|  |  |  |
| --- | --- | --- |
| Altomare Andrea | Lab. Sistemi esperienza n. 9 | Classe 3^A 4/02/2015 |

**Assembly 8086: cicli, vettori e salti**

**PROBLEMA**

Dato un vettore di dieci elementi noti, memorizzarne il valore maggiore in una variabile.

Vettore: *v db 10 , 5 , 31 , 40 , 18 , 22 , 27 , 3 , 9 , 15*

Variabile in cui verrà memorizzato il valore massimo del vettore: *max db 0* (valore iniziale a 0).

**LOOP**

Per riprodurre un ciclo (**for**, nei linguaggi ad alto livello) si usa l’istruzione **loop**. Questo comando permette, dopo aver inserito un’etichetta, di eseguire la parte di codice seguente a quest’ultima per un numero definito di volte. L’istruzione va ad agire implicitamente sul registro cx usandolo come contatore; per scrivere correttamente un ciclo con il loop bisogna quindi assegnare a cx un numero corrispondente a quante volte si vuole eseguire il ciclo. Eseguita la parte di codice, arrivati al loop, esso decrementa di 1 il valore di cx, ripetendo il ciclo finché cx non diventa 0 (a quel punto il ciclo si conclude).

**SPOSTARSI IN UN VETTORE**

Per spostarsi in un vettore bisogna tenere conto del tipo di vettore. Innanzi tutto va chiarito che esistono due registri adibiti al contenimento dell’indirizzo dell’indice del vettore, per cui, in genere, si utilizzano quelli per spostarsi in un vettore. Questi registri sono **SI** e **DI**, e possono essere utilizzati indifferentemente. Per muoversi quindi nel vettore, si inizia (per esempio) dalla posizione 0 che va memorizzata in uno dei due registri indice (per esempio DI), il valore del registro va poi incrementato di 1 o di 2 a seconda se il vettore è di tipo Byte o di tipo Word rispettivamente (dato che Word è il doppio di Byte). Questa tecnica fa quindi uso dei **registri indice**.

Un’altra tecnica utilizzata per spostarsi in un vettore è usando come indice direttamente l’indirizzo del vettore in questione. Per compiere questo tipo di tecnica bisogna spostare in un registro (che fungerà da indice) l’indirizzo della cella di memoria da cui inizia il vettore;

il comando è **MOV <registro>,OFFSET <vettore>** (per esempio: *MOV BX, OFFSET V*). Con la parola chiave **OFFSET** è quindi possibile prendere in considerazione non il valore contenuto nella variabile indicata ma l’indirizzo della cella di memoria in cui è ubicato. Per prendere il valore successivo del vettore, si incrementa il registro usato per contenere l’offset di 1 se l’array è di tipo Byte o di 2 se è di tipo Word.

Gli incrementi sono possibili mediante l’istruzione *ADD <registro>,1* oppure usando l’operatore **INC** che incrementa di 1 l’operando indicato secondo la sintassi **INC <operando>**.

**NB:** esiste anche l’operatore **DEC** che, al contrario di INC, decrementa l’operando indicato di 1 secondo la sintassi **DEC <operando>**.

