Sommario

Teorema di Böhm-Jacopini

Il teorema di Böhm-Jacopini, enunciato nel 1966 da due informatici italiani dai quale prende il nome, afferma che:
«qualunque algoritmo può essere implementato utilizzando tre sole strutture, **la sequenza, la selezione e l'iterazione**, che possono essere tra loro innestati fino a giungere ad un qualsivoglia livello di profondità finito (come le scatole cinesi)».
Questo serve per evitare la **programmazione a spaghetti (spaghetti code)** che è un termine dispregiativo per il codice sorgente di quei programmi per computer che hanno una struttura di controllo del flusso complessa e/o incomprensibile, con uso esagerato ed errato di go to, eccezioni, thread e altri costrutti di branching (diramazione del controllo) non strutturati.

Diagrammi Nassi-Shneiderman

I Diagrammi di Nassi-Shneiderman, anche detti **DNS,** sono strumenti semplici che si usano per scrivere algoritmi intrinsecamente strutturati e complessi.
Si collegano al TH Bohm-Jacopini.,associando infatti questo teorema al formalismo ideato da Isaac Nassi e Ben Shneiderman, si ottengono appunto i DNS
I concetti di sequenza, iterazione, selezione possono essere espressi, all'interno del diagramma, in blocchi che possono essere innestati e composti a piacere fino a giungere ad un qualsiasi livello di profondità.

Sequenza

Si esprime il concetto che ogni procedimento risolutivo si può comporre attraverso un insieme finito di passaggi, o di blocchi, che verranno eseguiti nell'ordine dall'alto verso il basso, senza la possibilità di saltare la valutazione di un passaggio intermedio.

Selezione

é il concetto attraverso il quale il percorso del flusso di esecuzione può prendere differenti cammini a seconda della condizione espressa in testa al blocco.
Può essere:

1. *Singola :* se la condizione è verificata viene eseguito un blocco, altrimenti si esce e si prosegue il flusso di esecuzione(**if-then** o **if-else**)
2. *Binaria :* se la condizione è verificata viene eseguito il blocco **then,** altrimenti il blocco **else(if-then-else)**
3. *Multipla* : controlla il valore di una variabile all'interno di un intervallo di valori predefiniti. Quando la variabile corrisponde a uno dei valori della selezione, viene seguito il blocco ad esso associato,altrimenti viene eseguito il blocco di default(**switch**)

Iterazione

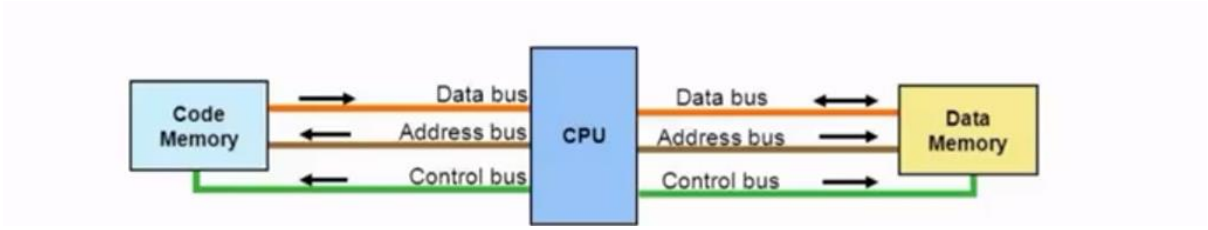
Si esprime la possibilità di ripetere l'esecuzione di un blocco interno fino a quando non si verificherà la condizione di uscita, per questa ragione è di uso comune denominarlo anche "ciclo". Esistono tre tipologie: **for, while, do-while**
Quando si sa a priori il numero di ripetizioni da eseguire, si usa il **for**
Quando c'è la possibilità che il blocco non venga mai eseguito, o l'iterazione è funzione di un evento non noto, si usa il **while**
Quando l'iterazione dipende da un evento non noto, ma il blocco deve essere eseguito almeno una volta, si usa il **do-while**

ARM

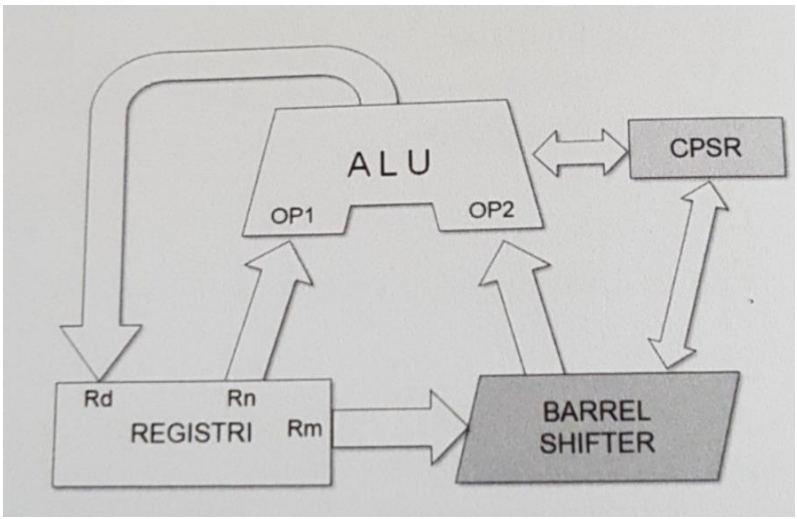
cos'è ARM?

- l'architettura ARM definisce una famiglia di microprocessori di tipo RISC (Poche e semplici istruzioni per massimizzare il throughput delle istruzioni).
- dal punto di vista circuitale è estremamente semplice rispetto a un CPU tradizionale e quindi richiede un numero minore di transistor e di silicio questo porta ad essere più economico sul mercato ed ha un basso consumo energetico rispetto alle sue prestazioni. (questa architetture si trova diffusa su cellulari, tablet, console, portatili, elettrodomestici e televisori)
- inoltre deve eseguire l'elaborazione dei dati direttamente nel processore senza intervento della memoria e quindi accede in memoria solo per operazioni di lettura e scrittura.

Struttura ARM

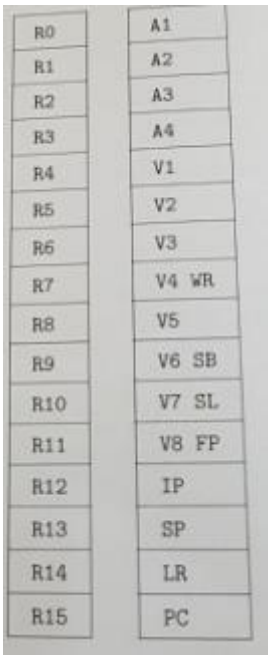
- è una architettura RISC a 32 bit con uno spazio di indirizzamento di 32 bit e la memoria è indirizzabile al singolo byte, a 16 bit (halfword) o 32 bit (word) dove il ARM può indirizzare fino a 2^{32} bit
- 

- l'organizzazione interna di ARM prevede l'utilizzo di 3 registri:
    1. **RD:** registro destinazione dove verrà scritto il risultato
    2. **RN:** registro sorgente che contiene il prImo operando ed è utilizzato direttamente
    3. **RM:** registro sorgente che contiene il secondo operando ed è utilizzato dopo il passaggio nel **barrel shift** (è un circuito che può shiftare una word di numero specificato di bit in un solo ciclo di clock)

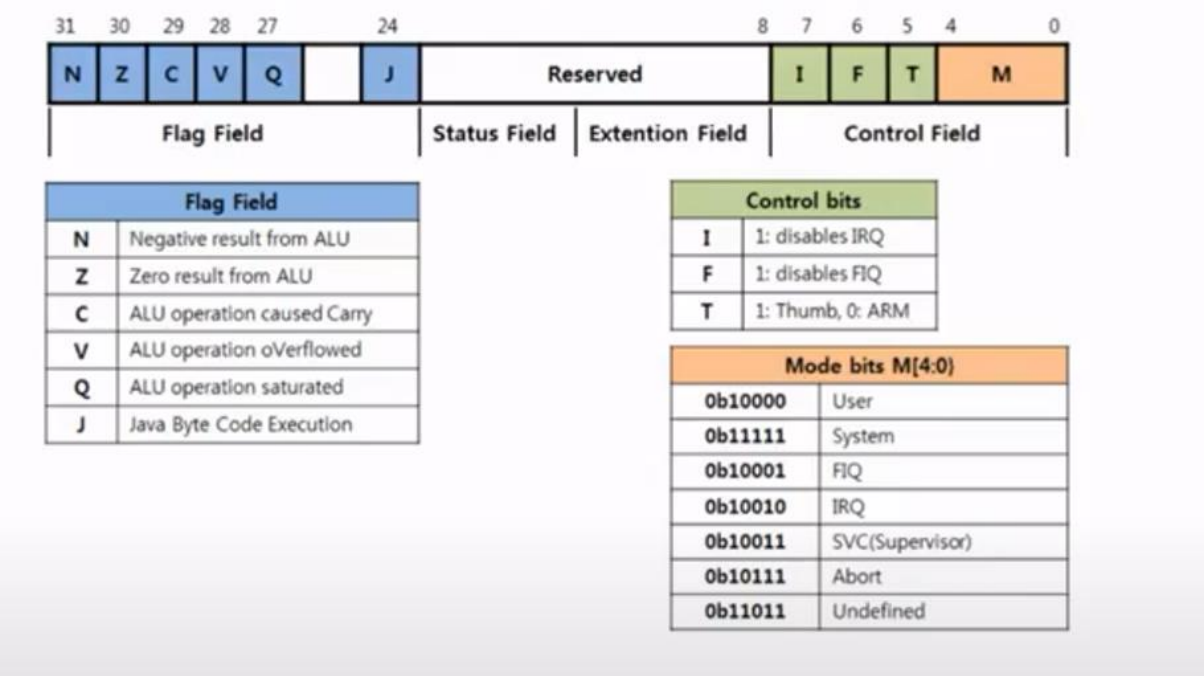


I registri ARM

- l'architettura ARM prevede la seguente organizzazione:
    - 16 registri denominati da **R0 a R15**
        1. **R0 - R12** di uso generale utilizzabili dal programmatore per realizzare applicazioni
        2. **R13** è lo stack pointer (**SP**) contiene l'indirizzo della locazione di memoria occupata dal inizio dello stack.
        3. **R14** è il subroutine link register (**LR**) cioè quel registro utilizzato per ripristinare il **PC** alla istruzione successiva alla chiamata ad una subroutine, quando quest'ultima ha terminato la sua esecuzione.
        4. **R15** è il program counter (**PC**) conserva l'indirizzo di memoria della successiva istruzione da eseguire.



- un registro di stato denominato current program status register (**CPSR**).

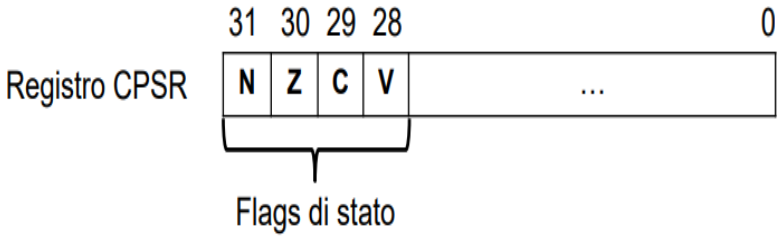


1. il quale tiene traccia dello stato del programma. Molto utili sono i 5 bit principali di stato (detti flags di stato) i quali esprimono le seguenti condizioni aritmetiche:
    A. Bit N negativo
    B. Bit Z zero;
    C. Bit C carry/overflow riporto
    D. Bit V overflow.
    E. Bit Q saturazione - impostato a 1 quando nel risultato è stato inserito un valore limite (estremo superiore o inferiore).

Sono presenti inoltre altri flag di stato cioè:

    F. Bit E Endian - se E=0 little-endian altrimenti se E=1 big-endian (modo di indirizzamento dei byte).
    G. Bit Do Not Modify, per futuri utilizzi.
    H. Bit Greater Than or Equal, vengono usati come flag di di maggiore di o uguale (GE).
    I. Bit Imprecise Data Abort enable/disable, abilita/disabilita ogni richiesta di abort dati imprecisi.
    J. Bit Interrupt enable/disable, abilita/disabilita ogni richiesta di interrupt a bassa priorità IRQ.
    K. Bit Fast Interrupt enable/disable, abilita/disabilita ogni richiesta di interrupt ad alta priorità FIQ.
    L. Bit di stato Thumb, il processore entra nello stato thumb e le istruzioni vengono allineate a 16 bit e il valore del PC è memorizzato nei 31 bit più significativi.
    M. Bit di stato Jazelle, il processore entra nello stato jazelle e permette il riconoscimento del byte code (Java) utilizzando un set di istruzioni a 8 bit, mantiene la modalità di accesso alla memoria a 32 bit quindi è in grado di leggere 4 istruzioni alla volta.
    N. Bit M0-M4 definiscono la mod di funzionamento del processore ARM:
        a. Utente - mod standard

b. Supervisor - si attiva al reset o quando il processore riceve un'istruzione di interrupt software (SWI)
c. Fast Interrupt Request - si attiva quando il bit FIQ viene asserito
d. Interrupt Request si attiva quando il bit IRQ viene asserito
e. Abort - si attiva in risposta ad una eccezione di abort
f. Undefined - si attiva quando si verifica un'eccezione dopo l'esecuzione di un'istruzione non supportata
g. System - Mod privilegiata

