Episodio 1:

Rappresentazione scritta del linguaggio macchina famiglia Intel x86 (processore 8086).

Il programma quando viene scritto si chiama programma sorgente, dopo averlo tradotto in linguaggio macchina viene chiamato programma oggetto

Successivamente, dopo un ultima fase chiamata link, diventa linguaggio eseguibile

Tutte le operazioni vengono effettuate usando i Registri, che sono memorie molto veloci presenti nel nostro processore. Il processore 8086 ne possiede 14, ma noi useremo i registri chiamati “Registri

general purporse”: AX, BX, CX, DX

Questi registri sono suddivisi a loro volta tra parte alta e parte bassa:

AX = AH (High, alta) e AL (Low, bassa)

BX = BH (High, alta) e BL (Low, bassa)

CX = CH (High, alta) e CL (Low, bassa)

DX = DH (High, alta) e DL (Low, bassa)

In ognuno di questi registri possiamo inserire fino a 16 bit (0 o 1)

Episodio 2 e 3:

Semplice tutorial su come scrivere in assembly

Episodio 4:

Struttura fissa dei programmi (i significati verranno spiegati successivamente):

.MODEL SMALL

.STACK 100H

.DATA

.CODE

Nella prima istruzione, .model, definisco un modello, in questo caso small, in assembly esistono diversi modelli (Tiny, small, large ecc). Small divide il nostro modello in due parti, una per i dati e l’altra per il codice

.stack 100h verrà spiegato poi

.data definisce l’area dedicata alle variabili, in assembly le variabili sono 2: DB(Byte, 1 byte) o DW(word, 2 byte). Per dichiarare una variabile si scrive inizialmente il nome della variabile, poi il tipo e infine il valore. ES:

VAR1 DB 2 (VAR1 variabile Byte dal valore 2) VAR2 DB 3 (VAR2 variabile Byte dal valore 3)

Risultato DB ? (Risultato variabile Byte dal valore non definito)

In .code verranno scritte tutte le istruzioni per il programma

.STARTUP serve per far capire al programma dove iniziare a eseguire il programma, END dove termina

il programma.

In assembly fare operazione direttamente tra variabili non è possibile, è fondamentale usare i registri

Fare direttamente ADD var1, var2 è errato, va spostata prima la variabile in un registro e poi va fatta la somma:

MOV AL, var1

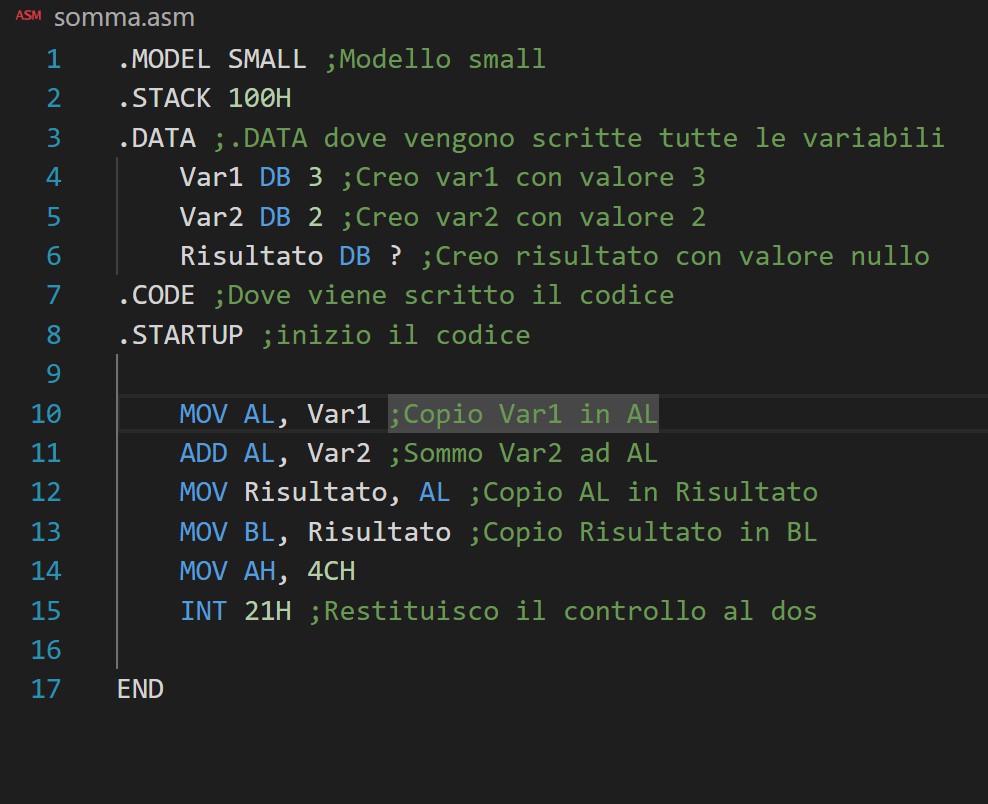
ADD AL, var2

A questo punto al conterrà il risultato dell’addizione. Per assegnare il risultato alla variabile Risultato

sarà necessario:

MOV Risultato, AL

Esempio di un programma per fare la somma:



N.B. L’istruzione MOV BL, Risultato serve solo per avere una visualizzazione del programma più leggibile

Come si può notare, son presenti MOV AH, 4CH e INT 21H.

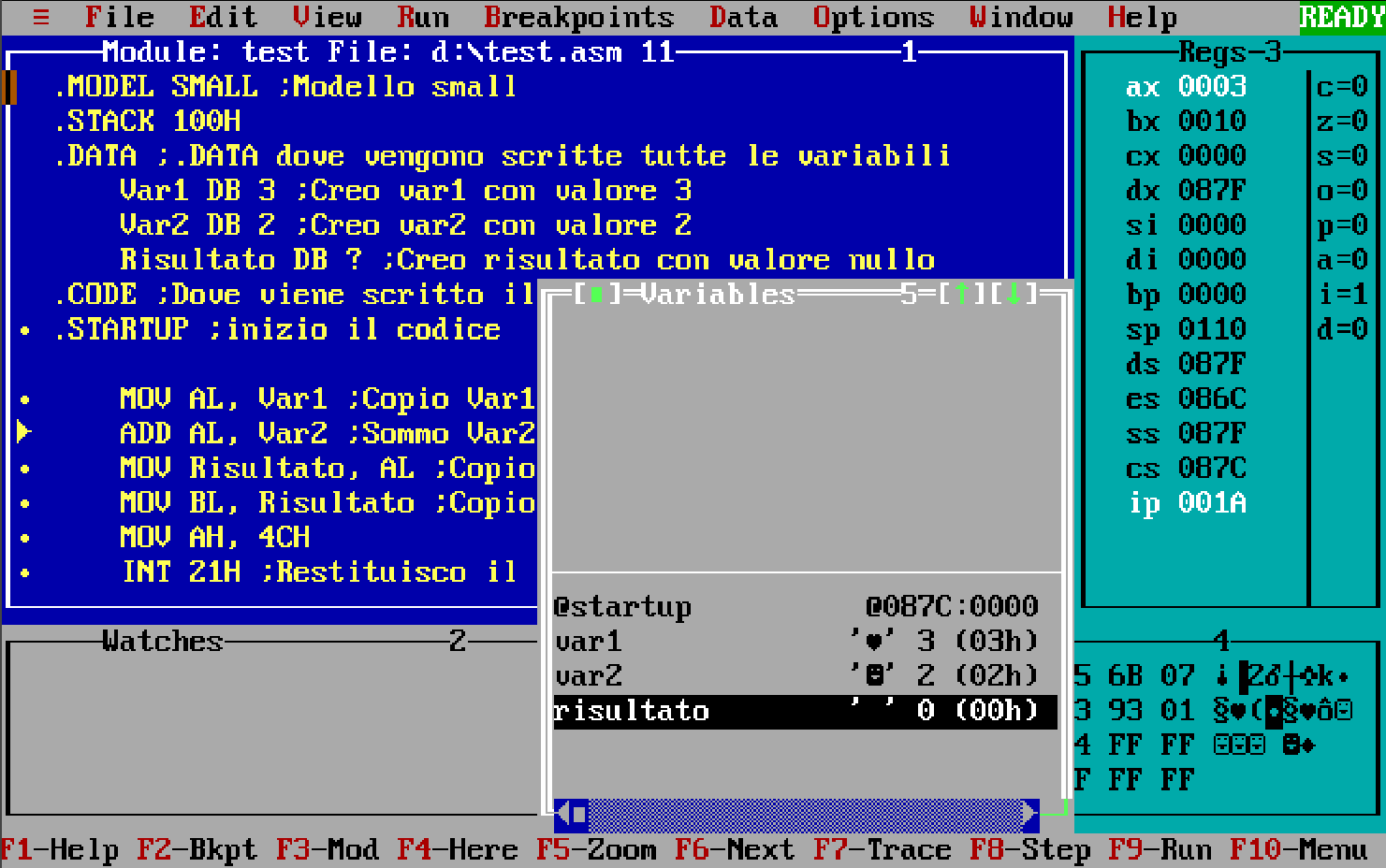
Queste due istruzioni servono a far capire al computer la fine del programma, quando il programma

viene eseguito prende “il controllo” del computer, con quelle due istruzioni restituiamo il controllo al dos.

