

## 5' Esercitazione

<https://politecnicomilano.webex.com/meet/gianenrico.conti>

Gian Enrico Conti  
MIPS and MARS

## Esercizi:

- 1) Recap sui registri
- 2) "Thinking in ASM"
- 3) MIPS asm e cross - compilazione (cenni)
- 4) MARS introduzione
- 5) Esempi di programmi semplici
- 6) Frame pointer: concetto ed uso:
  - Salvare i FP
  - Recuperare FP
  - Usare FP
- 7) Esempi di programmi semplici con chiamate a funzioni

(Nella prossima esercitazione 2 H saranno dedicati ad esempio call a funzioni più complesse con frame di attivazione completo)

# Outline

---

## ■ Linguaggio Assembly MIPS

- Simulatore MARS
- Struttura di programma
- Dichiarazione di dati
- Registri
- Istruzioni
  - Istruzioni standard
  - Pseudoistruzioni standard
  - Pseudoistruzioni estese
- Traduzione di costrutti
  - IF
  - IF - ELSE
  - WHILE
  - DO... WHILE
  - FOR
- Chiamata a funzione
  - Salvataggio di contesto

# Simulatore MARS

---

## ■ Il simulatore MARS

- IDE per il linguaggio assembly MIPS
- Implementato in Java ( quindi richiede una JRE )
- Sviluppato da Missouri State University
- Scaricabile da: <http://courses.missouristate.edu/kenvollmar/mars/>

## ■ Caratteristiche

- Interfaccia grafica e editor integrato
- Registri e memoria editabili
- Visualizza valori in decimale ed esadecimale
- Esecuzione passo-passo

# Simulatore MARS: UI

---

- Vediamo live...
- MARS suggerisce...

**MARS: Mips Assembler and Runtime Simulator**

Version 4.4 Copyright (c) 2003-2013

Pete Sanderson and Kenneth Vollmar

- download:
- <https://courses.missouristate.edu/KenVollmar/MARS/download.htm>

# Crosscompilazione

---

- Non tutti i  $\mu$ p sono disp x compilazione diretta
- Compilatore "cross" verso una o altra architettura
- Lib. Binarie apposite
  
- Opzioni di compilazione:
- p.es. `-target mipsel-linux-gnu`

# Crosscompilazione

---

```
sudo apt install gcc-mips-linux-gnu
```

# Crosscompilazione

---

```
short tot = 0;  
short  a=0xAA;  
short  b=0xBB;
```

```
int  main(void)  
{  
    tot = a + b;  
}
```

```
mips-linux-gnu-gcc test.c -S -o temp.asm
```



# Crosscompilazione

---

..

```
tot:
    .space 2
    .globl a
    .data
    .align 1
    .type a, @object
    .size a, 2
a:
    .half 170
    .globl b
    .align 1
    .type b, @object
    .size b, 2
b:
    .half 187
    .text
    .align 2
```

# Crosscompilazione

---

.. **.text**

```
        .align    2
        .globl    main
        .set      nomips16
        .set      nomicromips
        .ent      main
        .type     main, @function
main:
        .frame    $fp,8,$31
```

# Struttura di un programma

---

- File testuali con dichiarazione di dati, istruzioni ( estensione .asm per MARS )
- Sezione di dichiarazione dati seguita dalla sezione istruzioni
- **Dichiarazione dati ( .data, 0x10010000 )**
  - Posizionata in una sezione identificata dalla direttiva `.data`
  - Dichiarare i nomi delle variabili usate dal programma
  - Allocare in memoria centrale ( RAM )
- **Codice ( .text, 0x00400000 )**
  - Posizionate in una sezione identificata dalla direttiva `.text`
  - Contiene le istruzioni del programma
  - Inizio: identificato dall'etichetta `main`
  - Fine: dovrebbe utilizzare una `exit system call`
- **Commenti**
  - Tutto ciò che è seguito da un `#`
    - `# questo è considerato un commento`

# Template di un programma

---

```
#-----  
# Program      :  
# Written by   :  
# Date        :  
# Description:  
#-----  
  
# DATA Segment  
    .data  
  
# CODE Segment  
    .text  
main:                                # First instruction of the main file  
  
    li $v0,10                        # "tipo" (selettore)della syscall  
main_:    syscall
```