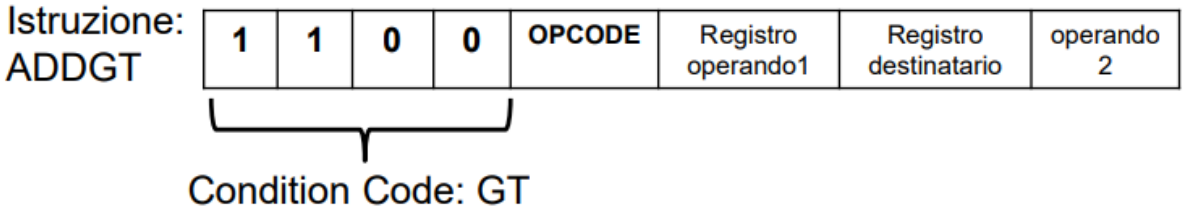


- 20 registri non accessibili nel modo utente/sistema e organizzati come segue:
    1. 5 registri da **R8_FIQ** a **R12_FIQ** attivi solo nella modalità di risposta ad una eccezione causata da un interrupt ad alta priorità
    2. 10 registri (**SP_FIQ, LR_FIQ, SP_IRQ, LR_IRQ, SP_SVC, LR_SVC, SP_ABT, LR_ABT, SP_UND** e **LR_UND**) attivi soltanto due per volta (**stack pointer** e **linker registri)** nella modalità di risposta ad un eccezione
    3. 5 registri (**SPSR_FIQ, SPSR_SVC, SPSR_ABT, SPSR_IRQ, SPSR_UND**) attivo uno per volta nella modalità di risposta ad un eccezione e che permette di salvare il contenuto del registro di stato corrente **CPSR** nel corrispondente registro sved program status register (**SPSR**)

Tabella delle condizioni aritmetico-logiche

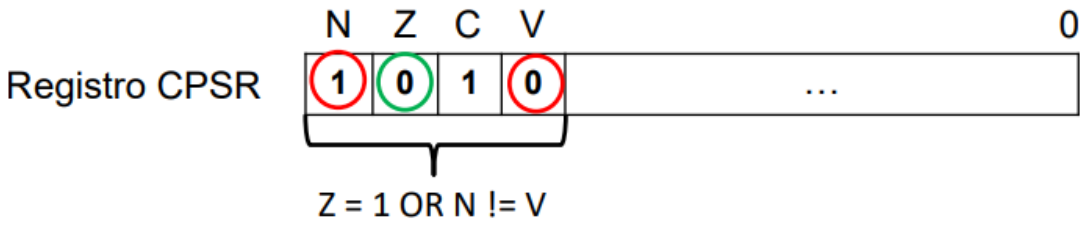
Suffisso	Descrizione	Flag testato	Condition Code
EQ	Equal (==)	Z = 1	0000
NE	Not Equal (!=)	Z = 0	0001
CS / HS	Unsigned maggiore uguale (>=)	C = 1	0010
CC / LO	Unisgned minore stretto (<)	C = 0	0011
MI	Negativo (< 0)	N = 1	0100
PL	Positivo o 0 (>= 0)	N = 0	0101
VS	Overflow	V = 1	0110
VC	No overflow	V = 0	0111
HI	Unsigned maggiore stretto (>)	C = 1 AND Z = 0	1000
LS	Unsigned minore uguale (<=)	C = 0 OR Z = 1	1001
GE	Maggiore uguale (>=)	N = V	1010
LT	Minore stretto (<)	N != V	1011
GT	Maggiore stretto (>)	Z = 0 AND N = V	1100
LE	Minore uguale (<=)	Z = 1 OR N != V	1101
AL	Always	----	1110

Esempio registro di stato più tabella delle condizioni aritmetiche logiche



l'istruzione **ADD** verrà eseguita solo se la condizione GT (maggiore stretto) sarà verificata. La condizione **GT** è identificata dal **codice 1100** nella sezione del registro chiamata **condition code**.

Esempio:
In questo esempio la condizione **LE** è verificata in quanto nel **registro di stato CPSR** i flags NZCV rispettano almeno una delle condizioni in tabella (Z = 1 OR N != V).



1.

Eccezioni ARM

Priorità	Livello	Eccezione	Causa
Max	1	Reset	Reset hardware o software
	2	Abort	Abort dati durante l'accesso in memoria
	3	FIQ	Interrupt ad alta priorità, *Fast Interrupt reQuest*
	4	IRQ	Interrupt a bassa priorità, *Interrupt ReQuest*
	5	Abort	Prefetch Abort durante la fase di fetch dell'istruzione
Min	6	Undefined	Istruzione non definita e software interrupt (`SWI`)

Particolarità ARM

- non esistono istruzioni di shift o rotazione di bit se non innestate all'interno di altre istruzioni di elaborazione (aritmetica, logica…) e possono essere applicate ad un solo operando
- riesce ad applicare una operazione aritmetica o logica e lo shift o di rotazione in una sola istruzione
- non esiste una istruzione per eseguire l'operazione di divisione questo viene delegato a un coprocessore che avendo un ambito di applicazione più specifico
- le istruzioni normalmente non scrivono il registro **CPSR** ma è possibile farlo tramite un suffisso **S** (forza la scrittura nel registro CPSR) aggiungendolo al codice mnemonico dell' istruzione.

Big endian e Little Endian

sono 2 tipi di ordinamento dei bit e dei byte.

1. Big Endian imposta i bit della word partendo da quello più significativo a quello meno significativo
2. Little Endian imposta i bit della word partendo da quello meno significativo a quello più significativo

Modalità di indirizzamento

Modalità base

- **a registro -** l'operando è contenuto nel registro
- **a registro con shift -** l'operando è contenuto nel registro, ma subirà un operazione di shift o rotazione
- **immediato -** operando contenuto nella codifica dell'istruzione
  sintassi:
  [<Rn>]
- **indiretto -**
  sintassi:

<Rn>, {+|-}[<Rm>]
esempio:
MOV R0, [R1]

Indirizzamento per le istruzioni load/store:

- **con offset immediato**
  l'offset è un valore senza segno specificato nell istruzione
  sintassi:
  [<Rn>, #{+|-}<$offset_{12}$>]
  esempio:
  MOV R0, #0x00000208
- **con offset a registro**
  l'offset è un valore contenuto in un registro generico (con esclusione del PC o R15)
  esempio di sintassi:
  [<Rn>, {+|-}<Rm>]
  esempio:
  MOV R0, R1
- **con offset a registro scalato**
  l'offset è un valore calcolato per traslazione o rotazione del contenuto in un registro generico (con esclusione del PC)
  sintassi:
  [<Rn>, {+|-}<Rm>, <SHIFT>]
  MOV R0, R1, LSL #4
- **assoluto**
  è l'unico metodo che ha una sintassi unica:
  LDR Rn, =#numero

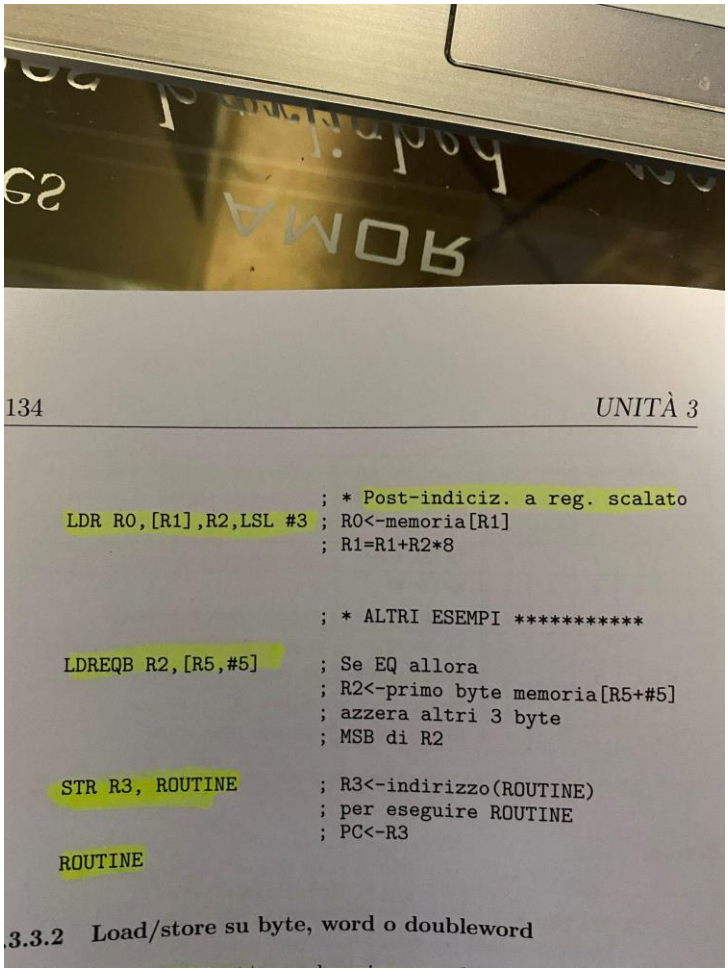
Esempi:

0 0 0 0	X	0	1	Rm LSR #<val>	Rm Logical Shift Right #<val>
X X X X					
0 0 0 0	0	1	0	Rm ASR #0	0 oppure 0xFFFFFFFF in base a Rm[31]
X X X X	X	1	0	Rm ASR #<val>	Rm Arithmetic Shift Right #<val>
0 0 0 0	0	1	1	Rm RRX	Rm Rotate Right eXetended
X X X X	1	1	1	Rm ROR #<val>	Rm Rotate Right #<val>

Tabella 3.33: Tipologie di indirizzamento a registro con scorrimento

Esempi

```
                                ; * Offset immediato
LDR R0,[R1,#2]                  ; R0<-memoria[R1+2]

                                ; * Offset a registro
STR R0,[R1,R2]                  ; R0->memoria[R1+R2]

                                ; * Offset a registro scalato
LDR R0,[R1,R2,LSL #3]           ; R0<-memoria[R1+R2*8]

                                ; * Pre-indiciz. immediato
STR R0,[R1,#2]!                 ; R0->memoria[R1+2]
                                ; R1=R1+2

                                ; * Pre-indiciz. a registro
LDR R0,[R1,R2]!                 ; R0<-memoria[R1+R2]
                                ; R1=R1+R2

                                ; * Pre-indiciz. a registro scalato
STR R0,[R1,R2,LSL #3]!          ; R0->memoria[R1+R2*8]
                                ; R1=R1+R2*8

                                ; * Post-indiciz. immediato
LDR R0,[R1],#2                  ; R0<-memoria[R1]
                                ; R1=R1+2

                                ; * Post-indiciz. a registro
STR R0,[R1],R2                  ; R0->memoria[R1]
                                ; R1=R1+R2
```



134 UNITÀ 3

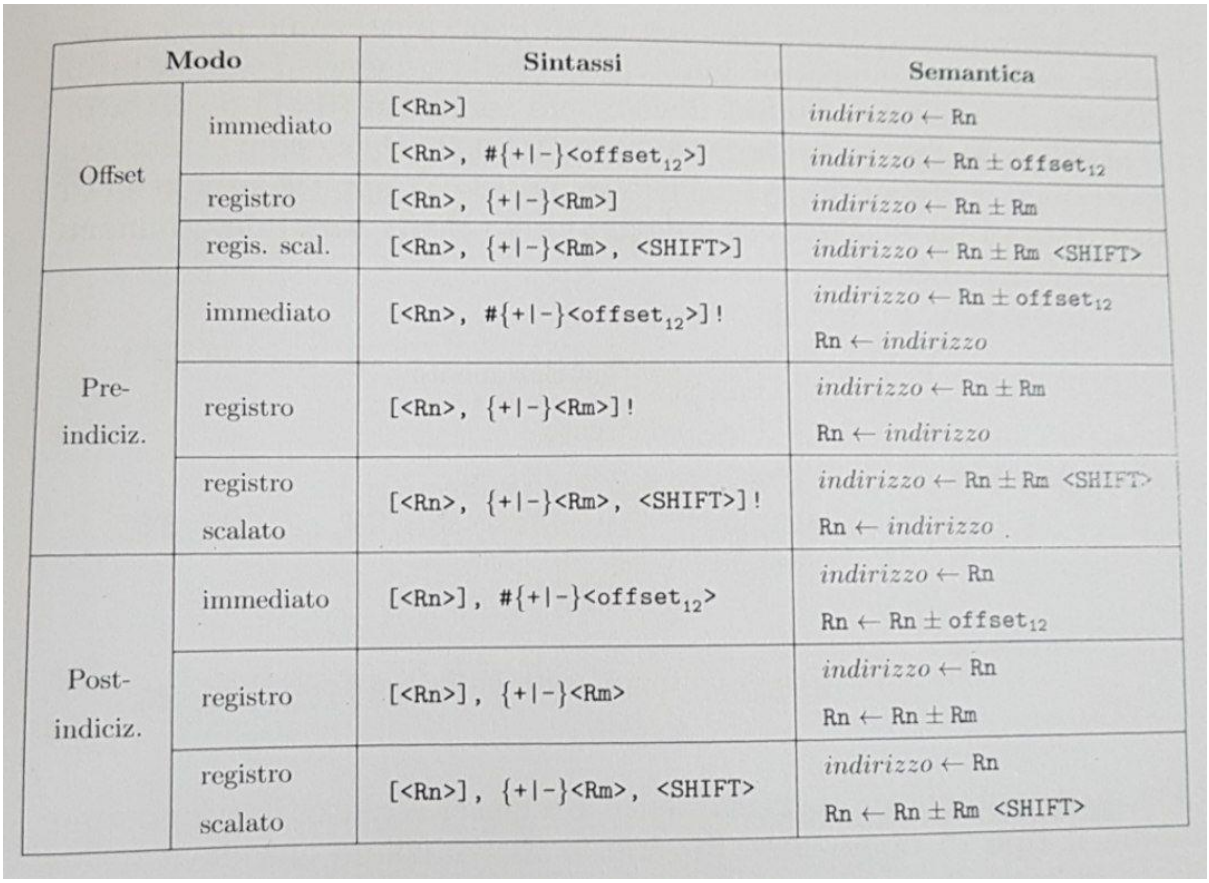
```
                          ; * Post-indiciz. a reg. scalato
LDR R0,[R1],R2,LSL #3     ; R0<-memoria[R1]
                          ; R1=R1+R2*8


                          ; * ALTRI ESEMPI ***********

LDREQB R2,[R5,#5]         ; Se EQ allora
                          ; R2<-primo byte memoria[R5+#5]
                          ; azzera altri 3 byte
                          ; MSB di R2

STR R3, ROUTINE           ; R3<-indirizzo(ROUTINE)
                          ; per eseguire ROUTINE
                          ; PC<-R3

ROUTINE
```

