北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Computer Organization and Architecture

知识点梳理（二）

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

第九章　计算机算术

- 基本概念
  - ALU
  - 溢出

第九章　计算机算术

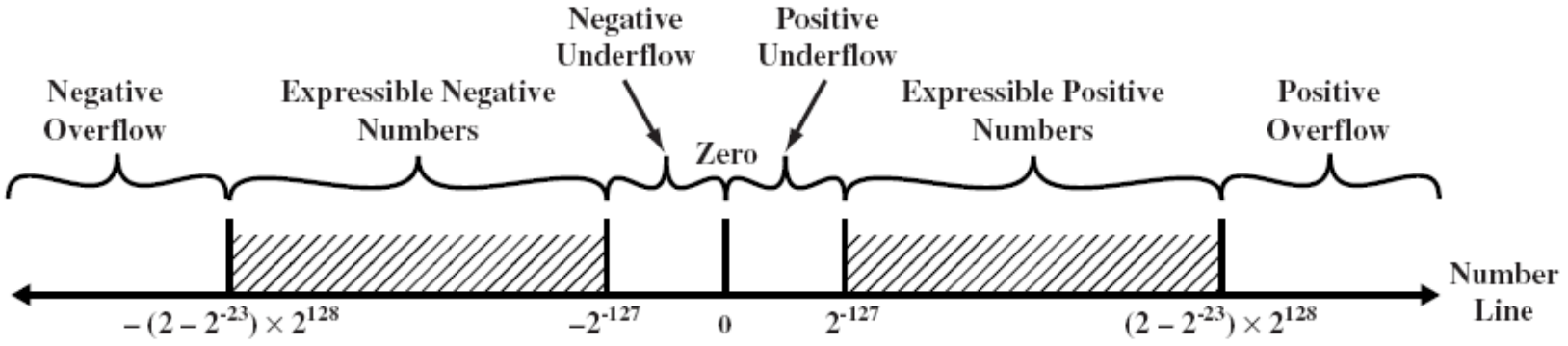
- 重点知识点
  - 数的各种表示形式，包括无符号、符号-幅值、反码、补码
  - 整数加减法的计算
  - 整数乘除法的计算
  - 浮点数表示法
  - 浮点数加减法的计算
  - 浮点数乘除法的计算

基本概念

- ALU
  - arithmetic logic unit，算术逻辑单元
  - 负责算术和逻辑运算
  - 整个计算的核心，其他部件都是为它服务
  - 主要是对整数进行计算，也可以对浮点数进行运算，也有专门的浮点运算单元，floating point unit，可以处理浮点数了，甚至有专门的处理浮点数的芯片

基本概念

- 溢出
  - 对于整数，溢出表示的是超过最大值或最小值
  - 对于浮点数，存在上溢出和下溢出
    - 上溢出：正数太大，或负数太小
    - 下溢出：负数太大，或正数太小



重点知识点

- 二进制和十进制的相互转换
  - 十进制转换到二进制
    - 十进制整数转换为二进制的方法是不停地除以2，并记录每次的余数，然后将所有的余数从小数点开始，往左写
    - 十进制小数转换为二进制的方法是不停地乘以2，并记录每次的整数项，然后将所有的整数项从小数点开始，往右写
  - 二进制转换到十进制
    - 整数部分：从小数点开始，往左，用每一位的值乘以它的权值，越往左，权值越大。往左一位，权值乘以2
    - 小数部分：从小数点开始，往右，也是用每一位的值乘以它的权值。越往右，权值越小。往右一位，权值除以2

重点知识点

- 二进制和十进制的相互转换举例

$(1101001.011)_2 =$

$= (2^6 + 2^5 + 2^3 + 2^0) . (2^{-2} + 2^{-3})$

$= (64 + 32 + 8 + 1) . (0.25 + 0.125)$

$= 105.375$ -------- **$(105.4)_{10}$**

重点知识点

- 整数的无符号表示法
  - 没有符号，只有非负值
  - N位值的范围：0 to 2^n - 1

重点知识点

- 整数的符号—幅值表示法
  - 最左边的是符号位，0表示整数，1表示负数
  - N位值的范围-（$2^{n-1}-1$）~ （$2^{n-1}-1$）
  - 0有两种表示法：+0和-0

重点知识点

- 整数的反码表示法　one’ s complement
  - 曾经很重要。但是现在用的不多，主要是用它来得到数的补码
  - 正整数使用和无符号相同的表示
  - 负数通过按位取反得到
  - N位值的范围-（$2^{n-1}-1$）~ （$2^{n-1}-1$）
  - 0有两种表示法：0000 0000和1111 1111（以8位为例）

重点知识点

- 整数的补码表示法 two’s complement
  - 反码的变种
  - 正整数使用和无符号相同的表示
  - 负数通过按位取反，然后+1得到
  - N位值的范围-2^{n-1} ~ （$2^{n-1}-1$）
  - 0只有一种表示法：0000 0000

重点知识点

- 表示法举例：1010 1001

  如果是无符号整数，直接计算就可以了，他的值是：

  $10101001=2^7+2^5+2^3+2^0=128+32+8+1=169$

  如果是符号-幅值表示法，那么第一个1表示它是负数，所以：

  $10101001=-（2^5+2^3+2^0）=-（32+8+1）=-41$

  如果是反码表示法，那么它的值是：

  $-1010110=-（2^6+2^4+2^2+2^1）=-(64+16+4+2)=-86$

  如果是补码表示法，那么它的值是：

  $-（（1010110）_2+1）=-（2^6+2^4+2^2+2^1+1）=-87$

  如果是用偏移量表示，那么它的值就是用当前值减去偏移量，所以：

  $10101001=（10101001-01111111）_2=（00101010）_2=42$

重点知识点

- 符号位扩展方法
  - 符号-幅值表示法：最高位加上符号位，然后补0
  - 补码表示法：正数补0，负数补1

重点知识点

- 整数取反操作
  - 符号-幅值表示法：翻转符号位
  - 补码表示法：包括符号位按位取反，然后作为无符号整数，加1

重点知识点

- 整数加减法
  - 加法
    - 正常二进制加法
    - 需要检查溢出
  - 减法
    - 得到减数的补码数
    - 加到被减数中
  - 加法溢出判断
    - 两个同号的数相加，符号位相异

重点知识点

- 整数乘除法
  - 乘法
    - 类似于竖式乘法，结果需要2倍长度存储
    - 对于补码，用布斯乘法可以直接相乘，结果也是补码形式
  - 除法
    - 长除法

重点知识点

- 整数乘法

```
      1011      Multiplicand (11 dec)
    x 1101      Multiplier   (13 dec)
-------------
      1011      Partial products
     0000
    1011
   1011
-------------
  10001111      Product (143 dec)
```

Note: if multiplier bit is 1 copy multiplicand (place value),otherwise zero

Note: need double length result

重点知识点

- 浮点表示法
  - 第一个部分是符号，用0或者1表示这个数的符号，一般0表示正数，1表示负数
  - 用移码表示的指数
  - 有效值。就是这个数的尾数部分。规范化的有效值的最高位是1，默认可以忽略
  - 小数点是在有效值的最高位的右边第一位。

重点知识点

- 浮点表示法举例

- $-1.1010001 \times 2^{10100}$:
- Sign: 1
- Exponent: 0111 1111 + 10100 = 1001 0011
- Significand: 101 0001 0000 0000 0000 0000
- The floating point representation for above number:
- 1 10010011 10100010000000000000000

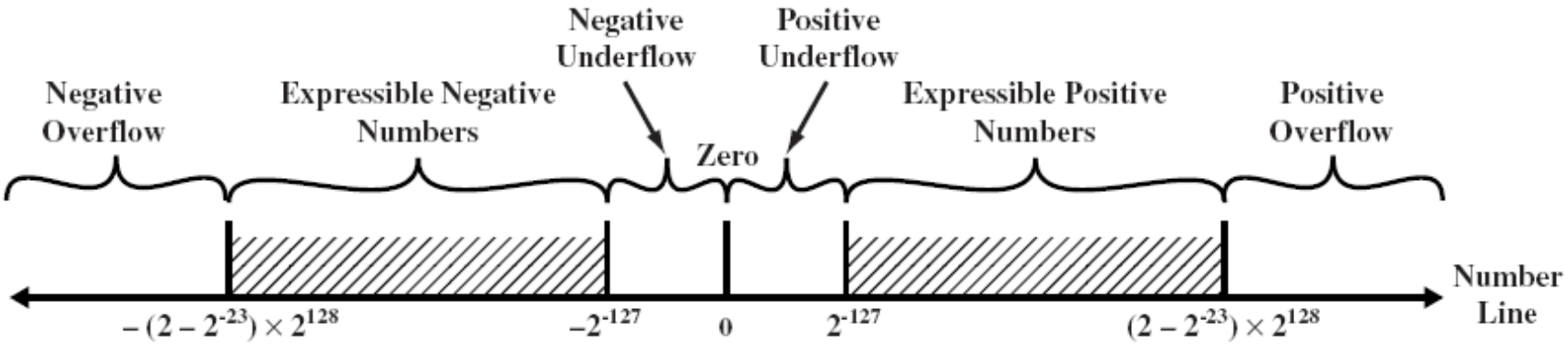
重点知识点

- 浮点数表示范围

- The smallest significand: 1.0 最小的尾数
- The biggest significand=1.11111……= 2- 2^{-23} 最大的尾数
- The smallest exponent = -127 最小的指数
- The biggest exponent = 128 最大的指数
- Negative numbers between: -(2- 2^{-23}) x 2^{128} and - 2^{-127} 负数
- Positive numbers between: 2^{-127} and (2- 2^{-23}) x 2^{128} 正数

重点知识点

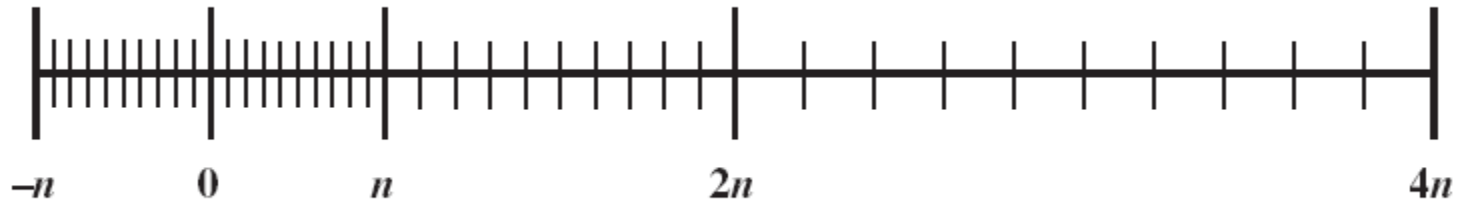
- 浮点数表示范围



- When > $(2-2^{-23})x2^{128}$ or < $-(2-2^{-23})x2^{128}$
  - **Overflow!** 上溢出！
- When value >0, < $(2-2^{-23})x2^{-127}$ or <0, > $(2-2^{-23})x2^{-127}$
  - **Underflow!** 下溢出！

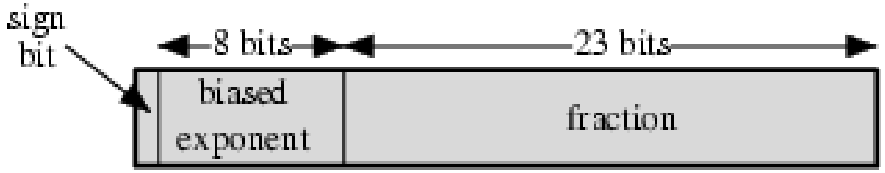
重点知识点

- 浮点数的精度
  - 对于定点数，在空间上是均匀的
  - 浮点数在空间上是不均匀的
  - 数越来越大，数之间的距离越来越大

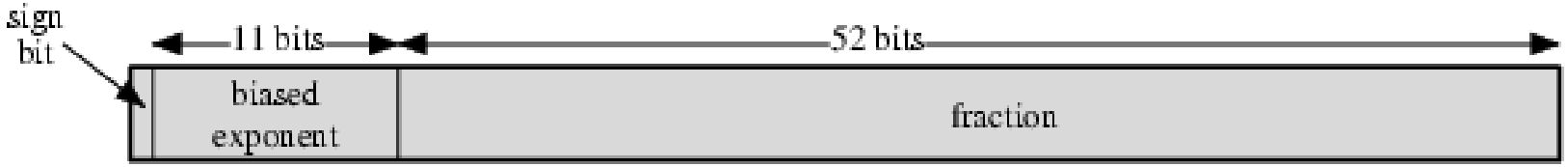


重点知识点

- 单精度和双精度



(a) Single format



(b) Double format

重点知识点

- 浮点数的加减
  - 检查是否为0
  - 调整指数，对齐有效位
  - 加减尾数
  - 规格化结果

重点知识点

- 浮点数的乘除
  - 检查是否为0
  - 加/减指数
  - 尾数相乘，考虑符号位
  - 规格化
  - 四舍五入
  - 中间结果需要双倍长度

第十章　操作数和操作类型

- 基本概念
  - 机器语言，汇编语言和高级语言
  - 指令集
  - 指令的要素
  - 操作码
  - 指令类型
  - 端序

第十章　操作数和操作类型

- 重点知识点
  - 指令设计的考虑因素
  - 指令包括哪些操作类型：7种

基本概念

- 机器语言、汇编语言、高级语言
  - 机器语言：机器可以直接执行的语言。由0、1组成
  - 汇编语言：符号化的机器语言
  - 高级语言：使用类似于日常语言的语句来编写程序
  - 编译器：将高级语言或者汇编语言翻译成机器语言，然后才能在计算机里进行运行

基本概念

- 指令集
  - CPU能够执行的各种不同指令的集合称为指令集。它是CPU能够理解的全部指令的集合
  - 用机器码表示的指令，指令形式是二进制的，但通常用汇编代码来表示
  - 指令集位于机器的硬件和软件之间，是联结硬件和软件的桥梁

基本概念

- 指令类型
  - 数据处理
  - 数据存储
  - 数据传送
  - 程序流控制

基本概念

- 指令要素
  - 操作码
  - 源操作数
  - 目的操作数
  - 下一个指令

基本概念

- 操作数
  - 地址
  - 数字
  - 字符
  - 逻辑数据

基本概念

- 操作类型
  - 数据传输
  - 算术运算
  - 逻辑运算
  - 转换
  - 输入/输出
  - 系统控制
  - 控制转移

基本概念

- 端序
  - 多字节数据在存储器中的存放顺序
  - 大端端序：将最高的地址位存放最低的数字位
  - 小端端序：最低地址位存放最低的数字位
  - 举例： 12345678的16进制数到地址184开始的存储单元

Address	Value
184	12
185	34
186	56
187	78

Big endian

Address	Value
184	78
185	56
186	34
187	12

Little endian

重点知识点

- 指令中地址的数量
  - 3个地址：2个源操作数地址，1个目的操作数地址
  - 2个地址：一个地址为源操作数和目的操作数复用
  - 1个地址：隐含了第二个操作数地址
  - 0个地址：均为默认，通常使用栈来处理
  - 地址多
    - 指令复杂，功能强大，程序中的指令数量少
  - 地址少
    - 指令相对简单，取指和执行更快

重点知识点

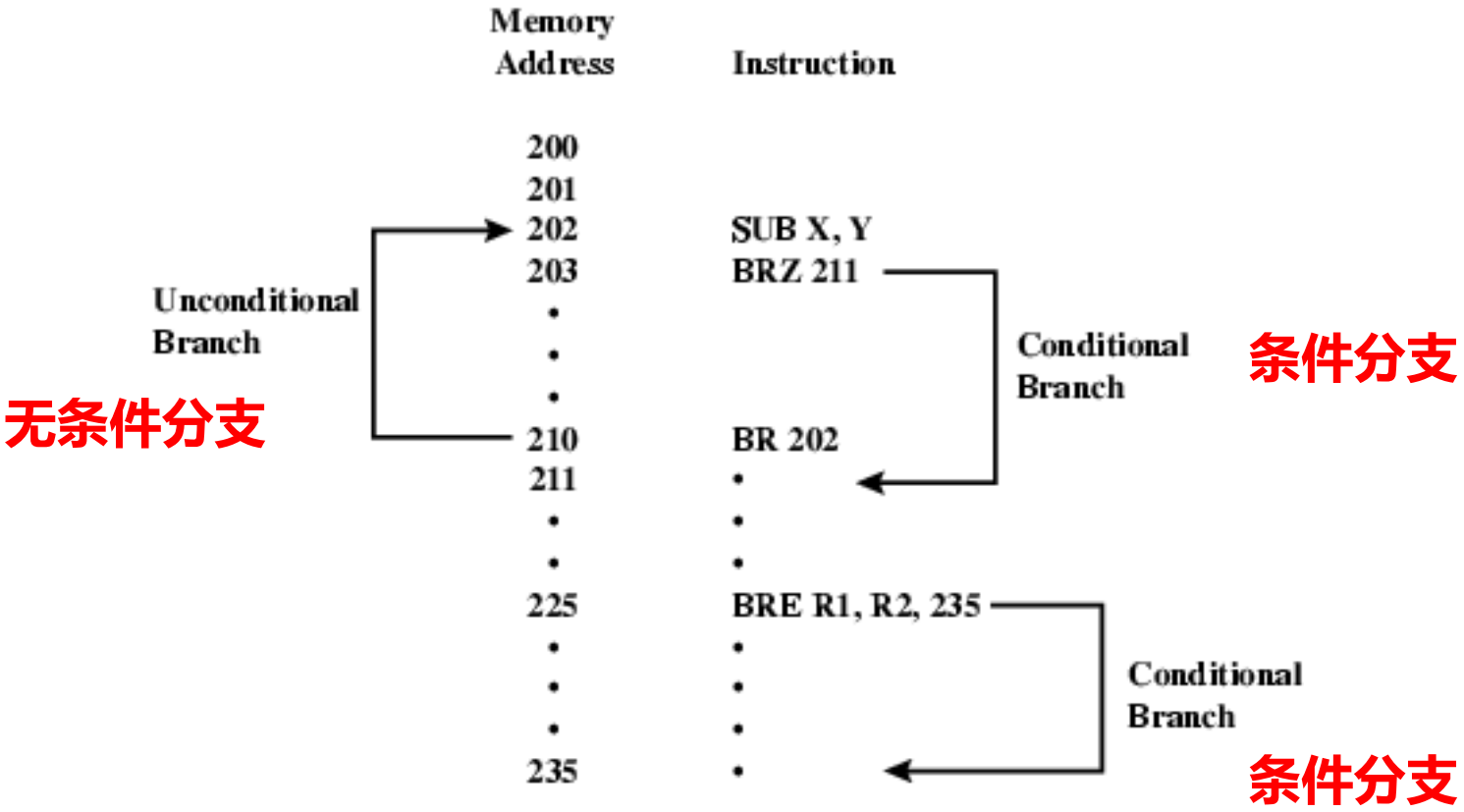
- 指令集设计要素
  - 操作指令表：多少种操作，能完成什么操作，复杂度
  - 数据类型：整型，浮点型，字符型
  - 指令格式，比如长度，地址数量等
  - 寄存器：可以用哪些寄存器
  - 访问内存的方式：直接寻址、间接寻址，等等

重点知识点

- 控制转移指令
  - 分支指令：操作数中包含跳转后的指令地址
  - 跳步指令：跳过下一个需要执行的地址
  - 过程调用指令
    - 调用和返回需要匹配
    - 过程调用可以嵌套
    - 需要保存返回地址，还需要传递参数

重点知识点

- 分支指令


Memory Address
Instruction
200
201
202
SUB X, Y
203
BRZ 211
Unconditional Branch
无条件分支
Conditional Branch
条件分支
210
BR 202
211
225
BRE R1, R2, 235
Conditional Branch
条件分支
235

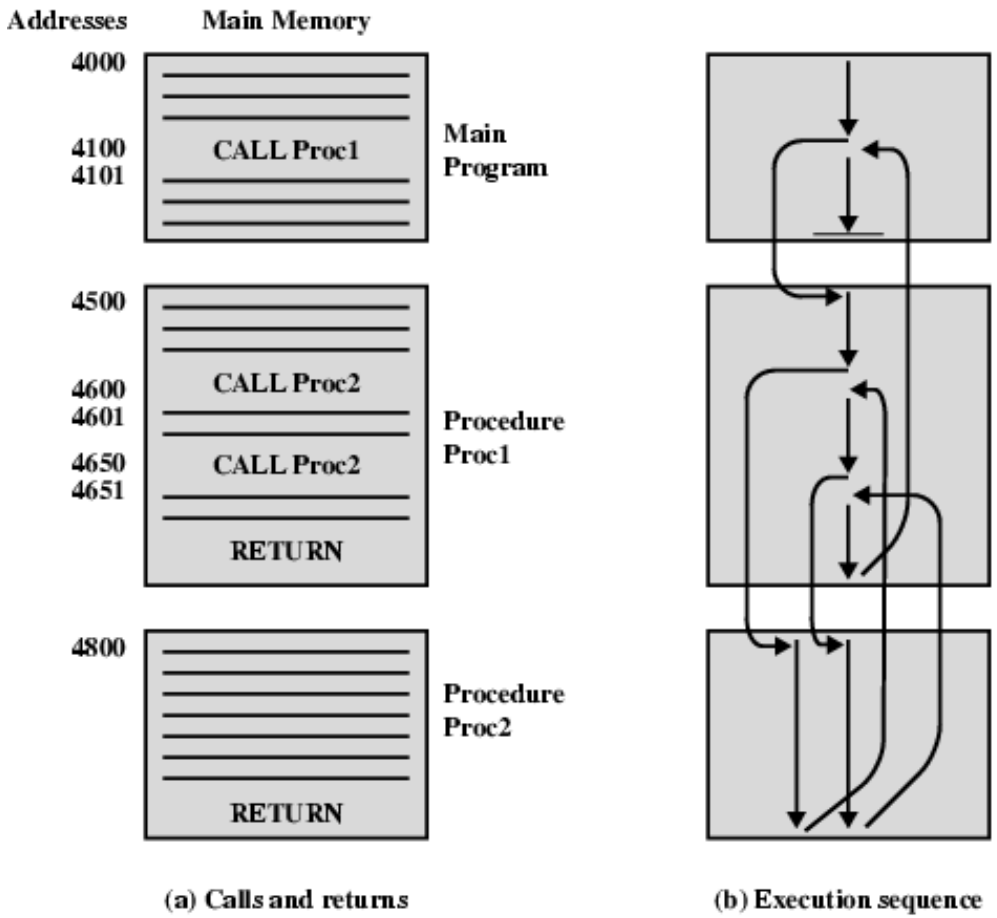
重点知识点

- 过程嵌套

主程序

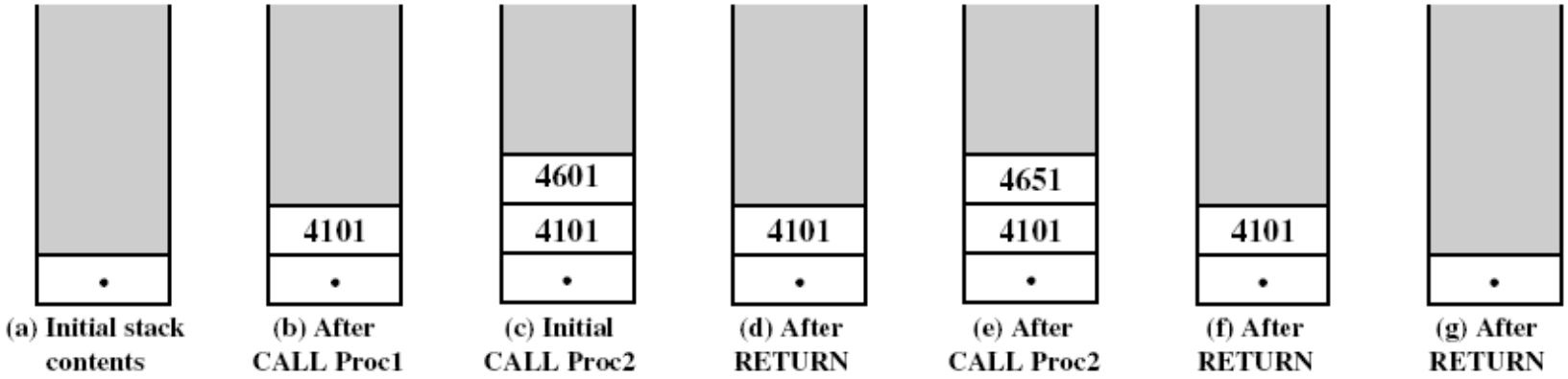
过程1

过程2



重点知识点

- 过程返回地址的记录



(a) Initial stack contents (b) After CALL Proc1 (c) Initial CALL Proc2 (d) After RETURN (e) After CALL Proc2 (f) After RETURN (g) After RETURN

第十一章　寻址模式和指令格式

- 基本概念
  - 寻址模式
  - 立即数寻址
  - 直接寻址
  - 间接寻址
  - 寄存器寻址
  - 寄存器间接寻址
  - 偏移寻址
  - 指令格式

