simo85

# ARM A32 ASSEMBLY - UN' INTRODUZIONE

11 August 2014

## Anteprima

L'idea di questo articolo arriva da uno stimolo del nostro caro amico **TardoFreak** [0], il quale, nel thread **Proposta per sistema di sviluppo ARM** [1] ha richiamato l'attenzione dei più volenterosi nel partecipare allo sviluppo di un sistema FREE o scrivere qualcosa riguardo l'architettura per architettura **ARM** [2] per **Electro You**.

Io, onestamente, non sapevo ben cosa fare.

- Programmi IDE, free & open source ce ne sono molti. Infatti dal thread linkato siamo venuti a sapere che **emIDE** [3] è basato su **Code::Blocks** [4].
- Gli strumenti free & open source per compilare i programmi ci sono. GNU li offre di base: **binutils** [5] e **gcc** [6] possono essere compilati per ARM.

È vero che più o meno tutti i giorni al lavoro scrivo programmi che devono essere eseguiti da una CPU ARM, però scrivere un articolo solo dedicato a come preparare un toolchain ARM mi sembrava troppo poco, oltretutto esistono già i toolchain free che funzionano bene.

Ecco dunque che dopo qualche pagina nel thread, mi è venuto lo stimolo finale di imparare e mettere in pratica un pò di Assembly ARM. Dato che mi sembra un ottimo argomento per scrivere qualcosa in un articolo, eccolo qui. :)

Non voglio anticipare troppi dettagli sui contenuti perché potrebbe essere prematuro, quindi mi limito a scrivere l'articolo con l'intenzione di offrire una piccola introduzione al set di istrizioni ARM32 e per avvicinarsi alla programmazione a basso livello per questa architettura. Spero quindi che sia un buon punto di partenza per chiunque abbia bisogno di familiarizzare con il linguaggio Assembly ARM.

L'**ARM Architecture Reference Manual** (**5428 pagine**) è il manuale di riferimento per l'architettura **ARMv8**, scaricabile effettuando il login al sito **ARM infocenter** [7].
Questo è poco ma è quanto basta per rendere bene l'idea che nel 2014 non serve un articolo di **simo85** che spieghi da cima a fondo l'architettura in questione. Dunque, non verrà trattata in maniera esaustiva l'architettura, ma bensì verrà offerta una panoramica il più possibile approfondita delle nozioni fondamentali necessarie per cominciare a programmare in Assembly e capire anche il funzionamento degli esempi scritti.

L'assembler utilizzato per compilare gli esempi è **arm-angstrom-linux-gnueabi-as**, il linker è **arm-angstrom-linux-ld** mentre **arm-angstrom-linux-gnueabi-gdb** è il debugger, fondamentale per analizzare i primi programmi che non stampano nessun carattere nello stdout (schermo).

I programmi citati sono nativamente disponibili nella distribuzione Linux **Angstrom** [8], ottimizzata per i sistemi embedded e nativamente installata anche sulla **Beaglebone Black** [9], cosi come in molte altre schede embedded sul mercato.

La scheda **Beaglebone Black** è quella su cui ho studiato e cominciato a programmare a basso livello per l'architettura in questione.

Nel caso si disponga della scheda però accidentalmente l'assembler, il linker ed il debugger non siano installati, a **questo** [10] indirizzo è possibile trovare i pacchetti formato **ipk** installabili con il gestore di pacchetti **opkg** [11], installato praticamente in tutte le distribuzioni Angstrom.

Dato che compilare l'assembler ed il linker, cosi come il debugger, è un lavoretto abbastanza semplice, prima di esporre gli esempi si spiegherà come compilare i programmi necessari da una distribuzione Linux.


*image.png*

Buona lettura.

## Installare GNU as e GNU gdb

Prima di cominciare a descrivere il set di istruzioni, mi è sembrato giusto anticipare il procedimento di base per compilare ed installare l'assembler ed il debugger per architettura ARM.

Il motivo per cui ho deciso di anticipare questa descrizione è che se si dovesse fare un debugging per qualsiasi esempio di istruzione postato nella prossima sezione, l'articolo potrebbe risultare eccessivamente lungo da seguire.

D'altra parte, il motivo per cui è fondamentale usare il debugger, è perché bisogna capire come si comportano le istruzioni, analizzando i valori salvati nei registri e gli indirizzi di memoria di questi ultimi. **Si conglia quindi di installare e di usare il debugger** (può essere gdb o no).

Prima di tutto, per chi fosse interessato suggerisco di scaricare ed installare il **Sourcery CodeBench arm-toolchain** direttamente dal sito di **MentorGraphics** [12].

La versione del toochain installate è **arm-none-eabi**, la quale indica che non è pensata per nessun sistema operativo in particolare. **eabi** significa **Embedded Binary Interface**.
Invece **arm-none-linux-gnueabi** o **arm-linux-gnueabi** indica appunto che il toolchain è compilato per essere eseguito su un sistema GNU/Linux, utilizzando quindi le librerie di sistema apposite (normalmente scritte in Assembly e C).

La versione lite si può usare tranquillamente e va bene anche per scrivere i primi programmi. Sono anche disponibili le versioni per Windows. Ad ogni modo con quel pacchetto si installa anche **gcc**.. Il che é una comodità.

Ora, siccome è be noto che io uso solo Linux, mi limito a descrivere il procedimento di compilazione ed installazione e **GNU as** [13] e **GNU gdb** [14] solo per questo sistema operativo.

**Compilare l'assembler**

Se non si dispone di un assembler, quest'ultimo può comodamente essere scaricato dalla pagina di **GNU binutils** [15] e compilato con **GNU make** [16]. L'assembler fa parte delle **GNU binutils** come si può constatre sempre dal link [15].

Una volta scompattato il file compresso con tar (leggere la relativa manpage per le varie opzioni disponibili), supponendo che la cartella estratta si chiami **binutils**, da terminale:

```
:~ $ cd binutils
:~ $ mkdir build
:~ $ cd build
:~ $ ../configure \
--prefix=/opt/arm \
--with-sysroot=/opt/arm/lib \
--with-lib-path=/opt/arm \
--target=arm-linux-gnueabi \
--disable-nls \
--disable-werror
:~ $ make
# make install
```

L'asterisco indica che il comando (in questo esempio **make install**) deve essere eseguito con permessi di amministratore (a meno che non abbiate i permessi delle cartelle del file system un pò sballati).

Dopo aver eseguito questo procedimento, l'assembler dovrebbe essere installato nella cartella **opt/arm**.
In ogni caso bisogna ricordarsi di aggiungere questo percorso alla variabile d'ambiente **PATH**. Non è obbligatorio ma è conveniente e comodo.

### Compilare il debugger

Il debugger può essere scaricato a **questo** [17] indirizzo e compilarlo è veramente semplice. Supponendo che la cartella estratta con **tar** si **gdb**:

```
:~ $ cd gdb
:~ $ mkdir build
:~ $ cd build
:~ $ ../configure --prefix=/opt/arm --target=arm-linux-gnueabi
:~ $ make
# make install
```

La documentazione ufficiale di GDB è disponibile online a **questo** [18] indirizzo.

**Si consiglia di leggere le sezioni relative alle istruzioni di breakpoint, stampa del codice e delle variabili e relativi indirizzi di memoria per capire bene gli esempi di codice postati nelle prossime sezioni.**
In particolare i comandi base da eseguire all'interno di gdb sono:

```
/* set a breakpoint at line <line number> */
b <line number>

/* step execution - 1 instruction per time */
s

/* step execution by <nsteps> (nsteps is an integer value) instructions per time */
s <nsteps>

/* disassembly <function> */
disass <function>

/* print value of <register>
p $<register>

/* analyze the memory content of <register> in hex, decimal, and string format */
```

```
x/x $<register>
x/d $<register>
x/s $<register>

/* analyze the memory content of <register> in 32 chars format */
x/32c $<register>
```

Tutti gli esempi scritti sono stati testati su una Beaglebone black.

