02 - INTRO AL PROCESSORE ARM

INTRODUZIONE

L'*architettura* di un elaboratore è la parte visibile ad un programmatore e include tutto ciò che serve per controllare la macchina:

- tipi di dato
- accesso ai dati
- insieme di istruzioni

Formalmente, un'architettura è definita dall'**INSTRUCTION SET ARCHITECTURE (ISA)**.
Ogni architettura presenta un'ISA diversa, che non è compatibile con le altre.
Noi studieremo l'architettura **ARM 32bit**.
L'ISA definisce ogni istruzione del linguaggio macchina e rappresenta un livello di astrazione al confine fra *hardware* e *software*.

- Non vede come funzionano i moduli.
- Vede solo un sottoinsieme dei registri.

Nell'architettura ARM, un'*istruzione macchina* è rappresentata da una sequenza di 32 bit (4 byte).
Per rendere più facilmente comprensibili le istruzioni, esiste il *linguaggio assembly*, che rappresenta un livello di astrazione appena superiore al linguaggio macchina

(binario/esadecimale).
In particolare, ogni istruzione macchina viene codificata in una stringa alfanumerica del linguaggio assembly.

Un processore ARM supporta dati di lunghezza:

- 8 bit (byte)
- 16 bit (halfword)
- 32 bit (word)

Inoltre, ARM richiede che dati di k bit siano allocati in indirizzi multiplo di k/8 bit:

- Byte: qualsiasi indirizzo.
- Halfword: indirizzi multipli di 2 (occupano 2 byte).
- Word: indirizzi multipli di 4 (occupano 4 byte).

REGISTRI ACCESSIBILI:

Nell'architettura ARM, le istruzioni hanno accesso in lettura/scrittura ai seguenti registri:

- **PC**: (Program Counter) punta alla prossima istruzione.
- **SP**: (Stack Pointer) punta alla testa dello stack.
- **FP**: (Frame Pointer).
- **Registri generici**: (R0, R1, ..., R10) utilizzati per salvare dati temporanei o usati di frequente.
- **Registro di stato**: contiene informazioni sullo stato del processore, divise in campi:
  - **N**: (Negative) N=1 se l'ultimo risultato ottenuto è negativo, 0 altrimenti.
  - **Z**: (Zero) Z=1 se l'ultimo risultato ottenuto è 0, 0 altrimenti.
  - **C**: (Carry) bit di riporto del bit più significativo (32esimo).
  - **V**: (Overflow) V=1 se l'ultima operazione aritmetica ha causato un overflow.

ISTRUZIONI

Ogni istruzione deve contenere tutte le informazioni necessarie per l'esecuzione.
Tali informazioni possono essere *implicite* o *esplicite*.

TIPI DI ISTRUZIONE

- **ARITMETICHE**
  - Operazioni aritmetiche standard su interi con o senza segno.
  - Operazioni aritmetiche su floating points.

    - Valore assoluto, negazione.
- **LOGICHE**
    - Confronti fra valori (<, =, >).
    - AND, OR, XOR, NOT su sequenze di bit.
    - *Shift* e *rotazioni* su vettori di bit.
- **TRASFERIMENTO DI DATI**
  Movimento di dati da e verso registri o locazioni di memoria.
  NOTA: ARM non ha istruzioni per spostamenti da memoria a memoria.
- **INPUT/OUTPUT**
  Accesso in lettura/scrittura a dati e programmi esterni al processore.
- **TRASFERIMENTO DI CONTROLLO**
  Determinano le prossime istruzioni da seguire.
    - Modifica manuale del program counter.
    - Salto ad altra porzione di codice (Branch).
    - Salto di istruzione.
    - Chiamata a procedura (Subroutine) cioè un programma autonomo all'interno di un programma più grande.

Ogni istruzione agisce su degli *operandi*, che possono essere:

- Indirizzo di memoria.
- Indirizzo di un dispositivo I/O.
- Indirizzo di un registro.
- Valore immediato.

Gli operandi specificano dove trovare i valori di input e dove salvare il risultato.
Gli operandi possono essere *espliciti* o *impliciti*. Nel caso di operandi impliciti, l'operando assume un valore predefinito e non viene rappresentato nell'istruzione.
Le istruzioni in ARM possono avere 0,1,2,3 o 4 operandi.

Un esempio di istruzioni a 0 indirizzi sono quelle che agiscono sullo *stack*:

- **PUSH**: aggiunge un nuovo dato sopra agli altri già presenti.
- **POP**: toglie dalla pila il dato che sta in cima.

Allo stack è associato un puntatore alla sua testa (stack pointer). In genere, ogni programma ha accesso ad uno stack.

ESEMPI:
Operazioni aritmetiche su elementi dello stack:

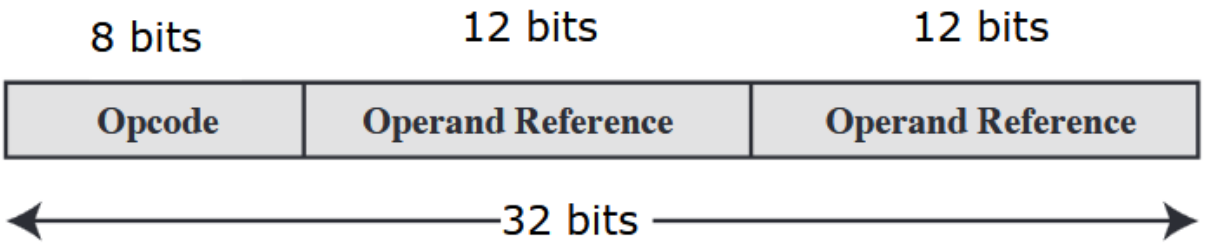
- **ADD**: pop + pop --> push
- **SUB**: pop --> T, pop -T --> push
- **MUL**: pop * pop --> push

- **DIV** pop --> T, pop /T --> push

INDIRIZZAMENTO E CODIFICA

Ogni istruzione è rappresentata da una sequenza di bit, organizzati in campi.

