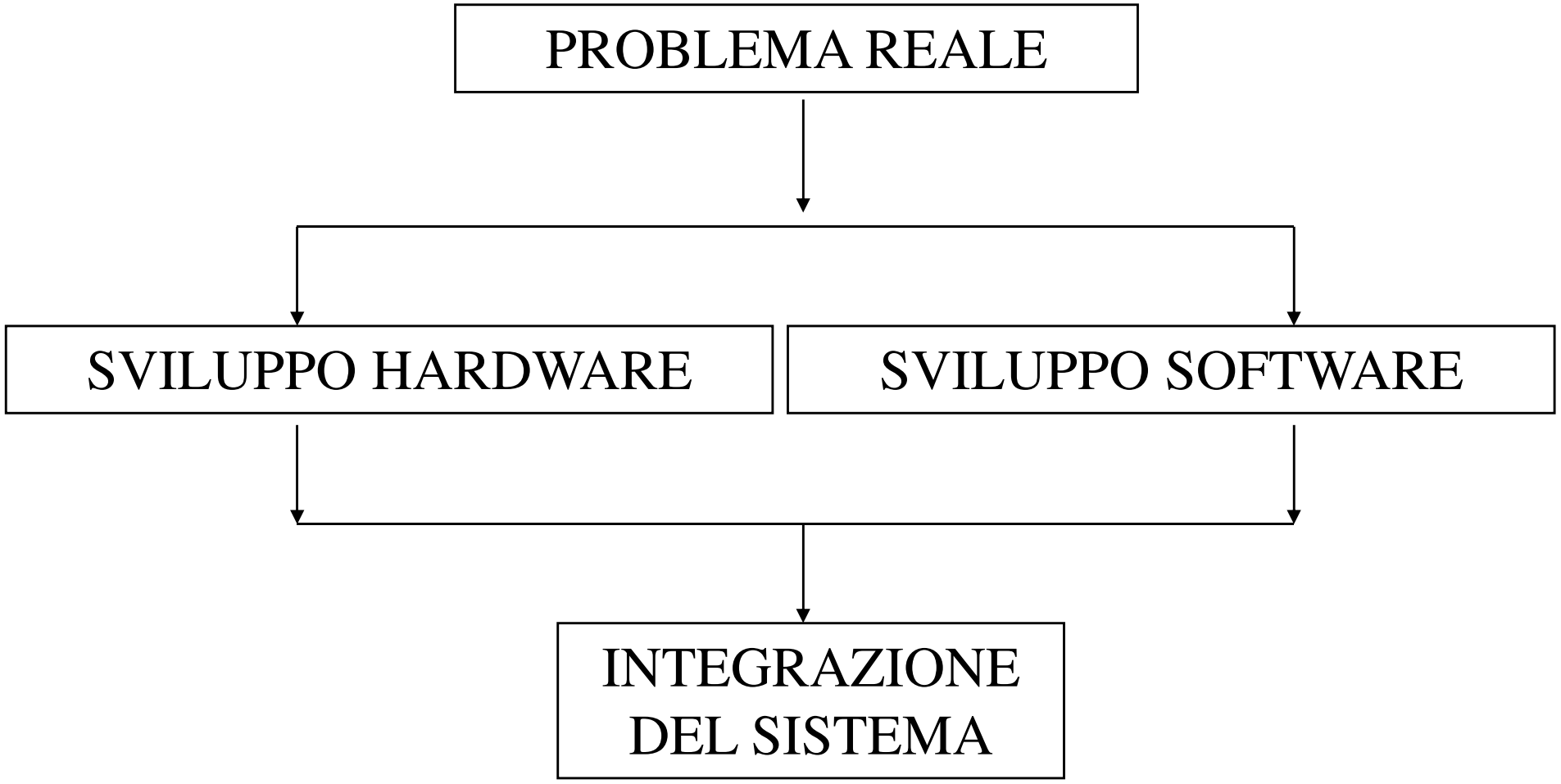
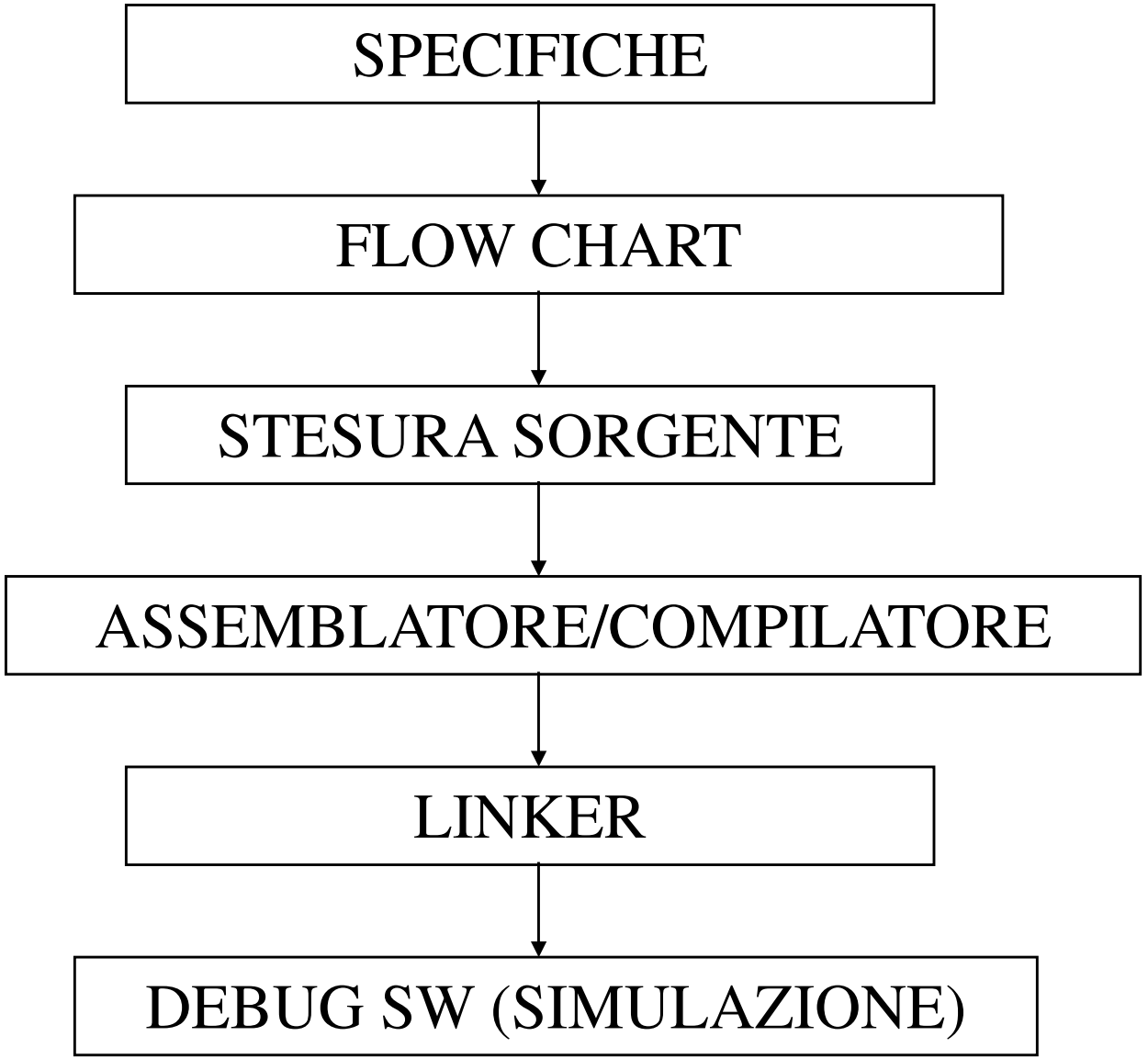
# Assemblatori, linker ed il simulatore SPIM

