Assembler ARMv7

Luca De Paulis

8 novembre 2020

1 REGISTRI

L'architettura ARMv7 ci permette di operare direttamente su 16 registri (da `R0` a `R15`), detti **registri generali**. Essi si dividono in 2 categorie più tre eccezioni:

- I registri da `R0` a `R3` vengono detti **registri temporanei**: vengono usati per contenere valori temporanei e non vi è alcuna garanzia che dopo una chiamata di funzione essi rimangano immutati. In particolare convenzionalmente essi vengono usati per il passaggio di parametri alle funzioni e il registro `R0` contiene il valore di ritorno di una funzione.
- I registri da `R4` a `R11` sono dei **registri permanenti**: possono essere usati per i calcoli, ma dobbiamo assicurarci che alla fine dell'esecuzione di una funzione essi abbiano lo stesso valore che avevano all'inizio (ad esempio salvando i dati sullo *stack* e poi rimettendoli nei registri).
- Il registro `R13` è lo **stack pointer** (`SP`): esso contiene l'indirizzo corrente dello stack, ovvero della struttura dati *LIFO* (Last In First Out) che usiamo per memorizzare i contenuti dei registri in memoria e per implementare le chiamate di funzione.
- Il registro `R14` è il **link register** (`LR`): viene usato nelle chiamate di funzione per memorizzare l'indirizzo di memoria contenente l'istruzione da cui ripartire quando la funzione restituisce il suo valore.
- Il registro `R15` è il **program counter** (`PC`): esso contiene l'indirizzo in memoria della prossima istruzione da eseguire.

Un altro registro utile, ma non direttamente modificabile, è il `CPSW` (ovvero Current Program Status Word, o semplicemente parola di stato): questo registro contiene in particolare 4 flag:

- un flag N (*Negative*) che è settato ad 1 se e solo se l'operazione precedente ha dato come risultato un numero negativo;
- un flag Z (*Zero*) che è settato ad 1 se e solo se l'operazione precedente ha dato come risultato uno zero;
- un flag C (*Carry*) che è settato ad 1 se e solo se l'operazione precedente ha generato un riporto;
- un flag V (*oVerflow*) che è settato ad 1 se e solo se l'operazione precedente è andata in overflow.

Come vedremo, di default la maggior parte delle operazioni non modificano i flag: per farlo abbiamo bisogno di alcune varianti speciali, che terminano per `S`.

2 OPERAZIONI ARITMETICO-LOGICHE

Nel caso delle operazioni aritmetico-logiche la destinazione e il primo operando devono essere necessariamente dei registri (eventualmente anche lo stesso), mentre il secondo operando può essere un registro oppure un immediato (ovvero una costante numerica, come ad esempio `#5`).

SOMME

```
ADD  <dest>, <op1>, <op2>
ADC  <dest>, <op1>, <op2>
ADDS <dest>, <op1>, <op2>
```

Calcola la somma tra <op1> e <op2> e la salva nel registro <dest>. La variante `ADC` aggiunge eventualmente il riporto contenuto nel flag **C**; la variante `ADDS` setta i flag dopo l'operazione.

SOTTRAZIONI

```
SUB  <dest>, <op1>, <op2>
SBC  <dest>, <op1>, <op2>
SUBS <dest>, <op1>, <op2>
```

Calcola la differenza tra <op1> e <op2> e la salva nel registro <dest>. La variante `ADC` aggiunge eventualmente il riporto contenuto nel flag **C**; la variante `ADDS` setta i flag dopo l'operazione.

```
RSB <dest>, <op1>, <op2>
```

Calcola la differenza tra <op2> ed <op1> (notare l'ordine): la sua utilità è nel fatto che con la `SUB` non possiamo calcolare differenze della forma "costante - registro" poiché solo il secondo operando può contenere immediati.

