回顾第16次课和第六章

定点数运算：由ALU + 移位器实现各种定点运算

- 移位运算：逻辑移位、算术移位、循环移位
- 扩展运算：零扩展、符号扩展
- 加减运算：补码加/减运算、原码加/减运算
- 乘法运算：用加法和右移实现：补码乘法、原码乘法
- 除法运算：用加/减法和左移实现：补码除法、原码除法

浮点数运算：由多个ALU + 移位器实现

定点数和浮点数计算的溢出（异常）判断

最终目标：定浮点运算部件、运算器、数据通路

第7章 指令系统

第1讲 概述与指令系统设计

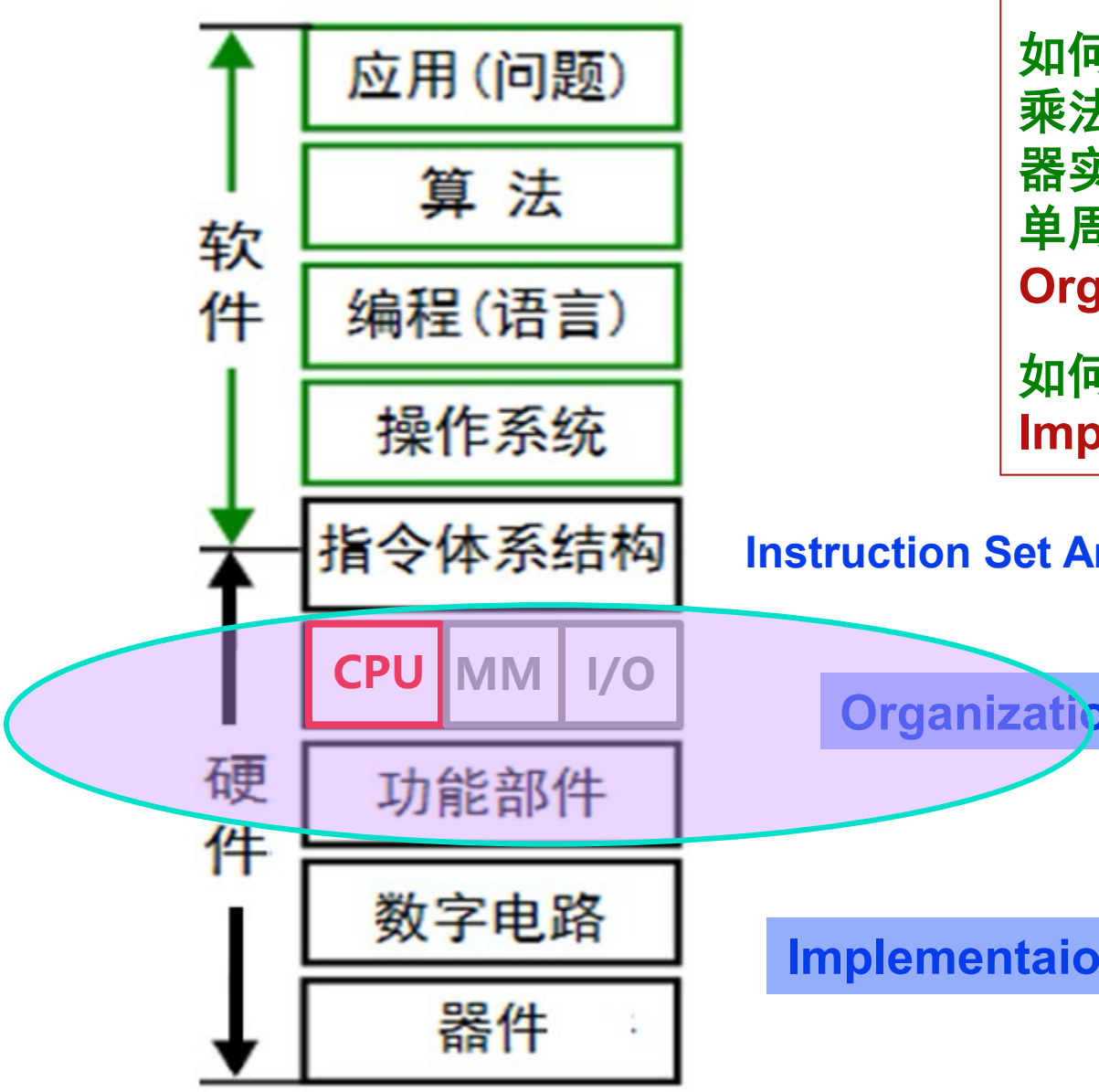
第2讲 指令系统实例：RISC-V架构

计算机组成，ISA

是否提供“乘法指令”：ISA

如何实现乘法指令（用快速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

回顾：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

指令集设计

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

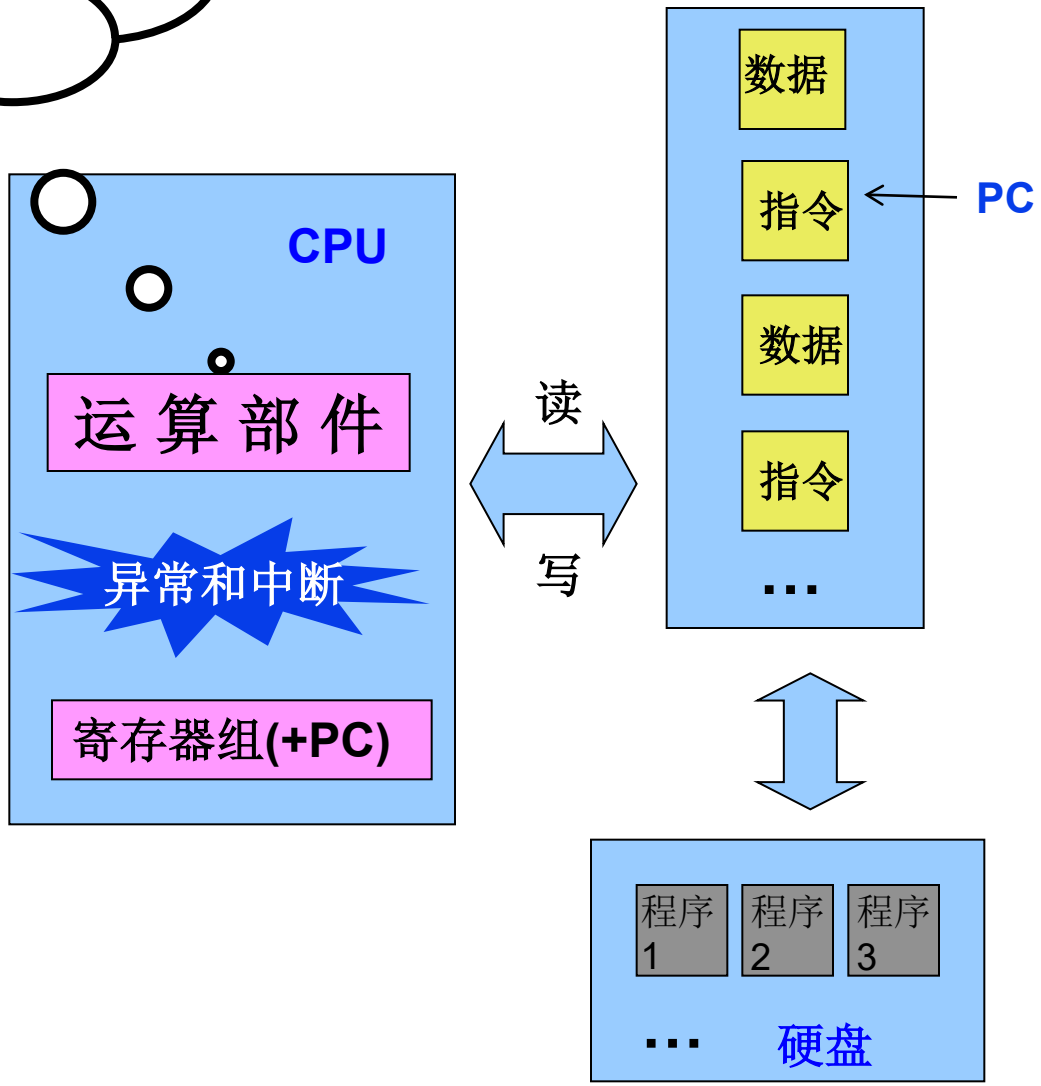
冯.诺依曼结构机器对指令规定：

- **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

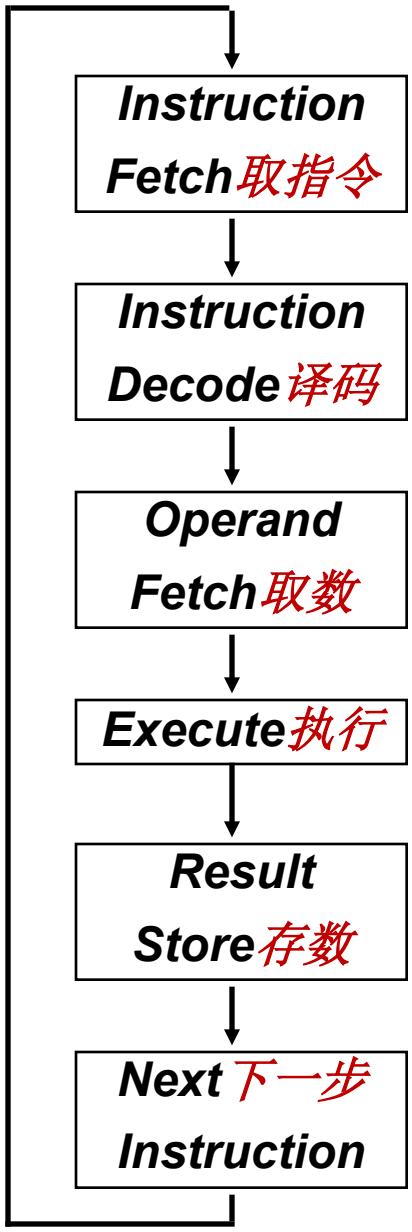
CPU负责周而复始的执行指令

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？

