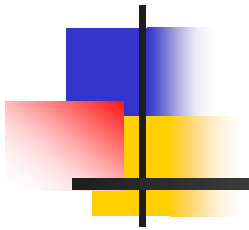


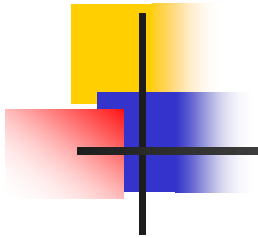
Università degli Studi di Roma "La Sapienza"

Architettura degli elaboratori II

Introduzione ai concetti ed al simulatore SPIM

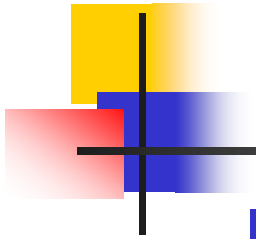
Andrea Sterbini e Franco Liberati

Tratto da dispense di Alberto Montresor, Riccardo Solmi - Univ. Di Bologna



Comunicare tra noi

- A lezione: giovedì dalle 10.30 alle 12.30
- Su **twiki.di.uniroma1.it**
  - Il simulatore SPIM, la documentazione dell'assembler e i programmi di esempio
  - Potete fare domande/risposte via web
- Per chi non ha PC: in lab. a via Salaria
  - Il mercoledì dalle 14 alle 19
- Orario di ricevimento: mercoledì 14-17



Struttura delle esercitazioni

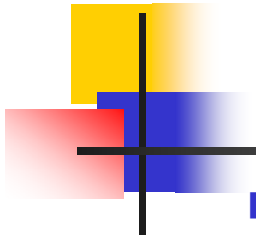
- A lezione: esercizi di assembler MIPS
  - Controllo del flusso del programma
  - Gestione di strutture dati
  - Funzioni ricorsive e gestione dello stack
  - Il gestore degli interrupt
  - Un piccolo scheduler
- 2 Esoneri scritti: a metà e fine corso
  - Piccoli esercizi di assembler (su carta)

Gli esoneri e esami

- 2 esoneri a metà e fine corso
  - Compito scritto di assembler (esercizi)
- Esame: scritto (ASM) + orale (teoria)
- Chi supera uno o più esoneri può non svolgere la parte corrispondente dell'esame scritto
- I voti vengono mantenuti "per sempre"
- Chi consegna uno scritto annulla i precedenti
- L'esame si passa solo se avete programmato veramente. CONSIGLIO: **usate SPIM**!!!!

Indice degli argomenti

- **Introduzione**
  - Assembler, compilatore, linker, programma eseguibile
- **Elementi di un programma assembly**
  - Direttive all'assembler
  - Etichette e riferimenti
  - Registri generali e speciali
  - Istruzioni e pseudoistruzioni
- **Uso del tool SPIM**
  - Un semplice simulatore del processore MIPS R2000/R3000
  - Come utilizzare SPIM



Alcuni concetti fondamentali

- **Linguaggio macchina**
  - Basato su valori numerici utilizzato dai computer per memorizzare ed eseguire programmi.
- **Linguaggio assembly**
  - Rappresentazione simbolica del linguaggio macchina, usato dai programmatori (utilizza simboli invece di numeri per rappresentare istruzioni, registri e dati).
- **Linguaggi ad alto livello**
  - Includono delle astrazioni che permettono al programmatore di non specificare certi tipi di dettagli implementativi della macchina.

Linguaggio basso/alto livello

Linguaggio C

```
/* esemp1.c */

void main()
{
    int a, b, c;
    a=4;
    b=6;
    c=a+b;
}
```

Linguaggio assembler

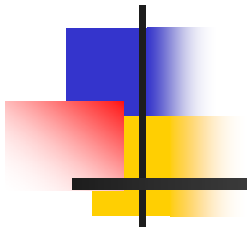
```
/* esemp1.s */

.text
lw $t1, pippo
lw $t2, paperino
add $t0,$t1,$t2

.data
pippo:     .word 4
paperino:  .word 6
```

