

# Architettura degli Elaboratori

Le istruzioni della CPU - Controllo del flusso



SAPIENZA  
UNIVERSITÀ DI ROMA

Alessandro Checco

[alessandro.checco@uniroma1.it](mailto:alessandro.checco@uniroma1.it)

[alessandrochecco.github.io](https://alessandrochecco.github.io)

Special thanks and credits:

Andrea Sterbini, Iacopo Masi,

Claudio di Ciccio

[S&PdC] 2.1 – 2.8



# Argomenti

---

## Argomenti della lezione

- Istruzioni Logiche
- Istruzioni per prendere decisioni (if then else, cicli)
- Il simulatore MARS
- Esercitazione

Scaricate il simulatore MARS da

<http://courses.missouristate.edu/kenvollmar/mars/index.htm>

- è scritto in Java (**gira su qualsiasi OS**)
- è **estensibile** (diversi plug-in per esaminare il comportamento della CPU e della Memoria)
- è un IDE integrato:
  - **Editor con evidenziazione della sintassi** assembly ed **autocompletamento** delle istruzioni
  - **Help completo** (istruzioni, syscall ...)
  - **Simulatore** con possibilità di **esecuzione passo-passo** e uso di **breakpoint**
  - Permette l'**esecuzione da riga di comando** e la **compilazione di più file**

---

## Istruzioni per prendere decisioni (istruzioni condizionali)



SAPIENZA  
UNIVERSITÀ DI ROMA

# Comparazione e salto condizionato

- `beq $s1,$s2,C` # Branch on equal
- `bne $s1,$s2,C` # Branch on not equal
- `blez $s1,C` # Branch on less-than or eq. 0
- `bgez $s1,C` # Branch on greater-than or eq.0
- `bltz $s1,C` # Branch on less than 0
- `bgtz $s1,C` # Branch on greater than 0

C è una ETICHETTA  
che identifica  
un'istruzione

Pseudocodice delle azioni sottostanti  
`PC += 4`  
`if ($s1 == $s2) { PC += C << 2 }`

## Comparazione e salto condizionato

---

- `beq $s1,$s2,C`                      # Branch on equal
- `bne $s1,$s2,C`                      # Branch on not equal
- `blez $s1,C`                          # Branch on less-than or eq. 0
- `bgez $s1,C`                          # Branch on greater-than or eq.0
- `bltz $s1,C`                          # Branch on less than 0
- `bgtz $s1,C`                          # Branch on greater than 0
  
- `slt $s0,$s1,$s2`                      # Set \$s0 to 1 if \$s1 l.than \$2
- `slti $s0,$s1,10`                      # Set \$s0 to 1 if \$s1 l.than 10

---

# MARS



SAPIENZA  
UNIVERSITÀ DI ROMA

# MARS – MIPS Assembler and Runtime Simulator



# MARS

---

Scaricate il simulatore MARS da

<http://courses.missouristate.edu/kenvollmar/mars/index.htm>

- è scritto in Java (**gira su qualsiasi OS**) ma vi serve la JVM (Java Virtual Machine)
- è **estensibile** (diversi plug-in per esaminare il comportamento della CPU e della Memoria)
- è un IDE integrato:
  - **Editor con evidenziazione della sintassi** assembly ed **autocompletamento** delle istruzioni
  - **Help completo** (istruzioni, syscall ...)
  - **Simulatore** con possibilità di **esecuzione passo-passo** e uso di **breakpoint**
  - Permette l'**esecuzione da riga di comando** e la **compilazione di più file**



# MARS - avvio

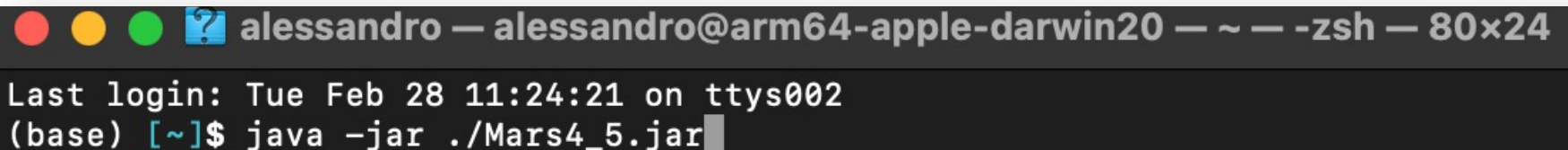
## Windows

Se avete JVM, potete semplicemente cliccarci sopra (**doppio click**). Dovrebbe funzionare

## Mac OS - Linux

Su Mac OS, se avete installato ultima versione JVM funziona anche con il doppio click ma non è facile salvare il file nella cartella dove si trova il jar. (I file di default vengono salvati in un dir temporanea). Quindi lo apro direttamente dalla dir in cui risiede da linea di comando.