MOLTIPLICAZIONI

```
MUL  <dest>, <op1>, <op2>
UMUL <dest1>, <dest2>, <op1>, <op2>
SMUL <dest1>, <dest2>, <op1>, <op2>
```

`MUL` calcola il prodotto tra <op1> e <op2>: siccome essi sono a 32 bit e il prodotto di due interi a 32 bit è un intero con al massimo 64 bit, `MUL` mette in <dest> i 32 bit meno significativi del risultato. `UMUL` e `SMUL` invece calcolano il risultato a 64 bit e mettono i 32 bit più significativi in <dest2> e i 32 bit meno significativi in <dest1>: la differenza tra le due è che `UMUL` considera gli operandi come interi *unsigned* (ovvero senza segno), mentre `SMUL` li considera con segno.

"MOVE"

```
MOV <dest>, <src>
MVN <dest>, <src>
```

`MOV` copia il contenuto di <src> dentro <dest>. `MVN` copia il contenuto *negato* di <src> dentro <dest>.

"COMPARE"

```
CMP <op1>, <op2>
```

`CMP` opera un confronto tra <op1> e <op2>: in particolare calcola la differenza tra i due valori e setta i flag.

OPERAZIONI LOGICHE

```
AND <dest>, <op1>, <op2>
ORR <dest>, <op1>, <op2>
EOR <dest>, <op1>, <op2>
```

Calcolano rispettivamente l'AND bit a bit, l'OR bit a bit e lo XOR bit a bit di <op1> e <op2>, mettendo il risultato in <dest>.

"BIT CLEAR"

```
BIC <dest>, <op1>, <op2>
```

Effettua un'operazione di *bit clear*: mette a 0 tutti i bit di <op1> che sono settati ad 1 in <op2> e mette il risultato in <dest>.

SHIFT BINARIO

```
LSL <dest>, <op1>, <op2>
LSR <dest>, <op1>, <op2>
ASL <dest>, <op1>, <op2>
ASR <dest>, <op1>, <op2>
```

Calcolano gli shift logici (L) o arimetici (A) a sinistra (L) o destra (R) di <op1> (il numero di posizioni da shiftare è dato da <op2>) e mettono il risultato in <dest>.

ROTAZIONE BINARIA

```
ROR <dest>, <op1>, <op2>
```

Ruota a destra i bit di <op1> di <op2> posizioni e mette il risultato in <dest>.

OPERAZIONI CONDIZIONALI Possiamo aggiungere a tutte le operazioni precedenti due lettere finali che indicano sotto quali condizioni (dei flag) l'operazione viene svolta. Le condizioni sono:

- `EQ`: sta per $Z = 1$;
- `NE`: sta per $Z = 0$;
- `GE`: sta per $N = V$;
- `GT`: sta per $Z = 0$, $N = V$;
- `LE`: sta per $Z = 1$, $N \neq V$;
- `LT`: sta per $N \neq V$.

Quindi ad esempio il codice

```
CMP   R0, #1
ADDEQ R0, R0, #1
```

aggiunge 1 al registro `R0` solo se la `CMP` ha dato come risultato `EQ`, ovvero se la differenza tra `R0` e 1 è 0 (e quindi sono uguali).

3 OPERAZIONI DA E PER LA MEMORIA

Esistono fondamentalmente due tipi di operazioni per la memoria:

- operazioni di *load*, che prendono il contenuto della memoria ad un certo indirizzo e lo caricano in un registro;
- operazioni di *store*, che prendono il contenuto di un registro e lo caricano in memoria in un certo indirizzo.

La sintassi base è

```
LDR <reg_dest>, <ind>
STR <reg_src>,  <ind>
```

Per stabilire l'indirizzo abbiamo diversi metodi.

VERSIONE BASE

```
LDR R0, [R1]
STR R0, [R1]
```

In questo caso l'indirizzo usato è l'indirizzo contenuto nel registro `R1`.

BASE + OFFSET

```
LDR R0, [R1, R2]
STR R0, [R1, R2]
```

In questo caso l'indirizzo usato è l'indirizzo dato da `R1` + `R2`: `R1` si comporta da *base*, mentre `R2` fa da *offset*. (Notare che siccome l'architettura ARMv7 è indirizzata al byte, per andare da una parola alla parola successiva l'offset deve essere di 4 e non di 1.)

