SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

# Architettura degli Elaboratori
# Lez. 5 – ASM: Funzioni e procedure

Prof. Andrea Sterbini – sterbini@di.uniroma1.it

# Argomenti

- **Argomenti della lezione**
  - Soluzione dell'esercizio
  - Definizione di funzioni/procedure semplici
  - Record di Attivazione

# Argomenti

- **Argomenti della lezione**
  - Soluzione dell'esercizio
  - Definizione di funzioni/procedure semplici
  - Record di Attivazione

- **Funzione / procedura:**

Frammento di codice che riceve degli argomenti e calcola un risultato

- ha un indirizzo di partenza
- riceve uno o più argomenti
- svolge un calcolo
- ritorna un risultato
- continua la sua esecuzione dall'istruzione seguente a quella che l'ha chiamata

# Istruzioni necessarie

**Per chiamare la funzione/procedura**

`jal` ***`etichetta`*** **J**ump **A**nd **L**ink

(simile ad un salto a subroutine ma più limitato)

ricorda nel registro **`$ra`** la posizione dell'istruzione successiva ( **$ra <- PC+4** )

cambia il PC per iniziare l'esecuzione del corpo della funzione ( **PC <- etichetta** )

# Istruzioni necessarie

**Per chiamare la funzione/procedura**

`jal` ***`etichetta`*** **J**ump **A**nd **L**ink

(simile ad un salto a subroutine ma più limitato)

ricorda nel registro `$ra` la posizione dell'istruzione successiva ( **$ra <- PC+4** )

cambia il PC per iniziare l'esecuzione del corpo della funzione ( **PC <- etichetta** )

**Per tornare e continuare l'esecuzione del programma chiamante**

`jr $ra` **J**ump to **R**egister

salta all'indirizzo contenuto nel registro indicato ( **PC <- $ra)** )

# Istruzioni necessarie

**Per chiamare la funzione/procedura**

`jal` ***`etichetta`*** **J**ump **A**nd **L**ink

(simile ad un salto a subroutine ma più limitato)

ricorda nel registro `$ra` la posizione dell'istruzione successiva ( **$ra <- PC+4** )

cambia il PC per iniziare l'esecuzione del corpo della funzione ( **PC <- etichetta** )

**Per tornare e continuare l'esecuzione del programma chiamante**

`jr $ra` **J**ump to **R**egister

salta all'indirizzo contenuto nel registro indicato ( **PC <- $ra)** )

**Come passare valori ALLA funzione (caso semplice)**

`$a0`, `$a1`, `$a2`, `$a3` 4 registri per passare fino a 4 valori a 32 bit o 2 a 64 bit

# Istruzioni necessarie

**Per chiamare la funzione/procedura**

`jal` ***`etichetta`*** **J**ump **A**nd **L**ink

(simile ad un salto a subroutine ma più limitato)

ricorda nel registro `$ra` la posizione dell'istruzione successiva ( **$ra <- PC+4** )

cambia il PC per iniziare l'esecuzione del corpo della funzione ( **PC <- etichetta** )

**Per tornare e continuare l'esecuzione del programma chiamante**

`jr $ra` **J**ump to **R**egister

salta all'indirizzo contenuto nel registro indicato ( **PC <- $ra)** )

**Come passare valori ALLA funzione (caso semplice)**

`$a0`, `$a1`, `$a2`, `$a3` 4 registri per passare fino a 4 valori a 32 bit o 2 a 64 bit

**e tornarli DALLA funzione**

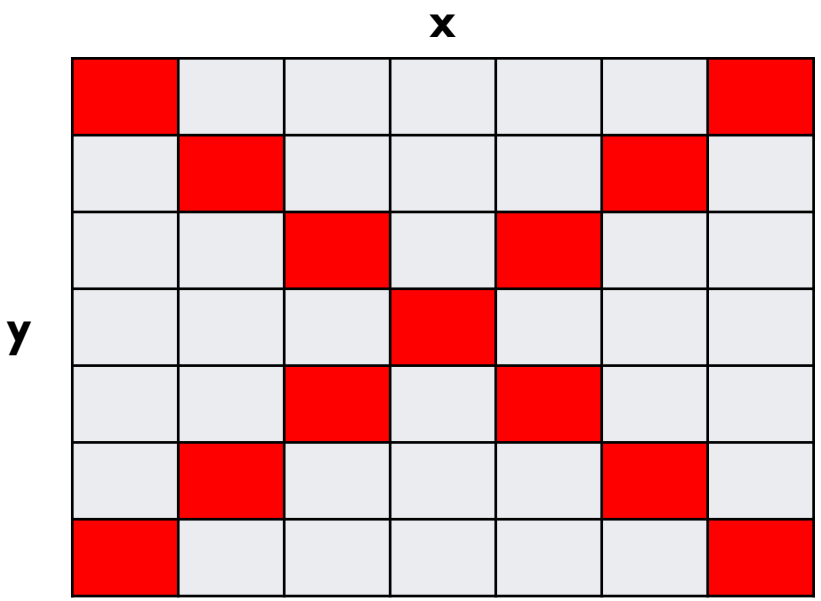
`$v0`, `$v1` 2 registri per tornare fino a 2 valori a 32 bit o 1 a 64 bit

# Esercizio

Calcolare e stampare la somma delle **due** diagonali di una matrice quadrata di word

Fate attenzione a non sommare due volte l'elemento centrale in caso di matrici di lato dispari

Suggerimento: scandite tutta la matrice e individuate le caselle sulle due diagonali



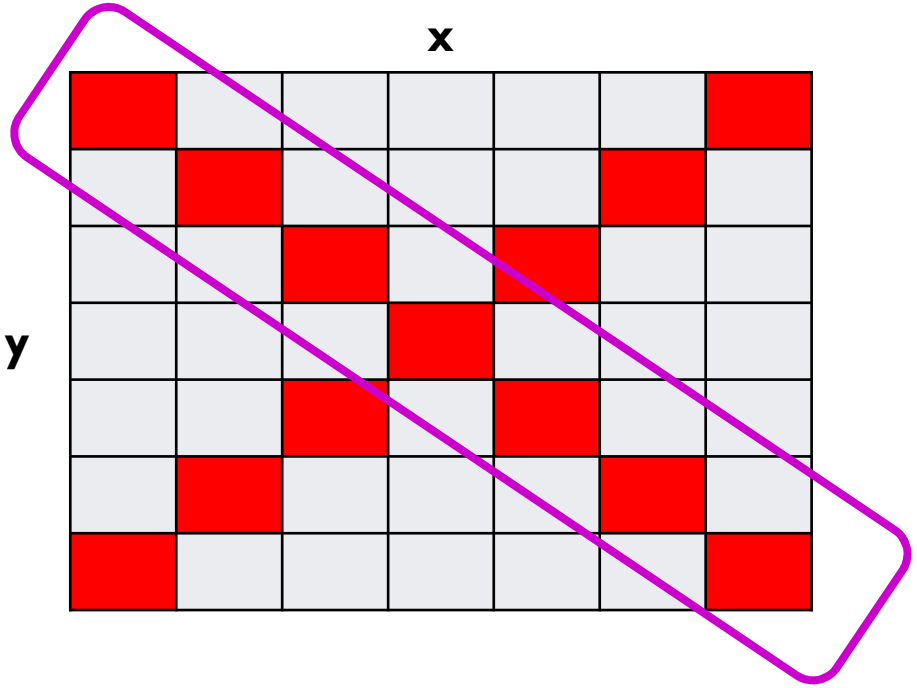
# Esercizio

Calcolare e stampare la somma delle **due** diagonali di una matrice quadrata di word