Il nome del file non deve superare gli 8 caratteri.

Il programma scritto non presenta istruzioni di output, di conseguenza se eseguiamo direttamente il

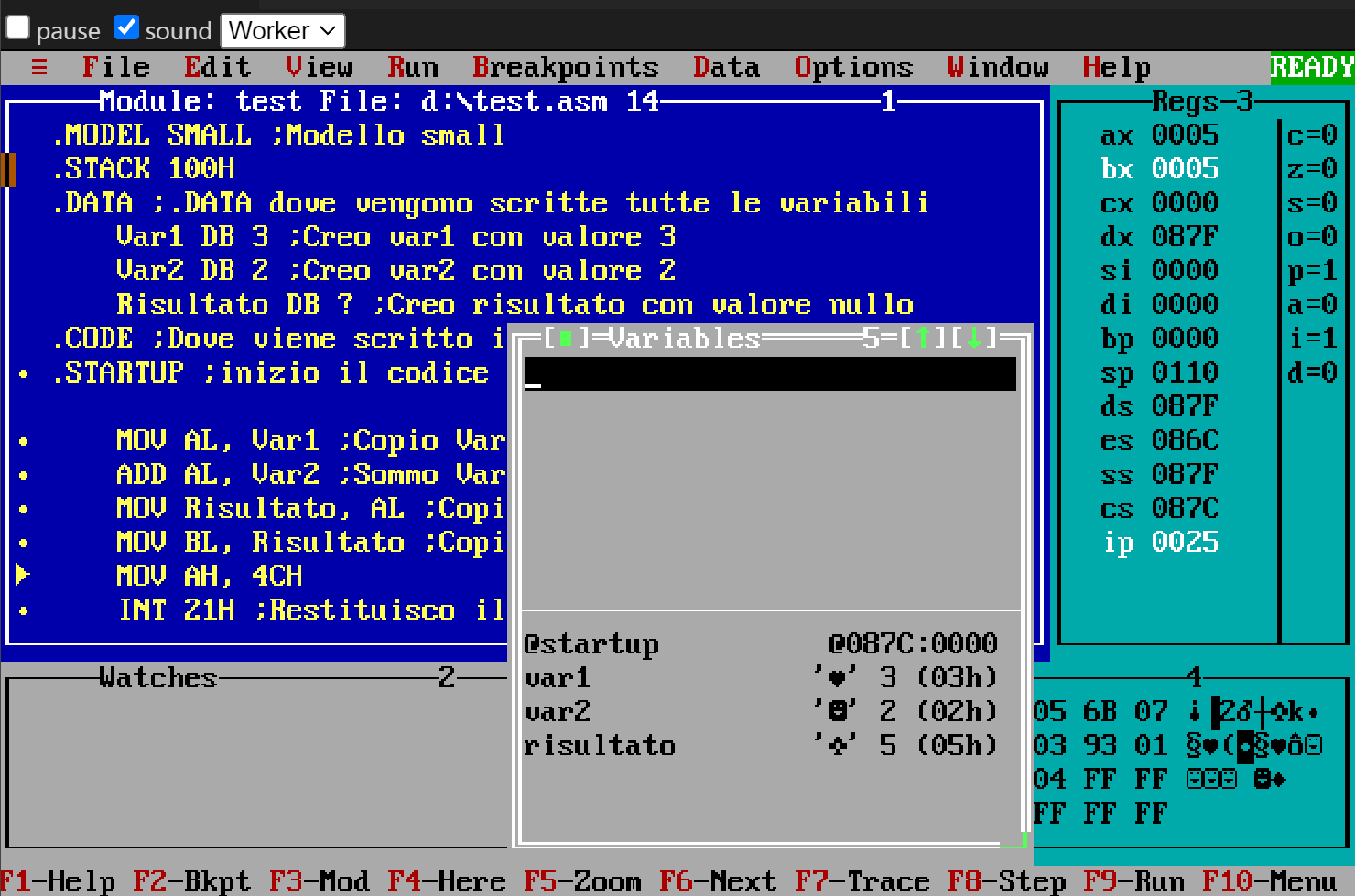
codice non noteremo nulla, è possibile però eseguire il codice passo passo tramite il debugger:



A destra si possono notare i registri, prediamo in esamina AX che è 0000, partendo da sinistra i primi

due zero formano AH (Alto) gli ultimi due formano AL (Basso).