ESEMPIO:



E' necessario stabilire come indicare dove si trova l'operando.

METODI DI INDIRIZZAMENTO

1. **IMMEDIATO**
   Il *valore* da utilizzare è indicato nel campo operando.
   Può essere usato per definire costanti o impostare il valore di una variabile.
   - *VANTAGGI:*
     Semplicità e niente accesso in memoria.
   - *SVANTAGGI:*
     Dimensione del numero limitata dalla dimensione del campo indirizzi.

   ```
   MOV R0, #5
   ```

2. **DIRETTO**
   Il campo indirizzo contiene l'*indirizzo in memoria* dell'operando.
   - *VANTAGGI:*
     Velocità di calcolo e richiede un solo accesso in memoria.
   - *SVANTAGGI:*
     Spazio di indirizzamento limitato dalla dimensione del campo.
     NOTA: *non* supportato in ARM.
3. **INDIRETTO**
   Il campo indirizzo contiene l'*indirizzo in memoria* di *una word contenente* l'indirizzo dell'operando. E' possibile una cascata di indirizzamenti indiretti.

- *VANTAGGI:*
  Ampio spazio di indirizzamento
- *SVANTAGGI:*
  Richiede 2 accessi in memoria.
  NOTA: *non* supportato in ARM.

4. **REGISTRO**
   Il campo indirizzo indica un *registro* che *contiene l'operando*.
   - *VANTAGGI:*
     Richiede pochi bit per il campo indirizzi e non necessita di accessi in memoria aggiuntivi.
   - *SVANTAGGI:*
     Spazio degli indirizzi limitato dal numero di registri.

```
MOV R0, R1
```

5. **REGISTRO INDIRETTO**
   Il campo indirizzo indica *un registro* che *contiene* l'*indirizzo in memoria* dell'operando.
   - *VANTAGGI:*
     Ampio spazio di indirizzamento.
   - *SVANTAGGI:*
     Richiede un accesso aggiuntivo in memoria.

```
MOV R0, [R1]
```

6. **CON OFFSET**
   L'istruzione contiene due campi:
   - Campo *INDIRIZZO* indica un registro che *contiene un valore* al quale viene aggiunto..
   - Campo *A* valore aggiunto a *INDIRIZZO* per calcolare l'*indirizzo in memoria* dell'operando.
     NOTA: Il registro *R* contiene un indirizzo di memoria, e il campo *A* contiene lo spostamento *offset* da questo.

- Con *registro base*:

```
LDR R1, [R0, #0xBB]
```

- Indirizzamento *autorelativo*:

```
LDR R0, [PC, #4]
```

7. **CON STACK**
   L'operando si trova nella testa di uno stack e l'indirizzamento è *implicito*.

PRE e POST INDICIZZAZIONE

- Copia in R1 la word in memoria all'indirizzo R0+1:

```
LDR R1, [R0, #1]
```

- Mette in R0 il valore R0+1, poi copia in R1 la word in memoria all'indirizzo R0 (nuovo valore):

```
LDR R1, [R0, #1]!
```

- Copia in R1 la word in memoria all'indirizzo R0 e poi calcola R0=R0+1:

```
LDR R1, [R0], #1
```

CODIFICA ISTRUZIONI

FORMATI DELLE ISTRUZIONI:

- I bit che codificano un'istruzione sono raggruppati in *campi*.
- I campi devono includere:
  - Tipo di istruzione (*opcode*).
  - Tipo di indirizzamento di ogni operando.
  - Indirizzo di ogni operando.
- Si possono usare formati diversi per istruzioni diverse.

LUNGHEZZA ISTRUZIONI

Dipende da tanti fattori:

- Dimensione memoria;
- Organizzazione memoria;

- Complessità del processore;
- Velocità del processore.

In alcune architetture, come x86, le istruzioni hanno lunghezza diversa. Ciò permette di aumentare il numero di istruzioni, però aumenta la complessità del processore (che deve eseguire istruzioni più articolate) e può richiedere più cicli di fetch (per leggere istruzioni più lunghe).
In particolare, nell'architettura x86 le istruzioni presentano campi di lunghezza variabile.

In ARM, le istruzioni hanno *lunghezza fissa*, il che riduce il numero di operazioni possibili, ma ne aumenta la *semplicità di interpretazione*.

ISTRUZIONI ARM

ARM Quick Reference Card

Tabella che riassume tutte le istruzioni ARM e la loro rappresentazione assembly.
NOTA Consultabile durante i laboratori e all'esame.

ISTRUZIONI CONDIZIONATE

Di default, solo le istruzioni di confronto modificano i bit NZCV.
Una generica istruzione modifica i bit di stato solo quando si aggiunge il suffisso S al simbolo dell'istruzione, in tal caso nella codifica binaria dell'istruzione il 20° bit viene posto a 1.
ES:

- La seguente istruzione *non* modifica i bit di stato:

```
ADD R2, R0, R1
```

- Questa invece si:

```
ADDS R2, R0, R1
```

Tutte le istruzioni ARM, comprese quelle di salto (branch), possono essere rese condizionate:
la loro esecuzione può essere fatta dipendere dallo stato dei bit ZNCV nel CPSR impostati da una precedente istruzione.
Per rendere un'istruzione condizionata è sufficiente aggiungere il suffisso della condizione

(anche in questo caso il 20° bit viene posto a 1).

