1 Istruzioni per il controllo del flusso

Ogni programma esegue automaticamente in successione le istruzioni i cui codici sono posti in memoria in locazioni successive. Ci si può immaginare che un programma che esegua solo istruzioni rigorosamente in fila non possa essere di grande utilità!
Infatti un programma deve essere in grado di eseguire parti di codice diverse in dipendenza dai risultati di confronti fra numeri; inoltre deve poter eseguire le stesse istruzioni molte volte.
In questo capitolo spieghiamo le istruzioni Assembly che ci permettono di controllare il flusso del nostro programma e come realizziamo in Assembly le strutture di controllo del flusso più usate nei linguaggi di alto livello.

1.1 Istruzioni per il controllo del flusso del programma

L'ordine con cui le istruzioni di un programma vengono eseguite viene detto "**flusso**" di quel programma.
Ogni programma deve poter modificare il flusso delle sue istruzioni in conseguenza degli ingressi che riceve e del risultato dei calcoli che esegue. Nei linguaggi macchina delle CPU devono perciò essere presenti istruzioni che permettano di cambiare l'ordine con cui il programma viene eseguito.
Queste istruzioni vengono dette **salti**, in Inglese "**jump**" o anche "**branch**". I salti possono essere incondizionati o condizionati.

1.1.1 Salti incondizionati

Un salto rompe la continuità del flusso delle istruzioni e fa in modo che la prossima istruzione eseguita non sia quella che segue nella memoria, ma un'altra. Il flusso del programma salta quindi ad un'istruzione diversa da quella "naturale".
Un salto è **incondizionato** quando viene eseguito sempre e comunque. Nell'8086 l'istruzione di salto incondizionato è la jump, che si scrive con la seguente sintassi:

```
JMP <label>
```

La sigla JMP proviene dalla parola **ju**mp; l'istruzione "salta" all'indirizzo della label indicata. Quando la CPU incontra il codice operativo di una JMP fa in modo che la prossima istruzione sia presa in fase di fetch all'indirizzo che corrisponde all'etichetta indicata.
La fase di execute di una jump incondizionata è quindi molto semplice: il program counter viene sovrascritto con l'indirizzo della label.

Vediamo un esempio in cui è indicato, con un numero nel commento, l'ordine di esecuzione delle istruzioni:

```
    ..
    ADD AX, BX       ; 1
    JMP SaltaQuella  ; 2
    MOV AX, 1        ; 0 !! istruzione che non viene mai eseguita !!
SaltaQuella:
    ADD AX, 1        ; 3
    ..
    JMP SaltaQuella  ; ? salto alla stessa etichetta della istruzione 2
    ..
```

Si noti che nel codice precedente la JMP "isola" l'istruzione numerata con 0, che non viene mai eseguita.
L'istruzione indicata con un ? viene eseguita in un tempo successivo, dopo altre istruzioni non specificate, indicate dal segno "..". E' stata aggiunta a questo esempio per indicare che in un programma possono esistere diverse jump che saltano alla stessa etichetta.

Nell'8086 al codice mnemonico JMP corrispondono più di una istruzione di macchina.
Infatti la JMP può avere un indirizzo assoluto o relativo. Se l'indirizzo è relativo l'istruzione è più compatta e prende meno memoria ma l'etichetta cui si salta deve essere "nei dintorni" della jump, 127 byte prima o dopo di essa.
Se l'indirizzo di arrivo della JMP è nei dintorni del punto da cui parte, essa viene detta jump "short" (corta).
L'Assembler decide automaticamente se usare il codice operativo per una jump relativa (short) o assoluta (jump "long").
Le jump short hanno come operando la variazione che deve subire IP alla fine dell'istruzione, rappresentata come numero in complemento a due di 8 bit.
Analoghe sono le jump long. La differenza è che la variazione di IP è espressa come numero di 16 bit.
L'esempio che segue dovrebbe illuminare:

```
Sorgente                                                    Indirizzo       L.m.
JMP QuiVicino ; jump short                                  0000            EB 01

QuiVicino:                                                  0002            E9 B403
JMP LaLontano  ; jump long (dopo
; allocazione di 1 kByte solo perché l'etichetta LaLontano
sia "lontana":
                                                                             .. 1024
SoloPerOccupareMemoria DB 1204 DUP (?)                      Da 0003 a 04B8  byte di dati

LaLontano:                                                  03B9             .. continua
```

Come si può notare nel brano di programma precedente lo stesso mnemonico JMP è stato tradotto con due codici operativi diversi: EB ed E9; l'operando della jump short è di un byte (01), mentre quello della jump long è di due byte (B404, che corrisponde al numero 04B4h).
Gli Assembler più comuni generano automaticamente una JMP short quando la label è distante meno di 127 byte dall'istruzione, all'avanti o all'indietro, mentre se ciò non è possibile usano l'indirizzo completo.
Dato che il "program counter" di un 8086 è costituito dai due registri CS e IP, le jump long possono essere di due tipi: jump NEAR, che modificano solo IP e jump FAR, che modificano sia CS che IP. Per il dettaglio sulla differenza fra jump NEAR e FAR si rimanda al capitolo relativo alla segmentazione.

Jump indirette
E' possibile fare jump incondizionate a indirizzi che si trovano in memoria. Prendiamo per un esempio il caso in cui si voglia effettuare il reset "software" del computer. Per far questo si può saltare all'indirizzo segmentato 0FFFF:0000, che è l'indirizzo a cui viene fatta la prima fase di fetch all'accensione, cioè l'inizio del programma con cui la CPU inizia a lavorare:

```
MOV Word PTR [IndirizzoInMemoria], 0      ; nella word bassa l'offset
MOV Word PTR [IndirizzoInMemoria + 2], 0FFFh  ; nella word alta il segmento
JMP [IndirizzoInMemoria] ; salta all'istruzione che si trova a FFFF:0000
```

1.1.2 Salti condizionati

Oltre a saltare, un programma deve anche essere in grado di fare confronti fra numeri e di eseguire parti di codice diverse in base ai risultati dei confronti.
Perché un programma "prenda decisioni" si devono introdurre delle jump che saltano a punti diversi in base al risultato delle operazioni precedenti, ossia le jump "**condizionate**.

