回顾第15次课

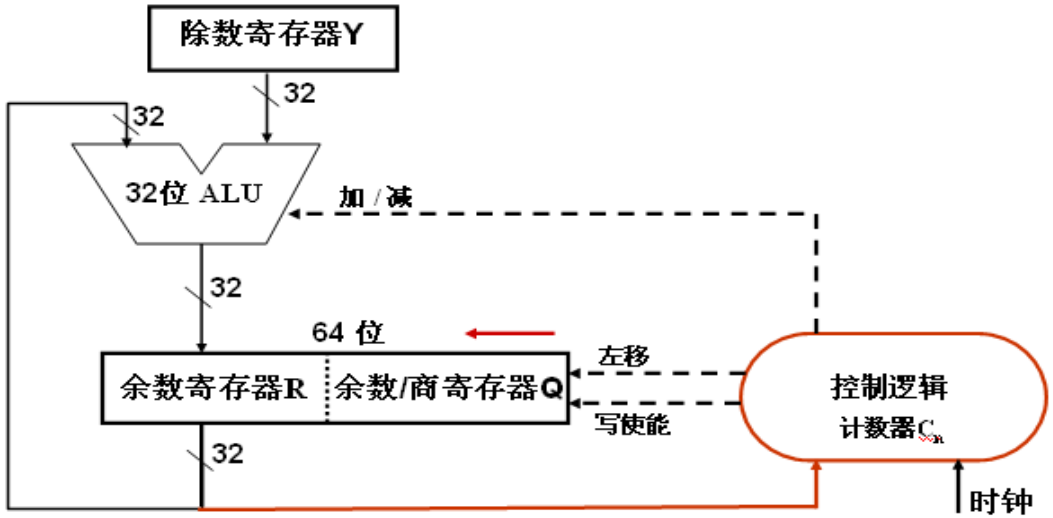
除法运算：（n位除以n位， 2n位除以n位，整数、小数）

无符号数除法：用“加/减”+“左移”，有恢复余数和不恢复余数两种

原码除法：符号和数值分开，数值部分用无符号数除法实现，用于浮点数尾数除法运算。

补码除法：符号位和数值位一起。不恢复余数法。

- **定点运算部件**
- **运算器**
- **数据通路**



实现同一个功能的硬件逻辑可以有很多种
就好像同样的软件功能，实现的代码可以千差万别
布线规模？布线方便程度？
是否易于增删改和复用？运行效率高低？
——最终都是一个权衡利弊、按实际需要进行取舍的过程

第7章 指令系统

第1讲 概述与指令系统设计

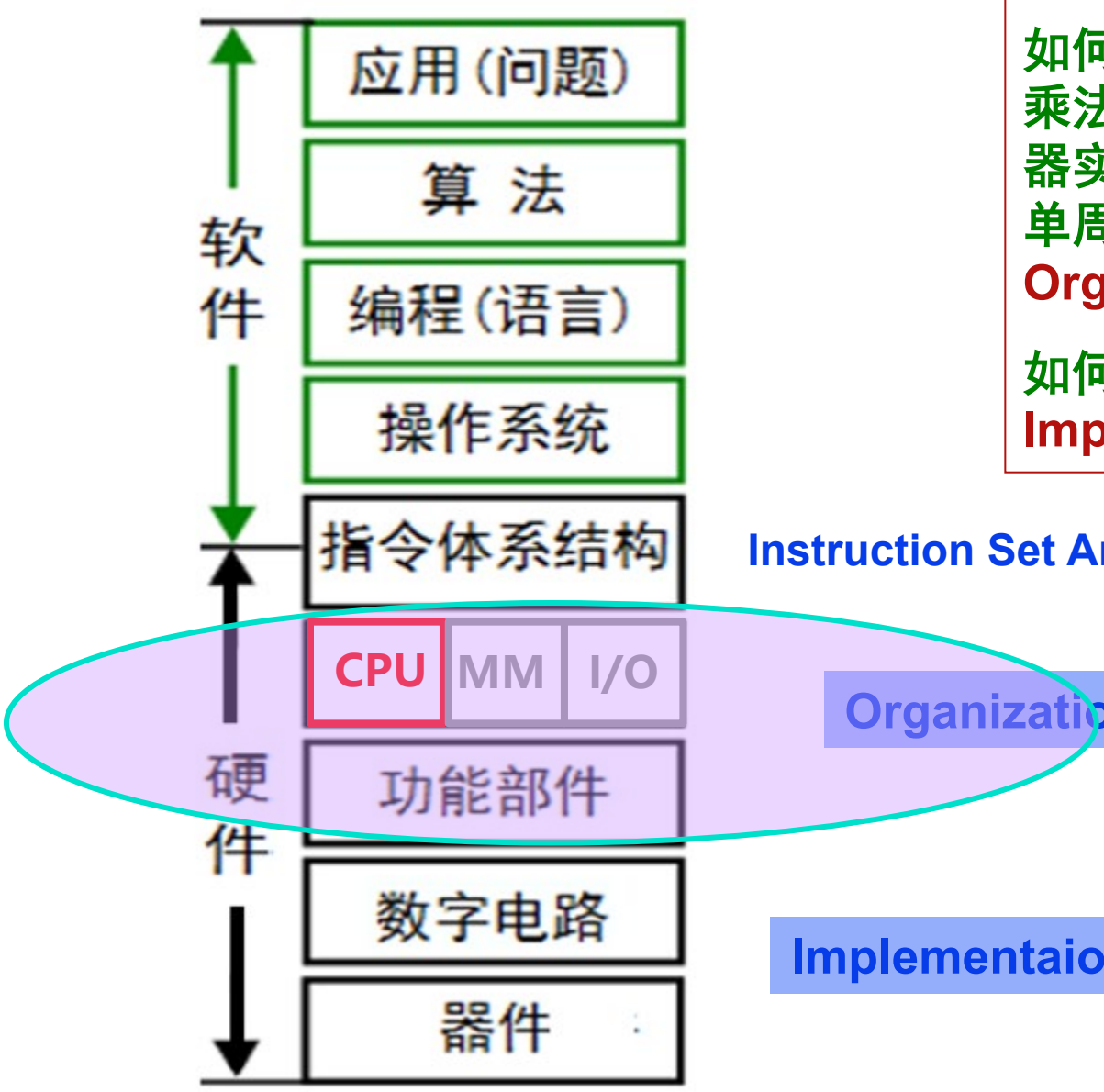
第2讲 指令系统实例：RISC-V架构

计算机组成，ISA

是否提供“乘法指令”：ISA

如何实现乘法指令（用专门乘法器还是用加法器+移位器实现）、如何实现CPU（单周期？流水线？etc）：Organization

如何布线、用什么材料等：Implementation



Instruction Set Architecture

Organization，组成

Implementaion，实现

例子：RISC-V中整数的乘、除运算处理

ISA要素：指令+数据类型+寄存器设计等

- **乘法指令: mul，mulh，mulhu，mulhsu**
  - mul rd, rs1, rs2：将低32位乘积存入结果寄存器rd
  - mulh、mulhu：将两个乘数同时按带符号整数（mulh）、同时按无符号整数（mulhu）相乘，高32位乘积存入rd中
  - mulhsu：将两个乘数分别作为带符号整数和无符号整数相乘后得到的高32位乘积存入rd中
  - 得到64位乘积需要两条连续的指令，其中一定有一条是mul指令，实际执行时只有一条指令
  - 两种乘法指令都不检测溢出，而是直接把结果写入结果寄存器。由软件根据结果寄存器的值自行判断和处理溢出

