Esempi di Programmi Assembly

Luigi Palopoli, Marco Roveri,
Luca Abeni

Scopo della lezione

- In questa lezione vedremo alcuni esempi di programmi (o frammenti di programmi) in vari linguaggi assembly per renderci conto delle differenze
- Partiremo da assembly RISC-V e ARM
- Successivamente passeremo all’assembly INTEL

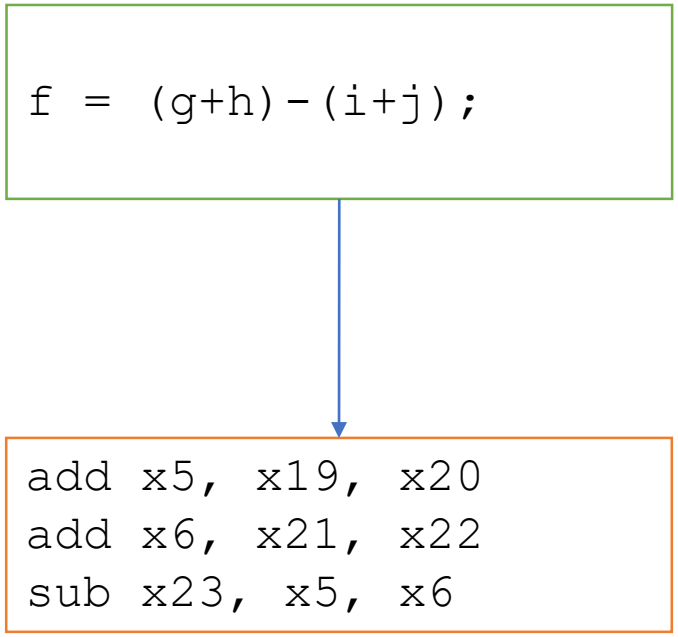
Semplici istruzioni aritmetiche logiche

- Partiamo dal semplicissimo frammento che abbiamo visto a lezione

```
f = (g + h) - (i + j);
```

Traduzione RISC-V

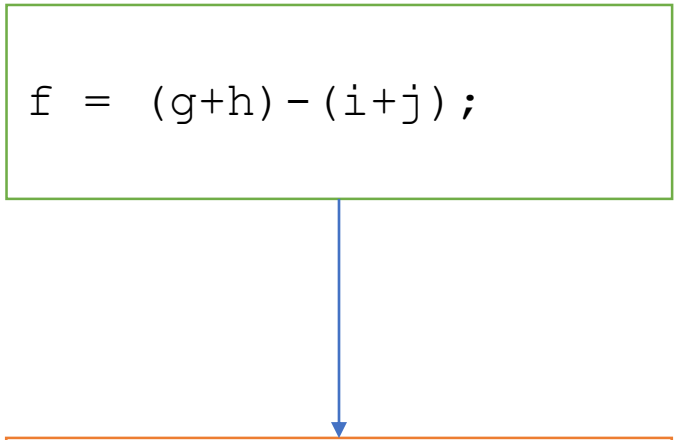
- Supponendo che g, h, i, j siano in x19, x20, x21, e x22, e che il risultato si voglia mettere in x23, la traduzione è semplicemente



Traduzione RISC-V (v2)

- Supponendo che g, h, i, j siano in x19, x20, x21, e x22, e che il risultato si voglia mettere in x23, la traduzione è semplicemente

```
f = (g+h)-(i+j);
```



```
add x23, x19, x20
add x6, x21, x22
sub x23, x23, x6
```

In questa versione é usato un registro in meno:
Il risultato intermedio é memorizzato in x23

