北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

# Computer Organization and Architecture

## Chapter 12

## Processor Structure and Function

School of Computer Science（National Pilot Software Engineering School）
AO XIONG (熊翱)
xiongao@bupt.edu.cn

# Preface

## We have learned:

- Overview
  - Basic Concepts and Computer Evolution 基本概念和计算机发展历史
  - Performance Issues 性能问题
- The computer system
  - Top level view of computer function and interconnection 计算机功能和互联结构顶层视图
  - Cache Memory cache存储器
  - Internal Memory 内部存储器
  - External Memory 外部存储器
  - Input& Output 输入输出
  - Operating System Support 操作系统支持

# Preface

## We have learned:

- Arithmetic and Logic 算术与逻辑
  - Computer arithmetic 计算机算术
- The central processing unit 中央处理器
  - Instruction sets: characteristics and function 指令集的特征和功能
  - Instruction Sets: Addressing Modes and Formats 指令集的寻址模式和格式
    - ✓ Addressing 寻址模式
    - ✓ x86 and ARM addressing modes
    - ✓ Instruction Formats 指令格式
    - ✓ x86 and ARM instruction formats
    - ✓ Assembler 汇编语言

# Preface

## We will focus the following contents today:

- Processor structure and function
  - What are the internal parts of CPU and how are the registers organized? CPU包括哪几个部分？CPU中寄存器是如何组织的？
  - How are computer instructions executed? 指令在CPU内部是如何执行的？
  - What strategies can be used to improve the efficiency of instruction execution? 有哪些措施可以提高指令执行的效率？

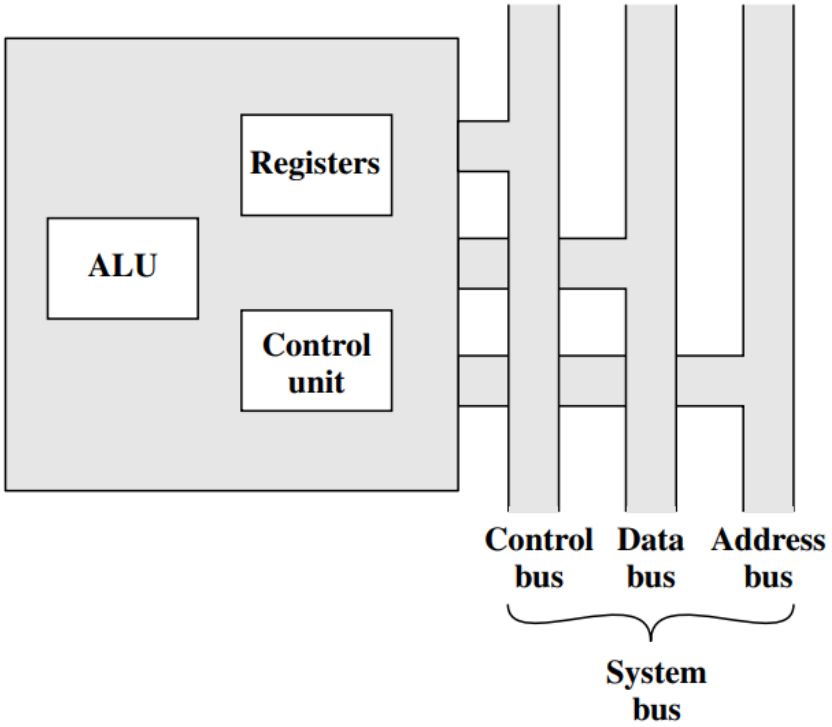
# Outline

- Processor Organization 处理器组成
- Register Organization 寄存器组成
- Instruction Cycle 指令周期
- Instruction Pipelining 指令流水线

# Functions of the processor 处理器的职能

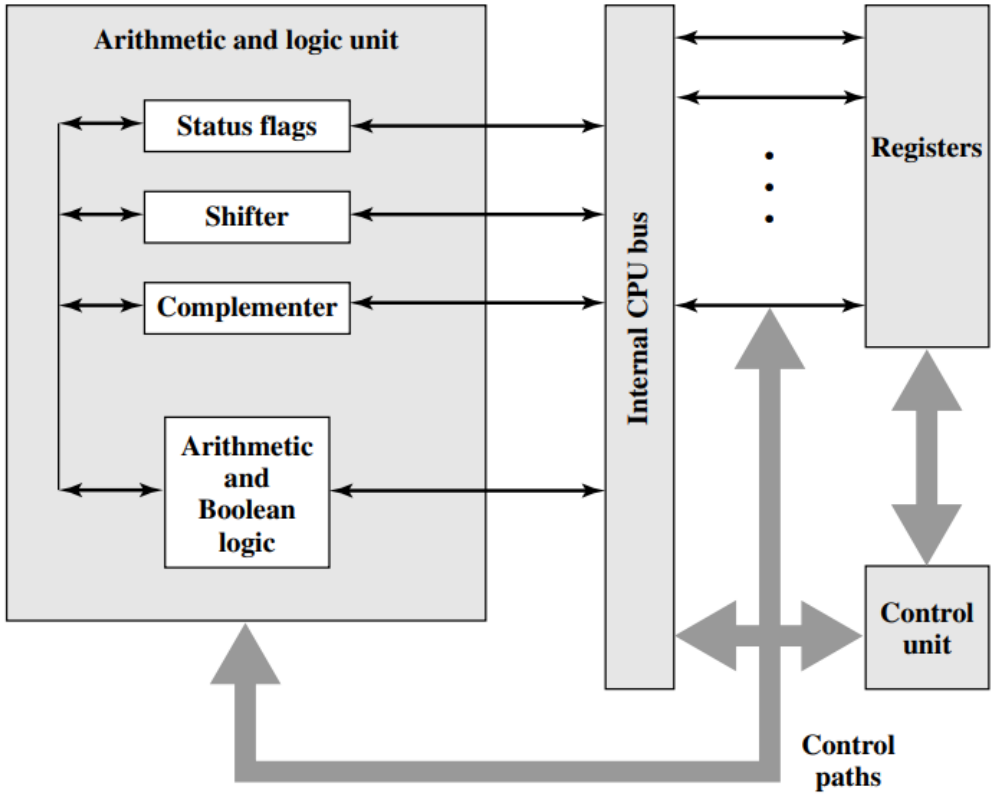
- A CPU must be able to CPU应该能够完成如下工作
  - Fetch instruction from memory 从内存中取指令
  - Decode the instruction to determine what action to do 指令解码，决定做什么操作
  - Fetch data 取操作数
  - Process data 处理操作数
  - Write data 写结果

# Processor organization 处理器组织




- CPU必须要能够暂时保存一些数据，以对数据进行处理
- CPU需要记住下一个指令的位置，这样才能在当前指令执行完成之后，能到找到下一个指令
- 处理过程中需要能够保存指令和数据
- CPU包括ALU，CU，还需要有一组存储部件——寄存器
- CPU通过一组系统总线和计算机的其他部件进行连接。系统总线包括控制总线、数据总线和地址总线

# CPU internal structure CPU内部结构




- CPU的内部总线把ALU、寄存器和CU连在一起，完成数据在寄存器和ALU的传送
- 控制单元对寄存器、内部总线和ALU进行控制，控制各个部件按照指令要求完成相应的处理
- 在ALU内部，还包括各种更小的组件，例如状态标志，移位器，求补器，以及算术和布尔逻辑等

# Processor organization 处理器组成

- ALU
  - Perform arithmetic and logical operations 进行算术和逻辑运算
  - In addition to arithmetic and logical units, it also includes some components such as status flag, shifter, etc. 除了算术和逻辑运算单元之外，还包括一些组件，如状态标志位，移位器等
- Register
  - Used to temporarily save instructions and data 用于暂时保存指令和数据
- CU
  - Control components to complete operations according to instructions 控制各个部件按照指令要求完成操作
- Internal CPU Bus
  - Connection channels of various components inside the CPU CPU内部各个部件的连接通道

# Outline

- Processor Organization 处理器组成
- Register Organization 寄存器组成
- Instruction Cycle 指令周期
- Instruction Pipelining 指令流水线

# Registers 寄存器

- CPU must have some working space (temporary storage) CPU需要有部件来临时存储数据
  - Called registers 称为寄存器
- Number and function vary between processor designs 处理器的设计不同，数量和功能也不同
- One of the major design decisions 处理器设计的重要考虑
- Top level of memory hierarchy 分级存储体系的顶层

# Registers organization 寄存器组织

Registers in the CPU Including two types CPU中的寄存器包括两类

- User-visible registers 用户可见的寄存器
  - Used by programmers 程序员编程使用
  - Reduce access to main memory and improve instruction processing efficiency by optimizing the use of registers 通过优化寄存器的使用，减少对主存的访问，提高指令处理效率
- Control and status registers 控制和状态寄存器
  - Used by control unit 由控制单元使用
  - Control the operation of the CPU and the execution of the program by the privileged operating system 控制CPU的操作，并由拥有特权的操作系统来控制程序的执行

# User-visible registers 用户可见寄存器

- User visible registers can be divided into four categories according to its purpose 用户可见寄存器根据用途，可以分为四类
  - General purpose： assigned to various purposes 通用寄存器：指派各种用途
  - Data： for data retention only 数据寄存器：仅用于保持数据
  - Address： used for some addressing mode 地址寄存器：用于某种寻址方式
  - Condition codes： also called flag register, it stores some flags of operation results 条件码寄存器，也称为标志寄存器，保存操作结果的一些标志

# General purpose registers 通用寄存器

- True general purpose 真正意义的通用寄存器
  - Registers and opcodes are orthogonal in the instruction set 寄存器和操作码在指令集中是正交的
  - Registers can be arbitrarily matched with opcodes 寄存器可以和操作码进行任意搭配
- Restricted general registers 受限制的通用寄存器
  - Specially used for floating point number or stack operation 专门用于浮点数或栈操作
- In some cases, general registers can be used for addressing 有些情况下通用寄存器可以用作寻址
  - Register indirect addressing寄存器间接寻址
  - Displacement addressing偏移寻址

# Data and address registers 数据和地址寄存器

- Data register 数据寄存器
  - It can only be used for store data, not for addressing 只能用于保持数据，不能用于寻址
  - Accumulator 累加器
- Addressing register 地址寄存器
  - Used for a specific addressing mode 用于某种具体的寻址方式
  - Segment pointer 段指针
  - Index register 变址寄存器
  - Stack pointer 栈指针

# Data and address registers 寄存器设计问题

- General or specialized 通用寄存器还是专用寄存器?
- How many registers? 多少个寄存器?
- How many bits are the register 寄存器的位数多大?

# General or Specialized 通用/专用?

- Whether registers are general or special affects the design of instruction sets 寄存器是通用的，还是专用的，影响指令集的设计
- Specialized 专用
  - Opcode can implicitly use a register group or a register 操作码中可以隐含使用寄存器组或某个寄存器
  - Smaller instructions 指令更小
  - Less flexibility 灵活性降低
- General purpose 通用
  - Increase flexibility and programmer options 提升了灵活性，编程者有更多的选择
  - Increase instruction size & complexity 增加了指令长度和复杂度

# How many? 多少个通用寄存器

- More registers require more bits to specify registers in instructions 寄存器数量多，指令中确定寄存器需要的位数多
- Fewer registers require more memory access 寄存器数量少，需要更多的存储器访问
- Too many registers does not reduce memory references remarkably and takes up processor real estate 寄存器太多不一定能减少内存引用，并且提升了处理器的成本
  - Between 8-32 is appropriate 一般在8-32个比较合适
- Using register files with RISC makes use of using more registers RISC架构中使用寄存器文件使得能够使用更多的寄存器

# How big? 多大的寄存器?

- Address register: large enough to hold full address 地址寄存器：足够大能容纳全部地址
- Data address: large enough to hold full word 数据寄存器：足够大能够容纳一个字
- Sometimes combine two data registers to hold double length data 有时会用两个数据寄存器来保持双倍长度的数据
  - In C language, there is a double integer and a long integer, both of which are two words long C语言中，有一个双倍整型数，和长整型数，都是占2个字长

# Condition code registers 条件码寄存器

- Also called flag registers, some of which are visible to users 也称为标志寄存器，有一部分对用户可见
- After operating，CPU set the condition bit according to the result CPU操作完成后，根据操作结果设置条件位
  - After arithmetic operation, positive, negative, zero or overflow may occur 算术运算完成时，可能会产生正的、负的、0或者溢出
  - These conditions will be set in 这些情况会设置到条件码中
- Programs are allowed to read the condition code and perform 系统允许在程序中隐含地读取条件码，并进行相应的操作
- Condition code cannot be modified by the program 条件码不能被程序修改

# Control registers 控制寄存器

- Registers for control purposes, generally not visible to users 控制作用的寄存器，一般对用户不可见
- Four registers are essential to instruction execution 程序执行有四个必须的寄存器
  - Program Counter (PC) 程序计数器
  - Instruction Register (IR) 指令寄存器
  - Memory Address Register (MAR) 内存地址寄存器
  - Memory Buffer Register (MBR) 内存缓冲寄存器
- Not all processors have MAR and MBR. However, the system still needs registers similar to these two registers 并不是所有的处理器都有MAR和MBR。但是系统中还是需要有类似于这2个寄存器的寄存器

# Status Register--Program status word 程序状态字

- It may be a register or a set of registers 可能是一个寄存器，也可能是一组寄存器
- PSW contains a set of status information PSW包含了一组状态信息
  - Sign, zero, carry, equal, overflow 符号位，0，进位或借位，相等，溢出
  - interrupt enable/disable 中断允许/禁止
  - Supervisor: indicates whether the CPU is executing in supervisor or user mode 监管模式：指示CPU是在监管模式还是在用户模式
- Supervisor mode: 监管模式
  - Not available to user programs 对用户程序不可用
  - Used by operating system 操作系统使用
  - Certain privileged instructions can be executed only in supervisor mode 特权指令只能在监管模式下运行

# Other status and control registers 其他状态和控制寄存器

- Other additional status and control registers 其他的状态和控制寄存器
  - Pointer register to process control block 指向进程控制块的指针寄存器
  - Interrupt vector register in vector interrupt computer 向量式中断的计算机中的中断向量寄存器
  - System stack pointer 系统栈指针
  - Page table pointer register in virtual memory 虚拟存储器中的页表指针寄存器
  - I/O operation related registers I/O操作相关的寄存器

# Design elements 设计考虑

- Control and status registers design elements 控制和状态寄存器设计的考虑
  - Need to support the operating system 需要对操作系统提供支持
  - Storage location in registers and memory 存放位置
  - Closely related to the design of the operating system 和操作系统的设计紧密关联

