IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

ARM汇编 – 基础

上海交通大学

https://www.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

大纲

- 为什么学习ARM ISA/汇编
- 从C语言到汇编语言
- 理解ARM汇编
- 理解机器执行

为什么学习ARM 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、内存安全等）
  - 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）
    - 有场景汇编代码比C代码高效很多（例如memcpy）

两个简单示例

```c
static void target(void)
{
  printf("Exploit succeeded!\n");
  exit(0);
}

static void vulnerable(void)
{
    uint64_t a[1];
    a[2] = (uint64_t)target;
}

int main()
{
    vulnerable();
    return 0;
}
```

ROP

Memory Copy

```c
void my_memcpy_c(void* dest, void* src, size_t n) {
    clock_t start = clock();

    unsigned char* d = (unsigned char*)dest;
    unsigned char* s = (unsigned char*)src;

    for (size_t i = 0; i < n; i++) {
        d[i] = s[i];
    }

    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("C implementation time taken: %f seconds\n", time_taken);
}

void my_memcpy_asm(void* dest, void* src, size_t n) {
    clock_t start = clock();

    asm volatile(
        "cld\n\t"            // 清除方向标志位，向前移动
        "rep movsb\n\t"      // 重复执行movsb指令，将n个字节从src复制到de
    :
    : "c"(n), "S"(src), "D"(dest)
    : "memory"
    );

    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Assembly implementation time taken: %f seconds\n", time_taken
}
```

为什么选择ARM

- **CPU体系结构**
  - x86、ARM、RISC-V、SPARC、LoongArch、…
- **ARM的应用**
  - 终端：手机、平台、智能终端、边缘设备
  - 车载：智能座舱、自动驾驶计算平台
  - 笔记本：Apple
  - 服务器：鲲鹏、亚马逊Graviton、Ampere、NVIDIA Grace
  - 航天：NASA

从C语言到汇编语言

为什么硬件不能直接运行高级语言的源代码

- **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


mstore.c
源程序（文本）
预处理器 (cpp)
mstore.i
经预处理的源程序（文本）
编译器 (cc1)
mstore.s
汇编程序（文本）
汇编器 (as)
mstore.o
可重定向目标程序（二进制）
链接器 (ld)
mstore
可执行目标程序（二进制）

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:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

```
//Binary code in mstore.o
fd 7b be a9 fd 03 00 91 f3 0b 00 f9
f3 03 02 aa 00 00 00 94 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8 c0 03 5f d6
```

```
//Binary code in mstore
fd 7b be a9 fd 03 00 91 f3 0b 00 f9
f3 03 02 aa f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8 c0 03 5f d6
```

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:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

指令

```
//Binary code in mstore.o
fd 7b be a9 fd 03 0
f3 03 02 aa 00 00 0
f3 0b 40 f9 fd 7b c2 a8 c0 03 5f d6
```

```
//Binary code in mstore
d 03 00 91 f3 0b 00 f9
0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8 c0 03 5f d6
```

机器码不能直接阅读

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:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

指令

理解ARM汇编

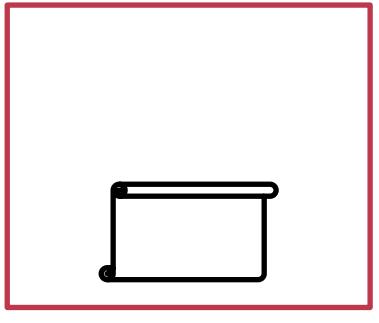
俯瞰指令执行：程序代码在哪

Mulstore程序

```
fd 7b be a9 fd 03 00 91
f3 0b 00 f9 f3 03 02 aa
f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8
c0 03 5f d6
```

CPU

内存

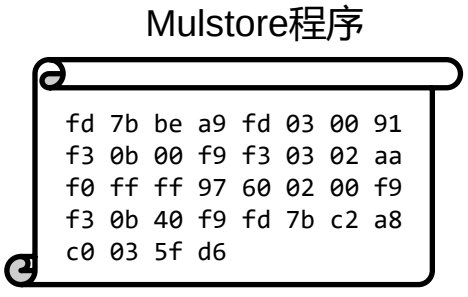


存储（硬盘）

俯瞰指令执行：代码加载

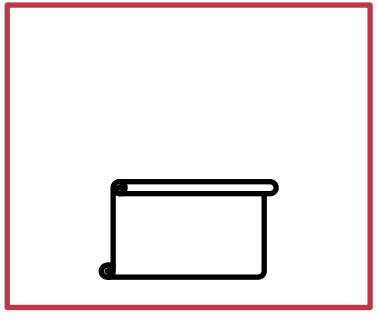
问：谁负责加载程序?

问：谁负责加载OS?