**SALTI**

Nel linguaggio Assembly l’esecuzione delle istruzioni scorre in sequenza incondizionatamente. Per riprodurre una selezione come un **IF** o un ciclo iterativo come un **WHILE** o un **DO WHILE**, o semplicemente per saltare incondizionatamente ad un’altra istruzione che non è in sequenza, bisogna quindi compiere dei salti *incondizionati* o *condizionati*, possibili con l’operatore **JUMP** (diverso a seconda delle esigenze e dell’uso che se ne deve fare). Un salto incondizionato si compie con l’istruzione **JMP <etichetta>**. Le etichette si mettono all’interno del programma per indicare un’istruzione o una serie di istruzioni a cui si vuol fare riferimento (quando, appunto, si salta, o per altri motivi di progettazione del programma stesso); un’etichetta va inserita nella forma **<nome\_etichetta>:** (ad esempio CICLO:). Utilizzando l’istruzione JMP si salta dunque incondizionatamente all’istruzione subito seguente all’etichetta indicata. I salti incondizionati si compiono invece in due passaggi: bisogna prima effettuare una comparazione tra due operandi, poi, a seconda dell’esito del confronto, saltare con un JUMP apposito ad una istruzione desiderata. I confronti si eseguono con l’istruzione **CMP <operando1>,<operando2>**; questo comando (*Compare*) in realtà compie una sottrazione tra il primo e il secondo operando e, a seconda del risultato, verranno automaticamente modificati dei valori nel registro dei **FLAG** per esempio in **CF**, **ZF**, ecc… Per esempio, se il registro ZF (*Zero Flag*) è 1, significa che la sottrazione (e quindi la comparazione) tra i due operandi, ha prodotto un risultato uguale a zero, per cui sono uguali. La seconda fase per compiere i salti condizionati consiste in seguito nel vero e proprio salto: si usa sempre l’istruzione JUMP, ma a seconda della condizione per cui si vuole saltare, si sceglie un JUMP diverso. Se, per esempio, voglio saltare solo se i valori contenuti nei registri AX e BX sono uguali, dopo aver scritto CMP AX,BX dovrò scrivere **JE** (o **JZ**) **<etichetta>**, se i valori di AX e BX sono uguali, il programma salterà quindi all’etichetta indicata, altrimenti procederà con la normale esecuzione in sequenza delle successive istruzioni nel programma.

Esempio completo:

* CMP AX,BX ; Confronta i valori di AX e BX “settando” ad 1 determinati valori del registro dei FLAGS
* JE ciclo ; Se i valori di AX e BX sono uguali (quindi il valore di ZF è 1) salta all’istruzione seguente all’etichetta “ciclo”, altrimenti procedi con l’esecuzione in sequenza del programma.

**NB:** Se si prende un’istruzione di salto condizionata e si mette una **N** (*Not*) prima delle condizioni (come *Equal*, *Greater*, ecc…) si crea un’istruzione JUMP che salta se la condizione specificata è **FALSA**. Esempio: **JNE** (*JUMP NOT EQUAL*) salta se i due operandi NON sono uguali. Ogni istruzione di salto condizionata ne ha quindi una rispettiva “*contraria*”.

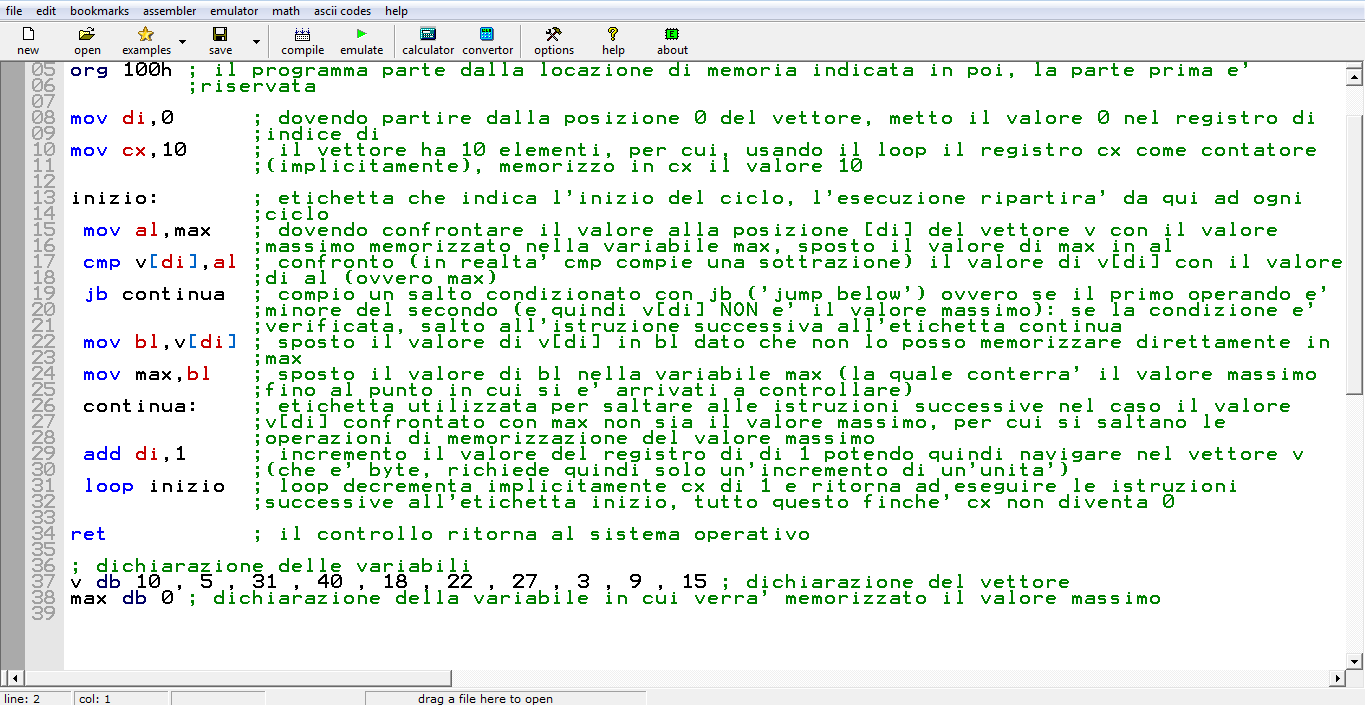
Altre principali istruzioni di salto condizionato sono:

* JE = salta se gli operandi sono *uguali*
* JG = salta se il primo operando è *maggiore* del secondo (considerando il segno)
* JL = salta se il primo operando è *minore* del secondo (considerando il segno)
* JA = salta se il primo operando è *maggiore* del secondo (NON considerando il segno)
* JB = salta se il primo operando è *minore* del secondo (NON considerando il segno)
* JGE = salta se il primo operando è *maggiore* o *uguale* del secondo (considerando il segno)
* JLE = salta se il primo operando è *minore* o *uguale* del secondo (considerando il segno)

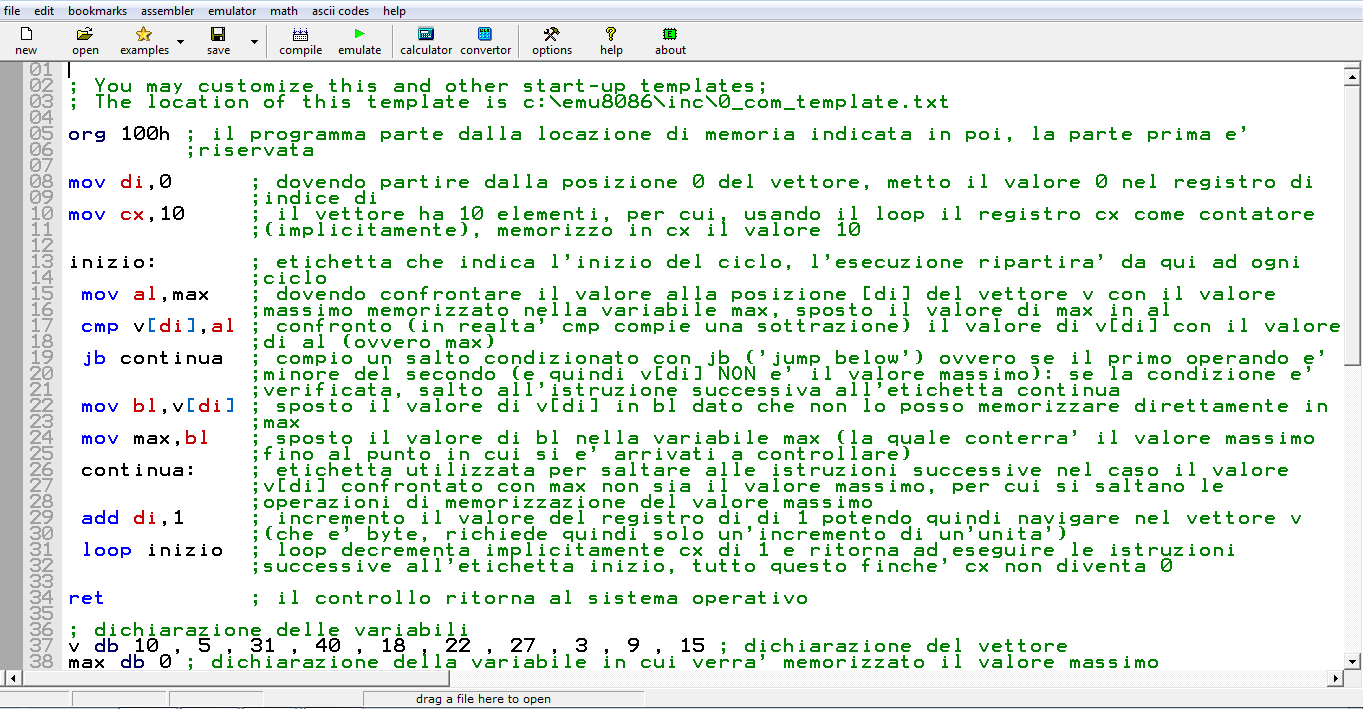