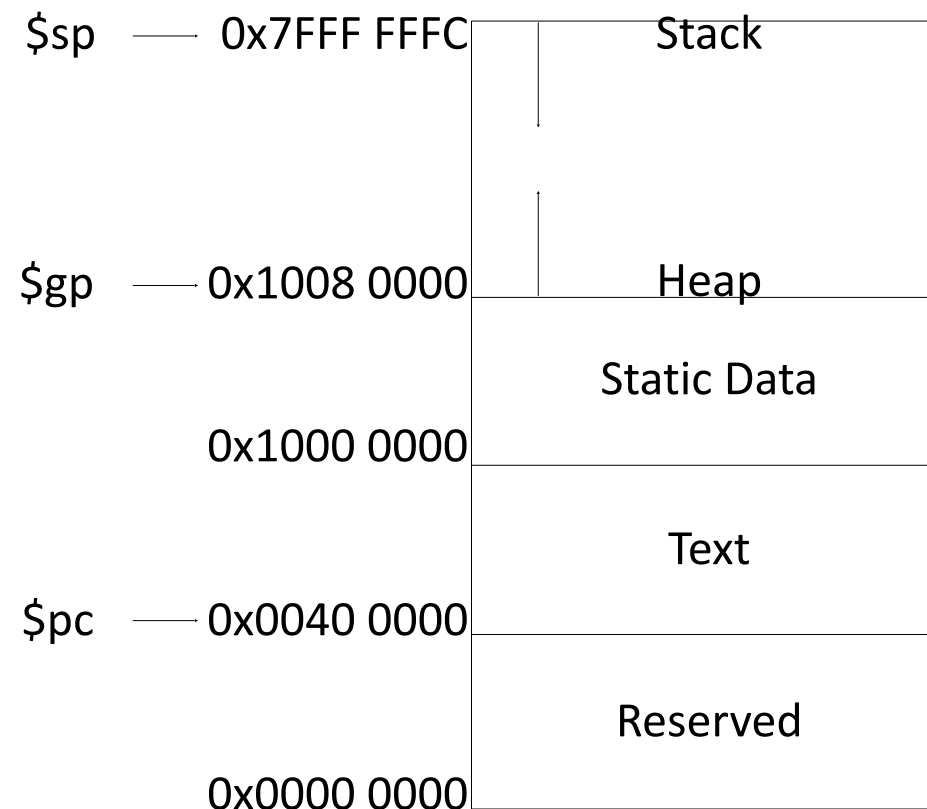
# Memoria di un programma

- **Architettura MIPS a 32 bit**

- Memoria indirizzabile → 4 GB

- **Struttura della memoria**

- Ogni segmento viene allocato in una posizione di memoria predeterminata



# Dichiarazione di dati (1)

---

- Il formato per la dichiarazione di dati è:

```
nome:           tipo_dato           valore(i)
```

- I tipi di dato principali sono:

- `.word` memorizza il dato in 32 bit ( 4 bytes )
- `.space` specifica il numero di bytes da utilizzare
- `.ascii` memorizza la stringa e aggiunge il terminatore di stringa (`'\0'`)
- `.float` memorizza il dato come numero a precisione singola ( 32 bit )
- `.double` memorizza il dato come numero a precisione doppia ( 64 bit )
- `.byte` memorizza il dato come singolo byte ( 8 bit )

# Dichiarazione di dati (2)

---

```
var1:      .word    3          # crea una singola variabile intera con valore
                                # iniziale 3

array1:    .byte     'a','b'   # crea un array di caratteri da due elementi
                                # inizializzato ad a e b

array2:    .space    40        # alloca 40 bytes consecutive, senza inizializzare
                                # lo spazio, che può essere usato come array di
                                # caratteri ma anche come array di interi. E'
                                # consigliato commentare specificando cosa si
                                # dovrebbe memorizzare!
```

# Istruzioni

---

- Le istruzioni supportate da MARS possono essere divise in tre diverse categorie:

- *Istruzioni standard*                      istruzioni nativamente supportate dall'architettura MIPS
- *Pseudoistruzioni standard*            istruzioni non supportate nativamente dall'architettura, ma che fanno parte dello standard
- *Pseudoistruzioni estese*              istruzioni non supportate nativamente dall'architettura, definite dal simulatore come utilità