第十一章　寻址模式和指令格式

- 重点知识点
  - 各种寻址模式的地址计算
  - X86的寻址模式
  - ARM的寻址模式
  - 指令位数分配中的考虑因素

基本概念

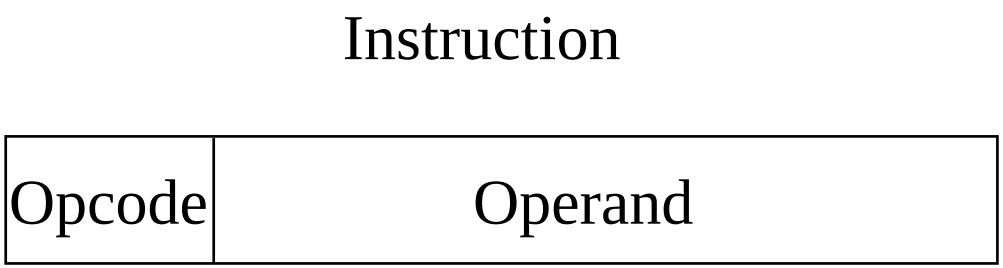
- 寻址模式
  - 确定怎么去获得指令中的操作数
  - 寄存器
  - 立即数
  - 存储器，包括直接寻址、间接寻址、寄存器间接寻址、偏移寻址、堆栈寻址等

基本概念

- 指令格式
  - 指令格式：根据指令的各个组成部分，确定一个指令中每一位的安排。一个指令会包含若干个部分，这些部分各在指令中占用多少个位，先后顺序是怎样
  - 指令必须包含操作码，决定指令做什么
  - 指令还必须包含操作数，可能显式地给出，或者是隐含给出的，决定了操作对象。
  - 大多数的指令集会采用多种指令格式，以满足不同应用场景的要求

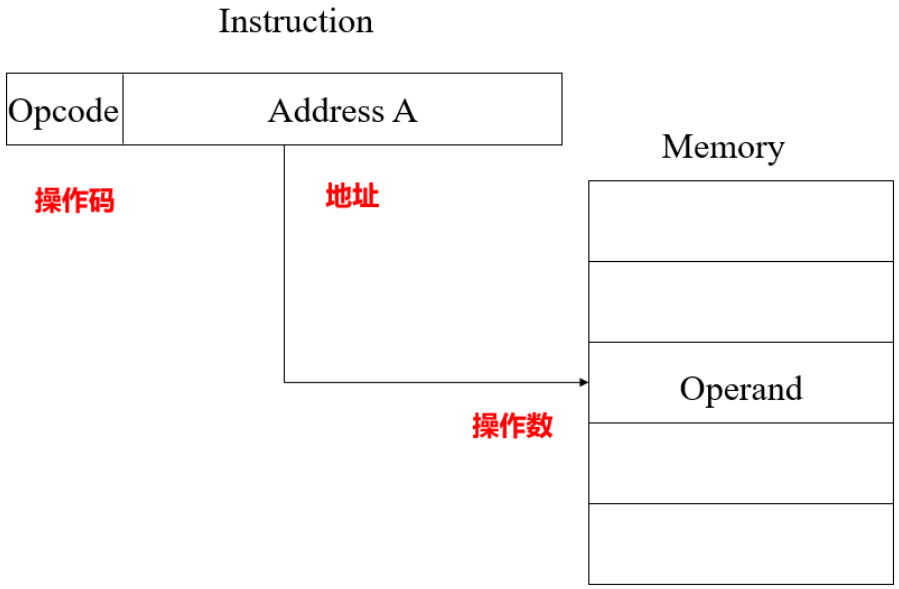
重点知识点

- 立即数寻址
  - 操作数在指令中的地址域部分
  - 不需要到内存中取数
  - 速度快，但不灵活，并且值域有限



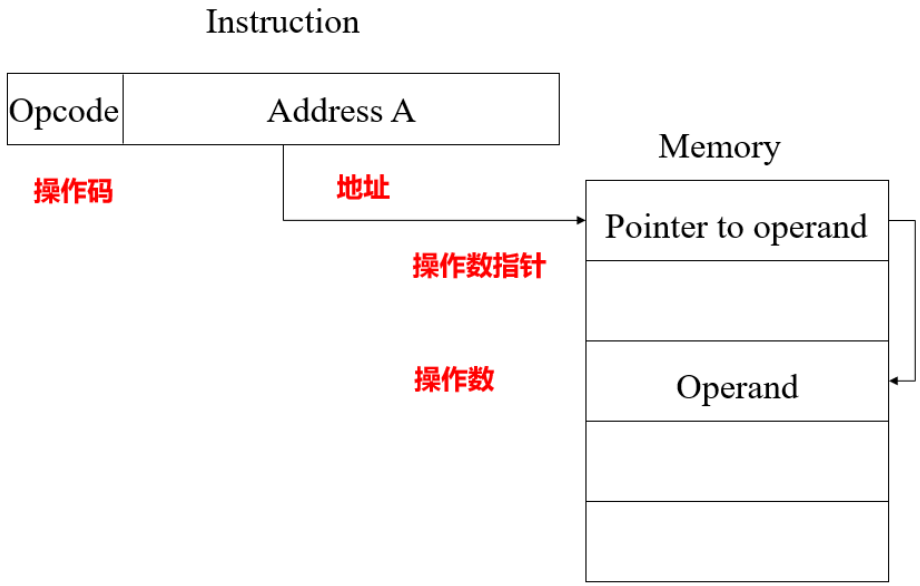
重点知识点

- 直接寻址
  - 操作数的地址在指令中的地址域部分， EA=A
  - 需要访问一次内存取数
  - 地址空间受到指令中地址域字段长度的限制



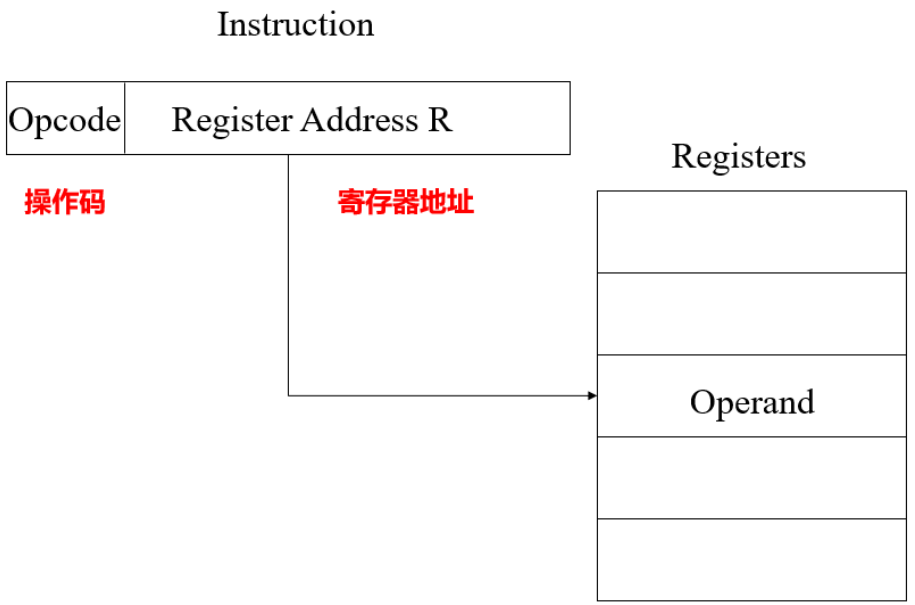
重点知识点

- 间接寻址
  - 操作数的地址在指令中的地址域对应的内存单元里， EA=（A）
  - 需要访问两次内存取数
  - 地址空间大，受到内存字长的限制
  - 可多次嵌套间接寻址
  - 速度慢



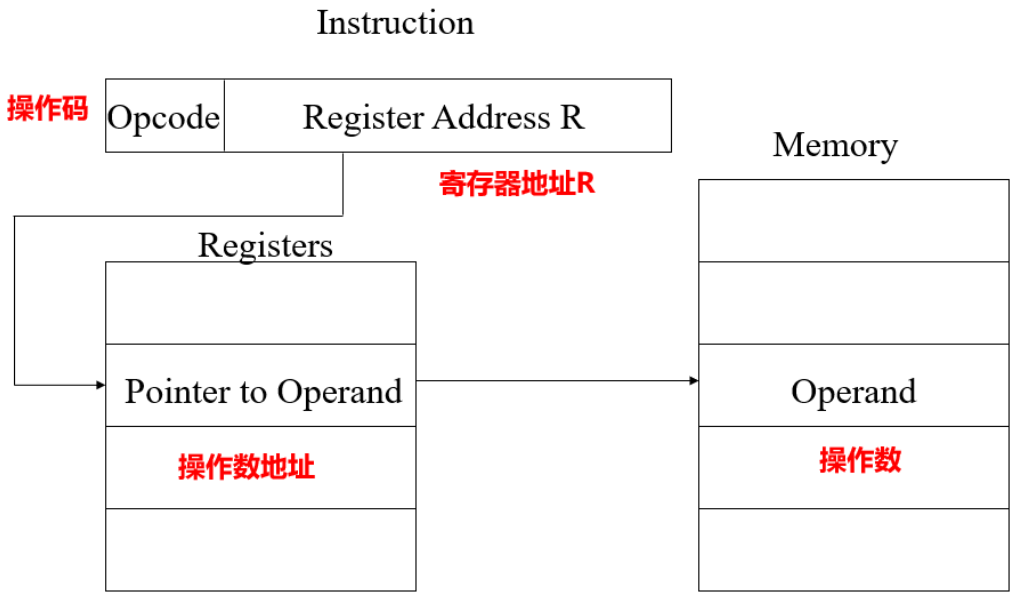
重点知识点

- 寄存器寻址
  - 操作数在地址域给定的寄存器里， EA=R
  - 不需要访问内存
  - 地址空间小，寄存器数量有限
  - 指令短
  - 速度块



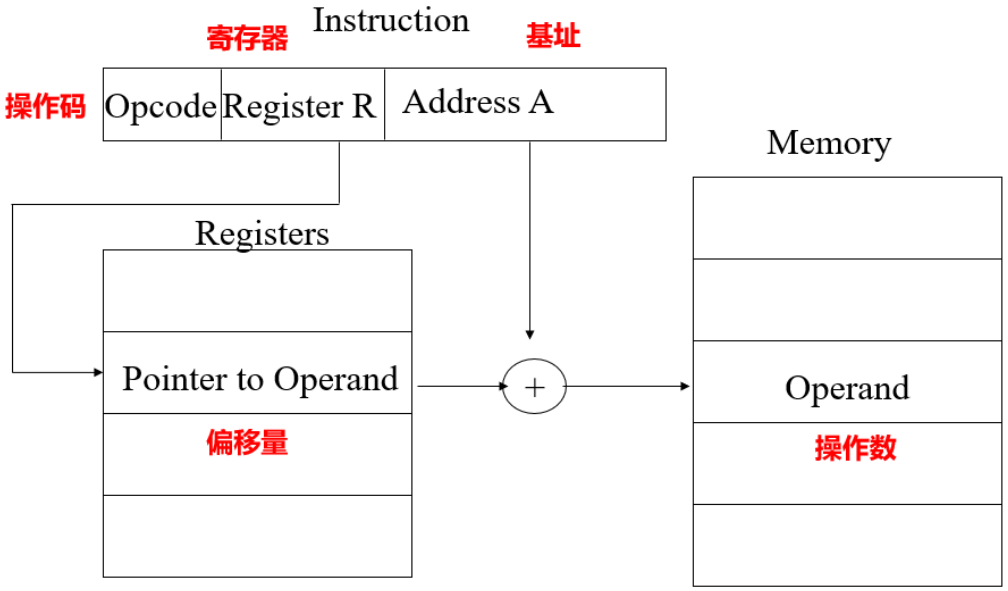
重点知识点

- 寄存器间接寻址
  - 操作数的地址在地址域给定的寄存器里， EA=（R）
  - 需要访问一次内存
  - 地址空间受限于寄存器的字长
  - 指令短



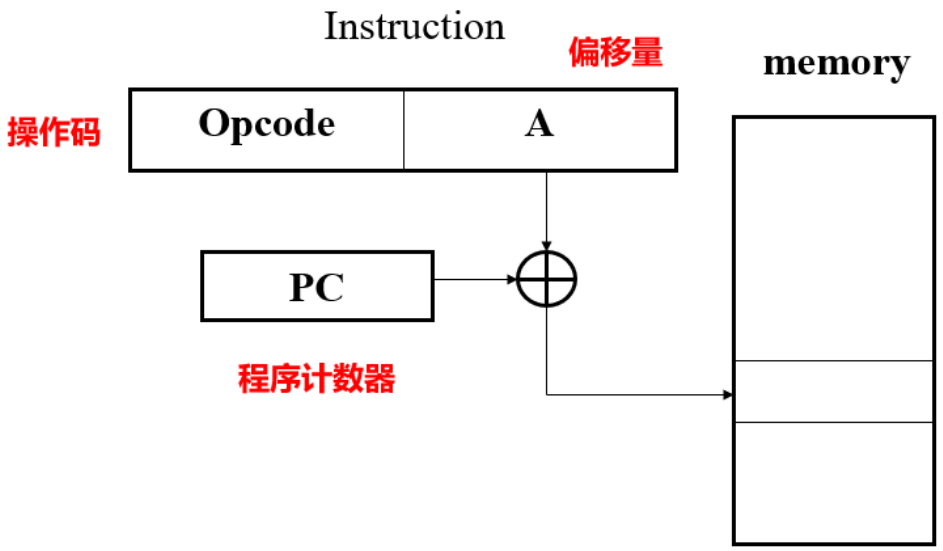
重点知识点

- 偏移寻址
  - EA=A+（R）基址+偏移量
  - 地址域包括基址和偏移量
  - 基址或偏移量可能会在寄存器中
  - 包括相对寻址、基址寄存器寻址、变址寻址