Linguaggio macchina

```
0000000100000100
0000001000000110
0000000000001010
```



Concetti di base

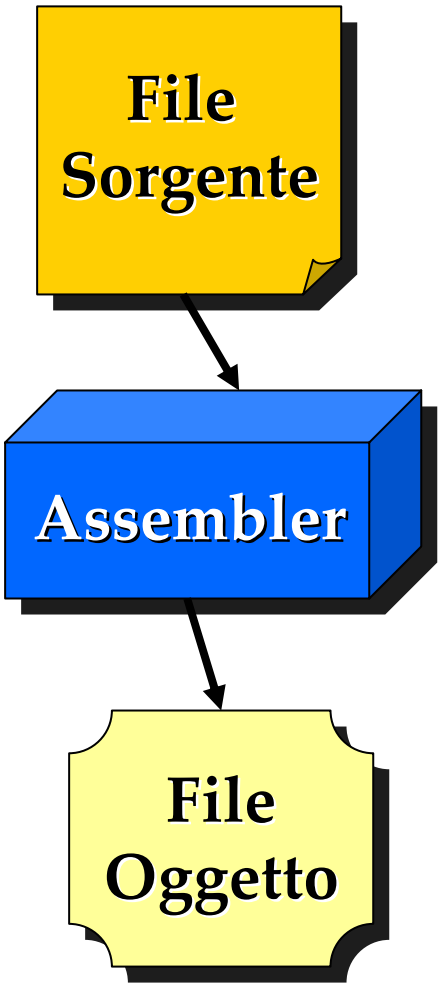
Assembler

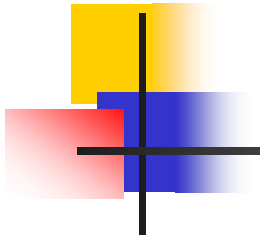
Compilatore

Linker

Assembler (assemblatore)

- L'**Assembler** traduce programmi scritti nel linguaggio assembly in linguaggio macchina.
- L'Assembler:
  - legge un ***file sorgente scritto in linguaggio assembly***
  - produce un ***file oggetto (o modulo)*** contenente linguaggio macchine ed altre informazioni necessarie per trasformare uno o più file oggetto in un programma eseguibile



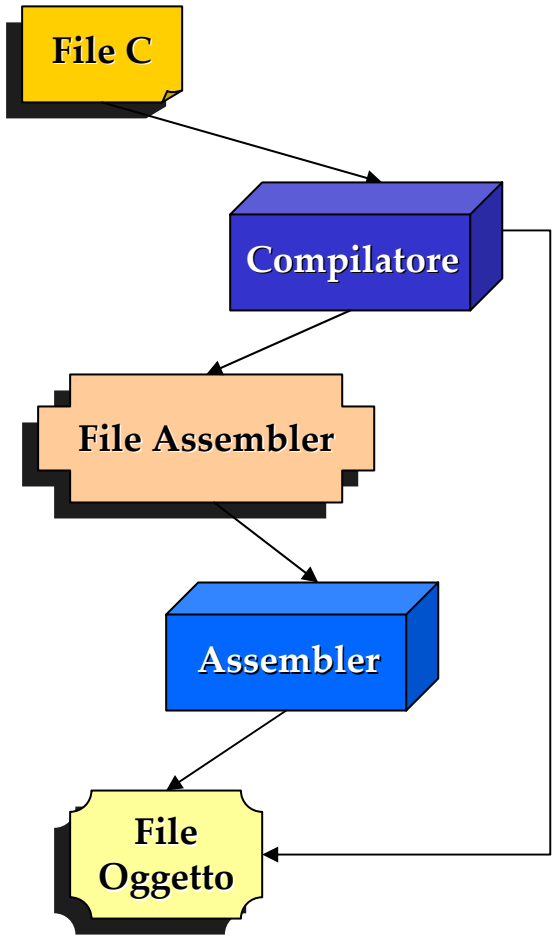


File oggetto (o modulo)

- **Un modulo può contenere**
  - Istruzioni (routine, subroutine, coroutine, ecc.)
  - Dati, variabili globali
  - Riferimenti a subroutines e dati di altri moduli (*unresolved references*)