- MIPS info Sheet:

[https://www.cs.tufts.edu/comp/140/lectures/Day\\_3/mips\\_summary.pdf](https://www.cs.tufts.edu/comp/140/lectures/Day_3/mips_summary.pdf)

-



# Pseudoistruzioni standard

---

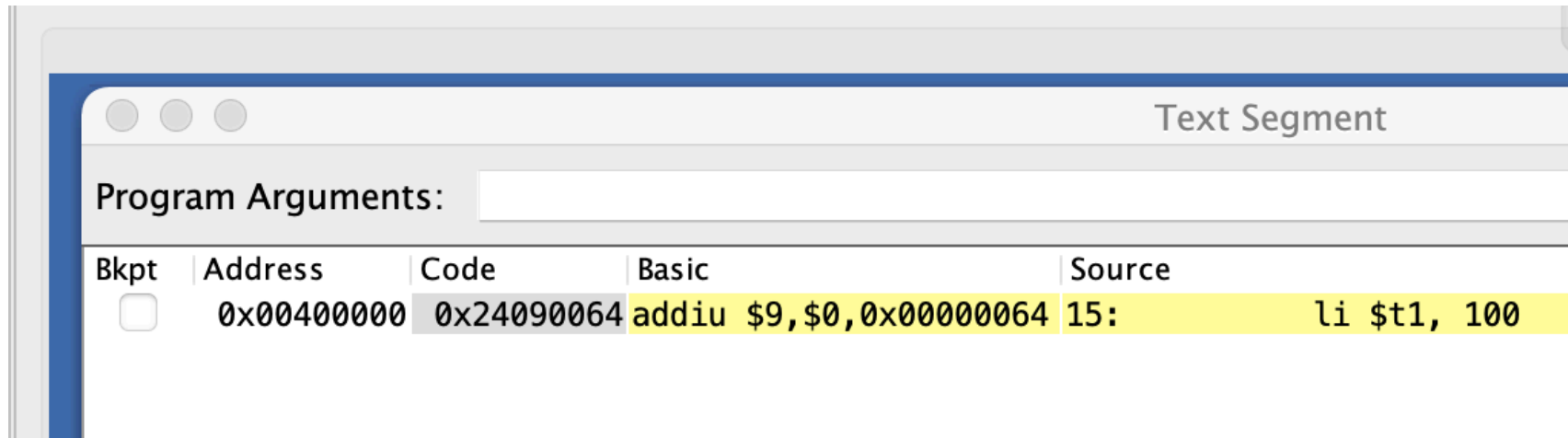
- Nella tabella qui sotto, la lista delle pseudoistruzioni standard MIPS

Pseudo instruction	
bge	rx,ry,imm
bgt	rx,ry,imm
ble	rx,ry,imm
blt	rx,ry,imm
la	rx,label
li	rx,imm
move	rx,ry
nop	

- Mars ve le fa vedere...

# Pseudoistruzioni in MARS

```
#-----  
# Program      :  
# Written by   :  
# Date        :  
# Description:  
#-----  
  
# DATA Segment  
    .data  
  
# CODE Segment  
    .text  
main:  
  
    li $t1, 100
```



Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x24090064	addiu \$9,\$0,0x00000064	15: li \$t1, 100

# Registri e convenzioni

---

- 2 formati formati x accesso ai dati:
  - using register number ex \$0 through \$31
  - using equivalent names ex \$t1, \$sp
- Convenzioni sui registri
  - \$t0 - \$t9 ( = \$8 - \$15, \$24, \$25) are general use registers; need **not be preserved** across procedure calls
  - \$s0 - \$s7 ( = \$16 - \$23) are general use registers; **should be preserved** across procedure calls
  - \$sp ( = \$29) is stack pointer
  - \$fp ( = \$30) is frame pointer
  - \$ra ( = \$31) is return address storage for subroutine call (**R**eturn **A**ddress)
  - \$a0 - \$a3 ( = \$4 - \$7) are used to pass arguments to subroutines
  - \$v0, \$v1 ( = \$2, \$3) are used to hold return values from subroutine
- special registers Lo and Hi used to store result of multiplication and division

# Alcuni esempi di codice

---

**Note:**

- presenteremo alcuni esempi C solo come pseudo code
- Pensate sempre in "ASM"
  - Registri
  - Celle
  - Indirizzi

