Assembler Intel 80x86: Struttura di un programma e Direttive

Calcolatori Elettronici B

a.a. 2004/2005

Massimiliano Giacomin

Istruzioni e direttive

- **Formato generale dei comandi**:

[nome] codice operazione [operandi] [; commento]

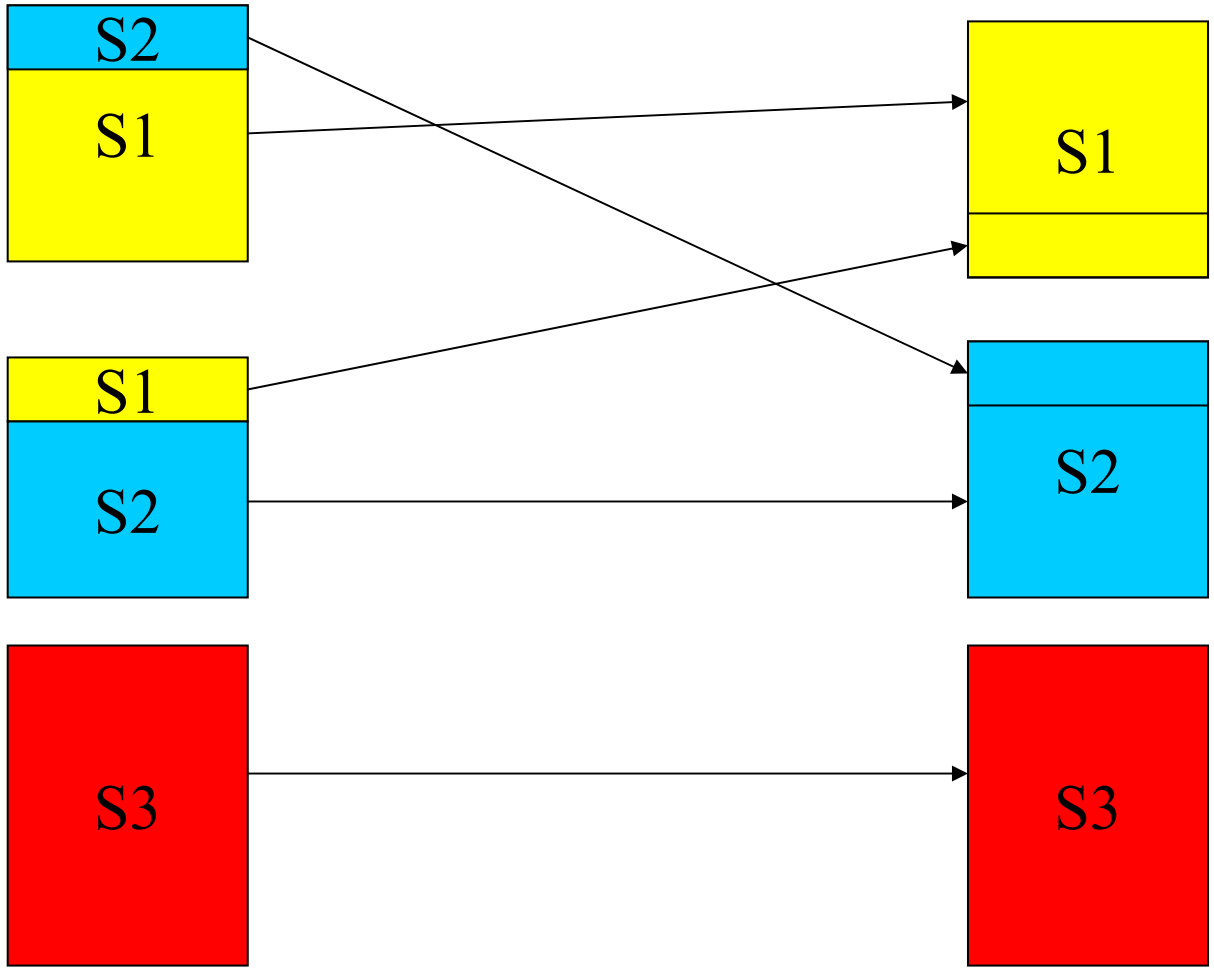
- Ci sono due tipi di comandi nel linguaggio assembler:
  - Le ***istruzioni***, che vengono tradotte in codice macchina dall'assemblatore
  - Le ***direttive***, che danno indicazioni all'assemblatore durante il processo di traduzione, ma non sono tradotte in istruzioni macchina

Struttura di un programma

- A livello simbolico, un programma è costituito da una serie di moduli. Questi vengono compilati e collegati dal linker per generare l'eseguibile.
- A livello di eseguibile, un programma è composto da una serie di segmenti, di 4 tipi diversi (stack, dati, codice, extra)
- Frammenti di segmento sono dichiarati nei diversi moduli:
  - I segmenti di tipo ***'stack segment'*** costituiscono quella zona di memoria in cui può essere fatto il push dei dati (funzionamento LIFO)
  - I segmenti di tipo ***'data segment'*** contengono le dichiarazioni delle variabili statiche (spazio di memoria allocato al momento della compilazione)
  - I segmenti di tipo ***'code segment'*** contengono le istruzioni e le direttive del programma da eseguire
  - I segmenti di tipo ***'extra segment'*** vengono definiti dall'utente e di solito vengono usati per dati addizionali
- I tipi dei segmenti sono ottenuti associando ai registri SS, DS, CS e ES l'indirizzo di partenza dei segmenti definiti (vedi dopo)
- **Come minimo**, per un programma, occorre definire il **segmento stack** e il **segmento codice**

