

IL LINGUAGGIO ASSEMBLY: PRIMI PROGRAMMI

PREMESSA

Come avete visto durante le lezioni di Informatica, un linguaggio di programmazione, per poter rappresentare efficacemente algoritmi, deve contenere strutture sintattiche in grado di descrivere in modo completo il tipo di dati su cui si intende lavorare, le azioni elementari che sui dati possono essere compiute e le strutture di controllo che individuano la sequenza con cui le azioni elementari devono essere eseguite.

A partire da questa unità didattica ci proponiamo di presentare, seguendo il medesimo schema concettuale, le nozioni che stanno alla base della programmazione assembly del microprocessore 8086 in ambiente MS-DOS. Per quanto riguarda la codifica e la prova dei programmi faremo riferimento ai prodotti della Borland Inc. (TASM, TLINK e TD), lasciando al lettore le eventuali piccole modifiche necessario per rendere il codice compatibile con altri tipi di assembler e debugger.

LA STRUTTURA DI UN PROGRAMMA ASSEMBLY

Un programma assembly, che utilizza solo le istruzioni di base, è generalmente formato dalle seguenti *sezioni* distinte:

- un'*intestazione*, in cui, tramite dei commenti, vengono specificati il nome e la funzione del programma;
- un *segmento dati*, in cui vengono descritti in modo formale gli oggetti su cui il programma dovrà operare;
- un *segmento di stack*, in cui viene definita la struttura dello stack associato al programma;
- un *segmento di codice*, in cui vengono descritte le azioni da compiere, collegandole tra loro con opportune strutture di controllo;
- una *chiusura*.

```
; nome del programma
    <intestazione>

    <segmento dati>

    <il segmento di stack>

    <il segmento di codice>
END
```

L'intestazione

Anche se l'intestazione, che serve esclusivamente a specificare con dei commenti le funzionalità del programma è opzionale e viene ignorata dall'assemblatore, è consigliabile porla sempre in testa al programma sorgente, in quanto costituisce un valido elemento di documentazione.

Il segmento dati

In questo segmento viene dichiarata la tipologia degli oggetti (variabili) su cui il programma stesso dovrà lavorare e racchiude in sé le due funzioni di dichiarazione e inizializzazione di variabili e costanti, in quanto consente di assegnare alle singole locazioni di memoria valori iniziali predefiniti direttamente in fase di dichiarazione.

La struttura del modulo dichiarativo è:

```
Dati SEGMENT
    <dichiarazione>
    ...
```

<dichiarazione>

Dati ENDS

Il segmento di stack

Questo segmento definisce un'area di memoria privilegiata, detta *stack*, che viene utilizzata sia dal programmatore che dal microprocessore per salvare e ripristinare all'occorrenza informazioni utili sullo stato del sistema.

Di questo segmento e del suo uso specifico in programmazione parleremo molto diffusamente nell'ultima parte del presente volume. Per ora, ci limitiamo a definirne la struttura, così da consentire al microprocessore di utilizzarlo autonomamente per garantire il corretto funzionamento dei nostri programmi:

```
Sistema SEGMENT STACK
                DW 100 DUP(?)
                Top LABEL WORD
Sistema ENDS
```

Il segmento di codice

Questo blocco che contiene, come abbiamo detto, le istruzioni del programma, ha una struttura interna ben definita e consente al microprocessore di:

- inizializzare correttamente i registri di segmento, ponendo in essi gli indirizzi di partenza dei segmenti Dati, Sistema e Codice;
- eseguire le istruzioni del nostro programma specifico;
- restituire il controllo al sistema operativo MS-DOS, considerando concluso il programma in esecuzione.

```
Codice SEGMENT
    ASSUME CS:Codice, DS:Dati, SS:Sistema, ES:Dati
    ; istruzioni di inizializzazione dei registri di segmento
    ...
    ; programma
    ...
    ; istruzioni di ritorno al DOS
    ...
Codice ENDS
```

La chiusura del programma

L'ultima riga del programma è composta dalla direttiva `END` che, oltre a definire la fine del programma sorgente, permette al programmatore di fornire al sistema l'indicazione del punto in cui, all'interno del segmento di codice, si trova l'istruzione da cui deve partire l'esecuzione del programma.

Il formato della direttiva è:

```
END <etichetta posizione di partenza>
```

LA DICHIARAZIONE DELLE VARIABILI

Tutte le variabili usate nel programma, eccezione fatta per i registri che possono essere considerati come variabili predefinite, devono essere dichiarate nel segmento dati in modo che l'assemblatore possa allocare in memoria il numero di byte necessari per la registrazione del loro valore.

Dato che il microprocessore opera solo con dati binari raggruppati in byte o parole di 16 bit, il linguaggio assembly, che ricalca le funzionalità del microprocessore 8086, prevede l'uso di *due soli tipi* di variabili: il tipo *byte* e il tipo *word*.

In particolare, la *dichiarazione di una variabile di tipo byte*, per la quale vengono riservati in memoria 8 bit, viene effettuata nel modo seguente:

```
<nome della variabile> DB <valore>
```

In modo analogo, la *dichiarazione di una variabile di tipo word*, per la quale vengono riservati in memoria 16 bit, viene effettuata nel modo seguente:

```
<nome della variabile> DW <valore>
```

Per comodità manterremo per i *nomi di variabili* le medesime convenzioni utilizzate per gli identificatori del Pascal ma scrivendo in maiuscolo solo il primo carattere.

Il *valore* di inizializzazione, invece, deve essere compatibile con la dimensione della variabile e deve appartenere ad una delle seguenti categorie di costanti:

- *costanti binarie*, rappresentate da stringhe di cifre binarie chiuse dal carattere b;

Per esempio:

00010011b valore binario a 8 bit
1000111000111010b valore binario a 16 bit

- *costanti ottali*, rappresentate da stringhe di cifre ottali chiuse dal carattere o;

Per esempio:

17o equivale al valore decimale 15

- *costanti decimali*, rappresentate da stringhe di cifre decimali chiuse dal carattere d;

Per esempio:

19d equivale al valore decimale 19

- *costanti esadecimali*, rappresentate da stringhe di cifre esadecimali, di cui la prima obbligatoriamente compresa tra O e 9, chiuse dal carattere h;

Per esempio:

0FFh equivale al valore decimale 255
14h equivale al valore decimale 20

- *costanti stringa*, rappresentate da stringhe di caratteri *racchiuse tra apici*. Dato che, in fase di interpretazione della stringa ad ogni carattere viene associato il corrispondente codice ASCII (un byte), una costante stringa usata in fase di inizializzazione di una variabile di tipo DB ne definisce implicitamente anche la lunghezza.

Per esempio:

'esempio' definisce per la variabile una lunghezza pari a 7

- Nel caso, poi, in cui si voglia definire una variabile priva di valore di inizializzazione, è sufficiente *inserire un punto interrogativo (?)* al posto del valore.

