



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Esempi di Programmi Assembly RISC-V e ARM

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
- Abbiamo visto esempi in assembly RISC-V e Intel
- In questa lezione rivedremo esempi RISC-V e corrispondenti esempi in ARM



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



ARM Nomi dei Registri ed USO

- Nomi: da r0 a r15
 - Tecnicamente, r15 non è un registro general purpose, spesso usato come PC
- Alcuni registri accessibili tramite un nome simbolico che ne identifica l'utilizzo
 - r13 == sp (stack pointer)
 - r14 == lr (link register)
- Primi 4 argomenti:
 - r0 ... r3
 - registri non preservati! Utilizzabili anche come registri "temporanei" da non salvare!
- Altri argomenti (4 → n): sullo stack
- Registri preservati: r4 ... r11
 - Eccezione: in alcuni ABI r9 non è preservato
- Valori di ritorno: r0 e r1
- I registri che una subroutine può utilizzare senza doverli salvare sono
 - r0, r1, r2, r3 ed r12
 - più eventualmente r9 (dipende da piattaforma / ABI)



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 ARM

- Traduzione in ARM pressoché uguale
 - eccetto per il nome dei registri
- La s dopo la add è per settare i flag
 - funzionerebbe anche senza

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

`adds r1, r0, r1
adds r3, r2, r3
subs r0, r1, r3`



Accesso alla memoria

- Riguardiamo ancora l'esempio visto a lezione assumendo che `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 ARM

- Anche in questo caso la traduzione è molto simile
 - Assumiamo di avere h in r0 e indirizzo di a in r1
- Si usa indirizzamento pre-indexed senza aggiornamento della base (senza !)

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

```
ldr    r3, [r1, #32]  
add    r0, r3, r0  
str    r0, [r1, #48]
```



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 ARM

- Questa volta diventa tutto più semplice per via delle istruzioni condizionali

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

```
cmp    r2, r3
addeq   r0, r0, r1
subne   r0, r0, r1
```



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 ARM

- Nel caso di ARM la traduzione con le istruzioni condizionali è simile
- Cambiano solo le condizioni... (lt e ge)

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



```
cmp     r2, r3
addlt   r0, r0, r1
subge   r0, r0, r1
```



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          // 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 ARM

- Assumendo che il valore di k sia inizialmente contenuto in $r0$, che l'indirizzo dell'array a sia inizialmente contenuto in $r1$ e che il valore di i vada salvato in $r0$
- Con ARM è possibile usare il pre-indexing per scorrere array
- Il codice è

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

```
ldr    r3, [r1]  
cmp    r0, r3  
bne    L2  
mov    r3, #0  
L1:    add    r3, r3, #1  
ldr    r2, [r1, #4]!  
cmp    r2, r0  
beq    L1  
b      L3  
L2:    mov    r3, #0  
L3:    mov    r0, r3
```



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 il return address e gestire i registri usati come parametri), mentre in ARM sono molto più semplici da trattare
 - non occorre salvare il return address, nè avere particolari accortezze sui registri usati come parametri.
- Prologo ed epilogo quindi in ARM sono dunque semplificati e lo diventano ancora di più se non usiamo registri da preservare e variabili da allocare nello stack



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




Traduzione ARM

- Traduzione molto simile
- Come esempio (non è realmente necessario in questo caso) rsb viene usato per invertire i due operandi

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



```
esempio_foglia:  
    add    r0, r0, r1  
    add    r3, r2, r3  
    rsb    r0, r3, r0 // r0=r0-r3 (equivalente a sub r0, r0, r3)  
    bx     lr
```



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 senza ottimizzazioni

- La traduzione di `f` richiede più attenzione. Supponiamo anche qui che `n` è in `x10 (a0)` e risultato anche esso 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 ARM

- La traduzione ARM di inc è praticamente identica
- Notare che r0 è sia usato come parametro di ingresso che come valore di ritorno

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



```
inc:  
    add    r0, r0, #1  
    bx     lr
```



Traduzione ARM

- La traduzione ARM può avvalersi del salvataggio multiplo di r4 e lr (con aggiornamento automatico di sp)
- Notare come il ripristino dei registri possa permettere automaticamente di caricare lr su pc e fare il return automaticamente

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

f:

```
    stmfd    sp!, {r4, lr}  
    bl      inc  
    sub     r0, r0, #4  
    ldmfid   sp!, {r4, pc}
```

stmfd = stm full descending = stmdb



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 (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 = j-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`. Assumiamo che i parametri siano memorizzati in `x10`, `x11` (`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
```



ARM

- Riguardiamo lo stesso codice implementato tramite ARM

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

```
sposta:  
    mov    r2, r1, asl #2          // r2 = i * 4  
    add    r3, r0, r2              // r3 = &v[i]  
    ldr    ip, [r0, r1, asl #2]    // ip = r12 (scratch) appoggio = v[i]  
    subs   r1, r1, #1              // j = i - 1
```



- Vediamo il ciclo

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

↓

```
bmi    L2                // salta se j < 0  
ldr    r2, [r3, #-4]     // r2 = v[j] = v[i-1]  
cmp    ip r2  
bge    L2  
L3:  
str    r2, [r3], #-4  
sub    r1, r1, #1  
cmp    r1, #1  
beq    L2  
ldr    r2, [r3, #-4]  
cmp    ip, r2  
blt    L3
```



- Uscita da sposta

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

L2:

```
add    r1, r1, #1  
str     ip, [r0, r1, asl #2]  
bx     lr
```




ARM

- Vediamo la procedura ordina, che non è foglia.
- Il salvataggio sullo stack dei registri avviene in un solo passo

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

```
ordina:  
    cmp    r1, #1  
    bxle   lr  
    stmfd  sp!, {r4, r5, r6, lr}  // full descending aka db  
    mov    r5, r1  
    mov    r6, r0  
    mov    r4, #1
```

stmfd = stmdb



ARM

- Il loop
- Notare uscita contestuale con ripristino registri

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

ldmfd = ldmia

L8:

```
mov    r1, r4  
mov    r0, r6  
bl     sposta  
add    r4, r4, #1  
cmp    r5, r4  
bne    L8  
ldmfd  sp!, {r4, r5, r6, pc}  // full descending
```



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 ARM

- Inizio

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

```
copia_stringa:  
    ldrb r3, [r1]  
    strb r3, [r0]  
    cmp  r3, #0  
    bxeq lr
```



Traduzione ARM

- Loop
- Notare come possiamo usare l'aggiornamento del registro per evitare un registro indice.

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

L3:

```
ldrb    r3, [r1, #1]!  
strb    r3, [r0, #1]!  
cmp     r3, #0  
bne     L3  
bx      lr
```