# Design of Registers 寄存器设计

- User-visible registers 用户可见的寄存器
  - Types of registers 寄存器类型
  - Number of registers 寄存器数量
  - Bites of registers 寄存器位数
- Control and status registers 控制和状态寄存器
  - Number of registers 寄存器数量
  - Relationship with operating system 和操作系统的关系

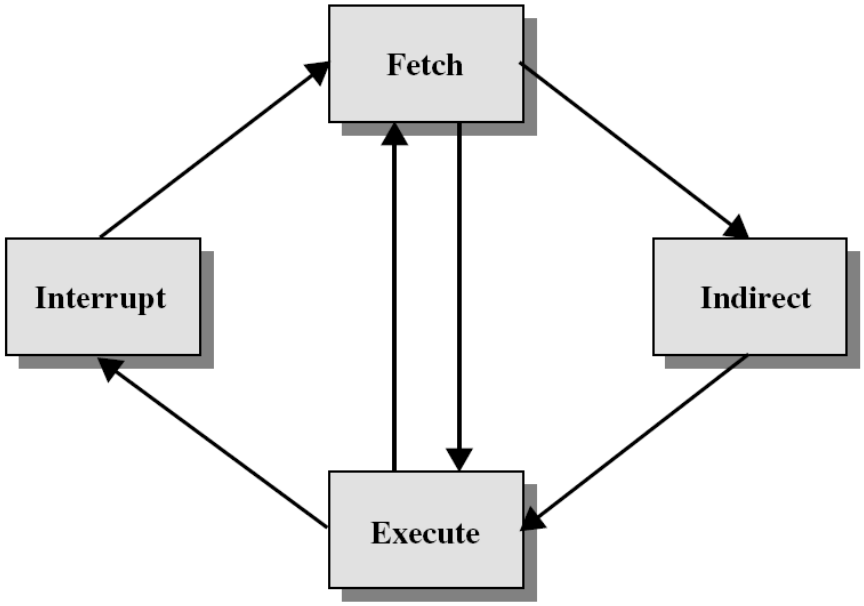
# Outline

- Processor Organization 处理器组成
- Register Organization 寄存器组成
- Instruction Cycle 指令周期
- Instruction Pipelining 指令流水线

# Instruction cycle 指令周期

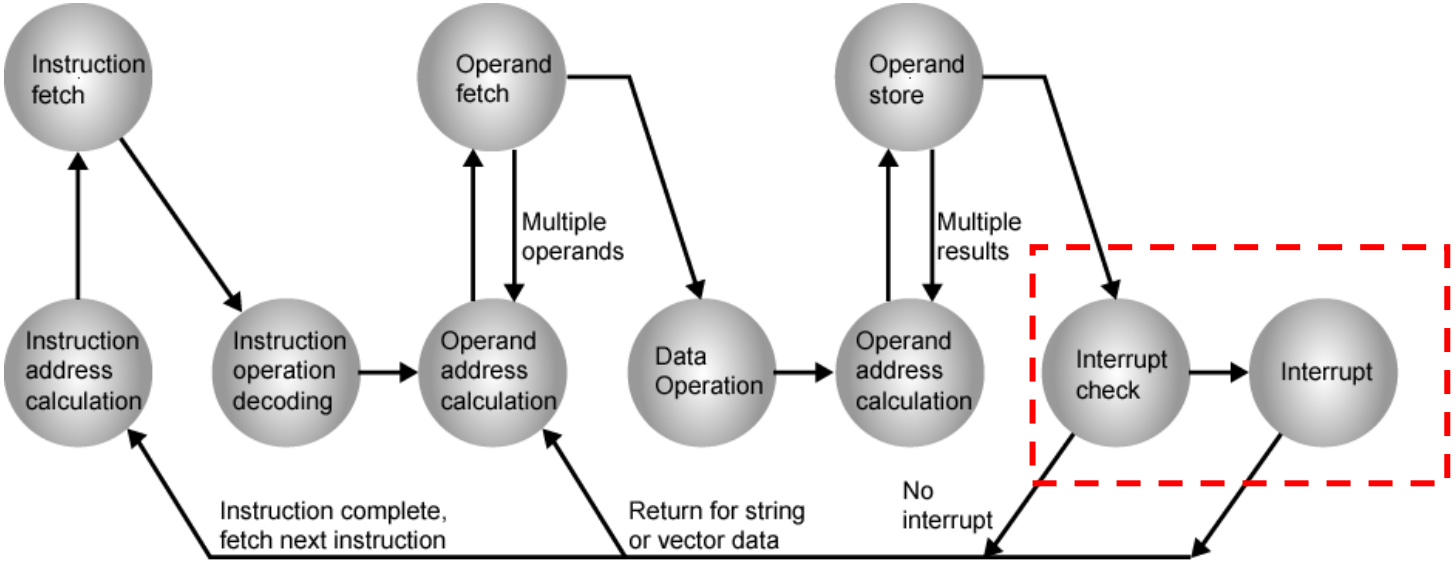
- Instruction cycle includes fetching cycle and execution cycle 指令周期包括取指周期和执行周期
- In execution cycle, first decode to get the operation type of the instruction 执行周期中，首先进行译码，得到指令的操作类型
- If instruction has operands, get the operand specifier in the instruction 如果指令有操作数，得到指令中的操作数指定符
  - Immediate 立即数
  - Register 寄存器
  - Direct addressing：memory access once 直接寻址，访问一次存储器
  - Indirect addressing：may requires more memory accesses 间接寻址：可能需要访问多次存储器
    - Also called “indirect cycle” 间接寻址的过程也称为“间接周期”

# Instruction cycle with indirect 带间接周期的指令周期



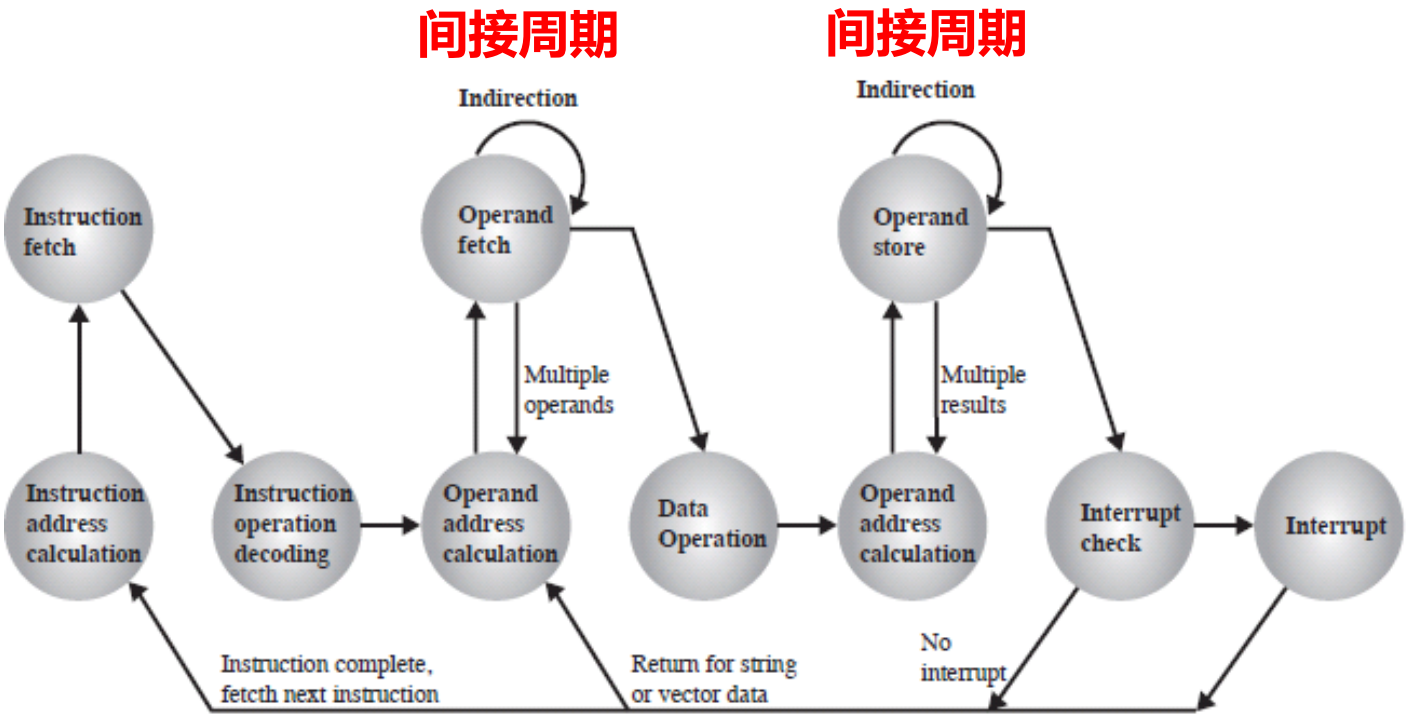

- 指令周期包括取指周期和执行周期，还可能包括间接周期和中断周期
- 取指后，通过译码确定是否包含需要间接寻址的操作数，如果有，进入间接周期
- 当前指令执行完成之后，检查是否有中断。如果有，进入中断周期

# Instruction Cycle (with Interrupts) - State Diagram 带中断的指令周期状态图




- 指令周期中，先取指，然后进行指令操作译码
- 如果涉及到操作数，进行操作数地址计算，然后取操作数
- 之后进行数据操作。操作结果如果要保存到存储器中，需要计算操作数的地址，然后保存
- 在这条指令执行完成之后，检测是否有中断。如果没有中断，继续执行下一条指令。如果有中断，就按照中断的处理规则，进行中断处理

# Indirect cycle 间接周期

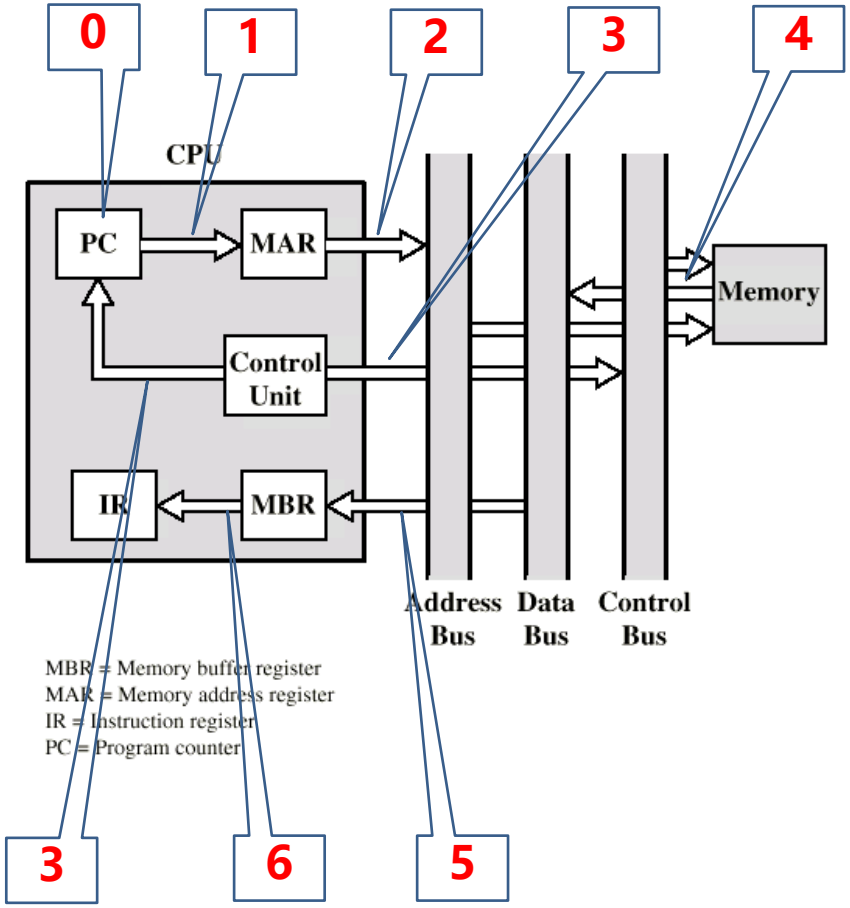



- 间接寻址过程中，由于操作数地址需要通过计算得到，所以在取操作数的过程中，可能会存在多次访问存储器的情况
- 取操作数和存结果的过程中，都可能会存在间接周期

# Data flow (instruction fetch) 取指的数据流

- Depends on CPU design 跟CPU设计有关
- Fetch inn general 一般的取指过程
  - PC contains address of next instruction PC包含下一个指令地址
  - Address moved to MAR 地址给MAR
  - Address placed on address bus 地址放在数据总线上
  - Control unit requests memory read 控制单元要求读主存
  - Result placed on data bus, copied to MBR, then to IR 结果放在数据总线上，存入MBR，以及IR
  - Meanwhile PC incremented by 1 同时，PC自动加1

# Data flow (fetch diagram) 取指的数据流

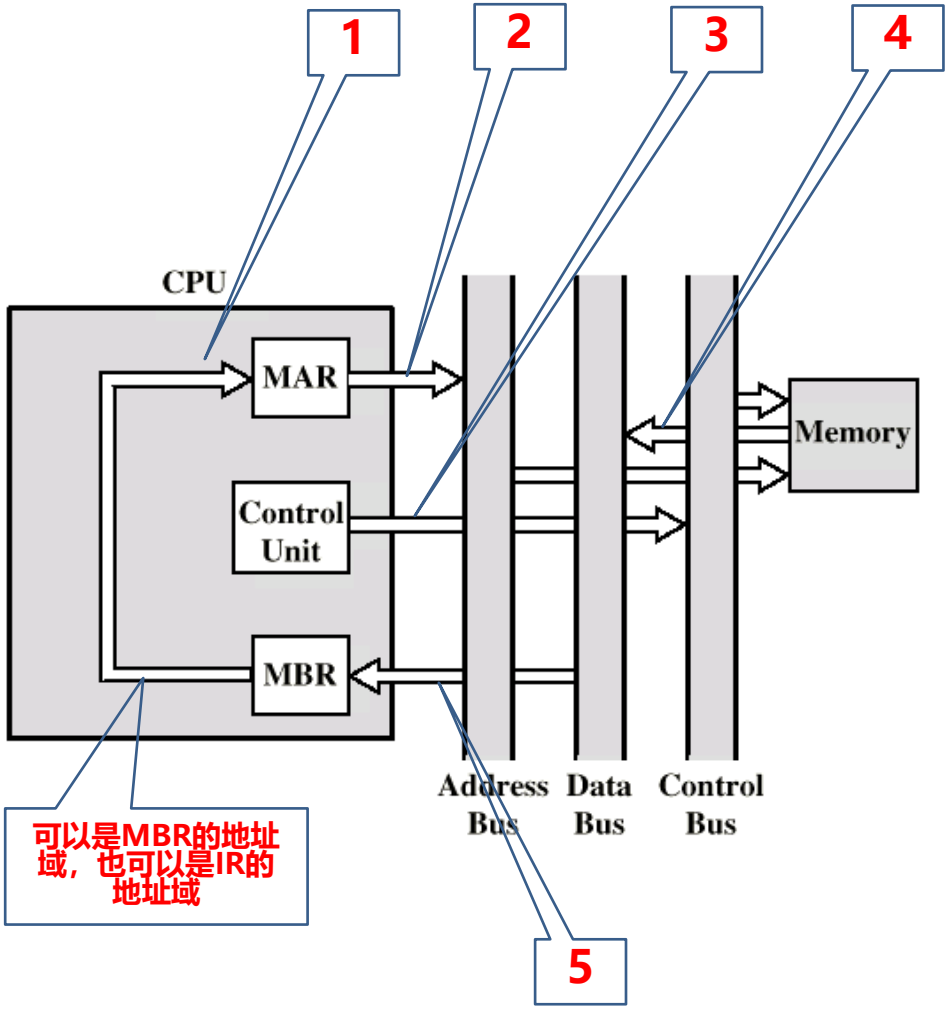



- 刚开始，下一个地址在PC中
- 地址给MAR
- 地址放到数据总线上
- 控制单元发起读控制
- 存储器把数据，也就是指令内容，放到数据总线上
- MBR读取数据总线内容，然后把指令给IR
- 控制单元还需要让PC+1，指向下一个指令