**REALIZZAZIONE DEL PROGRAMMA**

Per trovare il valore massimo di un vettore mi sono servito di un ciclo: controllo ciclicamente tutti i valori del vettore, e quando ne trovo uno maggiore rispetto a quello memorizzato nella variabile max (che conteneva il numero più grande tra tutti quelli controllati prima) lo memorizzo proprio in max, finché la variabile assuma il valore maggiore.

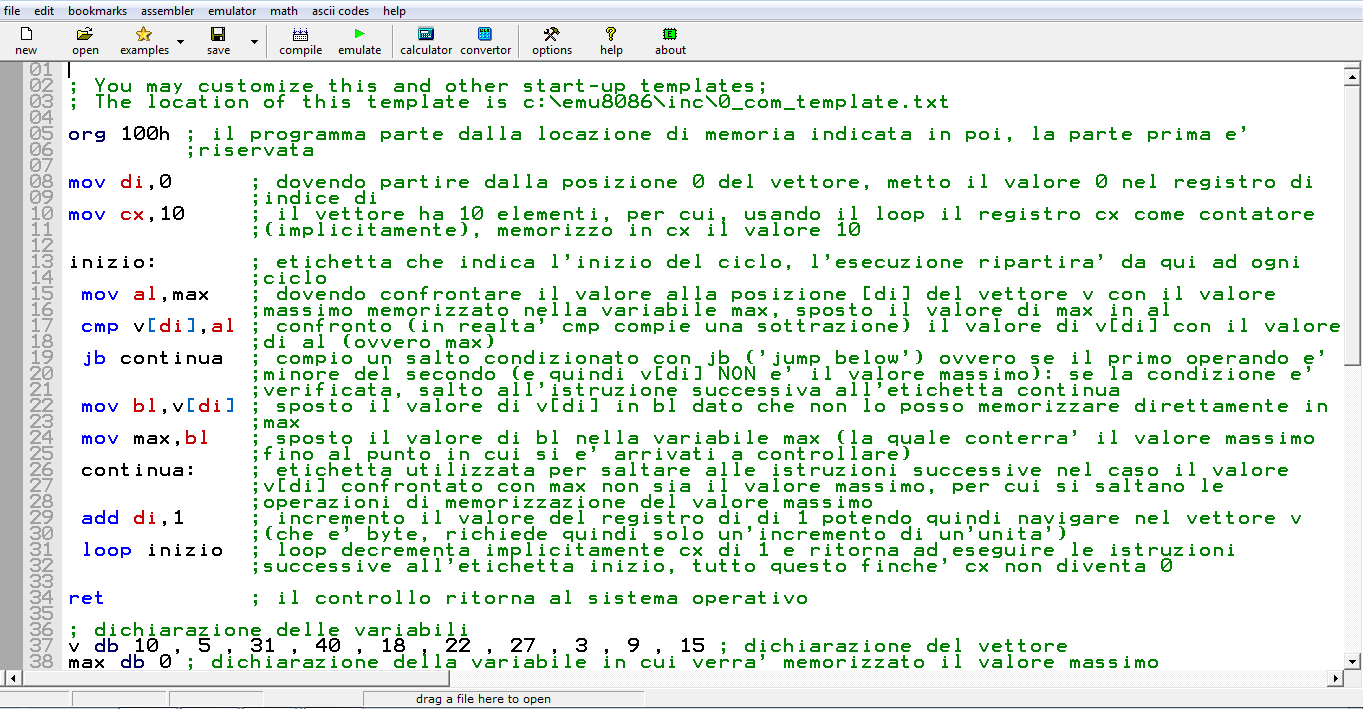
La dichiarazione delle variabili è stata effettuata come nell’immagine: i valori di un vettore si separano semplicemente ponendo una virgola tra loro, ma il tipo può essere comunque o DB o DW. In questo caso il vettore è di Byte (come la variabile max).