Fate attenzione a non sommare due volte l'elemento centrale in caso di matrici di lato dispari

Suggerimento: scandite tutta la matrice e individuate le caselle sulle due diagonali

**x = y**



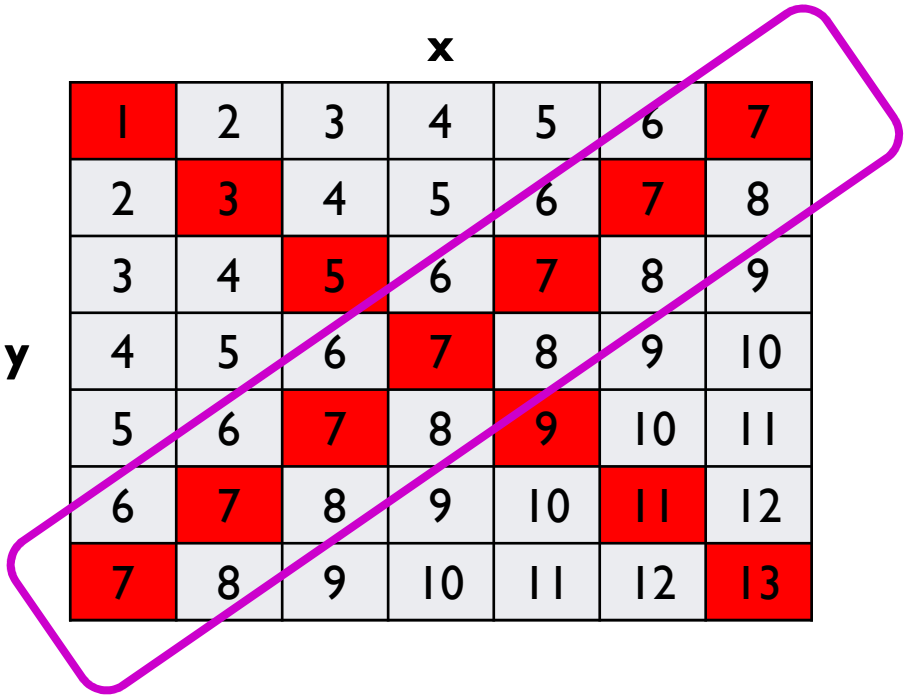
# Esercizio

Calcolare e stampare la somma delle **due** diagonali di una matrice quadrata di word

Fate attenzione a non sommare due volte l'elemento centrale in caso di matrici di lato dispari

Suggerimento: scandite tutta la matrice e individuate le caselle sulle due diagonali

**x = y**




**x+y+1 = lato**

# Soluzione con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
# $a0   coordinata x
# $a1   coordinata y
# $a2   lato della matrice
```

# Soluzione con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
        # $a0   coordinata x
        # $a1   coordinata y
        # $a2   lato della matrice
```

```
is_diagonal:
        beq     $a0, $a1, yes  # se x=y siamo sulla prima diagonale
```

# Soluzione con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
        # $a0   coordinata x
        # $a1   coordinata y
        # $a2   lato della matrice
```

```
is_diagonal:
        beq     $a0, $a1, yes  # se x=y siamo sulla prima diagonale
```

```
yes:
        li      $v0, 1          # il risultato è 1 (vero)
         # ritorno all'istruzione successiva alla chiamata
        jr      $ra
```

# Soluzione con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
        # $a0   coordinata x
        # $a1   coordinata y
        # $a2   lato della matrice
is_diagonal:
        beq     $a0, $a1, yes  # se x=y siamo sulla prima diagonale
        add     $v0, $a0, $a1  # altrimenti se la somma x+y+1
        addi    $v0, $v0, 1
        beq     $v0, $a2, yes  # è uguale al lato siamo sulla seconda




yes:
        li      $v0, 1         # il risultato è 1 (vero)
         # ritorno all'istruzione successiva alla chiamata
        jr      $ra
```

# Soluzione con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
        # $a0   coordinata x
        # $a1   coordinata y
        # $a2   lato della matrice
is_diagonal:
        beq     $a0, $a1, yes  # se x=y siamo sulla prima diagonale
        add     $v0, $a0, $a1  # altrimenti se la somma x+y+1
        addi    $v0, $v0, 1
        beq     $v0, $a2, yes  # è uguale al lato siamo sulla seconda
        # altrimenti non siamo sulle due diagonali principali
        li      $v0, 0         # il risultato è 0 (falso)
         # ritorno all'istruzione successiva alla chiamata
        jr      $ra
yes:
        li      $v0, 1         # il risultato è 1 (vero)
         # ritorno all'istruzione successiva alla chiamata
        jr      $ra
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y** (con gli stessi argomenti dell'altra)

```
        #       $a0     coordinata x
        #       $a1     coordinata y
        #       $a2     lato della matrice
leggi_elemento:
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y** (con gli stessi argomenti dell'altra)

```
        #       $a0     coordinata x
        #       $a1     coordinata y
        #       $a2     lato della matrice
leggi_elemento:
        mul     $v0, $a1, $a2           # y*LATO
        add     $v0, $v0, $a0           # x + y*LATO
        sll     $v0, $v0, 2             # offset = 4 * (x + y * LATO)
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y** (con gli stessi argomenti dell'altra)

```
        #       $a0     coordinata x
        #       $a1     coordinata y
        #       $a2     lato della matrice
leggi_elemento:
        mul     $v0, $a1, $a2           # y*LATO
        add     $v0, $v0, $a0           # x + y*LATO
        sll     $v0, $v0, 2             # offset = 4 * (x + y * LATO)
        lw      $v0, matrice($v0)       # leggo matrice[x][y]
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y** (con gli stessi argomenti dell'altra)

