



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Esempi di Programmi Assembly

RISC-V e Intel x86

Giovanni Iacca

(materiale preparato con Luigi Palopoli,
Marco Roveri, e 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 Intel
- Successivamente passeremo all'assembly ARM



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

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 si voglia mettere il risultato in x23, la traduzione è semplicemente

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

```
add x5, x19, x20  
add x6, x21, x22  
sub x23, x5, x6
```



Traduzione RISC-V (v2)

- Supponendo che g, h, i, j siano in x19, x20, x21, e x22, e che si voglia mettere il risultato 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 è usato
un registro in meno:
Il risultato intermedio è
memorizzato in x23



Traduzione INTEL

- Per INTEL, supponiamo che g, h, i, j siano in rdi, rsi, rdx, rcx e che si voglia salvare il risultato in rax
- Il problema è come fare la somma a due operandi e risultato in un terzo operando, cosa possibile usando l'istruzione lea

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

```
leaq (%rdi, %rsi), %rax    //rax = rdi + rsi
addq %rcx, %rdx            //rdx = rdx + rcx
subq %rdx, %rax            //rax = rax - rdx
```



Accesso alla memoria

- Riguardiamo ancora l'esempio visto a lezione assumendo `int a[]` e `int h`

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



Traduzione RISC-V

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

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

`lw x9, 32(x22) // x9 = a[8]`
`addw x9, x21, x9 // x9 = h + a[8]`
`sw x9, 48(x22) // a[12] = x9`



Traduzione INTEL

- Supponiamo di avere h in edi e l'indirizzo di a in rsi.
- Grazie al fatto di poter avere operandi in memoria stavolta ce la caviamo con due istruzioni

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



```
addl 32(%rsi), %edi    //edi = edi + a[8]  
movl %edi, 48(%rsi)    //a[12] = edi
```



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    // x19 = g - h
L3:
    ...
```



Traduzione INTEL

- Assumiamo che g, h, i, j siano in rdi, rsi, rdx, rcx e che si voglia salvare f in rax

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

```
cmpq  %rcx, %rdx          // i == j??
jne   L2
leaq  (%rdi, %rsi), %rax   // f = g + h
jmp   L3

L2:
movq  %rdi, %rax          // f = g
subq  %rsi, %rax          // f = f-h (nota complicazione)


L3:
...
```



Traduzione INTEL (ottimizzata)

- Possiamo fare di meglio usando uno strano oggetto (move condizionale) che fino ad ora i compilatori avevano evitato di usare

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



```
leaq    (%rdi, %rsi), %rax    //rax = g+h
subq    %rsi, %rdi           //rdi = g-h
cmpq    %rcx, %rdx
cmovne  %rdi, %rax           //spostiamo se il cmp precedente NE
```



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:
    ...
```




Traduzione INTEL

- Assumiamo che g, h, i, j siano in rdi, rsi, rdx, rcx e che si voglia salvare f in rax

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




```
cmpq  %rcx, %rdx          // i == j??
jge   L2
leaq  (%rdi, %rsi), %rax   // f = g + h
jmp   L3
L2:
movq  %rdi, %rax           // f = g
subq  %rsi, %rax           // f = f-h (nota complicazione)
L3:
...
```



Traduzione INTEL (ottimizzato)

- Ancora una volta possiamo usare la move condizionale

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



```
leaq    (%rdi, %rsi), %rax    //rax = g+h
subq    %rsi, %rdi            //rdi = g-h
cmpq    %rcx, %rdx
cmovge  %rdi, %rax            //spostiamo se il cmp precedente GE
```



Ciclo while

- Consideriamo il seguente ciclo while

```
i = 0;
```

```
while (a[i]==k)
```

```
    i += 1;
```



Traduzione RISC-V

- Supponendo di avere 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                // i = 0  
L1:  
    slli    x10, x22, 2     // x10 = i * 4  
    add     x10, x10, x25   // x10 = indirizzo di a[i]  
    lw      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:  
    ...
```



Traduzione INTEL

- Stavolta è possibile sfruttare la potenza del CISC

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