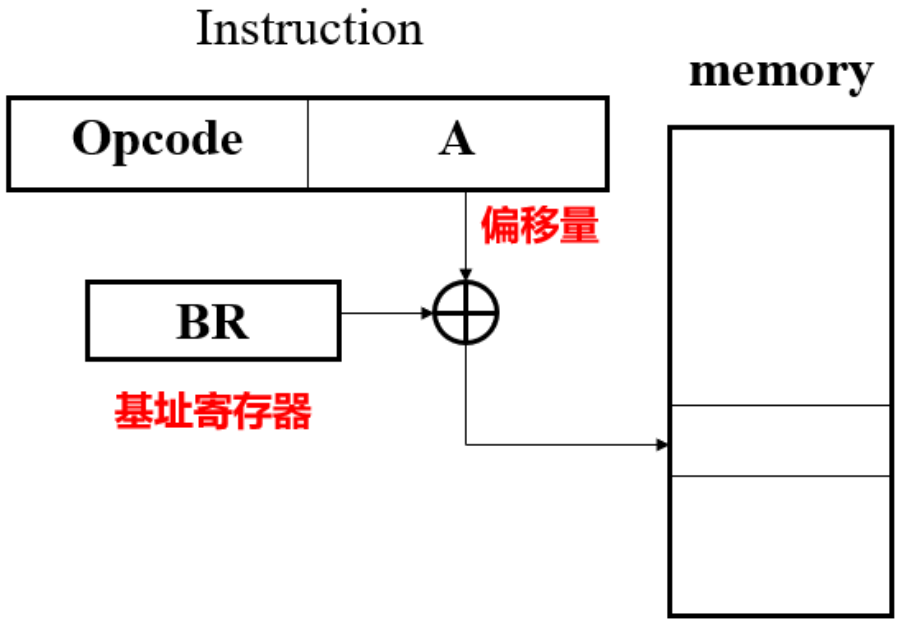
重点知识点

- 相对寻址
  - EA=A+（PC）基址
    +PC的值
  - 偏移寻址的特例
  - 基址或偏移量可能会在
    寄存器中



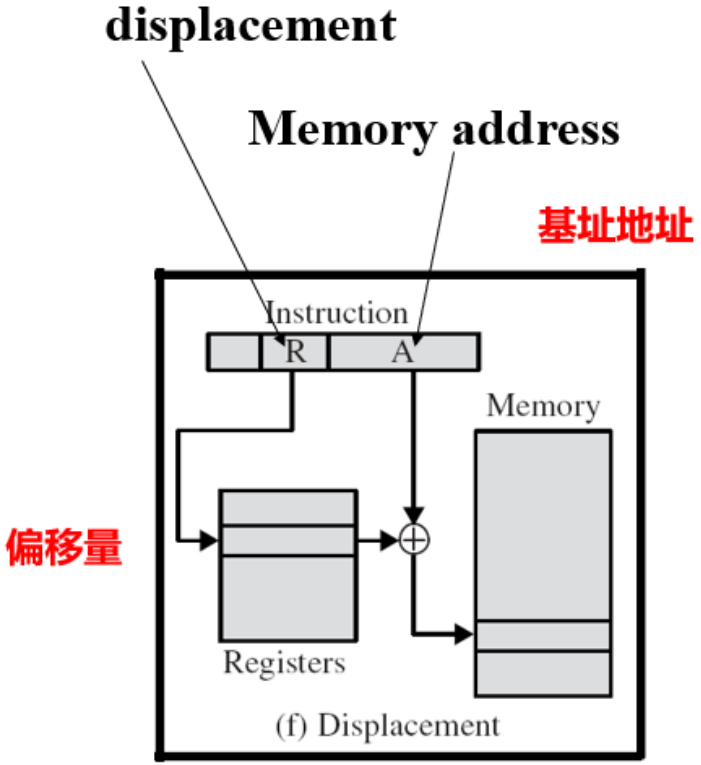
重点知识点

- 基址寄存器寻址
  - EA=（BR）+A
  - BR：基址寄存器
  - 寄存器作为基址



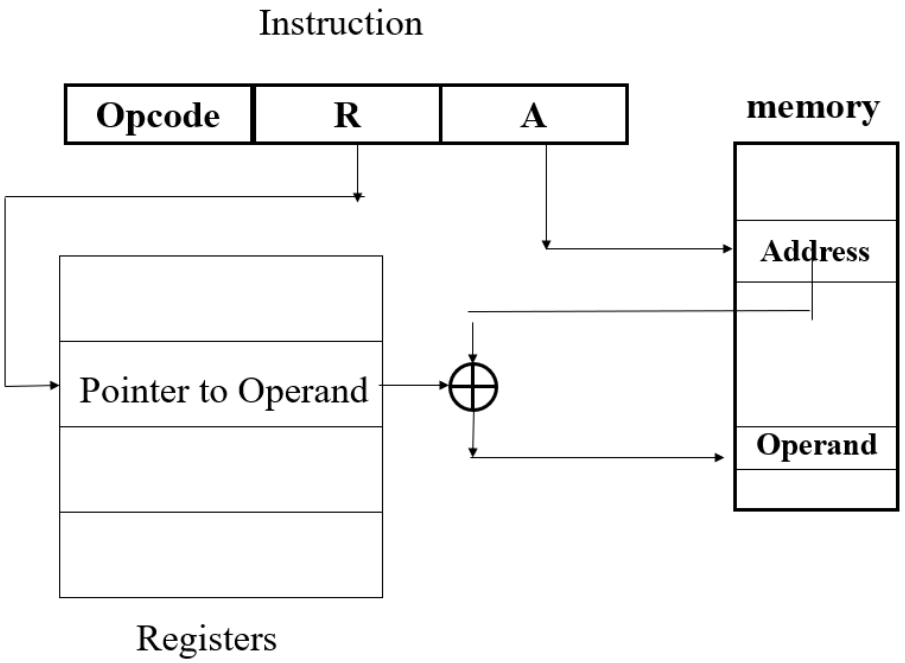
重点知识点

- 变址寻址
  - EA=A+（R）
  - R：变址寄存器
  - 指令中的地址域A作为基址



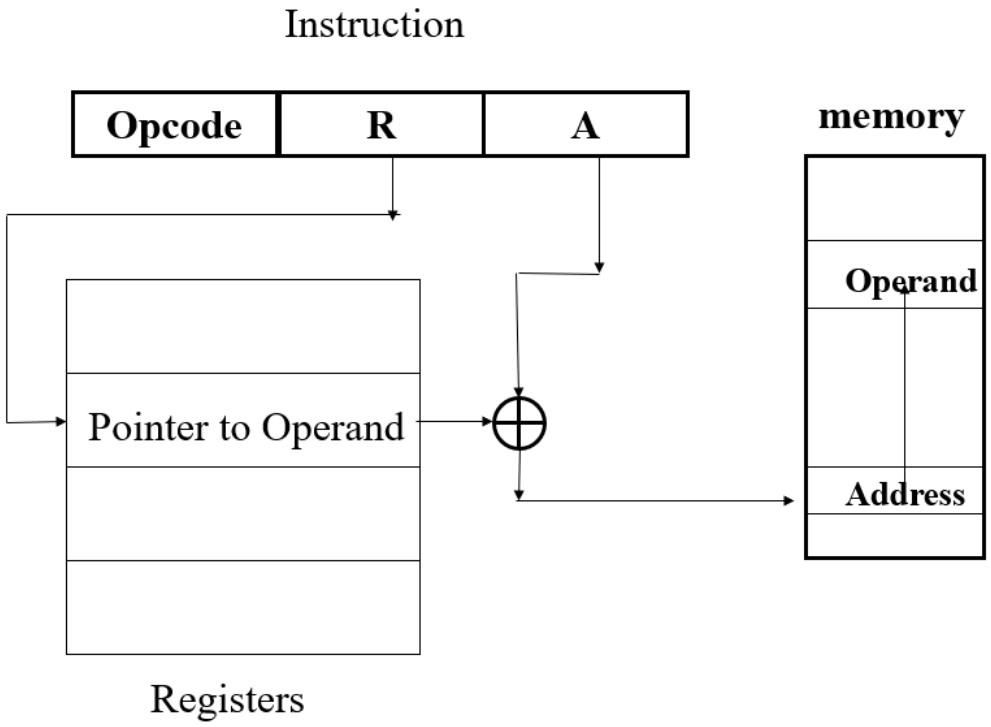
重点知识点

- 后变址寻址
  - EA=（A）+（R）
  - 先间接，后变址



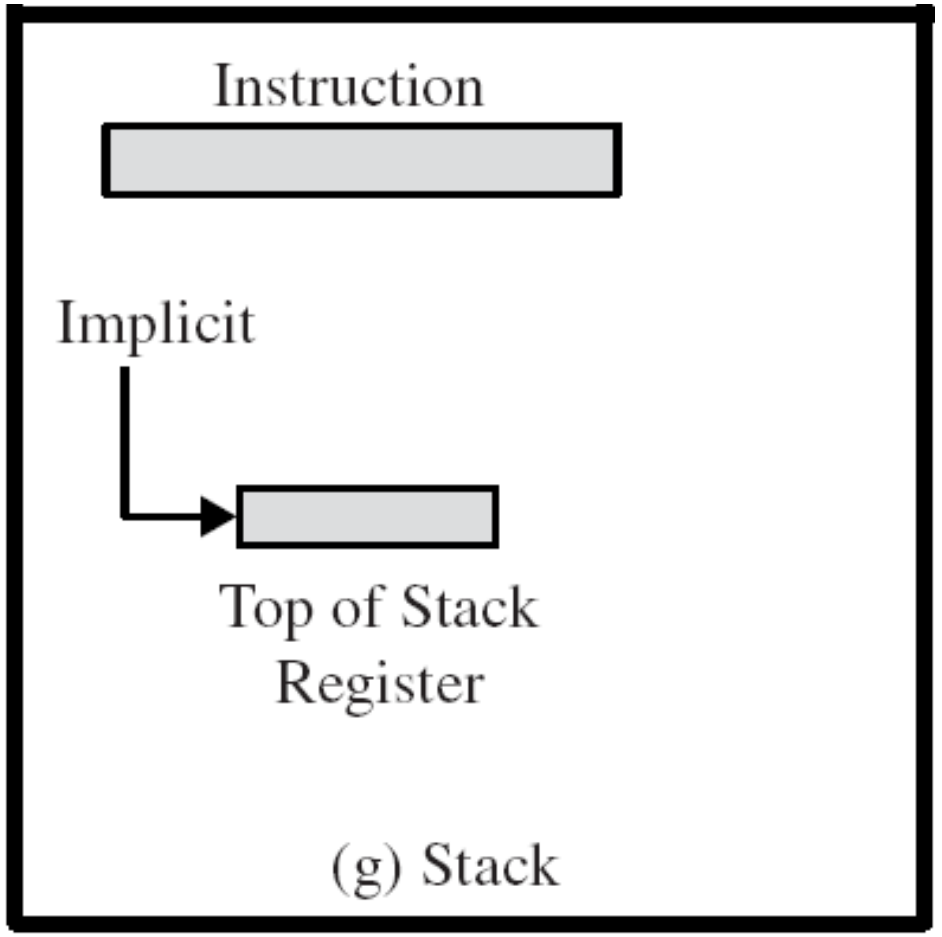
重点知识点

- 前变址寻址
  - EA=（A+（R））
  - 先变址，后间接



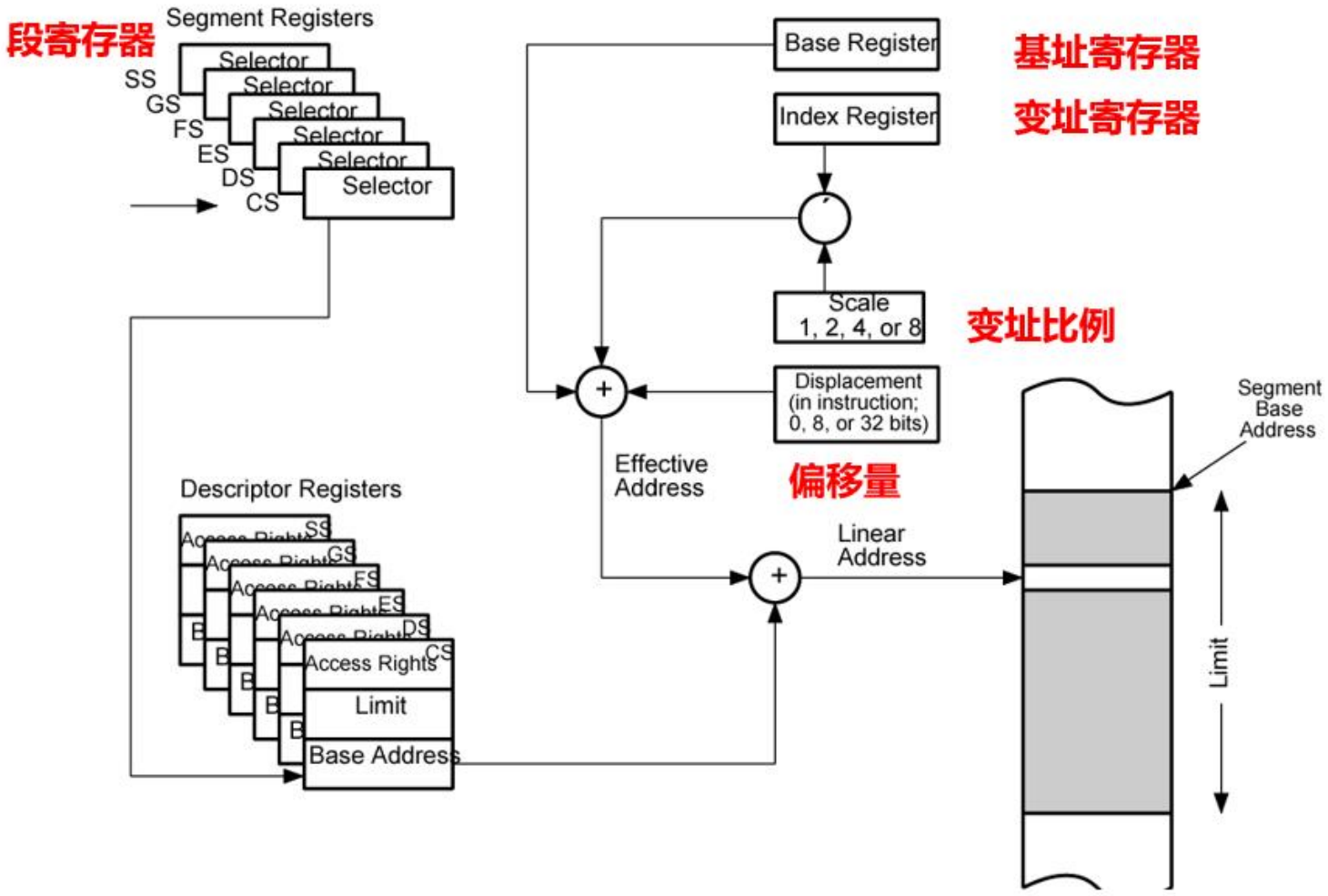
重点知识点

- 栈寻址
  - 操作数在栈顶
  - 没有操作地址



重点知识点

- X86寻址模式



重点知识点

- X86寻址模式

LA: linear address 线性地址

SR : segment Register 段寄存器

B: base register 基址寄存器

I : index register 变址寄存器

S: scale factor 变址比例因子

Mode	Algorithm
Immediate 立即数	Operand=A
Register Operand 寄存器	LA=R

重点知识点

- X86寻址模式

Mode	Algorithm
Displacement 偏移量	LA=(SR)+A
Base 基址	LA=(SR)+(B)
Base with Displacement 基址带偏移量	LA=(SR)+(B)+A
Scaled Index with Displacement 比例变址带偏移	LA=(SR)+(I)*S+A

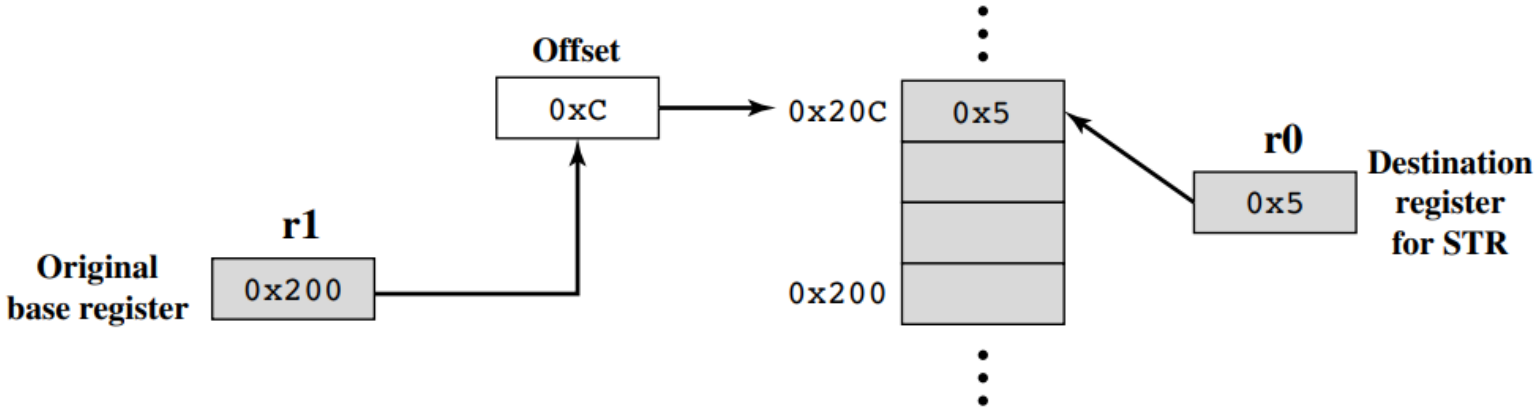
重点知识点

- X86寻址模式

Mode	Algorithm
Base with Index and Displacement 基址变址带偏移	LA=(SR)+(B)+(I)+A
Base with Scaled Index and Displacement 基址比例变址带偏移	LA=(SR)+(I)*S +(B)+A
Relative 相对寻址	LA=(PC)+A

重点知识点

- ARM寻址模式
  - 只有加载/保存指令能访问存储器
  - 间接寻址，通过基址加偏移
  - 提供两种变址寻址模式：前变址、后变址

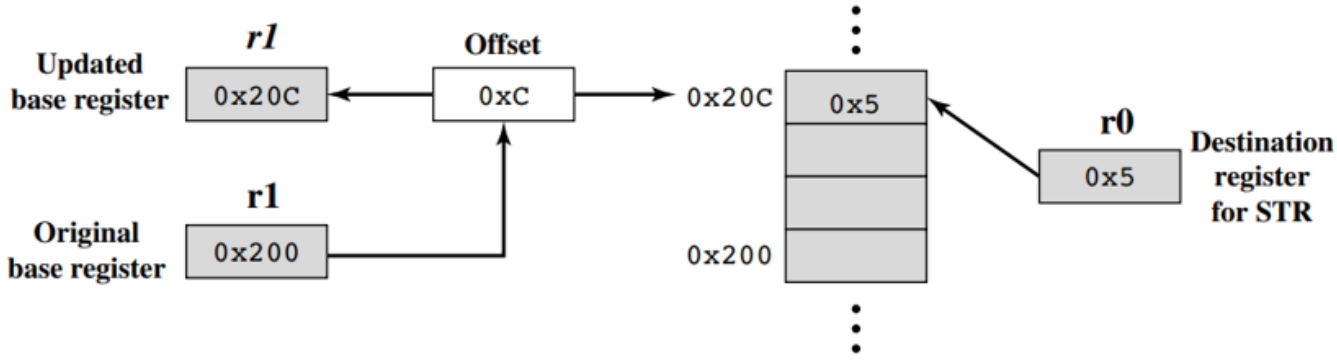


偏移寻址

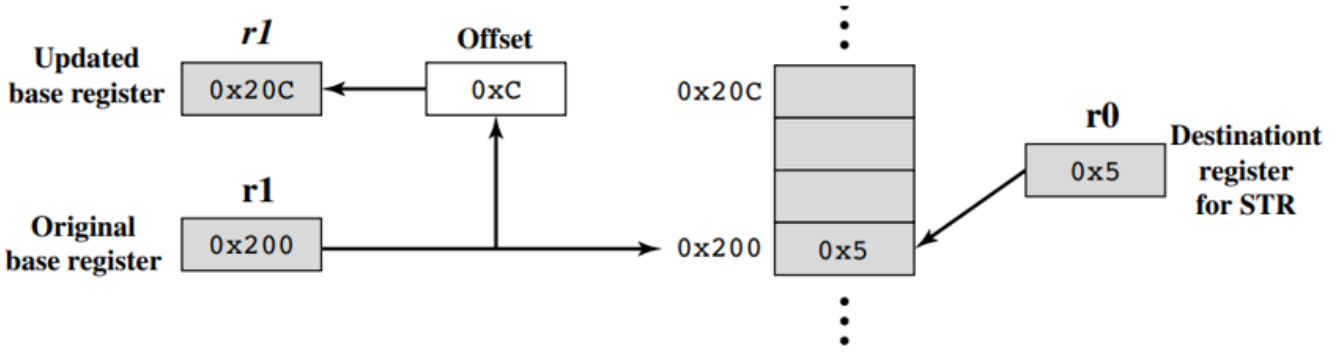
重点知识点

- ARM寻址模式

前变址



后变址

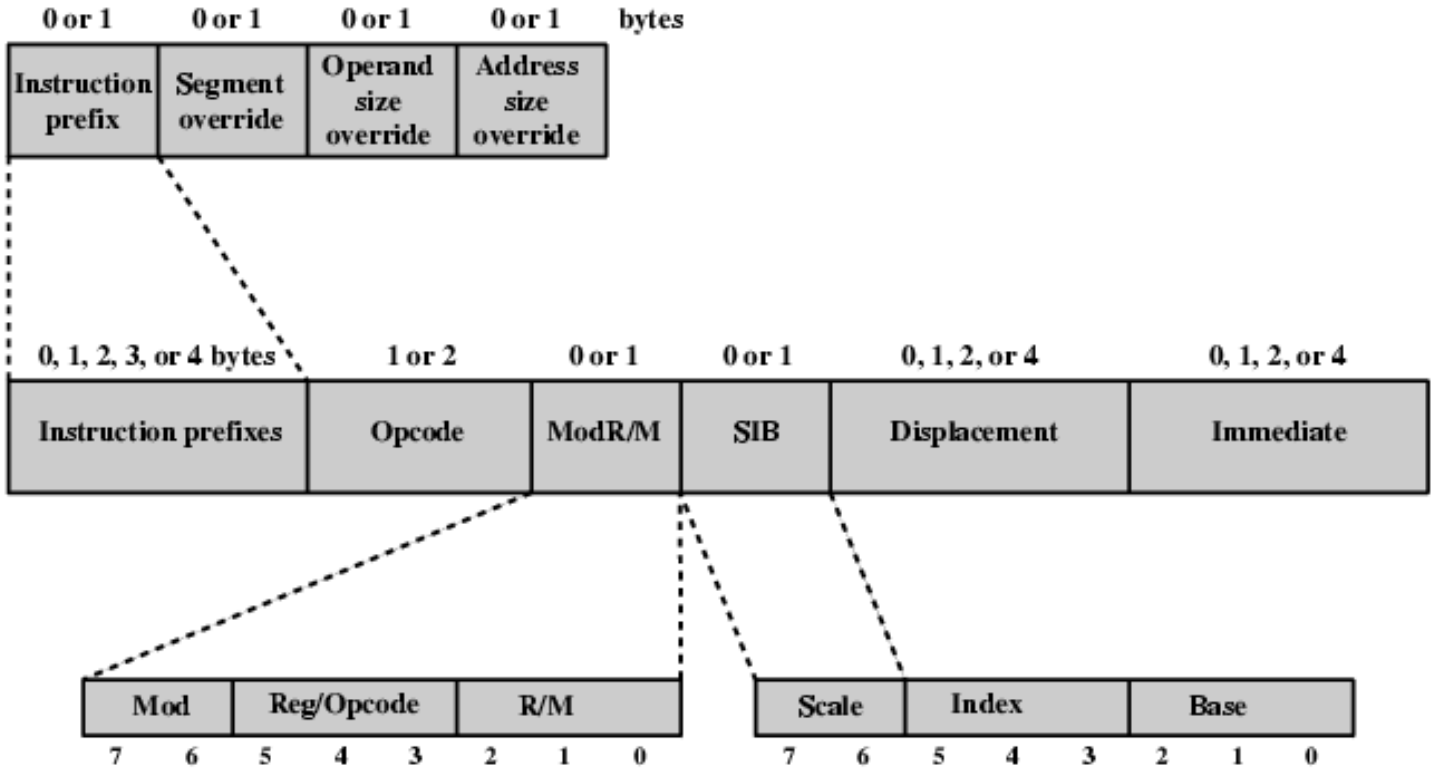


重点知识点

- 指令格式中的关键要素
  - 操作码宽度：决定了多少种操作
  - 操作数宽度：影响指令长度
  - 寻址模式：决定了复杂性和指令长度
  - 指令长度：
    - 操作码种类，操作数数量，寻址模式，地址域宽度，总线带宽，CPU速度
  - 位的分配
    - 寻址模式，操作数数量，寄存器还是存储器，地址空间，地址粒度

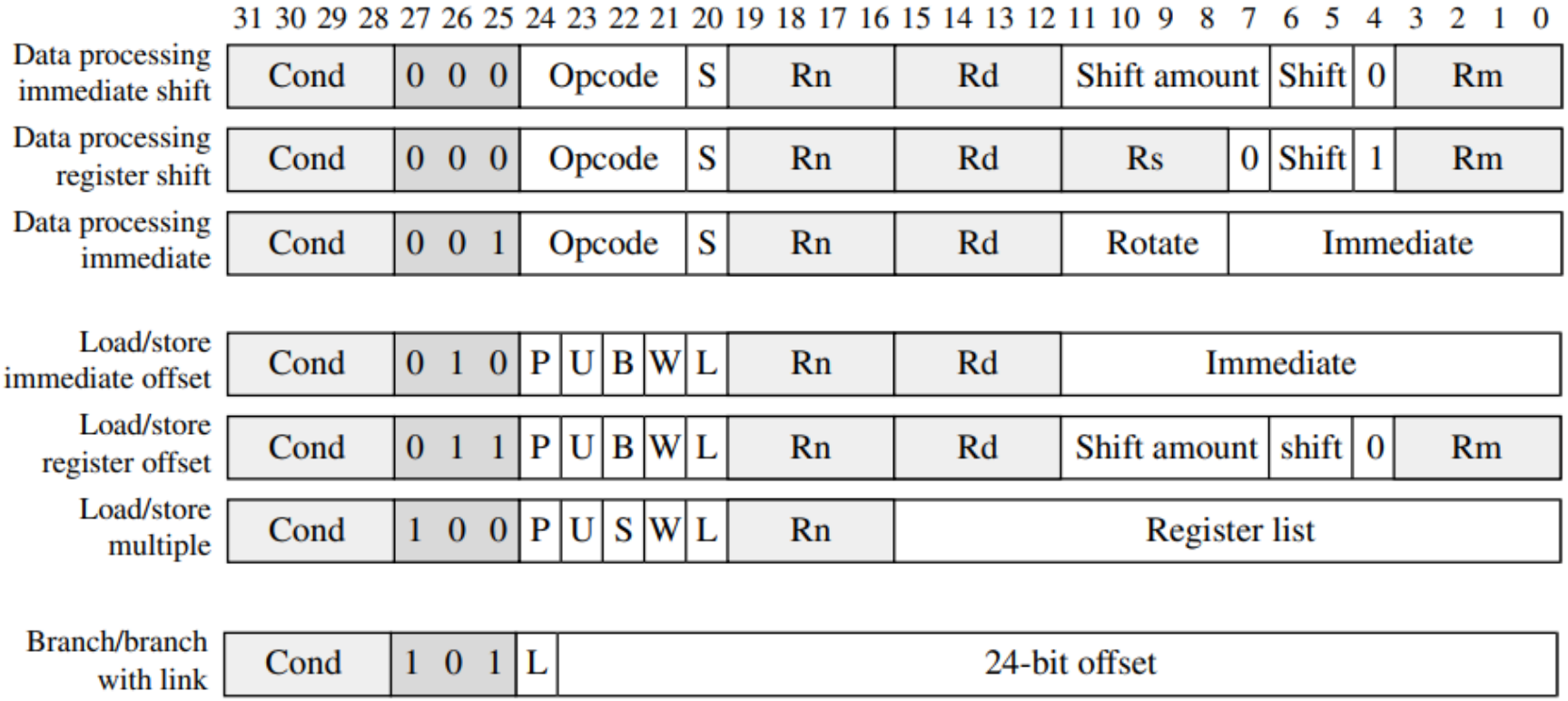
重点知识点

- X86指令格式



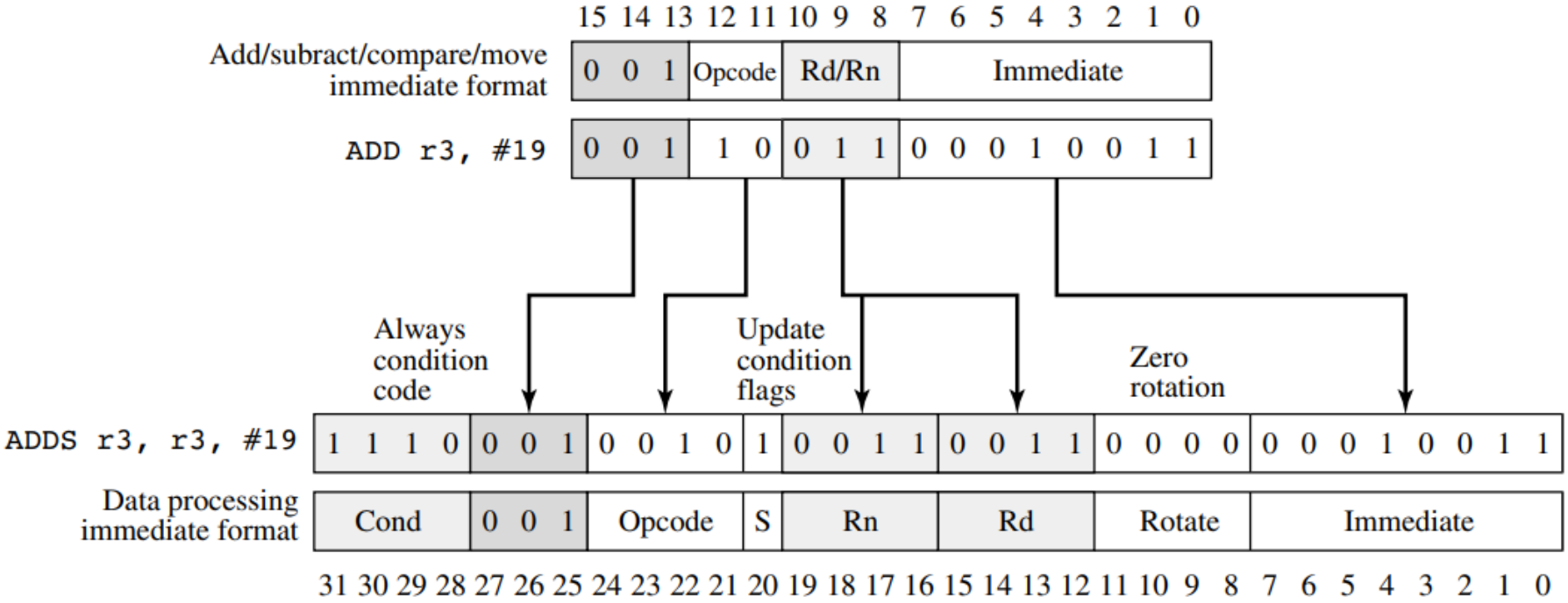
重点知识点

- ARM指令格式



重点知识点

- ARM压缩指令集



第十二章　CPU

- 基本概念
  - 寄存器
  - 指令周期
  - 指令流水线
  - 流水线冒险

第十二章 CPU

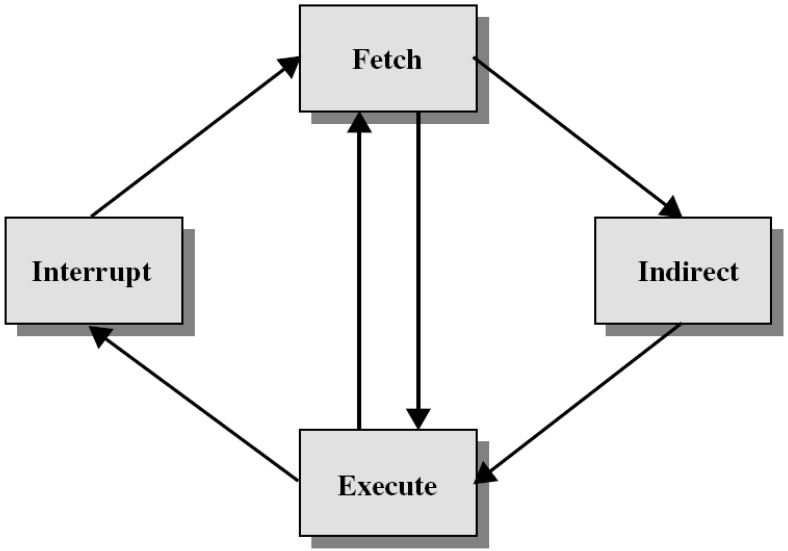
- 重点知识点
  - 处理器结构和寄存器
  - 指令执行过程
  - 指令流水线
  - 流水线中断原因及处理方法

基本概念

- 寄存器
  - CPU内临时存放数据的部件
  - 处理器的设计不同，数量和功能也不同
  - 处理器设计的重要因素，对指令集设计具有重要影响
  - 是计算机分级存储体系中的顶层，速度最快，数量最少
  - 分为用户可见寄存器，控制和状态寄存器
    - 可见寄存器：提高处理速度
    - 控制和状态寄存器：用于控制，以及处理的状态信息

基本概念

- 指令周期
  - 完成一个指令的过程，成为指令周期
  - 至少包括两个子周期：取指周期、执行周期
  - 如果有中断处理，还有中断周期
  - 如果有间接取指，还有间接周期



基本概念

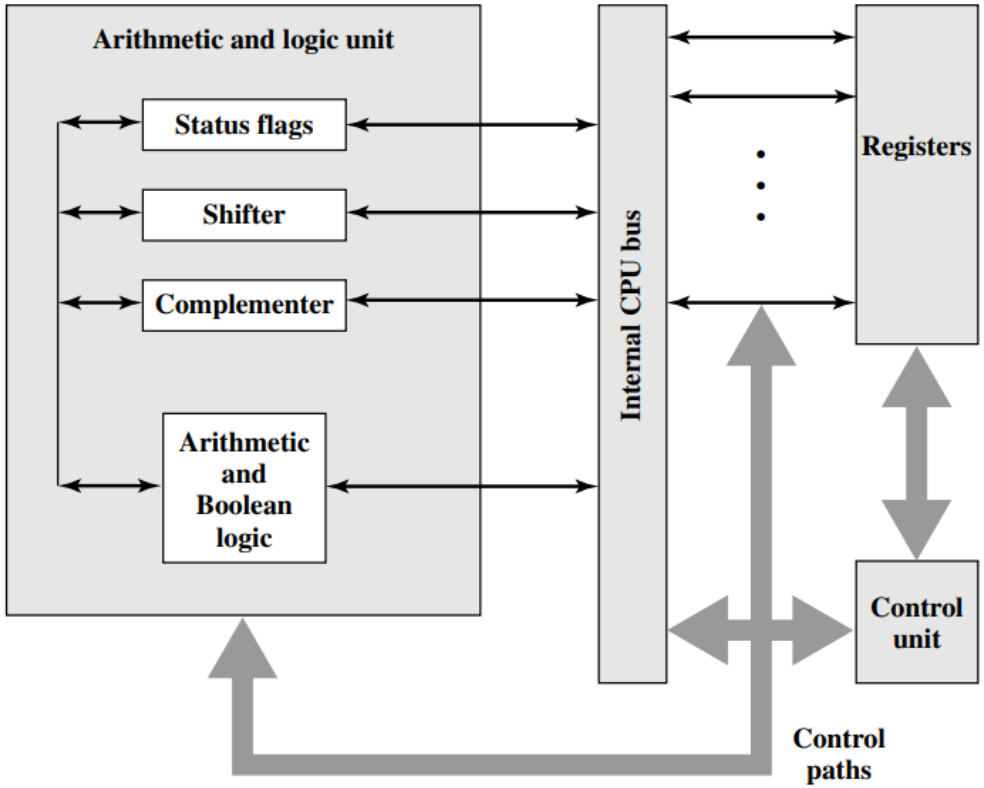
- 指令流水线
  - 不同指令的不同阶段进行并行处理的方式
  - 不能降低单个指令的执行时间
  - 提升整体系统的吞吐率
  - 最长阶段的执行时间决定了流水线的效率
  - 容易受到资源冲突、数据冲突以及指令流改变等因素的影响，导致效率降低
  - 指令分的阶段越多，理想的加速比越大，但是容易受到影响，而且到一定程度之后，收益递减