Per esempio:

Somma DW ? definisce la variabile Somma di tipo word senza assegnarle un valore

IL FORMATO DELLE ISTRUZIONI

All'interno del segmento di codice vengono poste le istruzioni del programma, ciascuna delle quali è in genere suddivisa in quattro campi:

- *un campo etichetta*, che rappresenta il nome simbolico di riferimento con cui individuare un'istruzione all'interno del programma. L'etichetta è opzionale, ma se presente, va obbligatoriamente fatta seguire dal carattere separatore : (due punti);
- *un campo codice operativo*, che contiene il nome simbolico dell'operazione da eseguire;
- *un campo operandi*, che contiene il riferimento ai vari operandi (*destinazione* e *sorgente*) coinvolti nell'operazione;
- *un campo commento*, aperto obbligatoriamente dal carattere ; (punto e virgola) che serve a descrivere un particolare aspetto del programma in quel punto, documentandone l'azione.

Di conseguenza, il formato generale di un'istruzione risulta essere il seguente:

<etichetta> : <cod_operativo> <dest>, <sorg> ; commento

L'INIZIALIZZAZIONE DEI SEGMENTI

La parte relativa all'inizializzazione dei segmenti è formata da poche istruzioni standard che consentono di porre all'interno dei registri CS, SS, DS e ES gli indirizzi associati dall'assemblatore ai segmenti Dati, Sistema e Codice.

```
Codice SEGMENT
    ; definizione dei segmenti
    ASSUME CS:Codice, SS:Sistema, DS:Dati, ES:Dati

    ; inizializzazione dei registri di segmento
Inizio: MOV AX, Sistema;
        MOV SS, AX
        LEA AX, Top
        MOV SP, AX
        MOV AX, Dati
        MOV DS, AX
        MOV ES, AX

    ; programma
    ...
```

Come si può notare, l'apertura della fase di inizializzazione è caratterizzata dalla presenza dell'etichetta inizio che definisce il punto da cui deve partire l'esecuzione.

La riga precedente, aperta dalla parola chiave ASSUME, rappresenta invece una direttiva che comunica all'assemblatore a quali segmenti deve fare riferimento nel calcolo degli indirizzi.

IL RITORNO AL SISTEMA OPERATIVO

Anche questa parte del programma, che restituisce il controllo al sistema operativo al termine dell'esecuzione, è composta da poche istruzioni standardizzate che, nel caso del sistema operativo MS-DOS risultano essere:

```
    ...
    ; ritorno al sistema operativo
    MOV AL, 0oh
    MOVAH, 4Ch
    INT21h
Codice ENDS
```

LA REALIZZAZIONE DI UN PROGRAMMA ASSEMBLY

Come avviene per i programmi scritti in un linguaggio ad alto livello, anche la realizzazione di un programma assembly attraversa le diverse fasi:

- **caricamento** del programma sorgente;
- **conversione** del programma sorgente nel corrispondente programma oggetto;
- **conversione** del programma oggetto in un programma eseguibile;
- **esecuzione e prova** del programma.

Analizziamo in modo più approfondito queste quattro fasi nello sviluppo di un programma assembly da realizzare in ambiente MS-DOS.

Caricamento del programma sorgente

Questa fase consiste nel caricamento, attraverso un qualsiasi editor di sistema (ad esempio l'EDIT del DOS) del testo del *programma sorgente* in un file caratterizzato dall'estensione ASM (per esempio PROVA.ASM).

Conversione del programma sorgente in un programma oggetto

Una volta caricato il programma sorgente occorre, tramite l'assemblatore, convertirlo nel corrispondente *programma oggetto* (per esempio PROVA.OBJ).

Nel caso in cui l'assemblatore abbia segnalato in fase di traduzione degli errori, occorrerà provvedere a correggerli richiamando il file sorgente tramite l'editor utilizzato precedentemente.

Conversione del programma oggetto in un programma eseguibile

In questa fase sarà sufficiente richiamare il programma linker associato all'assemblatore, che si preoccuperà di convertire il programma oggetto in un *programma eseguibile* (per esempio PROVA.EXE).

Esecuzione e prova del programma

Dato che i programmi assembler lavorano direttamente con i registri della CPU e con le singole locazioni di memoria, per verificare la correttezza di un programma occorrerà ricorrere ad uno strumento software che consenta di accedere alla memoria del microprocessore e di analizzare, dopo ogni istruzione, le modifiche apportate dal programma al contenuto dei registri e delle singole locazioni di memoria. Per questo motivo a tutti gli assembler è associato un *programma di debugging* che consente di eseguire un programma *passo dopo passo* e di analizzare, in ogni momento, il contenuto dei registri e della memoria.

Proposta di lavoro 1

Dopo aver caricato con un programma editor il seguente programma sorgente:

```
; programma per lo scambio di 2 byte in memoria
Dati SEGMENT
    Primo DB 01 h
    Secondo DB 02h
Dati ENDS

Sistema SEGMENT STACK
    DW 100 DUP(?)
    Top LABEL WORD
Sistema ENDS

Codice SEGMENT
    ; definizione dei segmenti
    ASSUME CS:Codice, SS:Sistema, DS:Dati, ES:Dati

    ;inizializzazione dei registri di segmento
Inizio: MOV AX, Sistema;
        MOV SS, AX
        LEA AX, Top
        MOV SP, AX
        MOV AX, Dati
        MOV DS, AX
        MOV ES, AX

        ;programma
        MOV AL, Primo ; carica in AL il valore della variabile Primo
        MOV AH, Secondo ; carica in AH il valore della variabile Secondo
        MOV Primo, AH ; carica nella variabile Primo il valore di AH
        MOV Secondo, AL ; carica nella variabile Secondo il valore di AL

        ; ritorno al sistema operativo
        MOV AL, 00h
        MOV AH, 4Ch
        INT 21h
Codice ENDS
END Inizio
```

assegnargli il nome SCAMBIA1.ASM, convertirlo in un programma eseguibile e, infine, controllarne l'effettivo funzionamento passo dopo passo con programma di debugging.

Risposta

Anche se nella nostra risposta, per quanto riguarda la forma, faremo riferimento prodotti TASM, TLINK e TD della Borland, la logica delle operazioni da compiere risultati ottenuti risultano essere facilmente trasferibili in un qualsiasi altro ambiente di lavoro. La prima operazione da compiere è quella di trasformare il programma sorgente nel relativo programma oggetto. Richiamiamo pertanto l'assemblatore TASM fornendogli come parametro il nome del file sorgente (se l'estensione è ASM può essere omessa).

```
C>TASM SCAMBIA1
```

Se non sono stati riscontrati errori la risposta sarà:

```
Turbo Assembler Version 3.1 Copyright (e) 1988, 1992 Borland International
Assembling file: scambia1.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 353k
C>
```

Chiamiamo ora il linker che ci fornirà direttamente il programma eseguibile:

```
C>TLINK SCAMBIA1
```

La risposta sarà:

```
Turbo Link Version 5.1 Copyright (e) 1992 Borland International
C>
```

A questo punto non ci resta che entrare nell'ambiente di debugging per testare il funzionamento del programma:

```
C>TD PROVA1
```

Il programma di debugging ci presenterà le seguenti finestre:

- in alto a sinistra il segmento di codice con il puntatore posizionato sulla prossima istruzione da eseguire;
- in basso a sinistra l'immagine del segmento dati;
- a destra l'immagine dei registri.

```

CPU 80386
cs:0000 88F152      mov ax, 52F1      ax 0000      c=0
cs:0003 SEDO      mov ss, ax        bx 0000      z=0
cs:0005 B8C800    mov ax, 00C8      cx 0000      s=0
cs:0008 8BEO      mov sp, ax        dx 0000      o=0
cs:000A B8F052    mov ax, 52F0      si 0000      p=0
cs:000D 8ED8      mov ds, ax        di 0000      a=0
cs:000F 8ECO      mov es, ax        bp 0000      i=0
cs:0011 A00000    mov al, [0000]    sp 00C8      d=0
cs:0014 8A260100  mov ah, [0001]    ds 52E0
cs:0018 88260000  mov [0000], ah    es 52E0
cs:001C A20100    mov [0001], al    ss 52F1
cs:001F B000      mov al, 00        cs 52FE
cs:0021 B44C      mov ah, 4C        ip 0000

ds:0005  20  20  20  20  20  20  20  20
ds:0008  20  20  20  20  20  20  20  20
ds:0010  20  20  20  20  20  20  20  20      ss:00CA  0000
ds:0018  20  20  20  20  20  20  20  20      ss:00C8  0000
```

Rispetto alla forma usata per rappresentare le istruzioni va solo notato che il debugger ha sostituito i nomi simbolici (identificatori) con gli indirizzi calcolati, posti tra parentesi quadre.

