Laboratorio di Architetture degli Elaboratori II (A.A. 2015-2016)

Matteo Re, matteo.re@unimi.it [cognomi G-Z] **(Nicola Basilico, [cognomi A-D])**

Ufficio S210, Dipartimento di Informatica, Via Comelico 39/41 - 20135 Milano (MI),

Ricevimento: su appuntamento, preferibilmente in aula a valle delle sessioni di laboratorio

Home page del corso per materiale e avvisi http://homes.di.unimi.it/re/arch2-lab-2015-2016.html

24 ore di lezione/esercitazione al computer

Esame: proposta, realizzazione e discussione di un programma in assembly

Assembly e il simulatore SPIM

UNIVERSITÀ DEGLI STUDI DI MILANO
matteo.re@unimi.it
http://homes.di.unimi.it/re/arch2-lab-2015-2016.html

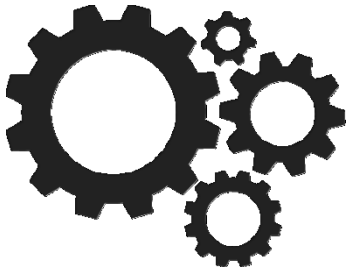
Introduzione

Linguaggio di alto livello

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

↓

compilatore



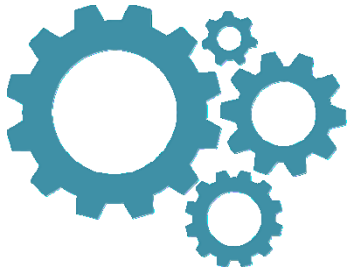
→

Assembly

```
multi $2, $5,4
add   $2, $4,$2
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
jr    $31
```

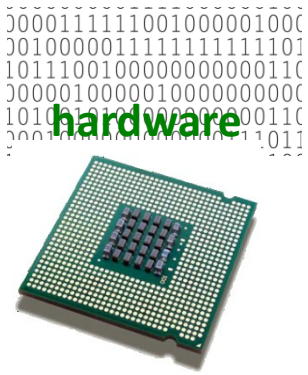
↓

Assembler + linker



→

Linguaggio macchina



Livello più basso (vicino all'hardware) dove poter programmare le istruzioni di un elaboratore

ISA RISC: MIPS (Microprocessor without Interlocked Pipeline Stages)

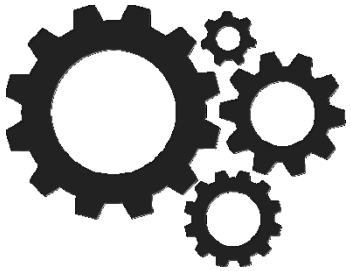
Introduzione

Linguaggio di alto livello

```
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Laboratorio di architetture II

compilatore

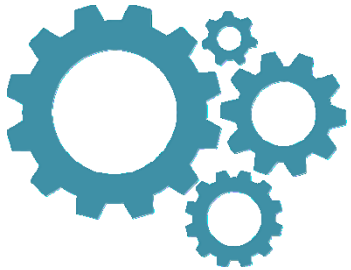


Assembly

