



**POLITECNICO  
DI TORINO**

**III Facoltà di Ingegneria – Laurea Magistrale in Ing. Informatica (Computer Engineering)**

**Architetture dei sistemi di elaborazione – Wikibook**

**ARM – Architettura e assembly**

**ARM<sup>®</sup>**

**Francesco Gavino Brundu, Matteo Ferrari, Guido Francioli, Daniele Milani**

---

## Indice

Indice .....	2
1. Architettura ARM .....	3
1.1 Introduzione e applicazioni [F.B.] .....	3
1.2 Toolchain e Boundary Scan [F.B.] .....	4
2. Assembly ARM .....	6
2.1 Instruction set [D.M., F.B., M.F., G.F.] .....	6
2.2 Interrupt [M.F.] .....	29
2.3 Endianess [F.B., M.F.] .....	33
2.4 Thumb Instruction Set [G.F.] .....	34
Riferimenti .....	36

# 1. Architettura ARM

## 1.1 Introduzione e applicazioni

L'architettura ARM indica una famiglia di microprocessori RISC a 32-bit utilizzata in una moltitudine di sistemi embedded. Grazie alle sue caratteristiche di basso consumo (rapportato alle prestazioni), l'architettura ARM domina il settore dei dispositivi mobili [1] dove il risparmio energetico delle batterie è fondamentale.



**Fig. 1:** un microprocessore ARM (Cortex).

I microprocessori vengono venduti come core, integrabili in System on Chip (SoCs), cioè in particolari circuiti integrati che in un solo chip contengono un intero sistema, o meglio, oltre al processore centrale, integrano anche un chipset ed eventualmente altri controller come quello per la memoria RAM, la circuiteria input/output o il sotto sistema video [2].

Le licenze dei core possono essere di due tipi, a seconda del prezzo e della documentazione fornita assieme all'hardware:

- Hard cores, la strada che la maggior parte delle licenze percorrono; si acquisisce una “scatola nera” che protegge la proprietà intellettuale. Il venditore fornisce un layout fisico fisso e non modificabile; le prestazioni e la densità del progetto sono ottimizzate. Per l’acquirente, gli svantaggi sono la scarsa portabilità del progetto e la massima difficoltà di riuso quando si voglia passare a una nuova tecnologia. La rigidità può creare difficoltà nell’organizzazione fisica del sistema complessivo.
- Soft cores, viene fornita la descrizione circuitale a livello RT o di porta logica. Può anche includere (piccole) sezioni di codice ad alto livello indipendenti dalla tecnologia (utili affinché il progettista possa parametrizzare il core). La proprietà intellettuale è chiaramente più rilevante e determina un costo maggiore rispetto all’hard core.

## 1.2 Toolchain e Boundary Scan

I sistemi operativi utilizzabili sopra un sistema embedded si dividono in:

- Microsoft Windows CE, proprietario
- GNU/Linux, open source
- Symbian OS, open source

La toolchain utilizzata nella programmazione di sistemi embedded si compone in diverse sezioni:

1. inizialmente viene scritto il codice che realizza le funzioni richieste, solitamente in linguaggio assembly oppure attraverso C o C++;
2. tutti i sorgenti vengono cross-compilati in un sistema diverso da quello embedded, attraverso un cross-compiler che tiene conto del Processor Information (ISA) cioè le informazioni sulla macchina target (sistema embedded);
3. a questo punto si crea il file di tipo eseguibile che può seguire due strade: in fase di test il risultato viene debuggato sulla stessa macchina che compila attraverso diversi software che simulano il target; in fase di rilascio il risultato viene caricato sul target attraverso un loader e testato via hardware eventualmente anche con l’uso dei pin che implementano la porta di test JTAG;

Nelle schede è infatti solitamente implementato il protocollo IEEE 1149.1 del consorzio JTAG, un protocollo standard per il test funzionale di dispositivi: infatti, sacrificando in piccola parte le risorse disponibili nel circuito integrato, tramite la porta JTAG è possibile un collaudo razionale della scheda, che sarebbe proibitivo in via manuale per via della crescente complessità dei sistemi.

Esistono diverse implementazioni della toolchain a seconda del sistema nel quale viene effettuata la cross-compilazione:

- per Windows è possibile usare i tools Keil [3], prodotti realizzati ad hoc dal venditore dei SoCs, oppure generici e personalizzabili come openwinice [4];
- su GNU/Linux esiste la sezione ARM della GNU toolchain.

## 2. Assembly ARM

### 2.1 Instruction set

#### 2.1.1 Formato delle istruzioni

Le seguenti caratteristiche contraddistinguono l'Instruction Set utilizzato dall'architettura ARM:

- lunghezza fissa: a differenza dell'architettura x86, nella quale le istruzioni hanno lunghezza variabile, nell'architettura ARM tutte le istruzioni sono lunghe 32 bit; ciò porta alla riduzione dei tempi di decodifica e di esecuzione (nota: a partire dal processore ARM7TDMI l'instruction set prevede anche la presenza di istruzioni a 16 bit denominate Thumb, di cui si parlerà nel Capitolo 10);
- esecuzione: la maggior parte delle istruzioni ARM è eseguita in un solo ciclo; inoltre, per migliorare le prestazioni della pipeline ogni istruzione può o meno essere eseguita (vedi esecuzione condizionale, paragrafo 7 Instruction Set);
- architettura load/store: questo tipo di architettura prevede una divisione tra operazioni di trasferimento dati (load/store) e operazioni logico matematiche: in particolare, le operazioni di load caricano un dato in un registro dalla memoria centrale, le operazioni di store scrivono il contenuto di un registro in memoria centrale, e quelle logico/matematiche operano solo su dati che siano già stati caricati nei registri. Ciò porta alla possibilità di migliorare le performance delle istruzioni aggiungendo alcuni accorgimenti: per quanto riguarda le operazioni logico/matematiche, la presenza di tre operandi (registro destinazione e registri sorgente) riduce sensibilmente i tempi di esecuzione, e, grazie all'azione combinata di ALU e Shifter, le istruzioni di manipolazione dei bit avvengono ad alta velocità; per quanto riguarda, invece, il trasferimento, la differenziazione tra load e store di registri singoli (8-16-32 bit) o multipli fa sì che le istruzioni siano ottimizzate per l'operazione richiesta;

- estensibilità : attraverso coprocessori esterni al core l'Instruction Set può essere esteso; in particolare i coprocessori possono occuparsi di gestire e implementare specifiche caratteristiche (come la gestione della memoria o la gestione delle operazioni in virgola mobile): in questo modo lo sviluppo di core e periferiche può procedere indipendentemente.

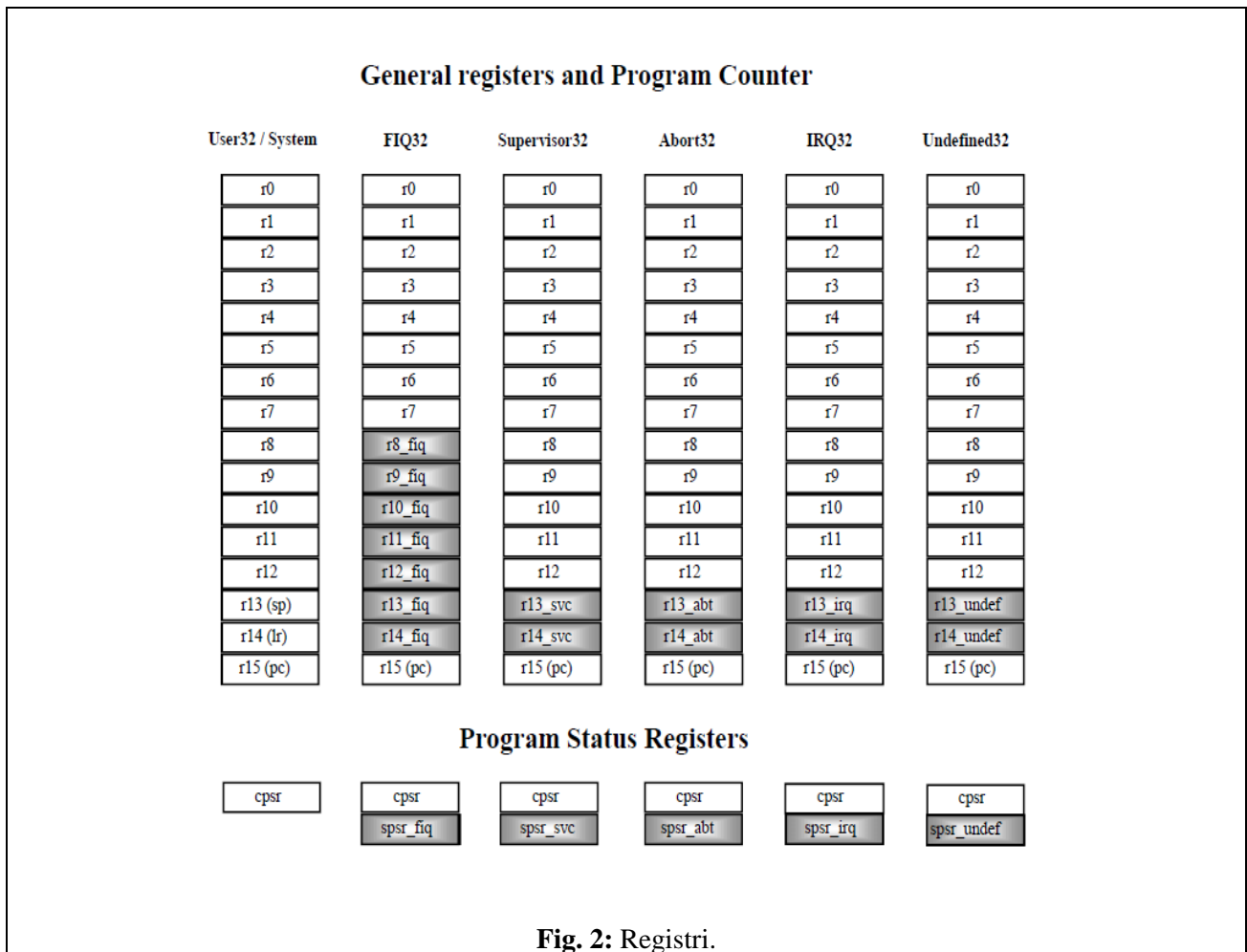
### 2.1.2 Modi del processore

Il processore ARM è nato per poter lavorare in sei differenti modi, anche se, a partire dalla versione 4 dell'architettura, è stato aggiunto un settimo modo (System). Ogni modo gestisce specifiche situazioni, in particolare:

- User, è il modo base di utilizzo, ed è l'unico che non ha nessun privilegio; la maggior parte dei task è eseguita in questo modo, nel quale il sistema garantisce l'isolamento e la protezione dell'applicazione;
- FIQ, il Fast Interrupt Mode è attivato dalla CPU quando viene lanciato un interrupt ad alta priorità (lanciato da una sorgente che il progettista ha segnalato come ad interruzione veloce);
- IRQ, il Normal interrupt Mode è attivato quando viene lanciato un interrupt da una qualsiasi sorgente non segnalata ad interruzione veloce.
- Supervisor, è attivato in seguito ad un interrupt software (una system call); si usa solitamente per invocare il sistema operativo;
- Abort, è attivato per gestire violazioni dei diritti di memoria;
- Undefined, è attivato per gestire il tentativo di esecuzione di istruzioni non supportate nell'architettura, né dai coprocessori collegati;
- System, è un modo privilegiato che usa lo stesso set di registri dello User Mode (vedi Registri, paragrafo successivo); è usato per eseguire compiti privilegiati del sistema operativo.

