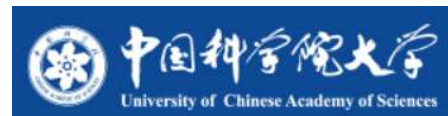


# 操作系统高级教程 ——前置知识

李鹏



中国科学院软件研究所  
Institute of Software, Chinese Academy  
of Sciences



# 开放性问题

- 操作系统与其他软件有什么区别？
- 为什么要学习操作系统？
- 你认为未来的操作系统应如何演化？

# 改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - <https://ipads.se.sjtu.edu.cn/courses/os/index.shtml>
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



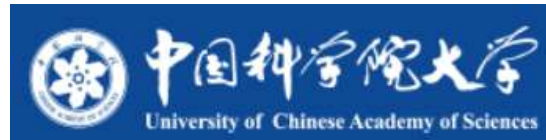
中国科学院软件研究所

Institute of Software, Chinese Academy of Sciences

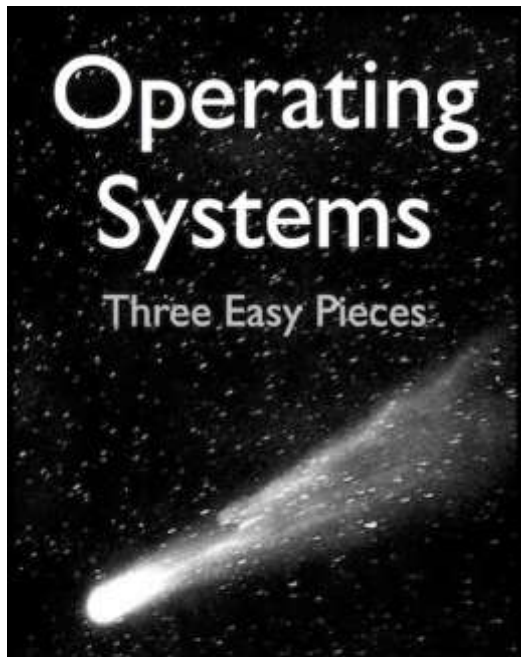


上海交通大学

SHANGHAI JIAO TONG UNIVERSITY



# 图书推荐



[www.ostep.org](http://www.ostep.org)

Remzi H. Arpaci-Dusseau  
Andrea C. Arpaci-Dusseau

# 大纲

- 为什么学习RISC-V ISA / 汇编
- 从C到汇编
- 理解RISC-V汇编
- 理解机器执行

## 为什么学习RISC-V ISA/汇编

# 指令集架构与操作系统

- **ISA (Instruction Set Architecture)**
  - CPU向软件（应用程序和操作系统）提供的接口
    - 理解软件在CPU上的运行（操作系统设计、程序调试等）
  - 操作系统中包含体系结构相关的汇编代码
    - 操作系统启动代码（例如栈尚未设置）
    - 部分操作C语言无法表达（例如获取系统状态、刷新TLB）
    - 部分场景下，汇编代码比C代码高效很多（例如memcpy）

# 为什么选择RISC-V

- **CPU体系结构**

- x86、ARM、RISC-V、SPARC、LoongArch（龙芯）、...

- **RISC-V的优势**

- 开放架构：其指令集和架构规范是公开可用的，没有专利限制
- 低成本和低功耗：RISC-V是精简的指令集架构，可以设计出高性能、低成本和低功耗的处理器
- 生态系统：RISC-V的生态系统正在迅速增长，包括处理器核心、开发工具、操作系统支持和社区支持
- 稳定性：RISC-V架构规范具有一定的稳定性，以确保向后兼容性，从而保护了现有的RISC-V生态系统投资



## 从C语言到汇编语言

# 为什么硬件不能直接运行C语言的源代码

- **硬件设计**

- 高级语言的表达能力很强
- 硬件理解高级语言的复杂度过高、难以高效设计

- **机器指令**

- 格式相对固定
- 功能相对简单
- 二进制编码

# C代码示例

```
long mult2(long a, long b)
{
    return a * b;
}
```

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

# 编译过程



# 从C程序到二进制编码

// C code in mstore.c

```
long mult2(long, long);
```

```
void multstore(long x,  
               long y, long *dest)
```

```
{  
    long t = mult2(x, y);  
    *dest = t;  
}
```

//Binary code in mstore.o

```
41 11 06 e4 22 e0 32 84 97 00 00 00  
e7 80 00 00 08 e0 a2 60 02 64 41 01  
82 80
```

