ELTD03z
Microcontroladores/Microprocessadores

Teoria_05a1_5

Prof. Enio R. Ribeiro

Universidade Federal de Itajubá - UNIFEI

T5.1) Instruções deslocamento/rotação

ASR, LSL, LSR, ROR, and RRX
Arithmetic shift right, logical shift left, logical shift right, rotate right, and rotate right with extend.

Syntax

```
op{S}{cond} Rd, Rm, Rs
op{S}{cond} Rd, Rm, #n
RRX{S}{cond} Rd, Rm
```

where:
• '*op*' is one of:
ASR: Arithmetic shift right
LSL: Logical shift left
LSR: Logical shift right
ROR: Rotate right

• '*S*' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see *Conditional execution on page 56*)
• '*Rd*' is the destination register
• '*Rm*' is the register holding the value to be shifted
• '*Rs*' is the register holding the shift length to apply to the value *Rm*. Only the least significant byte is used and can be in the range 0 to 255.
• '*n*' is the shift length. The range of shift lengths depend on the instruction as follows:
ASR: Shift length from 1 to 32; LSL: Shift length from 0 to 31
LSR: Shift length from 1 to 32; ROR: Shift length from 1 to 31

Fig. – Origem: **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.1a) Instrução: deslocamento aritmético à direita

ASR

Arithmetic shift right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. And it copies the original bit[31] of the register into the left-hand *n* bits of the result (see *Figure 13: ASR#3*).

You can use the ASR #*n* operation to divide the value in the register *Rm* by 2^n, with the result being rounded towards negative-infinity.

When the instruction is ASRS or when ASR #n is used in operand2 with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the **carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm***.

Note: 1) If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm.
2) If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm.

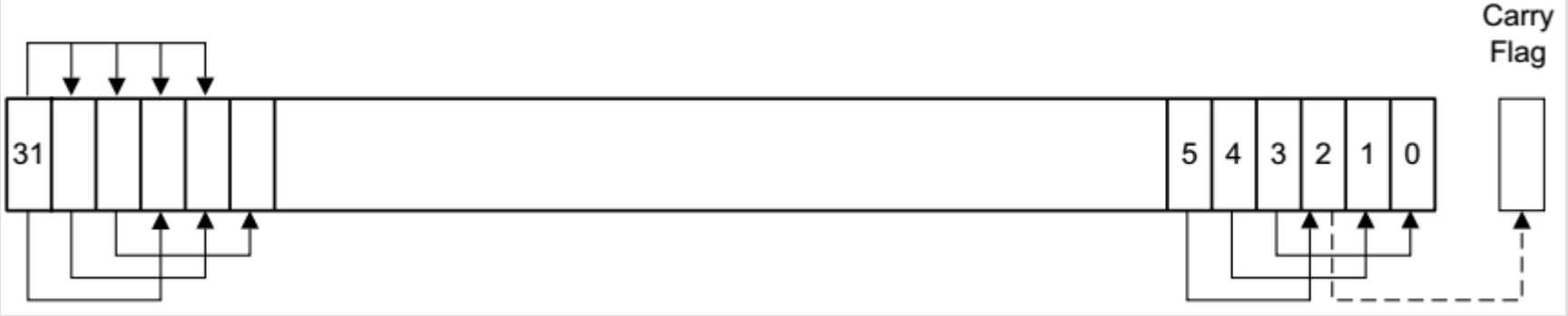
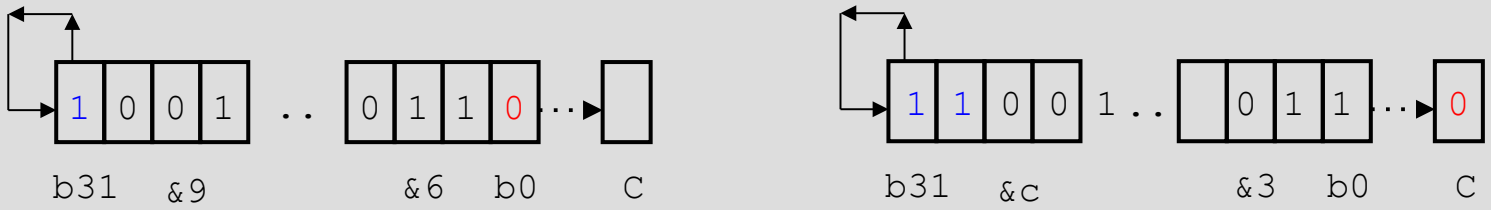


Fig. – ASR #3 : **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.1a) Instrução: deslocamento aritmético à direita - ASR



Efeito aritmético: **divide por 2^n** (n = número de deslocamentos) – **preserva sinal**.

Ex. 5.1a – O vetor vt1 tem 4 bytes. Faça um programa para dividir, cada byte de vt1, por 2 e guarde o resultado em vetor vt2. O programa é cíclico. Faça as designações e alocações necessárias (FDAN). Interpretação: binários sinalizados. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-incremento e outro modo.

Ex. 5.1b – O vetor vt1 tem 4 half-words. Faça um programa para dividir, cada half-word de vt1, por 4 e guarde o resultado em vetor vt2. O programa é cíclico. FDAN. Interpretação: binários sinalizados. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pós-incremento e outro modo.

T5.1b) Instrução: deslocamento lógico à direita

LSR

Logical shift right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. And it sets the left-hand *n* bits of the result to 0 (see *Figure 14*).

You can use the LSR #n operation to divide the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer.

When the instruction is LSRS or when LSR #n is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

Note: 1) If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm.
2) If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm.

