Il Linguaggio Assembler

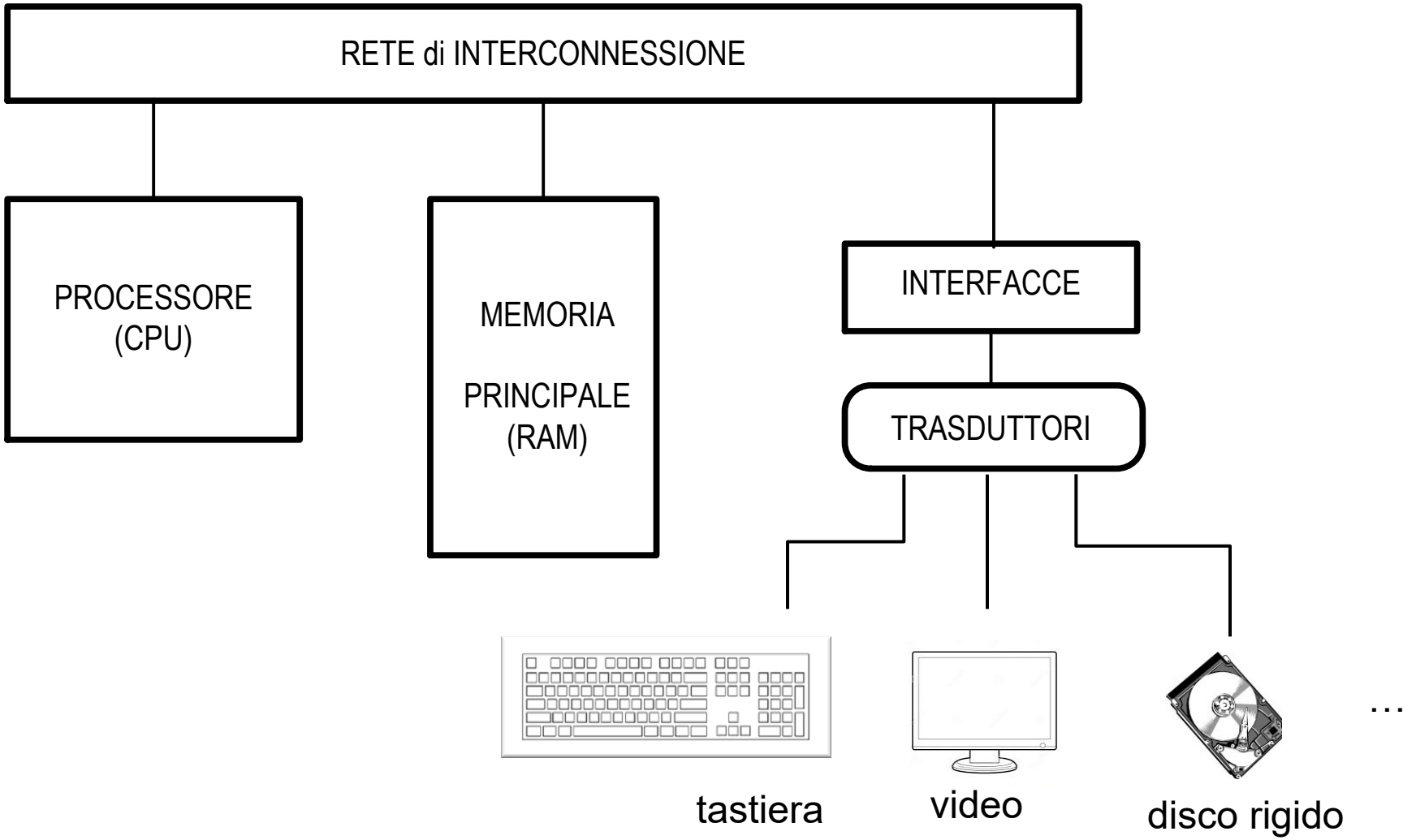
A cura del prof. Marco Cococcioni

aa. 2019-2020

Testo di riferimento

Schema a blocchi di un semplice calcolatore

Architettura di von Neumann (1946)



Funzionamento

La memoria contiene dati e programmi (istruzioni) codificati in forma binaria

Il processore ripete all'infinito le azioni seguenti :

- preleva una nuova istruzione dalla memoria
- la decodifica
- la esegue

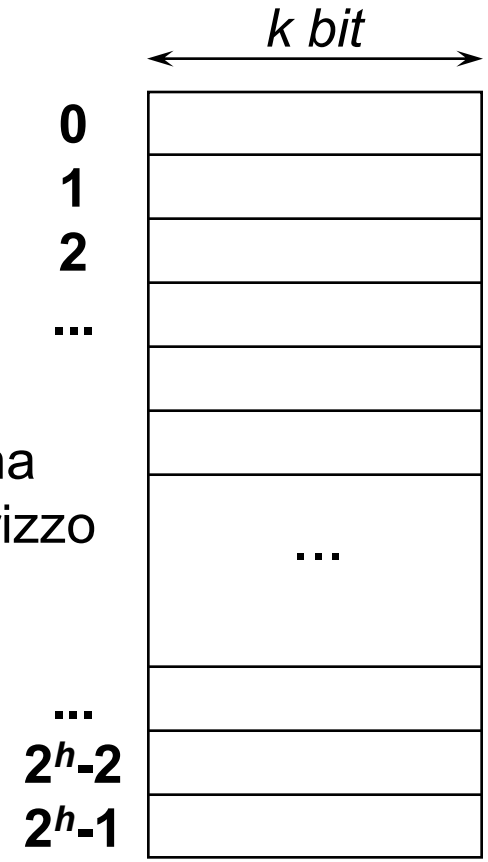
  L'esecuzione di un'istruzione può comportare

  » Elaborazione e/o Trasferimento (memoria $\leftrightarrow$ processore, I/O $\leftrightarrow$ processore)

