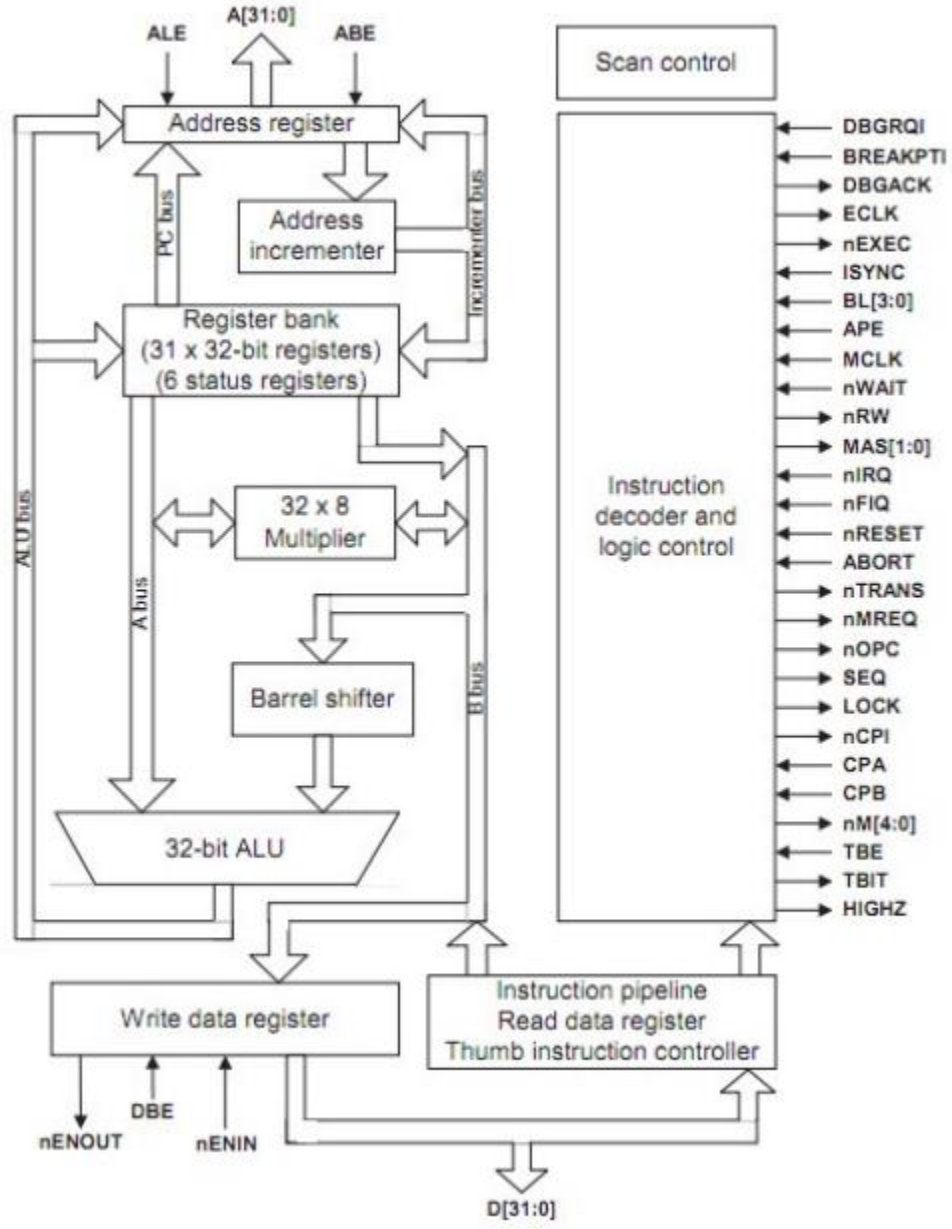
ARM7TDMI 架构
源于Yann DOUZE,

d’après les supports de cours de Sébastien Bilavarn

ARMv4架构

ARM7 TDMI 架构

- 32-bit 处理器
- 32-bit ALU
- 寄存器队列
- 移位寄存器
- 32x8bit乘法器



主要特点

- 加载存储架构
  - 指令仅处理寄存器数据，并将结果放入寄存器。访问内存的操作只有将内存值复制到寄存器（加载）的操作和将寄存器值复制到内存（存储）的操作。
- 固定指令编码格式
  - 所有指令均以 32 位编码.
- 处理指令的格式3地址
  - 2 个操作数寄存器和1个结果寄存器，可独立指定。
- 条件执行
  - 每条指令都可以有条件地执行
- 具体传输指令
  - 高性能多重存储器-寄存器传输指令
- ALU+移位
  - 可以在一条指令(1个周期)中执行算术或逻辑运算和移位，而在大多数其他处理器上，它们是由单独的指令 执行的

处理单元

- 处理功能单元:
  - 寄存器文件
    - 16 个寄存器，用于灵活处理数据和存储 ALU 结果
- 算术和逻辑单元。

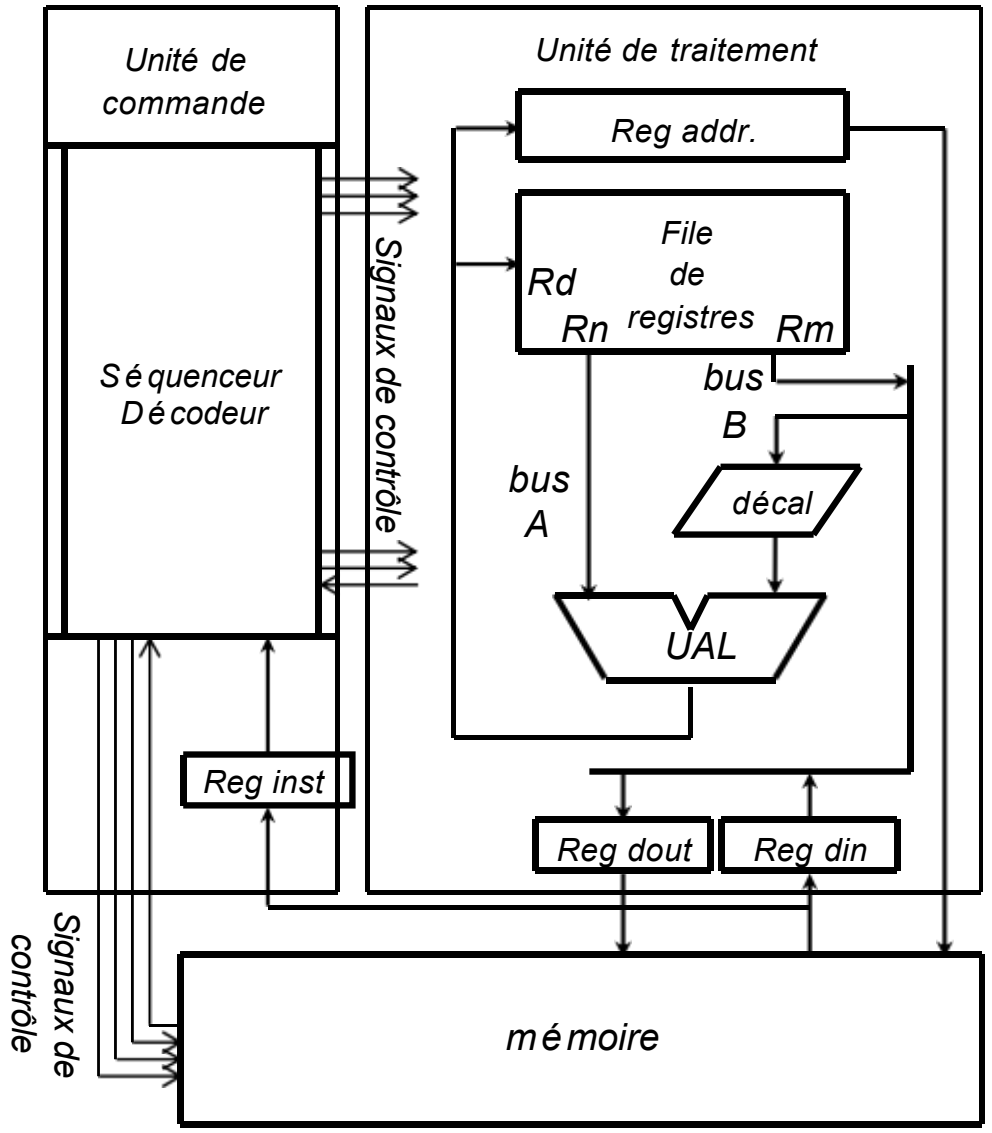
  2个操作数:来自寄存器的输入A数据，连接到移位器的输入 B数据

  ALU 结果:返回寄存器
- 移位寄存器

  移位操作(左移1次=x2，右移1次= 12)

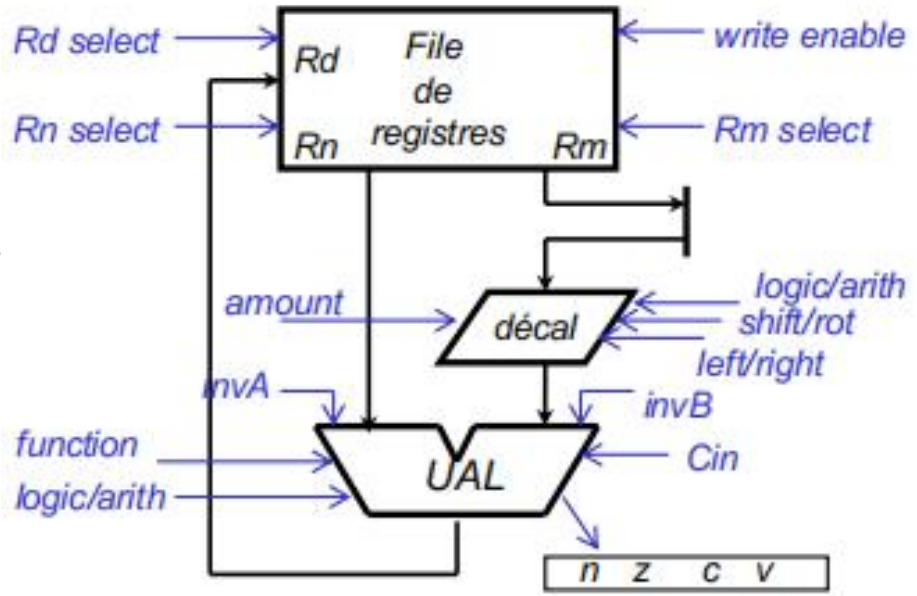
  循环操作(移位+重新注入丢失的比特)

  与 ALU 输入B相连，在一个周期内执行 ALU+移位指令

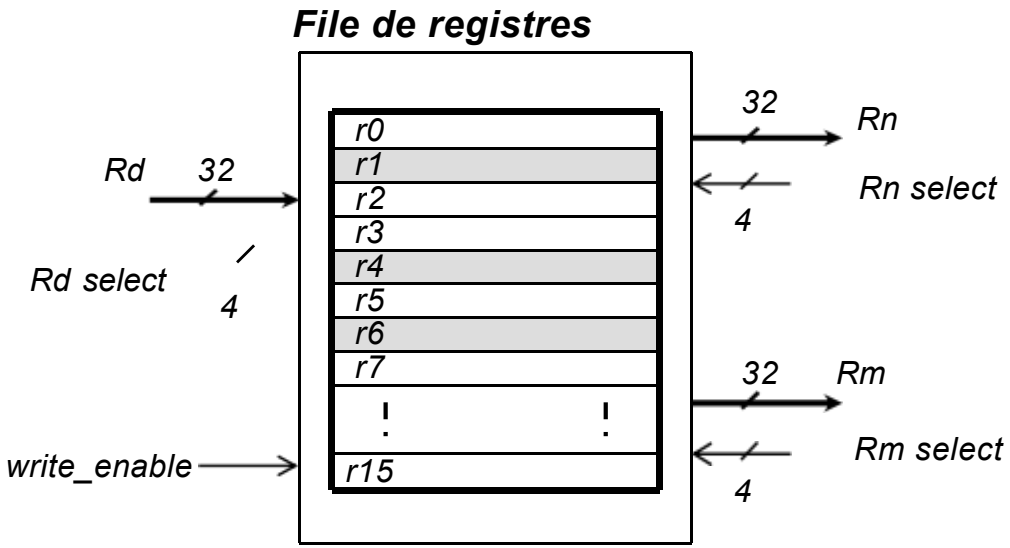


处理单元结构

- 处理单元的控制信号
  - 寄存器文件
    - Rd, Rn, Rm select：
      在16个寄存器队列中选择目 标寄存器
    - write enable:激活寄存器队列
  - ALU
    logic/arith+function ：逻辑或算术模式，在每种模式下可选择功能
    InvA、invB:操作数A和B的反转(非)
    Cin:注入保持位C
    指示器:指示 ALU 运算(如保持 C)后的状态
  - Décaleur
    - shift/rot:移位或旋转操作
    - logic/arith:逻辑或算术运算
    - Amount:移位/循环位数



