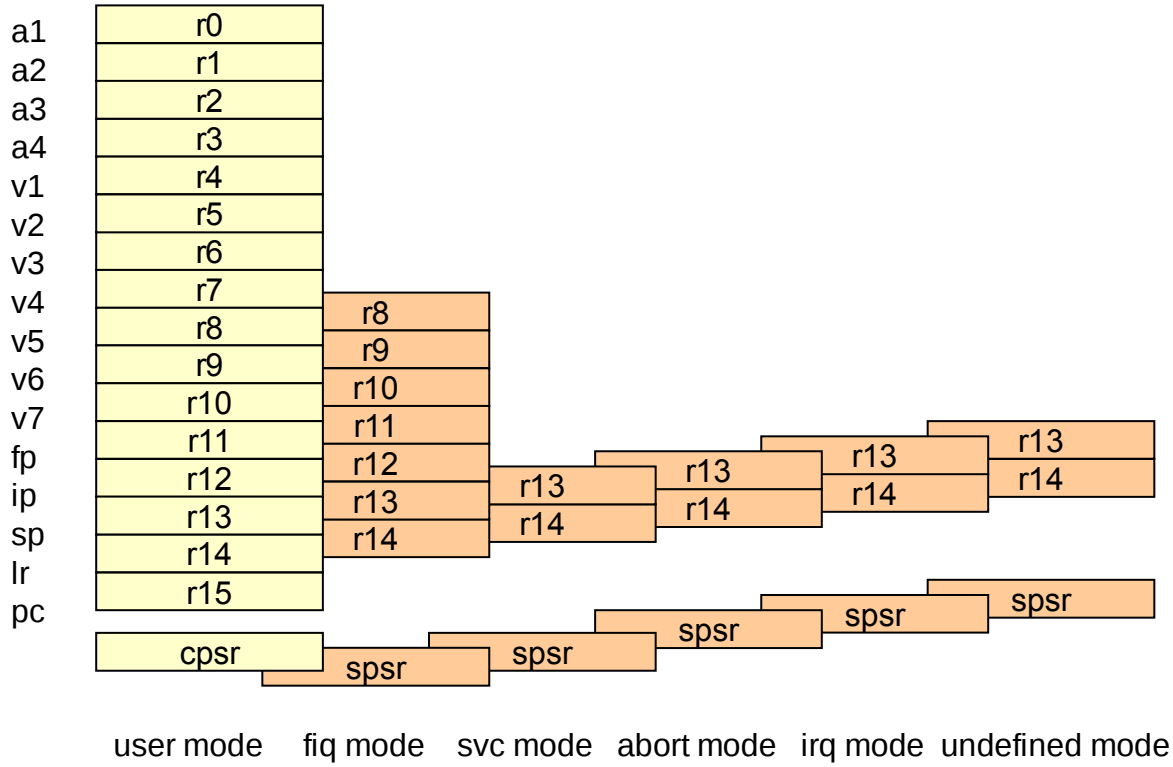
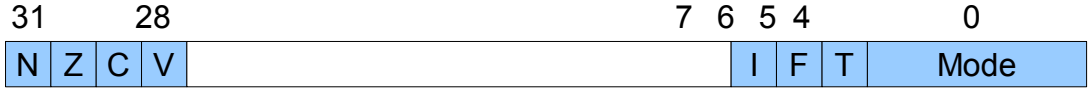
Introdução

A arquitectura ARM tem sido revista e modificada ao longo do tempo, dando origem a variantes como: v1, v2, v2a, v3, v4, v4T, v4TE.
As realizações do processador são também diversas, visam o bom desempenho e são designadas por *processor cores*. Exemplos: ARM7TDMI, ARM8, ARM9, ARM9TDMI.
Associados ao processador surgem funções como a gestão da memória, caches, inferfaces para memórias, etc. À combinação de *processor core* com estes elementos dá-se a designação de *cpu core*. Exemplos: ARM710T, ARM920T, StrongARM.
Por sua vez os fabricantes combinam o *cpu core* com periféricos de processador dando origem aos microcontroladores. Exemplos; AT91M55800, AT91RM9200, Xscale.
As definições que se seguem são comuns a todas as variantes da arquitectura.
Na disciplina de SE1 é usado o microcontrolador LPC2106 da Philips, com *processor core* ARM7TDMI-S a que corresponde a variante de arquitectura versão 4T.

Registos



ARM flags



N – a operação anterior produziu um resultado negativo.

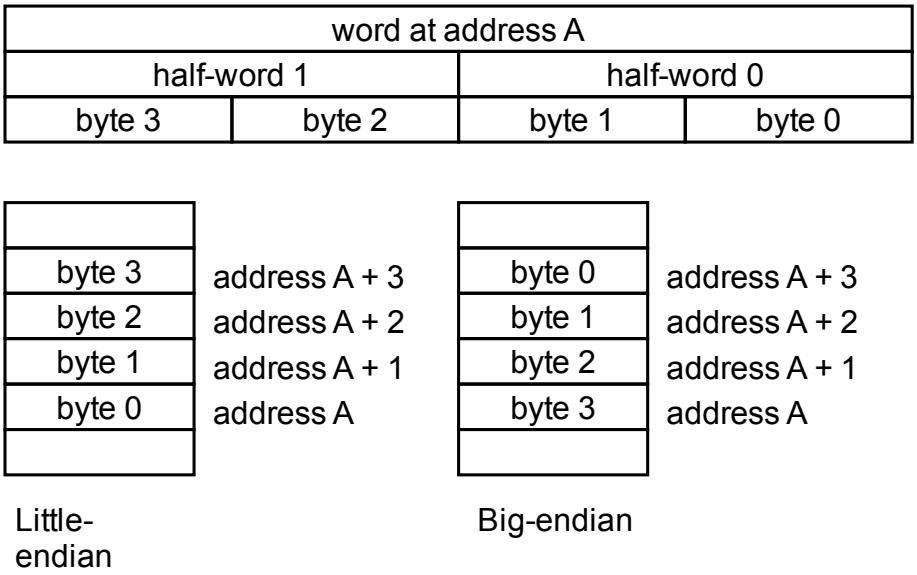
Z – a operação anterior produziu um resultado igual a zero.
C – a operação anterior gerou arrasto.
V – a operação anterior produziu um resultado inválido.

I – quando a 1 desactiva a entrada de interrupções IRQ.
F – quando a 1 desactiva a entrada de interrupções FIQ.
T – estado do processador ARM ou Thumb.

