操作系统原理

2018-2019学年第一学期

联系方式

- 教师姓名：张国强
- Tel: 13851435685
- Email: zgqoop_cn@hotmail.com; guoqiang@ict.ac.cn

- 办公室：S2-211

教学材料

- 费翔林主编，《操作系统教程》，高教出版社
- Walling Stallings, Operating Systems: Internals and Design Principles
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, Operating Systems: three easy pieces
- Randal E. Bryant, David R. O'Hallaron, Computer Systems: A Programmer's Perspective(深入理解计算机系统)
- Abraham Siberschatz, Peter Bear Galvin, Operating System Concepts, seventh Edition
- 费翔林主编，《Linux操作系统实验教程》，高教出版社 (实验课使用)

必备基础

- 计算机基础
- 汇编语言设计
- 程序设计
- 数据结构
- 计算机组成原理

第一章
计算机系统概述

本章教学目标

- 掌握计算机的组织结构
- 掌握指令执行流程
- 掌握中断处理机制
- 掌握存储器结构
- 掌握I/O通信的方式

操作系统和硬件的关系

- 操作系统利用硬件资源向用户提供一组服务
- 操作系统为用户管理CPU、内存、外存储器和I/O设备
- 因此，需要了解计算机硬件和组织结构的基本知识

提纲

- 处理器结构
- 指令执行
- 中断
- 存储器结构
- I/O通信技术

计算机系统基本元素

- 计算机系统包括四个主要部件
  - 处理器
    - 控制计算机的操作，执行数据处理功能
    - 当仅有一个处理器时，通常称为中央处理器(CPU)
  - 主存
    - 存储数据和程序
    - 易失性存储
  - I/O模块
    - 在计算机和外设之间移动数据。
    - 外设包括磁盘、通信设备、终端等。
  - 总线
    - 提供处理器、主存和I/O模块之间通信的通道

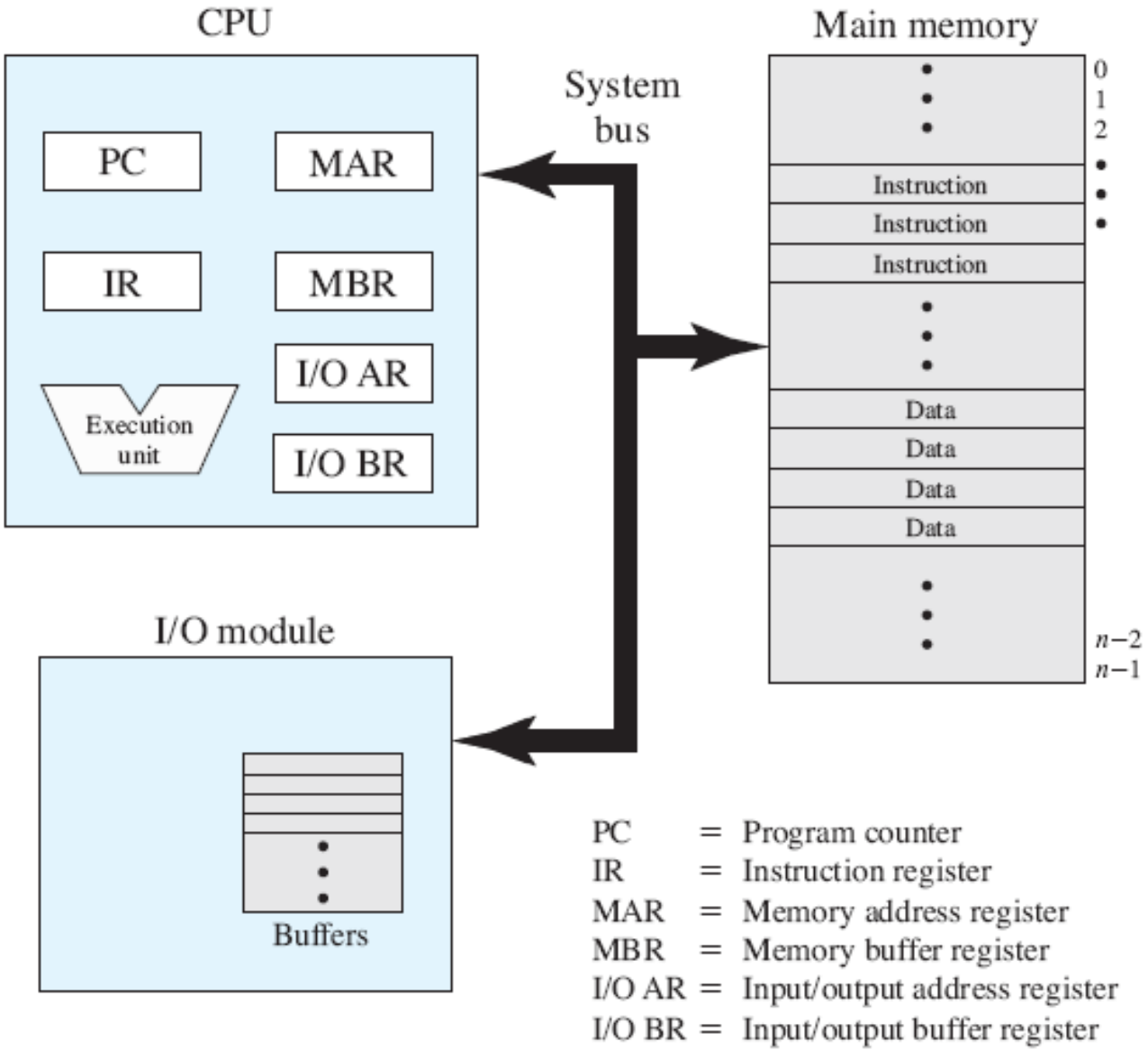


Figure 1.1 **Computer Components: Top-Level View**

注：总线更细致的分类可以分为system bus, memory bus和I/O bus

Bus本质上是一组用于传输**地址、数据或控制信号**的线

寄存器抽象分类

- MAR: 主存地址寄存器，用于给出下一次读/写内存的地址
- MBR：主存缓冲寄存器，用于存放将要写入主存或从主存读取的数据。
- I/O AR：I/O地址寄存器，给出特定的I/O设备
- I/O BR: I/O缓冲寄存器, 存储在处理器和I/O模块之间交换的数据

寄存器

- 用户可见(user visible)
  - 机器语言或汇编语言可以操控
  - 高级语言的编译优化
- 控制和状态(control and status)
  - 用于控制处理器的执行
  - 特权的操作系统例程用于控制程序的执行

用户可见寄存器

- 数据寄存器
- 地址寄存器
  - 索引寄存器：将索引加到基址寄存器，获得有效地址
  - 段指针寄存器：存放段基址，可能有多个，如，OS一个，正在执行的用户程序一个。
  - 栈指针寄存器：指向栈顶元素
- 过程调用是否存储这些寄存器？
  - 有些处理器自动存储，因此每个过程都可以独立使用这些寄存器
  - 有些处理器不自动存储，程序员需要存储相关的寄存器内容

控制和状态寄存器

- 程序计数器(Program Counter: PC)
  - 存放下一条将要执行的指令的地址
- 指令寄存器(Instruction Register: IR)
  - 存放当前取得的指令
- 条件码
  - 处理器硬件执行操作时设置的一些比特，属于执行的结果之一
  - 可以在随后用条件转移操作来测试条件代码
    - 例如，算术操作的结果为正数、负数、零、还是溢出
- 中断寄存器

控制和状态寄存器

- 控制和状态寄存器的设置需要了解操作系统的需求
  - 分页内存管理需要提供页表基址寄存器
  - 分段内存管理则需要段基址寄存器
  - 若中断例程较少，则可以用寄存器存放中断例程的地址
- 控制和状态寄存器的设置也取决于控制信息在寄存器和内存之间的分配
  - 通常会将内存低地址的一部分(如几百个字)用于控制目的

寄存器的变化： IA32 vs x86-64

- IA32中通用寄存器个数为8
- X86-64中通用寄存器个数为16
- 许多过程调用的参数传递不再通过栈传递，而是直接通过寄存器传递

提纲

- 处理器结构
- 指令执行
- 中断
- 存储器结构
- I/O通信技术

指令集

- 指令集是对操作系统硬件的第一次抽象
  - 虽然具体实现时同一时刻可能有多条指令并行执行，但对用户程序来说提供了指令按序执行的抽象保证
- 不同的机器有不同的指令集
  - 决定了兼容性
- 不同的机器可以提供相同的指令集，但其实现方式可以不一样
- 常见的指令集：
  - X86(IA32)
  - RISC
  - EM64T(x86-64)

指令执行

- 指令的执行涉及多个操作，取决于指令本身。
- 指令周期：一条指令的处理过程
  - *取指阶段*
  - *执行阶段*

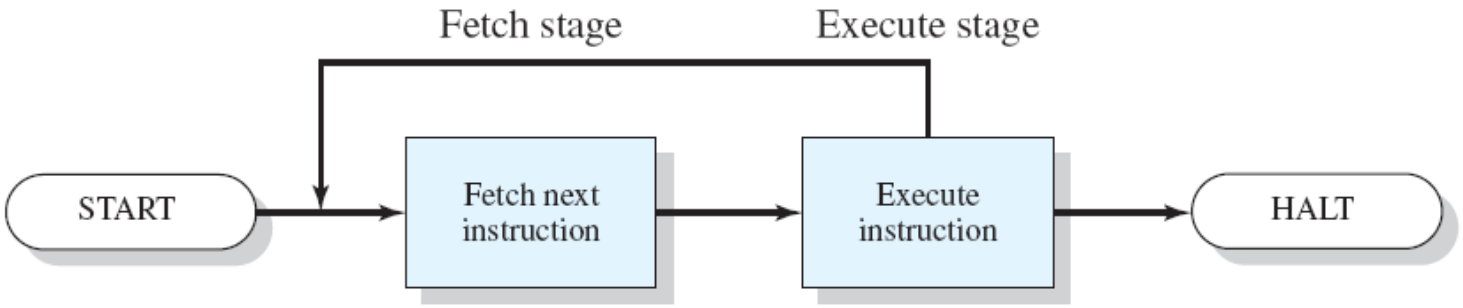


Figure 1.2 **Basic Instruction Cycle**

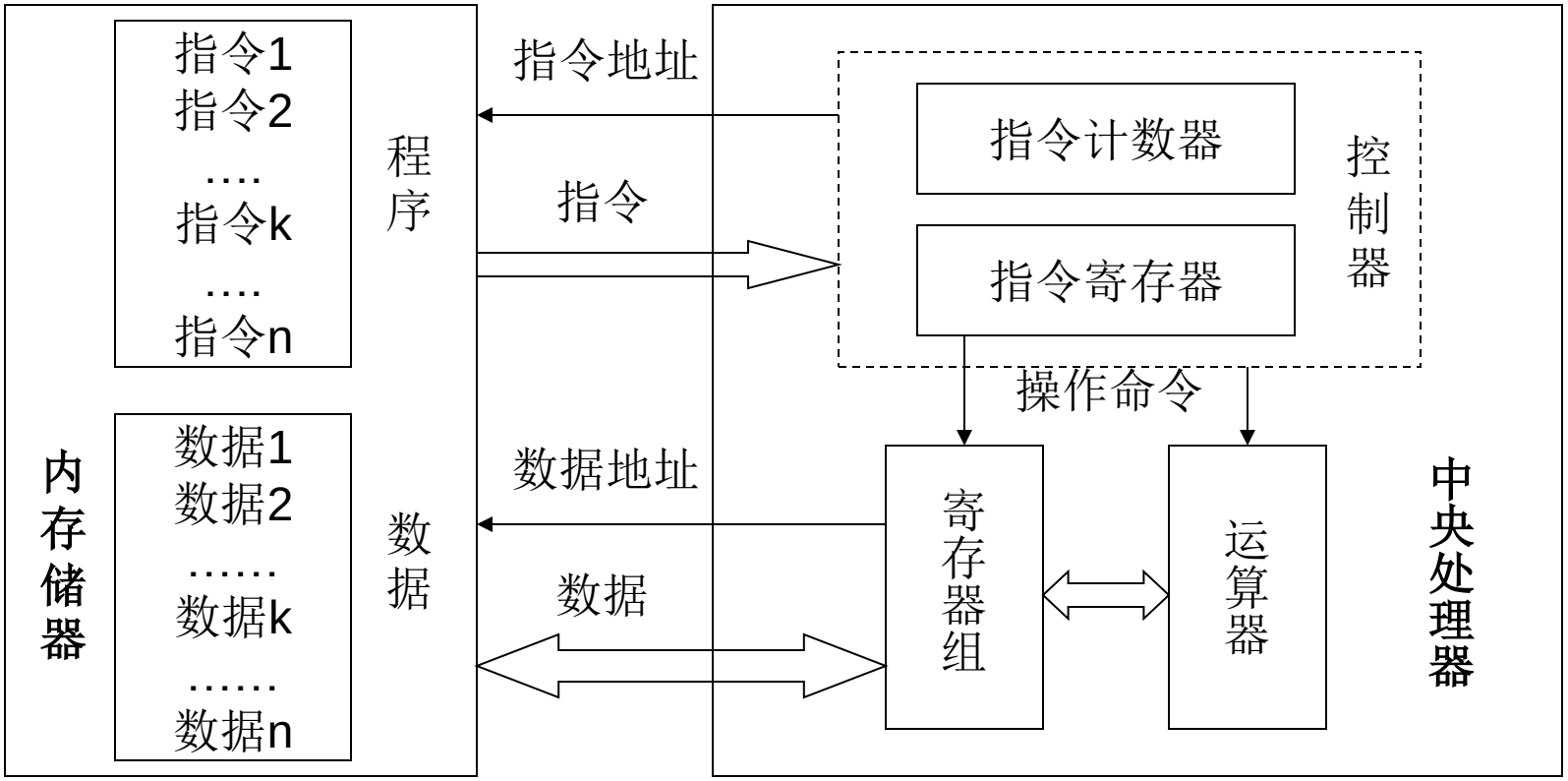
指令分类

- **处理器-主存**：在处理器和主存之间传输数据
  - LOAD
  - STORE
- **处理器-I/O**：在处理器和I/O之间传输数据
  - 读写、控制、查询I/O寄存器
- **数据处理**：处理器执行算术或逻辑运算
  - ADD
  - AND
  - OR
- **控制**：可以改变指令执行的流程, 如转移指令
  - JUMP
  - JZ