- **除法指令: div ，divu，rem，remu**
  - div / rem：按带符号整数做除法，得到商 / 余数
  - divu / remu：按无符号整数做除法，得到商 / 余数

C\C++\Python语言的语法设计

各种语言相应的、适用于X86（IA32）的编译器

各种语言相应的、适用于RISCV的编译器

X86（IA32）指令集设计

RISCV指令集设计

X86（IA32）芯片（CPU）

大量个人计算机

RISCV芯片（CPU）

采用RISC-V指令集架构的笔记本电脑ROMA

C\C++\Python语言编制的程序

各种语言相应的、适用于X86（IA32）的编译器

各种语言相应的、适用于RISCV的编译器

符合X86（IA32）指令集的机器指令序列

符合RISCV指令集的机器指令序列

CPU负责周而复始的执行指令

CPU负责周而复始的执行指令

系统程序员角度：通过系统来使用硬件，要求易于编写编译器

各种语言相应的编译器

计算机硬件

指令集体系结构（ISA）核心部分是指令系统，还包含数据类型和数据格式定义、寄存器设计、I/O空间的编址和数据传输方式等等

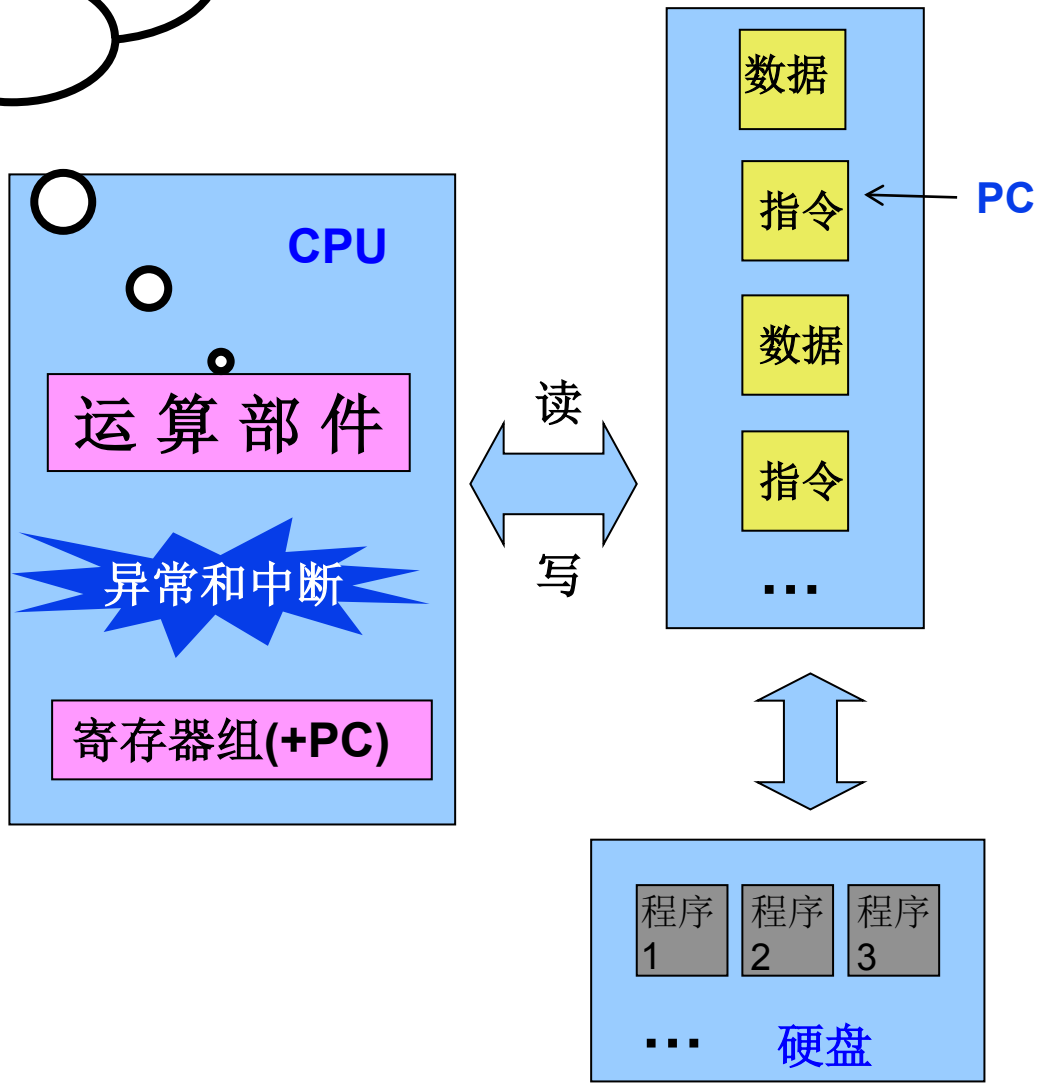
冯.诺依曼结构机器对指令规定：

- 由两部分组成：操作码（做什么事）和操作数或其地址码（对什么数据做）
- 和数据一样，是二进制且放在主存中

硬件设计者角度：指令系统为CPU提供功能需求，要求易于硬件设计

取指令，译码，取数，
运算，存数…
周而复始

内存（主存，实存）
物理地址



符合某指令集的
机器指令序列

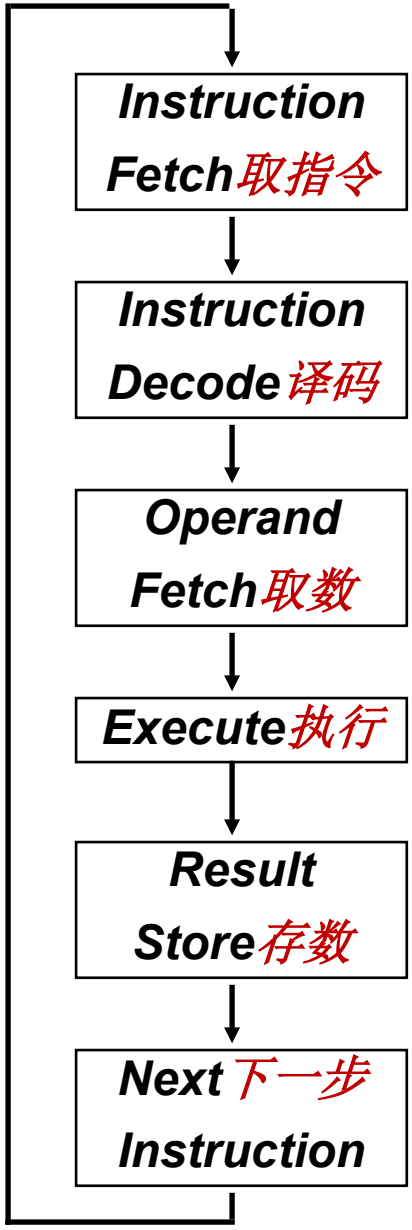
CPU负责周而复始的执行指令

第一讲 概述与指令系统设计

主 要 内 容

- 指令设计概述
- 操作数及其寻址方式
  - 立即 / 寄存器 / 寄存器间接 / 直接 / 间接 / 堆栈 / 偏移
- 操作类型和操作码编码
  - 定长编码法、变长扩展编码法
- 标志信息的生成与使用
- 指令设计风格
- 异常和中断处理机制

