IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

ARM汇编 – 函数调用

上海交通大学

https://www.sjtu.edu.cn

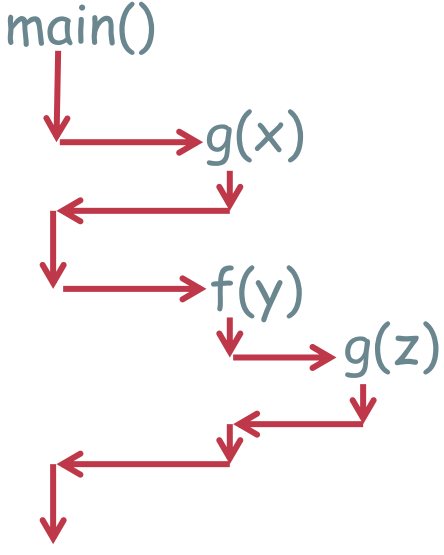
版权声明

- 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

函数调用

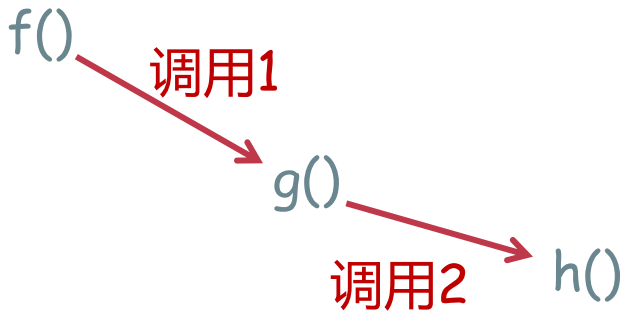
函数调用 vs. 无条件控制流跳转

- **函数调用是另一种形式的无条件跳转**
  - 相同点: 控制流在两段代码间的转移
  - 函数调用的特别之处:
    - 函数在调用后会返回
    - 函数调用涉及参数与返回值的传递
    - 函数存在局部变量
    - 函数调用需要保存某些寄存器（保证返回后能够继续执行）



基本概念

- **术语**
  - Caller 调用者
  - Callee 被调用者



- 调用1
  - 调用者: f
  - 被调用者: g
- 调用2
  - 调用者: g
  - 被调用者: h

函数调用与返回指令

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

```asm
square:
  mul     w0, w0, w0
  ret
cube:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     w19, w0
  bl      square
  mul     w0, w0, w19
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

函数调用指令（caller调用callee）

- **指令**
  - `bl  label`（直接调用，调用函数）
  - `blr  Rn`（间接调用，调用函数指针）

- **功能**
  - 将**返回地址**存储在**链接寄存器LR (x30寄存器的别名)**
  - 跳转到被调用者的**入口地址**

函数返回指令（callee返回caller）

- **指令**
  - `ret`（不区分直接调用与间接调用）
- **功能**
  - 跳转到**返回地址（链接寄存器LR，Link Register）**

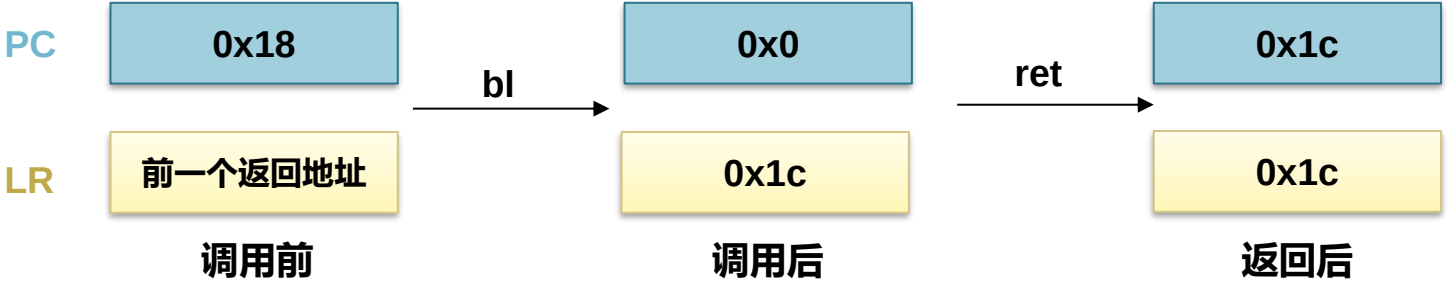
示例：PC与LR的变化

```
int square(int n) {
    return n * n;
}

int cube(int n) {
    return n * square(n);
}
```

```
0000000000000000 <square>:
   0: 00 7c 00 1b    mul  w0, w0, w0
   4: c0 03 5f d6    ret