### 2.1.3 Registri

Un processore ARM ha a disposizione un set di 37 registri lunghi 32 bit, 30 dei quali adoperati per usi generali, 6 come registri di stato, e uno contenente il program counter.



Ogni modo accede a 14 registri di tipo generale, e al program counter (cpsr, in modo condiviso), in particolare:

- i primi 8 registri generali (R0-R7) sono unbanked, cioè sono condivisi da tutti i modi;
- i registri R8-R12 sono condivisi dai modi User, IRQ, Supervisor, Abort, Undef e System;



- ogni modo sopracitato, ad eccezione di User e System, ha a disposizione due registri generali (R13-R14) dedicati;
- i modi User e System condividono anche i due registri R13-R14 (sp, stack pointer e lr, link register);
- il modo FIQ ha a disposizione un set di 7 registri generali (R8-R14) dedicati.

Inoltre, i modi protetti, ad eccezione del modo System, accedono a un registro di stato dedicato, denominato spsr, del quale si parlerà nel capitolo successivo. Per quanto riguarda l'accessibilità dei registri, invece, i registri di tipo generale sono accessibili direttamente da tutte le istruzioni, la maggior parte delle quali può accedere anche al program counter; invece i registri SPSR e CPSR, dei quali parleremo dopo, sono accessibili solamente attraverso istruzioni specifiche.

#### **2.1.4 Program Status Register – Flag di condizione**

Nei 32 bit dei registri CPSR (Current program status register) e SPSR (Saved program status register) sono contenute le informazioni relative allo stato della CPU nel momento attuale o nel momento in cui è stata lanciata l'interruzione (vedi eccezioni e vector table); in particolare:

- i bit 0-4 definiscono il modo del processore;
- il bit 5 (T-bit) è settato a uno se il processore è nello stato Thumb, a 0 se è nello stato Arm (vedi istruzioni thumb);
- i bit 6 e 7, detti Interrupt Disable bit, servono per abilitare o disabilitare FIQ (bit 6) e IRQ (bit 7);
- i bit 8-28 sono riservati.

I bit dal 28 al 31, infine, servono per ottenere informazioni sulle operazioni eseguite dalla ALU, e possono avere significato diverso a seconda del tipo dell'ultima operazione eseguita (logico o aritmetico); in particolare:

- il bit 28 (V-bit) indica se l'ultima operazione eseguita dalla ALU ha avuto, o meno, un overflow, e non ha alcun significato se l'ultima operazione è stata di tipo logico;
- il bit 29 (C-bit) denota il Carry dovuto all'ultima operazione: in caso di istruzione aritmetica indica che il risultato ha superato i 32 bit, in caso di istruzione di shift logico indica che un bit "1" è stato shiftato nel carry flag;

- il bit 30 (Z-bit) è una copia del flag di Zero della ALU, in caso di operazione aritmetica è settato a 1 se il risultato dell'operazione è stato 0, in caso di operazione logica è settato a 1 se il risultato dell'operazione su ogni singolo bit è stato 0;
- il bit 31 (N-bit), infine, è una copia del flag della ALU che indica se l'ultima operazione ha avuto risultato Negativo, e non ha alcun significato se l'ultima operazione è stata di tipo logico.

### 2.1.5 Gestione delle eccezioni e vector table

Quando viene lanciata un'eccezione, il processore, per permettere il corretto funzionamento, fa le seguenti operazioni:

- copia il registro di stato attuale nel registro SPSR del modo che è chiamato a gestire l'eccezione;
- setta il CPSR, cambiando i bit di modo e abilitando/disabilitando determinati interrupt;
- imposta i registri banked appropriati;
- mette l'indirizzo di ritorno nel R14-lr (link register);
- carica il Program Counter col valore appropriato.

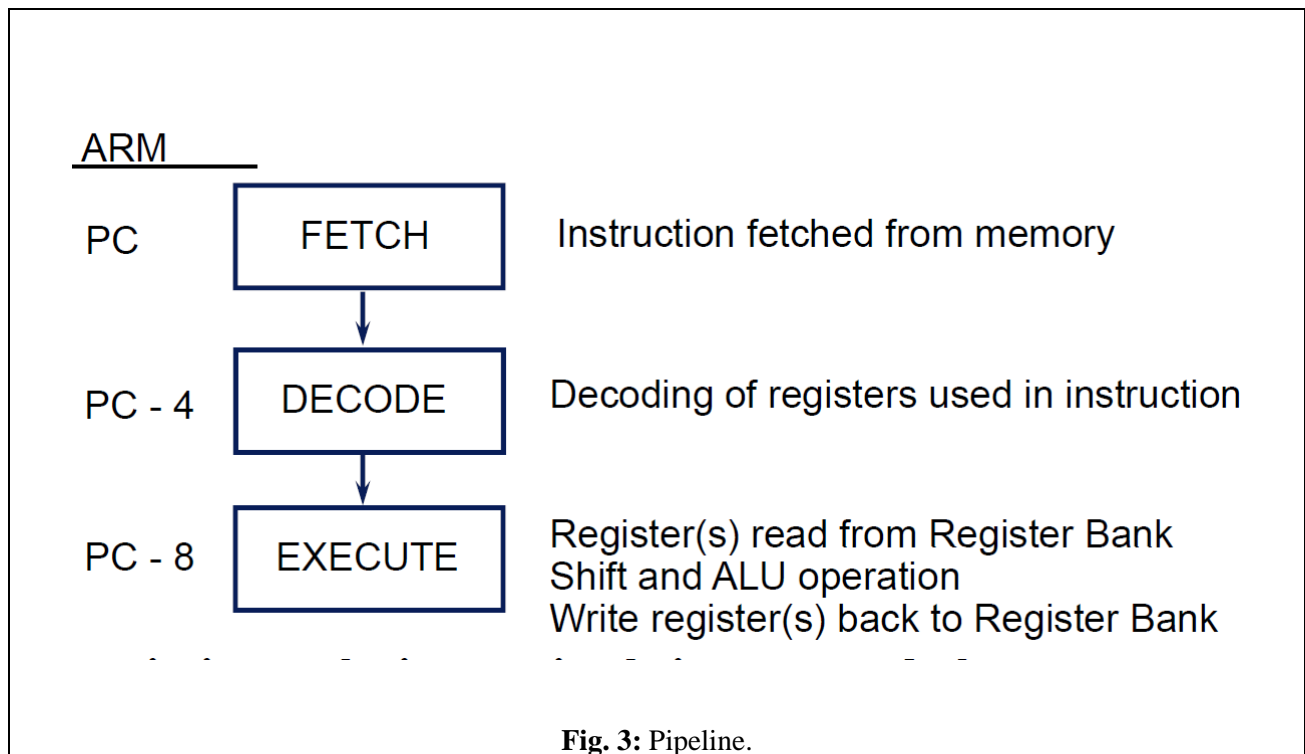
Una volta finita la gestione dell'interruzione, il processore ripristina i valori precedenti di CPSR (caricandolo dal SPSR) e PC (caricandolo dall'lr).

### 2.1.6 Pipeline

Per aumentare le prestazioni, nell'architettura ARM è stata inizialmente inserita una pipeline a tre stadi (fetch, decode, execute); nella fase di fetch il program counter punta all'istruzione da decodificare, invece che puntare all'istruzione in esecuzione in quel momento; in questa fase viene caricata l'istruzione da memoria; nella fase successiva, quella di decodifica, vengono decodificati i registri interessati dall'operazione.

L'ultima fase è quella nella quale viene effettivamente eseguita l'istruzione: inizialmente sono letti i registri interessati dal Register Bank; successivamente sono effettuate le operazioni di ALU/Shift e infine il risultato è scritto nel registro interessato, e i flag sono aggiornati.

Al giorno d'oggi i processori più evoluti prevedono l'utilizzo di pipe con molti più stadi (ad esempio il processore Cortex-A8 ha 13 stadi).



### 2.1.7 Istruzioni di Branch

Le istruzioni di Branch e di Branch con Link (B, BL) sono il modo standard per cambiare la sequenza delle istruzioni da eseguire ovvero di effettuare un salto verso una determinata porzione di codice (come le istruzioni di JMP nell'assembly x86); la sintassi è la seguente:

`B{L}{<cond>} LABEL`

dove L (opzionale) indica che l'operazione è di Branch con Link, {<cond>} è la condizione che deve essere soddisfatta affinché l'operazione di Branch venga eseguita, LABEL è l'etichetta della posizione del codice verso il quale saltare.

La variante di Branch con Link copia l'indirizzo successivo all'istruzione di Branch nel link register (r14), permettendo quindi il ritorno a questa effettuando la copia del link register nel program counter attraverso l'istruzione:

```
MOV          PC, r14
```

oppure

```
MOV          PC, LR
```

Questa variante viene utilizzata quando è necessario richiamare una subroutine: il programma chiamante effettua un'operazione di Branch con Link utilizzando l'etichetta (LABEL) della funzione; all'interno di quest'ultima, come ultima istruzione, si pone la copia del link register nel program counter attraverso l'istruzione precedentemente descritta.

