体系结构

考点:

1. 体系结构设计的四条准则(每个都能举例)
2. 常见的指令掌握(AND,SUB,CMP,MOV,ADC,SBC;移位指令:LSL,LSR)
3. 寄存器特殊用途
4. 常数与立即数
5. 存储器(按字节访问,读写)
6. 条件执行
7. 分支
8. 循环
9. 函数调用
10. 偏移地址
11. 索引模式

---

1. 四条准则

(1) 规整性支持简单设计

- 每条指令都有 2 位操作码。
- 每条指令都有 4 位条件码 。
- **ARM 针对最常用的指令采用 3 种指令格式**：数据处理格式（Data-processing）、存储器格式（Memory）以及分支格式（Branch）。
- **数据处理和存储器指令格式的操作数数量和顺序相似**。
- **每条指令的大小相同**，这使得译码硬件的设计变得简单。

(2) 加快常用功能

- **寄存器**使得访问最近使用的变量速度极快。
- **精简指令集计算机 (RISC) 架构**使得常用/简单的指令运行飞快，因为计算机只需处理少量的简单指令。
- **大多数指令需要完整的 32 位**，因此所有指令都被设定为 32 位（即便有些指令增加长度更有利，而有些指令缩短长度更有利）。选择固定的指令长度是为了确保常用指令的执行效率。

(3) 越小越快

- **寄存器堆仅包含 16 个寄存器**。
- **指令集架构 (ISA) 仅包含少量的常用指令**。这保持了硬件的精简，从而提高了速度。

- **保持较小的指令体积**以加快取指令速度。

(4) 好的设计需要好的折中

- **ARM 使用三种指令格式**（而非仅仅一种）。
- 虽然理想情况下所有访问都应像寄存器访问一样快，但 ARM 架构也支持**主存储器访问**，以便在快速访问时间和海量存储容量之间取得折中。
- 由于 ARM 是 RISC 架构，它仅包含一组简单指令，但它为用户和编译器提供了常用操作的**伪代码（如 NOP）**。
- **ARM 提供了三种（若计入移位的 5 位编码则为四种）立即数编码格式**：
  - 数据处理指令：$rot_{3:0}, imm8_{7:0}$（循环右移位及 8 位常数）。
  - 存储器指令：$imm12_{11:0}$（12 位偏移量）。
  - 分支指令：$imm24_{23:0}$（24 位偏移量）。

---

2. 指令掌握

数据处理指令主要用于对寄存器中的数据进行算术、逻辑或比较操作 。

助记符	指令含义	说明
MOV	数据传送 (Move)	将立即数或寄存器值传送到目的寄存器 。
AND	逻辑与 (Logical AND)	按位进行逻辑与操作，常用于位屏蔽 。
SUB	减法 (Subtract)	执行减法操作。
CMP	比较 (Compare)	执行减法但不保存结果，仅更新 CPSR 标志位 。
ADC	带进位加 (Add with Carry)	用于计算超过 32 位长的数。
SBC	带借位减 (Subtract with Carry)	逻辑等效于 $Rd = Rn - shifter\_operand + C - 1$ 。

1. MOV

- **用途**：初始化寄存器或在寄存器间传递数据 。
- **示例**：
  - `MOV R0, #23`： 将立即数 23 存入 R0 。
  - `MOV r0, r2`： 将 r2 的值复制到 r0 。

2. AND (逻辑与)

- **用途**：提取（屏蔽）特定位。
- **示例**：
  - `AND R3, R1, R2` 。

- **位屏蔽示例**：`0xF234012F AND 0x000000FF = 0x0000002F`（仅保留最低字节）。

3. SUB & CMP (减法与比较)

- **SUB**：执行减法并存储结果。例如 `SUB a, b, c` 对应 $a = b - c$。
- **CMP**：
  - 本质是执行减法操作（$Rn - shifter\_operand$），但不保存结果。
  - **标志位更新**：
    - 若结果为 0，则 $Z = 1$。
    - 若结果为负，则 $N = 1$。
    - 若产生进位，则 $C = 1$；若产生符号位溢出，则 $V = 1$。

