ASSEMBLY MIPS – GUIDA RAPIDA ALLA COMPRENSIONE

Autore: Borsato Claudio

Bibliografia: *Strutture e progetto dei calcolatori – Patterson, Hennessy*
Programmare in Assembly in GNU/Linux con sintassi AT&T – Ferroni
Lezione 10 Architettura degli Elaboratori - Orlando

INDICE:

1 – COS'È L'ASSEMBLY

L'Assembly è un linguaggio a basso livello utilizzato per scrivere istruzioni comprensibili dal computer in una maniera “umana”. Per rendere questo linguaggio comprensibile al computer viene usato un Assembler che traduce in codice binario le varie istruzioni. Per esempio:

```
00100111110001111000110001001110        add $s0, $s1, $s2
```

Il linguaggio Assembly per quanto complicato può risultare estremamente potente, per quel che riguarda il dover usare istruzioni che modificano determinate locazioni di memoria che con normali linguaggi compilati/interpretati sarebbe impossibile. Infatti, alcuni linguaggi (per esempio C) permettono l'inserimento di codice Assembly all'interno del proprio sorgente, così da poterlo eseguire direttamente.

2 – COMPILER / ASSEMBLER / LINKER

L'esecuzione di un programma può avvenire in 2 maniere differenti, o viene interpretato, o viene compilato (anche se in alcuni casi, come *Java* e *Python*, avviene prima una cosa e poi l'altra).
Nel primo caso, ovvero nel caso il linguaggio sia interpretato (es: *Perl, Ruby, OCaML)* il programma verrà eseguito traducendo le istruzioni eseguite, saltando tutte le altre. Questo rende il programma versatile per operazioni automatizzate (script) ma diventa pesante quando si tratta di creare programmi veri e propri. Per questo viene usato un Compiler. (linguaggi compilati es: *C*, *Fortran, Pascal*) Un **Compiler**, è un programma che ricevuto un codice sorgente, restituisce un eseguibile scritto in codice binario.
Il linguaggio **Assembly** usa un “Compiler” speciale chiamato **Assembler,** che genera un file binario detto “oggetto” che però non è completo. Per essere completo, esso dovrebbe essere anche “linkato”.
Che significa linkare un programma? Significa aggiungere al file oggetto i riferimenti a tutte quelle librerie che usa esternamente. Per esempio quando in *C* diamo il seguente comando:

```
$gcc -o prog prog.c
```

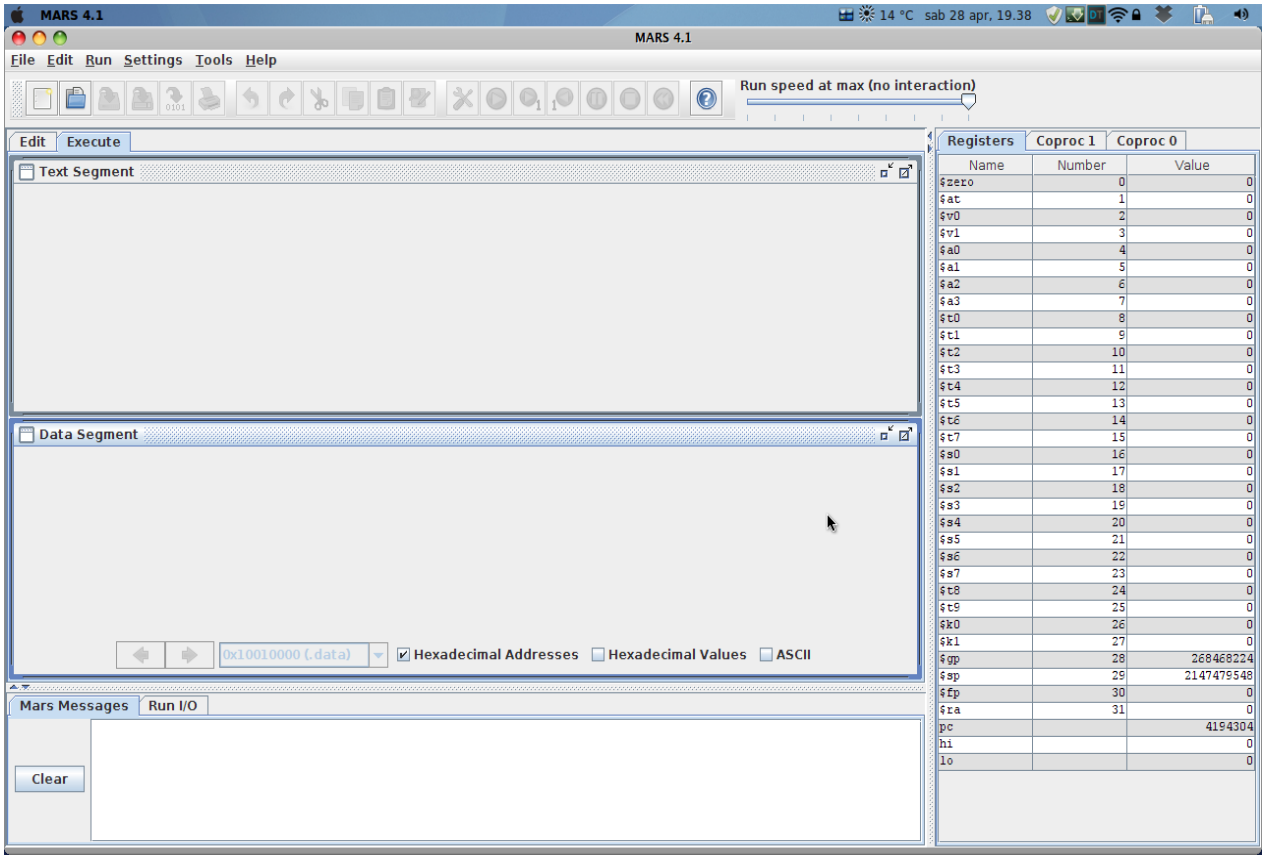
il compilatore *gcc* “linka” automaticamente al file *prog.o* (che verrà generato durante l'esecuzione di *prog*) con i link alle librerie definite in *prog.c*.
Immaginiamo quindi che in *prog.c* ci sia una libreria “*lib.h*” che vogliamo linkarla noi manualmente. Dovremmo fare così:

```
$ gcc -c lib.o lib.c #genero il file .o della libreria
$ gcc -c prog.o prog.c #genero il file.o del programma principale
$ gcc -o exe prog.o lib.o #genero l'eseguibile linkando lib con prog
```

in questo modo avremmo linkato i due file oggetto ottenendo l'eseguibile finale.