10000 – User
10001 – FIQ
10010 – IRQ
10011 – Supervisor
10111 – Abort
11011 – Undefined
11111 – System

Organização da memória

Uma posição de memória é formada por 8 bits.
O processador endereça a 2^32 posições de memória, com endereços entre `0x00000000` e `0xffffffff`.
A arquitectura define a existência de valores numéricos formados por 1 *byte*, 2 *bytes* (*half-word*) e 4 *bytes* (*word*).
Nas transferências de dados entre os registos do processador e a memória podem ser transferidos *bytes, half-words* e *words*, envolvendo assim várias posições de memória simultaneamente.
Por definição da arquitectura, as instruções de transferência de *half-word* e *word* só operam em endereços pares e múltiplos de quatro, respectivamente.
O armazenamento em memória de valores que envolvam múltiplos bytes usa o critério *little-endian*. Neste critério a parte mais baixa do valor é armazenada num endereço mais baixo.



Instruções

Processamento de dados

Operações aritméticas ou lógicas e cópias; os operandos são valores em registo ou valores imediatos.
AND, EOR, SUB RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN
MUL, MLA, UMULL, UMLAL, SMULL, SMLAL

Transferência de dados

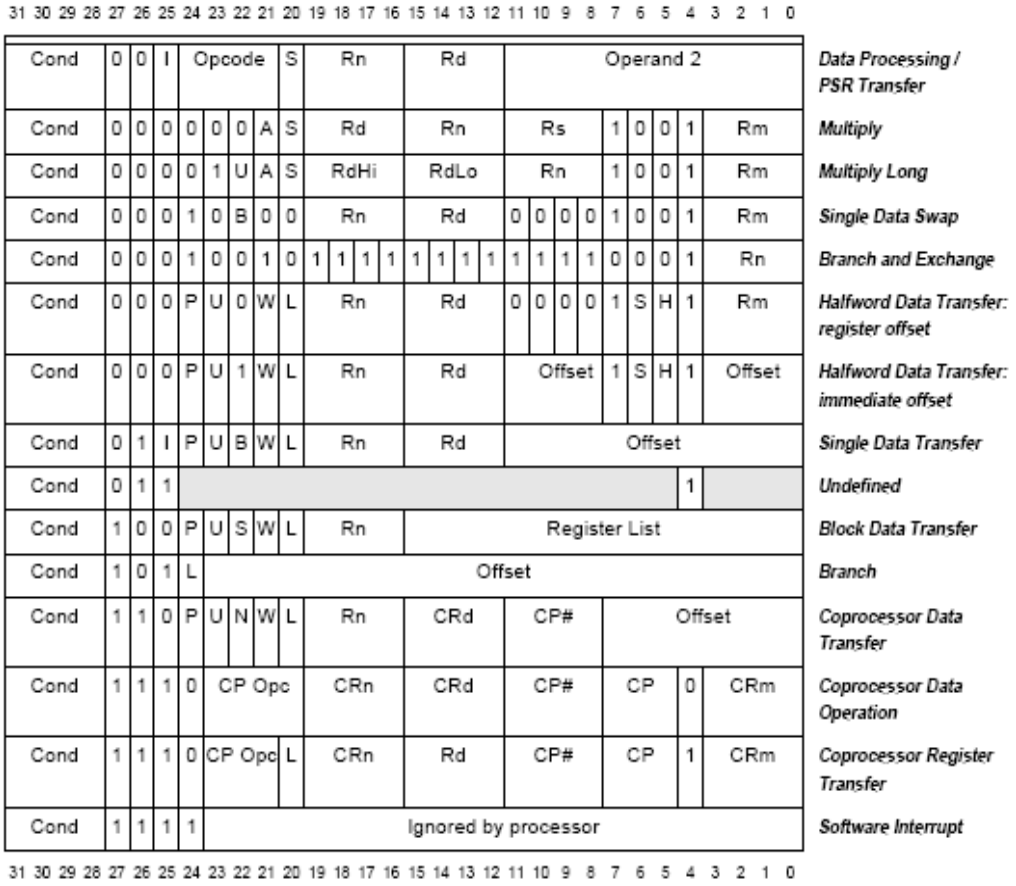
Cópia de dados da memória para registos e de registos para a memória.
Possibilidade de definir numa instrução a transferência de vários registos.
LDR, STR, LDM, STM, SWAP

Controlo de fluxo de execução

Saltos (branch). A definição do endereço para onde se salta é relativa ao PC.
Saltos com ligação (branch with link). O endereço seguinte é guardado no registo r14 (LR)
B, BL, SWI

Codificação das instruções

Todas as instruções são codificadas a 32 bits



Execução condicional

Todas as instruções são condicionais.
As condições baseiam-se nos valores das *flags.*

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Instruções de processamento de dados

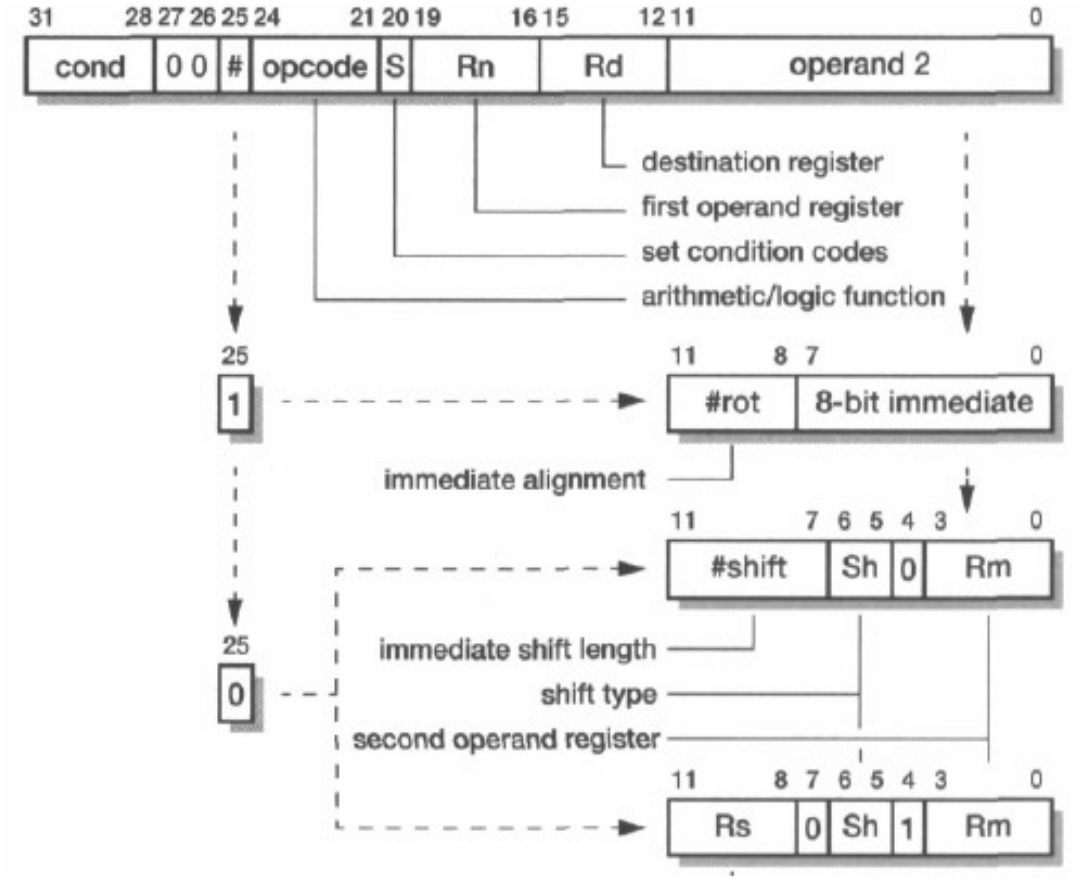
AND{cond}{S} Rd, Rn, Operand2	Logical bit-wise AND	Rd = Rn AND op2
EOR{cond}{S} Rd, Rn, Operand2	Logical bit-wise exclusive OR	Rd = Rn EOR op2
SUB{cond}{S} Rd, Rn, Operand2	Subtract	Rd = Rn - op2
RSB{cond}{S} Rd, Rn, Operand2	Reverse subtract	Rd = op2 – Rn
ADD{cond}{S} Rd, Rn, Operand2	Add	Rd = Rn + op2
ADC{cond}{S} Rd, Rn, Operand2	Add with carry	Rd = Rn + op2 + C
SBC{cond}{S} Rd, Rn, Operand2	Subtract with carry	Rd = Rn - op2 + C-1
RSC{cond}{S} Rd, Rn, Operand2	Reverse subtract with carry	Rd = op2 - Rn + C-1
TST{cond} Rn, Operand 2	Test	Scc on Rn AND op2
TEQ{cond} Rn, Operand 2	Test equivalence	Scc on Rn EOR op2
CMP{cond} Rn, Operand 2	Compare	Scc on Rn - op2
CMN{cond} Rn, Operand 2	Compare negated	Scc on Rn + op2
ORR{cond}{S} Rd, Rn, Operand2	Logical bit-wise OR	Scc on Rn OR op2
MOV{cond}{S} Rd, Operand 2	Move	Rd = op2
BIC{cond}{S} Rd, Rn, Operand2	Bit clear	Rd = Rn AND NOT op2
MVN{cond}{S} Rd, Operand 2	Move negated	Rd 0 NOT op2