# Data flow (data fetch) 获取操作数数据流

- IR is examined IR的内容进行检查
- If there is no indirect addressing, enter the execution cycle 如果没有间接寻址，进入执行周期
- If indirect addressing, indirect cycle is performed 如果有间接寻址，启动一个间接周期
  - Rightmost N bits of MBR transferred to MAR MBR最右边的N位传给MAR
  - Control unit requests memory read 控制单元发出读请求
  - Result (address of operand) moved to MBR 结果，也就是操作数的地址给MBR

# Data flow (indirect diagram) 间接寻址数据流




- MBR中的地址域给MAR，然后MAR放在地址总线上（也可以将IR的地址域的地址给MAR）
- 控制单元给内存发一个读请求
- 内存把数据放在数据总线上。这个时候的数据实际上是操作数的实际地址
- MBR把数据总线上的数据读出来
- MBR把地址给IR的地址域

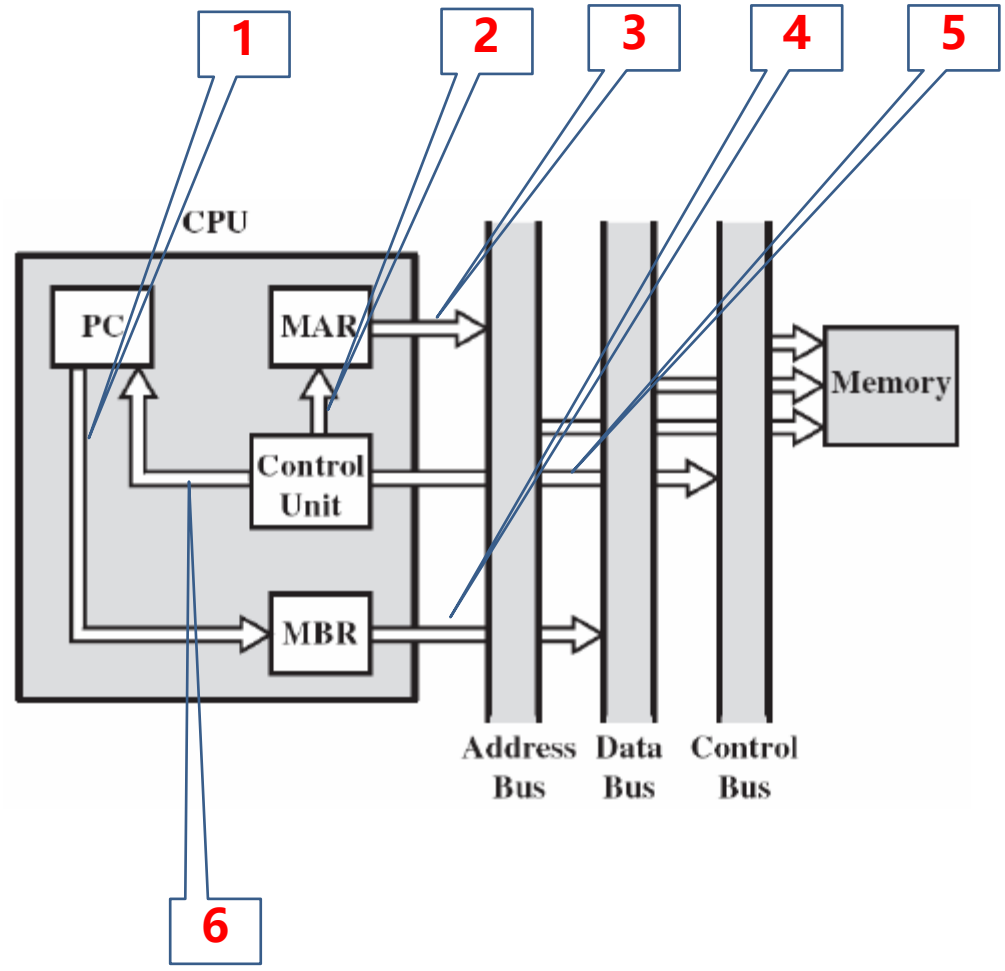
# Data flow (execute) 执行周期的数据流

- May take many forms 有多种形式
- Depends on instruction being executed 依赖于被执行的指令
- May include 可能包括
  - Memory read/write 读写内存
  - Input/Output 输入输出
  - Register transfers 寄存器传输
  - ALU operations ALU操作

# Data flow (interrupt) 中断周期的数据流

- Simple and predictable 简单，可预期
- Current PC saved to allow resumption after interrupt 保存PC，以便中断结束后返回
  - Contents of PC copied to MBR PC给MBR
  - Special memory location (e.g. stack pointer) loaded to MAR 特殊的存储位置给MAR
  - MBR written to memory MBR的内容给内存
- PC loaded with address of interrupt handling routine PC加载中断处理程序的地址
- Interrupt handler first instruction fetched 取中断程序第一个指令

# Data flow (interrupt diagram) 中断周期的数据流




- 保存返回地址
  - PC的内容给MBR
  - 控制单元把地址给MAR
  - MAR把地址放到数据总线上
  - MBR把数据放到数据总线
- 控制单元把中断处理程序的开始地址给PC
- 按照指令执行的过程，执行指令中断处理程序
- 可能还需要保存其他数据

# Instruction cycle 完整的指令周期

- Fetching cycle 取指周期
- Execution cycle 执行周期
- Indirect cycle 间接周期
- Interrupt cycle 中断周期

# Outline

- Processor Organization 处理器组成
- Register Organization 寄存器组成
- Instruction Cycle 指令周期
- Instruction Pipelining 指令流水线

# Why need pipeline?

- Development of computer application requires continuous improvement of processing capacity 计算机应用的发展要求处理能力不断提高
- The development of integrated circuit, clock frequency, registers, cache, etc. have reduced the instruction processing time and improved the processing ability 集成电路技术的发展，时钟频率的提高，寄存器的使用，cache的使用等等，减少了指令处理时间，提高了计算机的处理能力
- More and more difficult to solve problems by simply relying on the performance of hardware 单纯依靠硬件性能提升来解决问题越来越难
- The goal is the execution efficiency of instructions目标是指令的执行效率
- Better organization is needed to improve the efficiency of instruction execution 需要有更好的组织方式，以提高指令执行的效率

# What is pipeline?

- The working mode of factory assembly line is used for reference 借鉴了工厂流水线的工作模式
  - Divide the execution of instructions into several stages 将指令的执行分为几个阶段
  - Different stages of multiple instructions can be processed in parallel 多个指令的不同阶段可以并行处理
- Although execution time of each instruction is not shortened, the execution time of a group of instructions is shortened due to the parallel method 虽然每个指令的执行时间没有缩短，但是因为采用了并行的方法，一组指令的执行时间缩短了
- This is the basic idea of instruction pipeline 这就是指令流水线的基本思路

# Prefetch 预取

- Before pipelining, next instruction is taken after current instruction is executed 采用流水线之前，是当前指令执行完成之后，再去取下一个指令
- With pipelining, more than one instruction in different stages of the pipeline 流水线中不同阶段的指令超过1个
- How to get instructions is a problem 采用流水线后，如何取指令是一个问题
  - Fetch accessing main memory 取指需要访问内存
  - Execution usually does not access memory 执行一般不需要访问内存
  - Fetch next instruction during execution of current instruction 在执行当期指令的时候，取下一个指令
- Called instruction prefetch 这称为指令预取

# Advantage

- During execution of an instruction, a new instruction has entered the pipeline 在一个指令执行过程中，新的指令已经进入流水线
- After current instruction is executed, it can be executed immediately 当前指令执行完成后，可以立即开始执行
  - Next instruction has finished fetching 下一个指令已经完成取指
  - Save time for fetching 节省了取指的时间
- Accessing memory is required for fetching 取指需要访问存储器
  - If cache hits, take it directly 如果cache命中，直接取
  - If cache missing, access memory 如果cache缺失，访问存储器
- In fact, the instruction cycle is divided into more detailed stages, more pipeline stages, and more overlapping and efficient instruction execution stages 事实上，指令周期划分的阶段会更细，流水线的阶段更多，指令执行阶段重叠度更高，效率也更高

# Which instruction is Prefetched? 预取哪个指令?

- Which instruction is appropriate for prefetching? 预取哪个指令是合适的呢?
- Next instruction of the current instruction? 当前指令的下一个指令?
  - If it is executed sequentially, no problem 如果是顺序执行的，没问题
  - If there is a transition, the next instruction needs to be determined according to the conditions 如果有转移，需要根据条件来确定下一个指令
  - Hard to predict 很难预测
- Does a misprediction in prefetching affect correctness? 错误的预测是否影响准确性?
  - No, prefetched data at a "mis-predicted" address is simply not used 不会，预取错误仅仅就不用而已
  - There is no need for state recovery 不需要状态恢复

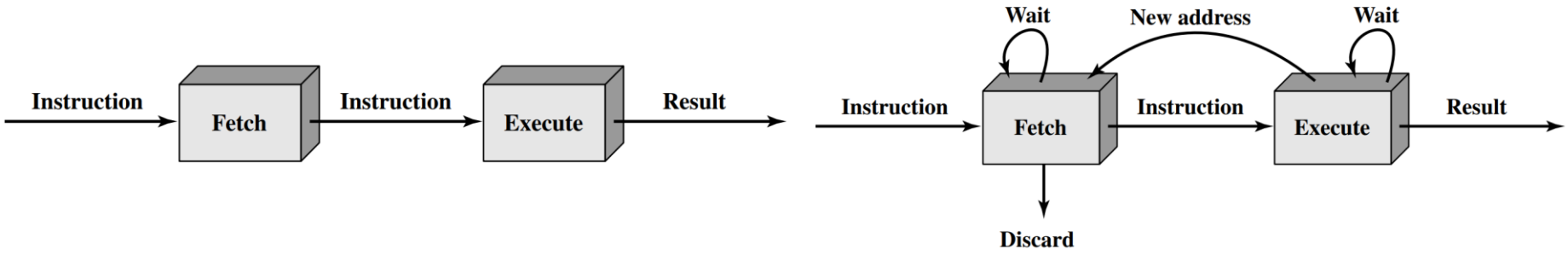
# Basics characteristics 基本特性

- In modern systems, prefetching is usually done in cache block granularity 现代系统中，预取通常在cache的块粒度
- Prefetching is a technique that can reduce both 预取效果
  - Miss rate 缺失率
  - Miss latency 减小延迟
- Prefetching can be done by 预取的途径
  - Hardware 硬件
  - Compiler 编译器
  - Programmer 程序员

# Prefetching: the four questions 预取的考虑

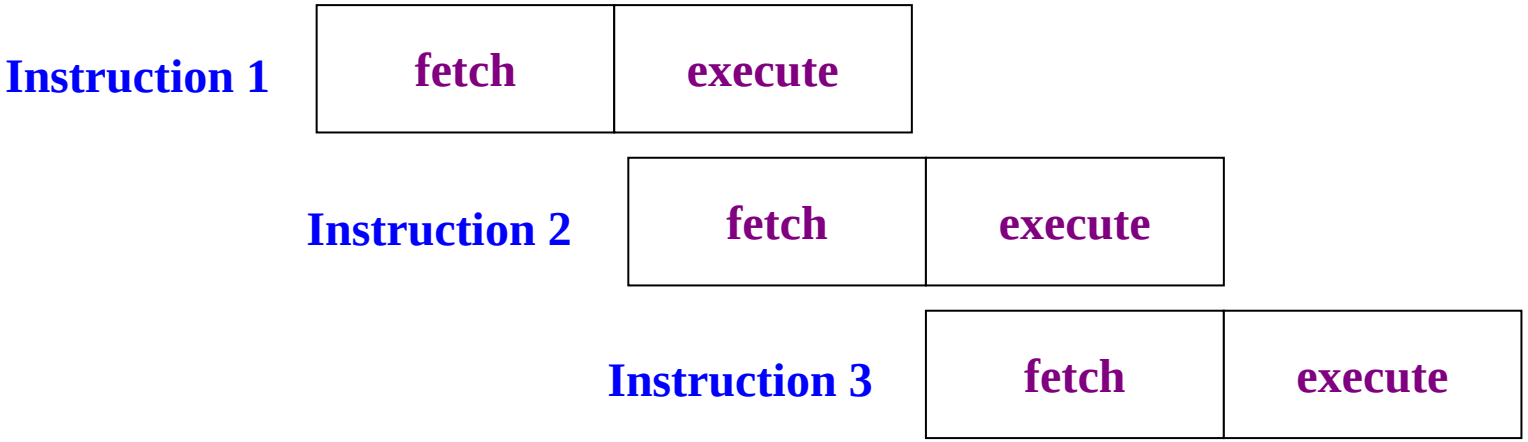
- What
  - What addresses to prefetch 预取哪个地址
- When
  - When to initiate a prefetch request 什么时候去取
- Where
  - Where to place the prefetched data 预期的数据放哪
- How
  - Software, hardware, execution-based, cooperative 软件，硬件，基于执行，合作等。

# Two stage instruction pipeline 两阶段指令流水




- 简单的指令过程就是串行处理，取指-执行-取指-执行，效率低
- 采用两阶段流水线后，在当前指令的执行过程中，进行下一个指令的取指
- 如果当前指令执行完成后，下一个指令不是预取的，需要重新取指
- 取指和执行指令的时间重叠，节省了时间
- 但是由于取指和执行指令的时间需要不一样，所以执行速度不能翻倍