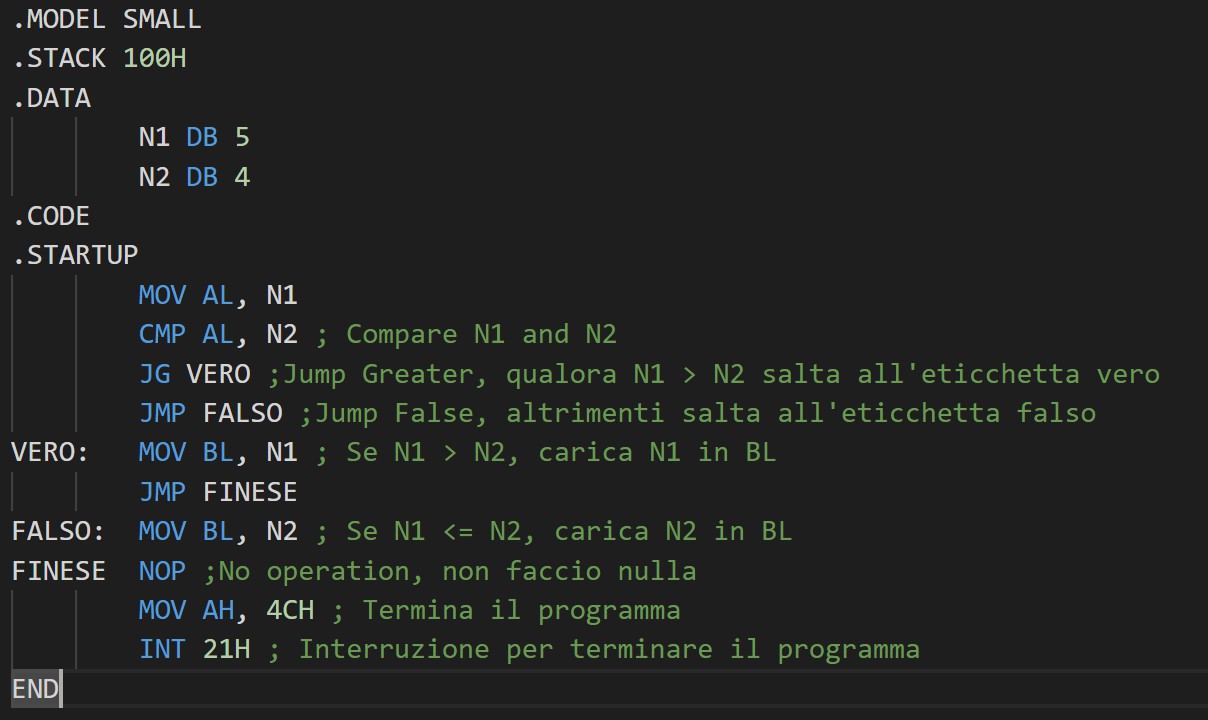
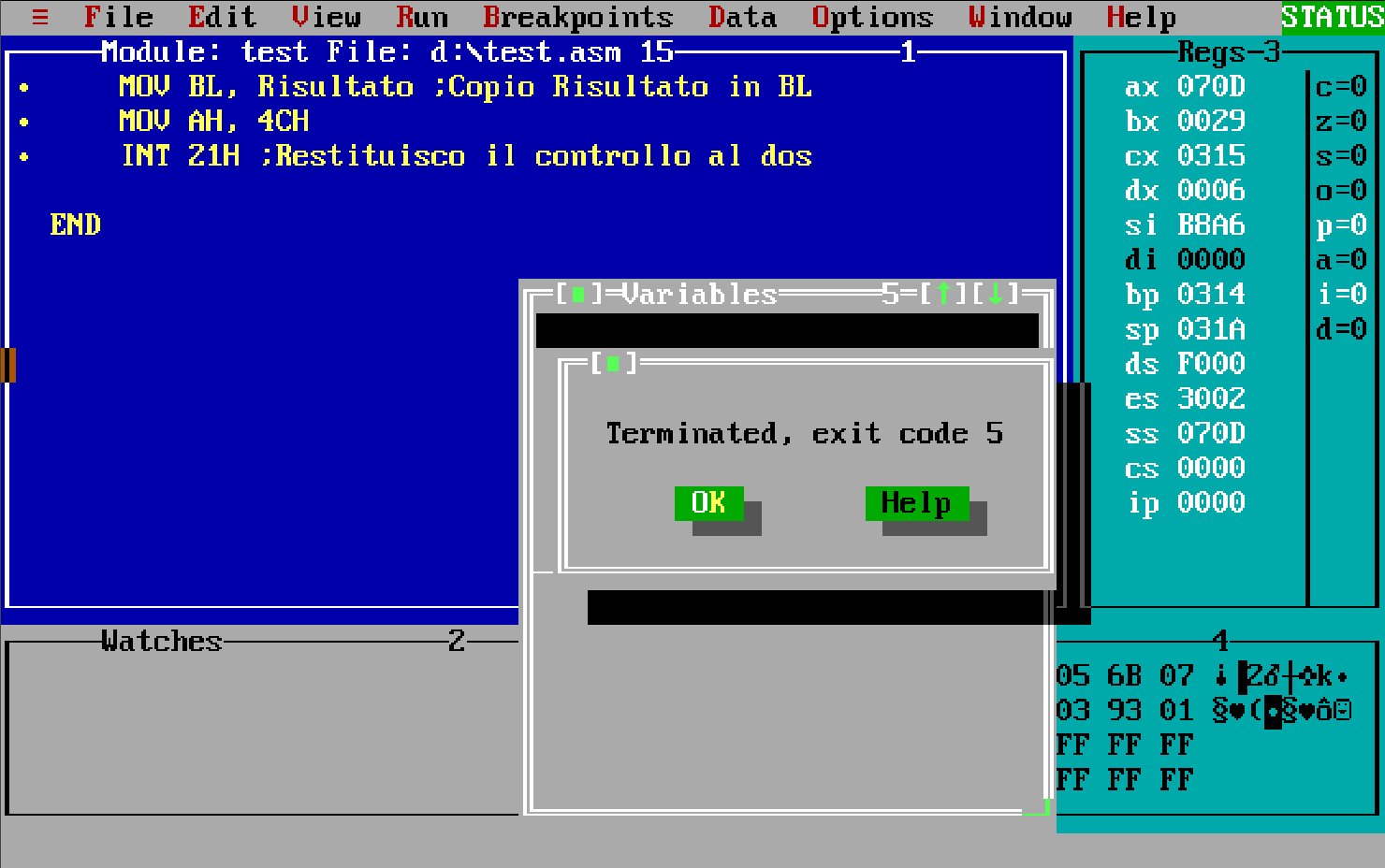
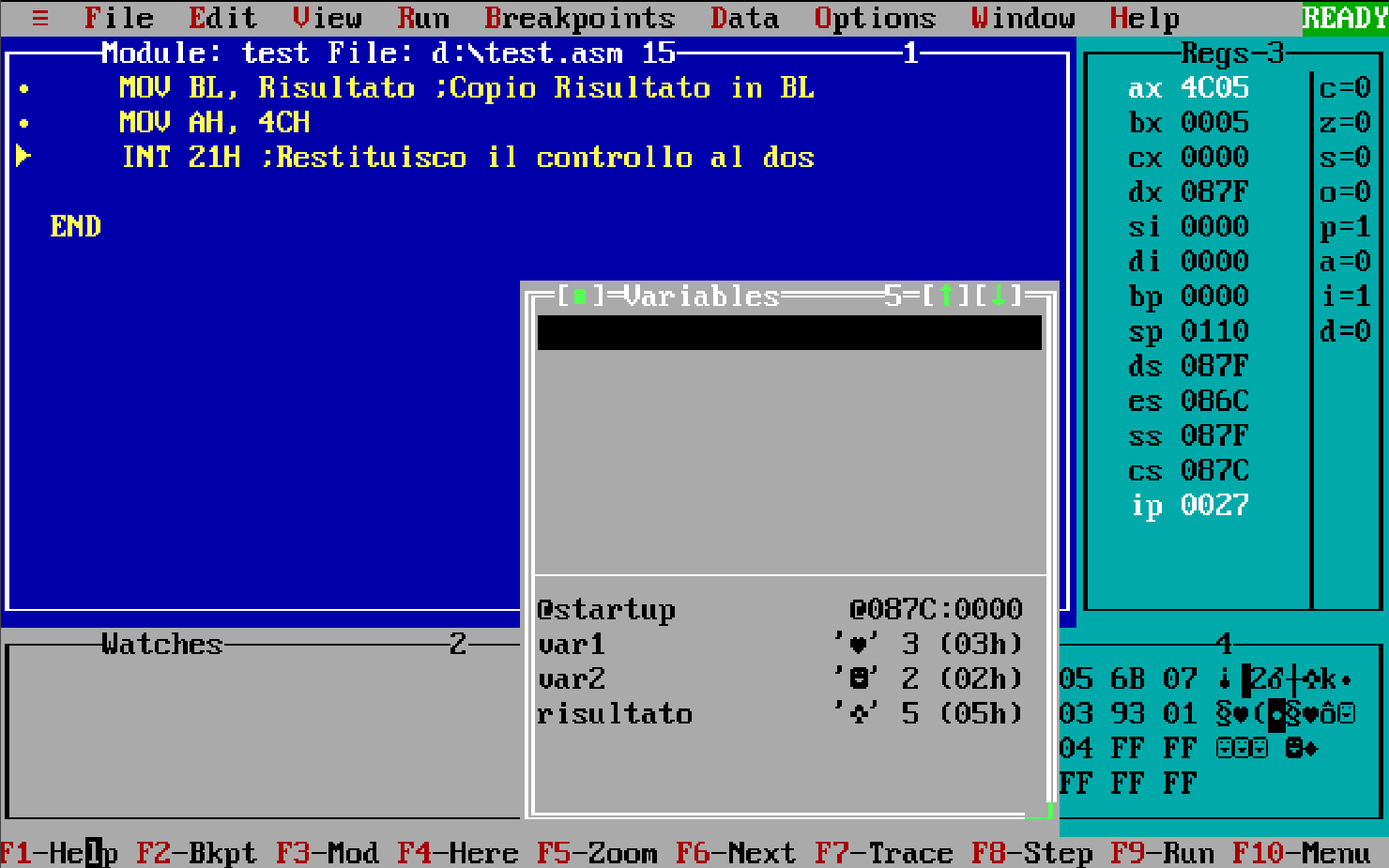
Come è possibile notare, dopo la mov AX è diventato 0003, questo perchè abbiamo spostato il valore di Var1 (3) in AL



Qua ha appena effettuato la somma, AX è diventato 0005, sommando il valore di Var2 (2)

Ora abbiamo effettuato la mov di AL in Risultato, di fatto se osserviamo il valore di risultato nella schermata “Variables” notiamo come risultato è passato da 0 a 5

Qui invece viene effettuata la mov di risultato in BL, quindi BX è diventato 0005



Fine del programma dopo aver restituito il controllo al dos.