Le jump condizionate saltano all'indirizzo indicato nell'istruzione solo se è vera una certa condizione, altrimenti il programma prosegue come se l'istruzione di salto non esistesse.

Nell'Assembly 8086 le jump condizionate hanno la seguente sintassi:

```
J[<negazione>]<condizione> <etichetta>
```

Tutte le istruzioni di salto iniziano con la lettera J (**J**ump = salto), poi seguono, senza spazi in mezzo, fino a tre lettere che specificano la condizione che fa saltare l'istruzione.

La sintassi della jump comprende una condizione che può essere vera o falsa. L'istruzione può fare due cose: se la condizione è vera esegue un salto all'etichetta indicata, se la condizione è falsa prosegue.
Con questa istruzione si può dividere il flusso di un programma in due percorsi; il programma può prendere strade diverse in base al risultato di istruzioni precedenti.
Se necessario si può anteporre alla <condizione> una <negazione>, costituita dalla lettera N (**N**ot), che inverte il significato di ciò che segue.
Ciò significa che JN<condizione> <etichetta> funziona in modo opposto a J<condizione> <etichetta> (Esempio: JC Errore salta alla label Errore se il flag di carry è uguale a 1, mentre JNC TuttoBene salta a TuttoBene se CF = 0)

La condizione può essere di due tipi; può essere relativa ad un flag o ad una condizione più complessa come "maggiore" o "minore":

```
<condizione> := <CondizioneSuFlag> | <CondizioneOrdine>
```

<CondizioneSuFlag> è una lettera che corrisponde ad uno dei flag della CPU:

```
<CondizioneSuFlag> := O | S | A | P | C | T | I | D
```

O = Overflow, S = segno, A = carry ausiliario, P = parità, C = carry, T = trap, I = interrupt, D = direction
(il significato di alcuni di questi flag verrà spiegato nel seguito).

<CondizioneOrdine> è relativa ad una relazione d'ordine (maggioranza, minoranza ..) e può essere una delle seguenti:

```
<CondizioneOrdine> := A | B | G | L | E [<uguaglianza>]
```

A = **A**bove ("sopra", maggiore), B = **B**elow ("sotto", minore), G = **G**reater ("più grande", maggiore), L = **L**ess ("meno", minore), E = **E**qual (uguale)

Dopo la lettera che esprima la <CondizioneOrdine> può essere aggiunto <uguaglianza>,cioè la lettera E (**E**qual, uguale), che aggiunge alla condizione anche l'uguaglianza ed è opzionale.

Alcuni esempi di sintassi di jump condizionate:

```
    JC  SaltaLi  ; Jump if Carry, salta alla label SaltaLi se CF = ON
    JE  SaltaLi  ; Jump if Equal, salta se precedentemente
                 ; ho confrontato due numeri uguali
    JNO          ; Jump if Not Overflow: salta se OF = 0
    JAE SaltaLi  ; Jump if Above or Equal
    JNGE SaltaLi ; Jump if Not Greater or Equal
    JAN SaltaLi  ; errato: N può essere solo subito dopo la J
    JOE SaltaLi  ; errato: la condizione sui flag non ammette l'uguaglianza (E)
```

Salti condizionati sui flag

In molte occasioni è necessario verificare la condizione di uno qualsiasi del flag della CPU.
Ciò può accadere per esempio dopo che si sono eseguite operazioni aritmetiche. Effettuando un salto sul flag di carry o su overflow si potrà verificare se il risultato dell'operazione è corretto e saltare ad una parte del programma che gestisce l'errore, usando il flag di carry nel caso di operazione fra numeri senza segno od overflow nel caso di numeri con segno.
Di seguito presentiamo un esempio:

```
    ..
    ; in AL un numero senza segno:
    MOV AL, 0FFh ; il numero più grande di 8 bit          1
    ADD AL,  1   ; la somma non ci sta in AL =>           2
                 ; in AL c'è 0 e il flag di carry C = 1
    JC  Tracimazione ; salta se c'è il carry              3
    ; qui il carry flag (CF) = 1, ZF = 1 (zero flag)
    ..
SaltaSu:     ;                                                6
    ..
Tracimazione:
    ADD AL, 1    ; 0 + 1 = 1 in AL, senza Carry (CF = 0)   4
    JNC SaltaSu  ;                                         5
```

Se si vuole sapere se il risultato dell'operazione precedente è zero si può usare una jump sul flag di zero (JZ).
Se si vuole verificare il segno di un numero in complemento a due (con segno) si verifica il flag S (JS), che può anche servire per sapere se il bit più significativo del risultato è uguale a uno.

Alcune istruzioni importanti non modificano i flag, per cui essi rimangono uguali a quelli sistemati da una operazione precedente. Questa caratteristica può essere sfruttata per fare alcuni "trucchi" di programmazione che possono migliorare l'efficienza dei programmi Assembly.
Le più importanti istruzioni che non modificano i flag sono le MOV e tutte le jump. Nel seguente esempio vengono fatte due jump di seguito, dopo una singola ADD, che modifica i flag per entrambe:

```
    ..
    ; nota: il programma considera che si usino numeri con segno
    ADD AX, BX   ; una somma
    JZ RisZero       ; salta se il risultato è zero
    ; JZ non modifica i flag, per cui in questo punto i flag sono quelli prodotti
    ; dalla ADD AX, BX
    JNS RisPos   ; salta se il risultato non è negativo
    ; il flusso del programma arriva qui se il risultato è strettamente positivo
    ..
```

Salti condizionati su condizioni aritmetiche

Nella programmazione è molto frequente la necessità di confrontare due numeri e di stabilire quale dei due è il maggiore. A questo scopo sono presenti nelle CPU alcune istruzioni specifiche che, controllando uno o più flag contemporaneamente, sono in grado di porre in una relazione d'ordine operandi numerici.
Per realizzare l'equivalente a basso livello dell'istruzione "if "di C o Pascal è necessario utilizzare due istruzioni Assembly, una che mette a confronto i due numeri, modificando i flag della CPU ed una che, utilizzando i valori dei flag appena calcolati, effettua un salto condizionato.
Nell'8086 l'istruzione che mette a confronto due numeri è la "compare".