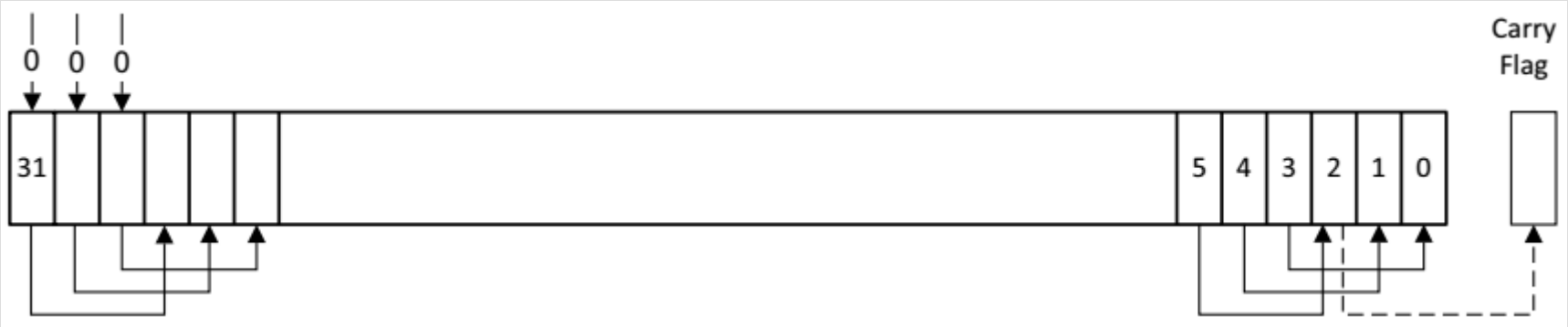
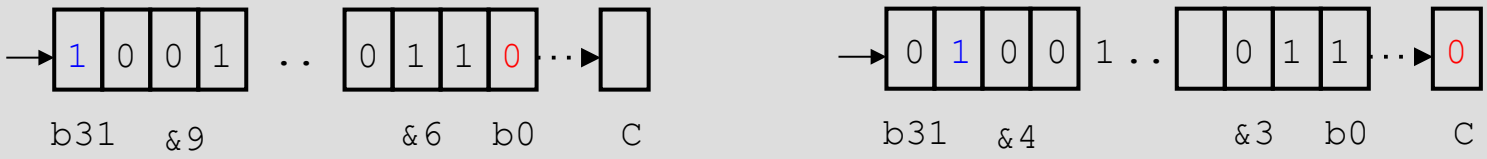


Fig. – LSR #3 : **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.1b) Instrução: deslocamento lógico à direita - LSR



Efeito aritmético: **divide por 2^n** (n = número de deslocamentos) – **não preserva sinal (valor interpretado -> binário puro)**.

Ex. 5.1c – O vetor vt1 tem 4 bytes. Faça um programa para dividir, cada byte de vt1, por 2 e guarde o resultado em vt2, se o valor original for par, ou em vt3, se o valor original for ímpar. O programa é cíclico. FDAN. Interpretação: binários puros. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-incremento e outro modo.

Ex. 5.1d – O vetor vt1 tem 4 half-words. Faça um programa para dividir, cada half-word de vt1, por 4 e guarde o resultado em vt2, se o valor original for par, ou em vt3, se o valor original for ímpar. O programa é cíclico. FDAN. Interpretação: binários puros. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-incremento e outro modo.

T5.1c) Instrução: deslocamento lógico à esquerda

LSL

Logical shift left by *n* bits moves the right-hand 32-*n* bits of the register *Rm*, to the left by *n* places, into the left-hand 32-*n* bits of the result. And it sets the right-hand *n* bits of the result to 0 (see *Figure 15: LSL#3*).

You can use the LSL #n operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LSLS or when LSL #*n*, with non-zero *n*, is used in *operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[32-*n*], of the register *Rm*. These instructions do not affect the carry flag when used with LSL #0.

Note: 1) If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm.
2) If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm.

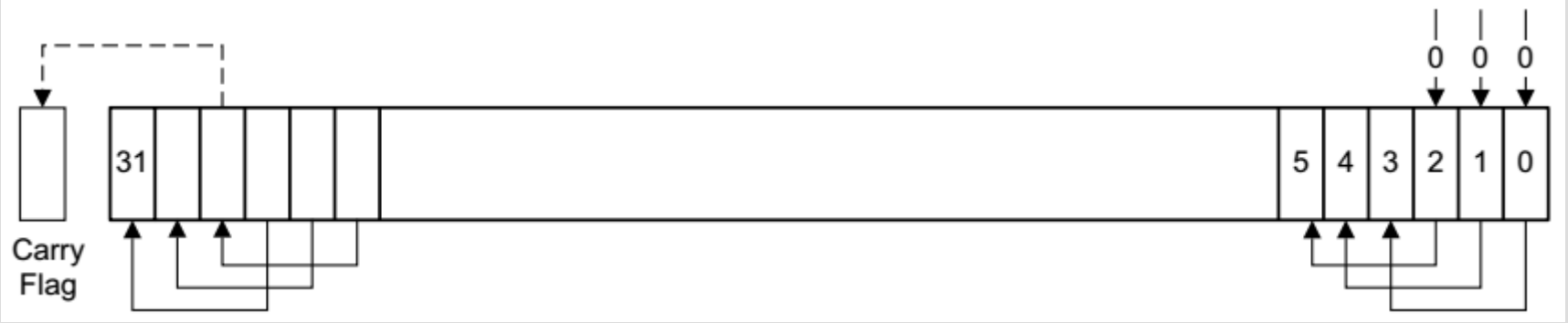


Fig. – LSL #3 : **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.1c) Instrução: deslocamento lógico à esquerda - LSL

Efeito aritmético: **multiplica por 2^n** (n = número de deslocamentos)

Ex. 5.1e – O vetor vt1 tem 4 bytes. Faça um programa para multiplicar, cada byte de vt1, por 2 e guarde o resultado em vt2. O programa é cíclico. FDAN. Interpretação: binário sinalizado. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-decremento e outro modo.

Ex. 5.1f – O vetor vt1 tem 4 half-words. Faça um programa para multiplicar, cada half-word de vt1, por 4 e guarde o resultado em vt2. O programa é cíclico. FDAN. Interpretação: binário sinalizado. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-decremento e outro modo.

T5.1d) Instrução: rotação à direita

ROR

Rotate right by n bits moves the left-hand 32-n bits of the register Rm, to the right by n places, into the right-hand 32-n bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result (see Figure 16).

When the instruction is RORS or when ROR #n is used in operand2 with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit rotation, bit[n-1], of the register Rm.

Note: 1) If n is 32, then the value of the result is same as the value in Rm, and if the carry flag is updated, it is updated to bit[31] of Rm.
2) ROR with shift length, n, more than 32 is the same as ROR with shift length n-32.