Episodio 7: IF

Esempio di un programma con if:

Cosa succede? Innanzitutto creo due variabili Byte, n1 e n2, rispettivamente dal valore di 5 e 4. Successivamente carico il valore di n1 in al (come accennato in precedenza, non è possibile eseguire operazioni direttamente tra variabili) e effettuo una CMP (Comando usato per chiamare l’operazione di compare) su al (il cui valore in memoria è uguale a N1) e N2, qualora AL è maggiore di N2, richiama

l’operatore “jump greater”, ovvero effettua una jump all’etichetta VERO solo e soltanto se AL è maggiore di N2. Qualora la condizione accennata precedentemente non si verifica, effettua una normale jump all’etichetta FALSO.

Il resto del codice è di facile comprensione, nell’etichetta vero carica in BL il valore di N1, ed effettua

una jump a FINESE (per evitare di effettuare anche il codice dell’etichetta falso), nell’etichetta falso

viene caricato in BL il valore di n2. In finese(mancavano i due punti) viene eseguita una nop,

letteralmente no operation. Successivamente viene ridato il controllo al dos e viene terminato il programma.

APPROFONDIMENTO CMP:

Quando effettuiamo una CMP tra due valori, questa effettua una sottrazione virtuale tra questi due e aggiorna una flag interna che saranno poi lette dalle istruzioni JG, JE, JL, ecc..:

CMP operando1, operando2 ——> operando1 - operando2

FLAG

NOME

SIGNIFICATO DOPO CMP

ZF Zero Flag

Vale 1 se operando1 == operando 2 (quindi il risultato della sottrazione è 0)

SF Sign Flag Vale 1 se il risultato è negativo

CF Carry Flag Vale 1 se c’è un prestito nella sottrazione

(usato per confronti senza segno)

OF Overflow Flag Vale 1 se c’è un overflow

PF Parity Flag

Vale 1 se il numero di bit a 1 nel risultato è pari

AF Auxiliary Flag Usato raramente (es. aritmetica BCD)

Un’occhiata anche ai JUMP più usati:

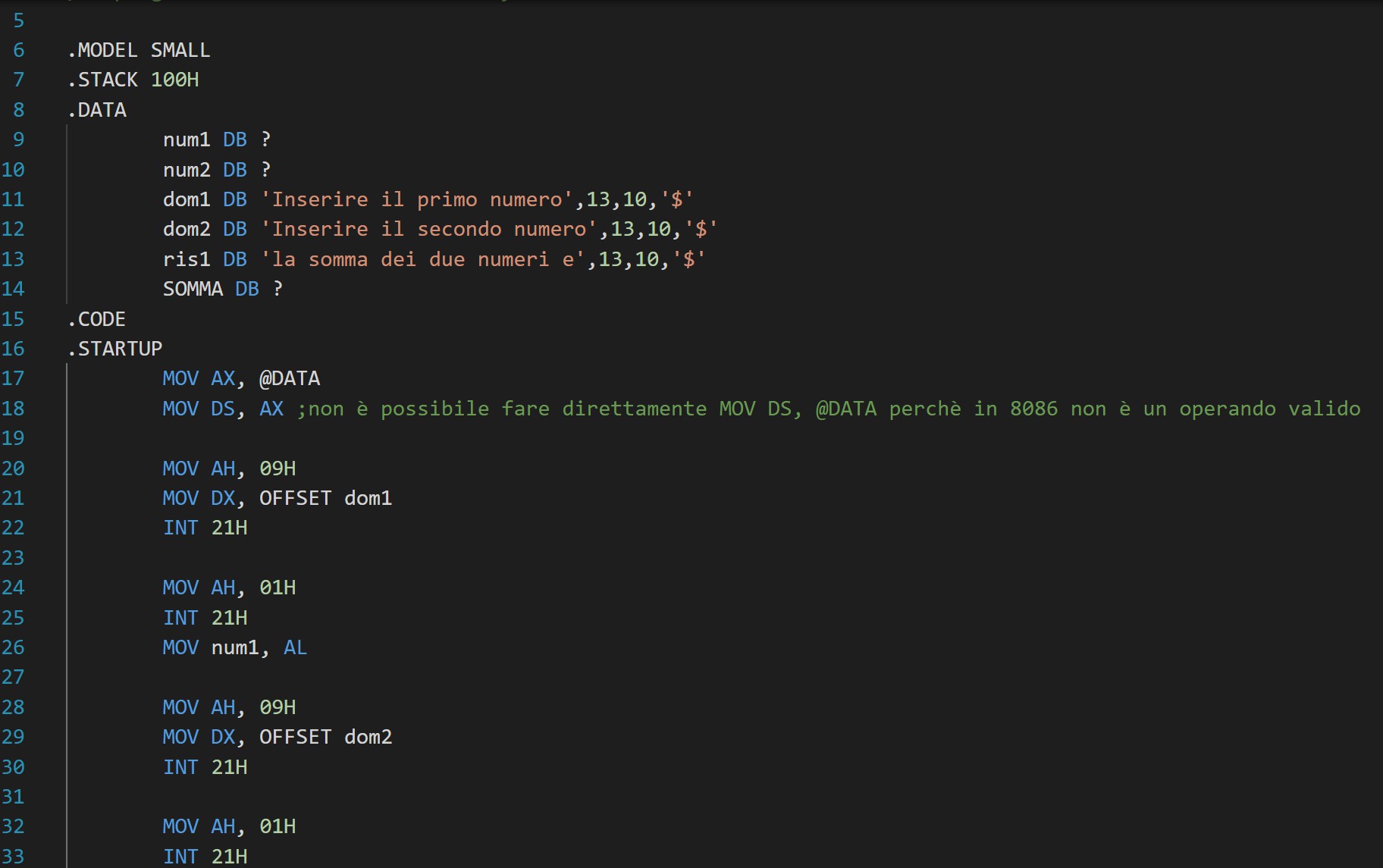
| **Istruzione** | **Significato** | **Quando si usa** | **Tipo di numeri** | **Flag coinvolti** |
| --- | --- | --- | --- | --- |
| JE / JZ | Jump if Equal / Zero | Se A == B | Tutti | ZF = 1 |
| JNE / JNZ | Jump if Not Equal | Se A ≠ B | Tutti | ZF = 0 |
| JG | Jump if Greater | Se A > B | **Con segno** | ZF = 0 e SF = OF |
| JGE | Jump if Greater/Equal | Se A ≥ B | **Con segno** | SF = OF |
| JL | Jump if Less | Se A < B | **Con segno** | SF ≠ OF |
| JLE | Jump if Less/Equal | Se A ≤ B | **Con segno** | ZF = 1 o SF ≠ OF |
| JA | Jump if Above | Se A > B | **Senza segno** | CF = 0 e ZF = 0 |
| JAE | Jump if Above/Equal | Se A ≥ B | **Senza segno** | CF = 0 |
| JB | Jump if Below | Se A < B | **Senza segno** | CF = 1 |
| JBE | Jump if Below/Equal | Se A ≤ B | **Senza segno** | CF = 1 o ZF = 1 |

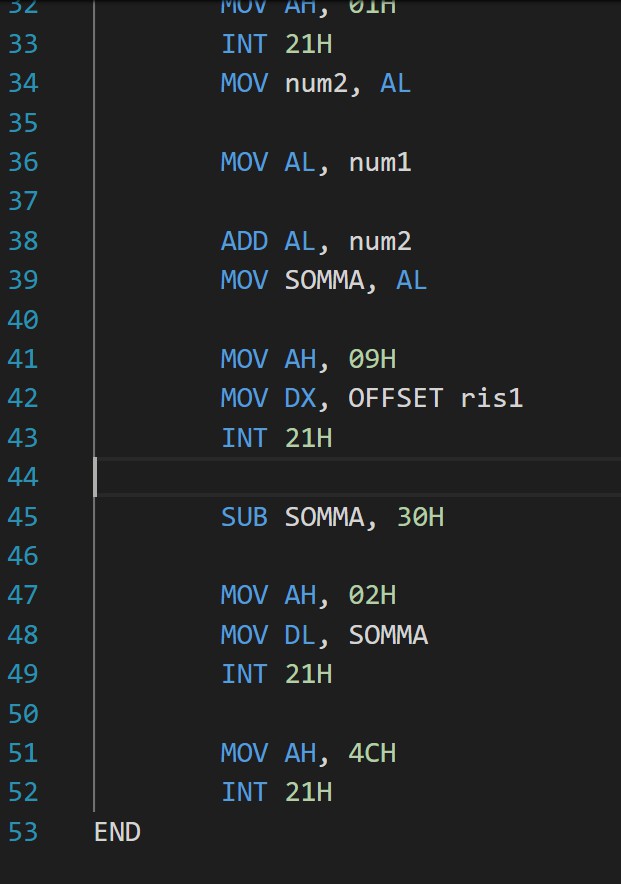
Le Jump più usate generalmente sono JE/JNE, JG/JL, JA/JB.

