中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

硬件环境与软件抽象：ARM指令集架构

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

大纲

- 为什么选择ARM
- 硬件执行逻辑
- ARM汇编语言
- 内存模型

大纲

- 为什么选择ARM
- 硬件执行逻辑
- ARM汇编语言
- 内存模型

ARM: 智能手机的模式指令集

华为海思：麒麟9000

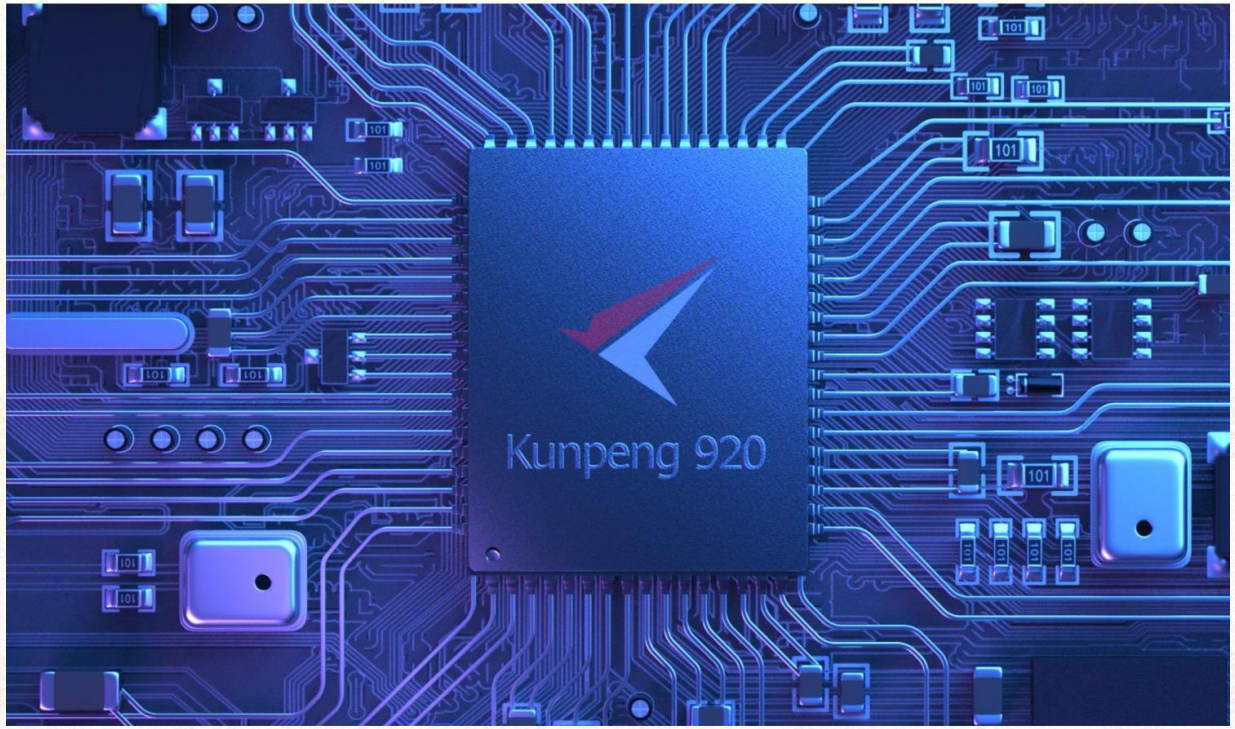
高通：骁龙8 GenX

联发科：天玑9000系列

苹果：A17
(iPhone 15 pro)

ARM：正在走向PC/服务器

苹果M1/M2/M3/M4 用于笔记本、台式机



华为鲲鹏920用于服务器

ARM发展

1993年
- ARMv4及之前
  - ARM 32位指令集

1998年
- ARMv4T、ARMv5TE
  - 加入Thumb 16位指令集

2001年
- ARMv6
  - 多核、SIMD、TrustZone®
  - Thumb-2 32位指令集

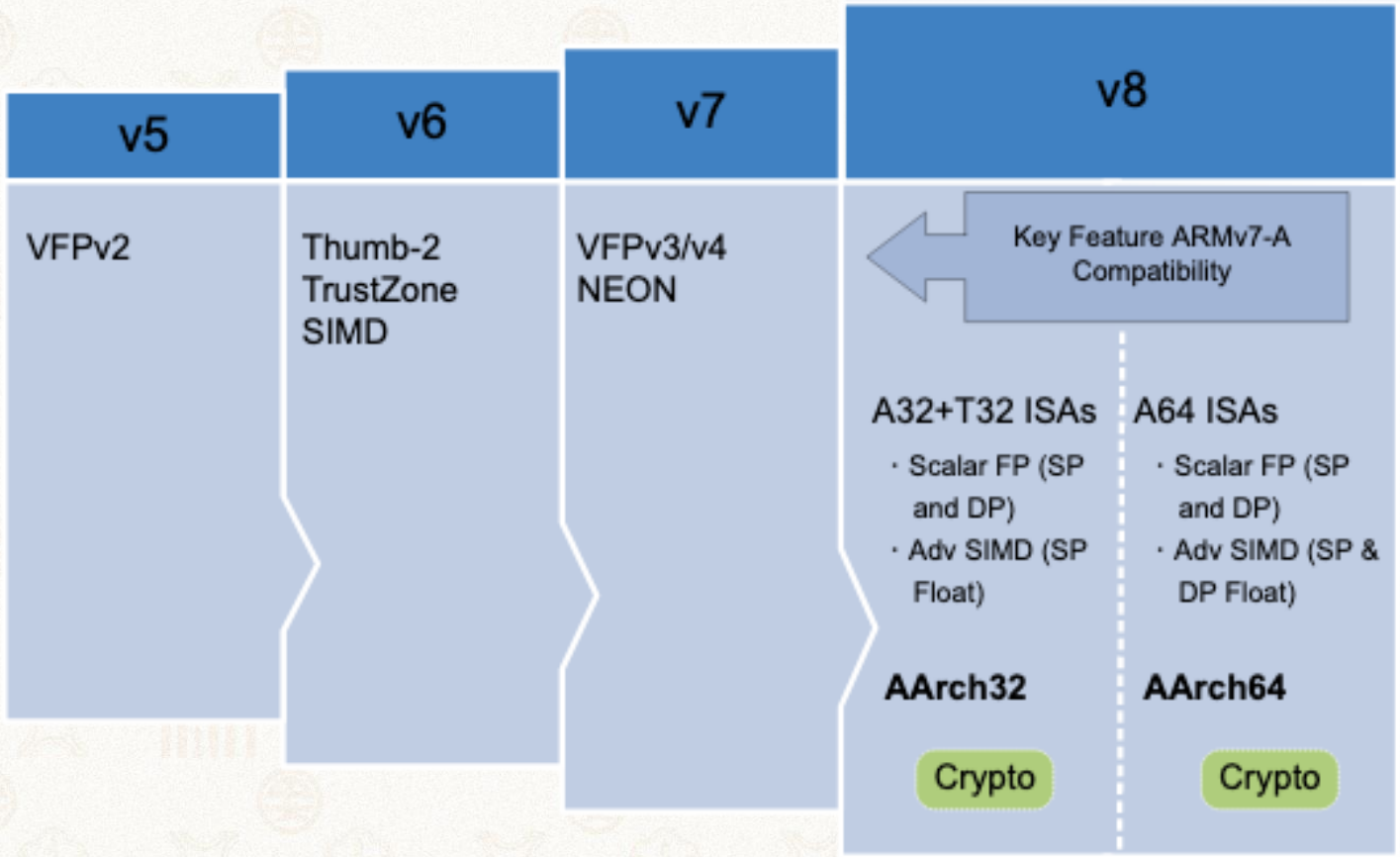
2004年
- ARMv7
  - 增强SIMD拓展——NEON
  - 全面支持平台操作系统，如Linux
  - 可预测实时高性能