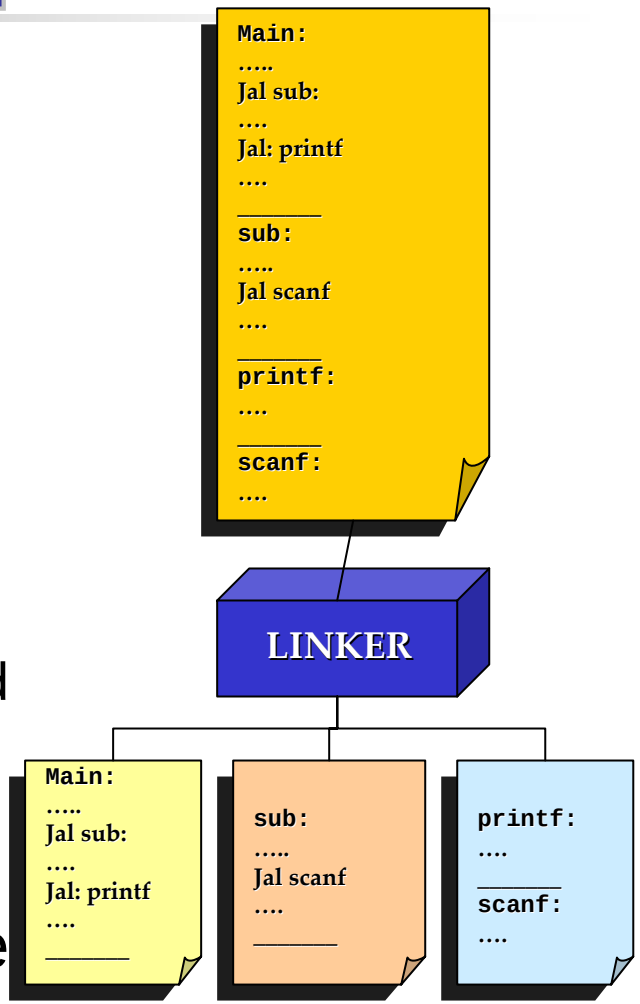
Compilatore

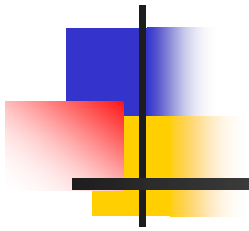
- Il **Compilatore** traduce un programma scritto in un linguaggio ad alto livello in un:
  - programma equivalente scritto in linguaggio assembly, che può essere trasformato in un file oggetto da un assembler
  - oppure, direttamente in un file oggetto



Linker

- Il **Linker** combina un *insieme di file oggetto* e *file libreria* in un *programma eseguibile*
- Il linker ha tre compiti:
  - Ricercare nei file libreria le routine di libreria utilizzate dal programma (es. *printf*)
  - Determinare le locazioni di memoria che il codice di ogni modulo andrà ad utilizzare e aggiornare i riferimenti assoluti in modo opportuno
  - Risolvere i riferimenti tra i diversi file
- Il programma eseguibile non deve contenere *unresolved references*





Assembler

Assembler (Assemblatore)

- **Compito principale di un assembler è quello di consentire una programmazione immediata**
- Utilizzo di parole mnemoniche per identificare le istruzioni del linguaggio macchina

  addiu $29, $29, -32

  piuttosto che

  00100111101111011111111111100000
- Aumento del numero di istruzioni disponibili per il programmatore attraverso l'uso di *pseudoistruzioni*
- Possibilità di definire **macro** **(non in SPIM)**
- Possibilità di definire **etichette** **(sì)**
- Possibilità di aggiungere **commenti** **(sì)**

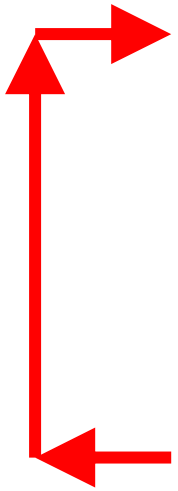
Assembler: vantaggi/svantaggi

- Realizzare sistemi real time
- Ottimizzare sezioni critiche dal punto di vista della performance di un programma
- Utilizzare *istruzioni particolari* del processore altrimenti non utilizzate dai compilatori (ad es., istruzioni MMX)
- Sviluppare parte del *kernel* di un sistema operativo, che necessita di istruzioni particolari per gestire la protezione della memoria
- Eliminare vincoli dettati dall'espressività dei costrutti di un linguaggio ad alto livello.
- Utilizzare le astrazioni dei linguaggi ad alto livello

- Complesso
- Specificare tutti i dettagli implementativi
- Illeggibilità del codice
- Scarsa gestibilità
- Programmatore è Supervisore e supergestore
- Scarsa produttività
- Non portabilità del codice
- **Massima efficienza dei compilatori moderni**

