

# Assembly in ARM

Argomento:

- Struttura di un programma assembly
- Esempi di costrutti base
- Primi programmi

# Struttura di un programma assembly ARM

# Struttura programma assembly

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

```
mov r0, #0x100  
mov r1, #0  
add r1, r1, r0  
subs r0, r0, #1
```

Istruzioni

Operandi

# Costruzione di un programma

- istruzione assembly  $\neq$  istruzione macchina
  - Istruzione macchina è la rappresentazione binaria di un'istruzione assembly
  - (quasi) perfetta corrispondenza
- Il programma eseguibile viene costruito dal codice assembly tramite il **compilatore**
  - La compilazione consiste di due fasi: assemblaggio e linkaggio
- Il programma eseguibile contiene sia istruzioni macchina che dati
  - Contiene un **punto di ingresso** che rappresenta la prima istruzione del programma
- Esempio:
  - `gcc -o sum sum.s`  $\rightarrow$  costruisce il programma eseguibile sum a partire dal sorgente assembly sum.s

# Cosa manca per scrivere un programma?

Come sommiamo i primi 0x100 numeri?

- R0: contatore
- R1: somma parziale

```
mov r0, #0x100  
mov r1, #0
```

```
add r1, r1, r0  
subs r0, r0, #1  
bne ????
```

```
str r1, [???
```

# Aiutare il compilatore

- Le istruzioni assembly non bastano per costruire un programma eseguibile
- Bisogna inoltre permettere di:
  - Indicare la prima istruzione
  - Allocazione di dati (e.g., allocare una cella di memoria con un valore numerico)
  - Indicare delle righe di codice (e.g., nei salti condizionati)
  - Inserire commenti
  - ...

# Direttive e simboli

- **Simboli**: sono delle stringhe che denotano dei particolari valori (indirizzi in memoria, costanti numeriche,...)
  - **Label**: è un particolare simbolo utilizzato per indicare l'indirizzo in memoria di una particolare istruzione
- **Direttive**: sono istruzioni per l'assemblatore che permettono di
  - Definire simboli
  - Allocare aree della memoria per salvataggio di dati
  - Inizializzare il contenuto delle aree di memoria
- Simboli e direttive scompaiono dopo la compilazione

# Esempio di file assembly

Simbolo  
(label)

```
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
```

Commento

```
      ldr r2, =output
      str r1, [r2] @ salva soluzione in memoria
```

Istruzione

```
output: .space 4
iterazioni: .word 0x100
```

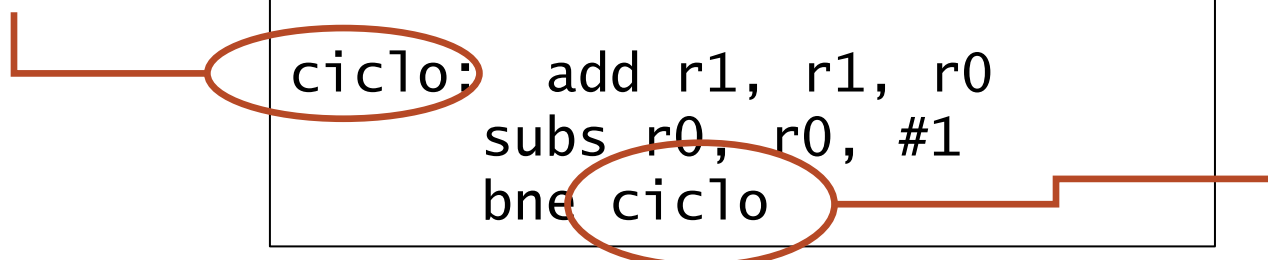
Direttiva



# Definizione e uso di label

Definizione  
di label

```
main: mov r0, #0x100
      mov r1, #0
      ciclo: add r1, r1, r0
            subs r0, r0, #1
            bne ciclo
```



Utilizzo  
della  
label

- Il compilatore calcola l'indirizzo dell'istruzione dove il label "ciclo" è definita
- Ogni volta che il label "ciclo" viene usato viene sostituita dall'indirizzo calcolato

# Punto di inizio

- Ogni programma eseguibile deve avere un punto di inizio, definito dal label `main`

```
.global main  
main:    mov r0, #0x100  
         mov r1, #0
```