基本概念

- 流水线冒险 pipeline hazard，pipeline bubble
  - 流水线冒险：数据依赖、控制依赖和资源依赖发生的时候，流水线会因此而暂停运行，称为流水线冒险
  - 资源冒险：流水线中的两个指令都需要使用同一个资源而引起
  - 数据冒险：对同一个操作数的访问出现冲突
  - 控制冒险：经常称为分支冒险。程序中有分支转移指令，而流水线在取指的时候，没有正确判断转移的结果，导致获取了错误的指令

重要知识点

- 处理器的功能和内部结构
  - 取指、解码、取数据、计算、写结果



重要知识点

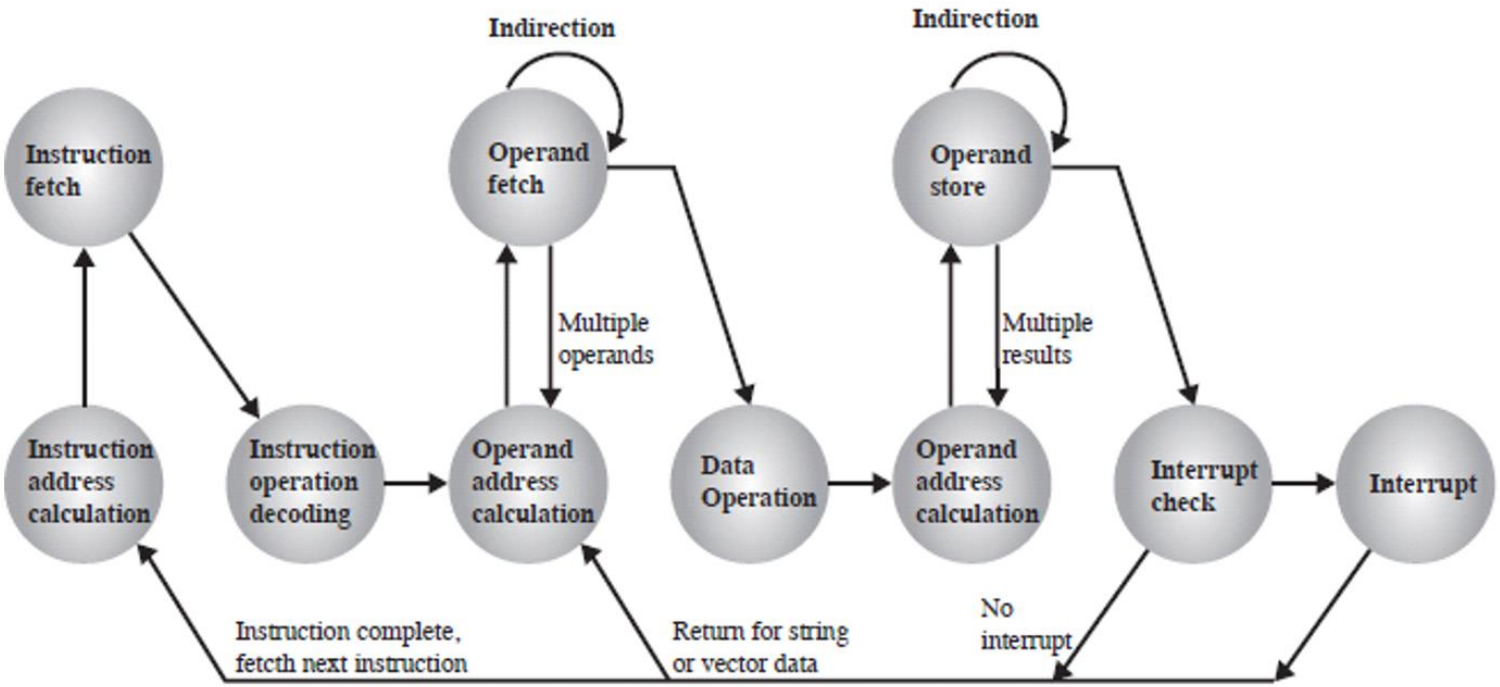
- 用户可见寄存器
  - 包括通用寄存器、数据寄存器、地址寄存器、条件码寄存器
  - 一般在8-32个左右
  - 寄存器较多，能减少内存引用，但是会增加成本。达到一定数量之后，增加寄存器对内存引用减少的效果不明显
  - RISC架构对寄存器的使用比较多

重要知识点

- 控制和状态寄存器
  - 程序计数器PC
  - 指令寄存器IR
  - 内存地址寄存器MAR
  - 内存缓冲寄存器MBR
  - 程序状态字寄存器PSW：包括一些标志

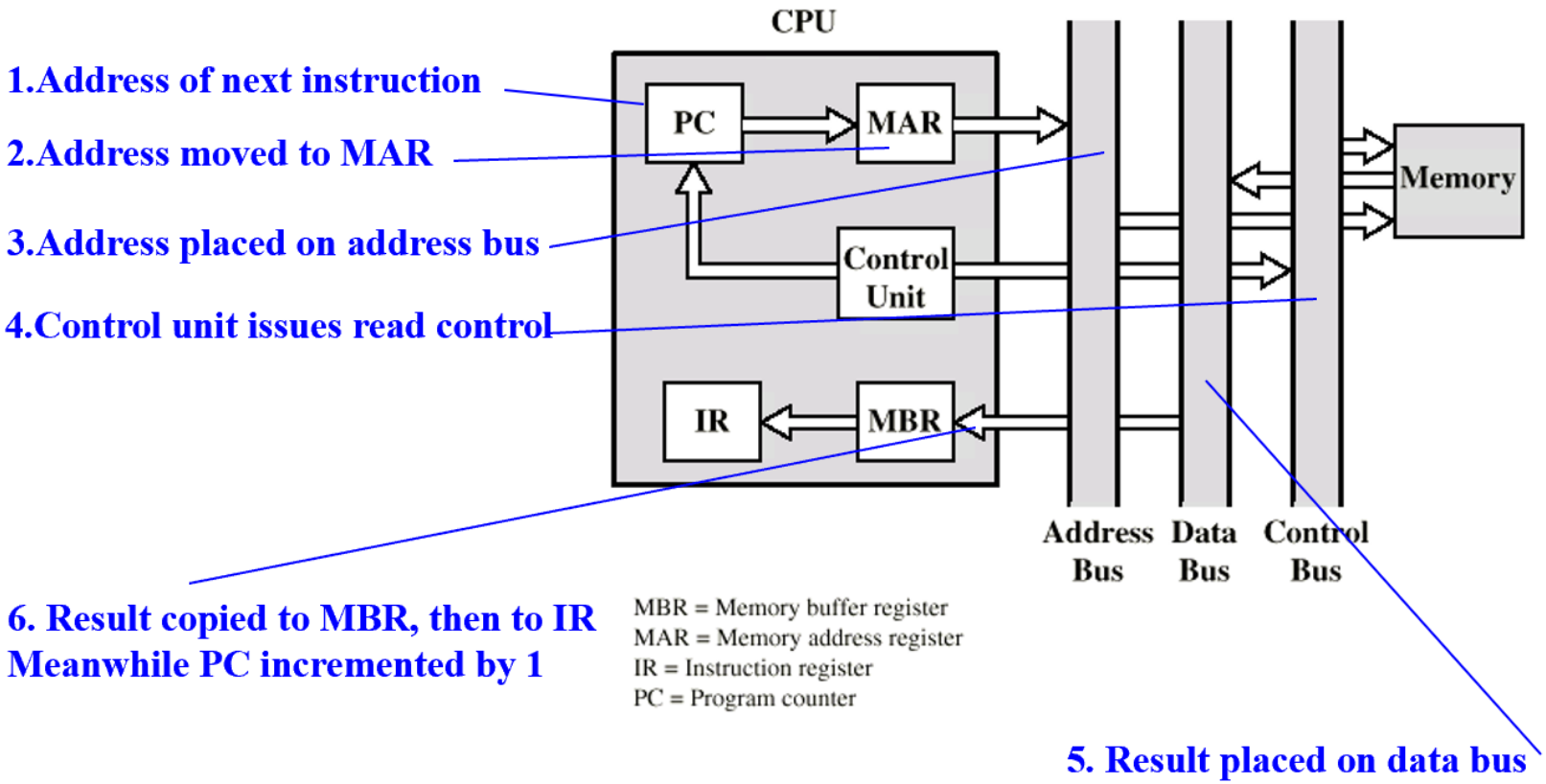
重要知识点

- 指令执行过程
  - 取指周期、执行周期、间接周期、中断周期



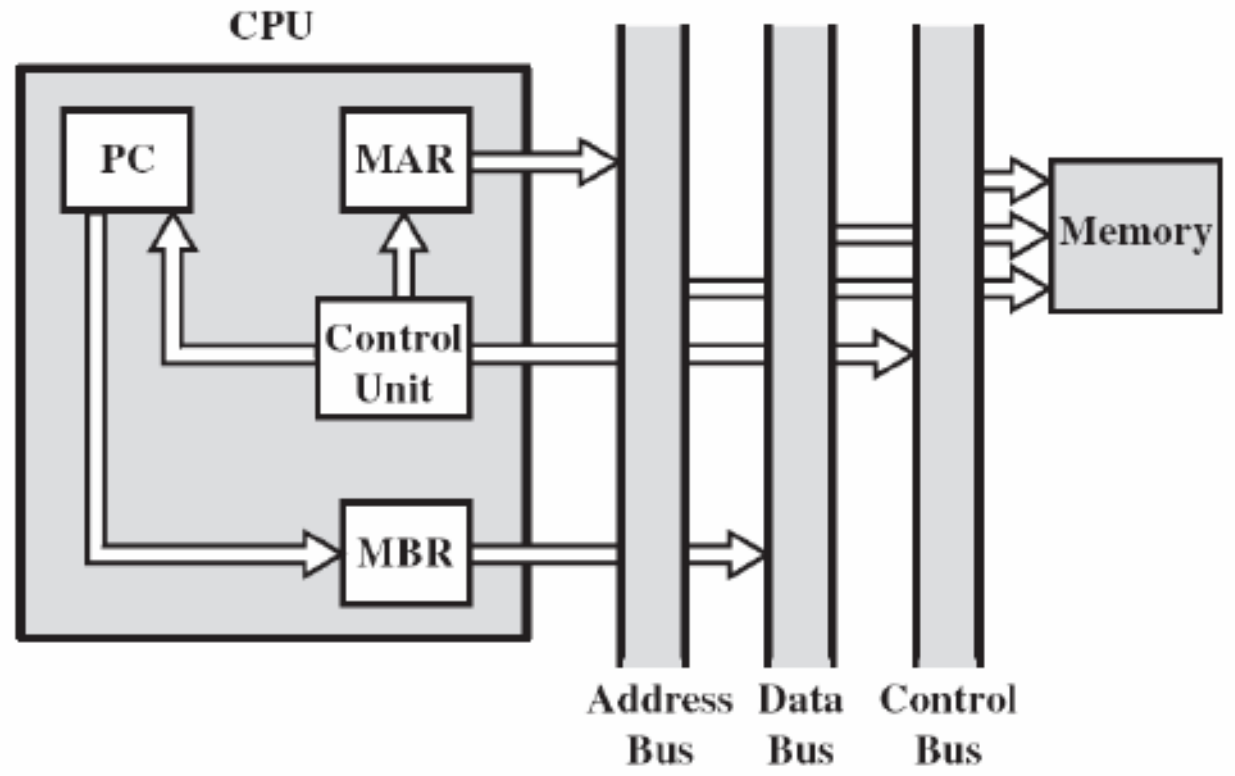
重要知识点

- 取指的数据流



重要知识点

- 中断周期的数据流



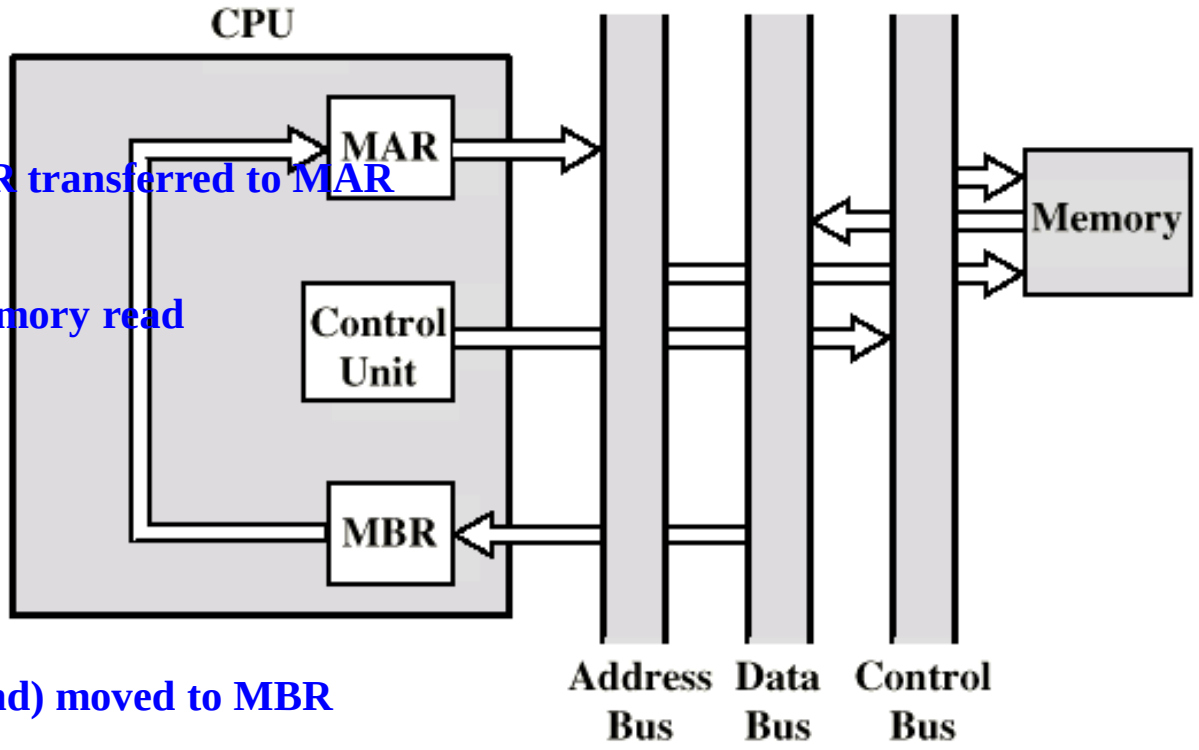
重要知识点

- 取操作数的数据流

1. Right most N bits of MBR transferred to MAR

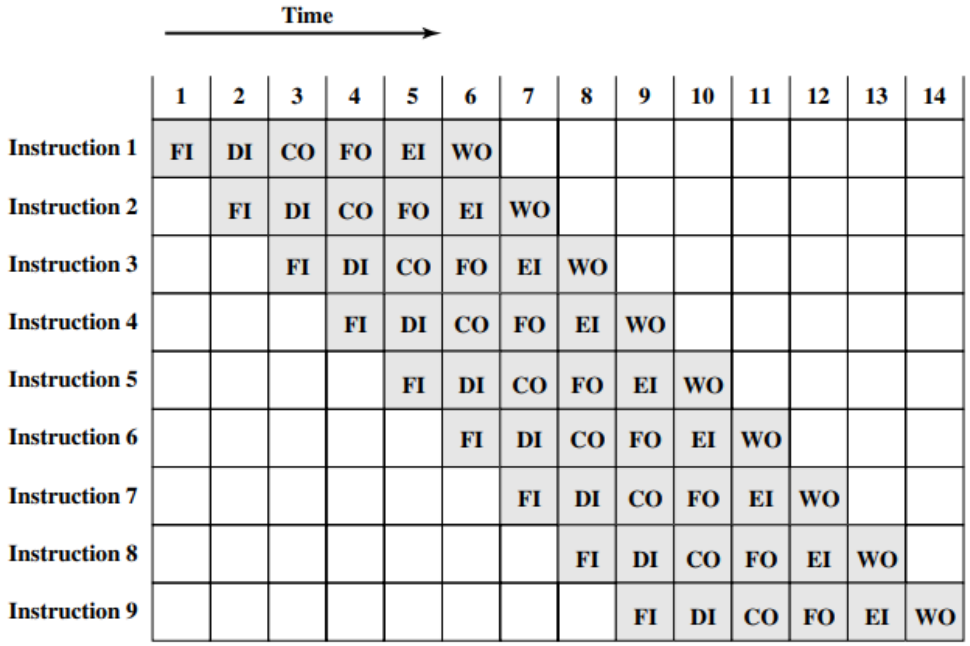
2. Control unit requests memory read

3. Result (address of operand) moved to MBR



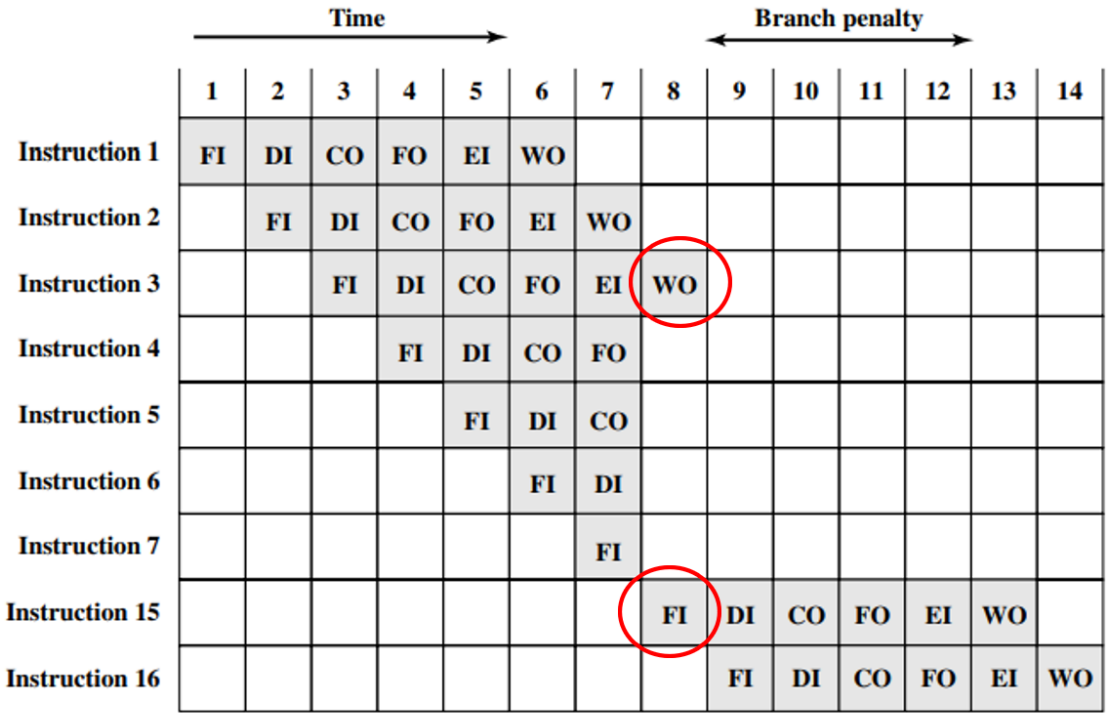
重要知识点

- 理想的流水线
  - 重复相同的操作
  - 操作独立性
  - 操作可分解为子操作，并且尽可能耗费相同的时间



重要知识点

- 带分支的流水线
  - 分支指令后的若干指令需要清空
  - 影响流水线的效率。

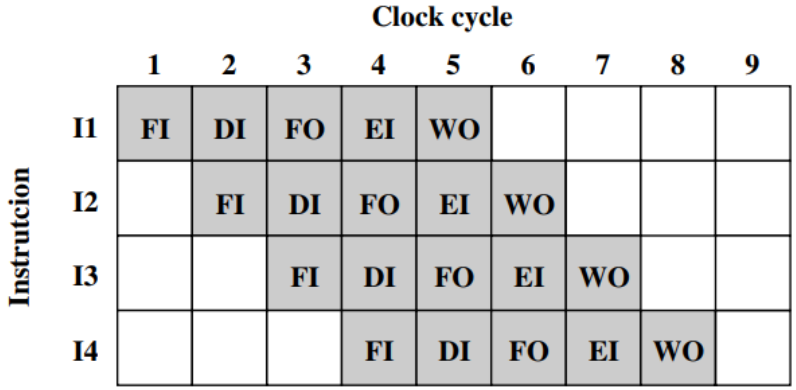


重要知识点

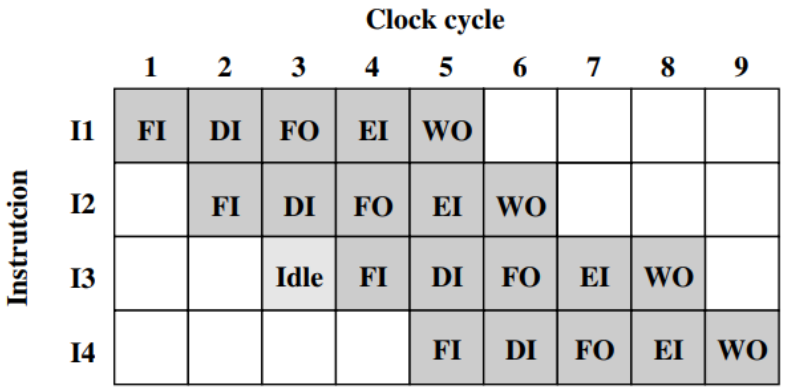
- 流水线中断的原因
  - 资源竞争
  - 相关性
    - 数据相关性
    - 控制流
  - 长延迟操作

重要知识点

- 资源冒险
  - 两个或多个指令需要同一个资源，也称为结构冒险
  - 处理方法：1. 增加资源；2. 串行执行



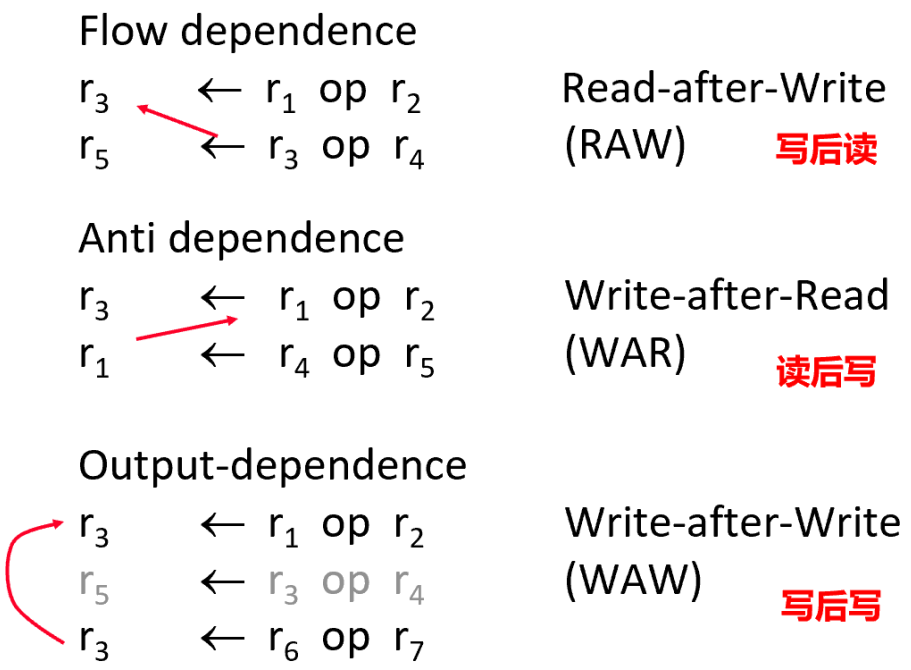
(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

重要知识点

- 数据冒险
  - 写后读，真相关 Read after write (RAW), or true dependency
  - 读后写，也称为反相关，Write after read (RAW), or anti-dependency
  - 写后写，也称为输出相关，Write after write (WAW), or output dependency