```
multi $2, $5,4
add   $2, $4,$2
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
jr    $31
```

Livello più basso (vicino all'hardware) dove poter programmare le istruzioni di un elaboratore

Assembler + linker



Linguaggio macchina

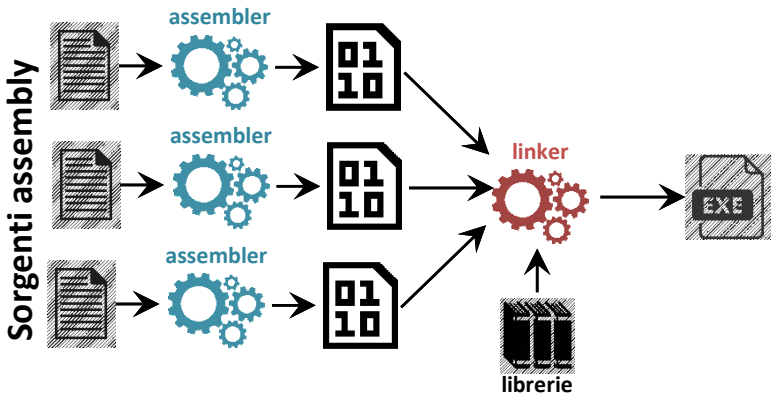
ISA RISC: MIPS (Microprocessor without Interlocked Pipeline Stages)

Assembly

- E’ la **<u>rappresentazione simbolica</u>** del linguaggio macchina di un elaboratore.

`1000110010100000` ⟷ `add A,B`

- Dà alle istruzioni una forma *human-readable* e permette di usare **label** per referenziare con un nome parole di memoria che contengono istruzioni o dati.



- Programmi coinvolti:
  - **assembler**: «traduce» le istruzioni assembly (da un **file sorgente**) nelle corrispondenti istruzioni macchina in formato binario (in un **file oggetto**);
  - **linker**: combina i files oggetto e le librerie in un **file eseguibile** dove la «destinazione» di ogni label è determinata.

Istruzioni MIPS32

ABS.D	ABS.S	ADD	ADD.D	ADD.S	ADDI	ADDIU	ADDU
AND	ANDI	BC1F	BC1FL	BC1T	BC1TL	BC2F	BC2FL
BC2T	BC2TL	BEQ	BEQL	BGEZ	BGEZAL	BGEZALL	BGEZL
BGTZ	BGTZL	BLEZ	BLEZL	BLTZ	BLTZAL	BLTZALL	BLTZL
BNE	BNEL	BREAK	C.cond.D	C.cond.S	CACHE	CEIL.W.D	CEIL.W.S
CFC1	CFC2	CLO	CLZ	COP2	CTC1	CTC2	CVT.D.S
CVT.D.W	CVT.S.D	CVT.S.W	CVT.W.D	CVT.W.S	DIV	DIV.D	DIV.S
DIVU	ERET	FLOOR.W.D	FLOOR.W.S	J	JAL	JALR	JR
LB	LBU	LDC1	LDC2	LH	LHU	LL	LUI
LW	LWC1	LWC2	LWL	LWR	MADD	MADDU	MFC0
MFC1	MFC2	MFHI	MFLO	MOV.D	MOV.S	MOVF	MOVF.D
MOVF.S	MOVN	MOVN.D	MOVN.S	MOVT	MOVT.D	MOVT.S	MOVZ
MOVZ.D	MOVZ.S	MSUB	MSUBU	MTC0	MTC1	MTC2	MTHI
MTLO	MUL	MUL.D	MUL.S	MULT	MULTU	NEG.D	NEG.S
NOR	OR	ORI	PREF	ROUND.W.D	ROUND.W.S	SB	SC
SDC1	SDC2	SH	SLL	SLLV	SLT	SLTI	SLTIU
SLTU	SQRT.D	SQRT.S	SRA	SRAV	SRL	SRLV	SSNOP
SUB	SUB.D	SUB.S	SUBU	SW	SWC1	SWC2	SWL
SWR	SYNC	SYSCALL	TEQ	TEQI	TGE	TGEI	TGEIU
TGEU	TLBP	TLBR	TLBWI	TLBWR	TLT	TLTI	TLTIU
TLTU	TNE	TNEI	TRUNC.W.D	TRUNC.W.S	WAIT	XOR	XORI

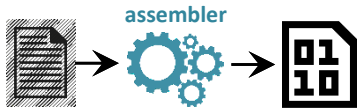
Assembly

- Il codice Assembly può essere il risultato di due processi:
  - *target language* del compilatore che traduce un programma in linguaggio di alto livello (C, Pascal, …) nell'equivalente assembly;
  - *linguaggio di programmazione* usato da un programmatore.

- Assembly è stato l'approccio principale con cui scrivere i programmi per i primi computer.

- Oggi la complessità dei programmi, l'invenzione di compilatori sempre migliori e la disponibilità crescente di memoria rendono conveniente programmare in linguaggi di alto livello.

- Assembly come linguaggio di programmazione è adatto in certi casi particolari:
  - ottimizzare le performance (anche in termini di prevedibilità) e spazio occupato da un programma (ad es., sistemi embedded);
  - eredità di certi sistemi vecchi, ma ancora in uso, dove Assembly è l'unico modo per scrivere programmi.

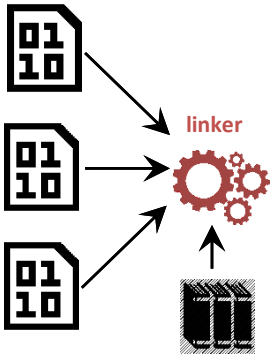
Assembler 

- Assembler traduce il sorgente Assembly in linguaggio macchina:
  1. associa ad ogni label il corrispondente indirizzo di memoria (label locali, cioè che danno il nome a oggetti referenziati solo dallo stesso file sorgente);
  2. associare ad ogni istruzione simbolica opcode e argomenti in codice binario.

- In generale, il file oggetto generato non può essere eseguito: Assembler non è in grado di risolvere le label esterne (cioè che danno il nome a oggetti che possono essere referenziati da **altri** file sorgenti).

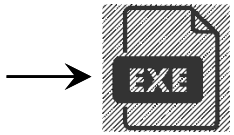
- Formato del file oggetto:
  - **segmento testo**: contiene le istruzioni;
  - **segmento dati**: contiene la rappresentazione binaria dei dati definiti nel file sorgente (ad esempio stringhe);
  - **informazione di rilocazione**: dice chi sono le istruzioni che usano indirizzi assoluti (ad esempio una chiamata a una routine esterna);
  - **tabella dei simboli:** per ogni label esterna dice quale è l'indirizzo associato ed elenca le label usate nel file che sono *unresolved;*
  - (**Informazione di debug:** informazioni riguardo al modo con cui si è svolta la compilazione.)

Linker



Il Linker combina tutti i file oggetto in un unico file che **può essere eseguito**.

- Determina quali librerie vengono usate e che quindi vanno incluse nel file eseguibile finale.
- Determina gli indirizzi di memoria a cui, nel file eseguibile, staranno le procedure e dati, «aggiusta», usando le informazioni di rilocazione, le istruzioni che fanno uso di indirizzi assoluti.
- Risolve le *unresolved* labels.



Il codice finale assemblato contiene ora in modo completo tutte le informazioni che servono per poterlo eseguire.

Fase di load

Il file eseguibile di solito risiede su una memoria di massa (o memoria secondaria), quando se ne invoca l'esecuzione deve essere caricato in memoria primaria.

Fase di load:

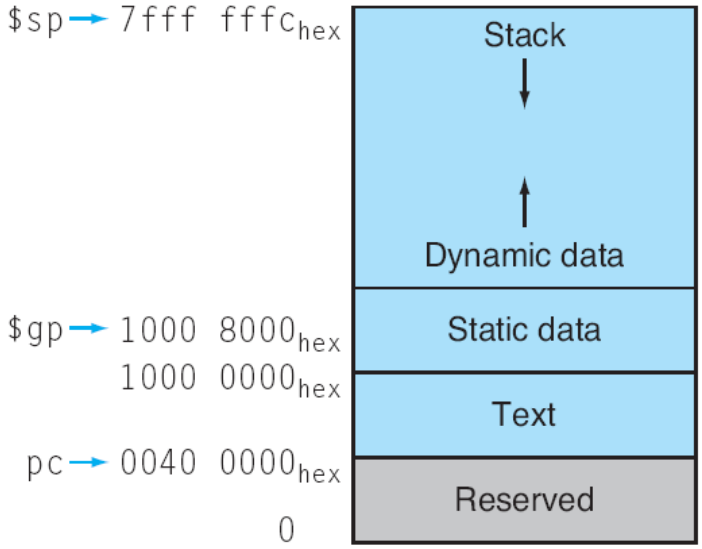
1. lettura dell'header per estrarre dimensione dei vari segmenti;
2. creazione dello spazio degli indirizzi in memoria e caricamento dei vari segmenti;
3. procedure di inizializzazione (clear dei registri, load sullo stack dei parametri, inizializzazione dello stack pointer, …);
4. chiama di una routine che invocherà a sua volta `main`.

Il programma in memoria (in MIPS)

- **Segmento testo**: contiene le **istruzioni** del programma.

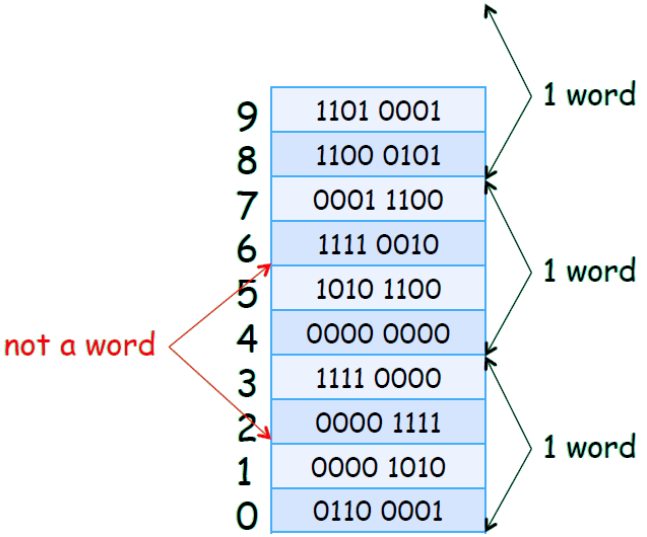
- **Segmento dati**:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici*:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*

- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) *(e.g., parametri di una procedura, valori di ritorno, etc.)*.

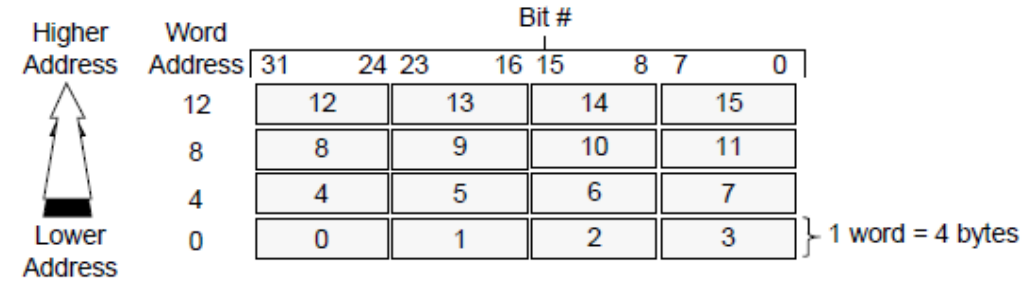


A proposito della memoria…

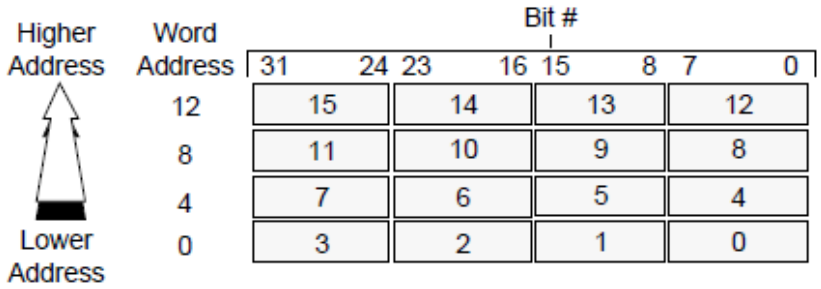
- La memoria è un array unidimensionale di byte indirizzabili individualmente. Parole di 32 bit sono allineate in blocchi da 4 byte
- 2^{32} byte con indirizzi da 0 a 2^{32}-1
- 2^{30} parole aventi indirizzi 0, 4, 8, .. , 2^{32}-4