```
        #       $a0     coordinata x
        #       $a1     coordinata y
        #       $a2     lato della matrice
leggi_elemento:
        mul     $v0, $a1, $a2           # y*LATO
        add     $v0, $v0, $a0           # x + y*LATO
        sll     $v0, $v0, 2             # offset = 4 * (x + y * LATO)
        lw      $v0, matrice($v0)       # leggo matrice[x][y]
        jr      $ra                     # torno  l'esecuzione al chiamante
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y** (con gli stessi argomenti dell'altra)

```
        #       $a0     coordinata x
        #       $a1     coordinata y
        #       $a2     lato della matrice
leggi_elemento:
        mul     $v0, $a1, $a2           # y*LATO
        add     $v0, $v0, $a0           # x + y*LATO
        sll     $v0, $v0, 2             # offset = 4 * (x + y * LATO)
        lw      $v0, matrice($v0)       # leggo matrice[x][y]
        jr      $ra                     # torno  l'esecuzione al chiamante
```

**NOTA: cerco di non usare registri ulteriori in modo da lasciare più libertà nel programma chiamante**

**NOTA: vedremo in seguito come preservare i registri**

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
cicloY:       beq    $a1, $a2, fine       # se ultima riga
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
cicloY:       beq    $a1, $a2, fine       # se ultima riga
cicloX:       beq    $a0, $a2, nextY      # se ultima colonna
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
cicloY:       beq    $a1, $a2, fine       # se ultima riga
cicloX:       beq    $a0, $a2, nextY      # se ultima colonna
       jal    is_diagonal           # test se sulla diagonale
       beqz   $v0, nextX            # se falso prossima X
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
cicloY:       beq    $a1, $a2, fine       # se ultima riga
cicloX:       beq    $a0, $a2, nextY      # se ultima colonna
       jal    is_diagonal           # test se sulla diagonale
       beqz   $v0, nextX            # se falso prossima X
       jal    leggi_elemento        # altrimenti leggi
       add    $t0, $t0, $v0         # e somma l'elemento
```

# Programma principale

```
.data
matrice:      .word  400:0          # matrice 20x20
LATO:  .word  20                    # lato della matrice
.text
main:  li     $a0, 0                # x = 0
       li     $a1, 0                # y = 0
       lw     $a2, LATO             # lato della matrice
       li     $t0, 0                # somma = 0
cicloY:       beq    $a1, $a2, fine       # se ultima riga
cicloX:       beq    $a0, $a2, nextY      # se ultima colonna
       jal    is_diagonal           # test se sulla diagonale
       beqz   $v0, nextX            # se falso prossima X
       jal    leggi_elemento        # altrimenti leggi
       add    $t0, $t0, $v0         # e somma l'elemento
nextX: addi   $a0, $a0, 1           # x += 1 (prossima colonna)
       j      cicloX
```

# (segue)