**Template**

Con il fine di semplificare il lavoro per chi si avvicinasse alla programmazione in linguaggio Assembly ARM, ho scritto un mini template con indicazioni di base per inserire il proprio codice:

```
.data
/* initialized data here */

.bss
/* uninitialized data here */

.file "stdTemplate.s"
.text

.globl _start
_start:

/* program code goes here */

.end
```

**Eseguire il compilatore (l' assembler) ed il debugger**

I comandi eseguiti per compilare (abilitando il debugging), linkare ed eseguire i programmi con il debugger sono rispettivamente:

```
:~$ arm-angstrom-linux-gnueabi-as -g file.s -o file.o
:~$ arm-angstrom-linux-gnueabi-ld file.o -o file.elf
:~$ arm-angstrom-linux-gnueabi-gdb file.elf
```

Ad ogni modo per i primi due comandi si può usare questo Makefile che ho scritto apposta per compilare:

```
# EY ARM asm Makefile
# author: simo85
```

```
TARGET = arm-angstrom-linux-gnueabi-
FDEBUG = -g
ASSEMBLER = as
LINKER = ld
FILE = file
EXTASM = .s
EXTOBJECT = .o
EXTELF = .elf
ALL = *

$(FILE): $(FILE).s
	$(TARGET)$(ASSEMBLER) $(FDEBUG) $(FILE)$(EXTASM) -o $(FILE)$(EXTOBJECT)
	$(TARGET)$(LINKER) $(FILE)$(EXTOBJECT) -o $(FILE)$(EXTELF)

clean:
	rm $(ALL)$(EXTOBJECT) $(ALL)$(EXTELF)
```

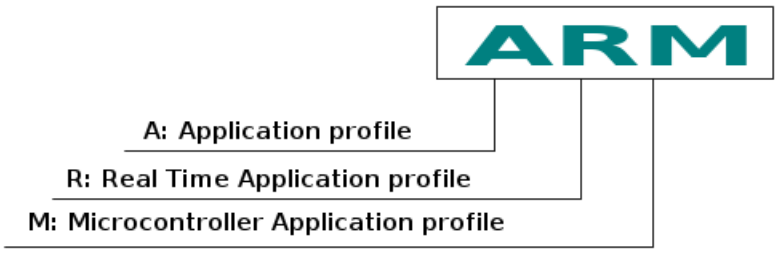
Ricordarsi solo di modificare la variabile **TARGET** e **FILE** nel caso fosse necessario..

## Introduzione ai Registri

### GPR

Prima di cominciare a scrivere qualche esempio mi sembra ragionevole e doveroso dedicare una sezione alla struttura dei registri e come usarli, riassumere in dettaglio il set di istruzioni e la sintassi del linguaggio Assembly ARM.

Attualmente l'architettura **ARM** (al momento di scrivere siamo alla versione **ARMv8**) si suddivide in 3 serie distinte:




- serie **A**: ovvero la "Application profile". È un profilo di architettura applicata alle CPU di carattere comune.

- serie **R**: ovvero la "Real Time profile". È un profilo dedicato alle CPU Real Time, ottimizzate appunto per il trattamento di dati in tempo reale. Esistono sistemi operativi dedicati a questo tipo di dispositivi, conosciuti meglio come gli **RTOS** [18].
- serie **M**: ovvero la "Microcontroller profile". È il profilo dedicato ai microcontrollori.

Si veda anche **Cortex-A** [19], **Cortex-R** [20] e **Cortex-M** [16].

La **pagina** [21] di Wikipedia è comunque ben aggiornata.

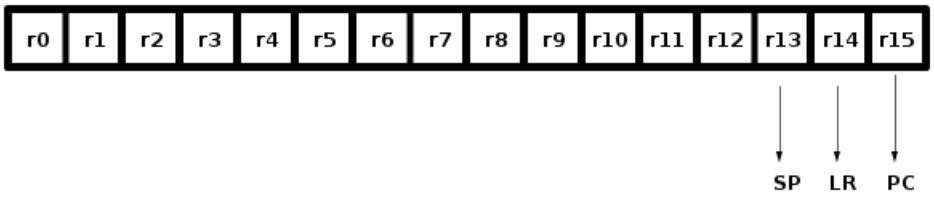
Queste 3 serie differiscono tra di loro in alcune implementazioni. Per esempio, l'**ARMv8-A** [22] supporta l' Assembly **A64**, **A32** e **T32** (**glossario** [23]), mentre un dispositivo **ARMv8-R** [24] supporta solo l'**A32** ed il **T32**.

Queste non sono le uniche differenze.

Il mio consiglio quindi è ovviamente di leggere attentamente le documentazioni relative al dispositivo che ci si accinge ad usare per una descrizione più che dettagliata di ciascuna famiglia, perché come ho anticipato al principio dell'articolo, è impossibile racchiuedere tutte le informazioni qui, inoltre stare dietro a tutte, detto proprio sinceramente, **è un discreto casino**.

**NB:** In questo articolo si fa riferimento all'architettura a 32 bit in quanto il processore usato per gli esempi illustrativi è un **AM3358 Cortex-A8** [25] a 32 bit, quindi il manuale di riferimento relativo è l'**ARM Cortex-A8 Technical Reference Manual** [26].

Di base, l'architettura ARM usa 16 registri a scopo generale (**GPR** - general purpose register) da 32 bit ciascuno, chiamati rispettivamente **r0**, **r1**, **r2**, **⋯** , **r15**.




13 di questi registri posso essere usati a scopo generale, ossia non hanno una scopo preciso e riservato ad un impiego in particolare, e possono essere usati abbastanza liberamente, ma questo è valido solo per l'uso generale, perché quando si tratta di programmare per un sistema operativo in particolare, questa assunzione, da un certo punto di vista, non è più valida. Più avanti vedremo di approfondire questo dettaglio.

Abbiamo quindi che i registro **r13**, **r15** ed **r16** sono chiamati rispettivamente **SP** (stack pointer ), **LR** (link register) e **PC** (program counter).

- **STACK POINTER** : è il puntatore allo stack. Punta all' ultimo elemento inserito nello stack.

- **LINK REGISTER** : è usato per salvare l'indirizzo di memoria della prossima istruzione dopo aver terminato l'esecuzione di una funzione eseguendo l'istruzione **BL** o **BLX**.
- **PROGRAM COUNTER** : salva l'indirizzo di memoria della prossima istruzione ad eseguire.

**NB:** anche **r13** ed **r14** possono essere usati come registri a scopo generale, ma quando si deve usare il puntatore allo stack ed il link register, questi due non possono essere sostituiti liberatamente con altri registri. È quindi chiaro che bisogna usare **r13** ed **r14** a questo scopo. Per **r15** invece non c'è speranza, per fortuna.

Arrivati a questo punto si può citare il **CPSR** (Current Program Status Register) ed il **SPSR** (Saved Program Status Register), i quali meritano anche loro qualche riga a riguardo.

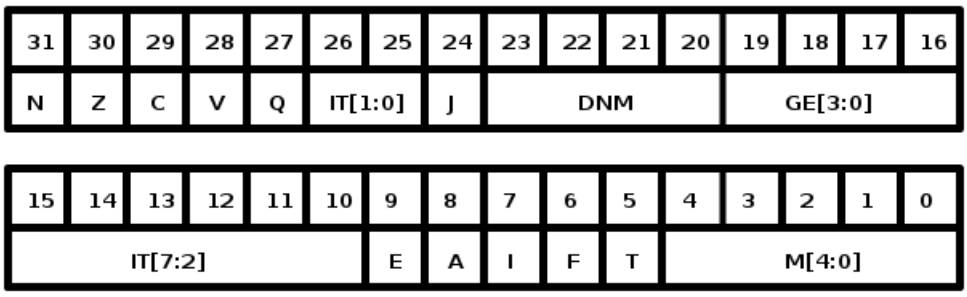
**CPSR e SPSR**

Il **CPSR**, come lascia intuire il suo stesso nome, è un registro di controllo dello stato attuale del processore. Qualsiasi micro-processore o micro-controllore ha un cosidetto **STATUS REGISTER**. Per esempio nell'architettura **x86** [27] è il **FLAGS REGISTER** [28].