// Assembly file in mstore.s

```
multstore:
```

```
    addi    sp,sp,-16  
    sd      ra,8(sp)  
    sd      s0,0(sp)  
    mv      s0,a2  
    call    mult2  
    sd      a0,0(s0)  
    ld      ra,8(sp)  
    ld      s0,0(sp)  
    addi    sp,sp,16  
    jr      ra
```

//Binary code in mstore

```
41 11 06 e4 22 e0 32 84 ef f0 3f ff  
08 e0 a2 60 02 64 41 01 82 80
```

# 从C程序到二进制编码

```
// C code in mstore.c
long mult2(long, long);

void multstore(long x,
               long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
// Assembly file in mstore.s
multstore:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    mv      s0,a2
    call    mult2
    sd      a0,0(s0)
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```

指令

//Binary code in mstore.o

```
41 11 06 e4 22 e0 32
e7 80 00 00 08 e0 a2
82 80
```

//Binary code in mstore

```
e0 32 84 ef f0 3f ff
64 41 01 82 80
```

机器码很难直接理解

# 从C程序到二进制编码

```
// C code in mstore.c  
long mult2(long, long);
```

```
void multstore(long x,  
               long y, long *dest)  
{  
    long t = mult2(x, y);  
    *dest = t;  
}
```

适合阅读

```
// Assembly file in mstore.s  
multstore:
```

```
    addi    sp,sp,-16  
    sd      ra,8(sp)  
    sd      s0,0(sp)  
    mv      s0,a2  
    call    mult2  
    sd      a0,0(s0)  
    ld      ra,8(sp)  
    ld      s0,0(sp)  
    addi    sp,sp,16  
    jr      ra
```

指令

## AT&T汇编



# AT&T汇编语言格式

- 汇编语言

- 汇编语言是各种CPU提供的机器指令的助记符的集合，使得我们可以用汇编语言直接控制硬件系统进行工作
- 主要有两种形式的汇编语言
  - Intel
    - DOS, Windows采用
  - AT&T
    - Unix, Linux采用

# AT&T汇编语言格式

- 汇编指令操作对象：寄存器、内存地址
- 寄存器
  - 逻辑寄存器
    - 架构相关
    - ISA提供给编译器可见，共16个通用寄存器
  - 物理寄存器
    - 实际上的硬件实现
    - 现代CPU可能有上百个物理寄存器组成

# AT&T汇编语言格式

寄存器	是否有约束	惯例/用途
rax	否	1, 系统调用时, 调用号; 2, 函数返回值; 3, 除法运算中, 存放除数、以及运算结果的商; 4, 乘法运算中, 存放被乘数、以及运算结果;
rbx	是, 被调用者保存	基址寄存器, 用来存储基础访问地址
rcx	否	1, 函数调用时, 第4个参数; 2, 有时用作counter;
rdx	否	1, 函数调用时, 第3个参数; 2, 除法运算中, 存放运算结果的余数; 3, 乘法运算中, 存放运算结果溢出的部分;
rbp	是, 被调用者保存	frame pointer, 存放当前函数调用时栈的基地址
rsp	是, 被调用者保存	时时刻刻指向栈顶
rdi	否	1, 函数调用时, 第1个参数; 2, rep movsb中的目的寄存器;
rsi	否	1, 函数调用时, 第2个参数; 2, rep movsb中的源寄存器;
r8	否	1, 函数调用时, 第5个参数
r9	否	1, 函数调用时, 第6个参数
r10、r11	否	
r12、r13、r14、r15	是, 被调用者保存	

# AT&T汇编语言格式

- X86的向后兼容性

- 64位: rax
- 32位: eax
- 16位: ax



# AT&T汇编语言格式

- **AT&T汇编语法格式**
  - 一条汇编语句包含4个部分
    - 标号(可选)
    - 操作码(指令助记符)
    - 操作数(由具体指令决定)
    - 注释(可选)

标号: 操作码 操作数1, 操作数2, ... #可选的注释

# AT&T汇编语言格式

- 操作码

- AT&T汇编中，操作码名称的最后一个字符指明操作数的宽度

- b, w, l, q分别表示byte(字节), word(字), long(双字), qword (四字)

```
movl var, %eax
```

把内存地址var处的4个字节载入到寄存器eax中

- 操作码前缀

- 用于修饰随后的操作码，常用于重复字符串指令、执行总线锁定操作或指定操作数和地址宽度

- 如

```
movl $0x0, %eax  
movl $0x12, %ecx  
rep stosl
```

# AT&T汇编语言格式

- AT&T汇编语法格式

- 操作数

- 一个指令可以含有0~3个操作数，中间用逗号分隔
    - 对于有两个操作数的指令，前者表示源操作数，后者表示目的操作数

- 如