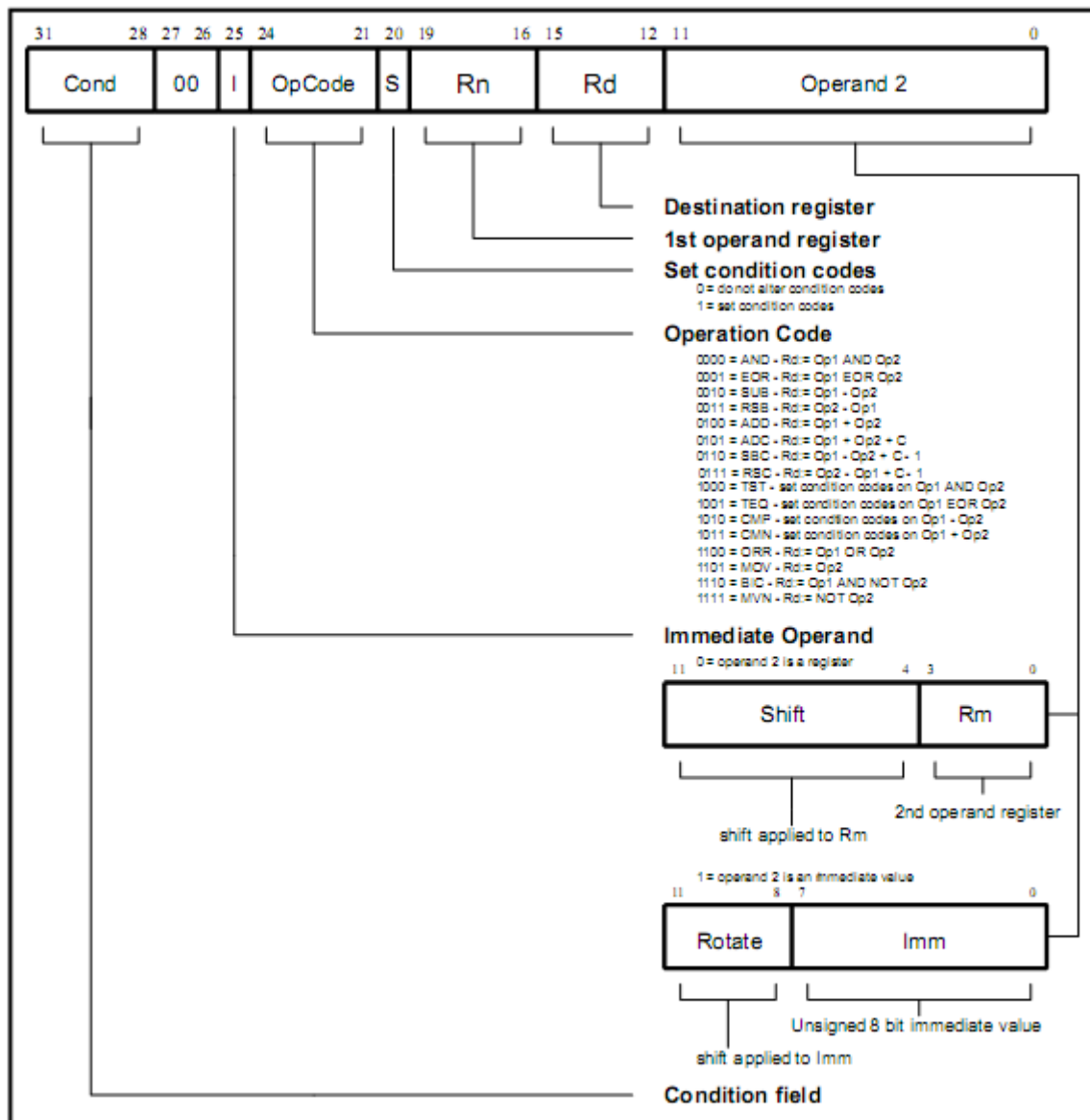
L'offset per le istruzioni di Branch viene calcolato dall'assemblatore in questo modo:

- si fa la differenza tra l'indirizzo dell'istruzione di arrivo e quello dell'istruzione di Branch e si sottrae 8;
- si ottiene quindi un indirizzo di 26 bit;
- i 2 bit meno significativi sono sicuramente 0 in quanto le word di istruzione (che sono su 32 bit/4 byte) sono allineate, e quindi il loro indirizzo risulta sempre un multiplo di 4 (byte);
- l'indirizzo ottenuto di 24 bit viene codificato nei bit meno significativi dell'istruzione e permette un range di salto da -32 a +32 Mbytes.

Al momento dell'esecuzione dell'istruzione il processore prende l'offset codificato nell'istruzione, aggiunge 2 bit a 0 a destra (moltiplica per 4 l'offset), estende l'offset su 32 bit (dai 26 del passo precedente) tenendo conto del segno, ed infine aggiunge l'offset al program counter. A questo punto l'esecuzione riprende dal nuovo PC, una volta riempita la pipeline.

### **2.1.8 Istruzioni aritmetiche, logiche, di spostamento**

Varie istruzioni destinate al Data Processing condividono lo stesso formato d'istruzione che qui riportiamo:



**Fig. 4:** Codifica delle istruzioni di data processing

Le istruzioni per il processo dei dati si dividono in:

- Operazioni Aritmetiche
- Operazioni di Comparazione
- Operazioni Logiche e di movimento

Queste istruzioni producono un risultato eseguendo specifiche operazioni aritmetiche o logiche tra uno o due operandi. Il primo operando è sempre un registro. Il secondo operando può essere uno "shifted register" cioè inviato dall'ALU via Barrel Shifter. Queste istruzioni lavorano sempre su registri e non con locazioni di memoria.

Lo stato codificato nel CPSR possono essere preservate o aggiornate mediante il risultato di queste istruzione mediante il valore dell'S-bit dell'istruzione.

Le operazioni aritmetiche trattano ogni operando come un intero a 32bit. La sintassi è la seguente:

`<Operation>{<cond>}{S} Rd, Rn, Operand2`

Se l'S bit è settato (e Rd non è R15):

- V flag → viene settato se un si ha un overflow sul bit 31 del risultato. Questo flag può essere ignorato se gli operandi sono considerati unsigned, ma è segno di un possibile errore se gli operandi sono considerati in complemento a 2;
- C flag → viene settato dal carry out del bit 31 dell'ALU;
- Z flag → viene settato solo se il risultato è 0;
- N flag → viene settato come il bit 31 del risultato (questo flag ci permette di sapere subito se il risultato è un numero positivo o negativo se gli operandi sono considerati in complemento a 2.

Le operazioni di comparazione non scrivono il risultato in Rd. Esse sono usate solamente per eseguire dei test e per settare il relativo stato nel CPSR e hanno sempre l'S bit settato.

Le operazioni logiche eseguono delle operazioni bit a bit. Se l'S bit è settato(e Rd non è R15):

- V flag → rimarrà inalterato
- C flag → viene settato dal carry out dal barrel shifter (o preservato quando l'operazione di shift è "LSL #0)
- Z flag → viene settata solamente se il risultato è 0

- N flag → viene settato al valore logico del bit 31 del risultato.

Di seguito riportiamo una tabella riassuntiva delle istruzioni di Data Processing

Assembler mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

**Fig. 5:** Codifica delle istruzioni di data processing e significato

L'istruzione set prevede 2 istruzioni di moltiplicazione:

- Multiply

Sintassi: `MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs`

Rn se inserito viene ignorato.

- Multiply Accumulate

Sintassi: `MLA{<cond>}{S} Rd, Rm, Rs, Rn ; Rd = (Rm * Rs) + Rn`

Quindi è possibile "risparmiare" un'operazione di ADD quando necessario.

I risultati di una moltiplicazione di tipo signed o di tipo unsigned con operandi di 32 bit differiscono soltanto per i 32 bit alti; i 32 bit bassi sono identici. Queste due istruzioni producono quindi soltanto i 32 bits bassi della moltiplicazione per questo possono essere usate sia per signed che unsigned.

Esempio: consideriamo la moltiplicazione di questi due operandi:

Operand A	Operand B	Result
0xFFFFFFFF6	0x00000014	xFFFFFFFF38

Se gli operandi sono considerati signed abbiamo: A = -10, B = 20 e Risultato = -200 che è correttamente espresso dal valore 0xFFFFFFFF38.

Se invece interpretiamo gli operandi unsigned: A = 4294967286, B = 20 e il risultato sarebbe 0x13FFFFFF38 ma dato che vengono presi soltanto i 32 bit più bassi Result sarà 0xFFFFFFFF38.

A causa del modo con cui vengono effettuate le moltiplicazioni, alcune combinazioni tra registri devono essere evitate (l'assemblatore darà un warning se queste restrizioni sono trascurate): il registro destinazione (Rd) non deve essere uguale al registro operando (Rm) perchè Rd vien usato per mantenere dei valori intermedi ed Rm è utilizzato ripetutamente durante la moltiplicazione. Se Rm=Rd la MUL darà un valore 0 invece una MLA ritornerà un valore senza senso logico. Il registro R15 non deve essere usato in nessun modo. Tutte le altre combinazioni di registri daranno un risultato corretto e Rd, Rn e Rs possono essere gli stessi registri se richiesto.



Esempi:

MUL R1,R2,R3 ;R1:=R2\*R3

MLAEQS R1,R2,R3,R4 ; previa condizione esegue R1:=R2\*R3+R4 settando i flag in CPSR

### 2.1.9 Valori immediati e Literal pool

Nonostante tutte le istruzioni ARM hanno lunghezza da 32 bit, non è possibile caricare un valore immediato di 32 bit. Il formato delle istruzioni rende disponibile 12 bit, ovvero un valore massimo di 4096. Nella realtà, si memorizzano costanti a 8 bit e si utilizza il barrel shifter per estendere il loro range. (ROR di 0, 2, 4, ... 30). Questo dà un largo range di costanti che possono essere direttamente caricate.

