二进制位表示8种合法的状态，就可以只使用其中的8个编码来表示，另外8个为非法编码。此时可以使合法码的码距为2。任何一位出错后都会成为非法码，有发现一位出错的能力

□ 合理增大码距，能提高发现错误的能力，但表示一定数量的合法码所使用的二进制位数要变多，增加了电子线路的复杂性和数据存储、数据传送的数量。

常用检错纠错码

- 数据编码 → 数据传输 → 数据译码
- 三种常用的检错纠错码：
 - 奇偶校验码：并行数据传输
 - 海明校验码：并行数据传输
 - 循环冗余校验码：串行数据传输

检错纠错过程：

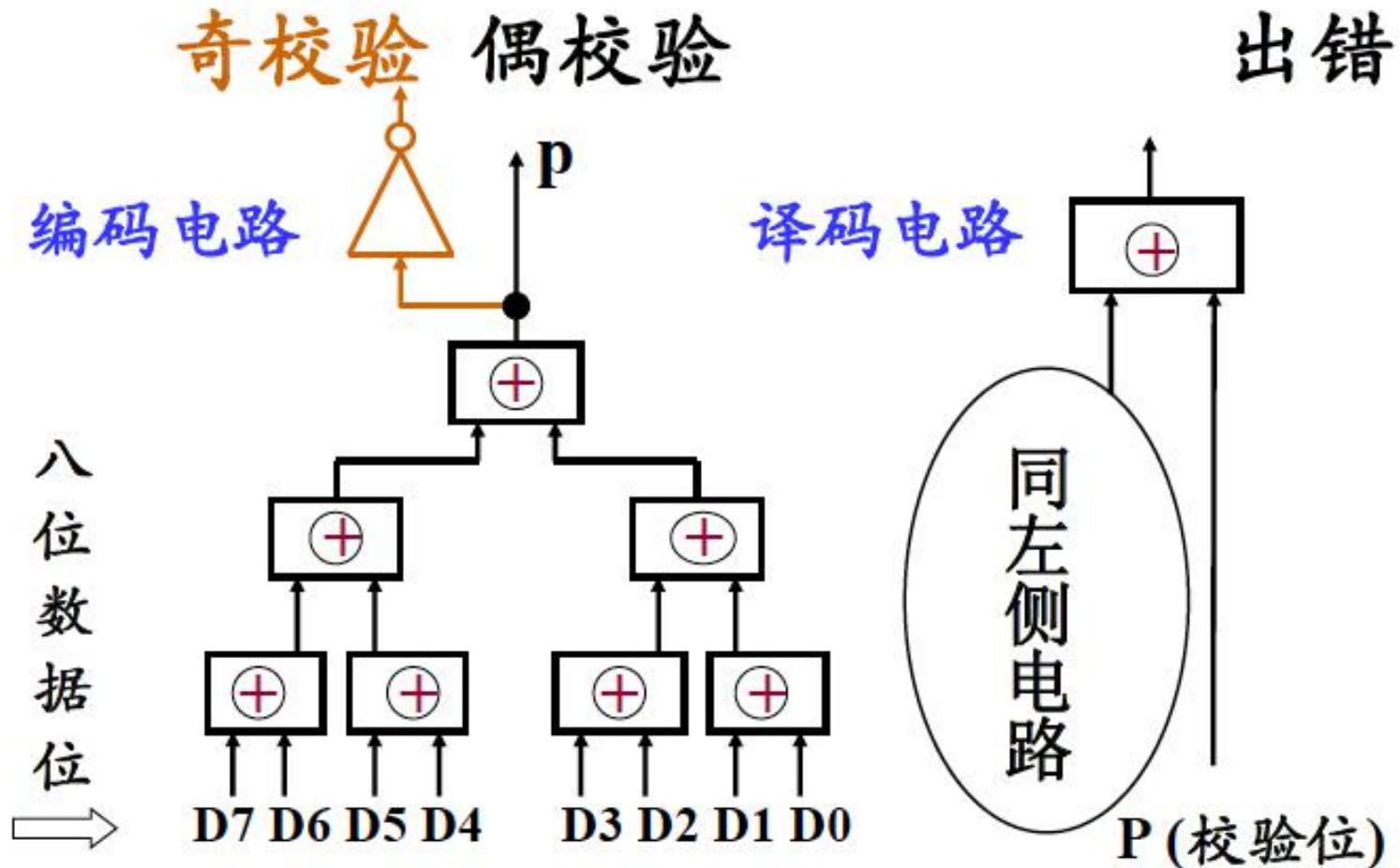


奇偶校验码

- 奇校验，偶校验
- 用于并行码检错
- 原理：在 k 位数据码之外增加 1 位校验位，
使 $k+1$ 位码字中取值为 1 的位数总保持为偶数（偶校验）或奇数（奇校验）。
- 有效数字：0001 例如：
- 有效数字：0101



奇偶校验的实现电路



海明校验码

- 用于多位并行数据检错纠错处理
- 实现：为 k 个数据位设立 r 个校验位，使 $k+r$ 位的码字同时具有这样两个特性
 - 能发现并改正 $k+r$ 位中任何一位出错
 - 能发现 $k+r$ 位中任何二位同时出错，但已无法改正
- k 与 r 之间应该满足什么样的关系？

海明码的编码方法

- 合理的使用 k 位数据形成 r 个校验位的值，保证用 k 个数据中不同的数据位组合来形成每个校验位的值，使任何一个数据位出错时，将影响 r 个校验位中不同的校验位组合起变化。这样一来，就可以通过检查哪种校验位组合起了变化，来推断是那个数据位错误造成的，对该位求反则实现纠错
- 有的时候两位出错与某种情况的一位出错对校验位组合的影响相同，必须加以区分与解决
- 位数 r 和 k 的关系： $2^r \geq k+r+1$ ，即用 2^r 个编码分别表示 k 个数据位， r 个校验位中哪一位出错，都不错
- $2^{r-1} \geq k+r$ ，用 $r-1$ 位校验码为出错位编码，再单独设一位以区分1位还是2位同时出错，更实用

海明码的实现

例如: $k = 3, r = 4$

D3	D2	D1	P4	P3	P2	P1	⊕ 表示异或
1	1	1	1	1	1	1	
1	1	0	0	1	0	0	$P1 = D2 \oplus D1$
1	0	1	0	0	1	0	$P2 = D3 \oplus D1$
0	1	1	0	0	0	1	$P3 = D3 \oplus D2$
$P4 = P3 \oplus P2 \oplus P1 \oplus D3 \oplus D2 \oplus D1$							
编码方案							

译码方案

$S1 = P1 \oplus D2 \oplus D1$
$S2 = P2 \oplus D3 \oplus D1$
$S3 = P3 \oplus D3 \oplus D2$
$S4 = P4 \oplus P3 \oplus P2 \oplus P1 \oplus D3 \oplus D2 \oplus D1$

海明码的实现方案

□ 如何分配不同的数据位组合来形成每个校验位的值

□ P1 P2 D1 P3 D2 D3 P4

□ 1 2 3 4 5 6 编码方案

□ (一) 准备工作:

□ (1) 从1~6按次序排列数据位、校验位,

□ (2) 将校验位P1、P2、P3依次安排在2的幂次方位。

□ (3) P4为总校验位，暂不考虑。

海明码的实现方案

□ 如何分配不同的数据位组合来形成每个校验位的值

□ P1 P2 D1 P3 D2 D3 P4

□ 1 2 3 4 5 6 编码方案

□ (二) 为各校验位分配数据位组合:

□ (1) 看数据位的编号分别为3、5、6，它们是校验位编号的组合：

□ 3=1+2、5= 1+4、6= 2+4

□ (2) 1出现在3和5中，则P1负责对D1和D2进行校验。

□ (3) 2出现在3和6中，则P2负责对D1和D3进行校验。

□ (4) 4出现在5和6中，则P3负责对D2和D3进行校验。

海明码的实现方案

□ 如何分配不同的数据位组合来形成每个校验位的值

□ P1 P2 D1 P3 D2 D3 P4

□ 1 2 3 4 5 6 编码方案

□ (三) 写出各校验位的编码逻辑表达式:

□ (1) 结果是:

□ $P1 = D2 \oplus D1; P2 = D3 \oplus D1; P3 = D3 \oplus D2$

□ (2) 用其他各校验位及各数据位进行异或运算求校验位
P4 的值, 用于区分无错、奇数位错、偶数位错3 种情况

□ 总校验位 $P4 = P3 \oplus P2 \oplus P1 \oplus D3 \oplus D2 \oplus D1$

海明码的译码方案

□ 译码方案是：

- 对接收到数据位再次编码，用得到的结果和传送过来的校验位的值相比较，二者相同表明无错，不同是有1位错了。或者将校验位与对应数据位进行异或，获得**S4~S1**值

□ 排查是哪一位错了，看**S4~S1** 这4 位的编码值

海明码的应用实例

- 如已有数据为**110**, 编码为:P1P2 **1** P3 **10** P4则有:
- P1=0, P2=1 P3=1, P4=0

请看如下 3 种情况:

无错,

单独 1 位错,

2 位同时错

若无错, 则

S4 S3 S2 S1=**0000**

4 位 S 全为 0

若仅 D1 错, 则

S4 S3 S2 S1=**1011**

S3S2S1 不为 000

其中 S4 必为 1

若 P2 D1 错, 则

S4 S3 S2 S1=**0001**

其中 S4 必为 0,

S3S2S1 不为 000

更多的海明码

Bit position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X	
	p2		X	X			X	X			X	X			X	X			X	X	
	p4			X	X	X	X					X	X	X	X						X
	p8							X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X

检错纠错码小结

- (1) k 位码有 2^k 个编码状态，全用于表示合法码，则任何一位出错，均会变成另一个合法码，不具有检错能力
- (2) 从一个合法码变成另外一个合法码，至少改变几位码的值，称为**最小码距**（码距），码距和编码方案将决定其检错纠错能力。
 - 奇偶校验码的码距为2
 - 海明码的码距为4

检错纠错能力

- (3) $k+1$ 位码，只用其 2^k 个状态，可以使码距为2，如果一个合法码中的一位错了，就称为非法码，通过检查码字的合法性，就得到检错能力，这就是奇偶校验码，只能发现1位错，不具备纠错能力
- (4) 对于 k 位数据位，当给出 r 位校验位时，要发现并改正一位错，须满足如下关系：
 - $2^r \geq k+r+1$
- 要发现并改正一位错，也能发现两位错，则应：
 - $2^{r-1} \geq k+r$

小结

□ 数据表示

- 通过二进制编码表示数据
- 逻辑型
- 字符型
- 整数

□ 检错和纠错

- 通过冗余的编码，使之满足某些规则，来检查编码在传输中是否发生错误，并进行改正
- 检错纠错能力

谢谢



控制器概述 指令与指令系统

2022年秋

主要内容和教学安排

- 第一讲指令系统概述，指令格式，寻址方式
- 第二讲RISC-V指令功能及实现
- 第三讲指令格式，数据通路
- 第四讲单周期**CPU**设计
- 第五讲多周期**CPU**设计
- 第六讲指令流水基本概念
- 第七讲流水中的结构冲突、数据冲突
- 第八讲控制冲突、中断的解决方案
- 第九讲**THINPAD**介绍大实验
- 第十讲大实验辅导及检查(1)
- 第十一讲大实验辅导及检查(2)
- 第十二讲大实验辅导及检查(3)

重点和难点

□ 单条指令功能的实现

- 如何设计指令的数据通路？
- 如何划分指令的执行步骤？
- 如何根据指令得到控制信号？

□ 机器语言程序的自动执行

- 指令之间如何衔接？

□ 提高性能

- 在不增加太多硬件的情况下如何提高性能？

□ 实现途径

- 控制信号生成：组合逻辑或微程序
- 程序自动执行：PC、节拍和下地址
- 指令系统：RISC和CISC
- 提高性能：指令流水

主要的指令集体系结构



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

笔记本电脑，台式机，服务器
(Core i3, i5, i7, M)
x86 Instruction Set



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985;
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility ^[1]
Endianness	Bi (little as default)

智能手机
(iPhone, iPad, Raspberry Pi)
[ARM Instruction Set](#)



MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981;
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

Digital home & networking equipment
(Blu-ray, PlayStation 2)
[MIPS Instruction Set](#)

RISC-V指令集体系结构



- RISC-V是一个基于精简指令集（RISC）原则的开源指令集架构（ISA）。该项目2010年始于Berkeley大学，许多贡献者是该大学以外的志愿者和工业界工作者
- 其设计使其适用于各种现代计算设备（云计算机中的服务器、台式机、智能手机和嵌入式系统等）
- RISC-V 指令使用模块化的设计，包括几个可以互相替换的基本指令集，以及额外可以选择的扩展指令集。所有基本跟扩展的指令集都是由科技产业，研究机构跟学术界合作开发的。基本指令集规范了指令跟他们的编码，控制流程，寄存器数目（以及它们的长度），存储器跟定址方式，逻辑（整数）运算以及其他。只要有软件以及一个通用的编译器的支持，只用基本指令集就可以用来制作一个简单的通用型的电脑。
- 标准的扩展指令集可以搭配所有的基本指令集，以及其他扩展指令集，而不会冲突。

RISC-V模块化设计（基本指令集）



指令集名称	描述	版本	状态
基本指令集			
RV32I	基本整数指令集, 32位	2.0	冻结
RV32E	基本整数指令集 (嵌入式系统), 32位, 16 寄存器	1.9	开放
RV64I	基本整数指令集, 64位	2.0	冻结
RV128I	基本整数指令集, 128位	1.7	开放

RISC-V模块化设计（扩展指令集）

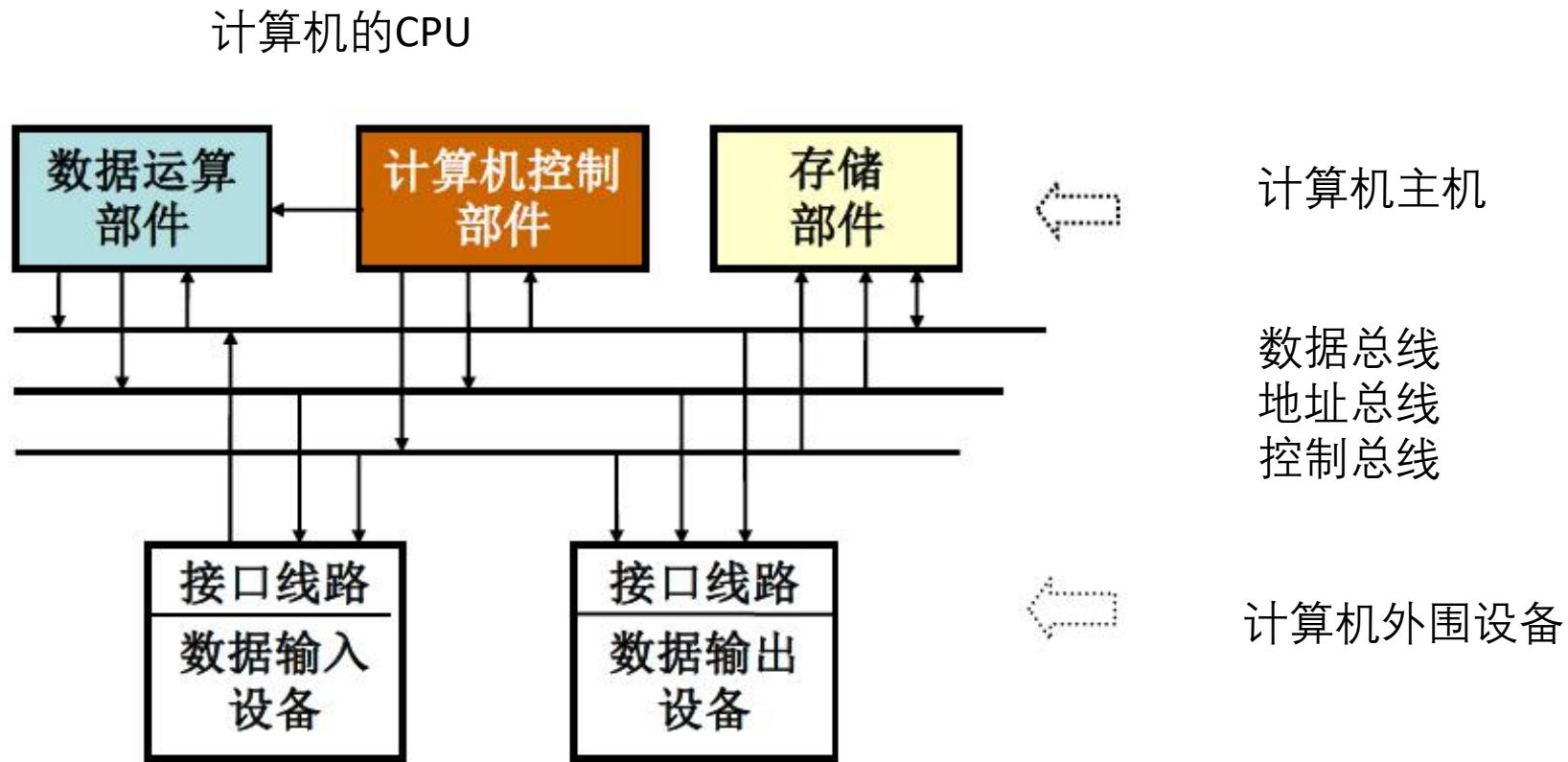


标准扩展指令集			
M	整数乘除法标准扩展	2.0	冻结
A	不可中断指令(Atomic)标准扩展	2.0	冻结
F	单精确度浮点运算标准扩展	2.0	冻结
D	双倍精确度浮点运算标准扩展	2.0	冻结
G	所有以上的扩展指令集以及基本指令集的总和的简称	不适用	不适用
Q	四倍精确度浮点运算标准扩展	2.0	冻结
L	十进制浮点运算标准扩展	0.0	开放
C	压缩指令标准扩展	2.0	冻结
B	位操作标准扩展	0.36	开放
J	动态指令翻译标准扩展	0.0	开放
T	顺序存储器访问标准扩展	0.0	开放
P	单指令多资料流 (SIMD) 运算标准扩展	0.1	开放
V	向量运算标准扩展	0.2	开放
N	用户中断标准扩展	1.1	开放

复杂指令集 (CISC) / 精简指令集 (RISC)

- 早期的趋势：为了能够做复杂的操作，增加越来越多的指令
 - 导致学习和理解上的困难
 - 导致硬件复杂（可能会变慢）
- 改变设计思路：Reduced Instruction Set Computing (RISC)
 - 使用更简单，更小的指令集，会更容易获得快速的硬件
 - 如果需要做复杂操作的话，让软件来做，组合几条简单的指令来完成工作

计算机硬件系统功能部件



控制器的作用

□ 计算机的基本功能

- 执行程序

□ 程序的构成

- 指令序列

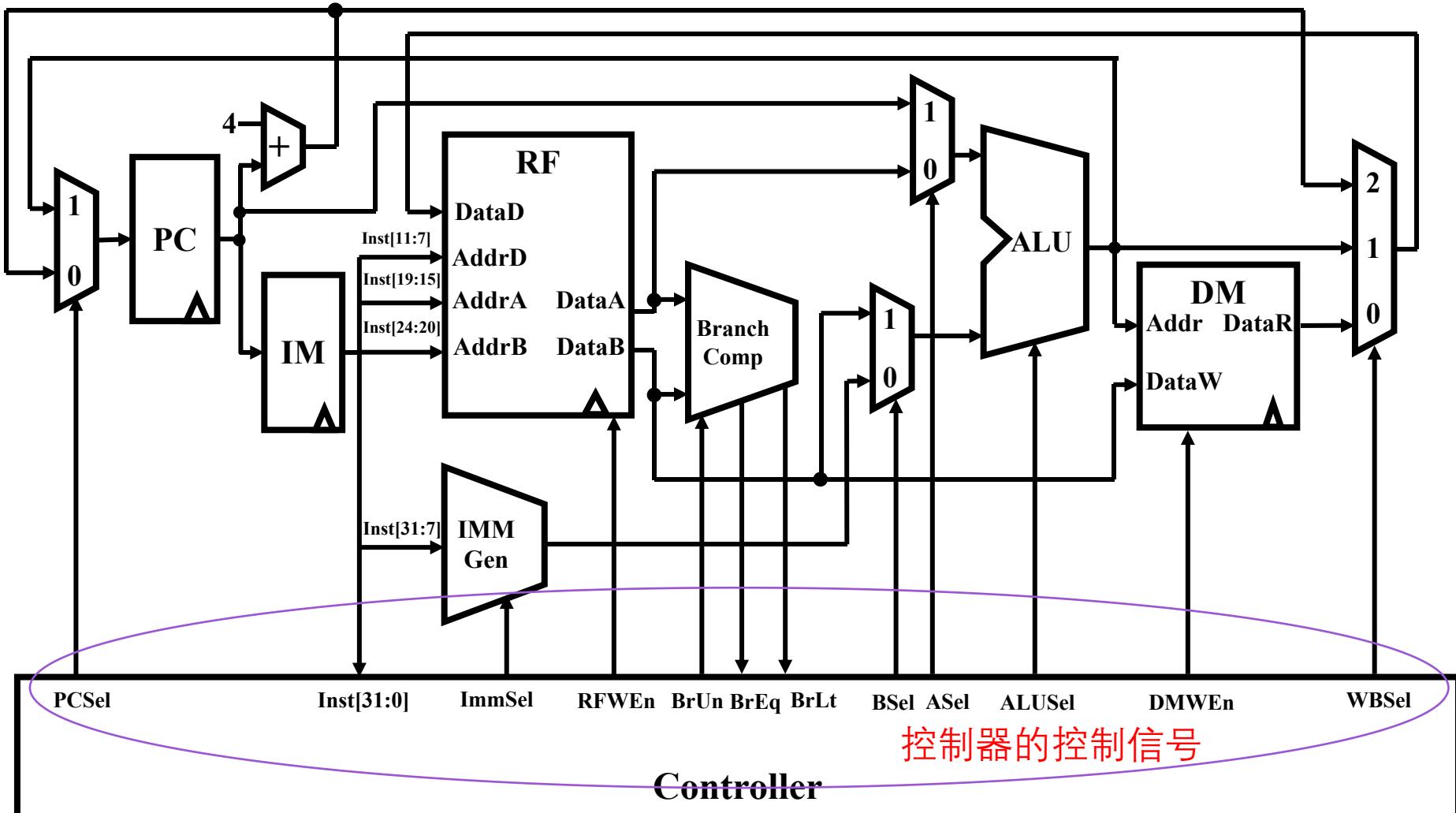
□ 控制器的作用

- 根据指令的要求，提供给各部件相应的控制信号，指挥、协调各部件共同完成指令规定的功能
- 自动执行下一条指令

□ 指令执行

- 取指令
- 分析指令
- 执行指令

单周期CPU



指令与指令系统

□ 指令与指令系统的概念

□ 指令系统设计要求

□ 指令功能和分类

□ 指令格式

- 变长指令字/定长指令字
- 操作码扩展

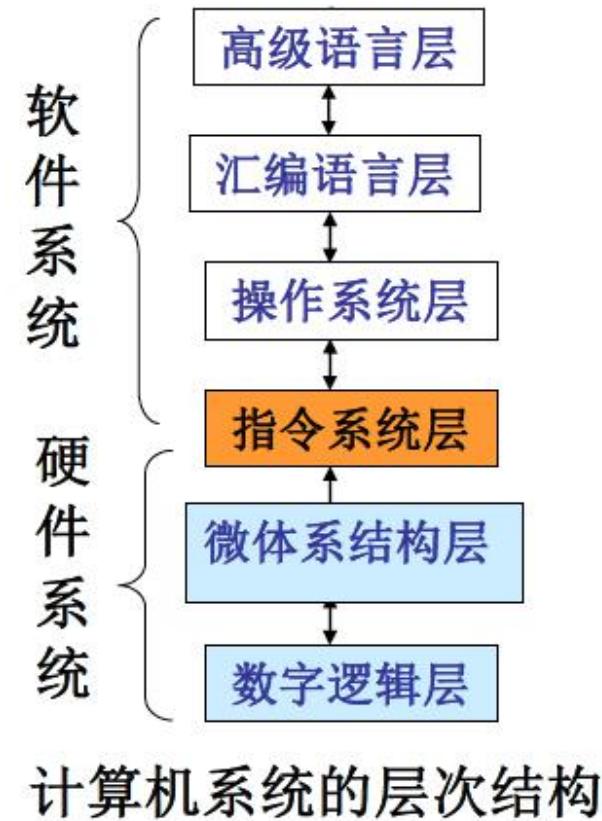
□ 寻址方式

指令与指令系统

- 计算机系统由硬件和软件两大部分组成。硬件指由中央处理器、存储器以及外围设备等组成 的实际装置。软件是为了使用计算机而编写的各种系统的和用户的程序，程序由一个序列的计算机指令组成。
- 指令是计算机运行的最小的功能单元，是指挥计算机硬件运行的命令，是由多个二进制位组成的位串，是计算机硬件可以直接识别和执行的一个信息体。
- 一台计算机提供的全部指令构成该计算机的指令系统。指令用于程序设计人员告知计算机执行一个最基本运算、处理功能，多条指令可以组成一个程序，完成一项预期的任务。

指令系统的地位

- 可以从6个层次分析和看待计算机系统的基本组成。
- 指令系统层处在硬件系统和软件系统之间，是硬、软件之间的接口部分，对两部分都有重要影响。
- 硬件系统用于实现每条指令的功能，解决指令之间的衔接关系；
- 软件由按一定规则组织起来的许多条指令组成，完成一定的数据运算或者事务处理功能。
- 指令系统优劣是一个计算机系统是否成功的关键因素。



指令系统设计要求

□ 完备性

- 指令功能齐全、编程方便

□ 规整性

- 指令格式简单、统一

□ 高效性

- 占内存少，运行高效

□ 兼容性

- 同一系列软件兼容

计算机中需配备的指令

- 指令是用户使用计算机和计算机本身运行的最小的功能单元：①指令是由多个二进制位组成的数串，②用于设计程序，③计算机硬件可直接识别和执行。通常情况下一台计算机需要提供哪些指令呢？
- 计算机用于计算和处理数据，为此，要在计算机硬件系统中设置5种类型的部件：运算器部件、控制器部件、存储器部件、输入设备、输出设备，各自承担数据运算、系统指挥控制、保存当前程序和数据、执行输入和执行输出的功能。需要在计算机中设置为使用和控制这几个部件运行的相应指令。

使用硬件系统的基本指令

JUMP
JRC
CALL
RET

控制器

运算器

ADD
SUB
AND
OR
MVRR
SHR
RCL

高速缓存

STORE
PUSH
LOAD
POP

主存储器

入出接口和总线

OUT
IN

输入设备

外存设备

输出设备

指令的功能和分类

□ 指令用于设计程序，指令系统构成最低级别的程序设计语言，程序设计人员通过指令直接指挥计算机的硬件完成某一个基本的运算、处理功能，例如：

- 对数值数据的算术运算，对逻辑数据的逻辑运算，
- 在计算机部件之间传送、保存数据，
- 从外部向计算机内输入数据，
- 把计算机内部计算结果输出出来，
- 按照某种条件控制计算机选择执行某段程序，
- 当然还有另外一些方面的更深层次的要求等；
- 可以按照指令执行的功能对它们进行分类。

指令的功能和分类

□ 算术与逻辑运算指令

- 加、减、乘、除、变符号等算术运算
- 与、或、非、异或等逻辑运算

□ 移位操作指令

- 算术移位（一般只右移）、逻辑移位、循环移位

□ 数据传送指令

- 通用寄存器之间传送
- 通用寄存器与主存储器存储单元之间传送
- 主存储器不同存储单元之间传送

□ 输入输出指令

- 通用寄存器与输入输出设备（接口）之间传送

指令的功能和分类

□ 转移指令

- 变动程序中指令执行次序的指令，分为无条件转移指令和条件转移指令

□ 子程序调用与返回指令

- 调用指令与返回指令二者要配合使用，子程序的最后一条指令一定是返回指令，执行结束后返回主程序断点

□ 堆栈操作指令

- 堆栈（**stack**）是由若干个连续存储单元组成的先进后出的存储区，有压入（即进栈）和弹出（即退栈）操作

□ 其他指令

- 置条件码指令、开中断指令、关中断指令
- 停机指令、空操作指令、特权指令

指令表示

- 指令中的内容，包括指令操作码（指出指令完成的运算处理功能和数据类型）和操作数或指令的地址（指明用到的数据或地址）两部分。例如：
 - 算逻运算中的运算功能，数据来源或结果去向
 - 数据传送指令中的数据原来位置和新的存储位置
 - 输入输出指令中用到的设备和数据来、去的位置
 - 转移指令的转移类别、转移条件和转移地址等
- 每一条指令必须指明它需要完成的功能，通常用几位指令操作码表示；还需要指明用到的数据、地址或设备，通常在地址字段给出，可能是：
 - (1)寄存器编号，(2)设备端口地址，
 - (3)存储器的单元地址(4) 数值等几种信息。

指令格式与指令字长

□ 指令字长是指组成一条指令的二进制数的位数，例如8 bits、16 bits、32 bits、64 bits等，指令格式与指令字长密切相关，指令字越长可以给出的信息越多。一个指令字通常由指令操作码和操作数地址两部分组成，如何把一个指令字划分成多个字段并分配各字段所表示的内容大有学问。

R型	funct7	rs2	rs1	funct3	rd	opcode
I型	imm[11:0]		rs1	funct3	rd	opcode
S型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB / B型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
UJ / J型		Imm[20,10:1,11,19:12]		rd	opcode	
U型		Imm[31:12]		rd	opcode	

RISC-V 指令格式 (32位)

op	rs	rt	rd	sa	func
op	rs	rt	immediate		
op			target		

MIPS 指令格式 (32位)

指令格式

- 指令字：完整的一条指令的二进制表示
- 指令字长：指令字中二进制代码的位数
 - 机器字长：计算机能直接处理的二进制数据的位数
 - 指令字长（字节倍数）=0.5、1、2...个机器字长
 - 定长指令字结构vs. 变长指令字结构
- 指令格式：指令字中操作码和操作数地址的二进制位的分配方案
 - A horizontal rectangle divided into two equal-width sections by a vertical line. The left section is labeled '操作码' (Operation Code) and the right section is labeled '操作数地址' (Operand Address). Both labels are in orange.
 - 操作码：指明本条指令的操作功能，
 - 每条指令有一个确定的操作码
 - 操作数地址：说明操作数存放的地址，有时是操作数本身

操作码组织与编码

□ 定长的操作码的组织方案

- 在指令字最高位部分分配固定若干位用于表示操作码，有利于简化计算机硬件设计，提高指令译码和识别速度
- 例如：**IBM360机、THINPAD教学机**

□ 变长的操作码的组织方案

- 在指令字最高位部分用一固定长度的字段来表示基本操作码，而对于部分操作数地址位数可以少的指令，则把另外多位辅助操作码扩充到该操作数地址字段，即操作码位数可变。
- 这种方法在不增加指令字长的情况下，可表示更多的指令，但增加了译码和分析难度，要求更多的硬件支持
- 例如：**PDP-11计算机、TEC-2000的8位机**

操作码组织与编码

- 操作码字段与操作数地址字段有所交叉的方案
- 不同指令的操作码长度可以不同，表示操作码所用到的一些二进制位不再集中在指令字的最高位部分，而是与用于表示操作数地址的一些字段有所交叉，操作码还被区分为主操作码和辅助操作码这样不同的两部分，这是一种比较特殊、不很常用的方法。
- 例如：NOVA (DJS-130) 计算机就采用这种方案

指令操作码的位数限制指令系统中的指令条数！

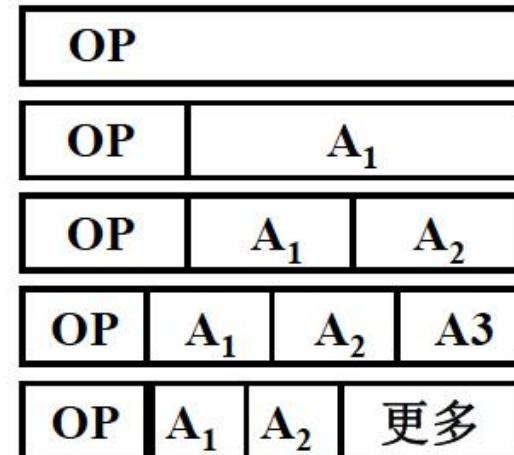
操作数个数与来源

□ 指令操作数个数

- 无操作数指令（零地址指令）
- 单操作数指令（一地址指令）
- 双操作数指令（二地址指令）
- 三操作数指令（三地址指令）
- 多操作数指令（多地址指令）

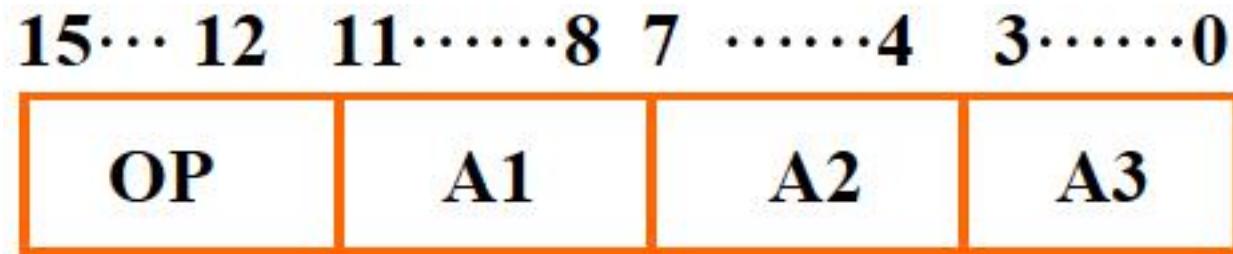
□ 指令操作数来源和去向

- CPU内部的通用寄存器
- 输入输出设备（接口）的一个寄存器
- 主存储器的一个存储单元



指令操作码的扩展技术

- 假设某机器的指令长度为16位，包括4位基本操作码和三个4位地址码段。



- 4位基本操作码可表示16个状态，
- 如用4位操作码，则能表示16条三地址指令，
- 若用8位操作码，则可表示256条二地址指令，
- 而用12位操作码，则可表示4096条一地址指令，
- 若16位全用作操作码，则可表示65536条零地址指令

指令操作码的扩展技术

- 若需要在**16位字长**的指令中能够同时支持三地址、二地址、一地址指令各**15条**，零地址指令**16条**，则可以选用如下方案的变长操作码实现：
- 15条三地址指令的操作码为：0000 ~ 1110
- 15条二地址指令的操作码的高4位选用1111，低4位用0000 ~ 1110，即得到：11110000 ~ 11111110
- 15条一地址指令的操作码的高8位选用11111111，低4位用0000 ~ 1110，即：111111110000 ~ 111111111110
- 16条零地址指令的操作码的高12位每位均用1，低4位随意，即：1111111111110000~1111111111111111

指令操作码扩展技术

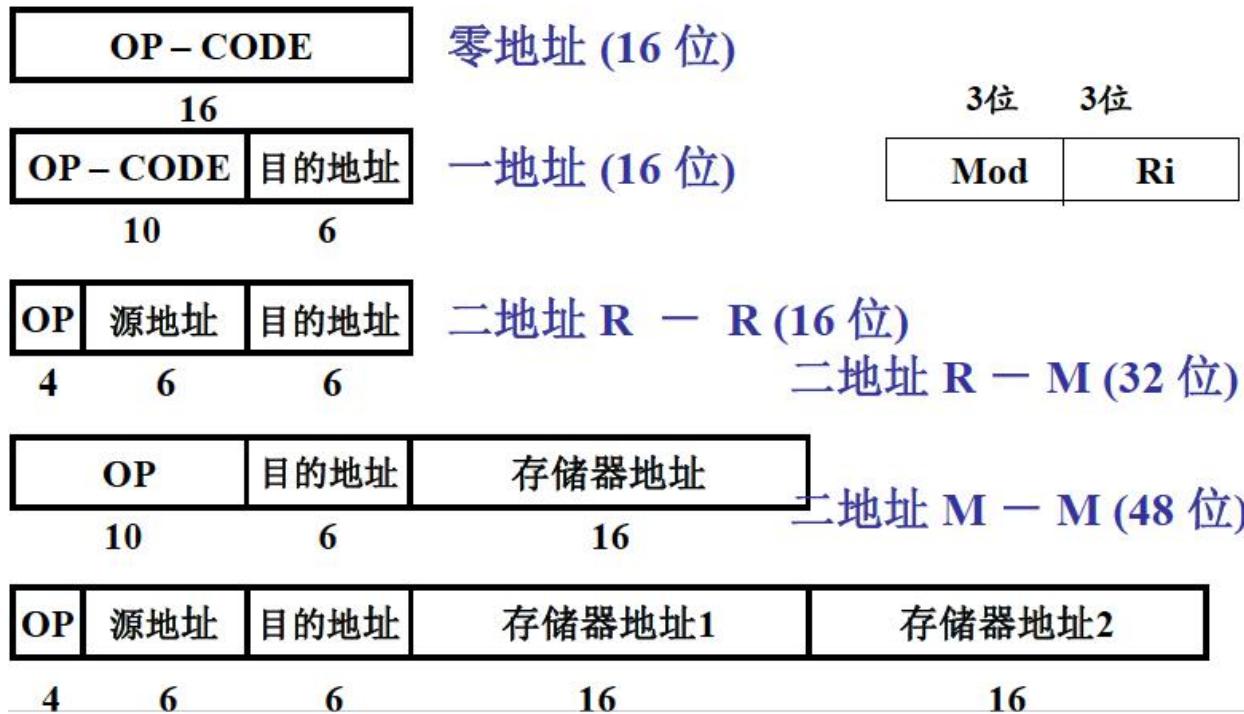
前面介绍的操作码扩展方案中，每次扩展4位并仅保留了一个编码用于接下来的扩展过程，当每次扩展的位数和保留的位数变化时，后面可扩展的指令条数就可以变化。例如在16位字中的指令字中，可以选用如下方案支持三地址指令、二地址指令、一地址指令和零地址指令14、30、31、16条：

- 14条三地址为：**0000 ~ 1101** (保留**1110、1111**两个码)
- 30条二地址为：**11100000 ~ 11111101** (保留2个码)
- 31条一地址为：**111111100000 ~ 111111111110** (保留1个码)
- 16条零地址为：**1111111111110000 ~ 1111111111111111**

指令操作码扩展技术

□ (PDP-11 指令为例)

□ 指令字长有16位、32位、48位三种(1字、2字、3字)



IBM 360指令格式

RR 格式	<table border="1"><tr><td>OP</td><td>R₁</td><td>R₂</td></tr></table>	OP	R ₁	R ₂	二地址 R — R			
OP	R ₁	R ₂						
	8 4 4							
RX 格式	<table border="1"><tr><td>OP</td><td>R₁</td><td>X</td><td>B</td><td>D</td></tr></table>	OP	R ₁	X	B	D	二地址 R — M 基址加变址寻址	
OP	R ₁	X	B	D				
	8 4 4 4 12							
RS 格式	<table border="1"><tr><td>OP</td><td>R₁</td><td>R₃</td><td>B</td><td>D</td></tr></table>	OP	R ₁	R ₃	B	D	三地址 R — M 基址寻址	
OP	R ₁	R ₃	B	D				
	8 4 4 4 12							
SI 格式	<table border="1"><tr><td>OP</td><td>I</td><td>B</td><td>D</td></tr></table>	OP	I	B	D	立即数 — M 基址寻址		
OP	I	B	D					
	8 8 4 12							
SS 格式	<table border="1"><tr><td>OP</td><td>L</td><td>B₁</td><td>D₁</td><td>B₂</td><td>D₂</td></tr></table>	OP	L	B ₁	D ₁	B ₂	D ₂	二地址 M — M 基址寻址
OP	L	B ₁	D ₁	B ₂	D ₂			
	8 8 4 12 4 12							

寻址方式

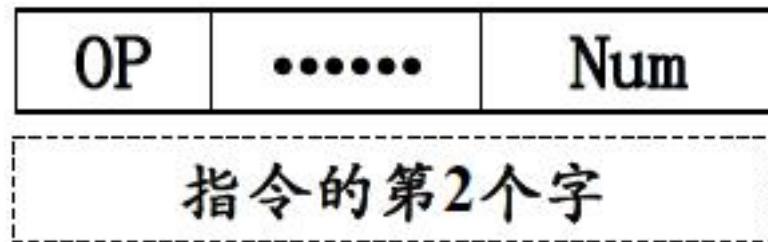
- 寻址方式（又称编址方式）指的是确定本条指令的操作数地址及下一条要执行的指令地址的方法。
- 不同的计算机系统,使用数目和功能不同的寻址方式，其实现的复杂程度和运行性能各不相同。有的计算机寻址方式较少，而有些计算机采用多种寻址方式。
- 通常需要在指令中为每一个操作数专设一个地址字段，用来表示数据的来源或去向的地址。在指令中给出的操作数（或指令）的地址被称为形式地址，使用形式地址信息并按一定规则计算出来或读操作得到的一个数值才是数据（或指令）的实际地址。在指令的操作数地址字段，可能要指出：
 - ①运算器中的累加器的编号或专用寄存器名称（编号）
 - ②输入/输出指令中用到的I/O 设备的入出端口地址
 - ③内存储器的一个存储单元（或一I/O设备）的地址
- 有多种基本寻址方式和某些复合寻址方式,简介如下。

立即数寻址

□ 所需的一个操作数在指令的地址字段部分直接给出。

□ Num 即为操作数的值

□



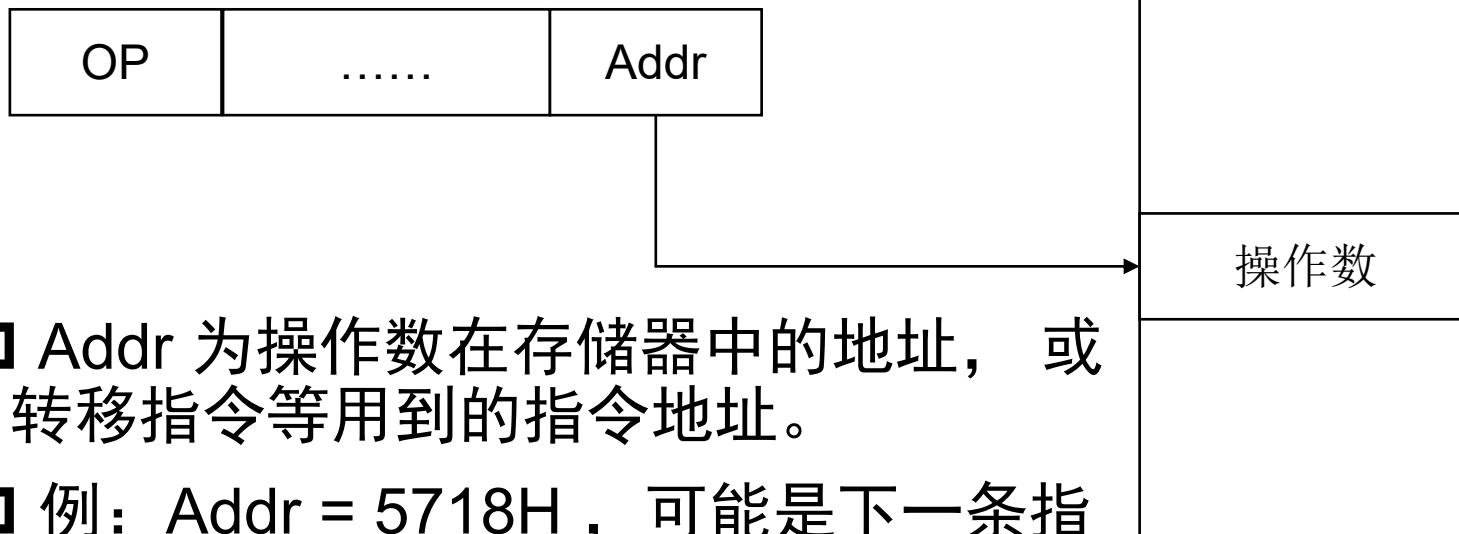
- 适用于操作数固定的情况，取指同时取得操作数，指令执行阶段不必到存储器中取操作数，提高了指令执行速度；
- 当该立即数的值较小(占用位数少)时，可在指令字第一个字中直接给出，否则需要用指令的第二个字提供。

□ 例：Num = 1234H，指令的一个操作数就是1234H

□ 这里的H 表示1234 是16 进制的值

直接寻址

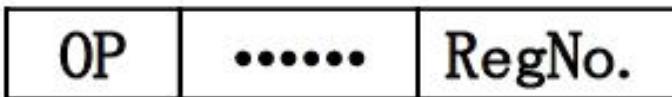
- 在指令的地址码字段，直接给出所需的操作数(或指令)在存储器中的地址。



- Addr 为操作数在存储器中的地址，或转移指令等用到的指令地址。
- 例：Addr = 5718H，可能是下一条指令的地址或一个操作数的地址，若 $[5718H] = 3$ ，则用5718H作地址，从内存储器单元中读出的操作数就是3。

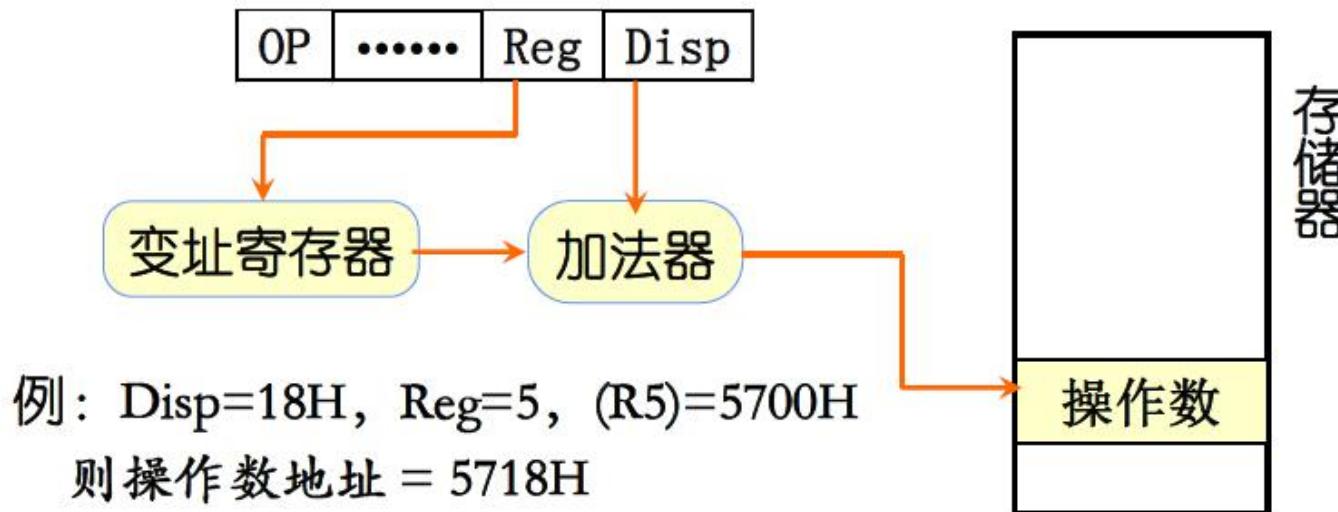
寄存器寻址/寄存器间接寻址

- 计算机的CPU中设置有一定数量的通用寄存器，用于存放操作数、操作数地址或中间结果。假如指令地址码字段给出某一通用寄存器的编号(地址)，且所需的操作数就在这一寄存器中，这就是寄存器寻址方式；若该寄存器中存放的是操作数在内存存储器中所在单元的地址，这就是寄存器间接寻址方式。可通过指令的操作码或另设一个字段，来区分这两种不同的寻址方式。

- 例：RegNo.=5，使用5# 累加器，
- 此时若5# 累加器中的内容为7，可记为(R5)=7，
- 对寄存器寻址，操作数就是寄存器中的数值7
- 对寄存器间接寻址，从内存7# 单元读出来的数才是操作数

变址寻址

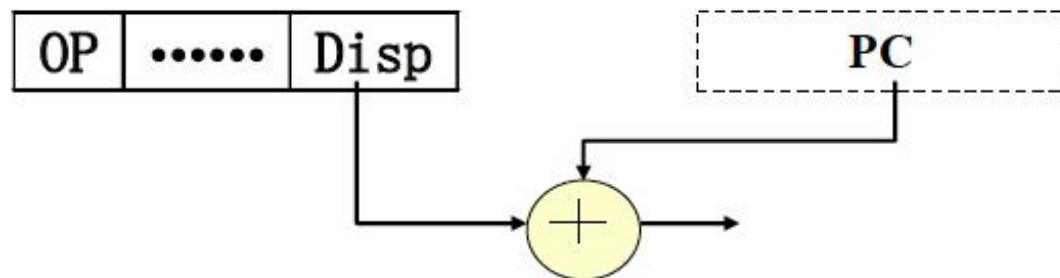
- 操作数的地址由指定的变址寄存器（由Reg指定）的内容和指令中的变址偏移量（Disp）相加得到。



- 变址寄存器内容变化，变址偏移量不变，便于读写数组中的元素，是计算机中常用的一种寻址方式。

相对寻址

- 指令的地址由程序计数器PC 的内容（即当前执行指令的地址）和指令的相对寻址偏移量相加得到。

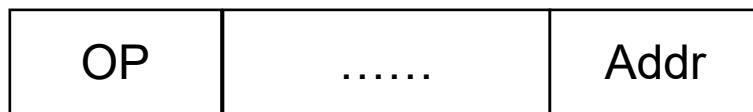


例：Disp = 48H (PC) = 5600H
则实际地址 = 5648H

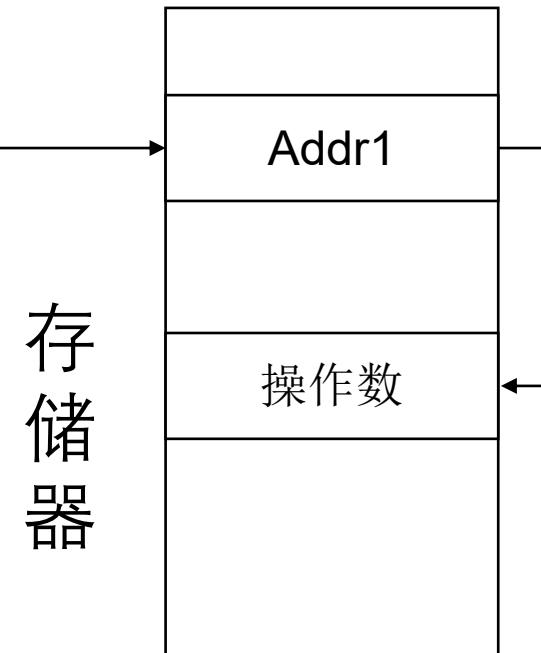
- (1) 主要用于转移指令，对浮动程序很有用。
- (2) 偏移量可正可负，通常用补码表示。

间接寻址

- 指令的地址码字段给出的内容既不是操作数，也不是操作数的地址，而是操作数（或指令）地址的地址，这被称为间接寻址方式，多一次读内存存储器的操作。

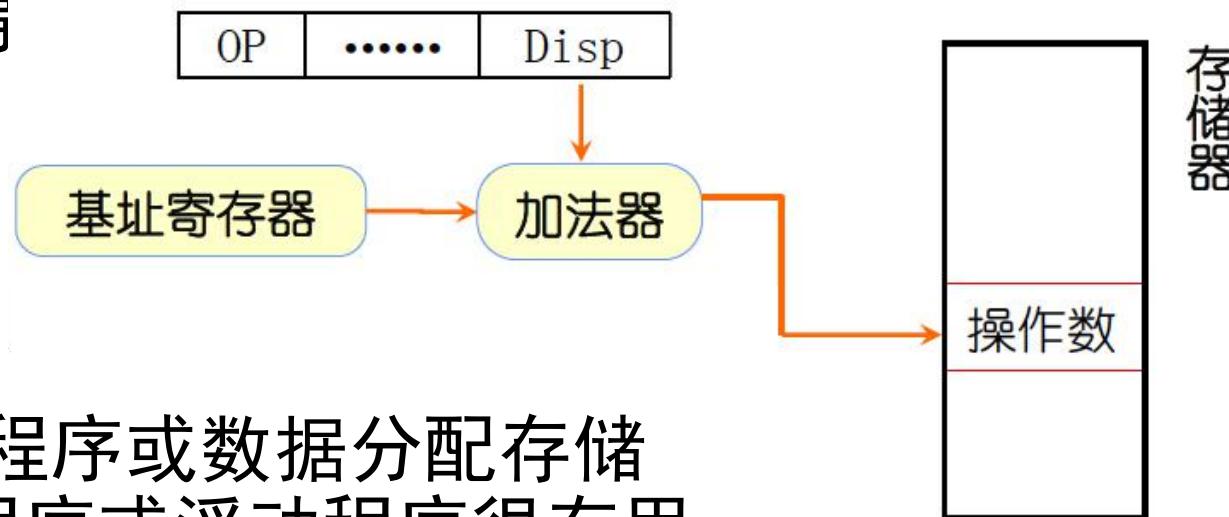


- 指令中的Addr可以用其他寻址方式给出，例如变址寻址，这就成为变址寻址与间接寻址的复合寻址方式



基址寻址

- 在计算机中设置一个专用的基址寄存器，操作数（或指令）的地址通过基址寄存器的内容和指令中的地址码相加得“”

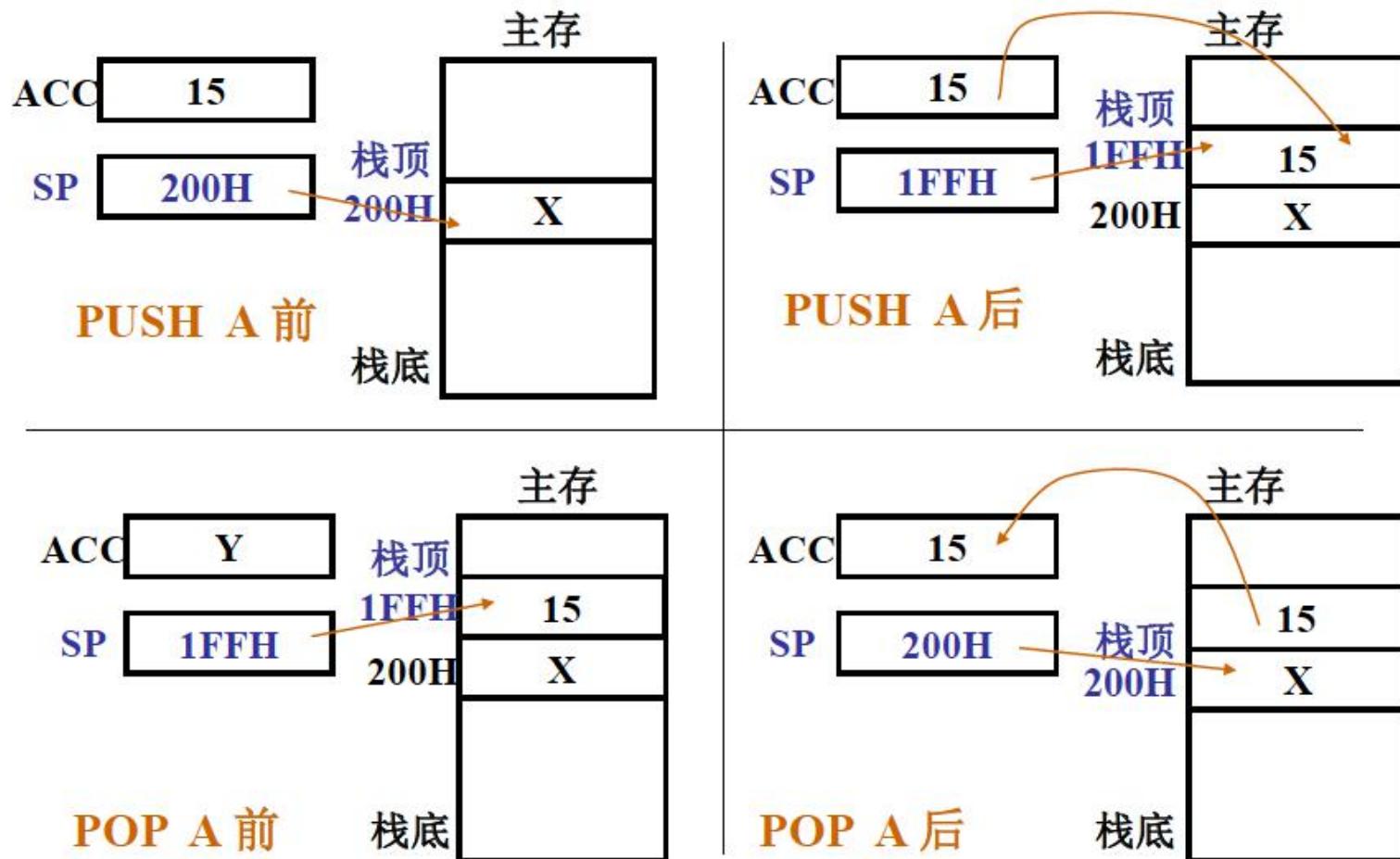


- 主要用于为程序或数据分配存储区，对多道程序或浮动程序很有用，解决了程序在存储器中的定位和扩大寻址空间等问题。

堆栈寻址

- 堆栈是内存储器中一块按“后进先出”原则进行读写的存储区，并通过一个专用的寄存器(称为**堆栈指针SP**)给出堆栈的栈顶地址，执行读写堆栈操作通常总在栈顶进行，故不必在指令中给出堆栈地址，而且在读写操作的前后伴随有自动修改SP内容的动作，确保使SP总是指向堆栈的栈顶。例如，**按字寻址时**：
- **入栈操作:** $SP - 1 \rightarrow SP$ 和 AR ，即SP的内容减1存回SP，并送入内存地址寄存器，接下来才可以把数据写到堆栈中，这是因为需要把数据写到新开辟出来的栈顶单元中。
- **出栈操作:** $SP \rightarrow AR$ ，完成一次读堆栈操作后，还要执行一次 $SP + 1 \rightarrow SP$ 的操作，用于修改SP内容，这是因为数据读出后原来它的下一个相邻单元变成为栈顶。

堆栈寻址举例



小结

□ 控制器的主要功能

- 正确执行指令规定的功能
- 自动、连续执行指令

□ 指令系统

- 是硬件和软件的接口
- CISC和RISC

□ 指令格式

- 指令如何用二进制编码表示，主要是操作码和操作数地址的安排方案

□ 寻址方式

- 操作数寻址方式
- 对操作系统、编译程序提供支持，方便程序员使用计算机
- 寻址方式的选择

谢谢



RISC-V指令系统

2022年秋

主要内容

□ RISC-V指令系统概述

□ RISC-V指令集与汇编语言概述

- 算术指令、逻辑指令、移位指令
- 数据传输指令（访存指令）
- 比较指令、有条件跳转指令、无条件跳转指令
- 伪指令
- 函数调用

RISC-V指令系统概述

RISC-V指令集历史

- 加州大学伯克利分校Krste Asanovic教授、Andrew Waterman和Yunsup Lee等开发人员于2010年发明。
 - 其中"RISC"表示精简指令集，而其中"V"表示伯克利分校从RISC I开始设计的第五代指令集。
- 基于BSD协议许可的免费开放的指令集架构
- 适合多层次计算机系统
 - 从 微控制器 到 超级计算机
 - 支持大量定制与加速功能
 - 32bit, 64bit, 128bit
- 规范由RISC-V非营利性基金会维护
 - RISC-V基金会负责维护RISC-V指令集标准手册与架构文档

RISC-V架构的特点

□ 指令集架构简单

- 指令集的238页，特权级编程手册135页，其中RV32I只有16页
- 作为对比，Intel的处理器手册有5000多页
- 新的体系结构设计吸取了经验和最新的研究成果
- 指令数量少，基本的RISC-V指令数目仅有40多条，加上其他的模块化扩展指令总共几十条指令。

□ 模块化的指令集设计

- 不同的部分还能以模块化的方式组织在一起
- ARM的架构分为A、R和M三个系列，分别针对于Application（应用操作系统）、Real-Time（实时）和Embedded（嵌入式）三个领域，彼此之间并不兼容
- RISC-V嵌入式场景，用户可以选择RV32IC组合的指令集，仅使用Machine Mode（机器模式）；而高性能操作系统场景则可以选择譬如RV32IMFDC的指令集，使用Machine Mode（机器模式）与User Mode（用户模式）两种模式，两种使用方式的共同部分相互兼容

RISC-V的模块化设计

- RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示
- RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集，使用该整数指令子集，便能够实现完整的软件编译器
- 其他的指令子集部分均为可选的模块，具有代表性的模块包括M/A/F/D/C

RISC-V模块化设计（基本指令集）



指令集名称	描述	版本	状态
基本指令集			
RV32I	基本整数指令集, 32位	2.0	冻结
RV32E	基本整数指令集 (嵌入式系统), 32位, 16 寄存器	1.9	开放
RV64I	基本整数指令集, 64位	2.0	冻结
RV128I	基本整数指令集, 128位	1.7	开放

RISC-V模块化设计（扩展指令集）



标准扩展指令集			
M	整数乘除法标准扩展	2.0	冻结
A	不可中断指令(Atomic)标准扩展	2.0	冻结
F	单精确度浮点运算标准扩展	2.0	冻结
D	双倍精确度浮点运算标准扩展	2.0	冻结
G	所有以上的扩展指令集以及基本指令集的总和的简称	不适用	不适用
Q	四倍精确度浮点运算标准扩展	2.0	冻结
L	十进制浮点运算标准扩展	0.0	开放
C	压缩指令标准扩展	2.0	冻结
B	位操作标准扩展	0.36	开放
J	动态指令翻译标准扩展	0.0	开放
T	顺序存储器访问标准扩展	0.0	开放
P	单指令多资料流 (SIMD) 运算标准扩展	0.1	开放
V	向量运算标准扩展	0.2	开放
N	用户中断标准扩展	1.1	开放

用户可以扩展自己的指令子集，RISC-V预留了大量的指令编码空间用于用户的自定义扩展，同时，还定义了四条Custom指令可供用户直接使用，每条Custom指令都有几个比特位的子编码空间预留，因此，用户可以直接使用四条Custom指令扩展出几十条自定义的指令。

可配置的通用寄存器组

□ 寄存器组主要包括通用寄存器（General Purpose Registers）和控制状态寄存器（Control and Status Registers）

- 32位架构(RV32I)32个32位的通用寄存器，64位架构(RV64I)32个64位的通用寄存器
- 嵌入式架构RV32E有16个32位的通用寄存器
- 支持单精度浮点数(F)，或者双精度浮点数(D)，另外增加一组独立的通用浮点寄存器组，f0~f31

□ CSR寄存器用于配置或记录一些运行的状态（后续异常和中断处理中会详细描述）

- CSR寄存器是处理器核内部的寄存器，使用专有的12位地址码空间

规整的指令编码

- 所有通用寄存器在指令码的位置是一样的，方便译码阶段的使用
- 所有的指令都是32位字长，有 6 种指令格式：寄存器型，立即数型，存储型，分支指令、跳转指令和大立即数

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB / B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
UJ / J 型		Imm[20,10:1,11,19:12]			rd	opcode
U 型		Imm[31:12]			rd	opcode

RISC-V的数据传输指令

- 专用内存到寄存器之间传输数据的指令，其它指令都只能操作寄存器
 - 简化硬件设计
 - 支持字节（8位），半字（16位），字（32位），双字（64位，64位架构）的数据传输
 - 推荐但不强制地址对齐
 - 小端机结构

RISC-V的特权模式

- RISC-V架构定义了三种工作模式，又称特权模式（Privileged Mode）：
 - Machine Mode：机器模式，简称M Mode
 - Supervisor Mode：监督模式，简称S Mode
 - User Mode：用户模式，简称U Mode
- RISC-V架构定义M Mode为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统
- 在异常和中断处理中会详细讨各个特权模式的机制

RISC-V 的指令集

□ RISC-V官方指令集手册

<https://riscv.org/specifications/isa-spec-pdf/>

□ 中文简化版

<http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf>

Base Integer Instructions: RV32I and RV64I				RV Privileged Instructions				
Category	Name	Fmt	RV32I Base	+RV64I	Category	Name	Fmt	RV mnemonic
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2	Trap	Mach-mode trap return	R	MRRET
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt		Supervisor-mode trap return	R	SRET
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2	Interrupt	Wait for Interrupt	R	WFI
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt	MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2	Examples of the 60 RV Pseudoinstructions			
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt	Branch = 0 (BRQ rs,x0,imm)	J	BEQZ rs,imm	
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2	Jump (uses JAL x0,imm)	J	J imm	
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm	MoVe (uses ADDI rd,rs,0)	R	MV rd,rs	
	Subtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2	REturn (uses JALR x0,0,rs)	I	RET	
	Load Upper Imm	U	LUI rd,imm		Optional Compressed (16-bit) Instruction Extension: RV32C			
Add Upper Imm to PC	U	AUIPC rd,imm		Category Name Fmt RVC RISC-V equivalent	Category	Name	Fmt	RVC
Logical	XOR	R	XOR rd,rs1,rs2	CL C.LW rd',rs1',imm	Loads	Load Word	CL	LW rd',rs1',imm^4
	XOR Immediate	I	XORI rd,rs1,imm	CL C.IWGP rd,imm		Load Word SP	CI	LW rd,sp,imm^4
	OR	R	OR rd,rs1,rs2	CL C.FLW rd',rs1',imm		Float Load Word	CL	FLW rd',rs1',imm^8
	OR Immediate	I	ORI rd,rs1,imm	CL C.FLGP rd,imm		Float Load Word SP	CI	FLW rd,sp,imm^8
	AND	R	AND rd,rs1,rs2	CL C.FLD rd',rs1',imm		Float Load Double	CL	FLD rd',rs1',imm^16
	AND Immediate	I	ANDI rd,rs1,imm	CL C.FLDGP rd,imm		Float Load Double SP	CI	FLD rd,sp,imm^16
Compare	Set <	R	SLT rd,rs1,rs2	CS C.SW rd',rs2',imm	Stores	Store Word	CS	SW rd',rs2',imm^4
	Set < Immediate	I	SLTI rd,rs1,imm	CS C.SWP rd,imm		Store Word SP	CI	SW rd,sp,imm^4
	Set < Unsigned	R	SLTU rd,rs1,rs2	CS C.PFW rd',rs2',imm		Float Store Word	CS	PFW rd',rs2',imm^8
	Set < Unsigned Imm	I	SLTIU rd,rs1,imm	CS C.PFGWP rd,imm		Float Store Word SP	CSS	PFGW rd,sp,imm^8
Branches	Branch =	B	BEQ rs1,rs2,imm	CS C.PFD rd',rs2',imm		Float Store Double	CS	PFD rd',rs2',imm^16
	Branch ≠	B	BNE rs1,rs2,imm	CS C.PFGDP rd,imm		Float Store Double SP	CSS	PFGDP rd,sp,imm^16
	Branch <	B	BLT rs1,rs2,imm		Category Name Fmt RVC RISC-V equivalent	Arithmetic	ADD	CR C.ADD rd,rs1
	Branch ≥	B	BGE rs1,rs2,imm			ADD Immediate	CI	C.ADDI rd,imm
	Branch < Unsigned	B	BLTU rs1,rs2,imm			ADD SP Imm * 16	CI	C.ADDI16SP x0,imm
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm			ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm
Jump & Link	JBL	J	JAL rd,imm	CR C.SUB rd,rs1		SUB	CR	SUB rd,rd,rs1
	Jump & Link Register	I	JALR rd,rs1,imm	CR C.AND rd,rs1		AND	CR	AND rd,rd,rs1
Synch	Synch thread	I	FENCE	CI C.ANDI rd,imm		AND Immediate	CI	ANDI rd,imm
	Synch Instr & Data	I	FENCE.I	CR C.OR rd,rs1		OR	CR	OR rd,rd,rs1
Environment	CALL	I	ECALL	CR C.XOR rd,rs1		eXclusive OR	CR	XOR rd,rd,rs1
	BREAK	I	EBREAK	CR C.MV rd,rs1		MoVe	CR	MV rd,rs1,x0
				CI C.LI rd,imm		Load Immediate	CI	ADDI rd,x0,imm
				CI C.LUI rd,imm		Load Upper Imm	CI	LUI rd,imm
Control Status Register (CSR)	Read/Write	I	CSRWR rd,csr,rs1	CI C.SLLI rd,imm	Shifts	Shift Left Imm	CI	SLLI rd,rd,imm
	Read & Set Bit	I	CSRRS rd,csr,rs1	CI C.SRAI rd,imm		Shift Right Arith. Imm.	CI	SRAI rd,rd,imm
	Read & Clear Bit	I	CSRRC rd,csr,rs1	CI C.SRLI rd,imm		Shift Right Log. Imm.	CI	SRLI rd,rd,imm
	Read/Write Imm	I	CSRWRNI rd,csr,imm	CB C.BEQZ rd',imm	Branches	Branch=0	CB	BEQ rd',x0,imm
	Read & Set Bit Imm	I	CSRRSNI rd,csr,imm	CB C.BNEZ rd',imm		Branch≠0	CB	BNE rd',x0,imm
	Read & Clear Bit Imm	I	CSRRCI rd,csr,imm			Jump	CJ	J imm
						Jump Register	CR	JALR x0,rs1,0
Loads	Load Byte	I	LB rd,rs1,imm			Jump & Link	CJ	JAL x0,imm
	Load Halfword	I	LH rd,rs1,imm				CR	JALR x0,rs1,0
	Load Byte Unsigned	I	LBU rd,rs1,imm		Optional Compressed Extension: RV64C			
	Load Half Unsigned	I	LBU rd,rs1,imm		All RV32C (except C.JAL, 4 word loads, 4 word stores) plus:			
	Load Word	I	LW rd,rs1,imm		ADD Word (C.ADDW)		Load Doubleword (C.LD)	
Stores	Store Byte	S	SB rs1,rs2,imm		ADD Imm. Word (C.ADDIW)		Load Doubleword SP (C.LDSP)	
	Store Halfword	S	SH rs1,rs2,imm		SUBtract Word (C.SUBW)		Store Doubleword (C.SD)	
	Store Word	S	SW rs1,rs2,imm				Store Doubleword SP (C.SDSP)	

32-bit Instruction Formats										16-bit (RVC) Instruction Formats																					
R	31	27	26	25	24	20	19	15	14	12	11	7	6	0	CR	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															CR	funct64	rd/rs1		rs2		op										
I															CI	funct3	imm	rd/rs1		imm		op									
S															CSS	funct3	imm		rs2		op										
B															CIW	funct3	imm		rd'		op										
															CL	funct3	imm	rs1'	imm	rd'	op										
															CS	funct3	imm	rs1'	imm	rs2'	op										
															CB	funct3	offset		offset		op										
															CJ				jump target		op										

RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers x1-x31 and the PC are 32 bits wide in RV32I and 64 in RV64I (x0=0). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

算术指令、逻辑指令、 移位指令

RISC-V指令集与汇编语言概述

RISC-V 汇编

□ 汇编指令格式

op dst, src1, src2

- 1个操作码，3个操作数
- op 操作的名字
- dst 目标寄存器
- src1 第一个源操作数寄存器
- src2 第二个源操作数寄存器

□ 通过一些限制来保持硬件简单

RISC-V 汇编格式

- 每一条指令只有一个操作，每一行最多一条指令
- 汇编指令与C语言的操作相关 (=, +, -, *, /, &, |, 等)
 - C语言中的操作会被分解为一条或者多条汇编指令
 - C语言中的一行会被编译为多行RISC-V汇编程序

RISC-V 中的寄存器

□ 在RISC-V中有32个寄存器（x0-x31）

- 每个寄存器的长度为32位
- X0是一个特殊的寄存器，只用于全零
- 每一个寄存器都有自己的别名，用于软件的使用的惯例，但是实际的硬件并没有任何区别

RISCV寄存器

寄存器	ABI名字	描述	保存者Saver
x0	zero	Hard-wired zero	--
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5	t0	Temporary/alternative link register	Caller
x6-7	t1-2	temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	temporaries	Caller

算术指令

- 变量a, b和c被分别放置在寄存器x1, x2, 和x3中
- 整数的加法 (add)
- C: $a = b + c$
- RISC-V: add x1, x2, x3
- 整数的减法 (sub)
- C: $a = b - c$
- RISC-V: sub x1, x2, x3

RISC-V 程序举例

- 假设 $a \rightarrow x_1, b \rightarrow x_2, c \rightarrow x_3, d \rightarrow x_4, e \rightarrow x_5$ 。下面会将一段C语言程序编译成RISC-V汇编指令

$$a = (b + c) - (d + e);$$

add x6, x4, x5 # tmp1 = d + e

add x7, x2, x3 # tmp2 = b + c

sub x1, x7, x6 # a = (b + c) - (d + e)

- 指令执行顺序反映了源程序的计算过程
- 指令中可以看到如何使用临时寄存器
- #符号后面是程序的注释

特殊的寄存器zero

- 0在程序中很常见， 拥有一个自己的寄存器
- x0, 或者zero是一个特殊的寄存器， 只拥有值0，并且不能被改变
 - 注意，在任意的指令中， 如果使用x0作为目标寄存器， 将没有任何效果， 仍然保持0不变
- 使用样例

- add x3, x0, x0 # c=0
 - add x1, x2, x0 # a=b
 - add x0, x0, x0 #nop

RISC-V 中的立即数

- 数值常数被称为是立即数 (immediates)
- 立即数有特殊的指令语法：

opi dst, src, imm

- 操作码的最后一个字符为i的，会将第二个操作数认为是一个立即数（经常用后缀来指明操作数的类型，例如无符号数unsigned的后缀为u）

- 指令举例

- addi x1, x2, 5 # a=b+5
- addi x3, x3, 1 # c++

- 问题：为何没有subi指令？

算术操作的溢出

- 溢出是因为计算机中表达数本身是有范围限制的
 - 计算的结果没有足够多的位数进行表达
- RISC-V 忽略溢出问题，高位被截断，低位写入到目标寄存器中

RISC-V 乘法与除法指令

- 积的长度是乘数和被乘数长度的和。将两个32位数相乘得到的是64位的乘积。
- 为了正确地得到一个有符号或无符号的64位积，RISC-V中带有四个乘法指令。
 - 要得到整数32位乘积（64位中的低32位）就用mul指令。
 - 要得到高32位，如果操作数都是有符号数，就用mulh指令；
 - 如果操作数都是无符号数，就用mulhu指令；
 - 如果一个有符号一个无符号，可以用mulhsu指令；
 - 如果需要获得完整的64位值，建议的指令序列为mulh[[s]u]rdh, rs1, rs2; mul rdl, rs1, rs2 (源寄存器必须使用相同的顺序，rdh要注意和rs1和rs2都不相同。这样底层的微体系结构就会把两条指令合并成一次乘法操作，而不是两次乘法操作)

除法指令举例

```
# mod using div: x5 = x6 mod x7  
mod:  
div x5, x6, x7          # x5 = x6/x7  
rem x5, x6, x7          # x5 = x6 mod x7
```

位操作指令

Note: a→x1, b→x2, c→x3

Instruction	C	RISC-V
And	a = b & c;	and x1, x2, x3
And Immediate	a = b & 0x1;	andi x1, x2, 0x1
Or	a = b c;	or x1, x2, x3
Or Immediate	a = b 0x5;	ori x1, x2, 0x5
Exclusive Or	a = b ^ c;	xor x1, x2, x3
Exclusive Or Immediate	a = b ^ 0xF;	xori x1, x2, 0xF

移位指令

- 左移相当于乘以2
 - 左移右边补0
 - 左移操作更快
- 逻辑右移：在最高位添加0
- 算术右移：在最高位添加符号位
- 移位的位数可以是立即数或者寄存器中的值

移位指令

Instruction Name	RISC-V
Shift Left Logical	sll rd, rs1, rs2
Shift Left Logical Imm.	slli rd, rs1, shamt
Shift Right Logical	srl rd, rs1, rs2
Shift Right Logical Imm.	srlti rd, rs1, shamt
Shift Right Arithmetic	sra rd, rs1, rs2
Shift Right Arithmetic Imm.	srai rd, rs1, shamt

□ slli, srli, srai只需要最多移动63位 (对64位寄存器), 只会使用immediate低6位的值 (I类型指令)

助记符的后缀

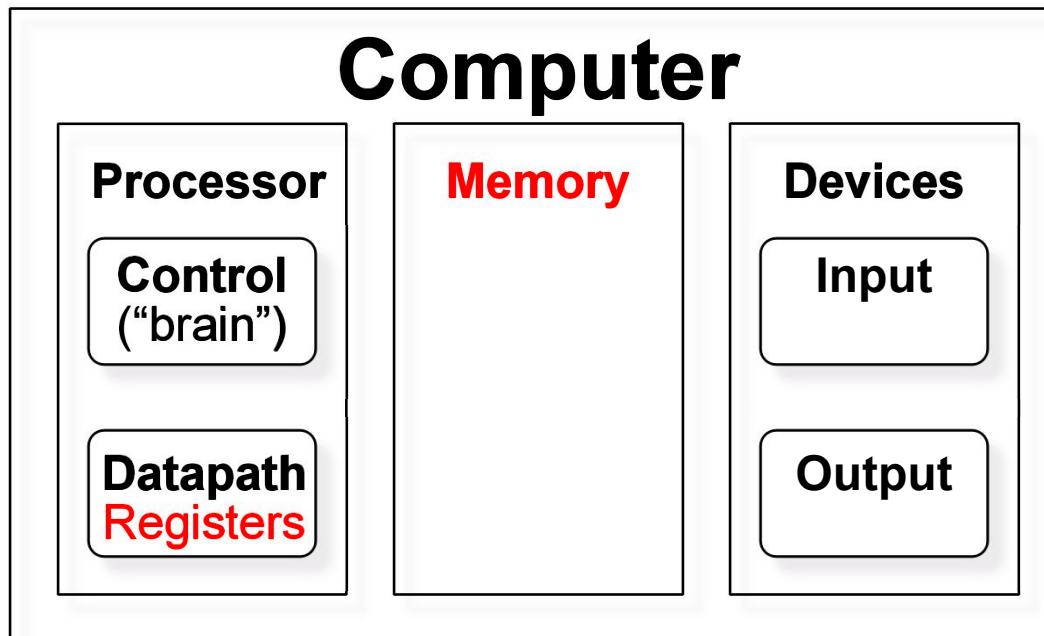
- i = “immediate” 是个整型的常量
- u = “unsigned” 无符号类型

数据传输指令（访存指令）

RISC-V指令集与汇编语言概述

数据传输指令

- 数据传输指令在寄存器（数据通路）和内存之间传输数据
 - 从内存中取出操作数或者将操作数保存到内存中



数据传输

- C语言中的变量会映射到寄存器中；而其它的大量的数据结构，例如数组会映射到内存中
- 内存是一维的数组，地址从0开始
- 所有的RISC-V的指令操作（除load/store）只会在寄存器中操作
- 特殊的数据传输指令在寄存器和内存之间传输数据
 - Store指令：从寄存器到内存
 - Load指令：从内存到寄存器

数据传输指令的格式

□ 数据传输指令的格式

memop reg, off (bAddr)

□ memop = 操作的名字 (load或者store)

□ reg = 寄存器的名字，源寄存器或者目标寄存器

□ bAddr = 指向内存的基地址寄存器 (base address)

□ off = 地址偏移，字节寻址，为立即数 (offset)

□ 访问的内存地址为 bAddr + off

□ 必须指向一个合法的地址

内存的字节寻址方式

- 在现代计算机中操作以8bits为单位，即一个字节
 - 一个word的定义依据不同的体系结构定义不同，这里定义1 word = 4 bytes
 - 内存是按照地址进行编址的，不是按照字进行编址的
- 字地址之间有4个字节的距离
 - 字的地址为其最低位的字节的地址
 - 按字对齐的话地址最后两位为0（地址为4的倍数）
- C语言会自动按照数据类型来计算地址，在汇编中需要程序员自己计算

...
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

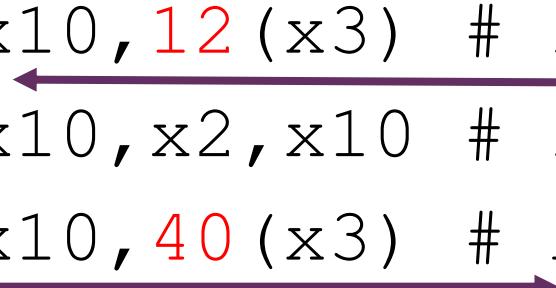
数据传输指令

□ 装入一个字(lw)

□ 写出一个字(sw)

□ 指令举例

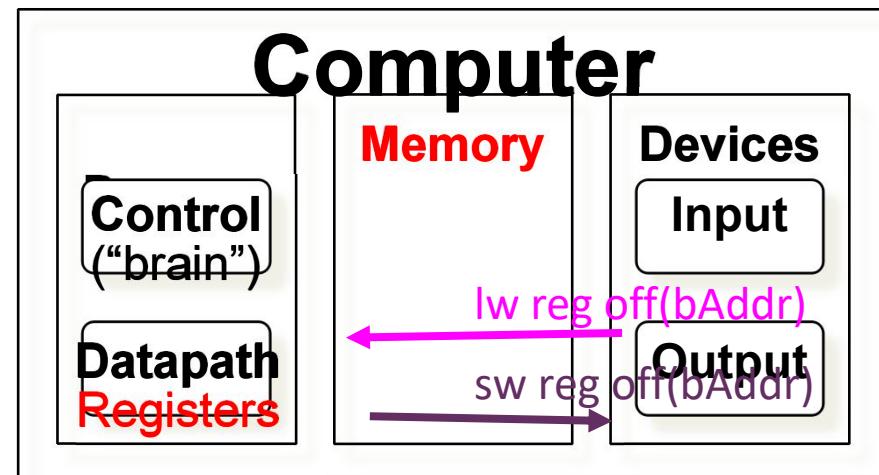
```
# addr of int A[] -> x3, a -> x2
lw    x10, 12(x3) # x10=A[3]
add  x10, x2, x10 # x10=A[3]+a
sw    x10, 40(x3) # A[10]=A[3]+a
```



内存与变量的大小 (variable size)

□ 数据传输指令

- `lw reg, off(bAddr)`
- `sw reg, off(bAddr)`
- `off+bAddr` 必须按照字进行对齐，即4的倍数
- 例如整数的数字
每个整数32位=4字节
- 如何传输1个字符的数据或者传输一个short的数据
(2个字节)
都不是4个字节的整数倍



传输一个字节数据

□ 还是使用字类型指令，配合位的掩码来达到目的

```
lw      x11, 0(x1)
```

```
andi x11, x11, 0xFF # lowest byte
```

□ 或者，使用字节传输指令

```
lb      x11, 1(x1)
```

```
sb      x11, 0(x1)
```

□ 上述指令无需字对齐

* (x0) = 0x000000180

00	00	01	80
----	----	----	----

字节排布

- 大端机：最高的字节在最低的地址，字的地址等于最高字节的地址
- 小端机：最低的字节在最低的地址，字的地址等于最低字节的地址

高地址 * (\$s0) = 0x00000180 低地址

80	01	00	00
----	----	----	----

Big Endian

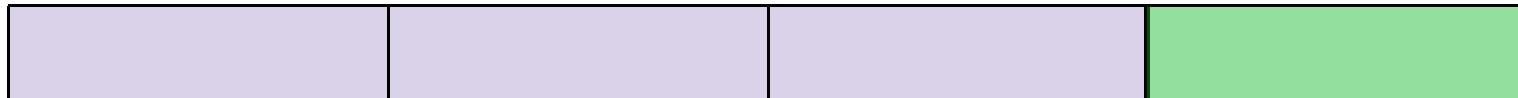
00	00	01	80
----	----	----	----

Little Endian

RISC-V 是 小端机

字节数据传输指令

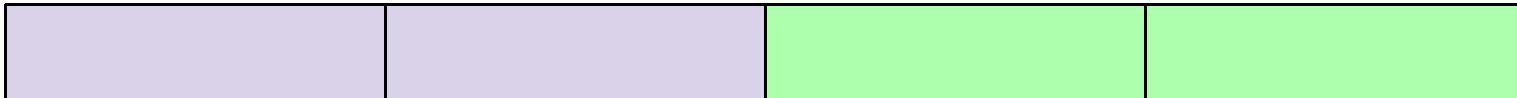
- lb/sb使用的是最低的字节
- 如果是sb指令，高24位被忽略
- 如果是lb指令，高24位做符号扩展



- 例如：let * (x1) = 0x00000180:

lb x11, 1(x1)	# x11=0x00000001
lb x12, 0(x1)	# x12=0xFFFFFFF80
sb x12, 2(x1)	# * (x1)=0x00800180

半字数据传输指令



- `lh reg, off(bAddr)` “load half”
- `sh reg, off(bAddr)` “store half”
 - `off(bAddr)` 必须是 2 的倍数
 - `sh` 指令中高 16 位忽略
 - `lh` 指令中高 16 位做符号扩展

无符号的版本

- `lhu reg, off(bAddr)` “load half unsigned”
- `lbu reg, off(bAddr)` “load byte unsigned”
 - `l(h)u` 指令, 高位都做 0 扩展

分支与跳转指令

RISC-V指令集与汇编语言概述

比较指令

- Set Less Than (slt)
 - `slt dst, reg1, reg2`
 - If value in `src1 < value in src2`, `dst = 1`, else `0`
- Set Less Than Immediate (slti)
 - `slti dst, reg1, imm`
 - If `value in reg1 < imm`, `dst = 1`, else `0`
- 如何完成无符号数的比较？

在 RISC-V 指令中有无符号数比较

- Unsigned versions of `slt(i)`:
 - `sltu` `dst,src1,src2:unsigned comparison`
 - `sltiu` `dst,src,imm: unsigned comparison against constant`

- 例子:

```
addi x10,x0,-1    # x10=0xFFFFFFFF  
slti x11,x10,1    # x11=1 (-1 < 1)  
sltiu x12,x10,1   # x12=0 (232-1 >>> 1)
```

RISC-V 中的有符号与无符号

□ 有符号和无符号在3个上下文环境中

- Signed vs. unsigned bit extension 符号扩展
 - `lb, lh`
 - `lbu, lhu`
- Signed vs. unsigned comparison 比较
 - `slt, slti`
 - `sltu, sltiu`
- Signed vs. unsigned branch 比较
 - `blt, bge`
 - `bltu, bgeu`

条件跳转指令

- C语言中有控制流
 - 比较语句/逻辑语句确定下一步执行的语句块
- RISC-V 汇编无法定义语句块，但是可以通过标记(Label) 的方式来定义语句块起始
 - 标记后面加一个冒号 (main:)
 - 汇编的控制流就是跳转到标记的位置
 - 在C语言中也有类似的结构，但是被认为是坏的编程风格 (C语言有goto语句，跳转到标记所在的位置)

条件跳转指令

- **Branch If Equal (beq)**
 - `beq reg1, reg2, label`
 - **If value in reg1 = value in reg2, go to label**
- **Branch If Not Equal (bne)**
 - `bne reg1, reg2, label`
 - **If value in reg1 ≠ value in reg2, go to label**
 - 注意没有依据标志位的跳转（与x86不同）

无条件跳转指令 (jal, jalr)

- jal 将某一条指令的地址放到寄存器ra
- RISC-V: 指令是4字节长度
 - 内存是按照字节编址的

0x0040061C jal newMoney

0x00400620 (add 4)

伪指令

RISC-V指令集与汇编语言概述

汇编中的伪指令

- 伪指令可以给程序员更加直观的指令，但不是直接通过硬件来实现
- 通过汇编器来翻译为实际的硬件指令
- 例子：

`move dst,src`

并没有实际的数据移动指令，被翻译为下面的指令

`addi dst,src,0 or add dst,src,x0`

其它的伪指令

- **Load Immediate** (li) 装入一个立即数
 - li dst, imm
 - 装入一个32位的立即数到 dst
 - 被翻译为: addi dst x0 imm
- **Load Address** (la) 装入一个地址
 - la dst, label
 - 装入由Label指定的地址到 dst
 - (思考一下如何翻译)

□ 指令手册

Pseudo	Real
nop	addi x0, x0, 0
not rd, rs	xori rd, rs, -1
beqz rs, offset	beq rs, x0, offset
bgt rs, rt, offset	blt rt, rs, offset
j offset	jal x0, offset
ret	jalr x0, x1, offset
call offset (if too big for just a jal)	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]
tail offset (if too far for a j)	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]

伪指令 vs. 硬件指令

- 硬件指令 (TAL, True Assembly Language)
 - 所有指令都是硬件可以直接执行的指令，在硬件中直接实现了
- 伪指令
 - 汇编语言程序员可以使用的指令（加上了部分硬件未真正实现的指令）
 - 每一条伪指令指令会被翻译为1条或者多条TAL指令
- 硬件指令 \subset 伪指令

函数调用

RISC-V指令集与汇编语言概述

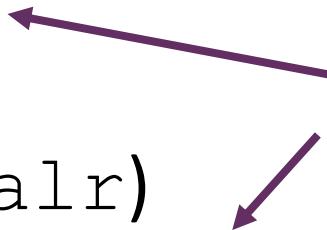
函数调用

1. 将参数放置在函数可以访问到的地方
2. 将控制流转到函数中
3. 函数获取任何其所需要的存储资源
4. 执行函数体，完成功能
5. 函数放置返回值，清理函数调用信息
6. 控制流返回给函数调用者

对函数调用的支持

- 如果有可能，尽可能使用寄存器，寄存器要比内存快得多
- $x_{10}-x_{17}$: 可以用来传递参数或返回值
- x_1 : 返回地址寄存器，用于返回到起始点
- 传递参数的时候，顺序是有用的，代表了程序中的参数的顺序
- 如果寄存器空间不够，则需要借助于在内存中的栈

RISC-V 中的函数调用

- **Jump and Link (jal)**
 - jal label
 - **Jump and Link Register (jalr)**
 - jalr src
 - **“and Link”:** 在调到对应函数内部之前，将下一条指令的地址放置在寄存器 x_1 中
 - x_1 : **ra** 返回地址寄存器
- 
- 用来调用一个
函数

RISC-V 中的寄存器

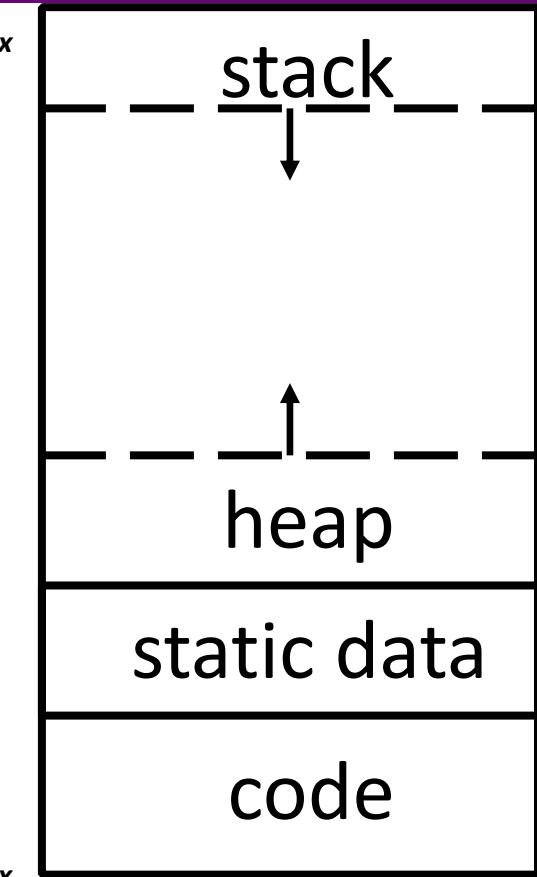
- $x_{10}-x_{17}$: 用以传递参数和返回值
- x_1 : ra 返回地址寄存器
- x_2 : sp 栈指针

指令地址

- 指令和数据都存放在同一个地址空间中
- 标记Label会被翻译为一个指令地址
- jal指令会把一条指令的地址放在寄存器 ra

$\sim FFFF\ FFFF_{hex}$

$\sim 0_{hex}$



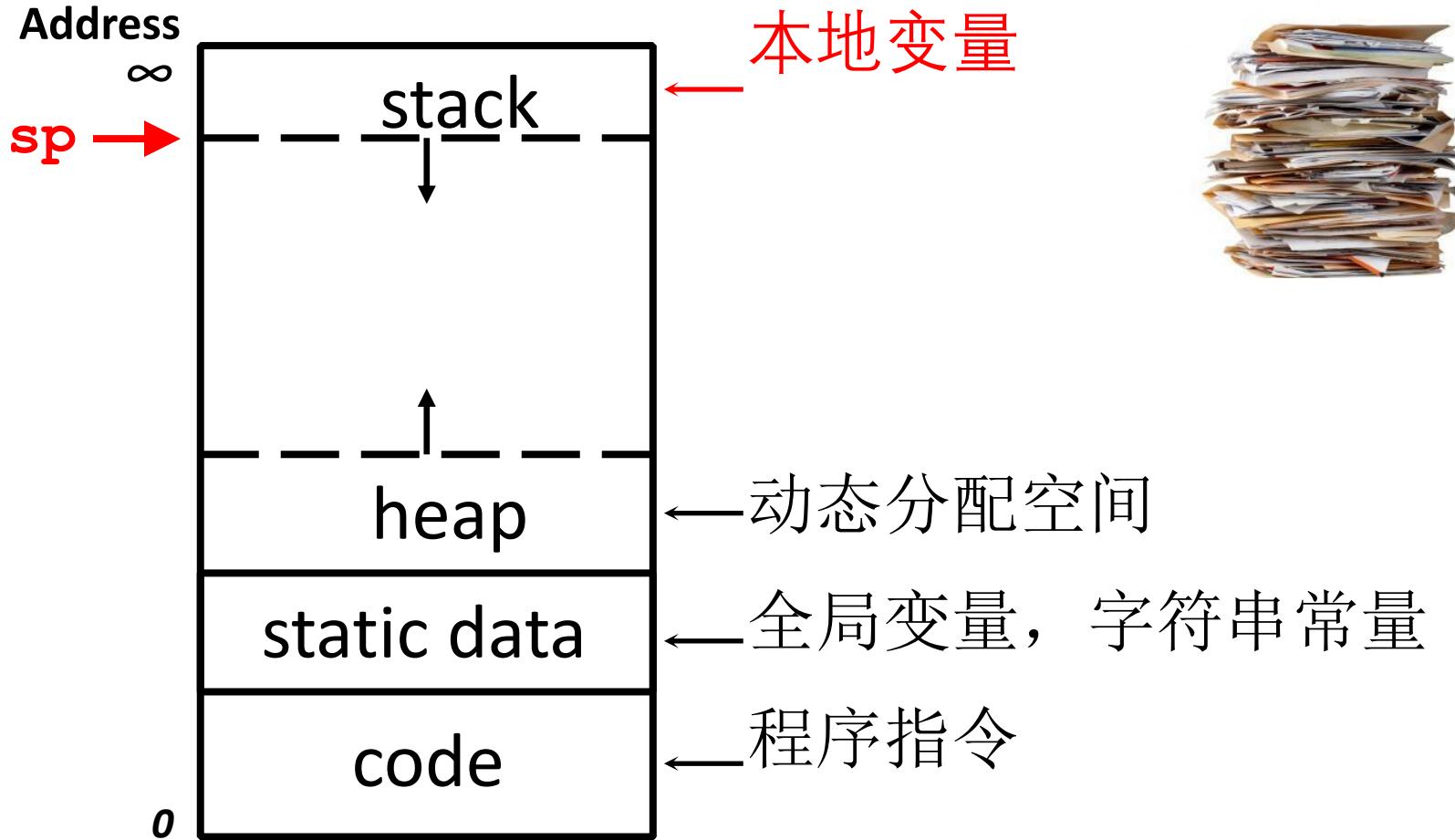
程序计数器

- 程序计数器（PC）指向的是当前正在执行的指令
(上下文不同的环境下，有时候也会说明为指向下一条指令，PC经过更新后指向下一条将执行的指令)
- 值在指令执行第一个阶段就会被更新
- PC值对于程序员是不可见的，但是可以被jal指令访问到
- 所有的分支指令(beq, bne, jal, jalr)，跳转指令都是通过更新PC来完成功能

寄存器惯例

- **CalleR**: 调用者函数
- **CalleE**: 被调用函数
- **寄存器使用惯例**: 寄存器约定的方案，调用者保存的寄存器在函数调用前后可能会被改变；被调用者保存的寄存器在调用前后不会被改变（jal）

数据的内存排布情况



被调用者保存寄存器 (Callee Saved Registers)

- `s0-s11`: `x8-x9 + x18-x27` (*callee saved registers*)
- `sp` (*stack pointer*)
 - 必须要指向相同的位置，否则调用者就找不到当前的栈帧
- 如果寄存器不够用，则可以将原值保存在栈上，待函数返回的时候恢复

调用者保存寄存器 (caller saved registers, Volatile Registers)

□ 被调用的函数可以自由使用

- 调用者如果需要使用这些值的话，调用者必须要自己去保存
- $x5-x7 + x28-x31$: $t0-t6$ (*temporary registers*, 临时寄存器)
- $x10-x11$: $a0-a1$ (*return values*, 返回值)
 - 保存需要传回来的返回值
- $x1$: ra (*return address*)

寄存器调用惯例

Register	ABI Name	Description	Saved By Callee?
x0	zero	Always Zero	N/A
x1	ra	Return Address	No
x2	sp	Stack Pointer	Yes
x3	gp	Global Pointer	N/A
x4	tp	Thread Pointer	N/A
x5-x7	t0-2	Temporary	No
x8	s0/fp	Saved Register/Frame Pointer	Yes
x9	s1	Saved Register	Yes
x10-x17	a0-7	Function Arguments/Return Values	No
x18-27	s2-11	Saved Registers	Yes
x28-31	t3-6	Temporaries	No

栈帧结构

□ Prologue

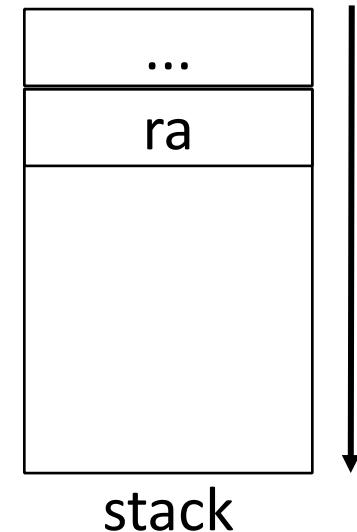
- func_label:
- addi sp, sp, -framesize
- sw ra, <framesize-4>(sp)
- save other regs if needed

□ Body (call other functions...)