4. ADC & SBC

- **用途**：常成对使用以实现 64 位算术运算。
- **64位加法示例 (结果存入 r5:r4)**：
  - `ADDS r4, r0, r2`：低 32 位相加并更新标志位（S 后缀）。
  - `ADC r5, r1, r3`：高 32 位相加，并计入低位的进位。
- **64位减法示例**：
  - `SUBS r4, r0, r2`：低 32 位相减并更新标志位。
  - `SBC r5, r1, r3`：高 32 位相减，并计入低位的借位。

5. 移位指令 (Shifts)

移位指令可以作为独立指令使用（在 Thumb 中），也可以作为 ARM 数据处理指令中第二个操作数（shifter_operand）的一部分。

LSL (逻辑左移)

- **操作**：寄存器中各位左移指定位数，空出的低位补 0。
- **效果**：$R0 = R7 \ll 5$。
- **示例**：`LSL R0, R7, #5`。

LSR (逻辑右移)

- **操作**：寄存器中各位右移指定位数，空出的高位补 0。
- **效果**：$R3 = R2 \gg 31$。
- **示例**：`LSR R3, R2, #31`。

---

3. 寄存器特殊用途

寄存器名	别名	主要用途
R0	-	参数传递 / 函数返回值 / 临时变量
R1-R3	-	参数传递 / 临时变量
R4-R11	-	保存变量
R12	-	临时变量
R13	**SP**	堆栈指针
R14	**LR**	链接寄存器
R15	**PC**	程序计数器

在函数调用（Procedure Calls）过程中，寄存器被分为两大类以保证程序的正确运行：

1. **受保护寄存器 (Callee-Saved)**：
   - 包括 **R4-R11, R13 (SP), R14 (LR)**。
   - **原则**：被调用者（Callee）在使用这些寄存器前必须先将其值压入栈中保护，返回前须恢复 。
2. **不受保护寄存器 (Caller-Saved)**：
   - 包括 **R0-R3, R12, CPSR**。
   - **原则**：调用者（Caller）若在调用后仍需使用这些值，必须在调用前自行保存 。

注意: 实际上

- SP 和 LR 不是普通意义的 callee-saved
- LR 是否保存 取决于是否发生嵌套调用
  不过老师说按PPT132页记忆即可

---

4. 常数与立即数

立即数定义

立即数是指直接编码在指令字中的常数。在 ARM 指令集中，数据处理指令（如 `ADD`、`MOV` 等）的 `shifter_operand` 可以是一个立即数。

立即数的编码规则 (8-4 规则)

ARM 指令是 32 位定长的。为了在指令中表示尽可能大范围的常数，同时受限于指令位数的限制，立即数在 32 位指令码中占用 **12 位**。

这 12 位的分配如下：

- **低 8 位 (immed_8)**：基本常数部分。
- **高 4 位 (rotate_imm)**：循环右移次数（单位是 **2 位**）。

计算公式：

$$\text{Immediate Value} = \text{immed\_8 ROR } (2 \times \text{rotate\_imm})$$

- ROR 表示循环右移。
- 因为 4 位可以表示 $0-15$ 的值，乘以 2 后，表示循环右移 $0, 2, 4, \ldots, 30$ 位。

非法立即数

基于上述编码规则，并非所有的 32 位常数都可以作为立即数出现在指令中。

非法立即数：

- 无法通过 8 位数循环右移偶数位得到的常数。
- **示例**：
  - `0x101` (二进制为 `1 0000 0001`，无法放入 8 位范围内)
  - `0x102` (二进制为 `1 0000 0010`，跨度超过 8 位)

