Calcolatori Elettronici A
a.a. 2008/2009

ISA e LINGUAGGIO ASSEMBLY MIPS
ESERCIZI

Massimiliano Giacomin

ESERCIZIO

Utilizzando la green card, tradurre in linguaggio macchina le due istruzioni in assembly MIPS seguenti

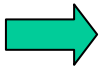
```
L1:   addi $t0, $t1, 1
      bne $t0, $s1, L1
```

ESERCIZIO

Utilizzando la green card, tradurre in linguaggio macchina le due istruzioni in assembly MIPS seguenti

```
L1:   addi $t0, $t1, 1
      bne $t0, $s1, L1
```

addi: formato-I, R[rt] = R[rs] + SignExtImm, codice op: 8_{hex}



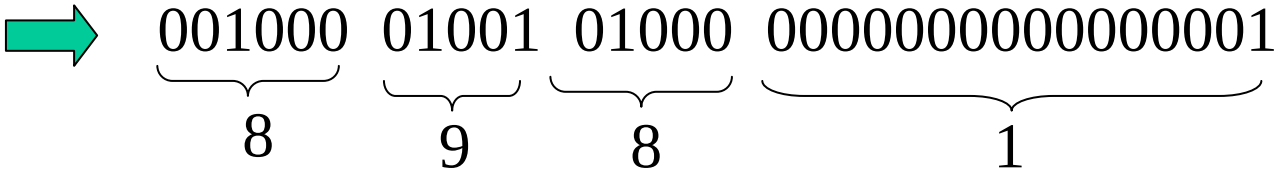
001000	01001	01000	0000000000000001
8	9	8	1

ESERCIZIO

Utilizzando la green card, tradurre in linguaggio macchina le due istruzioni in assembly MIPS seguenti

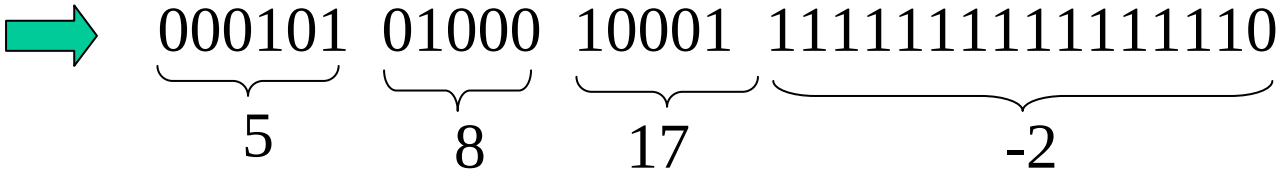
```
L1:   addi $t0, $t1, 1
      bne $t0, $s1, L1
```

addi: formato-I, R[rt] = R[rs] + SignExtImm, codice op: 8_{hex}



bne: formato-I, if(R[rs]!=R[rt]) PC=PC+4+BranchAddr, c. op: 5_{hex}

Branch Address: -2 (in istruzioni)



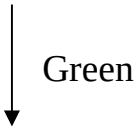
ESERCIZIO

Utilizzando la green card, ricavare dal formato numerico seguente la corrispondente istruzione assembly MIPS

AE090000

AE090000:

1010 1110 0000 1001 0000 0000 0000 0000



sw, che ha formato-I

AE090000:

1010 1110 0000 1001 0000 0000 0000 0000

Green card-retro

sw, che ha formato-I

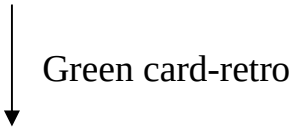
Opcode 10000 01001 0…..0

rs= r16 = $s0

rt= r9 = $t1

AE090000:

1010 1110 0000 1001 0000 0000 0000 0000



sw, che ha formato-I

➡ Opcode 10000 01001 0…..0

rs= r16 = $s0

rt= r9 = $t1

➡ sw $t1, 0($s0)

ESERCIZIO (condizione AND)