Endianness:

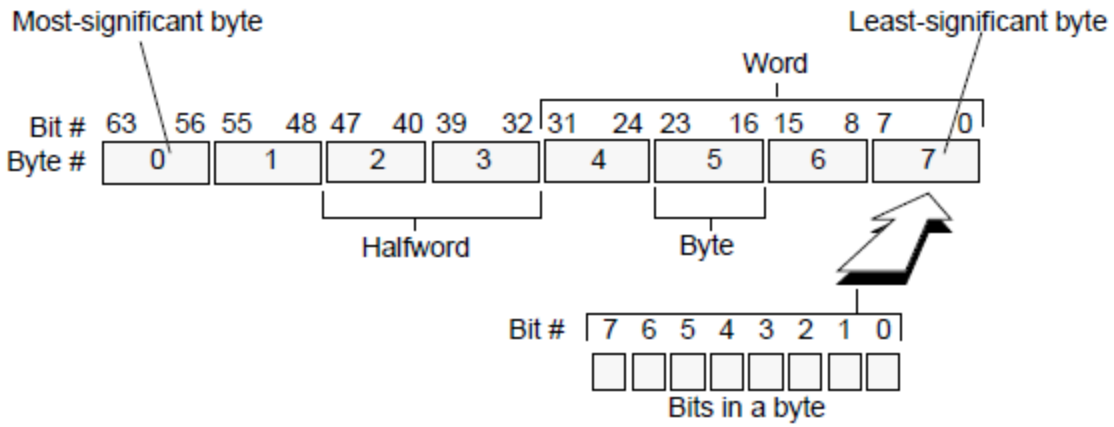


Big-Endian byte ordering

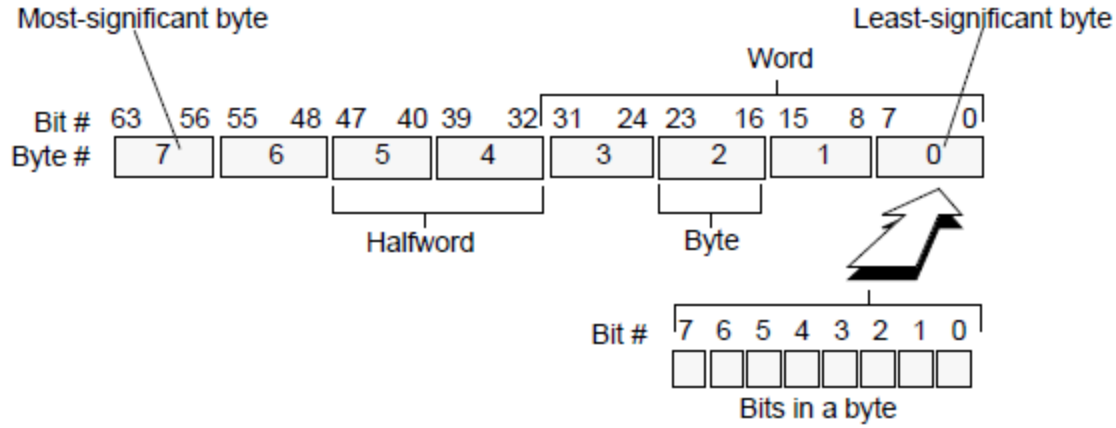


Little-Endian byte ordering

A proposito della memoria…

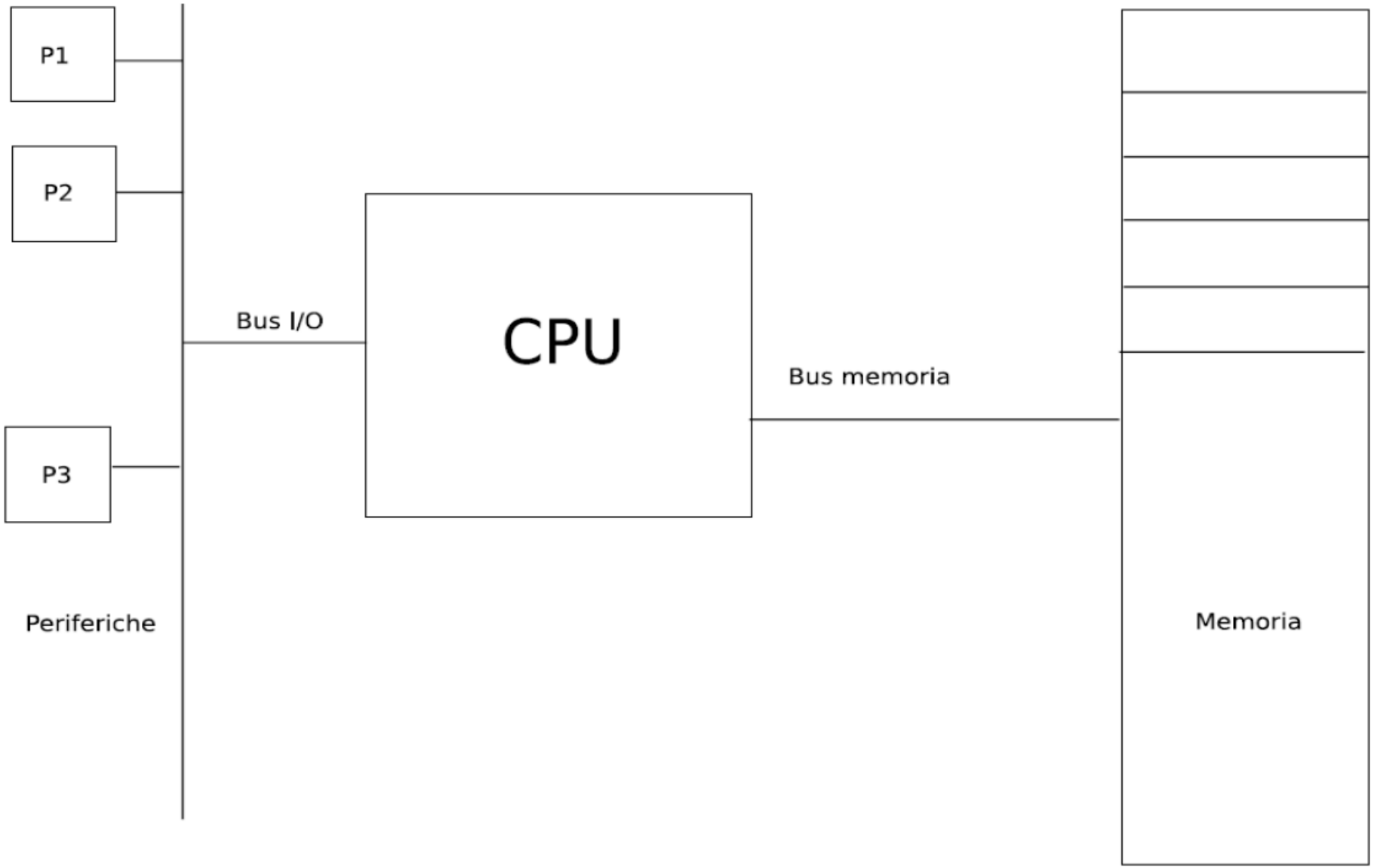


Big-Endian Data in Doubleword Format



Little-Endian Data in Doubleword Format

Macchina di von Neumann



Modello di programmazione (I)

- Definito da: formato dati CPU, Coprocessori (CP0-CP3), registri CPU, formato dati FPU, Endianness, tipi (modalità) di accesso alla memoria

CPU Data Formats

The CPU defines the following data formats:

- Bit (*b*)
- Byte (8 bits, *B*)
- Halfword (16 bits, *H*)
- Word (32 bits, *W*)
- Doubleword (64 bits, *D*)

FPU Data Formats

The FPU defines the following data formats:

- 32-bit single-precision floating point (.fmt type *S*)
- 32-bit single-precision floating point paired-single (.fmt type *PS*)
- 64-bit double-precision floating point (.fmt type *D*)
- 32-bit Word fixed point (.fmt type *W*)
- 64-bit Long fixed point (.fmt type *L*)

Coprocessors (CP0-CP3)

The MIPS Architecture defines four coprocessors (designated CP0, CP1, CP2, and CP3):

- Coprocessor 0 (**CP0**) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the *System Control Coprocessor*.
- Coprocessor 1 (**CP1**) is reserved for the floating point coprocessor, the FPU.
- Coprocessor 2 (**CP2**) is available for specific implementations.
- Coprocessor 3 (**CP3**) is reserved for the floating point unit in the MIPS64 Architecture.

CP0 translates virtual addresses into physical addresses, manages exceptions, and handles switches between kernel, supervisor, and user states. CP0 also controls the cache subsystem, as well as providing diagnostic control and error recovery facilities. The architectural features of CP0 are defined in Volume III.

Modello di programmazione (II)

- Definito da: formato dati CPU, Coprocessori (CP0-CP3), registri CPU, formato dati FPU, Endianness, tipi (modalità) di accesso alla memoria

CPU Registers

The MIPS32 Architecture defines the following CPU registers:

- 32 32-bit general purpose registers (GPRs)
- a pair of special-purpose registers to hold the results of integer multiply, divide, and multiply-accumulate operations (HI and LO)
- a special-purpose program counter (PC), which is affected only indirectly by certain instructions - it is not an architecturally-visible register.

Modello di programmazione (II)

- Definito da: formato dati CPU, Coprocessori (CP0-CP3), registri CPU, formato dati FPU, Endianness, tipi (modalità) di accesso alla memoria

CPU General-Purpose Registers

Two of the CPU general-purpose registers have assigned functions:

- *r0* is hard-wired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. *r0* can also be used as a source when a zero value is needed.
- *r31* is the destination register used by JAL, BLTZAL, BLTZALL, BGEZAL, and BGEZALL without being explicitly specified in the instruction word. Otherwise *r31* is used as a normal register.