Su Linux il doppio click funziona, ma in entrambi i casi (Mac e Linux) si può lanciare anche da riga di comando così:



```
alessandro — alessandro@arm64-apple-darwin20 — ~ — -zsh — 80x24
Last login: Tue Feb 28 11:24:21 on ttys002
(base) [~]$ java -jar ./Mars4_5.jar
```

# Il simulatore MARS

The image shows the MARS MIPS simulator interface. The main window is the **EDITOR**, which contains assembly code for calculating Fibonacci numbers. A yellow oval labeled **EDITOR** points to the code area. A yellow oval labeled **Auto Completamento** points to a dropdown menu that lists MIPS instructions like `add`, `addi`, `addiu`, etc. A yellow oval labeled **Syntax highlight** points to the assembly code, which is color-coded by instruction type. On the right side, there is a **REGS** (Registers) panel showing the state of MIPS registers, with a yellow oval labeled **REGS** pointing to it. At the bottom, there is a **CONSOLE** panel for input/output, with a yellow oval labeled **CONSOLE Stampe / Input da tastiera** pointing to it. The title bar of the window reads "D:\2010F CSC 320\MIPS assembly code\FibonacciWithStack.asm - MARS 4.0.1".

**EDITOR**

```
1 # Ken Vollmar
2 # Feb. 13, 2002
3 # Computing the i-th Fibonacci number using the stack
4
5 .data
6     # Put any data initializations between here and the .text section
7
8     str1: .asciiz "Which Fibonacci number do you want? "
9     str2: .asciiz "The Fibonacci number is "
10
11 .text
12
13     addi $sp, $sp, -4
14     lui $t0, 0
15     addi $t0, $t0, 0
16     sysc
17     addi $t0, $t0, 0
18     addi $t0, $t0, 0
19     addi $t0, $t0, 0
20     addi $t0, $t0, 0
21     addi $t0, $t0, 0
22     addi $t0, $t0, 0
23     addi $t0, $t0, 0
24     addi $t0, $t0, 0
25     addi $t0, $t0, 0
26     addi $t0, $t0, 0
27     addi $t0, $t0, 0
28     addi $t0, $t0, 0
29     addi $t0, $t0, 0
30     addi $t0, $t0, 0
31     addi $t0, $t0, 0
32     addi $t0, $t0, 0
33     addi $t0, $t0, 0
34     addi $t0, $t0, 0
35     addi $t0, $t0, 0
36     addi $t0, $t0, 0
37     addi $t0, $t0, 0
38     addi $t0, $t0, 0
39     addi $t0, $t0, 0
40     addi $t0, $t0, 0
41     addi $t0, $t0, 0
42     addi $t0, $t0, 0
43     addi $t0, $t0, 0
44     addi $t0, $t0, 0
45     addi $t0, $t0, 0
46     addi $t0, $t0, 0
47     addi $t0, $t0, 0
48     addi $t0, $t0, 0
49     addi $t0, $t0, 0
50     addi $t0, $t0, 0
51     addi $t0, $t0, 0
52     addi $t0, $t0, 0
53     addi $t0, $t0, 0
54     addi $t0, $t0, 0
55     addi $t0, $t0, 0
56     addi $t0, $t0, 0
57     addi $t0, $t0, 0
58     addi $t0, $t0, 0
59     addi $t0, $t0, 0
60     addi $t0, $t0, 0
61     addi $t0, $t0, 0
62     addi $t0, $t0, 0
63     addi $t0, $t0, 0
64     addi $t0, $t0, 0
65     addi $t0, $t0, 0
66     addi $t0, $t0, 0
67     addi $t0, $t0, 0
68     addi $t0, $t0, 0
69     addi $t0, $t0, 0
70     addi $t0, $t0, 0
71     addi $t0, $t0, 0
72     addi $t0, $t0, 0
73     addi $t0, $t0, 0
74     addi $t0, $t0, 0
75     addi $t0, $t0, 0
76     addi $t0, $t0, 0
77     addi $t0, $t0, 0
78     addi $t0, $t0, 0
79     addi $t0, $t0, 0
80     addi $t0, $t0, 0
81     addi $t0, $t0, 0
82     addi $t0, $t0, 0
83     addi $t0, $t0, 0
84     addi $t0, $t0, 0
85     addi $t0, $t0, 0
86     addi $t0, $t0, 0
87     addi $t0, $t0, 0
88     addi $t0, $t0, 0
89     addi $t0, $t0, 0
90     addi $t0, $t0, 0
91     addi $t0, $t0, 0
92     addi $t0, $t0, 0
93     addi $t0, $t0, 0
94     addi $t0, $t0, 0
95     addi $t0, $t0, 0
96     addi $t0, $t0, 0
97     addi $t0, $t0, 0
98     addi $t0, $t0, 0
99     addi $t0, $t0, 0
100    addi $t0, $t0, 0
```