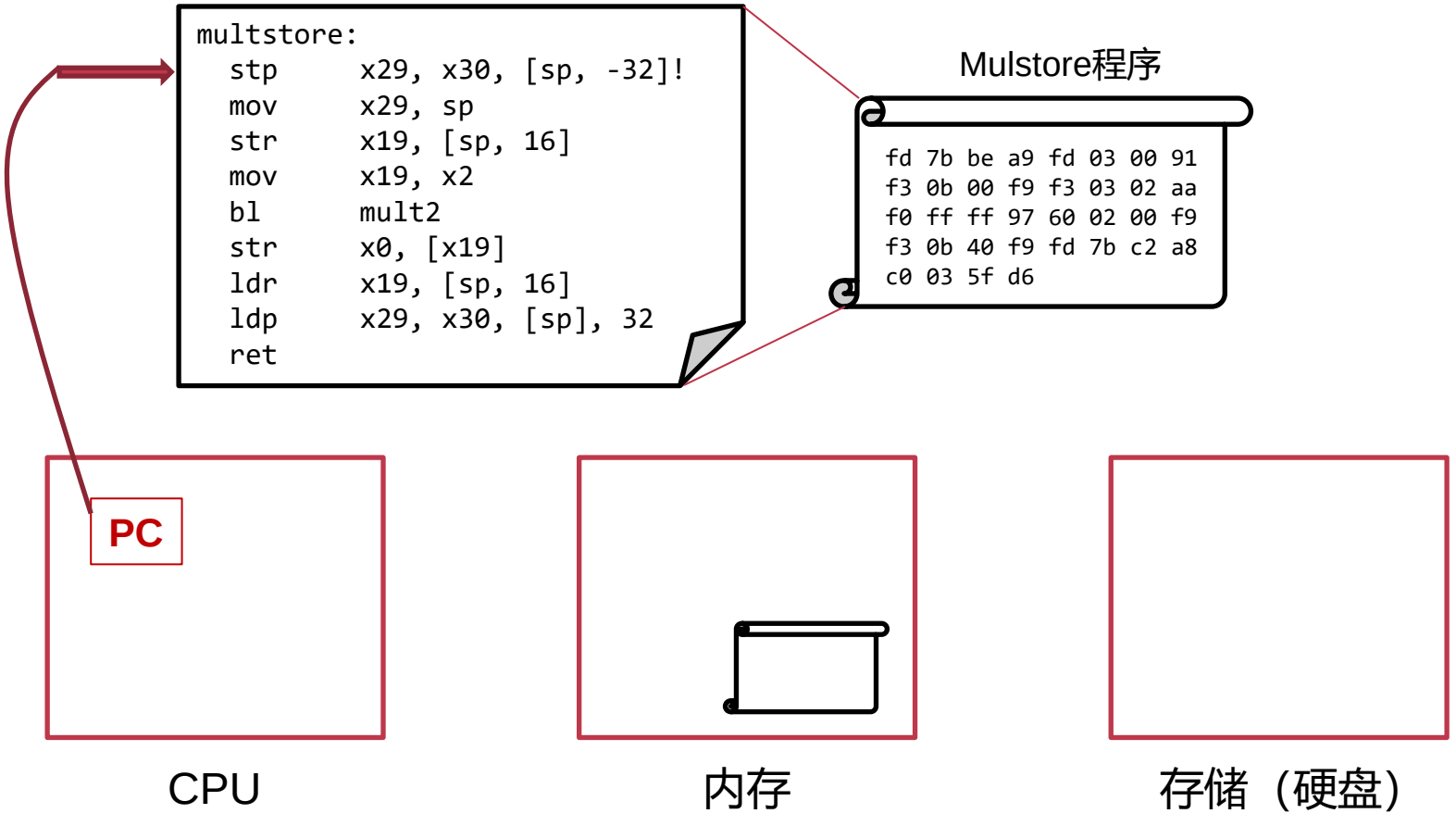
CPU

内存



存储（硬盘）

俯瞰指令执行：指令位置


multstore:
stp x29, x30, [sp, -32]!
mov x29, sp
str x19, [sp, 16]
mov x19, x2
bl mult2
str x0, [x19]
ldr x19, [sp, 16]
ldp x29, x30, [sp], 32
ret
Mulstore程序
fd 7b be a9 fd 03 00 91
f3 0b 00 f9 f3 03 02 aa
f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8
c0 03 5f d6
PC
CPU
内存
存储（硬盘）

俯瞰指令执行：更新PC找到下一条指令

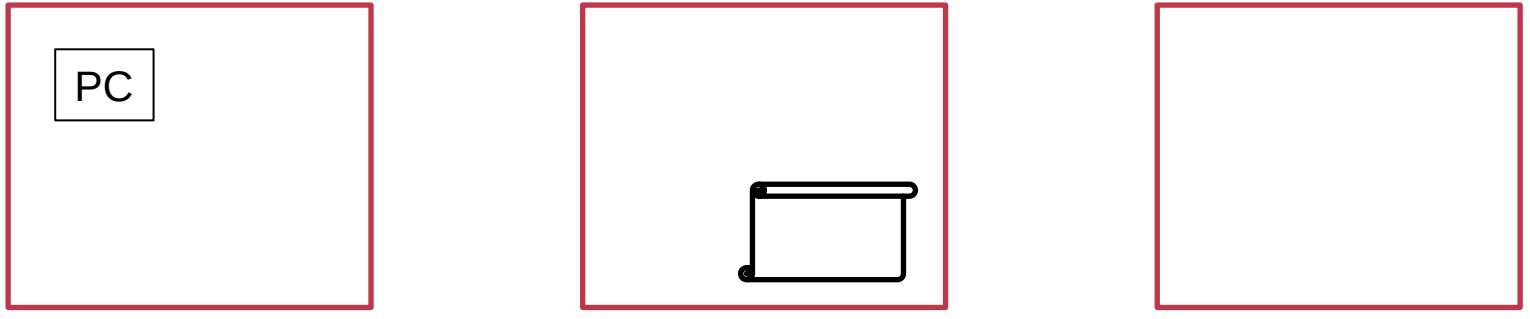
PC = PC +4

指令定长

```
multstore:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

Mulstore程序