3.3.2 Load/store su byte, word o doubleword

inoltre per favorire l'accesso a blocchi di memoria o a struttura complesse, sono state aggiunte le modalità indicizzate (istruzioni di accesso in memoria - LOAD/STORE):

Mod Indicizzate

- **Pre-indicizzata**
  prima si calcola l'indirizzo e successivamente si scrive a registri base
  esempio di sintassi:
  [<Rn>, {+|-}<Rm>]!
  (! indica la pre-indicizzazione)
- **Post-indicizzata**
  prima si utilizza il registro base come riferimento di memoria e dopo si aggiorna con il nuovo valore
  esempio di sintassi:
  [<Rn>], {+|-}<Rm>]
  (come sopra spiegato qui viene utilizzato il registro base come riferimento il registro in questione è evidenziato dal essere tra parentesi quadre [ ])
- **NB:** non tutte le istruzioni di load/store possono sfruttare queste combinazioni in quanto dipende dallo spazio disponibile nei 32 bit di codifica delle istruzioni stesse



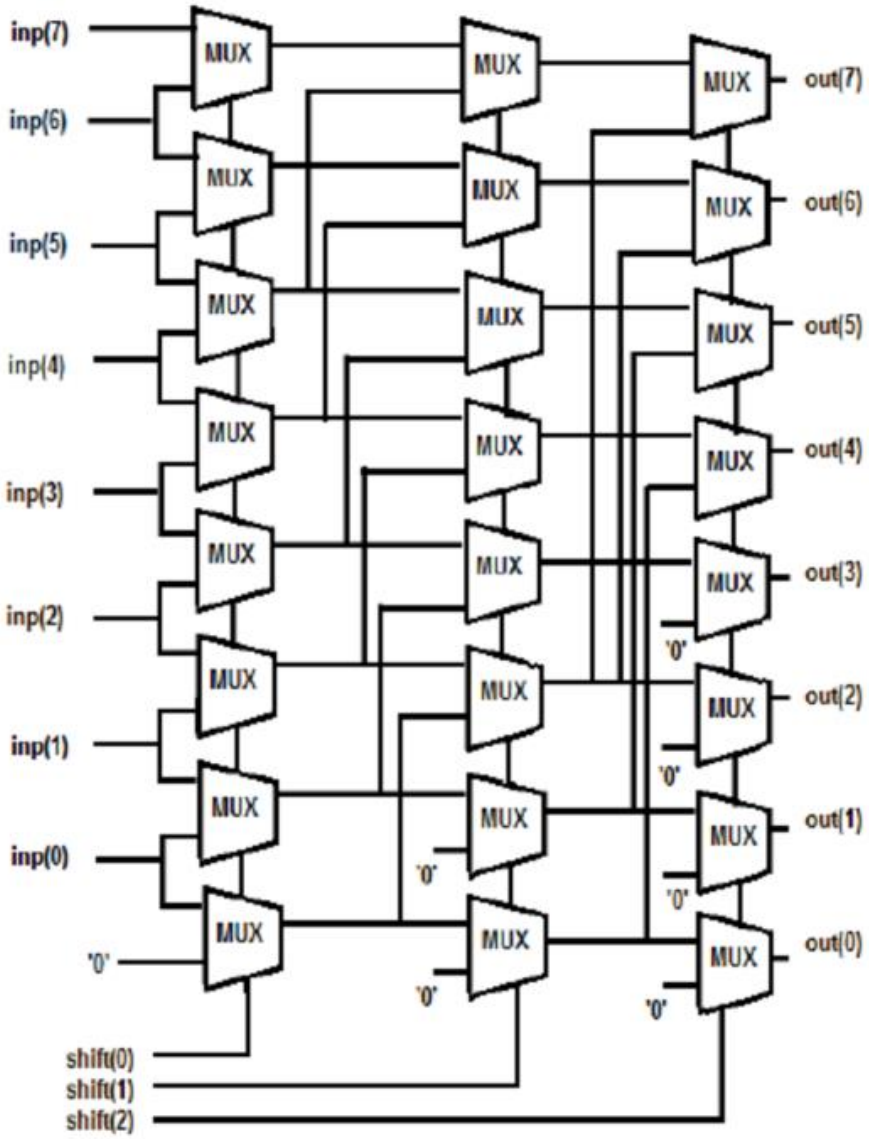
Modo		Sintassi	Semantica
Offset	immediato	[<Rn>]	*indirizzo* ← Rn
		[<Rn>, #{+|-}<offset$_{12}$>]	*indirizzo* ← Rn ± offset$_{12}$
	registro	[<Rn>, {+|-}<Rm>]	*indirizzo* ← Rn ± Rm
	regis. scal.	[<Rn>, {+|-}<Rm>, <SHIFT>]	*indirizzo* ← Rn ± Rm <SHIFT>
Pre-indiciz.	immediato	[<Rn>, #{+|-}<offset$_{12}$>]!	*indirizzo* ← Rn ± offset$_{12}$ Rn ← *indirizzo*
	registro	[<Rn>, {+|-}<Rm>]!	*indirizzo* ← Rn ± Rm Rn ← *indirizzo*
	registro scalato	[<Rn>, {+|-}<Rm>, <SHIFT>]!	*indirizzo* ← Rn ± Rm <SHIFT> Rn ← *indirizzo*
Post-indiciz.	immediato	[<Rn>], #{+|-}<offset$_{12}$>	*indirizzo* ← Rn Rn ← Rn ± offset$_{12}$
	registro	[<Rn>], {+|-}<Rm>	*indirizzo* ← Rn Rn ← Rn ± Rm
	registro scalato	[<Rn>], {+|-}<Rm>, <SHIFT>	*indirizzo* ← Rn Rn ← Rn ± Rm <SHIFT>

Istruzioni di elaborazione e trasferimento dati

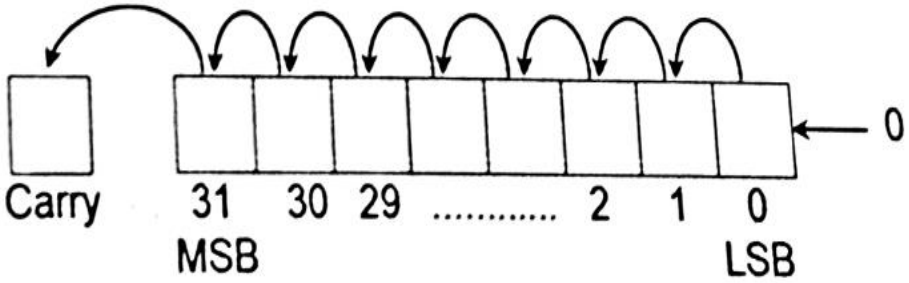
- le istruzioni di trasferimento dati si occupano di trasferire i valori nei registri o tra registri
- presentano sintassi differenti a seconda che si tratti di un trasferimento tra registri di uso generale o che invece coinvolgono i registri di stato
- Le istruzioni di elaborazione e trasferimento dati si classificano così:
    - Istruzioni aritmetico logiche
    - Istruzioni aritmetiche con saturazione
    - Istruzioni di confronto
    - Istruzioni SIMD(Single Instruction Multiple Data)
    - Istruzioni di selezione dei byte
    - IStruzioni di moltiplicazione
    - Istruzioni di trasferimento dati

barrel shifter o switch

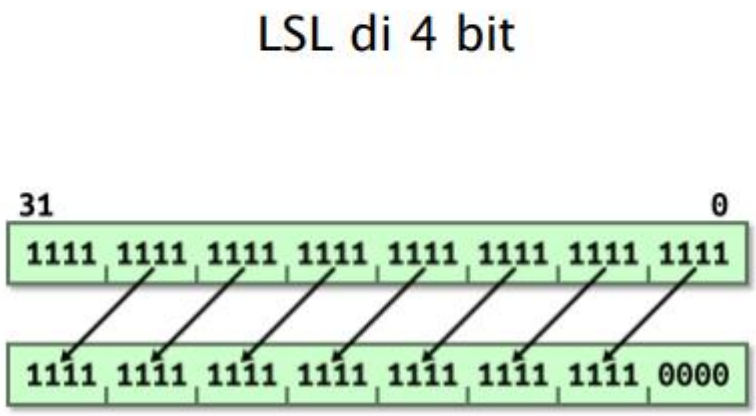
è un circuito digitale che esegue lo shift di una parola binaria di un numero specificato di bit in un solo ciclo di clock. questo è possibile con una sequenza di multiplex disposti in parallelo in cui le uscite di un livello sono collegate agli ingressi dei mux adiacenti nel livello successivo



- Logical shift left (LSL)
  - L'operazione **LSL** si applica a un registro sorgente e richiede un valore che determina il numero di volte in cui eseguire lo scorrimento a sinistra dei bit del registro
  - a ogni passaggio viene inserito il valore zero nel bit meno significativo del registro (**LSB** least significant bit)
  - lo shift a sinistra coincide col moltiplicare il valore del registro per la base (nel caso di ARM 2)
  - viene ripetuta **n** volte dove permette di moltiplicare il valore del registro per 2^n
  - l'ultimo bit più significativo che esce dal registro sorgente ovvero il bit in posizione **32-n** finisce nel flag **C** del registro **CPSR**



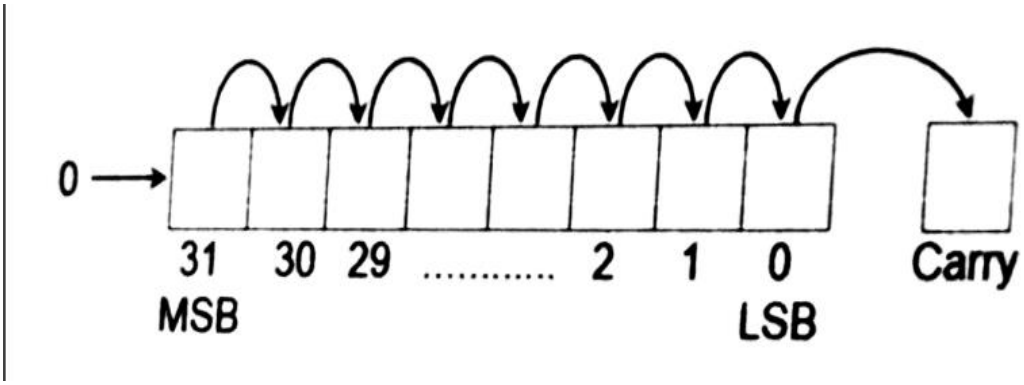
-

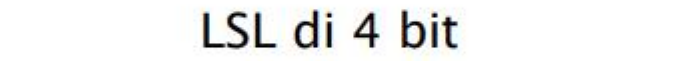


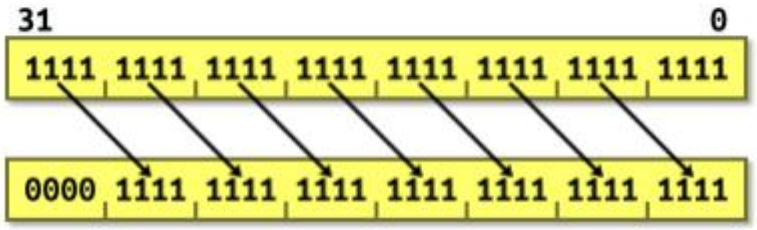
-

- Logical shift right (LSR)
  - L'operazione **LSR** si applica a un registro sorgente e richiede un valore che determina il numero di volte in cui eseguire lo scorrimento a destra dei bit del registro
  - a ogni passaggio viene inserito il valore zero nel bit più significativo del ldrregistro (**MSB** most significant bit)
  - lo shift a sinistra coincide col dividere il valore del registro per la base (nel caso di ARM 2)
  - viene ripetuta **n** volte dove permette di dividere il valore del registro per 2^n
  - l'ultimo bit meno significativo che esce dal registro sorgente ovvero il bit in posizione **n-1** finisce nel flag **C** del registro **CPSR**

