|  |  |  |
| --- | --- | --- |
| Altomare Andrea | Lab. Sistemi esperienza n. 8 | Classe 3^A 28/01/2015 |

**Assembly 8086**

L’Assembly è il linguaggio di programmazione più vicino alla macchina dopo il linguaggio macchina (costituito solo da 0 e 1): mentre per un linguaggio ad alto livello come il C o il Pascal, operazioni come *a=b+c* permettono di eseguire una semplice addizione e memorizzarne il risultato in una variabile, in Assembly bisogna tener conto di tutto (o quasi) ciò che il processore compie per eseguire quella determinata operazione. Per programmare in Assembly è quindi necessario conoscere i fondamenti che stanno dietro al funzionamento del microprocessore e, in generale, dell’Unità di Elaborazione. In breve: il **processore** (o **microprocesore**, o **CPU** [*Central Processing Unit*]) è un automa costituito da vari elementi che cooperano per il raggiungimento di un certo obiettivo (come l’esecuzione di una certa istruzione). I componenti costituenti principali sono i **registri** (celle di memoria ad altissima velocità [quella del clock] su cui vengono temporaneamente spostati i dati da elaborare), e la **A.L.U.** (*Arithmetic-Logic Unit* o *Unità Aritmetico-Logica*) che effettua operazioni aritmetiche e logiche (algebra booleana) su dei dati. L’**Unità di Elaborazione** (la quale contiene a sua volta il processore) è costituita inoltre dalla memoria **Cache** (chiamata in inglese “*Cache memory*”, è una memoria piccola e ad altissima velocità che viene usata dalla CPU poiché la RAM è troppo lenta), che può essere anche integrata col processore in taluni casi, e dalla **memoria centrale** (la RAM ad esempio) in cui sono caricati il sistema operativo, le istruzioni del processo in esecuzione e i dati da elaborare. Tutti questi elementi sono connessi da un sistema di **Bus** che permette la comunicazione intercomponente, ne esistono tre tipi: Bus indirizzi (nel quale viaggiano gli indirizzi delle celle di memoria da cui andare ad estrarre un dato o un’istruzione), Bus dati (su cui viaggiano i dati, o le istruzioni che poi verranno memorizzati nei registri interni al processore), e Bus di controllo (su cui viaggiano i segnali di controllo generati dall’Unità di controllo interna al microprocessore, come i comandi *Read* (R) per impostare una cella di memoria in modalità lettura per poi andare a leggere [quindi a prendere] il dato, oppure il comando *Write* (W) per impostare una cella di memoria in modalità scrittura per poi andare a scrivere [quindi a memorizzare] un altro dato in quella determinata cella di memoria. Va notato che, solitamente, la dimensione dei registri corrisponde a quella dei Bus (per esempio, se i Bus sono a 32 bit allora anche i registri saranno a 32 bit). Si nota inoltre che parti fondamentali del micro processore sono l’**Unità di controllo del bus** (che ha il compito di stabilire la comunicazione tra il processore e le unità esterne), l’**Unità di istruzione** (la quale estrae e decodifica le successive istruzioni del programma in esecuzione), l’Unità di indirizzi (specializzata nel calcolo degli indirizzi), e l’**Unità di controllo** vera e propria (che riceve gli input dall’Unità di istruzione e genera i segnali di controllo e temporizzazione necessari all’esecuzione dell’istruzione in fase di execute).

I registri si dividono in due tipi:

* Generali: visibili al programmatore e sui quali si può quindi intervenire manualmente;
* Di sistema: non visibili al programmatore e usati solo dal processore.

Quelli di sistema sono l’**I.R.** (*Instruction Register*) nel quale vengono memorizzate le istruzioni da eseguire; il **P.C.** (*Program Counter*) contenente l’indirizzo della successiva istruzione da eseguire; e il **P.S.W.** (*Program Status Word*) il quale contiene informazioni, opportunamente codificate, sull’esito dell’esecuzione dell’ultima istruzione, che serve soprattutto (parlando di Assembly) per il compimento di “salti” condizionati ad istruzioni (ovvero a seconda che si verifichi o meno una certa condizione specificata). Ci sono poi due registri che il processore utilizza per “comunicare” con l’esterno: il **MAR** (*Memory Address Register*) il quale contiene gli indirizzi delle celle di memoria usate per la lettura o la scrittura di un dato, e il **MDR** (*Memory Data Register*) contenente i dati che sono stati letti o che devono essere scritti nelle locazioni indicate dal MAR.