当需要加载一个不符合上述规则的 32 位常数到寄存器时，通常进行**指令组合**：

- 使用 `MOV` 和 `ORR` 多条指令分步用多个合法的立即数构造常数。

指令	操作	说明
`MOV R0, #0x7E000000`	$R0 = 0x7E000000$	加载最高 8 位，符合立即数规则。
`ORR R0, R0, #0xDC0000`	$R0 = R0 \mid 0x00DC0000$	将次高 8 位“或”入寄存器。
`ORR R0, R0, #0x8700`	$R0 = R0 \mid 0x00008700$	将第三个 8 位“或”入寄存器。
`ORR R0, R0, #0x65`	$R0 = R0 \mid 0x00000065$	将最后 8 位“或”入，完成构造。

---

5. 存储器(按字节访问,读写)

Load 与 Store

ARM 采用加载/存储（Load/Store）架构，即所有数据处理必须在寄存器中进行，访问存储器需使用专门指令：

- **Load (LDR)**：将存储器中的数据“加载”到寄存器。
- **Store (STR)**：将寄存器中的数据“存储”到存储器。

基本格式：`助记符 Rd, [Rn, #offset]`

- **Rd**：目标/源寄存器。
- **Rn**：基址寄存器，存有存储器的基础地址。
- **#offset**：立即数偏移量（十进制或十六进制）。

地址计算公式：

$$\text{有效地址 (Address)} = \text{Rn (基址)} + \text{offset (偏移量)}$$

字寻址与字节对齐

- **按字节寻址**：ARM 存储器被视为一个从零开始的线性字节数组。每个字节都有一个唯一的地址。
- **数据类型**：
  - **字节 (Byte)**：8 位。
  - **半字 (Halfword)**：16 位（占据 2 个字节地址）。
  - **字 (Word)**：32 位（占据 4 个字节地址）。

由于存储器按字节寻址（每个地址存 8 位），而一个字由 4 个字节组成，因此：

- **字对齐规则**：存储器数据字的地址必须是 4 的倍数。
- **字地址转换**：
  - 第 n 个字的物理地址 $= n \times 4$。
  - 字地址 2 对应物理地址 8；字地址 21 对应物理地址 84 ($0x54$)。

指令变体与数据长度

根据访问的数据长度不同，指令有以下变体：

助记符	功能说明	访问长度
LDR / STR	加载/存储字 (Word)	32 bits
LDRB / STRB	加载/存储字节 (Byte)	8 bits (低8位)
LDRH / STRH	加载/存储半字 (Halfword)	16 bits (低16位)
LDM / STM	多寄存器加载/存储 (Multiple)	一次操作多个寄存器

示例 A：从存储器读取数据 (Load)

任务：从存储器物理地址 8 处读取一个字到 R3。

```
MOV R2, #0          ; 初始化基址寄存器 R2 为 0
LDR R3, [R2, #8]    ; 计算地址 0 + 8 = 8，读取该地址的内容到 R3
```

结果：R3 将持有地址 8 处的 32 位数据。

示例 B：向存储器写入数据 (Store)

任务：将 R7 的值存入第 21 个存储字（Word 21）。

```
MOV R5, #0           ; 初始化基址寄存器 R5 为 0
STR R7, [R5, #0x54] ; 21 * 4 = 84 (即 0x54)，将 R7 存入地址 84
```

注意：偏移量可以使用十进制（`#84`）或十六进制（`#0x54`）。

字节序

存储器中字节的排列顺序有两种标准：

- **小端**：低位字节存储在低地址。这是 ARM 默认和最常用的方式。
- **大端**：高位字节存储在低地址。

---

6. 条件执行

非顺序执行

在程序中，指令并不总是顺序执行的（如 `if/else` 语句、 `while` 循环）。

- **分支（Branching）**：当特定条件为真时，跳转到代码的另一部分执行。
- **机制**：ARM 通过**条件标志位**来控制指令是否执行。

ARM 条件标志位

标志位存储在 **当前程序状态寄存器 (CPSR)** 的前四位[31:28]。

标志位	名称	描述
N	Negative	指令结果为负数时置位（结果的第 31 位为 1）。
Z	Zero	指令结果为零时置位。
C	Carry	指令导致无符号数进位（加法）或无借位（减法）时置位。
V	Overflow	指令导致带符号数溢出时置位。

