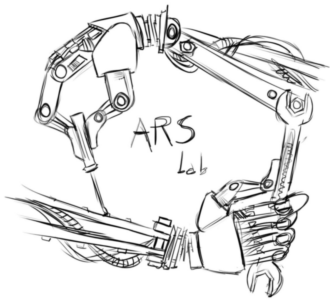
ARM Processor
Architettura e Instruction Set

Corrado Santoro

Dipartimento di Matematica e Informatica
santoro@dmi.unict.it



Corso di Architettura degli Elaboratori

Architettura del Processore ARM

Caratteristiche Generali

- 32-bit processor, RISC, 32-bit opcodes
- 13 Registri general purpose a 32 bit, **`R0–R12`**
- 1 Registro Stack Pointer, **`R13/SP`**
- 1 Link Register per le chiamate a subroutine, **`R14/LR`**
- 1 Program Counter, **`R15/PC`**
- 1 Processor Status Register, **`PSR`**, che memorizza i flag condizionali ed il modo di esecuzione
- Differenti **execution modes** (ARM, Thumb, User, System, Supervisor, IRQ, ...)

Architettura del Processore ARM

ARM Mode

- Modalità di “default”
- Istruction-set a 32 bit (ogni istruzione è lunga 32 bit)

Thumb Mode

- Modalità “ridotta”
- Istruction-set a 16 bit (ogni istruzione è lunga 16 bit)
- Esecuzione più veloce (soprattutto in ambienti con memoria lenta)
- Limitazioni su alcune istruzioni:
  - Istruzioni logico-aritmetiche limitate a 2 registri
  - LDR/STR limitate a R0-R7
  - LDM/STM e PUSH/POP limitate a R0-R7
  - Offset (su branch o istruzioni memory-relative) limitato

Istruzioni MOV

- Caricamento di un valore immediato su un registro
- Copia di un valore da un registro all'altro

`MOV Rn, #immediate16`	$Rn \leftarrow immediate16$
`MOV Rn, Rm`	$Rn \leftarrow [Rm]$

MOV-and-Negate (MVN)

- Copia di un valore da un registro all'altro con negazione bitwise

`MVN Rn, Rm`	$Rn \leftarrow not([Rm])$

Arithmetic-Logic Instruction

- Operazioni Logico-aritmetico tra 2 registri + valore, 3 registri, 1 registro + valore, 2 registri

`op Rd, Rn, #imm12`	$Rd \leftarrow [Rn]\ op\ \#imm12$
`op Rd, Rn, Rm`	$Rd \leftarrow [Rn]\ op\ [Rm]$
`op Rd, #imm12`	$Rd \leftarrow [Rd]\ op\ \#imm12$
`op Rd, Rn`	$Rd \leftarrow [Rd]\ op\ [Rn]$

Arithmetic Instructions

`ADD`	Somma
`ADC`	Somma con carry (riporto)
`SUB`	Sottrazione
`SBC`	Sottrazione con carry
`RSB`	Reverse Subtract
`RSC`	Reverse Subtract con carry

Logic Instructions

AND	And logico bitwise
ORR	Or logico bitwise
EOR	Exclusive-Or logico bitwise
BIC	operand1 AND NOT operand2 (bit clear)
ORN	operand1 OR NOT operand2

Confronto (CMP) e Program Status Register

- Confronta due registri, o un registro con un valore immediato effettuando una sottrazione
- Aggiorna i **flag** del processore (registro PSR) sulla base del risultato

`CMP Rn, #immediate16`	[*Rn*] *vs.* *immediate*16
`CMP Rn, Rm`	[*Rn*] *vs.* [*Rm*]

Program Status Register (PSR)

- E' un registro (bit-mapped) i cui bit rappresentano:
  - I **flags** relativi al risultato dell'ultima operazione di CMP
  - L'execution mode (User, System, Supervisor, ...)
  - I flag di interrupt
  - Il thumb mode

31			28	.....	7	6	5	4 ... 0
N	Z	C	V		I	F	T	Mode

- **N** Negative result
- **Z** Zero result
- **C** Carry set
- **V** Overflow

Salto condizionato - Bcond

```
B{cond} target
```

- L'istruzione effettua un salto alla destinazione (relativa al PC) specificata se la condizione specificata è vera

{**cond**}	**Flags**	**Significato**
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned $>=$)
CC or LO	C clear	Lower (unsigned $<$)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned $>$)
LS	C clear or Z set	Lower or same (unsigned $<=$)
GE	N and V the same	Signed $>=$
LT	N and V differ	Signed $<$
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed $<=$