```
nextY: addi   $a1, $a1, 1   # y += 1 (prossima riga)
       li     $a0, 0        # x = 0  (colonna 0)
       j      cicloY
fine:  move   $a0, $t0      # stampo la somma
       li     $v0, 1        # syscall 1 = print integer
       syscall
       li     $v0, 10       # syscall 10 = fine
       syscall
```

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di **`$ra`**

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
- ne vengono salvate altre
- …

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro `$sp` (Stack Pointer)

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

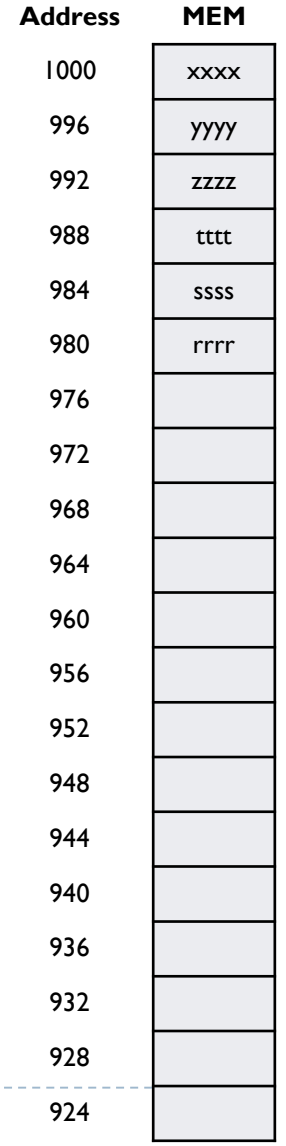
- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro `$sp` (Stack Pointer)



# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
  - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

`$sp` **980**

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro `$sp` (Stack Pointer)

| Address | MEM |
|---|---|
| 1000 | xxxx |
| 996 | yyyy |
| 992 | zzzz |
| 988 | tttt |
| 984 | ssss |
| 980 | rrrr |
| 976 | |
| 972 | |
| 968 | |
| 964 | |
| 960 | |
| 956 | |
| 952 | |
| 948 | |
| 944 | |
| 940 | |
| 936 | |
| 932 | |
| 928 | |
| 924 | |

# Preservare il contenuto dei registri

Conviene **preservare** il precedente contenuto dei registri usati dalla funzione e **ripristinarlo**

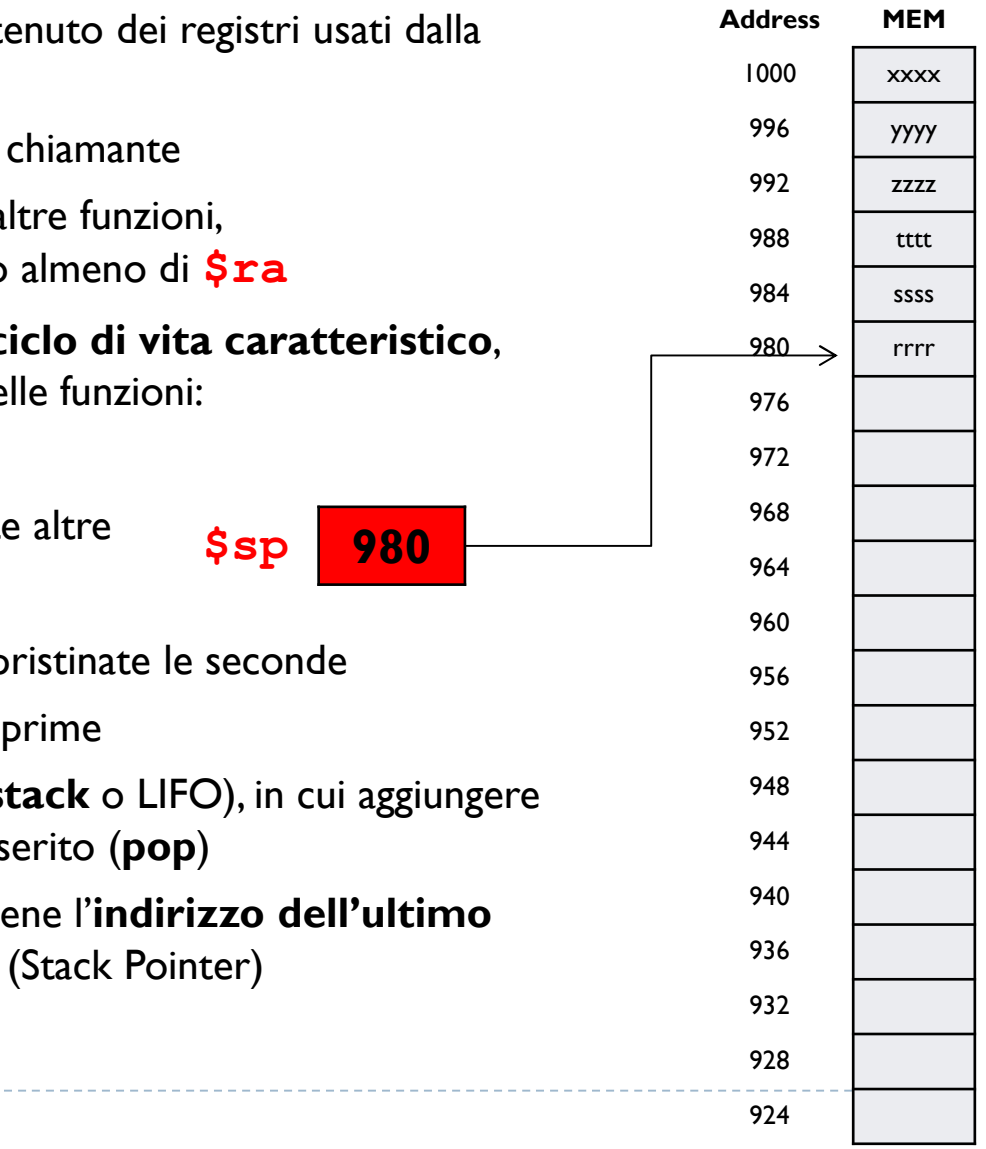
- **meno vincoli** alla funzione chiamante
- nelle funzioni che chiamano altre funzioni, che perderebbero il contenuto almeno di `$ra`

Le informazioni da preservare hanno un **ciclo di vita caratteristico**, dovuto al **nidificarsi delle chiamate** delle funzioni:

- vengono salvate le prime
  - ne vengono salvate altre
    - …
  - vengono prima ripristinate le seconde
- e poi vengono ripristinate le prime

Questo è il comportamento di una pila (**stack** o LIFO), in cui aggiungere un elemento (**push**) e togliere l'ultimo inserito (**pop**)

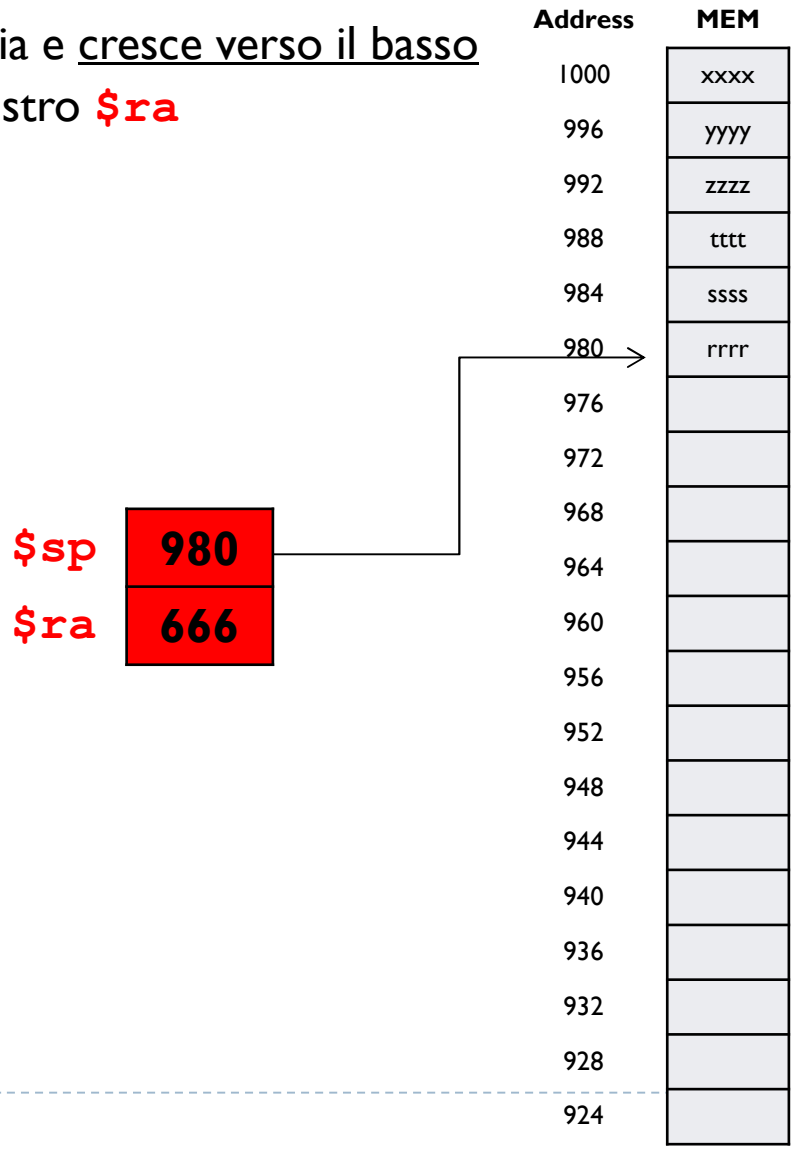
Viene realizzata con un vettore di cui si tiene l'**indirizzo dell'ultimo elemento occupato** nel registro `$sp` (Stack Pointer)



# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso

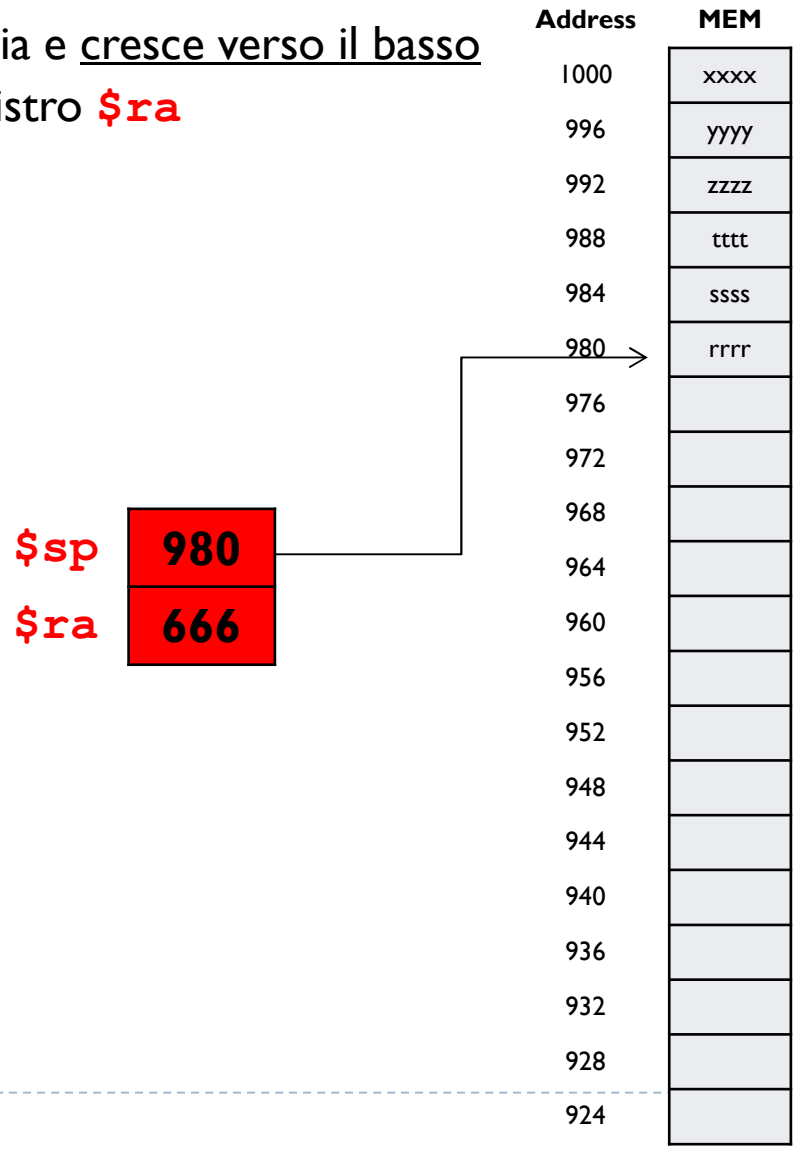
Supponiamo di voler salvare e ripristinare il registro `$ra`



# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**



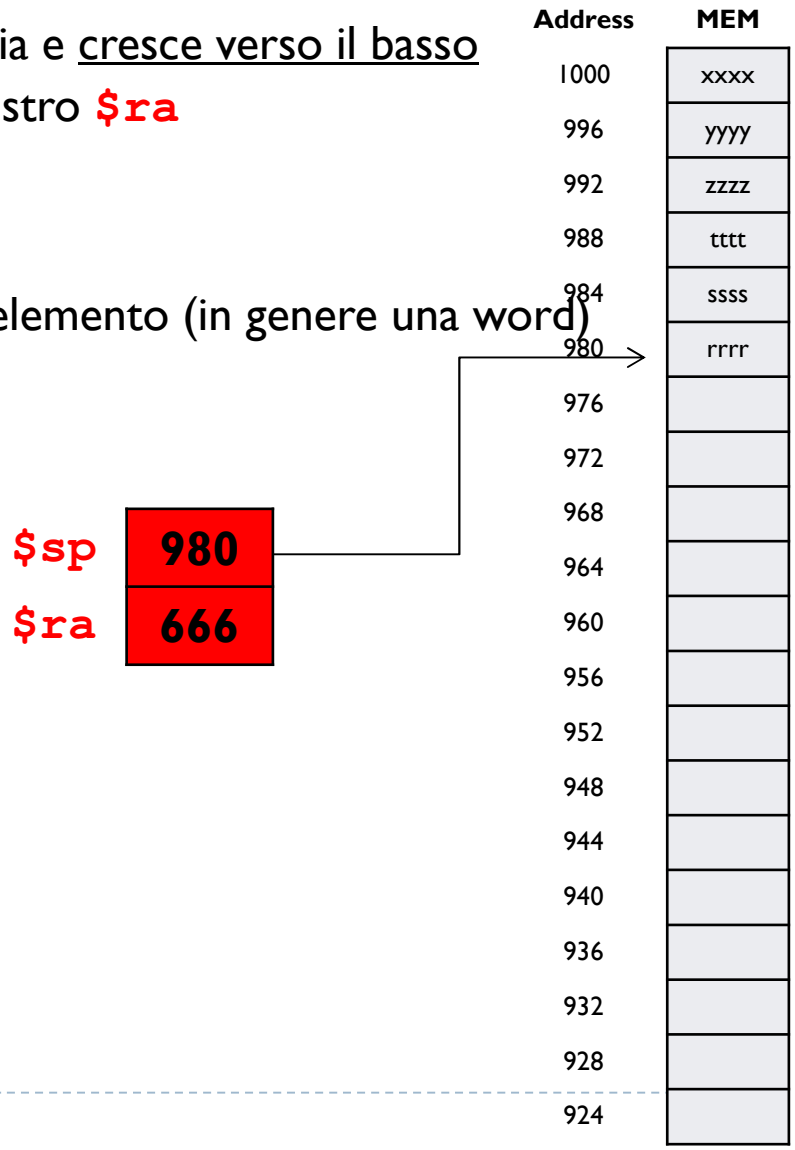
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)

