

# 第六章 指令系统

---

- 指令系统概述
- RISC-V指令系统
- RISC-V寻址方式

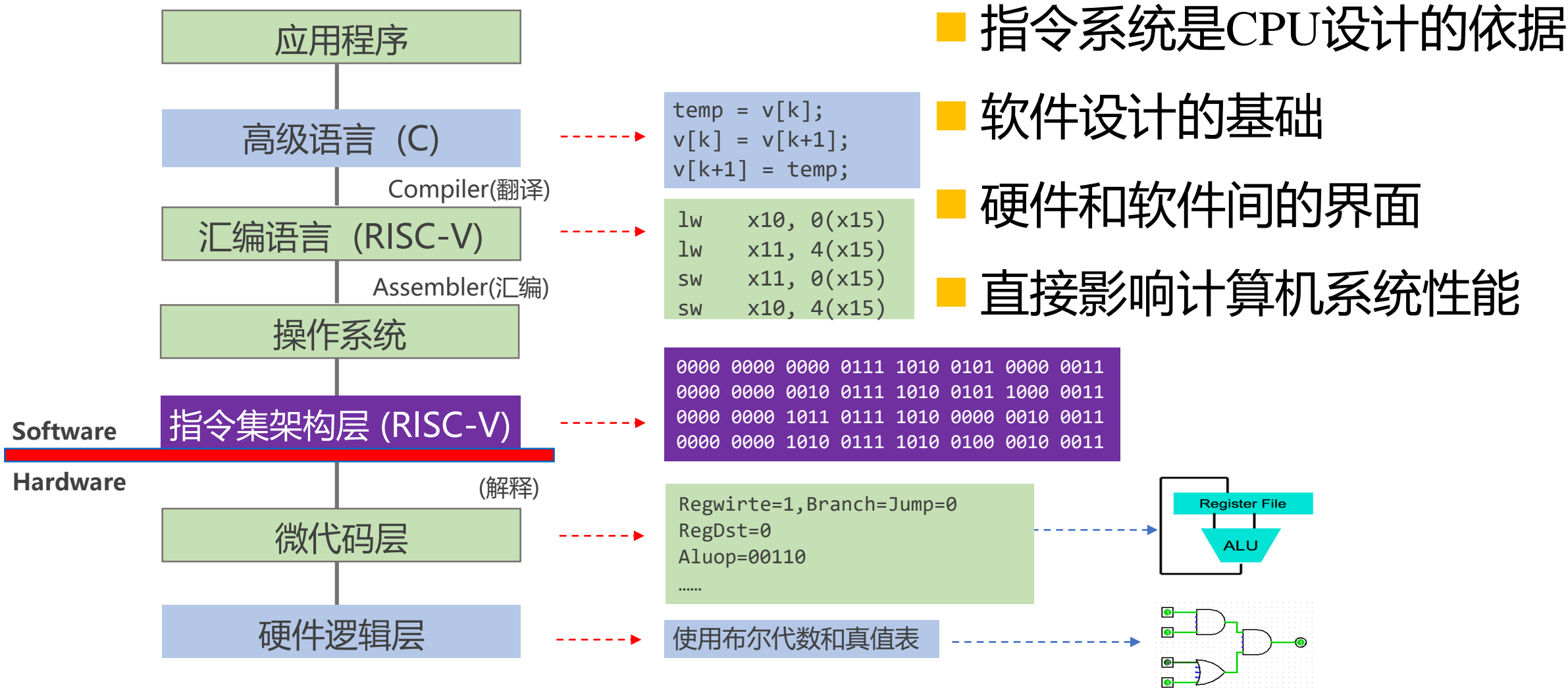


# 指令系统基本概念

---

- 机器指令（指令）
  - 计算机能直接识别、执行的某种操作命令
- 指令系统（指令集）IS:Instruction Set
  - 一台计算机中所有机器指令的集合
- 指令集系统架构（ISA: Instruction Set Architecture）
- 系列机
  - 基本指令系统相同，基本系统结构相同的计算机
    - IBM, PDP-11, VAX-11, Intel-x86
  - 解决软件兼容的问题

# 计算机指令系统层次



# 指令系统基本概念

---

- 完备性：指令丰富，功能齐全，使用方便
- 有效性：程序占空间小，执行速度快
- 规整性：
  - 对称性 （对不同寻址方式的支持）
  - 匀齐性 （对不同数据类型的支持）
  - 一致性 （指令长度和数据长度都是字节长度的整数倍）
- 兼容性：系列机软件向上兼容

# 第六章 指令系统

---

- 指令系统概述
- RISC-V指令系统
- RISC-V寻址方式



# 指令集体系结构 (ISA)


- 不同类型的CPU执行不同指令集，是设计CPU的依据

 1970 DEC PDP-11    1992 ALPHA(64位)

 1978 **x86**, 2001 IA64

 1980 PowerPC

 1981 **MIPS**

 1985 SPARC

 1991 arm

 2016 **RISC-V**

- 指令集优劣
  - 方便硬件设计，方便编译器实现，性能更优，成本功耗更低

# 指令集

- MIPS阵营
  - 龙芯

MIPS的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages), 其机制是尽量利用软件办法避免流水线中的数据相关问题。
- X86阵营
  - 兆芯 (VIA), 海光 (AMD)

X86架构 (The X86 architecture) 是微处理器执行的计算机语言指令集, 指一个intel通用计算机系列的标准编号缩写, 也标识一套通用的计算机指令集合。
- 自主指令
  - 申威 (Alpha指令集扩展)
- ARM阵营
  - 飞腾, 海思, 展讯, 松果

ARM处理器是英国Acorn有限公司设计的低功耗成本的第一款RISC微处理器。全称为Advanced RISC Machine。
- RISC-V阵营
  - 阿里-平头哥, 华米科技

RISC-V(读作“RISC-FIVE”)是基于精简指令集计算(RISC)原理建立的开放指令集架构(ISA), V表示为第五代RISC(精简指令集计算机),表示此前已经四代RISC处理器原型芯片。每一代RISC处理器都是在同一人带领下完成, 那就是加州大学伯克利分校的David A. Patterson教授。

# 指令系统发展方向

---

- CISC—复杂指令集计算机(Complex Instruction Set Computer)
  - 指令数量多，指令功能复杂，几百条指令。
  - 每条指令都有对应的电路设计，CPU电路设计复杂，功耗较大。
  - 对应编译器的设计简单（各种操作都有对应的指令）。
  - Intel x86
- RISC---精简指令集计算机(Reduced Instruction Set Computer)
  - 指令数量少，指令功能单一，通常只有几十条指令。
  - CPU设计相对简单，功耗较小。
  - 编译器的设计比较复杂（许多操作需要一些指令的灵活组合）
  - 1982年后的指令系统基本都是RISC
  - ARM、MIPS、RISC-V
- CISC、RISC互相融合



# x86指令集概述

---

- 1978年Intel发布了16位微处理器“8086”，x86架构诞生
  - 高性能
  - 扩展能力强
  - 操作系统的兼容性
  - 8086, 286, 386, 486, 586(Pentium)....等多个版本
- 大多数CPU产商(如Intel, AMD)使用的就是x86指令集架构
- x86架构由于其封闭性，相较于其他架构成本更高
- 有着更高的性能、更快的速度和兼容性

# 精减指令系统(RISC)

---

- 指令条数少，保留使用频率最高的简单指令，指令定长
  - 便于硬件实现，用软件实现复杂指令功能
- Load/Store架构
  - 只有存/取数指令才能访问存储器，其余指令的操作都在寄存器之间进行，便于硬件实现
- 指令长度固定，指令格式简单、寻址方式简单
  - 便于硬件实现
- CPU设置大量寄存器（32~192）
  - 便于编译器实现
- RISC CPU采用硬布线控制，CISC采用微程序
- 一个时钟周期完成一条机器指令（单周期模型）

# MIPS指令概述

---

- MIPS (Microprocessor without Interlocked Pipeline Stages)
  - 1981年斯坦福大学Hennessy教授研究小组研制并商用
  - 简单的Load/Store结构
  - 易于流水线CPU设计
  - 易于编译器开发
  - 寻址方式，指令操作非常简单
  - MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V多个版本
- 广泛用于嵌入式系统，在PC机、服务器中也有应用
- 更适合于教学，相比X86更加简洁雅致，不会陷入繁琐的细节



# MIPS X86 差异

	X86	MIPS
1	变长（1-15bytes）	定长指令
2	指令数多 CISC	指令数少 RISC
3	8个通用寄存器	32个通用寄存器
4	寻址方式复杂	寻址方式简单
5	有标志寄存器	无标志寄存器
6	最多两地址指令	三地址指令
7	无限制	只有Load/store能访问存储器
8	有堆栈指令 push, pop	无堆栈指令（访存指令代替）
9	有I/O指令	无I/O指令(设备统一编址)
10	参数传递：栈帧	参数传递（4寄存器+栈帧）

# RISC-V指令概述

- 完全开放的 ISA
- 大道至简，简单就是美
  - 包含一个最小的核心冻结的ISA（可支撑OS，方便教学）
  - 适合硬件实现，而不仅仅是适用于模拟或者二进制翻译
- 后发优势
  - 模块化的可扩展指令集
  - 方便简化硬件实现，提升性能
    - 更规整的指令编码、更简洁的运算指令、更简洁的访存模式：Load/Store架构
    - 高效分支跳转指令（减少指令数目）、简洁的子程序调用
    - 无条件码执行、无分支延迟槽