The remaining registers are available for general-purpose use.

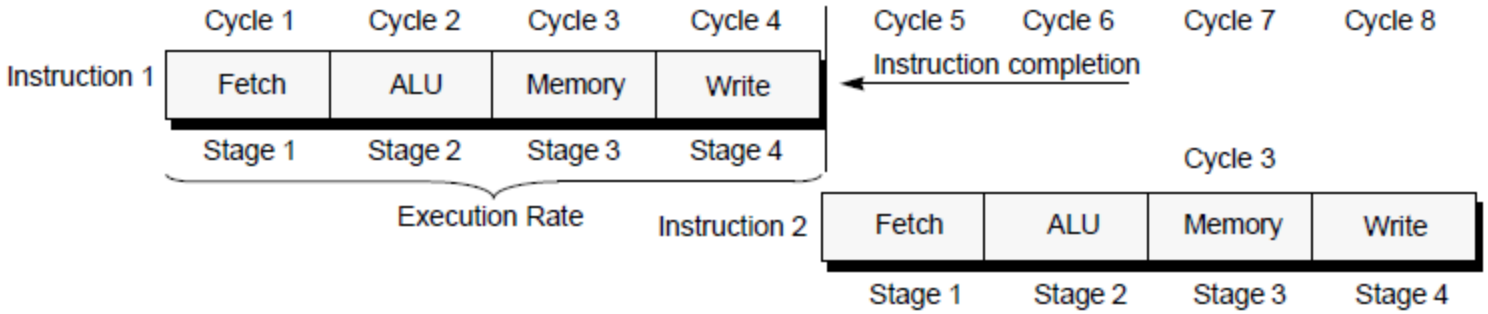
CPU Special-Purpose Registers

The CPU contains three special-purpose registers:

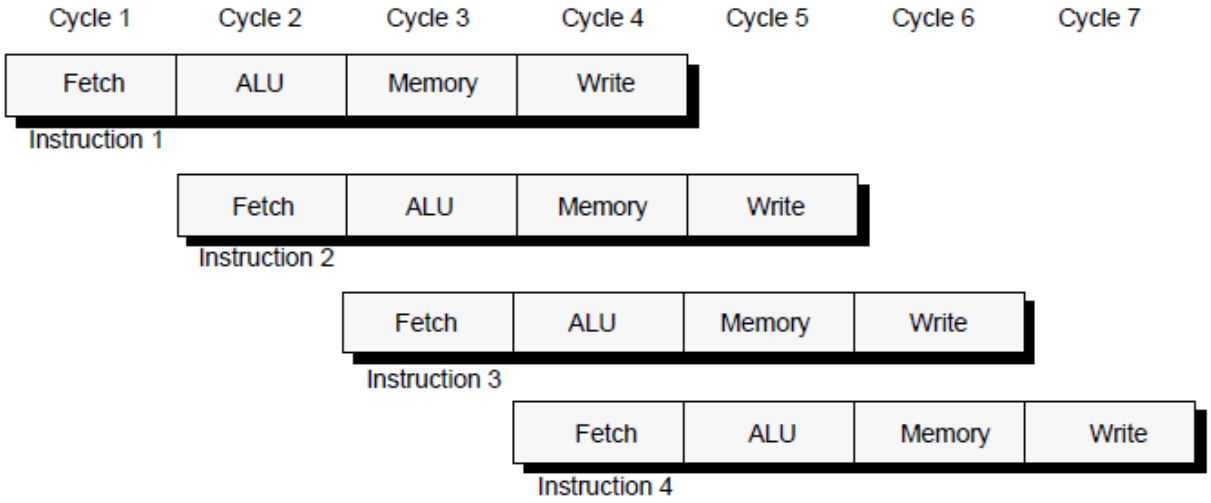
- *PC*—Program Counter register
- *HI*—Multiply and Divide register higher result
- *LO*—Multiply and Divide register lower result
  - During a multiply operation, the *HI* and *LO* registers store the product of integer multiply.
  - During a multiply-add or multiply-subtract operation, the *HI* and *LO* registers store the result of the integer multiply-add or multiply-subtract.
  - During a division, the *HI* and *LO* registers store the quotient (in *LO*) and remainder (in *HI*) of integer divide.
  - During a multiply-accumulate, the *HI* and *LO* registers store the accumulated result of the operation.

Ciclo di esecuzione

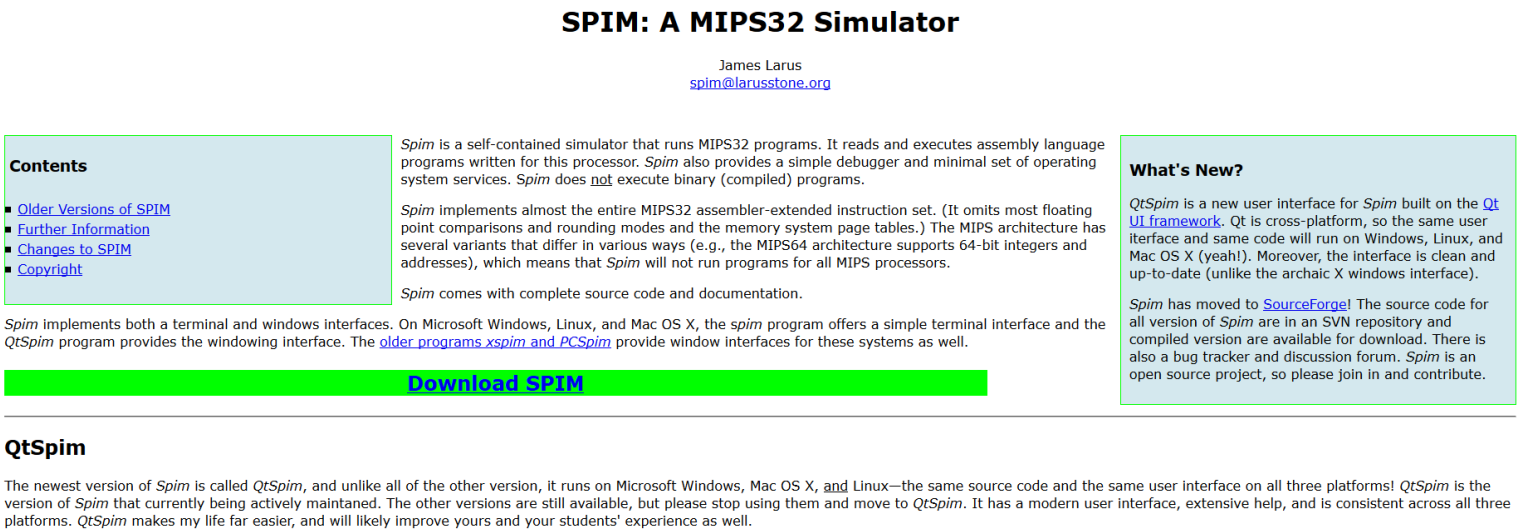
- Tutti i processori MIPS utilizzano delle pipeline. Una pipeline è suddivisa nei seguenti step: FETCH → OPERAZIONI_ARITMETICHE → ACCESSO_MEMORIA → SCRITTURA



Esistono varianti (ad es. esecuzione parallela di pipeline). Invece di attendere 4 colpi di clock per effettuare una nuova fetch una nuova istruzione viene recuperata ad ogni colpo di clock



SPIM



SPIM: A MIPS32 Simulator

James Larus
spim@larusstone.org

Contents

- Older Versions of SPIM
- Further Information
- Changes to SPIM
- Copyright

Spim is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor. *Spim* also provides a simple debugger and minimal set of operating system services. *Spim* does not execute binary (compiled) programs.

Spim implements almost the entire MIPS32 assembler-extended instruction set. (It omits most floating point comparisons and rounding modes and the memory system page tables.) The MIPS architecture has several variants that differ in various ways (e.g., the MIPS64 architecture supports 64-bit integers and addresses), which means that *Spim* will not run programs for all MIPS processors.

Spim comes with complete source code and documentation.

Spim implements both a terminal and windows interfaces. On Microsoft Windows, Linux, and Mac OS X, the *spim* program offers a simple terminal interface and the *QtSpim* program provides the windowing interface. The older programs *xspim* and *PCSpim* provide window interfaces for these systems as well.

Download SPIM

What's New?

QtSpim is a new user interface for *Spim* built on the Qt UI framework. Qt is cross-platform, so the same user interface and same code will run on Windows, Linux, and Mac OS X (yeah!). Moreover, the interface is clean and up-to-date (unlike the archaic X windows interface).

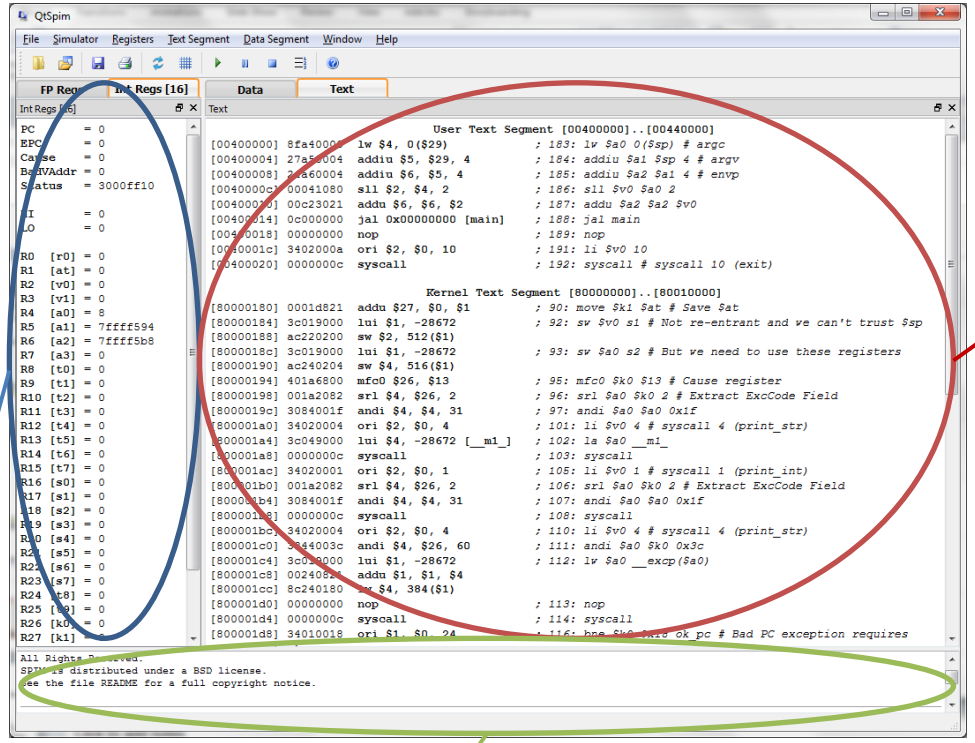
Spim has moved to SourceForge! The source code for all version of *Spim* are in an SVN repository and compiled version are available for download. There is also a bug tracker and discussion forum. *Spim* is an open source project, so please join in and contribute.