*Confronto fra numeri interi: CMP (**Comp**are)*

L'istruzione di confronto è una sottrazione "nascosta", che avviene fra due operandi. La sua sintassi nell'8086 è la seguente:

```
CMP <operando1> , <operando2>
```

<operando1> ed <operando2> sono sottoposti ai consueti vincoli per gli operandi delle istruzioni 8086 (possono essere registri, numeri in immediato o locazioni di memoria, ma uno solo degli operandi può essere una locazione di memoria, gli indirizzamenti sono quelli usuali ..).
CMP non modifica il valore di nessuno dei suoi operandi.
Esempi:

```
CMP DH, 28   ; confronto in immediato fra il valore corrente di DH ed il numero 28
CMP [13], AX ; confronto con un operando in memoria, indirizzamento diretto
CMP SI, [DI + BX]    ; confronto fra SI e la locazione di memoria di indirizzo
                     ; DI + BX
```

L'istruzione viene eseguita dalla CPU come se fosse una sottrazione: <operando1> - <operando2>. La differenza con un'istruzione di sottrazione (SUB) è che CMP influenza solo i flag e non lascia alcuna traccia sugli operandi (registri o memoria). Il risultato verrà calcolato dall'ALU ed i flag saranno calcolati come se si fosse eseguita una SUB, ma il risultato algebrico non verrà memorizzato da nessuna parte.

Jump aritmetiche

Dopo una CMP ci dovrebbe sempre essere una jump aritmetica, o almeno un'istruzione che non modifica i flag, seguita da una jump condizionata. Infatti non avrebbe alcun senso fare un confronto fra due numeri se il risultato di quel confronto non fosse subito sfruttato per saltare a parti diverse del programma.
Ricordando che la CMP fa una sottrazione fra i suoi operandi, vediamo come i flag possono indicare la relazione d'ordine fra i due numeri.

Uguaglianza

Consideriamo di aver appena eseguito una CMP <operando1> , <operando2>, che, ricordiamo, funziona come una sottrazione.
Se <operando1> = <operando2> allora <operando1> **-** <operando2> = 0.
Se gli operandi della CMP sono uguali la sottrazione dà risultato zero, per cui il flag di zero è ON dopo la compare.
Dunque l'istruzione JE (Jump if Equal) deve essere uguale a JZ. JE e JZ sono per la CPU la stessa istruzione di macchina ed il compilatore traduce entrambe nello stesso codice operativo (76h nell'8086).
Il programmatore potrà usare indifferentemente JE o JZ, ma per far capire meglio come funziona il suo programma userà JZ quando sta controllando che il risultato sia zero, JE quando sta verificando l'uguaglianza.
Esempio:

```
ADD DH, DL   ; una somma
JZ SommaZero ; salta se la somma dà zero
;^ si poteva usare anche JE
..
CMP AL, CL   ; un confronto
JE Uguali    ; salta se AL è uguale a CL
;^ si poteva usare anche JZ
```

Jump d'ordine per numeri senza segno

Supponiamo ora di trattare numeri senza segno, cioè numeri binari "ordinari".
Per controllare se un numero senza segno è maggiore o minore di un altro esistono le due istruzioni JA (Jump if Above ("sopra", maggiore)) e JB (Jump if Below ("sotto", minore)).
Esaminiamo come devono essere i flag dopo una CMP se <operando1> è minore di <operando2>:
Se <operando1> **<** <operando2> **=>** <operando1> **-** <operando2> < 0
Se i due operandi sono numeri senza segno ciò significa che la sottrazione deve dare luogo ad un prestito, quindi il flag di carry deve essere ON.
Quindi, come JE è uguale a JZ, così JB è uguale a JC.

"Dimostriamo" quest'ultima affermazione a compilando questo brano di programma, senza informazioni simboliche per il debugger:

```
    ..
    JZ SoloPerProva
    JE SoloPerProva
    JB SoloPerProva
    JC SoloPerProva
SoloPerProva:
    NOP
    ..
```

Il risultato, visto con la funzione disassembler del debugger è:

```
linguaggio macchina        codice disassemblato
    cs:0000 7406           je     0008
    cs:0002 7404           je     0008
    cs:0004 7202           jb     0008
    cs:0006 7200           jb     0008
    cs:0008 90             nop
```

Si può vedere come JZ e JE del programma originale siano state tradotte nello stesso codice operativo (74h), per cui il disassembler, che non ha informazioni simboliche, deve scegliere uno dei due codici mnemonici che corrispondono all'istruzione ed usare sempre quello. La stessa cosa accade per le istruzioni JB e JC, che hanno entrambe il codice operativo 72h.

Jump per numeri con segno

Per i confronti aritmetici fra numeri con segno, rappresentati in complemento a due, esistono due istruzioni analoghe a JA e JB. Esse sono JG (Greater ("più grande", maggiore)) e JL (Less ("meno", minore)).

Considerazioni analoghe a quelle fatte precedentemente portano a considerare, per le istruzioni JG, JL per le loro "variazioni", il flag di segno in luogo di quello di carry.

I flag controllati sono indicati nella tabella, nella quale si può notare che le istruzioni JS e JL sono in realtà identiche, come JE e JZ e JB e JC.

Nella tabella sono indicate tutte le forme di jump aritmetiche dell'Assembly 8086, anche quelle che fanno uso della negazione (N) e/o dell'uguaglianza (E). Come si può notare molte di queste forme sono equivalenti e corrispondono alla stessa istruzione di macchina.

