



第二章 ARM汇编语言

- 概述
- 寻址方式
- 数据处理指令
- 转移指令
- 函数调用
- 汇编语言
- 汇编程序举例
- 统一汇编语言
- 附录 1、ARM 和 Thumb-2 指令集
- 附录 2、机器指令格式

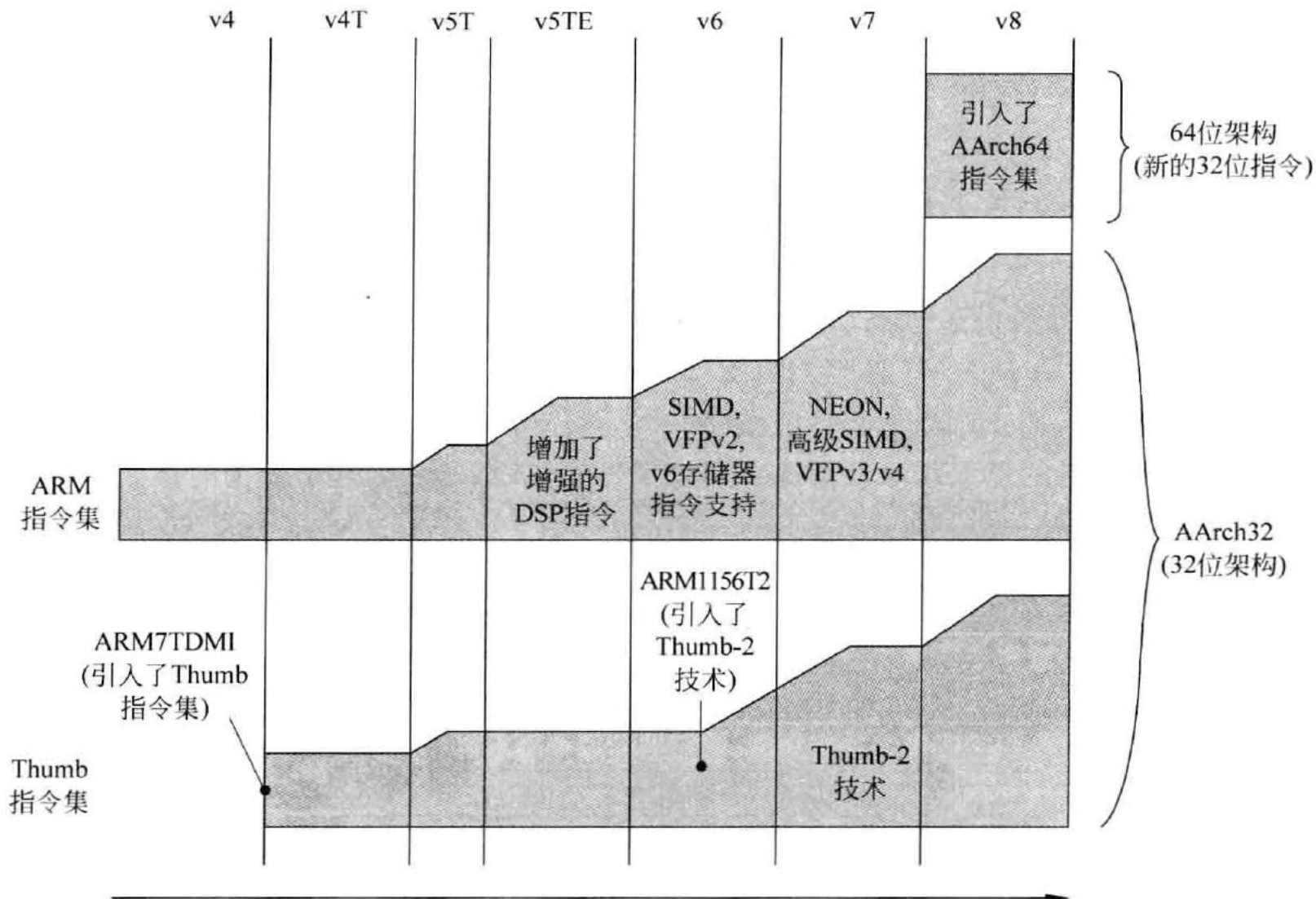


概述

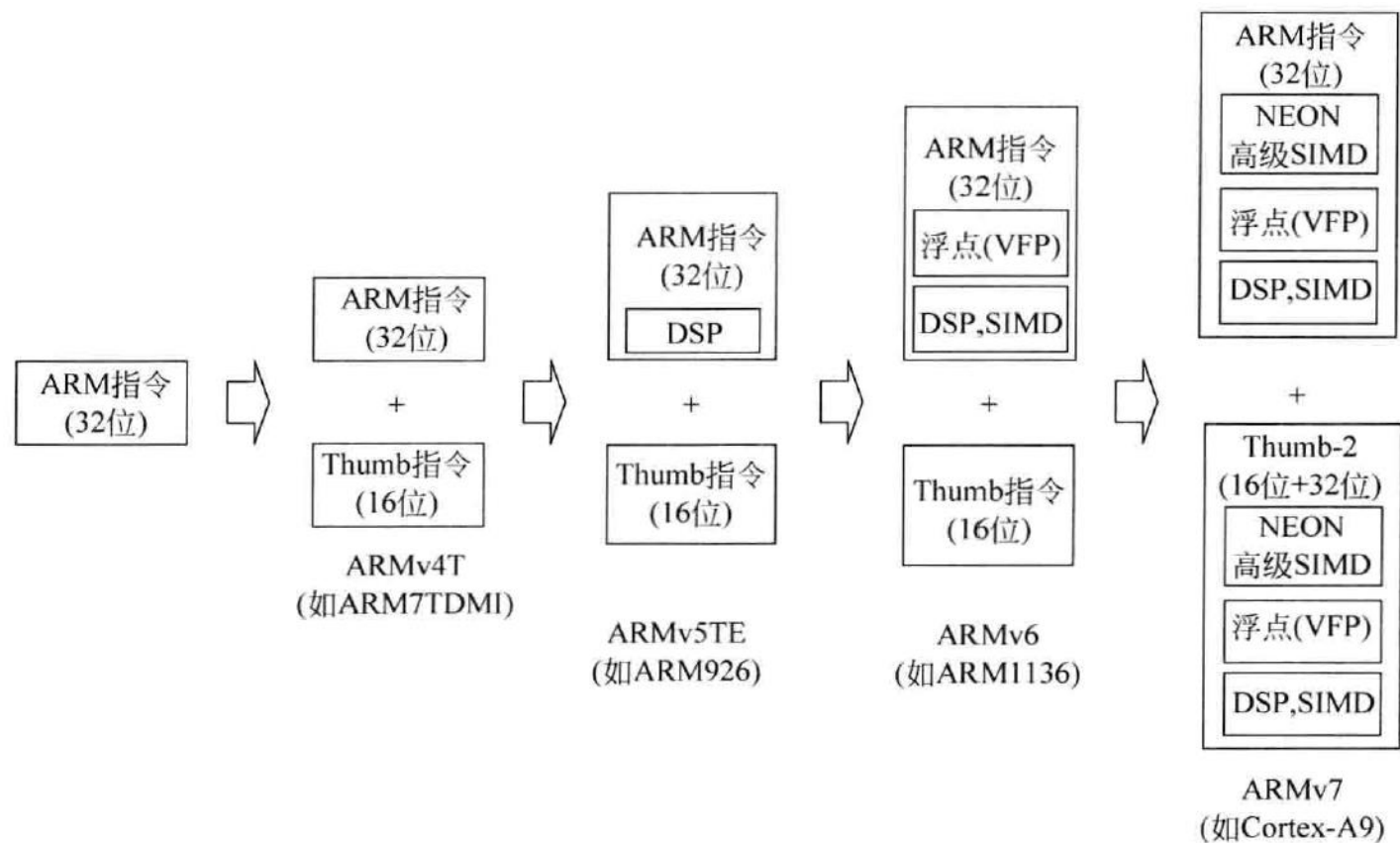
Thumb机器码

- 早期的ARM处理器仅支持32位ARM指令集。ARM指令集功能强大，但是需要更多的存储器空间。
- 为了减少程序占用的空间大小，在1995年，ARM推出了ARM7TDMI处理器，增加了一种16位指令集，即Thumb指令集。Thumb指令集提供了ARM指令集的大多数功能，但是可操作的寄存器，寻址模式，立即数范围都有所减少。
- ARM7TDMI通过ARM状态（默认）或Thumb状态下的切换可以交替使用这两种指令。它可以将Thumb指令翻译成ARM指令进行执行。
- 2003年，ARM推出了Thumb-2技术，将16位指令集和32位指令集集成到一种工作状态。Thumb-2指令集是Thumb指令集的超集，它的16位指令只能访问R0~R7，而32位指令(Thumb-2指令)则无任何限制，他们实现与ARM指令集一样的操作，但是指令编码方式不同。不过，有些指令不允许使用R15(PC)。
- Cortex-A处理器系列（Application）及Cortex-R处理器系列（Real-time），仍然支持ARM和Thumb两种工作状态。
- 所有的ARM Cortex-M 处理器(Microcontroller Processors)均基于Thumb-2技术，在一种工作状态中允许混合使用16位和32位指令。Cortex-M3处理器集成了Thumb-2技术，且仅支持Thumb工作态，但不支持ARM指令集。

- ARM指令集架构(ISA)发展



- ARM指令集架构(ISA)的演化



寻址方式

寻址方式就是确定操作数的来源，ARM有三种寻址方式：寄存器寻址，立即数寻址，存储器寻址。

它们的操作数分别存放在哪里？

存放在寄存器中，指令中，存储器中

每一种的优缺点？

速度快但容量小，速度较快但长度受限，容量大但速度慢

• 寄存器

高级语言

$a = b + c;$

ARM汇编语言

; R0 = a, R1 = b, R2 = c

ADD R0, R1, R2 ; $a = b + c$

高级语言

$a = b + c - d;$

ARM汇编语言

; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t

ADD R4, R1, R2 ; $t = b + c$

SUB R0, R4, R3 ; $a = t - d$

* R4为临时寄存器

高级语言

```
a = b - c;  
f = (g + h) - (i + j);
```

ARM汇编语言

```
; R0 = a, R1 = b, R2 = c, R3 = f  
; R4 = g, R5 = h, R6 = i, R7 = j  
SUB  R4, R1, R2 ; a = b - c  
ADD  R8, R4, R5 ; R8 = g + h  
ADD  R9, R6, R7 ; R9 = i + j  
SUB  R3, R8, R9 ; f = R8 - R9
```

ARM寄存器组

| | |
|----------|-------------|
| R0 | 参数/返回值/临时变量 |
| R1 ~ R3 | 参数/临时变量 |
| R4 ~ R11 | 保存的变量 |
| 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 |

• 立即数

高级语言

```
a = b + 27;  
b = a - 13;
```

ARM汇编语言

```
; R7 = a, R8 = b  
ADD  R7, R8, #27      ; a = a + 7  十进制  
SUB  R8, R7, #0xD     ; b = a - 13  十六进制
```

高级语言

```
i = 4080;  
x = 'A';
```

ARM汇编语言

```
; R4 = i, R5 = x  
MOV  R4, #0xFF0      ; i = 4080  
MOV  R5, #'A'         ; x = #65 'A'的ASCII码
```

• 存储器

- ARM作为RISC系统，数据处理指令的操作数只放在寄存器中，在存储器中的数据必须先移入寄存器才能处理。
- ARM系统采用32位存储器地址和32位数据字长，使用字节可寻址，每个字节都有一个地址。
- ARM系统一般采用小端序，即数据低字节存放在低地址处，数据的高字节存放在高地址处。大端序正好相反。

如果数据0x789AFFE0存放在地址0x00000088处：

| 地址 | 存储器 |
|------|-----|
| 0x8B | E0 |
| 0x8A | FF |
| 0x89 | 9A |
| 0x88 | 78 |
| | |

大端序

| 地址 | 存储器 |
|------|-----|
| 0x8B | 78 |
| 0x8A | 9A |
| 0x89 | FF |
| 0x88 | E0 |
| | |


小端序

高级语言

a = emp[3];

ARM汇编语言

```
; R7 = a
MOV R5, #0x80
LDR R7, [R5, #12]
```




```
; base address = 0x80
; R7<=value stored at R5+3*4
```

高级语言

emp[3] = 17;