Queste sopra descritte sono caratteristiche generali che riguardano, in linea generale, tutti i processori. Il linguaggio Assembly utilizzato in questo caso è specifico però per il microprocessore 8086 (appartenente alla famiglia x86), è quindi necessario compiere opportune distinzioni ed essere maggiormente dettagliati. L’8086 è “diviso” in due parti principali: l’*Execution Unit* (**EU**), parte più “interna” nella quale si trovano A.L.U. e registri interni, e la *Bus Interface Unit* (**BIU**), parte più “esterna” in cui sono situati (principalmente) bus e registri esterni. Questo processore ha la peculiarità di avere i bus degli indirizzi a 20 bit, mentre quelli dei dati a 16 bit; questi bus (dati e indirizzi) sono inoltre unificati, per cui sullo stesso bus passano una volta gli indirizzi e una volta i dati, alternandosi. Quando viaggiano gli indirizzi, vengono usati tutti e 20 i bit mentre ne vengono usati 16 per i dati quando viaggiano loro, più 4 eventuali segnali di controllo. L’8086 è quindi capace di gestire una memoria di **1 MB** (220), segmentati in blocchi di memoria (descritti in seguito) da 64 KB l’uno (216). La memoria, come accennato prima, è divisa in quattro blocchi da **64 KB**:

* Code Segment;
* Stack Segment;
* Data Segment;
* Extra Segment.

Ognuno di questi blocchi è gestito da registri da 16 bit (appunto: 216 => 64 KB) che sono, rispettivamente, **CS**, **SS**, **DS**, **ES**. Entrando nel dettaglio dei registri dell’8086, dopo questi che sono correlati ai blocchi di memoria, ci sono quelli di indice (utilizzati per lo scorrimento e la gestione degli array) **SI** e **DI**, i registri puntatori **SP** e **BP** (per lo stack di sistema), i registri dei FLAGS (corrispondenti al P.S.W.) nel quale ci sono (ad esempio, se si vogliono considerare le operazioni aritmetiche) l’**SF** (dal quale contenuto si evince il segno del risultato), il **ZF** (che contiene 0 se il risultato è diverso da 0, e contiene 1 se il risultato è 0), e il **CF** (che indica il carry), l’**IP** (Instruction Pointer), corrispondente (quasi) al P.C., e l’**Instruction Queue** (corrispondente all’I.R.): una coda, tipicamente da 6 byte, nel quale sono memorizzate le istruzioni da eseguire (in questo processore, al posto di prendere un’istruzione per volta come per l’I.R. ne vengono prese un po’). Quando si punta ad un istruzione bisogna fare riferimento al blocco di memoria interessato ed al segmento in cui è situato il dato con cui bisogna lavorare, mentre nel P.C. è contenuto sia il **segmento** (ovvero il blocco) che l’offset (nel modo *Segmento:Offset*) (l’**Offset** è la distanza tra l’inizio del segmento e l’elemento selezionato), nell’IP (utilizzato dall’8086) è contenuto solo l’offset; per cui, per puntare ad un certo dato, “lavorano” insieme i registri di segmento (come il CS) e l’IP. Per compiere il calcolo degli indirizzi (data la non corrispondenza tra i 16 bit dei dati e i 20 bit degli indirizzi), bisogna effettuare l’operazione **SEGMENTO\*16+IP**. Questo perché, se il contenuto del segmento da cui bisogna calcolare l’indirizzo è a 16 bit, bisogna moltiplicarlo per 16 per ottenere 20 bit (esempio: 5BF2\*16 = 5BF20, il quale, essendo un numero esadecimale, risulta 4 bit per 5 cifre, quindi 20 bit), si somma poi l’IP per ottenere la locazione di memoria precisa (esempio: 5BF20+000A7 = 5BFC7), va notato che non sempre si somma l’IP: col Data Segment, per esempio, non bisogna farlo; si è dunque ottenuto un indirizzo a 20 bit. Infine, ritornando ai registri dell’8086, vi sono quelli generali, ognuno dei quali, sebbene possa essere usato dal programmatore indifferentemente, presenta delle caratteristiche specifiche che lo rendono utile per l’esecuzione di determinate istruzioni che li utilizzano quindi implicitamente (come il MUL, che verrà spiegato in seguito). I registri generali sono:

* AX: accumulatore (usato soprattutto per le operazioni aritmetiche, anche implicitamente);
* BX: di base (usato per l’indirizzamento di una locazione di memoria);
* CX: contatore (usato implicitamente come contatore nei cicli ed esplicitamente come contatore nelle istruzioni di shift e rotazione);
* DX: dati (usato implicitamente come ampliamento di AX nelle operazioni di moltiplicazione e divisione).

Ognuno di questi registri (a 16 bit) è “diviso” in due parti: **Low** (parte bassa) e **High** (parte alta) (tutti e due ad 8 bit). Il loro impiego dipende dal contesto, cioè dall’istruzione che li coinvolge.

Chiarito ciò, bisogna descrivere il ciclo macchina e le sue fasi, ovvero l’esecuzione delle varie istruzioni di un programma.