2011年

ARMv8

- 2011年发布

- 扩大物理寻址
  - 4GB以外的物理地址
- 64位虚拟地址
- 自动事件信号
  - 低功耗、高性能的自旋锁
- 硬件加速加密
- 新的异常模型

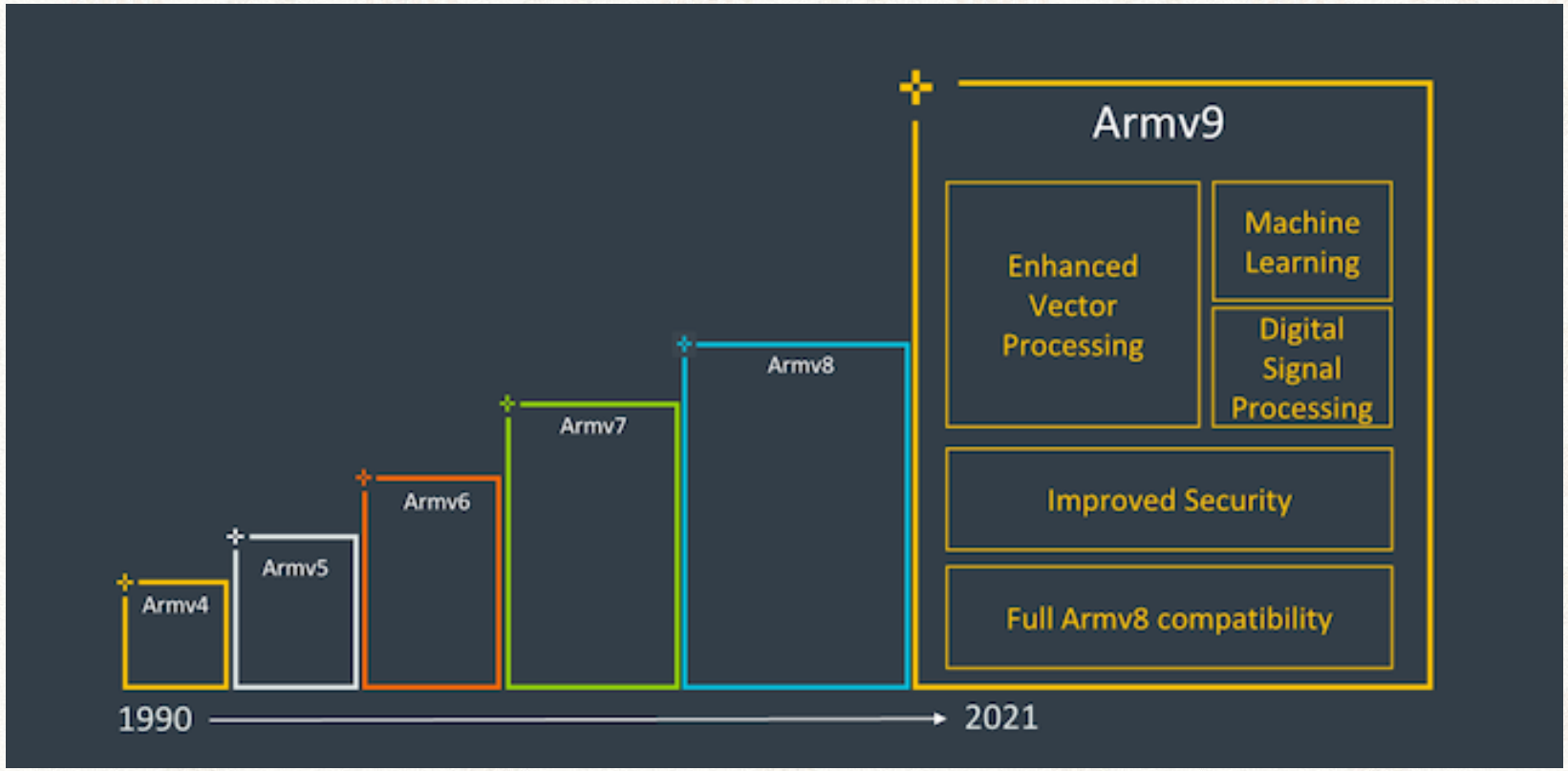


ARMv9

- 2021年公布

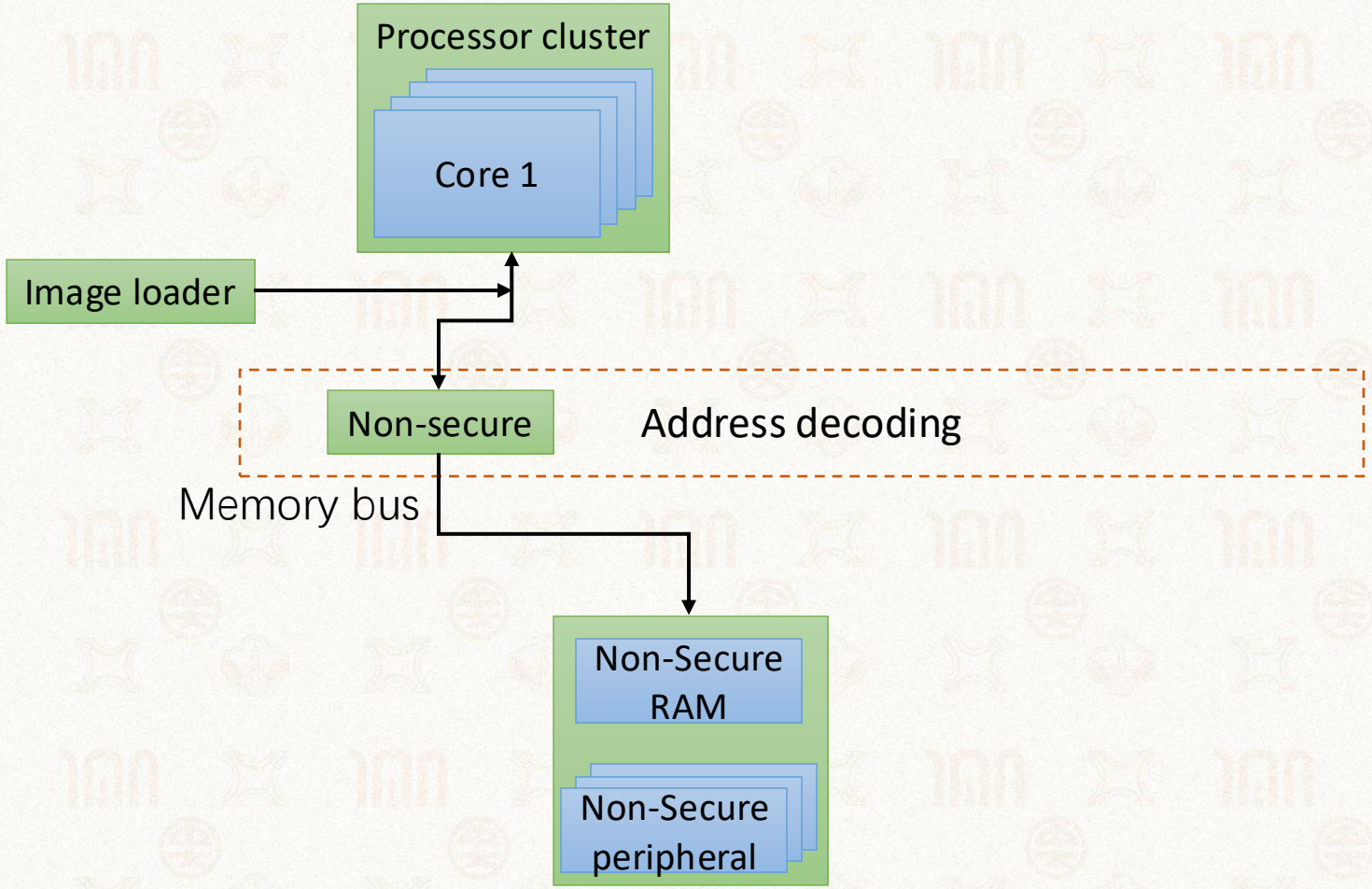
- 优化的领域：
  - 安全性
  - 机器学习
  - 矢量运算

- 基本架构与Armv8一致
  - 课程只涉及v8



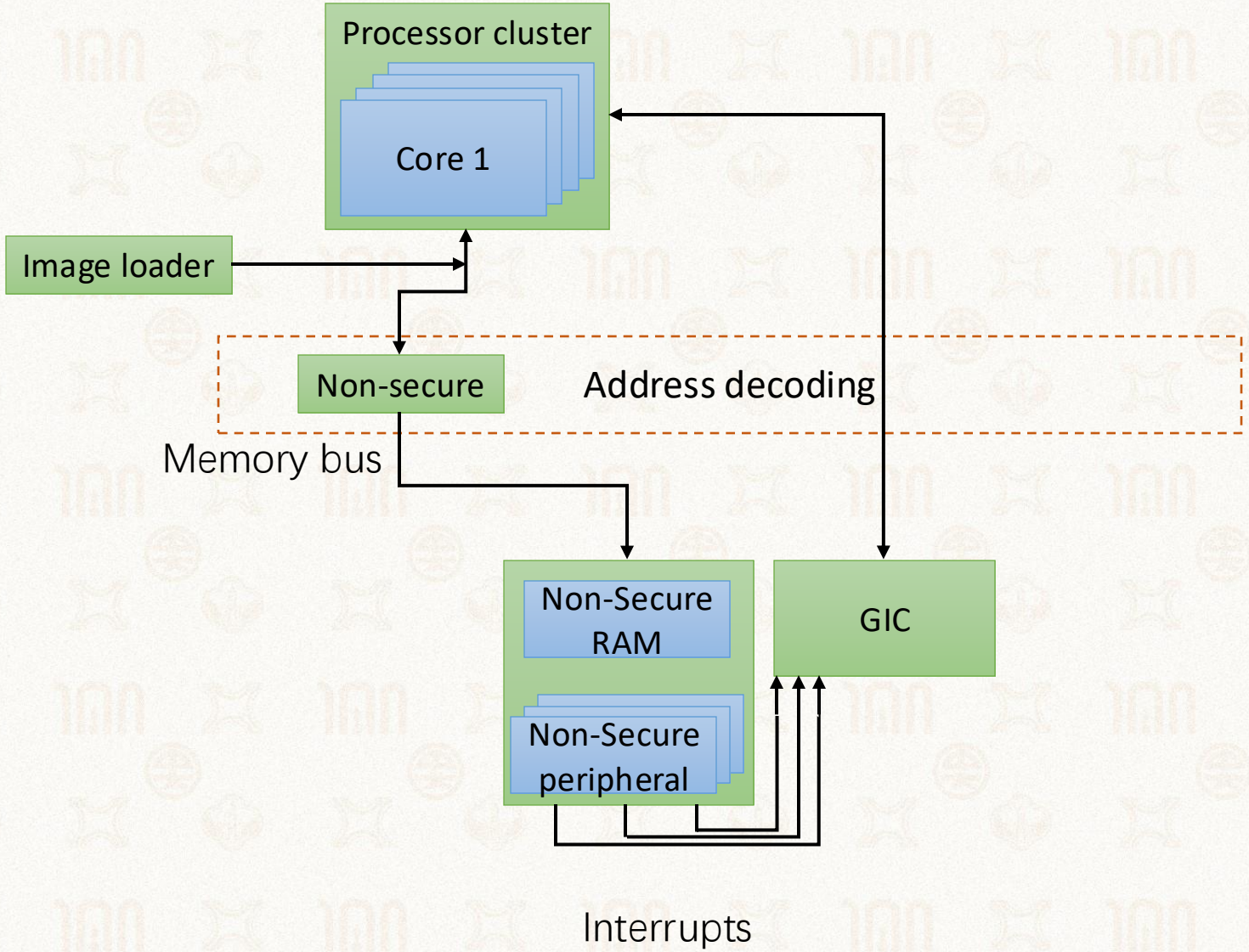
ARMv8基础平台

- 基本的冯诺依曼架构
  - 计算
  - 存储器
    - 内存(RAM)
  - 输入输出
    - 镜像加载
    - 外设(peripheral)



ARMv8基础平台

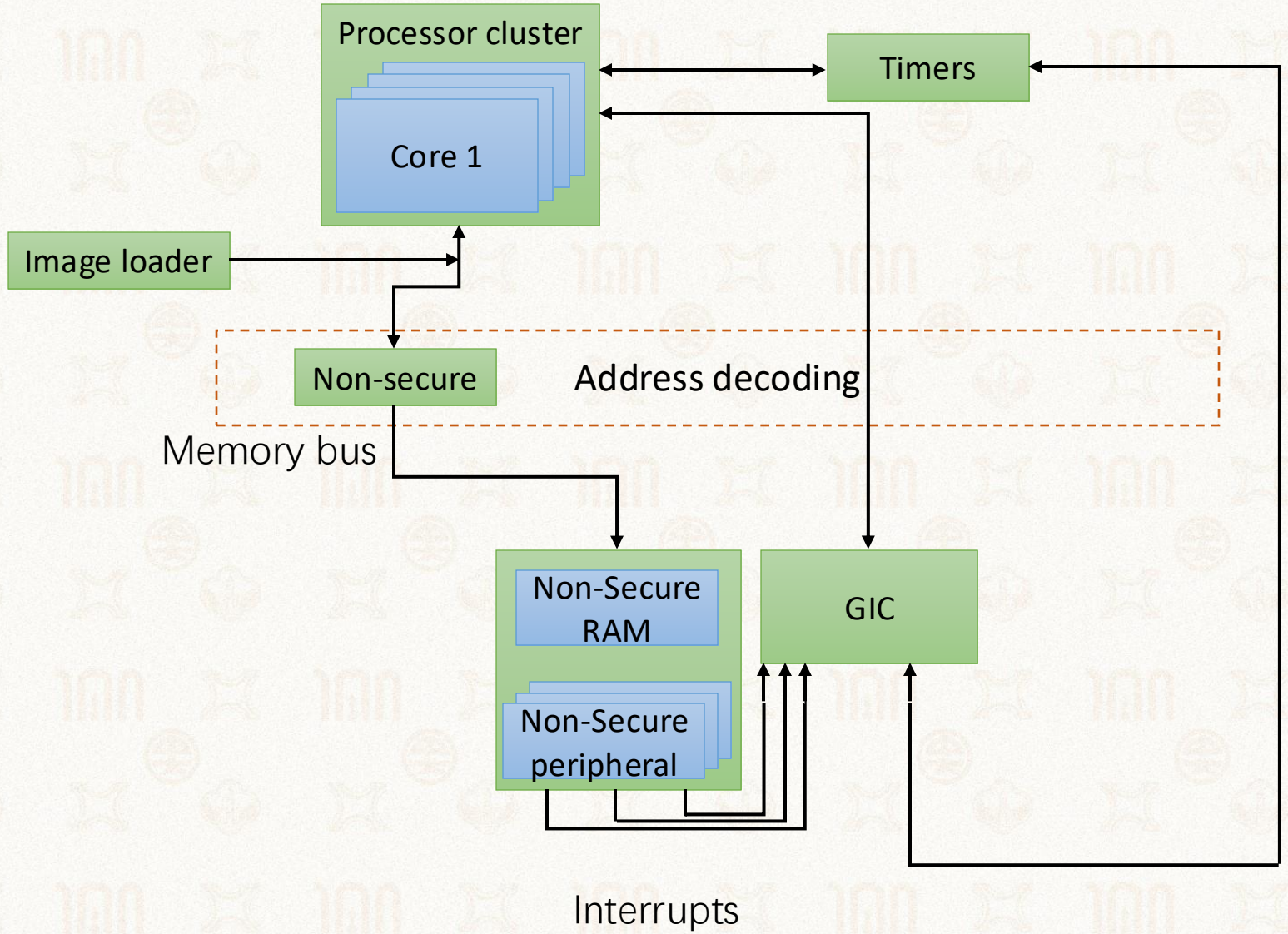
- 中断控制器
- 基本外设
  - 内存映射
  - 中断管理



GIC: Generic Interrupt Controller

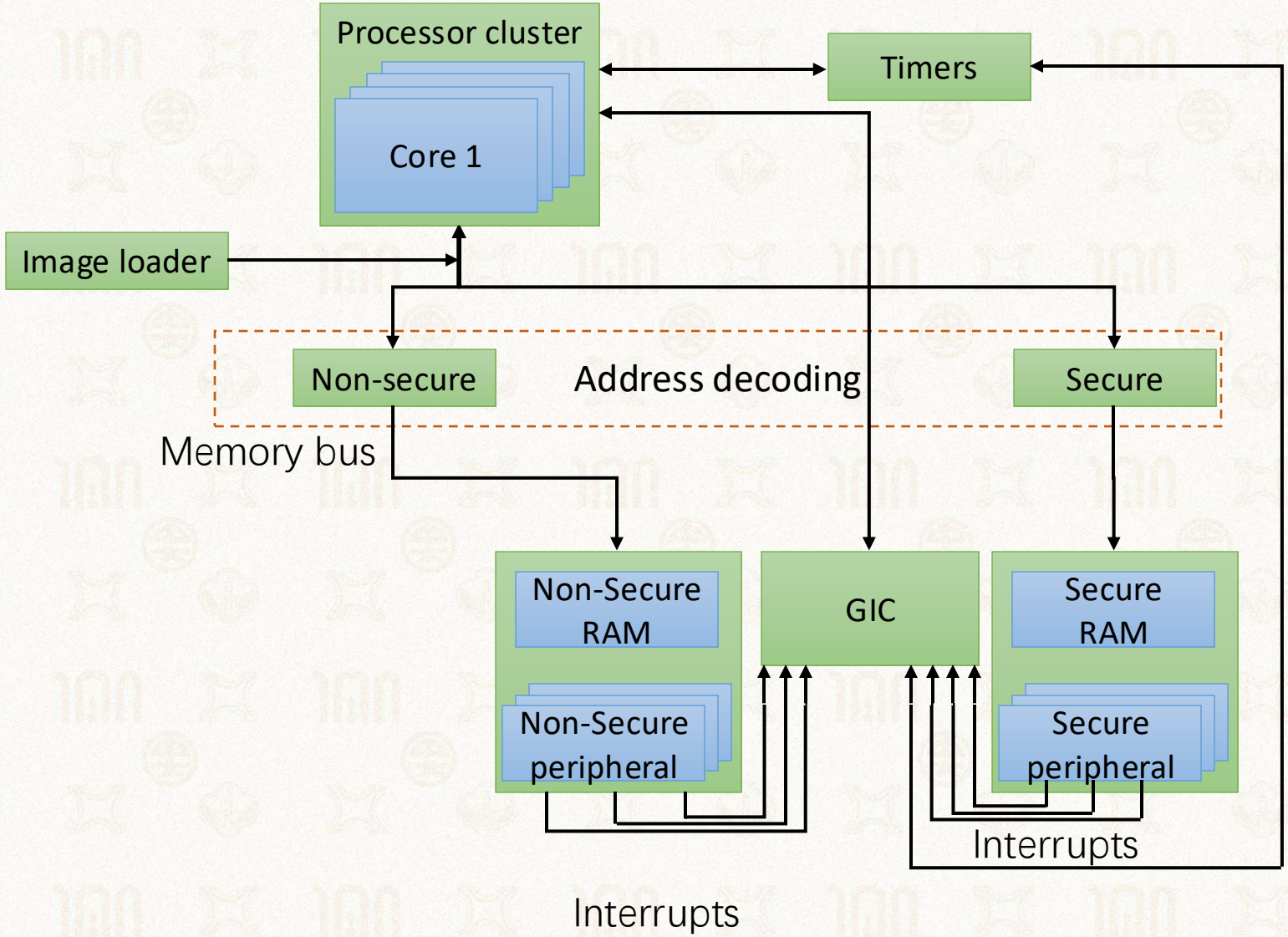
ARMv8基础平台

- 外设有多种计时器
  - 时钟
  - 看门狗计数器
  - 时间计数器
- 和中断响应控制器交互



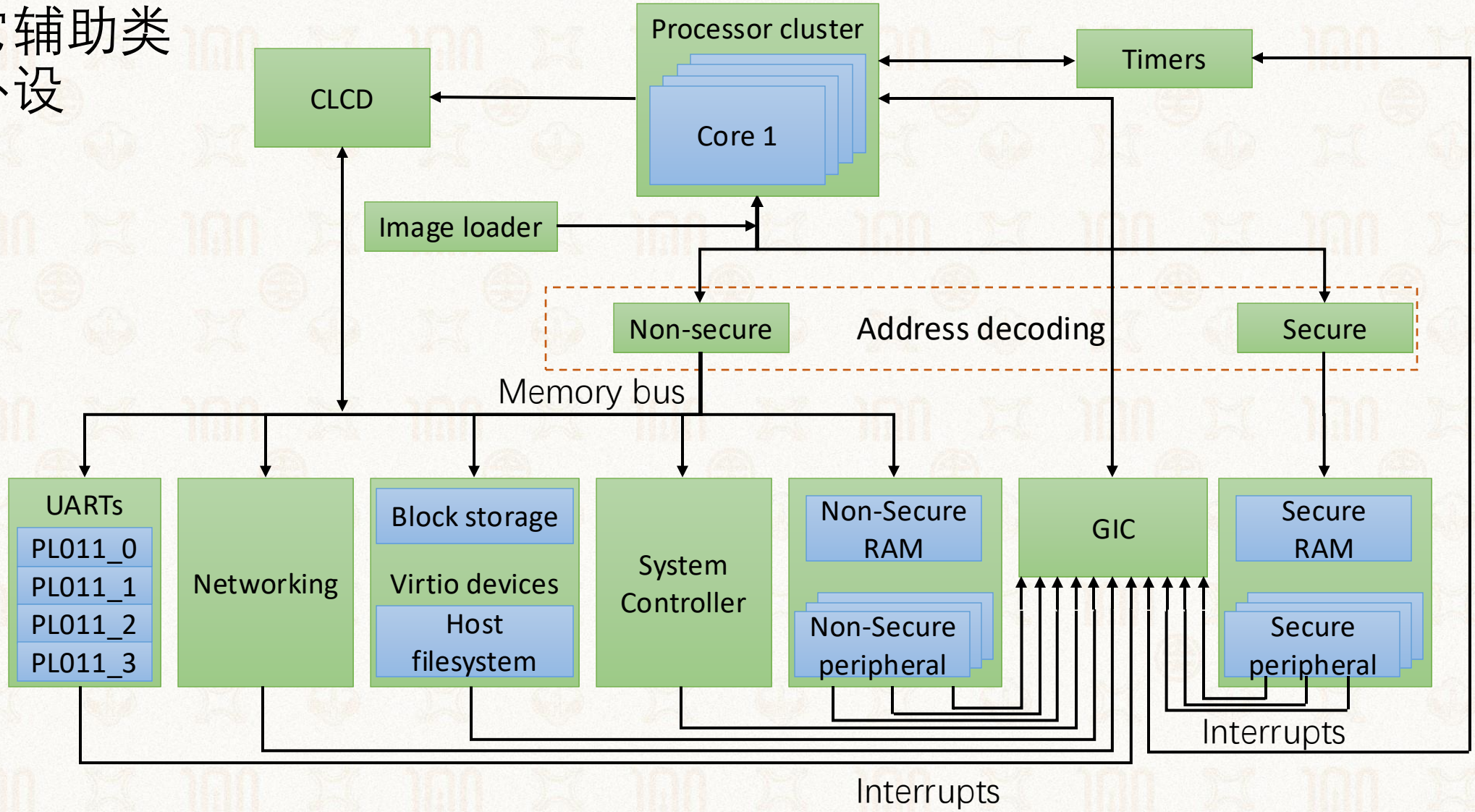
ARMv8基础平台

- ARMv8的核心特色：
  - 安全隔离模块
  - 小型独立的存储器、外设系统
  - 可信看门狗
  - 随机数生成器
  - 非易失性计数器(Non-volatile counters)
  - 根键存储器