ARM汇编语言

```
MOV R5, #0x80
MOV R9, #17
STR R9, [R5, #12]
```



```
; base address = 0x80
; #17 stored at R5+3*4
```


数据处理指令

数据处理指令包含加减乘除等算术运算指令、按位与或非等逻辑运算指令、逻辑移位和算术移位等移位指令。

• 算术运算指令

高级语言

$a = b - c * d;$

ARM汇编语言

$; R0 = a, R1 = b, R2 = c, R3 = d$

MUL R4, R2, R3 ; $R4 \leftarrow c * d$

SUB R0, R1, R4 ; $R0 \leftarrow b - R4$

| 常用算术指令(可选后缀未列出来) | 操 作 |
|---|----------------|
| ADD Rd, Rn, Rm ; $Rd = Rn + Rm$ | ADD 运算 |
| ADD Rd, Rn, # immed ; $Rd = Rn + \#immed$ | |
| ADC Rd, Rn, Rm ; $Rd = Rn + Rm + \text{进位}$ | 带进位的 ADD |
| ADC Rd, # immed ; $Rd = Rd + \#immed + \text{进位}$ | |
| ADDW Rd, Rn, # immed ; $Rd = Rn + \#immed$ | 寄存器和 12 位立即数相加 |
| SUB Rd, Rn, Rm ; $Rd = Rn - Rm$ | |
| SUB Rd, # immed ; $Rd = Rd - \#immed$ | 减法 |
| SUB Rd, Rn, # immed ; $Rd = Rn - \#immed$ | |
| SBC Rd, Rn, # immed ; $Rd = Rn - \#immed - \text{借位}$ | 带借位的减法 |
| SBC Rd, Rn, Rm ; $Rd = Rn - Rm - \text{借位}$ | |
| SUBW Rd, Rn, # immed ; $Rd = Rn - \#immed$ | 寄存器和 12 位立即数相减 |
| RSB Rd, Rn, # immed ; $Rd = \#immed - Rn$ | |
| RSB Rd, Rn, Rm ; $Rd = Rm - Rn$ | 减反转 |
| MUL Rd, Rn, Rm ; $Rd = Rn * Rm$ | |
| UDIV Rd, Rn, Rm ; $Rd = Rn / Rm$ | 无符号和有符号除法 |
| SDIV Rd, Rn, Rm ; $Rd = Rn / Rm$ | |

| 指令(由于 APSR 不更新,因此无 S 后缀) | 操 作 |
|--|-------------------------------|
| MLA Rd, Rn, Rm, Ra ; $Rd = Ra + Rn * Rm$ | 32 位 MAC 指令, 32 位结果 |
| MLS Rd, Rn, Rm, Ra ; $Rd = Ra - Rn * Rm$ | 32 位乘减指令, 32 位结果 |
| SMULL RdLo, RdHi, Rn, Rm ; $\{RdHi, RdLo\} = Rn * Rm$ | 有符号数据的 32 位乘 & MAC 指令, 64 位结果 |
| SMLAL RdLo, RdHi, Rn, Rm ; $\{RdHi, RdLo\} += Rn * Rm$ | |
| UMULL RdLo, RdHi, Rn, Rm ; $\{RdHi, RdLo\} = Rn * Rm$ | 无符号数据的 32 位乘 & MAC 指令, 64 位结果 |
| UMLAL RdLo, RdHi, Rn, Rm ; $\{RdHi, RdLo\} += Rn * Rm$ | |

• 逻辑运算指令

高级语言

$a = b \mid c \& d;$

* 按位与, 按位或 (32bits)

ARM汇编语言

; R0 = a, R1 = b, R2 = c, R3 = d

AND R4, R2, R3 ; R4 <= c & d

ORR R0, R1, R4 ; R0 <= b | R4

| 指令(可选的 S 后缀未列出来) | 操 作 |
|---|------|
| AND Rd, Rn ; Rd=Rd & Rn | 按位与 |
| AND Rd, Rn, # immed ; Rd=Rn & # immed | |
| AND Rd, Rn, Rm ; Rd=Rn & Rm | |
| ORR Rd, Rn ; Rd=Rd Rn | 按位或 |
| ORR Rd, Rn, # immed ; Rd=Rn # immed | |
| ORR Rd, Rn, Rm ; Rd=Rn Rm | |
| BIC Rd, Rn ; Rd=Rd & (~Rn) | 位清除 |
| BIC Rd, Rn, # immed ; Rd=Rn & (~ # immed) | |
| BIC Rd, Rn, Rm ; Rd=Rn & (~Rm) | |
| ORN Rd, Rn, # immed ; Rd=Rn (w # immed) | 按位或非 |
| ORN Rd, Rn, Rm ; Rd=Rn (wRm) | |
| EOR Rd, Rn ; Rd=Rd ^ Rn | 按位异或 |
| EOR Rd, Rn, # immed ; Rd=Rn # immed | |
| EOR Rd, Rn, Rm ; Rd=Rn Rm | |

• 移位指令

高级语言

a = b >> 4 + c;

* 按位与, 按位或 (32bits)

ARM汇编语言

; R0 = a, R1 = b, R2 = c

LSR R4, R1, #4 ; R4 <= b >> 4

ADD R0, R4, R2 ; R0 <= R4 + c

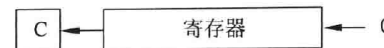
| | | Source register | | | |
|-----------------|----|-----------------|-----------|-----------|-----------|
| R5 | | 1111 1111 | 0001 1100 | 0001 0000 | 1110 0111 |
| Assembly Code | | Result | | | |
| LSL R0, R5, #7 | R0 | 1000 1110 | 0000 1000 | 0111 0011 | 1000 0000 |
| LSR R1, R5, #17 | R1 | 0000 0000 | 0000 0000 | 0111 1111 | 1000 1110 |
| ASR R2, R5, #3 | R2 | 1111 1111 | 1110 0011 | 1000 0010 | 0001 1100 |
| ROR R3, R5, #21 | R3 | 1110 0000 | 1000 0111 | 0011 1111 | 1111 1000 |

| | | Source registers | | | |
|----|--|------------------|-----------|-----------|-----------|
| R8 | | 0000 1000 | 0001 1100 | 0001 0110 | 1110 0111 |
| R6 | | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0100 |

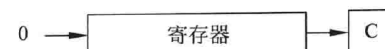
| Assembly code | | Result | | | |
|----------------|----|-----------|-----------|-----------|-----------|
| LSL R4, R8, R6 | R4 | 0110 1110 | 0111 0000 | 0000 0000 | 0000 0000 |
| ROR R5, R8, R6 | R5 | 1100 0001 | 0110 1110 | 0111 0000 | 1000 0001 |

LSR R1, R2 ; R1 <= R1>>R2

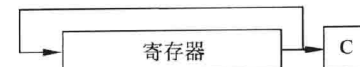
逻辑左移 (LSL)



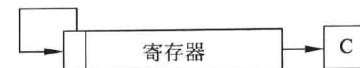
逻辑右移 (LSR)



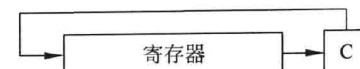
循环右移 (ROR)



算术右移 (ASR)



循环右移并展开 (RRX)



所有移出的位都移到进位位C中。

LSL - Logic Shift Left

LSR - Logic Shift Right

ASR - Arithmetic Shift Right

ROR - Rotate Right

条件运算指令

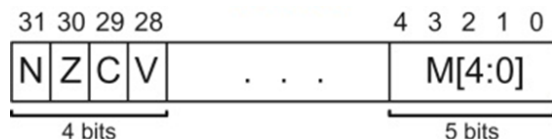
高级语言

```
a = 8; b = 10;
if(a==b) x=y+z;
if(a!=b) i=j-3;
if(a!=b) s=t+5;
```

ARM汇编语言

```
; R0 = a, R1 = b; R2 = x, R3 = y, R4 = z
; R5 = i, R6 = j, R8 = s, R9 = t
MOV R0, #8           ; a = 8
MOV R1, #10          ; b = 10
CMP R0, R1           ; R0-R1 只改变标志位
ITEE EQ              ; IF-THEN-ELSE-ELSE
ADD ADDEQ R2, R3, R4 ; if(a==b) x=y+z
SUBNE R5, R6, #3      ; if(a!=b) i=j-3;
ADDNE R8, R9, #5      ; if(a!=b) s=t+5;
```

CMP 指令会影响应用程序状态寄存器(APSR)中的标志位: N(Negative), Z(Zero), C(Carry), V(oVerflow)。



APSR-Application Program Status Register

TST R0, R1 ; R0 & R1, 只改变标志位
ADDEQ的条件靠IT指令, 其指令格式中没有EQ条件位。

| 条件码 | 助记符 | 说明 | 检测的条件位 |
|------|-------|--------------|-------------|
| 0000 | EQ | 相等/等于零 | Z=1 |
| 0001 | NE | 不等 | Z=0 |
| 0010 | CS/HS | 进位/无符号数大于或等于 | C=1 |
| 0011 | CC/LO | 无进位/无符号数小于 | C=0 |
| 0100 | MI | 负数 | N=1 |
| 0101 | PL | 正数或零 | N=0 |
| 0110 | VS | 溢出 | V=1 |
| 0111 | VC | 未溢出 | V=0 |
| 1000 | HI | 无符号数大于 | C=1 & Z=0 |
| 1001 | LS | 无符号数小于或等于 | C=0 or Z=1 |
| 1010 | GE | 有符号数大于或等于 | N=V |
| 1011 | LT | 有符号数小于 | N!=V |
| 1100 | GT | 有符号数大于 | Z=0 & N=V |
| 1101 | LE | 有符号数小于或等于 | Z=1 or N!=V |
| 1110 | AL | 总执行 | 任何状态 |
| 1111 | NV | 从不(不要使用) | 无 |

转移指令

如果希望要执行的下一条指令不是顺序的下一条指令，而是其他指令，就要使用转移指令。ARM的转移指令包含无条件分支指令、条件分支指令和分支链接(branch and link)指令。

高级语言

```
a = 8; b = 10;
if(a != b)
    x = y + z;
else
    x = y - 3;
```

ARM汇编语言

```
; R0 = a, R1 = b; R2 = x, R3 = y, R4 = z
MOV R0, #8          ; a = 8
MOV R1, #10         ; b = 10
CMP R0, R1          ; compare R0 with R1 (R0 - R1)
BEQ L1              ; if(a!=b) goto L1
ADD R2, R3, R4      ; x = y + z
B DONE              ; goto DONE
L1
SUB R2, R3, #3       ; x=y-3
DONE
```

| | |
|-------------------------------------|--------------|
| <u>BEQ - branch if equal (Z==1)</u> | <u>条件分支</u> |
| <u>B - branch</u> | <u>无条件分支</u> |

16位时PC加8位有符号数
16位时PC加11位有符号数

高级语言

```
switch(a) {  
    case 5:  b = 10; break;  
    case 8:  b = 20; break;  
    case 15: b = 36; break;  
    default: b = 48;  
}
```

ARM汇编语言

```
; R0 = a, R1 = b;  
CMP R0, #5  
BNE L1          ; if(a!=b) goto L1  
MOV R1, #10  
B    DONE       ; goto DONE  
L1  
CMP R0, #8  
BNE L2          ; if(a!=b) goto L1  
MOV R1, #20  
B    DONE       ; goto DONE  
L2  
CMP R0, #15  
BNE L3          ; if(a!=b) goto L1  
MOV R1, #36  
B    DONE       ; goto DONE  
L3  
MOV R1, #48  
  
DONE
```

高级语言

```
int pow = 1;
int x = 0;

while (pow!=128){
    pow = pow * 2;
    x = x + 1;
}
```

高级语言

```
int sum = 0;

for(int i=0;i<200;i++){
    sum = sum + i * i;
}
```

ARM汇编语言

```
; R0 = pow  R1 = x
MOV R0, #1           ; pow = 1
MOV R1, #0           ; x = 0
WHILE
    CMP R0, #128      ; pow==128?
    BEQ DONE          ; if(pow==128) goto DONE
    LSL R0, R0, #2
    ADD R1, R1, #1
    B WHILE           ; goto WHILE
DONE
```

ARM汇编语言

```
; R0 = i  R1 = sum
MOV R0, #0           ; i = 0
MOV R1, #0           ; sum = 0
LOOP
    CMP R0, #200      ; i>=200?
    BGE DONE          ; if(i>=200) goto DONE
    MUL R2, R0, R0     ; R2 = i * i
    ADD R1, R1, R2     ; sum = sum + i * i;
    ADD R0, R0, #1     ; i = i + 1;
    B LOOP            ; goto LOOP
DONE
```