```
    cmpl    (%rsi), %edi    // %rsi=&a, %edi=k  
    jne     L2              // ciclo mai eseguito  
    movq    $0, %rax        // i = 0  
L1:                                       
    addq    $1, %rax        // i++  
    cmpl    %edi, (%rsi, %rax, 4) // k ed a 32 bit  
    je      L1  
    jmp     L3  
L2: movq    $0, %rax  
L3:                                       
    ...
```



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

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
x12-x17	a2-a7	Argomenti di funzione	Chiamante
x18-x27	s2-s11	Registri salvati	Chiamato
x28-x31	t3-t6	Temporanei	Chiamante



Intel Nomi dei Registri ed Uso

- %rsp → stack pointer;
- %rbp → base pointer
- Primi 6 argomenti:
 - %rdi, %rsi, %rdx, %rcx, %r8 ed %r9
- Altri argomenti (7 → n): sullo stack
- Valori di ritorno:
 - %rax e %rdx
- Registri preservati:
 - %rbp, %rbx, %r12, %r13, %r14 ed %r15
- Registri non preservati:
 - %rax, %r10, %r11
 - registri per passaggio parametri: %rdi, %rsi, %rdx, %rcx, %r8 ed %r9

63	31	15	8	7	0
%rax	%eax	%ax	%ah	%al	
%rbx	%ebx	%bx	%bh	%bl	
%rcx	%ecx	%cx	%ch	%cl	
%rdx	%edx	%dx	%dh	%dl	
%rsi	%esi	%si		%sil	
%rdi	%edi	%di		%dil	
%rbp	%ebp	%bp		%bpl	
%rsp	%esp	%sp		%spl	
%r8	%r8d	%r8w		%r8b	
%r9	%r9d	%r9w		%r9b	
⋮					
%r15	%r15d	%r15w		%r15b	



Funzione Foglia

- Si definisce “foglia” una funzione che non ne chiama altre.
- Le funzioni foglia nel RISC-V, se non ottimizzate, sono trattate come qualunque altra funzione
 - occorre salvare (prologo) il return address e gestire i registri usati come parametri, e ripristinare (epilogo) tutto quello salvato



Esempio

```
int esempio_foglia(int g, int h,  
                  int i , int j) {  
  
    int 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

```
int esempio_foglia(int g, int h, int i, int j) {  
    int 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  
    addw x5, x10, x11 // x5 = g + h  
    addw x6, x12, x13 // x6 = i + j  
    subw 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 corrispondono ai registri da x10 a x13 (aka a0, a1, a2, a3), e che i temporanei possono essere non salvati/usati.

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



```
esempio_foglia:  
    addw    a0, a0, a1    // f = g+h (a0 corrisponde a x10)  
    addw    a3, a2, a3    // a3 = i+j  
    subw    a0, a0, a3    // f = f - a3  
    ret      // alias per jalr x0, 0(x1) o jalr zero, 0(ra)
```



Traduzione INTEL

- Traduzione ottimizzata

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



```
esempio_foglia:  
    leal    (%rdi,%rsi), %eax  
    addl    %ecx, %edx  
    subl    %edx, %eax  
    ret
```



Traduzione non ottimizzata

Senza ottimizzazioni il risultato è piuttosto diverso

esempio_foglia:

```
    pushq    %rbp                                //Prologo
    movq     %rsp, %rbp
    movl     %edi, -20(%rbp)                      //g
    movl     %esi, -24(%rbp)                      //h
    movl     %edx, -28(%rbp)                      //i
    movl     %ecx, -32(%rbp)                      //j

    movl     -20(%rbp), %edx                      //edx = g
    movl     -24(%rbp), %eax                      //eax = h
    leal     (%rdx,%rax), %ecx                    //f = ecx = g+h
    movl     -28(%rbp), %edx                      //edx = i
    movl     -32(%rbp), %eax                      //eax = j
    addl     %edx, %eax                          //eax = i+j
    subl     %eax, %ecx                          //ecx = g+h - (i+j)
    movl     %ecx, %eax                          //eax = ecx

    movl     %eax, -4(%rbp)                       //Epilogo
    movl     -4(%rbp), %eax
    popq     %rbp
    ret
```



Funzioni non foglia

- Consideriamo il seguente caso più complesso

```
int inc(int n)
{
    return n + 1;
}
int f(int x)
{
    return inc(x) - 4;
}
```



Traduzione RISC-V

- La traduzione di `inc` è simile alla precedente traduzione, supponendo che `n` è in `x10` (aka `a0`) e risultato in `x10` (aka `a0`)

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



```
inc:  
    addiw    a0, a0, 1  
    ret
```



Traduzione RISC-V

- La traduzione di `f` richiede più attenzione. Supponiamo anche qui che `n` sia in `x10 (a0)`

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