```
subi   $sp, $sp, 4
```



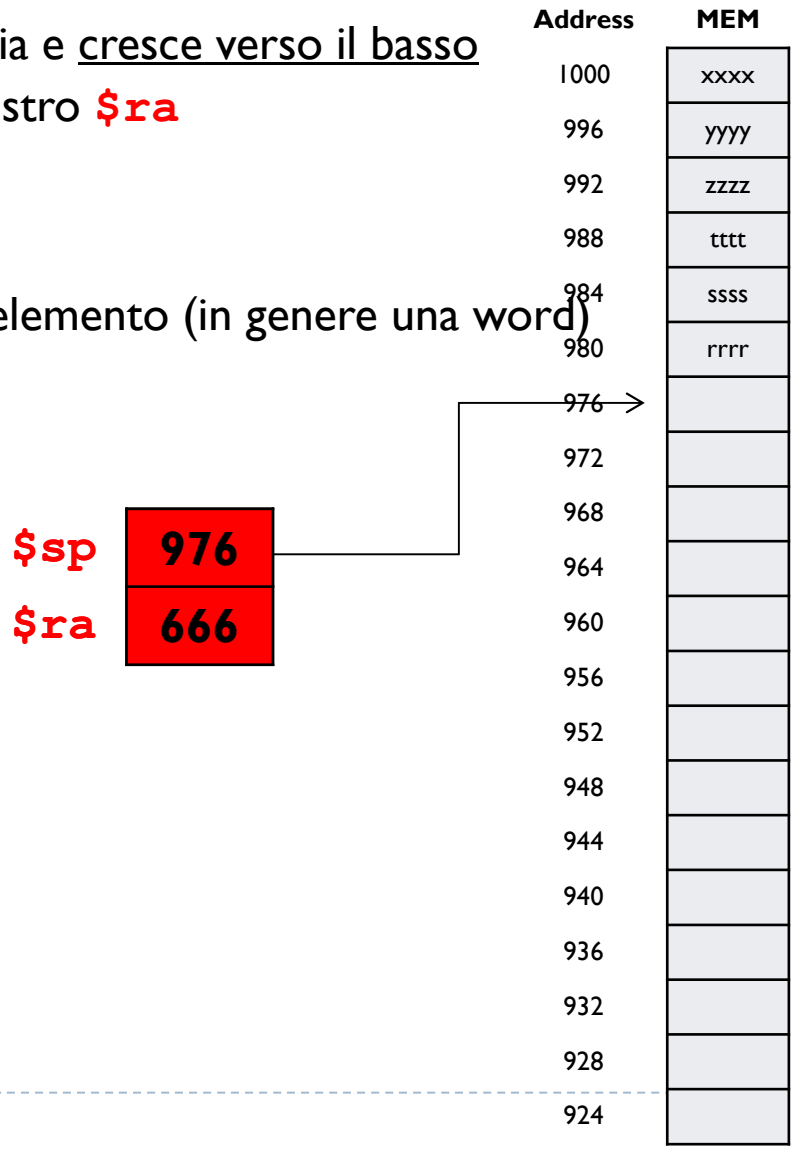
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)