```
fd 7b be a9 fd 03 00 91
f3 0b 00 f9 f3 03 02 aa
f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8
c0 03 5f d6
```



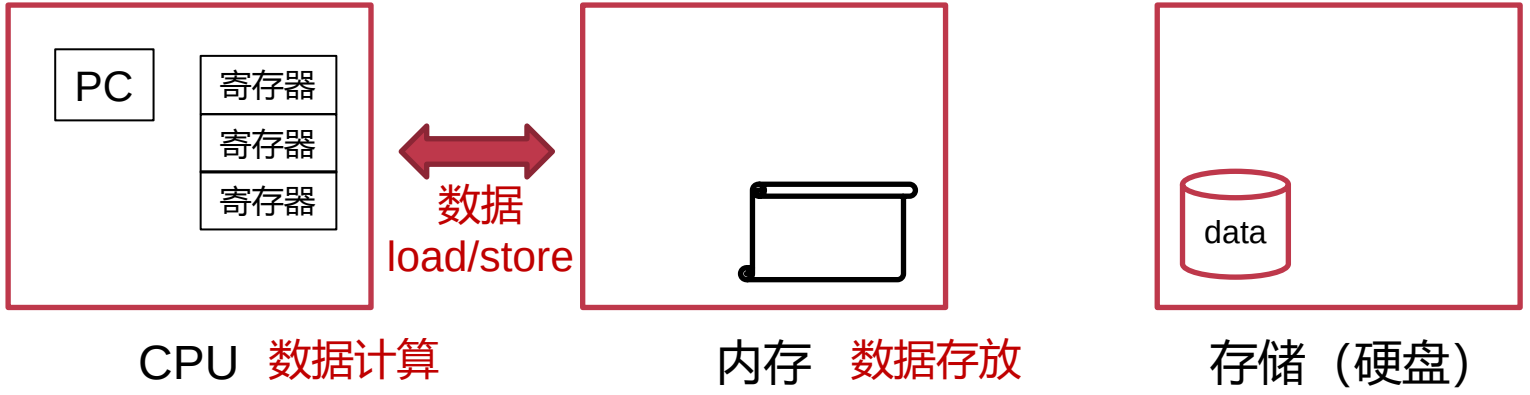
俯瞰指令执行：数据在哪

```
multstore:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

Mulstore程序

```
fd 7b be a9 fd 03 00 91
f3 0b 00 f9 f3 03 02 aa
f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8
c0 03 5f d6
```

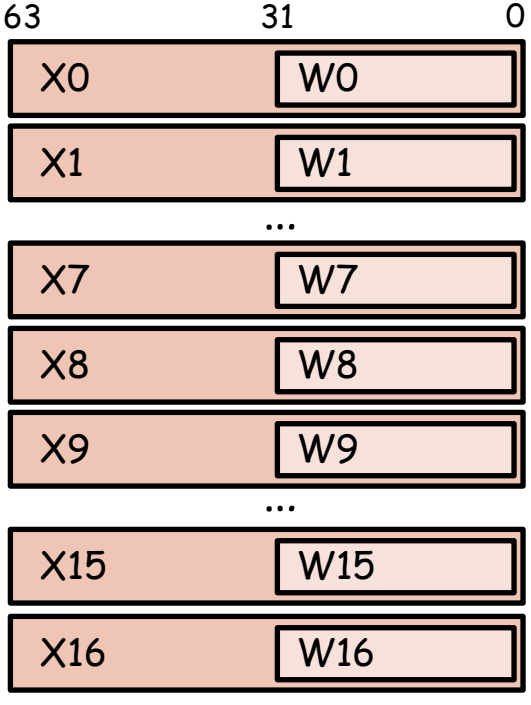
data

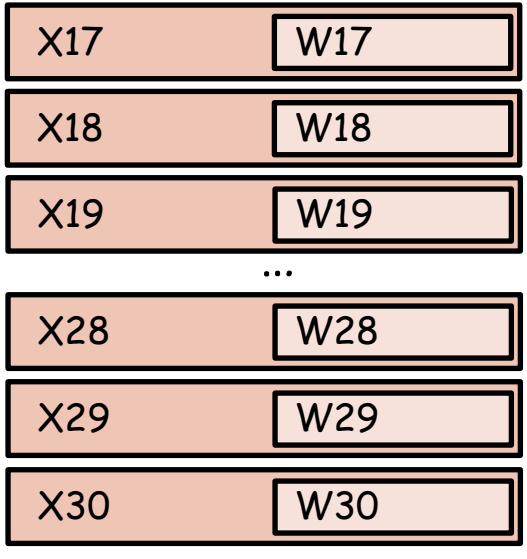


寄存器数据搬移指令

CPU中的寄存器（ARMv8）

- 31个64位通用寄存器：处理器内部的高速存储单元





寄存器间的数据搬移指令

- **指令格式：mov dst, src**
  - 源操作数**src**可以是：
    - 立即数
    - 寄存器
  - 目的操作数**dst**必须是寄存器