Frammento di codice C:

```
if  ((i == j) && (k == m))   f = g + h;
```

Supponendo che f, g, h, i, j, k, m corrispondano ai registri:

$s0, $s1, $s2, $s3, $s4, $s5, $s6 diventa:

ESERCIZIO (condizione AND)

Frammento di codice C:

```
if  ((i == j) && (k == m))   f = g + h;
```

Supponendo che f, g, h, i, j, k, m corrispondano ai registri:

$s0, $s1, $s2, $s3, $s4, $s5, $s6 diventa:

```
        bne $s3, $s4, ESCI
        bne $s5, $s6, ESCI
        add $s0, $s1, $s2
ESCI:
```

ESERCIZIO (condizione OR)

Frammento di codice C:

```
if  ((i == j) || (k == m))   f = g + h;
```

Supponendo che f, g, h, i, j, k, m corrispondano ai registri:

$s0, $s1, $s2, $s3, $s4, $s5, $s6 diventa:

ESERCIZIO (condizione OR)

Frammento di codice C:

```
if  ((i == j) || (k == m))   f = g + h;
```

Supponendo che f, g, h, i, j, k, m corrispondano ai registri:

$s0, $s1, $s2, $s3, $s4, $s5, $s6 diventa:

```
        beq $s3, $s4, L1
        bne $s5, $s6, ESCI
L1:     add $s0, $s1, $s2
ESCI:
```

ESERCIZIO

Tradurre in assembly MIPS la seguente procedura C, che copia la stringa *y* nella stringa *x*

```
void strcpy (char x[], char y[]){
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
       i += 1;
}
```

Supponendo che:

- gli indirizzi di *x* e *y* siano passati in ingresso
- per la variabile locali i venga usato *$s0*

```
<Predisponi lo stack e salva registri come chiamato>

      add  $s0, $zero, $zero # i = 0
                             # while((x[i]=y[i])!='\0')
  L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
      lbu  $t2, 0($t1)       # $t2 = y[i]
      add  $t3, $s0, $a0     # addr of x[i] in $t3
      sb   $t2, 0($t3)       # x[i] = y[i]
      beq  $t2, $zero, L2    # exit loop if y[i] == 0
      addi $s0, $s0, 1       # i = i + 1
      j    L1                # next iteration of loop
  L2:

<Ripristina registri salvati, svuota lo stack e ritorna>
```

NB: la procedura usa s0 (per la variabile locale)
e non ne chiama un'altra: deve solo salvare s0!

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

ESERCIZIO

Tradurre in assembly MIPS la seguente procedura C, che converte i caratteri della stringa *x* in maiuscolo se *m*=1, in minuscolo se *m*=0. Si ipotizzi di disporre delle procedure *tolower* e *toupper*

```
void strconv (char x[], int m){
    int i = 0;
    while (x[i] != '\0'){
       if(m==1)
         x[i]=toupper[x[i]];
         else x[i]=tolower[x[i]];
       i++;}
}
```

NB: l'indirizzo della stringa *x* e il parametro *m* sono in *$a0* e *$a1*

Prima versione del corpo della procedura

```
        add  $s0, $zero, $zero     # i = 0
                                   # while(x[i]!='\0')
loop:   add  $t1, $s0, $a0         # addr of x[i] in $t1
        lbu  $t2, 0($t1)           # $t2 = x[i]
        beq  $t2, $zero, exit      # exit loop if x[i] == 0
        move $a0, $t2              # ERRORE: cancelliamo a0!!!
        beq  $a1, zero, tolow
        jal  toupper               # if(m==1)
        j skip
tolow:  jal  tolower               # else…
skip:   sb $v0, 0($t1)             # x[i]=…
        addi $s0, $s0, 1           # i++
        j    loop                  # next iteration of loop
exit:
```