寄存器文件File de registres



- 16 用户寄存器 r0, …, r15
- 调用指令的符号:
  - INST Rd, Rn, Rm
    - 3地址格式 (2个操作数+结果)
    - Rn 寄存器操作数 1，连接到端口 A(32 位)
    - Rm 寄存器操作数 2，通过移位器连接到端口 B(32 位)-->输入B可以进行循环/
    - Rd 目的地寄存器(32 位)
    - 选择 4 位，从 16 个可用寄存器中选择一个

File de registres

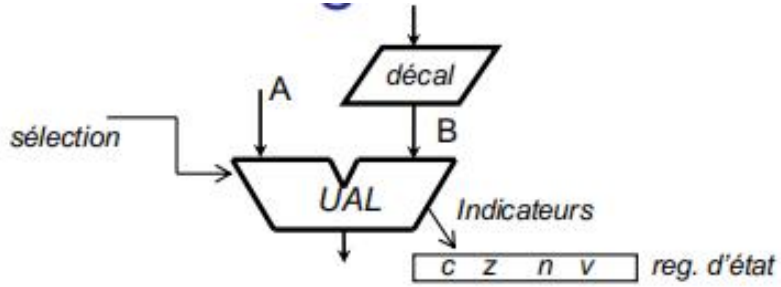
- 寄存器之间的数据移动指令
  - MOV (Move), MVN (Move not)
    - MOV Rd, *#literal*
    - MOV Rd, Rn
    - MOV Rd, Rm, *shift*

    - 寄存器之间的数据移动，或从常量到寄存器的数据移动。

    - *#literal*: 立即数(常量)
    - *shift*:第二个操作数可能需要移位

  - Exemples
    - `MOV r3, #2` `@ r3 ← 2`
    - `MOV r3, r4` `@ r3 ← r4`
    - `MOV r3, r4, LSL #2` `@ r3 ← r4<<2`

ALU 和移位寄存器



- 算术和逻辑运算（执行算术和逻辑运算）
  - (ADD, SUB, AND, OR, …)
- 与输入移位寄存器 B 相关
- 状态寄存器 （SR） 提供运算结果的指示：
  - C: 进位
    - 算术(或移位)运算的溢出指示位
  - Z: Zero
    - UAL 零结果指示位
  - N:负
    - UAL负结果的指示位
  - V: 溢出(oVerflow)
    - 结果溢出的指示位(修改符号位)

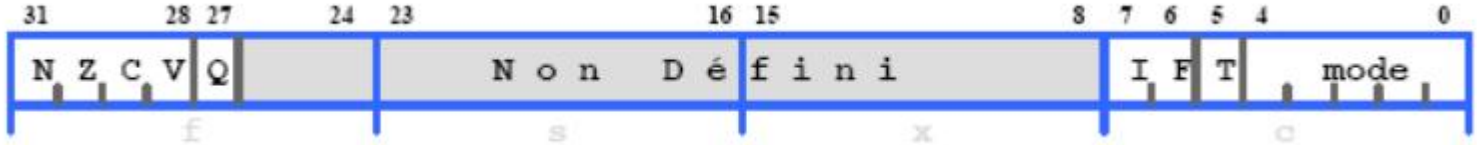
ALU和移位寄存器

- **4位示例**: 1 0 1 0 + 1 0 0 1 = (1) 0 0 1 1
  - ALU结果: 0 0 1 1
  - C = 1
  - Z = 0, 结果与 0 0 0 0 不同
  - N = 0, 0 0 1 1 是正数，因为符号位 (第 4 bit) = 0
  - V = 1, 因为 1 0 1 0 和 1 0 0 1 是负数，结果是正数

  - 根据对数字的解释，C、V、N 位将被检查或忽略
    - 如果数字采用无符号表示，则C有用，V和N无用
    - 如果数字是有符号表示，C没用，V和N有用
    - 这些标志位由ALU设置，程序员在使用或不使用它们时取决于其需要(例如，在示例的情况下从4位加法执行8位加法)

状态寄存器

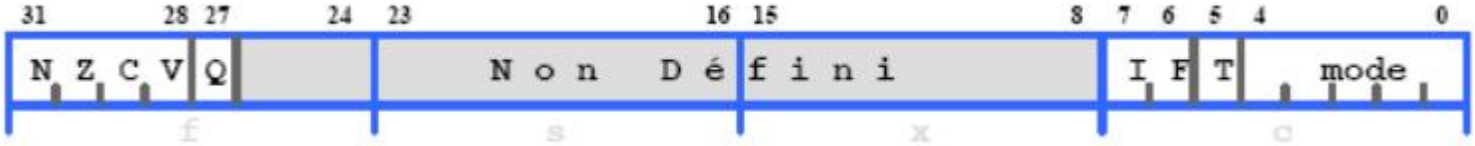
- CPSR:当前程序状态寄存器
- 包含指示符Z（零）、N（负）、C（进位）、V（溢出）
- 给出算术运算或比较结果的信息
- 允许根据结果执行或不执行指令(条件执行)
- 允许为超过 32 位的操作保留进位



状态寄存器

- 条件指示
  - Z 零结果
  - N 负结果
  - C 保留
  - V 溢出
- Q 溢出
  - 表示特定类型的溢出(饱和算术)
- 未定义区域

- 中断验证
  - I=1 禁用 IRQ
  - F=1 禁用 FIQ
- 拇指模式
  - T=0 mode ARM (32 bits)
  - T=1 mode thumb (16 bits)
- 模式指示
  - 指示激活模式：系统, IRQ, FIQ, 用户 …



ALU和移位寄存器

- **移位助记符** (*shift*)
  - LSL #*n* :逻辑左移(0 ≤ n ≤ 31)
  - LSR #*n* :逻辑右移(1 ≤ n ≤ 32)
  - ASR #*n* :算术右移(1 ≤ n ≤ 32)
  - ROR #*n* :向右循环(1 ≤ n ≤ 31)
  - RRX: 向右循环扩展 （ROR用状态寄存器的进位位C 扩展1位）

    - 移位 (>>, <<): 将字向右或向左移动。空闲位置用零(逻辑)或符号位(算术)填充。
      - Ex1: 0111, LSR #1 → 0011, LSL #1 → 1110
      - Ex2: 1011, ASR #1 → 1101, ASL #1 → 0110　无法进行符号扩展，与 LSL #1相同
    - Rotation: 循环移位，丢失的位被重新注入
      - Ex: 0111, ROR #1 → 1011
      - 带进位：再循环一位(C), ex (0) 0111 RRX → (1) 0011

ALU和移位寄存器

- Ex: 加法指令
  - ADD Rd, Rn, *#literal*
  - ADD Rd, Rn, Rm
  - ADD Rd, Rn, Rm, *shift*

- Exemples
  - `ADD r3, r2, #1 @ r3 ← r2 + 1`
    - `Opérande 2 (Rm) = constante`
  - `ADD r3, r2, r5 @ r3 ← r2 + r5`
  - `ADD r3, r2, r5, LSL #2 @ r3 ← r2 + (r5<<2)`
    - `opérande 2 (Rm) = opérande décalé.`
    - `Multiplie par 4 la valeur de r5, ajoute la valeur de r2 et stocke le résultat dans r3`

ALU和移位寄存器

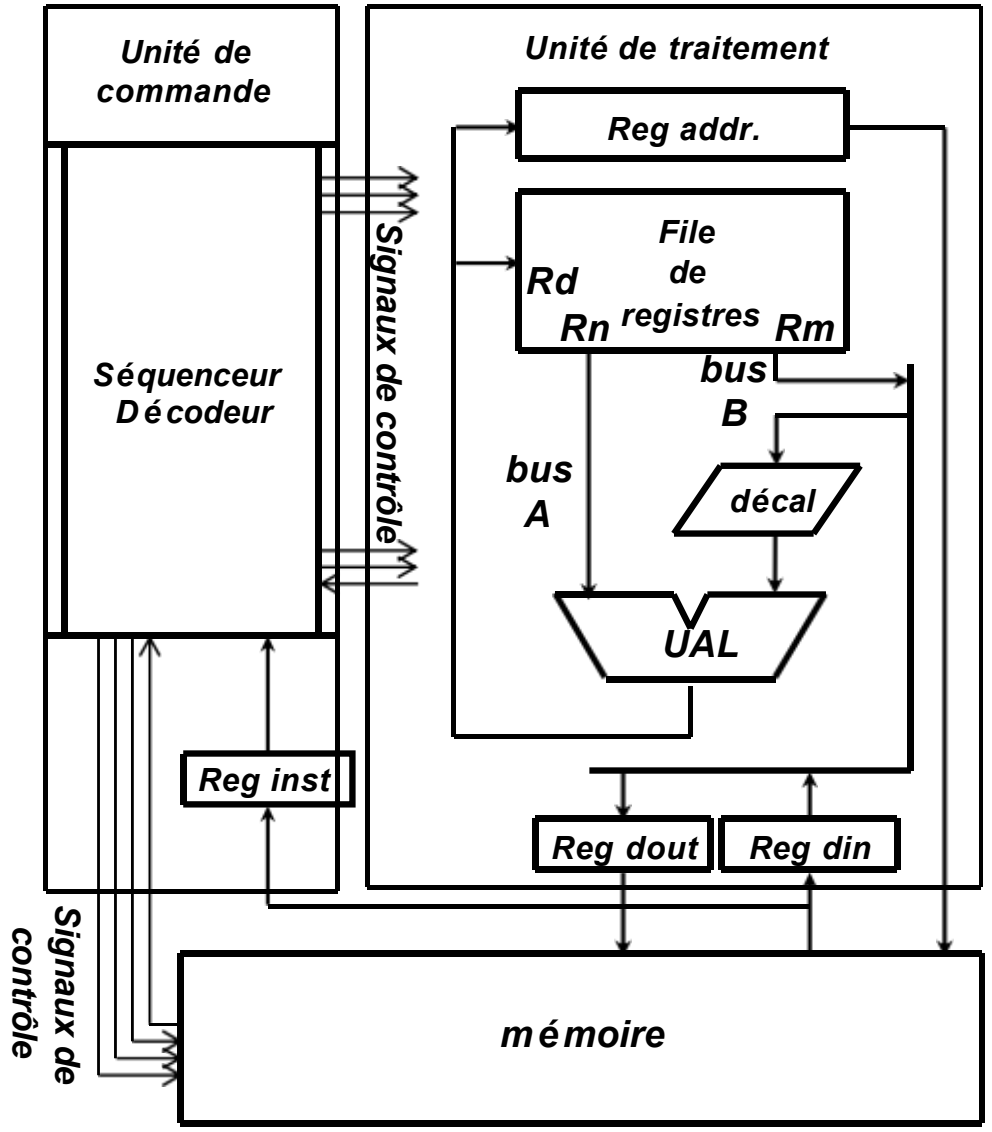
- Exercice

- Ecrire une séquence qui multiplie par 10 la valeur de r5 et stocke le résultat dans r6
  - 10xr5 = 8xr5 + 2xr5
  - 1) multiplier par 8 = 3 décalages à gauche
  - `MOV r1, r5, LSL #3` `@ r1 ← (r5<<3)`
  - 2) multiplier par 2 = 1 décalage à gauche
  - `MOV r2, r5, LSL #1` `@ r2 ← (r5<<1)`
  - 2) ajouter:
  - `ADD r6, r1, r2` `@ r6 ← r1 + r2`