Episodio 9-10-11: Input e Output.

Per eseguire una stampa su schermo è necessario creare una variabile contenente al suo interno il messaggio. Alla fine della variabile troveremo ,13,10, ‘$’. Con ,13,10 non facciamo altro che dire al computer di andare a capo (\n), con ‘$’ serve come “tappo”, ovvero diciamo al computer dove finisce

la stringa e quando smettere di stampare.

Quando si effettua un input/output è necessario spostare tutti i dati del segmento .data in un registro chiamato DS (data segment).



Analizziamo quindi cosa abbiamo scritto nel codice

Per scrivere delle istruzioni di output bisogna:

Aprire con MOV AH, 09H scrivere tutte le nostre istruzioni

chiudere con INT 21H, che andrà quindi ad eseguire tutto il blocco di istruzioni. 09H è il codice che

permette di visualizzare una variabile stringa contenuto in DX. Quindi, all’interno del blocco di istruzioni, è necessario spostare la variabile stringa in DX:

MOV DX, OFFSET DOM1

Offset non fa altro che indicare l’indirizzo dove si trova DOM1, questa cosa va fatta solo nelle stringhe.

Per l’input invece bisogna:

aprire con MOV AH, 01H (richiedo in input un carattere che andrà in AL)

scrivere INT 21H per eseguire questa istruzione

spostare ciò che contiene AL nella variabile da noi desiderata: MOV num1, AL.

A questo punto effettuo la somma utilizzando gli stessi criteri accennati precedentemente.

Per visualizzare una variabile in assembly, a differenza delle stringhe, usiamo il codice 02H invece di 09H:

MOV AH, 02H (vado a visualizzare il contenuto di DL)

MOV DL, SOMMA (sposto somma in DL)

INT 21H (eseguo il blocco di istruzioni)

N.B. In questo caso non è necessario specificare offset dinanzi la variabile, in quanto non di tipo stringa

La stringa SUB SOMMA, 30H ci permette di visualizzare il risultato, senza questa subtract otterremmo

in output il valore ASCII corrispondente al codice della somma. (Vedere episodio 11 minuto 5:50 per

delucidazioni

**Approfondimenti: Funzioni principali di INT 21H**

INT 21h è l’interrupt del DOS che fornisce vari servizi (stampa a video, lettura da tastiera, gestione file, ecc.), mentre AH contiene il numero della funzione specifica di DOS che vuoi chiamare. In sostanza, quando si muove in AH un numero specifico (01H, 09H, 4CH, ecc) si chiamano delle funzioni specifiche dell’interrupt INT 21H. Ad ogni fine di queste funzioni, troviamo INT 21H, quando si effettua questa operazione si effettua una “chiamata” ad un servizio che sta già scritto nel DOS o nel BIOS. Il programma “fa una telefonata” al sistema operativo, il sistema fa il lavoro richiesto e poi ritorna al tuo codice, subito dopo la INT.

Non esiste solo INT 21H, ma è generalmente quello più usato:

* INT 21h → servizi del DOS (gestione file, input/output, stringhe, data/ora, ecc.).
* INT 20h → terminazione programma (vecchissimo, sostituito da INT 21h/AH=4Ch).
* INT 10h → servizi video (BIOS video: cambio modalità video, scrittura caratteri sullo schermo, spostamento cursore).
* INT 16h → servizi tastiera (BIOS keyboard: leggi tasti senza eco, controlla se un tasto è premuto).
* INT 13h → servizi dischi (BIOS disk: lettura/scrittura settori).
* INT 12h → restituisce la quantità di memoria convenzionale installata.
* INT 1Ah → servizi tempo e data dal BIOS (tick del timer di sistema).

Le funzioni principali:

| **AH** | **Funzione** | **Descrizione** | **Input** | **Output** |
| --- | --- | --- | --- | --- |
| **01h** | Input char con eco | Legge un carattere da tastiera e lo mostra |  | AL = carattere ASCII |
| **02h** | Output char | Stampa un carattere su schermo | DL = carattere ASCII |  |
| **07h** | Input char senza eco (bloccante) | Legge un carattere, non lo mostra |  | AL = carattere |
| **08h** | Input char senza eco (controllo) | Come 07h, ma segnala se tasto pronto |  | AL = carattere |
| **09h** | Output stringa | Stampa stringa terminata da $ | DX = offset stringa |  |
| **0Ah** | Input stringa bufferizzata | Legge linea di testo in un buffer | DX = offset buffer | Buffer contiene la stringa |
| **19h** | Drive corrente | Restituisce disco attivo |  | AL = drive (0=A, 1=B…) |
| **2Ah** | Data di sistema | Ottiene la data |  | CX=anno, DH=mese, DL=giorno |
| **2Ch** | Ora di sistema | Ottiene l’ora |  | CH=ora, CL=min, DH=sec, DL=centesimi |
| **3Ch** | Crea file | Crea un file vuoto | DS:DX=nome file | AX = handle file |
| **3Dh** | Apri file | Apre file esistente | DS:DX=nome file | AX = handle |
| **3Eh** | Chiudi file | Chiude un file aperto | BX = handle |  |
| **3Fh** | Leggi file | Legge da file | BX=handle, CX=bytes, DS:DX=buffer | AX=bytes letti |
| **40h** | Scrivi file | Scrive su file | BX=handle, CX=bytes, DS:DX=buffer | AX=bytes scritti |
| **41h** | Cancella file | Elimina un file | DS:DX=nome file | CF=errore |
| **4Ch** | Uscita programma | Termina programma e torna a DOS | AL=exit code |  |

**Episodio 12, 13, 14: Semplice esercitazione su trovare il maggiore tra tre numeri**

**Episodio 15, 16, 17: Esercitazione trovare caratteri Maiuscoli e Minuscoli**

**Episodio 18 - 19: Cicli**

Per effettuare dei cicli in Assembler 8086 si usano etichette e il comando CMP, per un esempio controllare il programma chiamato cicli.asm

**Episodio 20 - 21: Divisioni**

Con questa tecnica, sarà possibile rappresentare numeri superiori al 9, fino al 99.

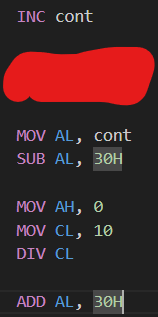
Per fare la divisione abbiamo bisogno che il numero da dividere sia in AX (DW, 2 Byte) , successivamente va messo in CL (1 Byte) il divisore (in questo caso, sarà 10), facciamo un esempio:

Immaginiamo di avere in AX 25 e vogliamo mostrarlo su schermo, a questo punto dividiamo il numero per 10 e avremo 2 col resto di 5.