-

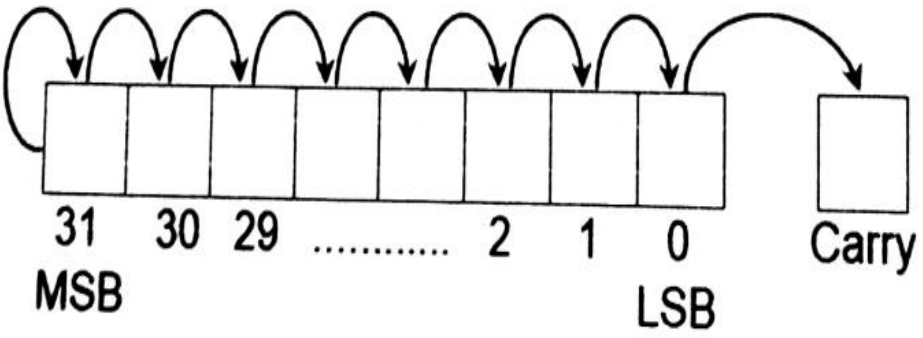




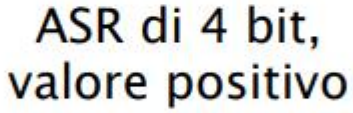


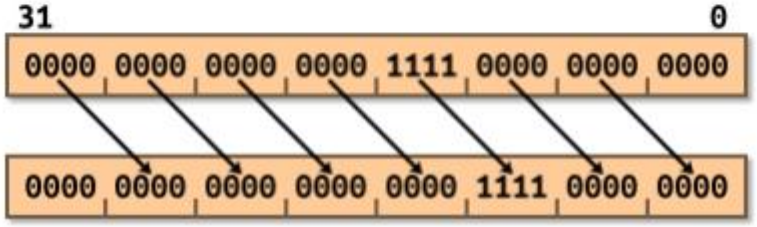
-

- Arithmetic shift right (ASR)
    - L'operazione **ASR** si applica a un registro sorgente e richiede un valore che determina il numero di volte in cui eseguire lo scorrimento a destra dei bit del registro
    - a ogni passaggio viene mantenuto il valore presente nel bit più significativo del registro (**MSB** most significant bit).
    - l'operazione di shift a destra coincide col dividere per la base preservando il segno del contenuto
    - viene ripetuta **n** volte dove permette di dividere il valore del registro per 2^n
    - l'ultimo bit meno significativo che esce dal registro sorgente ovvero il bit in posizione **n-1** finisce nel flag **C** del registro **CPSR**



-

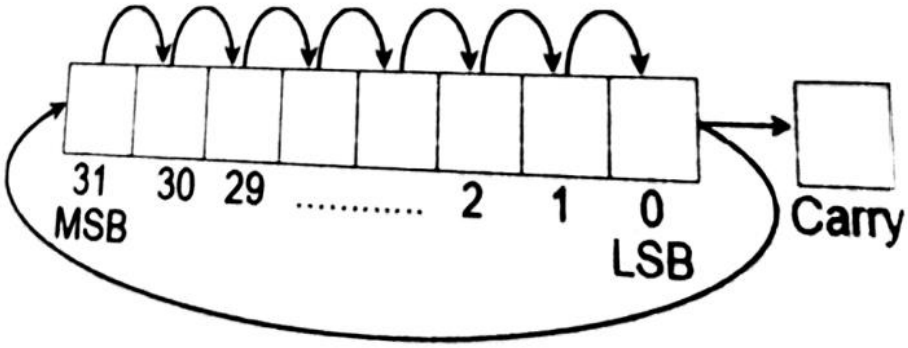




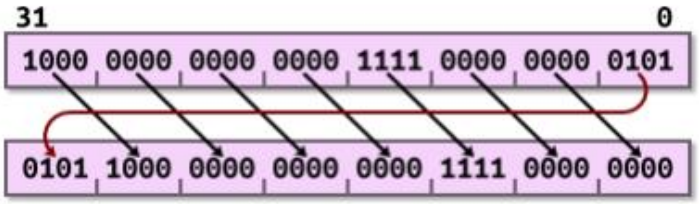
-

- Rotate right (ROR)
    - L’operazione **ROR** si applica a un registro sorgente e richiede un valore che determina il numero di volte in cui eseguire lo scorrimento a destra dei bit del registro
    - ad ogni passaggio **LSB** finisce in posizione **MSB** senza che nessun bit si perda.
    - l’ultimo bit meno significativo che esce dal registro sorgente per girare ovvero il bit in posizione **n-1** finisce nel flag **C** del registro **CPSR**

-

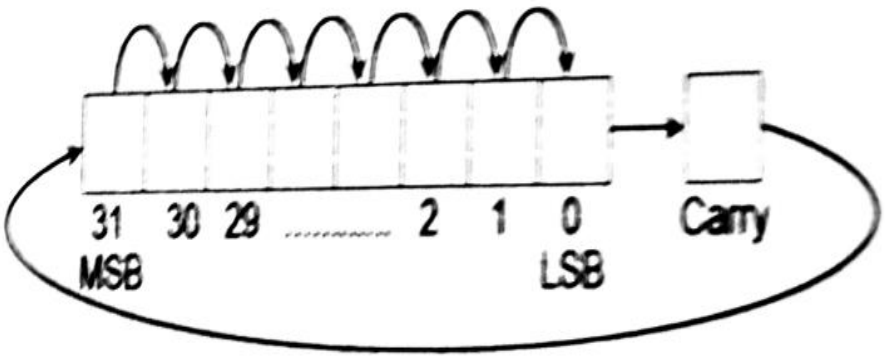


ROR di 4 bit



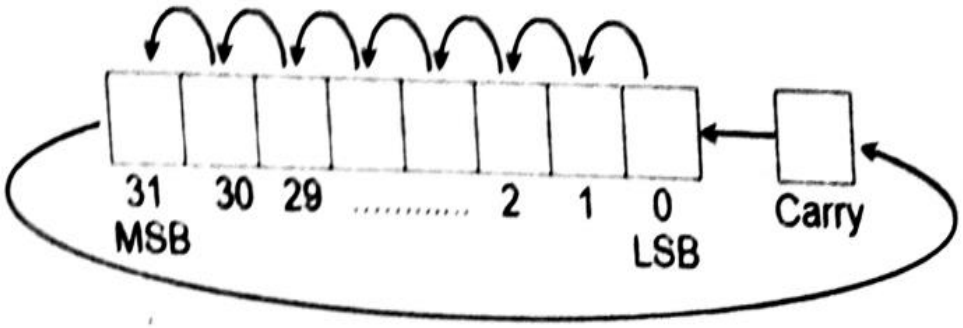
-

- Rotate right with extend (RRX)
  - l'operazione **RRX** (rotate right with extend) funziona esattamente come la **ROR** ad eccezione del fatto che considera il flag **C** come estensione del registro sorgente
  - ad ogni passaggio **LSB** finisce nel flag **C** il vecchio valore del flag **C** in posizione **MSB** e tutti gli altri si spostano verso destra senza che nessun bit venga perso.



-

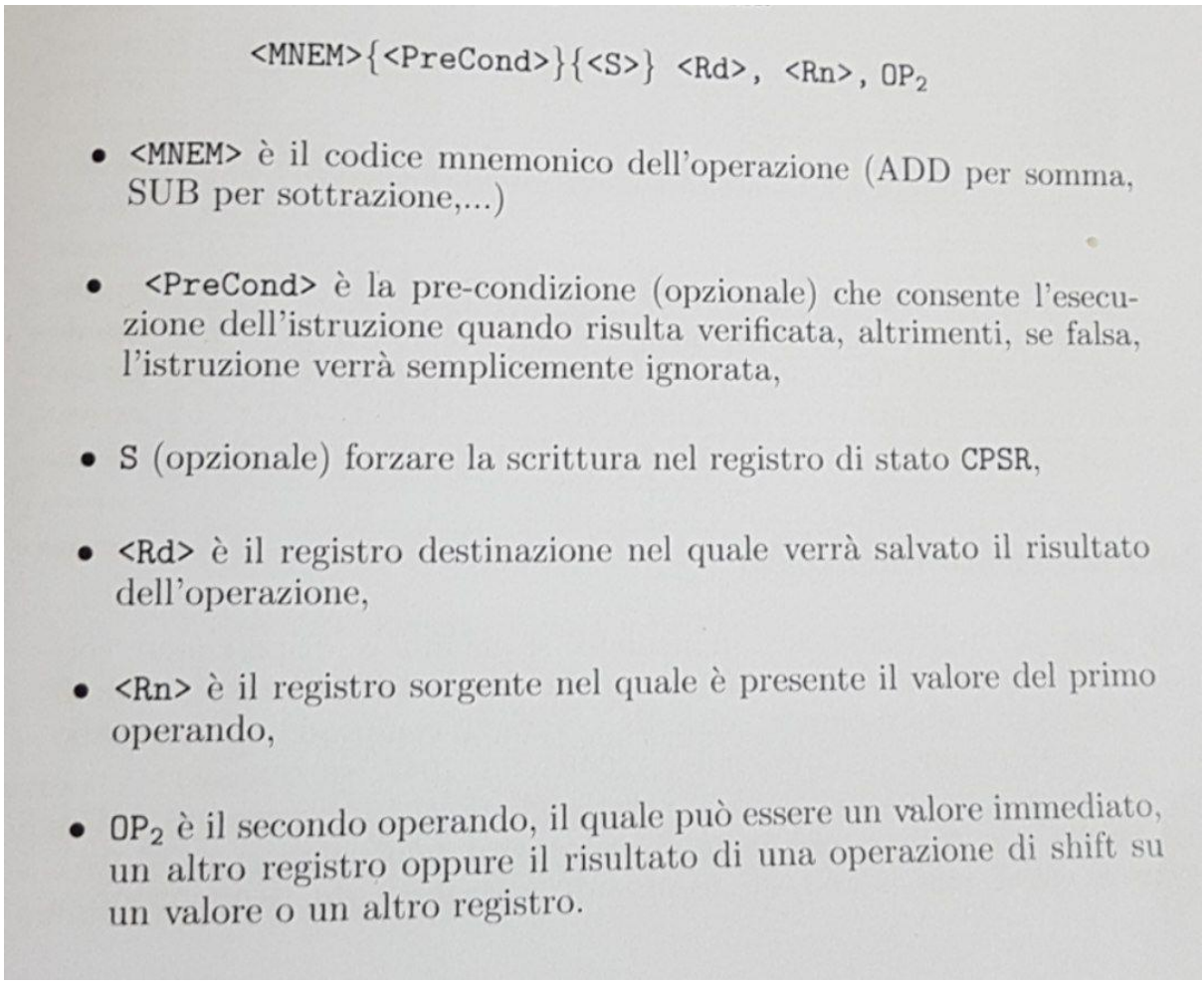
- Rotate left con estensione
  - Nel barrel shifter non esiste una operazione analoga alla **RRX** che sia in grado di eseguire la rotazione di bit verso sinistra (rotate left with extend)
  - si può utilizzare l'istruzione **ADCS** che produce lo stesso effetto
  - *ADCS R0, R0, R0 //RLX R0*



Istruzioni aritmetiche e logiche

- implementano le operazioni matematiche. Queste possono essere di tipo **aritmetiche** (somma, differenza, prodotto), **logiche** (operazioni booleane) o **relazionali** (comparazioni tra due valori).

- le istruzioni aritmetiche logiche operano al massimo su due operandi, possono aggiornare i flag del registro di stato



<MNEM>{<PreCond>}{<S>} <Rd>, <Rn>, OP_2

- <MNEM> è il codice mnemonico dell'operazione (ADD per somma, SUB per sottrazione,...)
- <PreCond> è la pre-condizione (opzionale) che consente l'esecuzione dell'istruzione quando risulta verificata, altrimenti, se falsa, l'istruzione verrà semplicemente ignorata,
- S (opzionale) forzare la scrittura nel registro di stato CPSR,
- <Rd> è il registro destinazione nel quale verrà salvato il risultato dell'operazione,
- <Rn> è il registro sorgente nel quale è presente il valore del primo operando,
- OP_2 è il secondo operando, il quale può essere un valore immediato, un altro registro oppure il risultato di una operazione di shift su un valore o un altro registro.

Codici Mnemonici:

OPCODE	MNEM	Descrizione	Semantica
0000	AND	Congiunzione logica	$Rd \leftarrow Rn \cdot OP_2$
0001	EOR	OR esclusivo	$Rd \leftarrow Rn \oplus OP_2$
0010	SUB	Sottrazione	$Rd \leftarrow Rn - OP_2$
0011	RSB	Sottrazione invertita	$Rd \leftarrow OP_2 - Rn$
0100	ADD	Somma	$Rd \leftarrow Rn + OP_2$
0101	ADC	Somma con riporto	$Rd \leftarrow Rn + OP_2 + C$
0110	SBC	Sottrazione con riporto	$Rd \leftarrow Rn - OP_2 - \overline{C}$
0111	RSC	Sottrazione invertita con riporto	$Rd \leftarrow OP_2 - Rn - \overline{C}$
1110	BIC	Azzera bit	$Rd \leftarrow Rn \cdot \overline{OP_2}$
1100	ORR	Disgiunzione logica	$Rd \leftarrow Rn + OP_2$

Descrizione	Opcode	Sintassi	Semantica
Addizione	ADD	ADD R_d, R_1, R_2/#imm	$R_d = R_1 + R_2$/#imm
Sottrazione	SUB	SUB R_d, R_1, R_2/#imm	$R_d = R_1 - R_2$/#imm
Moltiplicazione	MUL	MUL R_d, R_1, R_2/#imm	$R_d = R_1 \times R_2$/#imm
Carica nel registro	MOV	MOV R_d, R_1/#imm	$R_d \longleftarrow R_1$/#imm
Carica negato nel registro	MVN	MVN R_d, R_1/#imm	$R_d \longleftarrow -(R_1$/#imm)
AND logico	AND	AND R_d, R_1, R_2/#imm	$R_d = R_1 \wedge R_2$/#imm
OR logico	ORR	ORR R_d, R_1, R_2/#imm	$R_d = R_1 \vee R_2$/#imm
OR esclusivo	EOR	EOR R_d, R_1, R_2/#imm	$R_d = R_1 \oplus R_2$/#imm
AND con complemento del secondo operando	BIC	BIC R_d, R_1, R_2/#imm	$R_d = R_1 \wedge \neg R_2$/#imm
Shift logico sinistro	LSL	LSL R_d, R_1, R_2/#imm	$R_d = R_1 \ll R_2$/#imm
Shift logico destro	LSR	LSR R_d, R_1, R_2/#imm	$R_d = R_1 \gg R_2$/#imm
Rotazione destra	ROR	ROR R_d, R_1, R_2/#imm	$R_d = R_1 \circlearrowright R_2$/#imm
Confronto	CMP	CMP R_1, R_2	NZCV $\longleftarrow R_1 - R_2$
Negato del confronto	CMN	CMN R_1, R_2	NZCV $\longleftarrow R_1 + R_2$

