QXD0133 - Arquitetura e Organização de Computadores II

Universidade Federal do Ceará - Campus Quixadá

Thiago Werlley
thiagowerlley@ufc.br

18 de outubro de 2025

Capítulo 3

Capítulo 3

Conjunto de instruções ARM

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:
  - *Data processing instructions*

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:
  - *Data processing instructions*
  - *Branch instructions*

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:
  - *Data processing instructions*
  - *Branch instructions*
  - *Load-Store instructions*

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:
  - *Data processing instructions*
  - *Branch instructions*
  - *Load-Store instructions*
  - *Software interrupt instructions*

Introdução

- Instruções ARM processam dados armazenados nos registradores e somente acessam memória com instruções de *load* e *store*.
- Classes (ou grupos) de instruções:
  - *Data processing instructions*
  - *Branch instructions*
  - *Load-Store instructions*
  - *Software interrupt instructions*
  - *Program status register instructions*

Codificação das instruções

- As instruções seguem alguns padrões de codificação

Codificação das instruções

- As instruções seguem alguns padrões de codificação
- Instruções fora do padrão:

Codificação das instruções

- As instruções seguem alguns padrões de codificação
- Instruções fora do padrão:
  - Indefinidas (*Undefined Exception*)

Codificação das instruções

- As instruções seguem alguns padrões de codificação
- Instruções fora do padrão:
  - Indefinidas (*Undefined Exception*)
  - Comportamento imprevisível

	31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6 5	4	3 2 1 0
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount		shift	0	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	x	x x	0	x x x x
Data processing register shift [2]	cond [1]	0 0 0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	0	x x	1	x x x x
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0 0 0	x x x x	x	x x x x	x x x x	x x x x	1	x x	1	x x x x
Data processing immediate [2]	cond [1]	0 0 1	opcode	S	Rn	Rd	rotate	immediate			
Undefined instruction [3]	cond [1]	0 0 1	1 0 x 0	0	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	
Move immediate to status register	cond [1]	0 0 1	1 0 R 1	0	Mask	SBO	rotate	immediate			
Load/store immediate offset	cond [1]	0 1 0	P U B W	L	Rn	Rd	immediate				
Load/store register offset	cond [1]	0 1 1	P U B W	L	Rn	Rd	shift amount		shift	0	Rm
Undefined instruction	cond [1]	0 1 1	x x x x	x	x x x x	x x x x	x x x x	x x x x	x x x	1	x x x x
Undefined instruction [4,7]	1 1 1 1	0 x x x	x x x x	x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	
Load/store multiple	cond [1]	1 0 0	P U S W	L	Rn	register list					
Undefined instruction [4]	1 1 1 1	1 0 0	x x x x	x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	
Branch and branch with link	cond [1]	1 0 1	L	24-bit offset							
Branch and branch with link and change to Thumb [4]	1 1 1 1	1 0 1	H	24-bit offset							
Coprocessor load/store and double register transfers [6]	cond [5]	1 1 0	P U N W	L	Rn	CRd	cp_num	8-bit offset			
Coprocessor data processing	cond [5]	1 1 1 0	opcode1		CRn	CRd	cp_num	opcode2		0	CRm
Coprocessor register transfers	cond [5]	1 1 1 0	opcode1	L	CRn	Rd	cp_num	opcode2		1	CRm
Software interrupt	cond [1]	1 1 1 1	swi number								
Undefined instruction [4]	1 1 1 1	1 1 1 1	x x x x	x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	

Data Processing Instructions

- *Move Instructions*
- *Barrel Shifter*
- *Arithmetic Instructions*
- *Logical Instructions*
- *Comparison Instructions*
- *Multiply Instructions*