Con il tasto F8 possiamo ora guidare l'esecuzione passo a passo e controllare l'esito di ogni operazione compiuta dal programma.

LE ISTRUZIONI DI TRASFERIMENTO

Come possiamo intuire dall'analisi della struttura del microprocessore vista nell'unità didattica precedente, una delle operazioni fondamentali che il linguaggio assembly deve consentire di compiere è senz'altro quella che permette di trasferire *dati* ed *indirizzi* dalla memoria ai registri e viceversa.

In questo paragrafo quindi analizzeremo nel dettaglio le varie istruzioni di trasferimento dati che l'assembly del microprocessore 8086 mette a disposizione del programmatore.

Istruzioni di base per il trasferimento di dati

L'istruzione fondamentale per effettuare il trasferimento di dati a 8 bit o a 16 bit è **MOV**, il cui formato è:

MOV destinazione, sorgente

ove:

- *sorgente* può essere una costante o il riferimento (identificatore) a un registro o a una variabile;
- *destinazione* invece può essere *solo* il riferimento (identificatore) a un registro o a una variabile.

Inoltre, *sorgente e destinazione* devono sottostare alle seguenti regole:

- non possono riferirsi entrambi a due locazioni di memoria (variabili);
- devono essere fra loro compatibili in termini di dimensione.

Di conseguenza non è possibile, per esempio, trasferire direttamente un dato una variabile all'altra o il contenuto di un registro a 8 bit in uno a 16 e viceversa.

Nel caso di utilizzo di un riferimento a variabili, inoltre, bisogna tener ben presente il modo in cui esso viene interpretato dall'assemblatore durante la traduzione programma sorgente.

Infatti, come abbiamo visto nell'unità didattica 6, durante il primo passo l'assemblatore produce la cosiddetta *tabella dei simboli* in cui ad ogni nome simbolico (*identificatore*) viene associato un indirizzo di memoria, che, nel caso delle variabili corrisponde alla posizione occupata, all'interno del segmento dati, dal primo byte dell'area di memoria destinata a contenere i valori della variabile dichiarata (*offset address*).

Nel secondo passo poi, l'assemblatore, ogni volta che incontrerà quell'identificatore, gli assocerà l'indirizzo corrispondente nella tabella dei simboli, senza preoccuparsi della reale lunghezza della variabile.

Appare quindi evidente che quando si usa in un'istruzione di trasferimento dati un riferimento attraverso un identificatore, occorrerà specificare all'assemblatore anche il *numero di byte* che, a partire da quell'indirizzo, devono essere coinvolti nell'operazione.

L'Assembly dell'8086 per risolvere questo problema utilizza la convenzione di far precedere l'identificatore dalle parole **WORD PTR** nel caso in cui si vuole che vengano coinvolti nell'operazione 2 byte e **BYTE PTR** nel caso in cui ci si vuole riferire ad un singolo byte. Osserviamo a questo proposito che mentre la prima notazione risulta essere indispensabile, la seconda può essere omessa in tutti i casi in cui ciò non generi ambiguità.

Per esempio:

```
MOV AX, 12FFh    ;copia il valore a 16 bit 12FF in AX
MOV BL, Numero  ;copia un byte della variabile Numero in BL
MOV AH, AL      ; copia il valore di AL in AH
MOV WORD PTR Somma, AX ;copia il valore di AX nei due byte
                    ; indirizzati da Somma
```

Istruzioni di base per il trasferimento di indirizzi

In un calcolatore la gestione dei dati è meno delicata di quella degli indirizzi e ciò spiega il limitato numero di operazioni che è possibile compiere su questi ultimi. In particolare, per quanto riguarda le operazioni di trasferimento, occorre evidenziare il fatto che il linguaggio consente *solo* di trasferire, attraverso l'istruzione **LEA** (*Load Effective Address*), l'indirizzo di riferimento di una variabile all'interno di uno dei registri generali a 16 bit. Il formato dell'istruzione è:

LEA destinazione, sorgente

ove:

- *sorgente* è il riferimento (identificatore) a una variabile;
- *destinazione* è il riferimento (identificatore) a un registro a 16 bit (AX, BX, CX o DX).

Per esempio:

```
LEA AX, Somma   ; copia l'indirizzo di riferimento della variabile
                  ; Somma in AX
```

Istruzioni per l'inizializzazione dei registri di segmento

L'istruzione MOV viene anche utilizzata per trasferire un indirizzo in uno dei quattro registri di segmento, prelevandolo da una parola di memoria o, come più comunemente accade, da uno dei registri a 16 bit AX, BX, CX, DX.

Per esempio, per inizializzare il registro DS con l'indirizzo associato al segmento Dati, le istruzioni sono:

```
Dati      MOV AX, Dati      ; viene caricato in AX l'indirizzo associato a
          MOV DS, AX      ; viene caricato in DS il contenuto di AX
```

Proposta di lavoro 2

Modificare il programma precedente in modo che l'operazione di avvenga su dati a 16 bit.

Durante la prova riscontrerete sicuramente delle anomalie: prendetene nota.

Risposta

Il programma può essere modificato nel modo seguente:

```
;programma per lo scambio di 2 word in memoria
Dati SEGMENT
    Primo DW 0102h
    Secondo DW 0304h
Dati ENDS

Sistema SEGMENT STACK
    DW 100 DUP(?)
    Top LABEL WORD
Sistema ENDS

Codice SEGMENT
    ;definizione dei segmenti
    ASSUME CS:Codice, SS:Sistema, DS:Dati, ES:Dati
    ;inizializzazione dei registri di segmento
Inizio: MOV AX, Sistema
        MOV SS, AX
        LEA AX, Top
        MOV SP, AX
        MOV AX, Dati
        MOV DS, AX
        MOV ES, AX
    ; programma
        MOV AX, WORD PTR Primo    ; carica in AX due byte di Primo
        MOV BX, WORD PTR Secondo  ; carica in BX due byte di Secondo
        MOV WORD PTR Primo, BX    ; inverte i dati in memoria
        MOV WORD PTR Secondo, AX
    ; ritorno al sistema operativo
        MOV AL, 0ch
        MOV AH, 4Ch
        INT21h
Codice ENDS

END Inizio
```

Nota sulla memorizzazione di dati a 16 bit

Durante la prova del programma precedente, osservando i valori presenti nelle locazioni di memoria associate alle variabili, si possono riscontrare le seguenti anomalie:

- all'inizio dell'esecuzione, i valori con cui sono state inizializzate le variabili in memoria non corrispondono a quelle impostate nella dichiarazione, poiché i byte appaiono scambiati tra loro;
- durante il trasferimento di valori a 16 bit dalla memoria ai registri e viceversa, avviene automaticamente lo scambio dei due byte che compongono la parola.