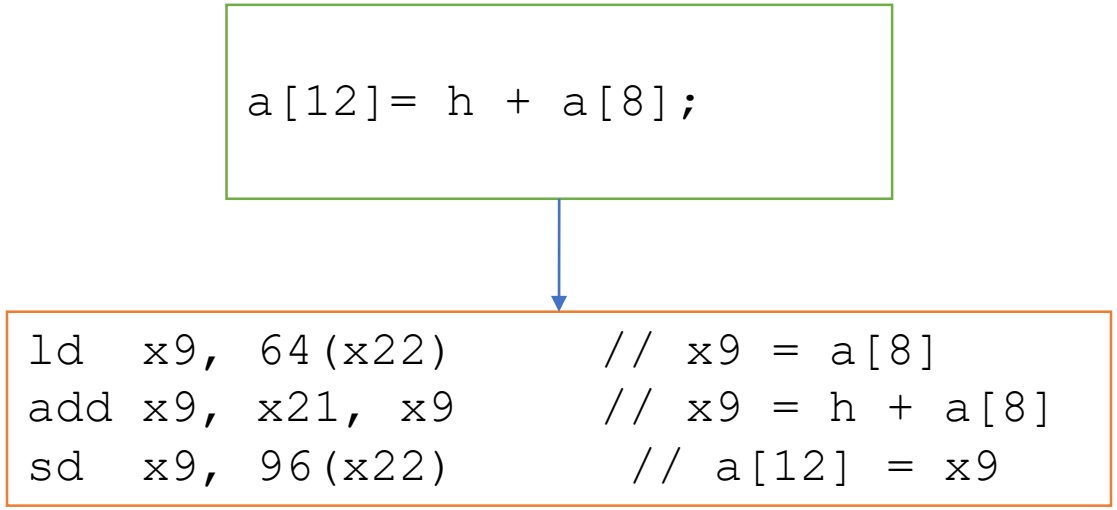
Accesso alla memoria

- Riguardiamo ancora l’esempio visto a lezione

```
a[12] = h + a[8];
```

Traduzione RISC-V

- Supponiamo che h isa in x21 e che il registro base del vettore a sia in x22



Blocchi condizionali

- Consideriamo il seguente blocco

```
if (i == j)
     f = g + h;
else
     f = g – h;
```

Traduzione RISC-V

- Supponendo di avere f, g, h, i, j nei registri da x19 a x23 avremo

```
if (i == j)
        f = g + h;
else
        f = g - h;
```



```
    bne x22, x23, L2     // se x22 neq x23 vai a L2
    add x19, x20, x21    // x19 = g + h
    beq x0, x0, L3       // se x0 == x0 vai a L3
L2:
    sub x19, x20, x21    // v0 = g - h
L3:
```

Condizione con disuguaglianza

- Supponiamo ora di avere:

```
if (i < j)
     f = g + h;
else
     f = g – h;
```

Traduzione RISC-V

- Supponendo di avere f, g, h, i, j nei registri da X19 a x23 avremo

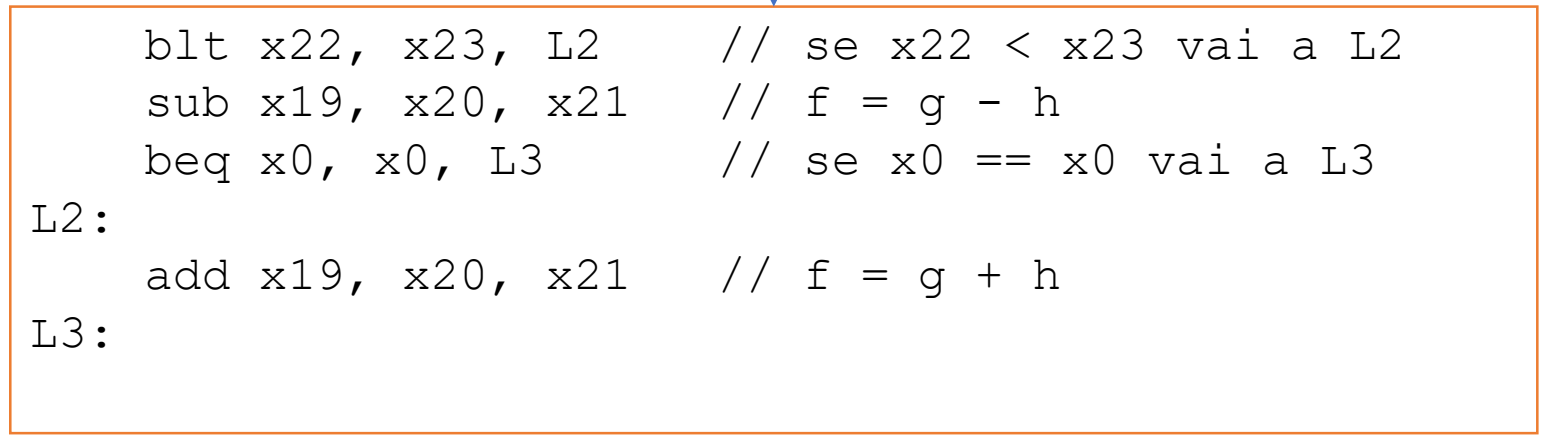
```
if (i < j)
        f = g + h;
else
        f = g - h;
```

```
    slt x5, x22, x23     // x5 = x22 < x23
    beq x5, x0, L2       // se x5 eq x0 vai a L2
    add x19, x20, x21    // f = g + h
    beq x0, x0, L3       // se x0 == x0 vai a L3
L2:
    sub x19, x20, x21    // f = g - h
L3:
```

