Corso di Architettura degli Elaboratori
Modulo di Assembly

ASSEMBLY 8088

Bruno Iafelice
Università di Bologna
iafelice at cs(dot)unibo(dot)it

Argomenti

- Formato delle istruzioni
- Indirizzamento
- Istruzioni di trasferimento, aritmetiche, logiche, con lo stack
- Cicli e operazioni iterative
- Istruzioni di salto
- Costrutto IF-ELSE

Formato delle istruzioni

[label:] istruzione [operando/i] [; commento]

- **label** consente di dare un **nome simbolico** (da utilizzare come operando in altre istruzioni) a **variabili di memoria, valori, singole istruzioni, procedure**
- **istruzione/direttiva** è il **mnemonico per un'istruzione eseguibile o una direttiva**: individua il tipo di operazione da eseguire e il numero e il tipo degli operandi
- **operando/i** è una combinazione di nessuna, una o più **costanti**, **riferimenti a registri** o **riferimenti alla memoria**;
  se un'istruzione ammette due operandi:
  - il primo è l'operando destinazione
  - il secondo è l'operando sorgente

  ad esempio, MOV AX, CX copia il contenuto del registro CX nel registro AX
- **commento** consente di rendere più leggibile il programma

- Assemblatore ←→ Notazione Assembler
  I nomi **mnemonici** possono cambiare tra diversi assemblatori
- **nomi simbolici** per le variabili e costanti (es. registri)
- **etichette** per le costanti
- **direttive** all'assemblatore (es. macro)
- **notazioni** per introdurre commenti

Costanti

- di default, tutti i valori numerici sono espressi **in base dieci**
- è possibile esprimere le costanti numeriche:
  - **in base 16** (esadecimale) mediante il suffisso **H** (il primo digit deve essere numerico)
  - **in base 8** (octal) mediante il suffisso **O**
  - **in base 2** (binario) mediante il suffisso **B**

Ad esempio:

0FFh
0h
777O

Un programma assembly è una sequenza di:

- Istruzioni dell'Architettura di riferimento
- Macro dell'assemblatore di riferimento
- Direttive dell'assemblatore di riferimento