Assembler: esempio

```
.text
.globl main
main:
lw $a0, Size
li $a1, 0
li $a2, 0
li $t2, 4
loop:
mul $t1, $a1, $t2
lw $a3, Array($t1)
add $a2, $a2, $a3
add $a1, $a1, 1
beq $a1, $a0, stor
j loop
```

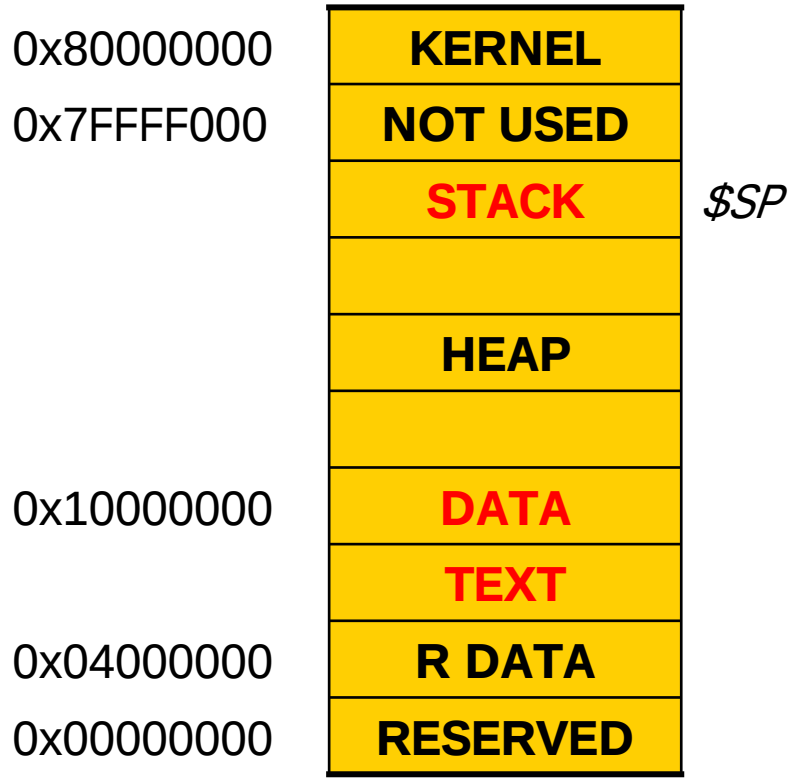


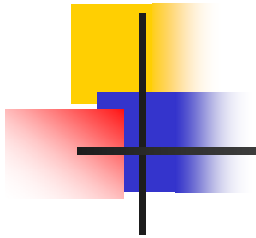
```
stor:
sw $a2, Result
…

.data
Array:  .word 1, 2, 3, 4, 5
Size:   .word 5
Result: .word 0
```

Organizzazione memoria

- La memoria viene suddivisa in segmenti
- Ogni segmento viene utilizzato per un particolare scopo
- **Segmenti principali:**
  - ***Text***: Contiene il codice dei programmi
  - ***Data***: Contiene i dati “globali” dei programmi
  - ***Stack*** Contiene i dati “locali” delle funzioni





Direttive

Direttive

- Forniscono informazioni addizionali utili all'assembler per gestire l'organizzazione del codice
- Iniziano con un *punto*

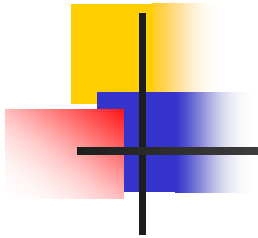
Esempi:

.text

indica che le linee successive contengono **istruzioni**

.data

indica che le linee successive contengono **dati**

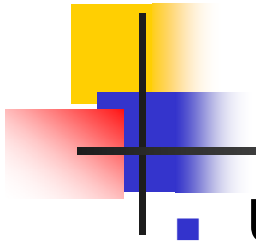


Identificatori

Identificatori

- Un identificatore è un nome associato ad una particolare posizione del programma assembly come l'indirizzo di una istruzione o di un dato.
- Un identificatore consiste in una sequenza case sensitive di caratteri alfanumerici.

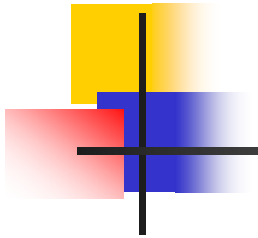
  Esempio: main, loop, stor, Size, Array, Result
- Ogni istruzione o dato si trova in un particolare indirizzo di memoria. Un identificatore ci permette di fare riferimento ad una particolare posizione senza sapere il suo indirizzo in memoria (che non ci interessa)



Etichette

- Una etichetta *introduce* un identificatore e lo associa al punto del programma in cui si trova.
- Un'etichetta consiste in un identificatore seguito dal simbolo *due punti*.

