|  |  |  |
| --- | --- | --- |
| Altomare Andrea | Lab. Sistemi esperienza n. 14 | Classe 3^A 9/06/2015 |

**Assembly 8086: memoria multisegmento, trasferimento dati su seriale e modalità grafica**

**MEMORIA MULTISEGMENTO**

I moderni programmi in un computer non utilizzano un unico blocco di memoria per gestire tutti i segmenti (modello **monosegmento**), ma più segmenti (modello **multisegmento**). Questi due tipi di programmi si possono distinguere dall’estensione (**.com** per il monosegmento, **.exe** per il multisegmento), ed usano due modelli di memoria differenti. Va chiarito che i programmi che utilizzano il modello di memoria monosegmento (.com) non sono più supportati nativamente dal sistema operativo **Microsoft Windows**.

In Assembly, quando si realizza un programma destinato ad utilizzare un modello di memoria multisegmento, bisogna inserire dei comandi per gestire i vari segmenti.

Innanzitutto si usa la direttiva **.model** per dichiarare il modello di memoria da utilizzare. Quello monosegmento utilizza il modello **tiny** (dato dalla direttiva **org 100h**), mentre il più semplice modello di memoria multisegmentata è **small** (il quale fa uso di un segmento per il codice ed uno per i dati). Esempio: **.model small**

Per usufruire dei segmenti si usano le direttive segmento.

Sintassi:

<NOME\_SEGMENTO> SEGMENT

-----------------

-----------------

-----------------

-----------------

<NOME\_SEGMENTO> ENDS

**Segment** è la parola chiave indicante un segmento, mentre **ends** è la parola chiave indicante la fine del codice di un segmento. All’interno di queste direttive segmento si mettono i dati o le istruzioni che devono contenere (a seconda di quale segmento si sta definendo). Alcuni assemblatori (come il MASM), per capire a quale segmento si riferisce il segmento definito con la direttiva sopra indicata, utilizzano la direttiva **ASSUME**. La sintassi è di seguito indicata:

ASSUME

CS:PIPPO

DS:PLUTO

A sinistra vanno specificati i segmenti reali (**Code**, **Data**, **Stack**, **Extra**) ai quali si riferiscono quelli a destra (definiti con un nome scelto dal programmatore).

Oppure si possono utilizzare le direttive segmento **semplificate**. Sintassi:

.DATA

----------

----------

----------

----------

.CODE

----------

----------

----------

----------

**.data** e **.code** rappresentano già i segmenti effettivi ai quali si riferiranno (rispettivamente) i dati ed il codice usati per definirli. Si può anche stabilire la dimensione che dovrà occupare il segmento definito semplicemente mettendo il valore (espresso in base esadecimale) a destra del nome del segmento. Esempio: **.DATA 100h** (in questo modo il segmento dati usato dal programma avrà a disposizione una dimensione di 100h).

In un programma con modello di memoria multisegmento è necessario settare il valore del registro **DS** (indicante l’inizio del Data Segment), poiché quest’operazione non avviene in automatico come per gli altri registri segmento. Innanzitutto si sposta nel registro **ax** il valore della **variabile d’ambiente** (o **di sistema**) “**data**” la quale indica l’inizio del segmento dati, poi si sposta il valore da **ax** a **ds** (il valore va spostato prima su ax poiché i registri di segmento non possono assumere un valore direttamente da un indirizzo di memoria ma deve passare prima per un registro del processore, proprio come avviene per le variabili normali). La sintassi è la seguente:

MOV AX,@DATA

MOV DS,AX

Il simbolo “**at**” (‘**@**’) prima di **DATA** non è obbligatorio per tutti gli assemblatori.

Alla conclusione di un programma **.exe** si mettono delle istruzioni che permettono una corretta “*chiusura*” del programma da parte del sistema operativo (se non le si mettono, potrebbero esserci degli errori).

Sintassi:

MOV AH,4Ch

MOV AL,00h

INT 21h

END

Oppure:

MOV AX,4C00h

INT 21h

END

Innanzitutto si setta il servizio di “***terminazione***” **4Ch** sul registro **ah**, poi si imposta il valore **00h** in **al**, e in ultimo si richiama l’**ISR** tramite l’interrupt DOS **21h**; a questo punto il sistema operativo eseguirà una serie di operazioni di rilascio della memoria per chiudere correttamente il processo con memoria multisegmento. Il set del servizio si può eseguire anche direttamente in tutto **ax** spostandovi il valore **4C00h** poiché le prime due cifre andranno in **ah** e le ultime due in **al** (ottenendo quindi lo stesso risultato). Infine si mette la parola chiave **end** indicante la fine di un programma Assembly multisegmento.

**INCLUDERE PROGRAMMI CON EMU8086**

In questa esperienza però abbiamo usato comunque un modello di memoria **monosegmento**. Ci siamo serviti inoltre (al fine dell’esperienza) di programmi già pronti creati appositamente per l’**emu8086** (scritti in **Visual Basic**). Per richiamare questo tipo di programmi si usa la direttiva **#start=<nome\_programma>#**. Questo non è un comando standard del linguaggio Assembly, è semplicemente una direttiva gestita dall’**emulatore emu8086** per permettere certi tipi di operazioni altrimenti impossibili se non si è in possesso di un’attrezzatura adeguata.

**TRASMISSIONE DI DATI SU COLLEGAMENTI SERIALI**

Se si lavora con una rete di computer, si ha il bisogno di trasferire i dati da un elaboratore all’altro. Ciò in Assembly è possibile attraverso due istruzioni.

**Lettura dati**

Si “legge” un dato da una certa porta e lo si mette in un registro apposito. Esempio: **in al,125**. Il comando **in** permette il ricevimento del dato, **al** è il registro su cui riceverlo (può essere solo **ax** o **al**) e **125** è la porta da cui riceverlo.

**Invio dati**

Si prende il dato da un registro e lo si invia ad una certa porta. Esempio: **out 127,al**. Il comando **out** permette l’invio del dato, **127** è la porta su cui inviarlo e **al** è il registro da cui prenderlo (può essere solo **ax** o **al**).

**MODALITÀ GRAFICA**

Per pilotare a piacimento i singoli **pixel** di uno schermo, si fa uso della **modalità grafica**. Per usufruirne, si deve innanzitutto settare la **Graphics Mode**. La sintassi è quella che segue:

MOV AL,13h

MOV AH,0

INT 10h

Dove **13h** è il servizio della modalità grafica, **0** è il valore da impostare su **ah** ed infine si richiama l’**ISR** tramite l’interrupt BIOS **10h**.

A questo punto sarà possibile manipolare i pixel. Per farlo, si setta prima il servizio per la manipolazione dei pixel **0Ch** nel registro **ah** e poi si impostano alcuni registri:

* **CX**: colonna del pixel da manipolare;
* **DX**: riga del pixel da manipolare;
* **AL**: colore che dovrà avere il pixel da manipolare.

A questo punto si richiama l’**ISR** tramite l’interrupt BIOS **10h**.