Questa istruzione può essere resa ancora più flessibile, in quanto al posto del secondo operando (nel nostro caso `R2`) possiamo inserire uno shift:

```
LDR R0, [R1, R2, LSL #2]
STR R0, [R1, R2, LSL #2]
```

Quindi in questo caso l'indirizzo di memoria è dato dalla somma del contenuto di `R1` con il contenuto di `R2` shiftato a sinistra di due posizioni, ovvero moltiplicato per $2^2 = 4$.

PRE-INDEX

```
LDR R0, [R1, R2]!
STR R0, [R1, R2]!
```

Questa modalità di accesso viene chiamata *Pre-Index*: l'indirizzo considerato è sempre quello dato da `R1` + `R2`, ma dopo aver caricato il valore nel registro `R0` aggiorniamo il registro `R1`, sommando ad esso il valore di `R2`. Quindi ad esempio nel caso della `LDR` la sintassi del Pre-Index è equivalente a

```
LDR R0, [R1, R2]
ADD R1, R1, R2
```

POST-INDEX

```
LDR R0, [R1], R2
STR R0, [R1], R2
```

Questa modalità di accesso viene chiamata *Post-Index*: l'indirizzo considerato è quello contenuto in `R1`, ma dopo aver caricato il valore nel registro `R0` aggiorniamo il registro `R1`, sommando ad esso il valore di `R2`. Quindi ad esempio nel caso della `LDR` la sintassi del Post-Index è equivalente a

```
LDR R0, [R1]
ADD R1, R1, R2
```

LOAD/STORE BYTE

```
LDRB R0, [R1]
STRB R0, [R1]
```

Stessa cosa della Load/Store normale, soltanto che viene caricato un singolo byte (e viene messo nella posizione meno significativa del registro).

LOAD/STORE MULTIPLE

```
LDMxx [R0], {R1, ...}
STMxx [R0], {R1, ...}
```

Queste istruzioni sono utilizzate per caricare in memoria più registri oppure per caricare in più informazioni dalla memoria nei registri. Esse possono essere utilizzate in quattro modi distinti, che vengono specificati rimpiazzando le lettere "xx" nel codice dell'istruzione con lettere adeguate:

- la prima lettera può essere rimpiazzata con `F` (per *full*) oppure `E` (per *empty*): nel caso del full si parte dalla prima posizione di memoria piena, mentre nel caso dell'empty si parte dalla prima posizione vuota;
- la seconda lettera può essere sostituita da `A` (per *ascending*) oppure `D` (per *descending*): nel primo caso la memoria viene letta andando verso indirizzi più grandi, mentre nel secondo viene letta andando verso indirizzi più piccoli.

Queste istruzioni vengono spesso utilizzate insieme allo Stack Pointer, dunque esistono due istruzioni equivalenti ma più brevi:

```
POP  {sequenza di registri}
PUSH {sequenza di registri}
```

4 OPERAZIONI DI SALTO

BRANCH

```
B <etich>
```

È l'operazione base per rappresentare i salti: il flusso del programma viene incondizionatamente interrotto e si salta all'istruzione contrassegnata dall'etichetta <etich> (modificando il contenuto del Program Counter). Aggiungendo i flag di condizione (come `EQ`, `NE`, ...) possiamo ottenere *salti condizionati*, cioè che avvengono solo sotto determinate condizioni.

BRANCH AND LINK

```
BL <etich>
```

Funziona come la branch normale, ma in più memorizza nel Link Register il contenuto corrente del Program Counter (ovvero l'indirizzo dell'istruzione che avremmo eseguito se non ci fosse il salto). Questo è particolarmente utile nel caso delle chiamate di funzione:se ci spostiamo al corpo della funzione con una `BL`, al termine della funzione possiamo semplicemente copiare il contenuto del `LR` nel `PC` per tornare ad eseguire la funzione chiamante.

BRANCH TRAMITE REGISTRO

```
BX <reg>
```

Fa in modo che la prossima istruzione sia quella il cui indirizzo è contenuto nel registro <reg>.