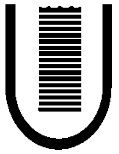
Assembly i80x86

Dr. Mauro Codella



Università degli Studi di Roma “Tor Vergata”

Overview

- Capire perché è utile imparare a programmare a basso livello;
- Conoscere le peculiarità del software che adotteremo per programmare;
- Acquisire i concetti base a monte della programmazione assembly;
- Acquisire una parte del set di istruzioni disponibile nella famiglia dei processori i80x86;
- Comprendere il funzionamento di piccoli programmi esistenti;
- Imparare a programmare, a basso livello, applicativi di piccola entità.

“Capire perché è utile imparare a programmare a basso livello”

Capire l'importanza della programmazione a basso livello

- Contro
  - Assenza quasi totale della gestione dei tipi di dato nei compilatori: è infatti il programmatore che si deve far carico del corretto utilizzo dei tipi;
  - Portabilità ridotta rispetto ai linguaggi più moderni;
  - Difficoltà nel comprendere programmi già scritti;
  - Difficoltà anche nello scrivere programmi.
  - Non è più utilizzato dalla maggior parte delle aziende che sviluppano software.

A proposito della difficoltà nello scrivere programmi in assembly...

Tratto direttamente dal sito di Wikipedia: Verso l'ottobre del 1980 la IBM stava cercando un sistema operativo per il suo nuovo prodotto, il PC IBM prossimo al lancio. Inizialmente si rivolse alla Digital Research di Gary Kildall, l'autore del CP/M che allora era lo standard per i microcomputer; ma l'affare, per ragioni mai del tutto chiarite, andò in fumo. Continuando la ricerca, bussarono anche alla porta della Microsoft di Bill Gates e Paul Allen che allora produceva quasi solo linguaggi (il Microsoft BASIC). I due, davanti all'occasione che si profilava loro, non esitarono a contattare la Seattle Computer Products che pochi mesi prima aveva scritto un clone a 16 bit del CP/M chiamato 86-DOS per i microcomputer che stava producendo, basati sull'8086 e sul bus S-100. Dopo una veloce revisione dei sorgenti, che consistevano in circa 4000 linee di codice assembler, il tutto fu mandato alla IBM per una valutazione. La IBM rimase soddisfatta e l'affare andò in porto. La Microsoft acquisì i diritti dell'86-DOS nel luglio 1981 e il mese dopo la prima versione di MS-DOS era sul mercato...

A proposito della difficoltà nello scrivere programmi in assembly...

...IBM però, avendola sottoposta ad un esteso controllo di qualità ed avendo trovato oltre 300 bug, ne riscrisse alcune parti: per questo motivo tale versione portò il nome di IBM PC-DOS 1.0 e fu licenziata congiuntamente da Microsoft e da IBM. Le versioni successive furono licenziate separatamente o da Microsoft (che le marcava come MS-DOS) o da IBM (con il nome di PC-DOS e in genere in coincidenza con l'uscita sul mercato di una nuova linea di personal computer).

Capire l'importanza della programmazione a basso livello

- Pro
  - Si percepisce come effettivamente si riescono a far svolgere delle attività ad un calcolatore;
  - Si ha la giusta percezione di cosa può e cosa non può fare un calcolatore;
  - Si possono ottimizzare particolari routine di un applicativo, specialmente se scritto in C o in C++;
  - Fornisce la possibilità di intuire e ricostruire il codice sorgente a monte di un eseguibile preesistente (Reverse Engineering).

“Conoscere le peculiarità del software che adotteremo per programmare”

Un po' di terminologia

- Emulatore: è un software che ricrea l'hardware di un sistema, solitamente differente da quello di cui si dispone, permettendo di far eseguire programmi di vario genere come se fossero eseguiti nell'hardware emulato.
- Simulatore: è un software che imita l'hardware di un sistema.
- La differenza tra i due concetti risiede nel fatto che mentre il primo ha come obiettivo primario il funzionamento del software, per il secondo tale obiettivo è secondario: l'attenzione è infatti principalmente rivolta alla rappresentazione degli aspetti architetturali e funzionali dell'hardware emulato.

Un po' di terminologia

- **Assembly**: (definizione tratta dal sito di Wikipedia) è il linguaggio di programmazione più vicino al linguaggio macchina vero e proprio. Infatti, esiste una corrispondenza pressoché biunivoca tra gli mnemonici del linguaggio assembly ed i corrispondenti codici macchina corrispondenti ai bitfields (campi di bit) che compongono le istruzioni direttamente eseguibili dal dispositivo elettronico (in genere una CPU) che si sta programmando.

Un po' di terminologia

- **Assembler**: (definizione tratta dal sito di Wikipedia) è un programma compilatore che si occupa di tradurre in linguaggio macchina (ossia una serie di bit 0 e 1 che costituiscono l'unico modo per comunicare con dispositivi elettronici), una serie di comandi scritti in linguaggio Assembly. Oltre a questo compito base, un Assembler si occupa spesso di ausiliare il programmatore, ad esempio consentendo l'utilizzo nel codice sorgente del programma di nomi mnemonici al posto di indirizzi esadecimali che costituiscono l'esatta collocazione di una variabile o di una porzione di programma nella memoria centrale del computer.

Conoscere gli strumenti

- Durante queste lezioni prenderemo in considerazione un emulatore di CPU i80x86;
- La scelta è ricaduta sull'emulatore emu8086, reperibile presso emu8086.com;
- Dispone di un editor di testi sensibile alla sintassi dell' assembly i80x86, di una calcolatrice, di un convertitore e altri strumenti utili ai nostri scopi.
- In realtà emu8086 è anche un assembler, ma questa funzionalità non la adopereremo per vari motivi.

Il perché della scelta

- Perché utilizzare un emulatore e non un assembler?
  - L'emulatore permette di analizzare le componenti salienti della CPU in modalità real-time: durante l'esecuzione delle istruzioni sono infatti visibili immediatamente le modifiche ai registri e ai flag;
  - Possiamo eseguire il nostro codice su un qualunque calcolatore in grado di eseguire l'emulatore;
  - Non vi è la necessità di ricompilare il programma ad ogni minima modifica al codice;
  - I vantaggi offerti dal compilatore (velocità ed ottimizzazione del codice) non sono pertinenti allo scopo di queste lezioni.

“Acquisire i concetti base a monte della programmazione assembly”

Rappresentazione in base 2 di un numero in base 10

- Vogliamo richiamare qui l'algoritmo per la rappresentazione in base 2 di un numero in base 10;
- Algoritmo di conversione:

I. scegliamo il numero in base 10 da convertire;

II. memorizziamo il numero in una variabile che chiameremo ***x***;

III. dividiamolo il valore in ***x*** per 2;

IV. riponiamo il resto della divisione in una pila ***P***;

V. se il risultato della divisione è uguale a 0 allora salta al passo VIII, altrimenti procedi con il passo successivo;

VI. riponiamo il risultato della divisione in ***x***;

VII. ritorniamo al passo III;

VIII. fine: nella pila troviamo il numero convertito in binario, con la cifra più significativa (MSB) sulla cima.

Rappresentazione in base 16 di un numero in base 10

- Vogliamo richiamare qui l'algoritmo per la rappresentazione in base 16 di un numero in base 10;
- Algoritmo di conversione:

I. scegliamo il numero in base 10 da convertire;

II. memorizziamo il numero in una variabile che chiameremo ***x***;

III. dividiamolo il valore in ***x*** per 16;

IV. riponiamo il resto della divisione in una pila ***P***;

V. se il risultato della divisione è uguale a 0 allora salta al passo VIII, altrimenti procedi con il passo successivo;

VI. riponiamo il risultato della divisione in ***x***; (*)

VII. ritorniamo al passo III;

VIII. fine: nella pila troviamo il numero convertito in esadecimale, con la cifra più significativa sulla cima.

Rappresentazione in base 16 di un numero in base 10

(*) incontriamo un problema: sapendo che, in generale, il resto di una divisione per ***d*** è pari ad un ***r*** nell'intervallo chiuso [0, ***d***-1], come ci comportiamo quando dividiamo per 16? Il resto in questo caso è nell'intervallo [0, 15], ma nella pila ***p*** siamo costretti ad inserire una sola cifra... cosa fare?

