1 Introduzione all'Assembly

Programmare in Assembly è piuttosto difficile. Una volta che, avendo studiato la sintassi del linguaggio si riesce a compilare regolarmente un programma, si è ancora piuttosto lontani dalla metà del cammino. La parte più difficile infatti non è riuscire a rispettare le regole del linguaggio, ma far sì che il programma faccia effettivamente quello che deve fare. Quindi molto del tempo per lo sviluppo di un programma dovrà essere speso nella fase di test e, dato che con l'Assembly le sorprese non mancano mai, nel debugging.
Per acquisire buone competenze nella programmazione Assembly bisogna affinare le stesse doti che sono necessarie ad un buon programmatore in C.
Come con il C, per ottenere risultati si impongono volontà, disciplina e "altruismo".
La volontà serve per ottenere con lo studio la competenza sul linguaggio e per acquisire la capacità di individuare e togliere gli errori, destinando molte ore a cercare in tutti i modi di far funzionare i programmi che si realizzano.
La disciplina serve per non lasciarsi prendere la mano, una volta divenuti dei draghi nel linguaggio, a non scegliere soluzioni troppo complicate ed a non utilizzare trucchi non documentati.
Il requisito dell'altruismo serve per fare programmi leggibili. Per scrivere codice che rispetti la persona che lo legge bisogna sapersi mettere nei suoi panni e saper spiegare i meccanismi tramite i quali si è giunti a quella soluzione. Un buon programma in Assembly, come un buon programma in C o in qualsiasi altro linguaggio, risolve in modo efficiente il problema per cui è stato scritto ma ha anche la soluzione più semplice e la spiegazione più chiara.
Quello che l'Assembly dà in più, a chi ha la pazienza di praticarlo, è una conoscenza "intima" di come funziona un computer, che serve molto come bagaglio culturale, per questo viene insegnato nelle scuole, e che dà sicurezza in ciò che si fa. Inoltre l'Assembly può servire per realizzare i programmi più veloci che un programmatore possa scrivere.

Per ragioni pratiche in questo capitolo e nel seguito si farà sempre riferimento, negli esempi e nell'impostazione, all'8086, anche se verranno fatte alcune, rare, digressioni su altre CPU. La trattazione di altre CPU sarà presente solo per esempio e per confronto, senza alcuna pretesa di insegnare Assembly "stranieri".

Vantaggi dell'Assembly

- Sviluppo più rapido e migliore manutenibilità e di alcuni tipi di applicazioni

Per alcuni tipi molto particolari di applicazioni, in particolare quelle a diretto contatto con l'hardware e fortemente dipendenti dalle caratteristiche dei microprocessori, l'Assembly è il linguaggio più facile e veloce da usare.

- Velocità

In genere la velocità di esecuzione dei programmi scritti in Assembly è molto più alta di quella dei programmi scritti nei linguaggi di più alto livello. Questo era ed è vero, basti pensare ai videogiochi, un tipo di programma che ha bisogno di molta velocità. Un tempo i videogiochi erano sempre programmati in Assembly, almeno nelle parti più importanti[1] (*). Peraltro i videogiochi di oggi sono divenuti così complicati che sono scritte in Assembly solo parti, sempre più piccole, delle librerie che essi utilizzano, mentre la parte più ad "alto livello" del gioco è fatta in qualche linguaggio ad alto livello, spesso in C.

- Controllo

Con l'Assembly si può far fare alla CPU tutto quello che essa può fare, la si può spingere al massimo delle sue possibilità sapendo esattamente cosa si sta facendo. Con tutti gli altri linguaggi il dettaglio di come sono realizzate le istruzioni a basso livello è lasciato al compilatore, bisogna "fidarsi" del compilatore oppure, per ottenere lo stesso controllo, conoscere molto bene come esso lavora.

- Occupazione di memoria

I programmi scritti direttamente in Assembly tendono ad essere molto più piccoli di programmi analoghi scritti in linguaggi d'alto livello e compilati.

- Fattibilità di certe operazioni

Certe istruzioni di I/O o di manipolazione dei bit non sono possibili in molti linguaggi ad alto livello. Al contrario tutto ciò che una CPU può fare può essere fatto nel suo Assembly.

- Esistenza del solo Assembly

Certi microcontrollori, in particolare quelli di nuova produzione, possono disporre solo del loro Assembly, i compilatori ad alto livello possono non esistere od essere troppo costosi.