Le periferiche permettono al calcolatore di interagire con il mondo esterno

Struttura logica della memoria

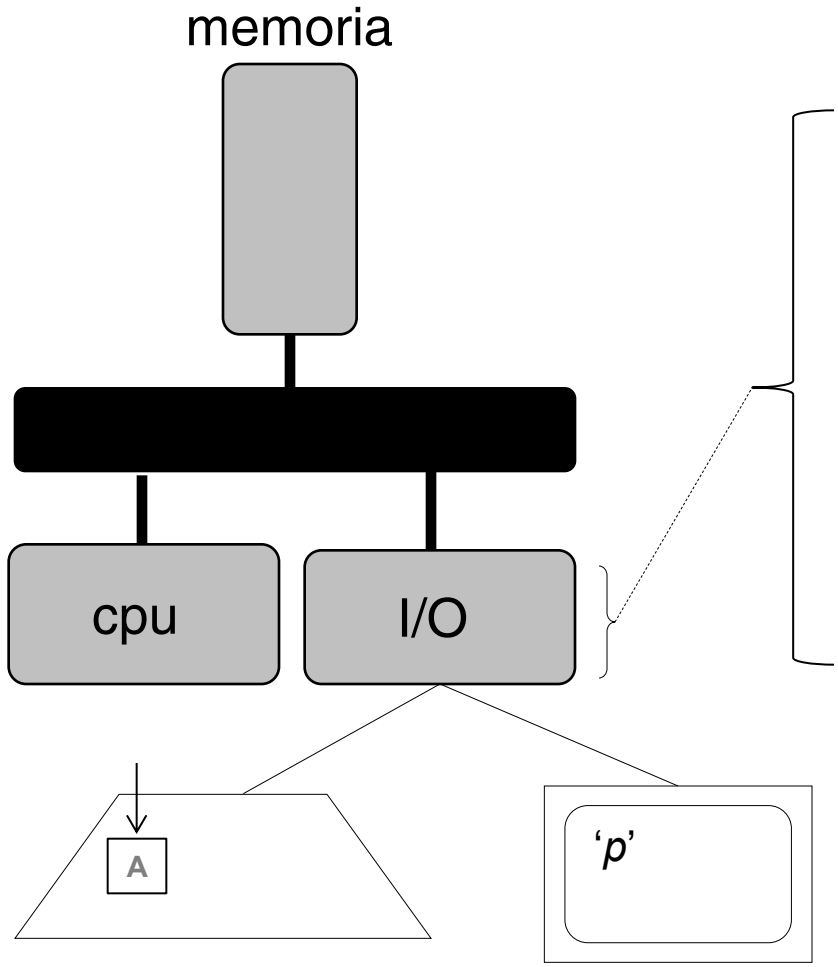
- 2^h *celle* di *k* bit ognuna
- Ogni cella ha un indirizzo