Devono essere preservati: possiamo usare registri s

Possiamo direttamente usare $t1 incrementandolo

```
        add  $s0, $zero, $zero     # i = 0
                                   # while(x[i]!='\0')
loop:   add  $t1, $s0, $a0         # addr of x[i] in $t1
        lbu  $t2, 0($t1)           # $t2 = x[i]
        beq  $t2, $zero, exit      # exit loop if x[i] == 0
        move $a0, $t2              # ERRORE: cancelliamo a0!!!
        beq  $a1, zero, tolow
        jal  toupper               # if(m==1)
        j skip
tolow:  jal  tolower               # else…
skip:   sb $v0, 0($t1)             # x[i]=…
        addi $s0, $s0, 1           # i++
        j    loop                  # next iteration of loop
exit:
```

```
        move $s0, $a0       # addr of x[] in $s0
        move $s1, $a1
loop:   lbu  $t0, 0($s0)            # $t0 = x[i]
        beq  $t0, $zero, exit       # exit loop if x[i] == 0
        move $a0, $t0               # per la procedura toxxx
        beq  $s1, zero, tolow
        jal  toupper                # if(m==1)
        j skip
tolow:  jal  tolower                # else…
skip:   sb $v0, 0($s0)              # x[i]=…
        addi $s0, $s0, 1            # i++
        j    loop                   # next iteration of loop
exit:
```

E’ chiaro che ora però dobbiamo salvare *s0* e *s1* come chiamato, oltre a *ra*: servirà spazio sullo stack per 3 elementi

```
strconv:    addi $sp, $sp, -12          # spazio stack per 3
                                        # elementi
            sw $ra, 0($sp)              #salvo ra, so, s1
            sw $s0, 4($sp)
            sw $s1, 8($sp)

            [corpo della procedura, come da lucido prec.]

            lw $ra, 0($sp)              #riprist. ra, so, s1
            lw $s0, 4($sp)
            lw $s1, 8($sp)
            addi $sp, $sp, 12           # svuoto stack
            jr $ra
```

INTRODUZIONE A SPIM

- SPIM (MIPS letto al contrario) è un interprete di programmi scritti per i processori MIPS R2000/R3000
- Include un debugger e fornisce dei servizi simili a quelli di un semplice sistema operativo
- Si trova nel CD del libro di testo. Documentazione nello stesso CD ed anche nell'appendice A
- Si presenta con 4 finestre:
  - registri
  - text
  - data
  - messaggi (es. nel caso di errori di sintassi)

Funzionalità di base

- I programmi vanno scritti come file di testo con estensione .s o .asm e possono essere caricati con File/Open
- Esecuzione di un programma: con Go (o F5) oppure passo-passo con Single step o Multiple step
- In esecuzione del programma, la lettura/scrittura di dati avviene via terminale attraverso la finestra Console
- La finestra registri permette di visualizzare come cambiano i valori

- Set value permette di caricare dei valori nei registri o in indirizzi di memoria
- Dato che il programma è chiamato come una procedura, deve contenere l'etichetta main e dovrebbe terminare con un'istruzione di ritorno jr $ra