必需 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 详细！

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：

OP

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：

OP	A1

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：

OP	A1	A2

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：

OP	A1	A2	A3

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

  无条件 有条件(分支) 调用 返回

- 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

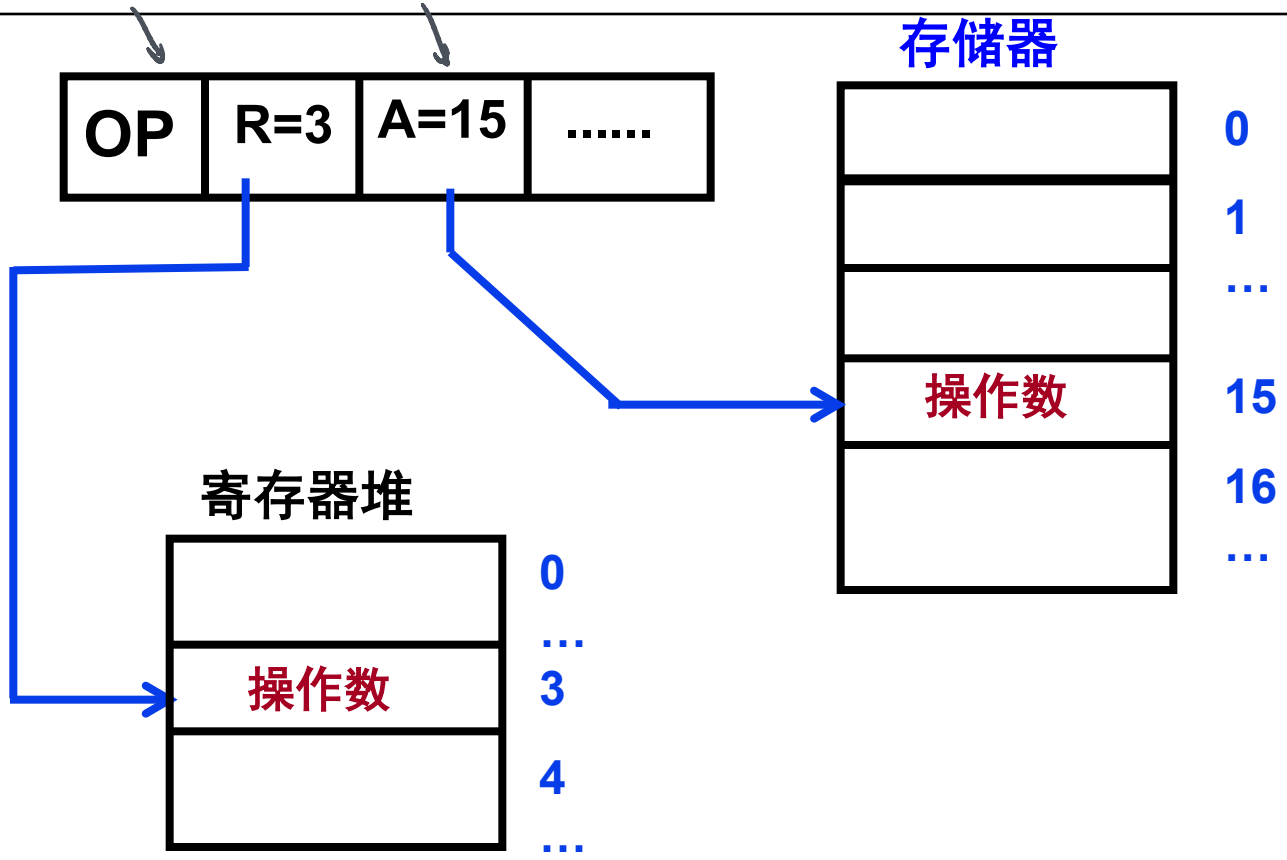
方式	算法	主要优点	主要缺点	操作数位置
立即数	操作数=A	指令执行速度快	操作数幅值有限	指令(寄存器)中
直接	EA=A	有效地址计算简单	地址范围有限	内存中
间接	EA=(A)	有效地址范围大	多次存储器访问	内存中
寄存器直接	操作数=(R)	指令执行快，指令短	地址范围有限	寄存器中
寄存器间接	EA=(R)	地址范围大	额外存储器访问	内存中
△ 偏移（直接+寄存器间接）	EA=A+(R)	灵活	复杂	内存中
(*) 栈	EA=栈顶	指令短	应用有限	内存中