Esempio con la sintassi completa:

MOV AH,0Ch

MOV CX,5

MOV DX,5

MOV AL,10

INT 10h

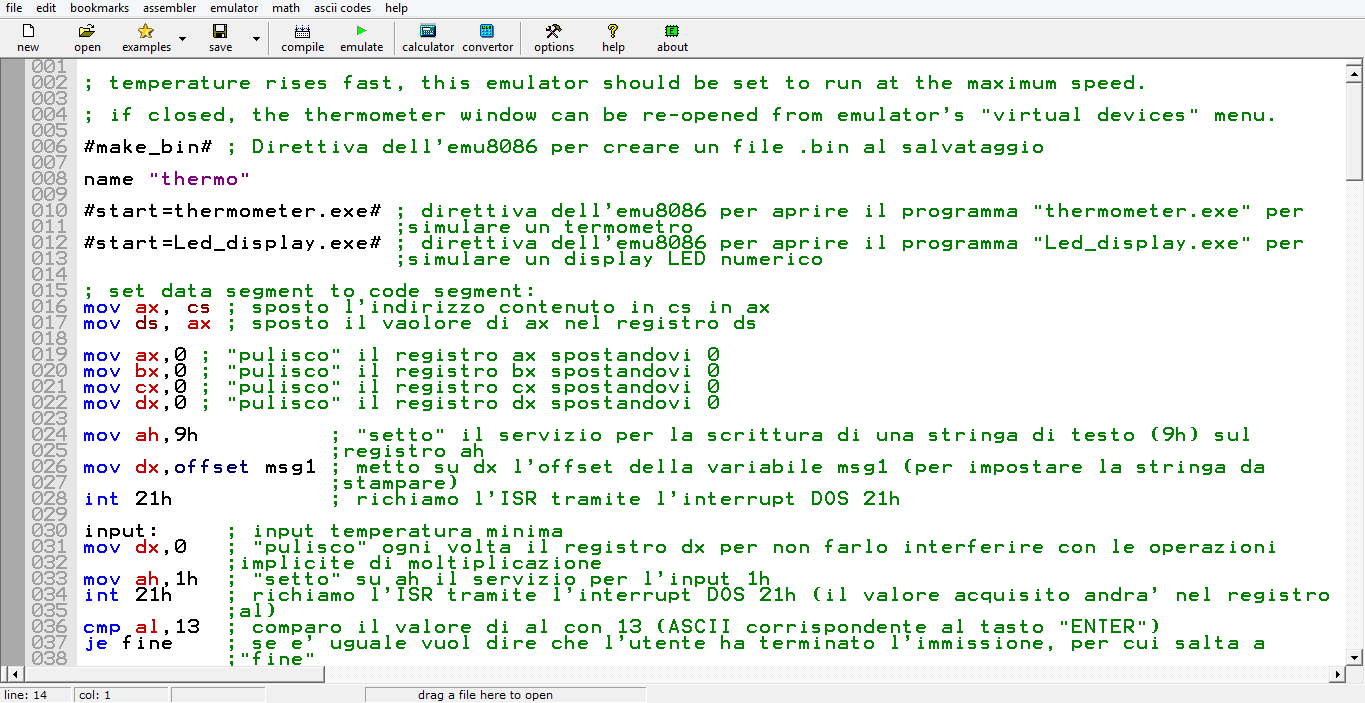
In questo modo il pixel di coordinate (**5;5**) si colorerà del colore **10**.

**PROBLEMA**

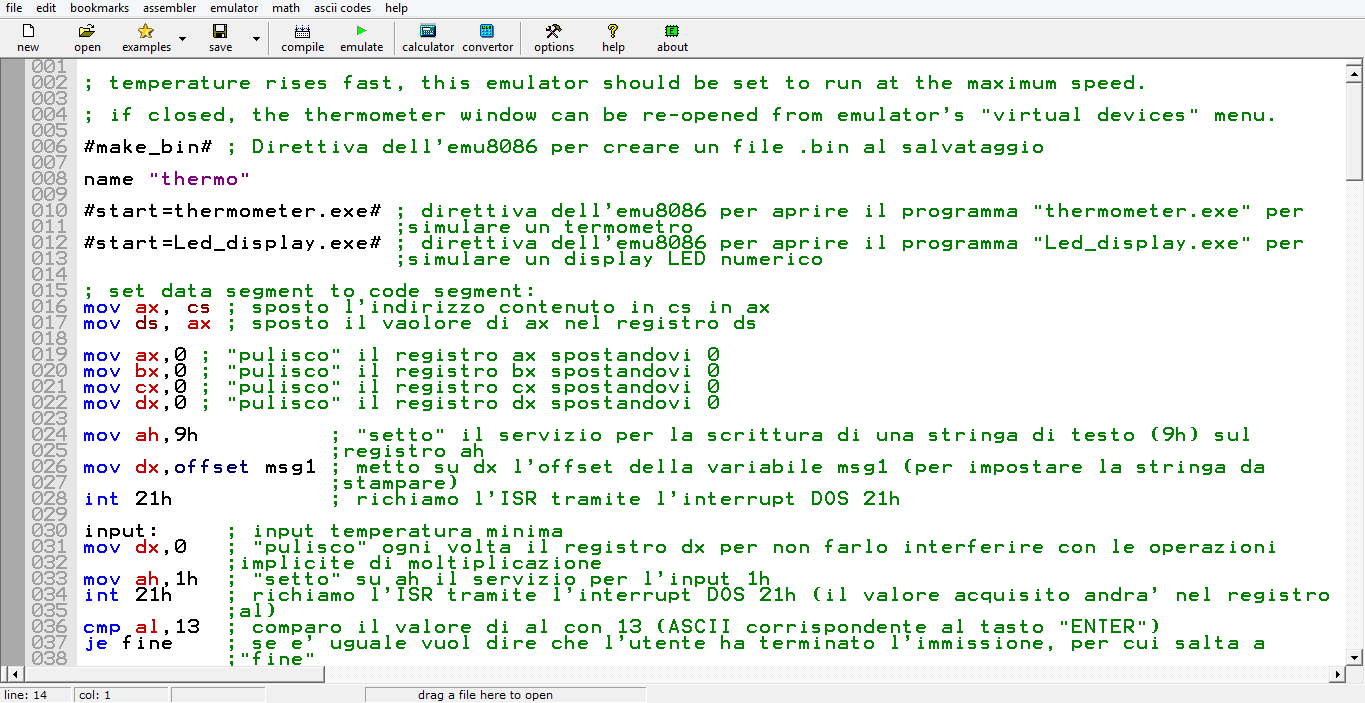
Manipolare la temperatura di un termometro virtuale **accendendo** e **spegnendo** opportunamente un bruciatore a seconda della temperatura **minima** e **massima** **immessa** dall’utente, mostrare la temperatura anche su un **display LED virtuale** e mostrare l’andamento della temperatura attraverso un **grafico** (opzionale) utilizzando la **modalità testo** o la **modalità grafica**.