Assembly 8088

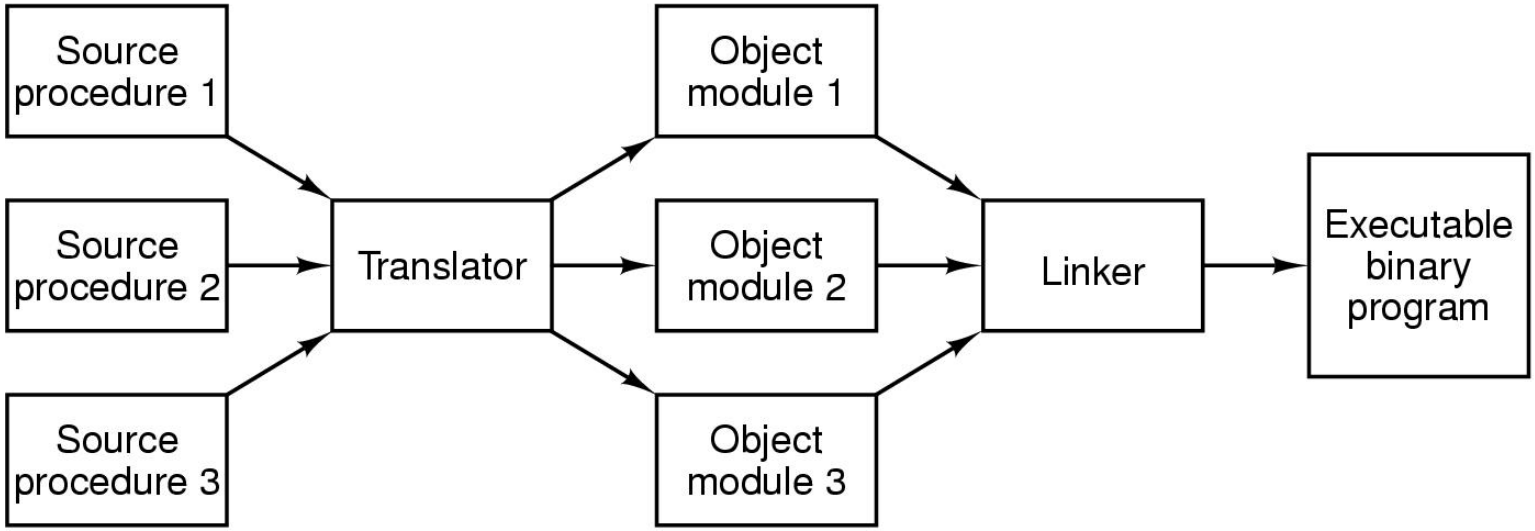
Assemblatore

Macro: non supportate dall'assemblatore del libro.

Direttiva: Controlla il comportamento dell'assemblatore in fase di traduzione.
Non necessariamente e' tradotta in codice macchina.

Assemblaggio ed Esecuzione

- Assemblaggio: prima passata e seconda passata; risoluzione dei riferimenti in avanti.
- Collegamento e caricamento (linking and loading): per unire le diverse procedure e caricare in memoria centrale
- Rilocazione in memoria centrale



INDIRIZZAMENTO

- L'8088 mette a disposizione diverse modalità di Indirizzamento
- Molte istruzioni sono a due operandi:

```
MOV DEST,SORG
```

- La destinazione deve permettere la *memorizzazione* dell'informazione: registro, stack, locazione di memoria
- Il sorgente può anche essere una costante:

```
MOV AX,0
```
azzera AX

- Ci sono anche istruzioni a 1 operando: incrementi, shift, negazioni, ...

Modalità di indirizzamento degli operandi

L'operando di un'istruzione puo' essere:

- in un **registro**
- **nell'istruzione stessa** (Operando Immediato)
- in **memoria (stack e variabili)**
- in una porta di **I/O**

OPERANDO REGISTRO

- Indirizzamento veloce poiché l'indirizzo (del registro) è già nella CPU
- Registro a 8 o 16 bit

```
MOV AX,BX
```

OPERANDO IMMEDIATO

- Operando contenuto nell'istruzione
- Accesso veloce
- Il dato può essere una costante

```
MOV AX,10
INC DI
```

OPERANDO IN MEMORIA – INDIRIZZAMENTO DIRETTO

OPERANDO IN MEMORIA – INDIRIZZAMENTO INDIRETTO MEDIANTE REGISTRO

Definizione:

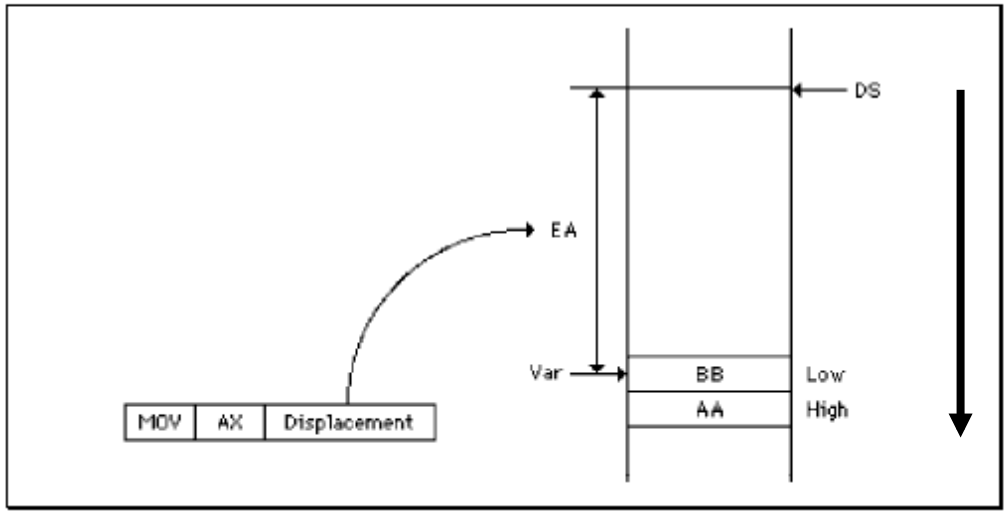
EA = **Effective Address**, indirizzo effettivo dell'operando in memoria.