函数调用

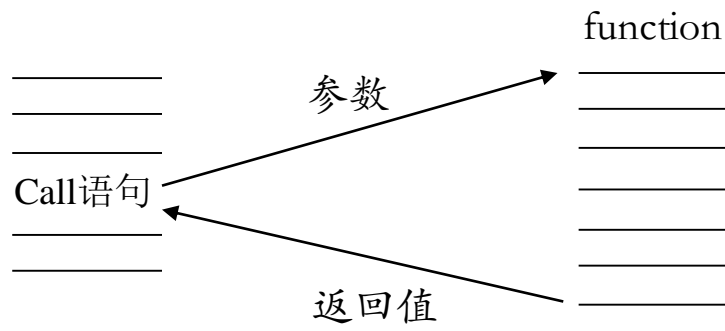
- 函数(function)调用可以在执行一段程序时转去执行另一段程序后再返回。
- 分支链接指令BL(branch and link)用于函数调用，它保存下一条指令的地址到LR(link register)，并转移到指定位置处。

高级语言

```
int main() {  
    simple();  
    ...  
}  
void simple(){  
    return;  
}
```

ARM汇编语言

```
0x00008000 MAIN    ...  
...  
0x00008020        BL SIMPLE  
...  
...  
0x0000902C SIMPLE MOV PC,LR    ; return  
...
```



调用前保存返回地址，以免函数内部对外部产生非预期影响。

函数(function)调用，也称为过程(procedure)或子程序(subroutine)

▫ 传递参数

高级语言

```
int main() {
    int y;
    ...
    y = diff(15,3,2,8);
    ...
}
int diff(int f, int g,
        int h, int i){
    result = (f + g) - (h + i)
    return result;
}
```

✓ ARM调用函数的惯例:

| | | |
|-------|----------------|------------|
| 返回地址 | <u>LR(R14)</u> | 蓝色为受保护寄存器 |
| 参数 | <u>R0~R3</u> | 黑色为不受保护寄存器 |
| 返回值 | <u>R0</u> | |
| 保存寄存器 | <u>R4~R11</u> | |
| 临时寄存器 | <u>R12</u> | |
| 栈指针 | <u>SP(R13)</u> | |
| 状态寄存器 | <u>APSR</u> | |

✓ 对于受保存寄存器, 函数中不能改变其值

✓ 一个没有调用其他函数的函数称为叶子函数。上面的DIFF就是叶子函数。

ARM汇编语言

```
; y = diff(15,3,2,8), R4 = y
```

MAIN

```
...
MOV R0, #15 ;argument0 = 15
MOV R1, #3  ;argument1 = 3
MOV R2, #2  ;argument2 = 2
MOV R3, #8  ;argument3 = 8
BL  DIFF
MOV R4, R0
...
```

; 参数: R0=f, R1=g, R2=h, R3=i

; 返回值: R0 = (f + g) - (h + i)

DIFF

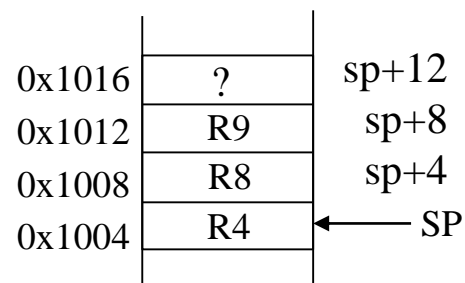
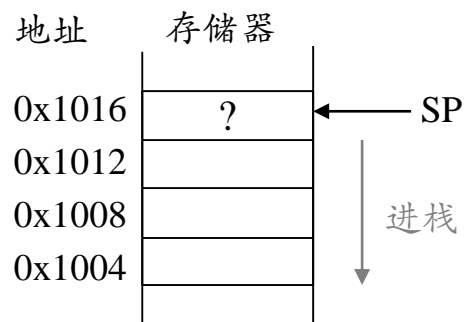
```
ADD R8, R0, R1 ; R8 = f + g
ADD R9, R2, R3 ; R9 = h + I
SUB R4, R8, R9 ; R4 = R8 - R9
MOV R0, R4
MOV PC, LR ; return
```

保存恢复现场

保存现场

DIFF

| | | |
|-------------------------|----------------|--------|
| <u>SUB SP, SP, #12</u> | ; 创建空间 | } 保存现场 |
| <u>STR R9, [SP, #8]</u> | ; 把R9保存到栈中 | |
| <u>STR R8, [SP, #4]</u> | ; 把R8保存到栈中 | |
| <u>STR R4, [SP]</u> | ; 把R4保存到栈中 | |
| ADD R8, R0, R1 | ; R8 = f + g | |
| ADD R9, R2, R3 | ; R9 = h + i | |
| SUB R4, R8, R9 | ; R4 = R8 - R9 | |
| MOV R0, R4 | ; 将返回值放在R0中 | |
| | | |
| <u>LDR R4, [SP]</u> | ; 从栈中恢复R4 | } 恢复现场 |
| <u>LDR R8, [SP, #4]</u> | ; 从栈中恢复R8 | |
| <u>LDR R9, [SP, #8]</u> | ; 从栈中恢复R9 | |
| <u>ADD SP, SP, #12</u> | ; 释放栈空间 | |
| MOV PC, LR | ; return | |



- ✓ 对于不受保存寄存器，在叶子函数中可以改变其值，但是受保护寄存器的值不能被改变。
- ✓ 进入函数时，要用栈来保存受保护寄存器的值，这叫**保存现场**，以便退出时恢复他们的原值，这叫**恢复现场**。
- ✓ 其他存储指令：
 - (1) 取存储单元(地址为Rn+offset)的一个字节/半个字/一个字到Rd中
LDRB/LDRH/LDR Rd, [Rn, #offset] (LDRSB/LDRSH - 符号扩展)
 - (2) 取存储单元(地址为Rn+offset)的双字到Rd1和Rd2中
LDRD Rd1, Rd2, [Rn, #offset]
 - (3) STRB/STRH/STR/STRD把字节/半字/字/双字保存到存储器中去，格式与(1)(2)类似

▣ 多寄存器存取指令

DIFF

STMDB SP!, {R4, R8, R9} ; SP每次减4, 依次把R4, R8和R9保存到栈中

ADD R8, R0, R1 ; R8 = f + g

ADD R9, R2, R3 ; R9 = h + i

SUB R4, R8, R9 ; R4 = R8 - R9

MOV R0, R4 ; 将返回值放在R0中

LDMIA SP!, {R4, R8, R9} ; SP每次加4, 依次把栈中内容取到R4, R8和R9

MOV PC, LR ; return

LDMIA/LDMDB/STMIA/STMDB Rn{!}, <reg list>

Rn - 保存地址的寄存器

IA - Increase address After STORE/LOAD

DB - Decrease address Before STORE/LOAD

! - 指令执行完修改Rn的值

<reg list> - 例: {R1, R4-R7, R9} 表示 R1, R4, R5, R6, R7, R9

LDMFD/LDMFA/LDMED/LDMEA STMFD/STMFA/STMED/STMEA

FD - 满下降, SP先减 (指向空位) 再存取

FA - 满上升, SP先加 (指向空位) 再存取

ED - 空下降, 先存取SP再先减 (指向空位)

EA - 空上升, 先存取SP再先加 (指向空位)

F-Full, SP始终指向栈顶元素, 即满的位置

E-EMPTY, SP始终指向紧邻栈顶元素的空位置

D - Decrease, A - Add

堆栈指令

DIFF

LR R4 R8 R9

PUSH {~~R4, R8, R9, LR~~} ; 依次把R4, R8, R9和LR的内容压到栈中

ADD R8, R0, R1 ; R8 = f + g

ADD R9, R2, R3 ; R9 = h + i

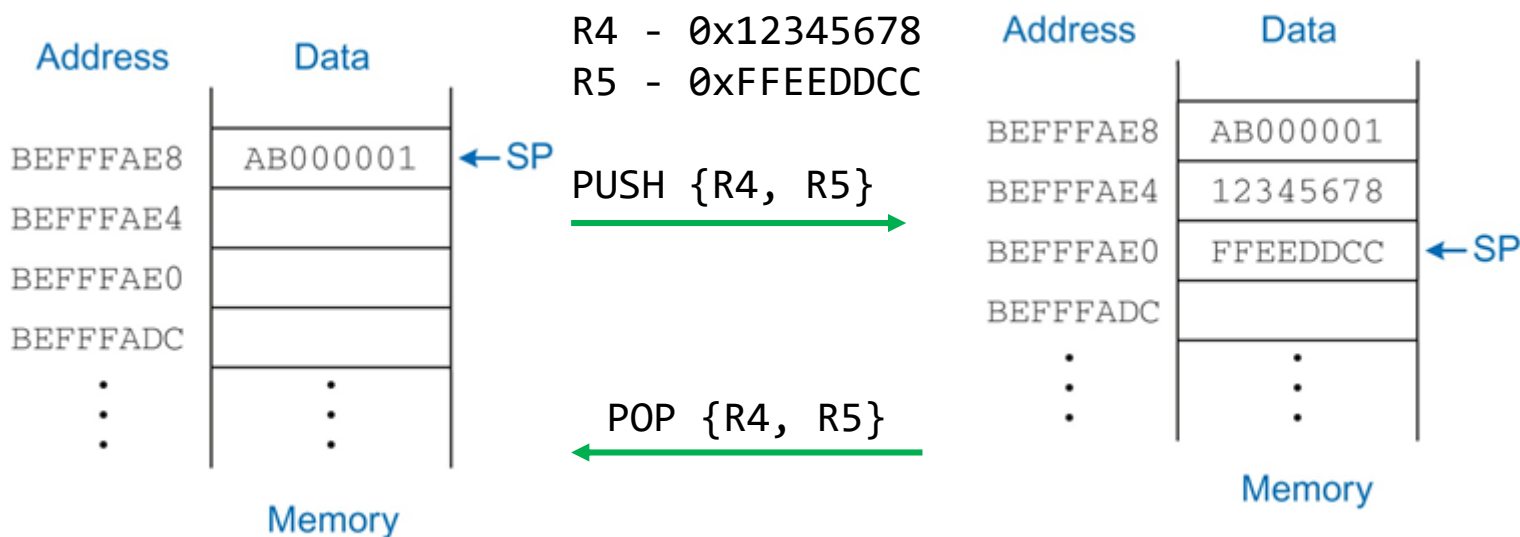
SUB R4, R8, R9 ; R4 = R8 - R9

MOV R0, R4 ; 将返回值放在R0中

POP {~~R4, R8, R9, PC~~} ; 把栈中内容弹出到R4, R8, R9和~~LR~~中

PC

PC R4 R8 R9



PUSH {R0, R4-R7, R9} ; 将R0, R4, R5, R6, R7, R9依次压入栈中

POP {R2, R6-R9} ; 将栈中内容依次取到R2, R6, R7, R8, R9中

▣ 非叶子函数

- ✓ 非叶函数是在函数内部调用了其他函数的函数。
- ✓ 对于非叶子函数，当它调用其他函数前需要保存不受保护寄存器的值，调用完毕后再恢复它们的值。

高级语言

```
; diff为非叶函数
int diff(int f, int g,
         int h, int i){
    result = sum(f,g) - sum(h,i)
    return result;
}

int sum(int x, int y){
    result = x + y
    return result;
}
```

不受保护的寄存器：参数R0~R3，返回值R0，临时寄存器R12

BL.W -- PC加22位有符号数

ARM汇编语言

```
; 参数:  R0=f, R1=g, R2=h, R3=i
; 返回值: R0 = sum(f, g) - sum(h,i)
DIFF    PUSH {R3,LR}    保护R3
        BL    SUM
        POP  {R3,LR}
        MOV   R8, R0      ;R8 = sum(f,g)
        MOV   R1, R2
        MOV   R2, R3
        PUSH {R3, LR}
        BL.W  SUM          ;加.W 32位指令
        POP  {R3, LR}
        SUB   R0, R8, R0
        MOV   PC, LR      ; return
```

```
; R1 = x, R2 = y
SUM     ADD   R3, R1, R2  改变了R3
        MOV   R0, R3
        MOV   PC, LR
```

▣ 递归函数

在函数内部调用自己的函数就是递归函数(recursive)。递归函数是非叶子函数。

高级语言

```
// n!
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return(n * fact(n-1))
    ...
}
```