Struttura di un programma (continua)

Moduli (livello simbolico) Segmenti (livello eseguibile)



Definizione di un (frammento di) segmento

Per indicare la definizione di un (frammento di) segmento si usa la direttiva SEGMENT:

nome-segmento SEGMENT *align-type combine-type 'class'*

corpo del segmento

nome-segmento ENDS

dove

- *nome-segmento* è l'identificatore del segmento cui il frammento appartiene
- *align-type* indica all'assembler come il frammento di segmento è allocato in memoria. Può essere:

  PARA deve iniziare a un indirizzo divisibile per 16 (paragrafo)

  BYTE può iniziare in qualsiasi locazione di memoria

  WORD deve iniziare in una locazione pari di memoria

  PAGE deve iniziare a un indirizzo in cui gli ultimi 8 bit sono 0

Definizione di segmento (continua…)

- *combine-type* indica come i frammenti di un segmento (aventi lo stesso nome) vanno combinati o caricati al momento del link. Useremo:

  PUBLIC i frammenti di segmento con lo stesso nome e pubblici vengono legati insieme, generando un segmento di lunghezza pari alla somma dei frammenti componenti.

  COMMON i frammenti vengono sovrapposti, generando un segmento di lunghezza pari alla massima tra i frammenti componenti.

  STACK per specificare che il frammento è parte dello stack. Simile a PUBLIC (dimensione stack = somma frammenti componenti) ma SS è necessariamente assunto come registro segmento (vedi poi) e SP è inizializzato con la dimensione finale dello stack, pari alla somma dei frammenti componenti.

Definizione di segmento (continua…)

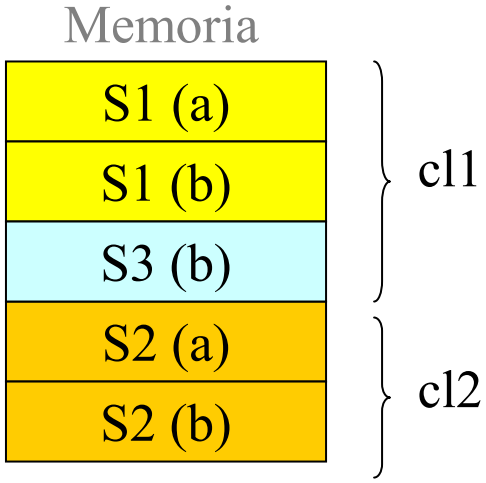
- *'class'* usato per far riferimento a una collezione di segmenti: definisce la "classe" cui appartiene il segmento.
  ⇒ I segmenti con lo stesso nome nel parametro 'class' (ovvero, appartenenti alla stessa classe) vengono allocati sequenzialmente in memoria

Esempio

M1
S1 SEGMENT PUBLIC 'cl1'
S2 SEGMENT PUBLIC 'cl2'

M2
S1 SEGMENT PUBLIC 'cl1'
S3 SEGMENT PUBLIC 'cl1'

M3
S2 SEGMENT PUBLIC 'cl2'



Esempio di programma

```
TITLE inutile.asm - Primo programma Assembler
COMMENT *
            DESCRIZIONE: questo programma inizializza AX a 0 e
            copia il valore 18 in AX. *
;
STACK SEGMENT PARA STACK 'STACK'
;           inizializzazione del segmento stack con 64 stringhe ‘STACK   ‘
;
            DB          64 DUP ('STACK     ')
STACK ENDS
;
CSEG SEGMENT PARA PUBLIC 'CODE'
            ASSUME  CS:CSEG, SS:STACK
;
INIT:       SUB         AX,  AX                 ; azzera il registro AX
            MOV         AX, 18D                 ; copia il valore 18 (decimale) in AX
;
            mov         AH, 4ch                 ; terminazione programma e ritorno al DOS
            int         21H
CSEG        ENDS
            END
```

Elementi del programma: direttive

- TITLE: specifica il nome del programma
- COMMENT: ciò che segue compreso fra due asterischi è un commento
- SEGMENT: già visto
- DB: serve per definire variabili (si veda dopo)
- ASSUME: ha la forma

  ASSUME *segment register:segment name, ...*

  nell'esempio il registro di segmento CS è associato al segmento CSEG, mentre il registro SS è associato al segmento STACK.

  Questo permette di dire all'assembler quale ***registro di segmento*** va usato per ogni segmento (in pratica permette di identificare il tipo del segmento)
- STACK ENDS, CSEG ENDS: indicano rispettivamente la fine del segmento stack e del segmento codice
- END: fine del modulo.
  Se vi sono più moduli, nel modulo principale viene specificato l'indirizzo della prima istruzione del programma (entry point):

  END prima_istruzione

Ancora sulla direttiva ASSUME

• L'informazione relativa alla direttiva ASSUME serve all'assemblatore per decidere quale registro segmento usare quando ci si riferisce ad una variabile definita in un determinato segmento.