ES

- Codice C:

```c
if(a > 10){
    a = 0;
}else{
    a = 1;
}
```

- Codice ARM - ASM:

```
CMP R0, #10
MOVGT R0, #0
MOVLE R0, #1
```

LISTA DELLE CONDIZIONI

- **EQ**: uguali
- **NE**: non uguali
- **CS**: carry attivato, maggiore o uguale (senza segno)
- **CC**: carry disattivato, minore (senza segno)
- **MI**: negativo
- **PL**: positivo o zero
- **VS**: overflow
- **VC**: non overflow
- **HI**: maggiore (senza segno)
- **LS**: minore o uguale (senza segno)
- **GE**: maggiore o uguale (con segno)
- **LT**: minore (con segno)
- **GT**: maggiore (con segno)
- **LE**: minore o uguale (con segno)
- **AL**: sempre (default)
- **NV**: mai

ESEMPI CON CONFRONTO E SALTO

```
CMP R1, #0        @confronta R1 e 0
BEQ label1        @salta a label1 se R1=0
```

```
CMP R1, R0        @confronta R1 e R0
BGT label2        @salta a label2 se R1>R0
```

```
TST R1, #0x42     @controlla i bit 1 e 6 di R1
BLNE errore       @salta alla soubroutine errore se almeno uno di essi è =1
```

CLASSI DI SINTASSI

ARM è un tipo di architettura RISC:

- Niente indirizzamento assoluto: non usa puntatori in memoria, ma solo nei registri.
- Indirizzamento in memoria (indiretto con registro) solo per le istruzioni LDR (load) e STR (store).
  Vediamo 3 *classi* di istruzioni che hanno sintassi simile:

1. **CLASSE 1**: per istruzione di elaborazione dati
   ADC, ADD, AND, BIC, CMN, CMP, EOR, MOV, MVN, ORR, RSB, RSC, SBC, SUB, TEQ, TST
2. **CLASSE 2**: per load e store di word o unsigned byte
   LDR, LDRB, STR, STRB
3. **CLASSE 3**: per load e store di halfword o signed byte
   LDRH, LDRSB, LDRSH, STRH

CLASSE 1: Operazioni a 3 indirizzi

- ADC, ADD, AND, BIC, EOR, ORR, RSB, RSC, SBC, SUB

FORMATO:
<opcode>{<cond>}{s} <Rd>, <Rn>, <shifter_operand>

- cond: stabilisce l'esecuzione condizionata
- s: stabilisce se modifica i bit di condizione NZCV
- Rd: destinazione (registro)
- Rn: primo operando (registro)
- shifter_operand: secondo operando (immediato oppure registro, potenzialmente con shift)

Esempio con shift:

```
ADD R3, R1, R2, LSL #2        @R3 <-- R1 + R2*4

ADD R3, R1, R2, ASR R5        @R3 <-- R1 + R2 / 2^R5
```

CLASSE 1: Operazioni a 2 indirizzi

- CMN, CMP, MOV, MVN, TEQ, TST

FORMATO:
<opcode>{<cond>}{s} <Rd>, <shifter_operand>

- cond: stabilisce l'esecuzione condizionata
- s: stabilisce se modifica i bit di condizione NZCV
- Rd: destinazione (registro)
- shifter_operand: secondo operando (immediato oppure registro, potenzialmente con shift)

CLASSI 2 e 3: Load e Store

- **LDR**: nel registro destinazione viene caricata la word della locazione di memoria specificata dal modo di indirizzamento.

```
LDR R0, [R1]        @ R0 <-- M[R1]
```

- **STR**: la word nel registro sorgente viene salvata nella locazione di memoria specificata dal modo di indirizzamento.

```
STR R0, [R1]        @ M[R1] <-- R0
```

Load e store leggono/scrivono word (32 bit).
E' possibile anche leggere o scrivere *byte* o *halfword* con o senza segno aggiungendo i suffissi **B, H, SB, SH**.

- **LDRSB, LDRSH**: il byte/halfword viene copiato in un registro a 32 bit con *estensione del segno*.
- **LDRB, LDRH**: il byte/halfword viene copiato in un registro a 32 bit e i bit più significativi vengono impostati a 0.
- Con **STR** non si usano i suffissi SB/SH: non ha senso estendere il segno in memoria.

CLASSE 2

Load/Store di *word* e *unsigned byte*.
FORMATO:
LDR | STR{B} Rd, <addressing_mode2>
<addressing_mode2> è un indirizzamento di registro indiretto con eventuale:

- senza offset ([R0])
- offset immediato ([Rn, #+/-offset_12])
- offset da registro ([Rn, +/-Rm])
- offset da registro scalato ([Rn, +/-Rm, <sop>#<shift_imm>])
- pre-incremento immediato ([Rn, #+/-<offset_12>]!)
- pre-incremento da registro ([Rn, +/-Rm]!)
- pre-incremento da registro scalato ([Rn, +/-Rm, <sop> #<shift_imm>]!)
- post-incremento immediato ([Rn], #+/-<offset_12>)
- post-incremento da registro ([Rn], +/-Rm)
- post-incremento da registro scalato ([Rn], +/-Rm, <sop> #<shift_imm>)

CLASSE 3

Load/Store di *halfword* e *signed byte*.
FORMATO:
STR | LDR[H] Rd, <addressing_mode3>
LDR[SH | SB] Rd, <addressing_mode3>
<addressing_mode3> è un indirizzamento di registro con eventuale:

- offset immediato
- offset da registro
- pre-incremento immediato
- pre-incremento da registro
- post-incremento immediato
- post-incremento da registro

Differenze rispetto alla classe 2:

- *Non* si possono scalare i registri.
- Gli offset immediati sono *a soli 8 bit*.