ARM汇编语言

```
0x8500 FACT CMP R0, #1 ; n<=1?
0x8504      BGT ELSE      ; 否, 转移
0x8508      MOV R0, #1     ; 是, n=1
0x850C      MOV PC, LR     ; 返回
0x8510 ELSE PUSH {R0, LR} ; n和LR进栈
0x8514      SUB R0, R0, #1 ; R0=n-1
0x8518      BL FACT       ; 调用fact(n-1)
0x851C      MOV R1, R0     ; R1=fact(n-1)
0x8520      POP {R0, PC} LR ; n和LR出栈
0x8524      MUL R0, R0, R1 ; n*fact(n-1)
0x8528      MOV PC, LR     ; return
```

- ✓ 对于非叶子函数，当它调用其他函数前需要保存不受保护寄存器的值，调用完后再恢复它们的值。

给出fact(4)的栈变化？调用前SP=0xF000

▣ 栈帧

如果参数或局部变量太多，寄存器不够用，要用栈保存多出参数或变量的值。每次调用函数在栈顶形成一个栈帧。

高级语言

```
int f0 () {
    ...
    f1(8,9);
    ...
}

int f1(int a, int b) {
    int x = 5, y = 6;
    int z = f2(a + x, b + y)+10;
    return z + f2(x, y)
}

int f2(int x, int y){
    int a = 7, b = 8;
    return a * x + b * y;
}
```

- ✓ 可以采用符号名表记录变量和参数，并给出每个数据相对于SP的存放位置。

ARM汇编语言

; 假设参数、返回结果和局部变量只放在栈中

| | | | | | |
|----|----------------------------------|----|-------|-------|--|
| F0 | ... | | | | |
| | MOV R1, #8 | | | | |
| | MOV R2, #9 | | | | |
| | PUSH {R0, R1, R2} | 栈帧 | | | |
| | BL F1 | | | | |
| | ... | | | | |
| F1 | PUSH {LR} | 栈帧 | 返回值 | sp+28 | |
| | ADD SP, SP, #12 | | 参数a=8 | sp+24 | |
| | MOV R0, #5 | | 参数b=9 | sp+20 | |
| | STR R0, {SP, #8} ; x | | 返回地址 | sp+16 | |
| | MOV R0, #6 | | x | sp+12 | |
| | STR R0, {SP, #4} ; y | | y | sp+8 | |
| | ... | | z | sp+4 | |
| | LDR R0, {SP, #20} ; a | | | sp | |
| | LDR R1, {SP, #8} ; x | | | | |
| | ADD R0, R0, R1 ; R0 = a + x (参数) | | | | |
| | ... | | | | |
| | MOV LR, {SP, #12} ; 返回地址 | | | | |
| | ADD SP, SP, #12 ; 退栈 | | | | |
| | MOV PC, LR | | | | |

汇编语言

- 汇编语句

label Mnemonic operand1, operand2, ... ; 注释

标号 (地址) 助记符(操作) 操作数1(目的) 操作数2(源)

- label前面没有空格，可以与指令放在同一行
- 指令前面有空格

• 定义常量

```
NVIC_IRQ_SETEN          EQU    0xE000E100    ;指示性(directive)语句
NVIC_IRQ0_ENABLE        EQU    0x1
```

```
LDR  R0, =NVIC_IRQ_SETEN    ; 伪指令, 将0xE000E100放入R0
MOV  R1, #NVIC_IRQ0_ENABLE
STR  R1, [R0]                ; 将0x1存入地址为0xE000E100的存储单元
```

- ✓ 指示性(directive)语句指示汇编器做某些事, EQU用于定义符号常量, 汇编器将会把汇编语句中的符号常量替换为实际的常量。
- ✓ 伪指令是不能于一条机器指令对应的指令, 伪指令转换举例:

| | | | |
|----------------------------|---|---------|----------------------|
| LDR R0, =NVIC_IRQ_SETEN | ⇒ | 0x10020 | LDR R0, [PC, #0x40] |
| MOV R1, #NVIC_IRQ0_ENABLE | | 0x10024 | MOV R1, 0x1 |
| ... | | | ... |
| | | 0x10064 | DD 0xE000E100 |

• 定义数据

| | | | |
|--------|--------------------------|-----------------|---------------------------------|
| COUNT | EQU 23007 | | |
| LDR | R0, #3 | | |
| LDR | R3, =DIST | 等同, 只是上面的R0的值变了 | |
| LDR | R4, [R3, R0, LSL 2] | | ; 将存储单元[R3+(R0<<2)]读入R4 |
| MOVS | R5, R4 | | ; 指令加S影响标志位, R4为0时ZF=1 |
| ADD.W | R1, R5, R2 | | ; <u>.W-32位指令, .N-16位指令(默认)</u> |
| | ... | | |
| LDR | R0, =MSG | | |
| BL | PrintText | | |
| | ... | | |
| ALIGN | 4 | | ; 地址最低2位为0 指示性语句 |
| SCORES | DCB 0xA2, 56, 89 | | ; 保留6个字节 指示性语句 |
| | DCB 90, 86, 75 | | |
| HEIGHT | DCW 0x1234, 0x12, 34, 91 | | ; 4个半字 |
| DIST | DCD 0xF7654321, 82, 1004 | | ; 6个字 |
| | DCD 0x3E0, 68, COUNT | | |
| MSG | DCB "Hello\n", 0 | | ; 定义字符串 |
| | SPACE 0x30-8 | 48-8 | ; 留空40个字节 |
| | FILL 64, 0xA007, 2 | | ; 用0xA007填充32个半字 (共64个字节) |
| INS | DCI 0xBE00 | | ; 定义指令 |

• 代码段和数据段

求数组DATA1和DATA2对应的数据之和，并存入数组SUM中，一直计算到两数之和为零时结束。

```
AREA MyCode, CODE, READONLY ; 定义代码段
ENTRY ; 定义
START LDR R1,=DATA1 ; 数组DATA1的首地址存入到R1
      LDR R2,=DATA2 ; 数组DATA2的首地址存入到R2
      LDR R3,=SUM ; 数组SUM的首地址存入到R3
      MOV R0,#0 ; 计数器R0的初始值置0
LOOP  LDR R4,[R1],#04 ; 读存储器[R1]，然后修改R1为R1+4
      LDR R5,[R2],#04 ; 读存储器[R2]，然后修改R2为R2+4
      ADDS R4,R4,R5 ; 两数相加并影响标志位
      ADD R0,R0,#1 ; 计数器加1
      STR R4,[R3],#04 ; 保存结果[R3]，然后修改R3为R3+4
      BNE LOOP ; 若相加的结果不为0则循环
```

```
AREA MyData, DATA, READWRITE ; 定义数据段
DATA1 DCD 2,5,0,3,-4,5,0,10,9 ; 数组DATA1
DATA2 DCD 3,5,4,-2,0,8,3,-10,5 ; 数组DATA2
SUM DCD 0,0,0,0,0,0,0,0,0 ; 数组SUM
END
```

汇编程序举例

```
;*****  
;  
;          STM32 LED单向跑马灯实验          ;  
;          8个LED接在PE口 (PE[0..7])          ;  
;*****  
BIT6      EQU 0X00000040  
GPIOE     EQU 0X40011800 ;GPIOE 地址  
GPIOE_CRL EQU 0X40011800 ;低配置寄存器  
GPIOE_CRH EQU 0X40011804 ;高配置寄存器  
GPIOE_ODR EQU 0X4001180C ;输出, 偏移地址0Ch  
GPIOE_BSRR EQU 0X40011810 ;低置位, 高清除偏移地址10h  
GPIOE_BRR  EQU 0X40011814 ;清除, 偏移地址14h  
IOPEEN     EQU BIT6      ;GPIOE使能位  
RCC_APB2ENR EQU 0X40021018
```

```
STACK_TOP EQU 0X20002000
```

```
AREA RESET, CODE, READONLY
```

```
DCD STACK_TOP ; MSP主堆栈指针
```

```
DCD START ; 复位后PC初始值
```

```
ENTRY ; 指示开始执行
```

```
START
```

```
LDR R1, =RCC_APB2ENR
```

```
LDR R0, [R1] ; 读
```

```
LDR R2, =IOPEEN
```

```
ORR R0, R2 ; 改
```

```
STR R0, [R1] ; 写, 使能GPIOE时钟
```

从start开始执行

;PE[0..7] 8个引脚均设置成推挽式输出

```
LDR    R0,=0x33333333
LDR    R1,=GPIOE_CRL
STR    R0,[R1]
```

```
LDR    R1,=GPIOE_ODR
LDR    R0,=0X7F
```

;初始时最高位点亮

LOOP

```
STR    R0,[R1]
PUSH   {R0}
MOV    R0,#300
BL.W   DELAY_NMS ;延时300ms .W 使用32位指令
POP    {R0}
BL.W   ByteRor1  ;状态位右循环移一位
B      LOOP
```

;延时R0 (us) , 误差0.5us

;延时超过100us时, 误差0.5/R0小于0.5%

DELAY_NUS

```
SUB    R0,#1
```

```
NOP
```

```
NOP
```

```
NOP
```

```
CMP    R0,#0
```

```
BNE    DELAY_NUS
```

```
BX     LR ;PC<=LR 返回
```

;延时R0 (ms) , 误差((R0-1)*4+12)/8us 延时较长时, 误差小于0.1%

DELAY_NMS PUSH {R1} ;2个周期

DELAY_NMS2

SUB R0,#1

MOV R1,#1000

DELAY_ONEUS

SUB R1,#1

NOP

NOP

NOP

CMP R1,#0

BNE DELAY_ONEUS

CMP R0,#0

BNE DELAY_NMS2

POP {R1}

BX LR

;子程序, 将R0低八位右循环移一位, 高位不变

ByteRor1

PUSH {R1,R2,R3}

LDR R3,=0xFFFFFFFF00

LSR R1,R0,#1

AND R1,#0X0000007F

AND R2,R0,#0X01

LSL R2,#7

ORR R1,R2

AND R0,R3

ORR R0,R1

POP {R1,R2,R3}

BX LR

END

统一汇编语言(UAL)

- 为了提高构架间的软件可移植性，**ARM**开发工具开始支持统一汇编语言(**UAL**)，相同的汇编语句由汇编器确定使用什么机器指令。采用伪指令**CODE16**和伪指令**THUMB**分别指出是使用传统的**Thumb**模式还是**UAL**模式。
- 在传统的**Thumb**模式下，几乎所有的数据处理指令都会更新**CPSR**，而**Thumb-2**技术出现后，只有加了**S**后缀的数据处理指令才会更新**CPSR**。
 AND R0, R1 ;传统的**Thumb** 语法
 ANDS R0, R0, R1 ;等值的**UAL** 语法（必须有**S** 后缀）
- 在**UAL**下允许三操作数的一个源操作数就是目的操作数。在传统的**Thumb**模式下，这种指令会被编译成两操作数指令：
 ADD R0, R1 ;使用传统的**Thumb** 语法
 ADD R0, R0, R1 ; **UAL** 语法允许的等值写法（**R0=R0+R1**）
- 在**Thumb - 2** 指令集中，有些操作既可以由**16** 位指令完成，也可以由**32** 位指令完成。在**UAL** 下，可以让汇编器决定用哪个，也可以手工指定用哪个：
 ADDS R0, #1 ;汇编器将为了节省空间而使用**16** 位指令
 ADDS.N R0, #1 ;指定使用**16** 位指令(**N=Narrow**) (汇编器首选)
 ADDS.W R0, #1 ;指定使用**32** 位指令(**W=Wide**)(立即数太大时使用)
- 与**ARM ISA**要求字对齐不同，**32** 位的**Thumb-2** 指令也可以按半字对齐。
 0x1000: LDR r0, [r1] ;一个**16** 位的指令
 0x1002: RBIT.W r0 ;一个**32** 位的指令，跨越了字的边界

