

# Assembler MIPS R2000/R3000

## Introduzione ai concetti e al simulatore SPIM

Riccardo Solmi

### Indice degli argomenti

---

- **Introduzione ai concetti di**
  - assembler, compilatore, linker, programma eseguibile
- **Elementi di un programma assembly**
  - Direttive all'assembler
  - Identificatori, 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

## Bibliografia

---

- *MIPS – Assembly Language Programmer’s Guide*
  - Manuale di programmazione assembly MIPS
- *Assemblers, Linkers, and the SPIM Simulator*
  - Contiene un tutorial per imparare ad usare il simulatore SPIM
- *SPIM – A MIPS R2000/R3000 Simulator*
  - Assemblatore, simulatore e debugger MIPS
- **Trovate tutto nella pagina web relativa a questo modulo:**
  - [http://www.cs.unibo.it/~solmi/teaching/arch\\_2003-2004.html](http://www.cs.unibo.it/~solmi/teaching/arch_2003-2004.html)

## Ricevimento

---

- *Sul newsgroup*
  - Avete a disposizione un newsgroup: [unibo.cs.informatica.architettura](mailto:unibo.cs.informatica.architettura)
  - Utilizzatelo il più possibile; se nessuno risponde in due-tre giorni, risponderà uno dei docenti
  - Valuteremo anche la partecipazione al newsgroup
- *Subito dopo le lezioni*
  - sono a vostra disposizione per chiarimenti
  - in aula e alla lavagna, in modo da rispondere a tutti
- *Su appuntamento in laboratorio dottorandi*
  - Scrivete a [solmi@cs.unibo.it](mailto:solmi@cs.unibo.it)
  - Fissiamo un appuntamento

## Alcuni concetti fondamentali

- **Linguaggio macchina**
  - Il linguaggio basato su valori numerici utilizzato dai computer per memorizzare ed eseguire programmi.
- **Linguaggio assembly**
  - Rappresentazione simbolica (quasi 1 a 1) del linguaggio macchina, usato dai programmatori perché utilizza simboli invece di numeri per rappresentare istruzioni, registri e dati.
- **Linguaggi ad alto livello**
  - Tutti gli altri linguaggi di programmazione sono detti ad alto livello perché includono delle astrazioni che permettono al programmatore di non specificare certi tipi di dettagli implementativi della macchina.

## Esempio in linguaggio Macchina e Assembly

00100111101010010001111011001011	←	addiu \$29, \$29, -32
00100110010011001001100100110010	←	sw \$31, 20(\$29)
01100100110010011001001100100110	←	sw \$4, 32(\$29)
00100110010011001001100100110010	←	sw \$5, 36(\$29)
11111000101001000111001010010100	←	sw \$5, 36(\$29)
0101010101111111111000001101001	←	sw \$0, 24(\$29)
10100101010101010011111000110001	←	sw \$0, 28(\$29)
11010010100100010101001010101011	←	lw \$14, 28(\$29)
10101010100000111111000011110001	←	lw \$24, 24(\$29)
100010001111100010100011111010101	←	multu \$14, \$14
10010011111011100000111100001111	←	addiu \$8, \$14, 1
...		...

Linguaggio macchina

Assembly

## Esempio in un linguaggio ad alto livello

---

```
/* prova.c */
int main(int argc, char *argv[]) {
    int i;
    int sum=0;
    for (i=0; i <= 100; i++)
        sum = sum + i;
    printf("The sum from 0 .. 100 is %d\n", sum);
}
```

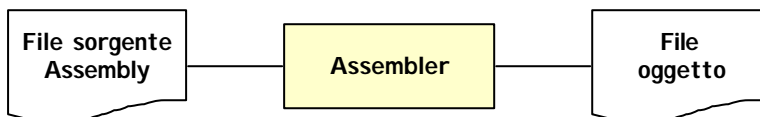
### Linguaggio C

## Assembler

---

### ▪ Assembler

- Uno strumento che traduce programmi scritti nel linguaggio assembly in linguaggio macchina. L'assembler
  - legge un *file sorgente in assembly*
  - produce un *file oggetto* contenente linguaggio macchina ed altre informazioni necessarie per trasformare uno o più file oggetto in un programma eseguibile