**REALIZZAZIONE DEL PROGRAMMA**

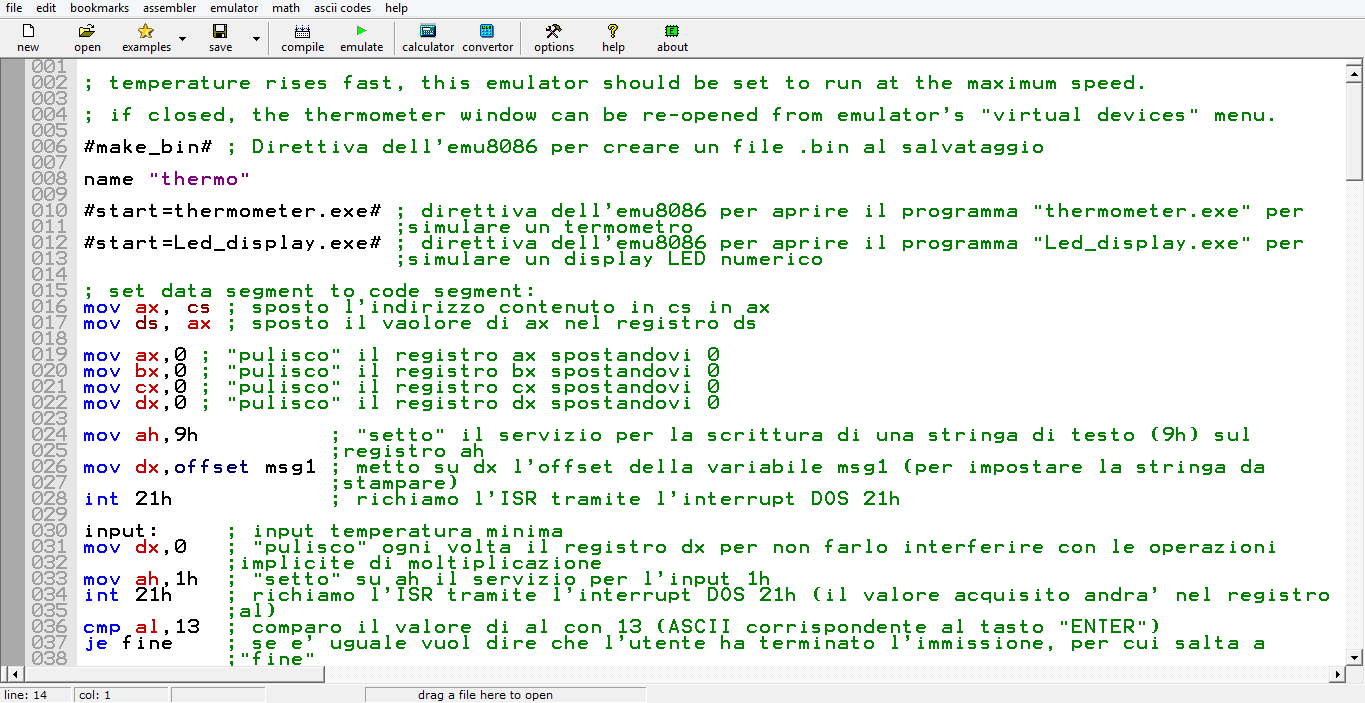
Innanzitutto ho messo le direttive per includere i programmi per emu8086 “**thermometer.exe**” (il termometro virtuale) e “**Led\_display.exe**” (il display LED virtuale). Poi, dato che ho lavorato con un modello **monosegmento**, ho settato il registro **ds** allo stesso indirizzo di **cs**.

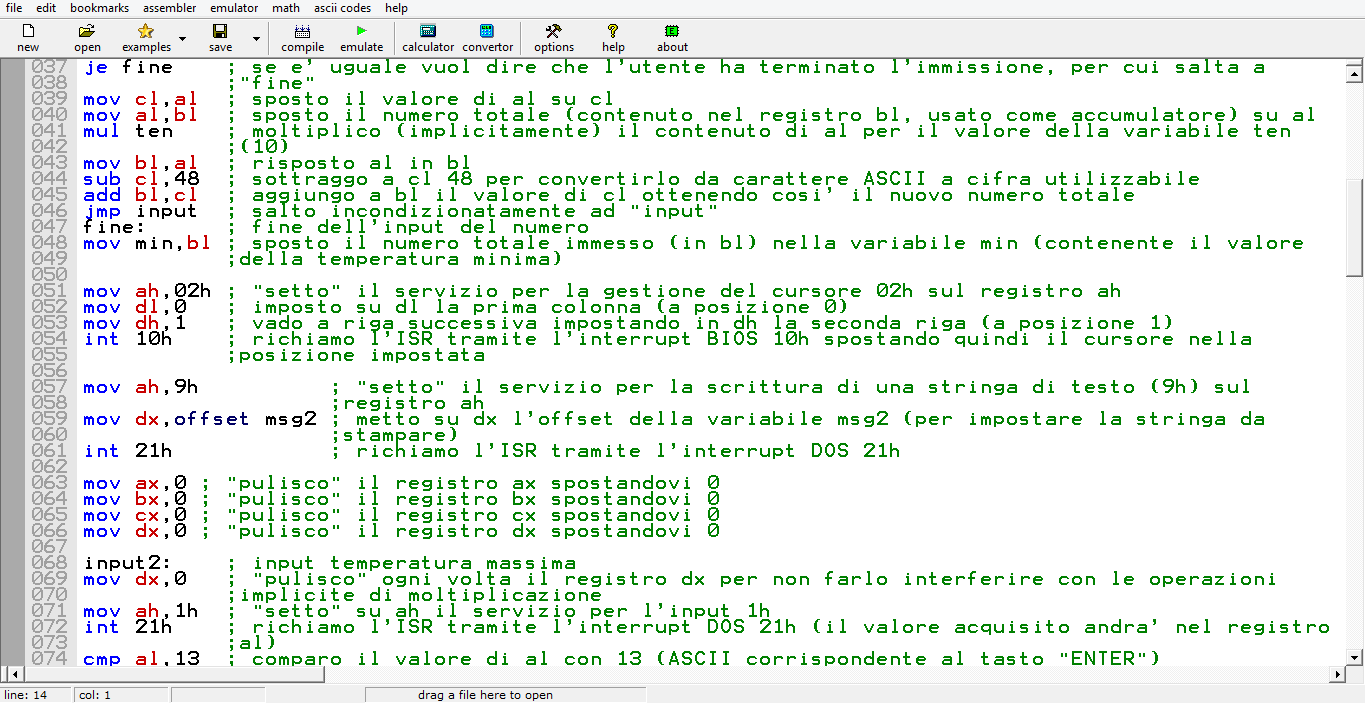


Poi ho pulito tutti i registri e sono passato alla stampa di **msg1** servendomi degli opportuni servizi **DOS**.

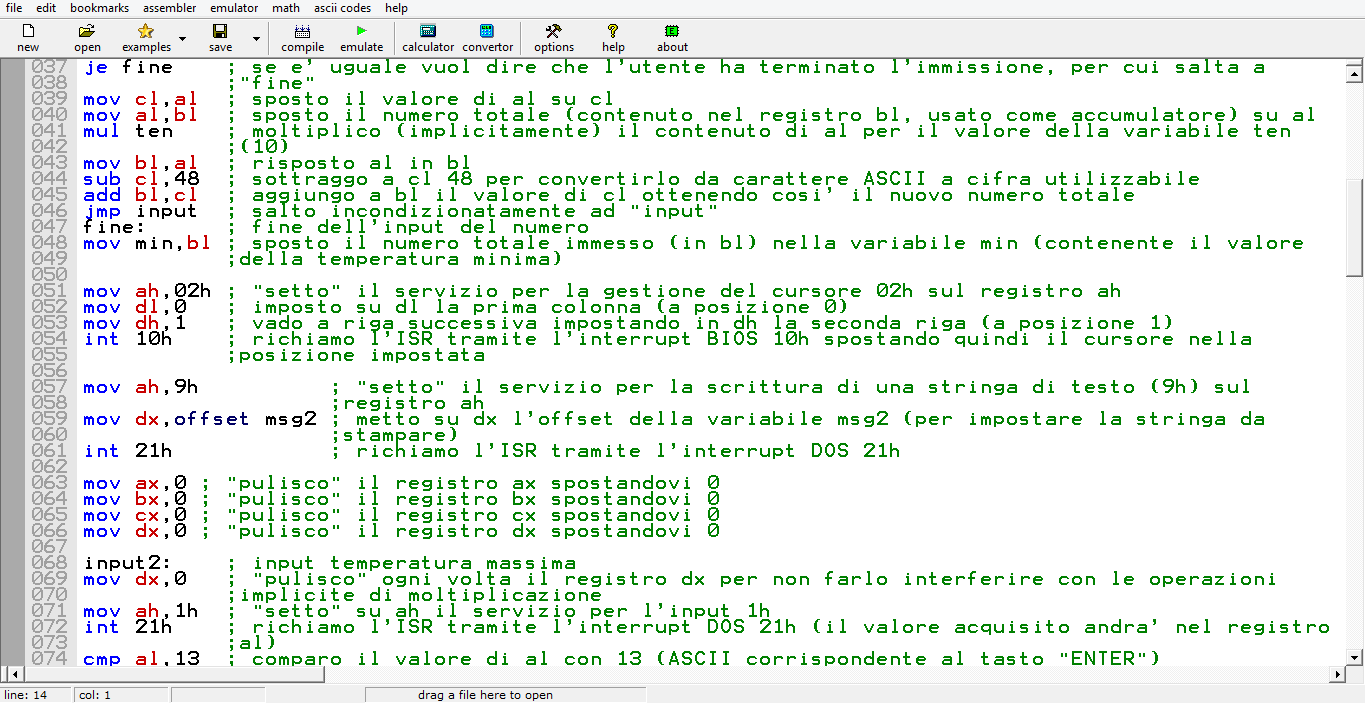


Ho quindi scritto il codice che permette all’utente di inserire un numero a più cifre (in questo caso per la temperatura minima) e salvato il valore immesso nella variabile **min**.