附录 1、ARM 和 Thumb-2 指令集

| 表关键字 | | | |
|---------------|---|--------------|---|
| Rm {, <opsh>} | 请参阅表寄存器, 可选择移动常数个位 | <reglist> | 以逗号隔开的寄存器列表, 括在大括号 { 和 } 内。 |
| <Operand2> | 请参阅表灵活的操作数 2。移位和循环移位只可用于 Operand2。 | <reglist-PC> | 作为 <reglist>, 不能包含 PC。 |
| <fields> | 请参阅表 PSR 字段。 | <reglist+PC> | 作为 <reglist>, 包含 PC。 |
| <PSR> | APSR (应用程序状态寄存器)、CPSR (当前处理器状态寄存器) 或 SPSR (保存的处理器状态寄存器) | <flags> | nzcvq (ALU 标记 PSR[31:27]) 或 g (SIMD GE 标记 PSR[19:16]) |
| C*, V* | 在体系结构 v4 及更早版本中, 标记不可预知; 在体系结构 v5 及以后版本中, 标记保持不变。 | \$ | 请参阅表 ARM 体系结构版本。 |
| <Rsh> | 可为 Rs 或一个立即数移位值。每种移位类型的允许值与表寄存器, 可选择移动常数个位 中的相同。 | +/- | + 或 -。(+ 可省略。) |
| x, y | B 或 T, B 表示半寄存器 [15:0], T 表示半寄存器 [31:16]。 | <iflags> | 中断标记。一个或多个 a、i、f (中止、中断、快速中断)。 |
| <imm8m> | ARM 32 位常数, 由 8 位值向右循环移偶数位生成。 Thumb 32 位常数, 由 8 位值左移任意位生成。 格式模式为 0xXYXYXYXY、0x0XY00XY 或 0xXY00XY00。 | <p_mode> | 请参阅表处理器模式 |
| <prefix> | 请参阅并行指令的前缀 | SPm | <p_mode> 所指定的处理模式的 SP |
| {IA IB DA DB} | 之后增加、之前增加、之后减小、之前减小。 IB 和 DA 不可用于 Thumb 状态下。如果省略, 则缺省为 IA。 | <lsb> | 位域的最低有效位。 |
| <size> | B、SB、H 或 SH, 含义分别为字节、有符号字节、半字和有符号半字。 SB 和 SH 不可用于 STR 指令。 | <width> | 位域宽度, <width> + <lsb> 必须 ≤ 32。 |
| | | {X} | 如果有 X, 则 RsX 为 Rs 循环 16 位生成。否则, RsX 为 Rs。 |
| | | {!} | 如果有 !, 则在数据传送完毕后更新基址寄存器 (前变址)。 |
| | | {S} | 如果有 S, 则更新条件标记。 |
| | | {T} | 如果有 T, 则带有用户模式特权。 |
| | | {R} | 如果存在 R, 则对结果进行舍入, 否则将其截断。 |

| 运算 | | \$ | 汇编器 | S 更新 | 操作 | 说明 |
|------|----------------------|----|---------------------------------|---------|---|-----|
| 加法 | 加法 | | ADD{S} Rd, Rn, <Operand2> | N Z C V | Rd := Rn + Operand2 | N |
| | 带进位 | | ADC{S} Rd, Rn, <Operand2> | N Z C V | Rd := Rn + Operand2 + 进位 | N |
| | 宽 | T2 | ADD Rd, Rn, #<imm12> | | Rd := Rn + imm12, imm12 的范围为 0-4095 | T、P |
| | 饱和 (加倍) | SE | Q{D}ADD Rd, Rm, Rn | | Rd := SAT(Rm + Rn) 加倍 Rd := SAT(Rm + SAT(Rn * 2)) | Q |
| 减法 | PC 相对的寻址 | | ADR Rd, <label> | | Rd := <label>, 有关 <label> 相对于当前指令的范围, 请参阅注释 L | N、L |
| | 减法 | | SUB{S} Rd, Rn, <Operand2> | N Z C V | Rd := Rn - Operand2 | N |
| | 带进位 | | SBC{S} Rd, Rn, <Operand2> | N Z C V | Rd := Rn - Operand2 - NOT (进位) | N |
| | 宽 | T2 | SUB Rd, Rn, #<imm12> | | Rd := Rn - imm12, imm12 的范围为 0-4095 | T、P |
| | 反向减法 | | RSB{S} Rd, Rn, <Operand2> | N Z C V | Rd := Operand2 - Rn | N |
| | 带进位反向减法 | | RSC{S} Rd, Rn, <Operand2> | N Z C V | Rd := Operand2 - Rn - NOT (进位) | A |
| 并行算法 | 饱和 (加倍) | SE | Q{D}SUB Rd, Rm, Rn | | Rd := SAT(Rm - Rn) 加倍 Rd := SAT(Rm - SAT(Rn * 2)) | Q |
| | 从异常中返回, 无出栈。 | | SUBS PC, LR, #<imm8> | N Z C V | PC = LR - imm8, CPSR = SPSR (当前模式), imm8 的范围为 0-255。 | |
| | 半字方式加法 | 6 | <prefix>ADD16 Rd, Rn, Rm | | Rd[31:16] := Rn[31:16] + Rm[31:16], Rd[15:0] := Rn[15:0] + Rm[15:0] | G |
| | 半字方式减法 | 6 | <prefix>SUB16 Rd, Rn, Rm | | Rd[31:16] := Rn[31:16] - Rm[31:16], Rd[15:0] := Rn[15:0] - Rm[15:0] | G |
| | 字节方式加法 | 6 | <prefix>ADD8 Rd, Rn, Rm | | Rd[31:24] := Rn[31:24] + Rm[31:24], Rd[23:16] := Rn[23:16] + Rm[23:16], Rd[15:8] := Rn[15:8] + Rm[15:8], Rd[7:0] := Rn[7:0] + Rm[7:0] | G |
| | 字节方式减法 | 6 | <prefix>SUB8 Rd, Rn, Rm | | Rd[31:24] := Rn[31:24] - Rm[31:24], Rd[23:16] := Rn[23:16] - Rm[23:16], Rd[15:8] := Rn[15:8] - Rm[15:8], Rd[7:0] := Rn[7:0] - Rm[7:0] | G |
| | 交换半字, 半字方式加法, 半字方式减法 | 6 | <prefix>ASX Rd, Rn, Rm | | Rd[31:16] := Rn[31:16] + Rm[15:0], Rd[15:0] := Rn[15:0] - Rm[31:16] | G |
| | 交换半字, 半字方减法, 半字方式加法 | 6 | <prefix>SAX Rd, Rn, Rm | | Rd[31:16] := Rn[31:16] - Rm[15:0], Rd[15:0] := Rn[15:0] + Rm[31:16] | G |
| 饱和 | 差值的绝对值无符号求和 | 6 | USAD8 Rd, Rm, Rs | | Rd := Abs(Rm[31:24] - Rs[31:24]) + Abs(Rm[23:16] - Rs[23:16]) + Abs(Rm[15:8] - Rs[15:8]) + Abs(Rm[7:0] - Rs[7:0]) | Q、R |
| | 差值的绝对值无符号求和, 再累加 | 6 | USADA8 Rd, Rm, Rs, Rn | | Rd := Rn + Abs(Rm[31:24] - Rs[31:24]) + Abs(Rm[23:16] - Rs[23:16]) + Abs(Rm[15:8] - Rs[15:8]) + Abs(Rm[7:0] - Rs[7:0]) | Q、R |
| | 有符号饱和和字, 右移 | 6 | SSAT Rd, #<sat>, Rm{, ASR <sh>} | | Rd := SignedSat((Rm ASR sh), sat), <sat> 的范围为 1-32, <sh> 的范围为 1-31。 | Q、R |
| | 有符号饱和和字, 左移 | 6 | SSAT Rd, #<sat>, Rm{, LSL <sh>} | | Rd := SignedSat((Rm LSL sh), sat), <sat> 的范围为 1-32, <sh> 的范围为 0-31。 | Q、R |
| | 有符号饱和和两个半字 | 6 | SSAT16 Rd, #<sat>, Rm | | Rd[31:16] := SignedSat(Rm[31:16], sat), Rd[15:0] := SignedSat(Rm[15:0], sat), <sat> 的范围为 1-16。 | Q |
| | 无符号饱和和字, 右移 | 6 | USAT Rd, #<sat>, Rm{, ASR <sh>} | | Rd := UnsignedSat((Rm ASR sh), sat), <sat> 的范围为 0-31, <sh> 的范围为 1-31。 | Q、R |
| 饱和 | 无符号饱和和字, 左移 | 6 | USAT Rd, #<sat>, Rm{, LSL <sh>} | | Rd := UnsignedSat((Rm LSL sh), sat), <sat> 的范围为 0-31, <sh> 的范围为 0-31。 | Q、R |
| | 无符号饱和和两个半字 | 6 | USAT16 Rd, #<sat>, Rm | | Rd[31:16] := UnsignedSat(Rm[31:16], sat), Rd[15:0] := UnsignedSat(Rm[15:0], sat), <sat> 的范围为 0-15。 | Q |