- Last, but not least: Didattica

Tutti i programmi vengono eseguiti in linguaggio macchina. Conoscere il linguaggio macchina, tramite l'Assembly, serve a farsi una cultura che può aiutare a scrivere programmi migliori anche con linguaggi di alto livello. Inoltre, dato che forse la difficoltà maggiore sta nel debugging, imparare a programmare in Assembly è una palestra per imparare a formulare ipotesi e teorie e metterle alla prova, cioè per imparare a pensare.

Svantaggi dell'Assembly

- E' difficile da programmare.

1 Negli ultimi tempi il vantaggio della maggiore velocità è di fatto venuto meno, almeno ella programmazione per PC. Ciò perchè le CPU più moderne che sono state pensate più per essere programmate con compilatori ottimizzanti che con l'Assembly.

Più che difficile è "diverso". Chi è abituato al Pascal non avrà problemi con il BASIC né troppe difficoltà con il C, che tutto sommato sono abbastanza simili, mentre ne avrà di più, per esempio, con il LISP od il Prolog che sono del tutto diversi. Parallelamente, chi sa usare l'Assembly 8086 non avrà grandi problemi con gli Assembly di altre CPU.
- E' difficile da leggere.
Come vedremo meglio in seguito, il flusso del programma può essere controllato solo con istruzioni di salto, per cui i programmi, costituiti di decine di salti anche lontani fra loro, divengono disordinati.
Per questo leggibilità di un programma Assembly è senz'altro bassa. Peraltro molto dipende dalla semplicità della soluzione e dalla buona educazione del programmatore che ha scritto il programma. Anche in C, se non si mantiene una stretta disciplina, è possibile scrivere programmi illeggibili.
- Non serve perché con macchine veloci e memoria poco costosa si possono scrivere programmi inefficienti.
Mentre ciò può essere parzialmente vero per certi tipi di programmi, c'è da rilevare però che nel corso della storia dell'Informatica la velocità delle CPU è sempre andata aumentando vertiginosamente, eppure le applicazioni sono sempre sembrate lente, ciò significa che scrivere programmi efficienti non andrà mai fuori moda.
- L'Assembly non è "portabile"
Questo è vero senza discussioni. Un programma scritto per un 8086 non potrà mai funzionare su un PowerPC o su un Alpha. C'è però da dire che più del 90% delle CPU esistenti nel mondo è della famiglia 80X86, oppure possono emularlo in modo efficiente, come il PowerPC.

Assembly

Per molti aspetti l'Assembly è un linguaggio di programmazione come gli altri; ciò che lo distingue è il fatto che ha una strettissima corrispondenza con il linguaggio macchina della CPU per cui è scritto.
Ogni codice operativo del set d'istruzioni di una CPU ha una corrispondente istruzione nel suo Assembly. Inoltre ogni istruzione Assembly può essere tradotta in una istruzione di macchina, anche se perché ciò sia vero è necessario dare una definizione di "istruzione" un po' restrittiva, come spiegheremo presto.
Almeno in linea di principio si può perciò affermare che fra Assembly e linguaggio macchina c'è una corrispondenza biunivoca (o 1 - 1) (vedi Figura 1: corrispondenza 1-1 fra istruzioni in Assembly e in linguaggio macchina).
La conseguenza principale della stretta commistione fra Assembly e linguaggio macchina è che ogni tipo di CPU ha il suo Assembly.
Le differenze fra i vari Assembly possono essere molto significative, ma il modo di programmare rimane simile. Chi impara a programmare con un Assembly non dovrebbe avere problemi ad iniziare a programmare con un altro. Successivamente, con il tempo e la pratica, potrà acquisire le competenze più specifiche per la nuova CPU.
Dato che l'Assembly è specifico di ogni famiglia di CPU esso è il tipo di linguaggio meno portabile.