## 可配置的通用寄存器组

RISC-V架构支持32位或者64位的架构，32位架构由RV32表示，其每个通用寄存器的宽度为32比特；64位架构由RV64表示，其每个通用寄存器的宽度为64比特。RISC-V架构的整数通用寄存器组，包含32个（I架构）或者16个（E架构）通用整数寄存器，其中整数寄存器0被预留为常数0，其他的31个（I架构）或者15个（E架构）为普通的通用整数寄存器。如果使用浮点模块（F或者D），则需要另外一个独立的浮点寄存器组，包含32个通用浮点寄存器。如果仅使用F模块的浮点指令子集，则每个通用浮点寄存器的宽度为32比特；如果使用了D模块的浮点指令子集，则每个通用浮点寄存器的宽度为64比特。

## 规整的指令编码

在流水线中能够尽快地读取通用寄存器组，往往是处理器流水线设计的期望之一，这样可以提高处理器性能和优化时序。这个看似简单的道理在很多现存的商用RISC架构中都难以实现，因为经过多年反复修改不断添加新指令后，其指令编码中的寄存器索引位置变得非常凌乱，给译码器造成了负担。得益于后发优势和总结了多年来处理器发展的经验，RISC-V的指令集编码非常规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置。因此指令译码器（Instruction Decoder）可以非常便捷地译码出寄存器索引，然后读取通用寄存器组（Register File，Regfile）。

## 简洁的存储器访问指令

与所有的RISC处理器架构一样，RISC-V架构使用专用的存储器读（Load）指令和存储器写（Store）指令访问存储器（Memory），其他的普通指令无法访问存储器，这种架构是RISC架构常用的一个基本策略。这种策略使得处理器核的硬件设计变得简单。存储器访问的基本单位是字节（Byte）。RISC-V的存储器读和存储器写指令支持一个字节（8位）、半字（16位）、单字（32位）为单位的存储器读写操作。如果是64位架构还可以支持一个双字（64位）为单位的存储器读写操作。RISC-V架构的存储器访问指令还有如下显著特点。为了提高存储器读写的性能，RISC-V架构推荐使用地址对齐的存储器读写操作，但是也支持地址非对齐的存储器操作RISC-V架构。处理器既可以选择用硬件来支持，也可以选择用软件来支持。由于现在的主流应用是小端格式（Little-Endian），RISC-V架构仅支持小端格式。有关小端格式和大端格式的定义和区别，在此不做过多介绍。若对此不太了解的初学者可以自行查阅学习。很多的RISC处理器都支持地址自增或者自减模式，这种自增或者自减的模式虽然能够提高处理器访问连续存储器地址区间的性能，但是也增加了设计处理器的难度。RISC-V架构的存储器读和存储器写指令不支持地址自增自减的模式。RISC-V架构采用松散存储器模型（Relaxed Memory Model），松散存储器模型对于访问不同地址的存储器读写指令的执行顺序不作要求，除非使用明确的存储器屏障（Fence）指令加以屏蔽。有关存储器模型（Memory Model）和存储器屏障指令的更多信息，请参见附录A13。这些选择都清楚地反映了RISC-V架构力图简化基本指令集，从而简化硬件设计的哲学。RISC-V架构如此定义是具有合理性的，能达到能屈能伸的效果。例如，对于低功耗的简单CPU，可以使用非常简单的硬件电路即可完成设计；而对于追求高性能的超标量处理器，则可以通过复杂设计的动态硬件调度能力来提高性能。

## 高效的分支跳转指令

RISC-V架构有两条无条件跳转指令（Unconditional Jump），即jal指令与jalr指令。跳转链接（Jump and Link）指令——jal指令可用于进行子程序调用，同时将子程序返回地址存在链接寄存器（Link Register，由某一个通用整数寄存器担任）中。跳转链接寄存器（Jump and Link-Register）指令——jalr指令能够用于子程序返回指令，通过将jal指令（跳转进入子程序）保存的链接寄存器用于jalr指令的基地址寄存器，则可以从子程序返回。

RISC-V架构有6条带条件跳转指令（Conditional Branch），这种带条件的跳转指令跟普通的运算指令一样直接使用两个整数操作数，然后对其进行比较。如果比较的条件满足，则进行跳转，因此此类指令将比较与跳转两个操作放在一条指令里完成。作为比较，很多其他的RISC架构的处理器需要使用两条独立的指令。\*\*\*条指令先使用比较指令，比较的结果被保存到状态寄存器之中；第二条指令使用跳转指令，判断前一条指令保存在状态寄存器当中的比较结果为真时，则进行跳转。相比而言，RISC-V的这种带条件跳转指令不仅减少了指令的条数，同时硬件设计上更加简单。

对于没有配备硬件分支预测器的低端CPU，为了保证其性能，RISC-V的架构明确要求采用默认的静态分支预测机制，即如果是向后跳转的条件跳转指令，则预测为“跳”；如果是向前跳转的条件跳转指令，则预测为“不跳”，并且RISC-V架构要求编译器也按照这种默认的静态分支预测机制来编译生成汇编代码，从而让低端的CPU也得到不错的性能。在低端的CPU中，为了使硬件设计尽量简单，RISC-V架构特地定义了所有的带条件跳转指令跳转目标的偏移量（相对于当前指令的地址）都是有符号数，并且其符号位被编码在固定的位置。因此这种静态预测机制在硬件上非常容易实现，硬件译码器可以轻松找到固定的位置，判断该位置的比特值为1，表示负数（反之则为正数）。根据静态分支预测机制，如果是负数，则表示跳转的目标地址为当前地址减去偏移量，也就是向后跳转，则预测为“跳”。当然，对于配备有硬件分支预测器的高端CPU，则还可以采用高级的动态分支预测机制来保证性能。



## 简洁的子程序调用

为了理解此节，需先对一般RISC架构中程序调用子函数的过程予以介绍，其过程如下。

进入子函数之后需要用存储器写（Store）指令来将当前的上下文（通用寄存器等的值）保存到系统存储器的堆栈区内，这个过程通常称为“保存现场”。

在退出子程序时，需要用存储器读（Load）指令来将之前保存的上下文（通用寄存器等的值）从系统存储器的堆栈区读出来，这个过程通常称为“恢复现场”。

“保存现场”和“恢复现场”的过程通常由编译器编译生成的指令完成，使用高层语言（例如C语言或者C++）开发的开发者对此可以不用太关心。高层语言的程序中直接写上一个子函数调用即可，但是这个底层发生的“保存现场”和“恢复现场”的过程却是实实在在地发生着（可以从编译出的汇编语言里面看到那些“保存现场”和“恢复现场”的汇编指令），并且还需要消耗若干的CPU执行时间。

为了加速“保存现场”和“恢复现场”的过程，有的RISC架构发明了一次写多个寄存器到存储器中（Store Multiple），或者一次从存储器中读多个寄存器出来（Load Multiple）的指令。此类指令的好处是一条指令就可以完成很多事情，从而减少汇编指令的代码量，节省代码的空间大小。但是“一次读多个寄存器指令”和“一次写多个寄存器指令”的弊端是会让CPU的硬件设计变得复杂，增加硬件的开销，也可能损伤时序，使得CPU的主频无法提高，作者曾经设计此类处理器时便深受其苦。

RISC-V架构则放弃使用“一次读多个寄存器指令”和“一次写多个寄存器指令”。如果有的场合比较介意“保存现场”和“恢复现场”的指令条数，那么可以使用公用的程序库（专门用于保存和恢复现场）来进行，这样就可以省掉在每个子函数调用的过程中都放置数目不等的“保存现场”和“恢复现场”的指令。此选择再次印证了RISC-V追求硬件简单的哲学，因为放弃“一次读多个寄存器指令”和“一次写多个寄存器指令”可以大幅简化CPU的硬件设计，对于低功耗小面积的CPU可以选择非常简单的电路进行实现；而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证CPU能够快速地跳转执行，从而可以选择使用公用的程序库（专门用于保存和恢复现场）的方式减少代码量，同时达到高性能。



## 无条件码执行

很多早期的RISC架构发明了带条件码的指令，例如在指令编码的头几位表示的是条件码（Conditional Code），只有该条件码对应的条件为真时，该指令才被真正执行。

这种将条件码编码到指令中的形式可以使编译器将短小的循环编译成带条件码的指令，而不用编译成分支跳转指令。这样便减少了分支跳转的出现，一方面减少了指令的数目；另一方面也避免了分支跳转带来的性能损失。然而，这种“条件码”指令的弊端同样会使CPU的硬件设计变得复杂，增加硬件的开销，也可能损伤时序使得CPU的主频无法提高。

RISC-V架构则放弃使用这种带“条件码”指令的方式，对于任何的条件判断都使用普通的带条件分支跳转指令。此选择再次印证了RISC-V追求硬件简单的哲学，因为放弃带“条件码”指令的方式可以大幅简化CPU的硬件设计，对于低功耗小面积的CPU可以选择非常简单的电路进行实现，而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证CPU能够快速地跳转执行达到高性能。

## 无分支延迟槽

很多早期的RISC架构均使用了“分支延迟槽（Delay Slot）”，具有代表性的便是MIPS架构。在很多经典的计算机体系结构教材中，均使用MIPS对分支延迟槽进行介绍。分支延迟槽就是指在每一条分支指令后面紧跟的一条或者若干条指令不受分支跳转的影响，不管分支是否跳转，这后面的几条指令都一定会被执行。

早期的RISC架构很多采用了分支延迟槽诞生的原因主要是当时的处理器流水线比较简单，没有使用高级的硬件动态分支预测器，使用分支延迟槽能够取得可观的性能效果。然而，这种分支延迟槽使得CPU的硬件设计变得极为别扭，CPU设计人员对此苦不堪言。

RISC-V架构则放弃了分支延迟槽，再次印证了RISC-V力图简化硬件的哲学，因为现代的高性能处理器的分支预测算法精度已经非常高，可以有强大的分支预测电路保证CPU能够准确地预测跳转执行达到高性能。而对于低功耗、小面积的CPU，由于无须支持分支延迟槽，硬件得到极大简化，也能进一步减少功耗和提高时序。