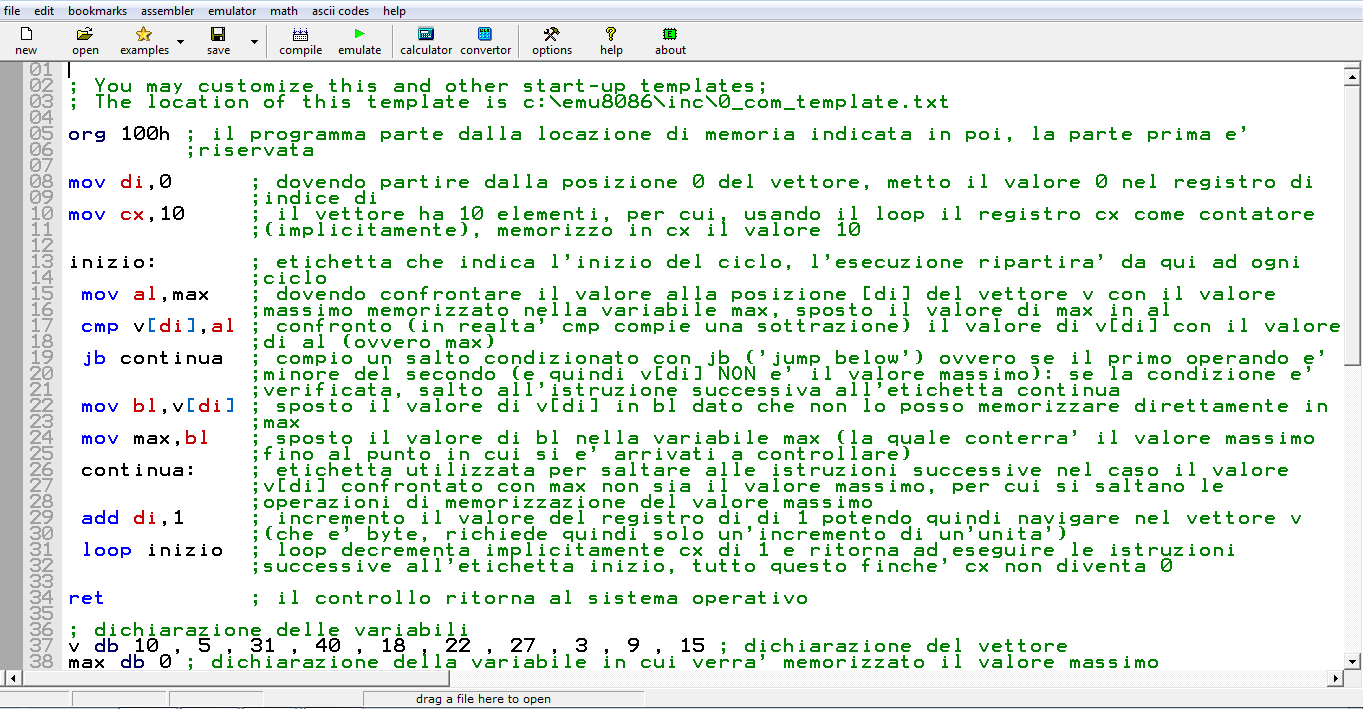
La prima stringa di codice in assoluto da scrivere in un programma Assembly, è *org 100h*, effettuato ciò, si passa al programma vero e proprio. Per effettuare il controllo su tutti i valori di un vettore ho bisogno di un ciclo, per sposto nel registro contatore CX il valore che rappresenta il numero di elementi presenti nel vettore (quindi da controllare ciclicamente), sposto quindi 10 su CX. Poi, sposto 0 sul registro di indice DI dato che devo partire dal primo valore del vettore il quale si trova in posizione 0.



Per distinguere l’inizio del ciclo, metto un etichetta “*inizio:*”. Sposto il valore della variabile max in al e poi confronto al con v[di] (dato che non si possono compiere comparazioni tra due variabili, quindi bisogna servirsi di un registro). Se v[di] è minore di al (quindi di max) non ne memorizzo il valore in max, per cui salto con un’istruzione di salto adeguata alla porzione di codice indicata dall’etichetta “*continua*”. Uso un JUMP che non tiene conto del segno dato che i valori del vettore sono tutti positivi.