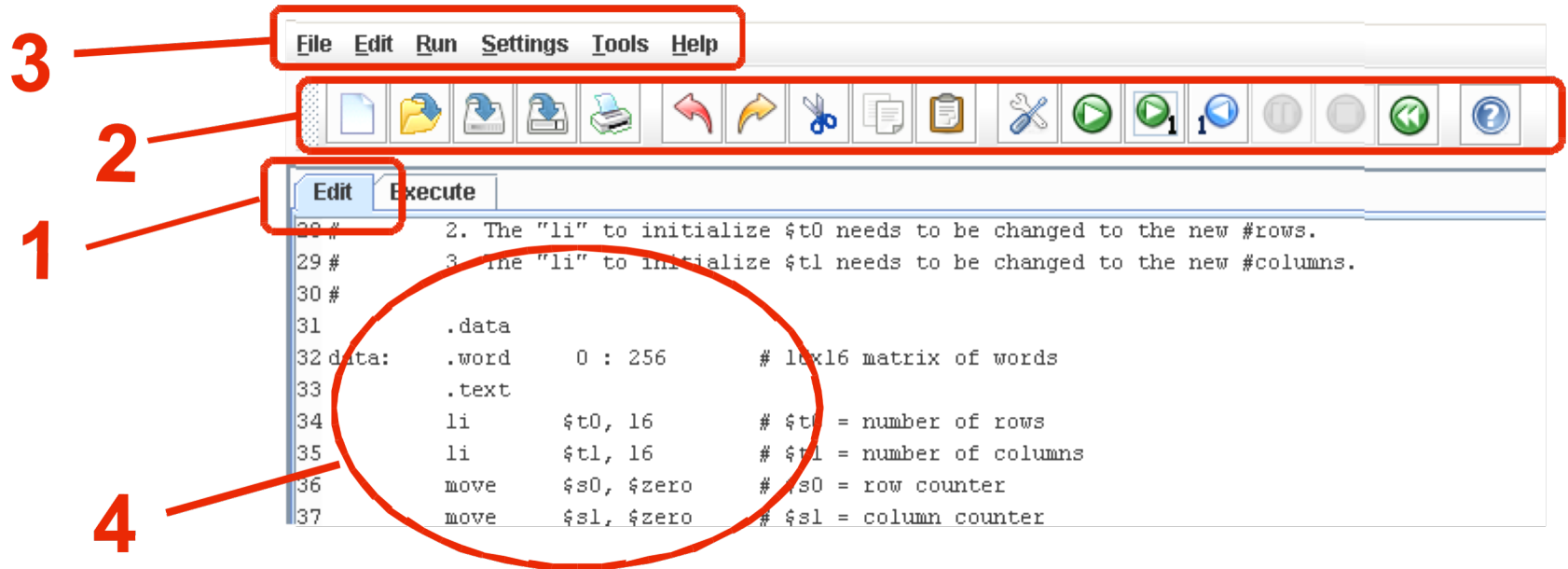
**Auto Completamento**

**Syntax highlight**

**REGS**

**CONSOLE**  
Stampe / Input da tastiera

# MARS: la toolbar



1 Interfaccia a tab (uno per ciascun file aperto, più il simulatore)

2 Toolbar (Compilazione, Esecuzione, Esecuzione passo-passo in avanti e indietro, Reset)

3 Menu

4 Finestra dell'editor

# Assembly MIPS

---

## Direttive principali per l'assemblatore

<b>.data</b>	definizione dei dati statici
<b>.text</b>	definizione del programma
<b>.ascii</b>	stringa terminata da zero
<b>.byte</b>	sequenza di byte
<b>.double</b>	sequenza di double
<b>.float</b>	sequenza di float
<b>.half</b>	sequenza di half words
<b>.word</b>	sequenza di words
<b>.globl</b> <i>sym</i>	dichiara il simbolo come globale e può essere referenziato da altri file

## Codici mnemonici delle **istruzioni**

add, sub, div, beq ...

## Codifica mnemonica dei **registri**

\$a0, \$sp, \$ra ... \$f0, \$f31

## **Etichette** (per calcolare gli indirizzi relativi) nome:

## L'assemblatore converte

- dal testo del programma in assembly
- al programma in codice macchina

## -Dalle etichette calcola gli indirizzi

Dei salti relativi

Delle strutture dati in memoria (offset)

Dei salti assoluti

**NOTA:** le strutture di controllo del flusso del programma vanno realizzate «a mano» usando i **salti condizionati e le etichette**

# Istruzioni condizionali

---

Alla fine di questa lezione sapremo come il calcolatore implementa i costrutti maggiormente usati in una programmazione di alto livello come C o C++:

Mappiamo in assembly le seguenti istruzioni:

1. **If then else**
2. **Iterazioni (cicli)**
  1. Do while loop
  2. While loop
  3. For loop
3. **If then else con «casi multiple» (switch statement)**

# Prendere decisioni (if then else)

---

## Esempio C

```
if ( X > 0 ) {  
  
    // codice da eseguire se il test è vero  
  
} else {  
    // codice da eseguire se il test è falso  
}  
  
// codice seguente
```


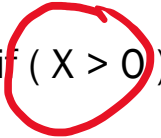
## Esempio Assembly

```
.text  
# uso il registro $t0 per la var. X  
blez $t0, else      # test X <= 0  
  
# codice da eseguire se il test è vero  
  
j endIF             # esco dall'IF  
  
else:  
# codice da eseguire se il test è falso  
  
endif:  
# codice seguente
```



NOTA: il test inserito è l'opposto dell'originale

# Prendere decisioni (if then else)

## Esempio C

  
  
`if ( X > 0 ) {  
 // codice da eseguire se il test è vero  
  
} else {  
 // codice da eseguire se il test è falso  
  
}  
// codice seguente`

## Esempio Assembly

`.text  
# uso il registro $t0 per la var. X  
blez $t0, else # test X <= 0  
# codice da eseguire se il test è vero  
  
j endif # esco dall'IF  
# codice da eseguire se il test è falso  
  
endif:  
# codice seguente`  
  


NOTA: il test inserito è l'opposto dell'originale


# Prendere decisioni (if then else)

## Esempio C

```
if ( X > 0 ) {  
    // codice da eseguire se il test è vero  
  
} else {  
    // codice da eseguire se il test è falso  
  
}  
  
// codice seguente
```

## Esempio Assembly

```
.text  
# uso il registro $t0 per la var. X  
blez $t0, else      # test X <= 0  
  
# codice da eseguire se il test è vero  
  
j endif             # esco dall'IF  
  
# codice da eseguire se il test è falso  
  
endif:  
# codice seguente
```

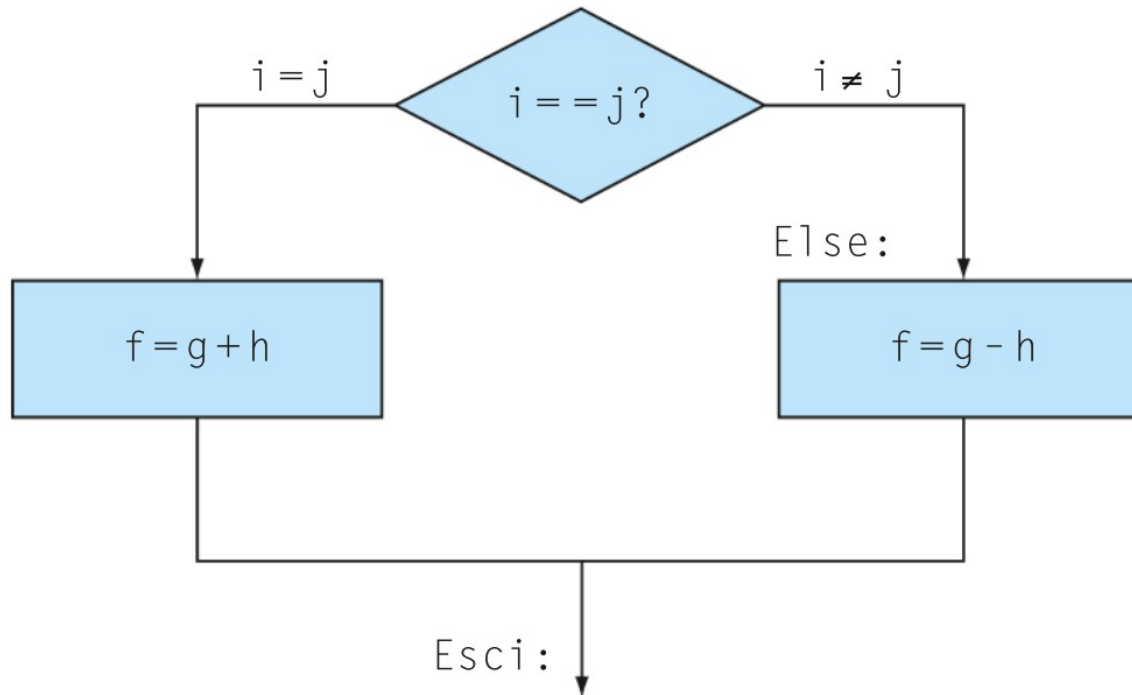


NOTA: il test inserito è l'opposto dell'originale



## Ancora, Prendere decisioni (if then else)

```
if (i == j) f = g + h; else f = g - h;
```



## Prendere decisioni (if then else)

---

```
if (i == j) f = g + h; else f = g - h;
```

```
bne $s3,$s4,Else # vai a Else se i ≠ j
```