  Esempio: ***main:***, ***loop:***, ***stor:***, ***Size:***, ***Array:***, ***Result:***
- L'identificatore introdotto può avere una visibilità locale o globale. Le etichette sono locali; l'uso della direttiva *.globl* rende un'etichetta globale
- Una etichetta **locale** può essere referenziata solo dall'interno del file in cui è definita; una etichetta **globale** può essere referenziata anche da file diversi da quello in cui è definita.



Riferimenti

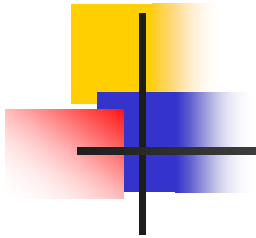
- Gli identificatori possono essere *usati* nelle istruzioni assembly e nei dati per fare riferimento alla posizione del programma associata all'identificatore.
- È sufficiente una etichetta anche per dati che occupano più byte; basta aggiungere uno *scostamento* o *offset* (calcolato in byte) al riferimento base.

  Esempio

  Array: .word 1, 2, 3, 4, 5

  # *Array* si può usare come riferimento al dato 1

  # *Array* + *4* è un riferimento al dato 2, ecc.



Allocazione di memoria per dati

Allocazione di memoria nel segmento *data*:

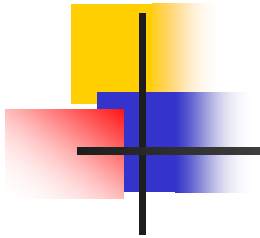
- È possibile allocare memoria nel segmento *data* utilizzando alcune direttive che permettono di specificare come la memoria sarà utilizzata e il valore iniziale della memoria stessa.
- Il valore iniziale può essere specificato tramite espressioni, costanti o stringhe.

  Esempi

  - “Hello World”
  - 0xAA+12
  - 10*10

Direttive per allocazione memoria

.byte b1 , …, bn	*Alloca n quantità a 8 bit in byte successivi in memoria*
.half h1, …, hn	*Alloca n quantità a 16 bit in halfword successive in memoria*
.word w1, …, wn	*Alloca n quantità a 32 bit in word successive in memoria*
.float f1, …, fn	*Alloca n valori floating point a singola precisione in locazioni successive in memoria*
.double d1, …, dn	*Alloca n valori floating point a doppia precisione in locazioni successive in memoria*
.asciiz str	*Alloca la stringa str in memoria, terminata con il valore 0*
.space n	*Alloca n byte, senza inizializzazione*



Registri generali

$0	**$zero**	*Valore fisso a 0*
$1	**$at**	*Riservato*
$2-$3	**$v0-$v1**	*Risultati di una funzione*
$4-$7	**$a0-$a3**	*Argomenti di una funzione*
$8-$15	**$t0-$t7**	*Temporanei (non preservati fra chiamate)*
$16-$23	**$s0-$s7**	*Temporanei (preservati fra le chiamate)*
$24-$25	**$t8-$t9**	*Temporanei (non preservati fra chiamate)*
$26-$27	**$k0-$k1**	*Riservate per OS kernel*
$28	**$gp**	*Pointer to global area*
$29	**$sp**	*Stack pointer*
$30	**$fp**	*Frame pointer*
$31	**$ra**	*Return address*

Registri speciali

PC	*Program counter*
HI	*Risultato di una moltiplicazione, parte più significativa*
LO	*Risultato di una moltiplicazione, parte meno significativa*

- **Nota**
- I registri generali possono essere utilizzati in qualunque istruzione, a scelta del programmatore (sebbene esistano delle convenzioni)
- I registri speciali *non* sono registri generali, quindi non possono essere utilizzati dalle istruzioni normali
- Esistono istruzioni speciali per gestire i registri speciali:
  - Istruzioni “Branch” e “Jump” per il PC
  - Istruzioni **mthi**, **mtlo**, **mfhi**, **mflo** per LO ed HI

Istruzioni e pseudo istruzioni

Istruzioni

- Un'istruzione inizia con una parola riservata (keyword) e continua a seconda della sua sintassi
- Ad ogni istruzione del linguaggio macchina MIPS corrisponde un'istruzione del linguaggio assembly