Data Processing Instructions - Move Instructions

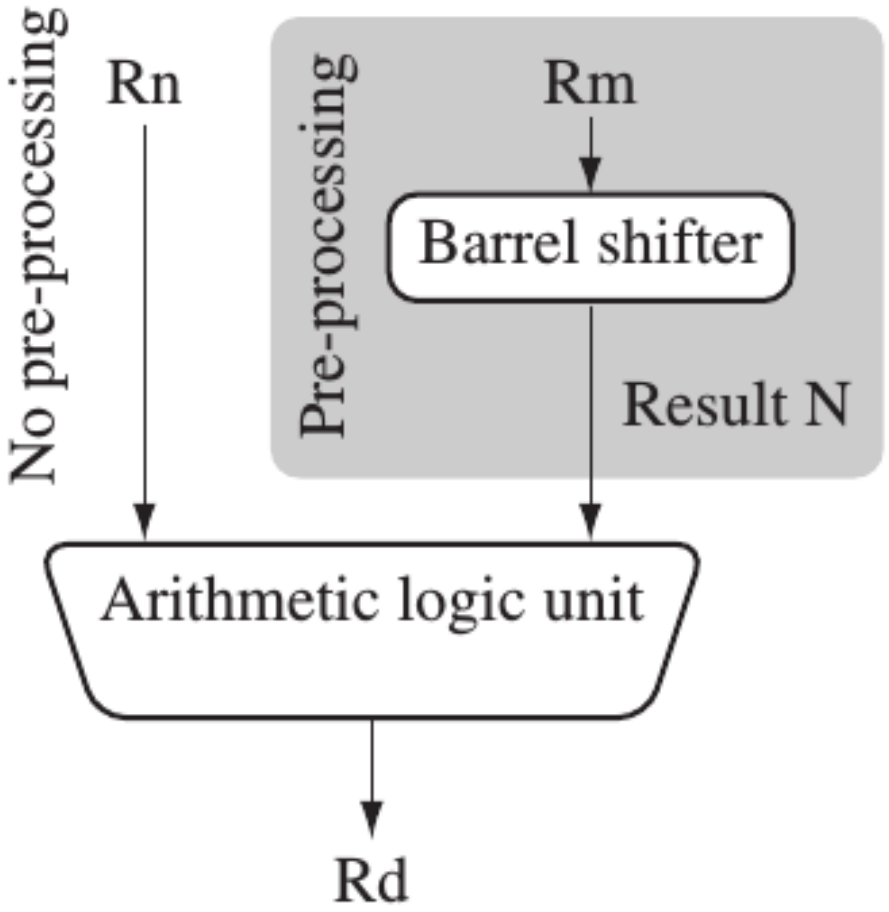
Syntax: <instruction>{<cond>}{S} Rd, N

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- Exemplo:

```
PRE     r5 = 5
        r7 = 8
        MOV    r7, r5    ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Data Processing Instructions - Barrel Shifter



Data Processing Instructions - Barrel Shifter

Table 3.2 Barrel shifter operations.

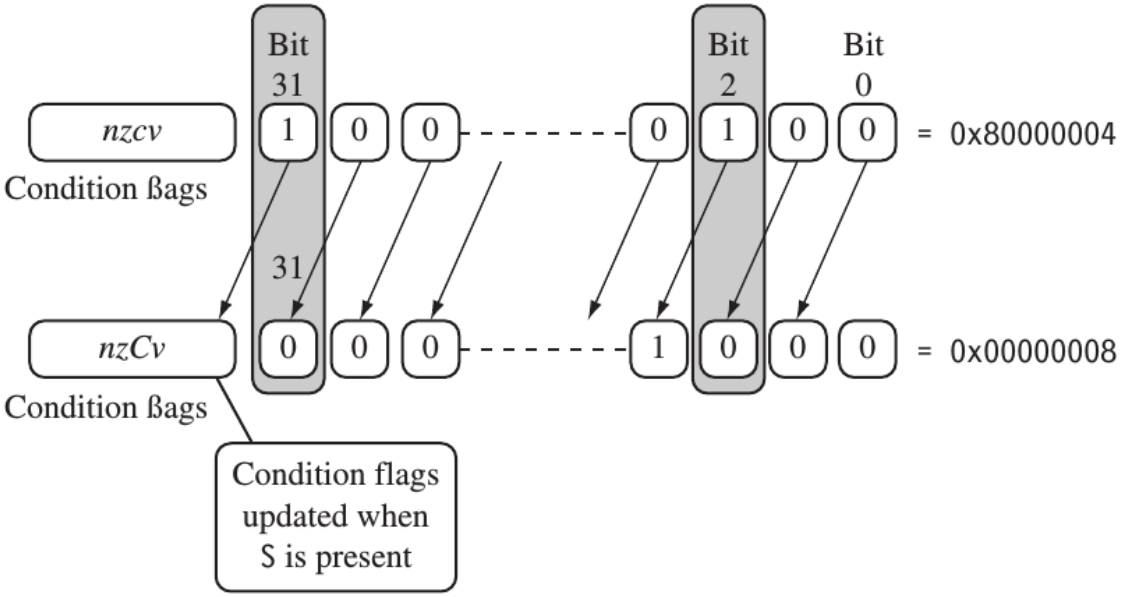
Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	xLSL y	$x \ll y$	#0–31 or Rs
LSR	logical shift right	xLSR y	(unsigned)$x \gg y$	#1–32 or Rs
ASR	arithmetic right shift	xASR y	(signed)$x \gg y$	#1–32 or Rs
ROR	rotate right	xROR y	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or Rs
RRX	rotate right extended	xRRX	(c flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

- Exemplo:

```
PRE     r5 = 5
        r7 = 8

        MOV     r7, r5, LSL #2   ; let r7 = r5*4 = (r5<<2)

POST    r5 = 5
        r7 = 20
```

Data Processing Instructions - Barrel Shifter

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```



Data Processing Instructions - Arithmetic Instructions

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

Data Processing Instructions - Arithmetic Instructions

Adição e subtração

- Exemplo 1:

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

- Exemplo 2:

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

Data Processing Instructions - Arithmetic Instructions

Subtração com carry

- No x86:
  - a – b, quando $a < b, C = 1$
  - C $\rightarrow$ Borrow
  - Exemplo: 0 – 1 = 0xFFFFFFFF $\rightarrow$ C = 1
- No ARM:
  - a – b = **a + not (b) + 1** $\rightarrow$ Complemento de 2
  - C $\rightarrow$ Carry
  - Exemplo 1: 0 – 1 = 0xFFFFFFFF $\rightarrow$ C = 0
  - Exemplo 2: 1 – 0 = 0x00000001 $\rightarrow$ C = 1

Data Processing Instructions

- Exemplo:

```
A = 0x80000001 00000000
;B = 0x00000000 00000001
;C = A - B = 0x80000000 ffffffff
;————————————————————
mov r0,#0                ; A Low
mov r1,#0x80000001       ; A High
;————————————————————
mov r2,#1                ; B Low
mov r3,#0                ; B High
;————————————————————
subs R4, R0, R2          ; C Low
Sbc R5, R1, R3           ; C High
```

Data Processing Instructions - Logical Instructions

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \,\&\, N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \,\hat{}\, N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \,\&\, \sim N$

Data Processing Instructions - Logical Instructions

- Exemplo:

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
Rd = Rn AND NOT(N)
```