# Instruction pipelining 指令流水线




- 两阶段流水线的执行过程
- 上一条指令的执行阶段和下一条指令的取指阶段在时间上是重叠的
- 每个指令的总体执行时间没有缩短，部指令的执行时间缩短了
- 如果取指和执行时间相同，那么流水线的执行时间是串行执行的一半，性能提升一倍

# Problem about performance 性能提升的问题

- But not doubled 没有双倍
  - Fetch usually shorter than execution 取指通常比执行要短
  - Instruction execution process is complex and time-consuming 指令执行过程复杂，耗时长
  - Execution time determines the improvement effect 执行时间决定了提升效果
- Jump or branch instruction 跳转指令
  - means that prefetched instructions are not the required instructions 预取的指令不是需要的指令
  - Get the actual instructions according to the results 根据结果再去取实际需要的指令
- Add more stages to improve performance 增加多个步骤去提升性能

# How to improve concurrency? 如何提高并发性

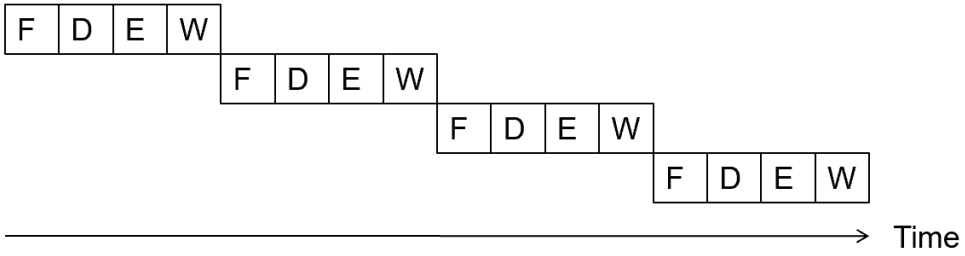
- Goal: More concurrency → Higher instruction throughput 目标：更多的并发性-更高的吞吐量
- Method: When an instruction is using some resources in its processing phase, process other instructions on idle resources 当指令在处理阶段使用某些资源时，在空闲资源上处理其他指令
  - Fetch next instruction when an instruction is being decoded 解码时取指
  - Decode an instruction when an instruction is being executed,执行时解码
  - Execute the next instruction when current instruction is accessing memory 访问内存时执行下一个指令
  - When an instruction is writing its result into the register file, access data memory for the next instruction 写结果的时候，下一个指令访问内存

# Summary

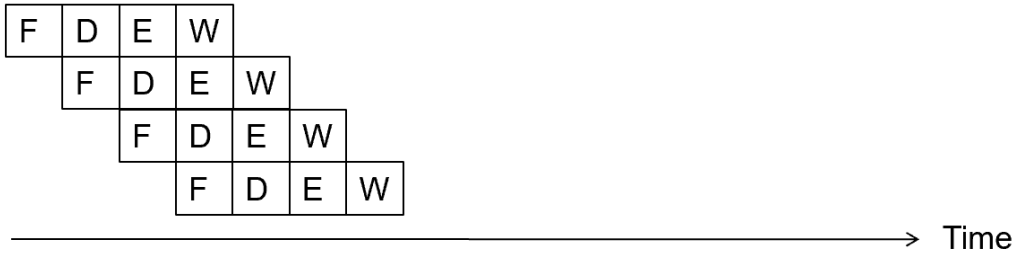
- Analogy: "Assembly line processing" of instructions 类比：指令的装配线处理
- Pipeline the execution of multiple instructions 执行多条指令的流水线
  - Divide the instruction processing cycle into distinct "stages" of processing 将指令处理周期划分为不同的处理"阶段"
  - Ensure there are enough hardware resources to process one instruction in each stage 确保有足够的硬件资源在每个阶段处理一条指令
  - Process a different instruction in each stage 在每个阶段处理不同的指令
  - Instructions are executed in the order of program 指令按照程序中的顺序执行
- Benefit: Increases instruction processing throughput 好处：增加指令处理吞吐量

# Example: execution of four independent add 4 阶段加法举例

- Multi-cycle: 4 cycles per instruction



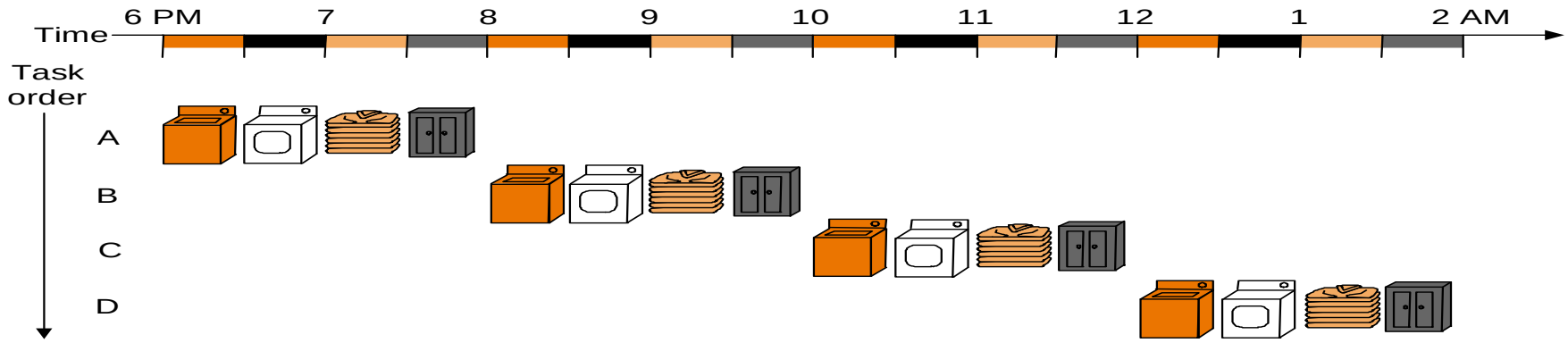

- Pipelined: 4 cycles per 4 instructions (steady state)




- 加法指令流水线
- 整个指令分为4个阶段：取指，译码，执行，写结果，均为t
- 采用串行方法，执行n个指令，需要4nt的时间
- 采用4阶段流水线，每个阶段完全独立，n个指令，需要nt+3t的时间
- 基本上是1/4的时间
- 如此美好？？

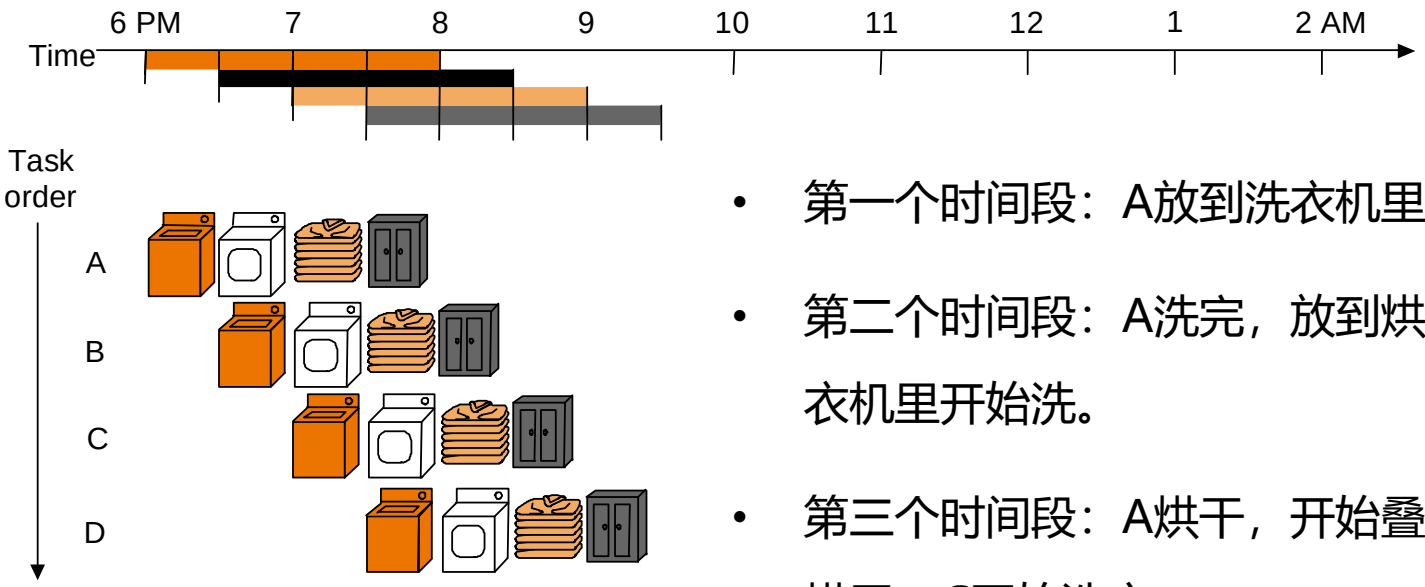

52

# The laundry analogy 洗衣服类比



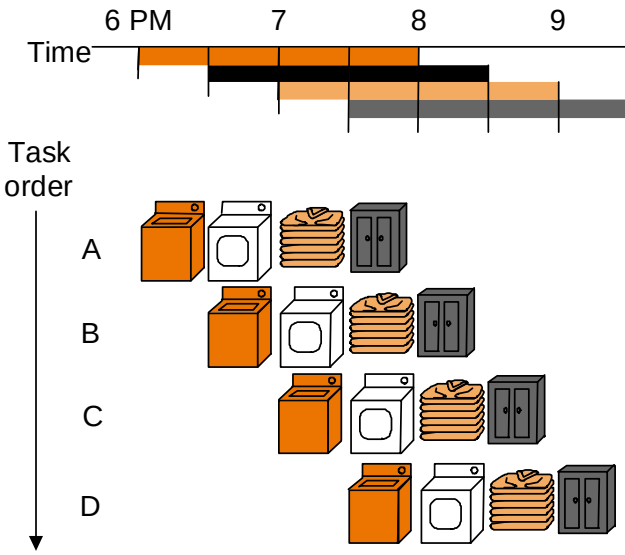

- 整个过程包括四个步骤：洗衣服，烘干，叠衣服，放到柜子里
- 过程的特点：
  - 四个步骤有先后顺序
  - 不同的衣服处理过程相互独立
  - 不同阶段不共享资源
- 比较适合采用流水线

# Pipelining Multiple Loads of Laundry 洗衣流水线



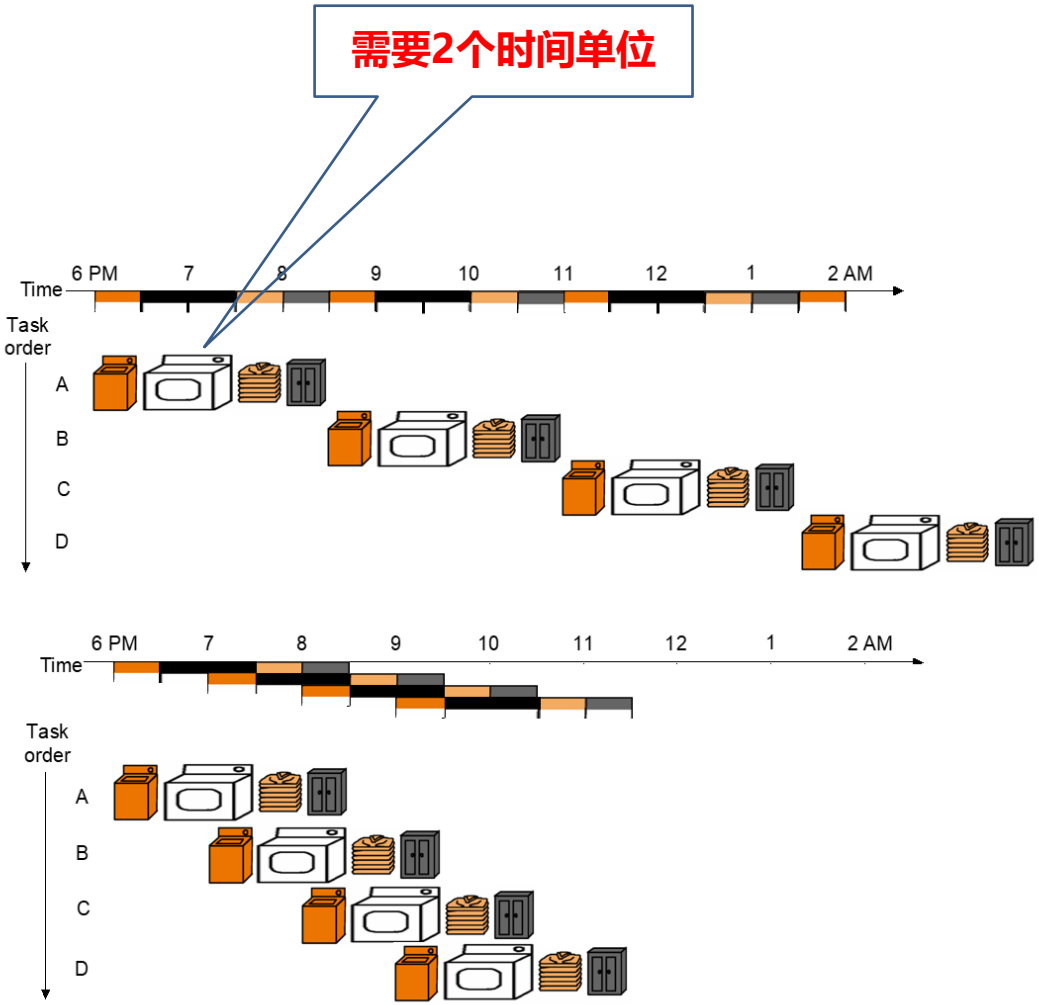

- 第一个时间段：A放到洗衣机里洗
- 第二个时间段：A洗完，放到烘干机；B放到洗衣机里开始洗。
- 第三个时间段：A烘干，开始叠。B洗完，开始烘干。C开始洗衣。
- 第四个时间段：A叠好，可以放柜子。B烘干，开始叠衣服。C洗完，开始烘干。D可以开始洗。
- 第五个时间段：A完成了。B开始放柜子，C开始叠，D开始烘干，E开始洗，……