OPERAZIONI

1. LETTURA di UNA cella
2. SCRITTURA di UNA cella

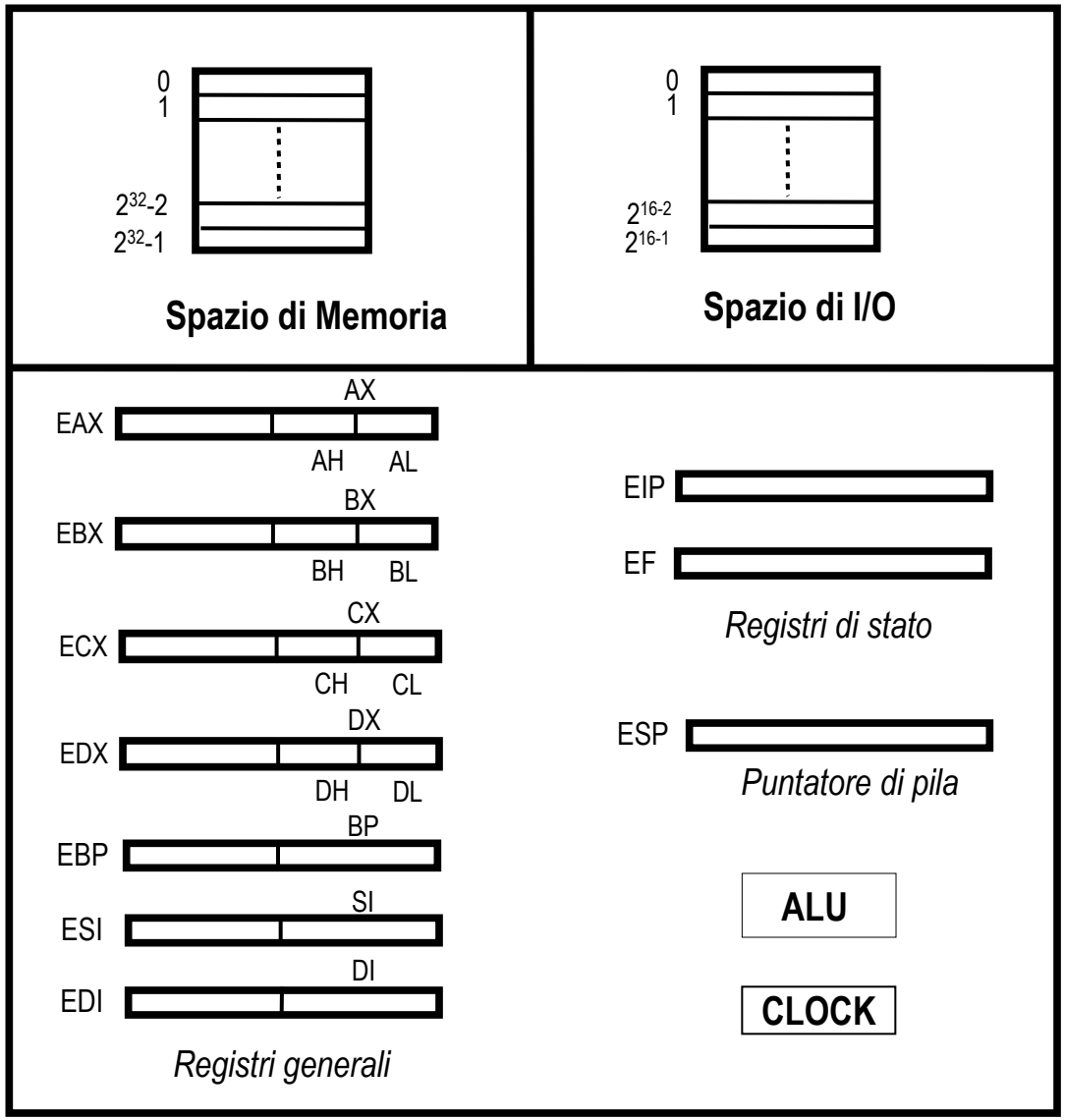
Lo spazio di ingresso/uscita



	...	...	
	(C1CF)	10010001	← Porta di IN
video	(C1D0)	*01110000*	← Porta di OUT
	(C1D1)	10100101	← Porta di OUT
	(C1D2)	00100101	← Porta di OUT
tastiera	(C1D3)	***01000001***	← Porta di IN
	(C1D4)	01011010	← Porta di OUT
	...	...	

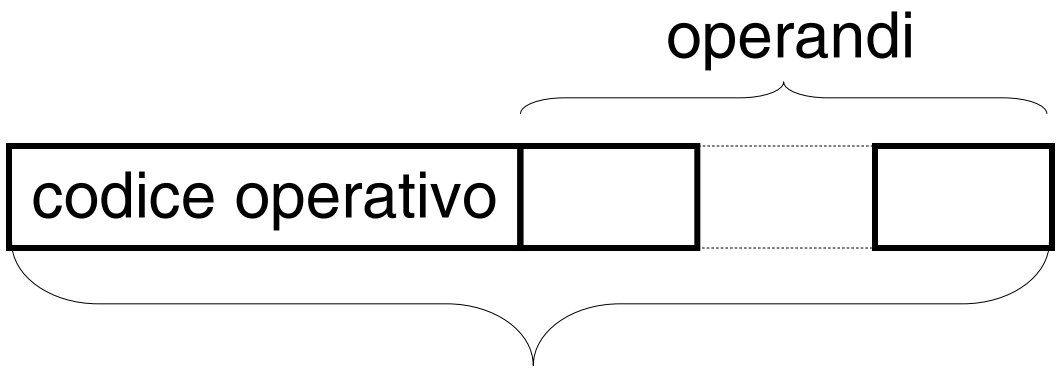
```
IN (C1D3), %AL        # carica in AL l'esadecimale 41, ossia 'A' (vedi cod. ASCII)
OUT $0x70, (C1D0)     # stampa a video 0x70, ossia il carattere 'p'
```

Struttura logica del processore (1)



Formato delle istruzioni (1/2)

Le istruzioni sono codificate come sequenze di bit.



Esempio

Codice Operativo	**Operandi**	
`10001011`	`01100100`	`00000001`
MOV	immediato 0x64	%AL

Esempio di programma in lunguaggio assembler

```
3F0D0100    00000000                Dato da elaborare (una WORD)
3F0D0102    00000000                Risultato (un BYTE)
3F0D0103
...
...
40B1A200    MOV   $0, %AL           Azzera  il contatore AL
40B1A203    MOV   (3F0D0100), %DX   Copia in DX il dato da elaborare
40B1A207    CMP   %DX, $0           Confronta il contenuto di DX con 0
40B1A211    JE    40B1A232          Salta se uguale
40B1A215    SHL   %DX               Trasla a sinistra il contenuto di DX
40B1A217    JC    40B1A225          Salta se CF è settato all'indirizzo 225
40B1A221    JMP   40B1A207          Salta incondizionatamente a 207
40B1A225    ADD   $1, %AL           Incrementa il contatore AL
40B1A228    JMP   40B1A207          Salta incondizionatamente
40B1A232    MOV   %AL, (3F0D0102)   Memorizza il risultato
40B1A236    HLT                     ALT
…
```

Livelli di astrazione dei Linguaggi

Esistono linguaggi a vari livelli di astrazione

Linguaggio macchina — sequenze di bit (difficile da leggere e capire)

```
0001001000110100
```

Linguaggio Assembler — istruzioni macchina espresse con nomi simbolici (dipendente dalla macchina)

```
MOV $35, %AL
```

Linguaggi ad Alto livello indipendenti dalla macchina

```
int main()
{
    …
}
```