```
f:  
    addi    sp, sp, -16           //Prologo  
    sd      ra, 8(sp)  
  
    jal     ra, inc  
    addiw   a0, a0, -4  
  
    ld      ra, 8(sp)            //Epilogo  
    addi    sp, sp, 16  
    ret
```




Traduzione RISC-V

- Con il gcc le cose sono un po' più complesse e vengono fatte più operazioni (senza ottimizzazioni).

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



```
f:      addi    sp,sp,-32    // estendiamo stack  
        sd      ra,24(sp)   // salviamo ra  
        sd      s0,16(sp)   // salviamo contenuto di s0  
        addi    s0,sp,32    // nuovo s0 = s0 + 32  
        mv      a5,a0       // carico in a5 il contenuto di a0  
        sw      a5,-20(s0)  // salvo a5 sullo stack  
        lw      a5,-20(s0)  // leggo a5 dallo stack  
        mv      a0,a5       // memorizzo a5 in a0  
        call    inc        // chiamo inc  
        mv      a5,a0       // copio risultato chiamata su a5  
        addiw   a5,a5,-4    // decremento di 4 il risultato  
        mv      a0,a5       // copio risultato intermedio in a0 per ritorno  
        ld      ra,24(sp)   // ripristino ra  
        ld      s0,16(sp)   // ripristino s0  
        addi    sp,sp,32    // svuoto stack  
        jr      ra         // ritorno
```



Traduzione INTEL

- La traduzione INTEL è più semplice

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



```
inc:  
    leal    1(%rdi), %eax  
    ret
```



Traduzione INTEL

- La traduzione INTEL è più semplice poiché il salvataggio del return address è fatto in automatico con la call

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



```
f:  
    call    inc  
    subl    $4, %eax  
    ret
```



Ordinamento di array

- Passiamo a qualcosa di più complesso: un algoritmo noto come «insertion 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(a0, a1) rispettivamente. Usiamo a3 per 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 = i-1)
```



Traduzione RISC-V

continua

- Ciclo

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

↓

```
.L3:  
    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 >= v[j] esci  
    li      a2, -1           // carica -1 in a2  
  
    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)        // a4 = v[j]  
    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)    // v[j+1] = appoggio  
    ret
```




Traduzione RISC-V

continua

- Passiamo ora alla funzione `ordina`. I parametri sono memorizzati in `a0`, `a1` rispettivamente.

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



```
ordina: 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
```



- Riguardiamo lo stesso codice implementato tramite INTEL

```
void sposta(int v[], size_t i) {  
    size_t j;  
    int appoggio;  
  
    appoggio = v[i];  
    j = i - 1;
```



```
sposta:    # rdi = v, rsi = i  
           # rax = j, r10d = appoggio  
    movq %rsi, %rax  
    movl (%rdi, %rax, 4), %r10d # appoggio = v[i]  
    dec %rax
```



- Vediamo il ciclo

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



```
ciclo:  
    cmpq $0, %rax          # confronta 0 e rax  
    jl  out                # esci se j < 0  
    movl (%rdi, %rax, 4), %r11d # metti v[j] in %r11d  
    cmpl %r10d, %r11d      # confronta v[j] e appoggio  
    jle out                # se v[j] < appoggio esci  
    movl %r11d, 4(%rdi, %rax, 4)  
    dec %rax  
    jmp ciclo  
out:
```



- Uscita da sposta

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



```
out:  
    movl %r10d, 4(%rdi, %rax, 4)  
    ret
```



- Vediamo la procedura ordina, che non è foglia.
- Il salvataggio sullo stack è semplificato

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



```
ordina:                                # rdi = v, rsi = n  
                                       # rbx = i  
    pushq %rbx  
    movq $1, %rbx
```



- Il loop

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



```
loop_ordina:  
    cmp %rbx, %rsi  
    jle out_ordina  
    pushq %rsi  
    movq %rbx, %rsi  
    call sposta  
    popq %rsi  
    inc %rbx  
    jmp loop_ordina  
out_ordina
```



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

INTEL

- Epilogo

```
out_ordina  
    popq %rbx  
    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 un elemento  
    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 elemento  
    jalr, x0, 0(x1)   // ritorna al chiamante
```



Traduzione INTEL

- Inizio

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

```
copia_stringa:  
    movzbl (%rsi), %eax  
    movb %al, (%rdi)  
    testb %al, %al  
    je L1  
    movl $0, %eax
```



Traduzione INTEL

- Loop

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

```
L3:  
    addl $1, %eax  
    movslq %eax, %rcx  
    movzbl (%rsi, %rcx), %edx  
    movb %dl, (%rdi, %rcx)  
    testb %dl, %dl  
    jne L3  
L1:  
    ret
```