Il 2 finirà in AL

Il 5 finirà in AH

È facile quindi capire che per mostrare il numero “25”, basterà mostrare a schermo prima AL e poi AH



Analizziamo cosa succede in questo blocco di codice:

* INC incrementa count (come se fosse count++)
* MOV AL, count copia il valore di count in AL
* SUB AL, 30H non fa altro che togliere 30H al valore di count. Questo accade perché in AL c’è bisogno del valore singolo e non il valore ASCII, essendo count inizializzato con 30H (quindi 0 carattere) l’incremento viene effettuato secondo la tabella ASCII, quindi bisogna effettuare una SUB 30H per riportare un valore normale. Il contatore poteva anche esser inizializzato direttamente con 0 numero e non ‘0’ carattere e avrebbe risparmiato questa stringa
* MOV AH, 0, come accennato prima la divisione verrà effettua su tutto AX e non sul singolo valore AL:

Il registro AX è formato da due parti AH e AL, in totale sono 4 valori (1234, di cui 12 AH e 34 AL), quando abbiam fatto la mov in AL, abbiamo copiato il valore nella parte bassa di AX, ma la parte alta è intoccata ed è necessario resettare anche quella per evitare che il risultato della divisione sia falsato

* MOV CL, 10, in CL troviamo il divisione della DIV
* DIV CL, in automatico il programma divide tutto quello che trova in AX per CL e posiziona automaticamente i resti. In assembly DIV 10 non è un’operazione valida
* ADD AL, 30H serve per trasformare nuovamente AL in carattere e per mostrarlo correttamente

**Episodio 22: Nulla riguardante la programmazione**

**Episodio 23: Visualizzare una variabile da 0 a 255**

Per rappresentare numeri dal valore superiore al 99 fino a 255, possiamo utilizzare la stessa tecnica vista negli episodi 20-21, con la differenza di dividere prima il numero per 100, mostrare il risultato e dividere nuovamente il resto per 10. Quindi immaginiamo di avere il numero 255:

Divido prima 255 per 100 che fa 2 con resto di 55, a questo punto visualizzo il due

Successivamente divido 55 per 10 che fa 5 col resto di 5, mi basterà visualizzare prima il quoziente e poi il resto

**Episodio 24: Push, Pop e stack**

Le operazioni Push e Pop operano direttamente sullo stack, ma cos’è di preciso uno stack?

Lo stack è una zona di memoria usata come una pila (stack in inglese significa pila/mazzo di piatti). Funziona con la regola LIFO = Last In, First Out (l’ultimo che entra è il primo che esce).

Quando usi l’istruzione PUSH valore, quel valore viene messo nello stack (in memoria).

Quando usi POP registro, viene tirato fuori l’ultimo valore inserito.

Lo stack serve anche quando chiami procedure o funzioni:

con CALL l’indirizzo di ritorno viene salvato nello stack,

con RET quel valore viene preso dallo stack per tornare al punto giusto.

Lo stack è gestito dai registri SS:SP (Stack Segment : Stack Pointer).

SS → dice in quale segmento di memoria si trova lo stack.

SP → indica la posizione corrente (punta all’elemento in cima alla pila).

Ogni PUSH fa scendere SP, ogni POP lo fa salire.

Lo stack viene definito da noi all’inizio di ogni programma con .STACK:

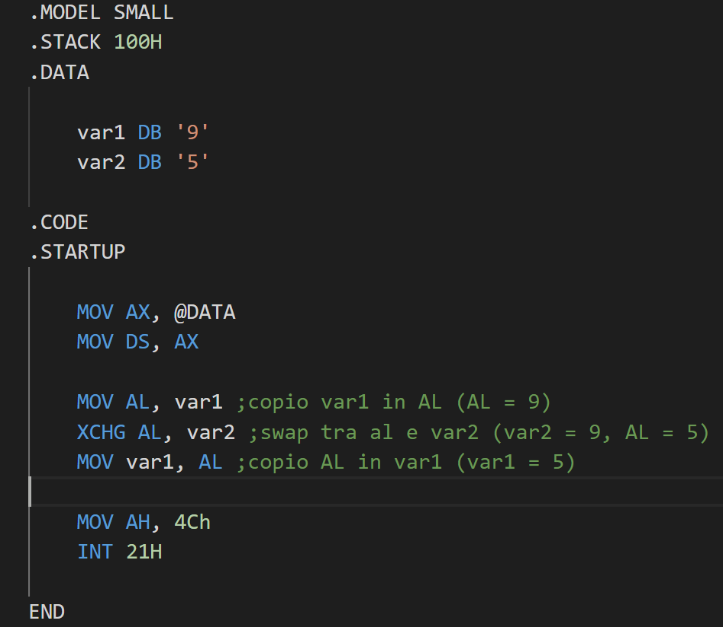
All’inizio di ogni programma, dopo .MODEL SMALL, inseriamo anche il comando .STACK 100h, questa istruzione serve a riservare memoria per lo stack del programma. .STACK indica al compilatore/linker che deve allocare una sezione di stack segment. 100h (cioè 256 byte) è la dimensione dello stack riservato.

Il valore 100h indica proprio 256 ma in esadecimale

**Episodio 25: XCHG**

La funzione XCHG fa una cosa molto semplice, effettua uno swap tra un registro e una variabile o tra due registri (ricordiamo che non è possibile effettuare direttamente operazioni tra due variabili), normalmente sarebbe necessario l’uso di due variabili + una temporanea, ma questa funzione permette di farlo in modo molto più comodo.

ES:



**Episodio 26-27-28: Array**

Prima di iniziare con la rappresentazione degli array bisogna capire la differenza tra SI (Source Index) e DI (Destination index):

In 8086 i registri **SI (Source Index)** e **DI (Destination Index)** sono due registri da 16 bit usati soprattutto per gestire operazioni con stringhe e array.

**🔹 Differenza principale**

* **SI = Source Index (indice sorgente)**  
  → Punta alla **zona di memoria da leggere** (origine dei dati).  
  → Usato implicitamente in istruzioni come LODS, MOVS, CMPS.  
  → Lavora sempre insieme al segmento **DS** (Data Segment).
* **DI = Destination Index (indice destinazione)**  
  → Punta alla **zona di memoria dove scrivere/confrontare** (destinazione dei dati).  
  → Usato implicitamente in istruzioni come STOS, MOVS, CMPS, SCAS.  
  → Lavora sempre insieme al segmento **ES** (Extra Segment).

**Come si dichiara un vettore in ASM 8086?**

Come le normali variabili, l’array in assembly 8086 si dichiara nel segmento .DATA:

* <nome vettore> <tipo> <grandezza vettore> DUP(<valore di ogni cella del vettore>)

In questo modo, dichiareremo un vettore con tutte le celle inizializzate con gli stessi valori, è possibile anche (esattamente come nel c) inizializzare il vettore specificando i valori di ogni cella:

* <nome vettore> <tipo> 10, 20, 30, 40, 50

E, come abbiamo visto in precedenza, è possibile anche avere vettori di tipo stringa, quando effettuavamo un output di una stringa e dichiaravamo la variabile nel segmento .DATA, creavamo un array di caratteri

* vet3 DB 'Hello World’, ‘$’
* vet3 DB 'HELLO$', 0

**Come vengono usati i vettori in ASM 8086?**

Quando dobbiamo riferirci a una cella specifica si usa un indice ma, a differenza di altri linguaggi, si usano i due registri accennati a inizio capitolo: SI e DI

Qualora si gestisca in modo manuale gli array in ASM, l’utilizzo di uno o l’altro è al quanto indifferente, mentre se si utilizzano istruzioni di stringa (MOVSB, CMPSB, SCASB, STOSB, ecc) la differenza diventa fondamentale:

* **SI (Source Index)** → punta alla **sorgente** (DS:SI).
* **DI (Destination Index)** → punta alla **destinazione** (ES:DI).

Esempio con copia stringa:

CLD ; clear direction flag (incrementa indici)

MOV SI, OFFSET sorgente

MOV DI, OFFSET destinazione

MOV CX, lunghezza

REP MOVSB ; copia CX byte da DS:SI a ES:DI

In questo caso non puoi scambiarli: SI = origine, DI = destinazione.