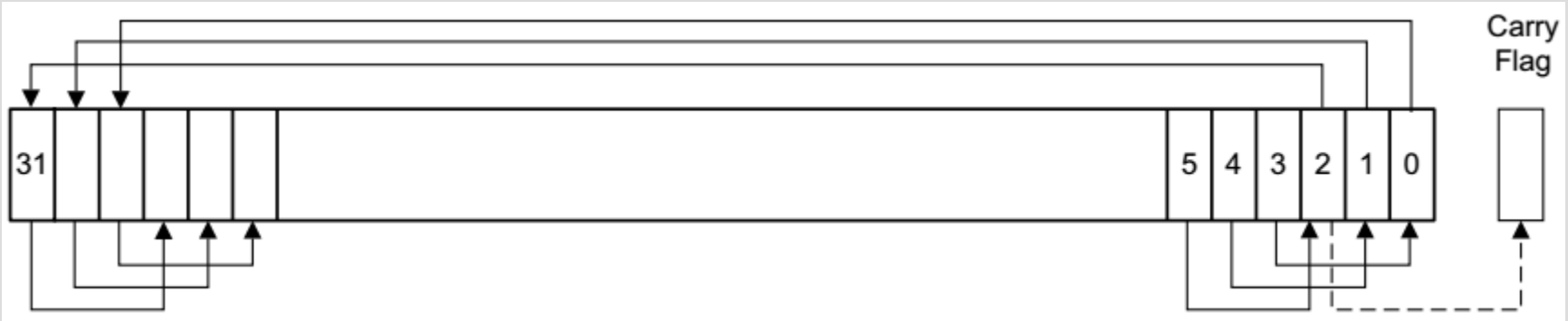
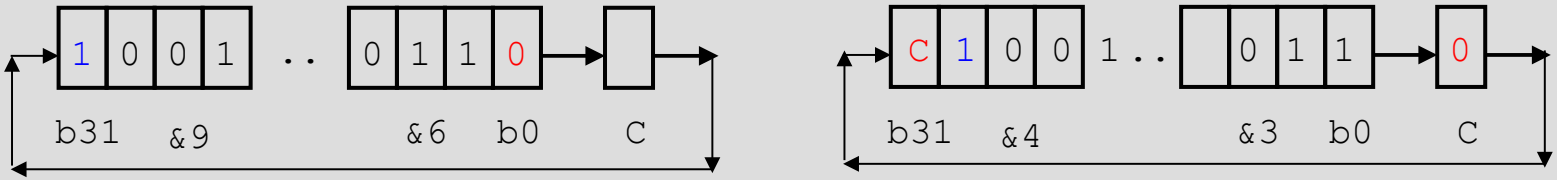


Fig. – ROR #3 : **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.1d) Instrução: rotação à direita - ROR

Efeito aritmético: **não considerado**

T6.4c) Instrução: rotação à direita - RRX



T5.1d) Instrução: rotação à direita - ROR

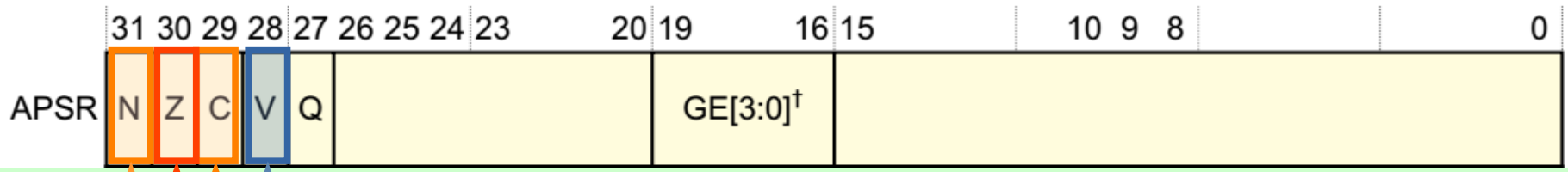
Efeito aritmético: **não considerado**

Ex. 5.1g – O vetor vt1 tem 4 bytes. Faça um programa para copiar cada byte de vt1: em vt2, se o conteúdo do byte for par; em vt3 se o conteúdo do byte for ímpar. O programa é cíclico. FDAN. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-decremento e outro modo.

Ex. 5.1h – O vetor vt1 tem 4 half-words. Faça um programa para copiar cada half-word de vt1: em vt2, se o conteúdo da half-word for par; em vt3 se o conteúdo da half word for ímpar. O programa é cíclico. FDAN. Interpretação: binário sinalizado. Usar endereçamento indexado com apenas um ponteiro. Usar ponteiro com auto pré-incremento e outro modo.

T5.2) REGISTRO : Current Program Status Register - (xPSR ~> APSR)

REGISTRO ESTADO (CONDIÇÃO!)



FLAG V – (Overflow): Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

FLAG C – (Carry or borrow)): Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit;
1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.

FLAG Z – (Zero): Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

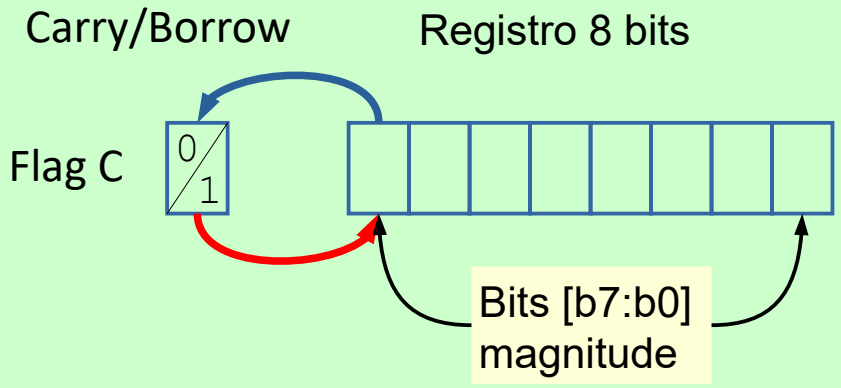
FLAG N – (Negativo): Negative condition code flag. Set to bit[31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

0: Operation result was positive, zero, greater than, or equal;
1: Operation result was negative or less than.