Esempi istruzioni aritmetico-logiche

1) *ADD R0, R1, R2*
   Carico in R0 la somma tra il contenuto del registro R1 e del registro R2 (indirizzamento a registro).
2) *ADD R0, R1, #16*
   Carico in R0 la somma tra il contenuto del registro R1 e il numero decimale 16 (indirizzamento immediato).
3) *ADD R0, R1, #0xF*
   Carico in R0 la somma tra il contenuto del registro R1 e il numero esadecimale F (indirizzamento immediato).
4) *ADDLT R0, R1, R2*
   Carico in R0 la somma tra il contenuto del registro R1 e del registro R2 solamente se la condizione indicata con il suffisso LT è verificata
5) *ADDS R0, R1, R2*
   Carico in R0 la somma tra il contenuto del registro R1 e del registro R2, inoltre carico lo stato del risultato nei flags NZCV del registro di stato CPSR. Per esempio, se il risultato della somma va in overflow i flag C (carry) e V (overflow) verranno settati a 1

Indirizzamento Immediato

Si hanno a disposizione solo 12 bit su 32 che costituiscono la word dell'istruzione, quindi si potrebbero rappresentare soltanto i numeri che vanno da 0…+4095 oppure da -2048…+2047. Questo limite non è accettabile perchè il processore ARM gestisce valore che hanno un campo di variazione da 0x00000000 a 0xFFFFFFFF
Vengono memorizzati nei 12 bit due campi: **posizione** (4 bit) e **valore** (8 bit)

Il numero descritto è ottenuto per traslazione, verso destra, dei bit del campo valore per un numero di posizioni *pari* al doppio del campo posizione
Questo implica che si possono esprimere tutti i valori che hanno in binario bit 1 a distanza massima, o al più uguale, ad 8 che possano essere ottenuti per traslazione in pos pari

in generale :

1. convertire il valore in binario e verificare che il bit 1 più significativo e ultimo bit 1 meno significativo siano a distanza massimo di 8 (compressi il primo bit 1 più significativo e ultimo bit 1 meno significativo)
2. verificare che il numero di zeri sia pari
    - il codice binario anche se in riga vanno immaginati come scritti su un cerchio perciò anche se 0xA000000A sembra non indirizzabile perchè in binario è 10100000000000000000000000001010 e sembra non rispettare i nessuno dei due punti in realtà **1010**00000000000000000000000**1010** le parti evidenziate sono come nelle mappe K vicine tra di loro

Esempio:

1. **#0x00000408** = 00000000000000000010000001000 rispetta il punto 1 ma non il punto 2 **00000000000000000001000000100**0 un numero di zeri dispari *(NO)*
2. **#0x55550000** = 10101010101010100000000000000000 non rispetta il punto 1 **1**01010101010**1**0000000000000000 sono a distanza 15 uno dall'altro *(NO)*
3. **#0x00000208** = 00000000000000000000000001000001000 rispetta il punto 1) 00000000000000000000000000**1**00000**1**000 sono a distanza 7 uno dall'altro ma non il punto 2) **00000000000000000000000000001000001000** ha un numero di zeri dispari *(SI)*
4. **#0x00000102** = 00000000000000000100000010 rispetta il punto 1 ma non il punto 2 **00000000000000000001000000100** un numero di zeri dispari *(NO)*
5. **#0x000000EE** = 00000000000000000000000011101110 rispetta il punto 1) 00000000000000000000000000**1**1101**1**10 sono a distanza 7 uno dall'altro e rispetta il punto 2) **00000000000000000000000000001**110**1**110 ha un numero di zeri pari *(SI)*
6. **#0x00002040** =00000000000000000010000001000000 rispetta il punto 1 00000000000000000**1**0000001000000 sono a distanza 8 uno dall'altro e rispetta il punto 2 **00000000000000000001000000100000** ha un numero di zeri pari *(SI)*
7. **#0x00002009** = 10000000001001 non rispetta il punto 1 e sembra come #0xA000000A ma non e cosi infatti #0x00002049 in realtà sarebbe = 00000000000000010000000001001 questo ci rivela e conferma che il punto 1 non è verificato *(NO)*
8. da **#0x00000000 - #0x0000000F** ovviamente sono indirizzabili anche se non soddisfano i due punti perchè fanno dei valori di base per ARM (ovviamente questo ragionamento si applica anche ai loro complementari)
9. **#0xFFFFFFFF** = 11111111111111111111111111111111 è un caso speciale in quanto e il reciproco di #0x00000000 poiché anche se non sembra rispettare i due punti fa parte dei valori di base che devono per forza esistere

Indirizzamento con shift immediato e shift a registro

Se il bit 25 vale 0, viene scritto, nei bit 5 e 6, il tipo di shift da applicare tra quelli disponibili nel barrel shifter (LSL, LSR, ASR, ROR, RRX) mentre il bit 4 specifica il tipo di indirizzamento.
Esempi:
ADD R3, R3, R3, LSL #2 —> R3 = R3+R3*4 INDIRIZ. CON SHIFT IMMED.

RSB R2, R2, R2, LSL # 4—> R2 = R2*16-R2
ADD R0, R1, R2, RRX —> R0 = R1 AND RRX (R2)

ADD R0, R1, R2, LSL R3 —> R0 = R1+R2*(2^R3)
ORR R0, R0, R2, LSL R3 —> R0 = R0 0R R2 * (2^R3)
AND R0, R0, R2, LSR R3 —>R0 = R0 AND R2 / (2^R3)
(INDIRIZZ. CON SHIFT A REGISTRO)

Indirizzamento a registro

il secondo operando è il registro Rm, senza che quest'ultimo subisca operazioni di shift
Questa istruzione è un caso particolare dell'istruzione LSL con indirizzamento immediato in cui i bit non sono traslati

Istruzioni aritmetiche con saturazione

se si utilizzano 32 bit per codificare i numeri interi con segno, l'intervallo di rappresentazione si divide in due sottoinsiemi:

- valori positivi da 1 a 2^{32-1}
- valori negativi da -1 a -2^{32-1}

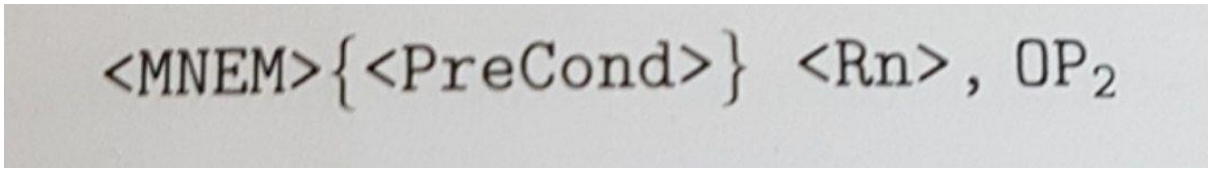
Se durante un'operazione aritmetica si ottiene un numero che eccede uno dei due limiti, il risultato potrebbe essere inserito per sbaglio nel limite opposto.
Per questo ARM usa la tecnica di aritmetica con saturazione, in cui tutte le operazioni finiscono in un dominio intero e finito di valori compresi tra un minimo e un massimo
Il flag Q del CPSR viene attivato.
Le istruzioni che supportano le operazioni aritmetiche con saturazione sono QADD e QSUB

I bit non utilizzati nella codifica dell'istruzione sono segnati con SBZ (Should Be Zero)

<MNEM>{<PreCond>} <Rd>, <Rm>, <Rn>

CODE	MNEM	Descrizione	Semantica
10000	QADD	Addizione con saturazione	Rd $\leftarrow sat(Rm + Rn)$
10100	QDADD	Addizione con doppia saturazione	Rd $\leftarrow sat(Rm + sat(2 \cdot Rn))$
10010	QSUB	Sottrazione con saturazione	Rd $\leftarrow sat(Rm - Rn)$
10110	QDSUB	Sottrazione con doppia saturazione	Rd $\leftarrow sat(Rm - sat(2 \cdot Rn))$

Istruzioni di confronto



OPCODE	MNEM	Descrizione	Semantica
1000	TST	Test bit	CPSR ← <Rn> · OP_2
1001	TEQ	Test uguaglianza bit-a-bit	CPSR ← <Rn> ⊕ OP_2
1010	CMP	Comparazione	CPSR ← <Rn> − OP_2
1011	CMN	comparazione negativa	CPSR ← <Rn> + OP_2

Utilizzano in input 2 operandi ma non hanno un registro destinazione, perché il loro effetto è nella scrittura del registro di stato in base al valore risultato.
Sono 4 ed hanno il seguente comportamento:

1. TST - Test Bit
2. TEQ - Test uguaglianza bit-a-bit
3. CMP - Comparazione
4. CMN - Comparazione negativa

Utilizzano lo stesso formato delle istruzioni aritmetico/logiche e quindi valgono le stesse regole per quanto riguarda le modalità di indirizzamento.
CMP si comporta come ADD, CMN come SUB, TST come AND, TEQ come EOR (Exor)

Istruzioni di moltiplicazione

- Tutte le istruzioni di moltiplicazione accettano due o tre registri come operandi sorgente e scrivono il risultato in un registro destinato
- hanno un funzionamento molto variegato perché consentono di eseguire operazioni di moltiplicazione e somma su word e halfword con o senza segno e di scrivere il risultato in word o double word

Moltiplicazione a due operandi

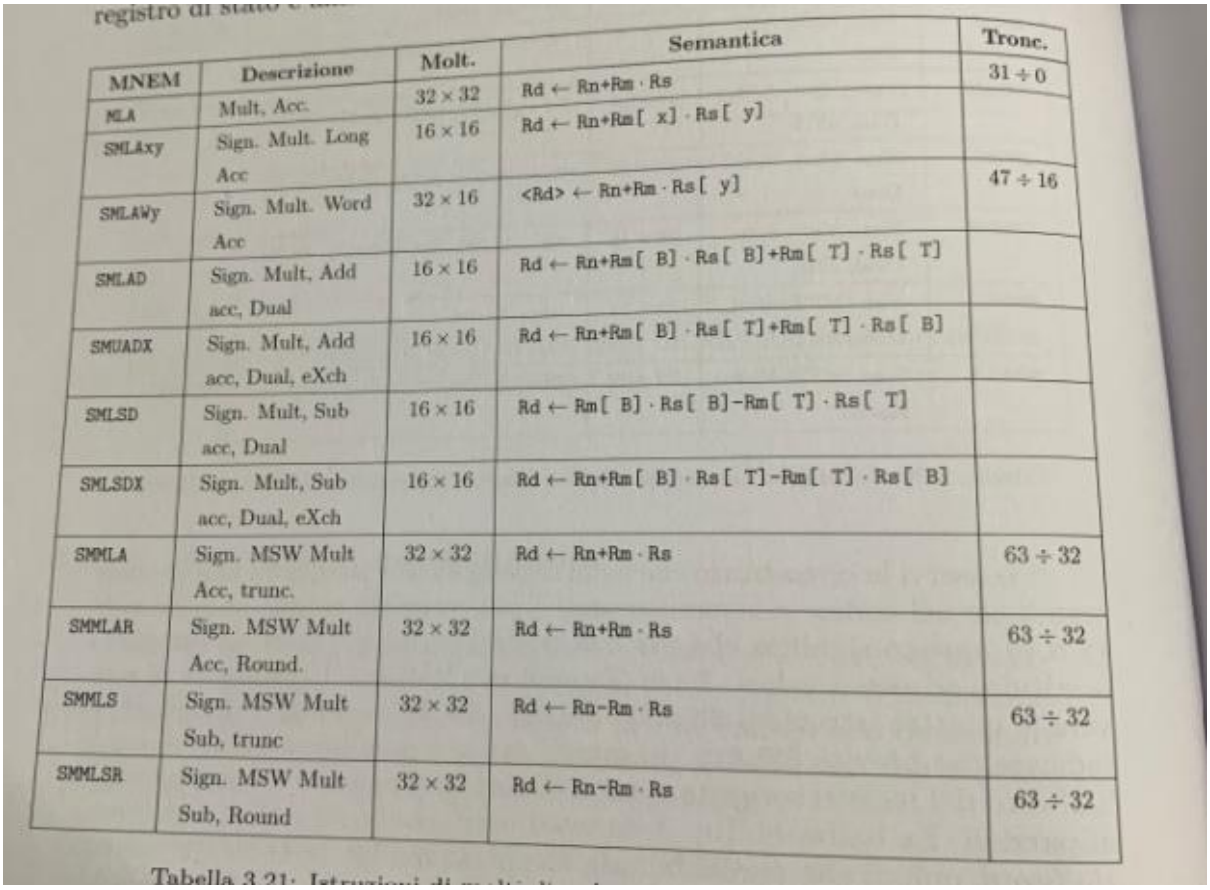
1. utilizza due operandi <Rm>, <Rs> come registri sorgente, mentre <Rd> è il registro di destinazione.
2. l'indicazione del suffisso **S** forza la la scrittura nel **CPSR** ed è utilizzabile solo nell'istruzione **MUL,** in tutte le altre istruzioni di moltiplicazione a due operandi non è possibile forzare la scrittura sul registro dei flag.