• Es: ADD AX, V1 *; V1 è una variabile, indirizzam. diretto*

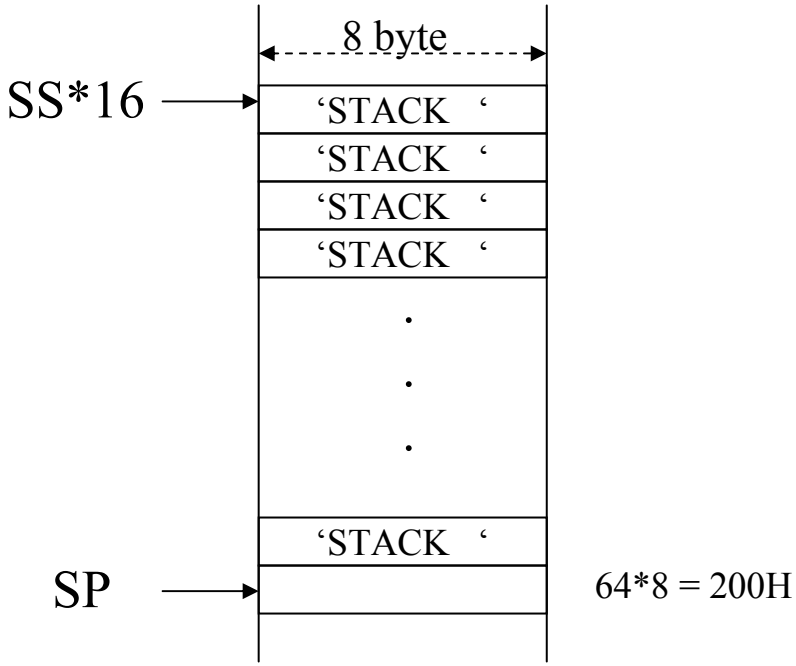
Di default si usa DS, ma se la variabile V1 si trova nell'extra segment (segmento cui è stato assegnato il registro segmento ES) l'assemblatore utilizza un segment override prefix per specificare l'uso di ES.

• ASSUME è una direttiva e come tale non comporta l'esecuzione di alcuna istruzione. Di conseguenza, essa non provvede all'inizializzazione del registro segmento specificato. Questo è compito del programmatore (a parte CS, modificato da istruzione di salto e SS, inizializzato dal DOS).

• Vedi il programma `segmenti.asm` per sperimentare l'uso dei registri segmento e della direttiva ASSUME

Nota sul segmento STACK

```
STACK SEGMENT PARA STACK 'STACK'
;                       inizializzazione del segmento stack con 64 stringhe ‘STACK   ‘
;
                        DB          64 DUP ('STACK    ')
STACK ENDS
```

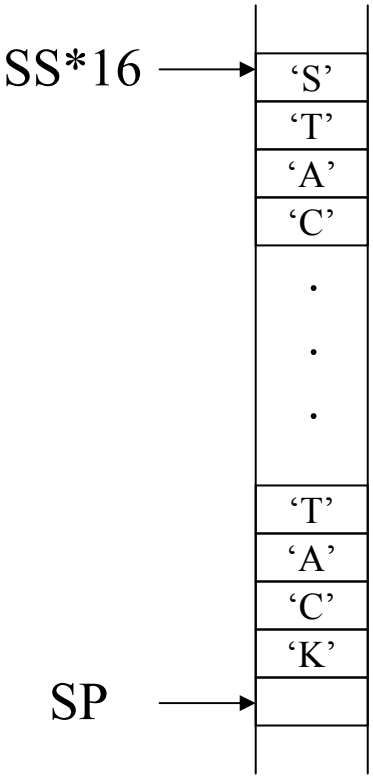


Il DOS inizializza:

- SS all’indirizzo dello stack
- SP in modo che punti alla prima posizione libera successiva allo stack

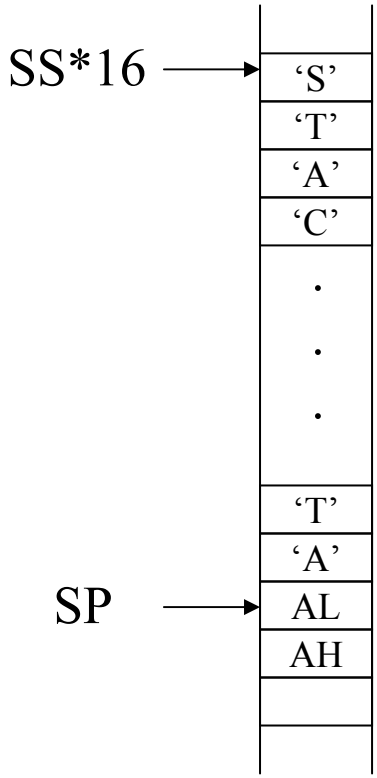
In tal modo le stringhe ‘stack’ sono sovrascritte dal push dei dati!

Nota sul segmento STACK (continua)



PUSH AX

- SP = SP-2
- M[SP] ← AX



Lo stack cresce verso indirizzi più bassi di memoria, decresce verso indirizzi più alti: l'effetto dell'istruzione DB nel segmento STACK è quello di allocare spazio!

Riassumendo

Abbiamo visto le direttive per definire i (frammenti di) segmenti.

- SEGMENT … ENDS:
  definisce un frammento di segmento

- ASSUME segment_reg:segment_name:
  informa l'assemblatore sul registro segmento associato in quel momento al segmento

Ora vediamo i simboli che il programmatore può definire:

- Etichette
- Variabili
- Costanti

Elementi del programma: istruzioni

```
INIT:   SUB    AX,  AX          ; azzera il registro AX
        MOV    AX, 18D          ; copia il valore 18 (decimale) in AX
```

Formato generale delle istruzioni:

[etichetta] simbolo mnemonico [operando][,operando] [; commento]

Etichetta: è un identificatore facoltativo che indica la locazione di partenza dell'istruzione (spesso usata come nome simbolico per i salti). Nell'es. l'etichetta INIT permette di avere salti all'istruzione sub

Simbolo mnemonico: è l'acronimo che indica una certa istruzione (mov, sub, add, etc.)