QtSpim

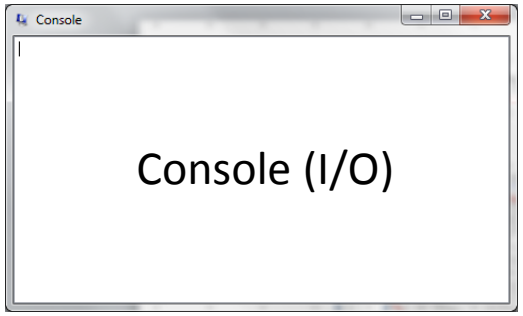
The newest version of *Spim* is called *QtSpim*, and unlike all of the other version, it runs on Microsoft Windows, Mac OS X, and Linux—the same source code and the same user interface on all three platforms! *QtSpim* is the version of *Spim* that currently being actively maintaned. The other versions are still available, but please stop using them and move to *QtSpim*. It has a modern user interface, extensive help, and is consistent across all three platforms. *QtSpim* makes my life far easier, and will likely improve yours and your students' experience as well.

- E' un simulatore di una CPU che obbedisce alle convenzioni MIPS.
- Perché usare un simulatore e non la macchina vera?
  - Usiamo tutti la stessa ISA indipendentemente dal calcolatore reale.
  - Ci offre una serie di strumenti che rendono la programmazione più comoda.
  - Maschera certi aspetti reali a cui non saremmo interessati (es, delays).
- Disponibile a questo URL http://spimsimulator.sourceforge.net/

SPIM (interfaccia)

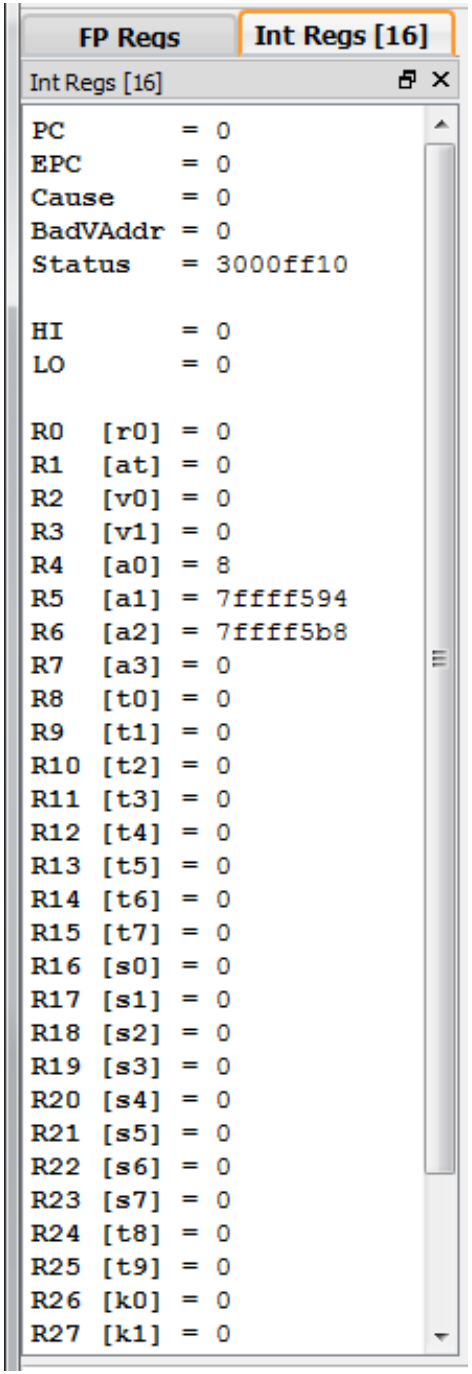


Memoria (Text e Data)



Banco Registri

Logs



SPIM (Registri)

- 32 registri a 32bit per operazioni su interi (**$0..$31**).
- 32 registri a 32 bit per operazioni in virgola mobile (**$FP0**..**$FP31**).
- registri speciali a 32bit:
  - il **Program Counter** (**PC**) l'indirizzo della prossima istruzione da eseguire;
  - **HI** e **LO** usati nella moltiplicazione e nella divisione;
  - **EPC, Cause, BadVAddr, Status** vengono usati nella gestione delle eccezioni.
- I registri general-purpose sono chiamati R0, R1, … R31, tra [] c'è il nome secondo la convenzione MIPS.
- Il loro valore è sempre visibile durante l'esecuzione di un programma, può essere visualizzato secondo diverse codifiche
  `(Register -> <binary|hex|decimal>.`

Richiamo sulle Convenzioni MIPS

Nome	**Numero**	**Utilizzo**	**Preservato durante le chiamate**
$zero	0	costante zero	*Riservato MIPS*
$at	1	riservato per l'assemblatore	*Riservato Compiler*
$v0-$v1	2-3	valori di ritorno di una procedura	No
$a0-$a3	4-7	argomenti di una procedura	No
$t0-$t7	8-15	registri temporanei (non salvati)	No
$s0-$s7	16-23	registri salvati	Si
$t8-$t9	24-25	registri temporanei (non salvati)	No
$k0-$k1	26-27	gestione delle eccezioni	*Riservato OS*
$gp	28	puntatore alla global area (dati)	Si
$sp	29	stack pointer	Si
$s8	30	registro salvato (fp)	Si
$ra	31	indirizzo di ritorno	No

Il registro **$1 ($at)** viene usato come variabile temporanea nell'implementazione delle pseudo-istruzioni.

Spim (Memoria)

- Un programma Assembly è composto da due elementi distinti, che risiedono nella RAM del calcolatore:
  - segmento testo (Text:, a partire da indirizzo 0x00400000);

indirizzo codice Rappr. simbolica Linea nel source file