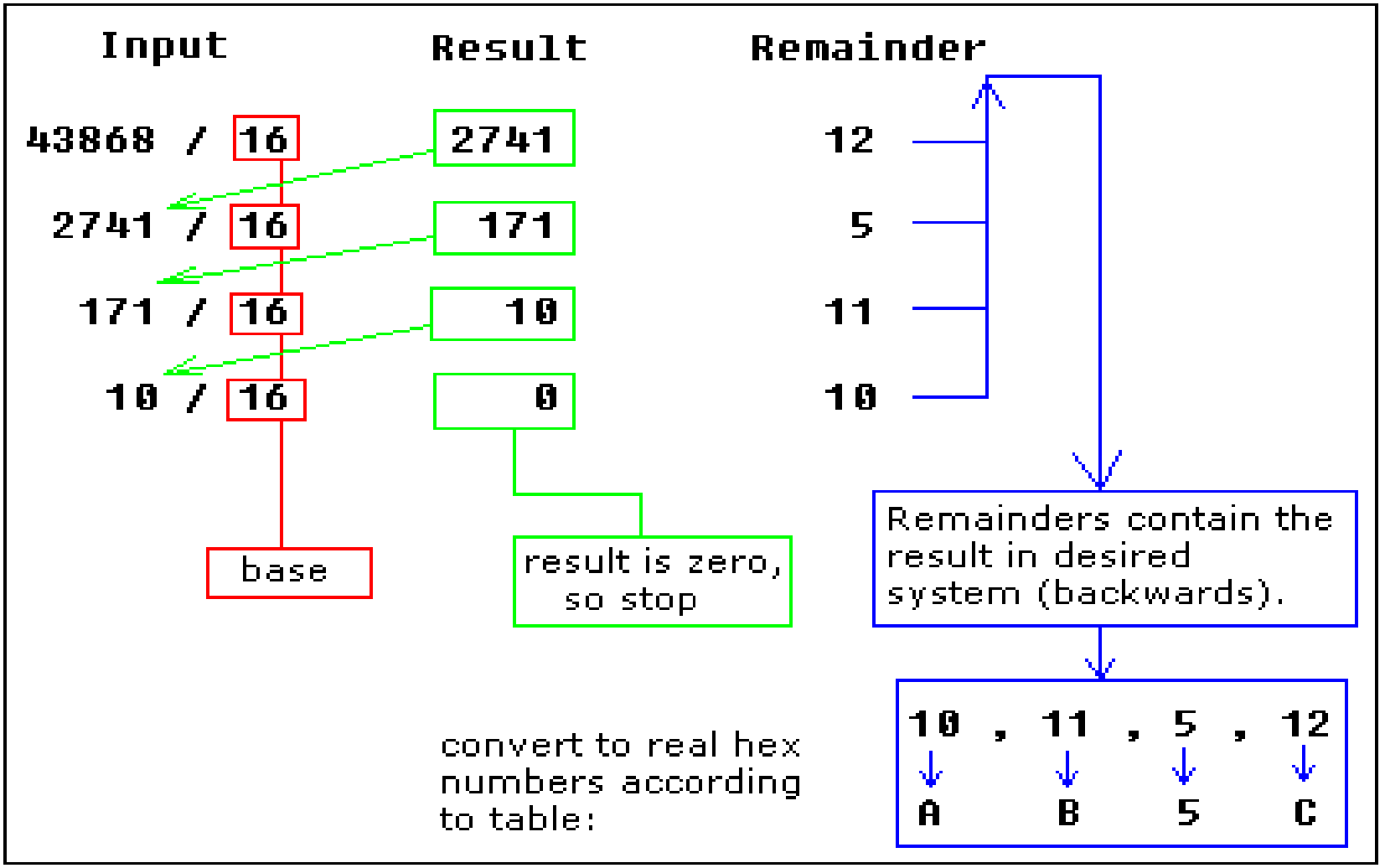
Rappresentazione in base 16 di un numero in base 10

(*) ... semplicemente associamo ai valori 10, 11, 12, 13, 14 e 15 dei simboli univoci che li rappresentano. In particolare si prende A per rappresentare 10, B per 11, C per 12, D per 13, E per 14 ed F per 15.

Tabella riassuntiva delle cifre esadecimali

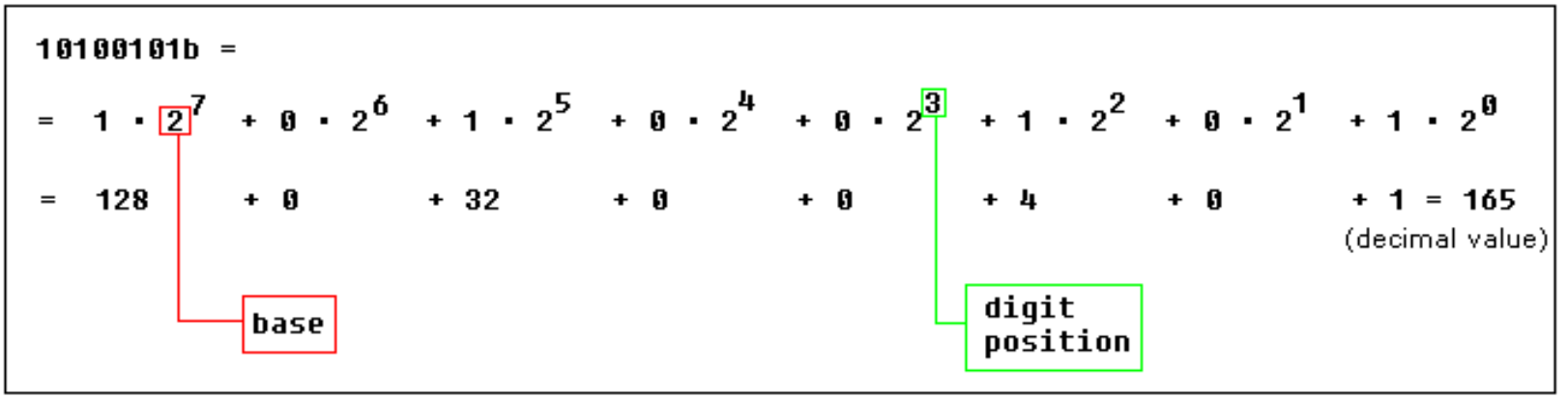
DEC	BIN	HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Rappresentazione in base 16 di un numero in base 10

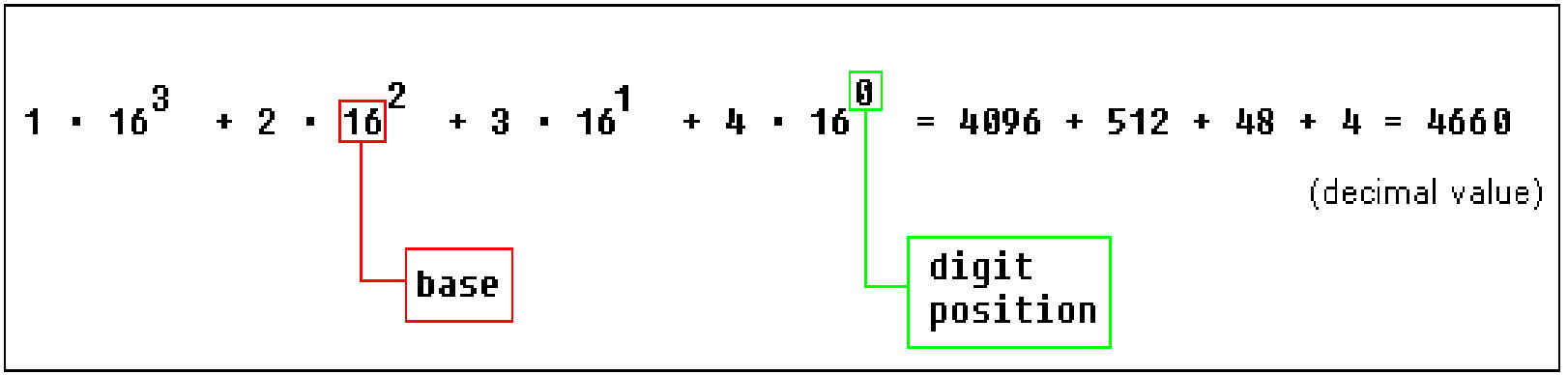


Conversioni da base 2, base 16 a base 10

- Da base 2 a base 10:

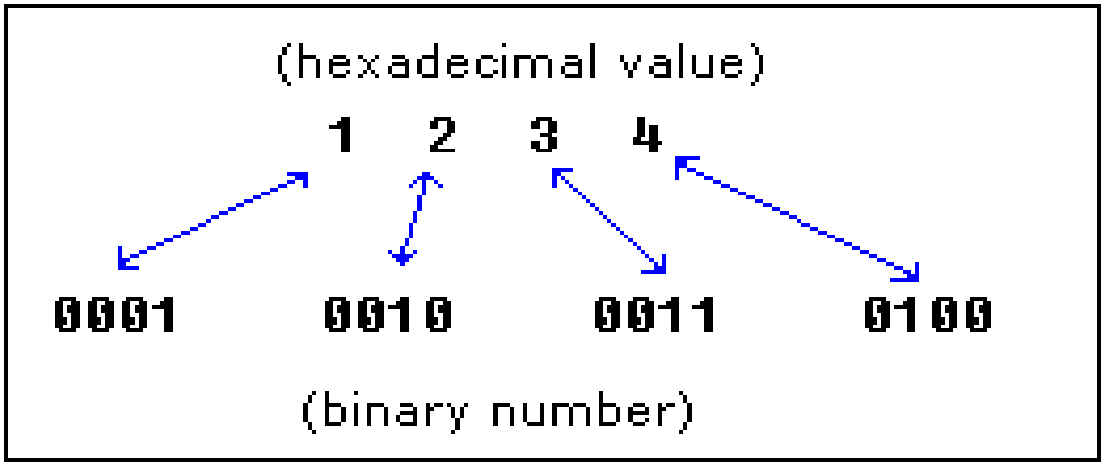


- Da base 16 a base 10:



Notiamo una certa caratteristica nella rappresentazione in base 16

- Dalla tabella riassuntiva delle cifre esadecimali si vede chiaramente che esiste una corrispondenza biunivoca tra codici binari a 4 bit e le cifre della rappresentazione esadecimale;
- Questo è il motivo chiave che ha spinto ad adottare la rappresentazione in base 16: permette infatti di rappresentare in forma compatta e facilmente leggibile i numeri binari di grande dimensione;
- Convertire un numero binario in uno esadecimale e viceversa è un'operazione banale. Infatti:



Convenzioni

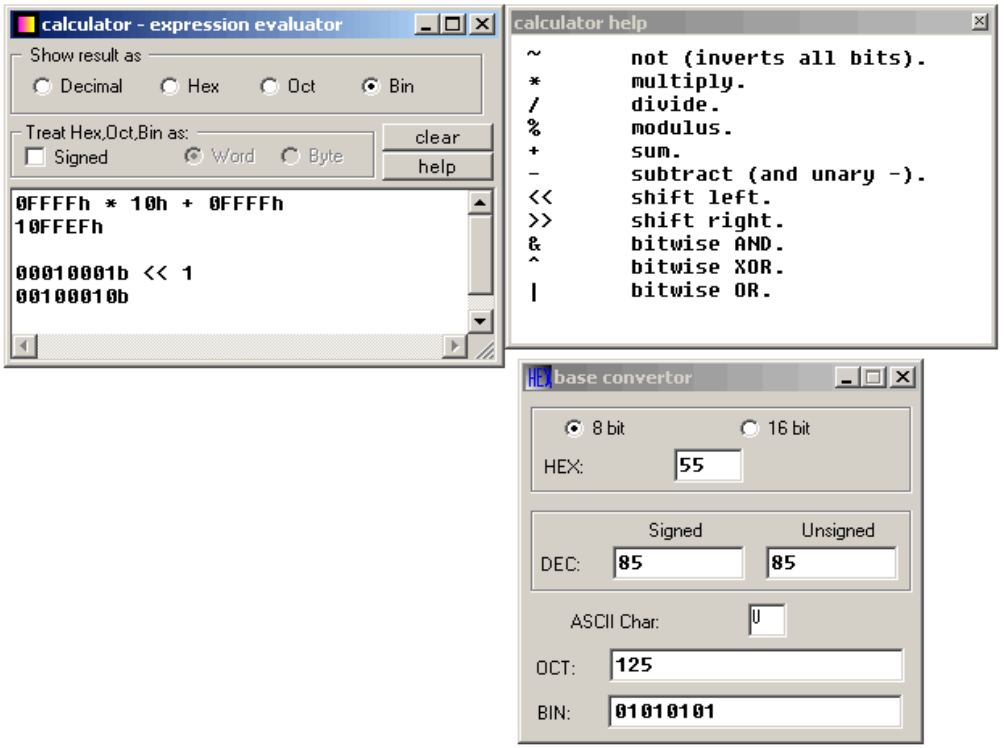
- Per indicare un numero in base 10, ad esempio ***x***, si utilizzerà $(\boldsymbol{x})_{10}$, o più semplicemente ***x***;
- Per indicare un numero in base 16, si utilizzerà la sintassi $(\boldsymbol{x})_{16}$, o più semplicemente ***x***h;
- Per indicare un numero in base 2, si utilizzerà la sintassi $(\boldsymbol{x})_{2}$, o più semplicemente ***x***b;
- Sia ***x*** un carattere: di seguito si intenderà con '***x***' il codice numerico associato alla codifica di ***x***.

Concludendo: perché tre basi diverse?

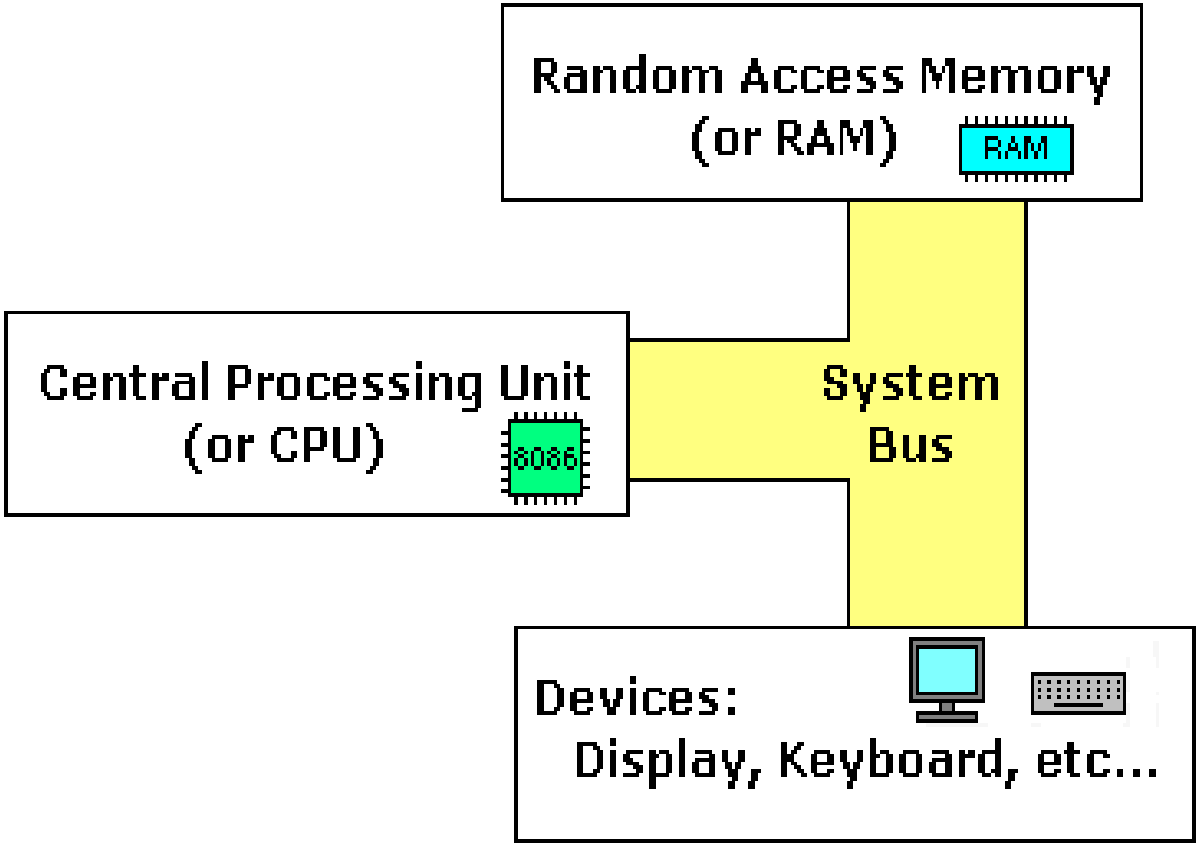
- Le ragioni che hanno spinto i programmatori a basso livello ad adottare tre basi diverse per indicare un numero sono le seguenti:
  - La base 10 permette di rappresentare i numeri nella forma che meglio conosciamo: sin dagli albori dell'umanità l'uomo ha contato utilizzando le dieci dita;
  - La base 2 permette di rappresentare i numeri nella forma canonica dei calcolatori: è particolarmente utile quando si vuole specificare una particolare sequenza di cifre binarie;
  - Spesso, le cifre binarie che si vogliono rappresentare sono molto lunghe: l'utilizzo della base 16 permette di rendere quattro volte più compatta la loro rappresentazione.