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！）

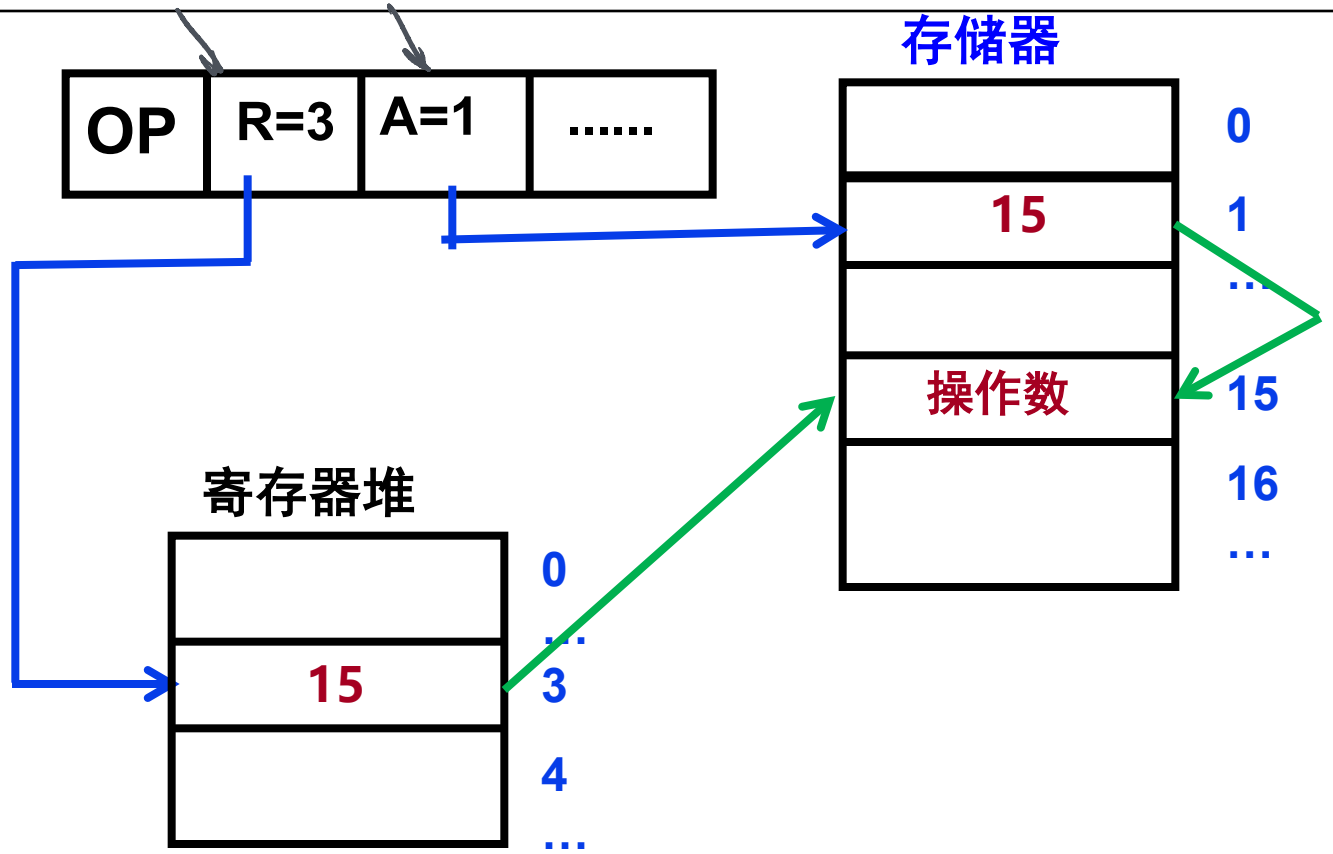
问题：以上各种寻址方式下，操作数在哪里？

EA→内存
操作数→寄存器

寄存器直接寻址、直接寻址

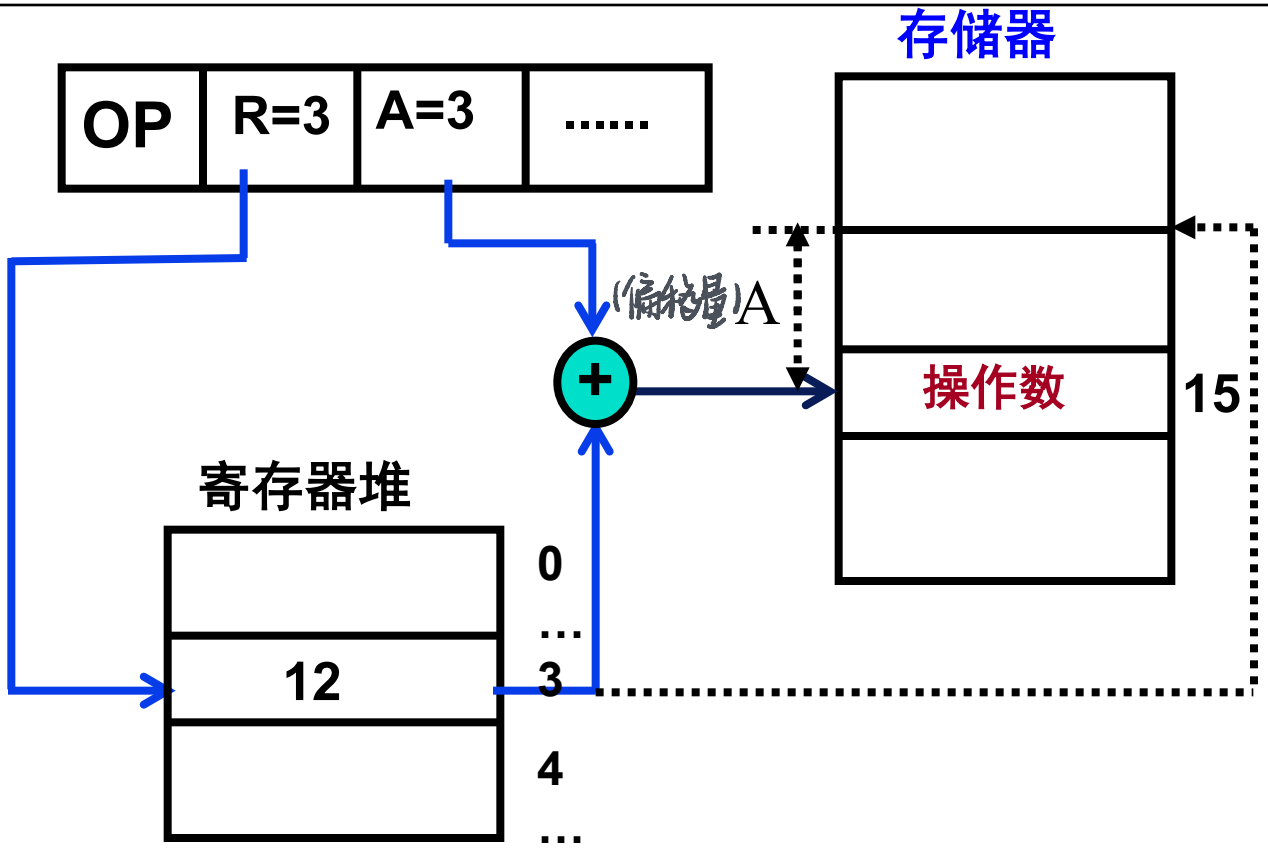


寄存器间接寻址、间接寻址



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**：数组

  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

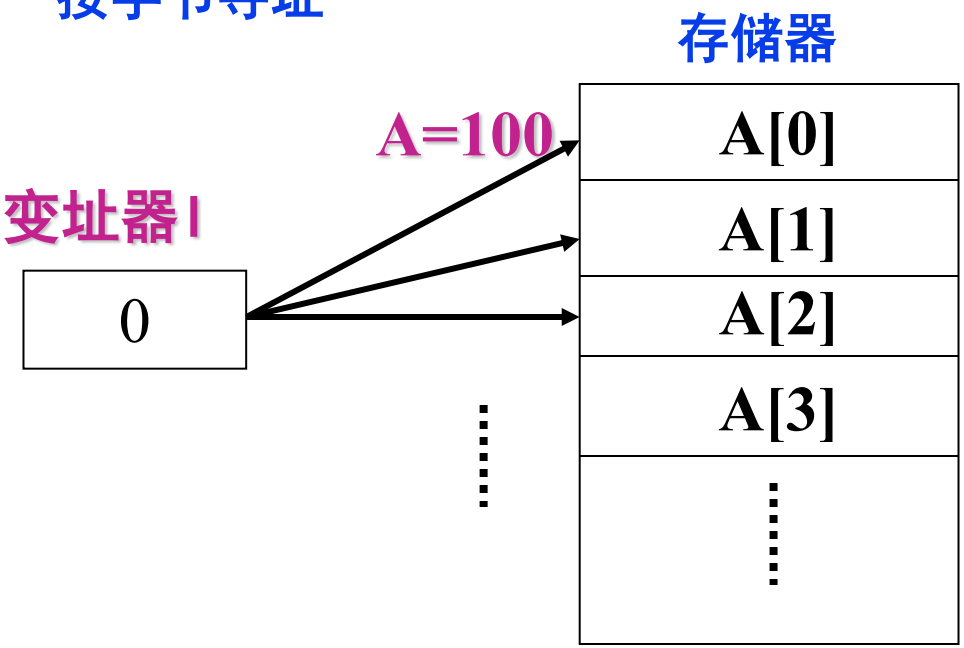
  例如，X86中的串操作指令

- **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