0000000000000008 <cube>:
    ...
  18: fa ff ff 97    bl   0x0 <square>
  1c: 00 7c 13 1b    mul  w0, w0, w19
    ...
```



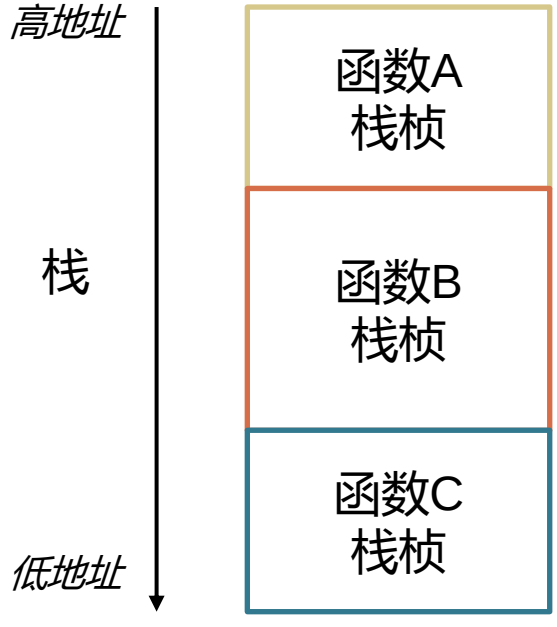
嵌套函数调用

- **cube调用square**
  - cube中的bl指令将返回地址保存在LR中
  - square中的ret指令返回到LR记录的地址
- **cube调用square，square调用foo**
  - LR首先存储了square返回cube的地址
  - 嵌套调用时发生覆盖：LR存储foo返回square的地址

Q：在嵌套函数调用过程中，LR寄存器只有一个，如何存放多个返回地址呢?

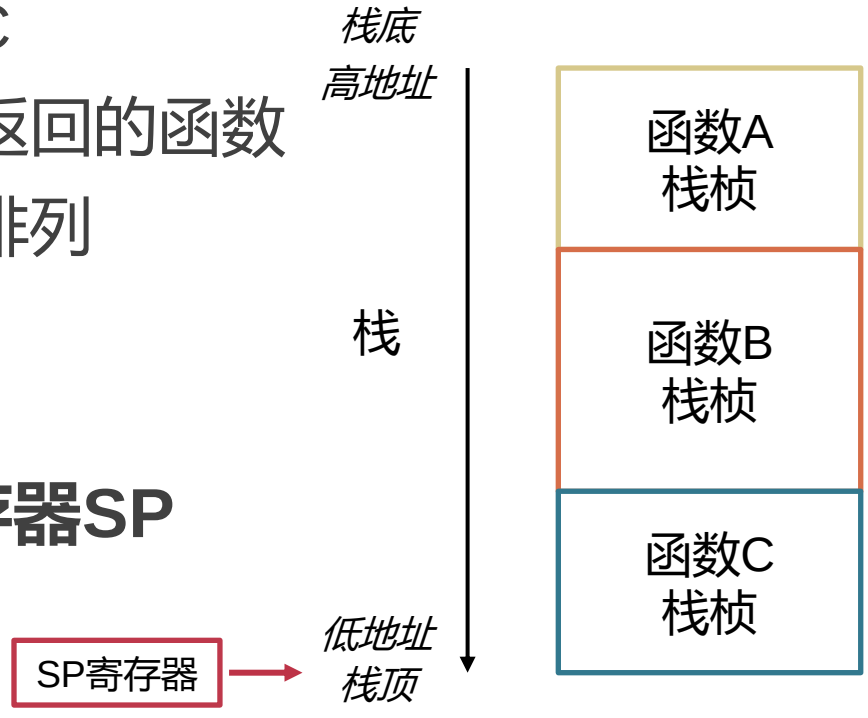
函数栈桢（Stack Frame）

- **栈桢：每个函数在运行期间使用的一段内存**
  - 生命周期：从被调用到返回前
  - 作用：存放其局部状态
    - **存放返回地址**
    - 存放上一个栈桢的位置
    - 存放局部变量
    - …

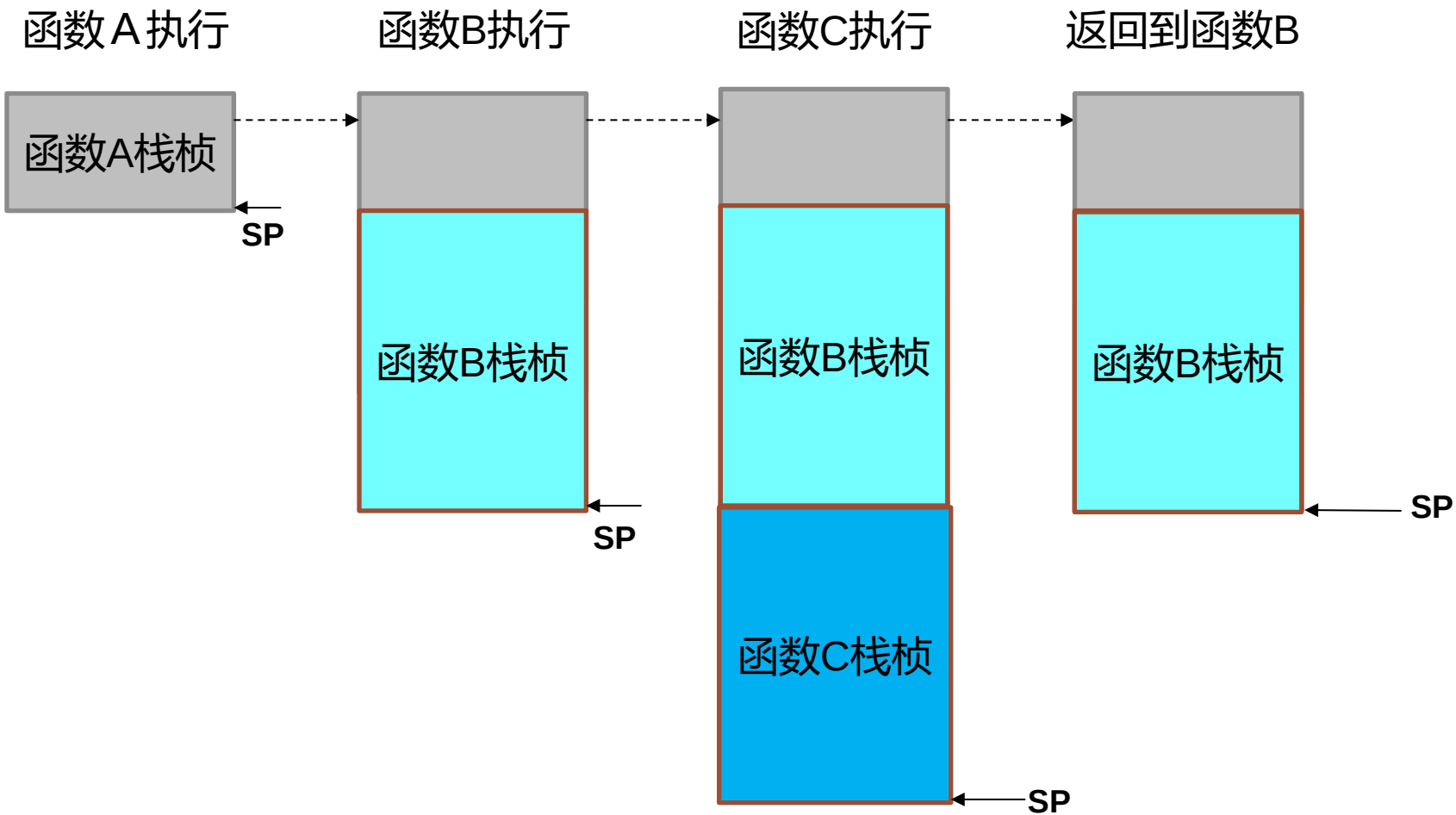


函数栈桢

- **嵌套函数调用**
  - 例如，A调用B、B调用C
  - 程序执行中存在多个未返回的函数
  - 函数栈桢按照调用顺序排列
    - 先被调用者后返回
    - 栈：后进先出
- **CPU中的另一个特殊寄存器SP**
  - 指向栈顶（低地址）



函数调用返回过程中栈的变化



实例：嵌套函数调用

```
0000000000000000 <foo>:
   0:  52800001   mov w1, #0x0
   4:  14000003   b 10 <foo+0x10>
   8:  d503201f   nop
   c:  11000421   add w1, w1, #0x1
  10:  6b00003f   cmp w1, w0
  14:  54ffffab   b.lt 8 <foo+0x8>  // b.tstop
  18:  d65f03c0   ret
```

```
000000000000001c <square>:
  1c:  a9be7bfd   stp x29, x30, [sp, #-32]!
  20:  910003fd   mov x29, sp
  24:  f9000bf3   str x19, [sp, #16]
  28:  2a0003f3   mov w19, w0
  2c:  94000000   bl 0 <foo>
  30:  1b137e60   mul w0, w19, w19
  34:  f9400bf3   ldr x19, [sp, #16]
  38:  a8c27bfd   ldp x29, x30, [sp], #32
  3c:  d65f03c0   ret
```

```
0000000000000040 <cube>:
  40:  a9be7bfd   stp x29, x30, [sp, #-32]!
  44:  910003fd   mov x29, sp
  48:  f9000bf3   str x19, [sp, #16]
  4c:  2a0003f3   mov w19, w0
  50:  94000000   bl 1c <square>
  54:  1b137c00   mul w0, w0, w19
  58:  f9400bf3   ldr x19, [sp, #16]
  5c:  a8c27bfd   ldp x29, x30, [sp], #32
  60:  d65f03c0   ret
```

```
1  void foo(int n)
2  {
3          int i;
4          for (i = 0; i < n; ++i) {
5                  asm volatile("nop");
6          }
7  }
8
9  int square(int n)
10 {
11         foo(n);
12         return n * n;
13 }
14
15 int cube(int n)
16 {
17         return n * square(n);
18 }
```

实例：嵌套函数调用

```
0000000000000000 <foo>:
   0:  52800001   mov w1, #0x0
   4:  14000003   b 10 <foo+0x10>
   8:  d503201f   nop
   c:  11000421   add w1, w1, #0x1
  10:  6b00003f   cmp w1, w0
  14:  54ffffab   b.lt 8 <foo+0x8>  // b.tstop
  18:  d65f03c0   ret
```

```
000000000000001c <square>:
  1c:  a9be7bfd   stp x29, x30, [sp, #-32]!
  20:  910003fd   mov x29, sp
  24:  f9000bf3   str x19, [sp, #16]
  28:  2a0003f3   mov w19, w0
  2c:  94000000   bl 0 <foo>
  30:  1b137e60   mul w0, w19, w19
  34:  f9400bf3   ldr x19, [sp, #16]
  38:  a8c27bfd   ldp x29, x30, [sp], #32
  3c:  d65f03c0   ret
```

```
0000000000000040 <cube>:
  40:  a9be7bfd   stp x29, x30, [sp, #-32]!
  44:  910003fd   mov x29, sp
  48:  f9000bf3   str x19, [sp, #16]
  4c:  2a0003f3   mov w19, w0
  50:  94000000   bl 1c <square>
  54:  1b137c00   mul w0, w0, w19
  58:  f9400bf3   ldr x19, [sp, #16]
  5c:  a8c27bfd   ldp x29, x30, [sp], #32
  60:  d65f03c0   ret
```

注意-1：x30就是LR寄存器

访存指令

R_s指寄存器的大小（字节数）

mem[a : b] 指地址 a 到地址 b 的内存范围

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

指令	效果
ldp R1, R2, addr	R1,R2 ← mem[addr : addr+$R1_s$+$R2_s$]
stp R1, R2, addr	mem[addr : addr+$R1_s$+$R2_s$] ← R1,R2

实例：嵌套函数调用