  - Plus efficace:
  - `MOV r1, r5, LSL #3` `@ r1 ← (r5<<3)`
  - `ADD r6, r1, r5, LSL #1` `@ r6 ← r1 + (r5<<1)`

控制单元的作用

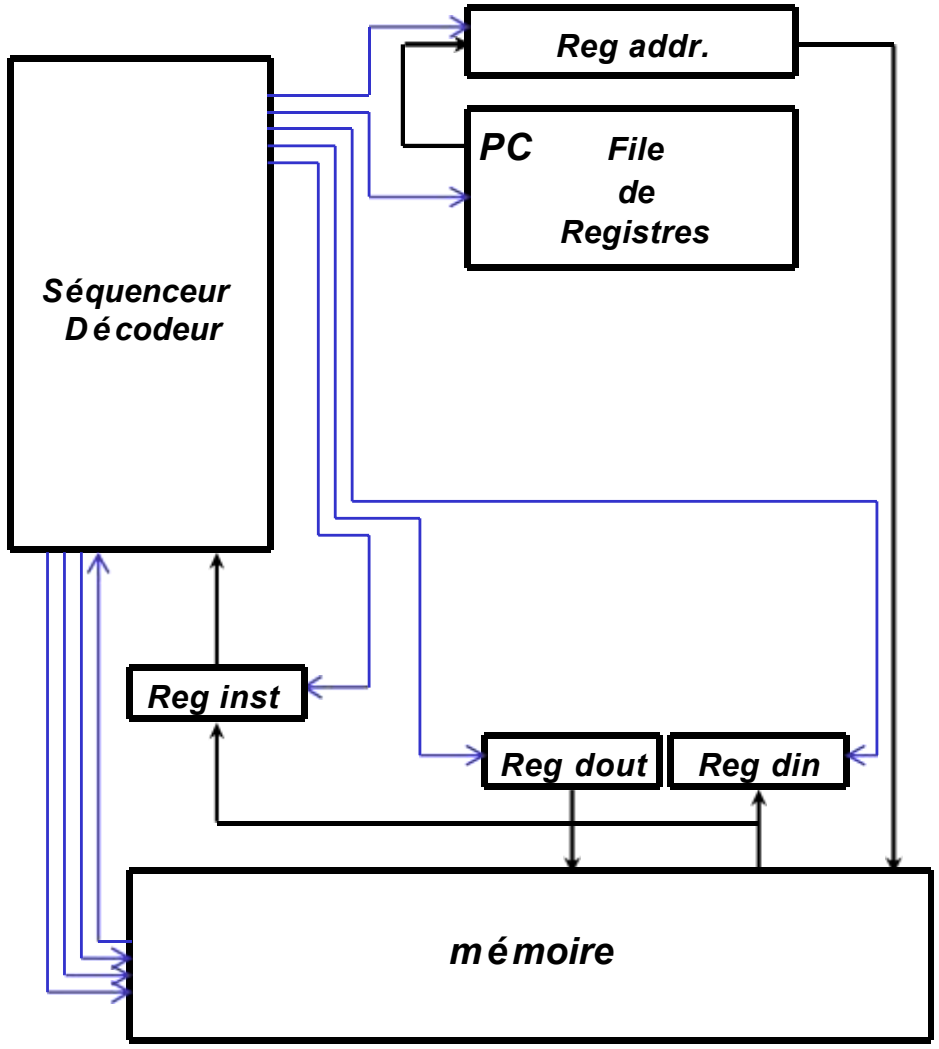
- 内存访问控制
  - 获取指令/数据
  - 设置访问内存中指令/数据所需的信号
- 解码指令
  - 指令译码
  - 将指令转化为控制信号的过程
- 处理单元控制
  - 执行指令
  - 设置执行指令所需的信号



控制单元的作用

■ 内存访问控制

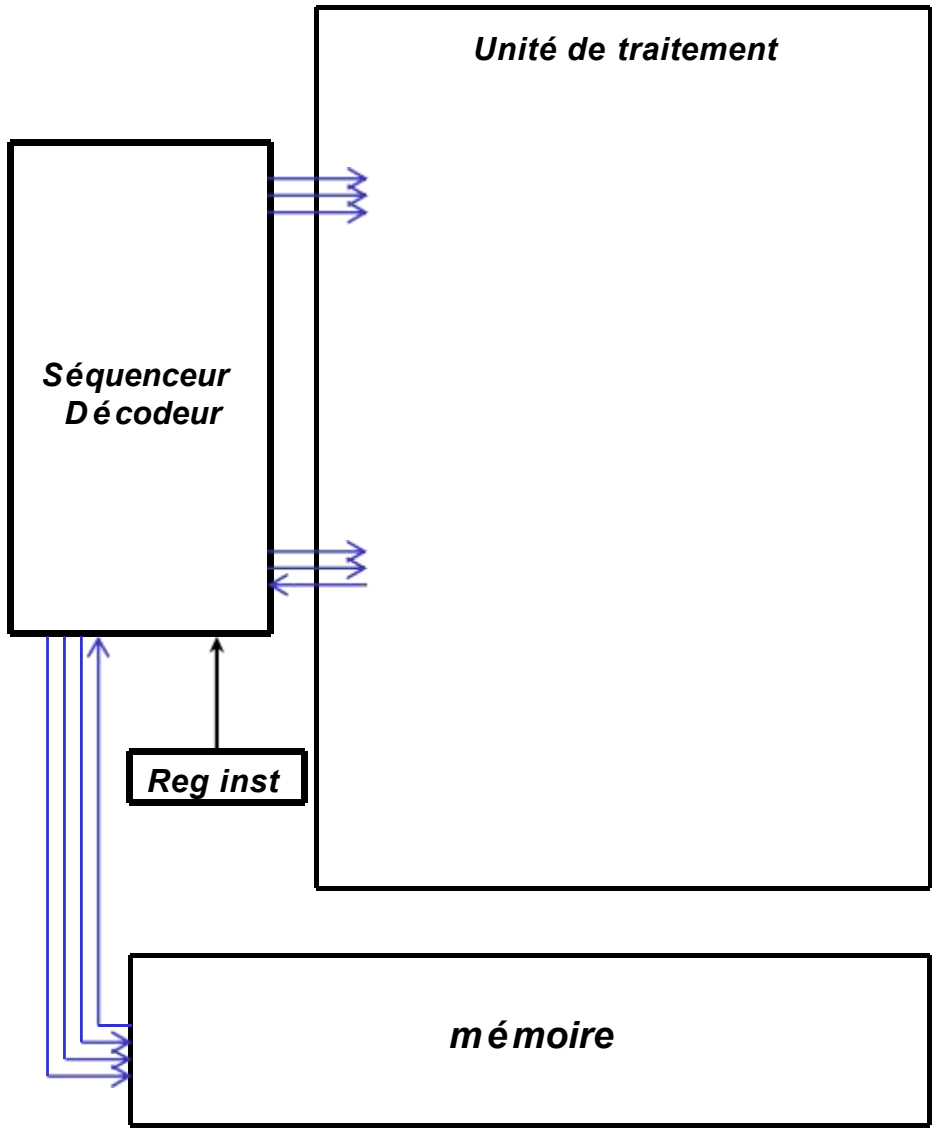
- 获取指令
  - ➢ 程序执行时，在内存中的指令搜索阶段
  - ➢ 地址寄存器: 包含待执行指令的地址
  - ➢ 指令寄存器:相应指令返回指令寄存器进行解码。
- 数据访问
  - ➢ 在执行加载/存储指令期间。
  - ➢ 数据寄存器(Reg din、Reg:dout)
- 信号产生
  - ➢ 设置所有必要的内存控制信号(nMREQ、nRW、MAS、寄存器



控制单元的作用

■ 解码指令

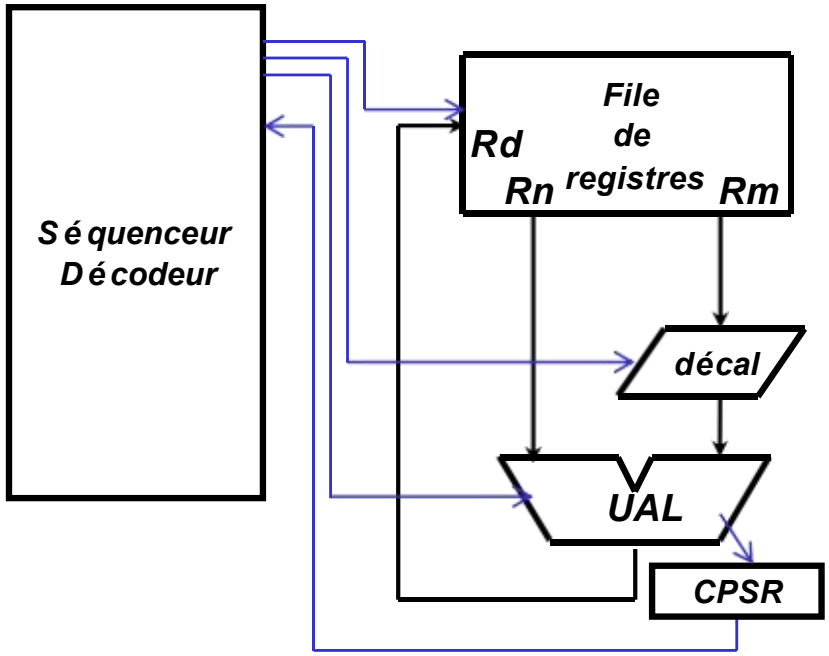
- 将指令转换为控制信号
  - 解码包括将指令的 32 位二进制表示转换为同步信号
  - 用于处理单元排序
  - 内存排序
- 取决于许多参数的复杂过程
  - 指令类型(处理、传输等)
  - 操作数(立即值、寄存器、移位寄存器)状态寄存器位，用于评估执行条件(如等值分支)
  - 寻址模式
  - 数据类型



控制单元的作用

■ 处理单元控制

- 处理单元中的数据流
  - ➢ 在寄存器队列中为每个操作数(Rn、Rm)和结果(Rd)选择寄存器
- 选择操作
  - ➢ 移位(算术、逻辑、循环、偏移次数)
  - ➢ UAL(算术、逻辑、函数)
- 指令的有条件执行
  - ➢ 当前程序状态寄存器



ARM7 指令周期

1. 搜索阶段(“取指”)
   - 在内存中搜索指令
2. 分析阶段(“译码”)
   - 指令解码：分析指令及其操作数的32位二进制表示，以便控制单元执行信号
   - 对于以下阶段，有2种可能的情况，具体取决于它是处理指令(请求处理单元)还是内存访问指令(请求处理和内存单元)