PSEUDO - INSTRUCTION

CARICARE UN VALORE IMMEDIATO

Non esiste un'istruzione per caricare in un registro un valore immediato o indirizzo *arbitrario*.
Per questo si usa la *pseudo-istruzione* **LDR Rd, =numero**:

```
LDR R0, =0x47A0
```

ASSEMBLY ARM

STRUTTURA DI UN PROGRAMMA ASSEMBLY

Un programma assembly consiste di uno o più file testuali con estensione **.s**.
Ogni file consiste di una sequenza di istruzioni assembly.

- **NOTA**: la corrispondenza tra istruzioni assembly e istruzioni macchina è (*quasi*) perfetta.
  Il programma eseguibile, costruito dal compilatore, contiene sia istruzioni macchina che dati e un *punto di ingresso* che rappresenta la prima istruzione del programma.
- **Esempio compilazione**: gcc -o sum sum.s (costruisce il programma eseguibile sum a partire dal sorgente assembly sum.s).
  Quindi le istruzioni assembly non bastano per costituire un programma eseguibile.
  Bisogna anche permettere di:
- Indicare la *prima istruzione*;
- Allocare dati in memoria;
- *Indicare* delle righe di codice;
- Inserire *commenti* (preceduti da @).

DIRETTIVE E SIMBOLI

SIMBOLI: sono delle stringhe che denotano particolari valori (indirizzi, costanti numeriche...)

- ES: *Label*: è un simbolo usato per indicare l'indirizzo in memoria di una particolare istruzione.
  Ogni volta che il label viene usato viene sostituito con l'indirizzo dell'istruzione a cui esso è riferito.
  DIRETTIVE: sono istruzioni per l'*assemblatore* che permettono di:
- Definire simboli;
- Allocare aree della memoria per determinati scopi (es salvataggio dati);
- Inizializzare il contenuto delle aree di memoria.
  Il punto di inizio viene definito da:

```
.global main
main:     MOV R0, #0x100
          MOV R1, #0
```

Simboli e direttive scompaiono dopo la compilazione.
Quindi la forma generale di una linea di codice assembly è:
label: istruzione operandi @commento

NOTA

- Ogni campo deve essere separato da uno o più spazi.
- I **label** devono iniziare dal *primo carattere della riga*.
- Le **istruzioni** *non* cominciano mai dal *primo carattere*: devono essere precedute da *almeno 1 spazio*.
- L'assembly è *case-insensitive*.

ALLOCAZIONE DI DATI

Le direttive permettono di allocare spazio per variabili e di inizializzare il loro valore.
Esempi:

```
maschera: .word 0xAAAAAAAA    @alloca e inizializza costanti in memoria
stringa:  .ascii "Pippo"      @alloca e inizializza una stringa
output:   .space 4            @alloca un'area di memoria
```

SEGMENTI DI UN PROGRAMMA

```
.text
.global main
main:  LDR R0, =iterazioni
       LDR R0, [R0]           @ numero iterazioni
       MOV R1, #0             @ registro per somme parziali

ciclo: ADD R1, R1, R0
       SUBS R0, R0, #1
       BNE ciclo

       LDR R2, =output
       STR R1, [R2]           @ salva soluzione in memoria

.bss
```

```
output: .space 4

.data
iterazioni: .word 0x100
```

Un programma assembly è diviso in 3 segmenti:

- **TEXT**: segmento contenente *codice* (istruzioni);
- **DATA**: segmento contenente *dati inizializzati*;
- **BSS**: segmento contenete *dati non inizializzati*.
  Oltre a label, è possibile definire simboli per valori numerici con **.equ**

```
.equ OFFSET, #1    @ definisce una costante OFFSET = 1
```

NOTA: .equ *non* alloca memoria (il valore viene tradotto ogni volta).
Altri esempi:

```
varA: .word 0xAA   @ alloca una word con il valore 0xAA
                   @ l'etichetta varA contiene l'indirizzo in memoria della word

st_base: .skip 128 @ alloca 128 byte, l'etichetta punta al byte con indirizzo min
```

ESEMPI DI COSTRUTTI BASE

IF THEN ELSE

In C:

```
int a = 10;
int b = 20;
int c;

if(a<b)
    c = a;
else
    c = b;
```

In Assembly:

```
.data
addr_a: .word 10 @ inizializza a=10
```

```
addr_b: .word 20 @ inizializza b=20

.bss
addr_c: .skip 4 @ alloca c (4 byte)

.text
.global main
main: LDR R0, =addr_a    @ carica indirizzo di a
      LDR R1, [R0]       @ copia a in R1
      LDR R0, =addr_b
      LDR R2, [R0]
      LDR R0, =addr_c

      CMP R1, R2
      BGE else_case      @ se R1 >= R2 salta
      STR R1, [R0]       @ mette a all'indirizzo addr_c (che è in R0)
      B end_if           @ salta alla fine del blocco

else_case: STR R2, [R0]  @ mette b all'indirizzo addr_c
end_if: NOP              @ non fare nulla
```