# 模块化的RISC-V 指令子集

- RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示
- RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集

基本指令集	指令数	描 述
RV32I	47	32位地址空间与整数指令，支持32个通用整数寄存器
RV32E	47	RV32I的子集，仅支持16个通用整数寄存器
RV64I	59	64位地址空间与整数指令及一部分32位的整数指令
RV128I	71	128位地址空间与整数指令及一部分64位和32位的指令

扩展指令集	指令数	描 述
M	8	整数乘法与除法指令
A	11	存储器原子（Atomic）操作指令和Load-Reserved/Store-Conditional指令
F	26	单精度（32比特）浮点指令
D	26	双精度（64比特）浮点指令，必须支持F扩展指令

# 硬件设计四原则

---

- 简单性来自规则性（Simplicity favors regularity）
  - 指令越规整设计越简单
- 越小越快（Smaller is faster）
- 加快经常性事件（Make the common case fast）
- 好的设计需要适度的折衷

Good design demands good compromises

# 汇编语言的变量---寄存器

---

- 汇编语言不能使用变量（C、JAVA可以）
  - `int a; float b;`
  - 寄存器变量没有数据类型
- 汇编语言的操作对象是寄存器
  - 好处：寄存器是最快的数据单元
  - 缺陷：寄存器数量有限，需仔细高效的使用各寄存器
- RISC-V包括32个通用寄存器（约定详见下表P75图2-14）
  - `x0, x1, x2, ... x30, x31`

# 32个RISC-V寄存器

寄存器	助记符	释义
x0	zero	固定值为0
x1	ra	返回地址(Return Address)
x2	sp	栈指针(Stack Pointer)
x3	gp	全局指针(Global Pointer)
x4	tp	线程指针(Thread Pointer)
x5-7	t0-2	临时寄存器
x8	s0/fp	save寄存器/帧指针(Frame Pointer)
x9	s1	save寄存器
x10-11	a0-1	函数参数 / 函数返回值
x12-17	a2-7	函数参数
x18-27	s2-11	save寄存器
x28-31	t3-6	临时寄存器

# 加减指令

---

- 加法

- $a = b + c$  (in C)
- `add x1, x2, x3` (in RISC-V)
- `a, b, c` 编译后对应寄存器 `x1, x2, x3`

- 减法

- $d = e - f$  (in C)
- `sub x3, x4, x5` (in RISC-V)
- `d, e, f` 编译后对应寄存器 `x3, x4, x5`

# 加减指令

---

- 如何编译下面的C语言表达式?

$a = b + c + d - e;$

- 编译成汇编指令

add x10, x1, x2

# temp = b + c

add x10, x10, x3

# temp = temp + d

sub x10, x10, x4

# a = temp - e

- 一个简单的C语言表达式变成多条汇编语句

# 立即数

---

- 立即数相加指令

$f = g + 10$  (in C)

`addi x3, x4, 10` (in RISC-V)

$f, g$ 编译后对应寄存器 `x3, x4`

- 语法与`add`指令类似，只是最后一个参数是一个数字而不是寄存器



# 立即数

---

- 在RISC-V中没有立即数减法：为什么？
  - 有add和sub，但没有addi对应的项
- 限制可执行的操作类型
  - 如果一个操作可以分解成一个更简单的操作，不要包含它

$f = g - 10$  (in C)

addi x3, x4, -10 (in RISC-V)

# 寄存器 x0

---

- 一个特殊的立即数——数字0，经常出现在代码中
- 所以寄存器0（x0）硬连线到值0

$f = g$  (in C)

add x3, x4, x0 (in RISC-V)

f, g编译后对应寄存器 x3, x4

# 内存数据访问指令

---

- 读内存指令

`int A[100];`                      (in C)

`g = h + A[3];`

`lw x10, 12(x15)`              # x15为A[0]地址 (in RISC-V)

`add x11, x12, x10`            # `g = h + A[3]`

- 基址寻址

- 基址寄存器(x15) + 偏移量(12)

# 内存数据访问指令

---

- 写内存指令

`int A[100];`                      (in C)

`A[10] = h + A[3];`

`lw x10, 12(x15)`              # x15为A[0]地址 (in RISC-V)

`add x10, x12, x10`            #  $h + A[3]$ 赋予寄存器x10

`sw x10, 40(x15)`              #  $A[10] = h + A[3]$

# 内存数据访问指令

---

- 除了字数据传输(lw, sw), RISC-V还有字节数据传输:
  - 读字节: lb
  - 写字节: sb
- 格式与lw, sw一致
  - 例如: lb x10, 3(x11)

# 条件判断分支转移指令

---

- Branch分支类型
  - Conditional Branch条件分支
    - 根据比较结果更改控制流
    - if equal (beq), if not equal (bne)
    - if less than (blt), if greater than or equal (bge)
  - Unconditional Branch无条件分支
    - 始终分支 jump (j)

# 条件判断分支转移指令

if-else

$f \rightarrow x10, g \rightarrow x11, h \rightarrow x12, i \rightarrow x13, j \rightarrow x14$

if ( $i == j$ )

$f = g + h;$

else

$f = g - h;$



bne x13, x14, Else

add x10, x11, x12

j Exit

Else: sub x10, x11, x12

Exit:

# 条件判断分支转移指令

---

- 大小比较

if (reg1 < reg2)            (in C)

    goto label;

- 将寄存器视为有符号整数

    blt reg1, reg2, label            (in RISC-V)

- 将寄存器视为无符号整数

    bltu reg1, reg2, label (in RISC-V)



# 循环结构

- `int A[20];`

`int sum = 0;`

`for (int i = 0; i < 20; i++)`

`sum += A[i];`

数组基地址保存在 x8.



`add x9, x8, x0    # x9=&A[0]`

`add x10, x0, x0   # sum=0`

`add x11, x0, x0   # i=0`

`addi x13, x0, 20   # x13=20`

Loop:

`bge x11, x13, Done`

`lw x12, 0(x9)     # x12=A[i]`

`add x10, x10, x12`

`addi x9, x9, 4     # &A[i+1]`

`addi x11, x11, 1   # i++`

`j Loop`

Done:

# 逻辑运算指令

- 逻辑运算

- 寄存器: and, or, xor      and x5, x6, x7      # x5 = x6 & x7

- 立即数: andi, ori, xori      andi x5, x6, 3      # x5 = x6 & 3

- RISC-V中没有NOT

- 对  $11111111_2$  使用xori

- 例: xori x5, x6, -1      # x5 =  $\overline{x6}$

# 逻辑运算指令

---

- 逻辑移位指令

- sll, slli, srl, srli      slli x11, x12, 2      # x11=x12<<2

- 算术移位指令

- sra, srai      srai x10, x10, 4      # x10=x10>>4

# 函数调用

---

- 函数调用的6个步骤
  - 将参数放在过程函数可以访问到的位置
  - 将控制转交给函数
  - 获取函数所需的存储资源
  - 执行所需的任务
  - 将结果值放在调用程序可以访问到的位置
  - 将控制返回到初始点，因为过程可以从程序中的多个点调用

# 函数调用

---

- C语言函数调用

- `int function(int a ,int b)      # a, b: s0, s1`
- `{ return (a+b); }`

- RISC-V实现过程调用的机制

- 返回地址寄存器      `x1(ra)`
- 参数寄存器      `x10-x17`
- 返回值寄存器      `x10-x11`
- 保存寄存器      `x8,x9,x18-x27`
- 临时寄存器      `x5-x7,x28-x31`
- 堆栈指针      `x2(sp)`

# 函数调用

- RISC-V 函数调用

1000 mv a0, s0

1004 mv a1, s1

1008 addi ra, zero, 1016      # ra = 1016

1012 j sum      # jump to sum

1016 ... # 下个指令

...

2000 sum: add a0, a0, a1

2004 jr ra      # jump register

- 为什么要使用jr而不使用j?
- sum可能会被很多地方调用，所以不能返回到一个固定地点。
- 调用sum函数必须能够以某种方式返回。

# 函数调用

---

- 一条指令实现跳转和保存返回地址
  - 跳转-链接指令 (jal)
- Before
  - 1008 addi ra, zero, 1016    # ra = 1016
  - 1012 j sum                    # jump to sum
- After
  - 1008 jal sum                # ra = 1012, jump to sum

# 栈的使用

---

- 在调用函数之前需要一个保存旧值的地方，在返回时还原它们，然后删除
- 堆栈：后进先出（LIFO）队列
  - push: 将数据放入堆栈
  - pop: 从堆栈中删除数据
- sp是RISC-V中的堆栈指针
- 按历史惯例，栈按照从高到低的地址顺序增长
  - push 减 sp, pop 增 sp