## Prendere decisioni (if then else)

---

```
if (i == j) f = g + h; else f = g - h;
```

```
bne $s3,$s4,Else # vai a Else se i ≠ j
```

```
add $s0,$s1,$s2 # f = g + h (saltata se i ≠ j)
```

## Prendere decisioni (if then else)

---

```
if (i == j) f = g + h; else f = g - h;
```

```
bne $s3,$s4,Else # vai a Else se i ≠ j
```

```
add $s0,$s1,$s2 # f = g + h (saltata se i ≠ j)
```

```
j Esci # vai a Esci
```

## Prendere decisioni (if then else)

---

```
if (i == j) f = g + h; else f = g - h;
```

```
bne $s3,$s4,Else # vai a Else se i ≠ j
```

```
add $s0,$s1,$s2 # f = g + h (saltata se i ≠ j)
```

```
j Esci # vai a Esci
```

```
Else: sub $s0,$s1,$s2 # f = g - h (saltata se i == j)  
Esci:
```

# Realizzare Iterazioni (ciclo do while)

---

## Esempio C

```
do {
```

```
    // codice da ripetere se x != 0
```

```
    // il corpo del ciclo DEVE aggiornare x
```

```
} while (x != 0);
```

```
    // codice seguente
```

## Esempio Assembly

```
.text
```

```
    # uso il registro $t0 per l'indice x
```

```
do:
```

```
    # codice da ripetere
```

```
bnez $t0, do    # test x != 0
```

```
    # codice seguente
```

NOTA: il test inserito è uguale all'originale

# Realizzare Iterazioni (while do)

---

## Esempio C

```
while (x != 0) {  
  
    // codice da ripetere se x != 0  
    // il corpo del ciclo DEVE aggiornare x  
}  
  
// codice seguente
```

## Esempio Assembly

```
.text  
    # uso il registro $t0 per l'indice x  
while:  
    beqz $t0, endWhile    # test x == 0  
  
    # codice da ripetere  
  
    j while                # loop  
  
endWhile:  
    # codice seguente
```

NOTA: il test inserito è l'opposto dell'originale

# Realizzare Iterazioni (for loop)

---

## Esempio C

```
for (i=0 ; i<N ; i++)  
{  
    // codice da ripetere  
}  
  
// codice seguente
```

## Esempio Assembly

```
.text  
  
# uso il registro $t0 per l'indice i  
# uso il registro $t1 per il limite N  
xor $t0, $t0, $t0      # azzero i  
li $t1, N               # limite del ciclo  
  
cicloFor:  
bge $t0, $t1, endFor  # test i>=N  
# codice da ripetere  
addi $t0, $t0, 1        # incremento di i  
j cicloFor              # loop  
  
endFor:  
# codice seguente
```

NOTA: il test inserito è l'opposto dell'originale



# SWITCH CASE

---

## Esempio C

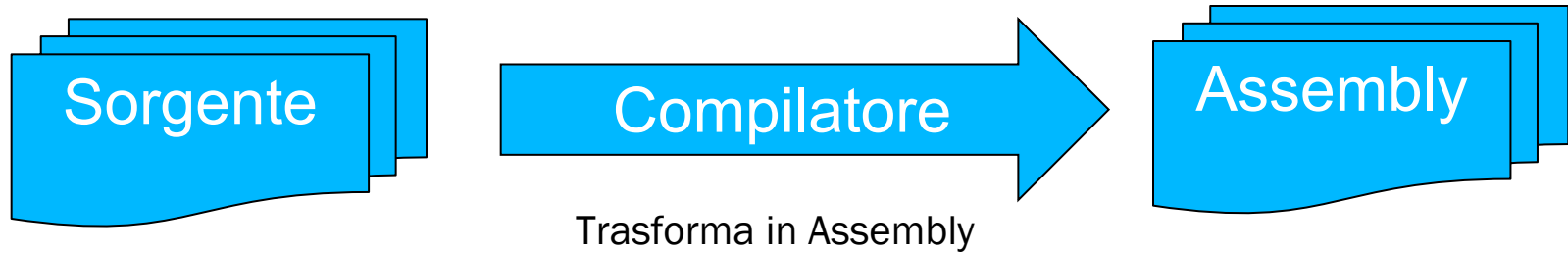
```
switch (A) {  
  case 0:    // codice del caso 0  
    break;  
  case 1:    // codice del caso 1  
    break;  
  // altri casi  
  case N:    // codice del caso 3  
    break;  
}  
// codice seguente
```