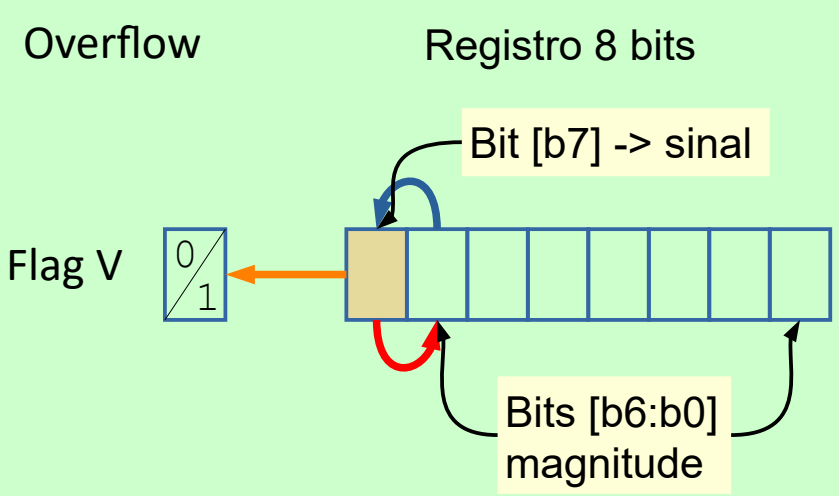
Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}-1 or less than -2^{31}.

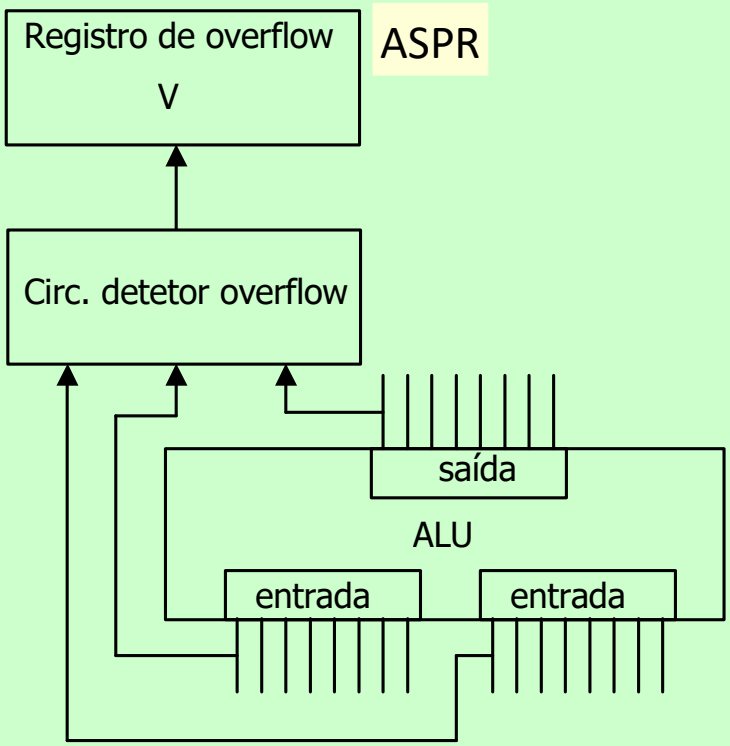
T5.2a) REGISTRO : (xPSR ~> APSR) - - condição de overflow

Interpretação: Binários puros (não sinalizados)

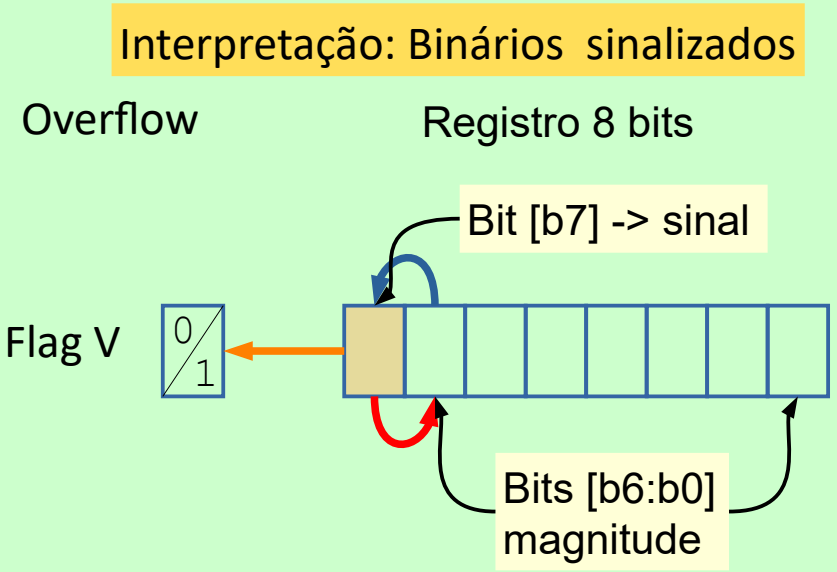


Interpretação: Binários sinalizados





T5.2b) REGISTRO : (xPSR ~> APSR) - condição de overflow (visualização gráfica)