- E' necessario inserire prima della definizione del `main` la direttiva `.global main`

# Commenti

Un commento inizia con @ e termina alla fine della riga

Esempio:

```
mov r0, #0x100 @ Numero iterazioni
```

# Sintassi di un'istruzione assembly

- La forma generale delle linee di codice assembly è:

**label: istruzione operandi @commento**

- Ogni campo deve essere separato da uno o più spazi
- Tutti e tre le sezioni ... : ... @ ... sono opzionali
- I label devono iniziare dal primo carattere della riga
- Le istruzioni non cominciano mai dal primo carattere della riga: 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
```

```
stringa: .ascii "Pippo»
```

```
output: .space 4
```

- **.word, .hword, .byte** alloca in memoria e inizializza costanti (4/2/1 byte)
- **.ascii** alloca e inizializza una stringa
- **.space, .skip** allocano aree di memoria (.space inizializza a 0)

# 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
```

# Segmenti di un programma assembly

Un programma assembly è diviso in tre segmenti:

**TEXT** → segmento contenente codice (istruzioni)  
inizia con `.text`

**DATA** → segmento contenente dati inizializzati  
inizia con `.data`

**BSS** → segmento contenente dati non inizializzati  
inizia con `.bss`

# Definizione di altri simboli

- Altre a label è possibile definire simboli per valore numerici con `.equ`

```
ciclo: add r1, r1, r0
      subs r0, r0, #1
      bne ciclo
```

```
.equ OFFSET, #1

ciclo: add r1, r1, r0
      subs r0, r0, #OFFSET
      bne ciclo
```

- `.equ` non definisce un valore in memoria!



# Esempi

`.EQU cost ,5`

Dichiara una costante `cost=5`

`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 l'indirizzo più piccolo

In Moodle trovate le quick reference sul linguaggio ARM e sulle direttive

## 19

# Costrutti di base (if, for)

# If-then-else

```
int a = 10;  
int b = 20;  
int c;  // c= min(a,b)
```

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

# If-then-else in ARM

**.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**

...

## If-then-else in ARM (2)

...

**.text**

LDR R0, =addr\_a @ carica indirizzo di a

LDR R1, [R0] @ copia a in R1

LDR R0, =addr\_b @ carica indirizzo di b

LDR R2, [R0] @ copia b in R2

LDR R0, =addr\_c @ carica indirizzo di c

CMP R1, R2 @ confronta R1(=a) e R2(=b)

BGE else\_case @ se R1>=R2 salta

STR R1, [R0] @ imposta c=a

B end\_if

else\_case: STR R2, [R0] @ imposta c=b

end\_if: NOP

# If-then-else in ARM (short version)

...

**.text**

```
LDR R0, =addr_a    @ carica indirizzo di a
LDR R1, [R0]        @ copia a in R1
LDR R0, =addr_b    @ carica indirizzo di b
LDR R2, [R0]        @ copia b in R2
LDR R0, =addr_c    @ carica indirizzo di c
CMP R1, R2          @ confronta R1 e R2
STRLT R1, [R0]      @ imposta c=a se a<b
STRGE R2, [R0]      @ imposta c=b se a>=b
```

# For loop

```
int sum = 0;  
int n = 100;  
for(int i=1; i<=n; i++){  
    sum = sum + i;  
}
```



# For loop in ARM

**.data**

addr\_n:       .word 100   @ inizializza n=100

**.bss**

add\_sum:       .skip 4     @ alloca sum (4 byte)

**.text**

...

## For loop in ARM (2)

...

**.text**

LDR R3, = addr\_n @ copia indirizzo di n

LDR R2, [R3] @ copia n in R2

MOV R1, #0 @ R1 contiene somma parziale

MOV R0, #0 @ R0 è il contatore i

...

## For loop in ARM (3)

```
...  
for_loop: ADD R1, R1, R0 @ calcola sum=sum+i  
          ADD R0, R0, #1 @ incremento contatore  
          CMP R0, R2      @ confronta se i<=n  
          BLE for_loop    @ se si, ripeti loop  
  
          LDR R3, =addr_sum  
          STR R1, [R3]
```