L'8086, come molti altri microprocessori, usa per le parole a 16 bit, un meccanismo di memorizzazione che prevede di riportare in memoria centrale i due byte del dato in modo che quello meno significativo preceda quello più significativo e ciò nella logica che alla parte alta del dato

corrisponda l'indirizzo più alto e non viceversa.

Esiste però un caso in cui, se non si presta la dovuta attenzione, questo modo di memorizzare i dati può provocare seri problemi. Infatti se, dopo aver trasferito in memoria una parola a 16 bit (quindi con i due byte scambiati tra loro), si vanno ad analizzare singolarmente le singole parti, trasferendole nella CPU separatamente, si può incorrere nell'errore di interpretare il primo byte come quello più significativo, mentre non lo è.

per esempio la sequenza di istruzioni:

```
MOV WORD PTR Primo, 0102h
MOV AL, Primo
```

porta nel registro AL il primo byte della variabile Primo che contiene 02, parte meno significativa del dato.

Di conseguenza, si consiglia di far precedere sempre un identificatore dalla dichiarazione dimensionale esplicita, in modo da garantire che tutte le operazioni di trasferimento vengano fatte nel modo corretto.

LE ISTRUZIONI ARITMETICHE

Come abbiamo visto nelle unità didattiche precedenti, all'interno del calcolatore i valori interi positivi vengono rappresentati nella loro naturale configurazione binaria, mentre quelli negativi assumono la notazione in complemento a 2.

Di conseguenza, anche le istruzioni che consentono di effettuare le quattro operazioni elementari tra valori interi (a 8 o a 16 bit) faranno riferimento a questo tipo di codifica e sarà compito del programmatore interpretare, di volta in volta, i risultati in maniera corretta.

Addizione

Il formato generale di un'istruzione aritmetica di addizione è:

ADD destinazione, sorgente

ed equivale alla forma Pascal:

destinazione := destinazione + sorgente

ove:

- *sorgente* può essere una costante o il riferimento (identificatore) a un registro o a una variabile;
- *destinazione* invece può essere solo il riferimento (identificatore) a un registro o a una variabile.

Inoltre, *sorgente* e *destinazione* devono sottostare alle seguenti regole:

- non possono riferirsi entrambi a due locazioni di memoria (variabili);
- devono essere fra loro compatibili in termini di dimensione.

Va notato a questo proposito che, poiché il risultato deve mantenere la stessa dimensione degli operandi, nel caso in cui si richieda un riporto sulla cifra più significativa il valore viene troncato a sinistra, perdendo proprio la cifra di maggior valore.

Per esempio, supponendo che:

```
AL = 00100011
AH = 11111011
```

l'istruzione di addizione:

```
ADD AH, AL
```

pone in AH il valore 00011110, in cui, per traboccamento, si è perso il bit più significativo del risultato.

In base poi al risultato ottenuto, vengono modificati i flag, come illustrato nel compendio completo delle istruzioni presente nell'appendice I.

In particolare, ricordiamo per ora che:

SF assume il valore del bit più significativo del risultato, che ne rappresenta il segno;

ZF assume il valore 0 se il risultato dell'operazione è diverso da zero, 1 se il risultato dell'operazione è 0;

CF assume il valore 1 se si è verificato un riporto sul bit più significativo, 0 negli altri casi.

Per esempio, al termine dell'addizione precedente, i flag avranno il seguente valore:

SF=0 ZF=0 CF=1

Esiste anche un'altra istruzione di addizione che consente di tener conto, nel calcolo, anche del riporto proveniente dall'istruzione precedente.

Il suo formato è:

ADC destinazione, sorgente

ed equivale alla forma Pascal:

destinazione := destinazione + sorgente + valore del flag di carry

Per esempio, supponendo che:

AL = 00100011
AH = 11111011

l'istruzione di addizione:

ADD AH, AL

pone in AH il valore 00011110 e il flag di carry uguale a 1. Se subito dopo viene eseguita l'istruzione:

ADD AH, AL

viene posto in AH il valore 10111110 e il flag di carry uguale a 0.

Proposta di lavoro 3

Studiando la programmazione in Pascal avrete certamente visto come il linguaggio, oltre al tipo integer organizzato su 16 bit, preveda anche l'uso di un tipo longint, che occupa 4 byte e consente di superare i limiti assai ristretti imposti dal tipo precedente (valori compresi tra -32768 e +32767).

Dopo aver analizzato il problema relativo alla rappresentazione di questo nuovo tipo di dato all'interno del calcolatore, scrivere un programma che calcoli la somma di due numeri a 32 bit.

Risposta

Analisi del problema

Dato che il microprocessore riconosce solo valori a 8 o a 16 bit, per rappresentare un valore a 32 bit dovremo ricorrere ad un artificio che consenta di dividere il numero stesso in due parti (alta e bassa) di 16 bit ciascuno.

In questo esercizio, in particolare, divideremo semplicemente a metà i 32 bit che compongono il numero.

Per esempio il valore:

000011111100000011111111110001101

viene così suddiviso:

parte alta	parte bassa
00001111110000001	11111111110001101

L'addizione tra due valori a 16 bit dovrà allora prevedere le seguenti fasi:

- addizione tra le due parti basse, che ovviamente può dare origine ad un riporto sulla cifra meno significativa della parte alta;
- addizione tra le due parti alte, considerando anche l'eventuale riporto proveniente dall'operazione precedente.

Per esempio per ottenere la somma tra i seguenti valori:

parte alta	parte bassa
00001111110000001	11111111110001101
0011101010101011	1111111101010100

si opererà nel modo seguente:
addizione tra la parte bassa dei due valori:

```
parte bassa
1111111011100001 con riporto pari a 1
```

addizione tra le parti alte con aggiunta del riporto

```
parte alta
0100101000101101
```

Costruzione dell'algoritmo

Dato che per il momento non abbiamo ancora a disposizione gli strumenti necessari per gestire l'input/output, utilizzeremo il debugger per poter, dopo la fase di inizializzazione dei registri di segmento, caricare direttamente i valori esadecimali da sommare nelle variabili definite per i dati e per verificare la correttezza del risultato prima di uscire dal programma. In queste ipotesi un primo modello del programma risulta essere il seguente:

```
; programma per la somma di due valori a 32 byte
Dati SEGMENT
    ...
Dati ENDS

Sistema SEGMENT STACK
    ...
Sistema ENDS

Codice SEGMENT
    ; definizione dei segmenti
    ; inizializzazione dei registri di segmento
Inizio:    ...
            ; qui viene simulato l'input dei dati con
            ; l'inserimento diretto
            ; dei valori in memoria in fase di debugging
            ; somma delle parti basse e memorizzazione del risultato
            ; somma delle parti alte e del riporto e memorizzazione del ris.
            ; qui viene controllata l'esattezza del risultato
            ; ritorno al sistema operativo
Codice ENDS

END Inizio
```