Il **ciclo macchina** si può schematizzare con un grafo ciclico. È costituito da tre fasi fondamentali: Fetch, Decoding, ed Execute. La fase di fetch consiste nel prelevare l’istruzione da eseguire, essa avviene secondo una sequenza prestabilita di micro-operazioni (descritte di seguito considerando i processori in generale, non l’8086 nello specifico):

1. Il PC invia sul bus di indirizzi interno l’indirizzo dell’istruzione da estrarre;
2. L’indirizzo viene posto nel MAR e impostato sul bus di indirizzi esterno;
3. L’istruzione viene letta dalla locazione di memoria indirizzata, e posta sul bus di dati esterno
4. L’unità di controllo del bus riceve l’istruzione e la pone sul bus di dati interno e nel registro dati MDR, liberando così il bus di dati esterno;
5. L’unità di istruzione conserva nel registro IR l’istruzione che deve essere eseguita;
6. Il bus di dati interno è liberato;
7. Il valore contenuto nel PC viene incrementato o manipolato secondo l’esito dell’esecuzione dell’ultima istruzione (salti condizionati, ecc…).

La fase di decoding consiste quindi nella “decodifica” dell’istruzione secondo le fasi di seguito descritte:

1. Una volta prelevata, l’istruzione viene “decodificata” dall’unità di controllo (per decodifica si intende l’interpretazione del codice operativo e degli operandi dato che un’istruzione è formata proprio da questi due elementi principali che sono, rispettivamente, il codice dell’istruzione vera e propria cioè il comando da eseguire, e i dati su cui bisogna agire o che bisogna usare per eseguire il comando, questi dati possono essere costanti o contenuti in variabili, quindi in locazioni di memoria);
2. L’unità di controllo determina la sequenza di **micro-operazioni** da effettuare per eseguire l’istruzione (esistono tante sequenze di micro-operazioni quanti sono i codici operativi del linguaggio macchina: per esempio, per l’istruzione MOV <operandi>, il processore avrà memorizzato una sequenza di micro-operazioni come mettere l’indirizzo della variabile indicata negli operandi sul bus indirizzi, mettere il dato prelevato sul bus dati, ecc… utili all’esecuzione dell’istruzione. Le micro-operazioni sono memorizzate in una piccola memoria ROM interna al processore, da non confondere con la ROM a cui ci si riferisce in generale dove è memorizzato il BIOS).

Nella fase di execute vi è quindi l’esecuzione delle micro-operazioni riguardanti l’istruzione prelevata e “decodificata” precedentemente.

Nell’analisi del ciclo macchina bisogna tenere in considerazione anche un eventuale segnale di **Interrupt**: esso può servire, ad esempio, per interrompere l’esecuzione di un processo per passare quindi all’esecuzione di un altro; in ogni caso, un segnale di interrupt può avvenire correttamente solo tra la fase di execute e quella di fetch, altrimenti si ha un malfunzionamento (per esempio, se ad un computer che ha bisogno di alimentazione elettrica, questa viene a mancare durante l’esecuzione di un processo, si ha certamente un malfunzionamento).

**LINGUAGGIO ASSEMBLY**

Avendo effettuato le dovute considerazioni, senza le quali risulta impossibile comprendere a pieno un linguaggio Assembly, si può procedere con la sintassi del linguaggio ed alcuni elementi di base.

Innanzi tutto, per programmare in Assembly, occorre un “assemblatore” (**assembler**), ovvero un programma che converta il codice scritto in Assembly nel linguaggio macchina (ovvero nella sequenza di 0 e 1) comprensibile al computer. L’assemblatore utilizzato in questo caso è in realtà un emulatore che permette di programmare in Assembly 8086 come se si stesse scrivendo codice proprio per un processore 8086, dando anche la possibilità di visionare il contenuto dei vari registri durante l’esecuzione delle varie istruzioni; il programma in questione è **emu8086**.

Quando si inizia un nuovo progetto con emu8086 verrà chiesto di scegliere tra **COM template** o **EXE template** (oltre che tra altre opzioni possibili). La differenza sta nel fatto che con il COM template si lavorerà monosegmento (con quello del codice, **Code Segment**), mentre con l’EXE template si lavorerà con più segmenti (quello del codice, quello dei dati, ed eventualmente quello dello stack; rispettivamente **Code Segment**, **Data Segment**, e **Stack Segment**).

L’Assembly non è **case sensitive**, quindi non esiste alcuna differenza nello scrivere lettere maiuscole o minuscole: verranno considerate tutte allo stesso modo (***case insensitive***).

La dichiarazione delle variabili avviene secondo la forma **<nome> <tipo> <valore iniziale>**. I tipi di variabile possibili, per quanto riguarda la versione Assembly dell’8086, sono **DB** e **DW**: con DB (*Define Byte*) si dichiara una variabile il cui contenuto potrà avere una dimensione massima di 1 Byte (8 bit), con DW (*Define Word*) si dichiara invece una variabile la cui dimensione non potrà superare i 2 Byte (16 bit). In altre versioni dell’Assembly, vi è anche la possibilità di definire variabili più grandi di 16 bit (come **DD**, *Double Word*, 32 bit quindi 4 Byte), ciò non è permesso però per la versione riguardante l’8086 date le caratteristiche descritte prima (i registri e il bus dati sono a 16 bit). Un esempio di dichiarazione di una variabile è “*a DB 0*” con questo codice si dichiarerà quindi una variabile di nome “a” la cui dimensione e spazio occupato in memoria è di 1 Byte, con un valore iniziale di 0.