Traduzione RISC-V (v2)

- Supponendo di avere f, g, h, i, j nei registri da X19 a x23 avremo

```
if (i < j)
        f = g + h;
else
        f = g - h;
```



```
    blt x22, x23, L2     // se x22 < x23 vai a L2
    sub x19, x20, x21    // f = g - h
    beq x0, x0, L3       // se x0 == x0 vai a L3
L2:
    add x19, x20, x21    // f = g + h
L3:
```

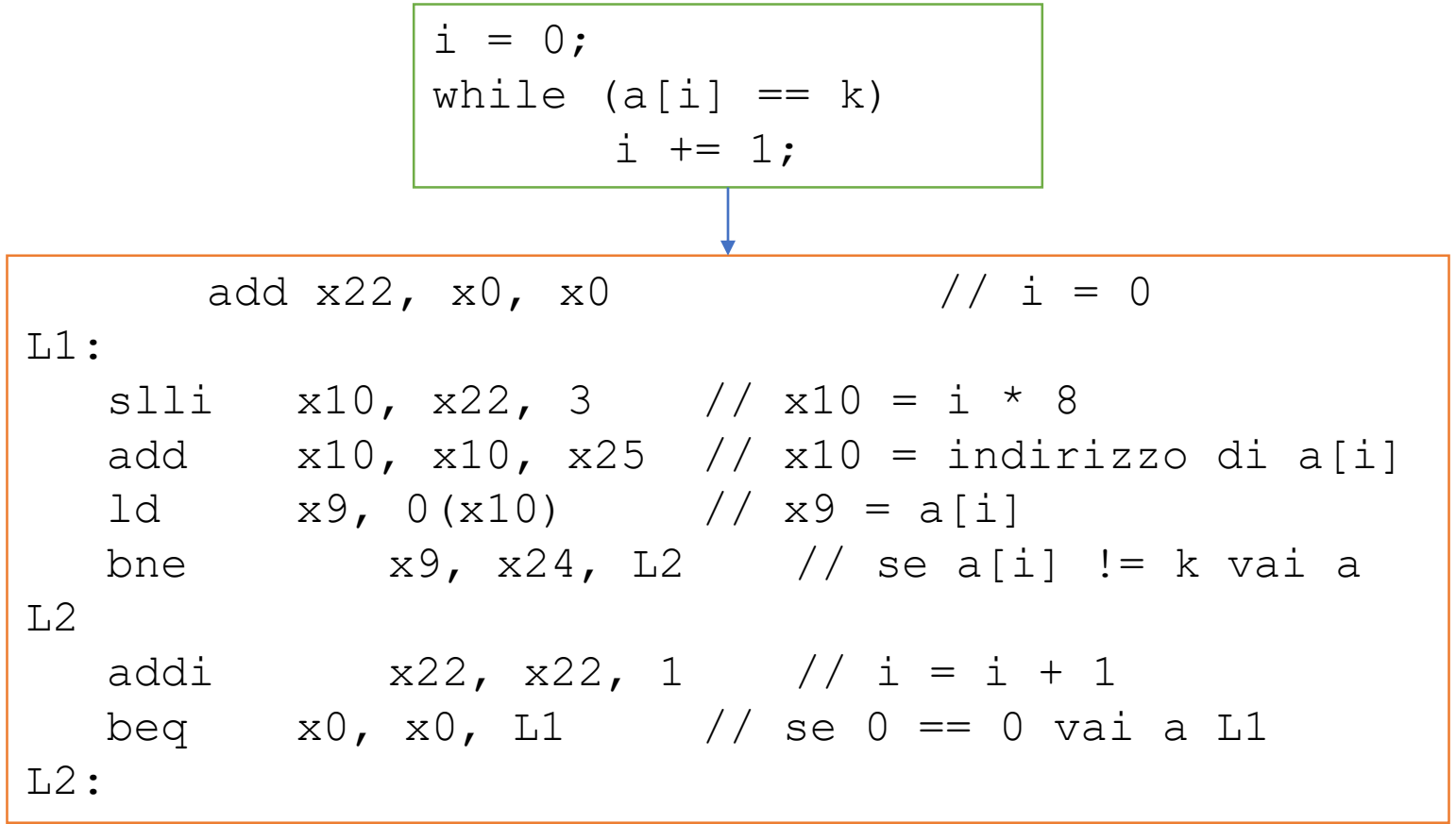
Ciclo while

- Consideriamo il seguente ciclo while

```
i = 0;
while (a[i] == k)
     i += 1;
```

Traduzione RISC-V

- Supponendo di tenere i in x22, k in x24 e l'indirizzo base di a sia in x25



```
i = 0;
while (a[i] == k)
        i += 1;
```

```
        add x22, x0, x0                 // i = 0
L1:
    slli    x10, x22, 3     // x10 = i * 8
    add     x10, x10, x25   // x10 = indirizzo di a[i]
    ld      x9, 0(x10)      // x9 = a[i]
    bne         x9, x24, L2     // se a[i] != k vai a
L2
    addi        x22, x22, 1     // i = i + 1
    beq     x0, x0, L1      // se 0 == 0 vai a L1
L2:
```

Funzione Foglia

- Si definisce “foglia” una funzione che non ne chiama altre.
- Se non facciamo delle ottimizzazioni andremo a salvare (prologo) e ripristinare (epilogo) registry in maniera poco furba

Esempio

```
typedef long long int int64;
int64 esempio foglia(int64 g, int64 h,
                     int64 i , int64 j) {
    int64 f;
    f = (g + h) - (i + j);
    return f ;
}
```

- Abbiamo una sola variabile locale (f) per la quale è possibile usare un registro

Traduzione RISC-V