```
0000000000000000 <foo>:
   0:  52800001   mov w1, #0x0
   4:  14000003   b 10 <foo+0x10>
   8:  d503201f   nop
   c:  11000421   add w1, w1, #0x1
  10:  6b00003f   cmp w1, w0
  14:  54ffffab   b.lt 8 <foo+0x8>  // b.tstop
  18:  d65f03c0   ret
```

```
000000000000001c <square>:
  1c:  a9be7bfd   stp x29, x30, [sp, #-32]!
  20:  910003fd   mov x29, sp
  24:  f9000bf3   str x19, [sp, #16]
  28:  2a0003f3   mov w19, w0
  2c:  94000000   bl 0 <foo>
  30:  1b137e60   mul w0, w19, w19
  34:  f9400bf3   ldr x19, [sp, #16]
  38:  a8c27bfd   ldp x29, x30, [sp], #32
  3c:  d65f03c0   ret
```

```
0000000000000040 <cube>:
  40:  a9be7bfd   stp x29, x30, [sp, #-32]!
  44:  910003fd   mov x29, sp
  48:  f9000bf3   str x19, [sp, #16]
  4c:  2a0003f3   mov w19, w0
  50:  94000000   bl 1c <square>
  54:  1b137c00   mul w0, w0, w19
  58:  f9400bf3   ldr x19, [sp, #16]
  5c:  a8c27bfd   ldp x29, x30, [sp], #32
  60:  d65f03c0   ret
```

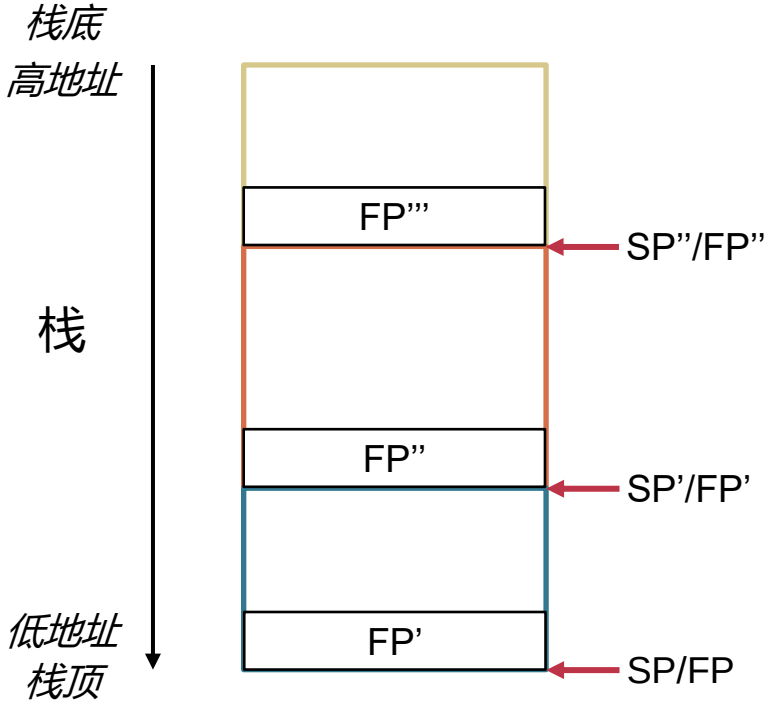
注意-1：x30就是LR寄存器

注意-2：同时保存了x29

帧指针FP：x29寄存器

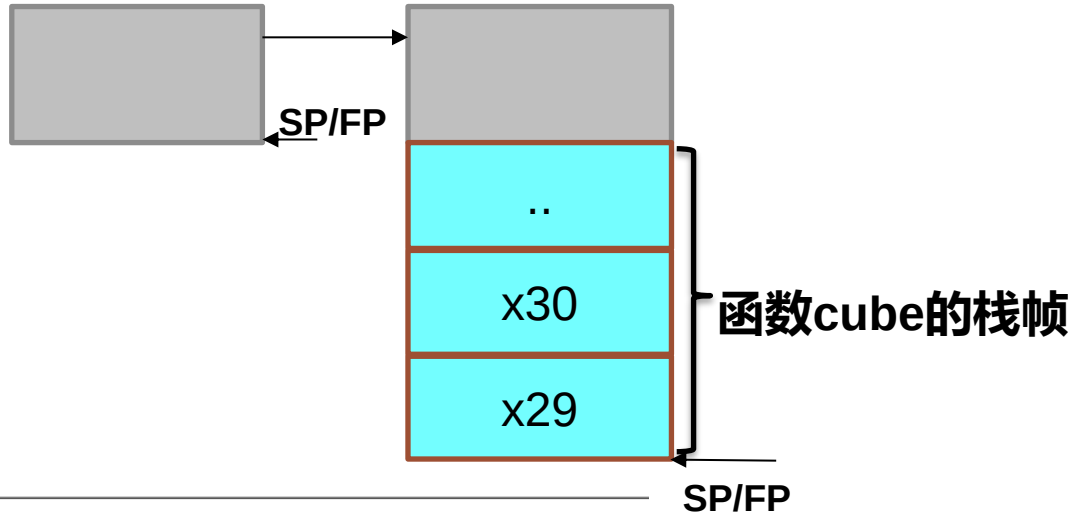
- **栈桢回溯**
  - 栈桢大小不一
  - 如何找到上一个栈桢（如调试）
    - 保存x29（上一个栈桢的SP）
    - 将当前SP写入x29（让callee能保存）