重要知识点

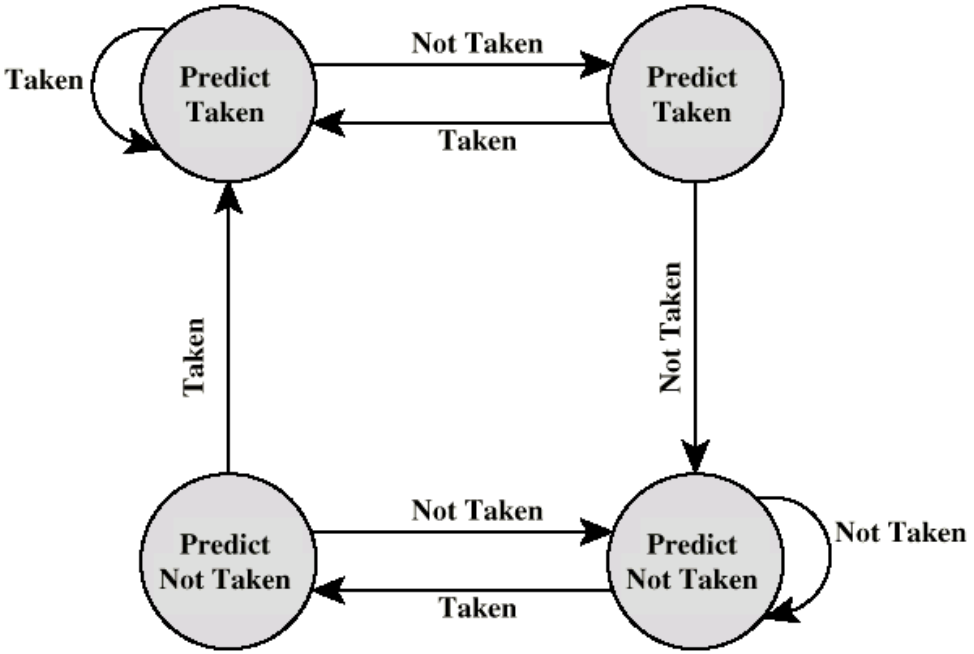
- 数据冒险的处理方法
  - 反相关、数据相关
    - 增加寄存器
  - 真相关
    - 延迟

重要知识点

- 控制冒险
  - 有些在解码后才能确定下一个指令的地址
  - 有些需要在指令执行完成之后才能确定下一个指令的地址
  - 丢弃不需要的指令
  - 处理方法：
    - 多指令流
    - 预取分支目标
    - 环形缓冲
    - 分支预测
    - 延迟分支

重要知识点

- 分支预测
  - 静态预测
    - 预测从不发生
    - 预测总是发生
    - 根据操作码预测
  - 动态预测
    - 根据条件转移指令的历史转移情况进行预测
    - 基于关联的预测



第十三章　RISC

- 基本概念
  - CISC
  - RISC

第十三章 RISC

- 重点知识点
  - CISC和RISC
  - 寄存器优化
  - CISC流水线

基本概念

- CISC： Complex Instruction Set Architecture
  - 指令操作码种类多
    - X86的操作码为1~2个字节
  - 指令长度可变
    - X86的指令长度为1~16字节
  - 寻址模式多种多样
    - X86有9种寻址模式，典型的如基址比例变址带偏移量

基本概念

- RISC：Reduced Instruction Set Architecture
  - 简化的指令集
    - 采用规范化的、定长的指令格式。ARM的指令均为32位
    - 有限的操作类型。操作码只有8位
    - 取指、解码变得更容易
  - 尽量使用寄存器
    - 除了加载/保存指令，其他指令都是针对寄存器操作
    - 内存访问只有三种寻址模式
  - 流水线设计
    - 精心设计指令流水线，更好地满足条件分支和过程调用对流水线的影响
    - 每个指令都是条件执行，这样可以减少分支

重点知识点

- CISC的原因和目的
  - 软件越来越复杂
  - 高级语言和指令集的差距（语义鸿沟，Semantic gap）越来越大，需要更复杂的指令集来翻译，执行效率低
  - 提供更复杂度指令集，提高效率
- 目标
  - 编译器容易编写
  - 提高程序效率
  - 支持更复杂的高级语言

重点知识点

- CISC的实际效果
  - 编译器也不简单：复杂的机器指令要求高，用的少，并且优化很困难
  - 程序也不省空间：看起来语句少，但不一定能省空间
  - 程序也不见得更快：控制复杂，降低了简单指令的指令的速度
- RISC的出现
  - 寻找一种截然不同的方法，同样是为了更好地支持HLL，解决HLL和指令集之间的语义鸿沟，但是采用更简单而不是更复杂的硬件结构，也就是RISC结构

重点知识点

- RISC的特点
  - 每个机器周期完成一个指令的执行
  - 大多数的操作都是寄存器到寄存器的，访问存储器的操作由Load/Store来完成
  - 少量且简单的寻址模式
  - 少量简单且固定的指令格式
  - 采用硬布线的方式来实现
  - 更有效的指令流水线技术

重点知识点

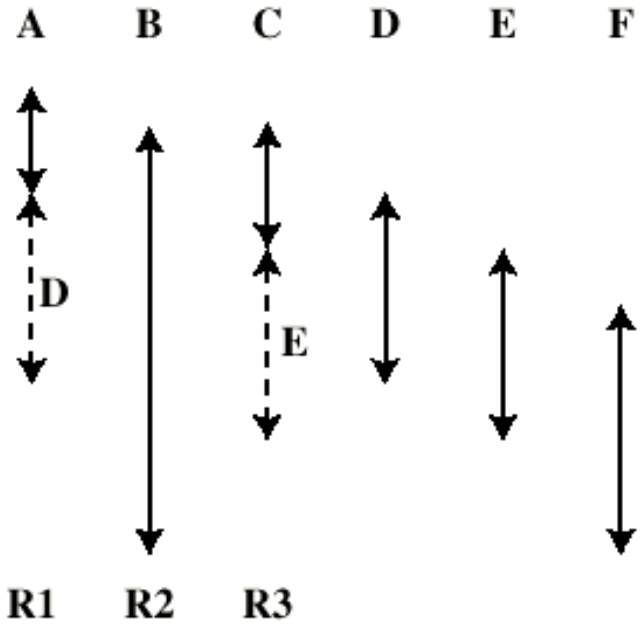
- 寄存器使用要点
  - 目标：最频繁访问的操作数保持在寄存器中，减少“寄存器-存储器”操作
  - 方法：
    - 软件方法：通过编译器对寄存器的分配，来最大化利用寄存器。在一个给定的时间范围内，将寄存器分配给最常使用的变量。这种方法要求使用复杂的程序分析方法，分析哪个变量使用最多。
    - 硬件方法：增加寄存器的数量，使得更多的变量能够更长时间地保留在寄存器中

重点知识点

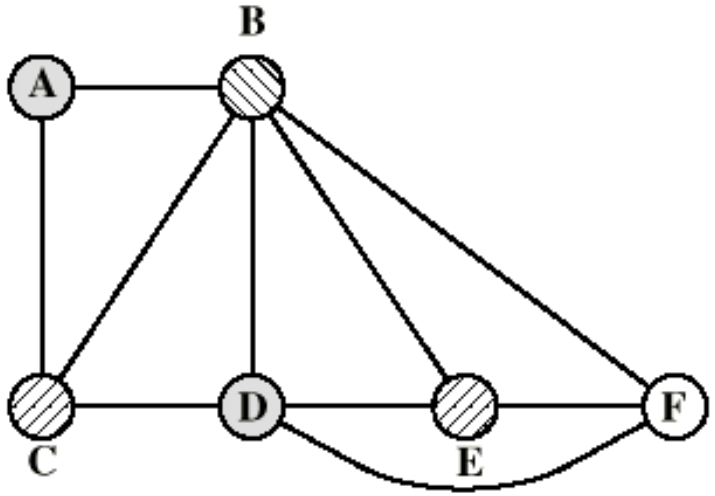
- 寄存器的分配优化-图着色问题
  - 构造一个无向图。
  - 节点是符号寄存器
  - 如果两个寄存器在同一个程序段，用一个边来连接这两个节点
  - 用N种颜色去给节点着色，要求两个相邻的（有边连接的）点不能用同一种颜色
  - N就是需要的实体寄存器的数量
  - 如果N>实际的寄存器数量，有些节点就要放到存储器中

重点知识点

- 寄存器的分配优化-图着色问题



(a) Time sequence of active use of registers



(b) Register interference graph

重点知识点

- RISC流水线

Load	rA ← M	I	E	D									
Load	rB ← M				I	E	D						
Add	rC ← rA + rB							I	E				
Store	M ← rC									I	E	D	
Branch	X											I	E

(a) Sequential execution

Load	rA ← M	I	E	D						
Load	rB ← M		I	E	D					
NOOP				I	E					
Add	rC ← rA + rB				I	E				
Store	M ← rC					I	E	D		
Branch	X						I	E		
NOOP								I	E	

(c) Three-stage pipelined timing

Load	rA ← M	I	E	D						
Load	rB ← M		I		E	D				
Add	rC ← rA + rB				I		E			
Store	M ← rC						I	E	D	
Branch	X							I		E
NOOP										I	E

(b) Two-stage pipelined timing

Load	rA ← M	I	E_1	E_2	D						
Load	rB ← M		I	E_1	E_2	D					
NOOP				I	E_1	E_2					
NOOP					I	E_1	E_2				
Add	rC ← rA + rB					I	E_1	E_2			
Store	M ← rC						I	E_1	E_2	D	
Branch	X							I	E_1	E_2	
NOOP									I	E_1	E_2
NOOP										I	E_1	E_2

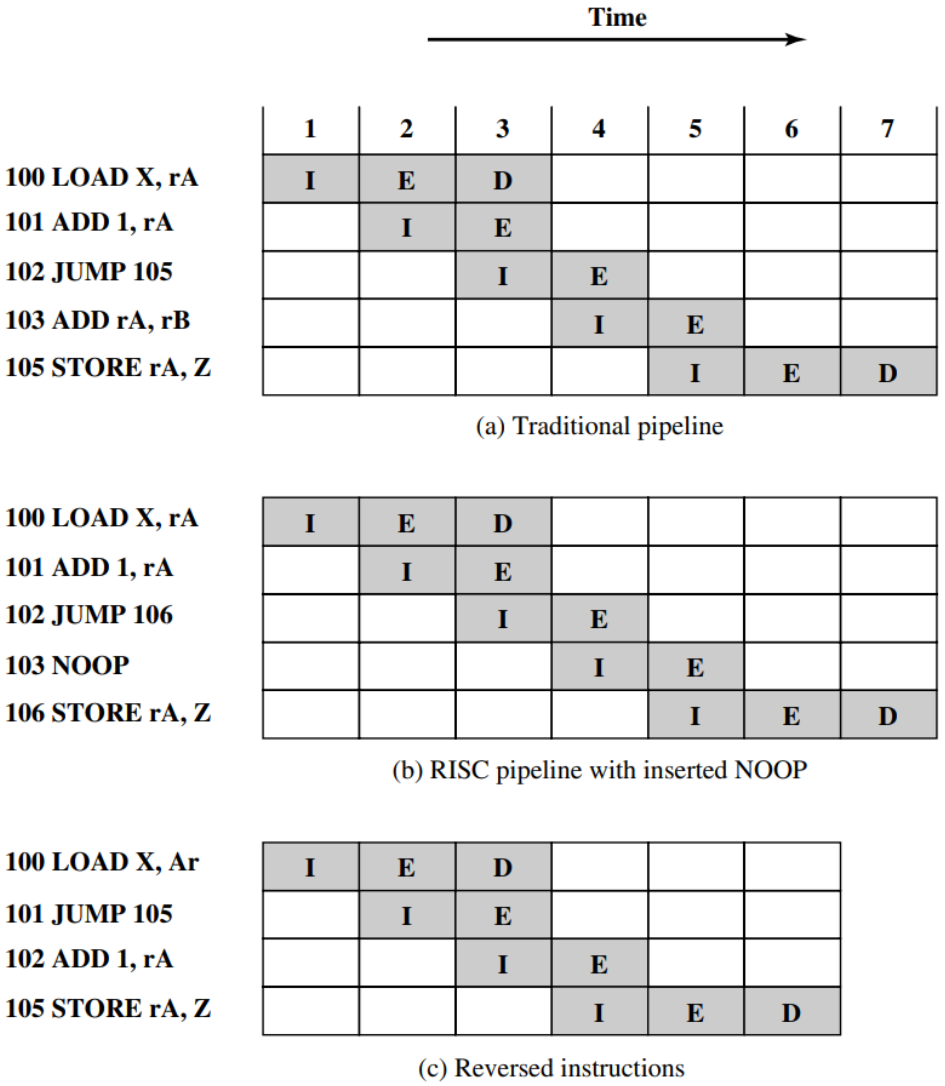
(d) Four-stage pipelined timing

重点知识点

- 延迟分支
  - 分支会导致流水线清空，效率降低
  - 是提高管道效率的一种方法，目的是保持流水线充满
  - 调整指令顺序，将不影响分支的指令和分支指令进行顺序交换，避免执行无效指令
  - 需要找到指令和分支指令进行顺序交换
    - 对于无条件分支：容易找到
    - 对于条件分支：不太容易

重点知识点

- 延迟分支举例



重点知识点

- 循环展开
  - 复制循环体
  - 降低迭代次数，减少循环开销
  - 提高指令并行性
  - 降低存储器访问次数

重点知识点

- RISC和CISC
  - 比较困难
    - 没有能直接进行对比的两个处理器
    - 没有合适的测试集
    - 处理器和编译器的效果不好区分
    - 没有量产的处理器进行对比
  - 相互融合，相互借鉴

第十四章　超标量

- 基本概念
  - 超级流水线
  - 超标量
  - 指令级并行与机器并行
  - 指令发射

第十四章　超标量

- 重点知识点
  - 超级流水线和超标量
  - 相关性对超标量的影响
  - 指令发射策略
  - 相关性的因素分析
  - 超标量执行步骤

基本概念

- 超级流水线 Super-pipelined
  - 很多流水线的阶段需要的时间不到一个时钟周期
  - 采用了比外部时钟快一倍的内部时钟来进行指令的调度
  - 在一个内部时钟周期内完成一个指令阶段
  - 从外部时钟周期来看，一个时钟周期可以完成2个指令阶段的处理。吞吐量提高了1倍
  - 本质是通过缩短指令流水线阶段的执行时间来提高效率

基本概念

- 超标量 superscalar
  - 真正的指令集并行
  - 多个指令同时进入处理器进行执行
  - 可以采用多个独立的流水线方式
  - 由于指令的并行执行，指令间的相关性更复杂，大大提高了指令调度的复杂性

基本概念

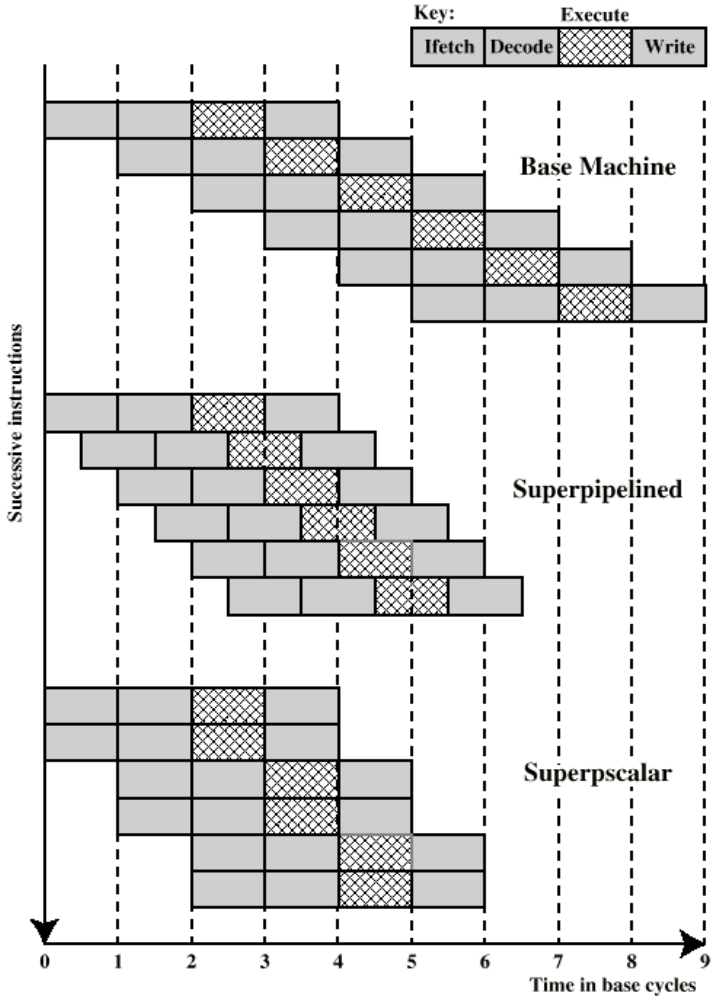
- 指令级并行性 Instruction-level Parallelism
  - 程序指令能够并行执行的程度
  - 程序具有指令级并行性，那么指令序列中的指令是独立的，并且能够重叠地执行指令
- 机器并行性 Machine Parallelism
  - 处理器获取指令级并行性好处的能力
  - 机器并行性一方面受限于流水线的数量，另一方面和处理器的结构有关系

基本概念

- 指令发射 Instruction issue
  - 启动指令去处理器的功能单元进行执行的过程
  - 指令发射的发生时间是在指令从流水线的译码阶段向指令执行的第一个阶段移动的时候
  - 为了提高并行性，必须要使用合理的发射顺序来进行指令的发射，而不能按照原始的指令顺序去发射指令。这种顺序，或排序方法，就称为指令发射策略
  - 约束是结果必须是正确的

重点知识点

- 超标量和超级流水线对比

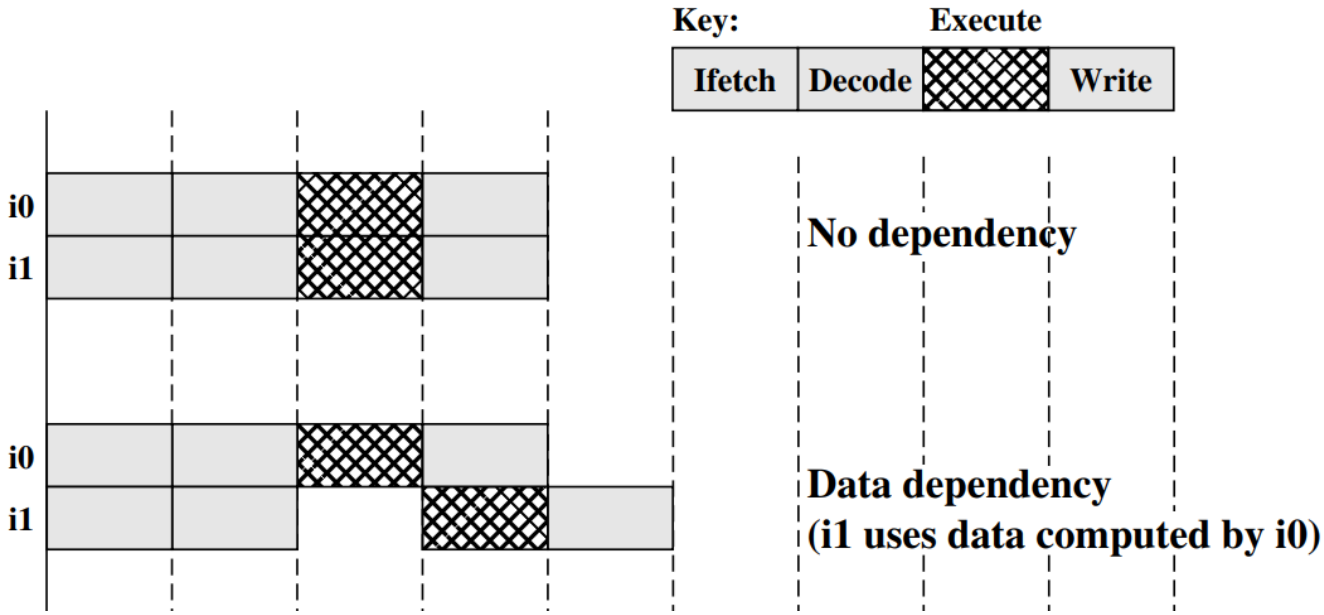


重点知识点

- 真数据相关性

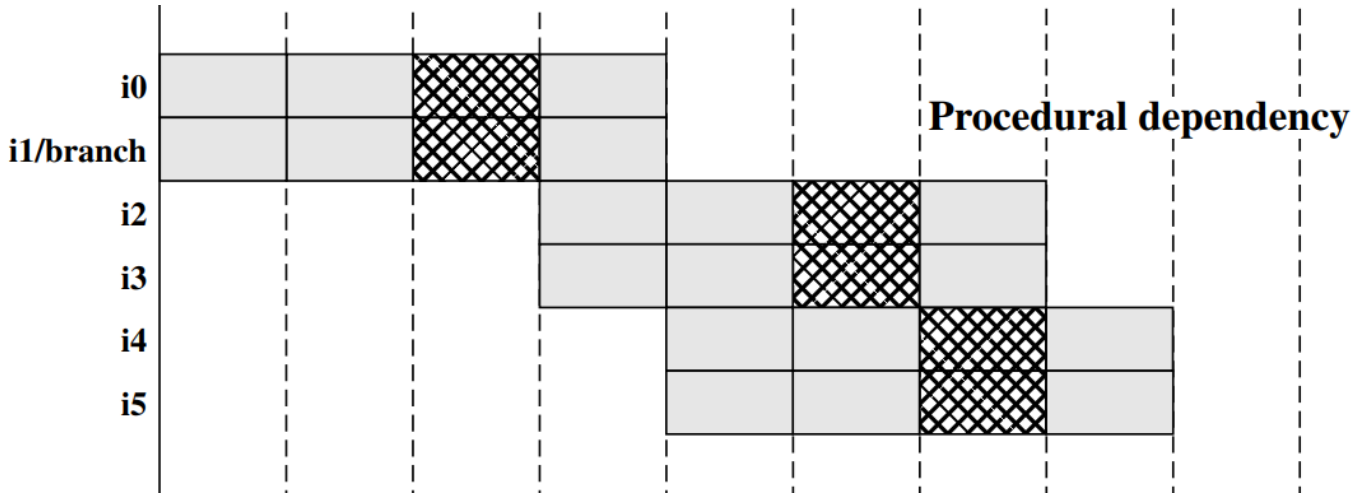
ADD r1, r2 (r1 := r1+r2;)

MOVE r3,r1 (r3 := r1;)



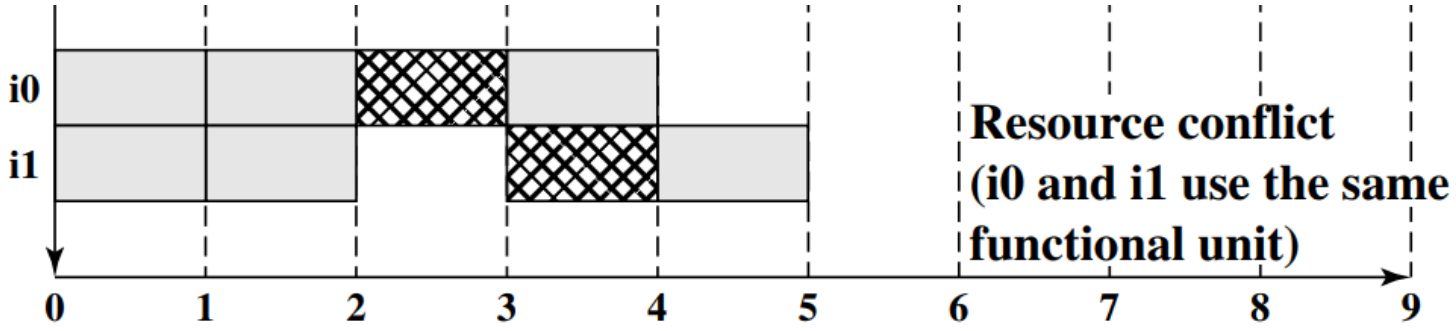
重点知识点

- 过程相关性
  - 分支前和分支后的指令不能并行执行
  - 非定长指令的取指问题



重点知识点

- 资源相关性
  - 资源冲突
  - 通过资源复制可以解决



重点知识点

- 指令发射策略
  - 按序发射，按序完成
  - 按序发射，乱序完成
  - 乱序发射，乱序完成

重点知识点

- 按序发射，按序完成
  - 按照指令的发生顺序完成指令发射和执行
  - 可能需要指令延迟

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

重点知识点

- 按序发射，乱序完成
  - 允许指令不按照发射的顺序进行指令执行
  - 会带来输出相关性问题

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

重点知识点

- 输出相关性
  - 写后写，write-write dependency，后一个写的指令先于前一个指令完成
    - R3:= R3 + R5; (I1)
    - R4:= R3 + 1; (I2)
    - R3:= R5 + 1; (I3)
  - 乱序执行的动态调度方案
    - 将有依赖关系的指令和独立指令分离，独立指令可以执行
    - 对有依赖关系的指令放在一个休息区，称为保留站
    - 对每个在休息区的指令监控它的关联值
    - 如果指令的所有关联值都可用，那么就发射这个指令

重点知识点

- 乱序发射，乱序完成
  - 译码段和执行段的流水线进行解耦
  - 指令窗口缓冲区存放解码完成的指令
  - 功能单元可用时，发射指令
  - 会带来反相关问题