MNEM	Descrizione	Molt.	Semantica	Tronc.
MUL	Multiply	32 × 32	Rd ← Rm · Rs	31 ÷ 0
SMULxy	Sign. Mult. Long	16 × 16	Rd ← Rm[x] · Rs[y]	
SMULWy	Sign. Mult. Word	32 × 16	Rd ← Rm · Rs[y]	47 ÷ 16
SMUAD	Sign. Mult, Add Dual	16 × 16	Rd ← Rm[B] · Rs[B]+Rm[T] · Rs[T]	
SMUADX	Sign. Mult, Add Dual, eXch	16 × 16	Rd ← Rm[B] · Rs[T]+Rm[T] · Rs[B]	
SMUSD	Sign. Mult, Sub Dual	16 × 16	Rd ← Rm[B] · Rs[B]-Rm[T] · Rs[T]	
SMUSDX	Sign. Mult, Sub Dual, eXch	16 × 16	Rd ← Rm[B] · Rs[T]-Rm[T] · Rs[B]	
SMMUL	Sign. MSW Mult truncate	32 × 32	Rd ← Rm · Rs	63 ÷ 32
SMMULR	Sign. MSW Mult. Round	32 × 32	Rd ← Rm · Rs	63 ÷ 32

Moltiplicazione a tre operandi

La sintassi delle moltiplicazioni che utilizzano 3 operandi e restituiscono un risultato in word (32 bit) è:
<MNEM>{<PreCond>}{<S>} <Rd>, <Rm>, <Rs>, <Rn>

1. <Rd> è il registro di destinazione , <Rm>, <Rs> e <Rn> sono i registri sorgente cioè gli operandi.
2. Il suffisso S che consente la scrittura dei flag del registro di stato (CPSR) è ammesso solo per l'istruzione MLA



registro di stato

MNEM	Descrizione	Molt.	Semantica	Tronc.
MLA	Mult, Acc.	32 × 32	Rd ← Rn+Rm · Rs	31 ÷ 0
SMLAxy	Sign. Mult. Long Acc	16 × 16	Rd ← Rn+Rm[x] · Rs[y]	
SMLAWy	Sign. Mult. Word Acc	32 × 16	<Rd> ← Rn+Rm · Rs[y]	47 ÷ 16
SMLAD	Sign. Mult, Add acc, Dual	16 × 16	Rd ← Rn+Rm[B] · Rs[B]+Rm[T] · Rs[T]	
SMUADX	Sign. Mult, Add acc, Dual, eXch	16 × 16	Rd ← Rn+Rm[B] · Rs[T]+Rm[T] · Rs[B]	
SMLSD	Sign. Mult, Sub acc, Dual	16 × 16	Rd ← Rm[B] · Rs[B]-Rm[T] · Rs[T]	
SMLSDX	Sign. Mult, Sub acc, Dual, eXch	16 × 16	Rd ← Rn+Rm[B] · Rs[T]-Rm[T] · Rs[B]	
SMMLA	Sign. MSW Mult Acc, trunc.	32 × 32	Rd ← Rn+Rm · Rs	63 ÷ 32
SMMLAR	Sign. MSW Mult Acc, Round.	32 × 32	Rd ← Rn+Rm · Rs	63 ÷ 32
SMMLS	Sign. MSW Mult Sub, trunc	32 × 32	Rd ← Rn-Rm · Rs	63 ÷ 32
SMMLSR	Sign. MSW Mult Sub, Round	32 × 32	Rd ← Rn-Rm · Rs	63 ÷ 32

Tabella 3.21: Istruzioni di

Moltiplicazione con risultato doubleword

La sintassi delle moltiplicazioni che utilizzano 3 operandi e restituiscono un risultato in word (64 bit) è:
<MNEM>{<PreCond>}{<S>} <RdL>, <RdH>, <Rm>, <Rs>

1. Il suffisso S che consente la scrittura dei flag del registro di stato (CPSR) è ammesso solo per le istruzioni: UMULL, UMLAL, SMULL e SMLAL.

MNEM	Descrizione	Molt.	Semantica
UMULL	Unsign. Mult. Long	32×32	[RdH|RdL] ← Rm · Rs
UMLAL	Unsign. Mult. Acc. Long	32×32	[RdH|RdL] ← [RdH|RdL]+Rm · Rs
UMAAL	Unsign. Mult. double Acc. Long	32×32	[RdH|RdL] ← RdH + RdL + Rm · Rs
SMULL	Sign. Mult. Long	32×32	[RdH|RdL] ← Rm · Rs
SMLAL	Sign. Mult. Acc. Long	32×32	[RdH|RdL] ← [RdH|RdL]+Rm · Rs
SMLALxy	Sign. Mult. Acc. Long	16×16	[RdH|RdL] ← [RdH|RdL] + Rm[x] · Rs[y]
SMLALD	Sign. Mult, Add acc. Long, Dual	16×16	[RdH|RdL] ← [RdH|RdL] + Rm[B] · Rs[B]+Rm[T] · Rs[T]
SMLALDX	Sign. Mult, Add acc. Long, Dual, eXch	16×16	[RdH|RdL] ← [RdH|RdL] + Rm[B] · Rs[T]+Rm[T] · Rs[B]
SMLSLD	Sign. Mult, Sub acc, Long, Dual	16×16	[RdH|RdL] ← [RdH|RdL] + Rm[B] · Rs[B]-Rm[T] · Rs[T]
SMLSLDX	Sign. Mult, Sub, acc Long, Dual, eXch	16×16	[RdH|RdL] ← [RdH|RdL] + Rm[B] · Rs[T]-Rm[T] · Rs[B]

Istruzioni di trasferimento dati

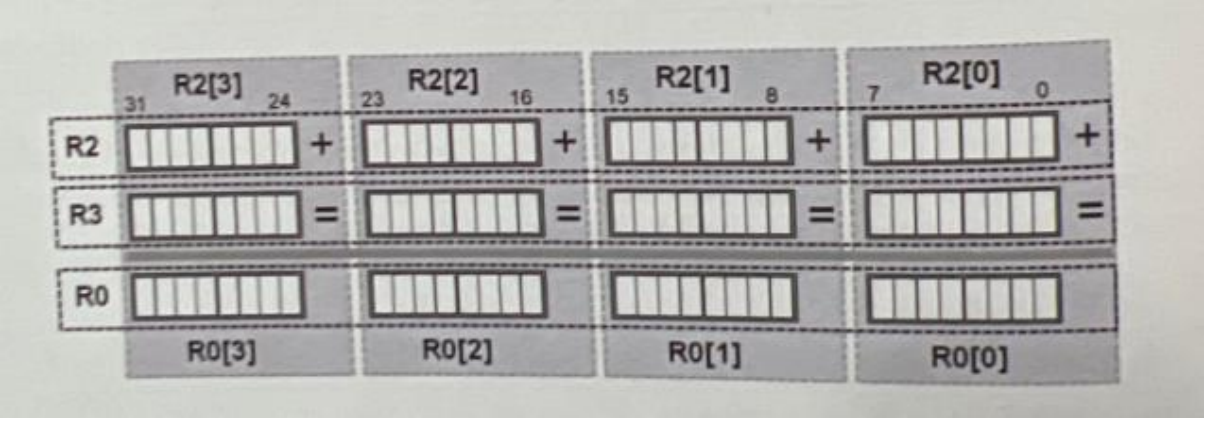
SI occupano di trasferire i valori tra i registri o nei registri
La codifica delle istruzioni di trasferimento ha la medesima forma di quelle di elaborazione dati.

OPCODE	MNEM	Descrizione	Semantica
1101	MOV	Carica registro con OP_2	Rd ← OP_2
1111	MVN	Carica registro con l'inverso di OP_2	Rd ← $\overline{OP_2}$

istruzione **MOV** nel caso si utilizza un **indirizzamento immediato con offset set** [<Rn>, #{+|-}<$offset_{12}$>]

Istruzioni SIMD

ARM ha introdotto delle istruzioni aritmetiche che agiscono sui registri considerandoli come dei contenitori di array definiti su word (32 bit) e halfword (16 bit). Una singola istruzione è in grado di operare contemporaneamente su più dati e quindi di eseguire, per esempio, la somma degli elementi di due array di 2 o 4 elementi.



UADD R0, R2, R3 ;

R0[0] = R2[0] + R3[0]
R0[1] = R2[1] + R3[1]
R0[2] = R2[2] + R3[2]
R0[3] = R2[3] + R3[3]

La sintassi generica di una generica istruzione aritmetica SIMD è la seguente:
<MNEM>{<PreCond>} <Rd>, <Rn>, <Rm>

Il codice mnemonico <MNEM> si ottiene unendo due elementi:

- Il prefisso, che definisce al massimo due modalità di esecuzione dell'operazione (U=Unsigned, Q=con saturazione, H=risultati dimezzati per prevenire overflow)
- Il tipo di istruzione combinata con la dimensione del singolo elemento dell'array (ADD8, SUBB8, ADD16, SUB16, ADDSUBX e SUBBADDX)

Le possibili combinazioni di prefissi applicabili alle istruzioni SIMD sono:

Prefisso	Aritmetica	bit CPSR
S	con segno, modulo 2^8 o 2^{16}	GE[0÷3]
Q	con segno e con saturazione	
SH	con segno e risultati dimezzati	
U	senza segno, modulo 2^8 o 2^{16}	GE[0÷3]
UQ	senza segno e con saturazione	
UH	senza segno e risultati dimezzati	

Le istruzioni SIMD utilizzano i bit GE del CPSR come flag per i byte o le halfword dei risultati. Questo permette di utilizzarli per controllare l'esecuzione di una successiva istruzione. Per le istruzioni che operano su due halfword:

- I bit GE[2 ÷ 3] sono impostati in base al risultato dell'halfword in posizione 1 (Most Significant Halfword).
- I bit GE[0 ÷ 1] sono impostati in base al risultato dell'halfword in posizione 0 (Least Significant Halfword)

Per le istruzioni che operano su 4 byte, i bit GE[i] sono impostati in base al risultato del byte in posizione i con i = [0, 1, 2, 3]. Questi bit sono aggiornati se durante una operazione di addizione di numeri senza segno si raggiunge, o si supera, il massimo valore rappresentabile. La stessa cosa accade se durante un'operazione di sottrazione di numeri con segno il risultato è maggiore o uguale a 0.

Le possibili tipologie di istruzioni SIMD sono:

Tipo	Semantica	Elemento
ADD8	Rd[i]=Rn[i]+ Rm[i], i∈ [0,1,2,3]	byte
SUB8	Rd[i]=Rn[i]- Rm[i], i∈ [0,1,2,3]	byte
ADD16	Rd[i]=Rn[i]+ Rm[i], i∈ [0,1]	halfword
SUB16	Rd[i]=Rn[i]-Rm[i], i∈ [0,1]	halfword
ADDSUBX	Rm[0]↔ Rm[1], Rd[1]=Rn[1]+Rm[1], Rd[0]=Rn[0]-Rm[0]	halfword
SUBADDX	Rm[0]↔ Rm[1], Rd[1]=Rn[1]-Rm[1], Rd[0]=Rn[0]+Rm[0]	halfword

Le istruzioni SIMD dell'architettura ARM che operano su un array di 4 byte sono:

MNEM	Tipo	con Segno	Saturazione	Risultati/2
QADD8	ADD8	✓	✓	
SADD8		✓		
SHADD8		✓		✓
UADD8				
UHADD8				✓
UQADD8			✓	
QSUB8	SUB8	✓	✓	
SSUB8		✓		
SHSUB8		✓		✓
USUB8				
UHSUB8				✓
UQSUB8			✓	

Interessante è l'istruzione USAD8 che esegue la somma dei valori assoluti delle differenze dei byte di due array, che presenta la stessa sintassi delle altre istruzioni SIMD. Esiste anche la variante del registro destinazione che permette di accumulare i risultati parziali (USADA8) in questo caso nella sintassi occorre aggiungere <Rn>.

MNEM	Descrizione	Semantica
USAD8	Unsign. Sum Abs. Diff.	$\texttt{<Rd>} \leftarrow \sum_{0}^{3}{}_{i} \lvert\texttt{Rm[i]-Rs[i]}\rvert$
USADA8	Unsign. Sum Abs. Diff. Acc	$\texttt{<Rd>} \leftarrow \texttt{<Rn>} + \sum_{0}^{3}{}_{i} \lvert\texttt{Rm[i]-Rs[i]}\rvert$

Tabella 3.16: Altre istruzioni SIMD su array di byte

Le istruzioni SIMD dell'architettura ARM che operano su array di due halfword sono:

MNEM	Tipo	con Segno	Saturazione	Risultati/2
QADD16	ADD16	✓	✓	
SADD16		✓		✓
SHADD16		✓		
UADD16				✓
UHADD16			✓	
UQADD16			✓	
QSUB16	SUB16	✓		
SSUB16		✓		✓
SHSUB16		✓		
USUB16				✓
UHSUB16			✓	
UQSUB16			✓	
QADDSUBX	ADDSUBX	✓		
SADDSUBX		✓		✓
SHADDSUBX		✓		
UADDSUBX				✓
UHADDSUBX				
UQADDSUBX			✓	
QSUBADDX	SUBADDX	✓	✓	
SSUBADDX		✓		
SHSUBADDX		✓		✓
USUBADDX				
UHSUBADDX				✓
UQSUBADDX			✓	

Tabella 3.17: Istruzioni aritmetiche SIMD che operano su array di halfword

Istruzioni di array (da pagina 173 a pag 175)
istruzione **DCD** crea un array abbia un formato valido cioè che nella prima word ci sia scritto il numero di elementi di cui si compone l'array la sintassi è:

DCD #valore, #valore

esempio:

```
V       DCD 3, 2, 5, 6
        LDR RO, =V
        LDR R2, [R0]
        LSL R3, R2, #4
```

V è una variabile con riferimento di memoria (o puntatore) relativo alla prima delle quattro word (o valori indicati)
R0 contiene l'indirizzo della prima cella della struttura (array)
R1 scorre gli elementi che compongono la struttura (array)
R2 contiene gli elementi estratti
R3 indica l'ultimo elemento della struttura (array)