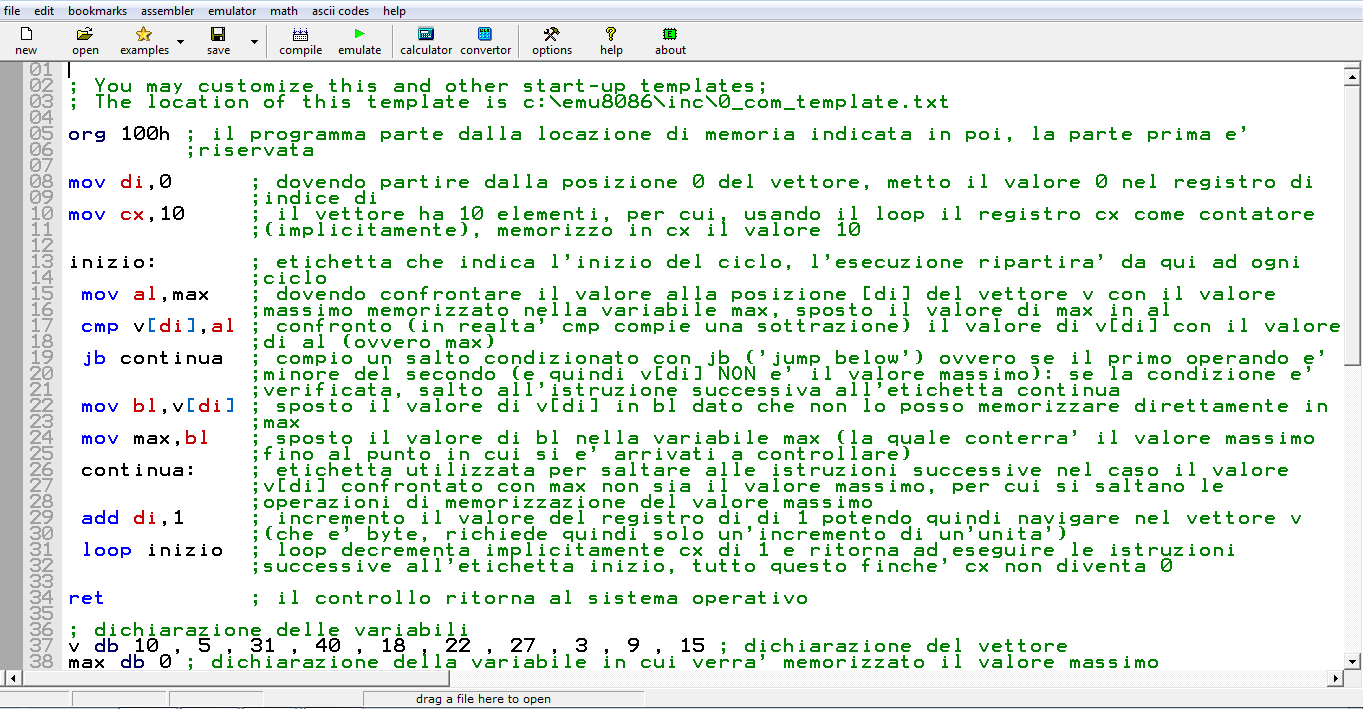


Se la condizione che v[di] sia minore di al (quindi di max) non è verificata, vuol dire che v[di] è maggiore, per cui non si salta a continua ma si esegue il codice immediatamente sottostante, ovvero quello che permette di memorizzare nella variabile max il valore di v[di]. Per prima cosa sposto in bl il valore di v[di], poi sposto il valore di bl in max.



Dopo queste istruzioni vi è appunto l’etichetta “*continua:*” e le istruzioni che permettono l’incremento di DI (per spostarsi nel vettore) ed il ritorno ciclico ad “*inizio*” per rieseguire il codice finché CX non sarà 0 (per cui si saranno controllati tutti i valori).

Infine c’è l’istruzione che ritorna il controllo al sistema operativo.



Ho infine verificato il corretto funzionamento del programma controllando che, alla fine dell’esecuzione, la variabile max contenesse il valore massimo del vettore *v*, ovvero 40.