## Esempio Assembly

```
.text  
  
    sll $t0, $t0, 2        # A*4  
    lw $t1, dest($t0)      # carico indirizzo +$t0  
    jr $t1                 # salto a registro  
  
caso0:    # codice del caso 0  
    j endSwitch  
  
caso1:    # codice del caso 1  
    j endSwitch  
  
# altri casi  
casoN:    # codice del caso N  
    j endSwitch  
  
endSwitch:  
    # codice seguente  
  
.data  
dest:     .word caso0, caso1, ..., casoN
```

# Compilatore / Assemblatore

---



Istruzioni/espressioni di alto livello => gruppi di istruzioni ASM

variabili temporanee => registri

Variabili globali e locali => etichette e direttive

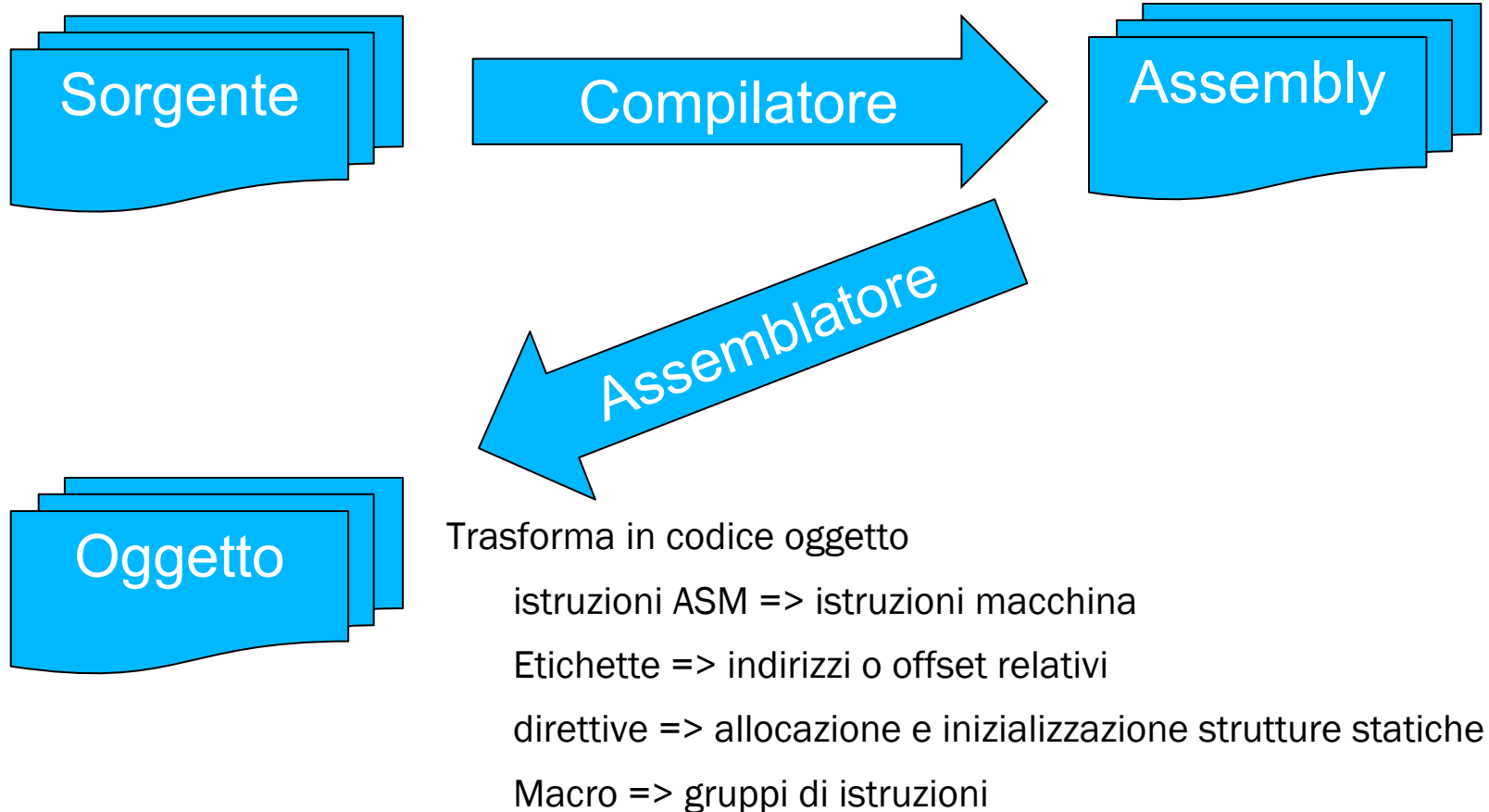
Strutture di controllo => salti ed etichette