Mnemonico	Condizione per il salto	Salta se i flag:
JZ = JE	Op1 = Op2	Z = 1
JNZ = JNE	Op1 != Op2	Z = 0
JB = JC = JNAE	Op1 < Op2 senza segno	C = 1
JAE = JNB = JNC	Op1 >= Op2 senza segno	C = 0
JA = JNBE	Op1 > Op2 senza segno	C = 0 and Z = 0
JBE = JNA	Op1 <= Op2 senza segno	C = 1 or Z = 1
JL = JS = JNGE	Op1 < Op2 con segno	S = 1
JGE = JNL = JNS	Op1 >= Op2 con segno	S = 0
JG = JNLE	Op1 > Op2 con segno	S = 0 and Z = 0
JLE = JNG	Op1 <= Op2 con segno	S = 1 or Z = 1

Tabella 1: Jump aritmetiche e condizioni dei flag che fanno fare il salto

Nelle CPU X86 precedenti al 386 le jump condizionate hanno un indirizzo relativo, sono cioè "short" e possono saltare avanti o indietro di 127 byte. Se si deve saltare più lontano è necessario inserire una jump incondizionata che può saltare ovunque. Segue un esempio in tal senso:

```
; se questa non funziona, perché PiuLontanoDi127byte è troppo lontano:
    JC PiuLontanoDi127byte

; Bisogna cambiare il programma così:
    JNC ContinuaQuiSotto
    JMP PiuLontanoDi127byte ; la JMP può andare lontano quanto voglio!
ContinuaQuiSotto:
```

Nelle CPU dal 386 in poi esistono jump condizionate "near" che permettono di saltare in qualsiasi punto del segmento di codice corrente.

Riandando al codice in linguaggio macchina presentato precedentemente si vede come tutte le istruzioni saltano alla stessa label SoloPerProva. L'etichetta ha indirizzo 0008 e le istruzioni, tutte short, per raggiungerla usano come operando numeri diversi: la prima JE usa 06 (come si vede dai numeri 7406 in linguaggio macchina), la seconda JE che è più vicina a 0008, usa 04, e così via, l'ultima JB, che ha già IP posto sull'istruzione NOP, non deve aggiungergli nulla (operando 00).

Oltre alle jump condizionate esistono altre istruzioni che fanno sempre salti relativi "short", analoghi a quelli delle jump condizionate, esse sono: tutte le LOOP, JCXZ, JECXZ, che vedremo fra breve.

Non sbagliare nell'uso delle jump condizionate è molto facile. Basta "pensarle in Inglese", poi scrivere le lettere corrispondenti ..

	Numeri senza segno	Numeri con segno
Salta se maggiore	JA (Above)	JG (Greater)
Salta se minore	JB (Below)	JL (Less)

Figura 1: Schema riassuntivo delle jump con condizioni d'ordine

JCXZ, JNCXZ

Esistono due jump particolari, che controllano se il valore del registro "contatore" dell'8086 è correntemente a zero.
La loro sintassi è la seguente:

```
JCXZ <label>        ; Jump on CX = Zero
JCNXZ <label>       ; Jump on CX Not = Zero
```

JCXZ fa il salto se il valore corrente del registro CX è zero, mentre JNCXZ, in coerenza con la sintassi delle altre jump, fa il contrario.

1.1.3 Realizzazione delle strutture di controllo dei linguaggi di alto livello

Istruzione if senza else

L'istruzione "if ", od una equivalente, è presente in tutti i linguaggi di programmazione ed è di fondamentale importanza.
La "if" dei linguaggi di alto livello è seguita da un blocco di istruzioni che vengono eseguite quando una condizione è vera.
In Assembly la situazione è un po' diversa; come abbiamo visto per realizzare una diramazione del flusso del programma si deve esprimere una condizione che, se vera, implicherà un salto nel flusso del programma. Il salto eluderà alcune istruzioni, che perciò vengono eseguite solo se la condizione è falsa. La logica con cui si pensa alle "if" in Assembly è dunque invertita rispetto agli altri linguaggi. Se in C si pensa a cosa si deve fare quando la condizione è vera, in Assembly si pensa a cosa non si deve fare.
Ciò si potrà vedere analizzando con attenzione l'esempio seguente:

```
// una if qualsiasi, in C++
if (a >= 10)
    a = 9;
..
```

L'istruzione assegna il valore 9 alla variabile a solo se a vale 10 o più di 10.
Traduciamo in Assembly in due modi, usando la stessa logica del C ed una logica "negata":

```
    ; traduzione con la logica del C:
    CMP [a], 10      ; confronto fra a e 10
    JAE Assegna9     ; salta se a >= 10
    JMP NonAssegnare  ; !!
    ; !! ^ devo mettere questa JMP per evitare che si faccia comunque
    ; !! l'assegnazione seguente
    ; giunge qui solo se a >= 10
    MOV [a], 9        ; assegnazione a = 9
NonAssegnare:
    ..

    ; traduzione con logica negata:
    CMP [a], 10
    JNAE NonAssegnare    ; la logica della if del C è invertita in Assembly
                         ; (è equivalente ad una JB, cioè salta se è minore)
Assegna9:                ; questa label è "inutile", ma non costa niente
                         ; e rende il programma più leggibile
    MOV [a], 9
NonAssegnare:
    ..
```

La versione con logica negata è migliore dell'altra. Infatti si risparmia un'istruzione perché non c'è la JMP NonAssegnare della prima versione.
Ma non solo, il vantaggio forse più importante è che la seconda soluzione è molto più "pulita" e lineare, più facile da comprendere per chi la legge. Il fatto che nella prima soluzione ci sia un salto in più rende il programma più contorto e difficile da interpretare.

Istruzione if con else

La presenza di una else nella if individua due blocchi di istruzioni, che devono essere eseguiti in alternativa, in base al risultato della condizione.

Vediamo come si può tradurre il seguente brano di programma C:

```
    if (a <= 0)
        a = 1
    else
        a = 0;
```

si traduce così:

```
    CMP [a], 0
    JBE CasoIf   ; in questo caso la logica della if è "dritta"
CasoElse:        ; anche questa label è "inutile", come nell'esempio precedente
    MOV [a], 0
    JMP Continua ; un "tappo" per evitare di finire nella parte di "CasoIf"
CasoIf:
    MOV [a], 1
Continua: ..
```

Nel caso della if con else non è conveniente "invertire la logica" del salto, perché ci sono due casi alternativi e ci sarebbe comunque un'istruzione analoga alla JMP Continua.
In casi come questi bisogna fare attenzione a non dimenticare il salto incondizionato in fondo al primo dei due blocchi di istruzioni. Senza la "JMP Continua" a "fare da tappo" il flusso del programma passa sia dal caso if che dal caso else.