示例：

```
mov    w0, #1

mov    x0, #1

mov    x0, w1

mov    x0, x1
```

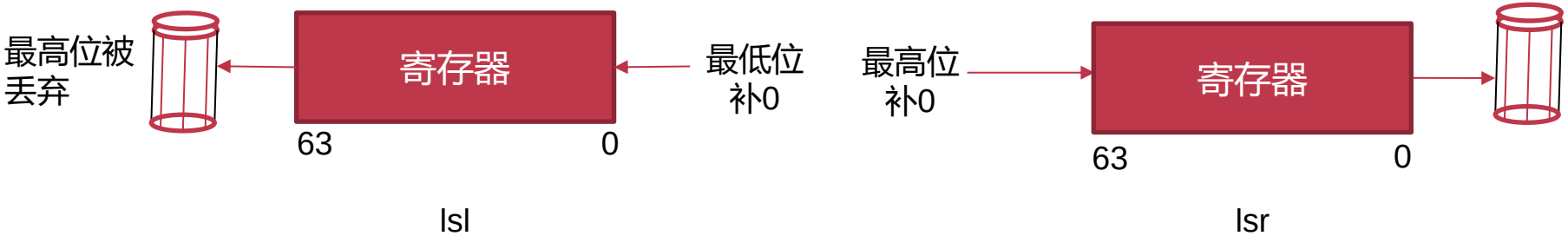
算术与逻辑运算指令

算术指令

指令	效果	描述
`add Rd,Rn,Op2`	Rd ← Rn + Op2	加
`sub Rd,Rn,Op2`	Rd ← Rn - Op2	减
`mul Rd,Rn,Op2`	Rd ← Rn × Op2	乘 (smul/umul)
`div Rd,Rn,Op2`	Rd ← Rn ÷ Op2	除 (sdiv/udiv)
`neg Rd,Rn`	Rd ← -Rn	取相反数

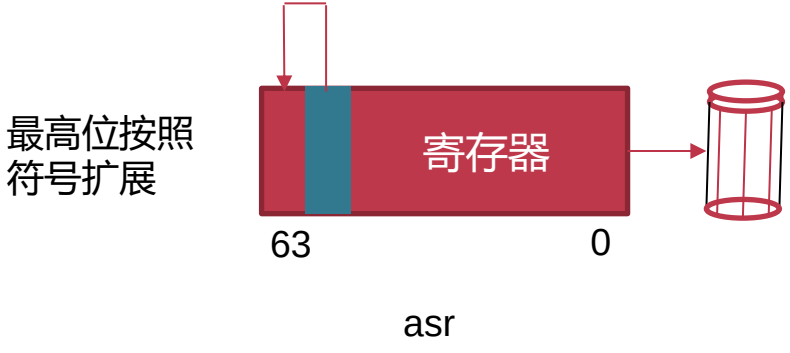
移位指令

指令	效果	描述
`lsl Rd,Rn,Op2`	`Rd ← Rn << Op2`	Logical left shift
`lsr Rd,Rn,Op2`	`Rd ← Rn >>L Op2`	Logical right shift



移位指令

指令	效果	描述
`lsl Rd,Rn,Op2`	Rd ← Rn << Op2	Logical left shift
`lsr Rd,Rn,Op2`	Rd ← Rn $>>_L$ Op2	Logical right shift
`asr Rd,Rn,Op2`	Rd ← Rn $>>_A$ Op2	Arithmetic right shift
`ror Rd,Rn,Op2`	Rd ← Rn $>>_R$ Op2	Rotate right



逻辑运算指令

指令	效果	描述
eor Rd,Rn,Op2	Rd ← Rn ^ Op2	按位异或
orr Rd,Rn,Op2	Rd ← Rn | Op2	按位或
and Rd,Rn,Op2	Rd ← Rn & Op2	按位与
mvn Rd,Rn	Rd ← ~ Rn	按位取反

算术运算汇编代码

```
int arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

```
eor     x0, x0, x1              # t1 = x ^ y
add     x2, x2, x2, lsl #1      # z = z * 3
lsl     x2, x2, 4               # t2 = z * 16
mov     w1, #0x0F0F0F0F         # tmp = 0x0F0F0F0F
and     x0, x0, x1              # t3 = t1 & const
sub     w0, w2, w0              # t4 = t2 - t3
ret                             # return t4
```

add	加
sub	减
mul	乘
div	除
neg	取反
lsl	逻辑左移
lsr	逻辑右移
asr	算数右移
ror	循环右移
eor	按位异或
orr	按位或
and	按位与
mvn	按位取反

初始时，寄存器 x0、x1、x2 分别对应变量 x、y、z

Modified Register：修改过的寄存器

- `add  x2, x2, x2, lsl #1`
  - 减少指令数量
  - 减少用于存放中间结果的寄存器占用量

Modified Register的常见用法

- **对操作数进行移位**
  - 例子： eor w0, w8, w8, asr #16

C