-

□ Epilogue

- restore other regs if needed
- lw ra, <framesize-4>(sp)
- addi sp, sp, framesize
- jalr x0, 0(ra)



小结

- 计算机理解对应ISA中的指令
- RISC的设计原则：更小更快，保持简单
- 指令类型
 - 算术指令、逻辑指令、移位指令
 - 数据传输指令（访存指令）
 - 比较指令、有条件跳转指令、无条件跳转指令
- 函数调用之间通过调用惯例来指导参数的放置以及寄存器的使用
 - 调用者（caller）和被调用者（callee）都有自己的可直接使用寄存器和需要保存的寄存器
 - 寄存器被分类为被保存的寄存器和易失的寄存器

阅读和思考

□ 阅读

□ 思考

- 计算机指令系统中哪些是必备指令?为什么?
- 指令寻址方式有哪些?这些寻址方式可以在高级语言程序中找到哪些影子?
- 分析ThinPAD RISCV指令系统的在寻址方式和指令格式方面的特点
- 根据ThinPAD RISCV指令系统要求, 确定ALU应具备的功能

谢谢

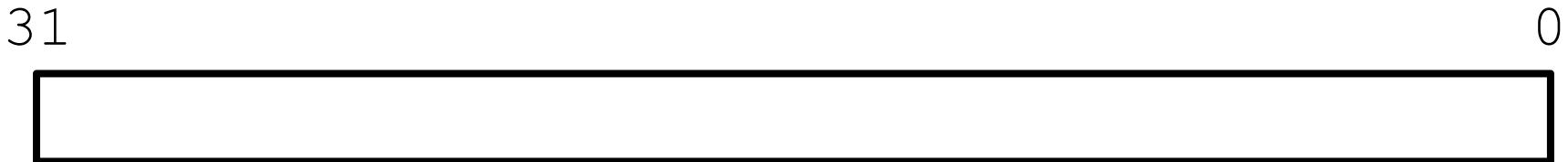


指令格式与数据通路设计

2022年秋

RISC-V 的指令格式

- 每一条指令是32位，4个字节（1个字）



- 指令格式将32位分为不同的“域”，就是不同的部分
- 定义了六种指令的格式
- R-格式 I-格式 S-格式 B格式 U格式 J-格式

RISC-V 的指令格式 (I)

- R-格式：寄存器指令，指定指令中的3个寄存器
- I-格式：指令中包含立即数，用于带一个常数的算术指令以及加载指令
- S-格式：store指令

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
		funct7		rs2		rs1	funct3		rd		opcode	R-type
		imm[11:0]			rs1	funct3		rd		opcode		I-type
		imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode		S-type
		imm[12]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
		imm[31:12]						rd		opcode		U-type
		imm[20]	imm[10:1]	imm[11]	imm[19:12]			rd		opcode		J-type

RISC-V 的指令格式 (II)

□ B-格式：分支指令

□ U-格式：大立即数

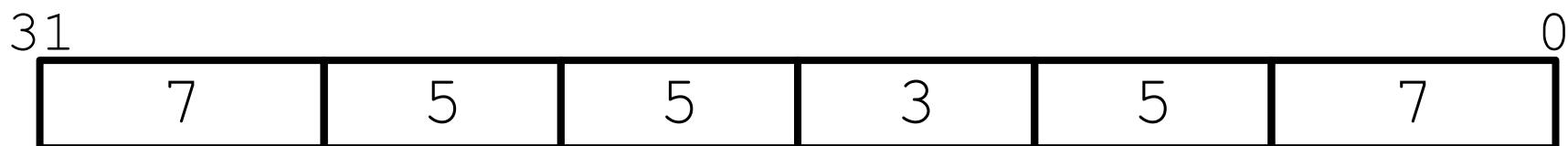
- 两个指令lui (load upper imm), auipc (add upper imm to PC)

□ J-格式：唯一指令jal

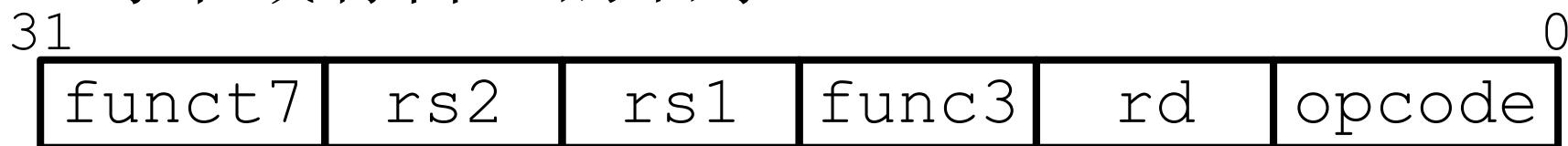
31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
	imm[11:0]			rs1	funct3		rd		opcode			I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode				S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode				B-type
	imm[31:12]					rd		opcode				U-type
imm[20]	imm[10:1]	imm[11]	imm[19:12]			rd		opcode				J-type

R类型指令

- R型指令每个指令字分成6个域:

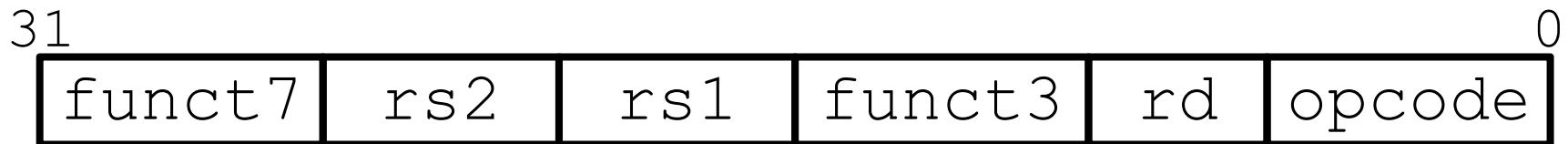


- 每个域有自己的名字:



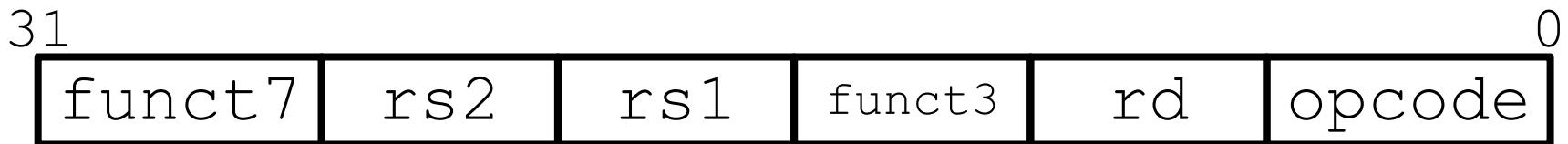
- 每个域有不同含义

R类型指令



- **opcode (7)**: 操作码，但未指明确切操作
 - R类型指令操作码都为 0110011
- **funct3 (3), funct7 (7)**: 与操作码 opcode 一起指定了指令的具体功能
- 计算一下能够编码多少R类型指令功能?
 - opcode 是固定的, 依据 $\text{funct: } 2^{10} = 1024$

R类型指令



- **rs1** (5): 第一个操作数，“source register”)
- **rs2** (5): 第二个操作数，“second source register”)
- **rd** (5): 目标寄存器，“destination register”)
- 每一个寄存器5位进行编码，可以指定32个寄存器，编码的是寄存器编号

R类型指令举例

add x18 x19 x10

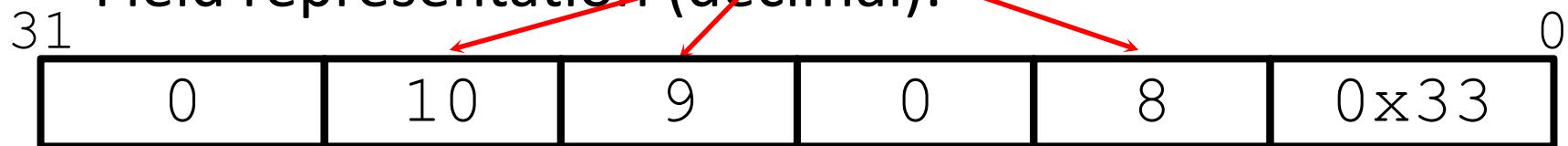
funct7	rs2	rs1	funct3	rd	opcode
0000000			000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	001	rd	0110011
0000000	rs2	rs1	010	rd	0110011
0000000	rs2	rs1	011	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	101	rd	0110011
0100000	rs2	rs1	101	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011

31	10	19	0	18	0
funct7	rs2	rs1	funct3	rd	opcode

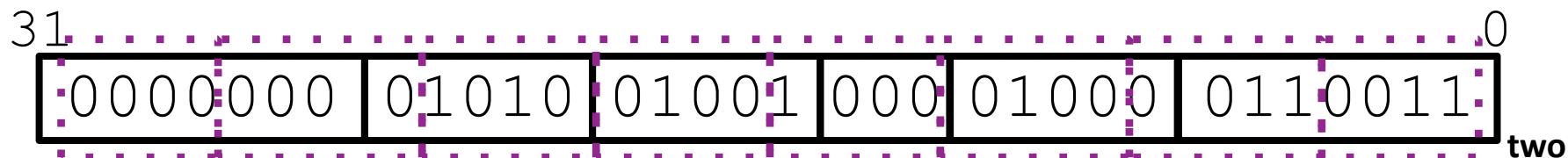
R类型机器码

- Instruction: add x8, x9, x10

~~Field representation (decimal):~~



~~Field representation (binary):~~



hex representation: 0x 00A4 8433

decimal representation: 10,781,747

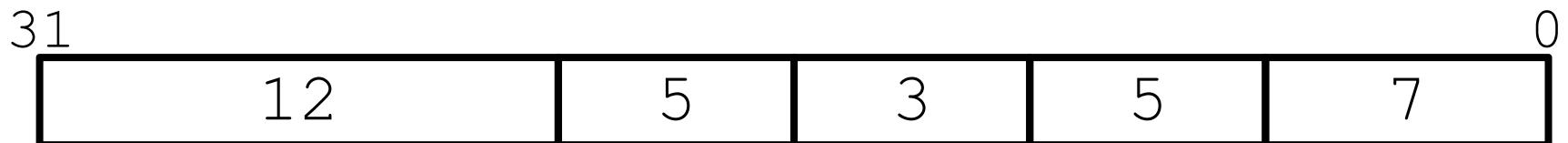
机器码

I类型指令

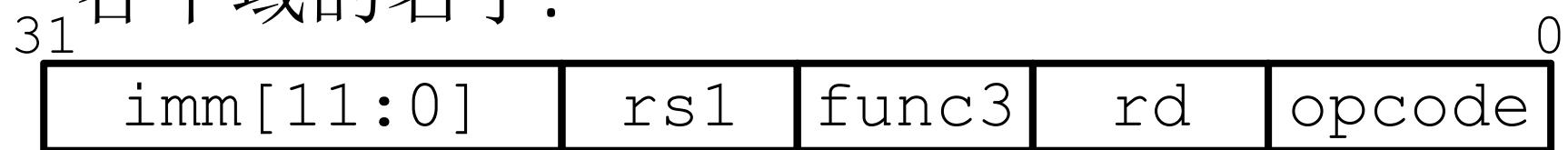
- 对于含有立即数的指令来说，5位的域能够表达的范围太小
- 理想来说，RISC-V 指令最好只有一种格式。但是实际情况下需要折衷
- 在实际中可以依据类似于R指令格式的方式定义新的指令格式
 - 如果一条指令使用了立即数，这条指令最多使用两个寄存器

I类型指令

- 下面是I类型指令的格式：

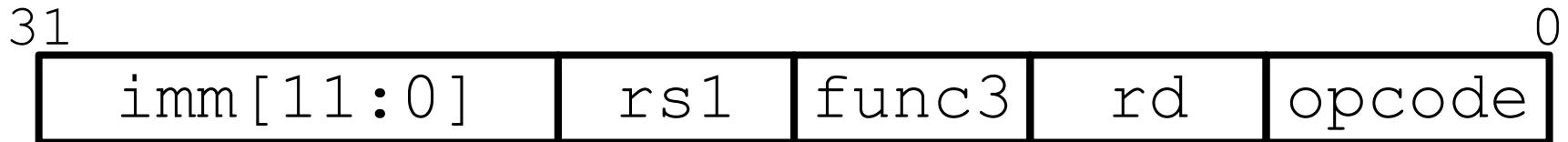


- 各个域的名字：



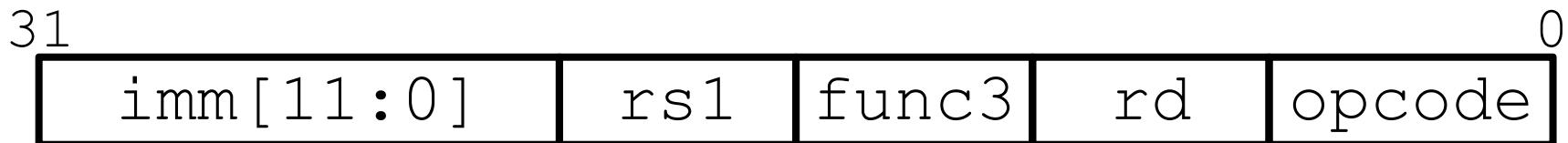
- 四个域与R类型指令是一样的
 - opcode 仍然放置在原来的位置

I类型指令



- **opcode** (7): 指定操作类型
- **rs1** (5): 指定一个寄存器操作数
- **rd** (5): 指定目标寄存器 (“destination register”)

|类型指令

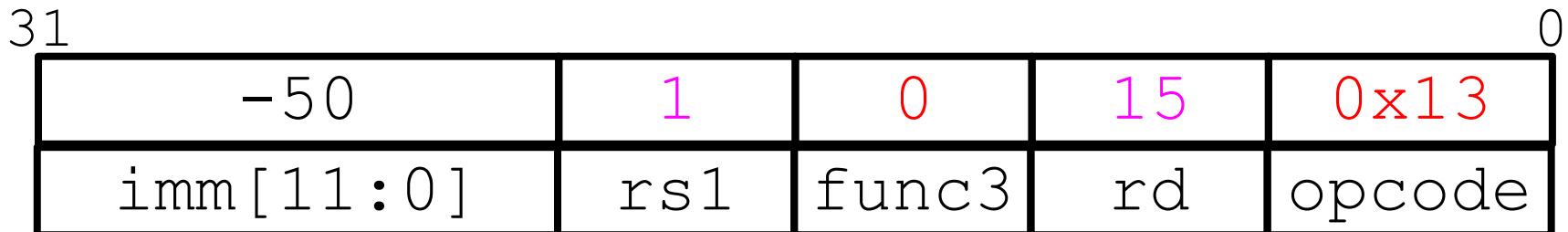


- **immediate (12)**: 12位立即数
 - 所有的计算是用字进行计算，即32位，必须要对立即数进行扩展
 - 采用符号扩展方式
- 12位表示范围 [-2048, +2047]
 - 如果立即数超过12位，怎么办？

I类型举例

addi x15, x1, -50

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	SLLI
0000000	shamt	rs1	101	rd	SRLI
0100000	shamt	rs1	101	rd	SRAI



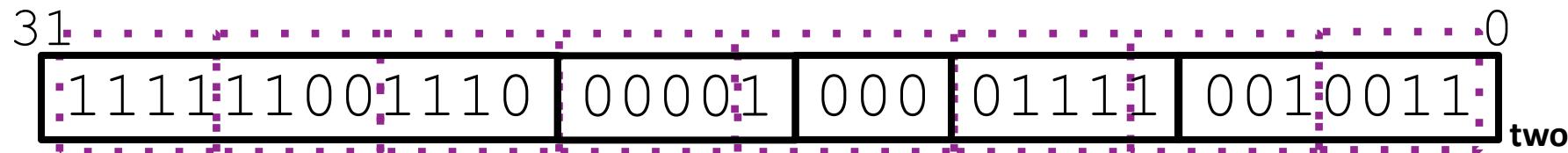
I类型举例

- 指令: addi x15, x1, -50

各个域的十进制表示:



各个域的二进制表示:



十六进制表示: 0xFCE08793

十进制表示: 4,242,573,203

I 类型举例 - load

□ 指令: lw x14, 8(x2)

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12 offset[11:0]	5 base	3 width	5 dest	7 LOAD	

□ **rd** (5): 结果放置的寄存器

□ **rs1** (5): 基地址寄存器

□ **immediate** (12)

■ 基地址寄存器的值 + 立即数 → load内存地址

I 类型举例 - load

口 指令: lw x14, 8(x2)

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
offset[11:0]	base	width	dest	7 LOAD	

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

000000001000	00010	010	01110	0000011
--------------	-------	-----	-------	---------

imm=+8

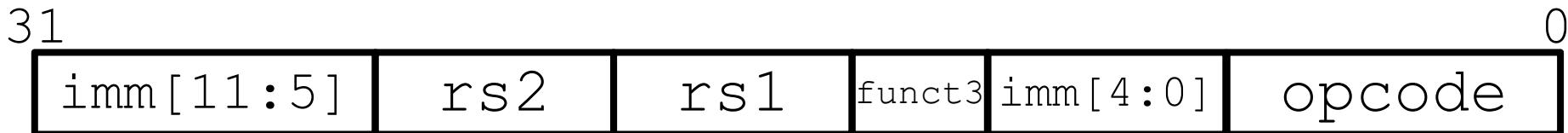
rs1=2

LW

rd=14

LOAD

S 类型指令



□ Store指令

- **rs1 (5)**: 基地址寄存器
- **rs2 (5)**: 数据寄存器
- **immediate (11:5) + immediate (4:0)** : 地址偏移

□ 为什么不把rs2用作imm的低位?

- RISC-V设计让rs1和rs2的位置保持固定

RISC-V 设计策略：让寄存器位置保持不变

- 对于所有的操作来说，关键路径包括获得寄存器的值
- 把读的寄存器都放到相同的位置，那么每次都可以立即去读寄存器文件，不需要做判断
 - 如果读出来的数据其实是不必要的（例如I类型），后续丢弃即可
- 其它的RISC体系结构的设计有些许的不同
 - 在译码阶段需要通过指令来判断需要读哪些寄存器
- 寄存器位置保持不变是RISC-V的体系结构上一系列微小的调整之一，可以让整个处理器工作流程更好

S 类型举例

口指令： sw x14 , 8 (x2)

31

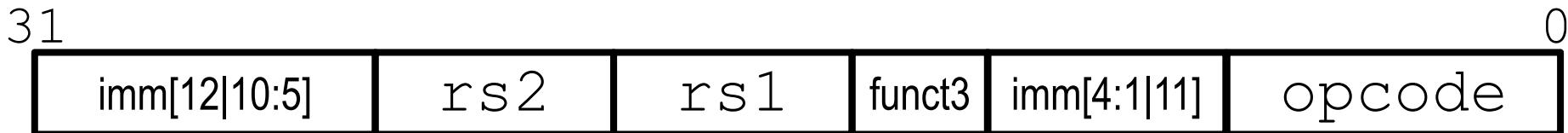
0

0	14	2	2	8	sw
imm [11:5]	rs2	rs1	funct3	imm [4:0]	opcode

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

0000000	01110	00010	010	01000	0100011	two
---------	-------	-------	-----	-------	---------	-----

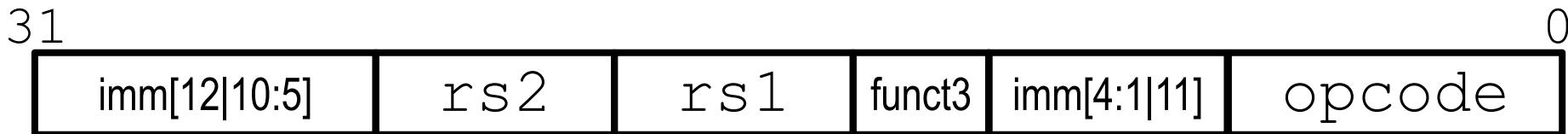
B 类型指令



口 分支指令

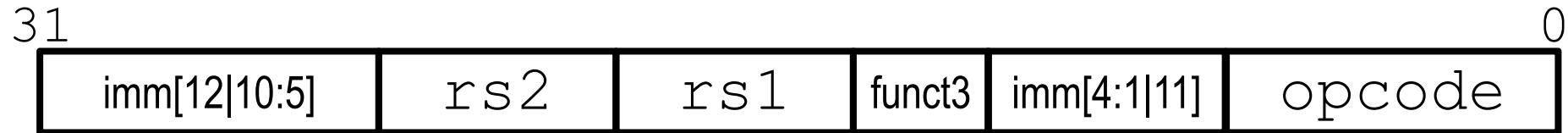
分支指令

- `beq` and `bne`
 - 需要指定目标地址
 - 也需要两个寄存器间比较



- `opcode` 指定 是`beq (4)` 还是 `bne (5)`
 - `rs1`, `rs2` 用来指定两个寄存器
 - 使用 `immediate`指定地址? (PC相对跳转, 循环的跳转一般在一个比较小的范围)

PC相对跳转寻址



- **PC-Relative Addressing:** 使用立即数作为补码，用作在PC上的偏移
- 类似于 lw/sw 的基址偏移寻址，源寄存器 rs1 为PC
- 这样可以指定从PC开始的 $\pm 2^{11}$ 的地址范围

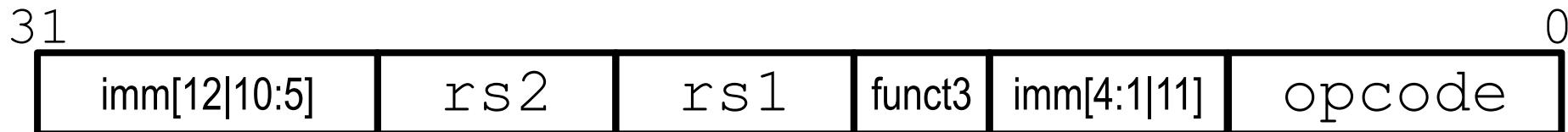
实际地址范围的计算

- 实际上 RISC-V 使用的是固定长度的指令字长，内存是按字节地址寻址的
- 指令是按照字对齐的，指令的地址总是2的倍数（末1位为0）（与Compact的指令格式兼容，需要最后一位为0即可，但是在RV32I中实际是4字节对齐的）
- PC总是指向一个指令地址，就可以做类型指针的操作
- 在指令地址范围计算的时候，可以指定 $\pm 2^{12}$ 范围的地址

分支的计算

- 不需要分支
 - $PC = PC + 4$
- 需要进行分支
 - $PC = PC + immediate$
- **Observations:**
 - immediate 是跳转字节数 (注意imm[12:1]需要加上最末尾隐含的0位)
 - 为向前 (+) , 或者向后 (-) 的偏移地址 (范围为 $\pm 4KB$)

B 类型指令



□ 与 S 类型指令非常相似

- 两个寄存器 (rs1, rs2) , 一个立即数 (12位)

□ Imm 表示了 [-4096, +4094] 范围的 2字节对齐 的偏移

□ Immediate 用 12比特 表示了 13比特的范围

- 因为2字节对齐，所以最低位总是为0，因而省略了
- B类型中imm[12:1] 与 S类型中imm[11:0] 不同

分支举例

- RISC-V Code:

```
Loop: beq x9, x0, End
      add x8, x8, x10
      addi x9, x9, -1
      j Loop
End:  <some instr>
```

A curly brace on the right side of the code groups the four instructions following the branch target: add, addi, j, and the start of the End block. To the right of the brace, the numbers 1, 2, 3, and 4 are listed vertically, with an arrow pointing from the bottom number 4 back to the brace, indicating they all count as instructions from the branch.

1 Count
2 instructions
3 from branch
4

- B-Format fields:

opcode = 0x67

rs1 = 9 (first operand)

rs2 = 0 (second operand)

immediate = ??? **4*4=16**

分支举例

- RISC-V Code:

Loop: **beq x9,x0,End**

add x8,x8,x10

addi x9,x9,-1

j Loop

End: <some instr>

imm = 16

0 0000 0001 0000

31 0					
		0	9	beq	BRANCH
imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode
00000000	00000	01001	000	10000	1100011

two

所有B型指令

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

PC相对寻址的特性

- PC相对寻址的情况下，如果整块代码进行移动，则代码中的相对寻址值无需更改，如果只移动其中的几行代码，则相对寻址值可能需要更改
- 如果需要移动的范围超过表达的指令范围 $> 2^{10}$
 - 可以使用其它的指令
 - ```
beq x10, x0, far
next instr -->
next: # next instr
```

|                                 |     |                                |
|---------------------------------|-----|--------------------------------|
| <code>beq x10, x0, far</code>   | --> | <code>bne x10, x0, next</code> |
| <code># next instr</code>       |     | <code>j far</code>             |
| <code>next: # next instr</code> |     |                                |

# U 类型指令



- 如何处理一个字长的立即数？指令字长总共32位
- lui: Load Upper Immediate
  - lui reg, imm
  - 将20位的立即数装入到寄存器reg的高20位
  - 将低12位清零
- auipc: Add Upper Immediate to PC

# 使用lui的例子

□ lui设置了高20位，通过addi设置低12位

□ 例1：设置0x87654321

```
lui x10, 0x87654 # x10=0x87654000
addi x10, x10, 0x321 # x10=0x87654321
```

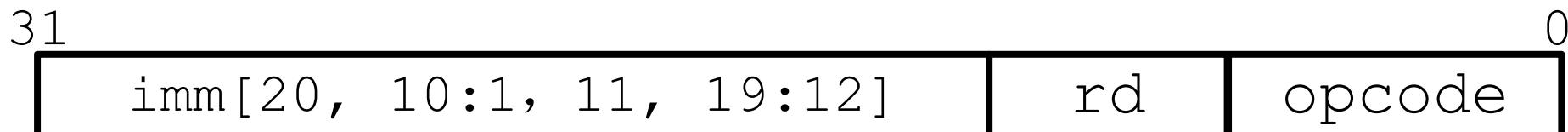
□ 例2：设置0xDEADBEEF

```
lui x10, 0xDEADB # x10=0xDEADB000
addi x10, x10, 0xEEF # x10=0xDEADAEEF
```

解决办法：lui 高20位 加1

伪指令li x10, 0xDEADBEEF：自动生成两条指令

# J 类型指令



## □ Jal 指令

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = PC + offset$

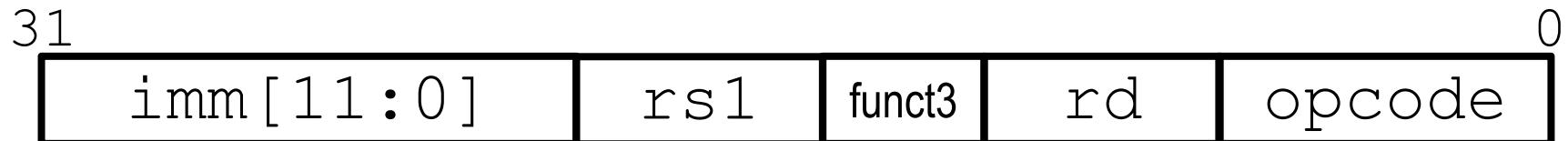
## □ 跳转范围: $[-2^{19}, +2^{19}]$ 个位置, 2字节为单位

- 实际跳转范围  $[-2^{18}, +2^{18}]$  个32-bit指令

## □ J 伪指令

- **# j pseudo-instruction**  
**j Label = jal x0, Label # Discard return address**
- **# Call function within  $2^{18}$  instructions of PC**  
**jal ra, FuncName**

# Jalr 指令（属于 I类型，不是 J类型）



## □ Jalr rd, rs, immediate

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = rs1 + offset$
- Immediate与 I类型指令中的算术和加载指令一样
  - 注意, 不需要 \* 2

# Jalr 用法例子

# ret and jr psuedo-instructions

ret = jr ra = jalr x0, 0(ra)

# Call function at any 32-bit absolute address

lui x1, <hi20bits>

jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset

auipc x1, <hi20bits> # Adds upper immediate value to  
# and places result in x1

jalr x0, x1, <lo12bits> # Same sign extension trick needed  
# as LUI

# RISC-V 指令格式总结

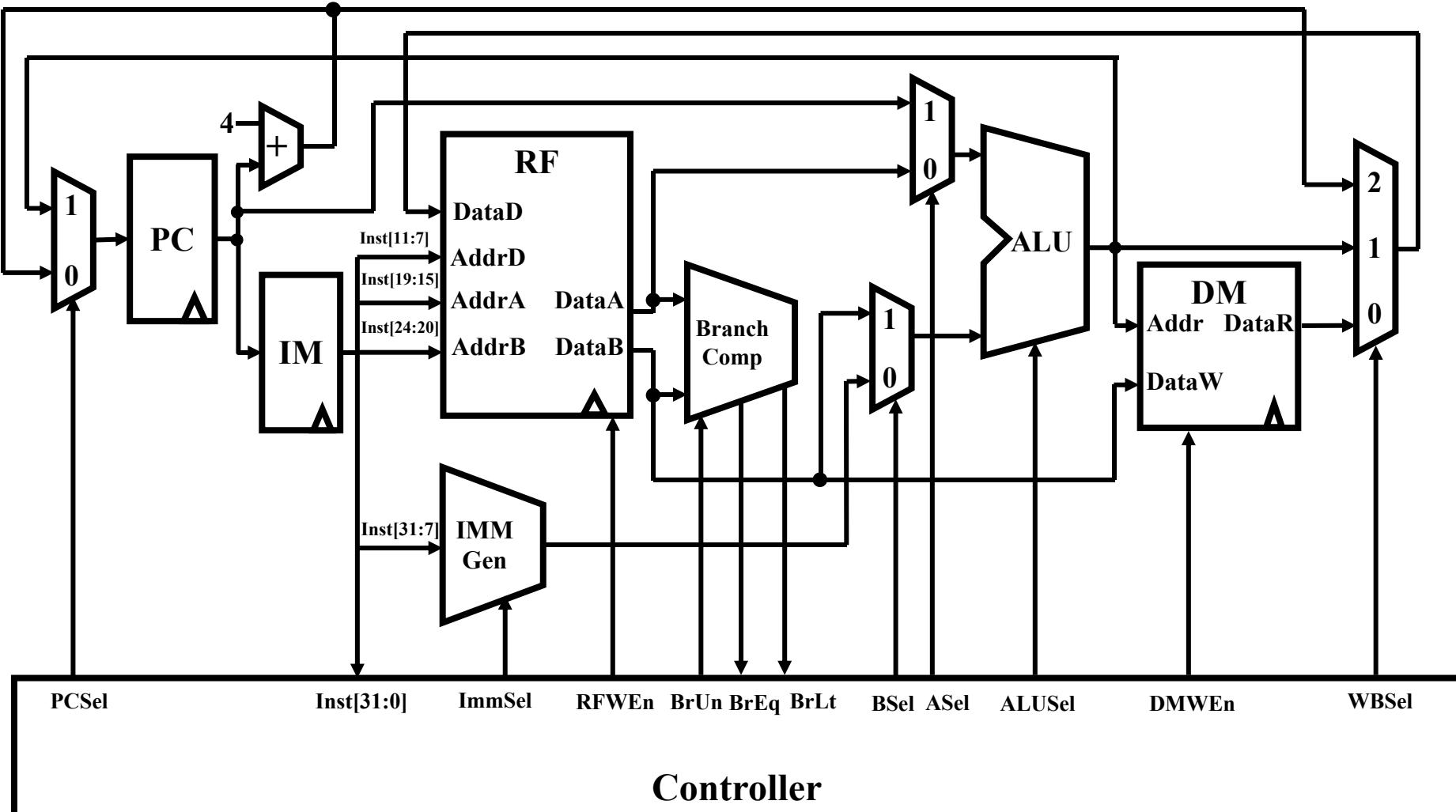
- 代码与数据一样保存在程序地址空间非常重要，硬件和软件在一定程度上同等处理
- RISC-V 机器语言指令：

| 31      | 30        | 25 24      | 21         | 20  | 19  | 15 14  | 12 11    | 8        | 7      | 6      | 0      |        |
|---------|-----------|------------|------------|-----|-----|--------|----------|----------|--------|--------|--------|--------|
|         |           | funct7     |            | rs2 |     | rs1    | funct3   |          | rd     |        | opcode | R-type |
|         |           | imm[11:0]  |            |     | rs1 | funct3 |          | rd       |        | opcode |        | I-type |
|         |           | imm[11:5]  |            | rs2 |     | rs1    | funct3   | imm[4:0] |        | opcode |        | S-type |
| imm[12] | imm[10:5] |            | rs2        |     | rs1 | funct3 | imm[4:1] | imm[11]  | opcode |        |        | B-type |
|         |           | imm[31:12] |            |     |     |        | rd       |          | opcode |        |        | U-type |
| imm[20] | imm[10:1] | imm[11]    | imm[19:12] |     |     |        | rd       |          | opcode |        |        | J-type |

---

# 控制器数据通路设计

# RISC-V RV32I 单周期数据通路



# R型指令—— add, sub, etc.

add \ sub

add rd rs1 rs2

sub rd rs1 rs2

指令功能

$R[rd] = R[rs1] + R[rs2]$

$R[rd] = R[rs1] - R[rs2]$

执行过程

取指令

分析指令

执行指令

- 取操作数

- 运算

- 结果写回

计算下一条指令的地址

31

funct7

rs2

rs1

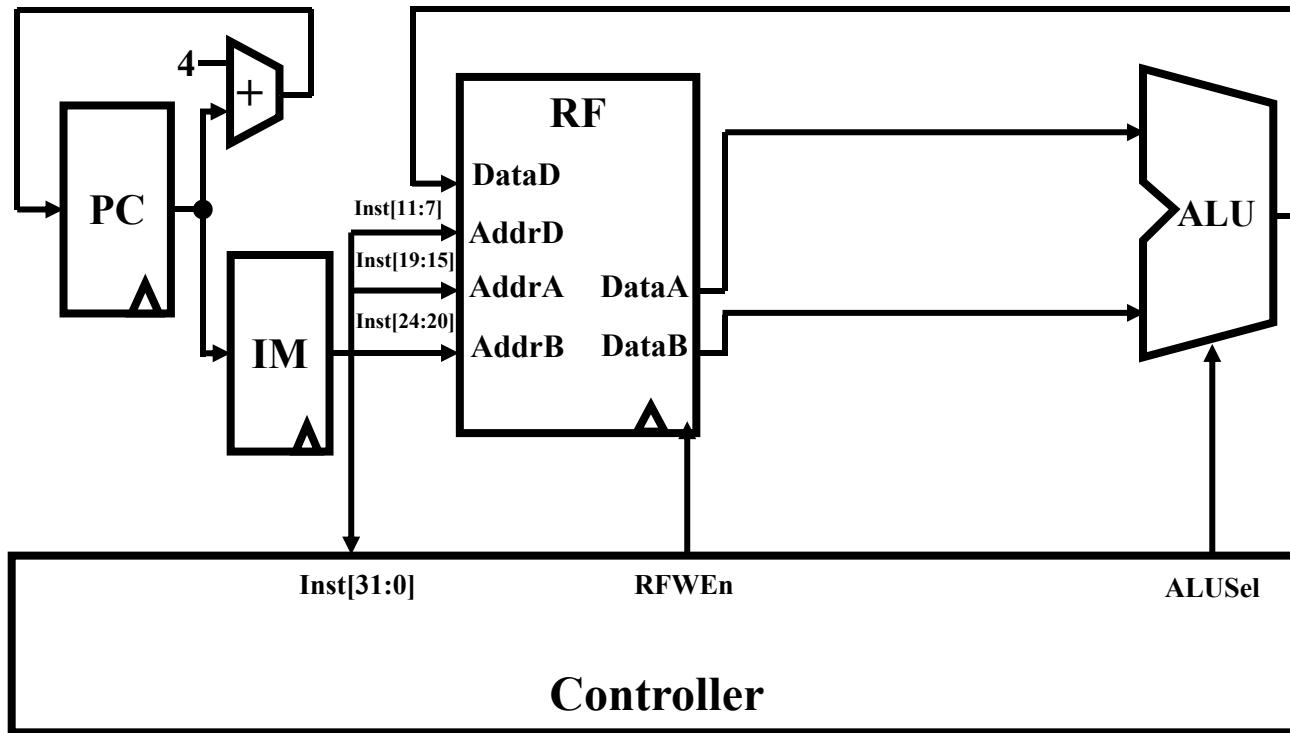
funct3

rd

opcode

0

# 数据通路设计1

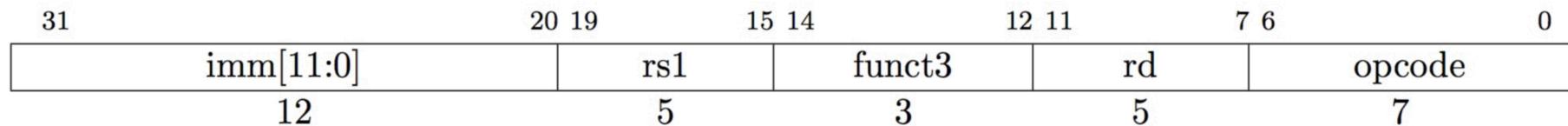


# 算术I型指令—— addi, etc.

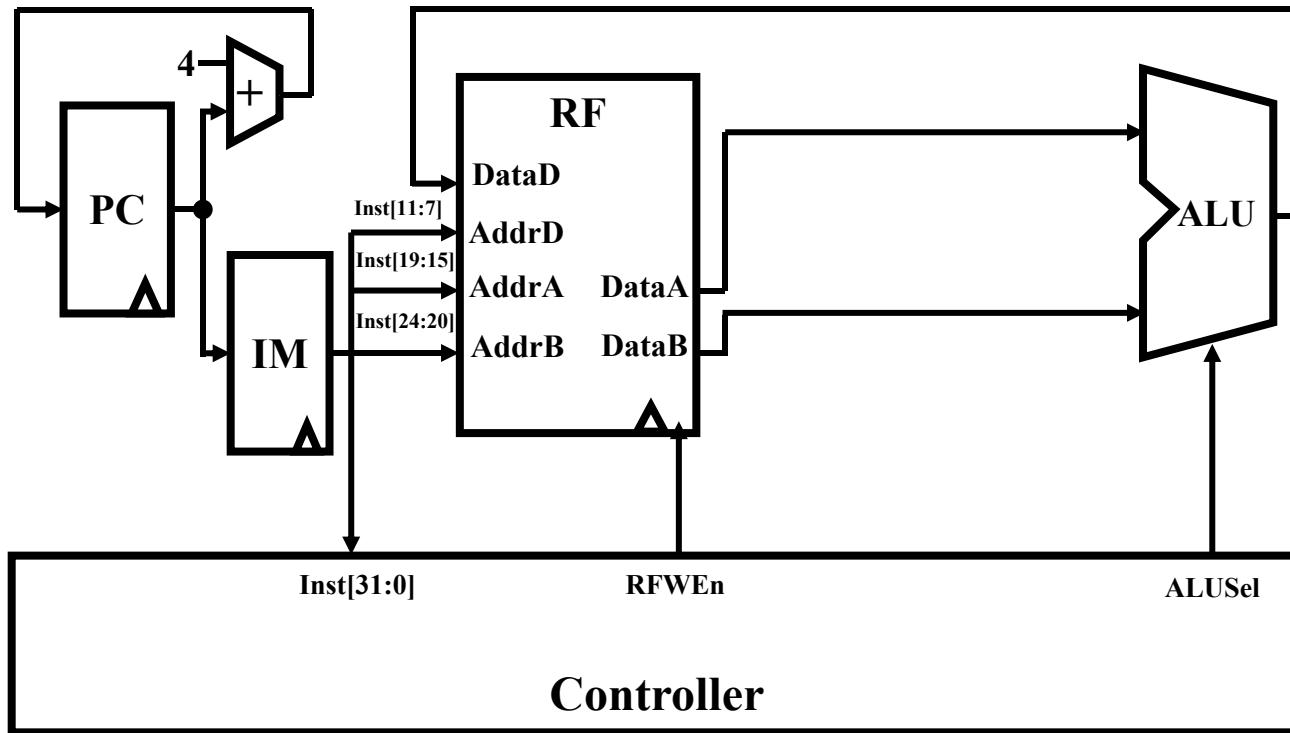
□ addi rd,rs1,imm

□ 指令功能

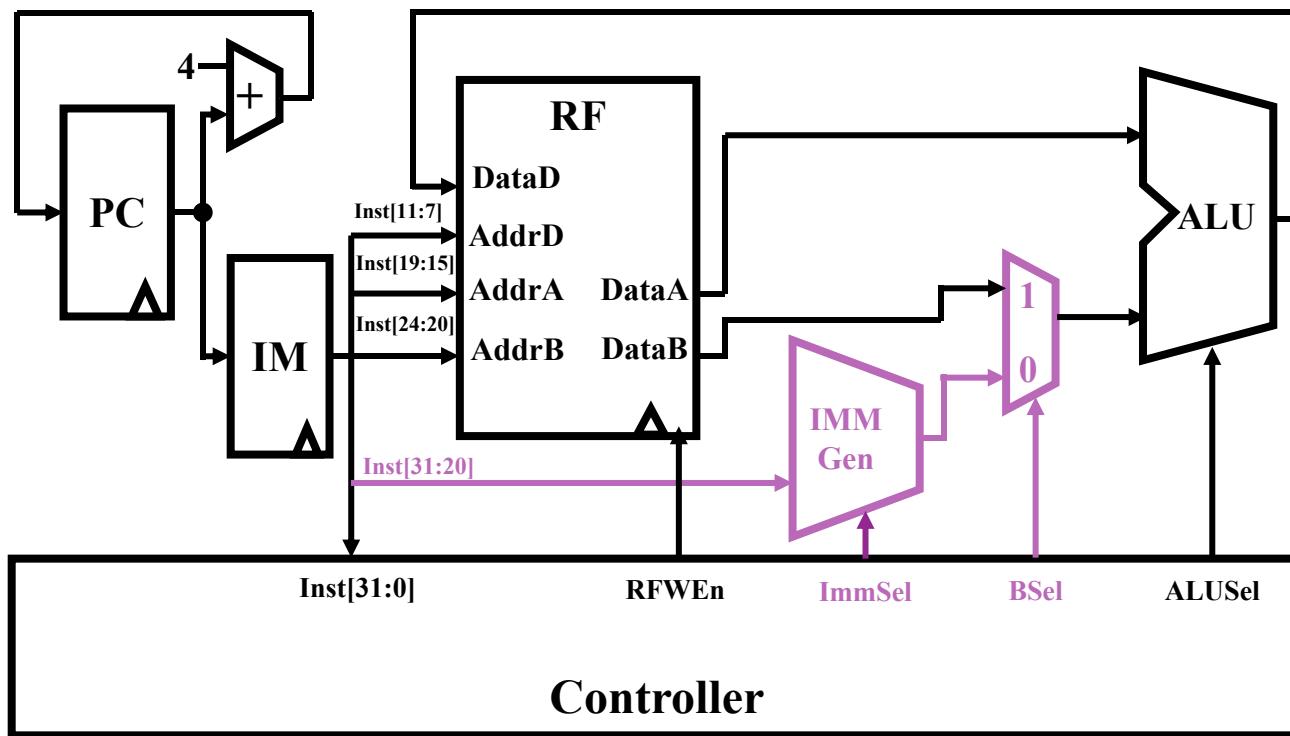
- $R[rd] = R[rs1] + \text{SignExt}(imm)$



# 数据通路设计1



# 数据通路设计2

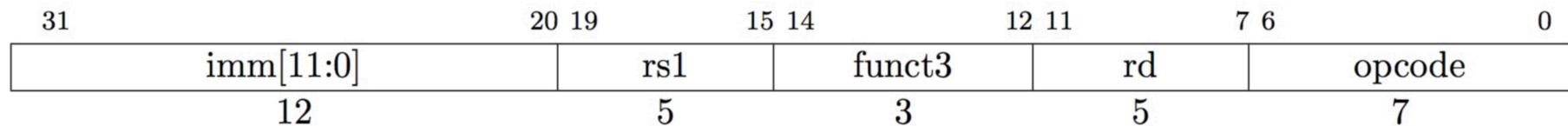


# 访存I型指令——Load系列

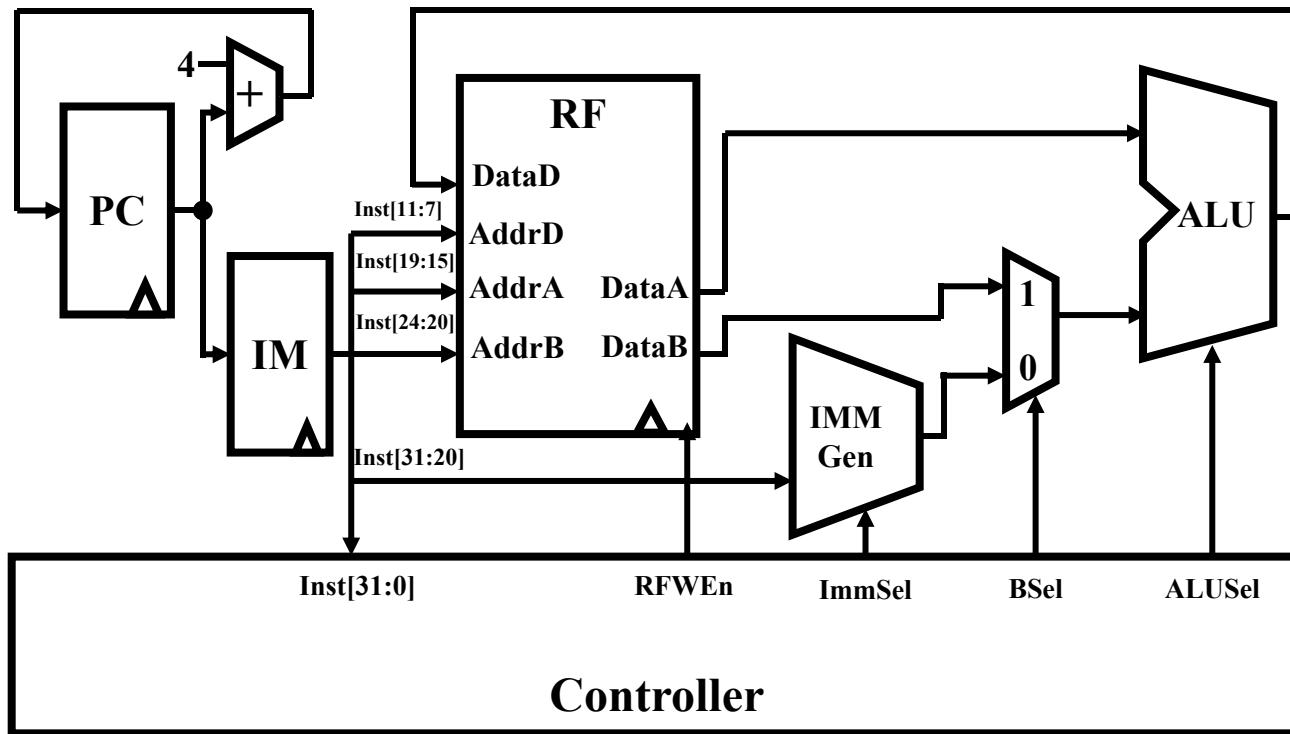
□ Load: lw rd rs1 imm

□ 指令功能

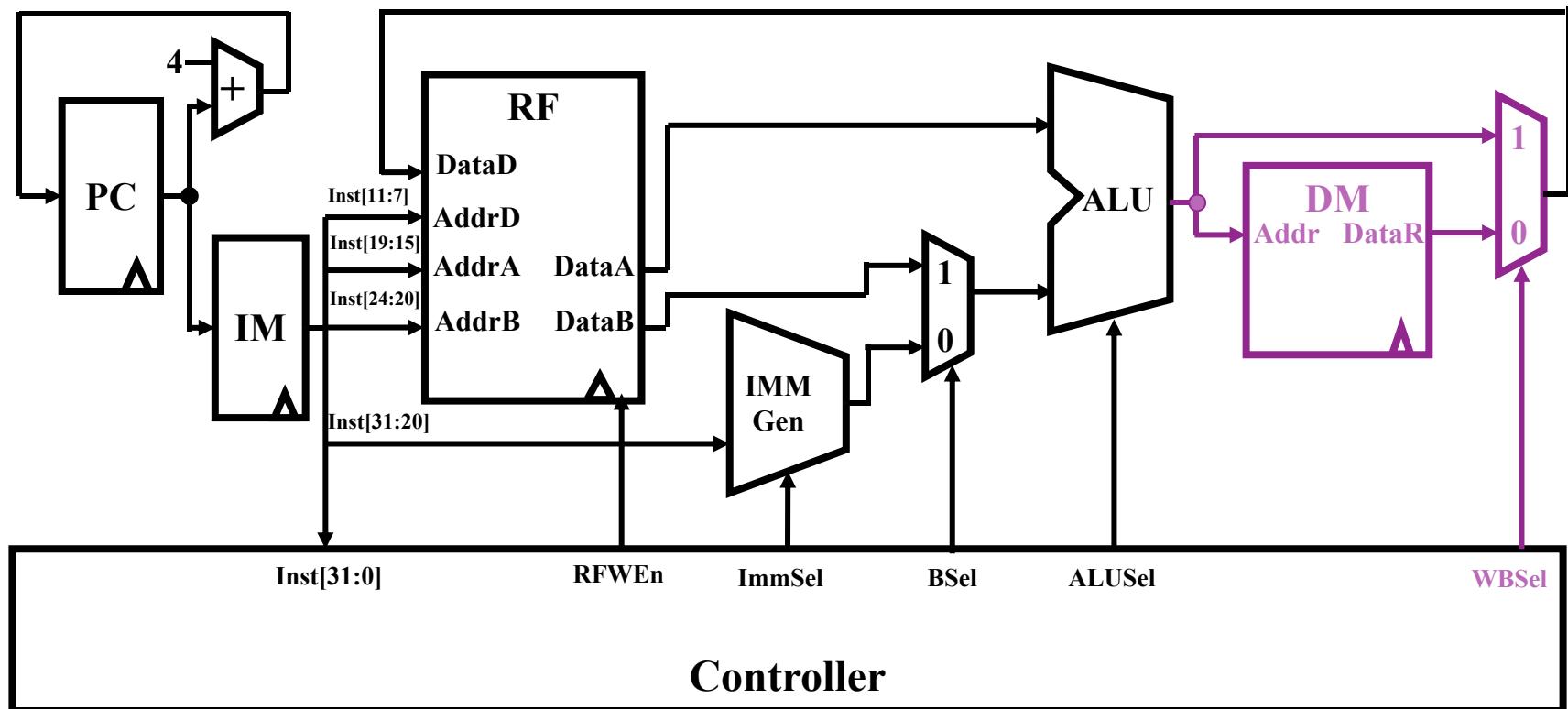
- $\text{Addr} = \text{R}[rs1] + \text{SignExt}(imm)$
- $\text{R}[rd] = \text{MEM}[\text{Addr}]$



# 数据通路设计2

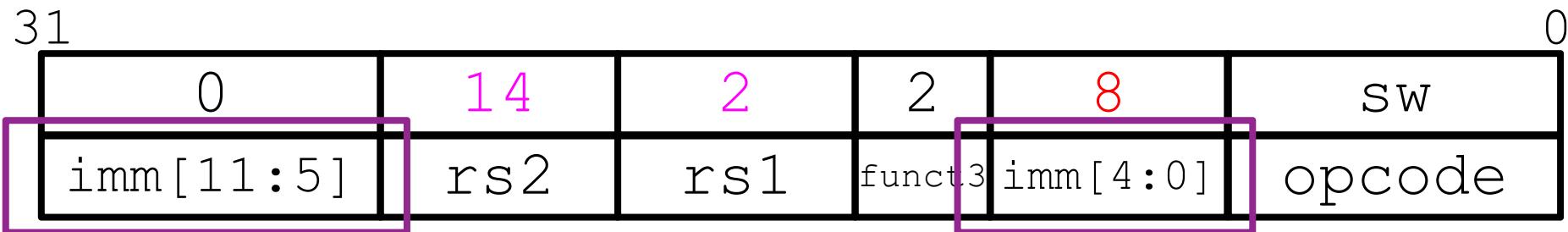


# 数据通路设计3

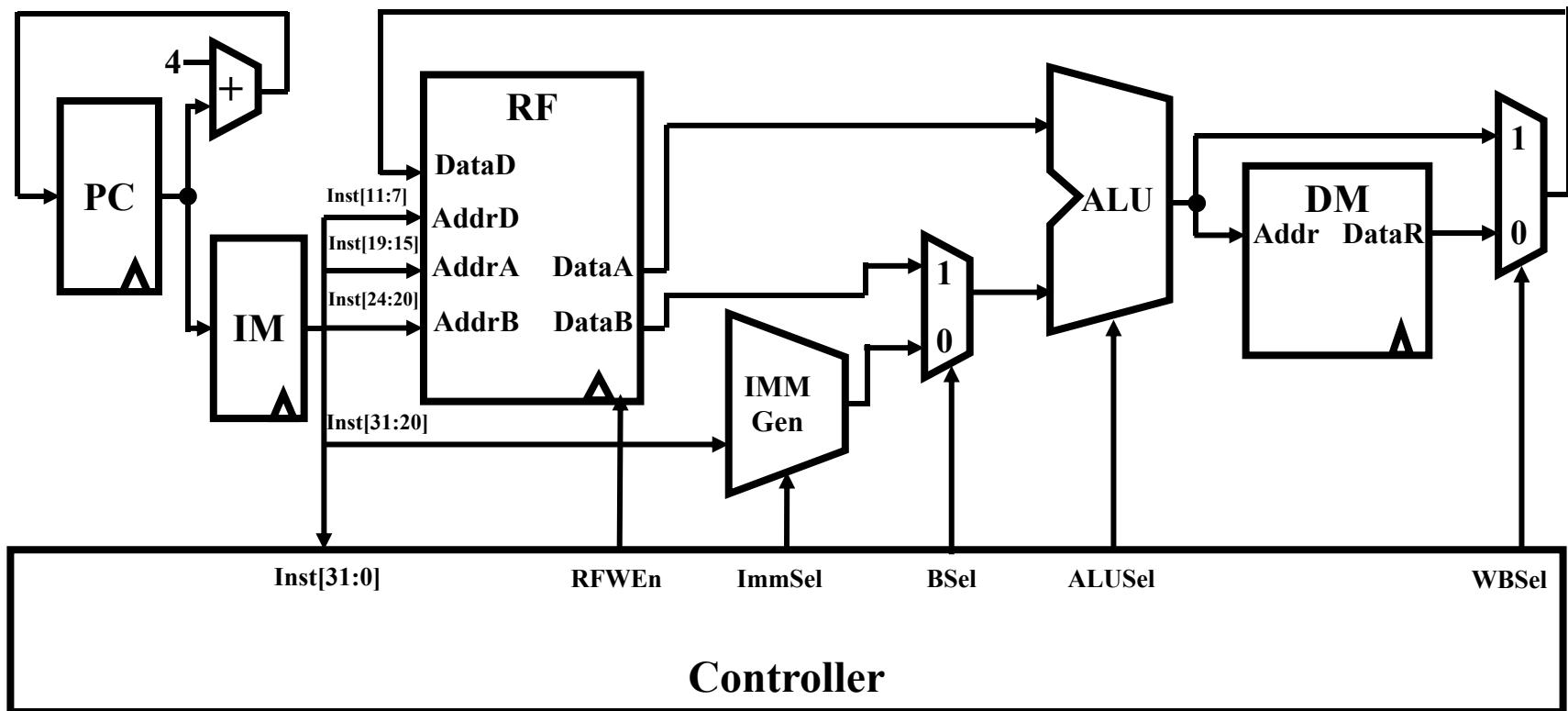


# S型指令——Store

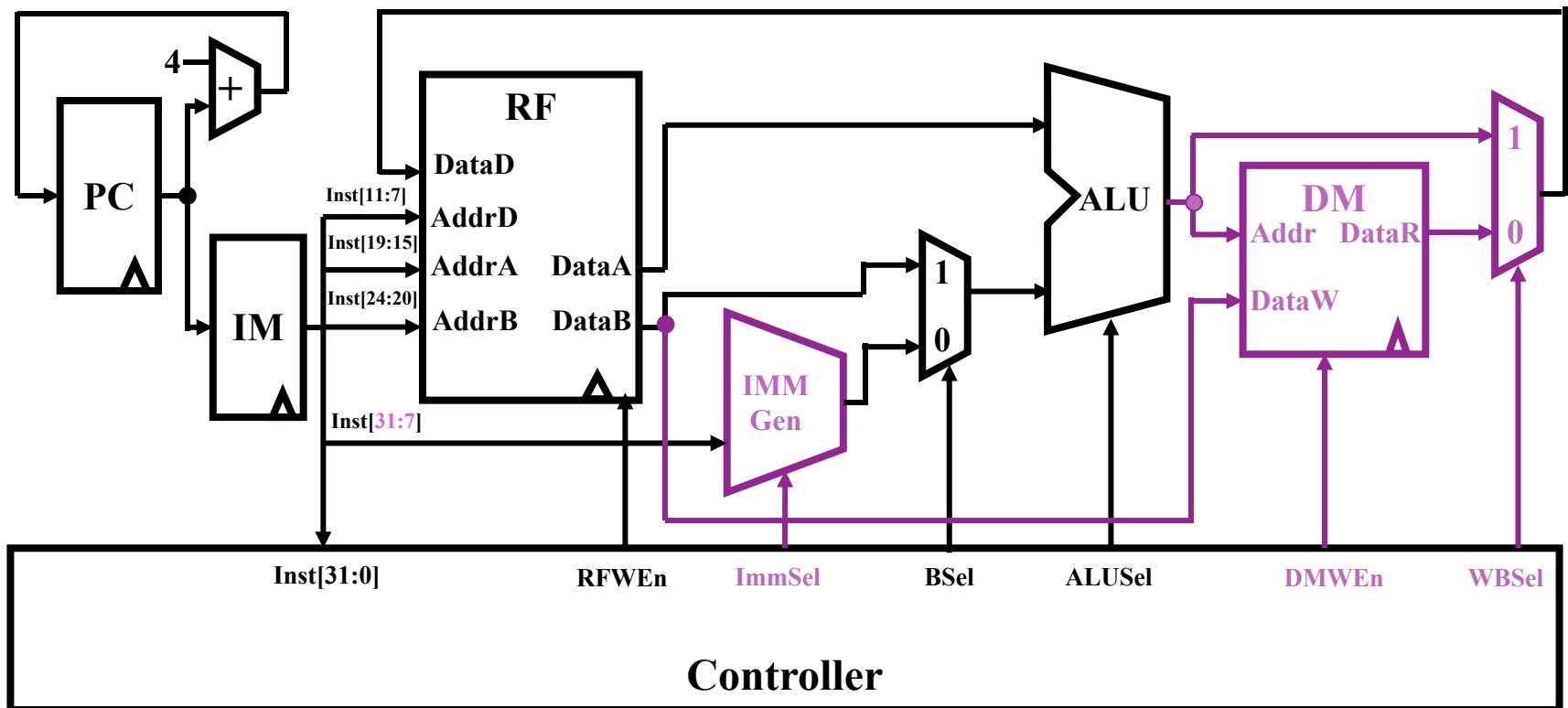
- Store: sw rs2 rs1 imm
- 指令功能
  - $\text{Addr} = \text{R}[\text{rs1}] + \text{SignExt}(\text{imm})$
  - $\text{MEM}[\text{Addr}] = \text{R}[\text{rs2}]$



# 数据通路设计3



# 数据通路设计4

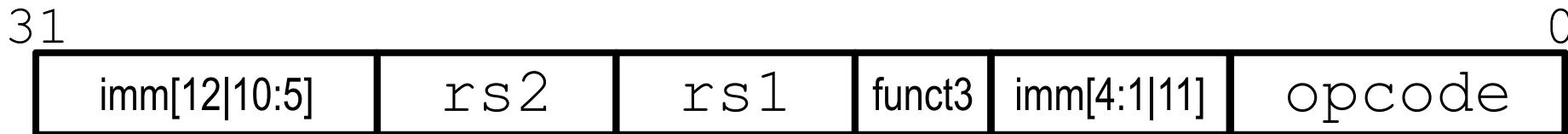


# B型指令——BEQ, etc.

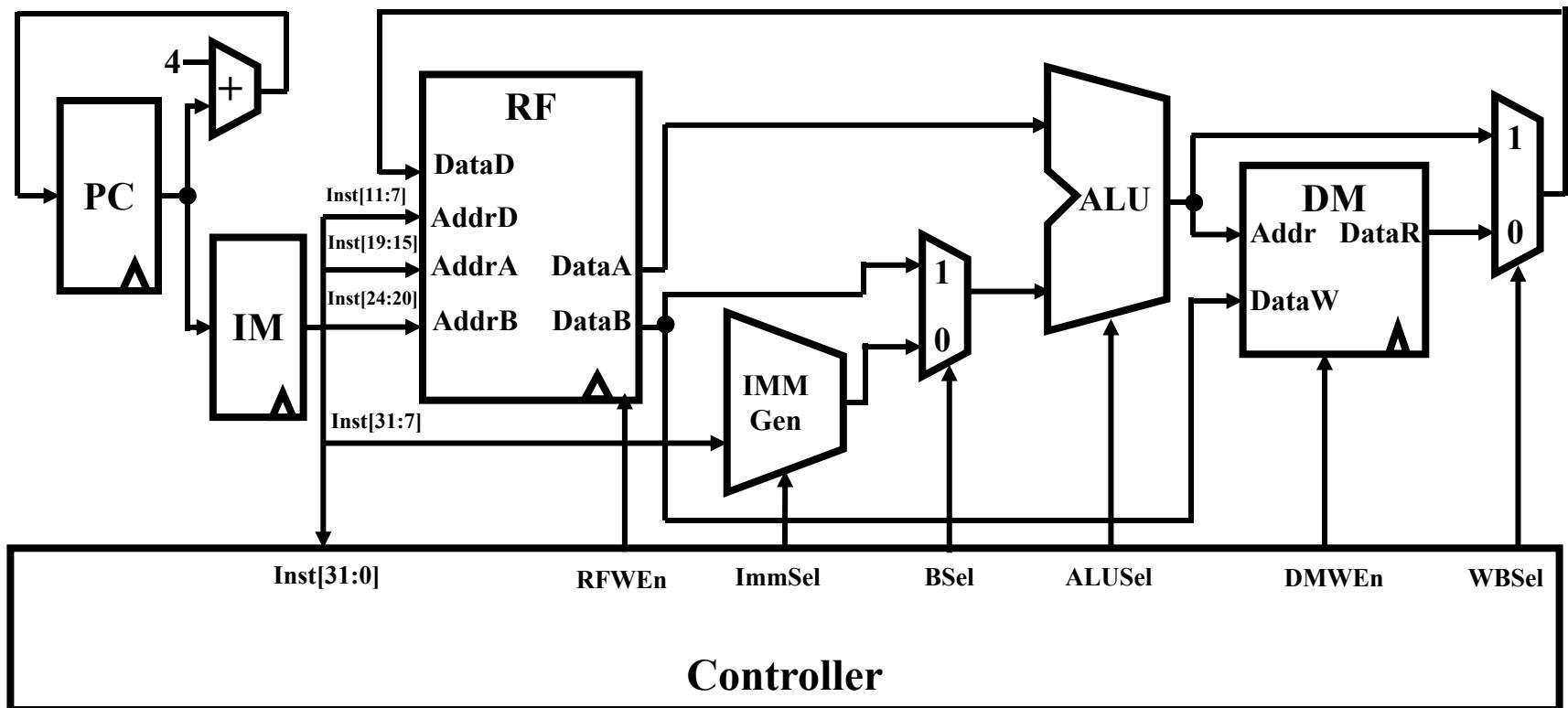
□ BEQ: beq rs1 rs2 label

□ 指令功能

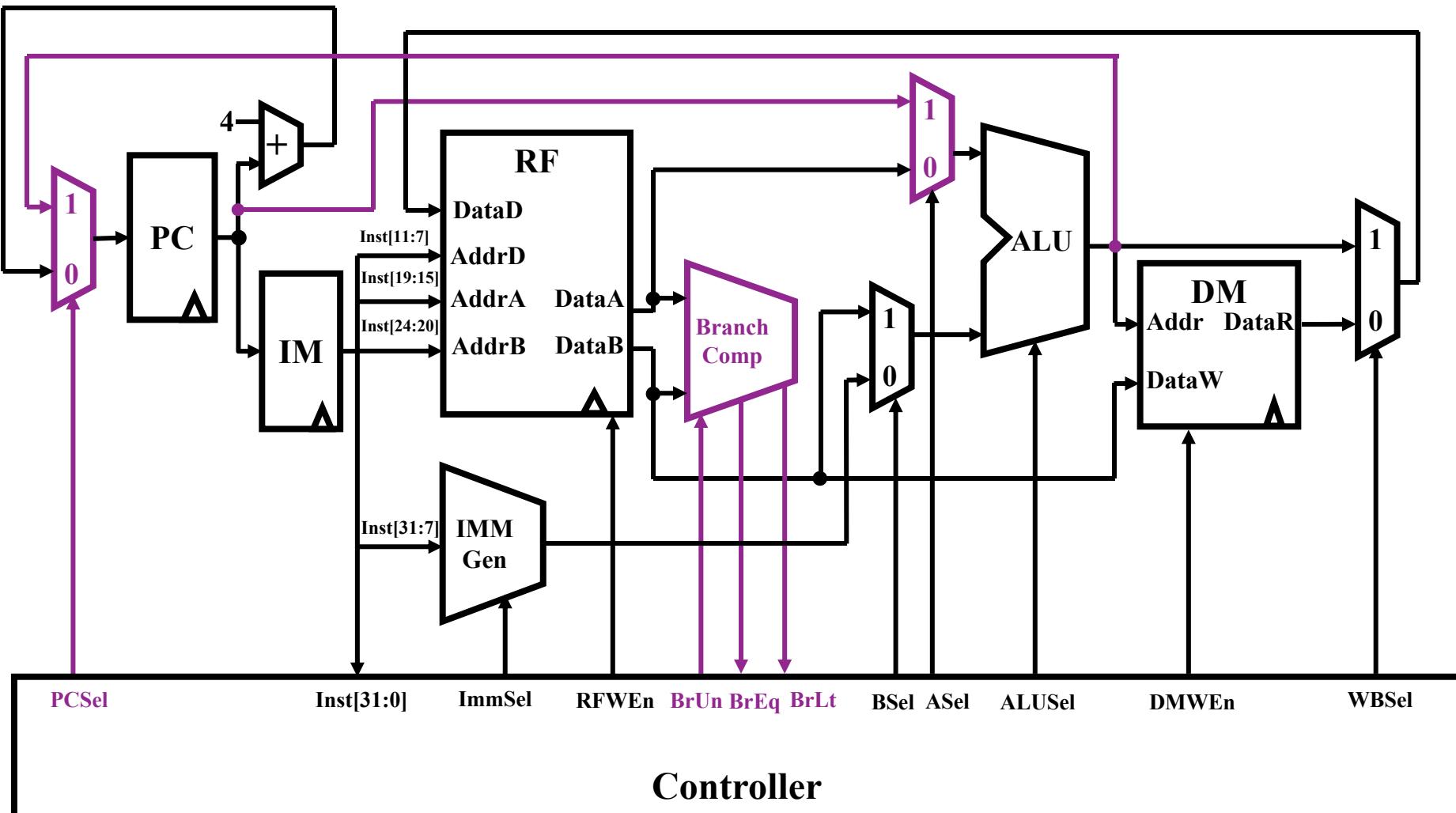
- if  $R[rs1] = R[rs2]$
- then  $PC \leftarrow PC + \text{SignExt}(imm)$
- Else  $PC \leftarrow PC + 4$



# 数据通路设计4



# 数据通路设计5



# U型指令——lui, etc.

□ lui reg, imm

□ 指令功能

- 将20位的立即数装入到寄存器reg的高20位
- 将低12位清零

□ 对数据通路没有提出新的需求



# J型指令——jal, etc.

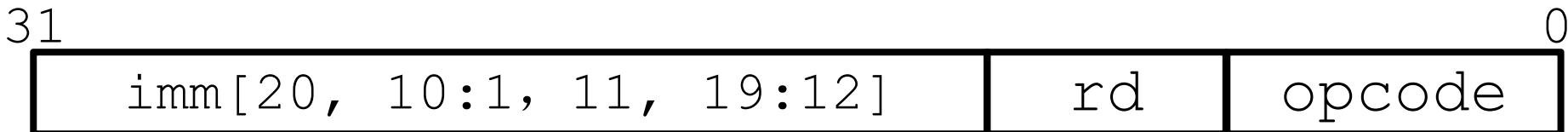
□ Jump: jal rd, imm

□ 指令功能：

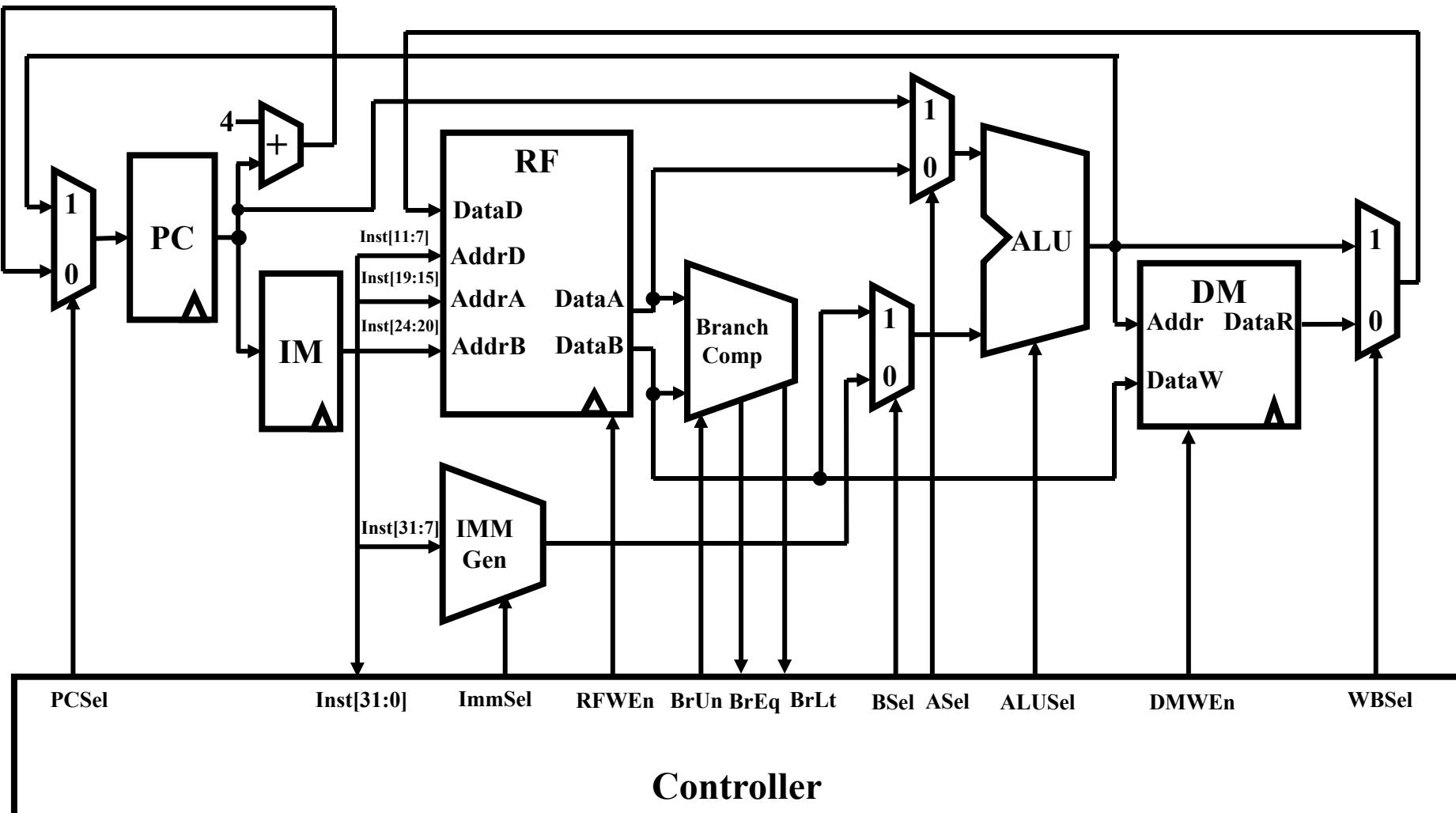
- 置rd值为返回地址 (PC+4)
- 设置跳转  $PC = PC + offset$

□ 思考：是否缺少PC+4到RF的数据通路？

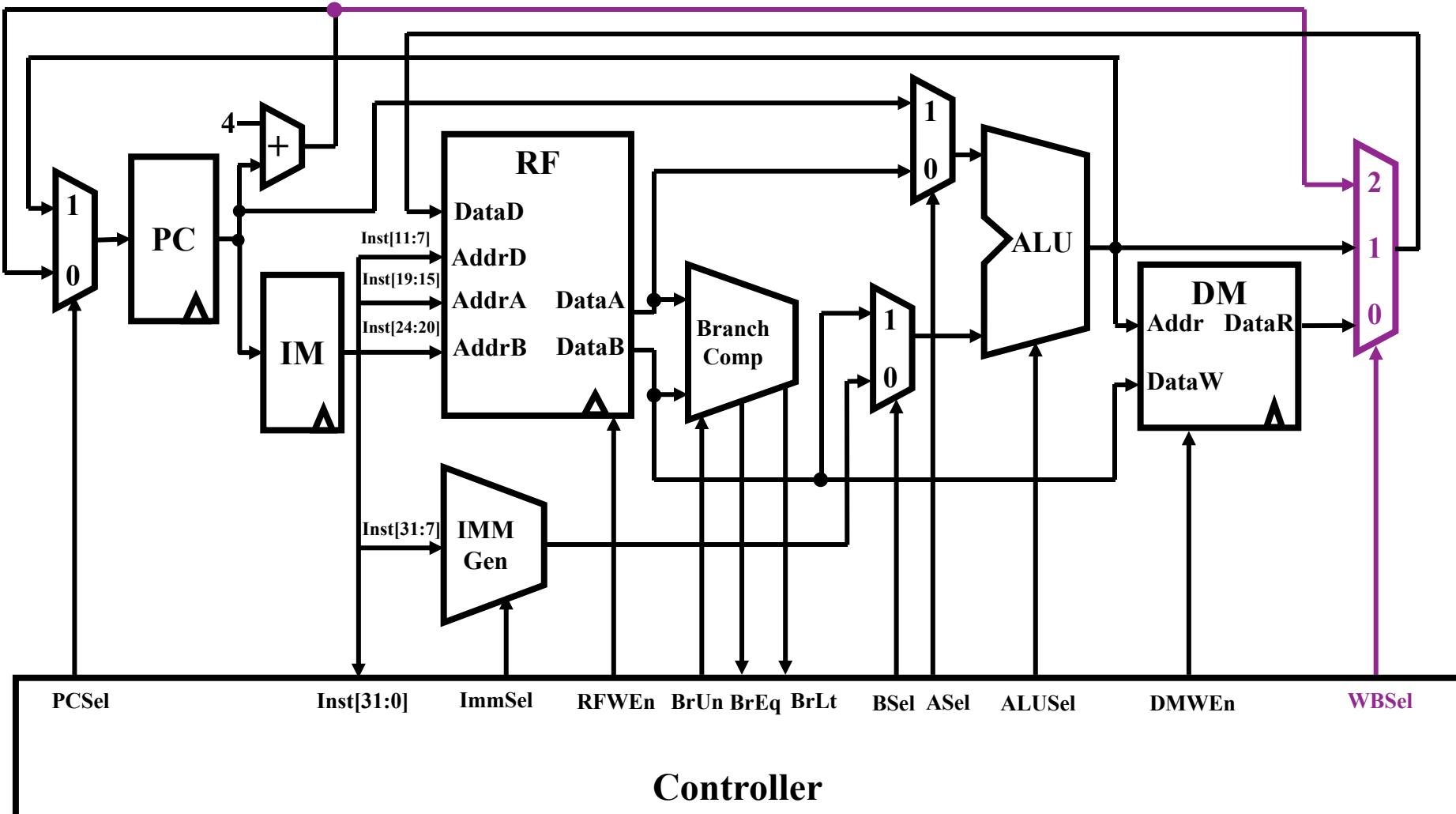
- 修改ALU？
- 新增数据通路？



# 数据通路设计5



# 数据通路设计6



# 跳转I型指令——jalr, etc.

## □ Jalr rd, rs, immediate

- Rd: 返回地址 (PC+4)
- Imm: 偏移offset, 设置跳转  $PC = rs1 + offset$
- Immediate与 I类型指令中的算术和加载指令一样

## □ 思考：现有数据通路是否可以满足该指令功能？

31



# 指令执行过程

---

- 取指令
- 分析指令
- 读取源操作数
  - 寄存器组
  - 立即数
- 计算
- 访存
- 写回寄存器组

# 指令功能如何实现？

---

## □ 硬件组成

- ALU
- MEM
- Register File
- ADDer
- PC
- Mux

## □ 数据通路

- 如何实现指令的功能？

# 指令的执行过程与控制

□ 冯·诺依曼结构的计算机，即存储程序计算机，设置内存，存放程序和数据，在程序运行之前存入。

□ 执行程序：

- 正确从程序首地址开始；
- 正确分步地执行每一条指令，
- 还要形成下条待执行指令的地址；
- 并保证自动地连续执行指令，
- 直到结束程序的最后一条指令。

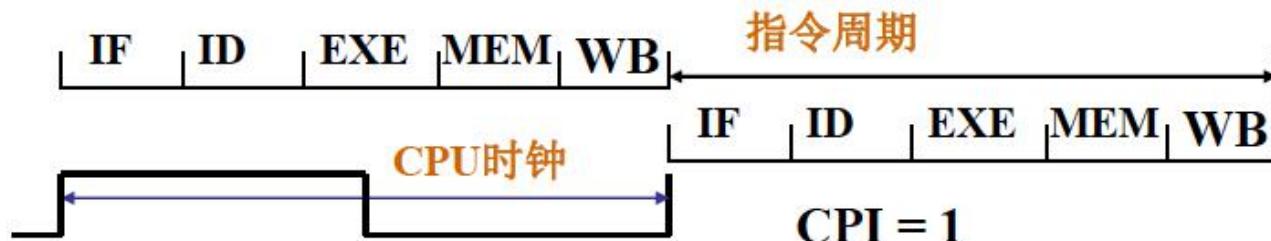
□ 从主存储器读来一条指令，分析指令，按指令的功能要求完成执行过程，本条指令完成后自动开始下一条指令的执行过程，由硬件本身完成。

# 指令的执行步骤

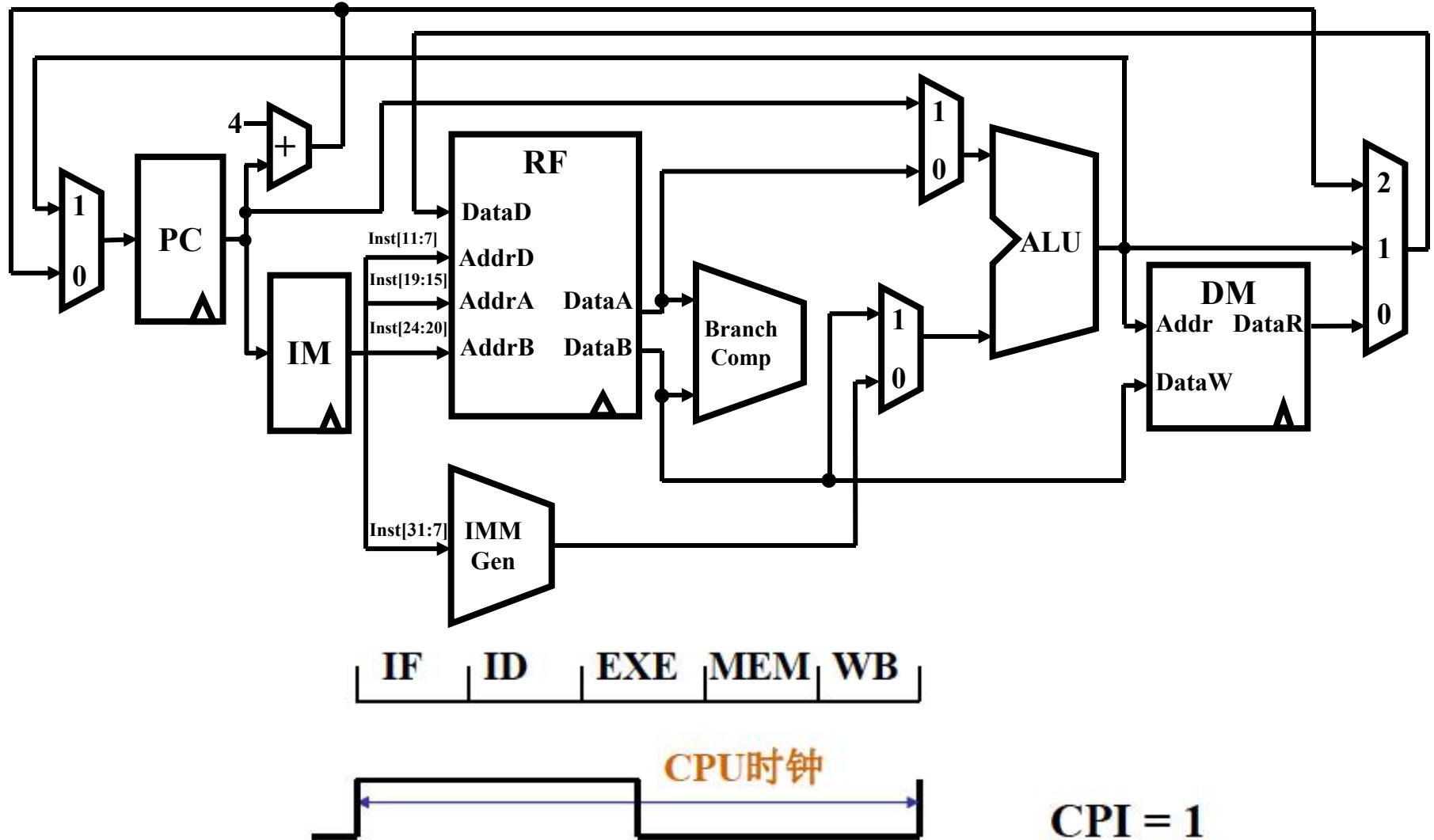
- 当前的计算机系统中，流行的做法是把一条指令的执行过程划分为如下的5个阶段，是由指令的功能和计算机硬件结构共同决定的，有内在的逻辑关系。
  - 读取指令(IF), 从存储器读来指令并形成下条指令地址
  - 指令译码(ID), 指令译码, 读寄存器堆为ALU准备数据
  - 执行运算(EXE), ALU 执行数据运算或计算存储器地址
  - 存储器读写(MEM), 完成存储器的读操作或者写操作
  - 写回(WB), 写ALU的结果或存储器读出数据到寄存器堆
- 从如何为不同指令安排这几个阶段，几个阶段如何衔接考虑，大体有3种可行方案，各自都对计算机的内部结构、部件设置及其控制方式有着不同要求。

# 单周期CPU

- 计算机一条指令的执行时间被称为**指令周期**，一个CPU时钟时间被称为**CPU周期**(在某些计算机中，还可再把一个CPU周期区分为几个更小的步骤，称其为节拍)。执行**每条指令平均使用的CPU周期个数**被称为**CPI**
- 全部指令都选用**一个CPU周期**完成的系统被称为**单周期CPU**，指令串行执行，前一条指令结束后才启动下条指令。每条指令都用5个步骤的时间完成，控制各部件运行的信号在整个指令周期不变化。**单周期CPU**用于早期计算机，系统性能和资源利用率很低，相对当前技术变得**不再实用**。

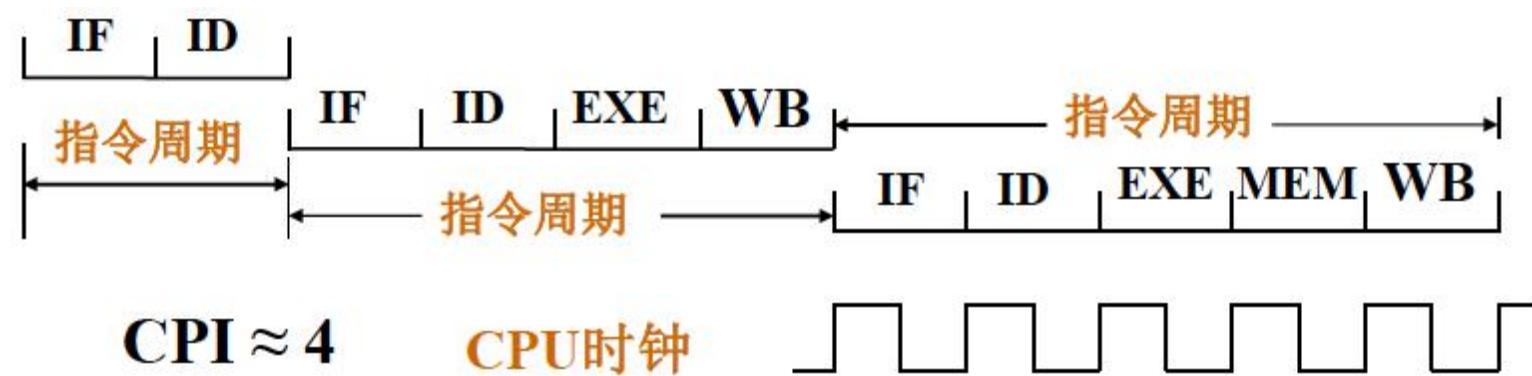


# 单周期CPU

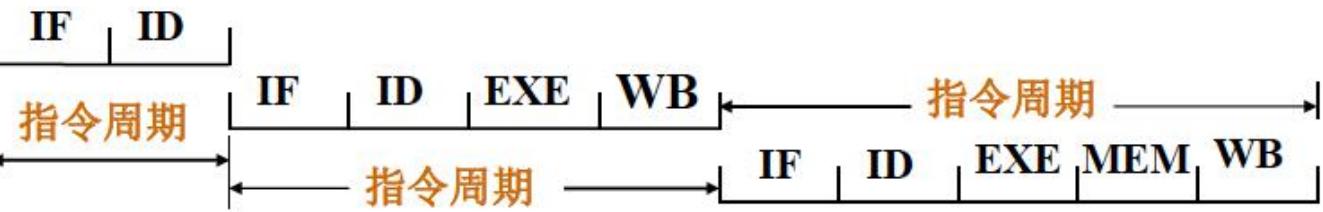


# 多周期CPU

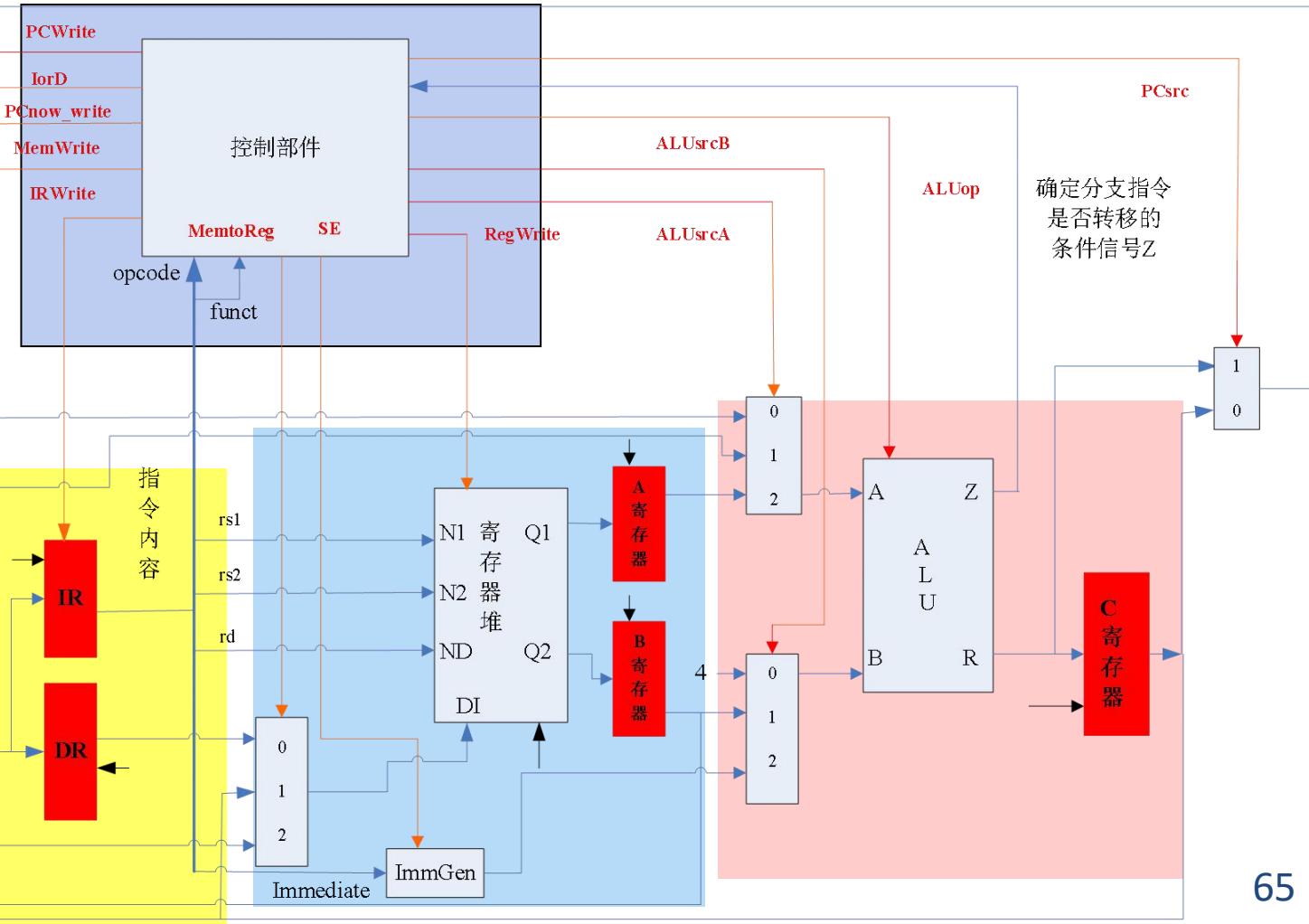
- 计算机一条指令的执行时间被称为指令周期，一个CPU时钟时间被称为CPU周期。
- 依据不同指令各自的功能需求为其选择不等的执行步骤的系统被称为多周期CPU，控制各部件运行的控制信号随着指令执行步骤改变，系统性能和资源利用率更高。相邻指令可以完全串行执行，也可能部分时间重叠，多周期CPU（相比单周期CPU）更实用。



# 多周期CPU

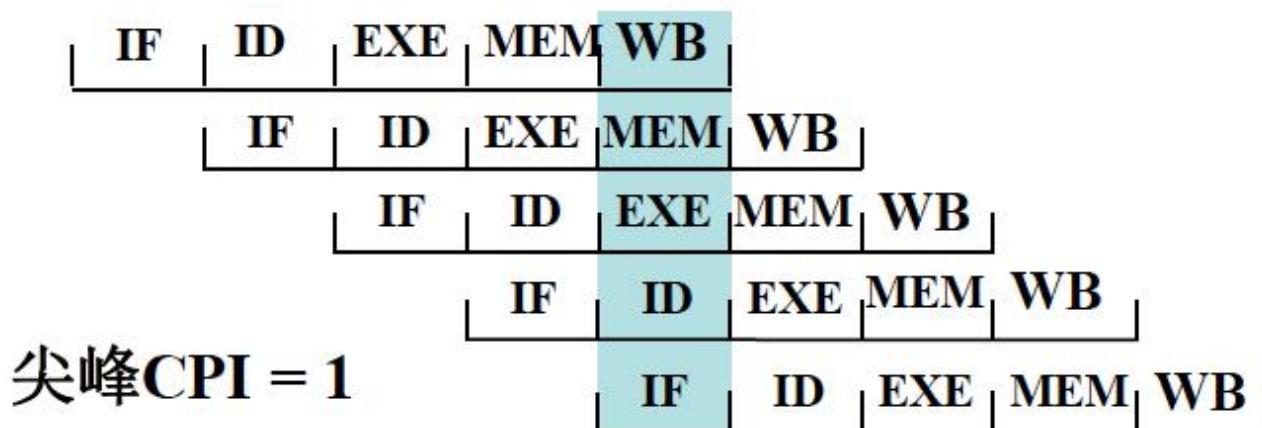


写入PC的指令地址



# 指令流水线CPU

- 全部指令都是选用5个步骤完成，执行时间相同,但相邻指令的执行并不是完全串行的，执行时间有所重叠，例如每结束指令的一个执行步骤就启动下条指令，这被称为**指令流水线技术**，所有部件都高速运行，尖峰速度每个CPU时钟执行一条指令，系统性能和资源利用率更高，显著地提高系统的性能价格比，但**计算机结构和控制器的设计、实现略显复杂**。**当前计算机中普遍使用这种方案。**



# 小结

---

- 指令系统受到技术条件的制约
- 兼容性是指令系统的重要要求
- RISC，以简洁换取性能提高
- CISC，以丰富换取编程方便
- 指令执行
  - 单步骤串行
  - 多步骤串行
  - 流水

---

谢谢



# 单周期CPU控制器设计

2022年秋

# 本讲提要

---

- 指令执行方式和步骤
- 单周期CPU设计
  - 指令集：9条指令为例
  - 数据通路
  - 控制器

# CPU设计

---

## □ 完成指令功能的主要部件

- ALU

## □ 指令功能

- 数据运算：算术、逻辑、移位
- 数据移动：内存到寄存器之间的数据拷贝
- 流程控制：转移、调用/返回、中断
- 其它：优先级控制，虚拟内存

## □ 如何实现？

- Datapath：实现数据的移动和运算
- 控制器：指挥数据的移动和运算

# 每条指令的执行过程

---

## □ 第一步

- 取指令 (IF)

## □ 第二步

- 指令译码 (ID)

## □ 第三步

- 执行指令 (EXE)

## □ 第四步

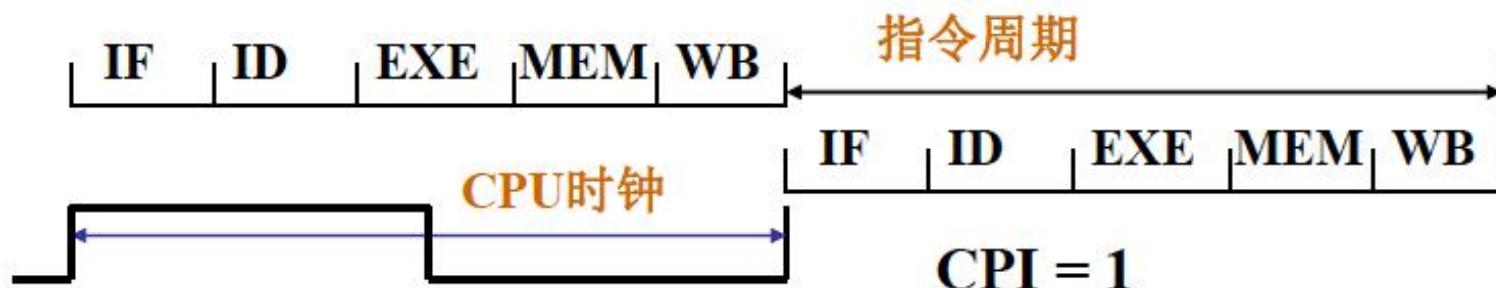
- 访问存储器 (MEM)

## □ 第五步

- 写回寄存器 (WB)

# 指令执行步骤 —— 单周期CPU

- 计算机一条指令的执行时间被称为**指令周期**，一个CPU时钟时间被称为**CPU周期**(在某些计算机中，还可再把一个CPU周期区分为几个更小的步骤，称其为**节拍**)。执行**每条指令平均使用的CPU周期个数**被称为**CPI**
- 全部指令都选用**一个CPU周期**完成的系统被称为**单周期CPU**，指令串行执行，前一条指令结束后才启动下条指令。每条指令都用5个步骤的时间完成，控制各部件运行的信号在整个指令周期不变化。**单周期CPU** 用于早期计算机，系统性能和资源利用率很低，相对当前技术变得**不再实用**。



# 实现的指令集

□ 选取RISC-V指令中9条典型指令组成的子集

- 访存指令：lw、sw
- 算逻运算指令：add、sub、andi、auipc
- 转移指令：beq、jal、jalr

□ 解决三个主要问题

- 数据通路设计
- 控制信号设计
- 执行时序设计

□ 其他指令的实现原理可以从中体现

- 上面9条指令覆盖了所有的RISC-V指令集的不同类型指令

# 设计思路

## □ 指令的执行

- 显然要设计一个时序逻辑电路
- 一条指令用一个CPU周期完成

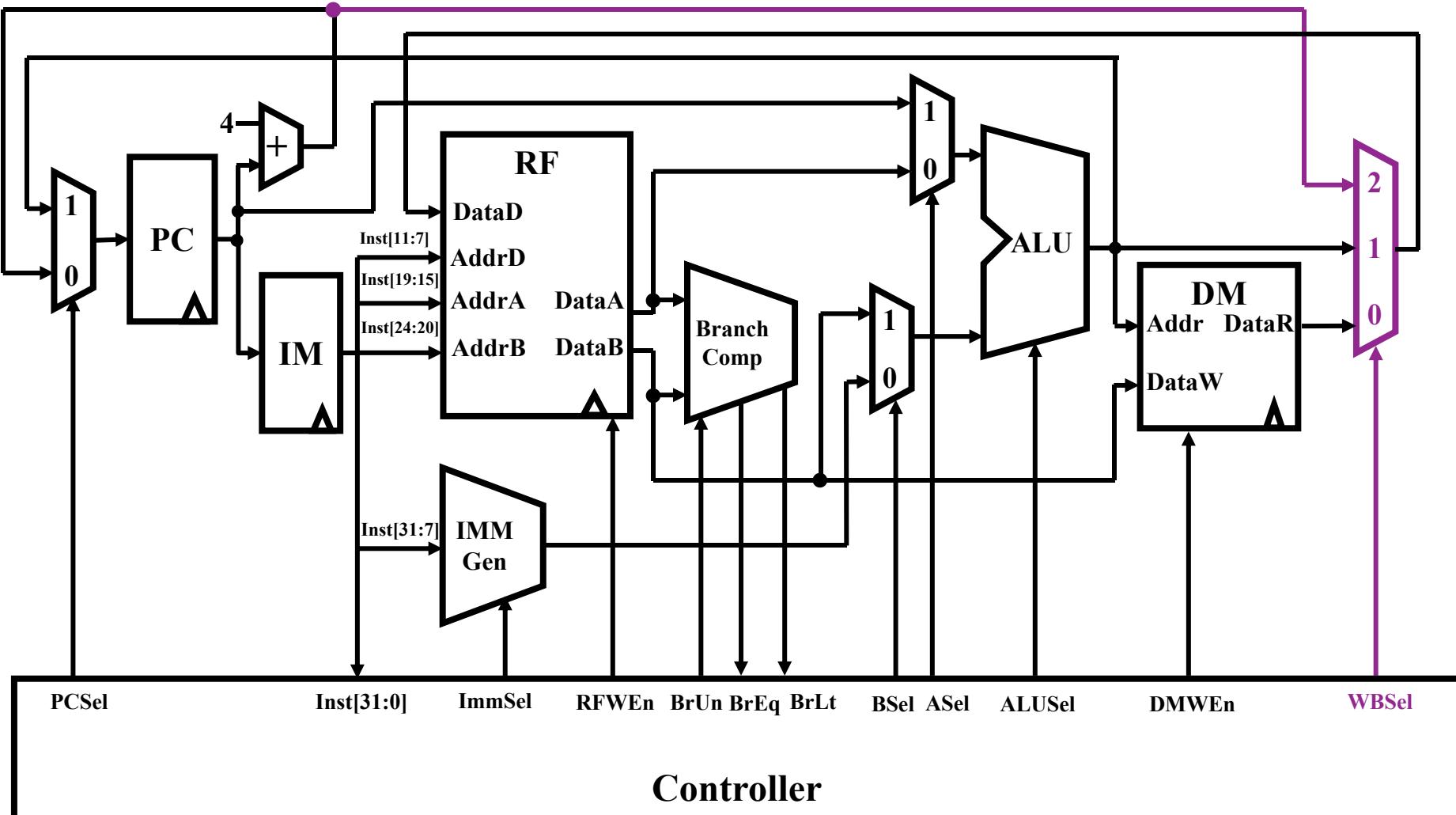
## □ 执行步骤的实现

- 取指：从指令存储器中读指令（地址：PC）
- 译码：读出一或两个源寄存器的值（寄存器组）
- 运算：进行指令规定的运算（ALU）
- 访存：读/写数据存储器
- 写回：将结果写入目的寄存器

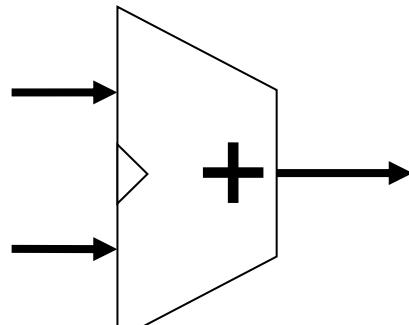
## □ 需要保存的值

- PC、寄存器组、存储器

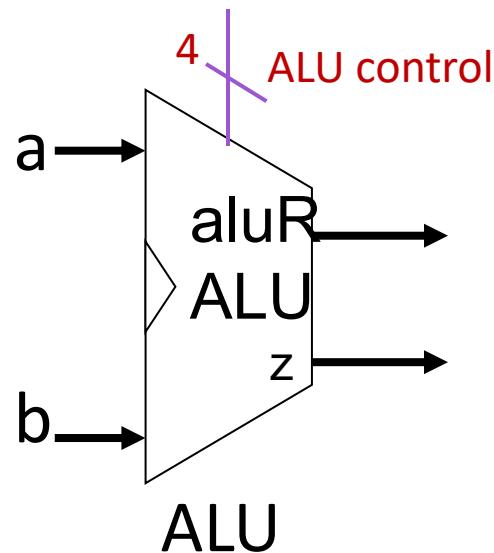
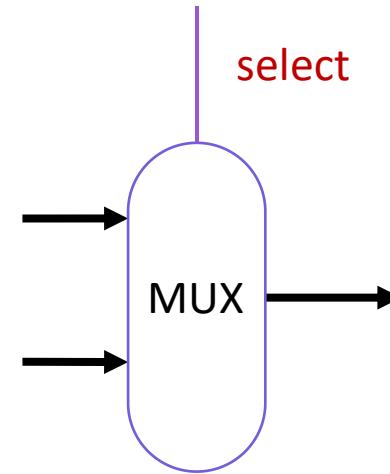
# 初步的Datapath