Operando: può non esserci, oppure ce ne può essere uno, oppure due separati da una virgola

Commento: tutto ciò che viene dopo il punto e virgola sulla stessa riga è un commento

Etichette

Etichetta: indica la locazione di partenza di una istruzione (spesso usata come nome simbolico per i salti). E' caratterizzata da:

- Un indirizzo (segmento + scostamento)
- Un attributo di distanza che può assumere due valori:
  - NEAR: può essere riferita solo dallo stesso segmento in cui si trova
  - FAR: può essere riferita anche da altri segmenti

  NB: è importante per distinguere le istruzioni di salto intersegment vs. intrasegment (modifica di CS o meno) e, come vedremo, nella definizione delle procedure.
- Un attributo di visibilità, riferita ai moduli, che può assumere tre valori:
  - INTERNAL: etichetta nota (e riferita) solo al modulo in cui è definita
  - PUBLIC: etichetta resa disponibile anche agli altri moduli
  - EXTERN: etichetta definita in un modulo diverso da quello corrente

Etichette: definizione dell'attributo distanza

Nell'esempio la definizione dell'etichetta era implicita:

```
INIT:    SUB    AX,  AX          ; azzera il registro AX
```

In questo caso per default l'etichetta è NEAR (e interna), ovvero può essere riferita solo all'interno del segmento e modulo in cui appare.

Per definire un'etichetta FAR è necessaria una definizione esplicita mediante la direttiva LABEL. Nell'esempio precedente:

```
INIT     LABEL FAR              ; dichiara l'etichetta INIT come FAR
         SUB    AX,  AX         ; azzera il registro AX
```

Variabili

Variabili: simboli mediante il quale il programmatore fa riferimento ai dati (diverse dalle etichette!!!)

Sono caratterizzate anch'esse da un indirizzo e da una visibilità, ma non da una distanza (nel loro accesso, non c'è la distinzione intrasegment vs. intersegment, poichè si fa sempre riferimento ad un dato registro segmento)

Come vedremo, sono invece caratterizzate da un tipo (es. Byte, Word, ecc.)

Di norma, sono definite nel segmento dati

Segmento dati

- Per la dichiarazione di variabili occorre scrivere il **segmento dati**, *prima del segmento codice*, ad esempio:

```
DATA    SEGMENT PARA PUBLIC 'DATA‘
;       dichiarazione delle variabili
DATA    ENDS
```

Dichiarazione di variabili: direttive DB, DW, DD

- Le direttive seguenti, poste nel segmento dati, effettuano la dichiarazione di due variabili, i e j, di tipo *byte*:

```
i       DB      ?       ; variabile i non inizializzata
j       DB      5       ; variabile j inizializzata al valore 5
```

- DW: è usato per dichiarare variabili di tipo *word*
- DD: è usato per dichiarare variabili di tipo *double word*
- Nelle versioni più recenti dell'assembler si possono usare byte e sbyte al posto di DB; word e sword al posto di DW; dword e sdword al posto di DD