Questa versione però è poco efficiente, vediamone una più corta (assumendo data e bss uguali):

```
.text
.global main
main: LDR R0, =addr_a
      LDR R1, [R0]
      LDR R0, =addr_b
      LDR R2, [R0]
      LDR R0, =addr_c

      CMP R1, R2
      STRLT R1, [R0]    @ c = a se a<b (less than)
      STRGE R2, [R0]    @ c = b se a>=b (greater or equal)
```

FOR LOOP

In C:

```
int sum = 0;
int n = 100;

for(int i=1, i<=100; i++){
```

```
    sum = sum + i;
}
```

In Assembly:

```
.data
addr_n: .word 100 @ inizializza n = 100

.bss
add_sum: .skip 4 @ alloca 4 byte per sum

.text
.global main
main: LDR R3, =addr_n        @ copia indirizzo di n in R3
      LDR R2, [R3]           @ copia n in R2
      MOV R1, #0             @ R1 contiene la somma parziale
      MOV R0, #0             @ R0 contiene il contatore i

for_loop: ADD R1, R1, R0     @ calcola sum = sum + i
          ADD R0, R0, #1     @ incrementa i di 1
          CMP R0, R2         @ confronta i con n
          BLE for_loop       @ salta a inizio loop se i <= n

          LDR R3, =addr_sum  @ R3 contiene l'indirizzo a cui salvare la somma
          STR R1, [R3]       @ la somma (R1) viene salvata a quell'indirizzo
```

PRIMI PROGRAMMI

COPIA DI UN VETTORE

Il vettore da copiare può essere collocato nella sezione data tramite direttiva .ascii, collocandolo all'indirizzo vett1 (label):

```
.data
vett1:
 .ascii "Registers R0 to R7 are unbanked registers. This me"
 .ascii "ans that each of them refers to the same 32-bit ph"
 .ascii "ysical register in all processor modes. They are c"
 .ascii "ompletely general-purpose registers, with no speci"
 .ascii "al uses implied by the architecture, and can be us"
 .ascii "ed wherever an instruction allows a general-purpos"
 .ascii "e register to be specified. Registers R8 to R14 ar"
 .ascii "e banked registers. The physical register referred"
```

```
 .ascii "to by each of them depends on the current processo"
 .ascii "r mode. Where a particular phy"
```

Nella sezione bss (dati non inizializzati) allochiamo lo spazio per il vettore destinato a ricevere la copia (vett2).

```
.bss
vett2: .skip 480 @ spazio di 480 byte (il vettore vett1 ha 480 caratteri)
```

Conviene mantenere nei registri le informazioni che servono per realizzare l'algoritmo:

```
.global main
.text
main: MOV R0, #480    @ lunghezza del vettore in byte
      LDR R1, =vett1  @ indirizzo del vettore sorgente
      LDR R2, =vett2  @ indirizzo del vettore destinazione
```

Determiniamo l'indirizzo di stop (*successivo* all'ultimo elemento):

```
      ADD R0, R1, R0  @ R1 (primo elem. di vett1) + R0 (dim di vett1)
```

Cuore del programma:

```
loop: LDR R3, [R1], #4  @ carica un word dal 1o vettore
                        @ e incrementa l'indice al prossimo elemento
      STR R3, [R2], #4  @ salva il word nel 2o vettore
                        @ e incrementa l'indice al prossimo elemento
      CMP R1, R0        @ raggiunta la fine?
      BLO loop          @ se minore (lower) ripete il loop
```

NOTA: Visto che la lunghezza del vettore è multiplo di 4 stiamo copiando un word (4 byte) alla volta. Se la lunghezza *non* fosse stata multiplo di 4 avremmo dovuto copiare un byte alla volta (LDRB e STRB).

LUNGHEZZA DI UN VETTORE

Scriviamo un programma che calcoli la lunghezza di una stringa.
NOTA: la stringa è terminata dal codice di controllo *EOS* (End Of String): byte con tutti i bit uguali a zero.
Il programma *non* fa uso di dati non inizializzati (niente sezione bss).
La sezione data viene invece usata:

```
.data
str1: .ascii "Questa stringa è lunga 36 caratteri"
      .byte 0 @ EOS
      .align  @ allineamento a indirizzo di word (indirizzo multiplo di 4)
```

Cuore del programma:

```
.text
.global main
main:   LDR R0, =str1      @ puntatore alla stringa str1
strlen: MOV R1, #0         @ R1: contatore lunghezza stringa
loop:   LDRB R2, [R0, R1]  @ carica R1-esimo byte (indirizzo base R0 + indice)
        CMP R2, #0         @ confronta il carattere con EOS
        ADDNE R1, R1, #1   @ se not-equal incrementa il conteggio caratteri
        BNE loop           @ ed esamina il successivo
```

FATTORIALE

Il programma non fa uso di dati inizializzati e neanche non-inizializzati.
Cuore del programma:

- R0 contenga:
  - *INIZIO:* numero n di cui calcolare il fattoriale.
  - *FINE:* n! o 0 in caso di overflow.

```
.text
.global main
main:  MOV R0, #6     @ n di cui calcolare il fattoriale
fact:  CMP R0, #12    @ 13! non sta in 32 bit
       BLS do_it      @ se minore o uguale (less/equal, senza segno) calcola
       MOV R0, #0     @ altrimenti metti 0 (overflow)
       B fine         @ salta a fine

do_it: MOV R1, R0     @ R1 iteratore
       MOV R0, #1     @ R0 prodotto parziale

ciclo: CMP R1, #0     @ controlla se R1 = 0
       BEQ fine       @ se si, fine
       MUL R0, R1, R0 @ moltiplica R0 (parziale) per R1 (n, n-1, n-2 ...)
       SUB R1, R1, #1 @ decrementa R1
       B ciclo        @ ripeti

fine:  NOP
```

STRUTTURE DATI

A livello di linguaggio assembly c'è completa visibilità di come sono organizzate in memoria le strutture di dati e i loro elementi.