Data Processing Instructions - Comparison Instructions

- Não alteram os registradores
- Não precisam do sufixo S para atualizar as flags do CPSR

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

Data Processing Instructions - Comparison Instructions

- Exemplo:

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

Data Processing Instructions - Multiply Instructions

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^* Rs) + Rn$
MUL	multiply	$Rd = Rm^* Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

Data Processing Instructions - Multiply Instructions

- Exemplo 1:

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2   ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

- Exemplo 2:

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Branch Instructions

- Instruções que alternam o fluxo do programa
  - Usadas em chamada de rotinas, loops e estruturas condicionais.
  - Endereço chamado deve estar distante no máximo 32MB (antes ou depois)
  - O fluxo do programa também pode ser alterado movendo-se um valor diretamente para o registrador **r15** (PC)

Branch Instructions

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

Branch Instructions

- Exemplo 1:

```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
```

```
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

Branch Instructions

- Exemplo 2:

```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

Branch conditions

Mnemonic	Name	Condition flags
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or same	C
CC LO	carry clear/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

Branches

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Load-Store Instructions

- Single-register

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

Load-Store Instructions

- Single-register

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Load-Store Instructions

- Single-register

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]            ; = STR r0, [r1, #0]
```

Load-Store Instructions

- Single-register Load-store addressing mode

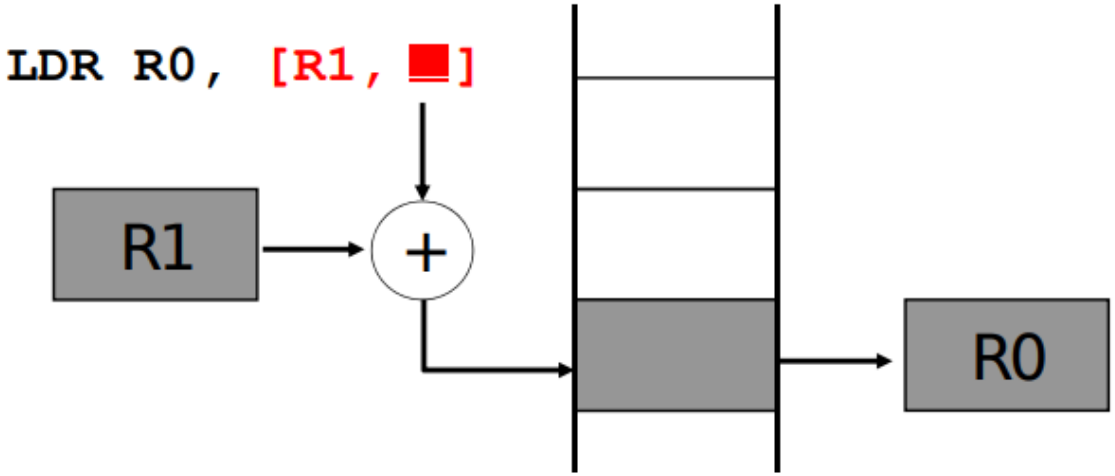
Index methods.

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

Pre-index addressing

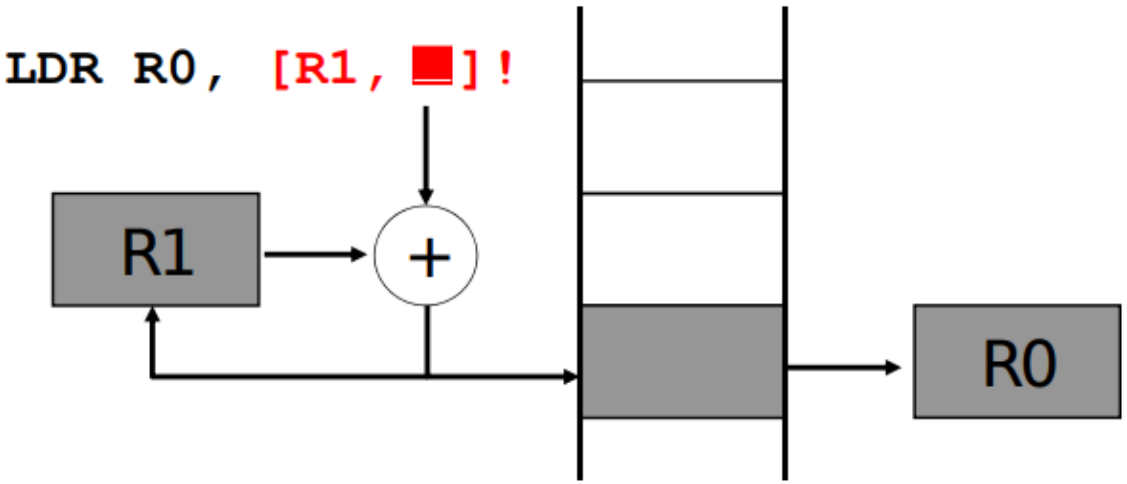
```
LDR  R0, [R1, #4]    @ R0=mem[R1+4]
                     @ R1 unchanged
```



Auto-indexing addressing

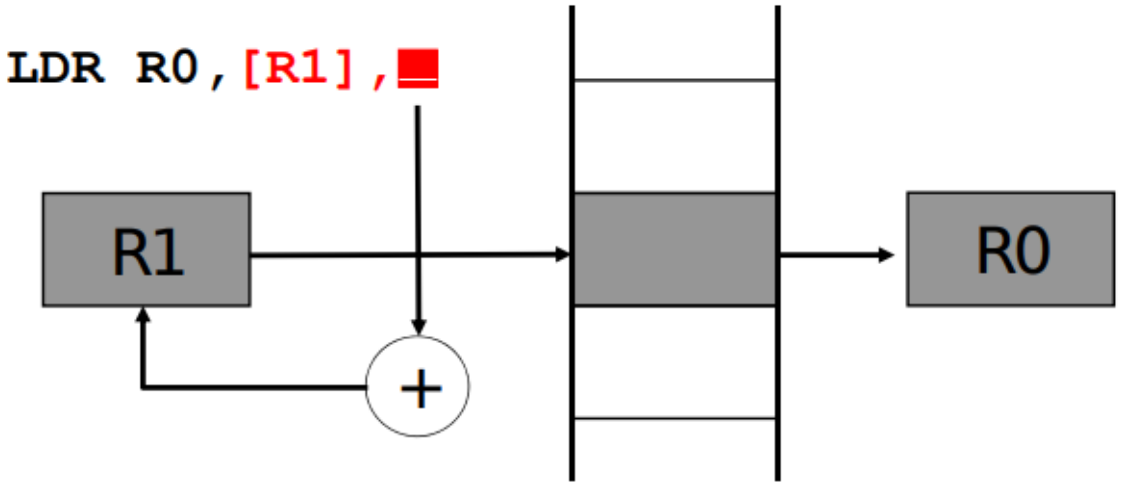
```
LDR  R0, [R1, #4]!   @ R0=mem[R1+4]
                     @ R1=R1+4
```

No extra time; Fast;



Post-index addressing