3 – ARCHITETTURA MIPS

In questa dispensa useremo l'**Assembly MIPS**, e l'emulatore **MARS** (vedi http://www.astromoodle.altervista.org/download/MIPSLinux.txt per sapere come installarlo).
Quindi, per prima cosa vediamo com'è definita l'architettura MIPS: essa è composta da *32 registri* da *32 byte* ciascuno utilizzabili tutti nello stesso modo, ma per convenzione alcuni vengono usati per certe cose, altri per altre. Aprendo l'emulatore MARS vedremo la seguente schermata cliccando sulla linguetta *Execute*:



A destra vediamo i 32 registri, mentre al centro verranno mostrati i *Text* e *Data Segment* (che fra poco vedremo). Nella linguetta *Edit* ci è possibile scrivere il nostro codice Assembly, che dopo aver salvato potremmo eseguire e vederne il risultato.

Per prima cosa quindi vediamo i nomi e l'utilizzo dei 32 registri del MIPS:

$zero	Contiene sempre zero. Scritto anche come $0
$at	Usato solo dall'Assembler. Scritto anche come $1
$v0	Usato per invocare funzioni “speciali” dell'Assembler. Scritto anche come $2
$v1	Usato per salvare il risultato delle funzioni. Scritto anche come $3
$a0	Usato come parametro per le funzioni. Scritto anche come $4
$a1	Usato come parametro per le funzioni. Scritto anche come $5
$a2	Usato come parametro per le funzioni. Scritto anche come $6
$a3	Usato come parametro per le funzioni. Scritto anche come $7
$t0	Registro temporaneo. Scritto anche come $8
$t1	Registro temporaneo. Scritto anche come $9
$t2	Registro temporaneo. Scritto anche come $10
$t3	Registro temporaneo. Scritto anche come $11
$t4	Registro temporaneo. Scritto anche come $12
$t5	Registro temporaneo. Scritto anche come $13
$t6	Registro temporaneo. Scritto anche come $14
$t7	Registro temporaneo. Scritto anche come $15
$s0	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $16
$s1	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $17
$s2	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $18
$s3	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $19
$s4	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $20
$s5	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $21
$s6	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $22
$s7	Registro salvato tra un'invocazione di una funzione e l'altra. Scritto anche come $23
$t8	Registro temporaneo. Scritto anche come $24
$t9	Registro temporaneo. Scritto anche come $25
$k0	Registro usato esclusivamente durante le chiamate al SO. Scritto anche come $26
$k1	Registro usato esclusivamente durante le chiamate al SO. Scritto anche come $27
$gp	Global Pointer. Scritto anche come $28
$sp	Stack Pointer. Scritto anche come $29
$fp	Frame Pointer. Scritto anche come $30
$ra	Return Address. Scritto anche come $31

In aggiunta a questi 32 registri ci sono anche i seguenti registri, che però non sono accessibili:

pc	Program Counter
hi	Registro per il risultato di moltiplicazioni e divisioni
lo	Registro per il risultato di moltiplicazioni e divisioni

4 – ISTRUZIONI E DIRETTIVE

L'**Assembly MIPS** è composto di varie istruzioni e direttive, di cui brevemente vedremo solo le più importanti. Prima di vederle però va sottolineato un particolare importante nell'Assembly MIPS: esso divide le istruzioni in due tipi: istruzioni normali e pseudo-istruzioni. Le **pseudo-istruzioni** non sono altro che istruzioni che al momento della traduzione vengono tradotte in più istruzioni normali.
Possiamo a nostra volta dividere le istruzioni complessivamente in 3 tipi:

- **direttive:** non sono istruzioni vere e proprie, ma solo linee guide per la traduzione
- **dichiarative**: sono le etichette, ovvero istruzioni che "dichiarano" qualcosa
- **esecutive**: le istruzioni vere e proprie

Vediamo quindi un breve elenco delle direttive più usate:

- **.asciiz**: dichiara una stringa terminata dal carattere nullo '\0'
- **.byte:** dichiara una variabile di 8 bit (1 byte)
- .**half:** dichiara una variabile di 16 bit (2 byte)
- **.word:** dichiara una variabile di 32 bit (4 byte)
- **.float:** dichiara una variabile con virgola mobile a singola precisione (4 byte)
- **.space**: prenota uno spazio di n (dove n è il valore scritto dopo .space)
- **.data**: definisce il segmento dati (Data-Segment)
- **.text:** definisce il segmento testo (Text-Segment)
- **.globl:** definisce un'etichetta globale

La dichiarazione di una variabile avviene in questo modo: `etichetta: .tipo valore`

```
1  .data
2  intero:     .word    100
3  stringa:    .asciiz  "sono una stringa"
4  riservato:  .space   4 #riservo 4 byte per la variabile riservato
5  .text
6  .globl main #etichetta gloable
```

Passiamo ora ad una serie di istruzioni generali (*P = Pseudo-Istruzione*):

Istruzioni di uso generale			
Nome Istruzione	**Esempio**	**P**	**Descrizione**
`nop`	`nop`		Non fa nulla
`move`	`move $s3, $s4`	Si	Sposta il valore del 2° registro nel 1°
`syscall`	`syscall`		Chiamata a sistema (richiama una funzione del S.O.)
`li`	`li $s3, 1`	Si	Carica una costante
`la`	`la $s3, var`	Si	Carica un indirizzo
`sw`	`sw $s3, 0($t0)`		Salva un valore di 4 Byte del 1° registro all'indirizzo dato
`sb`	`sb $s3, 0($t0)`		Salva un valore di 1 Byte del 1° registro all'indirizzo dato
`lbu`	`lb $s2, 0($t0)`		Carica nel byte più basso del 1° reg. il valore dall'indirizzo
`lw`	`lw $s2, 0($t0)`		Carica nel 1° registro il valore da 4 byte dall'indirizzo dato
`lui`	`lui $s2, 0($t0)`		Carica nei 16 bit più alti il val dell'indirizzo, il resto è a 0
`mfhi`	`mfhi $s2`		Sposta il valore del registro hi
`mflo`	`mflo $s3`		Sposta il valore dal registro lo