设置条件标志位的方法

在 ARM 中，标志位不会被所有指令自动更新，必须通过以下两种方式设置：

1. **比较指令：CMP**
   - **格式**：`CMP Rn, shifter_operand`
   - **原理**：执行减法运算（$Rn - shifter\_operand$）。
   - **特点**：**不保存运算结果**，仅根据结果更新 NZCV 标志位。
2. **使用“S”后缀**
   - **格式**：在指令助记符后加 `S`（如 `ADDS`，`SUBS`，`ANDS`）。
   - **特点**：**保存运算结果**到目的寄存器，并同时更新标志位。

条件助记符

指令可以根据标志位的状态决定是否执行。条件码作为后缀添加到指令助记符之后。

助记符	含义	执行条件 (Flags)
EQ	Equal (相等)	$Z = 1$
NE	Not Equal (不相等)	$Z = 0$
MI	Minus (负数)	$N = 1$
PL	Plus (正数或零)	$N = 0$
AL	Always (总是执行)	默认值，通常省略

示例解析

ARM 的经典编程模式是：**先使用一条指令设置标志位，随后紧跟带条件后缀的指令。**

```
CMP R5, R9      ; 执行 R5 - R9，并设置标志位
SUBEQ R1, R2, R3 ; 仅当 R5 == R9 (Z=1) 时，执行 R1 = R2 - R3
ORRMI R4, R0, R9 ; 仅当 R5 - R9 结果为负 (N=1) 时，执行逻辑或
```

假设运行数据：
若 $R5 = 17, R9 = 23$：

1. `CMP` 执行 $17 - 23 = -6$。
2. 标志位更新为：$N = 1$（负数），$Z = 0$（非零）。
3. `SUBEQ` **不执行**（因为 $Z = 0$）。
4. `ORRMI` **执行**（因为 $N = 1$）。

注意:

- CMP 的本质是只影响标志位而不改变寄存器数值的减法。
- S 后缀的区别：普通 `ADD` 不影响标志位， `ADDS` 才会。
- 条件码位置：条件后缀（如 `EQ`）紧跟在指令助记符之后（如 `BEQ`，`MOVEQ`）。
- 二进制补码运算：在判断 N 和 V 标志时，需考虑有符号数的补码表示及其溢出逻辑。

---

7. 分支

分支指令通过修改程序计数器（PC）的值，使指令不再按顺序执行，从而实现程序流程的跳转（如 `if/else`、 `loop` 或函数调用）。

分支指令分类

助记符	指令名称	描述
B	分支 (Branch)	跳转到指定的地址或标识（Label）。
BL	带链接分支 (Branch and Link)	跳转前将下一条指令的地址保存到 **R14 (LR)** 中，用于子程序调用。
BX	分支并切换 (Branch and Exchange)	根据目标地址的最低位切换 ARM/Thumb 状态（资料后续涉及）。

无条件分支与条件分支

(1) 无条件分支

无条件分支通常采用 `B` 或 `BL` 指令形式，当程序执行流到达此类指令时，硬件会直接修改程序计数器（PC）的值，强制指令流必然跳转至目标地址，而不依赖于任何条件码或状态位的判断。

- **示例**：

```
B TARGET          ; 直接跳到 TARGET 标号处
ORR R1, R1, #4    ; 此行被跳过，不会执行
TARGET
SUB R1, R1, #78   ; 执行此行
```

(2) 条件分支

条件分支通过在跳转指令 `B` 后附加特定的条件助记符（如 `BEQ` 表示相等跳转、`BNE` 表示不等跳转、`BMI` 表示负数跳转）来实现。在执行过程中，硬件会检查程序状态寄存器（CPSR）中的条件标志位，仅当标志位状态符合指令预设的逻辑要求时才触发跳转，否则程序将忽略跳转动作并顺序执行下一条指令。在实际代码中，这种机制通常与 `CMP`（比较）指令配合使用，通过 `CMP` 指令更新标志位，随后由条件分支指令根据该标志位决定控制流的方向。