ARRAY E MATRICI

ARRAY

La definizione di un *array* si effettua specificandone le seguenti informazioni:

- *indirizzo iniziale* in memoria V,
- *numero degli elementi* N,
- *lunghezza di ciascun elemento* (in byte) L.

$$V: \begin{bmatrix} V[0] \\ V[1] \\ \cdots \\ V[N-1] \end{bmatrix}$$

DEFINIZIONE ARRAY:

Un array si alloca con le seguenti istruzioni:

```
.bss
V: .skip N*L
```

ACCESSO AGLI ELEMENTI:

Per accedere agli elementi di un array si utilizza il metodo con *registro indice*:

- Una componente *fissa* costituita dall'indirizzo iniziale *V*.
- Una componente *variabile* che fornisce l'*offset* rispetto a V.

Se l'indice parte da zero, l'indirizzo dell'elemento *i-esimo* è:
Ind V[i] = V + i*L.

Se L è potenza intera di 2, allora il calcolo dell'offset può essere ottenuto con una semplice operazione di scorrimento verso sinistra.
Esempio: copia in R3 il valore V[i] con L=4.

```
LDR R0, =V          @ indirizzo di V in R0
MOV R1, #i          @ indice i in R1
```

```
LSL R1, R1, #2      @ calcola offset = i*4 (2 shift a sinistra)
LDR R3, [R0, R1]    @ copia (V[i] = R0 + offset) in R3

@ o più breve: LDR R3, [R0, R1, LSL, #2]
```

Se L *non* è una potenza intera di 2 si calcola il prodotto i*L con MUL.
(**NOTA:** una moltiplicazione è più costosa in termini di tempo di esecuzione di una somma o shift)
ESEMPIO:

```
LDR R0, =V        @ indirizzo di V in R0
MOV R1, #i        @ indice i in R1
MOV R2, #L        @ dim elementi L in R2
MUL R1, R1, R2    @ offset i*L
LDR R3, [R0, R1]
```

MATRICI

Gli elementi di una matrice possono essere collocati in memoria ordinati per *riga* o per *colonna*.
Ordinamento per righe:

$$M: \begin{bmatrix} M[0,0] \\ M[0,1] \\ \dots \\ M[0,C-1] \\ M[1,0] \\ M[1,1] \\ \dots \\ M[R-1,C-1] \end{bmatrix}$$

DEFINIZIONE MATRICE

Richiede:

- Indirizzo *iniziale* M;
- Numero di *righe* R;
- Numero di *colonne* C;
- *Lunghezza* L di ciascun elemento.

L'allocazione si effettua con le seguenti istruzioni:

```
.bss
M: .skip R*C*L
```

ACCESSO AGLI ELEMENTI

Per individuare un elemento si usano due indici: *riga i* e *colonna j*.
L'indirizzo dell'elemento **M[i,j]** è dato da:
Ind M[i,j] = M + (i*C+j)*L
Conviene usare un metodo di indirizzamento a due componenti:

- Indirizzo *base* costituito da M;
- *Offset* calcolato valutando l'espressione (I*C+j)*L.

Esempio: copiare R0 in M[i,j] (vettore di halfword, L=2), con i contenuto in R1, j in R2, M in R4.

```
LDR R3, =C          @ copia C (numero colonne) in R3
MUL R3, R1, R3      @ calcola R3 = i * C
ADD R3, R3, R2      @ calcola R3 = i * C + j
LSL R3, R3, #1      @ calcola R3 = (i*C+j) * 2
STRH R0, [R4, R3]   @ copia R0 in M[i,j]
```

MATRICE CON ARRAY AUSILIARIO DI PUNTATORI

Utilizzando un array ausiliario di puntatori si evita il calcolo del prodotto i*C (oneroso) e l'accesso alla matrice si riduce a 2 accessi ad array:
P[i] punta all'inizio dell'i-esima riga.

- Definizione della matrice e dell'array ausiliario P:

```
.bss
M: .skip R*C*L

.data
P: .word M, M+C*L, M+2*C*L, ..., M+(R-1)*C*L
```

Con L=2, i in R1, j in R2, P in R4 il trasferimento precedente si può ottenere così:

```
LSL R3, R1, #2          @ i-esima riga: i-esimo elemento di P (array di word)
LDR R3, [R4, R3]        @ R3 = R4 (P) + i*4
ADD R3, R3, R2, LSL #1  @ R3 = R3 + R2 (j) * 2
STRH R0, [R3]           @ copia R0 in M[i,j]
```

MATRICE CON ARRAY AUSILIARIO DI OFFSET

Al posto dell'array di puntatori P si può usare un array ausiliario O con gli *offset di ciascuna riga* rispetto all'indirizzo iniziale M.

```
.bss
M:  .skip R*C*L
```

```
.data
O: .hword 0, C*L, 2*C*L, ..., (R-1)*C*L
```

Mentre i puntatori dell'array P sono da 4 byte, gli offset di O possono essere da 1 o 2 byte.
Inoltre lo stesso array di offset può essere usato per *tutte le matrici dello stesso tipo* (stesso numero di righe e colonne e stessa lunghezza degli elementi).
Con L=2, i in R1, j in R2, M in R3, O in R4, il trasferimento precedente diventa:

```
LSL R5, R1, #1          @ i*2 perchè O contiene halfwords
LDRH R4, [R4, R5]       @ R4 = O + i*2 (i-esimo elemento di O)
ADD R4, R4, R2, LSL #1  @ R4 = (O + i*2) + j*2 (riga --> posizione nella riga)
STRH R0, [R3, R4]       @ copia R0 in M[i,j]
```

STACK

Lo **STACK** (*pila*) è una struttura *LIFO* (Last In First Out) che mantiene una serie di elementi "impilati" uno sopra l'altro.
Le operazioni sullo stack sono:

- **PUSH**: aggiunge un nuovo elemento sopra quelli già presenti.
- **POP**: toglie l'elemento che sta in cima.