Istruzioni aritmetiche			
add	`add $s3, $s2, $s4`		Somma il 2° col 3° registro e mette il risultato nel 1° registro
addi	`addi $s3, $s2, 0`		Somma il 2° registro con una costante e mette il risultato nel 1°
sub	`sub $s3, $s2, $s1`		Sottrae il 2° registro col 3° e mette il risultato nel 1°
mult	`mult $s2, $s1`		Moltiplica i 2 reg. e mette il risultato in hi:lo (vedi più avanti)
div	`div $s2, $s0`		Divide i 2 registri e mette i risultati in hi e lo (vedi più avanti)

Istruzioni logiche			
not	`not $s0, $s1`	Si	Nega il 2° registro e mette il risultato nel 1°
or	`or $s0, $s1, $s2`		Or bit-a-bit fra il 2° e il 3° registro. Risultato nel 1° registro.
ori	`ori $s0, $s1, 0`		Or bit-a-bit fra il 2° registro e una costante. Risultato nel 1° reg
and	`and $s0, $s1, $s2`		And bit-a-bit fra il 2° e il 3° registro. Risultato nel 1° registro
andi	`andi $s0, $s1, 1`		And bit-a-bit fra il 2° reg. e una costante. Risultato nel 1° reg.
xor	`xor $s0, $s1, $s2`		Or esclusivo fra il 2° e il 3° registro. Risultato nel 1° registro
xori	`xori $s0, $s1, 10`		Or esclusivo fra il 2° reg. e una costante. Risultato nel 1° reg.
sll	`sll $s0, $s1, 4`		Shift a sx del 2° regi. del valore della costante. Ris. nel 1° reg.
srl	`srl $s0, $s1, 4`		Shift a dx del 2° reg. del valore della costante. Ris. nel 1° reg.

Istruzioni di salto e di controllo (vedi più avanti)			
slt	`slt $s0, $s1, $s2`		Setta a 1 il 1° reg se il 2° reg è minore del 3°
seq	`seq $s0, $s1, $s2`	Si	Setta a 1 il 1° reg se il 2° reg è uguale al 3°
sge	`sge $s0, $s1, $s2`	Si	Setta a 1 il 1° reg se il 2° reg è maggiore o uguale al 3°
sgt	`sgt $s0, $s1, $s2`	Si	Setta a 1 il 1° reg se il 2° reg è maggiore del 3°
sle	`sle $s0, $s1, $s2`	Si	Setta a 1 il 1° reg se il 2° reg è minore o uguale al 3°
sne	`sne $s0, $s1, $s2`	Si	Setta a 1 il 1° reg se il 2° reg è diverso dal 3°
j	`j etichetta`		Salto incondizionato all'etichetta specificata
beq	`beq $s0, $s1, label`		Salta all'etichetta se il 1° registro è uguale al 2°
bne	`bne $s0, $s1, label`		Salta all'etichetta se il 1° registro è diverso dal 2°
beqz	`beqz $s0, label`	Si	Salta all'etichetta se il registro è uguale a 0
bge	`bge $s1, $s2, label`	Si	Salta all'etichetta se il 1° registro è maggiore o uguale al 2°
bgt	`bgt $s1, $s2, label`	Si	Salta all'etichetta se il 1° registro è maggiore del 2°
ble	`ble $s1, $s2, label`	Si	Salta all'etichetta se il 1° registro è minore o uguale al 2°
blt	`blt $s1, $s2, label`	Si	Salta all'etichetta se il 1° registro è minore del 2°
bnez	`bnez $s0, label`	Si	Salta all'etichetta se il 1° registro è diverso da 0
Istruzioni di salto usate nelle funzioni			
jal	`jal etichetta`		Salta all'etichetta (chiamata di funzione)
jr	`jr $ra`		Salta all'indirizzo contenuto nel registro

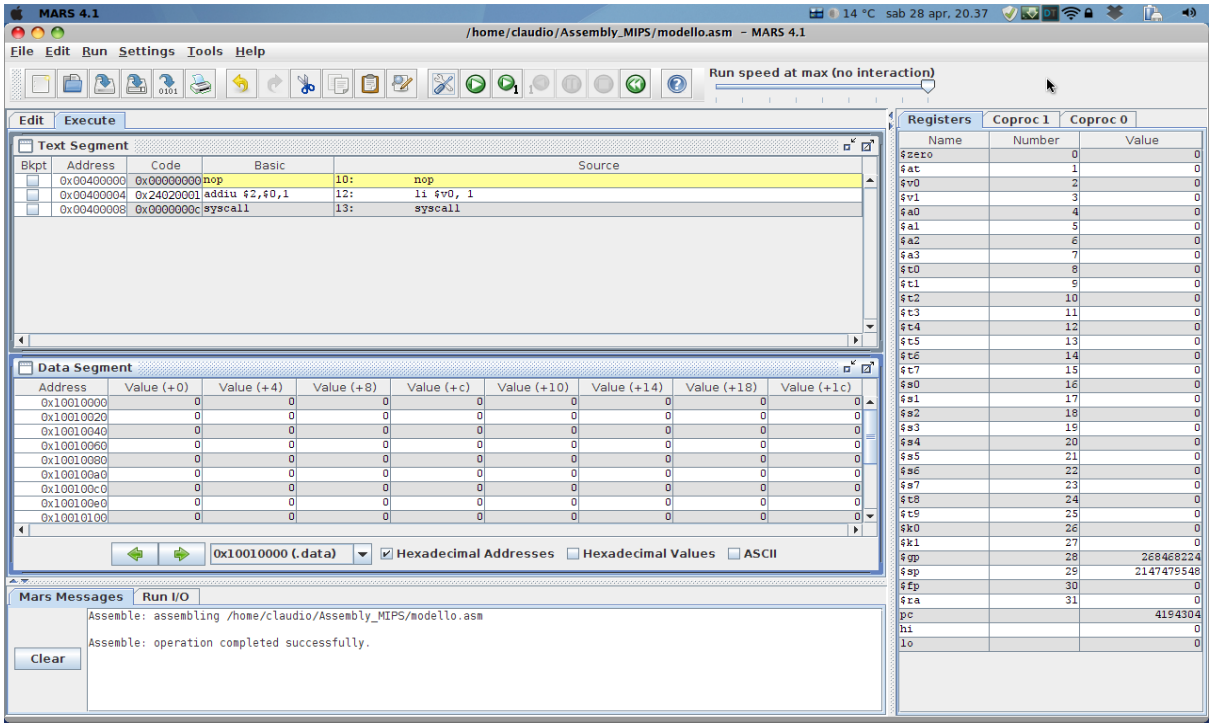
5 – STRUTTURA DEL SORGENTE

Un file sorgente in Assembly MIPS è codificato in questa maniera:

```
1   # Programma: modello.asm
2   #       Autore: Borsato Claudio
3   #       Data: 28/6/2012
4   #       Descrizione: Modello di Sorgente in Assembly MIPS
5
6   .data
7   .text
8   .globl main
9   main:
10          nop
11  fine:
12          li $v0, 1
13          syscall
14
```