指令

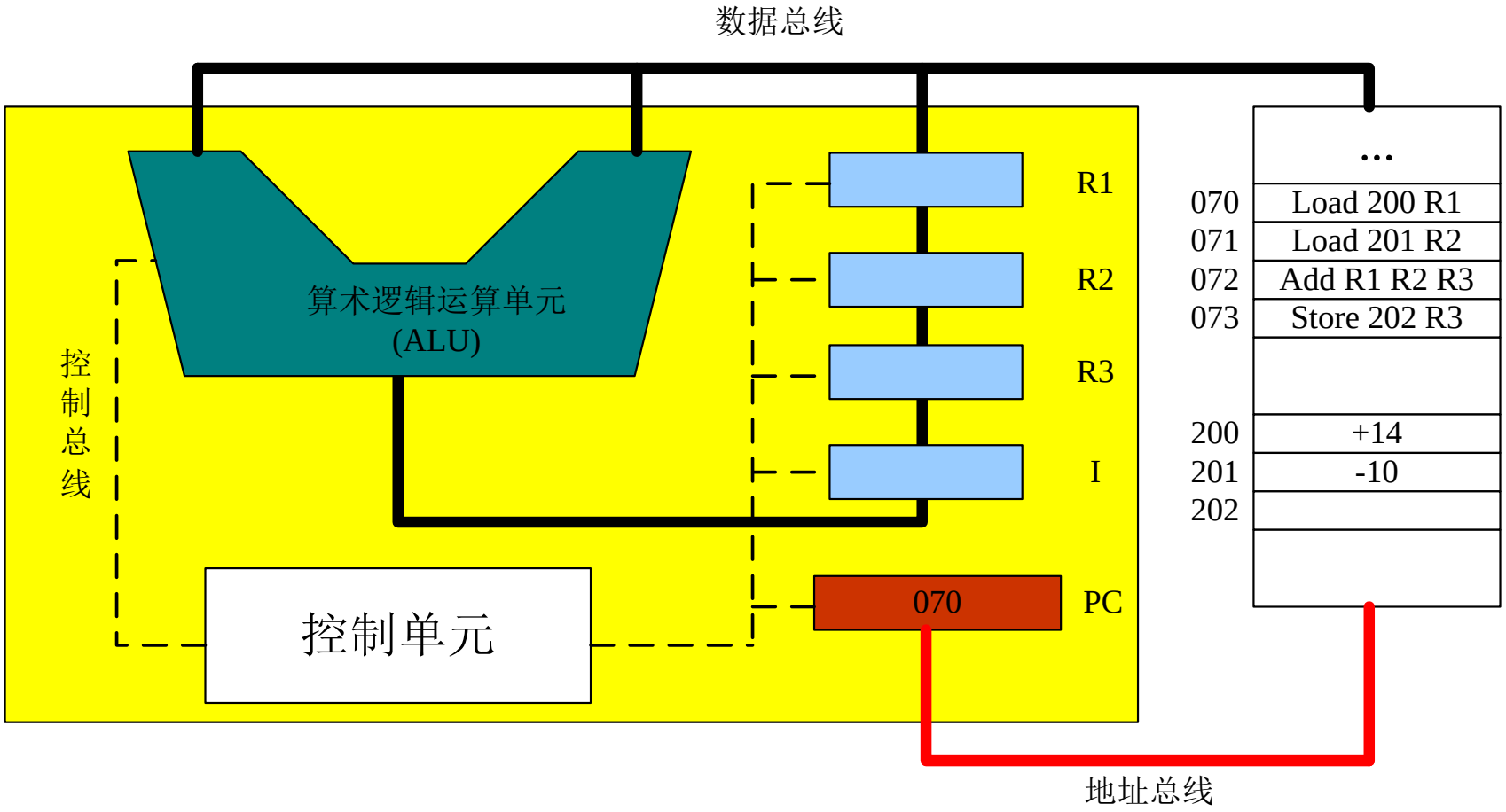
指令的格式

- **操作码**
  - 指出计算机应该执行何种操作的一个命令词，如加、减、乘、除、取数、存数等
- **操作数地址**
  - 指出该指令所操作的数据或数据所在的位置。操作数地址可能是1个、2个、甚至多个，由操作码决定
- 寻址方式
  - 直接
  - 间接

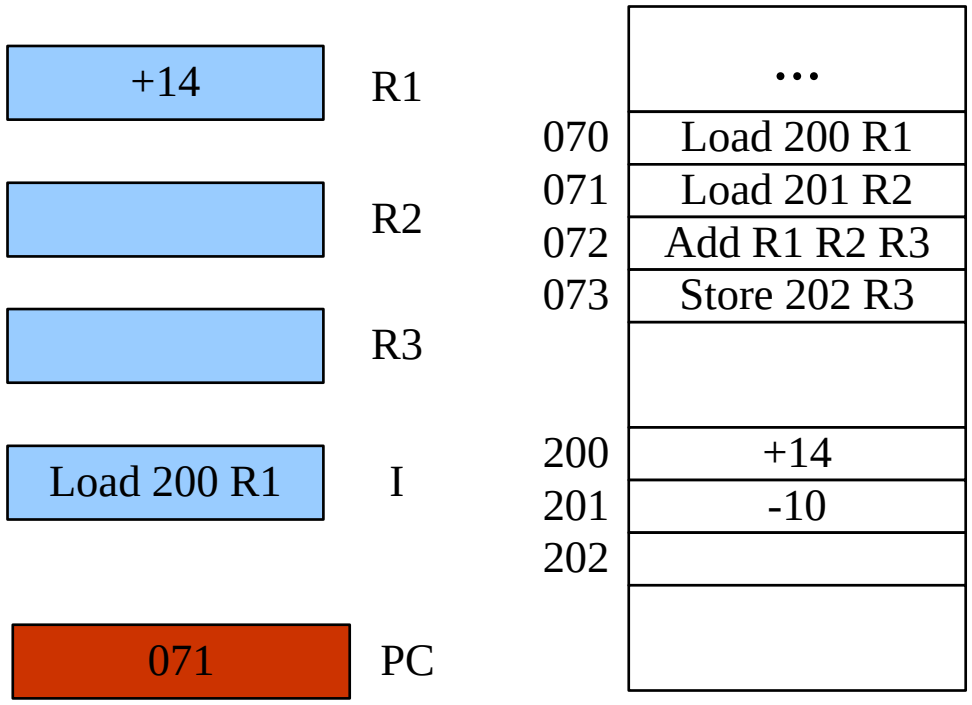


CPU的组成及其与内存的关系

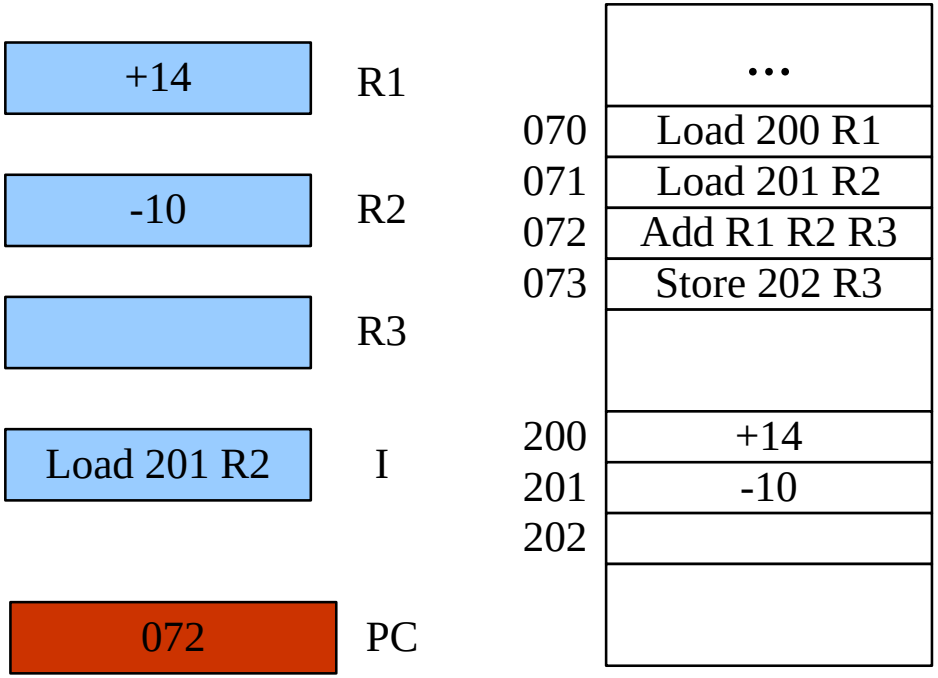
例子



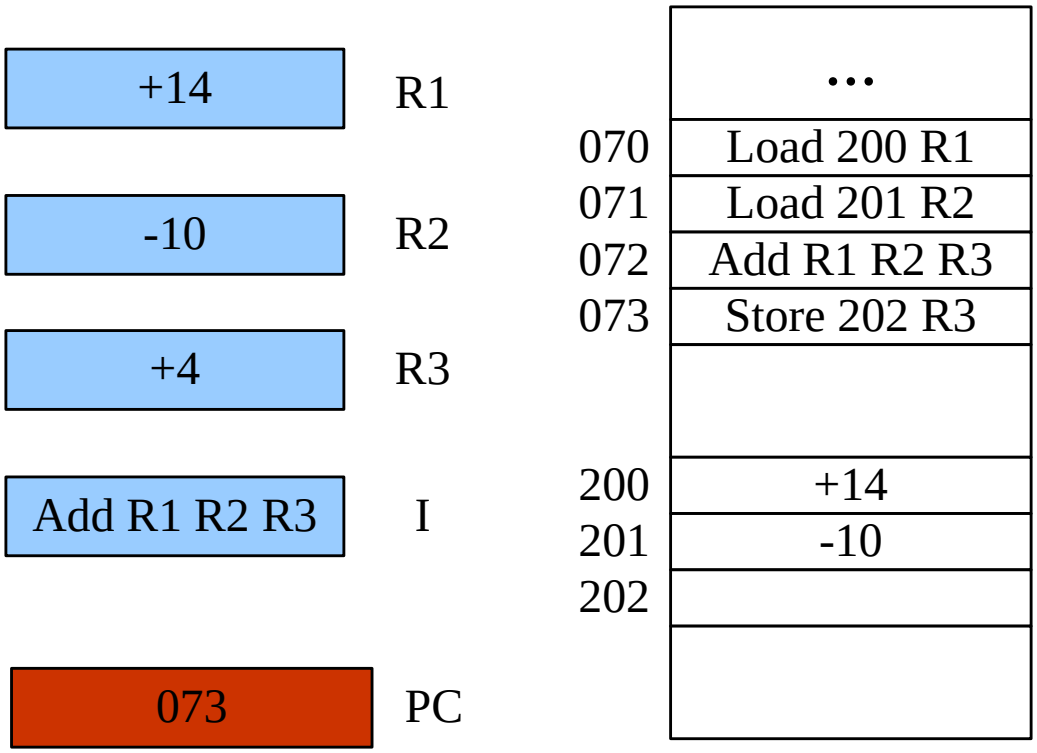
指令执行之前 内存和寄存器的状态


+14
R1
R2
R3
Load 200 R1
I
071
PC
…
070
Load 200 R1
071
Load 201 R2
072
Add R1 R2 R3
073
Store 202 R3
200
+14
201
-10
202

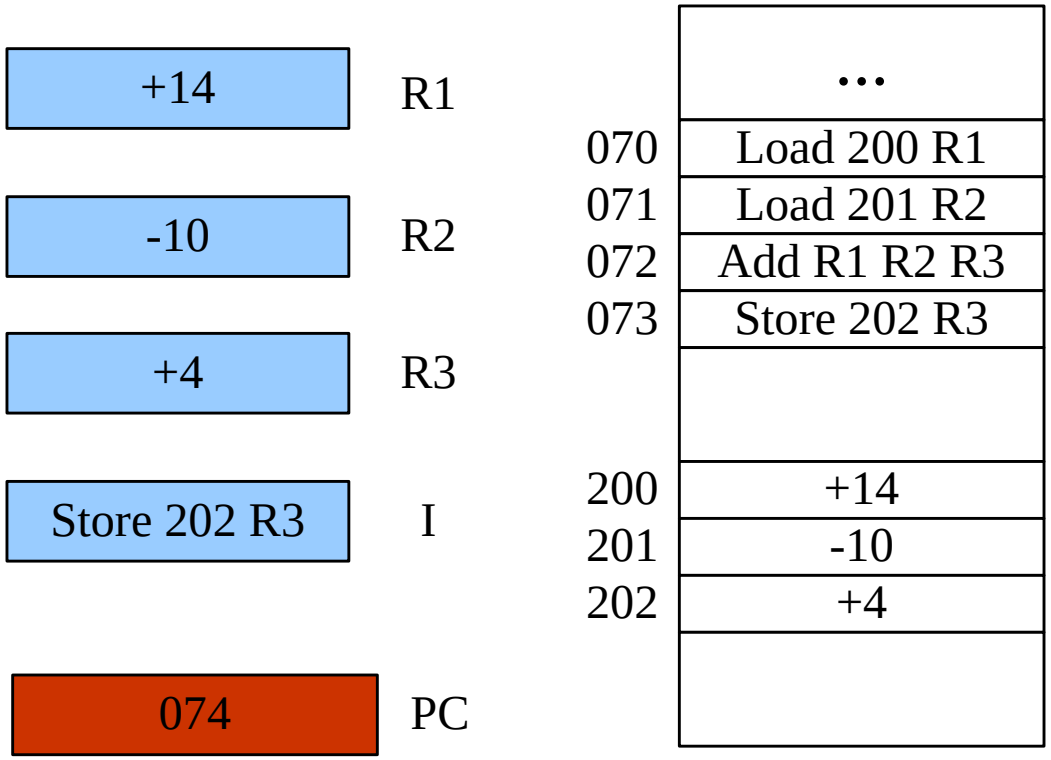
第一条指令执行后


+14
R1
-10
R2
R3
Load 201 R2
I
072
PC
...
070
Load 200 R1
071
Load 201 R2
072
Add R1 R2 R3
073
Store 202 R3
200
+14
201
-10
202

第二条指令执行后


+14
R1
-10
R2
+4
R3
Add R1 R2 R3
I
073
PC
...
070
Load 200 R1
071
Load 201 R2
072
Add R1 R2 R3
073
Store 202 R3
200
+14
201
-10
202

第三条指令执行后


+14
R1
-10
R2
+4
R3
Store 202 R3
I
074
PC
...
070
Load 200 R1
071
Load 201 R2
072
Add R1 R2 R3
073
Store 202 R3
200
+14
201
-10
202
+4

第四条指令执行后

指令执行

- 在上述例子中，4个指令周期完成加法
- 更为复杂的指令系统可以减少同样工作的指令周期数
  - 大多数现代处理器都包含能同时操作多个内存地址的指令
  - 这类指令的执行阶段会涉及多次访存

指令执行的并行性

- 采用流水线技术(pipelining)可以使得指令并行执行
- 本质上是将一条指令分成多个执行阶段，又称为微指令，从而充分发挥硬件的利用率
  - 1978-vintage Intel 8086需要3-10个时钟周期执行一条指令
  - 现在的处理器能达到每个时钟周期完成2-4条指令的速度
    - 每条指令都需要更长的时间完成，如20个时钟周期
    - 但同一时刻有多达100条指令在同时执行