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

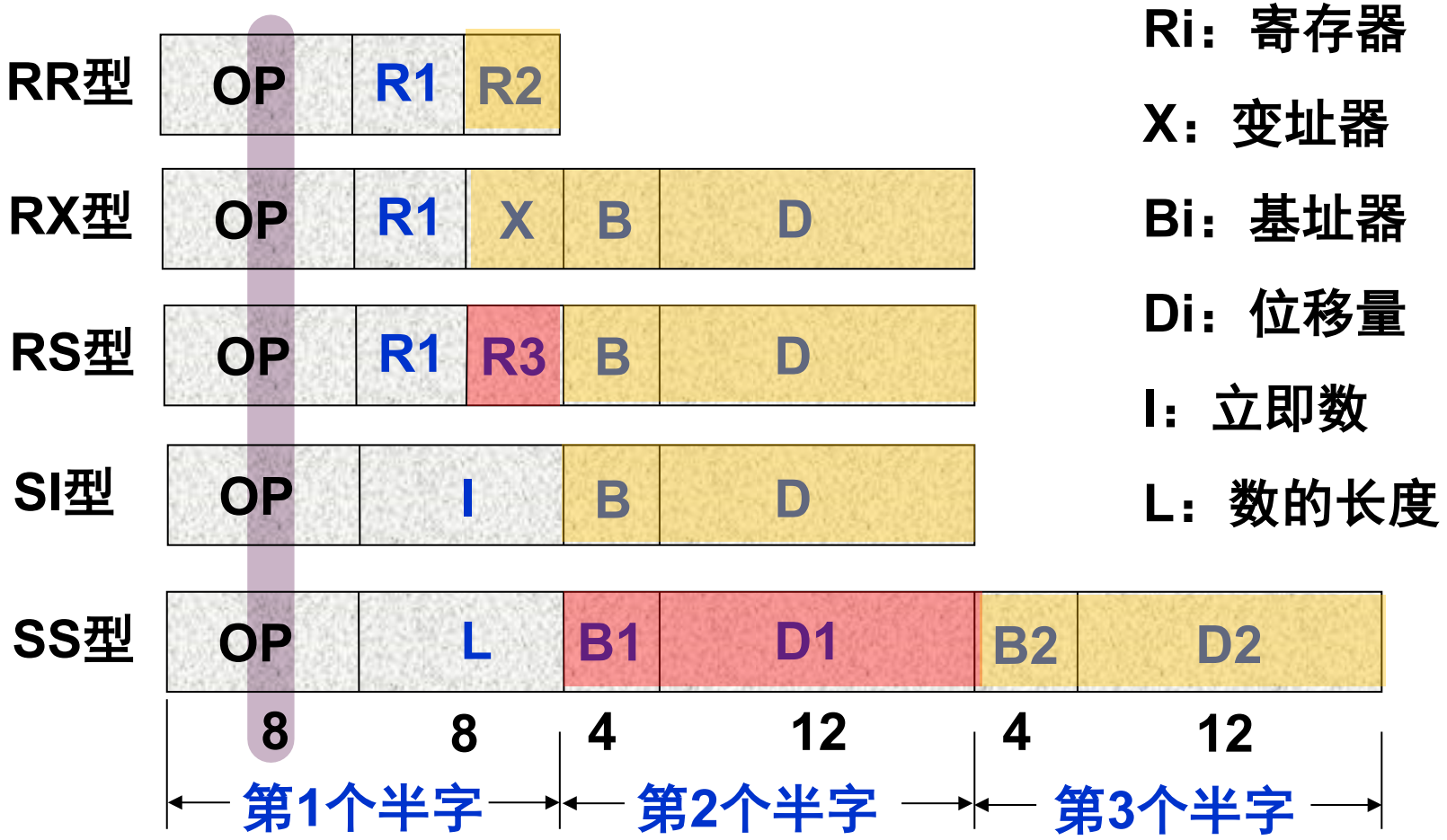
特点

- 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) 四位
- 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”无条件转移
  - （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：

001011	0011……1100

⇩

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**

  专用寄存器 or　**/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 / 指令格式
- (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