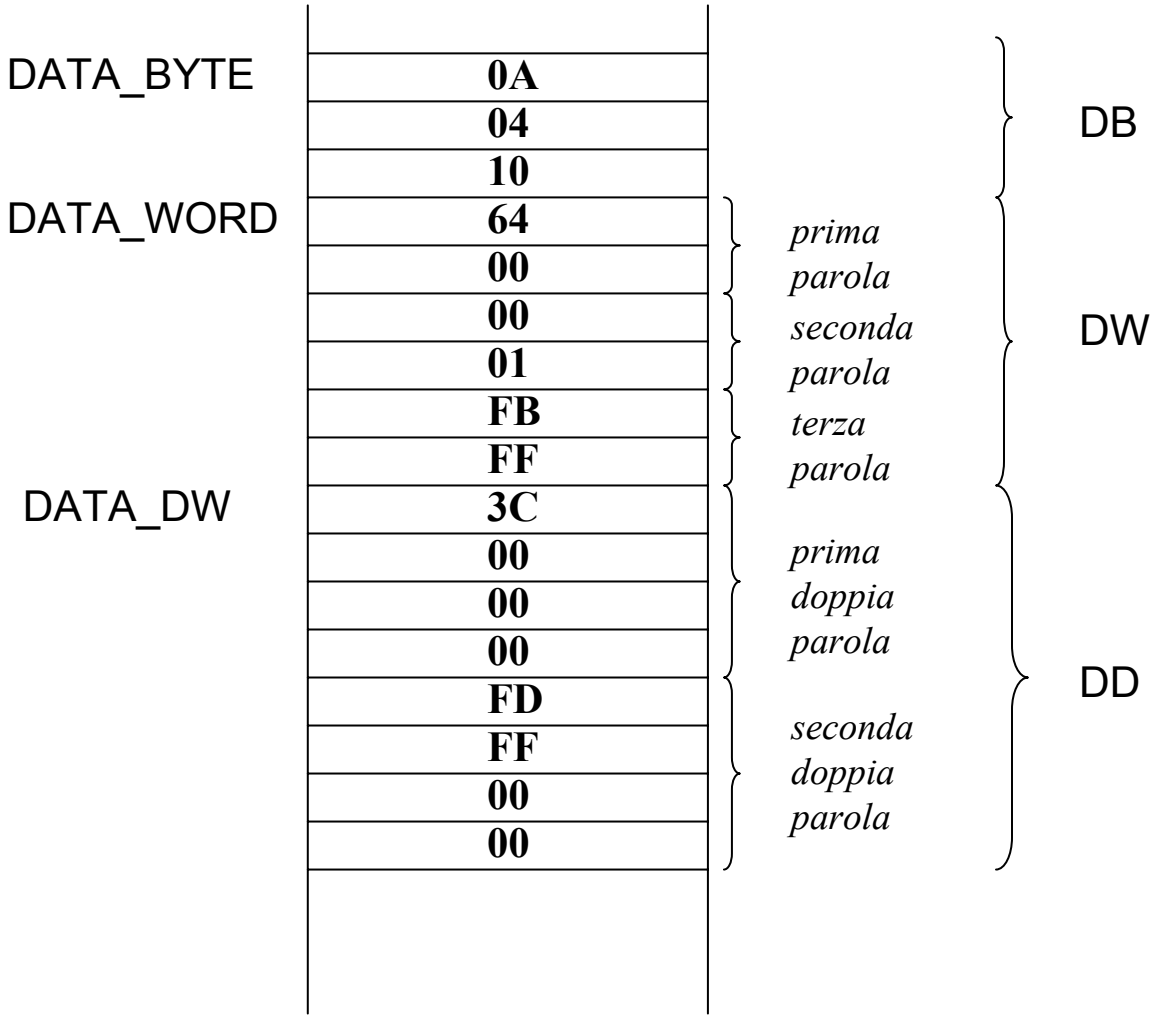
Direttive DB, DW, DD

- Le direttive DB, DW, DD sono anche usate per porre i dati in certe locazioni (si parla di *preassegnamento*) oppure solamente per allocare spazio in memoria
- Esempi:

```
DATA_BYTE    DB      10, 4, 10h
DATA_WORD    DW      100, 100H, -5
DATA_DW      DD      3*20,0FFFDh
```

- Questo causa il preassegnamento di una sequenza di byte (si veda figura successiva)
- DATA_BYTE, DATA_WORD, DATA_DW sono variabili: a ciascuna di essere è assegnato l'offset del primo byte riservato in memoria dalla corrispondente direttiva