I/O操作

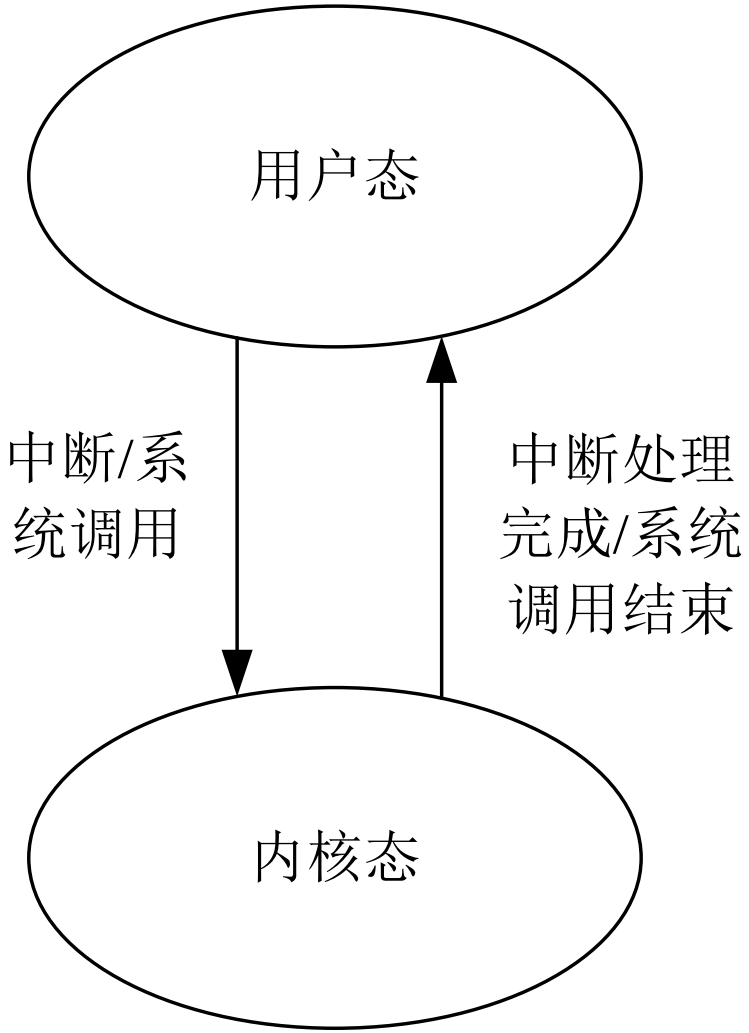
- 数据也可以直接在I/O控制器和CPU之间传输
  - I/O地址替代内存地址
  - I/O指令替代内存访问指令
- 数据也可以直接在I/O模块和内存之间进行
  - DMA
  - 由CPU授权I/O模块读写一个数据块

特权指令

- 操作系统核心程序可以使用全部机器指令，但用户程序只能使用机器指令系统的一个子集。
  - 用户执行有关资源管理的机器指令容易产生混乱，造成系统或用户信息的破坏。
- 指令分类：
  - **特权指令**：只能提供给操作系统核心程序使用的指令
    - 如启动输入输出设备，设置时钟，控制中断屏蔽位，清内存等
  - **非特权指令**：一般用户可以使用的指令

处理器状态

- 中央处理器依赖于处理器状态的标志确定当前运行的是操作系统还是一般用户程序。
- 处理器状态又称为处理器的运行模式
  - 系统状态
  - 用户状态
- **用户态向系统态转换的两类情况**
  - **执行系统调用**
  - **产生中断，执行中断处理程序**

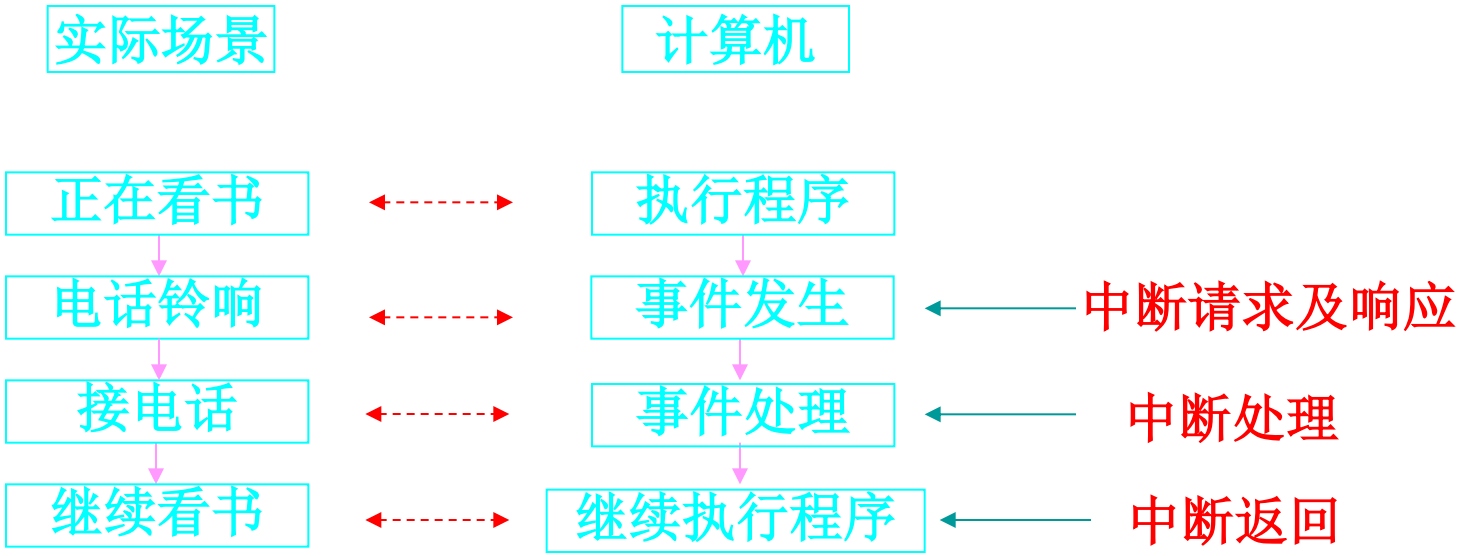

用户态
中断/系
统调用
中断处理
完成/系统
调用结束
内核态

提纲

- 处理器结构
- 指令执行
- 中断
- 存储器结构
- I/O通信技术

中断

- 中断的基本概念
- 什么是中断?
  - 与生活场景的比较



中断

- **中断的定义：**
  - 程序执行过程中，当出现急需处理的事件时，中止CPU上现行程序的运行，转而执行相应的事件处理程序，待处理结束后再*返回断点*或*调度其他程序执行*。
- **中断的作用：**
  - 最初作为设备向CPU报告I/O操作情况的一种手段，避免CPU不断轮询设备而耗费时间，解决了主机和设备的并行性问题。
  - 现在，中断技术应用范围越来越广，硬件故障、网络通信、人机交互和程序出错等，都以中断方式加以及时处理。可以说操作系统是*“中断驱动”*的。

中断的例子

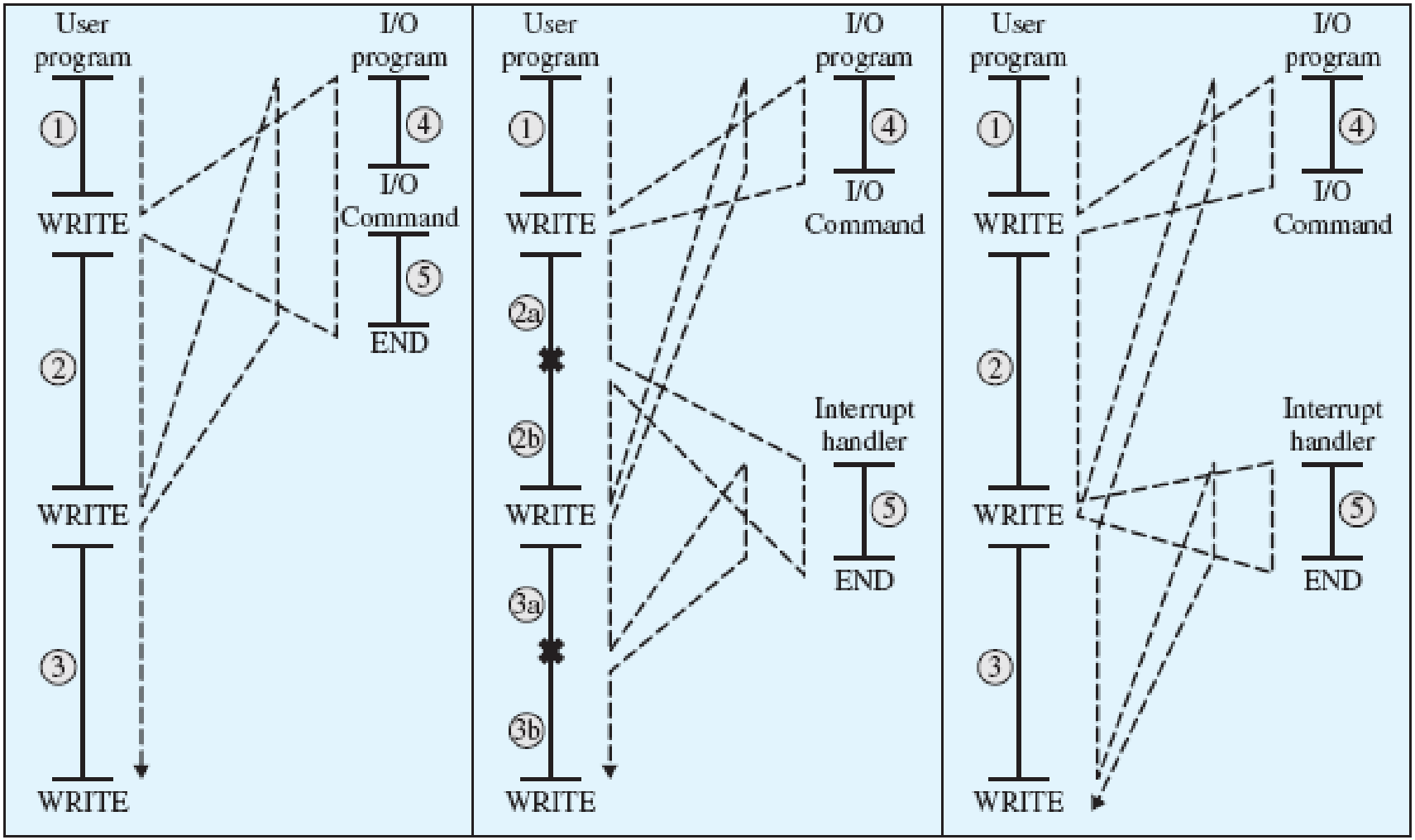
- 鼠标移动
- 按下键盘
- 电源掉电
- 打印机缺纸
- 打印完毕
- 定时器
- 磁盘读数据出错
- 内存缺页
- 除数为0
- 浮点溢出
- …

中断分类

- **程序性中断**
  - 由指令执行结果的某些条件产生
    - 算术操作溢出
    - 除数为0
    - 试图执行非法机器指令
    - 访问用户空间之外的地址
- **定时器中断**
  - 由处理器中的定时器产生
  - 允许操作系统以定时的方式执行某些任务
- **I/O中断**
  - 由I/O控制器产生
  - 用于报告I/O操作完成或操作错误
- **硬件错误**
  - 电源失效
  - 存储器奇偶校验错

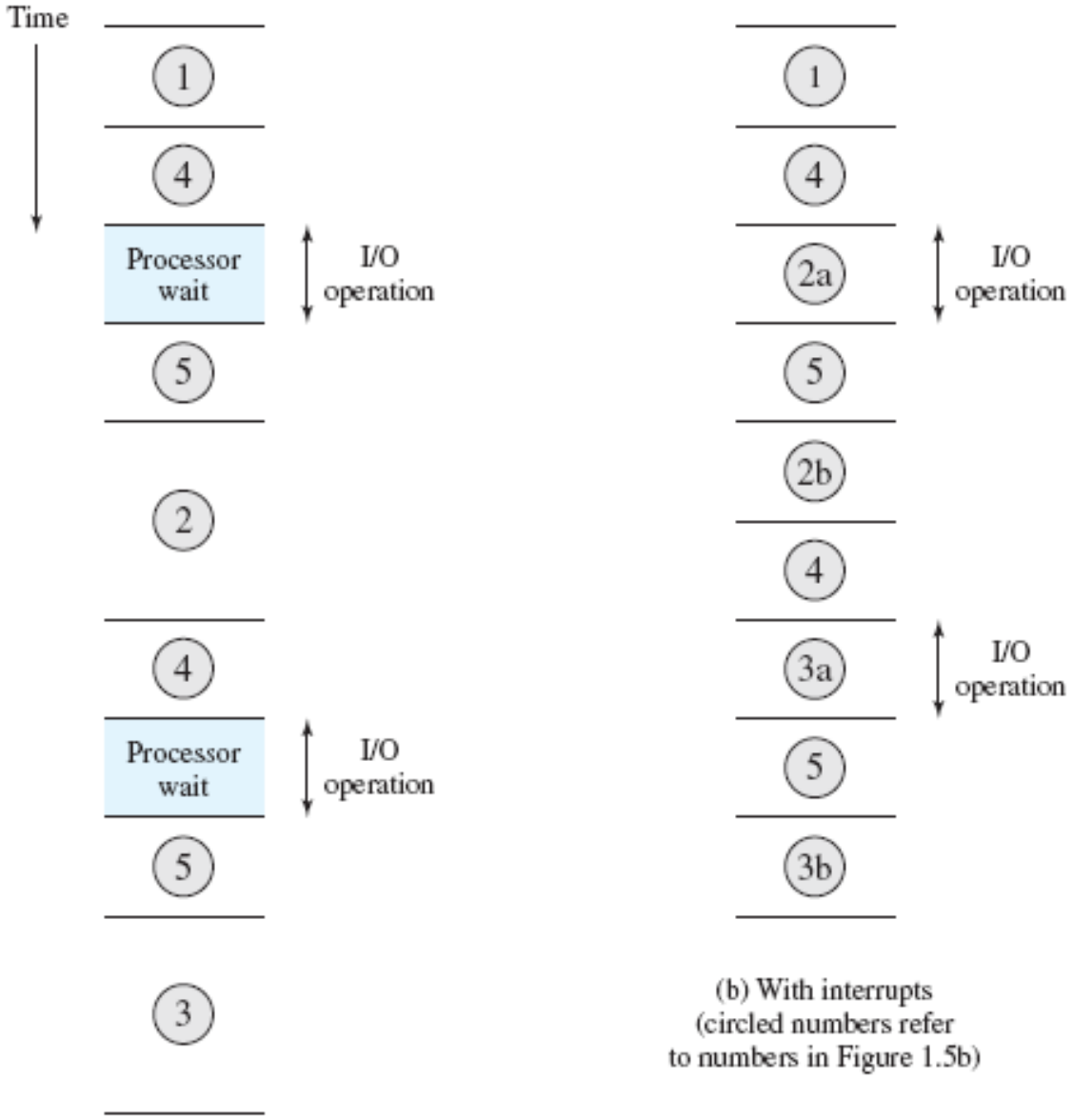
I/O程序的组成

- I/O程序由三部分组成
  - 准备指令
    - 一组用于准备实际I/O操作的指令，例如将用于输出的数据拷贝到一个特定的缓冲区，并准备设备命令的参数
  - 实际操作指令
    - 实际的I/O命令。若不采用中断，则程序必须等待I/O设备执行所请求的I/O功能（或是定期轮询I/O设备的状态）。
  - 结束指令
    - 一组结束I/O操作的指令。可能包括设置相应的标志位，表明操作的成功与否。