# Effect 效果



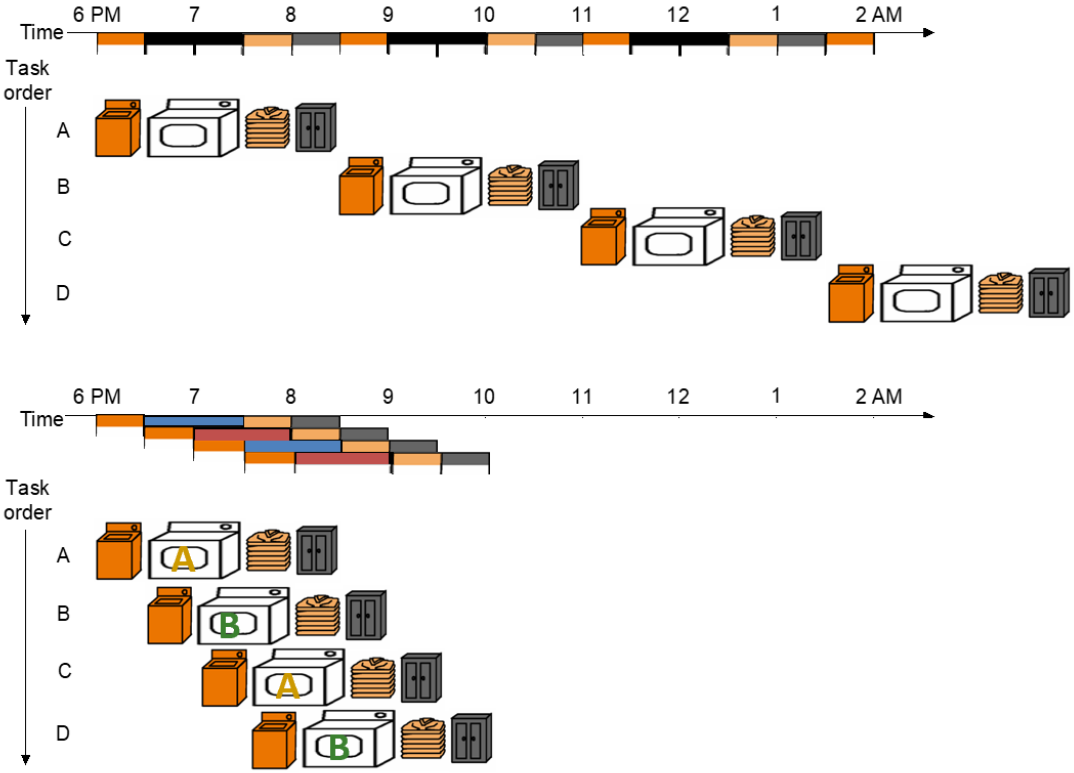

- 原来4件衣服，需要16个时间单位
- 采用流水线后，只需要7个时间单位
- 每件衣服的处理过程还是4个时间单位
- 充分利用所有的资源
- 提高总体吞吐量
- 假定洗衣、烘干、叠衣服、收纳这4个阶段花费的时间是一样的

# In practice 实际情况




- 烘干衣服需要2个时间单位，这样，如果完全串行，需要20个时间单位
- 采用流水线后，有等待烘干机的时间
- 4件衣服需要11个时间单位
- 理论上的速度为非流水线的2.5倍
- 最慢的步骤决定了整个系统的吞吐量

# How?




- 烘干机成为整个系统的瓶颈
- 补充资源，配置2个烘干机
- 下一个衣服洗完后，不需要等待上一个衣服的烘干，用另一台烘干机
- 关键环节增加资源，使得整个吞吐量回到之前的情况
- 代价就是配置额外的资源

# Goal of pipeline 流水线的目标

- Increase instruction throughput with little increase in cost 在几乎不增加成本的前提下，提高指令的吞吐量
  - Process instructions in the order required by the program 能够按照程序要求的顺序处理指令
  - Hardware cost cannot be increased too much 硬件成本不能增加太多
  - Instruction throughput can be greatly increased 指令的吞吐量能够有大幅度增加

# An ideal pipeline -1 理想的流水线1

- Repetition of identical operations 重复相同的操作
  - Same operation, different operation objects 相同的操作，操作对象不一样
  - Automobiles of the same model can be produced on one assembly line 生产同型号的汽车可以在一个流水线上
  - Different operations require different steps, which affects the operation of the pipeline 不同的操作，要求的步骤不一样，影响流水线的运行
  - The production of automobiles and motorcycles requires different steps and cannot be put on the same assembly line 生产汽车和生产摩托车，要求的步骤不同，不能放在同一个流水线上

# An ideal pipeline -2 理想的流水线2

- Operating objects are independent of each other 操作对象之间相互独立
  - There is no dependency between each operation object 各个操作对象之间没有依赖关系
  - For example, there is no relationship between cars produced on the assembly line 例如，流水线上生产的车之间没有关系
  - Operating objects with sequential dependencies affect each other during parallel operations 有先后依赖关系的操作对象，在并行操作的时候相互影响

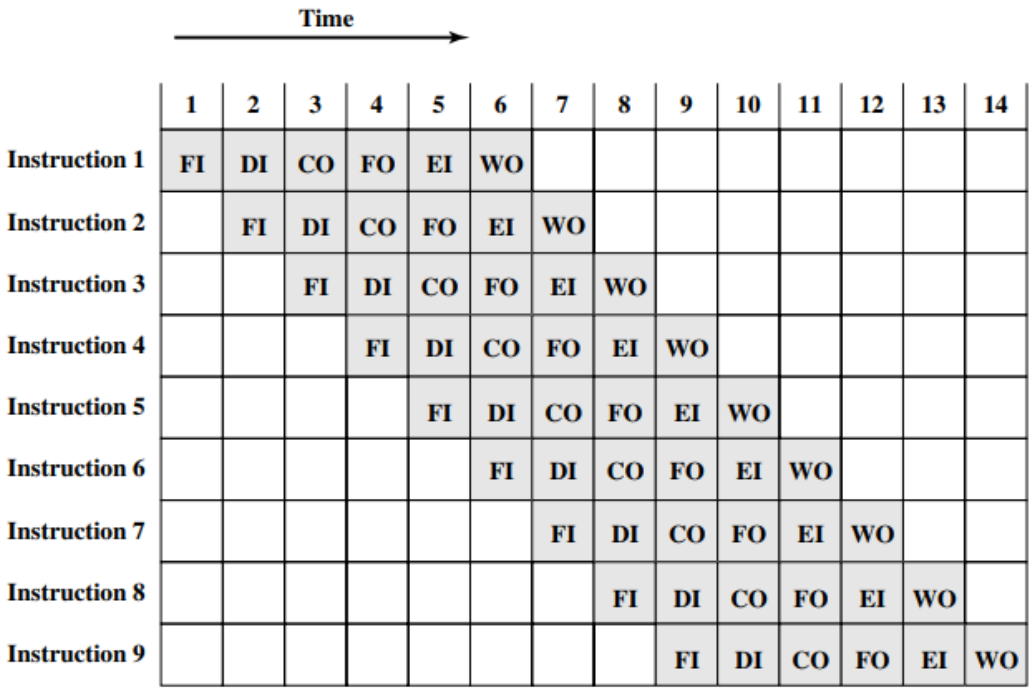
# An ideal pipeline -3 理想的流水线3

- A complete operation can be decomposed into several sub operations 完整的操作可以分解若干个子操作
  - Each sub operation takes the same time 每个子操作需要的时间一样
  - Each sub operation requires independent resources and does not share resources 每个子操作需要的资源独立，不共享资源
  - If sub operation requires different time, some sub operations must wait 如果子操作需要的时间不一样，有些子操作必须等待
  - Resource sharing leads to resource contention 共享资源导致资源争抢

# Instruction pipeline 指令流水线

- For the pipeline design of instructions, we divide the execution of instructions into six stages 为了进行指令的流水线设计，我们把指令的执行分为6个阶段
  - Fetch instruction(FI) 取指
  - Decode instruction(DI) 指令译码
  - Calculate operands（CO） 计算操作数地址
  - Fetch operands(FO) 取操作数
  - Execute instructions(EI) 执行指令
  - Write result(WO) 写结果
- Overlap these operations重复这些操作
- Looks good 看起来还不错

# Timing of pipeline 流水线的时序

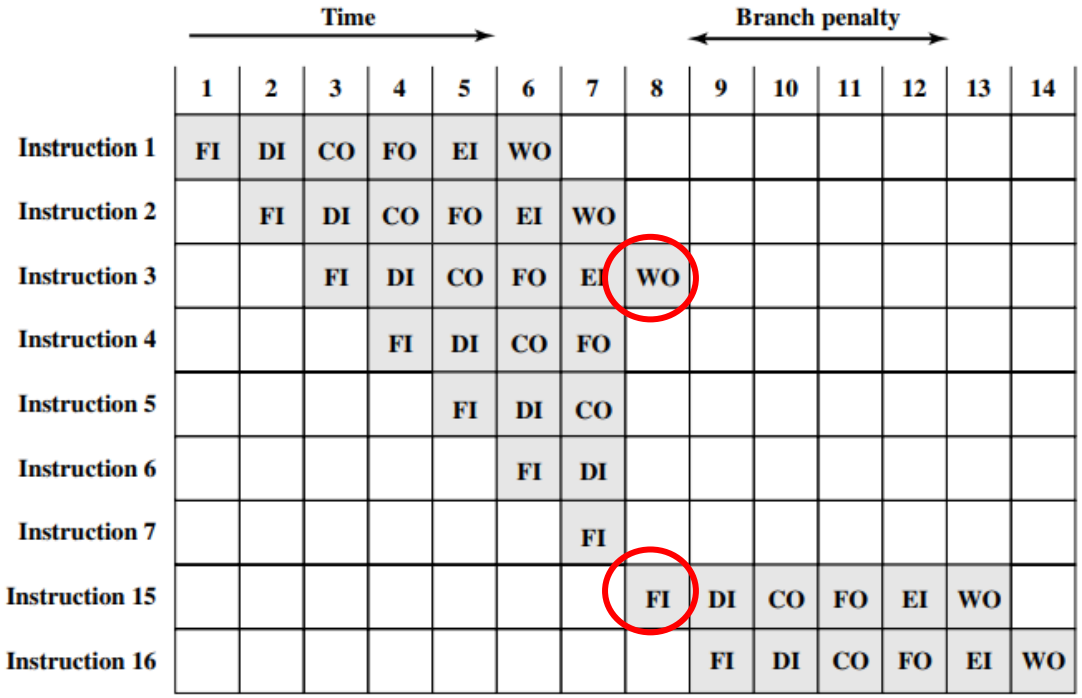



- 理想的指令流水线的执行过程
- 指令执行分为6个阶段，相互之间不共享资源
- 按照流水线的方式来执行，从第六个时间单位开始，每个时间单位都会有1个指令完成执行
- 指令数量足够多时，执行效率为原来的6倍

# Summary 小结

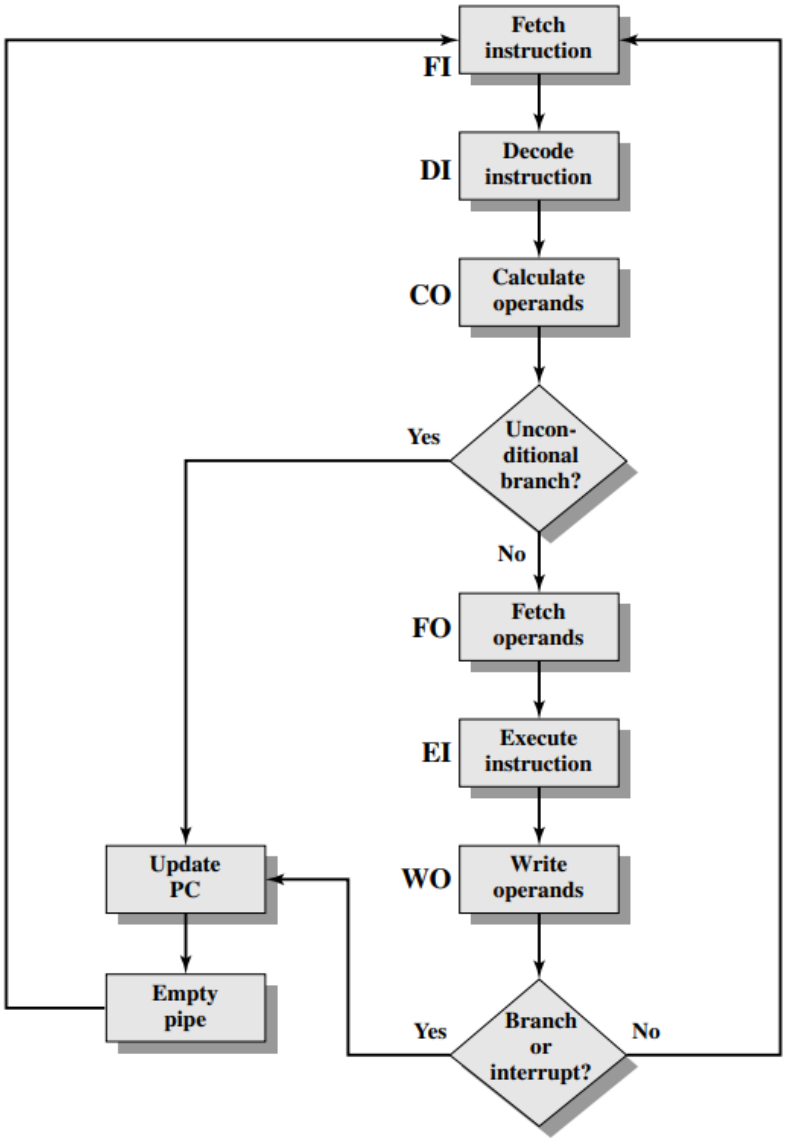
- The total execution time for each individual instruction is not changed by pipelining. 每条指令的总执行时间不会因流水线而改变
  - It still takes an instruction cycle to make it all the way through the processor. 它仍然需要一个指令周期才能通过处理器
- Pipelining doesn't speed up instruction execution time 流水线没有加快指令执行时间
- It does speed up program execution time by increasing the number of instructions finished per unit time 通过增加每单位时间内完成的指令数来加快程序执行时间
- But，if branch？但是，如果分支呢？