# esercizio1.asm

---

```
# Written by : ingconti  
# Date      :  
# Description: EX05_es1  
#-----
```

```
    .data  
    .word    10, 11, 12  
  
    .text  
addi    $s0, $zero, 2  
addi    $s1, $s0,   5  
mul     $s2, $s1, $s0  
add     $s2, $s2, $s0  
sub     $s3, $s2, $s1
```

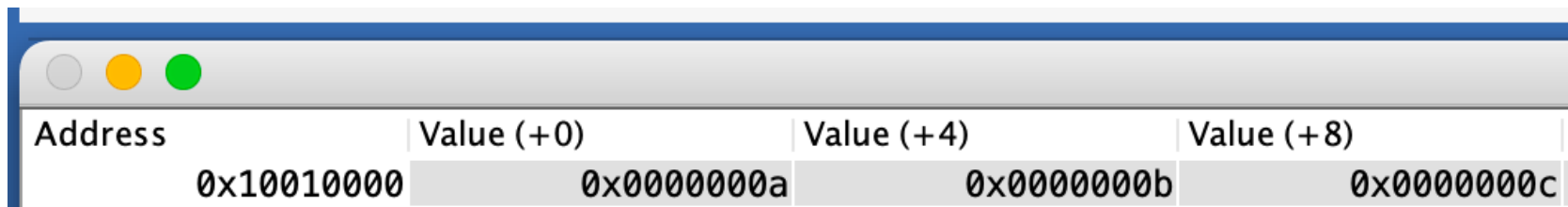
# EX05\_es1.asm

```
.data
.word    10, 11, 12

.text
addi     $s0, $zero, 2
addi     $s1, $s0,   5
mul      $s2, $s1, $s0
add      $s2, $s2, $s0
sub      $s3, $s2, $s1
```

**Si noti la RAM**

**e i registri in MARS..**



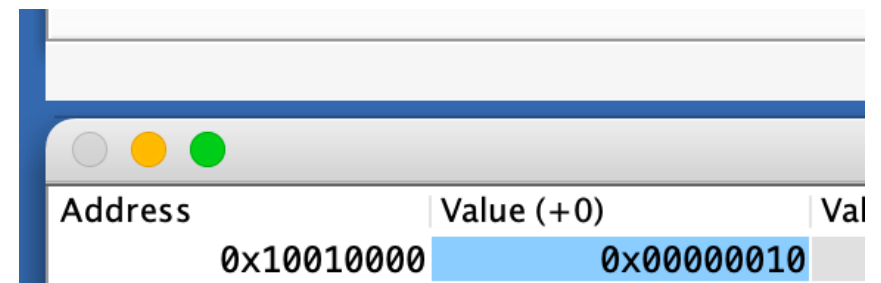
Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x0000000a	0x0000000b	0x0000000c

# EX05\_es2.asm

```
#-----  
# Program      :  
# Written by   : ingconti  
# Date        :  
# Description: EX05_es2.asm:  
#using memory and addresses  
#-----  
.data  
A:      .word      21  
  
.text  
    la      $s0, A  
    lw      $s1, ($s0)  
    addi    $s1, $s1, -5  
    sw      $s1, ($s0)
```

Accediamo alla v. A e  
E la cambiamo.

ALLa fine:  
A = 16



The screenshot shows a memory viewer window with a title bar and three colored buttons (gray, yellow, green). Below the title bar is a table with three columns: 'Address', 'Value (+0)', and 'Val'. The first row of the table shows the address '0x10010000' and the value '0x00000010'.

Address	Value (+0)	Val
0x10010000	0x00000010	

# Traduzione - IF - prerequisiti: Control Flow Instructions

---

## Branches

- comparison for conditional branches is built into instruction

b	target	#	unconditional branch to program label target
beq	\$t0,\$t1,target	#	branch to target if \$t0 = \$t1
blt	\$t0,\$t1,target	#	branch to target if \$t0 < \$t1
ble	\$t0,\$t1,target	#	branch to target if \$t0 <= \$t1
bgt	\$t0,\$t1,target	#	branch to target if \$t0 > \$t1
bge	\$t0,\$t1,target	#	branch to target if \$t0 >= \$t1
bne	\$t0,\$t1,target	#	branch to target if \$t0 <> \$t1

