

MIPS Instruction Set 1

Architettura degli Elaboratori e Laboratorio

20 marzo 2012

Istruzioni Comuni 1/3

- Istruzioni Aritmetiche:

Instruction	Example	Meaning	Comments
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	Always 3 operands
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	Always 3 operands
add immediate	addi \$1,\$2,10	$\$1 = \$2 + 10$	add constant
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	Always 3 operations
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	Always 3 operations
add immed.unsigned	addiu \$1,\$2,10	$\$1 = \$2 + 10$	Always 3 operations

- Istruzioni Logiche:

Instruction	Example	Meaning	Comments
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	3 register operands
or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	3 register operands
and immediate	andi \$1,\$2,10	$\$1 = \$2 \& 10$	AND constant
or immediate	ori \$1,\$2,10	$\$1 = \$2 10$	OR constant
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant

Istruzioni Comuni 2/3

- Istruzioni di Trasferimento:

Instruction	Example	Meaning	Comments
load word	lw \$1,10(\$2)	$\$1 = \text{mem}[\$2+10]$	mem to reg
store word	sw \$1,10(\$2)	$\text{mem}[\$2+10] = \1	reg to mem
load upper immed.	lui \$1,10	$\$1 = 10 \times 2^{16}$	const into upper 16 bits

Istruzioni Comuni 3/3

- Salti Condizionati:

Instruction	Example	Meaning	Comments
branch on equal	beq \$1,\$2,10	if(\$1==\$2)go to PC+4+10	Equal test
branch on not equal	bne \$1,\$2,10	if(\$1!=\$2)go to PC+4+10	Not equal test
set on less then	slt \$1,\$2,\$3	if(\$2<\$3)\$1=1;else \$1=0	Less than cmp
set on less then imm	slti \$1,\$2,100	if(\$2<100)\$1=1;else \$1=0	Less than imm cmp

- Salti Incondizionati:

Instruction	Example	Meaning	Comments
jump	j 1000	go to 1000	Jump to target address
jump register	jr \$31	go to \$31	For switch, procedure return
jump and link	jal 1000	\$31=PC+4;go to 1000	For procedure call

Registri

MIPS reference cards:

<http://refcards.com/docs/waetzigj/mips/mipsref.pdf>

<http://www.mips.com/media/files/MD00565-2B-MIPS32-QRC-01.01.pdf>

- 32 registri general purpose a 32bit:

Numero	Nome	Uso	Preservati dal chiamato?
\$0	\$zero	costante 0	N/A
\$1	\$at	temp assembler	No
\$2-\$3	\$v0-\$v1	retval funzioni e expr eval	No
\$4-\$7	\$a0-\$a3	args funzione	No
\$8-\$15	\$t0-\$t7	temp	No
\$16-\$23	\$s0-\$s7	temp da salvare	Si
\$24-\$25	\$t8-\$t9	temp	No
\$26-\$27	\$k0-\$k1	riservati kernel	No
\$28	\$gp	global pointer	Si
\$29	\$sp	stack pointer	Si
\$30	\$fp	frame pointer	Si
\$31	\$ra	return address	N/A

Sintassi Assembly

- Programma include **.data** e **.text**
- Commenti iniziano con **#**, il resto della linea e' ignorato
- Nomi identificatori: sequenze di lettere, numeri, underbars (**_**) e punti (**.**)
- Le etichette sono dichiarate mettendole ad inizio riga seguite da :
- Formato delle istruzioni: opcode seguito da uno o piu' operandi: `addi $t0, $t0, 1`
- Operandi devono essere valori letterali o registri
- I registri possono contenere valori a 32 bit
- I numeri sono in base 10 di default. Il prefisso **0x** indica gli esadecimali
- Le stringhe sono racchiuse tra apici, possono includere `\n` o `\t`

Scheletro di file assembly

```
                                # PROLOGO
        .text
        .align 2
main:                                # PROGRAMMA
< il programma va qui >

$FINE:                            # EPILOGO
        jr      $31
        .end    main

                                # DATI
        .data
        .align 2

< le variabili vanno qui >
```


Variabili, Semplici Operazioni

- Dato un semplice (e storico) programma C, traduciamolo in assembly e simuliamone il funzionamento.

```
/* variabili globali */  
int x, y, z[4];  
  
void main (void) {  
    x = 8;  
    y = x - 1;  
    z[2] = y;  
}
```

Variabili, Semplici Operazioni

- Alcune versioni di SPIM limitano l'uso dei nomi di variabile dato che alcune lettere sono considerate **parole chiave**, buona regola puo' essere usare i nomi delle variabili **raddoppiati**, tipo $x \rightarrow xx$.

```

li    $2, 8           # x = 8
sw    $2, xx          # salva x modificato in memoria

lw    $2, xx          # carica x dalla memoria
addi  $2, $2, -1     # x - 1
sw    $2, yy          # salva y modificato in memoria

lw    $2, yy          # carica y dalla memoria
addi  $3, $0, 8      # punta al secondo elemento di z
sw    $2, zz($3)     # z[2] = y