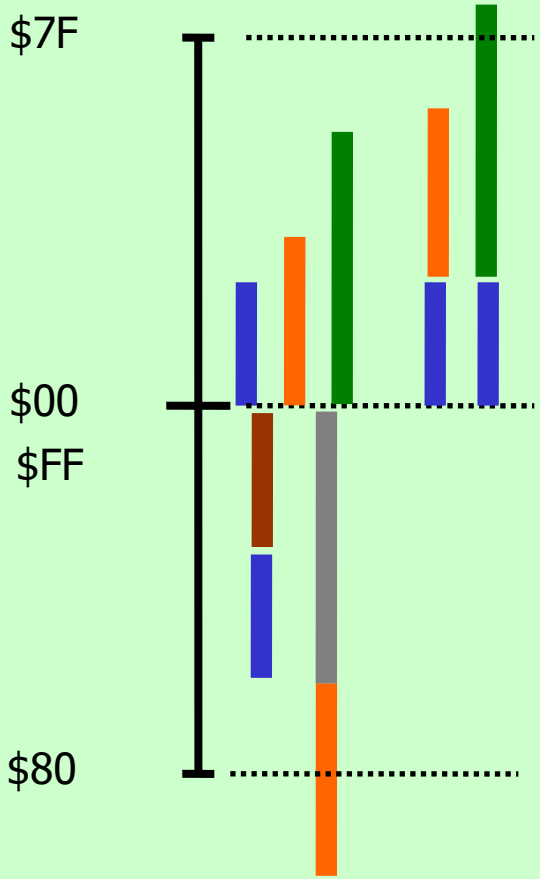
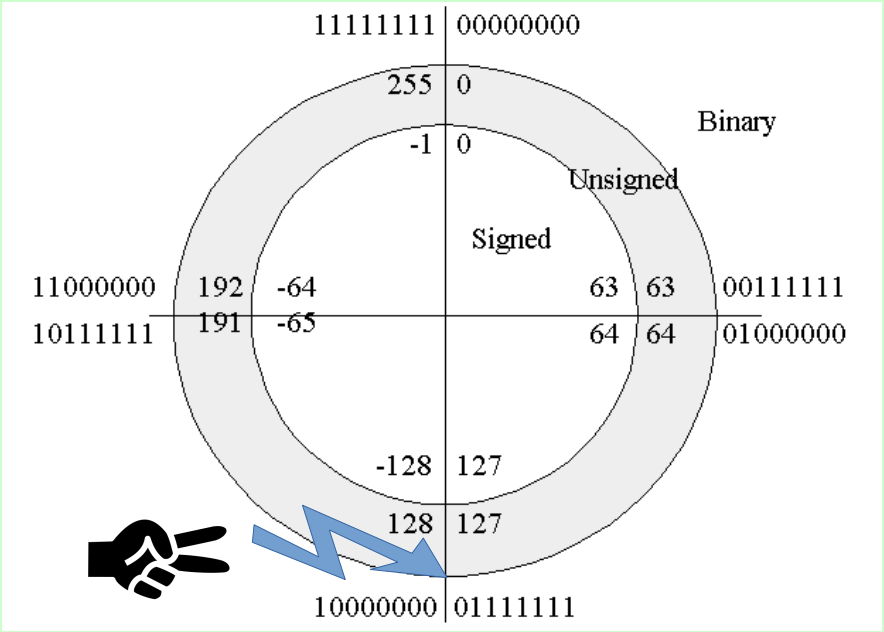


Fig. – Origem: **https://www.massey.ac.nz/~mjjohnso/notes/59102/notes/l2.html**



- **VP (+) VP = VP -> VN -> V=1**
- **VN (+) VN = VN -> VP -> V=1**
- **VP (-) VN = VP -> VN -> V=1**
- **VN (-) VP = VN -> VP -> V=1**

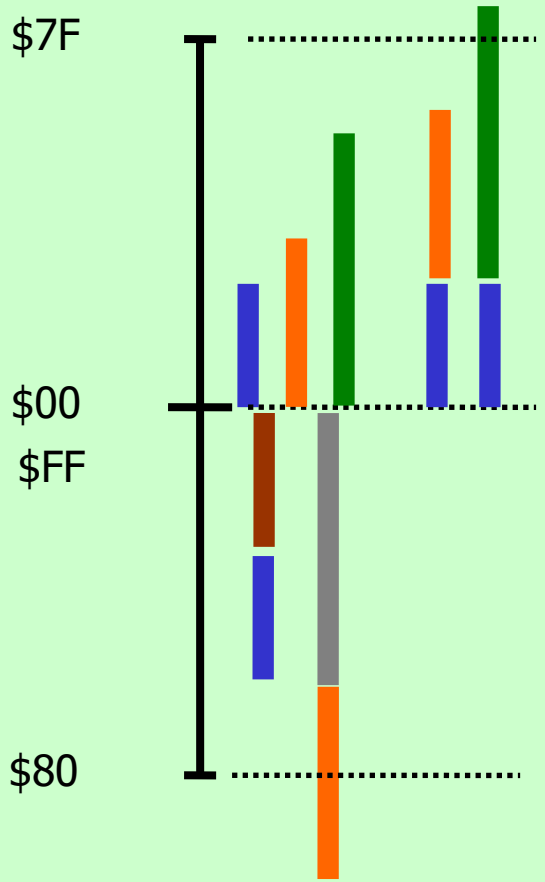
T5.2c) REGISTRO : (xPSR ~> APSR) - - instruções relacionadas ao overflow

Table 23. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N ! = V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N ! = V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix is specified.

Fig. 1 – Fonte da "Table 23": **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.2d) REGISTRO : APSR – Flag V - exercícios



- **VP (+) VP = VP -> VN -> V=1 - [a]**
- **VN (+) VN =VN -> VP -> V=1 - [b]**
- **VP (-) VN = VP -> VN -> V=1 - [c]**
- **VN (-) VP = VN -> VP -> V=1 - [d]**

Ex. 5.2a) As variáveis g1 e h1 são de 1 byte e binários sinalizados (0 ≤ g1,h1 < &80). Adicione g1 e h1 e coloque o resultado: (a) em r1b (1 byte) se ele puder ser representado com 1 byte; (b) ou em r2b (2 bytes) se ele necessitar de ser representado com 2 bytes. O programa é cíclico. FDAN. (Usar Flag V).

Ex. 5.2b) Refaça o exercício Ex. 6.a para condição [b] e considere as variáveis g1 e h1 com os seguintes valores: (&80 ≤ g1,h1 ≤ &ff). (Usar Flag V).

Ex. 5.2c) Refaça o exercício Ex. 6.a para condição [c], isto é, fazer (g1-h1). Considere as variáveis g1 e h1 com os seguintes valores: (&0 ≤ g1 ≤ &7f) e (&80 ≤ h1 ≤ &ff). (Usar Flag V).

Ex. 5.2d) Refaça o exercício Ex. 6.a para condição [d], isto é, fazer (g1-h1). Considere as variáveis g1 e h1 com os seguintes valores: (&80 ≤ g1 ≤ &ff) e (&0 ≤ h1 ≤ &7f). (Usar Flag V).

T5.7a) MODO ENDEREÇAMENTO INDEXADO -> possibilidades

Instruction: **LDR/STR**

LDR instructions load one (or two) register(s) with a value from memory. STR instructions store one (or two) register value(s) to memory.