# 使用的组合逻辑部件

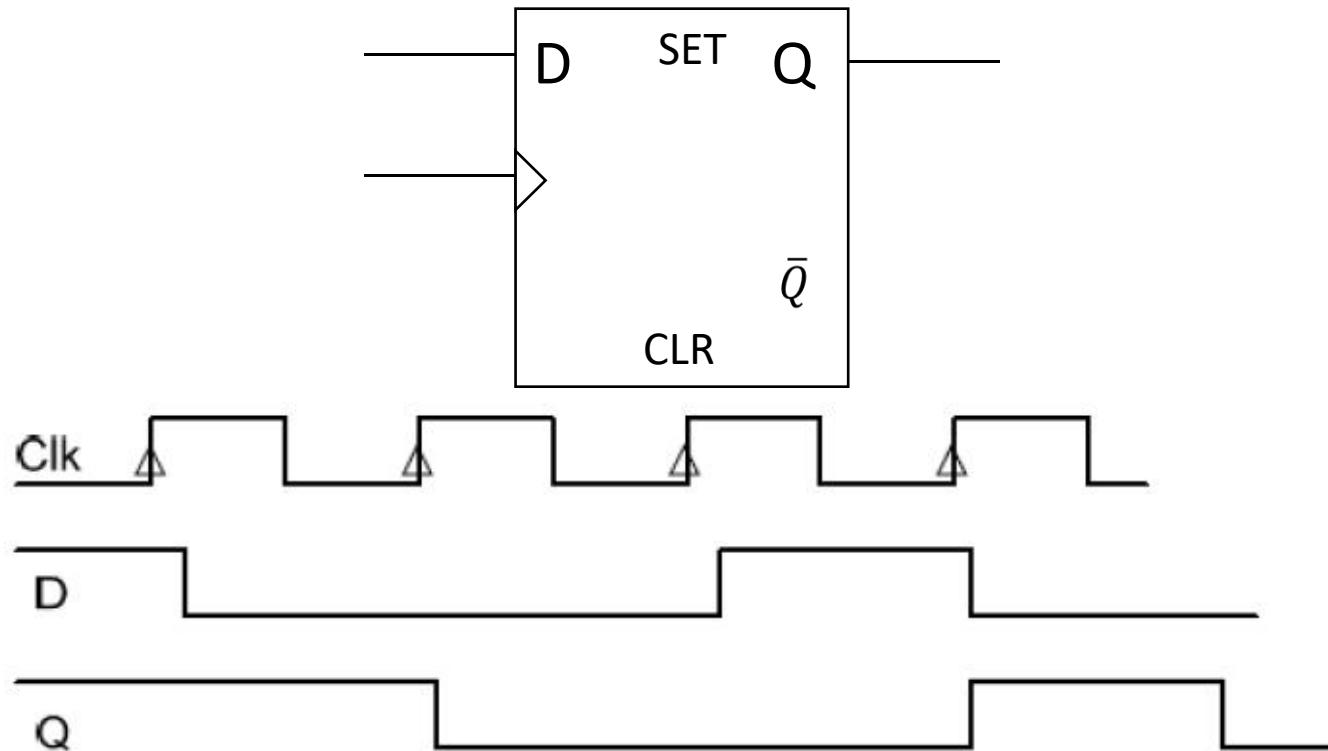


Adder

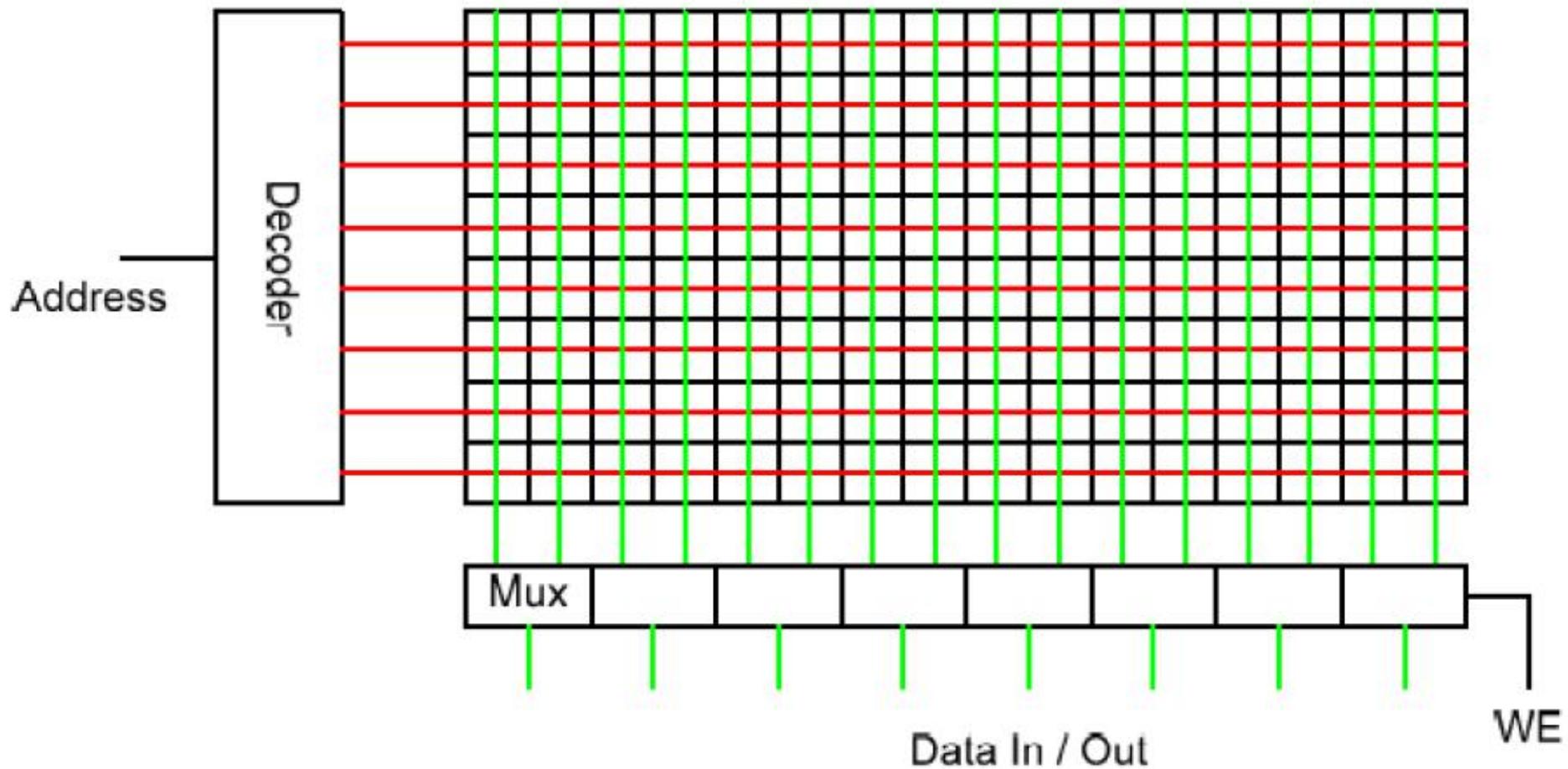


# D触发器

- 在时钟的上升沿写入输入数据
- 一直保持到下一个上升沿



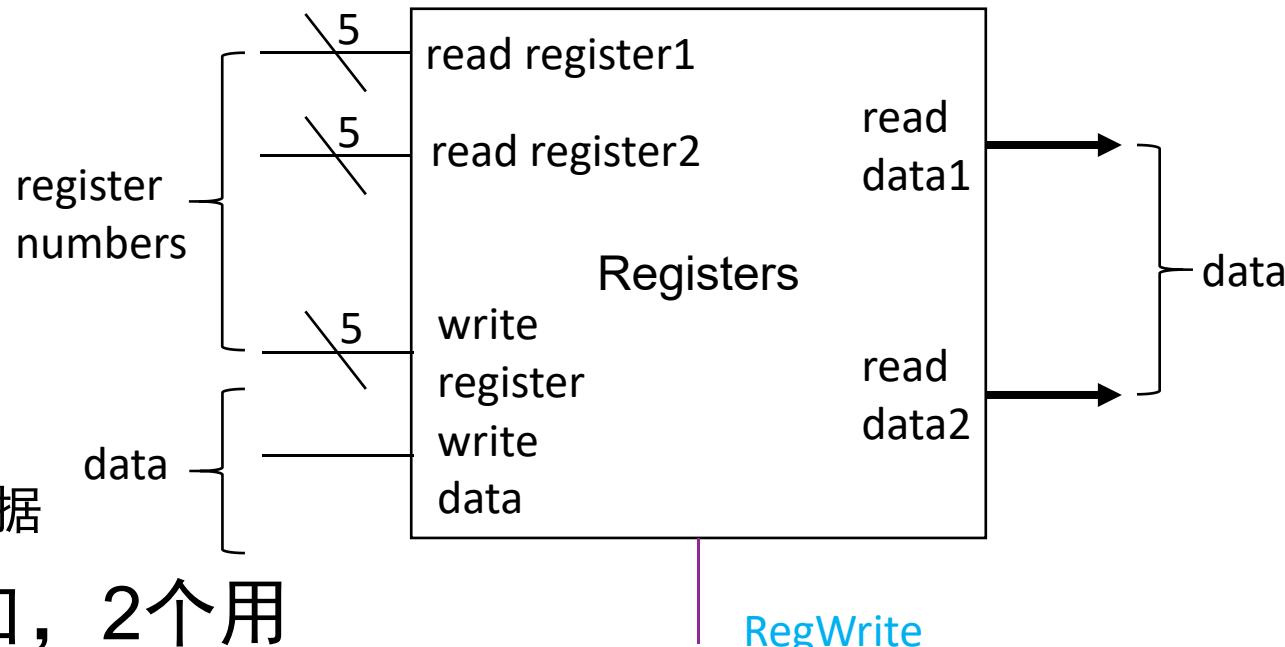
# 存储器



# 寄存器组

## □ 接口简单

- 输入
  - 地址
  - 写入数据
  - 写信号
- 输出
  - 读出的数据



□ 3个地址端口，2个用

来读，1个用来写

□ 每个时钟周期可以完  
成3次访问

# 时序设计

## □ 每条指令占用一个时钟周期

- 取指令后分析指令，并给出整个执行期间的全部信号
- 不需要状态信息，在时钟的结束的边沿写入结果

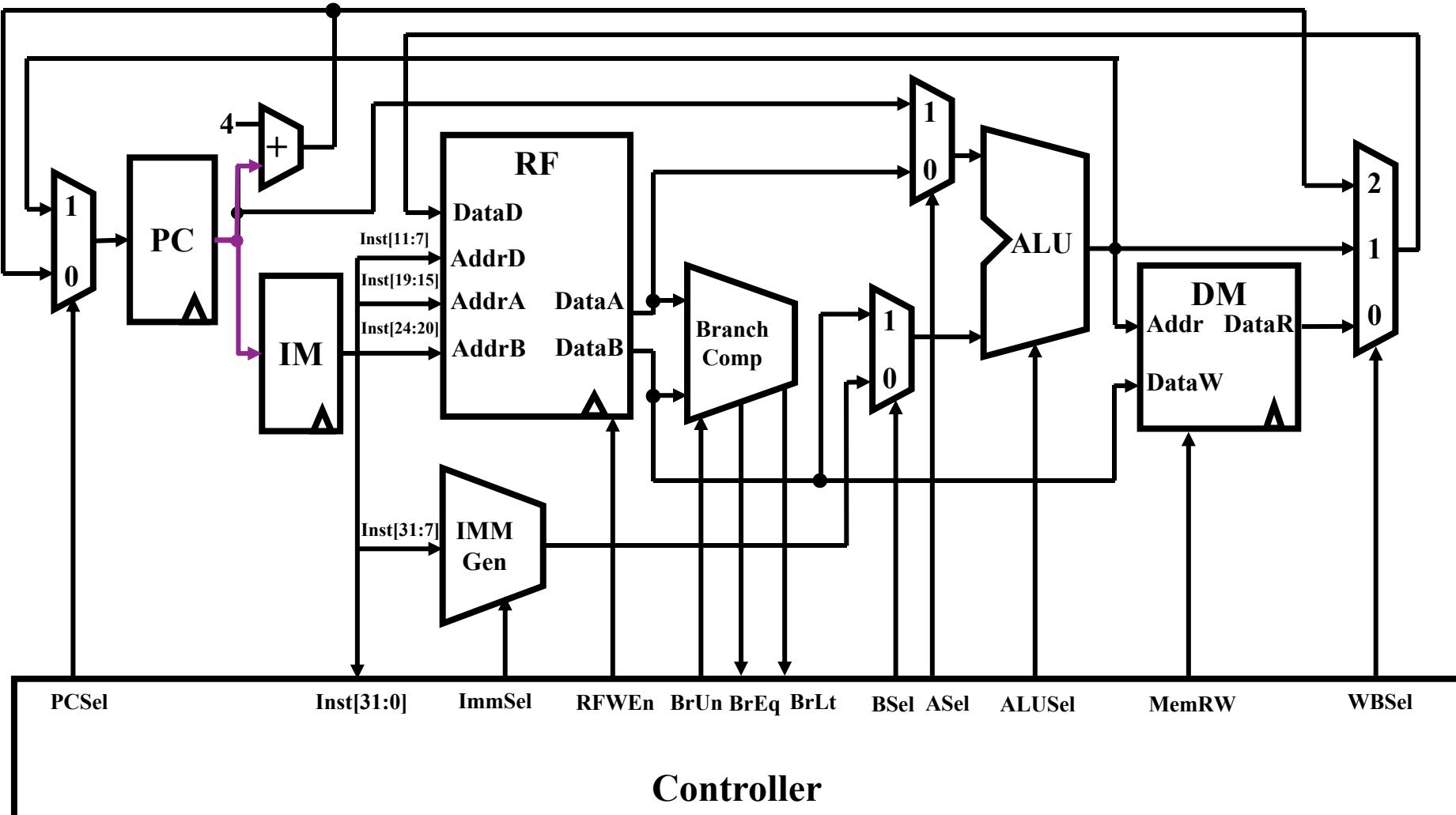
## □ 控制对象

- ALU的运算
- 寄存器组和存储器的写入
- 多路选通器

## □ 时钟周期开始时读取指令

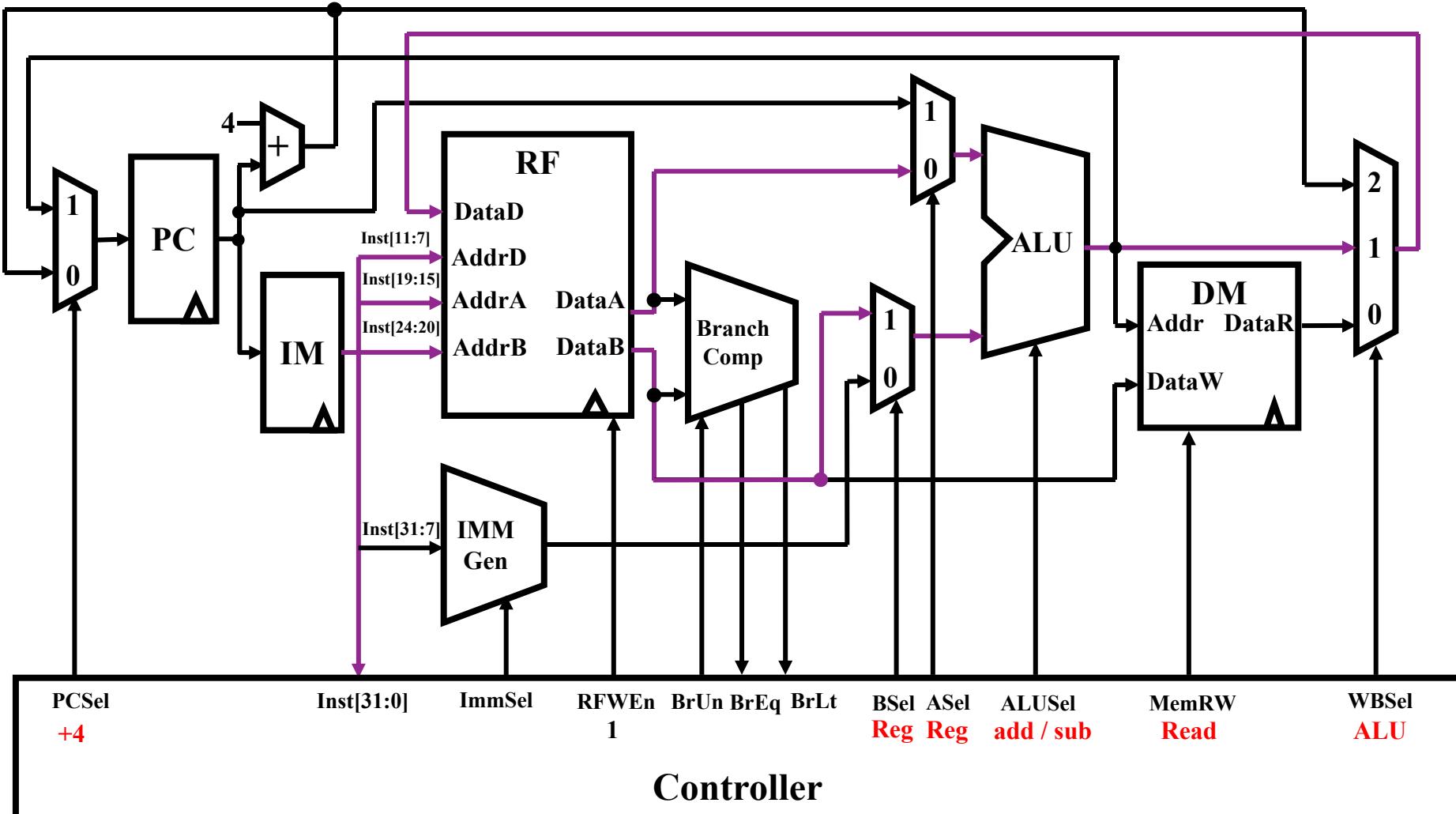
- 与具体指令无关

# 时钟周期开始时的控制



# 算术逻辑运算指令的控制

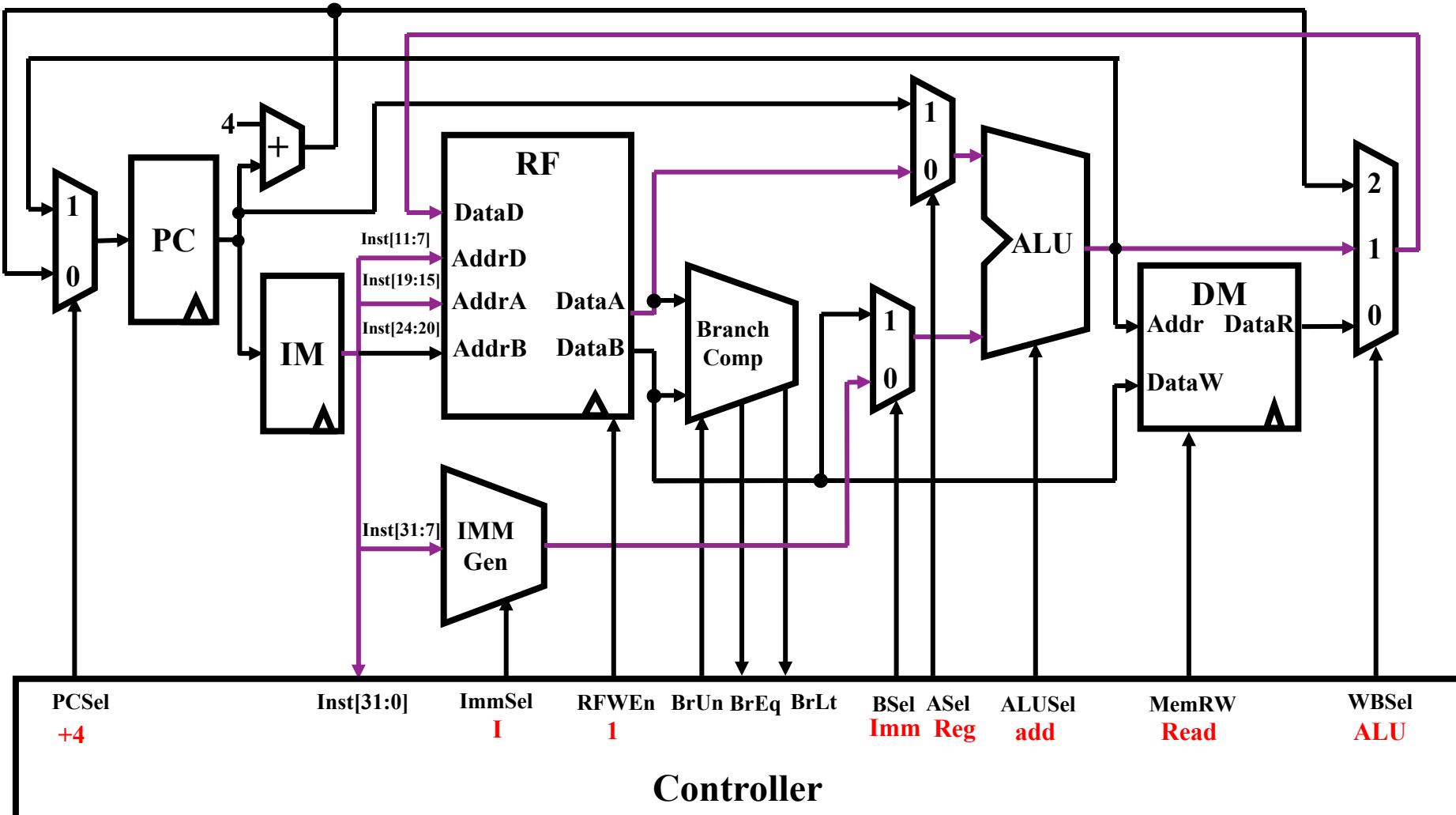
Add rd r1 r2  
Sub rd r1 r2



Controller

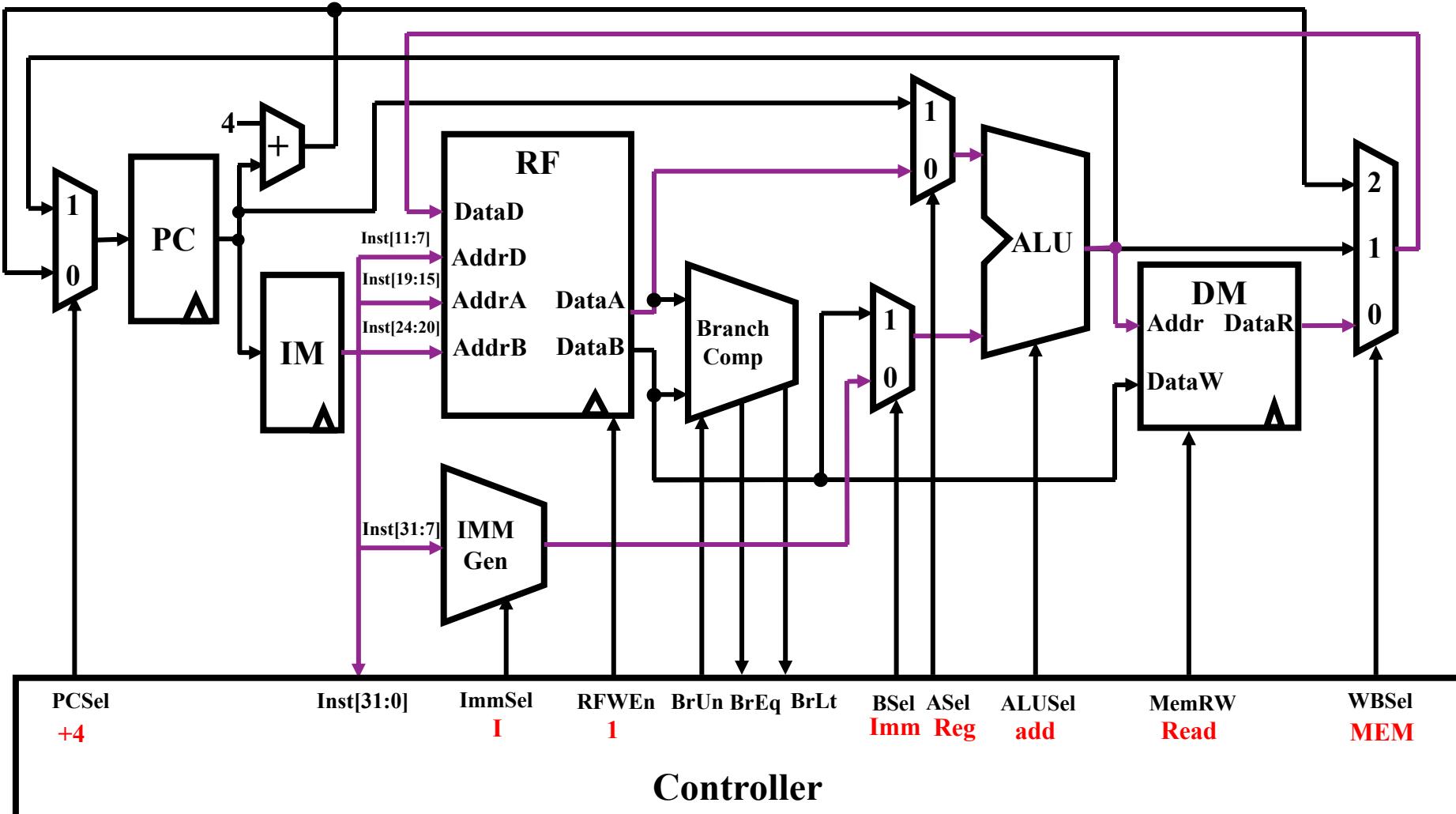
# 算术逻辑运算指令的控制

Addi rd r1 imm



# Load指令

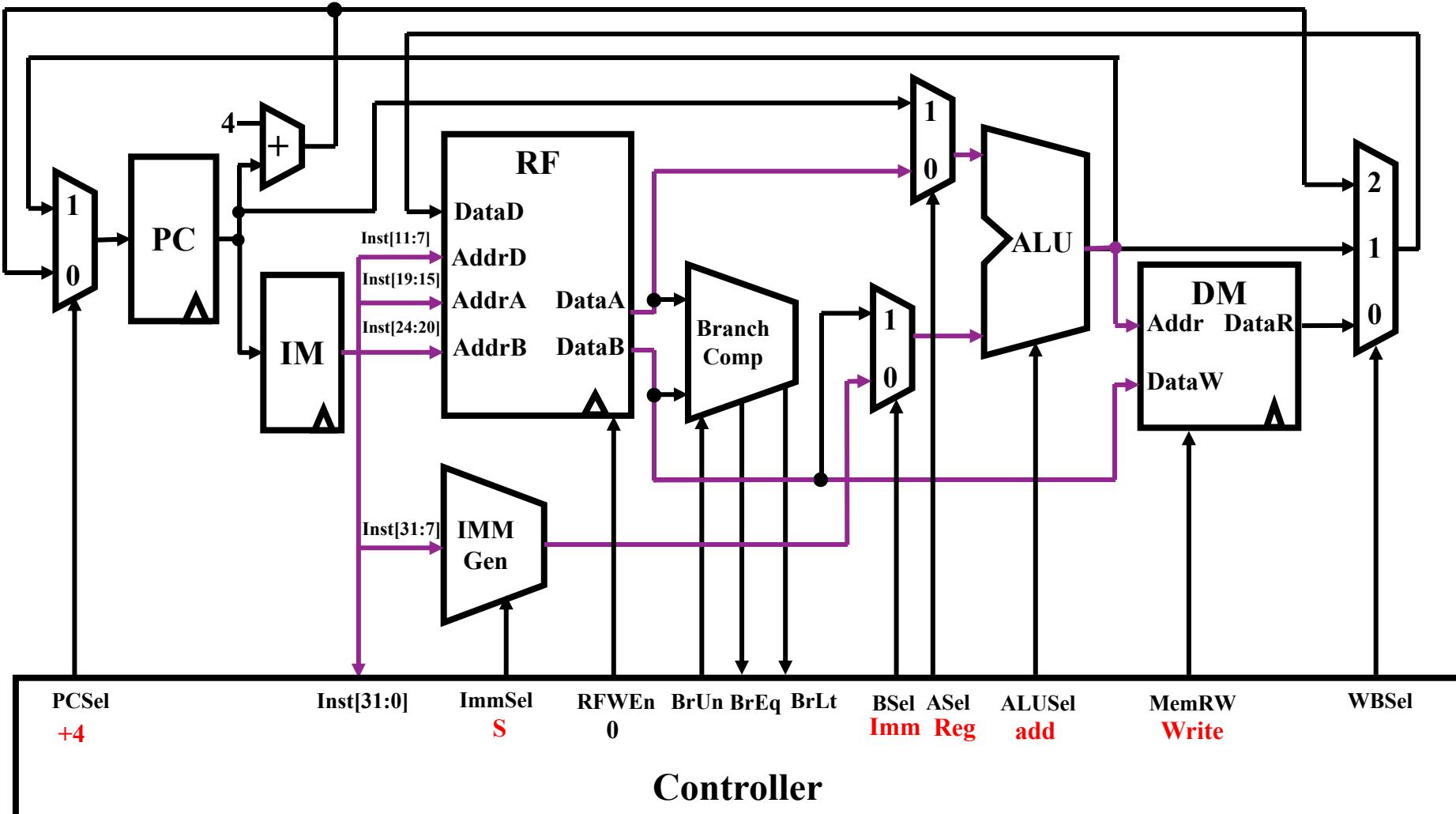
lw rd, rs1, imm



Controller

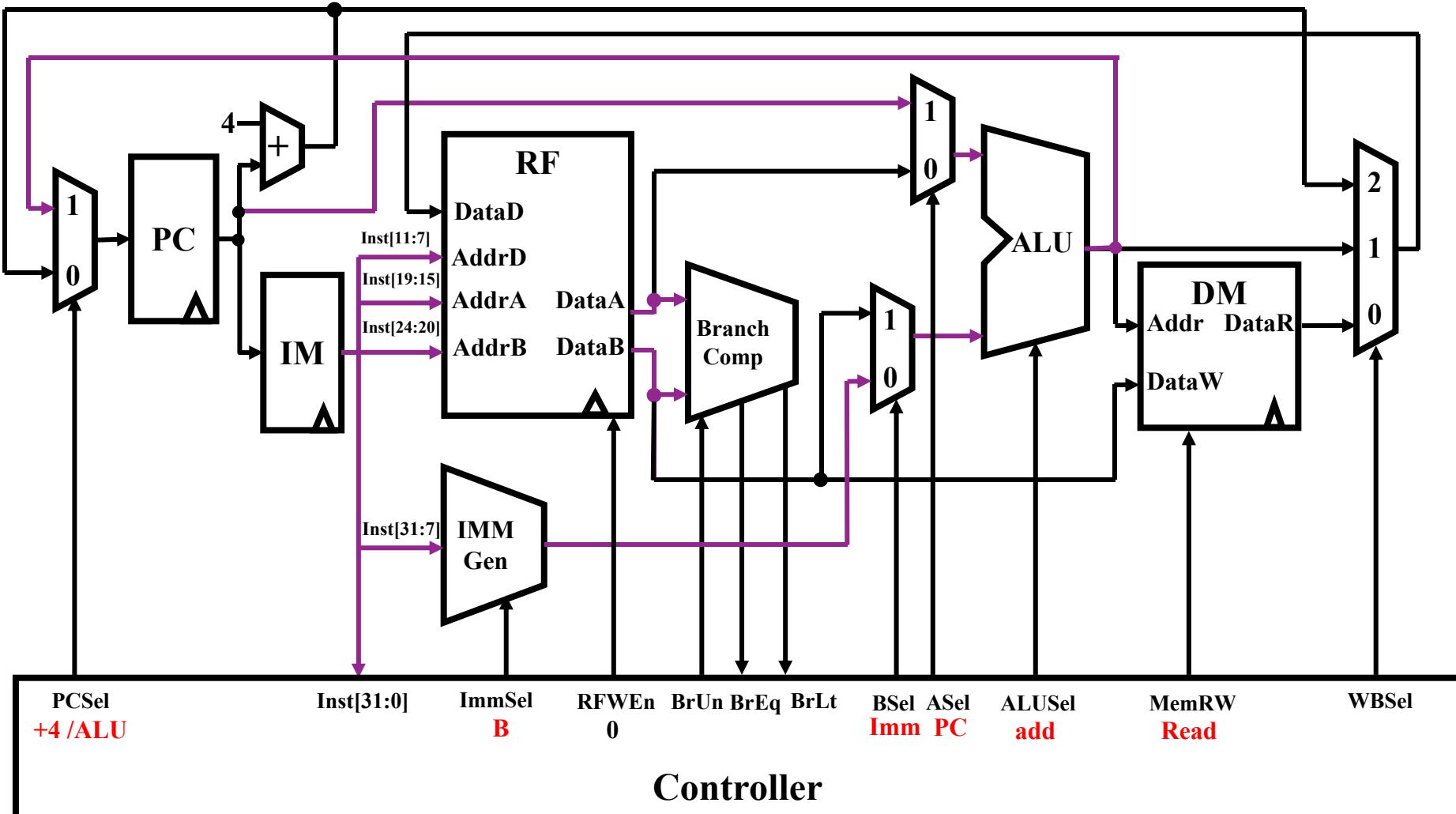
# Store指令

SW rs2, rs1, imm



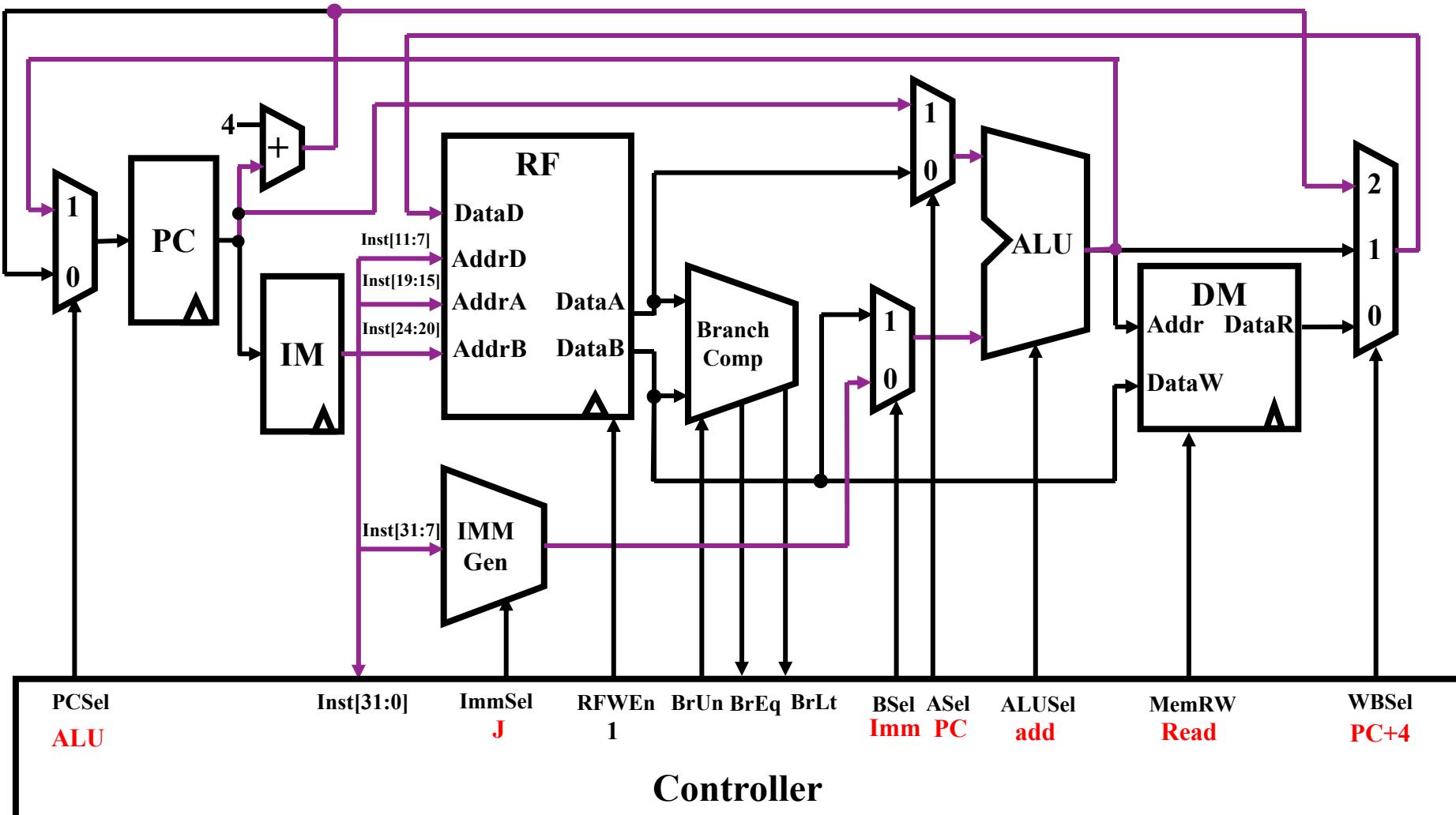
# Beq指令

beq rs1, rs2, label



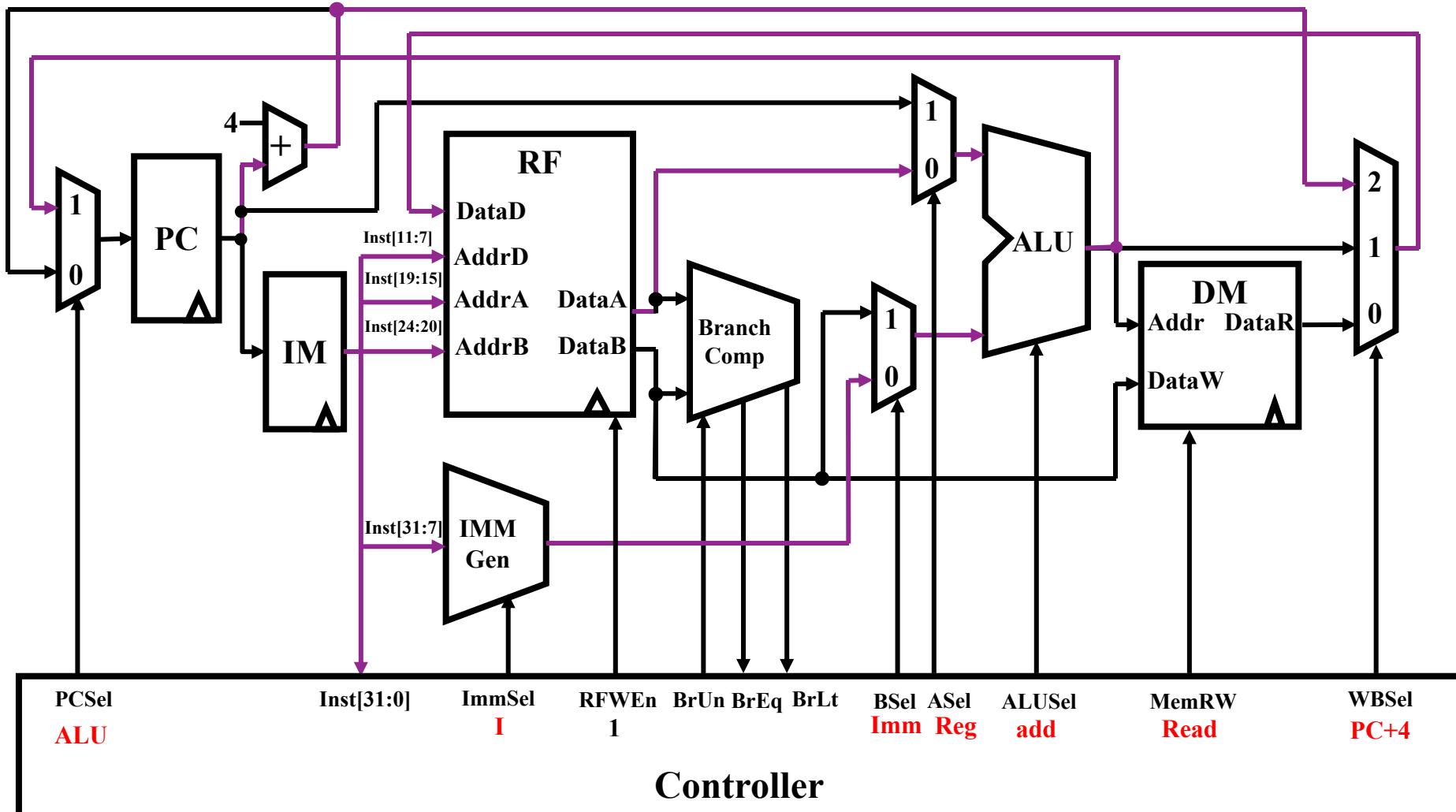
# Jal 指令

jal rd, imm



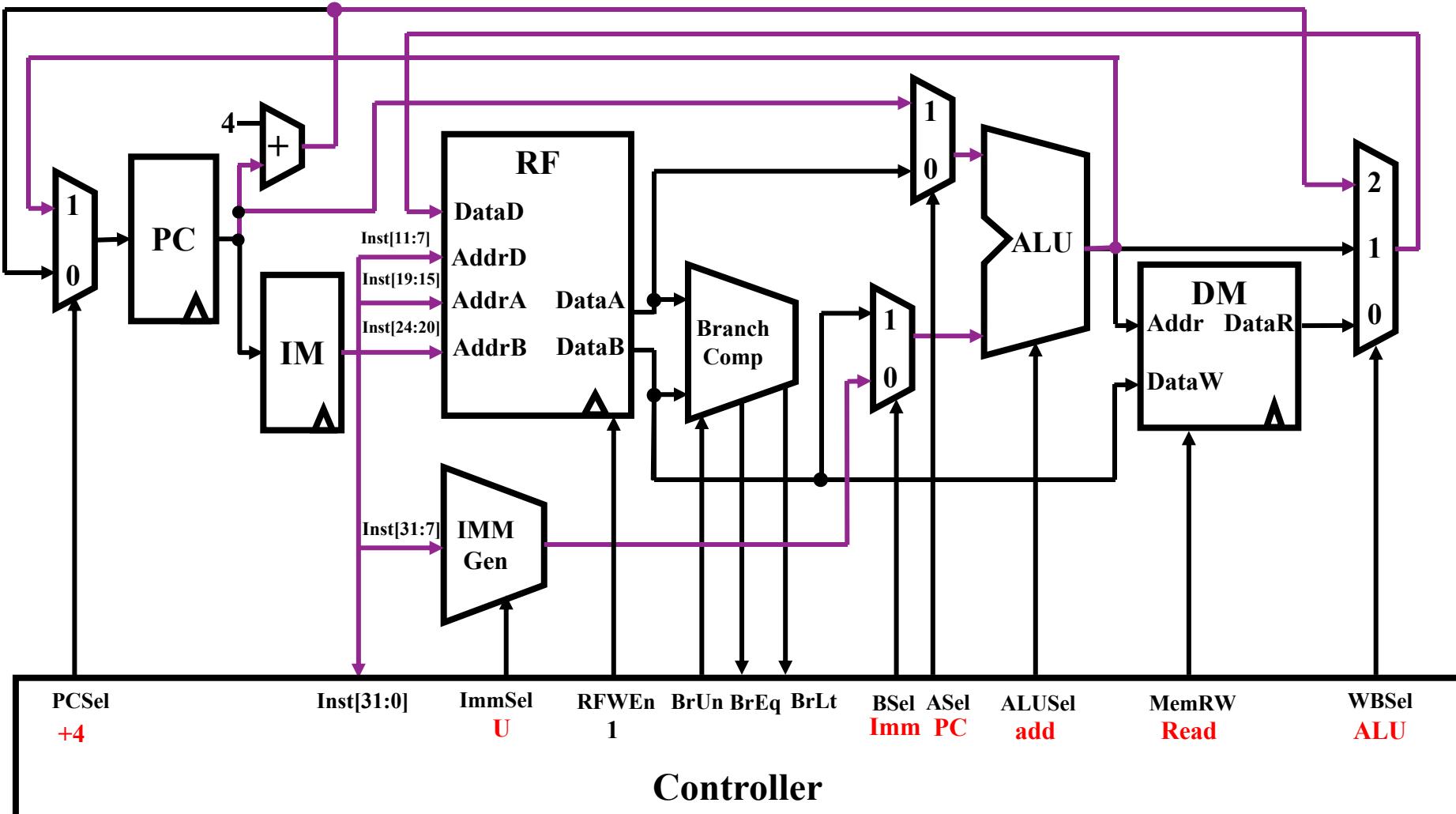
# Jalr 指令

jalr rd, rs1, imm



# auipc 指令

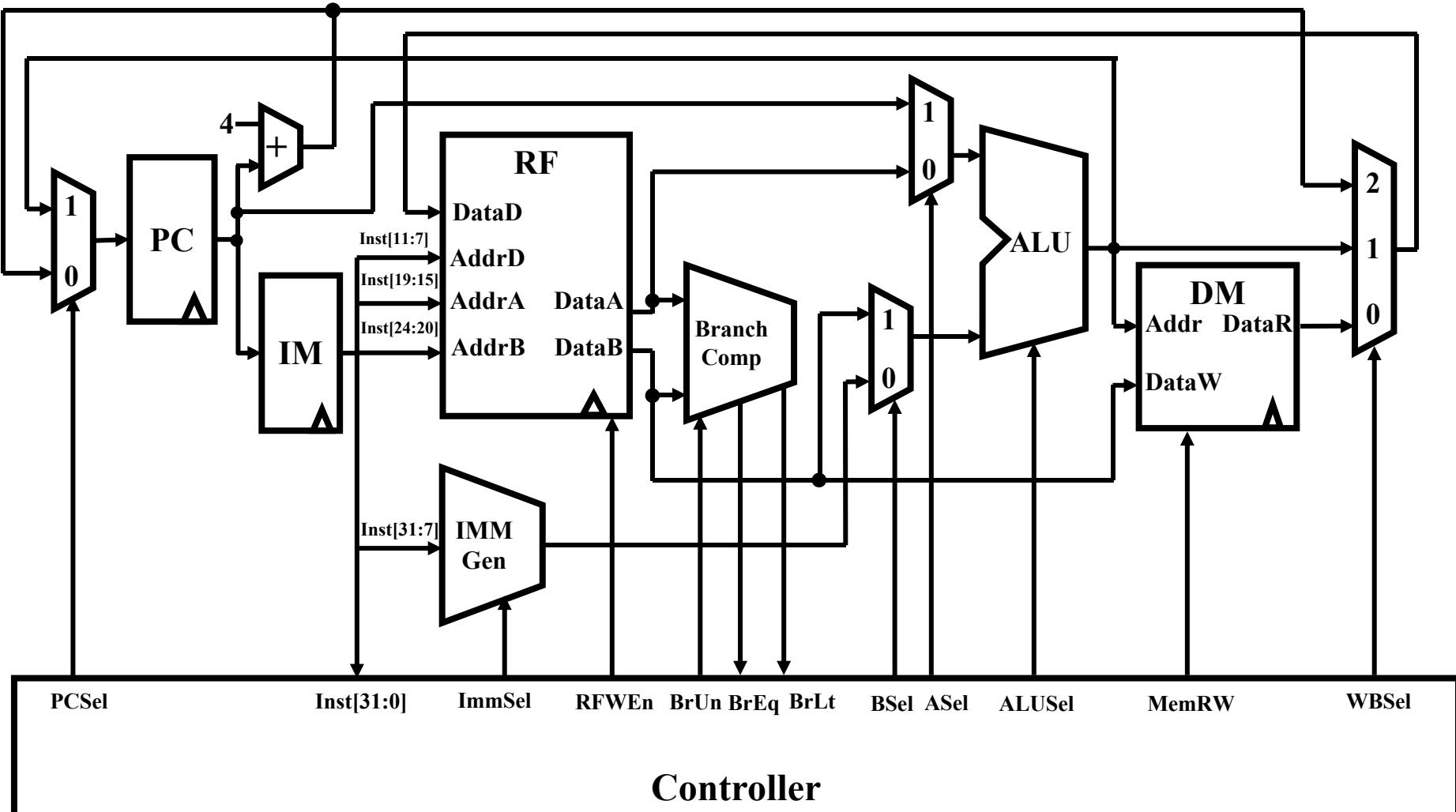
auipc rd, imm



# 信号整理

| inst[31:0]      | BrEq | BrLT | PCSel | ImmSel | BrUn | ASel | BSel | ALUSel | MemR W | RegWE n | WBSel |
|-----------------|------|------|-------|--------|------|------|------|--------|--------|---------|-------|
| <b>add</b>      | -    | -    | +4    | -      | -    | Reg  | Reg  | Add    | Read   | 1       | ALU   |
| <b>sub</b>      | -    | -    | +4    | -      | -    | Reg  | Reg  | Sub    | Read   | 1       | ALU   |
| <b>(R-R Op)</b> | -    | -    | +4    | -      | -    | Req  | Req  | (Op)   | Read   | 1       | ALU   |
| <b>addi</b>     | -    | -    | +4    | I      | -    | Reg  | Imm  | Add    | Read   | 1       | ALU   |
| <b>lw</b>       | -    | -    | +4    | I      | -    | Reg  | Imm  | Add    | Read   | 1       | Mem   |
| <b>sw</b>       | -    | -    | +4    | S      | -    | Reg  | Imm  | Add    | Write  | 0       | -     |
| <b>beq</b>      | 0    | -    | +4    | B      | -    | PC   | Imm  | Add    | Read   | 0       | -     |
|                 | 1    | -    | ALU   | B      | -    | PC   | Imm  | Add    | Read   | 0       | -     |
| <b>bne</b>      | 0    | -    | ALU   | B      | -    | PC   | Imm  | Add    | Read   | 0       | -     |
|                 | 1    | -    | +4    | B      | -    | PC   | Imm  | Add    | Read   | 0       | -     |
| <b>blt</b>      | -    | 1    | ALU   | B      | 0    | PC   | Imm  | Add    | Read   | 0       | -     |
| <b>bltu</b>     | -    | 1    | ALU   | B      | 1    | PC   | Imm  | Add    | Read   | 0       | -     |
| <b>jalr</b>     | -    | -    | ALU   | I      | -    | Reg  | Imm  | Add    | Read   | 1       | PC+4  |
| <b>jal</b>      | -    | -    | ALU   | J      | -    | PC   | Imm  | Add    | Read   | 1       | PC+4  |
| <b>auipc</b>    | -    | -    | +4    | U      | -    | PC   | Imm  | Add    | Read   | 1       | ALU   |

# 完整的单周期CPU



# 单周期CPU特点

---

## □ 优点

- 每条指令占用一个时钟周期
- 逻辑设计简单，时序设计也简单

## □ 缺点

- 各组成部件的利用率不高
  - 维持有效信号
- 时钟周期应满足执行时间最长指令的要求
  - Load指令

## □ CPI =1

# 小结

---

## □ 单周期CPU设计

- 全部控制信号
- 无需状态信息
- 控制信号生成：组合逻辑电路
  - 输入：inst, breq, brlt
  - 输出：完成指令功能所需要的全部控制信号

---

谢谢



# 动态存储器

2022年秋

# Look back the CPU ISA

---

When RISC first came out, x86 was half microcode. So if you look at the die, half the chip is a ROM, or maybe a third or something. And the RISC guys could say that there is no ROM on a RISC chip, so we get more performance. But now the ROM is so small, you can't find it. Actually, the adder is so small, you can hardly find it? **What limits computer performance today is predictability**, and the two big ones are **instruction/branch predictability**, and **data locality**.

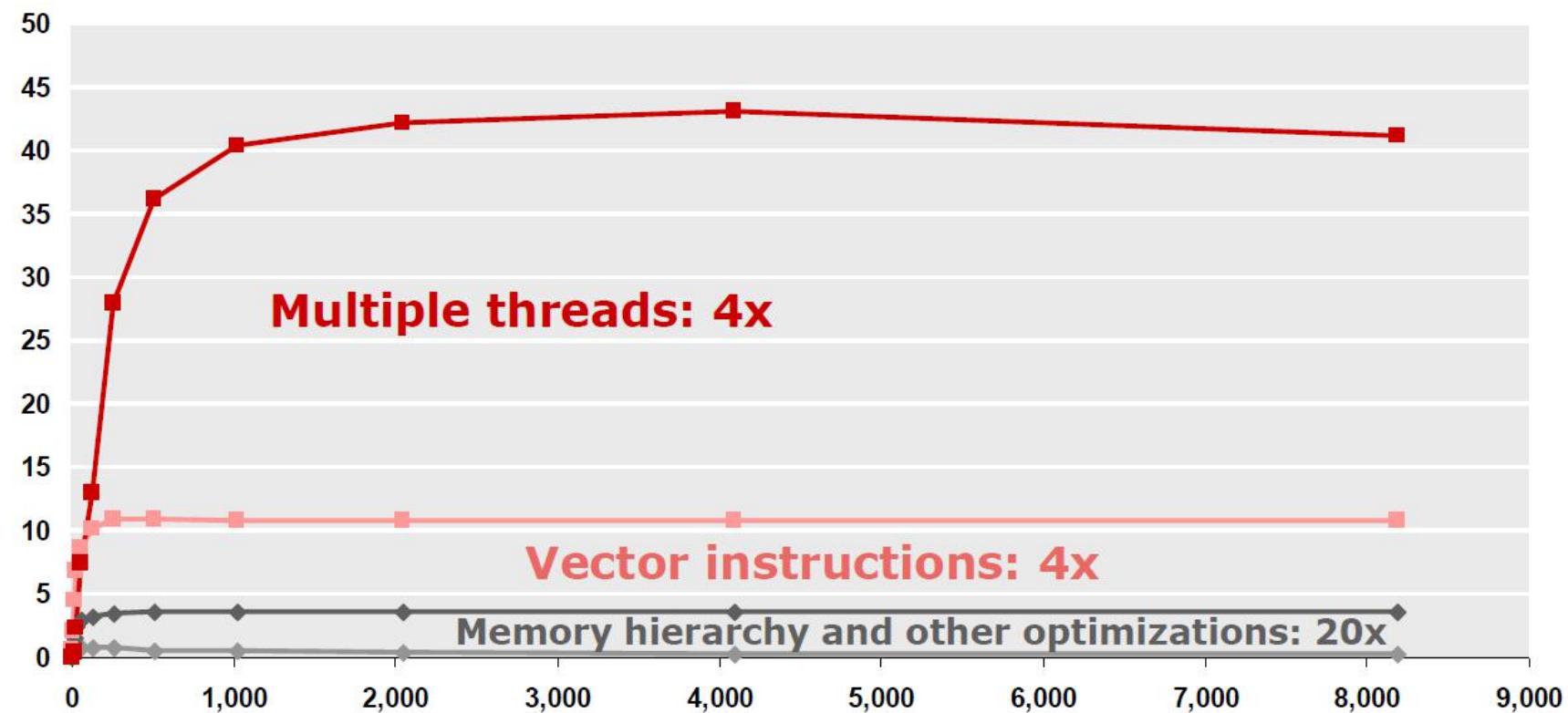
- Jim Keller

**James B. Keller**<sup>[1]</sup> (born 1958/1959)<sup>[2]</sup> is a [microprocessor](#) engineer best known for his work at [AMD](#) and [Apple](#). He was the lead architect of the [AMD K8](#) microarchitecture<sup>[3][4][5]</sup> (including the original [Athlon 64](#))<sup>[3][6][7]</sup> and was involved in designing the [Athlon](#) (K7)<sup>[5]</sup> and [Apple A4/A5](#) processors.<sup>[3][8][9][10]</sup> He was also the coauthor of the specifications for the [x86-64](#) instruction set<sup>[8][11]</sup> and [HyperTransport](#) interconnect.<sup>[3][11][12]</sup> From 2012 to 2015, he returned to AMD to work on the [AMD K12](#)<sup>[13]</sup> and [Zen](#) microarchitectures.<sup>[14][15]</sup>

# 一个例子：矩阵乘在计算机上的实现

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s



Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice

*Effect: less register spills, less L1/L2 cache misses, less TLB misses*

# 本单元内容提要

---

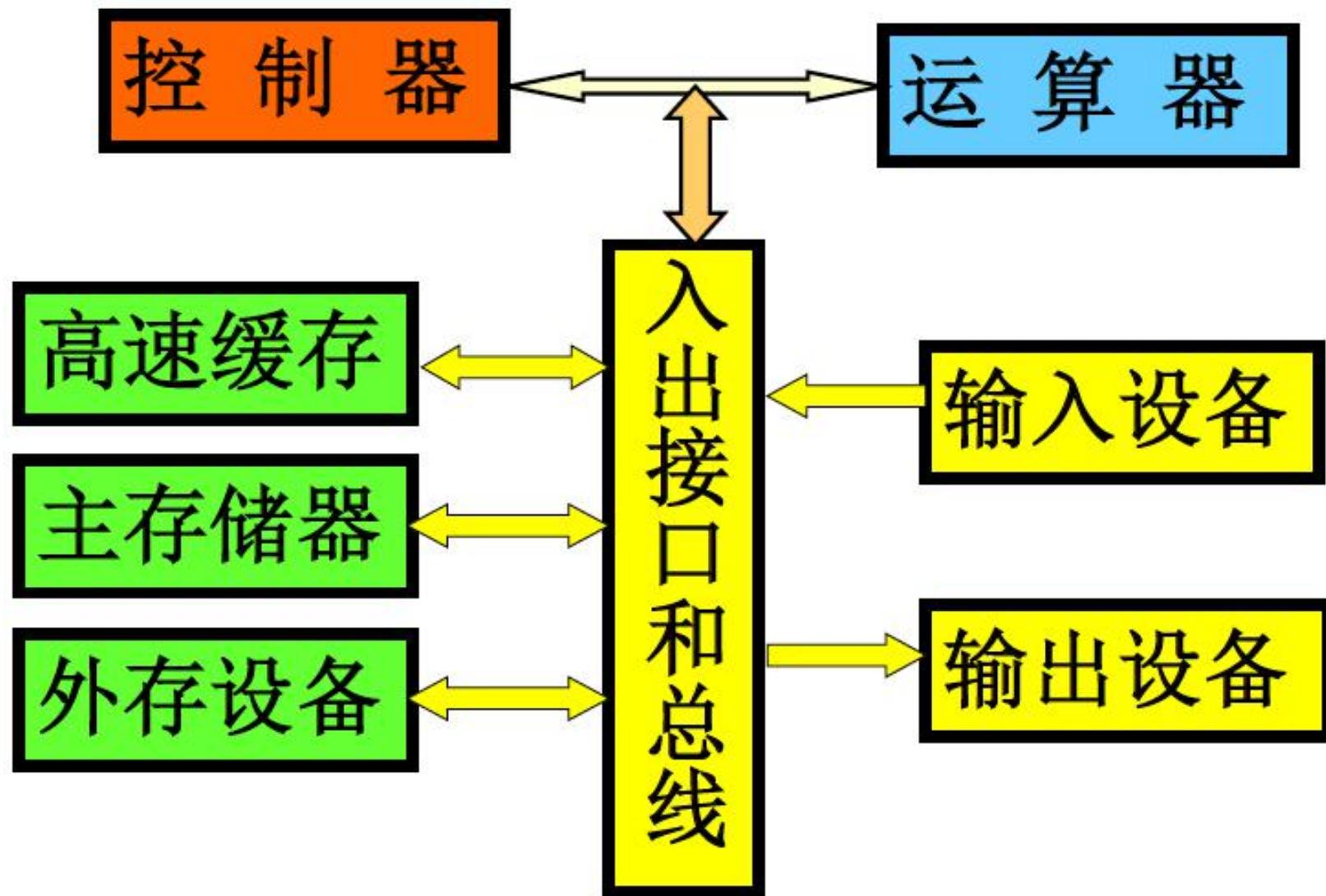
- 第一讲层次存储器系统概述及动态存储器
- 第二讲静态存储器及高速缓冲存储器
- 第三讲高速缓冲存储器的组成与运行原理
- 第四讲虚拟存储器的运行原理
- 第五讲磁表面存储设备的存储原理与组成
- 第六讲**RISC-V**系统异常处理和响应，虚拟内存

# 本讲概要

---

- 存储器系统功能
- 存储器系统的设计目标
- 需要解决的问题
- 层次存储器系统
- 动态存储器的组成与原理

# 计算机硬件系统



# 存储器地位和作用

---

- 存储程序使计算机走向通用。
- 计算机中用来存放程序和数据的部件，是Von Neumann结构计算机的重要组成，是计算机的中心。
- 程序和数据的特点
  - 源程序、汇编程序、机器语言程序
  - 各种类型的数据
  - 共同点：二进制数据

# 对存储介质的基本要求

---

- 能够有两个稳定状态来表示二进制中的“0”和“1”
- 容易识别
- 两个状态能方便地进行转换
- 几种常用的存储方式
  - 磁颗粒、半导体(电平/电容)、光

# 早期存储器

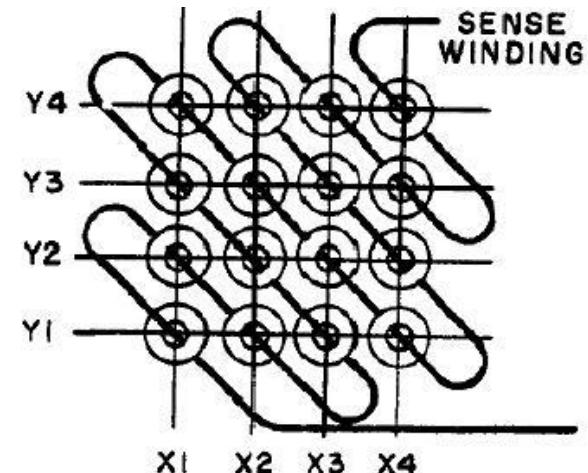
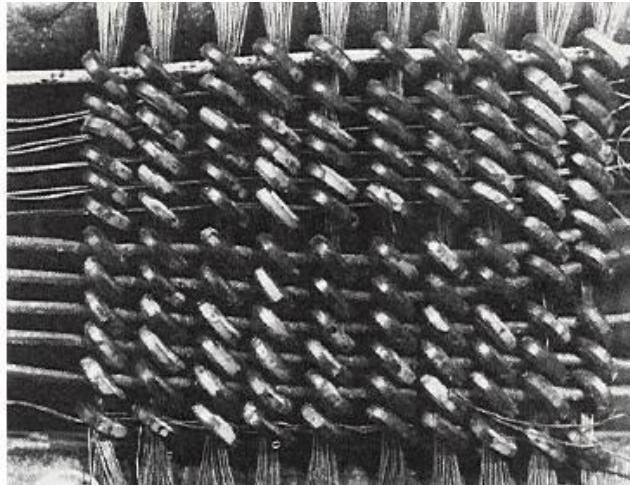
- 水银延迟线存储器
- EDSAC, 1949
- Maurice Wilkes
- 1967年 Turing 奖
- 存储原理
  - 水波



# 磁芯存储器

- 圆柱型陶瓷上涂磁粉
- 手工穿线，水手结
- 消磁后重写

16K!



# 半导体存储器

---

## □ 存储原理

- MOS管寄生电容
- 触发器

## □ 访问机制

- 随机访问

## □ 分类

- ROM、RAM
- SRAM、DRAM

# 按访问方式分类

---

## □ 随机访问存储器 (RAM)

- 访问时间与存放位置无关
- 半导体存储器

## □ 顺序访问存储器 (SAM)

- 按照存储位置依次访问
- 磁带存储器

## □ 直接访问存储器 (DAM)

- 随机+顺序
- 磁盘存储器

## □ 关联访问存储器 (CAM)

- 根据内容访问
- Cache和TLB

# 存储器系统设计目标

---

## □ 尽可能快的存取速度

- 应能基本满足CPU对数据的访问要求

## □ 尽可能大的存储空间

- 可以满足程序对存储空间的要求

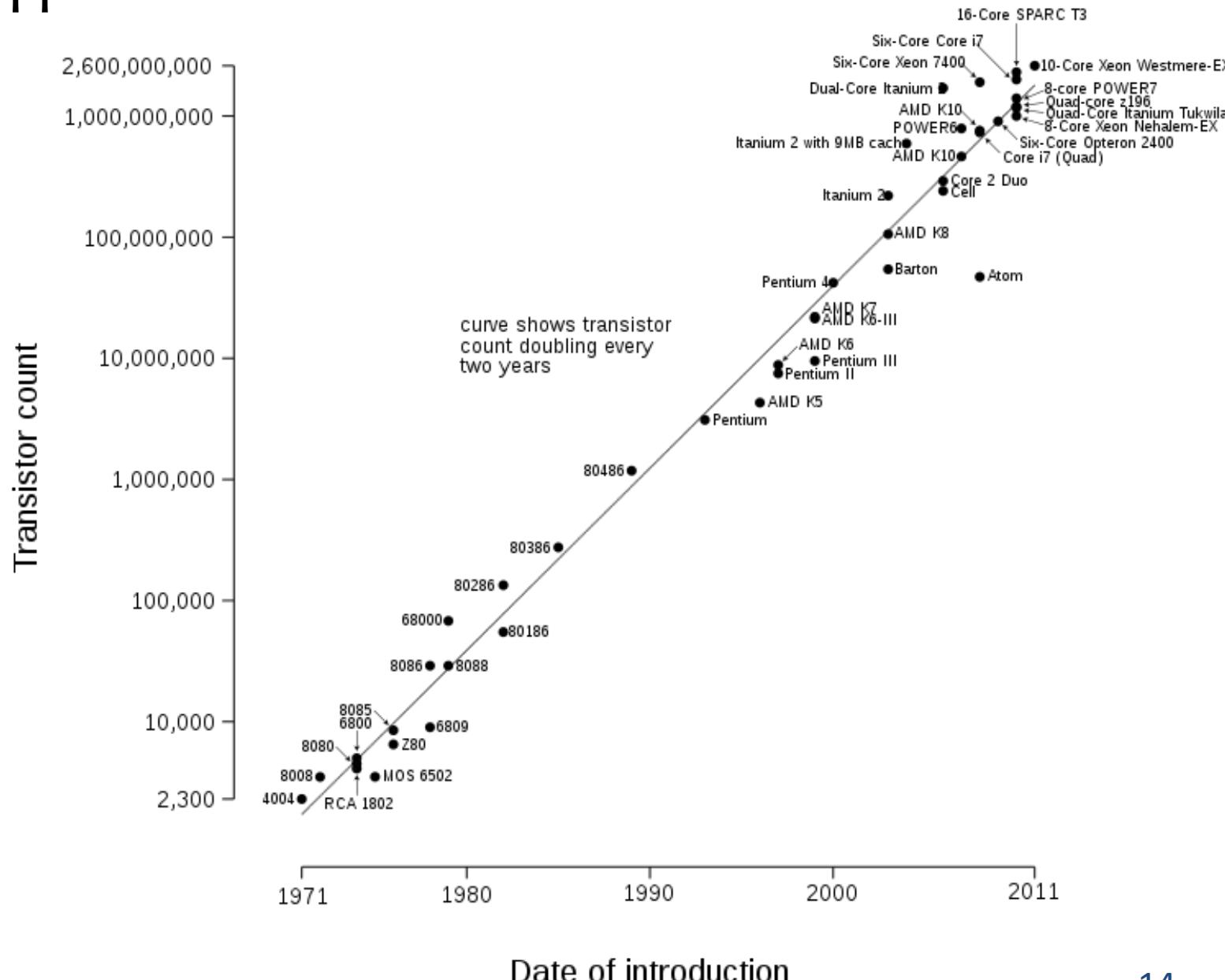
## □ 尽可能低的单位成本（价格/位）

- 用户能够承受的范围内

## □ 较高的可靠性

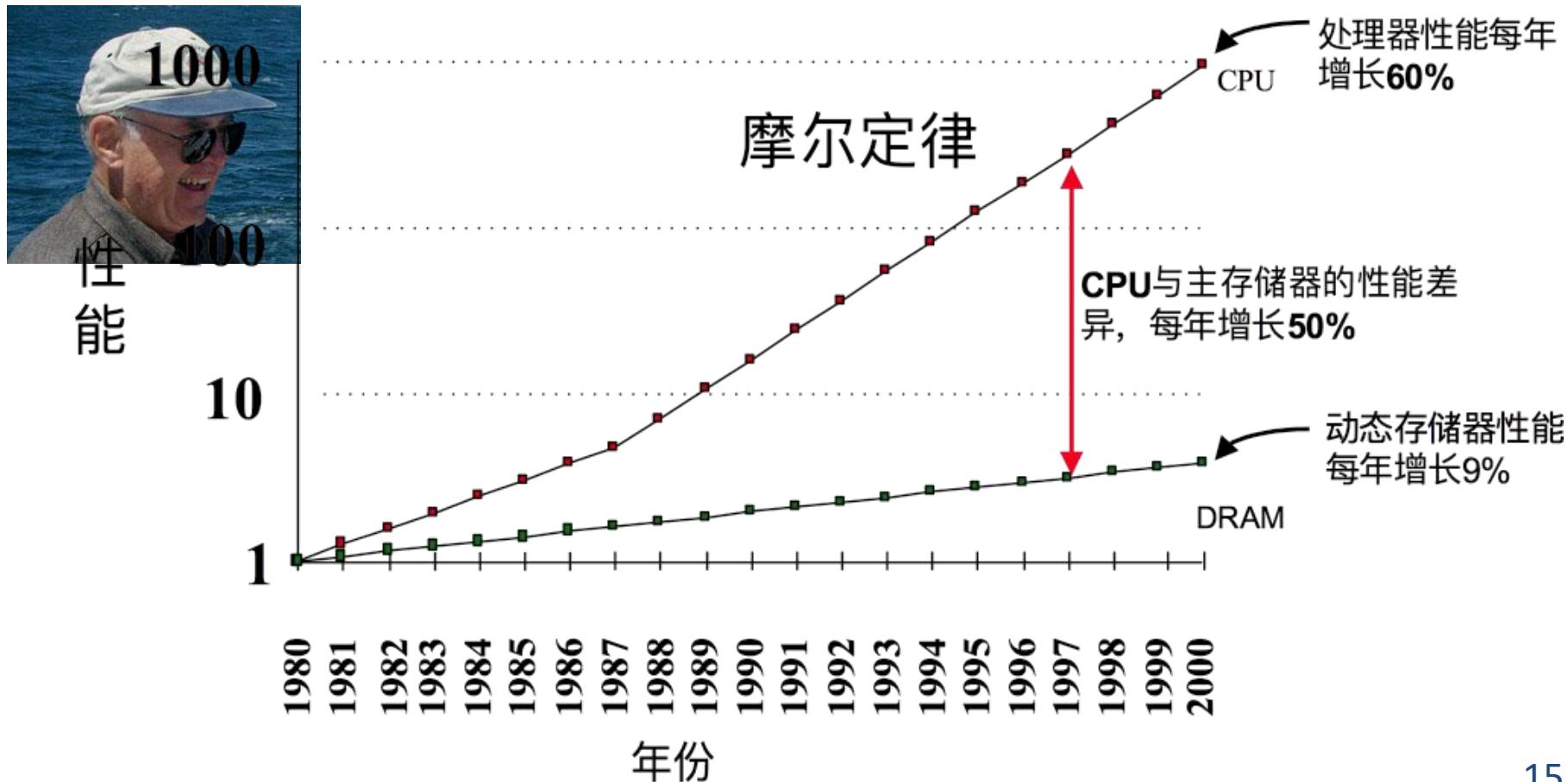
# 摩尔定律

Microprocessor Transistor Counts 1971-2011 & Moore's Law



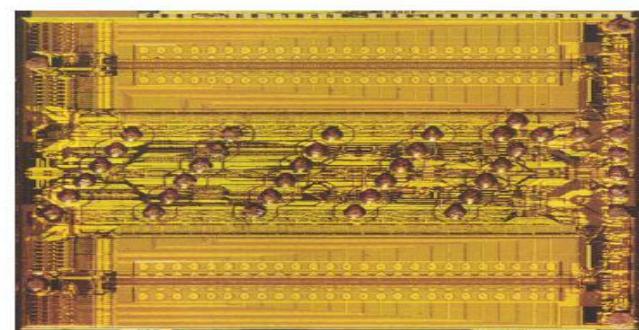
# Moore定律

- 1965年，Intel公司创始人之一Gordon Moore提出
- 芯片上集成的晶体管数量每18个月翻一番



# 摩尔定律

| 年代   | 容量       | 价格<br>(\$/MB) | 总访问时间<br>(新行 / 列) | 列访问时间<br>(现访问行) |
|------|----------|---------------|-------------------|-----------------|
| 1980 | 64 Kbit  | 1500          | 250 ns            | 150 ns          |
| 1983 | 256 Kbit | 500           | 185 ns            | 100 ns          |
| 1985 | 1 Mbit   | 200           | 135 ns            | 40 ns           |
| 1989 | 4 Mbit   | 50            | 110 ns            | 40 ns           |
| 1992 | 16 Mbit  | 15            | 90 ns             | 30 ns           |
| 1996 | 64 Mbit  | 10            | 60 ns             | 20 ns           |
| 1998 | 128 Mbit | 4             | 60 ns             | 10 ns           |
| 2000 | 256 Mbit | 1             | 55 ns             | 7 ns            |
| 2004 | 512 Mbit | 0.25          | 50 ns             | 5 ns            |
| 2007 | 1 Gbit   | 0.05          | 40 ns             | 1.25 ns         |



假定某台计算机的处理器工作在：

主频= 1GHz (机器周期为1 ns)

CPI = 1.1

50% 算逻指令, 30% 存取指令, 20% 转移指令

再假定其中10% 的存取指令会发生数据缺失，需要50个周期的延迟。

CPI = [填空1]

正常使用填空题需3.0以上版本雨课堂

作答

# 存储器对性能的影响

□ 假定某台计算机的处理器工作在：

- 主频= 1GHz (机器周期为1 ns)
- CPI = 1.1
- 50% 算逻指令, 30% 存取指令, 20% 转移指令

□ 再假定其中10% 的存取指令会发生数据缺失，需要50个周期的延迟。

- CPI = 理想CPI + 每条指令的平均延迟=  $1.1 + (0.30 \times 0.10 \times 50) = 1.1 \text{ cycle} + 1.5 \text{ cycle} = 2.6 \text{ CPI!}$
- 也就是说，处理器58 %的时间花在等待存储器给出数据上面！

□ 每1% 的指令的数据缺失将给CPI附加0.5个周期！

# 存储器设计目标

---

## □ 目标

- 大容量、高速度、低成本、高可靠性

## □ 目前现实

- 大容量存储器速度慢
- 快速存储器容量小

## □ 如何实现我们的目标呢？

- 层次存储器系统

# 问题

□ CPU clock rates ~0.33ns –2ns (3GHz-500MHz)

| Memory technology | Access time in nanosecs (ns) | Access time in cycles | \$ per GB in 2012 | Capacity |
|-------------------|------------------------------|-----------------------|-------------------|----------|
| SRAM (on chip)    | 0.5-2.5 ns                   | 1-3 cycles            | \$4k              | 256 KB   |
| SRAM (off chip)   | 1.5-30 ns                    | 5-15 cycles           | \$4k              | 32 MB    |
| DRAM              | 50-70 ns                     | 150-200 cycles        | \$10-\$20         | 8 GB     |
| SSD (Flash)       | 5k-50k ns                    | Tens of thousands     | \$0.75-\$1        | 512 GB   |
| Disk              | 5M-20M ns                    | Millions              | \$0.05-\$0.1      | 4 TB     |

# 层次存储器系统

---

## □ 高速度

- 静态存储器速度高
- 设置较小容量的高速缓冲存储器

## □ 大容量

- 动态存储器价格适中，速度适中
- 可作为主存储器

## □ 低成本

- 磁盘存储器价格低廉
- 作为辅助存储器，暂存CPU访问频率不高的数据和程序
- 作为虚拟存储器的载体

# 程序运行的局部性原理

---

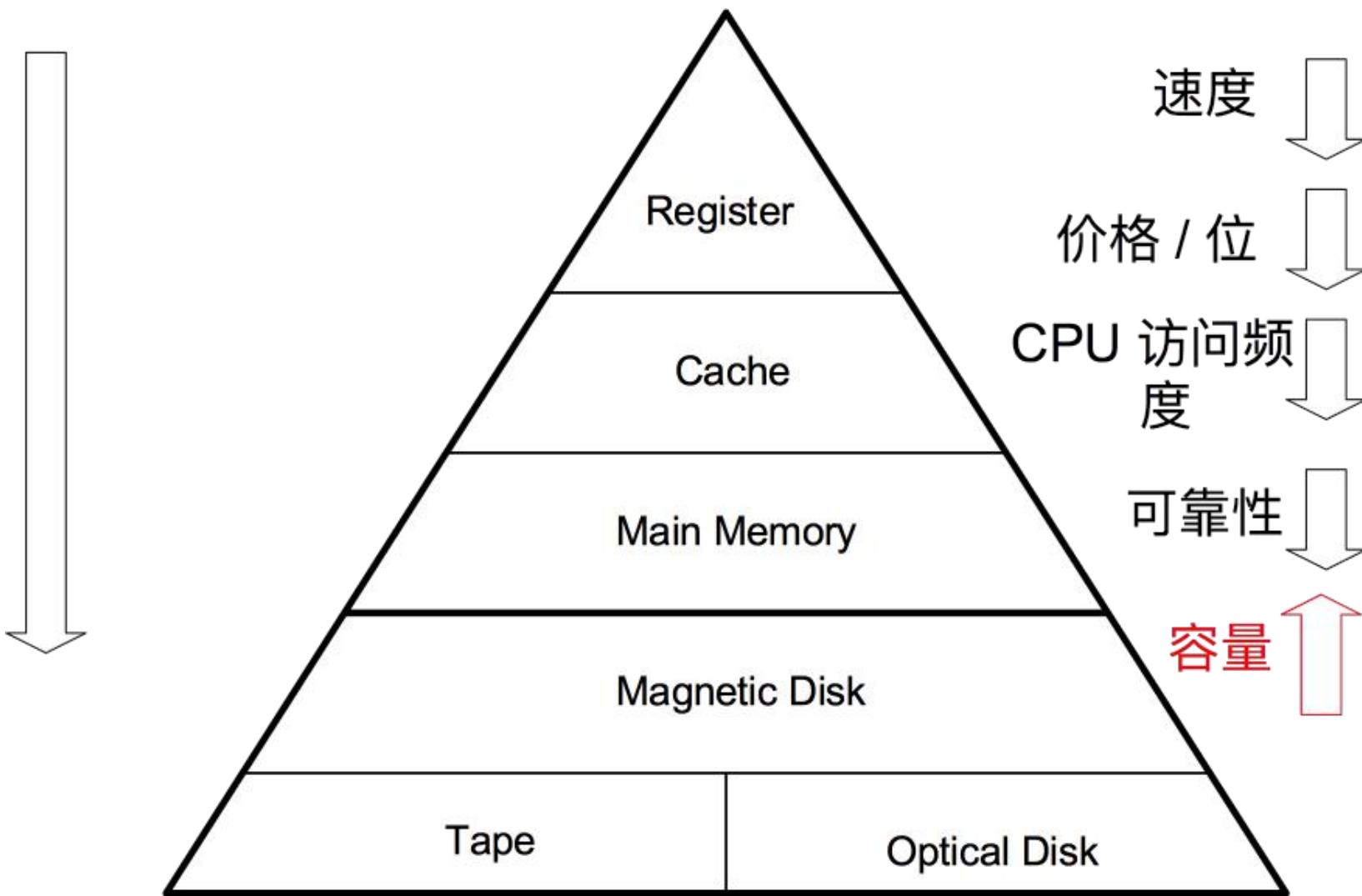
- 程序运行时的局部性原理表现在：
- 在一小段**时间**内，最近被访问过的程序和数据很可能再次被访问
- 在**空间**上这些被访问的程序和数据往往集中在一小片存储区
- 在访问**顺序**上，指令顺序执行比转移执行的可能性大(大约5:1 )
- 合理地把程序和数据分配在不同存储介质中

# 层次之间应满足的原则

---

- (1). **一致性原则：**处在不同层次存储器中的同一个信息应保持相同的值。
- (2). **包含性原则：**处在内层的信息一定被包含在其外层的存储器中，反之则不成立,即内层存储器中的全部信息，是其相邻外层存储器中一部分信息的复制品。

# 不同类型存储器比较



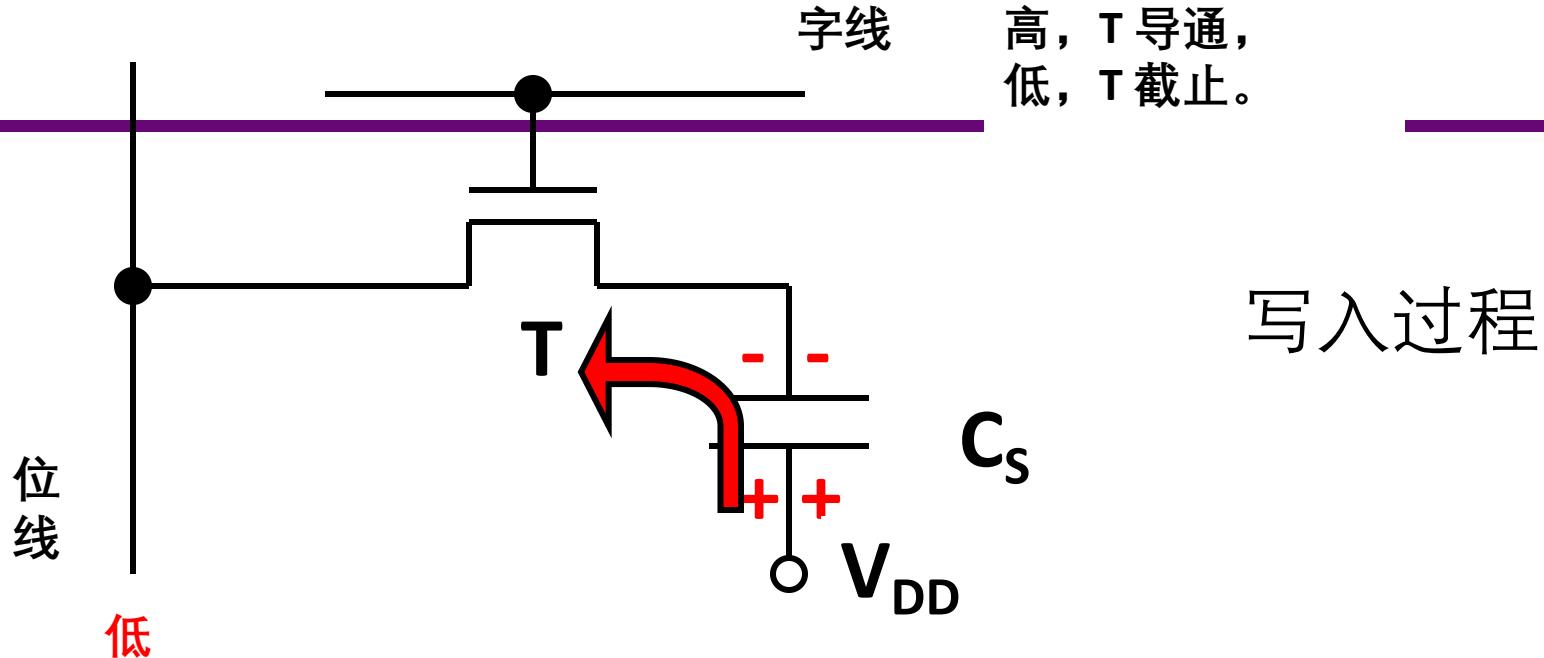
---

# **DRAM**

# 动态存储器的存储原理

---

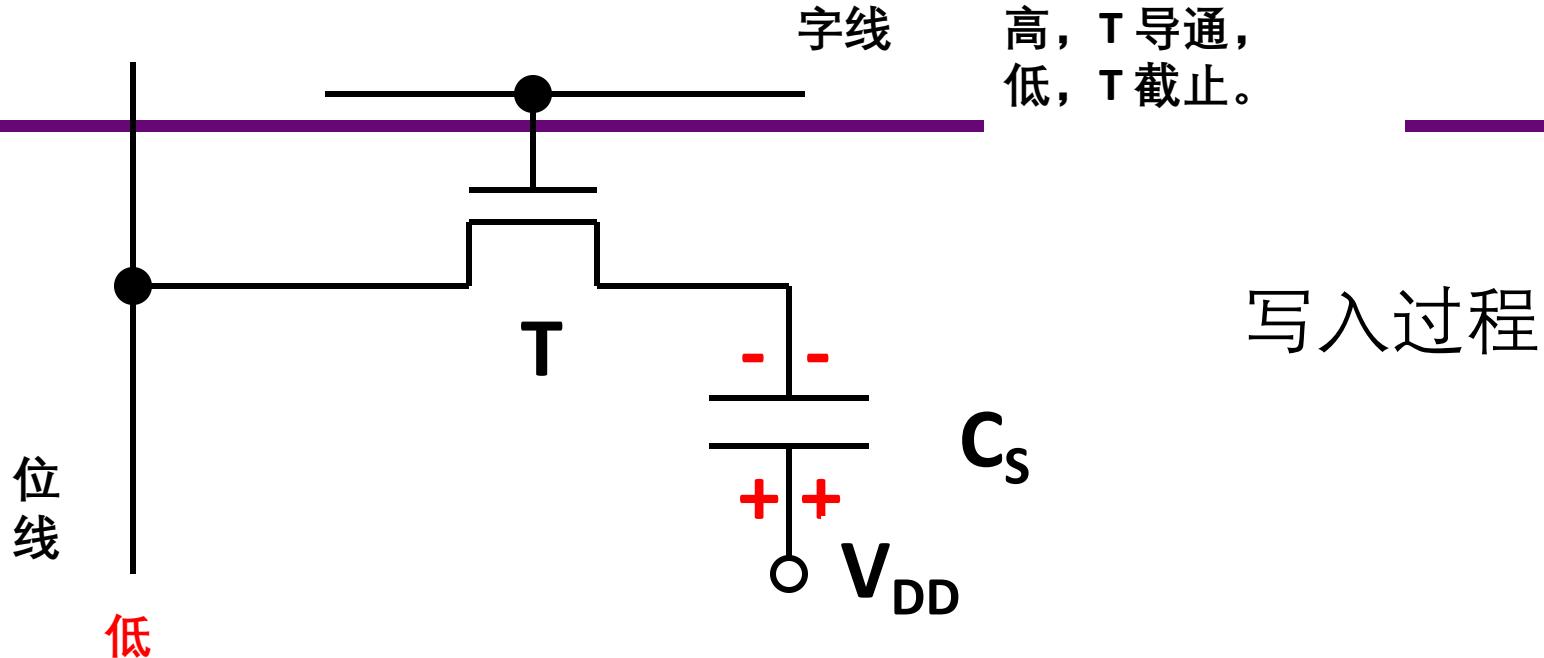
□ 动态存储器，是用金属氧化物半导体（MOS）的单个MOS管来存储一个二进制位（bit）信息的。信息被存储在MOS管T的源极的寄生电容CS中，例如，用CS中存储有电荷表示1，无电荷表示0。



写 1：使位线为低电平，

若  $C_s$  上无电荷，则  $V_{DD}$  向  $C_s$  充电；  
把 1 信号写入了电容  $C_s$  中。

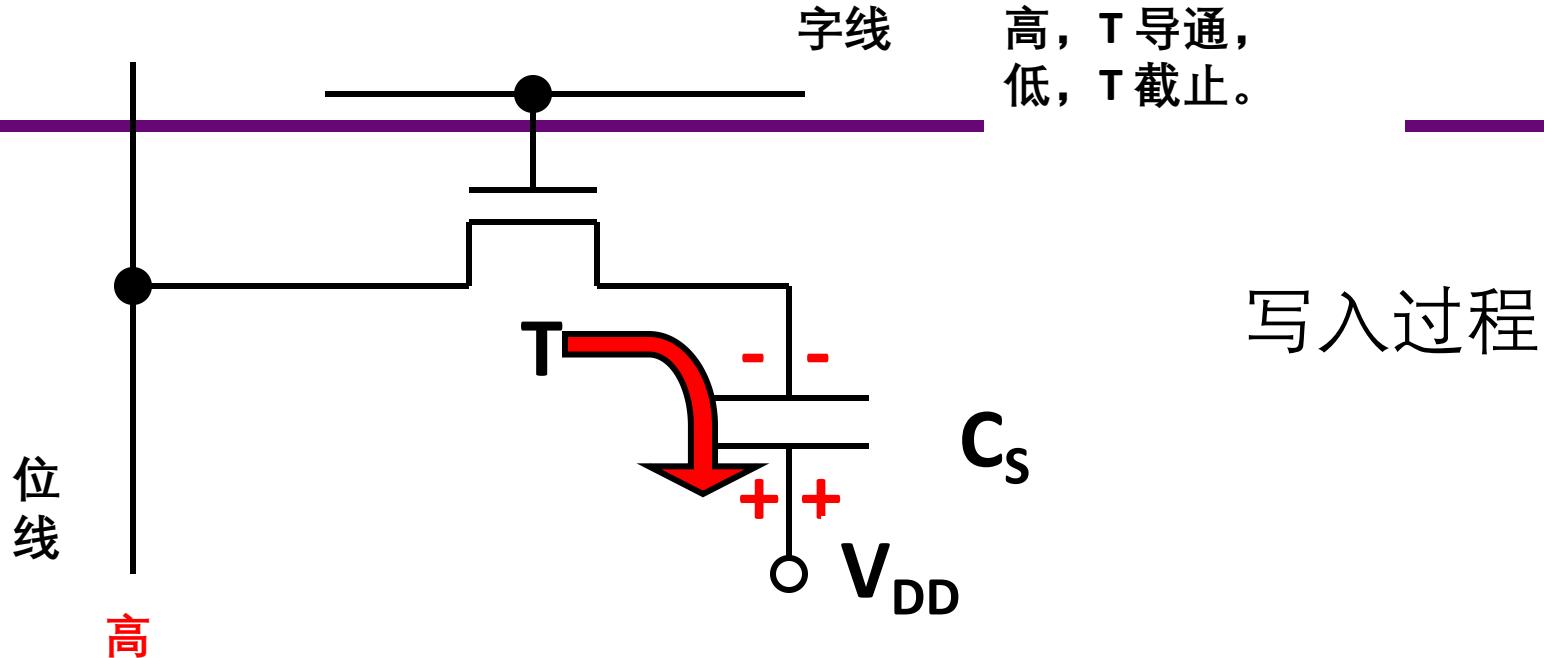
若  $C_s$  上有电荷，则  $C_s$  的电荷不变，  
保持原记忆的 1 信号不变。



写 1：使位线为低电平，

若  $C_s$  上无电荷，则  $V_{DD}$  向  $C_s$  充电；  
把 1 信号写入了电容  $C_s$  中。

若  $C_s$  上有电荷，则  $C_s$  的电荷不变，  
保持原有的内容 1 不变；

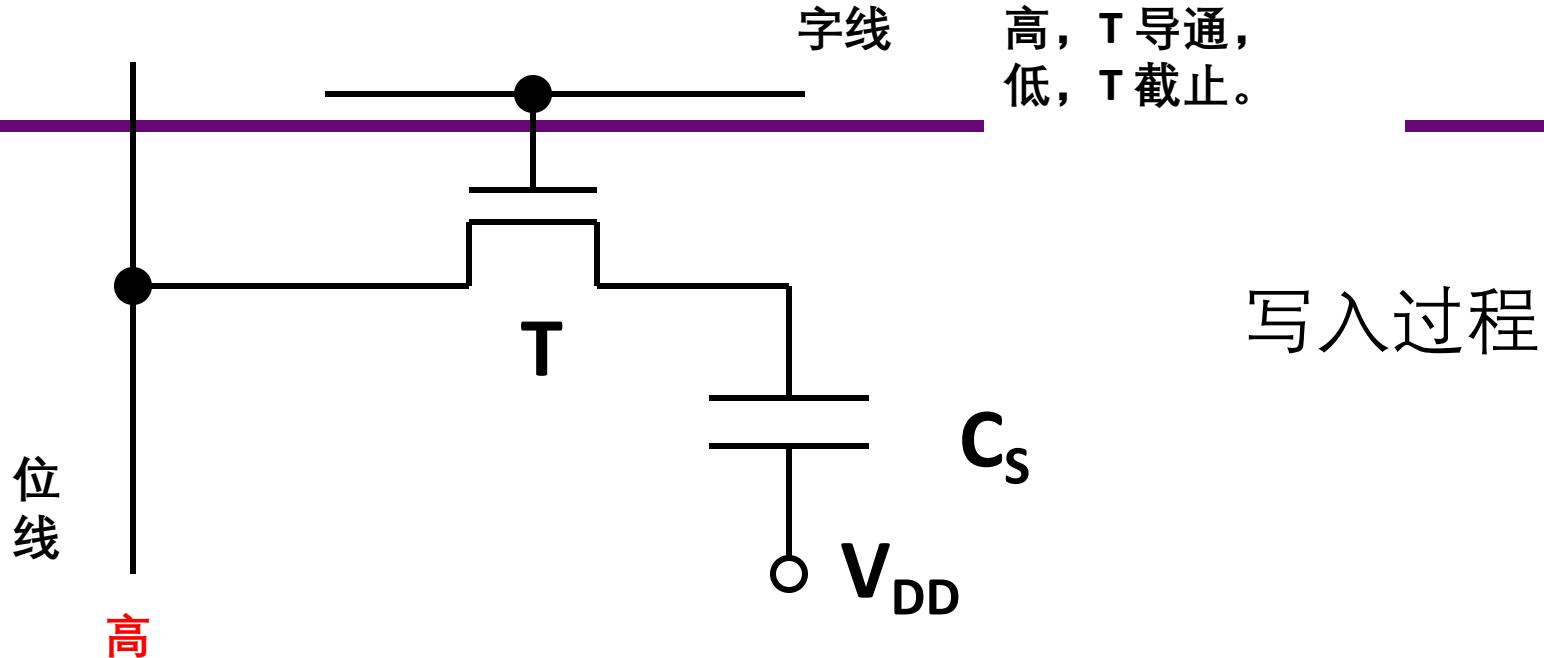


写 0：使位线为高电平，

若  $C_s$  上有电荷，则  $C_s$  通过 T 放电；

把 0 信号写入了电容  $C_s$  中。

若  $C_s$  上无电荷，则  $C_s$  无充放电动作，  
保持原记忆的 0 信号不变。

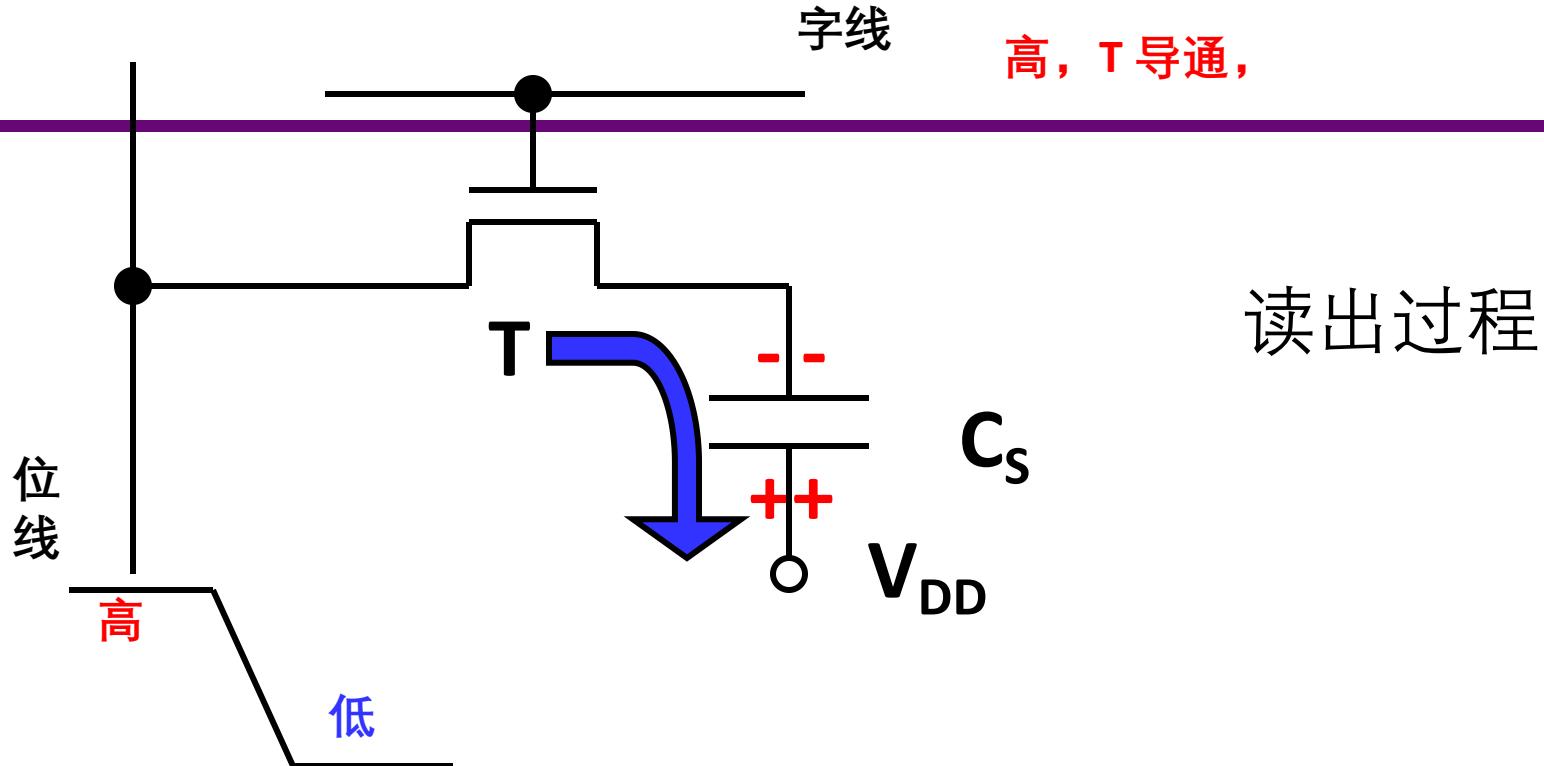


写 0：使位线为高电平，

若  $C_S$  上有电荷，则  $C_S$  通过 T 放电；

把 0 信号写入了电容  $C_S$  中。

若  $C_S$  上无电荷，则  $C_S$  无充放电动作，  
保持原记忆的 0 信号不变。



读操作：首先使位线充电至高电平，当字线来高电平时， $T$ 导通，

1. 若  $C_s$  上无电荷，则位线上无电位变化，读出为 0；
2. 若  $C_s$  上有电荷，则会放电，并使位线电位由高变低，

接在位线上的读出放大器会感知这种变化，读出为 1。

# 动态存储器的工作特点

---

## □ 破坏性读出

- 读出时被强制清零
- 预充电延迟

## □ 需定期刷新

- 集中刷新
  - 停止读写，逐行刷新
- 分散刷新
  - 定时周期性刷新

## □ 快速分页组织

**破坏性读出**：读操作后，被读单元的内容一定被清为零，必须把刚读出的内容立即写回去，通常称其为预充电延迟，它影响存储器的工作频率，在结束预充电前不能开始下一次读。

**要定期刷新**：在不进行读写操作时，DRAM 存储器的各单元处于断路状态，由于漏电的存在，保存在电容 $C_s$ 上的电荷会慢慢地漏掉，为此必须定时予以补充，通常称其为刷新操作。刷新不是按字处理，而是每次刷新一行，即为连接在同一行上所有存储单元的电容补充一次能量。刷新有**两种常用方式**：

**集中刷新**，停止内存读写操作，逐行将所有各行刷新一遍；

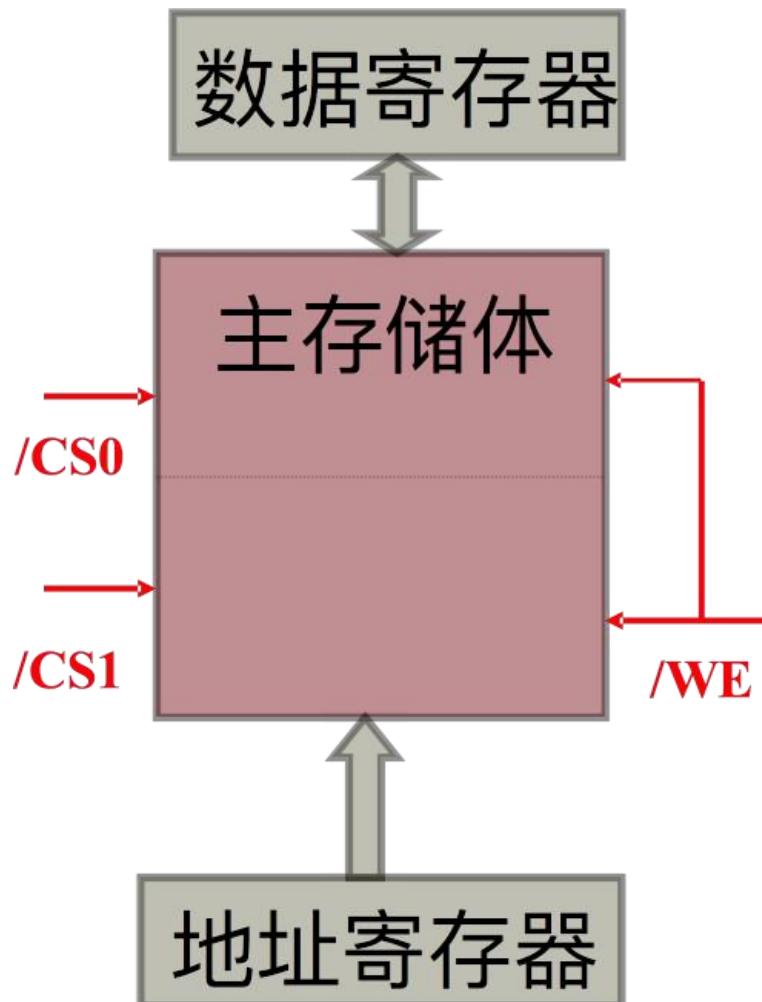
**分散刷新**，每一次内存读写后，刷新一行，各行轮流进行。

或在规定的期间内，如 2 ms，能轮流把所有各行刷新一遍。

**快速分页组织的存储器：**

行、列地址要分两次给出，但连续地读写用到相同的行地址时，也可以在前一次将行地址锁存，之后仅送列地址，以节省送地址的时间，支持这种运行方式的被称为快速分页组织的存储器。

# 主存储器的读写过程



## 口读过程:

- 给出地址
- 给出片选与读命令
- 保存读出内容

## 口写过程:

- 给出地址
- 给出片选与数据
- 给出写命令

# DRAM Bank Operation

Access Address:

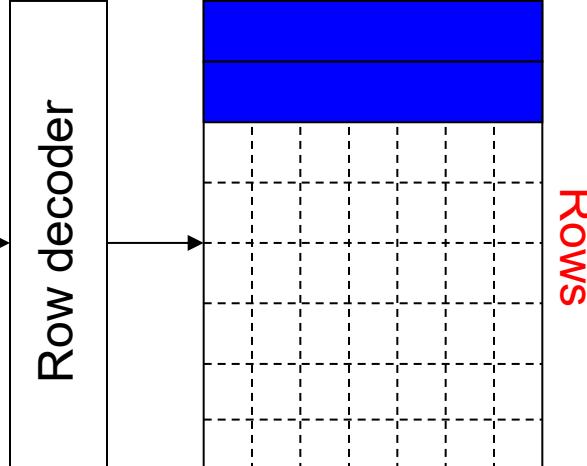
(Row 0, Column 0)

(Row 0, Column 1)

(Row 0, Column 85)

(Row 1, Column 0)

Row address 0

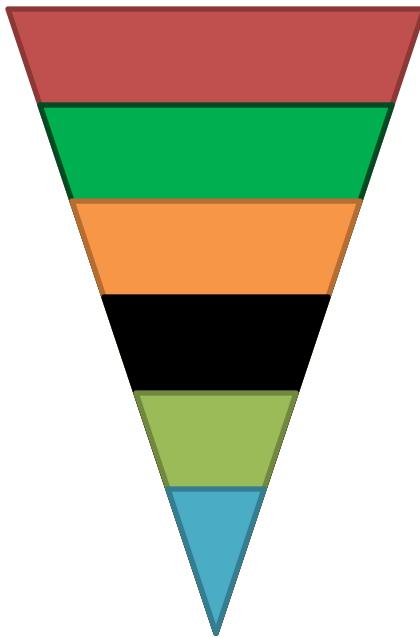


Column address 85

Data

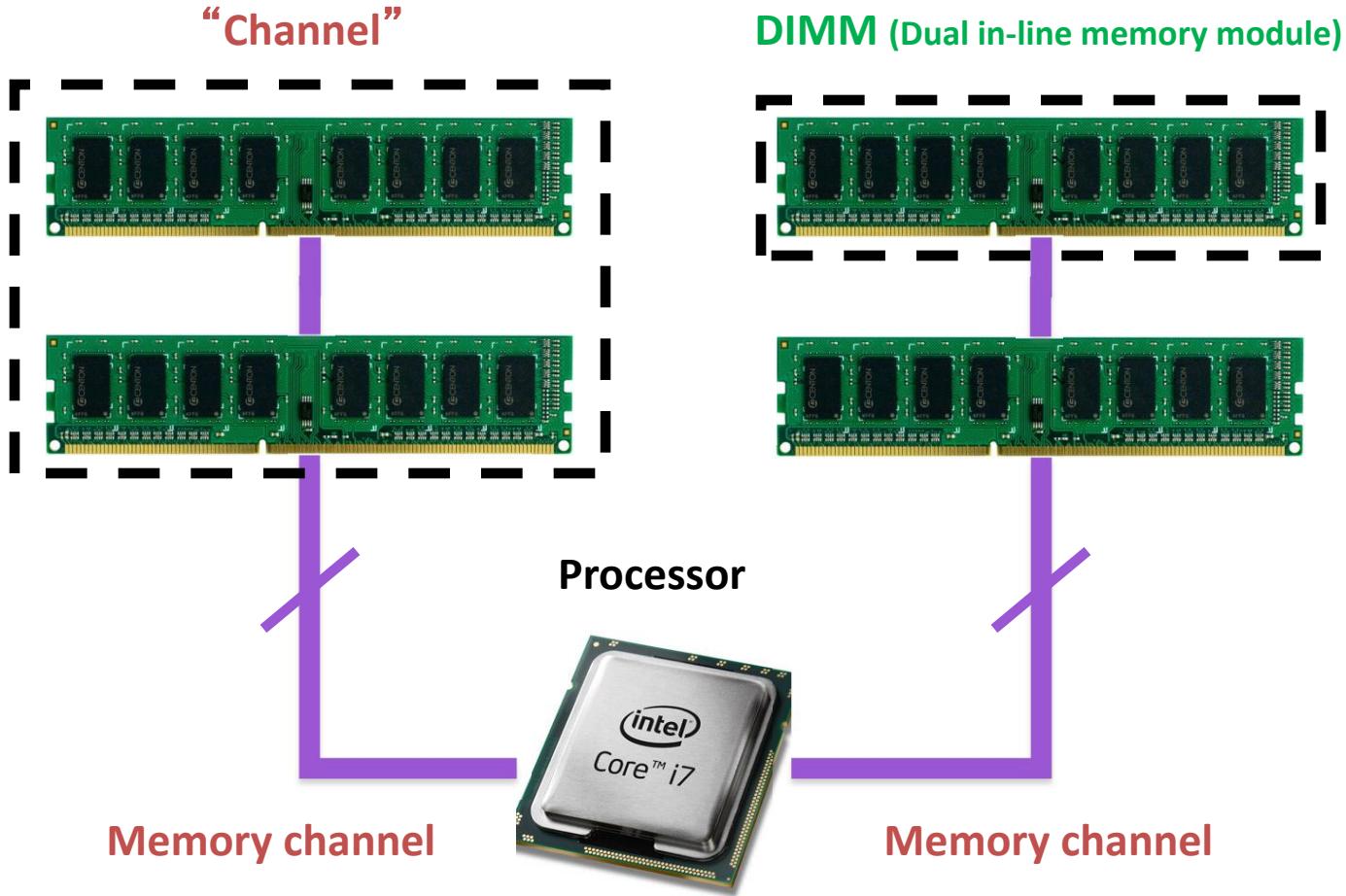
# DRAM Subsystem Organization

- ❑ Channel
- ❑ DIMM
- ❑ Rank
- ❑ Chip
- ❑ Bank
- ❑ Row/Column



Top-Down View

# The DRAM subsystem



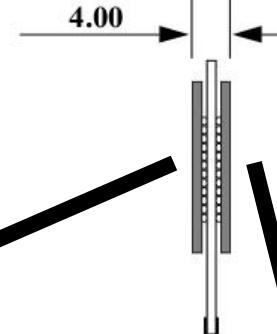
# Breaking down a DIMM

DIMM (Dual in-line memory module)

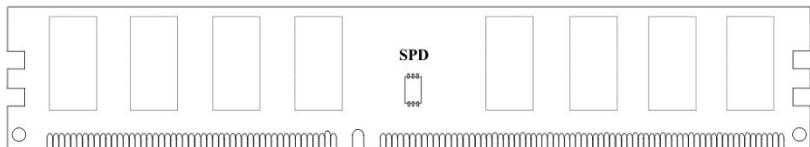


Side view

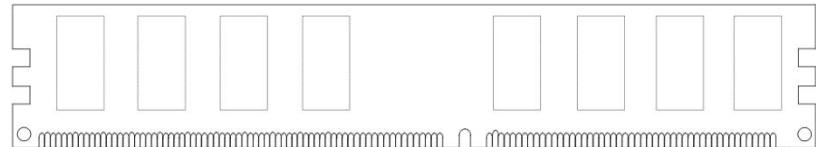
SIDE



Front of DIMM



Back of DIMM

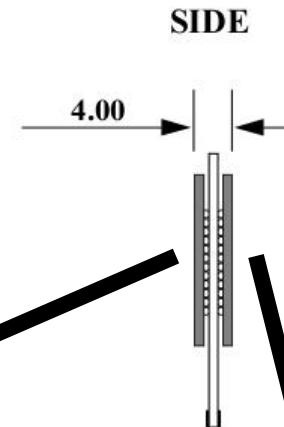


# Breaking down a DIMM

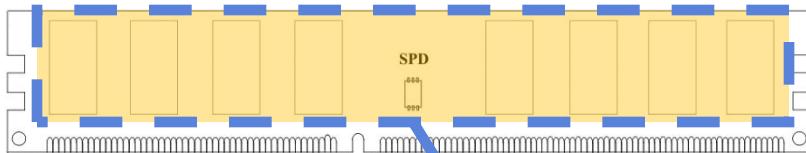
DIMM (Dual in-line memory module)



Side view

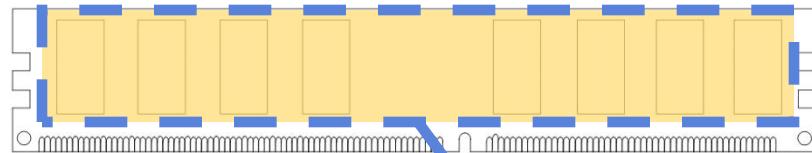


Front of DIMM



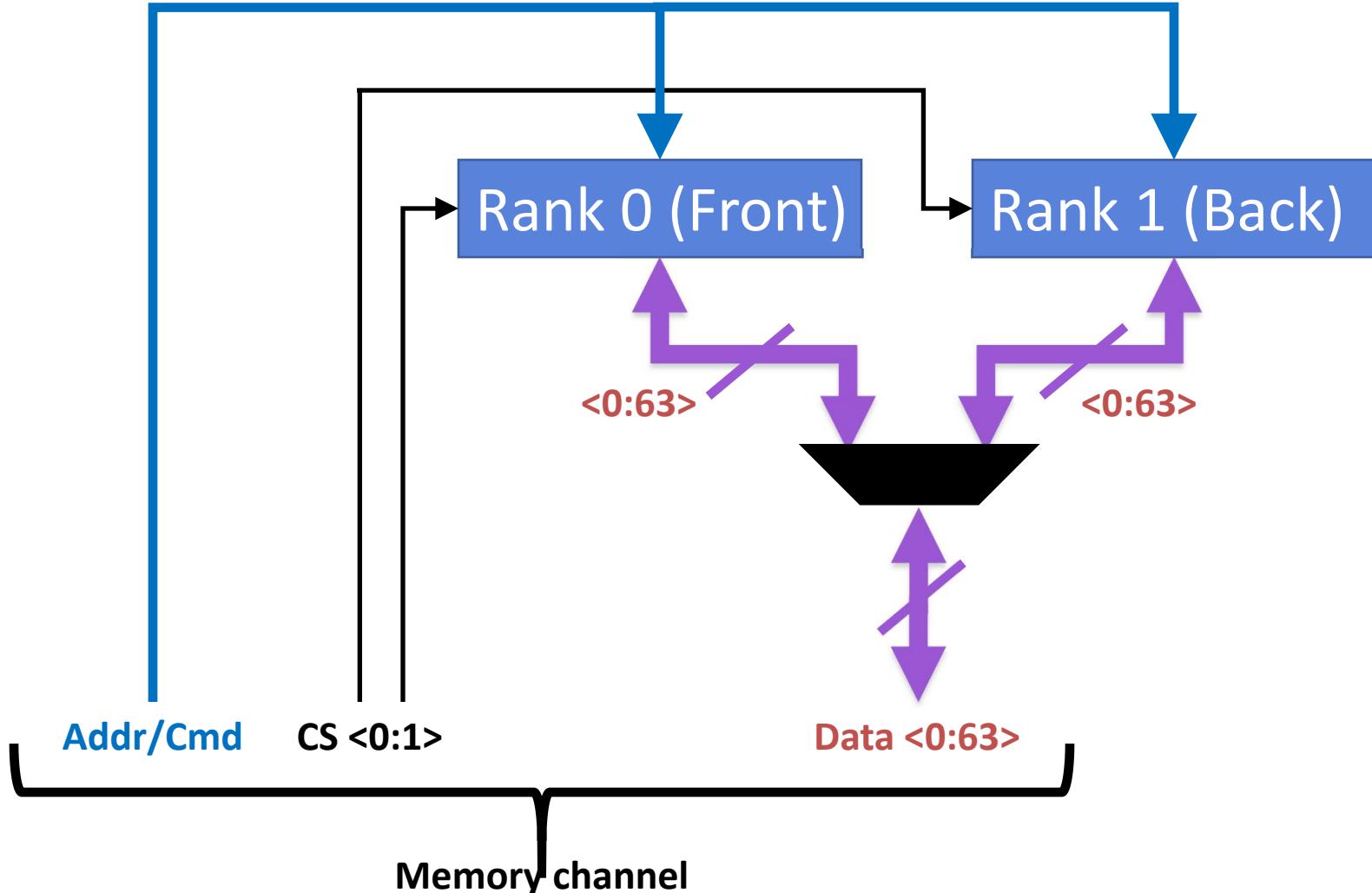
Rank 0: collection of 8 chips

Back of DIMM

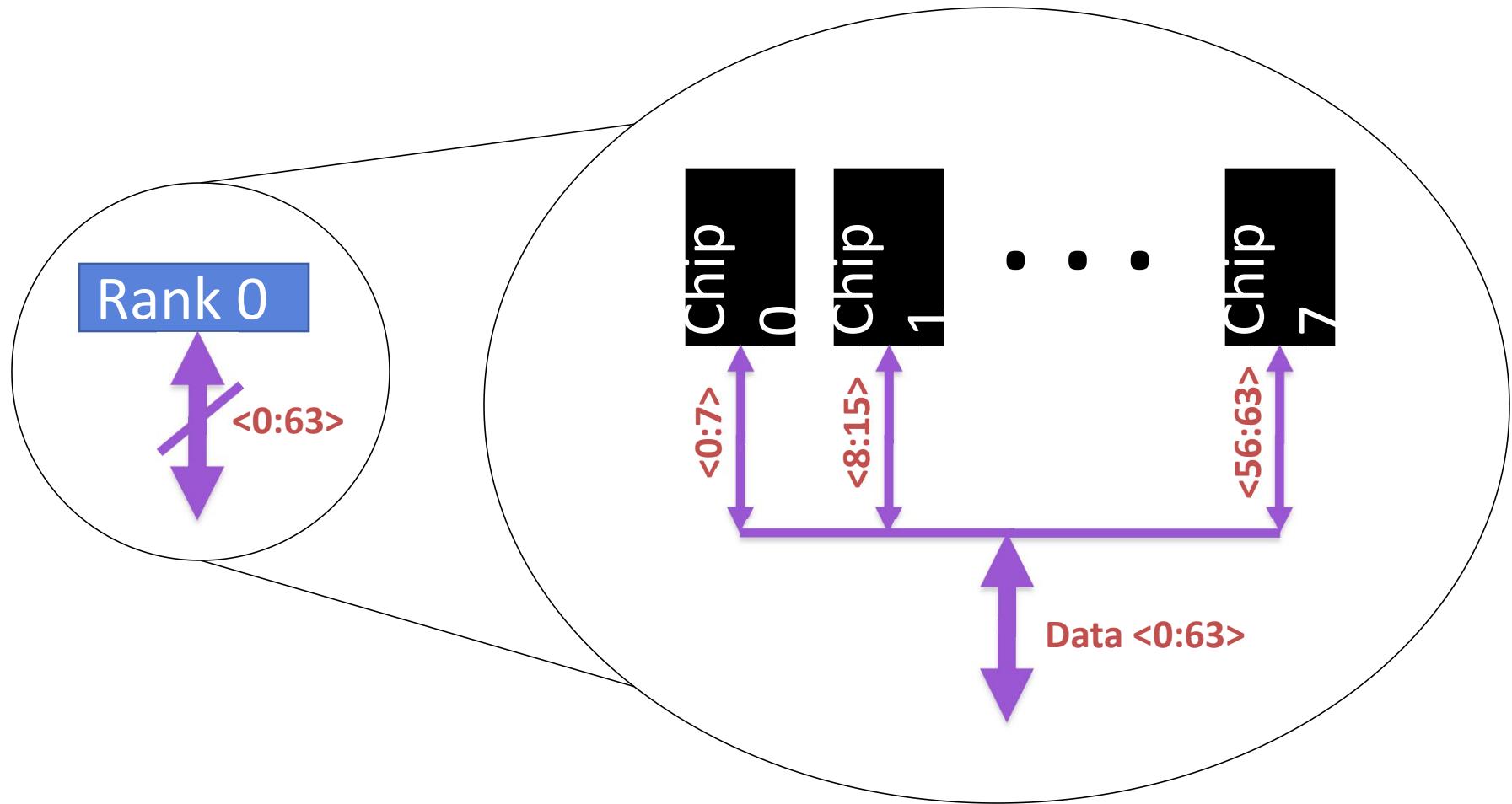


Rank 1

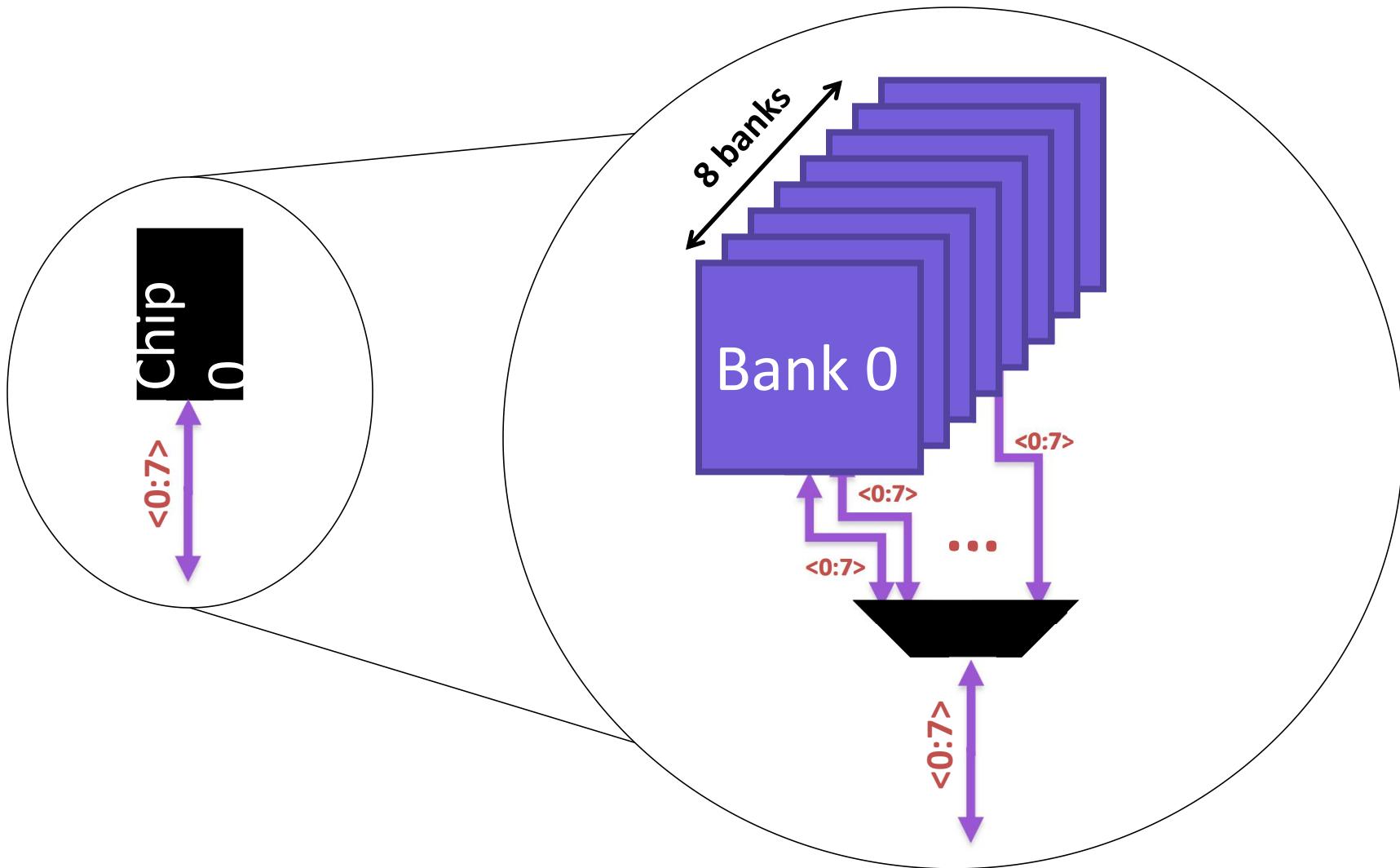
# Rank



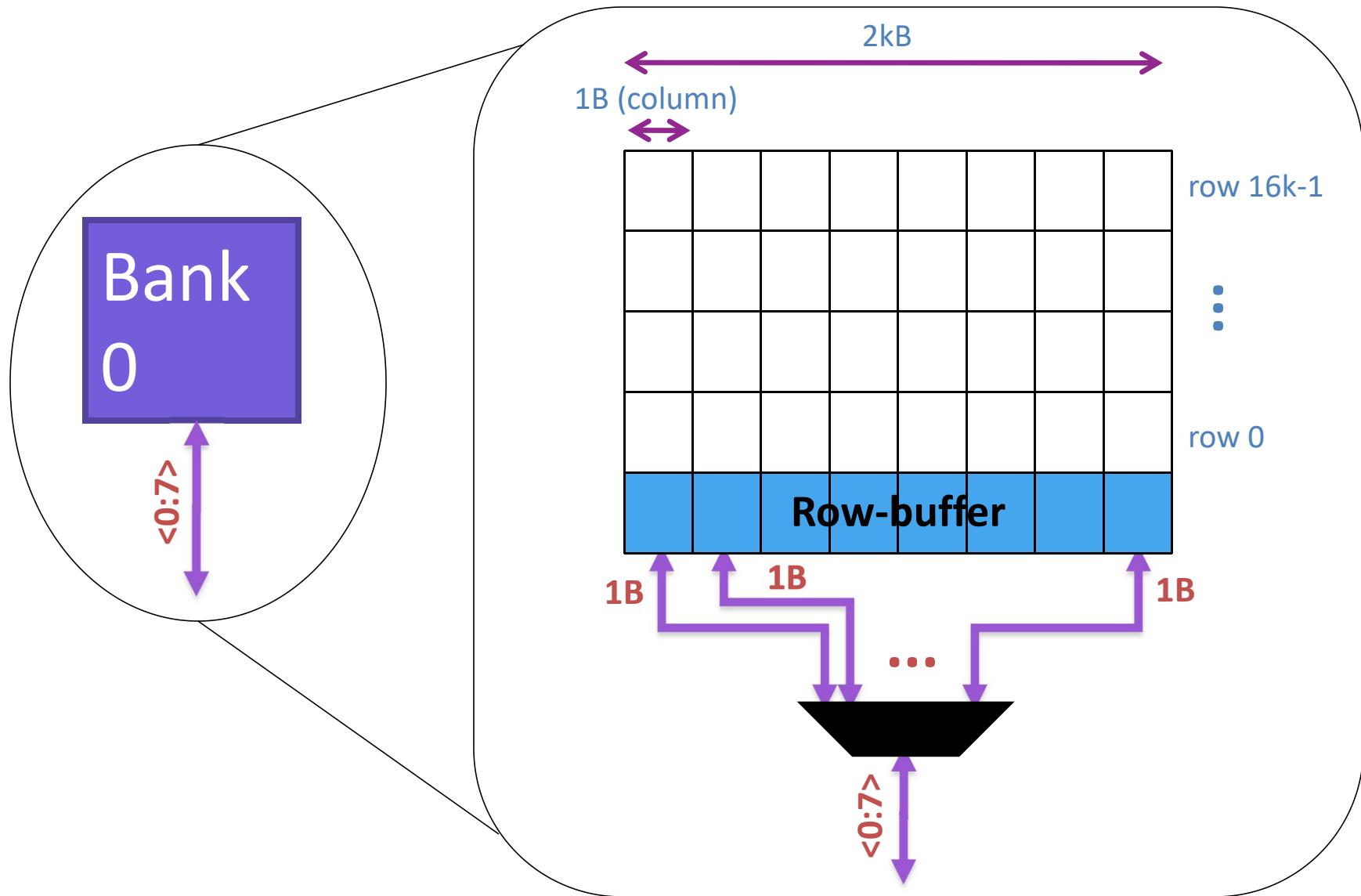
# Breaking down a Rank



# Breaking down a Chip

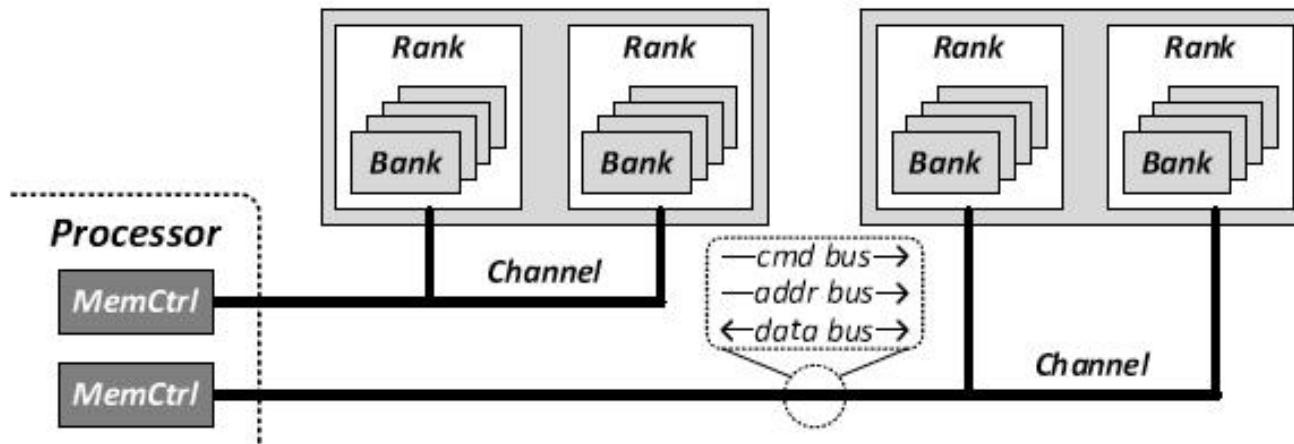


# Breaking down a Bank



# 主存储器的作用和连接

- 存储正处在运行中的程序和数据(或一部分) 的部件，通过地址数据控制三类总线与CPU、与其它部件连通



# 地址总线

---

□ 地址总线用于选择主存储器的一个存储单元（字或字节），其位数决定了能够访问的存储单元的最大数目，称为最大可寻址空间。例如，当按字节寻址时，20位的地址可以访问1MB的存储空间，32位的地址可以访问4GB的存储空间。