La sintassi per le istruzioni varia invece a seconda dell’istruzione stessa. In questa esperienza ci siamo serviti delle istruzioni **MOV**, **ADD**, **MUL**, e **DIV**.

L’istruzione MOV permette il trasferimento di dati. La sintassi è **MOV <destinazione>, <sorgente>**. Quando la destinazione è un registro (come AX per esempio), il sorgente può essere una costante (come il valore decimale 12), un altro registro (come BX, ovviamente quando si parla di locazioni di memoria e registri come sorgenti/destinazioni si intende il valore contenuto al loro interno), o una locazione di memoria (praticamente una variabile o proprio un indirizzo di una locazione di memoria come 178h ad esempio, il quale, essendo un indirizzo, andrà espresso dentro parentesi quadre per distinguersi dalle costanti: [178h]). Se invece la destinazione è una locazione di memoria (o una variabile), il sorgente può essere una costante o un registro.

L’istruzione ADD serve a sommare il valore del sorgente a quello della destinazione secondo la sintassi (identica a quella del MOV) **ADD <destinazione>, <sorgente>**. Anche per l’ADD le regole imposte per le destinazioni e i sorgenti sono le medesime del MOV. Va notato che questa istruzione andrà a modificare implicitamente anche alcuni valori di registri dei FLAGS come SF, ZF e CF (il cui funzionamento è illustrato in precedenza) a seconda dell’esito dell’operazione aritmetica.

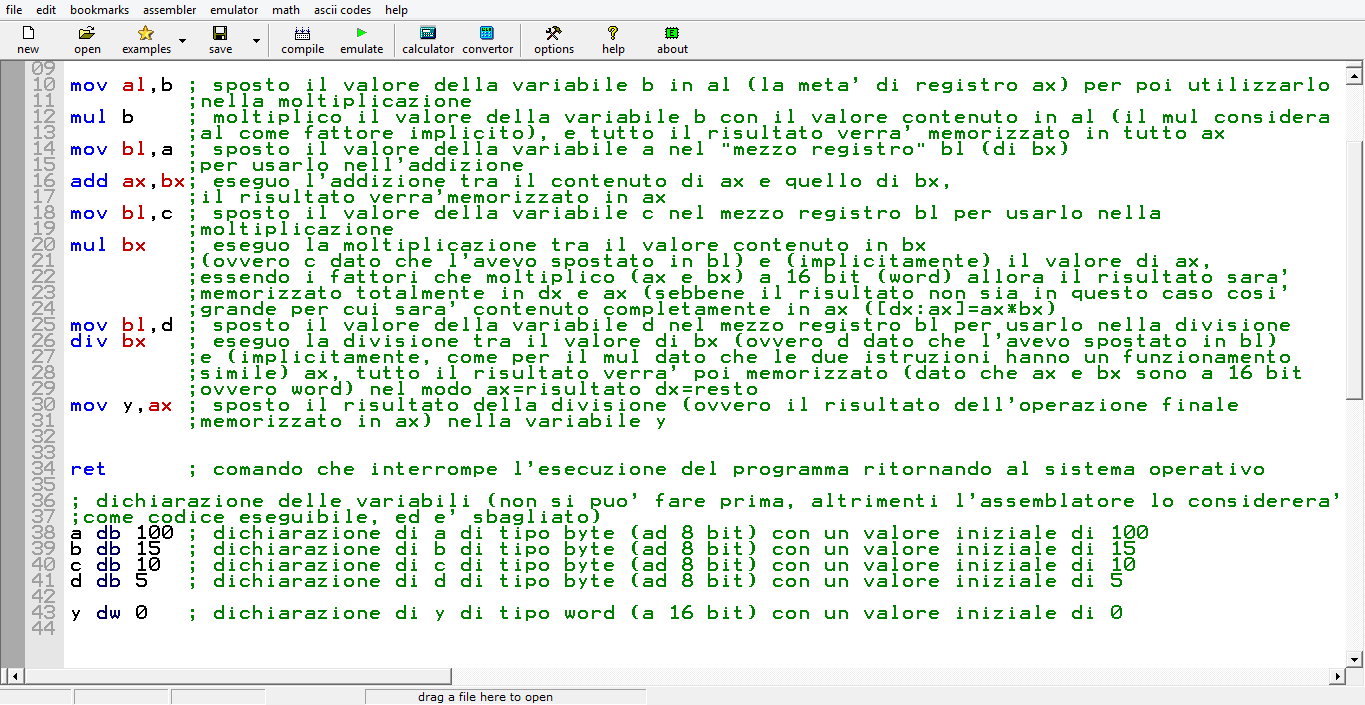
L’istruzione MUL serve a moltiplicare il valore del sorgente con quello del registro AL se il sorgente è ad 8 bit, o con quello di tutto AX se il sorgente è invece a 16 bit. La sintassi è la seguente: **MUL <sorgente>**. A differenza delle istruzioni viste prima, il MUL ha solo il sorgente come operando, questo perché la moltiplicazione col valore contenuto in AL o in AX (secondo i criteri sopra descritti), avviene implicitamente; non è quindi possibile moltiplicare direttamente un valore del sorgente con un registro che non sia AL o AX quando si usa l’istruzione MUL. Il risultato dell’operazione sarà memorizzata in AX nel caso in cui il sorgente sia Byte e quindi venga moltiplicato per AL, oppure in DX ed AX se il sorgente sia Word e venga moltiplicato per AX.