# Progetto di un sistema basato su microprocessore

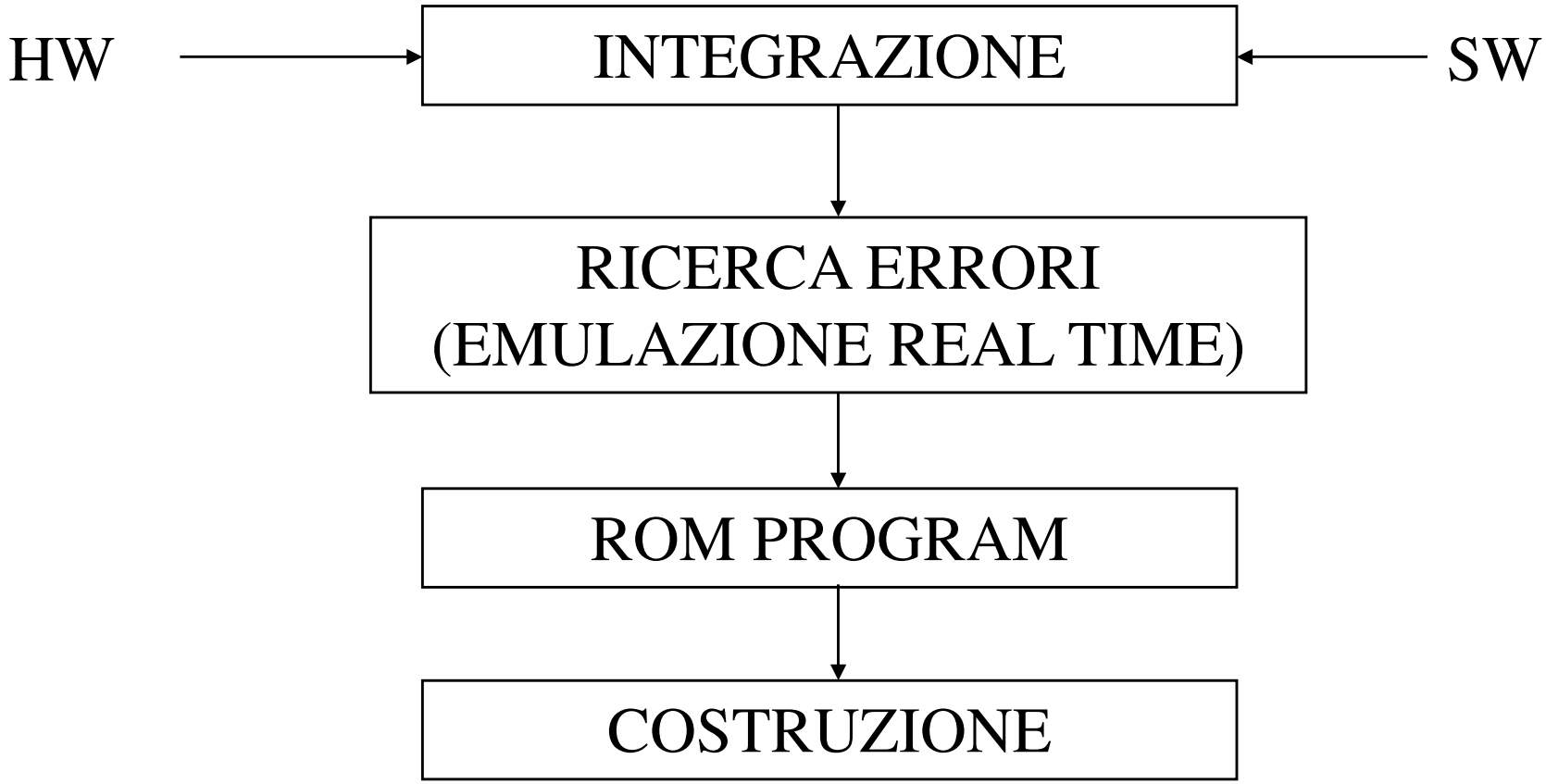


# Sviluppo hardware

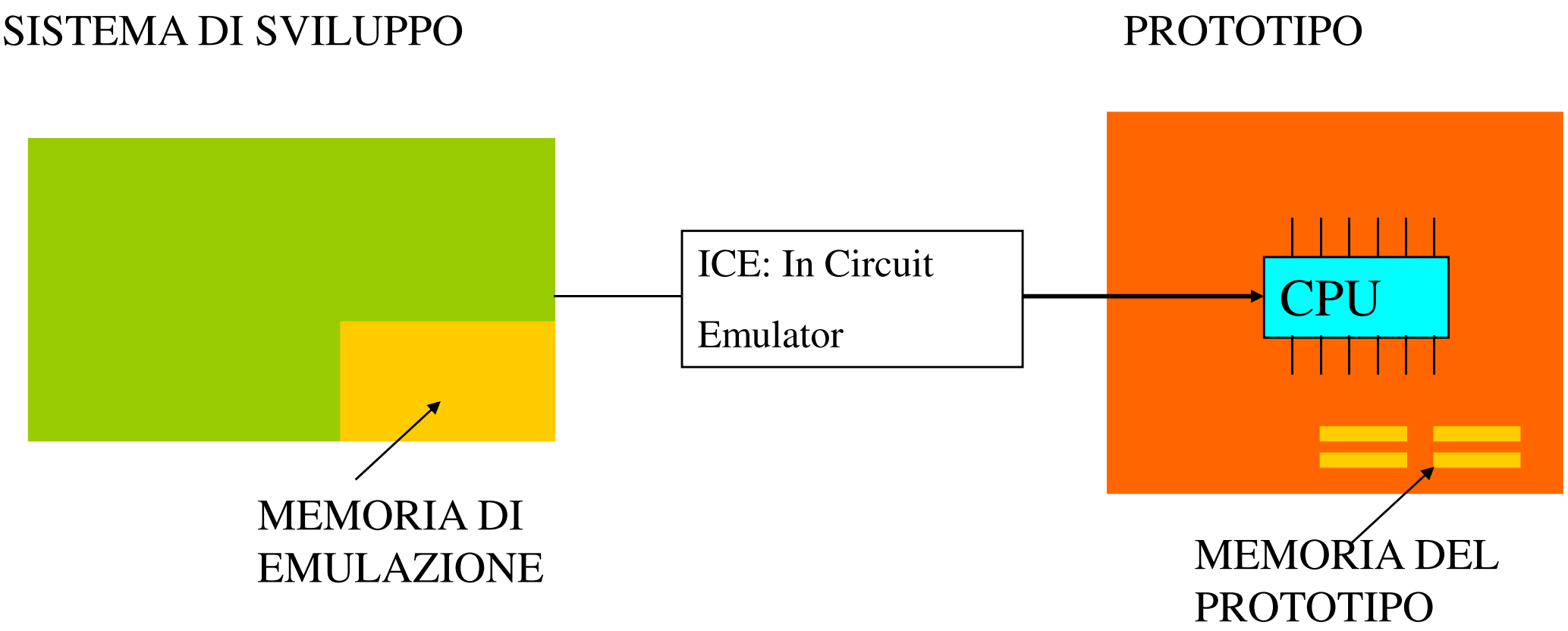
# Sviluppo software



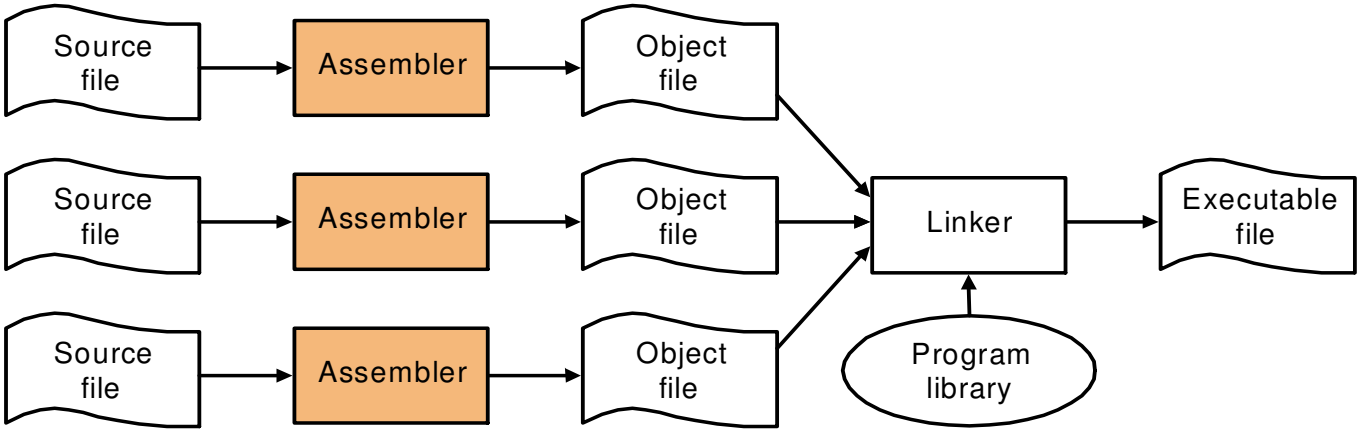
# Integrazione hardware/software



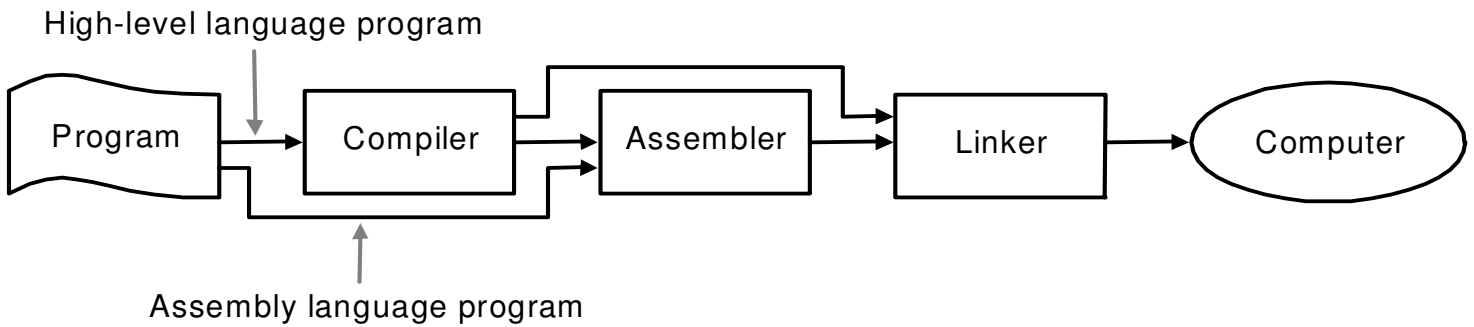
# Emulazione in tempo reale



# Creazione di un eseguibile



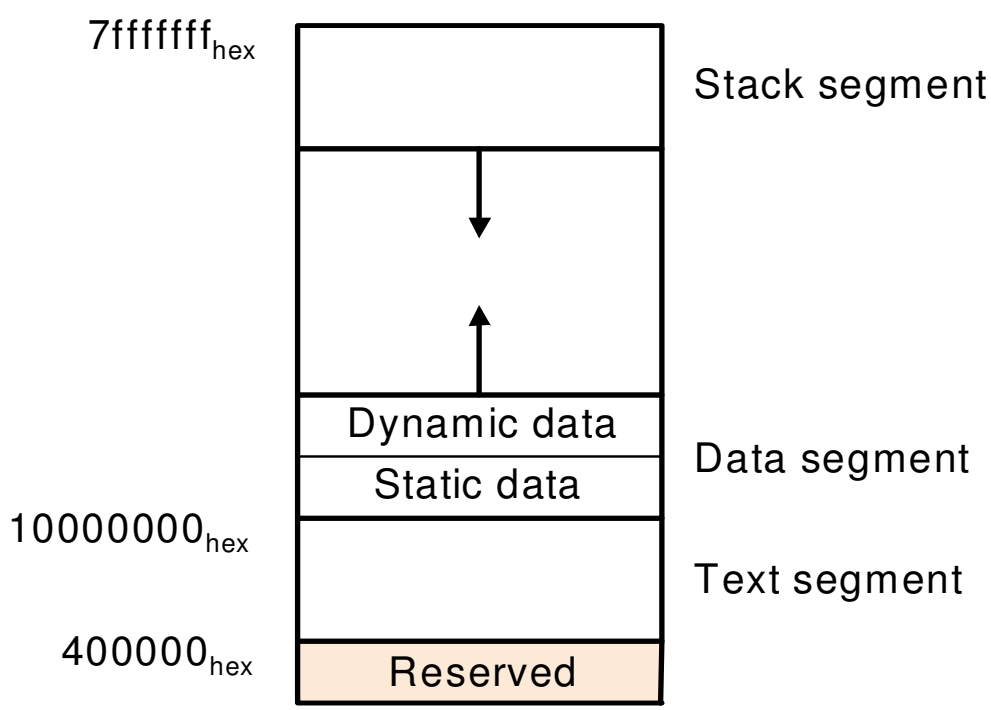

- Utilizzo di linguaggio assemblativo quando occupazione di memoria e velocità sono critici (sistemi embedded e applicazioni real time)
- Approccio ibrido con parti scritte in linguaggio ad alto livello e parti critiche scritte in linguaggio assemblativo
- Svantaggi: lunghezza, leggibilità, portabilità



# Memoria

Suddivisione in 3 segmenti:

- *testo*: istruzioni di programma a partire da 0x400000
- dati a partire da 0x10000000 (statici e dinamici)
- *stack*: a partire da 0x7fffffff (si espande in senso opposto all'area dati)



# Memoria

Load and store instructions cannot directly reference data objects with their 16-bit offset fields

Example:
Load the word in the data segment at address 10008000 hex into register $v0

```
lui     $s0,0x1000
lw      $v0,0x8000($s0)
```