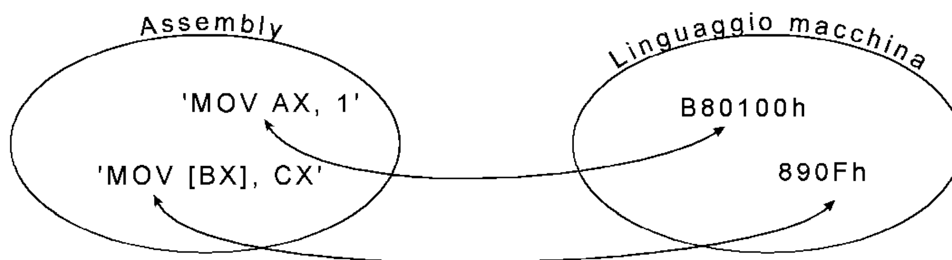


Figura 1: corrispondenza 1-1 fra istruzioni in Assembly e in linguaggio macchina

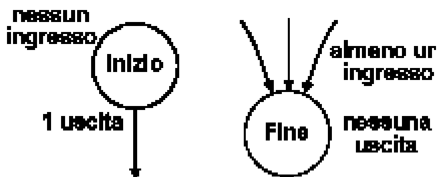
1.0.1 Diagrammi di flusso

Nella programmazione Assembly è importante usare tecniche di documentazione che aiutino anche nella fase di analisi e progettazione dell'algoritmo. La più tipica di queste tecniche è quella dei diagrammi di flusso, che in Assembly sono particolarmente utili perché è molto più difficile che con gli altri linguaggi capire a prima vista il funzionamento dei programmi.
Il **diagramma di flusso** è uno schema che ha il compito di illustrare graficamente il funzionamento di un algoritmo, evidenziando la sequenza dei passi che si devono compiere nel suo svolgimento.
Esso comprende alcuni tipi di blocchi, entro i quali vengono scritte frasi o espressioni che descrivono l'algoritmo. I blocchi sono collegati da frecce.

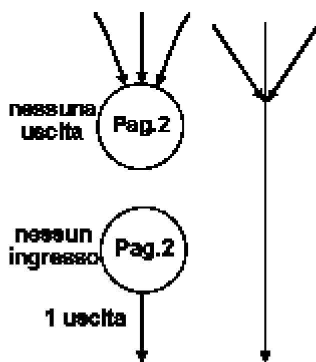
Frecce
Le frecce di un diagramma di flusso illustrano l'ordine di esecuzione dei blocchi.
Proprio perché devono spiegare l'ordine di esecuzione esse non possono essere archi o segmenti ma devono essere vere e proprie frecce orientate, con una coda che indica ciò che avviene prima e una punta che porta a ciò che avviene dopo.

Blocchi di inizio e fine
Ogni algoritmo deve avere un punto di inizio, dal quale parta il suo flusso. Nel diagramma indicheremo con un cerchio il punto di inizio. All'interno del cerchio scriveremo "Inizio", oppure "I".

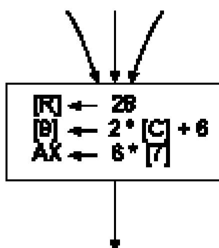
In ogni istante il flusso di un programma può essere in un solo punto, per cui da un blocco di inizio può uscire una sola freccia. Dato che il blocco di inizio è il primo al suo interno non può entrare nessuna freccia.
Ogni algoritmo deve giungere alla sua conclusione in un numero finito di passi, per cui dovrà essere presente un blocco che indica che il procedimento è terminato. Per questo scopo useremo un blocco di fine, che disegneremo con un cerchio, scrivendo all'interno "Fine" o "F".
Dato che l'algoritmo può terminare in ogni momento il suo flusso può saltare da ogni punto ad un blocco di fine, per cui in un blocco di fine possono entrare più di una freccia. I blocchi di fine presenti nel diagramma possono essere più di uno, ma ce ne deve essere almeno uno. Naturalmente da un blocco di fine non dovrà uscire nessuna freccia.



Rimandi
Può accadere che un diagramma di flusso non si possa disegnare tutto nella stessa facciata di un foglio. In questo caso è necessario ricorrere a dei rimandi, che indicheremo con un cerchio che racchiude un simbolo univoco, che viene ripreso anche da un altro blocco di rimando. Il flusso che entra in un blocco di rimando esce dall'altro blocco che porta lo stesso simbolo, per cui due blocchi di rimando che contengano lo stesso simbolo sostituiscono una freccia, come indicato nella seguente figura:



Blocchi di assegnazione
Rappresentiamo i blocchi di assegnazione come rettangoli. All'interno dei blocchi di assegnazione scriviamo una lista di espressioni che vengono eseguite nell'ordine dall'alto verso il basso.
Le espressioni indicano che un elemento di memoria (registro o locazione) viene modificato.
Le espressioni sono composte di due parti, separate da una freccia. Dalla parte puntata dalla freccia si scrive l'elemento di memoria che deve essere modificato, dall'altra parte una espressione aritmetica, il cui valore viene calcolato e copiato nell'elemento di memoria indicato.
Indichiamole locazioni di memoria fra parentesi quadre ed i registri con il loro nome (p.es. [10] o AX).
Diamo anche nomi simbolici alle locazioni di memoria, in modo da non doverne specificare l'indirizzo e da poter utilizzare un nome che spieghi a cosa serve quella locazione.
Nel seguente esempio



Le espressioni significano:
- scrivi il numero 28 nella locazione di memoria che chiamo R
- scrivi nella locazione di indirizzo 9 il doppio del valore della locazione che ha nome C, cui aggiungi 6

- scrivi nel registro AX il valore contenuto all'indirizzo 7 moltiplicato per 6

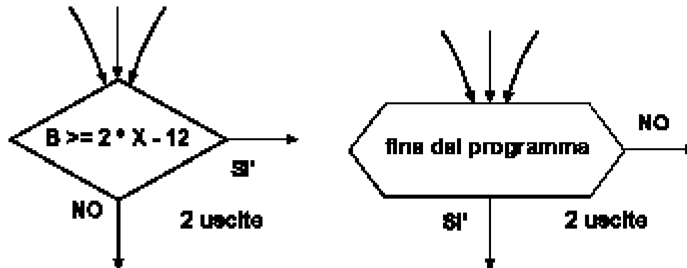
Gli elementi di memoria modificati vengono sovrascritti. Ciò significa che i valori vecchi di quegli elementi di memoria sono perduti.

Blocchi condizionali

I blocchi condizionali, o di controllo, sono gli unici che hanno due frecce in uscita. Infatti in questi blocchi il flusso dell'algoritmo si dirama, in base al verificarsi o meno di una condizione logica.

La condizione deve essere specificata all'interno del blocco stesso. Potremo esprimerla in forma generica, come una domanda cui si possa rispondere solo sì o no, o come una vera e propria espressione logica.

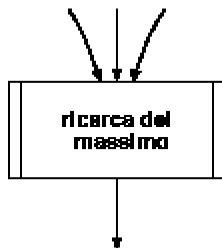
Per comodità nel disegno useremo spesso in luogo del rombo un esagono irregolare, che ha il pregio di usare meno area nel foglio, quando le condizioni sono lunghe.



Blocchi non specificati

Molto spesso i diagrammi di flusso non si disegnano per specificare fino in fondo il funzionamento dell'algoritmo, ma piuttosto per dare un'indicazione generale su come funziona, lasciando il dettaglio al codice in linguaggio di programmazione.

Distingueremo i blocchi "generali" da quelli di assegnazione disegnando due barre sui lati destro e sinistro.



Una frase scritta entro il blocco indicherà la sua funzione.

Un blocco generale rappresenta una serie di istruzioni non specificate, che possono anche essere un intero "sottoprogramma". Per particolari tipi di sottoprogrammi, detti procedure, introdurremo un diverso tipo di blocco.

1.0.2 Istruzioni e pseudoistruzioni

Come tutti i linguaggi di programmazione anche l'Assembly ha le sue parole chiave, che vengono "capite" dal compilatore. Esso risponde a parole chiave diverse facendo cose diverse.

Dato il rapporto diretto che c'è fra l'Assembly ed il linguaggio macchina è possibile fare una distinzione, che negli altri linguaggi non è applicabile, fra i diversi "comandi" dell'Assembly.

Chiamiamo "**istruzioni**" quelle parole chiave del linguaggio Assembly che implicano l'aggiunta di un codice operativo di macchina al file .OBJ. Quindi un'istruzione si traduce in un codice di macchina che viene preso dalla memoria a run-time, in fase di fetch, per essere successivamente decodificato ed eseguito dalla CU.

Chiamiamo "**pseudoistruzioni**", o anche "**direttive**", quelle parole chiave del linguaggio Assembly che sono solamente comandi per il compilatore. Il loro risultato modificherà "globalmente" il file .OBJ, perché in conseguenza di una direttiva il compilatore si comporta in modo diverso. Peraltro una pseudoistruzione non aggiungerà codici operativi all'OBJ ed il suo effetto sarà finito con la fine della compilazione.