```
4    cube:
5      stp     x29, x30, [sp, -32]!
6      mov     x29, sp
7      str     x19, [sp, 16]
8      mov     w19, w0
9      bl      square
10     mul     w0, w0, w19
11     ldr     x19, [sp, 16]
12     ldp     x29, x30, [sp], 32
13     ret
```



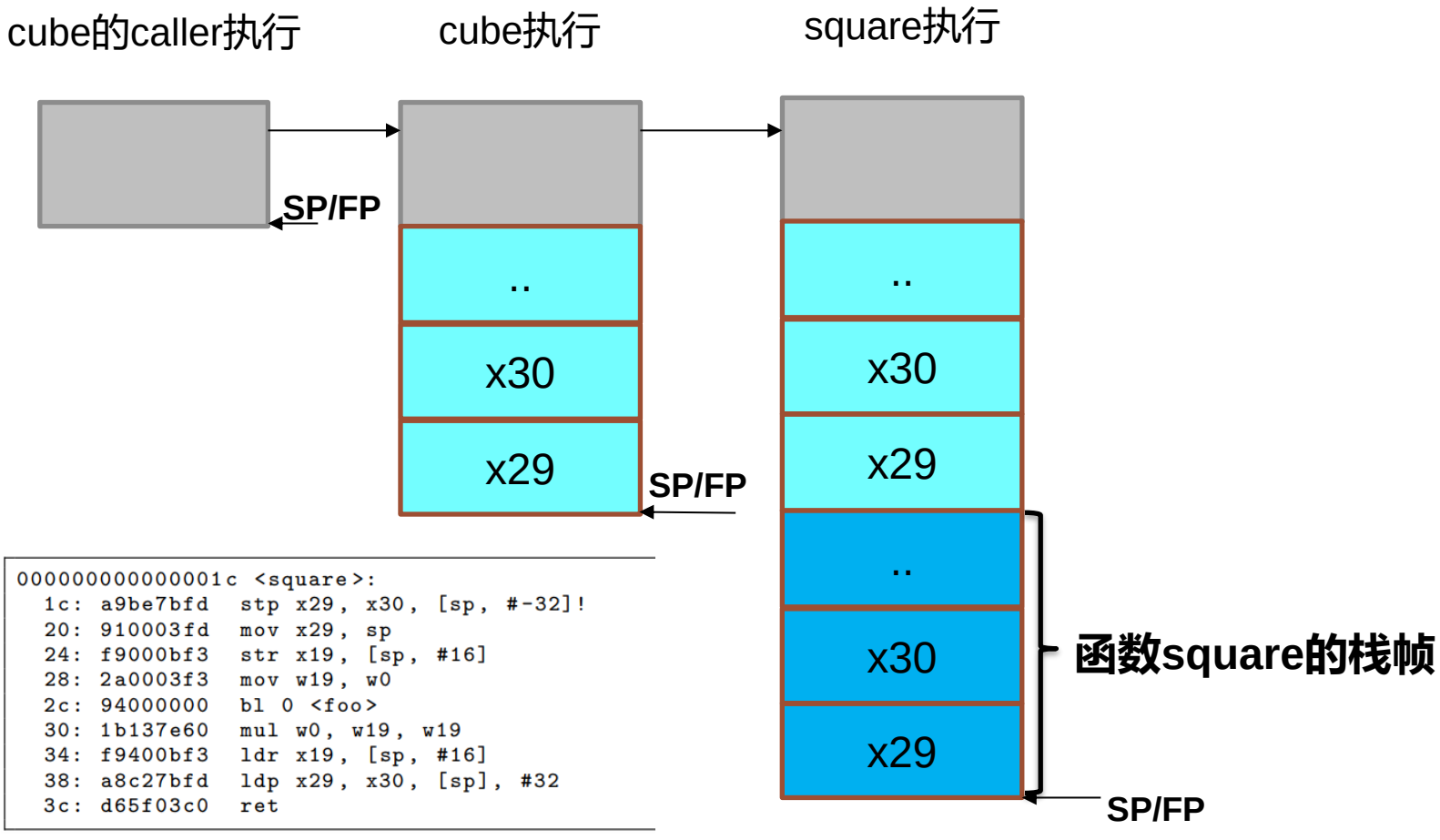
函数的调用、返回与栈

cube的caller执行　　cube执行



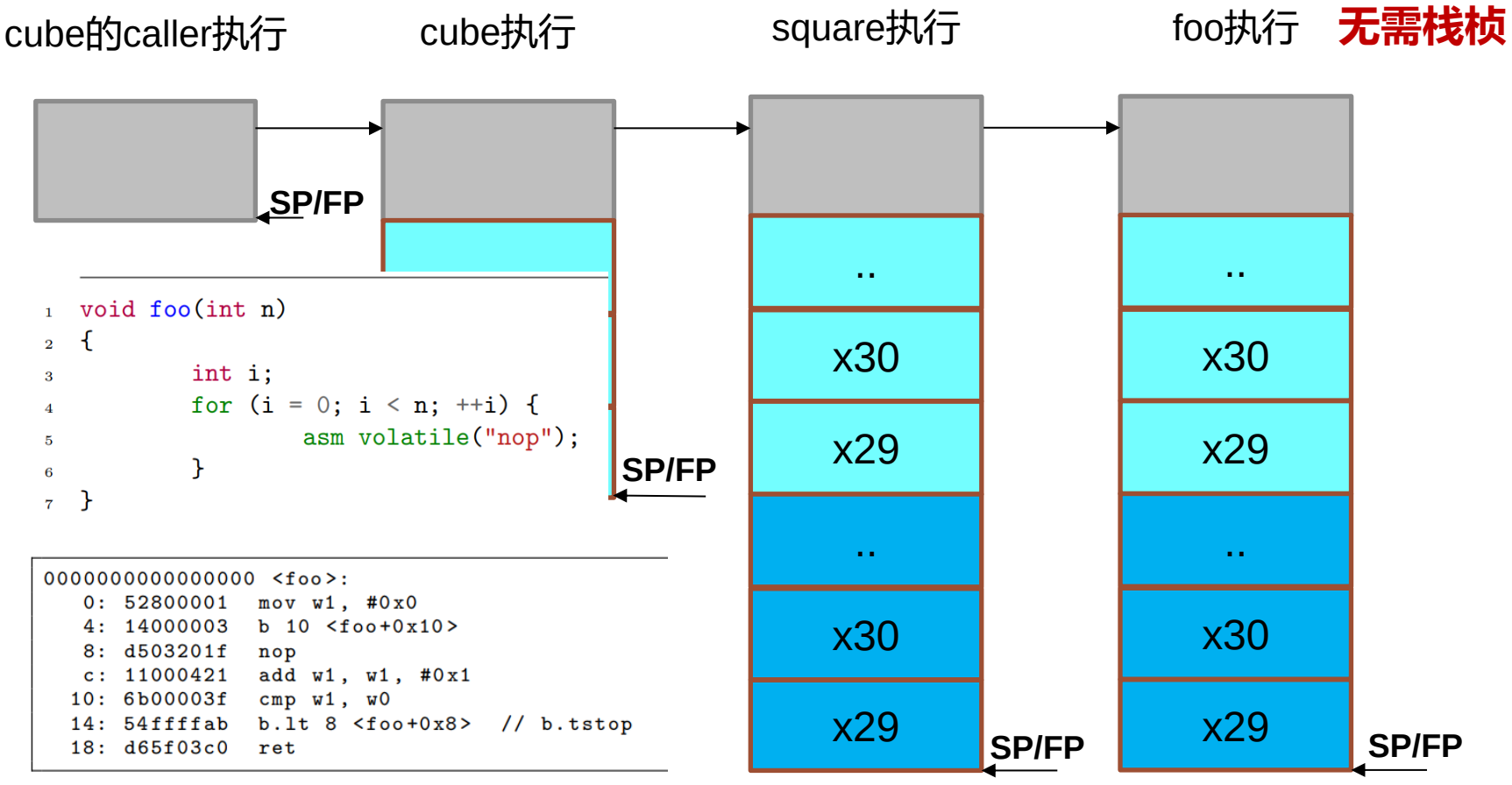

```
0000000000000040 <cube>:
  40: a9be7bfd   stp x29, x30, [sp, #-32]!
  44: 910003fd   mov x29, sp
  48: f9000bf3   str x19, [sp, #16]
  4c: 2a0003f3   mov w19, w0
  50: 94000000   bl 1c <square>
  54: 1b137c00   mul w0, w0, w19
  58: f9400bf3   ldr x19, [sp, #16]
  5c: a8c27bfd   ldp x29, x30, [sp], #32
  60: d65f03c0   ret
```

函数的调用、返回与栈

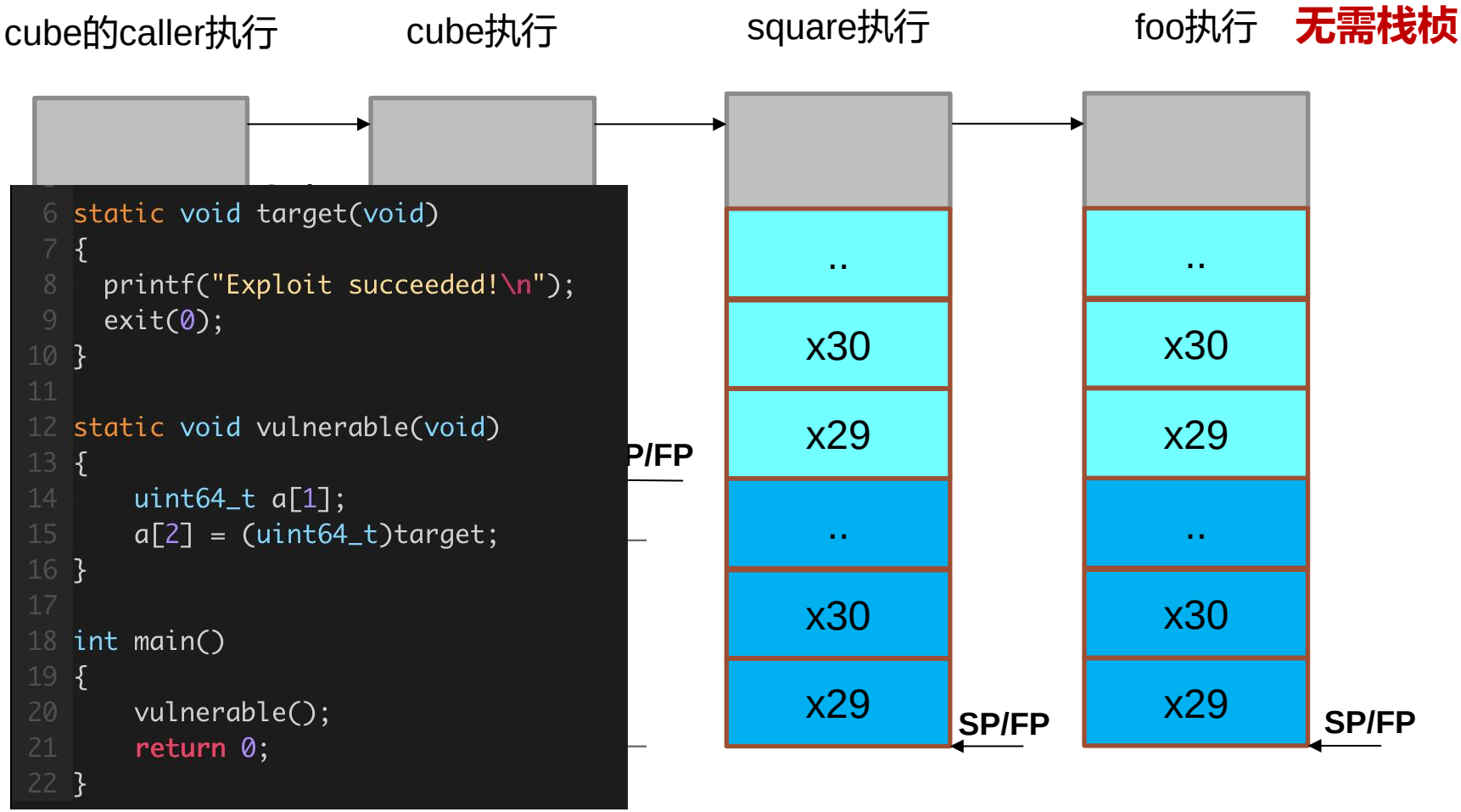