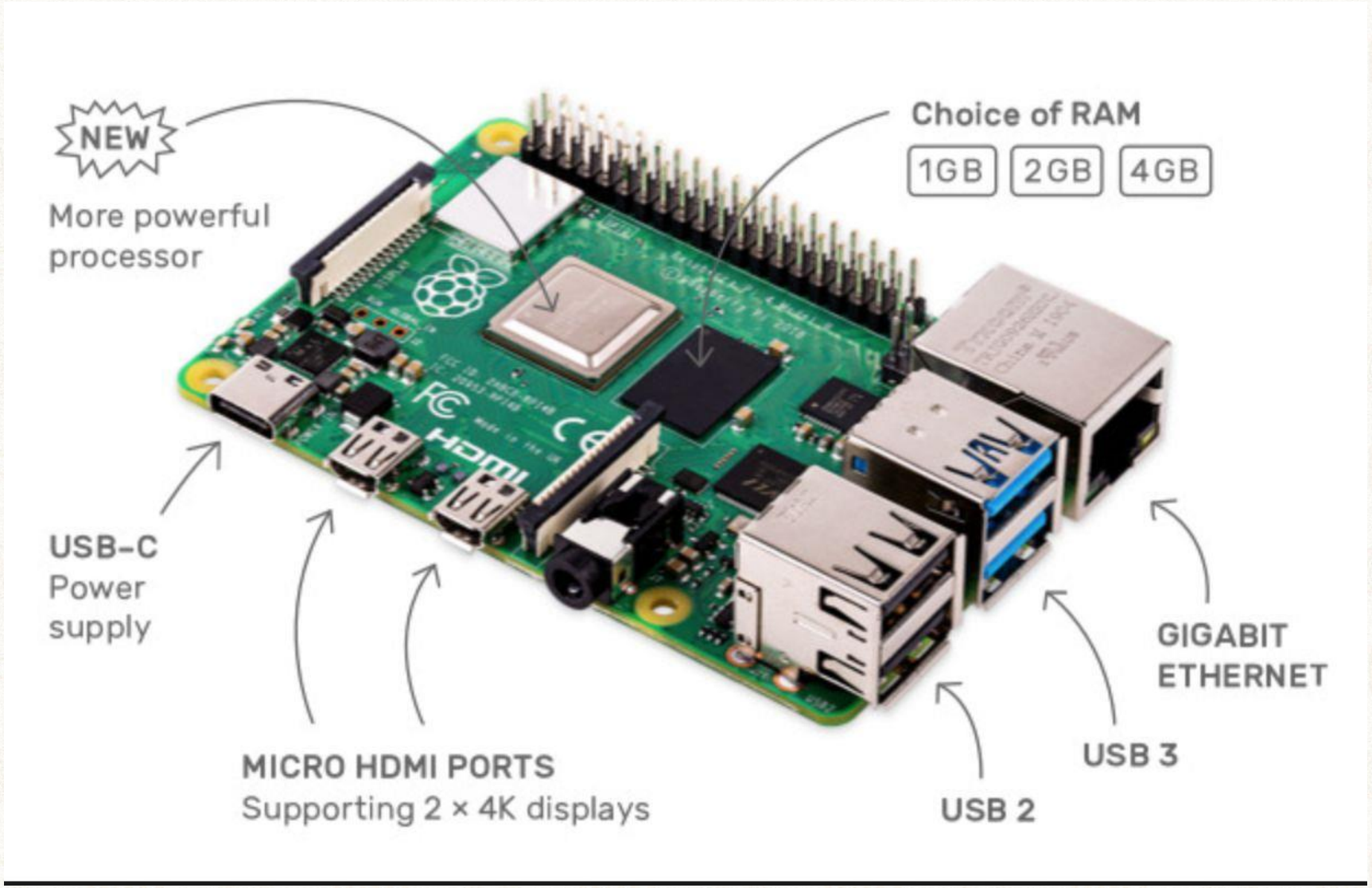
ARMv8基础平台

- 其它辅助类的外设



GIC: Generic Interrupt Controller UART: Universal Asynchronous Receiver/Transmitters CLCD: Color LCD

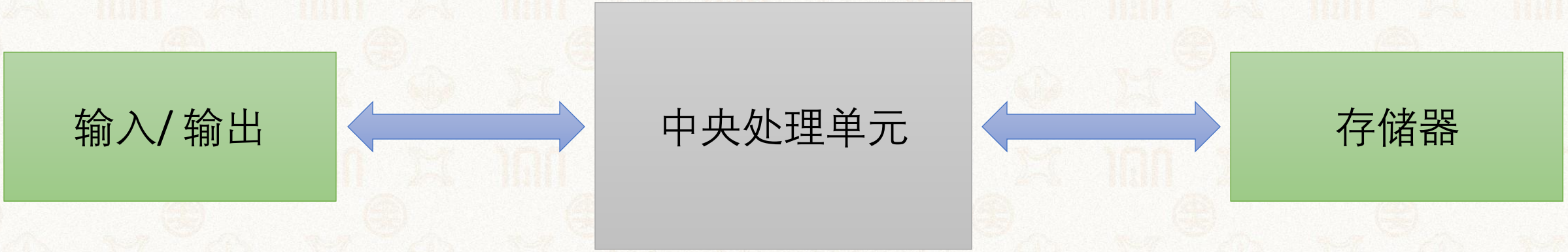
ARM开发板


NEW
More powerful processor
Choice of RAM
1GB
2GB
4GB
USB-C Power supply
MICRO HDMI PORTS
Supporting 2 × 4K displays
USB 2
USB 3
GIGABIT ETHERNET

大纲

- 为什么选择ARM
- 硬件执行逻辑
- ARM汇编语言
- 内存模型

冯诺依曼架构



- 输入/输出: 和设备之间交互数据
- CPU: 包括处理单元和控制单元
- 存储器:内存

冯诺依曼架构

CPU

```
for (;;) {
    next instruction
}
```

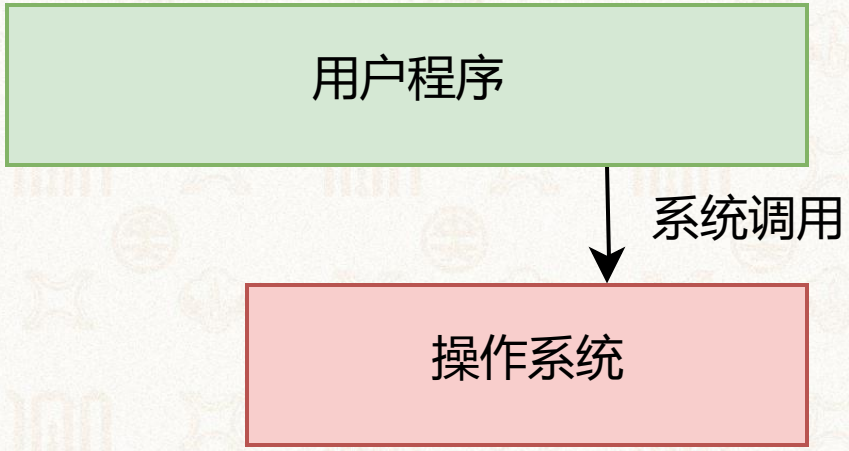
内存

```
Instruction
Instruction
Instruction
...
data
data
data
```

- CPU：解析指令
- 内存：存储指令和数据

应用程序的硬件执行环境

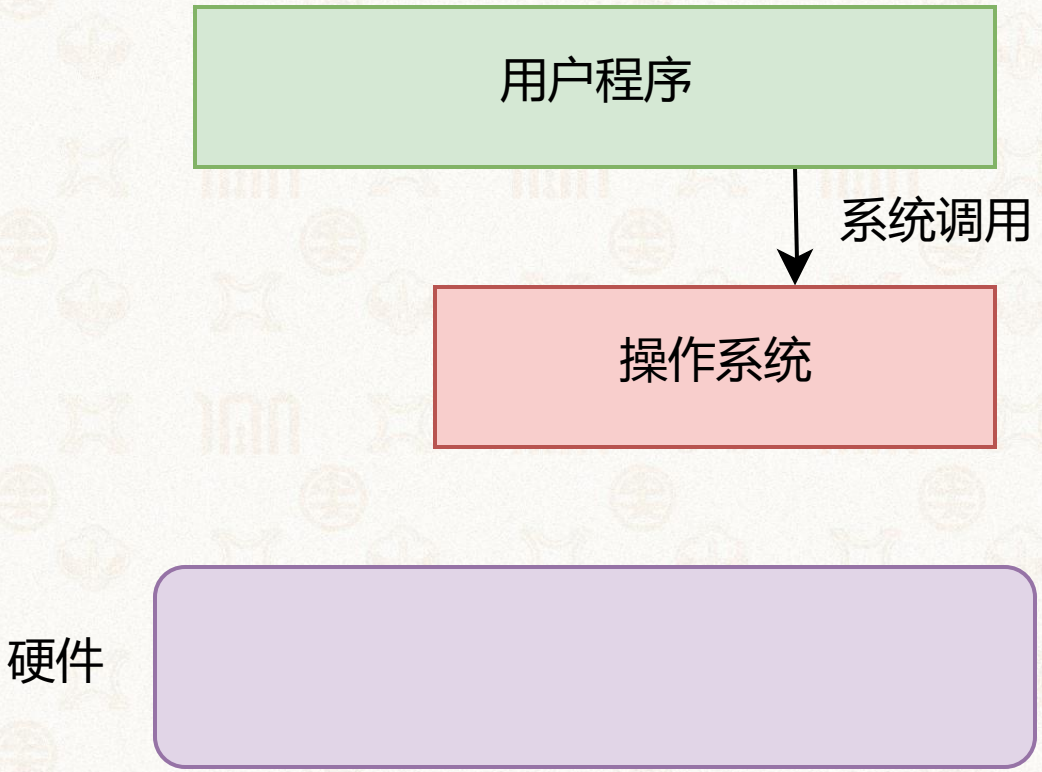
- 操作系统也是程序
  - 大量的CPU指令



应用程序的硬件执行环境

- 操作系统也是程序
  - 大量的CPU指令

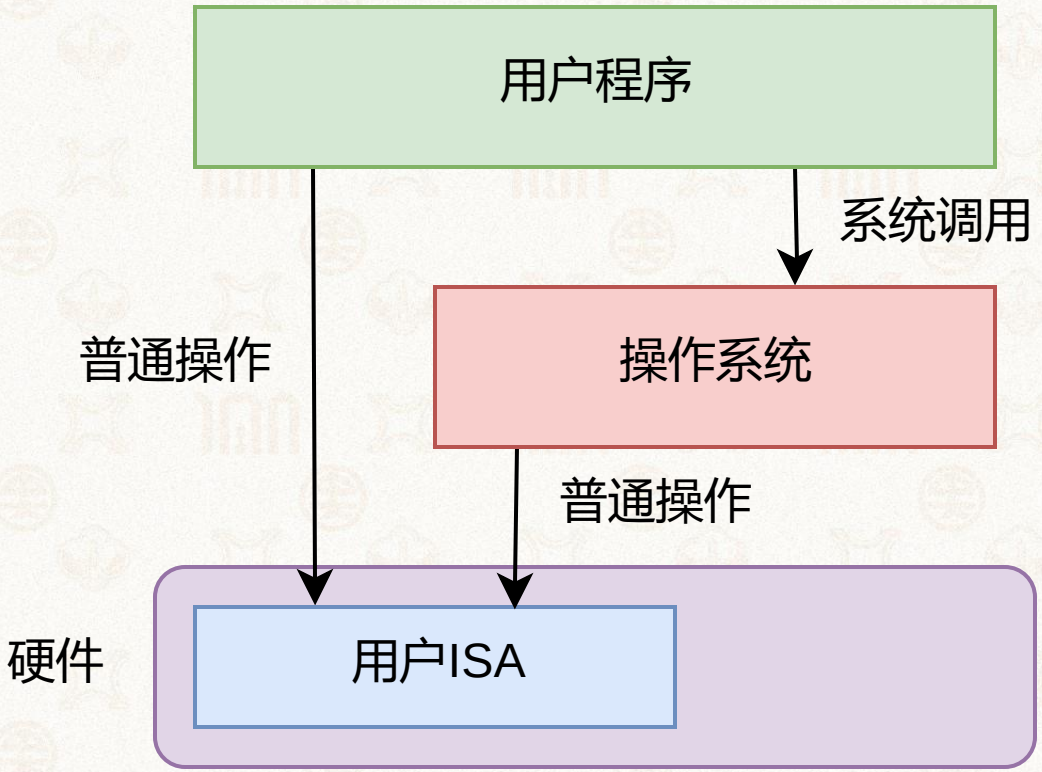
- 指令集架构(Instruction Set Architecture, ISA)
  - 与硬件绑定



应用程序的硬件执行环境

- 操作系统也是程序
  - 大量的CPU指令

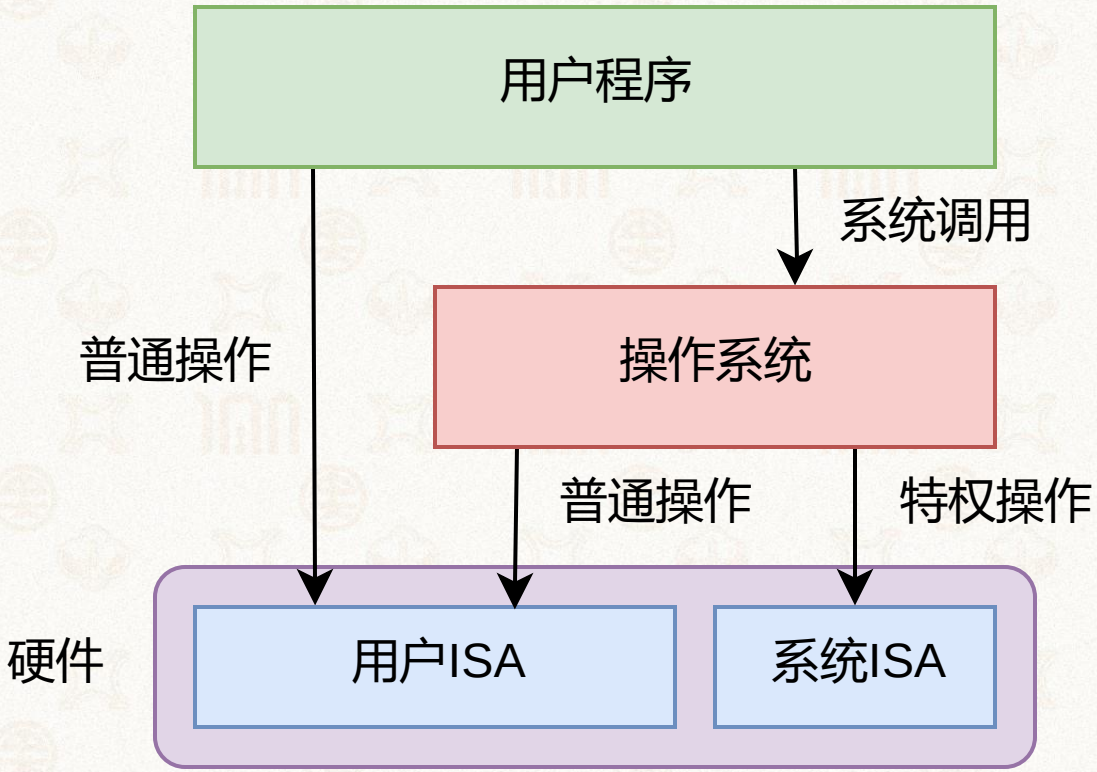
- 指令集架构(Instruction Set Architecture, ISA)
  - 与硬件绑定
  - 用户ISA示例
    - `mov x0, sp`
    - `add x0, x0, #1`