```
movw %ax, %bx
```

- 操作数可以是立即数、寄存器、内存

- 立即操作数前面需加上\$
      - 寄存器操作数前面需加上%
      - 内存操作数

- » 引用内存地址为 $\text{base} + \text{index} * \text{scale} + \text{disp}$ 的变量

```
disp(base, index, scale)
```

- » 如表示把内存地址var处的内容载入%eax

```
mov var, %eax
```

# AT&T汇编语言格式

- **运算指令**

- `add %rax,%rbx`
- `add $5,%r11`
- `div %r11`
- `inc %r11`
- `mul %r11`

- **拷贝指令**

- `mov %rax,%rbx`
- `mov $22,%rax`
- `mov %r10,(%r11)`
- `mov (%r10),%r11`
- `push %rax`
- `pop %rax`



# AT&T汇编语言格式

- 流程控制指令

- `cmp %rax,%rbx`
- `jmp label` //跳转到标号
- `je label` //如果相等，跳转到标号
- `jne label`
- `jl label`
- `jg label`
- `call label`
- `ret`
- `syscall`

```
1:  Cmpb var,%a1
    je 1f
    movb %a1,%d1
    jmp 1b
1:  pushw %dx
```

# AT&T汇编语言格式

- C语言 内联汇编

```
#include <stdio.h>
int main()
{
    int a = 10, b;
    __asm__( "movl %1, %%eax\n\t"