## Jumps

j	target	#	unconditional jump to program label target
jr	\$t3	#	jump to address contained in \$t3 ("jump register")



# Traduzione - IF

---

Linguaggio C

```
A
if( cond ) {
    B
}
C
```

Assembly MIPS

```
 $\tau(A)$ 
 $\tau(cond)$ 
BEQ cond, zero, end
 $\tau(B)$ 
end:  $\tau(C)$ 
```

# Traduzione - IF: esempio

---

Linguaggio C

```
int a = 1;
int b = 3;

a = a + b;
if( a > b )
{
    b = b +
a;
}
```

Assembly MIPS

```
τ(A)
τ(cond)
BEQ cond, zero, end
τ(B)
end: τ(C)

"SALTO SE ZERO",
invertire logica.
```

# EX05\_es3\_IF.asm

---

```
int a = 1;
int b = 3;

a = a + b;
if( a > b ) {
    b = b + a;
}
```

2 valori 32 bit  
X semplicità 2 registri

= -> load

Ma poco efficiente..  
ADD!

....

(Nota: potremmo usare  
anche .data...)

# esercizio1.asm

2 valori 32 bit

= -> load

```
li $s1, 1
```

```
li $s2, 3
```

MARS lo fa x voi..

Text Segment		
:		
Code	Basic	Source
0x24110001	addiu \$17,\$0,0x0000...	5: li \$s1, 1
0x24120003	addiu \$18,\$0,0x0000...	6: li \$s2, 3
0x02118020	add \$16,\$16,\$17	7: add \$s0, \$s0, \$s1
0x0230902a	slt \$18,\$17,\$16	8: sgt \$s2, \$s0, \$s1
0x12400001	beq \$18,\$0,0x00000001	9: beq \$s2, \$zero, end
0x02308820	add \$17,\$17,\$16	10: add \$s1, \$s1, \$s0

# esercizio1.asm

---

```
int a = 1;  
int b = 3;
```

```
a = a + b;  
if( a > b ) {  
    b = b + a;  
}
```

2 valori 32 bit  
X semplicità 2 registri  
= -> load

Somma, ma va spostato tutto nei  
registri..

**If ->**

**sgt Rdest, Rsrc1, Src2 +  
 Jmp condizionato..**

**sgt Rdest, Rsrc1, Src2 Set Greater Than**

Set register Rdest to 1 if register Rsrc1 is greater than Src2  
and to 0 otherwise.

# EX05\_es3\_IF.asm

```
int a = 1;
int b = 3;

a = a + b;
if( a > b ) {
    b = b + a;
}
```

#EX05\_es3\_IF.asm

.data

.text

```
addi    $s0, $zero, 1
addi    $s1, $zero, 3
```

```
add     $s0, $s0, $s1 #sum: a = a + b
sgt     $s2, $s0, $s1 #cmp s0 > s1
                        #i.e. a > b
```

```
beq     $s2, $zero, end
add     $s1, $s1, $s0
```

end:

Alla fine \$s1 = 7

\$t7	15	0x00000000
\$s0	16	0x00000004
\$s1	17	0x00000007

# Traduzione – IF ... ELSE

---

## Linguaggio C

```
A
if( cond ) {
    B
}
else {
    C
}
D
```

## Assembly MIPS

```
 $\tau(A)$ 
 $\tau(cond)$ 
BEQ cond, zero, else
 $\tau(B)$ 
    B end // ATTENZIONE!
else:  $\tau(C)$ 
end:  $\tau(D)$ 
```

Qui:

- 1) "doppio" salto, 1 condizionato + 1 assoluto
- 2) Due strade x AND
  - A) NON ho AND, quindi "spezzo" in due e salto sull' opposto delle due condizioni
  - B) **AND fra registri...**

# Traduzione – IF ... ELSE

```
int a = 2;  
int b = 3;
```

```
int c;  
if( a > b && b > 0 ){  
    c = a + b;  
}  
else {  
    c = a - b;  
}
```

```
#-----  
# Description: EX05_es4_IF_ELSE.asm  
#-----  
        .data  
  
        .text  
addi    $s0, $zero, 2  
addi    $s1, $zero, 3  
sgt     $s2, $s0, $s1    # s0>s1, i.e. a>b  
sgt     $s3, $s1, $zero  # s1>0  
and     $s2, $s2, $s3    # AND e testero' s2  
beq     $s2, $zero, else  
add     $s4, $s0, $s1  
j       end              #salto assoluto (else)  
else:   sub     $s4, $s0, $s1  
end:
```