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1, I2*	I1	I2				2
I5	I6	*I3, I4*	I1		I3	I2		3
		I4, I5, I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

重点知识点

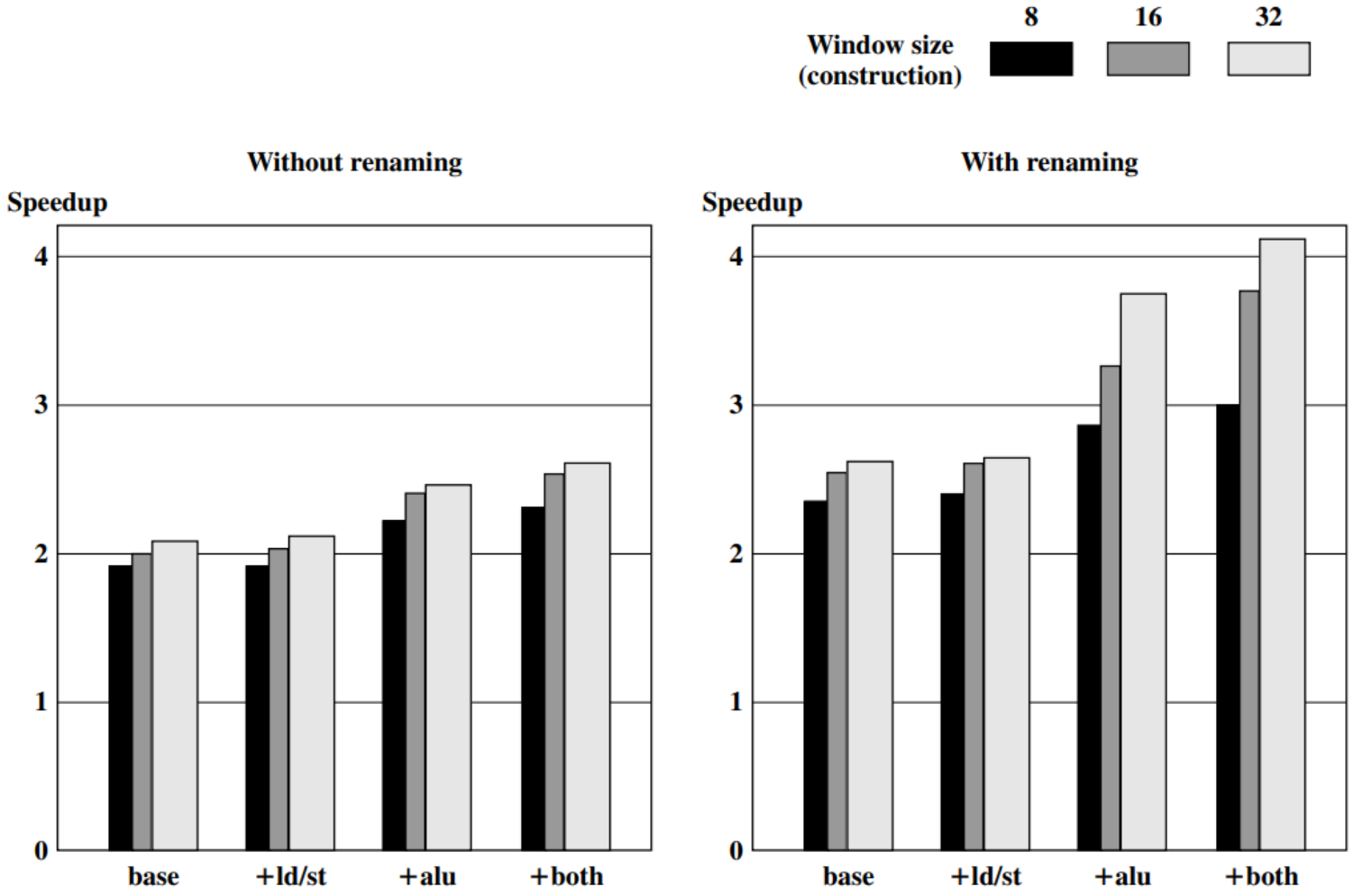
- 反相关性
  - 读后写， Write-after-read dependency ，后一个写的指令先于前一个写的指令完成
    - R3:=R3 + R5; (I1)
    - R4:=R3 + 1; (I2)
    - R3:=R5 + 1; (I3)
    - R7:=R3 + R4; (I4)
  - 解决方法：寄存器重命名
    - 输出和反相关性发生的原因是寄存器内容可能无法反映程序的正确顺序
    - 根据需要重新分配寄存器

重点知识点

- 提高并行性因素的相关性
  - 三个因素：执行指令的资源，重命名机制，指令缓冲区
  - 没有寄存器重命名的机制，单纯依靠资源复制，性能改善不会很明显。使用了重命名，取消了反相关性和输出相关性，增加功能单元能显著提高执行速度
  - 乱序发射中，使用了寄存器窗口来缓存译码之后的指令，处理器可以先行查看的机制，识别能独立执行的指令。如果指令窗口很小，识别成功的概率会非常低。所以，指令窗口需要足够大

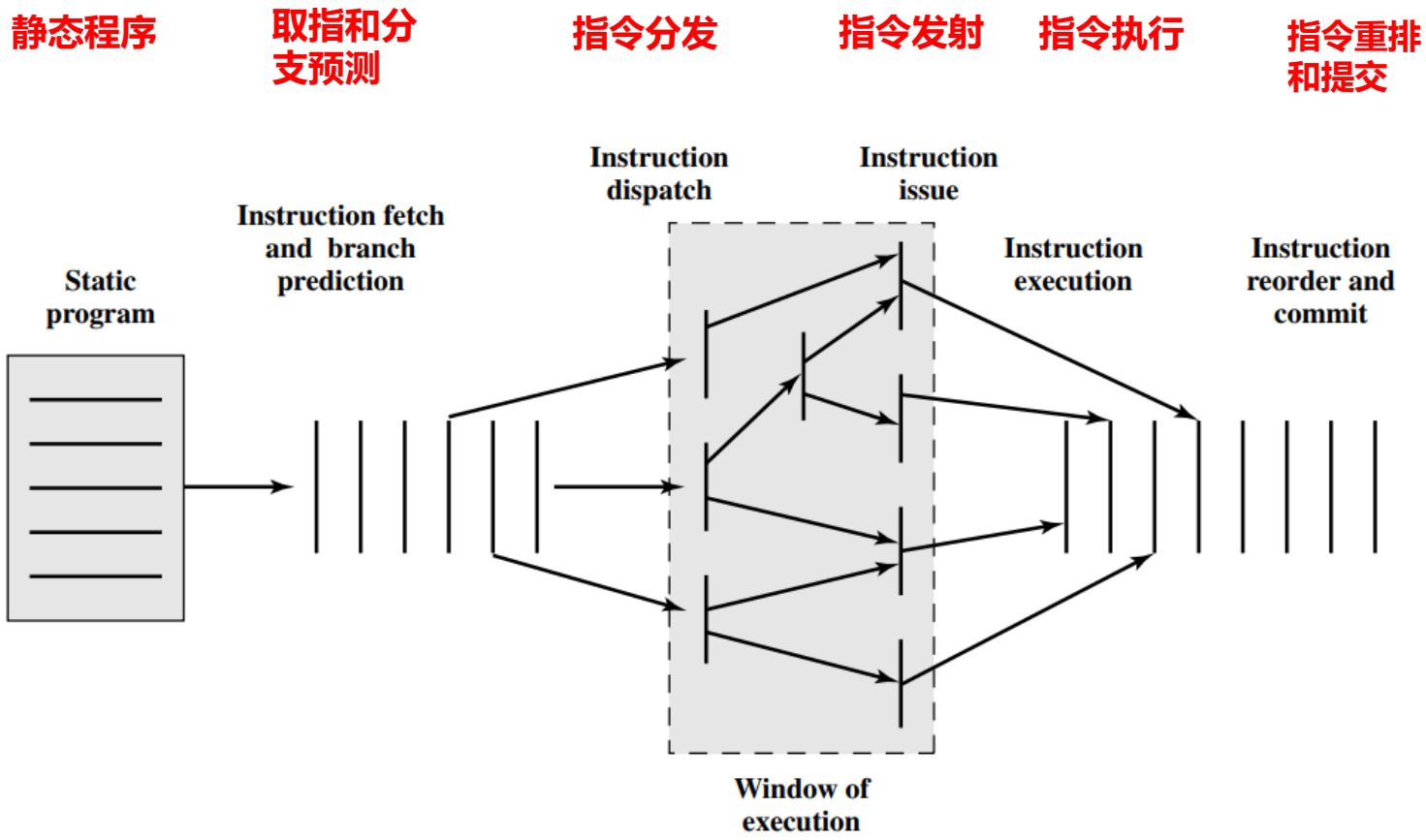
重点知识点

- 提高并行性因素的相关性



重点知识点

- 超标量执行步骤



第十五~十六章　控制器操作

- 基本概念
  - 微操作
  - 硬布线控制
  - 微程序控制

第十五~十六章　控制器操作

- 重点知识点
  - 微操作
  - 控制器模型
  - 控制信号
  - 硬布线方式
  - 微程序实现方式
  - 微程序定序和执行

基本概念

- 微操作
  - 程序由一系列的指令组成
  - 指令由指令周期来实现，一个指令周期执行一个机器指令
  - 每个指令周期由可以发分为若干个小的周期，比如取指周期、间接周期、执行周期、中断周期等
  - 周期又进一步分解为更小的步骤，称为微操作
  - 每个微操作完成一个很小的动作，在一个时钟周期内完成，涉及CPU寄存器的原子操作

基本概念

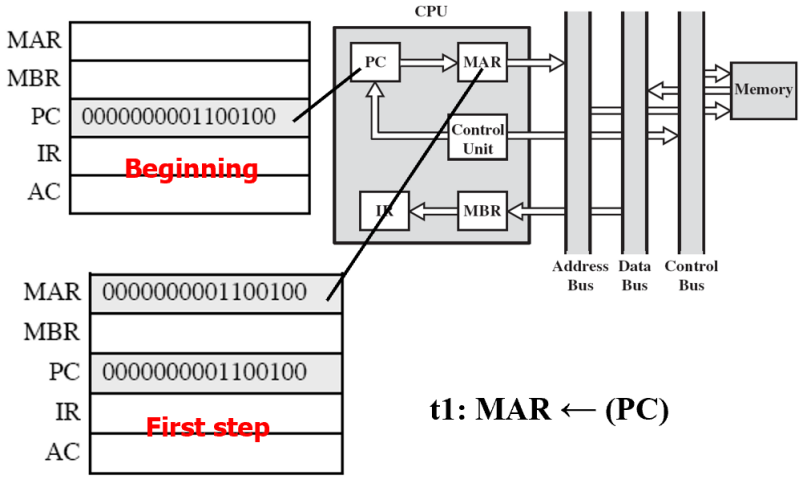
- 硬布线方式
  - 控制单元是一个组合电路
  - 根据输入信号，产生一组输出的控制信号，控制处理器内部的各个离散门
  - 可以理解为从离散门的角度，看哪些指令需要控制这个离散门的开关
  - 结构复杂
  - 新加指令，需要重新设计控制逻辑

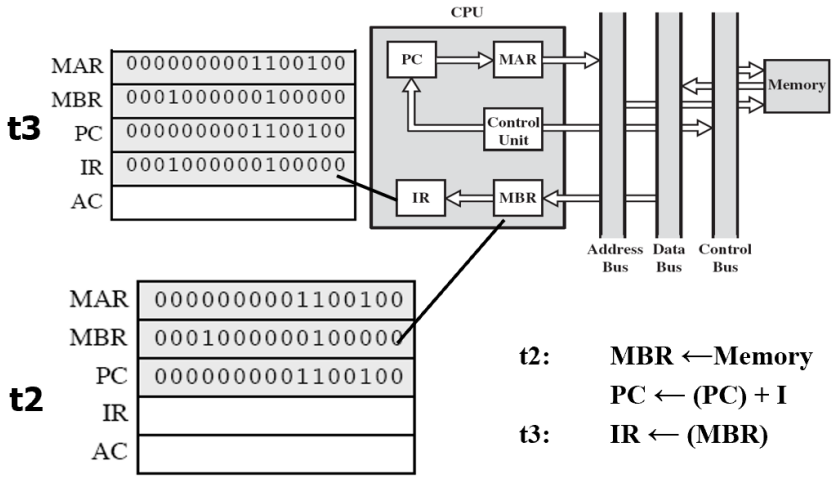
基本概念

- 微程序方式
  - 控制单元可以通过称为微程序控制单元的更灵活的技术来实现
  - 控制单元的逻辑由微程序描述
  - 微程序中的每行描述一组微操作
  - 这个微操作产生一组控制信号，控制各个离散门的开关
  - 可以理解为从指令的角度，看这个指令去控制哪些离散门的开关
  - 优点：简化控制器设计，不容易出错
  - 缺点：需要查询控制存储器，速度稍慢

重点知识点

- 取指的微操作过程





- t1: MAR <- (PC)
- t2: MBR <- (memory)

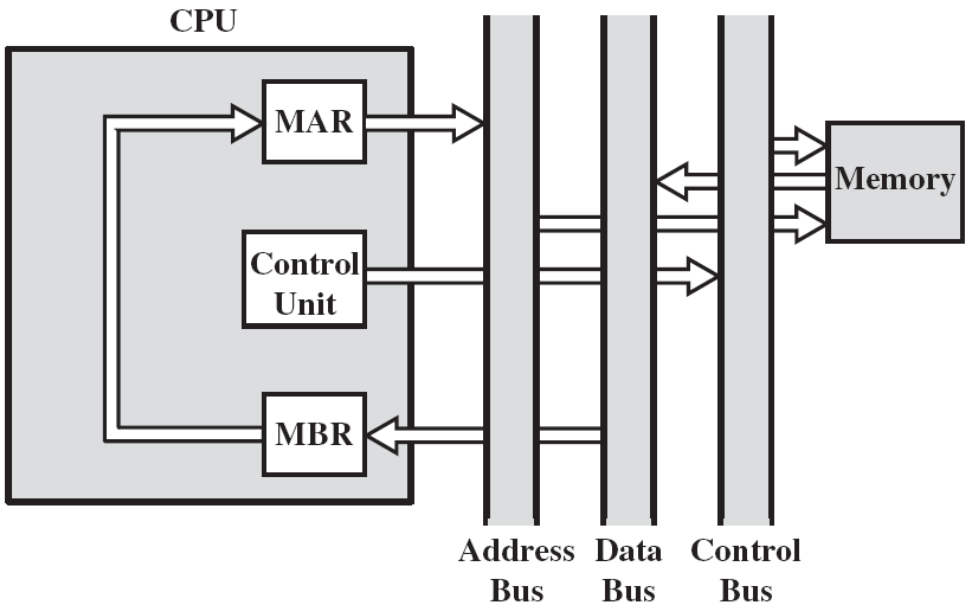
  PC <- (PC) +I
- t3: IR <- (MBR

- t1: MAR <- (PC)
- t2: MBR <- (memory)
- t3: PC <- (PC) +1

  IR <- (MBR)

重点知识点

- 间接周期的微操作过程



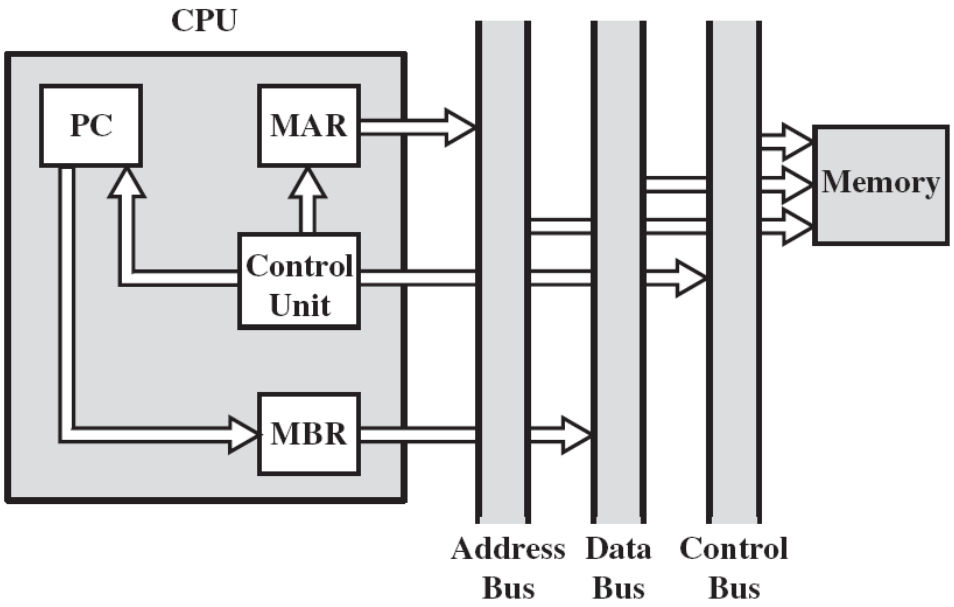
t1: MAR ← (IR(address))

t2: MBR ← Memory

t3: IR(address) ← (MBR(address))

重点知识点

- 中断周期的微操作过程



t1: MBR ← (PC)

t2: MAR ← save_address

PC ← Routine_address

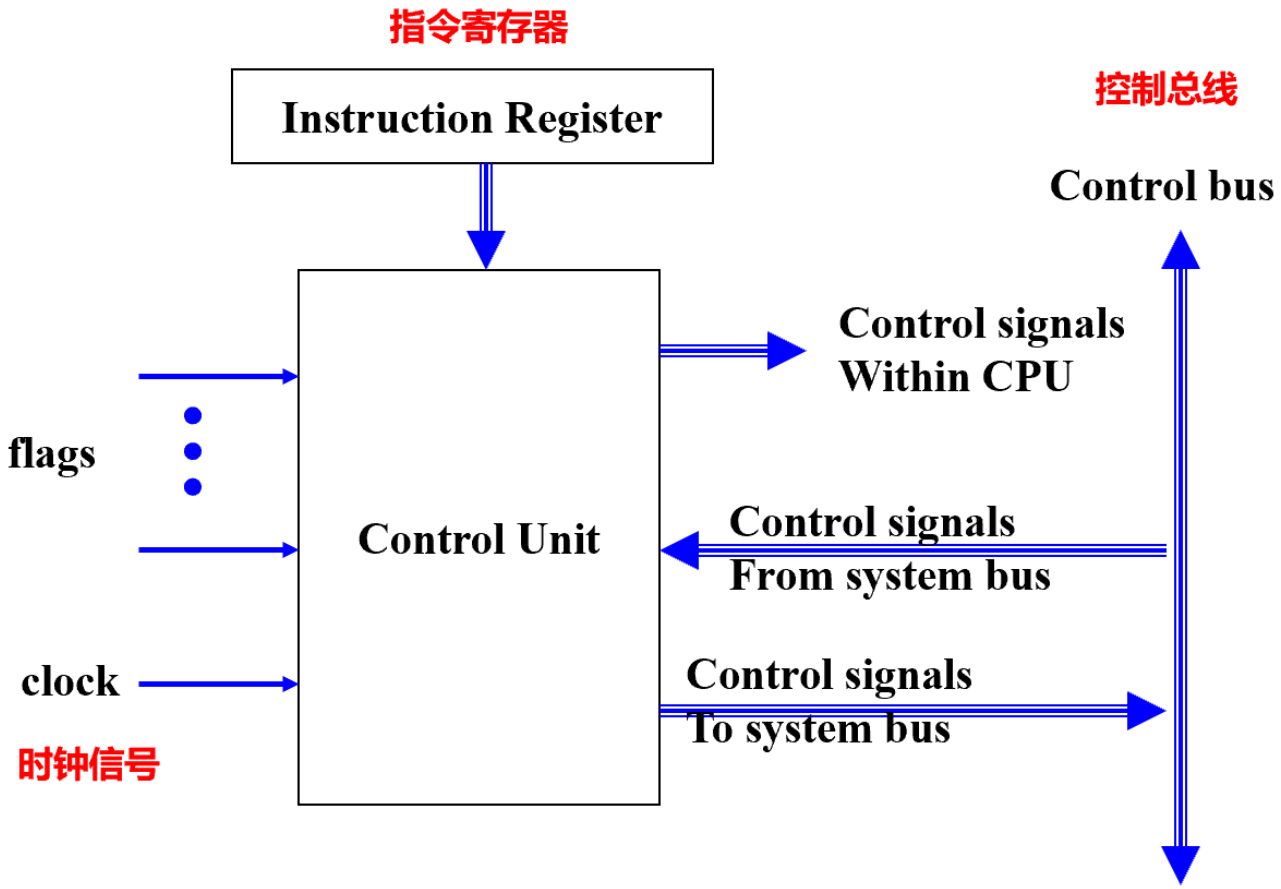
t3: Memory ← MBR

重点知识点

- 控制器的设计要求
  - 确定处理器的基本元素，也就是控制对象：ALU，Register，内部总线，外部总线，CU
  - 描述处理器需要完成的操作，也就是做什么：内部数据传送，内部和外部的数据传送，算术或逻辑运算
  - 确定处理器的各个部件如何执行这些操作：定序和执行。同时产生一组控制信号来实现：
    - 定序：使CPU按照顺序执行一系列微操作
    - 执行：控制完成每个微操作

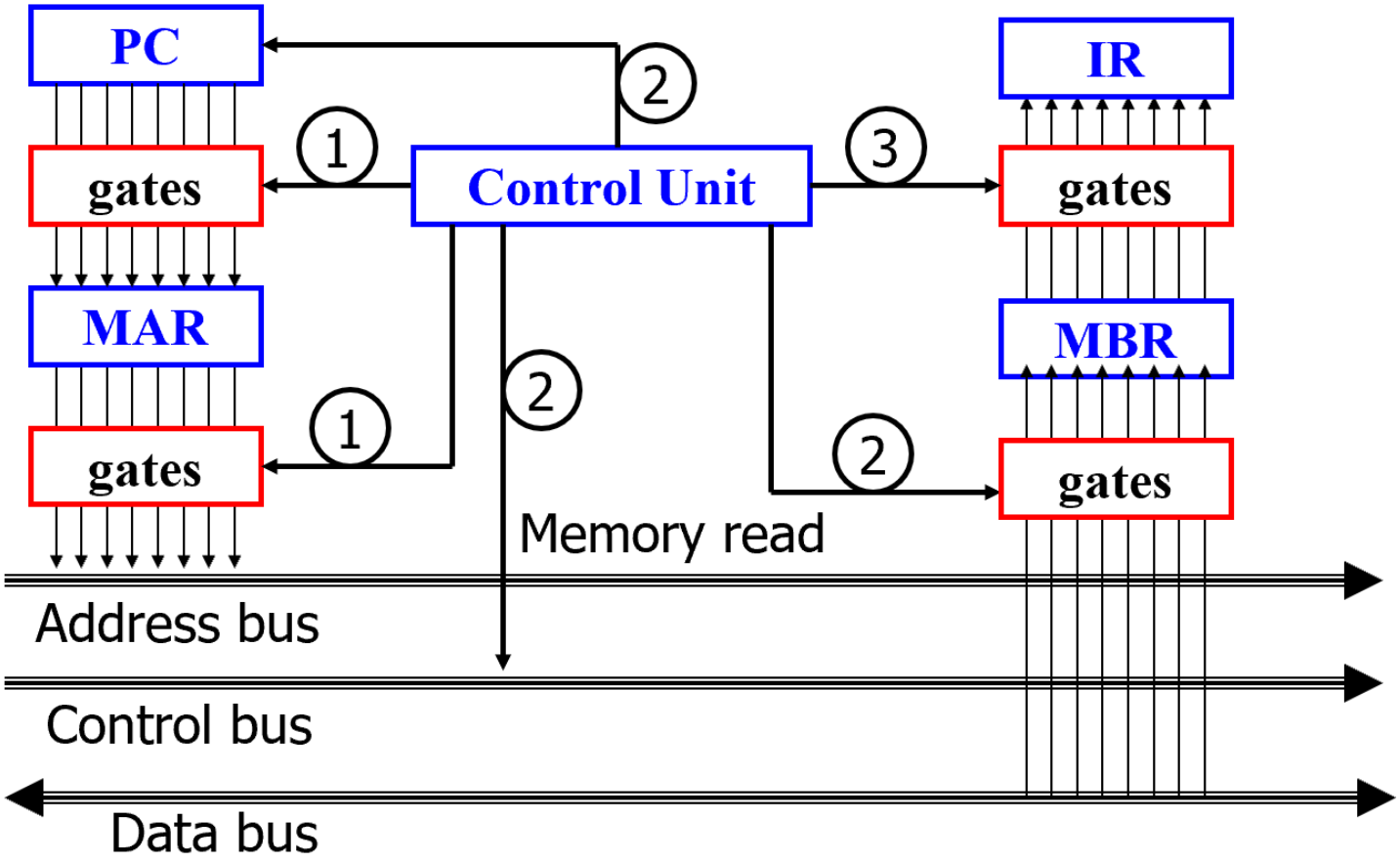
重点知识点

- 控制器模型


指令寄存器
Instruction Register
控制总线
Control bus
Control signals
Within CPU
flags
Control Unit
Control signals
From system bus
clock
Control signals
To system bus
时钟信号

