UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Struttura di un programma Assembly

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 11 marzo 2026

AGENDA DI OGGI

- *Quiz autovalutazione con peer-review*
- Struttura di un programma in Assembly (parte 2)
- *Catena programmativa (parte 2)*
- *Lab#2*

STRUTTURA DI UN PROGRAMMA ASSEMBLY

```
# Title:                                  Filename:
# Author:                                 Date:
# Description:
# Input:
# Output:
################## Data segment ########################
.data
  . . .
################## Code segment ########################
.text
.globl main
main:                                     # main program entry
  . . .
li $v0, 10                                # Exit program
syscall
```

Sezione di commenti (#)

Sezione per dati utenti

Sezione del programma effettivo

preambolo

Dati utente

Istruzioni del programma

STRUTTURA DI UN PROGRAMMA ASSEMBLY

```
# Title:                                  Filename:
# Author:                                 Date:
# Description:
# Input:
# Output:
################## Data segment ########################
.data
  . . .
################## Code segment ########################
.text
.globl main
main:                                     # main program entry
. . .
li $v0, 10                                # Exit program
syscall
```

preambolo

Dati utente

Istruzioni del programma

Direttive all'assemblatore

etichetta

mnemonico di istruzione e nome convenzionale registro

DIRETTIVE DELL'ASSEMBLATORE

NON corrispondono a istruzioni macchina

Sono indicazioni date all'assembler per consentirgli di associare etichette simboliche a indirizzi, allocare spazio di memoria per le variabili, decidere in quali zone di memoria allocare istruzioni e dati

Esempi di direttive

.data <addr> → quel che segue va nel segmento dati (eventualmente dall'indirizzo addr)

.byte b1,....,bn → inizializza i valori in byte successivi in RAM (sezione dati). Es. .byte 5

.word w1,....,wn → inizializza i valori in word successive in RAM (sezione dati).

.text <addr> → quel che segue va nel segmento text (eventualmente dall'indirizzo addr)

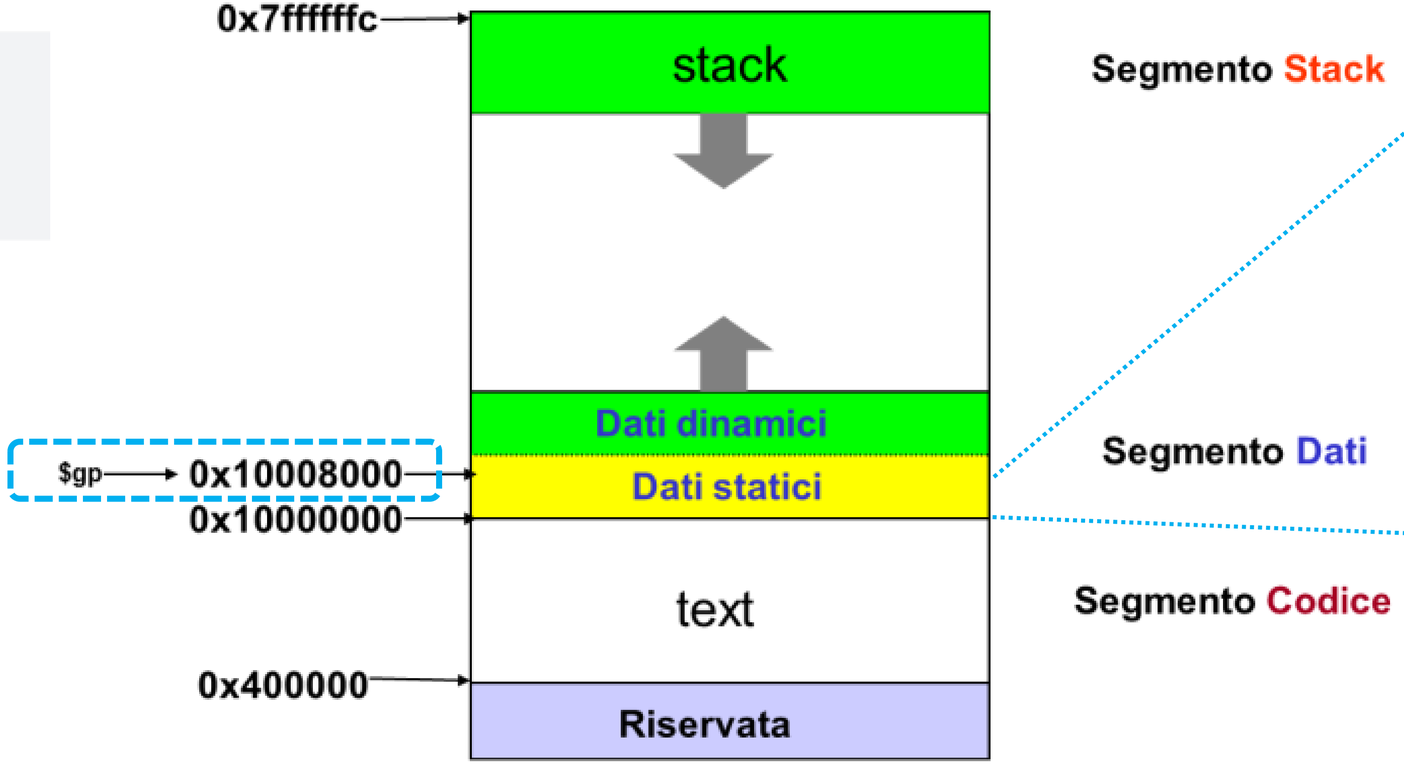
.space 100 → Riserva 100 byte non inizializzati