Salto incondizionato

- L'istruzione effettua un salto alla destinazione specificata

`B target`

Salto incondizionato con collegamento (branch with link)

- L'istruzione effettua un salto alla destinazione specificata copiando preventivamente su Link Register (LR/R14) il valore del Program Counter (PC/R15)

`BL target`	$LR \leftarrow [PC]$, jump to target

Accesso alla memoria: LOAD

- L'istruzione preleva una word dall'indirizzo di memoria indicato e la trasferisce ad un registro di destinazione
- L'indirizzo è può essere indicato da un registro (puntatore) più un eventuale offset *immediato* o *registro*

`LDR Rd, [Rs]`	$Rd \leftarrow [[Rs]]$, register indirect
`LDR Rd, [Rs, #offset]`	$Rd \leftarrow [[Rs] + offset]$, register indirect + immediate offset
`LDR Rd, [Rs, #offset]!`	$Rs \leftarrow [Rs] + offset$, $Rd \leftarrow [[Rs]]$, pre-indexed
`LDR Rd, [Rs], #offset`	$Rd \leftarrow [[Rs]]$, $Rs \leftarrow [Rs] + offset$, post-indexed

Accesso alla memoria: LOAD

- L'istruzione preleva una word dall'indirizzo di memoria indicato e la trasferisce ad un registro di destinazione
- L'indirizzo è può essere indicato da un registro (puntatore) più un eventuale offset *immediato* o *registro*

`LDR Rd, [Rs, +/- Ro]`	$Rd \leftarrow [[Rs] \pm [Ro]]$, register indirect + register offset
`LDR Rd, [Rs, +/- Ro]!`	$Rs \leftarrow [Rs] \pm [Ro], Rd \leftarrow [[Rs]]$, pre-indexed
`LDR Rd, [Rs], +/-Ro`	$Rd \leftarrow [[Rs]], Rs \leftarrow [Rs] \pm [Ro]$, post-indexed

Accesso alla memoria: STORE

- L'istruzione deposita una word, contenuta in un registro sorgente, all'indirizzo di memoria indicato
- L'indirizzo è può essere indicato da un registro (puntatore) più un eventuale offset *immediato* o *registro*

`STR Rs, [Rd]`	$[[Rd]] \leftarrow [Rs]$, register indirect
`STR Rs, [Rd, #offset]`	$[[Rd] + offset] \leftarrow [Rs]$, register indirect + immediate offset
`STR Rs, [Rd, #offset]!`	$Rd \leftarrow [Rd] + offset, [[Rd]] \leftarrow [Rs]$, pre-indexed
`STR Rs, [Rd], #offset`	$[[Rd]] \leftarrow [Rs], Rd \leftarrow [Rd] + offset$, post-indexed

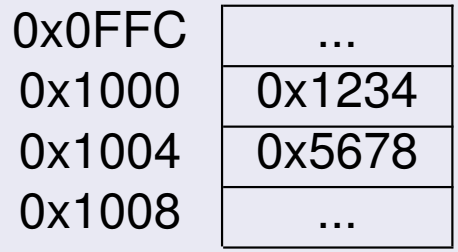
Accesso alla memoria: LOAD

- L'istruzione deposita una word, contenuta in un registro sorgente, all'indirizzo di memoria indicato
- L'indirizzo è può essere indicato da un registro (puntatore) più un eventuale offset *immediato* o *registro*

`STR Rs, [Rd, +/- Ro]`	$[[Rd] \pm [Ro]] \leftarrow [Rs]$, register indirect + register offset
`STR Rs, [Rd, +/- Ro]!`	$Rd \leftarrow [Rd] \pm [Ro], [[Rd]] \leftarrow [Rs]$, pre-indexed
`STR Rs, [Rd], +/-Ro`	$[[Rd]] \leftarrow [Rs], Rd \leftarrow [Rd] \pm [Ro]$, post-indexed

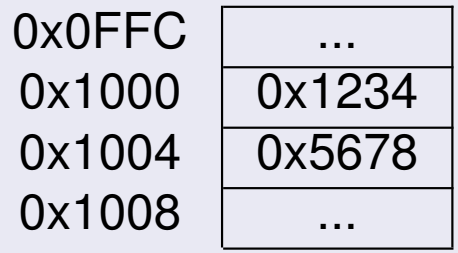
Modalità di Indirizzamento

Register-indexed




```
MOV R1,#0x1000
LDR R0, [R1]      ;; R0 ← 0x1234
                  ;; R1 unchanged
```