Ecco come emu8086 ci assiste in task di conversione di base

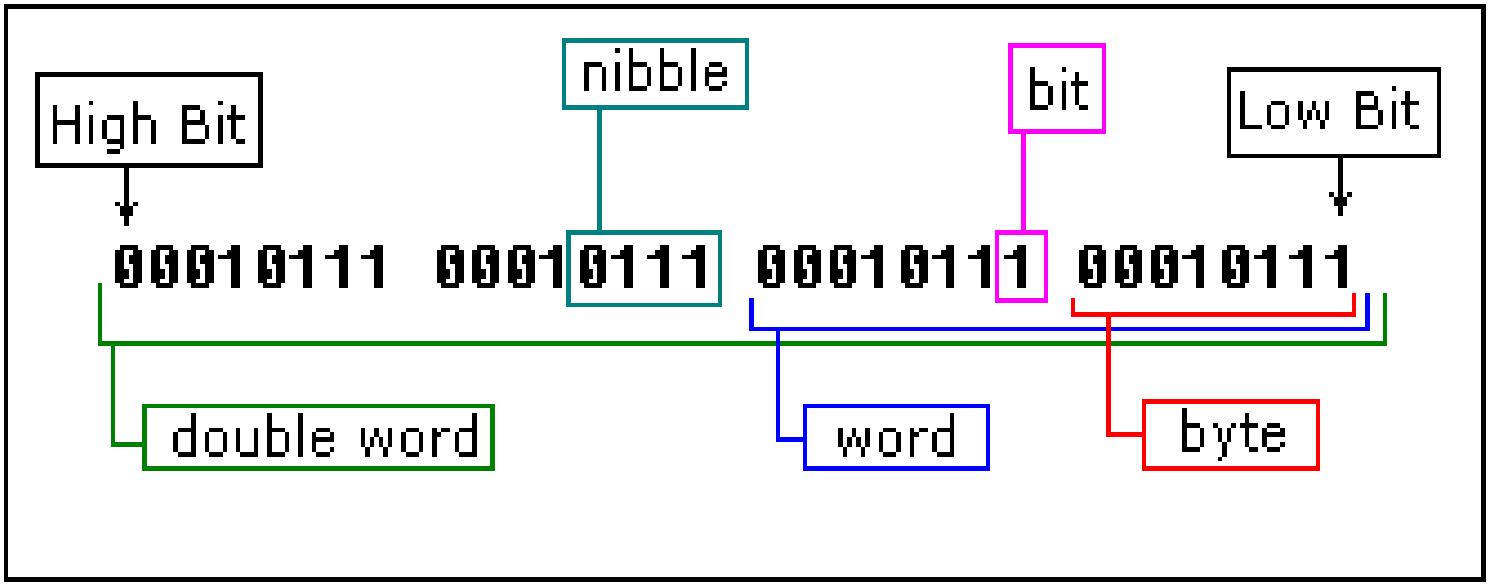
- In emu8086 disponiamo di una calcolatrice e di un convertitore.



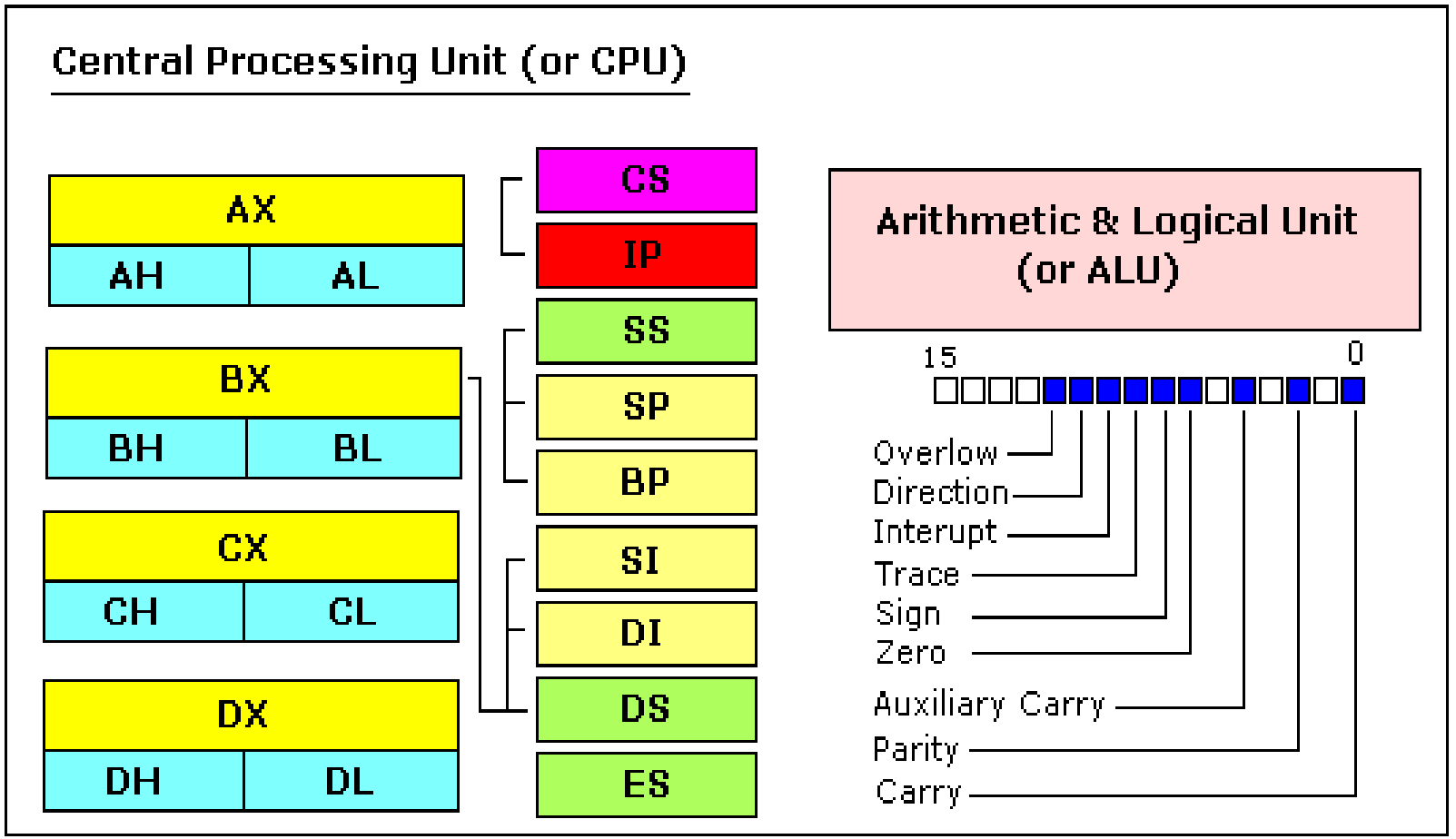
Come è fatto il nostro calcolatore



Un po' di terminologia



Come è fatta la una CPU i80x86 a 16 bit al suo interno



I registri general purpose

- Le CPU i80x86 hanno otto registri general purpose, ognuno dei quali ha un proprio nome:
  - ***AX***: il registro accumulatore (diviso su ***AH*** / ***AL***);
  - ***BX***: il registro di indirizzo base (diviso su ***BH*** / **BL**);
  - ***CX***: il registro contatore (diviso su ***CH*** / ***CL***);
  - ***DX***: il registro dei dati (diviso su ***DH*** / ***DL***);
  - ***SI***: registro di indice sorgente;
  - ***DI***: registro di indice destinazione;
  - ***BP***: registro base pointer;
  - ***SP***: registro puntatore alla pila.
- Anche se i registri hanno un determinato scopo, è comunque il programmatore che decide cosa memorizzarci.

Quanto sono grandi i registri?

- In linea con la definizione introdotta precedentemente, possiamo affermare che AX, BX, CX, DX, SI, DI BP e SP sono delle ***word***, mentre AH, AL, BH, BL, CH, CH, DH e DL sono dei ***byte***.