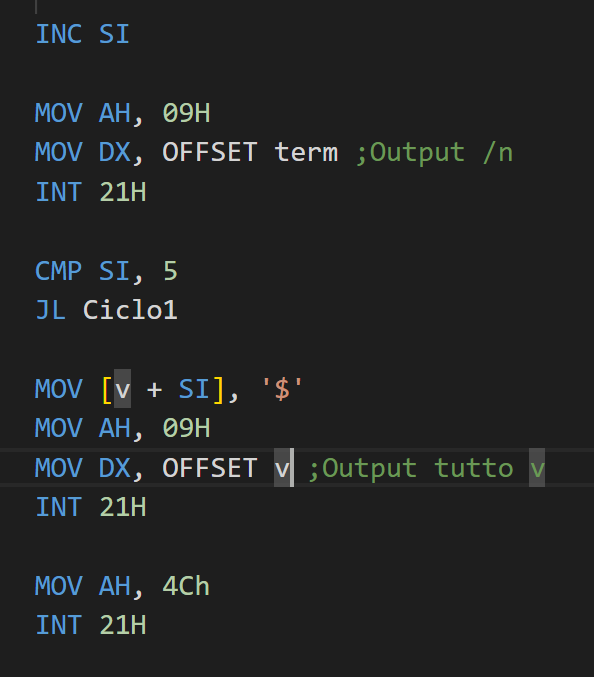
**Regola pratica**

* Se usi il tuo array **manualmente** (con MOV, ADD, ecc.) → puoi usare **SI o DI liberamente**.
* Se usi le **istruzioni stringa** → devi usare **SI per leggere** e **DI per scrivere**.

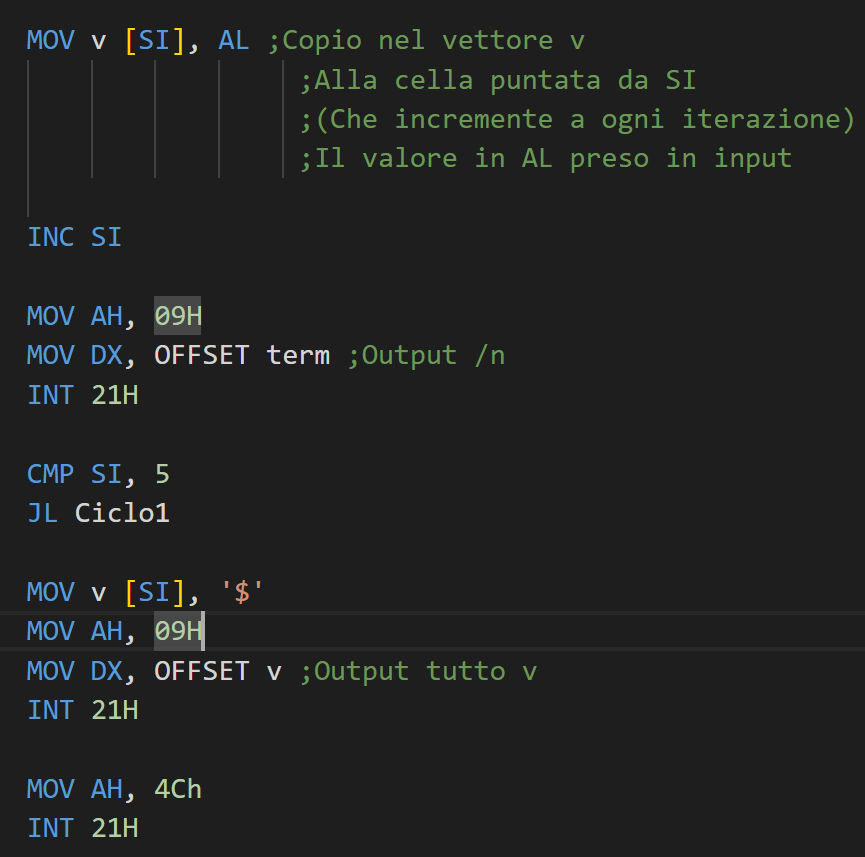
**Vediamo in pratica come usare gli array:**

Immagine che contiene testo, schermata, software, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.



Vediamo anche la seconda forma per usare gli array in ASM:



Ciò che cambia è il modo in cui ci riferiamo all’indice del vettore, entrambe le forme sono equivalenti tra di loro ma c’è una leggera differenza:

Nonostante la seconda forma sia meno macchinosa e più comoda da scrivere, non tutti i compilatori la accettano ed è generalmente meglio riferirsi all’array con [<nome vettore> + SI/DI]

N.B. Approfondimenti al minuto 2:10 del tutorial 28

**Cosa succede precisamente in questo programma?**

Il programma fa una cosa semplicissima, chiede 5 caratteri all’utente e poi li stampa tutti a schermo;

* Inizialmente mi assicuro che SI sia azzerato, quindi effettuo una mov per copiare il valore 0
* Poi faccio un output di dom1
* Richiedo in input di un carattere da tastiera e copio quest’ultimo nella cella del vettore v puntata da SI
* Successivamente incremento SI per puntare alla cella successiva
* Ripeto il ciclo fino a quando SI non diventa 5
* Alla fine del vettore aggiungo anche il terminatore ‘$’ (ricordiamo che inserire questo terminatore alla fine delle stringhe è fondamentale per evitare comportamenti indesiderati da parte del compilatore)

**Episodio 29-30: Input di stringa fino a invio (esercitazione vedere yt)**

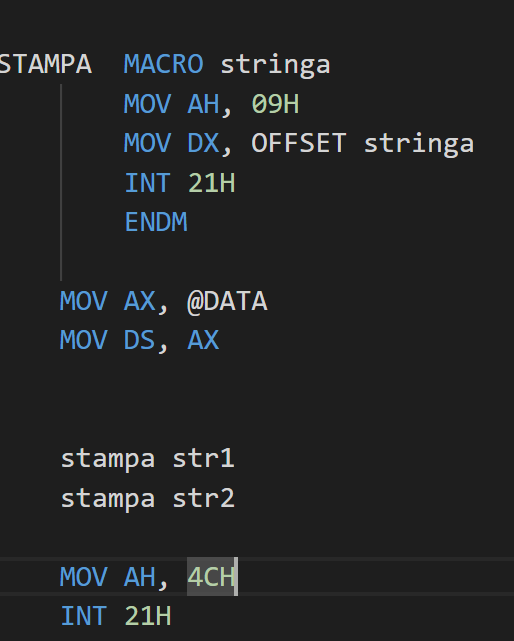
**Episodio 31: Macro**

Le macro sono istruzioni che contengono al loro interno altre istruzioni, in modo da poterle scrivere una sola volta senza riscriverle da capo ogni volta (ES: print, lettura, ecc).

Le macro devono essere dichiarate all’interno del .CODE.

Per dichiararne una bisogna prima assegnarle un nome (simile a un’etichetta di un if, ma senza i due punti). Dopo il nome si scrive la parola chiave MACRO (obbligatoriamente di fianco), seguita dalle istruzioni che devono far parte della macro. La definizione si conclude con la parola ENDM (end macro). Le istruzioni contenute nella macro non vengono eseguite subito: verranno eseguite solo quando la macro sarà richiamata nel programma.

Le macro hanno anche la possibilità di passare parametri, semplicemente mettendo uno spazio dopo l’istruzione MACRO:



In questo esempio vediamo perfettamente come le macro vanno usate, noi scriviamo le istruzioni con l’etichetta STAMPA, poi passiamo il parametro della stringa che vogliamo stampare e quest’ultima prenderà il posto di stringa

**Episodio 32-33: Esercitazione**

v[SI – 1] è una notazione valida

**Episodio 34-35: BubbleSort**

Per gli output è possibile scrivere (dopo MOV AH, 09H) LEA DX, MSG invece di MOV DX, OFFSET MSG. Lea significa Load Effected Addres, quindi carica l’indirizzo di inizio

**Episodio 36: Ricerca Valore nel vettore**

**Episodio 37, 38, 39, 40: Acquisire numeri di due cifre e sommarli**

**Episodio 41, 42, 43: Ricerca binaria**

**Episodio 44: Puntatori**

I puntatori permetto di ‘puntare’ una variabile e poter operare su di essa, questi sono molto utili quando useremo le procedure. Per usare i puntatori si usano i registri DI e SI. Come in C, per caricare l’indirizzo di memoria di una variabile in DI si usa LEA DI, var (mov DI, OFFSET var) mentre, per copiare il valore all’interno della variabile usando i puntatori si usa MOV BYTE PTR [DI], 9/ MOV WORD PTR [DI], 8 (byte o word in questo caso serve ad indicare il tipo della variabile)

**Episodio 45-46: Operatori Logici + numeri negativi**

Gli **operatori logici** AND e OR in Assembly (e in generale nei linguaggi a basso livello) lavorano **bit per bit** sui numeri.  
Significa che prendono ciascun bit del primo numero e lo confrontano con il bit corrispondente del secondo numero, applicando la logica booleana.