Dunque le istruzioni Assembly corrispondono ai codici mnemonici delle istruzioni di macchina di quel tipo di CPU e determinano il comportamento della CPU, mentre le pseudoistruzioni sono le altre parole chiave di quell'Assembly ed influenzano il comportamento del compilatore. In sintesi: un'istruzione è un ordine dato alla CPU, mentre una direttiva è un ordine dato al compilatore.

Come esempio di un'istruzione Assembly prendiamo una MOV, la più comune istruzione 8086:

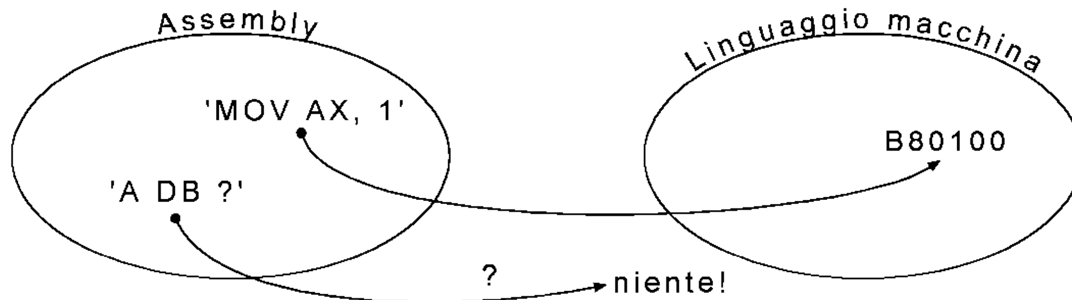
```
MOV AX, 1
```

questa istruzione aggiunge al codice oggetto i numeri B8 01 00 (esadecimali)

Al contrario la pseudoistruzione:

```
MiaLocazione DB (?)
```

Farà sì che il compilatore riservi un byte di spazio in memoria che nel seguito del programma sarà chiamato "MiaLocazione". Il file .OBJ conterrà un byte in più, destinato a "MiaLocazione", ma non verrà aggiunto nessun codice operativo, dato che "MiaLocazione" farà parte di un'area di dati e non di codice.



	Effetto	Fine dell'effetto	Corrispondenza con linguaggio Macchina
Istruzioni	Aggiunta di codici operativi all'OBJ	Run time	Sì
Direttive	Modifiche globali all'OBJ	Compile time	No

1.0.3 Formato delle istruzioni Assembly

Il formato delle istruzioni può essere diverso da Assembly a Assembly. Vediamo un formato molto utilizzato, che è quello che si usa anche nell'8086, in particolare per quel che riguarda l'ordine di destinazione e sorgente (*).

Formato di una pseudoistruzione Assembly (**):

```
[<Etichetta>]  <DirettivaAssembly> [; <Commento>]
```

Formato di una istruzione Assembly:

```
[<Etichetta>:] <CodiceMnemonico>  [<Destinazione>] [, <Sorgente>] _
                [; <Commento>]
```

Nell'Assembly dell'8086 ogni singola istruzione deve stare in un'unica linea del file sorgente. Questa non è una limitazione significativa, perché le istruzioni Assembly sono sempre piuttosto corte.

(*) In altri Assembly, come per esempio per le CPU 68000, l'ordine di destinazione e sorgente è invertito
(**) In questo paragrafo si introduce per la prima volta la notazione usata in tutto il resto del libro per presentare la sintassi delle istruzioni dei linguaggi di programmazione. Nella legenda all'inizio del volume si può trovare una spiegazione delle convenzioni usate.

<Etichetta> (**label**) è un nome qualunque, definito dall'utente e diverso da tutte le parole chiave del linguaggio. E' sempre un riferimento ad un indirizzo. Sulla cosa torneremo più avanti ma dev'essere ben chiaro fin da ora che ogni nome definito dall'utente, come p.es. quel "MiaLocazione" visto qualche riga più sopra, è sempre e solo un modo comodo per indicare un indirizzo della memoria principale. Il nome di una etichetta può essere soggetto a restrizioni, la più tipica delle quali è che debba necessariamente iniziare con una lettera. Alcuni Assembler hanno limitazioni significative sul numero dei caratteri delle etichette.
Come indicato dalla scrittura fra parentesi quadre, nelle linee di codice Assembly <Etichetta> non è obbligatorio: si mette solo se serve.