             "movl %%eax, %0\n\t"
             : "=r"(b) /* output */
             : "r"(a) /* input */
             : "%eax" /* clobbered register */ );
    printf("Result: %d, %d\n", a, b);
    return 0;
}
```

# AT&T汇编语言格式

- 汇编指示符
  - .byte 0x80, 0xca
  - .word 0x800
  - .text
  - .code16

## 理解RISC-V汇编

# 俯瞰指令执行：程序代码在哪

Mulstore程序

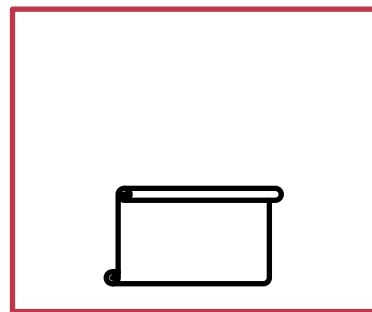
```
41 11 06 e4 22 e0 32 84  
ef f0 3f ff 08 e0 a2 60  
02 64 41 01 82 80
```



CPU



内存



存储（硬盘）

# 俯瞰指令执行：代码加载

问：谁负责加载程序？

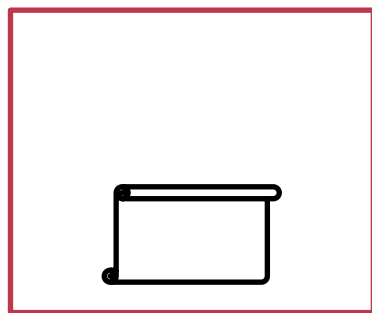
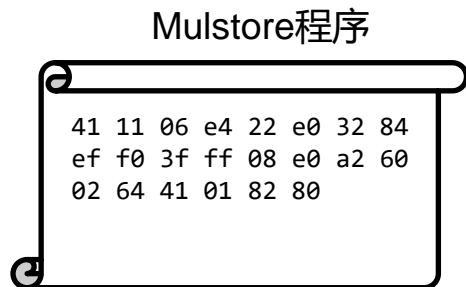
问：谁负责加载OS？



CPU

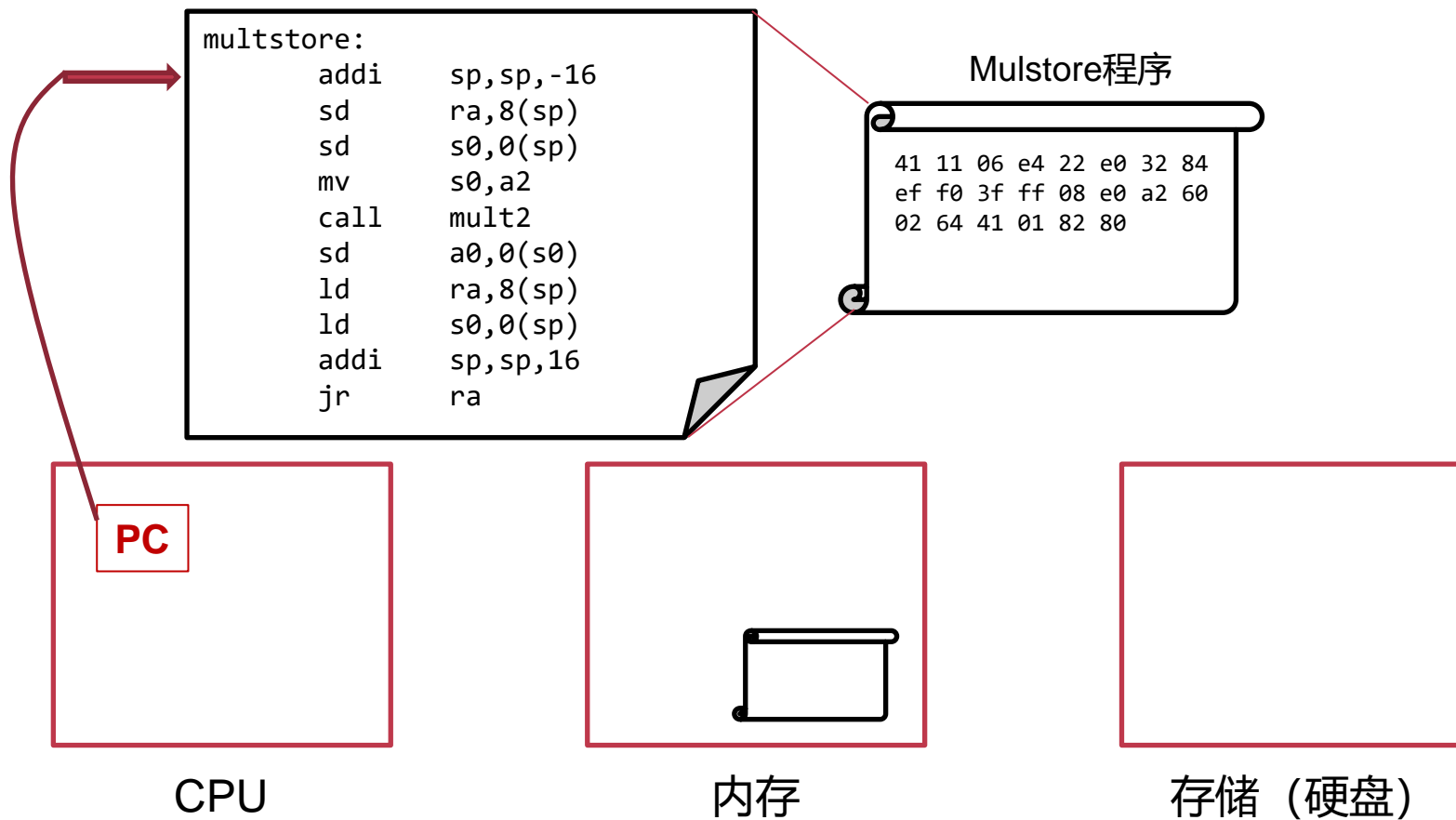


内存

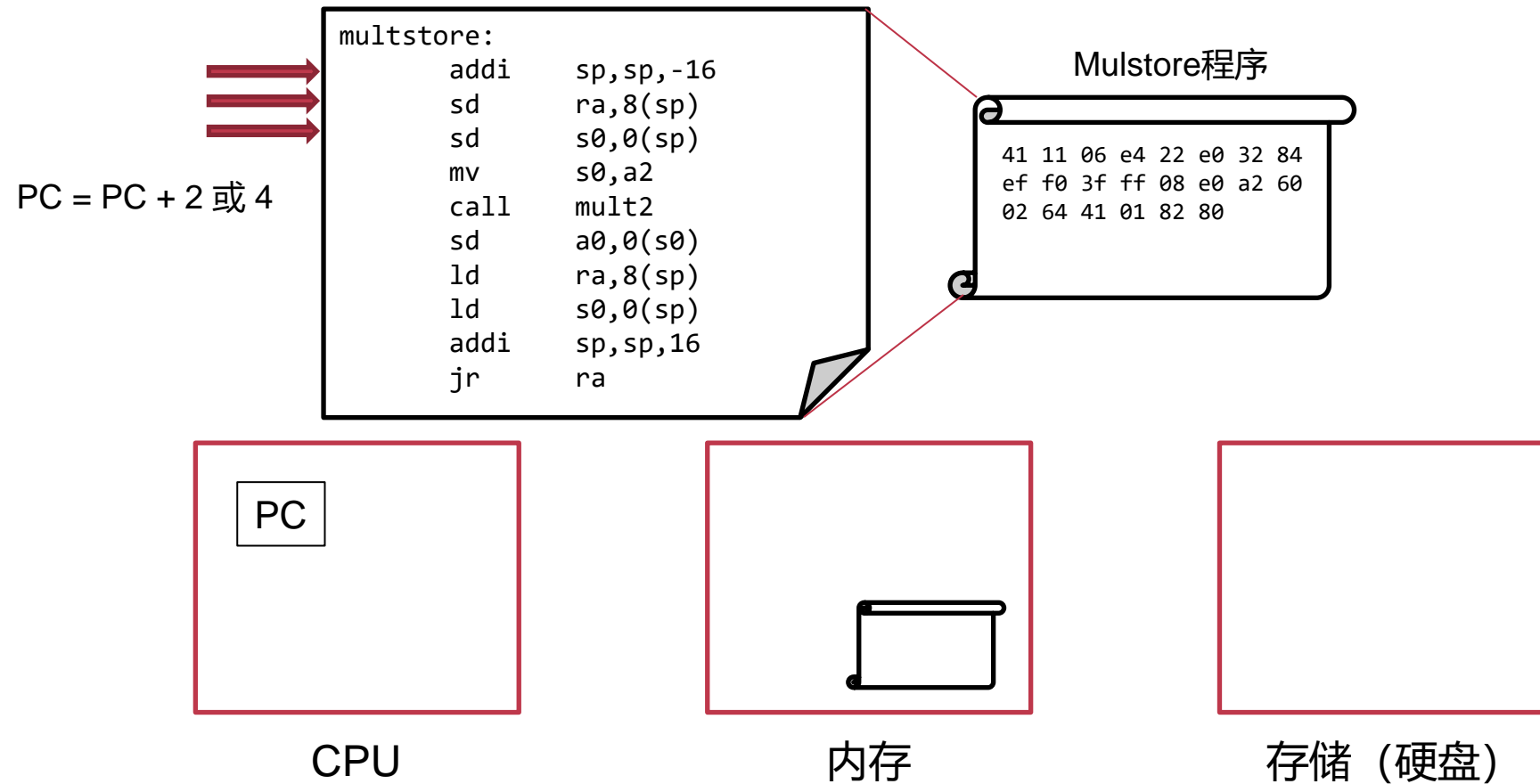


存储（硬盘）

# 俯瞰指令执行：指令位置



# 俯瞰指令执行：更新PC找到下一条指令





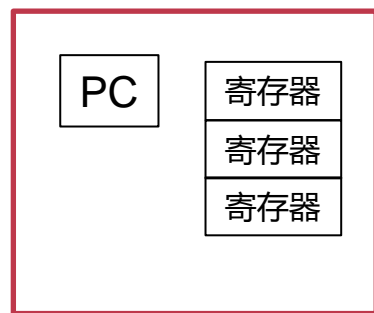
# 俯瞰指令执行：数据在哪