```
LDR  R0, R1, #4      @ R0=mem[R1]
                     @ R1=R1+4
```



Load-Store Instructions

- Exemplo 1:

```
PRE       r0 = 0x00000000
          r1 = 0x00090000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR     r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)   r0 = 0x02020202
          r1 = 0x00009004
```

Load-Store Instructions

- Exemplo 2:

```
PRE        r0 = 0x00000000
           r1 = 0x00090000
           mem32[0x00009000] = 0x01010101
           mem32[0x00009004] = 0x02020202

           LDR     r0, [r1, #4]
```

Preindexing:

```
POST(2)    r0 = 0x02020202
           r1 = 0x00009000
```

Load-Store Instructions

- Exemplo 3:

```
PRE      r0 = 0x00000000
         r1 = 0x00090000
         mem32[0x00009000] = 0x01010101
         mem32[0x00009004] = 0x02020202

         LDR     r0, [r1], #4
```

Postindexing:

```
POST(3)  r0 = 0x01010101
         r1 = 0x00009004
```

Addressing

Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

Addressing

Table 3.6 Examples of LDR instructions using different addressing modes.

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

Addressing

Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

Addressing

Variations of STRH instructions.

	Instruction	Result	$r1 +=$
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	0x4
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	r2
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	0x4
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	r2

Load-Store Instructions

- Multiple-register

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

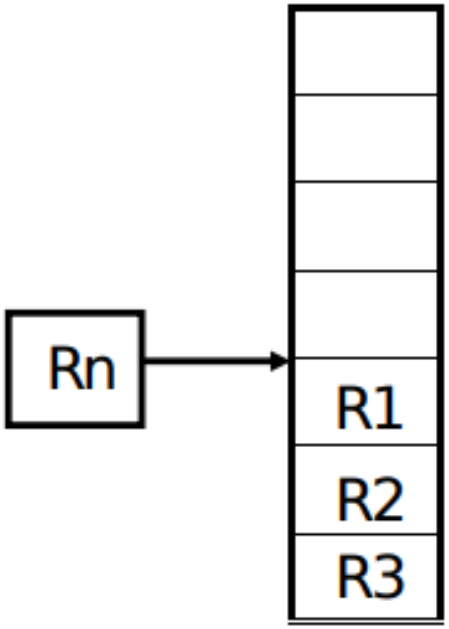
LDM	load multiple registers	$\{Rd\}^{*N}$ <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	$\{Rd\}^{*N}$ -> mem32[start address + 4*N] optional Rn updated

Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

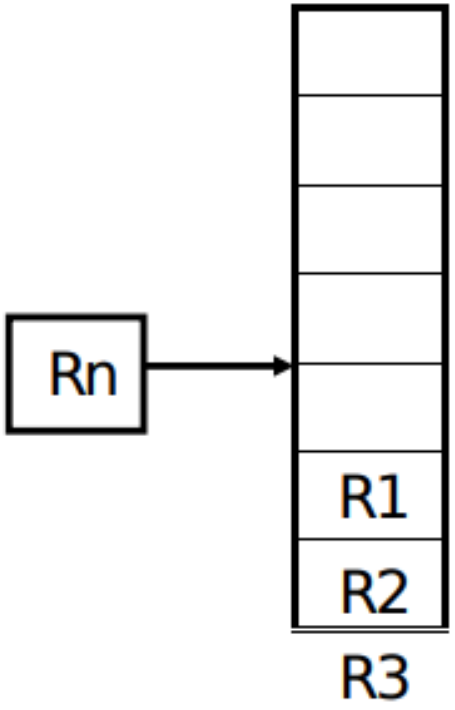
Load-Store: Multiple-register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



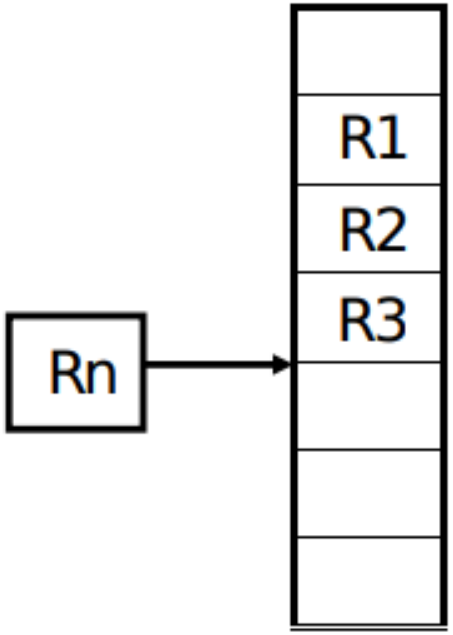
Load-Store: Multiple-register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



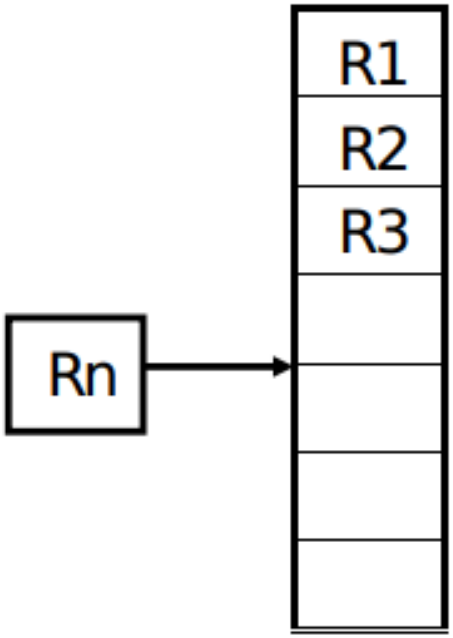
Load-Store: Multiple-register