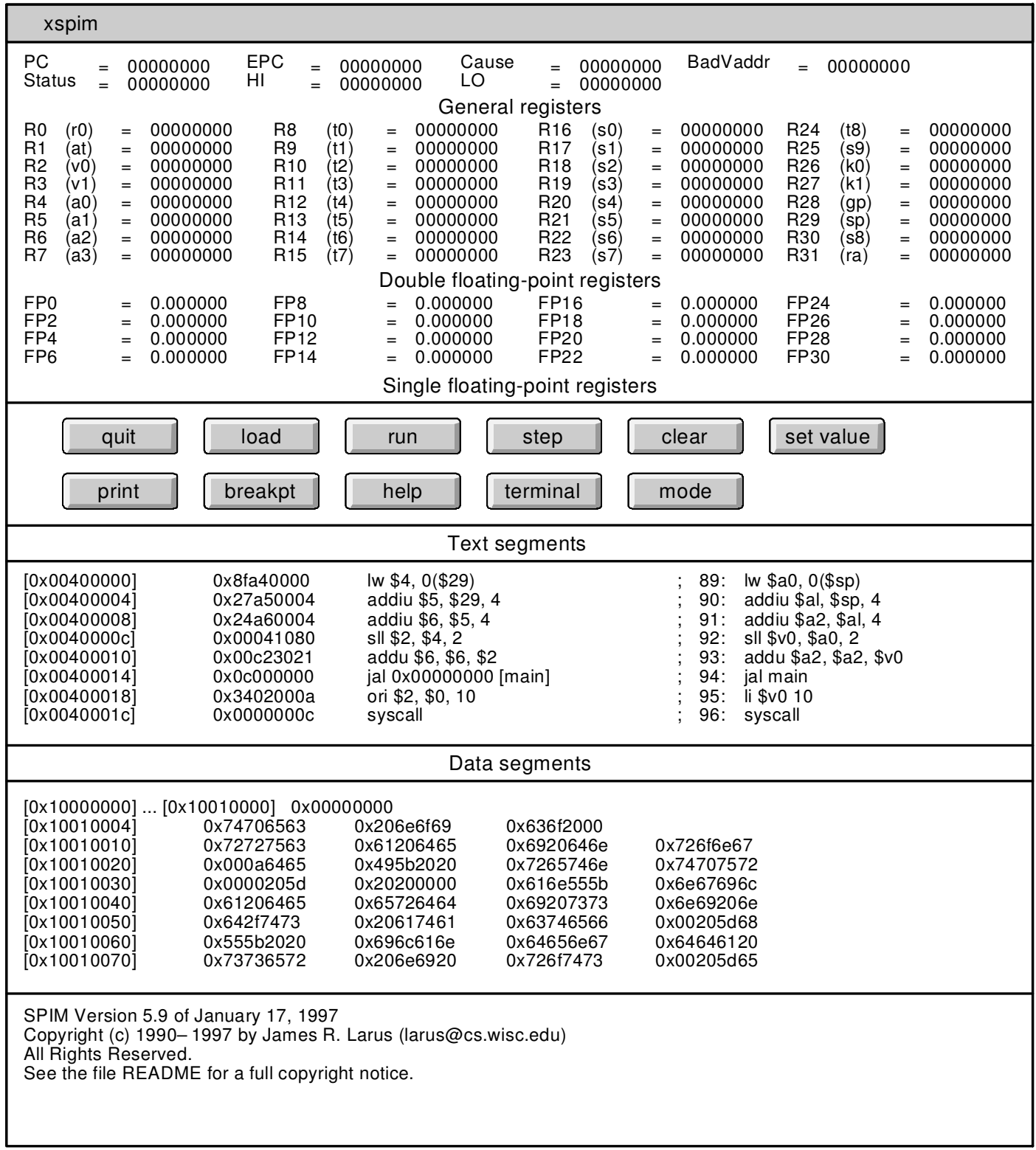
or

```
lw      $v0,0($gp)
```

$gp *global pointer* to the static data segment. This register contains address 10008000 hex

# Spim

- Simulatore: software che esegue programmi scritti in linguaggio assemblativo MIPS (debugging)
- Pseudoistruzioni che vengono tradotte in sequenze di istruzioni MIPS
- Interfaccia Windows/Unix
- Display dei registri
- Pulsanti di controllo (Unix)
- Segmento testo: istruzioni
- Segmenti dati e Stack
- Messaggi




Register display

Control buttons

Text segments

Data and stack segments

SPIM messages

# Le direttive dell'assemblatore

| | |
|---|---|
| `.align n` | Align the next datum on a $2^n$ byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive. |
| `.ascii str` | Store the string *`str`* in memory, but do not null-terminate it. |
| `.asciiz str` | Store the string *`str`* in memory and null-terminate it. |
| `.byte b1,..., bn` | Store the *`n`* values in successive bytes of memory. |

# Le direttive dell'assemblatore

| | |
|---|---|
| `.data <addr>` | Subsequent items are stored in the data segment. If the optional argument *`addr`* is present, subsequent items are stored starting at address *`addr`*. |
| `.double d1,..., dn` | Store the *`n`* floating-point double precision numbers in successive memory locations. |
| `.extern sym size` | Declare that the datum stored at *`sym`* is *`size`* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register $gp. |
| `.float f1,..., fn` | Store the *`n`* floating-point single precision numbers in successive memory locations. |
| `.globl sym` | Declare that label *`sym`* is global and can be referenced from other files. |

# Le direttive dell'assemblatore

| | |
|---|---|
| `.half h1,..., hn` | Store the *n* 16-bit quantities in successive memory halfwords. |
| `.kdata <addr>` | Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*. |
| `.ktext <addr>` | Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*. |

# Le direttive dell'assemblatore

| | |
|---|---|
| `.set noat` and `.set at` | The first directive prevents SPIM from complaining about subsequent instructions that use register $at. The second directive reenables the warning. Since pseudoinstructions expand into code that uses register $at, programmers must be very careful about leaving values in this register. |
| `.space n` | Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM). |

# Le direttive dell'assemblatore

| | |
|---|---|
| `.text <addr>` | Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive). If the optional argument *`addr`* is present, subsequent items are stored starting at address *`addr`*. |
| `.word w1,..., wn` | Store the *`n`* 32-bit quantities in successive memory words. |