Due if con una sola compare

```
    if (c = 10) c = c - 1;
    if (c > 10) c = 0;
```

Queste due if, che confrontano entrambe con lo stesso numero, si possono realizzare in Assembly con una sola CMP:

```
    CMP [c], 10
    JE Cuguale10     ; due jump di seguito! La prima delle due non modifica
                     ; nessun flag, per cui la seconda lavora sui flag della CMP
ControllaAncheMaggiore:
    JNG Continua
Cmaggiore10: ; label inutile, è qui solo per documentazione
    MOV [c], 0
    JMP Continua
Cuguale10:
    DEC [c]
    ; DEC ha modificato i flag, per cui bisogna rifare la CMP, prima
    ; di tornare alla JNG:
    CMP [c], 10
    JMP ControllaAncheMaggiore
    ; ^ si torna alla seconda istruzione C
Continua:
    ..
```

If con condizioni multiple

In questo paragrafo illustriamo alcuni esempi con condizioni leggermente più complicate.

```
    // l'istruzione in C:
    if (a >= 1 && a < 7) a = 0;

    ; può essere tradotta così:
    CMP [a], 1
    JNAE NonCancellare
    CMP [a], 7
    JNB NonCancellare
    MOV [a], 0   ; a = 0
NonCancellare: ..
    // l'istruzione in C:
    if (a = 3 || a >= 6) a = 9;

    ; tradotta:
    CMP [a], 3
    JE Scrivi    ; solo qui la logica non è invertita!
    CMP [a], 6
    JNAE NonScrivere ; qui la logica è invertita!
Scrivi:
    MOV [a], 9
```

```
NonScrivere: ..
```

Cicli

In questo paragrafo introduciamo uno dei mattoni fondamentali con cui si possono costruire tutti i programmi.
In ogni programma non banale capita spesse volte di dover ripetere le stesse operazioni per un certo numero di volte.
Un **ciclo** ("loop") è una struttura di controllo di un programma che esegue per molte volte lo stesso insieme di istruzioni.
Per indicare che il ciclo viene percorso molte volte si dice che il ciclo è una struttura "**iterativa**".
In un ciclo si possono distinguere quattro fasi: inizializzazione, corpo, aggiornamento e controllo (o test).
Il **corpo** è l'insieme delle istruzioni che devono essere eseguite per molte volte ("iterativamente").
L'**inizializzazione** è la fase di preparazione del ciclo, nella quale si fa in modo che il corpo sia eseguito correttamente per la prima volta.
Durante la fase di **aggiornamento** il programma si prepara ad eseguire il corpo per un'altra volta.
La fase di **controllo**, o "**test**", serve per valutare se si deve eseguire un'altra iterazione del ciclo o se ne deve uscire.
Come indicato anche dalla Figura 2 in un ciclo il controllo può essere fatto prima di entrarvi per la prima volta (controllo "in testa") o dopo aver eseguito il corpo almeno una volta (controllo "in coda").
Per ragioni di integrità e maggiore "sicurezza" del programma è di norma preferibile realizzare cicli con controllo in testa. Peraltro programmando in Assembly questa indicazione viene spesso disattesa, perché il programma risulta di lettura più semplice se il controllo è in coda ed anche perché l'istruzione specifica per realizzare i cicli (loop) usa un controllo in coda.

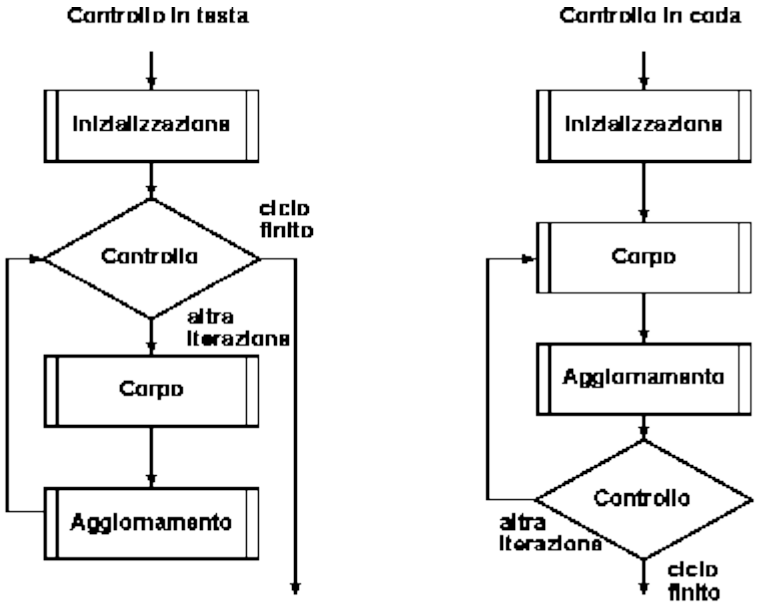

Figura 2: Cicli con controllo in testa o in coda

Nei diagrammi di flusso della figura si coglie a prima vista come il programma, "giri" dentro al ciclo fino a che la condizione di controllo non lo fa uscire. Un "giro" di un ciclo viene detto "**iterazione**".

Cicli con un numero noto di iterazioni

Passiamo ora alla realizzazione dei cicli più comuni nella programmazione, quelli nei quali si sa a priori, prima di entrarvi, quante volte bisognerà girare.
In questo tipo di cicli nell'inizializzazione si deve impostare un contatore, che terrà traccia in ogni istante del numero di giri sono stati compiuti. Chiameremo questo contatore "contatore di controllo".
La fase di aggiornamento modifica il contatore per segnalare che è stata eseguita un'altra iterazione.
Il test stabilisce se il ciclo è stato percorso il giusto numero di volte e se è giunto il momento di uscirne.