```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



Load-Store: Multiple-register

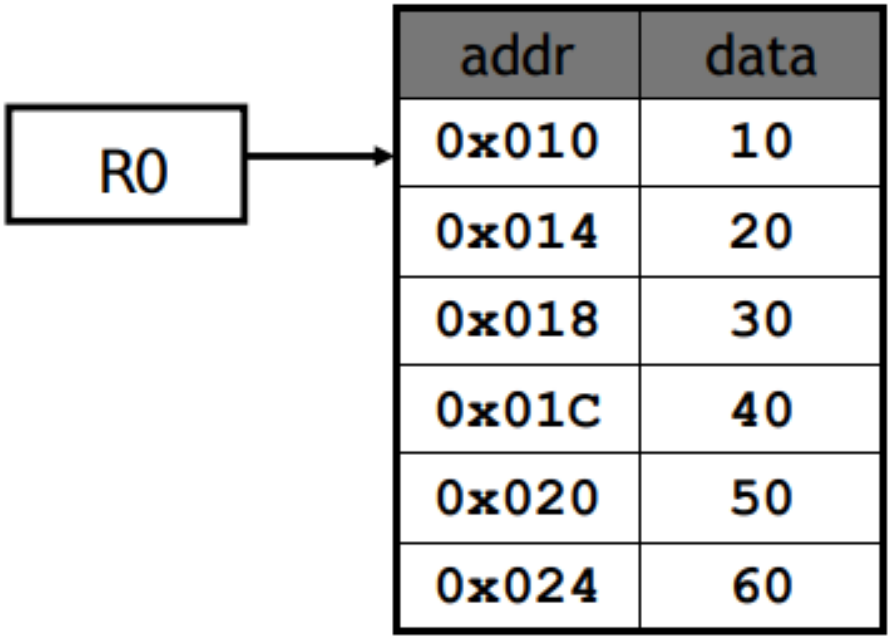
```
LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
  IA: addr:=addr+4
  DA: addr:=addr-4
<!>: Rn:=addr
```



Load-Store: Multiple-register

```
LDMIA R0, {R1,R2,R3}
 or
LDMIA R0, {R1-R3}
```