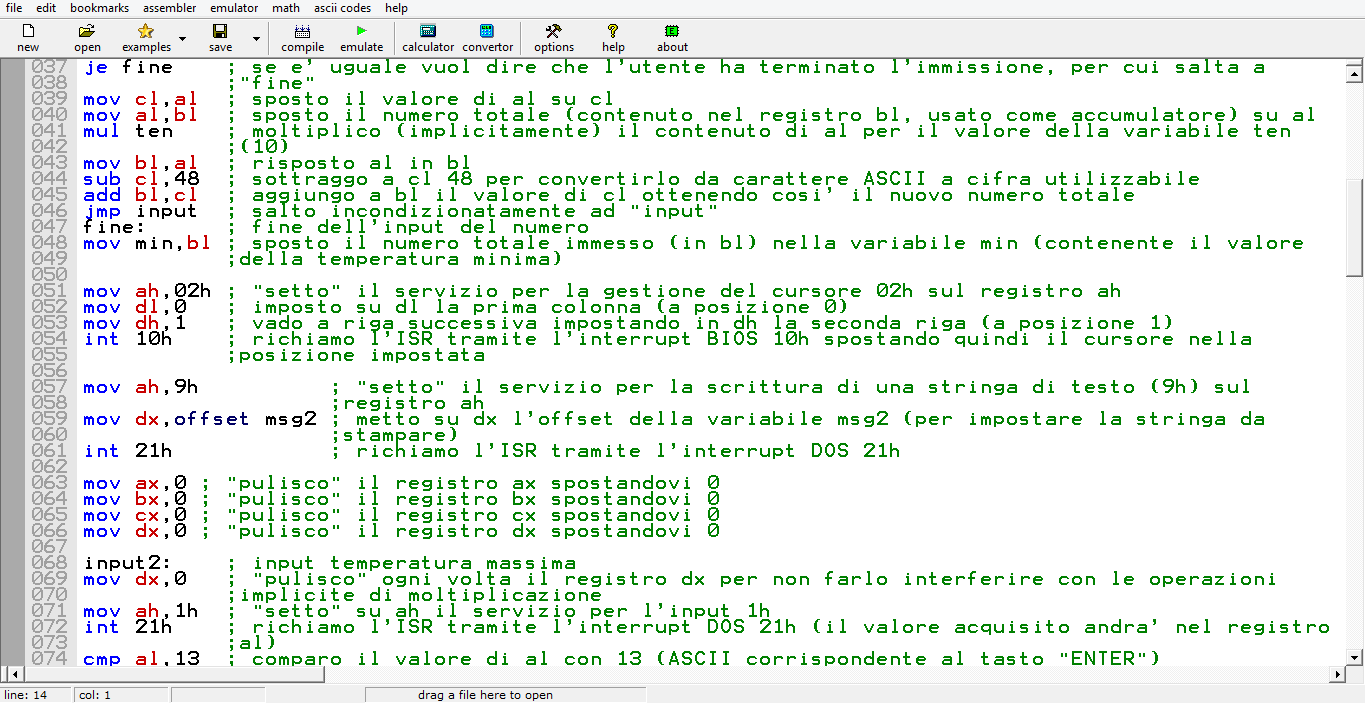


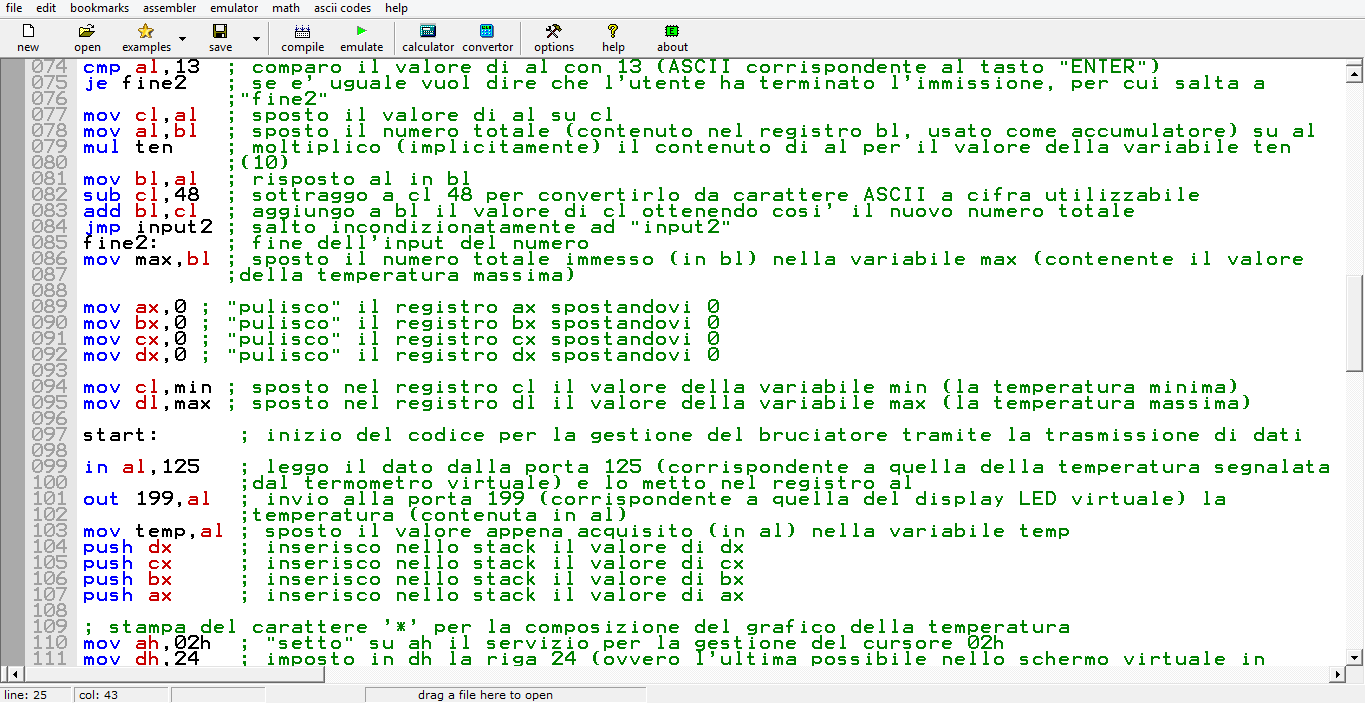


Sposto dunque il cursore alla riga successiva e stampo **msg2**.