对于数据处理指令：

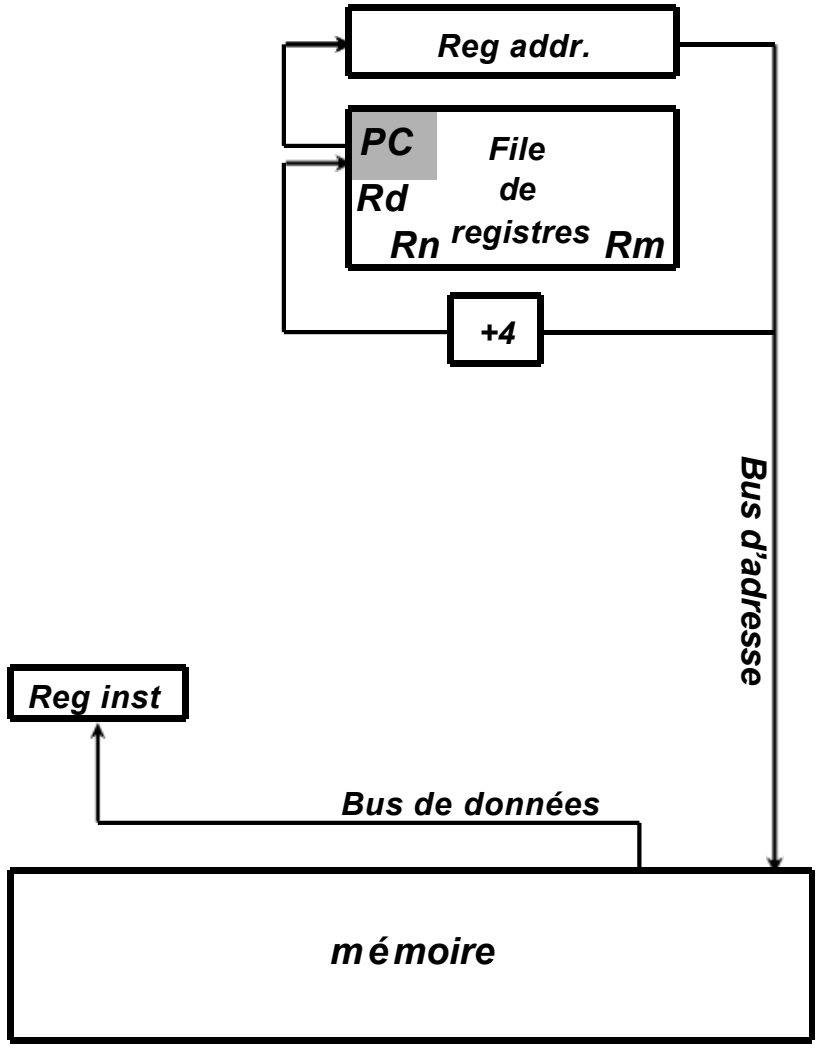
3. 执行阶段(“执行”)
   - n数据流、操作选择、条件执行

对于内存访问指令：

3. 地址计算阶段
   - 地址由UAL计算并存储在地址寄存器中
4. 内存访问阶段(“数据传输”)
   - 地址被发送到内存，内存读取/写入数据

指令周期

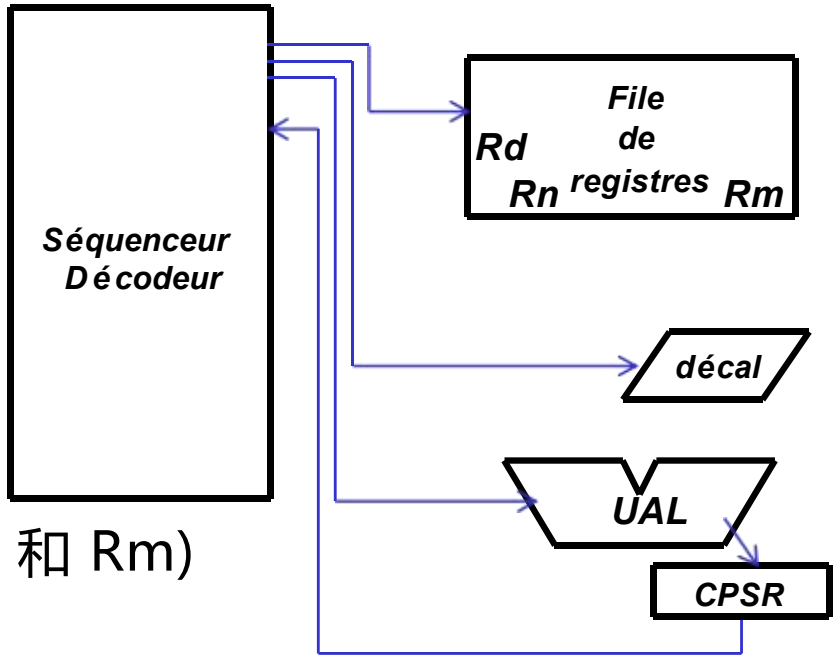
- 取指令阶段 ("取指")
  - 加载地址寄存器
    - 要执行的指令的地址包含在PC寄存器(程序计数器)中
    - 该地址被复制到地址寄存器
  - 内存读取
    - 地址被放置在地址总线上
    - 我们计算下一条指令的地址(PC+4)
    - 加载指令寄存器
    - 指令在数据总线上可用，并复制到指令寄存器
    - PC更新(地址+4)



指令周期

■ 分析阶段(“译码”)

- 读取指令寄存器
  - 代表指令的 32 位字取自指令寄存器
  - 对指令进行解码，以确定要执行的操作顺序和使用的数据。
- 检查状态寄存器
  - 条件指示
- 源操作数/目的操作数选择
  - 源操作数选择(立即值或选择寄存器 Rn 和 Rm)
  - 结果(选择 Rd)
- 选择要执行的操作
  - UAL(逻辑/逻辑思维、函数、invA、invB、Cin)
  - 移位寄存器(逻辑/算数、移位/旋转、左/右、数量)



ARM7 指令周期

1. 搜索阶段(“获取”)
2. 分析阶段(“解码”)

对于数据处理指令:

3. 执行阶段(“执行”)

对于内存访问指令:

3. 地址计算阶段
4. 内存访问阶段(“数据传输”)

指令周期

- 处理指令：执行阶段
  - 单指令类型的情况：

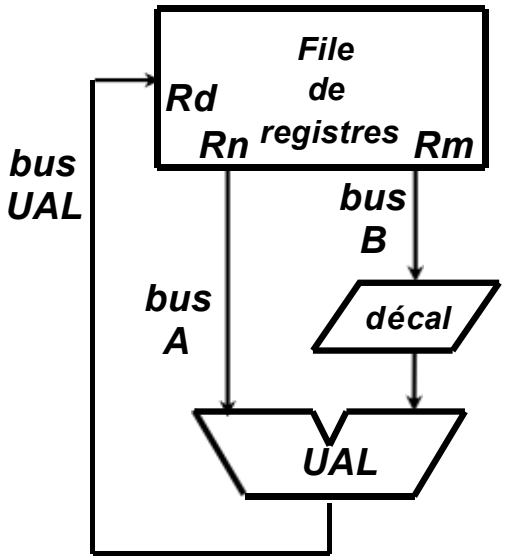
    ADD Rd, Rn, Rm, *shift*

  - 操作数从寄存器队列中读取。

    Rn 位于 ALU 的第一个输入端(A总线)。

    Rm 位于移位器输入端(B总线)。

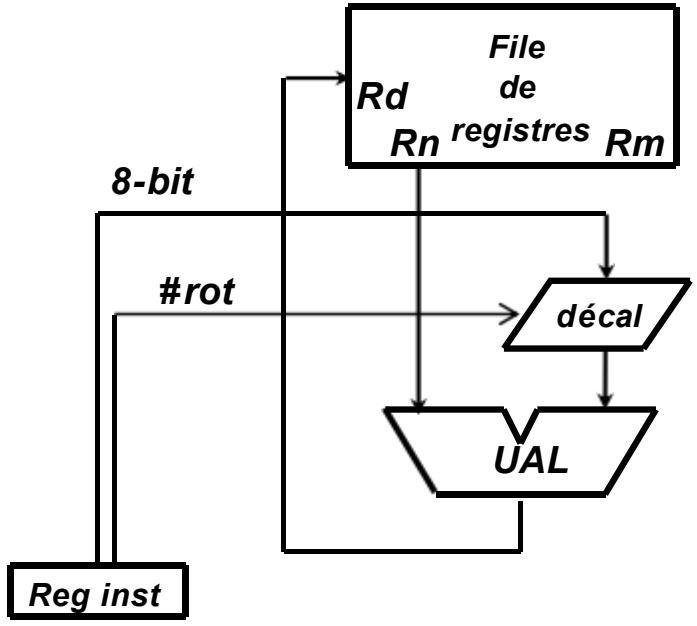
  - 移位操作与算术或逻辑运算一样执行。
  - 结果返回目的寄存器(Rd)



指令周期

- 处理指令：执行阶段
  - 单指令类型的情况

    ADD Rd, Rn, *#litteral*
    - 从寄存器队列中读取 Rn
    - #itteral 从指令中获取(在指令调用中以 8 位+循环编码)
    - 算术或逻辑运算的计算。
    - 结果返回目的寄存器(Rd)



ARM7 指令周期

1. 搜索阶段(“获取”)
2. 分析阶段(“解码”)

对于数据处理指令:

3. 执行阶段(“执行”)

对于内存访问指令:

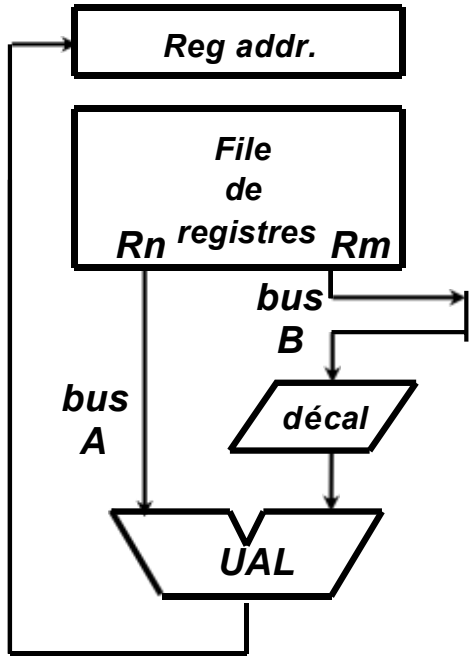
3. 地址计算阶段
4. 内存访问阶段(“数据传输”)

指令周期

■指令的情况

STR Rd, [Rn, +/-Rm, *shift*]

- 地址计算阶段
  - ➢ 地址计算由ALU 进行。计算结果返回到地址寄存器。

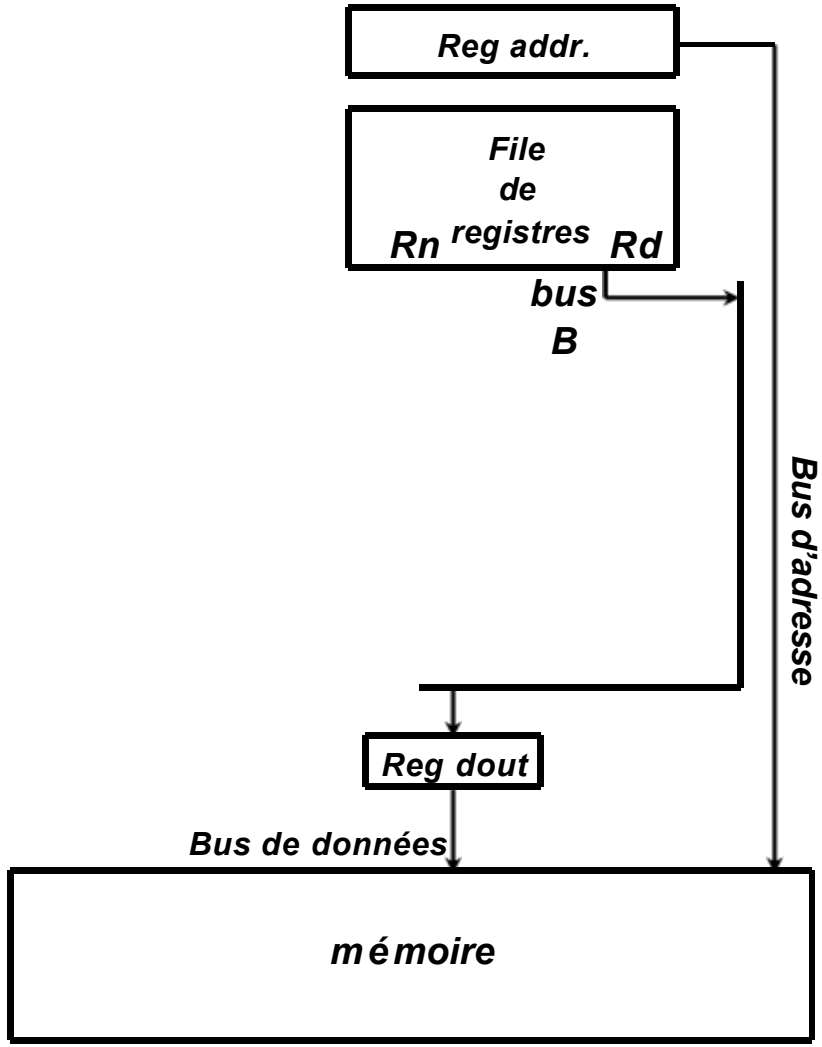


指令周期

指令的情况

STR Rd, [Rn, +/-Rm, *shift*]