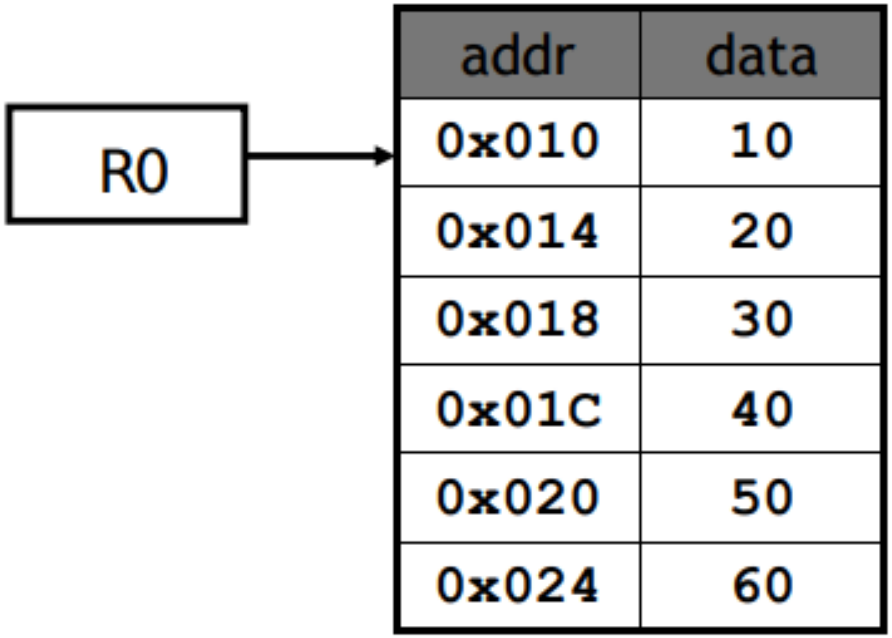
```
R1: 10
R2: 20
R3: 30
R0: 0x10
```



Load-Store: Multiple-register

```
LDMIA R0!, {R1,R2,R3}
```

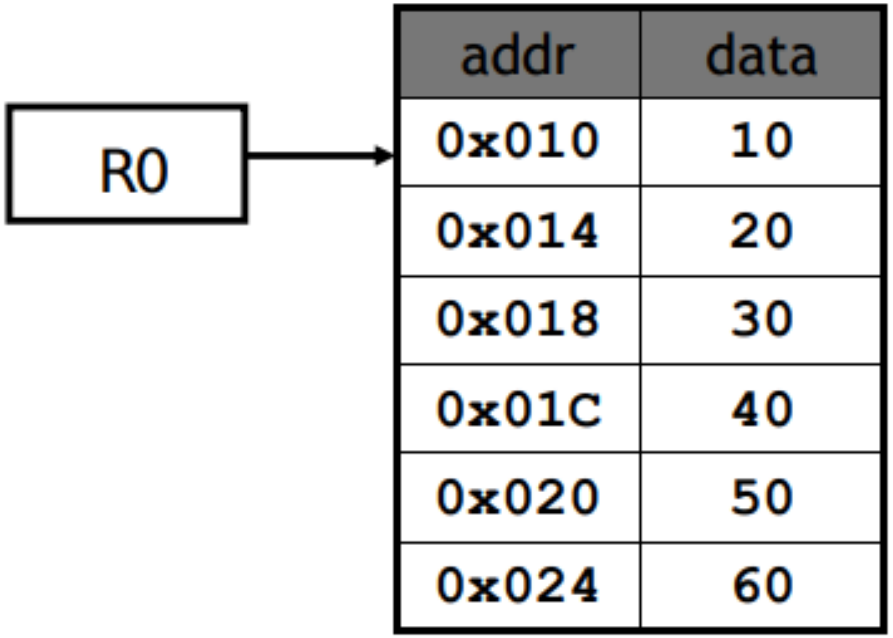
```
R1: 10
R2: 20
R3: 30
R0: 0x01C
```



Load-Store: Multiple-register

```
LDMIB R0!, {R1,R2,R3}
```

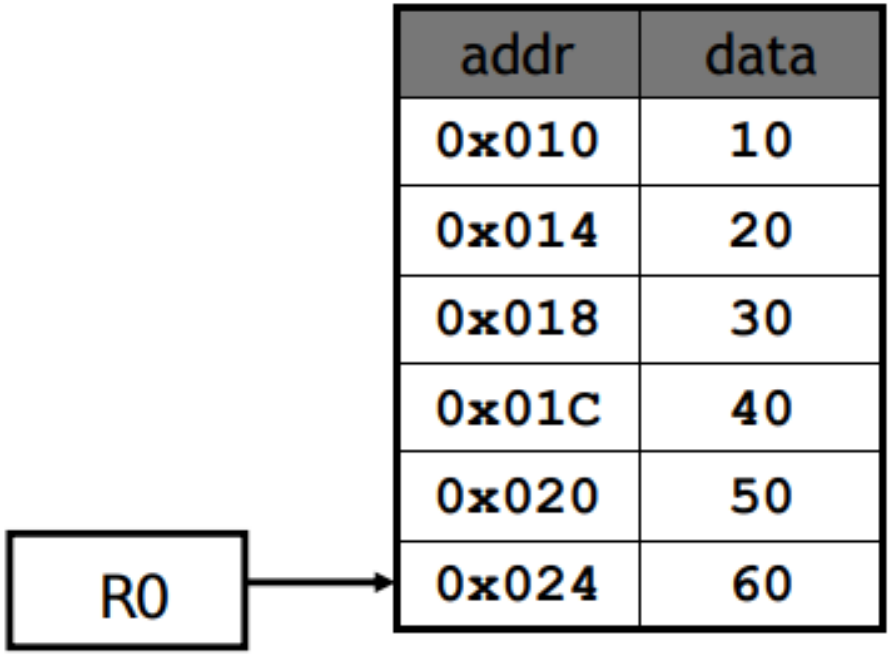
```
R1: 20
R2: 30
R3: 40
R0: 0x01C
```



Load-Store: Multiple-register

```
LDMDA R0!, {R1,R2,R3}
```

```
R1: 40
R2: 50
R3: 60
R0: 0x018
```