L'**SPSR** (Saved Program Status Register) è una copia del **CPSR**.

Si ricorda che esiste più di un registro **CPSR** ma è il processore che decide automaticamente quale usare.

La lunghezza del registro **CPSR** è di 32 bit (potrebbe cambiare da serie a serie e dal processore, non si sà mai !), e questa a seguire ne è la rappresentazione:




Ecco che tocca fare un pò di chiarezza sui vari bit.

- **N** : Negative Flag - il bit vale 1 quando il risultato di una operazione è un numero inferiore a 0. 0 altrimenti.
- **Z** : Zero Flag - il bit vale 1 quando il risultato di una operazione è uguale a 0. 0 altrimenti.
- **C** : Carry Flag - il bit vale 1 quando il risultato di una operazione (somma o moltiplicazione) richiede il bit di carry .

- **V** : Overflow Flag - il bit vale 1 quando un numero con segno ha effettuato un overflow di valore di un numero con segno.
- **Q** : Saturation Flag - il bit viene settato quando il risultato di una operazione richiede una dimensione più grande di quella che il registro in cui deve essere salvato suddetto valore. In altre parole, un registro non basta per salvare il valore dell'operazione.
- **IT [7:0]** : IT execution state bits - sono i bit di stato del blocco condizionale **IT** (If-Then), introdotto nella modalità **Thumb-2**. Più avanti vedremo come usarle il blocco **IT** ed analizzaremo i relativi bit di stato.
- **J** : è il bit di stato che indica se il processore è nello stato **Thumb** o in **ThumbEE**. In funzione dello stato del bit **T** il processore può lavorare in modalità **Thumb**, **Thumb-2** o **ThumbEE**. La modalità **Thumb** offre un set di istruzioni ridotto a 16 bit, cosi come minori prestazioni rispetto alla modalità a 32 bit. La modalità **Thumb-2** è un' espansione della modalità **Thumb**, mentre la modalità **ThumbEEE** è una variante della modalità **Thumb-2**. La modalità **Thumb** viene attivata settando il bit **T** a 1. Se il bit **J** è settato a 0 si lavora in **Thumb-2**, altrimenti si lavora in **ThumbEE**.

In questo articolo non verranno discusse in particolare la modalità **Thumb** e le evoluzioni citate in precedenza. Verranno comunque proposti link di approfondimento.

- **DNM** : (**D**o **N**ot **M**odify) - Il manuale suggerisce di non alterare il valore di questi bit via software.

Precisamente: **DNM fields read as unpredictable values, and must only be written with the same value read from the same field on the same processor**.

tradotto in Italiano, se vogliamo modificare il valore di un registro con campi di bit dichiati come **DNM**, è bene leggere e scrivere sul registro senza però alterare il valore di questi bit. Questi campi sono riservati a future implementazioni.

- **GE [3:0]** : (**G**reat or **E**qual) - Sono i bit di stato che possono essere settati per testare i risultati con istruzioni **SIMD** [29] che operano con valori a 2 byte (16 bit) o 1 byte (8 bit).
- **E** : **Endianess** [30] bit - È possibile scegliere l'ordinamento della rappresentazione dei dati cambiando il valore di questo bit di configurazione.
- **A** : Data Abort Bit Mask - Cambiando il valore di questo bit, è possibile permettere alla CPU di entrare in uno stato temporaneamente irrecuperabile in caso di errore interno o esterno. **Qui** [31] una spiegazione dettagliata. Questa caratteristica viene disabilitata settando il bit a 1.
- **I** : Interrupt Enable / Disable Bit - Se settato a 1, disabilita ogni richiesta di interruzione **IRQ** (interrupt request).
- **F** : Fast Interrupt Enable / Disable Bit - Se settato a 1, disabilita ogni richiesta di interruzione **FIQ** (fast interrupt request). La diffetenza tra **IRQ** e **FIQ** è che quest'ultime hanno un livello di priorità maggiore.
- **T** : Thumb mode bit - Quando il bit è settato a 1 il processore lavora in modalità **Thumb**. La scelta della modalità dipende poi dal valore del bit **J** come spiegato anteriormente. Se il bit **T** è settato a 0, il processore lavora in modalità **ARM**.

- **M[4:0]** : Mode bits - Scelta della modalità di esecuzione: in base alla configurazione scelta, il programma può essere eguito in **User mode** (modalità non privilegiata - normale esecuzione), **FIQ**, **IRQ**, **Supervisor**, **Abort**, **System** e **Secure Monitor**. Queste modalità sono spiegate più in dettaglio nel manuale di riferimento (citato in precedenza) a pag. 41. Altre info **qui** [32].

**Attenzione**: Non tutti i bit del **CPSR** possono essere settati a piacere durante l'esecuzione del programma.

Per esempio, i bit **N**, **Z**, **C**, **V**, **Q**, **GE[3:0]** ed **E** possono essere configurati indipendetemente dalla modalità di esecuzione (privilegiata o no), mentre i bit **A**, **I**, **F** e **M[4:0]** possono essere modificati solo in modalità privilegiata.

Dopo aver offerto questa breve introduzione, e dopo aver capito come come poter usare i registri a scopo generale ed il registro di status, possiamo dedicare qualche riga al set di istruzioni.

## Introduzione alla sintassi ed al set di istruzioni

La sintassi *generica* di una istruzione in linguaggio Assembly ARM è quella rappresentata nel seguente schema:

**INSTRUCTION {COND} {S}  R1, R2  shifter**

Siccome non tutte le istruzioni richiedono lo stesso numero di operandi, ho preferito rappresentare i blocchi con colori diversi.

- Il **Rosso** indica che l'operando è obbligatorio. Infatti in una istruzione non può mancare l'istruzione stessa, e nemmeno il valore di destinazione o il valore cui deve operare una determinata istruzione.
- Il **Blu** indica che l'operando è opzionale. Non è obbligatorio, ma il suo utilizo può essere molto utile.

Attenzione però che alcune istruzioni accettano fino a 4 operandi, alcune, come vedremo a continuazione ne accettano solo 4..

In aggiunta, ci sono istruzioni che non accettano l'uso del barrel shifter, come ad esempio l'istruzione **MUL**. Insomma, si conglia di leggere bene le istruzioni.