应用程序的硬件执行环境

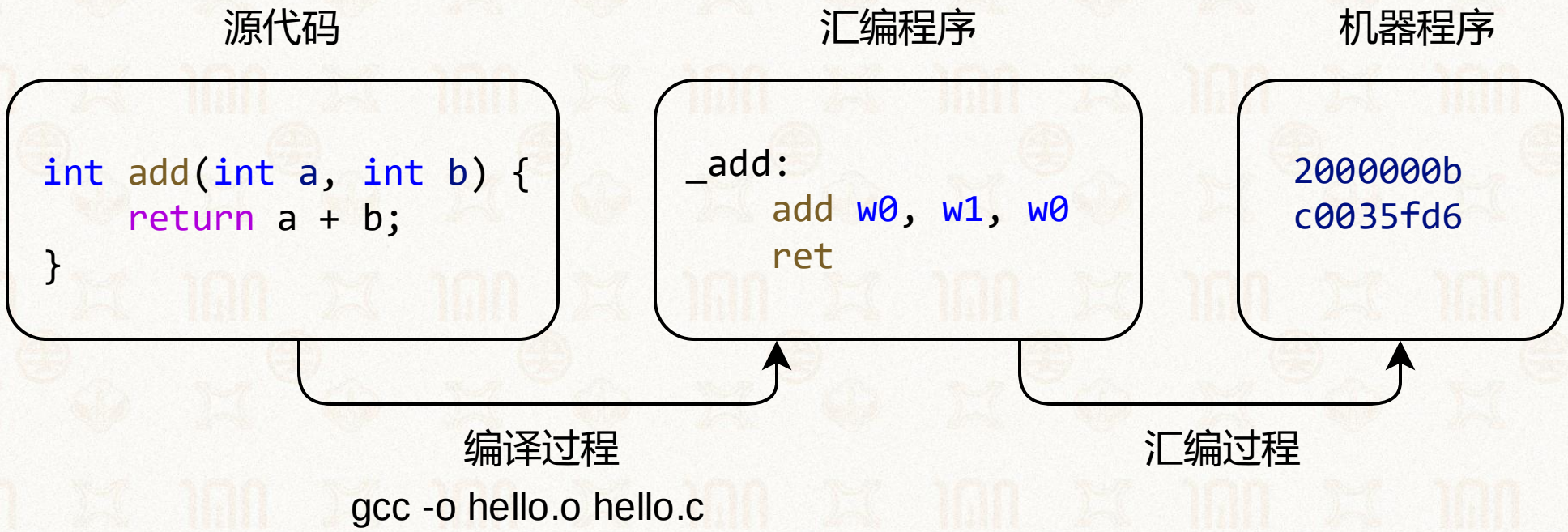
- 操作系统也是程序
  - 大量的CPU指令
  - 拥有一些特权指令

- 指令集架构(Instruction Set Architecture, ISA)
  - 与硬件绑定
  - 用户ISA示例
    - `mov x0, sp`
    - `add x0, x0, #1`
  - 系统ISA示例
    - `msr vbar_el1, x0`



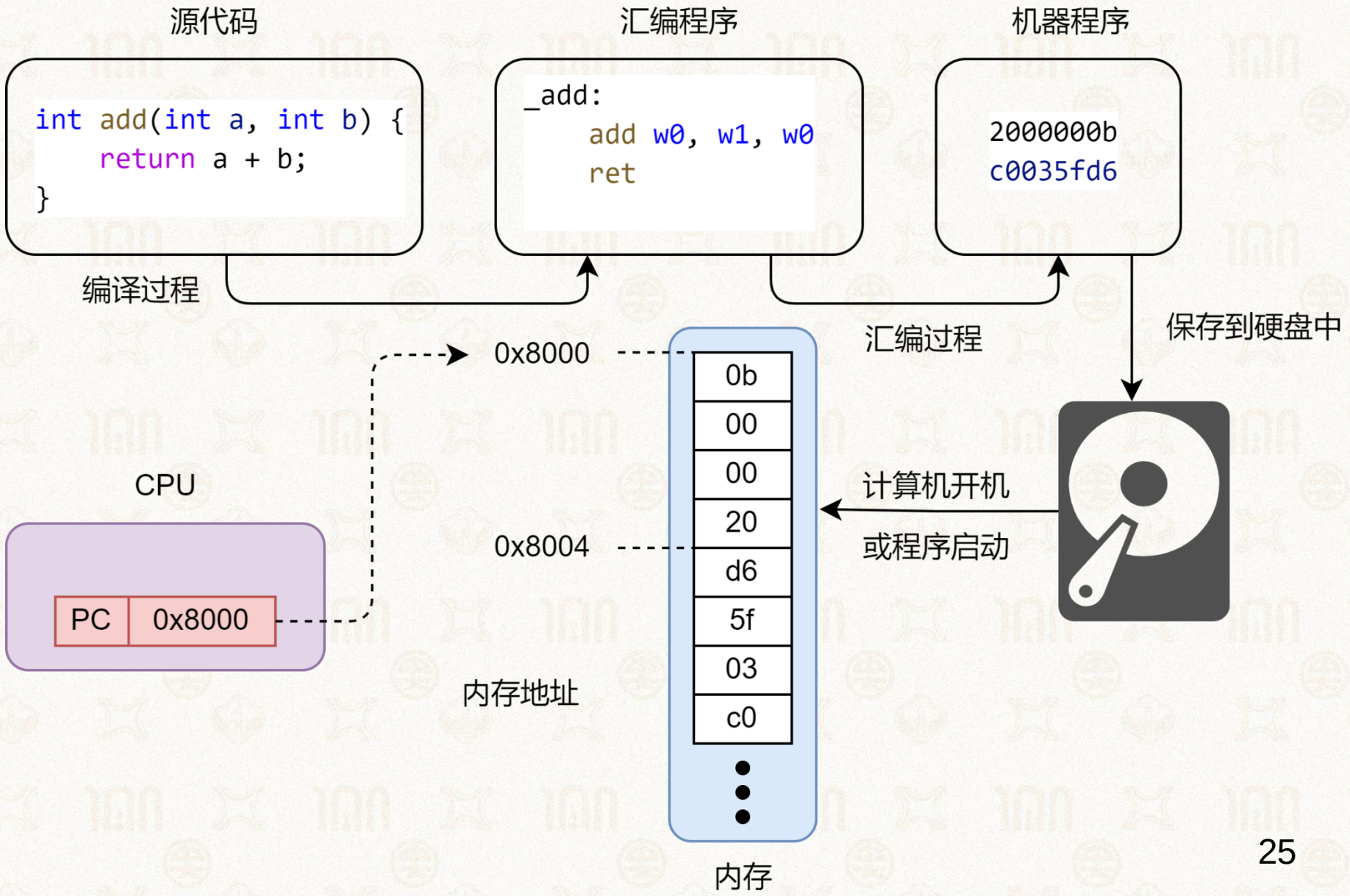
源代码和机器指令的关系

- 给人看的：源代码
- 给机器看的：机器码 (汇编程序)

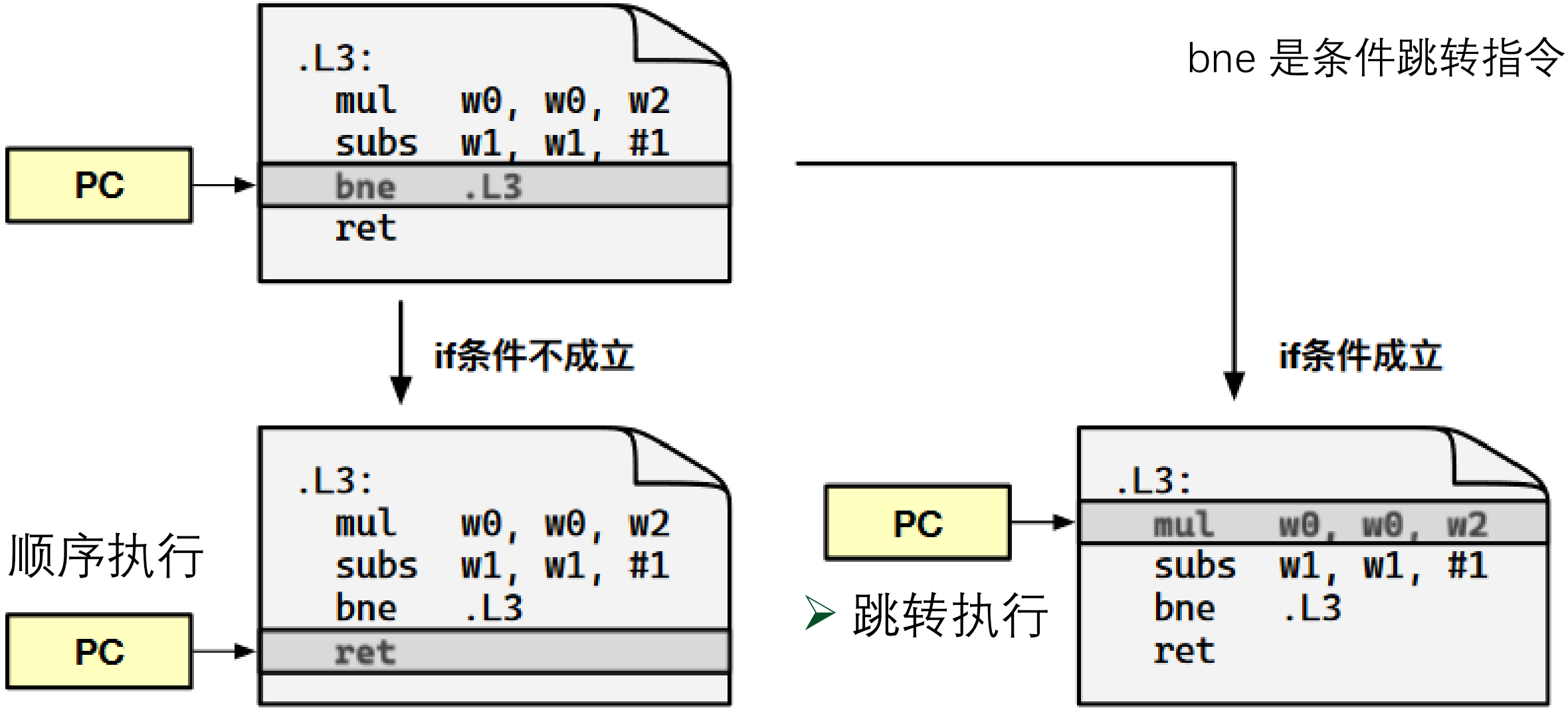


程序是怎么运行的

- 程序启动，将机器码从硬盘中复制到内存中
- CPU中的特殊寄存器：程序记数器(Program Counter, PC)
  - 记录下一条指令在内存中的位置
- 图中隐藏了一个知识点：小端模式



程序计数器的两种更新方式



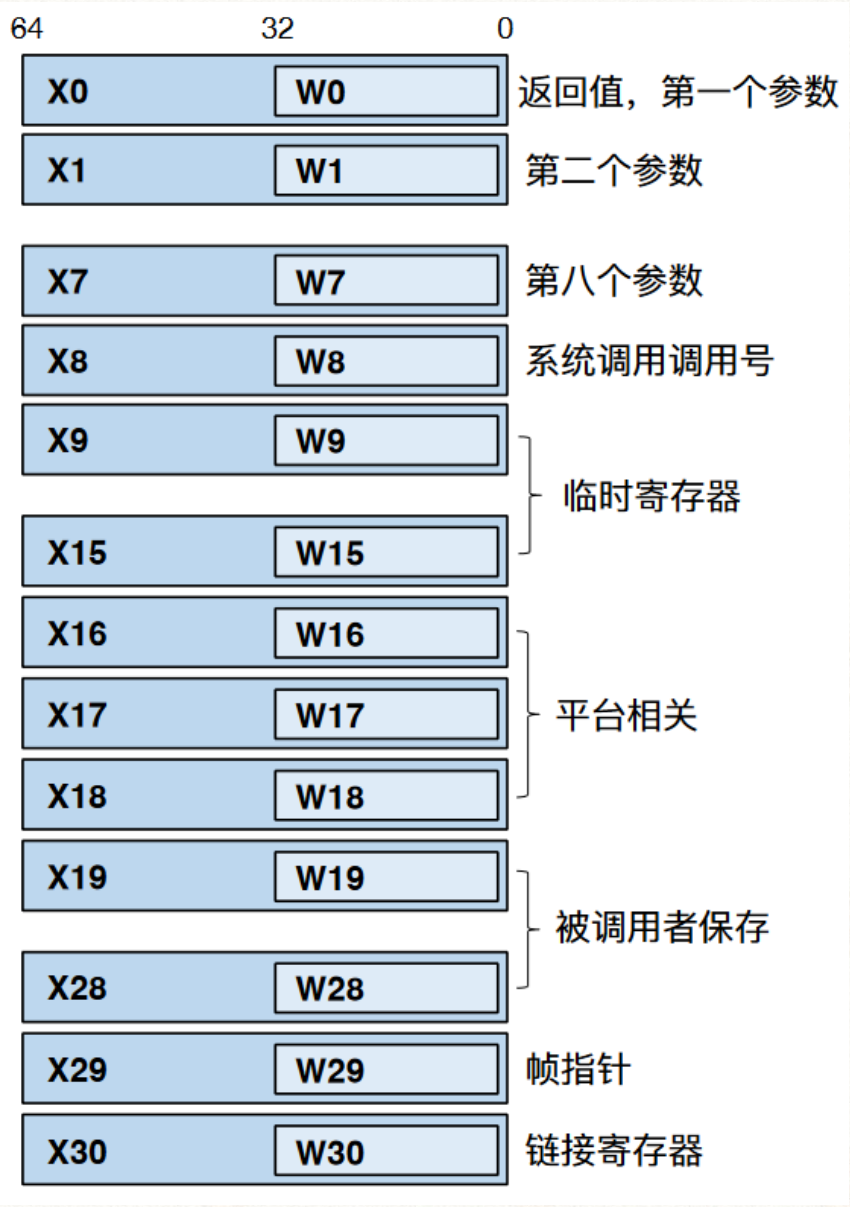
大纲

- 为什么选择ARM
- 硬件执行逻辑
- ARM汇编语言
- 内存模型

ARM中常用的寄存器

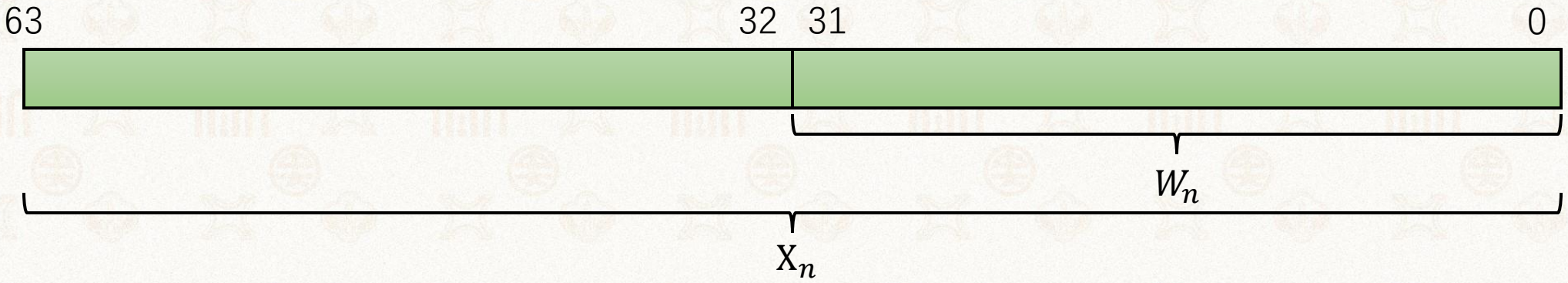
- 31个64位通用寄存器
  - X0-X30

- 注意几个常用的寄存器
  - X0, X1, X8, X29, X30



寄存器的多模态(register banks)