```
multstore:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    mv      s0,a2
    call    mult2
    sd      a0,0(s0)
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```

Mulstore程序

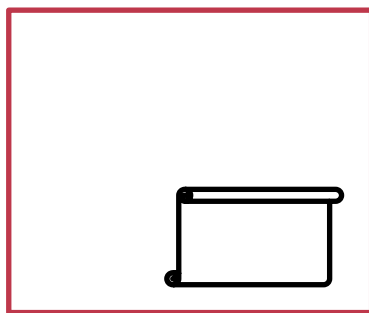
41 11 06 e4 22 e0 32 84  
ef f0 3f ff 08 e0 a2 60  
02 64 41 01 82 80

data



CPU 数据计算

数据  
load/store



内存 数据存放



存储 (硬盘)

## ▶ 算术与逻辑运算指令

# 寄存器：处理器内部的高速存储单元

**x0/zero** 写入忽略，读出永远是 0

Table 1. Integer register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
x0	zero	Zero	— (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	— (Unallocatable)
x4	tp	Thread pointer	— (Unallocatable)
x5 - x7	t0 - t2	Temporary registers	No
x8 - x9	s0 - s1	Callee-saved registers	Yes
x10 - x17	a0 - a7	Argument registers	No
x18 - x27	s2 - s11	Callee-saved registers	Yes
x28 - x31	t3 - t6	Temporary registers	No