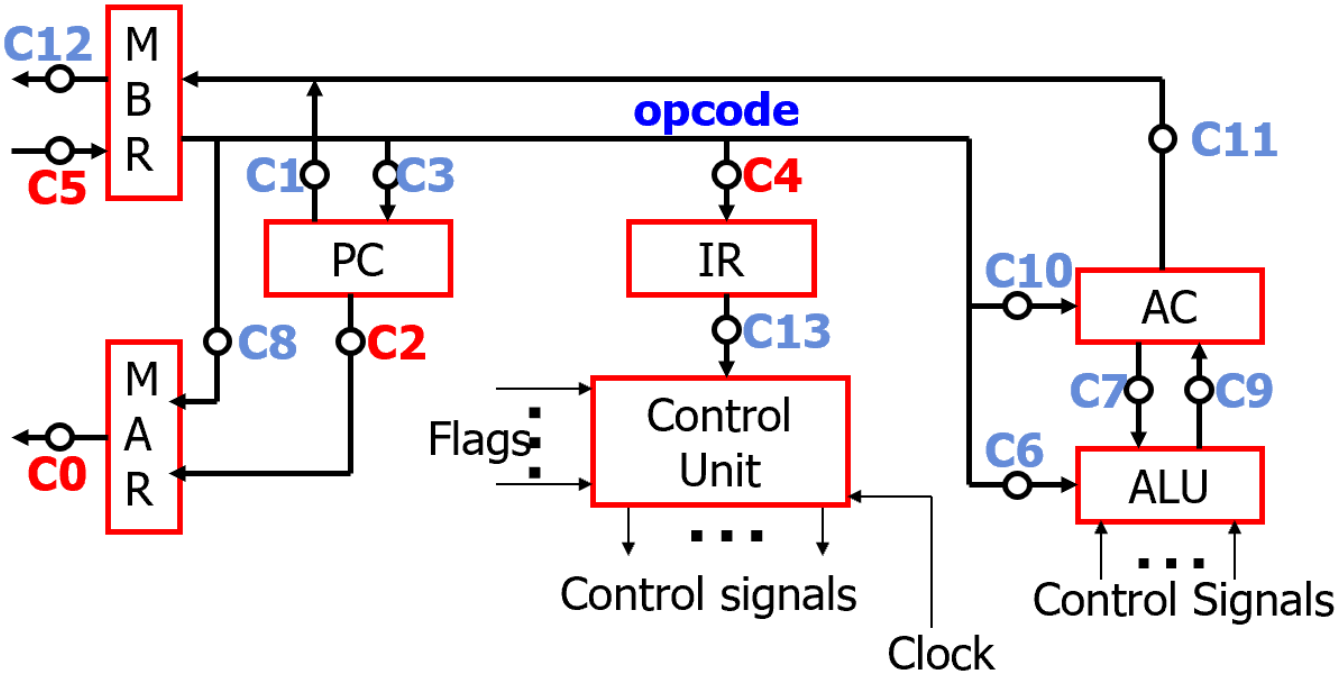
重点知识点

- 取指周期的控制信号



重点知识点

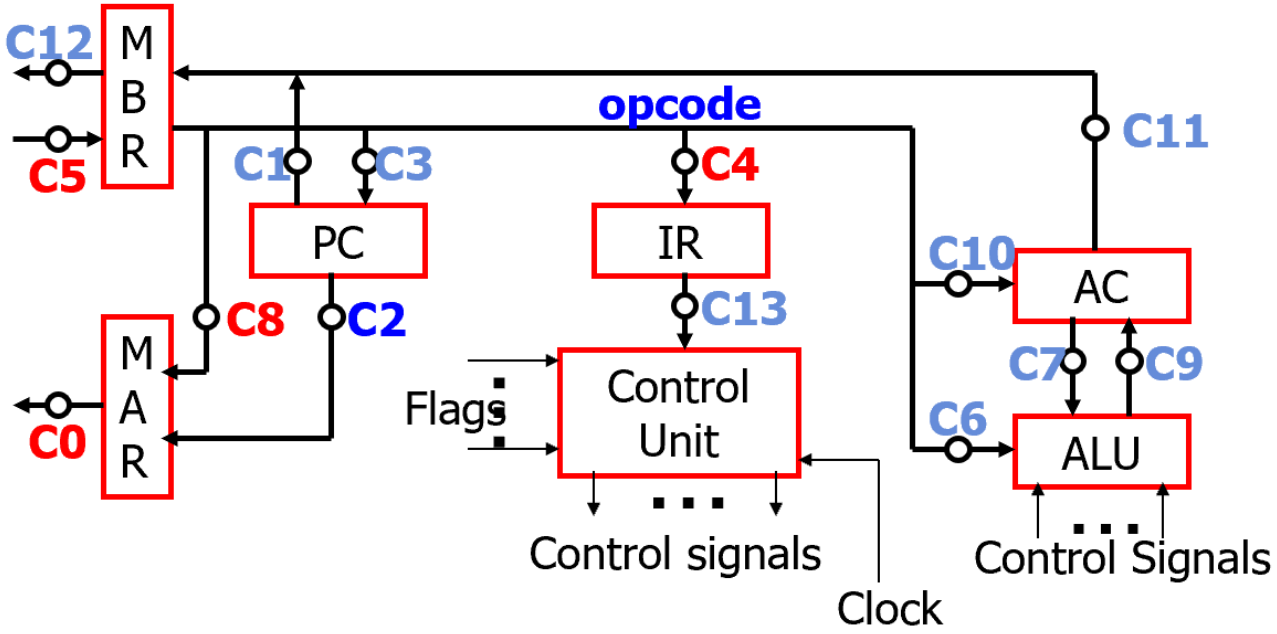
- 取指周期的门控制



Fetch:	**t1: MAR ← (PC)**	**C2, C0**
	t2: MBR ← Memory	**C5, CR**
	PC ← (PC) + I	
	t3: IR ← (MBR)	**C4**

重点知识点

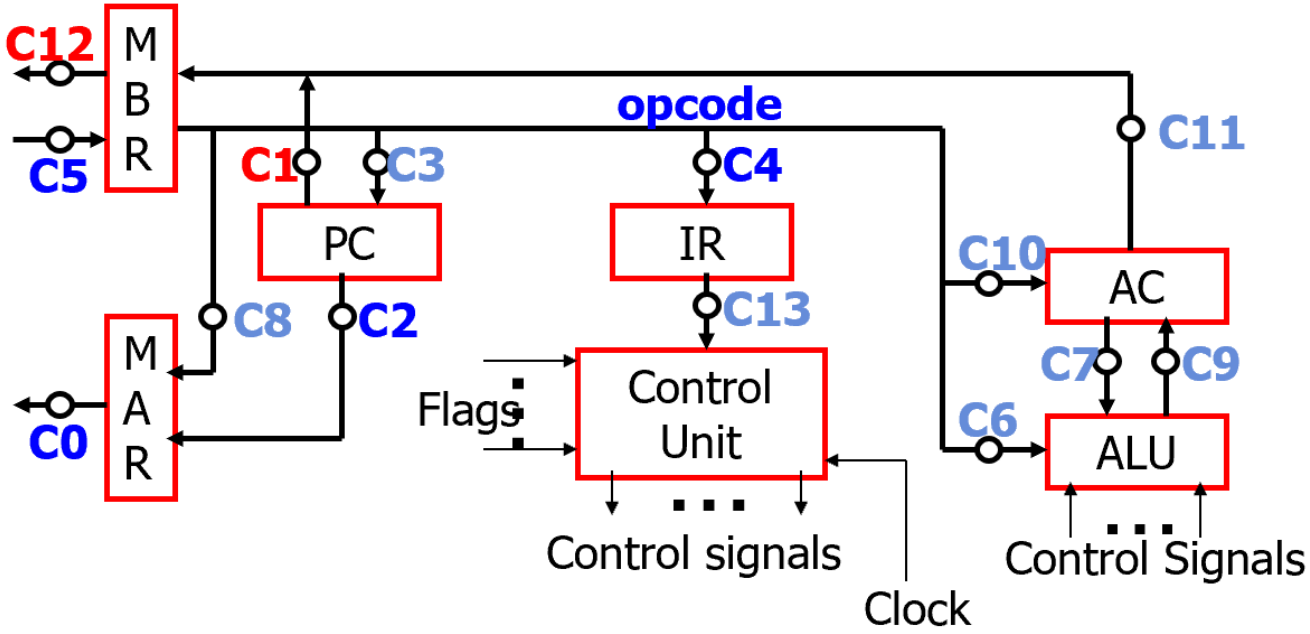
- 间接周期的门控制



Indirect:	**t1: MAR ← (IR(address))**	**C8, C0**
	t2: MBR ← Memory	**C5, CR**
	t3: IR(address) ← (MBR(address))	**C4**

重点知识点

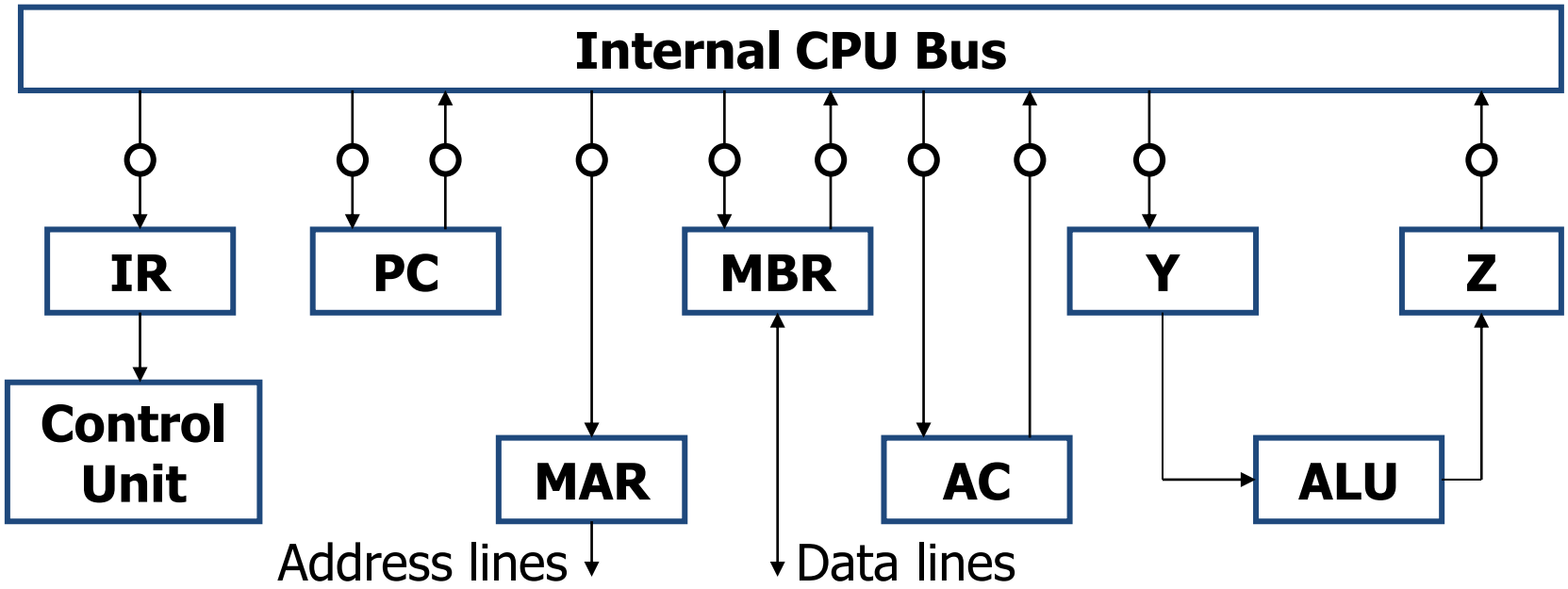
- 中断周期的门控制



Interrupt:	**t1: MBR ← (PC)**	**C1**
	t2: MAR ← Save-address	
	t2: PC ← Routine-address	
	t3: Memory ← (MBR)	**C12, CW**

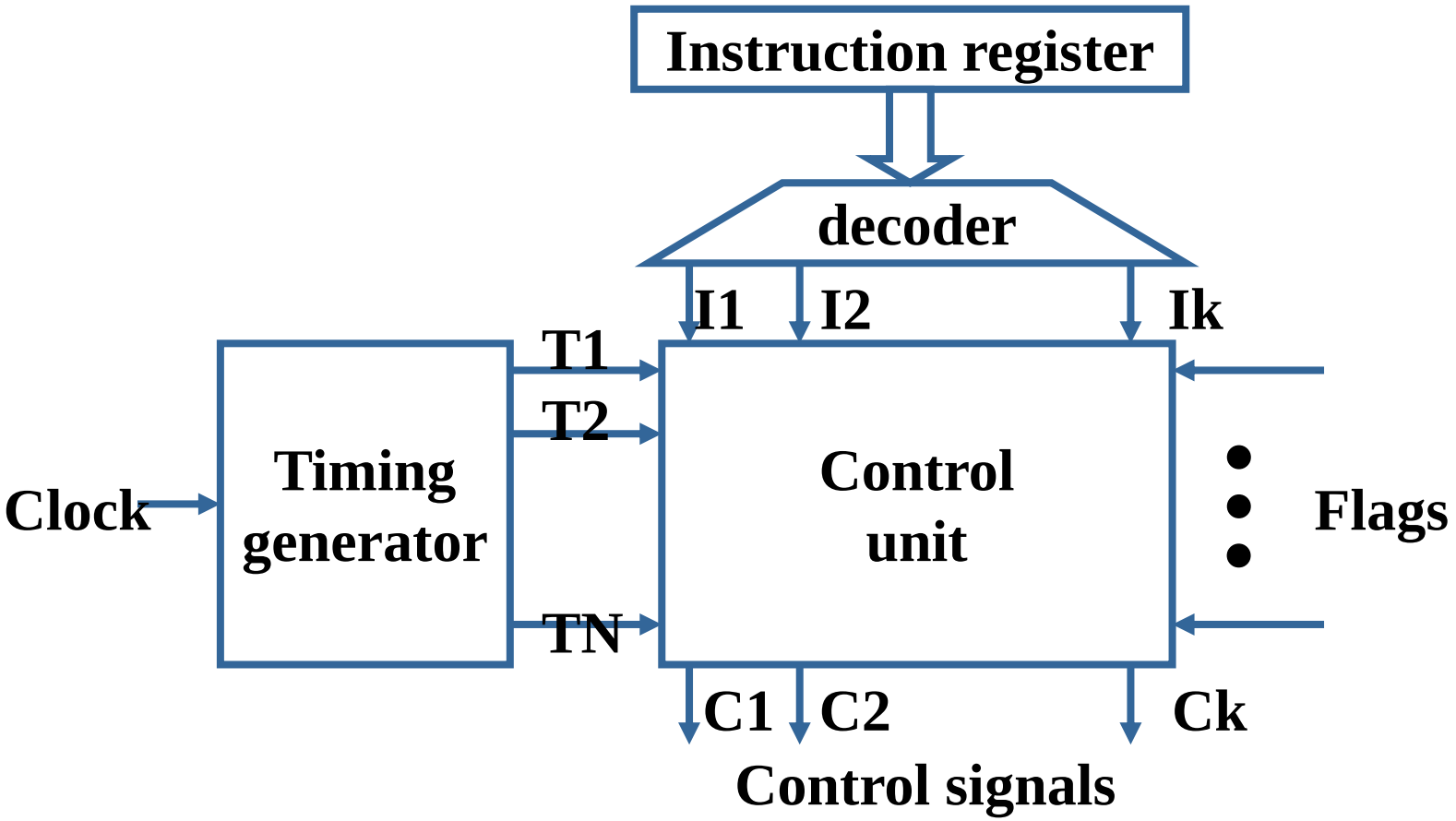
重点知识点

- 处理器内部的模块组织



重点知识点

- 控制器逻辑


Instruction register
decoder
I1
I2
Ik
T1
T2
TN
Clock
Timing generator
Control unit
Flags
C1
C2
Ck
Control signals

重点知识点

- 硬布线实现方式
  - 控制器的输入包括：指令寄存器、时钟信号、标志、控制总线
  - 控制单元为每一个控制信号设计一个布尔表达式。根据这个布尔表达式，可以控制这个控制信号是有效还是无效的
  - 控制信号的布尔表达式比较复杂，所以通常由PLA，Programmable logic array，可编程逻辑阵列来完成
  - 对于更复杂的控制，则还会通过分层的方式来实现，这样可以提高速度

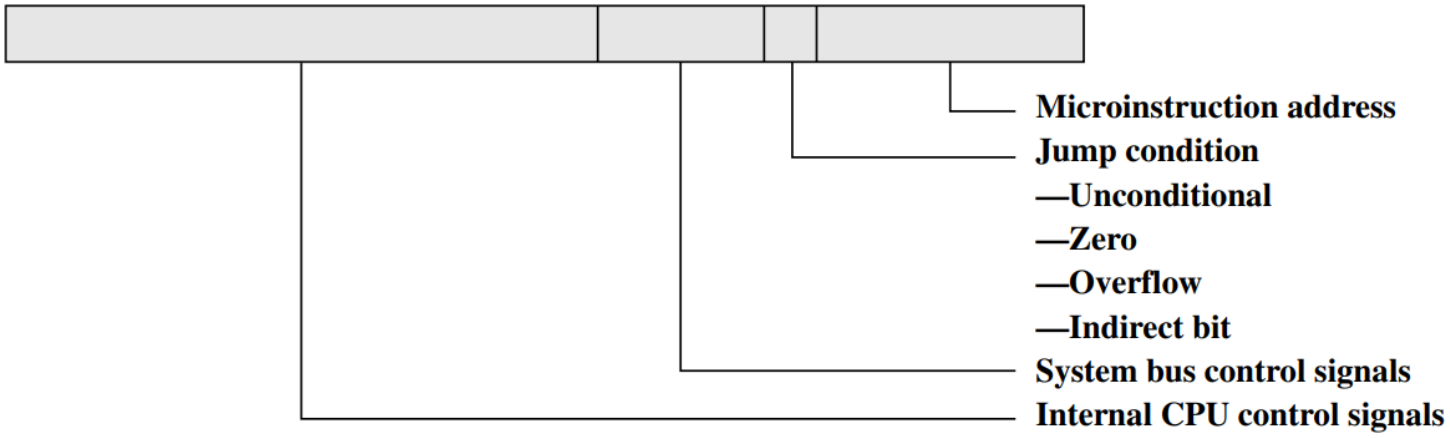
重点知识点

- 微程序控制实现方式
  - 对每一个微操作，构造一个控制字，表示这个微操作需要的控制信号，这个控制字就是微指令
  - 指令的周期包括若干个这样的微指令组成的序列
  - 控制信号由微指令产生
  - 包括水平微指令和垂直微指令这两种方式

Internal CPU control signals	System bus control signals	Jump conditions	Micro-instruction address

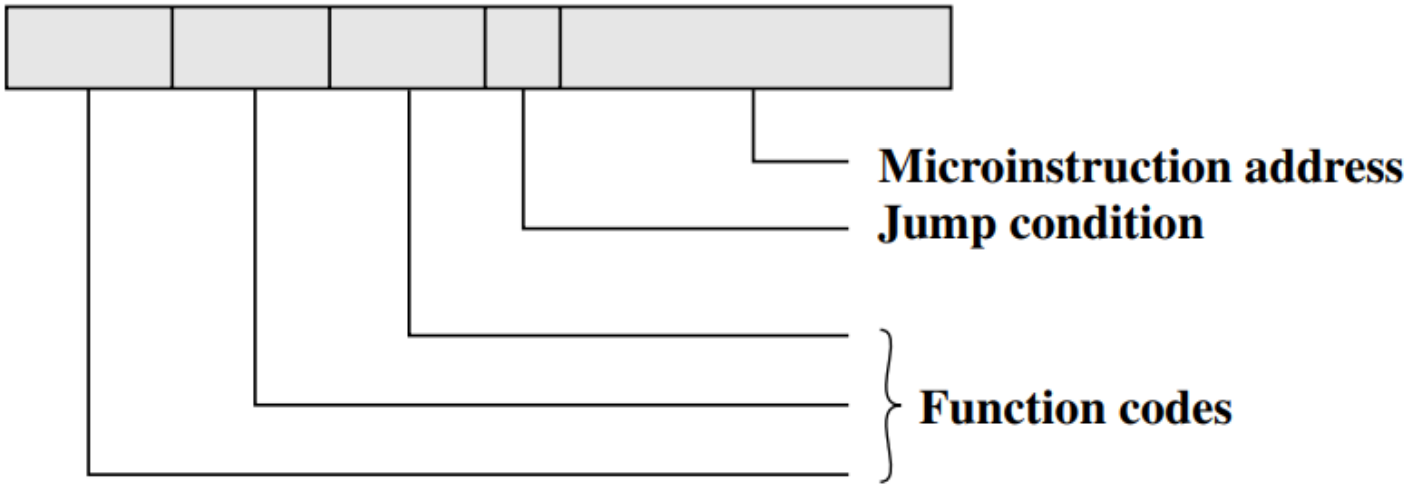
重点知识点

- 水平微指令
  - 每个控制信号占1个bit单独控制
  - 微指令地址字段的宽度决定了控制存储器的空间
  - 转移条件的位数由控制转移的判断标志数量来确定
  - 存储字宽，高并发



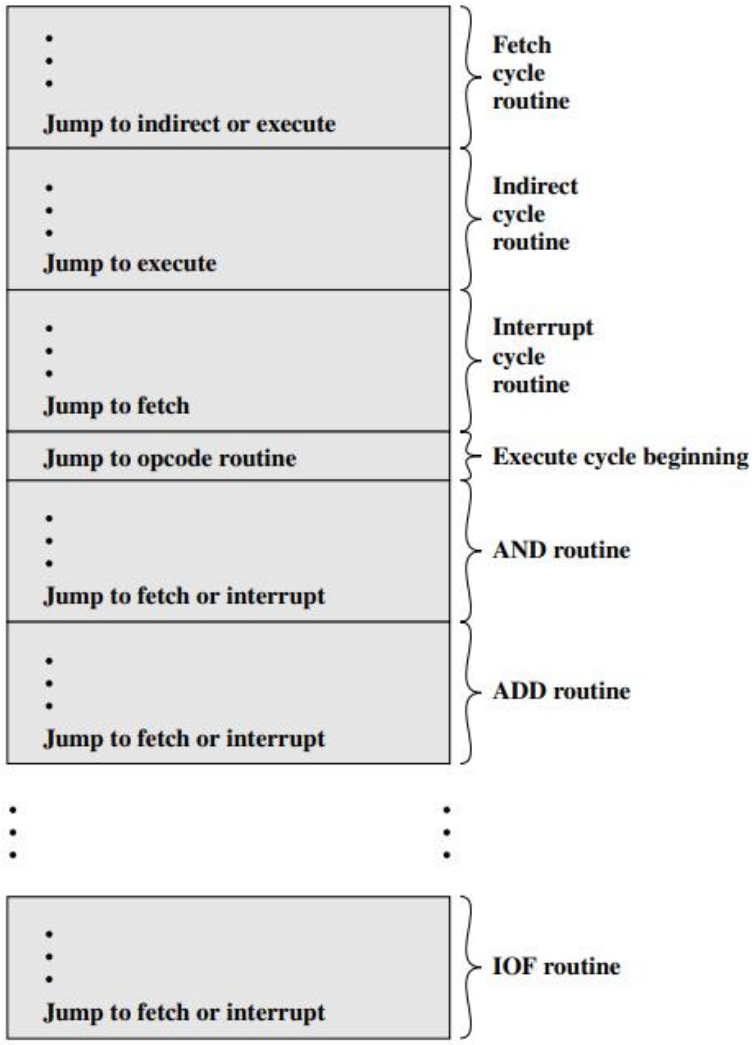
重点知识点

- 垂直微指令
  - 控制信号需要将功能码通过译码器进行转换
  - 存储字较短，高并有限
  - 速度慢



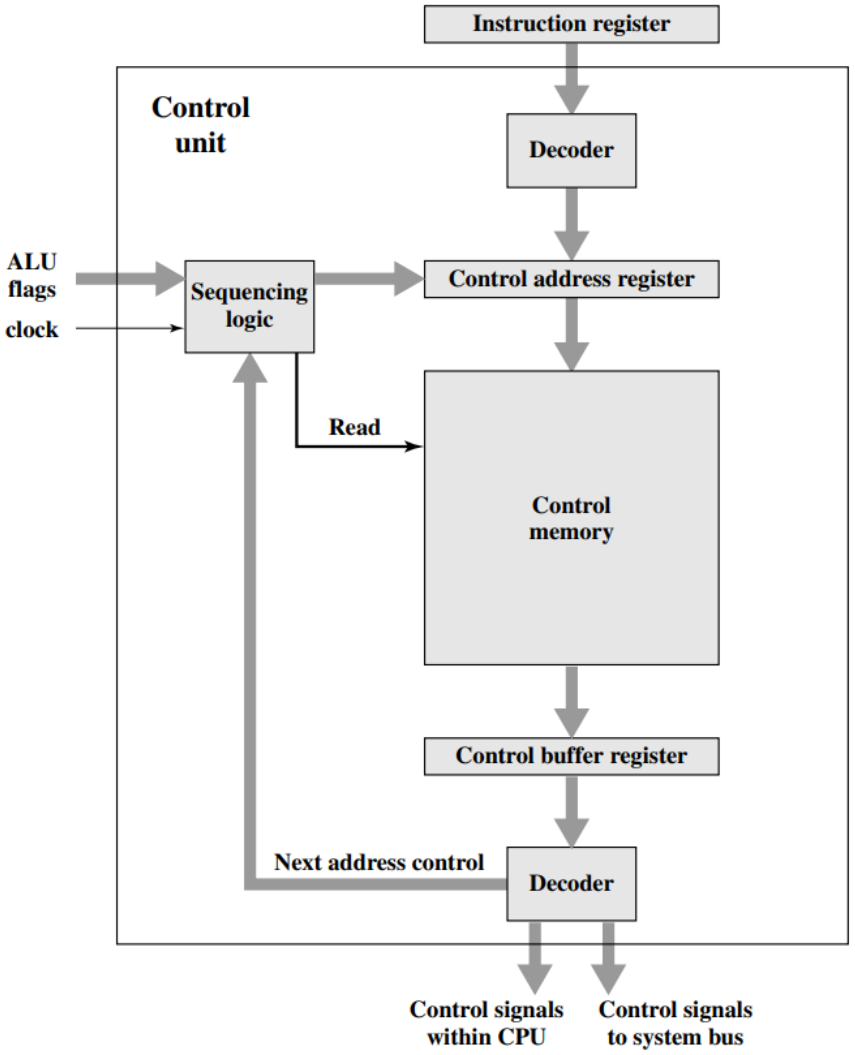
重点知识点

- 控制存储器
  - 控制字被放入一个称为控制存储器的特殊内存块中
  - 微指令组成成不同的例程
  - 每个例程顺序执行
  - 例程结束的地方有分支



重点知识点

- 微程序控制器工作方式
  - 定序逻辑读指令
  - 控制地址寄存器中的字通过控制存储器查询到控制字，送入控制缓冲寄存器
  - 控制缓冲寄存器生成控制信号、下一个微指令的地址
  - 定序逻辑根据控制缓冲寄存器、ALU标志等，确定下一个微指令地址