```
w0 = ((w8 >> 16) ^ w8);
```

- **对操作数进行位扩展**
  - 例子： add x19, x19, w0, sxtw
  - 无符号扩展: uxtb, uxth, uxtw
  - 符号扩展: sxtb, sxth, sxtw

C

```
long x19; int w0; x19 += w0;
```

算术运算汇编代码

```
int arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

```
eor     x0, x0, x1              # t1 = x ^ y
add     x2, x2, x2, lsl #1      # z = z * 3
lsl     x2, x2, 4               # t2 = z * 16
mov     w1, #0x0F0F0F0F         # tmp = 0x0F0F0F0F
and     x0, x0, x1              # t3 = t1 & const
sub     w0, w2, w0              # t4 = t2 - t3
ret                             # return t4
```

add	加
sub	减
mul	乘
div	除
neg	取反
lsl	逻辑左移
lsr	逻辑右移
asr	算数右移
ror	循环右移
eor	按位异或
orr	按位或
and	按位与
mvn	按位取反

初始时，寄存器 x0、x1、x2 分别对应变量 x、y、z

访存指令

回顾

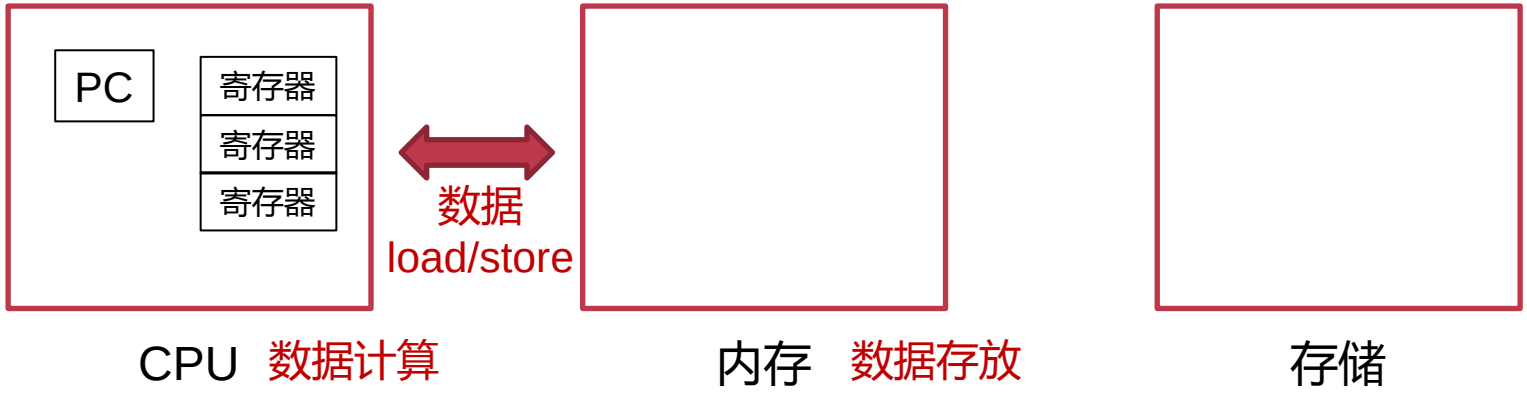
```
multstore:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     x19, x2
  bl      mult2
  str     x0, [x19]
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

Mulstore程序