S – afectar as flags

Operand 2 – #<immed_8r> | Rm {, <shift>}

<immed8_r> – valor imediato a 32 bits codificável (0 -> 255) x 2^{2n} com 0<=n<=12

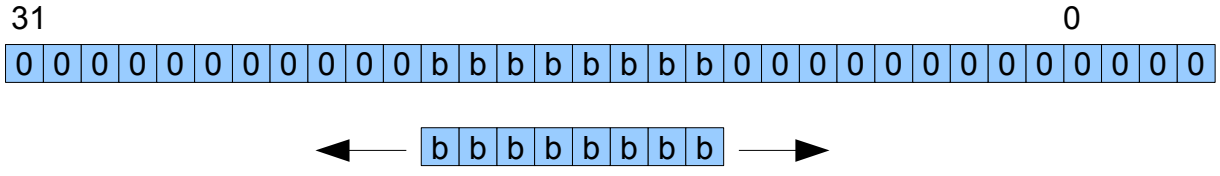
<shift> – [ASL | LSL | LSR | ASR | ROR] < <register> | #immed_5> > | RRX



Operando imediato

Designa-se operando imediato ao operando cujo valor é incluído no código das instruções. Na arquitectura ARM, como todas as instruções são codificadas a 32 bits, não é possível incluir numa instrução um operando imediato de 32 bits.
Nesta arquitectura é utilizado um esquema de codificação de operandos imediatos que usa 12 bits.

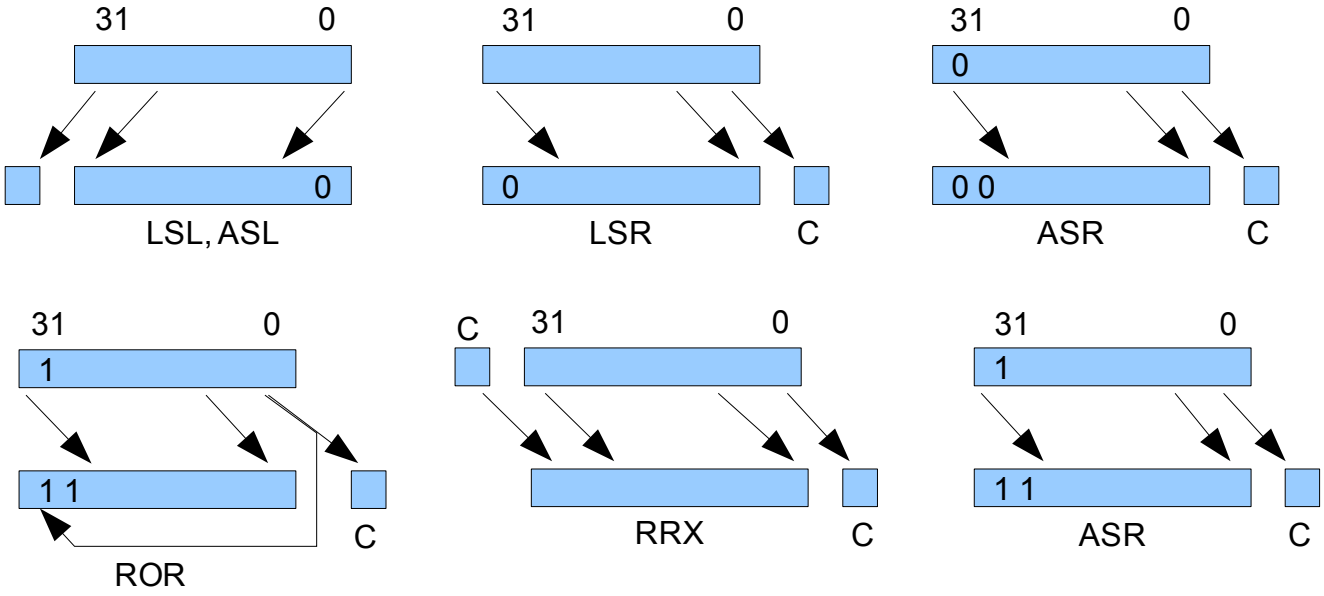
Os valores codificáveis são dados pela expressão m x 2^{2n} sendo m um valor entre 0 e 255 e n um valor entre 0 e 12.
Uma descrição prática: só se podem codificar valores em que os bits a 1 não estejam afastados mais de 7 posições. Os bits extremos ocupam posições pares.



Exemplo1: **`mov r0,#0xF500`** será codificado da seguinte forma:

Exemplo 2: **`mov r0,#0x101`** não é possível codificar.