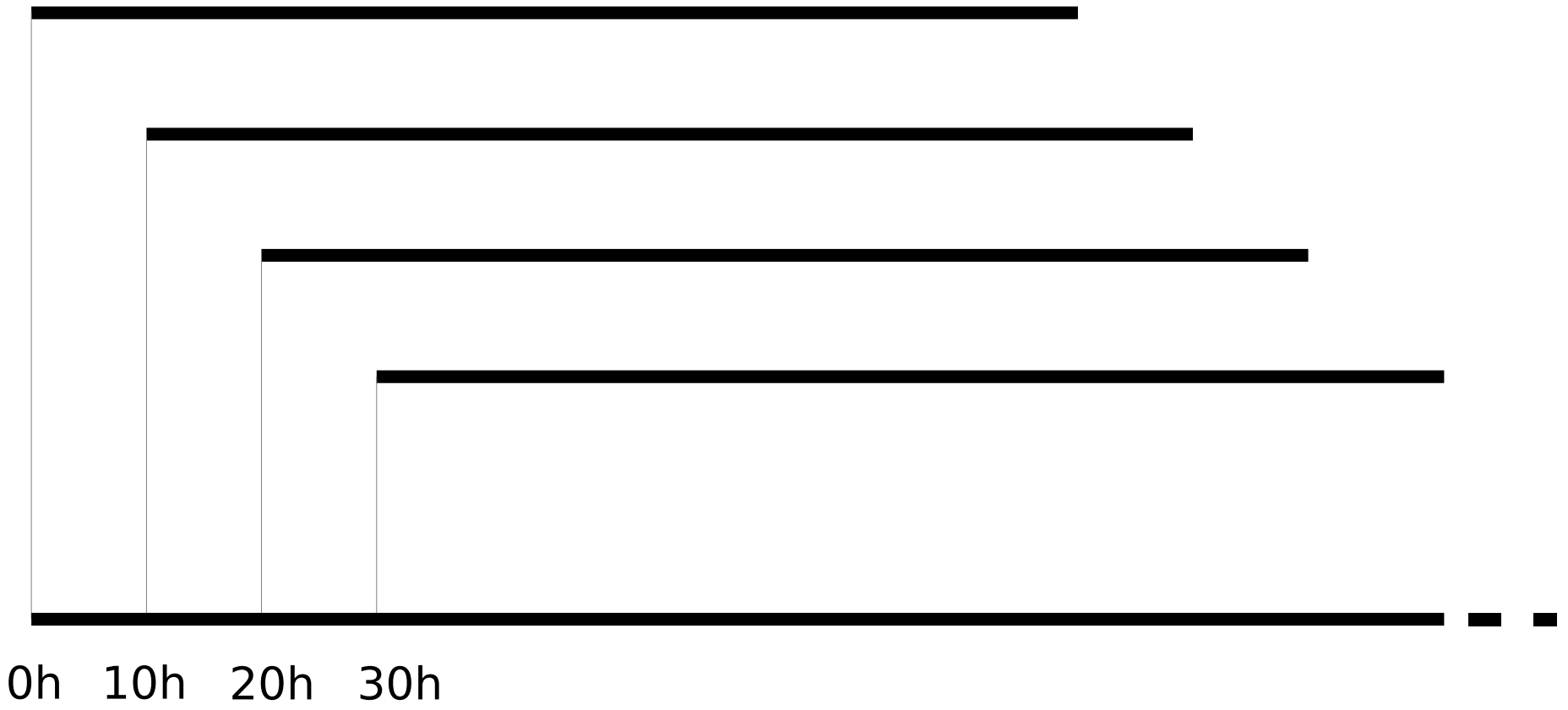
Cosa significa, ad esempio, “diviso su ***AH*** / ***AL***”

- Con la definizione “***AX***: il registro accumulatore (diviso su ***AH*** / ***AL***);” si intende dire che AH è il byte più significativo di AX, mentre AL è quello meno significativo;
- Modificando AX si effettuano implicitamente anche delle modifiche su AH e AL, a seconda della modifica che si sta apportando;
- Modificando AH o AL si fanno inevitabilmente delle modifiche anche su AX.

Segmentazione della memoria centrale

- Le prime CPU i80x86 a 16 bit adottavano un sistema di indirizzamento basato sulla ***segmentazione***;
- I ***segmenti*** sono porzioni di memoria sovrapposti. Il primo segmento comincia alla locazione 0h, e il segmento n-esimo inizia a 10h celle di istanza dall'inizio del segmento (n-1)-esimo. Ad esempio, il secondo segmento inizia alla cella 10h, il terzo segmento alla cella 20h e così via;
- Per indirizzare un segmento viene utilizzata una word;
- I segmenti permettono di contestualizzate gli accessi in memoria: in questo modo è infatti possibile identificare le variabili come ad esempio “la cella di memoria 24h del segmento 41h”. Grazie a questo sistema è possibile creare e utilizzare più istanze delle stesse variabili semplicemente cambiando il numero di segmento;
- Successivamente, per indicare l'indirizzo della cella YYYYh del segmento XXXXh utilizzeremo la sintassi ***XXXX:YYYY***.

Segmentazione della memoria centrale: esempio a quattro segmenti



I registri segment

- Le CPU i80x86 hanno quattro registri segment, ognuno dei quali ha un proprio nome:
  - ***CS***: punta al segmento che contiene il codice del programma che si sta eseguendo;
  - ***DS***: in genere punta al segmento dove sono situati i dati di interesse per l'applicazione;
  - ***ES***: registro di segmento extra: è il programmatore che ne stabilisce l'utilizzo;
  - ***SS***: punta al segmento contenente lo stack del processo corrente.

I registri speciali

- Le CPU i80x86 hanno due registri speciali:
  - ***P***: è l'instruction pointer;
  - ***flags register***: determina lo stato corrente della CPU.

Indirizzamento

- I registri ***BX***, ***DI***, ***SI***, ***BP*** possono essere utilizzati per effettuare degli indirizzamenti;
- Combinando questi registri all'interno di una coppia di parentesi quadre è possibile raggiungere differenti locazioni di memoria;
- E' possibile anche utilizzare solo uno di questi registri per effettuare un indirizzamento;
- Segue la tabella che mostra in che modo possono essere presi questi registri: quelli nella stessa colonna possono essere presi in mutua esclusione;

BX	SI	+ disp
BP	DI	

Indirizzamento

- Il ***disp*** rappresenta un displacement, ovvero uno spostamento, positivo o negativo, rappresentato da una costante che può essere sia una word che un byte;
- Il displacement può trovarsi sia dentro che fuori dalle parentesi quadre;
- E' comunque possibile utilizzare solo il displacement per effettuare un indirizzamento.

Indirizzamento

- A monte di un qualsiasi indirizzamento che non contempli il registri BP, vi è associato il segmento memorizzato in DS;
- Con ciò si intende dire che viene implicitamente effettuata una somma con DS*10h, per motivi visti in precedenza;
- Per quegli indirizzamenti che contemplano il registro BP, invece, gli viene associato il registro SS;
- Come abbiamo detto in precedenza, i registri DS ed SS sono utilizzati per specificare il segmento di dati e il segmento della pila, per rendere relativi i riferimenti alle variabili.

Indirizzamento: word o byte?

- Necessitiamo di un meccanismo per indicare alla CPU se ci riferiamo ad una word o ad un byte;
- Per fare ciò, è stata introdotta la seguente sintassi:
  - ***byte ptr [...]*** per accedere ad un byte;
  - ***word ptr [...]*** per accedere ad una word.

“Acquisire una parte del set di istruzioni disponibile nella famiglia di processori i80x86”

Prima di iniziare...

- Prima di cominciare ad analizzare le istruzioni, è necessario introdurre la sintassi per utilizzare le variabili nei nostri programmi e la motivazione che l'hanno portata ad essere introdotta nei compilatori;
- Per scrivere programmi è indubbio che sia necessario disporre di variabili, vettori e altro;
- E' altrettanto indubbio che per il programmatore è più facile utilizzare le variabili tramite un nome simbolico (ad esempio “contatore”) piuttosto che un indirizzo reale (ad esempio [0029:0132]).