Esempio:

bne reg1, reg2, address (branch if not equal)

Pseudoistruzioni:

- Una pseudoistruzione è una istruzione fornita dall'assembler ma non implementata in hardware

Esempio:

blt reg1, reg2, address (branch if less than)

diventa:

slt $at, reg1, reg2 (set less than)

bne $at, $zero, address (branch if not equal)

SPIM

SPIM

- SPIM è un simulatore che esegue programmi per le architetture RISC R2000/R3000 (PowerPC = Mac)
- SPIM può leggere ed assemblare programmi scritti in linguaggio assembly MIPS
- SPIM contiene inoltre un debugger per poter analizzare il funzionamento dei programmi prodotti passo passo

Riferimento per il download di SPIM

http://www.cs.wisc.edu/~larus/spim.html

… e il solito http://twiki.di.uniroma1.it

SPIM: installazione

SPIM installazione Windows

- Scaricare *pcspim.zip* (archivio di installazione)
- Scompattare e fare doppio clic su *Setup*
- Seguite le indicazioni …

SPIM Installazione Unix

- Scaricare *spim.tar.gz*
- Scompattare (*tar zxvf spim.tar.gz*)
- Spostarsi nella directory spim-6.5
- Digitare *make install* per compilare spim e xspim

Installazioni più complesse possono essere realizzate seguendo le indicazioni nel file Readme

SPIM esecuzione Windows

- Eseguire **PCSpim.exe** nel menu programmi

SPIM esecuzione Unix/Linux

- Digitare **xspim** (avendo cura di fare in modo che il file xspim sia nel vostro path)

SPIM: interfaccia

PCSpim

File Simulator Window Help

```
 PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                 General Registers
R0   (r0) = 00000000  R8   (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1   (at) = 00000000  R9   (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2   (v0) = 00000000  R10  (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3   (v1) = 00000000  R11  (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4   (a0) = 00000000  R12  (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000


        KERNEL



        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
[0x90000000]...[0x90010000]     0x00000000

SPIM Version Version 7.1 of January 2, 2005
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
```

For Help, press F1 PC=0x00000000 EPC=0x00000000 Cause=0x00000000

SPIM: comandi principali

- **Comando LOAD (Unix) e File – Open (Windows)** 

Carica un file scritto in assembly (estensione .s, .asm) e ne assembla il contenuto in memoria

- **Comando RUN (Unix) e Simulator – Go (Windows)** 

Esegue il programma, fino alla terminazione o fino all'incontro di un breakpoint

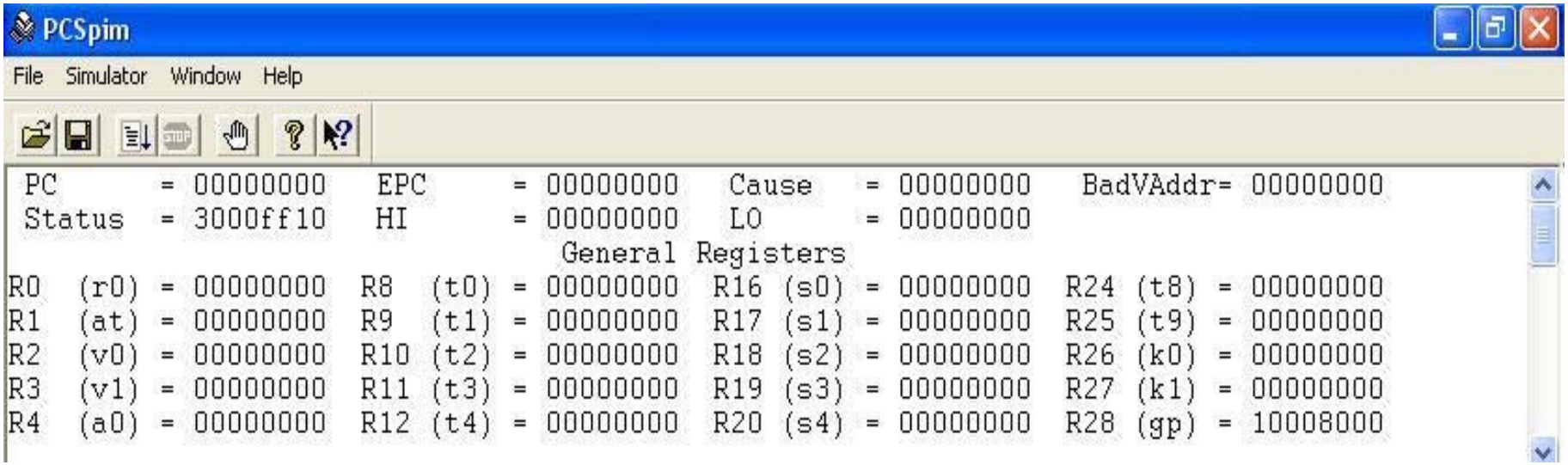
- **Comando STEP (Unix) e Simulator – Step (Windows)** 