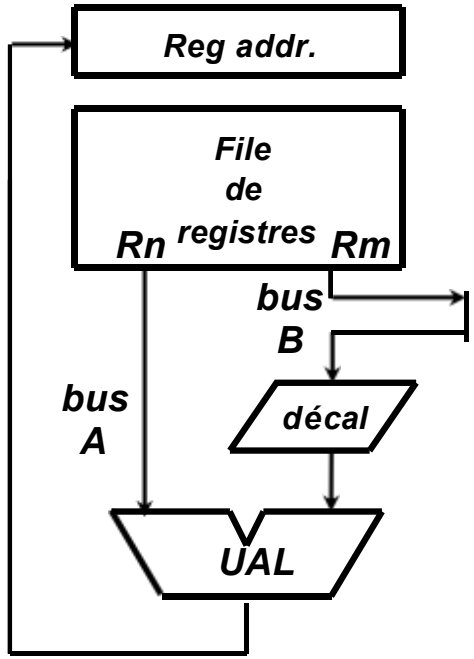
- 访问阶段
  - 地址寄存器中包含的写入地址被发送到地址总线上
  - 内存中待复制寄存器(Rd)的值被复制到 Reg dout 寄存器中。
  - 数据通过数据总线发送并写入内存。



指令周期

■ LDR Rd, [Rn, +/−Rm, 移位]

类型指令的情况



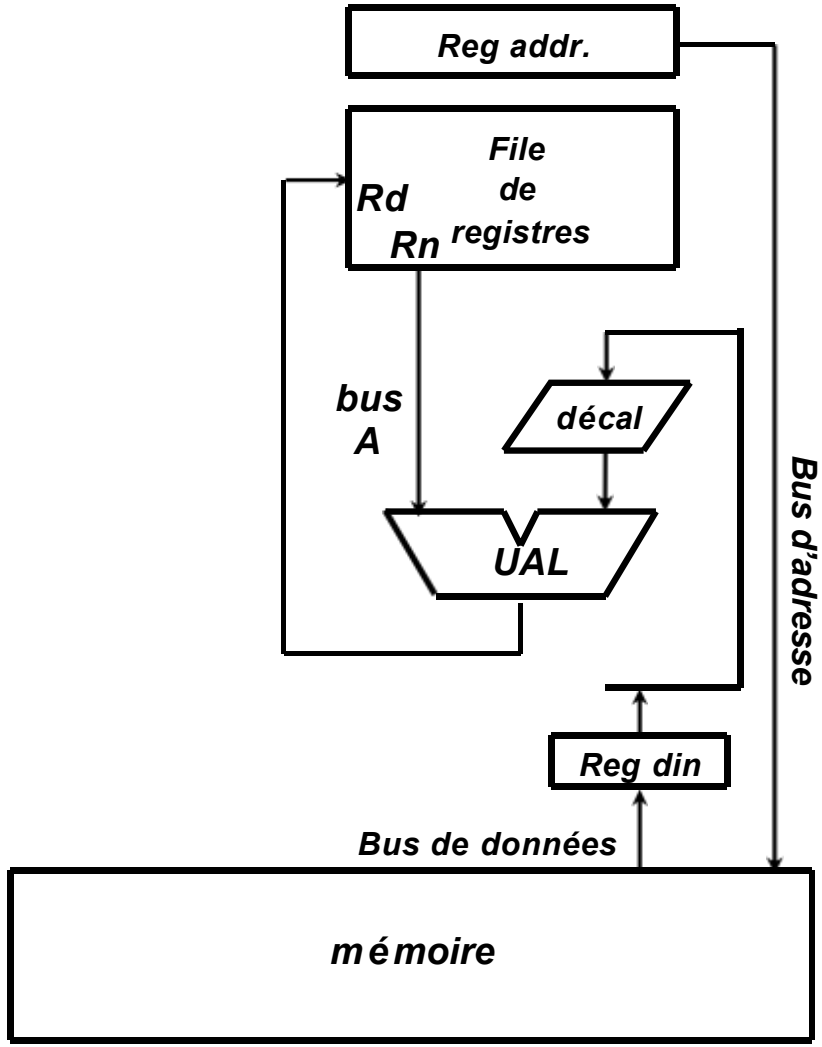
- 地址计算阶段
  - ➢ 地址计算由 ALU 完成。计算结果返回地址寄存器。

指令周期

- LDR Rd, [Rn, +/-Rm, *shift*]

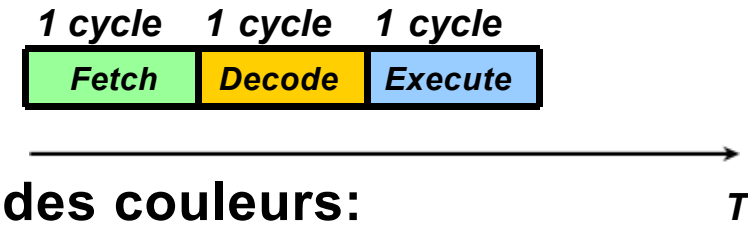
  类型指令的情况

- 读写阶段
  - 在地址总线上发送读取地址
  - 从内存中读取数据
  - 数据通过数据总线发送
  - 数据被路由到目标寄存器(Rd)



流水线执行

- 在 ARM7 上，“取指”、“译码”和“执行”阶段，每个阶段需要一个时钟周期
- **因此，每条处理指令的执行时间为三个时钟周期**



Convention des couleurs:

- 绿色：使用内存（从内存中读取指令）
- 橙色：使用解码器/定序器（指令分析）
- 蓝色：使用处理单元（执行处理指令）

Exécution pipeline

- 执行以下指令序列需要多长时间？

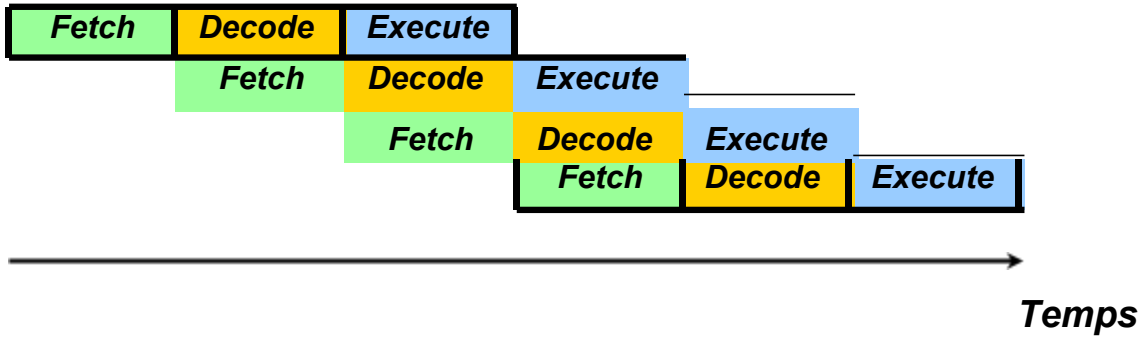
```
MOV  r5, r3, LSL #1
ADD  r5, r5, r3, LSL #3
MOV  r7, r5, LSL #1
ADD  r5, r5, r3, LSL #3
```

- 4 条指令 x 3 = 12 个时钟周期？

流水线执行

- 答案：6 个时钟周期!!

```
MOV  r5,  r3,  LSL  #1
ADD  r5,  r5,  r3,  LSL  #3
MOV  r7,  r5,  LSL  #1
ADD  r5,  r5,  r3,  LSL  #3
```

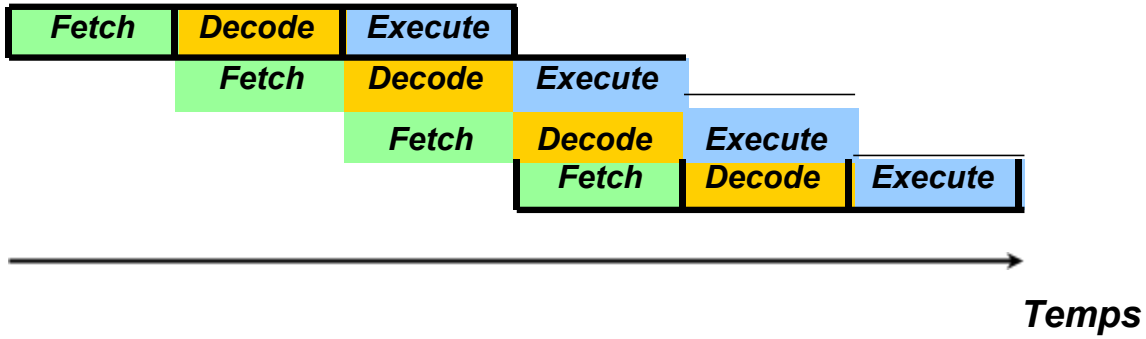


- 流水线使得各个阶段能够重叠并行执行
  - 将一条指令划分为独立的阶段（取指、解码、执行），意味着在前一条指令执行完毕之前就可以开始执行新指令。
    - 取指阶段可以从上一个指令的译码阶段开始；
    - 解码阶段最早可在上一个实例的执行阶段开始。

Exécution pipeline

- 答案：6 个时钟周期!!

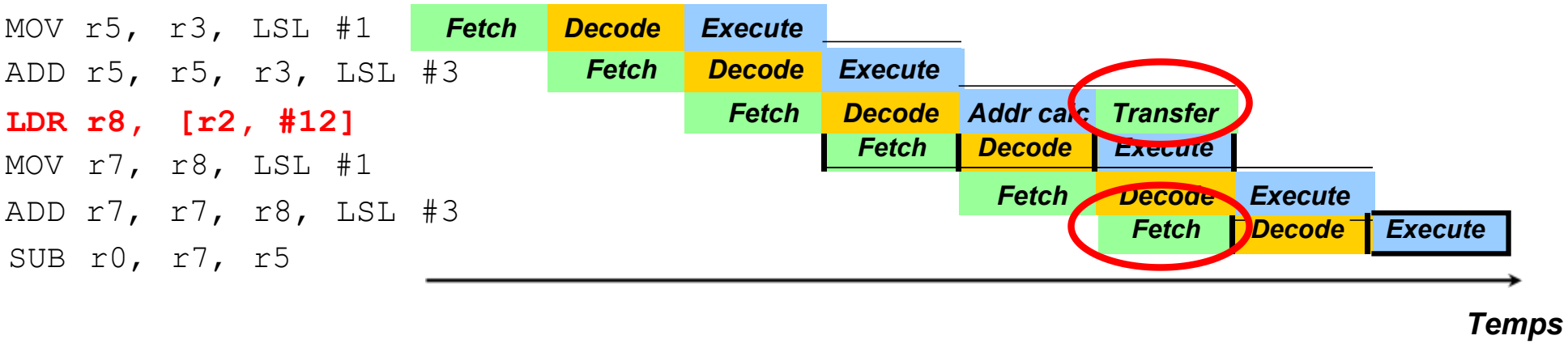
```
MOV  r5,  r3,  LSL  #1
ADD  r5,  r5,  r3,  LSL  #3
MOV  r7,  r5,  LSL  #1
ADD  r5,  r5,  r3,  LSL  #3
```



- 每个时钟周期可以执行一条指令
  - 从周期3开始，每个时钟周期结束一个执行阶段。
  - 充分利用资源：3个阶段有时同时进行
  - 条件：同一周期内不得出现两次相同颜色！！