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]    ; immediate offset
op{type}{cond} Rt, [Rn, #offset]!      ; pre-indexed
op{type}{cond} Rt, [Rn], #offset       ; post-indexed
```

- '*op* ' is either LDR (load register) or STR (store register);
- '*type* ' is one of the following: **B**, **SB**, **H**, **SH** or omit, for word;
- '*cond* ' is an optiional conditional code;
- '*Rt* ' is the register to load or store;
- '*Rn* ' is the register on which the memory address is based;
- '*offset* ' is an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*.

(memory to/from register) – **não é pseudo-instrução**

- **LDRH** – load halfword (16 bit unsigned #) / zero-extend to 32 bits
- **LDRSH** – load signed halfword / sign-extend to 32 bits
- **LDRB** – load byte (8 bit unsigned #) / zero-extend to 32 bits
- **LDRSB** – load signed byte / sign-extend to 32 bits
- **STRH** – store 16-bit halfword (right-most16 bits of register)
- **STRB** - store 8-bit byte (right-most 8 bits of register)

T5.7b) MODO ENDEREÇ. INDEXADO OFFSET IMMEDIATE -> offset fixo

Operation

Load and store instructions with immediate offset can use the following addressing modes:

• Offset addressing (or Offset indexed) => **END. INDEXADO COM OFFSET IMEDIATO (FIXO)**

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access. The register *Rn* is unaltered. The assembly language syntax for this mode is:

[*Rn*, *#offset*]

• Pre-indexed addressing => **END. INDEXADO COM (AUTO) PRÉ-(INCREMENTO/DECREMENTO)**

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access and written back into the register *Rn*. The assembly language syntax for this mode is:

[*Rn*, *#offset*]!

• Post-indexed addressing => **END. INDEXADO COM (AUTO) PÓS-(INCREMENTO/DECREMENTO)**

The address obtained from the register *Rn* is used as the address for the memory access. The offset value is added to or subtracted from the address, and written back into the register *Rn*. The assembly language syntax for this mode is:

[*Rn*], *#offset*.

Table 25. Immediate, pre-indexed and post-indexed offset ranges

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	-255 to 4095	-255 to 255	-255 to 255

Fig. 2 – Fonte da "Table 25": **cd00228163-stm32f10xxx-cortex-m3-programming-manual-stmicroelectronics.pdf**

T5.8.1a) MODO END. INDEX. COM OFFSET IMMEDIATE (fixo)

O offset é um número sinalizado a ser adicionado ao conteúdo do registro base (ou a ser subtraído do conteúdo do registro base). O resultado é usado como endereço para acesso à memória. **O registro Rn não é alterado (Rn é o registro base)**.

Immediate offset indexed addressing é útil para acessar elementos que estão a certa distância do local indicado/apontado pelo registro base (base/index/pointer register), por exemplo, as stack offsets and input/output registers.

Syntax:

Load data from memory: `LDR{type} {cond} Rt, [Rn {, #imm }]`

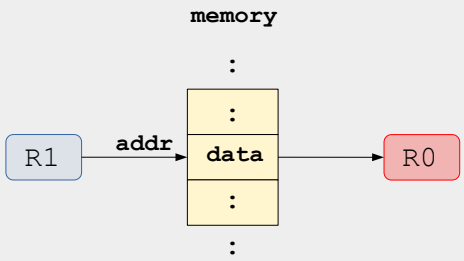
Save data to memory: `STR{type} {cond} Rt, [Rn {, #imm }]`

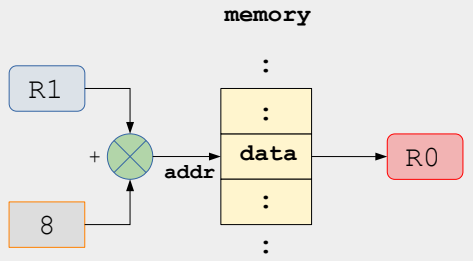
Load with zero immediate offset

```
LDR   R0, [R1]    ;R0<=[R1]
LDR   R0, [R1,#0] ;R0<=[R1]
```

Load with immediate offset (addressed by R1+8)

```
LDR       R0,[R1,#8]    ;R0<=[R1+8]
```

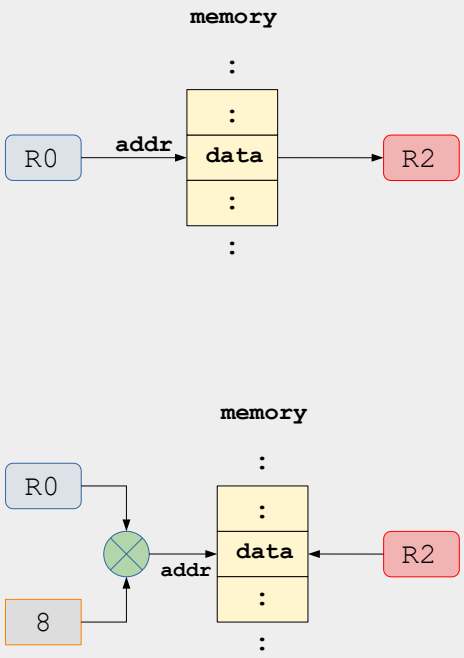




T5.8.1b) MODO END. INDEX. COM OFFSET IMMEDIATE (fixo)

Ex. 1d – O vetor vt1 tem 8 bytes. Faça um programa para copiar os bytes de vt1 no vetor vt2. O programa é cíclico. Faça as designações e alocações necessárias (FDAN). (O usuário calcula o offset entre os vetores!).

```
;---Ex. 1d - End.Index. Offset imediato
    export   __main
;=== diretiva area - dados (sram)
    area     dds1, data, readwrite
vt1 space    8
vt2 space    8
;=== diretiva area - prog. (flash)
    area     m_prog, code, readonly
__main
    ldr      r0,=vt1  ;load pointer vt1
    mov      r1,#8    ;counter
pk0 ldrsb    r2,[r0]  ;ler vt1(i)
    strb     r2,[r0,#8];salvar em vt2(i)
    add      r0,#1    ;inc.pointer
    subs     r1,#1    ;
    bne      pk0      ;prox.elem.
    b        __main   ;
    end
```



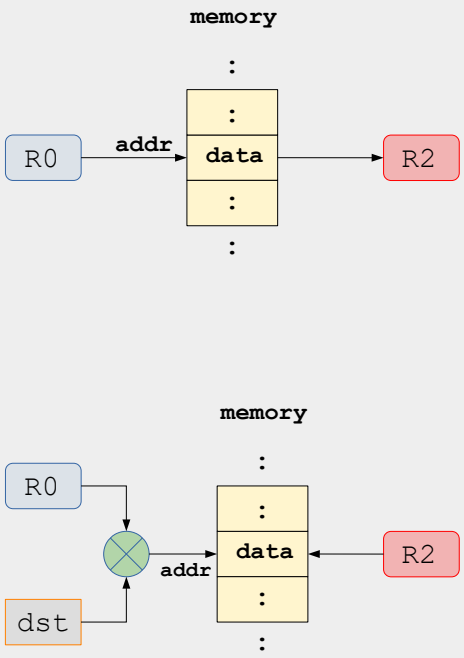
Ex. 1d_1 – Refaça o (Ex.1d) carregando o registro r0 com o endereço de vt2. Acrescente, entre vt1 e vt2, um vetor de 4 bytes. (O usuário calcula o offset entre os vetores!).