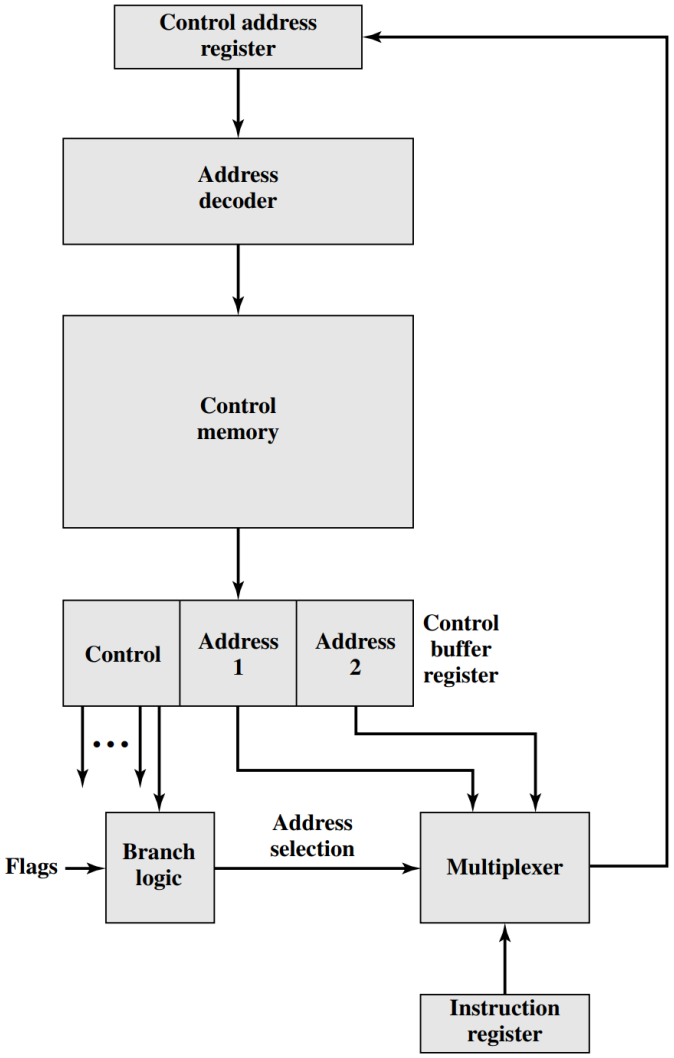


重点知识点

- 微程序定序
  - 基于当前微指令，条件标志，IR的内容, 生成下一个微指令在控制寄存器中的地址
  - 微指令转移的三种方式：
    - 双地址字段：微指令中给出了2个地址，用于进行转移选择
    - 单地址字段：微指令中只有1个地址，复用控制地址寄存器的地址
    - 可变格式：通过标志位来确定格式。

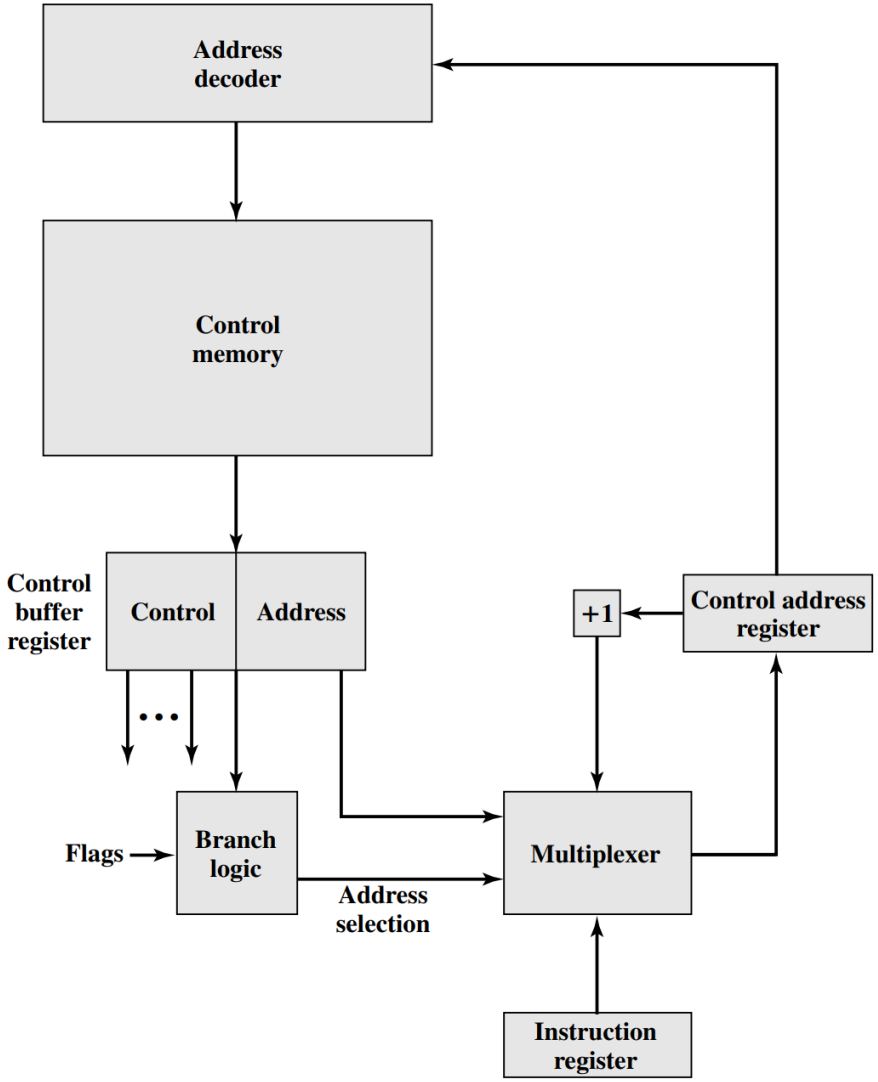
重点知识点

- 微程序定序-双地址



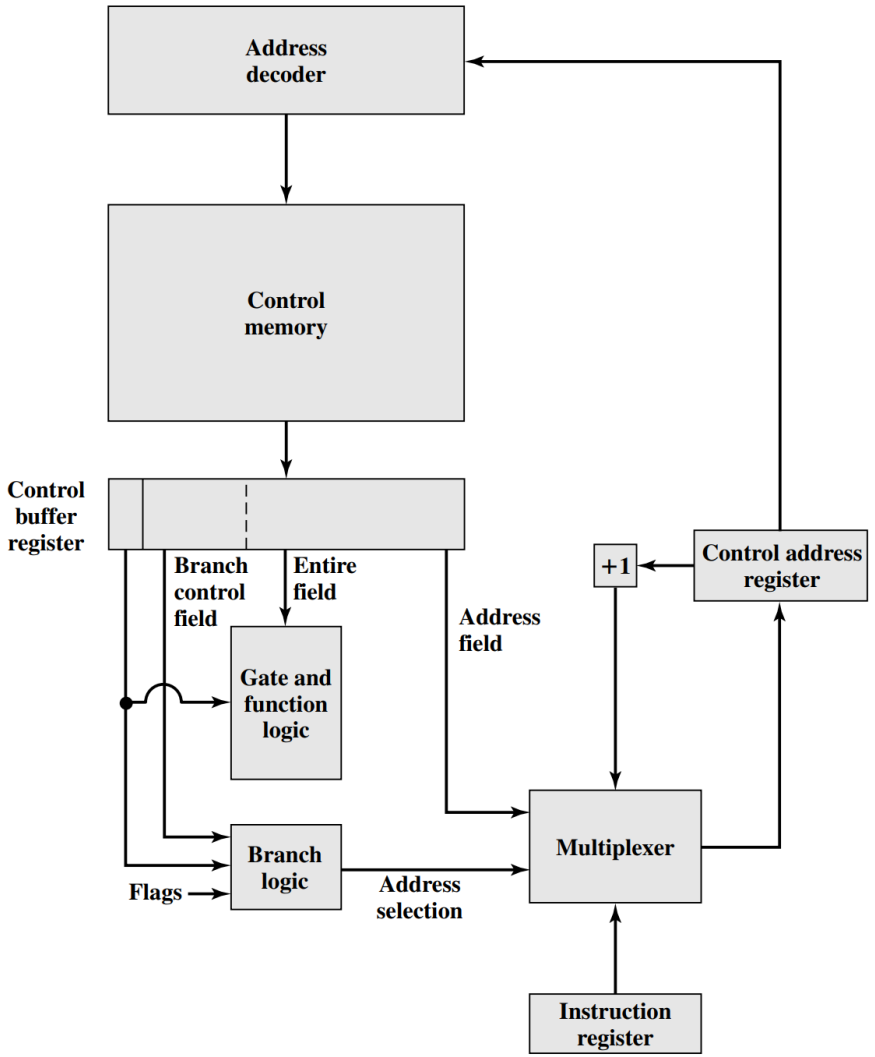
重点知识点

- 微程序定序-单地址



重点知识点

- 微程序定序-可变格式



重点知识点

- 微指令执行
  - 微指令的执行实际也是一个小的周期性动作，我们称这个周期为微指令周期
  - 微指令周期需要完成：读取微指令，微指令的执行
  - 执行的目的是产生控制信号，读取微指令的时候，就产生了控制信号
  - 一些控制信号发到处理器内部，一些发到控制总线或者其他接口

第十七~十八章　并行组织

- 基本概念
  - 对称多处理
  - 集群
  - 非均匀存储器访问
  - 向量计算
  - 多核处理器

第十七~十八章　并行组织

- 重点知识点
  - 并行处理的组织方式
  - 多核的发展原因和组织方式

基本概念

- 对称多处理　Symmetric Multiprocessor Organization
  - 紧耦合　tightly coupled
  - 多个功能相似的处理器共享单一的存储器和I/O
  - 处理器都挂在共享的总线上，通过总线来访问存储器和I/O
  - 每个处理器访问内存区域的访问时间大致相同
  - 由集中的操作系统控制
  - 高性能、高可靠性、可扩展性、缩放性

基本概念

- 集群 Cluster
  - 松耦合 loosely coupled
  - 由若干个独立的单处理器或者对称多处理器构成，相互连接形成集群
  - 通过固定路径或网络进行连接，实现相互间的通信
  - 高性能、高可用性
  - 可扩展性比SMP更强
  - 性价比更好

基本概念

- 非均匀存储器访问　NUMA
  - 紧耦合　tightly coupled
  - SMP对处理器的数量有实际限制
  - 集群中，每个结点都有自己的内存，应用程序看不到全局内存
  - NUMA保留SMP风格，同时提供大规模多处理
  - 不同的处理器以不同的速度访问不同的内存区域
  - 不同处理器的cache之间保持cache一致性

基本概念

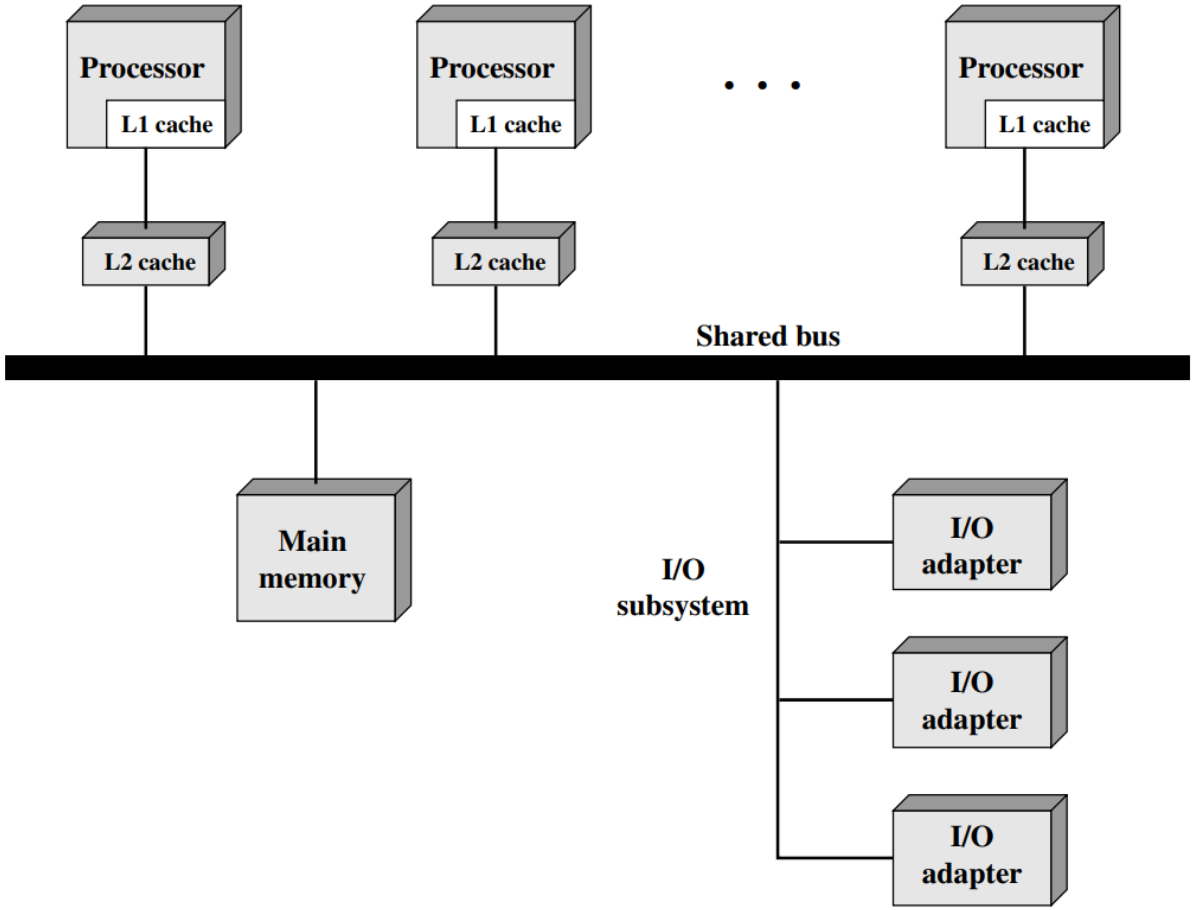
- 向量计算　Vector computation
  - 涉及物理过程的数学问题计算量巨大，并且大部分是向量计算
  - 超级计算机价格昂贵
  - 采用阵列处理器，只处理问题中的向量部分

重点知识点

- 多处理器类型
  - SISD：Single instruction, single data stream 单指令单数据流
    - 单个处理器，单指令流，数据在单一存储器，单一内核处理器
  - SIMD Single instruction, multiple data stream 单指令多数据流
    - 单个控制单元，多个处理单元，每个处理单元都有辅助的存储器，由不同的处理器在不同的数据集上执行的每条指令。典型的是向量和数组处理器
  - MISD Multiple instruction, single data stream 多指令单数据流
    - 没有实现过
  - MIMD Multiple instruction, multiple data stream 多指令多数据流
    - 一组处理器同时执行不同的指令序列，典型的是对称多处理，集群，NUMA

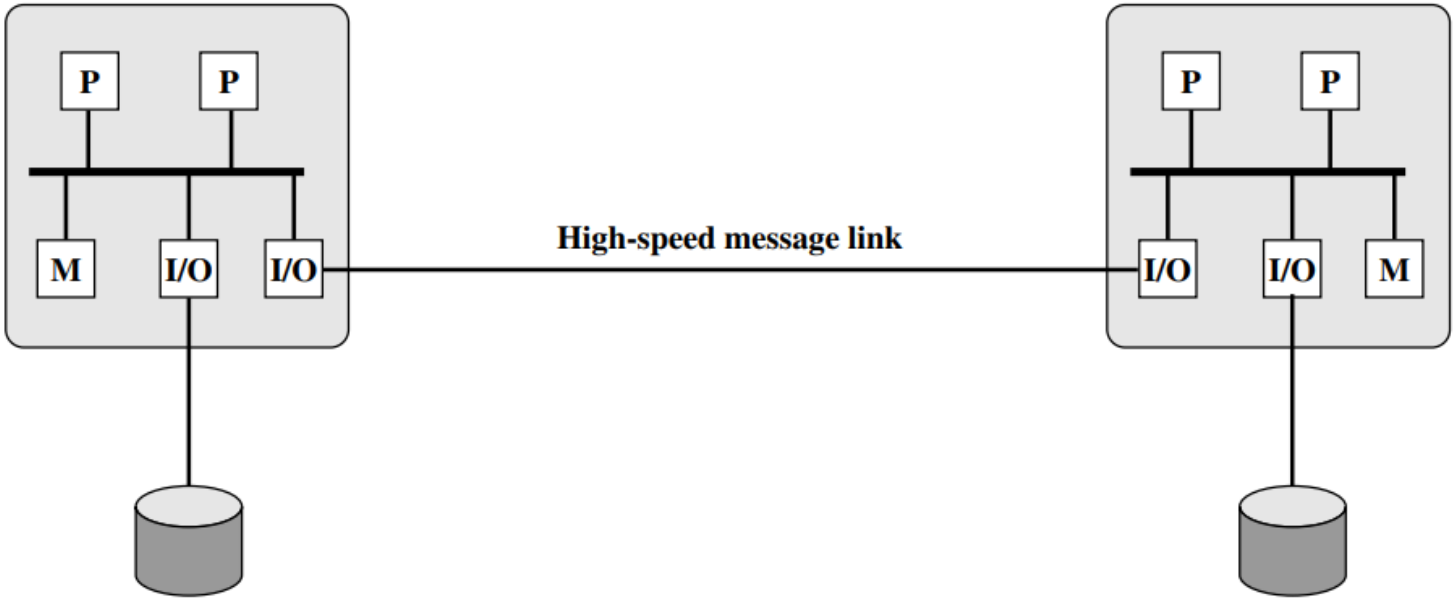
重点知识点

- 对称多处理结构



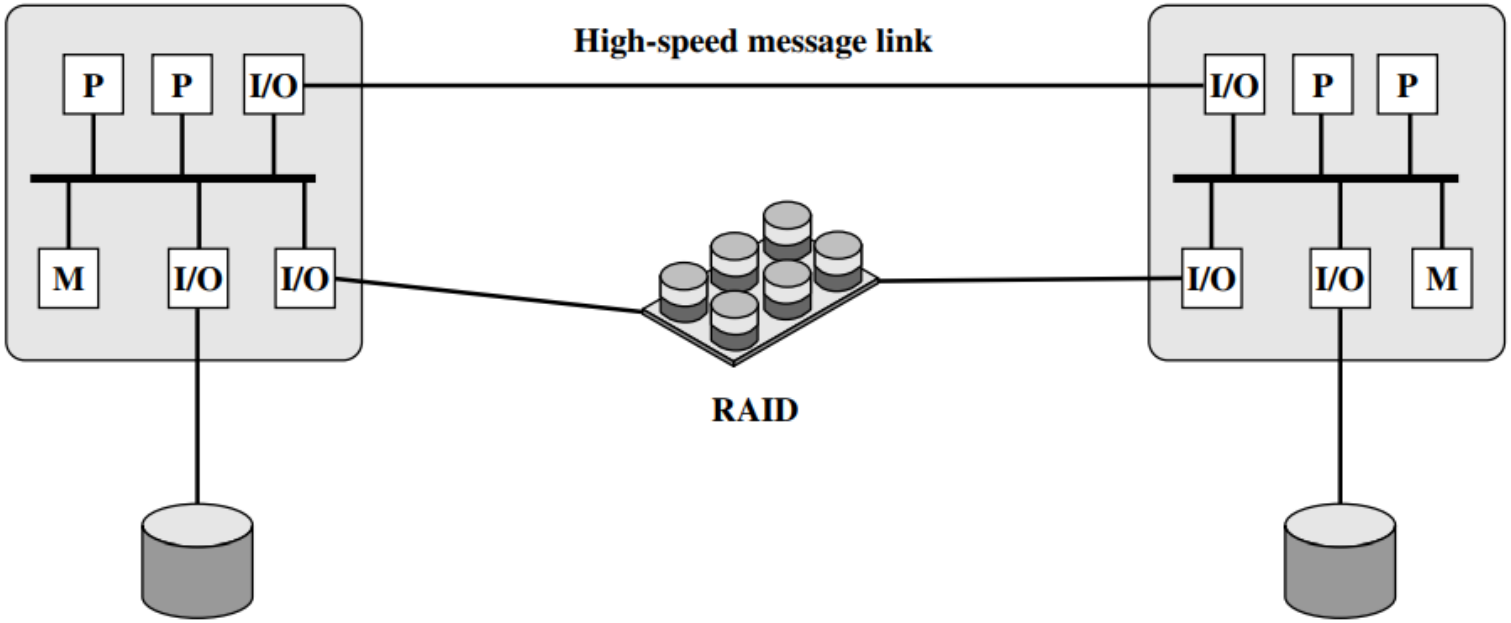
重点知识点

- 集群结构1：不共享磁盘



重点知识点

- 集群结构2：共享磁盘



重点知识点

- 集群结构的实现——刀片服务器　blade servers
  - 服务器包含多个服务器模块（刀片服务器) 在单机箱中，由机箱提供电源给各个服务器
  - 每个刀片服务器都有处理器、内存和磁盘
  - 节省空间
  - 简化系统管理

重点知识点

- 集群和对称多处理的对比
  - 两者都为高需求应用程序提供多处理器支持
  - 两者都可以在市场上买到，SMP的上市时间更长一点
  - SMP易于管理和控制，更接近单处理器系统
  - 集群：扩张性更强，可用性也很好，价格便宜

重点知识点

- 多核的发展原因--处理器硬件性能问题
  - 芯片架构的调整，还有主频的提高，硬件性能得到了快速提升
  - 处理器复杂度越来越高，需要更多的逻辑电路、内部连接以及控制信号
  - 芯片的设计和制造、调试更加困难
  - 芯片能耗需求随着芯片密度和时钟频率呈指数增长
  - 性能增长与复杂度的平方根成正比
  - 多核处理器的处理能力和核的数量接近于线性关系

重点知识点

- 多核的发展原因--软件性能问题
  - 性能取决于有效利用并行资源的能力
  - 少量的串行代码也会影响性能
  - 有些应用能够有效利用多核处理器：数据库，处理独立事务的服务器，多线程本地应用，Java应用，多实例应用，游戏等等

重点知识点

- 几种典型的多核处理器
  - Intel Core Duo：
    - 2个超标量处理器，独享L1 cache，共享L2 cache，每个核都有热控制单元
  - Intel Core i7
    - 4个 x86 并发多线程处理器（SMT， simultaneous multi-threading ），每个支持4个线程，看起来像是16个内核。独享L2 cache，共享L3 cache
  - ARM 11 MPCore
    - 最多4个处理器核，每个核有自己的数据和指令cache。每个核称为MP11。每个核可能会配置Vector floating-point unit 向量浮点单元