```
fd 7b be a9 fd 03 00 91
f3 0b 00 f9 f3 03 02 aa
f0 ff ff 97 60 02 00 f9
f3 0b 40 f9 fd 7b c2 a8
c0 03 5f d6
```

data

计算指令
mov指令
操作CPU内部的数据



访存指令

指令	效果	描述
`ldr R,addr`	R ← mem[addr : addr+R_s]	从内存加载数据到寄存器
`str R,addr`	mem[addr : addr+R_s] ← R	把寄存器中数据写到内存

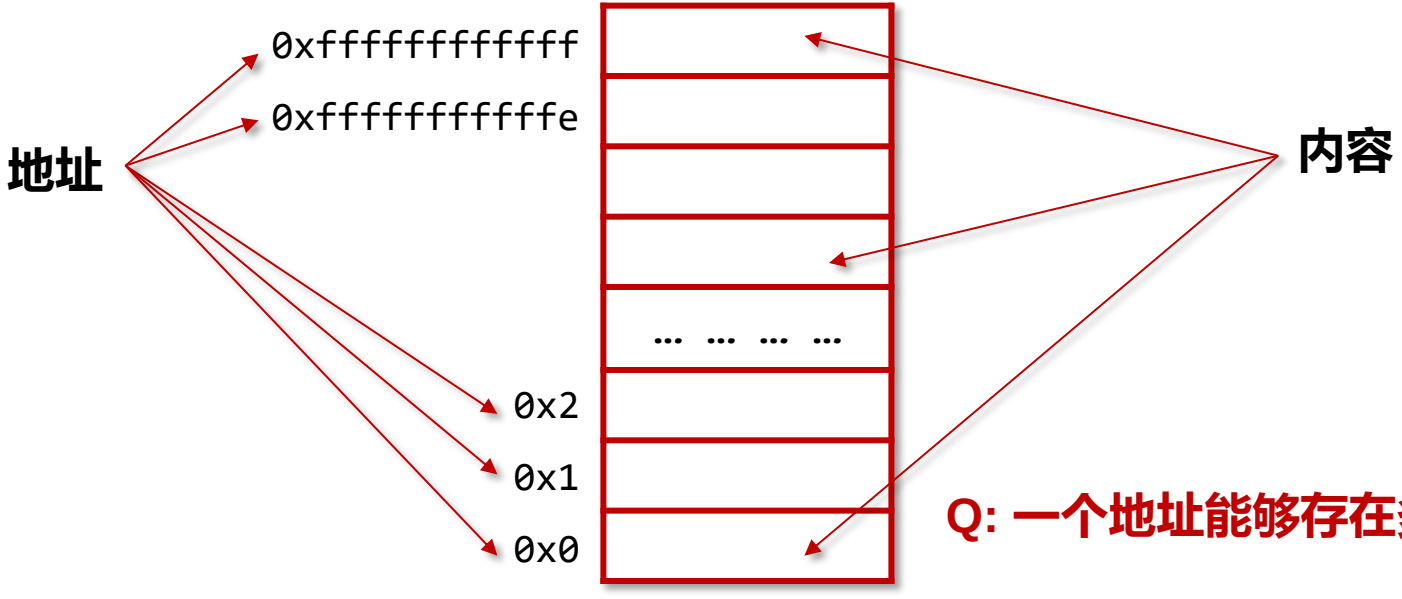
mem[a : b] 指地址 a 到地址 b 的内存范围

load/store操作的内存大小由寄存器大小决定:

R_s指寄存器的大小（字节数）

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



Q: 一个地址能够存在多大的数据量?

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在这里被用作数组索引
  - 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**

寻址模式

- **寻址模式是表示内存地址的表达式**
  - 基地址模式（索引寻址）
    - $[r_b]$
  - 基地址加偏移量模式（偏移量寻址）
    - $[r_b, offset]$

示例：基地址模式

```
1    void swap(int* a, int* b)
2    {
3      int temp = *a;
4      *a = *b;
5      *b = temp;
6    }
```

初始时x0、x1

分别对应a、b

```
1  swap:
2    ldr      w3, [x1]
3    ldr      w2, [x0]
4    str      w3, [x0]
5    str      w2, [x1]
6    ret
```

基地址加偏移量模式

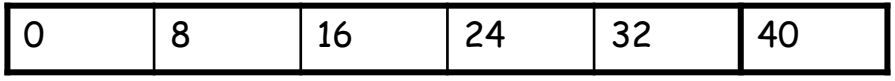
- **引用M[r_b, Offset]处的数据**
- **基地址寄存器r_b：64位通用寄存器**
- **偏移量Offset可以是下列选项之一**
  1. 立即数 **#imm**
  2. 64位通用寄存器 **r_i**
  3. 修改过的寄存器 **r_m, op**，在这里**op**可以是
     - 移位运算: **lsl #3**
     - 位扩展: **sxtw**

```
ldr w8, [x0, #8]
ldr w9, [x1, x0]
str w9, [x0, x0, lsl #2]
str w9, [x0, w0, sxtw]
```

示例：基地址加偏移量模式

- **假设E是一个长整型数组**