L’istruzione DIV permette di dividere il contenuto di AL per quello del sorgente nel caso in cui quest’ultimo sia Byte, oppure di dividere il contenuto di tutto il registro AX per il sorgente se questo è Word. La sintassi è **DIV <sorgente>**. Come per il MUL, anche per il DIV il valore da dividere per il sorgente è implicito, con questa istruzione non si può infatti dividere il valore di un registro che non sia AL o l’intero AX (sempre rispettando i criteri scritti prima). Va inoltre notato che DIV non accetta come sorgente valori costanti (come 15, ad esempio), bisognerà necessariamente appoggiare l’operando per il quale si vuole dividere il contenuto di AL o AX in un altro registro. Il risultato di una divisione effettuata mediante il DIV verrà memorizzato in AL (e il resto andrà in AH) nel caso in cui il sorgente sia Byte, oppure verrà memorizzato in tutto AX (e il resto andrà in DX) se il sorgente è Word.

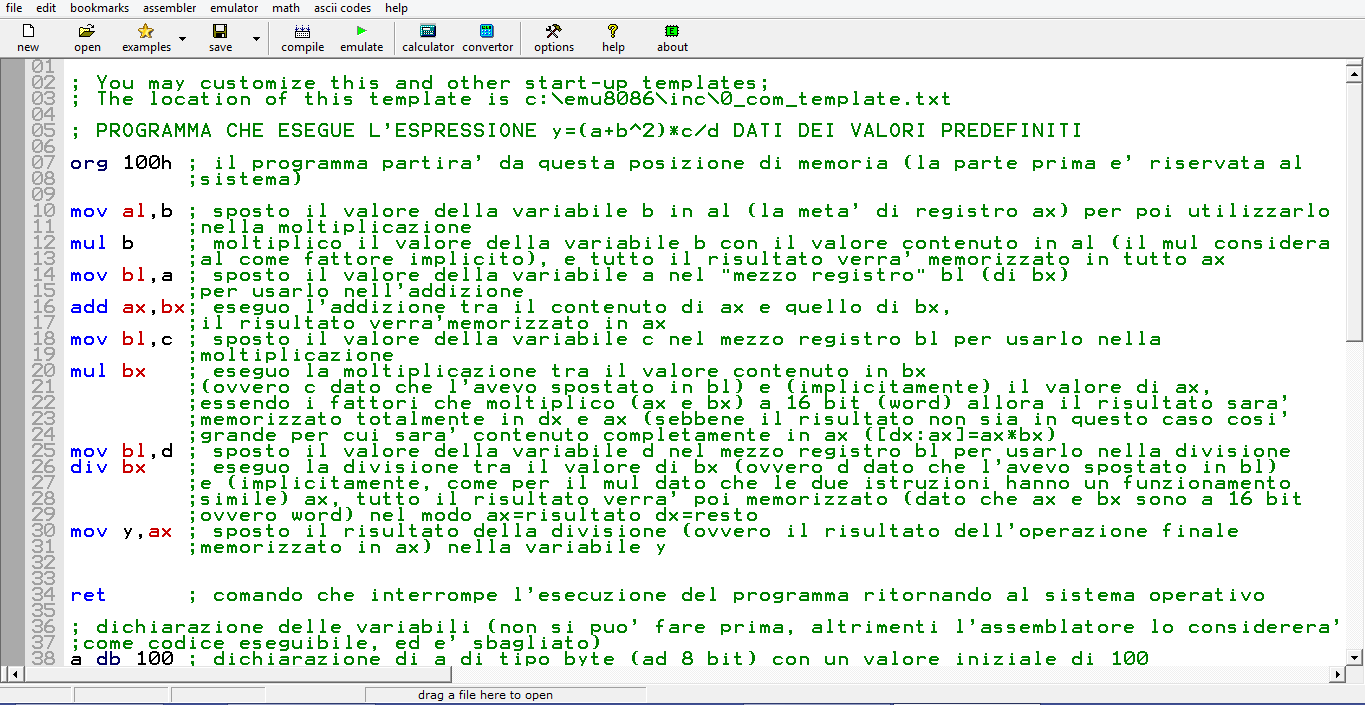
Va specificato infine il funzionamento di due istruzioni fondamentali: **org 100h** e **ret**. Con org 100h si intende che il programma inizia da una certa posizione: parte dall’indirizzo di memoria indicato (100h) in poi, la parte prima è riservata. L’istruzione ret invece ferma il programma: il controllo ritorna al sistema operativo. La dichiarazione delle variabili va effettuata dopo ret dato che se si fa prima il compilatore lo interpreta come codice eseguibile, avendo quindi un effetto indesiderato. I commenti in Assembly si ottengono anteponendo un *punto e virgola* (**;**) al commento stesso.