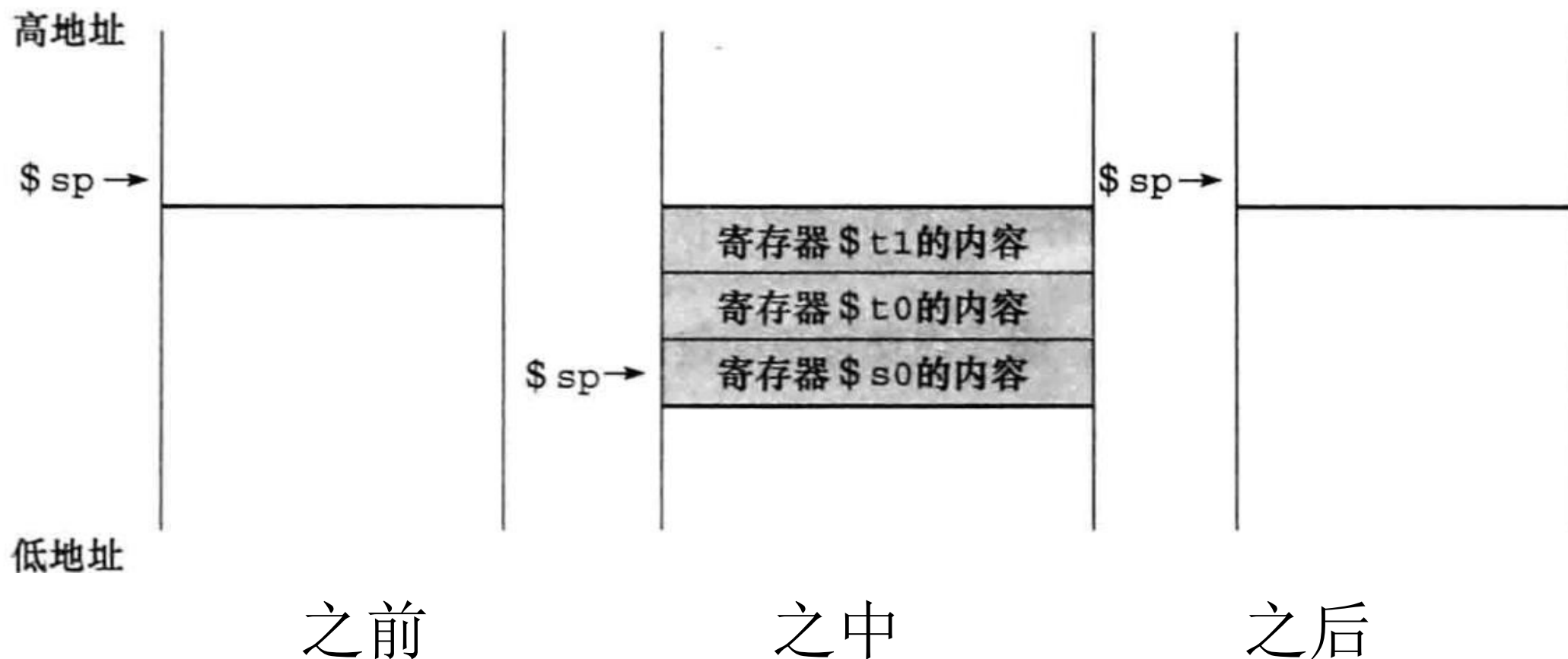
# 栈的使用

---

- 堆栈框架包括：
  - 返回“指令”地址
  - 参数
  - 其他局部变量的空间
- 堆栈帧占据连续的内存块，堆栈指针指示堆栈帧底部的位置
- 当过程结束时，堆栈帧从堆栈中弹出，为将来的堆栈帧释放内存

# 栈的使用

- 函数调用之前、之中和之后栈的情况



# 栈的使用

---

- 如果一个函数调用一个函数呢？递归函数调用如何进行？
- 嵌套过程

```
int numSquare (int x, int y){  
    return mult (x, x)+y;  
}
```

- 函数numSquare调用函数mult，寄存器ra保存的返回地址上存在冲突
- 调用mult前需要使用stack保存numSquare的返回地址

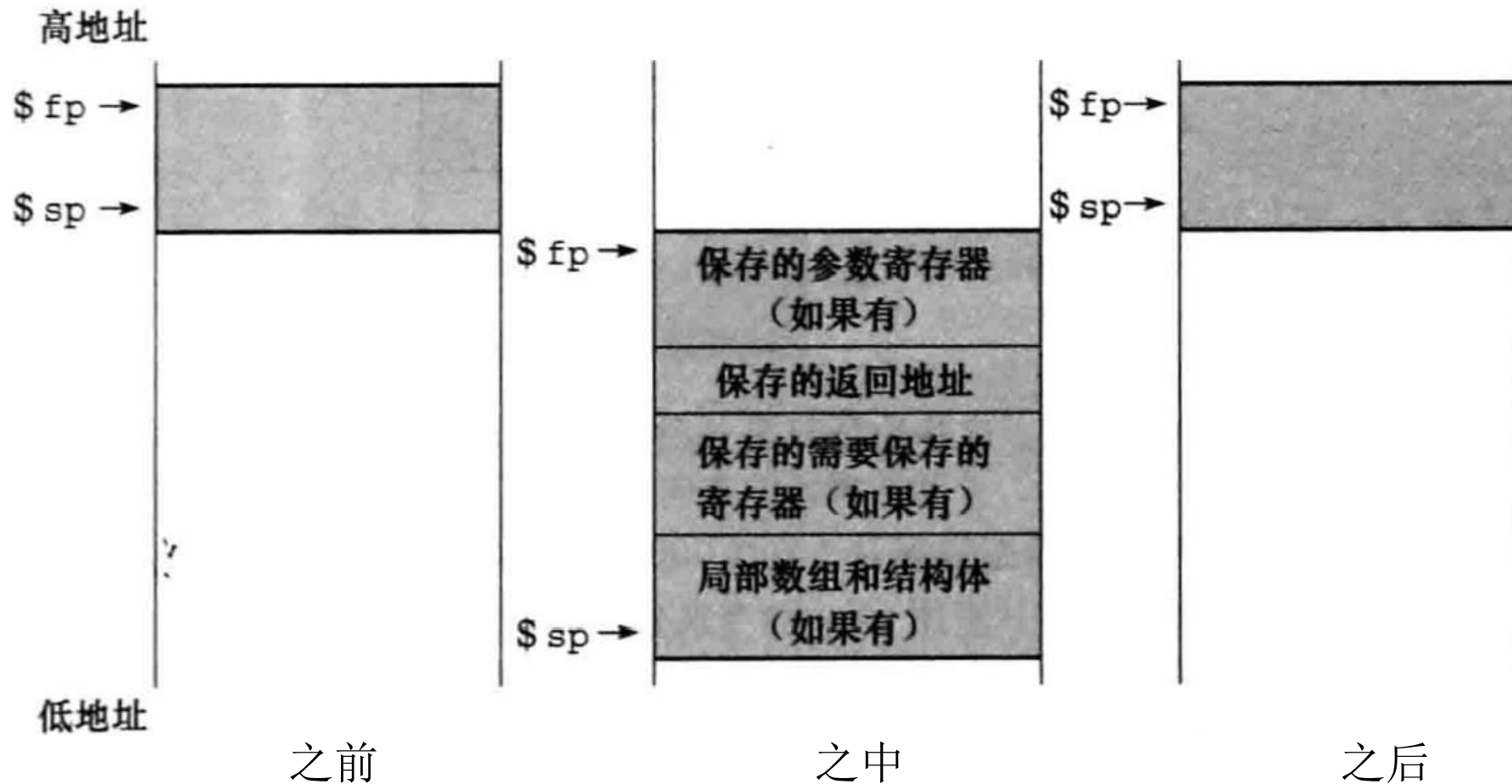
# 栈的使用

---

- 当被调用函数从执行中返回时，调用函数需要知道哪些寄存器可能已经更改，哪些寄存器保证不变。
- RISC-V函数调用约定将寄存器分为两类：
  - 在函数调用中保留      `sp, gp, tp, x8-x9, x18-x27`
  - 在函数调用中不保留   `x10-x17, x1(ra), x5-x7, x28-x31`

# 栈的使用

- 函数调用之前，之中和之后栈的情况



# 栈的使用

---

- 寄存器sp总是指向堆栈中最后使用的空间
- 为了使用stack，将这个指针减少所需的空间量，然后用信息填充它
- 如何编译？

```
int numSquare (int x, int y){  
    return mult (x, x)+y; }  
}
```

## 栈的使用

- sumSquare:

```
push    addi sp, sp, -8    # space on stack  
        sw ra, 4(sp)      # save return addr  
        sw a1, 0(sp)      # save y  
        mv a1, a0          # mult(x,x)  
        jal mult          # call mult  
        lw a1, 0(sp)      # restore y  
        add a0, a0, a1     # mult()+y  
pop     lw ra, 4(sp)      # get ret addr  
        addi sp, sp, 8    # restore stack  
        jr ra  
mult:...
```

# RISC-V指令表示

---

- 我们处理的大多数数据都是字（32-bit）：
  - lw和sw一次访问一个字的内存
- 那么我们如何表示指令呢？
  - 记住：计算机只懂1和0，所以汇编字符串“add x10, x11, x0”对硬件来说毫无意义
  - RISC-V追求简单性：因为数据是以字为单位的，所以指令也应该是固定大小的32位字
    - 用于RV32、RV64、RV128的32位指令相同



# RISC-V指令表示

- 一个字是32位，所以把指令字分成“字段”
- 每个字段都告诉处理器一些关于指令的信息
- 理论上可以为每条指令定义不同的字段
- RISC-V定义了六种基本类型的指令格式：
  - R型指令                      用于寄存器 - 寄存器操作
  - I型指令                      用于短立即数和访存 load 操作
  - S型指令                      用于访存 store 操作
  - B型指令                      用于条件跳转操作
  - U型指令                      用于长立即数操作
  - J型指令                      用于无条件跳转操作

# 指令格式

---

- 表示一条指令的机器字，称为指令字，简称指令
- 指令格式：用二进制代码表示指令的结构形式
  - 指令要求计算机处理什么数据？
  - 指令要求计算机对数据做什么处理？
  - 计算机怎样才能得到要处理的数据？

操作码字段	操作数字段
-------	-------

# 操作码(OP)与地址码(AC)

---

- 操作码字段长度决定指令系统规模
  - 每条指令对应一个操作码
  - 定长操作码  $\text{Length}_{\text{OP}} = \log_2 n$
  - 变长操作码 操作码向不用的地址码字段扩展
- 操作数字段可能有多个
  - 寻址方式字段 长度与寻址方式种类有关，可能隐含在操作码字段
  - 地址码字段 作用及影响、长度和寻址方式有关