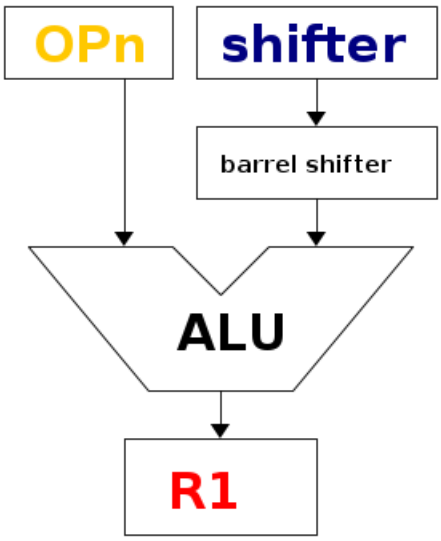
L'uso dell' operatore **COND** (condition) perme di lavorare con i valori del registro **CPRS**. I possibili flag sono:

| COND | Description | CPSR Flag |
|---|---|---|
| EQ | Equal (the result is zero) | Z == 1 |
| NEQ | not Equal (the result is not 0) | Z == 0 |
| CS or HS | the Carry bit is set | C == 1 |

| | | |
|---|---|---|
| **CC or LO** | **the Carry bit is cleared** | **C == 0** |
| **MI** | **MInus (the result is negative)** | **N == 1** |
| **PL** | **PLus (the result is positive)** | **N == 0** |
| **VS** | **the V flag is set (signed overflow)** | **V == 1** |
| **VC** | **the V flag is Cleared (no signed overflow)** | **V == 0** |
| **HI** | **unsigned Higher (after comparision)** | **C == 0 && Z == 0** |
| **LS** | **unsigned Lower (after comparision)** | **C == 0 && Z == 1** |
| **GE** | **signed Greater than or Equal** | **N == V** |
| **LT** | **signed Less Than** | **N != V** |
| **GT** | **signed Greater Than** | **N == V && Z == 0** |
| **LE** | **signed Less than or Equal** | **N != V && Z == 1** |
| **NV** | **NeVer (ARMv1 and ARMv2 only) do not use** | **FALSE** |

- Il suffisso **S** invece permette di attualizzare i valori dei flag nel registro **CPRS** qualora le istruzioni lo permettano, perché il comportamento predefinito delle istruzioni è quello di non attualizzare i valori di quest'ultimo registro, ad eccezione delle istruzioni di comparazione.
- Gli operandi **R2** e **R3** possono essere registri o valori numerici. Nel caso si usino più di due operandi, solo l'ultimo può essere un valore numerico. Se si usando **R1**, **R2** ed **R3** solo quest'ultimo può essere un valore numerico. Se si usa anche l'operatore **shifter**, **R1**, **R2** ed **R3** devono essere registri.
- L'operando **shifter** è un operatore mnemonico che indica lo scorrimento (shift) usando il **barrel shifter** [20] interno alla CPU. Questo permette, con una sola istruzione, di effettuare uno scorrimento a destra o a sinistra, quindi permettendo una moltiplicazione o una divisione implicita.

Ecco un schema interno semplificato dell'interazione tra la ALU, il barrel shifter e gli operandi **R1**, **R2** ed **R3**:



Questo significa che è sempre il valore dell'ultimo operando a destra (che deve essere un registro) che viene fatto scorrere a destra o sinistra, mentre il valore dell'operando stesso non viene alterato.

I possibili valori per ottenere l'operazione di scorrimento sono:

| SHIFT | Description |
|---|---|
| **LSL #k** | **Logical Left Shift by k bits** |
| **LSR #k** | **Logical Right Shift by k bits** |
| **ASR #k** | **Arithmetic Right Shift by k bits** |
| **ROR #k** | **Rotary Right Shift by k bits (no Carry)** |
| **RRX #k** | **Rotary Right Shift by k bits (through Carry)** |

Per maggiori informazioni sullo shift logico, shift aritmetico e shift rotativo (con o senza carry bit) si può consultare la pagina di Wikipedia dedicata alle operazioni bit a bit alla sezione **bit shift** [21].

Una volta chiariti questi concetti si può passare alla descrizione del set di istruzioni.

Quest'ultimo, come in qualsiasi architettura, si suddivide in cinque gruppi di istruzioni. Queste sono:

- **DPI - Data Processing Instructions**: sono le istruzioni che permettono lo spostamento di valori tra registri e registri o valori numerici. Si suddividono in:
  - Operazioni aritmetiche.
  - Operazioni logiche.
  - Assegnazioni di valori.
  - Operazioni di comparazione.
- **BI - Branch Instructions**: servono per cambiare il flusso di esecuzione del programma, appunto un **salto**.
- **LSI - Load Store Instructions**: servono per il trasferimento di dati da registri a memoria e viceversa.

Esistono 3 tipi di queste istruzioni:

  - **Single Register Transfer** : permettono il trasferimento di dati da un registro ad un altro.
  - **Multiple Register Transfer** : permettono il trasferimento di dati usando più registri simultaneamente. In questa categoria rientrano le istruzioni di manipolazione dello stack.
  - **Swap**: effettua lo swap tra un registro ed un dato in memoria.

- **SWI**: **Software Interrupt Instruction**: permette l'esecuzione di una interruzione.
- **PSRI**: **Program Status Register Instructions**: permettono l'assegnazione di valori tra registri **GPR** e **CPSR**.

A seguire si elencano le relative istruzioni, suddivise per tipologia, accompagnate da piccoli esempi di codice.

**Nelle sottosezioni a seguire bisognerà fare attenzione alla sintassi.**

Negli esempi di comportamento non è incluso lo shifter ma se è dichiarato nella sintassi è perché può essere utilizzato.

**DPI**

## MOV

**Sintassi:**

**INSTRUCTION {COND} {S}  R1, R2, shifter**

**Comportamento:**

**R1 = R2**

**Esempio di utilizzo:**

```
/* move the value 10 into r0 */
moveq r0, $10

/* move the value 10 into r0, updating CPRS */
moveqs r0, $10

/* move the value stored in r1 into r0 */
mov r0, r1

/* move the bitwise NOT value stored in r1 into r0 */
movn r0, r1

/* move the value stored in r1 into r0 updating CPRS */
movs r0, r1

/* move the value stored in r1 shifted by 2 bit into r0 */
mov r0, r1, LSL #2
```

**Attenzione:** L'istruzione **MOV** permette l'assegnazione immediata, nel registro **R1**, un valore massimo di **8 bit**. Se si vuole fare una assegnazione immediata con un valore più grande, è meglio usare l'istruzione **LDR** (si veda l'esempio relativo all'istruzione **STR** più avanti).

In alternativa si può usare l'istruzione **MOV** con il barrel shifter (come appena rappresentato) e successivamente si può fare uso delle istruzioni aritmetiche che vengono spiegate a continuazione.

## ADD, ADC, SUB, SBC, RSB, RSC

**Sintassi:**

**INSTRUCTION {COND} {S} R1, R2, R3, shifter**

**Comportamento:**

**ADD**: Addizione

**R1 = R1 + R2**
**R1 = R2 + R3**

**ADC**: Addizione con carry bit

**R1 = R1 + R2 + C**
**R1 = R2 + R3 + C**

**SUB**: Sottrazione

**R1 = R1 - R2**
**R1 = R2 - R3**

**SBC**: Sottrazione con carry bit

**R1 = R1 - R2 - ~(C)**
**R1 = R2 - R3 - ~(C)**

**RSB**: Sottrazione inversa

**R1 = R3 - R2 - ~(C)**

**Esempio di utilizzo:**

```
/* r1 += r2 */
add r1, r2

/* r1 = r2 + 10 */
add r1, r2, $10

/* r0 = r1 + (r2 << 2) */
add r0, r1, r2, LSL #2
```

```
/* r0 = r0 - 1 */
sub r0, $1

/* r0 = r0 - 1 - ~C */
subc r0, $1

/* r0 = r1 - r2 */
sub r0, r1, r2

/* r0 = r2 - r1 */
rsb r0, r1, r2

/* r0 = r2 - r1 - ~C */
rbc r0, r1, r2
```

## MUL

**Sintassi:**

**INSTRUCTION {COND} {S} R1, R2, R3**

**Comportamento:**

**MUL**: Moltiplicazione tra registri

**R1 = R1 * R2**

**R1 = R2 * R3**

**Esempio di utilizzo:**

```
/* r1 = r1 * r2 */
mul r1, r2

/* r1 = r2 * r3 */
mul r1, r2, r3
```

## MLA, SMLAL, SMULL, UMLAL, UMULL

**Sintassi:**

**INSTRUCTION {COND} {S} R1, R2, R3, OP4**

**Comportamento:**

**MLA**: Moltiplicazione e somma tra registri

**R1 = (R2 * R3) + OP4**

**SMLAL**: Moltiplicazione e somma con risultato Signed Long 64 bit tra registri

In questo caso **R1** ed **R2** rappresentano un registro a 64 bit dato che lo scopo è quello di ottenere un valore a 64 bit a partire da 2 registri a 32 bit.