- **示例**：

```
CMP R1, R2        ; 比较 R1 和 R2
BEQ EQUAL_LABEL   ; 如果 R1 == R2 (Z=1)，则跳转
ADD R0, R0, #1    ; 如果不相等，执行此行
```

标识的使用

- **定义**：标识用于代表指令在存储器中的地址。
- **命名规则**：标识名称不能与指令助记符（如 `ADD`，`ORR` 等）冲突。
- **汇编器处理**：在汇编过程中，汇编器会自动计算跳转目标与当前 PC 之间的偏移量（Offset），并编码进指令。

分支范围与 PC 相对寻址

- **寻址原理**：ARM 的分支指令通常使用 **PC 相对寻址**。
- **偏移量限制**：由于指令长度为 32 位，其中一部分位用于存储操作码，剩余位用于存储跳转偏移量。
  - 在 ARM 状态下，`B` 指令通常支持 $\pm 32\text{MB}$ 的跳转范围。
  - 注：偏移量在编码时会进行向右移 2 位的处理（因为指令地址必定 4 字节对齐）。

针对 `if` 和 `if/else` 语句的两种实现

- **分支跳转法 `if`**：适用于较长的代码块（超过 3-4 条指令）。
- **条件执行法 `if/else`**：对于短的代码块（如 1-2 条指令）更高效，因为它避免了分支指令带来的流水线刷新开销。

1. if 语句 (C 代码: `if (i == j) f = g + h; f = f - i;`)

写法一：分支跳转

利用反向逻辑，若条件不满足则跳过代码块。

```
; R0=f, R1=g, R2=h, R3=i, R4=j
CMP R3, R4        ; 比较 i 和 j
BNE L1            ; 如果 i != j，跳转到 L1（跳过加法）
ADD R0, R1, R2    ; f = g + h
L1 SUB R0, R0, R3   ; f = f - i
```

写法二：条件执行

直接在指令后添加条件后缀，无需标号。

```
CMP R3, R4        ; 比较 i 和 j
ADDEQ R0, R1, R2 ; 仅当 i == j 时执行: f = g + h
SUB R0, R0, R3    ; 无论如何执行: f = f - i
```

2. if/else 语句 (C 代码: `if (i == j) f = g + h; else f = f - i;`)

写法一：分支跳转

需要使用两个跳转标号来分隔 `if` 和 `else` 代码块。

```
CMP R3, R4        ; 比较 i 和 j
BNE L1            ; 如果 i != j，跳转到 L1 (即执行 else 部分)
ADD R0, R1, R2    ; if 部分: f = g + h
B L2              ; 执行完 if 后跳转到 L2，绕过 else 部分
L1 SUB R0, R0, R3   ; else 部分: f = f - i
L2                ; 结构结束点
```

写法二：条件执行

利用互补的条件后缀（`EQ` 和 `NE`），代码最为精简。

```
CMP R3, R4       ; 比较 i 和 j
ADDEQ R0, R1, R2 ; 如果 i == j，执行 f = g + h
SUBNE R0, R0, R3 ; 如果 i != j，执行 f = f - i
```

---

8. 循环

循环的基本实现逻辑

在汇编中，循环通过标号、比较指令、条件分支和无条件分支共同实现。其核心是在循环体结束处跳转回顶部，或在不满足条件时跳转到循环外。

While 循环

逻辑描述：在进入循环体前先检查条件。如果条件不满足（在汇编中通常检测“退出条件”），则直接跳过循环体。

汇编结构示例：

```
; R0 = pow
MOV   R0, #1
WHILE
    CMP   R0, #128     ; 检查条件
    BEQ   DONE         ; 如果 pow == 128，退出循环
    LSL   R0, R0, #1   ; 循环体：pow = pow * 2
    B     WHILE        ; 无条件跳转回顶部再次检查
DONE
```