Exécution pipeline

- 流水线减速：内存访问指令案例（一）

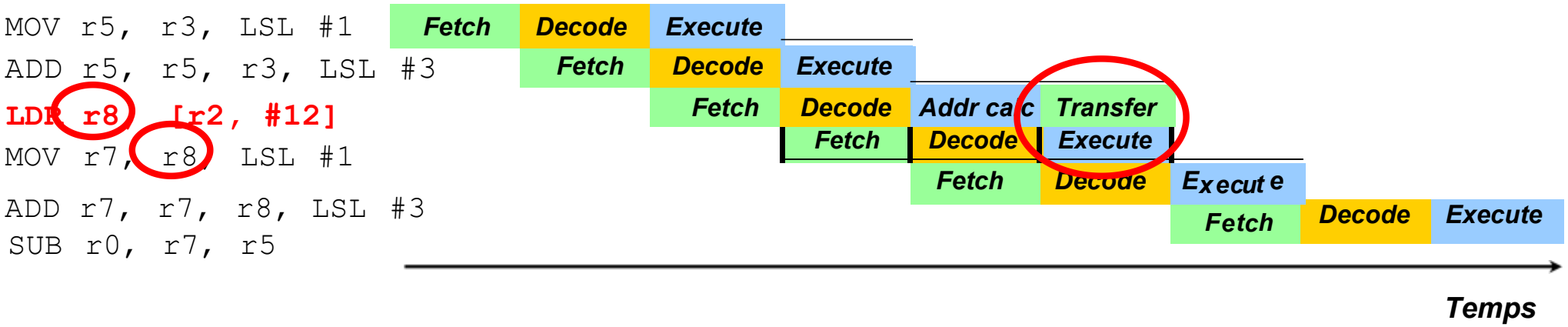


插入一条内存访问指令（LDR r8，[r2，#12]）

- Addr calc 地址计算阶段使用处理单元（蓝色）
- Transfer 传输阶段使用内存（绿色）
- 两个传输级（LDR）和取指级（SUB）不能使用内存

Exécution pipeline

- 流水线减速：内存访问指令案例（二）

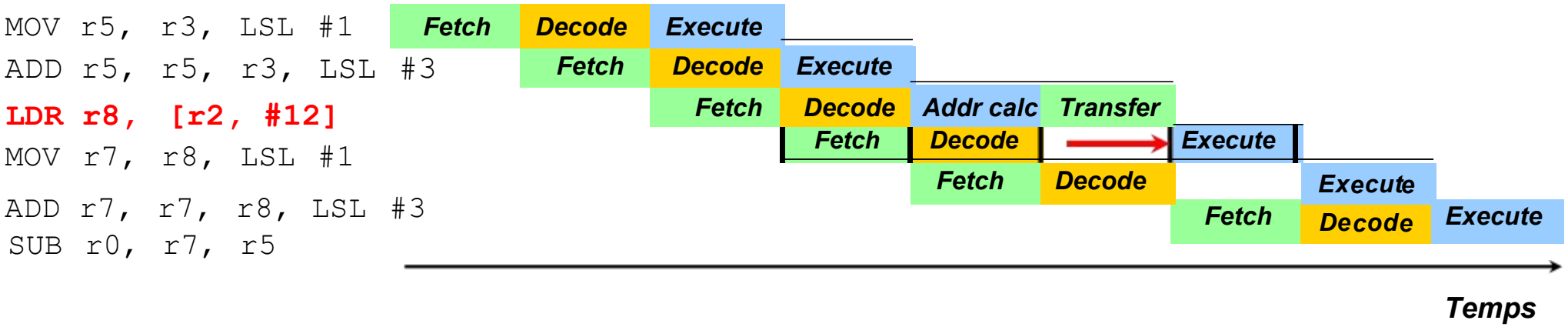


解决办法：将最后一条指令移位1个周期

- 将SUB指令的相位移位 1 个周期
- 但是：LDR（MOV）后面的指令需要寄存器 r8 的值，但该值不可用（传输未完成），因此无法执行。
- MOV指令和ADD指令的执行阶段需要重新移位

Exécution pipeline

- 流水线减速：内存访问指令案例（三）

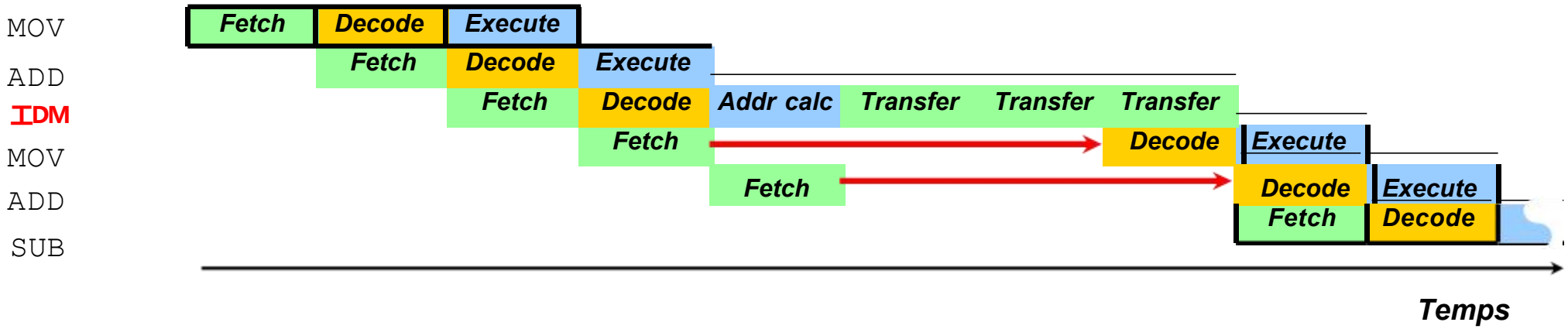


解决方案：错开倒数第二条指令的执行阶段

- 流水线减速：当前指令周期停止，直到数据可用（由处理器管理）
- 流水线中出现中断（“气泡”）现象
- 流水线的实施会带来额外的复杂性成本

Exécution pipeline

- **流水线减速：多内存访问指令的情况**

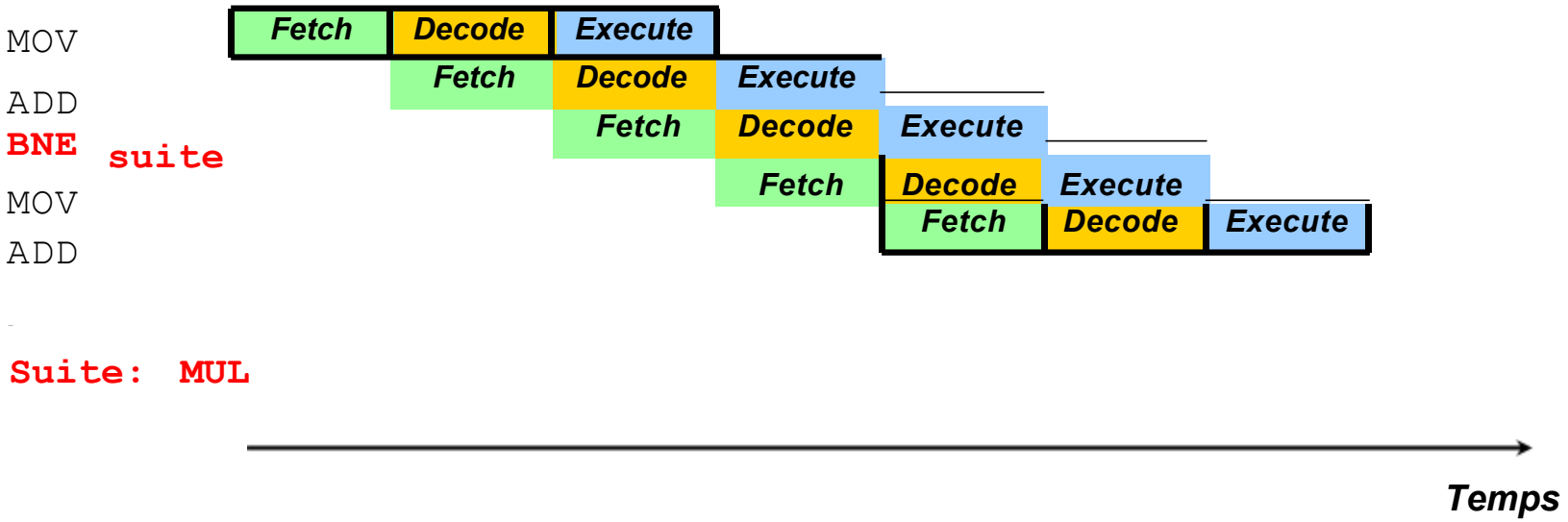


其他流水线减速的情况

- 执行速度因传输阶段的持续时间而减慢

Exécution pipeline

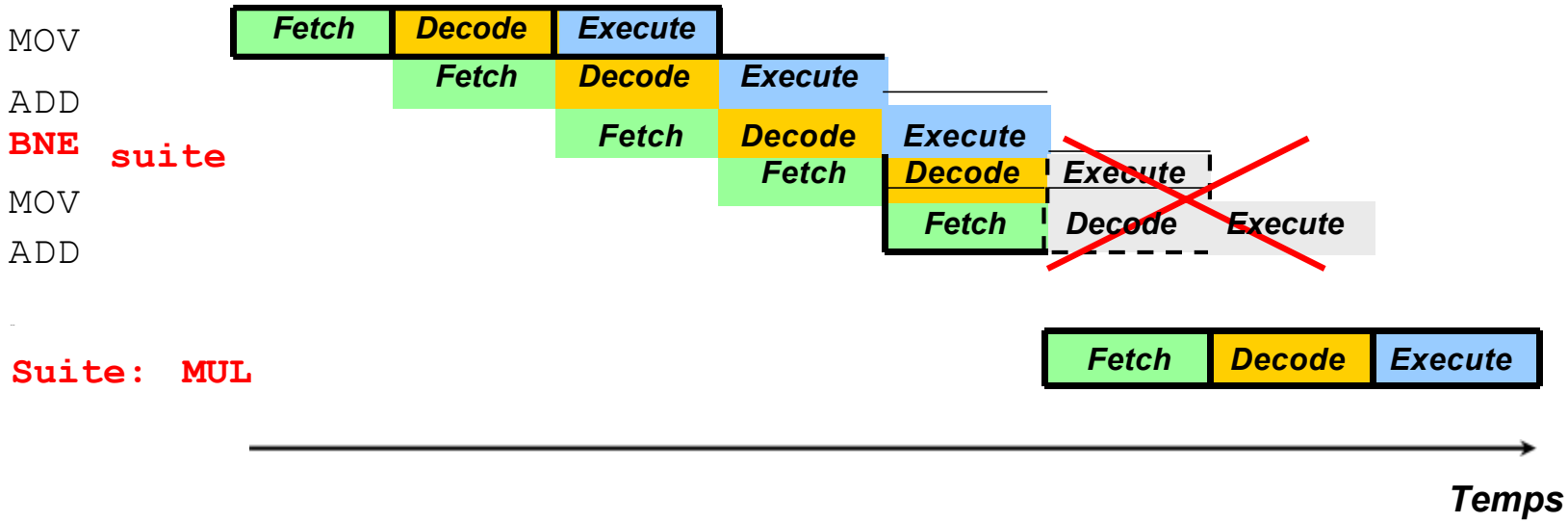
- 流水线减速：分支指令的情况（1）



  - 如果分支条件（BNE）为假，我们继续执行指令序列（MOV，ADD）。没有流水线中断。

Exécution pipeline

- 流水线减速：分支指令的情况（2）



- 如果分支条件（BNE）为真，我们取消执行接下来两条指令的执行，继续执行下一个标签（流水线重置）。
- 程序员的任务是编写汇编代码，有效地利用流水线来优化程序执行。
- 例如，通过使用条件执行而不是分支指令（如果可能），或者通过重新排列指令，以避免因数据不可用而导致速度减慢。