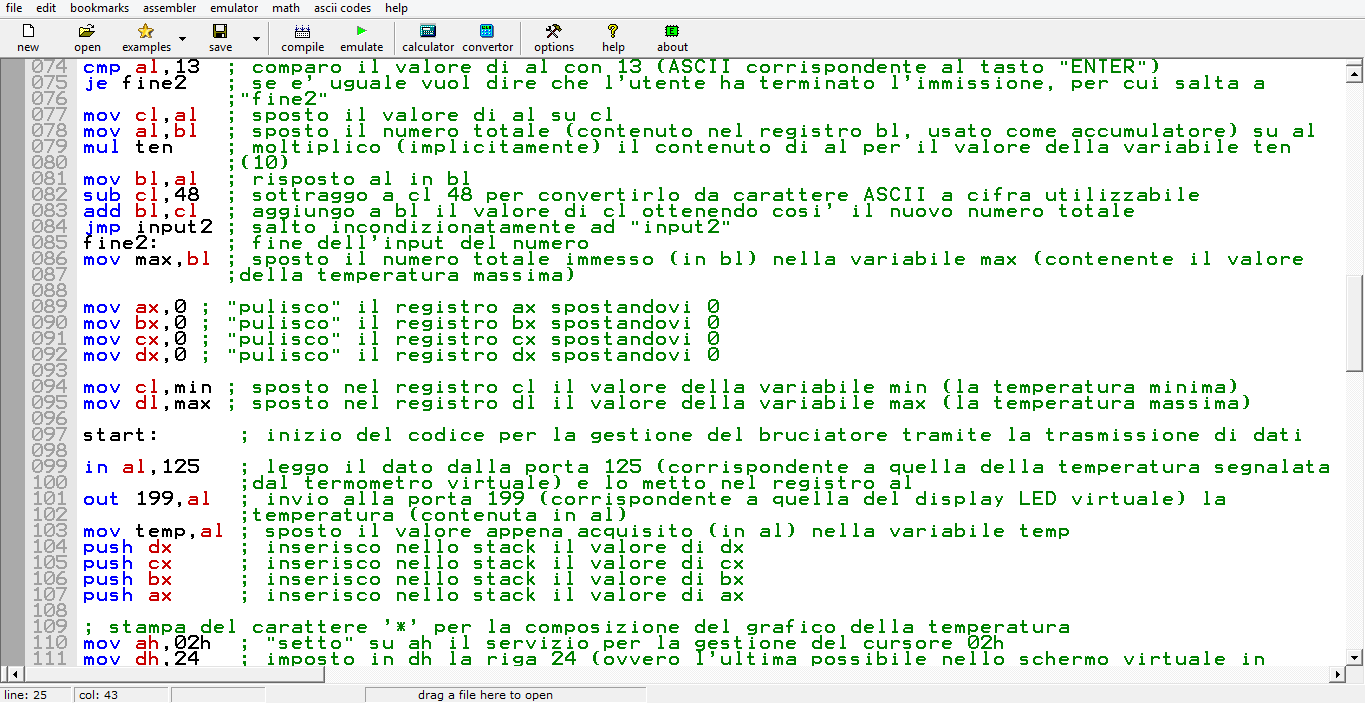


Pulisco nuovamente tutti i **registri general purpose** e procedo con l’acquisizione della temperatura massima (salvata successivamente nella variabile **max**) tramite l’input di un numero a più cifre.



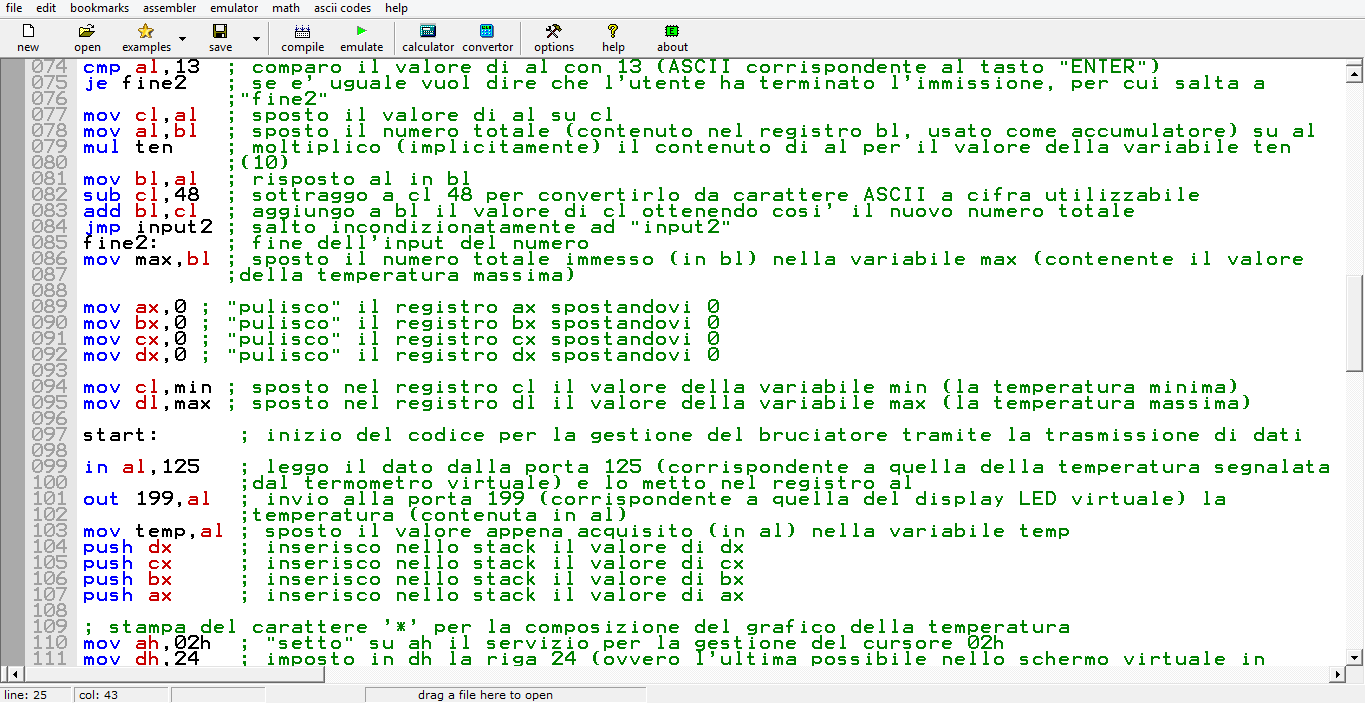


Pulisco di nuovo tutti i registri general purpose. Sposto poi sui registri **cl** e **dl**, rispettivamente, la temperatura minima e quella massima per eseguire le comparazioni con la temperatura in **al** più velocemente dato che in questo modo si risparmieranno accessi in memoria più dispendiosi in termini di tempo.