**Esempio con due numeri: 5 e 3**

1. Prima si convertono in binario:

* 5 = **0101**
* 3 = **0011**

**Operazione AND**

Confrontiamo i bit uno per uno. La regola è:

* 1 AND 1 = 1
* 1 AND 0 = 0
* 0 AND 1 = 0
* 0 AND 0 = 0

Quindi:

0101 (5)

0011 (3)

-------

0001 (1)

Per la OR e la NOT succede la stessa cosa semplicemente utilizzando la loro porta logica

Per rappresentare un numero negativo, è necessario effettuare il complemento A2:

* Scrivi il numero in binario (in base al numero di bit che usi, ad esempio 8 bit). Esempio: +5 → 00000101
* Inverti tutti i bit (complemento a 1). 00000101 → 11111010
* Aggiungi 1 al risultato. 11111010 + 1 = 11111011
* Questo è il valore di -5 in complemento a 2.

Per riconvertire il valore da negativo a positivo, baserà rieffettuare la not e riaggiungere 1 binario

(In assembly tutto ciò si traduce prima effettuando una NOT di un numero e poi incrementando quest’ultimo)

Per effettuare queste istruzioni in modo automatizzato è possibile usare l’istruzione NEG

**Episodio 47: Procedure (senza parametri)**

Le procedure vanno dichiarate tra il .CODE e il .STARTUP:

.CODE

proc1 PROC NEAR/FAR

;codice

RET

proc1 ENDP

.STARTUP

Queste vengon dichiarate in questo modo, com’è possibile notare c’è una differenza tra NEAR e FAR:

**Procedura NEAR**

* Una procedura **NEAR** si trova **nello stesso segmento di codice** della chiamata (CS non cambia).
* Quando fai CALL, viene salvato **solo l’offset** (IP) sullo stack.
* Al ritorno (RET), viene ripristinato solo l’IP.
* Vantaggio: più veloce e occupa meno memoria nello stack.

**Procedura FAR**

* Una procedura **FAR** si trova **in un altro segmento di codice** (potenzialmente diverso da CS).
* Quando fai CALL, vengono salvati sia **CS** (segmento) sia **IP** (offset) sullo stack.
* Al ritorno (RET), vengono ripristinati sia CS che IP.
* Vantaggio: puoi organizzare il programma su più segmenti di codice.
* Svantaggio: un po’ più lenta e lo stack si riempie di più.

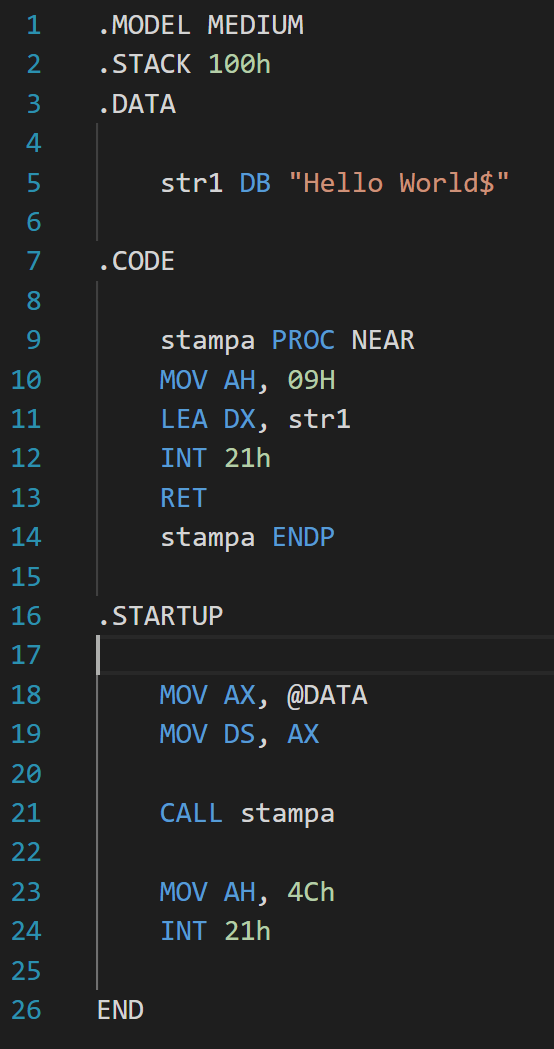
**Differenza riassunta:**

| **Tipo** | **Dove si trova la procedura** | **Cosa salva CALL** | **Ritorno con** | **Velocità** |
| --- | --- | --- | --- | --- |
| NEAR | Stesso segmento di codice | solo IP | RET | più veloce |
| FAR | Altro segmento di codice | CS + IP | RET (a 32 bit) | più lenta |

Nei programmi **.MODEL SMALL**, quasi sempre usi **NEAR**, perché codice e dati stanno nello stesso segmento.  
Nei programmi più grandi (**.MODEL MEDIUM, LARGE…**) si usano anche le **FAR**.

CS sta per Code Segment, è un registro di segmento dell’8086 (16 bit) che contiene l’indirizzo di base del segmento di codice, cioè la zona di memoria dove risiede il programma da eseguire.

In breve:  
🔹 CS = “dove inizia il mio codice”  
🔹 IP = “a che punto del codice sono arrivato”



Questo è un esempio di programma che utilizza una **procedura senza parametri**.

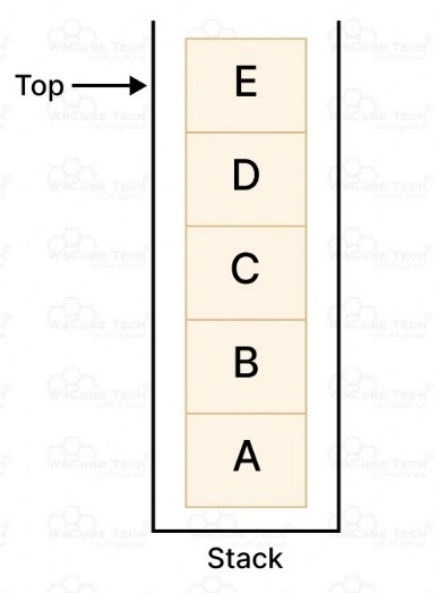
Quando viene eseguita l’istruzione **CALL**, il processore salva nello **stack** l’indirizzo della **prossima istruzione** del programma principale (cioè quella che segue subito dopo la CALL, in questo caso la riga 23).

All’interno della procedura, quando incontriamo l’istruzione **RET**, la CPU legge l’indirizzo salvato in cima allo stack e riprende l’esecuzione da lì.

Se nello stack non vengono fatte altre operazioni, tutto funziona senza problemi.  
Tuttavia, è importante ricordare che se durante la procedura si alterano in modo scorretto i dati nello stack, il ritorno (RET) potrebbe riprendere da un indirizzo sbagliato e causare errori nel programma.

**Episodio 48-49-50: Procedure (con parametri e variabili locali)**

Prima di vedere come si effettua il passaggio di parametri nelle procedure, dobbiamo prima capire bene il funzionamento dello stack, cos’è il Base Pointer (BP) e lo Stack Pointer (SP)



Ora, immaginiamo che questo sia un nostro stack, il Base Pointer punta alla base del nostro stack quindi, in questo caso, ad A. Lo stack pointer invece punta all’ultimo elemento inserito nel nostro stack quindi, in questo caso, ad E.

Gli indirizzi dello stack funzionano in modo inversamente proporzionale all’ordine dello stack stesso:

Immaginiamo di trovarci nella situazione in cui abbiamo fatto solo la push di A, quindi il momento in cui SP e BP sono coincidenti, quando effettueremo una nuova push il valore di SP decrementerà e lo farà per ogni valore che inseriamo nello stack, al contrario quando effettueremo una pop il valore di sp incrementerà.

Siccome per dichiarare variabili locali e fare un passaggio di parametri usiamo lo stack, è importante salvare e successivamente ripristinare i valori di SP e BP per evitare che, quando effettuiamo il comando RET nella procedura non verranno riscontrati problemi.