Abbiamo quindi che **R2** rappresenta i 32 bit più significativi (MS 32bit) mentre **R1** rappresenta i 32 bit meno significativi (LS 32bit). Per semplicità chiamiamo quindi **R2:1** il registro che ne risulta.

Il risultato dell'istruzione **SMLAL** è quindi una moltiplicazione tra i registri **R3** ed **OP4**, a sua volta sommato ad **R2:1**.

Quindi:

**R2:1 = (R2[MS32bit] : R1[LS32bit])**

**R2:1 = R2:1 + (R3 * OP4)**

Attenzione a non confondere l'ordine dei registri con l'ordine dei bit.

**SMULL**: Moltiplicazione con risultato Signed Long 64 bit tra registri

Analogamente all'istruzione **SMLAL**, **R1** ed **R2** svolgono lo stesso ruolo anche nell'istruzione **SMULL**. In questo caso però non avviene nessuna somma tra risultati, ma bensì una moltiplicazione diretta, quindi:

**R2:1 = (R2[MS32bit] : R1[LS32bit])**

**R2:1 = (R3 * OP4)**

Il comportamento dei registri, cosi come la sintassi per le istruzioni **UMLAL** e **UMULL** è analogo rispettivamente alle istruzioni **SMLAL** e **SMULL**, ma questa volta il risultato è senza segno, quindi Unsigned Long a 64 bit. Infatti cambia solo una lettera tra le rispettive istruzioni.

**Esempio di utilizzo:**

```
/* r1 = (r2 * r3) + r4 */
mla r1, r2, r3, r4

/*
   (Signed Long 64 bit)
   r1[LS32bit] : r2[MS32bit]) = (r2[MS32bit] : r1[LS32bit]) + (r3 * r4)
*/
```

```
smlal r1, r2, r3, r4

/*
   (Signed Long 64 bit) r1[LS32bit] : r2[MS32bit]) = (r3 * r4)
*/

smull r1, r2, r3, r4

/*
   (Unsigned Long 64 bit)
   r1[LS32bit] : r2[MS32bit]) = (r2[MS32bit] : r1[LS32bit]) + (r3 * r4)
*/
umlal r1, r2, r3, r4

/*
   (Unsigned Long 64 bit)
   r1[LS32bit] : r2[MS32bit]) = (r3 * r4)
*/
umull r1, r2, r3, r4
```

Nel caso non fosse molto chiaro si consiglia seriamente l'utilizzo del debugger.

## AND, ORR, EOR, BIC

**Sintassi:**

**INSTRUCTION {COND} {S}  R1, R2, R3, shifter**

**Comportamento:**

**AND**: AND logico

**R1 = R1 & R2**
**R1 = R2 & R3**

**ORR**: OR logico

**R1 = R1 | R2**
**R1 = R2 | R3**

**EOR**: EXOR logico

**R1 = R1 ^ R2**
**R1 = R2 ^ R3**

**BIC**: Bit Clear

**R1 = R1 & ~(R2)**
**R1 = R2 & ~(R3)**

**Esempio di utilizzo:**

```
/* r0 &= r2 */
and r0, r2

/* r0 = r1 & r2 */
and r0, r1, r2

/* r0 = r0 | (r3 << 3) */
orr r0, r3, LSL #3

/* r0 ^ r1 */
eor r0, r1

/* r0 = r1 & ~(r2) */
bic r0, r1, r2
```

## CMP, CMN, TEQ, TST

**Sintassi:**

**INSTRUCTION {COND} R1, R2**

**Comportamento:**

L'esecuzione di queste istruzioni non altera il valore di **R1** ed **R2**. Diversamente, modificano i flag **N**, **Z**, **C** e **V** del registro **CPSR**.

**CMN:** Comparazione Negata

Flag settati in base al risultato dell'operazione:

**R1 + R2**

**CMP:** Comparazione

Flag settati in base al risultato dell'operazione:

**R1 - R2**

**TEQ:** Test di uguaglianza

Flag settati in base al risultato dell'operazione:

**R1 ^ R2**

**TST:** Test bit a bit

Flag settati in base al risultato dell'operazione:

**R1 & R2**

**Esempio di utilizzo:**

```
/* logical NOT comparision */
cmn r1, r2
cmn r3, $8

/* normal comparision */
cmp r1, r2
cmp r2, $2

/* test equality */
teq r4, r8
teq r2, $4

/* test bit by bit */
tst r2, r1
tst r3, $7
```

**BI**

## B, BL

**Sintassi:**

**INSTRUCTION {COND} LABEL**

**Comportamento:**

**B**: Salto incondizionato alla posizione etichetta come **LABEL**

**PC = (indirizzo di memoria dell'etichetta LABEL)**

**BL**: Salto incondizionato alla posizione etichetta come **LABEL**, il comportamento è lo stesso dell'istruzione **B** ma in questo caso si altera il valore del **LINK REGISTER**.

**PC = (indirizzo di memoria dell'etichetta LABEL)**

**LR = (indirizzo della prossima istruzione ad eseguire)**

**Esempio di utilizzo:**

```
/* assign values to registers */
mov r0, $1
mov r1, $2
mov r2, $3

/* declare label _addition: */
_addition:
add r1, $4

/* jump to "multiply" label */
b _multiply

/* declare label _multiply: */
_multiply:

/* r1 = r2 * r3 */
mul r1, r2, r3

/* branch with link to "_addition" label */
bl _addition
```

Nel semplice esempio proposto gli underscore prima dei nomi delle etichette non sono obbligatori. Di norma le dichiarazioni riservate al compilatore usano questo tipo di nomenclatura.

## BX

**Sintassi:**

**INSTRUCTION {COND} R1**

**Comportamento:**

**BX**: Salto incondizionato di intercambio

**BX R1**

In questo caso **R1** è un registro nel quale è salvato l'indirizzo di memoria dell'etichetta. È principalmente usata per un intercambio tra la modalità **Thumb** e la modalità **ARM**.

**PC = (indirizzo di memoria dell'etichetta LABEL) & 0xfffffffe**

**T = R1 & 1**

**Esempio di utilizzo:**

```
/* branch to the address stored into r1 */
bx $r1
```

## BLX

**Sintassi:**

**INSTRUCTION {COND} R1 | LABEL**

**Comportamento:**

**BLX**: Salto incondizionato di intercambio con attualizzazione del **LINK REGISTER**

**BXL R1**

**BXL LABEL**

Anche in questo caso **R1** è un registro nel quale è salvato l'indirizzo di memoria dell'etichetta. Il comportamento è simile a **BX**, con la differenza che viene alterato il valore del **LINK REGISTER**.

**PC = (indirizzo di memoria dell'etichetta LABEL) & oxfffffffe**

**PC = (indirizzo di memoria salvato in R1) & oxfffffffe**

**T = R1 & 1**

**T = (indirizzo di memoria salvato in R1) & 1**

**LR = (indirizzo della prossima istruzione ad eseguire)**

```
/* branch to the address stored into r1 */
blx $r1

/* branch to label "_label" */
blx _label
```

**LSI**

## LDR, LDRB, LDRH, LDRSB, LDRSH

**Single Register Transfer Load indexing**

**Sintassi:**

**INSTRUCTION {COND} R1, R2, OFFSET**

**INSTRUCTION {COND} R1, R2, R3, shifter**

**Comportamento:**

**LDR**: Salva una word dalla memoria puntata da **R2** nel registro **R1**
**LDRB**: Salva un byte dalla memoria puntata da **R2** nel registro **R1**
**LDRB**: Salva una half word dalla memoria puntata da **R2** nel registro **R1**
**LDRSB**: Salva un signed byte dalla memoria puntata da **R2** nel registro **R1**
**LDRSH**: Salva una signed half word dalla memoria puntata da **R2** nel registro **R1**

Come anticipato anteriormente, sono istruzioni lavorano con dati salvati previamente in memoria.