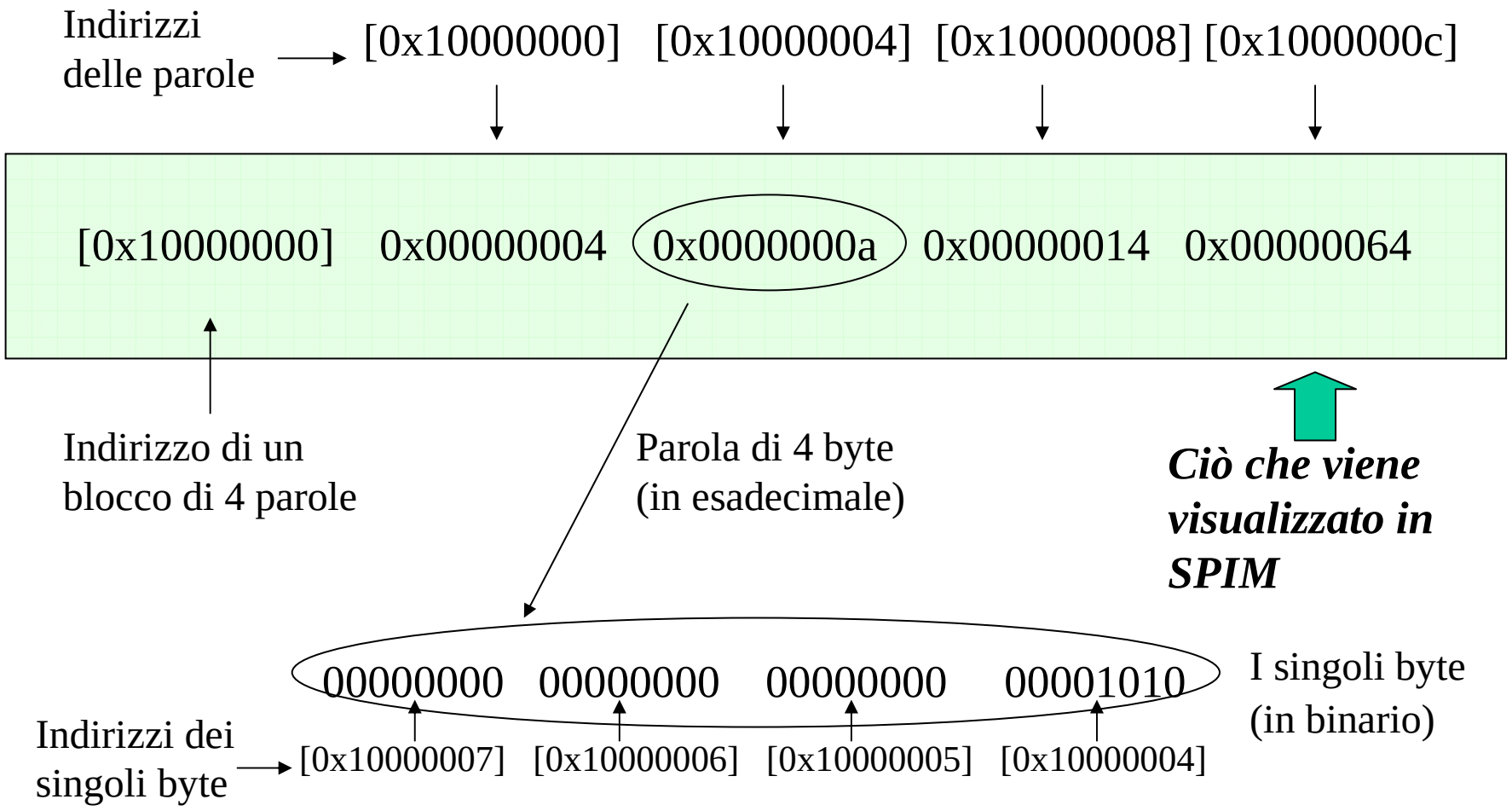
La finestra di testo

- Ciascuna istruzione è visualizzata su una linea del tipo

  [0x00400024] 0x02288820 add $17, $17, $8 ; 3: add $s1, $s1, $t0

- Il primo numero, che compare fra parentesi quadre, è l'**indirizzo esadecimale** dell'istruzione
- Il secondo numero è la **codifica esadecimale** dell'istruzione
- Il terzo elemento è la **descrizione mnemonica** dell'istruzione
- Ciò che segue dopo il punto e virgola è l'effettiva **linea di codice del programma utente** che ha generato l'istruzione; il numero 3 è la linea del file in cui si trova l'istruzione
- Se non compare la parte dopo il punto e virgola significa che l'istruzione è stata generata da SPIM come parte della traduzione di una pseudoistruzione