Le prime righe sono dei commenti, quindi esse non sono essenziali. Concentriamoci invece sulla *riga 7*: essa denota l'inizio del segmento testo, il quale segnala all'assemblatore l'inizio delle istruzioni. La direttiva **.globl** dichiara l'etichetta *main* come globale, ovvero visualizzabile in tutto il contesto del programma.
Dopo di che viene effettivamente usata l'etichetta *main* (che denota il corpo principale del programma).
L'istruzione **nop** non fa nulla, e se eseguissimo questo programma con MARS non succederebbe nulla, perché questo programma non fa proprio niente!
Infine, le ultime due istruzioni: essei sono una chiamata al S.O. al quale il programma Assembly segnala il termine della propria esecuzione.

Per eseguire il programma col nostro simulatore ci basterà cliccare sull'icona con il cacciavite e la chiave inglese, e verremmo portati nella linguetta *Execute* che apparirà così:



Nel *text-segment* vedremmo le varie istruzioni, i loro indirizzo ed eventuali *breakpoint* (i breakpoint servono se vogliamo eseguire il programma tutto in un colpo, esso si fermerà quando troverà un breakpoint), il codice in esadecimale dell'istruzione, l'istruzione normale e l'istruzione com'è stata scritta nel sorgente. Come possiamo vedere l'istruzione **li** è stata tradotta con una **addiu** (*add immediate unsigned*).
Nel data segment invece vedremo i valori di eventuali variabili dichiarate, e nella combo-box potremmo scegliere se vedere il segmento **.data** (variabili globali) o dello **stack** (variabili locali).
Per eseguire il programma ci basterà cliccare sull'icona *PLAY* (cerchio verde con triangolo bianco) o se vogliamo eseguire istruzione per istruzione sull'icona *RUN-STEP* (cerchio verde con triangolo bianco con 1 sull'angolo).

Vediamo ora però le istruzioni **mult** e **div**, che per se, sono un attimo differenti dalle altre: l'istruzione **mult** moltiplica 2 numeri ed il risultato viene diviso in due parti: la parte più alta (16 bit più a sinistra) vengono posti nell'indirizzo **hi,** mentre i restanti nell'indirizzo **lo**.
La divisione invece mette il risultato in **lo**, e il resto della divisione in **hi**.

Questo semplice programma mostrerà l'esecuzione delle varie operazioni aritmetiche prima elencate:

```
1  # Programma: operazioni.asm
2  #       Autore: Borsato Claudio
3  #       Data: 28/6/2012
4  #       Descrizione: Operazioni Aritmetiche
5
6  .data
7  .text
8  .globl main
9  main:
10         li $s1, 2
11         li $s2, 3
12         add $t0, $s1, $s2  # $t0 = $s1 + $s2
13         addi $t1, $s1, 2   # $t1 = $s1 + 2
14         sub $t2, $s1, $s2  # $t2 = $s1 - $s2
15         mult $s1, $s2
16         mflo $t3           # $t3 = $s1 * $s2
17         div $s1, $s2
18         mfhi $t5           # hi = $s1 / $s2
19         mflo $t4           # lo = $s1 % s2
20
21 fine:
22         li $v0, 1
23         syscall
24
```

Se guardiamo a destra vedremo che i registri **$t0, $t1, $t2, $t3, $t4** e **$t5** conterranno rispettivamente 5, 4, -1, 6, 0 e 2.

Ora vediamo un esempio per lo spostamento dei dati da delle locazioni di memoria, il quale prende il valore di *dato1* e lo mette in *dato2*:

```
# Programma: operazioni2.asm
#       Autore: Borsato Claudio
#       Data: 28/6/2012
#       Descrizione: Operazioni di spostamento

.data
dato1:  .word 10
dato2:  .word 0
.text
.globl main
main:
        la $t0, dato1 # $t0 contiene l'indirizzo di dato1
        la $t1, dato2 # $t1 contiene l'indirizzo di dato2
        lw $s0, 0($t0) # carico il valore di dato1 in s0
        sw $s0, 0($t1) # salvo il valore in s0 in dato2
                       # il cui indirizzo è contenuto in s0

fine:
        li $v0, 1
        syscall

```

Andando sulla linguetta *Execute* vedrete sicuramente che la prima e la seconda casella del *data-segment* contengono tutte e due 10, come detto.

Infine vediamo le operazioni logiche:

```
# Programma: operazioni3.asm
#       Autore: Borsato Claudio
#       Data: 28/6/2012
#       Descrizione: Operazioni logiche

.data
var1: .byte 1
var2: .byte 0
.text
.globl main
main:
        la $t0, var1 # $t0 contiene l'indirizzo di var1
        la $t1, var2 # $t1 contiene l'indirizzo di var2
        lb $s0, 0($t0) # carico il valore di var1 in s0
        lb $s1, 0($t1) # carico il valore di var2 in s1
        and $t3, $s0, $s1 # $t3 = $s0 & $s1
        or $t4, $s0, $s1 # $t4 = $s0 | $s1
        not $t5, $s1 # !$s0
        xor $t6, $s0, $s1 # $t6 = $s0 xor $s1

fine:
        li $v0, 1
        syscall

```

Come potete vedere l'esecuzione di questo programma darà vari risultati. Interessante però è il caso della **not.** Se visualizzerete i valori in esadecimale invece che in decimale vedrete che il risultato della **not** è *0xfff..f,* ovvero -1. *0xfff..f* in binario è *111..1* che è la negazione di *000..0* e quindi è giusto.