- Traduzione tenendo conto che g,h, i, j corrispondono ai registri da x10 a x13, mentre f corrisponde a x20

```
int64 esempio foglia(int64 g, int64 h,
                              int64 i , int64 j) {
     int64 f;
     f = (g + h) - (i + j);
     return f ;
}
```

```
esempio_foglia:
   addi sp, sp, -24  // aggiornamento stack per fare posto a tre elementi
   sd   x5, 16(sp)   // salvataggio x5 per usarlo dopo
   sd   x6, 8(sp)    // salvataggio x6 per usarlo dopo
   sd   x20, 0(sp)   // salvataggio x20 per usarlo dopo
   add  x5, x10, x11 // x5 = g + h
   add  x6, x12, x13 // x6 = i + j
   sub  x20, x5, x6  // f = (g+h)- (i+j)
   addi x10, x20, 0  // restituzione di f (x10 = x20 + 0)
   ld   x20, 0(sp)   // ripristino x20 per il chiamante
   ld   x6, 8(sp)    // ripristino x5 per il chiamante
   ld   x5, 16(sp)   // ripristino x5 per il chiamante
   addi sp, sp, 24   // aggiornamento sp con eliminazione tre elementi
   jalr x0, 0(x1)    // ritorno al programma chiamante
```

Traduzione RISC-V Ottimizzata

- Traduzione tenendo conto che g,h, i, j che I registri temporanei non devono essere salvaguardati

```
long long int esempio foglia(long long int g, long long int h,
                             long long int i , long long int j) {
     long long int f;
     f = (g + h) - (i + j);
     return f ;
}
```

```
esempio_foglia:
   add  x5, x10, x11 // x5 = g + h
   add  x6, x12, x13 // x6 = i + j
   sub  x7, x5, x6  // f = (g+h)- (i+j)
   addi x10, x7, 0  // restituzione di f (x10 = x7 + 0)
   jalr x0, 0(x1)    // ritorno al programma chiamante
```