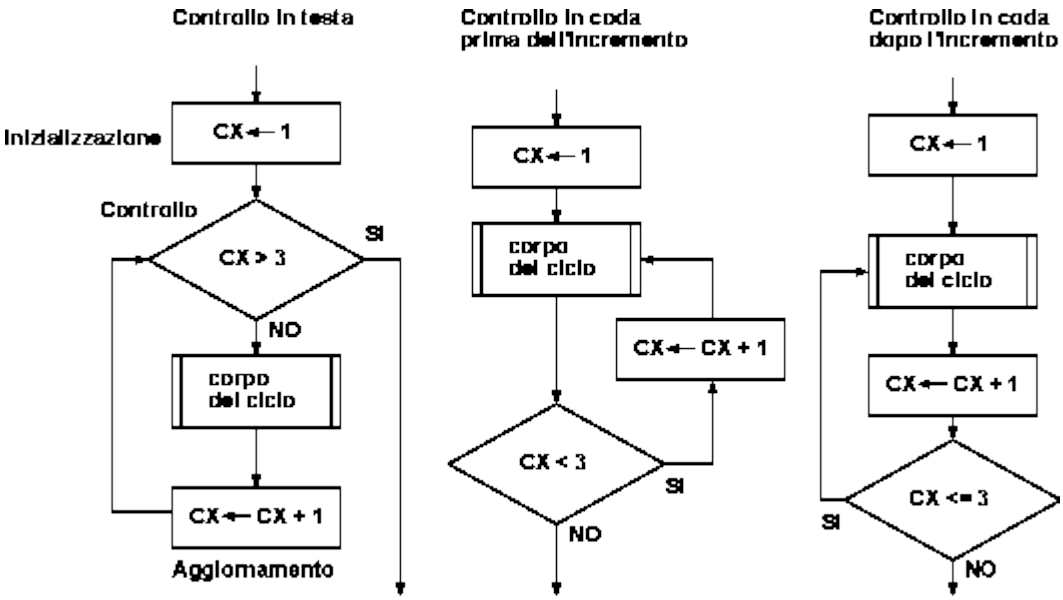

Figura 3: tre modi per fare un ciclo con 3 iterazioni

Vediamo ora la realizzazione in Assembly dei tre cicli illustrati dai diagrammi di flusso.

Versione con controllo in testa:

```
    MOV CX, 1    ; inizializzazione del contatore di controllo del ciclo
                 ; uso CX perché è il "contatore" dell'8086
                 ; ed anche per un'altra ragione, che vedremo presto
AltroGiro:
    CMP CX, 3    ; controllo in testa
    JA Fuori

    .. corpo del ciclo ..

    INC CX
    JMP AltroGiro
Fuori: ; da qui il programma prosegue
    ..
```

Versione con test in fondo, prima dell'incremento:

```
    MOV CX, 1    ; inizializzazione del contatore di controllo
InizioCorpo:

    .. corpo del ciclo ..

    CMP CX, 3    ; test in coda, prima dell'incremento
    JE Fuori
    INC CX       ; incremento dopo il test
    JMP InizioCorpo
Fuori: ; da qui il programma prosegue
    ..
```

Versione con test in fondo, dopo l'incremento:

```
    MOV CX, 1    ; inizializzazione del contatore
InizioCorpo:

    .. corpo del ciclo ..

    INC CX       ; incremento prima del test
    CMP CX, 3    ; test in coda, dopo l'incremento
    JBE InizioCorpo  ; JBE perché al primo giro qui CX è già a 2
```

```
    ; da qui il programma prosegue, ma non c'è bisogno di JMP Fuori !
    ..
```

Si noti che se si vuole fare lo stesso numero di iterazioni la versione con ciclo in testa e quella con ciclo in coda devono fare controlli diversi (JA contro JE).
La terza versione, con controllo in coda dopo l'incremento, dà luogo ad un programma Assembly migliore, nel quale manca la JMP incondizionata che c'è negli altri due. Inoltre la terza versione è molto più lineare e semplice da capire, dote che non si mancherà mai di magnificare in un programma Assembly.
La versione con il test all'inizio permette di non eseguire mai il corpo del ciclo, qualora ve ne sia bisogno.

Il modo più efficiente di fare i cicli in Assembly non è uno di quelli appena illustrati, ma un modo un po' originale nel quale la variabile di controllo procede "a rovescio".
Vediamo dunque il diagramma di flusso per il miglior ciclo Assembly:

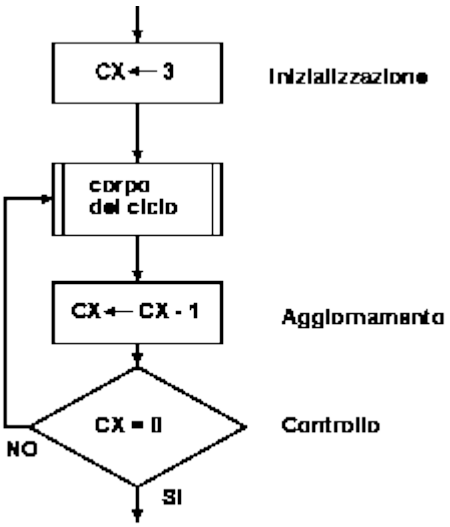

Figura 4: ciclo con decremento
Il ciclo ha controllo in coda dopo l'incremento, come accade nel terzo caso della figura precedente, ma la sua realizzazione in Assembly presenta un vantaggio:

```
    MOV CX, 3    ; inizializzazione "DALLA FINE" del contatore del ciclo
InizioCorpo:

    .. corpo del ciclo ..

    DEC CX       ; DECREMENTO: CX deve andare da 3 a zero
    ; dopo la DEC se il risultato è zero il flag zero (ZF) è GIA' on,
    ; senza bisogno di nessuna CMP
    JNZ InizioCorpo
    ; anche qui ma non c'è bisogno di JMP Fuori, per proseguire !
    ..
```

Come si vede nel ciclo appena scritto manca la CMP finale che non è necessaria perché il ciclo, andando a rovescio, finisce sempre con zero e la condizione di zero viene rilevata direttamente dalla DEC (ZF = 1).
Con quest'ultima versione "risparmiamo" una CMP per ogni iterazione del ciclo.
Il risparmio di un'istruzione può sembrare un'inezia, ma diventa significativo se fatto per milioni di volte al secondo, come accade per esempio nei programmi che fanno parte del sistema operativo, che sono molto usati.