```
long E[6] ;
```



  - E的起始地址存放在x0寄存器中
  - E的索引 **i** 存放在x1寄存器中

- **那么数组元素E[i]的访问在ARM汇编中为：**
  - **ldr x2, [x0, x1, lsl #3]**

寻址模式

- **寻址模式是表示内存地址的表达式**
  - 基地址模式（索引寻址）
    - $[r_b]$
  - 基地址加偏移量模式（偏移量寻址）
    - $[r_b, offset]$
  - 前索引寻址（寻址操作前更新基地址）
    - $[r_b, offset]!$ r_b += Offset; **寻址**M$[r_b]$
  - 后索引寻址（寻址操作后更新基地址）
    - $[r_b]$, offset **寻址**M$[r_b]$; r_b += Offset

练习题

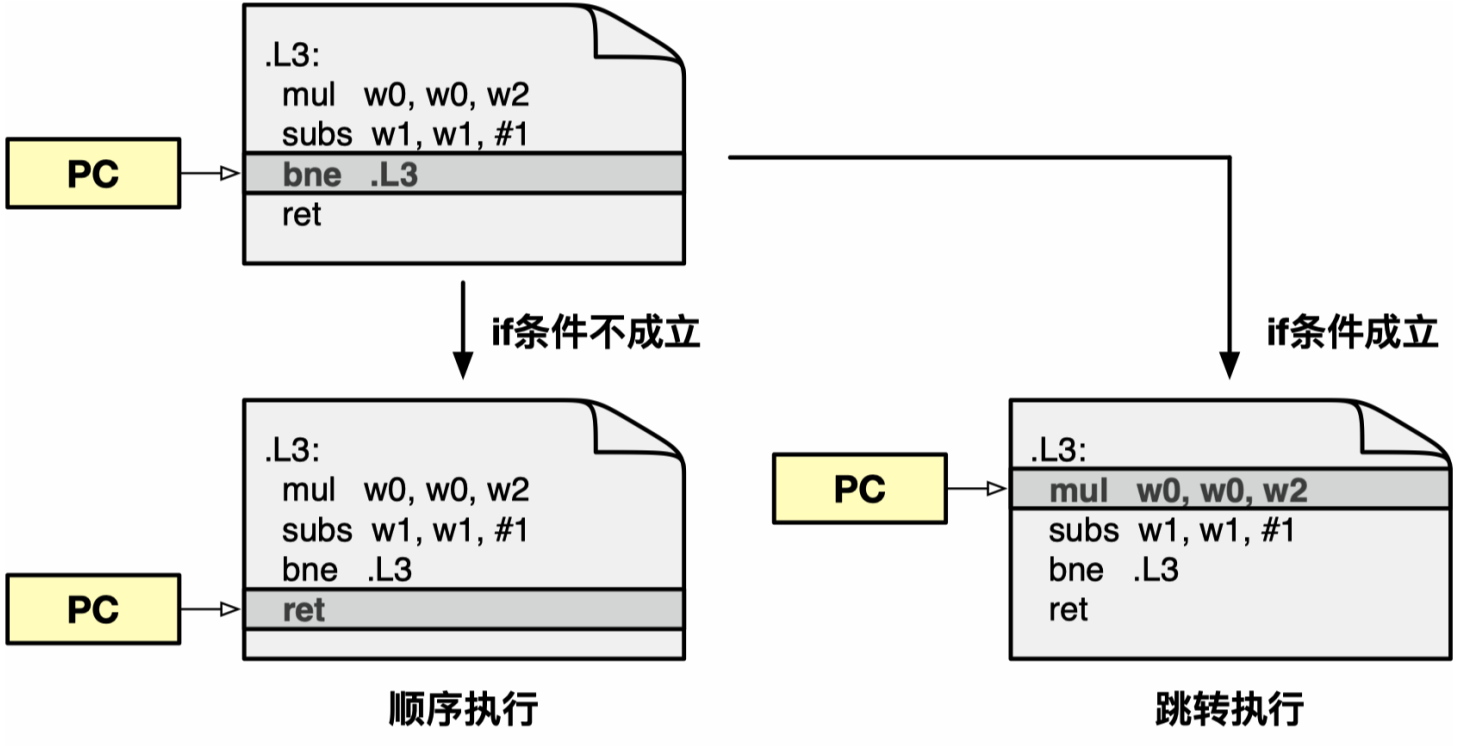
内存地址	值
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

寄存器	值
X0	0x100
X1	0x8
X2	0x2

操作	（地址）值
X0	0x100
[X0]	0xFF
[X0, X1]	0xAB
[X0, X2, lsl #3]	(0x110) 0x13
[X0, #0x18]!	(0x118) 0x11
[X0], #0x18	(0x100) 0xFF

条件码

PC更新：顺序执行和跳转执行



控制流跳转：标签与分支指令

```c
int power(int x, unsigned int n)
{
  int result = 1;
  for (unsigned int i = n; i > 0; i--)
    result *= x;
  return result;
}
```

```
power:
  mov     w2, w0
  mov     w0, 1
  cbz     w1, .L1
.L3:
  mul     w0, w0, w2
  subs    w1, w1, #1
  bne     .L3
.L1:
  ret
```

Label仅存在于汇编中（方便阅读），
二进制代码中没有

控制流跳转：标签与分支指令

```c
int power(int x, unsigned int n)
{
  int result = 1;
  for (unsigned int i = n; i > 0; i--)
    result *= x;
  return result;
}
```