Come consuetudine della tecnica top-down, sviluppando i singoli commenti otteniamo il modello conclusivo, in cui per brevità, abbiamo tralasciato di riportare le parti di dichiarazione del segmento di stack, di inizializzazione dei registri di segmento e di ritorno al DOS, che risultano essere identici in ogni programma.

```
; programma per la somma di due valori a 32 byte
Dati SEGMENT
    Add1_alto DW 0000h
    Add1_basso DW 0000h
    Add2_alto DW 0000h
    Add2_basso DW 0000h
    Ris_alto DW 0000h
    Ris_basso DW 0000h
Dati ENDS

Sistema SEGMENT STACK
    ...
Sistema ENDS

Codice SEGMENT
    ; definizione dei segmenti
    ASSUME .....
    ; inizializzazione dei registri di segmento
Inizio:    ...
            ; qui viene simulato l'input dei dati con
```

```

; l'inserimento diretto
; dei valori in memoria in fase di debugging
; somma delle parti basse e memorizzazione del risultato
MOV AX, WORD PTR Add1_basso
ADD AX, WORD PTR Add2_basso
MOV WORD PTR Ris_basso, AX
; somma delle parti alte e del riporto e memorizzazione del ris.
MOV AX, WORD PTR Add1_alto
ACD AX, WORD PTR Add2_alto
MOV WORD PTR Ris_alto, AX
; qui viene controllata l'esattezza del risultato
; ritorno al sistema operativo
Codice ENDS
END Inizio

```

Sottrazione

Il formato generale di un'istruzione aritmetica di sottrazione è:

SUB *destinazione, sorgente*

ed equivale alla forma Pascal:

destinazione := destinazione - sorgente

in cui *sorgente* e *destinazione* devono sottostare alle medesime regole enunciate per l'addizione.

Va notato inoltre che, se nell'esecuzione dell'istruzione si richiede un riporto sulla cifra più significativa, l'operazione viene conclusa come se la cifra mancante del primo operando fosse pari al

Per esempio, supponendo che:

```

AH = 00100011
AL = 11111011

```

l'istruzione di sottrazione:

```
SUB AH, AL
```

pone in AH il valore 00101000

Infine ricordiamo che, in base al risultato, vengono modificati i flag, esattamente come avveniva nel caso dell'addizione.

Per esempio, al termine della sottrazione precedente, si avrà:

```
SF = 0   ZF = 0   CF = 1
```

Come per l'addizione, anche per la sottrazione esiste una particolare forma che consente di tener conto, nel calcolo, anche del riporto proveniente dall'istruzione precedente.

Il suo formato è:

SBB *destinazione, sorgente*

ed equivale alla forma Pascal:

destinazione := destinazione - sorgente - valore del flag di carri/

Per esempio, se subito dopo l'istruzione presentata nell'esempio precedente venisse eseguita l'istruzione:

```
SBB AH, AL
```

verrebbe posto in AH il valore 00101 100

Proposta di lavoro 4

Scrivere un programma che calcoli la differenza di due numeri a 32 bit.

Risposta

Data l'evidente somiglianza con il programma relativo all'addizione presentiamo direttamente il listato del programma assembly.

```
; programma per il calcolo della differenza di due valori a 32 byte
Dati SEGMENT
    Op1_alto DW 0000h
    Op1_basso DW 0000h
    Op2_alto DW 0000h
    Op2_basso DW 0000h
    Ris_alto DW 0000h
    Ris_basso DW 0000h
Dati ENDS

Sistema SEGMENT STACK
    ...
Sistema ENDS

Codice SEGMENT
    ; definizione dei segmenti
    ASSUME .....

    ; inizializzazione dei registri di segmento
Inizio:    ...

                ; qui viene simulato l'input dei dati con
                ; l'inserimento diretto
                ; dei valori in memoria in fase di debugging

    ; sottrazione tra le parti basse e memorizzazione del risultato
    MOV AX, WORD PTR Op1_basso
    SUB AX, WORD PTR Op2_basso
    MOV WORD PTR Ris_basso, AX

    ; sottrazione tra le parti alte e il riporto e memorizzazione del
ris.
    MOV AX, WORD PTR Op1_alto
    SBB AX, WORD PTR Op2_alto
    MOV WORD PTR Ris_alto, AX

                ; qui viene controllata l'esattezza del risultato

    ; ritorno al sistema operativo
Codice ENDS

END Inizio
```

Moltiplicazione

Il linguaggio assembly prevede istruzioni diverse per la moltiplicazione, a seconda che i fattori vengano interpretati come numeri naturali (senza segno) o come numeri relativi (con segno).

In particolare, l'istruzione di moltiplicazione tra numeri naturali, che prevede sempre implicitamente l'uso dell'accumulatore, ha il seguente formato:

MUL sorgente

ove *sorgente* può essere il riferimento a un registro o a una variabile.

Il modo di operare del processore dipende, in questo caso, dalla dimensione dell'operando sorgente, dato che:

- se *sorgente* è di tipo byte (registro a 8 bit o riferimento BYTE TPR), il valore sorgente viene moltiplicato per quello contenuto in AL e il risultato, calcolato su 16 bit, viene posto in AX.

Gli unici flag modificati in modo significativo dall'operazione sono CF e OF che vengono entrambi posti a 0 se nel risultato la parte alta (AH) è uguale a zero, mentre vengono posti a 1 negli altri casi;

- se *sorgente* è di tipo word (registro a 16 bit o riferimento WORD TPR), il valore di *sorgente* viene moltiplicato per quello contenuto in AX e del risultato, calcolato su 32 bit, i sedici bit più significativi vengono posti in DX, mentre i sedici bit meno significativi vengono posti in AX.

Gli unici flag modificati in modo significativo dall'operazione sono CF e OF che vengono entrambi posti a 0 se nel risultato la parte alta (DX) è uguale a zero, mentre vengono posti a 1 negli altri casi.

Per esempio, supponendo che:

```
AL = 11100011
BL = 01101111
```

l'istruzione di moltiplicazione:

```
MUL BL
```

produrrebbe i seguenti effetti:

```
AX = 0110001001101101
OF = CF = 1
```

In modo analogo, supponendo che:

```
AX = 1110001111110000
BX = 0110111111111111
```

l'istruzione di moltiplicazione:

```
MUL BX
```

produrrebbe i seguenti effetti:

```
AX = 0001110000010000
DX = 0110001110111000
OF = CF = 1
```

L'istruzione di moltiplicazione tra numeri relativi:

IMUL sorgente

opera in modo del tutto analogo alla precedente, con l'unica differenza che il bit *più significativo degli operandi viene considerato come bit di segno* e i flag OF e CF vengono azzerati solo se la parte più significativa del risultato rappresenta un'estensione a 16 bit del bit di segno (tutti i bit pari a 0 o a 1 e uguali al bit più significativo della parte bassa).

Per esempio, supponendo che:

```
AL = 11100011
BL = 01101111
```

l'istruzione di moltiplicazione:

```
IMUL BL
```

produrrebbe i seguenti effetti:

```
AX = 1111001101101101
OF = CF = 1
```

In modo analogo, supponendo che:

```
AX = 1110001111110000
BX = 0110111111111111
```

l'istruzione di moltiplicazione:

```
MUL BX
```

produrrebbe i seguenti effetti:

```
AX = 0001110000010000
DX = 1111001110111001
OF = CF = 1
```

Proposta di lavoro 5

Scrivere un programma che consenta di calcolare il prodotto tra due valori senza segno a 32 bit.

