# Lezione 25 C e assembly 3

mercoledì 29 novembre 2023 09:45

Andiamo ora a vedere le **istruzioni di classe 6 : le istruzioni di controllo del flusso :** permettono di specificare se e quando ed in quale ordine devono essere eseguite le istruzioni che compongono il sorgente : nel caso di C è molto semplice in quanto vi sono poche strutture di controllo , ma in assembler leggermente più "difficile": in quanto servono controlli di condizione , iterazioni e salti a funzioni (sotto-programmi). Vediamo un esempio :

```
.org 0x800
.data
        message: .ascii "Hello world!"
        counter: .byte 0
.text
main:
        movq $message,%rax
        .repeat:
                 cmpb $0,(%rax)
                 jz .end
                addb $1,counter
                addq $1,%rax
                jmp .repeat
        .end:
                hlt
```

Questo programma conta la lunghezza della stringa passata in input, usando un confronto (cmpb-> COMPARE BIT) ed un salto .jz( jump zero -> salta se zero ) e jmp (salto "semplice).

Andiamo a vedere come funziona in dettaglio l'istruzione di compare (CMP): unica istruzione che allineandosi con il bit di stato , permette di valutare una condizione . In dettaglio :

| | |
|---|---|
| **CMP B,E** | **B e E sono operandi . Confronto tra il valore di B ed E , calcolando B-E , e d il risultato viene scartato.** |

Vediamo un esempio :

```
if(x == 1) {
    CODE BLOCK A;
} else if(x == 2) {
    CODE BLOCK B;
} else {
    CODE BLOCK C;
}

if(TEST)
  SINGOLO STATEMENT;
else
  SINGOLO STATEMENT;
```

```
    cmpb $1, %al
    jnz .elseif
    # CODE BLOCK A
    jmp .endif
.elseif:
    cmpb $2, %al
    jnz .else
    # CODE BLOCK B
    jmp .endif
.else:
    # CODE BLOCK C
.endif:
```

Andiamo nel dettaglio : come già detto il confronto tra due variabili, si basa su una sottrazione tra i due operandi , aggiornando gli opportuni bit del registro flags : andiamo a vedere un esempio :

**A-B --> A-B=0**

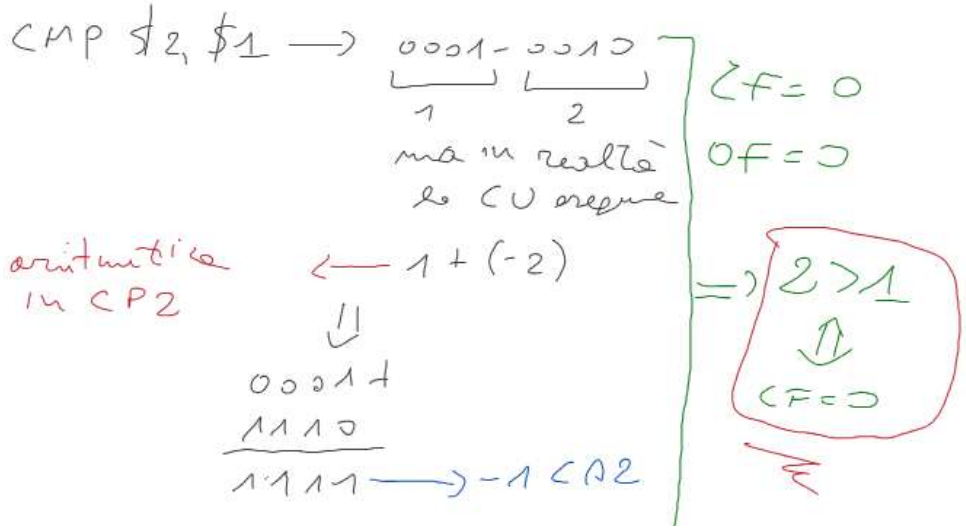
1. Se a-b>0 il risultato va bene , senza prestito aggiuntivo

2. Se a-b<0 , allora si deve "prestare un'unita a b"
3. **In generale quindi : nel caso di sottrazione la CU rileva la necessità di prestito verificando se addizione in complemento a 2 genera un riporto .**
    a. Se nell'addizione **non** c'è un riporto , allora **non** c'è prestito nella sottrazione
    b. Se nella c'è un riporto, allora non c'è prestito nella sottrazione.
4. Se nel caso invece di sub , il CF viene negato .