# RISC-V指令格式

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

# RISC-V指令格式

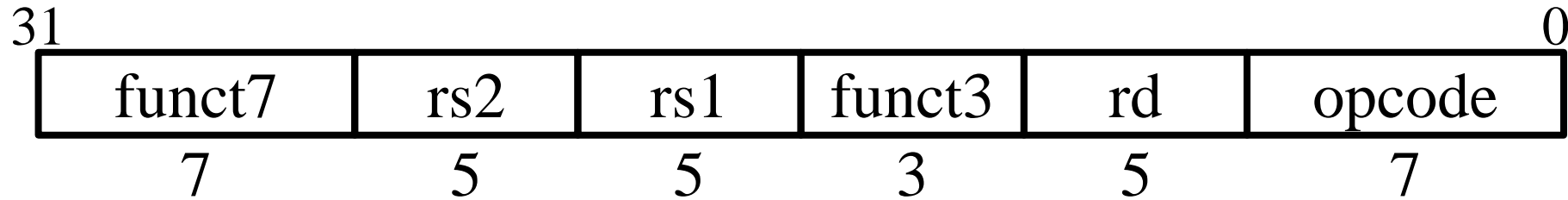
---

**RISC-V** 标准指令集主要有如下几种框架：

- **R-format** for **R**egister-register arithmetic/logical operations
- **I-format** for register-**I**mmEDIATE arith/logical operations and loads
- **S-format** for **S**tore
- **B-format** for **B**ranch
- **U-format** for 20-bit **U**pper immediate instructions
- **J-format** for **J**ump
- Others: Used for OS & Synchronization

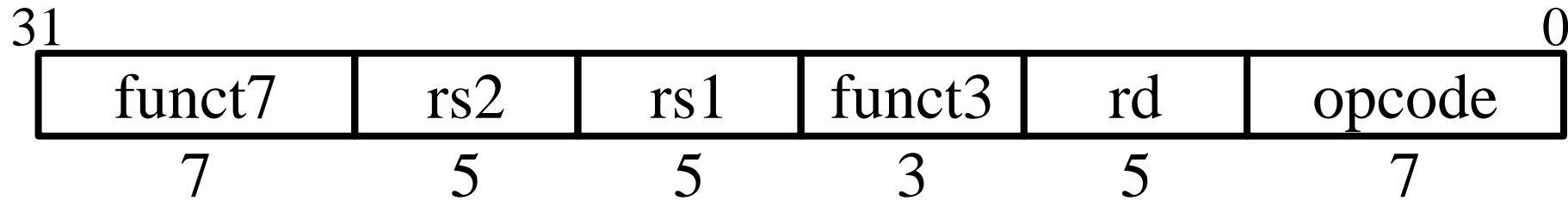
**R**即**Reg**相关；**I**即当即数相关；**S**存储相关；**B**分支相关；**U**高位数相关（由于一条32位指令中没法表示高达32位的数据）；**J**跳转相关。

# R型指令



- 32位指令字，分为6个不同位数的字段
- opcode(操作码): 指定它是什么指令
  - 对于所有R型指令，此字段等于**0110011**
- funct7+funct3: 这两个字段结合操作码描述要执行的操作

# R型指令

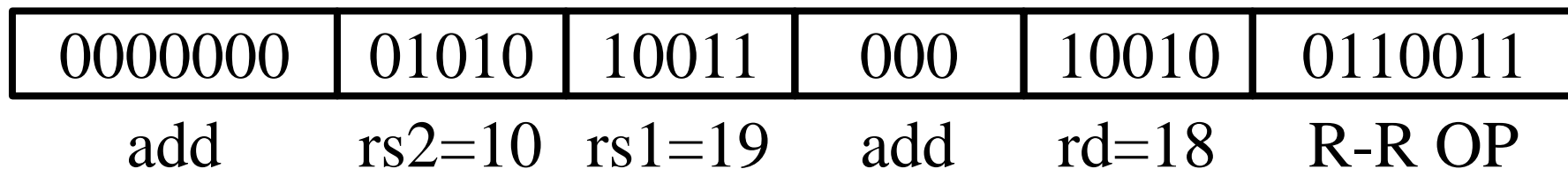
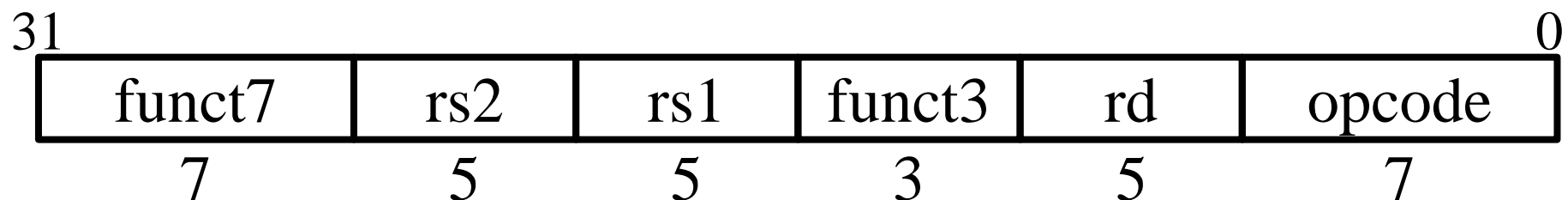


- rs1（源寄存器1）：指定第一个寄存器操作数
- rs2：指定第二个寄存器操作数
- rd（目的寄存器）：指定接收计算结果的寄存器
- 每个字段保存一个5位无符号整数（0-31），对应于一个寄存器号（x0-x31）（寄存器约定详见P75图2-14）

# R型指令

- RISC-V汇编指令:

- add x18, x19, x10

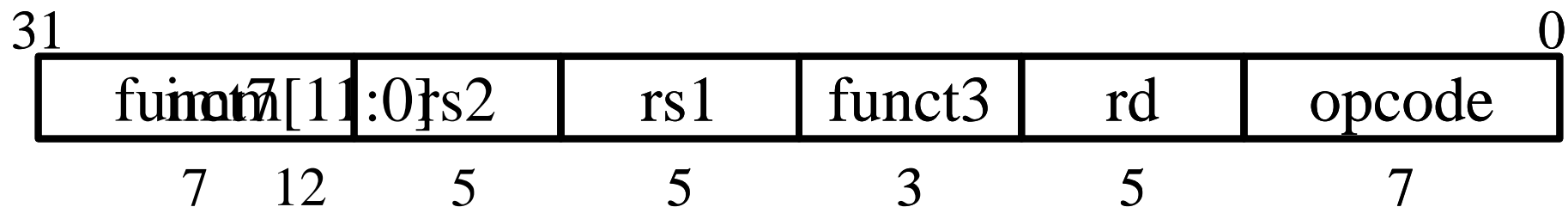




# R型指令

00000000	rs2	rs1	000	rd	0110011	add
01000000	rs2	rs1	000	rd	0110011	sub
00000000	rs2	rs1	001	rd	0110011	sll
00000000	rs2	rs1	010	rd	0110011	slt
00000000	rs2	rs1	011	rd	0110011	sltu
00000000	rs2	rs1	100	rd	0110011	xor
00000000	rs2	rs1	101	rd	0110011	srl
01000000	rs2	rs1	101	rd	0110011	sra
00000000	rs2	rs1	110	rd	0110011	or
00000000	rs2	rs1	111	rd	0110011	and

# I型指令

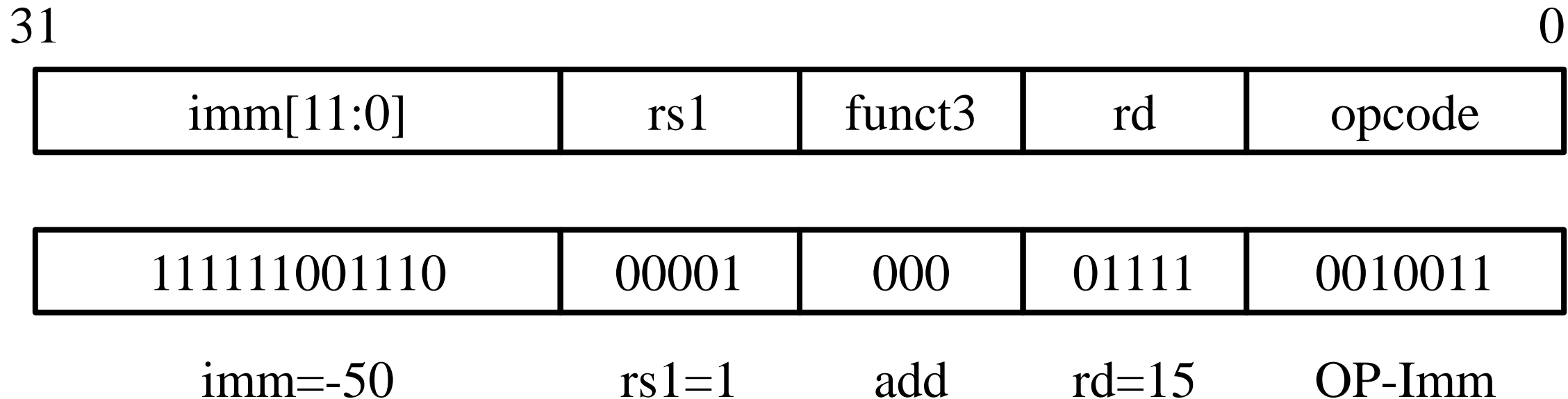


- 只有一个字段与R型指令不同，rs2和funct7被12位有符号立即数imm[11:0]替换
- 其余字段（rs1、funct3、rd、opcode）与之前相同
- 在算术运算前，立即数总是符号扩展到32位
- 如何处理大于12位的立即数？

# I型指令

- RISC-V汇编指令:

- addi x15, x1, -50



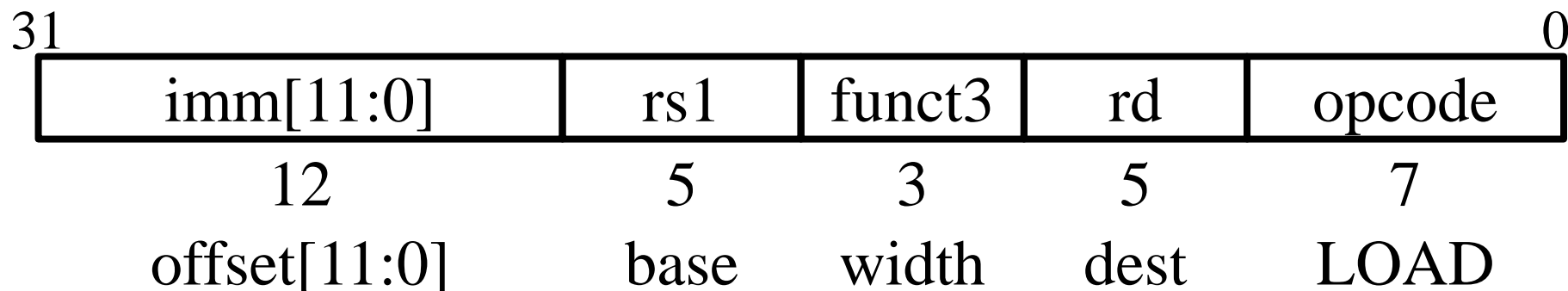
# I型指令

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
000000	shamt	rs1	001	rd	0010011	slli
000000	shamt	rs1	101	rd	0010011	srli
010000	shamt	rs1	101	rd	0010011	srai

其中一个高阶立即数位用于区分  
“逻辑右移”（SRLI）和“算术  
右移”（SRAI）

“按立即数移位”指令仅将立  
即数的低位6位用作移位量（只  
能移位0-63位位置）

# I型指令

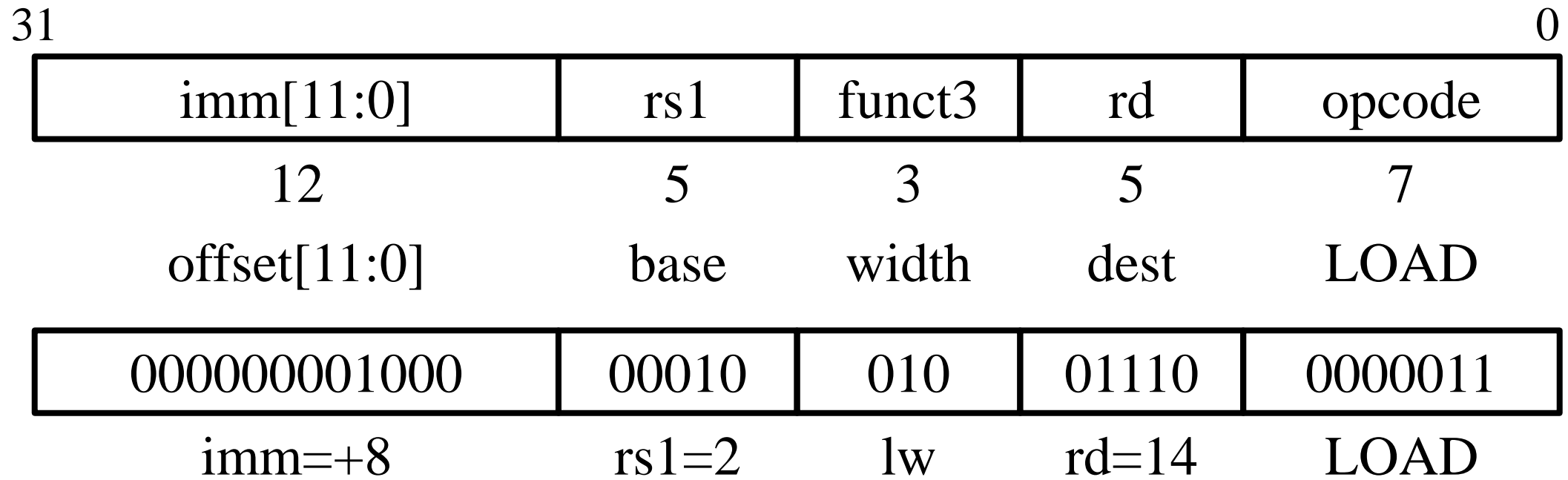


- Load指令也是I型指令
- 12位有符号立即数加寄存器rs1的基址，形成内存地址
  - 这与add immediate操作非常相似，但用于创建地址
- 从内存读取到的值存储在寄存器rd中

# I型指令

- RISC-V汇编指令:

- lw x14, 8(x2)



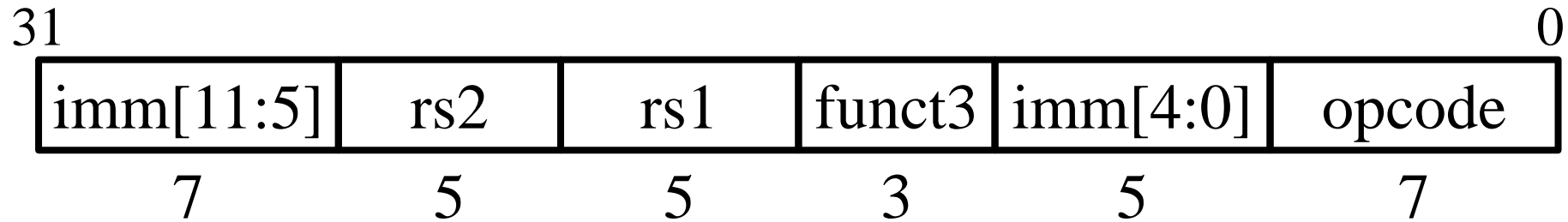
# I型指令

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	011	rd	0000011	ld
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu
imm[11:0]	rs1	110	rd	0000011	lwu

funct3字段对读取数据的大小  
和“有符号性”进行编码

- LBU是“读取无符号字节”
- LH是“读取半字”，读取16位（2字节）并扩展以填充32位寄存器
- LHU是“读取无符号半字”，零扩展为16位以填充32位寄存器

# S型指令



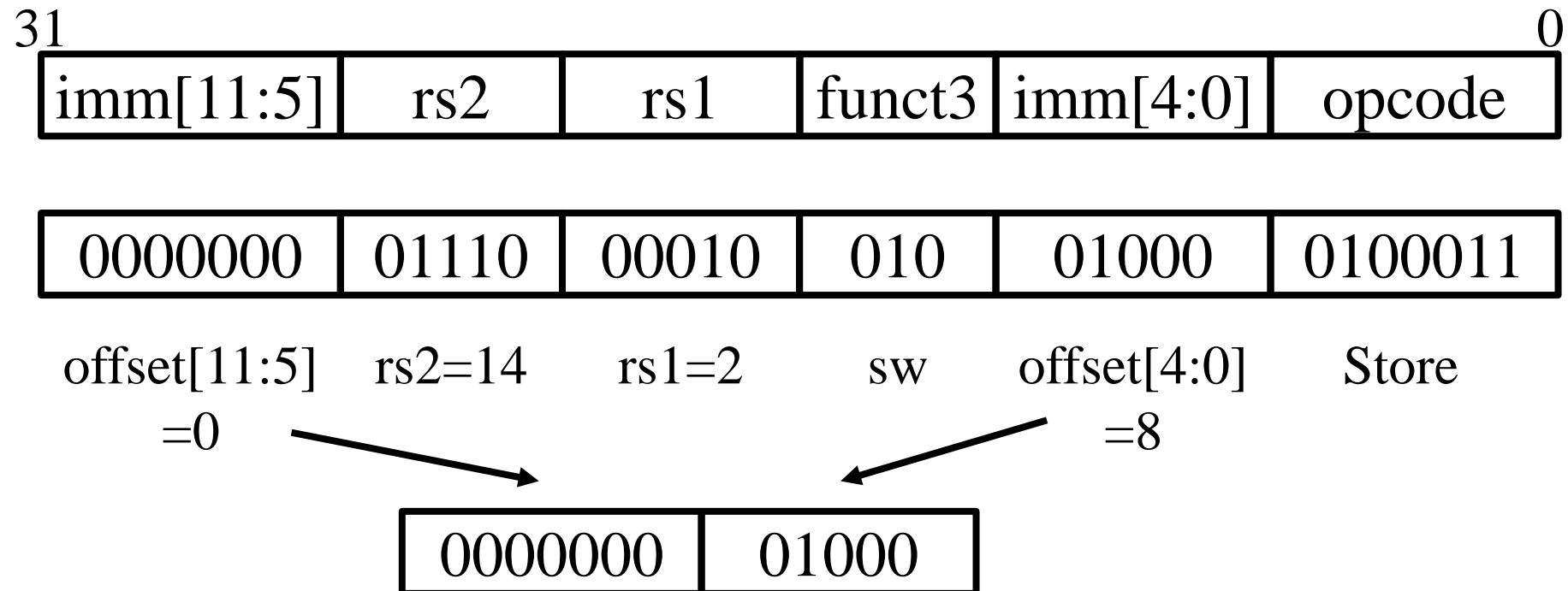
- 存储需要读取两个寄存器，rs1为内存地址，rs2为要写入的数据，以及立即偏移量offset
- 不能将rs2和immediate与其他指令放在同一位置
- S型指令不会将值写入寄存器，所以没有rd
- RISC-V的设计决定是将低位的5位立即数移到其他指令中的rd字段所在的位置——保持rs1/rs2字段在同一位置



# S型指令

- RISC-V汇编S型指令

sw x14, 8(x2)