# 数据总线

□ 数据总线用于在计算机各功能部件之间传送数据，数据总线的位数（总线的宽度）与总线时钟频率的乘积，与该总线所支持的最高数据吞吐（输入/输出）能力成正比。

| Names                | Memory clock | I/O bus clock | Transfer rate | Theoretical bandwidth |
|----------------------|--------------|---------------|---------------|-----------------------|
| DDR-200, PC-1600     | 100 MHz      | 100 MHz       | 200 MT/s      | 1.6 GB/s              |
| DDR-400, PC-3200     | 200 MHz      | 200 MHz       | 400 MT/s      | 3.2 GB/s              |
| DDR2-800, PC2-6400   | 200 MHz      | 400 MHz       | 800 MT/s      | 6.4 GB/s              |
| DDR3-1600, PC3-12800 | 200 MHz      | 800 MHz       | 1600 MT/s     | 12.8 GB/s             |
| DDR4-2400, PC4-19200 | 300 MHz      | 1200 MHz      | 2400 MT/s     | 19.2 GB/s             |
| DDR4-3200, PC4-25600 | 400 MHz      | 1600 MHz      | 3200 MT/s     | 25.6 GB/s             |
| DDR5-4800, PC5-38400 | 300 MHz      | 2400 MHz      | 4800 MT/s     | 38.4 GB/s             |
| DDR5-6400, PC5-51200 | 400 MHz      | 3200 MHz      | 6400 MT/s     | 51.2 GB/s             |

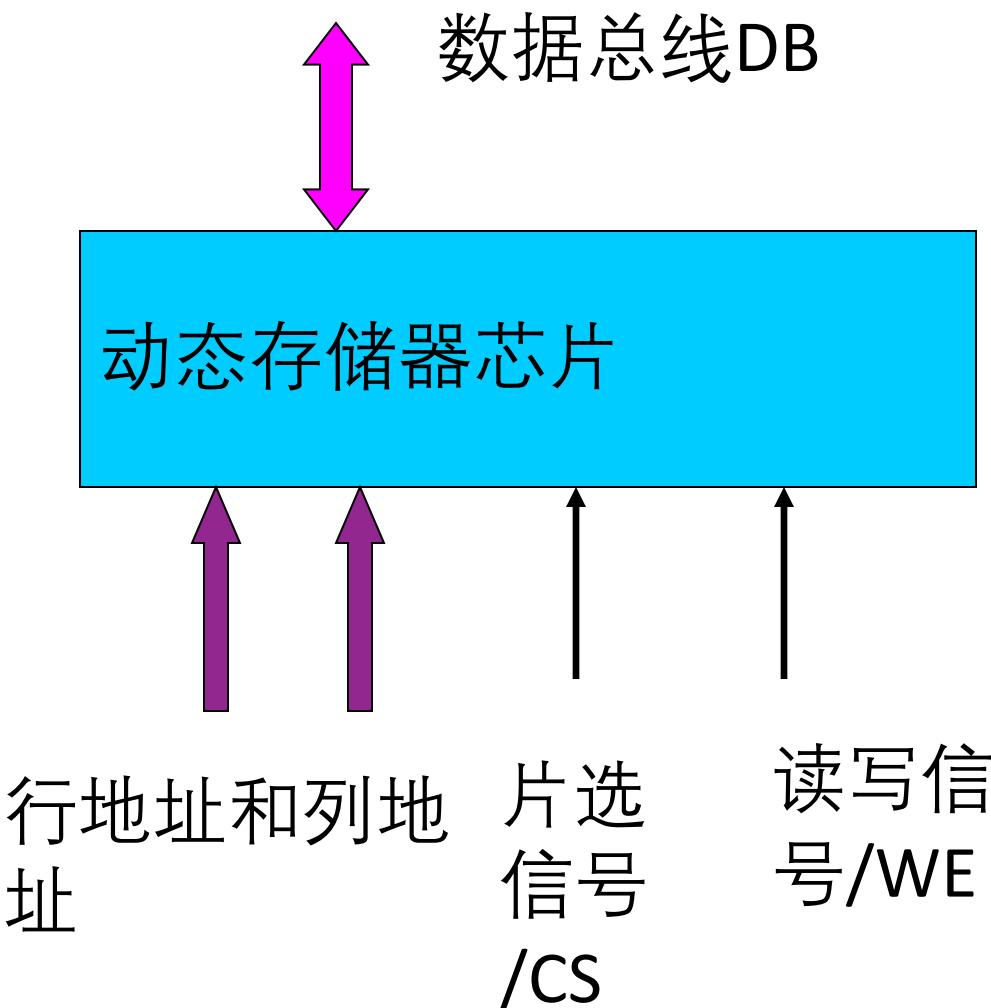
# 控制总线

---

□ 控制总线用于指明总线的工作周期类型和本次入/出完成的时刻。总线的工作周期可以包括主存储器读周期、主存储器写周期、I/O设备读周期、I/O设备写周期，即用不同的总线周期来区分要用哪个部件（主存或I/O设备）和操作的性质（读或写）；还有直接存储器访问（DMA）总线周期等。

# 动态存储器读写过程

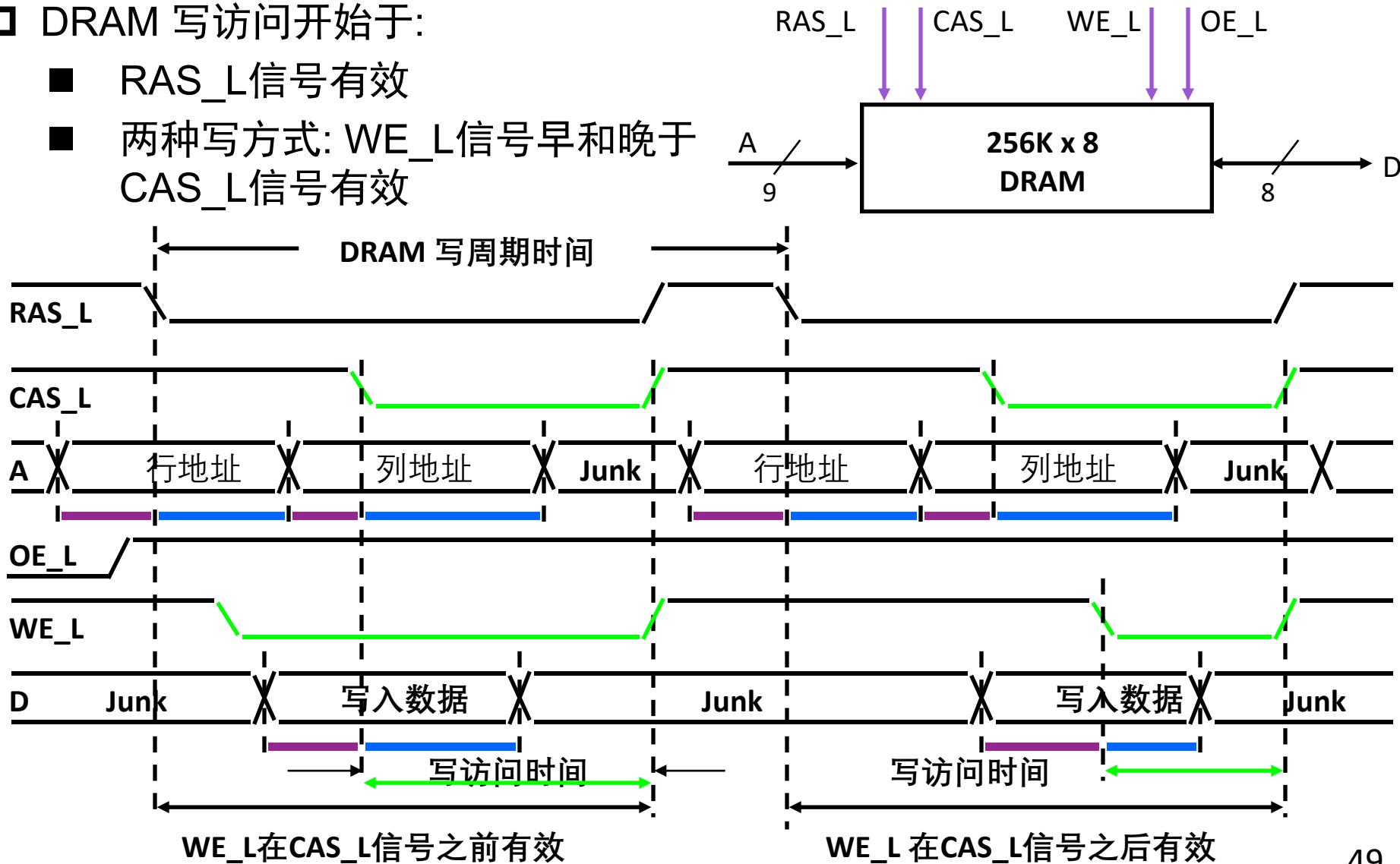
动态存储器集成度高，存储容量大，为节约管脚数，地址分为行地址和列地址



# DRAM 写时序

□ DRAM 写访问开始于:

- RAS\_L信号有效
- 两种写方式: WE\_L信号早和晚于 CAS\_L信号有效



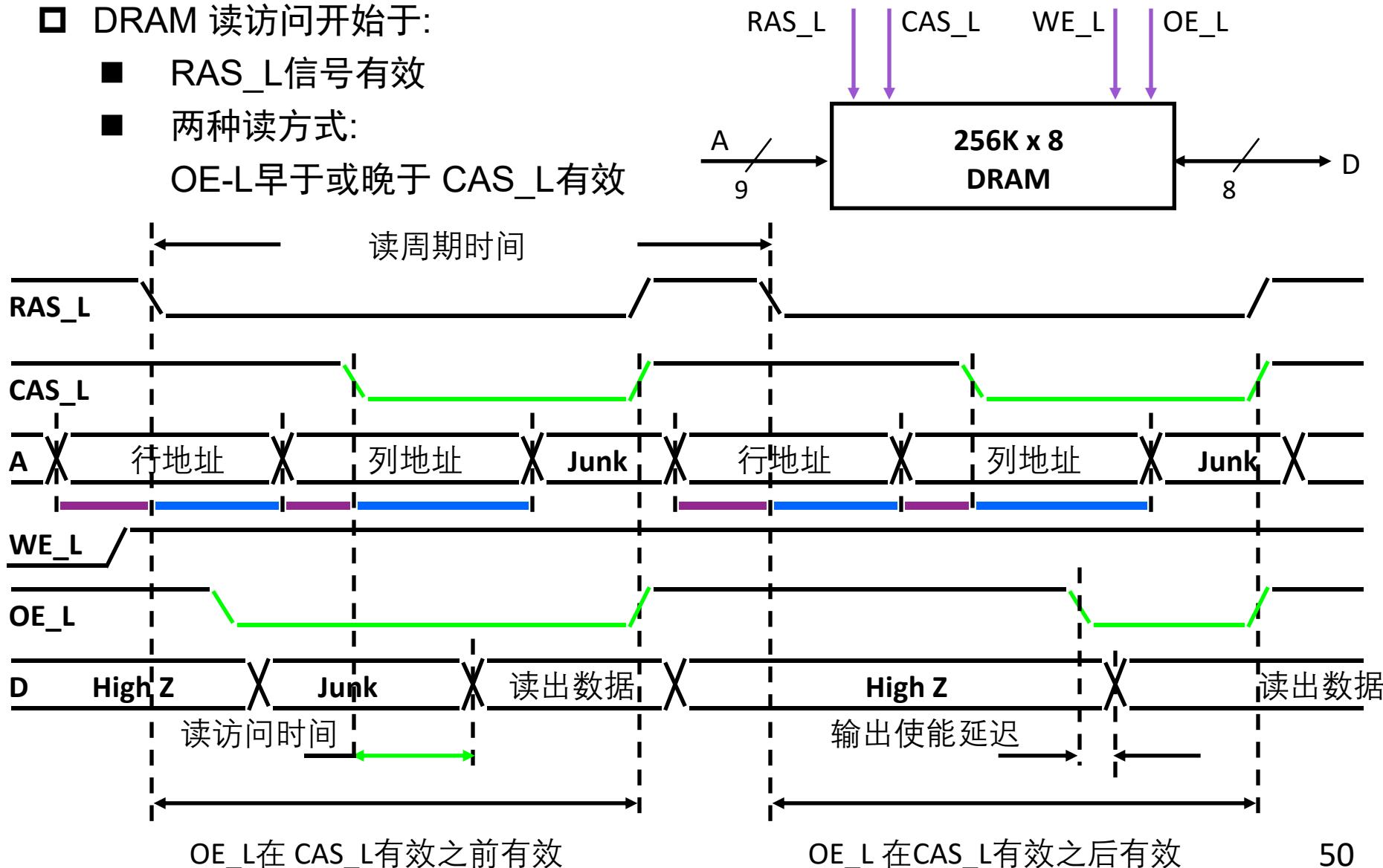
# DRAM 读时序

□ DRAM 读访问开始于:

- RAS\_L信号有效

- 两种读方式:

- OE\_L早于或晚于 CAS\_L有效



# 小结

## □ 程序的局部性原理：

- 时间局部性：最近被访问过的程序和数据很可能再次被访问
- 空间局部性：**CPU**很可能访问最近被访问过的地址单元附近的地址单元。

## □ 利用程序的局部性原理：

- 使用尽可能大容量的廉价、低速存储器存放程序和数据。
- 使用高速存储器来满足CPU对速度的要求。

## □ 动态存储器DRAM

- 电容充放电来存储数据
- 集成度高、容量大、能耗低、速度慢

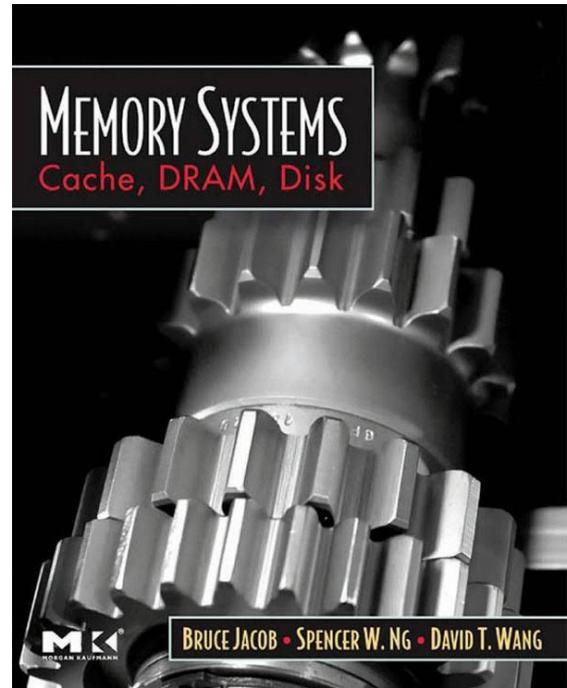
# 阅读和思考

## □ 阅读

- [自选] Memory Systems: Cache, DRAM, Disk (作者: Bruce Jacob, Spencer W. Ng, David T. Wang)
- 更细节的内容参考第二大部分(Chapter 10,11,13)

## □ 思考

- 程序的局部性原理指什么?为什么层次存储器系统能同时达到高性能/低成本/大容量的指标?



---

谢谢

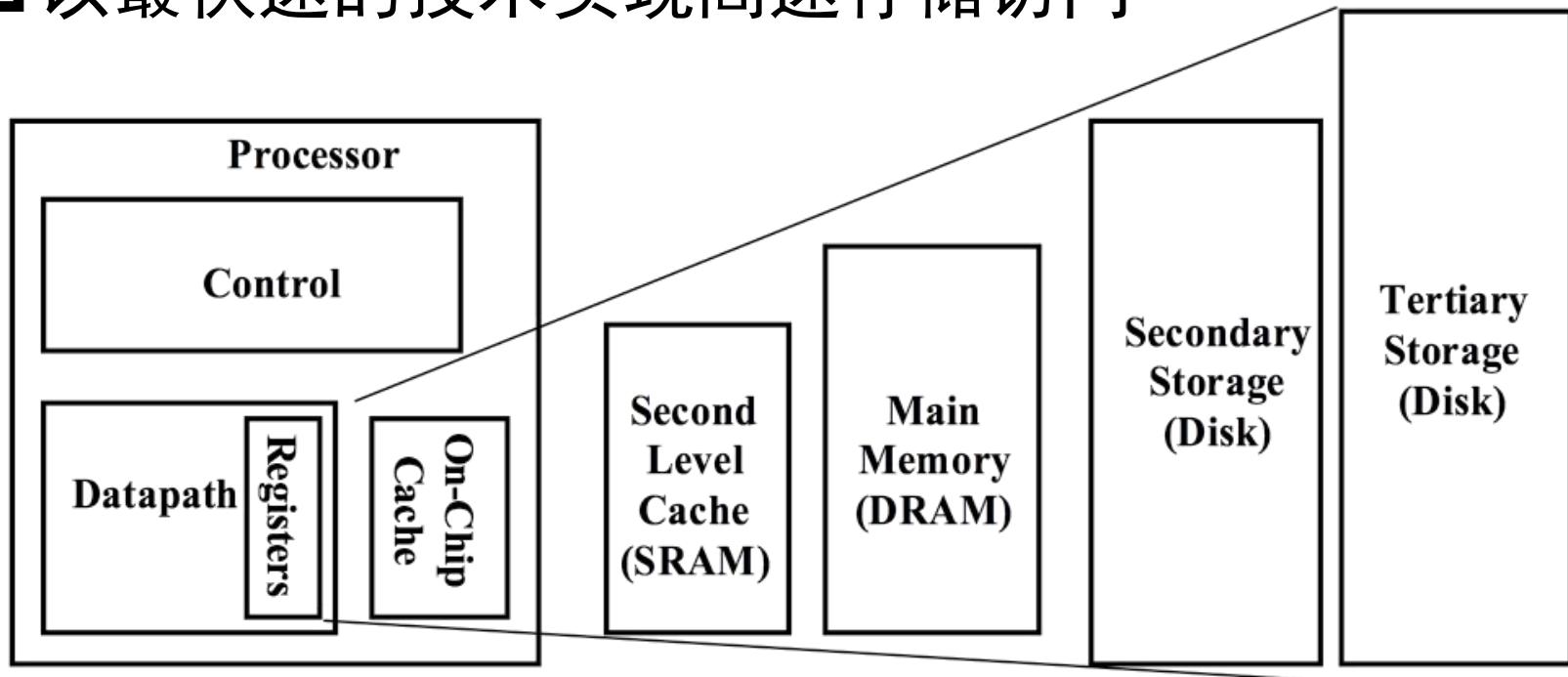
# 最新的一些磁盘性能数据

## Comparing Drive Types

|                      | HDD                           | Flash-based SSD                 |
|----------------------|-------------------------------|---------------------------------|
| Durability and Noise | Loud and susceptible to shock | No moving parts!                |
| Access Time          | ~ 12 ms<br>≈ 30M clock cycles | ~ 0.1 ms<br>≈ 250K clock cycles |
| Relative Power       | 1                             | 1/3                             |
| Cost                 | ~ \$0.035 / GB                | ~ \$0.35 / GB                   |
| Capacity             | 500GB - 12TB                  | 128GB - 2TB                     |
| Other Problems       | Fragmentation                 | Limited Writes                  |
| Lifespan             | 5-10 years                    | Avg Failure rate 6 years        |

# 层次存储器系统

- 利用程序的局部性原理:
- 以最低廉的价格提供尽可能大的存储空间
- 以最快速的技术实现高速存储访问



Speed (ns): 1ns

10ns

50-100ns  
MB-GB

Milliseconds  
GB

Seconds  
Terabytes

# 现代计算机存储器系统

---

## □ 主存储器

- 寄存器Register
- 高速缓存Cache
- 主存储器Main Memory

## □ 辅助存储器

- 磁盘Disk
- 磁带Tape
- 光盘Compact Disc

# 并行技术

---

## □ 主存的一体多字

- 一个读写体，每次多个字

## □ 单字多体

- 多个读写体，交叉编址

## □ 多端口存储器



# 静态存储器 及高速缓冲存储器

2022年秋

# 本节内容提要

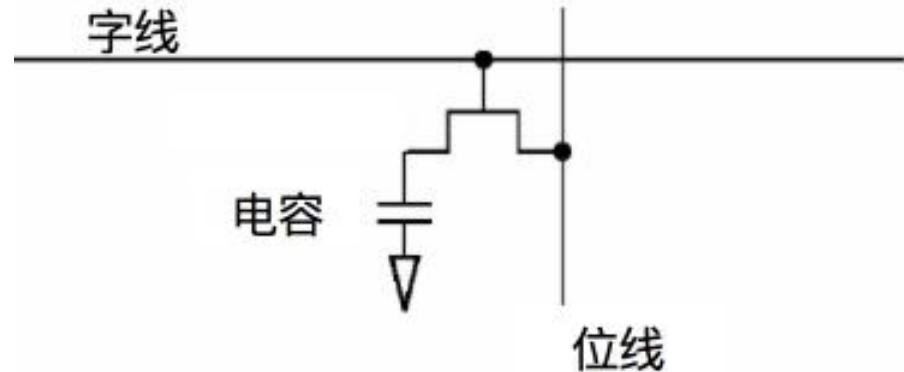
---

- 动态存储器存储原理
- 静态存储器存储原理
- 高速缓冲存储器（Cache）概述
- Cache的地址映射
  - 直接映射
  - 全相联
  - 多路组相连

# 动态存储器原理

## □ 写

- 往位线上送数据
- 选择字线



## □ 读

- 将位线上置高电平
- 选中字线
- 感知电容是否放电并放大
- 写回

## □ 刷新

- 定期的批量读操作

# 动态存储器的特点

---

□ 存储容量高

- 单位存储单元面积小

□ 访问速度慢

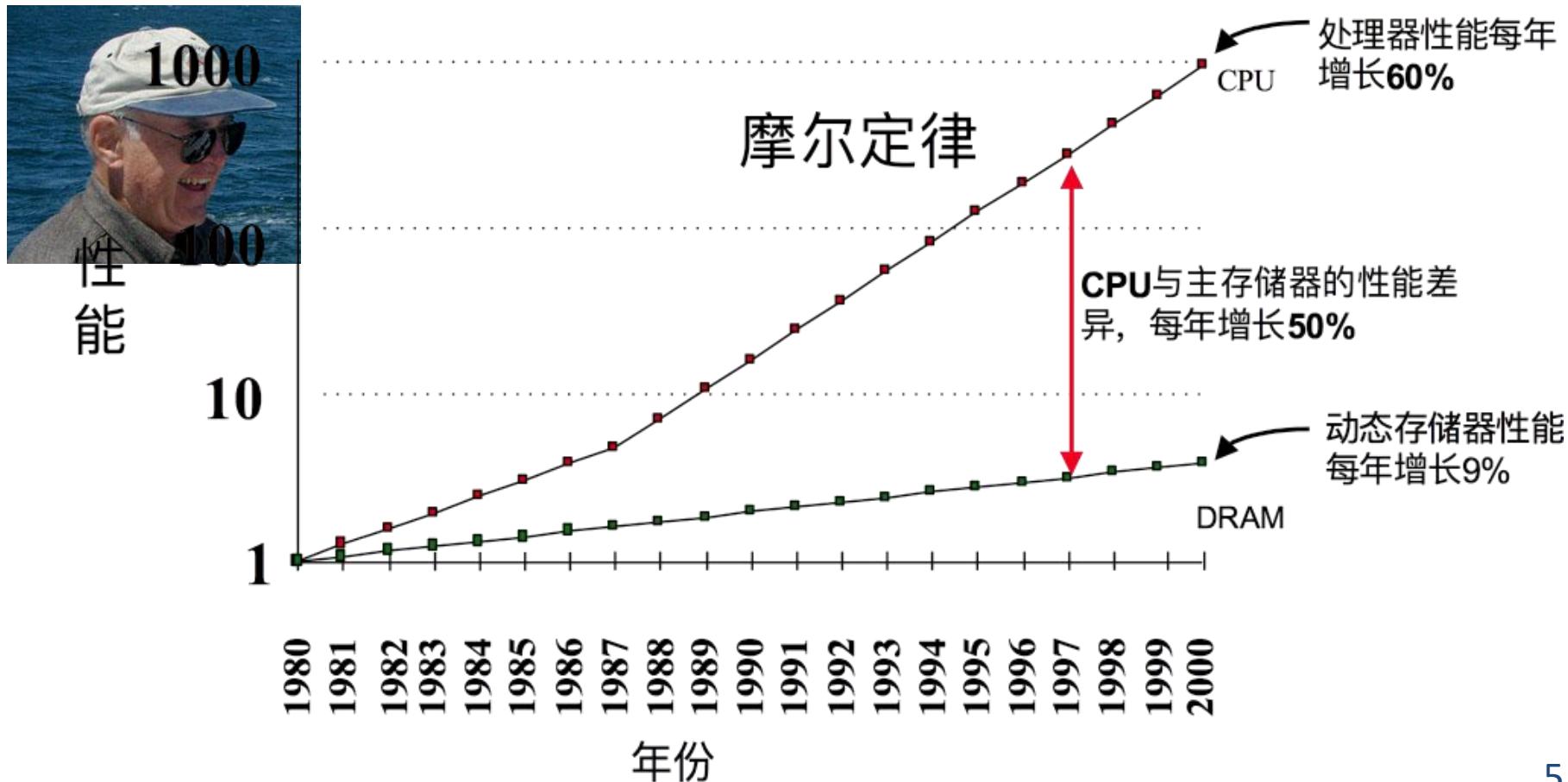
- 电容充放电
- 刷新

□ 能耗低

□ 成本低

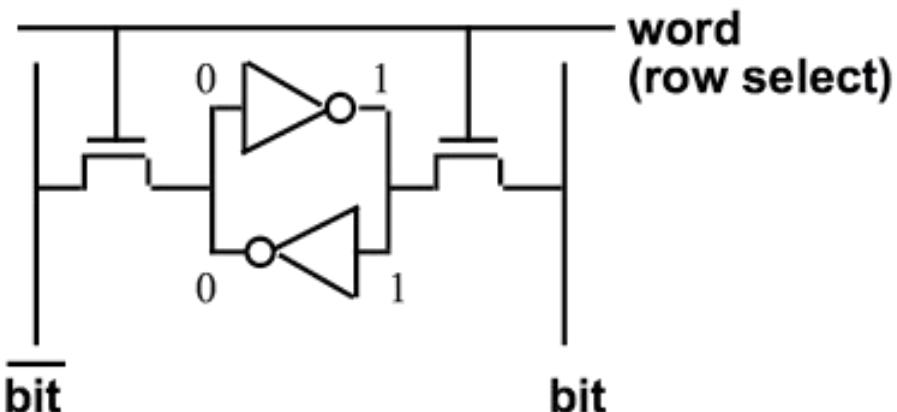
# Moore定律

- 1965年，Intel公司创始人之一Gordon Moore提出
- 芯片上集成的晶体管数量每18个月翻一番



# 静态存储器存储单元

6-Transistor SRAM Cell



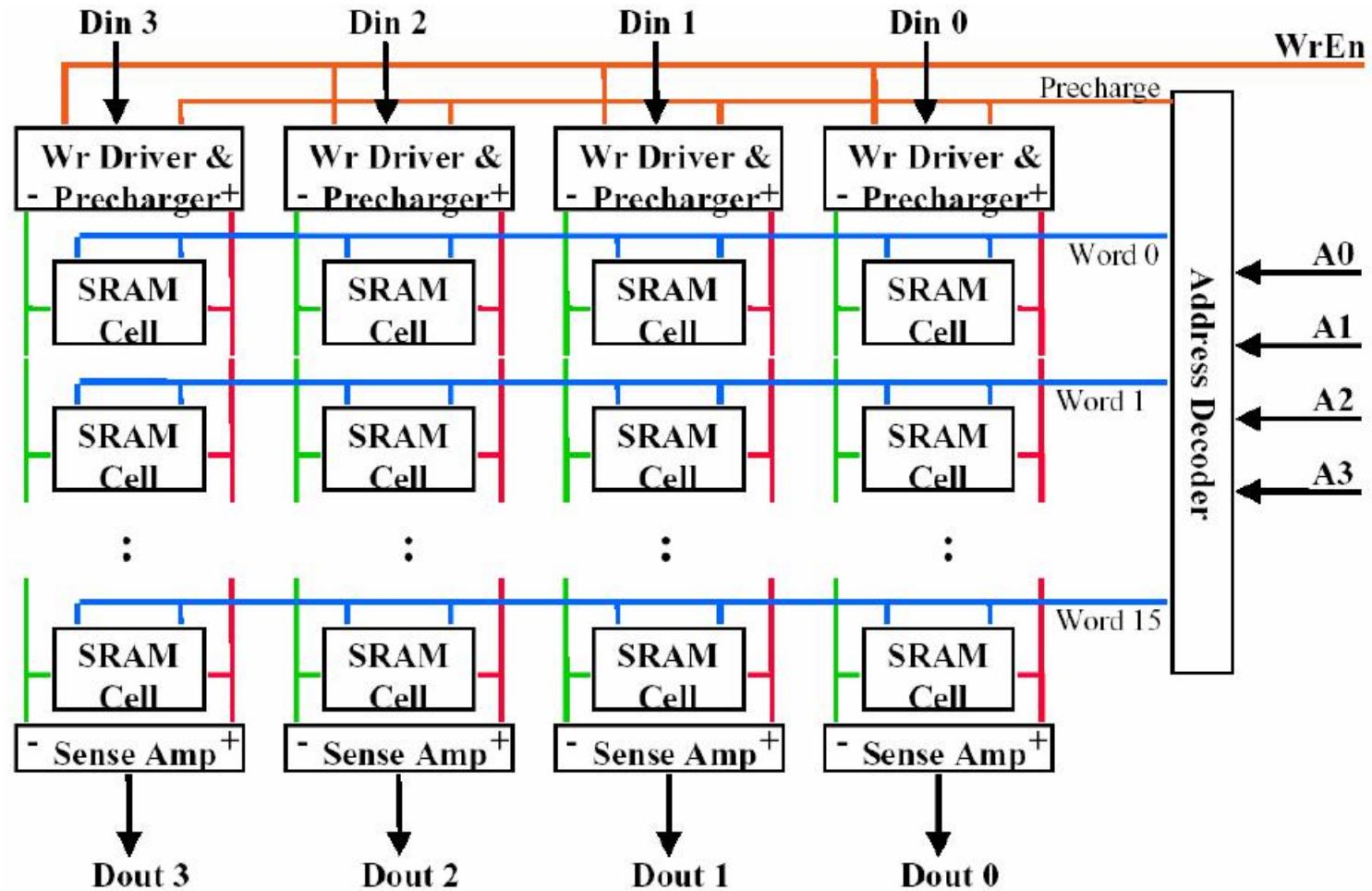
□ 写1:

- 1. 在位线上设置使( $\overline{\text{bit}}=1, \text{bit}=0$ )
- 2. 使字线选通

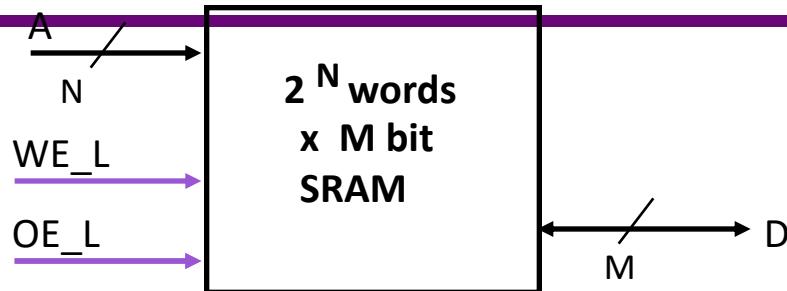
□ 读:

- 1. 使 $\overline{\text{bit}}$  和 $\text{bit}$  都充为高电平 $V_{dd}$
- 2. 使字线选通
- 3. 根据触发器的状态, 将使其中一条位线电平为低
- 4. 放大器感知 $\text{bit}$  和 $\overline{\text{bit}}$ 的变化, 读出存储的值

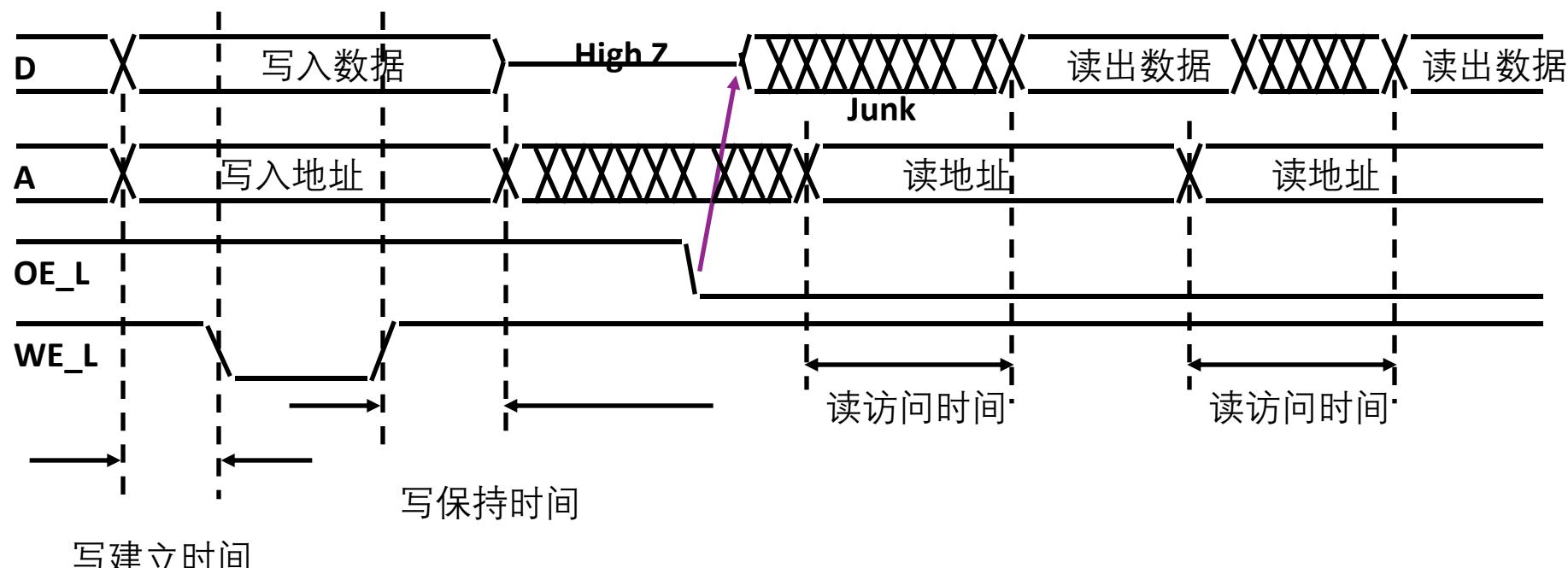
# 静态存储器典型组织方式



# SRAM典型时序

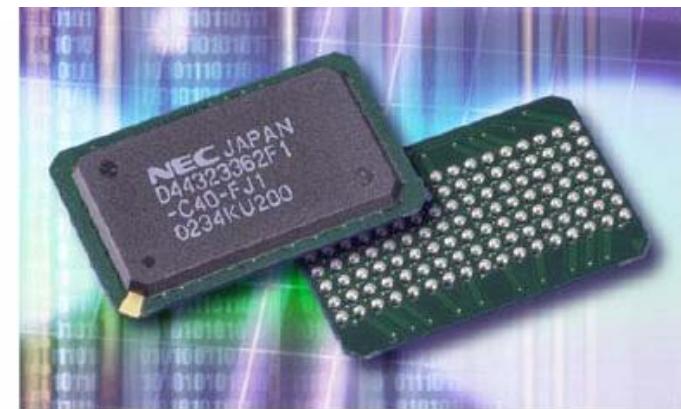
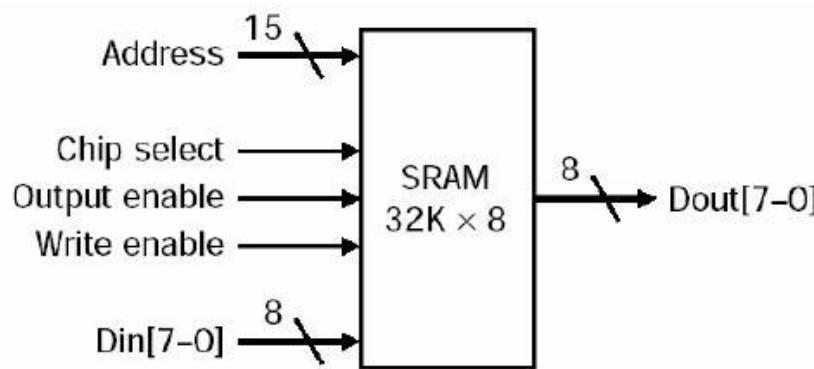


写时序:



# 静态存储器

- 速度快
- 存储密度低，单位面积存储容量小
- 数据入/出共用管脚
- 能耗高
- 成本高



# 静态和动态存储器芯片特性

|       | SRAM | DRAM |
|-------|------|------|
| 存储信息  | 触发器  | 电容   |
| 破坏性读出 | 非    | 是    |
| 需要刷新  | 不要   | 需要   |
| 送行列地址 | 同时送  | 分两次送 |
| 运行速度  | 快    | 慢    |
| 集成度   | 低    | 高    |
| 发热量   | 大    | 小    |
| 存储成本  | 高    | 低    |

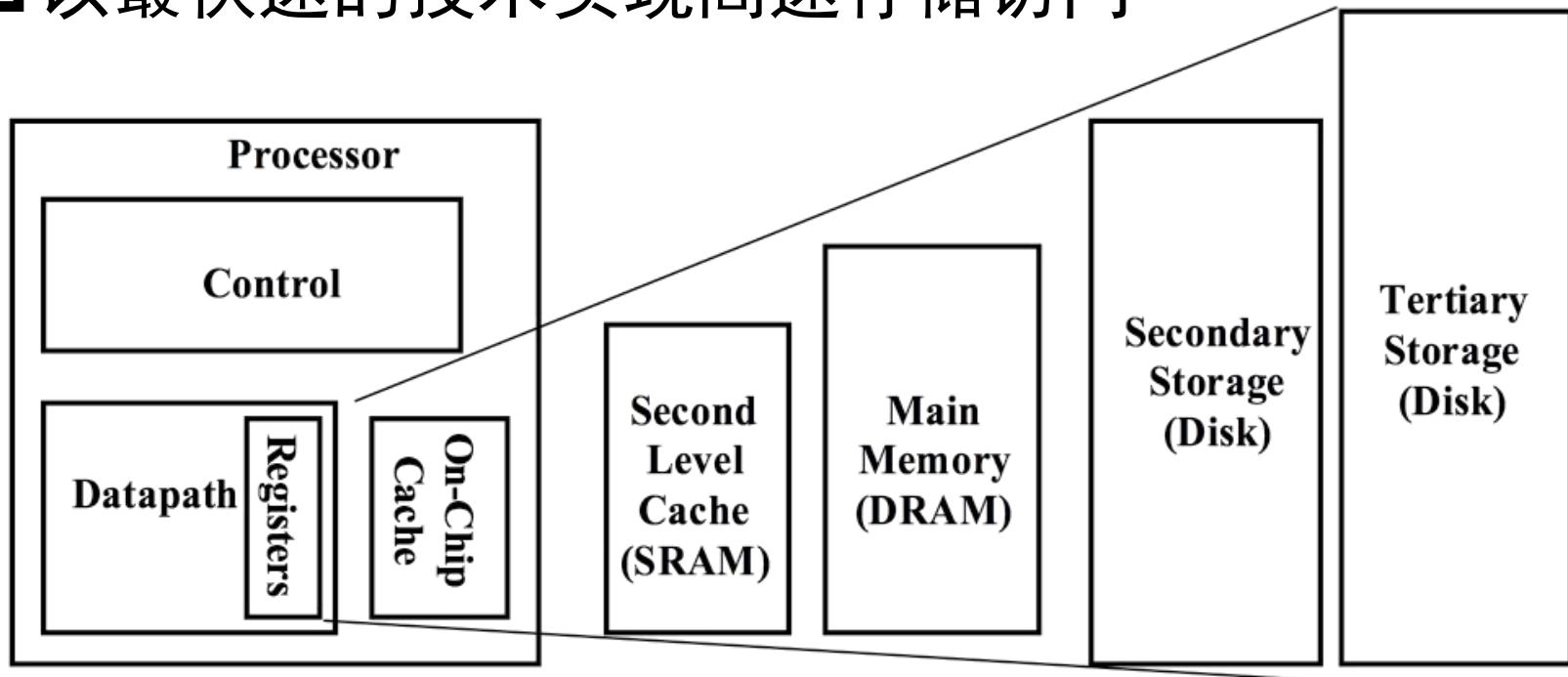
# 程序运行的局部性原理

---

- 程序运行时的局部性原理表现在：
- 在一小段**时间**内，最近被访问过的程序和数据很可能再次被访问
- 在**空间**上这些被访问的程序和数据往往集中在一小片存储区
- 在访问**顺序**上，指令顺序执行比转移执行的可能性大(大约5:1 )
- 合理地把程序和数据分配在不同存储介质中

# 层次存储器系统

- 利用程序的局部性原理:
- 以最低廉的价格提供尽可能大的存储空间
- 以最快速的技术实现高速存储访问



Speed (ns): 1ns

10ns

50-100ns  
MB-GB

Milliseconds  
GB

Seconds  
Terabytes

# 程序的局部性原理

```
for (i=0; i<1000; i++) {
 for (j=0; j<1000; j++) {
 a[i] = b[i] + c[i];
 }
}

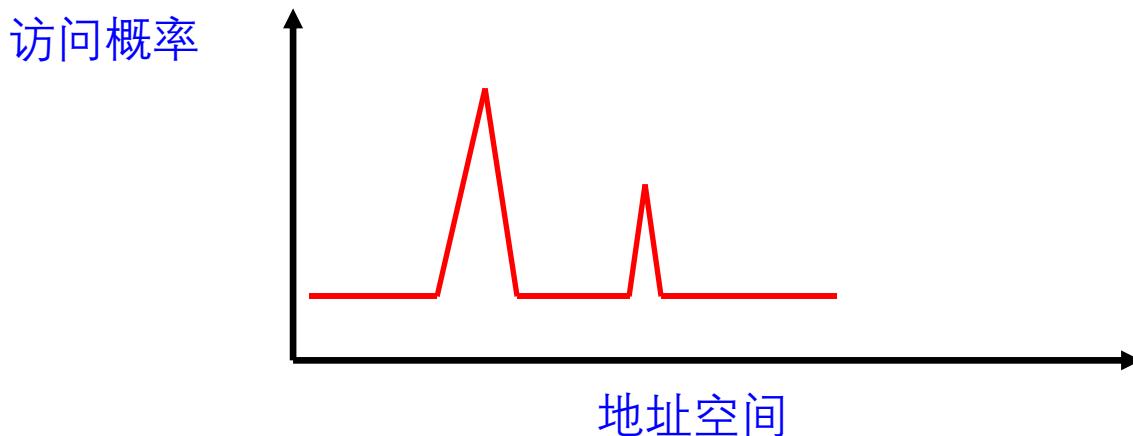
If err {}
else for (i=0; i<1000; i++) {
 for (j=0; j<1000; j++) {
 e[i] = d[i] * a[i];
 }
}
.....
```

□ 数据流访问的局部性

□ 指令访问的局部性

□ 不同的程序段可能访问不同的内存空间

# 程序的局部性原理



- ▶ 程序在一定的时间段内通常只访问较小的地址空间
- ▶ 两种局部性：
  - ▶ 时间局部性
  - ▶ 空间局部性

# 层次存储器系统

- 使用高速缓冲存储器Cache来提高CPU对存储器的平均访问速度。
- 时间局部性：最近被访问的信息很可能还要被访问。
  - 将最近被访问的信息项装入到Cache中。
- 空间局部性：最近被访问的信息临近的信息也可能被访问。
  - 将最近被访问的信息项临近的信息一起装入到Cache中。

# 高速缓冲存储器Cache

---

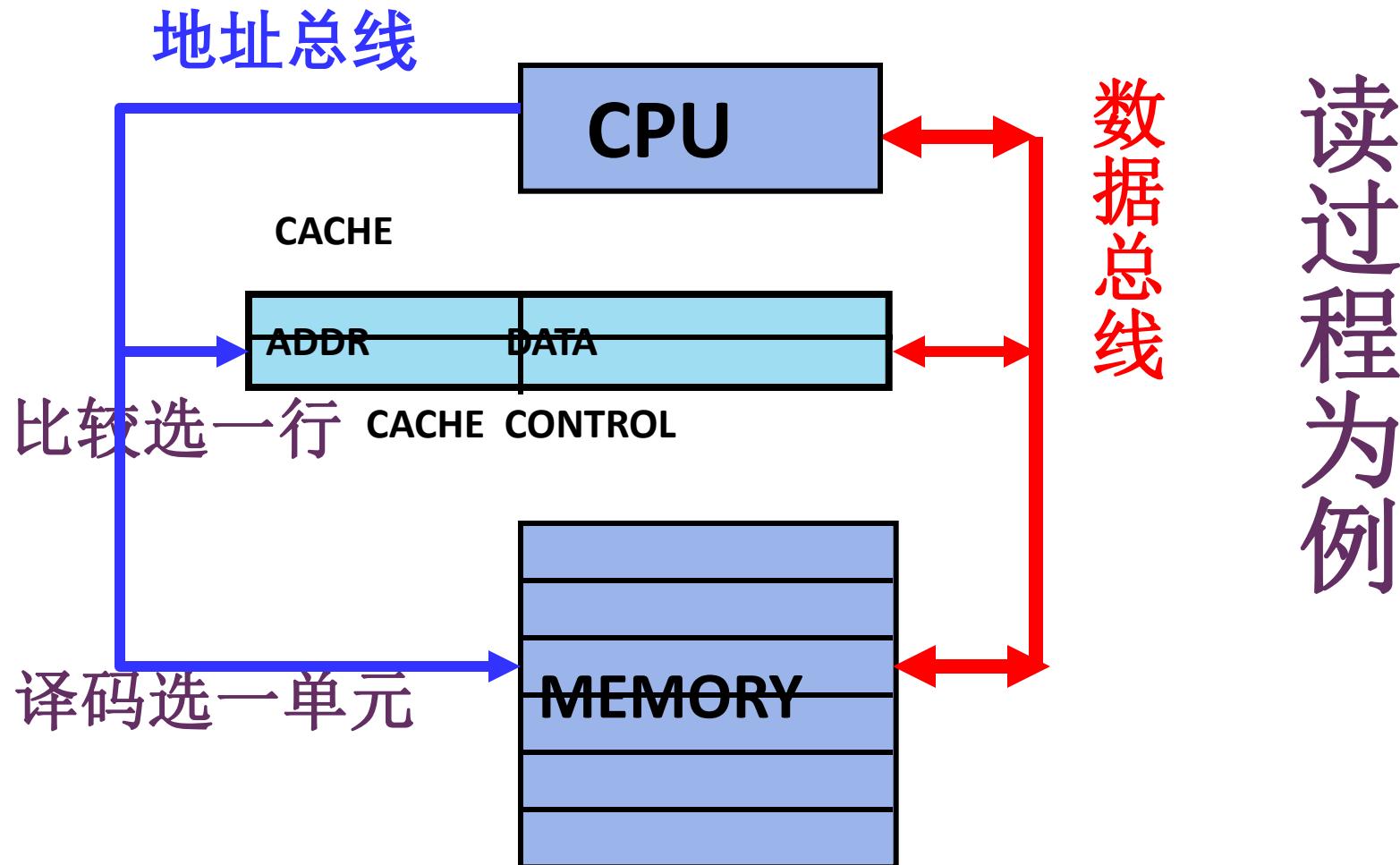
## □ 定义

- 设置于主存和CPU之间的存储器，用高速的静态存储器实现，缓存了CPU频繁访问的信息。

## □ 特点

- 高速：与CPU的运行速度基本匹配
- 透明：完全硬件管理，对程序员透明

# Cache的基本运行原理



# 要解决的问题

---

1. 地址和Cache行之间的映射关系：  
如何根据主存地址得到Cache中的数据？
  
2. 数据之间一致性：  
Cache中的内容是否已经是主存对应地址的内容？
  
3. 数据交换的粒度：  
Cache中的内容与主存内容以多大的粒度交换？
  
4. Cache内容装入和替换策略  
如何提高Cache的命中率？

# Cache参数

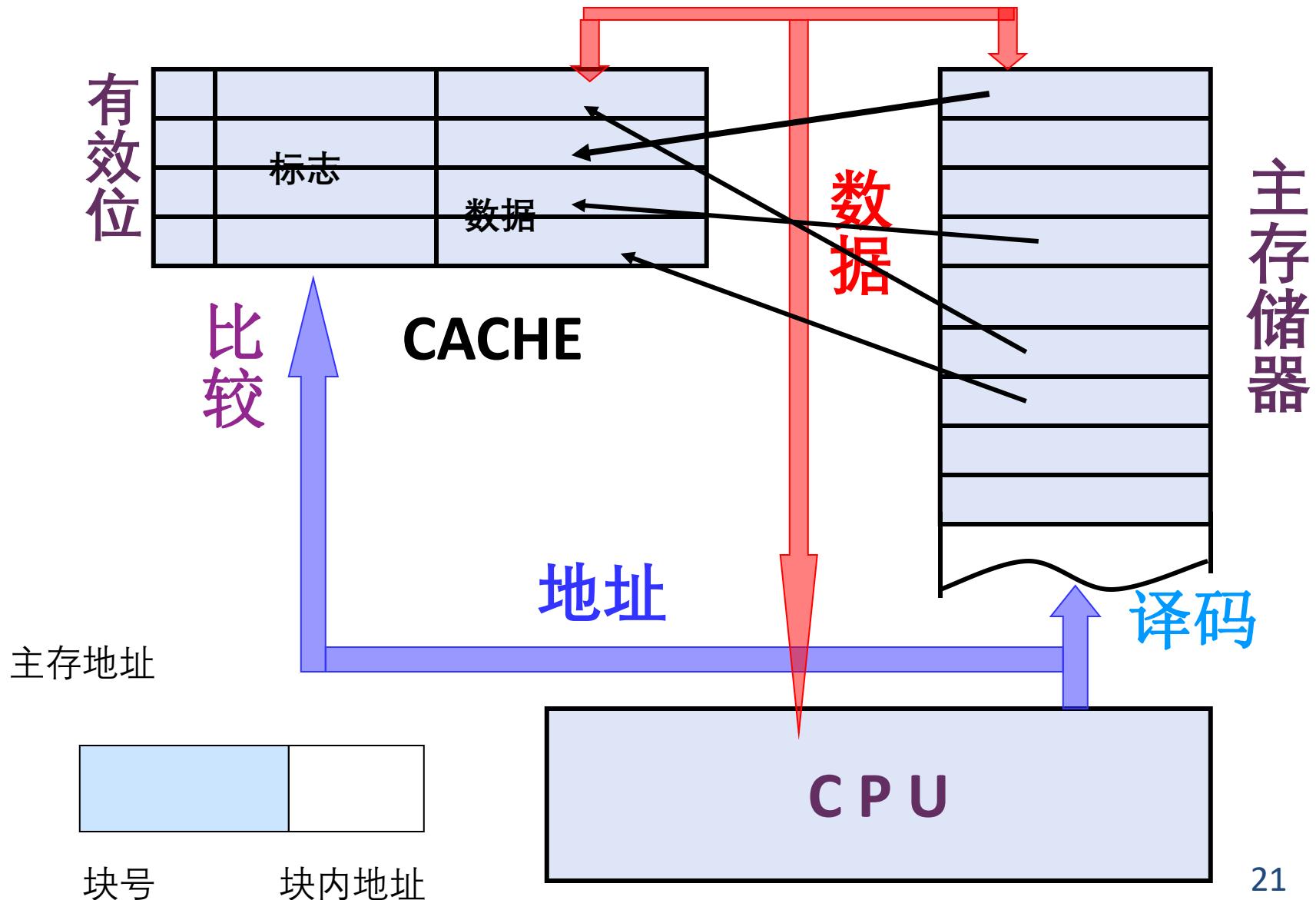
- 块 (Line) : 数据交换的最小单位
- 命中 (Hit) : 在较高层次中发现要访问的内容
  - 命中率 (Hit Rate) : 命中次数/访问次数
  - 命中时间: 访问在较高层次中数据的时间
- 缺失 (Miss) : 需要在较低层次中访问块
  - 缺失率 (Miss Rate) : 1-命中率
  - 缺失损失 (Miss Penalty) : 替换较高层次数据块的时间 + 将该块交付给处理器的时间
- 命中时间 << 缺失损失
- 平均访问时间 =  $HR * \text{命中时间} + (1-HR) * \text{缺失损失}$

# 参数典型数值

---

- 块大小： 4~128 Bytes
- 命中时间： 1~4周期
- 失效损失：
  - 访问时间： 6~10个周期
  - 传输时间： 2~22个周期
- 命中率： 80%~99%
- Cache容量： 1KB~256KB

# 全相联方式



# 全相连映射硬件实现举例

---

主存: 4GB, Cache: 4KB, 块大小: 4B, 全相联  
标记位数?

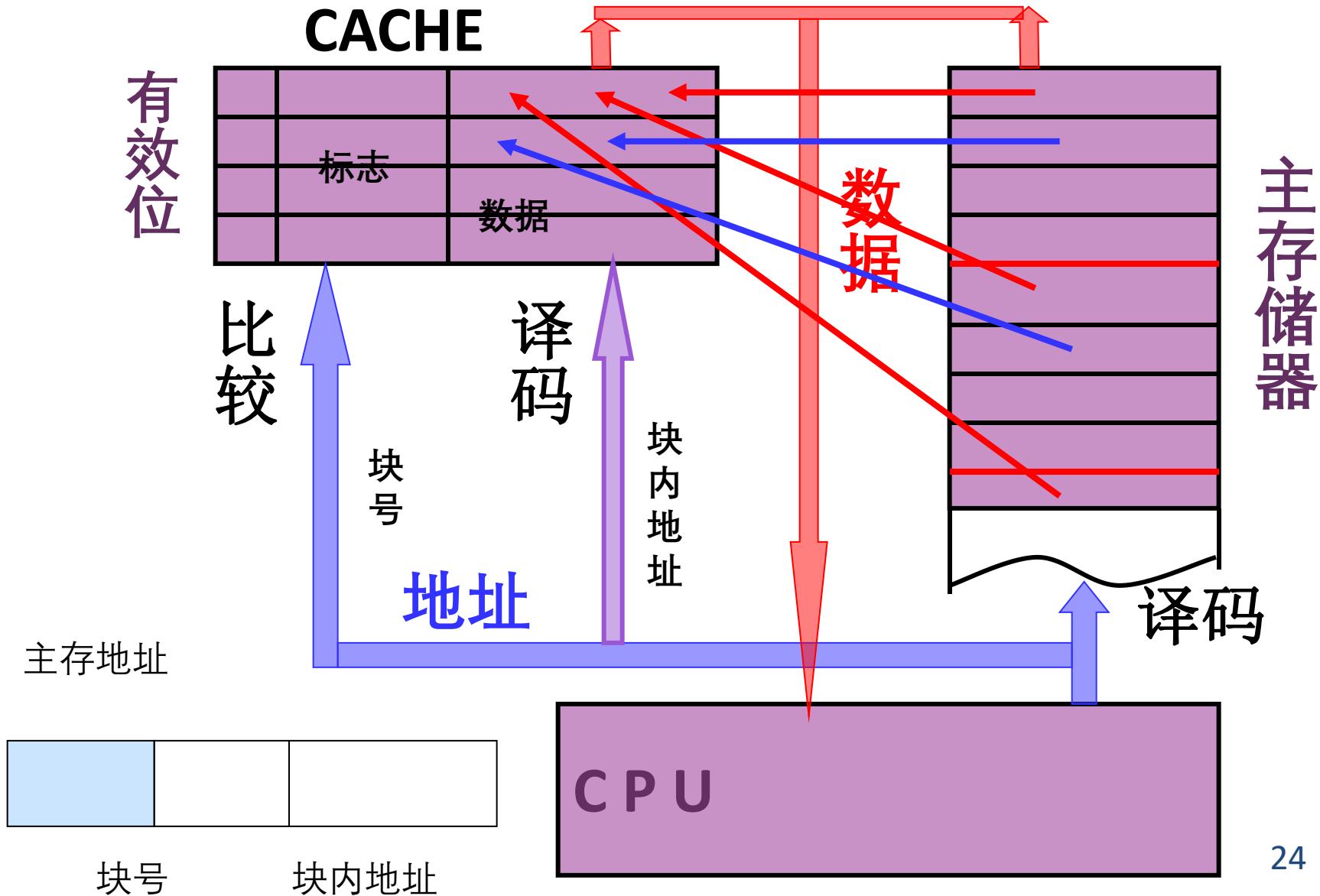
# 全相联方式的地址映射关系

## 特点

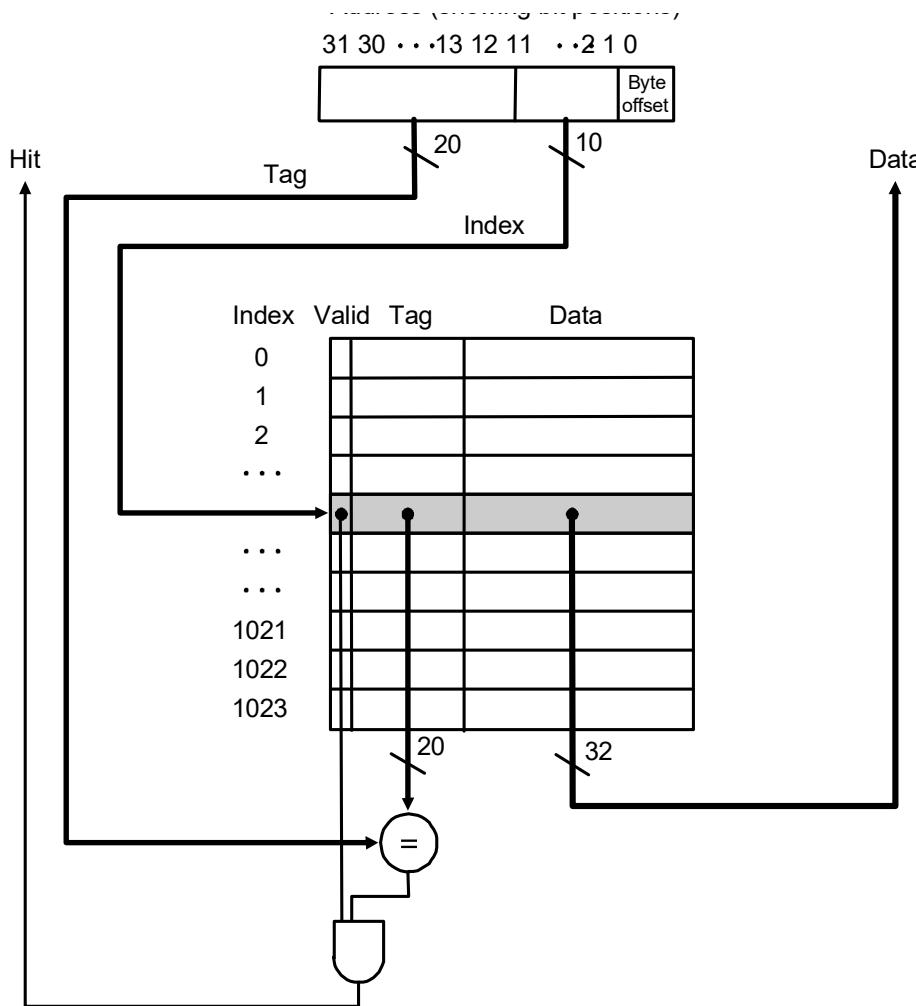
1. 主存的字块可以和Cache的任何字块对应，利用率高，方式灵活。
2. 标志位较长，比较电路的成本太高。如果主存空间有 $2^m$ 块，则标志位要有m位。**同时，如果Cache有n块，则需要有n个比较电路。**

使用成本太高

# 直接映射方式



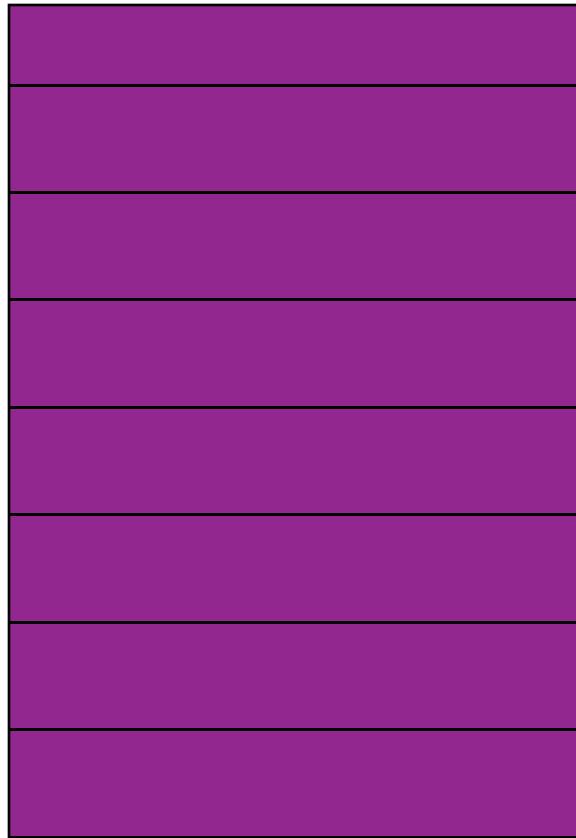
# 直接映射 Cache: 硬件实现



- 主存: 4GB
- Cache: 4KB
- 块大小: 4B
- 直接映射
- 标记位数?
- 索引位数?

# Cache 举例

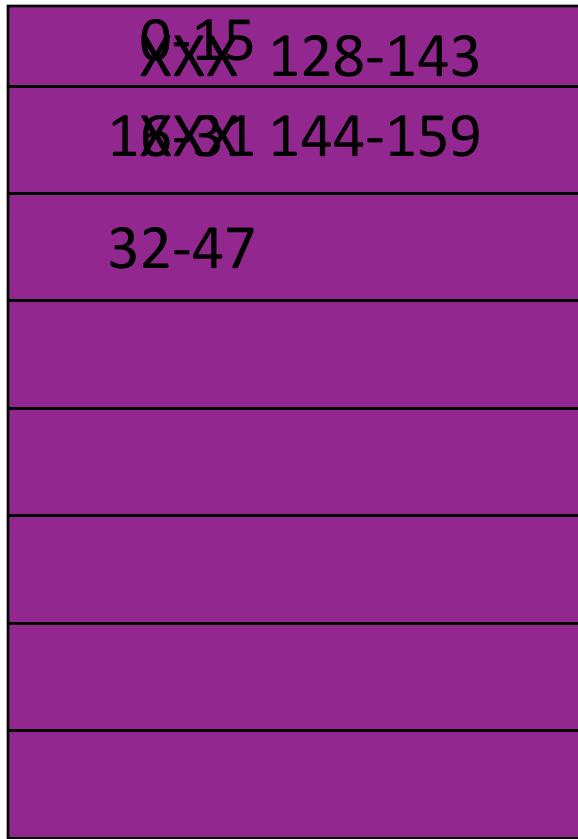
- 8 块 cache
- 每块 16 字节
- “直接映射”：内存中的每个单元在 Cache 中只会有一个唯一的位置和它对应。



|       |         |
|-------|---------|
| 0-15  | 128-143 |
| 16-31 | 144-159 |
| 32-47 | 160-175 |
| ...   | ...     |

# 直接映射Cache 举例

|       |         |
|-------|---------|
| 0-15  | 128-143 |
| 16-31 | 144-159 |
| 32-47 | 160-175 |
| ...   | ...     |



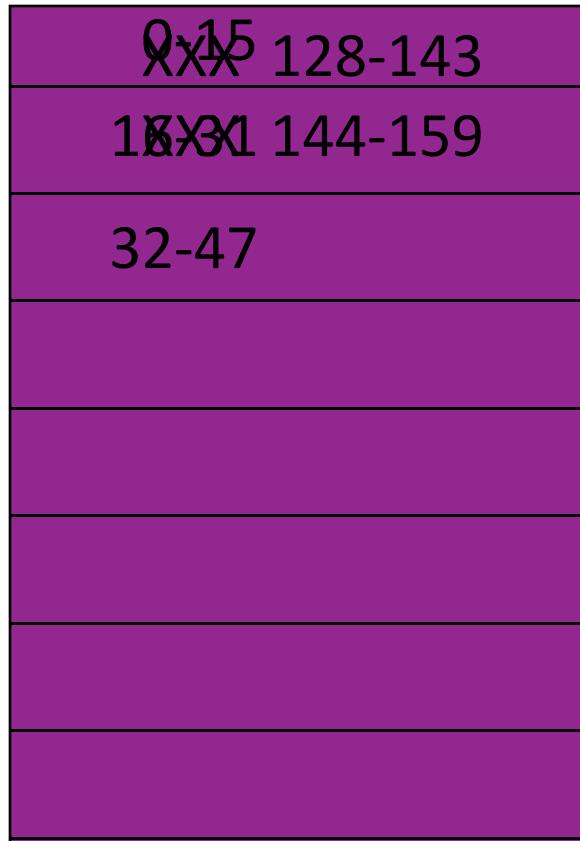
□ 假定有如下访问操作:

- Read location 0
- Read location 16
- Read location 32
- Read location 4
- Read location 8
- Read location 36
- Read location 32
- Read location 128
- Read location 148

□ cache中命中和缺失各有多少次?

# Cache 举例：续

|       |         |
|-------|---------|
| 0-15  | 128-143 |
| 16-31 | 144-159 |
| 32-47 | 160-175 |
| ...   | ...     |



## □ Cache中命中和缺失次数?

- Read location 0: Miss
- Read location 16: Miss
- Read location 32: Miss
- Read location 4: Hit
- Read location 8: Hit
- Read location 36: Hit
- Read location 32: Hit
- Read location 128: Miss
- Read location 148: Miss

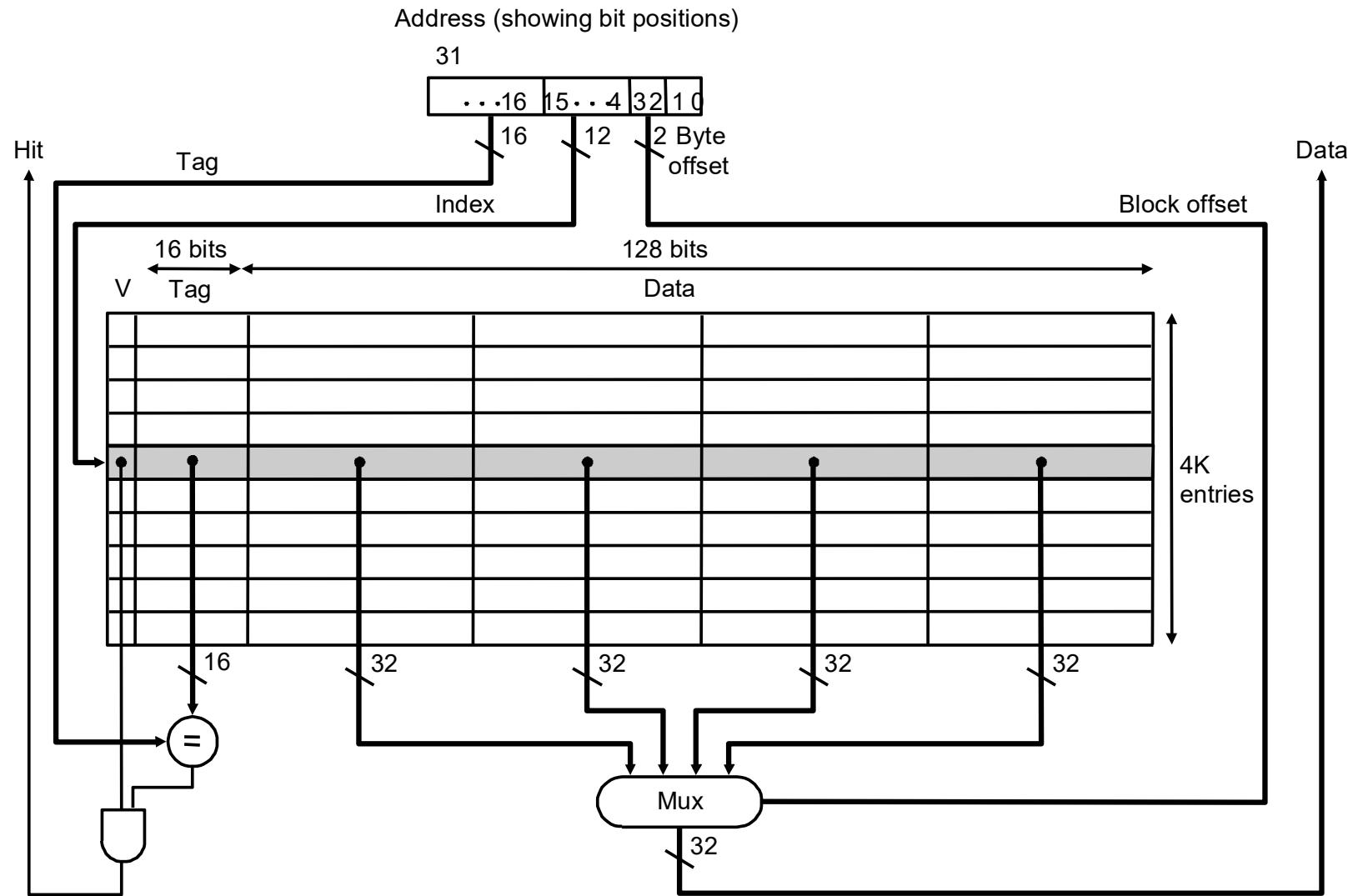
□ 命中率 = 4/9 = 45%

## □ 注意:失效的原因

- 启动失效
- 冲突失效

# 直接映射 Cache: 硬件实现

- 增加块大小可以更好地利用空间局部性



# 直接映射方式的地址映射

## 特点

1. 主存的字块只可以和固定的Cache字块对应，方式直接，利用率低。
2. 标志位较短，比较电路的成本低。如果主存空间有 $2^m$ 块，Cache中字块有 $2^c$ 块，则标志位只要有 $m-c$ 位。**且仅需要比较一次。**

利用率低，命中率低，效率较低

# 小结

---

## □ 静态存储器

- 存储速度快
- 集成度低，容量小
- 成本高

## □ Cache

- 在CPU和主存储器之间设置
- 提高访问存储器的速度
- Cache和主存地址映射方式
  - 全相联
  - 直接映射

---

谢谢



# 高速缓存

2022年秋

# 内容提要

---

## □ Cache的地址映射

- 全相联映射
- 直接映射
- 多路组相联

## □ Cache写策略

## □ 提高Cache性能的途径

- 组织结构
- Cache参数（大小、块大小、替换策略）

# 层次存储器系统

---

- 使用高速缓冲存储器Cache来提高CPU对存储器的平均访问速度。
- 时间局部性：最近被访问的信息很可能还要被访问。
  - 将最近被访问的信息项装入到Cache中。
- 空间局部性：最近被访问的信息临近的信息也可能被访问。
  - 将最近被访问的信息项临近的信息一起装入到Cache中。

# 高速缓冲存储器Cache

---

- 基于程序的局部性原理
  - 时间局部性
  - 空间局部性
- 利用静态存储器的高速特性
- 设置于主存储器与CPU之间
- 缓存CPU频繁访问的信息
- 提高CPU访问存储器的整体性能

# 需要解决的问题

---

## □ 如何通过主存地址去访问Cache?

- 全相联
- 直接映射
- 多路组相联

## □ 如何保证层次间一致性?

- 有效位、写策略

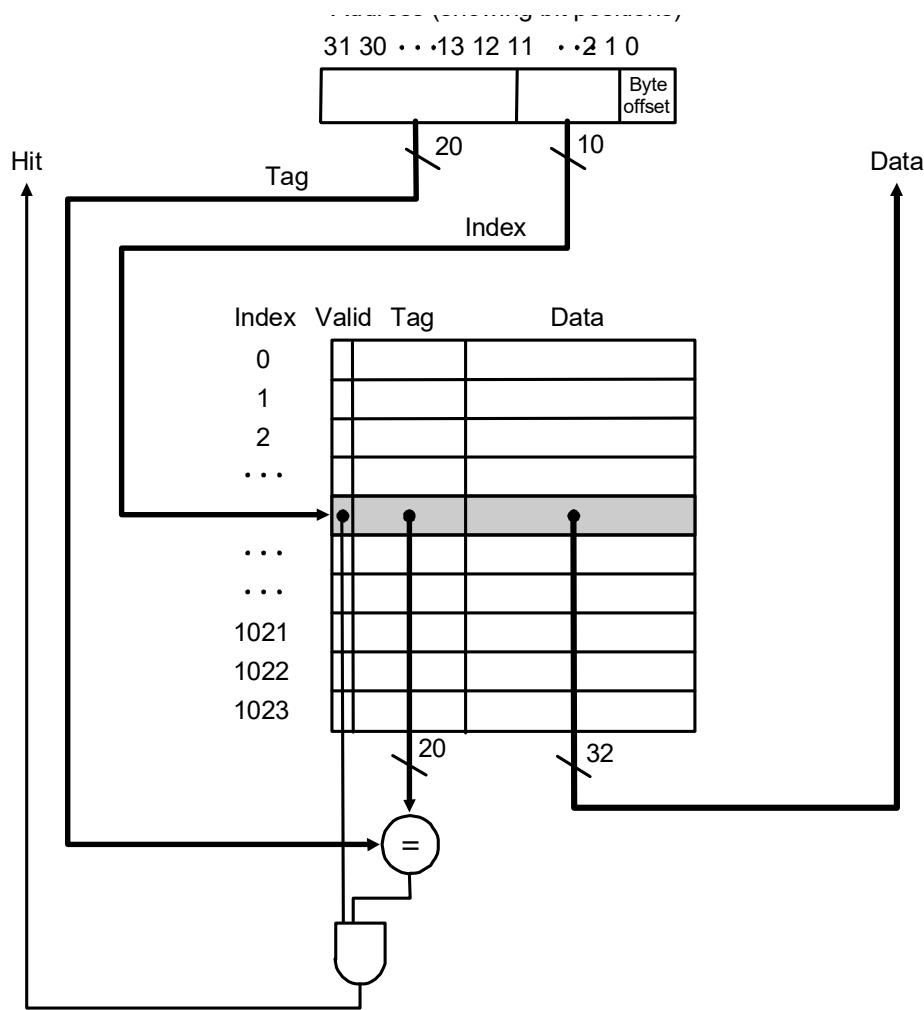
## □ Cache参数对性能的影响

- Cache的组织：块大小
- 替换策略
- 接入方式

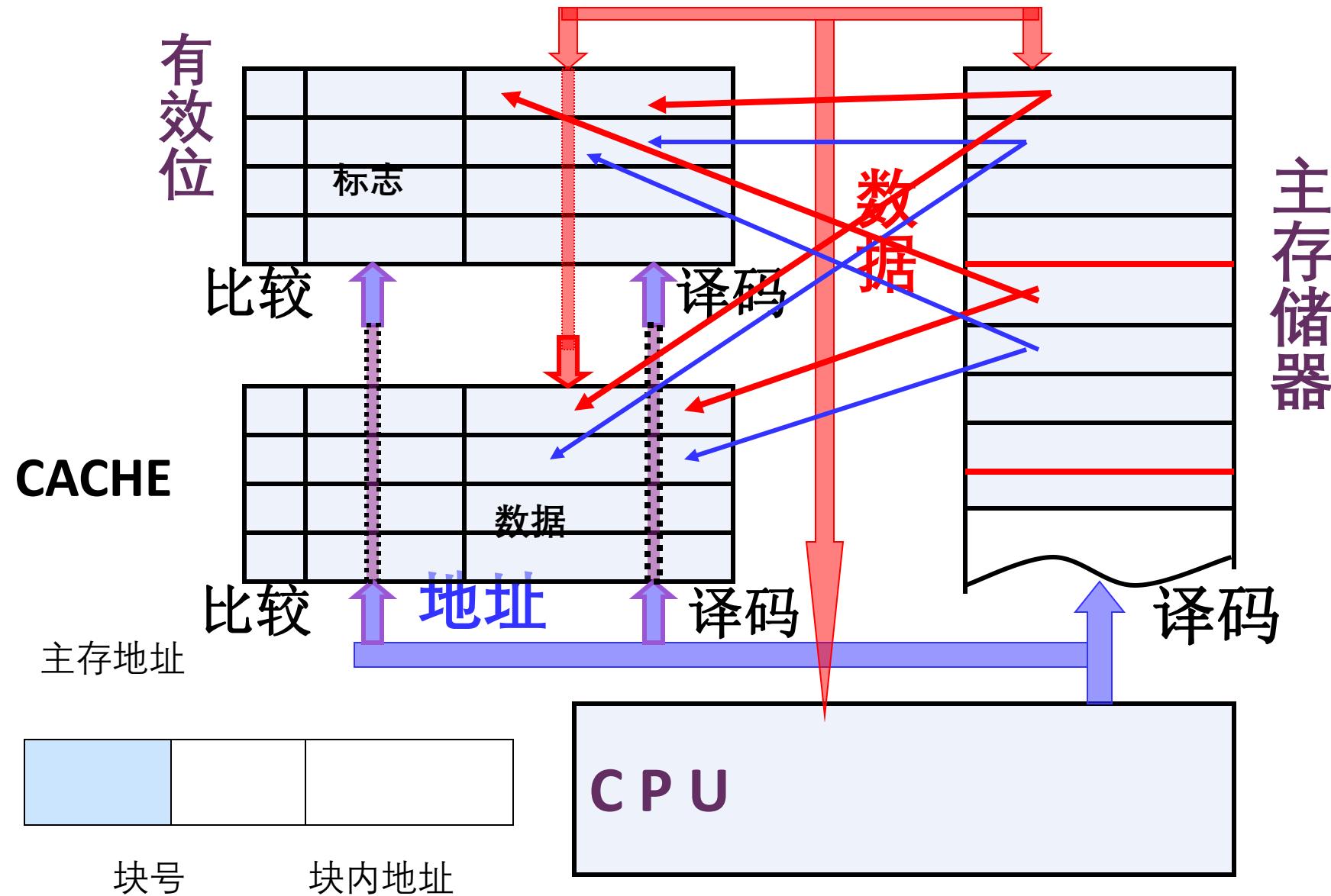
# 全相联映射硬件实现

---

# 直接映射Cache硬件实现



# 两路组相联方式



# 两路组组相联方式的地址映射

## 特点

1. 前两种方式的折衷方案。组内为全相连，组间为直接映射。
2. 集中了两个方式的优点。**成本也不太高。**

是常用的方式

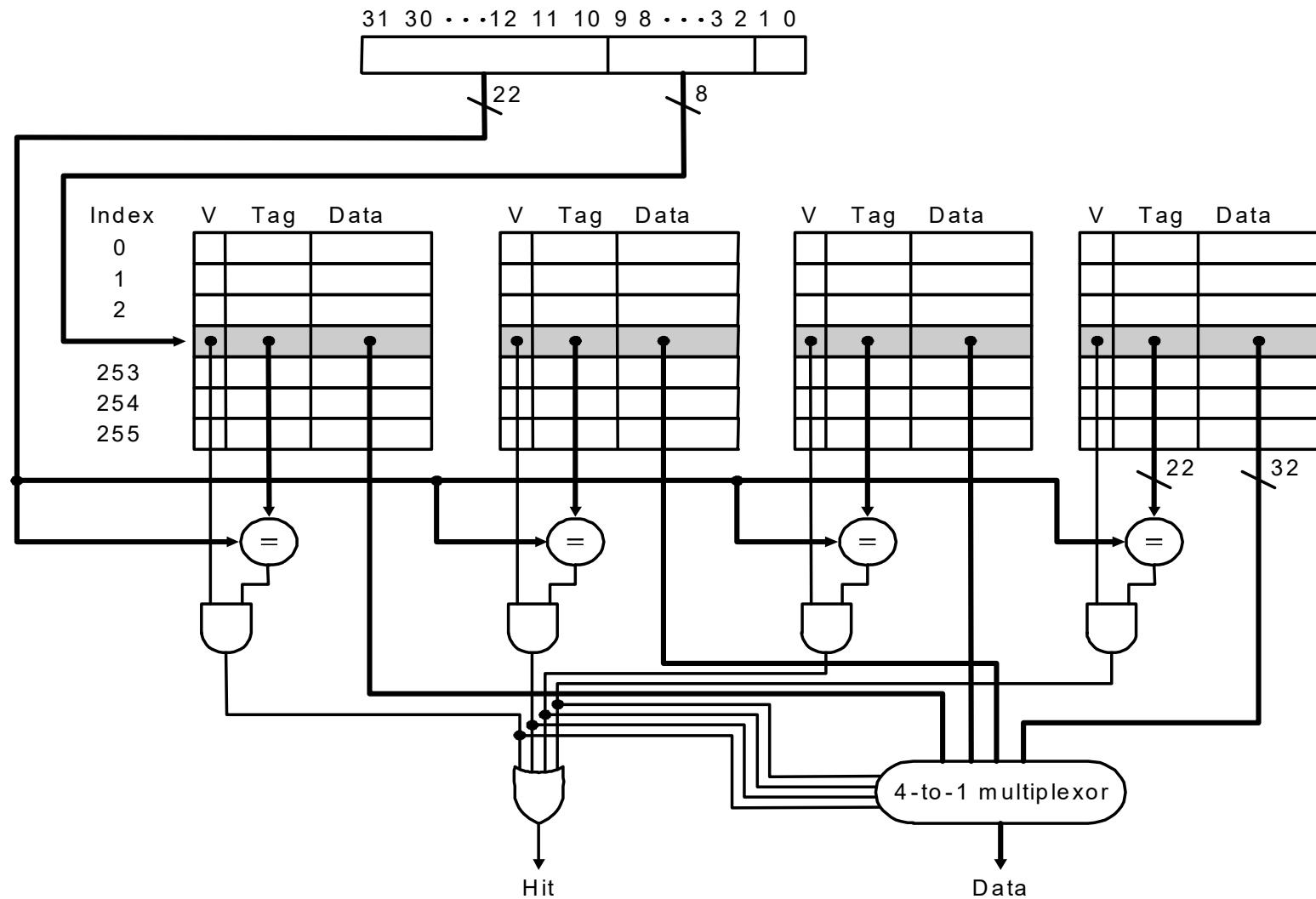
# 组相连Cache访问举例

0-15  
64-79  
...  
0-15  
64-79  
...

|       |         |
|-------|---------|
| 0-15  | 128-143 |
| 16-31 | 144-159 |
| 32-47 |         |
|       |         |

- ⌘ 假设有下列访问主存顺序：
- ↖ Read location 0: Miss
  - ↖ Read location 16: Miss
  - ↖ Read location 32: Miss
  - ↖ Read location 4: Hit
  - ↖ Read location 8: Hit
  - ↖ Read location 36: Hit
  - ↖ Read location 32: Hit
  - ↖ Read location 128: Miss
  - ↖ Read location 148: Miss
  - ↖ Read location 0: Hit
  - ↖ Read location 128: Hit
  - ↖ Read location 4: Hit
  - ↖ Read location 132: Hit

# 四路组相连的Cache实现方式



# 直接映射到全相联

One-way set associative  
(direct mapped)

| Block | Tag | Data |
|-------|-----|------|
| 0     |     |      |
| 1     |     |      |
| 2     |     |      |
| 3     |     |      |
| 4     |     |      |
| 5     |     |      |
| 6     |     |      |
| 7     |     |      |

Two-way set associative

| Set | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|
| 0   |     |      |     |      |
| 1   |     |      |     |      |
| 2   |     |      |     |      |
| 3   |     |      |     |      |

Four-way set associative

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0   |     |      |     |      |     |      |     |      |
| 1   |     |      |     |      |     |      |     |      |

Eight-way set associative (fully associative)

| Tag | Data | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|     |      |     |      |     |      |     |      |     |      |     |      |     |      |     |      |

# 三种映射方式比较

## □ 直接映射

- 主存中的一块只能映射到Cache中唯一的一个位置
- 定位时，不需要判断，只需替换

## □ 全相连映射

- 主存中的一块可以映射到Cache中任何一个位置

## □ N路组相连映射

- 主存中的一块可以选择映射到Cache中N个位置

## □ 全相连映射和N路组相连映射的失效处理

- 从主存中取出新块
- 为了腾出Cache空间，需要替换出一个Cache块
- 不唯一，则需要判断应替出哪块

# 一致性保证

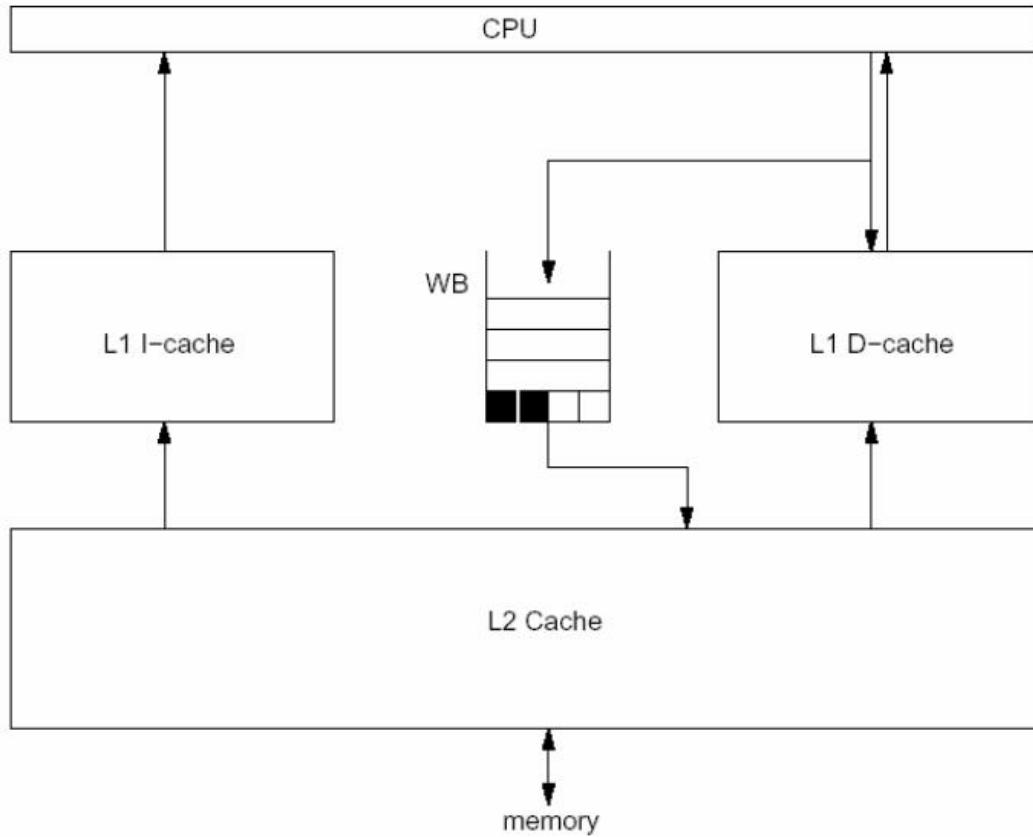
## □ 写直达 (Write through)

- 强一致性保证，效率低
- 在Cache中命中
  - 同时修改Cache和对应的主存内容
- 没有在Cache中命中
  - 写分配 (Write allocate)
  - 非写分配 (not Write allocate )

## □ 拖后写 (Write back)

- 弱一致性保证，替换时再写主存
  - 主动替换
  - 被动替换
- 通过监听总线上的访问操作来实现
- 实现复杂，效率比较高

# Cache写（命中）



# Cache (不命中) 写策略

| Steps | Write through    |                     |                     |                                | Write back           |                     |
|-------|------------------|---------------------|---------------------|--------------------------------|----------------------|---------------------|
|       | Write allocate   | No write allocate   | Write allocate      | Write back                     | Write allocate       | No write allocate   |
| 1     | pick replacement | pick replacement    | <u>write around</u> | write invalidate<br><b>Hit</b> | <u>fetch on miss</u> | no fetch on miss    |
| 2     |                  |                     |                     | invalidate tag                 | [write back]         | [write back]        |
| 3     | fetch block      |                     |                     |                                | fetch block          |                     |
| 4     | write cache      | write partial cache |                     |                                | write cache          | write partial cache |
| 5     | write memory     | write memory        | write memory        | write memory                   |                      |                     |

# 提高存储访问的性能

---

□ 平均访问时间 =

$$\text{命中时间} \times \text{命中率} + \text{缺失损失} \times \text{缺失率}$$

□ 提高命中率

□ 缩短缺失时的访问时间

□ 提高Cache本身的速度

# Cache缺失的四类原因

---

## □ 必然缺失 (Compulsory Miss)

- 开机或者是进程切换
- 首次访问数据块

## □ 容量缺失 (Capacity Miss)

- 活动数据集超出了Cache的大小

## □ 冲突缺失 (Conflict Miss)

- 多个内存块映射到同一Cache块
- 某一Cache组块已满，但空闲的Cache块在其他组

## □ 无效缺失

- 其他进程修改了主存数据

# 对策

## □ 必然缺失

- 世事总有缺憾
- 如果程序访问存储器的次数足够多，也就可以忽略了
- 策略
  - 预取

## □ 容量缺失

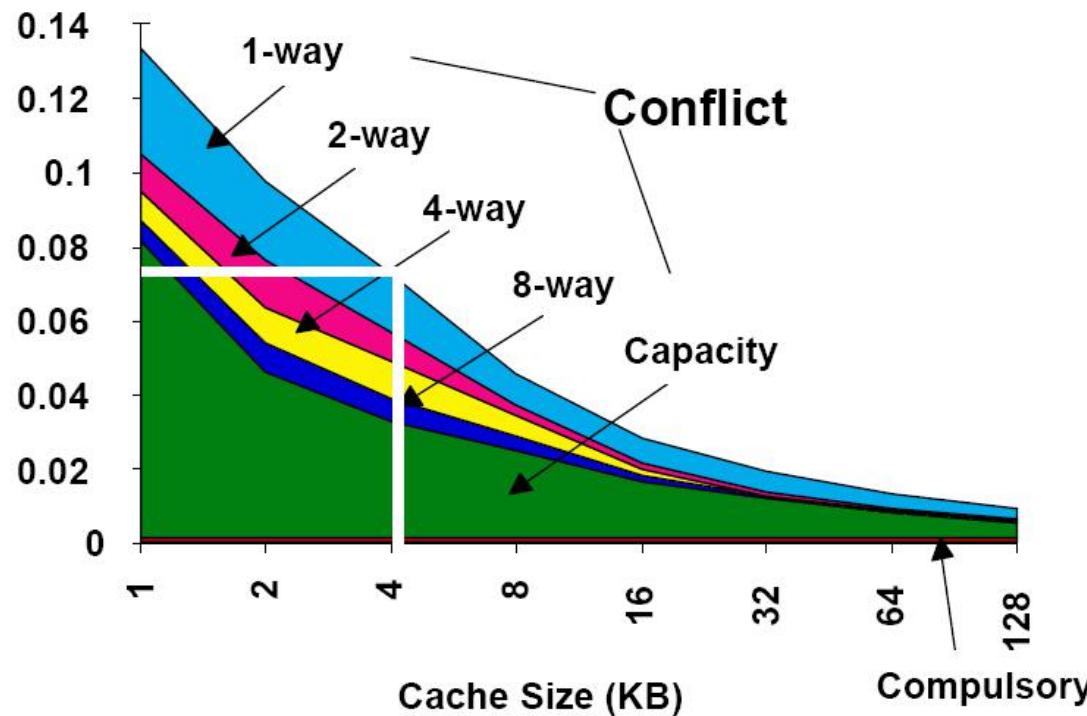
- 出现在Cache容量太小的时候
- 增加Cache容量，可缓解缺失现象

## □ 冲突缺失

- 两块不同的内存块映射到相同的Cache块
- 对直接映射的Cache，这个问题尤其突出
  - 增加Cache容量有助于缓解冲突
  - 增加相联的路数有助于缓解冲突

# 影响Cache缺失率的因素

□ 经验总结：容量为N、采用直接映射方式Cache的缺失率和容量为N/2、采用2路组相联映射方式Cache的缺失率相当



# 影响Cache命中率的因素

---

## □ Cache容量

- 大容量可以提高命中率，但是.....