```
000000000000001c <square>:
  1c: a9be7bfd   stp x29, x30, [sp, #-32]!
  20: 910003fd   mov x29, sp
  24: f9000bf3   str x19, [sp, #16]
  28: 2a0003f3   mov w19, w0
  2c: 94000000   bl 0 <foo>
  30: 1b137e60   mul w0, w19, w19
  34: f9400bf3   ldr x19, [sp, #16]
  38: a8c27bfd   ldp x29, x30, [sp], #32
  3c: d65f03c0   ret
```

函数的调用、返回与栈


cube的caller执行
cube执行
square执行
foo执行
无需栈桢
SP/FP
SP/FP
SP/FP
SP/FP
..
x30
x29
..
x30
x29
..
x30
x29
1 void foo(int n)
2 {
3 int i;
4 for (i = 0; i < n; ++i) {
5 asm volatile("nop");
6 }
7 }
0000000000000000 <foo>:
0: 52800001 mov w1, #0x0
4: 14000003 b 10 <foo+0x10>
8: d503201f nop
c: 11000421 add w1, w1, #0x1
10: 6b00003f cmp w1, w0
14: 54ffffab b.lt 8 <foo+0x8> // b.tstop
18: d65f03c0 ret

Q：ROP攻击如何实现?


cube的caller执行
cube执行
square执行
foo执行
无需栈桢
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;
}
..
x30
x29
..
x30
x29
SP/FP
..
x30
x29
SP/FP

函数参数与返回值

通过寄存器传递数据

- 调用者使用x0 ~ x7寄存器传递前8个参数
- 被调用者使用x0寄存器传递返回值

回顾：参数与寄存器的对应关系

```
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, #0xF0F0F0F          # tmp = 0x0F0F0F0F
and     x0, x0, x1              # t3 = t1 & const
sub     w0, w2, w0              # t4 = t2 - t3
ret                             # return t4
```

初始时x0、x1、x2分别对应x、y、z

实例：通过寄存器传递参数与返回值

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

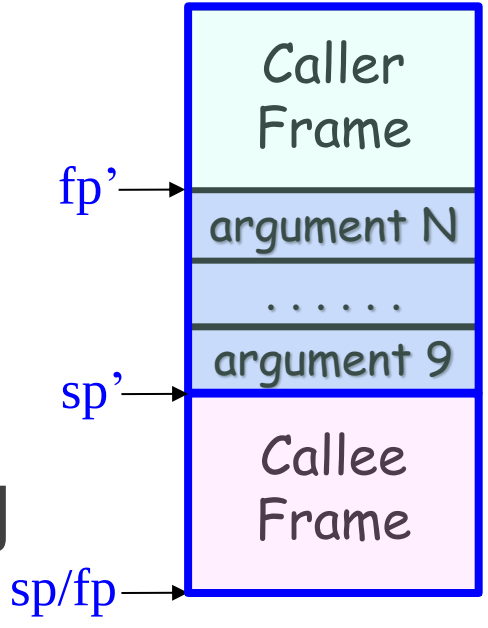
```asm
square:
    mul     w0, w0, w0
    ret
cube:
    stp     x29, x30, [sp, -32]!
    mov     x29, sp
    str     x19, [sp, 16]
    mov     w19, w0
    bl      square
    mul     w0, w0, w19
    ldr     x19, [sp, 16]
    ldp     x29, x30, [sp], 32
    ret
```

square和cube函数均是：

1. 参数n在w0中
2. 计算结果写入w0返回

传递数据

- **调用者压到栈上的数据**
  - 第8个之后的参数
  - 按声明顺序**从右到左**
  - 所有数据对齐到8字节
- **被调用者通过SP+偏移量访问**



传递数据：参数

```
void proc(  long  a1,  long  *a1p,
            int   a2,  int   *a2p,
            short a3,  short *a3p,
            char  a4,  char  *a4p,
            char  a5,  char  *a5p) {
       *a1p += a1 ;
       *a2p += a2 ;
       *a3p += a3 ;
       *a4p += a4 ;
       *a5p += a5 ;
}