Istruzioni MOV e JMP

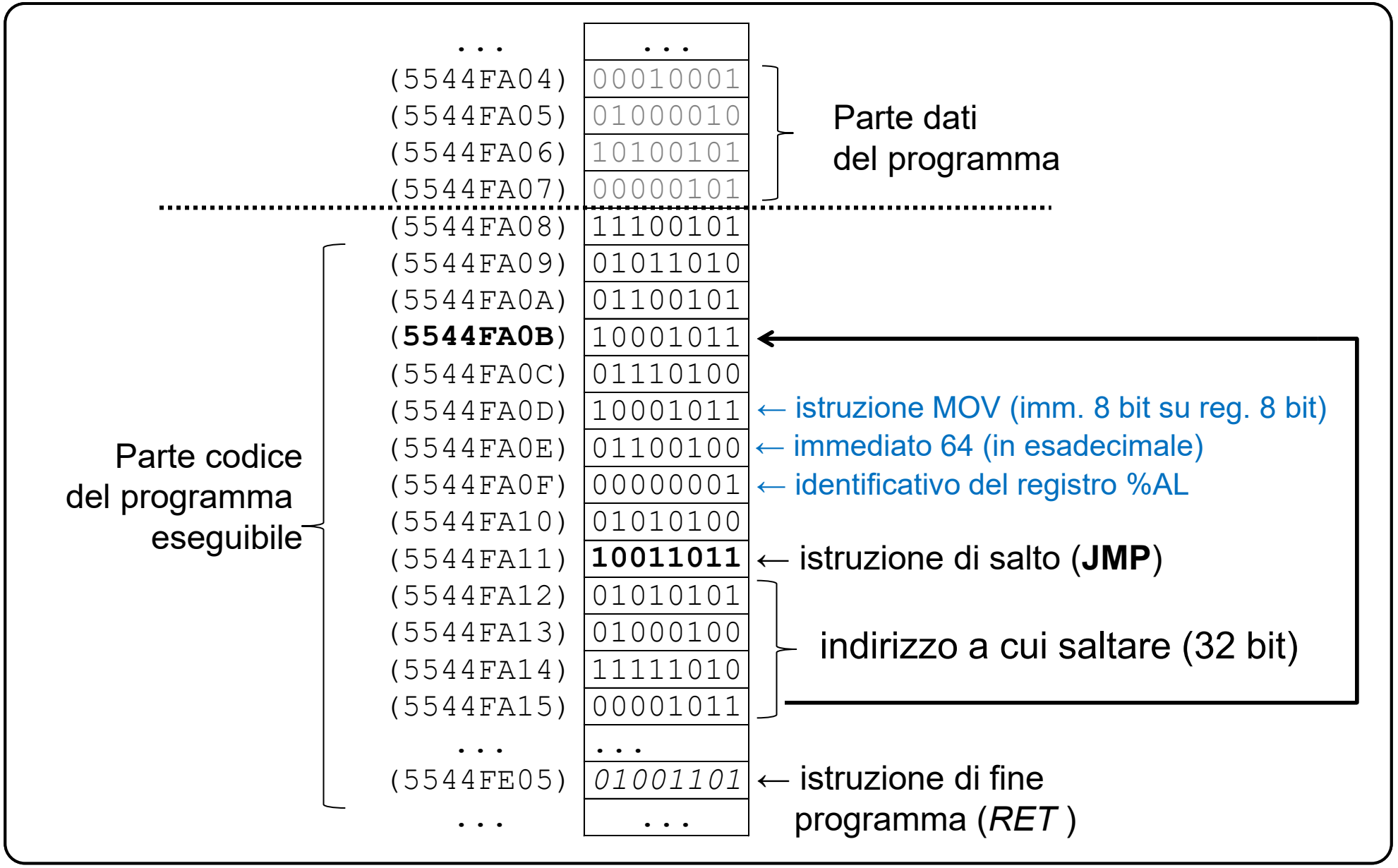
L'istruzione MOV del processore DP.86.32 permette di:

- Copiare un valore (immediato) in un registro
  Es: MOV $0x64, %AL
- Copiare il contenuto di una cella di memoria in un registro:
  Es: MOV (5544FA04), %BH
- Copiare il contenuto della cella puntata da un registro a 32 bit in un altro registro a 8 bit:
  Es: MOV (%EBX), %CH
- Copiare il contenuto delle **2 celle consecutive** di cui la prima è puntata da un registro a 32 bit in un altro registro a 16 bit:
  Es: MOV (%EDX), %BX
- Copiare il contenuto delle **4 celle consecutive** di cui la prima è puntata da un registro a 32 bit in un altro registro a **32 bit**:
  Es: MOV (%ECX), %EAX

L'istruzione JMP del processore DP.86.32 permette di saltare all'indirizzo (sempre a 32 bit) specificato:

Es: JMP 5544FA0B

Esempio di programma in memoria

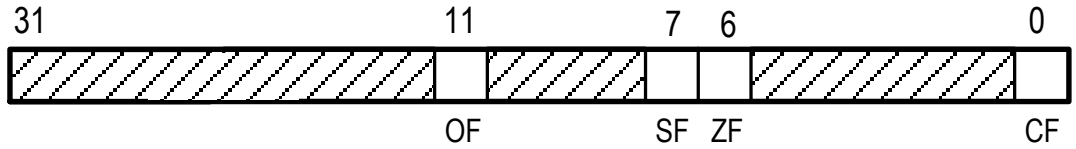


Controllo del flusso: salto condizionato e registro dei flag

Registri di stato:

Extended Instruction Pointer EIP: registro che contiene l'indirizzo della prossima istruzione da eseguire

Extended Flag register EF



Carry Flag CF, Zero Flag ZF

SignFlag SF, Overflow Flag OF

Esempio di test del contenuto dei flag:

Es1: CMP $0, %AL

JZ *indirizzo_di_memoria*

Es2: ADD %AL,%BL

JO *indirizzo_di_memoria*

NB:

Alcune istruzioni influenzano i flag, altre no

Ad esempio: la ADD influenza sia CF che OF, ma

- CF va testato nel caso si lavori sui naturali
- OF va testato nel caso si lavori sugli interi

Labels

Ogni istruzione assembler inizia ad un certo indirizzo in memoria (l'indirizzo della cella contenente il primo byte dell'istruzione)

L'indirizzo assoluto di una istruzione sarà noto solo a tempo di esecuzione, quando il programma assembler, reso eseguibile dall'assemblatore e dal linker, verrà caricato in memoria.

Il programmatore può creare delle etichette (dette *labels*), che possono essere utilizzate dal programmatore per indicare l'indirizzo di una istruzione. Sarà compito dell'assemblatore e del linker sostituire l'etichetta con l'indirizzo fisico assoluto.

Esempio di un programma che stampa 5 asterischi a video (il sottoprogramma *output* stampa a video il carattere la cui codifica ASCII si trova in %AL):

```
_main:    MOV   $5, %AH
          MOV   $'*', %AL
label:    CALL  output        # label conterrà l'indirizzo dell'istruzione
          DEC   %AH           # CALL output
          JNZ   label
          RET
.INCLUDE "utility"
```

Le direttive .BYTE, .FILL e .ASCII

In assembler si può allocare memoria «statica» per delle «variabili». Si tratta dell'equivalente C/C++ delle variabili globali.