Meltdown和spectrum漏洞

https://www.macg.co/materiel/2018/01/meltdown-et-spectre-tout-savoir-sur-les-failles-historiques-des-processeurs-100954

https://www.ovh.com/fr/blog/failles-de-securite-spectre-meltdown-explication-3-failles-mesures-Correctives-public-averti /

https://meltdownattack.com/meltdown.pdf

指令的二进制编码

- 一条指令代表处理器的一个基本操作
  - 可用指令的集合称为指令集
  - 每条指令定义了机器执行的精确程序
- 在汇编程序中，指令被写成一个关键字，后跟着操作数。
  - 我们使用助记符是为了方便（以便更容易记住编写汇编程序）
  - 然后，汇编程序将助记符指令序列转换为机器可解释的二进制代码序列。
- 对于处理器来说，一条指令是由一个32位数编码的（二进制编码）。
  - 32位指令的每一位都有特定的作用

指令的二进制编码

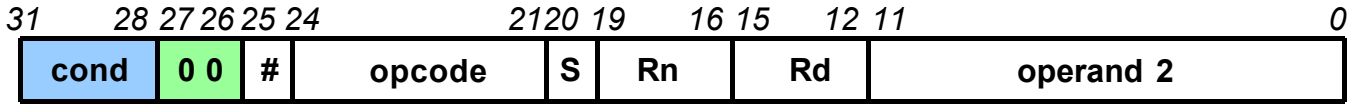
- 处理指令由字段组成
  - 结果寄存器
    - Rd: R0…R15
  - 第一个操作数（始终是寄存器）
    - Rn: R0…R15
  - 第二个操作数
    - Rm: R0…R15
    - Rm:带偏移量的R0…r15 (对数/算术/循环，数量)
    - 立即值（常量）
  - 由ALU执行的操作
    - 操作码
  - 影响 CPSR 状态
    - 后缀 S
  - 条件
    - 条件执行(EQ, LE, 等)

ARM指令

- ARM指令格式一般如下:
- <opcode>{<cond> {s} <Rd>, <Rn>{,<OP2>}
  - 格式中<>的内容是必不可少的，{}中的内容可忽略
  - <opcode>表示操作码。如ADD表示算术加法
  - {<cond>} 表示指令执行的条件域。如EQ、NE等，缺省为AL。
  - (S}决定指令的执行结果是否影响CPSR的值，使用该后缀则指令执行结果影响CPSR的值，否则不影响
  - <Rd> 表示目的寄存器
  - <Rn> 表示第一个操作数，为寄存器
  - <op2>表示第二个操作数，可以是立即数。寄存器和寄存器移位操作数

指令的二进制编码

- 处理指令 (1)



- *Cond*: 如果状态寄存器CPSR 满足指定条件，则执行该指令

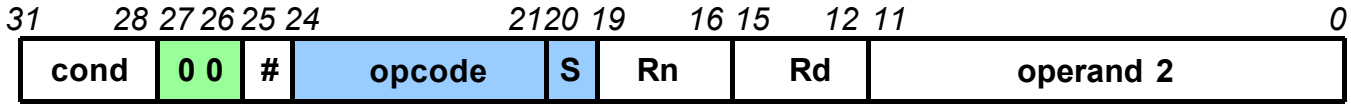
Asm	Cond
EQ	0000
NE	0001
CS/HS	0010
CC/LO	0011
MI	0100
PL	0101

Asm	Cond
VS	0110
VC	0111
HI	1000
LS	1001
GE	1010
LT	1011

Asm	Cond
GT	1100
LE	1101
AL	1110
NV	1111

Codage binaire

- 处理指令 (2)



- *S = 1*：影响 CPSR
- *Opcode操作码*：要执行的操作代码

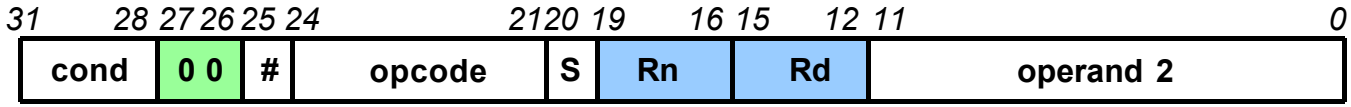
Asm	Opcode
AND	0000
EOR	0001
SUB	0010
RSB	0011
ADD	0100
ADC	0101

Asm	Opcode
SBC	0110
RSC	0111
TST	1000
TEQ	1001
CMP	1010
CMN	1011

Asm	Opcode
ORR	1100
MOV	1101
BIC	1110
MVN	1111

Codage binaire

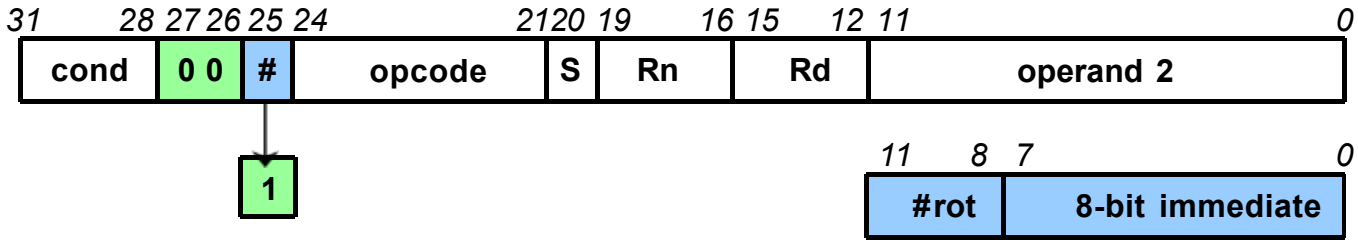
- 处理指令 (3)



- *Rd* → 目的寄存器编号
  - 4 位用于在 16 个可能的寄存器中进行选择
- *Rn* → 作为第一个操作数的寄存器编号
  - 4 位用于在 16 个可能的寄存器中进行选择
- *operand2 操作数2* → 连接到输入B
  - 可能循环/移位

Codage binaire

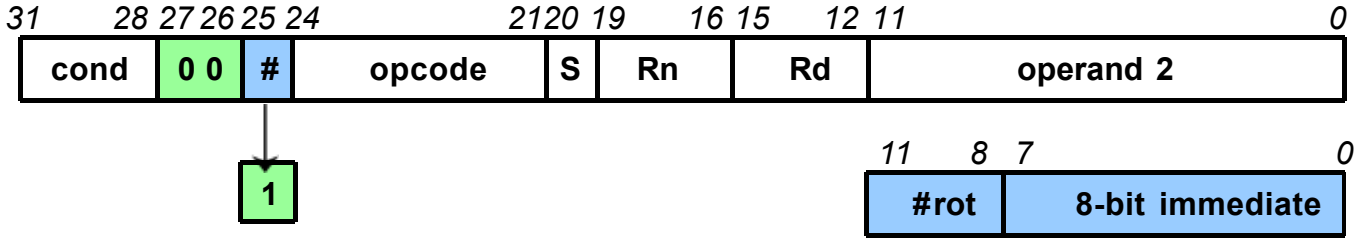
- 处理指令 (4)



- 如果位#为 1，则操作数2 为 32 位的立即数。
  - 一条指令以 32 位编码的，因此不可能对任何 32 位的数字值进行编码。
  - 该值是通过将 8 位值向右循环构造的。

Codage binaire

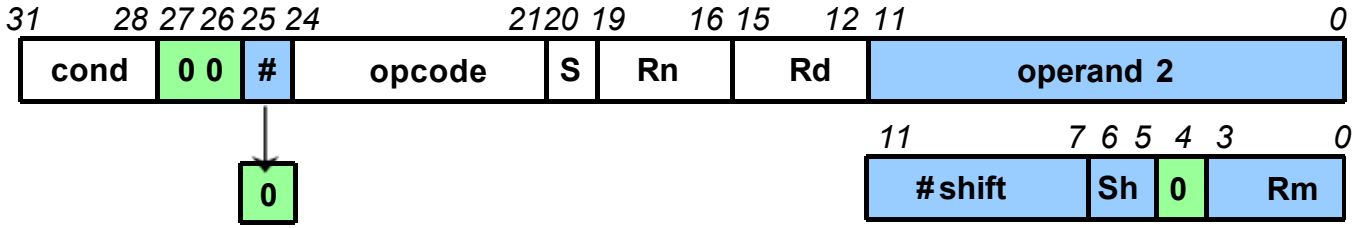
- 处理指令 (5)



- 该值是通过将 8 位值向右循环构造的。

###Codage binaire

- 处理指令 (6)



Asm	n	Sh	#shift
无	-	00	0
LSL #n	0 à 31	00	n
LSR #n	1 à 32	01	取n当n<32, 取0当n=32
ASR #n	1 à 32	10	取n当n<31, 取0当n=32
ROR #n	1 à 31	11	n
RRX	-	11	0

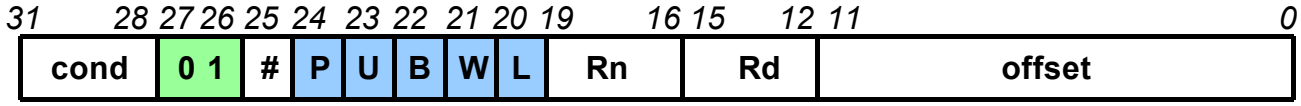
- 如果位#为 0，则操作数 2是应用（或不应用）循环/移位的寄存器(Rm)
  - 移位位数为
    - #shift 字面（5 位 32 值）
    - 寄存器 Rs 的内容（4位，用于从 R0… R15 中选择）

指令的二进制编码

- 传输指令
  - 读/写
  - 访问数据
    - 字、半字、字节
    - 有符号/无符号
  - 访问模式（前/后索引、+/-　偏移、回写）
  - 单次/多次传输
  - 源/目标寄存器、寄存器列表
  - 基址寄存器
  - 偏移
  - 条件
    - 有条件执行传输

Codage binaire

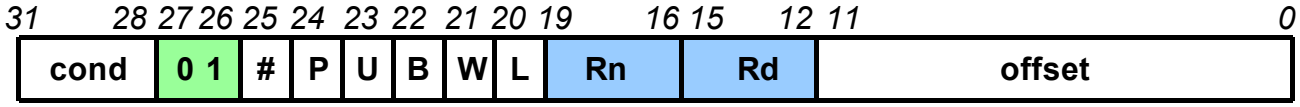
- 无符号字或字节传输指令 (LDR, STR, LDRB, STRB)



- *P*：前/后索引
  - 1个预索引，0个后索引
- *U*：上/下
  - 1 + 偏移, 0 –偏移
- *B*：字节/字
  - 1 访问 8 bits, 0 访问 32 bits
- *W*：回写
  - 如果 P=1, W=1 自动预索引寻址
- *L*：load/store
  - 1 加载, 0 存储

Codage binaire

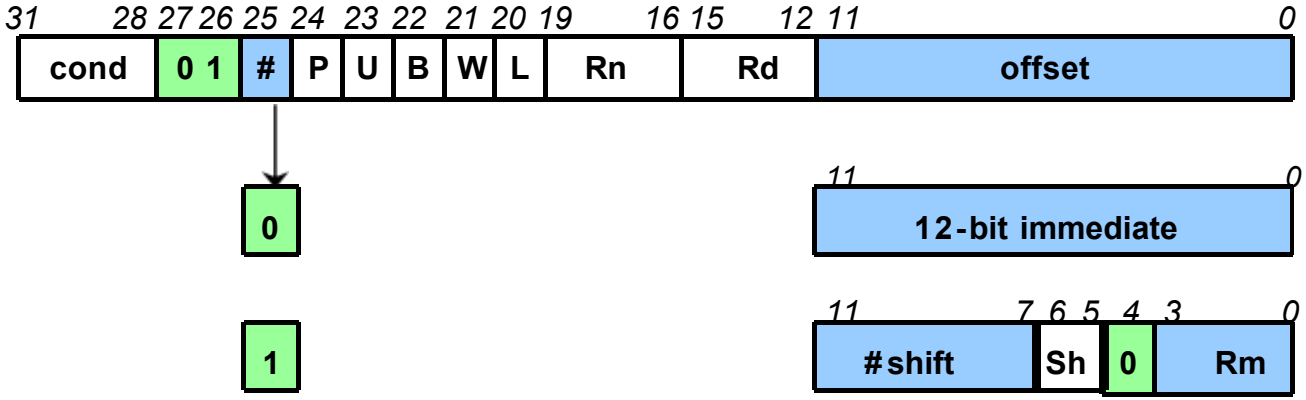
- 无符号字或字节传输指令 (LDR, STR, LDRB, STRB)