Risposta

Analisi del problema

Il problema, supponendo di rappresentare i due fattori e il risultato in modo analogo a quanto fatto nell'esercizio precedente, può essere riportato a quello della moltiplicazione tra valori decimali di due cifre, in cui le unità rappresentano la parte bassa e le decine la parte alta.

Osserviamo innanzitutto che il valore più grande assunto dal risultato risulta pari $99 * 99 = 9810$ che presenta un numero di cifre doppio rispetto agli operandi. Nel nostro caso sarà quindi necessario impostare il risultato su quattro word di memoria.

Analizziamo ora come nell'aritmetica tradizionale viene composto il risultato di $52 * 75$ a partire dagli operandi.

Se scriviamo gli operandi in forma polinomiale otteniamo:

$$\begin{aligned}52 &= 2 + 5 * 10 \\75 &= 5 + 7 * 10\end{aligned}$$

da cui, moltiplicando termine a termine:

$$52 * 75 = 2 * 5 + (2 * 7 + 5 * 5) * 10 + 5 * 7 * 100$$

Da questa espressione si possono ricavare i passi generali che, operando sulle singole cifre, occorre compiere per giungere al risultato:

1. calcolo delle unità come prodotto tra le unità: l'operazione può dar luogo ad un eventuale riporto (nel nostro esempio 1) di cui si dovrà tener conto nel calcolo successivo;
2. calcolo delle decine come prodotto incrociato tra decine e unità, a cui va sommato l'eventuale riporto proveniente dal passo precedente: l'operazione può dar luogo ad un eventuale riporto (nel nostro esempio 4) di cui si dovrà tener conto nel calcolo successivo;
3. calcolo delle centinaia come prodotto delle decine, a cui va sommato l'eventuale riporto proveniente dal passo precedente: l'operazione può dar luogo ad un eventuale riporto (nel nostro esempio 3) di cui si dovrà tener conto nel calcolo successivo;
4. calcolo delle migliaia come riporto del passo precedente.

Costruzione dell'algoritmo

Esplicitando i passi precedenti nel caso di valori a 32 bit (due "cifre" a 16 bit) otteniamo il primo modello del programma:

```
; programma per il prodotto di due valori a 32 byte
Dati SEGMENT
    ...
Dati ENDS

Sistema SEGMENT STACK
    ...
Sistema ENDS

Codice SEGMENT
    ; inizializzazione dei registri di segmento
Inizio:    ...

                ; qui viene simulato l'input dei dati con
                ; l'inserimento diretto
                ; dei valori in memoria in fase di debugging

    ; calcolo del prodotto delle prime "cifre" degli operandi
    ; memorizzazione della parte bassa nella prima "cifra" del risultato
    ; memorizzazione della parte alta (riporto) in un'area di transito
    ; calcolo del prodotto incrociato tra prime e seconde "cifre" degli
operandi

    ; aggiunta del riporto
    ; memorizzazione della parte bassa nella seconda "cifra" del ris.
    ; memorizzazione della parte alta (riporto) in un'area di transito
    ; calcolo del prodotto tra le parti alte degli operandi
```

```

; aggiunta del riporto
; memorizzazione della parte bassa nella terza "cifra" del risultato
; memorizzazione della parte alta (riporto) nella quarta "cifra" del
ris.

; qui viene controllata l'esattezza del risultato

; ritorno al sistema operativo
Codice ENDS

END Inizio

```

È evidente a questo punto che il problema più grosso da risolvere è quello relativo al prodotto incrociato, dato che in questo caso il riporto può superare anche i limiti dei 16 bit.

Riprendendo il parallelo con l'aritmetica decimale, è facile riconoscere che esistono casi (ad esempio $99 \cdot 99$) in cui il valore del prodotto incrociato supera il limite delle due cifre (nel nostro caso $81+81 = 162$). È sufficiente allora calcolare separatamente i due prodotti ed effettuare la somma su un'area di quattro cifre, di cui le prime due rappresentano il riporto.

In base a questa considerazione possiamo completare il nostro programma nel modo seguente:

```

Dati SEGMENT
    Cifra1_fatt1 DW 0000h
    Cifra2_fatt1 DW 0000h
    Cifra1_fatt2 DW 0000h
    Cifra2_fatt2 DW 0000h
    Cifra1_ris DW 0000h
    Cifra2_ris DW 0000h
    Cifra3_ris DW 0000h
    Cifra4_ris DW 0000h
    Cifra1_rip DW 0000h
    Cifra2_rip DW 0000h
    Cifra1_somma DW 0000h
    Cifra2_somma DW 0000h
    Cifra3_somma DW 0000h
Dati ENDS
Sistema SEGMENT STACK
    ...
Sistema ENDS

Codice SEGMENT
    ; definizione dei segmenti
    ASSUME .....

    ; inizializzazione dei registri di segmento
Inizio:    ...

; qui viene simulato l'input dei dati con
; l'inserimento diretto
; dei valori in memoria in fase di debugging

; calcolo del prodotto della parte bassa degli operandi
MOV AX, WORD PTR Cifra1_fatt1
MUL WORD PTR Cifra1_fatt2

; memorizzazione della parte bassa nella prima "cifra" del risultato
MOV WORD PTR Cifra1_ris, AX

; memorizzazione della parte alta (riporto) in un'area di transito
MOV WORD PTR Cifra1_rip, DX

; calcolo del primo prodotto incrociato tra prime e seconde "cifre"
degli operandi
MOV AX, WORD PTR Cifra1_fatt1
MUL WORD PTR Cifra2_fatt2

; memorizzazione delle due parti del risultato nei registri BX e CX
MOV BX, AX ; parte bassa
MOV CX, DX ; parte alta

; calcolo del secondo prodotto incrociato
MOV AX, WORD PTR Cifra2_fatt1
MUL WORD PTR Cifra1_fatt2

; calcolo della somma
ADD AX, BX
MOV WORD PTR Cifra1_somma, AX
ADC DX, CX

```

```

MOV WORD PTR Cifra2_somma, DX
MOV AX, 0000h
ADC AX, 0000h
MOV WORD PTR Cifra3_somma, AX

; aggiunta del riporto
MOV AX, WORD PTR Cifra1_somma
ADD AX, WORD PTR Cifra1_rip
MOV WORD PTR Cifra1_somma, AX
MOV AX, WORD PTR Cifra2_somma
ADC AX, WORD PTR Cifra2_rip
MOV WORD PTR Cifra2_somma, AX
MOV AX, WORD PTR Cifra3_somma
ADC AX, WORD PTR Cifra3_rip
MOV WORD PTR Cifra3_somma, AX

; memorizzazione della parte bassa nella seconda "cifra" del ris.
MOV AX, WORD PTR Cifra1_somma
MOV WORD PTR Cifra2_ris, AX

; memorizzazione della parte alta (riporto)
MOV AX, WORD PTR Cifra2_somma
MOV WORD PTR Cifra1_rip, AX
MOV AX, WORD PTR Cifra3_somma
MOV WORD PTR Cifra2_rip, AX

; calcolo del prodotto tra le parti alte degli operandi
MOV AX, WORD PTR Cifra2_fatt1
MUL WORD PTR Cifra2_fatt2

; aggiunta del riporto
ADD AX, WORD PTR Cifra1_rip
ADC DX, WORD PTR Cifra2_rip

; memorizzazione della parte bassa nella terza "cifra" del risultato
MOV WORD PTR Cifra3_ris, AX

; memorizzazione della parte alta (riporto) nella quarta "cifra" del
ris.
MOV WORD PTR Cifra4_ris, DX

; qui viene controllata l'esattezza del risultato

; ritorno al sistema operativo
Codice ENDS

END Inizio

```

Divisione

Anche per la divisione il linguaggio prevede istruzioni diverse a seconda che i fattori vengano interpretati come numeri naturali (senza segno) o come numeri relativi (con segno).