Preassegnamento di dati usando DB, DW, DD



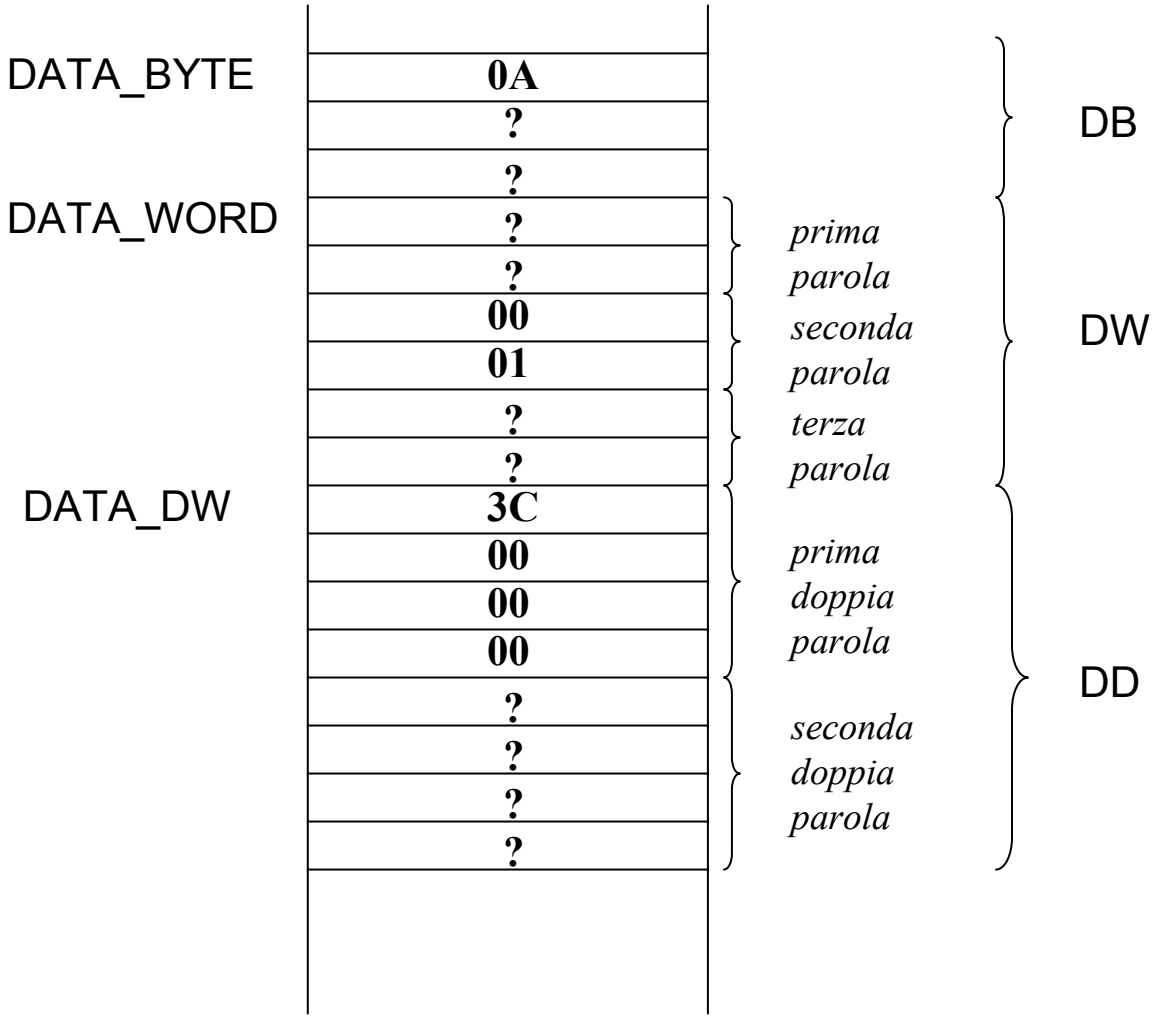
Allocazione senza preassegnamento

- E’ possibile fare l’allocazione dello spazio in memoria senza fare il preassegnamento
- Occorre inserire al posto degli operandi dei punti interrogativi (?)
- Ad esempio, se nei comandi precedenti si facesse

```
DATA_BYTE   DB    10, ?, ?
DATA_WORD   DW    ?, 100H, ?
DATA_DW     DD    3*20,?
```

- Verrebbe allocato lo stesso ammontare di spazio visto prima (3 byte per DATA_BYTE, 6 byte per DATA_WORD e 8 byte per DATA_DW)

Preassegnamento e allocazione con DB, DW, DD



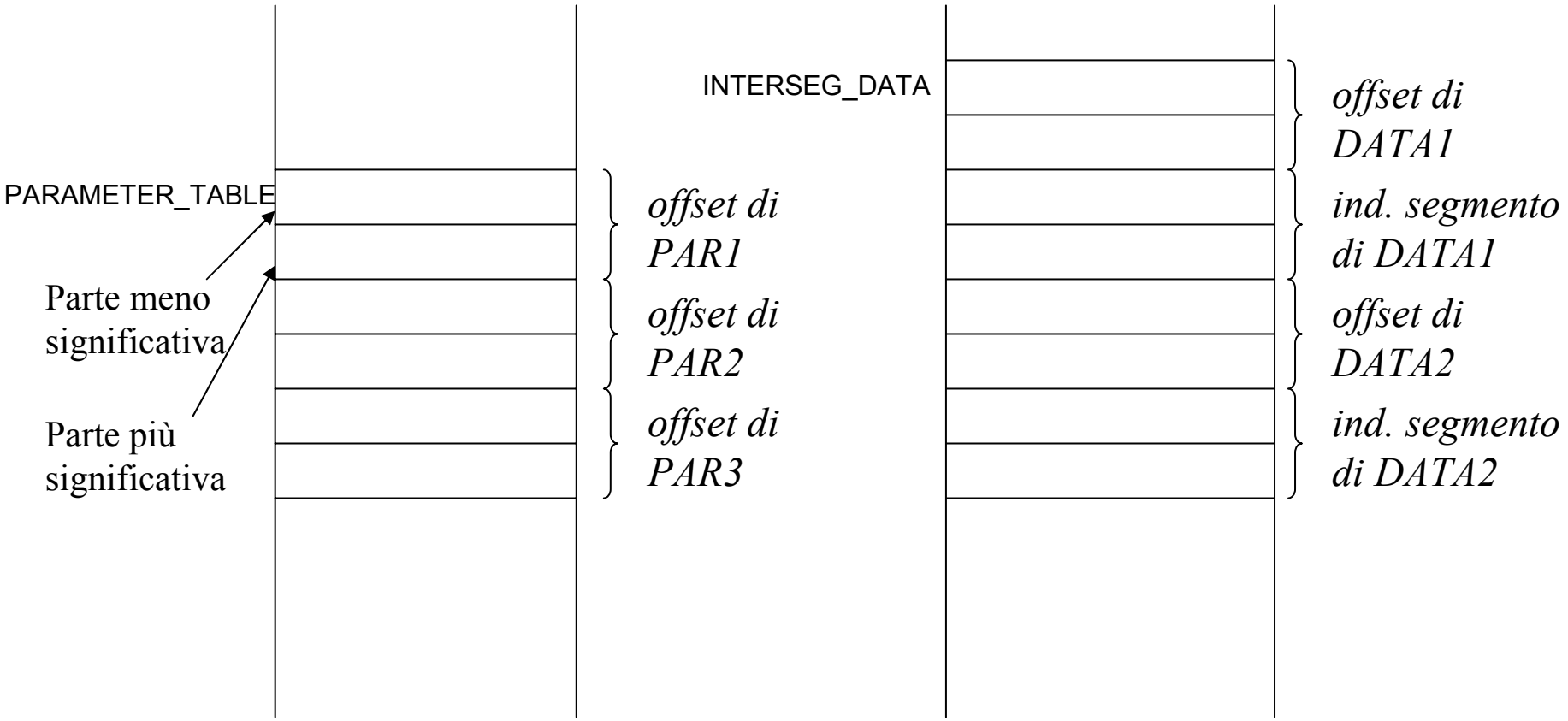
Preassegnamento di un offset o di un indirizzo completo