<DirettivaAssembly> è un comando rivolto al compilatore, vedremo molti esempi per l'Assembly 8086.

<CodiceMnemonico> è un simbolo alfanumerico che fa parte delle parole chiave del linguaggio e che indica quale istruzione di macchina il compilatore deve mettere nel programma oggetto.

Ogni codice mnemonico dell'Assembly di una CPU ha corrispondenza diretta con un codice operativo del linguaggio di macchina di quella CPU. Per il compilatore è semplice sostituire al codice mnemonico il relativo codice operativo.

<Destinazione> e <Sorgente> sono gli operandi dell'istruzione, cioè i numeri che l'istruzione deve elaborare. Sono separati da una virgola.

<Destinazione> è il registro o la locazione di memoria da dove viene preso uno degli operandi dell'istruzione e dove finisce il risultato. Se si escludono alcune rare eccezioni l'esecuzione dell'istruzione modifica il contenuto dell'operando destinazione, sovrascrivendolo. Ciò significa che il valore che aveva l'operando destinazione prima dell'esecuzione dell'istruzione è perduto per sempre. Se non si vuole "dimenticarlo" è necessario mantenerlo da qualche parte, per esempio in un altro registro. La destinazione è "opzionale" nel senso che alcune istruzioni, come per esempio la NOP dell'8086, non hanno alcun operando.

<Sorgente> è un operando dell'istruzione. Può essere un registro, una locazione di memoria od un numero fisso. L'operando sorgente non viene mai modificato dall'istruzione.

Alcune istruzioni possono avere un solo operando. In questo caso <Sorgente> o <Destinazione> manca. Per esempio, nell'8086, si può prendere la MUL che ha solo l'operando sorgente, mentre la destinazione è sempre quella, implicita, e non viene mai indicata.

<Commento> ogni istruzione Assembly ammette dei commenti, cioè caratteri che non sono analizzati dal compilatore. In Assembly 8086 il simbolo che indica un commento è ";" (punto e virgola). Quando l'Assembler incontra un punto e virgola ignora tutti i caratteri successivi sulla stessa linea e riprende ad analizzare il sorgente solo dalla linea successiva. Non sono previsti commenti che coprano più di una linea, nei commenti voluminosi ogni linea dovrà cominciare con punto e virgola.

Le istruzioni dell'8086 hanno un numero di operandi che va da 0 a 2. Alcune CPU più moderne hanno istruzioni a tre operandi, nelle quali si possono specificare due sorgenti ed una destinazione diversi.

Nella seguente figura sono indicati alcuni esempi di direttive ed istruzioni, che verranno illustrate in dettaglio nel seguito:

```
Vettore DW 1000 DUP (?) ; "Prenota" memoria per 1000 word

SaltaQui: ADD [25], 7   ; somma 7 al contenuto della locazione 25

 NOP ; non fa nulla              istruzione   direttiva   etichetta

nValori EQU 60 ; dichiara che "nValori" deve sempre
               ; essere sostituito con 60

Data SEGMENT ; segna l'inizio di un'area di memoria
```

Figura 2: esempi di direttive ed istruzioni

1.0.4 Formato dei programmi Assembly

I linguaggi di alto livello stabiliscono convenzioni da seguire per stabilire dove scrivere nel programma sorgente le dichiarazioni e dove le istruzioni di codice. Per esempio il Pascal rende obbligatorio scrivere tutte le dichiarazioni prima delle parti di codice.

In Assembly non si è vincolati ad alcun formato, istruzioni e dati possono essere mescolati a piacimento, a patto che tutto funzioni!