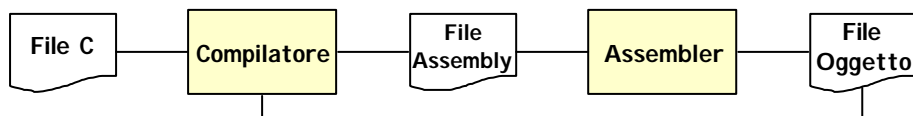


## Compilatore

---

- **Compilatore**

- Uno strumento che 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



## File oggetto (o modulo)

---

- **Un modulo può contenere**

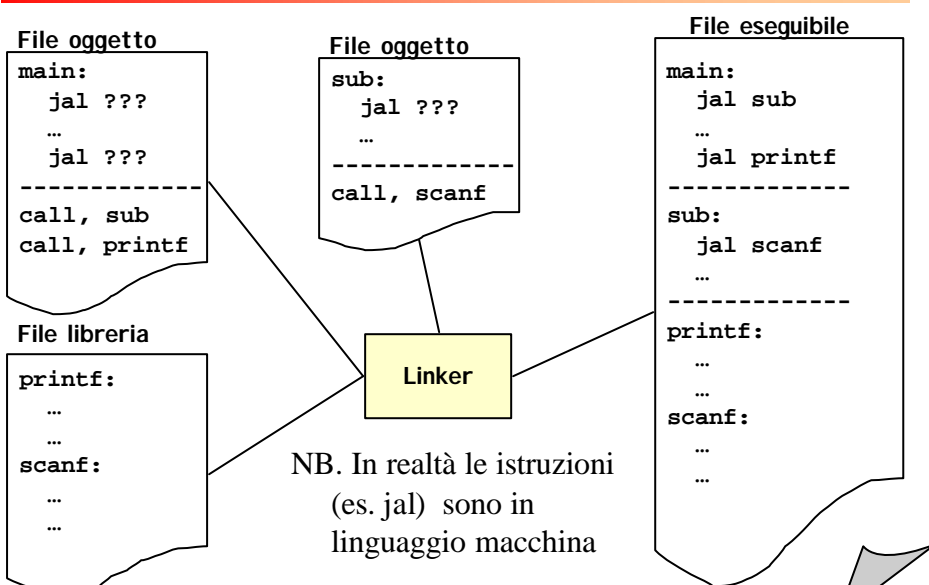
- Istruzioni (routine, subroutine, coroutine, ecc.)
- Dati
- Riferimenti a subroutine e dati di altri moduli (unresolved references)

# Linker

## ▪ Linker

- Uno strumento che 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

# Linker



## Assembler

- **Compito principale di un assembler è quello di semplificare il più possibile la vita del programmatore rispetto all'uso diretto del linguaggio macchina:**

- Utilizzo di parole mnemoniche per identificare le istruzioni del linguaggio macchina

`addiu $29, $29, -32`

vs

`001001111011110111111111111100000`

- Aumento del numero di istruzioni disponibili per il programmatore attraverso l'uso di *pseudoistruzioni*
- Possibilità di definire macro
- Possibilità di definire etichette
- Possibilità di aggiungere commenti