Questi dati possono essere dichiarati fin dall'inizio del programma, e possono essere variabili o costanti.

L'uso di queste istruzioni porta ad illustrare i diversi tipi di indirizzamenti di memoria ed utilizzi delle sintassi delle istruzioni in questione.

| Indexing | Syntax | Value of R1 (Destination Register) | Value of R2 (Source Register) |
|---|---|---|---|
| **Preindex with writeback** | **INTRUCTION R1, [R2, #OFFSET] !** | **mem(addr + offset)** | **addr + offset** |
| **Preindex** | **INTRUCTION R1, [R2, #OFFSET]** | **mem(addr + offset)** | **not updated** |
| **Postindex** | **INTRUCTION R1, [R2], #OFFSET** | **mem(addr)** | **addr + offset** |

Si precisa che la nomenclatura **mem(addr + offset)** fa riferimento al contenuto in memoria all'indirizzo di memoria **addr + offset**. Oltre a questo, l'offset può essere positivo o negativo, quindi n byte prima o dopo.

In aggiunta, l'uso del punto esclamativo **!** significa che il valore del registro / operando **R1** viene alterato con il valore dell'operando **R2**. Questo piccolo dettaglio è importante e da ricordare anche per le prossime istruzioni di tipo load/store.

Nel caso di usare la sintassi **Multi Register Transfer** il comportamento dell'indirizzamento cambia in funzione del terzo registro (che si sostituisce all'offset) e lo scostamento usando l'operatore shifter.

| Indexing | Syntax | Value of R1 (Destination Register) | Value of R2 (Source Register) |
|---|---|---|---|
| **Preindex with writeback** | **INTRUCTION R1, [R2, R3] !** | **mem(R2 + R3)** | **R2 + R3** |
| **Preindex with writeback** | **INTRUCTION R1, [R2, R3 shifter] !** | **mem(R2 + (R3 shifted))** | **R2 + (R3 shifted)** |
| **Preindex** | **INTRUCTION R1, [R2, R3]** | **mem(R2 + R3)** | **not updated** |
| **Preindex** | **INTRUCTION R1, [R2, R3, shifter]** | **mem(R2 + (R3 shifted))** | **not updated** |
| **Postindex** | **INTRUCTION R1, [R2], R3** | **mem(R2)** | **R2 + R3** |
| **Postindex** | **INTRUCTION R1, [R2], R3 shifer** | **mem(R2)** | **R2 + (R3 shifted)** |

Con lo scopo di rendere il tutto un pò più semplice da capire, ho scritto un piccolo codice di esempio per riassumere i vari tipi di indirizzamento, che illustro a continuazione:

**Esempio di utilizzo:**

```
.data
/* initialized data declaration */

a:
.word 0xffffffff

b:
.byte 0xab

c:
.hword 0x1234

.file "Load-IndexingExample.s"
.globl _start

_start:

/* load the address of variable 'a' into r0 */
ldr r0, =a

/*
   store the content of the address of r0
   with an offset of +4 bytes into r1
```

```
*/
ldr r1, [r0, #4]

/* reload the address of variable 'a' into r0 */
ldr r0, =a

/*
   store the content of the address of r0
   with an offset of +4 bytes into r1
   making r0 to store the address of the variable a
   with an offset of 4 bytes
*/
ldr r1, [r0, #4]!

/* again, reload the address of variable 'a' into r0 */
ldr r0, =a

/*
   store the content of the address of r0 into r1
   making r0 to store the address of the variable a
   with an offset of 4 bytes,
   so the offset is applied just to r0 and not to r1
*/
ldr r1, [r0], #4

.end
```

Si noti come nell'esempio vengano dichiarati una variabile 'a' di lunghezza 32 bit (word), una variabile 'b' di lunghezza 8 bit (byte) ed una variabile 'c' di lunghezza 2 byte (halfword).

In questo esempio è stata usata solo la istruzione **LDR**, ma il codice è facilmente applicabile alle istruzioni **LDRB**, **LDRH**, **LDRSB** e **LDRSH**, utilizzando ovviamente i tipi di dati per cui le istruzioni sono pensate.

**Esempio di utilizzo:**

```
/*
   syntax example of Load indexing with more than one register
*/
ldr r1, [r2, r3]!
ldr r1, [r2, r3]
ldr r1, [r2, r3, LSL #4]
ldr r1, [r2], r3, LSR #4
```

## STR, STRB, STRH

**Sintassi:**

**Single Register Transfer Store indexing**

**INSTRUCTION {COND} R1, R2, OFFSET**
**INSTRUCTION {COND} R1, R2, R3, shifter**

**Comportamento:**

**STR**: Salva un byte o una word contenuta in **R1** nell'indirizzo di memoria salvato in **R2**.

**STRB**: Salva un byte contenuto in **R1** nell'indirizzo di memoria salvato in **R2**.

**STRH**: Salva una half word contenuta in **R1** nell'indirizzo di memoria salvato in **R2**.

In questo caso la sintassi delle istruzioni (a parte il mnemonico dell'istruzione stessa) è uguale al caso anteriore.

L'unica differenza è che nel caso precedente, il registro il risultato viene salvato dalla memoria nel registro **R1**, mentre in questo caso è il contenuto del registro **R1** che viene trasferito in memoria.

**Esempio di utilizzo:**

Ecco un esempio in cui il contenuto della variabile 'a' viene sovrascritto con il contenuto del registro **r0**:

```
.data

a:
.word 0x12345678

b:
.byte 0xab

c:
.hword 0x1234

.file "Store-IndexingExample.s"

.bss
/* uninitialized data here */

.text
```

```
.globl _start

_start:

/* load the value 0xfafafafa into r0 */
ldr r0, =0xfafafafa

/* load the address of the variable 'a' into r1 */
ldr r1, =a

/*
   write the value stored into r0
   into the memory address pointed to by r1
   strictly speaking, override the value of 'a'
*/

str r0, [r1, #0]

.end
```

Durante il debugging, fermando l'esecuzione dell'esempio con un breakpoint dopo l'istruzione **STR**, osservando il contenuto di memoria della variabile 'a', questo dovrebbe essere diverso dal contenuto iniziale.

Ecco che con un pò di pratica si potranno dominare i *puntatori* in Assembly ARM.. :)

## LDM, STM

### Multiple Register Transfer Load / Store indexing

**INSTRUCTION {COND} {Addressing Mode} R1 {!} , OP-REGISTER_LIST**

**Comportamento:**

**LDM:** incrementa l'indirizzo di memoria puntato da **R1** di **4 x N** bytes, dove **N** è il numero di registri listati nel secondo operando.
**STM:** salva l'indirizzo di memoria puntato da **R1** nell'indirizzo di memoria con base al primo registro tra quelli listati nel secondo operando, incrementando questo indirizzo di **4 x N** bytes, dove **N** è il numero di registri passati indicati nel secondo operando.

La spiegazione può non essere del tutto intuitiva quindi si illustrerà il comportamento attraverso un piccolo esempio di codice.