funzioni e chiamate => etichette e salti a funzione

chiamate a funzioni esterne => tabella x linker

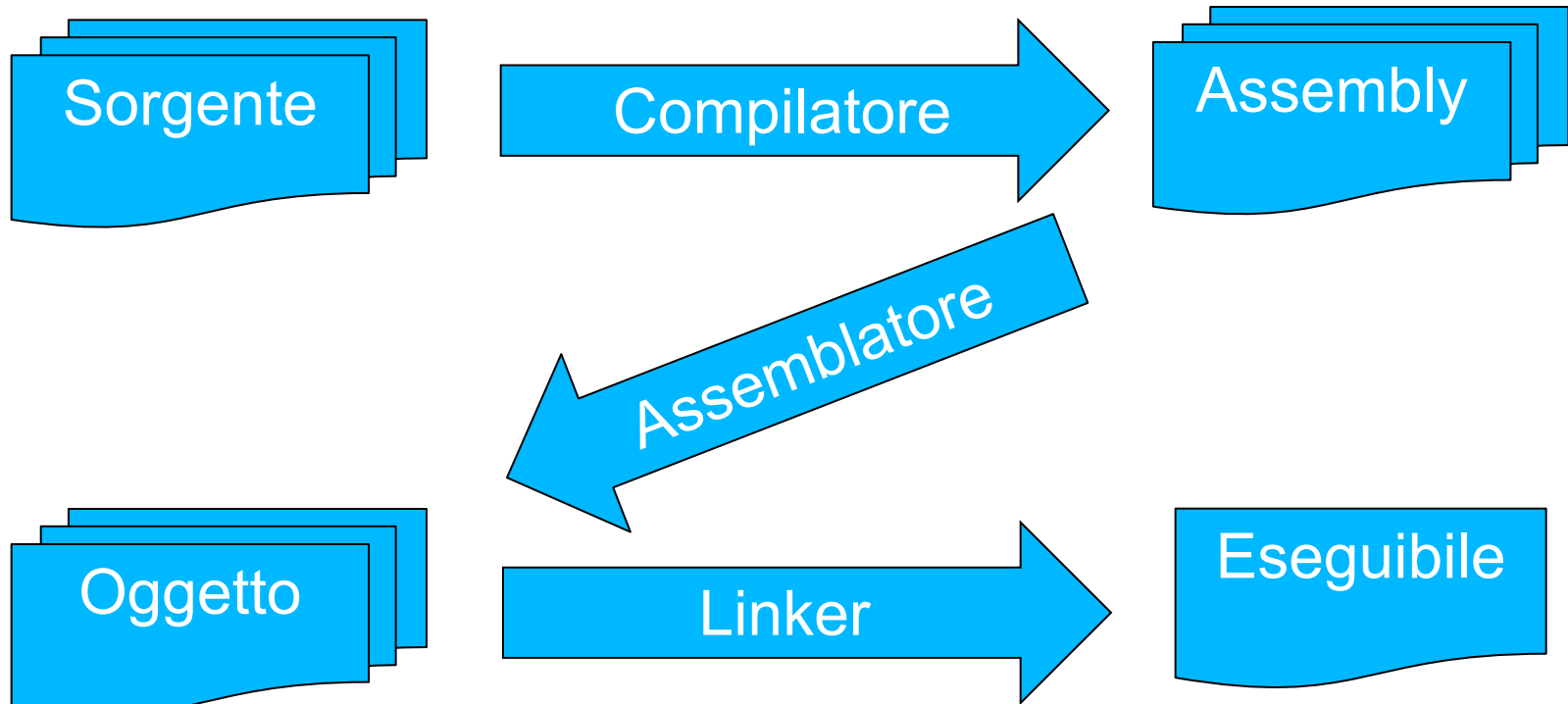
# Compilatore / Assemblatore

---



# Compilatore / Assemblatore

---



Definisce la pos. In memoria delle strutture dati statiche e del codice

«Collega» i riferimenti a

chiamate di funzioni esterne => salti non relativi

strutture dati esterne => indirizzamenti non relativi

# La direttiva globl

---

La useremo ma non è essenziale quando si lavora con un solo file come nel nostro caso.

E' utile quando abbiamo **più file da gestire** e abbiamo delle parti del codice che referenziano etichette che sono in file diversi.

Chi dice all'assemblatore dove andare a prendere le etichette non definite in quel file?  
Possiamo farlo con la direttiva globl.

```
.globl main  
.text  
main:  
    addi $t1,$zero,-100
```

File main.S

```
....  
.text  
  
j main  
  
.....  
.....
```

File file\_B.S

```
% as -o main.o main.S
```

 Assembla e crea file oggetto

```
% as -o file_B.o file_B.s
```

 Assembla e crea file oggetto

```
% ls -o main.bin file_B.o main.o
```

 Linka il tutto e crea eseguibile (su windows sarebbero i file .exe)

# La direttiva `globl`

La useremo ma non è essenziale quando si lavora con un solo file come nel nostro caso.

E' utile quando abbiamo **più file da gestire** e abbiamo delle parti del codice che referenziano etichette che sono in file diversi.