```
subi   $sp, $sp, 4
```



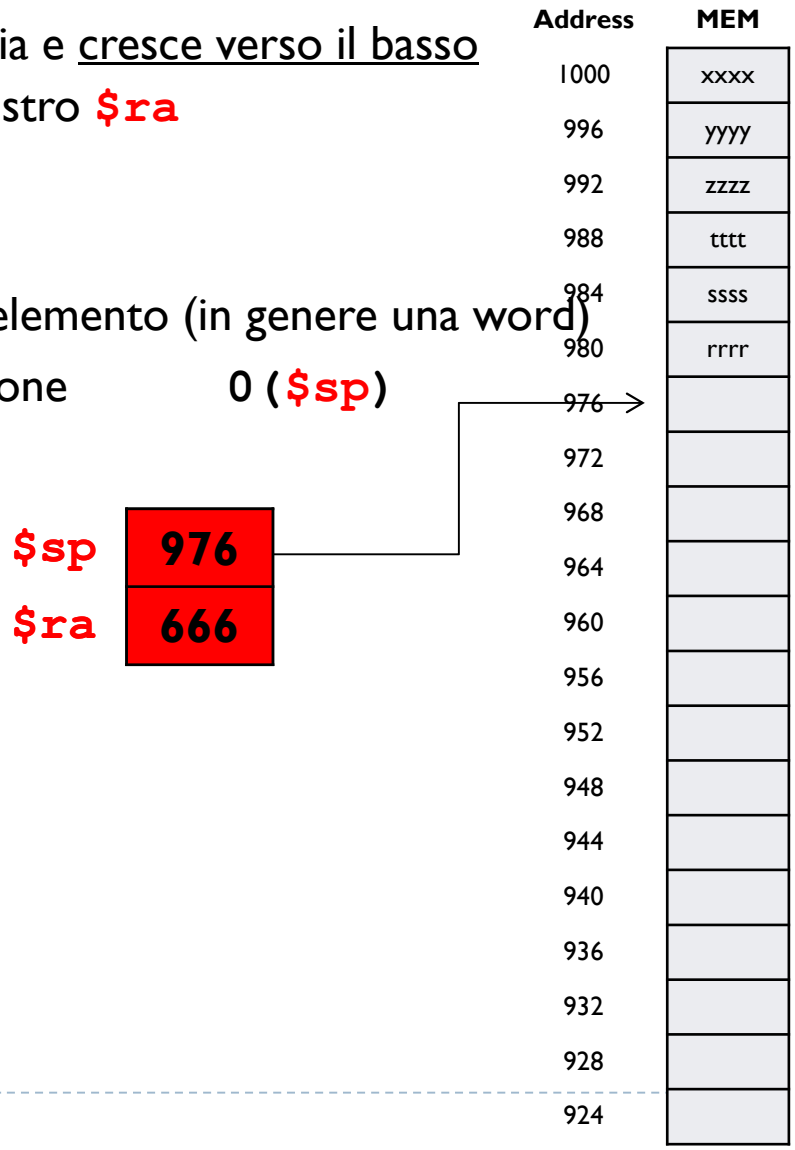
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```



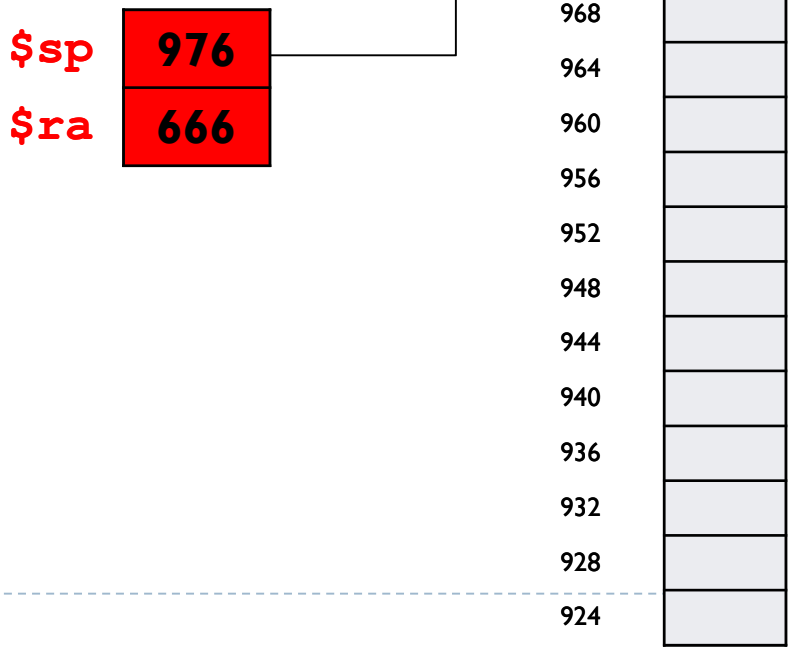
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```



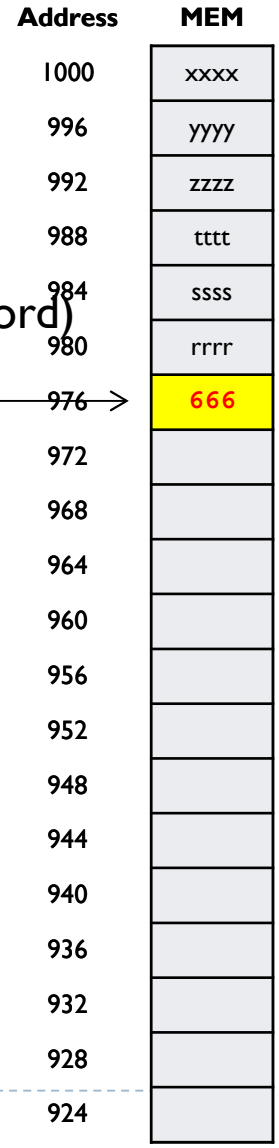
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```


**Come recuperare un elemento (pop):**

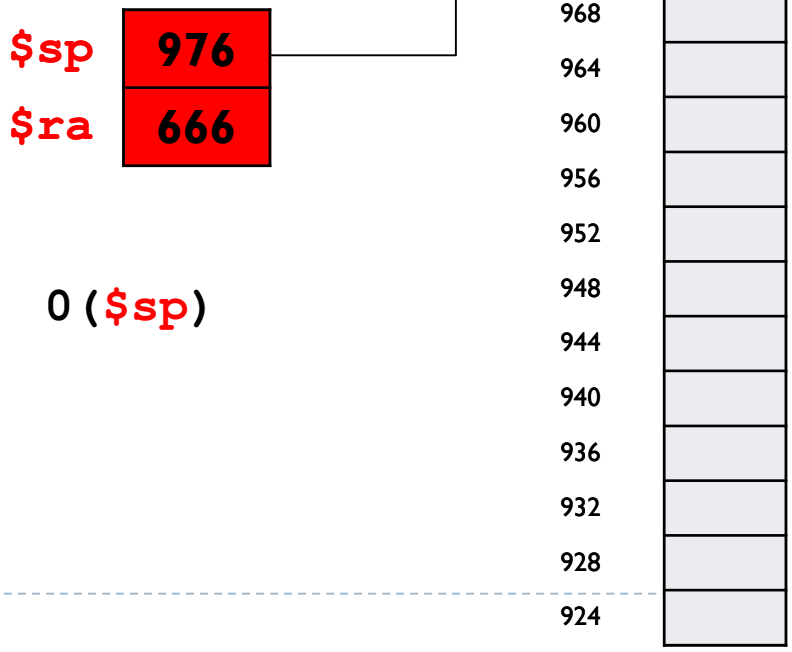
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```




**Come recuperare un elemento (pop):**

- si legge l'elemento dalla posizione `0($sp)`