RISC-V Nomi dei Registri ed uso

Registro	Nome	Uso	Chi salva
x0	zero	Costante 0	N.A.
x1	ra	Indirizzo di ritorno	Chiamante
x2	sp	Stack pointer	Chiamato
x3	gp	Global pointer	---
x4	tp	Thread pointer	---
x5-x7	t0-t2	Temporanei	Chiamante
x8	s0/fp	Salvato/Puntatore a frame	Chiamato
x9	s1	Salvato	Chiamato
x10-x11	a0-a1	Argomenti di funzione/valori restituiti	Chiamante
x11-x17	a2-a7	Argomenti di funzione	Chiamante
x18-x27	s2-s6	Registri salvati	Chiamato
x28-x31	t3-t6	Temporanei	Chiamante

Traduzione RISC-V Traduzione gcc

- La traduzione gcc è un po' più articolata, e usa nomi mnemonic per I registri

```
esempio_foglia:
 addi    sp, sp, -64
         sd      ra, 56(sp)          //salvataggio ra (x1)
         sd      s0, 48(sp)          //slavataggio s0, cioè fp (x8)
         addi    s0, sp, 64
         sd      a0, -24(s0)         //salvo registro FP
         sd      a1, -32(s0)
         sd      a2, -40(s0)
         sd      a3, -48(s0)
         ld      a0, -24(s0)
         ld      a1, -32(s0)
         add     a0, a0, a1
         ld      a1, -40(s0)
         ld      a2, -48(s0)
         add     a1, a1, a2
         sub     a0, a0, a1
         sd      a0, -56(s0)
         ld      a0, -56(s0)
         ld      s0, 48(sp)
         ld      ra, 56(sp)
         addi    sp, sp, 64
         ret
```

Traduzione RISC-V Traduzione gcc (O2)

- La traduzione gcc diventa più ragionevole richiedendo di ottimizzare

```
esempio_foglia:
    add     a0, a0, a1
    sub     a0, a0, a2
     sub     a0, a0, a3
     ret                                //Macro per jlr x0, 0(ra)
```

Funzioni non foglia

- Consideriamo il seguente caso più complesso

```
int64 inc(int64 n)
{
    return n + 1;
}
int64 f(int64 x)
{
    return inc(x) – 4;
}
```

Traduzione RISC-V (gcc)

- La traduzione di inc è simile alla precedente traduzione, supponendo che n è in x10 e il risultato va in x10

```
long long int inc(long long int n) {
    return n + 1;
}
```



```
inc:
        addi    sp, sp, -32     //Spazio nello stack
        sd      x1, 24(sp)      //Salvo indirizzo di ritorno
        sd      x8, 16(sp)      //SSalvo Frame pointer
        addi    x8, sp, 32      //Frame Pointer = inizio frame attivazione
        sd      x10, -24(x8)    //Salvo X10
        ld      x10, -24(x8)    //Ripristino x10
        addi    x10, x10, 1     //Incremento x10 per predisporre il ritorno
        ld      x10, 16(sp)     //Ripristino Fram pointer
        ld      x10, 24(sp)     //Ripristino return address
        addi    sp, sp, 32      //Ripulisco Stack
        ret
```

Traduzione RISC-V

- La traduzione di inc con ottimizzazioni è la seguente

```
long long int inc(long long int n) {
    return n + 1;
}
```



```
inc:
    addi x10, x10, 1
    jalr x0, 0(x1)    // ritorno al programma chiamante
```