void caller(long *n) {
    proc(1, 0x2000, 3, 0x4000, 5, 0x6000, 7, 0x8000, 9, 0xA000);
}
```

传递数据：参数

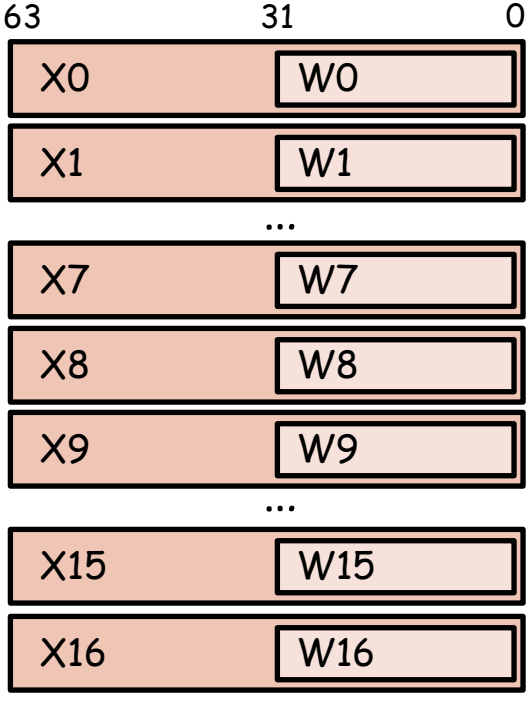
```
0000000000000048 <caller>:
  48:   d10083ff        sub     sp, sp, #0x20
  4c:   a9017bfd        stp     x29, x30, [sp, #16]
  50:   910043fd        add     x29, sp, #0x10
  54:   d2940000        mov     x0, #0xa000                     // #40960
  58:   f90007e0        str     x0, [sp, #8]
  5c:   52800120        mov     w0, #0x9                        // #9
  60:   390003e0        strb    w0, [sp]
  64:   d2900007        mov     x7, #0x8000                     // #32768
  68:   528000e6        mov     w6, #0x7                        // #7
  6c:   d28c0005        mov     x5, #0x6000                     // #24576
  70:   528000a4        mov     w4, #0x5                        // #5
  74:   d2880003        mov     x3, #0x4000                     // #16384
  78:   52800062        mov     w2, #0x3                        // #3
  7c:   d2840001        mov     x1, #0x2000                     // #8192
  80:   d2800020        mov     x0, #0x1                        // #1
  84:   94000000        bl      0 <proc>
  88:   a9417bfd        ldp     x29, x30, [sp, #16]
  8c:   910083ff        add     sp, sp, #0x20
  90:   d65f03c0        ret
```

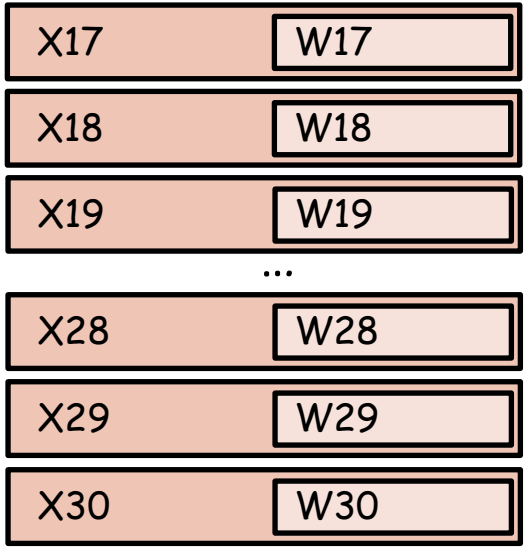
寄存器保存

通用寄存器保存

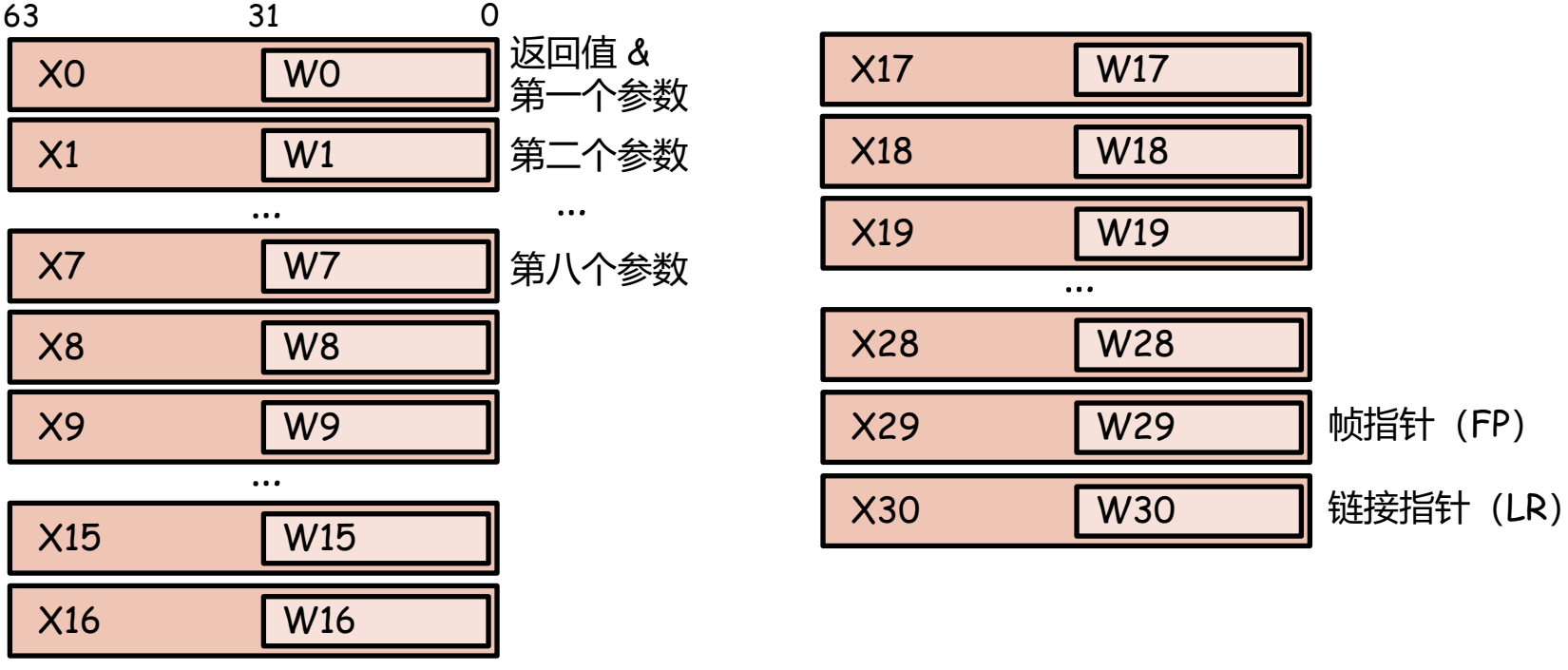
- **嵌套函数共享同一批通用寄存器**
  - 因此能够通过寄存器传递参数和返回值
  - 不同的函数对通用寄存器的使用会存在冲突—无覆盖
- **避免冲突的思路**
  - 函数在使用某个寄存器之前保存该寄存器，返回前恢复它
  - 保存在哪：函数栈桢中
  - 效率问题：存在实际无需保存的情况，因此不用每次都保存
    - 例如，函数不再调用其它函数
    - 例如，函数A在调用函数B之前使用x9，但其实之后A不再需要

31个通用寄存器


63
31
0
X0
W0
X1
W1
...
X7
W7
X8
W8
X9
W9
...
X15
W15
X16
W16

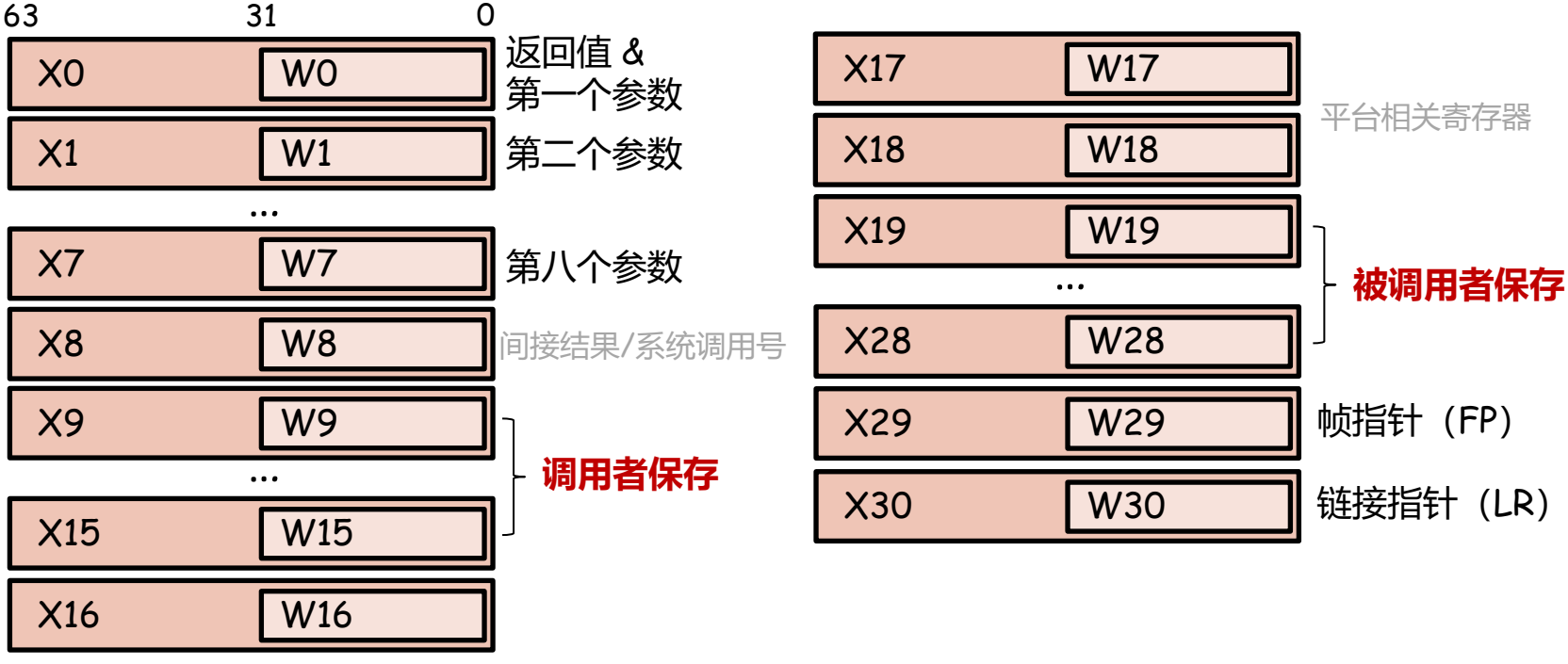

X17
W17
X18
W18
X19
W19
...
X28
W28
X29
W29
X30
W30

31个通用寄存器


63
31
0
X0
W0
返回值 &
第一个参数
X1
W1
第二个参数
...
...
X7
W7
第八个参数
X8
W8
X9
W9
...
X15
W15
X16
W16
X17
W17
X18
W18
X19
W19
...
X28
W28
X29
W29
帧指针（FP）
X30
W30
链接指针（LR）

31个通用寄存器

约定：x9-x15调用者保存；x19-x28被调用者保存



寄存器使用约定

- **调用者保存的寄存器包括X9~X15**
  - 调用者在调用前**按需（仅考虑自己是否需要）**进行保存
    - 调用者在被调用者返回后**恢复**它们的值
  - 被调用者可以随意使用
    - 调用者视角：这些寄存器在函数调用之后的值可能发生改变

寄存器使用约定

- **被调用者保存的寄存器包括X19~X28**
  - 被调用者在使用前进行**保存**
  - 被调用者在返回前进行**恢复**
  - 调用者视角：这些寄存器的值在函数调用前后不会改变

实例：保存寄存器

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

```
square:
  mul     w0, w0, w0
  ret
cube:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     w19, w0
  bl      square
  mul     w0, w0, w19
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

cube作为被调用者（比如main函数调用cube），在使用x19前需要在栈上保存它

问：若使用调用者保存的寄存器（如x9），是否能够避免保存?