Indirizzamento diretto

- E' l'indirizzamento di memoria più semplice
- L'EA è contenuto nel campo **displacement** dell'istruzione

```
Var     DW     0AABBh
        . . .
        MOV  AX, Var        ;  AX <- Var (16 bit)

        MOV  AX, [Var]      ;  AX <- Var (16 bit)
```




```
AH  <-  0AAh
AL  <-  0BBh
```

Indirizzamento indiretto tramite registro

L'EA si ottiene sommando:

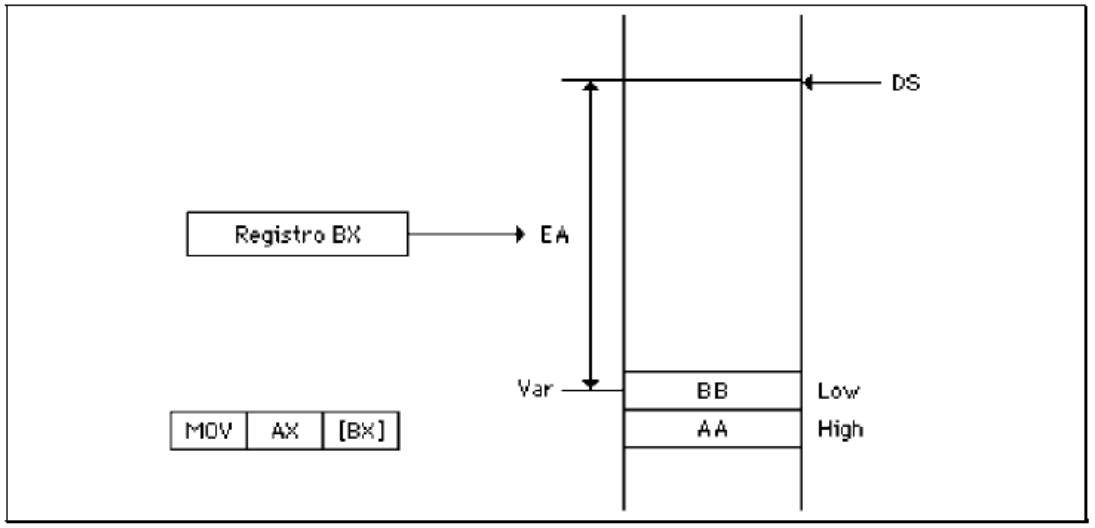
- un registro di base (BX o BP)

con uno spiazzamento (**displacement**) dato da

- un registro indice (DI o SI)
- e/o un (costante)

Un registro è opzionale (almeno uno o entrambi)!

```
MOV  BX, OFFSET Var  ;  BX <- offset di Var
MOV  AX, [BX]        ;  AX <- [BX] (16 bit)
```



- Utilizzando BP il segmento di default è SS

- Esempi:

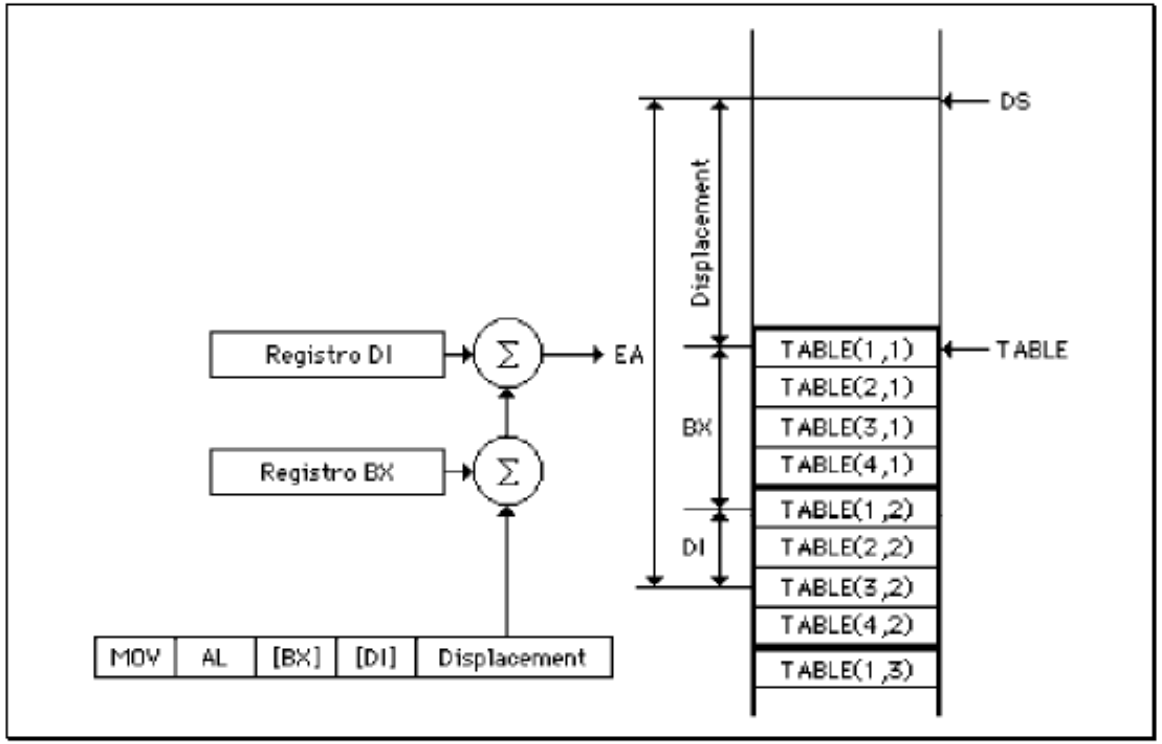
```
MOV AX,[BP+2]          AX ← SS: [BP+2]
MOV [BX+2+DI],AX       [BX+DI+2] ← AX
MOV [BX],0             8 o 16 bit?
```

MOV AX,[BX+2+DI]

BX contiene il displacement tra l'indirizzo di partenza della matrice e la colonna selezionata (la seconda)

DI contiene il displacement tra l'indirizzo di partenza della colonna selezionata e la riga selezionata (la terza)

Utile per es. per accedere ai dati di una tabella, cioè contenuti in locazioni di memoria successive.



NOTA: Assemblatore del Libro e Queste dispense

Il libro di testo del Tanenbaum usa la notazione “(BX)” con “(” “)” per deferenziare un indirizzo. Questi lucidi usano la notazione [BX] attribuendone lo stesso significato.

```
MOV [BX+2+DI],AX
```

è equivalente a scrivere

```
MOV 2(DI)(BX),AX
```

Modalità	Operando	Esempi
Indirizzamento a registro		
Registro da un byte	Registro da un byte	AH, AL, BH, BL, CH, CL, DH, DL
Registro da una parola	Registro da una parola	AX, BX, CX, DX, SP, BP, SI, DI
Indirizzamento del segmento dati		
Indirizzo diretto	L'indirizzo segue l'opcode	(#)
A registro indiretto	L'indirizzo è nel registro	(SI), (DI), (BX)
A registro con spiazzamento	L'indirizzo è dato da registro + spiazz.	#(SI), #(DI), #(BX)
A registro con indice	L'indirizzo è BX + SI/DI	(BX)(SI), (BX)(DI)
A registro con indice e spiazzamento	BX + SI/DI + spiazzamento	#(BX)(SI), #(BX)(DI)
Indirizzamento del segmento di stack		
Indiretto a puntatore base	L'indirizzo è nel registro	(BP)
A puntatore base con spiazzamento	L'indirizzo è BP + spiazz.	#(BP)
A puntatore base con indice	L'indirizzo è BP + SI/DI	(BP)(SI), (BP)(DI)
A puntatore base con indice e spiazz.	BP + SI/DI + spiazzamento	#(BP)(SI), #(BP)(DI)
Dati immediati		
byte/parola immediato/a	I dati sono parte dell'istruzione	#
Indirizzamento implicito		
Istruzione di push/pop	Indirizzo indiretto (SP)	PUSH, POP, PUSHF, POPF
Flag di load/store	registro dei flag di stato	LAHF, STC, CLC, CMC
Traduzione XLAT	AL, BX	XLAT
Istruzioni reiterative su stringhe	(SI), (DI), (CX)	MOVS, CMPS, SCAS
Istruzioni di input/out	AX, AL	IN #, OUT #
Conversioni di byte o parole	AL, AX, DX	CBW, CWD