**CODICE DEL PROGRAMMA DELL’ESPERIENZA**

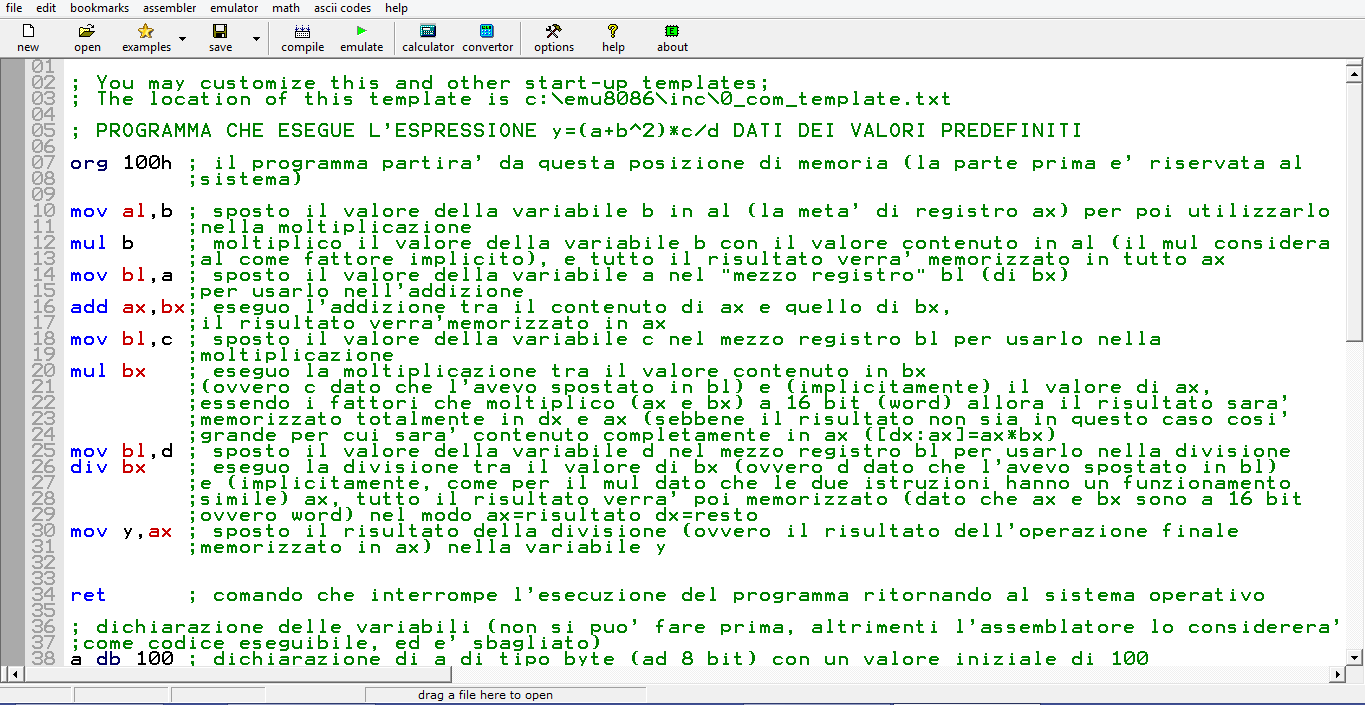
Per questa esperienza abbiamo realizzato un programma che effettua calcola il risultato dell’espressione **y=(a+b2)\*c/d** dando alle variabili dei valori prestabiliti (in questo caso, y=0; a=100; b=15; c=10; d=5). Tutte le variabili sono di tipo Byte, fatta eccezione per y che è Word.