6 – SALTI CONDIZIONATI E INCONDIZIONATI

In tutti i programmi che si rispettino c'è un controllo di flusso fatto da istruzioni quali *IF ELSE, WHILE, REPEAT UNTILE, FOR* e tanti altri. In Assembly purtroppo questi tipi di istruzioni non sono presenti, ma ci è possibile "ricrearle" attraverso le istruzioni Assembly viste prima.
L'Assembly MIPS ci da 2 tipi di istruzioni di salto, i **set** e i **branch**.
Le istruzioni set hanno questa forma:

```
setx $r1, $r2, $r3 # r1 = 1 se $r2 x $r3 è vera
```

Ovvero il registro **$r1** viene settato a **1** se la condizione **x** risulta vera fra **$r2 e $r3**
L'altra istruzione è l'istruzione **branch,** la quale ha questa forma:

```
bx $r1, $r2, label # salta a label se $r1 x $r2 è vera
```

Come prima, **bx** salta all'etichetta **label** se la condizione **x** fra **$r1 e $r2** risulta vera. Bado alle ciance, e vediamo come tradurre un *IF ELSE.*
La forma dell'*IF ELSE* è la seguente:

```
IF:
        confronto
        salto condizionato a ALTRIMENTI
        istruzioni di uno dei 2 rami
        salto incondizionato a FINESE
ALTRIMENTI:
        istruzioni dell'altro ramo
FINESE:
```

Detto questo vediamo il nostro programma Assembly, il quale dati 2 numeri, se uguali, mette il registro **$s2** a **1,** altrimenti a **0**.

```
1  # Programma: if.asm
2  #       Autore: Borsato Claudio
3  #       Data: 28/6/2012
4  #       Descrizione: costrutto if
5
6  .data
7  var1: .word 1
8  var2: .word 0
9  .text
10 .globl main
11 main:
12         la $t0, var1
13         la $t1, var2
14         lw $s0, 0($t0) # carico il valore di var1 in s0
15         lw $s1, 0($t1) # carico il valore di var2 in s1
16
17 #INIZIOSE
18 IF:
19         bne $s0, $s1, ALTRIMENTI  #confronto #salto condizionato
20         li $s2, 1                 #istruzioni del ramo vero
21         j FINESE                  #salto incondizionato
22 ALTRIMENTI:
23         li $s2, 0                 #istruzioni del ramo falso
24 FINESE:
25
26 fine:
27         li $v0, 1
28         syscall
29
```

Come visto nell'esempio abbiamo decodificato l'*IF* nelle sue varie fasi. L'equivalente codice in *C* sarebbe stato:

```
IF(var1==var2){
	$s2=1;
}
ELSE {
	$s2=0;
}
```

Ma ora vediamo i cilci.
Cominciamo con un ciclo semplice, il *WHILE*.

```
INIZIOCICLO:
	confronto
	salto condizionato a FINECICLO
	istruzioni del corpo del ciclo
	salto incondizionato a INIZIOCICLO
FINECICLO:
```

Similmente all'*IF* prima faremo il controllo, e se la condizione risulta falsa salteremo, altrimenti continueremo nell'esecuzione del ciclo.

Anche qui, ora faremmo un semplice programma che incrementa un numero finché è minore di un numero dato.

```
1  # Programma: while.asm
2  #       Autore: Borsato Claudio
3  #       Data: 28/6/2012
4  #       Descrizione: costrutto while
5
6  .data
7  var: .word 5
8  .text
9  .globl main
10 main:
11         la $t0, var
12         lw $s0, 0($t0) # carico il valore di var in s0
13         addi $s1, $zero, 0 #inizializzo $s1 a 0
14
15 INIZIOCICLO:
16         slt $t1, $s0, $s1       #confronto
17         bne $t1, 1, FINECICLO   #salto condizionato
18         addi $s1, $s1, 1        #istruzioni del ciclo
19         j INIZIOCICLO           #salto incondizionato
20 FINECICLO:
21
22 fine:
23         li $v0, 1
24         syscall
25
```

Di nuovo abbiamo tradotto le varie parti del ciclo. In *C* il ciclo sarebbe stato definito così:

```
while($s1<var){
	$s1=$s1+1;
}
```

Il ciclo *FOR* è equivalente al ciclo *WHILE* appena visto quindi non lo vedremo. Quindi l'unico ciclo mancante è il *DO...WHILE o REPEAT...UNTIL* che è composto così:

```
INIZIOCICLO:
      istruzioni del corpo del ciclo
      confronto
      salto condizionato a INIZIOCICLO
```

Similmente faremmo come prima, ma in questo caso la somma avverrà sempre e comunque almeno una volta (ovvero prima avverrà l'incremento, poi il controllo).

```
INIZIOCICLO:
        addi $s1, $s1, 1          #istruzioni del ciclo
        slt $t1, $s0, $s1         #confronto
        bne $t1, 1, INIZIOCICLO   #salto incondizionato
#FINECICLO:
```

Mantenendo il corpo del programma di prima, vediamo che anche il *DO...WHILE* è stato tradotto con successo. Tradotto in *C:*

```
do {
      $s1=$s1+1;
} while ($s1<var);
```

7 – INPUT / OUTPUT

Le operazioni di **I/O** in Assembly MIPS sono assai semplici, visto che si tratta solo di un "setta-valore" e richiama la funzione di sistema collegata. Nel seguente programma vedremo un semplice programma richiede in input un numero, lo moltiplica per 5, e ne stampa il risultato.

```
1  # Programma: io.asm
2  #       Autore: Borsato Claudio
3  #       Data: 28/6/2012
4  #       Descrizione: operazioni di I/O
5
6  .data
7  mex: .asciiz "Inserisci un numero: "
8  .text
9  .globl main
10 main:
11         addi $v0, $zero, 4
12         la $a0, mex
13         syscall          #stampo mex
14         addi $v0, $zero, 5
15         syscall          #leggo un intero da input
16         li $s0, 5
17         mult $v0, $s0    #moltiplico
18         mflo $s1
19         addi $v0, $zero, 1
20         add $a0, $zero, $s1
21         syscall         #stampo il risultato
22
23
24 fine:
25         li $v0, 1
26         syscall
27
```

Seguendo i seguenti codici si possono conoscere le operazioni di **I/O**:

Valore in $v0	***Valore in $a0***	***Valore in $a1***	***Risultato in $v0***	***Effetto:***
1	Numero	N/A	N/A	*Stampa il valore in $a0*
4	Indirizzo	N/A	N/A	*Stampa la stringa il cui indirizzo è in $a0*
5	N/A	N/A	Numero	*Restituisce il valore dato in input*
8	Indirizzo	Lunghezza	N/A	*Salva $a1 caratteri all'indirizzo in $a0 la stringa data in input*

8 – ARRAY

Un'importante struttura in *C* è l'**array**, ovvero una lista di lunghezza fissa di elementi. In Assembly è possibile gestire gli array un po' come in *C.* Prendiamo per esempio il seguente pezzo di codice in *C*:

```
for(i=0; i<n; i++){
      r=r+v[i];
}
```

Come codificarlo in Assembly MIPS? Detto fatto:

```
        #$s0 sarà i
        #$t0 conterrà l'indirizzo di del vettore v
        #$s1 sarà n
        #$v0 sarà r
        addi $s0, $zero, 0   #i=0
FOR:
        slt  $t1, $s0, $s1
        bne  $t1, 1, FOR_END #i<n
        li   $t2, 4
        mult $t2, $s0
        mflo $t3
        add  $t4, $t0, $t3
        lw   $t5, 0($t4)     #v[i]
        add  $v0, $v0, $t5   #r = r + v[i]
        addi $s0, $s0, 1     #i++
FOR_END:
```

Vediamo in dettaglio questo codice passo per passo:
Dopo il confronto carico in **$t2 4**, questo perché il vettore è di interi, quindi avrà sicuramente una dimensione che è multipla di **4**.
Moltiplico **$t2** con **$s0** per conoscere di quanto devo saltare per trovare la posizione del vettore..
V[0] => &V+0; V[1] => &V+4; V[2] => &V+8; etc..
Ottenuto il valore da sommare a l'indirizzo di v, lo faccio, e sposto il valore contenuto in quello specifico valore nel registro **$t5;**
In **$t5** ora ho l'indirizzo del valore a cui voglio accedere. Mi basterà fare una **lw** per tirare fuori il valore e sommarlo ad **r**.
Gli array in Assembly MIPS si dichiarano così:

```
vet: .word 1,2,-5,80 #vettore di interi
chr: .byte 'a','b','c' #vettore di char
str: .asciiz "hello"  #stringa terminata da '\0'
```

9 – FUNZIONI E PUNTATORI (STACK)

La traduzione di una funzione in Assembly avviene in 8 fasi:
1- *salvataggio dell'indirizzo di ritorno prima di chiamare una funzione*
2- *passaggio dei parametri attraverso* ***$a0,...,$a3***
3- *chiamata alla funzione*
4- *salvataggio dei registri* ***$s0,...,$s7,$a0,..,$a3*** *(se modificati nella funzione chiamata)*
5- *salvataggio del valore di ritorno in* ***$v0*** *(se c'è)*
6- *ricaricare i registri* ***$s0,...,$s7,$a0,...,$a3*** *(se modificati nella funzione chiamata)*
7- *salto all'indirizzo di ritorno*
8- *ricarico il vecchio indirizzo di ritorno*

Alcuni di questi passi (i passaggi 2, 4, 5, 6) non sono obbligatori, ma è convenzione per i programmatori MIPS di fare questi passaggi. Quest'operazione di salvataggio è detta **callee-save,** ed i registri salvati sono detti **callee-save registers**.
Però per salvare questi registri abbiamo bisogno di un'area di memoria abbastanza capiente per tenerli. Quest'area di memoria è lo **stack**.
L'indirizzo di ritorno delle funzioni viene sempre salvato in **$ra**.

```
1  # Programma: fun.asm
2  #       Autore: Borsato Claudio
3  #       Data: 28/6/2012
4  #       Descrizione: chiamata di funzione
5  .data
6  .text
7  .globl main
8  main:
9          li $s0, 5 # primo valore
10         li $s1, 6 # secondo valore
11         #1) salvo l'indirizzo di ritorno del main
12         addi $sp, $sp, -4 #faccio spazio nello stack per $ra
13         sw $ra 0($sp)     #salvo $ra
14         #2) passaggio dei parametri attravero $a0, $a1
15         move $a0, $s0
16         move $a1, $s1
17         #3) richiamo la funzione sum
18         jal sum
19         #8) ricarico il vecchio valore di $ra
20         lw $ra, 0($sp)
21         addi $sp, $sp, 4
22
23 ######### function sum ###########
24 sum:
25         #4) salvo i vecchi valori di $s0, $s1
26         addi $sp, $sp, -8
27         sw $s0 0($sp)
28         sw $s1 4($sp)
29         move $s0, $a1 #carico i valori
30         move $s1, $a0
31         addi $s2, $s0, $s1 #eseguo la somma
32         #5) salvo il valore di ritorno in $v0
33         move $v0, $s2
34         #6) ricarico i valori di $s0, $s1
35         lw $s0, 0($sp)
36         lw $s1, 4($sp)
37         addi $sp, $sp, 8 #svuoto lo stack
38         #7) salto all'indirizzo di ritorno
39         jr $ra
```

Il programma di cui sopra semplicemente passa 2 valori alla funzione **sum**, li somma, e ne

ritorna il risultato. Per comprendere meglio vediamo questo semplice schema:

Registri	***Prima di 3)***	***Prima di 5)***	***Prima di 7)***	***Dopo 8)***
$s0	5	6	5	5
$s1	6	5	6	6
$s2		11	11	11
$a0	5	5	5	5
$a1	6	6	6	6
$v0			11	11
$ra	"$ra di main"	"$ra di sum"	"$ra di sum"	"$ ra di main"
Stack				
$sp - 0	"$ra di main"	"$ra di main"	"$ra di main"	
$sp - 4		5		
$sp - 8		6		
$sp - 12				