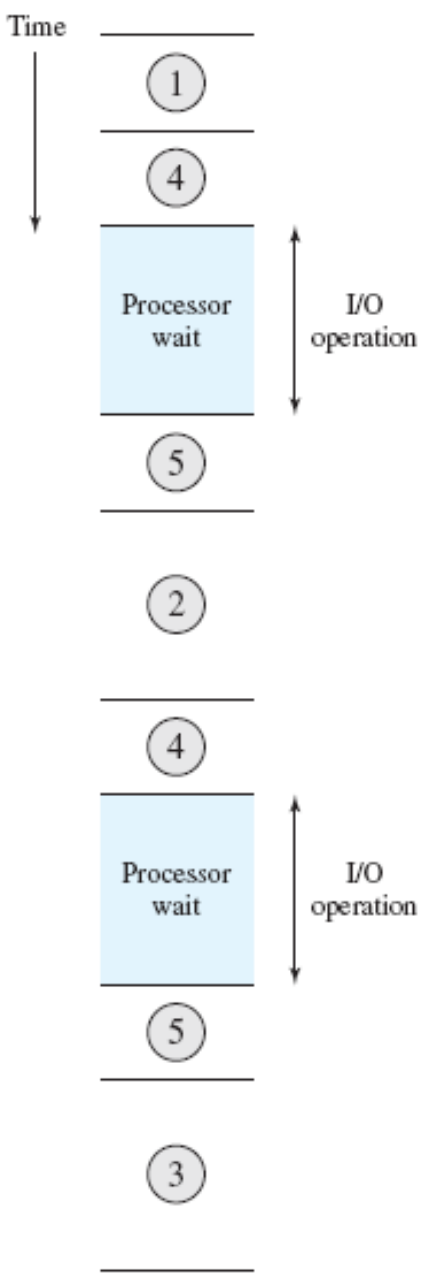


(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

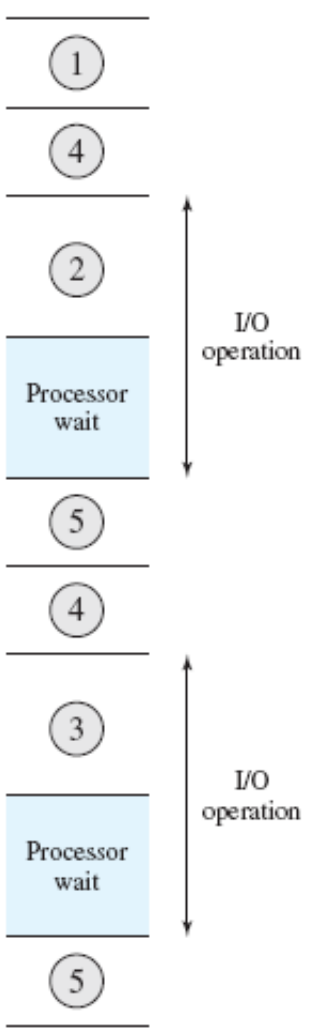
Figure 1.5 Program Flow of Control without and with Interrupts



I/O等待时间较短的程序执行时序图



无中断机制



采用中断

I/O等待时间较长的程序时序图

多道程序设计

- 当I/O等待时间较长时，即便采用中断，处理器也必须长时间等待
  - 如何充分发挥CPU的效率？
- 一个解决办法是允许多个用户程序同时运行
  - I/O密集型和CPU密集型平衡

中断周期

- 当外设准备好（例如数据传输完成），I/O模块将通过CPU的中断请求线向CPU发送中断请求信号。
- 处理器将暂停当前程序的操作，转向服务于特定I/O设备的服务例程，称为**中断处理程序**，并在结束处理后返回之前程序的执行或调度其他程序执行。
- 用户程序无需包含任何处理中断的代码，处理器和操作系统负责挂起用户程序并在相同点重启程序。

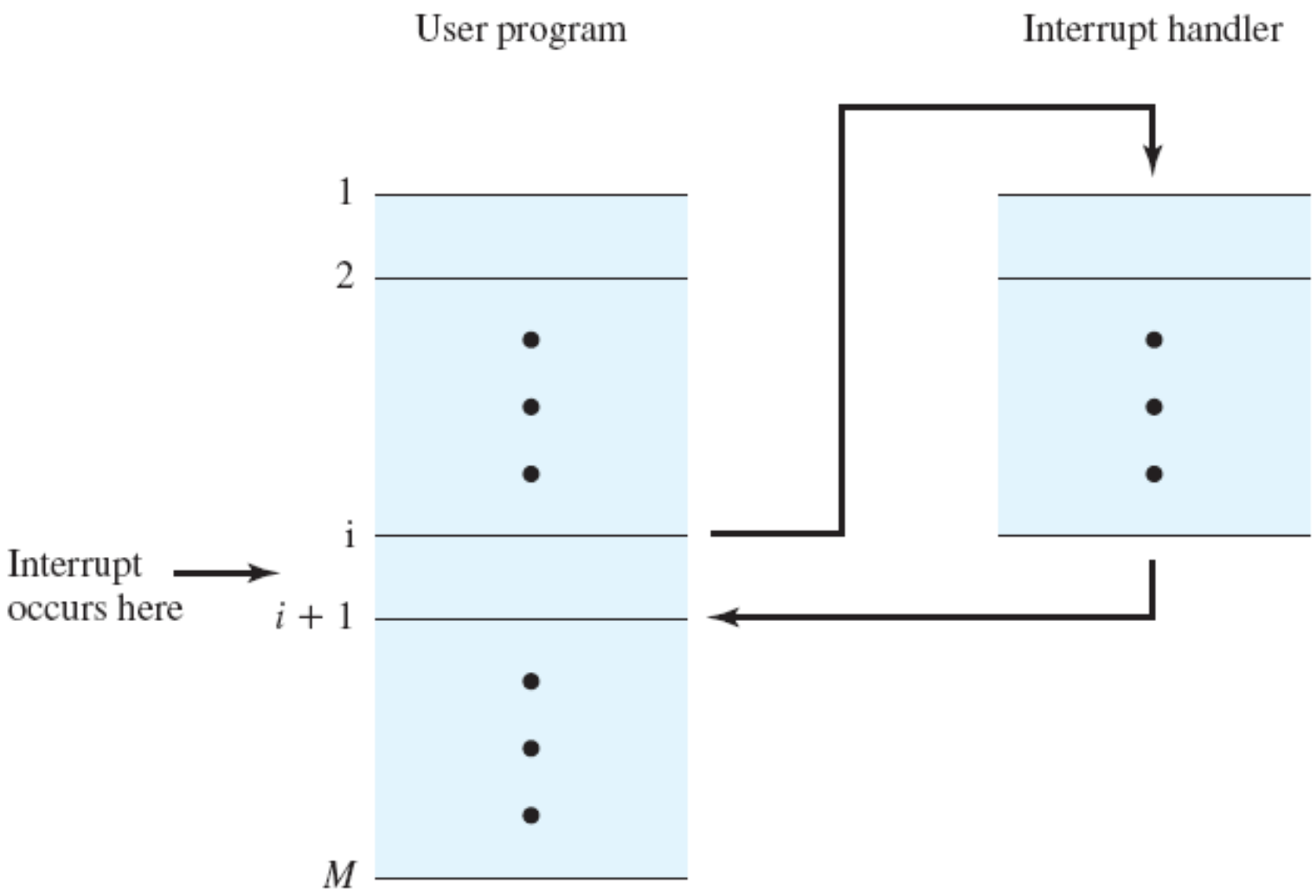


Figure 1.6 **Transfer of Control via Interrupts**

具有中断周期的指令周期

- 取指阶段
- 执行阶段
- 中断检测阶段
  - 在中断未被禁止时发生

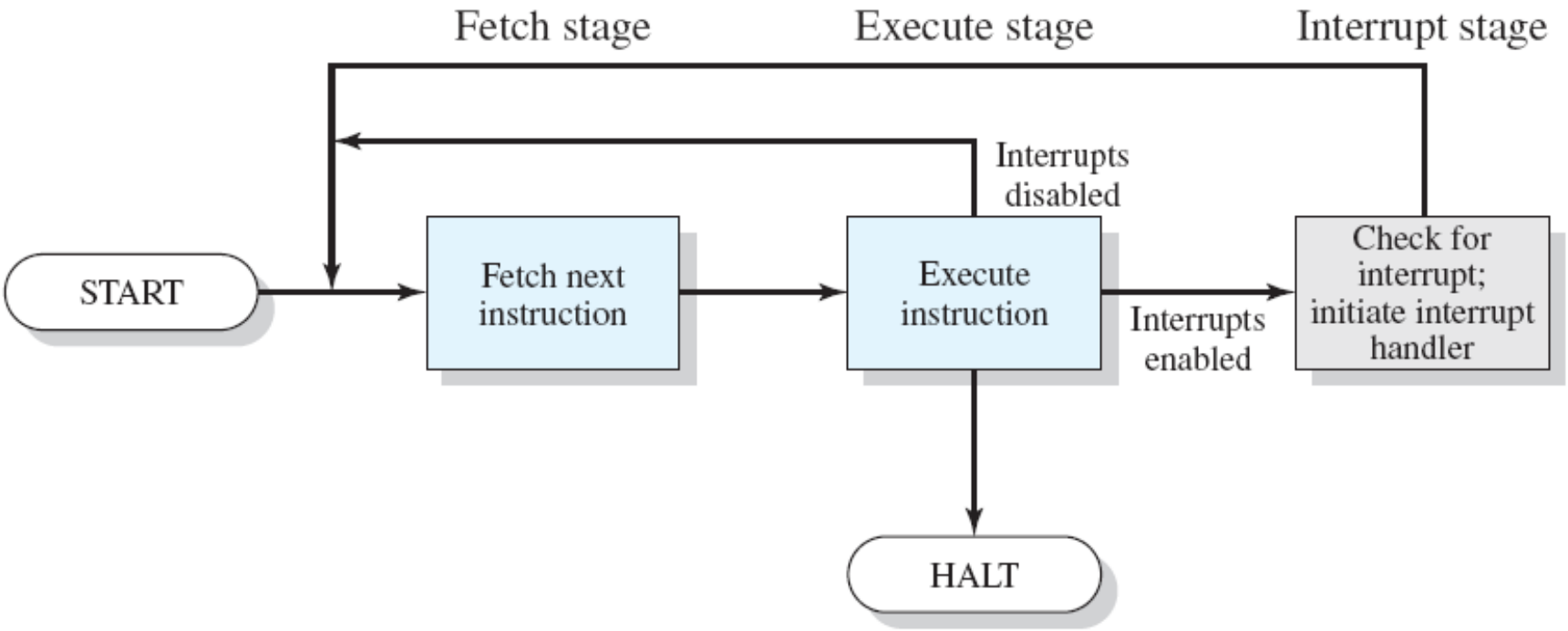
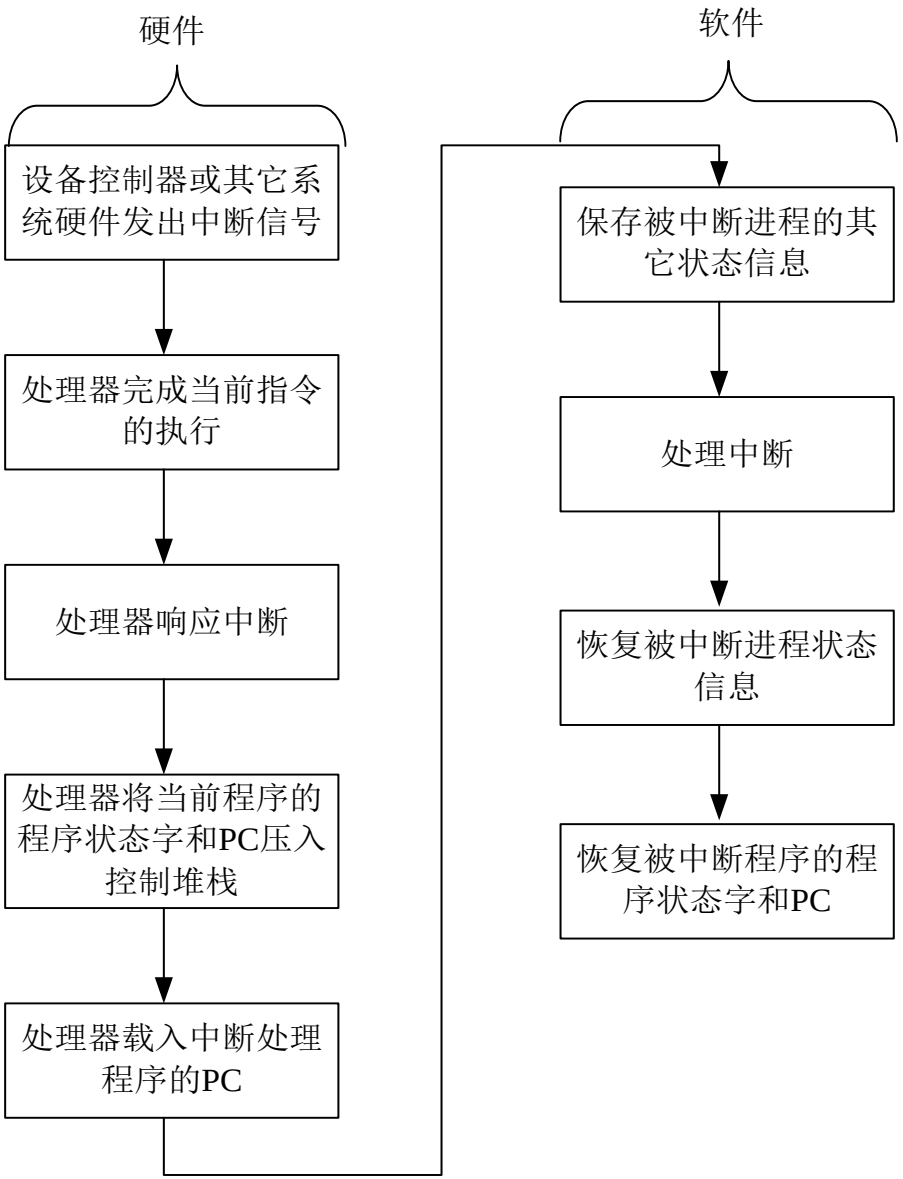
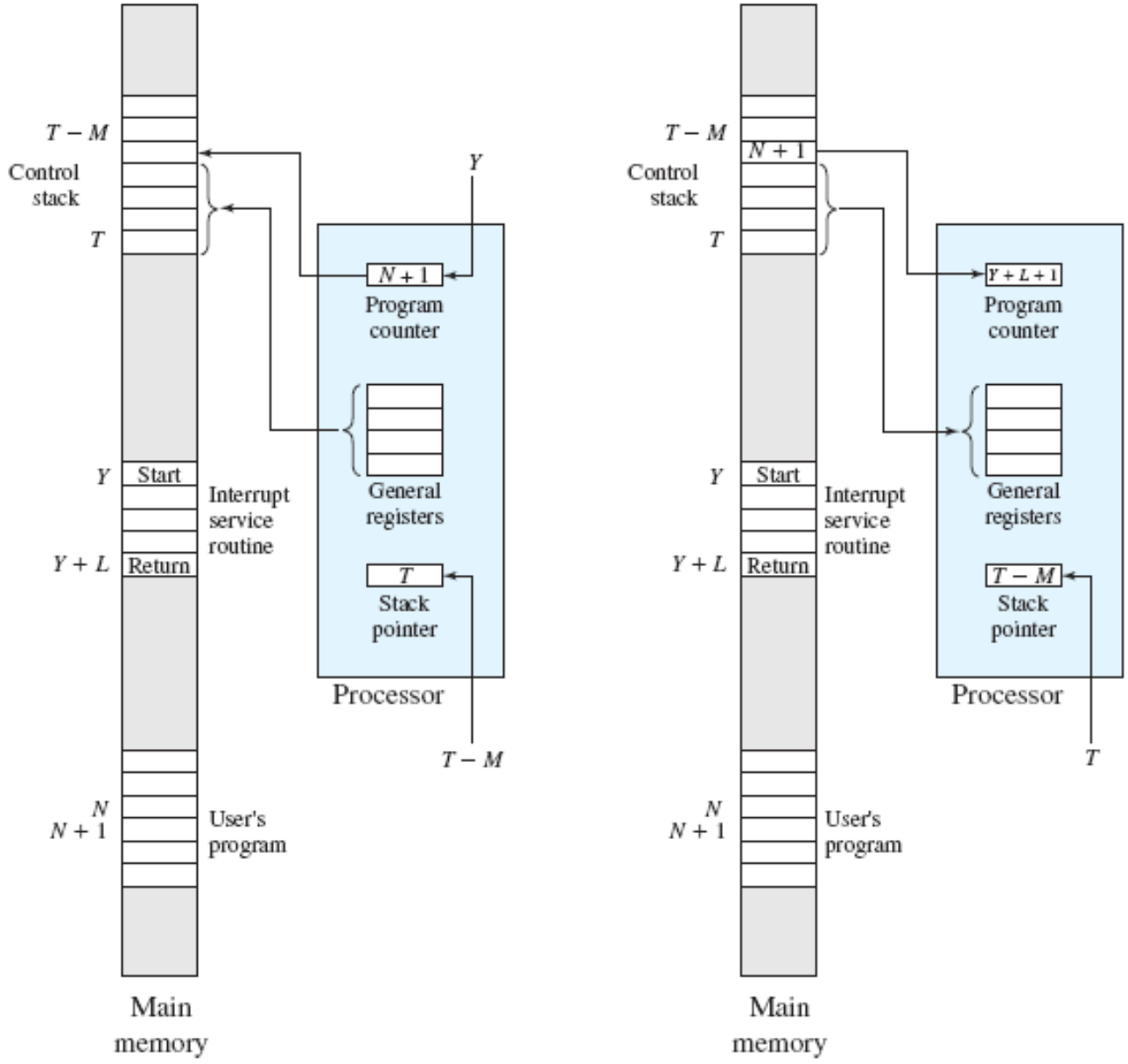