## □ Cache块大小

- 选择多大的行，还真是个问题

## □ 地址映射方式

- 多路组相联，但到底多少路呢？

## □ 替换算法

- 替换哪行出去呢？

## □ 多级Cache

- 给用户更多的选择

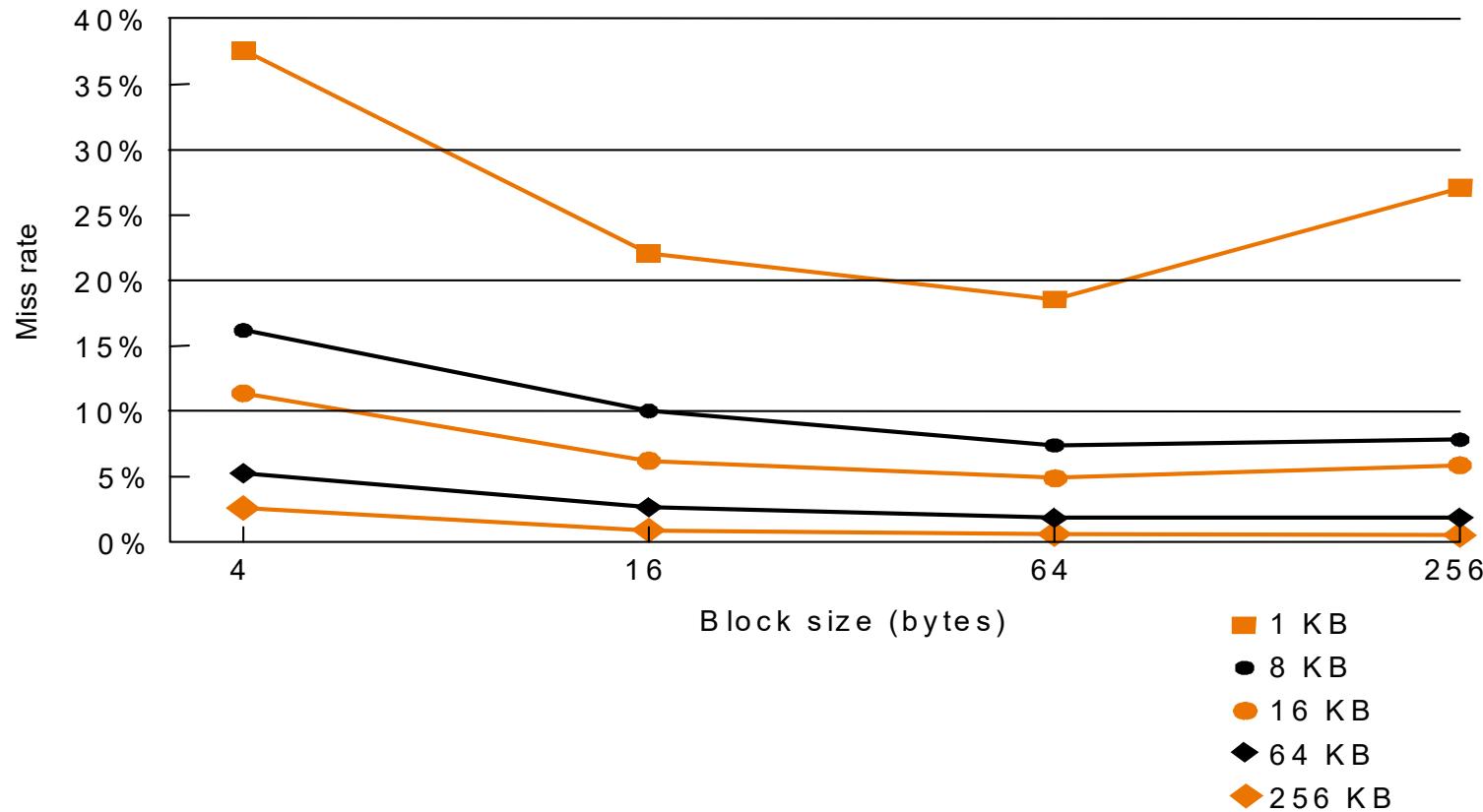
# 命中率和容量的关系

---

Hit Rate

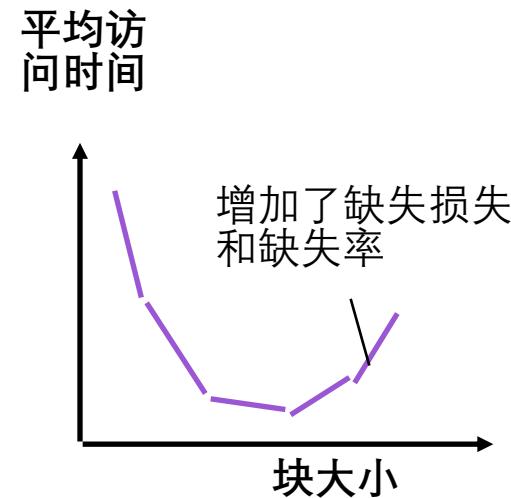
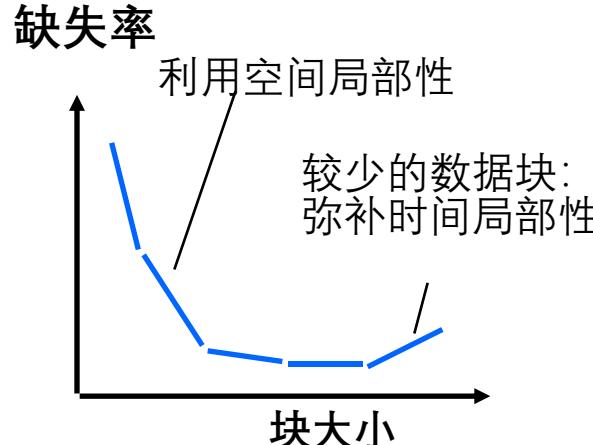
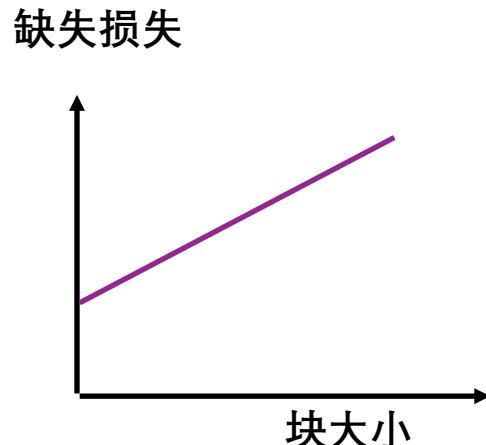
Cache Size in KB

# 块大小和缺失率的关系



# 块大小的权衡

- ▶ 一般来说，数据块较大可以更好地利用空间局部性，但是：
  - ▶ 数据块大意味着缺失损失的增大：
    - ▶ 需要花费更长的时间来装入数据块
  - ▶ 若块大小相对Cache总容量来说太大的话，命中率将降低
    - ▶ Cache块数太少
- ▶ 一般来说， $\text{平均访问时间} = \text{命中时间} \times \text{命中率} + \text{失效损失} \times \text{缺失率}$



# 块替换策略

## □ 直接映射

- 主存中的一块只能映射到Cache中唯一的一个位置
- 定位时，不需要选择，只需替换

## □ 全相联映射

- 主存中的一块可以映射到Cache中任何一个位置

## □ N路组相联映射

- 主存中的一块可以选择映射到Cache中N个位置

## □ 全相联映射和N路组相联映射的失效处理

- 从主存中取出新块
- 为了腾出Cache空间，需要替换出一个Cache块
- 不唯一，则需要选择应替出哪块

# 替换策略

## □ 最近最少使用LRU

- 满足程序局部性要求
- 有较高命中率
- 硬件实现复杂

## □ 先进先出FIFO

- 满足时间局部性
- 实现比较简单

## □ 随机替换RAND

- 实现简单
- 命中率也不太低

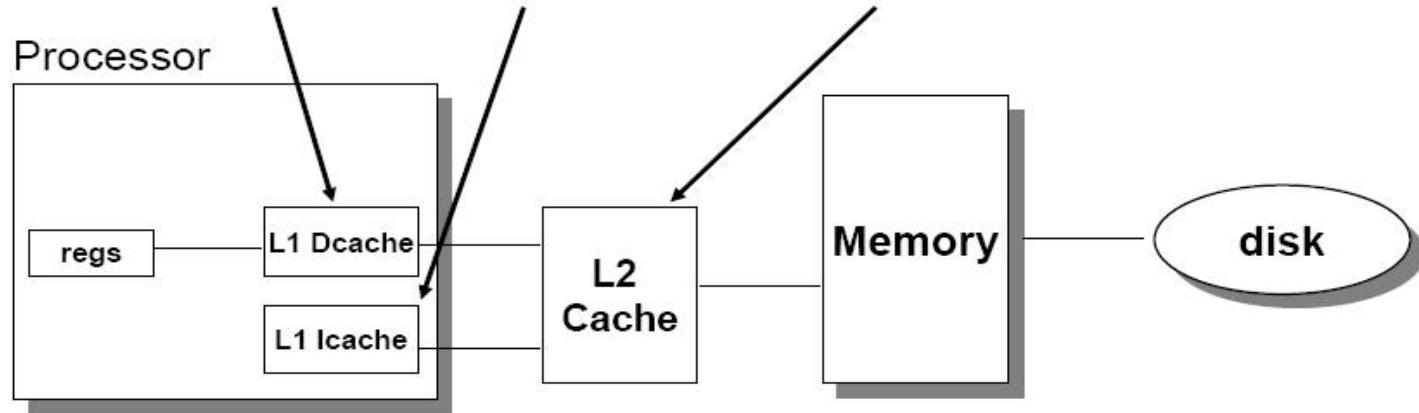
# 多级Cache

---

- 采用两级或更多级cache来提高命中率
  - 增加Cache层次
  - 增加了用户的选择
- 将Cache分解为指令Cache和数据Cache
  - 指令流水的现实要求
  - 根据具体情况，选用不同的组织方式、容量

# 多级Cache

Options: *separate* data and instruction caches, or a *unified* cache

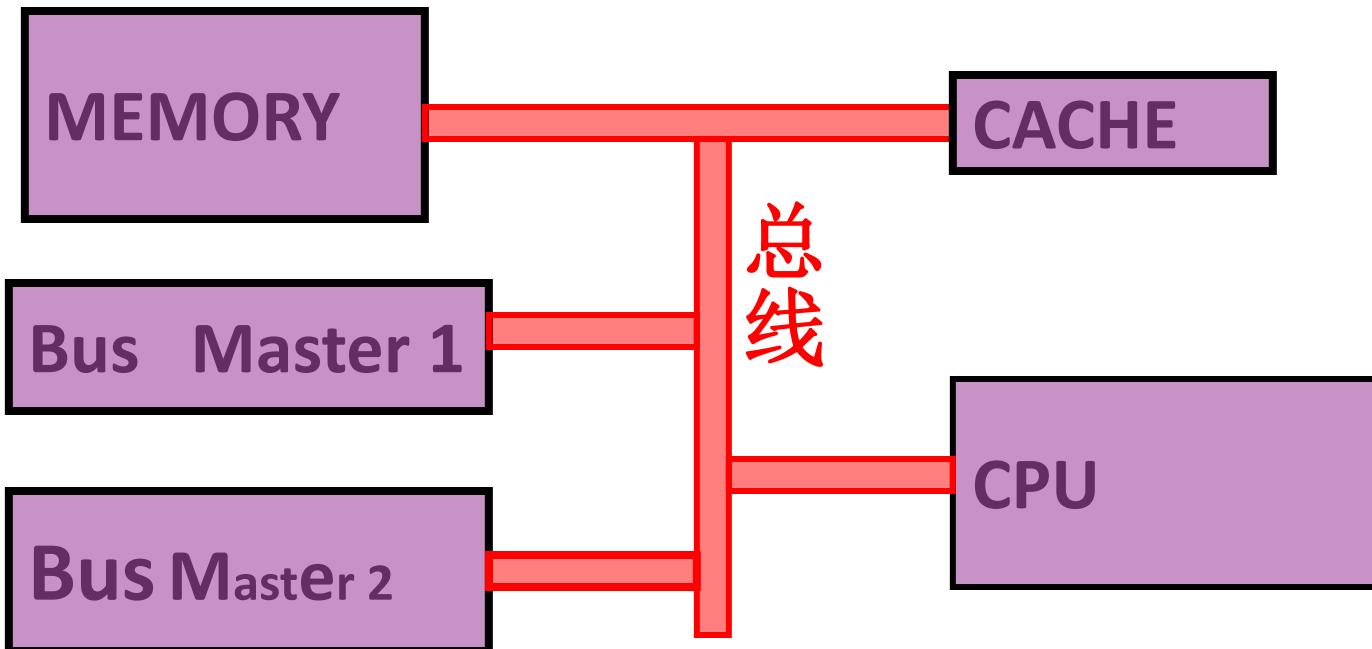


Inclusive vs. Exclusive

- 在处理器中设置独立的数据Cache和指令Cache
- 在处理器外设置第二级Cache，甚至是第三级Cache

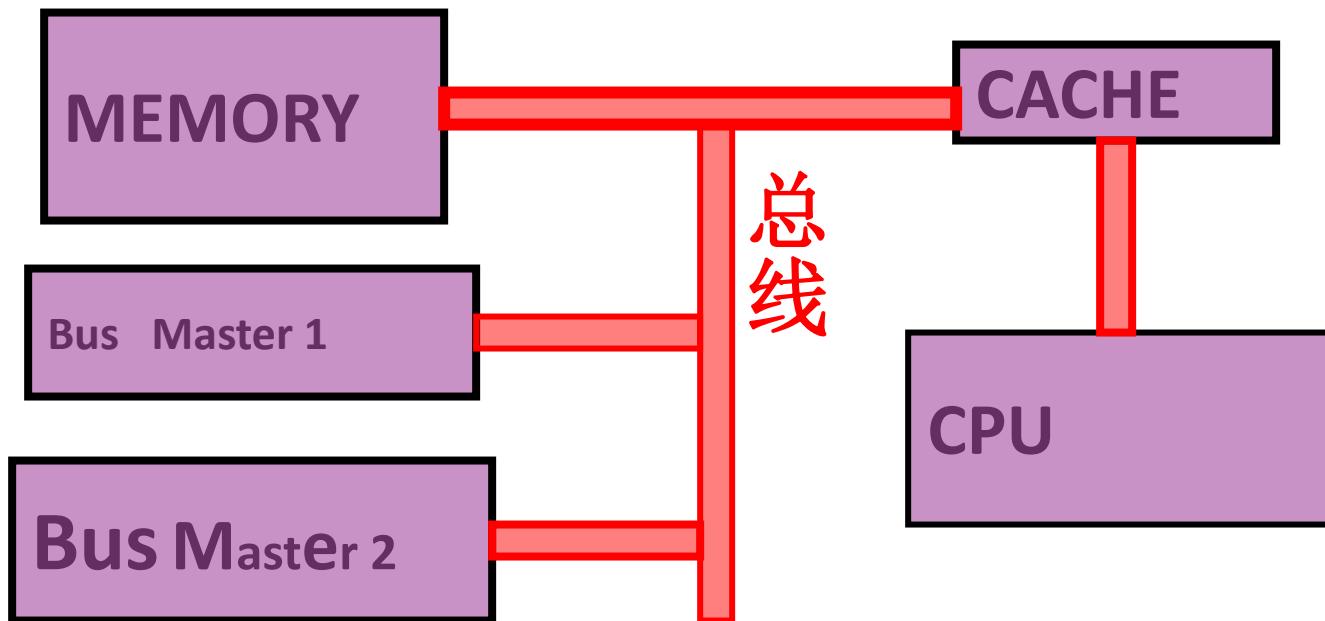
# Cache接入系统的体系结构

1. 侧接法：像插入设备似的连接到总线上，优点是结构简单，成本低，缺点是不利于降低总线占用率。



# CACHE 接入系统的体系结构

2. 隔断法：把原来的总线打断为两段，使 CACHE 处在两段之间，优点是有利于提高总线利用率，支持总线并发操作，缺点是结构复杂，成本较高。

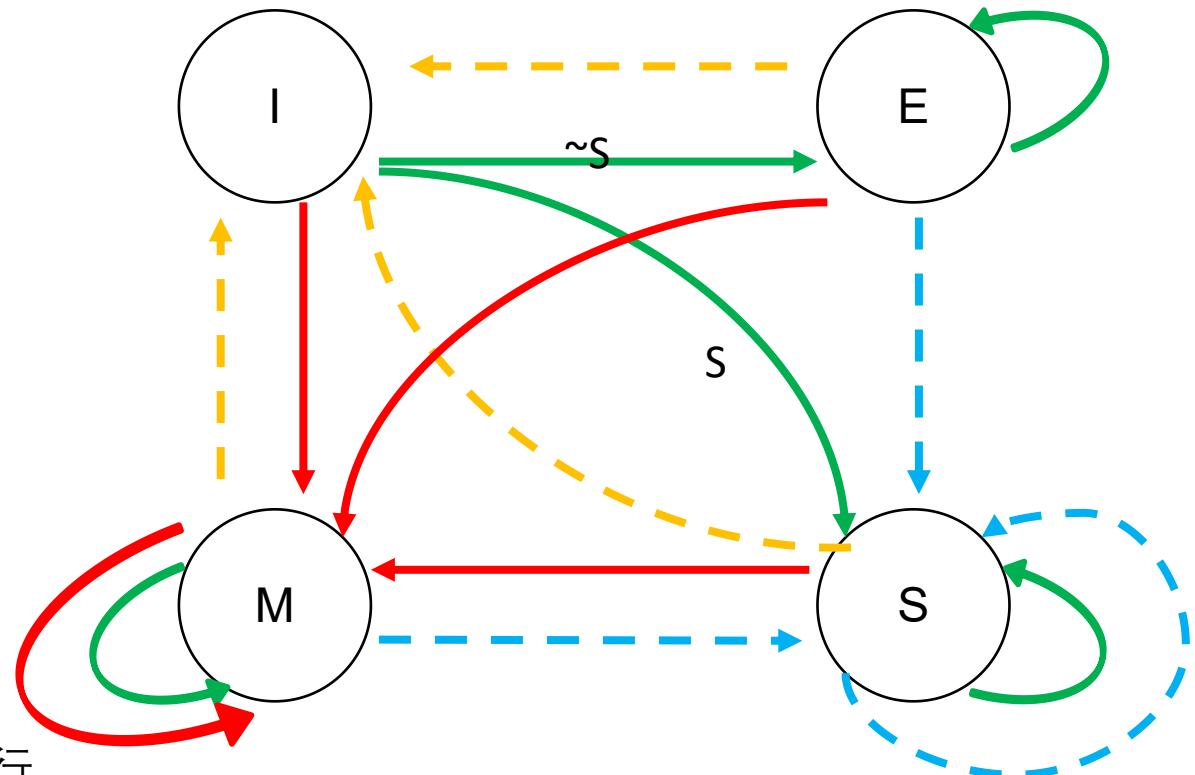


# 一致性保证策略 (MESI)

- 要保证本地cache的数据，其它核cache的数据，内存的数据有一个一致的视图
- 修改态 (M)：处于这个状态的cache块中的数据已经被修改过，和主存中对应的数据已不同，只能从cache中读到正确的数据
- 独占态 (E)：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且在其它cache中没有副本
- 共享态 (S)：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且可能在其它cache中有该块的副本
- 无效态 (I)：处于本状态的cache块中尚未装入数据

# 缓存行的状态转换

本地读  
本地写  
远程读  
远程写



S:其它Cache缓存了相同行  
 $\sim S$ : 其它Cache没有缓存

# Cache

---

## □ 目标

- 提高CPU访问存储器系统的平均速度

## □ 策略

- 利用一容量较小（降低成本）的高速缓冲存储器

## □ 组织方式

- 全相连、直接映射、组相连

# Cache

---

## □ 包含性保证

- cache中的块和主存中的块进行映射

## □ 一致性保证

- 写回主存策略

## □ 提高命中率

- 容量
- 关联方式
- 块替换算法

# Cache

## □ 局部性原理：

- 任何时候，程序需要访问的只是相对较小的一些地址空间。
  - 时间局部性
  - 空间局部性

## □ 三类主要的Cache缺失原因：

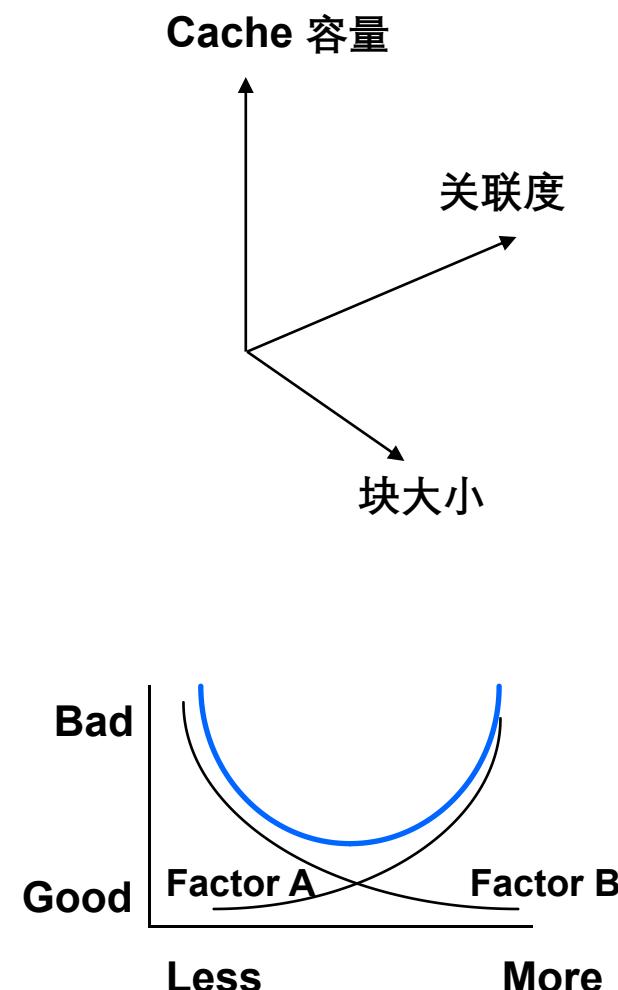
- 无法避免的缺失：如：第一次装入
- 块冲突：增大Cache容量、改进组织方式  
避免不断的块冲突
- 容量冲突：增大Cache容量

## □ Cache设计

- 总容量、块大小、组织方式
- 替换算法
- 写策略（命中时）：写直达、拖后写
- 写策略（不命中时）：是否装入到Cache？

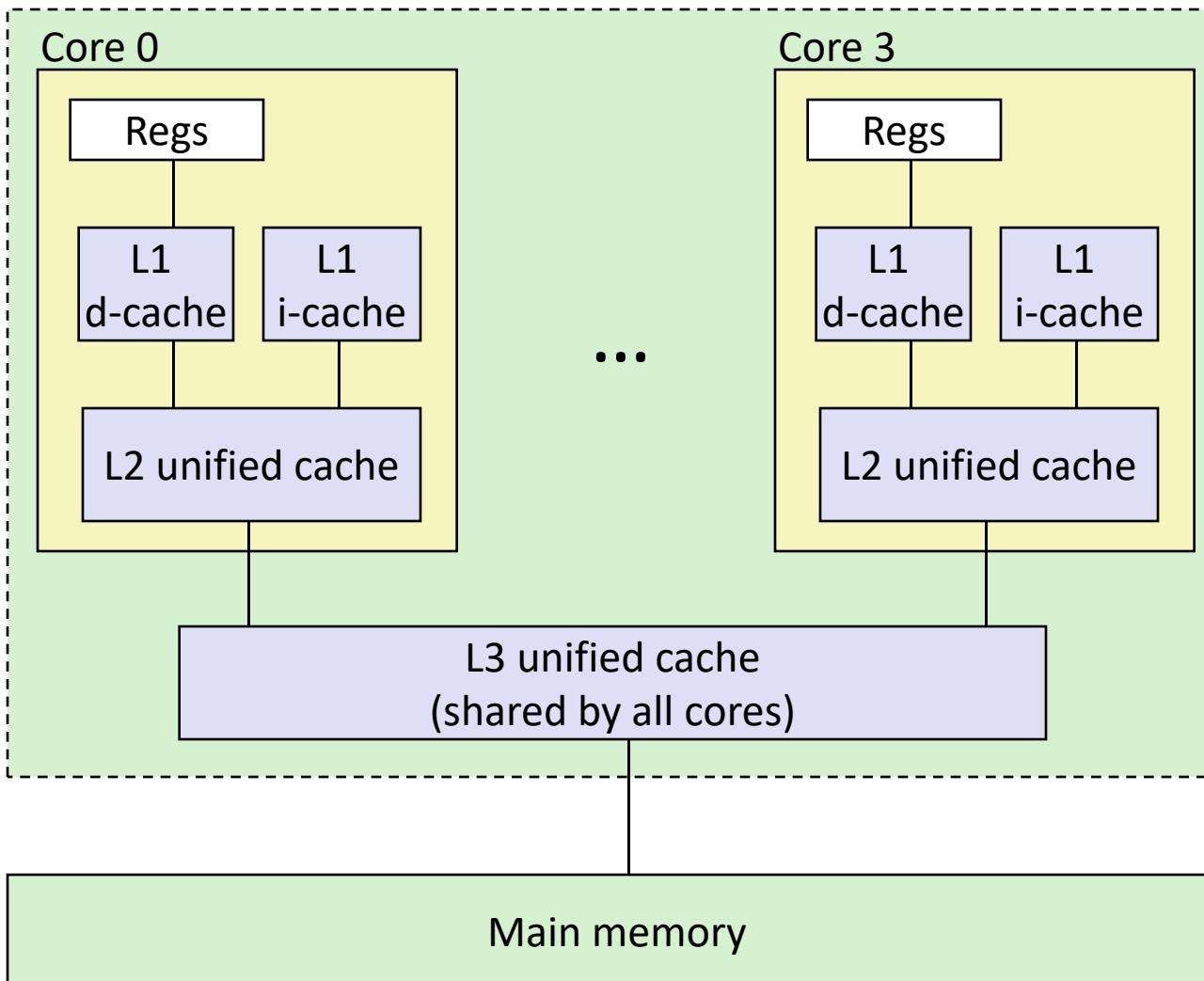
# 设计Cache

- ▶ 有关方案
  - ▶ cache 容量
  - ▶ 块大小
  - ▶ 组织方式
  - ▶ 替换算法
  - ▶ 写策略
- ▶ 方案优化
  - ▶ 根据用途选择
    - ▶ 海量数据处理
    - ▶ 指令数据平衡 (I-cache, D-cache)
  - ▶ 根据成本优化
- ▶ 简单化常常就是优化



# Intel Core i7 Cache Hierarchy

Processor package



L1 i-cache and d-cache:  
32 KB, 8-way,  
Access: 4 cycles

L2 unified cache:  
256 KB, 8-way,  
Access: 10 cycles

L3 unified cache:  
8 MB, 16-way,  
Access: 40-75 cycles

Block size: 64 bytes for  
all caches.

---

谢谢



# 虚拟内存

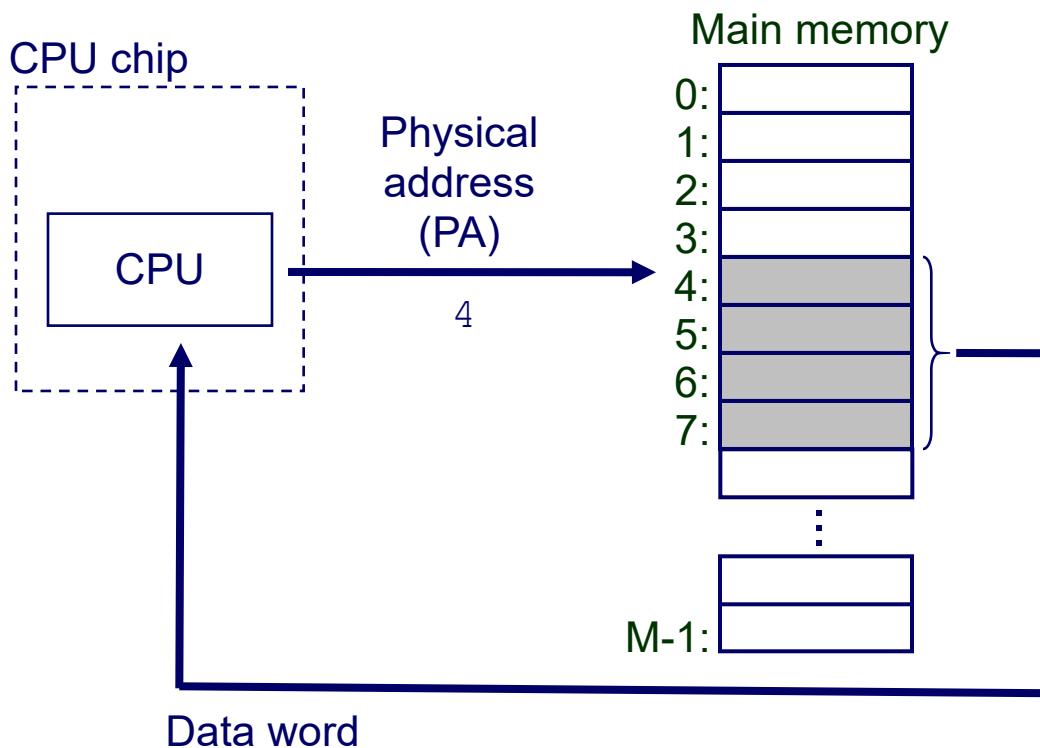
2022年秋

# 内容提要

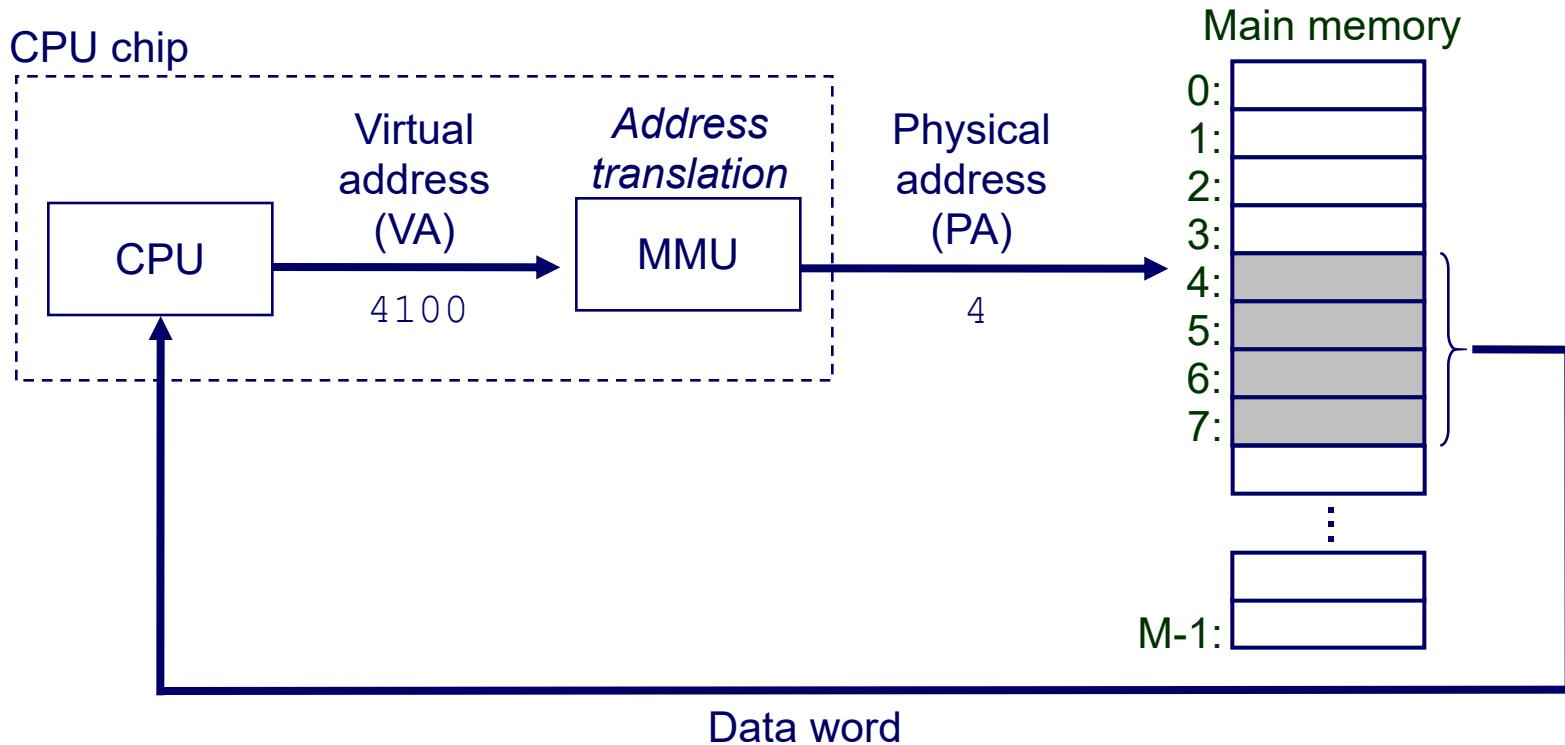
---

- 为什么需要虚拟内存？
- 页式内存管理
  - TLB
- 例子：RISC-V 页表管理
- 段式内存管理与段页式内存管理
- 例子：x86 页表管理

# 物理内存访问



# 虚拟内存访问

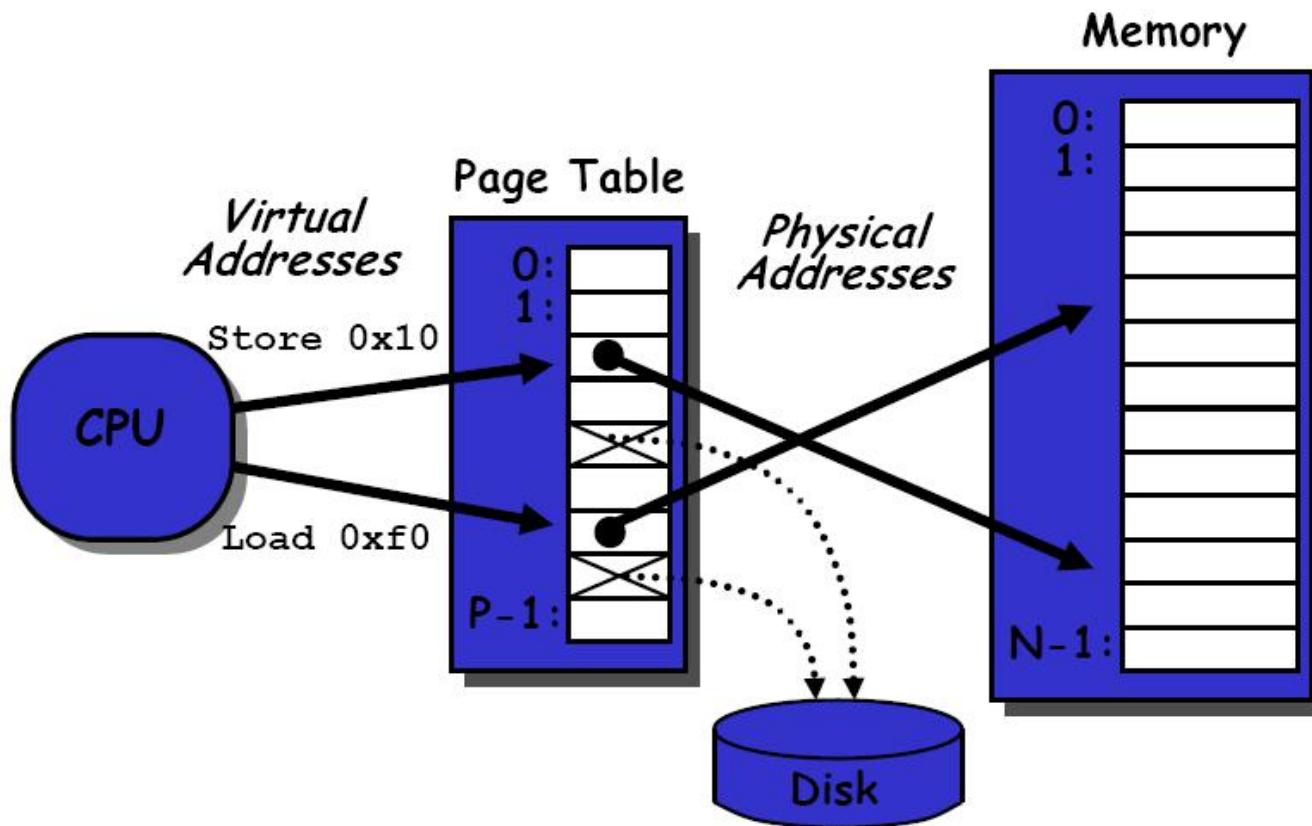


- 虚拟地址（逻辑地址）：程序员编程使用的地址
- 物理地址（实地址）：物理存储器的地址

# 两个问题

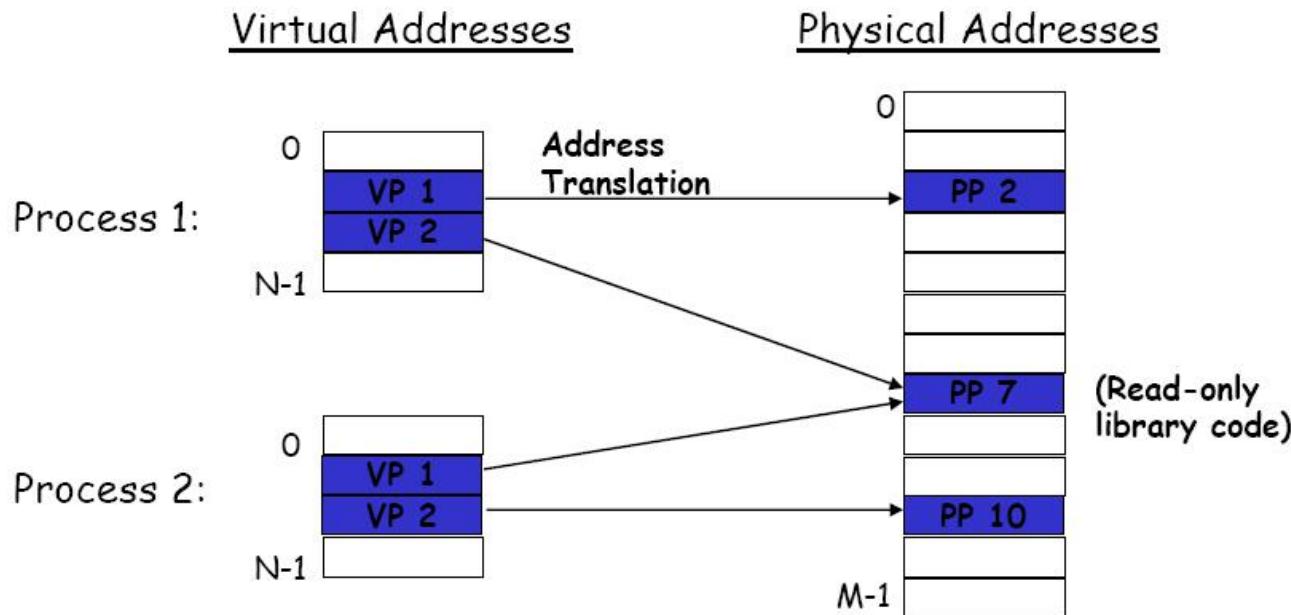
- 如果 程序的工作集大小 大于 物理内存大小， 程序还能执行吗？
  - 缓存的思想（虚拟化）
- 如果多个程序共享使用， 那么
  - 程序员在编程时， 怎么知道在哪儿分配内存？
  - 多个程序需要共享时， 怎么办？
  - 多个程序同时执行时， 某进程（程序）不想让另一个进程看到或者修改本进程的内容， 怎么办？
  - 思想： 程序员编程使用的空间与程序运行空间相互独立

# (1) 独立的逻辑地址空间



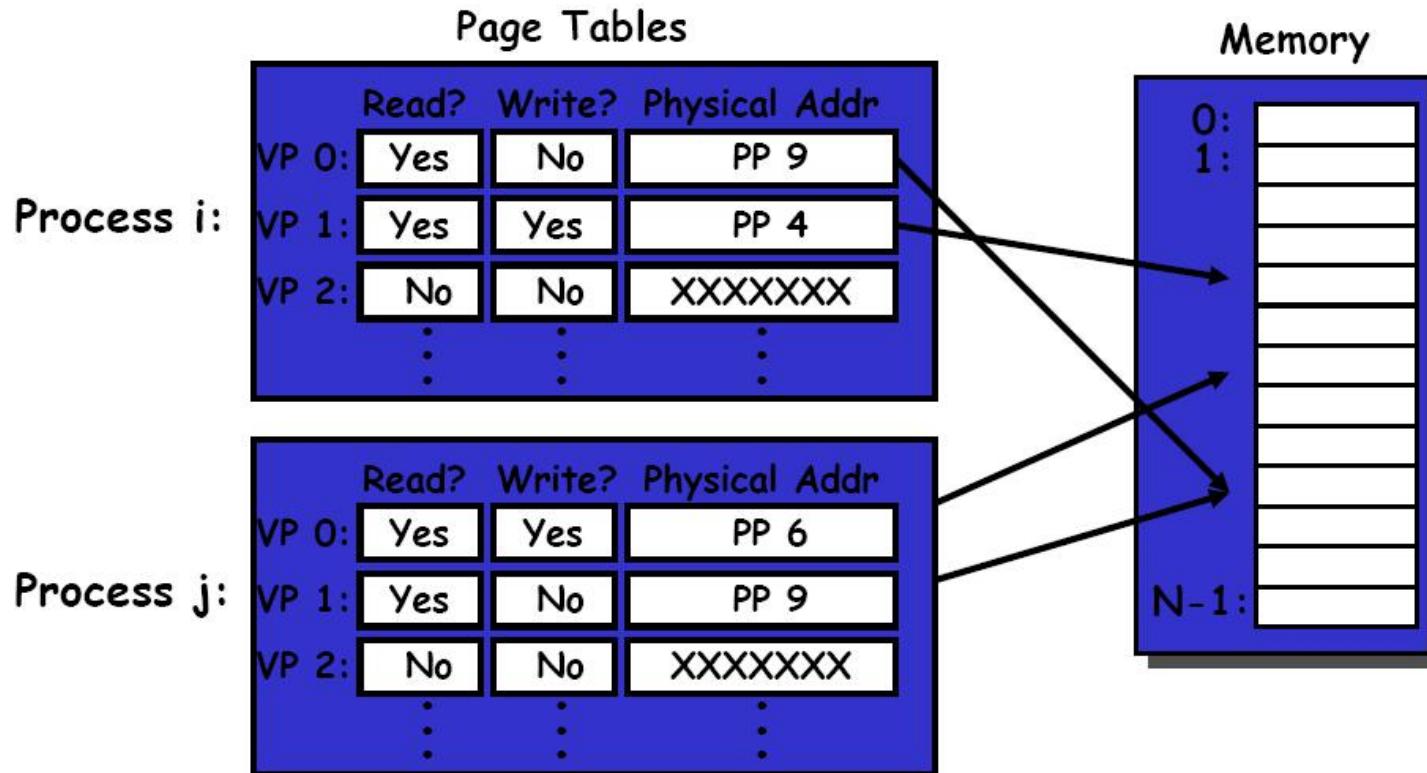
□ 通过页表将虚地址转换为实地址

## (2) 实现内存共享



- 每个进程有独立的逻辑地址空间
- 建立逻辑地址和物理地址的转换机制

### (3) 实现内存的保护



□ 页表中存放有访问权限

- 通过硬件来保证权限（操作系统的“陷阱”操作）

# 虚拟存储器的目的

## □ 容量

- 获得运行比物理存储器更大空间程序的能力

## □ 存储管理

- 内存的分配以及虚实地址转换

## □ 保护

- 操作系统可以对虚拟存储空间进行特定的保护...

## □ 灵活

- 程序的某部分可以装入主存的任意位置

## □ 提高存储效率

- 只在主存储器中保留最重要的部分

## □ 提高并行度

- 在进行段页替换的同时可以执行其它进程

## □ 可扩展

- 为对象提供了扩展空间的能力.

# 虚拟内存与高速缓存的比较

|           | 虚拟内存                              | 高速缓存                                 |
|-----------|-----------------------------------|--------------------------------------|
| 层次        | “内存-外存”层次                         | “CPU缓存-内存”层次                         |
| 主要目的      | 解决存储容量的问题                         | 解决主存储器与CPU性能的差距                      |
| 数据交换粒度与频次 | 单位时间内数据交换次数较少，但每次交换的数据量大，达几十至几千字节 | 单位时间内数据交换的次数较多，每次交换的数据量较小，只有几个到几十个字节 |
| 实现主体      | 主要由操作系统管理                         | 由硬件实现                                |

# 内容提要

---

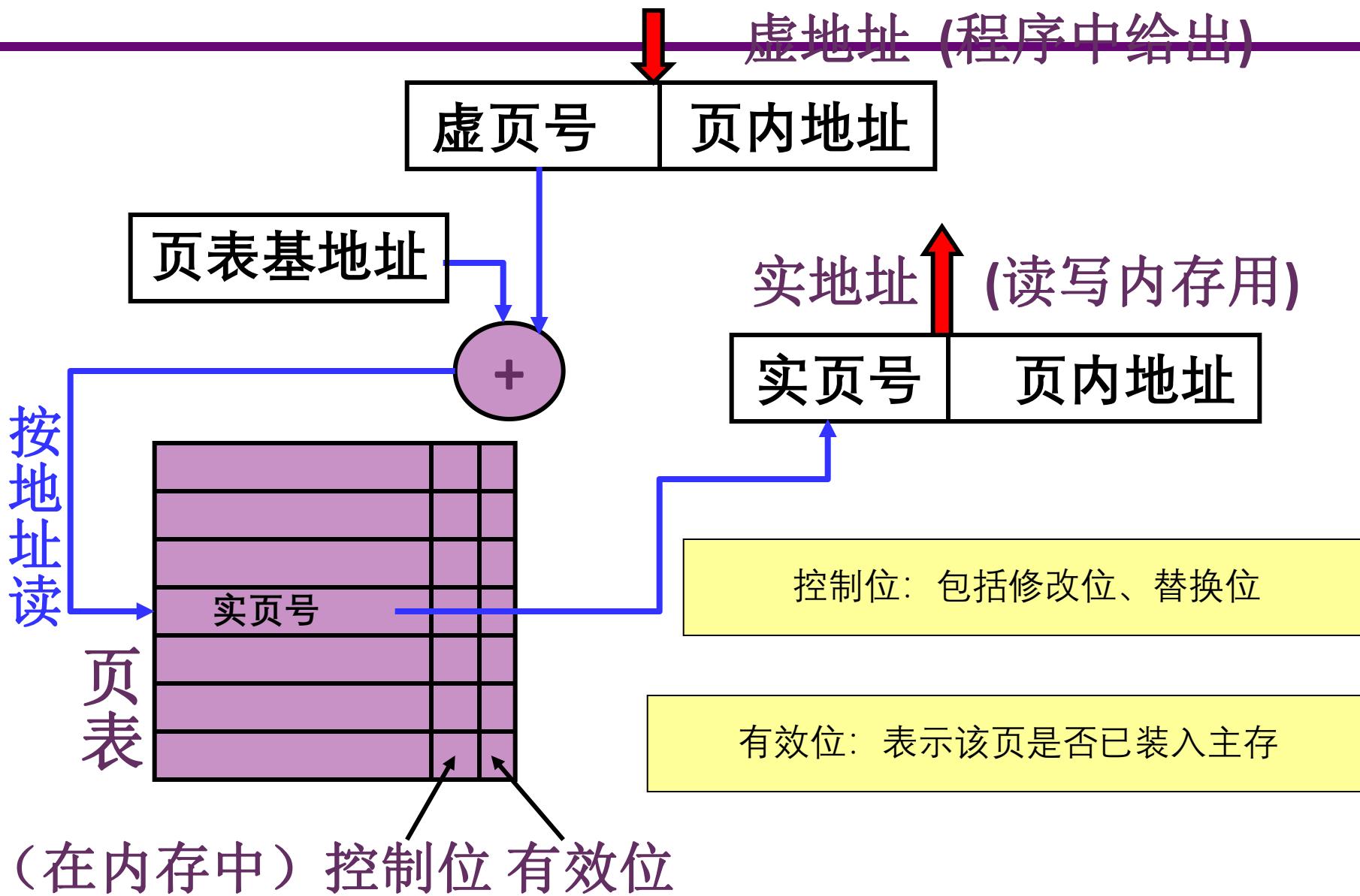
- 为什么需要虚拟内存？
- 页式内存管理
  - TLB
- 例子：RISC-V 页表管理
- 段式内存管理与段页式内存管理
- 例子：x86 页表管理

# 页式存储管理

---

- 将主存和虚存划分为固定大小的页
- 以页为单位进行管理和数据交换
- 虚地址=虚页号+页内地址
- 实地址=实页号+页内地址
- 通过页表进行管理
  - 页表基址寄存器
  - 实页号
  - 控制位

# 页表内容和页式管理



# 页表大小

## □ 与虚页数直接相关，但是

- 虽然理论上每个进程的逻辑空间很大，但其实大部分应该是不活跃的
- 实际调入到内存的内容不可能超过物理存储空间

## □ 如何减少页表本身所占的空间？

- 而且还要实现简单
  - 页表访问频繁

## □ 两种途径

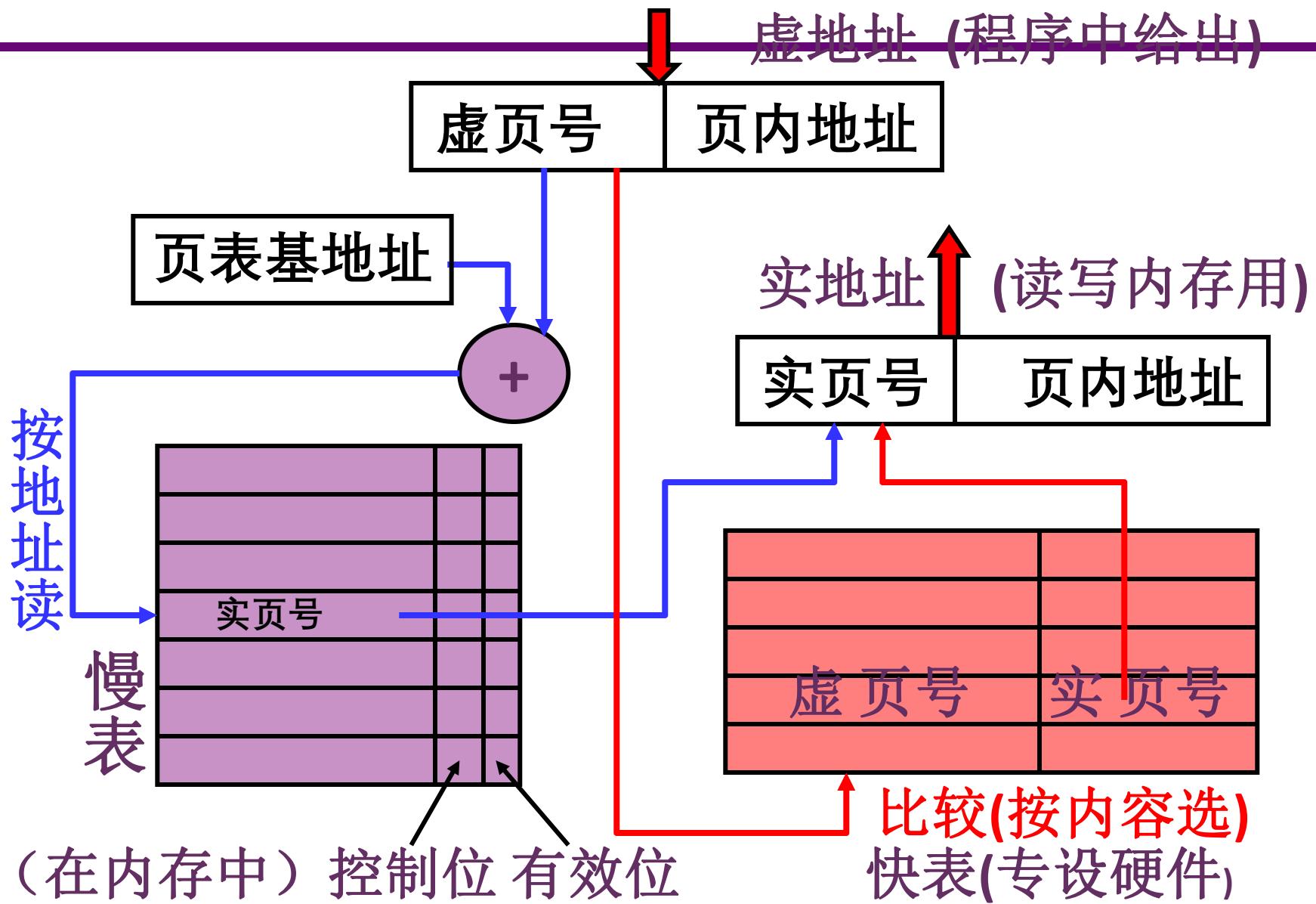
- 层次页表 (hierarchical page table)
- 反转页表 (inverted page table)

# 页式虚拟存储器的访问过程

- 1. 得到程序给出的虚地址；
- 2. 由虚地址得到虚页号；
- 3. 访问页表，得到对应的实页号；
- 4. 若该页已在内存中，则根据实页号得到实地址，访问内存；
- 5. 否则，启动输入输出系统，读出对应页装入主存，再进行访问。

增加由硬件实现的快表，提高访问速度

# 页表内容和页式管理



# 转换旁路缓冲 (TLB)

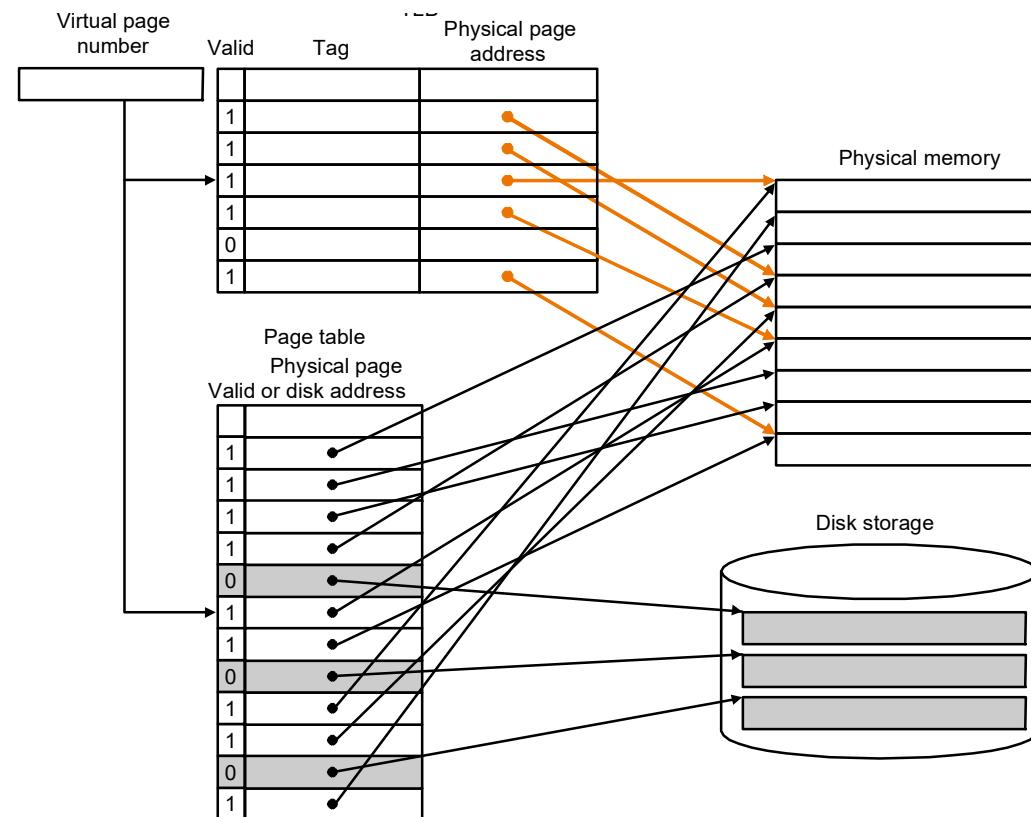
□ 访问频繁:速度是第一位的

□ TLB 缺失将造成:

- 流水线停止
- 通知操作系统
- 读页表
- 将表项写入 TLB
- 返回到用户程序
- 重新访问

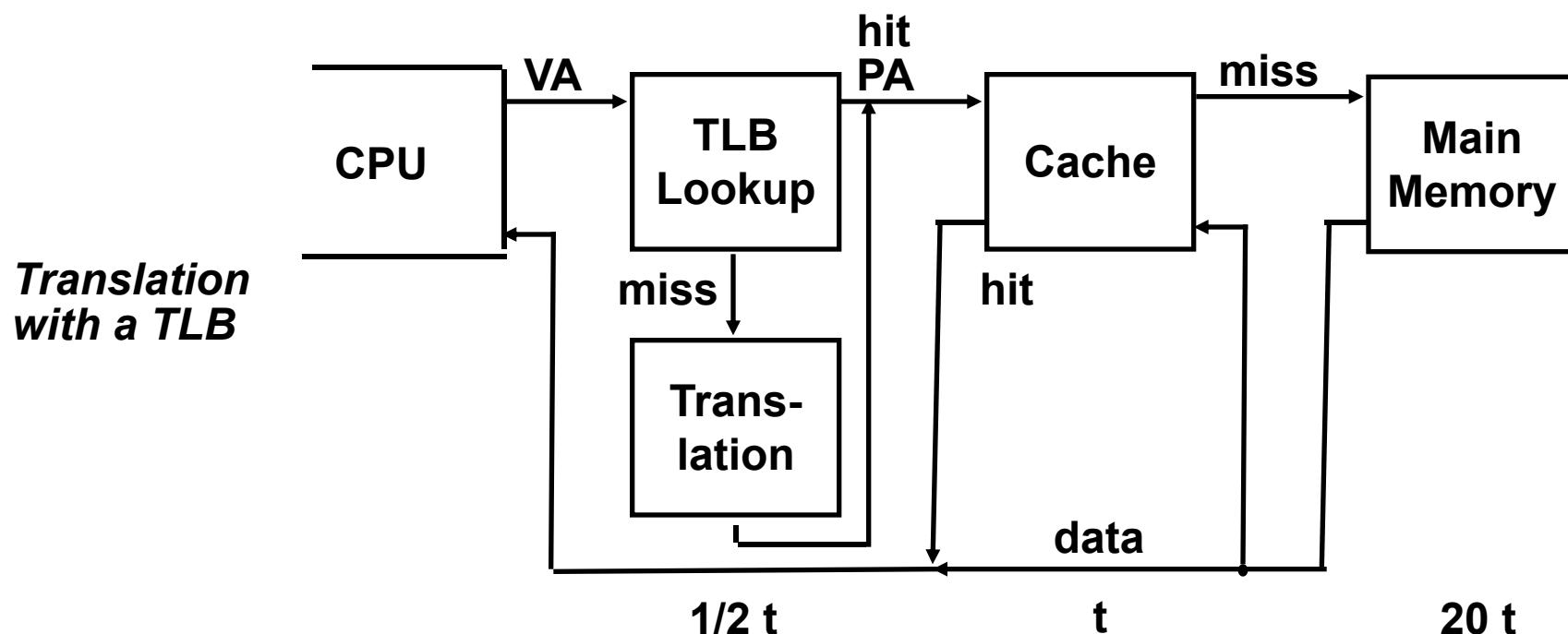
□ 因此, 应尽量减少缺失:

- 多路组相连
- 再尽量提高TLB的容量



# 转换旁路缓冲 (TLB)

- TLB 可以组织成全相连，组相连或直接映射方式
- TLBs 通常为容量较小，甚至在高端计算机上也一般不超过128 – 256 个表项。这样，可以使用全相连映射方式。在大多数中档计算机上，一般采用N路组相联映射方式。



# 页面大小的选择

## □ 减少内部碎片

- 缩小页面大小可以减少内部碎片
- 但是：需要更大的页表

## □ 趋势：增大页面大小

- RAM价格下降，内存储器容量增大
- 内存和外存性能差距增大
- 程序员需要更大的地址空间

## □ 目前：页面大小为4K左右？（1MB, 2MB, 4MB, 1GB）

- Linux内核的Huge Page机制

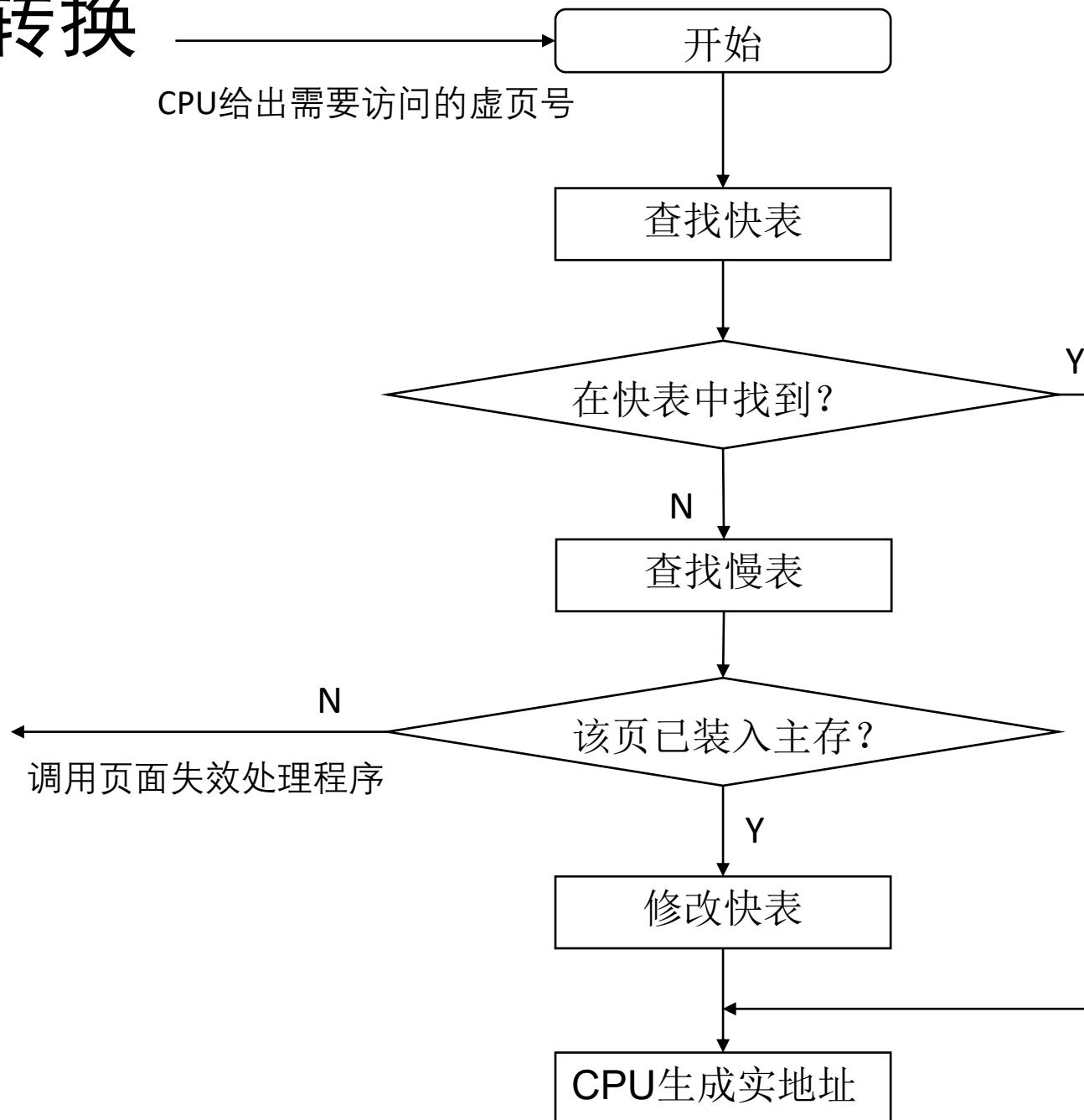
# 页面替换算法

---

## □ 最近最少使用 (LRU)

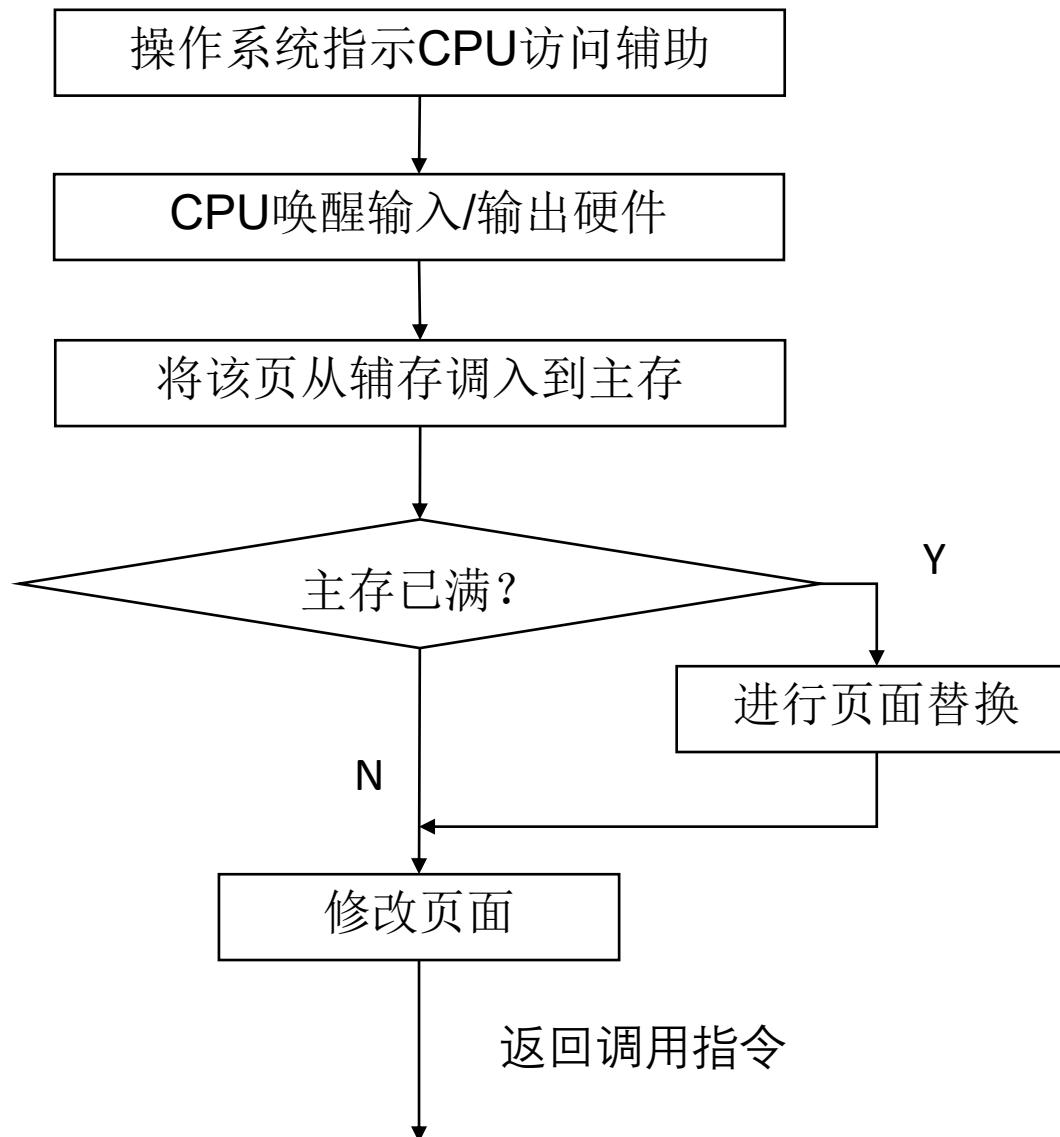
- 将页帧按照最近最多使用到最近最少使用进行排序，再次访问一个页帧时，将该页帧移到表头，替换时将表尾的页帧换出。
- 一点改进：替换出其中一个“干净”的页帧。

# 地址转换



# 页失效处理

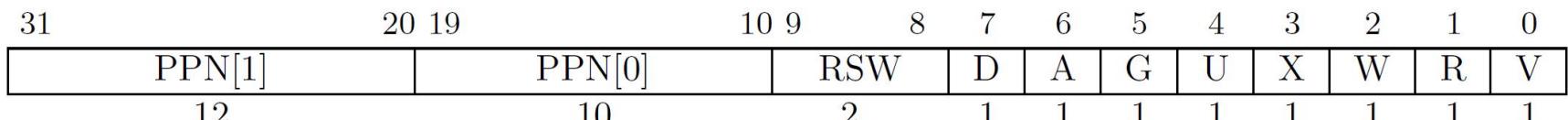
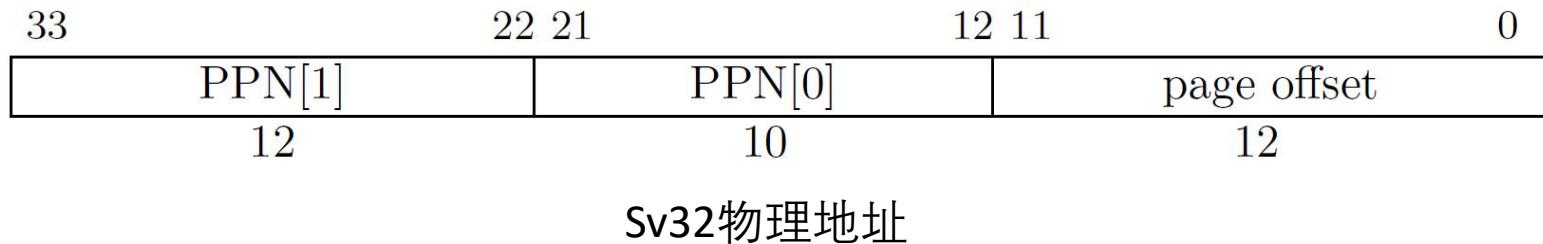
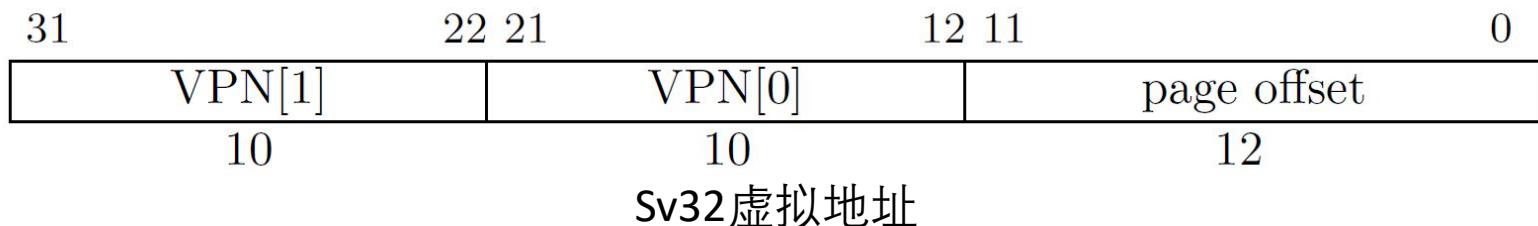
## 页面失效处理程序



# 基于页面的虚拟内存

- 分页命名模式：SvX，其中X是以位为单位的虚拟地址长度
- 内存划分为固定大小的页面进行地址转换和对内存内容的保护（页面大小通常为4KB，也有大页面粒度）
- 启用分页的时候，大多数地址（包括load和store的有效地址和PC中的地址）都是虚拟地址
- 要访问物理内存，虚拟地址必须被转换为真正的物理地址
- 通过页表的结构来进行转换
- 权限位指示那些权限模式和通过哪种类型的访问可以操作这个页
- 访问未被映射的页或者访问权限不足会导致页面错误异常（page fault exception）

# Sv32的虚拟地址与物理地址



# RV32 Sv32页表项 (PTE: page table entry)

|        |        |      |   |   |   |   |   |   |   |   |   |
|--------|--------|------|---|---|---|---|---|---|---|---|---|
| 31     | 20 19  | 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PPN[1] | PPN[0] | RSW  | D | A | G | U | X | W | R | V |   |
| 12     | 10     | 2    | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |   |

- V: 有效位
- R,W,X: 读, 写, 执行位
- U: 0代表U模式不能访问, 但是S模式可以; 1代表U模式可以访问, S模式可以
- G: Global是否对所有地址空间有效
- A: Access, 是否被访问过
- D: Dirty, 是否被修改过
- RSW: 操作系统使用, 被硬件忽略
- PPN: 物理页号, 这是物理地址的一部分。如果这个页表项是一个叶子结点, 那么PPN是转换后物理地址的一部分。否则PPN给出的是下一级页表的地址

# Sv39的虚拟地址与物理地址

|        |        |        |             |   |
|--------|--------|--------|-------------|---|
| 38     | 30 29  | 21 20  | 12 11       | 0 |
| VPN[2] | VPN[1] | VPN[0] | page offset |   |
| 9      | 9      | 9      | 12          |   |

Sv39虚拟地址

|        |        |        |             |   |
|--------|--------|--------|-------------|---|
| 55     | 30 29  | 21 20  | 12 11       | 0 |
| PPN[2] | PPN[1] | PPN[0] | page offset |   |
| 26     | 9      | 9      | 12          |   |

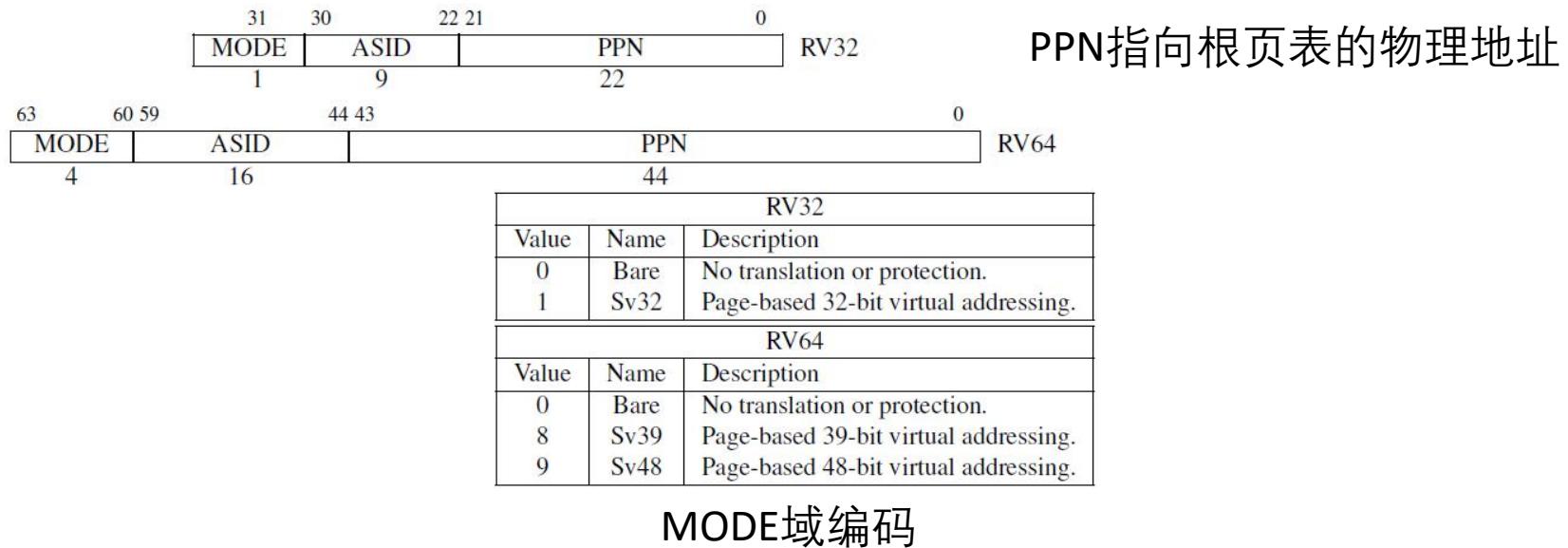
Sv39物理地址

|          |        |        |        |      |   |   |   |   |   |   |   |   |   |
|----------|--------|--------|--------|------|---|---|---|---|---|---|---|---|---|
| 63       | 54 53  | 28 27  | 19 18  | 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | PPN[2] | PPN[1] | PPN[0] | RSW  | D | A | G | U | X | W | R | V |   |
| 10       | 26     | 9      | 9      | 2    | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Sv39页表项

# satp寄存器

- satp (Supervisor Address Translation and Protection, 监管者地址转换和保护) S 模式控制状态寄存器控制了分页系统
- ASID: Address Space Identifier, 地址空间标识符(可选), 用以降低上下文切换开销



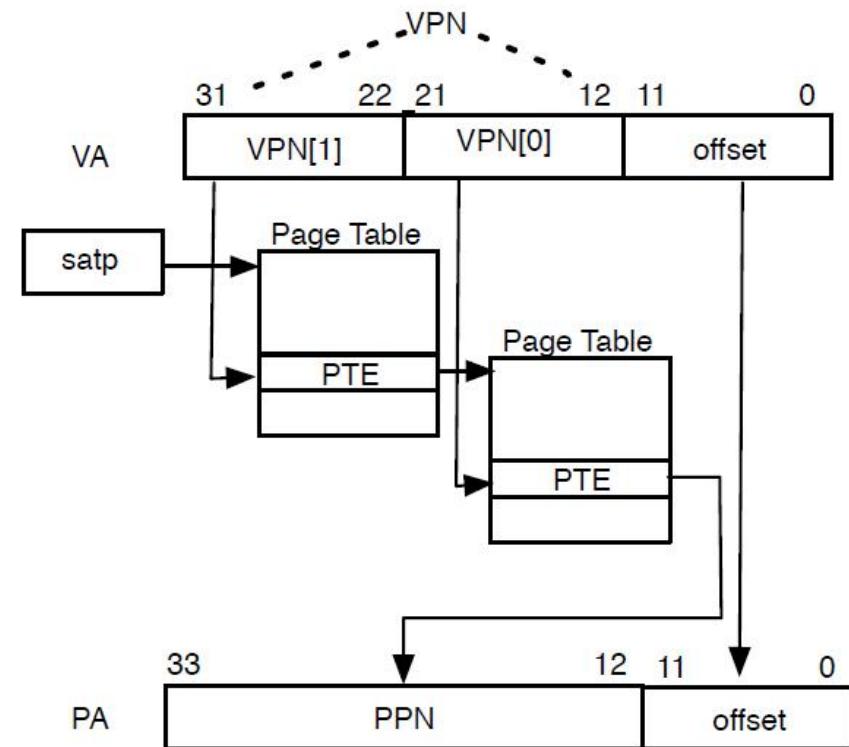
# satp寄存器初始化

---

- M模式的程序在第一次进入S模式之前会把0写入 satp，以禁用分页
- 然后S模式的程序在初始化页表以后会再次进行satp 寄存器的写操作

# 虚拟地址到物理地址的转换

- satp.PPN 给出了一级页表的基址，VA[31:22]给出了一级页号，因此处理器会读取位于地址( $\text{satp}.PPN \times 4096 + \text{VA}[31:22] \times 4$ )的页表项。
- 该 PTE 包含二级页表的基址，VA[21:12]给出了二级页号，因此处理器读取位于地址( $\text{PTE}.PPN \times 4096 + \text{VA}[21:12] \times 4$ )的叶节点页表项。
- 叶节点页表项的 PPN 字段和页内偏移（原始虚址的最低 12 个有效位）组成了最终结果：物理地址就是( $\text{LeafPTE}.PPN \times 4096 + \text{VA}[11:0]$ )



# 虚拟地址到物理地址的转换

1. Let  $a$  be  $\text{satp}.ppn} \times \text{PAGESIZE}$ , and let  $i = \text{LEVELS} - 1$ .
2. Let  $pte$  be the value of the PTE at address  $a + va.vpn[i] \times \text{PTESIZE}$ .
3. If  $pte.v = 0$ , or if  $pte.r = 0$  and  $pte.w = 1$ , stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If  $pte.r = 1$  or  $pte.x = 1$ , go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let  $i = i - 1$ . If  $i < 0$ , stop and raise a page-fault exception. Otherwise, let  $a = pte.ppn} \times \text{PAGESIZE}$  and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the  $pte.r$ ,  $pte.w$ ,  $pte.x$ , and  $pte.u$  bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If  $i > 0$  and  $pa.ppn[i - 1 : 0] \neq 0$ , this is a misaligned superpage; stop and raise a page-fault exception.
7. If  $pte.a = 0$ , or if the memory access is a store and  $pte.d = 0$ , then either:
  - Raise a page-fault exception, or:
  - Set  $pte.a$  to 1 and, if the memory access is a store, also set  $pte.d$  to 1.
8. The translation is successful. The translated physical address is given as follows:
  - $pa.pgoff = va.pgoff$ .
  - If  $i > 0$ , then this is a superpage translation and  $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$ .
  - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$ .

虚址到物理地址转换的完整算法。va 是输入的虚拟地址，pa 是输出的物理地址。PAGESIZE 是常数  $2^{12}$ 。在 Sv32 中，LEVELS = 2 且 PTESIZE = 4；而在 Sv39 中，LEVELS = 3 且 PTESIZE = 8

# TLB

- 如果取指，load，store要访问多次页表，将会大大降低性能
- 所有的现代处理器都使用地址转换缓存（TLB：Translation Lookaside Buffer）来减少这种开销
- 如果操作系统修改了页表，TLB就会变得不可用
- sfence.vma通知处理器，软件可能已经修改了页表，处理器可以刷新TLB
  - rs1指示哪个虚拟地址对应的转换被修改了
  - rs2指示被修改页表的地址空间标识符（一般相当于进程）ASID
  - 如果两者都是x0，整个TLB会被刷新

# 内容提要

---

- 为什么需要虚拟内存？
- 页式内存管理
  - TLB
- 例子：RISC-V 页表管理
- 段式内存管理与段页式内存管理
- 例子：x86 页表管理

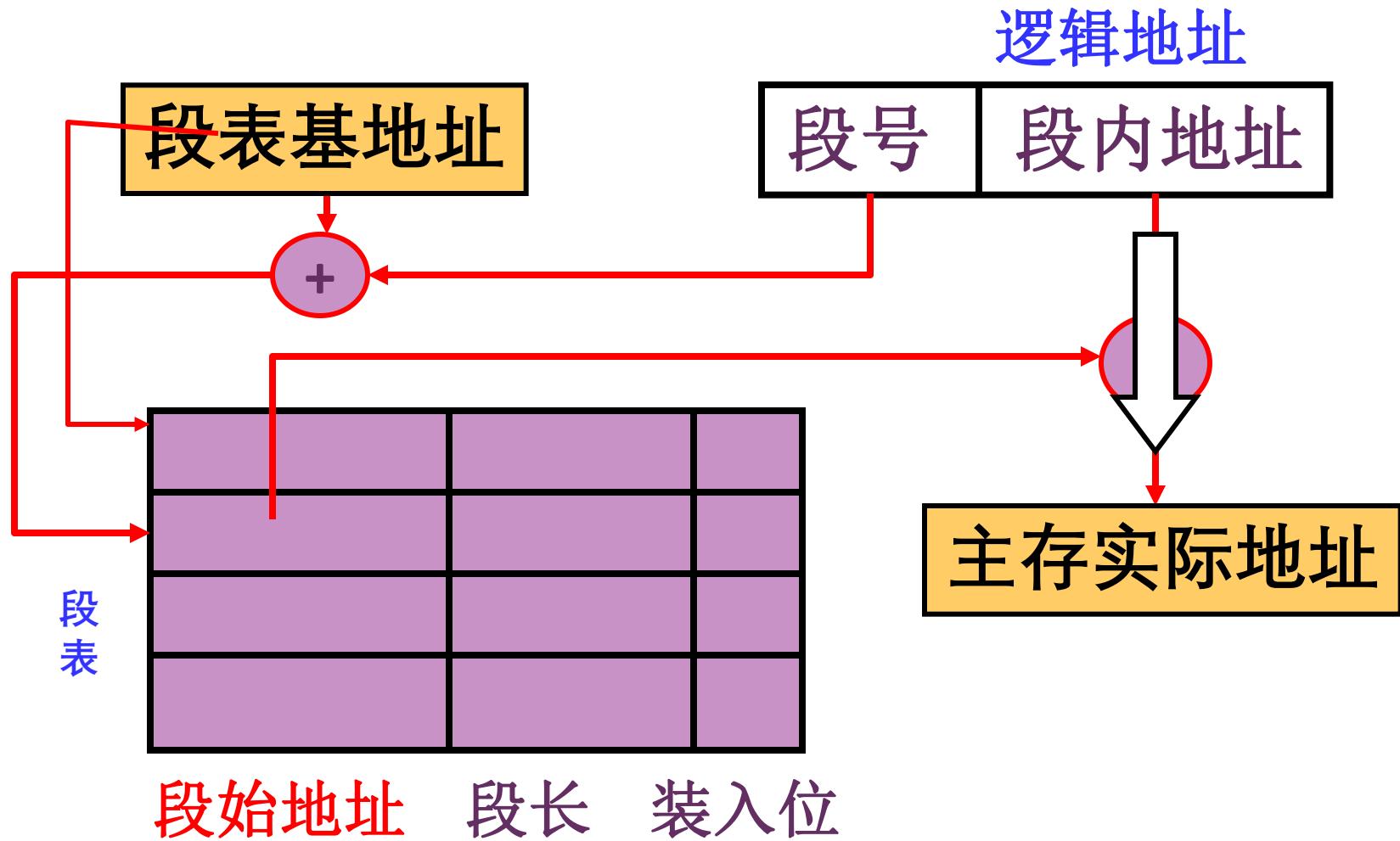
# 段式存储管理的实现

---

## □ 设置段表进行管理

- 段表基地址
- 段起始地址
- 段长
- 装入位
- 保护、共享等标志

# 段式管理地址转换

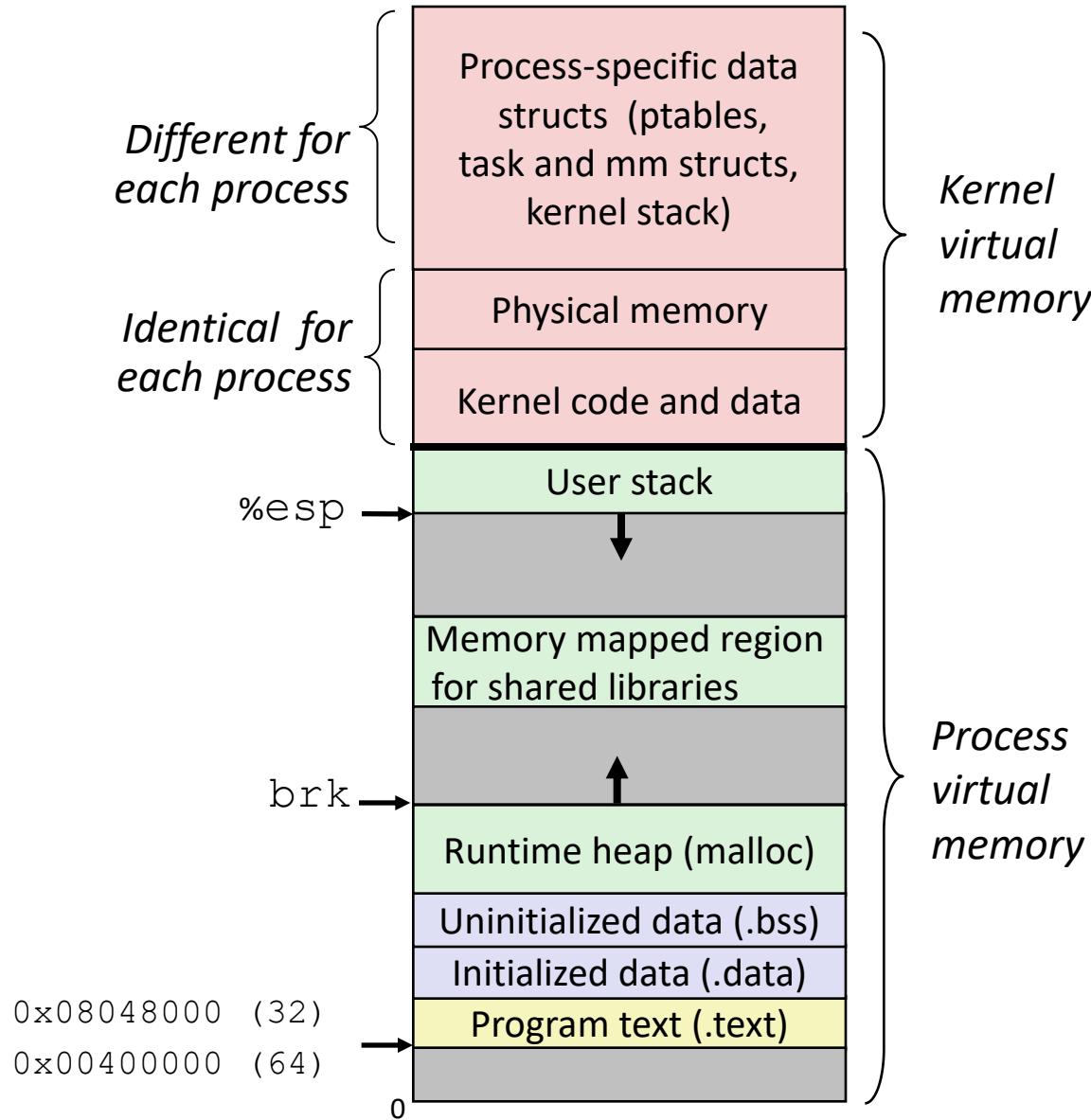


# 段式存储管理

---

- 段的分界与程序和数据的自然分界相对应
- 易于编译、管理、修改和保护，便于多道程序共享
- 段长动态可变（？）
- 段起点、终点不定（？）
- 空间分配困难，容易产生碎片

# Virtual Memory of a Linux Process



# 段页式虚拟存储管理

□ 是段式虚拟存储器和页式虚拟存储器的综合。它先把程序按逻辑单位分为段，再把每段分成固定大小的页。操作系统对主存的调入调出是按页面进行的，但它又可以按段实现共享和保护，可以兼取页式和段式系统的优点。其缺点是需要在地址映射过程中多次查表。其地址映射通过一个段表和一组页表来进行。

# x86的虚存管理

- 不分段也不分页模式：在这种模式下，虚拟存储的地址空间和物理存储空间大小相同，可以用在复杂度较低但对性能有较高要求的场合。
- 页式管理模式：这种模式将主存分成固定长度的页，通过页进行存储保护和管理。
- 段式管理模式：段式管理模式按程序本身的逻辑段来划分主存空间，与页式管理相比，段的长度可变
- 段页式管理模式：为了兼容旧的模式，在x86中段式内存管理和页式内存管理都是支持的。程序地址首先经过段式内存转换，再通过页式内存转换，最终转换为物理地址。

# 32位x86虚实地址的转换

## □ 虚地址（逻辑地址）：

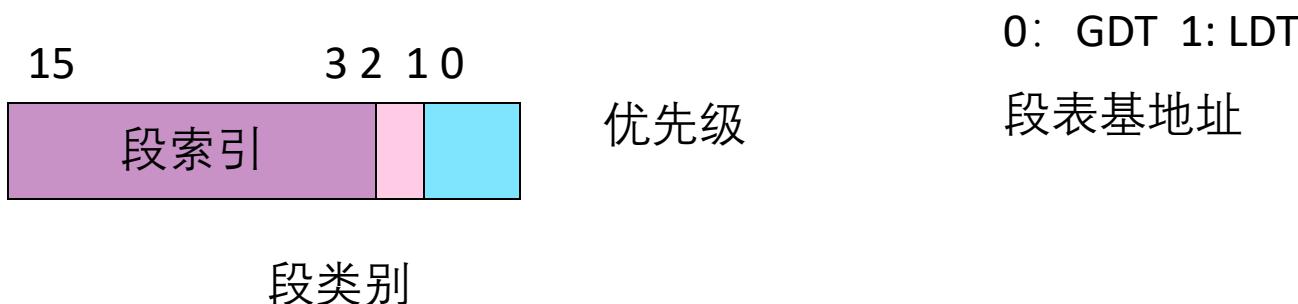
- 程序员给出的虚拟地址，格式为段号+段内偏移（16位+32位），每段大小不超过4GB，一共不超过 $2^{14}$ 段。（段号中有两位用来表示段优先级）

## □ 实地址：

- 32位的实际内存地址。

# 段号和段表的格式

段号：

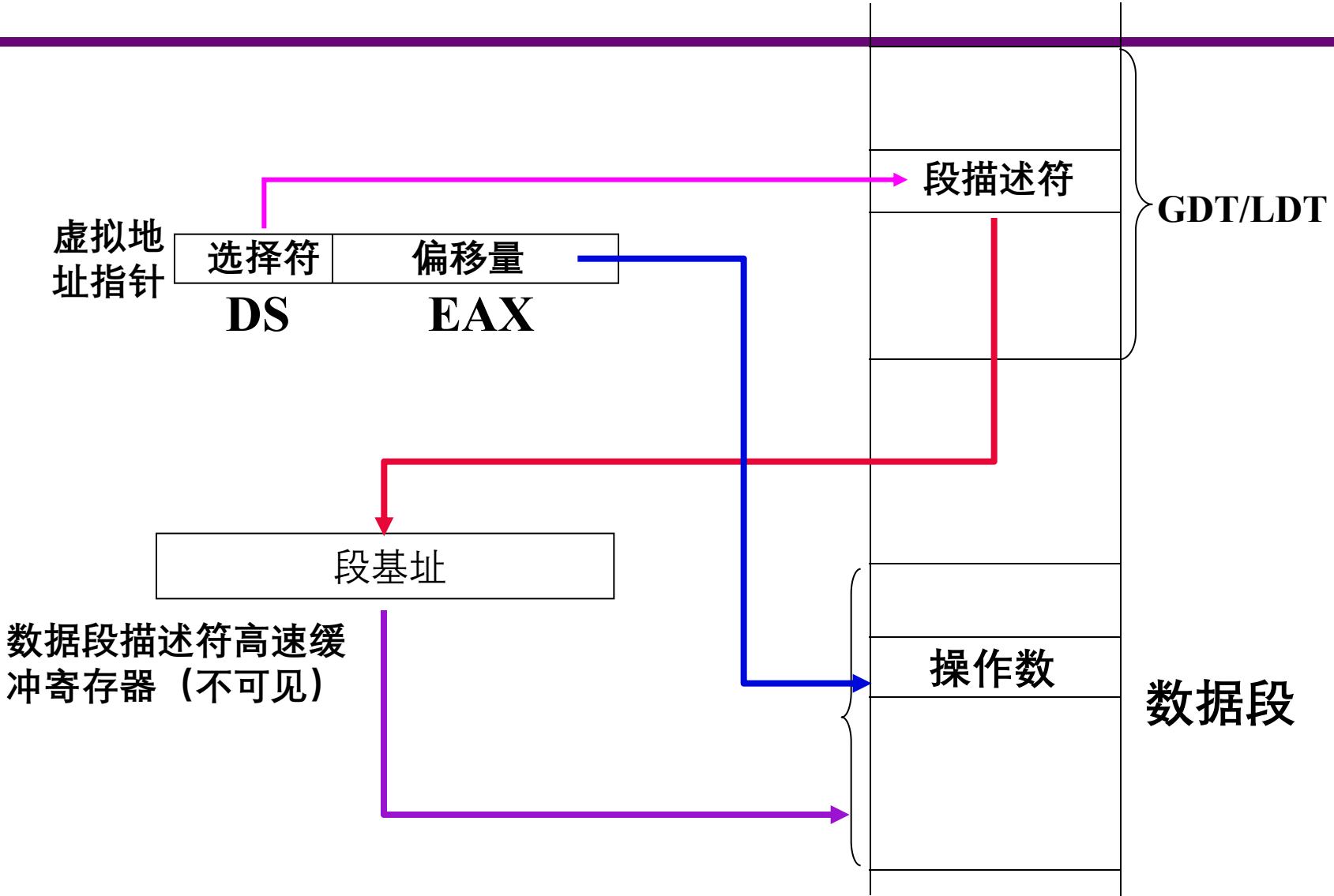


段表：

|                |   |             |  |                         |  |             |
|----------------|---|-------------|--|-------------------------|--|-------------|
| BASE<br>31..24 | G | D<br>/<br>B |  | Segment limit<br>19..16 |  | BASE 23..16 |
| BASE 15..0     |   |             |  | Segment Limit 15..0     |  |             |

线性地址=BASE + OFFSET

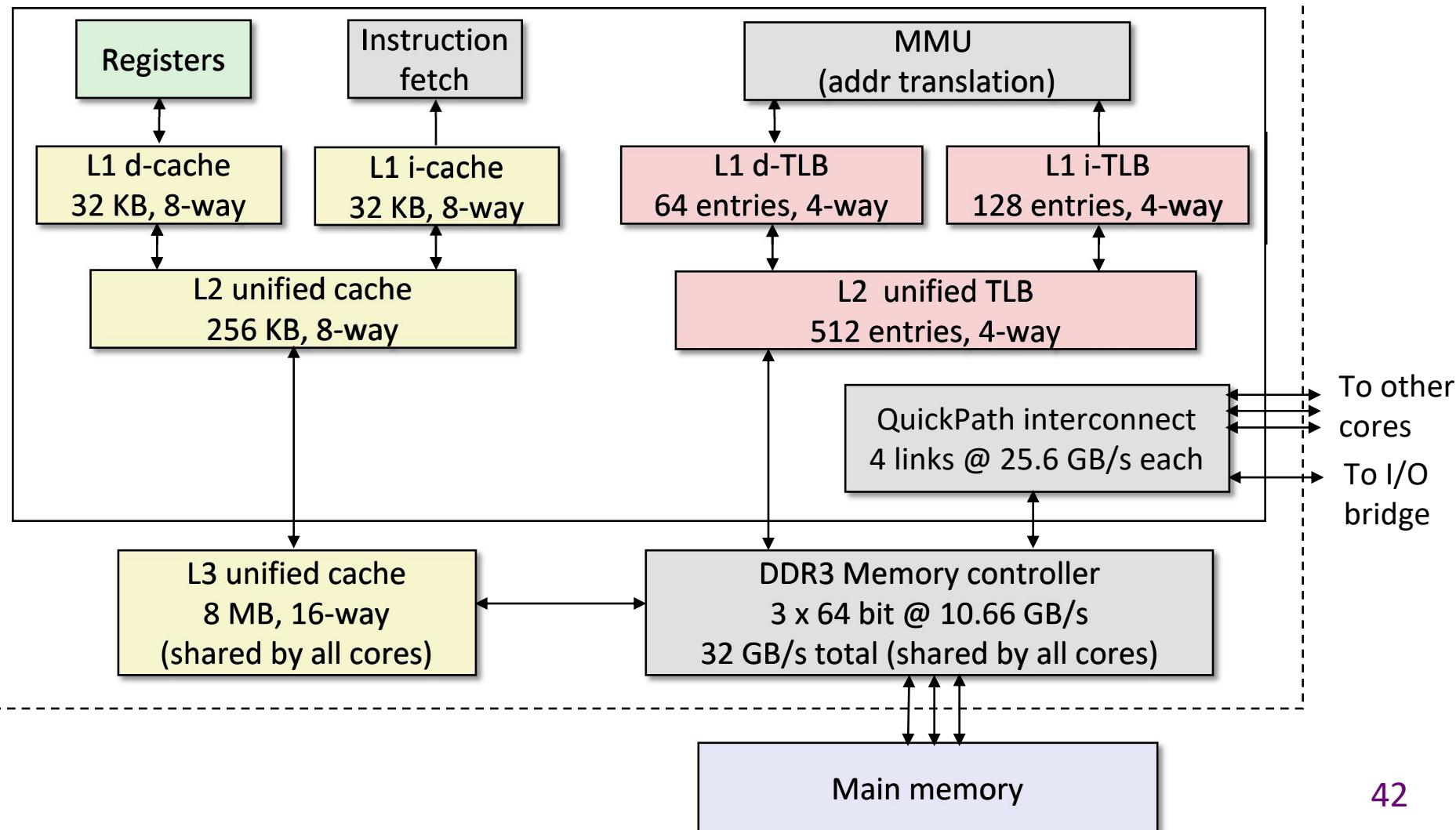
# 段地址转换



# Intel Core i7 Memory System

Processor package

Core x4



# 符号解释

## ■ 基本参数

- $N = 2^n$ : 虚拟地址空间中的地址数量
- $M = 2^m$ : 物理地址空间中的地址数量
- $P = 2^p$  : 页大小 (bytes)

## ■ 虚拟地址的组成部分 (VA)

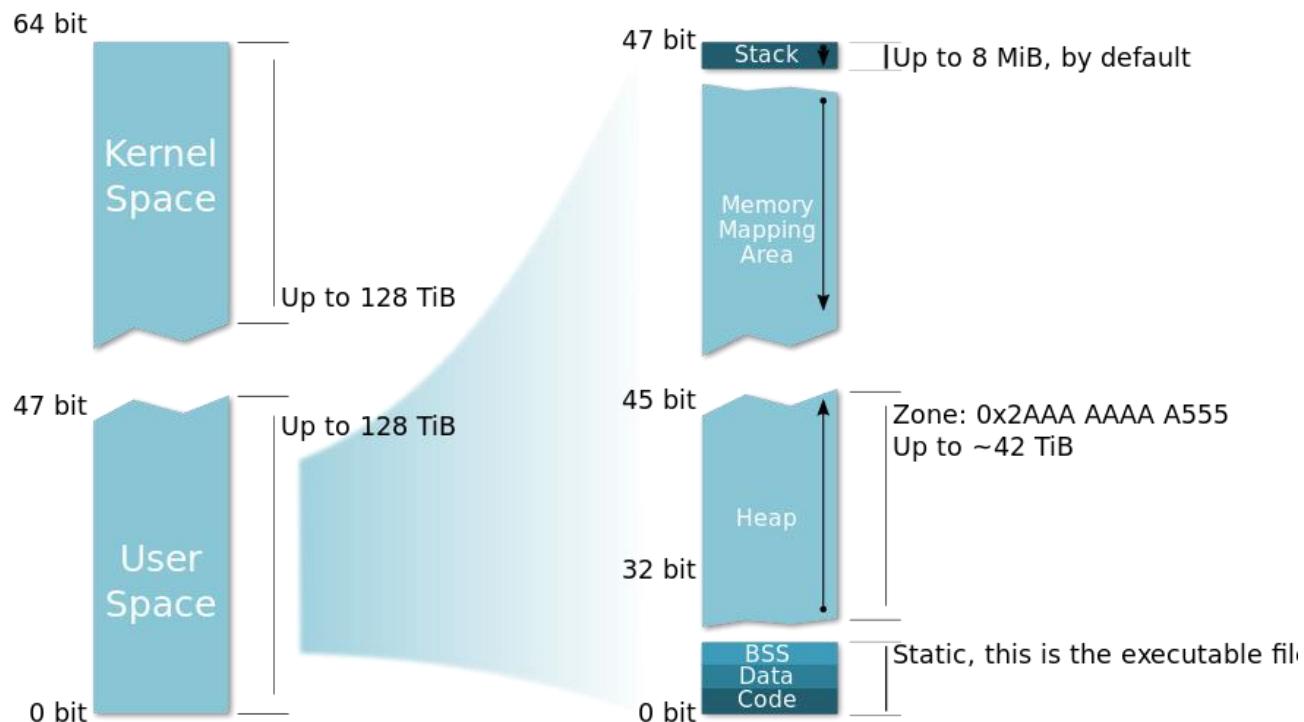
- TLBI: TLB index 索引
- TLBT: TLB tag 标记
- VPO: Virtual page offset 虚拟页偏移
- VPN: Virtual page number 虚拟页号

## ■ 物理地址的组成部分 (PA)

- PPO: Physical page offset 物理页偏移 (同 VPO)
- PPN: Physical page number 物理页号
- CO: Byte offset within cache line 缓存行内偏移
- CI: Cache index 缓存索引
- CT: Cache tag 缓存标记

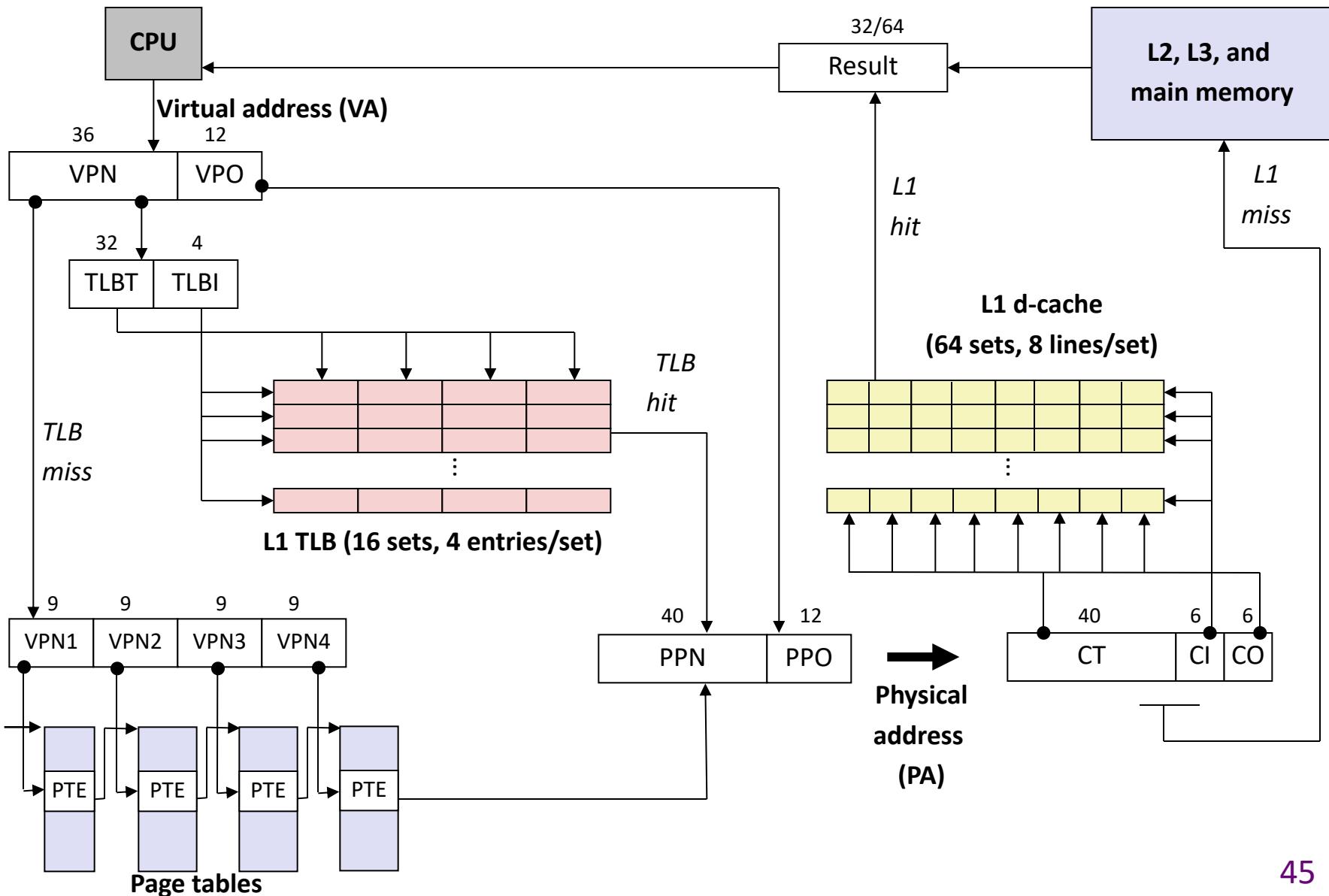
# X86-64 Linux layout

- Only 48 of 64bit used for virtual memory



So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

# End-to-end Core i7 Address Translation



# Core i7 Level 1-3 Page Table Entries

|                                                |        |                                  |    |    |        |   |    |   |   |    |    |     |     |     |     |
|------------------------------------------------|--------|----------------------------------|----|----|--------|---|----|---|---|----|----|-----|-----|-----|-----|
| 63                                             | 62     | 52                               | 51 | 12 | 11     | 9 | 8  | 7 | 6 | 5  | 4  | 3   | 2   | 1   | 0   |
| XD                                             | Unused | Page table physical base address |    |    | Unused | G | PS |   | A | CD | WT | U/S | R/W | P=1 |     |
| Available for OS (page table location on disk) |        |                                  |    |    |        |   |    |   |   |    |    |     |     |     | P=0 |

**Each entry references a 4K child page table**

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

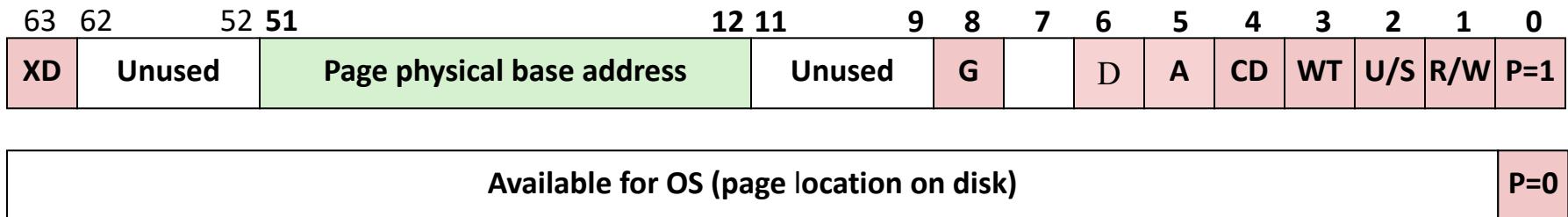
A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

**Page table physical base address:** 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

# Core i7 Level 4 Page Table Entries



**Each entry references a 4K child page**

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

CD: Cache disabled (1) or enabled (0)

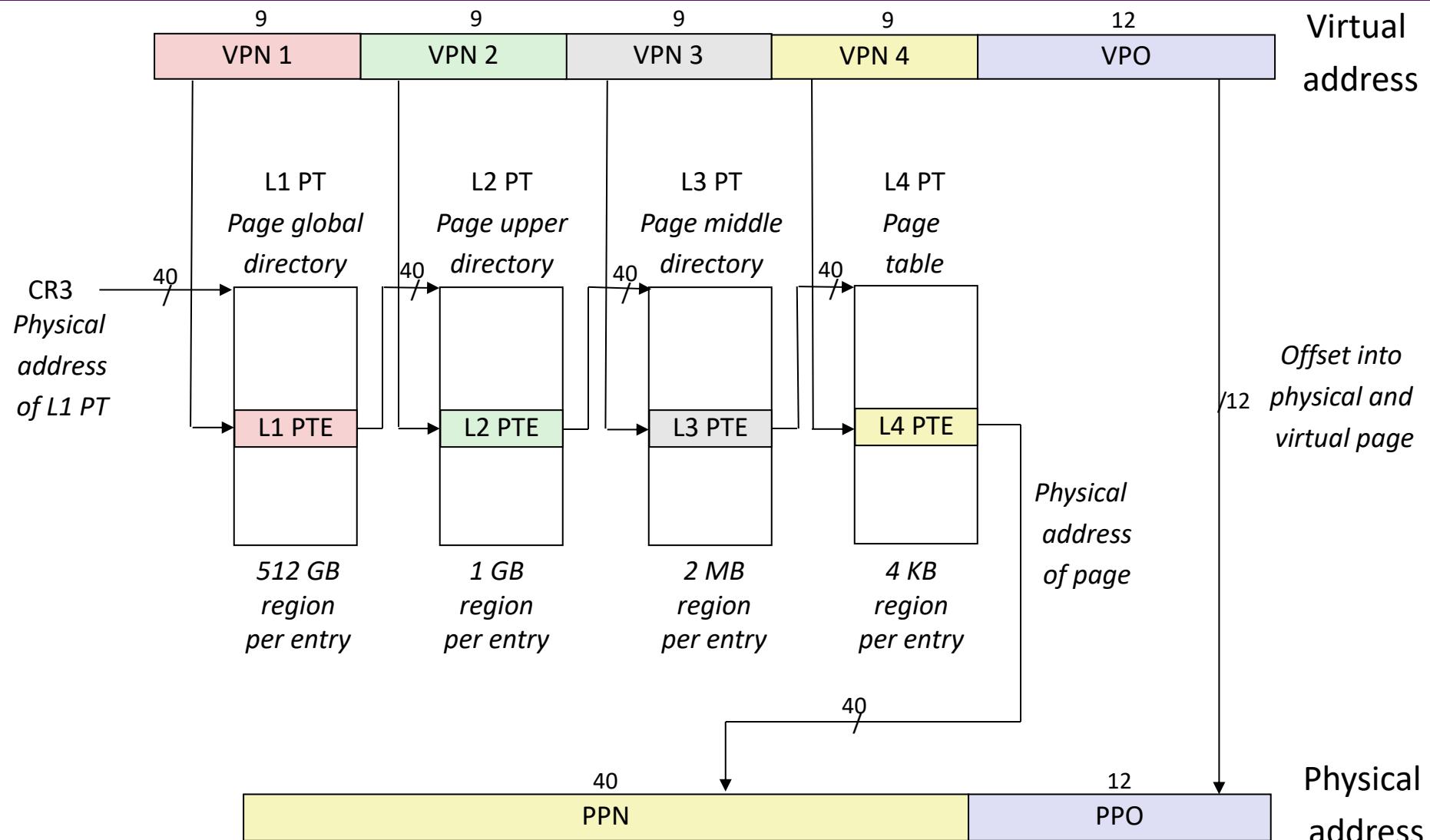
A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

G: Global page (don't evict from TLB on task switch)

**Page physical base address:** 40 most significant bits of physical page address  
(forces pages to be 4KB aligned)

# Core i7 Page Table Translation



# 虚拟索引虚拟标记VIVT(Virtually Indexed Virtually Tagged)

- 虚拟地址作为查找对象，不需要每次读取或者写入操作的时候把虚拟地址经过MMU转换为物理地址，提高性能
- 如果cache miss，则把虚拟地址发往MMU，经过MMU转换成物理地址
- 歧义(ambiguity): 歧义是指不同的数据在cache中具有相同的tag和index
  - 相同的虚拟地址映射不同的物理地址就会出现歧义（例如多进程）
  - 解决：进程切换清除缓存
- 别名(alias): 不同的虚拟地址映射相同的物理地址，而这些虚拟地址的index不同
  - 解决：nocache，仅映射到相同的物理地址

# 物理索引物理标记PIPT(Physically Indexed Physically Tagged)

- tag和index都取自物理地址， 不会出现歧义和别名
- 硬件设计上比VIVT复杂很多， 硬件成本高
- 每次都要翻译成物理地址， 性能下降
  - TLB可以缓解这个问题

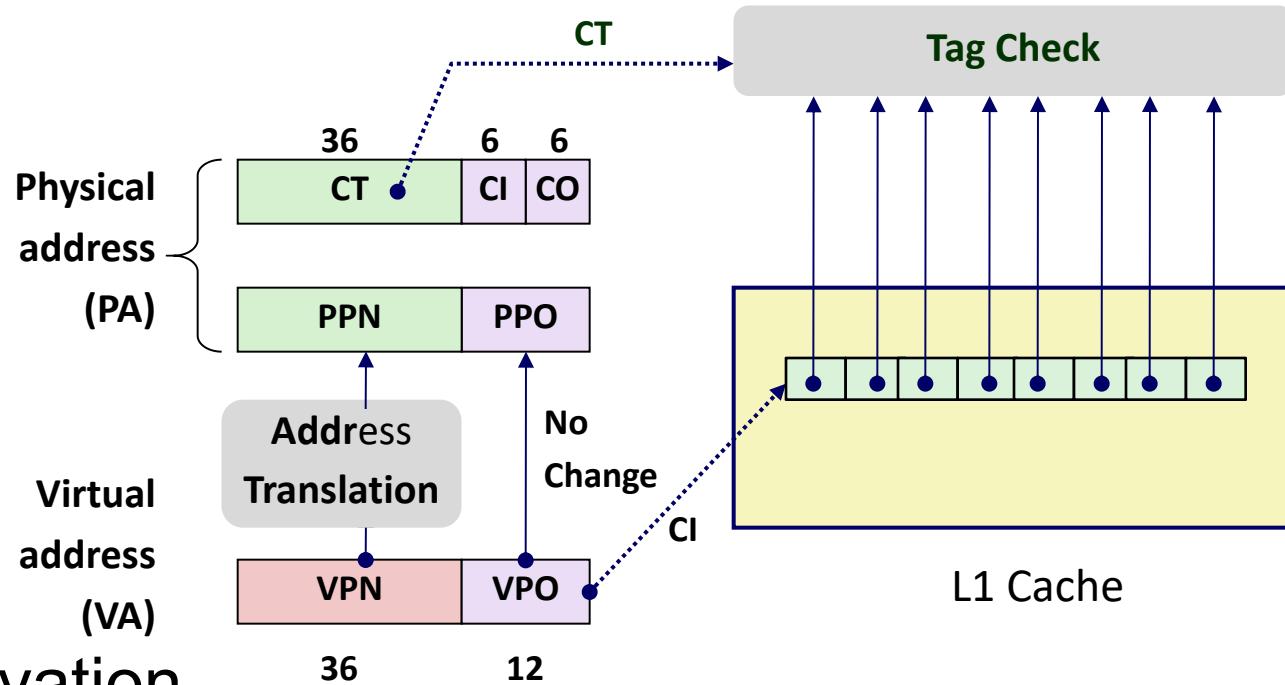
# 虚拟索引物理标记VIPT(Virtually Indexed Physically Tagged)

- 使用虚拟地址对应的index位查找cache，与此同时(硬件上同时进行)将虚拟地址发到MMU转换成物理地址，比较cacheline对应的tag和物理地址tag域判断是否是cache hit还是cache miss
- VIPT以物理地址部分位作为tag，不会存在歧义问题
- 采用虚拟地址作为index，存在别名问题
  - 不同的虚拟地址对应的物理地址是一样的
  - 因为是物理标记，则缓存行中的标记是一样的
  - 但是，虚拟地址不一样，因此index不一样，占用两个或者多个缓存行

# VIPT的别名（一个物理地址映射为多个虚拟地址）

- 映射的最小单位是页，页的大小为4KB，虚拟地址和映射的物理地址[11..0]是一样的
- 若缓存大小小于4KB，那么index部分物理地址和虚拟地址是一样的，等同于PIPT，不存在别名
- 缓存大小为8KB，缓存行大小为256B，直接映射index来自于[12..8]。此时[11..8]部分的物理地址和虚拟地址是一样的。而位12的物理地址和虚拟地址不一样。物理地址0x4000可以被映射到0x0000和0x1000两个虚拟地址，装入到两个cacheline，造成了别名。
- 不产生别名的条件：Index位数+cacheline大小位数<= Page大小位数

# Cute Trick for Speeding Up L1 Access



## Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

---

谢谢



# 外存储器

2022年秋

# 内容提要

---

- 硬盘存储
- RAID技术
- SSD存储

# 存储设备概述

- 锁存器和触发器 (Latch and FlipFlop)
  - 非常快, 可以并行访问
  - 非常昂贵 (1bit 需要几十个晶体管)
- 静态内存Static RAM (SRAM)
  - 相对来说很快Relatively fast
  - 昂贵 (1 bit 需要 6+ 左右的晶体管)
- 动态内存 (DRAM)
  - 相对慢, 读破坏数据 (需要刷新), 需要特殊的制造工艺 (挖一个电容出来)
  - 相对便宜 (1bit 需要 1个晶体管加1个电容)
- 非易失性存储 (闪存, 硬盘, 磁带)
  - 非常慢, 延迟长, 非易失
  - 非常便宜

# 非易失性存储

## 口易失性存储:

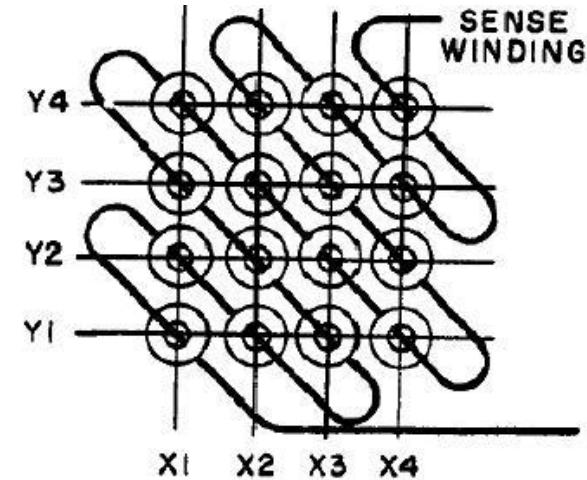
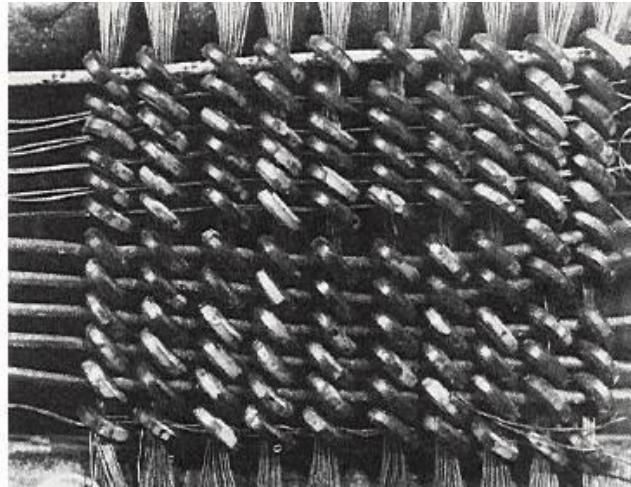
- 静态存储器: SRAM, Cache
- 动态存储器: DRAM
- 特点: 快速, 掉电后信息丢失, 访问粒度小 (字节, 缓存块)

## 口非易失性存储器:

- 磁盘, 磁带: 磁表面存储器
- 光盘
- SSD, 固态存储器
- 特点: 慢速, 掉电后信息不丢失, 访问粒度大 (以数据块为访问单位)

# 磁芯存储器

- 圆柱型陶瓷上涂磁粉
- 手工穿线，水手结
- 消磁后重写
- 存储原理简单
- 工艺复杂
- 可靠性低
- 大存储容量
- 成本低廉
- 断电后保存数据



# 磁表面存储设备

- 磁颗粒的不同偏转方向来区分不同的状态
- 主存中存放CPU要立即访问的程序和数据
- 辅助存储器中存放CPU不立即使用的信息，在需要时再调入主存中
  - 一般为磁盘、光盘等
  - 容量大、成本低、断电后还可以保存信息，能脱机保存信息，弥补了主存的不足
  - 串行访问、数据交换频率低、数据交换量大

# 随机访问和串行访问

---

## □ 随机访问

- 随机访问任何单元，访问时间与信息存放位置无关
- 每一位都有各自的读写设备

## □ 串行访问

- 顺序地一位一位地进行，访问时间与存储位的物理位置有关
- 共用一个读写设备
- 顺序访问和直接访问

# 主要技术指标

---

## □ 存储密度

- 单位长度（磁带）或单位面积（磁盘）磁层表面所存储的二进制信息量

## □ 存储容量

- 磁表面存储器所能存储的二进制信息的总量，以字节为单位

## □ 寻址时间

## □ 数据传输率

## □ 误码率

## □ 价格

# 磁表面存储设备

---

□ 如何保存?