```
[00400000] 8fa40000  lw $4, 0($29)           ; 183: lw $a0 0($sp) # argc
```

  - segmento dati (Data:, a partire da indirizzo 0x10000000).

```
User Stack [7ffff590]..[80000000]
[7ffff590]    00000008  7ffff696  7ffff686  7ffff67f    . . . . . .
```

indirizzo dati Rappr. stringa

- Il programma Assembly viene caricato da un file sorgente e rilocato a partire dall'indirizzo 0x00400024 in Text.
- Dall'indirizzo 0x00400000 a 0x00400024 c'è un preambolo di inizializzazione, una chiamata all'istruzione che sta alla label `main` e infine, una volta concluso `main`, la syscall `exit`.

Richiamo di istruzioni aritmetiche (somma, sottrazione)

```
add $s1, $s2, s3 # $s1 = $s2 + s3, rileva overflow

sub $s1, $s2, s3 # $s1 = $s2 - s3, rileva overflow

addi $s1, $s2, 13# $s1 = $s2 + 13, rileva overflow

addu $s1, $s2, s3# $s1 = $s2 + s3, unsigned, non rileva overflow

subu $s1, $s2, s3# $s1 = $s2 - s3, unsigned, non rileva overflow

addui $s1, $s2, 27        # $s1 = $s2 + 17, unsigned, non rileva overflow
```

- Convenzioni di notazione:
  - Identificativo con iniziale minuscola: deve essere un registro o un valore immediato (intero con segno su 16 bit);
  - Identificativo con iniziale «$»: deve essere un registro.

Es. 1.1

- Si scriva il codice Assembly che:
  - metta il valore 5 nel registro $s1,
  - metta il valore 7 nel registro $s2,
  - metta la somma dei due nel registro $s0.

Es. 1.1 (step by step)

- *(1) scrivere il codice assembly in un file di testo*
- *(2) caricare il file in SPIM e osservare il segmento testo*
- *(3) lanciare l'esecuzione con* F5, cosa succede?
- *(4) osservare come variano i registri coinvolti nelle operazioni*
- *(5) ripetere mediante l'uso di un break point, aggiornare il source file, re-inizializzare il simulatore e ricominciare da (2)*

Es. 1.2

- Si traduca in Assembly la seguente riga di codice:

  A = B+C-(D+E),

  assegnando alle variabili A, B, C, D, E i registri `$s0, ..., $s4`.
- Si assumano valori iniziali 1, 2, 3 e 4

Es. 1.2 Soluzione e osservazioni