再看CUBE函数

实例：理解cube函数汇编

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

```
square:
  mul     w0, w0, w0
  ret
cube:
  stp     x29, x30, [sp, -32]!
  mov     x29, sp
  str     x19, [sp, 16]
  mov     w19, w0
  bl      square
  mul     w0, w0, w19
  ldr     x19, [sp, 16]
  ldp     x29, x30, [sp], 32
  ret
```

1. 开辟栈桢
2. 保存caller栈桢的FP（x29）
3. 保存返回地址LR（x30）

实例：理解cube函数汇编

```
1   int square(int n)
2   {
3     return n * n;
4   }
5
6   int cube(int n)
7   {
8     return n * square(n);
9   }
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

将当前栈桢SP写入FP

实例：理解cube函数汇编

```
1   int square(int n)
2   {
3     return n * n;
4   }
5
6   int cube(int n)
7   {
8     return n * square(n);
9   }
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

保存被调用者保存寄存器x19

把参数n的值写入w19

实例：理解cube函数汇编

```
1   int square(int n)
2   {
3     return n * n;
4   }
5
6   int cube(int n)
7   {
8     return n * square(n);
9   }
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

调用square函数（结果在w0中）

实例：理解cube函数汇编

```
1   int square(int n)
2   {
3     return n * n;
4   }
5
6   int cube(int n)
7   {
8     return n * square(n);
9   }
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

n * square(n)
n在w19中，square(n)在w0中

实例：理解cube函数汇编

```c
int square(int n)
{
  return n * n;
}

int cube(int n)
{
  return n * square(n);
}
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

恢复被调用者保存寄存器x19

实例：理解cube函数汇编

```
1   int square(int n)
2   {
3     return n * n;
4   }
5
6   int cube(int n)
7   {
8     return n * square(n);
9   }
```

```
1   square:
2     mul     w0, w0, w0
3     ret
4   cube:
5     stp     x29, x30, [sp, -32]!
6     mov     x29, sp
7     str     x19, [sp, 16]
8     mov     w19, w0
9     bl      square
10    mul     w0, w0, w19
11    ldr     x19, [sp, 16]
12    ldp     x29, x30, [sp], 32
13    ret
```

1. 释放栈桢，恢复SP
2. 恢复caller栈桢的FP（x29）
3. 恢复返回地址LR（x30）

返回到x30中存储的返回地址

局部变量

函数局部变量存放在函数栈桢中

- **为什么不直接把局部变量存储在寄存器?**
  - 寄存器数量有限
  - 数组和结构体等复杂数据结构
  - 局部变量可能需要寻址 (如&a)

局部变量

- **局部变量的分配**
  - 在分配栈帧时被一起分配
- **局部变量的释放**
  - 在返回前释放栈帧时释放
- **局部变量通过SP相对地址引用** **(例如ldr x1, [sp, #8])**

实例：栈上局部变量

```
long caller()
{
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;

        return sum * diff;
}
```

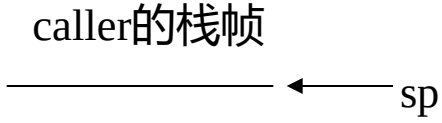
```
long swap_add(long *xp, long *yp)
{
        long x = *xp;
        long y = *yp;

        *xp = y;
        *yp = x;
        return x + y;
}
```

栈上局部变量的例子

caller:

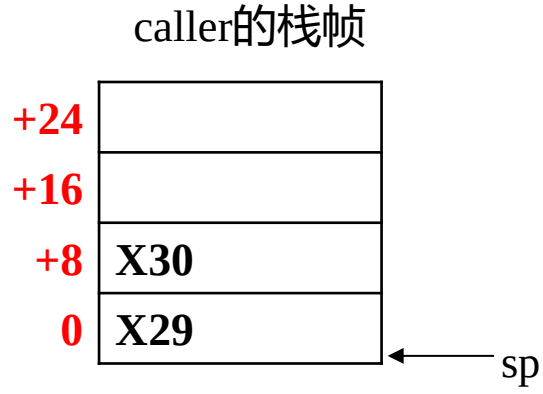
```
stp     x29, x30, [sp, -32]!
mov     x29, sp
```




```
long caller():
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
```

栈上局部变量的例子

```
caller:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        mov     x0, 534
        str     x0, [sp, 24]
```

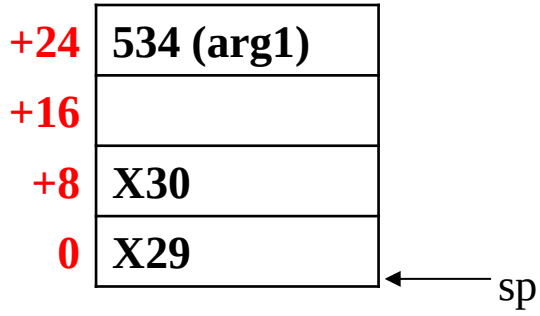



```
long caller():
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
```

栈上局部变量的例子

```
caller:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        mov     x0, 534
        str     x0, [sp, 24]
        mov     x0, 1057
        str     x0, [sp, 16]
```

caller的栈帧

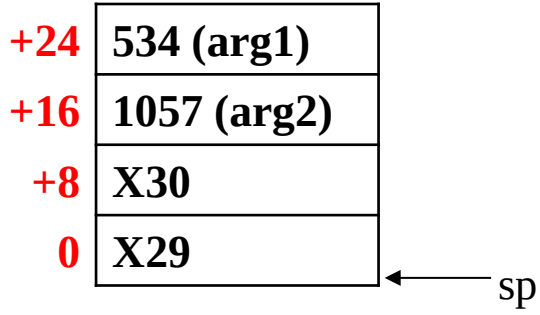



```
long caller():
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
```

栈上局部变量的例子

```
caller:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        mov     x0, 534
        str     x0, [sp, 24]
        mov     x0, 1057
        str     x0, [sp, 16]
        add     x1, sp, 16
        add     x0, sp, 24
```

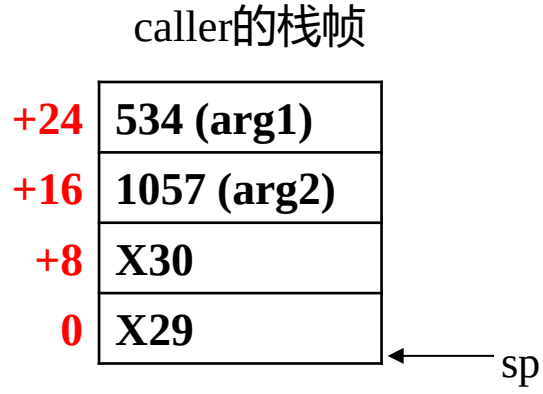
caller的栈帧




```
long caller():
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
```

栈上局部变量的例子

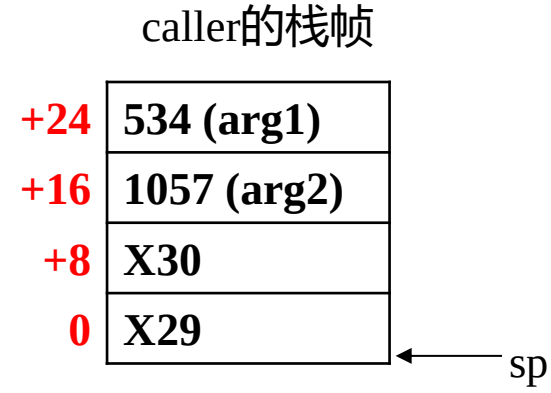
```
caller:
        stp     x29, x30, [sp, -32]!
        mov     x29, sp
        mov     x0, 534
        str     x0, [sp, 24]
        mov     x0, 1057
        str     x0, [sp, 16]
        add     x1, sp, 16
        add     x0, sp, 24
        bl      swap_add
```