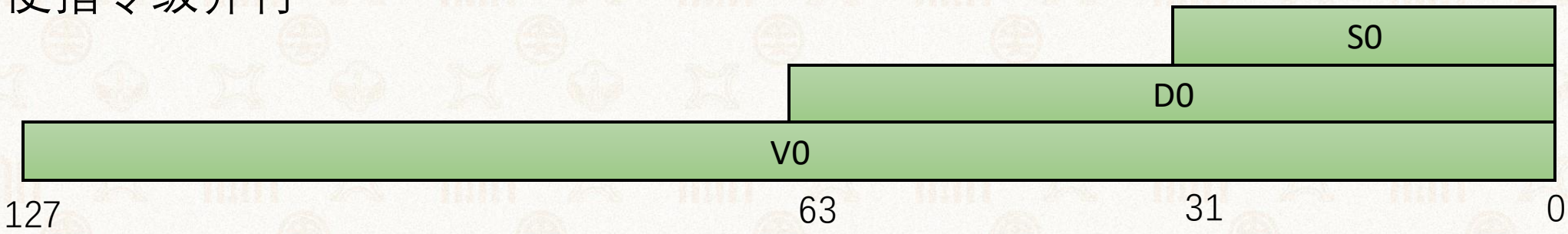
- 31个通用寄存器，每个寄存器有多种形式：
  - 32位形式：w0 – w30
  - 64位形式：x0 – x30
  - 由系统模式决定



寄存器的多模态(register banks)

- V_n：额外的寄存器表
  - 用于浮点计算、SIMD和加密操作
  - 32个128位寄存器
  - 32位形式：S_n
  - 64位形式：D_n

- 这样设计有什么好处？
  - 方便指令级并行

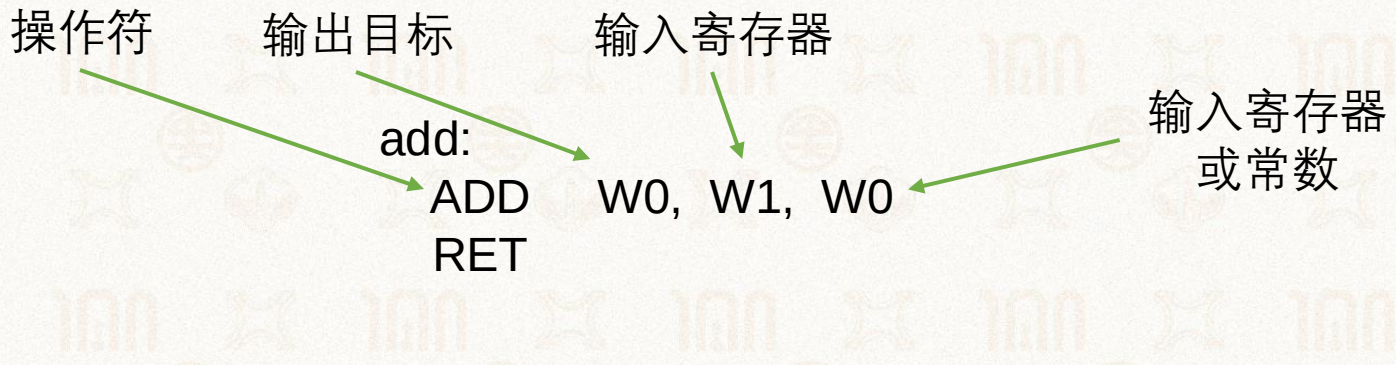


算术逻辑运算

```
uint32_t add (uint32_t a, uint32_t b)
{
    return a + b;
}
```

```
uint64_t sub (uint64_t a)
{
    return a - 1;
}
```

```
uint32_t mul (uint32_t a, uint32_t b, uint32_t c)
{
    return c + (a * b);
}
```




```
sub:
    SUB   X0, X0, #1
    RET
```

```
mul:
    MADD  W0, W1, W0, W2
```

复杂一点的例子：哈希函数

```c
int hash(int src) {
    src = ((src >> 16) ^ src) + 0xbeef;
    return (src >> 16) ^ src;
}
```

```asm
hash:
    eor w0, w0, w0, asr 16
    mov w1, 48879
    add w0, w0, w1
    eor w0, w0, w0, asr 16
    ret
```

ARM中常用的数据处理指令

指令类型	指令	效果	指令描述
算术运算指令	add $Rd, Rn, Op2$	$Rd \leftarrow Rn + Op2$	加法运算
	sub $Rd, Rn, Op2$	$Rd \leftarrow Rn - Op2$	减法运算
	mul $Rd, Rn, Op2$	$Rd \leftarrow Rn * Op2$	无符号乘法运算
	div $Rd, Rn, Op2$	$Rd \leftarrow Rn/Op2$	无符号除法运算
	neg Rd, Rn	$Rd \leftarrow -Rn$	取相反数
逻辑运算指令	and $Rd, Rn, Op2$	$Rd \leftarrow Rn\&Op2$	按位与
	orr $Rd, Rn, Op2$	$Rd \leftarrow Rn\|Op2$	按位或
	eor $Rd, Rn, Op2$	$Rd \leftarrow Rn \oplus Op2$	按位异或
	mvn Rd, Rn	$Rd \leftarrow\sim Rn$	按位取反
移位指令	asr $Rd, Rn, Op2$	$Rd \leftarrow Rn >>_A Op2$	算术右移
	lsl $Rd, Rn, Op2$	$Rd \leftarrow Rn << Op2$	逻辑左移
	lsr $Rd, Rn, Op2$	$Rd \leftarrow Rn >>_L Op2$	逻辑右移
	ror $Rd, Rn, Op2$	$Rd \leftarrow Rn >>_R Op2$	循环右移
数据搬移指令	mov $Rd, Op2$	$Rd \leftarrow Rn$	数据移动

单选题 1分

设置

假设 X0 = 0x123456789ABCDEF0，执行 “LSR X0, X0, #4” 指令后，X0 的值为（ ）

A 0x123456789ABCDEF0

B 0x0123456789ABCDEF

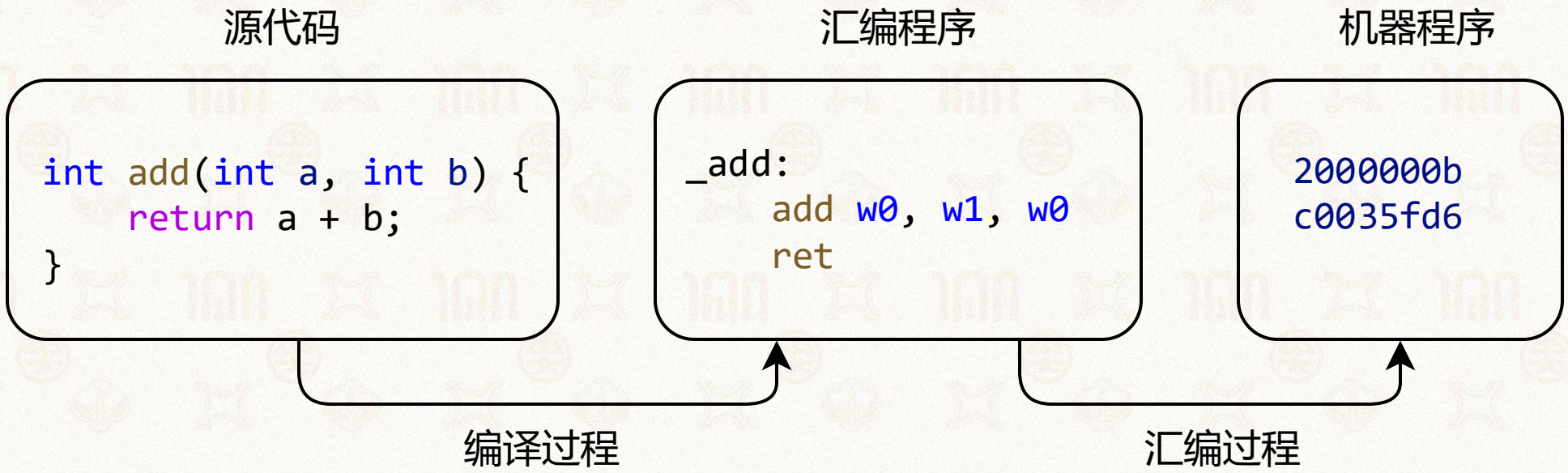
C 0xF0123456789ABCDE

D 0x123456789ABCDE0F

提交

CPU 视角下的内存

- 寄存器里的数据都是哪来的?
  - W0存有第一个参数的值
  - 运行结束后，W0存有返回值
  - 数值都是从内存里读来的，也要写到内存中去
  - CPU需要指令，完成对内存的读和写



存取指令

- 两个基本指令，用于:
  - 取：LDR (load)
  - 存：STR (store)

指令类型	指令	效果	指令描述
加载指令	ldr $R, addr$	$R \leftarrow mem[addr : addr + R_s]$	从内存加载数据到寄存器
	ldp $R1, R2, addr$	$R1, R2 \leftarrow mem[addr : addr + R1_s + R2_s]$	从内存加载数据到两个寄存器
存储指令	str $R, addr$	$R \rightarrow mem[addr : addr + R_s]$	将寄存器中的数据存储到内存
	stp $R1, R2, addr$	$R1, R2 \rightarrow mem[addr : addr + R1_s + R2_s]$	将两个寄存器中的数据存储到内存

- 由寄存器类型、名字决定存取数据长度

LDR W0, [<address>]	从地址处读取32位数据
LDR X0, [<address>]	从地址处读取64位数据
STRB W0, [<address>]	把寄存器W0的低字节(8位)存到地址处
STRH W0, [<address>]	把寄存器W0的低半字(16位)存到地址处
STRW X0, [<address>]	把寄存器W0的低字(32位)存到地址处

B: Byte (8位)，H: Half word (16位)，W: word (32位)

例子：交换函数

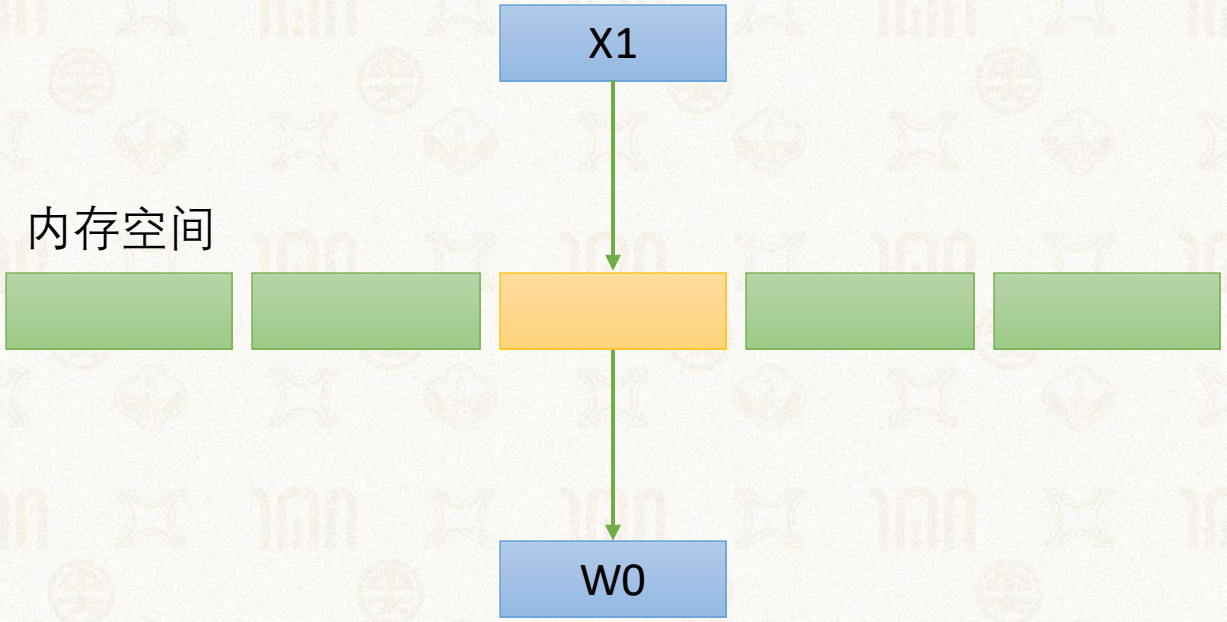
```c
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```asm
swap:
    ldr w3, [x1]
    ldr w2, [x0]
    str w3, [x0]
    str w2, [x1]
    ret
```

寻址方式

- 基址寻址方式(Base register addressing)

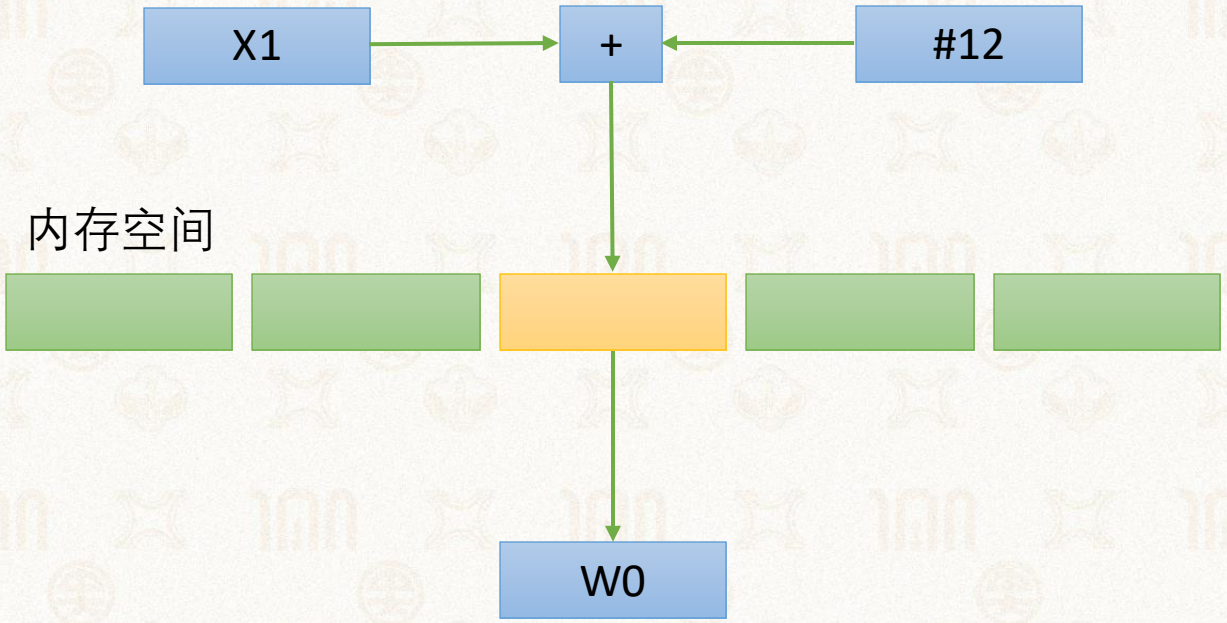
LDR W0, [X1]