```
lw      $ra, 0($sp)
```

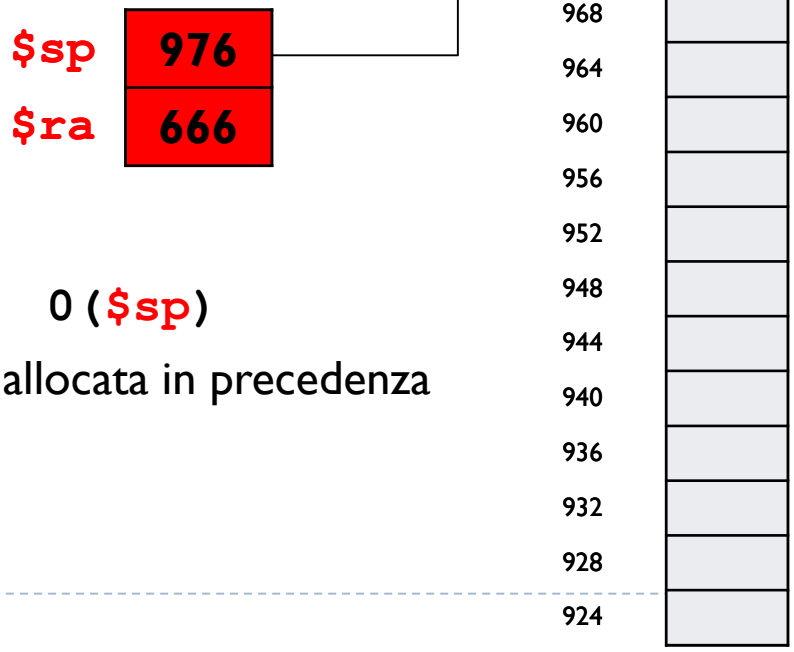
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```




**Come recuperare un elemento (pop):**

- si legge l'elemento dalla posizione `0($sp)`
- si incrementa lo `$sp` della quantità allocata in precedenza

```
lw      $ra, 0($sp)
addi    $sp, $sp, 4
```

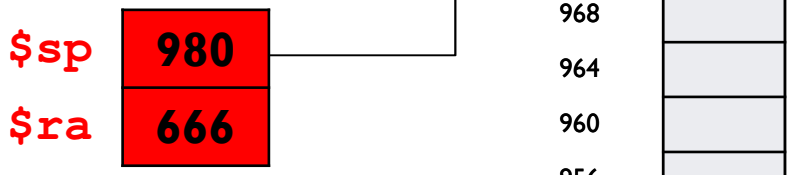
# Uso dello stack

Lo stack si trova nella parte «alta» della memoria e cresce verso il basso
Supponiamo di voler salvare e ripristinare il registro `$ra`

**Come salvare un elemento (push):**

- si decrementa lo `$sp` della dim. dell'elemento (in genere una word)
- si memorizza l'elemento nella posizione `0($sp)`

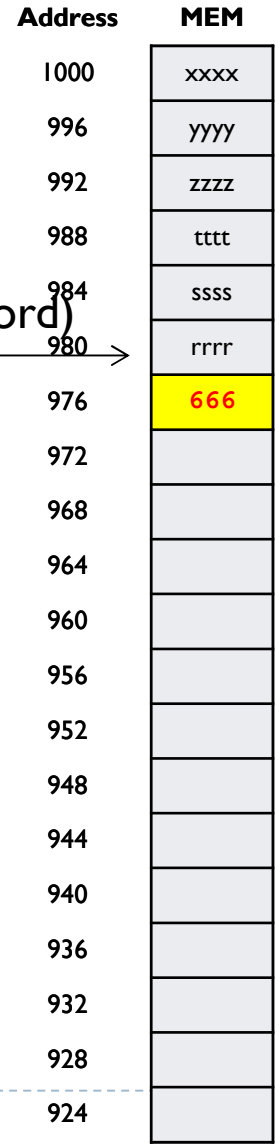
```
subi    $sp, $sp, 4
sw      $ra, 0($sp)
```




**Come recuperare un elemento (pop):**

- si legge l'elemento dalla posizione `0($sp)`
- si incrementa lo `$sp` della quantità allocata in precedenza

```
lw      $ra, 0($sp)
addi    $sp, $sp, 4
```



# Uso dello stack in una funzione

**All'inizio della funzione:**

```
funzione:

# corpo della funzione
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare

```
funzione:
        subi    $sp, $sp, 12

        # corpo della funzione
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare

- **salvare** su stack i registri, ad offset multipli di 4 rispetto a `$sp`

```
funzione:
        subi    $sp, $sp, 12
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        sw      $a1, 8($sp)