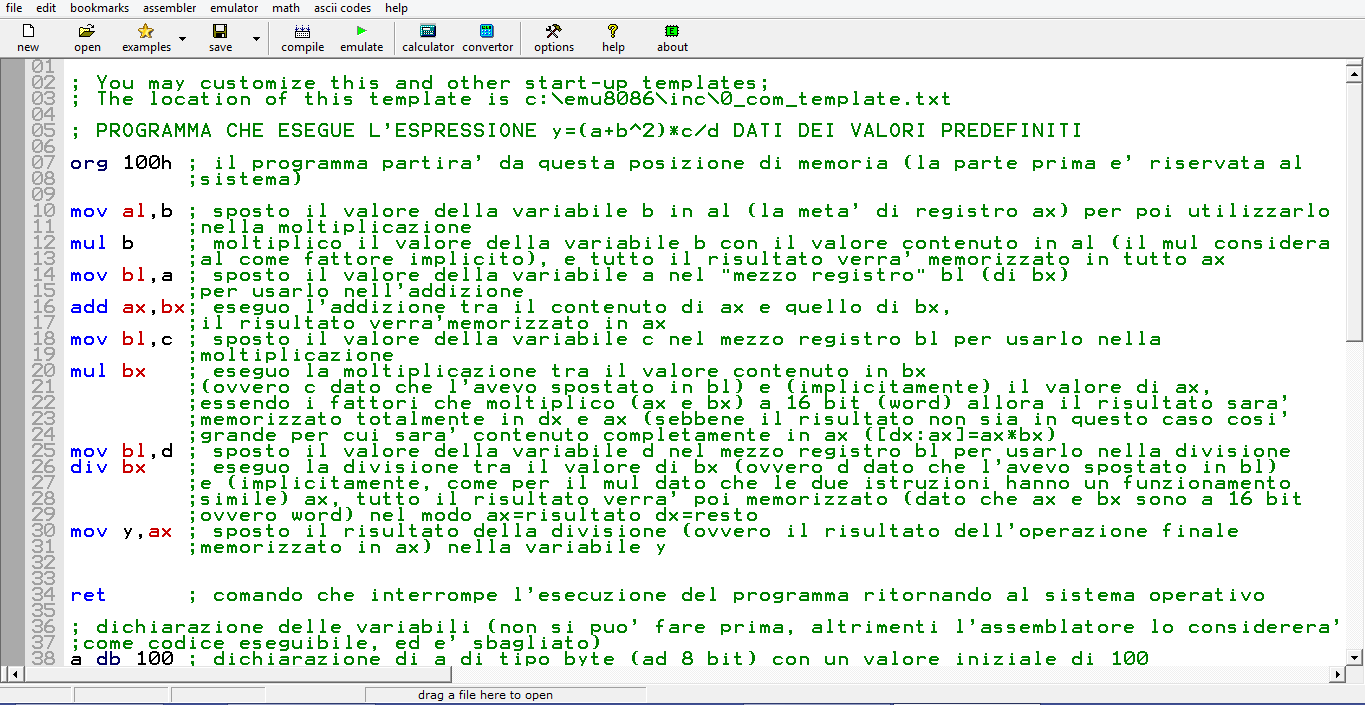
Innanzi tutto ho spostato il valore di b in al per poi moltiplicarlo per se stesso mediante la seconda istruzione, il risultato è quindi memorizzato in ax.



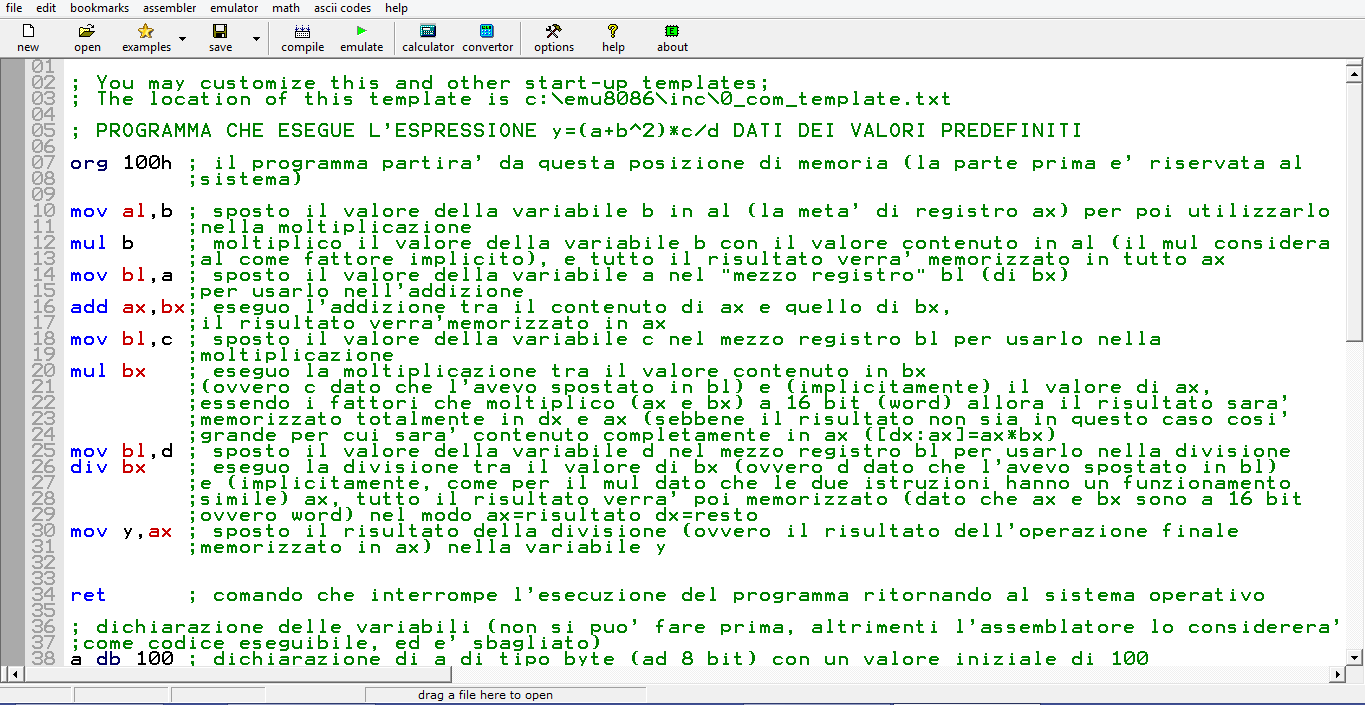
Dovendo effettuare l’addizione tra a e b2 ho spostato il valore di a in bl e quindi addizionato tutto il valore di bx (ovvero il valore di a memorizzato poco prima in bl) a quello di ax in cui era invece memorizzato il risultato di b\*b (ovvero b2); non ho potuto scrivere direttamente add ax, a dato che ci sarebbe stata un’incompatibilità di dimensione tra a (ad 8 bit) ed ax (a 16 bit), mi sono quindi servito di bx che è proprio di 16 bit. Tutto il risultato è quindi contenuto in ax.