Traduzione RISC-V

- Vediamo ora come il gcc traduce la funzione non foglia (senza ottimizzazioni)

```
long long int f(long long int n) {
    return inc(n) - 4;
}
```

```
f:
        addi    sp,sp,-32   // estendiamo lo stack
        sd      ra,24(sp)   // salviamo ra
        sd      s0,16(sp)   // salviamo contenuto di s0 (alias per x8)
        addi    s0,sp,32    // nuovo x8 = sp + 32
        sd      a0,-24(s0)  // salvo in s0 - 24 contenuto di a0 (aka x10, n)
        ld      a0,-24(s0)  // carico in a0 (X10) il contenuto di s0 - 24
        call    inc         // Equivale a jal x1, inc
        mv      a5,a0       // copia risultato a0 della call in a5
        addi    a5,a5,-4    // decrementa risultato di 4
        mv      a0,a5       // copia risultato nel registro di ritorno a0
        ld      ra,24(sp)   // recuperiamo ra
        ld      s0,16(sp)   // recuperiamo s0
        addi    sp,sp,32    // rilasciamo lo stack
        ret                // ritorniamo a ra
```

Traduzione RISC-V

- Con il gcc le cose sono un pò più complesse e vengono fatte più operazioni (con O1)

```
long long int f(long long int n) {
    return inc(n) - 4;
}
```



```
f:
        addi    sp, sp, -16     //Spazio nello stack
        sd      ra, 8(sp)       //Memorizzo return address
        call    inc             //jal x1, inc
        addi    a0, a0, -4      //Decremento a0 (x10)
        ld      ra, 8(sp)       //Ripristino ra (x1)
        addi    sp, sp, 16      //Dealloco stack
        ret
```

Ordinamento di array

- Passiamo a qualcosa di più complesso. Un algoritmo noto come insert sort

```
void sposta(int v[], size_t i) {
  size_t j;
  int appoggio;

  appoggio = v[i];
  j = i - 1;
  while ((j >= 0) && (v[j] > appoggio)) {
    v[j+1] = v[j];
    j = j-1;
  }
  v[j+1] = appoggio;
}
```

```
void ordina(int v[], size_t n) {
   size_t i;
   i = 1;
   while (i < n) {
      sposta(v, i);
      i = i+1;
   }
}
```

Ordinamento di array

- Passiamo a qualcosa di più complesso. Un algoritmo noto come insert sort

```
void sposta(int v[], size_t i) {
  size_t j;
  int appoggio;

  appoggio = v[i];
  j = i - 1;
  while ((j >= 0) && (v[j] > appoggio)) {
    v[j+1] = v[j];
    j = j-1;
  }
  v[j+1] = appoggio;
}
```

```
void ordina(int v[], size_t n) {
   size_t i;
   i = 1;
   while (i < n) {
      sposta(v, i);
      i = i+1;
   }
}
```

Traduzione RISC-V

- Cominciamo da sposta. Stavolta le cose sono più complesse. Assumiamo che i parametri siano memorizzati in x10, x11 rispettivamente. Usiamo x5, x6, x28 per j e appoggio.

```
void sposta(int v[], size_t i) {
  size_t j;
  int appoggio;

  appoggio = v[i];
  j = i - 1;
```



```
sposta:
        slli    a4,a1,2             //a4 = i*4
        add     a5,a0,a4            //a5 = &v[i]
        lw      a3,0(a5)            //a3 = v[i]
        addiw   a1,a1,-1  //a1 = a1-1 (i = j-1)
```

Traduzione RISC-V continua

- Ciclo

```
while ((j >= 0) && (v[j] > appoggio)) {
   v[j+1] = v[j];
   j = j-1;
 }
```

```
        bltz    a1,.L2                  //se j < 0 esci dal ciclo
        lw      a4,-4(a5)               // a4 = v[i-1]=v[j]
        bge     a3,a4,.L2       //se appoggio è >= v[j] esci
        li      a2,-1                   //carica -1 in a2
.L3:
        sw      a4,0(a5)                //Memorizza v[j] (a4) in v[j+1)
        addiw   a1,a1,-1        //a1 = a1-1
        beq     a1,a2,.L4       // salta se a1 = -1
        addi    a5,a5,-4                // j=j-1
        lw      a4,-4(a5)
        bgt     a4,a3,.L3               //Salta se v[j] > appoggio
        j       .L2
```

Traduzione RISC-V continua

- Uscita da sposta

```
  v[j+1] = appoggio;
}
```

```
.L4:
        li      a1,-1
.L2:
        addi    a1,a1,1
        slli    a1,a1,2
        add     a1,a0,a1
        sw      a3,0(a1)
        ret
```