Deslocamentos



Nos deslocamentos aritméticos há que preservar o sinal do operando.
Para fazer uma rotação para a esquerda, **rol**, pode executar-se **ror** um número complementar de vezes.
A *flag carry* será afectada apenas se a instrução tiver o sufixo S.

Instruções de processamento de dados – uso de r15

r15 pode ser utilizado como operando, excepto quando é usado um deslocamento cujo valor é especificado por registo.
Quando r15 é usado como operando o seu valor é o endereço da instrução corrente mais 8. O deslocamento de 8 evidencia o funcionamento em *pipeline* do processador.
r15 pode ser usado como destino o que transforma uma operação de transferência de dados num salto.
Quando r15 é usado como destino o sufixo S controla se o registo CPSR deve ser afectado por SPSR. Isto é a forma de retornar duma excepção – restaurando o r15 e CPSR simultaneamente.

Instruções de processamento de dados - exemplos

Copia de valores - afecta r0 com o valor de r1.

```
mov  r0, r1
```

Adição de valores - afecta r3 com r4 + r5.

```
add    r3, r4, r5
```

Subtracção de valores - afecta r0 com r8 – r12

```
sub    r0, r8, r12
```

Subtracção de valores com arrastamento – afecta r0 com r8 – r12 - borrow

```
sbc    r0, r8, r12
```

Simétrico de um valor inteiro – nega o valor de r0

```
mvn    r0, r0
```

Colocar bits a zero - coloca o bit 5 de r3 a zero.

```
bic    r3, r3, #32
```

Colocar bits a um - Coloca o bit 4 de r4 a um.

```
orr    r4, r4, #16
```

Comparação de valores numéricos - Afecta as flags com o resultado de r3 – r2

```
cmp    r3, r2
```

Comparação de igualdade de valores - executa r3 ^ (r4 >> r5) e afecta as flags

```
teq    r3, r4, lsr r5
```

Testar o valor de um bit - executa r3 & (r4 >> r5) e afecta as flags

```
tst    r3, r4, lsr r5
```

Multiplicação por constante - multiplica r0 por 10.

```
mov    r0, r0, lsl #1
add    r0, r0, lsl #2
```

Soma de valores representados a 64 bits

Primeiro operando em r1:r0, segundo operando em r3:r2, resultado em r5:r4.

```
adds r4, r2, r0
adc  r5, r3, r1
```

Subtracção de valores representados a 64 bits

Primeiro operando em r1:r0, segundo operando em r3:r2, resultado em r5:r4.

```
subs r4, r2, r0
sbb  r5, r3, r1
```

Deslocamento de valores a 64 bits para a direita

Valor a 64 bits depositado nos registos r1:r0.

Deslocar uma posição.

```
    movs r1, r1, lsr #1
    mov  r0, r0, rrx
```

Deslocar N (constante) posições.

```
    mov  r0, r0, lsr #4
    add  r0, r0, r1, lsl #(32 - 4)
    mov  r1, r1, lsr #4
```

Deslocar N (variável em r2) posições.

```
    mov  r0, r0, lsr r2
    rsb  r3, r2, #32
    add  r0, r0, r1, lsl r3
    mov  r1, r1, lsr r2
```

Deslocamento de valores a 64 bits para a esquerda

Valor a 64 bits depositado nos registos r1:r0.

Deslocar uma posição.

```
movs r0, r0, lsl #1
adc  r1, r1, r1
```

Deslocar N (constante) posições.

```
mov  r1, r1, lsl #4
add  r1, r1, r0, lsr #(32 - 4)
mov  r0, r0, lsl #4
```

Deslocar N (constante) posições.

```
mov  r1, r1, lsl r2
rsb  r3, r2, #32
add  r1, r1, r0, lsr r3
mov  r0, r0, lsl #4
```

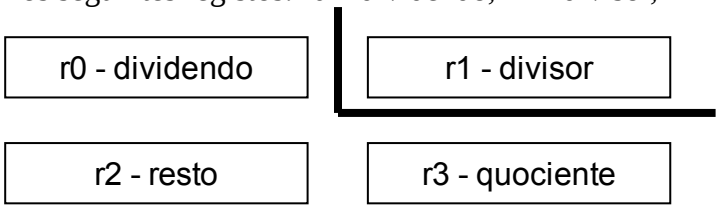
Operação de divisão – algoritmo “subtracções sucessivas”.

Vai-se subtraindo o divisor do dividendo até já não se poder subtrair mais. O número de subtracções é o resultado.
No troço de programa que a seguir se apresenta assume-se que os operandos e o resultado se encontram nos seguintes registos: r0 – dividendo, r1 – divisor, r0 – resto, r2– quociente.

```
    mov  r2, #0         /* quociente */
1:
    cmp  r0, r1         /* termina quando dividendo < divisor */
    blo  2f
    sub  r0, r1         /* dividendo – divisor */
    add  r2, r2, #1     /* quociente += 1 */
    b    1b
2:
```

Operação de divisão – algoritmo “sub-and-shift”.

No troço de programa que a seguir se apresenta assume-se que os operandos e o resultado se encontram nos seguintes registos: r0 – dividendo, r1 – divisor, r2 – resto, r3 – quociente.




```
    mov  r2, #0
    mov  r3, #0
    mov  r4, #32
1:
    movs r0, r0, lsl #1     desloca o conjunto r2:r0 - resto e dividendo,
    adc  r2, r2, r2         um bit para esquerda, inserindo na posição de menor
                            peso do resto o bit de maior peso do dividendo

    cmp  r2, r1             se o resto for maior que o divisor
```

```
    subcs r2, r2, r1        subtrai o dividor do resto e
    adc  r3, r3, r3         insere 1 no bit de menor peso do quociente
                            ao mesmo tempo que o desloca para a direita
    subs r4, r4, #1
    bne  1b                 repete para os 32 bits
```

Operação de multiplicação – algoritmo “adições sucessivas”.

Assume-se que os operandos e o resultado se encontram nos seguintes registos: r0 – multiplicando, r1 – multiplicador, r3:r2 – resultado.

```
    mov  r2, #0
    mov  r3, #0
    cmp  r1, #0
    jeq  2f
1:
    adds r2, r2, r0
    adc  r3, r3, #0
    subs r1, r1, #1
    bne  1b
2:
```

Operação de multiplicação – algoritmo “add-and-shift”.

Para operandos a 32 bits o algoritmo processa-se em 32 passos. Em cada passo o multiplicando é multiplicado por um bit do multiplicador e somado ao registo de resultado parcial.

Assume-se que os operandos e o resultado se encontram nos seguintes registos: r1:r0 – multiplicando, r2 – multiplicador, r4:r3 – resultado.

```
    mov       r3, #0            r4:r3 – resultado
    mov       r4, #0
    ldr       r2, [r2]
    movs      r2, r2            multiplicador igual a 0 ?
    beq       2f
1:
    tst       r2, #1            bit de menor peso de multiplicador?
    addnes    r3, r3, r0        resultado += multiplicando
    adcne     r4, r4, r1
    adds      r0, r0, r0        mutiplicando = multiplicando << 1
    adc       r1, r1, r1

    movs      r2, r2, lsr #1    multiplicador = multiplicador >> 1
    bne       1b
2:
```

Conversão de binário para caracteres.