L'istruzione di divisione tra numeri naturali, che prevede sempre la dichiarazione implicita del primo operando, ha il seguente formato:

DIV *divisore*

ove *divisore* può essere il riferimento a un registro o a una variabile.

Il modo di operare del processore dipende, anche in questo caso, dalla dimensione dell'operando specificato. In particolare:

- se il divisore è di tipo byte (registro a 8 bit o riferimento BYTE TPR), viene effettuata la divisione tra il valore contenuto in AX e il divisore: il quoziente viene posto in AL e il resto in AH.

I flag vengono modificati dall'operazione, ma il loro valore finale è privo di ogni significato;

- se il divisore è di tipo word (registro a 16 bit o riferimento WORD TPR), viene effettuata la divisione tra il valore a 32 bit contenuto nella coppia di registri DX e AX ed il divisore: il quoziente viene posto in AX e il resto in DX. I flag vengono modificati dall'operazione, ma il loro valore finale è privo di ogni significato.

Per esempio, supponendo che:

```

AX = 0000000000000101
BL = 00000010

```

l'istruzione di divisione:

DIV BL

produrrebbe i seguenti effetti:

AL = 00000010
AH = 00000001

L'istruzione di divisione tra numeri relativi:

IDIV sorgente

opera in modo del tutto analogo alla precedente.

Incremento e decremento

Oltre alle quattro operazioni aritmetiche tradizionali, il linguaggio assembly prevede delle istruzioni particolari per velocizzare le operazioni di incremento e decremento unitario di registri e variabili.

Il loro formato è:

INC sorgente

DEC sorgente

ove *sorgente* può essere il riferimento a un registro o a una variabile.

LE STRUTTURE DI CONTROLLO

Come avete visto nelle lezioni di informatica, all'interno di un programma è possibile, attraverso le strutture fondamentali di selezione ed iterazione, modificare la naturale sequenza di esecuzione delle istruzioni in base al verificarsi o meno di determinate condizioni.

Ciò significa che l'esecutore, in base al verificarsi o meno della condizione testata deve riprendere l'esecuzione delle istruzioni da un punto diverso del programma, *saltando in avanti* (selezione) o *all'indietro* (iterazione).

Le istruzioni di salto

Questo tipo di operazione, a livello di linguaggio macchina, vuol dire *forzare* nel contatore di programma (registro IP) il riferimento alla posizione a cui si intende saltare per modificare la sequenza d'esecuzione.

L'istruzione che impone al processore di forzare in ogni caso un altro indirizzo nel program counter, istruzione detta *di salto incondizionato* è:

JMP etichetta di riferimento

Oltre a questa istruzione ne esistono molte altre che consentono di condizionare l'effettuazione del salto al verificarsi o meno di una certa condizione sui valori assunti dai flag. Logicamente, quindi, il salto condizionato risulta essere un'operazione del tipo:

```
se il valore dei flag soddisfa una certa condizione allora
    forza in IP l'indirizzo ....
altrimenti
    prosegui in sequenza
finesse
```

L'assembly dell'8086 prevede le seguenti istruzioni di salto:

Formato dell'istruzione	Descrizione
JA <i>etichetta di riferimento</i>	salta se CF=0 e ZF=0

JAE	<i>etichetta di riferimento</i>	salta se CF=0
JB	<i>etichetta di riferimento</i>	salta se CF=1
JBE	<i>etichetta di riferimento</i>	salta se CF=1 o ZF=0
JC	<i>etichetta di riferimento</i>	salta se CF=1
JCXZ	<i>etichetta di riferimento</i>	salta se CX=0
JE	<i>etichetta di riferimento</i>	salta se ZF=1
JG	<i>etichetta di riferimento</i>	salta se ZF=0 e SF=0
JGE	<i>etichetta di riferimento</i>	salta se SF=0
JL	<i>etichetta di riferimento</i>	salta se SF0
JLE	<i>etichetta di riferimento</i>	salta se ZF=1 o SF0
JNA	<i>etichetta di riferimento</i>	salta se CF=1 o ZF=1
JNAE	<i>etichetta di riferimento</i>	salta se CF=1
JNB	<i>etichetta di riferimento</i>	salta se CF=0
JNBE	<i>etichetta di riferimento</i>	salta se CF=0 e ZF=0
JNC	<i>etichetta di riferimento</i>	salta se CF=0
JNE	<i>etichetta di riferimento</i>	salta se ZF=0
JNG	<i>etichetta di riferimento</i>	salta se ZF=1 o SF0
JNGE	<i>etichetta di riferimento</i>	salta se SF0
JNL	<i>etichetta di riferimento</i>	salta se S=0
JNLE	<i>etichetta di riferimento</i>	salta se ZF=0 e SF=0
JNO	<i>etichetta di riferimento</i>	salta se OF=0
JNP	<i>etichetta di riferimento</i>	salta se PF=0
JNS	<i>etichetta di riferimento</i>	salta se SF=0
JNZ	<i>etichetta di riferimento</i>	salta se ZF=0
JO	<i>etichetta di riferimento</i>	salta se OF=1
JP	<i>etichetta di riferimento</i>	salta se PF=1
JPE	<i>etichetta di riferimento</i>	salta se PF=1
JPO	<i>etichetta di riferimento</i>	salta se PF=0
JS	<i>etichetta di riferimento</i>	salta se SF=1
JZ	<i>etichetta di riferimento</i>	salta se ZF=1

Le istruzioni di confronto

Dato che molto spesso capita di dover scegliere la sequenza di istruzioni da eseguire in base al risultato di un confronto tra due valori (per esempio, salta se. $A > 0$) i progettisti del microprocessore 8086 hanno pensato di inserire nel set di istruzioni un particolare tipo di operazione, detta *compare*, che permette di effettuare la sottrazione tra i due operandi, modificando i flag, *senza però memorizzare il risultato*.

Si tratta, in pratica di mettere a confronto aritmetico i due valori e registrare, *senza modificare i valori stessi*, l'esito del confronto a livello dei flag.

La forma sintattica dell'istruzione è:

CMP op1, op2

ove *op1* e *op2* devono avere la medesima dimensione e dove *op1* non può mai essere una costante.

Per esempio, l'istruzione:

CMP AX, BX

modifica i flag come SUB AX, BX ma il risultato della sottrazione non viene memorizzato in AX.

L'istruzione di confronto viene generalmente posta subito prima di un'istruzione di salto condizionato, in modo che questo venga effettuato in base al risultato del confronto stesso.

In questo caso, alcune tra le istruzioni di salto viste nel paragrafo precedente possono assumere un significato logico simile a quello usato nei linguaggi evoluti.