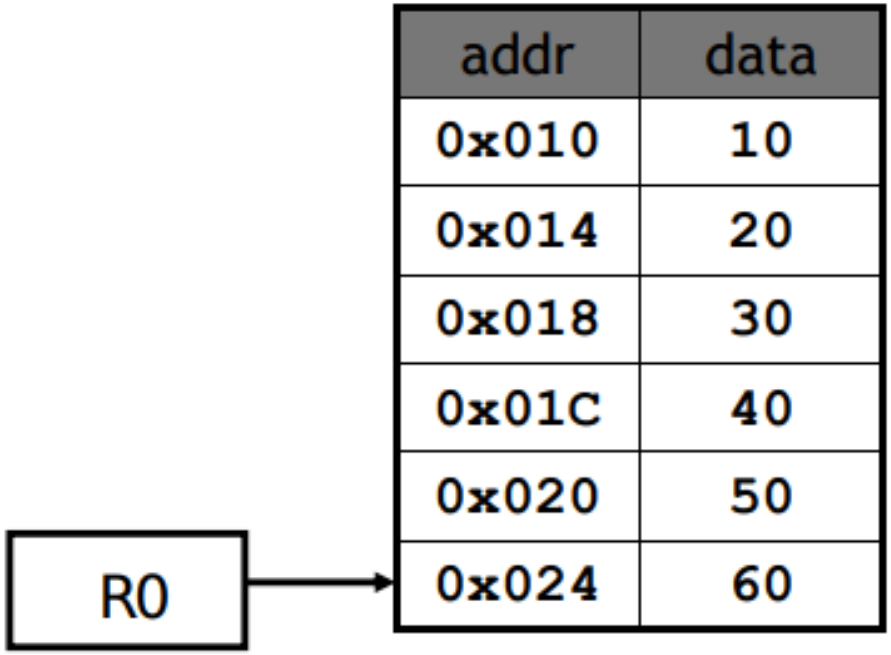


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Load-Store: Multiple-register

```
LDMDB R0!, {R1,R2,R3}
```

```
R1: 30
R2: 40
R3: 50
R0: 0x018
```



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Load-Store: Multiple-register

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

        LDMIA   r0!, {r1-r3}

POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```

Address pointer	Memory address	Data
	0x80020	0x00000005
	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
***r0* = 0x80010** →	0x80010	0x00000001
	0x8000c	0x00000000

Address pointer	Memory address	Data
	0x80020	0x00000005
***r0* = 0x8001c** →	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
	0x80010	0x00000001
	0x8000c	0x00000000

Load-Store: Multiple-register

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

        LDMIB    r0!, {r1-r3}

POST    r0 =
        r1 =
        r2 =
        r3 =
```

Address pointer	Memory address	Data
	0x80020	0x00000005
	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
r0 = 0x80010 →	0x80010	0x00000001
	0x8000c	0x00000000

	0x80020	0x00000005
r0 = 0x8001c →	0x8001c	0x00000004
	0x80018	0x00000003
	0x80014	0x00000002
	0x80010	0x00000001
	0x8000c	0x00000000

Load-Store: Multiple pairs

Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Load-Store: Multiple pairs

- Exemplo:

```
PRE      r0 = 0x00009000
         r1 = 0x00000009
         r2 = 0x00000008
         r3 = 0x00000007

         STMIB   r0!, {r1-r3}

         MOV     r1, #1
         MOV     r2, #2
         MOV     r3, #3

PRE(2)   r0 = 0x0000900c
         r1 = 0x00000001
         r2 = 0x00000002
         r3 = 0x00000003
```

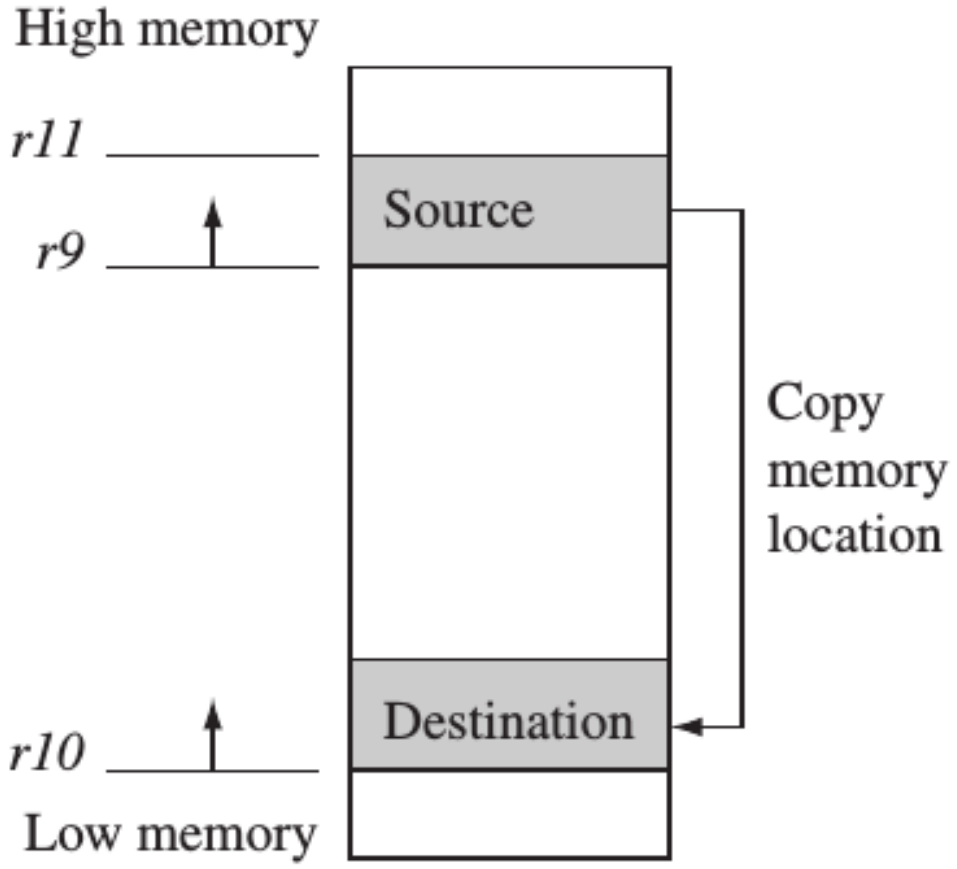
Load-Store: Multiple-register

Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