从指令执行周期看指令设计涉及的问题



从存储器取指令

指令地址、指令长度（定长/变长）

对指令译码，以确定将要做什么操作

指令格式、操作码编码、操作数类型

计算操作数地址并取操作数

地址码、寻址方式、操作数格式和存放

进行相应计算，并得到标志位

操作类型、标志或条件码

将计算结果保存到目的地

结果数据位置（目的操作数）

计算下条指令地址（通常和取指令同时进行）

下条指令地址（顺序 / 转移）

指令执行的每一步都可能发生异常或中断，因此，指令集系统架构（ISA）还需要考虑异常和中断机制

一条指令须包含的信息

指令格式 | 0000 00 | 10 001 | 1 0010 | 0100 0 | 000 0010 0000

一条指令必须明显或隐含包含的信息有哪些？

1 操作码：指定操作类型（对何种类型数据做何种操作）

（操作码长度：固定 / 可变）

2 源操作数参照：一个或多个源操作数所在的地址

（操作数来源：主(虚)存/寄存器/I/O端口/指令本身）

3 结果值参照：产生的结果存放何处（目的操作数）

（结果地址：主(虚)存/寄存器/I/O端口）

4 下一条指令地址：下条指令存放何处

（下条指令地址 ：主(虚)存）

（正常情况隐含在PC中，改变顺序时由指令给出）

与指令集设计相关的重要方面

- 操作码的全部组成：操作码个数 / 种类 / 复杂度

  LD/ST/INC/BRN 四种指令已足够编制任何可计算程序，但程序会很长
- 数据类型：对哪几种数据类型完成操作
- 指令格式：指令长度 / 地址码个数 / 各字段长度
- 通用寄存器：个数 / 功能 / 长度
- 寻址方式：操作数地址的指定方式
- 下条指令的地址如何确定：顺序，PC+1；条件转移；无条件转移；……
- 异常和中断机制，包括存储保护方式等

指令格式的设计

指令格式的选择应遵循的几条基本原则

- 应尽量短
- 要有足够的操作码位数
- 合理地选择地址字段的个数
- 指令编码必须有唯一的解释，否则是不合法的指令
- 指令字长应是字节的整数倍
- 指令尽量规整

一般通过对操作码进行不同的编码来定义不同的含义，操作码相同时，再由功能码定义不同的含义

一条指令中应该有几个地址码字段？

零地址指令

（1）无需操作数　如：空操作 / 停机等

（2）所需操作数为默认的　如：堆栈 / 累加器等

形式：

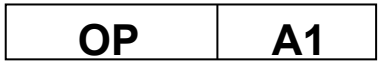
一地址指令

其地址既是操作数的地址，也是结果的地址

（1）单目运算：如：取反 / 取负等

（2）双目运算：另一操作数为默认的　如：累加器等

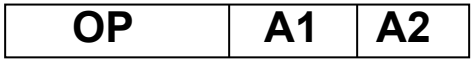
形式：



二地址指令（最常用）

分别存放双目运算中两个操作数，并将其中一个地址作为结果的地址。

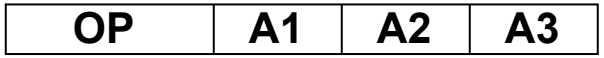
形式：



三地址指令（RISC风格）

分别作为双目运算中两个源操作数的地址和一个结果的地址。

形式：



多地址指令

用于成批数据处理的指令，如:向量 / 矩阵等运算的SIMD指令。

操作数类型和存储方式

操作数是指令处理的对象，与高级语言数据类型对应，基本类型有哪些?

地址（指针）

被看成无符号整数，寄存器编号，也可参加运算以确定主(虚)存地址

数值数据

定点数(整数)：一般用二进制补码表示

浮点数(实数)：大多数机器采用IEEE754标准

十进制数：用NBCD码表示，压缩/非压缩（汇编程序设计时用）

位、位串、字符和字符串

用来表示文本、声音和图像等

- 4 bits is a nibble（一个十六进制数字）
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word

逻辑(布尔)数据

按位操作（0-假 / 1-真）

操作数存放在哪里?

寄存器或内存单元中，也可以立即数的方式直接出现在指令中

IA-32 & RISC-V Data Type

- IA-32
  - 基本类型：
    - » 字节、字(16位)、双字(32位)、四字(64位)
  - 整数：
    - » 16位、32位、64位三种2-补码表示的整数
    - » 18位压缩8421 BCD码表示的十进制整数
  - 无符号整数（8、16或32位）
  - 近指针：32位段内偏移（有效地址）
  - 浮点数：IEEE 754（80位扩展精度浮点数寄存器）
- RISC-V
  - 基本类型：
    - » 字节、半字(16位)、字(32位)、双字(64位)
  - 整数： 16位、32位、64位三种2-补码表示的整数
  - 无符号整数： （16、32位）
  - 浮点数：IEEE 754（32位/64位浮点数寄存器）

Addressing Modes（寻址方式）

- 什么是“寻址方式”？

  指令或操作数地址的指定方式。即：根据地址找到指令或操作数的方法。

- 地址码编码由操作数的寻址方式决定
- 地址码编码原则：

  为什么?

  指令地址码尽量短 → 目标代码短，省空间

  操作数存放位置灵活，空间应尽量大 → 利于编译器优化产生高效代码

  地址计算过程尽量简单 → 指令执行快

- 指令的寻址----简单

  正常：PC增值

  跳转（jump / branch / call / return）：同操作数的寻址

- 操作数的寻址----复杂

  操作数来源：寄存器 / 外设端口 / 主(虚)存 / 栈顶

  操作数结构：位 / 字节 / 半字 / 字 / 双字 / 一维表 / 二维表 /...

通常寻址方式指“操作数的寻址方式”

Addressing Modes

- 寻址方式的确定

  （1）没有专门的寻址方式位（由操作码确定寻址方式）

  即：只要知道是什么指令，就知道去哪里找操作数。

  （2）有专门的寻址方式位

  即：指令中可以看出有多个操作数，但每个操作数去哪里找，还需要指令中再专门记录有它们的寻址方式位。

- 有效地址的含义

  操作数所在存储单元的地址

  可通过指令的寻址方式和地址码算出有效地址

操作数存放在哪里?