```
power:
  mov     w2, w0
  mov     w0, 1
  cbz     w1, .L1
.L3:
  mul     w0, w0, w2
  subs    w1, w1, #1
  bne     .L3
.L1:
  ret
```

Q：之前的指令的执行状态如何传递给bne呢?

条件码

- **条件码是一组标志位的统称**
  - 条件码由PSTATE寄存器维护
  - N（Negative）、Z（Zero）、C（Carry）、V（Overflow）
- **条件码保留之前相关指令的执行状态，这种指令包括：**
  - 带有**s**后缀的算术或逻辑运算指令（如subs、adds）
  - **比较指令**
    - cmp：操作数之差；例如 cmp x0, x1
    - cmn：操作数之和；例如 cmn x0, x1
    - tst ：操作数相与；例如 tst x0, x1

条件码的设置

- **通过s后缀数据处理指令隐式设置**

  `adds Rd, Rn, Op2`

  等价于C语言中的: `t = a + b`

  - **C**：当运算产生进位时被设置
  - **Z**：当t为0时被设置
  - **N**：当t小于0时被设置
  - **V**：当运算产生有符号溢出时被设置

    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

条件码的设置

- **通过比较指令cmp显式设置**

  `cmp Src1, Src2`

  计算`Src1 - Src2`，但不存储结果

  - **C**：当减法产生借位时被设置
  - **Z**：当两个操作数相等时被设置
  - **N**：当Src1小于Src2时被设置
  - **V**：当运算产生有符号溢出时被设置

    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

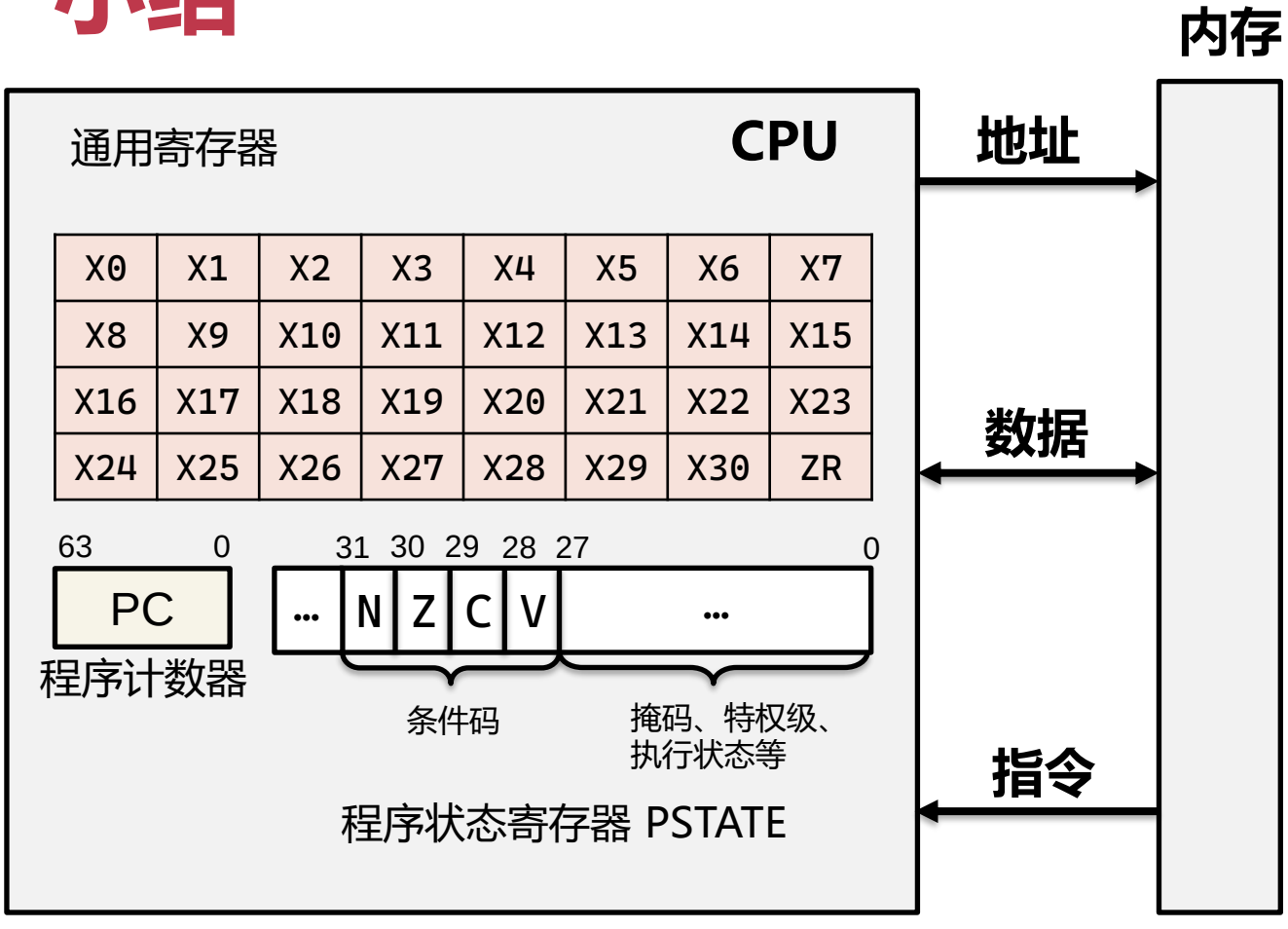
跳转条件

条件	条件码组合	条件含义
EQ	Z	相等或为0
NE	~Z	不等或非0
MI	N	负数
PL	~N	非负数
LT	N^V	有符号小于
LE	(N^V)|Z	有符号小于或等于
GT	~(N^V)&~Z	有符号大于
GE	~(N^V)	有符号大于或等于
HI	C&~Z	无符号大于
LS	~C|Z	无符号小于或等于
LO	~C	无符号小于

跳转指令

- **直接分支指令**
  - 以标签对应的地址作为跳转目标
  - 无条件分支指令 b <label>
  - 有条件分支指令bcond <label>，例如beq，bne，ble
- **间接分支指令**
  - 以寄存器中的地址作为跳转目标
  - br reg，例如 br x0

小结



- 寄存器数据搬移指令
- 算术与逻辑运算指令
- 访存指令
- 分支指令