Una «variabile» assembler è una etichetta (che corrisponde all'indirizzo della prima cella di memoria in cui inizia la «variabile») seguita da una direttiva assembler che riserva area di memoria.

```
v1:   .BYTE 0x6F,'k' # alloca un vettore di 2 byte con "ok"
v2:   .FILL   12, 1  # alloca un vettore di 12 byte (1=byte)
v3:   .FILL   25, 2  # alloca un vettore di 25 word (2=word = 2byte)
v4:   .FILL   10, 4  # alloca un vettore di 10 long (4=long = 4byte)
str:  .ASCII "Ciao Mondo"

_main:
      MOV   $str, %EBX  # copia l'indirizzo di str in EBX
      MOV   $10,  %CX
      CALL  outmess
      RET
```

NB: Le direttive .BYTE e .ASCII calcolano il numero di celle di memoria da allocare ed inoltre provvedono anche alla loro inizializzazione. La .FILL non inizializza.

Istruzioni operative (1)

Istruzioni per il trasferimento di dati

MOV	Movimento (ricopiamento)
IN	Ingresso dati
OUT	Uscita dati
PUSH	Immissione di un long (32 bit) nella pila
POP	Estrazione di un long (32 bit) dalla pila

Istruzioni aritmetiche

ADD	Somma (fra interi oppure naturali)
SUB	Sottrazione (fra interi oppure naturali)
CMP	Confronto (fra interi oppure naturali)
MUL	Moltiplicazione (fra naturali)
DIV	Divisione (fra naturali)
IMUL	Moltiplicazione (fra interi)
IDIV	Divisione (fra interi)

Istruzioni operative (2)

Istruzioni logiche

NOT	Not logico bit a bit
AND	And logico bit a bit
OR	Or logico bit a bit

Istruzioni di traslazione/rotazione

SHL	Traslazione a sinistra
SHR	Traslazione a destra
ROL	Rotazione a sinistra
ROR	Rotazione a destra

Istruzioni di controllo

Istruzioni di salto

JMP	Salto incondizionato
Jcond	Salto sotto condizione

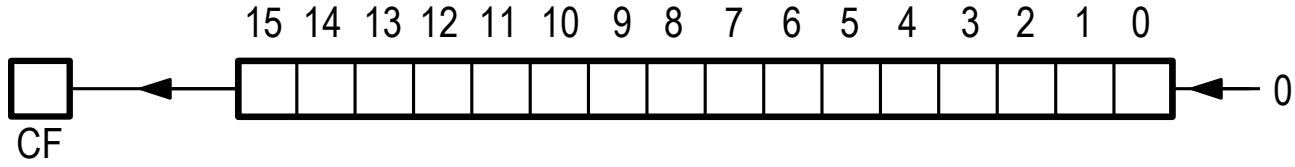
Istruzioni per la gestione dei sottoprogrammi

CALL	Chiamata di sottoprogramma
RET	Ritorno da sottoprogramma

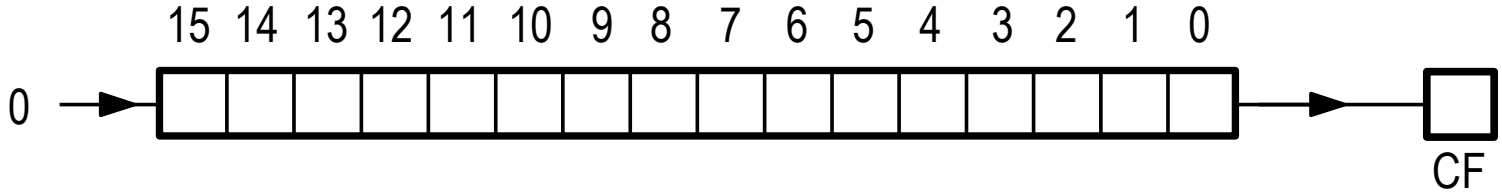
Istruzione di alt

HLT	Alt

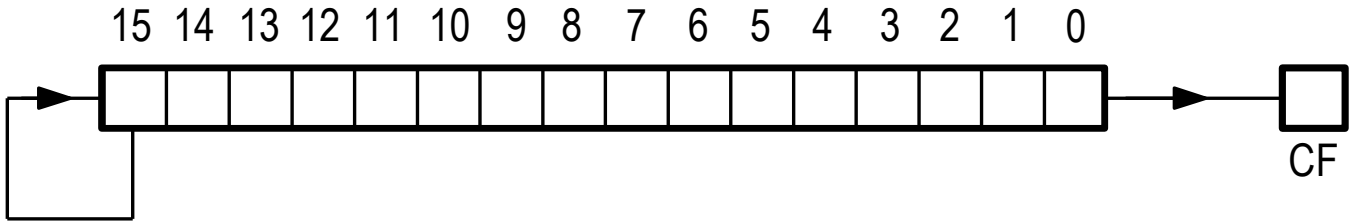
SHL, SAL (shift logico/aritmetico, left)



SHR (shift logico right)