I possibili range sono:

- 0 - 255 [0 - 0xff]
- 256,260,264,...,1020 [0x100-0x3fc, step 4, 0x40-0xff ror 30]
- 1024,1040,1056,...,4080 [0x400-0xff0, step 16, 0x40-0xff ror 28]
- 4096,4160, 4224,...,16320 [0x1000-0x3fc0, step 64, 0x40-0xff ror 26]

Il range standard va da 0 a 255. Se volessi un range più elevato, ad esempio a partire da 256, potrei moltiplicare per 4 (ror 30). L'insieme dei valori non risulta però continuo, ma a buchi, proprio a causa della moltiplicazione. Nell'esempio ho moltiplicato per 4 perciò avrò valori distanziati di 4 unità: 256, 260, 264, 268...

Tramite l'istruzione LDR è possibile forzare l'acquisizione di un numero di lunghezza maggiore di 32 bit all'interno di un registro. La sintassi dell'istruzione è la seguente:

```
LDR rd,=numeric constant
```

Esempi:

```
LDR r0,=0x42 ; genera MOV r0,#0x42
```

```
LDR r0,=0x55555555 ; genera LDR r0,[pc, offset literal pool]
```

L'istruzione funziona mediante il meccanismo di "literal pool". Nella fase di compilazione, allorchè trovo un valore da caricare in un registro di lunghezza maggiore a 8 bit, il compilatore alloca il numero nella zona di memoria ove vi è il pool e nell'esecuzione lo andrà a riprendere per trasportarlo nel registro.

### 2.1.10 Istruzioni di Load e Store

L'architettura ARM presenta una particolarità che la differenzia notevolmente da quella x86: ogni dato per essere utilizzato deve essere obbligatoriamente presente in un registro; in tal modo non è possibile effettuare operazioni su un dato in memoria ma occorre prima di tutto caricare il dato in un registro, ed in seguito all'operazione voluta salvarlo all'interno della locazione di memoria prestabilita.

Esistono 3 tipi di istruzioni che operano con la memoria:

- trasferimenti di singoli registri di dato (LDR/STR)
- trasferimenti di blocchi di dati (LDM/STM)
- scambi di dati singoli (SWP)

Nel caso di trasferimenti di singoli registri esistono le operazioni di Load:

LDR/LDRB

che caricano una word o un byte di dato in un registro, e le operazioni di Store:

STR/STRB

che salvano in memoria una word o il byte meno significativo di un registro.

La versione 4 dell'architettura ARM ha introdotto anche il supporto per dati di 16 bit (halfword):

LDRH/STRH

e per l'estensione del segno per dati di 16 bit (halfword) o 8 bit (byte):

LDRSB/LDRSH

La sintassi generale è del tipo:

```
<LDR|STR>{<cond>}{<size>} Rd, <address>
```

dove il primo campo indica l'operazione, <cond> indica la condizione che deve essere soddisfatta affinché l'operazione venga eseguita, <size> indica la dimensione del dato, Rd è il registro con il quale si sta operando, <address> è l'indirizzo della locazione di memoria sorgente o destinazione acceduto tramite un registro (Base Register). Per esempio un'operazione di Store del registro r0 in una locazione di memoria il cui indirizzo è contenuto il r1 si scrive:

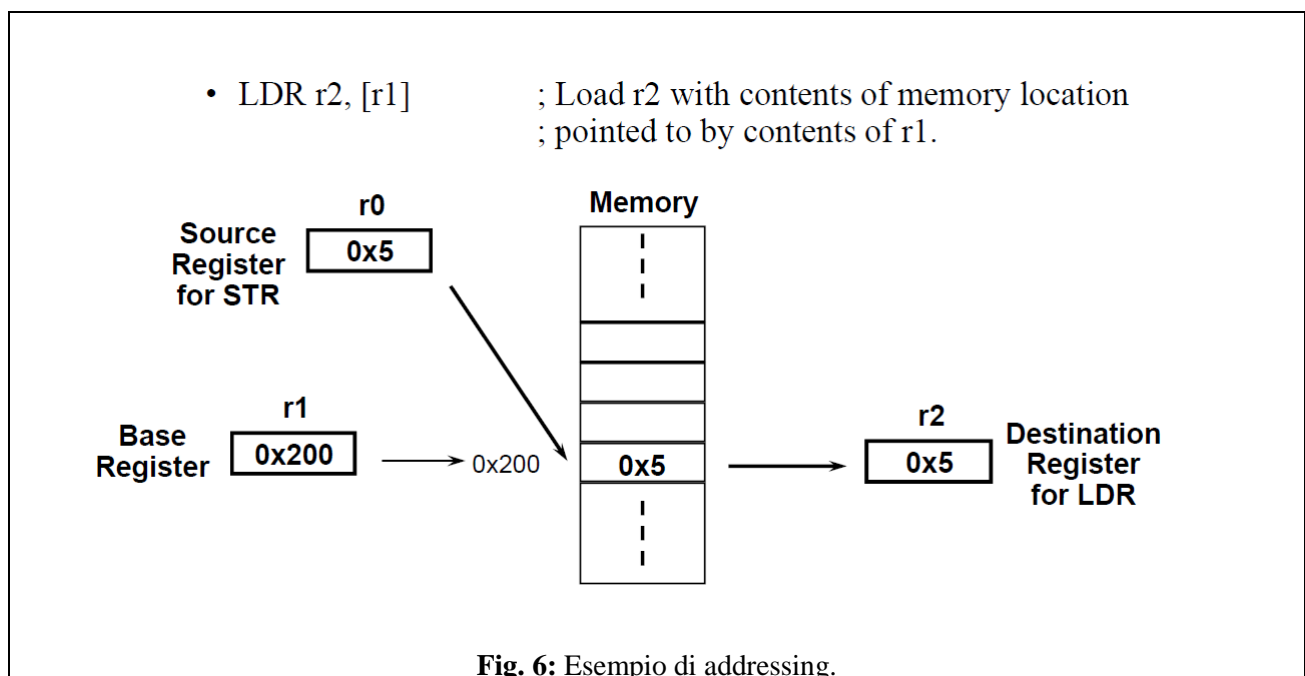
```
STR          r0, [r1]
```

### 2.1.11 Addressing

Il meccanismo di scrittura in memoria è simile al quello di indirizzamento indiretto dell'8086. La locazione di memoria su cui dobbiamo accedere è contenuta in un registro di base.

Esempi:

```
STR r0, [r1]      ;Store contents of r0 to location pointed to by contents of r1.
LDR r2, [r1]      ;Load r2 with contents of memory location pointed to by contents of r1.
```



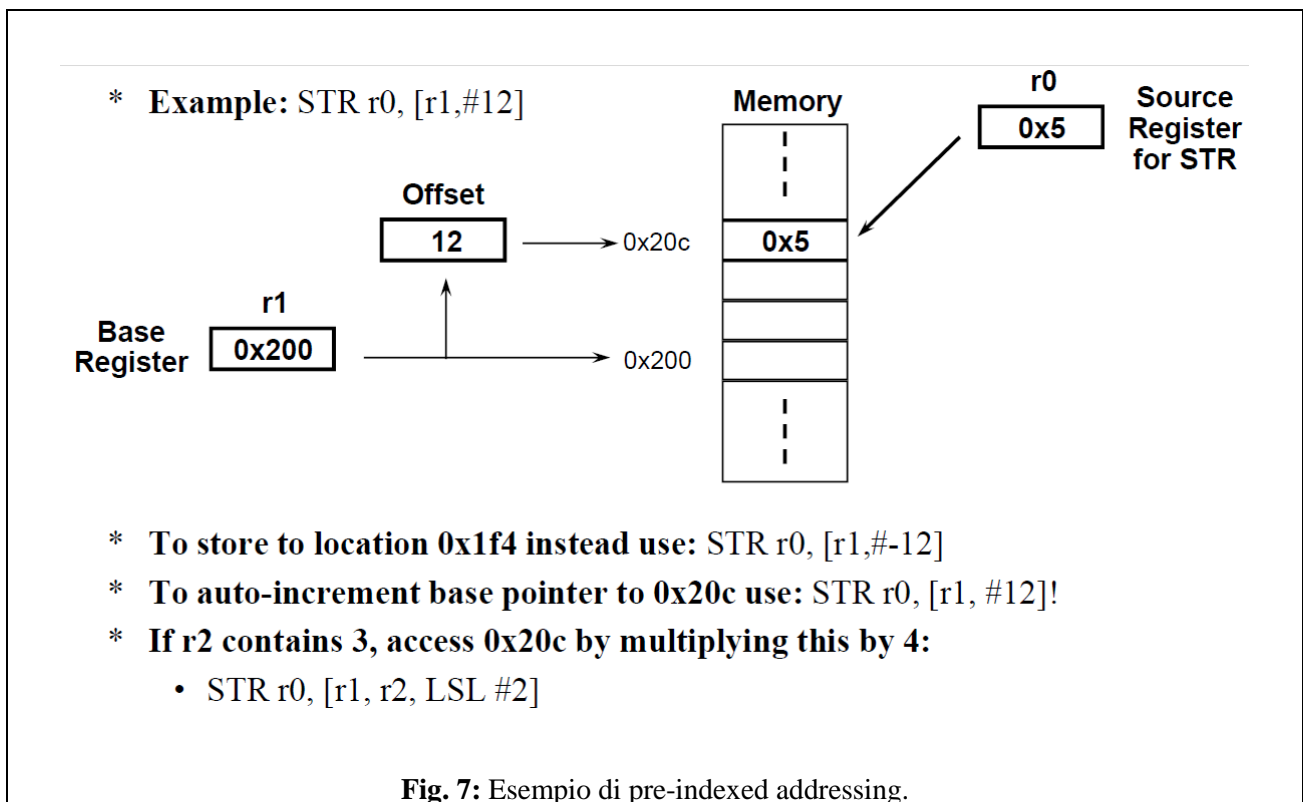
In questi due casi all'interno di r1 si ha l'indirizzo della memoria in cui noi vogliamo accedere.

### *Load and Store Word or Byte: Offsets from the Base Register*

Con queste istruzioni abbiamo la possibilità di accedere in memoria ad una posizione dovuta al base register sommato ad un offset. Quest'ultimo può essere un valore immediato unsigned su 12 bit (0 - 4095 bytes) oppure un altro registro che può essere anche shiftato da un valore immediato. L'offset può essere sommato (default) o sottratto al base register premettendo all'offset il carattere + o -.