寄存器或内存单元中，也可以立即数的方式直接出现在指令中

存放在内存时才涉及到有效地址的计算

- 基本寻址方式

  立即 / 直接 / 间接 / 寄存器 / 寄存器间接 / 偏移 / 栈

- 基本寻址方式的算法及优缺点

基本寻址方式的算法和优缺点

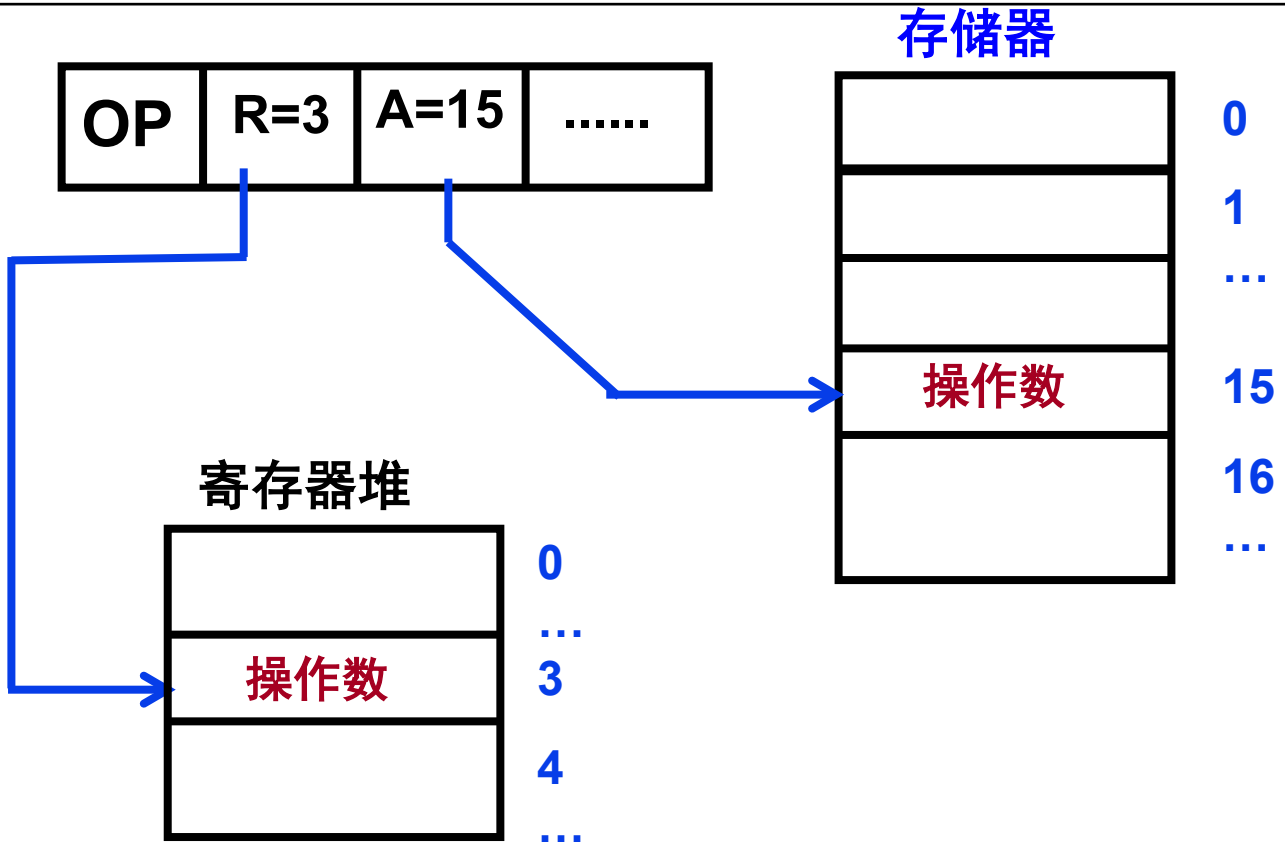
假设：A=地址字段值，R=寄存器编号，
EA=有效地址，(X)=X中的内容

方式	算法	主要优点	主要缺点	操作数位置
立即数	操作数=A	指令执行速度快	操作数幅值有限	指令(寄存器)中
直接	EA=A	有效地址计算简单	地址范围有限	内存中
间接	EA=(A)	有效地址范围大	多次存储器访问	内存中
寄存器直接	操作数=(R)	指令执行快，指令短	地址范围有限	寄存器中
寄存器间接	EA=(R)	地址范围大	额外存储器访问	内存中
偏移	EA=A+(R)	灵活	复杂	内存中
栈	EA=栈顶	指令短	应用有限	内存中

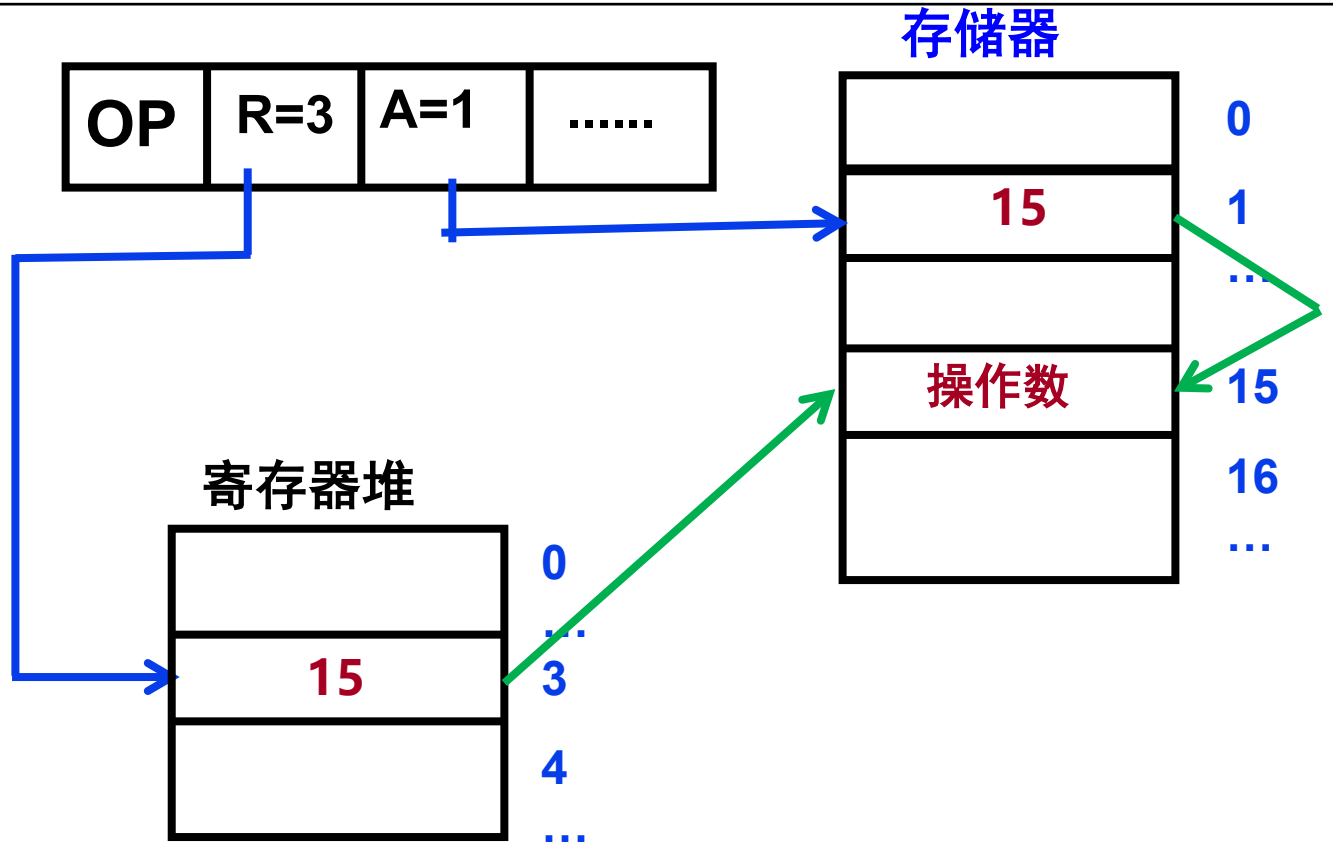
偏移方式：将直接方式和寄存器间接方式结合起来。
有：相对 / 基址 / 变址三种 （见后面几页！）

问题：以上各种寻址方式下，操作数在哪里?