# Esercizio 1

Commentare il seguente codice MIPS, assumendo che `$a0` sia usato per l'ingresso dei dati e che inizialmente contenga `n` (intero positivo). Si assuma inoltre che `$v0` sia usato per l'uscita.

```
inizio:     addi    $t0, $zero, 0
            addi    $t1, $zero, 1
ciclo:      slt     $t2, $a0, $t1
            bne     $t2, $zero, fine
            add     $t0, $t0, $t1
            addi    $t1, $t1, 2
            j       ciclo
fine:       add     $v0, $t0, $zero
stop:       j       stop
```

# Esercizio 2

Il seguente frammento di codice lavora su un vettore e genera due valori rilevanti nei registri `$v0` e `$v1`. Si assuma che il vettore sia lungo 5000 parole, con l'indice che varia da 0 a 4999, che il suo indirizzo di base si trovi in `$a0` e la sua dimensione (5000) in `$a1`. Commentare il codice mettendo in evidenza che cosa viene restituito in `$v0` e `$v1`.

```
        add     $a1, $a1, $a1
        add     $a1, $a1, $a1
        add     $v0, $zero, $zero
        add     $t0, $zero, $zero
esterno:
        add     $t4, $a0, $t0
        lw      $t4, 0($t4)
        add     $t5, $zero, $zero
        add     $t1, $zero, $zero
interno:
        add     $t3, $a0, $t1
        lw      $t3, 0($t3)
        bne     $t3, $t4, salta
        addi    $t5, $t5, 1
salta:
        addi    $t1, $t1, 4
        bne     $t1, $a1, interno
        slt     $t2, $t5, $v0
        bne     $t2, $zero, avanti
        add     $v0, $t5, $zero
        add     $v1, $t4, $zero
avanti:
        addi    $t0, $t0, 4
        bne     $t0, $a1, esterno
```

# Esercizio 3

Il seguente programma cerca di copiare dati di tipo parola dall'indirizzo memorizzato in `$a0` all'indirizzo che si trova in `$a1`, scrivendo il numero di parole copiate nel registro `$v0`. Il programma termina quando viene trovata una parola che ha valore 0. Non è necessario preservare il contenuto dei registri `$v1`, `$a0` e `$a1`. L'ultima parola deve essere copiata, ma non contata. Ci sono molti errori in questo programma MIPS: correggerli e scrivere il programma nella versione corretta.

```
                .data
                .word   0x12345678, 0x87654321, 0x12348765, 0x56781234
                .word   0x13572468, 0x21436587, 0x31425364, 0x0
                .text
                lui     $a0,0x1001
                lui     $a1,0x1000
ciclo:          lw      $v1, 0($a0)
                addi    $v0, $v0, 1
                sw      $v1, 0($a1)
                addi    $a0, $a0, 1
                addi    $a1, $a1, 1
                bne     $v1, $zero, ciclo
```

# Esercizio 4

Programma per il test del funzionamento di un'area di memoria Ram: l'area è costituita da 64k word a partire dall'indirizzo 0x10000000. La memoria viene completamente scritta e riletta effettuando un confronto fra i dati scritto e letto. In ogni cella si memorizza un dato uguale all'indirizzo della cella. `$t0` usato come puntatore alla memoria e come dato da memorizzare/controllare, `$t1` usato come indice di fine check e `$t2` usato per il dato riletto.

```
                li      $t0, 0x10000000
                lui     $t1, 0x1001
write:          sw      $t0, 0($t0)
                addi    $t0, $t0, 4
                bne     $t1, $t0, write
                lui     $t0, 0x1000
read:           lw      $t2, 0($t0)
                sub     $t3, $t0, $t2
                bne     $t3, $zero, error
                addi    $t0, $t0, 4
                bne     $t1, $t0, read
loop:           j       loop
error:          j       error
```

# Esercizio 5 (1)

Programma per la somma di 4 coppie di numeri memorizzati in codice ASCII. I risultati vengono memorizzati in un'area che segue quella in cui ci sono i dati.

```
                .data
dati:           .byte   0x33, 0x36, 0x34, 0x38
                .byte   0x30, 0x39, 0x31, 0x33
# oppure avrei potuto scrivere:
# dati:         .ascii  "36480913"
                .text
        li      $s0, 0x10010000
        addi    $s3, $s0, 8
 ciclo: lb      $a0, 0($s0)
        jal     check
        bne     $v0, $zero, error
        move    $s1, $a0
        andi    $s1, $s1, 0xf
        lb      $a0, 1($s0)
        jal     check
        bne     $v0, $zero, error
        move    $s2, $a0
        andi    $s2, $s2, 0xf
                add     $s1, $s1, $s2
        slti    $t0, $s1, 0xa
        beq     $t0, $zero, aggBCD
 ascii: move    $s2, $s1
        andi    $s1, $s1, 0xf
        ori     $s1, $s1, 0x30
        sb      $s1, 8($s0)
        srl     $s1, $s2, 4
        ori     $s1, $s1, 0x30
        sb      $s1, 9($s0)
        addi    $s0, $s0, 2
        bne     $s0, $s3, ciclo
fineOK: j       fineOK
```

# Esercizio 5 (2)

```
aggBCD: addi    $s1, $s1, 6
        j       ascii

error:  j       error

check:  add     $v0, $zero, $zero
        andi    $t0, $a0, 0x7f
        slti    $t1, $t0, 0x30
        bne     $t1, $zero, errore
        slti    $t1, $t0, 0x3a
        beq     $t1, $zero, errore
        j       $ra

errore: ori     $v0, $v0, 1
        j       $ra
```

# Esercizio 6 (1)

Un sistema dotato di un opportuno rivelatore viene investito da corpi dotati di varia energia (un corpo alla volta). Il sistema è in grado di rivelare l'energia associata ai corpi incidenti su di esso: in particolare si supponga che un'opportuna routine abbia acquisito la misura dell'energia e memorizzata la stessa come un numero da 8 bit in valore assoluto. L'intera fase di acquisizione abbia riempito un'area di memoria ampia al massimo 4096 byte, di cui la prima word rappresenta il numero dei corpi di cui e' stata rivelata l'energia.