Come il BIU risolve gli indirizzi

- ***Ogni*** riferimento alla memoria richiede l'uso di un registro di segmento.
- Se in un'istruzione viene specificato un indirizzo senza specificare un registro di segmento allora si interpreta automaticamente come riferita al segmento Dati e si usa DS.
  ES: `es mov [bx],ax` oppure `mov [es:bx],ax`
- Il registro di segmento è la BASE a cui aggiungere l'OFFSET per ottenere la Word o il Byte puntato.
- Le Variabili di programma di norma sono sul data segment corrente (DS)
- Quando in un'istruzione viene utilizzato BP come registro base, se non è indicato diversamente, il BIU suppone che l'operando sia sul segmento Stack.

Come il BIU risolve gli indirizzi

Tipo di riferimento alla memoria	Segmento di default	Altri segmenti utilizzabili	OFFSET
Fetch dell'istruzione	CS	Nessuno	IP
Operazione sullo stack	SS	Nessuno	SP
Variabile (tranne il caso seguente)	DS	CS, ES, SS	Indirizzo effettivo
BP come Registro Base	SS	CS, DS, ES	Indirizzo effettivo

segmento : offset

ISTRUZIONI di Trasferimenti, copia, aritmetiche

Abbreviazione	Descrizione	Operandi	Flag di stato			
			O	S	Z	C
MOV(B)	Trasferimento di parola o byte	r ← e, e ← r, e ← #	-	-	-	-
XCHG(B)	Scambio di parole	r ↔ e	-	-	-	-
LEA	Caricamento di un indirizzo fisico	r ← #e	-	-	-	-
PUSH	Push sullo stack	e, #	-	-	-	-
POP	Pop dallo stack	e	-	-	-	-
PUSHF	Push dei flag	-	-	-	-	-
POPF	Pop dei flag	-	-	-	-	-
XLAT	Traduzione di AL	-	-	-	-	-
ADD(B)	Somma di parole	r ← e, e ← r, e ← #	*	*	*	*
ADC(B)	Somma di parole con riporto	r ← e, e ← r, e ← #	*	*	*	*
SUB(B)	Sottrazione di parole	r ← e, e ← r, e ← #	*	*	*	*
SBB(B)	Sottrazione di parole con prestito	r ← e, e ← r, e ← #	*	*	*	*
IMUL(B)	Moltiplicazione con segno	e	*	U	U	*
MUL(B)	Moltiplicazione senza segno	e	*	U	U	*
IDIV(B)	Divisione con segno	e	U	U	U	U
DIV(B)	Divisione senza segno	e	U	U	U	U

- e indirizzo effettivo
- r registro del processore
- # operando immediato

ISTRUZIONI di Scrittura/Lettura con lo Stack

- PUSH e POP aggiungono/rimuovono un elemento dalla cima dello stack selezionato da SS:SP.
- Le operazioni sullo stack modificano il valore di SP
  - L'operazione di PUSH decrementa SP
  - L'operazione di POP incrementa SP

- PUSH: operando immediato o indirizzo effettivo
  `PUSH 30` operando immediato
  `PUSH BX` indirizzo effettivo
  è implicito il "dove mettere il dato", cioè SP è implicito

- POP: indirizzo effettivo
  `POP BX`
  è implicito il "da dove prelevare il dato", cioè SP è implicito

- Le operazioni PUSHF e POPF trasferiscono il contenuto del registro flag nella cima dello stack e viceversa.

ISTRUZIONI Aritmetiche

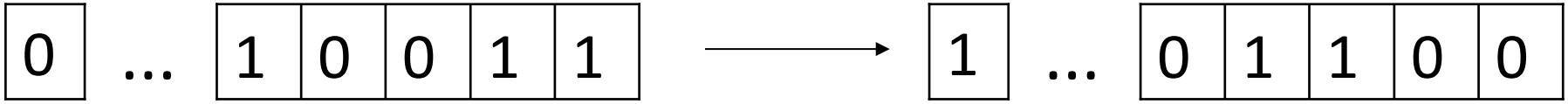
- ADD e ADC sommano l'operando sorgente all'operando destinazione e memorizzano il risultato nell'operando destinazione. ADC comprende nella somma il flag del riporto.
- SUB e SUBB sottraggono l'operando sorgente all'operando destinazione con o senza il flag del riporto.
- MUL e IMUL moltiplicano due operandi con/senza segno. Il risultato è in AL:AH se si moltiplicano byte, in AX:DX se si moltiplicano due parole.
- DIV e IDIV dividono due operandi con/senza segno. Il risultato è in AL(quoziente):AH(resto) se si dividono byte, in AX (quoziente):DX(resto) se si dividono due parole.