Um valor em binário, num registo do processador ou na memória, pode ser visualizado em caracteres numéricos em base decimal, octal, binária ou hexadecimal.
O troço de programa que se segue determina os caracteres hexadecimais que representam um byte em r0. Os caracteres resultantes irão ser colocador nos registos r4 e r5 pela ordem do respectivo peso.

```
    mov       r12, #0xf
```

```
and     r4, r12, r0             nible de peso 0
cmp     r4, #10
addlo   r4, r4, #'0'            menor que 10 soma '0'
addhs   r4, r4, #('A' - 10)     maior ou igual a 10 soma 'A' - 10

and     r5, r12, r0, lsr #4     nible de peso 4
cmp     r5, #10
addlo   r5, r5, #'0'
addhs   r5, r5, #('A' - 10)
```

Instruções de transferência de dados

Transferência de **word** e **unsigned byte**

LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}	**Rd <-> [Rn + <offset>];** **eventualmente Rn = Rn + <offset>**
LDR|STR{<cond>}{B} Rd, [Rn]{, <offset>}	**Rd <-> [Rn];** **Rn = Rn + <offset>**

<offset> - #{+ | -} <immed_12> | {+ | -} Rm {, <shift> }
<shift> - [ASL | LSL | LSR | ASR | ROR] #<immed_5> | RRX
! - actualiza o registo index
B - unsigned byte

Transferência de **half-word** e **signed byte**

LDR|STR{cond}H|SH|SB Rd, [Rn, <offset>]{!}
LDR|STR{cond}H|SH|SB Rd, [Rn]{, <offset>}

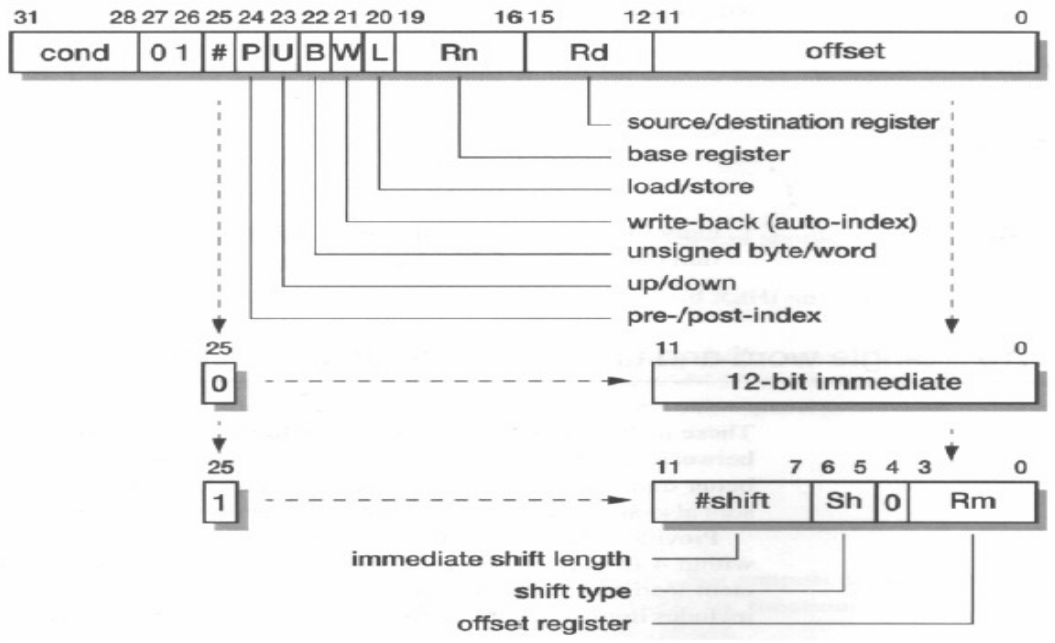
SB – signed byte
H – unsigned half-word
SH – signed half-word
<offset> – {+ | -} Rm | #{+ | -} <immed_8>

Forma pré-indexada: [Rn, <offset>]

O endereço da memória usado na transferência é Rn + offset. Se o sinal '!' estiver presente, Rn é actualizado com o endereço da memória depois deste ser usado na transferência.

Forma pós-indexada: [Rn], {<offset>} address = Rn; Rn = Rn + offset

O endereço da memória usado na transferência é Rn. Depois da transferência, o valor offset é adicionado a Rn.



Carregamento de constantes em registos

Carregar um valor a 32 bits num registo

Depositar esse valor em memória com a directiva **.word** na mesma secção do código (normalmente **.text**) e com a instrução **ldr** transferir da memória para o registo, usando endereçamento relativo ao **pc**.

```
address_of_value:
     .word     0x12345678              02000000
     ...
     ldr  r0, address_of_value         02000020
     ...
```

Se o endereço de address_of_value for 0x2000000 e o endereço da instrução **ldr** for 0x2000020, a instrução **ldr** será codificada da seguinte forma:

```
    ldr    r0, [pc, #-0x28]
```

Carregar o endereço de uma label num registo

Se a *label* pertencer a uma secção diferente da do código usa-se o método anterior. Se a *label* pertencer à mesma secção, adicionar (ou subtrair) do PC o deslocamento da posição actual até à *label*.

```
     .text
table:
    .byte    1, 2, 3, 4, 5, 6
     ...
     sub    r0, pc, #8 + . - table
     ...
```

Pseudo-instruções

nop

Instrução que não faz nada. É substituída por **mov r0, r0**.

ldr <register>, = <constant>

Carregar uma constante num registo. Se a constante for codificável no código da instrução usa mov ou mvn. Senão, a constante é colocada em memória e carregada no registo com uma instrução ldr com endereçamento relativo ao PC.

adr <register> <label>

Carregar o endereço definido por *label* num registo. Usa a instrução add ou sub para colocar no registo o valor de PC somado ou subtraído da distância até à label. A label deve estar no mesmo módulo e secção de adr.

adrl <register> <label>

O mesmo que a anterior mas pode usar duas instruções add ou sub, aumentando o alcance para o dobro. Se a segunda instrução não for necessária insere um nop ocupando sempre 2 words na codificação das instruções. A label deve estar no mesmo módulo e secção de adrl.