        # corpo della funzione
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare

- **salvare** su stack i registri, ad offset multipli di 4 rispetto a `$sp`

**NOTA:** conviene allocare tutto lo spazio assieme per avere **offset che restano costanti** durante tutta l'esecuzione della funzione

```
funzione:
        subi    $sp, $sp, 12
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        sw      $a1, 8($sp)


        # corpo della funzione
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare

- **salvare** su stack i registri, ad offset multipli di 4 rispetto a `$sp`

**NOTA:** conviene allocare tutto lo spazio assieme per avere **offset che restano costanti** durante tutta l'esecuzione della funzione

**All'uscita della funzione:**

- **ripristinare** da stack i registri salvati, agli stessi offset usati precedentemente

```
funzione:
        subi    $sp, $sp, 12
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        sw      $a1, 8($sp)

        # corpo della funzione

        lw      $ra, 0($sp)
        lw      $a0, 4($sp)
        lw      $a1, 8($sp)
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare
- **salvare** su stack i registri, ad offset multipli di 4 rispetto a `$sp`

**NOTA:** conviene allocare tutto lo spazio assieme per avere **offset che restano costanti** durante tutta l'esecuzione della funzione

**All'uscita della funzione:**

- **ripristinare** da stack i registri salvati, agli stessi offset usati precedentemente
- **disallocare** da stack lo stesso spazio allocato in precedenza

```
funzione:
        subi    $sp, $sp, 12
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        sw      $a1, 8($sp)

        # corpo della funzione

        lw      $ra, 0($sp)
        lw      $a0, 4($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 12
```

# Uso dello stack in una funzione

**All'inizio della funzione:**

- **allocare** su stack abbastanza word da contenere i registri da preservare
- **salvare** su stack i registri, ad offset multipli di 4 rispetto a `$sp`

**NOTA:** conviene allocare tutto lo spazio assieme per avere **offset che restano costanti** durante tutta l'esecuzione della funzione

**All'uscita della funzione:**

- **ripristinare** da stack i registri salvati, agli stessi offset usati precedentemente
- **disallocare** da stack lo stesso spazio allocato in precedenza
- **tornare** alla funzione chiamante

```
funzione:
        subi    $sp, $sp, 12
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)
        sw      $a1, 8($sp)

        # corpo della funzione

        lw      $ra, 0($sp)
        lw      $a0, 4($sp)
        lw      $a1, 8($sp)
        addi    $sp, $sp, 12
        jr      $ra
```

## Record di attivazione

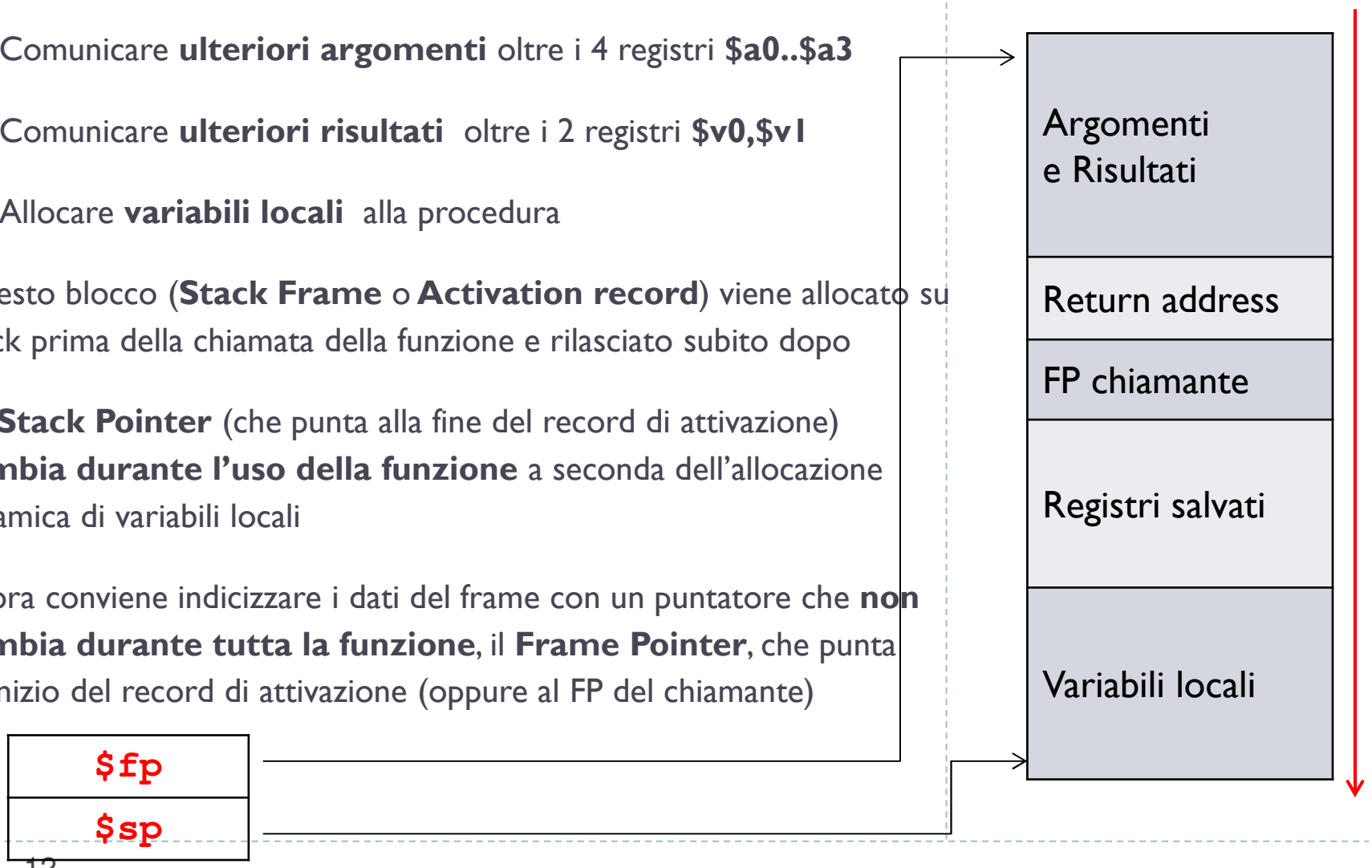
La stack è usata anche per:

- Comunicare **ulteriori argomenti** oltre i 4 registri **$a0..$a3**
- Comunicare **ulteriori risultati** oltre i 2 registri **$v0,$v1**
- Allocare **variabili locali** alla procedura

Questo blocco (**Stack Frame** o **Activation record**) viene allocato su stack prima della chiamata della funzione e rilasciato subito dopo

Lo **Stack Pointer** (che punta alla fine del record di attivazione) **cambia durante l'uso della funzione** a seconda dell'allocazione dinamica di variabili locali

Allora conviene indicizzare i dati del frame con un puntatore che **non cambia durante tutta la funzione**, il **Frame Pointer**, che punta all'inizio del record di attivazione (oppure al FP del chiamante)



# Schema di una chiamata