**x1 – x31:**

31个64位寄存器

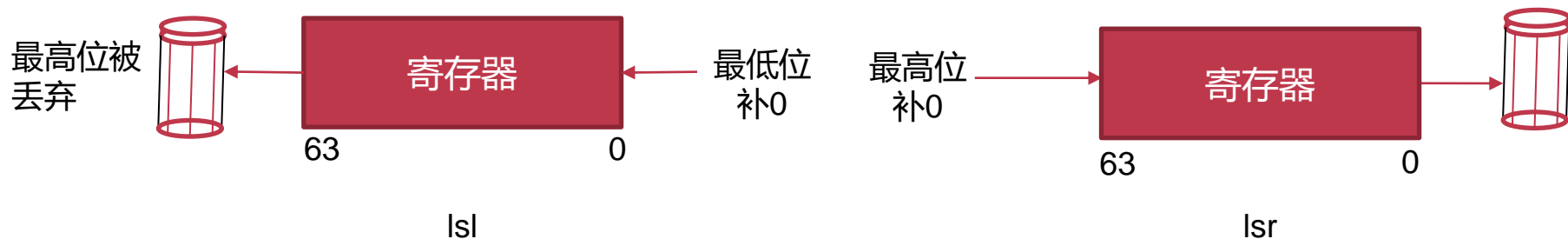
**RISC-V (RV64) 寄存器**

# 算术指令

指令	效果	描述
add rd,rs1,rs2	$rd \leftarrow rs1 + rs2$	加
addi rd,rs1,imm	$rd \leftarrow rs1 + imm$	加立即数
sub rd,rs1,rs2	$rd \leftarrow rs1 - rs2$	减
mul rd,rs1,rs2	$rd \leftarrow rs1 \times rs2$	乘 (mul/mulu)
div rd,rs1,rs2	$rd \leftarrow rs1 \div rs2$	除 (div/divu)

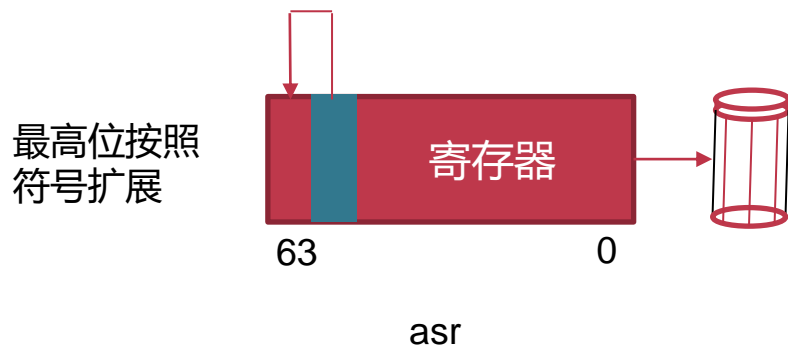
# 移位指令

指令	效果	描述
sll rd,rs1,rs2	$rd \leftarrow rs1 \ll_L rs2$	(Logical) left shift
slli rd,rs1,imm	$rd \leftarrow rs1 \ll_L imm$	(Logical) left shift
srl rd,rs1,rs2	$rd \leftarrow rs1 \gg_L rs2$	Logical right shift
srlr rd,rs1,imm	$rd \leftarrow rs1 \gg_L imm$	Logical right shift



# 移位指令

指令	效果	描述
sll rd,rs1,rs2	$rd \leftarrow rs1 \ll_L rs2$	(Logical) left shift
srl rd,rs1,rs2	$rd \leftarrow rs1 \gg_L rs2$	Logical right shift
sra rd,rs1,rs2	$rd \leftarrow rs1 \gg_A rs2$	Arithmetic right shift
srai rd,rs1,imm	$rd \leftarrow rs1 \gg_A imm$	Arithmetic right shift



# 逻辑运算指令

指令	效果	描述
xor rd,rs1,rs2	$rd \leftarrow rs1 \wedge rs2$	按位异或
xori rd,rs1,imm	$rd \leftarrow rs1 \wedge imm$	按位异或立即数
or rd,rs1,rs2	$rd \leftarrow rs1 \vee rs2$	按位或
ori rd,rs1,imm	$rd \leftarrow rs1 \vee imm$	按位或立即数
and rd,rs1,rs2	$rd \leftarrow rs1 \& rs2$	按位与
andi rd,rs1,imm	$rd \leftarrow rs1 \& imm$	按位与立即数

# 高位指令

指令	效果	描述
lui rd,imm	$rd \leftarrow imm \ll 12$	高位立即数
auipc rd,imm	$rd \leftarrow pc + (imm \ll 12)$	PC 加高位立即数

# 加载大立即数到寄存器

```
lui a0, 0x76543
```

# a0 = 0x76543000

```
addi a0, a0, 0x210
```

# a0 = 0x76543210



# 伪指令：已有指令的多种用法

汇编伪指令	实际指令	描述
neg rd, rs1	sub rd, zero, rs1	取相反数
not rd, rs1	xori rd, rs1, -1	按位取反
mv rd, rs1	addi rd, rs1, 0	寄存器间复制
li rd, imm	(根据 imm 不同)	加载立即数到寄存器