- E' possibile preassegnare l'offset o l'indirizzo completo di una variabile o di un'etichetta
- Se l'operando è in un comando DW solo l'offset può essere memorizzato
- Se l'operando è in un comando DD viene memorizzato sia l'offset che l'indirizzo di segmento, e l'offset viene messo nella prima parola
- Esempi (PAR*i* e DATA*i* sono variabili o etichette):

```
PARAMETER_TABLE     DW      PAR1
                    DW      PAR2
                    DW      PAR3

INTERSEG_DATA       DD      DATA1
                    DD      DATA2
```

Preassegnamento di un offset o di un indirizzo completo: esempi



Stringhe

- Per fare il preassegnamento di stringhe di caratteri ASCII si può scrivere:

  ```
  MESSAGE   DB      'C','I','A','O'
  ```

  oppure, equivalentemente

  ```
  MESSAGE   DB      'CIAO'
  ```

- Questo comando pone i codici ASCII corrispondenti alle lettere C, I, A, O in byte consecutivi a partire dal byte il cui indirizzo è associato alla variabile MESSAGE

Dichiarazione di array

- Dichiarazione dell'array ArrayA con 10 elementi non inizializzati:

```
ArrayA      DB      10 DUP (?)
```

- Dichiarazione dell'array ArrayB con 10 elementi inizializzati al valore 1:

```
ArrayB      DB      10 DUP (1)
```

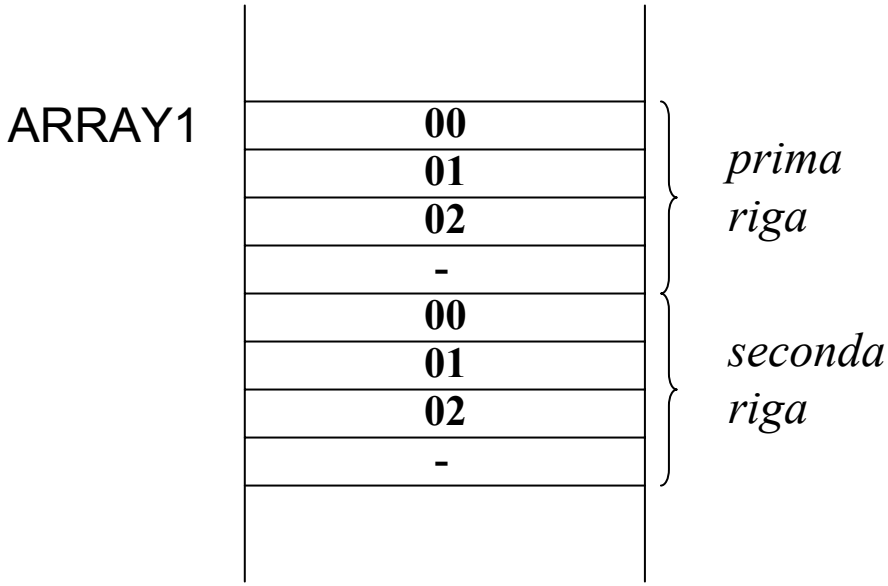
- Dichiarazione dell'array ArrayC con 11 elementi inizializzati con i quadrati dei valori da 0 a 10:

```
ArrayC      DB      0, 1, 4, 9, 16, 25, 36
            DB      49, 64, 81, 100
```

Altri esempi di dichiarazioni di array

```
ARRAY1          DB        2 DUP (0,1,2,?)
```

- Vengono allocati 8 byte per ARRAY1, si può parlare in questo caso di array a 2 dimensioni: 2×4 (2 righe e 4 colonne)



Costanti

- Ovviamente, è possibile usare valori costanti, che possono indicare numeri interi e stringhe:
  - 1011B numero binario
  - 223D numero decimale
  - 0B25AH numero esadecimale
  - 087O numero ottale
  - ‘Axy’ stringa

- E’ possibile, mediante operatori logici e aritmetici, costruire espressioni costanti:
  - 2 + 3
  - 3 * (100B + 087O)
  - ecc.

Espressioni (costanti)

- Se un'espressione compare diverse volte in un programma conviene darle un nome attraverso il seguente comando:

  nome_espressione EQU *espressione*

- Ad esempio, se si è definito

```
DECIMO_CHAR  EQU   CHAR_ARRAY[SI+10]
```

- Si può poi fare

```
MOV          AL, DECIMO_CHAR
```

- Altri esempi:

```
ALPHA        EQU   7
BETA         EQU   ALPHA+1
ADDR         EQU   VAR+BETA
```

- E' bene definire prima i dati usati dai comandi EQU: nell'esempio sopra, prima di definire ADDR occorre definire VAR e BETA
- I comandi EQU andrebbero messi, per evitare errori, dopo il data segment e prima del code segment

Riassumendo

Abbiamo visto i simboli che il programmatore può definire:

- Etichette:
  indicano la locazione di una istruzione
- Variabili:
  indicano la locazione di un dato
- Costanti:
  indicano un valore

Dato un identificatore, l'assemblatore è ovviamente in grado di riconoscere se è un'etichetta, variabile o costante.