PROGRAMMARE IN ASSEMBLY

.data: introduce dati da caricare nel segmento dati

Il segmento Dati va da `0x10000000` (praticamente in realtà da `0x10010000`)

I dati sono caricati nel segmento dati a partire dall'indirizzo 0x10000000 (= 228) in modo da poter essere agevolmente indirizzati tramite $gp



Nota sul $gp e l'area «small data».

Esiste una zona chiamata «small data» da 0x10000000 a 0x10010000 (≈64KB).

Se il $gp all'inizio punta a 0x10008000, con una sola istruzione lw (offset di 16 bit in CA2, range ±32KB) si può accedere a tutta l'area(*):

$gp - 32KB ≈ 0x10000000

$gp + 32KB ≈ 0x10010000

(*) **0x10000=1×16^4** = 65536 byte ➔ 65536/1024 = **64 KB**
Offset di **16 bit in CA2 copre un range di [-32768, + 32767] byte,** circa **±32 KB** rispetto a un registro base ($gp).

PROGRAMMARE IN ASSEMBLY

.text introduce istruzioni e word da caricare nel segmento testo (da `0x00400000`)

Esempio:

```
.text
.globl main:
        move $t1,$zero  # copia il contenuto del registro $zero nel registro $t1
        ...
```

PROGRAMMARE IN ASSEMBLY

.space n alloca spazio per n byte nel segmento corrente

E' usato nel segmento dati per dichiarare array ed altre strutture dati.

Esempio: buffer: .space 100

.ascii *stringa* Carica *stringa* in memoria, senza terminarla con *null*

.asciiz *stringa* Carica stringa in memoria terminandola con *null*

Il carattere (non visibile) ***null o \0*** permette di segnalare che la stringa è finita. E' molto usato in vari linguaggi di programmazione (es. C). In In MIPS, le syscall come print_string **leggono la stringa fino al primo \0**.

Esempio:

.asciiz "Immettere dati:\n"

Va a capo

PROGRAMMARE IN ASSEMBLY

Caricare più numeri nei formati byte, halfword, word, float e double in posizioni consecutive di memoria

.byte b1, ..., bn

.half h1, ..., hn

.word w1, ..., wn

.float f1, ..., fn

.double d1, ..., dn

Gli interi sono esprimibili in decimale o esadecimale.