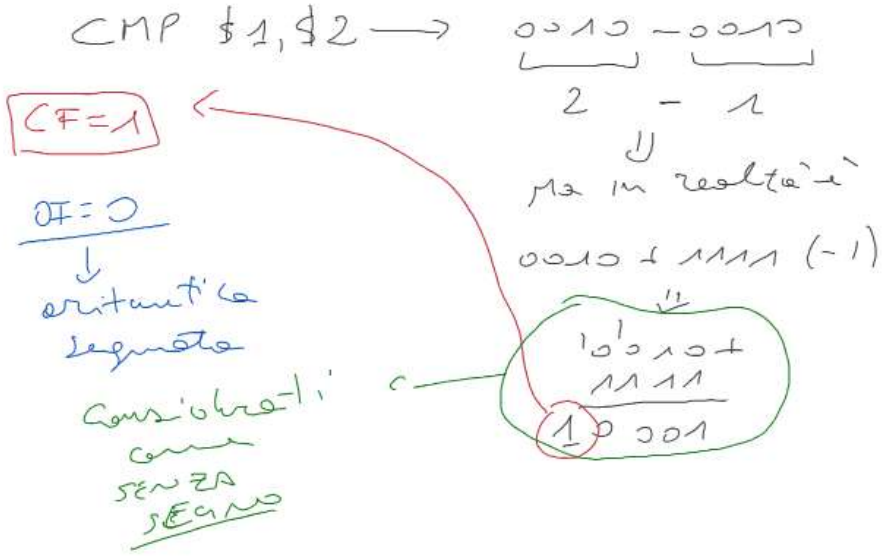
Andiamo a vedere i tipi di confronti , ovvero quelli tra **operandi senza segno** e quelli tra **operandi con segno** . Andiamo a vedere i primi : si assume che entrambi gli operandi siano >= 0 , quindi si deve controllare il valore di CF e ZF per determinare l'ordine . Quindi in modo più generale una generica istruzione **CMPX src,dst** , dove la "X" rappresenta il tipo di salto , si ha che :

| Condizione | Primo controllo | Secondo controllo |
|---|---|---|
| dest < source | CF = 1 | |
| dest ≥ source | CF = 0 | |
| dest > source | CF = 0 | ZF = 0 |
| dest = source | ZF = 1 | |
| dest ≠ source | ZF = 0 | |

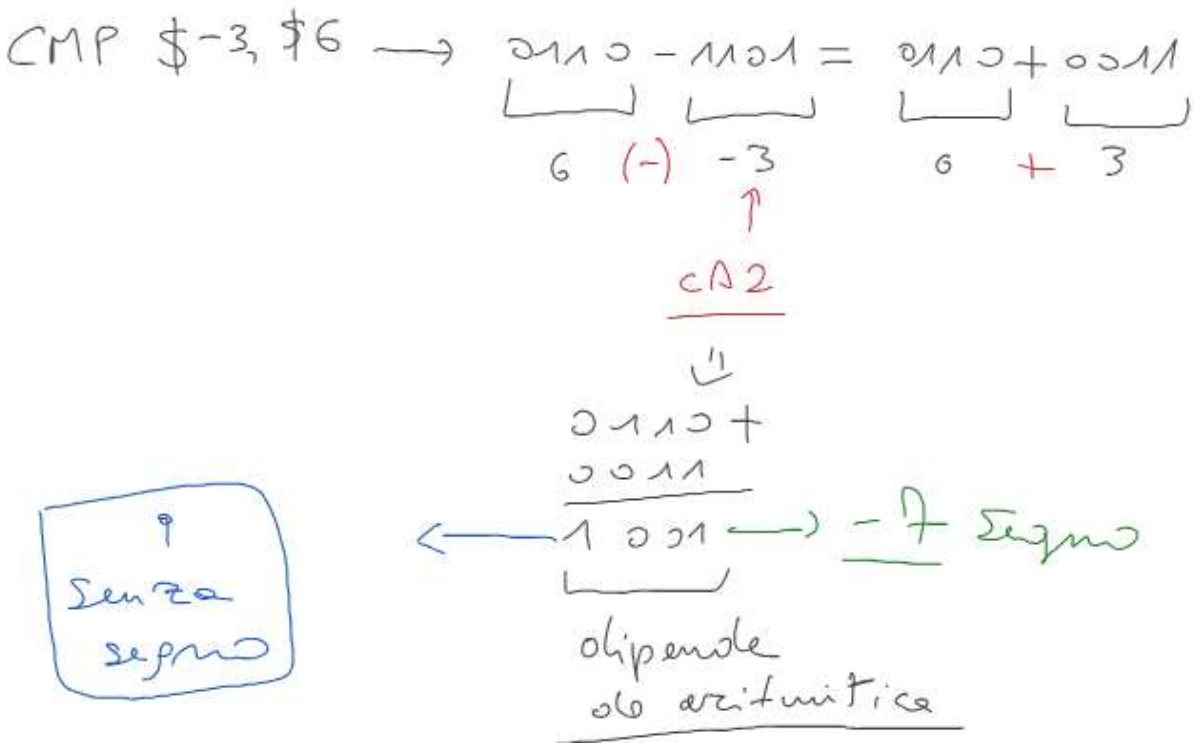
Vediamo ora due esempi : il primo vediamo se 2>1 (il che è ovvio), ma andiamo a vedere il dettaglio :