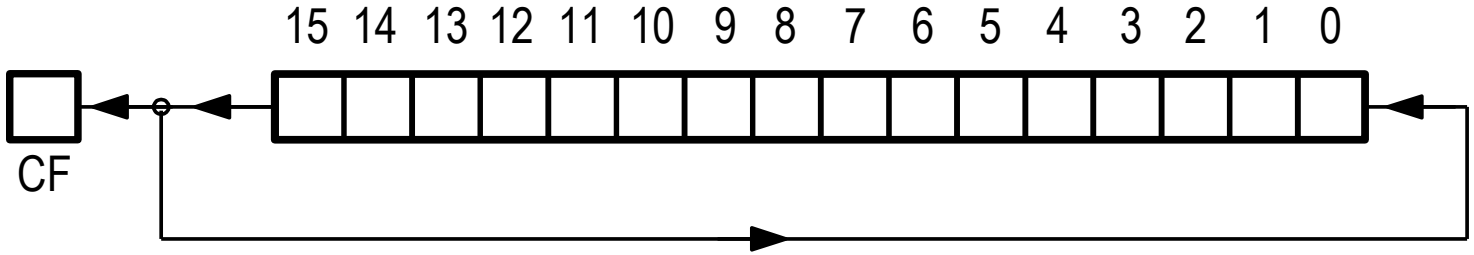


SAR (shift aritmetico right)

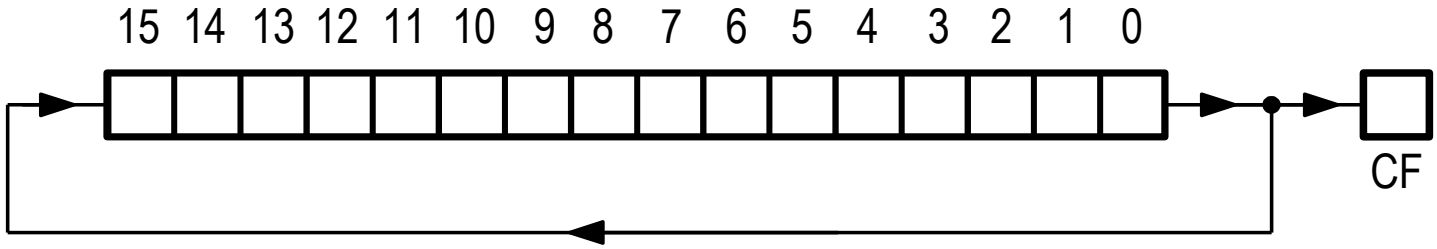


Istruzioni operative (4)

ROL



ROR



Istruzioni di somma e sottrazione

FORMATO: **ADD** *source*, *dest* # dest ← dest + source
SUB *source*, *dest* # dest ← dest - source

AZIONE: Modificano l'operando destinatario sommandovi (istruzione **ADD**) o sottraendovi (istruzione **SUB**) l'operando sorgente; il risultato dell'operazione è consistente **sia** se gli operandi sono interpretati come **numeri naturali sia** se gli operandi sono interpretati **come numeri interi**. Mettono a 1 il contenuto del flag CF se, interpretando gli operandi come numeri naturali, si è verificato un riporto (istruzione **ADD**) o un prestito (istruzione **SUB**); mettono a 1 il contenuto del flag OF se, interpretando gli operandi come numeri interi, si è verificato un **traboccamento**.

FLAG di cui viene modificato il contenuto: ***tutti***

Operandi	**Esempi**
Memoria, Registro Generale	OPCODE 0x00002000,EDX
Registro Generale, Memoria	OPCODE CL,0x12AB1024
Registro Generale, Registro Generale	OPCODE AX,DX
Immediato, Memoria	OPCODEB $0x5B,(EDI)
Immediato, Registro Generale	OPCODE $0x54A3,AX

CMP: CONFRONTA DUE OPERANDI

FORMATO: **CMP** *source*, *destination*

AZIONE: Verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, **sia interpretando gli operandi come numeri naturali che come numeri interi**. Aggiorna poi il contenuto dei flag tenendo conto del risultato della verifica (i due operandi rimangono inalterati).

FLAG di cui viene modificato il contenuto: ***tutti***

Operandi	**Esempi**
Memoria, Registro Generale	CMP 0x00002000,EDX
Registro Generale, Memoria	CMP CL,0x12AB1024
Registro Generale, Registro Generale	CMP AX,DX
Immediato, Memoria	CMPB $0x5B,(EDI)
Immediato, Registro Generale	CMP $0x45AB54A3,EAX

Istruzioni di salto condizionato J*cond* (1)

JE	(Jump if Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era uguale all'operando sorgente.
JNE	(Jump if Not Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario non era uguale all'operando sorgente.
JA	(Jump if Above) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
JAE	(Jump if Above or Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
JB	(Jump if Below) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
JBE	(Jump if Below or Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era minore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
JG	(Jump if Greater) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.

Istruzioni di salto condizionato J*cond* (2)

JGE	(Jump if Greater or Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
JL	(Jump if Less) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
JLE	(Jump if Less or Equal) Segue una istruzione CMP e la condizione per effettuare il salto è soddisfatta se l'istruzione CMP ha verificato che l'operando destinatario era minore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
JZ	(Jump if Zero) La condizione per effettuare il salto è soddisfatta se il risultato dell'istruzione precedente è stato zero (cioè se ZF contiene 1).
JNZ	(Jump if Not Zero) La condizione per effettuare il salto è soddisfatta se il risultato dell'istruzione precedente è stato diverso da zero (cioè se ZF contiene 0).
JC	(Jump if Carry) la condizione è soddisfatta se CF contiene 1.
JNC	(Jump if No Carry) la condizione è soddisfatta se CF contiene 0.
JO	(Jump if Overflow) la condizione è soddisfatta se OF contiene 1.
JNO	(Jump if No Overflow) la condizione è soddisfatta se OF contiene 0.
JS	(Jump if Sign) la condizione è soddisfatta se SF contiene 1.
JNS	(Jump if No Sign) la condizione è soddisfatta se SF contiene 0.