Ad uno stack è associato un puntatore alla testa dello stesso: *Stack Pointer*.
In genere, ogni programma ha accesso ad uno stack di sistema, il cui puntatore è mantenuto nel registro *SP*.

Noi utilizzeremo uno stack *full descending*:

- SP *decresce* con PUSH.
- SP punta all'*ultimo elemento usato*.

Definizione di un'*area di memoria riservata* per uno stack di 4KB e *inizializzazione* dello stack pointer:

```
.data
       .equ STL, 4096    @ costante StackLength = 4096

.bss
       .skip STL         @ allocazione memoria
stack: .skip 4           @ indirizzo puntato da stack

.text
```

```
LDR SP, =stack          @ nota: SP inizialmente punta a un elemento
                        @ al di fuori dello stack
```

OPERAZIONI

1. Operazione *PUSH*
   PUSH {R0} : SP = SP - 4, M[SP] = R0 (stack cresce)
2. Operazione *POP*
   POP {R0}: R0 = M[SP], SP = SP + 4 (stack decresce)
3. Push e Pop *multipli*:
   PUSH {R0-R7}
   POP {R0-R7}

LISTE CONCATENATE

Ad ogni elemento di una *lista concatenata* è associato un *puntatore* che individua l'elemento *successivo*. Gli elementi contengono un campo puntatore e possono trovarsi *ovunque in memoria*.
La lista contiene un puntatore **HEAD** al primo elemento o a **NIL** (ultimo nodo).

Inizializzazione di una lista concatenata *vuota*:

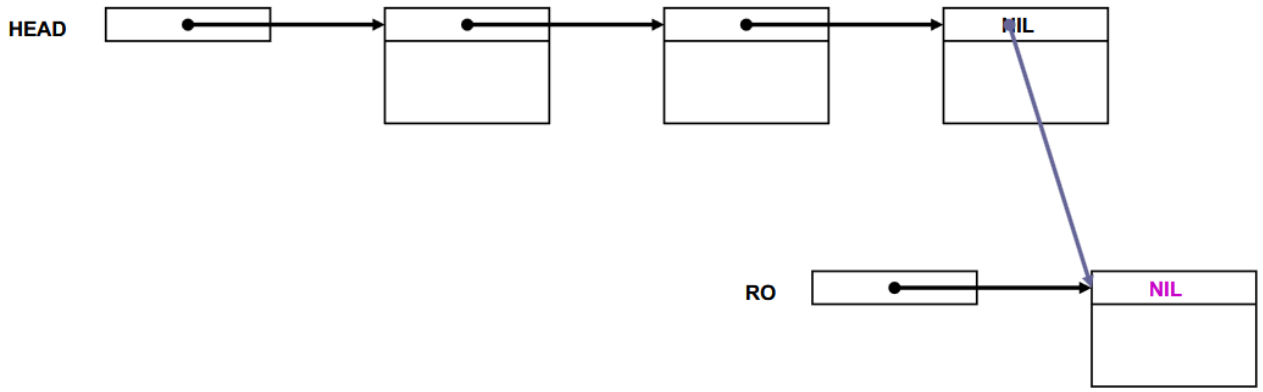
```
.equ NIL, -1    @ definizione costante NIL = -1
HEAD: .word NIL
```

Se R1 punta a un elemento, con l'istruzione seguente si percorre la lista:

```
LDR R1, [R1]    @ avanzamento lungo la lista
```

INSERZIONE

Inserzione a fine lista di un elemento puntato da R0:

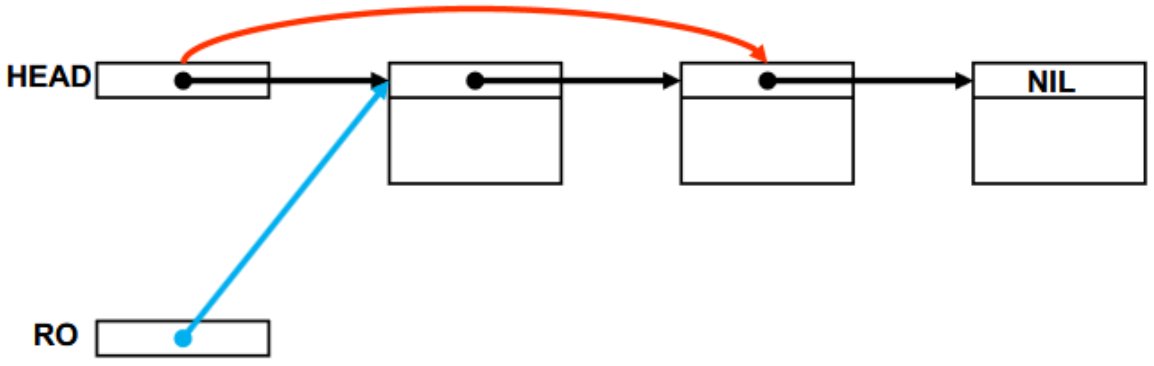


Codice:

```
        LDR R1, =HEAD   @ puntatore iniziale
        MOV R3, #NIL    @ copia NIL in R3

Cerca:  LDR R2, [R1]    @ carica indirizzo in R2
        CMP R2, #NIL    @ è l'ultimo elemento? (R2=NIL?)
        BEQ Ultimo      @ Se lo è, salta
        MOV R1, R2      @ altrimenti si passa al prossimo
        B Cerca         @ ripetere

Ultimo: STR R0, [R1]    @ Aggancio: R1 conteneva l'indirizzo di NIL. Adesso a
                        @ quell'indirizzo viene scritto R0, che punta a [R0]
        STR R3, [R0]    @ L'ultimo elemento ([R0]) punta di nuovo a NIL
```

ESTRAZIONE DEL PRIMO ELEMENTO

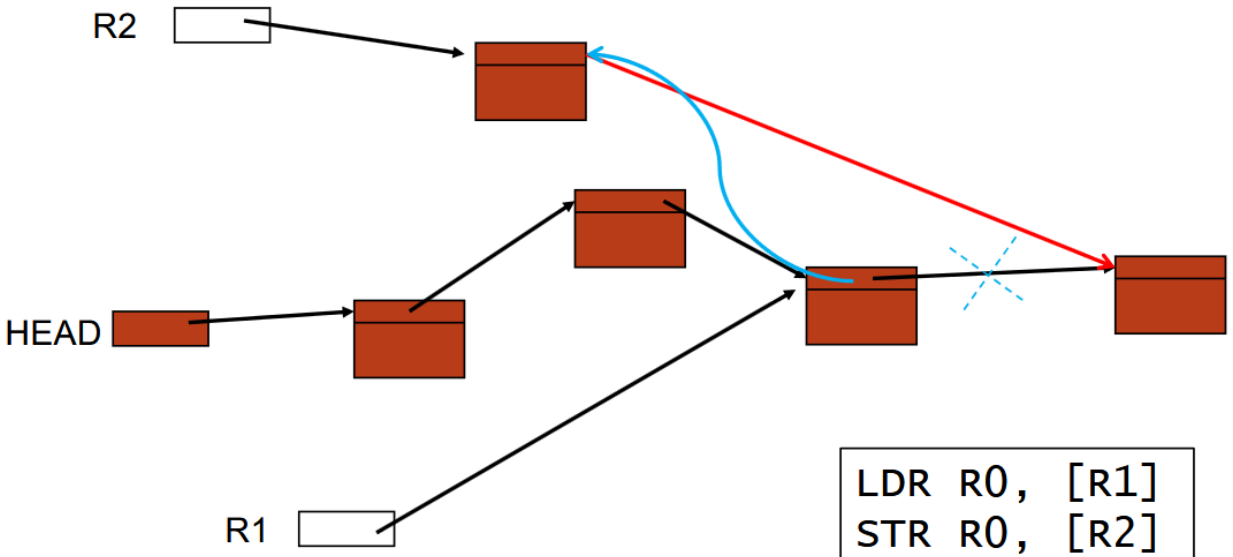
Rimozione del primo elemento della lista e inserzione di un puntatore all'elemento rimosso in R0:



Codice:

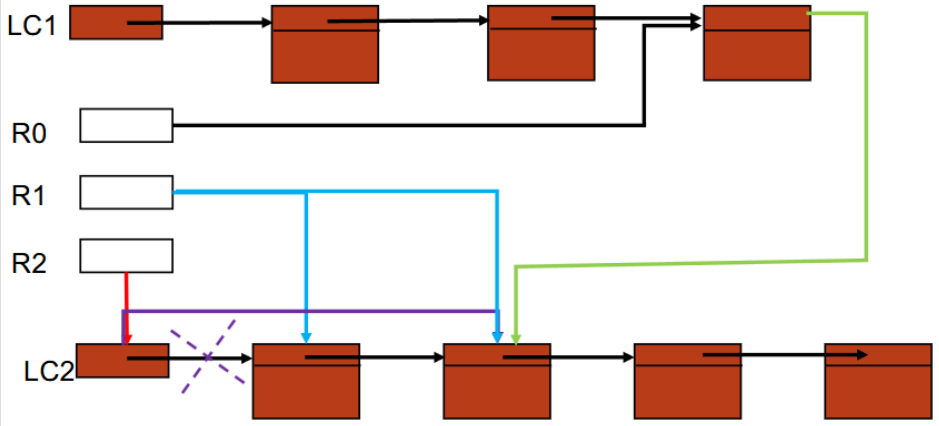
```
        LDR R1, =HEAD @ puntatore iniziale
        LDR R0, [R1]  @ puntatore al primo elemento
        CMP R0, #NIL  @ lista vuota?
        BEQ Vuota     @ se si, non estrarre
        LDR R2, [R0]  @ ora R2 contiene l'indirizzo del prossimo elemento..
        STR R2, [R1]  @ che viene messo in HEAD

Vuota: ...
```

INSERZIONE IN POSIZIONE GENERICA



UNIONE DI DUE LISTE

LC1 e LC2 sono 2 liste concatenate. In R0 vi sia un puntatore all'ultimo elemento di LC1.
Rimuovere il primo elemento di LC2 e agganciare l'intera lista LC2 appena modificata in coda ad LC1.




```
LDR R2, =LC2
LDR R1, [R2]
LDR R1, [R1]
STR R1, [R2]
STR R1, [R0]
```