Scrivere un programma in grado di discriminare fra 32 intervalli di energia di uguale dimensione (energie rappresentate da numeri compresi fra 0 e 7 appartengono al primo range di energia, da 8 a 15 al secondo, ... , da 248 a 255 all'ultimo); il programma deve permettere di contare il numero di corpi giunti al trasduttore dotati di energie che ricadono nel medesimo intervallo. Completata l'esecuzione il programma deve essere rieseguito iterativamente.

# Esercizio 6 (2)

```
        .data
        .word   20
        .byte   0xff, 0x14, 0x11, 0x78, 0x12
        .byte   0xa2, 0xf8, 0xe5, 0x84, 0x9b
        .byte   0x88, 0xa3, 0x50, 0x63, 0xba
        .byte   0xd1, 0xc0, 0x31, 0x74, 0x05

        .text
inizio: li      $t4, 0x10010000
        li      $t5, 0x10011000
        li      $t1, 32
        move    $t2, $t5
azzer:  sw      $zero, 0($t2)
        addi    $t2, $t2, 4
        addi    $t1, $t1, -1
        bne     $t1, $zero, azzer
        lw      $t8, 0($t4)
        addiu   $t4, $t4, 4
ciclo:  lbu     $t0, 0($t4)
        addiu   $t4, $t4, 1
        srl     $t0, $t0, 3
        sll     $t0, $t0, 2
        add     $t2, $t5, $t0
        lw      $t3, 0($t2)
        addiu   $t3, $t3, 1
        sw      $t3, 0($t2)
        addi    $t8, $t8, -1
        bne     $t8, $zero, ciclo
        j       inizio
```

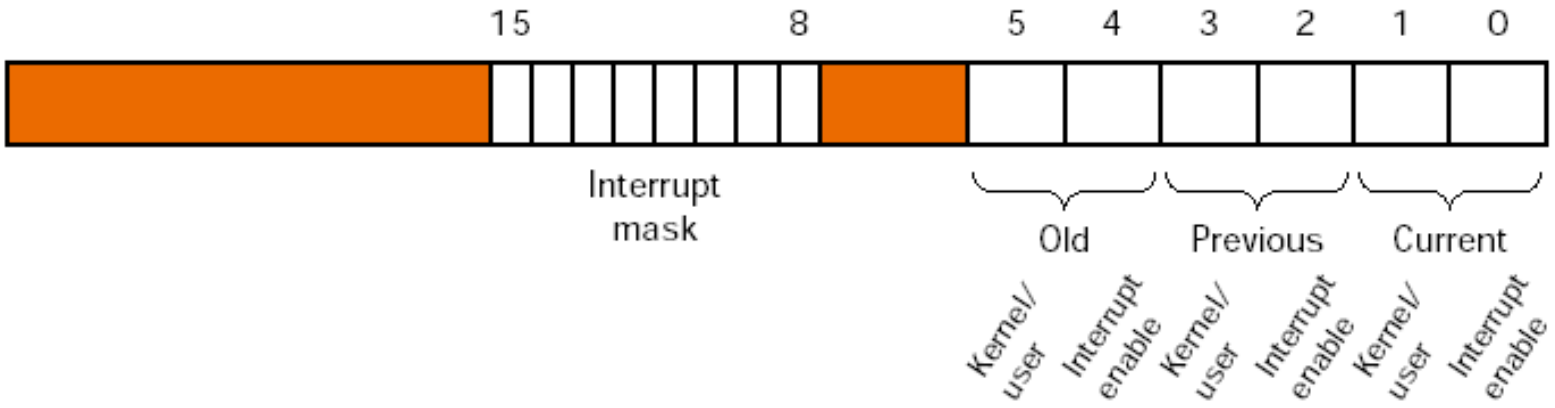
# Eccezioni e interruzioni

| Register name | Register number | Usage |
|---|---|---|
| BadVAddr | 8 | register containing the memory address at which memory reference occurred |
| Status | 12 | interrupt mask and enable bits |
| Cause | 13 | exception type and pending interrupt bits |
| EPC | 14 | register containing address of instruction that caused exception |

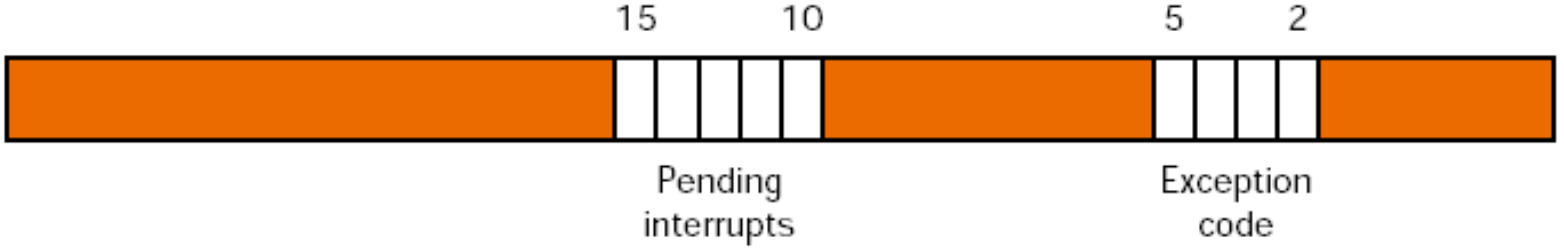
| Number | Name | Description |
|---|---|---|
| 0 | INT | external interrupt |
| 4 | ADDRL | address error exception (load or instruction fetch) |
| 5 | ADDRS | address error exception (store) |
| 6 | IBUS | bus error on instruction fetch |
| 7 | DBUS | bus error on data load or store |
| 8 | SYSCALL | syscall exception |
| 9 | BKPT | breakpoint exception |
| 10 | RI | reserved instruction exception |
| 12 | OVF | arithmetic overflow exception |