Se l'offset viene applicato prima che il trasferimento sia compiuto abbiamo il Pre-Indexed addressing (opzionalmente possiamo effettuare l'auto incremento del base register mettendo alla fine dell'istruzione il carattere "!" che effettua un'operazione di aggiornamento del base register); altrimenti se viene fatto dopo il trasferimento abbiamo il Post-indexed addressing.

### *Pre-indexed Addressing*



Se vogliamo effettuare un'operazione di store alla locazione 0xF4 possiamo usare

```
STR r0, [r1, #-12]
```

Per auto incrementare il puntatore base ad 0x20c possiamo usare

```
STR r0, [r1, #12]!
```

Come funziona il "!"?

Se presente attiva il "write back" del registro di base.

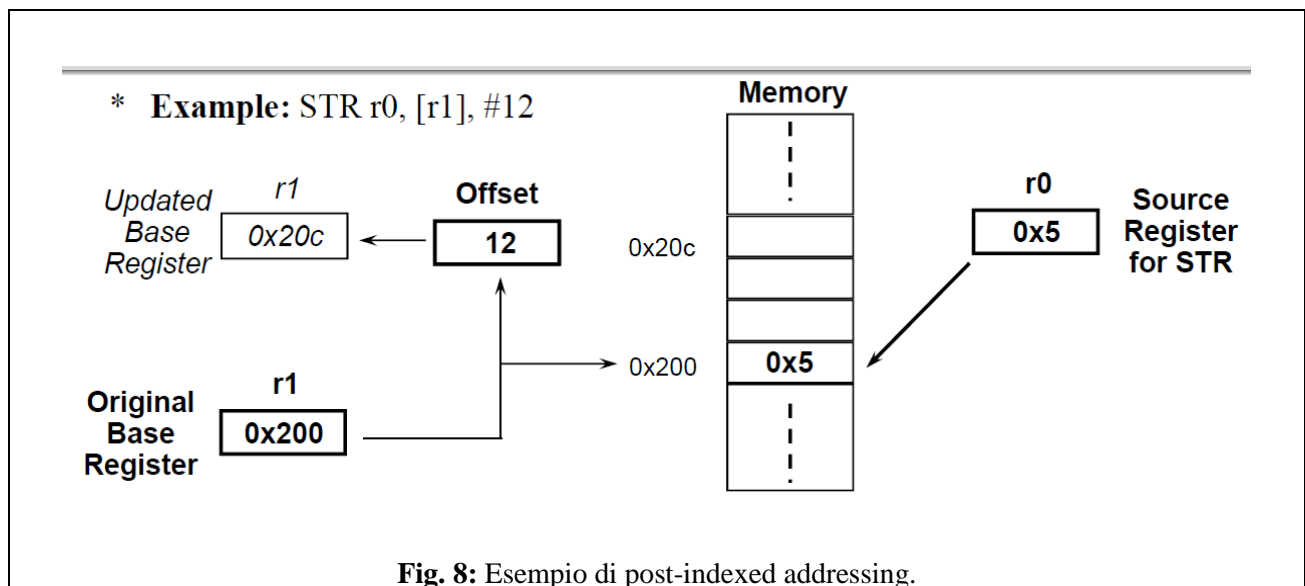
Esempio:

```
STR R1,[R2,R4]!      ;store di R1 a R2+R4 e write back di questo indirizzo (R2+R4) in R2
```

Uso del barrel shifter: se per esempio r2 contiene 3 e noi vogliamo accedere alla locazione 0x29c dobbiamo moltiplicare 3 con 4. Possiamo fare questo con questa istruzione:

```
STR r0 [r1, r2, LSL #2]    ; Store di r0 in r1+(r2*4)
```

### *Post-indexed Addressing*



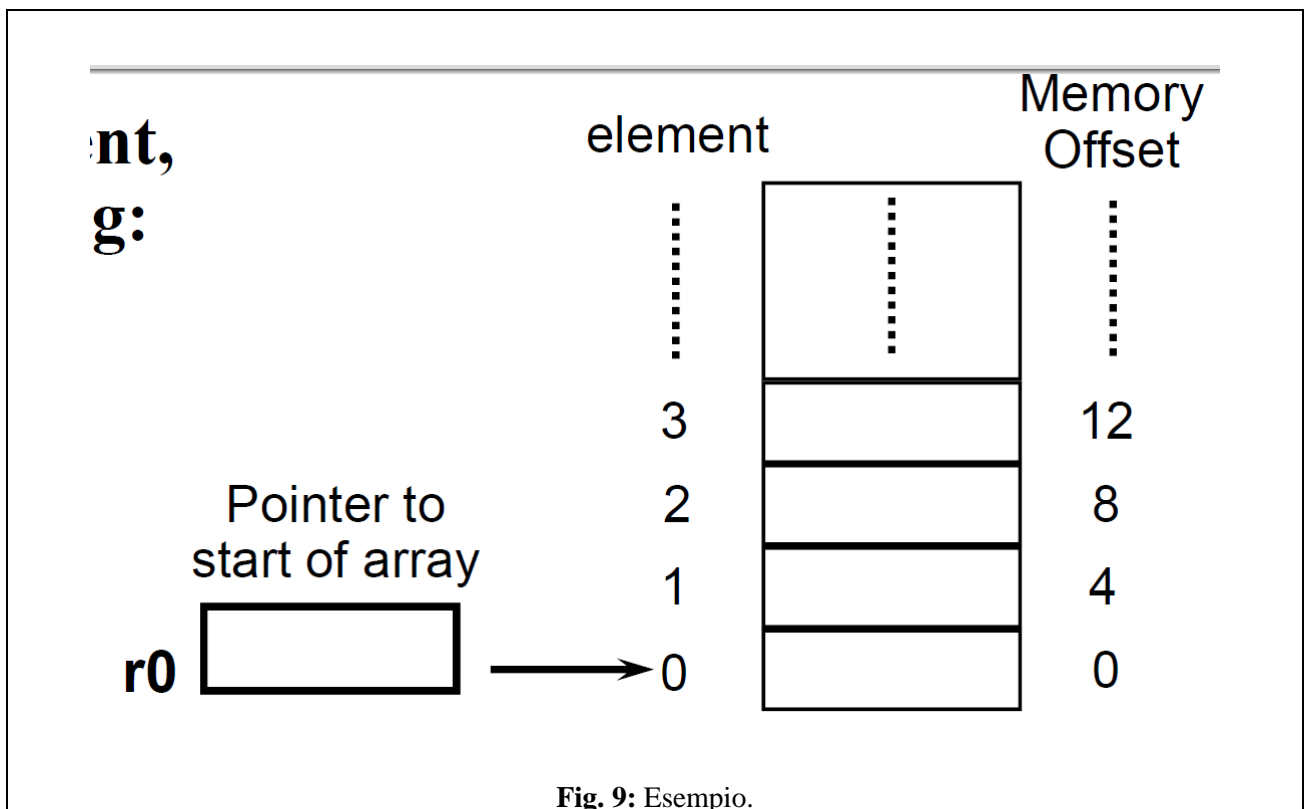
Se vogliamo auto incrementare il registro base alla locazione 0x1F4 possiamo utilizzare: `STR r0, [r1], #-12`

Con il barrel shifter: se r2 contiene 3 possiamo auto incrementare il registro base fino a 0x20c moltiplicando questo valore per 4 con:

```
STR r0, [r1], r2, LSL #2
```

Esempio di usi Pre e post indexed addressing.

Immaginiamo di avere un array dove il primo elemento è puntato dall'indirizzo contenuto in r0.



Se noi vogliamo accedere ad un particolare elemento dell'array con indice contenuto in r1 ci basta eseguire questa istruzione (che fa uso del barrel shifter):

```
LDR r2, [r0, r1, LSL #2] ;Load in r2 del valore presente nel vettore puntato da r0 all'indice r1
```

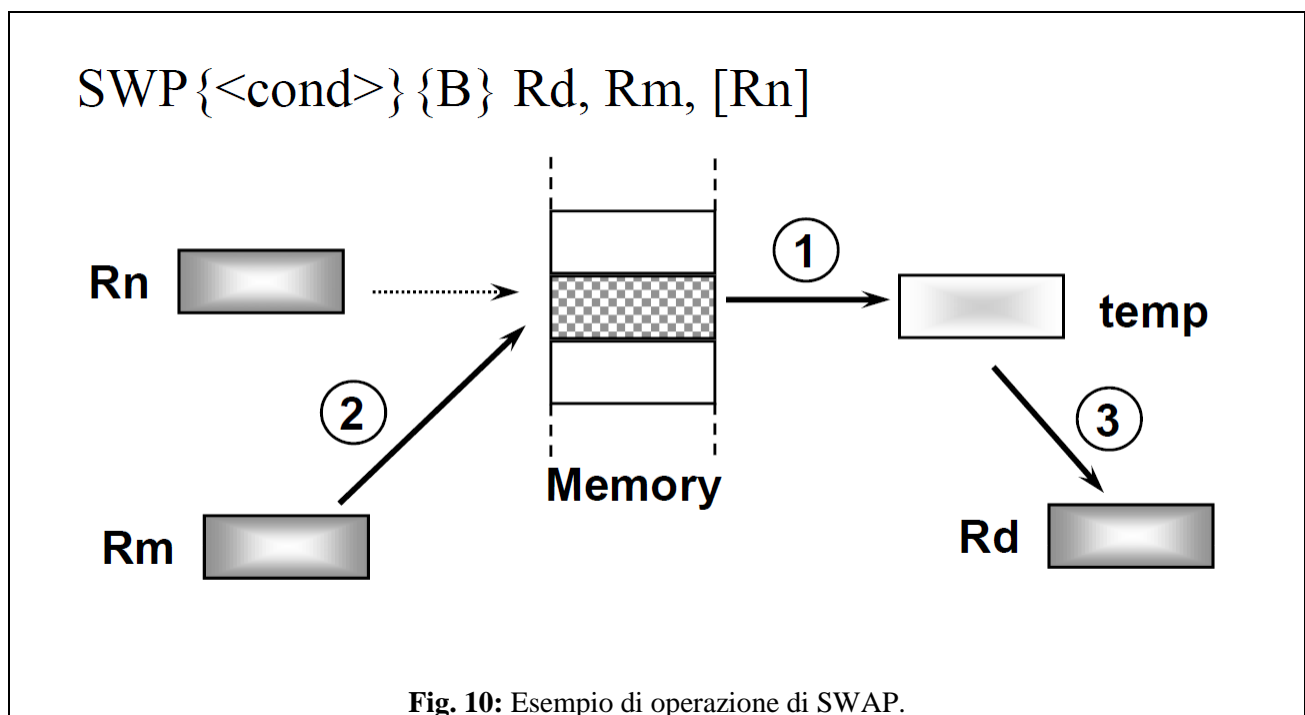
Se vogliamo invece scandire ogni elemento dell'array possiamo utilizzare il post-indexed addressing all'interno di un loop. In r1 memorizziamo in ogni ciclo l'indirizzo dell'elemento corrente (ovviamente al primo ciclo r1 sarà uguale a r0):

LDR r2, [r1], #4 ;Load in r2 del valore puntato da r1. In seguito fai  $r1 = r1 + 4$ .