Ora vediamo come si esprimono in assembler le modalità di indirizzamento ai dati (es. accesso al valore di una variabile)

Modalità di indirizzamento ai dati

1. **Immediate**: *Espressione costante*.
   Esempi:

```
ADD AX, 18D          ; sommo ad AX il decimale 18
ADD AX, CONST        ; sommo ad AX la costante CONST
```

2. **Direct:** *Variabile ± Espressione costante*
   Esempi:

```
MOV AX, ARRAY        ; sommo ad AX il primo elemento dell'array
MOV AX, ARRAY+2      ; sommo ad AX il secondo elemento
```

3. **Register:** *Registro*
   Esempio:

```
ADD AX, BX
```

4. **Register indirect:** *[Registro]*
   Esempio:

```
ADD AX, [BX]
```

Modalità di indirizzamento ai dati (2)

5. **Register relative:** *Variabile[Registro ± Espressione costante]* o *[Registro ± Espressione costante]*

Esempi:

```
MOV AX, VAR_X[BX]      ; EA= BX + offset di VAR_X
MOV AX, [BX +1]        ; EA = BX +1
MOV AX, VAR_X[BX +1]   ; EA = BX + 1 + offset di VAR_X
```

6. **Based indexed:** *[Registro base][Registro indice]*

Esempio:

```
MOV AX, [BP][DI]       ; EA= BP + DI
```

7. **Relative based indexed:**

Variabile[Reg. Base ± Espressione costante] [Reg. Indice ± Espressione costante]

oppure

[Reg. Base ± Espressione costante] [Reg. Indice ± Espressione costante]

Esempi:

```
MOV AX, VAR[BX+5][SI-2]      ; EA=offset(VAR)+BX+5+SI-2
MOV AX, [BP+2][DI-9]         ; EA=BP+2+DI-9
```

Modalità di indirizzamento ai dati (3)

- In pratica, una variabile indica il relativo OFFSET (o EA)
  rispetto al segmento di appartenenza
- Le costanti vengono sommate per generare l'OFFSET complessivo
- Tutti i calcoli sono fatti in byte!

```
MOV AX, ARRAY+2          ; sommo ad AX il secondo elemento
```

In questo caso l'offset calcolato è OFFSET(ARRAY) +2: poiché l'array è costituito di parole, tale OFFSET corrisponde al secondo elemento.

Ruolo del tipo delle variabili

- La variabile associata ad una direttiva DB, DW o DD rappresenta il primo item ed è rispettivamente di tipo Byte, Word o Double_Word. P.es.

```
ARRAYA      DB       2 DUP (0,1,2,?)
```

ARRAYA è di tipo byte ed indica il primo elemento (di valore 0).

- Il tipo è usato per determinare se le istruzioni operano sul byte o word, inoltre per effettuare un certo controllo sui tipi.

Esempio 1

```
OPER1  DB       ?,?
OPER2  DW       ?,?
…
…
MOV OPER1, 0            ; carica in OPER1 il byte 0
MOV OPER2, 0            ; carica in OPER2 la parola 0
```

Ruolo del tipo delle variabili (continua)

Esempio 2

```
OPER1  DB      ?,?
OPER2  DW      ?,?
…
…
MOV OPER1, AX                ; genera errore: word in byte
MOV OPER2, AL                ; genera errore: byte in word (improbabile)
```

NB: E' però possibile forzare esplicitamente il tipo di un operando: la seconda istruzione precedente può essere ammissibile $\Rightarrow$ operatore PTR

```
MOV BYTE PTR OPER2, AL       ; opera sul primo byte della variabile
                             ; OPER2 di tipo WORD
```

Due operatori utili: OFFSET e SEG

- OFFSET etichetta oppure
  OFFSET variabile

  Restituisce l'offset di una etichetta o variabile rispetto al proprio segmento

Esempio

```
MOV BX, OFFSET VAR        ; indirizzamento immediato, non diretto!!!
MOV AX, [BX]              ; carica in AX il valore della variabile VAR
```

- SEG etichetta oppure
  SEG variabile

  Restituisce l'indirizzo di segmento (16 bit) cui appartiene una etichetta o variabile

Inizializzazione del registro DS

- All'inizio del programma, il registro DS ha in generale un valore indefinito (mentre CS e SS sono già inizializzati).
- E' necessario quindi caricare il registro DS con l'indirizzo del segmento dati usato dal programma utente
- Supponendo che il segmento dati sia definito come

```
DATA SEGMENT  …
…
DATA ENDS
```

è sufficiente fare:

```
MOV   AX, SEG DATA
MOV   DS, AX
```

dove SEG serve per indicare l'indirizzo di partenza del segmento indicato (DATA nel nostro caso)

- Occorrono due istruzioni perché mov non può copiare un operando immediato in un registro segmento
- Naturalmente, ciò va fatto anche per ES se è previsto l'uso di un segmento dati addizionale

Esempio di programma

```
TITLE inutile2.asm - Primo programma Assembler con definizione di segmento dati

STACK SEGMENT PARA STACK 'STACK'
;               inizializzazione del segmento stack con 64 stringhe ‘STACK   ‘
;
                DB          64 DUP ('STACK     ')
STACK ENDS
;
DATA SEGMENT PARA PUBLIC  'DATA‘
;               inizializzazione del segmento dati con una variabile I = 5 di tipo byte
     I          DB          5
DATA ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
                ASSUME  CS:CSEG, SS:STACK, DS:DATA
;
INIT:
                mov         AX,  SEG DATA       ; uso SEG con il nome del segmento
                mov         DS, AX              ; DS contiene indirizzo di segmento dati
;
                mov         I, 6                ; pone I = 6 (operazione byte)
;
                mov         AH, 4ch             ; terminazione programma e ritorno al DOS
                int         21H
CSEG        ENDS
                END
```