Esegue il programma passo-passo, ovvero una istruzione alla volta

Questa modalità permette di studiare nel dettaglio il funzionamento del programma

SPIM interfaccia:sezione registri

- Mostra il valore di tutti i registri della CPU e della FPU MIPS
- Notare che i registri generali vengono identificati sia dal numero progressivo (**R28**) che dall'identificatore mnemonico (**$gp**)
- Il valore dei registri viene aggiornato ogni qualvolta il vostro programma viene interrotto



SPIM interfaccia: sezione codice

- Mostra le istruzioni dei programmi e del codice di sistema che viene caricato automaticamente alla partenza di SPIM

```
[0x00400000]    0x3c011001  lui $1, 4097 [pippo]          ; 2: lw $t1,pippo
[0x00400004]    0x8c290000  lw $9, 0($1) [pippo]
[0x00400008]    0x3c011001  lui $1, 4097 [paperino]       ; 3: lw $t2,paperino
[0x0040000c]    0x8c2a0004  lw $10, 4($1) [paperino]
[0x00400010]    0x012a4020  add $8, $9, $10               ; 5: add $t0,$t1,$t2
```

Descrizione degli elementi

1. Indirizzo esadecimale dell'istruzione
2. Codifica numerica esadecimale dell'istruzione in linguaggio macchina
3. Descrizione mnemonica dell'istruzione in linguaggio macchina
4. Linea effettiva presente nel file assembly che si sta eseguendo

Nota: Ad alcune istruzioni assembly (pseudoistruzioni) corrispondono più istruzioni in linguaggio macchina

SPIM interfaccia: sezione dati

Segmenti data e stack

- Mostra il contenuto del segmento dati e stack della memoria del programma
- I valori contenuti nella memoria vengono aggiornati ogni qualvolta il vostro programma viene interrotto

```
        DATA
[0x10000000]...[0x10010000]     0x00000000
[0x10010000]                    0x00000004  0x00000006  0x00000000  0x00000000
[0x10010010]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000
```