E' importante far notare quanto la posizione dell'etichetta che torna all'inizio del ciclo sia decisiva.
Guardiamo cosa succede nell'ultimo programma presentato se l'etichetta InizioCorpo viene messa un'istruzione più in alto. Ogni volta che si esegue la JNZ finale e si torna su a InizioCorpo, CX viene rimesso a 3 ed il ciclo non esce più (CX vale sempre 2 al momento della JNZ).
Anche se l'etichetta viene messa più in giù si causano danni, perché una o più istruzioni del corpo non vengono eseguite.

Sbagliare la posizione dell'etichetta che fa ripetere un ciclo è un errore piuttosto comune, per cui bisogna prestare doppia attenzione quando la si mette.

Cicli "automatici": LOOP
Per migliorare ulteriormente la velocità[1] di elaborazione e la compattezza del codice dei programmi realizzati, il set d'istruzioni dell'8086 contiene un'istruzione apposita, che sintetizza le istruzioni che concludono un ciclo con controllo in coda e variabile decrementata. Si tratta dell'istruzione LOOP, che ha la seguente sintassi:

```
LOOP <Label>
```

Esempio:

```
LOOP Gira256volte
```

Naturalmente nel programma ci dovrà essere una etichetta "Gira256volte:", dove l'istruzione LOOP deve saltare fino a che CX non diviene zero.
A prima vista questa istruzione lascia interdetti, perché si presenta come una jump incondizionata; la sintassi dell'istruzione non dà alcuna indicazione su come l'istruzione si comporta.
Dunque per usare una LOOP quindi bisogna sapere (e ricordare) come funziona internamente.
LOOP esegue due istruzioni "insieme" nel modo illustrato dalla figura:

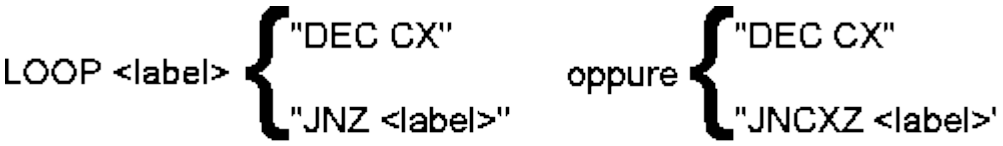

Figura 5: funzionamento di una LOOP

Scrivendo le istruzioni fra virgolette si vuole significare che queste sono istruzioni "interne" alla CPU, il programmatore non dovrà scriverle, perché la scrittura di LOOP <Label> equivale a scrivere entrambe le istruzioni fra virgolette.
Dunque la LOOP funziona esattamente come le due istruzioni che concludevano il ciclo fatto "a mano" precedentemente.
E' importante notare che l'istruzione LOOP usa esclusivamente il registro CX, che perciò assume qui quel significato di contatore che gli avevamo attribuito precedentemente. Anche se non è evidente dalla scrittura dell'istruzione il registro CX viene sempre modificato dalla LOOP. Ogni volta che usiamo una LOOP dobbiamo chiederci cosa c'è attualmente in CX e se quello che c'è è il valore giusto per il contatore del ciclo.
Se in CX non c'è il valore giusto o se CX viene incautamente modificato nel corpo del ciclo la LOOP non funziona!

Per usare la LOOP si deve:
1 – Inizializzare CX con il numero di iterazioni che si devono effettuare
2 - Scrivere il corpo del ciclo, che non deve modificare CX, neanche indirettamente
3 – Scrivere l'istruzione LOOP con <Label> che punta all'inizio del corpo del ciclo

Vediamo dunque come realizzare un ciclo da tre iterazioni, come quello illustrato nel paragrafo precedente.

```
    MOV CX, 3    ; inizializzazione "DALLA FINE" del contatore del ciclo
InizioCorpo:

    .. corpo del ciclo ..

    LOOP InizioCorpo
```

Come si vede il codice diventa più compatto e, una volta che ci si ricorda come funziona la LOOP, anche più leggibile.

Naturalmente LOOP non ci è utile per tutti quei casi in cui il ciclo debba procedere per passi diversi da 1.
Per quanto sia possibile farlo è sconsigliabile modificare il valore di CX all'interno del corpo del ciclo. Già l'istruzione è abbastanza misteriosa, se si fanno anche trucchi su CX il codice diventa tanto illeggibile da essere pericoloso.
I problemi maggiori dati dalle LOOP si hanno proprio quando CX viene modificato all'interno del corpo del ciclo. Ciò può avvenire perché ci si dimentica che CX non va modificato oppure perché il flusso di controllo del programma salta da qualche parte e quando torna CX è stato modificato (come vedremo questo può succedere nelle chiamate a procedure e a funzioni del S.O.).
Una cosa importante da rimarcare è che LOOP non modifica i flag, per cui dopo una LOOP i flag sono quelli dell'istruzione precedente.
La LOOP che fa il massimo numero di giri è quella che parte con CX = 0 (è importante capire il perché, ma la scoperta verrà lasciata al lettore ..).

1 in verità nelle CPU X86 dal 486 in avanti l'uso della LOOP non è più veloce delle altre istruzioni equivalenti ed in certi casi è addirittura penalizzante.