# Eccezioni e interruzioni: registri di stato e causa

- Registro di stato (status register)



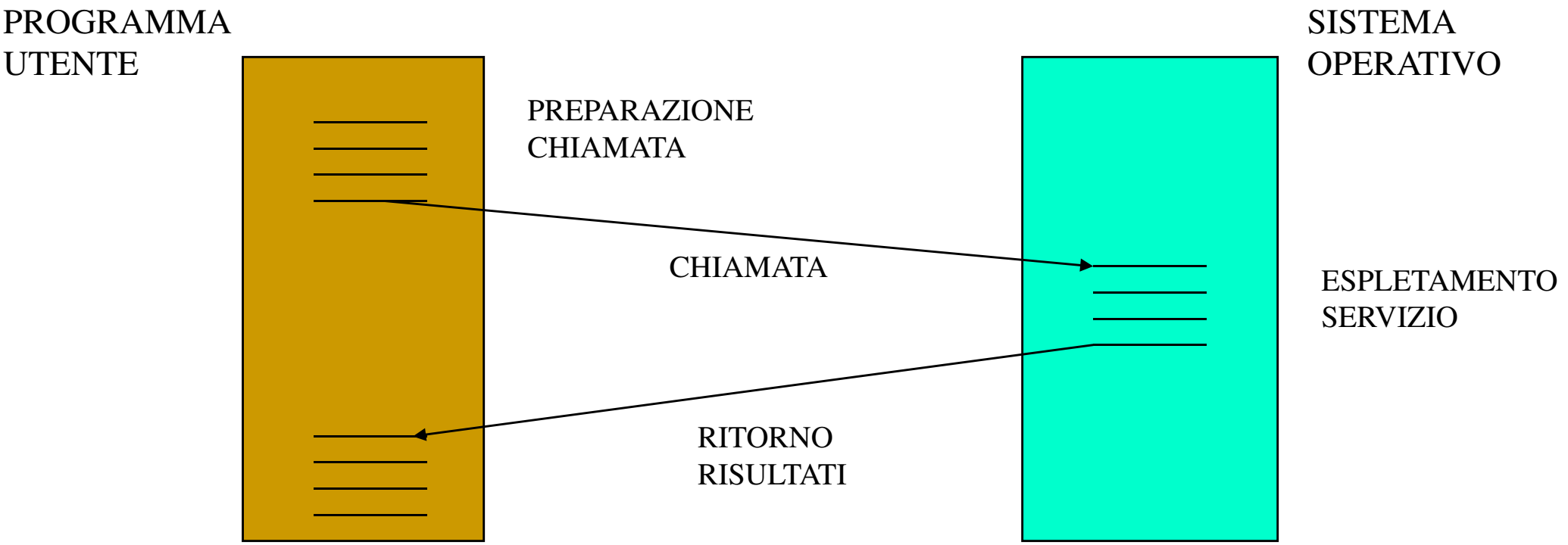

- Registro causa (cause register)



# Il gestore delle interruzioni

```
        .ktext  0x80000080
        sw      $a0,save0
        sw      $a1,save1
        mfc0    $k0,$13                 #Move Cause into $k0
        mfc0    $k1,$14                 #Move EPC into $k1
        sgt     $v0,$k0,0x44            #Ignore interrupts
        bgtz    $v0,done
        move    $a0,$k0
        move    $a1,$k1
        jal     print_excp
done:   lw      $a0,save0
        lw      $a1,save1
        addiu   $k1,$k1,4
        rfe                             #Restore interrupt state
        jr      $k1
        .kdata
save0:  .word   0
save1:  .word   0
```

# Chiamate di sistema




SPIM fornisce tramite System Call (Syscall) servizi analoghi a quelli del sistema operativo. Syscall vista come l'invocazione di una subroutine.

Passaggio parametri alla syscall attraverso $a0-$a3, se interi, o $f12, se in virgola mobile; codice della chiamata attraverso $v0. Risultati restituiti in $v0, o $f0, se in virgola mobile.

Se ci sono più parametri si usa la memoria.

# Chiamate di sistema

Caricare il codice della chiamata di sistema nel registro `$v0` e gli argomenti nei registri `$a0-$a3`. Le chiamate di sistema che forniscono un valore in uscita mettono il risultato nel registro `$v0`

```
                .data
str:            .asciiz “the answer = ”
                .text
                li      $v0,4           #system call code for
                                        #print_str
                la      $a0,str         #address of string to print
                syscall                 #print the string
                li      $v0,1           #system call code for
                                        #print_int
                li      $a0,5           #integer to print
                syscall                 #print it
```

# Chiamate di sistema

| Comando | Codice Chiamata | Input | Output |
|---|---|---|---|
| Print_int | 1 | $a0 | |
| Print_float | 2 | $f12 | |
| Print_double | 3 | $f12 | |
| Print_string | 4 | $a0 | |
| Read_int | 5 | | $v0 |
| Read_float | 6 | | $f0 |
| Read_double | 7 | | $f0 |
| Read_string | 8 | $a0=buffer<br>$a1=lungh. | |