Vediamo quindi la prima fase *(prima di 3)*:
In **$s0, $s1** ci sono i 2 nostri valori da sommare, che vengono passati come parametri attraverso **$a0** e **$a1.** Dopo di ciò viene salvato il valore di **$ra** nello stack.
A questo punto viene chiamata la funzione **sum** (seconda fase, *prima di 5*), e dopo aver eseguito i vari calcoli ci troviamo al punto di prepararci al ritornare al **main**. Vediamo quindi lo stato dei vari registri: **$s0** e **$s1** hanno i nuovi valori ottenuti attraverso il caricamento da **$a0** e **$a1** (**righe 29 e 30)**, e **$s2** riceve il risultato di tale somma.
La terza frase *(prima di 7)*, segue il ritorno dei valori e il ripristino di quelli vecchi, che come vedete avviene, a partire dal posizionare il risultato da ritornare in **$v0 (riga 33)**; dopo ciò ricarico i vecchi valori di **$s0** e **$s1** e ritorno al **main** attraverso il valore di **$ra**.
Nell'ultima fase *(dopo di 8)* ricarico il vecchio valore di **$ra,** ovvero quello di **main**.
Questo come vedete è un esempio facile facile, ma proviamo a complicare un po' le cose, facciamo di avere un programma che non solo ha una funzione, ma bensì 3 funzioni, di cui 2 vengono richiamate da altre funzioni. Per esempio immaginiamoci un programma che in *C* sarebbe scritto così:

Main	***funz1***	***Funz2***	***funz3***

`void main(){`
`  funz1(10);`
`}`
	`void funz1(int a){`
	`  a++;`
	`  int b=funz2();`
	`}`
		`int funz2(){`
		`  funz3();`
		`  return 5;`
		`}`
			`void funz3(){`
			`}`

In Assembly MIPS come lo scriveremo?
Come vediamo **funz1** ha una variabile locale, ma fino ad ora noi abbiamo visto solo che in Assembly si possono definire variabili globali attraverso le etichette.. ma come fare per dichiarare delle variabili locali? Come accade in *C,* useremo lo *stack*.
Lo *stack* verrà quindi usato per salvare i vari dati come abbiamo fatto prima per i c*allee-save registers*, solo che ora lo useremo anche per le variabili locali. Prima però vediamo il codice delle 4 funzioni:

```
main:
        li $s0, 10
        #salvo $ra di main
        addi $sp, $sp, -4
        sw $ra, 0($sp)
        #metto in $a0 il parametro da passare
        move $a0, $s0
        #(1)
        jal funz1
        #ricarico $ra di main
        sw $ra 0($sp)
        addi $sp, $sp, 4
        #(9)
        jr $ra

funz1:
        #salvo il valore di $s0
        addi $sp, $sp, -4
        sw $s0, 0($sp)
        #incremento a salvandolo in un registro
        addi $s0, $a0, 1
        #faccio spazio nello stack per la variabile b
        addi $sp, $sp, -4
        #mi salvo l'indirizzo di b in $s2
        move $s2, $sp
        #salvo $ra di funz1 nello stack
        #questo perché b riceverà il valore di ritorno
        #di funz2 che è una funzione, quindi devo fare
        #una chiamata a funzione
        addi $sp, $sp, -4
        sw $ra, 0($sp)
        #(2)
        jal funz2
        #ricarico $ra di funz1
        lw $ra 0($sp)
        addi $sp, $sp, 4
        #(7)
        #salvo il valore di ritorni in b
        sw $v0 0($s2)
        #(7.5)
        #sto uscendo dalla funzione, b non mi serve
        #più quindi la rimuovo
        addi $sp, $sp, 4
        #ricarico il vecchio valore di $s0
        lw $s0 0($sp)
        addi $sp, $sp, 4
        #(8)
        jr $ra
funz2:
        #funz2 come prima istruzione ha una chiamata a
        #funz3, quindi salvo immediatamente $ra di funz2
        addi $sp, $sp, -4
        sw $ra, 0($sp)
        #(3)
        jal funz3
        #ricarico il $ra di funz2
        lw $ra 0($sp)
        addi $sp, $sp, 4
        #(5)
        #funz2 modifica $s2, quindi devo salvare $s2
        addi $sp, $sp, -4
        sw $s2, 0($sp)
        li $s2, 5
        #ritorno il valore di $s2
        move $v0, $s2
        #(5.5)
        #ricarico il vecchio valore di $s2
        lw $s2, 0($sp)
        addi $sp, $sp, 4
        #(6)
        jr $ra

funz3:
        #funz3 non fa nulla, non modifica alcun registro,
        #quindi non ha bisogno ne di salvare $ra, ne alcun
        #altro registro.
        #(4)
        jr $ra
```

Nei commenti vedete dei numeri. Ecco, quelli sono i vari punti in cui andremmo a ispezionare i registri e lo stack con i loro vari dati. Cominciamo con i punti *1) 2) 3) 4)*

Registri	***1)***	***2)***	***3)***	***4)***
$s0	10	11	11	11
$s2		"indirizzo di b"	"indirizzo di b"	"indirizzo di b"
$a0	10	10	10	10
$v0				
$ra	"$ra di main"	"$ra di funz1"	"$ra di funz2"	"$ra di funz3"
Stack				
$sp - 0	"$ra di main"	"$ra di main"	"$ra di main"	"$ra di main"
$sp - 4		10	10	10
$sp - 8		"var b"	"var b"	"var b"
$sp - 12		"$ra di funz1"	"$ra di funz1"	$ra di funz1"
$sp - 16			"$ra di funz2"	"$ra di funz2"
$sp - 20				

Soffermiamoci intanto in questi 4 punti. Nel punto *1)* abbiamo il caricamento del valore 10 da passare da **main** a **funz1**. Nell'avviare la chiamata **main** salva il proprio **$ra** e chiama **funz1**.
Entrando in **funz2** vediamo il salvataggio di **$s0** (che in **funz1** verrà modificato) e la creazione di uno spazio libero per la variabile **b**. Dopo ciò, come visto nel codice *C,* **b = funz2()**, quindi arriviamo nel punto *3)* dove abbiamo una chiamata a **funz2** (che come prima facciamo prima salvando il **$ra** della funzione chiamante). Il punto *4) è* invariato (a eccezione di **$ra** che ora contiene l'indirizzo della propria chiamata) dal punto *3*) perché la **funz3()** non fa nulla, quindi ritorna immediatamente in **funz2** senza attuare alcuna modifica. Ora vedremo cosa accadrà nei punti *5) 5.5) 6) e 7)*