Esempio di uso combinato della CMP e della J*cond*

Esempio di utilizzo dell'istruzione CMP
Il seguente progamma stampa a video 5 asterischi, usando l'istruzione CMP ed quella di salto condizionato JE

```
_main: MOV $0, %BL
       MOV $'*', %AL
loop:  CALL output
       INC %BL
       CMP $5,%BL
       JE fine
       JMP loop
fine:  RET

.include "utility"
```

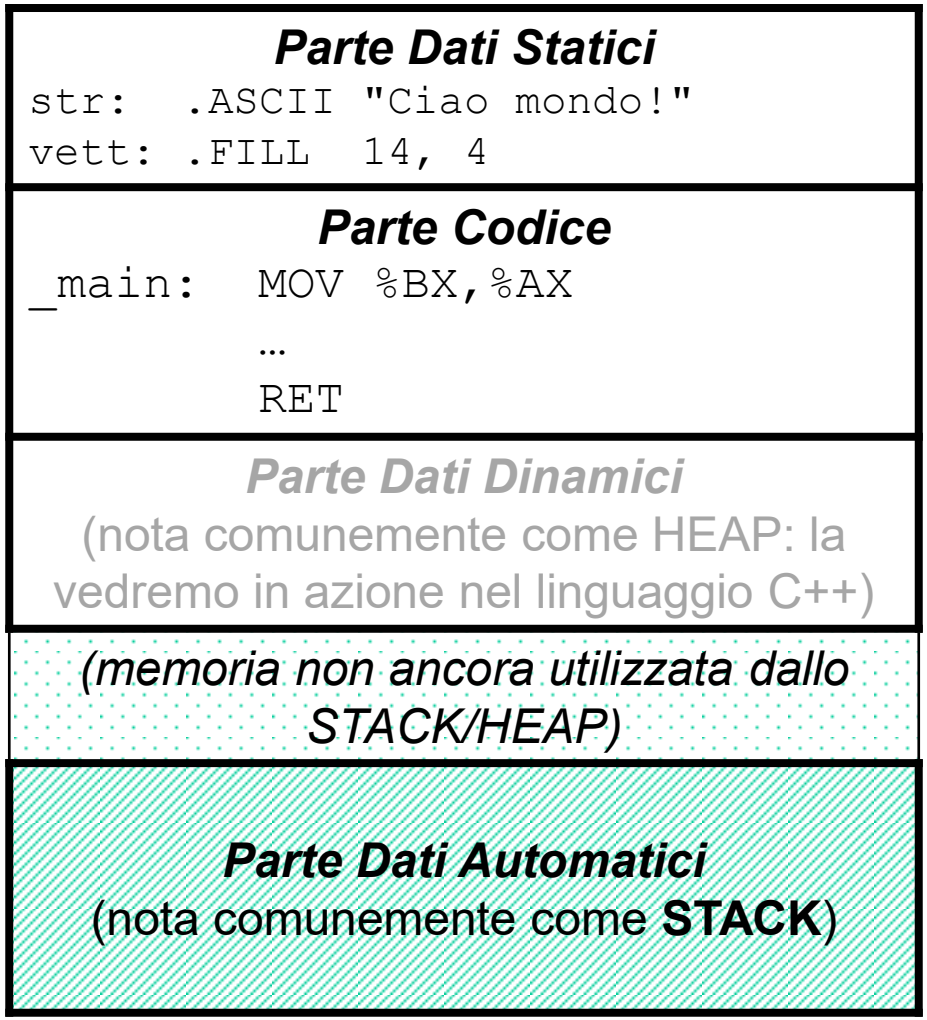
JE *label* # jump if equal

Comportamento:
Provoca il salto a *label* solo nel caso in cui i due operandi confrontati dalla precedente istruzione CMP erano uguali

Lo Stack (1/7)

- Lo STACK è un'area di memoria a disposizione dei programmi (sia Assembler che C/C++)
- Viene gestita mediante strategia **LIFO** (last in-first out): un nuovo dato può essere immesso oppure prelevato dalla cima allo STACK
- Sfrutta una registro speciale del processore, l'Extended Stack Pointer (ESP)
- **L'ESP punta sempre alla prima locazione libera dello STACK**

Struttura di un programma eseguibile in memoria



Le istruzioni PUSH e POP

- L'istruzione PUSH salva il contenuto di un registro a 32 bit nelle 4 celle in cima allo STACK e decrementa ESP di 4

  Esempio:

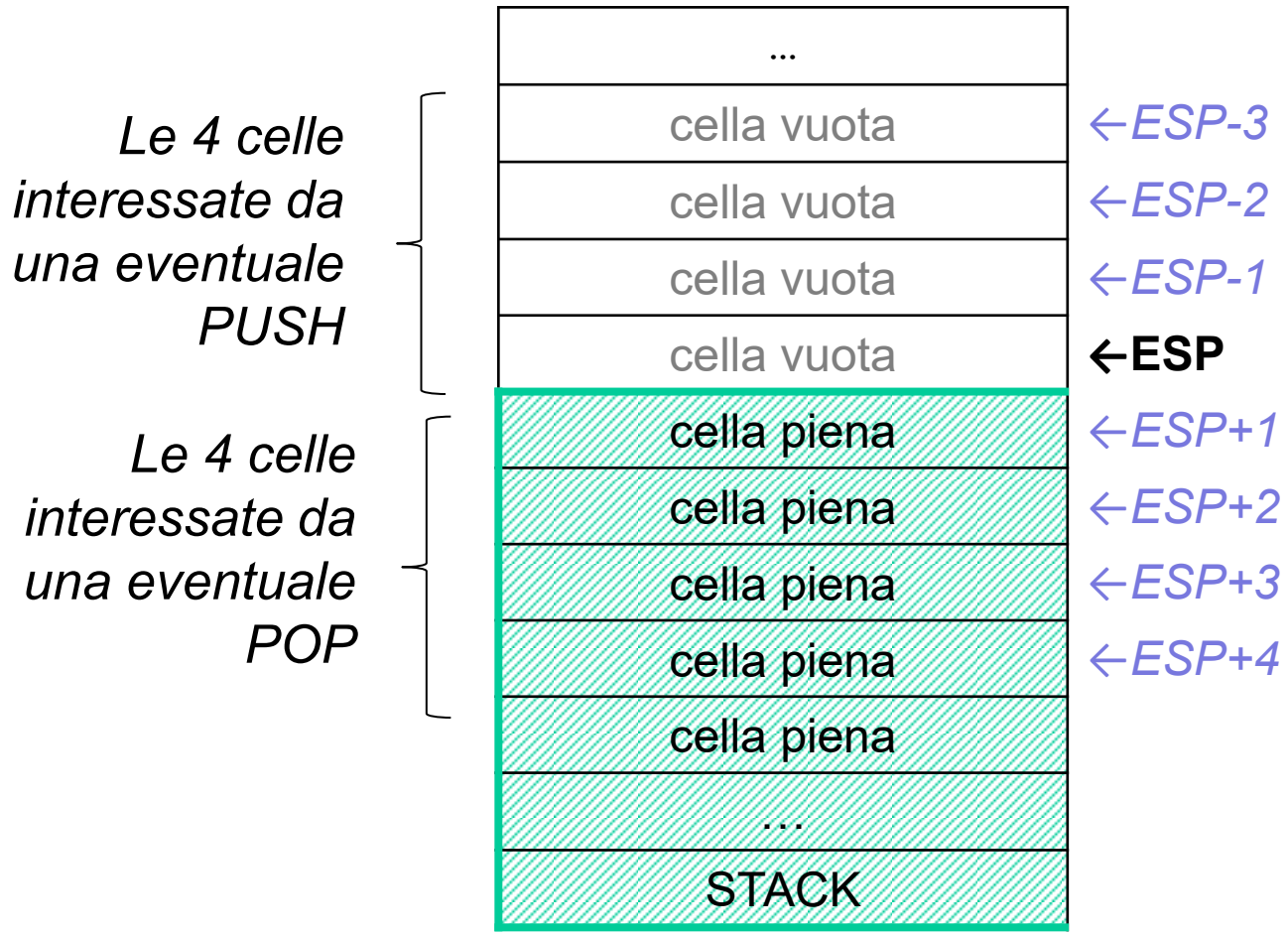
  ```
  PUSH %EAX
  ```

- L'istruzione POP recupera il contenuto delle 4 locazioni in cima allo STACK, lo salva in un registro a 32 bit e incrementa ESP di 4

  Esempio

  ```
  POP %EBX
  ```

Funzionamento delle istruzioni PUSH e POP



L'istruzione CALL

- L'istruzione CALL permette di andare ad eseguire un sottoprogramma
- Esempio di chiamata del sottoprogramma `input`:

```
04AAFFB3  CALL input
04AAFFB8  INC  %EAX
          ...
```

← *quando si arriva ad eseguire la CALL in EIP ci sarà indirizzo dell'istruzione INC (ad esempio, 04AAFFB8)*

- Concettualmente la CALL equivale alle seguenti istruzioni:

```
PUSH %EIP      # salva 04AAFFB8 sullo STACK
JMP 1E22AF0C   # se input inizia a 1E22AF0C
```

NB: la `PUSH %EIP` non si può fare esplicitamente in quanto non fa parte del set delle istruzioni del processore!

L'istruzione RET

- L'istruzione RET permette di tornare ad eseguire il sottoprogramma chiamante a partire dall'istruzione successiva alla CALL

Esempio:

```
1E22AF0C  input: IN (0AFB), %AL
                 ...
                 RET
```

- Concettualmente, effettua le seguenti operazioni:

```
POP %EIP
JMP %EIP
```

Lo Stack (7/7)

Sintetizzando, le istruzioni che utilizzano lo STACK sono:

PUSH, POP, PUSHAD, POPAD, CALL e RET

NB:

- Tutte modificano l'ESP
- Alcune modificano anche l'EIP (CALL e RET)

Sottoprogrammi (1/2)

Qualunque blocco di istruzioni assembler consecutive che inizia con una label e termina con una RET.

Esempio di un sottoprogramma che stampa a video un asterisco:

```
subprog:  PUSHAD          # salva il contenuto di tutti i registri gen.
          MOV '*', %AL
          CALL output
          POPAD           # ripristina il contenuto di tutti i reg. gen.
          RET
```

Le informazioni fra il sottoprogramma chiamante (ad esempio il `_main`) e quello chiamato avviene tramite registri.

```
subprog:  CALL output  # stampa il carattere messo in %AL dal chiamante
          CALL input   # legge un nuovo carattere e lo restituisce in %AL
          RET
```

Il sottoprogramma principale: _main

Ogni programma Assembler deve prevedere un sottoprogramma principale, denominato `_main`

Quando il file ascii (con estensione .s) viene compilato per generare il file oggetto con estensione .o e poi linkato in un .exe, la prima istruzione assembler che verrà eseguita sarà quella che si trova all'indirizzo dell'etichetta `_main`

Sottoprogrammi (3/3)

Alcuni sottoprogrammi di utilità, contenuti nel file assembler `"utility"`

Input ed output di caratteri e stringhe

`input` aspetta un carattere dalla tastiera e mette il suo codice ASCII in AL

`output` stampa a video il carattere contenuto in %AL

`outmess` stampa a video gli N caratteri che si trovano in memoria a partire dall'indirizzo contenuto in EBX. N è il naturale contenuto in CX.

Input ed output di naturali e interi su 8 bit

`inbyte` aspetta che vengano inseriti due caratteri (entrambi fra 0-9 o A-F) e poi, interpretandoli come cifre esadecimali, assegna gli 8 bit di %AL. Esempio 'E3' => in AL ci finisce 11100011

`outbyte` stampa a video i due caratteri ASCII corrispondenti alle due cifre esadecimali associate al corrente contenuto di %AL. Esempio: se il contenuto di AL è 01101010 a video viene stampata la stringa `"6A"`

`newline` Porta il cursore all'inizio della linea seguente (`CALL newline` equivale al `cout<<endl`)