Ovviamente ci dobbiamo ricordare di memorizzare l'indirizzo dell'elemento finale per poter uscire correttamente dall'array.

### 2.1.12 Istruzioni di Swap

L'istruzione SWP viene usata per scambiare un byte o una parola tra registri e memoria esterna. Questa istruzione è implementata come una lettura da memoria seguita da una scrittura in memoria entrambe "bloccati" e rese atomiche (il processore non può essere interrotto fino a quando entrambe le operazioni non sono state completate, e il gestore di memoria è avvisato di trattarli come inseparabili). Istruzione molto utile per l'implementazione di semafori



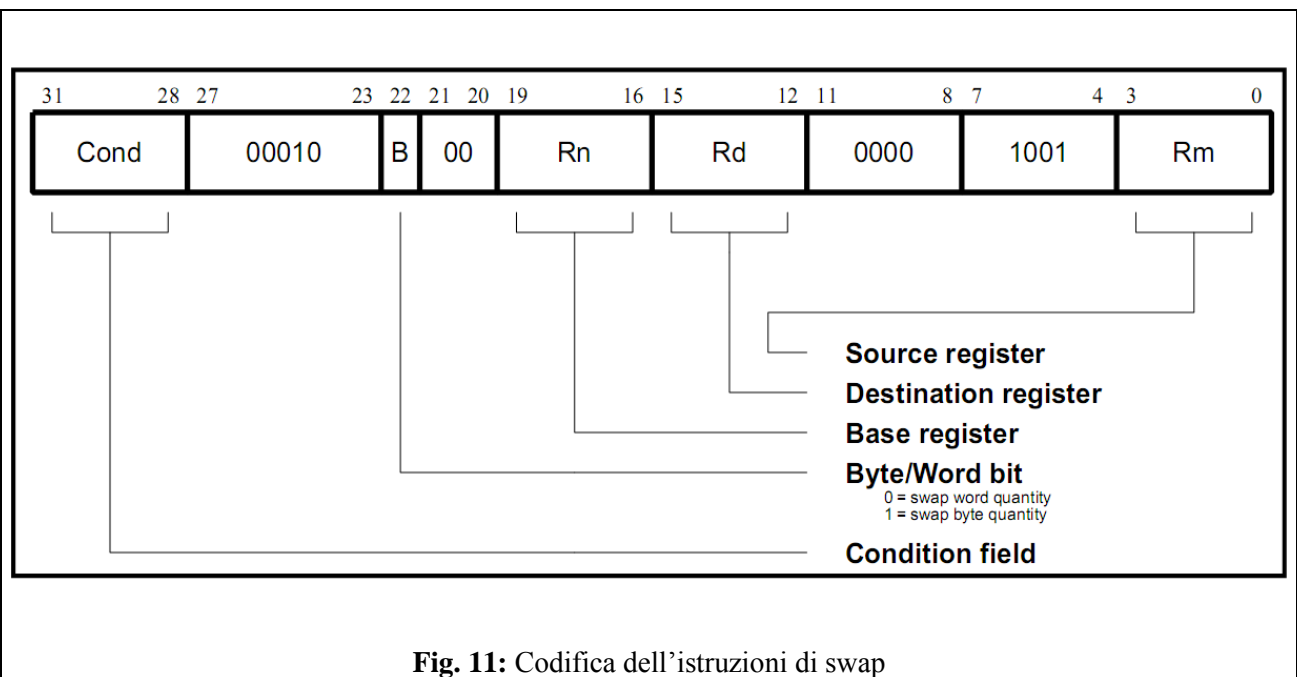
Syntax:

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

{cond} two-character condition mnemonic,

{B} se B è presente si ha un trasferimento di un byte, altrimenti di una parola

Rd,Rm,Rn registri tra cui effettuare lo swap



Esempi:

SWP R0,R1,[R2] ;load R0 with the word addressed by R2, and store R1 at R2

SWPB R2,R3,[R4] ;load R2 with the byte addressed by R4, and store bits 0 to 7 of R3 at R4

SWPEQ R0,R0,[R1] ;conditionally swap the contents of R1 with R0

Se Rd = Rm si ha un normale swap dei contenuti.



### 2.1.13 Trasferimenti di blocchi di dato

Esistono istruzioni generiche che permettono di accedere alla memoria laddove sia leggibile/scrivibile in blocco. Queste istruzioni sono molto efficienti per:

- salvare e ripristinare il contesto (stack)
- trasferire grossi blocchi di dati nella memoria.

Quando operazioni di load e store non possono essere utilizzate con lo stack, possiamo usare delle istruzioni specifiche per l'accesso alla memoria. Le istruzioni disponibili, in ordine, sono:

- STMIA / LDMIA : Increment After (incremento dopo)
- STMIB / LDMIB : Increment Before (incremento prima)
- STMDA / LDMDA : Decrement After (decremento dopo)
- STMDB / LDMDB : Decrement Before (decremento prima)

*Esempio: copia di un blocco*

Nell'esempio viene riportata la copia del blocco di memoria puntato da r12 a quello puntato da r13. R14 punta la fine del blocco da copiare. Questo ciclo trasferisce 48 byte in 31 cicli.

NB: Il "!" indica un autoaggiornamento: a seguito dell'operazione di load, r12 si deve aggiornare e andare all'indirizzo successivo all'ultimo che è stato copiato.

```
; r12 punta l'inizio dei dati da copiare
; r14 punta la fine dei dati da copiare
; r13 punta l'inizio della zona di memoria di destinazione
loop LDMIA r12!, {r0-r11} ; load 48 bytes
    STMIA r13!, { r0-r11} ; and store them
    CMP r12, r14 ; check for the end
    BNE loop ; and loop until done
```

### 2.1.14 Stack

Lo stack è un'area di memoria LIFO (Last In First Out) che cresce quando nuovi dati vengono immessi “push” e decresce quando vengono tolti “pop”. L'ultimo valore immesso è sarà il primo ad essere tolto. Lo stack è definito da due puntatori:

- BASE: usato per puntare la locazione di partenza dello stack
- SP (Stack Pointer): usato per puntare alla cima dello stack.

Tradizionalmente lo stack cresce verso il "basso" nella memoria, ma l'ARM supporta sia l'inserimento verso il basso sia quello verso l'alto.

Un'altra distinzione riguarda la posizione dello stack pointer che può puntare:

- all'ultima posizione occupata (Full stack), e quindi viene pre-decrementato (prima della push)
- alla prima posizione libera (Empty stack) e quindi viene post-decrementato (dopo la push)

Le istruzioni utilizzabili per gli stack sono delle store e load:

- STMFD / LDMFD : Full Descending stack (in discesa)
- STMFA / LDMFA : Full Ascending stack (in ascesa)
- STMED / LDMED : Empty Descending stack (in discesa)
- STMEA / LDMEA : Empty Ascending stack (in ascesa)

**Nota:** il compilatore ARM usa sempre il Full descending stack.

#### *Stack e subroutine*

Uno degli usi principali dello stack è il temporaneo salvataggio di registri per le subroutines. I registri che devono essere preservati, possono essere memorizzati nello start dalla subroutine e caricati alla fine di essa.

Template di una subroutine:

```
STMFD sp!,{r0-r12, lr} ; tutti i registri nello stack
.....
.....

LDMFD sp!,{r0-r12, pc} ; estrae dallo stack tutti i registri
; e ritorna automaticamente
```

Nota: usare '^' al fondo di una istruzione per eseguirla in modalità supervisor, modalità privilegiata rispetto a quella utente poichè permette di eseguire operazioni altrimenti potenzialmente mascherate in certe situazioni dalla modalità utente.

### 2.1.15 Barrel shifter e shift

L'ARM non ha le istruzioni di shift esplicite. L'ARM permette di fare delle operazioni aritmetiche più evolute e lo fa utilizzando il **barrel shifter**, un registro che è posto a monte della ALU rispetto al secondo operando.

Se interpellato, il barrel manipola l'operando 2 appena prima di eseguire l'operazione aritmetica con l'operando 1.

Le etichette che vanno messe alla fine di una istruzione sono:

- Logical Shift Left: LSL #5 : shift a sinistra di alcuni bit dell'op2 (moltiplica per potenze di 2, nell'esempio per 32)
- Logical Shift Right: LSR #5 : shift a destra di alcuni bit dell'op2 (divide per potenze di 2, nell'esempio per 32)
- Arithmetic Shift Right: ASR #5: shift a destra di alcuni bit dell'op2 (divide per potenze di 2, specifico per i numeri in complemento a 2)
- Rotate Right: ROR #5 : ruota il dato e mette l'LSB nel Carry flag

- Rotate Right Extended: ROX #5 : ruota il dato, e il ciclo di rotazione coinvolge anche il CF.

### *L'utilizzo del barrel shifter*

Un'istruzione di moltiplicazione per una costante significa caricare la costante in un registro ed aspettare un numero di cicli interni fino al completamento dell'istruzione. Una soluzione ottimizzata è ottenuta utilizzando combinazioni tra istruzioni MOVs, ADDs, SUBs e RSBs ed il barrel shifter.

La moltiplicazione per costanti uguali a  $((\text{potenza di } 2) \pm 1)$  può essere svolta in un ciclo.