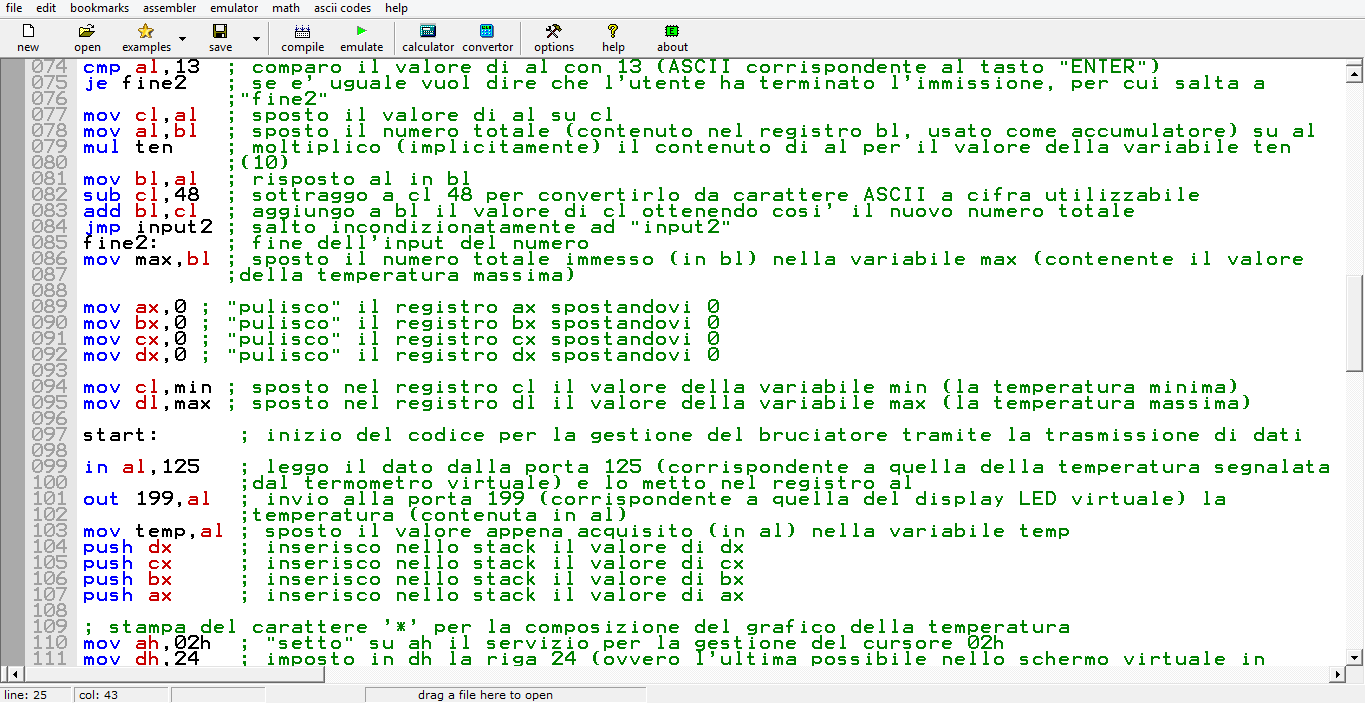
Inizio quindi il codice in loop per la gestione del termometro, del bruciatore, e del display LED.

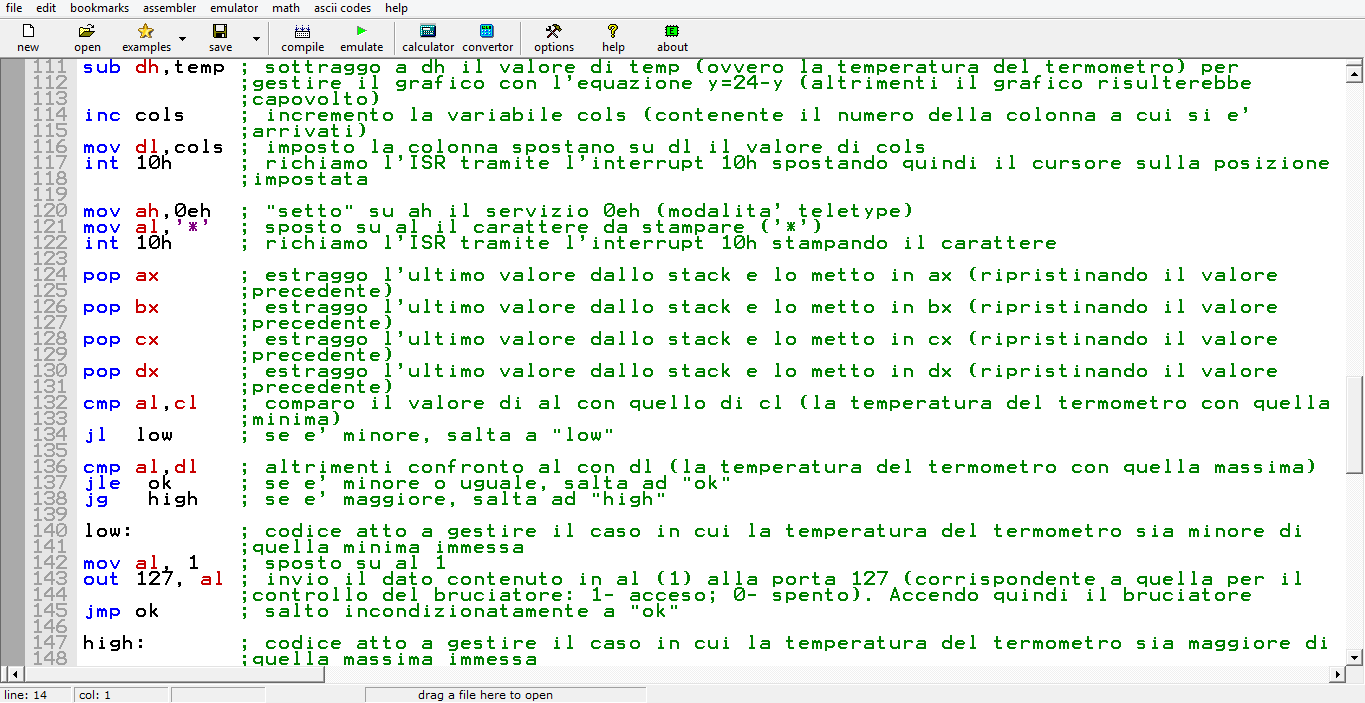
Innanzitutto leggo la temperatura segnalata dal termometro dalla porta **125**, e la metto in **al**; poi la do in output sul display LED inviandogliela alla porta **199**; sposto quindi il valore della temperatura nella variabile d’appoggio **temp**.



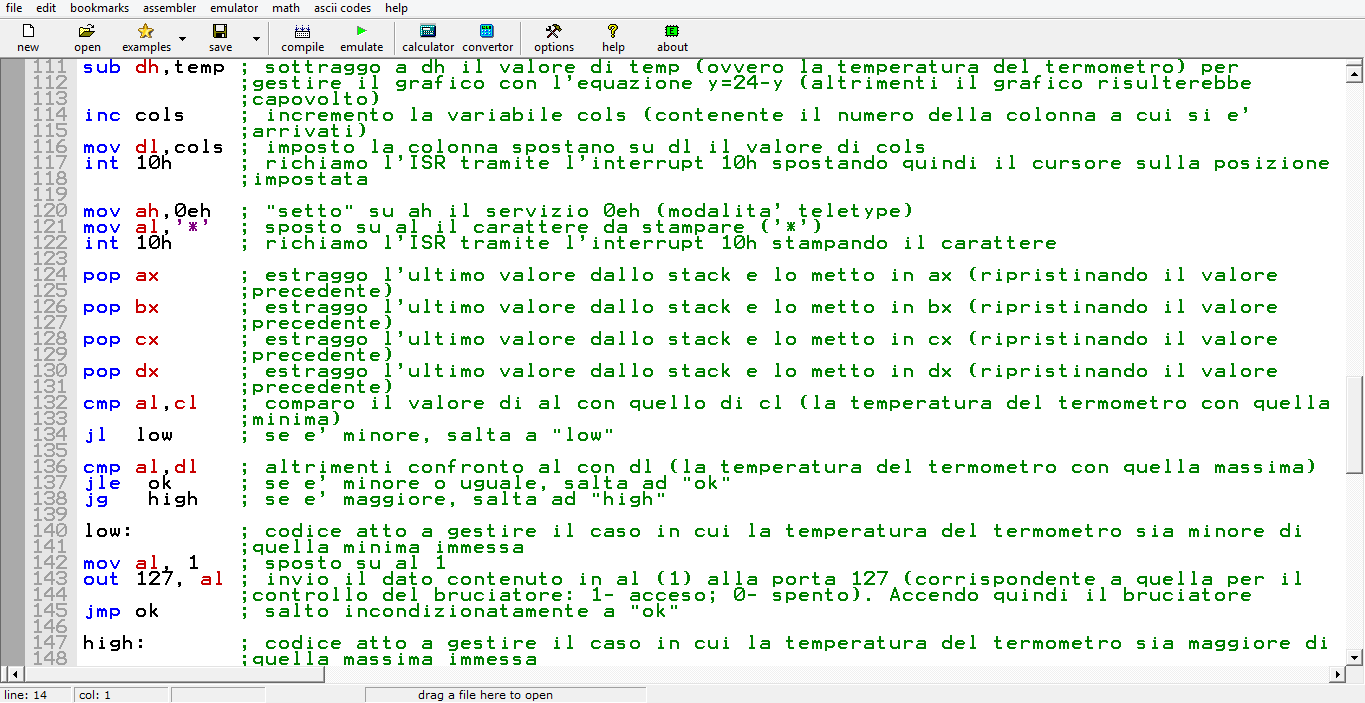
In seguito, conservo i valori dei registri general purpose inserendoli nello **stack** (dato che per le operazioni che dovrò eseguire successivamente ci sarà bisogno del loro utilizzo).