寻址方式

- 变址寻址方式(offset addressing)

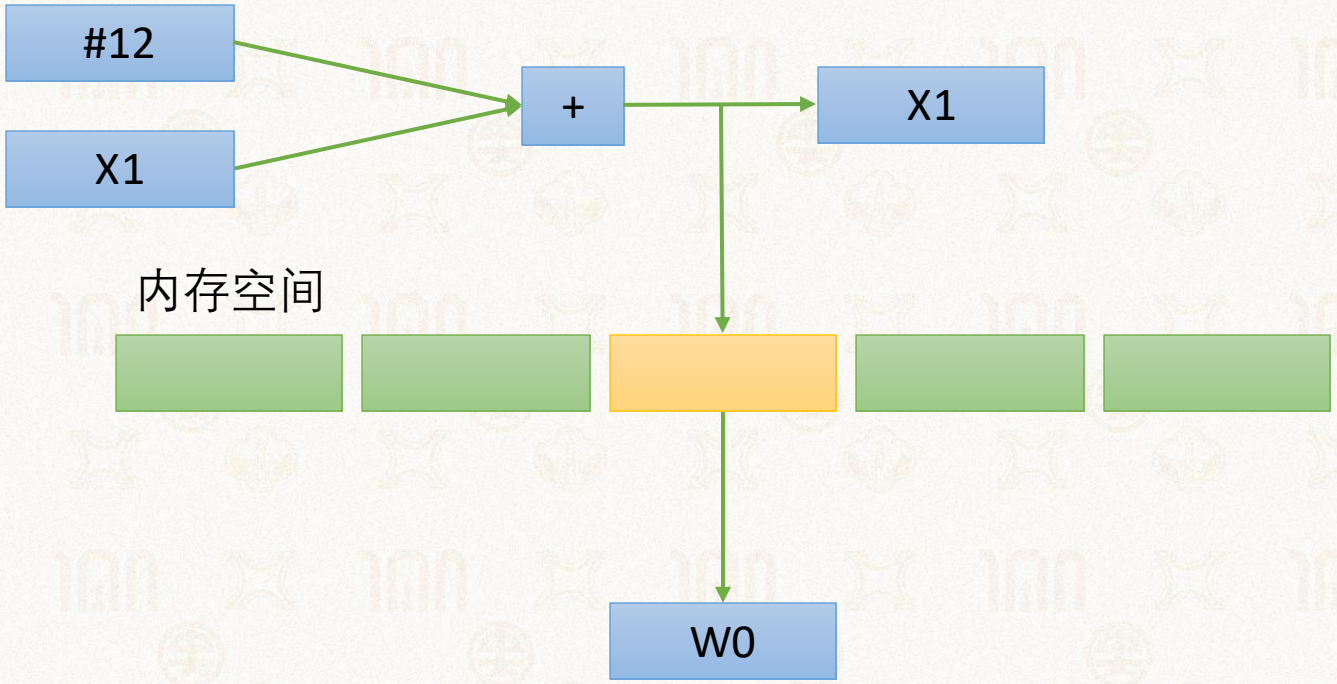
LDR W0, [X1, #12]



寻址方式

- 前变址寻址方式(pre-index addressing)：X1先更新再寻址

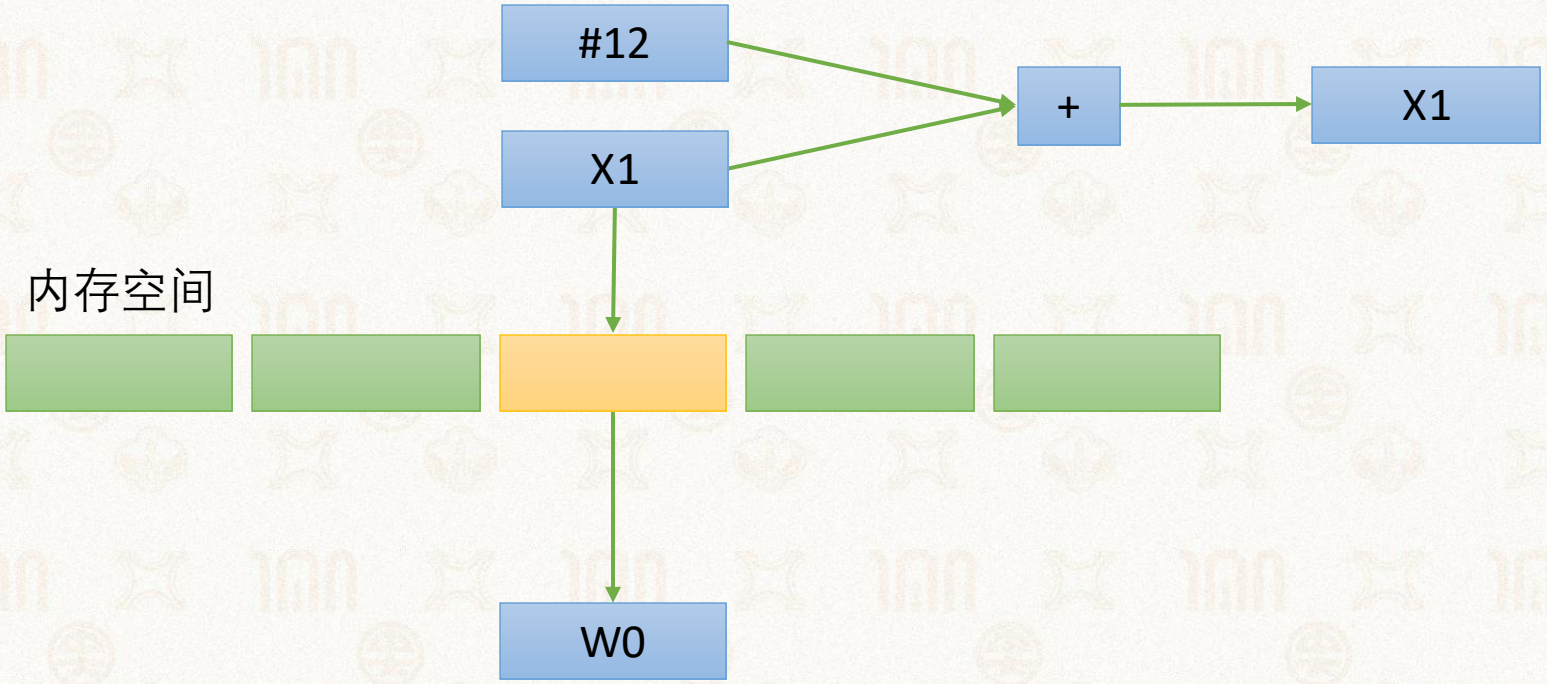
LDR W0, [X1, #12]!



寻址方式

- 后变址寻址方式(post-index addressing)：X1先寻址再更新

LDR　W0,　[X1],　#12



条件分支与条件码

```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
```

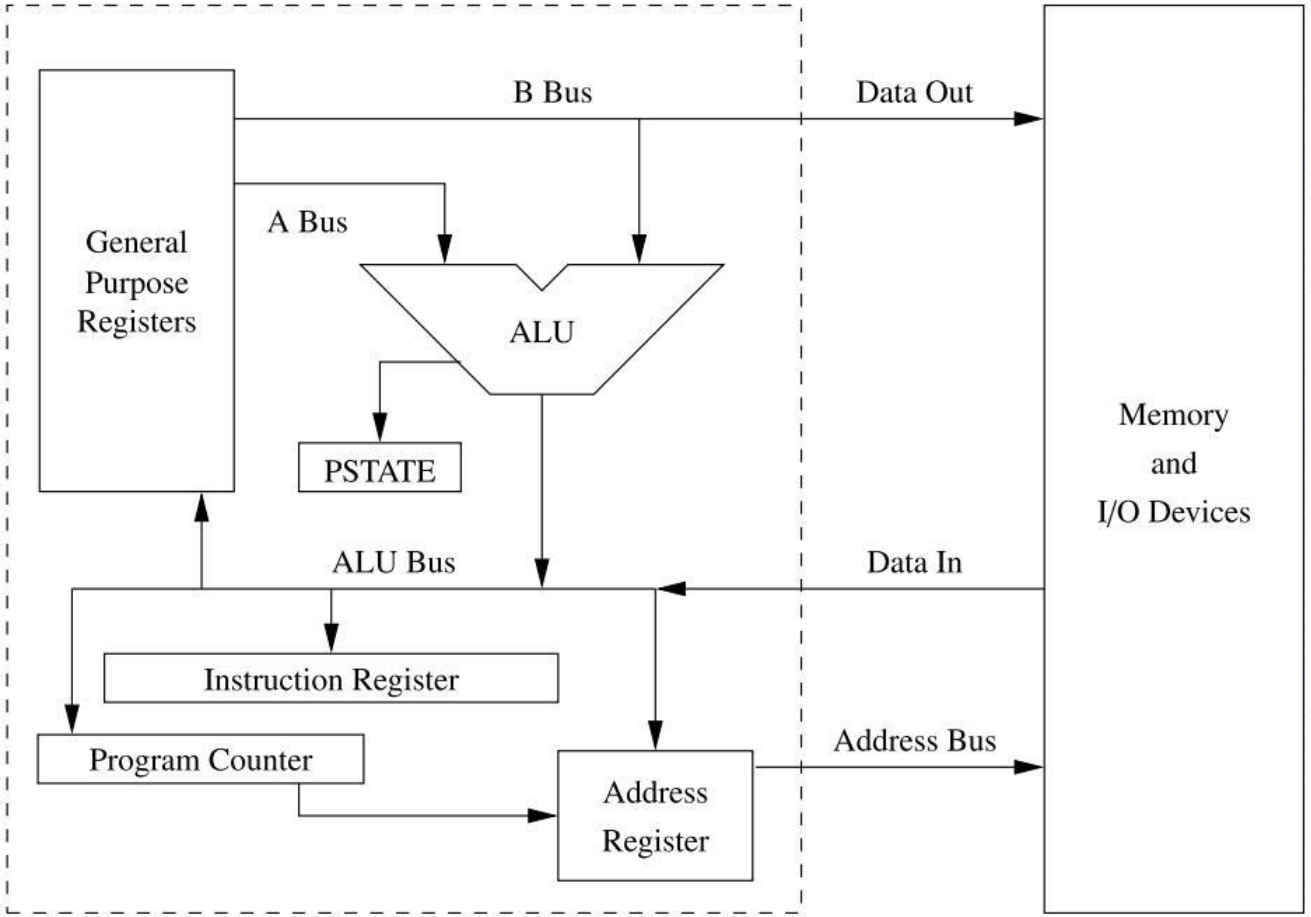
- 根据前一条的状态决定是否跳转
  - bne: 不等于零时跳转
- 根据本条语句的状态决定是否跳转
  - cbz: 寄存器值为0时跳转
- 助记：
  - b: branch
  - ne: not equal
  - c: compare
  - z: zero

条件分支与条件码

- 根据前一条的状态决定是否跳转
  - 上一条的状态在哪?
  - 保存在了特殊寄存器里：状态寄存器(PSTATE)的条件码

条件码：N（Negative）、Z（Zero）、C（Carry）、V（Overflow）

条件	含义	对应的条件码（NZCV）
EQ	相等	Z = 1
NE	不等	Z = 0
MI	负数	N = 1
PL	非负数	N = 0
HI	无符号大于	C = 1 且 Z = 0
LO	无符号小于	C = 0
LS	无符号小于或等于	C = 0 或 Z = 1
GE	有符号大于或等于	N = V
LT	有符号小于	N != V
GT	有符号大于	Z = 0 且 N = V
LE	有符号小于或等于	Z = 1 或 N != V



函数的调用、返回与栈

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

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

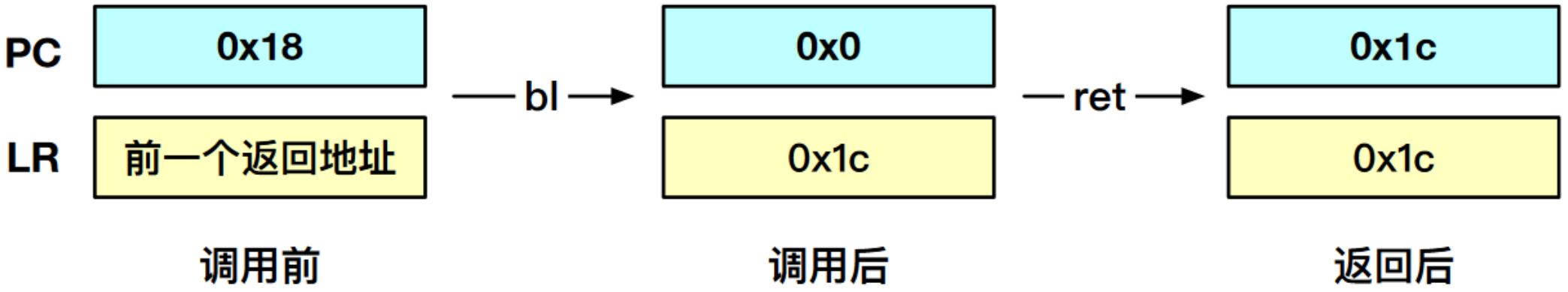
```asm
square:
    mul w0, w0, w0
    ret
cube:
    stp x29, x30, [sp, -32]!
    mov x29, sp
    str x19, [sp, 16]
    mov w19, w0
    bl square
    mul w0, w0, w19
    ldr x19, [sp, 16]
    ldp x29, x30, [sp], 32
    ret
```

- 被调函数如何得知结束后应该返回哪里？
- bl 执行会将返回地址写入“返回地址寄存器”
  - x30寄存器
  - 别名：LR(Link Register)