In questo caso i tipi di indirizzamento sono 4:

**IA**: Increment after.
**IB**: Increment before.
**DA**: Decrement after.
**DB**: Decrement before.

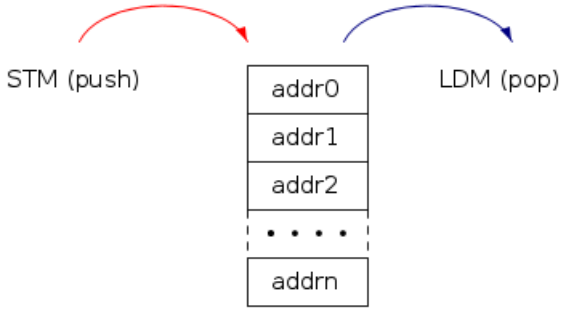
| Addressing Mode | Description | Start Address | End Address | R1! |
|---|---|---|---|---|
| IA | Increment after | Rn | R1 + 4 x N - 4 | R1 + 4 x N |
| IB | Increment before | Rn + 4 | R1 + 4 x N | R1 + 4 x N |
| DA | Decrement after | Rn - 4 x N + 4 | R1 | R1 - 4 x N |
| BD | Decrement before | Rn - 4 x N | R1 - 4 | R1 - 4 x N |

**Esempio di utilizzo:**

```
.data
a:
.word 0x00000000

.bss
/* uninitialized data here */

.file "ldm_stm.s"
.text
.globl _start

_start:

/* load the value of variable 'a' into r0 */
ldr r0, =a

/* load the address of the variable 'a' into r1 */
mov r1, $1
mov r2, $2
mov r3, $3

ldmia r0!, {r1-r3}

/* ... */

stmia r0!, {r1-r3}

.end
```

## Uso e manipolazione dati nella pila di memoria (Stack)

Per quanto riguarda le operazioni sullo Stack, l'architettura ARM fa uso delle operazioni di tipo load/store con molteplici registri, che sono appunto le istruzioni descritte in precedenza: **LDM** (pop) e **STM** (push).
Si ricorda che una istruzione di tipo **load** salva in un registro, specificato come primo operando, il contenuto di memoria all'indirizzo relativo alla lista di registri specificata come secondo operando, mentre una istruzione di tipo **store** si comporta in maniera esattamente contraria.




La rappresentazione non descrive del tutto il funzionamento della pila in questione, ma aiuta solo a capire e relazionare i due tipi di istruzioni. Putroppo non è tutto cosi immediato.

La sintassi per l'uso delle istruzioni **LDM** e **STM** è lo stessa usata negli esempi appena esposti, con la differenza che per lo scopo in questione, il registro usato come primo operando è **r13** ovvero **sp** (stack pointer), e per far riferimento allo stack si usano dei flag distinti rispetto a quelli descritti nelle sottosezioni precendenti.

Lo stack pointer **sp** può puntare all'ultimo elemento inserito nella pila usando l'apposito flag **F** (**Full Stack**), o al primo elemento libero nella pila usando il flag **E** (**Empty Stack**).

Successivamente, è possibile scegliere il tipo di indirizzamento: **ascendente** o **decrescente**.
Nel caso di scegliere un tipo di indirizzamento ascendente usando l'apposito flag **A**, il puntatore allo stack punterà ad indirizzi di memoria maggiori man mano che vengano inseriti gli elementi. Viceversa, scegliendo l'indirizzamento decrescente usando l'apposito flag **D**, il puntatore allo stack punterà ad indirizzi di memoria minori man mano che vengano inseriti gli elementi.

Riassumendo:

| Flag pair | Description |
|---|---|
| **FA** | **Full Ascending** |
| **FD** | **Full Descending** |
| **EA** | **Empty Ascending** |
| **ED** | **Empty Descending** |

**Sintassi:**

**INSTRUCTION R1 !, {R-LIST},**

**Esempio di utilizzo:**

```
.data
    /* initialized data here */

.bss
    /* uninitialized data here */

.file "stdTemplate.s"
.text

.globl _start
_start:

   ldr r1, =0x01010101
   ldr r2, =0x02020202
   ldr r3, =0x03030303
   ldr r4, =0x04040404

   /*
      stack 'push' instruction example
   */

   stmfd sp!, {r1-r4}

   /*
    other code here
    ...
   */
.end
```

Esempi di sintassi.

```
/* full ascending store multiple 'push' */
stmfa sp!, {r1-r4}

/* full descending store multiple 'push' */
stmfd sp!, {r1-r4}

/* empty ascending store multiple 'push' */
```

```
stmea sp!, {r1-r4}

/* empty descending store multiple 'push' */
stmed sp!, {r1-r4}

/* full ascending load multiple 'pop' */
ldmfa sp!, {r1-r4}

/* full descending load multiple 'pop' */
ldmfd sp!, {r1-r4}

/* empty ascending load multiple 'pop' */
ldmea sp!, {r1-r4}

/* empty descending load multiple 'pop' */
ldmed sp!, {r1-r4}
```

**SWI**

**Sintassi:**

**INSTRUCTION {COND} IRQ**

**Comportamento:**

**SWI**: Esegue l'interruzione enumetara come **IRQ**.

Viene generalmente utilizzata nei sistemi operativi. Altera alcuni valori del **CPSR**, tra questi il **PC** in cui viene salvato il valore dell'indirizzo di memoria della tabella delle interruzioni (la cosidetta **vector table**).

**Esempio di utilizzo:**

```
/* execute software interrupt */
swi $0
```

**PSRI**

### *MSR, MRS*

**Sintassi:**

**INSTRUCTION {COND} R1, {CPRS | SPSR}**
**INSTRUCTION {COND} {CPRS | SPSR} {fields} , R2**

**(MSR Only)**

**INSTRUCTION {COND} {CPRS | SPSR} {fields} , #Immediate_Value**

**(MSR Only)**

**Comportamento:**

**MSR:** Copia il contenuto di **R1** in **R2**. Quest'ultimo può essere il **CPSR** o il **SPSR**.
**MRS:** Copia il contenuto del registro **CPSR** o **SPSR** (dipende con quale dei due registri si vuole lavorare), in **R2**. Oltre alla differenza di comportamento, in questa istruzione il registro **R1** deve essere il registro **CPSR** oppure il **SPSR**.

Queste sono le uniche istruzioni che permettono di lavorare con il **CPSR** (Current Program Status Register) e **SPSR** (Saved Program Status Register). Non esistono istruzioni di lettura diretta se non attraverso un registro **GPR**.

Come si è accennato nella sezione dedicata ai **CPSR** e **SPSR**, non tutti i bit sono editabili. Dipendendo dalla modalità in cui lavora il processore, o dallo stato di esecuzione,

**Esempio di utilizzo:**

```
.data
                    /* initialized data here */
a:
.word 0xabcdef01

.bss
                    /* uninitialized data here */
.file "stdTemplate.s"
.text

.globl _start
_start:

   /* load a into r1 */
   ldr r1, =a

   /* load cpsr into r0 */
   mrs r0, cpsr
   /* r1 = (r1 & r0) << 2 */
   and r1, r0, LSL #2
   /* store cpsr into r1 */
   msr cpsr, r1

.end
```

In quest'ultimo esempio non ho fatto molto caso ai valori dei singoli bit del registro di status, in quanto mi sono limitato ad una semplice esempio di sintassi.
Ma la verità è che al momento di scrivere è notte fonda..

### Thumb, Thumb-2 & ThumbEE

Il set di istruzioni *Thumb'* è un subset delle istruzioni **ARM**. </br> Fu pensato per i sistemi con indirizzamento di memoria a 16 bit ed il vantaggio che si può trarre da questo set di istruzioni è che la **densità di codice** è più alta rispetto al set di istruzioni ARM.

Nel set di istruzioni **Thumb** non tutti i registri sono accessibili. **R0** ed **R7** sono completamente accessibili, **R8 ... R12** sono accessibili solo attraverso le istruzioni **MOV**, **ADD** e **CMP**, mentre i restanti **R13 ... R15** hanno restrizioni di accesso.