# 练习

寄存器	初始值
a0	0xffffffff00000000
a1	0x1
a2	0x3

指令	目标寄存器	结果
add a2,a2,a1	a2	0x4
addi a1,a1,-1	a1	0x0
mul a0,a1,a2	a0	0x3
xor a2,a1,a0	a2	0xffffffff00000001
srai a0,a0,1	a0	0xffffffff80000000
srli a0,a0,4	a0	0x0fffffffff0000000

add	加
sub	减
mul	乘
div	除
sll	逻辑左移
srl	逻辑右移
sra	算数右移
xor	按位异或
or	按位或
and	按位与

# 算术运算汇编代码

```
long arith(long x, long y, long z)
{
    long q1 = x ^ y;
    long q2 = z * 48;
    long q3 = q1 & 0xF0F0F0F;
    long q4 = q2 - q3;
    return q4;
}
```

```
slli a5, a2, 1      # q2 = z * 2
add a5, a5, a2      # q2 = z * 2 + z = z * 3
slli a5, a5, 4      # q2 = (z * 3) * 16 = z * 48
xor a0, a0, a1      # q1 = x ^ y
li a4, 0xF0F0F0F    # a4 = 0xF0F0F0F
and a0, a0, a4      # q3 = q1 & a4
sub a0, a5, a0      # q4 = q2 - q3
ret                 # Return
```

add	加
sub	减
mul	乘
div	除
sll	逻辑左移
srl	逻辑右移
sra	算数右移
xor	按位异或
or	按位或
and	按位与

初始时，寄存器 a0、a1、a2 分别对应变量 x、y、z

## 32位运算指令

- 部分指令有 32 位版本
  - `addiw`, `slliw`, `srliw`, `sraiw`
  - `addw`, `subw`, `sllw`, `srlw`, `sraw`
- 功能与不带w后缀的指令相同，但只对操作数低32位运算，结果存入寄存器时高32位为符号扩展