## Linguaggio Assembly: Pro

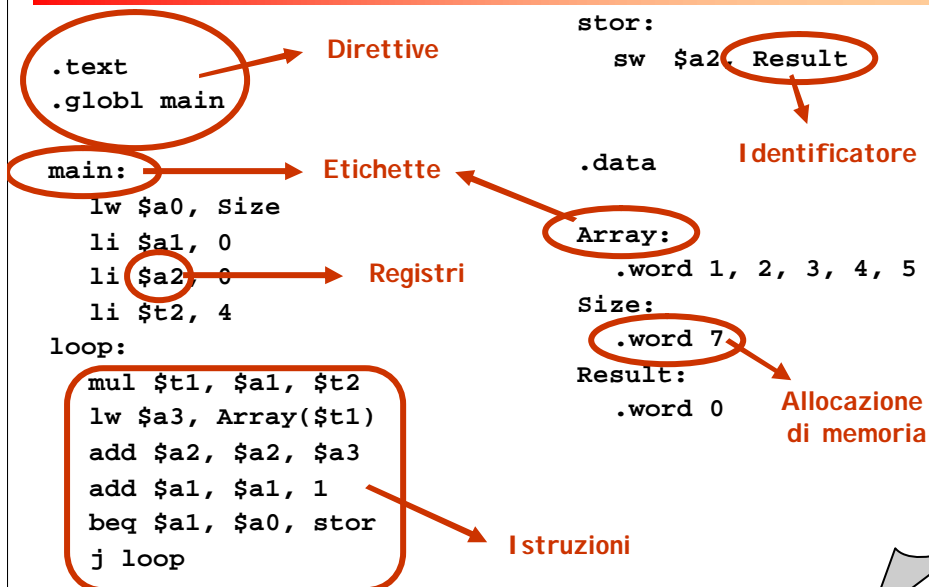
- **Perché utilizzare il linguaggio assembly?**

- Quando dimensioni e velocità del programma sono fattori critici
- Per ottimizzare sezioni critiche dal punto di vista della performance di un programma
- Per utilizzare *istruzioni particolari* del processore altrimenti non utilizzate dai compilatori (ad es., istruzioni MMX)
- Per sviluppare il *kernel* di un sistema operativo, che necessita di istruzioni particolari per gestire la protezione della memoria
- Per non essere limitati dall'espressività dei costrutti di un linguaggio ad alto livello.
- Per imparare ad usare con più consapevolezza le astrazioni dei linguaggi ad alto livello

## Linguaggio Assembly: Contro

- **Perché non utilizzare il linguaggio assembly?**
  - E' complesso
    - Devo specificare tutti i dettagli implementativi
    - Il codice prodotto è poco leggibile
  - E' error-prone
    - Salto dove voglio
    - Leggo/scrivo quello che mi pare
  - E' meno produttivo
  - Il codice prodotto non è portabile
  - I compilatori sono spesso più bravi di voi

## Elementi di un programma assembly



## Direttive

---

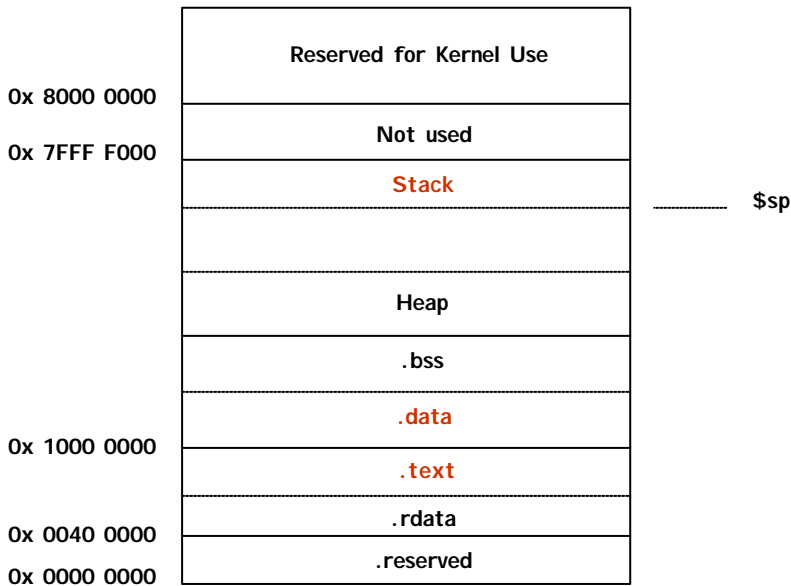
- **Direttive**
  - Forniscono informazioni aggizionali utili all'assembler per gestire l'organizzazione del codice
  - Iniziano con un *punto*
- **Esempi:**
  - *.text*  
indica che le linee successive contengono istruzioni
  - *.data*  
indica che le linee successive contengono dati