寄存器直接寻址、直接寻址

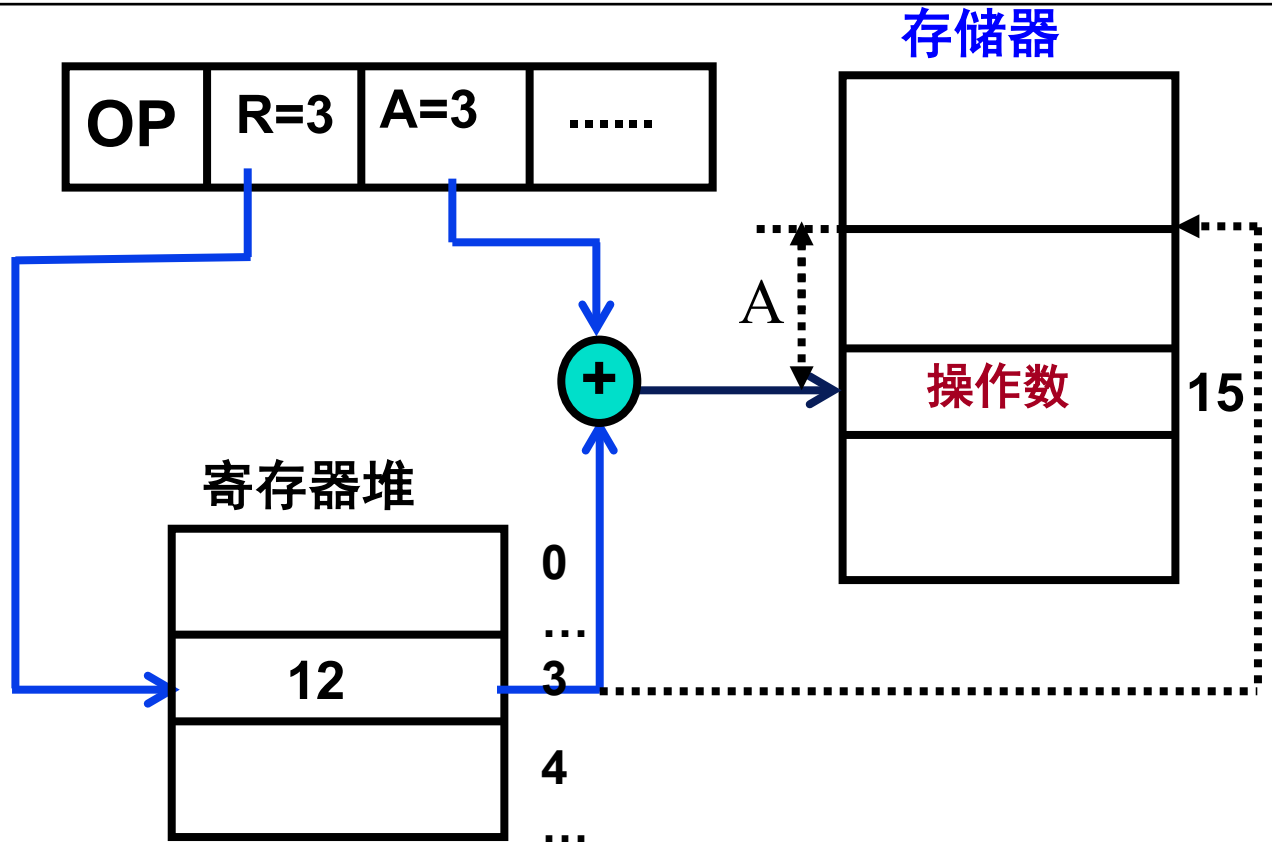


寄存器间接寻址、间接寻址



偏移寻址方式

指令中给出的地址码A称为形式地址



偏移寻址：EA=A+(R)

——R可以明显给出，也可以隐含给出

——R可以为PC、基址寄存器B、变址寄存器 I

偏移寻址方式

- **相对寻址**
- 指令地址码给出一个偏移量(带符号数)，基准地址R隐含由PC给出。
- 即：EA=(PC)+A——相对于当前指令处位移量为A的单元
- 可用来实现程序(公共子程序)的浮动 或 指定转移目标地址
- 注意：当前PC的值可以是正在执行指令的地址或下条指令的地址
- **基址寻址**
- 指令地址码给出一个偏移量，基准地址R明显或隐含由基址寄存器B给出
  即：EA=(B)+A—— 相对于基址(B)处位移量为A的单元
- 可用来实现多道程序重定位 或 过程调用中参数的访问
- **变址寻址**
- 指令地址码给出一个基准地址，而偏移量(无符号数)R明显或隐含由变址寄存器 I 给出。即：EA=(I)+A——相对于首址A处位移量为(I)的单元
- 可为循环重复操作提供一种高效机制，如实现对线性表的方便操作

变址寻址实现线性表元素的存取

- **自动变址**

  指令中的地址码A给定数组首址，变址器I每次自动加/减数组元素的长度x。

  EA=(I)+A

  I=(I) ± x

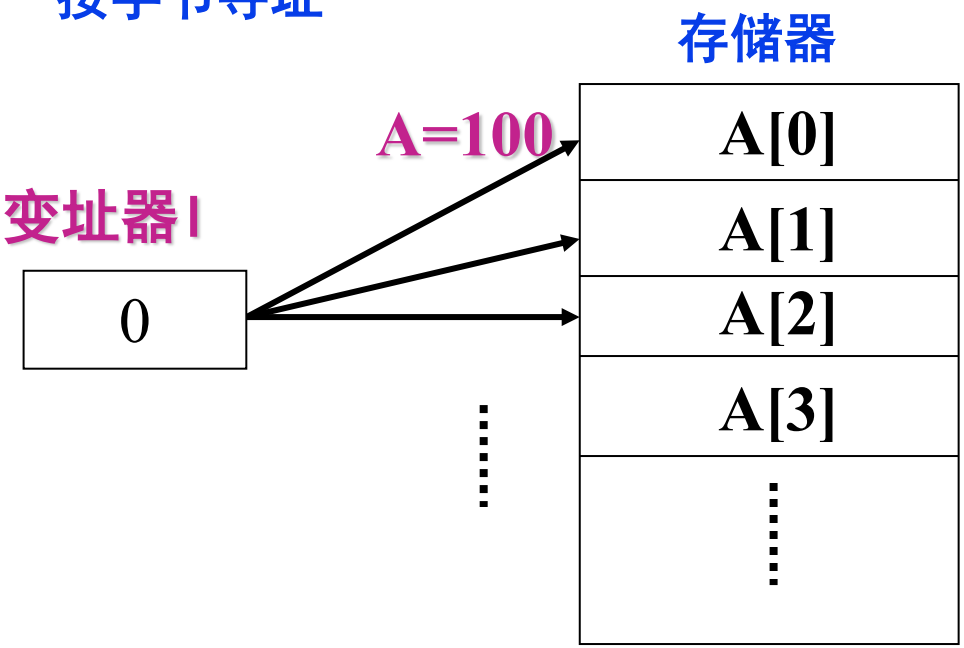
  例如，X86中的串操作指令

- **对于“for (i=0;i<N;i++) ....”，即地址从低→高增长：加**
- **对于“for (i=N-1;i>=0;i--) ....”,即地址从高→低增长：减**

- **可提供对线性表的方便访问**

假定一维数组A从内存100号单元开始

按字节寻址



若每个元素为一个字节，则 I=(I) ± 1

若每个元素为4个字节，则 I=(I) ± 4

Instruction Format(指令格式)

- 操作码的编码有两种方式
  - Fixed Length Opcodes (定长操作码法)
  - Expanding Opcodes (扩展操作码编法)
- instructions size
  - 代码长度更重要时：采用变长指令字、变长操作码
  - 性能更重要时：采用定长指令字、定长操作码

  为什么?