格式少，指令执行一致

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

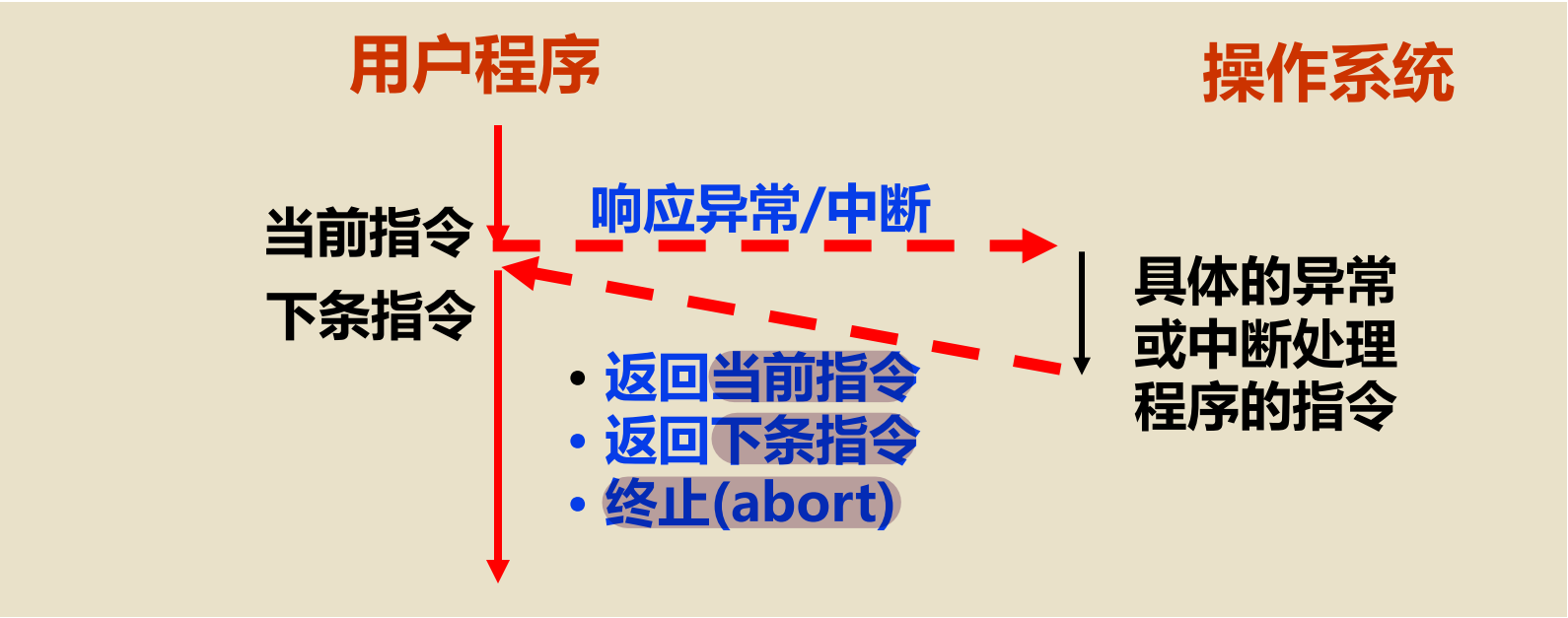
    硬故障中断：如电源掉电、硬件线路故障等

    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

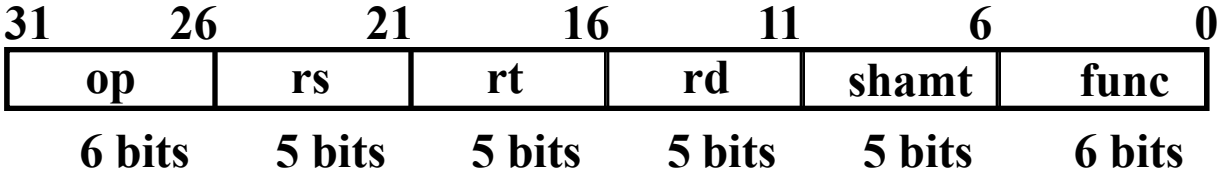
ld/st型、RISC型

- 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



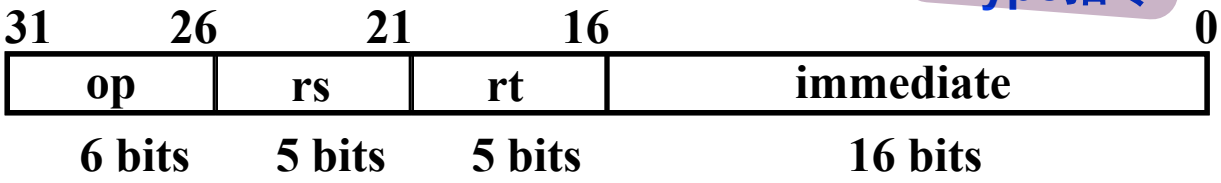
– R-Type

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

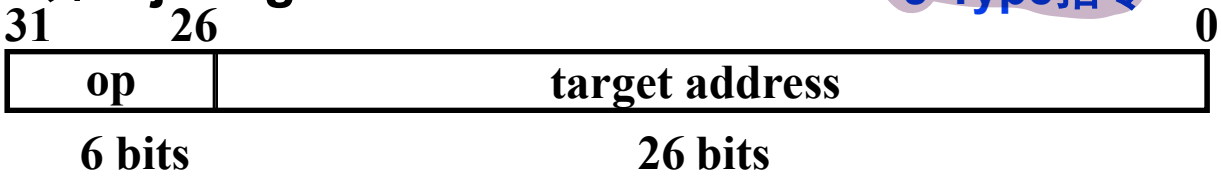
I-Type指令



– 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

Mod	Reg/OP	R/M
7 6	5 4 3	2 1 0

SS	Index	Base
7 6	5 4 3	2 1 0

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

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

第二讲指令系统实例：RISC-V

主要内容

- RISC-V指令系统概述
- RISC-V指令参考卡和指令格式
- RISC-V基础整数指令集
  - 整数运算
  - 控制转移
  - 存储访问
  - 系统控制
- RISC-V可选的扩展指令集

RISC-V指令系统概述

- 设计目标
  - 广泛的适应性：从最袖珍的嵌入式微控制器，到最快的高性能计算机
  - 支持各种异构处理架构，成为定制加速器的基础
  - 稳定的基础指令集架构，并能灵活扩展，且扩展时不影响基础部分
- 开源理念和设计原则
  - 本着“指令集应自由（Instruction Set Want to be Free）”的理念，指令集完全公开，且无需为指令集付费
  - 由一个非盈利性质的基金会管理，以保持指令集稳定，加快生态建设
  - 2020年基金会总部从美国迁到中立国瑞士，坚持开放自由、坚持为全世界服务的理念，被卡脖子的情况大大减少
  - 与以前的增量ISA不同，遵循“大道至简”的设计哲学，采用模块化设计，既保持基础指令集的稳定，也保证扩展指令集的灵活配置
  - 特点：具有模块化结构，稳定性和可扩展性好，在简洁性、实现成本、功耗、性能和程序代码量等各方面具有显著优势

RISC-V的模块化结构

- 核心：RV32I 32位/64位
- 标准扩展集：RV32M、RV32F、RV32D、RV32A

- 32位架构RV32G = RV32IMAFD
  - » 其压缩指令集RV32C（指令长度16位）
- 64位架构RV64G = RV64IMAFD
  - » 其压缩指令集RV64C（指令长度16位）

指令长度
机器字长
通用寄存器长度
定点运算器
处理数据的长度

同一条指令在RV64I和RV32I中都存在时，其具体行为也是不一样的

- 向量计算RV32V和RV64V；
- 嵌入式RV32E（RV32I的子集，16个通用寄存器）

指令参考卡①

- **核心指令集**：基础整数指令集 **RV32I 和 RV64I**
- **特权指令**：（参考教材pp196）
- **伪指令举例**-增加汇编程序可读性
- **压缩指令集**：**RV32C和RV64C**

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2	SLLW rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	SLLIW rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2	SRLW rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	SRLIW rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRAW rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	SRAIW rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADDW rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDIW rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUBW rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	B	BEQ rs1,rs2,imm	
	Branch ≠	B	BNE rs1,rs2,imm	
	Branch <	B	BLT rs1,rs2,imm	
	Branch ≥	B	BGE rs1,rs2,imm	
	Branch < Unsigned	B	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm	
Jump & Link	J&L	J	JAL rd,imm	
	Jump & Link Register	I	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
Environment	CALL	I	ECALL	
	BREAK	I	EBREAK	