Vediamo ora un esempio completo: un programma che fa la somma del valori contenuti in 10 word consecutive in memoria:

```
UnSoloSegmento SEGMENT
    ; zona per i dati, inizializzata a dieci valori "a caso":
    DieciWord DW 3, 9, 0F104h, 0A10h, 48, 9391, 5382, 110010010b, -29, 2
    ; zona per il codice
ASSUME CS:UnSoloSegmento, DS:UnSoloSegmento
INIZIO: ; il programma inizia da qui
    ; carico DS col valore giusto, per puntare a DieciWord:
    MOV AX, SEG UnSoloSegmento
    MOV DS, AX
    MOV BX, 0    ; inizializzo un "puntatore" che permetta di "spostarsi"
                 ; nel vettore DieciWord
    MOV CX, 10   ; inizializzo il contatore del ciclo
UnAltroElemento:
    ADD AX, [DieciWord + BX] ; sommo in AX una delle dieci word
    ; guardo se c'è errore, supponendo che i numeri siano con segno:
    JO ErroreAritmetico
    ADD BX, 2    ; con BX mi sposto nella locazione successiva
    LOOP UnAltroElemento
    ; dopo avere completato le dieci somme si passa qui
    ; il totale è in AX, poi il programma proseguirà
    ; .. altre cose ..
    ; e finirà:
Fine:
    MOV AH, 4Ch  ; servizio DOS "terminate program"
    INT 21h
ErroreAritmetico:

    ; .. qui si fa la visualizzazione di un messaggio di errore (omessa)..
    JMP Fine
UnSoloSegmento ENDS
END INIZIO
```

LOOP a 32 bit
Come tutti si aspettano, nelle CPU X86 a 32 bit che funzionano in modalità "protetta"[2], l'istruzione LOOP, e le sue "parenti" che vedremo in seguito, invece di CX usano ECX.
Dunque la LOOP a 32 bit può fare fino a 4G giri.

LOOPE e LOOPZ, LOOPNE e LOOPNZ
Le istruzioni che trattiamo in questo paragrafo permettono di uscire da un ciclo in due modi. Un modo è percorrere tutto il ciclo, mandando CX fino a zero come in una normale LOOP. L'altro modo per uscire è verificare una condizione sul flag di zero.
Usate con attenzione queste istruzioni permettono di realizzare cicli molto efficienti.
La loro sintassi è identica alla loop:

```
LOOPE <LabelCorpoCiclo>    ; Loop when Equal AND CX <> 0
                           ; salta se CX = 0 and ZF = 1

LOOPNE <LabelCorpoCiclo>   ; Loop when NOT Equal AND CX <> 0
                           ; salta se CX = 0 and ZF = 0
```

L'istruzione LOOPE è una LOOP che salta a <Label> se CX è zero e se il flag di zero è 1. Il che significa che LOOPE esce dal ciclo in due casi: se esso è terminato regolarmente oppure se ZF è zero.
Il programma può uscire dal ciclo nei due modi indicati eseguendo un'istruzione che sistemi il flag di zero subito prima della LOOPE.
Come indicato dalla N del codice mnemonico, LOOPNE funziona in modo inverso a LOOPE, per quel che riguarda il flag di zero.

2 La modalità protetta è il "vero" funzionamento a 32 bit 386

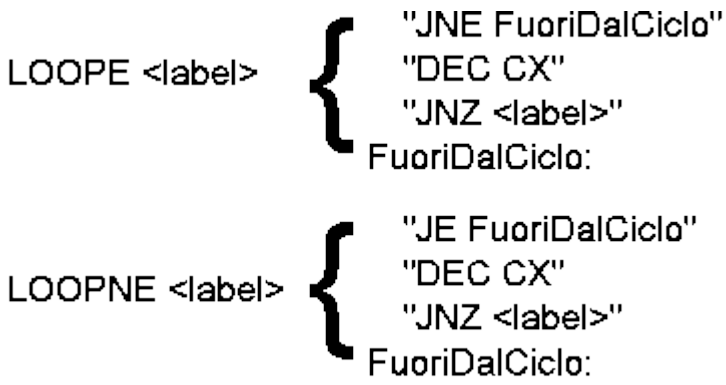

Figura 6: istruzioni equivalenti di LOOPE e LOOPNE

Analogamente a JE e JZ, LOOPE e LOOPZ sono la stessa istruzione, si userà il mnemonico LOOPE quando si vorrà porre l'accento sull'uscita con la disuguaglianza, LOOPZ per evidenziare la differenza da zero.
Le istruzioni come LOOPE non modificano i flag, per cui dopo l'uscita si può verificare la ragione per cui si è usciti dal ciclo con una semplice JZ (il flag di zero è stato sistemato prima della LOOPE).

Come esempio vediamo la ricerca di un carattere in una stringa della quale si sa la lunghezza, con la LOOPE potrò uscire dal ciclo sia nel caso che la stringa sia finita, sia nel caso che abbia trovato il carattere che cerco:

```
    MOV BX, offset Stringa
    DEC BX       ; punto un byte più su perché nel ciclo la INC viene prima della CMP
    MOV CX, 10   ; la stringa è lunga 10 caratteri, preparo una loop "da 10"
GiraAlMassimo10Volte:
    INC BX               ; punta al prossimo carattere della stringa
    CMP [Stringa + BX], 'A'  ; il flag zero viene sistemato da questa compare
    LOOPNZ GiraAlMassimo10Volte
    ; esce dal ciclo se la stringa è finita OPPURE se ho trovato una 'A'
    JE Trovata   ; se il flag di zero è ON la LOOPNE è uscita perché
                 ; la CMP ha trovato la 'A'
    ; se arrivo qui vuol dire che sono arrivato in fondo alla stringa

    ..

Trovata: ; qui tratto il caso della lettera trovata
```

Cicli "annidati" (nested)
Mentre si percorre un ciclo è possibile che sorga la necessita di iniziarne un altro.
Due cicli uno dentro all'altro vengono dette "**nidificati**" o "**annidati**"; questa strana parola è la traduzione letterale di "**nested**", usato in questi casi dagli anglosassoni ("nest" significa "nido"). Qualcuno chiama anche "innestati" questo tipo di cicli.

Cicli con un numero incognito di iterazioni
Se non sappiamo il numero di iterazioni da compiere prima di entrare in un ciclo la fase di aggiornamento e la valutazione della condizione di uscita tendono ad essere più complicati che nel caso dei normali cicli "a contare".
In questi frangenti ogni ciclo è diverso dall'altro e non si possono dare indicazioni generali, tranne sconsigliare l'uso della LOOP, che tende ad oscurare ancor di più il senso di un programma che potrebbe non essere semplice.