变长指令字和变长操作码使机器代码更紧凑；

定长指令字和定长操作码便于快速访问和译码。

定长操作码，也可以是变长指令字

但变长操作码，一般不会是定长指令字

回顾第16次课

ISA的要素

——指令集设计（考虑数据类型），寄存器设计

指令的要素

——操作码，地址码

编译→机器指令→CPU（取指令，译码，取数，运算，存数，下一条）

操作数的寻址方式

——立即寻址、寄存器直接（间接）寻址、直接（间接）寻址、偏移、栈

操作数可以存放在哪里：内存，寄存器，（立即数在指令中）

指令的寻址方式：

顺序（PC+“1”），跳转（相对寻址方式，(PC)+A）（其它）

定长操作码编码 Fixed Length Opcodes

基本思想

- 指令的操作码部分采用固定长度的编码
- 如：假设操作码固定为6位，则系统最多可表示64种指令

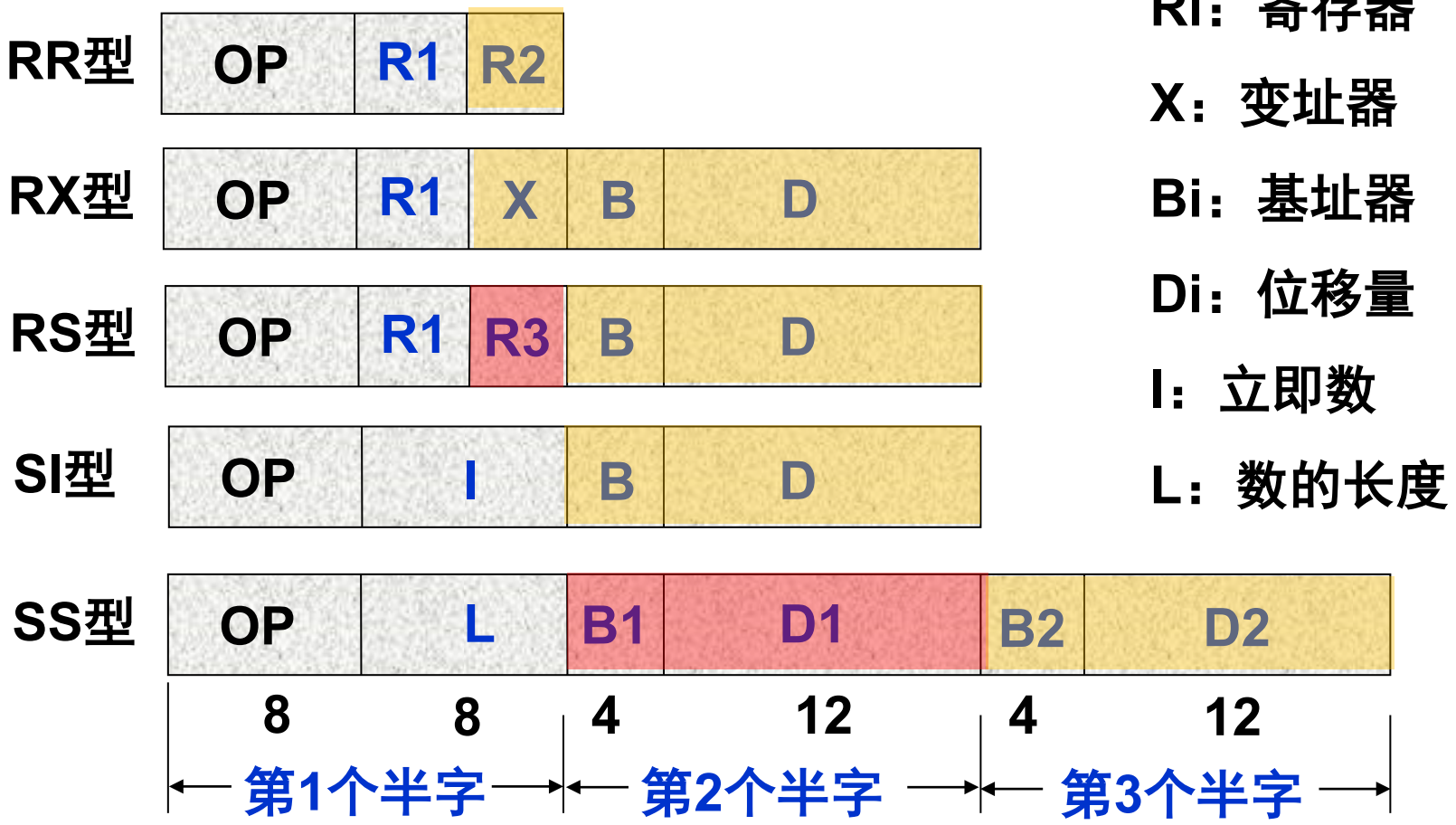
特点

- 译码方便，但有信息冗余

举例

- IBM360/370采用:
- 8位定长操作码，最多可有256条指令
- 只提供了183条指令，有73种编码为冗余信息
- 机器字长32位，按字节编址
- 有16个32位通用寄存器，基址器B和变址器X可用其中任意一个
- 问题：通用寄存器编号有几位？B和X的编号占几位？ 都是4位！

IBM370指令格式



Ri：寄存器

X：变址器

Bi：基址器

Di：位移量

I：立即数

L：数的长度

RR：寄存器 - 寄存器

RX：寄存器 - 变址存储器

RS：寄存器 – 基址存储器

SS：基址存储器 - 基址存储器

SI：基址存储器 - 立即数

格式：定长操作码、变长指令字

扩展（变长）操作码编码 Expanding Opcodes

基本思想

- 将操作码的编码长度分成几种固定长的格式。被很多指令集采用。
- PDP-11是典型的变长操作码机器。

种类

- 等长扩展法：4-8-12；3-6-9；..... / 不等长扩展法

举例说明如何扩展

- 设某指令系统指令字是16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条?
- 解:操作码按短到长进行扩展编码
- 二地址指令: (0000 ~ 1110)
- 一地址指令: 11110 (00000 ~ 11111); 11111 (00000 ~ 00001)
- 零地址指令: 11111 (00010 ~ 11111) (000000 ~ 111111)
- 故零地址指令最多有 $30x2^6=15x2^7$ 种

下一条指令去哪里找——顺序 or 条件测试方式

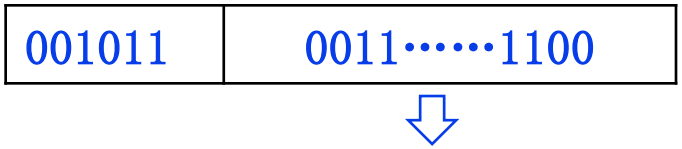
- 正常情况隐含在PC中——顺序执行
- 改变顺序时由指令给出

（1）指令中显式给出“下条指令地址”

（2）条件转移指令：通常根据Condition Codes (条件码 CC/ 状态位 / 标志位)转移： 执行算术指令或显式比较指令来设置CC

ex: sub r1, r2, r3 ;r2和r3相减, 结果在r1中，并生成标志位ZF、CF等
bz label ;标志位ZF=1时转到label处执行；否则顺序执行