- 磁颗粒的不同磁化偏转方向

□ 如何表示?

- 磁记录方式

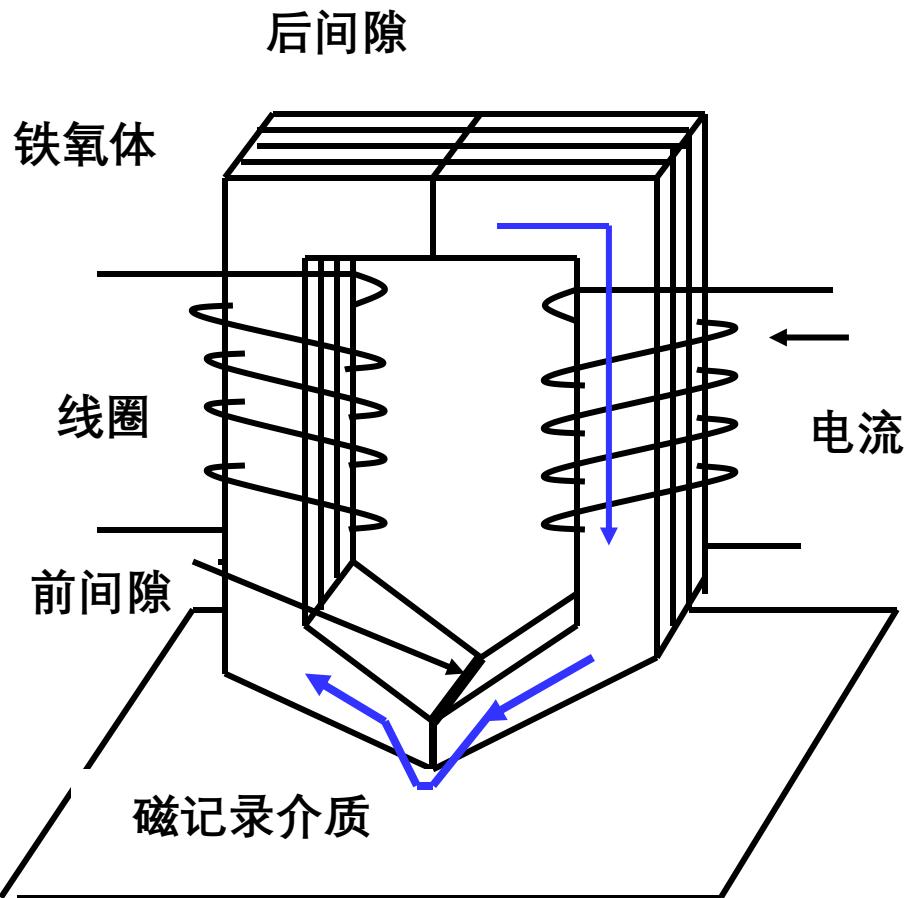
□ 如何组织?

- 扇区、磁道、柱面、硬盘

□ 如何管理?

- 操作系统的文件系统

# 磁记录原理



磁头，软磁材料  
导磁率高，饱和磁感应强度大  
矫顽力小，剩余磁感应强度小

磁记录材料，硬磁材料  
记录密度高，记录信息时间长  
输出信号幅度大，噪声低  
表面组织紧密、光滑、无麻点  
薄厚均匀，温度、湿度影响小

磁头结构和电磁转换示意图

# 磁记录方式

## 口磁记录方式

- 指一种编码方法，即如何将一串二进制信息，通过读写电路变换成磁层介质中的磁化翻转序列。

## 口评价标准

- 编码效率
  - 表示一个二进制位数据需要使用多少个磁颗粒？
- 自同步能力
  - 读写时准确定位二进制数据位的能力
- 读写可靠性

# 磁记录方式

## □ 归零制 (RZ)

- 线圈中正脉冲为“1”，负脉冲表示“0”，两位信息位之间线圈中电流为零。

## □ 不归零制 (NRZ)

- 线圈中一直有正或负脉冲（包括两位信息位之间）。

## □ 见1翻转的不归零制 (NRZ1)

- 只有见到“1”才改变电流的方向

# 磁记录方式

## □ 调相制 (PM)

- 用脉冲的边沿来表示 “0” 和 “1”

## □ 调频制 (FM)

- “1”: 位周期中心和位与位之间都翻转
- “0”: 位周期中心不翻转，位与位之间翻转

## □ 改进的调频制 (MFM)

- 只有连续两个或以上的 “0” 时，才在位周期的起始位置翻转

# 常用磁记录方式波形图

位周期

位信息

1 0 1 1 1 0 0 0

1

NRZ

NRZI

PM

FM

MFM

RZ

# 磁盘

## □ 目的

- 长期存储、断电后存储
- 容量大、价格低廉，但速度慢
- 可用在层次存储器的最底层

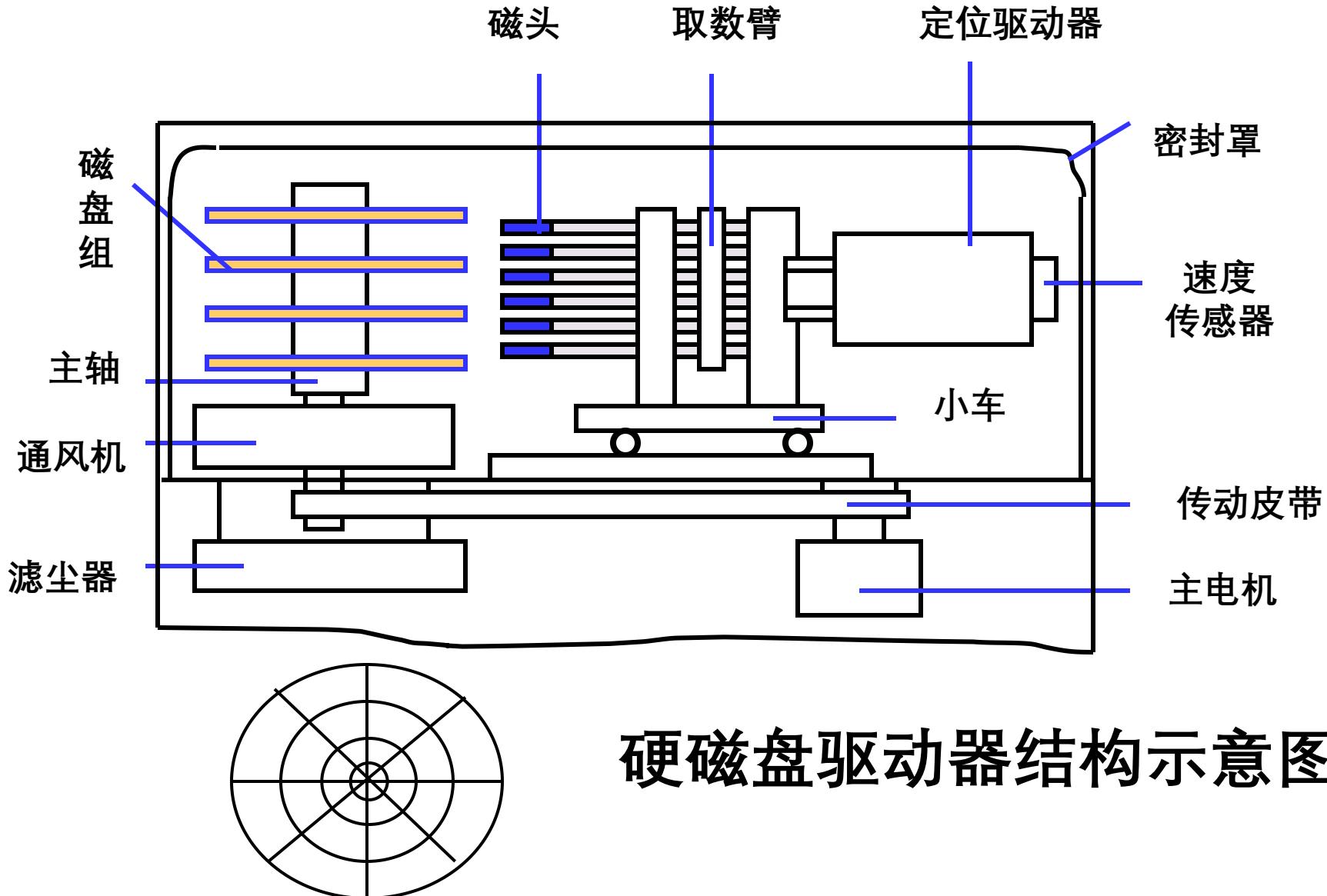
## □ 特点

- 使用旋转托盘上的表面磁颗粒来存储数据
- 可移动的读/写头来访问磁盘

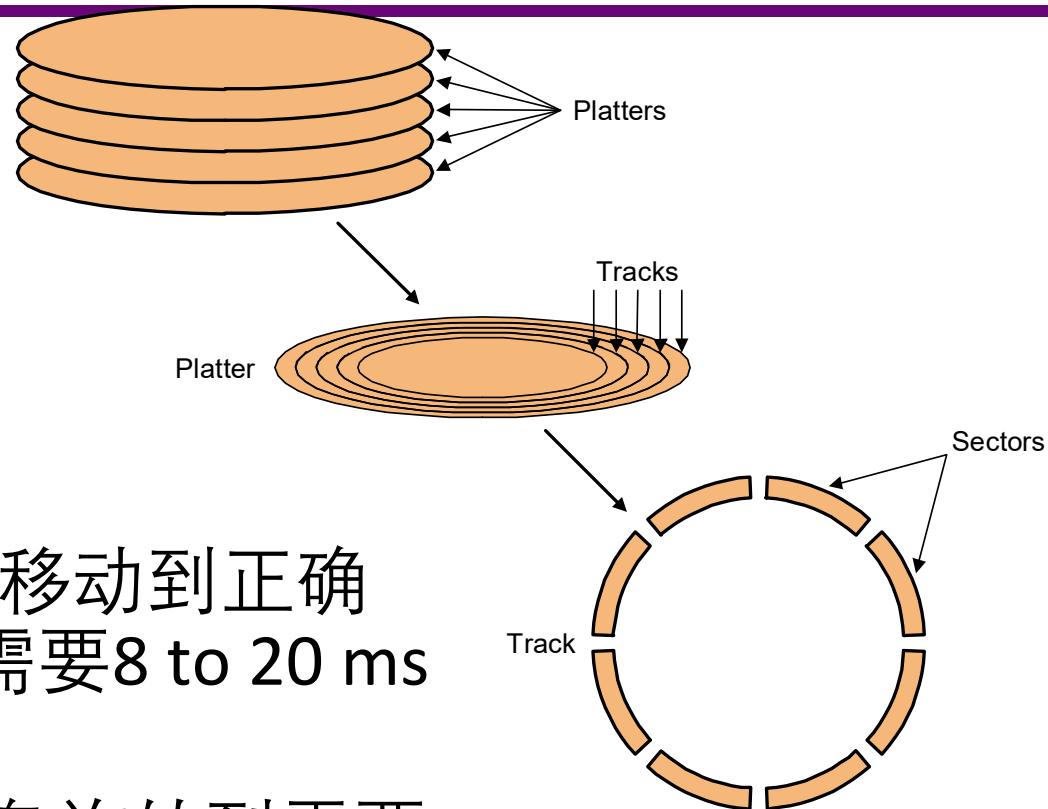
## □ 硬盘、软盘比较

- 硬质托盘（金属铝），面积可以比较大；
- 由于可被精确控制，密度可以更高
- 旋转速度快，传输率高
- 可以多个盘片组合

# 硬磁盘设备



# 硬磁盘内部结构



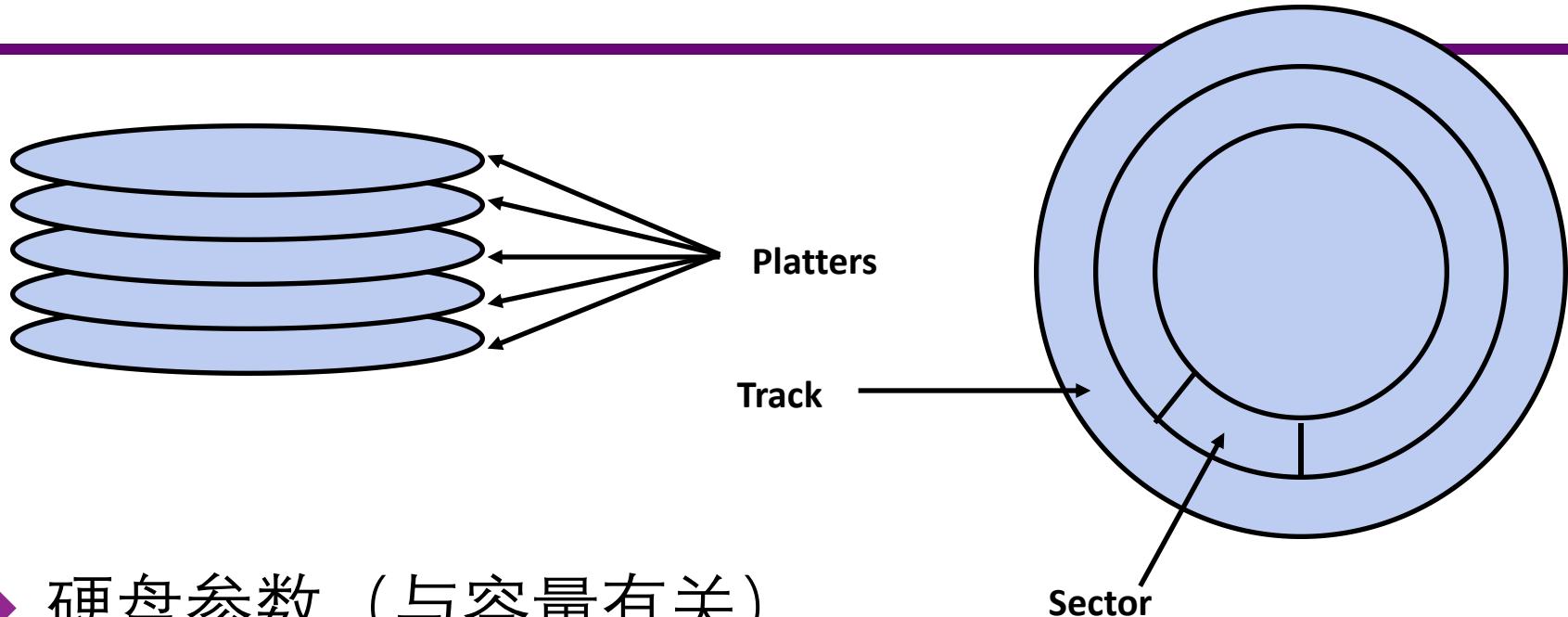
磁盘访问过程：

寻道：将读写磁头 移动到正确的磁道上（平均需要8 to 20 ms）

寻找扇区：等待磁盘旋转到需要访问的扇区 (.5 / RPM)

数据传输：读写数据（1个或多个扇区）（2 to 15 MB/sec）

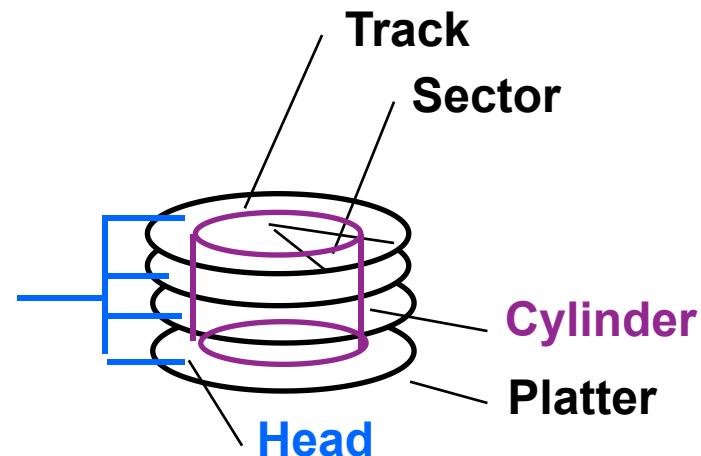
# 硬磁盘内部结构



- ▶ 硬盘参数（与容量有关）
  - ▶ 500 至 2,000 磁道（每面）
  - ▶ 32 至 128 个扇区（每个磁道）
    - ▶ 扇区是磁盘访问的最小单位
- ▶ 早期硬盘上每个磁道上的扇区数是一样的
- ▶ 增加容量
  - ▶ 位密度不变：外磁道比内磁道扇区数多一些

# 硬磁盘参数

- ▶ 柱面：位于同一半径的磁道集合
- ▶ 读/写数据的三个步骤：
  - ▶ 寻道时间：将磁头移动到正确的磁道上
  - ▶ 旋转延迟：等待磁盘上扇区旋转到磁头下
  - ▶ 传输时间：真正的数据读/写时间
- ▶ 当前平均寻道时间：
  - ▶ 一般为 8 至 12 ms



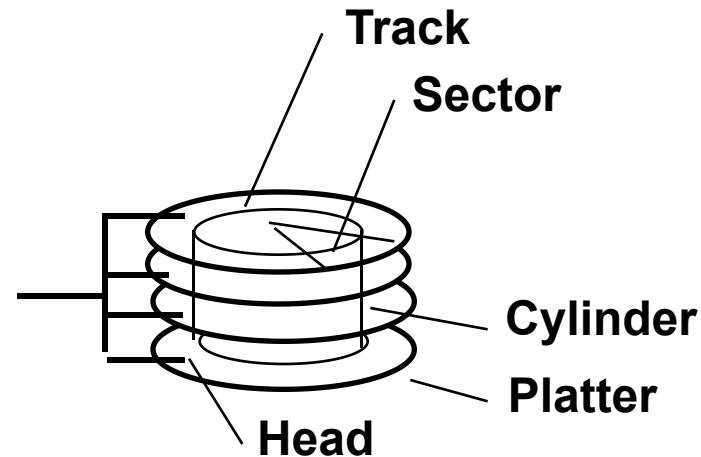
# 硬磁盘参数

## ▶ 旋转延迟：

- ▶ 旋转速度：3,600至7200 RPM
- ▶ 旋转时间：16 ms至8 ms每转
- ▶ 平均寻址时间8ms至4ms

## ▶ 访问速度：

- ▶ 数据量（通常为1个扇区）：1 KB / sector
- ▶ 旋转速度：3600 RPM至7200 RPM
- ▶ 存储密度：磁道上单位长度存储的位数
- ▶ 磁盘直径：2.5至 5.25 in
- ▶ 一般为：2 至12 MB每秒



# 硬磁盘访问时间

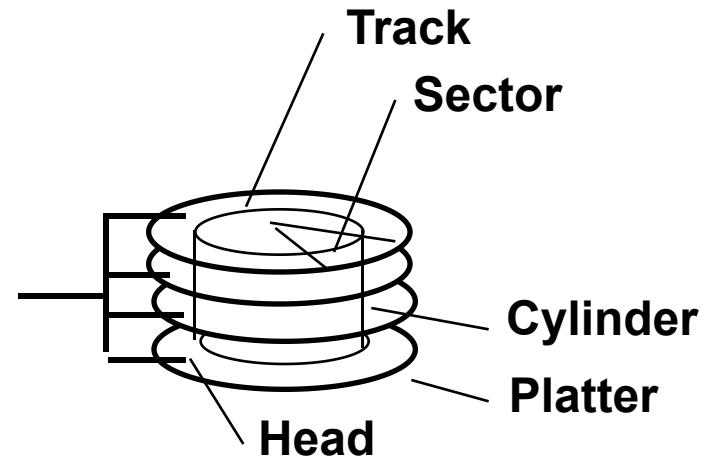
- ▶ 磁盘访问时间 = 寻道时间 + 旋转延迟 + 传输时间 + 磁盘控制器延迟
- ▶ 举例：
  - ▶ 平均寻道时间 = 12ms;
  - ▶ 旋转速度 = 5400rpm
  - ▶ 磁盘控制器延迟： 2ms
  - ▶ 传输速度 = 5MB
  - ▶ 扇区大小 = 512 bytes
  - ▶ 读取一页（8KB）需要多少时间？
- ▶ 旋转延迟： 平均旋转延迟应为磁盘旋转半周的时间。
  - ▶ 旋转1周 =  $1/5400 \text{ minutes}$   
 $= 11.1\text{ms} \Rightarrow \frac{1}{2} \text{ 周: } 5.6 \text{ ms}$
  - ▶ 读1个扇区时间 =  $12\text{ms} + 5.6\text{ms} + .5\text{K}/5\text{MB} + 2\text{ms}$   
 $= 12 + 5.6 + .1\text{ms} + 2\text{ms}$   
 $= 19.7 \text{ ms}$
  - ▶ 读1页的时间 =  
 $= 12 \text{ ms} + 5.6\text{ms} + 8\text{K}/5\text{MBpersec} + 2\text{ms}$   
 $= 12\text{ms} + 5.6\text{ms} + 1.6\text{ms} + 2\text{ms}$   
 $= 21.2 \text{ ms}$

# 对磁盘访问的思考

页容量大，为什么扇区却如此小呢？

- ▶ 理由 #1：可用性。可以在扇区物理损坏时不再使用该扇区。
- ▶ 理由 #2：还是可用性。检错纠错码分布在每个扇区，扇区容量小，检错速度快，效率高。
- ▶ 理由 #3：灵活性。使用不同的操作系统，不同的页面大小。

- ▶ 采用并行方式和大容量传输方式克服磁盘控制器延迟
- ▶ 大容量传输：每次读取多个扇区，可以节约时间。
- ▶ 也可以分担部分总线延迟...
- ▶ 并行 #1：并行读多个层面
- ▶ 并行 #2：并行读多个磁盘



# 结论

---

□ 应该记住以下两点：

- 额外开销在总开销中比例较大 =>一次传输大量数据比较有效
- 将页面存放在相邻扇区中可以避免额外的寻道开销

# 访问磁盘过程

- ▶ 对磁盘的访问总是由缺页引起的：
  - ▶ CPU给出地址，需要访问某存储单元；
  - ▶ 并行进行TLB查找和cache查找；
  - ▶ TLB查找后申明没有找到；
  - ▶ 停止并行查找，并通知操作系统处理；
  - ▶ 操作系统检查页表，发现该页不在内存中，需要从硬盘调入。应该如何进行呢？
  - ▶ 操作系统从主存中选择一页准备换出，为调入的页安排存放空间；
  - ▶ 若被换出的页是“脏”页，需要将其写回磁盘存储；
  - ▶ 操作系统申请I/O总线；
  - ▶ 获得批准后，发送写命令给I/O设备（磁盘）。紧跟着传送需要写回的页的全部数据。
  - ▶ I/O控制器发现发给自己的写命令，加入到握手协议，并接受数据。
  - ▶ 根据数据要写入的地址，读/写头移动到正确的柱面，同时，将数据接收到缓冲区。
  - ▶ 寻道结束后，等待相应的扇区旋转到磁头下面，将数据写入扇区中。
  - ▶ 在写入数据间隙，计算校验码并写入扇区中。

# 磁盘访问过程

- 下一步，操作系统继续申请总线（如果还保持总线控制权，则不必申请）。
- 得到授权后，向磁盘发出读命令。
- 然后，磁盘识别地址，并转换为相应的地址段。
- 寻道，将读/写头移动到指定位置。
- 从指定扇区中读去数据，并进行校验。
- 磁盘申请I/O总线。
- 得到授权后，将数据通过总线送到内存。

# 可靠性和可用性

## □ 两个经常混淆的词汇：

- 可靠性：设备出现故障的几率来衡量。
- 可用性：系统能正常运行的几率来衡量。

## □ 可用性可以增加硬件冗余来提高：

- 例如：在存储器中增加校验码。

## □ 可靠性只能通过下面途径提高：

- 改善使用环境
- 提高各部件的可靠性
- 减少组成部件
  - 可用性的提高可能带来可靠性的降低

# RAID的提出

---

□CPU性能在过去的十年中有了极大地提高，几乎是每18个月翻一番。但磁盘的性能却没能跟上。在70年代，小型机磁盘的平均查找时间为50到100毫秒，现在是10毫秒。在许多行业（如汽车或航空业），如果性能的提高能达到这个速度，即20年内提高5到10倍，那就会是头条新闻，但对计算机行业，这却成了一个障碍。因为CPU性能和磁盘性能间的差距这些年来越来越大。

# RAID的提出

在提高CPU性能方面，并行处理技术已得到广泛使用。这些年来，许多人意识到，并行I/O也是一个提高磁盘性能的好办法。1988年，Patterson et al.在他的一篇文章中建议用6个特定的磁盘组织来提高磁盘的性能或可用性，或两方面都同时提高。这个建议很快就被采用，并导致了一种新的I/O设备的诞生，这就是**RAID盘**。Patterson et al.把RAID定义为**廉价磁盘的冗余阵列**（Redundant Array of Inexpensive Disks），但工业界把“I”由“廉价的（Inexpensive）”替换成“独立的（Independent）”。

# RAID

## □ RAID定义

- 廉价磁盘的冗余阵列（Redundant Arrays of Inexpensive Disks）
- 用N个低价磁盘构成一个统一管理的阵列，以取代特贵单一磁盘

## □ RAID目标

- N个磁盘的容量
- 1/N的访问时间
- 更高的性价比
- 采用冗余技术提高存储信息的可用性

**RAID0: Data Striping**

**RAID1: Drive Mirroring**

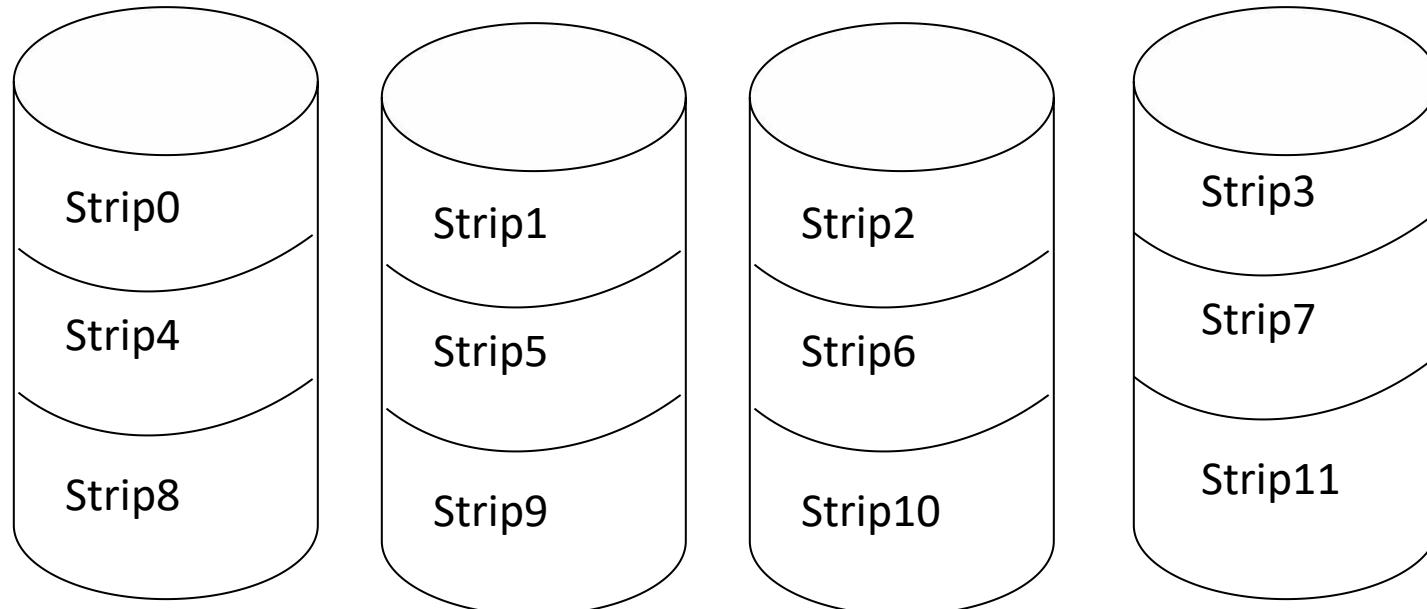
**RAID4: Data Guarding**

**RAID5: Distributed Data Guarding**

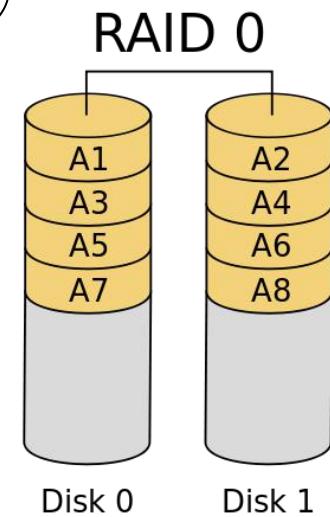
# RAID0

- RAID0将由RAID模拟的单个虚拟磁盘划分成带（strip），每带 $k$ 个扇区。第0带为第0到第 $k - 1$ 扇区，第1带为第 $k$ 扇区到第 $2k - 1$ 扇区，等等。对 $k=1$ ，每个带为1个扇区；对 $k=2$ ，每带有2个扇区；等等。RAID 0以交叉循环的方式将数据写到连续的带中，下图描述的就是有4个磁盘驱动器的RAID盘。这种在多个驱动器上分布数据的方式叫作分带。如果软件发出从带的边界开始读四个连续带的数据块的命令，RAID控制器将把这个命令分解成四个单独的读命令，四个驱动器每个一个，让它们并行执行。这样，就实现了对软件透明的并行I/O操作。

# RAID0



1. 适合数据请求量比较大的情况
2. 没有冗余，可靠性差，不算真正的RAID

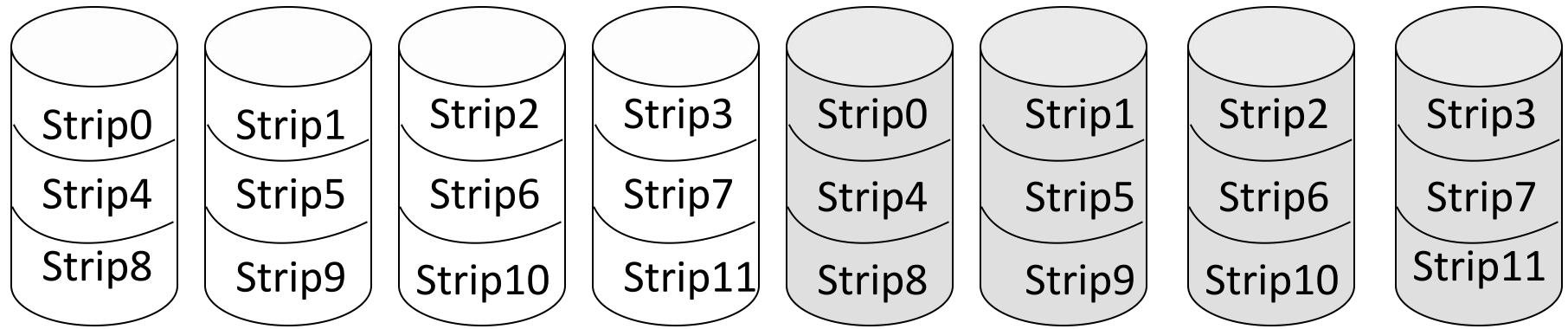


# RAID1

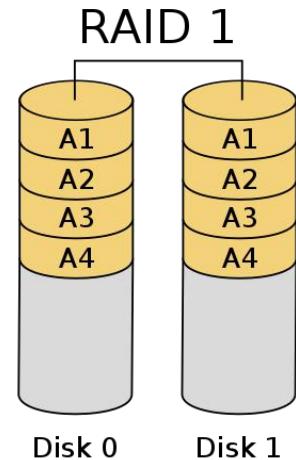
---

□ 它复制了所有的磁盘，所以有四块主磁盘和四块辅助磁盘。每个对磁盘的写操作都进行两次，而每次读操作则可以读任意一个备份，把负载均衡分布到不同的驱动器上。这样，写操作的性能并不比单个磁盘好，但读磁盘的性能却比单个磁盘高了两倍。容错性能就更好了，如果一个驱动器崩溃的话，只要简单的用备份驱动器代替就行了。恢复整个磁盘的操作包括两个步骤：装上一个新的驱动器，然后将整个备份驱动器的内容拷贝到新的驱动器上。

# RAID1



1. 元余备份，可靠性高
2. 写性能不高，但读性能却提高了两倍
3. 成本较高

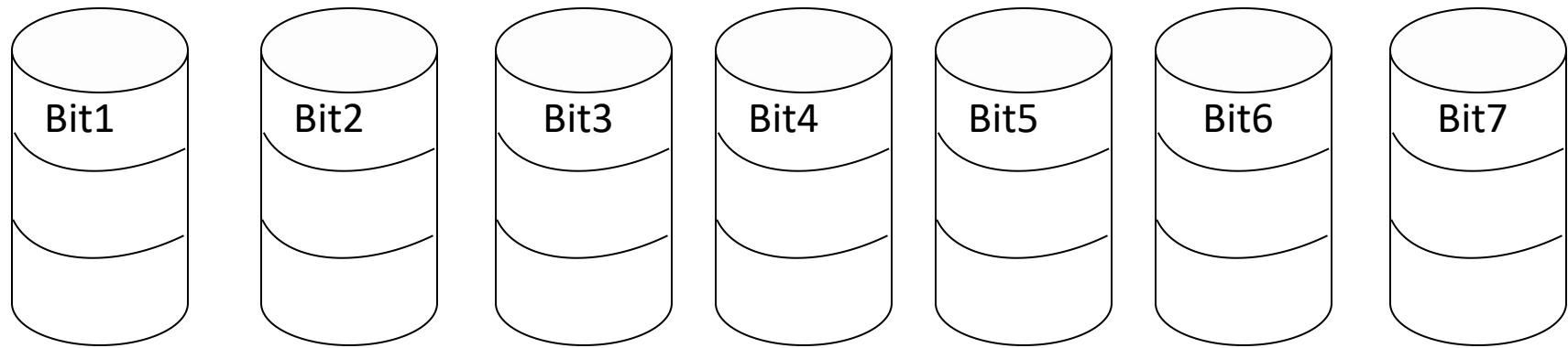


# RAID2

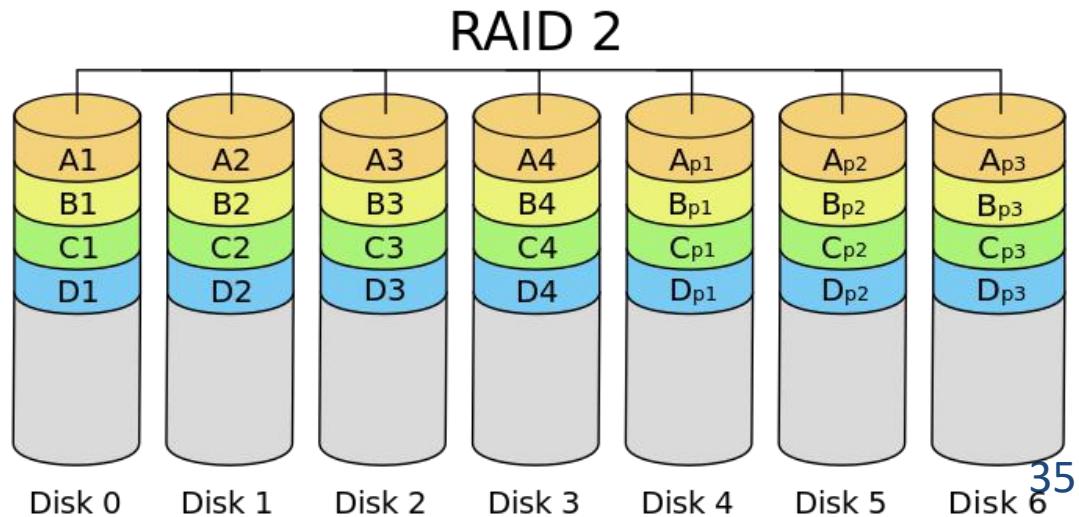
---

□ RAID 2的工作单位为字，可能的话甚至可以是字节。首先我们可以想象将单个虚拟磁盘上的字节分解成一对4位的半字节，对每个半字节加上3位海明码形成7位字，即其中1、2、4位做校验位。然后，用下图所示的七个驱动器的磁头和旋转同步，就可能将整个海明码字写在七个驱动器上，每个驱动器一位。

# RAID2



1. 驱动器必须同步旋转
2. 驱动器个数要足够多
3. 需要多个控制器



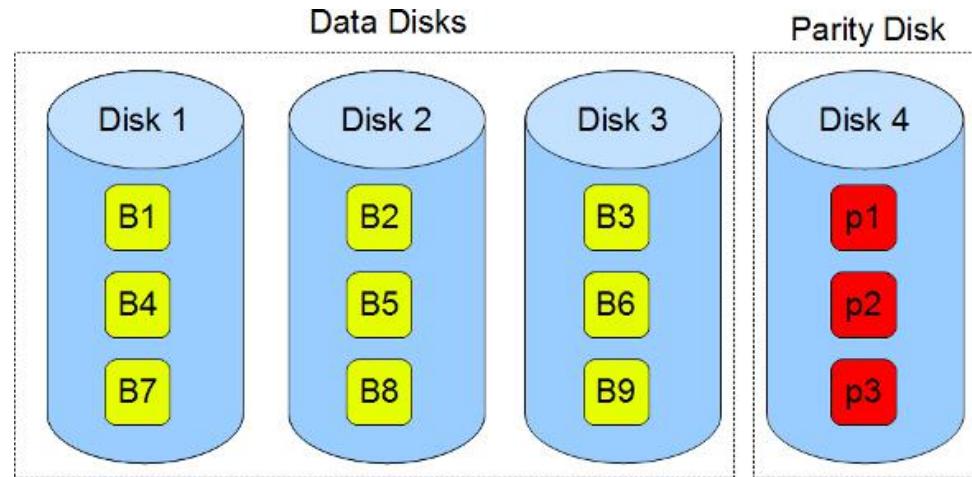
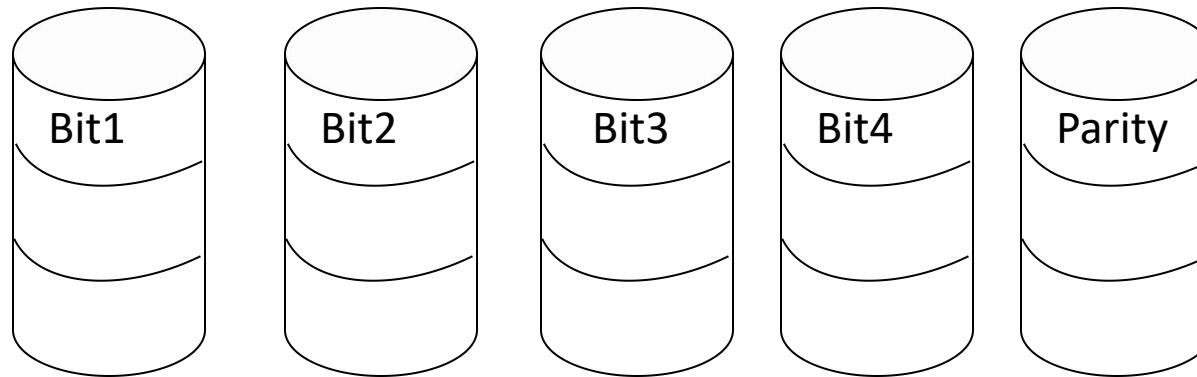
# RAID3

---

- RAID 3是RAID 2的一个简化版本，它只需对每个字计算一个校验位，写到一个校验驱动器上。和RAID 2相同，驱动器之间必须严格同步，因为一个字被分布到多个驱动器中。

# RAID3

1. 驱动器之间要严格同步
2. 对整个磁盘崩溃的错误，能够进行恢复



RAID 3 – Bytes Striped. ( and Dedicated Parity Disk)

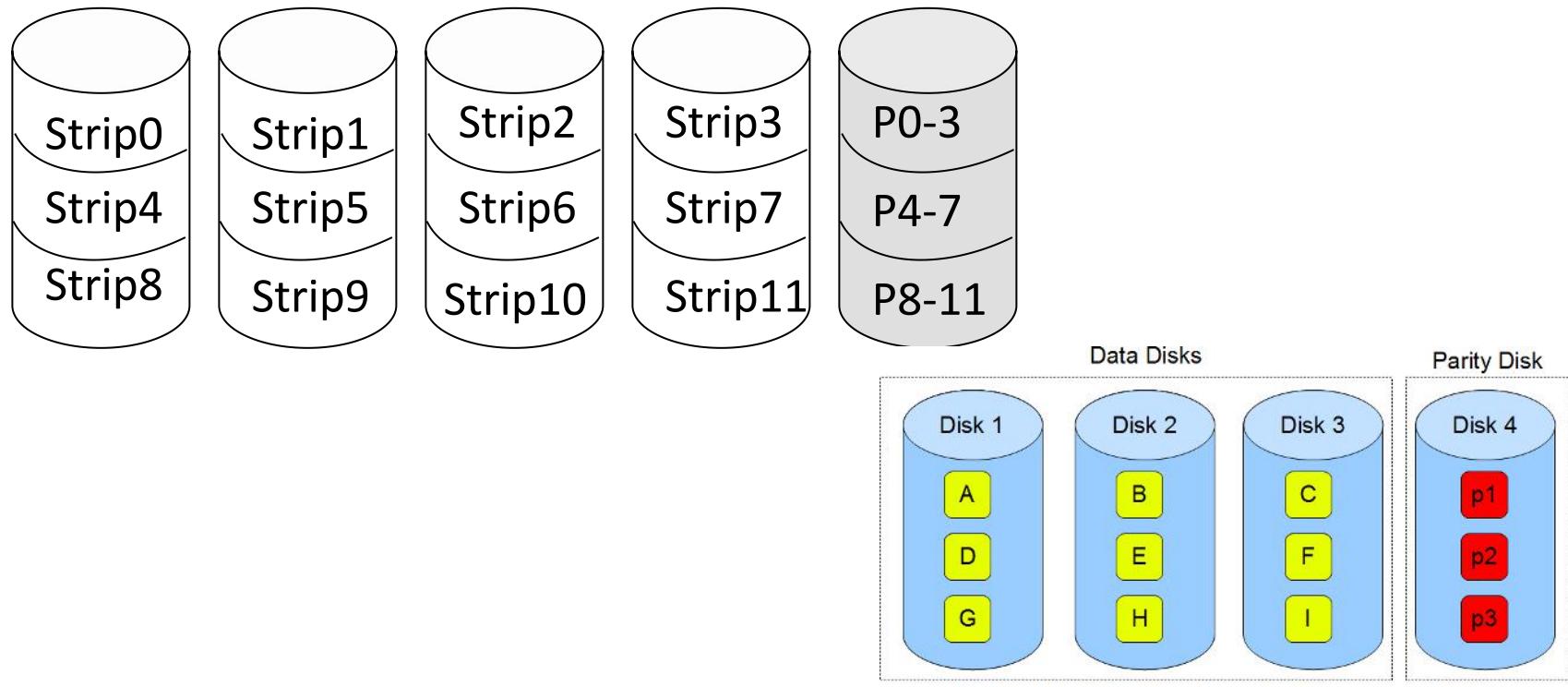
# RAID4

---

□ RAID 4和RAID 0类似，将对带的校验写在额外的驱动器上。例如，若带的长度是 $k$ 个字节，将所有的带异或到一起，产生一个 $k$ 字节长的校验带。如果其中一块磁盘崩溃的话，它的内容可以从校验磁盘上重新计算出来。

# RAID4

1. 不对字进行校验，也不需要驱动器同步
2. 可以防止整块盘崩溃，但对盘上部分字节数据出错的纠错性能相当差
3. 校验盘负载沉重

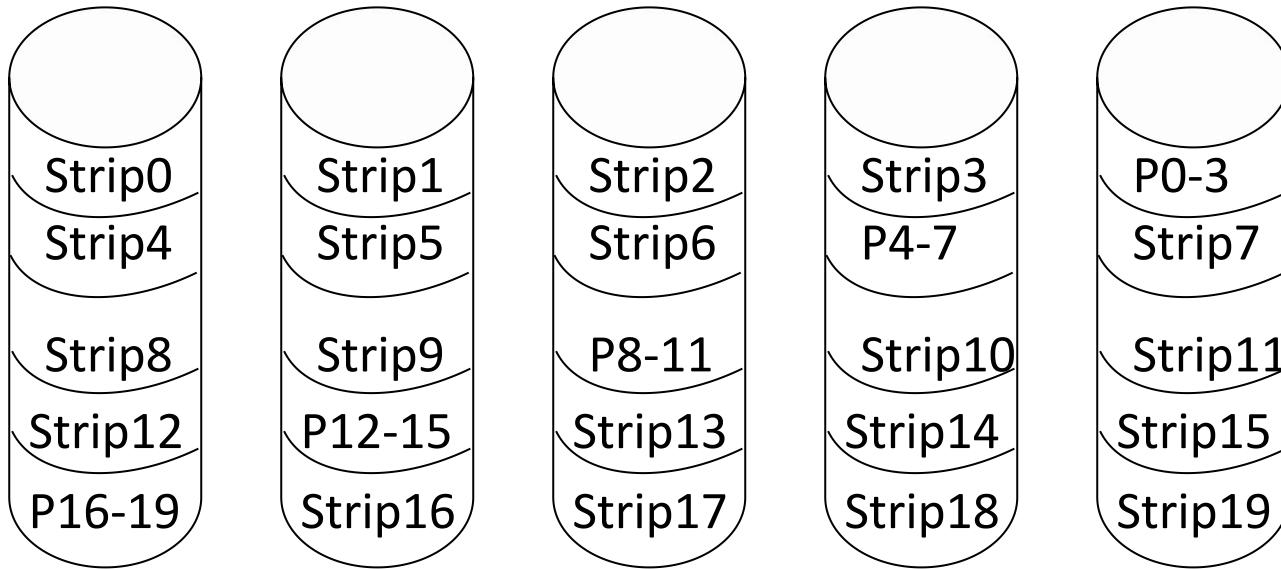


# RAID5

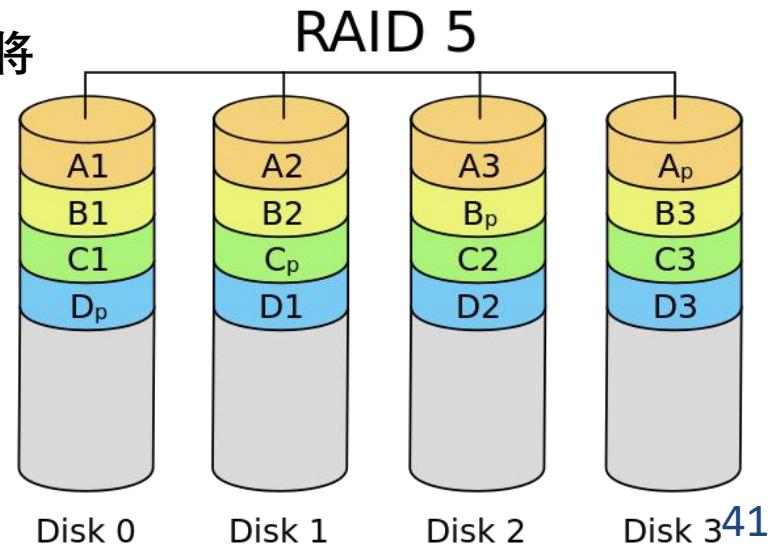
---

- RAID 5为减少校验盘的负载， 将校验位循环均匀分布到所有的驱动器上。

# RAID5



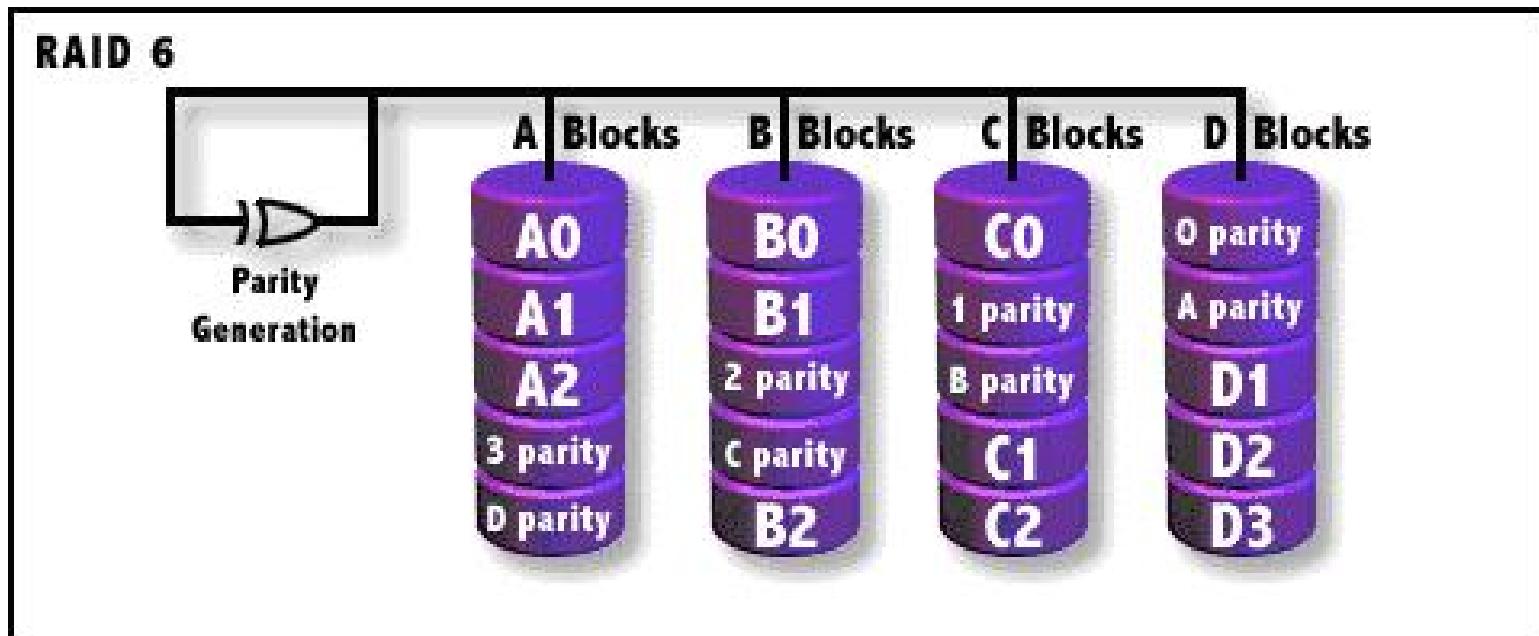
1.如果RAID 5的磁盘崩溃的话，修复磁盘内容的将是一个复杂的过程。



# RAID6

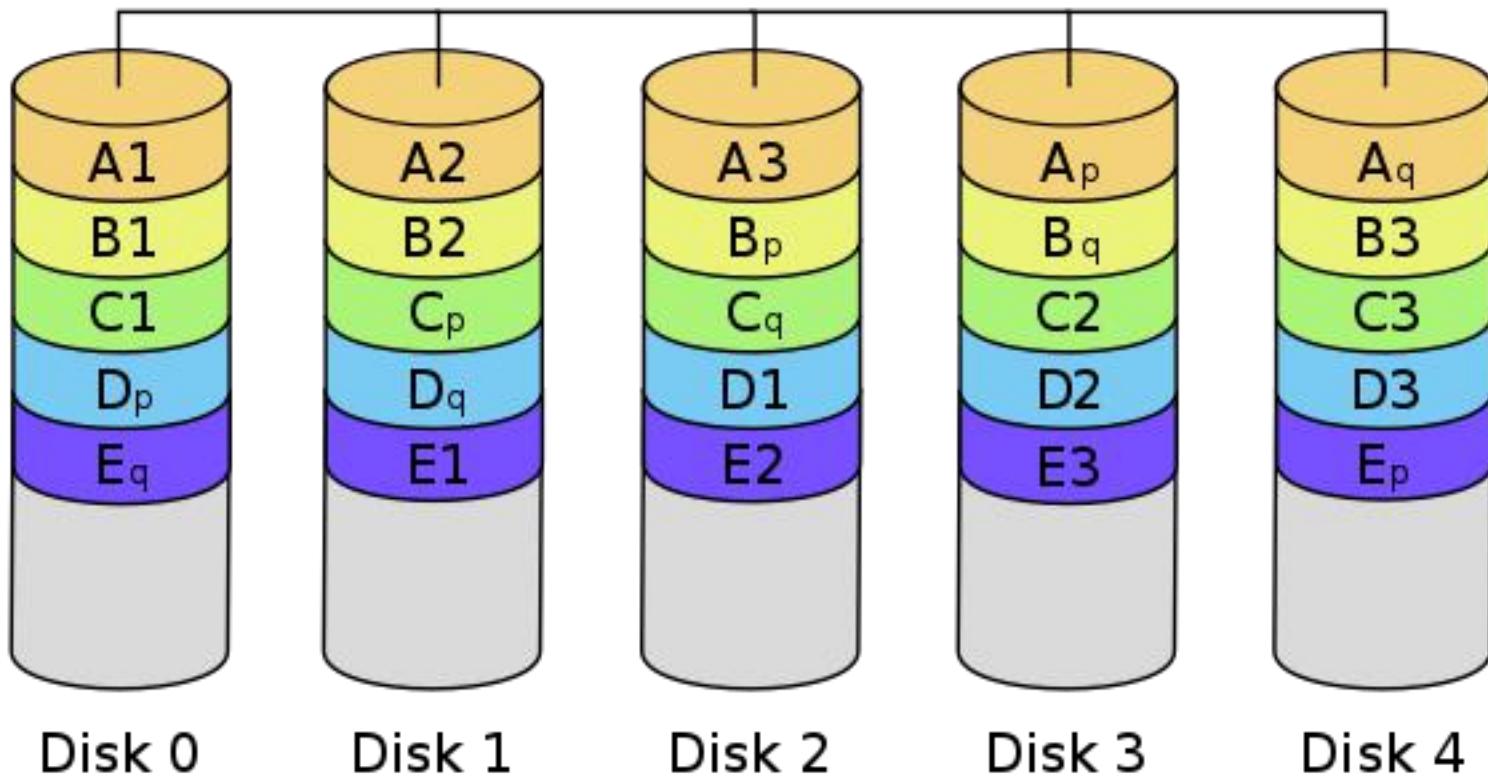
## □ 如果两个磁盘出错呢？

- Independent Data disks with two independent distributed parity schemes
- 二维校验



# RAID6

RAID 6



# 磁存储小结

---

## □ 磁表面存储设备

- 用磁颗粒的不同磁化方向表示0和1
- 弥补了主存的不足
- 磁盘存储原理及磁记录方式

## □ 磁盘的访问过程

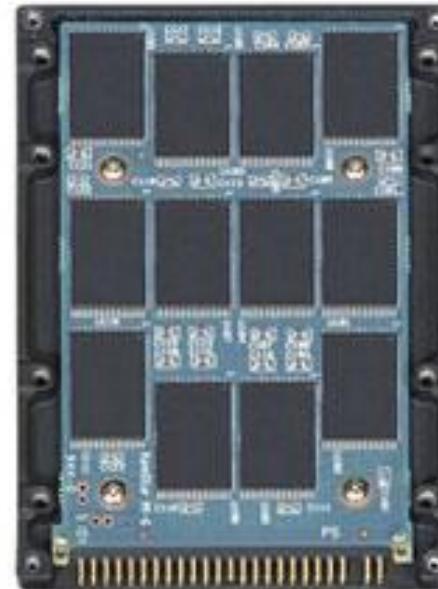
- 寻道、寻找扇区、访问

## □ RAID技术

- 提高磁盘的可用性和性能

# 固态硬盘

- 固态硬盘没有机械结构， 没有移动的部分
- 安静， 低功耗， 高性能， 不怕摔， 低发热
- 价格比硬盘高（将来会下降）， 有限的擦除次数



# 固态硬盘存储

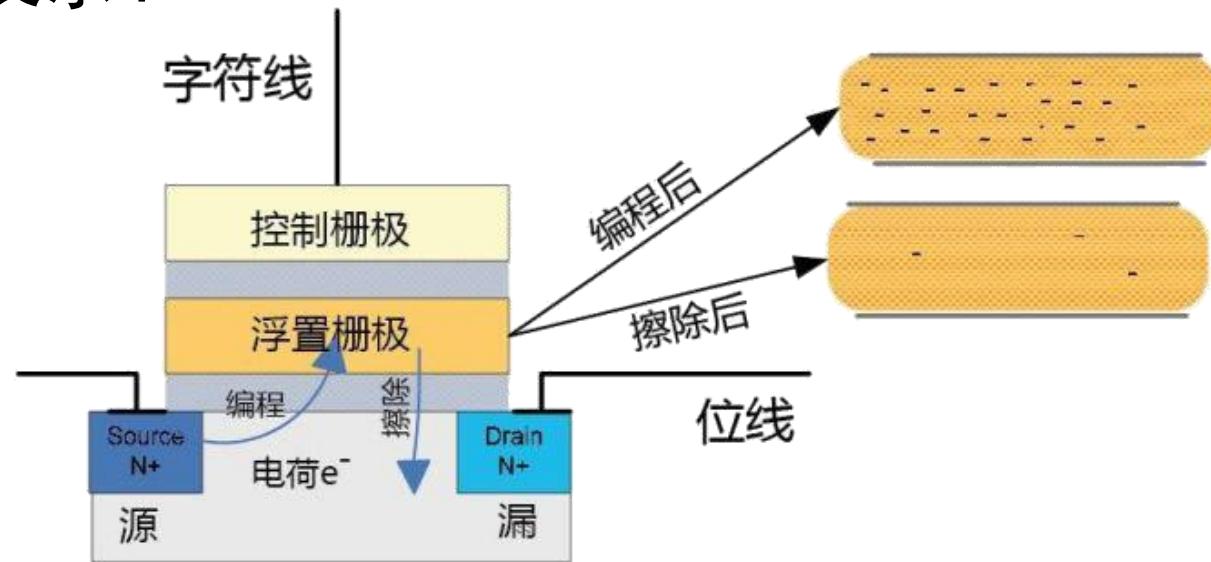
□ 固态硬盘（Solid State Drives），用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元（FLASH芯片、DRAM芯片）组成。

SSD内部主要部件分布图（以三星840EVO为例）

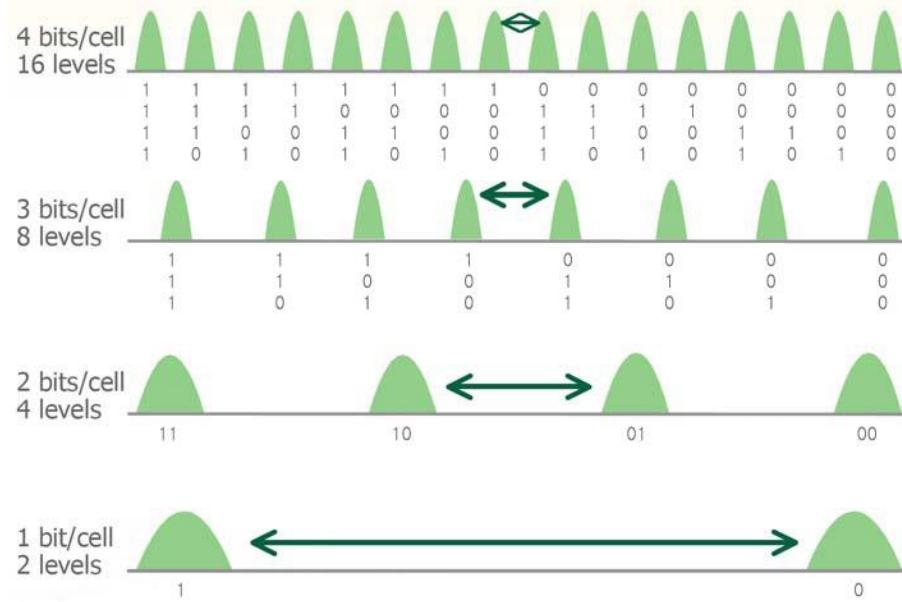


# 固态硬盘的存储单元

- 闪存的内部存储结构是金属-氧化层-半导体-场效应管(MOSFET): 源极, 漏极和栅极, 增加了浮置栅极
- 对于闪存的写入, 即控制栅极去充电, 对栅极加压, 使得浮置栅极存储的电荷越多, 超过阈值, 就表示0
- 对于闪存的擦除, 即对浮置栅极进行放电, 低于阈值, 就表示1



# SLC, MLC, TLC, QLC



- 按照每个存储单元能够存储的位数分为SLC, MLC, TLC, QLC
- 多比特单元采用格雷码编码

# 固态硬盘存储单元的擦除次数

## □ 有限次擦除

- 随着擦除次数的增加，存储单元不能可靠地保持状态（存储数据）
  - 耐久性 Endurance
  - 保持力 Retention

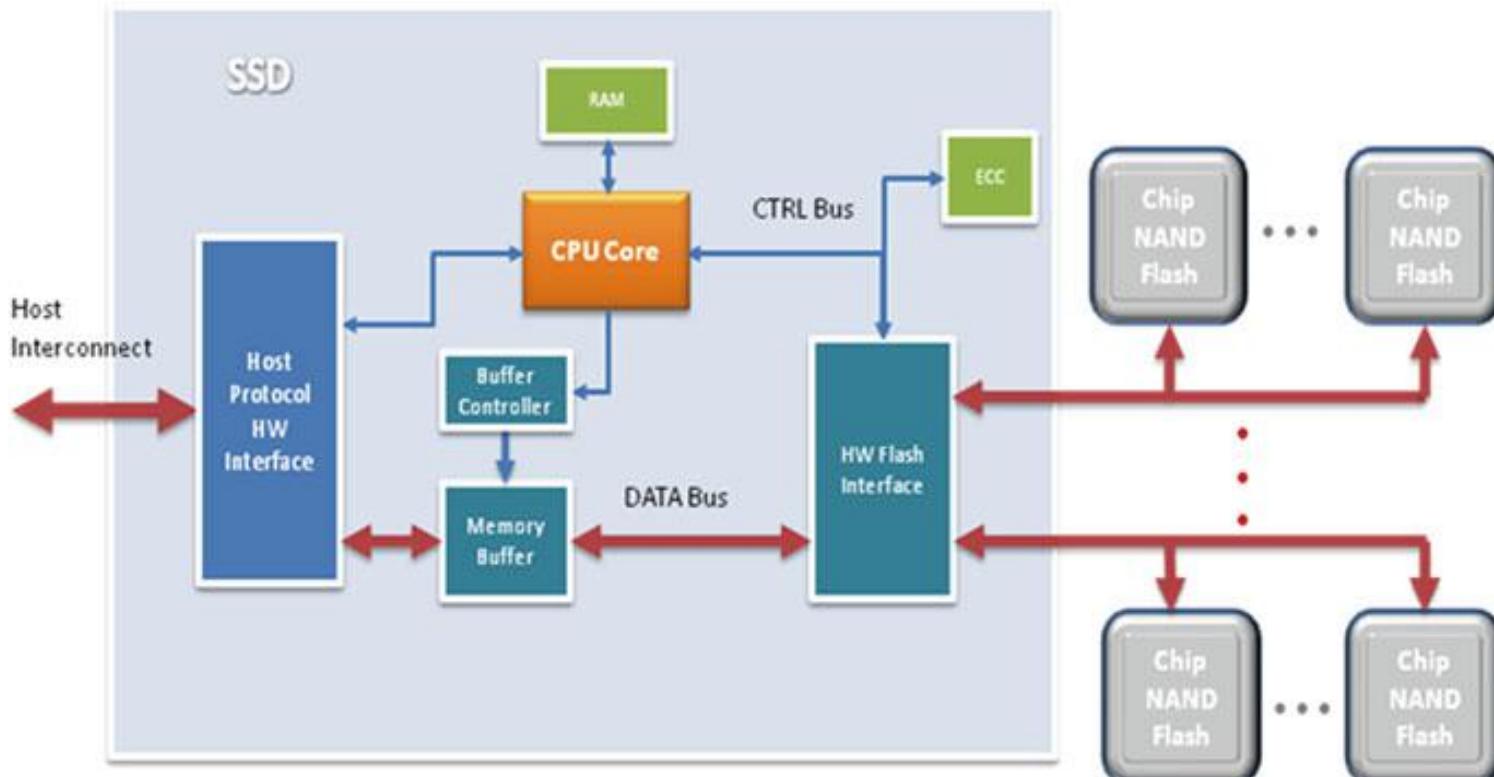
□ SLC：100,000次

□ MLC：10,000次

□ TLC：1,000次

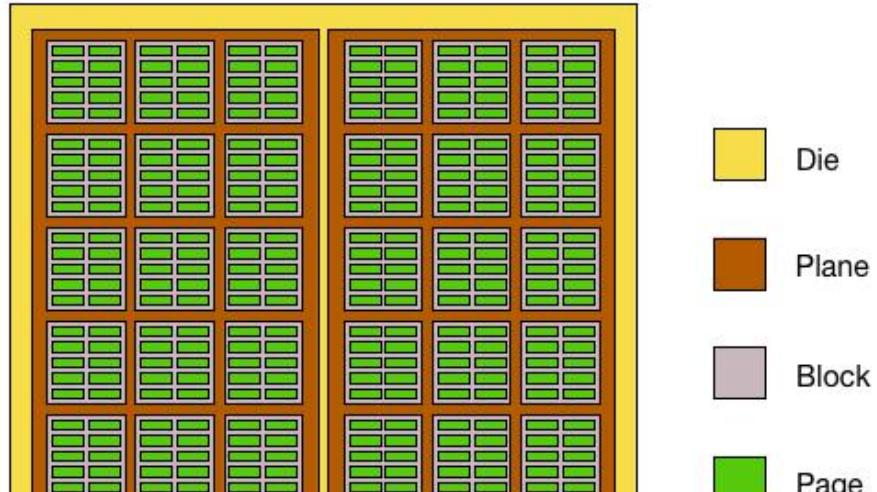
# 从存储器件到固态硬盘

- SSD主要由SSD控制器，FLASH存储阵列，板上DRAM（可选），以及跟HOST接口（诸如SATA, SAS, PCIe等）组成。
- 三个重要组成部分：主存芯片，闪存芯片，固件算法



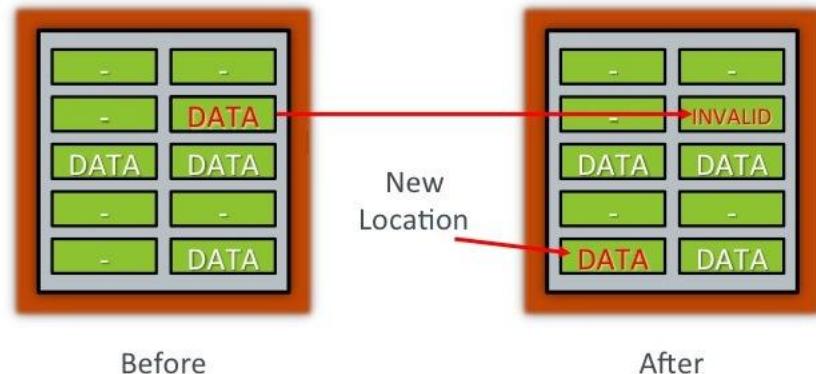
# 固态硬盘存储介质的组织

- 一个 package, 即一个存储芯片, 包含多个Die (典型1, 2, 4个)
- 一个Die包含1个或者2个plane, 可并行操作
- 每一个plane包含多个block, block是最小的擦除单位
- 每一个block里面有很多页, 页是最小的读写单位



- ▶ 闪存页 – 读写粒度
  - ▶ E.g., 4KB, 8KB, 16KB
  - ▶ us延迟
- ▶ 闪存块 – 擦除粒度
  - ▶ E.g., 2MB, 4MB, 8MB
  - ▶ ms延迟

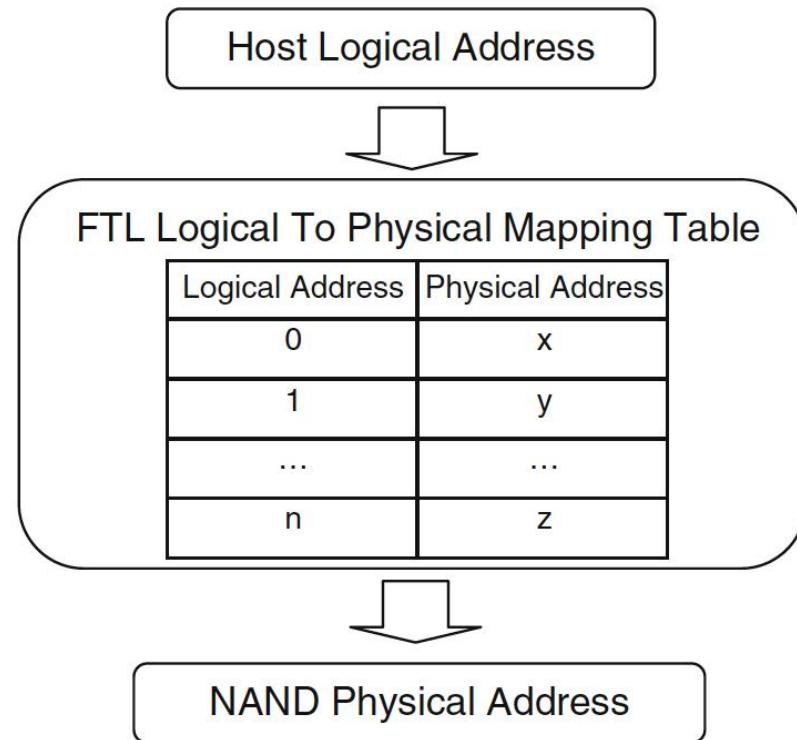
# SSD的写入



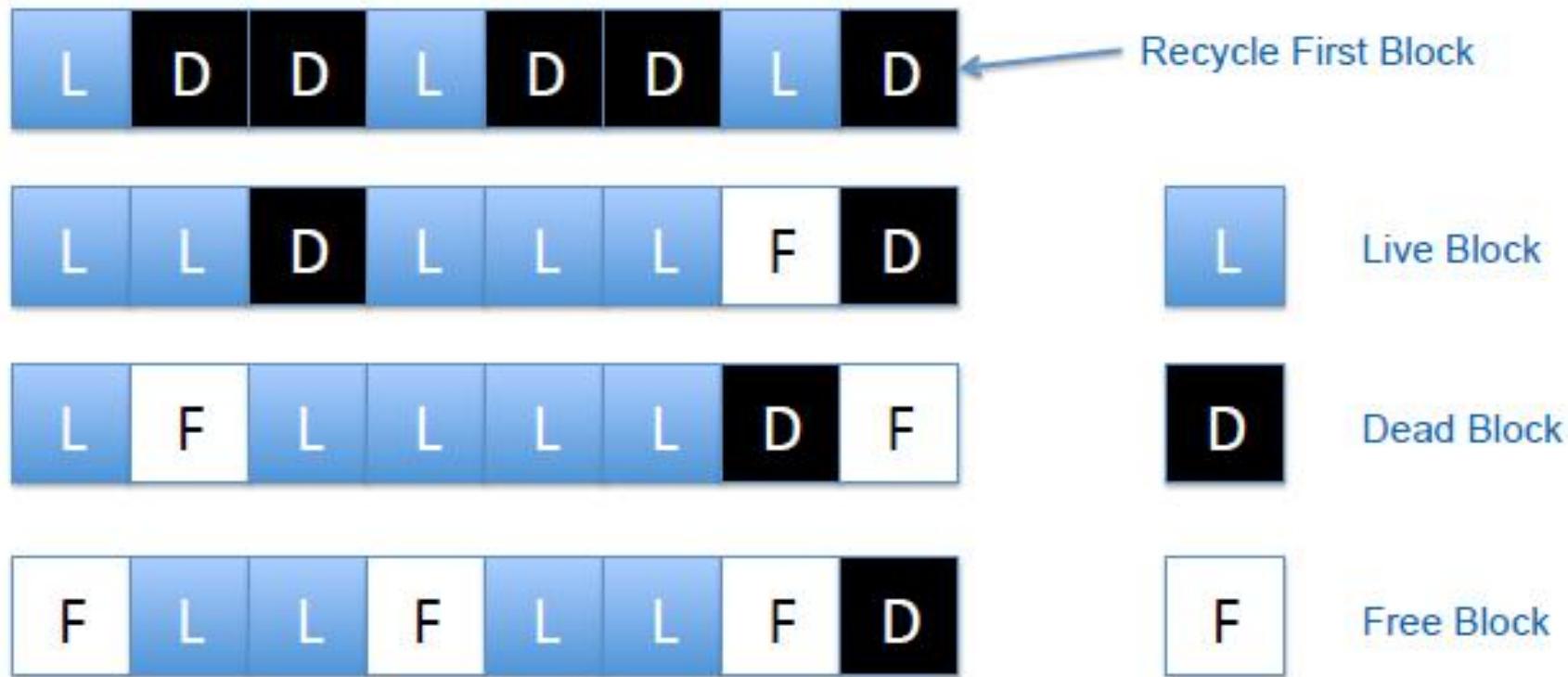
- 与磁盘不一样，不会写入到原来的page，写入之前需要进行擦除操作
- 写入的时候不会在原来的page中写入，会在一个新的页面（可以在同一个块，也可以在不同的块，可能不同的plane，甚至不同的die上面）
- 这样，需要维护上层管理软件的逻辑地址和底层的物理地址之间的映射关系
- 这个工作交给FTL层来完成

# SSD的地址转换FTL

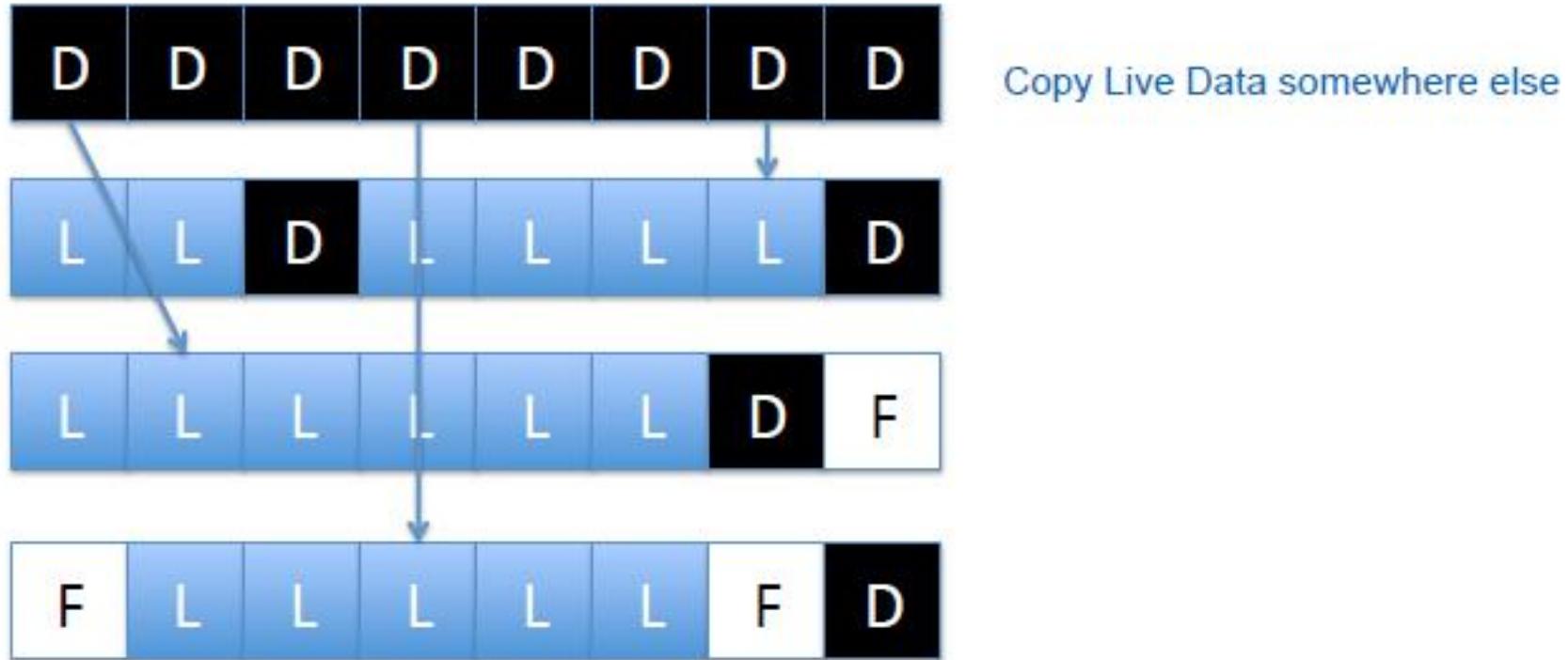
- FTL: Flash Translation Layer, 做逻辑地址到物理地址的翻译
- 除了做地址转换, FTL还帮助完成磨损均衡
- 写入之前必须要进行擦除, 但是每一个块的擦除的次数有限
- 写入的时候需要挑选位于擦除次数最少的块中的页面, 完成磨损均衡



# SSD上的垃圾搜集 (1)



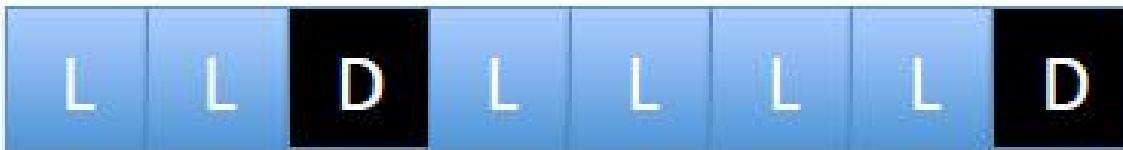
# SSD上的垃圾搜集 (2)



# SSD上的垃圾搜集 (3)

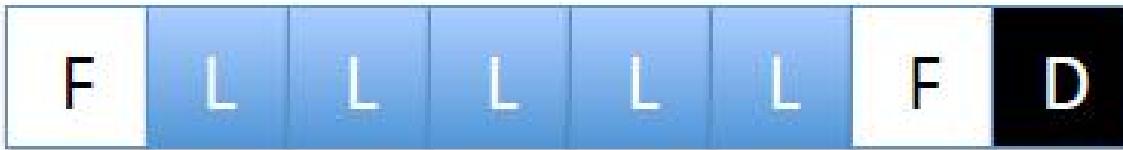
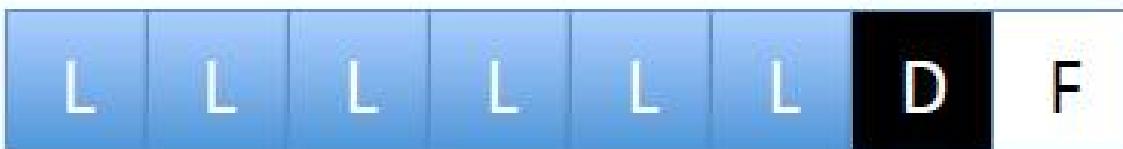


Erase Data Block



Overhead:

- Copying of Live Data
- Block Erasing



# 典型的SSD性能数据

---

□ 依据不同SSD的型号具有不同的性能数据

□ 读数据的性能：

- 20-100微秒的延迟
- 100-500MB/s的带宽

□ 擦除的性能：

- 2ms，即毫秒级

□ 写数据的性能：

- 200微秒的延迟
- 100-200MB/s的带宽

# 最新的性能数据

---

- DELTA MAX SSD硬盘为SATA 6Gbps接口
- 有250GB、500B、1TB及2TB容量
- 其中250GB的读取、写入速度分别是560MB/s、500MB/s
- 其他容量的读取、写入分别是560MB/s、510MB/s
- 4K随机读写最高90K、80K IOPS

# SSD总结

- 机械式存储设备转化为电子式存储设备
- 不同的操作粒度
  - 以页的方式读写
  - 以块的方式擦除
- 擦除次数有限：需要磨损均衡
- FTL：
  - 逻辑地址到物理地址的转换
  - 磨损均衡
  - 垃圾回收

---

谢谢

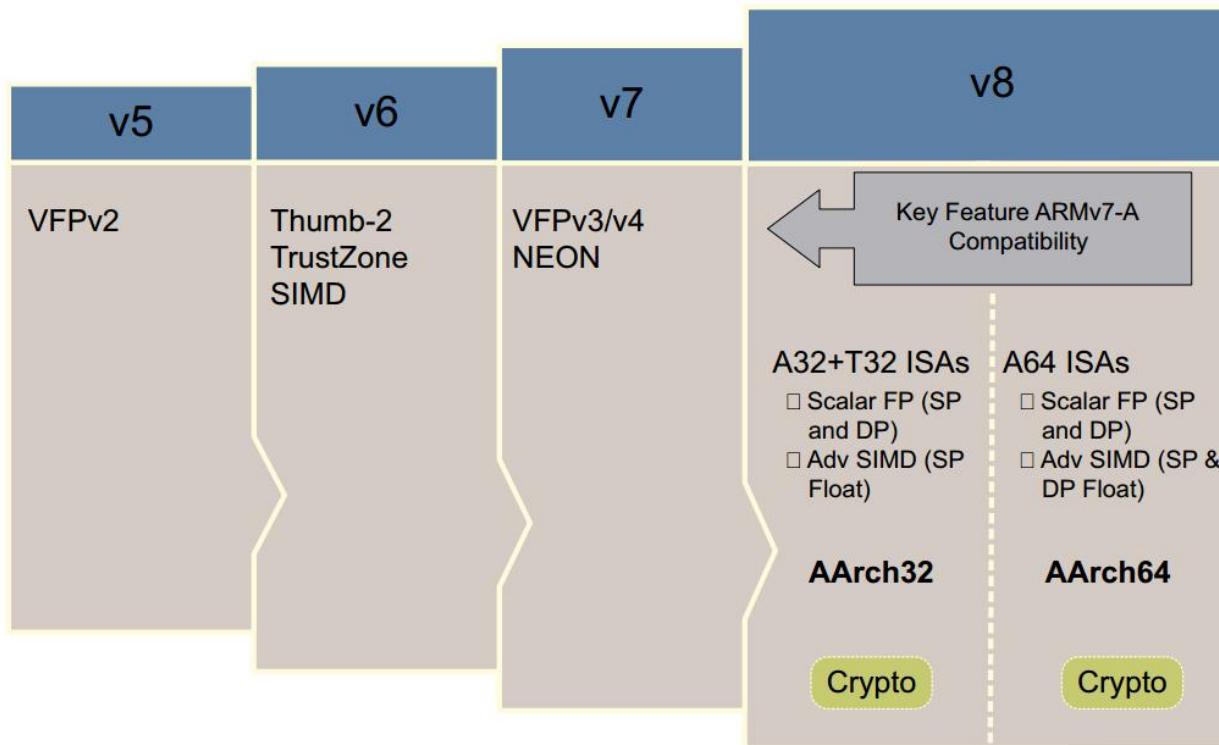
# ARM处理器

华为智能基座支持课程内容

# 内容简介

- ARM处理器（华为鲲鹏）
- ARM系统
- 硬件和操作系统的衔接
- 不同的处理器在针对相同的问题的时候有不同的做法，扩充知识面

# ARM8处理器的发展



<https://winddying.github.io/> Figure 2-1 Development of the ARMv8 architecture

# ARM体系结构规范

- ARM Holdings plc编写ARM体系结构的规范
  - 指令集，包括多媒体/DSP的指令集
  - MMU
  - 中断和异常处理
  - 缓存
  - 虚拟化
  - Etc.
- 开发了很多的版本 ARMv4, ARMv5, ARMv6, ARMv7, ARMv8

ARM® Architecture Reference Manual  
ARMv7-A and ARMv7-R edition

Copyright © 1996-1998, 2000, 2004-2012, 2014 ARM. All rights reserved.  
ARM DDI 0409C.c (ID051414)

ARM®

# ARM cores: 处理器的实际实现

- ARM Holdings plc 也实现 IP 核心，实现规范
- IP核 = VHDL or Verilog 物理逻辑实现arm的规范
- IP核的例子：
  - ARM926 = ARMv5实现
  - ARM1176 = ARMv6实现
  - Cortex-A15 = ARMv7-A实现
  - Cortex-A53 = ARMv8-A实现
- 相同的体系结构会有不同的可能的实现
  - 所有的Cortex-A5,7,8,9,12,15实现的都是ARMv7-A
  - Cortex-A5是低功耗低性能的实现, Cortex-A15高性能，但是也需要高能耗
  - 内部的微体系结构不同：流水线的深度，乱序执行，缓存的大小等
- 设计和实际的制造不同，ARM不卖硬件，只卖设计

# 相同架构下的不同实现

**Table 2-1 Comparison of ARMv8-A processors**

| Processor                        |                                                   |                                                    |
|----------------------------------|---------------------------------------------------|----------------------------------------------------|
|                                  | Cortex-A53                                        | Cortex-A57                                         |
| Release date                     | July 2014                                         | January 2015                                       |
| Typical clock speed              | 2GHz on 28nm                                      | 1.5 to 2.5 GHz on 20nm                             |
| Execution order                  | In-order                                          | Out of order, speculative issue, superscalar       |
| Cores                            | 1 to 4                                            | 1 to 4                                             |
| Integer Peak throughput          | 2.3MIPS/MHz                                       | 4.1 to 4.76MIPS/MHz <sup>a</sup>                   |
| Floating-point Unit              | Yes                                               | Yes                                                |
| Half-precision                   | Yes                                               | Yes                                                |
| Hardware Divide                  | Yes                                               | Yes                                                |
| Fused Multiply Accumulate        | Yes                                               | Yes                                                |
| Pipeline stages                  | 8                                                 | 15+                                                |
| Return stack entries             | 4                                                 | 8                                                  |
| Generic Interrupt Controller     | External                                          | External                                           |
| AMBA interface                   | 64-bit I/F AMBA 4<br>(Supports AMBA 4 and AMBA 5) | 128-bit I/F AMBA 4<br>(Supports AMBA 4 and AMBA 5) |
| L1 Cache size (Instruction)      | 8KB to 64 KB                                      | 48KB                                               |
| L1 Cache structure (Instruction) | 2-way set associative                             | 3-way set associative                              |
| L1 Cache size (Data)             | 8KB to 64KB                                       | 32KB                                               |
| L1 Cache structure (Data)        | 4-way set associative                             | 2-way set associative                              |
| L2 Cache                         | Optional                                          | Integrated                                         |
| L2 Cache size                    | 128KB to 2MB                                      | 512KB to 2MB                                       |
| L2 Cache structure               | 16-way set associative                            | 16-way set associative                             |
| Main TLB entries                 | 512                                               | 1024                                               |
| uTLB entries                     | 10                                                | 48 I-side<br>32 D-side                             |

A. IMPLEMENTATION DEFINED

# ARM 处理器

- 64位整数运算与编址
- 4个执行优先级
- 支持虚拟地址
- 支持多核
- 集成了协处理器
  - 16/32/64-bit 浮点数
  - SIMD单指令多数据的指令
  - 可选的密码学算法功能
- 多任务支持
- 通常会在一颗芯片中集成CPU和GPU
- 应用范围： Apple iPhone, iPad, Android, Raspberry Pi

# 寄存器

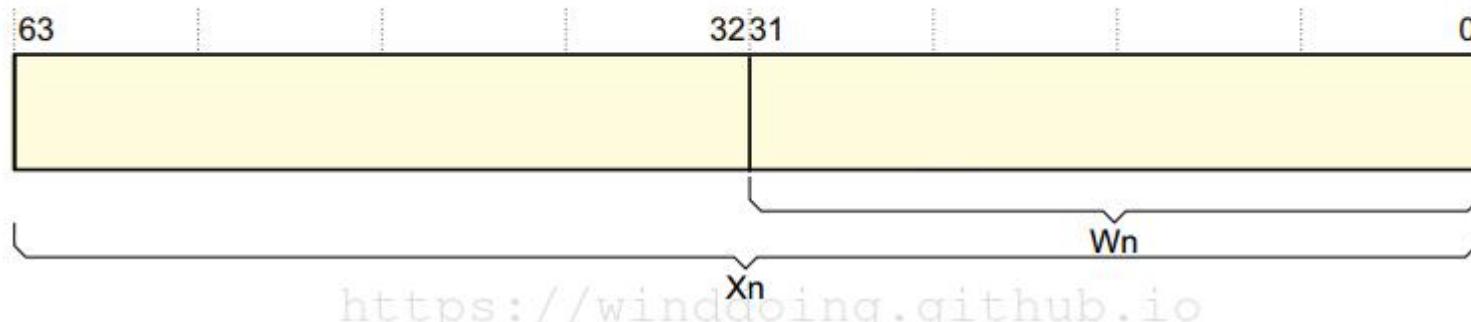
- 31 64-bit 整数寄存器, x0 ~ x30
- XZR 常数0寄存器
- SP 栈指针
  - XZR 和 SP 实际上都是 x31 (如何区分? )
- 低32-位命名为 w0 ~ w30 以及 WZR
- PC : Program Counter
- SPSR : Program Status Register
- 针对SPSR, SP在各个执行级别上都有影子寄存器, 用以对多任务以及中断进行支持

# ARMv8寄存器

- AArch 拥有 31 个通用寄存器，系统运行在 64 位状态下的时候名字叫 Xn，运行在 32 位的时候就叫 Wn.

**Figure 4-1 AArch64 general-purpose registers**

Each AArch64 64-bit general-purpose register (X0-X30) also has a 32-bit (W0-W30) form.