# S型指令

- 所有RV32 S型指令

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sd

# B型指令

---

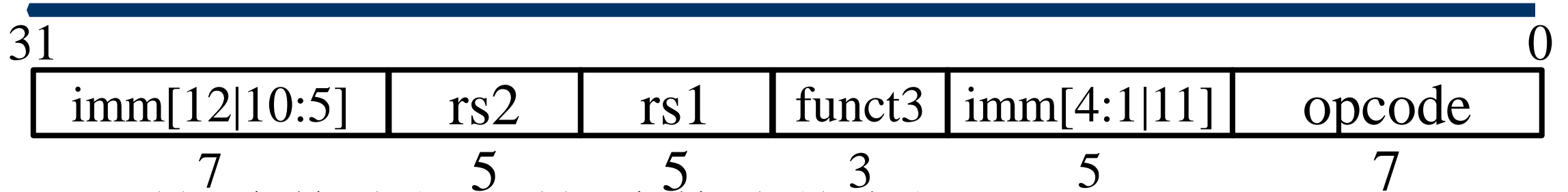
- 条件分支指令
  - `beq x1, x2, Label`
- B型指令读取两个寄存器，但不写回寄存器
  - 类似于S型指令
- 如何对Label进行编码，转移位置如何得到？

# B型指令

---

- 用于条件语句或循环语句（if-else、while、for）
  - 条件或循环体通常很小（<50条指令）
- 指令存储在比较集中的区域
  - 最大分支转移距离与代码量有关
  - 当前指令的地址存储在程序计数器（PC）中

# B型指令



- B型指令格式与S型指令格式基本相同
- PC相对寻址：将立即数字段作为相对PC的补码偏移量
- 立即数字段以2字节为增量表示-4096到+4094的偏移量
- 用12位立即数字段表示13位有符号字节地址的偏移量
  - 偏移量的最低位始终为零，因此无需存储

# B型指令

---

- 为什么不使用字节作为PC的偏移量单位？
  - 因为指令是32位（4字节）
- 将偏移量视为以字为单位，在相对寻址前，先将偏移量乘以4,得到字节偏移量，再加上PC中的基地址
  - 可以访问到相对PC地址前后 $2^{12} \times 32$ 位范围内的所有指令
  - 比使用字节偏移量大四倍的转移范围

# B型指令

---

- 基于RISC-V的扩展指令集支持16位压缩编码指令，也支持16位长度倍数的可变长度指令
- 为了实现对可能的扩展指令的支持，在实际的RISC-V指令设计中，分支转移指令的偏移量都是以2字节为单位
- RISC-V条件分支指令实际上只能访问相对PC地址前后 $2^{11} \times 32$ 位范围内的指令

# B型指令

- RISC-V汇编指令:

- Loop: beq x19, x10, End

- add x18, x18, x10

- addi x19, x19, -1

- j Loop

- End: # 目标指令


???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	Branch



# B型指令

- RISC-V汇编指令:

- Loop:   beq x19, x10, End  
          add x18, x18, x10  
          addi x19, x19, -1  
          j Loop



1 Count  
2 instructions  
3 from branch  
4

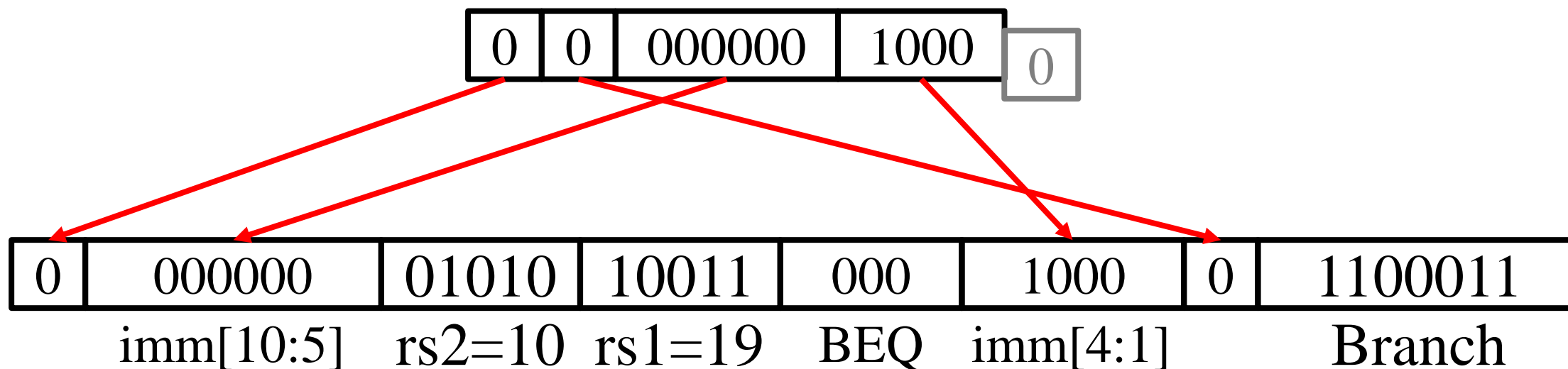
End:   # 目标指令

- $\text{offset} = 4 \times 32\text{-bit instructions} = 16 \text{ bytes} = 8 \times 2 \text{ bytes}$

# B型指令

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	Branch

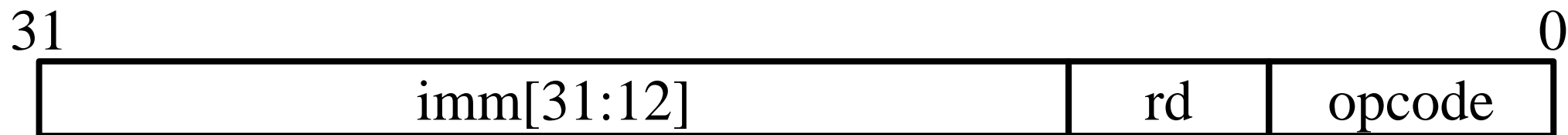
**beq x19, x10, offset =  $8 \times 2$  bytes**



# RISC-V所有B型指令

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU

# U型指令



U-immediate[31:12]

dest

LUI

AUIPC

- ADDI指令中的立即数最大是多少？
- U型指令进行长立即数操作
  - 高20位为长度为20位的立即数字段
  - 5位的目的地寄存器字段
- 两个指令
  - LUI(Load Upper Immediate)——将长立即数写入目的寄存器
  - AUIPC——将PC与长立即数相加结果写入目的寄存器

# U型指令LUI的应用

---

- LUI将长立即数写入目的寄存器的高20位，并清除低12位。
- LUI与ADDI一起可在寄存器中创建任何32位值

LUI      x10, 0x87654      # x10 = 0x87654000

ADDI    x10, x10, 0x321    # x10 = 0x87654321

# U型指令LUI的应用

---

- 如何创建 0xDEADBEEF?

LUI      x10, 0xDEADB    # x10 = 0xDEADB000

ADDI    x10, x10, 0xEEF   # x10 = 0xDEADAEFF?

- ADDI 立即数总是进行符号扩展，如果高位为1，将从高位的20位中减去1

LUI      x10, 0xDEADC    # x10 = 0xDEADC000

ADDI    x10, x10, 0xEEF   # x10 = 0xDEADBEEF

- 伪指令    li x10, 0xDEADBEEF   # 创建两条指令

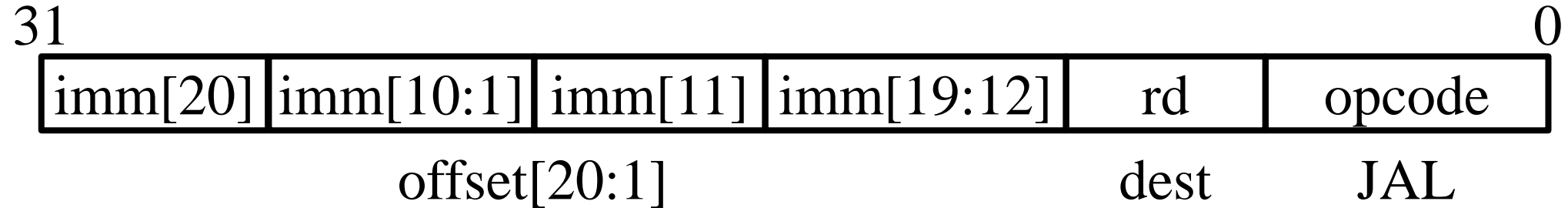
# U型指令

---

- AUIPC (Add Upper Immediate to PC)
  - 将长立即数值加到PC并写入目的寄存器
  - 用于PC相对寻址

Label: AUIPC x10, 0      # 将Label地址放入x10

# J型指令



- JAL将PC+4写入目的寄存器rd中
  - J是伪指令，等同于JAL,但设置rd = x0不保存返回地址
- $PC = PC + offset$ （PC相对寻址）
- 访问相对PC,以2字节为单位的 $\pm 2^{19}$ 范围内的地址空间
  - $\pm 2^{18}$  32-bit指令空间
- 立即数字段编码的优化类似于B型指令，以降低硬件成本