Figure 1.7 Instruction Cycle with Interrupts

中断处理过程


硬件
软件
设备控制器或其它系统硬件发出中断信号
处理器完成当前指令的执行
处理器响应中断
处理器将当前程序的程序状态字和PC压入控制堆栈
处理器载入中断处理程序的PC
保存被中断进程的其它状态信息
处理中断
恢复被中断进程状态信息
恢复被中断程序的程序状态字和PC



中断处理时内存和寄存器的变化情况

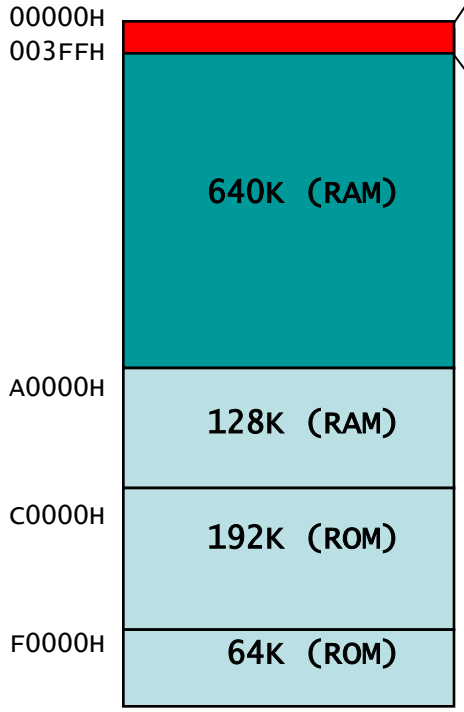
中断处理机制

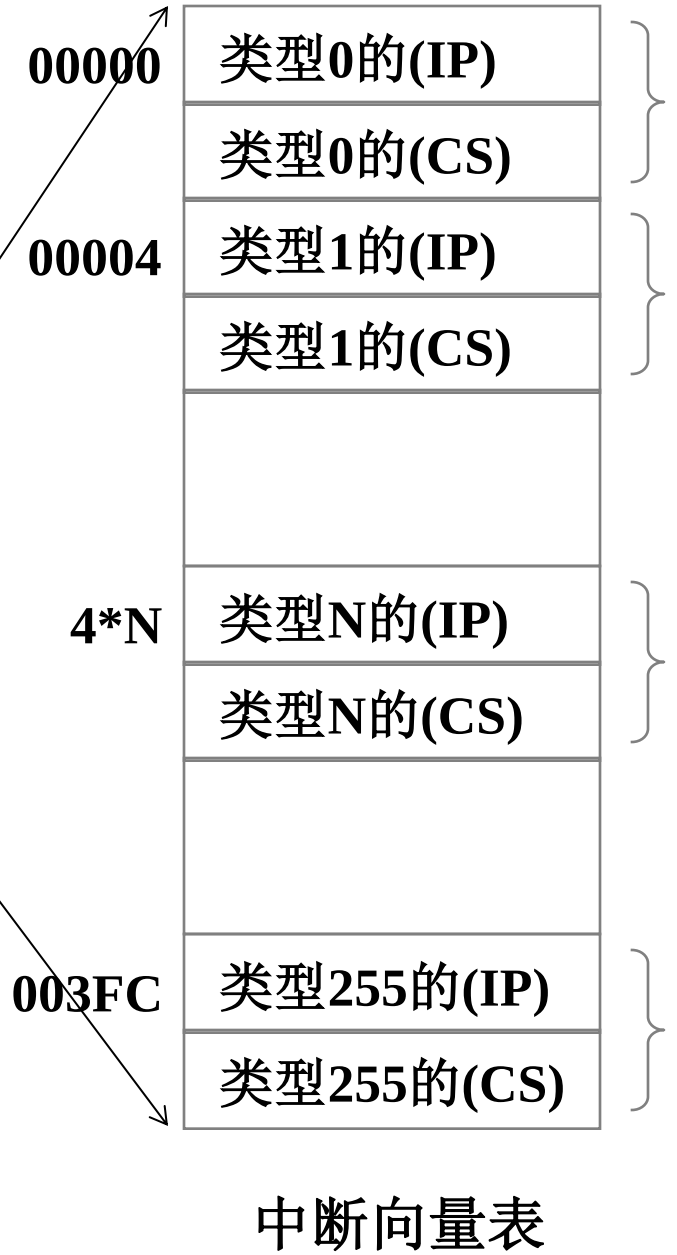
- 方法一：提供一个通用的例程，用于检查中断信息，进而调用与特定中断相关的处理程序。
- 方法二：中断数量有限，提供一张中断服务表加快中断处理。
  - 表中存放了中断服务例程的地址
  - 中断服务表存放在低内存（例如，前100的位置）。
  - 这些位置保存了不同的中断服务例程的地址。
  - 这个表也被称为**中断向量**。
  - 中断请求给出了一个中断号，用于查找中断向量表，从而找到中断服务例程的地址。

中断指令：INT N

中断向量：

中断例行程序的入口地址，
存放于中断向量区。





中断向量表

中断指令：　**INT　TYPE　或　INT**

执行操作：　**(SP) ← (SP) - 2**

((SP)+1,(SP)) ← (FLAGS)

(SP) ← (SP) - 2

((SP)+1,(SP)) ← (CS)

(SP) ← (SP) - 2

((SP)+1,(SP)) ← (IP)

(IP) ← (TYPE*4)

(CS) ← (TYPE*4+2)

从中断返回指令：IRET

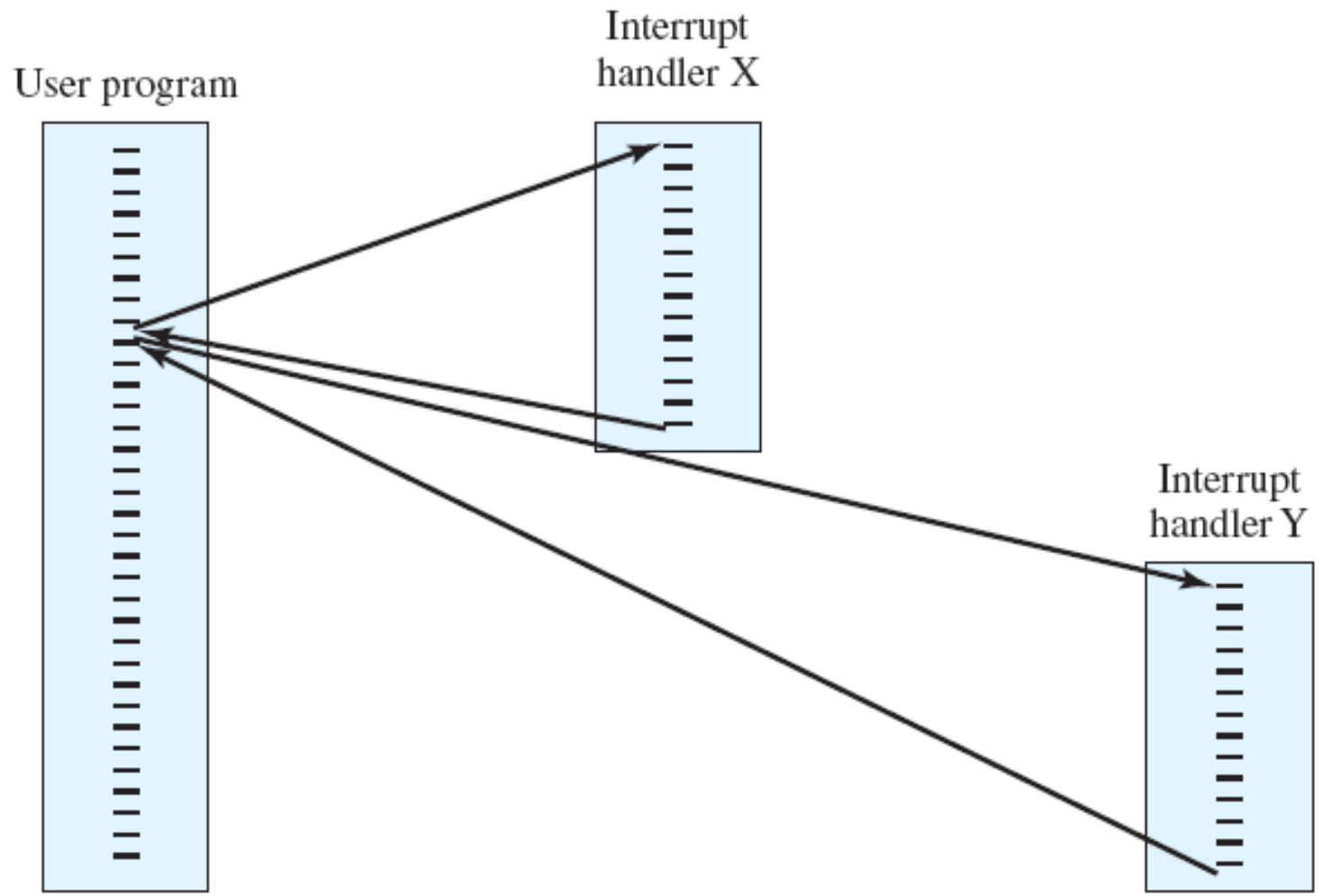
执行操作：
$$(IP) \leftarrow ((SP)+1,(SP))$$
$$(SP) \leftarrow (SP) + 2$$
$$(CS) \leftarrow ((SP)+1,(SP))$$
$$(SP) \leftarrow (SP) + 2$$
$$(FLAGS) \leftarrow ((SP)+1,(SP))$$
$$(SP) \leftarrow (SP) + 2$$