In questa parte di codice gestisco la stampa del grafico per l’andamento della temperatura: il numero di colonna viene sempre incrementato, mentre la riga viene calcolata secondo l’equazione **y=24-y** poiché l’ultimo carattere stampabile nello schermo virtuale dell’emulatore si trova in posizione di riga **24**, perciò, togliendogli il valore della temperatura (alla quale dovremo stampare il carattere se lo schermo non avesse la riga **0** al vertice in alto a sinistra) si ottiene un grafico regolare (altrimenti risulterebbe rovesciato). Poi, utilizzando la modalità telescrivente, stampo il carattere ‘**\***’ che andrà a costituire il grafico.

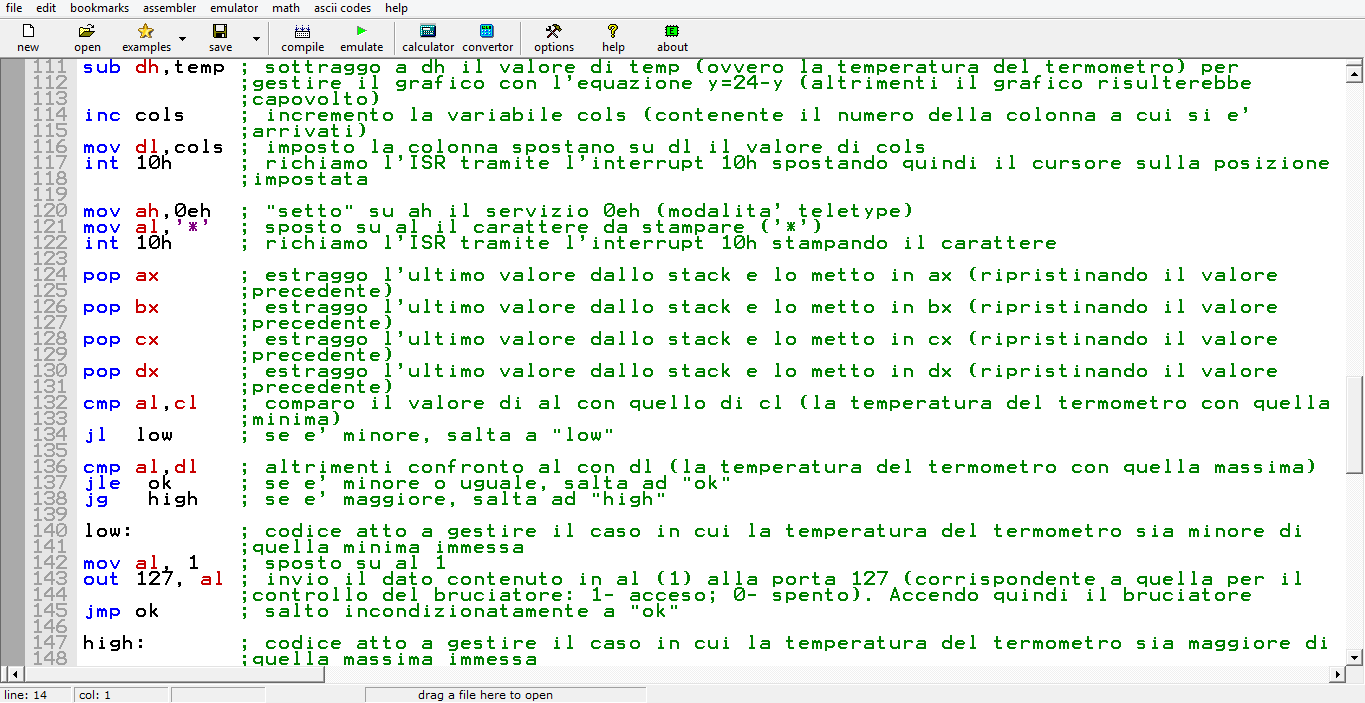




Estraggo uno ad uno gli ultimi valori dallo **stack** e li metto nei registri general purpose ripristinandone quindi i valori precedenti.



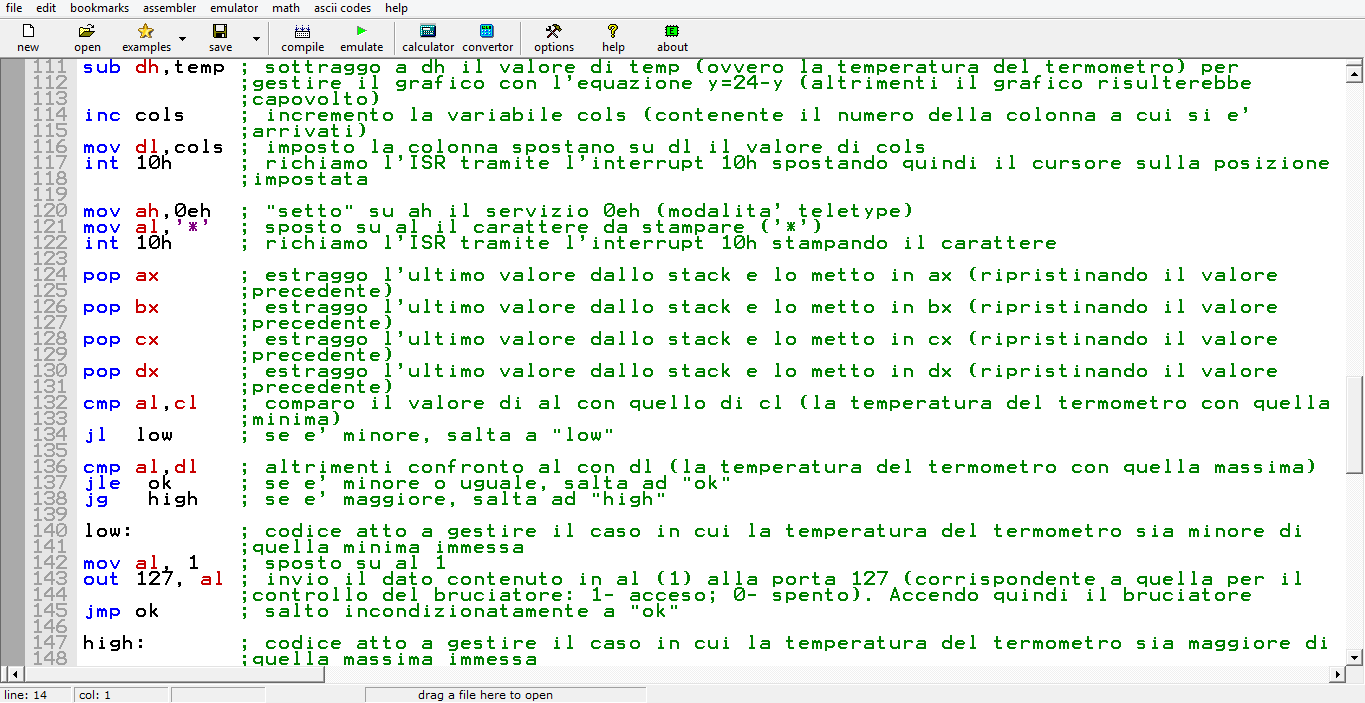
Poi comparo il valore della temperatura (contenuto in **al**) con quella minima (in **cl**), se è minore passo a “*low*”, altrimenti continuo con l’esecuzione del codice; confronto al con la temperatura massima (in **dl**), se è minore o uguale salta ad “*ok*”, altrimenti, se è maggiore, salta ad “*high*”.