| 运算 | | \$ | 汇编器 | S 更新 | 操作 | 说明 |
|---------|------------------|----|------------------------------|-----------|--|------|
| 乘法 | 乘法 | | MUL{S} Rd, Rm, Rs | N Z C* | Rd := (Rm * Rs)[31:0] (如果 Rs 为 Rd, 则 S 可用于 Thumb-2 中) | N, S |
| | 并累加 | | MLA{S} Rd, Rm, Rs, Rn | N Z C* | Rd := (Rn + (Rm * Rs))[31:0] | S |
| | 乘减 | T2 | MLS Rd, Rm, Rs, Rn | | Rd := (Rn - (Rm * Rs))[31:0] | |
| | 无符号长乘法 | | UMULL{S} RdLo, RdHi, Rm, Rs | N Z C* V* | RdHi, RdLo := unsigned(Rm * Rs) | S |
| | 长整数无符号乘法 | | UMLAL{S} RdLo, RdHi, Rm, Rs | N Z C* V* | RdHi, RdLo := unsigned(RdHi, RdLo + Rm * Rs) | S |
| | 无符号长乘法, 两次加法 | 6 | UMAAAL RdLo, RdHi, Rm, Rs | | RdHi, RdLo := unsigned(RdHi + RdLo + Rm * Rs) | |
| | 长整数有符号乘法 | | SMULL{S} RdLo, RdHi, Rm, Rs | N Z C* V* | RdHi, RdLo := signed(Rm * Rs) | S |
| | 并累加 (长整数) | | SMLAL{S} RdLo, RdHi, Rm, Rs | N Z C* V* | RdHi, RdLo := signed(RdHi, RdLo + Rm * Rs) | S |
| | 16 * 16 位 | SE | SMULxy Rd, Rm, Rs | | Rd := Rm[x] * Rs[y] | |
| | 32 * 16 位 | SE | SMULWy Rd, Rm, Rs | | Rd := (Rm * Rs[y])[47:16] | |
| | 16 * 16 位并累加 | SE | SMLAxy Rd, Rm, Rs, Rn | | Rd := Rn + Rm[x] * Rs[y] | Q |
| | 32 * 16 位并累加 | SE | SMLAWy Rd, Rm, Rs, Rn | | Rd := Rn + (Rm * Rs[y])[47:16] | Q |
| | 长整数 16 * 16 位并累加 | SE | SMLALxy RdLo, RdHi, Rm, Rs | | RdHi, RdLo := RdHi, RdLo + Rm[x] * Rs[y] | |
| | 两次有符号乘法, 乘积相加 | 6 | SMUAD{X} Rd, Rm, Rs | | Rd := Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] | Q |
| | 并累加 | 6 | SMLAD{X} Rd, Rm, Rs, Rn | | Rd := Rn + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] | Q |
| | 并累加 (长整数) | 6 | SMLALD{X} RdLo, RdHi, Rm, Rs | | RdHi, RdLo := RdHi, RdLo + Rm[15:0] * RsX[15:0] + Rm[31:16] * RsX[31:16] | |
| | 两次有符号乘法, 乘积相减 | 6 | SMUSD{X} Rd, Rm, Rs | | Rd := Rm[15:0] * RsX[15:0] - Rm[31:16] * RsX[31:16] | Q |
| | 并累加 | 6 | SMLSD{X} Rd, Rm, Rs, Rn | | Rd := Rn + Rm[15:0] * RsX[15:0] - Rm[31:16] * RsX[31:16] | Q |
| | 并累加 (长整数) | 6 | SMLSLD{X} RdLo, RdHi, Rm, Rs | | RdHi, RdLo := RdHi, RdLo + Rm[15:0] * RsX[15:0] - Rm[31:16] * RsX[31:16] | |
| | 有符号高位字乘法 | 6 | SMMUL{R} Rd, Rm, Rs | | Rd := (Rm * Rs)[63:32] | |
| | 并累加 | 6 | SMMLA{R} Rd, Rm, Rs, Rn | | Rd := Rn + (Rm * Rs)[63:32] | |
| | 乘减 | 6 | SMMLS{R} Rd, Rm, Rs, Rn | | Rd := Rn - (Rm * Rs)[63:32] | |
| 除法 | 带内部 40 位累加 | XS | MIA Ac, Rm, Rs | | Ac := Ac + Rm * Rs | |
| | 组合半字 | XS | MIAPH Ac, Rm, Rs | | Ac := Ac + Rm[15:0] * Rs[15:0] + Rm[31:16] * Rs[31:16] | |
| | 半字 | XS | MIAxy Ac, Rm, Rs | | Ac := Ac + Rm[x] * Rs[y] | |
| | | | | | | |
| 移动数据 | 有符号或无符号 | RM | <op> Rd, Rn, Rm | | Rd := Rn / Rm <op> 为 SDIV (有符号) 或 UDIV (无符号) | T |
| | 移动 | | MOV{S} Rd, <Operand2> | N Z C | Rd := Operand2 请参阅移位指令 | N |
| | 求反移动 | | MVN{S} Rd, <Operand2> | N Z C | Rd := 0xFFFFFFFF EOR Operand2 | N |
| | 移到顶部 | T2 | MOVT Rd, #<imm16> | | Rd[31:16] := imm16, Rd[15:0] 不受影响, imm16 的范围为 0-65535 | |
| | 宽 | T2 | MOV Rd, #<imm16> | | Rd[15:0] := imm16, Rd[31:16] = 0, imm16 范围为 0-65535 | |
| 移位 | 40 位累加器到寄存器 | XS | MRA RdLo, RdHi, Ac | | RdLo := Ac[31:0], RdHi := Ac[39:32] | |
| | 寄存器到 40 位累加器 | XS | MAR Ac, RdLo, RdHi | | Ac[31:0] := RdLo, Ac[39:32] := RdHi[7:0] | |
| | 算术右移 | | ASR{S} Rd, Rm, <Rs sh> | N Z C | Rd := ASR(Rm, Rs sh) 与 MOV{S} Rd, Rm, ASR <Rs sh> 相同 | N |
| | 逻辑左移 | | LSL{S} Rd, Rm, <Rs sh> | N Z C | Rd := LSL(Rm, Rs sh) 与 MOV{S} Rd, Rm, LSL <Rs sh> 相同 | N |
| | 逻辑右移 | | LSR{S} Rd, Rm, <Rs sh> | N Z C | Rd := LSR(Rm, Rs sh) 与 MOV{S} Rd, Rm, LSR <Rs sh> 相同 | N |
| 计算前导零数目 | 向右循环移 | | ROR{S} Rd, Rm, <Rs sh> | N Z C | Rd := ROR(Rm, Rs sh) 与 MOV{S} Rd, Rm, ROR <Rs sh> 相同 | N |
| | 带扩展的向右循环移 | | RRX{S} Rd, Rm | N Z C | Rd := RRX(Rm) 与 MOV{S} Rd, Rm, RRX 相同 | |
| | | 5 | CLZ Rd, Rm | | Rd := Rm 中的前导零的数目 | |
| 比较 | 比较 | | CMP Rn, <Operand2> | N Z C V | 更新 Rn - Operand2 的 CPSR 标记 | N |
| | 与负数比较 | | CMN Rn, <Operand2> | N Z C V | 更新 Rn + Operand2 的 CPSR 标记 | N |
| 逻辑 | 测试 | | TST Rn, <Operand2> | N Z C | 更新 Rn AND Operand2 的 CPSR 标记 | N |
| | 相等测试 | | TEQ Rn, <Operand2> | N Z C | 更新 Rn EOR Operand2 的 CPSR 标记 | N |
| | 与 | | AND{S} Rd, Rn, <Operand2> | N Z C | Rd := Rn AND Operand2 | N |
| | 异或 | | EOR{S} Rd, Rn, <Operand2> | N Z C | Rd := Rn EOR Operand2 | N |
| | 或 | | ORR{S} Rd, Rn, <Operand2> | N Z C | Rd := Rn OR Operand2 | N |
| | 或非 | T2 | ORN{S} Rd, Rn, <Operand2> | N Z C | Rd := Rn OR NOT Operand2 | T |
| | 位清零 | | BIC{S} Rd, Rn, <Operand2> | N Z C | Rd := Rn AND NOT Operand2 | N |
| | | | | | | |

| 运算 | | S | 汇编器 | 操作 | 说明 |
|------------------|--------------------|----|---------------------------------|---|-------|
| 位域 | 位域清零 | T2 | BFC Rd, #<lsb>, #<width> | Rd[(width+lsb-1):lsb] := 0, Rd 的其他位不受影响 | |
| | 位域插入 | T2 | BFI Rd, Rn, #<lsb>, #<width> | Rd[(width+lsb-1):lsb] := Rn[(width-1):0], Rd 的其他位不受影响 | |
| | 有符号位域提取 | T2 | SBFX Rd, Rn, #<lsb>, #<width> | Rd[(width-1):0] = Rn[(width+lsb-1):lsb], Rd[31:width] = 复制 (Rn[width+lsb-1]) | |
| | 无符号位域提取 | T2 | UBFX Rd, Rn, #<lsb>, #<width> | Rd[(width-1):0] = Rn[(width+lsb-1):lsb], Rd[31:width] = 复制 (0) | |
| 组合 | 组合 低半字 + 高半字 | 6 | PKHBT Rd, Rn, Rm{, LSL #<sh>} | Rd[15:0] := Rn[15:0], Rd[31:16] := (Rm LSL sh)[31:16], sh 的范围为 0-31。 | |
| | 组合 高半字 + 低半字 | 6 | PKHTB Rd, Rn, Rm{, ASR #<sh>} | Rd[31:16] := Rn[31:16], Rd[15:0] := (Rm ASR sh)[15:0], sh 的范围为 1-32。 | |
| 有符号扩展 | 半字到字 | 6 | SXTH Rd, Rm{, ROR #<sh>} | Rd[31:0] := SignExtend((Rm ROR (8 * sh))[15:0]), sh 的范围为 0-3。 | N |
| | 两个字节到半字 | 6 | SXTB16 Rd, Rm{, ROR #<sh>} | Rd[31:16] := SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := SignExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| | 字节到字 | 6 | SXTB Rd, Rm{, ROR #<sh>} | Rd[31:0] := SignExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | N |
| 无符号扩展 | 半字到字 | 6 | UXTH Rd, Rm{, ROR #<sh>} | Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[15:0]), sh 的范围为 0-3。 | N |
| | 两个字节到半字 | 6 | UXTB16 Rd, Rm{, ROR #<sh>} | Rd[31:16] := ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| | 字节到字 | 6 | UXTB Rd, Rm{, ROR #<sh>} | Rd[31:0] := ZeroExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | N |
| 有符号扩展, 带加法 | 半字到字, 加法 | 6 | SXTAH Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[15:0]), sh 的范围为 0-3。 | |
| | 两个字节到半字, 加法 | 6 | SXTAB16 Rd, Rn, Rm{, ROR #<sh>} | Rd[31:16] := Rn[31:16] + SignExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + SignExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| | 字节到字, 加法 | 6 | SXTAB Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + SignExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| 无符号扩展, 带加法 | 半字到字, 加法 | 6 | UXTAH Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[15:0]), sh 的范围为 0-3。 | |
| | 两个字节到半字, 加法 | 6 | UXTAB16 Rd, Rn, Rm{, ROR #<sh>} | Rd[31:16] := Rn[31:16] + ZeroExtend((Rm ROR (8 * sh))[23:16]), Rd[15:0] := Rn[15:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| | 字节到字, 加法 | 6 | UXTAB Rd, Rn, Rm{, ROR #<sh>} | Rd[31:0] := Rn[31:0] + ZeroExtend((Rm ROR (8 * sh))[7:0]), sh 的范围为 0-3。 | |
| 反转 | 字中的位 | T2 | RBIT Rd, Rm | For (i = 0; i < 32; i++) : Rd[i] = Rm[31-i] | |
| | 字中的字节 | 6 | REV Rd, Rm | Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24] | N |
| | 两个半字中的字节 | 6 | REV16 Rd, Rm | Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24] | N |
| | 低半字中的字节, 符号扩展 | 6 | REVSH Rd, Rm | Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * 0xFFFF | N |
| 选择 | 选择字节 | 6 | SEL Rd, Rn, Rm | 如果 GE[0] = 1, 则 Rd[7:0] := Rn[7:0], 否则 Rd[7:0] := Rm[7:0] GE[1]、GE[2]、GE[3] 时位 [15:8]、[23:16]、[31:24] 的选择方法与 GE[0] 相似 | |
| 条件判断 | 条件判断 | T2 | IT{pattern} {cond} | 依据不同的模式, 最多由连续四个条件指令句组成。模式为一个字符串, 最多三个字母。所有字母都可作为 T (然后) 或 E (否则)。 IT 之后的第一条指令可有条件 cond。如果相应的字母为 T, 则后续指令可有条件 cond; 如果相应的字母为 E, 则为该 cond 的反面情况。 有关可用条件代码的信息, 请参阅表 条件字段 。 | T、U |
| 跳转 | 跳转 | | B <label> | PC := label, label 为此指令 ±32MB (T2: ±16MB, T: -252 - +256B) | N、B |
| | 带链接的跳转 | | BL <label> | LR := 下一指令的地址, PC := label, label 为此指令 ±32MB (T2: ±16MB)。 | |
| | 跳转并交换 | 4T | BX Rm | PC := Rm。如果 Rm[0] 为 1, 目标为 Thumb; 如果 Rm[0] 为 0, 目标则为 ARM。 | N |
| | 带链接和交换 (1) | 5T | BLX <label> | LR := 下一指令的地址, PC := label, 更改指令集。 label 为此指令 ±32MB (T2: ±16MB)。 | C |
| | 带链接和交换 (2) | 5 | BLX Rm | LR := 下一指令的地址, PC := Rm[31:1]。如果 Rm[0] 为 1, 更改为 Thumb; 如果 Rm[0] 为 0, 则更改为 ARM。 | N |
| | 跳转并更改为 Jazelle 状态 | 5J | BXJ Rm | 如果可用, 更改为 Jazelle | |
| | 比较, 如果为 (非) 零, 则跳转 | T2 | CB{N}Z Rn, <label> | 如果 Rn {== 或 !=} 0, 则 PC := label, label 为 (此指令 + 4-130)。 | N、T、U |
| 表跳转 | 表跳转字节 | T2 | TBB [Rn, Rm] | PC = PC + ZeroExtend(Memory(Rn + Rm, 1) << 1)。跳转范围为 4-512。Rn 可为 PC。 | T、U |
| | 表跳转半字 | T2 | TBH [Rn, Rm, LSL #1] | PC = PC + ZeroExtend(Memory(Rn + Rm << 1, 2) << 1)。跳转范围为 4-131072。Rn 可为 PC。 | T、U |
| 移到 PSR 或从 PSR 移出 | PSR 到寄存器 | | MRS Rd, <PSR> | Rd := PSR | |
| | 寄存器标记到 APSR 标记 | | MSR APSR_<flags>, Rm | APSR_<flags> := Rm | |
| | 立即数标记到 APSR 标记 | | MSR APSR_<flags>, #<imm8m> | APSR_<flags> := imm8_r | |
| | 寄存器到 PSR | | MSR <PSR>_<fields>, Rm | PSR := Rm (仅选择字节) | |
| | 立即数到 PSR | | MSR <PSR>_<fields>, #<imm8m> | PSR := imm8_r (仅选择字节) | |
| 处理器状态更改 | 更改处理器状态 | 6 | CPSID <iflags> {, #<p_mode>} | 禁用指定的中断, 可选择更改模式。 | U、N |
| | 改变处理器模式 | 6 | CPSIE <iflags> {, #<p_mode>} | 启用指定的中断, 可选择更改模式。 | U、N |
| | 设置端标记 | 6 | CPS #<p_mode> | | U |
| | | 6 | SETEND <endianness> | 为加载和存储设置端标记。<endianness> 可为 BE (大端) 或 LE (小端)。 | U、N |