ISTRUZIONI Logiche

Abbreviazione	Descrizione	Operandi	Flag di stato			
			O	S	Z	C
CBW	Estensione da byte a parola	-	-	-	-	-
CWD	Estensione da parola a double	-	-	-	-	-
NEG(B)	Complemento binario	e	*	*	*	*
NOT(B)	Complemento logico	e	-	-	-	-
INC(B)	Incremento della destinazione	e	*	*	*	-
DEC(B)	Decremento della destinazione	e	*	*	*	-
AND(B)	AND	e ← r, r ← e, e ← #	0	*	*	0
OR(B)	OR	e ← r, r ← e, e ← #	0	*	*	0
XOR(B)	OR esclusivo	e ← r, r ← e, e ← #	0	*	*	0
SHR(B)	Scorrimento logico verso destra	e ← 1, e ← CL	*	*	*	*
SAR(B)	Scorrimento aritmetico verso destra	e ← 1, e ← CL	*	*	*	*
SAL(B) (=SHL(B))	Scorrimento logico verso sinistra	e ← 1, e ← CL	*	*	*	*
ROL(B)	Rotazione a sinistra	e ← 1, e ← CL	*	-	-	*
ROR(B)	Rotazione a destra	e ← 1, e ← CL	*	-	-	*
RCL(B)	Rotazione a sinistra con riporto	e ← 1, e ← CL	*	-	-	*
RCR(B)	Rotazione a destra con riporto	e ← 1, e ← CL	*	-	-	*

NOT e NEG

- NOT, complemento LOGICO (nega bit a bit)



- NEG, complemento BINARIO (nega il *numero*)

5 ⟶ -5

NEG
calcolo del complemento a due

- Per calcolare il Negato si usa la notazione di Complemento a due per rappresentare i numeri negativi

- *Facciamo un esempio rappresentando il numero -5 con 8 bit in complemento a 2:*

  `0000 0101` (5)

  `1111 1010` (5 in Complemento a 1)

  `1111 1011` (-5, in Complemento a due)

- *Algoritmo:*

  1. Negazione bit a bit
  2. somma di “1”

OPERAZIONI ITERATIVE

- Cicli (comando `LOOP`): esegue N volte un blocco di codice
- Salti (comando `JMP`): salta ad una particolare linea di codice.
- I cicli e i salti sono condizionati dai valori dei FLAG. Si compie prima un'operazione di comparazione (`CMP AX, BX`) che modifica il registro dei flag e in seguito si invoca l'istruzione di ciclo o salto.

Cicli

- `LOOP label`
- `LOOP`
  - (usa `CX` in modo implicito)
  - decrementa `CX` di 1
  - in base al risultato:
    - se CX è positivo (>0) salta all'etichetta specificata come operando
    - se CX non è positivo continua con l'istruzione seguente
- Il "punto di salto" deve trovarsi entro 128byte dalla posizione corrente

```
        MOV  CX, 100     ;  CX <- Numero di iterazioni
Start:
        < corpo del ciclo >
        LOOP  Start      ;  se CX diverso da zero
                         ;  salta a Start
        . . .            ;  altrimenti continua
```

esegue 100 volte il blocco indicato dall'etichetta `Start`.