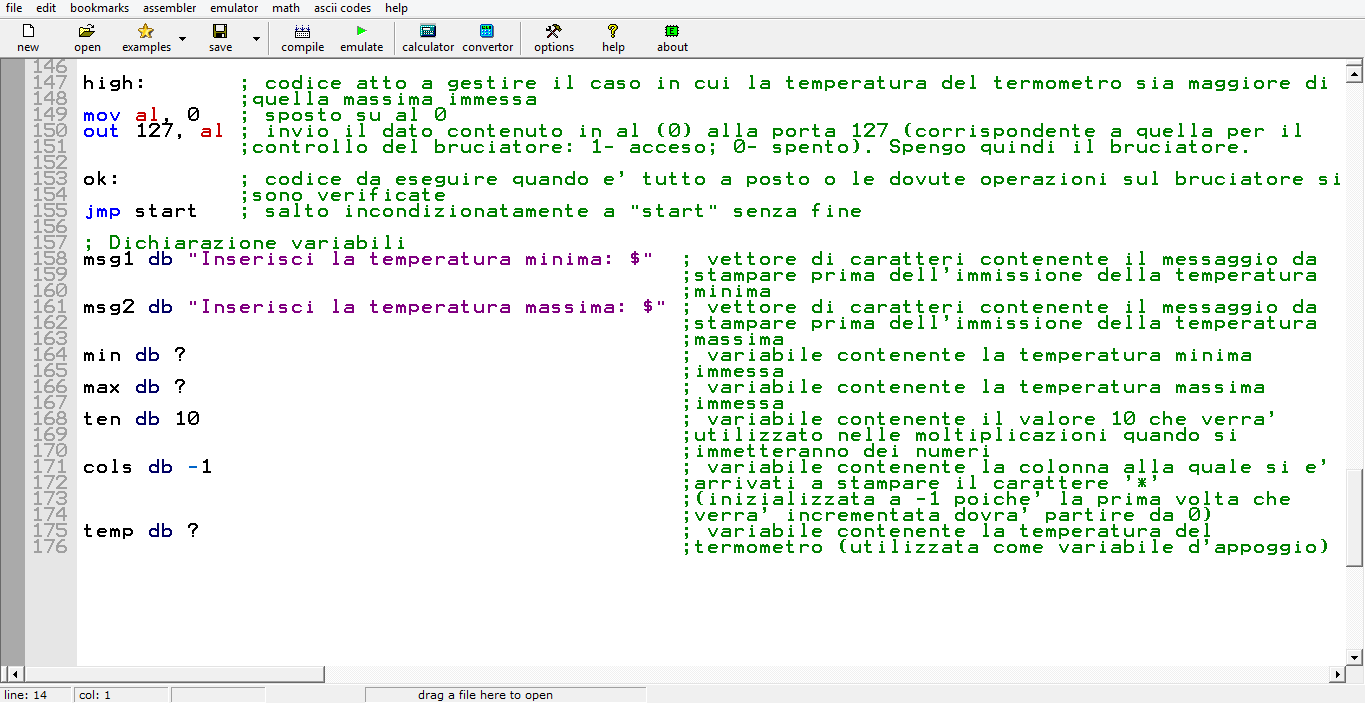


Quando passo a “*low*” vuol dire che la temperatura del termometro è più bassa di quella minima immessa dall’utente, per cui accendo il bruciatore inviando sulla porta **127** il dato **1** (dal registro **al**), poi salto incondizionatamente a “*ok*”.

Quando passo ad “*high*”, invece, vuol dire che la temperatura del termometro è più alta di quella massima immessa dall’utente, per cui spengo il bruciatore inviando sulla porta **127** il dato **0** (dal registro **al**).

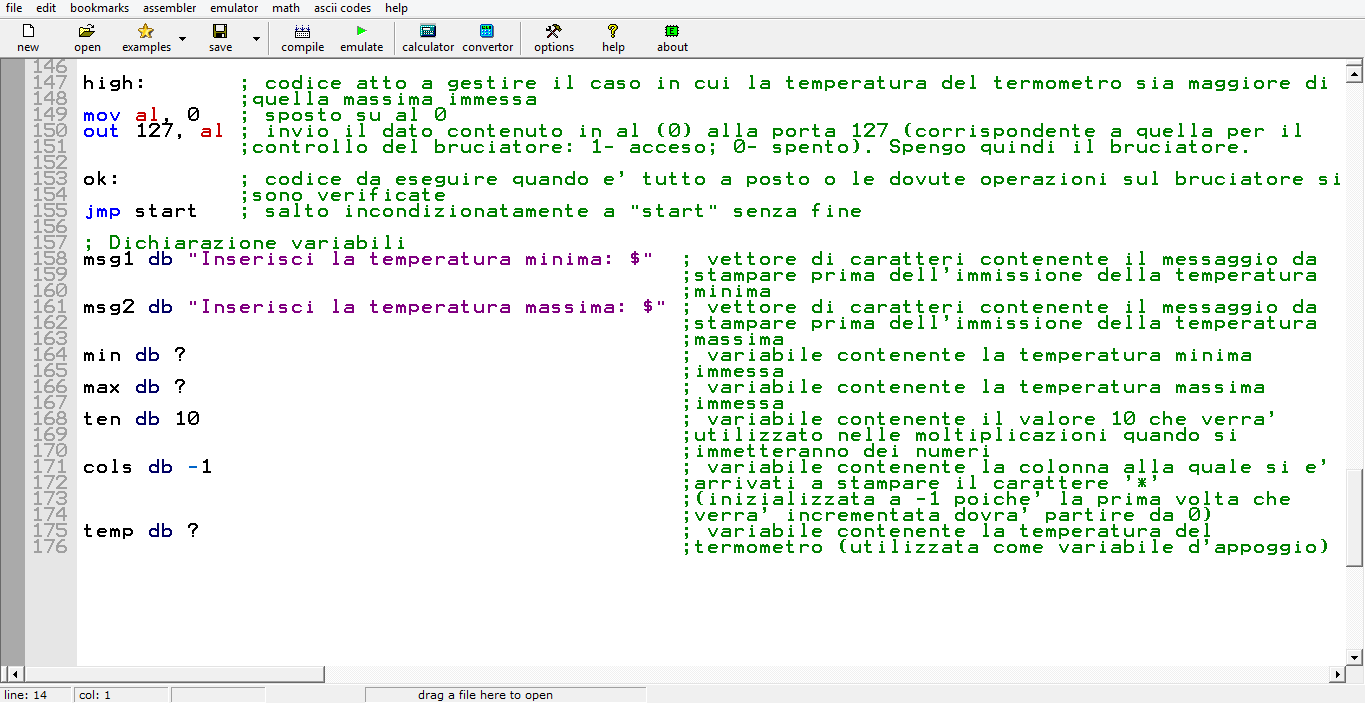
Dall’etichetta “*ok*” semplicemente salto incondizionatamente a “*start*” ricominciando nuovamente il ciclo.





Le variabili usate sono:

* **msg1**: primo messaggio per l’input della temperatura minima;
* **msg2**: secondo messaggio per l’input della temperatura massima;
* **min**: temperatura minima;
* **max**: temperatura massima;
* **ten**: contiene il valore 10 (per le moltiplicazioni);
* **cols**: contiene il numero della colonna (per la gestione del grafico);
* **temp**: variabile d’appoggio su cui sarà contenuta la temperatura del termometro.



Infine ho verificato il corretto funzionamento del programma mandandolo in esecuzione.