Control Status Register (CSR)				
	Read/Write	I	CSRRW	rd,csr,rs1
	Read & Set Bit	I	CSRRS	rd,csr,rs1
	Read & Clear Bit	I	CSRRC	rd,csr,rs1
	Read/Write Imm	I	CSRRWI	rd,csr,imm
	Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
	Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm

+RV64I		
LWU	rd,rs1,imm	
LD	rd,rs1,imm	
SD	rs1,rs2,imm	

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	B	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8
	Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Float Store Word	CS	C.FSW	rs1',rs2',imm	FSW	rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8
	Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

Optional Compressed Extention: RV64C

All RV32C (except C.JAL, 4 word loads, 4 word strores) plus:

- ADD Word (C.ADDW)
- ADD Imm. Word (C.ADDIW)
- SUBtract Word (C.SUBW)
- Load Doubleword (C.LD)
- Load Doubleword SP (C.LDSP)
- Store Doubleword (C.SD)
- Store Doubleword SP (C.SDSP)

指令参考卡②

- 扩展指令集

乘除运算指令集RVM、原子操作指令集RVA、浮点运算指令集RVF和RVD、向量操作指令集RVV

- 通用寄存器的调用约定

32个定点通用寄存器x0~x31；32个浮点寄存器f0~f31；

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-0,ft0-7	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

Optional Vector Extension: RVV

Name	Fmt	RV32V/R64V	
SET Vector Len.	R	SETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,rs3
Multiply-SUB	R	VFMSUB	rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2
MERGE	R	VMERGE	rd,rs1,rs2
SELECT	R	VSELECT	rd,rs1,rs2

七专用

定点通用寄存器的功能定义和两种汇编表示

寄存器	ABI 名	功能描述	被调用过程保存?
x0	zero	硬编码 0	—
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	—
x4	tp	线程指针	—
x5	t0	临时寄存器	否
x6~x7	t1~t2	临时寄存器	否
x8	s0/fp	保存寄存器/帧指针	是
x9	s1	保存寄存器	是
x10~x11	a0~a1	过程参数/返回值	否
x12~x17	a2~a7	过程参数	否
x18~x27	s2~s11	保存寄存器	是
x28~x31	t3~t6	临时寄存器	否

函数调用-参数传递

指令长度为32位的RISC-V指令格式

- 共有6种指令格式
  - R-型为寄存器操作数指令
  - I-型为短立即数或装入（Load）指令
  - S-型为存储（Store）指令
  - B-型为条件跳转指令
  - U-型为长立即数操作指令
  - J-型为无条件跳转指令

- opcode：7位操作码字段
- rd、rs1和rs2：通用寄存器编号
- imm：立即数，其位数在括号[ ]中表示
- funct3和funct7：分别表示3位功能码和7位功能码，和opcode字段一起定义指令的操作功能

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

(*) 指令长度为16位的RISC-V压缩指令格式

- 共有8种指令格式。与32位指令相比，16位指令中的一部分寄存器编号还是占5位。指令变短了，但还是32位架构，处理的还是32位数据，还是有32个通用寄存器。
- 为了缩短指令长度，操作码op、功能码funct、立即数imm和另一部分寄存器编号的位数都减少了。(bit数减少)
- 每条16位指令都有功能完全相同的32位指令，在执行时由硬件先转换为32位指令再执行。目的是：缩短程序代码量，用少量时间换空间!

	15 14 13	12	11 10 9	8 7	6 5	4 3 2	1 0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm				rd′	op
CL	funct3	imm	rs1′		imm	rd′	op
CS	funct3	imm	rs1′		imm	rs2′	op
CB	funct3	offset	rs1′		offset		op
CJ	funct3	jump target					op

RISC-V基础整数指令集（RV32I）

- 包含：
  - 移位（Shifts）
  - 算术运算（Arithmetic）
  - 逻辑运算（Logical）
  - 比较（Compare）

  } 整数运算类指令

  - 分支（Branch）
  - 跳转链接（Jump & Link）

  } 控制转移类指令

  - 同步（Synch）
  - 环境（Environment）
  - 控制状态寄存器（Control Status Register）

  } 系统控制类指令

  - 取数（Load）
  - 存数（Store）

  } 存储访问类指令

Register Transfer Language

RTL规定：

R[r]：通用寄存器r的内容

M[addr]：存储单元addr的内容

M[R[r]]：寄存器r的内容所指存储单元的内容 间接寻址

PC：PC的内容

M[PC]：PC所指存储单元的内容 指令

SEXT[imm]：对imm进行符号扩展

ZEXT[imm]：对imm进行零扩展

传送方向用←表示，即传送源在右，传送目的在左

RISC-V基础整数指令集（RV32I）

整数运算类指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli 逻左
0000000	shamt	rs1	101	rd	0010011	I srli 逻右
0100000	shamt	rs1	101	rd	0010011	I srai 算右
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

RISC-V基础整数指令集（RV32I）

U型指令共2条

imm[31:12]	rd	0110111	U lui
imm[31:12]	rd	0010111	U auipc

lui rd, imm20：将立即数imm20存到rd寄存器高20位，低12位为0。该指令和“addi rd, rs1, imm12”结合，可以实现对一个32位变量赋初值。

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

举例：请给出C语句“int x=-8191;”对应的RISC-V机器级代码（就是编译）

解：对应的RISC-V机器指令和汇编指令为：

寻址方式？

#R[x5]←FFFFE000H (-8192)

1111 1111 1111 1111 1110 00101 0110111 lui x5, 1048574 #R[x5]←

0000 0000 0001 00101 000 00101 0010011 addi x5, x5, 1 #R[x5]←R

#R[x5]←R[x5]+SEXT[0001H]

立即
寄存器直接

SEXT表示符号扩展

-8191的机器数为：1111 1111 1111 1111 1110 0000 0000 0001

auipc rd, imm20：将立即数imm20加到PC（32位）的高20位上，结果存rd

可用指令“auipc x10, 0”获取当前PC的内容，存入寄存器x10中。

RISC-V基础整数指令集（RV32I）

I 型指令共9条，其中三条为用立即数指定所移位数的移位指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai

操作码opcode：都是0010011，其功能由funct3指定，而当funct3=101时，再有高7位区分是算术右移（srai）还是逻辑右移（srli）。

imm[11:0]：12位立即数，**符号扩展为32位**，作为第2个源操作数，和R[rs1]（寄存器rs1中的内容）进行运算，结果存rd。

shamt：指出移位位数，因为最多移31位，故用5位即可。**无算术左移指令**

RISC-V基础整数指令集（RV32I）

举例：请给出C语句“int x=8191;”对应的RISC-V机器级代码。

解：8191的机器数为：0000 0000 0000 0000 0001 1111 1111 1111

“lui rd, imm20” 和 “addi rd, rs1, imm12”结合。如下，对不对?

```
0000 0000 0000 0000 0001 00101 0110111  lui x5, 1 #R[x5]← 0000 1000H   (4096)
1111 1111 1111 00101 000 00101 0010011  addi x5, x5,-1#R[x5]←R[x5]+SEXT[FFFH]
```