# 访存指令

# 回顾

```
multstore:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    mv      s0,a2
    call    mult2
    sd      a0,0(s0)
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```

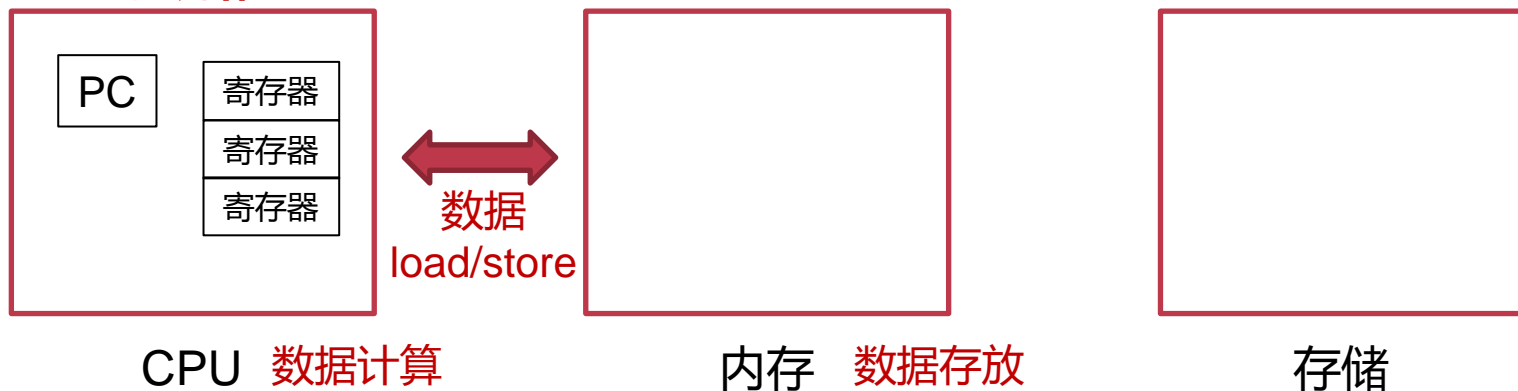
Mulstore程序

41 11 06 e4 22 e0 32 84  
ef f0 3f ff 08 e0 a2 60  
02 64 41 01 82 80

data

计算指令  
mv指令

操作CPU内部的数据



# 访存指令

指令	效果	描述
lb rd, offset(rs1)	$Rd \leftarrow \text{mem}[x[rs1] + \text{offset}]$	从内存地址 $x[rs1] + \text{offset}$ 加载byte 大小的数据到寄存器
sb rd, offset(rs1)	$\text{mem}[x[rs1] + \text{offset}] \leftarrow Rd$	从寄存器rd中将数据写入 地址为 $x[rs1] + \text{offset}$ 的 内存位置

lb也可以是lw/ld/lh/lhu/lbu

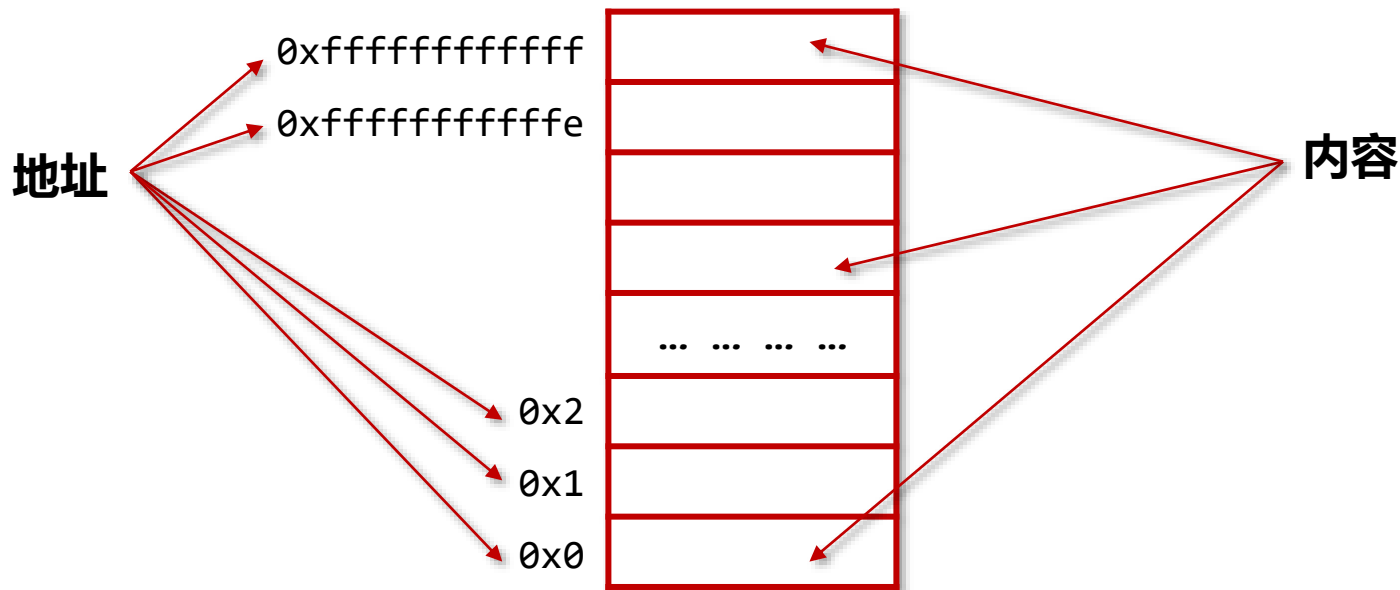
b: byte字节, w: word字, h: halfword半字, u: unsigned无符号

sb也可以是sw/sh

offset为符号位扩展的12位立即数

# 处理器视角下的内存

- 内存可以被视为一个很大的**字节数组**
- 数组中每个元素可以由唯一的地址来索引





# 内存地址

- 将“内存数组”的名称记为 **M**
  - 若 **addr** 为要访问的内存地址,  
**M[addr]** 即为由 **addr** 开始的内存单元的内容
    - **addr** 在这里被用作“内存数组”的索引
  - 内存单元的大小由上下文决定
- **addr** 的具体格式由 **寻址模式** 决定

# 基址址加偏移量模式

- 引用  $M[r_b, \text{Offset}]$  处的数据
- 基址址寄存器  $r_b$
- RISC-V 中偏移量 **Offset** 通常是一个立即数

如: `lw x1, 100(x2)` # 加载寄存器 x1 中的值, 该值位于基址址寄存器 x2 加上立即数 100 的内存位置

如果需要使用寄存器中的值作为偏移量, 加载或计算偏移量, 然后将其与基址址相加。

如: `add x4, x2, x3` # 计算偏移量

`lw x1, 0(x4)` # 使用计算得到的偏移量加载数据到 x1

加载寄存器 x1 中的值, 该值位于基址址寄存器 x2 加上偏移量寄存器 x3 的内存位置

# 示例：基址加偏移量模式

`long E[6] ;`

0	8	16	24	32	40
---	---	----	----	----	----

- 假设**E**是一个整型数组
  - E的起始地址存放在s2寄存器中
- 访问数组元素E[1]的RISC-V汇编为：
  - `lb s1,8(s2)`

# 练习题

内存地址	值
0x100	0xFF
0x108	0xAB

寄存器	值
X0	0x100

操作	(地址) 值
X0	0x100
0(X0)	0xFF
8(X0)	0xAB