```
PRE      r0 = 0x00009000
         r1 = 0x00000009
         r2 = 0x00000008
         r3 = 0x00000007

         STMIB   r0!, {r1-r3}

         MOV   r1, #1
         MOV   r2, #2
         MOV   r3, #3

PRE(2)   r0 = 0x0000900c
         r1 = 0x00000001
         r2 = 0x00000002
         r3 = 0x00000003

         LDMDA r0!, {r1-r3}

POST     r0 = 0x00009000
         r1 = 0x00000009
         r2 = 0x00000008
         r3 = 0x00000007
```

Block Memory Copy



Block memory copy in the memory map.

Block Memory Copy

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source

loop
        ; load 32 bytes from source and update r9 pointer
        LDMIA   r9!, {r0-r7}

        ; store 32 bytes to destination and update r10 pointer
        STMIA   r10!, {r0-r7} ; and store them

        ; have we reached the end
        CMP    r9, r11
        BNE    loop
```

Stack operations

Addressing methods for stack operations.

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

- A →Pilha cresce de forma ascendente
- D →Pilha cresce de forma descendente
- F →sp aponta para o último endereço usado (útimo item da pilha)
- E →sp aponta para o primeiro endereço não usado (após o útimo item da pilha)
- FD → Padrão ATPCS para PUSH e POP

Stack operations – Full Descending

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014

        STMFD   sp!, {r1,r4}

POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x0008000c
```

PRE	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

Stack operations – Empty Descending

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1,r4}

POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080008
```

PRE	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Stack operations – Overflow

```
; check for stack overflow

SUB sp, sp, #size
CMP sp, r10
BLLO _stack_overflow ; condition
```

Swap Instruction

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Swap Instruction

- Exemplo:

```
PRE     mem32[0x9000] = 0x12345678
         r0 = 0x00000000
         r1 = 0x11112222
         r2 = 0x00009000

         SWP    r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
         r0 = 0x12345678
         r1 = 0x11112222
         r2 = 0x00009000
```

Software Interrupt Instruction

```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	$lr\_svc =$ address of instruction following the SWI $spsr\_svc = cpsr$ $pc =$ vectors + 0x8 $cpsr$ mode $= SVC$ $cpsr\ I = 1$ (mask IRQ interrupts)

```
SWI_Number = <SWI instruction> AND NOT(0xff000000)
```

Software Interrupt Instruction

- Exemplo:

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff; lr = r14
        r0 = 0x12

      0x00008000   SWI     0x123456

POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

Software Interrupt Instruction

```
SWI_handler
        ;
        ; Store registers r0-r12 and the link register
        ;
        STMFD   sp!, {r0-r12, lr}

        ; Read the SWI instruction
        LDR       r10, [lr, #-4]

        ; Mask off top 8 bits
        BIC       r10, r10, #0xff000000

        ; r10 - contains the SWI number
        BL        service_routine

        ; return from SWI handler
        LDMFD   sp!, {r0-r12, pc}^
```

Program Status Register Instructions

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

MRS	copy program status register to a general-purpose register	$Rd = psr$
MSR	move a general-purpose register to a program status register	$psr[field] = Rm$
MSR	move an immediate value to a program status register	$psr[field] = immediate$

Program Status Register Instructions

- Exemplo:

```
PRE     cpsr = nzcvqIFt_SVC

        MRS     r1, cpsr
        BIC     r1, r1, #0x80 ; 0b01000000
        MSR     cpsr_c, r1

POST    cpsr = nzcvqiFt_SVC
```

Mnemônicos de condição – Exemplo

```
If (r0 != 10)
{
r1 = r1 + r0 - r2
}
```

Mnemônicos de condição – Exemplo

```
If (r0 != 10)
{
r1 = r1 + r0 - r2
}
```

Sem condicional

```
CMP r0, #10
BEQ FUNC
ADD r1, r1, r0
SUB r1, r1, r2
FUNC: ...
```

Mnemônicos de condição – Exemplo

```
If (r0 != 10)
{
r1 = r1 + r0 - r2
}
```

Sem condicional

```
CMP r0, #10
BEQ FUNC
ADD r1, r1, r0
SUB r1, r1, r2
FUNC: ...
```

Com condicional

```
CMP r0, #10
ADDNE r1,r1,r0
SUBNE r1,r1,r2
```

Exercício

- 1) Escrever o código ao lado em assembly ARM, em duas versões:
  a) Sem mnemônicos de condição
  b) Com mnemônicos de condição

```
X = 0
A = 0
B = 1
while ( X < 5)
{
   if (X > 3) {
      B = X + A;
   }
   else if (X == 3){
      B = A = X;
   }
   else {
      A = X + B;
   }
   B += B*A;
   X = X + 1;
}
```

Trabalho

- Escrever um código em C, que, dado um arquivo de texto, contendo valores em hexadecimal, decodifique-o como sendo um código escrito com instruções THUMB.
  - Ver livro
    - ARM System Developer's Guide (Sloss), apêndice B.

QXD0133 - Arquitetura e Organização de Computadores II

Universidade Federal do Ceará - Campus Quixadá

Thiago Werlley
thiagowerlley@ufc.br

18 de outubro de 2025

Capítulo 3