# Branch in a pipeline 流水线中的分支



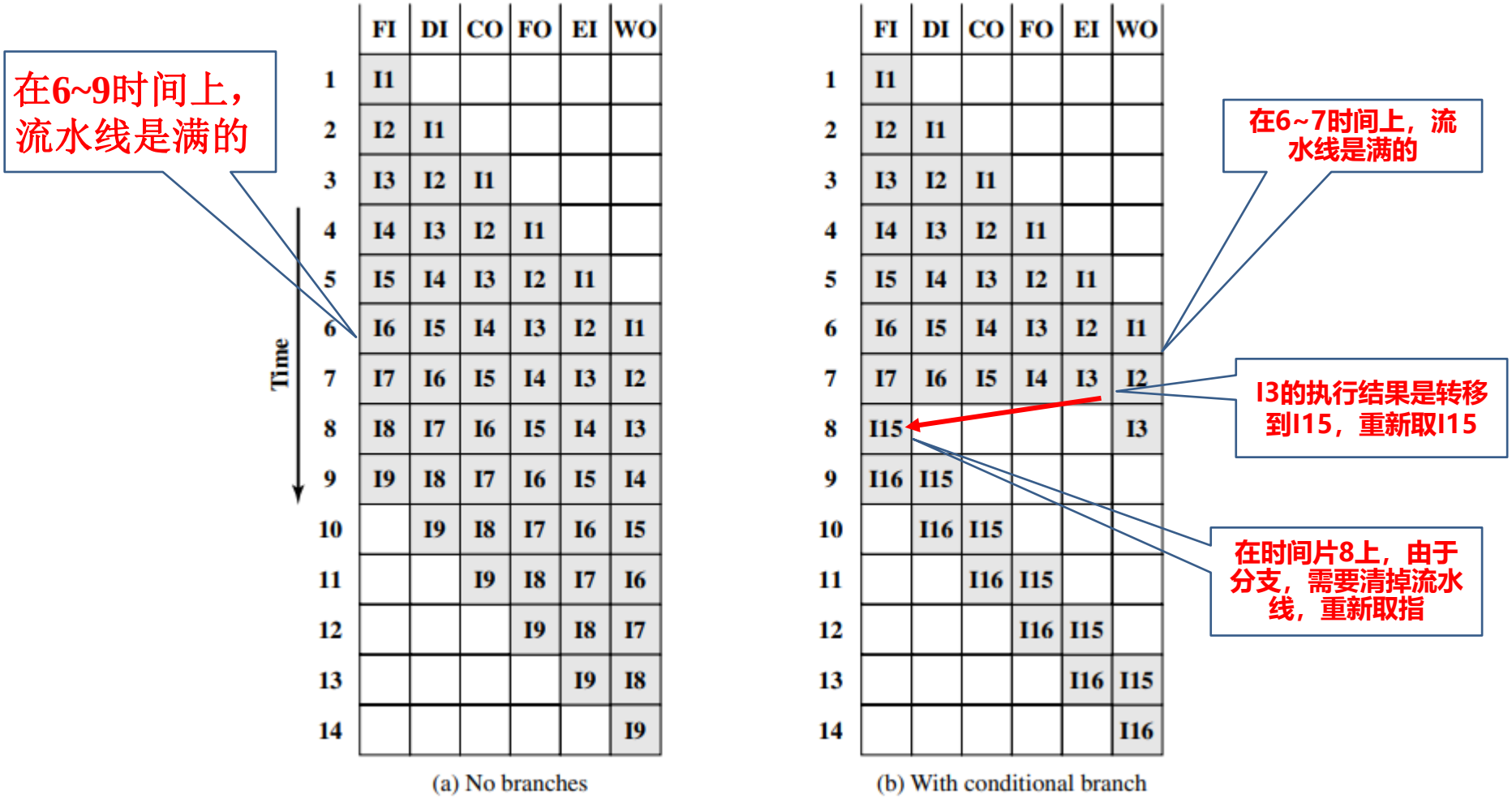

- 指令1和2的执行都是正常的
- 指令3在时间片8时，需要跳转到指令15的执行
- 指令4~7已经完成的处理作废
- 需要重新开始指令15的取指
- 第9到第12时间片，没有指令完成执行，称为分支惩罚
- 分支越多，分支惩罚就越多，整个程序的指令吞吐率就越低

# Six stage instruction pipeline 六阶段流水线流程图




- 第一步是取指，之后是指令译码，并计算操作数地址
- 此时，需要判断指令是否是无条件转移，如果是，那么更新PC，并清空流水线，继续开始取指
- 如果不转移，正常执行指令，取操作数，然后执行指令，并写操作数
- 判断是否进行分支，或者是否有中断。如果是，那么和无条件分支一样，更改PC，清空流水线，继续往下执行后续指令

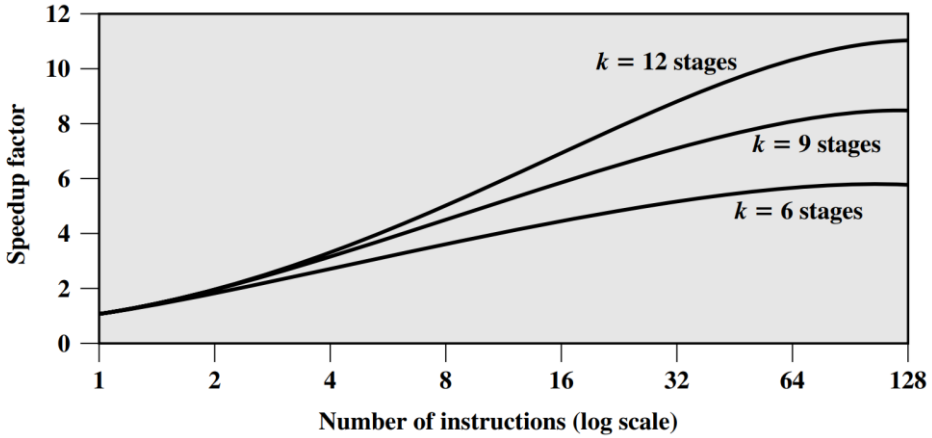
# Alternative Pipeline Depiction 换一种描述

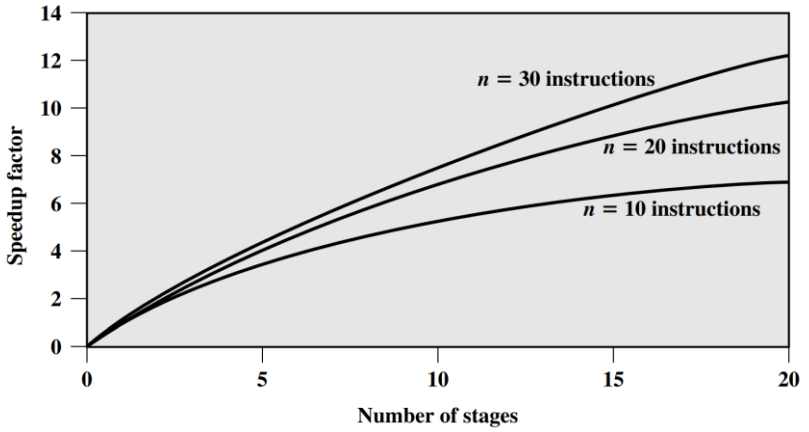


# Other factors 其他因素

- Besides branches, are there other factors that affect the pipeline? 除了分支之外，是否还有其他因素影响流水线?
- Data transmission between different parts takes time 数据在不同的部件之间的传送，需要花费时间
- Theoretically, the more stages, the higher the efficiency of instruction execution 理论上分的阶段越多，指令执行的效率越高
  - The more stages are divided, the more complex the control between stages will be 分的阶段越多，阶段之间的控制会越复杂
  - Latching delay, buffering between phases takes a certain time 锁存延迟，阶段之间的缓冲需要一定的时间
  - Need reasonable design 需要合理设计

# Speedup factors with instruction pipelining 加速比




- 假定总共需要执行n条指令，采用的流水线段数为k，那么使用指令流水线相对于不使用流水线的加速比的定义是：$S_k=nk/(k+n-1)$
- 图a是流水线阶段数为6，9，12时，需要执行的指令数和加速比的关系。可以看到随着指令数的增加，加速比趋向于流水线的阶段
- 图b从另一个角度来分析加速比。指令数越多，加速比越接近理论上的加速比。而随着段数的增加，加速比增加缓慢
- 流水线段数能带来更好的潜在加速比，但同时也带来很多问题。比如分支时需要清空流水线，段间延时也需要考虑

# Analysis of instruction pipeline 指令流水线分析

- Is the instruction pipeline an ideal pipeline? 指令流水线是不是理想的流水线?
- What are the characteristics of an ideal pipeline? 理想流水线的特点有哪些?
  - Repetition of identical operations 重复相同的操作
  - Operating objects are independent of each other 操作对象之间相互独立
  - A complete operation can be decomposed into several sub operations 完整的操作可以分解若干个子操作

# Characteristic 1 特征1

- Identical operations ... NOT! 相同的指令？不是

  ⇒ different instructions → not all need the same stages 不是所有的指令都需要相同的步骤

- Forcing different instructions to go through the same pipe stages 要求不同指令经过同一个流水线
- Some pipeline stages are idle 有些流水线阶段空闲
- Leading to a waste of time, called external fragmentation 导致时间上的浪费，称为外部碎片

# Characteristic 2 特征2

- Independent operations ... NOT! 独立操作？不是

  ⇒ instructions are not independent of each other 要求指令相互独立，但指令并不独立

- Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results 需要检测和解决指令间依赖关系，以确保流水线提供正确的结果
- Pipeline stalls frequently due to branch 经常由于分支导致流水线暂停
- Poor operation of the pipeline 流水线运行不畅

# Characteristic 3 特征3

- Uniform suboperations ... NOT! 一致的子操作?不是

  ⇒ different pipeline stages → not the same latency 不同的流水线阶段的延迟是不一样的

- Need to force each stage to be controlled by the same clock 需要强迫所有的阶段由同一个时钟来控制
- Some pipe stages are too fast but all take the same clock cycle time 某些流水线阶段执行的速度非常快，但仍要用统一的时钟
- These wasted time are called internal fragmentation 这些浪费的时间称为内部碎片

# Issues in pipeline design 流水线设计中的问题

- Reasonably divide the stages of instructions 合理划分指令的各个阶段
  - How many stages is the instruction cycle divided into? 指令周期分为多少个阶段?
  - what is done in each stage 每个阶段需要完成哪些工作
- Handling exceptions, interrupts 处理意外和中断
- Keeping the pipeline correct, moving, and full 确保流水线正常运转，并且是满载运行
  - Data dependences 数据相关性
  - Control dependences 控制相关性
  - Resource conflict 资源冲突
  - Long-latency (or multi-cycle) operations 长延迟多周期的操作

# Causes of pipeline stalls 流水线暂停的原因

- Pipeline stall: A condition when the pipeline stops moving 流水线暂停：停止移动的状态
- Causes of stall
  - Resource contention 资源竞争
  - Dependences between instructions, including data dependence and control dependence 指令间的相关性，包括数据相关性和控制相关性
  - Long-latency (multi-cycle) operations 长延迟操作

# Dependences and Their Types 相关性及类型

- Also called “hazard” or “pipeline bubble” 也称为冒险或流水线空泡
- Dependences dictate ordering requirements between instructions 依赖于指令之间的特定顺序
- Two types 两种类型
  - Data dependence 数据相关性
  - Control dependence 控制相关性
- Resource contention is sometimes called resource dependence 资源竞争有时也称为资源相关性
- When dependency occurs, the pipeline will be suspended, which is called pipeline adventure 依赖发生时，流水线暂停运行，称为流水线冒险

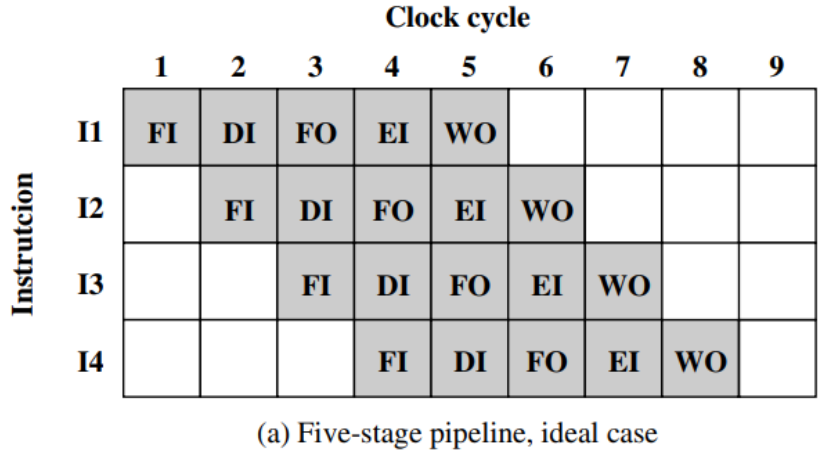
# Resource hazards 资源冒险

- Two (or more) instructions in pipeline need same resource 两个或多个指令需要相同的资源
  - Executed in serial rather than parallel for part of pipeline 需要顺序执行，不能并行
  - Also called *structural hazard* *也称为结构冒险*
- It is caused by unreasonable structure or insufficient resources 结构不合理，或者资源不够导致的
  - Such as using the same register 比如都要用同一个寄存器
- The solution is generally to increase available resources, such as adding a dryer in the previous example 解决办法一般是增加可用资源，比如前面例子中增加烘干机

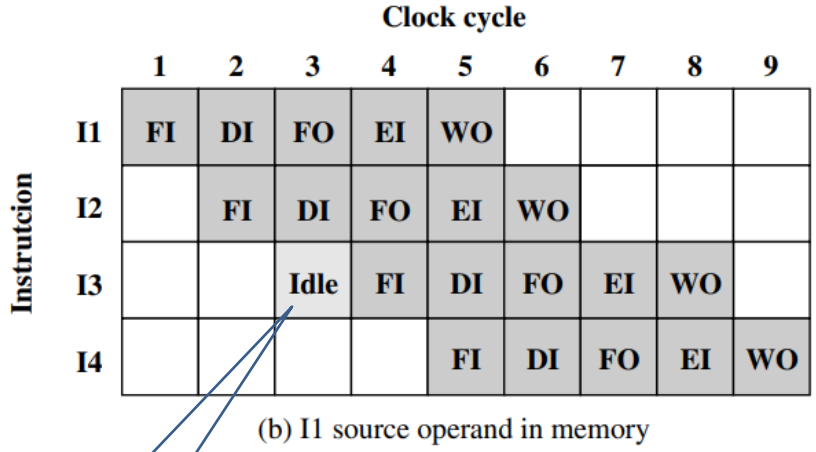
# Example

- Assume simplified five-stage pipeline 例：一个五段流水线
  - Each stage takes one clock cycle 每个阶段一个时钟
  - Ideal case is new instruction enters pipeline each clock cycle 理想状态是每个时钟周期有一个新指令进入流水线
- Assume main memory has single port，and ignore the cache 假定主存只有1个端口，并且不考虑cache
- Instruction fetches and data reads and writes performed only one at a time 这样同一时刻，取指、读数和写数只能执行1个
- Operand read or write cannot be performed in parallel with instruction fetch 操作数的读、写不能和取指并行来做

# Example




(a) Five-stage pipeline, ideal case




(b) I1 source operand in memory

空1个周期

- 第3个时钟周期，I1需要读取内存取操作数，同时I3也需要取指
- 两个指令读需要读存储器，发生资源冲突
- I3需要空一个时钟周期，等到第4个时钟周期的时候，才去取指
- 因为资源冲突而浪费了1个时钟周期
- 如果只有一个ALU，执行指令也可能会冲突