# Traduzione – WHILE

---

## Linguaggio C

```
A
while( cond ) {
    B
}    /// SALTO incodizionato!
C
```

## Assembly MIPS

```
          τ (A)
cond:     τ (cond)
          BEQ      cond, zero, end
          τ (B)
          B         cond
end:      τ (C)
```

# WHILE codice

---

```
int i = 3;
while( i > 0 ) {
    i--;
}
//end
```

```
.data
.text
addi    $s0, $zero, 3
cond:   sgt    $s1, $s0, $zero
beq     $s1, $zero, end
addi    $s0, $s0, -1
b       cond
end:
```

# Traduzione – DO... WHILE

---

Linguaggio C

```
A
do {
    B
} while( cond );
C
```

Assembly MIPS

```
        τ (A)
do:      τ (B)
        τ (cond)
        BNE    cond, zero, do
end:     τ (C)
```

# Traduzione – DO... WHILE

---

```
int sum = 0;
int i    = 0;

do {
    sum = sum + i;
    i   = i   + 1;
} while( i < 5 );
```

```
#-----
# Description: EX05_es5_D0.asm
#-----
.data
.text
addi    $s0, $zero, 3
cond:   sgt    $s1, $s0, $zero
beq     $s1, $zero, end
addi    $s0, $s0, -1
b       cond
end:
```

# Traduzione – FOR

---

## Linguaggio C

```
A
for( init; cond; inc ){
    B
}
C
```

## Assembly MIPS

```

    τ (A)
init:  τ (init)
cond:  τ (cond)
       BEQ      cond, zero, end
       τ (B)
inc:   τ (inc)
       B        cond
end:   τ (C)
```

# FOR

---

```
int i;  
int f = 1;  
  
for( i = 0; i < 5; i++ )  
{  
    f = f * i;  
}
```

```
#-----  
# Description: EX05_es7_FOR.asm  
#-----  
.data  
.text  
addi    $s0, $zero, 1  
init:   add    $s1, $zero, $zero  
cond:   sle    $s2, $s1, 5  
        bne    $s2, $zero, end  
        mul    $s0, $s0, $s1  
inc:    addi   $s1, $s1, 1  
        b      cond  
end:
```

# Chiamata a funzione

---

## ■ Esistono delle strutture hardware a supporto delle chiamate a funzione

subroutine call: "jump and link" instruction

```
jal sub_label# "jump and link"
```

- copy program counter (return address) to register \$ra (return address register)
- jump to program statement at sub\_label

subroutine return: "jump register" instruction

```
jr $ra # "jump register"
```

- jump to return address in \$ra (stored by jal instruction)

Note: return address stored in register \$ra; if subroutine will call other subroutines, or is recursive, return address should be copied from \$ra onto stack to preserve it, since jal always places return address in this register and hence will overwrite previous value

# Chiamata a funzione: EX05\_es8\_CALL\_F.asm

```
int main( void )
{
    int a = leaf( 1, 2, 3, 4 );
}

int leaf( int g, int h, int i, int j )
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

```
.data
.text
main: addi $a0, $zero, 1
      addi $a1, $zero, 2
      addi $a2, $zero, 3
      addi $a3, $zero, 4
      jal  leaf
      add  $s0, $v0, $zero
      addi $v0, $zero, 10
      syscall

leaf: addi $sp, $sp, -12
      sw   $t1, 8($sp)
      sw   $t0, 4($sp)
      sw   $s0, 0($sp)
      add  $t0, $a0, $a1
      add  $t1, $a2, $a3
      sub  $s0, $t0, $t1
      add  $v0, $s0, $zero
      lw   $s0, 0($sp)
      lw   $t0, 4($sp)
      lw   $t1, 8($sp)
      addi $sp, $sp, 12
      jr   $ra
```



# Salvataggio di contesto

- Durante la chiamata a funzione può rendersi necessario salvare alcuni registri (contesto) per far sì che il chiamante continui a funzionare correttamente