多重中断

- 计算机系统运行过程中，可能同时出现多个中断，或者前一个中断尚未处理完，又发生新的中断。
- 多重中断的处理方式
  - 禁中断
  - 中断优先级

禁中断

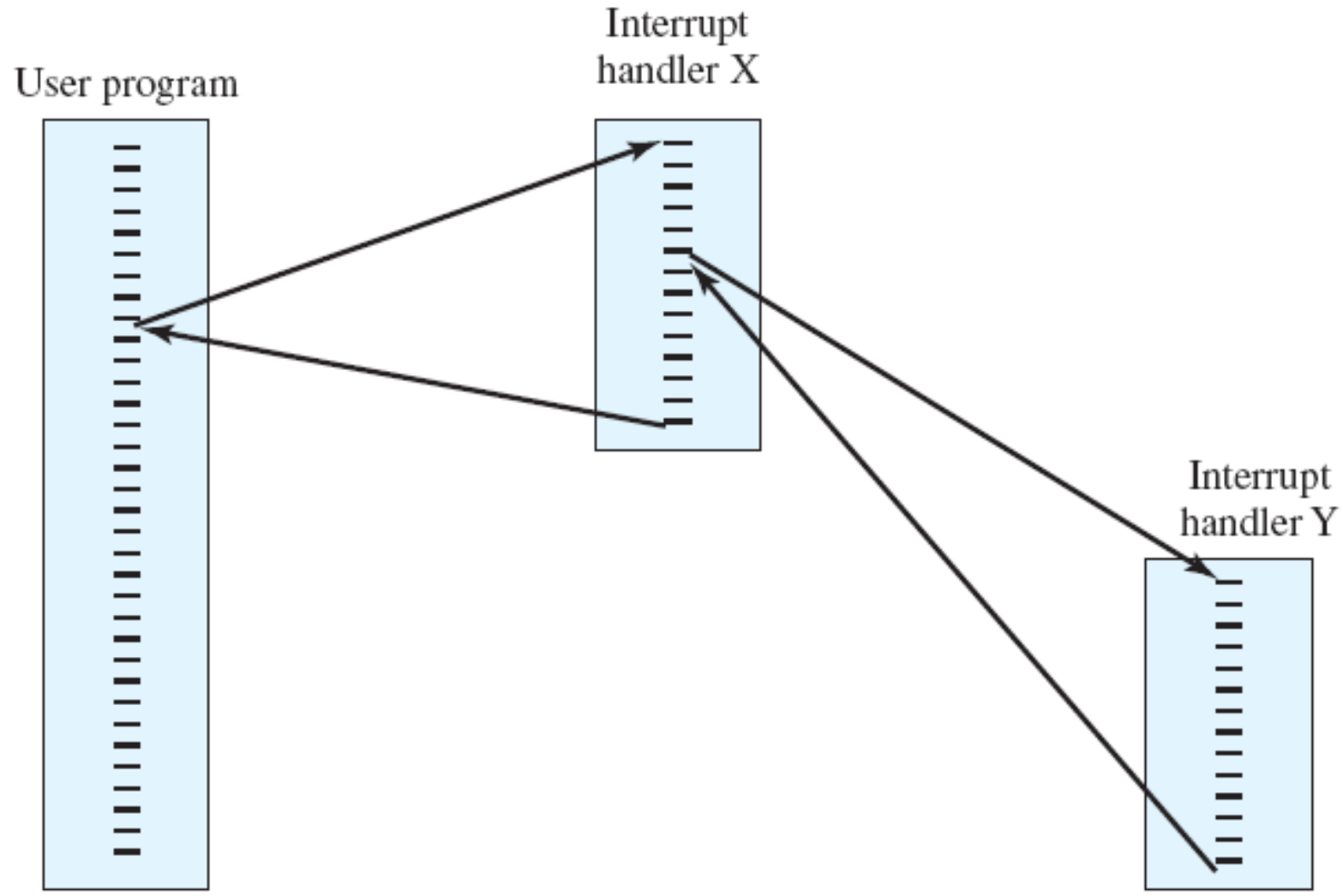
- 在一个中断执行期间，禁止响应其它中断，期间发生的中断信号将保持，待中断处理程序结束时得到服务
- 中断按发生的次序***线性执行***



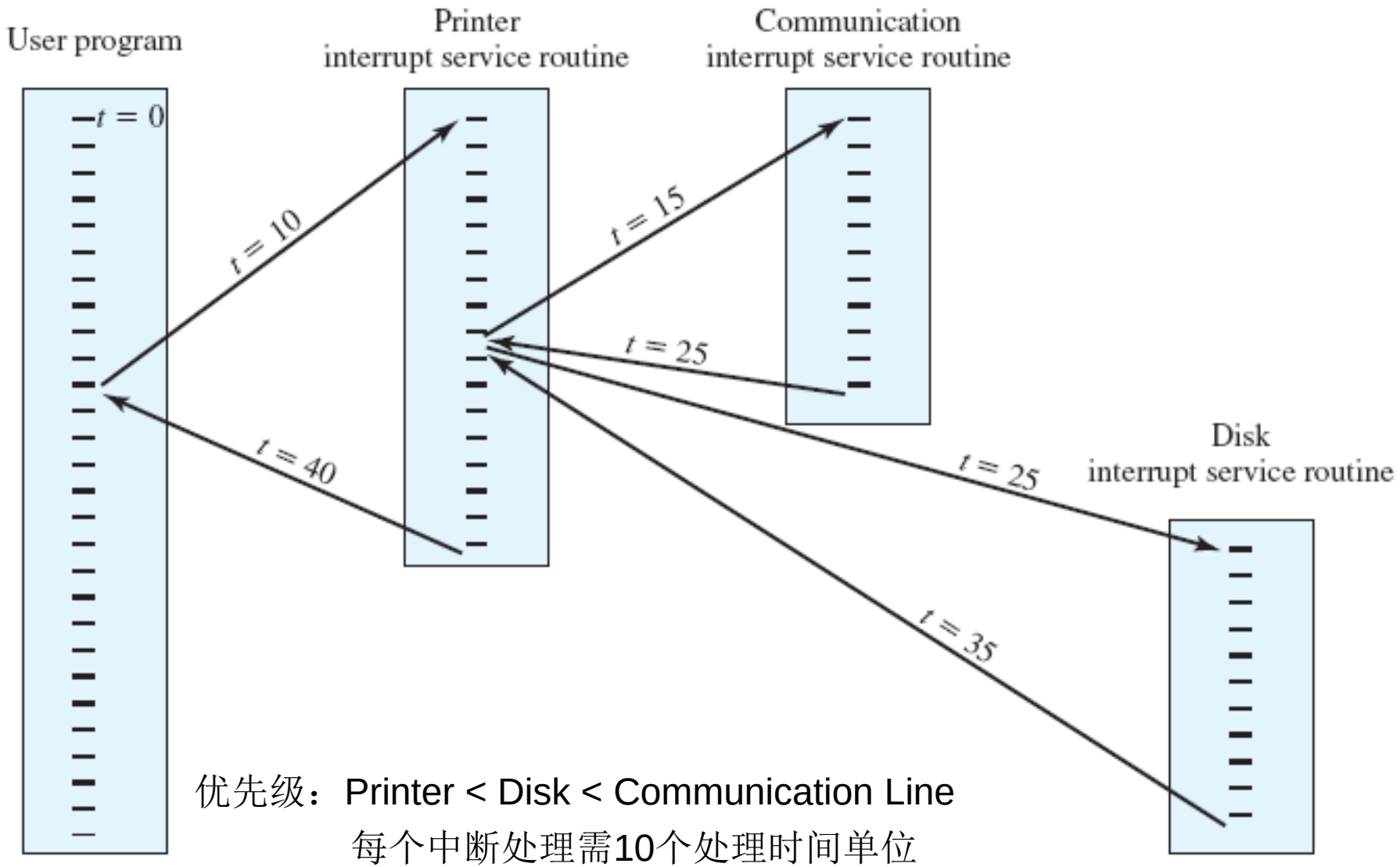
(a) Sequential interrupt processing

中断优先级

- 某些中断的延迟执行会导致数据丢失
  - 如数据不断填满I/O缓冲
- 对中断进行优先级分类，允许高优先级中断抢占低优先级中断的执行
- 中断以***嵌套方式执行***



(b) Nested interrupt processing



优先级：Printer < Disk < Communication Line

每个中断处理需10个处理时间单位

t=10, Printer中断

t=15, Communication Line中断

t=20, disk中断

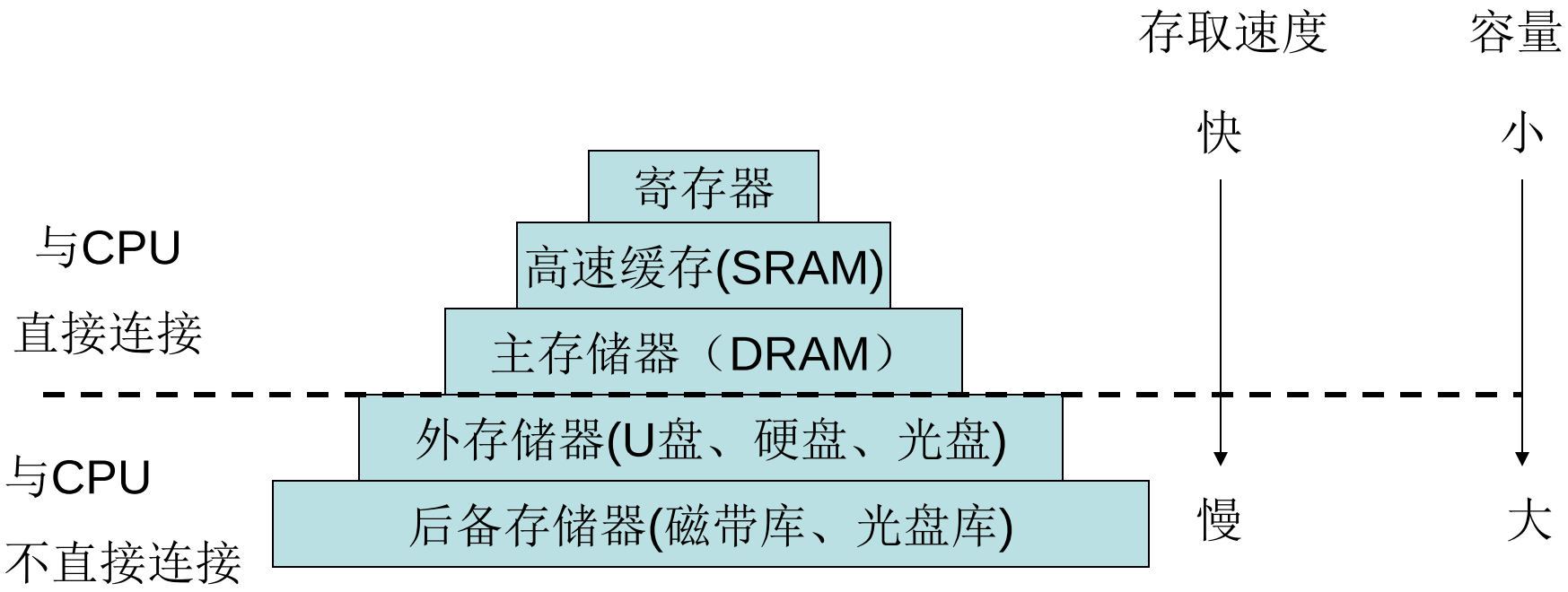
提纲

- 处理器结构
- 指令执行
- 中断
- 存储器结构
- I/O通信技术

存储器结构

- 计算机存储系统的设计因素
  - 容量
  - 访问速度
  - 价格
- 一般规律
  - 访问速度越快，每比特的代价越高
  - 容量越大， 每比特的代价越低
  - 容量越大， 访问速度越慢

存储器的层次结构



- 从上到下，单位存储的价格越来越低
- 从上到下，存储速度越来越慢

存储器层次化的特点

- 从上到下
  - 每个比特的价格越来越低
  - 容量越来越大
  - 访问速度越来越慢

  硬件技术决定
  - 被CPU访问的频率越来越低 → 程序访问局部性原理

各类存储技术的发展趋势

存储技术	指标	1980年	2010年
SRAM	$/MB	19200	60
	Access(ns)	300	1.5
DRAM	$/MB	8000	0.06
	Access(ns)	375	40
Rotating Disk	$/MB	500	0.0003
	Seek time(ms)	87	3
Intel CPU	Cycle time(ns)	1000	0.4
	Cores	1	4

主存储器和磁盘的访问速度在30年仅增长了9倍和30倍，但CPU的时钟频率增长了2500倍，速度差距在不断拉大！

- 例：
  - 两级存储器，第一级容量1000字节，访问时间0.1us, 第二级100，000字节，访问速度1us。CPU可以直接访问第一级存储器，但若被访问数据在第二级存储器，则需要首先将其传输到第一级存储器，然后才能由处理器访问。假设数据位于第一级存储器的概率为0.95，则平均访问时间为：

0.95*0.1us+0.05*(0.1us+1us)=0.15us

地址空间大小

- IA32：地址用32位来表示，最多访问4G的内存
- X86-64：扩展到64位，理论上可以访问2^{64}的内存空间，但实际上目前仅有48位用于地址(256T)

机器运行速度受制于内存

高速缓存

- 高速缓存对OS、程序员都不可见
- 引入高速缓存的原因
  - 每个指令周期，处理器至少访存一次（取指令），因此访存时间极大地限制了CPU执行指令的速度
  - CPU和存储器速度之间的不匹配，且差距越来越大
- 引入高速缓存的目的
  - 利用程序的局部性原理，使访问时间接近于高速缓存，而存储容量却是主存提供的容量