不对！因为低12位中第一位为1，addi按符号扩展相加！结果为4095。

【注意：机器指令 转换成 汇编形式时，都是按该指令的设计规定来完成的】

可利用addi符号扩展特性进行调整！因为 imm12范围为-2048~2047，故可用lui先装入一个距离目标常数小于2048的数，再通过 addi 进行 加 或 减（imm12为负时）来调整！

这里 8191=8192-1，故可先装入8192，再用 addi 减1 （加全1）！

```
0000 0000 0000 0000 0010 00101 0110111  lui x5, 2  #R[x5]← 0000 2000H   (8192)
1111 1111 1111 00101 000 00101 0010011  addi x5, x5,-1 #R[x5]←R[x5]+SEXT[FFFH]
```

RISC-V基础整数指令集（RV32I）

R型指令共10条

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

操作码opcode：都是0110011，其功能由funct3指定，而当funct3=000、101时，再由funct7区分是加（add）还是减（sub）、逻辑右移（srl）还是算术右移（sra）。 移位位数放在rs2中.

rs1、rs2、rd：5位通用寄存编号，共32个；两个源操作数分别在rs1和rs2寄存器中，结果存rd。

sll：逻辑左移指令，无算术左移指令。因逻辑左移和算术左移结果完全相同

RISC-V基础整数指令集（RV32I）

4条比较指令：带符号小于（slt、slti）、无符号小于（sltu、sltiu）

例如，“sltiu rd, rs1, imm12”功能为：将rs1内容与imm12符号扩展结果按无符号整数比较，若小于，则1存入rd中；否则，0存入rd中。

imm[11:0]	rs1	010	rd	0010011	I slti
imm[11:0]	rs1	011	rd	0010011	I sltiu

0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu

RISC-V基础整数指令集（RV32I）

举例：假定变量x、y和z都是long long型，占64位，
x的高、低32位分别存放在寄存器x13、x12中；
y的高、低32位分别存放在寄存器x15、x14中；
z的高、低32位分别存放在寄存器x11、x10中

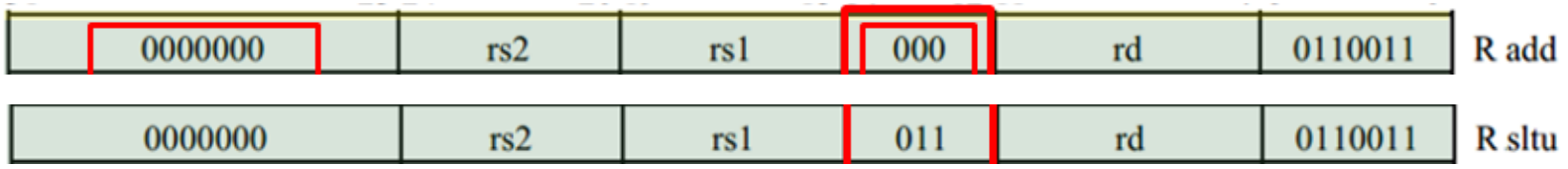
请写出C语句“z=x+y;”对应的32位字长RISC-V机器级代码。

解：可通过sltu指令将低32位的进位加入到高32位中。

低32位不涉及符号位，所以使用sltu。用add和addu都可以。

```
0000000 01110 01100 000 01010 0110011 add x10,x12,x14 #R[x10]←R[x12]+R[x14]
0000000 01100 01010 011 01011 0110011 sltu x11,x10,x12 #若R[x10]<R[x12]，则
                                      #  R[x11]←1（若和比加数小，则一定有进位）
0000000 01111 01101 000 10000 0110011 add x16,x13,x15 #R[x16]←R[x13]+R[x15]
0000000 10000 01011 000 01011 0110011 add x11,x11,x16 #R[x11]←R[x11]+R[x16]
```

使用了临时寄存器x16



RISC-V基础整数指令集（RV32I）

控制转移类指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq =
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne ≠
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt <
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge ≥
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu

跳得很远

J 型：jal 功能为: PC←PC+SEXT[imm[20:1]<<1]；R[rd]←PC+4 下一条地址

Jump and link

禁止写入，永远为0，异常出不会再回来

jal x1/rd/,imm实现过程调用； jal x0/rd/,imm 实现无条件跳转。

I 型：jalr功能为: PC←R[rs1]+SEXT[imm[12]]；R[rd]←PC+4

Jump and link, return

$x_1 + 0 = x_1$ ⇒返回地址不变

指令 “jalr x0/rd/,x1/rs1/,0” 可实现过程调用的返回。

B型：皆为分支指令，其中，bltu、bgeu分别为无符号数比较小于、大于等于转移。转移目标地址=PC+SEXT[imm[12:1]<<1]

跳得较近

<<1: 指令地址总是2的倍数（RV32G、RV32C指令分别为4、2字节长）

RISC-V基础整数指令集（RV32I）

举例：若int型变量x、y、z分别存放在寄存器x5、x6、x7中，写出C语句“z=x+y;”对应的RISC-V机器级代码，要求检测是否溢出。

解：当x、y为int类型时，若“y<0且x+y≥x”或者 “y≥0且x+y<x”，则x+y溢出。可通过slti指令对y与0进行比较。

0000000 00110 00101 000 00111 0110011 add x7,x5,x6 #R[x7]←R[x5]+R[x6]

0000 0000 0000 00110 010 11100 0010011 slti x28,x6,0 #若R[x6]<0，则R[x28]←1

0000000 00101 00111 010 11101 0110011 slt x29,x7,x5 #若R[x7]<R[x5] 则R[x29]←1

0000010 11101 11100 001 10000 1100011 bne x28,x29,overflew #若R[x28]≠R[x29]

符号拓展

PC+Imm← #则转溢出处理

……

overflew: xxxxxxxx （某指令）

bne这条指令的二进制编码是怎么确定的?

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

RISC-V基础整数指令集（RV32I）

0000010 11101 11100 001 10000 1100011 bne x28,x29,overflew #若R[x28]≠R[x29]

…… #则转溢出处理

overflew: xxxxxxxx （某指令）

bne:

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

假定标号为overflew的指令地址与“bne x28…”这条指令地址相距80字节，

则“bne x28….”指令中的偏移量应为80，

因此，指令中的立即数为40 —— 40=0000 0010 1000B，

立即数与转移指令的偏移

按照B-型格式对立即数重组，

所以，该指令的机器码为“0000010 11101 11100 001 10000 1100011”

RISC-V基础整数指令集（RV32I）

存储访问指令

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

I 型：5条取数（Load）指令。功能: R[rd]←M[R[rs1]+SEXT[imm[12]]

lbu、lhu：分别为无符号字节、半字取，取出数据按0扩展为32位，装入rd

S型：3条存数（Store）指令。功能: M[R[rs1]+SEXT[imm[12]]←R[rs2]

sb、sh: 分别将rs2寄存器中低8、低16位写入存储单元中。

汇编形式如： lw rd, imm12(rs1)， sw rs2, imm12(rs1)