For 循环

构成要素：初始化、条件测试 、循环操作、语句块。

汇编映射关系：

- **初始化**：在循环标号之前执行（如 `MOV R0, #1`）。
- **条件测试**：在循环顶部使用 `CMP` 和 `BEQ`。
- **循环操作**：在循环体末尾执行（如 `ADD R0, R0, #1`），然后执行 `B` 返回顶部。

标准递增循环示例:

```
; R0 = i, R1 = sum
    MOV   R0, #1       ; 初始化: i = 1
    MOV   R1, #0       ; 初始化: sum = 0
FOR
```

```
    CMP   R0, #10       ; 条件检查: R0 - 10
    BEQ   DONE          ; 如果 i == 10，退出循环
    ADD   R1, R1, R0    ; 循环体: sum = sum + i
    ADD   R0, R0, #1    ; 循环操作: i = i + 1
    B     FOR           ; 重复循环
DONE
```

4. 性能优化：递减循环(可以不掌握)

对于 ARM 体系结构，递减循环比递增循环更高效。

效率差异

- **普通循环 (Incremented)**：需要 `ADD`（自增）+ `CMP`（比较）+ `B`（跳转）。
- **递减循环 (Decremented)**：仅需 `SUBS` + `B`。

优化原理

- **指令合并**：使用 `SUBS` 指令。它在执行减法的同时会根据结果更新 CPSR 标志位。
- **减少指令数**：
  - `SUBS R0, R0, #1`：既完成了变量递减，又通过结果是否为零或负数设置了标志。
  - `BGE FOR`：只要结果大于或等于零就跳转。
- **结论**：每轮循环可节省 **2 条指令**（省去了显式的 `CMP` 指令，并减少了分支判断的复杂度）。

递减循环示例:

```
; R0 = i, R1 = sum
    MOV   R0, #9        ; 初始化: i = 9
    MOV   R1, #0        ; 初始化: sum = 0
FOR
    ADD   R1, R1, R0    ; 循环体: sum = sum + i
    SUBS  R0, R0, #1    ; 递减并设置标志位: i = i - 1
    BGE   FOR           ; 如果 i >= 1，跳转回 FOR (重复循环)
```

---

9. 函数调用

- **调用函数 (Caller)**：发起调用的函数（如 `main`）。其职责是传递参数并跳转至子程序。
- **被调用函数 (Callee)**：被执行的函数（如 `sum`）。其职责是执行计算、返回结果，并确保**不破坏**调用函数所需的寄存器和内存环境。

ARM 函数调用惯例

项目	规则	说明
跳转指令	`BL` (Branch and Link)	调用函数时使用。
返回指令	`MOV PC, LR`	函数结束时将返回地址由 `LR` 送回 `PC`。
参数传递	**R0 - R3**	前 4 个参数分别存放在 R0-R3 寄存器中。
返回值	**R0**	函数的计算结果必须存放在 R0 中返回。

(1) BL (Branch and Link) 指令

- **动作 1**：将下一条指令的地址保存到 **LR (R14)** 寄存器中，作为返回地址。
- **动作 2**：跳转到目标函数地址执行。
- **示例**：`BL SIMPLE` 会让 `LR = 0x00000204`（假设 `BL` 指令在 `0x00000200`）。

(2) 函数返回

- **指令**：`MOV PC, LR`。
- **原理**：将 `LR` 中保存的返回地址赋值给程序计数器 `PC`，使程序跳回到调用点之后。

带参数的函数调用

以 C 代码 `y = diffofsums(2, 3, 4, 5);` 为例：

汇编代码流程：

1. **参数准备**（由 Caller 完成）：

```
MOV R0, #2    ; 参数 0 = 2
MOV R1, #3    ; 参数 1 = 3
MOV R2, #4    ; 参数 2 = 4
MOV R3, #5    ; 参数 3 = 5
```