Traduzione RISC-V continua

- Passiamo ora alla funzione ordina. Assumiamo che i parametri siano memorizzati in x10, x11 rispettivamente

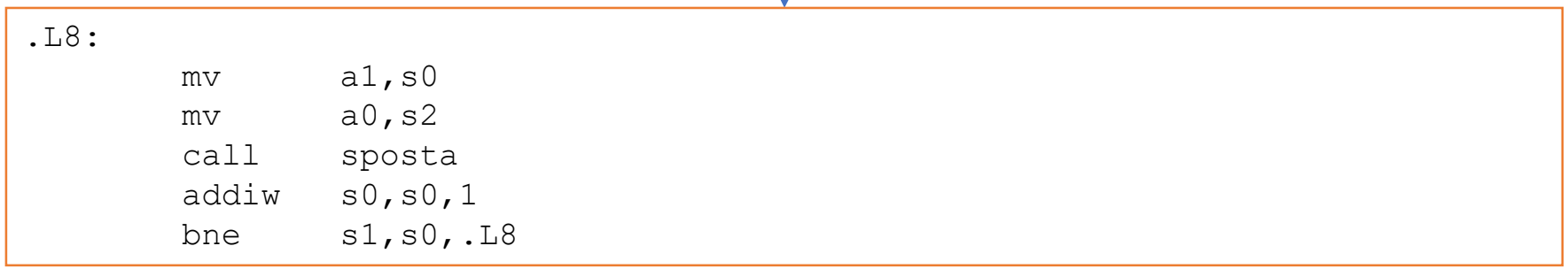
```
void ordina(int v[], size_t n) {
  size_t i;
  i = 1;
```

```
        li      a5,1
         ble     a1,a5,.L11
         addi    sp,sp,-32
         sd      ra,24(sp)
         sd      s0,16(sp)
         sd      s1,8(sp)
         sd      s2,0(sp)
         mv      s1,a1
         mv      s2,a0
         li      s0,1
```

Traduzione RISC-V continua

- Passiamo al loop

```
while (i < n) {
   sposta(v, i);
   i = i+1;
}
```



```
.L8:
        mv      a1,s0
        mv      a0,s2
        call    sposta
        addiw   s0,s0,1
        bne     s1,s0,.L8
```

Traduzione RISC-V continua

- Epilogo ordina

```
  ld      ra,24(sp)
        ld      s0,16(sp)
        ld      s1,8(sp)
        ld      s2,0(sp)
        addi    sp,sp,32
        jr      ra
.L11:
        ret
```

Copia Stringhe

- Consideriamo ora

```
void copia_stringa(char d[], const char s[]) {
  size_t i = 0;

  while ((d[i] = s[i]) != `0`) {
    i += 1;
  }
}
```

Traduzione RISC-V

- Traduzione RISC-V

```
void copia_stringa(char d[], const char s[]) {
  size_t i = 0;
```



```
copia_stringa:
    addi sp, sp, -8      // aggiorna stack per inserire 1 element
    sd x19, 0(sp)        // salva x19
    add x19, x0, x0      // i = 0
```

Traduzione RISC-V continua

- Loop

```
while ((d[i] = s[i]) != `0`) {
  i += 1;
}
```



```
LoopCopiaStringa:
    add x5, x19, x11                  // indirizzo di s[i]
    lbu x6, 0(x5)                     // x6 = s[i]
    add x7, x19, x10                  // indirizzo di d[i]
    sb x6, 0(x7)                      // d[i] = s[i]
    beq x6, x0, LoopCopiaStringaEnd   // se`0` vai a LoopCopiaStringaEnd
    addi x19, x19, 1                  // i += 1
    jal x0, LoopCopiaStringa          // salta a LoopCopiaStringa
LoopCopiaStringaEnd
```

Traduzione RISC-V continua

- Chiusura

```
LoopCopiaStringaEnd
    ld x19, 0(sp)    // ripristina contenuto di x19
    addi sp, sp, 8   // aggiorna lo stack eliminando un element
    jral, x0, 0(x1)  // ritorna al chiamante
```