- *Rd* → 源寄存器 (如果 L=0, store存储) 或 目标寄存器 (如果 L=1, 加载)
- *Rn* → 基址寄存器

Codage binaire

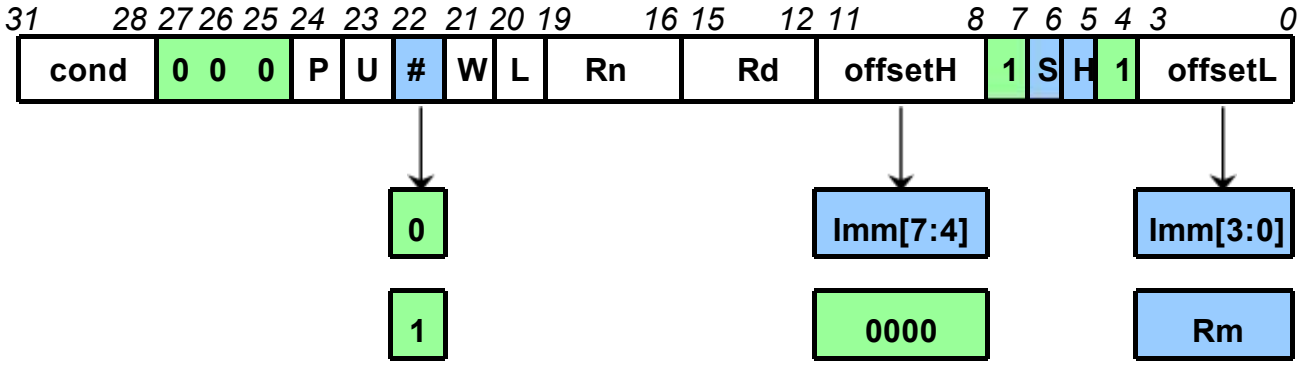
- 无符号字或字节传输指令 (LDR, STR, LDRB, STRB)



- 偏移量：一个12 位无符号数，或一个可按固定位数(#shift)移位的索引寄存器(Rm)

Codage binaire

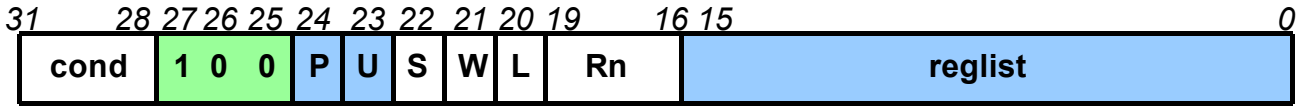
- 有符号半字或字节传输指令(LDRH, STRH, LDRSH, STRSH, LDRSB, STRSB)



- S：符号
  - 1 有符号, 0 无符号
- H：半字/字节
  - 1 访问16 bits, 0 访问 8 bits
- 偏移量：8 位立即数(Imm)或无偏移索引寄存器(Rm)

Codage binaire

- 多重传输指令 (LDM*mode*, STM*mode*)



mode	U	P
DA	0	0
DB	0	1
IA	1	0
IB	1	1

- reglist的每一位都控制着寄存器的传输：如果reglist[i] = 1，则寄存器ri将被传输。

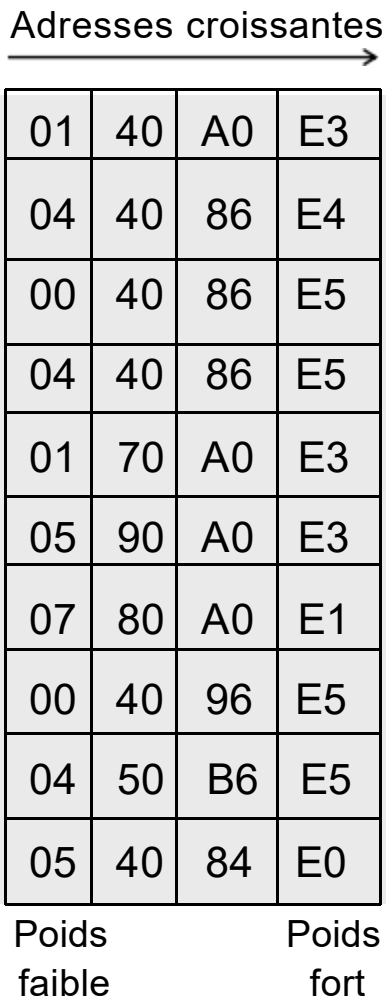
程序

- 对我们来说，程序就是一个在文本文件中一个接一个地排列并（可能）用标签标记的指令序列
- 对于处理器来说，一个程序是
  - 一系列32位字
  - … 符合有关处理器的指令编码约定
  - …以连续地址存储在内存中
  - 一条指令由其内存地址来标识

Programme

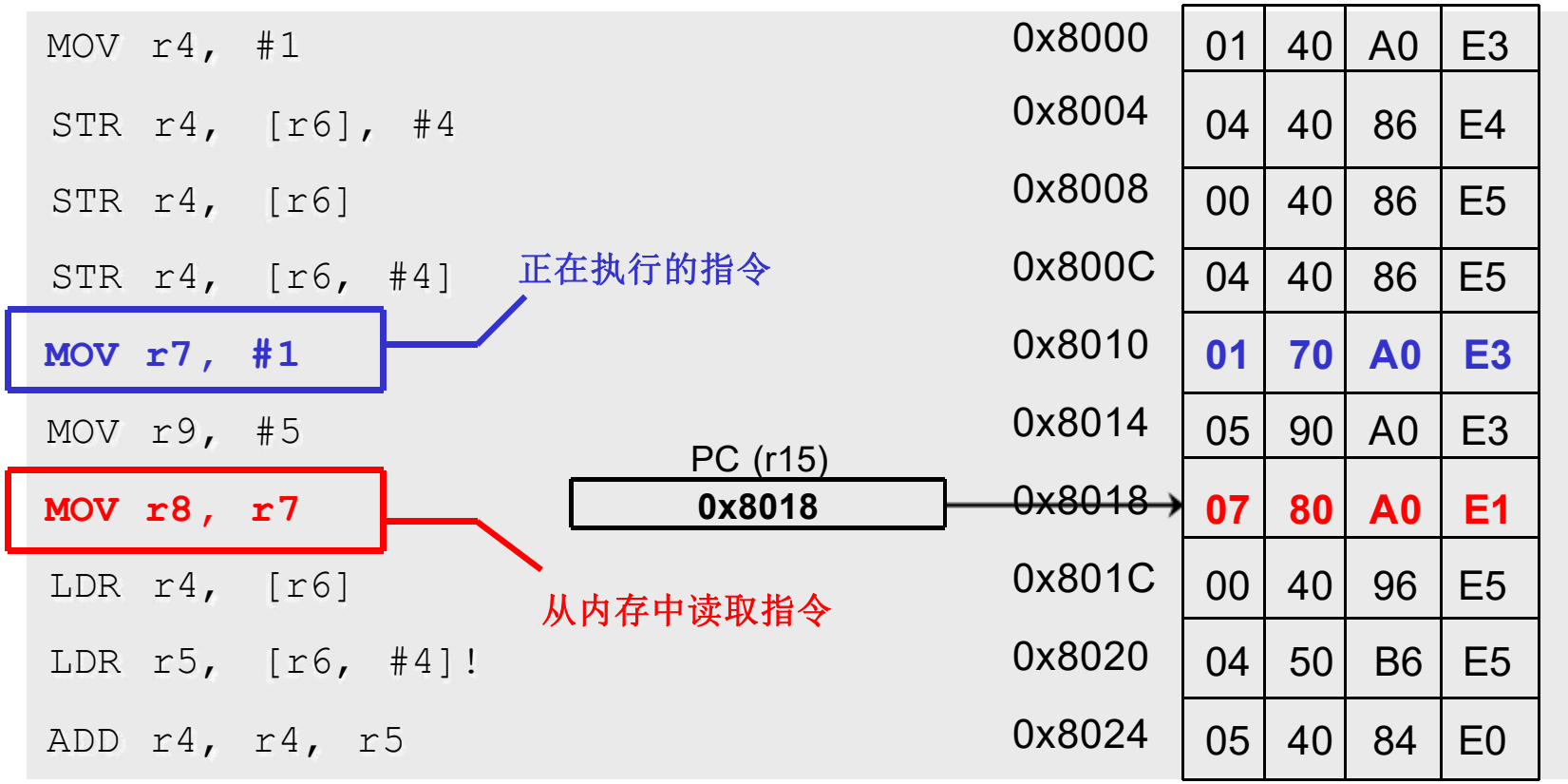
- 示例：在地址 0x8000 处执行的 ARM 汇编程序，以小端方式结构存储

```
MOV  r4,  #1
STR  r4,  [r6],  #4
STR  r4,  [r6]
STR  r4,  [r6,  #4]
MOV  r7,  #1
MOV  r9,  #5
MOV  r8,  r7
LDR  r4,  [r6]
LDR  r5,  [r6,  #4]!
ADD  r4,  r4,  r5
```



Programme

- “程序计数器” PC
  - 寄存器r15也称为PC（程序计数器）
  - 包含要读入内存的下一条指令的地址
  - 在执行过程中，以 4 累加
  - 下一条指令之前的两条指令



关系分支指令

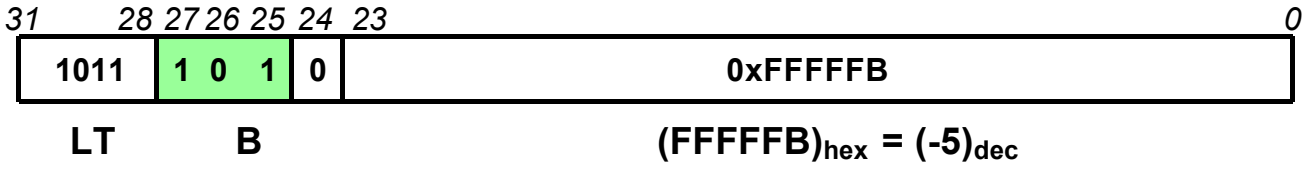
- Branch (B)
- Branch and Link (BL)

- offset：24 位有符号移动
- L：链接 (0 分支, 1 分支和链接)
- 效果:
  - B: PC ← PC + offset
  - BL: r14 ← PC – 4; PC = PC + offset
  - r14 ：链接寄存器(返回地址内容)

关系分支指令

- *for*循环的翻译示例
- 在汇编语言中使用标签：位移由汇编程序自动计算

```
                MOV     r4, #0          @ tmp=0
                MOV     r5, #0          @ i=0
loop:           ADD     r4, r4, r5      @ tmp+=i
                ADD     r5, r5, #1      @ i++
                CMP     r5, #5
                BLT     loop            @ i<5 : réitérer
                …
```



Représentation binaire: BAFFFFFB

关系分支指令

- *for循环的翻译示例*
- 汇编语言中地址的使用：由汇编器自动计算位移

```
          0x8000        MOV     r4, #0          @ tmp=0
          0x8004        MOV     r5, #0          @ i=0
(Loop:)   0x8008        ADD     r4, r4, r5      @ tmp+=i
          0x800C        ADD     r5, r5, #1      @ i++
          0x8010        CMP     r5, #5
          0x8014        BLT     0x8008 (Loop)   @ i<5 : réitérer
          0x8018        …
```