# Handling resource contention 处理资源竞争

- Solution 1: Eliminate the cause of contention 方法1：源头解决
  - Duplicate the resource or increase its throughput 复制资源或提高吞吐量
  - E.g., use separate instruction and data memories (caches) 比如说用独立的指令和数据内存（cache）
  - E.g., use multiple ports for memory structures 比如在存储器结构中使用多端口机制
- Solution 2: Detect the resource contention and stall one 检测冲突，停止其中一个
  - Need to decide which one to stop 需要确定停止哪一个

# Data hazards 数据冒险

- Conflict in access of an operand 访问同一个操作数
  - E.g. ,both instructions access a particular memory or register operand 两个指令都要访问存储器或寄存器操作数
- If two instructions are executed serially in strict order, that is one instruction executes after the finish of the previous instruction execution. No problem. 如果两个指令严格按照顺序串行执行，一个指令执行完才执行另一个，没有问题
- If in a pipeline, operand value could be updated so as to produce different result from strict sequential execution 在流水线中，操作数的值可能会改变，产生和顺序执行不同的结果
- Data Hazard is caused by the conflict of access to the same operand location 数据冒险是源于对同一个操作数位置的访问出现的冲突

# Types of data hazard 数据冒险的类型

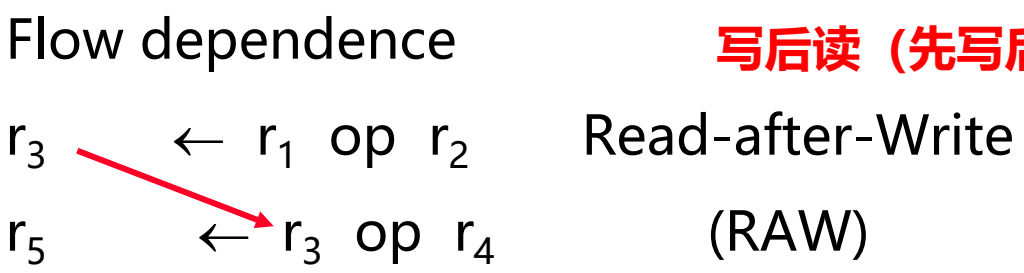
- Types of data dependences:
  - read after write 写后读
    - Called "True dependence" 真相关
  - write after read 读后写
    - Called "Anti dependence" 反相关
  - write after write 写后写
    - Called "Output dependence" 输出相关

# True dependency 真相关

- Read after write (RAW), or true dependency 写后读，也称为真相关
  - An instruction modifies a register or memory location 一个指令需要修改寄存器或存储器中的数据
  - Succeeding instruction reads data in that location 后面的指令需要去读这个位置的数
  - Hazard occurs if read takes place before write complete 如果读操作发生在写操作之前，就出现了数据冒险
  - What needs to be read by succeeding instruction is the modified data 后续指令需要读的是修改后的数据下一个指令
  - After the pipeline is adopted, the read data becomes the data before writing 采用流水线之后，读的数据变成了写之前的数据

# True dependency 真相关

Flow dependence 写后读（先写后读）




Read-after-Write (RAW)

- 第一个指令需要写r3
- 第二个指令需要读r3
- 第二个指令必须要等第一个指令执行完成之后并写了r3，才能完成读操作数的指令，否则读取的r3不是需要的数

# Anti dependence 反相关

- Write after read (RAW), or anti-dependency 读后写，也称为反相关
  - An instruction reads a register or memory location 一个指令需要先读寄存器或存储器的某个位置
  - Succeeding instruction writes to location 后续的指令需要写这个位置
  - Hazard occur if write completes before read takes place 如果写操作在读操作之前完成，这样就出现数据冒险
  - The data of the first instruction read operation is incorrect 第一个指令的读操作得到的数据是不正确的数据

# Anti dependence 反相关

Anti dependence

读后写（先读后写）

$r_3 \leftarrow r_1$ op $r_2$ Write-after-Read

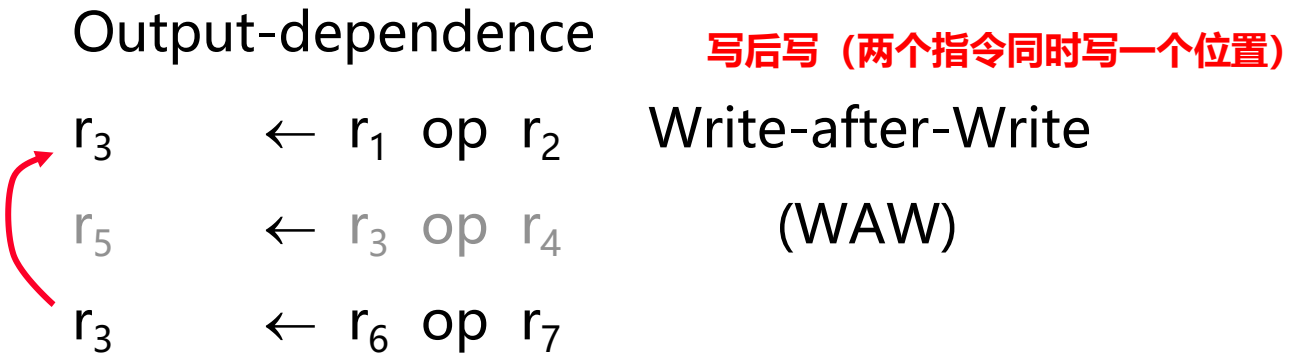
$r_1 \leftarrow r_4$ op $r_5$ (WAR)

- 第一个指令读r1
- 第二个指令写r1
- 如果先执行了第二个指令，那么结果也不正确
- 普通流水线中不会出现这种情况，在超标量中会出现这种情况

# Output dependence 输出相关

- Write after write (WAW), or output dependency 写后写，也称为输出相关
  - Two instructions both write to same location 两个指令都要先向同一个位置写数据
  - Hazard if writes take place in reverse of order intended sequence 如果写操作的顺序和实际指令的顺序相反，就出现结果错误的情况
  - The data to be stored is the data written by the second instruction 需要存储的数是第二个指令写的数
  - In the pipeline, the data actually saved is the data written by the first instruction 在流水线中，实际保存的数是第一个指令写的数
  - Data of memory or register is not required 存储器或寄存器的数据不是需要的数据

# Output dependence 输出相关




- 第一个执行写r3
- 第三个指令也写r3
- 如果第三个指令先执行了，也结果不正确
- 普通流水线不会出现这种情况，在超标量中会出现这种情况

# How?

- For all of them, we need to ensure semantics of the program is correct 都需要处理，保证程序的语义正确
- True dependences always need to be obeyed because they constitute true dependence on a value 真相关需要遵守因为是实际的依赖性
- Anti and output dependences exist due to limited number of architectural registers 反相关和输出相关是因为寄存器不够
  - They are dependence on a name, not a value 依赖一个寄存器名称，而不是值
  - We will later see what we can do about them 后面讨论如何处理

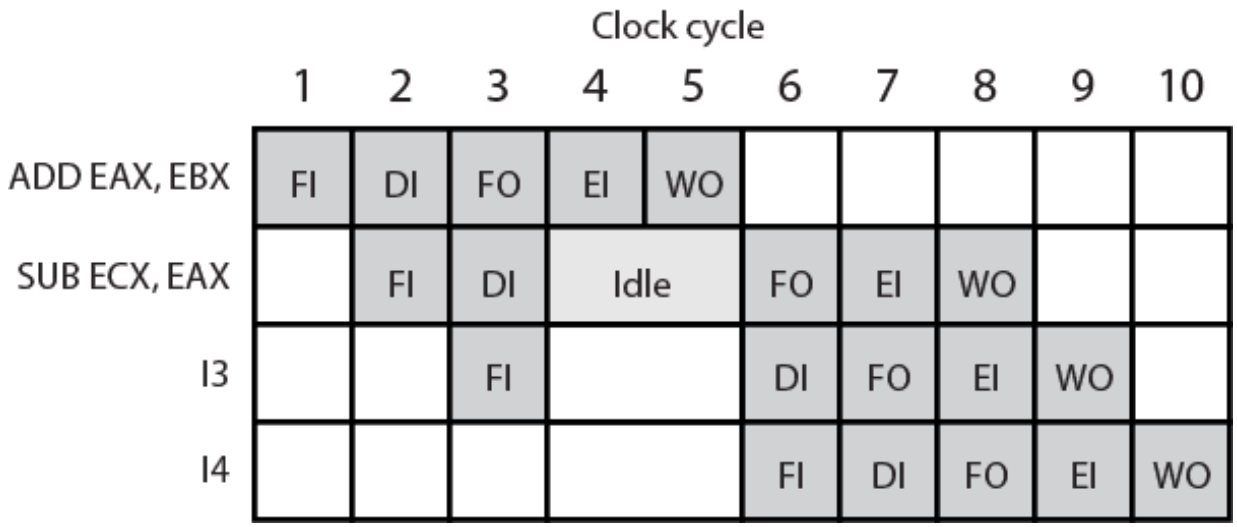
# Example

- E.g. x86 machine instruction sequence: x86指令序列

```
ADD EAX, EBX        /* EAX = EAX + EBX
SUB ECX, EAX        /* ECX = ECX – EAX
```

  - ADD instruction does not update EAX until end of stage 5, at clock cycle 5 加法指令直到第五步才更新EAX
  - SUB instruction needs value at beginning of its stage 3, at clock cycle 4 减法指令在第三步，也就是时钟周期4需要数据
- EAX taken by the subtraction instruction is not the result of addition 减法指令取的EAX不是加法的结果
- Pipeline must stall for two clocks cycles 流水线需要停止2个周期
- Without special hardware and specific avoidance algorithms, results in inefficient pipeline usage 没有特殊的硬件和特殊的算法时，流水线的效率降低

# Data hazard diagram 数据冒险图示




- 在第五个时钟周期，加法指令写EAX
- 第四个时钟周期，减法要用EAX
- 如果第二个指令不等待，那取的EAX还是最早的EAX，不是加法的结果
- 所以减法指令需要停顿2个时钟周期，到第六个时钟周期才会去取操作数
- 浪费了2个时钟周期

# Method of handle 处理方法1

- True dependences are more interesting 真相关比较麻烦
  - Actual interdependence between data，requires waiting 数据之间实际相关，需要等待
- Anti and output dependences are easier to handle 反相关和输出相关相对容易处理
  - It ‘s all about writing 都是和写操作相关的
  - Use more registers 增加寄存器
  - Use different registers to eliminate possible correlation 采用不同的寄存器，来消除可能的相关性

# Method of handle 处理方法2

- Some fundamental ways of handling true dependences 几种基本的处理方法
  - Detect and wait until value is available in register file 检查并等待寄存器的值可用
  - Detect and eliminate the dependence at the software level 检查并消除依赖性
    - Register renaming 寄存器重命名
    - Discussed later 后面会讨论
  - Predict the needed value(s), execute "speculatively", and verify 预测所需的值，"推测性"执行并验证

# Control dependence 控制相关性

- Also called “control hazard” “branch hazard” 也称为控制冒险，分支冒险
- A Special Case of Data Dependence 一种特殊的数据相关性
- Occurs when the pipeline makes a wrong judgment on branch transfer 发生在流水线对分支转移做出错误的判断
- Brings instructions into pipeline that must subsequently be discarded 指令进入流水线后，还必须丢弃掉
- The pipeline cannot run with full load 导致流水线无法满载运行

# Control Dependence -1 控制相关性1

- Question: What should the fetch PC be in the next cycle? 问题：PC中的内容在下一个周期中是什么?
- Answer: The address of the next instruction 回答：下一条指令的地址
  - We assume that programs run sequentially 我们假定程序都是顺序运行的
  - All instructions are control dependent on previous ones. 所有指令依赖于之前指令的控制
- If the fetched instruction is a non-control-flow instruction 如果提取的指令是非控制流指令
  - Next Fetch PC is the address of the next-sequential instruction PC中是下一个顺序执行的指令地址
  - Easy to determine if we know the size of the fetched instruction 如果我们知道指令的大小的话，很容易确定

# Control Dependence -2 控制相关性2

- If the instruction that is fetched is a control-flow instruction:

  如果取到的指令是控制流指令

  - How do we get the correct next instruction address from the PC?

    我们如何从PC中得到的正确的下一个指令地址?

- In fact, how do we know whether or not the fetched instruction is a control-flow instruction? 事实上，我们怎么知道获取的指令是不是控制流指令?
- It is difficult for us to get the real address of the next instruction from the PC PC中不一定是真正的下一个指令地址

# How to solve?

- **Several approaches have been taken 采用了多种方法**
  - Multiple Streams 多指令流
  - Prefetch Branch Target 预取分支目标
  - Loop buffer 环形缓存
  - Branch prediction 分支预测
  - Delayed branching 延迟分支

# Multiple streams 多指令流

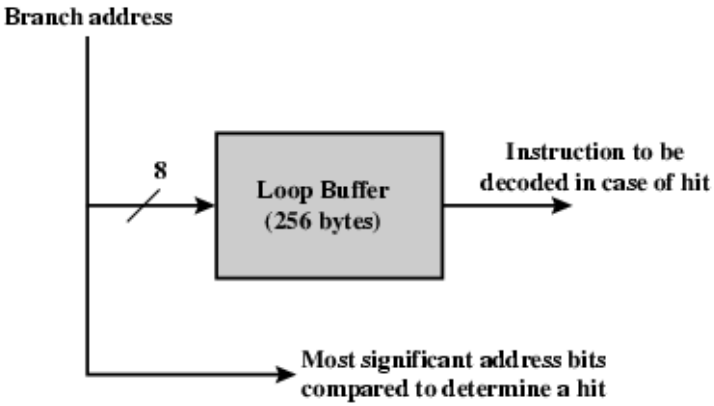
- Have two pipelines for each branch 有两个流水线给每个分支
  - Prefetch each branch into a separate pipeline 在每个独立的流水线中预取一个分支的指令
- Finally, determine which pipeline to use according to the branching conditions 最后根据分支条件确定使用哪一个流水线
- Shortcoming 缺点
  - Leads to bus & register contention 带来总线或寄存器竞争
  - Multiple branches lead to further pipelines being needed 多个分支会导致更多的流水线

# Prefetch branch target 预取分支目标

- Target of branch is prefetched in addition to instructions following branch 除了分支之后的指令外，还预取了分支的目标
- It is not executed after prefetching, but fetching and decoding 预取之后并不执行，只是进行取指和解码
- Keep target until branch is executed 保留目标直到分支被执行
- It can save the time of fetching and decoding 能够节省取指和解码的时间
- Used by IBM 360/91

# Loop buffer 循环缓冲器

- Very fast memory 快速的存储器
- Contains n instructions taken in the most recent order 包含n条最近顺序取过来的指令
- When a branch may occur, first check whether the transfer target is in the buffer 在转移可能要发生的时候，先检查转移目标是否在缓冲器里面
- Very good for small loops or jumps 对于小的循环或跳转有效