```

Variabili, Dichiarazione

- Come istruire in assembly sulle dimensioni delle variabili nei nostri esempi?
Nella parte dati del codice occorre specificare:
 - da che indirizzo cominciano i dati (**.data**)
 - il tipo di allineamento dei dati in memoria (**.align**) specificato in base 2
 - ciascuna variabile con il relativo numero di byte che occuperà' (**.space**)

```
# variabili globali

        .data 0x10002000 # indirizzo di partenza dei dati
        .align 2         # allineamento in memoria 2^2 = 4 byte
xx:     .space 4         # numero di byte allocati per x
yy:     .space 4         # numero di byte allocati per y
zz:     .space 16        # numero di byte allocati per z[]
```

Istruzione Condizionale IF

- Se la condizione e' vera eseguo il corpo, altrimenti salto all'istruzione successiva. In assembly e' generalmente piu' vantaggioso lavorare con la condizione **negata** rispetto al C. Perché?
- Vedi file "exIF0.s" e "exIF1.s"

```

if ( i == j ) {
    i++;
}

j--;

```

Supponiamo **i** in \$2, **j** in \$3

```

    bne $2, $3, L1    # branch if ! ( i == j )
    addi $2, $2, 1    # i++
L1:
    addi $3, $3, -1   # j--

```

Istruzione Condizionale IF/ELSE

- Se la condizione e' vera eseguo il ramo IF, altrimenti salto al ramo ELSE, ricordando che in ambedue i casi devo comunque passare all'istruzione successiva. In assembly e' generalmente piu' vantaggioso lavorare con la condizione **negata** rispetto al C.
- Vedi file "exIFEL0.s" e "exIFEL1.s"

```

if ( i == j ) {
    i++;
} else {
    j--;
}

j += i;

```

Supponiamo **i** in \$2, **j** in \$3

```

        bne  $2, $3, ELSE    # branch if ! ( i == j )
        addi $2, $2, 1      # i++
        j    L1             # jump over else
ELSE:
        addi $3, $3, -1     # j--
L1:
        add  $3, $3, $2     # j += i

```

Ciclo WHILE

- Diversamente dai *condizionali*, i *cicli* non hanno istruzioni assembly che li supportino direttamente, occorre quindi convertirli in *condizionali* + *GOTO*.
- Ricordiamo che il corpo del WHILE potrebbe non essere eseguito.
- Vedi file "exWHILE.s"

```
while ( i < j ) {
    k++;
    i = i * 2;
}
```

⇒

```
L1:
    if ( i < j ) {
        k++;
        i = i * 2;
        goto L1;
    }
```

Supponiamo **i** in \$2, **j** in \$3, **k** in \$8

L1:

```
bge $2, $3, EXIT # branch if ! ( i < j )
addi $8, $8, 1 # k++
add $2, $2, $2 # i = i * 2
j L1 # jump back to top of loop
```

EXIT:

- C'e' un altro modo di moltiplicare un numero per le potenze di 2 ?

Ciclo DO/WHILE

- Ricordiamo che il corpo del DO/WHILE deve essere eseguito almeno una volta.
- Vedi file “exDOWHILE.s”

```
do {
    k++;
    i = i * 2;
} while ( i < j )
```

⇒

```
L1:
    k++;
    i = i * 2;
    if ( i < j ) {
        goto L1;
    }
```

Supponiamo **i** in \$2, **j** in \$3, **k** in \$8

L1:

```
addi $8, $8, 1      # k++
add  $2, $2, $2     # i = i * 2
bge  $2, $3, EXIT   # branch if ! ( i < j )
j    L1              # jump back to top of loop
```

EXIT:

Ciclo FOR

- Ricordiamo che l'inizializzazione della variabile di controllo fa parte del FOR.
- Vedi file "exFOR0.s"

```
for ( i = 0; i < j; i++ ) {
    k[i] = i * 2;
}
```

⇒

```

i = 0;
L1:
    if ( i < j ) {
        k[i] = i * 2;
        i++;
        goto L1;
    }
```

Supponiamo **i** in \$2, **j** in \$3, **addr(k)** in \$8

```

    la    $8, kk          # load addr(k)
    li    $2, 0          # i = 0
L1:
    bge   $2, $3, EXIT   # branch if ! ( i < j )
    add   $9, $2, $2     # i * 2
    sw    $9, 0($8)     # k[i] = i * 2
    addi  $2, $2, 1     # i++
    addi  $8, $8, 4     # update addr(k)
    j     L1             # jump back to top of loop
EXIT:
```


Cicli Annidati 1/2

- Prendiamo come esempio l'inizializzazione di una matrice $a \times b$, costituita da due cicli FOR annidati.
- Vedi file "exFOR1.s" e "exFOR2.s"

```

for ( i = 0; i < a; i++ ) {
    for ( j = 0; j < b; j++ ) {
        k[i][j] = i + j;
    }
}

```

⇒

```

    i = 0;
LI:
    if ( i < a ) {
        j = 0;
LJ:
        if ( j < b ) {
            k[i][j] = i + j;
            j++;
            goto LJ;
        }
        i++;
        goto LI;
    }