RISC-V基础整数指令集（RV32I）

系统控制类指令

31 25	24 20	19 15	14 12	11 7	6 0	
0000	pred	succ	00000	000	00000	0001111	I fence
0000	0000	0000	00000	001	00000	0001111	I fence.i
000000000000			00000	000	00000	1110011	I ecall
000000000000			00000	000	00000	1110011	I ebreak
csr			rs1	001	rd	1110011	I csrrw
csr			rs1	010	rd	1110011	I csrrs
csr			rs1	011	rd	1110011	I csrrc
csr			zimm	101	rd	1110011	I csrrwi
csr			zimm	110	rd	1110011	I csrrsi
csr			zimm	111	rd	1110011	I csrrci

fence：RISC-V架构在不同硬件线程之间使用宽松一致性模型，fence和fence.i 两条屏障指令，用于保证一定的存储访问顺序。

ecall和ebreak：陷阱（trap）指令，也称自陷指令，主要用于从用户程序陷入到操作系统内核（ecall）或调试环境（ebreak）执行，因此也称为环境（Environment）类指令。

csrxxx: 6条csr指令用于设置和读取相应的控制状态寄存器（CSR）。

(*) RISC-V可选的扩展指令集

- 标准扩展指令集
  - RV32I基础指令集之上，可标准扩展RV32M、RV32F/D、RV32A，以形成32位架构合集RV32IMAFD，也称为RV32G
  - RV32G基础上，对每个指令集进行调整和添加，可形成64位架构RV64G，原先在RV32G中处理的数据将调整为64位。但为了支持32位数据操作，每个64位架构指令集中都会添加少量32位数据处理指令。
- RISC-V扩展集包括
  - 针对64位架构需要，在47条RV32I指令基础上，增加12条整数指令（+RV64I），包括6条32位移位指令、3条32位加减运算指令、两条64位装入（Load）指令和1条64位存储（Store）指令，故RV64I共59条指令。
  - 针对乘除运算需要，提供了32位架构乘除运算指令集RV32M中的8条指令，并在此基础上增加了4条RV64M专用指令（+RV64M）
  - 针对浮点数运算的需要，提供了32位架构的单精度浮点处理指令集RV32F和双精度浮点处理指令集RV32D，并在此基础上分别增加了RV64F和RV64D专用指令集（+RV64F）和（+RV64D）。
  - 针对事务处理和操作原子性的需要，提供了32位架构原子操作指令集RV32A以及RV64A专用指令集（+RV64A）。关于事务处理和原子性操作问题的说明可参考第8章。
- 向量处理指令集RVV、未来可选扩展指令集RVB、RVE、RVH、……

64位架构指令举例

例：在64位RISC-V架构中，如何实现将一个32位常数00000000 00111101 00000101 00000000装入64位寄存器a0中?

注：在64位架构中，lui指令和32位架构中类似。将一个常数的高20位装入到64位寄存器中，具体规定如下

解： lui指令将常数中的第31~12位0000 0000 0011 1101 0000（976）装入到a0寄存器的第31~12位，同时，a0寄存器的第11~0位为全0，高32位按符号扩展（第31位为符号）为全0。

再将常数的低12位0101 0000 0000（对应十进制数1280）加到a0中。

因此，实现上述功能对应的汇编指令序列为：

```
lui     a0, 976
addi    a0, a0, 1280
```

回顾：RISC-V中整数的乘、除运算处理

- 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：将两个乘数同时按带符号整数相乘，高32位乘积存入rd中
  - mulhu：将两个乘数同时按无符号整数相乘，高32位乘积存入rd中
  - mulhsu：将两个乘数分别作为带符和无符整数相乘，高32位乘积存入rd
  - 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，硬件实际执行时其实只是执行了一条指令
  - 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
- RISC-V指令不检测和发出异常（除0），而是由系统软件自行处理
- 乘法指令（硬件）也可以生成溢出标志，只是RISCV没有这样做。

进一步思考（续第6章内容，请按需回顾）

- 可以根据高位乘积寄存器和低位乘积寄存器的内容来进行溢出判断（编译器可以生成相关的判断指令，由指令计算并存放溢出标志）

在字长为32位的计算机上，某C函数原型声明为：int imul_overflow(int x, int y); 该函数用于对两个int型变量x和y的乘积（也是int类型）判断是否溢出，若溢出则返回非0，否则返回0。请完成下列任务或回答下列问题。

（2）已知入口参数x、y分别在寄存器a0、a1中，返回值在a0中，写出实现imul_overflow函数功能的RISC-V汇编指令序列，并给出注解。（编译器中判断溢出的代码）

实现该功能的汇编指令序列不唯一，可能如下——

```
mul     t0, a0, a1          # x*y的低32位在t0中
mulh    a0, a0, a1          # x*y的高32位在a0中
srai    t0, t0, 31          # 乘积的低32位算术右移31位
xor     a0, a0, t0          # 按位异或，若结果为0，表示不溢出
```

（本处按题意，返回值写入a0，而不是写入溢出标志位）

进一步思考：除法结果的处理

- **RISC-V指令不检测和发出异常，而是由系统软件自行处理**

  如除法错， 不触发异常，而用特殊的商和余数来表示

Condition	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	$2^L - 1$	x	-1	x
Overflow (signed only)	–	–	-2^{L-1}	0

若整数x除以0，则指令执行结果为：商为全1，余数为x。

当最小的负整数除以-1时，会发生结果溢出，此时，相应指令执行结果为：商为被除数（即最小负整数），余数为0。

这样做的好处是：简化流水线的硬件实现

若编译器对除法错进行处理，可查看商和余数来判断

若编译器不处理除法错，则程序就得到错误结果，这种情况下需要程序员进行相应处理

本章总结1

- 指令格式
  - 定长指令字：所有指令长度一致
  - 变长指令字：指令长度有长有短
- 操作类型
  - 数据传送：数据在寄存器、主存单元、栈顶等处进行传送
  - 操作运算：各种算术运算、逻辑运算
  - 字符串处理：字符串查找、扫描、转换等
  - I/O操作： 与外设接口进行数据/状态/命令信息的交换
  - 程序流控制：条件转移、无条件转移、转子、返回等
  - 系统控制：启动、停止、陷阱指令（自愿访管）、空操作等
- 操作数类型（以Pentium处理器数据类型为例）
  - 序数或指针：8位、16位、32位无符号整数表示
  - 整数：16位、32位、64位三种补码表示的整数
  - 实数：IEEE754浮点数格式
  - 十进制数：18位十进制数，用80个二进位表示
  - 字符串：字节为单位的字符序列，一般用ASCII码表示
- 操作数宽度：有多种，如：字节、16位、32位、64位等

本章总结2

作业：习题2(4)、2(9)、3、6、7、8、10、11、13、15、17。5.12晚上24点截止

- 寻址方式
  - 立即：地址码直接给出操作数本身
  - 直接：地址码给出操作数所在的内存单元地址
  - 间接：地址码给出操作数所在的内存单元地址所在的内存单元地址
  - 寄存器：地址码给出操作数所在的寄存器编号
  - 寄存器间接：地址码给出操作数所在单元的地址所在的寄存器编号
  - 栈：操作数约定在栈中，总是从栈顶取数或存数
  - 偏移寻址：用基地址+形式地址得到操作数所在的内存单元地址

- 指令系统：决定了处理器的设计
  - 按地址码指定风格来分
    - 累加器型：一个操作数和结果都隐含在累加器中
    - 堆栈型：操作数和结果都隐含在堆栈中
    - 通用寄存器型：操作数明显地指定在哪个通用寄存器中
    - 装入/存储型：运算类指令的操作数只能在寄存器中，只有装入(Load)指令和存储(Store)指令才能访问内存
  - 按指令系统的复杂度来分
    - CISC：复杂指令系统计算机
    - RISC：精简指令系统计算机