Instruções de transferência de dados - exemplos

Procurar o maior elemento de uma tabela de inteiros

Percorre um array de valores inteiros, reresentados a 32 bits, e determina qual é o maior valor.

```
     .equ SIZE_BLOCK, 40

     .data
table:
     .space    SIZE_BLOCK, 1      bloco de bytes com a dimensão SIZE_BLOCK
bigger:
     .word

     .text
address_table:     .word     table          auxiliares para carregamento
address_size:      .word     SIZE_BLOCK     de valores a 32 bits
address_bigger:    .word     big

     .global  _start
_start:
     ldr  r0, address_table       carrega endereço da tabela
     ldr  r1, address_size        carraga dimensão da tabela
     mov  r2, #0                  r2 é usado como variável auxiliar
     cmp  r1, #0                  verifica se está no fim
     beq  end
     ldr  r2, [r0], #4            carrega a primeira posição
     cmp  r1, #1                  verifica se é a única
     beq  end
loop:
     ldr  r3, [r0], #4            carrega o valor seguinte
     cmp  r2, r3                  compara se é maior que o actual em r2
```

```
     movlo     r2, r3
     subs r1, r1, #4              decrementa a dimensão restante
     bne  loop
end:
     ldr  r0, address_bigger
     str  r2, [r0]
```

Copiar um bloco de dados

Este algoritmo é bastante ineficiente porque transfere um *byte* de cada vez. Pode ser melhorado se transferir palavra a palavra e se usar instruções de transferência múltipla.

```
     .equ DIM, 40
     .data
block1:   .space    DIM, 1
block2:   .space    DIM, 2

     .text
address_block1:    .word    block1
address_block2:    .word    block1
address_SIZE: .word    DIM

     ldr  r0, address_block1
     ldr  r1, address_block2
     ldr  r2, address_SIZE
     cmp  r2, #0
     beq  end
loop:
     ldrb r3, [r1], #1
     strb r3, [r0], #1
     subs r2, r2, #1
     bne  loop
end:
```

Exemplos de codificação de expressões da linguagem C envolvendo ponteiros.

`char * cp;`	`r1`
`int i;`	`r2`
`char c;`	`r4`
`c = *cp;`	`ldrb  r4, [r1]`
`c = *cp++;`	`ldrb  r4, [r1], #1`
`c = *++cp;`	`ldrb  r4, [r1, #4]!`
`c = cp[i];`	`ldrb  r4, [r1, r2]`
`c = cp[i++];`	`ldrb  r2, [r0, r2]` `add  r3, r3, #1`

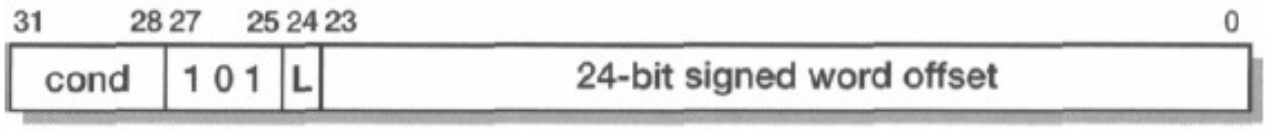
Instruções de controlo de fluxo

branch e *branch with link*

```
b{l}{<cond>}       <address>
```

O endereço final (definido por uma *label*) não é codificado em absoluto na instrução, este é obtido pela soma de **pc** com um deslocamento, este sim, codificado na instrução.
O deslocamento é calculado pelo compilador e corresponde à distância da instrução corrente até à label, medida em words.
O deslocamento é codificado a 24 bit com sinal, proporcionando um alcance de +/- 32 Mbyte.
Na instrução **bl** o endereço da instrução seguinte é guardado em r14. Este mecanismo serve para retornar de funções. r14 é também designado por **lr** (link register).
Pode ser usada a instrução de processamento de dados na forma mov r15, ... para executar saltos.
Saltos condicionais – todas as instruções são condicionais.

Codificacão



Instruções de controlo de fluxo – exemplo

```
...                                    ...
02000100    mov  r0, #23          xx: 02000200    add   r0, r0, r1
02000104    add  r1, #25              ...
02000108    bl   xx                   02000214    mov   r15, r14
0200010c    mov  r2, r0
...
```

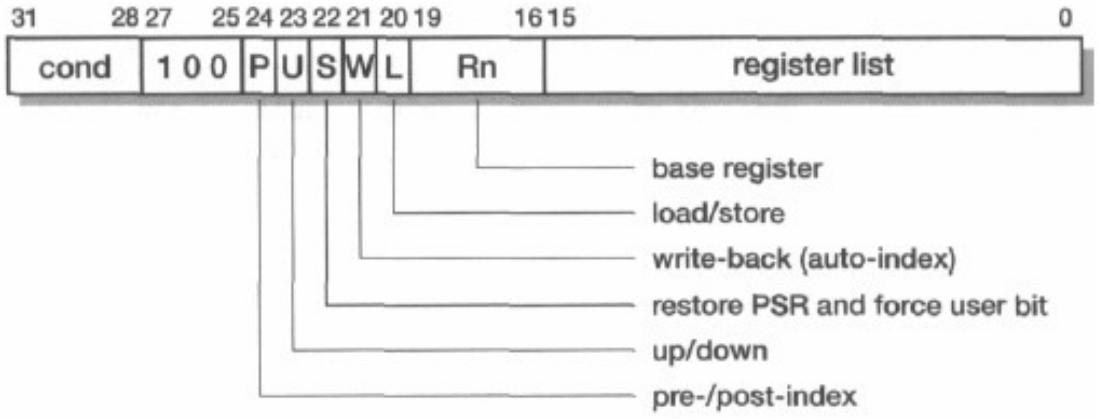
O offset é calculado pelo compilador (valor 0xf0).
A instrução de salto afecta **pc** com **pc** + offset (0x02000110 + 0xf0 = 0x02000200);
O registo **lr** é afectado com 0x200010C.

Instruções de transferência múltipla

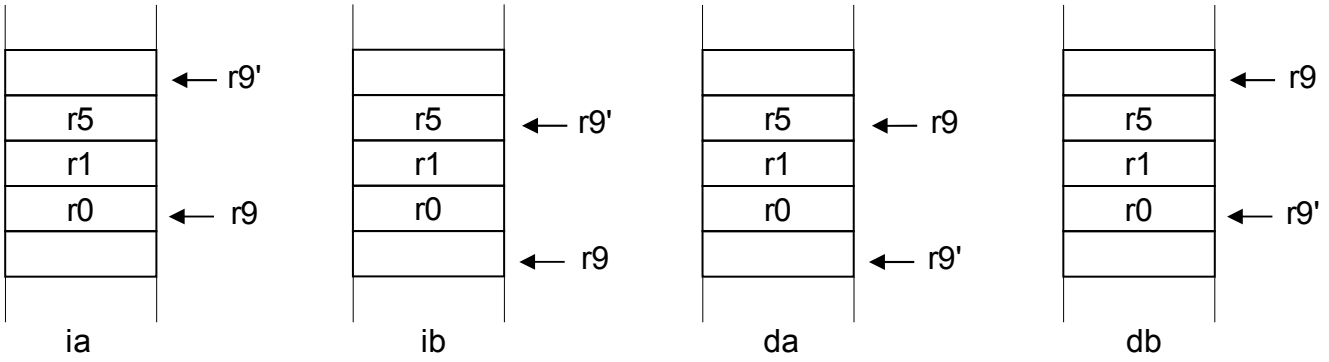
LDM|STM{cond}<add mode> Rn{!}, <registers>
LDM{cond}<add mode> Rn{!}, <registers + PC>^
LDM|STM{cond}<add mode> Rn, <registers – PC>^

A indicação dos registos pode ser feita separando por virgulas ex: {r0, r3, pc} e por ordem ascendente ou por gamas de registos ex: {r2–r9}
Em **ldm**, se for indicado o sinal ^ e a lista de registos incluir **pc**, **cpsr** é actualizado com **spsr**.
Se for indicado o sinal ^ e o pc não fizer parte da lista, os registos utilizados são os de modo utilizador.
Se o sinal ! for indicado Rn é actualizado com o novo endereço.
<add_mode> indica se Rn é incrementado ou decrementado, antes ou depois da transferência (ia, ib, da, db).