Registri	**5)**	**5.5)**	**6)**	**7)**
$s0	11	11	11	11
$s2	"indirizzo di b"	5	"indirizzo di b"	"indirizzo di b"
$a0	10	10	10	10
$v0		5	5	5
$ra	"$ra di funz2"	"$ra di funz2"	"$ra di funz2"	"$ra di funz1"
Stack				
$sp - 0	"$ra di main"	"$ra di main"	"$ra di main"	"$ra di main"
$sp - 4	10	10	10	10
$sp - 8	"var b"	"var b"	"var b"	"var b"
$sp - 12	"$ra di funz1"	"$ra di funz1"	"$ra di funz1"	
$sp - 16		"indirizzo di b"		
$sp - 20				

Nel punto *5)* siamo tornanti in **funz2** e quindi abbiamo ricaricato il **$ra** di **funz2,** dopo di ciò nel punto *5.5)* abbiamo prima salvato il registro **$s2,** poi modificato per caricarci il valore 5, che a sua volta è passato a **$v0** come valore di ritorno della funzione **funz2()** (vedi **b=funz2()**). Fatto ciò, ci apprestiamo a ritornare nella funzione **funz1()**, quindi dobbiamo ricaricare il vecchio valore di **$s2**, ovvero l'indirizzo di **b.** Nel punto *7*) siamo di nuovo in **funz1,** quindi ricarichiamo il **$ra** di **funz1.**
A questo punto vedremo l'ultima fase del programma, ovvero i punti *7.5) 8) e 9)*

Registri	**7.5)**	**8)**	**9)**	
$s0	`11`	`10`	`10`	
$s2	`"indirizzo di b"`	`"indirizzo di b"`	`"indirizzo di b"`	
$a0	`10`	`10`	`10`	
$v0	`5`	`5`	`5`	
$ra	`"$ra di funz1"`	`"$ra di funz1"`	`"$ra di main"`	
Stack				
$sp - 0	`"$ra di main"`	`"$ra di main"`		
$sp - 4	`10`			
$sp - 8	`5 (var b)`			
$sp - 12				
$sp - 16				
$sp - 20				

In *7.5)* vediamo che b ha ricevuto il valore di 5, che era il valore di ritorno di **funz2()**. A questo punto la funzione ha cessato di far quel che deve fare, e quindi esce per tornare in **main**. Per fare ciò deve fare 2 cose: rimuovere le variabili locali e ricaricare i vecchi valori di eventuali registri modificati. Come possiamo vedere in *8)* la variabile b è sparita e **$s0** ha ricevuto il suo vecchio valore.
Il *9)* ci mostra lo stato finale quando siamo tornati nel **main**. Molti registri hanno vari valori sporchi, ottenuti nel corso dell'esecuzione del programma, ma tutti i valori che inizialmente erano stati settati al lancio di **main** sono ancora lì: **$s0** ha il suo valore 10 e **$ra** è quello di **main.**

Nella stessa maniera avviene la ricorsione, lo stack salva istanza su istanza, ed ogni volta che torna alla chiamata precedente ricarica i valori di quell'istanza.

10 – ESERCIZI

In quest'ultima parte riporto una serie di esercizi-tipo che potrebbero esservi utili nel comprendere l'Assembly. Li ho divisi per tipologia (operazioni, I/O, array, funzioni-stack).

Operazioni:

- Mettere in $s0 il più grande valore tra $s1 e $s2
- Mettere in $s0 il più piccolo valore tra $s1, $s2 e $s2
- Fare un programma dove il registro $t0 conta da -25 a 0 con passi di +1 (-25,-24,...,-1,0)
- Fare un programma dove il registro $t0 conta da -25 a +25 con passi di +1 (-25,-24,...,24,25)
- Fare un programma dove il registro $t0 conta da $s0 a 0 con passi di -1 ($s0 positivo)
- Fare un programma dove il registro $t0 conta da $s0 a 0 con passi di +1 ($s0 negativo)
- Fare un programma dove il registro $t0 conta da $s0 a $s1 con passi di -1 ($s0 > $s1)
- Fare un programma dove il registro $t0 conta da $s0 a $s1 con passi di +1 ($s0 < $s1)
- Fare un programma che fa una moltiplicazione attraverso un ciclo di somme. I due fattori sono in $s0 e $s1. Il risultato va in $v0
- Scrivere un programma che scambia i valori di 2 registri
- Dati 2 valori in due variabili, scrivere il valore di metà in una terza variabile (approssimando il valore se non intero). (1,2 → 1) (3,5 → 4) (2,1 → 1) (2, -2 → 0)
- Scrivere un programma che fa 'n' nop, dove n è il valore in $a0
- Dati 4 valori nei registri $s0, $s1, $s2, $s3 mettere in $t0 il maggiore e in $t1 il minore
- Fare un programma che fa le divisioni mediante sottrazioni successive. Mettere in $a0 1 se la divisione è indeterminata/impossibile. Mettere in $a1 il risultato della divisione. Mettere in $a2 il resto. Il divisore e il dividendo sono rispettivamente in $s0 e $s1.

I/O:

- Scrivere un programma che riceve da input una stringa e la stampa

Array:

- Scrivere un programma che data una stringa di caratteri, trasforma tutti i caratteri maiuscoli in caratteri minuscoli (CIAO → ciao)
- Scrivere un programma che data una stringa di caratteri, trasforma tutti i caratteri minuscoli in caratteri maiuscoli (ciao → CIAO)
- Scrivere un programma che data una stringa di caratteri, trasforma tutte le lettere minuscole in lettere maiuscole, viceversa le lettere maiuscole. Gli altri caratteri non vengono toccati (cIAo!! → CiaO!!)
- Scrivere un programma che dato un array di interi dichiarato come var. globale salva in $v0 la somma dei suoi valori.
- Dati due array, uno di soli veri e l'altro di interi, copia i valori del secondo nell'altro

Funzioni-stack:

- Scrivere un programma che riceve da input un intero, passa quel valore ad una funzione che ne calcola il quadrato, e poi tornando il risultato alla funzione chiamante, lo stampa
- Scrivere un programma che passando due valori in $a0 e $a1, li scambia e all'uscita li restituisce in $v0 e $v1.
- Scrivere un programma che ricevuta da input una stringa di '(' e ')' la passa ad una funzione controlla che restituisce 1 se le parentesi sono ordinate, 0 altrimenti. Se il valore di ritorno è 0 il programma stamperà “La stringa non è corretta”.
  Esempio: “()(()())” → 1 “(()))(“ → 0