Anche l'accesso ai registri di status è limitato.

Il passaggio dall'uso del set di istruzioni **Thumb** a **ARM** e viceversa è possibile, come è stato specificato nella sezione **CPSR**, manipolando il bit **T** per la modalità **Thumb**, ed il bit **J** per le sue varianti **Thumb-2** e **ThumbEE**.

Detto questo il passaggio può essere semplicemente effettuato con un AND logico basato sul valore del registro **CPSR**, previamente salvato in un altro registro, o come meglio si preferisce.

```
/*
  suppose r1 have already a properly
  value assigned ...

  load cpsr into r0
*/
mrs r0, cpsr

/*
  logical AND
  r0 = (r0 & r1)
*/
and r0, r0, r1

/* store cpsr into r1 */
msr cpsr, r0
```

Purtroppo, non mi è possibile continuare l'articolo descrivendo il set di istruzioni **Thumb**.</br>
Penso che l'articolo potrebbe diventare eccessivamente lungo e forse difficile da seguire (già cosi potrebbe essere complicato).

Posso dire che il set è ridotto (quindi vi sono meno istruzioni disponibili rispetto alla set a 32 bit) ma comunque quasi tutte le istruzioni sono uguali ed una volta che ci si è fatta la mano con quelle descritte in precedenza, non costa nulla interagire queste due modalità.

Per chi vuole approfondire seriamente l'argomento, consiglio di iscriversi al sito ARM linkato in precedenza, effettuare il login e scaricare il PDF del manuale di riferimento dell'architettura. Si trovano veramente tutte le informazioni di cui si ahanno bisogno, anche riguardo all' **A64**, al set di istruzioni **VFP** (**Virtual Floating Point**) e molto altro.

Ricordo che anche Wikipedia ha una sezione dedicata ala modalità **Thumb, Thumb-2 e ThumbEE**.

### Hello, world!

Ecco che, una volta terminata la descrizione delle istruzioni possiamo scrivere il nostro primo hello world..
Il registro **R8** viene usato come contatore da 0 a 10. Si salta alla fuzione in cui viene stampato il messaggio nello **stdout**, altrimenti, si esce dal programma.

```
/* hello world ARM program */
.data

msg:
.ascii "Hello, world!\n"
len = . - msg

.text

.globl _start
_start:

    /* mov value 0 into r8 */
    mov r8, $0

loop:
    /* call write_msg */
    bl write_msg
```

```
increment:

    /* increment r8 by one */
    add r8, $1
    /* r8 == 10 ? */
    cmp r8, $10
    /* if not equal, branch to the 'loop' label */
    bne loop
    /* otherwise exit program */
    bl exit

write_msg:

    /*
       write syscall
       on Linux OS, the syscall write is the number #4
       and it is best known as:

       write(int fd, char *msg, int msgLen)

       the 'mov r7, $4' instruction
       just tells to the OS we are executing
       the syscall number 4 :)
    */

    /* write function */

    /* load file descriptor 1, which is stdout */
    mov r0, $1
    /* load message into r1 */
    ldr r1, =msg
    /* load the message lenght into r2 */
    ldr r2, =len
    /* tell Linux we are executing syscall #4 */
    mov r7, $4
    /* execute software interrupt */
    swi $0
    /* return to increment label */
    bl increment

exit:
    /* exit function */
```

```
    /* load exit status into r0 */
    mov r0, $0
    /* tell Linux we are executing syscall  #1 */
    mov r7, $1
    /* execute software interrupt */
    swi $0

.end
```

Ecco un possibile codice in C che svolge la stessa funzionalità:

```
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    int i = 0;
    char msg[] = "Hello world!\n";

    while(i++ != 10)
        write(1, msg, sizeof(msg));

    exit(0);
}
```

Come si può vedere, in Assembly non esiste nessuna funzione analoga alla **printf** in C, che ci permetta di stampare 'automaticamente' a video i dati (questo vale per tutti i sistemi operativi). Nel codice Assembly è stata usata la syscall # 4 **write**. Anche in C, con Linux, la funzione **write** usa internamente questa chiamata di sistema.

Invece, come possiamo vedere nella funzione **write_msg**, i dati vengono distribuiti negli appositi registri. In particolare viene salvato il **file descriptor** 1 (in Linux il valore dello standard output è 1 su altri SO non sò..) nel registro **R0**, mentre il numero della chiamata di sistema (**syscall**) viene salvato nel registro **R7**.

Nel sistema operativo Linux le chiamate di sistema sono tutte numerate in un apposito header. Ecco un estratto dal file (nel mio caso) della libreria C:

**/usr/include/asm/unistd_32.h**

```
/*
 * This file contains the system call numbers.
 */
```

```
#define __NR_restart_syscall      0
#define __NR_exit                  1
#define __NR_fork                  2
#define __NR_read                  3
#define __NR_write                  4
#define __NR_open                  5
#define __NR_close                  6
```

Da cui si può notare che la chiamata di sistema #4 è nominata **__NR_write**.
Qualcosa ricorda molto bene la funzione **write**..

Ad ogni modo queste enumerazioni devono coincidere con quelle definite nei file Assembly del sistema operativo.
Ecco infatti un estratto del contenuto del file appartenente ai sorgenti del kernel Linux:

**arch/arm/kernel/calls.S**

```
/* 0 */                CALL(sys_restart_syscall)
                CALL(sys_exit)
                CALL(sys_fork)
                CALL(sys_read)
                CALL(sys_write)
/* 5 */                CALL(sys_open)
                CALL(sys_close)
```

Si ricorda che da architettura a architettura le enumerazioni possono essere distinte..

Se avessi salvato il numero della chiamata di sistema in un registro differente da **R7**, il programma non avrebbe funzionato (perché ho provato in precendeza). Perché proprio **R7**?

Il motivo è che il sistema operativo in questione, quando riceve una richiesta di interruzione, valuta proprio il valore del registro **R7** nel quale viene salvato appunto il numero della chiamata di sistema **syscno**. Questo è possibile comprovarlo analizzando i sorgenti del kernel (reperibili attraverso **www.kernel.org**), in particolare i file Assembly che si trovano nella cartella**arch/arm/kernel**.

Corpo dell'articolo assente

## Conclusioni

Scrivere questo articolo non è stato facile. Lo avevo cominciato prima del semestre universitario e poi ho dovuto continuarlo a spezzoni, anche quando l'aspirina non ha fatto il suo effetto..

Mi sarebbe piaciuto andare oltre alla descrizione del set di istruzioni offrendo qualche esempio in più, ma purtroppo non sono potuto andare oltre, anche perché l'articolo potrebbe essere eccessivamente lungo, oltre al fatto che non sò quando potrei dedicare più tempo di quello che ho dedicato fin'ora.

I dettagli riguardo all'architettura non sono molti ... sono moltissimi, e stare dietro a tutti è alquanto difficile. Se qualcuno dovesse notare qualche imprecisione, lo ringrazierò per la segnalazione ed applicherò le dovute modifiche.

La mia intenzione era quella di scrivere una piccola introduzione al linguaggio offrendo qualche esempio sul come usare le istruzioni. Spero di esserci riuscito.

Per chi cercasse un libro dedicato all'architettura ed alla ottimizzazione del codice C e Assembly, consiglio **ARM System Developer's Guide** di Andrew N. Sloss, Dominic Symes & Chris Wright (link a seguire). È del 2004 ma è molto valido e ricco di contenuti utili per gli sviluppatori.

Ringrazio in anticipo tutti quelli che aprezzeranno e troveranno utile questo articolo.

Un saluto.

## Libri consigliati e link utili

**ARM infocenter**

**ARM System Developer's Guide**.

**ARM System-on-Chip Architecture (2nd Edition)**.

**ARM7TDMI Instruction Set**

**community.arm.com**