Invece il secondo esempio : 1>2 (la condizione fallisce ):




Vediamo ora i secondi : quelli in aritmetica segnata : da notare che CF in questo caso non ha in significato preciso in aritmetica a complemento a 2 , in quanto si vedono solo gli ultimi2 riporti. Quindi in generale facciamo **dst-src<0** : poniamo attenzione a SF( sign flag). Vediamo un esempio :




Da notare che se consideriamo solo quella segnata : **si ha che la somma è 9 , ma in CA2 non può essere rappresentato, quindi vi è stato overflow.** Riassumendo quindi :

| Condizione | Primo controllo | Secondo controllo |
|---|---|---|
| dest < source | SF ≠ OF | |
| dest ≥ source | SF = OF | |
| dest > source | ZF = 0 | SF = OF |
| dest = source | ZF = 1 | |
| dest ≠ source | ZF = 0 | |

Vediamo ora dal punto di vista dell'assembler:

```
.org 0x800
.data
        x: .word 3
        y: .word -2
.text
        # Imposta a 1 l'indirizzo 0x1280 solo se x > y
        # Assumo che x ed y possano assumere valori negativi
        movw x, %ax
        movw y, %bx
        cmpw %bx, %ax
        jz .nonImpostare
        js .SFset
        jo .nonImpostare # eseguita se SF = 0. Se OF = 1 allora SF != OF
        jmp .set
        .SFset:
        jno .dont # eseguita se SF = 1. Se OF = 0 allora Sf != OF
        .set:
        movb $1, 0x1280
        .dont:
        hlt
```

Dopo questa intro, andiamo ora a vedere la classe 6 : **attenzione al tipo di operando coinvolti** . Questa classe si occupa delle istruzioni di controllo condizionale del flusso : in base ad una certa condizione nel programma, devo poter alterare il flusso del programma. Per fare questa alterazione del flusso , **si devono vedere i bit nel registro flag** . Vediamo ora tutti i tipi di salto :

| Istruzione | Descrizione | Aritmetica | Condizione controllata |
|---|---|---|---|
| jb | Jump if below | non segnata | CF = 1 |
| jnae | Jump if not above or equal | | |
| jnb | Jump if not below | non segnata | CF = 0 |
| jae | Jump if above or equal | | |
| jbe | Jump if below or equal | non segnata | CF = 1 o ZF = 1 |
| jna | Jump if not above | | |
| ja | Jump if above | non segnata | CF = 0 e ZF = 0 |
| jnbe | Jump if not below or equal | | |
| jl | Jump if less | segnata | SF ≠ OF |
| jnge | Jump if not greater or equal | | |
| jge | Jump if greater or equal | segnata | SF = OF |
| jnl | Jump if not less | | |
| jle | Jump if less or equal | segnata | ZF = 1 o SF ≠ OF |
| jng | Jump if not greater | | |
| jg | Jump if greater | segnata | ZF = 0 e SF = OF |
| jnle | Jump if not less or equal | | |

Andiamo a vedere ora il codice per implementare i salti :

1. **Switch** :
   a.

```
switch (OPERAND) {
   case CONSTANT:
      CODE;
      break;
   default:
      CODE;
   }
}
```

```
movl var, %eax
shll $3, %eax
movq branchTable(%rax), %rax
jmp *%rax
```

   b. In generale :
      i.

```
switch (age) {
  case 1:  printf("You're one.");            break;
  case 2:  printf("You're two.");            break;
  case 3:  printf("You're three."); __attribute__((fallthrough));
  case 4:  printf("You're three or four."); break;
  default: printf("You're not 1, 2, 3 nor 4!");
}
```

2. **While** :
   a.
   b.

```
while(TEST) {
   CODE;
}
```

```
while(TEST) {
   if(OTHER_TEST) {
      continue;
   }
   CODE;
}
```

```
while(TEST) {
   if(OTHER_TEST) {
      break;
   }
   CODE;
}
```

```
.test:
    cmpb ...
    jnz .skip
    # <codice>
    jmp .test
.skip:
```

3. **Do/while - for :**

```
do {
    CODE;
} while(TEST);
```

```
for(INIT; TEST; POST) {
    CODE;
}
```