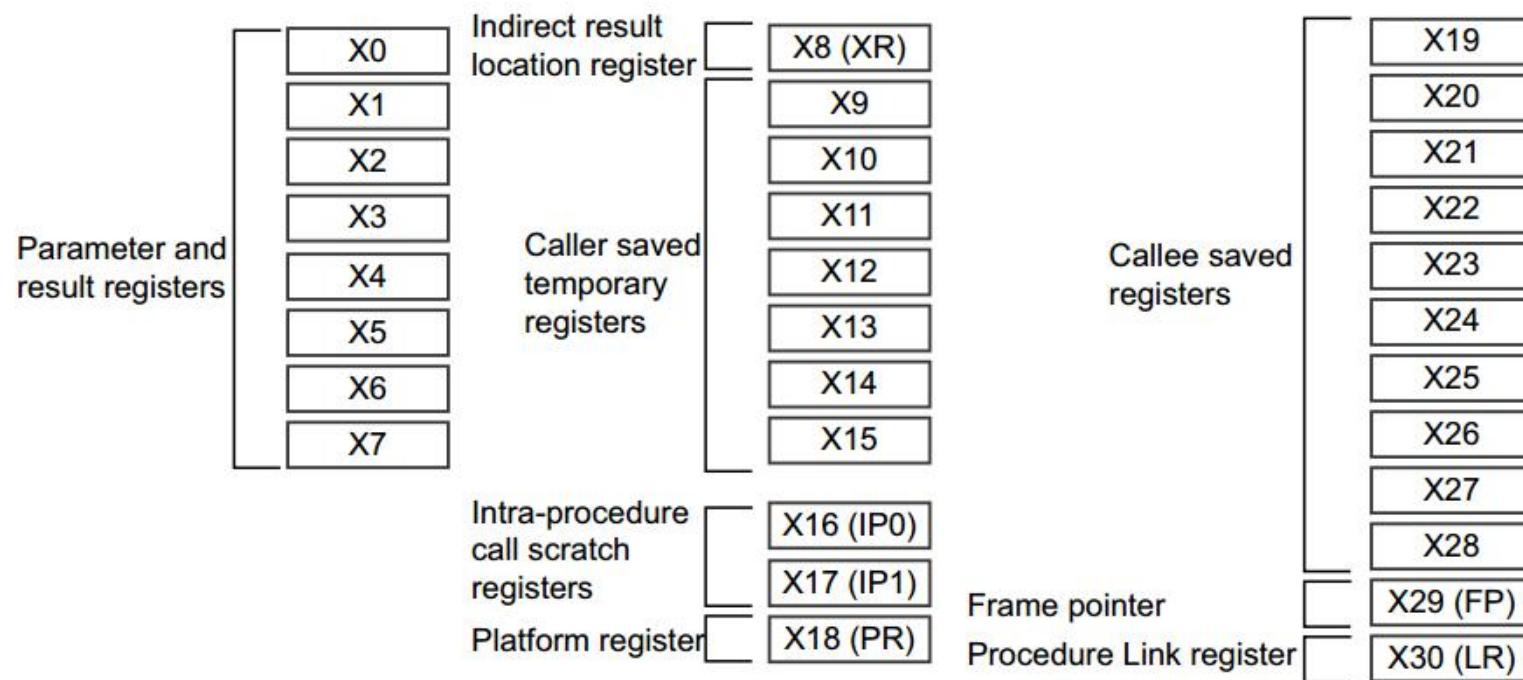


# 特殊寄存器

|                   |                         |          |     |          |          |
|-------------------|-------------------------|----------|-----|----------|----------|
| Special registers | Zero register           | XZR/WZR  |     |          |          |
|                   | Program counter         | PC       |     |          |          |
|                   | Stack pointer           | SP_EL0   |     | SP_EL1   | SP_EL2   |
|                   | Program Status Register | SPSR_EL1 |     | SPSR_EL2 | SPSR_EL3 |
|                   | Exception Link Register | ELR_EL1  |     | ELR_EL2  | ELR_EL3  |
|                   |                         | EL0      | EL1 | EL2      | EL3      |

Figure 4-3 AArch64 special registers

# 通用寄存器组



<https://winddding.github.io/> Figure 9-1 General-purpose register use in the ABI

# 寄存器使用惯例

- 参数寄存器 (X0-X7) : 用作临时寄存器或可以保存的调用者保存的寄存器变量函数内的中间值，调用其他函数之间的值 (8个寄存器可用于传递参数)
- 调用者保存的临时寄存器 (X9-X15) : 如果调用者要求在任何这些寄存器中保留值调用另一个函数，调用者必须将受影响的寄存器保存在自己的堆栈中。它们可以通过被调用的子程序进行修改，而无需保存并在返回调用者之前恢复它们
- 被调用者保存的寄存器 (X19-X29) : 这些寄存器保存在被调用者帧中。它们可以被被调用者修改子程序，只要它们在返回之前保存并恢复
- 特殊用途寄存器 (X8, X16-X18, X29, X30) :
  - X8: 是间接结果寄存器，用于保存子程序返回地址，尽量不使用
  - X16 和 X17: 程序内调用临时寄存器
  - X18: 平台寄存器，保留用于平台 ABI，尽量不使用
  - X29: 帧指针寄存器 (FP)
  - X30: 链接寄存器 (LR)
  - X31: 堆栈指针寄存器 SP 或零寄存器 ZXR

# 32-bit 指令

- 每条指令4个字节
  - 但是寄存器保存了64位的值，地址同样是64位
- ARM64 有31个整数寄存器
  - 寄存器地址编码需要5位
- Limits the number of bits used for the opcode
  - Hence reduced number of instructions
- 立即数要远远小于整数表达范围

| 31       | 30     | 29                 | 28-24   | 23-22 | 21 | 20-16 | 15-10 | 9-5 | 4-0 |
|----------|--------|--------------------|---------|-------|----|-------|-------|-----|-----|
| Bits w/x | opcode | Set condition code | opcod e | shift | 0  | rm    | imm6  | rn  | rd  |

# 0寄存器

- MOV X3, #3 是伪指令：
  - ORR X3, XZR, #3
- CMP X3, #3 也是伪指令
  - SUBS XZR, X3, #3
- 通过0寄存器可以减少opcode的使用
- 使用伪指令提高了程序的可读性
- RISC-V 情况类似

# ALU

- ADD, SUB, MUL, DIV
- 乘加指令
- 比较指令
- 逻辑指令 AND, XOR
- 位操作指令

# 如何装入一个64位的寄存器1

- 最大的立即数操作数是16位
  - 大部分的立即数都比较小
- 可以使用4条指令来装入
- //0x1234FEDC4F5D6E3A装入到x2寄存器
- MOV X2, #0xE3A
- MOVK X2, #0x4F5D, LSL #16
- MOVK X2, #0xFEDC, LSL #32
- MOVK X2, #0x1234, LSL #48

# 如何装入一个64位的寄存器2

- 引用另外一个寄存器
  - 例如SP 或者 PC
- LDR X1, = 0x1234ABCD1234ABCD
- 汇编为以下的语句
  - ldr x1, #8 //相对于pc的偏移
  - .quad 0x1234abcd1234abcd
- LDR X1,=helloworld
- ...
- Helloworld:"Hello World"

# LDR指令

- LDR{条件} 目的寄存器 <存储器地址>

作用：将 存储器地址 所指地址处连续的4个字节（1个字）的数据传送到目的寄存器中，比riscv灵活很多

LDR R0, [R1] ; 将存储器地址为R1的字数据读入寄存器R0

LDR R0, [R1,R2] ; 将存储器地址为R1+R2的字数据读入寄存器R0

LDR R0, [R1,#8] ; 将存储器地址为R1+8的字数据读入寄存器R0

LDR R0, [R1],R2 ; 将存储器地址为R1的字数据读入寄存器R0，然后R1=R2

LDR R0, [R1],#8 ; 将存储器地址为R1的字数据读入寄存器R0，并将R1+8的值存入R1

LDR R0, [R1,R2]! ; 将存储器地址为R1+R2的字数据读入寄存器R0，并将R1+R2的值存入R1

LDR R0, [R1,LSL #3] ; 将存储器地址为R1\*8的字数据读入寄存器R0

LDR R0, [R1,R2,LSL #2] ; 将存储器地址为R1+R2\*4的字数据读入寄存器R0

LDR R0, [R1,,R2, LSL #2]! ; 将存储器地址为R1+R2\*4的字数据读入寄存器R0，并将R1+R2\*4的值存入R1

LDR R0, [R1],R2, LSL #2 ; 将存储器地址为R1的字数据读入寄存器R0，并将R1+R2\*4的值存入R1

LDR R0, Label ; Label为程序标号，Label必须是当前指令的-4~4KB范围内

# Load/Store体系结构

- LDR x2, [x1]
- LDR X3, [x1, #8]
- ADD x4, X2, X3
- STR X4, [X1]

# 数据同步

- LDR X0, [X1]
- Wait: CMP X0, #0
- BNE wait
- ADD X0, #1
- STR X0, [X1]
- LDR X0, [X1]
- Wait: CMP X0, #0
- BNE wait
- ADD X0, #1
- STR X0, [X1]
- 两个核同时执行上述的指令，会导致锁的失败
- 需要引入原子指令来解决上述的问题
- 实现方案：锁总线，原子增加
- Load Acquire/Store Release 指令 (LDXR 以及 STLXR)

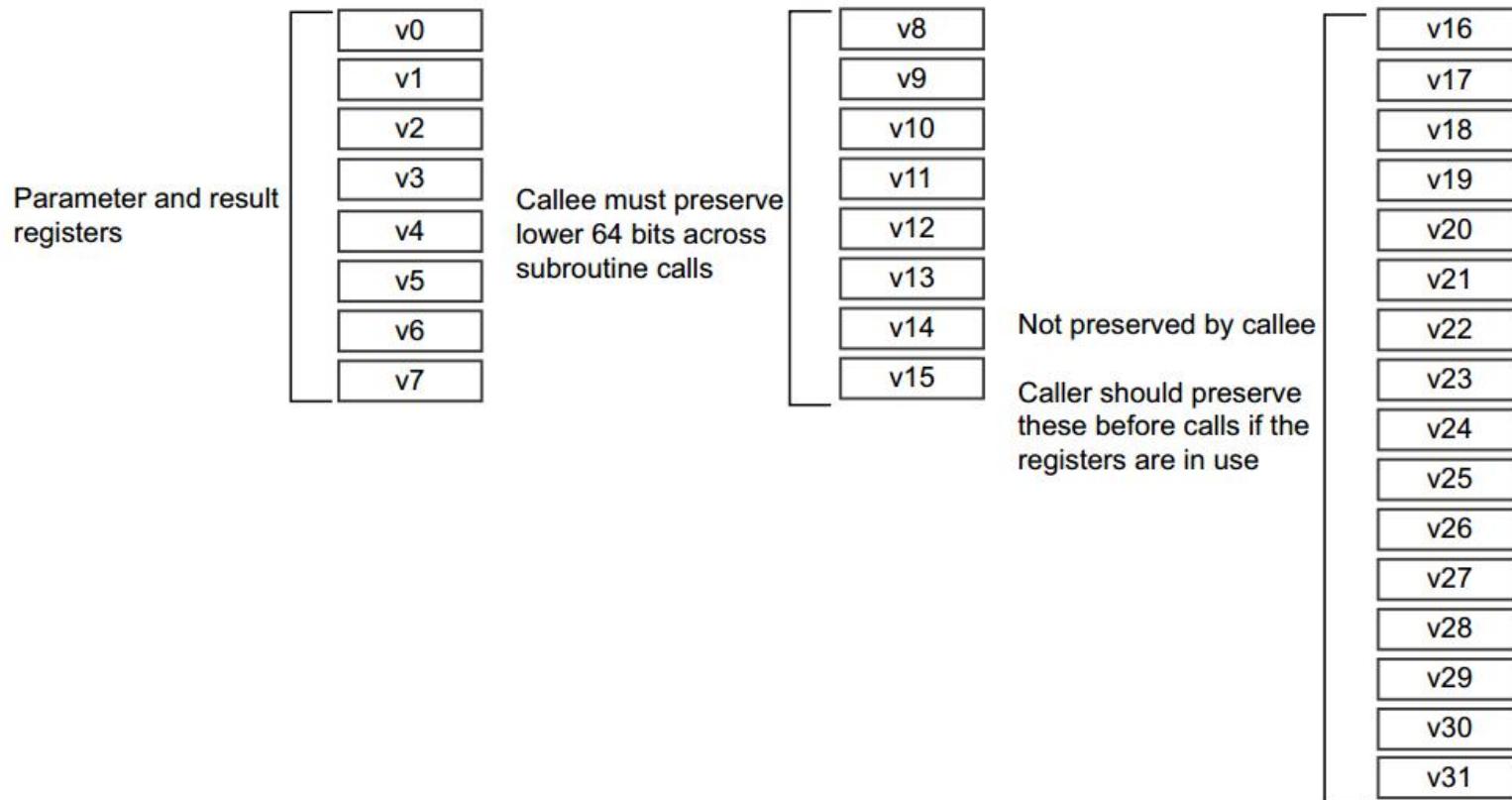
# 协处理器

- Floating Point Unit (FPU)
- NEON SIMD
- 32 128-bit registers FPU和NEON共享

| Bits                    | 127 – 112 | 111 – 96 | 95 – 80 | 79 – 64           | 63 – 48                                | 47 - 32                 | 31 – 16      | 15 – 0 |
|-------------------------|-----------|----------|---------|-------------------|----------------------------------------|-------------------------|--------------|--------|
| 128-bit NEON V Register |           |          |         | All 128-bits used |                                        |                         |              |        |
| 64-bit FPU D Register   |           | Set to 0 |         |                   | 64-bit double precision floating point |                         |              |        |
| 32-bit FPU S Register   |           | Set to 0 |         |                   |                                        | 32-bit single precision |              |        |
| 16-bit FPU H Register   |           | Set to 0 |         |                   |                                        |                         | 16-bit float |        |

- LDP S2, S3, [X0]
- FSUB S4, S2, S0

# NEON和浮点寄存器



<https://windd.org/guides/abi.html> Figure 9-3 SIMD and floating-point registers in the ABI

# NEON Lanes

| V1     |   |   |   |                     |   |   |   |        |   |   |   |        |   |   |   |
|--------|---|---|---|---------------------|---|---|---|--------|---|---|---|--------|---|---|---|
| D      |   |   |   |                     |   |   |   | D      |   |   |   |        |   |   |   |
| S      |   |   |   | S                   |   |   |   | S      |   |   |   | S      |   |   |   |
| H      | H | H | H | H                   | H | H | H | H      | H | H | H | H      | H | H | H |
| B      | B | B | B | B                   | B | B | B | B      | B | B | B | B      | B | B | B |
| Lane 1 |   |   |   | Lane 2              |   |   |   | Lane 3 |   |   |   | Lane 4 |   |   |   |
| V0     | 4 |   |   | 3                   |   |   |   | 6      |   |   |   | 7      |   |   |   |
| V1     | 2 |   |   | 5                   |   |   |   | 8      |   |   |   | 9      |   |   |   |
| ADD    |   |   |   | V2.4S, V0.4S, V1.4S |   |   |   |        |   |   |   |        |   |   |   |
| V2     | 6 |   |   | 8                   |   |   |   | 14     |   |   |   | 16     |   |   |   |

V1.2D  
V1.4S  
V1.8H  
V1.16B

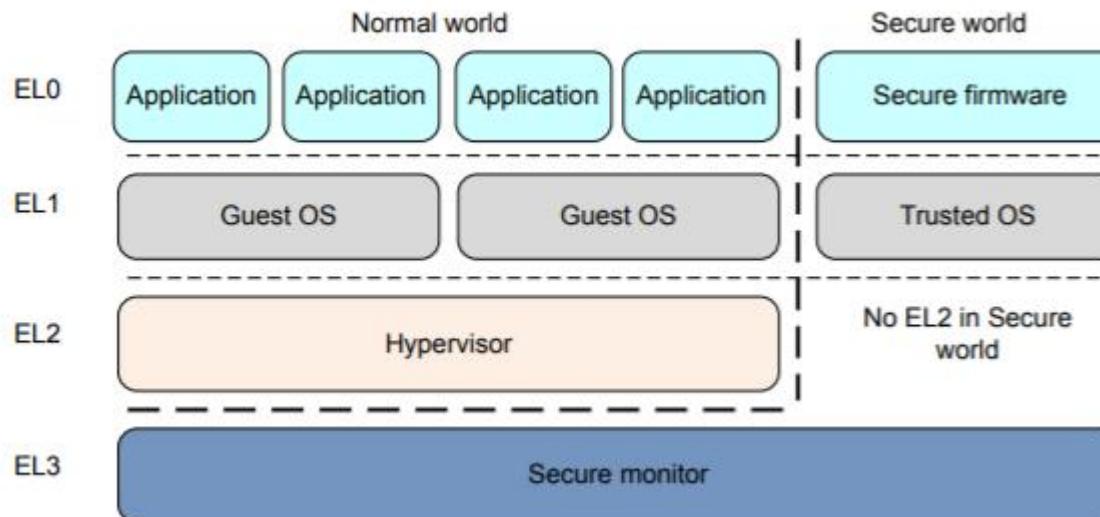
# Linux uses NEON for Encryption

- For instance: arch/arm64/crypto/aes\_neon.S
- /\* apply SubBytes transformation using the the preloaded Sbox \*/

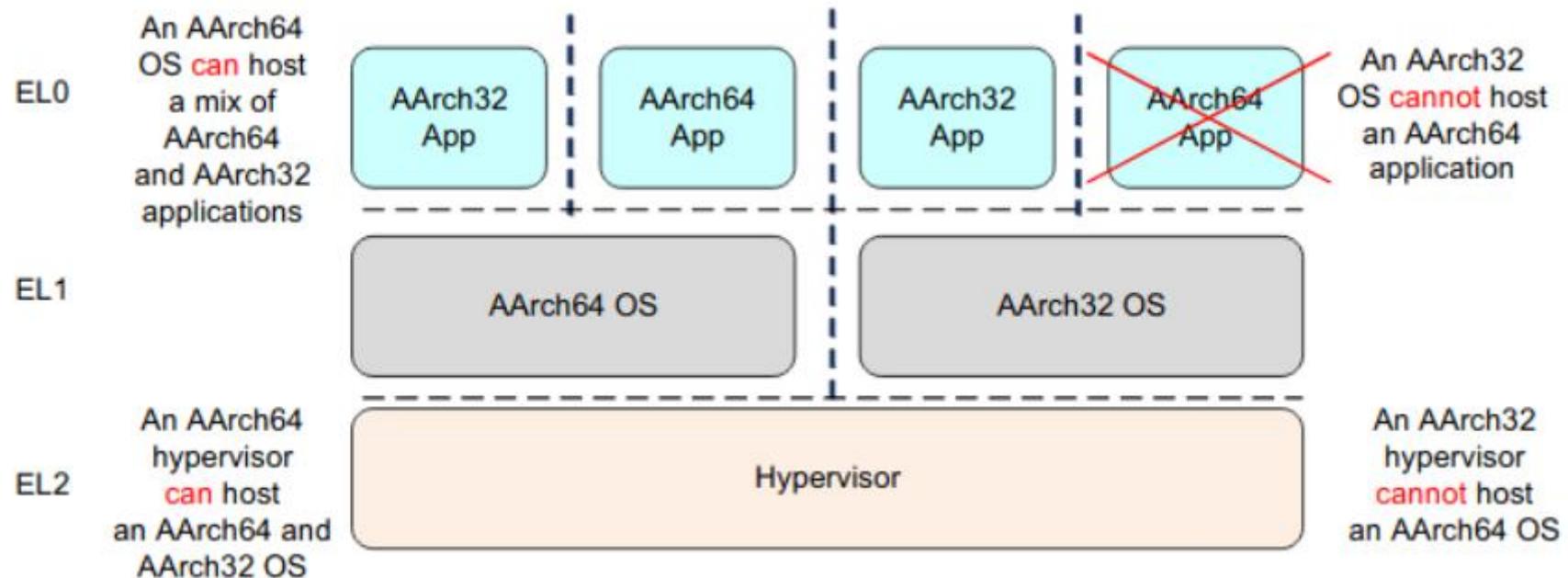
```
.macro sub_bytes, in
 sub v9.16b, \in\().16b, v15.16b
 tbl \in\().16b, {v16.16b-v19.16b}, \in\().16b
 sub v10.16b, v9.16b, v15.16b
 tbx \in\().16b, {v20.16b-v23.16b}, v9.16b
 sub v11.16b, v10.16b, v15.16b
 tbx \in\().16b, {v24.16b-v27.16b}, v10.16b
 tbx \in\().16b, {v28.16b-v31.16b}, v11.16b
.endm
```

# 异常等级

- EL0: 普通用户应用程序
- EL1: 操作系统内核通常被描述为特权
- EL2: 管理程序
- EL3: 低级固件, 包括安全监视器



# 执行状态



# 用于异常处理的特殊寄存器

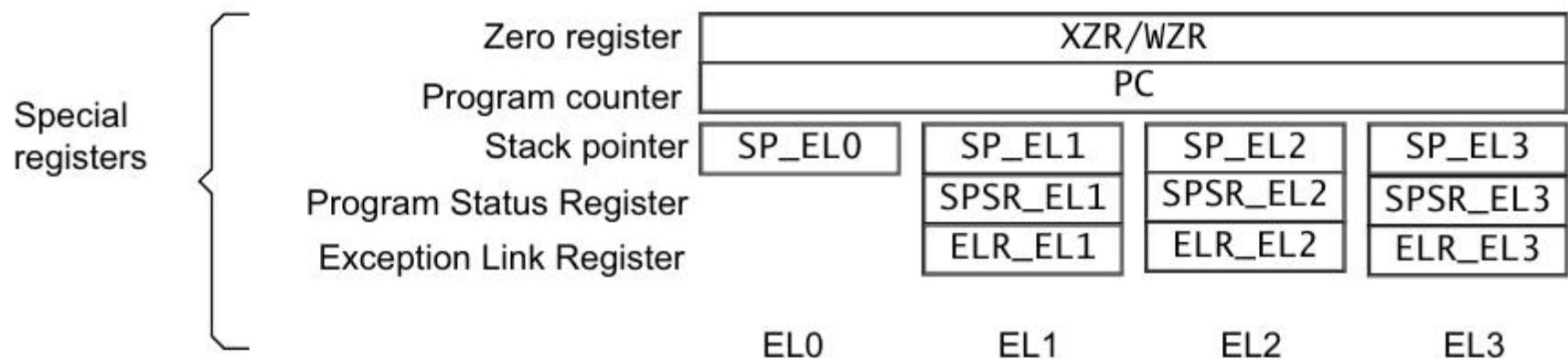
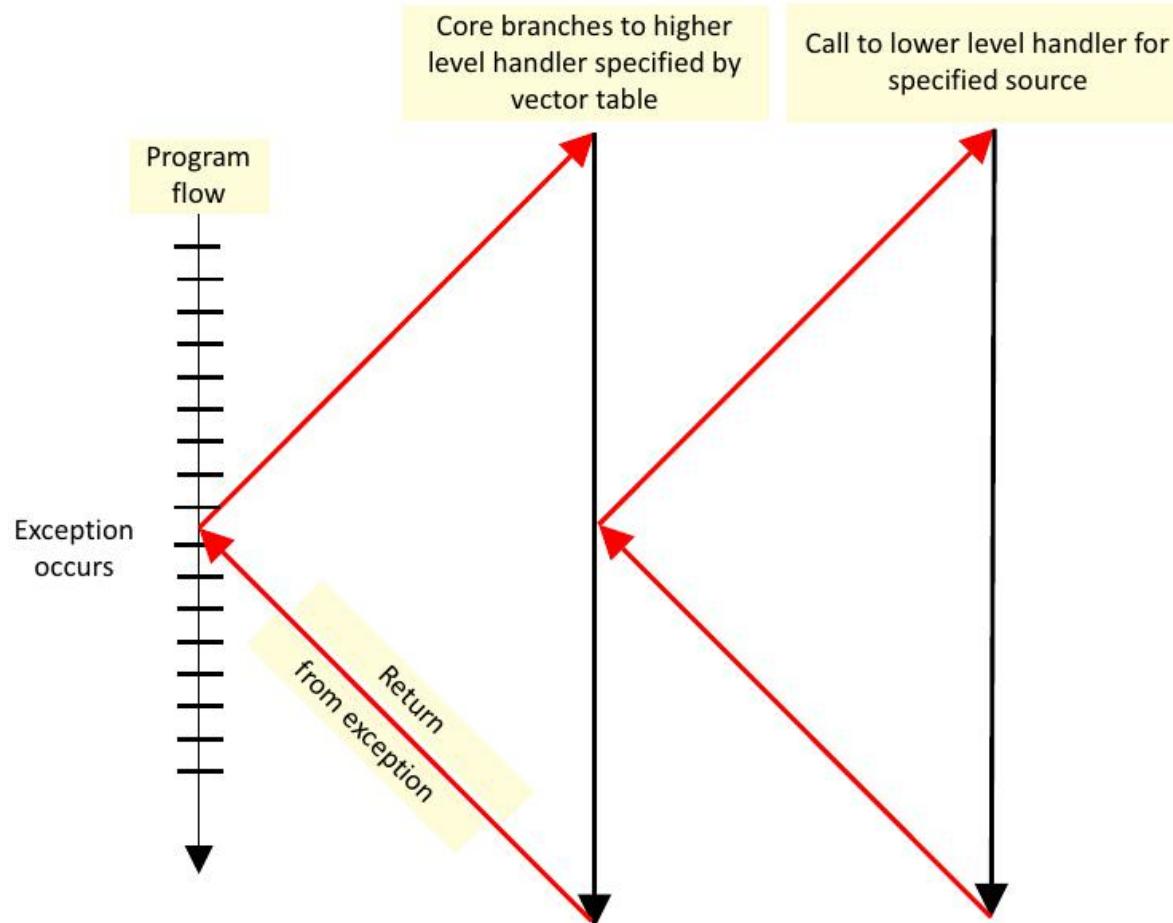


Figure 4-3 AArch64 special registers

# 异常处理的通用流程



<https://winddoing.github.io> Figure 10-1 Exception flow

# 异常发生时的硬件操作

- 处理器状态保存到目标异常级别的 SPSR\_ELx 中
- 返回地址保存到目标异常级别的 ELR\_ELx 中。
- 如果异常是同步异常或SError 中断，异常的表征信息将保存在目标异常级别的 ESR\_ELx 中
- 如果是指令止异常(Instruction Abort exception), 数据中止异常(Data Abort exception,), PC对齐错误异常(PC alignment fault exception), 故障的虚拟地址将保存在 FAR\_ELx 中
- 堆栈指针保存到目标异常级别的专用堆栈指针寄存器 SP\_ELx
- 执行移至目标异常级别，并从异常向量定义的地址开始执行

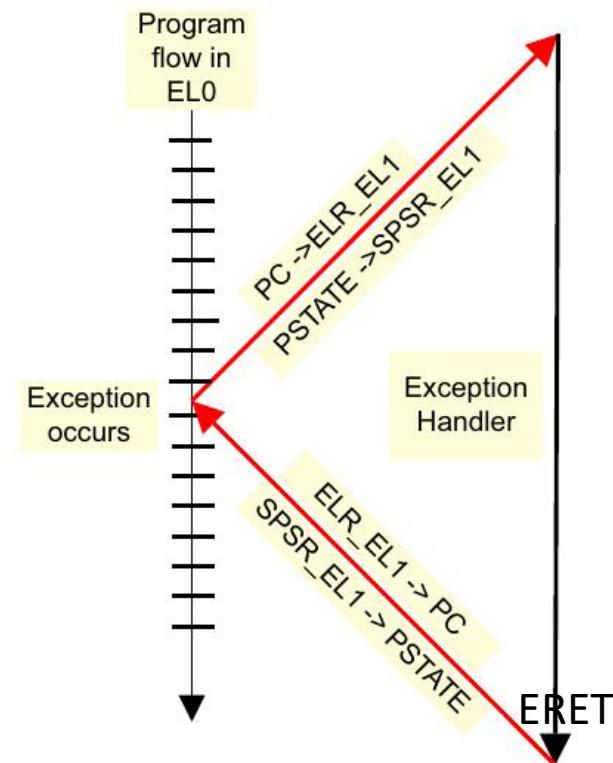


Figure 10-4 Exception handling  
<https://winddoing.github.io>

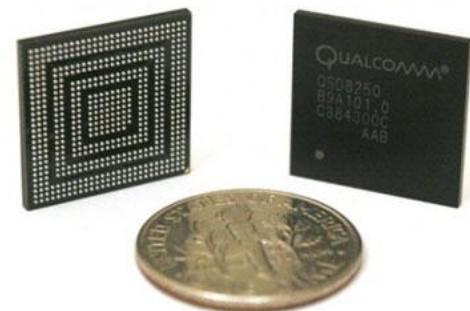
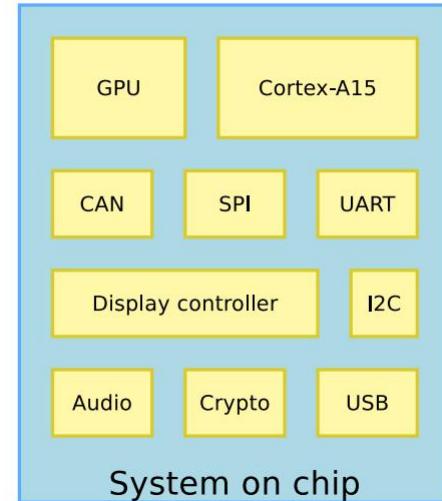
# 状态寄存器的信息

- 条件标志位NZCV
- 异常掩码位DAIF，异常掩码位 (DAIF) 允许屏蔽异常事件，设置该位时不发生异常。
- SPSel: SP 选择 (EL0 或 ELn)，不适用于 EL0
- E: 数据字节序 (仅限 AArch32)

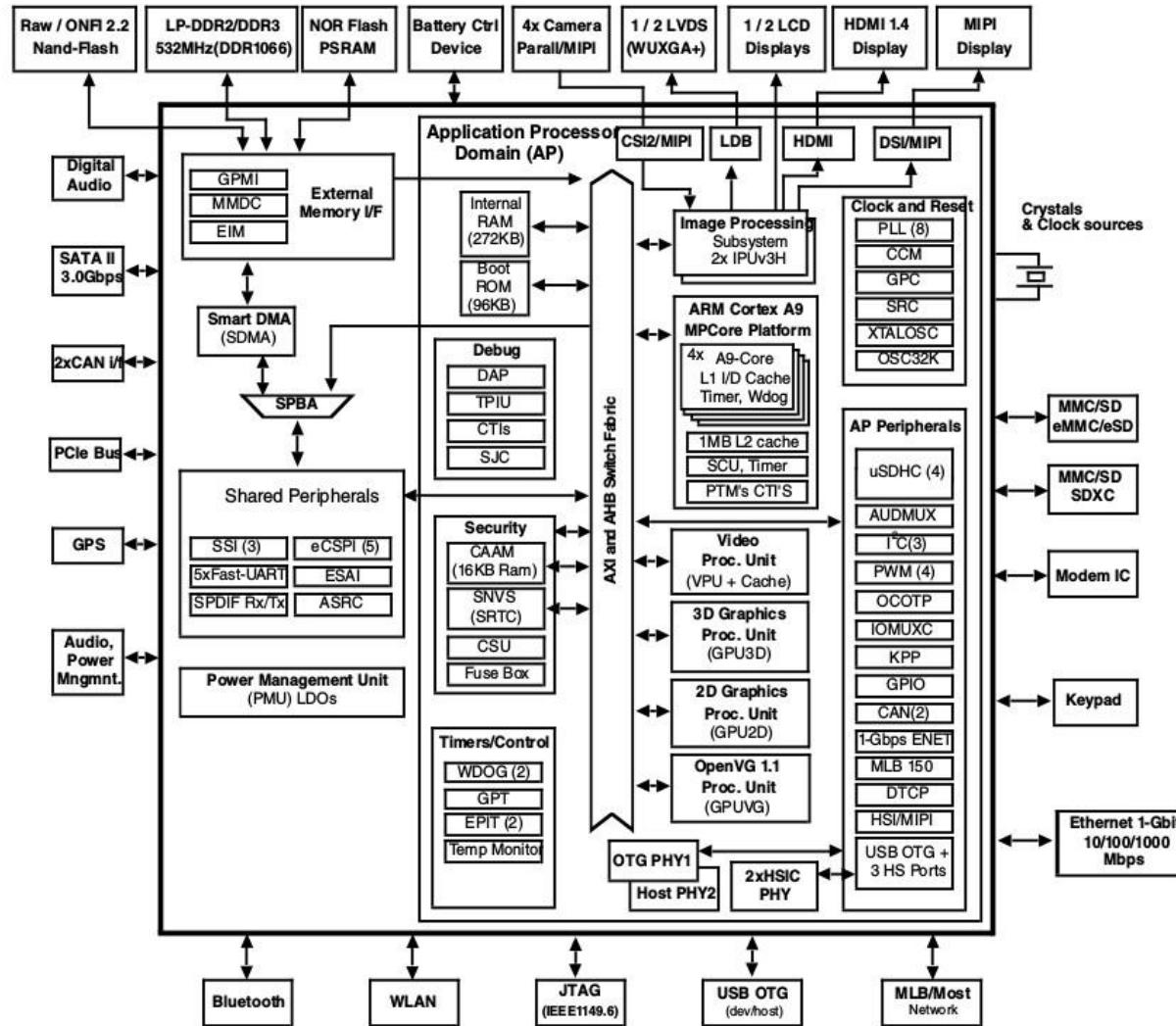
Arm系统

# ARM System-on-Chip

- 片上系统：集成了计算机系统所有组件的集成电路
  - CPU, 以及外设: Ethernet, USB, UART, SPI, I2C, GPU, display, audio, etc.
  - 集成在一个芯片中：更容易使用，更具成本效益
- SoC供应商
  - 从ARM公司购买一个ARM内核
  - 集成其他IP模块，可以是内部设计的，也可以是向其他供应商购买的
  - 创造和销售产品
- 大范围的SoCs，可满足非常不同的市场：汽车、移动、工业、低功耗、机顶盒等。

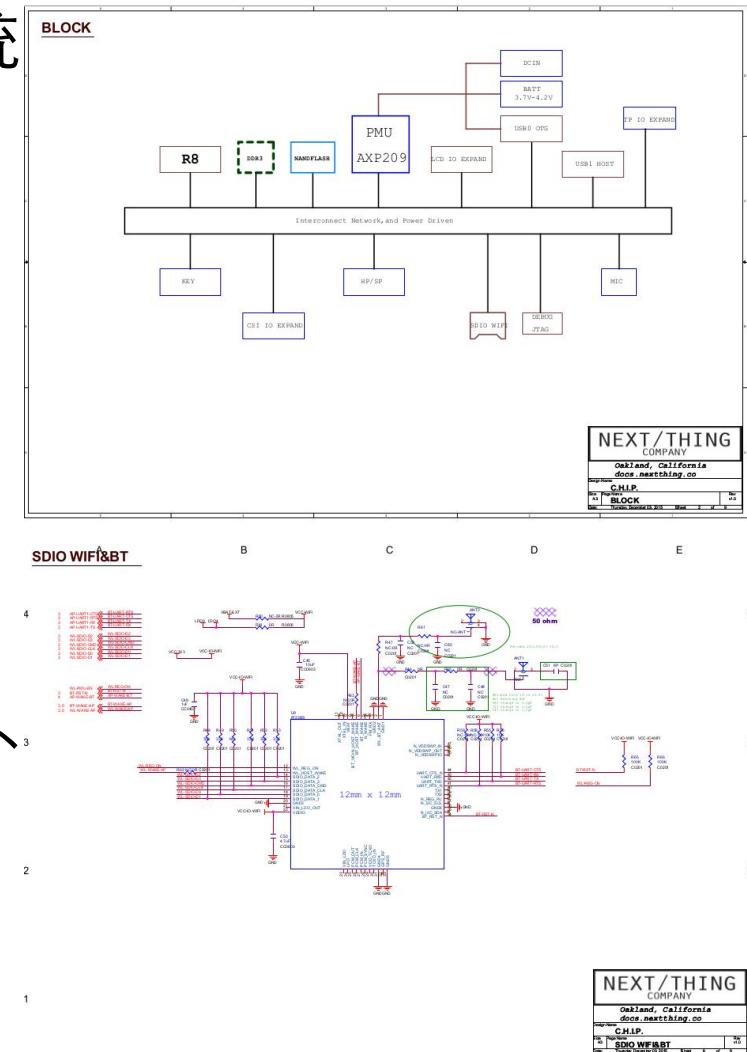


# SoC example: Freescale i.MX6 block diagram

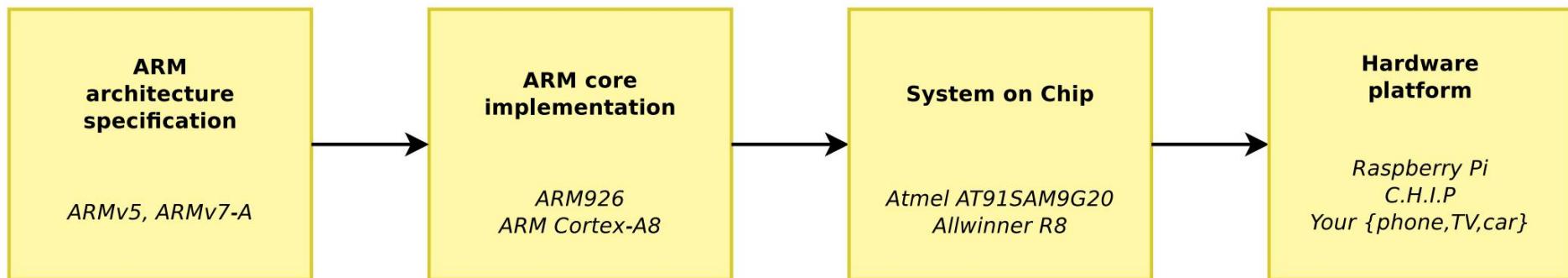


# ARM hardware platform

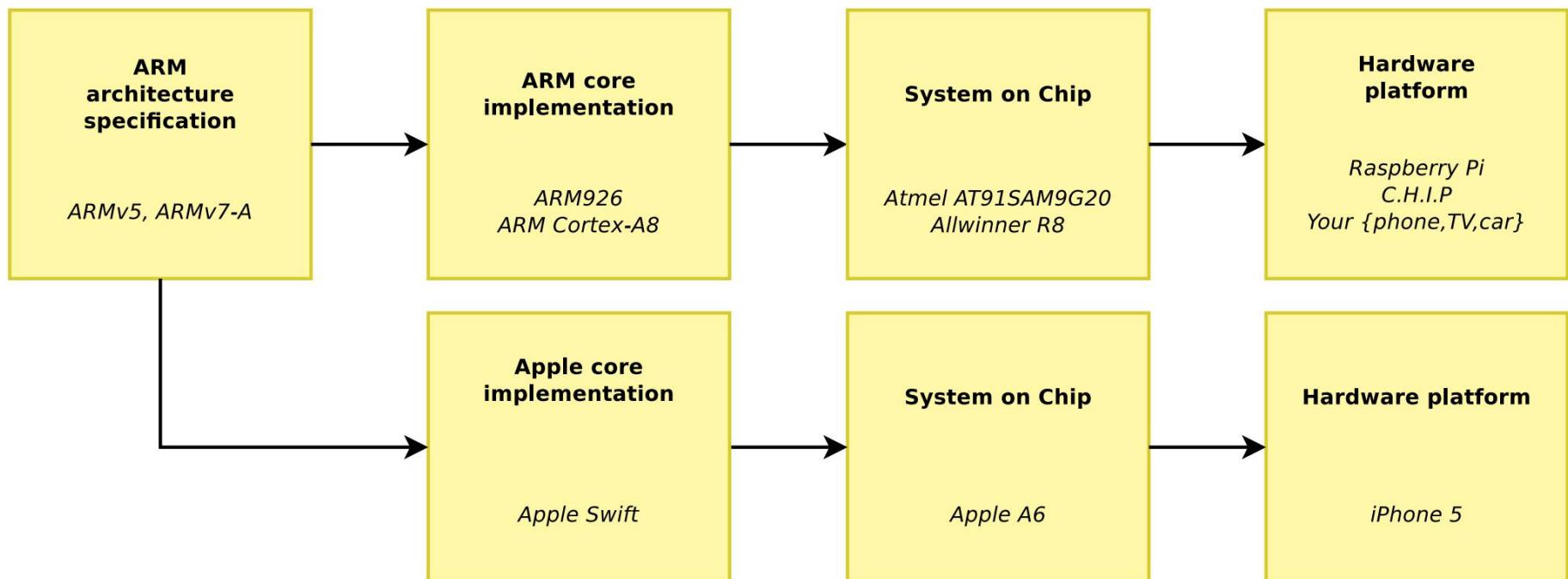
- 尽管SoC是一个芯片上的完整系统  
但它通常还不是完整硬件平台
  - RAM, NAND flash or eMMC, power circuitry
  - Display panel and touchscreen
  - WiFi and Bluetooth chip
  - Ethernet PHY
  - HDMI transceiver
  - CAN transceiver
  - Connectors
- SoC通过各种总线与各种各样的外设相连
- 铺设在印刷电路板上，上面焊接有元件



# ARM: 从架构到开发板

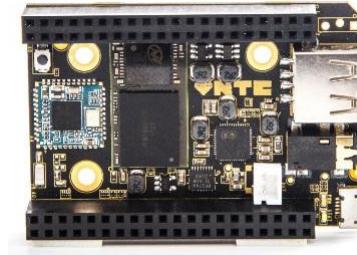


# 从架构到开发板



# ARM开发板的例子

- RaspberryPi 1
  - SoC: Broadcom 2835
  - ARM core: ARM1176JZF (single)
  - ARM architecture: ARMv6
- RaspberryPi 2
  - SoC: Broadcom 2836
  - ARM core: Cortex-A7 (quad)
  - ARM architecture: ARMv7-A
- C.H.I.P
  - SoC: Allwinner R8
  - ARM core: Cortex-A8 (single)
  - ARM architecture: ARMv7-A
- ESPRESSOBIN
  - SoC: Marvell Armada 3700
  - ARM core: Cortex-A53 (dual)
  - ARM architecture: ARMv8-A



# 对硬件层的软件支持

- 问 "Linux是否支持ARM? "并没有什么意义
- 三个 "级别 "的硬件， 三个 "级别 "的软件支持
  - 1. ARM core
  - 2. SoC
  - 3. board
- 所有这三个级别都需要支持
- 仅用串行端口和以太网支持一个平台与完全支持一个平台（图形、音频、电源管理等）是非常不同的。

# 缺乏标准化

- 指令集在所有ARMv7内核之间、所有ARMv8内核之间兼容
  - 可以在任何ARMv7平台上运行为ARMv7构建的Linux用户空间代码（只要不与特定硬件相关）。
  - 一些可选的功能（如NEON）
  - 允许在任何ARMv7平台上运行Ubuntu（为ARMv7构建）
  - Ubuntu（为ARMv7构建）将不能在RaspberryPi 1（ARMv6）上运行
- 然而，其他硬件组件几乎没有标准化：SoC内部和电路板
  - 需要在每个SoC和电路板的引导程序和Linux内核层面上进行具体处理
  - 在大多数ARM SoC上，芯片内部的硬件是内存映射的。没有动态发现的能力

# 没有标准化，但有大量的硬件 重复使用

- 处理器内核的兼容性：符合ARM规范
- 对于其他硬件块，SoC供应商通常会
  - 从第三方供应商那里购买IP块。ARM、Cadence、Synopsys、Mentor Graphics、Imagination Technologies，等等
  - 在其不同的SoC之间广泛地重复使用IP块
- 举例：
  - Mentor Graphics MUSB（USB小工具控制器）用于TI、Allwinner和ST SoC，也用于Blackfin和一些MIPS处理器
  - Marvell的SPI控制器在15年以上的Marvell处理器中被重新使用，ARMv5 Orions→ARMv8处理器
- 这使得我们可以大规模地重复使用驱动程序
- 有时，要弄清楚不同SoC中的两个IP块实际上是相同的并不那么容易

# BIOS

- 在启动过程方面，没有像X86机器那样的标准化BIOS或固件
- 每个ARM SoC都有自己的ROM代码，实现了SoC特定的启动机制
- 因此，启动过程的早期阶段是针对每个SoC的
- 一般来说：能够从非易失性存储（NAND、MMC、USB）向处理器内部的SRAM加载少量代码（外部DRAM尚未被初始化）
- 通常还提供一种恢复方法，以解除平台的故障（USB、串行或有时以太网）
- 用于将第一级引导程序加载到SRAM中，它本身将初始化DRAM并将第二级加载/运行到DRAM中

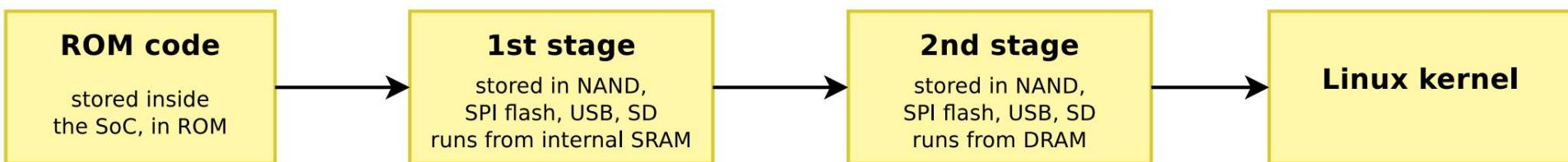
# Bootloaders(1)

- Grub(2)通常不在ARM平台上广泛使用
- U-Boot, 事实上的标准, 在大多数开发板和社区平台上使用。
- Barebox, 有意思但是使用不太广泛。
- 自制的引导程序, 特别是当涉及到安全/DRM时  
(电话、机顶盒等)
- Grub开始获得一些吸引力, 特别是在ARM64上,  
用于服务器市场
- RaspberryPi是一个非常特殊的情况, 一些固件在  
GPU上执行, 并直接加载Linux内核

# Bootloaders(2)

- 第一阶段的引导器由以下两个方面提供：
  - 一个单独的项目。例如。用于Atmel平台的 AT91Bootstrap
  - U-Boot/Barebox本身。SPL的概念：适合第一级约束的最小版本的bootloader
- 与引导程序的互动通常是通过串口进行的
  - U-Boot和Barebox提供了一个shell，有bootloader特定的命令
  - 有时屏幕/键盘的交互是可能的，但不是常规
  - 没有串口的嵌入式是很奇怪的！

# Booting process diagram



# 描述硬件

- 在X86上，大多数硬件可以在运行时动态地被发现
  - PCI和USB提供动态枚举功能
  - 对于其他的硬件，ACPI提供了描述硬件的表格
  - 由于这一点，内核不需要事先知道它将在哪些硬件上运行。
- 在ARM上，在硬件级别上不存在这种机制。
  - 在过去（2011年之前），内核代码本身包含了它必须支持的所有HW平台的描述。
  - 在~ 2011年，ARM内核开发人员转向了不同的HW描述解决方案。设备树
  - 与称为多平台ARM内核的工作一起完成的

# 设备树

- 描述硬件的节点树
- 提供诸如寄存器地址、中断线、DMA通道、硬件类型等信息。
- 由固件提供给操作系统
- 与操作系统无关，不针对Linux
- 可以被引导程序、BSD等使用。
- 起源于PowerPC世界，在那里已经使用了很多年了
- 由开发者编写的源格式（dts），被编译成操作系统理解的二进制格式（dtb）。
  - 每个硬件平台有一个.dts

# Device Tree example

## sun5i.dtsi

```
/ {
 cpus {
 cpu0: cpu@0 {
 device_type = "cpu";
 compatible = "arm,cortex-a8";
 reg = <0x0>;
 };
 };

 soc@01c00000 {
 compatible = "simple-bus";
 ranges;

 uart1: serial@01c28400 {
 compatible = "snps,dw-apb-uart";
 reg = <0x01c28400 0x400>;
 interrupts = <2>;
 clocks = <&apb1_gates 17>;
 status = "disabled";
 };

 uart3: serial@01c28c00 {
 compatible = "snps,dw-apb-uart";
 reg = <0x01c28c00 0x400>;
 interrupts = <4>;
 clocks = <&apb1_gates 19>;
 status = "disabled";
 };
 [...]
 };
};
```

## sun5i-r8-chip.dts

```
/ {
 model = "NextThing C.H.I.P.";
 compatible = "nextthing,chip", "allwinner,sun5i-r8",
 "allwinner,sun5i-a13";

 leds {
 compatible = "gpio-leds";

 status {
 label = "chip:white:status";
 gpios = <&xp_gpio 2 GPIO_ACTIVE_HIGH>;
 default-state = "on";
 };
 };
 [...]

 &uart1 {
 pinctrl-names = "default";
 pinctrl-0 = <&uart1_pins_b>;
 status = "okay";
 };
};
```

# 设备树

- 几乎用于Linux中的所有ARM以及ARM64平台
- 在引导程序（如 U-Boot 或 Barebox）也会使用
- 设备树源代码存储在Linux内核源码中
  - 根据需要在 U-Boot/Barebox 源代码中进行复制
  - 原先有过计划建立一个中央存储库（现在也没有）
    - *arch/arm/boot/dts/*
- 设计是与操作系统无关的，向后兼容
  - 在实践中，经常被改变以适应Linux内核的变化
- 与Linux内核镜像一起，由引导程序加载到内存中。
- 在启动时由Linux内核解析，以了解哪些硬件是可用的

# Linux Kernel (分工合作)

- 对ARM核心的支持通常由ARM工程师完成
  - MMU, caches, virtualization等.
  - arch/arm, arch/arm64
- 仅仅支持核心是不够的，还需要支持ARM SoC 以及硬件平台
  - 每个硬件的模块都需要相应的驱动，包括SoC内部的硬件以及板上的硬件，位于drivers 目录中
  - 需要设备树arch/arm(64)/boot/dts
  - 板子多样性，有的由主线支持，有的由设备商自己支持

謝謝！



# 输入输出系统

2022年秋

# 期末相关安排

---

- 大实验检查（12月5日-8日）
  - 提供线上和线下检查方式，同学们灵活选择
  - 检查时间预约表格需关注网络学堂通知
- 实验测试（12月9日，随堂）
  - 线上（git + THINPAD-Cloud）
- 分组答辩（12月16日）
  - 预计分为8组，线上方式，具体安排需关注网络学堂通知
  - 需要提前准备PPT
- 期末考试时间
  - 预计时间12月31日，具体时间等待教务最终通知

# 教学内容安排

---

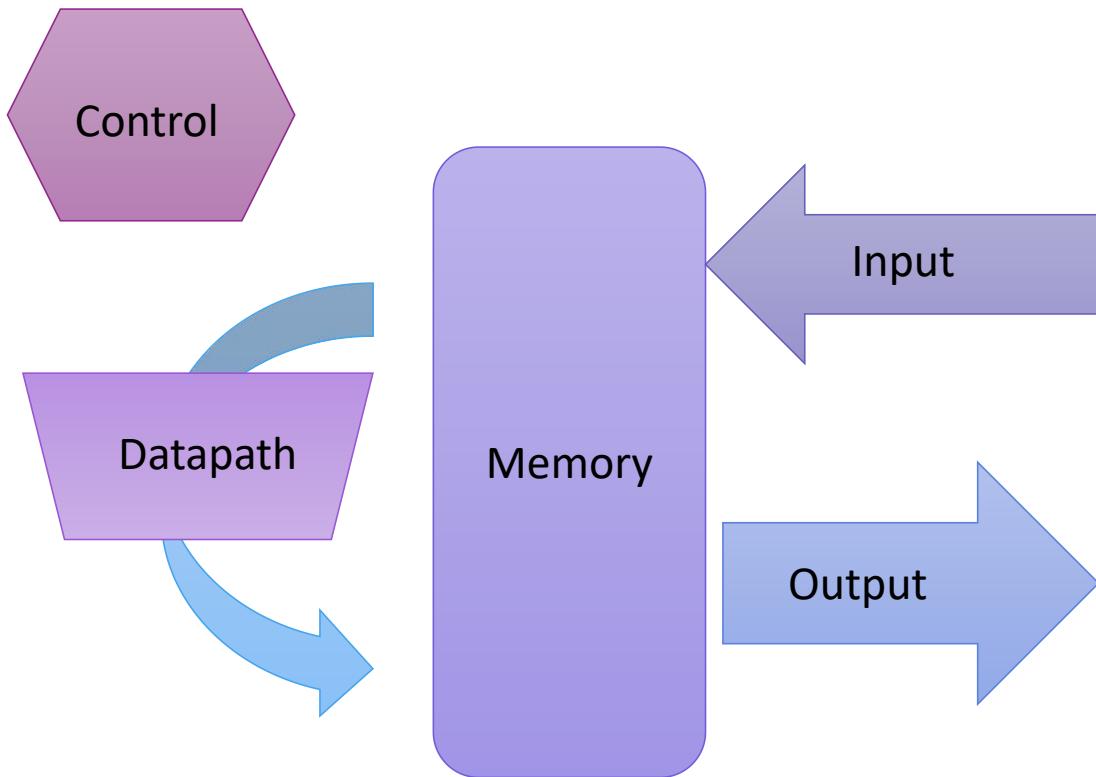
- 第一讲：输入/输出系统概述和输入/输出方式
- 第二讲：总线
- 大实验答辩
- 第三讲：接口电路和外部设备
- 课程总结
- 分组答辩
- 期末考试

# 主要教学内容

---

- 输入输出系统的作用、功能及与其他系统的关系
- 输入/输出系统组成
- 要解决的问题
- 输入/输出方式
  - 程序直接控制
  - 中断
  - DMA
  - 通道
  - 外围处理机

# 计算机运行机制

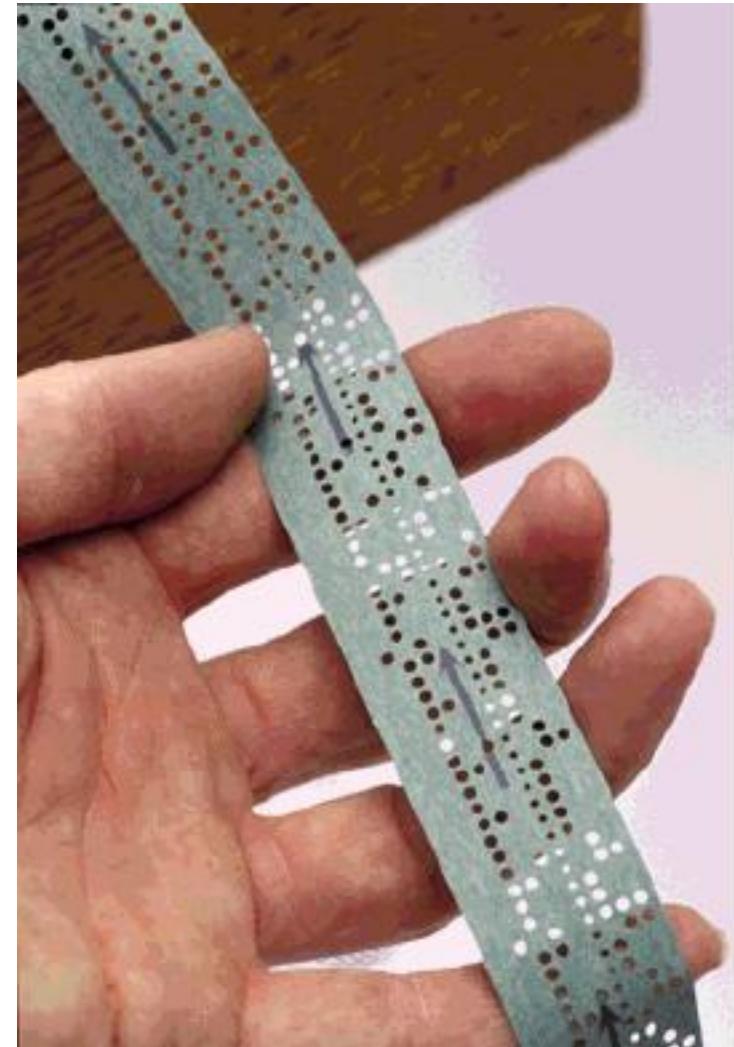


- Datapath: 完成算术和逻辑运算，通常包括其中的寄存器。
- Control: CPU的组成部分，它根据程序指令来指挥 datapath, memory以及I/O 运行，共同完成程序功能。
- Memory: 存放运行时程序及其所需要的数据的场所。
- Input: 信息进入计算机的设备，如键盘、鼠标等。
- Output: 将计算结果展示给用户的设备，如显示器、打印机、喇叭等。

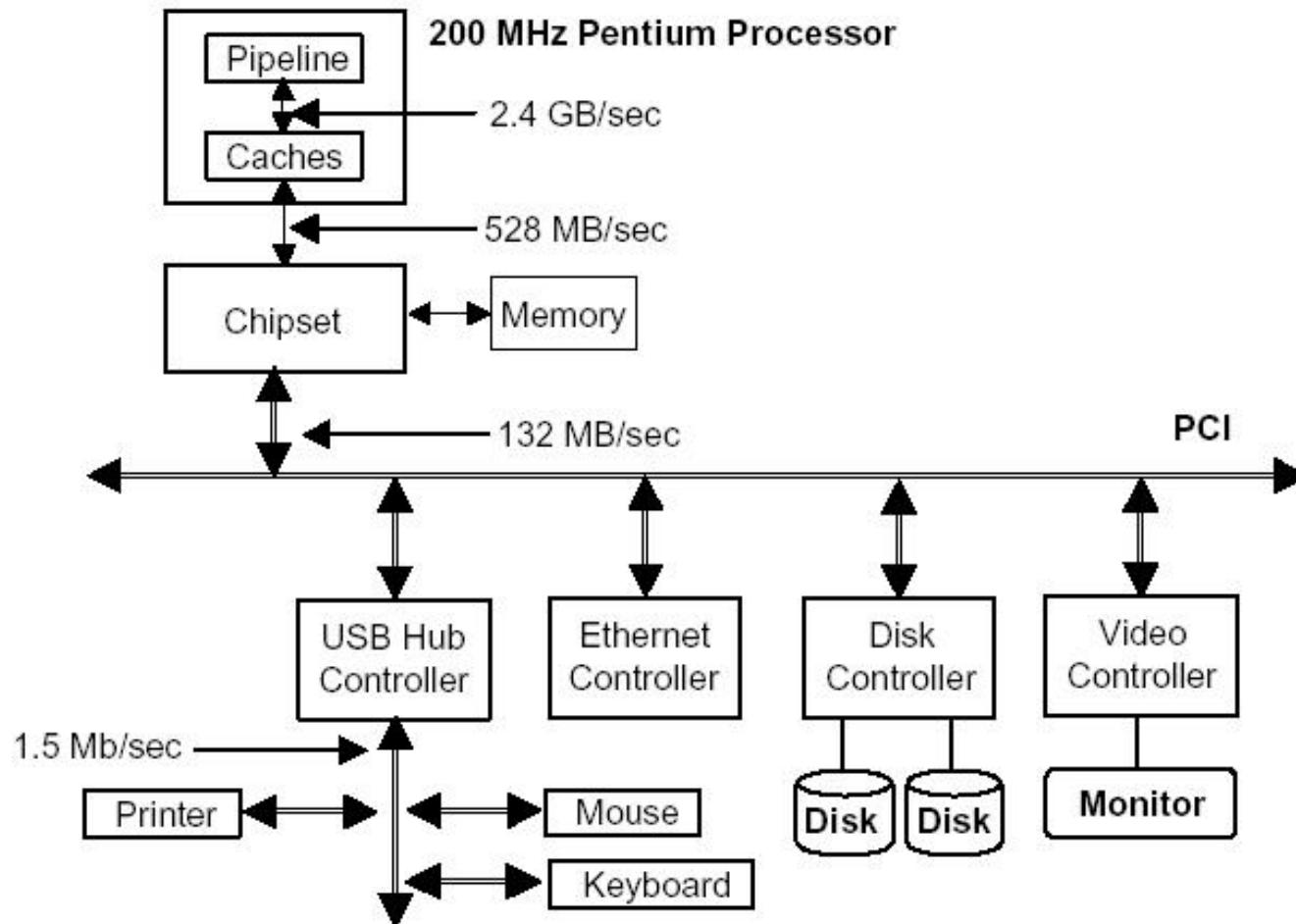
# 作用和功能

## □ 与计算机外部交换信息的通道

- 早期
  - 穿孔机、纸带
- 中期
  - 键盘、显示器、打印机、鼠标
- 现在
  - 语音、图象、图形等多媒体数据（扫描仪、DC）
  - 计算机网络
- 将来
  - 无所不在的计算、普适计算
  - 人机交互、脑机交互



# 个人计算机的组成



# I/O设备

| <b>Device</b>    | <b>Behavior</b> | <b>Partner</b> | <b>Data rate (KB/sec)</b> |
|------------------|-----------------|----------------|---------------------------|
| Keyboard         | input           | human          | 0.01                      |
| Mouse            | input           | human          | 0.02                      |
| Voice input      | input           | human          | 0.02                      |
| Scanner          | input           | human          | 400.00                    |
| Voice output     | output          | human          | 0.60                      |
| Line printer     | output          | human          | 1.00                      |
| Laser printer    | output          | human          | 200.00                    |
| Graphics display | output          | human          | 60,000.00                 |
| Modem            | input or output | machine        | 2.00-8.00                 |
| Network/LAN      | input or output | machine        | 500.00-6000.00            |
| Floppy disk      | storage         | machine        | 100.00                    |
| Optical disk     | storage         | machine        | 1000.00                   |
| Magnetic tape    | storage         | machine        | 2000.00                   |
| Magnetic disk    | storage         | machine        | 2000.00-10,000.00         |

# I/O设备

## 口 繁多的输入/输出设备

- 功能多样
  - 满足各种要求
- 服务对象不同
  - 人、计算机、其他设备
- 数据传输率差别很大
  - 键盘、鼠标
  - 显示器、网卡

口 多：种类繁多

口 杂：功能繁杂

口 异：速度不一

如何管理外  
部设备

# 要解决的问题

---

## □ 控制方式

- CPU如何控制输入/输出？（输入/输出方式）

## □ 传输方式

- 传输通道、方式、速率等（总线、接口）

## □ 数据识别和转换

- 数/模转换、语音识别等，转换为字符、数据等计算机能识别的格式（设备）

# 输入输出方式

---

## □ 程序直接控制

- CPU直接使用输入/输出指令来控制外部设备

## □ 程序中断

- 外部设备请求，CPU响应，CPU与外设并行工作

## □ 直接存储访问（DMA）

- 专用输入/输出控制器

## □ 通道

## □ 外围处理机

# 程序直接控制

```
READ_SERIAL:
 li t0, COM1
.TESTR:
 lb t1, %lo(COM_LSR_OFFSET)(t0)
 andi t1, t1, COM_LSR_DR // 截取读状态位
 bne t1, zero, .RSERIAL // 状态位非零可读进入读
 j .TESTR // 检测验证
.RSERIAL:
 lb a0, %lo(COM_RBR_OFFSET)(t0)
 jr ra
```

// 读串口：将读到的数据写入a0低八位

CPU方：

查询接口状态（循环等待）

直到接口已经接收到该字符

读字符

外设方：

往接口数据缓冲中送字符

处理完后，置状态寄存器

等待下一个字符

# 程序直接控制方式特点

---

- 成本低
- 效率低
- 严重占用CPU资源
- 适用情况
  - 早期计算机中高速设备

# 程序中断方式

---

- CPU和外部设备同时工作
  - 外部设备发起请求
  - CPU暂停正在执行的程序，进行响应
  - 处理完成后，继续执行原来的程序
- 提高CPU的效率
- 可以同时管理多个外部设备

# 中断的一些概念

## □ 中断源

- 外中断：I/O设备等
- 异常(内中断)：处理器硬件故障、程序“出错”，Trap
- 中断触发器
- 中断状态寄存器

## □ 中断优先级

- 响应中断的顺序

## □ 禁止中断与中断屏蔽

- 中断允许触发器（EI、DI）
- 有选择封锁

# 中断的完整过程

## □ 中断请求

- 中断源设备设置中断触发器
  - 每个中断源有1个中断触发器
  - 同时可设置1个中断屏蔽触发器

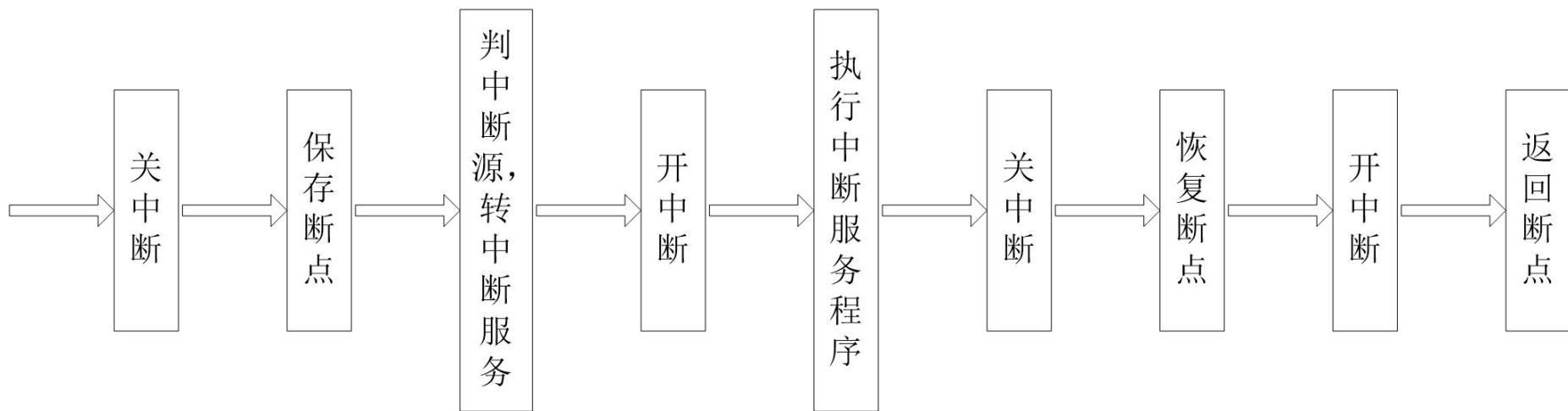
## □ 中断响应

- 响应条件
  - 允许中断、当前指令结束、优先级
- 响应实现
  - 硬件实现的中断隐指令，保存断点

## □ 中断处理

- 保存现场信息
- 运行中断服务程序
- 中断返回

# 中断处理过程

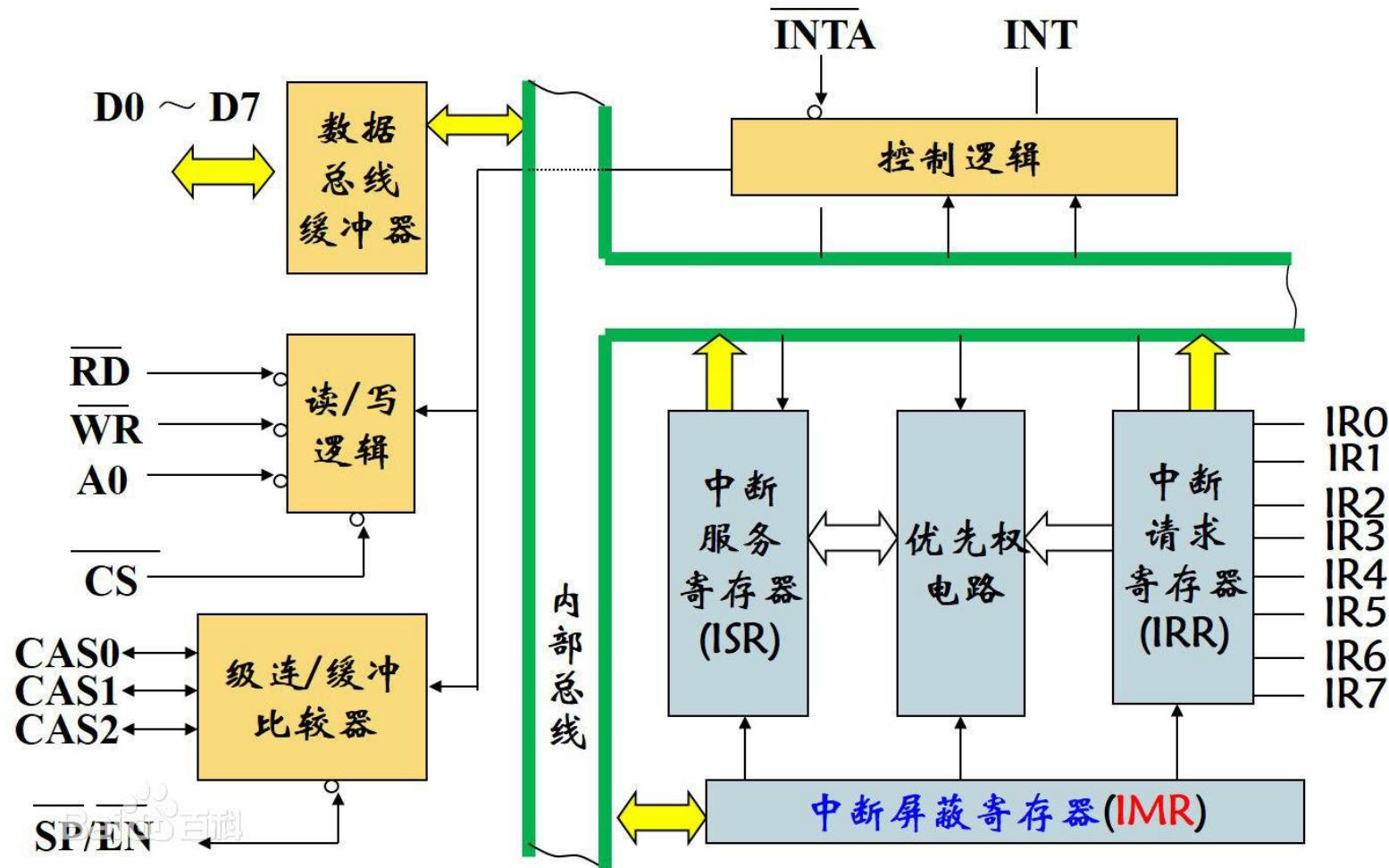


# 中断设备接口组成

---

- 中断请求寄存器
- 中断屏蔽寄存器
- 优先级排队线路
- 数据缓冲寄存器
- 中断控制和工作状态逻辑
- 设备选择器
- 中断向量表

# 8259A中断控制器



# 程序中断方式应用场景

---

- CPU与外部设备并行工作
- 硬件故障处理
- 人机交互
- 多道程序和分时操作
- 实时处理（监控）
- 应用程序和操作系统之间的联系
- 多处理机中各处理机之间联系

# 中断控制方式特点

---

## □ 适用情况

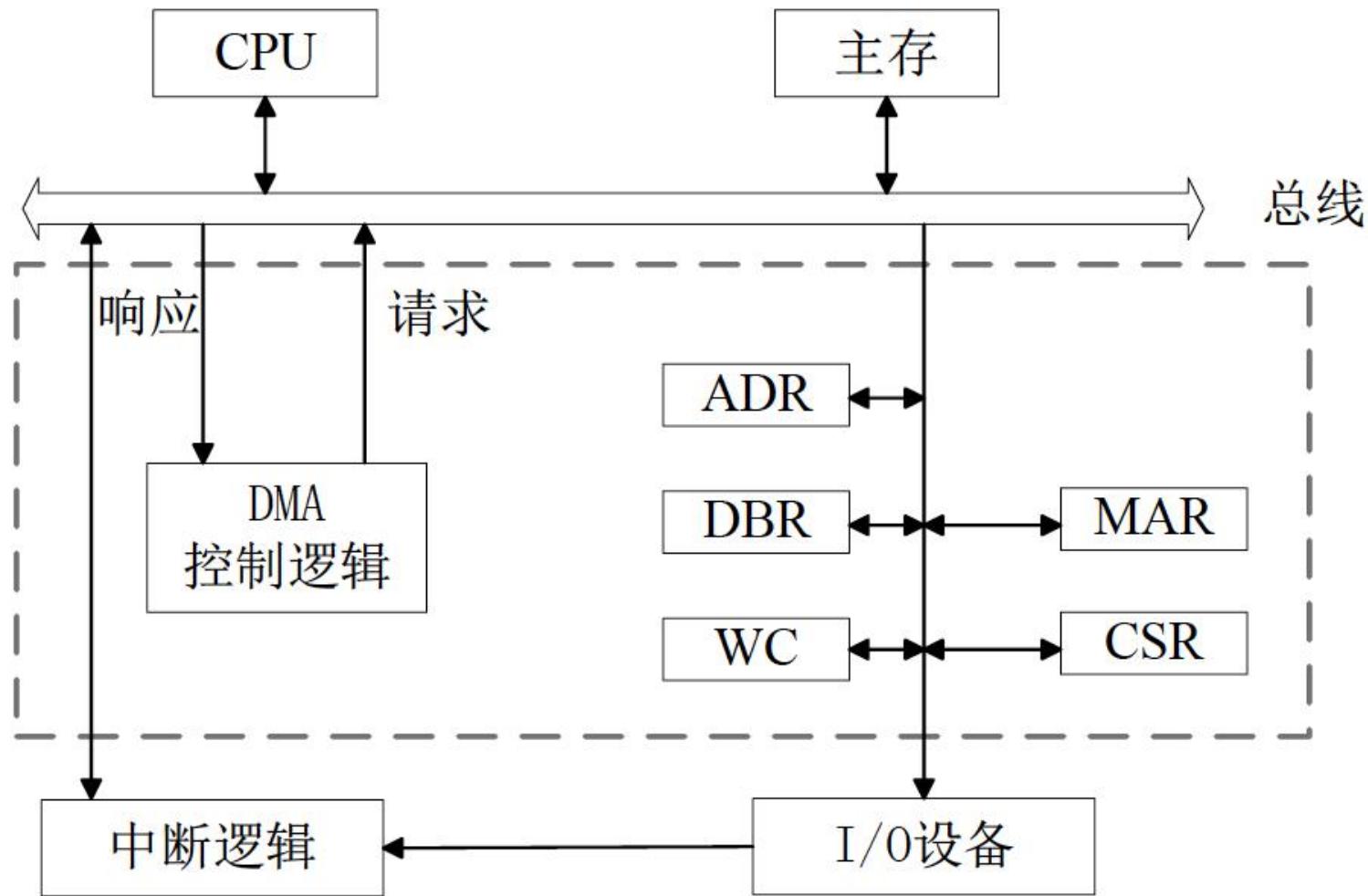
- 传输速度不高
- 传输量不大

## □ 对CPU干扰较大

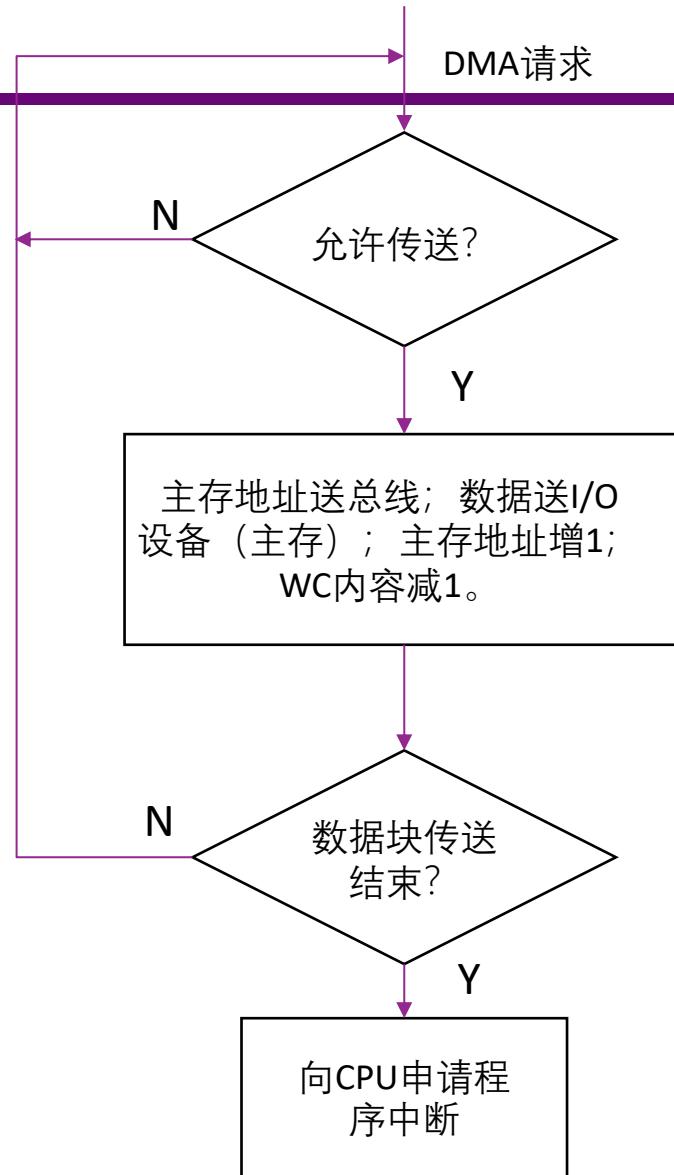
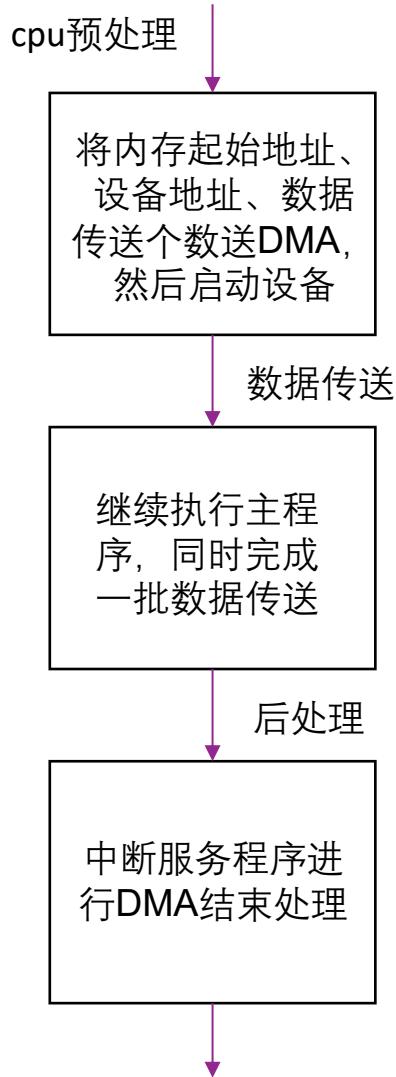
# 直接存储访问 (DMA)

- I/O设备和主存储器之间的直接数据通路，为专设的硬件，用于高速I/O设备和主存储器之间成组传送数据。
- 数据传输过程由DMA自行控制
- 主存储器需要支持成组传送
- 数据传送开始前和结束后通过程序或中断方式对DMA进行预处理和后处理
- DMA工作方式
  - 独占总线方式
  - 周期窃取方式

# DMA控制器组成



# DMA数据传送过程



# DMA方式的问题

## □ 虚拟地址和实地址

- DMA采用实地址：虚拟地址连续，但实地址不连续
- 采用虚拟地址：DMA进行虚实地址转换

## □ Cache一致性

- 主存中的数据可能不是最新的
- 采用直接写会带来性能的降低
- DMA查询Cache，降低性能
- 直接设计硬件控制

# DMA方式的特点

---

## □ 与设备一对一服务

- 多DMA控制器同时工作可能发生冲突

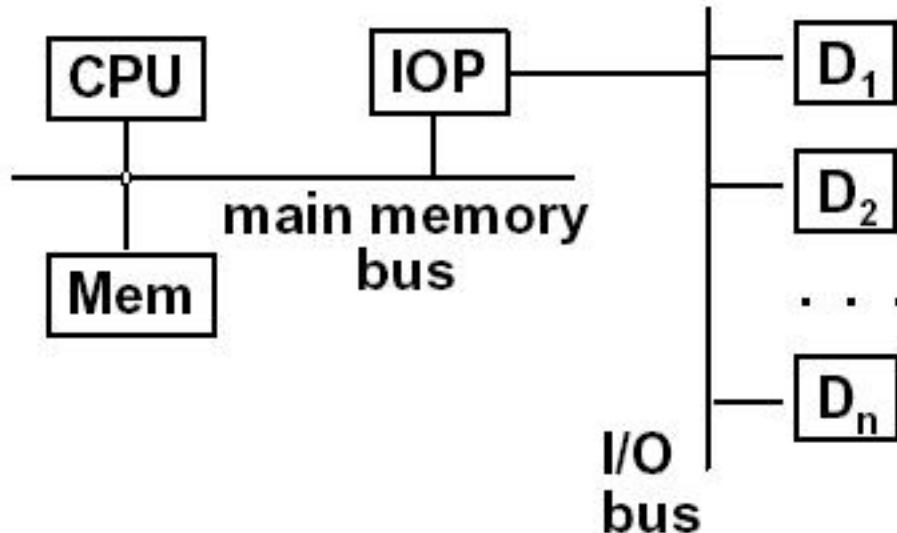
## □ 对CPU打扰适中

- 初始化
- 周期挪用

## □ 无法适用大量高速设备的管理

# 通道控制方式

- I/O通道是计算机系统中代替CPU管理控制外设的独立部件，是一种能执行有限I/O指令集合——通道命令的I/O处理机。
- 一对多的连接关系
- 适应不同速度、不同种类的外部设备，可并行工作



# 通道的功能

---

- 根据CPU要求选择某一指定外设与系统相连，向该外设发出操作命令，进行初始化
- 指出外设读/写信息的位置以及与外设交换信息的主存缓冲区地址
- 控制外设与主存之间的数据交换
- 指定数据传送结束时的操作内容，检查外设的状态

# 通道的类型

---

## □ 字节多路通道

- 简单的共享通道，分时处理，面向低、中速字符设备

## □ 选择通道

- 选择一台外设独占整个通道，以成组传送方式传送数据块，效率高，适合快速设备

## □ 数组多路通道

- 上两种方式的结合，效率高，控制复杂

# 外围处理机

---

## □ 通道型处理机

- 共享内存

## □ 外围处理机

- 通用计算机
- 独立完成输入/输出功能
- 通过通道方式与主机进行交互

# 设计输入/输出系统

## □ 性能

- 考虑吞吐量和延迟
- 适应各种不同类别的设备的性能的差异
- 从操作系统、驱动程序等各方面综合考虑
- 考虑到设备性能的提高

## □ 可扩展性

- 允许更多的设备接入到输入/输出系统

## □ 可适应性

- 设备有无
- 设备故障

# 输入/输出系统

---

- 输入/输出设备多，功能复杂，速度不一
- 多种控制方式，解决速度不一的问题，尽量少地占用CPU资源
- 操作系统管理
- 硬件直接支持
- 与不同的设备有直接的依赖关系（驱动程序）
- 尽量使设备使用统一的标准——虚拟设备

# 阅读和思考

---

## □ 阅读

- 教材相关章节

## □ 思考

- 输入/输出方式解决了什么问题？
- 它们各自有哪些特点？

## □ 实践

- 完成实验报告
- 完成第三单元作业

---

谢谢

# 计算机组成原理



## 总线

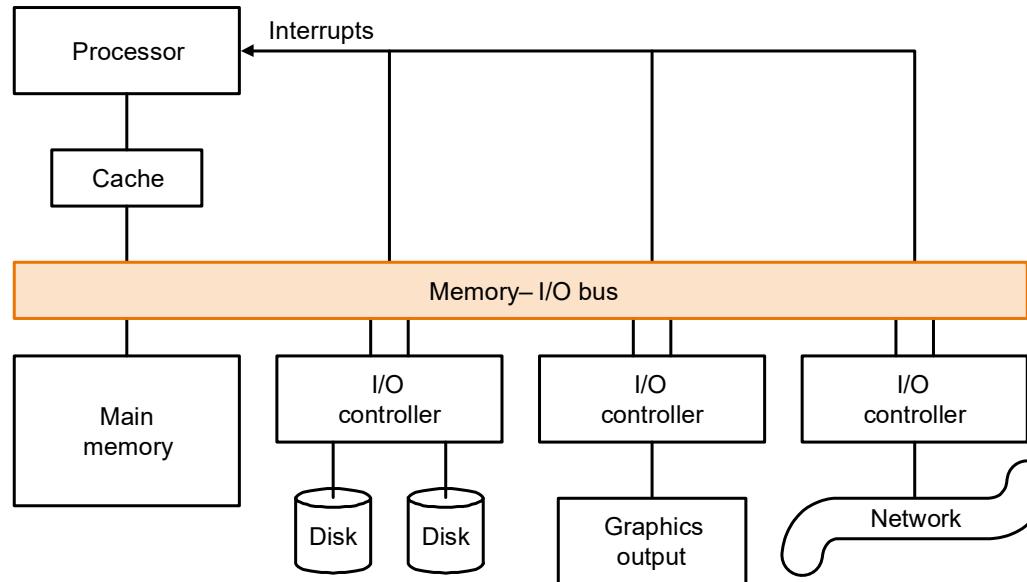
2022年秋

# 主要教学内容

---

- 总线概念
- 总线分类
- 总线组成
- 设计总线的关键问题
  - 总线仲裁
  - 通信方式
- 总线举例

# 处理器和其他组成部分的接口



- 输入/输出系统设计受到多方面因素的影响（可扩展性，可恢复性等）
- 性能：
  - 访问延迟
  - 吞吐量
  - 设备和系统的连接关系
  - 层次存储系统
  - 操作系统
- 用户和应用也各不相同

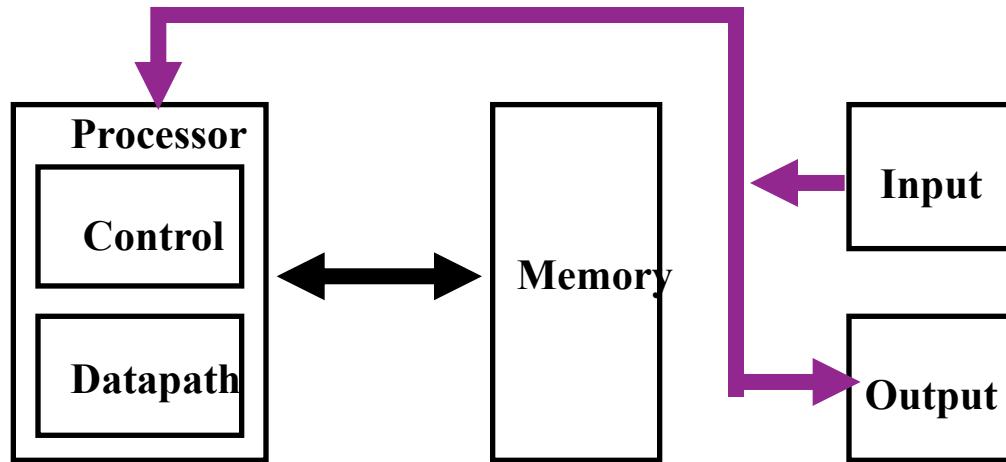
# 什么是 bus?



- 公共汽车：一种大众交通工具
- 一组导线
- 共同点：多个使用者共享通道

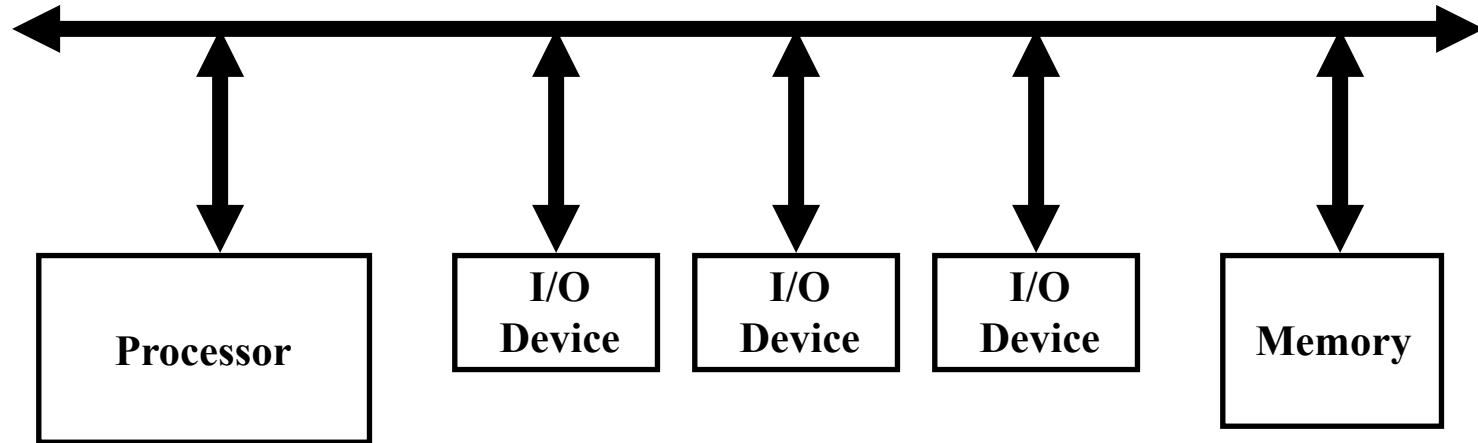
# 计算机总线

- 共享的信息通道
- 用于连接计算机多个子系统（部件）



- 总线也是连接复杂巨系统的一种基本工具
  - 功能抽象

# 使用总线的好处



□ 解决外部设备“杂”的问题：

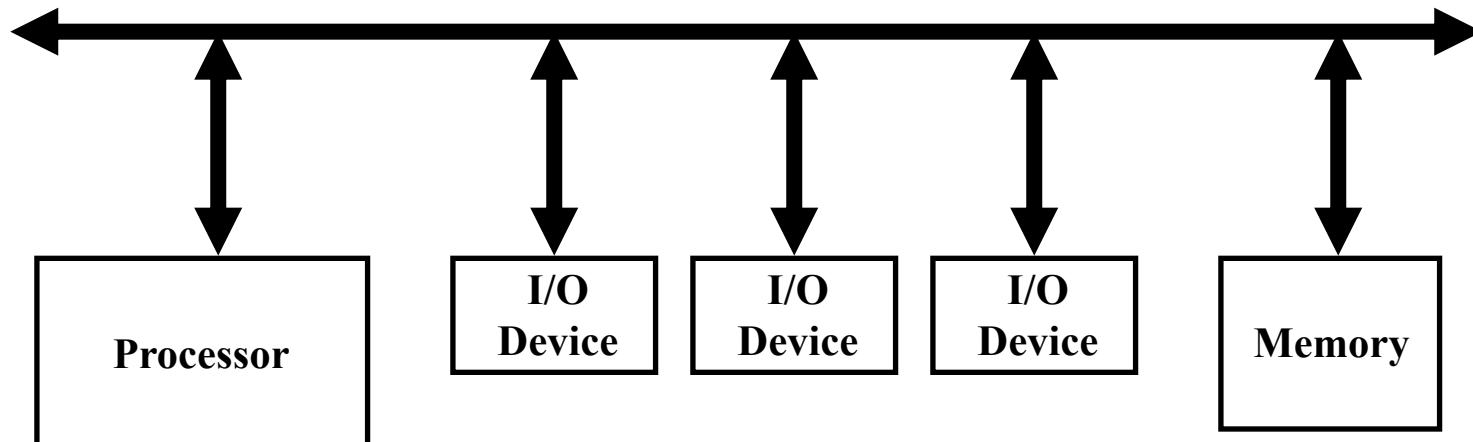
- 容易增添新的设备
- 使用相同总线标准的外设容易在不同计算机间兼容

□ 降低成本：

- 总线可供多个设备共享

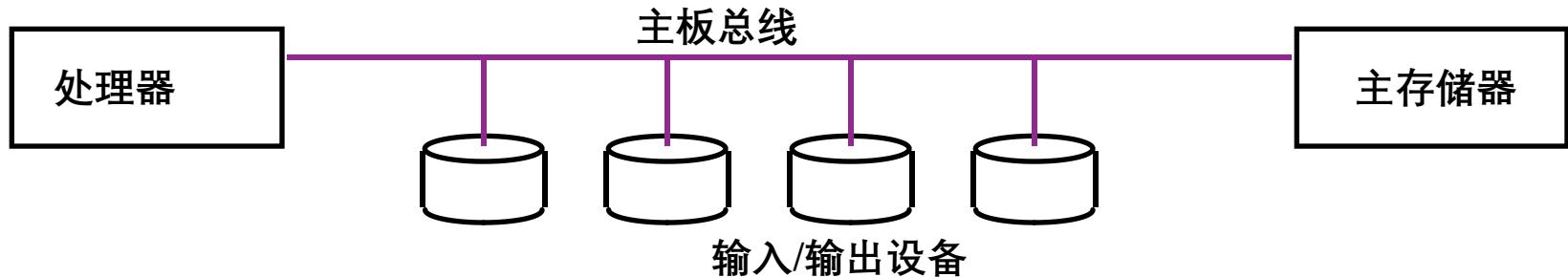
□ 简化设计

# 总线的不足



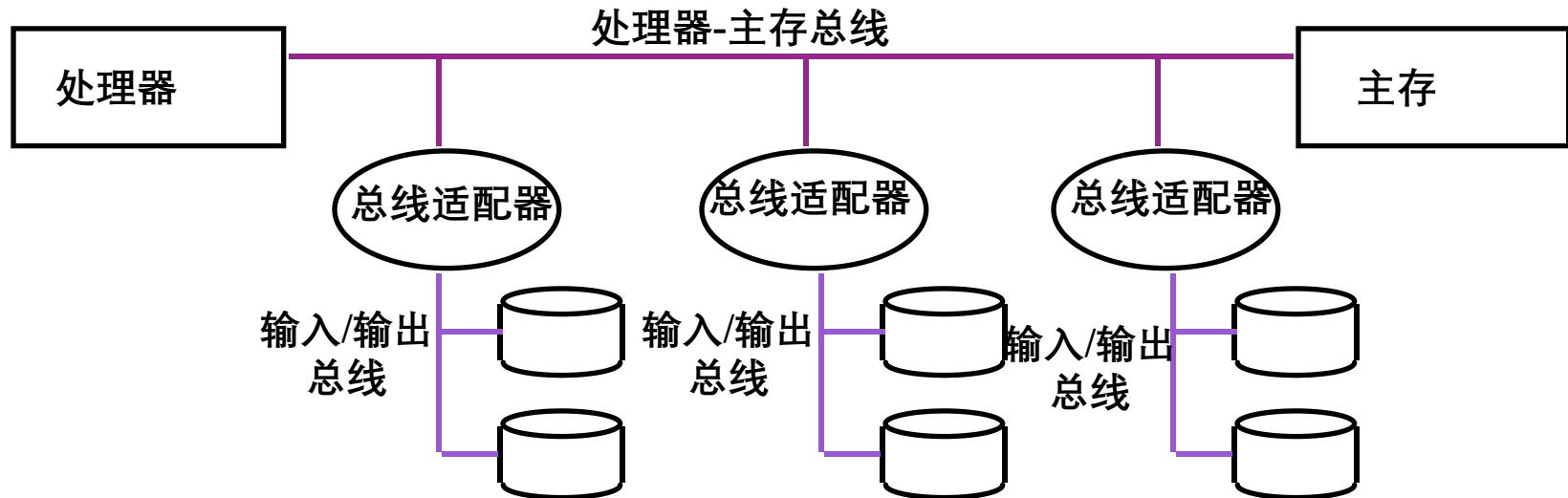
- 容易成为信息通道的瓶颈
  - 总线带宽限制了整条总线的吞吐量
- 总线的最高速度主要由下列因素决定：
  - 总线长度
  - 总线负载的设备数
  - 负载设备的特性
    - 延迟是否差异较大？
    - 数据传输率差异较大？

# 单总线计算机：主板总线



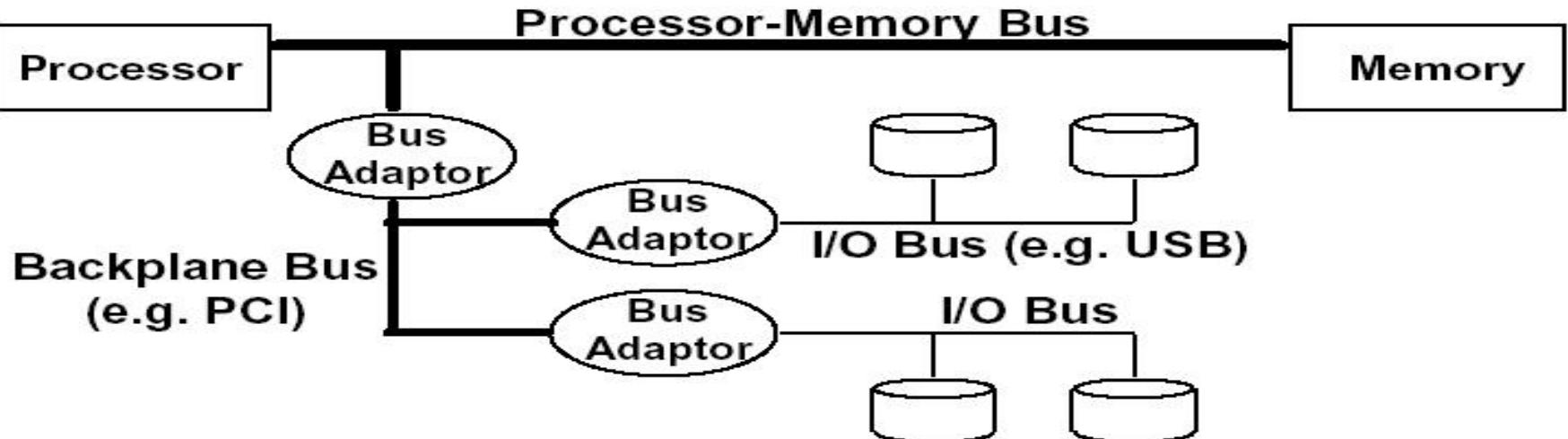
- 使用一条总线：
  - 处理器和主存储器之间通信
  - 主存储器和输入/输出设备之间通信
- 优点：简单、成本低
- 缺点：速度慢，总线将成为系统瓶颈
- 应用：IBM PC – ISA EISA、PDP-1

# 双总线系统



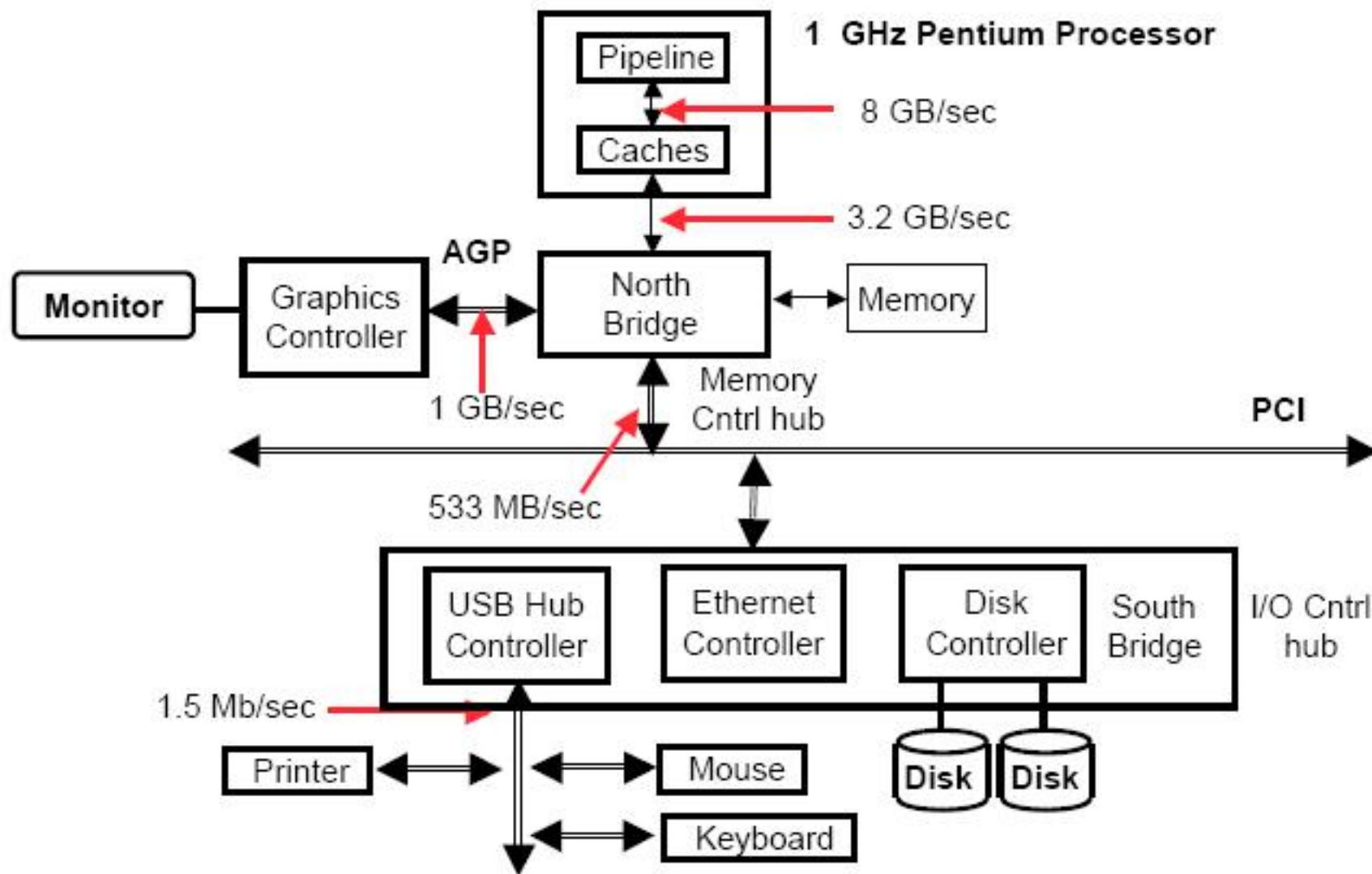
- 输入/输出总线通过适配器和处理器-主存总线相连：
  - 处理器-主存总线：主要用于处理器和主存储器之间的通信
  - 输入/输出总线：为输入/输出设备提供信息
- 应用举例：
  - Apple Macintosh II
    - NuBus：处理器、主存和选定的少量I/O设备
    - SCSI总线：其余I/O设备

# 三总线系统

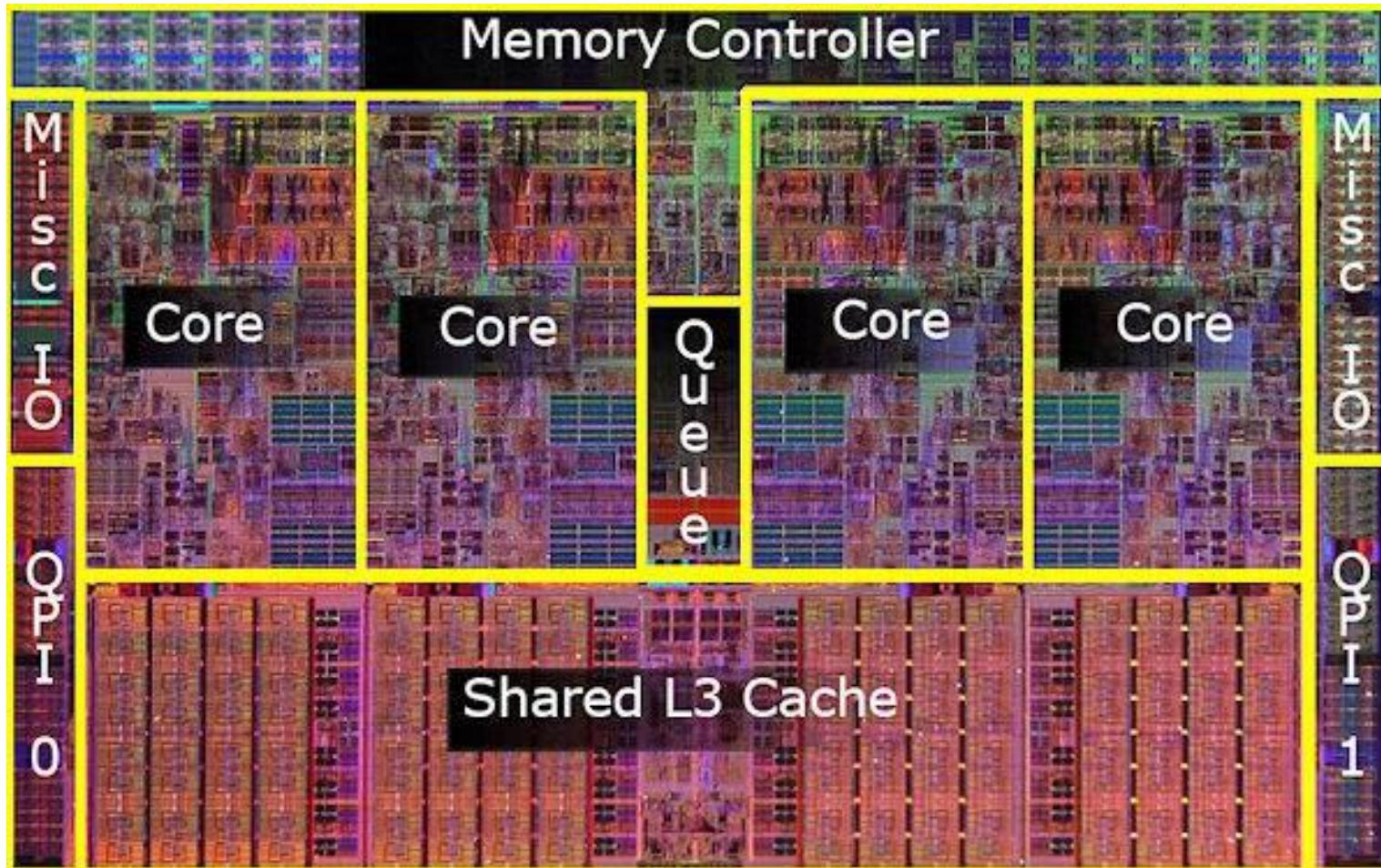


- 主板总线连接到处理器-主存总线
  - 处理器-主存总线主要用于处理器和主存之间数据交换
  - I/O总线连接到主板总线
- 优点
  - 大大减少处理器-主存总线负载
- 例：现代PC基本采用的结构

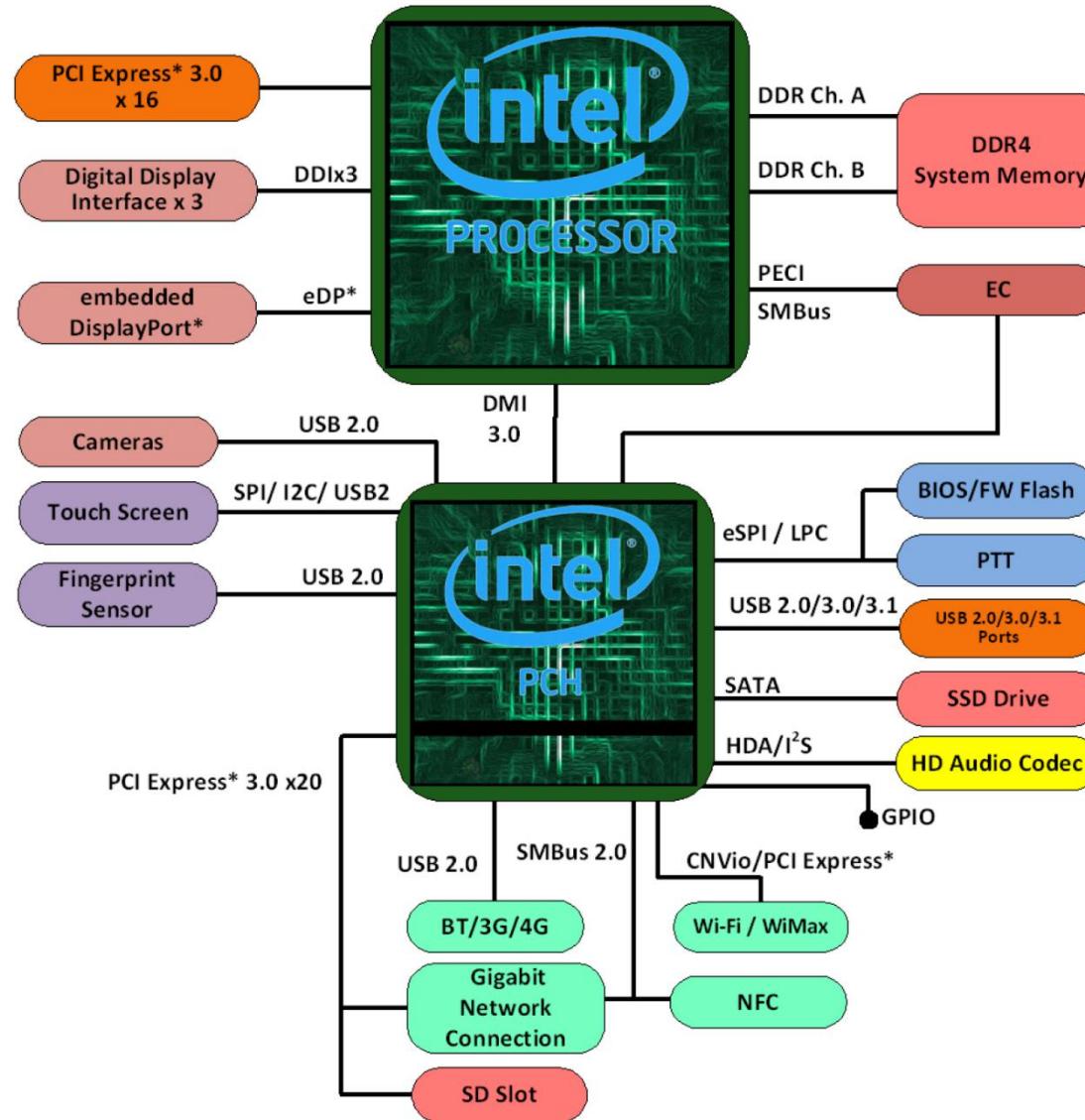
# 传统的x86计算机的总线



# 新的i7处理器



# 新x86系统的总线连接



# 总线类型

## □ 处理器-主存总线（专用）

- 传输距离短、速度高
- 主存储器专用
  - 保证主存储器-处理器之间的高带宽
- 直接和处理器连接
- 优化处理使之适应Cache块传送

## □ 输入/输出总线（行业标准）

- 通常距离较长，速度较慢
- 需要适应多种输入/输出设备
- 和处理器-主存总线通过桥连接（或通过主板总线）

## □ 主板总线（行业标准或专门设计）

- 主板：连接各部件器件的底盘
- 应允许处理器、主存储器和输入/输出设备互连
- 应有价格优势：所有组件连接在一条总线上

# 总线的一般组成



## □ 控制线：

- 总线请求信号及数据接收信号
- 指明数据线上传输信息的类型

## □ 数据线 在源设备和目标设备间传送信息

- 数据和地址
- 复杂的命令

# 总线标准

- 设备用于人机交互
  - 总线定义了交互的通信协议/标准：
    - PCI
    - EISA
    - SCSI
    - USB
    - Bluetooth
    - ...
- 标准十分重要：
- 不同公司设计的外部设备，应该能在同一计算机上安装使用。
  - 不同公司的计算机也应该可以使用某一外部设备。
  - 外部设备的通讯速度差异很大
- 标准是抽象的设计
- 标准可以影响性价比，可靠性等

# 总线结构

事务协议

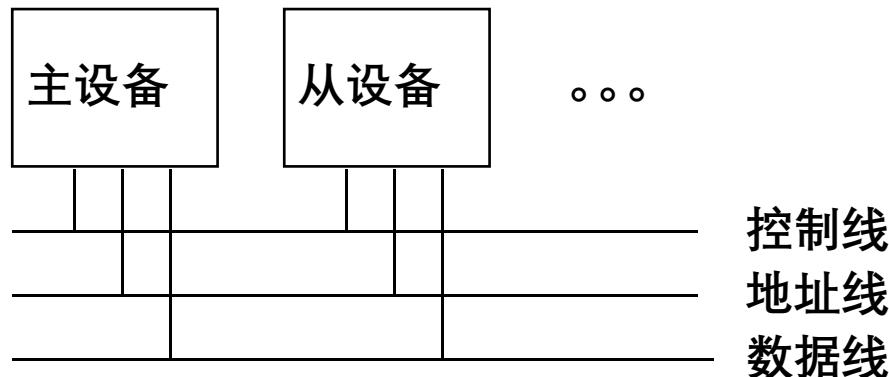
时序和信号规范

一组导线

电气信号规范

接口的物理/机械特性

# 总线相关概念



总线主设备：有能力控制总线，发起总线事务

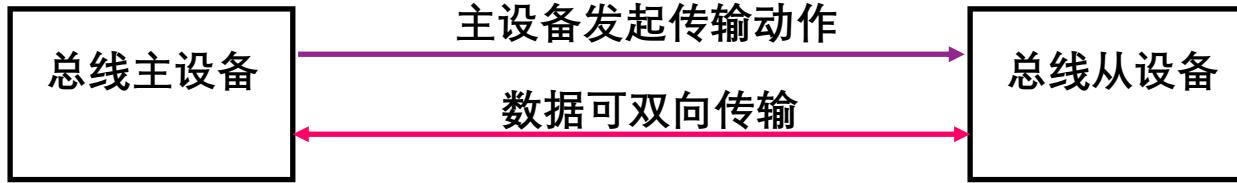
总线从设备：响应主设备请求

总线通信协议：定义总线传输中的事件顺序和时序要求

异步总线传输：控制信号（请求，应答）作为总控信号

同步总线传输：使用共同的时钟信号

# 主设备和从设备



□ 总线事务包括两个部分：

- 发起命令（和地址）
- 传输数据

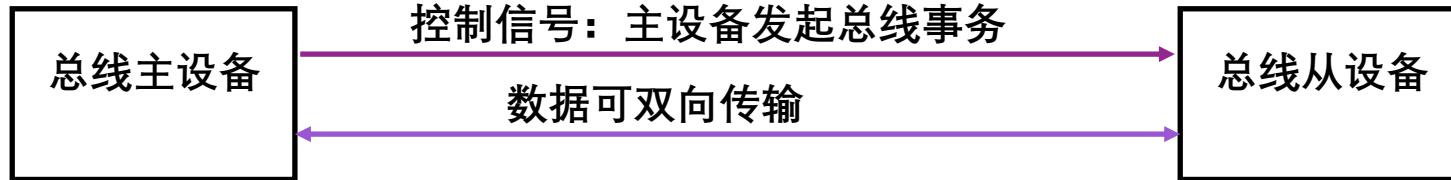
□ 主设备是总线事务的发起者：

- 发出命令（和地址）

□ 从设备是总线事务的响应者：

- 若主设备发出的是读命令，则将数据发送到主设备
- 否则，接收主设备发来的写入数据

# 仲裁：获得总线使用权



## □ 总线设计中重要问题之一：

- 如何为需要使用总线的设备保留总线？

## □ 可通过主—从设备的安排来避免冲突：

- 只允许总线主设备发起总线事务，控制所有总线请求
- 从设备响应主设备的读写请求

## □ 最简单的设计：

- 处理器作为唯一的总线主设备
- 所有总线请求均由处理器控制
- 主要缺点：处理器被卷入到每一个总线事务中

# 多个总线主设备

## □ 总线仲裁的基本要求：

- 某总线主设备使用总线前应发出总线请求
- 只有得到授权后，主设备才能使用总线
- 使用完毕后，主设备应通知仲裁器

## □ 总线仲裁器在以下两方面取得平衡：

- 优先权：优先级高的设备应该得到优先服务
- 公平性：最低优先级的设备也不能永远被排除在总线服务之外

## □ 总线仲裁方式：

- 集中仲裁和分布仲裁
  - 集中仲裁：例如，交通警察在路口指挥交通
  - 分布仲裁：路口没有交通警察，所有车辆先停下，确认其他方向没有来车后通行
- 按优先级仲裁或轮循仲裁
  - 优先级仲裁：例如，救护车在道路上有高优先级