La finestra della memoria dati



Input\output con SPIM

- Un programma eseguito con SPIM può leggere i caratteri inseriti dall'utente e visualizzare eventuali uscite sulla finestra di console
- In pratica, si usano chiamate di sistema (syscall) per scrivere su (o leggere da) console
- Il programma deve caricare il codice della chiamata di sistema nel registro $v0 e gli argomenti nei registri $a0-$a3
- Le chiamate di sistema che forniscono un valore in uscita mettono il risultato in $v0 (o in $f0)
- Se si usano le chiamate di sistema occorre disabilitare l'opzione "*mapped I/O*"

Servizi di sistema e codici corrispondenti

Servizio	Codice della chiamata di sistema	Argomenti	Risultato
print_int	1	$a0 = intero	
print_float	2	$f12 = virgola mobile	
print_double	3	$f12 = doppia precisione	
print_string	4	$a0 = stringa	
read_int	5		intero (in $v0)
read_float	6		virgola mobile (in $f0)
read_double	7		doppia precisione (in $f0)
read_string	8	$a0=buffer, $a1=lunghezza	
sbrk	9	$a0=quantità	indirizzo (in $v0)
exit	10		

Esempio: codice per una stampa semplice

Il seguente codice stampa “risposta = 5”

```
main:
        .data
str:    .asciiz "risposta = "
        .text
        li $v0, 4           # codice della chiamata di  sistema per
                            # print_string
        la $a0, str         # carica in $a0 l’indirizzo della stringa da
                            # stampare
        syscall             # stampa della stringa
        li $v0, 1           # codice della chiamata di sistema per
                            # print_int
        li $a0, 5           # carica in $a0 numero intero da stampare
        syscall             # stampa del numero risultato
```

Esempio: programma con lettura da tastiera

- Programma che legge da console dei numeri interi e li somma finchè in ingresso non viene immesso 0
- Il corpo principale del programma è il ciclo seguente:

```
ciclo:  li $v0, 5                   # codice della chiamata di sistema per
                                    # read_int
        syscall                     # leggi numero da tastiera in $v0
        beq $v0, $zero, fine        # se $v0 vale 0 vai a fine
        add $t1, $t1, $v0           # somma nel registro $t1
        j ciclo                     # torna ad acquisire nuovo dato
```

- I numeri vengono inseriti uno alla volta dando “invio” dopo ciascuno

Programma completo

```
main:
            .text
            move $t1, $zero       # inizializza sommatore
ciclo:      li $v0, 5             # codice della chiamata di sistema per read_int
            syscall               # leggi numero da tastiera in $v0
            beq $v0, $zero, fine  # se $v0 vale 0 vai a fine
            add $t1, $t1, $v0     # somma nel registro $t1
            j ciclo               # torna ad acquisire nuovo dato
fine:
            li $v0, 4             # codice della chiamata di sistema per print_string
            la $a0, str           # indirizzo della stringa da stampare
            syscall               # stampa della stringa
            li $v0, 1             # codice della chiamata di sistema per print_int
            move $a0, $t1         # numero intero da stampare
            syscall               # stampa del numero risultato
            .data
str:        .asciiz “ECCO IL RISULTATO: "
```

Esercizi Proposti

- Traduzione di istruzioni assembly in linguaggio macchina e viceversa. Verifica con SPIM (attenzione: sbaglia la traduzione della costante nei salti condizionati)

- Tradurre in linguaggio assembly la seguente procedura, che stampa la somma dei quadrati da 1 ad un argomento passato in ingresso.

```
void sommaquad (int n){
    int somma=0;
    for(i=1; i<=n; i++)
       somma += i*i;
    printf("La somma vale %d\n", somma);
}
```

- Tradurre in linguaggio assembly la seguente procedura, che calcola il numero di Fibonacci di posizione *n*
  [si ricorda che F(0)=0, F(1)=1 e per k>1 F(k)=F(k-1)+F(k-2)]

```
int Fibonacci (int n){
    if(n==0)
      return 0;
      else if(n==1)
            return 1;
            else return Fibonacci(n-1)+Fibonacci(n-2);
}
```