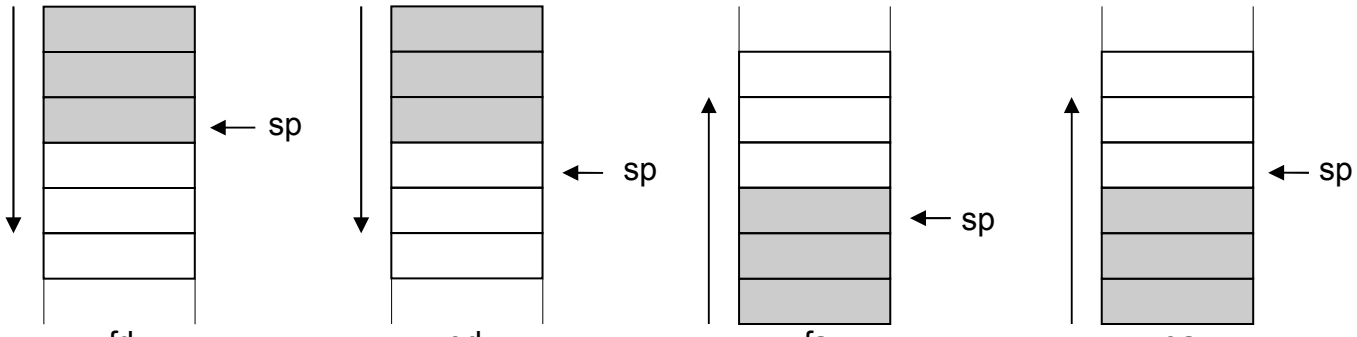
Codificação



Apresentam-se os efeitos da execução e ldm|stm<add_mode> r9!, {r0, r1, r5} considerando os diferentes sufixos add_mode. r9 representao valor deste registo antes da execuça e r9'representa o seu valor depois da execução.



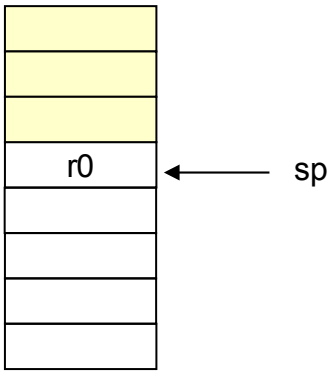
Perspectiva de stack



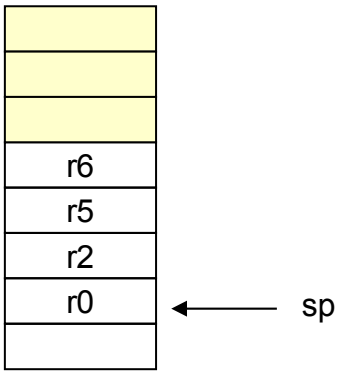
push	**stmdb**	**stmfd**	**stmda**	**stmed**	**stmib**	**stmfa**	**stmia**	**stmea**
pop	**ldmia**	**ldmfd**	**ldmib**	**ldmed**	**ldmda**	**stmfa**	**ldmdb**	**ldmea**

Full Descending Stack

```
push  r0  -> stm    r0, [sp, #-4]!
pop   r0  -> ldm    r0, [sp], #4
```




```
push  r0, r2, r5, r6   ->   stmdb  sp!, {r0, r2, r5, r6}
pop   r0, r2, r5, r6   ->   ldmia  sp!, {r0, r2, r5, r6}
```



Exemplos

Copiar um bloco de dados (optimizado)

A função memmove pertence à biblioteca normalizada da linguagem C. Copia os dados da zona de memória definida pelo ponteiro **src** e dimensão **size** para a zona de memória definida pelo ponteiro **dst** e dimensão **size**. Estas zonas de memória podem ter sobreposição.

```
void * memmove(void * dst, void * src, size_t size);
```

```
     r0                r0       r1       r2

     .text
     .global   memmove
memmove:
     cmp       r2, #0
     moveq     pc, lr         Se a dimensão for zero retorna já

     cmp       r0, r1
     moveq     pc, lr         Se os endereços forem iguais retorna já

     blo       5f

     ------------ O bloco de destino tem endereço mais alto -----------
     a transferência vai ser processada do fim para o principio

     add       r0, r0, r2     Colocar ponteiros (r0 e r1) no fim dos blocos
     add       r1, r1, r2
1:
     tst       r0, #3             Ajustar num endereço múltiplo de 4
     beq       2f
     ldrb      r3, [r1, #-1]!
     strb      r3, [r0, #-1]!
     subs      r2, r2, #1
     moveq     pc, lr             Parar se atingir o fim do bloco
     b         1b
2:
     tst       r1, #3             Ambos os endereço são multiplos de 4 ?
     bne       3f
3:
     cmp       r2, #4 * 10        Transferir blocos de 40 bytes
     blo       4f
     ldmdb     r1!, {r3 - r10, r12, r14}
     stmdb     r0!, {r3 - r10, r12, r14}
     subs      r2, r2, #4 * 10
     bne       3b
     moveq     pc, lr
4:
     ldrb      r3, [r1, #-1]!     Transferir o restante byte a byte
     strb      r3, [r0, #-1]!
     subs      r2, r2, #1
     bne       4b
     mov       pc, lr

     ------------ O bloco de destino tem endereço mais baixo -----------
     a transferência vai ser processada do principio para o fim
5:
     tst       r0, #3             Ajustar num endereço múltiplo de 4
     beq       6f
     ldrb      r3, [r1], #1
     strb      r3, [r0], #1
     subs      r2, r2, #1
     beq       10f                Retornar se atingir o fim do bloco
     b         5b
6:
     tst       r1, #3             Ambos os endereço são multiplos de 4 ?
     bne       8f
```

```
7:
    cmp     r2, #4 * 10       Transferir blocos de 40 bytes
    blo     8f
    ldmia   r1!, {r3 - r10, r12, r14}
    stmia   r0!, {r3 - r10, r12, r14}
    subs    r2, r2, #4 * 10
    bne     7b
    moveq   pc, lr
8:
    ldrb    r3, [r1], #1      Transferir o restante
    strb    r3, [r0], #1
    subs    r2, r2, #1
    bne     8b
    mov     pc, lr
```