函数的调用、返回与栈

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

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

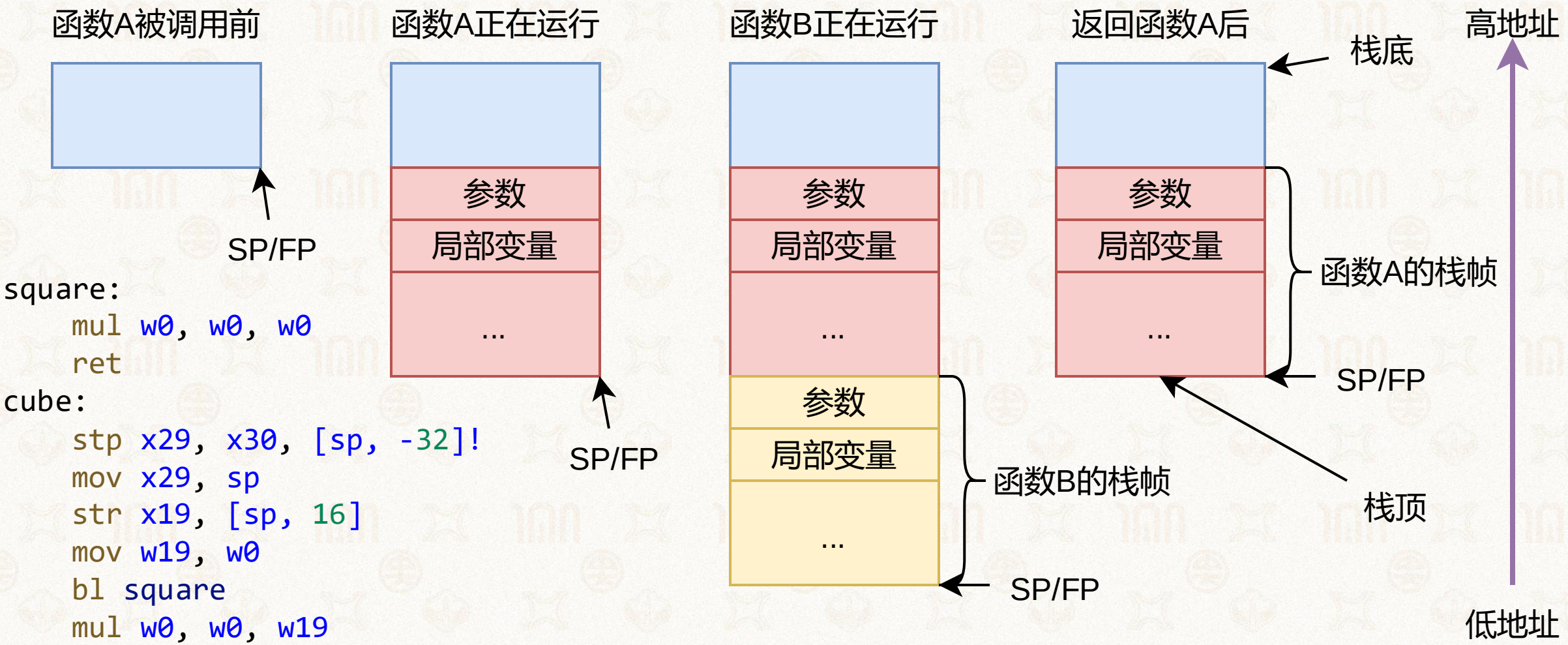
```
0000000000000000 <square>:
    0: 00 7c 00 1b    mul  w0, w0, w0
    4: c0 03 5f d6    ret

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



运行时栈：保存函数中的局部状态

- 函数有局部变量需要保存，特别是递归函数
- 每个函数拥有的连续内存空间称为函数的栈帧（Stack Frame）
- 栈帧在内存中的起始位置称为帧指针(Frame Pointer, FP)
  - 存于x29 通用寄存器

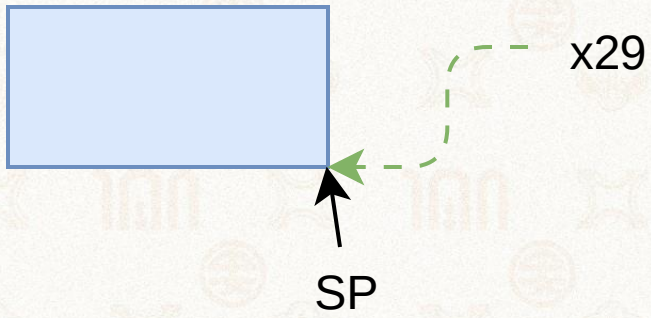
运行时栈：保存函数中的局部状态




```
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
```

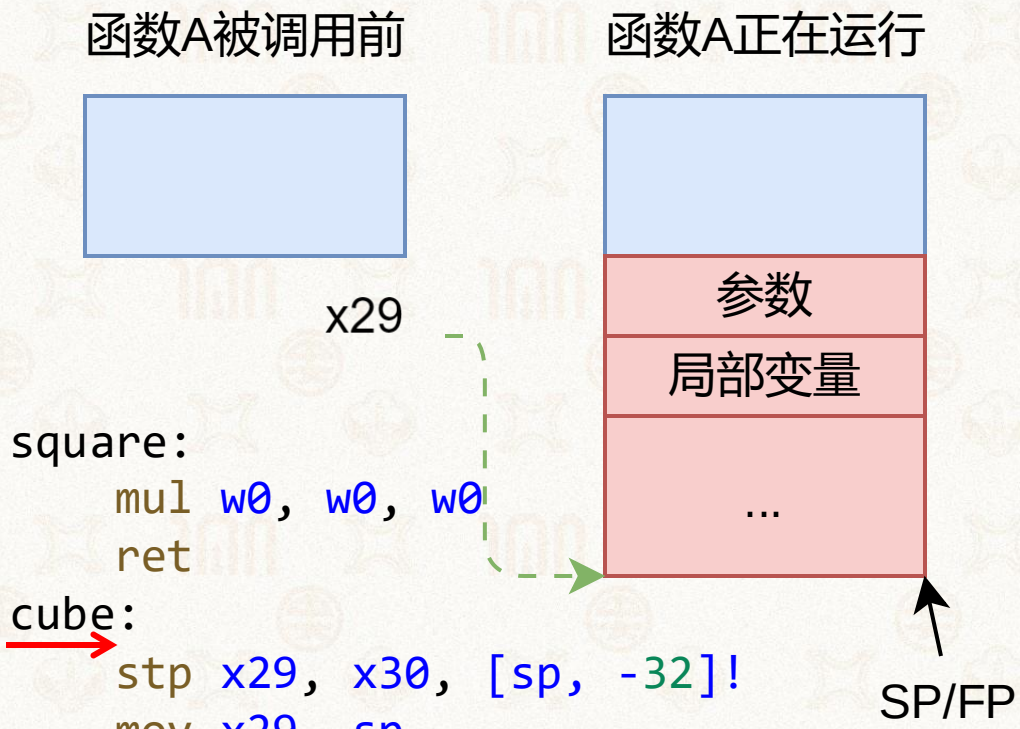
运行时栈：保存函数中的局部状态

函数A被调用前




```
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
```

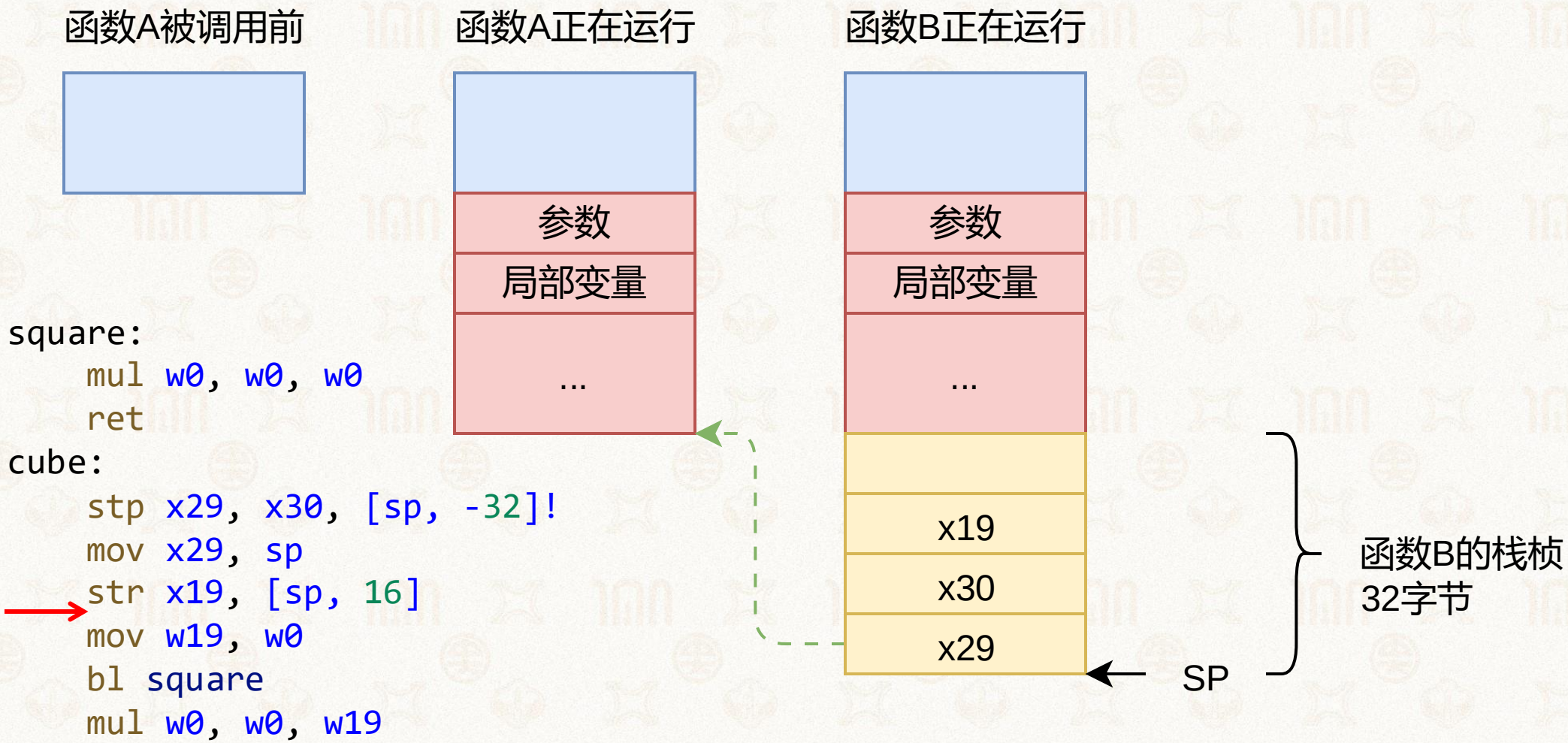
运行时栈：保存函数中的局部状态




```
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
```

要开始调用square函数了，
需要把x29, x30保存起来

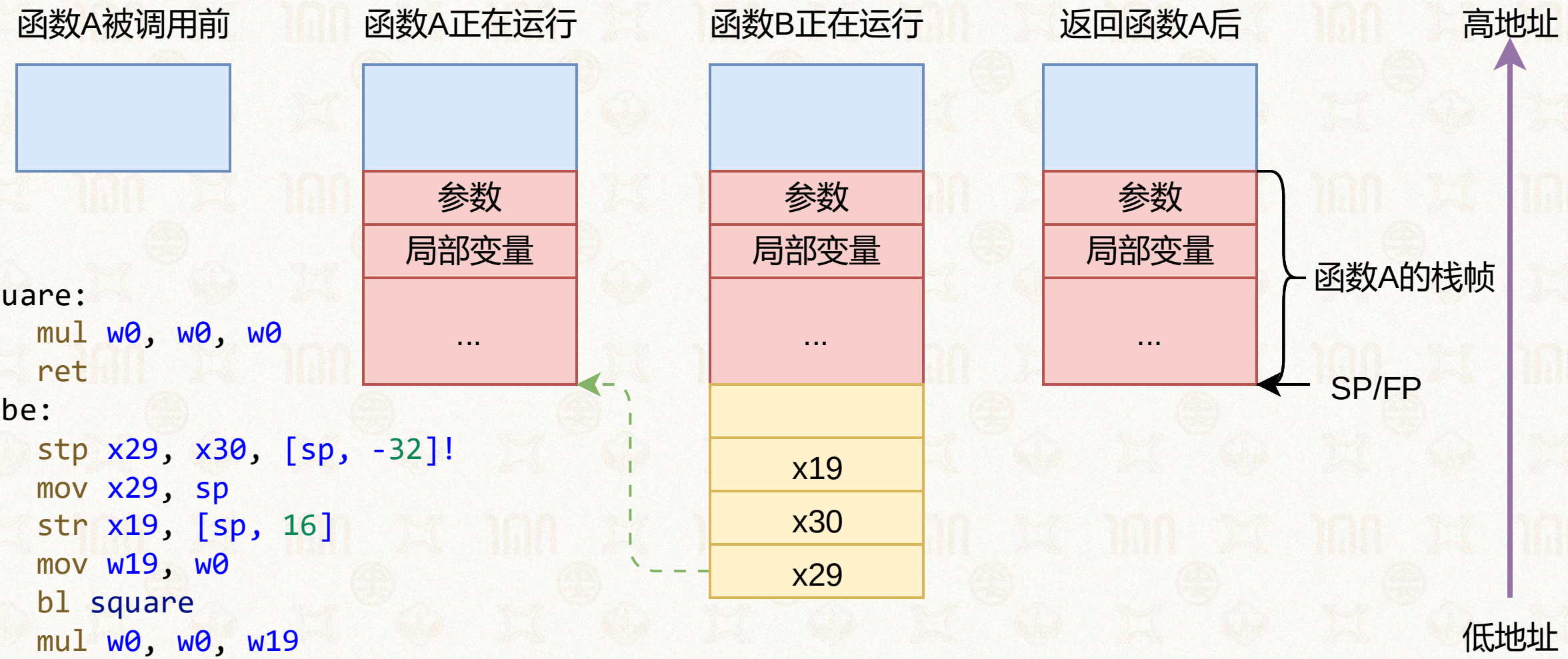
运行时栈：保存函数中的局部状态




```
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
```

把x19保存一下，因为接下来square可能要用到x19，值会被覆盖

运行时栈：保存函数中的局部状态



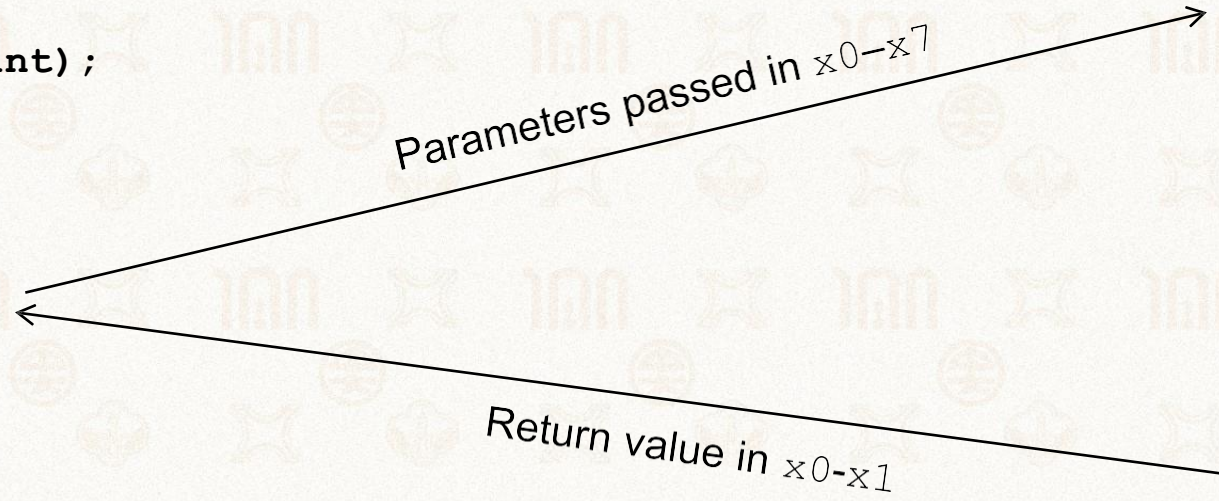

```
square:
    mul w0, w0, w0
    ret
cube:
    stp x29, x30, [sp, -32]!
    mov x29, sp
    str x19, [sp, 16]
    mov w19, w0
    bl square
    mul w0, w0, w19
    ldr x19, [sp, 16]
    ldp x29, x30, [sp], 32
    ret
```

通放释放函数B(square)的栈帧，可以恢复函数A(cube)运行时x29, x30的状态

过程调用准则(Procedure Call Standard, PCS)

- 规定在过程(函数)调用过程中使用寄存器的标准