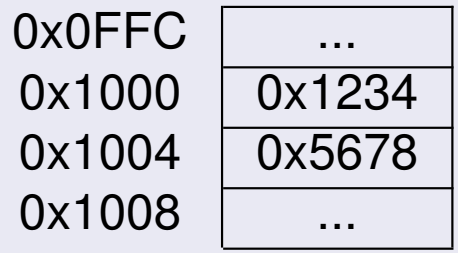
Register + offset




```
MOV R1,#0x1000
LDR R0, [R1, #4]  ;; R0 ← 0x5678
                  ;; R1 unchanged
```

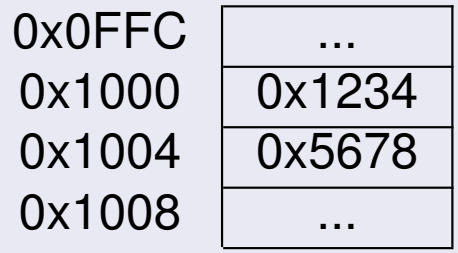
Modalità di Indirizzamento

Pre-indexed




```
MOV R1,#0x1000
LDR R0, [R1, #4]!   ;; R1 ← R1 + 4
                    ;; R1 ← 0x1004
                    ;; R0 ← 0x5678
```

Post-indexed




```
MOV R1,#0x1000
LDR R0, [R1], #4    ;; R0 ← 0x1234
                    ;; R1 ← 0x1004
```

LOAD/STORE Multiple Registers

Istruzione **LDM/STM**

- Le istruzioni LDM/STM permetteno di leggere/memorizzare un *insieme di registri* in una zona di memoria attraverso un'unica operazione
- Il registro puntatore viene aggiornato opportunamente durante l'operazione
- L'accesso alla memoria può avvenire in **"senso crescente"** (dal punto di vista degli indirizzi), o in **"senso decrescente"**

Istruzione **LDM/STM**: Varianti

- **`LDMIA/STMIA`**: Load/Store multiple-registers, **increment after**
  - Il registro puntatore viene incrementato **dopo** l'operazione di load/store
- **`LDMDB/STMDB`**: Load/Store multiple-registers, **decrement before**
  - Il registro puntatore viene decrementato **prima** dell'operazione di load/store

LOAD Multiple Registers

Sintassi **LDM**

`LDMIA Rd, {Rx, Ry, ..., Rz}`	Load Multiple Registers Increment After Rd = registro puntatore $Rx, Ry, .., Rz$ = registri da leggere
`LDMIA Rd!, {Rx, Ry, ..., Rz}`	Come la precedente, ma il registro puntatore è aggiornato a fine operazione
`LDMDB Rd, {Rx, Ry, ..., Rz}`	Load Multiple Registers Decrement Before Rd = registro puntatore $Rx, Ry, .., Rz$ = registri da leggere
`LDMDB Rd!, {Rx, Ry, ..., Rz}`	Come la precedente, ma il registro puntatore è aggiornato a fine operazione

STORE Multiple Registers

Sintassi **STM**

`STMIA Rd, {Rx, Ry, ..., Rz}`	Store Multiple Registers Increment After Rd = registro puntatore $Rx, Ry, .., Rz$ = registri da leggere
`STMIA Rd!, {Rx, Ry, ..., Rz}`	Come la precedente, ma il registro puntatore è aggiornato a fine operazione
`STMDB Rd, {Rx, Ry, ..., Rz}`	Store Multiple Registers Decrement Before Rd = registro puntatore $Rx, Ry, .., Rz$ = registri da leggere
`STMDB Rd!, {Rx, Ry, ..., Rz}`	Come la precedente, ma il registro puntatore è aggiornato a fine operazione

Esempio di LDM - Increment After

LDMIA

0x0FFC	...
0x1000	0x1234
0x1004	0x5678
0x1008	0x9abc
0x100c	...

```
MOV R1,#0x1000
LDMIA R1, {R4, R6, R7}

;; R1 ← unchanged
;; R4 ← 0x1234
;; R6 ← 0x5678
;; R7 ← 0x9abc
```

LDMIA!

0x0FFC	...
0x1000	0x1234
0x1004	0x5678
0x1008	0x9abc
0x100c	...

```
MOV R1,#0x1000
LDMIA R1!, {R4, R6, R7}

;; R4 ← 0x1234
;; R6 ← 0x5678
;; R7 ← 0x9abc
;; R1 ← 0x100c
```

Esempio di LDM - Decrement Before

LDMDB

0x0FFC	...
0x1000	0x1234
0x1004	0x5678
0x1008	0x9abc
0x100c	...

```
MOV R1,#0x100c
LDMDB R1, {R4, R6, R7}

;; R1 ← unchanged
;; R4 ← 0x9abc
;; R6 ← 0x5678
;; R7 ← 0x1234
```

LDMDB!