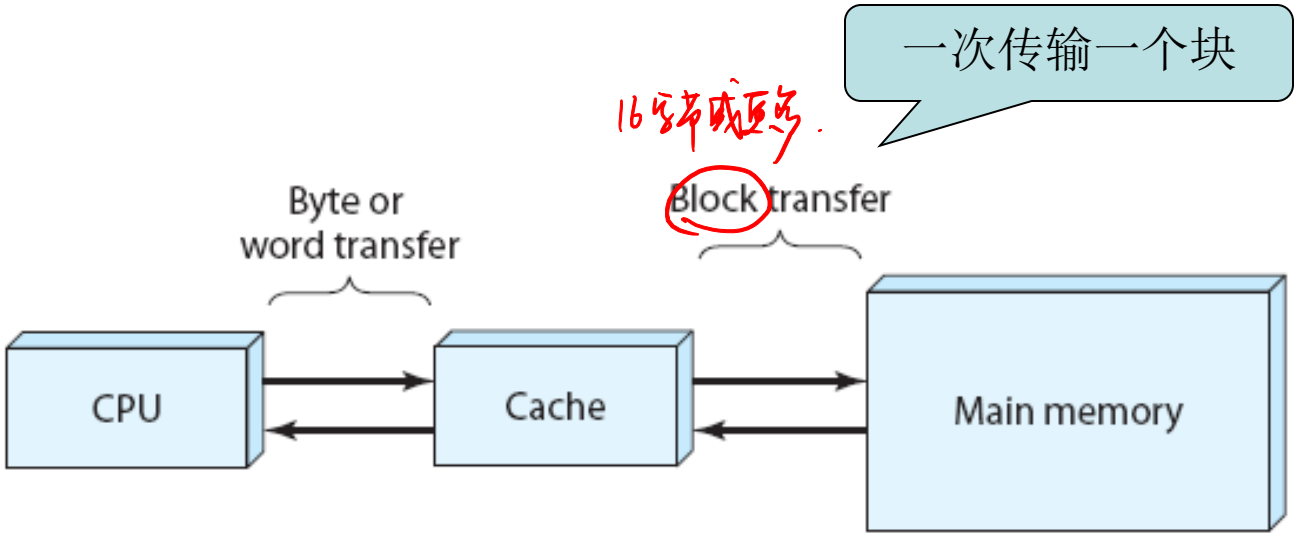


Figure 1.16 Cache and Main Memory

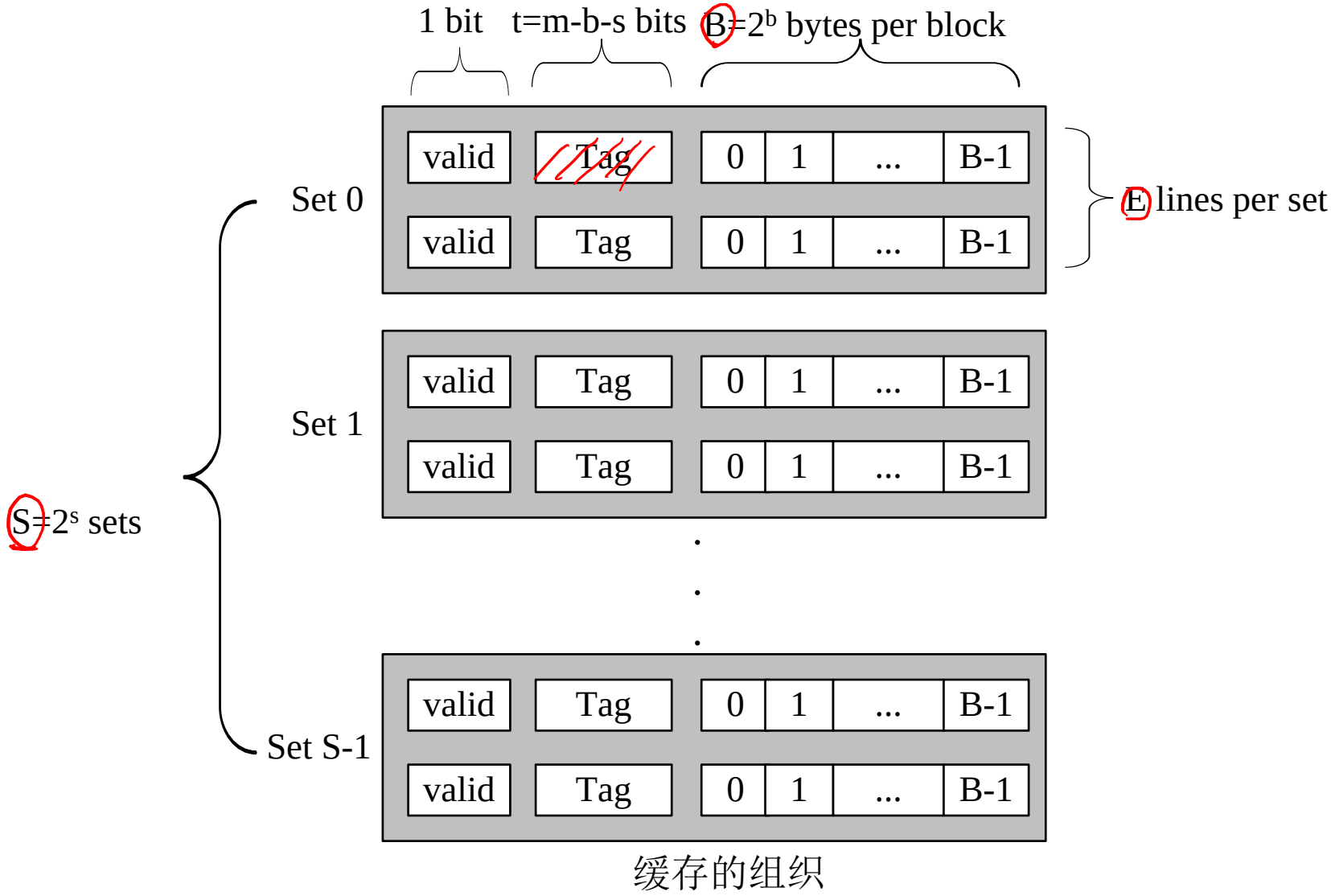
高速缓存

- 高速缓存包含主存内容的一小部分
- 每当处理器要读写一个字节/字，首先检查该字/字节是否位于高速缓存
- 如果位于高速缓存，则直接从高速缓存中取
- 否则，将一个***数据块(block)***从主存读入高速缓存，并将给定的字/字节传输给CPU

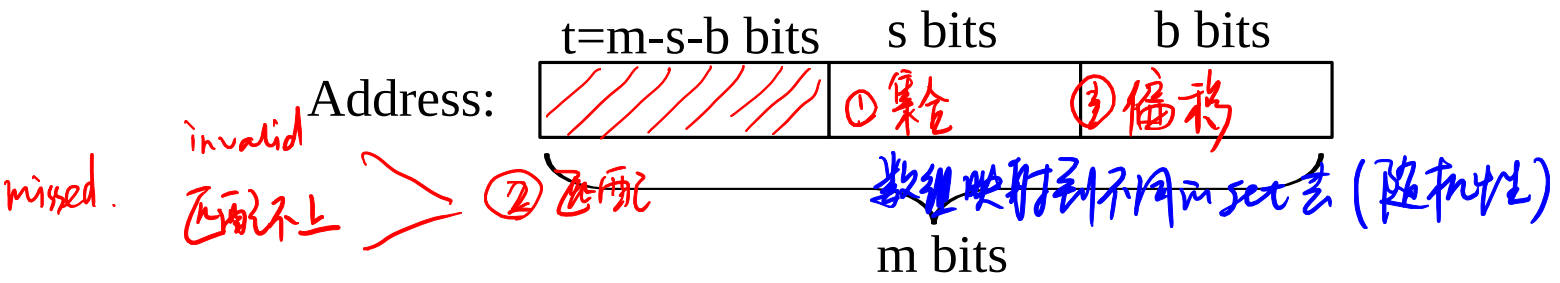
高速缓存的组织结构

- 假设内存地址长度为m位，即共有2^m个字节/字
- 缓存包含$S=2^s$个集合
- 每个集合中包含E个缓存块
  - 当E=1时，称为直接映射缓存
  - 当E>1时，称为相联缓存
  - 当只有一个集合时，称为全相联缓存
- 每个缓存块大小为$B=2^b$个字节/字

- 缓存总大小为S*E*B



缓存的组织



missed.
invalid
匹配不上
②匹配
数组映射到不同set去(随机性)

缓存的行为

- Set选择
  - 依据地址中间的s比特（为什么选择中间的s比特？）反之痛也
- Line匹配
  - 依据valid比特以及Tag字段和地址开头的t比特进行匹配
- 字节/字选择
  - 依据地址最后的b比特作为缓存block的偏移

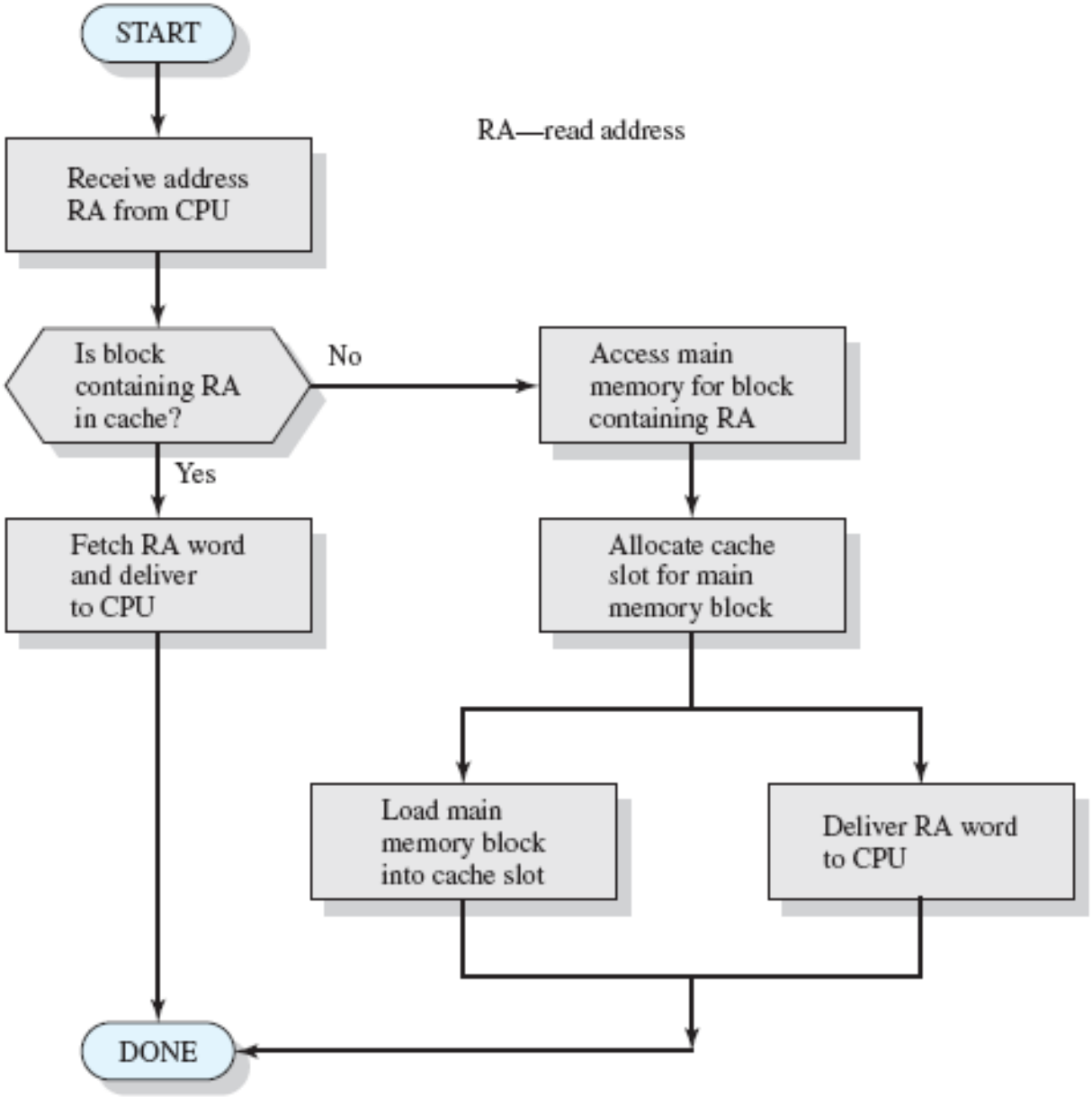


Figure 1.18 Cache Read Operation

缓存不命中的分类

- Cold misses
  - 系统初始化时，缓存为空，任何访问都会造成缓存不命中
- Conflict misses
  - 主存容量一般远远大于缓存容量
  - 不同的主存块映射到相同的缓存块，产生冲突，即便存在其它空闲块
- Capacity misses
  - 当程序循环所访问的程序和数据的大小（工作集）大于缓存空间的大小，会持续产生缓存不命中
  - 意味着缓存容量太小

如何处理写操作？

- 当缓存命中时，如何处理写操作？
  - write-through
    - 每当缓存中的内容被更新后，同时写低一级存储对应的内容
    - 写操作过于频繁
  - write-back
    - 仅当缓存块要被替换时，才将其写回低一级存储
    - 利用了locality, 大大降低了写低级存储的频度
    - 需要一个额外的bit标记缓存是否被更新过
- 当缓存不命中时，如何处理写操作？
  - write-allocate
    - 将要访问的数据首先调入缓存，然后再更新缓存中的值
  - no-write-allocate
    - 不将要改写的数据调入缓存，直接写低一级存储

利用访问局部性原理优化代码

- 程序执行期间，处理器对内存的访问，无论是指令或是数据，都倾向于成簇访问。
  - 时间局部性
  - 空间局部性

    - 注意编程时对表和数组的访问

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[3][0]	A[3][1]	A[3][2]	A[3][3]

数组按行存储，求数组所有元素的和，哪个效率高？

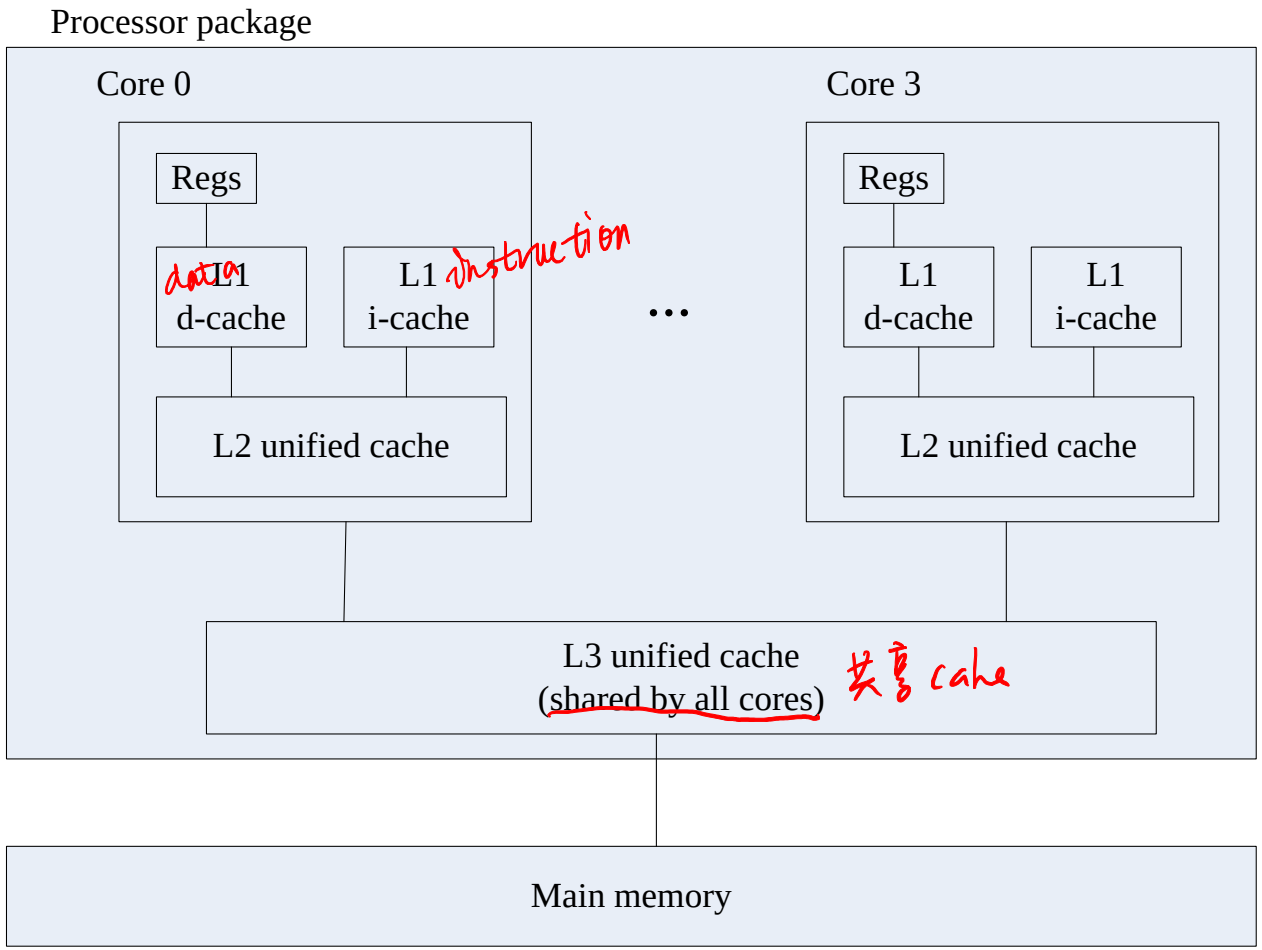
1、按行访问

```
for(int i=0;i<4;i++)
  for(int j = 0;j<4;j++)
     sum+=A[i][j];
```