```
extern int foo(int, int);

int main(void)
{
   ...
   a = foo(b, c);
   ...
}
```




```
...
...
...
...
...
...
...
...
...
RET
ENDP
```

Must preserve
X19 - X29

Can corrupt:
X0 - X18

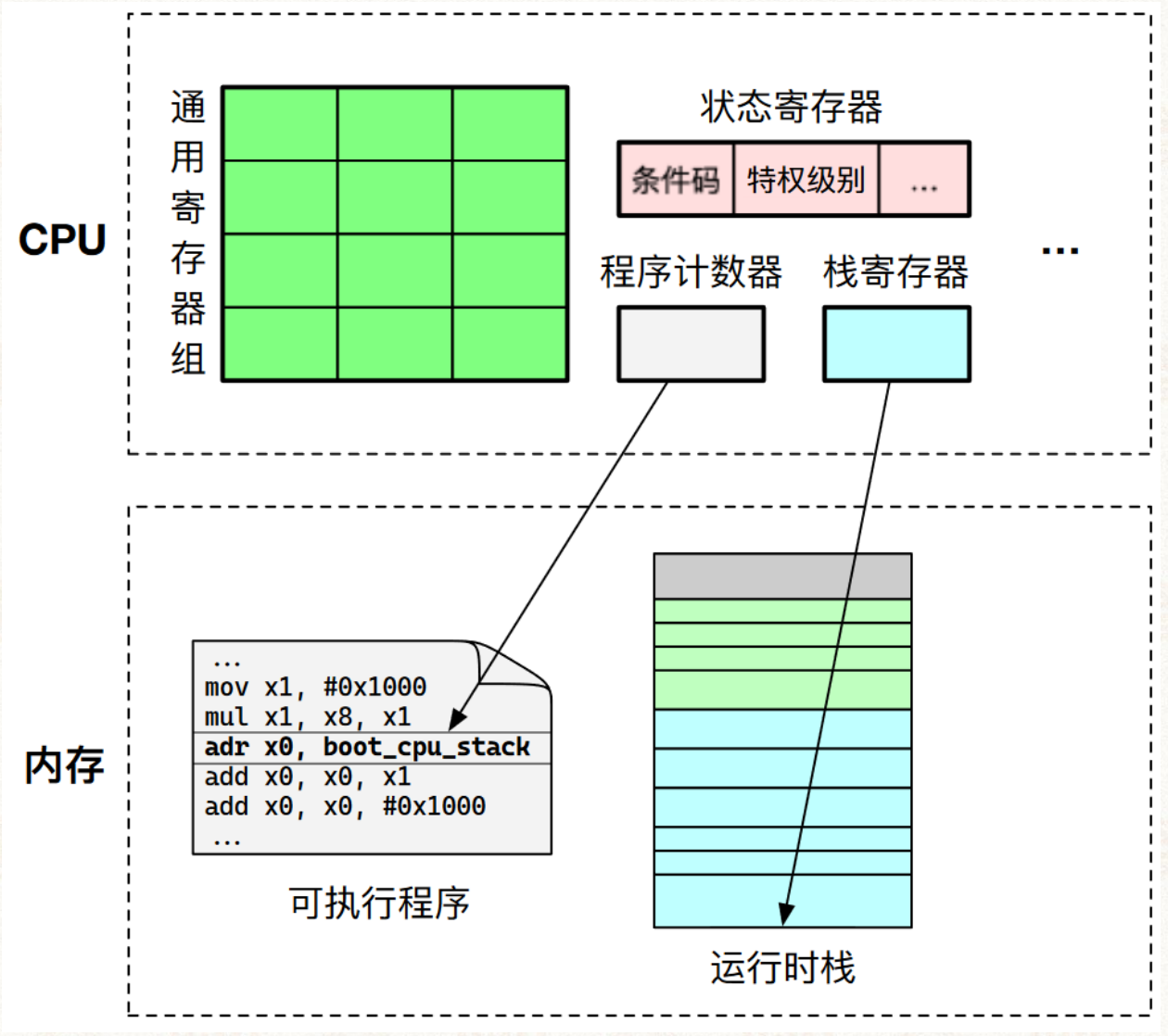
Return address: **X30 (LR)**

过程调用准则(Procedure Call Standard, PCS)

<table>
<tr><th>X0-X7</th><th>X8-X15</th><th>X16-X23</th><th>X24-X30</th></tr>
<tr><td rowspan="5">参数 / 返回结果
(X0-7)</td><td>XR (X8)</td><td>IP0 (X16)</td><td rowspan="3">被调用过程
使用的
(X24-28)</td></tr>
<tr><td rowspan="4">可以被修
改的寄存
器
(X9-15)</td><td>IP1 (X17)</td></tr>
<tr><td>PR (X18)</td></tr>
<tr><td rowspan="3">被调用过程
使用的
(X19-23)</td><td>FP (X29)</td></tr>
<tr><td>LR (X30)</td></tr>
</table>

- IP0 & IP1: Intra-procedure-call temporary registers
- XR: Indirect result location parameter
- PR: 系统平台寄存器(Platform registers)
- FP: 栈帧指针寄存器(Frame pointer)

应用程序依赖的处理器状态

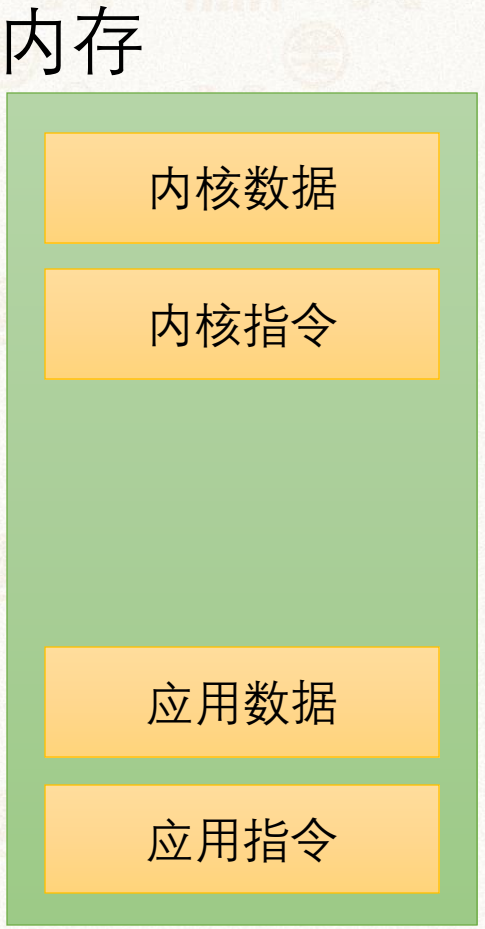


大纲

- 为什么选择ARM
- 硬件执行逻辑
- ARM汇编语言
- 内存模型

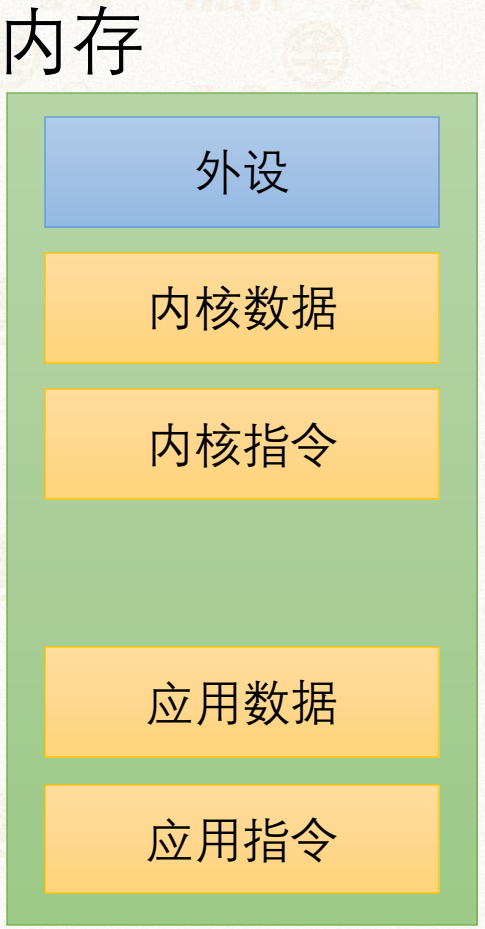
内存空间

- 在冯诺依曼架构中，数据程序均在内存中，且寻址空间相同
  - 内核代码不能改，数据不能乱写
  - 如何做权限管理？
  - CPU是一样的，但内存大小差别很大
    - 如何适应不同内存空间？
  - 外设非常多，如何统一管理？



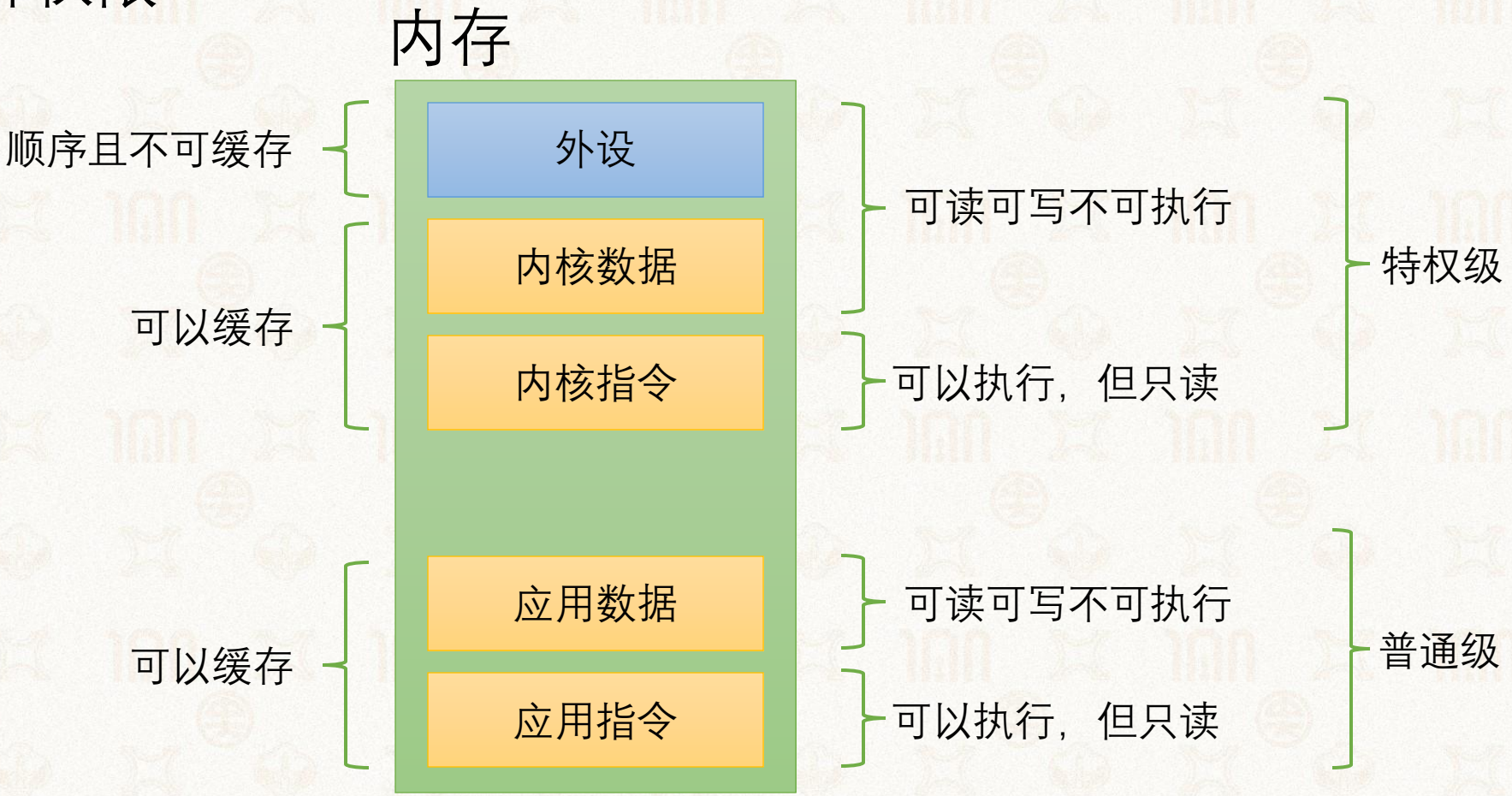
内存空间

- 在冯诺依曼架构中，数据程序均在内存中，且寻址空间相同
  - 内核代码不能改，数据不能乱写
  - 如何做权限管理？
  - CPU是一样的，但内存大小差别很大
    - 如何适应不同内存空间？
  - 外设非常多，如何统一管理？
    - 将外设映射到内存空间中



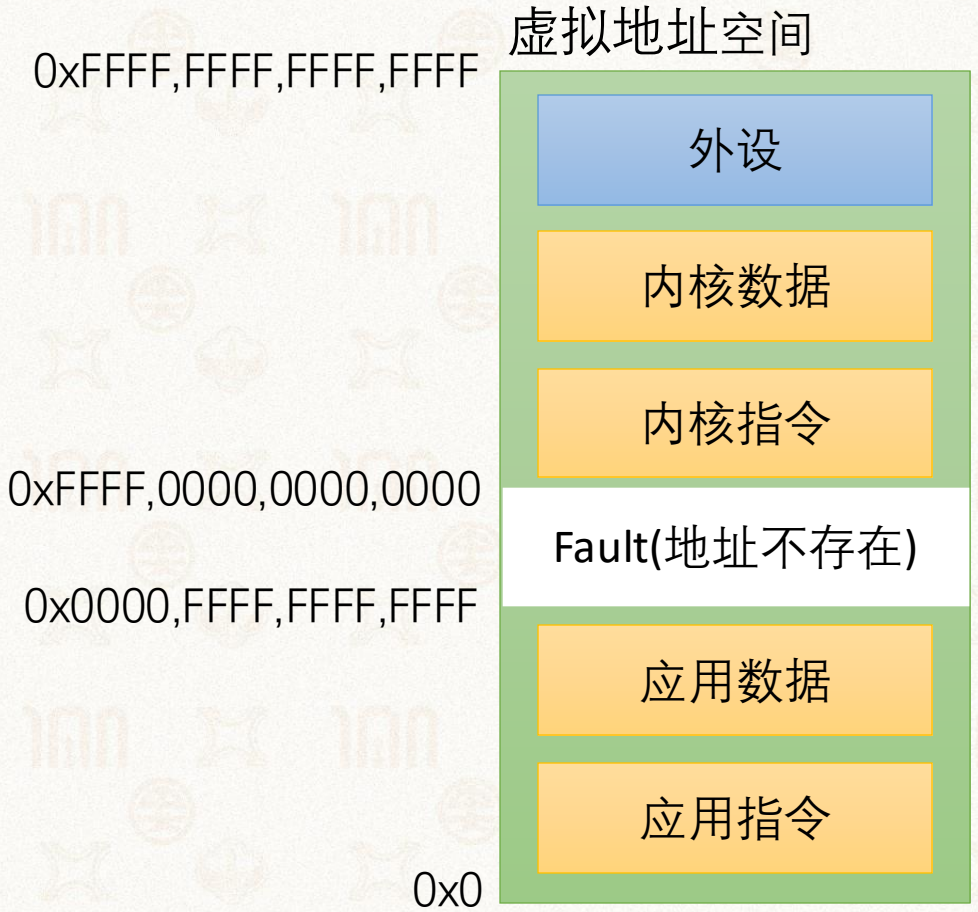
内存空间

- 需要多种权限


内存
外设
内核数据
内核指令
应用数据
应用指令
顺序且不可缓存
可以缓存
可以缓存
可读可写不可执行
可以执行，但只读
可读可写不可执行
可以执行，但只读
特权级
普通级

统一内存映射

- 虚拟地址空间
- 前2^{48}字节的地址空间给普通应用
- 操作系统、外设的地址均为0xFFFF开头



谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course