# J型指令

---

- j 伪指令

- j Label = jal x0, Label      # 设置目的寄存器为x0

- jal 指令

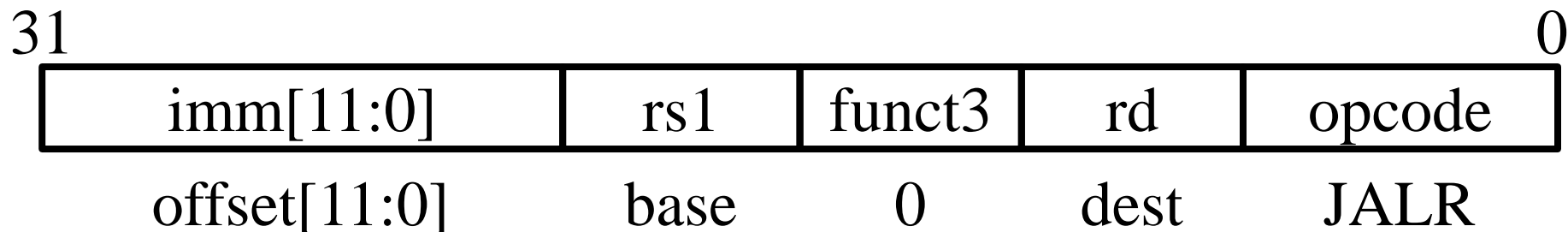
- jal ra, FuncName

# lui,auipc,jalr指令的应用

---

- ret 和 jr 伪指令
  - $\text{ret} = \text{jr ra} = \text{jalr x0}, 0(\text{ra})$
- 对任意32位绝对地址处的函数进行调用
  - $\text{lui x1}, \langle \text{hi20bits} \rangle$
  - $\text{jalr ra}, \text{x1}, \langle \text{lo12bits} \rangle$
- 相对PC地址32位偏移量的相对寻址
  - $\text{auipc x1}, \langle \text{hi20bits} \rangle$
  - $\text{jalr x0}, \text{x1}, \langle \text{lo12bits} \rangle$

# jalr指令



- JALR属于I型指令
- 将 $PC + 4$ 保存在rd中
- 设置  $PC = rs + immediate$
- 与load指令得到地址的方式类似
- 与B型指令和jal不同，不将立即数 $\times 2$ 字节

# RISC-V指令格式

R 型	funct7	rs2	rs1	funct3	rd	opcode
I 型	imm[11:0]		rs1	funct3	rd	opcode
S 型	Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B 型	Imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode
J 型	Imm[20,10:1,11,19:12]				rd	opcode
U 型	Imm[31:12]				rd	opcode

# 第四章 指令系统

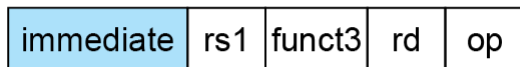
---

- 指令系统概述
- RISC-V指令系统
- RISC-V寻址方式

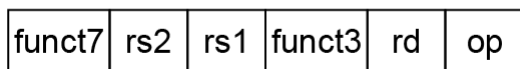


# RISC-V的寻址模式

## 1. Immediate addressing



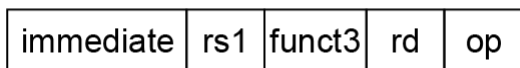
## 2. Register addressing



Registers

Register

## 3. Base addressing



Memory

Register

+

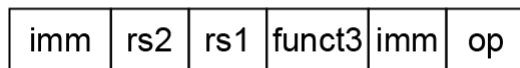
Byte

Halfword

Word

Doubleword

## 4. PC-relative addressing



Memory

PC

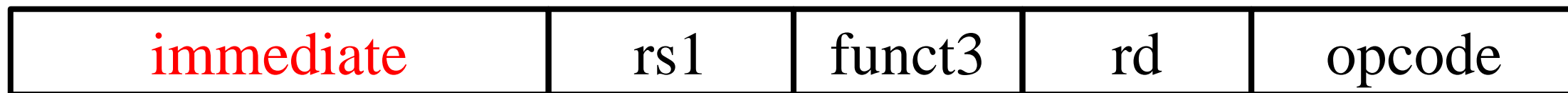
+

Word

# 立即数寻址

---

- 操作数是指令本身的常量
  - `addi x3, x4, 10`
  - `andi x5, x6, 3`

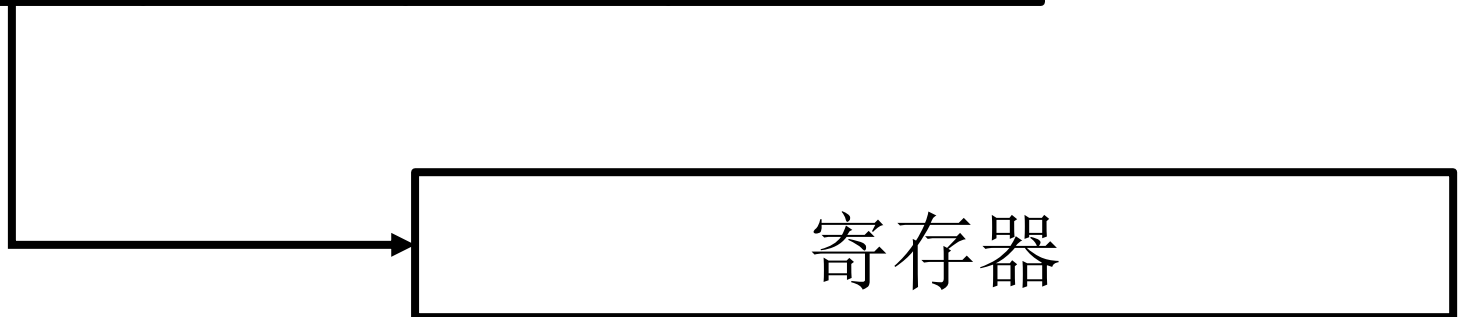
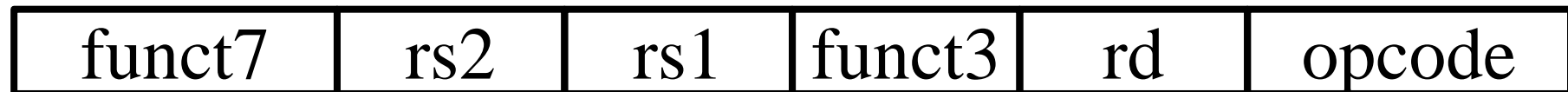


# 寄存器寻址

- 操作数在寄存器中

- add x1, x2, x3

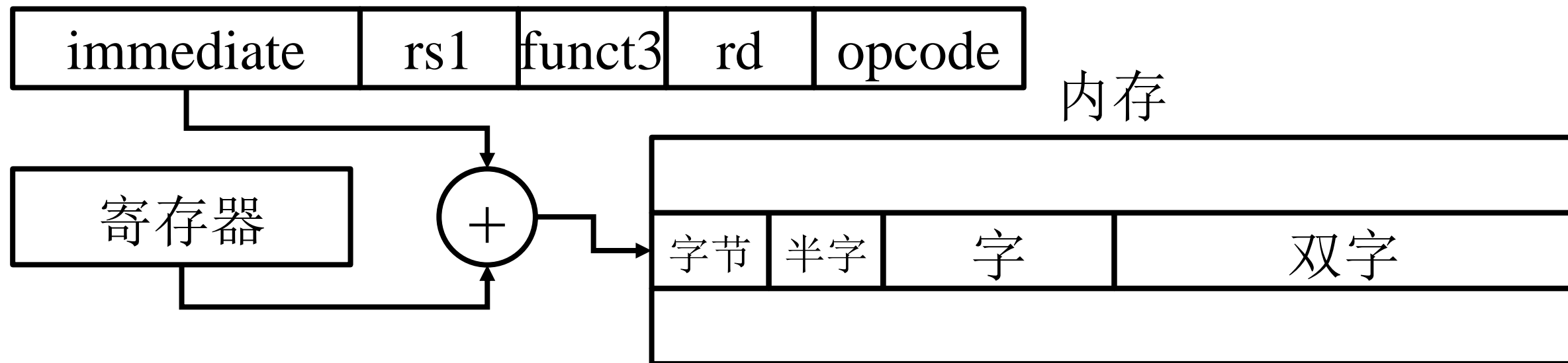
- and x5, x6, x7





# 基址或偏移寻址

- 操作数于内存中，其地址是寄存器和指令中的常量之和
  - `lw x10, 12(x15)`
  - 基址寄存器(x15) + 偏移量(12)



# PC相对寻址

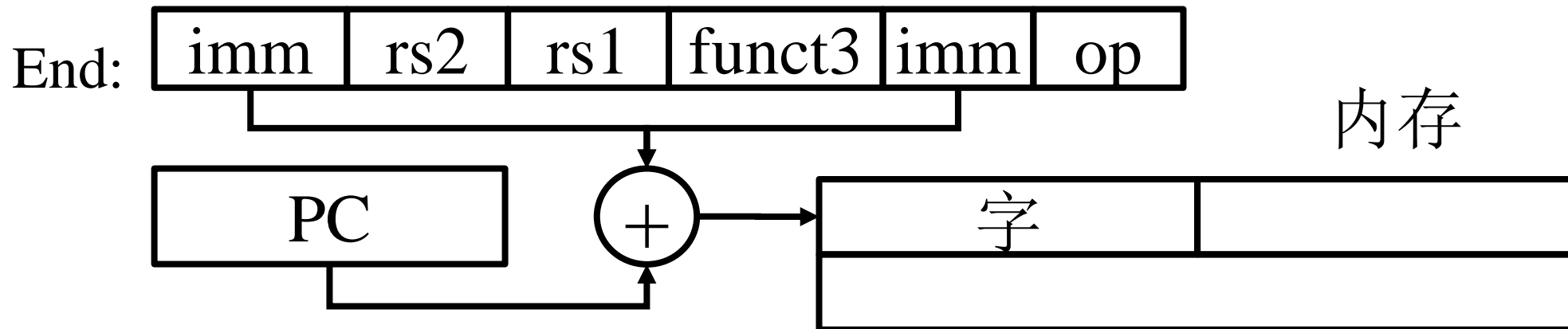
- 分支地址是PC和指令中常量之和

- Loop: beq x19, x10, End

add x18, x18, x10

addi x19, x19, -1

j Loop



# 第四章 指令系统

---

- 指令系统概述
- RISC-V指令系统
- RISC-V寻址方式