Come dichiarare le variabili

- La sintassi per dichiarare una variabile è la seguente:
  - *nome* DB *valore*
  - *nome* DW *valore*
  - *nome* DD *valore*
- *DB* è l'acronimo di “Define Byte”, *DW* di “Define Word” e *DD* di “Define Double word”;
- *nome* è una sequenza case insensitive di caratteri alfanumerici, sebbene il primo carattere debba essere necessariamente una lettera;

Come dichiarare le variabili

- Se il nome è assente, la variabile non avrà un nome, ma sarà associata comunque ad un indirizzo;
- *valore è il valore di inizializzazione della variabile; può essere un numero in una delle basi supportare (binario, decimale, esadecimale e ottale) e se è pari a '**?**' la variabile non viene inizializzata.*
- *In realtà è possibile specificare una sequenza di valori, anche in basi diverse, separati da una virgola: in questo modo è possibile ricreare un vettore in cui il nome è il puntatore al primo elemento della sequenza;*
  - *Vettore di tre elementi: vettore_byte DB 0Ah, 13, 10100111b*

Come dichiarare le variabili

- E' possibile specificare un byte anche per mezzo del carattere che esso rappresenta per mezzo del codice ASCII a 8 bit.
  - ES: 'A' corrisponde al codice ASCII 65 decimale.
- In realtà il nome associato ad una variabile rappresenta l'indirizzo in memoria del suo byte meno significativo;
- Questo accade perché l'indirizzamento è livello di byte: se così non fosse non potremmo indirizzare le variabili di tipo byte;
- E' possibile dichiarare anche variabili stringa sotto forma di sequenza di byte nel seguente modo:
  - *nome* **DB** "*stringa*$"

Come dichiarare le variabili

- In questa maniera, vengono allocati tanti bytes quanto è lunga la stringa, compreso il simbolo '**$**';
- '**$**' è, per convenzione con il S.O., il simbolo di terminazione stringa;
- Il *nome* si riferisce all'indirizzo in memoria del primo byte che costituisce la stringa.

Dichiarare una costante

- Una costante rappresenta una espressione che non può essere modificata a run-time come succede per le variabili, poiché il compilatore sostituisce il suo nome con il suo valore;
- Si dichiara nel seguente modo:
  - *nome* EQU *espressione*
- Viene utilizzata per dare un nome simbolico a delle espressioni con un determinato significato;
  - Es: NUMERO_FATTURE EQU 10

Esempio di dichiarazione di variabili

```
ORG 100h ; obbligatorio.

MOV AL, var1 ; qualche istruzione..
MOV BX, var2 ; ..di esempio

RET  ; arresta il programma, ritornando il
       ; controllo al sistema operativo

VAR1 DB 7
var2 DW 1234h
```

Nota sul formato COM

- Per semplicità simuleremo programmi che verranno compilati come COM nei sistemi Microsoft;
- Per fare ciò è *necessario* che il nostro programma cominci a distanza 100h dall'inizio della memoria riservata al processo, perché il S.O. utilizza questi primi 100h per memorizzare dati relativi all'eseguibile, come ad esempio i parametri passati su riga di comando;
- Negli eseguibili COM i registri di segmento CS, DS, ES, SS puntano inizialmente tutti allo stesso segmento;
- Il loro valore può essere cambiato a run-time.

L'istruzione MOV

- Questa istruzione permette di copiare un dato da una locazione ad un'altra;
  - MOV DEST, SRC
  - Es: MOV AX, VAR1 ; copia il contenuto di var1 nel registro AX
- La locazione sorgente e destinazione devono avere la stessa “capienza”;
  - Non è possibile, ad esempio, effettuare una copia di dati dal registro BL ad AX
- Si noti che questo genere di vincoli, anche se sarebbero in qualche maniera gestibili, rimangono comunque validi perché sono legati all'architettura della CPU.

L'istruzione MOV

- Quando si utilizza l'istruzione MOV specificando come sorgente il nome di una variabile, si sta inserendo nella locazione di destinazione il *contenuto* della variabile e non il suo indirizzo;
- Per permettere ciò si deve far precedere il nome della variabile dalla parola chiave *offset*
  - Es: MOV DX, offset VAR1

L'istruzione XCHG

- Questa istruzione permette di scambiare “in un colpo solo” il contenuto di due locazioni di pari dimensione;
  - XCHG AX, BX ; quello che c'era in AX ora si trova in BX e viceversa

L'istruzione LEA

- L'istruzione LEA (Load Effective Address) permette di caricare nella locazione di destinazione l'indirizzo della locazione sorgente (che in questo caso non può essere un registro);
- La si può vedere come se fosse una MOV utilizzata con la keyword offset, ma in realtà è in grado di ricavare l'indirizzo di locazioni di memoria in modo più complesso;
  - LEA DX, array[DI]

L'istruzione AND, TEST, OR, XOR

- Queste funzioni sono l'implementazione degli omonimi operatori bitwize;
- Hanno due parametri:
  - Es: AND AX, BX ; parlando più ad alto livello corrisponde ad un AX = AX & BX
- L'istruzione TEST effettua le stesse operazioni dell'istruzione AND, con la sola differenza che non memorizza il risultato;
- Viene utilizzato per ricreare lo stesso effetto che l'istruzione AND ha sui flag;

L'istruzione ADD

- Permette di addizionare due valori: in particolare preleva il valore della prima locazione specificata come argomento, lo addiziona a quello della seconda e il risultato lo ripone nella prima;
  - Es: ADD AX, VAR1
- Imposta il flag C (Carry) se c'è stato overflow in una operazione tra numeri unsigned;
- Imposta il flag O (Overflow) se c'è stato un overflow in una operazione tra numeri signed;

L'istruzione SUB / CMP

- L'istruzione di SUB è analoga a quella di ADD, con la sola differenze che effettua, per l'appunto, una sottrazione;
- E' utile per effettuare dei controlli, ad esempio se due locazioni contengono lo stesso valore;
  - Es: SUB AX, BX ; viene configurato a 1 il flag Z (Zero) se AX e BX sono uguali;
- Il problema dell'istruzione SUB per fare confronti tra locazioni consiste con il fatto si deve preventivamente salvare il contenuto del primo operando;
- Per ovviare a questo problema è stata introdotta l'istruzione CMP, che effettua una sottrazione esattamente come SUB con la sola differenza che il risultato non viene memorizzato in nessuna locazione, ma ricrea lo stesso effetto che SUB ha sui flag;

L'istruzione INC / DEC

- L'istruzione INC equivale a tutti gli effetti ad un ADD, il cui primo parametro è lo stesso passato ad INC mentre il secondo è la costante 1;
  - Es: INC AX
- Analogamente, l'istruzione DEC equivale a tutti gli effetti ad un SUB, il cui primo parametro è lo stesso passato ad DEC mentre il secondo è la costante 1.
  - Es: DEC BX

L'istruzione MUL / IMUL

- Analogamente alle altre istruzioni aritmetiche, MUL effettua la moltiplicazione tra due numeri unsigned;
  - Es: MUL CH
- L'istruzione IMUL effettua, invece, la moltiplicazione tra due numero signed;
- In entrambi i casi, viene passato un solo argomento: in particolare:
  - se è un byte, allora la moltiplicazione viene effettuata con AL e il risultato viene posto in AX;
    - Effetto MUL: flag C=0 e flag O=0 se e solo se AH=0
    - Effetto IMUL: flag C=0 e flag O=0 se e solo se AH è estensione di segno di AL
  - se è una word, allora la moltiplicazione viene effettuata con AX e il risultato viene posto in DX:AX (notazione spiegata in seguito)
    - Effetto MUL: flag C=0 e flag O=0 se e solo se DX=0

L'istruzione MUL / IMUL

- L'argomento all'istruzione non deve essere una locazioni modificate per memorizzare il risultato;
- Con la notazione *DX:AX* si intende che la word più significativa viene posta in DX e quella meno significativa in AX;

L'istruzione DIV / IDIV

- Analogamente alle altre istruzioni aritmetiche, DIV effettua la divisione tra due numeri unsigned;
- L'istruzione IDIV effettua, invece, la divisione tra due numero signed;
- L'operando non può essere uno di seguenti registri: AH, AL, AX.
- In entrambi i casi, viene passato come argomento il divisore: in particolare:
  - se voglio dividere una word per un byte devo mettere in AX: il risultato sarà posto in AL e il resto in AH;
  - se voglio dividere una double word (dword in breve) per una word, allora devo porre il dividendo in DX:AX: il risultato sarà posto in AX e il resto in DX;

L'istruzione DIV / IDIV

- L'argomento all'istruzione non deve essere una locazioni modificate per memorizzare il risultato;
- Se si tenta una divisione per zero, la CPU esegue l'interrupt 0h, termina l'esecuzione del processo, e stampa a video una stringa simile a “divisione per zero”;
- Gli interrupt verranno affrontati in seguito;
- Se il risultato è troppo grande per entrare in AL/AX per le divisioni su byte/word allora verrà stampato a schermo un errore simile a “overflow di divisione” e l'operazione non produrrà nessun risultato;

L'istruzione CWD

- Generalmente viene utilizzata in corrispondenza di una IDIV e il suo obiettivo è quello di estendere il segno del valore in AX anche in DX, per poter così effettuare la divisione di una dword con una word.