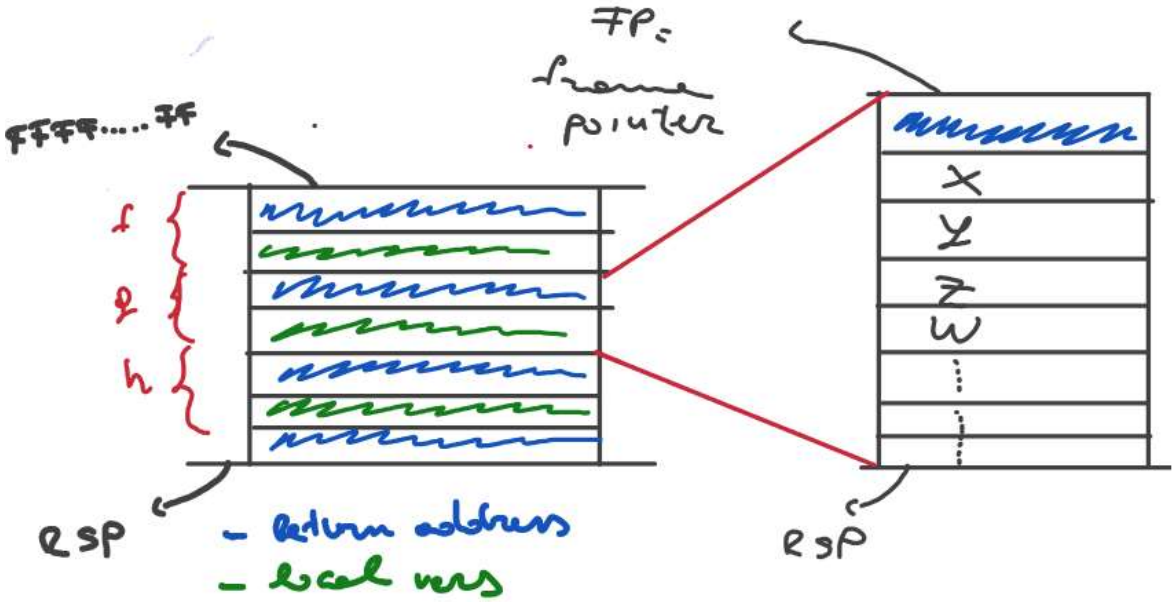
a.

```
.begin:
    # <codice>
    cmpb ...
    jz .begin
```

```
movq $0, %rcx
movq $3, %rbx
.test: cmpq %rbx, %rcx
    jz .end
    # <codice>
    addq $1, %rcx
    jmp .test
.end:
```

4. **Salti e chiamate a funzioni** : da notare che il goto è equivalente di jmp ; se in presenza di goto o jmp si ha una chiamata di istruzione CALL.

**Concentriamoci ora sul C:** ogni variabile in questo linguaggio ha associato un tipo che la caratterizza , e questi tipi sono raggruppati in 3 categorie : **primitivi, aggregati e puntatori** . Solo i primi due hanno una corrispondenza in assembler ed in più i secondi vengono convertiti in accesso ai tipi primitivi . Ognuna delle variabili ha due contesti (scope) di ambito e di classe : le prime (**anche dette globali**), risiedono all'interno delle sezioni .data e .bss , mentre le locali (**anche dette automatiche**), occupano memoria all'interno dello stack , mentre le ultime(**statiche**) sono delle locali, ma ristrette all'interno della funzione/modulo . Per una qualunque funzione (almeno in C), si ha un **record di attivazione** : zona di memoria che contiene i parametri formali e le variabili locali della funzione stessa. Non ha una dimensione fissa, in quanto dipende dal tipo della funzione (parametri). Quando la funzione termina, il record di attivazione viene cancellato dallo stack , **ovvero invalidato logicamente,** liberando cosi la memoria per successivi record di attivazione. Vediamo logicamente come è fatto :




Notiamo che RSP (Register stack pointer) punta alla fine della memoria, quindi all'inizio dello stack. Mentre se si parla di FP(frame pointer) indica la base del record di attivazione (FFFFF...FFFF). Se si ha una sola variabile accedere all'indirizzo è facile , uso l'indirizzo della prima cella di RSP , mentre se devo prendere l'n-esima cella, devo spiazzarmi dentro RSP. Vediamo esempio : ho x e y variabili , andiamo a caricare l'indirizzo:

| X | Y |
|---|---|
| movl (%rsp),%eax | Movl 4(%rsp),%eax |

Iterando il procedimento funzionerebbe, ma si potrebbe perdere il conto della cella , quindi optiamo per

una soluzione simili : stesso principio base , ma salviamo la base dello stack pointer in un registro ( rsp-> rbp) : Register base pointer , il quale viene decrementtato in base al numero di celle che servono :

```
void function() {
    int x = 128;
    ...
    return;
}
```

```
function:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $4, %rsp
        movl    $128, -4(%rbp)
        ...
        addq    $8, %rsp
        leave
        ret
```

**Le prime 3 istruzioni ( pushq, movq,subq) sono intese come preambolo , mentre le ultime 3 sono considerate come tail ( rispristino stack e ritorno ad indirizzo del chiamante +1 ) .**

Riassumendo quindi :