Istruzioni di branch

- Le istruzioni di **branch** (o **jump**) sono utili per il controllo del flusso di esecuzione delle istruzioni. Le principali istruzioni sono **B** (branch) e **BL** (branch with link).
- **L'istruzione B** carica nel PC (R15) l'indirizzo della prima istruzione della procedura che si desidera eseguire, causando così un salto nel flusso di esecuzione delle istruzioni. Per comodità le procedure vengono identificate univocamente con dei nomi detti **label** (o **etichetta**).
- **L'istruzione BL** è simile all'istruzione B, in più però carica nel registro LR (R14) l'indirizzo di ritorno della procedura, ovvero il valore del PC nel momento in cui viene eseguita l'istruzione BL.
- Le istruzioni di branch sono necessarie per l'implementazione di istruzioni di condizione (if-thenelse), cicli e chiamate di funzioni.
- Per calcolare l'indirizzo di arrivo, ARM fa:
    - estende il segno a 30 bit il valore presente nei 24-bit di offset
    - esegue uno shift di 2 bit per arrivare a coprire i 32 richiesti
    - aggiunge il valore del PC

Istruzione di Load e Store

muovono dati da (load) e verso (store) la memoria principale e viceversa.
L'istruzione **LDR** (**Load Register**) carica in un registro destinatario un byte, una half word o una full world contenuta in una data locazione della memoria principale. La sintassi è:

LDR{type}{condition} dest, [pointer]
LDR{type}{condition} dest, [pointer,offset]

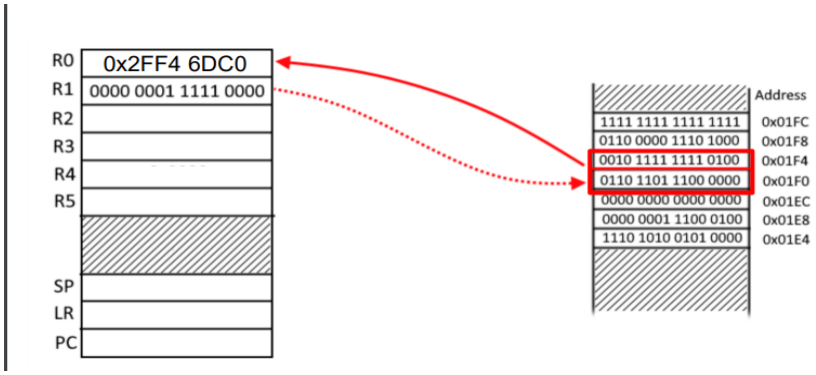
Il campo **dest** è il registro destinatario nel quale caricare il contenuto prelevato dalla memoria.
Il campo **[pointer]** indica un registro puntatore il quale definisce un contenuto che punta all'indirizzo in memoria della **cella** che si desidera referenziare. É possibile inoltre definire un **offset** il quale verrà sommato al puntatore.

L'istruzione **STR** (**Store Register**) esegue esattamente l'operazione inversa di LDR; carica in memoria il contenuto di un registro **sorgente** all'indirizzo definito dal **puntatore**.

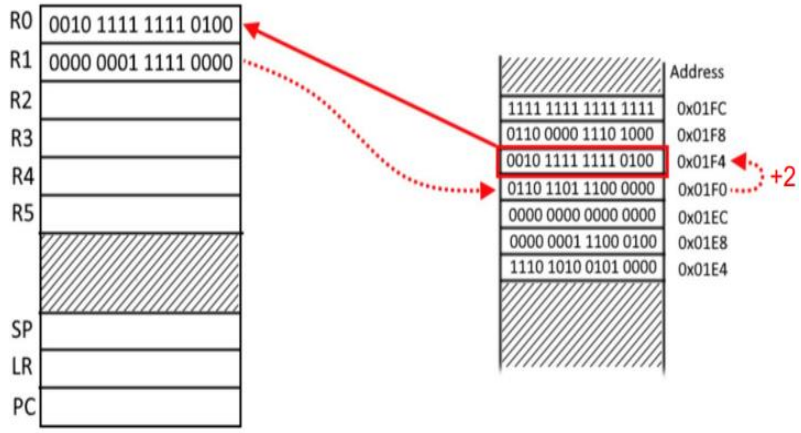
STR{condition} source, [pointer,offset]

Esempi istruzioni di load e store

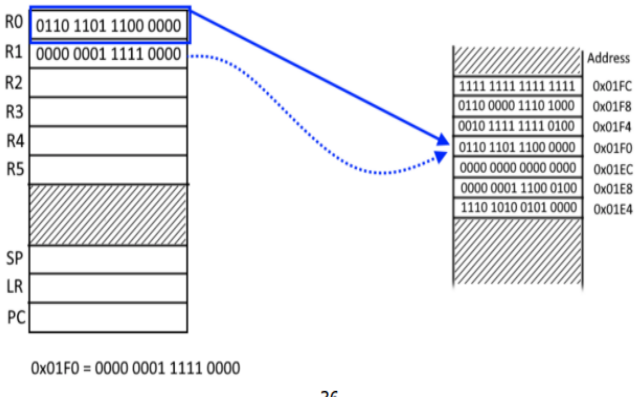
1) *LDR R0, [R1, #2]*
   Carico in R0 la parola in memoria puntata dal registro R1



2) *LDRH R0, [R1, #2]*
   Carico in R0 la HalfWord (**H**) 2 byte a partire dalla cella puntata dal registro R1



3) *STR R0, [R1]*
   Carico all'indirizzo puntato da R1 la parola contenuta nel registro R0



istruzione di load e store su byte, word o doubleword

le operazioni di load/store che agiscono su byte con segno, halfword (con o senza segno) e word (con e senza segno) la sintassi è

LDR {<Precond>}{SB|H|SH|} <Rd>, <indirizzamento>
STR {<Precond>}{H} <Rd>, <indirizzamento>

Suffisso	Descrizione
B	Byte senza segno
SB	Byte con segno
H	Halfword senza segno
SH	Halfword con segno
<non indicato>	Word

istruzioni di load e store su registro multipli

le istruzioni di load/store multipli (**LDM** e **STM**) eseguono un trasferimento di un qualsiasi numero di registri da o verso la memoria questa istruzioni sono pensate per far corrispondere al registro con indice più piccolo l'indirizzo di memoria più basso e, viceversa, a quello con indice più grande l'indirizzo più alto. la sintassi delle istruzioni **LDM** e **STM** è la seguente:
<MNEM>{<Precond>}{<Modo agg>} <Rn>{!}.<registri>{^}

Modo Aggiornamento		Memoria		Calcolo di Rn
Sintassi	Descrizione	Ind. Inizio	Ind. Fine	PreCond · W
IA	*Increment After* post-incremento	Rn	Rn+4 · NumReg−4	Rn ← Rn+4 · NumReg
IB	*Increment Before* pre-incremento	Rn+4	Rn+4 · NumReg	Rn ← Rn+4 · NumReg
DA	*Decrement After* post-decremento	Rn−4 · NumReg+4	Rn	Rn ← Rn−4 · NumReg
DB	*Decrement Before* pre-decremento	Rn−4 · NumReg	Rn−4	Rn ← Rn−4 · NumReg

esempio:

LDR R7, {R1, R4, R5}
STMDB R3!, {R4-R6, R11-R12}
STMFD R13!, {R0-R10, LR}
LDMFD R13!, {R1-R5, PC}

istruzioni di accesso esclusivo alla memoria

- queste istruzioni nascono per la sincronizzazione dei processi in ambiente multi processore basato su memoria condivisa. sono istruzioni progettate per avere il comportamento atomico richiesto nei semafori senza bloccare tutte le risorse di sistema durante le fasi di accesso alla memoria condivisa.
- l' istruzione **LDREX** carica in modo condizionale nel registro **Rd** la word presente nell'indirizzo indicato da **Rn** (**Rd**←**memoria[Rd]**) la sintassi è:
  <LDREX>{<Precond>} <Rd>, [<Rn>]
- l'istruzione **STREX** carica in modo condizionale nel registro **Rm** in memoria (**memoria[Rn]**←**Rm**) la sintassi è:
  <STREX>{<Precond>} <Rd>, <Rm>, {<Rn>}

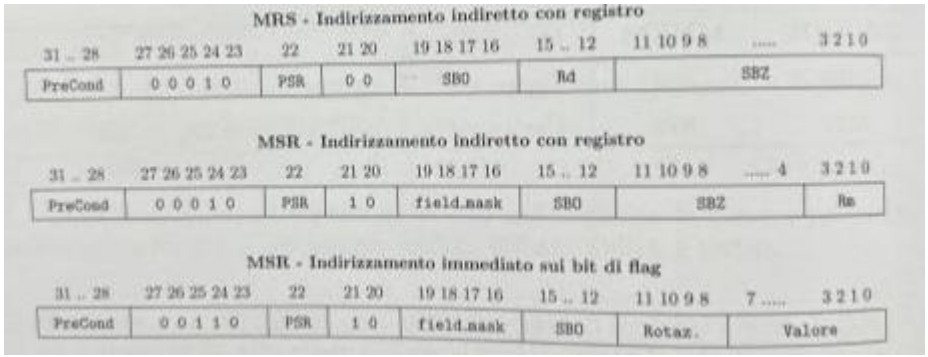
Istruzioni di accesso ai registri di stato

Le istruzioni per l'accesso ai registri di stato sono:

1. **MRS**(Move to Register from Status register) copia il valore del CPSR o del SPSR nell'attuale modo di funzionamento del processore, all'interno dei registri di uso generale, la sintassi è :
   MRS {<PreCond>} <Rd>, {CPSR|SPSR}
2. **MSR(**Move to Status register from Register**)** copia il contenuto di un registro o una costante in una o più ambiti del registro CPSR o del SPSR nell'attuale modo di funzionamento del processore, le sintassi:
   MSR{<PreCond>} {CPSR|SPSR}{_<ambiti>}, <Rm>

MSR{<PreCond>}{CPSR|SPSR}{_<ambiti>}, #<valore>

Il campo <ambiti> permette di specificare attraverso una lista di caratteri in sequenza(c = controllo, x = estensione, s = stato, f = flag), quali sezioni del registro di stato devono essere sovrascritte con il contenuto del secondo operando.



Il bit 25 rappresenta il tipo di operando(0 = registro, 1 = valore immediato), il bit 22 specifica il tipo di registro di stato(0 = CPSR o 1 = SPSR), il campo field_mask, è costituito da 4 bit, responsabile del mascheramento dei diversi ambiti dei registri di stato.

La sintassi dell'istruzione MSR permette di inserire nel campo ambito una sequenza di letterali, che corrisponde ai diversi ambiti da mascherare:

Istruzione MSR		Program Status Register	
bit codifica	letterale	ambito	bit coinvolti
16	c	controllo	7÷0
17	x	estensione	15÷8
18	s	stato	23÷16
19	f	flag	31÷24

Per esempio è divenuta obsoleta la lettura del bit E con un'istruzione MRS.

I bit di stato di esecuzione nel CPSR, diversi dal bit E, possono essere letti soltanto quando il processore è nello stato di debug. è stato anche introdotto un registro chiamato APSR (Application Program Status Register) che contiene i flag di stato dell'ALU e che andrebbe utilizzato per accedere modalità Utente ai flag delle condizioni.

Esempio

```
MRS R0    , CPSR        ; R0=CPSR (lettura stato)
BIC R0    , R0, #0x1F   ; Ripulisce mode bit
ORR R0    , R0, #0x13   ; Imposta modo Supervisor
MSR CPSR_c, R0          ; CPSR=R0 (scrittura stato)
```

Istruzioni verso i coprocessori

- il processore ARM può richiedere ai vari coprocessori operazioni di calcolo numerico (istruzioni di elaborazione dati) mentre continua a svolgere altri compiti

- ARM può arrivare ad avere fino a 16 coprocessori che sono identificati nelle istruzioni con gli alias **P0-P15**
- Le istruzioni fanno parte del modo 5 di indirizzamento

Istruzioni di elaborazione dati

L'istruzione CDP(Coprocessor Data Processing) consente di richiedere ad un coprocessore disponibile (tra P0…P15) di eseguire una istruzione tra quelle disponibili nel suo Instruction Set

Istruzioni di trasferimento dati in memoria

Le istruzioni di trasferimento in memoria sono:

1. **LDC**(Load to Coprocessor)
2. **STC**(Store from Coprocessor)

LDC carica i registri del coprocessore con i dati contenuti in una sequenza contigua di locazioni in memoria.
STC memorizza in una sequenza di indirizzi di memoria contigui i dati contenuti nei registri del coprocessore

Il campo CP_Num indica con un numero intero quale coprocessore dovrà rispondere alla chiamata
il bit L, come per le istruzioni Load/Store, indica se l'istruzione è LDC(L=1) o STC(L=0)
il bit U determina se l'offset va sommato o sottratto
i bit P e W determinano il tipo di indirizzamento

Istruzioni di trasferimento dati tra registri

Le istruzioni di trasferimento dati tra registri sono:

1. **MCR(**Move to Coprocessor from ARM register**)**
2. **MCRR(**Move to Coprocessor from two ARM register**)**
3. **MRC(**Move to ARM register from Coprocessor**)**
4. **MRRC(**Move to two ARM register from Coprocessor**)**

MCR trasferisce il contenuto del registro RD del processore nel registro del coprocessore,mentre **MRC** trasferisce il contenuto dei registri del coprocessore nel registro RD del processore o nei flag del registro di stato
Nel caso di **MRC ,** se viene indicato R15, l'effetto ottenuto è l'aggiornamento dell'ambito dei flag nel registro CPSR
Il bit **r/c(negato)** individua il tipo di istruzione: se 1 si tratta di MRC altrimento di MCR
per MCRR e MRRC: se il bit **r/c(negato)** vale 1 si tratta di MRRC altrimenti di MCRR