```

Cicli Annidati 2/2

Supponiamo **i** in \$2, **j** in \$3, **addr(k)** in \$8, **a** in \$4, **b** in \$5

```

        la    $8, kk          # carica addr(k)
        li    $2, 0           # i = 0, vai alla prima riga
LI:
        bge   $2, $4, EXIT    # tutte le righe sono state azzerate?
        li    $3, 0           # j = 0, vai alla prima colonna
LJ:
        bge   $3, $5, NROW    # tutte le colonne sono state azzerate?
        add   $9, $2, $3      # i + j
        sw    $9, 0($8)       # k[i][j] = i + j
        addi  $3, $3, 1        # j++
        addi  $8, $8, 4        # update addr(k)
        j     LJ              # vai ad aggiornare la prossima colonna
NROW:
        addi  $2, $2, 1        # i++
        j     LI              # vai ad aggiornare la prossima riga
EXIT:

```

Formato Istruzioni 1/2

MIPS reference cards:

<http://refcards.com/docs/waetzig/mips/mipsref.pdf>

<http://www.mips.com/media/files/MD00565-2B-MIPS32-QRC-01.01.pdf>

- Formato R (Register) per istruzioni aritmetiche
- Formato I (Immediate) per istruzioni di trasferimento, branch o immediati
- Formato J (Jump) per istruzioni di salto

Formato	31-26	25-21	20-16	15-11	10-6	5-0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	address/immediate		
J	op	target address				

op	Operation code
rs	First source register operand
rt	Second source register operand
rd	Destination register operand
shamt	Shift amount - used in shift instructions
funct	Select the variant of the operation in the op code field

Formato Istruzioni 2/2

Ad esempio, l'istruzione **addu \$30, \$0, \$29** è codificata nella word

0x001df021

in binario

000000000011101 11110000010001

perché:

i bit **31-26** sono sempre a zero;

i bit **25-21** codificano il registro sorgente 1 (\$0 e' il numero 0);

i bit **20-16** codificano il registro sorgente 2 (\$29 e' il numero 29, bin 11101);

i bit **15-11** codificano il registro destinazione (\$30 e' il numero 30, bin 11110);

i bit **10-6** sono sempre a zero;

i bit **5-0** codificano il codice dell'istruzione (addu e' la n. 33, bin 100001).

SPIM

- emulatore software architettura MIPS R2000/R3000
- debugger + servizi base di sistema operativo
- realizzazione/verifica di semplici programmi assembler

- Sito ufficiale:
<http://pages.cs.wisc.edu/larus/spim.html>
- Documentazione di riferimento in appendice A del libro di testo o
http://pages.cs.wisc.edu/larus/HP_AppA.pdf
- Libro utile:
Dominic Sweetman, See MIPS Run, Morgan Kaufmann Publishers,
AcademicPress, 2002

A cosa ci servira'

Tradurre e verificare un semplice codice C in assembler MIPS.

SPIM Interfaccia

Area dei registri

```

PC      = 00400000      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 (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
R5 (a1) = 00000000      R13 (t5) = 00000000     R21 (s5) = 00000000     R29 (sp) = 7ffffcfc
R6 (a2) = 00000000      R14 (t6) = 00000000     R22 (s6) = 00000000     R30 (s8) = 00000000
R7 (a3) = 00000000      R15 (t7) = 00000000     R23 (s7) = 00000000     R31 (ra) = 00000000

Double Floating Point Registers
FP0 = 0.00000      FP8 = 0.00000      FP16 = 0.00000      FP24 = 0.00000
FP2 = 0.00000      FP10 = 0.00000     FP18 = 0.00000     FP26 = 0.00000
FP4 = 0.00000      FP12 = 0.00000     FP20 = 0.00000     FP28 = 0.00000
FP6 = 0.00000      FP14 = 0.00000     FP22 = 0.00000     FP30 = 0.00000

Single Floating Point Registers

```

Comandi del simulatore/
debugger

quit load reload run step clear

set value print breakpoints help terminal mode

Area del programma
"text segment"

```

Text Segments
[0x00400000] 0x8fa40000 lw $4, 0($29) ; 140: lw $a0, 0($sp)
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 141: addiu $a1, $sp, 4
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 142: addiu $a2, $a1, 4
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 143: sll $v0, $a0, 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 144: addu $a2, $a2, $v0
[0x00400014] 0x0c000000 jal 0x00000000 [main] ; 145: jal main
[0x00400018] 0x00000000 nop ; 146: nop
[0x0040001c] 0x3402000a ori $2, $0, 10 ; 148: li $v0 10
[0x00400020] 0x0000000c syscall ; 149: syscall

```

Area dei dati
"data segment"

```

Data Segments

DATA
[0x10000000]... [0x10020000] 0x00000000

STACK
[0x7ffffcfc] 0x00000000

KERNEL DATA
[0x90000000] 0x78452020 0x74706563 0x206e6669 0x636f2000
[0x90000010] 0x72727563 0x61206465 0x6920646e 0x72666e67

```

Area dei messaggi

```

SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /trsp.handler

```