示例：



label: 指令中的地址码
——可长可短
——据此计算目标指令地址
——可有多种计算方式

条件测试方式（续）

常用的标志（条件码）有四种（怎么生成？参考之前ppt）

SF – negative　OF – overflow　　CF – 进位/借位　ZF – zero

对于带符号和无符号整数加减运算，标志生成方式有没有不同？

没有，因为加法电路不知道是无符号整数还是带符号整数。

- **标志可存在：标志寄存器/条件码寄存器**
  /状态寄存器/程序状态字寄存器

也可由指定的通用寄存器来存放状态位

Ex:　cmp r1, r2, r3　;比较r2和r3, 标志位存储在r1中

bgt r1, label　;判断r1是否大于0，是则转移到label处

示例：

000011	00001	0011……1100

不同处理器对标志位的处理不同

IA-32中的条件转移指令

分三类：

(1)根据单个标志的值转移

(2)按无符号整数比较转移

(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A＞B
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 A≥B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A＜B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A≤B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A＞B
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 A≥B
15	jl/jnge label	SF≠OF AND ZF=	带符号整数 A＜B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A≤B

指令设计风格 -- 按操作数位置指定风格来分

Accumulator: (earliest machines) 累加器型

其中一个操作数和目的操作数总在累加器中

Stack: (e.g. HP calculator, Java virtual machines) 栈型

总是将栈顶两个操作数进行运算，指令无需指定操作数地址

General Purpose Register: (e.g. IA-32) 通用寄存器型

操作数可以是寄存器或存储器数据

Load/Store: (e.g. SPARC, MIPS, RISC-V) 装入/存储型

运算操作数只能是寄存器数据，只有load/store能访问存储器

比较

思考：指令长度？指令条数？指令执行效率？

。表达式 C = A + B 可以怎么实现：

Stack	Accumulator	Register (register- memory)	Register (load - store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

指令条数较少

复杂表达式时，累加器型风格指令条数变多，因为所有运算都要用累加器，使得程序中多出许多移入 / 移出累加器的指令！

75年开始，寄存器型占主导地位

- 寄存器速度快，使用大量通用寄存器可减少访存操作
- 表达式编译时与顺序无关（相对于Stack）

指令设计风格 – 按指令格式的复杂度来分

按指令格式的复杂度来分，有两种类型计算机：

复杂指令集计算机CISC (Complex Instruction Set Computer)

精简指令集计算机RISC (Reduce Instruction Set Computer)

早期CISC设计风格的主要特点

- (1) 指令系统复杂
- 变长操作码 / 变长指令字 / 指令多 / 寻址方式多 / 指令格式多
- (2) 指令周期长
- 绝大多数指令需要多个时钟周期才能完成
- (3) 各种指令都能访问存储器
- 除了专门的存储器读写指令外，运算指令也能访问存储器
- (4) 采用微程序控制
- (5) 有专用寄存器
- (6) 难以进行编译优化来生成高效目标代码

例如，VAX-11/780小型机

16种寻址方式；9种数据格式；303条指令；

一条指令包括1～2个字节的操作码和下续N个操作数说明符。

一个说明符的长度达1 ～10个字节。

复杂指令集计算机CISC

- CISC的缺陷
  - 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。
- 1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC (Reduce Instruction Set Computer)。
- 对CISC进行测试，发现一个事实：
  - 在程序中各种指令出现的频率悬殊很大，最常使用的是一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%。
- 1982年美国加州伯克利大学的RISC-I，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

SKIP

Top 10 80x86 Instructions

° Rank instruction Integer Average Percent total executed

1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency

（简单指令占主要部分，使用频率高！）

BACK

RISC设计风格的主要特点

- (1) 简化的指令系统
- 指令少 / 寻址方式少 / 指令格
- (2) 以RR方式工作
- 除Load/Store指令可访问存储器外，其余指令都只访问寄存器。
- (3) 指令周期短
- 以流水线方式工作， 因而除Load/Store指令外，其他简单指令都只需一个或一个不到的时钟周期就可完成。
- (4) 采用大量通用寄存器，以减少访存次数
- (5) 采用组合逻辑电路控制，不用或少用微程序控制
- (6) 采用优化的编译系统，力求有效地支持高级语言程序

Load/store型机器指令一般都是隐含寻址方式（不需要专门的寻址方式位）

一般RISC机器不提供自动变址寻址，并将变址和基址寻址统一成一种偏移寻址方式

MIPS、RISC(RISC-I到RISC-V）系列架构是典型的RISC处理器，82年以来新的指令集大多采用RISC体系结构

x86因为“兼容”的需要，保留了CISC的风格，同时也借鉴了RISC思想

Examples of Register Usage

每条典型ALU指令中的存储器地址个数

每条典型ALU指令中的最多操作数个数

		Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC， RISC-V
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

In VAX(CISC): ADDL (R9), (R10), (R11) 一条指令！

; mem[R9] ←mem[R10] + mem[R11]

In MIPS(RISC):

```
lw   R1, (R10)     :  R1 ← mem[R10]
lw   R2, (R11)     :  R2 ← mem[R11]
add  R3, R1, R2    :  R3 ← R1+R2
sw   R3, (R9)      :  mem[R9] ← R3
```

四条指令！

哪一种风格更好呢？学了CPU设计之后会有更深的体会！

异常和中断

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被 “中断” 的事件有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件

    按发生原因分为硬故障中断和程序性中断两类

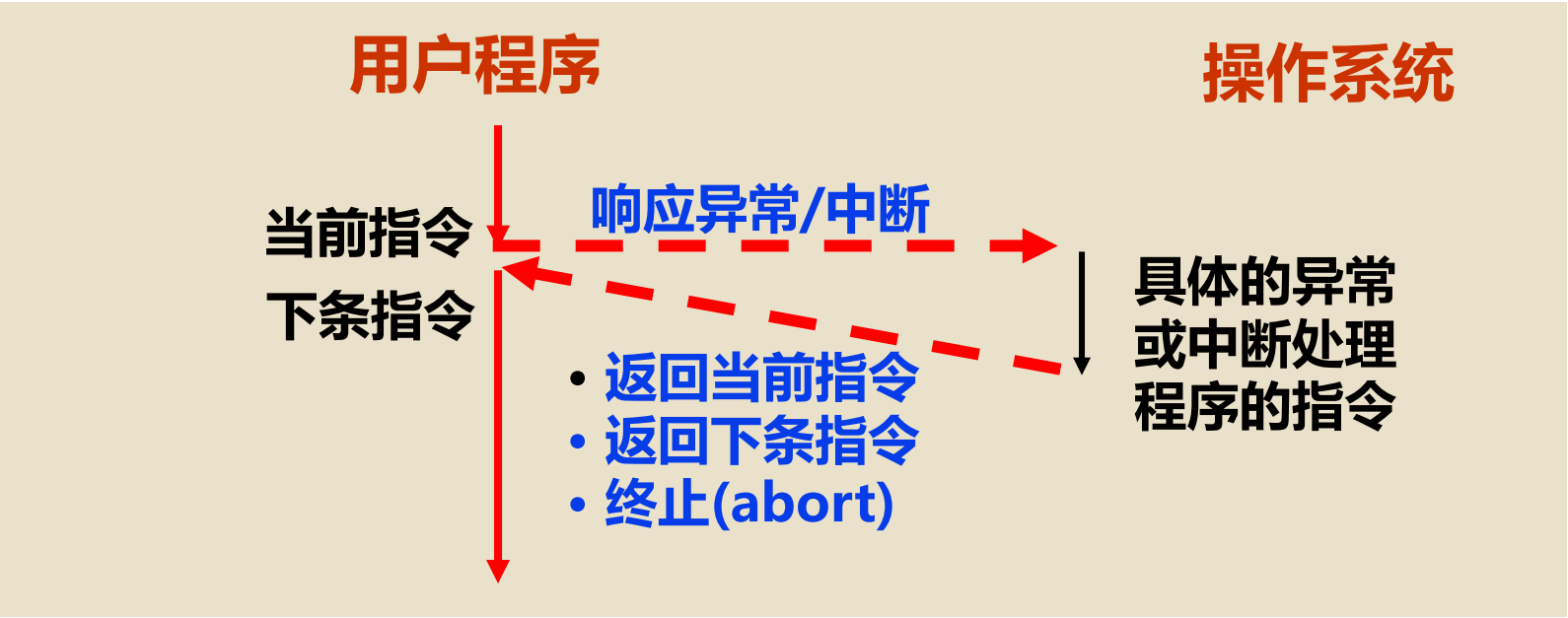
    硬故障中断：如电源掉电、硬件线路故障等

    程序性中断：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。

异常和中断的处理

◆ 发生异常(exception)和中断(interrupt)事件后——

(不包括硬故障)