## Organizzazione della memoria

---

- **Come è organizzata la 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

## Organizzazione della memoria



© 2002-2004 Alberto Montresor, Riccardo Solmi

19

## 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. Ad 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)
- Nei prossimi due lucidi vedremo come definirli e usarli

© 2002-2004 Alberto Montresor, Riccardo Solmi

20

## Etichette e loro visibilità

---

### ▪ 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*. Ad esempio: *main:*, *loop:*, *stor:*, *Size:*, *Array:*, *Result:*
- L'identificatore introdotto può avere una visibilità locale o globale. Le etichette sono locali per default; 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

---

### ▪ Riferimenti

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

*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 i dati

- **Allocazione di memoria nel segmento data:**
  - E' possibile allocare memoria nel segmento data utilizzando alcune direttive che permettono di specificare come la memoria verrà utilizzata e il valore iniziale della memoria stessa.
  - Il valore iniziale può essere specificato tramite espressioni, costanti o stringhe. Esempi possibili possono essere:
    - "Hello World"
    - 0xAA+12
    - 10\*10

## Direttive per allocare memoria

- **Direttive di allocazione di memoria:**
  - *.byte b<sub>1</sub>, ..., b<sub>n</sub>*  
Alloca *n* quantità a 8 bit in byte successivi in memoria
  - *.half h<sub>1</sub>, ..., h<sub>n</sub>*  
Alloca *n* quantità a 16 bit in halfword successive in memoria
  - *.word w<sub>1</sub>, ..., w<sub>n</sub>*  
Alloca *n* quantità a 32 bit in word successive in memoria
  - *.float f<sub>1</sub>, ..., f<sub>n</sub>*  
Alloca *n* valori floating point a singola precisione in locazioni successive in memoria
  - *.double d<sub>1</sub>, ..., d<sub>n</sub>*  
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

### ▪ 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  \$*k*0-\$*k*1    Riservate per OS kernel
- \$28      \$gp          Pointer to global area
- \$29      \$sp          Stack pointer
- \$30      \$fp          Frame pointer
- \$31      \$ra          Return address

## Registri speciali

### ▪ 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 pseudoistruzioni

---

### ▪ 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

Es: `bne reg1, reg2, address` (branch if not equal)

### ▪ Pseudoistruzioni:

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

---

### ▪ Cos'è SPIM e cosa fa?

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

### ▪ Utilizzeremo SPIM per tutti gli esercizi in linguaggio assembly che vedremo

## Uso di un simulatore

---

- **Perché utilizzare un simulatore MIPS ?**
  - Ambiente controllato:
    - Controllo degli errori
    - Meccanismi di debug più sofisticati (potenzialmente)
  - Possibilità di utilizzare un ambiente MIPS in qualunque tipo di ambiente (uniformità):
    - A casa con Windows, MacOS o Linux su macchine x86 o PowerPC
    - In laboratorio con Linux su macchine x86
  - L'alternativa più comune (assembly x86) è molto complessa e non adatta ad un corso del primo anno

## SPIM

---

- **Dove trovare SPIM?**
  - <http://www.cs.wisc.edu/~larus/spim.html>
- **Esistono due versioni di SPIM**
  - MacOS/Linux/Unix (xspim)
  - Windows (winspim)
- **Per installare SPIM nella versione Windows**
  - Scaricate pcpim.zip (archivio di installazione)
  - Scompartate e fate doppio clic su Setup
  - Seguite le indicazioni ...
- **Per eseguire SPIM nella versione Windows**
  - Trovate la voce PCSpim nel menu programmi