I float e double vanno espressi in decimale, con "." obbligatorio dopo parte intera (esponente "e" opzionale).

Esempio: pi: .float 3.14

PROGRAMMARE IN ASSEMBLY

`__start:` E' l'etichetta che, per default, specifica la prima istruzione eseguibile del programma.

E' di solito definita nell'*exception handler* (**codice di start-up**, caricato all'inizio della zona text della memoria), dove è associata al codice

```
__start:
    . . .
    jal main
    . . .
    li $v0, 10  # syscall 10 (Exit)
    syscall
```

In questo caso **è sufficiente che il programma contenga la procedura main.**

Nota: in QtSpim il codice di start-up è pre-caricato (*exceptions.s*).

ISTRUZIONI VARIE (Appendix A, da pagina A-51)

Istruzioni per operazioni aritmetiche

Istruzioni per operazioni logiche

Branch

Jump

Confronti

Manipolazione di costanti

Load/store

FUNZIONI LOGICHE PRINCIPALI

AND o prodotto logico

A	B	A•B
0	0	0
0	1	0
1	0	0
1	**1**	**1**

OR o somma logica

A	B	A+B
0	0	0
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

XOR (OR esclusivo)

A	B	A⊕B
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

NOT o negazione logica

A	$\bar{A}$
0	**1**
1	0

FUNZIONI LOGICHE DERIVATE

NAND

A	B	A•B	A NAND B
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

NOR

A	B	A+B	A NOR B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

FUNZIONI LOGICHE SU SEQUENZE DI BIT

E’ possibile applicare queste funzioni anche a sequenze di bit. Si procede al **confronto bit per bit**, su bit di ordine corrispondente nelle due sequenze

A = 11001100

B = 10101010

A = 11001100

AND B = 10101010

= 10001000 = 0x88

```
.text
main:
    li $t0, 0xCC          # A = 11001100
    li $t1, 0xAA          # B = 10101010
    and $t2, $t0, $t1     # $t2 = A AND B = 10001000
```

ISTRUZIONI PER OPERAZIONI ARITMETICHE E LOGICHE

`add rd, rs, rt`
- addizione rs + rt -> rd
- (con overflow...ne parliamo in seguito...)

`addi rd, rs, imm`
- addizione immediata
- sign-extended imm + rs -> rd (con overflow)

`sll rd, rt, shamt`
- shift left rt della distanza shamt -> rd

`and rd, rs, rt`
- and bit a bit di rs e rt -> rd

`or rd, rs, rt`
- or bit a bit (logical or) di rs e rt -> rd

`ori rt, rs, imm`
- or bit a bit (logical or) di rs e zero-extended imm -> rt

PSEUDOISTRUZIONI

- istruzione assembly che non ha una corrispondente istruzione macchina
- tradotta dall'assembler in una sequenza di istruzioni

Esempi

```
li rdest, imm
```

- load immediate
- caricare una costante di 32 bit nel registro rdest

```
li $s0, 0x12345678
```

tradotta dall'assemblatore come in slide successiva

```
move $t0, $t1
```

tradotta dall'assemblatore in `add $t0, $zero, $t1`

MANIPOLAZIONE COSTANTI

Il caso più semplice è di voler caricare un valore in un registro (come fatto manualmente durante il lab).

Si potrebbe usare un'istruzione come:

Esempio
li $t0, 10
load immediate

li è pseudoistruzione, quindi diventa:

Esempio
addiu $t0, $zero, 10
add immediate unsigned

Se il valore entra in **16 bit con segno**, l'assembler usa una sola istruzione **I-type.**

*...e se si vuole **caricare una costante di 32 bit?*** Immediate è di 16 bit.