Formato dell'istruzione	Descrizione	
JA <i>indirizzo di riferimento</i>	salta se $op1 > op2$	<i>op1 e op2 interi senza segno</i>
JAE <i>indirizzo di riferimento</i>	salta se $op1 \geq op2$	<i>op1 e op2 interi senza segno</i>
JB <i>indirizzo di riferimento</i>	Salta se $op1 < op2$	<i>op1 e op2 interi senza segno</i>
JBE <i>indirizzo di riferimento</i>	Salta se $op1 \leq op2$	<i>op1 e op2 interi senza segno</i>
JE <i>indirizzo di riferimento</i>	Salta se $op1 = op2$	

Formato dell'istruzione	Descrizione	
JG <i>indirizzo di riferimento</i>	Salta se op1 > op2	<i>op1 e op2 interi con segno</i>
JGE <i>indirizzo di riferimento</i>	Salta se op1 >= op2	<i>op1 e op2 interi con segno</i>
JL <i>indirizzo di riferimento</i>	Salta se op1 < op2	<i>op1 e op2 interi con segno</i>
JLE <i>indirizzo di riferimento</i>	Salta se op1 <= op2	<i>op1 e op2 interi con segno</i>
JNA <i>indirizzo di riferimento</i>	Salta se op1 <= op2	<i>op1 e op2 interi senza segno</i>
JNAE <i>indirizzo di riferimento</i>	Salta se op1 < op2	<i>op1 e op2 interi senza segno</i>
JNB <i>indirizzo di riferimento</i>	Salta se op1 >= op2	<i>op1 e op2 interi senza segno</i>
JNBE <i>indirizzo di riferimento</i>	Salta se op1 > op2	<i>op1 e op2 interi senza segno</i>
JNE <i>indirizzo di riferimento</i>	Salta se op1 <> op2	
JNG <i>indirizzo di riferimento</i>	Salta se op1 <= op2	<i>op1 e op2 interi con segno</i>
JNGE <i>indirizzo di riferimento</i>	Salta se op1 < op2	<i>op1 e op2 interi con segno</i>
JNL <i>indirizzo di riferimento</i>	Salta se op1 >= op2	<i>op1 e op2 interi con segno</i>
JNLE <i>indirizzo di riferimento</i>	Salta se op1 > op2	<i>op1 e op2 interi con segno</i>
JNZ <i>indirizzo di riferimento</i>	Salta se op1 <> op2	
JZ <i>indirizzo di riferimento</i>	Salta se op1 = op2	

La struttura di selezione

In un linguaggio evoluto la *selezione* è una struttura che permette, in base al verificarsi o meno di una condizione, di scegliere tra due blocchi di istruzioni (istruzioni strutturate), di cui uno eventualmente vuoto.

La sua rappresentazione formale è:

```

se <Condizione> allora
    sequenza 1
altrimenti
    sequenza 2
fineSe

```

Facendo riferimento al modo di operare del processore questa struttura può essere interpretata nel modo seguente:

```

se <Condizione> allora
    esegui la sequenza 1
    salta la sequenza 2
altrimenti
    salta la sequenza 1
    esegui la sequenza 2
fineSe
.... prosecuzione programma

```

Utilizzando le istruzioni di salto presentate nei paragrafi precedenti è possibile allora realizzare la struttura di selezione nel modo seguente:

```

Blocco_altrimenti:      se <Condizione> salta al Blocco_allora
                       .... sequenza 2 ....
                       salta al fine_se
Blocco_allora:         .... sequenza 1 ....
fine_se:                .... prosecuzione programma

```

Traducendo in assembly questa struttura logica otteniamo:

```

; blocco di selezione
      J<Condizione> allora
altrimenti:  .... sequenza 2 ....
            JMP fineSe
allora:     .... sequenza 1 ....
fineSe:     ..... prosecuzione programma

```

Per esempio, la struttura:

```

se AX > 0 allora
    AX=0
altrimenti
    AX= FFFFh
fineSe
.....

```

viene trasformata, in assembly nel seguente blocco di programma:

```

    ; Blocco di selezione
    CMP AX, 0000h
    JG allora

    ; blocco altrimenti
    MOV AX, 0FFFFh
    JMP fineSe

    ; blocco allora
    allora: MOV AX, 0000h
fineSe: .....

```

Il ciclo a controllo in coda

L'iterazione è una struttura che permette di ripetere più volte l'esecuzione di un'istruzione strutturata sotto il controllo di una condizione.

Il ciclo a controllo in coda, nei linguaggi ad alto livello, è rappresentato da forme sintattiche del tipo:

```

ripeti
    sequenza
finché <Condizione>

```

che, attraverso il concetto di salto, può essere interpretato nel modo seguente:

```

ripeti
    .... sequenza ....
se non è vera <Condizione> salta a ripeti

```

La struttura assembly che ne deriva è:

```

ripeti: ... sequenza ....
    se non è vera <Condizione> salta a ripeti

```

Per esempio, la struttura:

```

A := 0;
ripeti
    A := A + B
finché A > B

```

diviene:

```

    MOV AX, 0000h
ripeti: ADD AX, BX
    CMP AX, BX
    JNG ripeti

```

Il ciclo a controllo in testa

La tipica struttura iterativa a controllo in testa è del tipo:

```

mentre <Condizione>
    sequenza
fineMentre

```

che, sfruttando il concetto di salto, può anche essere espressa come:

```

inizioMentre
    se non è vera Condizione salta a fineMentre

```

```

    .... sequenza....
    salta a inizioMentre
fineMentre

```

La struttura assembly corrispondente risulta essere:

```

mentre:      JN<Condizione> fineMentre
             .... sequenza ....
             JMP  mentre
fineMentre:  ..... prosecuzione programma

```

Per esempio, la struttura:

```

mentre (A < B)
    A := A + B
fineMentre

```

diviene:

```

mentre:      CMP AX, BX
             JNL fineMentre
             ADD AX, BX
             JMP  mentre
fineMentre:  .....

```

I cicli a contatore

La struttura tipica di un ciclo a contatore è:

```

ripeti per N volte
    sequenza
fineRipeti

```

che rende implicita l'operazione di conteggio.

Esplicitando questa operazione otteniamo:

```

inizializza un contatore a N
ripeti
    .... sequenza ....
    decrementa il contatore
finché CONTATORE vale 0

```

La struttura assembly corrispondente potrebbe essere:

```

MOV CX, n
ripeti: .... sequenza ....
DEC CX
JNZ ripeti

```

Il linguaggio prevede anche l'istruzione LOOP che rende implicita l'operazione di decremento sul registro CX, il cui significato può essere riassunto come segue:

- decrementa CX e salta se $CX \neq 0$

Sfruttando questa nuova istruzione, il ciclo a contatore assume la seguente forma:

```

MOV CX, n
ripeti: .... sequenza....
LOOP ripeti

```

Per esempio, la struttura:

```

ripeti per 10 volte
    A := A + B
fineRipeti

```

diviene:

```
MOV CX, 000Ah
ripeti: ADD AX, BX
        LOOP ripeti
```

Proposta di lavoro 6

Scrivere un programma che consenta di calcolare la quarta potenza di un numero naturale a 8 bit.

Risposta

Analisi del problema

Dato che un numero a 8 bit assume al massimo il valore 255, il valore massimo del risultato è $255^4 = 4228250625$ che, espresso in esadecimale, risulta essere FC05FC01, rappresentabile in un valore a 32 bit.