```
main:

addi $s1, $zero, 1  # $s1=1, B=1
addi $s2, $zero, 2  # $s2=2, C=2
addi $s3, $zero, 3  # $s3=3, D=3
addi $s4, $zero, 4  # $s4=4, E=4

add $t0, $s1, $s2   # $t0=$s1+$s2, $t0=B+C
add $t1, $s3, $s4   # $t1=$s3+$s4, $t1=D+E
sub $s0, $t0, $t1   # $s0=$t0-$t1, $s0=(B+C)-
(D+E)
jr $ra
```

- Il risultato finale ottenuto nel registro $t0 è corretto e pari a 0xfffffffc.
- Prova:
  - (1+2)-(3+4) = 3-7 = -4
  - 0xfffffffc = $[1111,1111,1111,1111,1111,1111,1111,1100]_{C2}$ = ...
  - ... = -{[0000,0000,0000,0000,0000,0000,0000,0011]+1}$_2$ = -{100}$_2$ = -4

Osservazioni

- Filosofia RISC: un'operazione che implica più di due addendi viene divisa in una sequenza di operazioni (HW più semplice se il numero di operatori è costante).

A=(B+C)-(D+E)

```
add $t0, $s1, $s2# $t0=$s1+$s2, $t0=B+C
add $t1, $s3, $s4# $t1=$s3+$s4, $t1=D+E
sub $s0, $t0, $t1# $s0=$t0-$t1, $s0=(B+C)-(D+E)
```

- Spetta al compilatore (o al programmatore Assembly) il compito di ottimizzare la sequenza di operazioni.

Istruzioni: moltiplicazione

- Due istruzioni:
  - `mult $rs $rt`
  - `multu $rs $rt` `# unsigned`

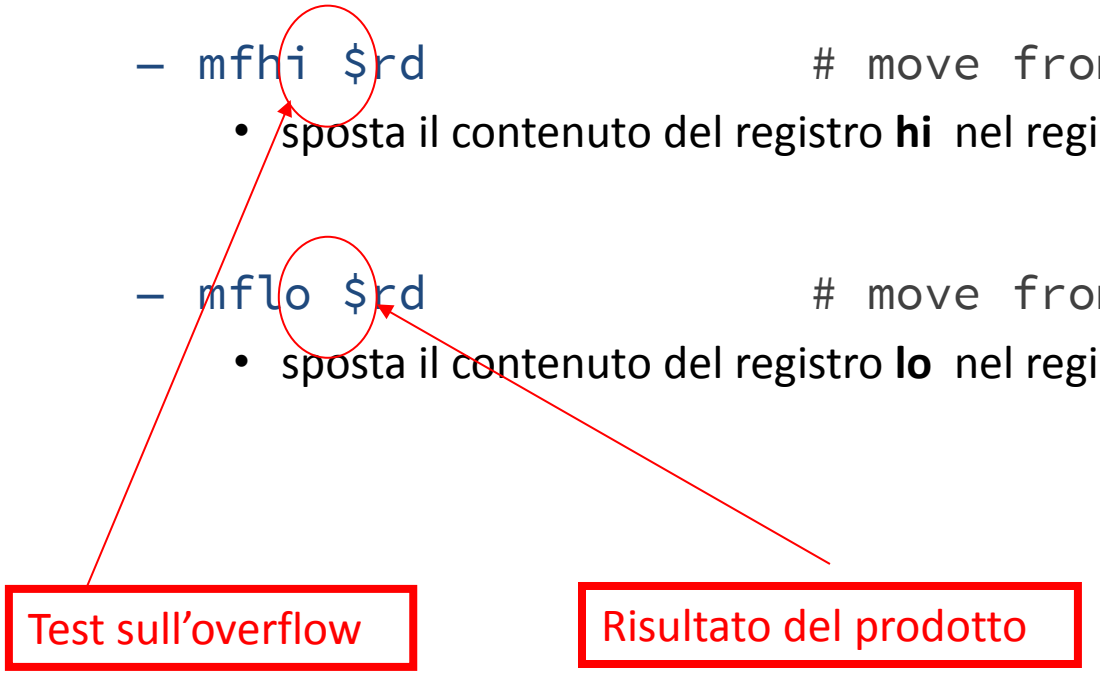
- Il registro destinazione è **implicito.**

- Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati **hi (High order word)** e **lo (Low order word)**.

- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.

Istruzioni: moltiplicazione

- Il risultato della moltiplicazione si preleva dal registro **hi** e dal registro **lo** utilizzando le due istruzioni:
  - `mfhi $rd` `# move from hi`
    - sposta il contenuto del registro **hi** nel registro **rd**;
  - `mflo $rd` `# move from lo`
    - sposta il contenuto del registro **lo** nel registro **rd**.



Operazioni aritmetiche: divisione

```
div $s2, $s3  # $s2 / $s3, divisione intera
```

- Il risultato della divisione intera va in:
  - Lo: $s2 / $s3 [quoziente];
  - Hi: $s2 mod $s3 [resto].

- Il risultato va quindi prelevato dai registri Hi e Lo utilizzando ancora la `mfhi` e `mflo`.

Istruzioni: pseudo-istruzioni

- Le pseudoistruzioni sono un modo compatto ed intuitivo di specificare un insieme di istruzioni.
- La traduzione della pseudoistruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore.

Esempi:

- `move $t0, $t1` `# pseudo istruzione`
  - `add $t0, $zero, $t1` `# (in alternativa) addi $t0, $t1, 0`

- `mul $s0, $t1, $t2` `# pseudo istruzione`
  - `mult $t1, $t2`
  - `mflo $s0`

- `div  $s0, $t1, $t2` `# pseudo istruzione`
  - `div $t1, $t2`
  - `mflo $s0`

Esercizio 1.3

- Si implementi il codice Assembly che effettua la moltiplicazione e la divisione tra i numeri 100 e 45, utilizzando le istruzioni dell'ISA e le pseudoistruzioni.

Esercizio 1.3 – Soluzione & Osservazioni

```
main:

addi $s1, $zero, 100                    # $s1 = 100
addi $s2, $zero, 45             # $s2 = 45

mult $s1, $s2                           # [Hi, Lo] = $s1 * $s2
mflo $s0                        # $s0 = Lo

move $s0, $zero                 # Reset $s0
mul $s0, $s1, $s2               # $s0 = $s1 * $s2

move $s0, $zero                 # Reset $s0
div $s1, $s2                            # Hi = $s1 % $s2, Lo = $s1 / $s2
mflo $s0                        # $s0 = Lo

addi $s0, $zero, 0              # Reset $s0
div $s0, $s1, $s2               # $s0 = $s1 / $s2
```

- SPIM implementa l'operazione div a tre operatori con un'eccezione (si ossevino i valori di PC, ovvero le righe di memoria eseguite dal simulatore...).

- L'opzione *bare machine* deve essere disattivata per usare `div` a tre operatori.