Istruzioni di PUSH / POP

- Queste due istruzioni agiscono sulla pila del processo associato al nostro programma;
- La pila è una struttura dati LIFO: questo acronimo sta per Last In First Out;
- Letteralmente, “L'ultimo elemento che inseriamo sarà il primo ad uscire”

Istruzioni PUSH / POP

- Le operazioni possibili sulla pila sono due:
  - PUSH: inserimento di un nuovo elemento sulla cima della pila;
  - POP: estrazione dell'ultimo elemento inserito nella pila
- Per fare un esempio concreto, le seguenti istruzioni PUSH AX, POP BX eseguite in successione corrispondono ad effettuare un MOV BX, AX.
- In realtà le istruzioni di PUSH e POP vengono utilizzate per due scopi fondamentali:
  - Salvataggio di dati posti in locazioni che devono essere temporaneamente modificate;
  - Passaggio dei parametri effettivi alle chiamate a funzione;

L'istruzione SHR / SHL

- SHR effettua tanti shift a destra al suo primo operando quanti ne sono indicati nel secondo;
  - Es: SHR AX, 2
- Si noti che SHR X, Y equivale a dividere X per 2^Y.
- In maniera analoga, SHL effettua shift a sinistra al suo primo operando quanti ne sono indicati nel secondo;
- Si noti che SHL X, Y equivale a moltiplicare X per 2^Y.
- I'ultimo bit eliminato dagli shift viene posto nel flag C: nel caso di uno shift a destra, il flag Ca contiene in particolare il resto della divisione.

L'istruzione ROR / ROL

- ROR effettua una “rotazione a destra”: con ciò si intende che viene effettuato uno shift a destra, con la sola differenza che il valore del bit appena uscito (quello meno significativo) viene riposto in quello più significativo;
- Il nome deriva proprio dal senso di rotazione che fornisce ai bit;
- La sintassi è la stessa di SHR;
- ROL effettua una “rotazione a sinistra”, con un effetto sui bit speculare a quello fornito da ROR;

L'istruzione LOOP

- L'istruzione accetta un solo parametro: la locazione di memoria dove saltare se il ciclo non è ancor terminato;
- Tipicamente, questo valore viene tradotto in un offset (o displacement) relativo alla posizione in memoria del LOOP;
- Verosimilmente, la locazione di salto si troverà in una locazione di memoria più bassa (con address inferiore) rispetto a quella dove è posta l'istruzione LOOP;
- Decrementa CX e se è diverso da zero (flag Z configurato a 0) salta alla locazione di memoria specificata;
- Se CX è uguale a zero, allora l'esecuzione riprenderà dall'istruzione immediatamente successiva a LOOP;
- Viene utilizzata per implementare i loop semplici, come ad esempio i cicli *for* ad un contatore;

Il salto JMP

- JMP è l'istruzione di salto incondizionato: viene effettuato a prescindere da tutto;
- L'argomento passato è un indirizzo in memoria (espresso sotto forma di etichetta, come avremo modo di vedere in seguito) al quale saltare.

Il salto J<f> / JN<f>

- Queste istruzioni effettuano i salti in relazione ai valori dei flags;
- Con <f> si intende una qualunque delle lettere simboliche associate ai flags:
  - C è il flag Carry
  - Z è il flag Zero
  - O è il flag Overflow
  - P è il flag Parity
- In particolare, l'istruzione J<f> salta all'indirizzo specificato come parametro se e soltanto se il flag <f> è configurato a uno;
- Di contro, l'istruzione JN<f> salta all'indirizzo specificato come parametro se e soltanto se il flag <f> è configurato a zero;

I salti di relazione

- Questi salti vengono solitamente utilizzati in prossimità di un'istruzione CMP;
- Il loro scopo è verificare in che relazione si trovano i due argomenti passati: poniamo che il primo argomento passato alla CMP sia X e che il secondo sia Y, allora:
- Si distinguono due categorie di salti di relazione
  - Salti condizionati su operazioni signed;
  - Salti condizionati su operazioni unsigned;

Salti condizionati signed

- JG: salta se X > Y
- JNG: salta se X <= Y
- JGE: salta se X >= Y
- JNGE: salta se X < Y
- JL: salta se X < Y
- JNL: salta se X >= Y
- JLE: salta se X <= Y
- JNLE: salta se X > Y
- JE: salta se X = Y

Salti condizionati unsigned

- JA: salta se X > Y
- JNA: salta se X <= Y
- JAE: salta se X >= Y
- JNAE: salta se X < Y
- JB: salta se X < Y
- JNB: salta se X >= Y
- JBE: salta se X <= Y
- JNBE: salta se X > Y
- JE: salta se X = Y (coincide con i condizionati unsigned)

L'istruzione RET

- Questa istruzione ritorna il controllo alla procedura chiamante;
- Per ora si sappia che nel nostro contesto iniziale non faremo alcuna chiamata a procedura, e che quindi una RET provoca la terminazione del processo;
- Questo accade perché la procedura chiamante del nostro programma è il Sistema Operativo.

Le etichette

- Le etichette sono utilizzate per specificare una particolare locazione in memoria;
- In particolare, assumono come valore l'indirizzo della prima istruzione che la segue.
- Rendono estremamente comodo l'utilizzo dei salti e affini, come ad esempio l'istruzione LOOP.

Gli Interrupts

- Un interrupt può essere visto come una collezione di servizi;
- Ogni servizio è univocamente identificato da un codice, un byte nella fattispecie, che deve essere posizionato in un registro candidato dal progettatore dell'interrupt;
- Gli interrupt sono solitamente offerti sia dal BIOS del calcolatore che dal Sistema Operativo;

L'istruzione INT

- Questa istruzione permette di eseguire un l'interrupt specificato come parametro;
  - Es: INT 21h
- Può essere visto come una chiamata a funzione con qualche dote in più.
- Solitamente, come accade anche nel nostro caso, gli interrupt sono servizi offerti da terze parti per permette una agevole programmazione del calcolatore;
- Nel seguito faremo una panoramica di quelli disponibili in emu8086 e quindi anche in un calcolatore munito di MS-DOS;

Rapido overview degli interrupts

- 10h, 11h, 12h, 13h, 15h, 16h, 19h, 1Ah, 20h sono gli interrupt del BIOS del calcolatore;
- Rendono possibile l'accesso alle peculiarità della scheda video e ad una serie di aspetti i basso livello;
- 21h, 33h sono gli interrupt forniti dall'MS-DOS;
- Permettono tutti quegli aspetti legati ai compiti del sistema operativo;

Alcuni interrupt utili

- Per convenzione il numero di servizio viene inserito in AH;
- INT 21h, servizio 1h: legge un carattere da tastiera (con echo)
  - Ritorno
    - AL: il carattere appena letto
- INT 21h, servizio 2h: scrive un carattere sullo standard output
  - Parametri
    - DL: la codifica ASCII del carattere da scrivere;
  - Ritorno
    - AH = DL
- INT 21h, servizio 9h: scrive una stringa terminata con '$' sullo standard output.
  - Parametri
    - DS:DX deve contenere l'indirizzo della stringa;

Alcuni interrupt utili

- INT 16h, servizio 0h: preleva lo scancode (un carattere) dalla tastiera, senza stamparlo sullo standard output
  - Ritorno
    - AH contiene lo scancode del BIOS;
    - AL contiene il codice ASCII associato al carattere appena letto;
  - Nota: se era presente uno scancode nel buffer della tastiera, esso viene rimosso.

Definizione di funzione

- Sintassi di apertura di una routine:
  - <NOME_PROCEDURA> PROC <NEAR|FAR>
- Sostanzialmente, una procedura si apre con una riga che contiene il suo nome, la keyword PROC e una parola chiave che può essere o NEAR o FAR;
- Sintassi di chiusura di una routine:
  - <NOME_PROCEDURA> ENDP
- Una procedura si chiude scrivendo una riga che contiene il nome della procedura da chiudere seguita dalla parola chiave ENDP.

Definizione di funzione

- La parola chiave NEAR indica che la procedura deve essere sufficientemente vicina alla locazione dove viene effettuata la chiamata, e che verrà generato un codice di chiamata di 3 bytes;
- La parola chiave FAR indica che la procedura deve essere sufficientemente lontana dalla locazione dove viene effettuata la chiamata, e che verrà generato un codice di chiamata di 5 bytes;
- Per semplicità utilizzeremo tutte procedure NEAR.