SPIM interfaccia: messaggi

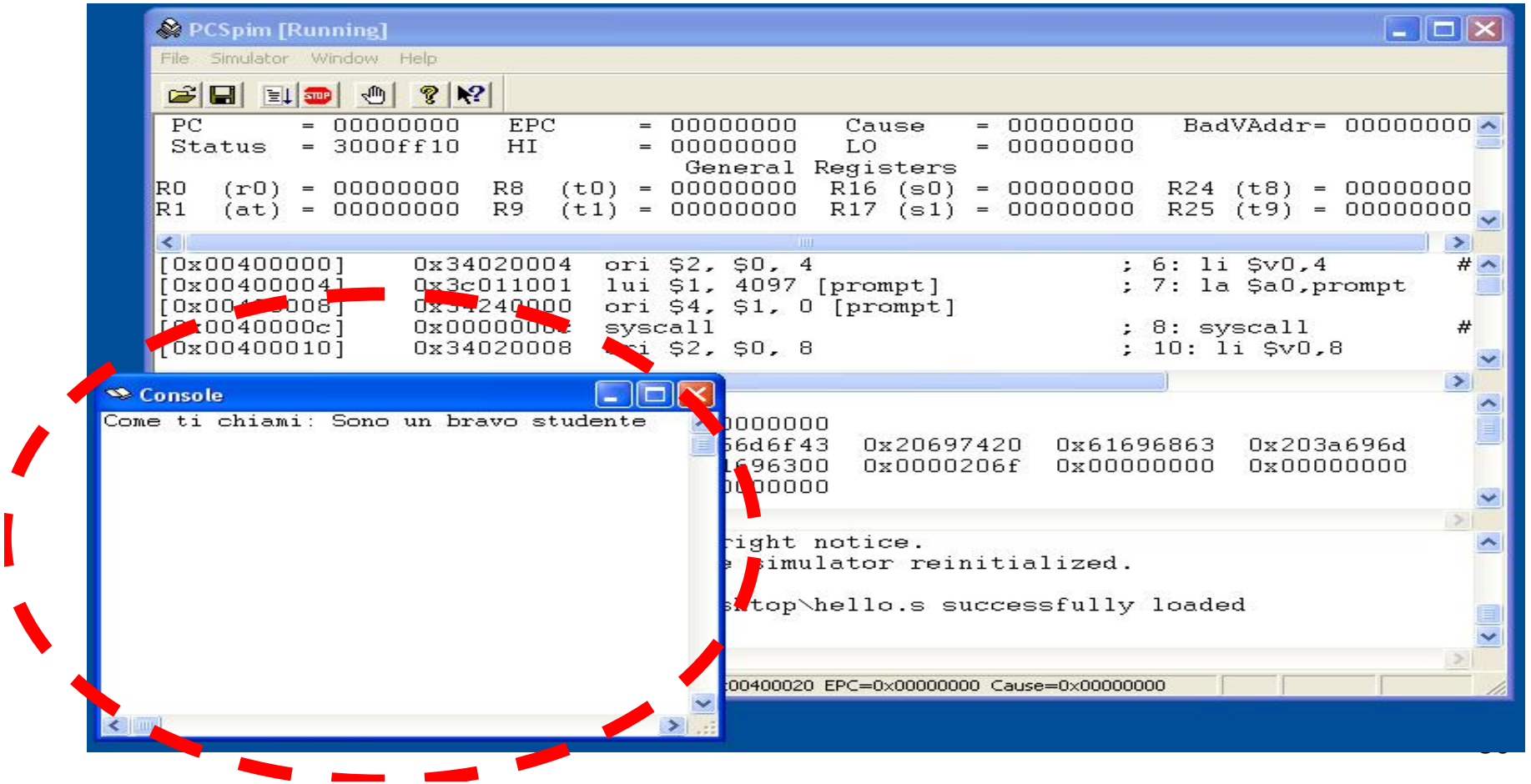
- Utilizzata da SPIM per mostrare messaggi, come ad esempio messaggi di errore o di corretto caricamento del file ed esecuzione

```
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Batman\Desktop\somma.s successfully loaded
```

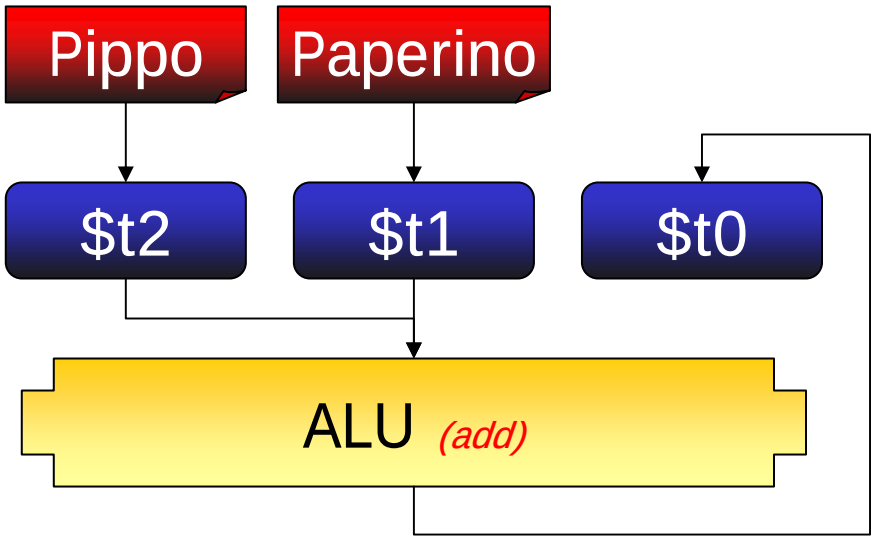
SPIM interfaccia: console

- Shell in cui vengono visualizzati i messaggi stampati dal programma



SPIM: un primo programma

Scrivere, utilizzando un qualsiasi editor (es. Notepad), ed eseguire il seguente esercizio ***SOMMA DI DUE NUMERI***:




```
.text
lw  $t1,pippo
lw  $t2,paperino
add $t0,$t1,$t2

.data

pippo:      .word 4
paperino:   .word 6
```