Esempio: caricare in $s0 il valore 0x12345678

```
li $t0, 0x12345678    # l'assemblatore espande la pseudoistruzione in 2 istruzioni

lui $s0, 0x1234          # load upper immediate
# contenuto di $s0: 0x12340000

ori $s0, $s0, 0x5678    # OR bit a bit tra un registro e un immediate («i»)
# contenuto di $s0: 0x12340000 | 0x5678 = 0x12345678
```

NOMI SIMBOLICI RISERVATI

Nome Simbolico	Numero	Uso
$zero	0	Costante 0
$at	1	Assembler temporary
$v0-$v1	2-3	Functions and expressions evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

- usati da assembler, compilatore, sistema operativo
- secondo specifiche convenzioni
- da trattare con cautela se si programma in assembly!!! (sono nomi riservati)

ESEMPIO DI PROGRAMMA ASSEMBLY

```
.data
vet: .word 2, 4, 6, 8, 10  # vettore di 5 interi

.text
.globl main

main:
    la $t0, vet         # Carica in $t0 l'indirizzo base del vettore
    lw $t1, 0($t0)      # Carica primo elemento (vet[0]=2) in $t1
    lw $t2, 16($t0)     # Carica terzo elemento (vet[2]=10) in $t2
    add $t3, $t1, $t2 # Somma $t1+$t2 → risultato in $t3
                       # risultato finale in $t3 (=12)
```

Nota. lw $t2, **16**($t0) = lw $t2, **0x10**($t0)

ASSEMBLY E SISTEMA OPERATIVO

Il sistema operativo (SO) è un insieme di programmi che:

- stanno in un'area protetta della memoria (*kernel*)
- svolgono funzioni di utilità generale (in particolare, I/O) richiamabili dai programmi utente.

Il simulatore QtSpim fornisce alcune funzioni elementari che simulano alcune funzionalità base del SO richiamabili attraverso il meccanismo di **syscall**, concettualmente analogo a una chiamata a procedura

PROGRAMMARE IN ASSEMBLY

Chiamate a procedure di I/O del sistema operativo

Modalità d'uso:

– mettere il/i parametro/i nei registri ($a0, $a1)

– specificare il tipo di system call, scrivendo un codice opportuno nel registro $v0

– syscall

Esempio: stampa dell'intero con segno in $t2

```
move $a0, $t2      # $a0=$t2
li $v0, 1          # codice 1 in $v0
syscall            # chiamata di sistema
```

SYSCALL

SYS-CALL: chiamata al sistema (operativo)

Analogo a una chiamata a procedura:

- Convenzioni per le syscall: Tabella a pag. A43 (Appendice A)
- Impostare i parametri nei registri $a0-$a3 (come da tabella)
- Impostare nel registro $v0 il codice della chiamata

Syscall essenziali:

- exit codice 10: uscita dalla procedura
- exit2 codice 17: terminazione del programma!
- read_int
- print_int codice 1 (parametro passato per valore!!)
- read_string
- print_string (parametro passato per indirizzo!!)

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

FIGURE A.9.1 System services.

Esempio

```
.data
msg: .asciiz "ciao"

.text
la $a0, msg
li $v0, 4
syscall
```

SYSCALL: ESEMPIO

Scrivere un codice che stampa la stringa «La risposta è 5 »

```
.data
str: .asciiz “La risposta è “
numero: .word 5

.text
.globl main
main:
li $v0, 4              #Codice della chiamata di sistema per print_str
la $a0, str            #Indirizzo stringa da stampare (passato per indirizzo!)
syscall                #Stampa la stringa
li $v0, 1              #Codice della chiamata di sistema per print_int
lw $a0, numero         #Intero da stampare (passato per valore!!!)
syscall                #Stampa l’intero
```

Materiale per la lezione

- *Hennessy-Patterson, cap. 1 pp.14-16*
- *Appendix A.2, A.3, A.4, A.5*