# 总线仲裁方式

---

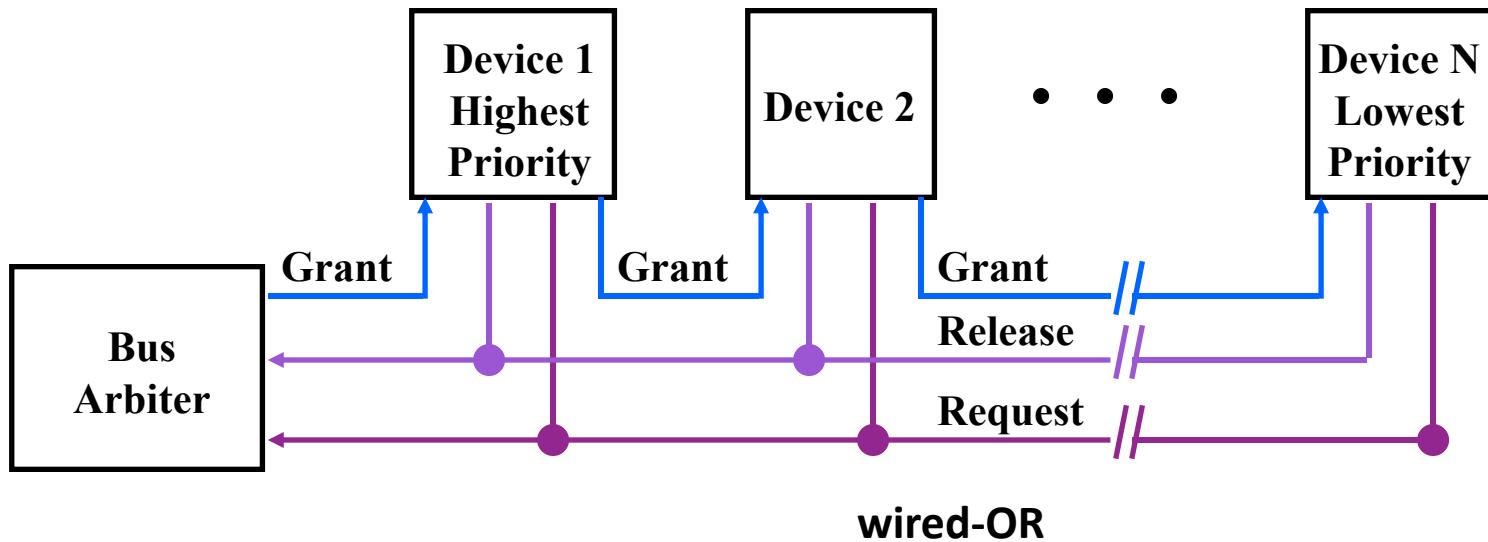
## □ 集中仲裁

- 菊链仲裁
  - 所有设备共用一个总线请求信号
- 集中平行仲裁
  - 通过集中的仲裁器进行

## □ 分布仲裁

- 通过自我选择进行分布式仲裁
  - 每个要使用总线的设备将自己的标识放在总线上
- 碰撞检测
  - 以太网

# 菊链仲裁



□ 优点：简单

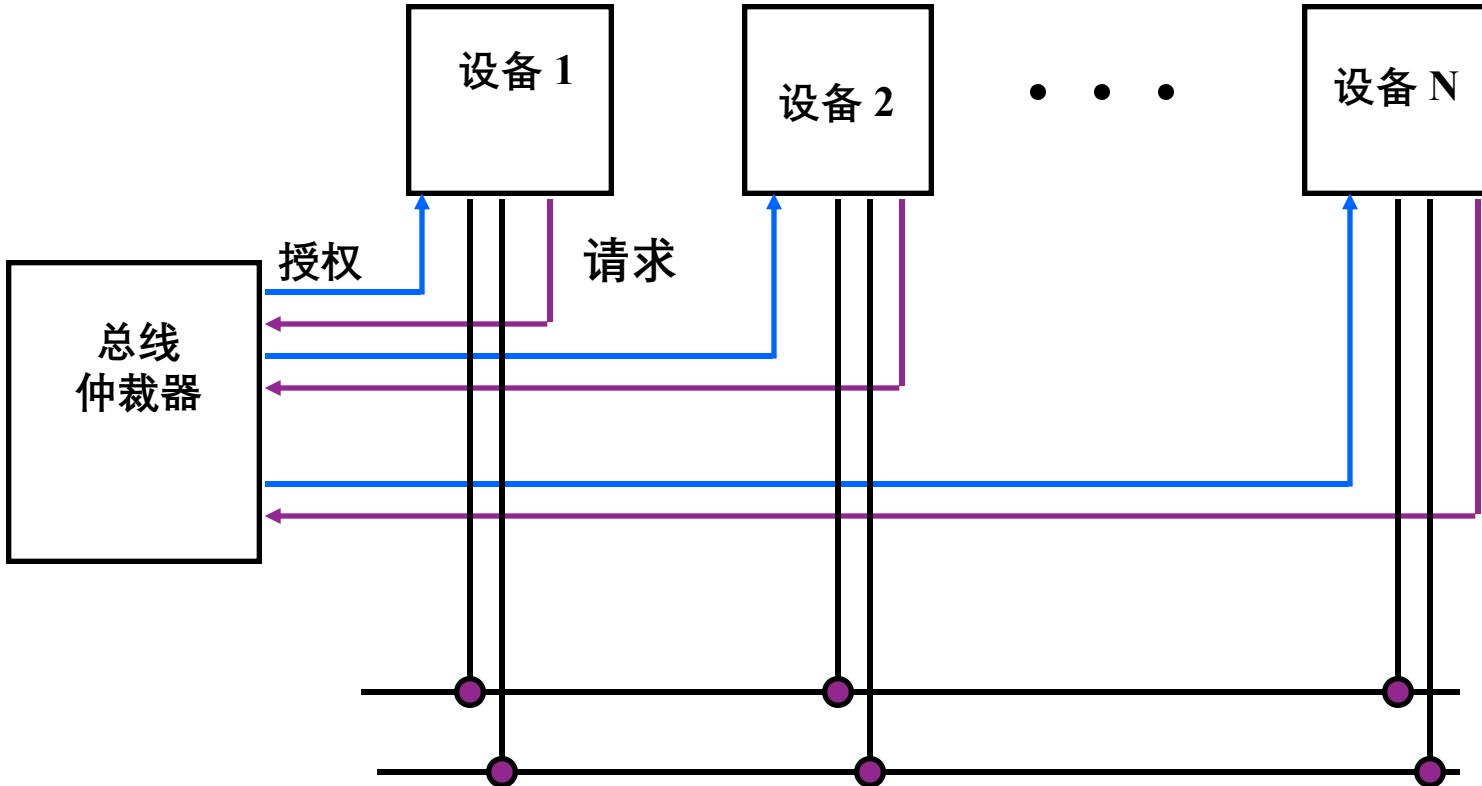
□ 缺点：

- 无法保证公平性

    低优先级设备可能得不到总线使用权

- 总线授权信号的逐级传递限制了总线的速度

# 集中平行仲裁



- 用于几乎所有处理器-主存总线和一些高速输入/输出总线

# 同步和异步总线

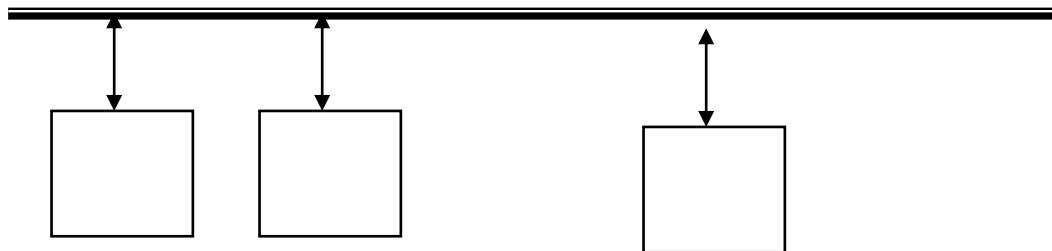
## □ 同步总线：

- 控制线中包含有一根时钟信号线
- 传输协议根据时钟信号制定：
  - 例如：主设备提出总线请求后5个时钟周期，可以获得能否使用总线的信号。
- 优点：逻辑简单、高速
- 缺点：
  - 总线上所有设备必须按时钟频率工作
  - 为防止时钟信号扭曲，高速工作时，总线距离必须足够短

## □ 异步总线：

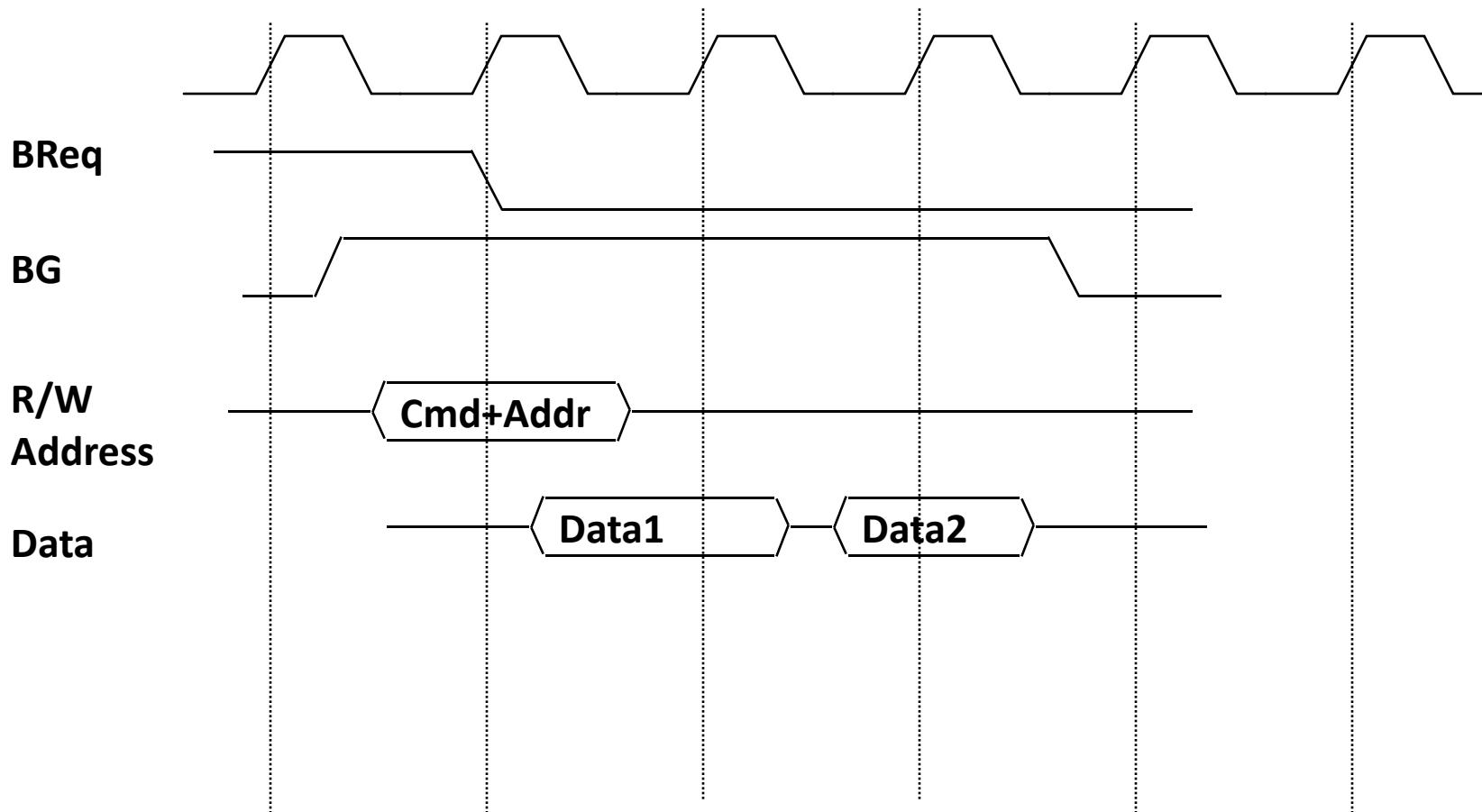
- 不使用统一的时钟
- 可适应设备的不同速度
- 不用担心时钟信号扭曲，距离可较长
- 使用握手协议

# 最简单的总线模式



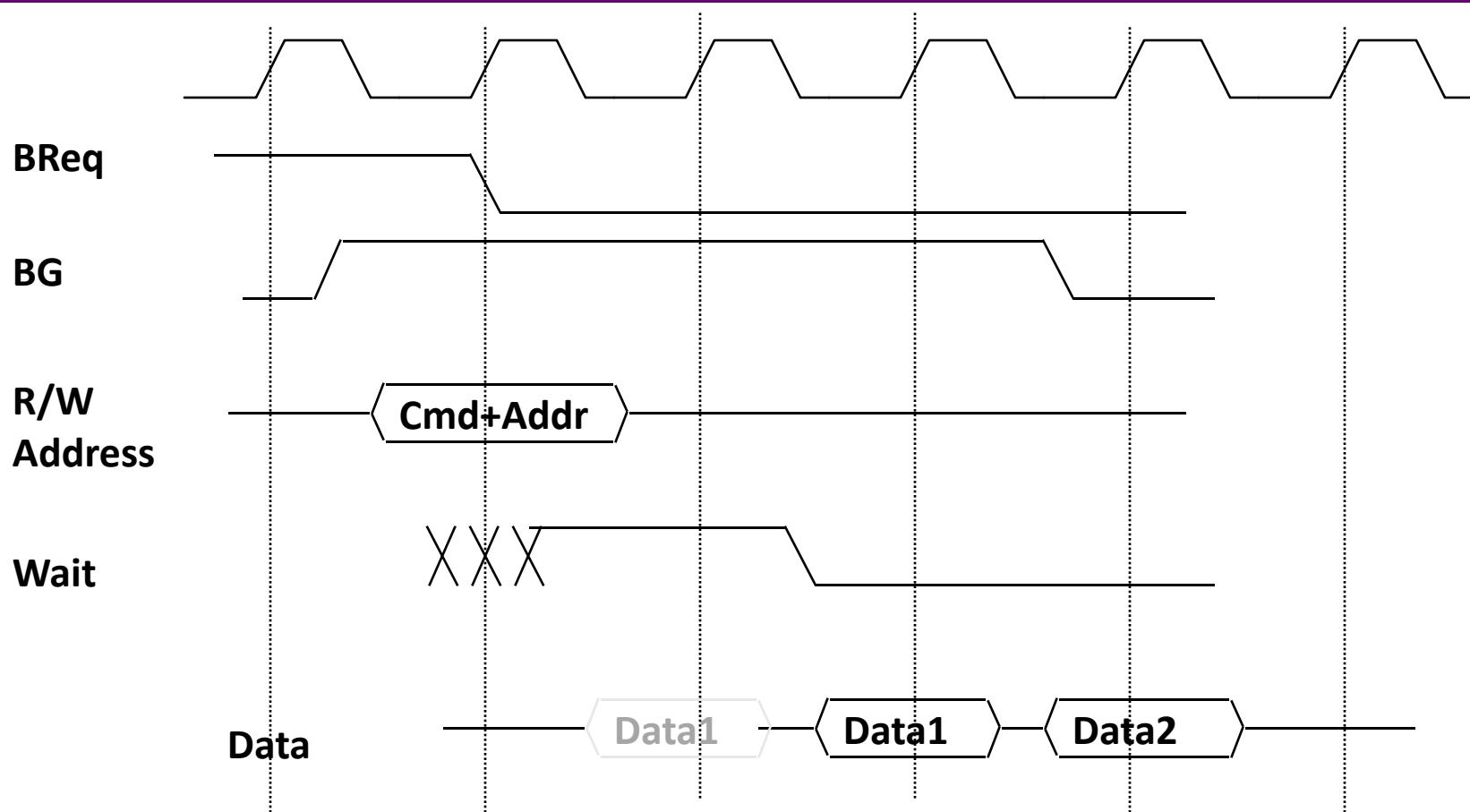
- 所有设备同步工作
- 所有设备以同样的速度工作
- => 简单的协议
  - 只需管理源和目标

# 简单的同步协议



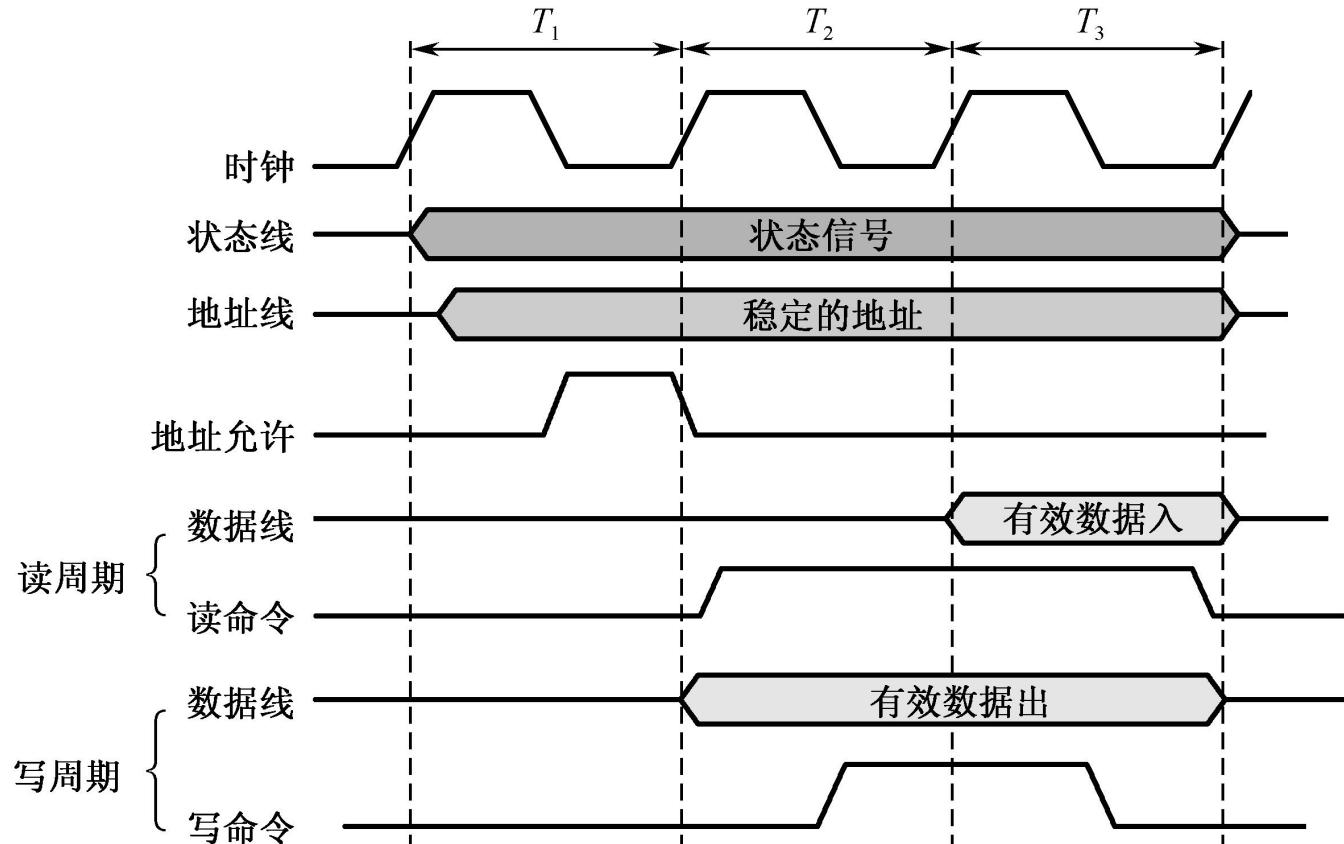
- 就是处理器-主存储器总线也比它复杂
  - 主存（从设备）需要响应时间
  - 需要控制数据速度

# 典型的同步协议

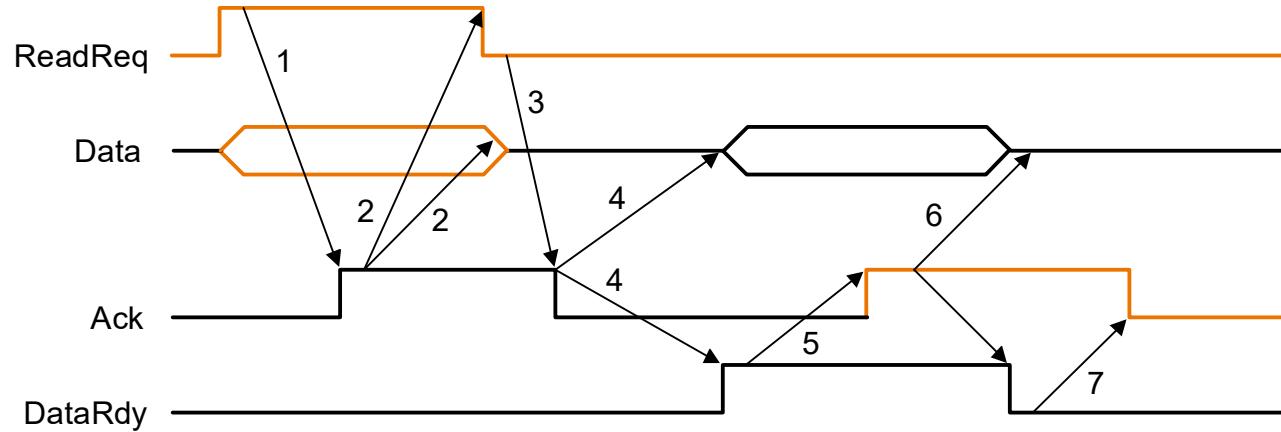


- 从设备指示何时开始传送数据
- 实际传送开始后，按总线时钟传送数据

# 同步定时

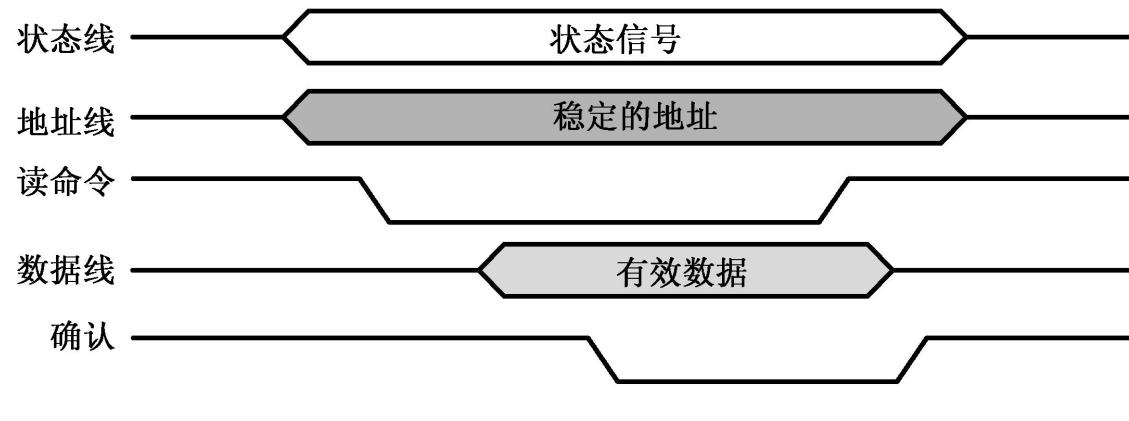


# 典型的异步协议

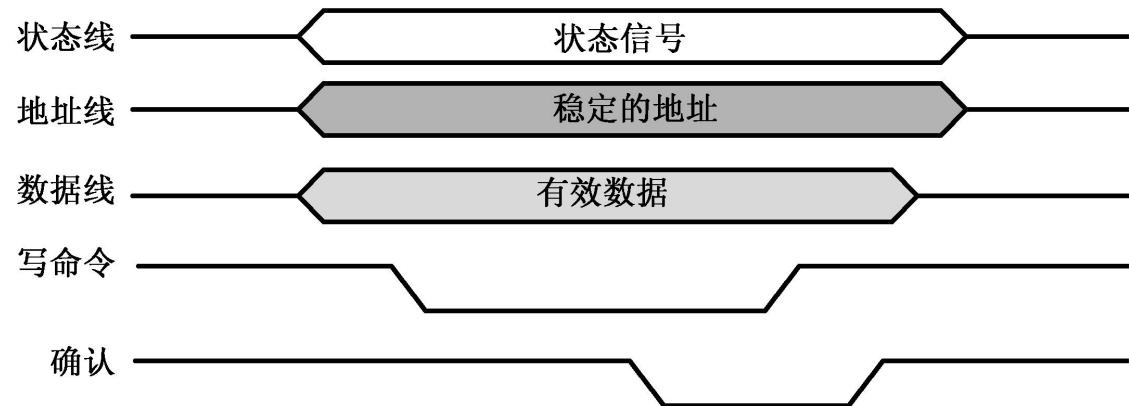


1. 主存储器收到外部设备发出的ReadReq信号，从数据总线读到地址，并发出Ack信号。
2. 外部设备发现Ack信号为高 => 释放ReadReq和数据
3. 主存发现ReadReq信号为低，将 Ack信号置低
4. 主存读出数据后，将数据送总线，并将DataRdy置高
5. 外部设备发现DataRdy为高，读数据，并发出Ack信号
6. 主存发现Ack为高，将DataRdy拉低，并释放数据线
7. 外部设备发现DataRdy为低，拉低 Ack信号，指示传送结束

# 异步定时



(a) 系统总线读周期

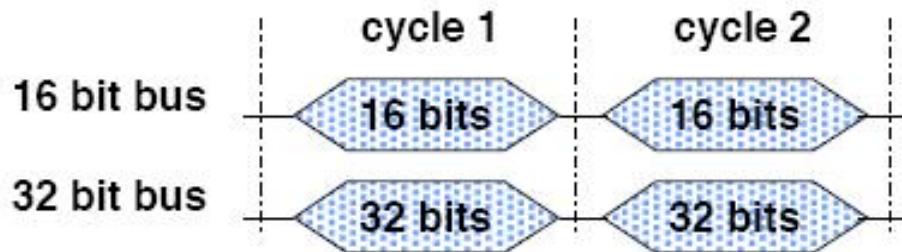


(b) 系统总线写周期

# 增加总线带宽

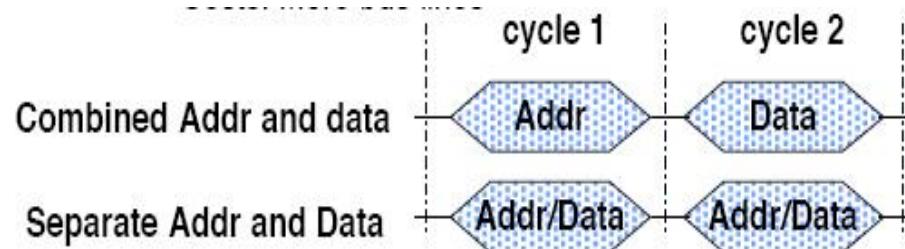
## □ 增加总线的宽度

- 可增加每个周期传送数据的量
- 提高了成本



## □ 分别设置数据总线和地址总线

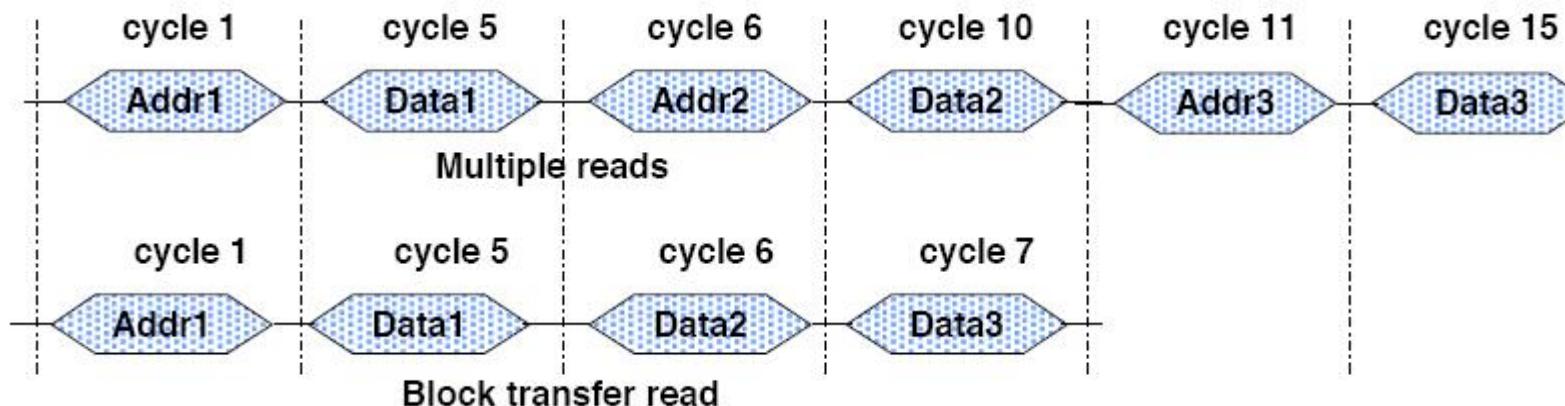
- 可同时传送数据和地址
- 提高了成本



# 增加总线带宽

## □ 采用成组传送方式

- 一个总线事务传送多个数据
- 每次只需要在开始的时候传送一个地址
- 直到数据传送完毕才释放总线
- 代价
  - 复杂度提高
  - 延长后续总线请求的等待时间



# 多主设备总线提高事务数量

---

## □ 仲裁重叠

- 在当前事务时，为下一总线事务进行仲裁

## □ 总线占用

- 在没有其他主设备请求总线的情况下，某主设备一直占用总线，完成多个总线事务

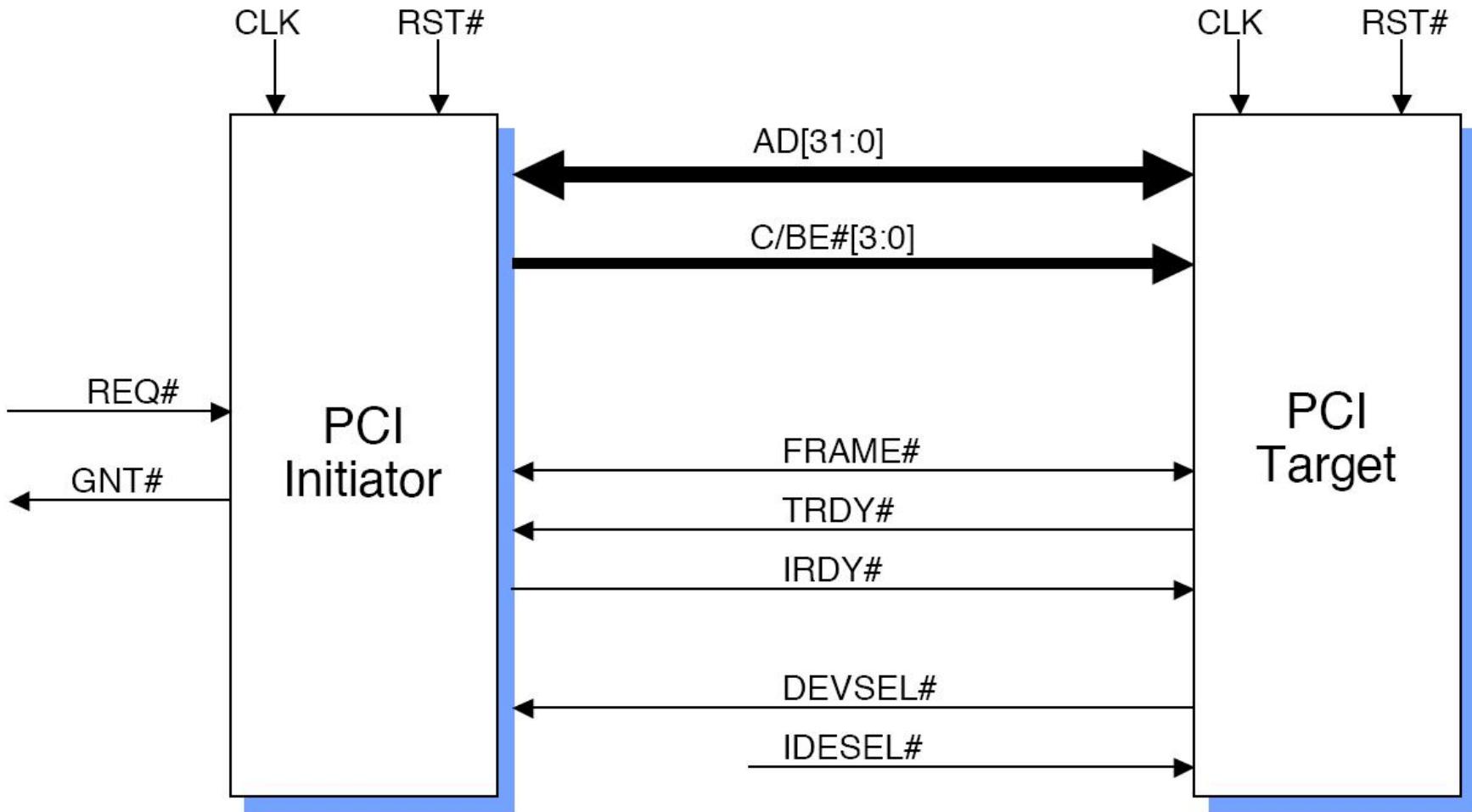
## □ 地址、数据传送重叠

## □ 在现代内存总线上，应用了上述全部技术

# PCI总线

- 外部组件互连总线
- 时钟频率：33MHz或66MHz (CLK)
- 集中仲裁方式 (REQ#、GNT#)
  - 和上一事务重叠
- 32位地址和数据线互用 (AD)
  - V2.1 为64位
- 总线协议
  - 总线周期：内存读、内存写、内存成组读等 (C/BE#)
  - 地址握手和保持 (FRAME#、IRDY#)
  - 数据宽度 (C/BE# )
  - 通过IRDY#和TRDY#握手信号传输变长的数据块
- 最大带宽达133MB (33MHz) 或528MB (66MHz)

# 32位PCI总线的信号

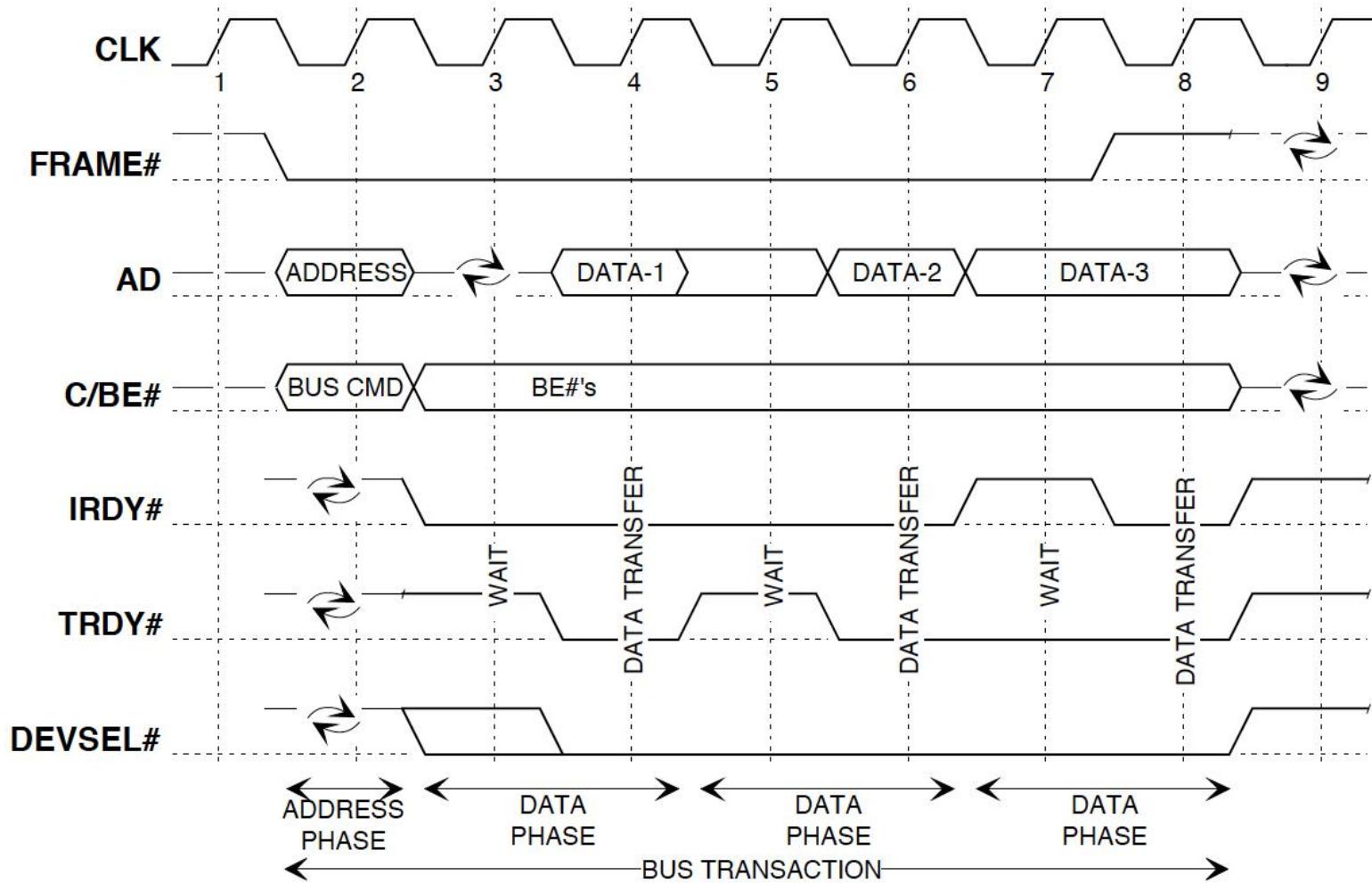


# PCI 总线的读/写事务

---

- 所有信号在时钟正边沿采样
- 集中平行仲裁
  - 和上一事务重叠
- 所有事务可无限制成组传送
- 地址段起始于FRAME#信号有效
- 第一时钟周期主设备发出cmd和address
- 数据传送
  - 当主设备准备好传输数据，主设备发出IRDY# 信号
  - 从设备准备好传输数据，发出TRDY#信号
  - 上述两个信号均有效时的时钟上升沿开始传送数据
- 主设备准备结束数据传送时，将FRAME#信号失效

# PCI 总线读事务



# PCI读

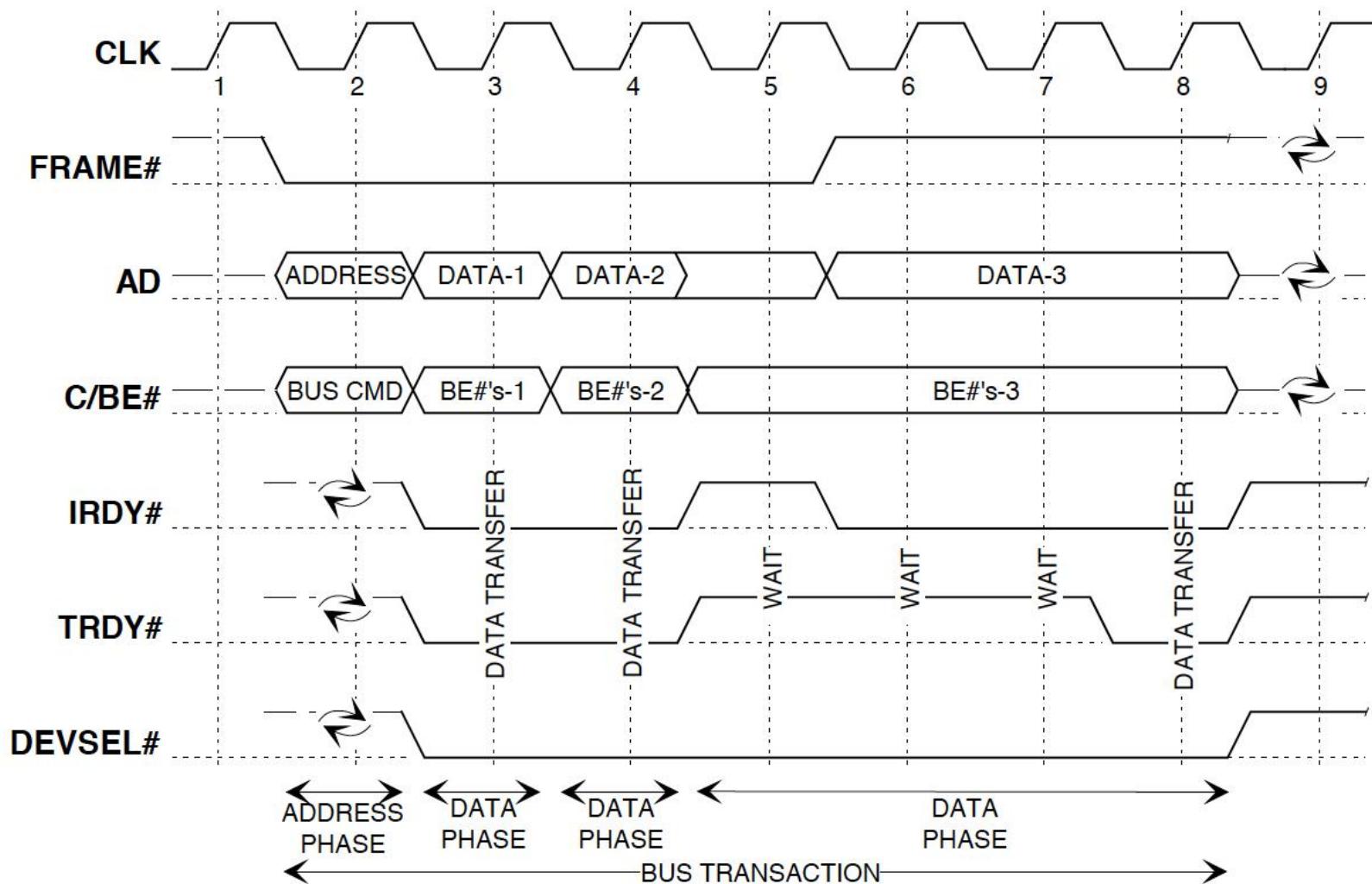
- 总线主设备得到授权后，将FRAME#置为有效，开始读事务。并通过AD发送要读的地址，C/BE#发送读命令
- 从设备从AD上识别是否被选中
- 主设备释放对AD的控制，同时，在C/BE#上给出AD上哪些位是有用的（1~4Bytes）。并置IRDY#为有效，表示已准备好，可以接收数据。
- 被选中的从设备置DEVSEL信号，表示已收到命令并可响应。将读出的数据送AD，并置TRDY#通知主设备接收。

# PCI读

---

- 主设备可在周期4读到第一个数据。并根据需要决定是否要改变C/BE#的值。
- 如果从设备的速度不高，则需要插入等待周期。
- 主设备通过FRAME信号通知从设备结束数据传输，并将IRDY置高。
- 从设备相应地将TRDY和DEVSEL信号置高，总线返回到空闲状态

# PCI写事务



# PCI优化

## □ 尽量使总线有效传输

- 可采用类似RISC技术，仲裁和数据传输并行进行

## □ 总线占用

- 为上一主设备保留总线授权，直到有其他主设备申请使用总线
- 得到授权的主设备可在不仲裁的情况下直接开始下一传送过程

## □ 仲裁时长

- 主设备和从设备尽力延长传输流（使用xRDY）
- 从设备使用STOP (abort or retry)信号终止连接
- 主设备通过FRAME信号终止连接
- 仲裁器通过GNT信号终止连接

## □ 延迟(挂起, 时段分离)事务

- 对慢速设备，在请求后暂时释放总线

# PCI的其它问题

---

## □ 中断：

- 用于支持控制I/O设备

## □ Cache一致性：

- 用于支持I/O和多处理器

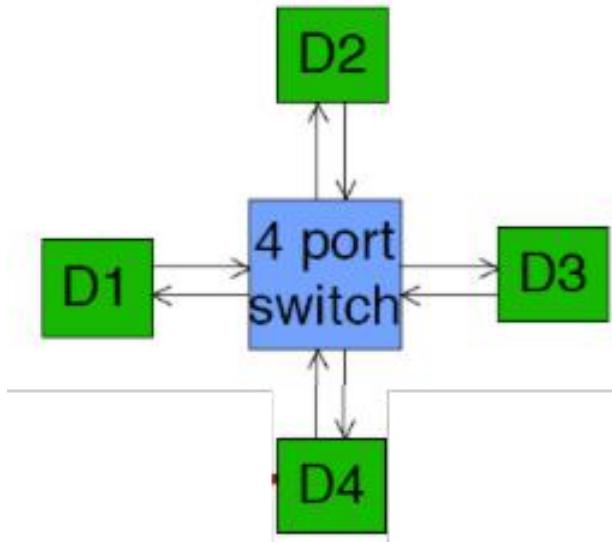
## □ 加锁：

- 支持分时操作, I/O和多处理器

## □ 可配置地址空间

# 总线发展趋势

- 逻辑总线，物理交换
- 许多总线已采用新的点到点标准
  - 3GIO
  - PCI Express
  - Serial ADA



# 总线参数选择

| 选择     | 提高性能                   | 降低成本                  |
|--------|------------------------|-----------------------|
| 总线宽度   | 将地址和数据线分开              | 互用地址和数据线              |
| 数据宽度大小 | 越宽越快（32位）<br>多字可减少总线开销 | 越窄越廉价（8位）<br>传送单字传送简单 |
| 主设备    | 多主设备（仲裁）               | 单主设备                  |
| 时钟     | 同步                     | 异步                    |
| 协议     | 并行                     | 串行                    |

# DMA使用内存总线的方式

- 独占使用：当外设要求传送一批数据时，由DMA控制器发一个信号给CPU。DMA控制器获得总线控制权后，开始进行数据传送。一批数据传送完毕后，DMA控制器通知CPU可以使用内存，并把总线控制权交还给CPU。
- 周期挪用（周期窃取）：当I/O设备没有 DMA请求时，CPU按程序要求访问内存：一旦 I/O设备有DMA请求，则I/O设备挪用一个或几个周期。（随时，一旦冲突， DMA优先）
- DMA与CPU交替访内：一个CPU周期可分为2个周期，一个专供DMA控制器访内，另一个专供CPU访内。不需要总线使用权的申请、建立和归还过程。

# 小结

## □ 总线

- 多个部件之间进行数据传送的共享通道
- 计算机总线
  - 处理器内部总线
  - 系统总线
  - I/O总线

## □ 总线设计

- 总线仲裁
- 数据传输模式
- 提高总线性能

# 阅读和思考

---

## □ 阅读

- 参考书相关内容

## □ 思考

- 计算机总线作用？总线仲裁应考虑哪些方面？
- 总线数据传输模式有哪些？各有什么特点？

## □ 实践

- 本单元作业

## □ 小组答辩

- 请准备好PPT

---

谢谢



# 接口电路和外部设备

2022年秋

# 期末考试时间与地点

---

□ 考试时间：12月27日 下午 14:30 – 16:30

□ 考试地点：

- 课堂1（刘卫东）：一教201
- 课堂2（陈康）：一教205
- 课堂3（陆游游）：一教101

提醒：本周五为Session课（大实验汇报），分成小课堂上课

# 主要教学内容

---

- 接口电路的作用
- 接口电路的一般组成
- 串行接口
- USB接口
- 输入/输出设备

# 输入/输出系统

---

- 控制方式：处理器管理输入/输出的机制
- 总线：数据传输
- 接口：总线和外部设备的连接
  - 总线由多个设备共享
  - 设备之间存在差异
- 设备：完成输入/输出任务
  - 完成数字信号到其它系统可识别信号的转换
  - 是多个学科的交叉和综合

# 接口的基本功能

- 提供主机识别（指定、找到）使用的I/O设备的支持
  - 为每个设备规定几个地址码或编号
- 建立主机和设备之间的控制与通信机制
  - 接收处理器（主设备）的命令，并提交给外部设备，同时，为主设备提供外部设备的状态
- 提供主机和设备之间信息交换过程中的数据缓冲机构
- 提供主机和设备之间信息交换过程中的其他特别需求支持
  - 屏蔽外部设备的差异

# 通用可编程接口电路

## □ 通用

- 能有多种用法与入/出功能

## □ 可编程

- 能通过指令指定接口的功能和运行控制参数

## □ 接口内部组成

- 设备识别电路
- 数据缓冲寄存器（输入/输出）
- 控制寄存器
- 状态寄存器
- 中断电路
- 其他电路

# 串行接口芯片8251A

□ 串行接口，可用于同步或异步传送

□ 同步传送

- 5~8位/字
- 支持内同步或外同步
- 自动插入同步字符

□ 异步传送

- 5~8位/字
- 时钟：1、16或64倍波特率
- 停止位：1、1.5或2位
- 可检测假启动
- 全双工
- 双缓冲发送器和接受器
- 可检测奇偶错、数据丢失错和帧错

# 串行通信

## □ 同步传送

- 采用同步信号
  - 内同步：同步字符
  - 外同步：硬件同步信号

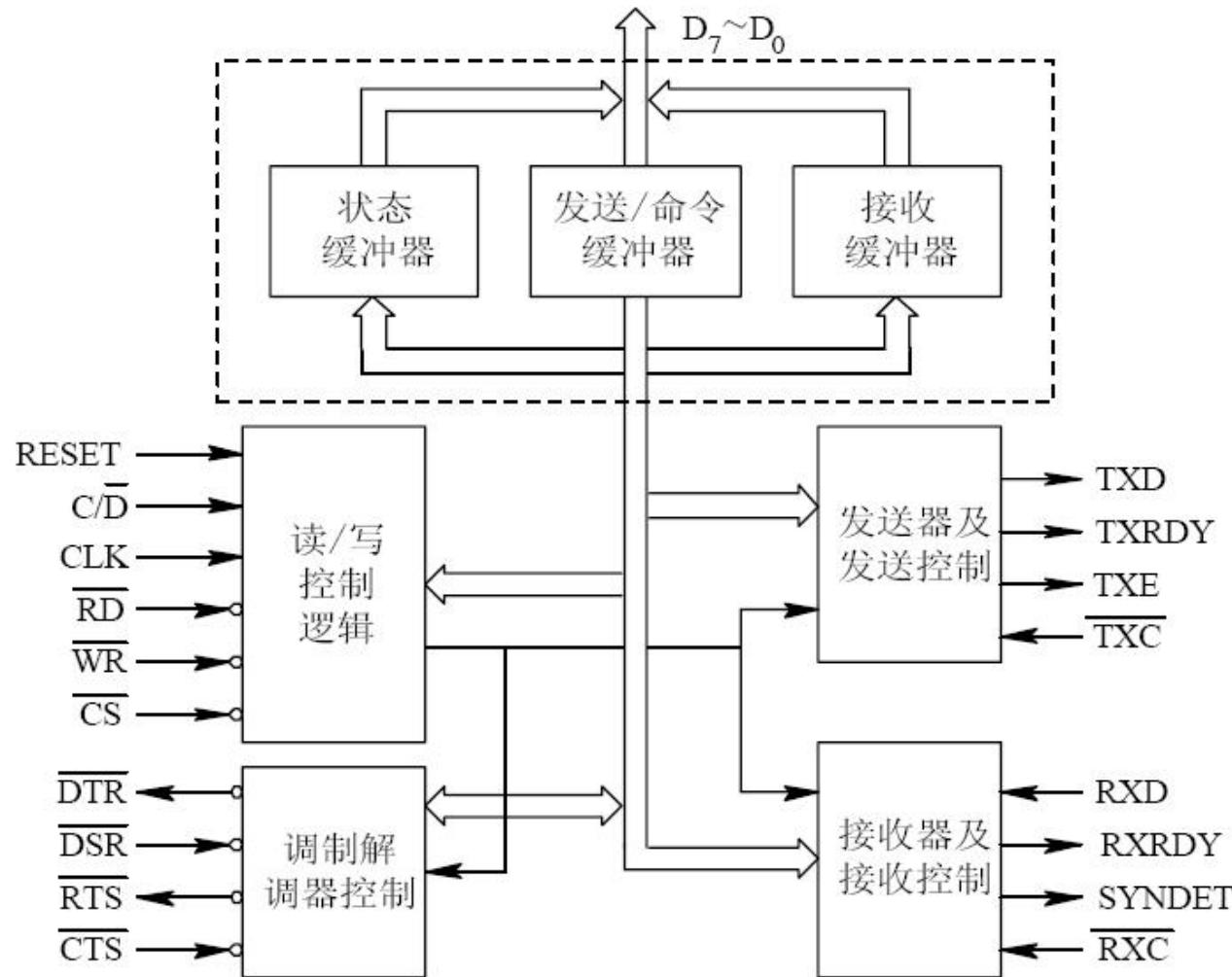
## □ 异步传送

- 起始位、停止位
- 波特率

## □ 全双工

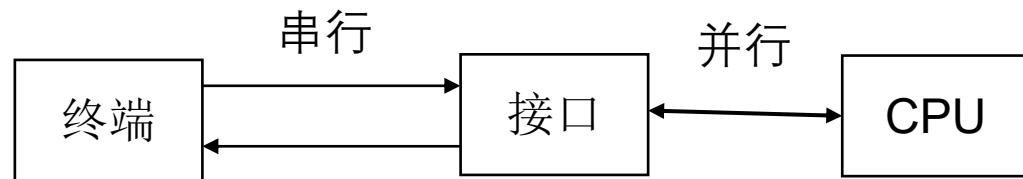
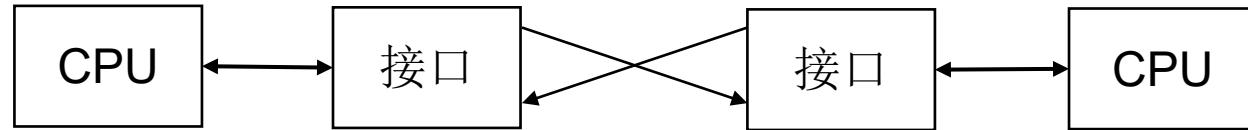
- 通信双方有各自的接收和发送部件，两条数据线

# 8251A结构框图



# 串行传送中的有关概念

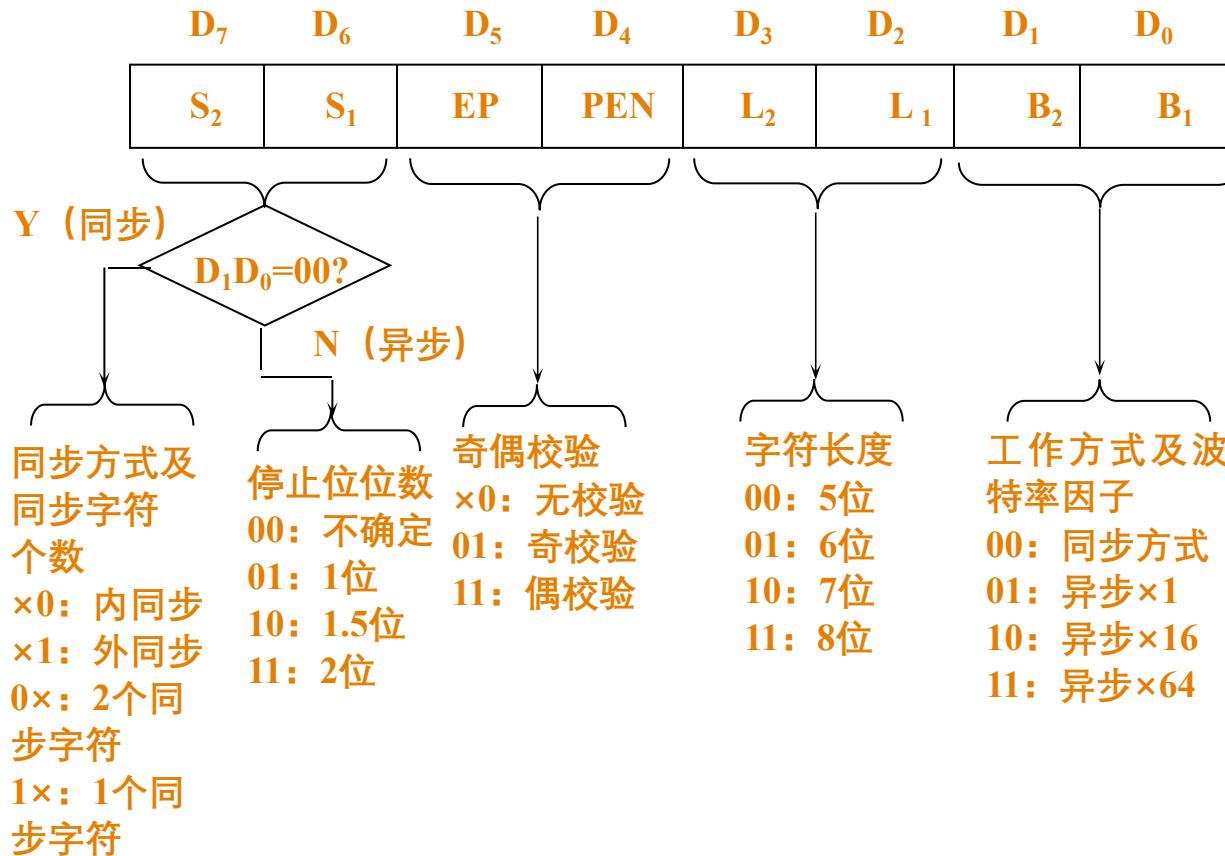
- 串行:
- 异步、同步:
- 单工, 半双工, 全双工:
- 停止位:
- 数据位:
- 起始位:



- 

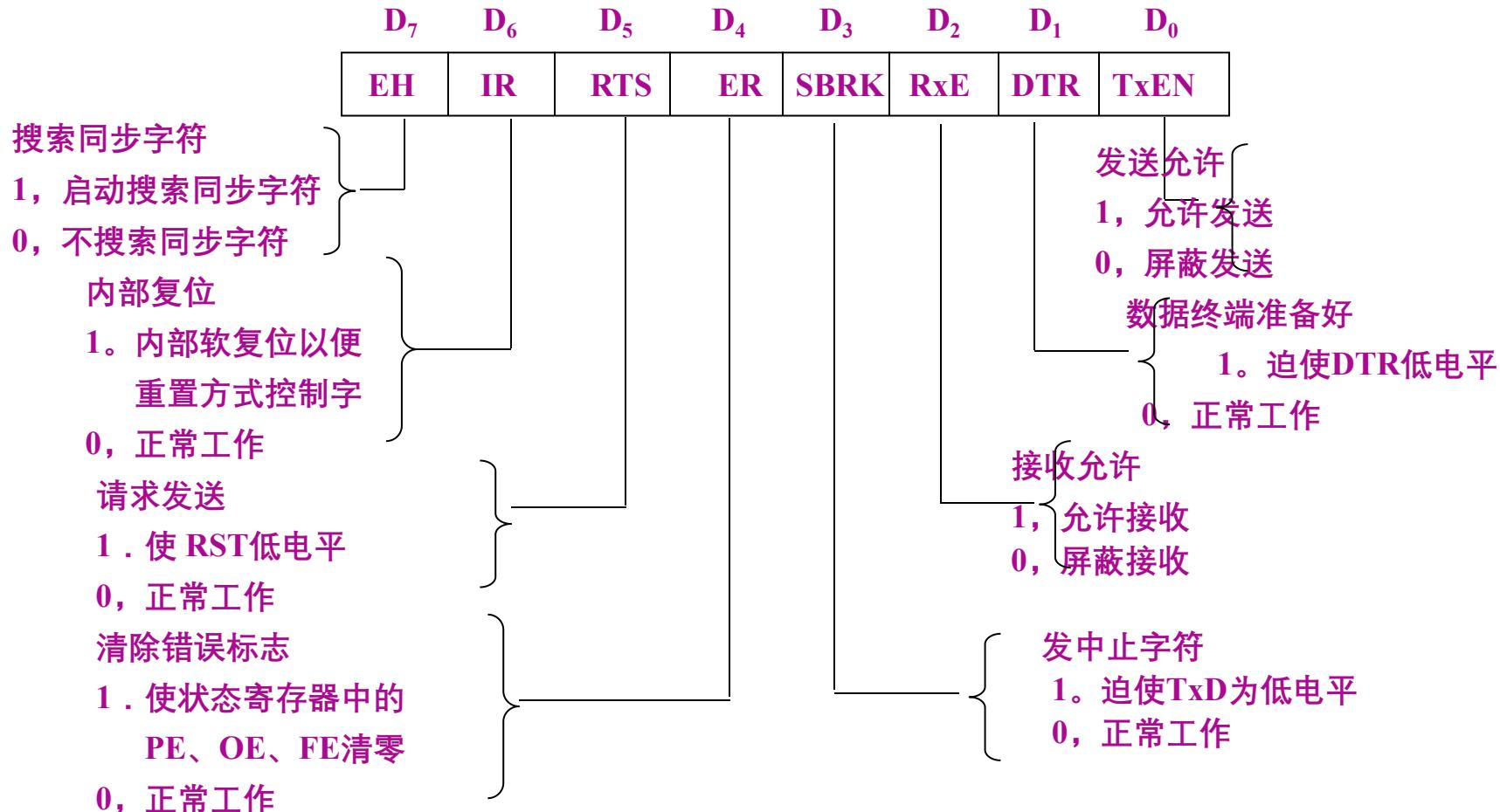
- 奇偶校验:
  - 传送的波特率:
  - 波特率因子:
  - 数据采样:
- 
- The diagram illustrates the structure of a serial data frame. It shows two frames: the **第n个字符 (8~12位)** and the **第n+1个字符**. Each frame consists of several fields:
  - 起始位**: A single bit (0).
  - 5~8位数据位 (先送最低位)**: A group of bits representing the data payload.
  - 奇偶校验位**: A bit used for error detection.
  - 停止位 (1, 1½ 或2位)**: One or more bits used to indicate the end of a character.
  - 空闲位**: A bit used for synchronization.

# 方式命令字的格式



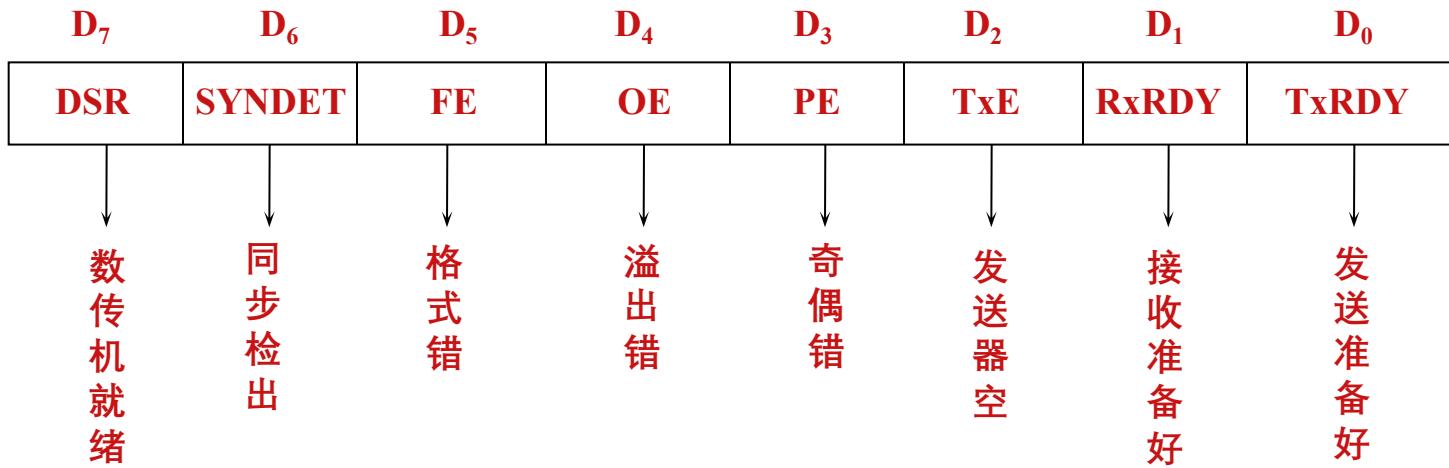
8251A 方式控制字格式

# 工作命令字的格式



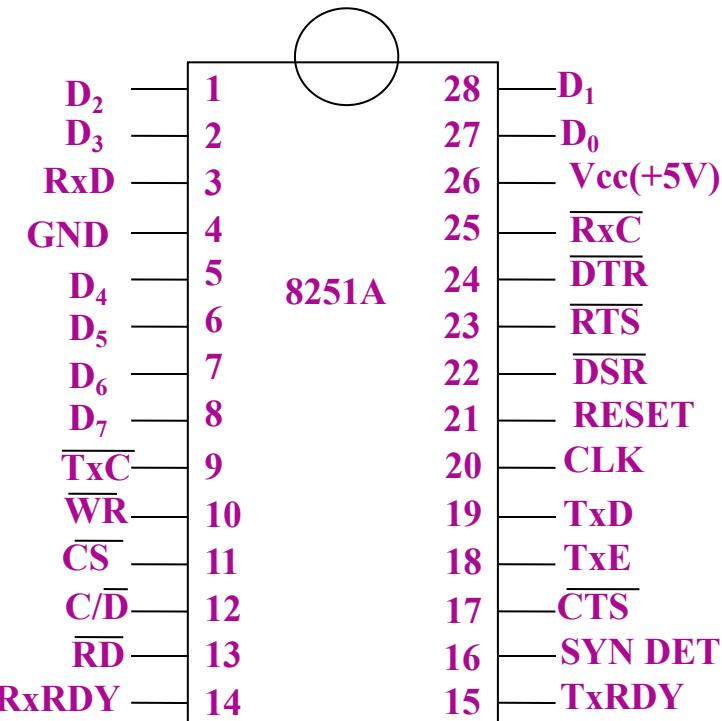
操作控制字格式

# 接口状态寄存器的内容格式



状态字格式

# Intel 8251A串行接口芯片



器件引脚图

- D7~D0 : I/O数据
- CLK: 主时钟
- /Rx<sub>C</sub>, Rx<sub>D</sub>: 接收时钟、 数据
- /Tx<sub>C</sub>, Tx<sub>D</sub>: 发送时钟、 数据
- /WR、 /RD: 写、 读命令
- /CS: 片选信号
- C/ D: 控制/ 数据信号
- RESET: 总清信号
- RxRDY: 接收准备就绪
- TxRDY: 发送准备就绪
- TxEMPTY: 发送寄存器空
- /DTR、 /DSR:
- /RTS、 /CTS:

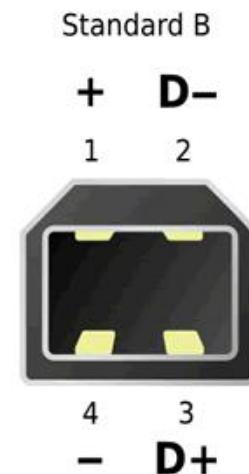
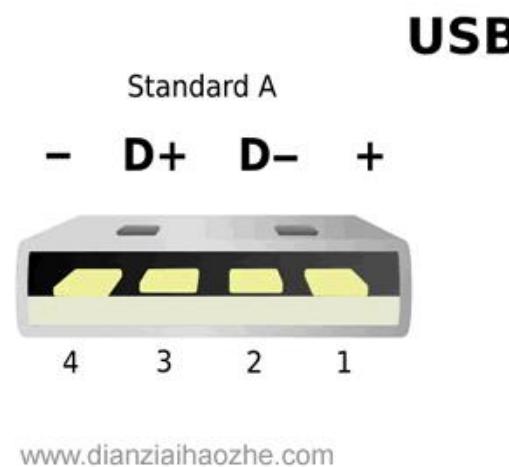
# USB接口

---

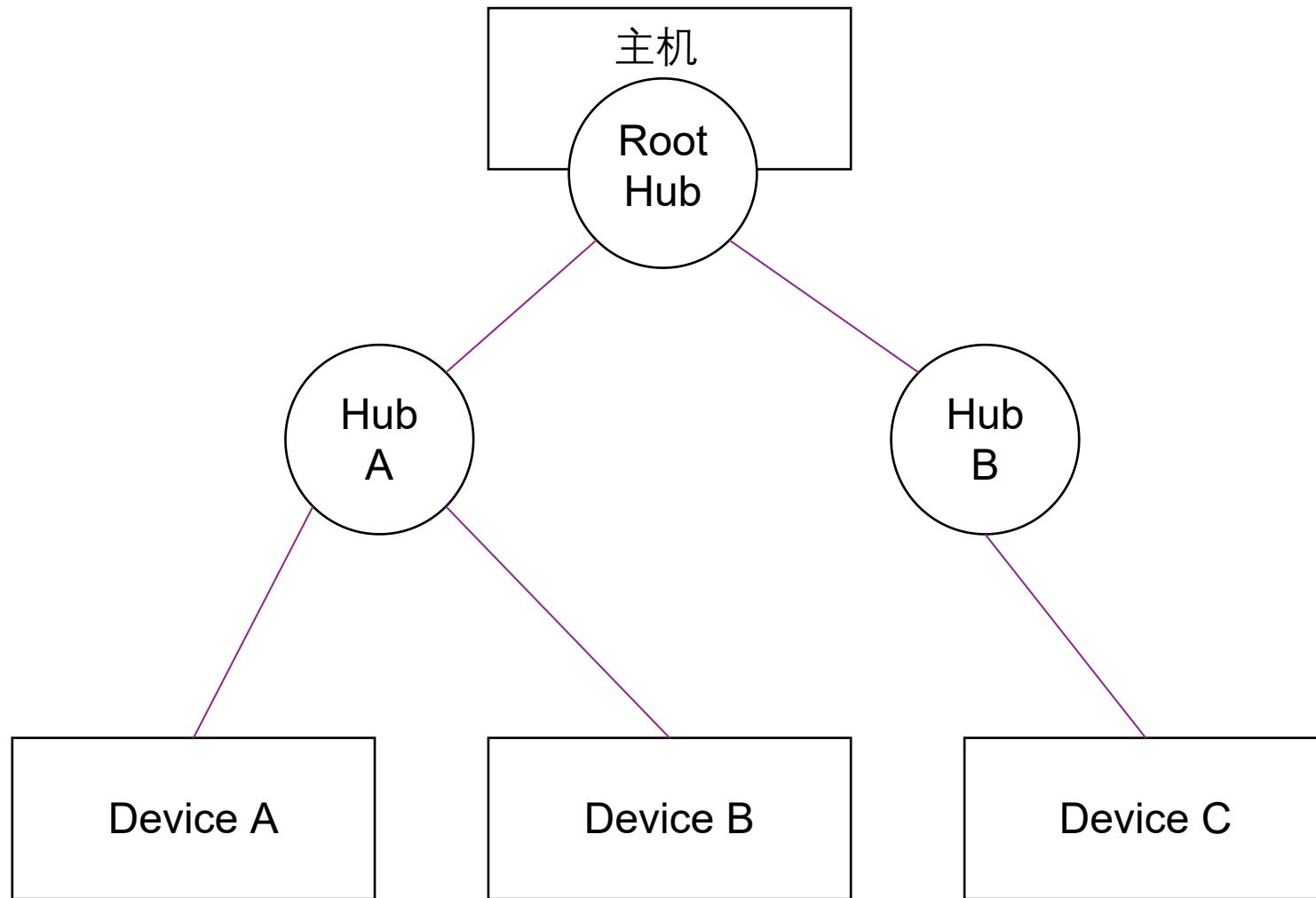
- 用户不必再设置卡上、设备上的开关或跳线
- 不必打开机箱来安装新的输入输出设备
- 应该只需要一根电缆线就可以将所有设备连接起来
- 输入/输出设备应可以从电缆上得到电源
- 单台计算机最多可以连接127个设备
- 系统应能支持实时设备（声卡、电话）
- 可在计算机运行时安装设备
- 不必重新启动计算机
- 成本低

# USB线缆

- 由4根线组成，电源、地和双数据线。
- 同步传输方式



# USB结构



# USB接口工作原理

## □ USB结构

- 根HUB、层次结构

## □ 设备检测

- 根HUB定时查询接口状态，若检测到有设备接入到接口上，则为该设备赋地址（7位）。设备初始地址为0，每个设备上应有ROM，保存设备参数。

## □ 识别设备类型后，由设备驱动程序管理和使用设备。

- 操作系统支持

## □ 只有1个主设备，不需要仲裁，采用轮询方式，适合低速设备使用。

## □ 设备带宽为1.5MB/s。可适合一般的语音设备。

- V2.0 60MB/s
- V3.0 500MB/s

# USB帧

---

## □ 控制帧

- 配置设备，对设备发出命令，查询设备状态

## □ 同步帧

- 实时设备同步

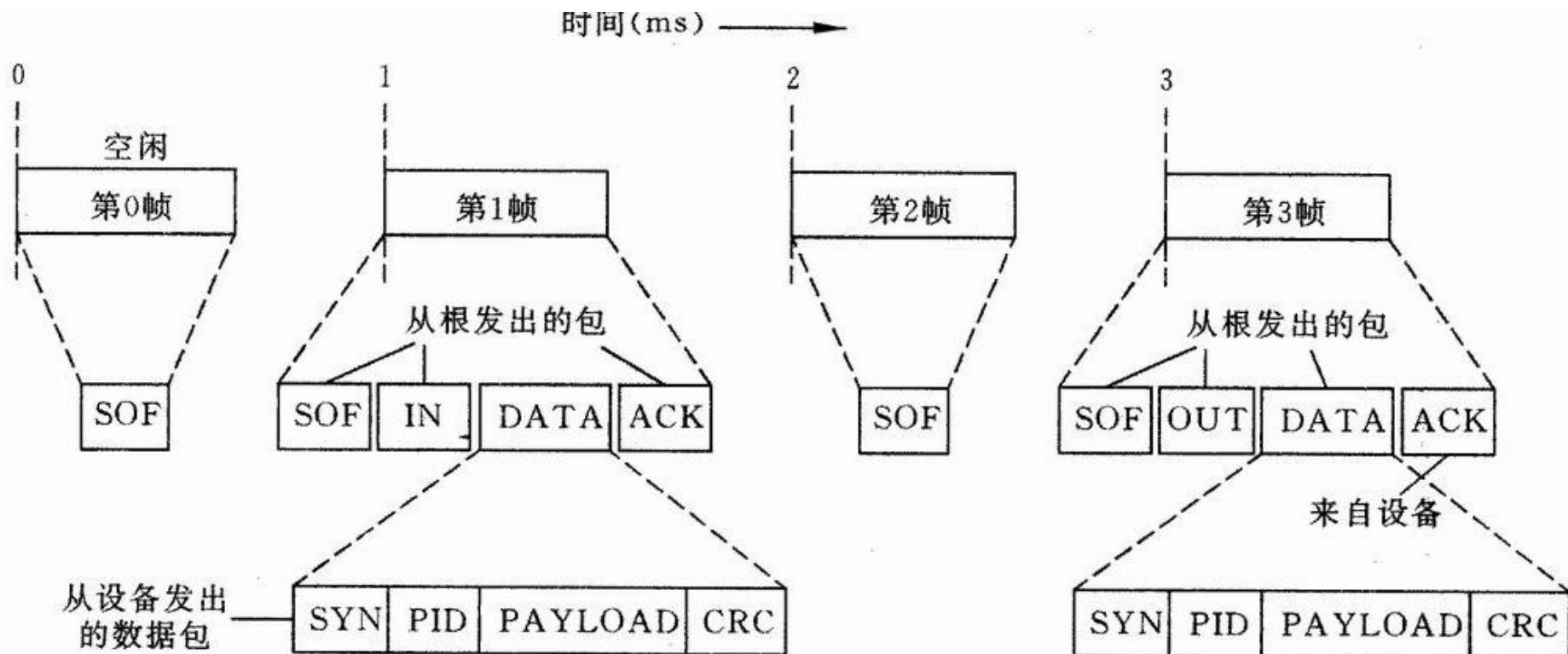
## □ 块传送帧

- 非实时设备的大量数据传送

## □ 中断帧

- 发出中断帧，收集设备数据

# USB协议



# USB协议

- 每1ms， 定时发出一个SOF包， 进行时间同步（所有设备）。
- 协议包
  - 令牌包 (SOF、 IN、 OUT、 SETUP)
  - 数据包(Data)
  - 握手包(ACK、 NAK、 STALL)
  - 特别包
- 第1帧： 根发出读命令 (IN) ， 包含有地址； 设备返回数据包DATA(最多64位)， 其中， SYN同步字段 (8位) 、 PID为包类型 (8位) 、 载荷(Payload)， 和16位校验码； ACK为根接收到数据后返回给设备的确认包。
- 第3帧： 往设备写数据。

# 接口

---

## □ 连接外部设备

- 设备识别
- 数据缓冲
- 协议实现
- 屏蔽差异

## □ 通过总线与主机进行通信

# 外部设备

---

- 输入/输出设备
- 外存储器
- 脱机输入/输出设备
- 主要完成人机交互
- 是电子、机械、光学、化学等多学科的交叉
- Anyway, Anywhere, Anytime, Anyone
- 智能化

# 外部设备功能

---

□ 完成数据的输入和/或输出

- 信号转换
- 数据采样

□ 与接口进行连接

- 接口信号，电平标准等

□ 与主机进行通信

- 通过总线进行
- 速度
- 控制方式

# 键盘

---

## □ 功能要求

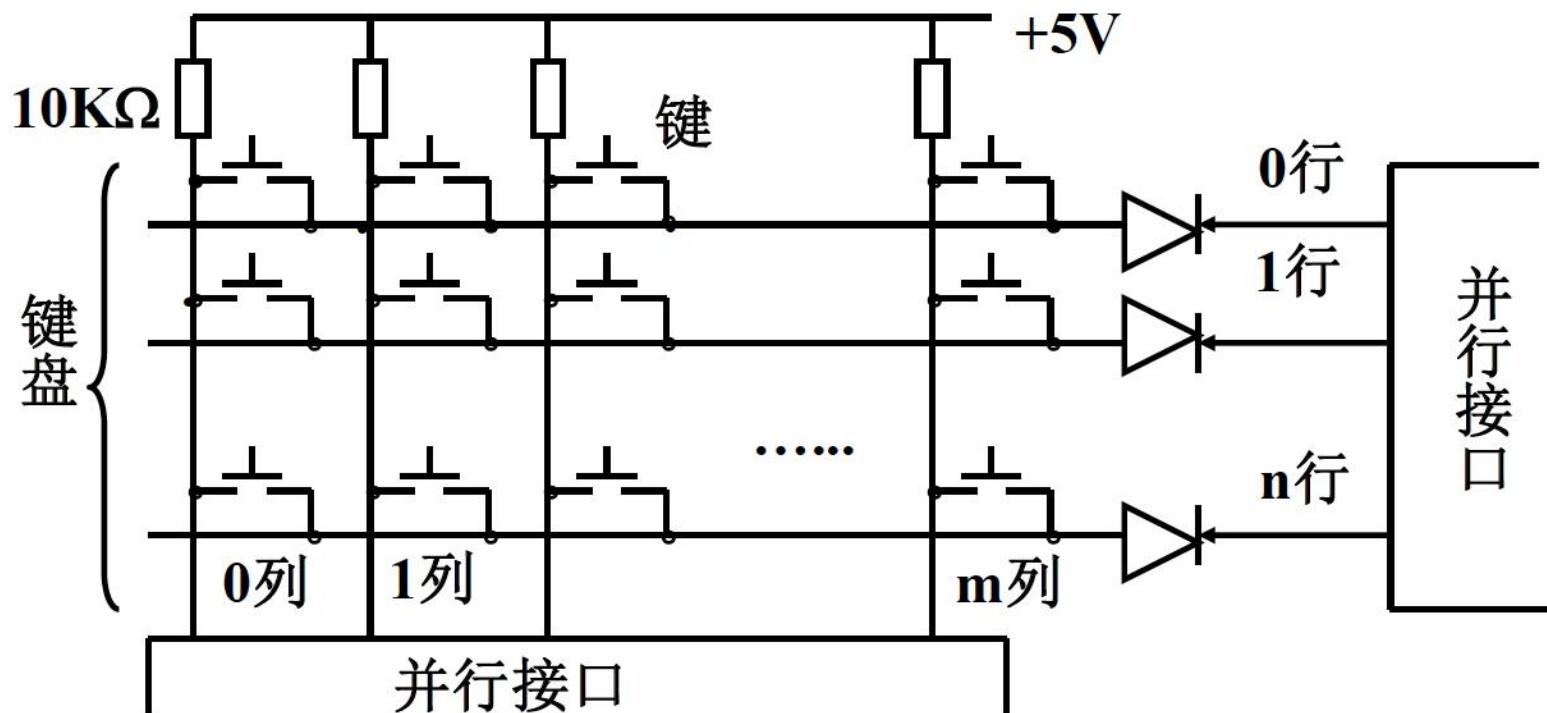
- 能完成字符的输入

## □ 设计要求

- 完成功能
- 稳定可靠

# 键盘运行原理

计算机的键盘，用于向主机内敲入字符、功能键、汉字等符号，通过逐次敲击键盘上不同的键来完成。被敲击的键将以一个特定的编码被表示并被存入计算机主机。故键盘的运行原理，是把敲击的键在键盘上的位置对应为一个编码。

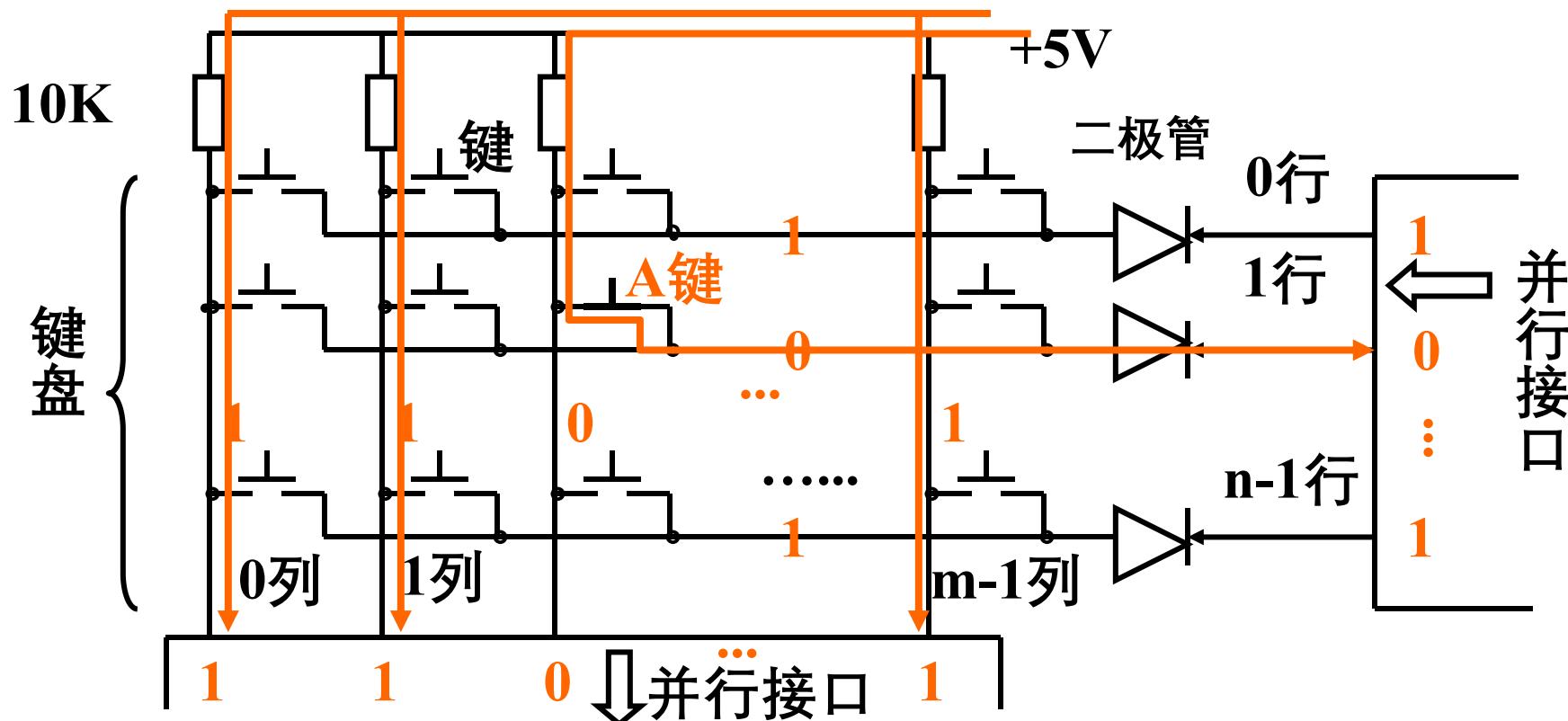


# 键盘运行原理

- 把每个键在键盘上的位置对应为一个编码。
- 具体实现：是用行列扫描的方法，即把每个键分配在一个 $m$  列\*  $n$  行矩阵的一个交叉点上，通过并行接口向 $n$  行依次送出仅有一行为零、其余各行均为一的值，再用并行接口读入 $m$  列上的取值。
- 当该值不为FFH（全1码）时，表明有键按下，若该值仅含一位零，表明取值为0的行、列的交叉点的键被按下，用一个对照表即可得到相应键的编码。
- 尚需解决如下的一些问题：键的抖动、多键同时按下、由哪个部件完成这些操作过程。

# 键盘的运行原理

- 并行接口送来  $1\ 0 \dots 1$  的  $n$  位数值到二极管的负极，并行接口接收 键盘线路  $m$  列送出的  $m$  位数据。当 A 键按下去后， $5V$  电源送出经电阻、A 键、二极管到 0 信号处的电流，从而在 第 2 列产生 0 电平（红线所示），其他各列都给出高电平（黑线所示），故并行接口接收到的是  $1\ 1\ 0 \dots 1$  这样的  $m$  位数据



# 键盘接口

---

- 采用串行口或者并行口
- 中断方式
- 总线
  - USB
  - 慢速总线

# 鼠标

## □ 鼠标的产生

- 图形界面的出现，需要鼠标来进行拖动等操作

## □ 鼠标的的功能

- 根据鼠标的移动，在屏幕上移动位置
- 选中某个对象，进而执行某些操作

## □ 鼠标的种类

- 机械式鼠标
- 光电式鼠标

## □ 鼠标的接口

- 串口、PS2接口、USB接口

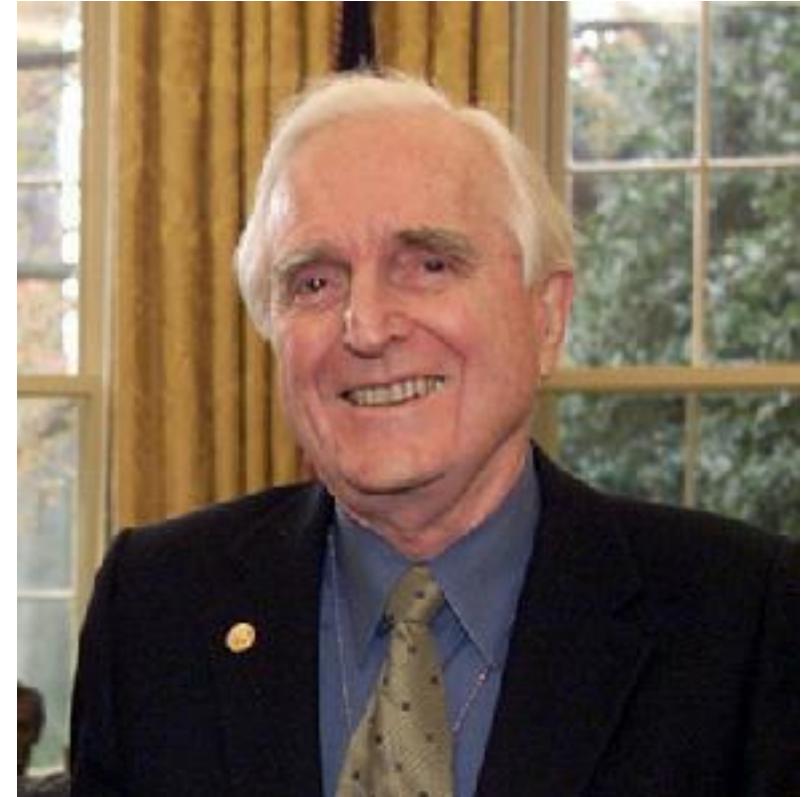


# 鼠标的发明

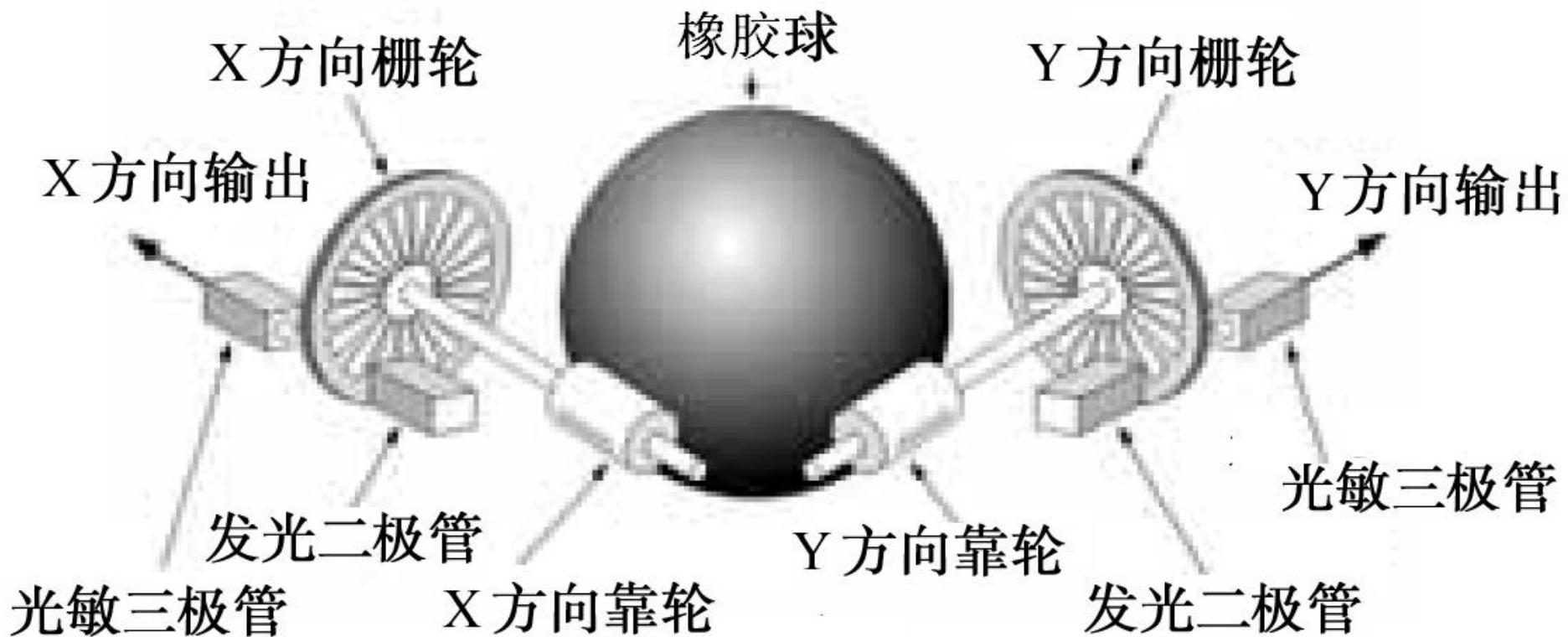


# 鼠标的发明

- 道格拉斯·恩格尔巴特 (Dr. Douglas C. Engelbart, 1925年1月30日—2013年7月2日)
- 早在20世纪60年代初，他就发表了一篇名为“放大人类智力” (Augmenting the Human Intellect)的论文，提出了计算机是人类智力的“放大器”的观点。为此，他认为必须改善人机交互方式，发展交互式计算技术。1997年Turing奖获得者



# 机械式鼠标



# 机械式鼠标

- 鼠标内部有一个橡胶球，橡胶球紧贴着两个互相垂直的轴（X、Y轴），每个轴上有一个光栅轮，光栅轮两边对应着有发光二极管和光敏三极管。
- 鼠标在移动的时候，橡胶球便带动两个轴旋转，时光栅轮也就开始旋转，光敏三极管在接收发光二极管发出的光时被光栅轮间断地阻挡，从而产生脉冲信号，通过鼠标内部的芯片处理之后被CPU接受。
- 脉冲信号的频率和数量，经过CPU计算后则表示为屏幕上的距离和速度。

# 智能输入设备

---

- 语音识别
- 手写体识别
- 印刷体识别

# 输出设备概述

---

## □ 点阵式输出设备（视觉）

- 以点阵的组合来表示不同的形状
- 提供每个点的存储输出属性
- 点阵输出设备将点按属性规定的颜色和灰度输出

## □ 听觉

- 音乐、语音合成

## □ 触觉

- 可穿戴计算机

# 点阵输出设备

---

## □ 显示器

- CRT
- LCD
- PDP

## □ 打印机

- 针式打印机
- 激光打印机
- 喷墨打印机

# 阴极射线管(CRT)显示器

---

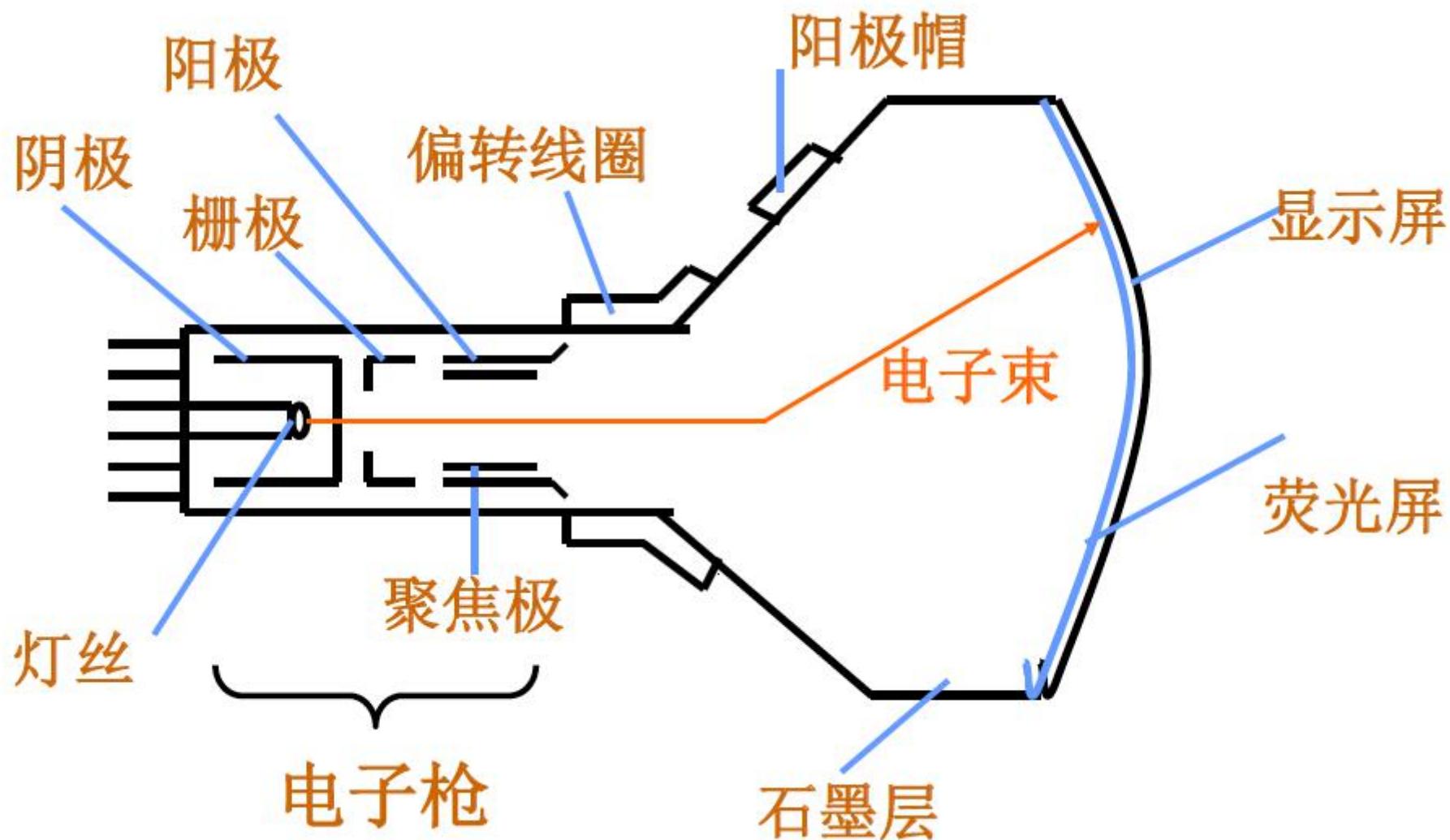
## □ 成像原理

- 通过电子束撞击荧光板上的荧光粉，发光产生亮点

## □ 组成

- 电子枪、显示屏和偏转控制装置

# 阴极射线管（CRT）的构成



# CRT的几个概念

## □ 光栅扫描和随机扫描

- 电子束从左到右，从上到下扫描整个屏幕
- 只扫描需要显示的点

## □ 刷新和帧存储器

- 为了得到稳定的图象，需要重复扫描整个屏幕
- 为了重复扫描，需要存储图象信息。

## □ 分辨率和灰度级

- 像素个数
- 亮暗差别

## □ 图形和图像

- 线条的有无表示
- 自然景物、照片等

# CRT图形显示器

---

- 容量大的VRAM
- 存储点阵属性
- 分辨率： 1024\*768， 真彩色  
 $1024 \times 768 \times 3\text{Byte} = 2.3\text{MB}$
- 高速总线
- 50场/秒， 带宽为 $2.3 \times 50\text{MB/s} = 112.5\text{MB/s}$
- 需要连接PCI总线
- 专用接口
- 分辨率更高的图形设备将采用专用接口

# 液晶显示器

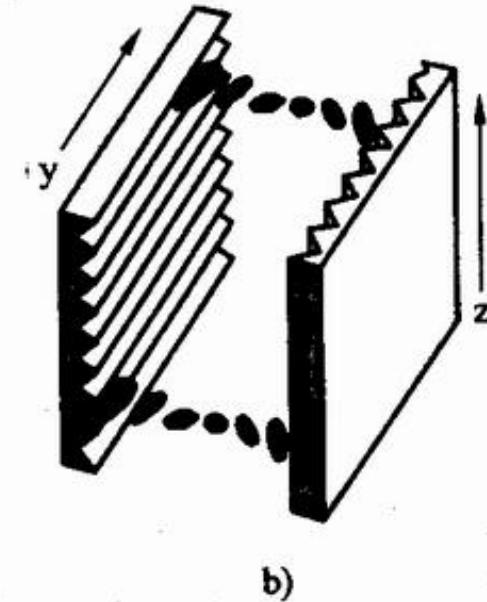
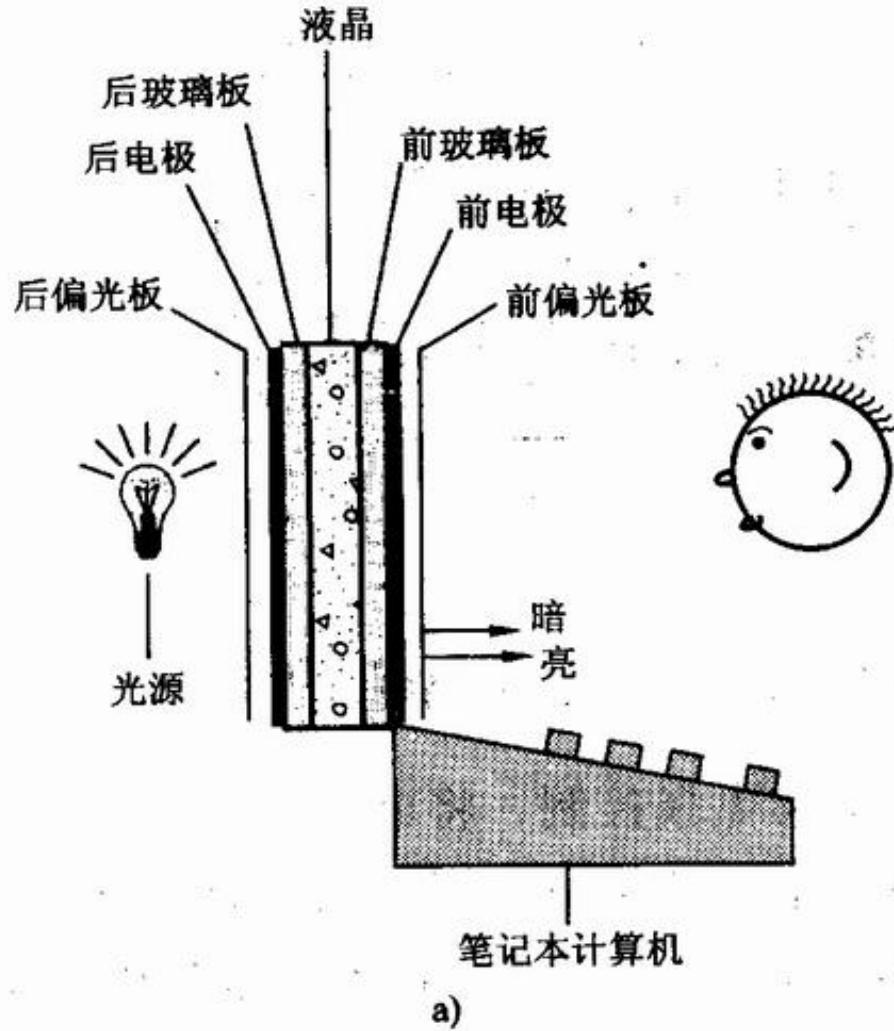
## □ 显示原理

- 利用液晶的光学特性
- 平板后面设置光源
- 通过液晶改变透射光的偏振性（从水平到垂直）
- 电场控制

## □ 特点

- 平板显示，不需要高压电，移动方便
- 无辐射
- 价格较高

# 液晶显示器



# 等离子显示器

## □ 成像原理

- 利用惰性气体在一定电压作用下产生气体放电的特性
- 产生紫外线，紫外线激发荧光粉发光
- 在玻璃板之间隔开成象素，每个象素点内有惰性气体和三色荧光粉，用电极控制

## □ 特点

- 易于实现大画面显示
- 全色显示，色纯度与CRT相当
- 视角达160度
- 寿命长
- 功耗大、成本高、对比度差。

# 激光打印机

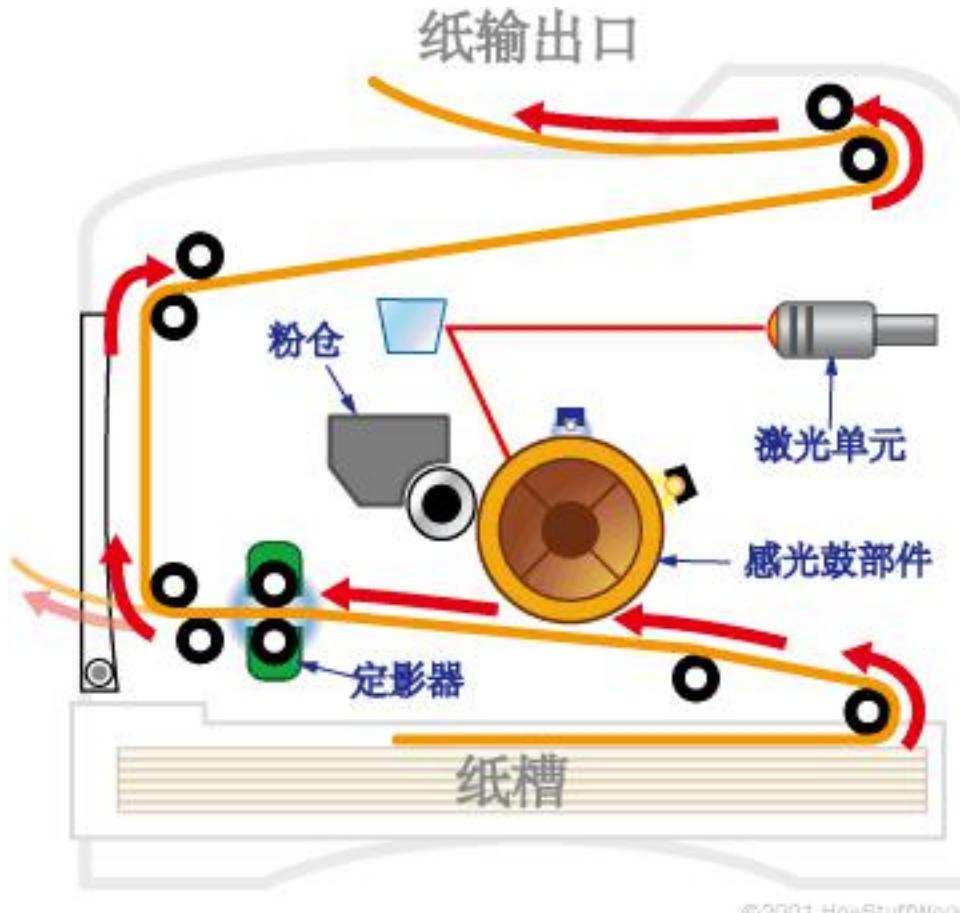
## □ 输出原理

- 利用激光束照射硒鼓，使之放电，不再吸附墨粉来产生打印的形状

## □ 输出过程

- 硒鼓带电后吸附墨粉
- 激光束使硒鼓表面被照射的部分放电，释放墨粉
- 将墨粉压到纸上，并用高温烘烤，使之固化在打印纸上
- 将硒鼓放电，清扫剩余墨粉

# 激光打印机组成



©2001 HowStuffWorks

# 打印机

---

## □ 接口

- 并行接口

## □ 总线

- 慢速总线

## □ 协议

# 输入/输出设备

---

- 种类多样，功能繁杂，速度不一
- 满足计算机和外界进行信息交换的需要
- 人机交互的界面

---

谢谢