```
                __main:
0x0800024C 4805      LDR      r0,[pc,#20]   ;@0x08000264
0x0800024E F04F0108  MOV      r1,#0x08
0x08000252 F9902000  LDRSB    r2,[r0,#0x00]
0x08000256 7202      STRB     r2,[r0,#0x08]
0x08000258 F1000001  ADD      r0,r0,#0x01
0x0800025C 3901      SUBS     r1,r1,#0x01
0x0800025E D1F8      BNE      0x08000252
0x08000260 E7F4      B        0x0800024C __main
```

T5.8.1c) MODO END. INDEX. COM OFFSET IMMEDIATE (fixo)

Ex. 1e – O vetor vt1 tem 8 bytes. Faça um programa para copiar os bytes de vt1 no vetor vt2. O programa é cíclico. Faça as designações e alocações necessárias. O cálculo do offset entre os vetores deve ser dinâmico (pelo software).

```
;---Ex. 1e - End.Index. Offset imediato
    export   __main
;=== dir. equate
dst equ      vt2-vt1  ;(final-inicial)
;=== diretiva area - dados (sram)
    area     dds1, data, readwrite
vt1 space    8
vtw dcb      &a7,&3b,&5e
vtz dcw      &39af,&b287
vt2 space    8
;=== diretiva area - prog. (flash)
    area     m_prog, code, readonly
__main
    ldr      r0,=vt1  ;load pointer vt1
    mov      r1,#8    ;counter
pk0 ldrsb    r2,[r0]  ;ler vt1(i)
;   strb     r2,[r0,#?];salvar em vt2(i)
    strb     r2,[r0,#dst];salvar em vt2(i)
    add      r0,#1    ;inc.pointer
    subs     r1,#1    ;
    bne      pk0      ;prox.elem.
    b        __main   ;
    end
```




```
                __main:
0x0800024C 4805      LDR      r0,[pc,#20]   ; @0x08000264
0x0800024E F04F0108  MOV      r1,#0x08
0x08000252 F9902000  LDRSB    r2,[r0,#0x00]
0x08000256 7402      STRB     r2,[r0,#0x10]
0x08000258 F1000001  ADD      r0,r0,#0x01
0x0800025C 3901      SUBS     r1,r1,#0x01
0x0800025E D1F8      BNE      0x08000252
0x08000260 E7F4      B        0x0800024C __main
```

Ex.1e_1 – Refaça o (Ex.1e) iniciando a cópia dos bytes de vt1 a partir do último byte de vt1.

T5.8.2a) MODO END. INDEX. (AUTO) PRÉ-INCREMENTO/DECREMENTO => Offset Immediate

O offset é um número sinalizado a ser adicionado ao conteúdo do registro base (ou a ser subtraído do conteúdo do registro base). O resultado é escrito (atualizado) no registro Rn e, em seguida, é usado como endereço para o acesso à memória.

Pre-indexed = INDEXADO (AUTO) PRÉ-INCREMENTO/DECREMENTO

Pre-indexed addressing (**endereçamento indexado com pré-incremento/decremento**) é útil para acessar elementos que estão a certa distância do local indicado/apontado pelo registro base (base/index/pointer register), por exemplo, como stack offsets and input/output registers.

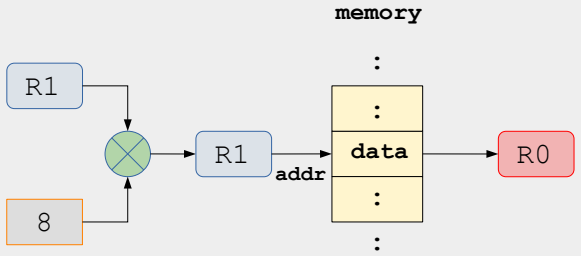
Syntax:

Load data from memory: `LDR{size} {cond} Rt, [Rn {, #imm }]!`

Save data to memory: `STR{size} {cond} Rt, [Rn {, #imm }]!`

Load with pre-indexed immediate offset (addressed by R1+8)

```
LDR      R0,[R1,#8]!   ;R1=R1+8, R0<=[R1+8]
```

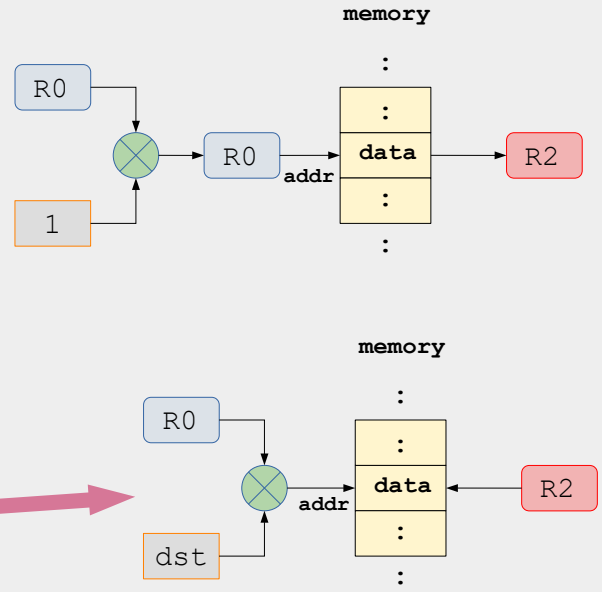


T5.8.2b) MODO END. INDEX. (AUTO) PRÉ-INCREMENTO/DECREMENTO => Offset Immediate

Ex. 1f – O vetor vt1 tem 8 bytes. Faça um programa para copiar os bytes de vt1 no vetor vt2. O programa é cíclico. FDAN. O cálculo do offset entre os vetores deve ser dinâmico (pelo software). Use **pre-increment/decrement indexed offset addressing**.