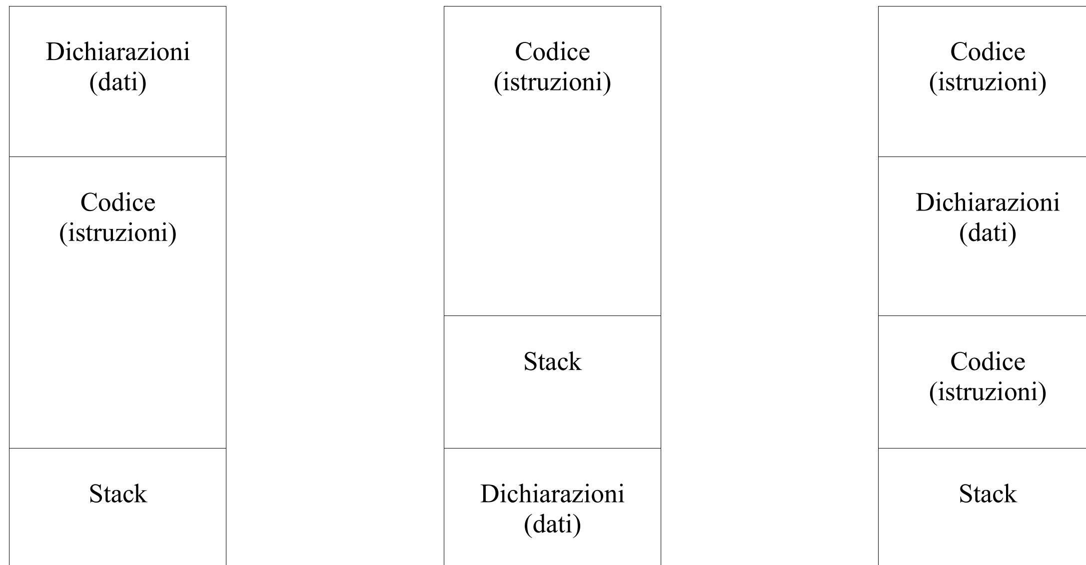


Figura 3: tre esempi mostrano che non c'è un ordine obbligatorio per istruzioni e dati.

E' comunque sempre una buona idea darsi uno standard ed usare sempre le stesso.
In casi particolari, alcuni dei quali vedremo molto più avanti, è importante l'ordine con cui istruzioni e dati vengono allocati in memoria, per cui non è la stessa cosa scrivere in un ordine od in un altro.

Infatti l'ordine di scrittura corrisponde all'ordine con il quale il programma verrà compilato e caricato in memoria.

Se si scrive prima il codice, esso verrà sistemato agli indirizzi più bassi, se si scrivono prima i dati saranno loro ad avere gli indirizzi più bassi.

Si tenga però presente che a quest'affermazione si può fare eccezione usando particolari direttive dell'Assembly, che permettono di stabilire in quale ordine ed a quali posizioni di memoria l'Assembler deve allocare le parti di programma.
In questo libro faremo solo un breve cenno, più avanti, a direttive di questo tipo.

1.0.5 Gli Assembler per 80X86

In genere i produttori delle CPU vendono anche un Assembler, ed un compilatore C, come supporto allo sviluppo di programmi per il loro hardware. In genere la sintassi dell'Assembly viene stabilita dal produttore della CPU, che conosce bene le caratteristiche del microprocessore e le sue possibilità.
Per le CPU che acquisiscono un mercato considerevole anche i produttori di strumenti di sviluppo software entrano in gioco, realizzando compilatori per i vari linguaggi.
Per quanto riguarda le CPU X86 gli Assembler più utilizzati sono MASM (Microsoft Assembler) e TASM (Turbo Assembler, della Borland), che hanno caratteristiche molto simili., tanto che il codice sorgente scritto per un compilatore può essere compilato con l'altro senza modifiche.
In altri ambienti di sviluppo l'Assembly può essere molto diverso, anche per la stessa famiglia di CPU. Per esempio l'Assembler GNU per X86, utilizzato per compilare i programmi in Linux, ha una sintassi completamente diversa da MASM e TASM, e definisce un formato che agevola "portabilità" su diverse famiglie di CPU.
Esiste anche un Assembler gratuito, NASM (**N**etwide **As**se**m**bler), realizzato "su Internet" da un team internazionale. NASM può compilare file oggetto per DOS; Windows e Linux, è molto simile a MASM-TASM, ma non è del tutto compatibile. La sua sintassi impedisce alcune ambiguità presenti in MASM-TASM e può richiedere un certo lavoro sul codice sorgente (vedi http://www.cryogen.com/Nasm).
Infine è presente in rete il MASM32, strumento basato sul compilatore MASM a 32 bit di Microsoft, che è disponibile gratuitamente e che è stato "rivestito " di codice e librerie da un team di volontari in Rete. Compilatore, librerie e documentazione sono disponibili on line a: XXXX (!!!!aggiungere)
In questo testo la trattazione della sintassi delle istruzioni Assembly è incentrata su TASM-MASM per la programmazione a 16 bit in MS-DOS, e sul MASM32 per la programmazione a 32 bit in Windows.