0x0FFC	...
0x1000	0x1234
0x1004	0x5678
0x1008	0x9abc
0x100c	...

```
MOV R1,#0x100c
LDMDB R1!, {R4, R6, R7}

;; R4 ← 0x9abc
;; R6 ← 0x5678
;; R7 ← 0x1234
;; R1 ← 0x1000
```

Esempio di STM - Increment After

STMIA

0x0FFC	...
0x1000	0x0003
0x1004	0x0002
0x1008	0x0001
0x100c	...

```
MOV R1,#0x1000
STMIA R1, {R4, R6, R7}

;; R1 ← unchanged
;; R4 = 0x0001
;; R6 = 0x0002
;; R7 = 0x0003
```

STMIA!

0x0FFC	...
0x1000	0x0003
0x1004	0x0002
0x1008	0x0001
0x100c	...

```
MOV R1,#0x1000
STMIA R1!, {R4, R6, R7}

;; R4 = 0x0001
;; R6 = 0x0002
;; R7 = 0x0003
;; R1 ← 0x100c
```

Esempio di STM - Decrement Before

STMDB

0x0FFC	...
0x1000	0x0001
0x1004	0x0002
0x1008	0x0003
0x100c	...

```
MOV R1,#0x100c
STMDB R1, {R4, R6, R7}

;; R1 ← unchanged
;; R4 = 0x0001
;; R6 = 0x0002
;; R7 = 0x0003
```

STMDB!

0x0FFC	...
0x1000	0x0001
0x1004	0x0002
0x1008	0x0003
0x100c	...

```
MOV R1,#0x100c
STMDB R1!, {R4, R6, R7}

;; R4 = 0x0001
;; R6 = 0x0002
;; R7 = 0x0003
;; R1 ← 0x1000
```

Pseudo istruzione LDR

- La codifica di una operazione di assegnazione di un valore immediato ad un registro non è sempre possibile, soprattutto quando l'operando è un valore "full 32-bit". Esempio:

```
MOV R0,#0xffff1234
```

- Poichè l'intera istruzione deve essere codificata in 32 bit, non è possibile includere sia l'operando (che è già a 32 bit), sia i bit che rappresentano l'istruzione stessa
- La soluzione è usare la pseudo istruzione LDR:

`LDR Rn,=value`	$Rn \leftarrow value$

Pseudo istruzione LDR

`LDR Rn,=value`	$Rn \leftarrow value$

- L'assemblatore riconosce la pseudo-istruzione
- Se l'istruzione è codificabile in 32 bit (16 bit in Thumb Mode) essa viene sostituita da una **MOV**
- In caso contrario, l'operando viene posto in memoria (programma) e l'istruzione è sostituita da una **LDR** PC-relative

```
0x100:   LDR R1,[pc, #0x20]
0x104:   ...
   ...   ...
0x124:   .word value
```

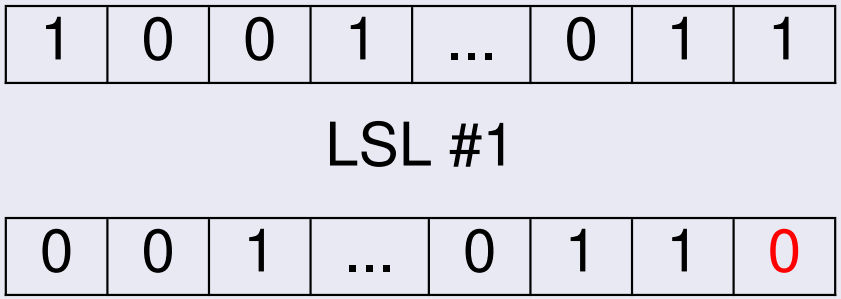
Shifting Instruction

`op Rd, Rn, #sh`	$Rd \leftarrow [Rn]\ op\ \#sh$
`op Rd, Rn, Rs`	$Rd \leftarrow [Rn]\ op\ [Rs]$