```
long caller():
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
```

栈上局部变量的例子

```
swap_add:
        ldr     x3, [x0]
        ldr     x2, [x1]
        str     x2, [x0]
        str     x3, [x1]
        add     x0, x3, x2
        ret
```

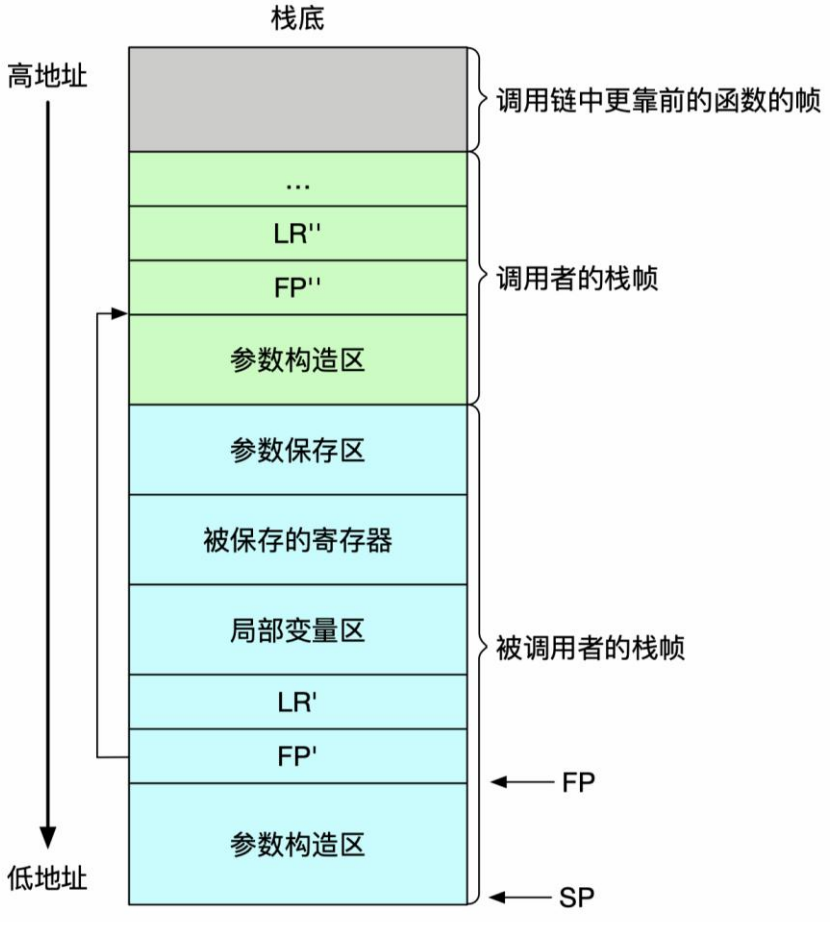



```
long swap_add(long *xp, long *yp):
        long x = *xp;
        long y = *yp;
        *xp = y;
        *yp = x;
        return x + y;
```

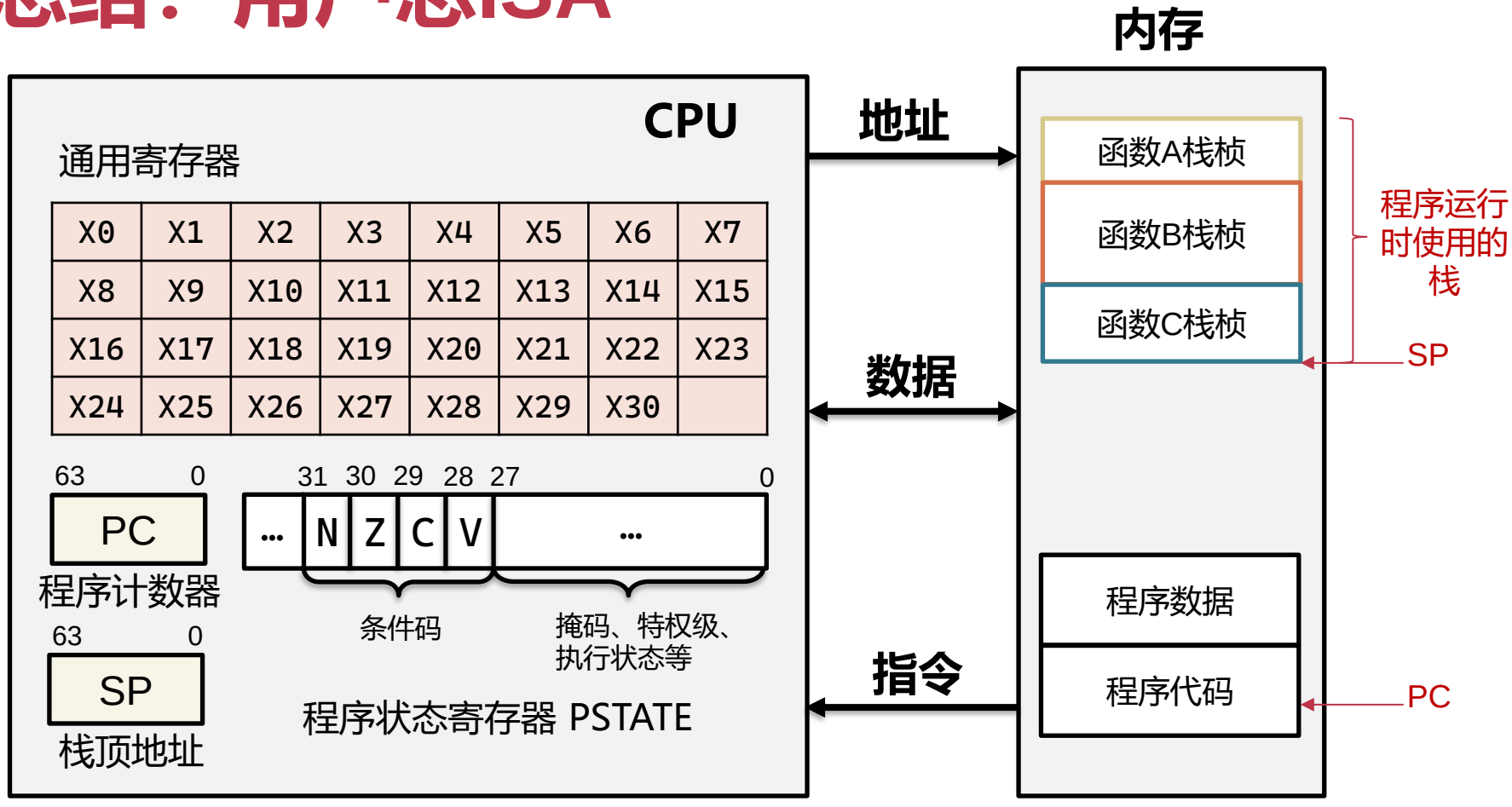
小结：函数调用

- ✓ 调用被调用者：**bl**指令
- ✓ 返回到调用者：**ret**指令
- ✓ 传递数据：**寄存器**与**栈**
- ✓ 寄存器使用约定：**调用者保存、被调用者保存**
- ✓ 局部变量：存在函数**栈桢中**

小结：栈的全貌


栈底
高地址
调用链中更靠前的函数的帧
...
LR''
FP''
调用者的栈帧
参数构造区
参数保存区
被保存的寄存器
局部变量区
被调用者的栈帧
LR'
FP'
FP
参数构造区
SP
低地址

总结：用户态ISA


内存
CPU
通用寄存器
X0 X1 X2 X3 X4 X5 X6 X7
X8 X9 X10 X11 X12 X13 X14 X15
X16 X17 X18 X19 X20 X21 X22 X23
X24 X25 X26 X27 X28 X29 X30
63 0
PC
程序计数器
63 0
SP
栈顶地址
31 30 29 28 27 0
… N Z C V …
条件码
掩码、特权级、
执行状态等
程序状态寄存器 PSTATE
地址
数据
指令
函数A栈桢
函数B栈桢
函数C栈桢
程序运行时使用的栈
SP
程序数据
程序代码
PC

课程实验：拆炸弹