# SPIM

- **Per installare SPIM nella versione Unix/Linux**
  - Scaricate `spim.tar.gz`
  - Scompattatelo (`tar zxvf spim.tar.gz`)
  - Spostatevi nella directory `spim-6.5`
  - Digitate `make install` per compilare `spim` e `xspim`
  - Installazioni più complesse possono essere realizzate seguendo le indicazioni nel file `Readme`
- **Per eseguire SPIM nella versione Unix/Linux**
  - Digitate `xspim` (avendo cura di fare in modo che il file `xspim` sia nel vostro path)

## Interfaccia di PCSpim

The screenshot displays the PCSpim simulator interface with the following sections:

- Data Segment:** Shows memory addresses and their contents. For example, `[0x10000000] ... [0x1000ffff] 0x00000000`.
- Registers:** Lists the status of various registers. The PC is at `0040000c`. General registers R0-R31 are listed with their current values, such as `R0 (r0) = 00000000`.
- Text Segment:** Shows assembly instructions and their addresses. The current instruction at `0x0040000c` is `sl1 $2, $4, 2`.

At the bottom, there is a status bar with the text: `Bare=0; Pseudo=1; Mapped=1; LoadTrap=` and a `NUM` field.

# Interfaccia di XSPIM

xspim							
PC = 00000000 EPC = 00000000 Cause = 00000000 BadVaddr = 00000000							
Status = 00000000 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 (s9) = 00000000				
R2 (v0) = 00000000	R10 (t2) = 00000000	R18 (s2) = 00000000	R26 (t0) = 00000000				
R3 (v1) = 00000000	R11 (t3) = 00000000	R19 (s3) = 00000000	R27 (t1) = 00000000				
R4 (a0) = 00000000	R12 (t4) = 00000000	R20 (s4) = 00000000	R28 (t2) = 00000000				
R5 (a1) = 00000000	R13 (t5) = 00000000	R21 (s5) = 00000000	R29 (t3) = 00000000				
R6 (a2) = 00000000	R14 (t6) = 00000000	R22 (s6) = 00000000	R30 (t4) = 00000000				
R7 (a3) = 00000000	R15 (t7) = 00000000	R23 (s7) = 00000000	R31 (ra) = 00000000				
Double floating-point registers							
FP0 = 0.000000	FP8 = 0.000000	FP16 = 0.000000	FP24 = 0.000000				
FP2 = 0.000000	FP10 = 0.000000	FP18 = 0.000000	FP26 = 0.000000				
FP4 = 0.000000	FP12 = 0.000000	FP20 = 0.000000	FP28 = 0.000000				
FP6 = 0.000000	FP14 = 0.000000	FP22 = 0.000000	FP30 = 0.000000				
Single floating-point registers							
<input type="button" value="quit"/> <input type="button" value="load"/> <input type="button" value="run"/> <input type="button" value="step"/> <input type="button" value="clear"/> <input type="button" value="set value"/>							
<input type="button" value="print"/> <input type="button" value="breakpt"/> <input type="button" value="help"/> <input type="button" value="terminal"/> <input type="button" value="mode"/>							
Text segments							
[0x00400000] 0x3fa40000 lw \$4, 0(\$29) ; 0x89: lw \$a0, 0(\$sp)	[0x00400040] 0x27a50004 addiu \$5, \$29, 4 ; 0x90: addiu \$a1, \$sp, 4						
[0x00400080] 0x24a60004 addiu \$6, \$5, 4 ; 0x91: addiu \$a2, \$a1, 4	[0x004000c0] 0x00041080 sl \$2, \$4, 2 ; 0x92: sl \$f0, \$a0, 2						
[0x00400100] 0x00c22021 addiu \$9, \$6, \$2 ; 0x93: addiu \$a2, \$a2, \$v0	[0x00400140] 0x0c000000 jal 0x00000000 [main] ; 0x94: jal main						
[0x00400180] 0x3402000a ori \$2, \$0, 10 ; 0x95: li \$f0, 10	[0x004001c] 0x0000000c syscall ; 0x96: syscall						
Data segments							
[0x10000000] ... [0x10010000] 0x00000000	[0x10010004] 0x74706563	[0x10010008] 0x206e6f69	[0x1001000c] 0x636f2000	[0x10010010] 0x72727563	[0x10010014] 0x61206465	[0x10010018] 0x6920646e	[0x1001001c] 0x726f6e67
[0x10010020] 0x000a6465	[0x10010024] 0x496b2020	[0x10010028] 0x726f746e	[0x1001002c] 0x74707572	[0x10010030] 0x0000205d	[0x10010034] 0x20200000	[0x10010038] 0x616e555b	[0x1001003c] 0x6e67696c
[0x10010040] 0x61206465	[0x10010044] 0x65726464	[0x10010048] 0x69207373	[0x1001004c] 0x6e69206e	[0x10010050] 0x642f7473	[0x10010054] 0x20617461	[0x10010058] 0x63746566	[0x1001005c] 0x00205d68
[0x10010060] 0x555b2020	[0x10010064] 0x696e616e	[0x10010068] 0x64656e67	[0x1001006c] 0x64646120	[0x10010070] 0x73736572	[0x10010074] 0x206e6920	[0x10010078] 0x726f7473	[0x1001007c] 0x00205d65