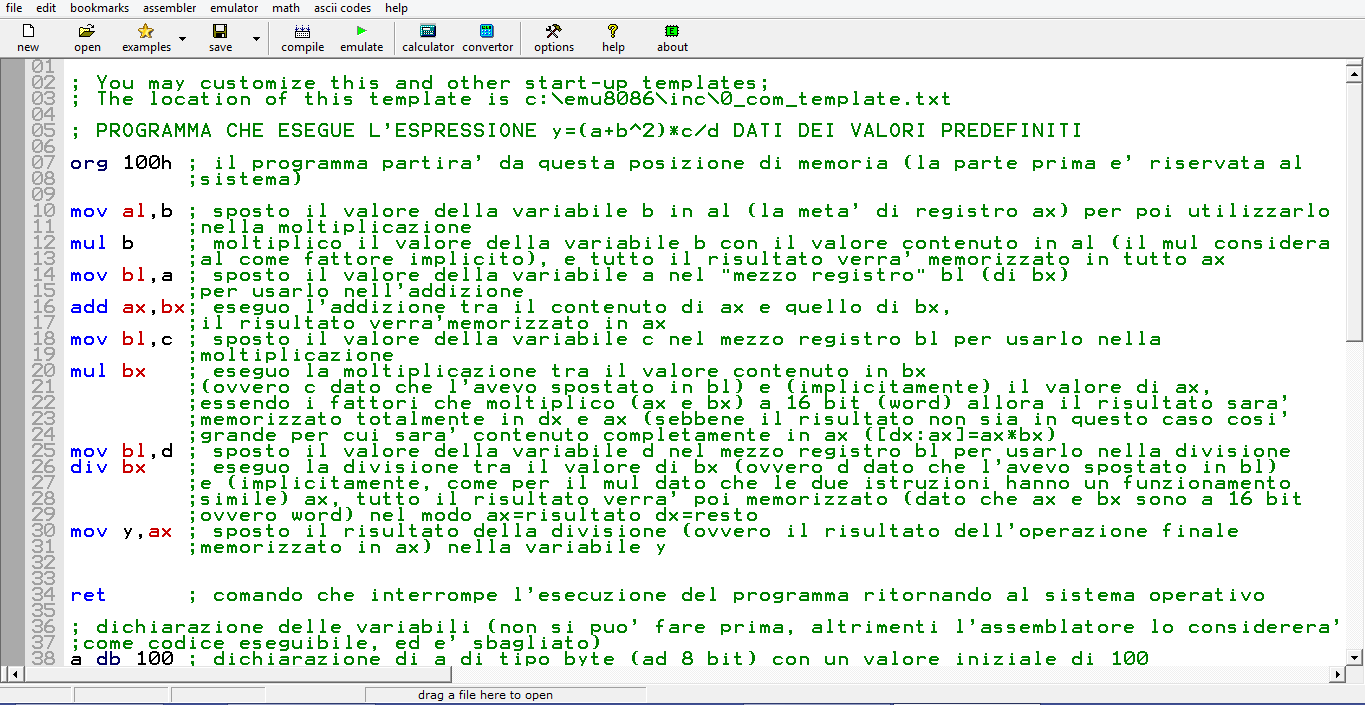
A questo punto, dovendo effettuare la moltiplicazione tra c (ad 8 bit) e tutto il risultato contenuto in ax (a 16 bit), sono ricorso alla tecnica di prima: ho spostato il valore di c in bl, e poi moltiplicato tutto bx per (implicitamente) ax. Tutto il risultato a questo punto è memorizzato in dx e ax, ma (dato che i valori dati sono predefiniti) questo non supera comunque i 16 bit, per cui si troverà interamente in ax.



A questo punto sono ricorso alla stessa tecnica anche per effettuare la divisione fra tutto il risultato dell’espressione fin ora contenuta in ax (a 16 bit) e la variabile d (ad 8 bit), ho dunque spostato d in bl e diviso tutto il contenuto di ax per tutto il registro bx mediante l’istruzione DIV.



Ho spostato quindi il risultato memorizzato quindi in ax nella variabile y (senza problemi di incompatibilità dato che tutti e due sono a 16 bit).



Ho infine controllato in fase di emulazione il contenuto di y, verificando la buona riuscita del programma avendo ottenuto il corretto risultato di 650.

