Esercitazione 3

Architettura degli Elaboratori e Laboratorio

12 aprile 2013

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

Registri

MIPS reference card:
http://refcards.com/docs/waetzigj/mips/mipsref.pdf

- 32 registri general purpose a 32bit:

Numero	Nome	Uso	Preservati dal chiamato?
$0	$zero	constante 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

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	or $1,$2,10	$1=$2|10	OR constant
shift left logical	sll $1,$2,10	$1=$2«10	Shift left by constant
shift right logical	srl $1,$2,10	$1=$2»10	Shift right by constant

Istruzioni Comuni 2/3

- Istruzioni di Trasferimento:

Instruction	**Example**	**Meaning**	**Comments**
load word	lw $1,10($2)	$1=mem[$2+10]	mem to reg
store word	sw $1,10($2)	mem[$2+10]=$1	reg to mem
load upper immed.	lui $1,10	$1=10x2$\hat{}$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 compare

- 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

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

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
addu $2, $2, -1     # x - 1
sw   $2, yy         # salva y modificato in memoria

lw   $2, yy         # carica y dalla memoria
addu $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 occupera' (**.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[]
```

Esempio di file assembly

```
                                    # PROLOGO
        .text
        .align 2
main:                               # PROGRAMMA
        li      $2, 8
        sw      $2, xx
        lw      $2, xx
        addu    $2, $2, -1
        sw      $2, yy
        lw      $2, yy
        addu    $3, $0, 8
        sw      $2, zz($3)
$FINE:                              # EPILOGO
        j       $31
        .end    main
                                    # DATI
        .data   0x10002000
        .align  2
xx:     .space 4
yy:     .space 4
zz:     .space 16
```

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.
- 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 )
```

$\Rightarrow$

```
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;
  }
}
```

$\Rightarrow$

```
  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?
        addu $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 card:
http://refcards.com/docs/waetzigj/mips/mipsref.pdf

- Formato R (Register) per istruzioni aritmetiche
- Formato I (Immediate) per istruzioni di trasferimento, brach 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
00000000000011101 11110000000100001
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).

Esercitazione 1/2

```
#define NUM 8

int A[NUM], B[NUM], C[NUM];

void main(void) {
  register int i, j;

  // Inizializza array A
  j = 2;
  for( i = 0; i < NUM; i++ ) {
    A[i] = i << j;
    j++;
  }
```

Esercitazione 2/2

```
// Inizializza array B
i = NUM - 1;
j = 2;
while( i >= 0 ) {
  B[i] = i + j ;
   j = B[i];
   i--;
}

// Inizializza array C
for ( i = (NUM-1); i > 0; i-- ) {
  for ( j = 0; j < NUM; j++ ) {
    C[j] = (A[i] + B[j]) << j;
  }
}
```

Scheletro di file assembly

```
                                # PROLOGO
        .text
        .align 2
main:                           # PROGRAMMA

< il programma va qui >

$FINE:                          # EPILOGO
        j       $31
        .end    main
                                # DATI
        .data
        .align  2

< le variabili vanno qui >
```

Requisiti Fondamentali

1. Una volta scritto l'assembler MIPS corrispondente, da inserire tra il prologo e l'epilogo, **lo si commenti riga per riga** in modo tale che sia chiaro dove vengono scritte o lette le singole variabili ed il funzionamento generale del programma.
2. Gli elementi degli array A, B e C **vanno mantenuti in memoria**, quindi se il loro valore viene **modificato** anche la memoria **deve essere aggiornata**.
3. Al termine della simulazione con SPIM, si catturi in un file di testo il **valore di tutti i registri** (finestra in alto di SPIM, tralasciare i registri floating point) **e della memoria** (finestra "data segment") giustificando i valori contenuti nei registri e nella memoria.
4. Considerare infine 2 istruzioni del programma e spiegare, in un altro file di testo, come i vari campi dell'istruzione vengono codificati in una singola word, come si può notare nella finestra centrale di SPIM.

Consegna

La relazione da consegnare e' formata da:

1. un file di testo .s contenente la traduzione in assembler MIPS del programma C
2. un file di testo con la copia degli output del simulatore SPIM al termine dell'esecuzione
3. un file di testo contenente la spiegazione del formato delle due istruzioni MIPS

- NON usate la cattura di immagini da schermo: copiate ed incollate usando un editor di testo
- NON includete file BINARI o creati con WORD, EXCEL...
- NON verranno considerate mail con piu' di 3 file allegati!

Spedite tutto a arc1@fe.infn.it entro le ore 23:59:59 di venerdí 19 aprile.
L'oggetto della mail deve essere nella forma:
LAB1-N#esercitazione-#gruppo
(es: LAB1-N3-99)