Chiamate a funzione

- La sintassi per effettuare una chiamata è la seguente:
  - CALL <NOME_PROCEDURA>
- Si noti l'assenza del meccanismo per il passaggio dei parametri...

Chiamate a funzione

- A differenza dei linguaggi ad alto livello, non viene generato automaticamente il codice necessario ad effettuare il ritorno dalla funzione;
- L'istruzione RET deve essere scritta esplicitamente alla fine della procedura;
- In base al tipo di procedura (NEAR o FAR) viene generato il codice più opportuno per RET.

Passaggio dei parametri

- Come si è avuto modo di notare, la sintassi per la dichiarazione di procedure non permette di specificare ne' quanti sono i parametri alla chiamata e ne' di che tipo essi devono essere;
- Questo aspetto è completamente delegato al programmatore!

Passaggio dei parametri

- Dobbiamo trovare un modo più o meno conveniente di passare i parametri alle funzioni in modo efficace;
- Per semplicità possiamo utilizzare i registri per la memorizzazione dei parametri;
- Dobbiamo però tenere presente i pro e i contro:
  - CONTRO:
    - Numero di parametri limitato al numero dei registri;
    - Potenzialmente i registri utilizzati per il passaggio dei parametri hanno un contenuto utile che deve essere salvato, per poter essere poi utilizzato, ad esempio, al ritorno della chiamata;
  - PRO:
    - Il passaggio dei parametri diventa un'operazione estremamente banale;

Passaggio dei parametri

- Come risolviamo gli aspetti negativi di questa tecnica di passaggio?
  - Il numero di parametri in realtà non è un problema.
    - Es: supponiamo di dover passare 160 parametri ad una procedura: un modo per riuscire nell'intento è quello di creare un vettore di 160 elementi e passare tramite un registro solamente il puntatore a questo vettore.
  - Il salvataggio dei registri può essere fatto sfruttando lo stack del processo:
    - Effettuiamo il PUSH di un registro qualora esso venga utilizzato per passare un parametro, immediatamente prima che esso venga utilizzato per tale scopo;
    - Effettuiamo il POP del registro appena ritorniamo dalla procedura.

Uso dei registri all'interno di una procedura

- Se la procedura che stiamo scrivendo non è banale, sicuramente avrà bisogno di utilizzare dei registri;
- In particolar modo, per evitare di perdere i dati precedentemente contenuti nei registri, dobbiamo trovare il modo per poter salvare il loro contenuto da qualche parte;
- Utilizziamo la stessa soluzione adottata per salvare il contenuto dei registri al momento del passaggio dei parametri alla procedura.
- L'unica differenza è che questa volta la procedura i salvataggio deve essere effettuata internamente alla procedura.

Uso di PUSH e POP

>>!!! ATTENZIONE !!!<<

PUSH e POP operano su una PILA: questo vuol dire che dobbiamo fare attenzione all'ordine con il quale eseguiamo queste istruzioni.
Se volessi salvare AX e BX sullo stack e poi volessi successivamente ripristinarli, devo eseguire le istruzioni in questa sequenza:

PUSH AX; PUSH BX; POP BX; POP AX

sostanzialmente, l'ordine di esecuzione dei POP è l'inverso di quello delle PUSH.

Riassumendo

- Per salvare i dati dei registri per il passaggio dei parametri vanno effettuate delle PUSH immediatamente prima della chiamata a funzione;
- Per salvare il contenuto dei registri da utilizzare all'interno della procedura si devono effettuare delle PUSH prima di effettuare qualunque altra operazione;
- Per ripristinare il contenuto dei registri utilizzati internamente alla procedura si devono effettuare delle POP immediatamente prima dell'istruzione di return (RET), nell'ordine inverso con il quale sono stati effettuati i PUSH.
- Per ripristinare il contenuto dei registri utilizzati per il passaggio dei parametri si devono effettuare delle POP immediatamente dopo l'istruzione di chiamata a funzione (CALL), nell'ordine inverso con il quale sono stati effettuati i PUSH.

Esempio riassuntivo

```
org 100h

jmp start

hw db "hello world!$"

start:

push cx ; salva il registro per potevi inserire il parametro
mov cx, offset hw
call PROC_PROVA
pop cx ; ripristina lo stato del registro
ret ; ritorna il controllo al S.O.

PROC_PROVA PROC NEAR
    push dx ; salva il registro utilizzato dalla procedura
    mov ah, 9
    mov dx, cx
    int 21h
    pop dx ; ripristina il contenuto di DX
    ret ; se lo dimentichiamo la procedura NON termina!
PROC_PROVA ENDP
```

Altre istruzioni

- Nel seguito vedremo altre istruzioni presenti nel set i80x86 di base;
- Il motivo per il quale si è deciso di affrontarle solo ora è stato per dare modo di fare un po' di pratica con l'ambiente dell'emulatore e dei programmi scritti in assembly.

L'istruzione IN / OUT

- Le porte sono un insieme di word, di numero di gran lunga più piccolo rispetto a quello della momoria centrale, che permette di far comunicare la CPU con i dispositivi esterni;
- Sia la CPU che i dispositivi possono leggere e scrivere sulle porte: in questo modo è possibile creare un flusso di dati tra i dispositivi e la CPU;
- L'istruzione IN permette alla CPU di leggere un valore da una porta;
  - IN <AX|AL>, <port>
- L'istruzione OUT permette alla CPU di scrivere un valore su una porta;
  - OUT <port>, <AX|AL>

L'istruzione IN /OUT

- Si noti che, per motivi architetturali, gli unici due registri che possono interagire con le porte sono AL e AX;
- Abbiamo a disposizione AX per poter effettuare delle letture/scritture di word sulle porte;
- Abbiamo a disposizione AL per poter effettuare delle letture/scritture di byte sulle porte;
- In realtà, quando si interagisce con il registro AL, si sta leggendo/scrivendo il byte meno significativo della word contenuta nella porta.

L'istruzione NOP

- Non fa nulla...
- La sua codifica è lunga un byte;
- Permette di ottimizzare l'esecuzione dei programmi, ad esempio allineando l'inizio delle istruzioni a multipli di word;
- Si noti che la word è tipicamente la dimensione del bus: gli allineamenti permetterebbero di migliorare la fase di fetch delle istruzioni, evitando così dei riallineamenti per permette alla CPU di eseguire i codici operativi.

L'istruzione ST<f> / CL<f>

- ST<f> configura 1 il flag <f>;
- CL<f> configura a 0 il flag <f>;

L'istruzione CMPSB

- CMPSB effettua un confronto tra due sequenze di byte;
- La prima sequenza è puntata da [DS:SI];
- La seconda sequenza è puntata da [ES:DI];
- L'istruzione è anticipata dalla parola chiave REPE (o REPNE), per indicare che il confronto deve continuare fintantoché trova byte uguali (diversi);
- Il numero massimo di confronti deve essere inserito in CX;
- Il flag D (Direction) stabilisce qual'è la “direzione” del confronto:
  - D=0 indica che si vuole confrontare le stringhe partendo dal byte con indirizzo più basso verso quello più alto;
  - D=1 indica l'esatto opposto di D=0;

L'istruzione CMPSB

- I flag cono configurati dall'ultimo confronto:
  - Es: Z=1 dopo un REPE CMPSB vuol dire che le sequenze sono uguale nei primi CX byte.
- Si noti che questa istruzione si applica bene nel confronto di stringhe, dato che le codifiche ASCII dei caratteri occupano esattamente un byte;

L'istruzione CMPSW

- L'istruzione CMPSW è esattamente uguale a CMPSB, con la sola differenza che opera confrontando delle sequenze di word.

L'istruzione SCASB

- SCASB effettua una scansione del carattere la cui codifica è memorizzata in AL;
- La sequenza di byte nella quale cercare è puntata da [ES:DI];
- L'istruzione è anticipata dalla parola chiave REPE (o REPNE), per indicare chela ricerca deve continuare fintantoché trova byte uguali (diversi) a quello contenuto in AL;
- Il numero massimo di confronti deve essere inserito in CX;
- Il flag D (Direction) stabilisce qual'è la “direzione” del confronto:
  - D=0 indica che si vuole scansionare la stringa partendo dal byte con indirizzo più basso verso quello più alto;
  - D=1 indica l'esatto opposto di D=0;
- I flag sono configurati in modo analogo a CMPSB;

L'istruzione SCASW

- L'istruzione SCASW è esattamente uguale a SCASB, con la sola differenza che opera confrontando delle sequenze di word.