Instruções de multiplicação

mul {<cond>} {S} Rd, Rm, Rs	**Resultado a 32 bits.**	**Rd = (Rm * RS)**
mla {<cond>} {S} Rd, Rm, Rs, Rn	**Resultado a 32 bits com acumulação.**	**Rd = (Rm * RS + Rn)**
umull {<cond>} {S} RdLo, RdHi, Rm, Rs	**Resultado a 64 bits. Multiplicação sem sinal.**	**RdHi:RdLo = Rm * Rs**
umlal {<cond>} {S} RdLo, RdHi, Rm, Rs	**Resultado a 64 bits com acumulação. Multiplicação sem sinal.**	**RdHi:RdLo += Rm * Rs**
smull {<cond>} {S} RdLo, RdHi, Rm, Rs	**Resultado a 64 bits. Multiplicação com sinal.**	**RdHi:RdLo = Rm * Rs**
smlal {<cond>} {S} RdLo, RdHi, Rm, Rs	**Resultado a 64 bits com acumulação. Multiplicação com sinal.**	**RdHi:RdLo += Rm * Rs**

Exemplos:

```
mul  r0, r2, r3
mla  r0, r2, r3, r0
umull r1, r0, r7, r8
```

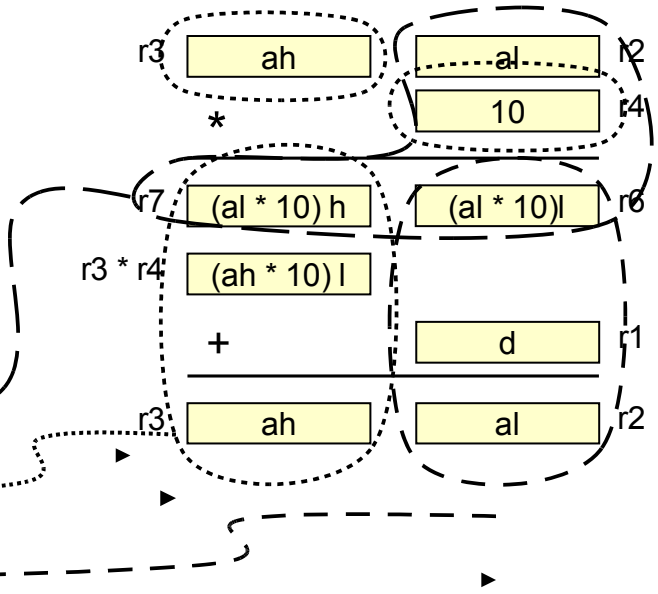
Não se pode usar r15.
Rd, RdHi e RdLo têm que ser diferente de Rm.

Exercícios

Conversão de decimal para binário

```
     .data
string:
     .asciz  "858458458893"
number:
     .word   0,0

     .text
     ldr     r0, =string
     mov     r2, #0
     mov     r3, #0
     mov     r4, #10
1:
     ldrb    r1, [r0], #1
     cmp     r1, #0
     beq     1f
     umull   r6, r7, r2, r4
     mla     r3, r4, r3, r7
     sub     r1, r1, #'0'
     adds    r2, r6, r1
     adc     r3, r3, #0
     b       1b
1:
     ldr     r0, =number
     str     r2, [r0], #4
     str     r3, [r0], #4
     b       .
```



Multiplicação de valores a 64 bits

r5:r4 = r1:r0 x r3:r2

```
     umull     r4, r5, r2, r0
     umlal     r5, r6, r2, r1
     umlal     r5, r6, r3, r0
```

Excepções

O mecanismo de excepções é usado para tratar acontecimentos inesperados durante a execução de um programa.

As excepções são enquadradas em três grupos:

1. Geradas como um efeito directo ao executar uma instrução: interrupções de software, instruções indefinidas, prefetch aborts.
2. Geradas como um efeito secundário de uma instrução: data aborts, memory fault durante um store ou um load
3. Geradas externamente, assincronamente em relação ao fluxo de execução: IRQ, FIQ e Reset.

Na arquitectura ARM estão previstas as seguintes excepções:

Exception	Mode	Address
Reset	SVC	0x00000000
Undefined Instruction	Undefined	0x00000004

Software Interrupt	SVC	0x00000008
Prefetch abort	Abort	0x0000000C
Data abort	Abort	0x00000010
Reserved		0x00000014
IRQ	IRQ	0x00000018
FIQ	FIQ	0x0000001C

Cada modo tem um registo **lr** para guardar o endereço de retorno ao modo anterior; um registo **spsr** para guardar a o **cpsr** do modo interrompida e um registo **sp** próprio para facilitar a mudança de contexto. O modo FIQ tem um banco de registos privado maior que os restantes modos. O que possibilita mudar de contexto mais rapidamente que em outros modos.

Entrada na excepção

Quando uma excepção ocorre o processador termina a execução da instrução corrente. As excepções externas e as de efeito secundário usurpam a execução da próxima instrução.

Muda o modo de operação para o modo correspondente à excepção.
Guarda o valor corrente do **pc** no **lr** do novo modo.
Salva **cpsr** em **spsr** do novo modo.
Desactiva a *flag* IRQ e se a excepção for FIQ desactiva a *flag* FIQ
Força o processador a executar no endereço associado à excepção.

Saída da excepção

Depois da excepção ser tratada o programa interrompido deve ser retomado.
Os registos utilizados devem ser repostos.
O **cpsr** deve ser reposto a partir de **spsr**.
O **pc** deve ser posicionado na instrução apropriada.
Estas duas últimas operações são executadas simultaneamente.
Admitindo que o endereço de retorno está em **lr,** as instruções adequadas para retorno, dependendo da excepção ocorrida, são as seguintes:

`mov pc, lr`	para SWI ou undefined
`subs pc, lr, #4`	para IRQ, FIQ ou prefectch abort
`subs pc, lr, #8`	para data abort

Quando a instrução tem o modificador 's' e o registo de destino é o **pc** então o registo **cpsr** recebe o **spsr**.

Se o endereço de retorno estiver no *stack*, devidamente ajustado, a instrução adquada para repor o **pc** e outros registos é

```
ldmfd sp!, {r0 - r3, pc}^
```

O sinal ^ depois da lista de registos (que inclui necessáriamente o **pc**) provoca o restauro de **cpsr** com o

spsr.

Faltam exemplos de código:
- para preencher a tabela de execepções;
- rotina de tratamento de excepção/ interrupção

Referências

ARM system-on-chip architecture
Capítulo 5