`#sh, Rs` = number of shifts

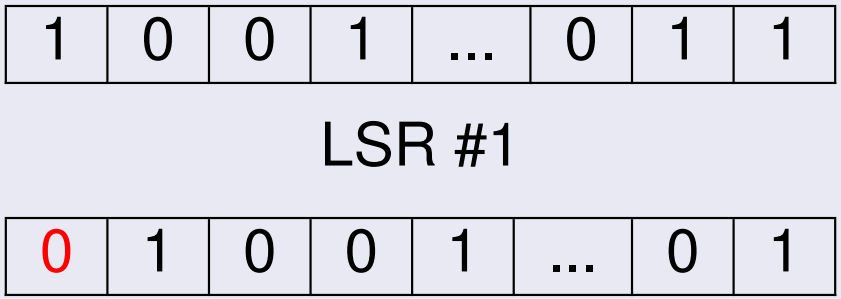
op	
LSL	Logic Shift Left
LSR	Logic Shift Right
ASR	Arithmetic Shift Right
ROR	Rotate Right

LSL = Logic Shift Left



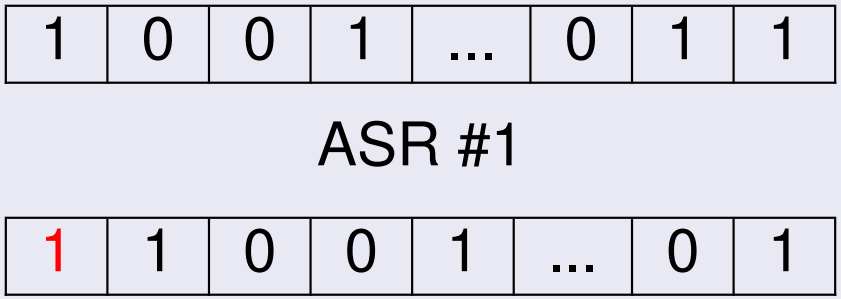
L'istruzione LSL moltiplica il numero intero per 2

LSR = Logic Shift Right



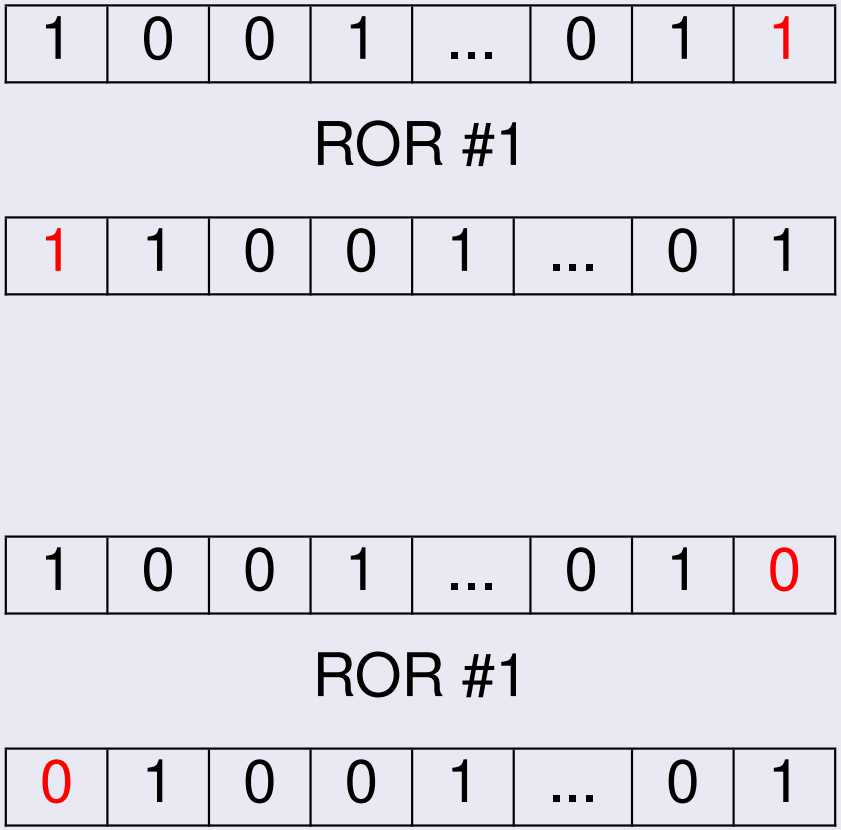
L'istruzione LSR divide il numero intero per 2

ASR = Arithmetic Shift Right



L'istruzione ASR equivale a LSR ma propaga anche il bit di segno

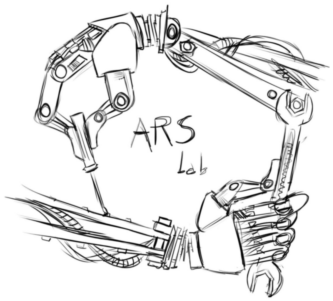
ROR = Rotate Right



ARM Processor
Architettura e Instruction Set

Corrado Santoro

Dipartimento di Matematica e Informatica
santoro@dmi.unict.it



Corso di Architettura degli Elaboratori