| 加载和存储单个数据项 | | \$ | 汇编器 | 当 <op> 为 LDR 时执行的操作 | 当 <op> 为 STR 时执行的操作 | 说明 |
|--------------|----------|----|---|--|--|-----|
| 加载或存储字、字节或半字 | 直接偏移量 | | <op>{size}{T} Rd, [Rn {, #<offset>}]{!} | Rd := [address, size] | [address, size] := Rd | 1、N |
| | 后变址, 立即数 | | <op>{size}{T} Rd, [Rn], #<offset> | Rd := [address, size] | [address, size] := Rd | 2 |
| | 寄存器偏移量 | | <op>{size} Rd, [Rn, +/-Rm {, <opsh>}]{!} | Rd := [address, size] | [address, size] := Rd | 3、N |
| | 后变址, 寄存器 | | <op>{size}{T} Rd, [Rn], +/-Rm {, <opsh>} | Rd := [address, size] | [address, size] := Rd | 4 |
| 加载或存储双字 | PC 相对的 | | <op>{size} Rd, <label> | Rd := [label, size] | 不可用 | 5、N |
| | 直接偏移量 | 5E | <op>D Rd1, Rd2, [Rn {, #<offset>}]{!} | Rd1 := [address], Rd2 := [address + 4] | [address] := Rd1, [address + 4] := Rd2 | 6、9 |
| | 后变址, 立即数 | 5E | <op>D Rd1, Rd2, [Rn], #<offset> | Rd1 := [address], Rd2 := [address + 4] | [address] := Rd1, [address + 4] := Rd2 | 6、9 |
| | 寄存器偏移量 | 5E | <op>D Rd1, Rd2, [Rn, +/-Rm {, <opsh>}]{!} | Rd1 := [address], Rd2 := [address + 4] | [address] := Rd1, [address + 4] := Rd2 | 7、9 |
| | 后变址, 寄存器 | 5E | <op>D Rd1, Rd2, [Rn], +/-Rm {, <opsh>} | Rd1 := [address], Rd2 := [address + 4] | [address] := Rd1, [address + 4] := Rd2 | 7、9 |
| | PC 相对的 | 5E | <op>D Rd1, Rd2, <label> | Rd1 := [label], Rd2 := [label + 4] | 不可用 | 8、9 |

| 预载数据或指令 | | \$(PLD) | \$(PLI) | \$(PLDW) | 汇编器 | 当 <op> 为 PLD 时执行的操作 | 当 <op> 为 PLI 时执行的操作 | 当 <op> 为 PLDW 时执行的操作 | 说明 |
|---------|--------|---------|---------|----------|-----------------------------|-----------------------|-----------------------|--------------------------|-----|
| | 直接偏移量 | 5E | 7 | 7MP | <op> [Rn {, #<offset>}] | 预载 [address, 32] (数据) | 预载 [address, 32] (指令) | 预载以写入 [address, 32] (数据) | 1、C |
| | 寄存器偏移量 | 5E | 7 | 7MP | <op> [Rn, +/-Rm {, <opsh>}] | 预载 [address, 32] (数据) | 预载 [address, 32] (指令) | 预载以写入 [address, 32] (数据) | 3、C |
| | PC 相对的 | 5E | 7 | | <op> <label> | 预载 [label, 32] (数据) | 预载 [label, 32] (指令) | | 5、C |

| 其他内存操作 | | \$ | 汇编器 | 操作 | 说明 |
|--------|-----------|----|--|---|-----|
| 加载多个 | 数据块加载 | | LDM{IA IB DA DB} Rn{!}, <reglist>-PC | 从 [Rn] 加载寄存器列表 | N、I |
| | 返回 (并交换) | | LDM{IA IB DA DB} Rn{!}, <reglist>+PC | 加载寄存器, PC := [address][31:1] (§ 5E 当 [address][0] 为 1 时, 更改为 Thumb) | I |
| | 并恢复 CPSR | | LDM{IA IB DA DB} Rn{!}, <reglist>+PC^A | 加载寄存器, 跳转 (§ 5E 并交换), CPSR := SPSR。仅限异常模式。 | I |
| | 用户模式寄存器 | | LDM{IA IB DA DB} Rn, <reglist>-PC^A | 从 [Rn] 加载用户模式寄存器列表。仅限特权模式。 | I |
| 弹出 | | | POP <reglist> | LDM SP!, <reglist> 的规范格式 | N |
| 加载独占 | 信号运算 | 6 | LDREX Rd, [Rn] | Rd := [Rn], 将地址标记为独占访问。如果不是共享地址, 则为突出显示的标记设置。Rd、Rn 不可为 PC。 | |
| | 半字或字节 | 6K | LDREX{H B} Rd, [Rn] | Rd[15:0] := [Rn] 或 Rd[7:0] := [Rn], 将地址标记为独占访问。如果不是共享地址, 则为突出显示的标记设置。Rd、Rn 不可为 PC。 | |
| | 双字 | 6K | LDREXD Rd1, Rd2, [Rn] | Rd1 := [Rn], Rd2 := [Rn+4], 将地址标记为独占访问。如果不是共享地址, 则为突出显示的标记设置。Rd1、Rd2、Rn 不可为 PC。 | 9 |
| 存储多个 | 推入或阻止数据存储 | | STM{IA IB DA DB} Rn{!}, <reglist> | 将寄存器列表存储到 [Rn] 中 | N、I |
| | 用户模式寄存器 | | STM{IA IB DA DB} Rn{!}, <reglist>^A | 将用户模式寄存器列表存储到 [Rn] 中。仅限特权模式。 | I |
| 推入 | | | PUSH <reglist> | STMDB SP!, <reglist> 的规范格式 | N |
| 存储独占 | 信号运算 | 6 | STREX Rd, Rm, [Rn] | 如果允许, 则 [Rn] := Rm, 清除独占标记, Rd := 0。否则 Rd := 1。Rd、Rm、Rn 不可为 PC。 | |
| | 半字或字节 | 6K | STREX{H B} Rd, Rm, [Rn] | 如果允许, 则 [Rn] := Rm[15:0] 或 [Rn] := Rm[7:0], 清除独占标记, Rd := 0。否则 Rd := 1。Rd、Rm、Rn 不可为 PC。 | |
| | 双字 | 6K | STREXD Rd, Rm1, Rm2, [Rn] | 如果允许, 则 [Rn] := Rm1, [Rn+4] := Rm2, 清除独占标记, Rd := 0。否则 Rd := 1。Rd、Rm1、Rm2、Rn 不可为 PC。 | 10 |
| 清除独占 | | 6K | CLREX | 清除局部处理器独占标记 | C |

| 说明 加载、存储和预载操作的可用性和选项范围 | | | | | |
|------------------------|--------------------------------------|-----------------|-------------------|--|-------------------------|
| 注释 | ARM 字、B、D | ARM SB、H、SH | ARM T、BT | Thumb-2 字、B、SB、H、SH、D | Thumb-2 T、BT、SBT、HT、SHT |
| 1 | 偏移量 -4095 到 +4095 | 偏移量 -255 到 +255 | 不可用 | 偏移量 如果回写, 则为 -255 到 +255, 否则, 为 -255 到 +4095 | 偏移量 0 到 +255, 不允许回写 |
| 2 | 偏移量 -4095 到 +4095 | 偏移量 -255 到 +255 | 偏移量 -4095 到 +4095 | 偏移量 -255 到 +255 | 不可用 |
| 3 | 整个 {, <opsh>} 范围 | {, <opsh>} 不允许 | 不可用 | <opsh> 限制为 LSL #<sh>, <sh> 的范围为 0 到 3 | 不可用 |
| 4 | 整个 {, <opsh>} 范围 | {, <opsh>} 不允许 | 整个 {, <opsh>} 范围 | 不可用 | 不可用 |
| 5 | 当前指令的 +/- 4092 范围内的标签 | 不可用 | 不可用 | 当前指令的 +/- 4092 范围内的标签 | 不可用 |
| 6 | 偏移量 -255 到 +255 | - | - | 偏移量 -1020 到 +1020, 必须是 4 的倍数。 | - |
| 7 | {, <opsh>} 不允许 | - | - | 不可用 | - |
| 8 | 当前指令的 +/- 252 范围内的标签 | - | - | 不可用 | - |
| 9 | Rd1 编号为偶数, 但不可为 r14, Rd2 == Rd1 + 1。 | - | - | Rd1 != PC, Rd2 != PC | - |
| 10 | Rm1 编号为偶数, 但不可为 r14, Rm2 == Rm1 + 1。 | - | - | Rm1 != PC, Rm2 != PC | - |

| 协处理器运算 | \$ | 汇编器 | 操作 | 说明 |
|-----------------|----|---|---------------------------------------|---------|
| 数据操作 | | CDP{2} <copr>, <op1>, CRd, CRn, CRm{, <op2>} | | 由协处理器定义 |
| 从协处理器移到 ARM 寄存器 | | MRC{2} <copr>, <op1>, Rd, CRn, CRm{, <op2>} | | 由协处理器定义 |
| 两个 ARM 寄存器移动 | 5E | MRRC <copr>, <op1>, Rd, Rn, CRm | | 由协处理器定义 |
| 另两个 ARM 寄存器移动 | 6 | MRRC2 <copr>, <op1>, Rd, Rn, CRm | | 由协处理器定义 |
| 从 ARM 寄存器移到协处理器 | | MCR{2} <copr>, <op1>, Rd, CRn, CRm{, <op2>} | | 由协处理器定义 |
| 两个 ARM 寄存器移动 | 5E | MCCR <copr>, <op1>, Rd, Rn, CRm | | 由协处理器定义 |
| 另两个 ARM 寄存器移动 | 6 | MCCR2 <copr>, <op1>, Rd, Rn, CRm | | 由协处理器定义 |
| 加载和存储, 前变址 | | <op>{2} <copr>, CRd, [Rn, #+/-<offset8*4>] {!} | op LDC 或 STC。偏移量: 0 到 1020 范围内 4 的倍数。 | 由协处理器定义 |
| 加载和存储, 零偏移量 | | <op>{2} <copr>, CRd, [Rn] {, 8-bit copro. option} | op LDC 或 STC。 | 由协处理器定义 |
| 加载和存储, 后变址 | | <op>{2} <copr>, CRd, [Rn], #+/-<offset8*4> | op LDC 或 STC。偏移量: 0 到 1020 范围内 4 的倍数。 | 由协处理器定义 |

| 其他运算 | \$ | 汇编器 | 操作 | 说明 |
|--------|----|-----------------------------------|---|-----|
| 交换字 | | SWP Rd, Rm, [Rn] | temp := [Rn], [Rn] := Rm, Rd := temp。 | A、D |
| 交换字节 | | SWPB Rd, Rm, [Rn] | temp := ZeroExtend([Rn][7:0]), [Rn][7:0] := Rm[7:0], Rd := temp | A、D |
| 存储返回状态 | 6 | SRS{IA IB DA DB} SP{!}, #<p_mode> | [SPm] := LR, [SPm + 4] := CPSR | C、I |
| 从异常中返回 | 6 | RFE{IA IB DA DB} Rn{!} | PC := [Rn], CPSR := [Rn + 4] | C、I |
| 断点 | 5 | BKPT <imm16> | 预取中止或进入调试状态。指令中编码为 16 位的位域。 | C、N |
| 安全监控调用 | Z | SMC <imm4> | 安全监控调用异常。指令中编码为 4 位的位域。以前为 SMI。 | |
| 超级用户调用 | | SVC <imm24> | 超级用户调用异常。指令中编码为 24 位的位域。以前为 SWI。 | N |
| 无操作 | 6K | NOP | 无操作, 可能不花费任何时间。 | N、V |
| 提示 | | | | |
| 调试提示 | 7 | DBG | 向调试系统及其相关系统发送提示。 | |
| 数据内存屏障 | 7 | DMB | 确保内存访问的观察顺序。 | C |
| 数据同步屏障 | 7 | DSB | 确保内存访问完成。 | C |
| 指令同步屏障 | 7 | ISB | 刷新处理器管道并跳转预测逻辑。 | C |
| 设置事件 | 6K | SEV | 向多处理器系统发送事件信号。如果不执行, 则为 NOP。 | N |
| 等待事件 | 6K | WFE | 等待事件、IRQ、FIQ、不精确的中止或调试进入请求。如果不执行, 则为 NOP。 | N |
| 等待中断 | 6K | WFI | 等待 IRQ、FIQ、不精确的中止或调试进入请求。如果不执行, 则为 NOP。 | N |
| Yield | 6K | YIELD | 生成对其他线程的控制。如果不执行, 则为 NOP。 | N |