## Interfaccia utente: sezione registri

### Sezione registers:

- Mostra il valore di tutti i registri della CPU e della FPU MIPS
- Notare che i registri generali vengono identificati sia dal numero progressivo (R29) che dall'identificatore mnemonico (\$sp)
- Il valore dei registri viene aggiornato ogni qualvolta il vostro programma viene interrotto

### Pannello di controllo (in xspim) o menù (in PCSpim)

- Contiene i comandi che possono essere utilizzati su SPIM, come ad esempio run, load, etc.

## Interfaccia utente: sezione codice

---

### ▪ **Segmento text**

- Mostra le istruzioni dei vostri programmi e del codice di sistema che viene caricato automaticamente alla partenza di SPIM
- La prossima istruzione da eseguire viene evidenziata invertendo il colore della linea contenente l'istruzione

### ▪ **Le istruzioni vengono visualizzate nel seguente modo:**

```
[0x00400020] 0x3c011001 lui $1, 4097 [Size] ; 6: lw $a0, Size
[0x00400024] 0x8c240014 lw $4, 20($1) [Size]
[0x00400028] 0x34050000 ori $5, $0, 0 ; 7: li $a1, 0
```

①

②

③

④

## Visualizzazione istruzioni

---

### ▪ **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

## Interfaccia utente: sezioni dati e messaggi

---

- **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
- **Sezione messaggi SPIM**
  - Utilizzata da SPIM per mostrare messaggi, come ad esempio messaggi di errore
- **Console**
  - Shell in cui vengono visualizzati i messaggi stampati dai vostri programmi

## Interfaccia utente: 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

## Esercizio sull'uso di SPIM

- **Scrivete il seguente programma e dopo averlo caricato in SPIM provate a seguirne passo a passo il funzionamento**

```
.text                # TotaleSpese.s
.globl main
main:                la $t0, spese
loop:                lw $t1, ($t0)
                    add $t2, $t2, $t1
                    addi $t0, $t0, 4
                    bnez $t1, loop
                    sw $t2, result
                    jr $ra
.data
spese:               .word 15, 10, 8, 0
result:              .word 0
```

## Strumenti standalone per architettura MIPS

- **SPIM vs. un assembler standalone**
  - SPIM è solo uno strumento didattico; integra le funzionalità di un assembler con quelle di un simulatore.
  - In generale ho bisogno di un assembler che mi trasformi i sorgenti in un eseguibile.
- **Compilatori come strumenti didattici**
  - Un compilatore di un linguaggio ad alto livello traduce i costrutti del linguaggio in sequenze di istruzioni assembly
  - Confrontare il codice assembly prodotto con il sorgente da cui deriva è un ottimo strumento didattico.