指令系统举例: Address & Registers

Intel 8086	2^{20} x 8 bit bytes AX, BX, CX, DX SP, BP, SI, DI CS, SS, DS IP, Flags	acc, index, count, quot stack, stack frame, string code,stack,data segment
VAX 11	2^{32} x 8 bit bytes 16 x 32 bit GPRs	r15-- program counter r14-- stack pointer r13-- frame pointer r12-- argument pointer
MC 68000	2^{24} x 8 bit bytes 8 x 32 bit GPRs 7 x 32 bit addr reg 1 x 32 bit SP 1 x 32 bit PC	Flags：状态标志寄存器 GPR：通用寄存器堆
MIPS32	2^{32} x 8 bit bytes 32 x 32 bit GPRs 32 x 32 bit FPRs HI, LO, PC	HI和LO是MIPS内部的乘商寄存器

指令系统举例：MIPS32的指令格式

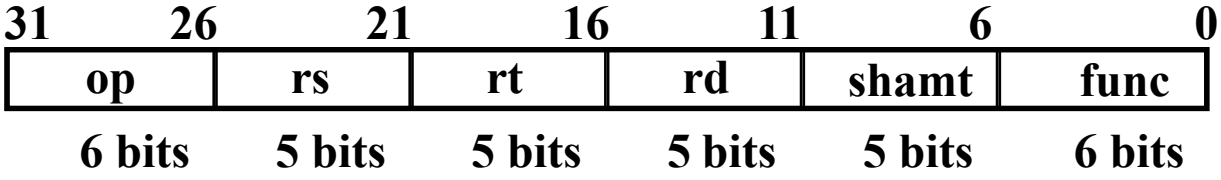
ld/st型、RISC型

- 32个通用寄存器，所有指令都是32位宽，须按字地址对齐

  字地址为4的倍数！

- 有三种指令格式

R-Type指令



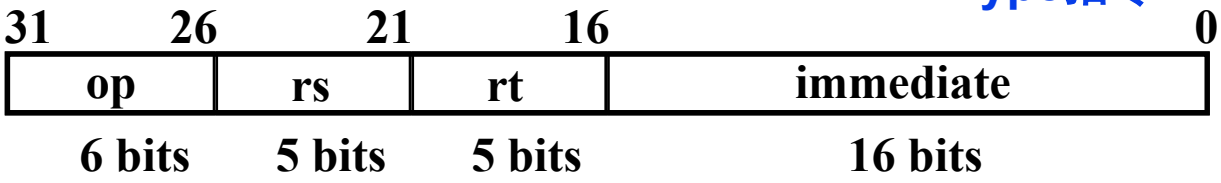
– R-Type

两个操作数和结果都在寄存器的运算指令。如：sub rd, rs, rt

– I-Type

- 运算指令：一个寄存器、一个立即数。如：ori rt, rs, imm16
- LOAD和STORE指令。如：lw rt, rs, imm16
- 条件分支指令。如：beq rs, rt, imm16

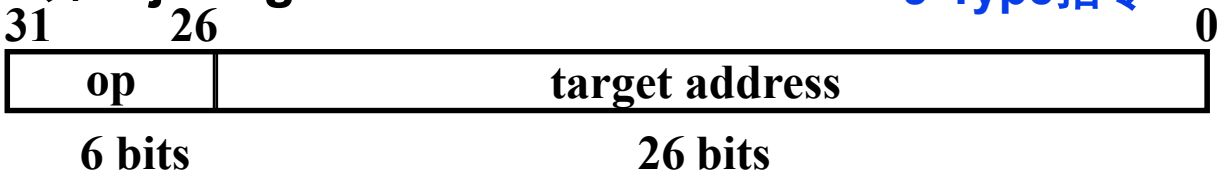
I-Type指令



– J-Type

无条件跳转指令。如：j target

J-Type指令



指令系统举例：IA-32的指令格式

前缀：包括指令、段、操作数长度、地址长度四种类型

前缀类型：	指令前缀	段前缀	操作数长度	地址长度
字节数：	0或1	0或1	0或1	0或1

指令：含操作码、寻址方式、SIB、位移量和直接数据五部分，位移量和立即数都可是1/2/4B。SIB中基址B和变址I都可是8个GRS中任一个。SS给出比例因子。操作码：opcode; w：与机器模式（16 / 32位）一起确定寄存器位数（AL / AX / EAX）；d：操作方向；寻址方式： mod、r/m、 reg/op三个字段与w字段和机器模式一起确定操作数所在的寄存器编号或有效地址计算方式

指令段：	操作码	寻址方式	SIB	位移	直接数据
字节数：	1或2	0或1	0或1	1、2、4	立即数

变长指令字：1B~17B
变长操作码：4b / 5b / 6b / 7b / 8b /……
变长操作数：Byte / Word / DW / QW
变长寄存器：8位 / 16位 /32位

ALU指令中的一个操作数可来自存储器

通用寄存器型、CISC型

第一讲小结（重要）

- 一个计算机系统中需要定义多条指令
- 指令的功能/含义由操作码(或加上某些功能字段)来决定
- 每条指令中的二进制位具体怎么使用呢？
  - 操作码+地址码+可能需要的其它附加字段
  - 在一条指令中可以有0-N个地址码
  - 指令的设计离不开寄存器的设计（数量、功能等必须预先确定）
  - 一个机器中所有指令的长度可相同，也可各不相同
- 需要多少种操作码呢？
  - 由所需的功能（运算，控制等）来决定
- 需要处理哪些数据类型呢？
  - 也由所需运算类型来决定
- 如何完成指令中对操作数的存取要求呢（核心功能）？
  - 寻址方式可以有多种，灵活使用
- 如何控制“周而复始的执行指令”呢？
  - 隐式的自动按顺序取
  - 显式的在指令中给出“下条指令地址”
  - 条件测试后计算出“转移目标地址”

第一讲小结

- 操作类型
  - 传送 / 算术 / 逻辑 / 移位 / 字符串 / 转移控制 / 调用 / 中断 / 信号同步
- 操作数类型
  - 整数（带符号、无符号、十进制）、浮点数、位、位串
- 地址码的编码要考虑：
  - 操作数的个数
  - 寻址方式：立即 / 寄存器 / 寄间 / 直接 / 间接 / 相对 / 基址 / 变址 / 堆栈
- 操作码的编码要考虑：
  - 定长操作码 / 扩展操作码
- 条件码的生成
  - 四种基本标志：NF（SF） / VF（OF） / CF / ZF
- 指令设计风格：
  - 按操作数地址指定方式来分：
    - » 累加器型 、通用寄存器型、load/store型、栈型
  - 按指令格式的复杂度来分
    - » 复杂指令集计算机CISC、精简指令集计算机RISC
- 典型指令系统举例
  - 以下将详细介绍RISC-V指令系统