- `LOOPZ` Loop if Zero
- `LOOPE` Loop if Equal
- `LOOPNZ` Loop if NotZero
- `LOOPNE` Loop if NotEqual
- come `LOOP`
- ma in più “saltano” se è verificata la condizione sul flag `Z`

Salti

- `JMP label`
- Salta all'**etichetta** indicata come operando
- `LOOP` → salto condizionato
- `JMP` → salto incondizionato

```
        JMP  Label1
        . . .
Label1:
        . . .
```

- ...oppure salta all'**indirizzo effettivo** passato come operando

- 2 tipi di salto:
  - Corto
    - la destinazione si trova nel segmento codice corrente
    - si modifica solo IP
  - Lungo
    - la dest. si trova in altro seg. codice
    - il salto modifica sia CS sia IP
    - Destinazione: `CS:label`

Salto condizionato

- 15 istruzioni di salto condizionato in base ai Flag
- Distanza massima di salto 128Byte (per as88)

Istruzione	Descrizione	Condizione
JNA, JBE	Al di sotto o uguale	CF=1 o ZF=1
JNB, JAE, JNC	Non al di sotto	CF=0
JE, JZ	Zero, uguale	ZF=1
JNLE, JG	Maggiore di	SF=OF o ZF=0
JGE, JNL	Maggiore o uguale	SF=OF
JO	Overflow	OF=1
JS	Segno negativo	SF=1
JCXZ	CX vale zero	CX=0
JB, JNAE, JC	Al di sotto	CF=1
JNBE, JA	Al di sopra	CF=0 & ZF=0
JNE, JNZ	Non zero, diverso	ZF=0
JL, JNGE	Minore di	SF<>OF
JLE, JNG	Minore o uguale	SF<>OF o ZF=1
JNO	Non overflow	OF=0
JNS	Non negativo	SF=0

Istruzione	Descrizione	Salta se ...
JA	Jump if Above	CF = 0 e ZF = 0
JAE	Jump if Above or Equal	CF = 0
JB	Jump if Below	CF = 1
JBE	Jump if Below or Equal	CF = 1 o ZF = 1
JC	Jump if Carry	CF = 1
JE	Jump if Equal	ZF = 1
JG	Jump if Greater	ZF = 0 e SF = OF
JGE	Jump if Greater or Equal	SF = OF
JL	Jump if Less	SF ≠ OF
JLE	Jump if Less or Equal	ZF = 1 o SF ≠ OF
JNA	Jump if Not Above	CF = 1 o ZF = 1
JNAE	Jump if Not Above nor Equal	CF = 1
JNB	Jump if Not Below	CF = 0
JNBE	Jump if Not Below nor Equal	CF = 0 e ZF = 0
JNC	Jump if No Carry	CF = 0
JNE	Jump if Not Equal	ZF = 0
JNG	Jump if Not Greater	ZF = 1 o SF ≠ OF
JNGE	Jump if Not Greater nor Equal	SF ≠ OF
JNL	Jump if Not Less	SF = OF
JNLE	Jump if Not Less nor Equal	ZF = 0 e SF = OF
JNO	Jump if No Overflow	OF= 0
JNP	Jump if No Parity (odd)	PF = 0
JNS	Jump if No Sign	SF = 0
JNZ	Jump if Not Zero	ZF = 0
JO	Jump on Overflow	OF =1
JP	Jump on Parity (even)	PF = 1
JPE	Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JS	Jump on Sign	SF = 1
JZ	Jump if Zero	ZF = 1

COSTRUTTO IF

Per saltare se	Valori senza segno	Valori con segno
Destinazione = Sorgente	JE	JE
Destinazione ≠ Sorgente	JNE	JNE
Destinazione > Sorgente	JA	JG
Destinazione ≥ Sorgente	JAE	JGE
Destinazione < Sorgente	JB	JL
Destinazione ≤ Sorgente	JBE	JLE

Esempio: realizzazione di un test a tre vie

```
if (A==B) then istruzione_1;  A=B
       else
       if (A<B) then istruzione_2; A<B
       else istruzione_3; A>B


    CMP  AX, BX
    JAE  LabelGreaterOrEqual ; salta se AX>=BX
    . . .            ; istruzioni eseguite se AX < BX
    . . .
LabelGreaterOrEqual:
    JA LabelGreater      ; salta se AX>BX
    . . .            ; istruzioni eseguite se AX = BX
    . . .
LabelGreater:
    . . .            ; istruzioni eseguite se AX > BX
    . . .
```


Cortesia Antonio Corradi

COSTRUTTO IF e CICLO WHILE

```
while (a>1000)
        ...
        ...
end while;
```

```
whileSum:
    CMP AX,1000
    JNL end_while
     ...
   JMP whileSum
end_while:
```


Cortesia Marco Di Felice