```
;---ex1_f - End.Index. Offset imediato
;(auto)com pré-incremento
    export   __main

;===== diretiva area - dados (sram)
    area     dds1, data, readwrite
vt1 space    8
vtz dcw      &39af,&b287
vt2 space    8
;=== diretiva area - prog. (flash)
    area     m_prog, code, readonly
__main
    ldr      r0,=(vt1-1)  ;load pointer vt1
    mov      r1,#8        ;counter
pk0 ldrsb    r2,[r0,#1]!  ;ler vt1(i)
    strb     r2,[r0,#vt2-vt1] ;save em vt2(i)
;   add      r0,#1        ;inc.pointer
    subs     r1,#1        ;decr.counter
    bne      pk0          ;prox.elem.
    b        __main       ;
    end
```




```
                __main:
0x080002BC 4804      LDR      r0,[pc,#16]  ; @0x080002D0
0x080002BE F04F0108  MOV      r1,#0x08
0x080002C2 F8102F01  LDRSB    r2,[r0,#0x01]!
0x080002C6 7302      STRB     r2,[r0,#0x0C]
0x080002C8 3901      SUBS     r1,r1,#0x01
0x080002CA D1FA      BNE      0x08000252
0x080002CC E7F6      B        0x0800024C __main
```

Ex.1f_1 – Refaça o (Ex.1f) iniciando a cópia dos bytes de vt1 a partir do último byte de vt1.

T5.8.3a) MODO END. INDEX. (AUTO) PÓS-INCREMENTO/DECREMENTO => Offset Immediate

Primeiro, o endereço obtido do registro *Rn* é usado como o endereço de acesso à memória. O offset é um número sinalizado, que será adicionado ao conteúdo do registro base (ou será subtraído do conteúdo do registro base). O resultado dessa operação é escrito (atualizado) no registro Rn.

Post-indexed = INDEXADO (AUTO) PÓS-INCREMENTO/DECREMENTO

Post-indexed addressing (**endereçamento indexado com pós-incremento/decremento**) é útil para acessar elementos que estão a certa distância do local indicado/apontado pelo registro base (base/index/pointer register), por exemplo, como stack offsets and input/output registers.

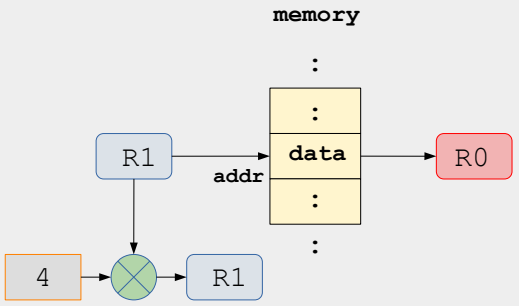
Syntax:

Load data from memory: `LDR{size} {cond} Rt, [Rn], #imm`

Save data to memory: `STR{size} {cond} Rt, [Rn], #imm`

Load with post-indexed immediate offset (addressed by R1+4)

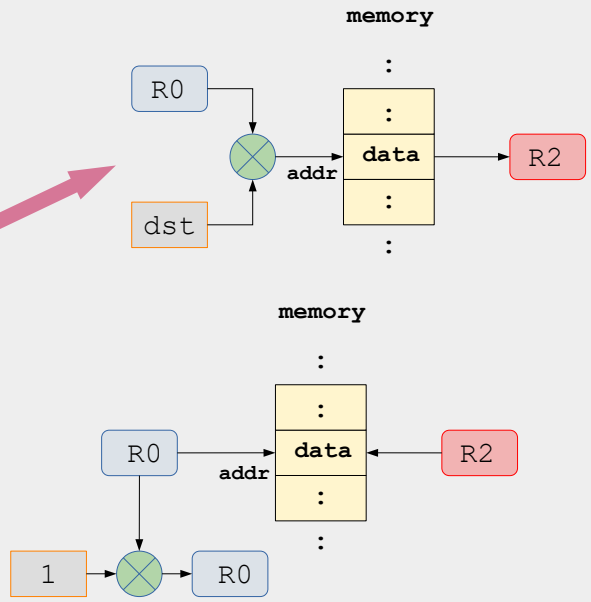
```
LDR      R0,[R1],#4   ;R0<=[R1], R1=R1+4
```



T5.8.3b) MODO END. INDEX. (AUTO) PÓS-INCREMENTO/DECREMENTO => Offset Immediate

Ex. 1_g – O vetor vt1 tem 8 bytes. Faça um programa para copiar os bytes de vt1 no vetor vt2. O programa é cíclico. FDAN. O cálculo do offset entre os vetores deve ser dinâmico (pelo software). Use post-indexed offset addressing.

```
;---ex1_g - End.index.offset imediato
;(auto) pós-incremento
    export   __main
;=====
;===== diretiva area - dados (sram)
    area     dds1, data, readwrite
vt1 space    8
vtz dcw      &39af,&b287
vtw dcb      &a7,&3b,&c5
vt2 space    8
;=== diretiva area - prog. (flash)
    area     m_prog, code, readonly
__main
    ldr      r0,=vt2        ;load pointer vt2
    mov      r1,#8          ;counter
pk0 ldrb     r2,[r0,#vt1-vt2] ;ler vt1(i)
    strb     r2,[r0],#1     ;salvar em vt2(i)
;   add      r0,#1          ;inc.pointer NÃO!!!
    subs     r1,#1          ;decr.counter
    bne      pk0            ;prox.elem.
    b        __main         ;
    end
```




```
                __main:
0x0800024C 4804      LDR      r0,[pc,#16]   ;@0x08000260
0x0800024E F04F0108  MOV      r1,#0x08
0x08000252 F8102C0F  LDRB     r2,[r0,#-0x0F]
0x08000256 F8002B01  STRB     r2,[r0],#0x01
0x0800025A 3901      SUBS     r1,r1,#0x01
0x0800025C D1F9      BNE      0x08000252
0x0800025E E7F5      B        0x0800024C __main
```

Ex.1_g_1 – Refaça o (Ex.1_g) iniciando a cópia dos bytes de vt1 a partir do último byte de vt1.