2. **调用函数**：

```
BL DIFFOFSUMS ; 跳转并保存返回地址到 LR
```

3. **函数内部执行**（由 Callee 完成）：

```
; 运算结果存入 R4 (result = (R0+R1) - (R2+R3))
ADD R8, R0, R1
ADD R9, R2, R3
SUB R4, R8, R9
```

4. **设置返回值并返回**：

```
MOV R0, R4    ; 将计算结果(result)存入 R0
MOV PC, LR    ; 返回到调用者
```

- 如果函数内部还需要调用其他函数（嵌套调用），必须先将当前的 `LR` 压栈保存，否则会被新的 `BL` 覆盖。
- **寄存器限制**：ARM 规定参数使用 R0-R3。如果参数超过 4 个，通常需要使用**堆栈**。
- 被调用者不能破坏调用者正在使用的寄存器。如果必须使用 R4-R11，则需要通过栈操作进行保护和恢复。
- `BL` 执行后，`PC` 指向被调函数，而 `LR` 指向 `BL` 的下一条指令。

栈

- **定义**：存储器中用于临时存储数据的特定区域，遵循“后进先出”(LIFO）原则。
- **SP (Stack Pointer)**：寄存器 **R13**，始终指向当前的栈顶地址。
- **增长方向**：在 ARM 中，栈向**低地址**方向增长（向下增长）。
  - **入栈 (Push)**：减小 SP 值，腾出空间，存入数据。
  - **出栈 (Pop)**：读取数据，增加 SP 值，释放空间。

---

10. 偏移地址

ARM Assembly	Memory Address
`LDR R0, [R3, #4]`	R3 + 4
`LDR R0, [R5, #-16]`	R5 – 16
`LDR R1, [R6, R7]`	R6 + R7
`LDR R2, [R8, -R9]`	R8 – R9
`LDR R3, [R10, R11, LSL #2]`	R10 + (R11 << 2)
`LDR R4, [R1, -R12, ASR #4]`	R1 – (R12 >>> 4)
`LDR R0, [R9]`	R9

> **把内存某个地址处的数据读出来，放到寄存器 Rd**
> 这个“某个地址”由 `[]` 里面的表达式算出来。

图里每一行都对应一种常见的寻址写法：

1 `LDR R0, [R3, #4]`

- 地址 = R3 + 4
- `#4` 是立即数偏移（单位是字节）

2 `LDR R0, [R5, #-16]`

- 地址 = R5 - 16

3 `LDR R1, [R6, R7]`

- 地址 = R6 + R7
- 用寄存器 `R7` 当偏移量

4 `LDR R2, [R8, -R9]`

- 地址 = R8 - R9
- 负号表示偏移寄存器取反方向

5 `LDR R3, [R10, R11, LSL #2]`

- 地址 = R10 + (R11 << 2)
- `LSL #2` 左移 2 位，相当于 `R11 * 4`
- 常见用途：数组下标转字节偏移（比如 4 字节一个元素）

6 `LDR R4, [R1, -R12, ASR #4]`

- 地址 = R1 - (R12 >>> 4)（算术右移）
- `ASR` 是算术右移，会保留符号位（适合有符号数）

7 `LDR R0, [R9]`

- 地址 = R9（纯基址，无偏移）

---

11. 索引模式

Mode	Address	Base Reg. Update
Offset	Base register ± Offset	No change
Preindex	Base register ± Offset	Base register ± Offset
Postindex	Base register	Base register ± Offset

Examples

- Offset:
```
LDR R1, [R2, #4]     ; R1 = mem[R2+4]
```
- Preindex:
```
LDR R3, [R5, #16]!   ; R3 = mem[R5+16]
                     ; R5 = R5 + 16
```
- Postindex:
```
LDR R8, [R1], #8     ; R8 = mem[R1]
                     ; R1 = R1 + 8
```

- **偏移**：用 `Rn+offset` 读，但 Rn 不改变
- **前索引**`!`：先改 Rn 再读（`!`=写回）
- **后索引**：**先读**再改 Rn（偏移写在 `]` 外）