## Compilatore gcc e assembler as

- **Tool a disposizione (in laboratorio, macchine Linux)**
  - **gcc: compilatore C/C++, ...**
    - prende in input un file C con estensione `.c`
    - normalmente:
      - ritorna un file eseguibile chiamato `a.out`
    - con opzione `-s`:
      - ritorna un file in assembly, stesso nome, ma con estensione `.s`
    - con opzione `-c`:
      - ritorna un file oggetto, stesso nome, ma con estensione `.o`
  - **as: assembler**
    - Prende in input un file in assembly con estensione `.s`
    - Ritorna un file oggetto, stesso nome, ma con estensione `.o`

## Cross-compiling

- **Nota:**
  - Le versioni di GCC installate in laboratorio producono programmi eseguibili su processori x86
  - Noi vorremmo utilizzare compilatori e assembleri per processori MIPS
- **Cross-compiling:**
  - Creazione di programmi eseguibili tramite compilatori funzionanti su architetture diverse da quella target
    - Esempio: utilizzo di PC per creare applicazioni per Palm
- **Nel laboratorio sono disponibili:**
  - `mipsel-linux-gcc`
  - `mipsel-linux-as`

## Esempio di compilazione in più fasi

```
vinsanto:~> mipsel-linux-gcc -S prova.c
vinsanto:~> cat prova.s
.file 1 "prova.c"
.abicalls
# GNU C 2.7.2 [AL 1.1, MM 40] Linux/MIPSEL compiled by GNU C
# Cc1 defaults:
# -mgas -mabicalls
# Cc1 arguments (-G value = 0, Cpu = 3000, ISA = 1):
# -quiet -dumpbase -o

gcc2_compiled.:
.align 2
.LC0:
.string "The sum from 0 .. 100 is %d\n"
.text
.align 2
.globl main
.type main,@function
.ent main
```

© 2002-2004 Alberto Montresor, Riccardo Solmi

43

## Esempio di compilazione in più fasi

```
main:
.frame $fp,48,$31
.mask 0xd0000000,-8
.fmask 0x00000000,0
.set noreorder
.cpload $25
.set reorder
subu $sp,$sp,48
.cprestore 16
sw $31,40($sp)
sw $fp,36($sp)
sw $28,32($sp)
move $fp,$sp
sw $4,48($fp)
sw $5,52($fp)
sw $0,28($fp)
sw $0,24($fp)

.L2:
.L5:
.L4:
.L3:
.L3:
jal printf
```

© 2002-2004 Alberto Montresor, Riccardo Solmi

44

## Esempio di compilazione in più fasi

```
.L1:
    move    $sp,$fp
    lw     $31,40($sp)
    lw     $fp,36($sp)
    addu   $sp,$sp,48
    j      $31
    .end    main

.Lfel:
    .size   main,.Lfel-main
```

## Esempio di compilazione in più fasi

```
vinsanto:~> mipsel-linux-as prova.s
vinsanto:~> mipsel-linux-objdump --all-headers prova.o
prova.o:      file format elf32-littlemips
prova.o
architecture: mips:3000,flags 0x00000011: HAS_RELOC, HAS_SYMS
start address 0x00000000
CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
00000000 l      .text  00000000 gcc2_compiled.
00000000 l      .rodata 00000000 .LC0
00000000 g      F .text  000000b4 main
00000000 o      *UND* 00000000 _gp_disp
00000034 l      .text  00000000 .L2
00000050 l      .text  00000000 .L5
0000007c l      .text  00000000 .L3
00000064 l      .text  00000000 .L4
00000000 o      *UND* 00000000 printf
000000a0 l      .text  00000000 .L1
000000b4 l      .text  00000000 .Lfel
```