Esempio:

$$r0 = r1 * 5$$

$$= r1 + (r1 * 4)$$

ADD r0, r1, r1, LSL #2

Esempio2:  $r2 = r3 * 105$

$$= r3 * 15 * 7$$

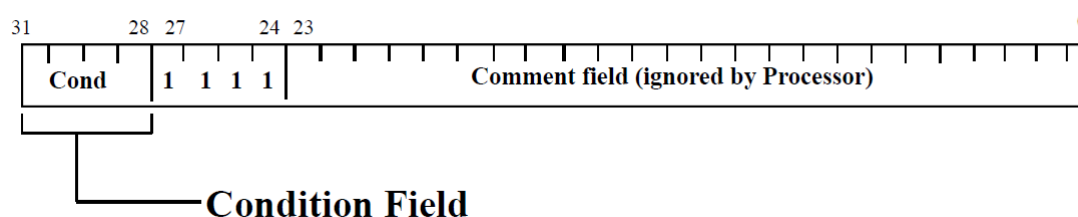
$$= r3 * (16 - 1) * (8 - 1)$$

RSB r2, r3, r3, LSL #4 ;  $r2 = r3 * 15$

RSB r2, r2, r2, LSL #3 ;  $r2 = r2 * 7$

## 2.2 Interrupt

### *Software Interrupt (SWI)*



**Fig. 12:** Codifica interrupt.

Il software interrupt è simile all'INT nell'8086. All'istruzione SWI (software interrupt) segue un codice che indica l'indice dell'interruzione che noi vogliamo eseguire. Nella pratica la SWI è una istruzione User-defined e cioè sta al programmatore gestire le chiamate alle varie routine. Quando si esegue questa istruzione si va a saltare nella tabella dei vettori, si recupera un indirizzo all'interno della Vector Table, e si ha il passaggio di contesto da modo utente (che è il modo in cui facciamo funzionare i nostri programmi) a modo supervisor. Questa istruzione ci mette in uno stato privilegiato che ci permette di eseguire operazioni, per esempio di manipolazione del PC, che altrimenti in USER MODE non sempre è possibile fare.

Cosa succede in pratica quando abbiamo un'esecuzione di una SWI?

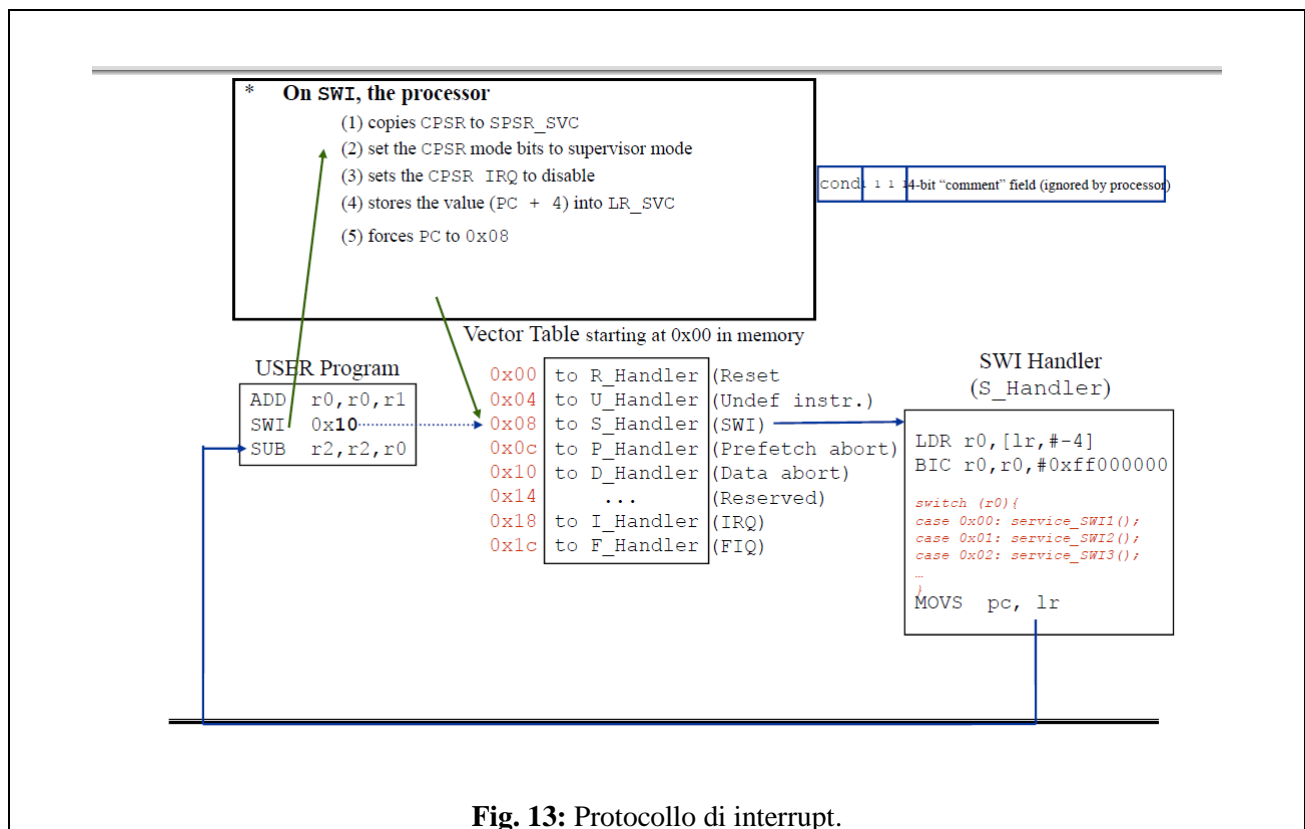
Eseguita una SWI col suo codice (in questo esempio 10, tale codice viene memorizzato nel Comment field) il programma passa in modalità Supervisor (settando LR\_svc con PC+4, SPSR\_svc con CPSR, CPSR in modo supervisor e disabilitando le IRQ) e salta nella tabella dei vettori dove recupera l'indirizzo dell'handler (precisamente all'indirizzo 0x08) che gestisce le chiamate di tipo SWI (S\_Handler). Un handler non è altro che una funzione per la gestione degli interrupt. Una volta recuperato l'indirizzo della funzione di Handler il programma salta in quest'area di memoria e

inizia ad eseguire le istruzioni presenti nell'Handler. Quindi noi in questa funzione dobbiamo creare un meccanismo che, grazie al codice stesso della SWI, ci permette di distinguere i vari interrupt e dunque di eseguire determinate istruzioni in base al codice. Alla fine della procedura di Handler viene eseguita una "MOVS pc, lr" con la quale si riporta il programma all'istruzione successiva all'istruzione SWI.

Come facciamo a recuperare il codice all'interno dell'Handler?

Quando viene eseguita una SWI l'hardware si incarica, in automatico, di andare a salvare il program counter di ritorno dentro il link register. Uno dei modi per determinare l'id dell'interruzione software è quello di caricare in R0 l'opcode SWI che ha scatenato l'interrupt (mediante un'operazione di LOAD -> LDR R0, [lr,#-4], il -4 indica che io voglio andare a prendere dalla mia memoria di codice, il codice esecutivo dell'istruzione precedente a quella in cui ritornerò). L'opCode di una SWI è formato da una parte riservata alle condizioni e una parte di descrizione: in seguito è presente un segmento nel quale viene memorizzato il codice dell'interrupt che nel nostro caso è 10. Ora possiamo eseguire una operazione di mascheramento con la quale azzerò i bit di condizione e descrizione ritrovandomi in R0 semplicemente il codice che a noi serve. Questo è uno dei modi per determinare l'id dell'interruzione software.

Avendo in R0 il codice id possiamo quindi costruire un meccanismo di switching per gestire svariati id che l'handler dovrà gestire.