# Branch prediction 分支预测

- There are two types of branch predictions 两种分支预测
  - Static branch predictions: the branch does not depend on the execution history 静态分支预测：不依赖于执行历史
  - Dynamic branch prediction: the branch depends on the execution history 动态预测：依赖于执行历史
- Static branching 静态预测
  - Predict never taken 预测从不发生
  - Predict always taken 预测总是发生
  - Predict by Opcode 根据操作码预测
- Dynamic branching 动态预测
  - Taken/not taken switch 发生/不发生转换
  - Branch history table 分支历史表

# Static branch prediction – 1 静态预测1

- Predict never taken 预测从不发生
  - Assume that jump will not happen 假定跳转不会发生
  - Always fetch next instruction 总是取下一个指令
  - 68020 & VAX 11/780
- Predict always taken 假定总是发生
  - Assume that jump will happen 假定跳转会发生
  - Always fetch target instruction 总是取目标指令
- “Predict always taken ” are most used 假定总是发生用的更多

# Static branch prediction – 2 静态预测2

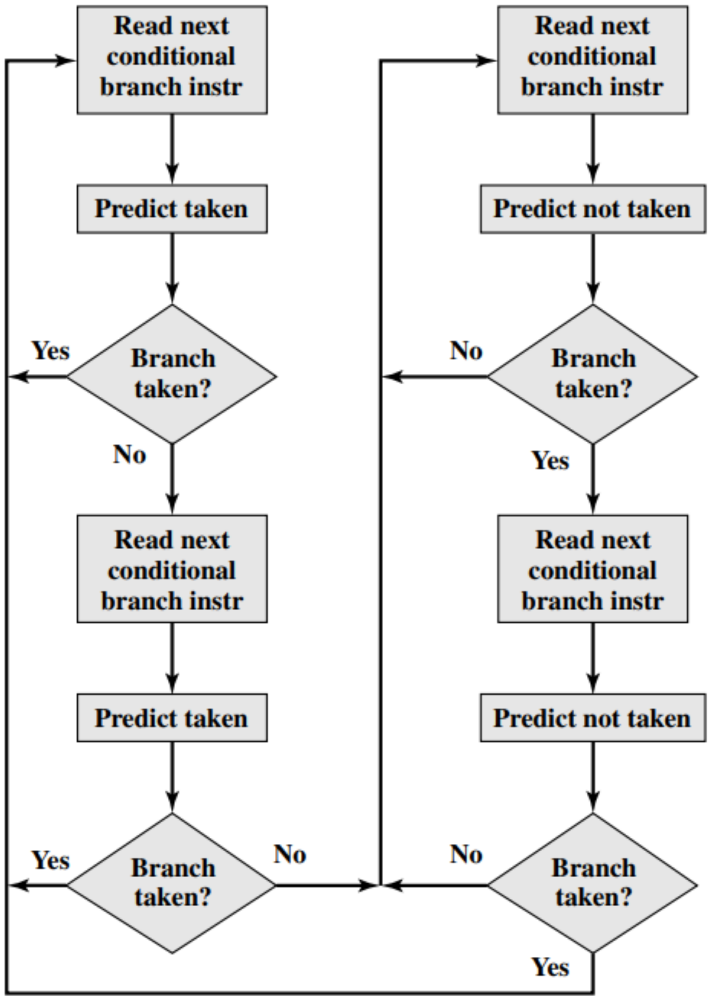
- Predict by Opcode 根据操作码预测
  - Some instructions are more likely to result in a jump than others

    有些指令跳转的可能性会更大一些
  - Can get up to 75% success 能够得到75%的成功率

# Dynamic branch prediction 动态分支预测

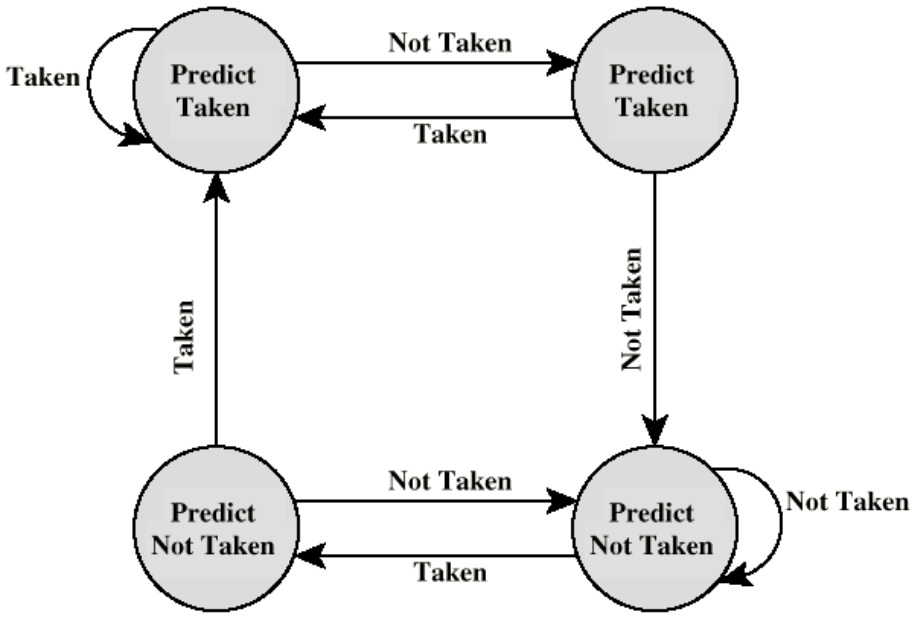
- Record the history of conditional branch instructions in a program 记录条件转移指令在程序中的历史情况
- Taken/not taken switch: One or more bits are used to indicate recent history of the instruction 发生/不发的切换：用一个或多个位用于指示指令的最近历史记录
  - The branching decision is depended on these bits 分支决定依赖于这些位
  - Based on previous history 基于历史
  - Good for loops 对于循环比较适合

# Taken/not taken switch flowchart 转移开关流程图




- 用2位来记录转移情况
- 首先预测是发生，如果实际发生了，继续往下
- 如果转移没有发生，下次还是预测发生，如果确实发生了，那么保留在预测发生的这部分
- 只有当第二次还是没有发生，就走到流程图的右半部分
- 这边都是预测不发生。如果连续两次都发生了转移，那么就又回到左边。

# Taken/not taken switch state diagram 状态图




- 状态转移图
- 上面两个状态都预测发生，下面2个状态都预测不发生
- 如果连续两次的预测都不正确，就会进入到另一个状态

# Branch history table 转移历史表

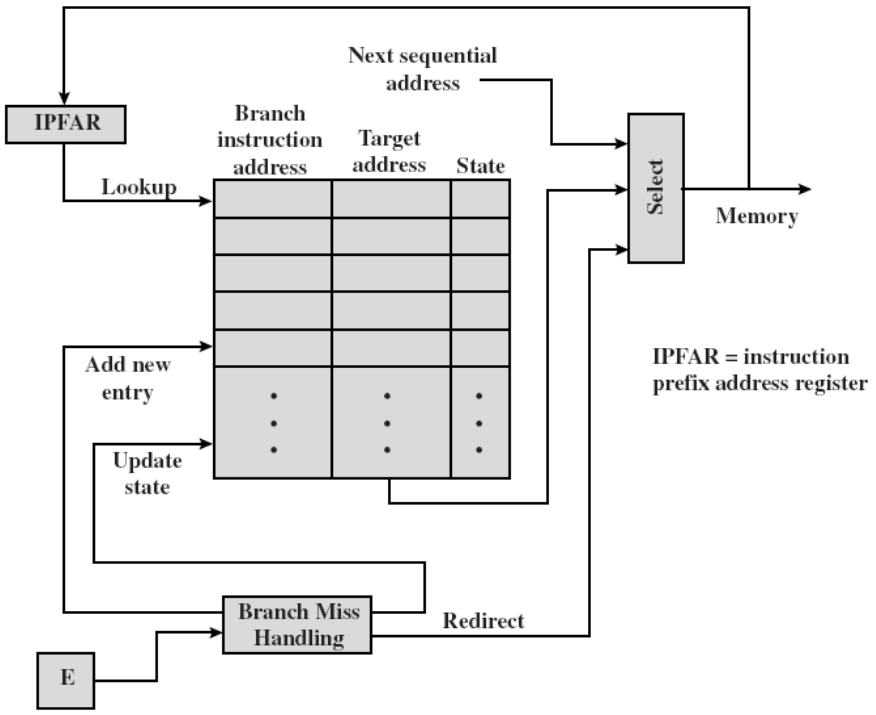
- If Predict branch，target address can only be obtained by decoding instructions 预测转移，需要解析指令才能得到目标地址
  - A waiting time is required 需要一个等待时间
  - How to improve efficiency? 那如何提高效率呢?
- A storage area called branch target buffer is designed 设计了一种叫做分支目标缓冲器的存储区域
  - Also called branch history table 也叫做转移历史表
  - It records information related to branch transfer, including branch instruction address, transfer history bit, and target address information 记录了和分支转移相关的信息，包括分支指令地址、转移历史位，以及目标地址信息

# Branch history table 转移历史表

- Target address information 目标地址信息
  - Can be target instruction 可以是目标指令
    - Use this instruction directly 直接用这个指令
    - Less time 时间上更省
    - It will take up more space 占用的空间会大一些
  - Can be the target instruction address 可以是目标指令地址
    - Less space 可以省空间
    - More time 但是耗时间
  - Whether to save time or space depends on the specific situation

    是省时间，还是省空间，需要根据具体情况来定

# Branch history table 转移历史表




- 预测转移后，指令预取的时候，先去转移历史表中查询
  - 如果有的话，根据指令状态进行预测，可能是目标地址，或者是下一顺序地址
  - 如果不匹配，顺序取下一个指令
- 分支指令执行时，根据实际是否发生了转移，更新转移历史表中的状态位
- 如果条件分支指令不在表中的时候，需要把指令加到这个表中，同时需要替换到当前表中的一项。替换方法可以采用很多种方法，类似于cache的替换策略
- 转移历史表动态自动维护

# Correlation-based prediction 基于关联的预测

- The execution effect of the branch history table in the loop statement is good 转移历史表在循环语句中的执行效果不错
- In more complex structures, branch instruction directly correlates with that of related branches instruction 在更复杂的结构中，分支指令和其他相关的分支指令也有关系
- A method called Correlation-based branch history is proposed 提出了一种叫做基于关联的转移历史方法
  - Create a global branch history table 建立一张全局的转移历史表
  - Predict by combining global and current branch instructions 通过将全局和当前分支指令进行结合，来进行预测

# Delayed Branch 延迟分支

- A method of instruction rearrangement 一种指令重排的方法
- Delayed branches need to calculate the impact of branches and determine which instructions are not affected before prefetching unwanted instructions 延迟分支需要在预取不需要的指令之前，计算分支带来的影响，确定哪些指令不受影响
  - Execute such an instruction immediately after the branch instruction 在分支指令之后立即执行这样一个指令
  - The execution of this instruction keeps the pipeline in a full rotation state, and the clock cycle will not be wasted due to waiting 通过这个指令的执行保持流水线处于满转的状态，不会因为等待而浪费时钟周期
- Discussion later 后面详细讨论

# Example：80486 pipelining 80486的流水线

- The 80486 implements a five-stage pipeline 5阶段流水线
- Stage 1: Fetch from cache/memory and put in prefetch buffers
  从cache或内存中取指令，放到预取缓冲器中
  - 80486 has variable instruction length 80486的指令长度可变
  - Two 16 byte prefetch buffers are set 设置了2个16字节的预取缓冲器
  - The target of fetching is to fill the prefetch buffer 取指的目标是要把预取缓冲器放满
  - As soon as the instruction enters the decoder, take the next instruction immediately 只要指令进入到了译码器，立即取下一个指令

# Decode 译码阶段

- Decode stage is divided into two stages 译码阶段分成了2个阶段
- Stage 2: Decode stage 1 译码1
  - Opcode & address-mode information are decoded 操作码和寻址方式的译码
  - These information are passed to D2 stage 传递到D2阶段
- Stage 3: Decode stage 2 译码2
  - Converts opcode into control signals for ALU 将操作码转成控制信号给ALU
  - Control the computation of complex address modes（indirect addressing） 复杂的寻址方式的计算（间接寻址）

# Execute and Write back 执行和写回

- Stage4：Execute 执行
  - Access cache Cache 访问
  - ALU operation ALU运算
  - Register modification 寄存器修改
- Stage5：Write back 写回
  - Setting of status flags 状态标志的设置
  - Cache update or memory write back Cache的更新或存储器的回写

# Instruction flow 指令流

(a)


(b)


- （a）图上的指令流中，三个指令可以完全在流水线上并行运行
- （b）图下面的指令流中，因为reg1需要作为第二个指令的间接寻址的存储器地址，所以需要有一个时钟周期的延时。
- D2不需要等WB完成之后才进行，是因为采用了旁路信号，允许执行结果直接给D2解码，可以节省一个周期的时间。

# Branch instruction 分支指令


- 比较Reg1和Imm，根据比较情况来确定转移
- 跳转指令的执行阶段，和比较指令的回写阶段可以同时进行，也是因为有一个旁路信号允许这2个阶段可以同时使用条件码
- 分支指令的执行阶段，处理器会并行地预取跳转的目标指令。如果跳转成功，那么就执行这个指令。如果跳转不成功，继续执行顺序指令。顺序指令早已经在预取缓冲器中了。这里又能够节省一个时钟周期。

# Summary-Instruction pipeline 指令流水线小结

- Why use instruction pipeline 为什么采用流水线
- Instruction pipeline execution process 指令流水线执行过程
- Problems in the instruction pipeline (Hazard) 指令流水线中存在的问题（冒险）
  - Resource hazard 资源冒险
  - Data hazard 数据冒险
  - Control hazard 控制冒险
- Ways to solve hazard 解决冒险的方法

# Key Terms

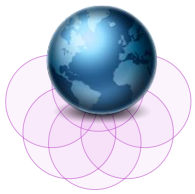
| | | | |
|---|---|---|---|
| Branch prediction | Delayed branch | Instruction cycle | Instruction prefetch |
| Condition code | flag | Instruction pipeline | Program status word (PSW) |

# Summary and Question

- 小结
  - 对CPU的内部结构进行了分析
  - 对指令周期的数据流进行了分析
  - 对指令流水线、流水线冒险等进行了详细讨论
- 问题
  - 问题1：简述取指过程中的数据流
  - 问题2：指令流水线和理想的流水线的差距表现在哪几个方面？
  - 流水线冒险包括哪几类？

# Assignments

- Review Questions
  - 12.1~12.6
- Problems
  - 12.1 12.3 12.8 12.9



# 谢谢大家！