2、按列访问

```
for(int j=0;j<4;j++)
   for(int i=0; j<4;i++)
      sum+=A[i][j];
```

缓存与多处理器



Intel Core i7组织结构

提纲

- 处理器结构
- 指令执行
- 中断
- 存储器结构
- I/O通信技术

I/O通信技术

- 可编程I/O
- 中断驱动I/O
- 直接内存访问(DMA)

可编程I/O

- CPU执行指令，向I/O模块发出I/O操作命令
- I/O模块执行操作，完成后将相应的I/O状态寄存器位置位
- CPU通过主动的方式来确定I/O操作是否完成，如定期检查I/O的状态位(POLL)

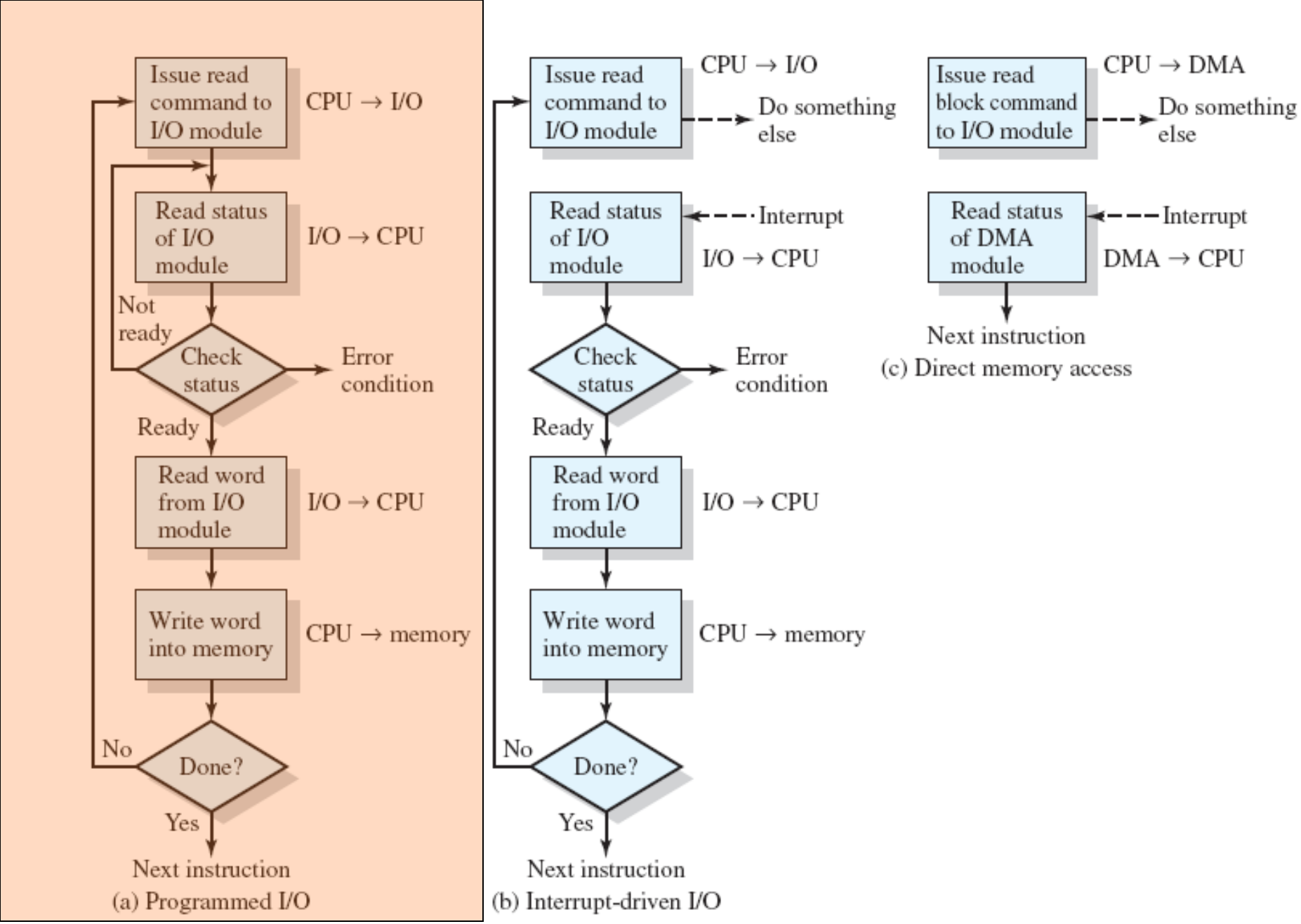


Figure 1.19 **Three Techniques for Input of a Block of Data**

中断驱动I/O

- CPU向I/O模块发出I/O操作命令，然后执行其它有用程序
- I/O模块完成操作后，向CPU发出中断信号，请求CPU服务。
- CPU暂停当前执行程序，转向中断处理，执行中断处理程序。
- 之后依据处理器调度算法选择程序继续执行。

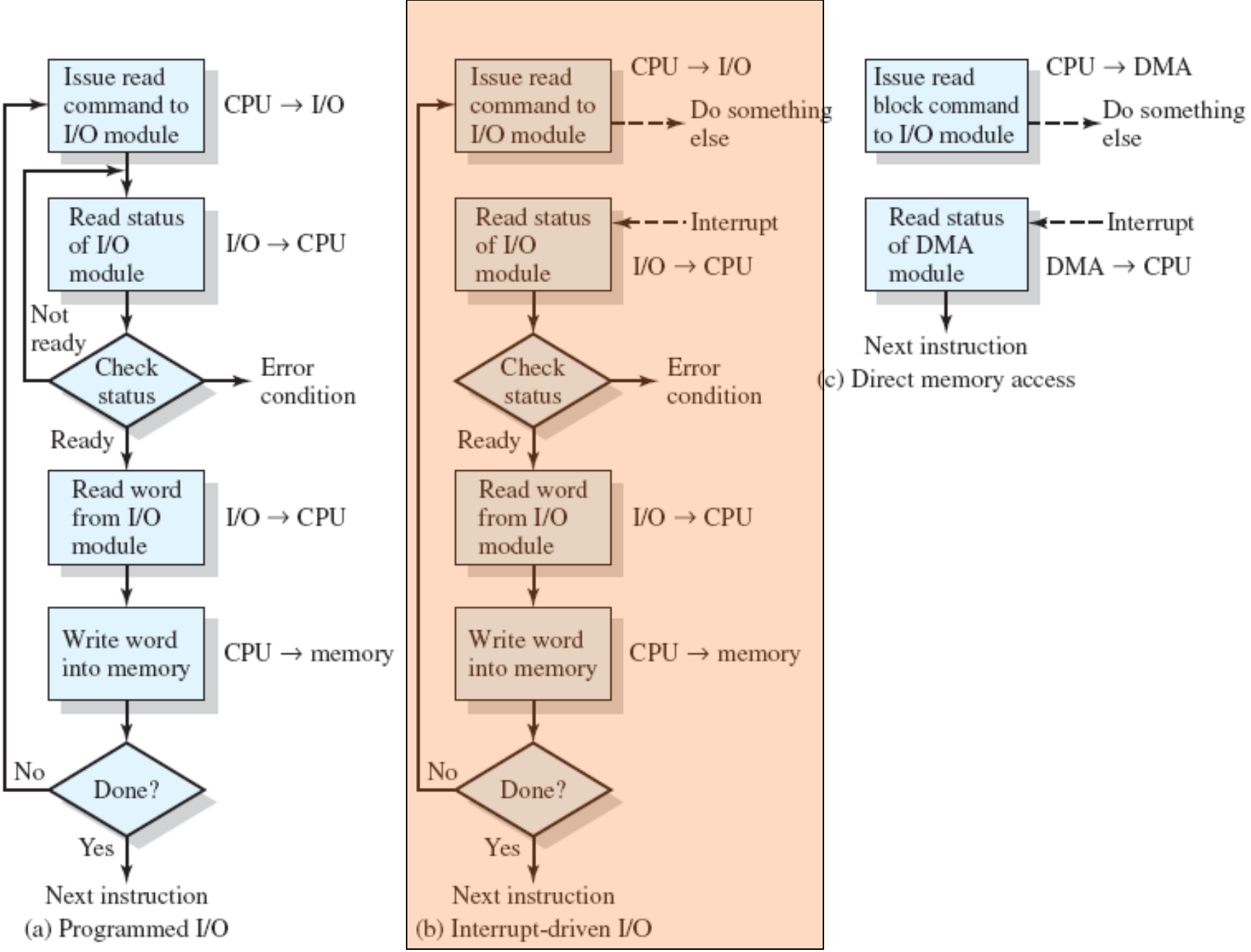


Figure 1.19 **Three Techniques for Input of a Block of Data**

中断驱动I/O

- 中断源识别（两种方式）
  - 多条中断信号线，每个I/O模块都在不同的中断信号线上发出中断信号
  - 单个中断信号线+设备地址线

DMA

- 中断I/O依然要求处理器干预I/O模块与内存之间的数据传输，不利于大量的数据传输。
- DMA功能可以作为一个单独的模块连接在总线上，也可以嵌入到I/O模块内。
- DMA的工作方式
  - 当CPU希望读或写一块数据时，向DMA模块发送命令，包括以下信息
    - 读命令/写命令
    - I/O设备的地址
    - 读/写 数据的内存起始地址
    - 读/写 数据的字数
  - 处理器然后执行其它工作，将该I/O操作交给DMA模块处理。
  - DMA模块完成整个数据块的传输，每次一个字，传输直接在内存和I/O缓冲之间进行，无需处理器参与。
  - 当传输完成后，DMA模块向处理器发送中断信号

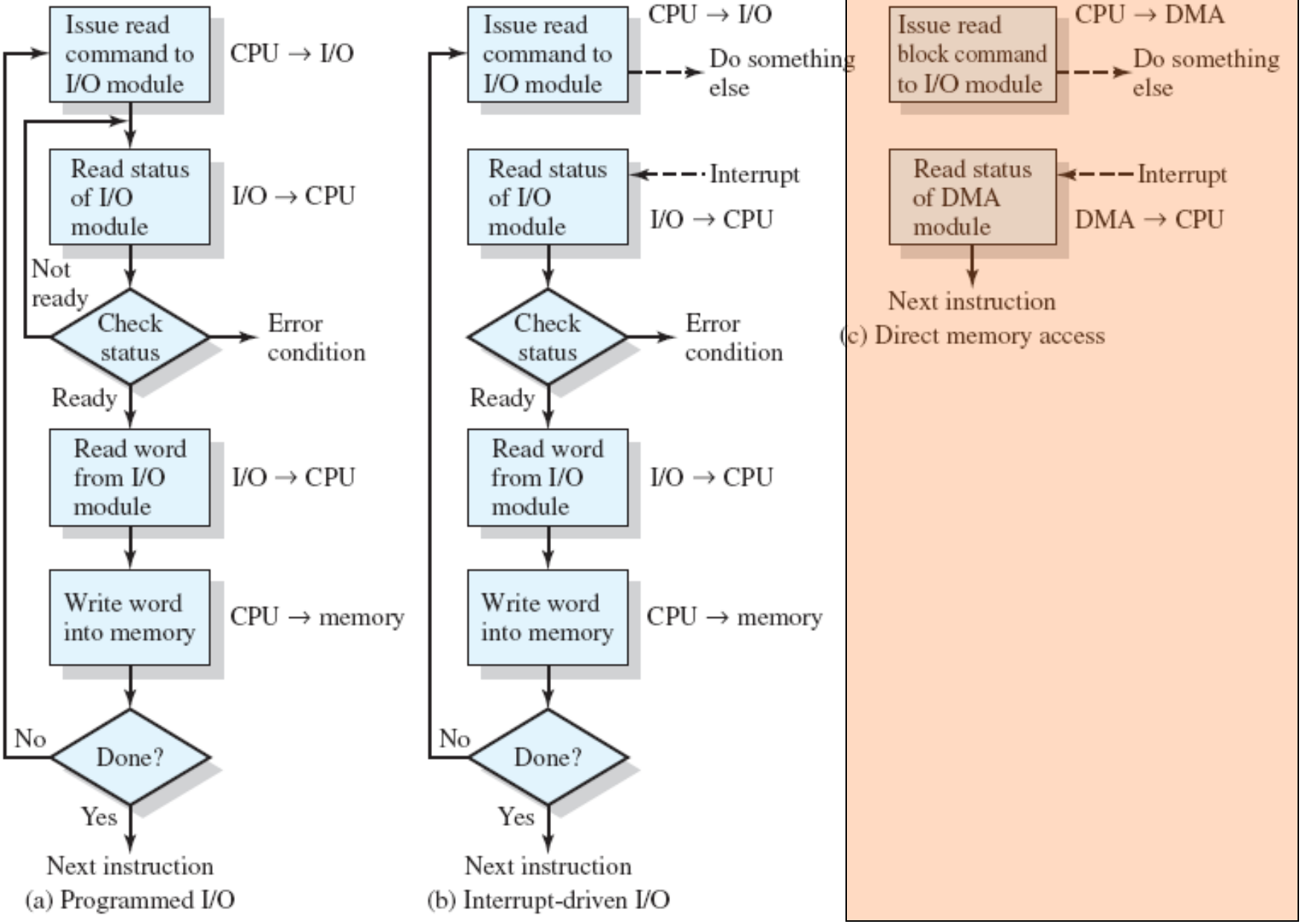


Figure 1.19 **Three Techniques for Input of a Block of Data**

DMA

- DMA模块和CPU模块会竞争总线的使用
  - 处理器和DMA传送不完全并行
  - 处理器希望占用总线时可能需要等待，从而减缓处理器执行速度
  - 不会引起中断
  - 不引起程序上下文的保存

计算机系统的分类

- 单指令流单数据流(SISD)
  - 一个处理器在一个存储器中的数据上执行单条指令流，如单核的PC机
- 单指令流多数据流(SIMD)
  - 单条指令流控制多个处理单元同时执行，每个处理单元包括处理器和相关的数据存储。例如，向量机。
- 多指令流单数据流(MISD)
  - 一个数据流被传送到一组处理器，每个处理器执行不同的指令操作
- 多指令流多数据流(MIMD)
  - 多个处理器对各自不同的数据集同时执行不同的指令流

MIMD分类

- 紧密耦合
  - 共享内存
  - 可进一步分为
    - **主从式系统(MSP)**：一个特定的处理器运行操作系统内核，其他处理器运行用户程序和操作系统例行程序。内核负责分配和调度各个处理器。
    - **对称式系统(SMP)**：操作系统内核可以运行在任意一个处理器上，每个处理器都可以自我调度运行的进程和线程
- 松散耦合
  - 每个处理单元都有一个独立的内存，如集群系统。