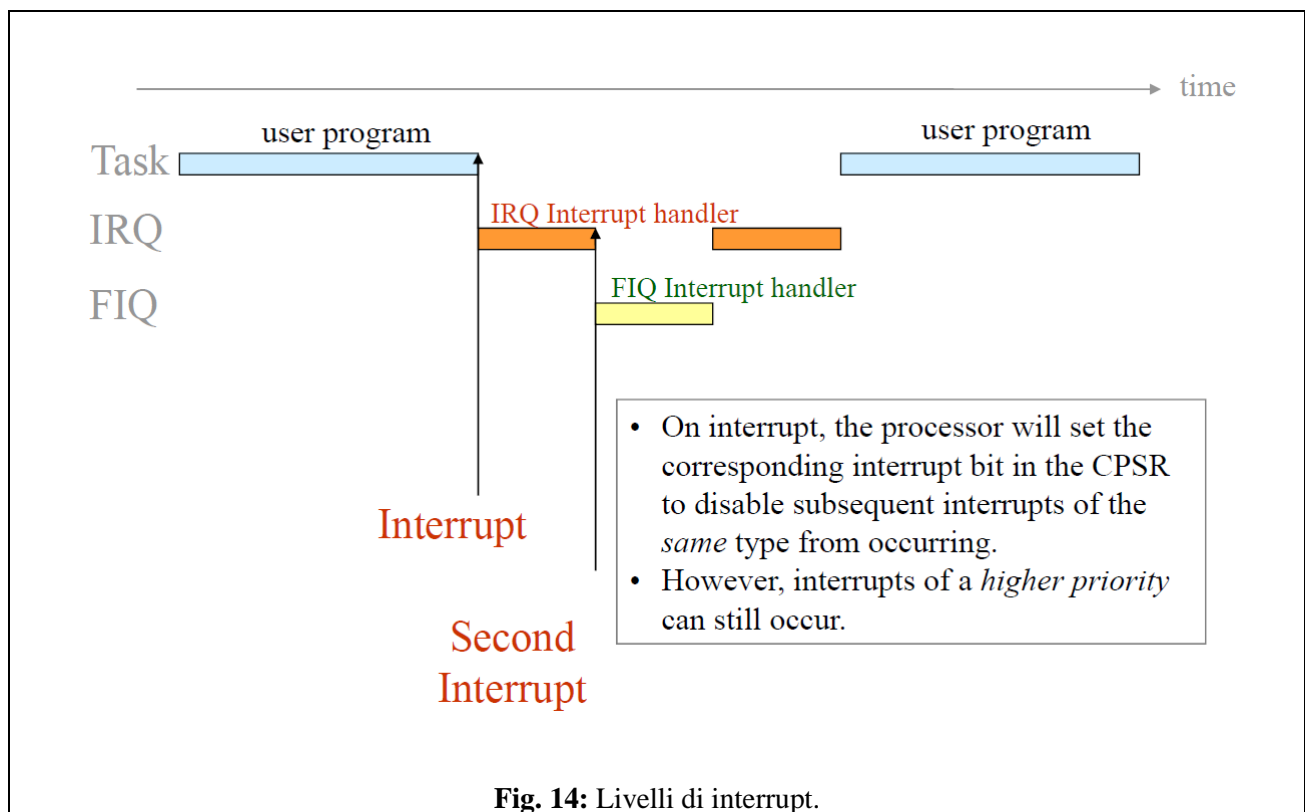
### IRQ vs FIQ

IRQ e FIQ stanno per Interrupt Request e Fast Interrupt Request. le FIQ sono più veloci e hanno maggiore priorità delle IRQ. Quando si hanno chiamate multiple di interruzioni le FIQ sono servite sempre prima delle IRQ. Se si sta gestendo una FIQ le IRQ vengono disabilitate (non è vero il viceversa). Le FIQ sono più veloci per due ragioni. 1) Quando entriamo in un modo FIQ abbandoniamo il modo USER per rimappare un gran numero di registri specifici per questo modo di utilizzo. Dal punto di vista della velocità questo significa che il programma ha bisogno di salvare nello stack meno informazioni e di conseguenza sarà più veloce a passare da un modo user ad un modo FIQ piuttosto che passare da USER a IRQ. In quest'ultimo abbiamo meno registri e di conseguenza il contest switching sarà più lungo. 2) La seconda ragione per cui le procedure di gestione delle FIQ sono più veloci è perchè il FIQ è memorizzata all'ultimo elemento nella tabella dei vettori, di conseguenza posso accedere direttamente al blocco che gestisce l'handler FIQ che è

sequenziale all'ultimo blocco della vector table. Quindi si inizia ad eseguire la procedura di Handler senza fare altri salti.

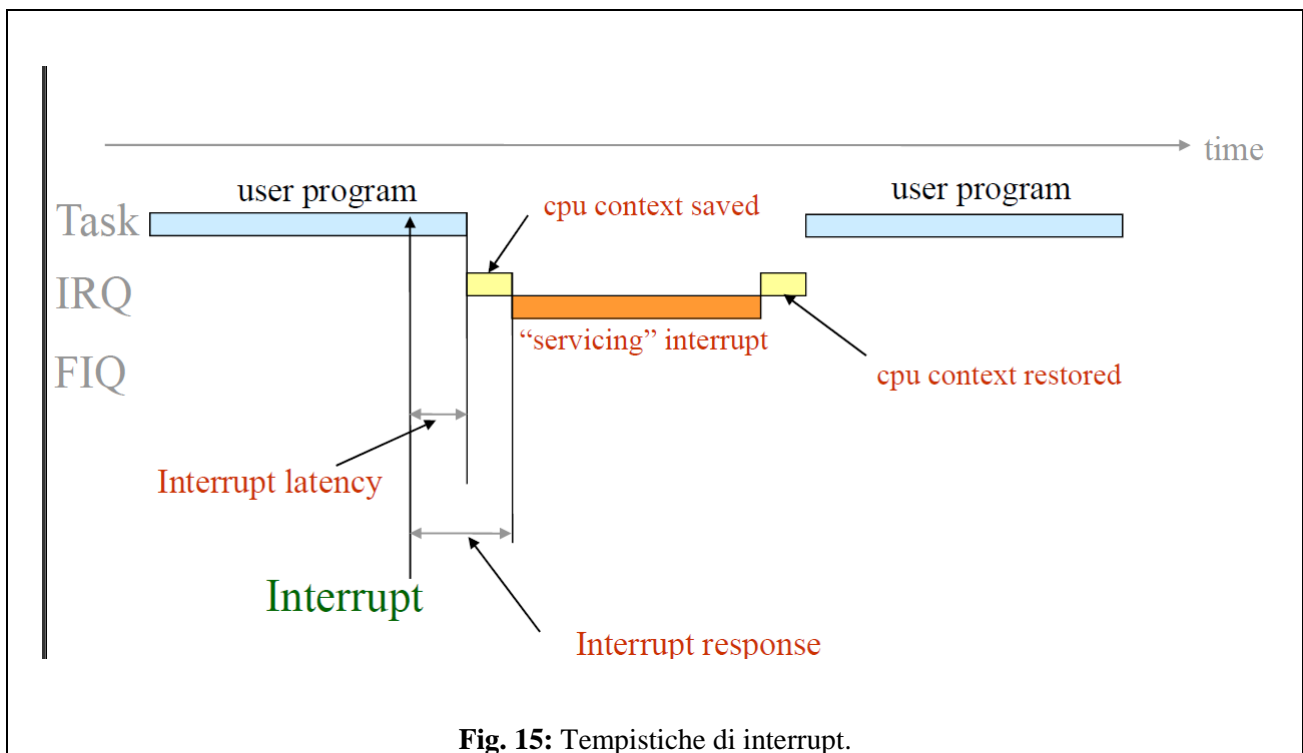
Ovviamente dire che sono molto più veloci è assolutamente relativo. Dal punto di vista umano della percezione hanno una velocità di servizio che è assolutamente comparabile.

La priorità delle FIQ è più alta delle IRQ e quindi può succedere di dover gestire una FIQ mentre una IRQ è attiva.



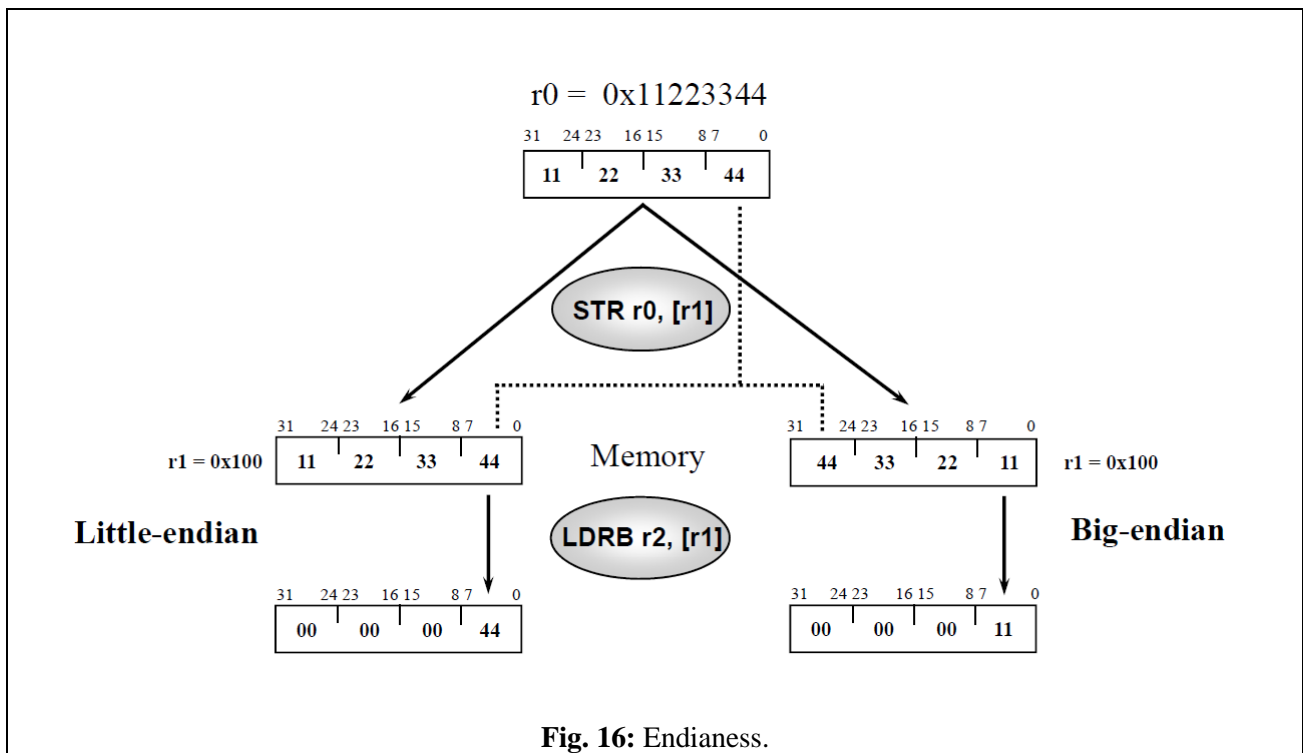
Per quanto riguarda le tempistiche nella seguente immagine si può notare la differenza di velocità nel passare da modo utente a modo Interrupt di una IRQ ed una FIQ: si ha un certo tempo che è destinato al salvataggio del contesto e, dopo aver eseguito l'handler, altro tempo per il passaggio di contesto al fine di ripristinare lo stato utente. FIQ ha molti registri extra che servono per minimizzare questi periodi di passaggi di contesto.





## 2.3 Endianess

L'architettura ARM permette un accesso ai dati sia di tipo little che di tipo big endian: nel primo caso il byte meno significativo di una word sta nel primo indirizzo, nel secondo caso è contenuto nell'ultimo byte della word. Non è un'informazione rilevante a meno che non si vada ad accedere ai dati in porzioni più piccole di una word, come halfword o byte.



## 2.4 Thumb Instruction Set

L'ARM permette di utilizzare due instruction set. Il Thumb è la versione "light" con:

- istruzioni codificate su 16 bit
- istruzioni meno potenti e più corte

Di conseguenza i programmi codificati con istruzioni Thumb risulteranno:

- con un maggior numero di istruzioni, ma con minor memoria di codice
- con un'esecuzione più lenta, che richiede meno potenza.

Le istruzioni Thumb sono quindi adatte a sistemi a basso costo e applicazioni a basse performance.

### *Il T bit*

Il meccanismo per fare lo switch da instruction set completo a thumb instruction set avviene settando il T bit dell'aparoletta di controllo CPSR:

- se  $T=1$ , il processore interpreta il codice come sequenza di istruzioni Thumb
- se  $T=0$ , il processore interpreta il codice come sequenza di usuali istruzioni ARM.

Il valore di T può essere cambiato via software.

## Riferimenti

- [1] <http://www.arm.com/annualreport09/overview/industry-dynamics.html>
- [2] <http://it.wikipedia.org/wiki/System-on-a-chip>
- [3] <http://www.keil.com>
- [4] <http://openwince.sourceforge.net/>