| 说明 | | | | |
|----|--|---|--|--|
| A | Thumb 状态下不可用。 | P | 在 Thumb 状态下, 此指令中的 Rn 可为 PC。 | |
| B | 在 Thumb 状态下可带有条件, 且无须在 IT 块内。 | Q | 如果发生饱和 (加法或减法) 或溢出 (乘法), 则设置 Q 标记。使用 MRS 和 MSR 读取和重置 Q 标记。 | |
| C | ARM 状态中不允许使用条件代码。 | R | 在 ARM 指令中, <sh> 范围为 1-32。 | |
| C2 | 各选格式 2 可用于 ARMv5 中。它可提供另一种各选运算。在 ARM 状态下, 各选格式不允许使用条件代码。 | S | S 修饰符在 Thumb-2 指令中不可用。 | |
| D | 已弃用。使用 LDREX 和 STREX 来代替。 | T | ARM 状态中不可用。 | |
| G | 根据各个运算的结果更新 CPSR 中的 4 个 GE 标记。 | U | 不允许在 IT 块中使用。不允许在 ARM 或 Thumb 状态下使用条件代码。 | |
| I | IA 是缺省值, 通常省略。 | V | 如果 NOP 指令不可用, 汇编器会插入适当的指令。 | |
| L | ARM <imm8m>。16 位 Thumb 0 到 1020 范围内 4 的倍数。32 位 Thumb 0-4095。 | | | |
| N | 在 Thumb-2 代码中, 此指令的某些格式或所有格式为 16 位 (窄) 指令。有关详细信息, 请参阅 <i>Thumb 16 位指令集 (UAL) 快速参考卡</i> 。 | | | |

| ARM 体系结构版本 | |
|-----------------------|---|
| <i>n</i> | ARM 体系结构版本 <i>n</i> 及更高版本 |
| <i>nT</i> , <i>nJ</i> | ARM 体系结构版本 <i>n</i> 及更高版本的 T 或 J 变体 |
| 5E | ARM v5E、6 版及更高版本 |
| T2 | ARM v6 及更高版本的所有 Thumb-2 版本 |
| 6K | 支持 ARM 指令的 ARMv6K 及更高版本, 支持 Thumb 的 ARMv7 |
| 7MP | 实现多重处理扩展的 ARMv7 体系结构 |
| Z | ARMv6 及更高版本的所有安全扩展版本 |
| RM | 仅限 ARMv7-R 和 ARMv7-M |
| XS | XScale 协处理器指令 |

灵活的操作数 2

| | |
|------------------------|---------------|
| 立即值 | #<imm8m> |
| 寄存器, 可选择移动常数个位 (请参阅下文) | Rm {, <opsh>} |
| 寄存器, 寄存器逻辑左移 | Rm, LSL Rs |
| 寄存器, 寄存器逻辑右移 | Rm, LSR Rs |
| 寄存器, 寄存器算术右移 | Rm, ASR Rs |
| 寄存器, 寄存器向右循环移 | Rm, ROR Rs |

寄存器, 可选择移动常数个位

| | | |
|-----------|------------------|-----------------|
| (不进行移位) | Rm | 与 Rm, LSL #0 相同 |
| 逻辑左移 | Rm, LSL #<shift> | 允许移动 0-31 位 |
| 逻辑右移 | Rm, LSR #<shift> | 允许移动 1-32 位 |
| 算术右移 | Rm, ASR #<shift> | 允许移动 1-32 位 |
| 向右循环移 | Rm, ROR #<shift> | 允许移动 1-31 位 |
| 带扩展的向右循环移 | Rm, RRX | |

| PSR 字段 (至少使用一个后缀) | | |
|-------------------|----------|------------|
| 后缀 | 含义 | |
| C | 控制字段掩码字节 | PSR[7:0] |
| F | 标记字段掩码字节 | PSR[31:24] |
| S | 状态字段掩码字节 | PSR[23:16] |
| X | 扩展字段掩码字节 | PSR[15:8] |

所有权声明

除非本所有权声明在下面另有说明, 否则带有® 或™ 标记的词语和徽标是 ARM Limited 在欧盟和其他国家/地区的注册商标或商标。此处提及的其他品牌和名称可能是其各自所有者的商标。

除非事先得到版权所有人的书面许可, 否则不得以任何形式修改或复制本文档包含的部分或全部信息以及产品说明。

本文档描述的产品还将不断发展和完善。ARM Limited 将如实提供本文档所述产品的所有特性及其使用方法。但是, 所有暗示或明示的担保, 包括但不限于对特定用途适用性或适用性的担保, 均不包括在内。

本参考卡仅旨在帮助读者使用产品。对由于使用本参考卡中的任何信息, 或由于本参考卡的信息错误、遗漏, 以及产品的错误使用所造成的任何损失, ARM Limited 概不负责。

| 条件字段 | | |
|---------|---------------|-----------------|
| 助记符 | 说明 | 说明 (VFP) |
| EQ | 等于 | 等于 |
| NE | 不等于 | 不等于或无序 |
| CS / HS | 进位设置/无符号大于或相同 | 大于或等于或无序 |
| CC / LO | 进位清零/无符号小于 | 小于 |
| MI | 求反 | 小于 |
| PL | 正数或零 | 大于或等于或无序 |
| VS | 溢出 | 无序 (至少一个非数字操作数) |
| VC | 无溢出 | 非无序的 |
| HI | 无符号大于 | 大于或无序 |
| LS | 无符号小于或相同 | 小于或等于 |
| GE | 有符号大于或等于 | 大于或等于 |
| LT | 有符号小于 | 小于或无序 |
| GT | 有符号大于 | 大于 |
| LE | 有符号小于或等于 | 小于或等于或无序 |
| AL | 始终 (通常省略) | 始终 (通常省略) |

所有 ARM 指令 (带有注释 C 或注释 U 的除外) 可在指令助记符后 (即, 本卡中显示的指令中的第一个空格前) 带有这些条件代码之一。此条件在指令中编码。

所有 Thumb-2 指令 (带有注释 U 的除外) 可在指令助记符后带有这些条件代码之一。此条件在前面的 IT 指令中进行编码 (条件跳转指令除外)。指令中的条件代码必须与前面 IT 指令中的条件代码相匹配。

对于不带 Thumb-2 的处理器, 唯一可带有条件代码的 Thumb 指令为 B <label>。

| 处理器模式 | |
|-------|----------|
| 16 | 用户 |
| 17 | FIQ 快速中断 |
| 18 | IRQ 中断 |
| 19 | 超级用户 |
| 23 | 中止 |
| 27 | 未定义 |
| 31 | 系统 |

| 并行指令的前缀 | |
|---------|--|
| S | 对 2^8 或 2^{16} 有符号算术求模, 设置 CPSR GE 位 |
| Q | 有符号饱和算法 |
| SH | 有符号算法, 将结果减半 |
| U | 对 2^8 或 2^{16} 无符号算术求模, 设置 CPSR GE 位 |
| UQ | 无符号饱和算法 |
| UH | 无符号算法, 将结果减半 |

文档编号

ARM QRC 0001M

变更记录

| 发行号 | 日期 | 变更 | 发行号 | 日期 | 变更 |
|-----|-------------|----------|-----|-------------|--------------|
| A | 1995 年 6 月 | 第一版 | B | 1996 年 9 月 | 第二版 |
| C | 1998 年 11 月 | 第三版 | D | 1999 年 10 月 | 第四版 |
| E | 2000 年 10 月 | 第五版 | F | 2001 年 9 月 | 第六版 |
| G | 2003 年 1 月 | 第七版 | H | 2003 年 10 月 | 第八版 |
| I | 2004 年 12 月 | 第九版 | J | 2005 年 5 月 | RVCT 2.2 SP1 |
| K | 2006 年 3 月 | RVCT 3.0 | L | 2007 年 3 月 | RVCT 3.1 |
| M | 2008 年 9 月 | RVCT 4.0 | | | |

附录 2、机器指令格式

LDR<c> <Rt>, [<Rn>, <Rm>]

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|----|---|---|----|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | Rm | | | Rn | | | Rt | | |

LDR<c>.W <Rt>, [<Rn>, <Rm>{, LSL #<imm2>}]

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|------|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Rn | | | | Rt | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | imm2 | Rm | | | |

LSL<c> <Rd>, <Rm>, #<imm5>

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|------|---|---|---|---|----|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | imm5 | | | | | Rm | | | Rd | | |

LDR<c> <Rt>, [<Rn>{, #<imm5>}]

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|------|---|---|---|---|----|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | imm5 | | | | | Rn | | | Rt | | |

LDR<c> <Rt>, [SP{, #<imm8>}]

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | Rt | | | imm8 | | | | | | | |

imm32 = ZeroExtend(imm8:'00', 32);

LDR<c>.W <Rt>, [<Rn>{, #<imm12>}]

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Rn | | | | Rt | | | | imm12 | | | | | | | | | | | |

imm32 = ZeroExtend(imm12, 32)

LDR<c> <Rt>, <label>

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | Rt | | | imm8 | | | | | | | |

imm32 = ZeroExtend(imm8:'00', 32)

LDR<c>.W <Rt>, <label>

LDR<c>.W <Rt>, [PC, #-0]

LDR<c><q> <Rt>, [PC, #+/-<imm>]

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|-------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | U | 1 | 0 | 1 | 1 | 1 | 1 | 1 | Rt | | | | imm12 | | | | | | | | | | | |

imm32 = ZeroExtend(imm12, 32)

B<c> <label>

| | | | | | | | | | | | | | | | |
|----|----|----|----|------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 1 | cond | | | | imm8 | | | | | | | |

imm32 = SignExtend(imm8:'0', 32);

B<c> <label>

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|-------|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | imm11 | | | | | | | | | | |

imm32 = SignExtend(imm11:'0', 32);

B<c>.W <label>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|------|---|---|------|---|---|---|---|---|---|----|----|----|----|-------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | S | cond | | | imm6 | | | | | | 1 | 0 | J1 | 0 | J2 | imm11 | | | | | | | | | | | |

imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);

B<c>.W <label>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|-------|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | S | imm10 | | | | | | | | | | 1 | 0 | J1 | 1 | J2 | imm11 | | | | | | | | | | |

imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);

BL<c> <label>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|-------|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | S | imm10 | | | | | | | | | | 1 | 1 | J1 | 1 | J2 | imm11 | | | | | | | | | | |

I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);

MOV{S}<c><q> <Rd>, #<const>

0-255

MOVW<c><q> <Rd>, #<const>

0-65535

MOVS <Rd>, #<imm8>

MOV<c> <Rd>, #<imm8>

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|------|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | Rd | | | | | imm8 | | | | | |

imm32 = ZeroExtend(imm8, 32); carry = APSR.C;

MOV{S}<c>.W <Rd>, #<const>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 0 | i | 0 | 0 | 0 | 1 | 0 | S | 1 | 1 | 1 | 1 | 0 | imm3 | | | Rd | | | | | imm8 | | | | | | | |

(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);

MOVW<c> <Rd>, #<imm16>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|------|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 0 | i | 1 | 0 | 0 | 1 | 0 | 0 | imm4 | | | | 0 | imm3 | | | Rd | | | | | imm8 | | | | | | | |

imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);

PUSH<c> <registers>

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | M | register_list | | | | | | | |

M - LR P - PC

PUSH<c>.W <registers>

<registers> contains more than one register

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|-----|----|-----|---------------|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | (0) | M | (0) | register_list | | | | | | | | | | | | |

PUSH<c>.W <registers>

<registers> contains one register, <Rt>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | Rt | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |

POP<c> <registers>

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | P | register_list | | | | | | | |

POP<c>.W <registers>

<registers> contains more than one register

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|-----|---------------|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | P | M | (0) | register_list | | | | | | | | | | | | |

registers = P:M:'0':register_list;

POP<c>.W <registers>

<registers> contains one register, <Rt>

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | Rt | | | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

参考资料

- S. L. Harris, D. M. Harris, 数字设计与计算机体系结构, 机械工业出版社, 2019.7
- 文全刚, 郝志刚主编, 汇编语言程序设计-基于ARM (第三版), 北航出版社, 2016.4
- J. Yiu, ARM Cortex-M3与Cortex-M4权威指南, 清华大学出版社, 2015.10
- ARM®v7-M Architecture Reference Manual, ARM, 2018
- ARM® and Thumb®-2 Instruction Set Quick Reference Card

总结

- 概述
- 寻址方式
- 数据处理指令
- 转移指令
- 函数调用
- 汇编语言
- 汇编程序举例
- 统一汇编语言
- 附录 1、ARM 和 Thumb-2 指令集
- 附录 2、机器指令格式