Tipi di istruzioni	MNEM	Possibile comportamento
Elaborazione Dati	CPD	*Coprocessor Data Operations* CRd ← OPCODE$_1$(CRn,OPCODE$_2$(CRm))
Trasferimento in Memoria	LDC	*Load Coprocessor Register* CRd ← memoria[indiriz(Rn)]
	STC	*Store Coprocessor Register* memoria[indiriz(Rn ← OPCODE$_1$(CRd)
Trasferimento nei Registri	MCR	*Move to Coprocessor from ARM Register* CRn ← OPCODE$_1$(Rd,OPCODE$_2$(CRm))
	MCRR	*Move to Coprocessor from two ARM Registers* CRm ← OPCODE$_1$(Rd,Rn)
	MRC	*Move to ARM Register from Coprocessor* Rd ← OPCODE$_1$(CRn,OPCODE$_2$(CRm))
	MRRC	*Move to two ARM Registers from Coprocessor* Rd,Rn ← OPCODE$_1$(CRn)

Istruzioni per generare eccezioni

- software interrupt
  - istruzione **SWI** genera una eccezione **SWI** che porta il processore nello stato ARM e la modalità cambia in **supervisor**
  - quindi il registro **CPSR** viene salvato nel registro **SPSR_SVC**
  - questa è la modalità attraverso la quale un processo utente invoca una system call del sistema operativo
  - sintassi:
    SWI{<Precond>} <$costante_{24}$>
    L'indirizzamento è immediato

- breakpoint software
  - istruzione **BKPT (**breakpoint) è utilizzata per introdurre punti di interruzione nel codice e causare una eccezione di **prefetch abort**
  - sintassi
    BKPT<$costante_{16}$>
    L'indirizzamento è immediato

Istruzioni per il packing e l'unpacking dei dati

- packing
  - le istruzioni per il packing dei dati sono due:
    1) PKHBT (pack halfword bottom top) unisce le cifre meno significative del primo operando con quelle più significative del secondo
       esempio:
       LDR R1, =0x11112222
       LDR R2, =0x33334444

       PKHBT R3, R1, R2 //R3=0x33332222

    2) PKHTB (pack halfword top bottom) unisce le cifre più significative del primo operando con quella meno significative del secondo
       esempio:
       LDR R1, =0x11112222
       LDR R2, =0x33334444

       PKHBT R3, R1, R2 //R3=0x11114444

- unpacking

  sono istruzioni che permettono di ruotare di 0, 8, 16 o 24 bit il contenuto di un registro, inoltre possono accumulare il valore ottenuto, nei vari incrementi, in un registro destinazione.

istruzioni di saturazione

Controllano se un valore è al di fuori di un intervallo prestabilito e in questo caso di assegnare l'estremo più vicino dell'intervallo stesso
ci sono 4 versioni di queste istruzioni, a seconda che si considerino valori con segno o meno e se agiscano su intere word o halfword

1. **SSAT(Signed Saturate)**
2. **USAT(Unsigned Saturate)**
3. **SSAT16(Signed Saturate two 16-bit values)**
4. **USAT16(Unsigned Saturate two 16-bit values)**

istruzione di conteggio degli zeri finali

- istruzione **CLZ** (count leading zero) rientra nella categoria dell'elaborazione dati e fornisce quanti bit sono zero a partire dal bit più significativo, spostandosi verso la cifra meno significativa
- la sintassi:
  CLZ{<Precond>} <Rd>, <Rm>
- Esempi:

```
LDR R1, =0x0000FFFF
MOV R3, #0x10000000
CLZ R2, R1                  // R2=0x0000010
CLZ R4, R3                  // R4=0x0000003
```

istruzioni di inversione di byte

- Queste istruzioni converte i dati dal formato little-endian a quello big-endian e viceversa
- sintassi:
  <MNEM>{<Precod>} <Rd>, <Rm>

MNEM	Nome esteso e semantica
REV	*byte-Reverse word* $Rd[B_3] \leftarrow Rm[B_0]$ $Rd[B_2] \leftarrow Rm[B_1]$ $Rd[B_1] \leftarrow Rm[B_2]$ $Rd[B_0] \leftarrow Rm[B_3]$
REV16	*byte-Reverse packed 16-bit* $Rd[B_1] \leftarrow Rm[B_0]$ $Rd[B_0] \leftarrow Rm[B_1]$ $Rd[B_3] \leftarrow Rm[B_2]$ $Rd[B_2] \leftarrow Rm[B_3]$
REVSH	*byte-Reverse signed 16-bit* $Rd[B_1] \leftarrow Rm[B_0]$ $Rd[B_0] \leftarrow Rm[B_1]$ $Rd[B_2B_3] \leftarrow \begin{cases} 0xFFFF & se\ Rm[7]=1 \\ 0x0000 & se\ Rm[7]=0 \end{cases}$

- Esempio
  LDR R1, =0x0FE10547
  REV R2, R1 // R2=0x4705E10F NB: vengono girati a coppie es 47, 05, E1 ecc
  REV16 R3, R1 // R3=0xE10F4705
  REVSH R4, R1 // R4=0x00004705 Halfword

pseudo-istruzioni

- No Operation
  - la pseudo-istruzione **NOP** è una operazione che non esegue alcun compito
  - può essere utilizzata come segnaposto nel corpo del programma da sostituire in seguito con istruzioni attive

- può invalidare un istruzione esistente (es: un branch) per scopi di debug
- può richiedere un ciclo di clock
- in alcune circostanze il processore potrebbe rimuoverla dalla pipeline prima che raggiunga la fase di esecuzione
- ARM potrebbe tradurre NOP con:

  MOV R0, R0 //nella modalità arm

  oppure

  MOV R8, R8 //nella modalità Thumb

Shifting e rotazione

- ARM non dispone di istruzioni di shifting e rotazione dirette se non attraverso l'innesto all'interno di istruzioni di elaborazione dati
- l'assembler (ARM) riconosce 5 pseudo-istruzioni (**LSL, LSR, ASR, ROR** e **RRX**) che sono poi tradotte utilizzando l'istruzione **MOV**
- **RRX** utilizza un solo operando ed è differente dalle altre istruzioni di shift o rotate con la sintassi:
  RRX {<Precond>}{<S>} <Rd>, <Rn>
- mentre per le istruzioni **LSL, LSR, ASR, ROR** le sintassi possibili sono:
  <MNEM>{<Precond>}{<S>} <Rd>, <Rn>, Rs
  <MNEM>{<Precond>}{<S>} <Rd>, <Rn>, #valore
- il registro **Rs** indica nel suo Byte meno significativo il numero di cifre da scorrere
- il **#valore** è un numero intero compreso tra 1 e 31

MNEM	Descrizione	Semantica
LSL	*Shift Left*	<Rd> ← <Rn> << #valore
LSR	*Shift Right*	<Rd> ← <Rn> >> #valore
ASR	*Shift Right with sign extend*	<Rd> ← <Rn> >> #valore
ROR	*Rotate Right*	<Rd>,C ← <Rn> >> #valore
RRX	*Rotate Right with eXtend*	[<Rd>|C] ← [<Rd>|C] >> 1

Copia registro

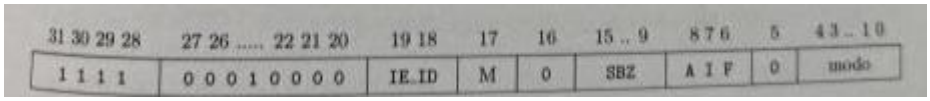
- istruzione **CPY** è una pseudo-istruzione sinonima della **MOV offset a registro** ma, differentemente da quest'ultimo non ha la possibilità di impostare i flag del registro di stato (**S**) e neanche di eseguire lo shift sul secondo operando la sua sintassi risulta:
  CPY{<Precond>} <Rd>, <Rm>

Change Processor State

CPS (Change Processor State) è una istruzione molto duttile che può essere utilizzata per abilitare o disabilitare le eccezioni (FIQ, IRQ e data Abort mask) e allo stesso tempo modificare il modo di funzionamento del processore (Utente, FIQ, IRQ, Supervisor, Abort, Undefined e System) nel registro CPSR. La stessa istruzione si può utilizzare esclusivamente per alterare la modalità di funzionamento, senza alterare le abilitazioni sul trattamento delle eccezioni.
La sintassi dell'istruzione CPS risulta quindi:
CPS(IE|ID} <int_flag> {,#<modo>}
Il campo int_flag indica quali bit nel registro di stato verranno abilitati o disabilitati.

Sintassi	Current Program Status Register	
int_flag	bit	significato
a	A	Imprecise data Abort
i	I	Interrupt Request
f	F	Fast Interrupt Request

La codifica dell'istruzione è la seguente:



Il campo IE_ID permette di specificare se si tratta di un Interrupt Enable (IE_ID=10) oppure un Interrupt Disable (IE ID=11). Se il bit M è asserito allora significa che l'istruzione contiene un cambio di modalità di funzionamento e pertanto negli ultimi 5 bit sarà indicato il relativo codice.

Esempi

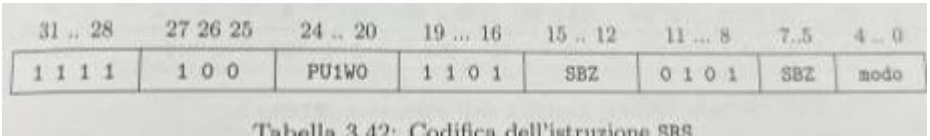
```
CPSIE a,#17 ; abilita i data Abort imprecisi
            ; e cambia il modo FIQ
CPSID if    ; disabilita le interruzioni FIQ e IRQ
CPS   #31   ; imposta il modo di funzionamento a System
```

Store Return State onto a Stack

L'istruzione **SRS** (Store Return State onto a stack), memorizza rispettivamente il LR e SPR dell'attuale modalità di funzionamento, nelle due word successive a partire da (SP) della modalità indicata all'interno dell'istruzione. Grazie a questa istruzione è possibile predisporre lo stato di ritorno da un exception handler su uno stack diverso da quel-
lo utilizzato automaticamente dal processore. Le due sintassi equivalenti dell'istruzione risultano:

SRS{<Modo Agg>}{<PreCond>} SP{!}, #‹Modo>
SRS{<Modo Agg›}{<PreCond>} #‹Modo>{!}

L'istruzione **SRS** si comporta come una STM che però opera sulla coppia di registri: R14 e SPSR del corrente modo di funzionamento. Utilizza la medesima modalità di aggiornamento (IA, IB, DA e DB) e utilizza come registro base il registro R13 relativo alla modalità di funzionamento indicata nel campo numerico <Modo>. Il punto esclamativo permette di aggiornare il registro base con la nuova posizione di memoria ed agisce sul bit di writeback (W). La codifica dell'istruzione SRS risulta:



31 .. 28	27 26 25	24 .. 20	19 ... 16	15 .. 12	11 ... 8	7..5	4 .. 0
1 1 1 1	1 0 0	PU1W0	1 1 0 1	SBZ	0 1 0 1	SBZ	modo

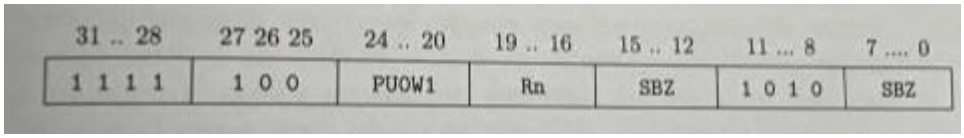
Tabella 3.42: Codifica dell'istruzione SRS

Return From Exception

L'istruzione **RFE** (Return From Exception) permette di ripristinare lo stato del processore salvato da una istruzione di SRS al termine dell'esecuzione di una routine di servizio di una eccezione.
La RFE permette di caricare la coppia di registri PC e CPSR a partire dall'indirizzo contenuto in <Rn> e nella successiva word. La sintassi dell'istruzione risulta:

RFE<Modo Agg> <Rn>{!]

L'istruzione RFE si comporta come una LDM che opera sulla coppia di registri: PC e CPSR. Utilizza la stessa modalità di aggiornamento (IA, IB, DA e DB). Come per la SRS, il punto esclamativo permette di aggiornare il registro base con la nuova posizione di memoria ed agisce sul bit di writeback (W). L'istruzione è codificata nel seguente modo:



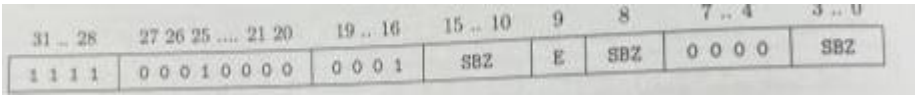
31 .. 28	27 26 25	24 .. 20	19 .. 16	15 .. 12	11 ... 8	7 0
1 1 1 1	1 0 0	PUOW1	Rn	SBZ	1 0 1 0	SBZ

Set Endianess

Questa istruzione permette di modificare l'ordinamento dei byte (big-endian o little-endian) durante le operazioni di accesso in memoria. L'istruzione SETEND è stata introdotta per aumentare l'efficienza delle applicazioni rendendo possibile un accesso ottimizzato in funzione del formato di memorizzazione. Differentemente dalle altre istruzioni, la SETEND non può essere eseguita in modo condizionato e agisce direttamente sul bit E del flag di stato CPSR senza alterarne gli altri bit. Sono possibili due sintassi, la prima per abilitare l'ordinamento big-endian (E=1) e la seconda per il little-endian (E=0):

SETEND (BE|LE}

L’istruzione è codificata nel seguente modo:



il bit 9 contiene il valore che sarà impostato nel bit E del registro CPSR, ovvero E = 1 se l’istruzione è SETEND BE.

Esempio

```
SETEND  BE                ; E=1 accesso Big-Endian
LDR     R0, [R1,#Tabella] ; lettura dati nel formato BE
SETEND  LE                ; E=0 accesso Little-Endian
```