Chi dice all'assemblatore dove andare a prendere le etichette non definite in quel file?  
Possiamo farlo con la direttiva `globl`.

```
.globl main
.text
main:
    addi $t1,$zero,-100
```

File `main.S`

```
....
.text

j main

.....
.....
```

File `file_B.S`

Se i global non sono specificati bene  
si verifica un errore al momento di  
**Linking** (quando si crea eseguibile)  
Perché il linker non sa come risolvere  
`label_fileB`

```
% as -o main.o main.S
```

 Assembla e crea file oggetto

```
% as -o file_B.o file_B.s
```

 Assembla e crea file oggetto

```
% ls -o main.bin file_B.o main.o
```

 Linka il tutto e crea eseguibile (su windows sarebbero i file `.exe`)

# Ancora sui registri MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	results from callee
3	v1	returned to caller
4	a0	arguments to callee
5	a1	from caller: caller saves
6	a2	
7	a3	
8	t0	temporary
...		
15	t7	
16	s0	callee saves
...		
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return Address caller saves

# Le pseudoistruzioni

---

```
lui $1,0x00001001    14:  lw $s0, values    # carico a in S0  
lw $16,0x00000000($1)
```

- La mia istruzioni sulla destra è stata mappata in due istruzioni sulla sinistra dall'assemblatore.
- Inoltre notate come viene usato il registro di appoggio **\$1 (\$at)** per caricare indirizzo etichetta **values**

values = 0x10000004

lw \$t0,values == lui \$1,0x1000

lw \$8,4(\$1)



# Le pseudoistruzioni

---

```
addu $8,$0,$16      21:  move $t0,$s0
```

- Il move è fittizio. E' in realtà stato implementato come **addizione senza segno** fra il registro 16 e il registro 0 (che come sapete contiene sempre zero)

# Le pseudoistruzioni

---

```
slt $1,$8,$17      23:  ble $s1,$t0,checkC  
beq $1,$0,0x00000001
```

- Il **ble** (branch if less than) è fittizio. E' in realtà stato implementato **set if less than**. Nuovamente il risultato del set è scritto su registro **\$1** che è **\$at**.
- Successivamente viene usato **beq** controllando **\$1** per decidere se saltare a **0x00000001** (indirizzo memoria di **checkC**)
- Da notare ancora **beq** confronta **\$1** con **\$0** (registro di tutti zero)
- Abbiamo già incontrare diverse istruzioni reali e come vi ho detto il registro **\$zero** è già comparso diverse volte. **Velocizzare le situazioni più comuni** quindi è importante che la codifica della zero sia sempre disponibile

## Es.: trova il max di un vettore

### Esempio C

```
// definizione dei dati
int vettore[6] = { 11, 35, 2, 17, 29, 95 };
int N = 6;

int max = vettore[0];
// scandisco il vettore
for (i=1 ; i<N ; i++) {

    int el = vettore[i];    // el. corrente
    if (elemento > max)
        max = elemento;

}
```

### Esempio Assembly

```
.data
vettore:    .word 11, 35, 2, 17, 29, 95
N:         .word 6

.text
    lw      $t0, vettore($zero)    # max → $t0
    lw      $t1, N                  # N → $t1
    li      $t2, 1                   # i = 1
for: bge    $t2, $t1, endFor
    sll     $t3, $t2, 2               # i*4
    lw      $t4, vettore($t3)       # el. = vettore[i]
    ble     $t4, $t0, else           # if (el >= max)
    move    $t0, $t4                 # max = el.
else:
    addi    $t2, $t2, 1              # i++
    j for
endFor:
```

# Compito per casa (o per qualsiasi altro edificio 😊)



1. Installare **MARS** e prendere familiarità con interfaccia
2. Scrivere i programmi visti a lezione e provare ad eseguirli e debuggarli passo passo
  - Controllare come cambiano i registri sulla destra in base ai passi svolti.
  - Ispezionare come cambiano le pseudo-istruzioni immesse nelle istruzioni che poi svolge veramente il calcolatore
3. Partendo dal programma che trova il max in un vettore scrivere un programma in linguaggio assembly MIPS con MARS che dato un vettore ingresso **vector** e la sua dimensione **N** calcoli due somme dei numeri del vettore.



1. La prima somma deve sommare i valori del vettore di indice **dispari**. (Indice parte da 1)
2. La seconda somma deve sommare i valori di un vettore con **indice pari**. (Indice parte da 0)

Vector .word 4, -1, 5, 500, 0, 10000, -256  
N .word 5

Somme: .word 0, 0

Una volta scritto con questi dati cambiate N e controllate se continua a funzionare

Name	Num...	Value
\$ZERO	0	0
\$at	1	0
\$v0	2	0
\$v1	3	0
\$a0	4	0
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	0
\$t2	10	0
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194384
hi		0
lo		0