

# 1 Lezione del 26-09-24

## 1.1 Istruzioni di trasferimento

Le istruzioni di trasferimento spostano memoria:

- Dalla memoria a un registro;
- Da un registro a un registro;
- Dallo spazio I/O a un registro.

Non esistono altre possibilità, ergo non si può (per quanto interessa a noi) spostare da memoria a memoria. In verità esistono alcune istruzioni nei processori di nuova generazione che ottimizzano operazioni di questo tipo, che verranno viste in seguito. Sfruttando i registri, il trasferimento da memoria a memoria si fa attraverso un registro, in due istruzioni.

Nessuna istruzione di trasferimento modifica i flag.

### 1.1.1 MOVE

- **Formato:** **MOV** source, destination
- **Azione:** sostituisce l'operando destinatario con una copia dell'operando sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale	<b>MOV</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>MOV</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>MOV</b> %AX, %DX
Immediato, Memoria	<b>MOVB</b> \$0x5B, (%EDI)
Immediato, Registro generale	<b>MOV</b> \$0x54A3, %AX

### 1.1.2 LOAD EFFECTIVE ADDRESS

- **Formato:** **LEA** source, destination
- **Azione:** sostituisce l'operando destinatario con l'espressione indirizzo contenuta nell'operando sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale a 32 bit	<b>LEA</b> 0x00002000, %EDX
	<b>LEA</b> 0x00213AB1 (%EAX,%EBX,4), %ECX

A differenza di MOV, LEA calcola l'indirizzo della locazione di memoria cercata come  $\text{base} + \text{index} \times \text{scala} \pm \text{displacement}$ , e carica quell'indirizzo nella destinazione, non il valore contenuto in esso. Nel primo esempio, questo equivale alla MOV con indirizzamento immediato. In altri casi permette di ricavare esplicitamente il valore ottenuto dall'indirizzamento complesso.

### 1.1.3 EXCHANGE

- **Formato:** **XCHG** source, destination
- **Azione:** sostituisce l'operando destinatario con l'operando sorgente e viceversa. Questa operazione è l'unica che modifica il sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale	<b>XCHG</b> 0x00002000, \%DX
Registro Generale, Memoria	<b>XCHG</b> \%AL, 0x000A2003
Registro Generale, Registro Generale	<b>XCHG</b> \%EAX, \%EDX

Grazie a quest'istruzione in assembler si possono scambiare due operandi con una sola istruzione (**non trasparenza** dei registri) **atomica**. Questo è particolarmente utile nel caso di esecuzione concorrente.

### 1.1.4 INPUT

- **Formato:**
  - **IN** indirizzo, \%AL (8 bit)
  - **IN** indirizzo, \%AX (16 bit)
  - **IN** (\%DX), \%AX (8 bit)
  - **IN** (\%DX), \%AX (16 bit)
- **Azione:** sostituisce il contenuto del registro destinatario (AL 8 bit, AX 16 bit) con il contenuto di un adeguato numero di porte consecutive. L'indirizzo è specificato direttamente (per porte con indirizzo < 256), o indirettamente usando il registro DX.
- **Flag:** nessuno.

### 1.1.5 OUTPUT

- **Formato:**
  - **OUT** \%AL, indirizzo (8 bit)
  - **IN** \%AX, indirizzo (16 bit)
  - **IN** \%AX, (%DX) (8 bit)
  - **IN** \%AX, (%DX) (16 bit)
- **Azione:** copia il contenuto del registro sorgente (AL 8 bit, AX 16 bit) su un adeguato numero di porte consecutive. L'indirizzo è specificato direttamente (per porte con indirizzo < 256), o indirettamente usando il registro DX.
- **Flag:** nessuno.

### 1.1.6 Non ortogonalità INPUT/OUTPUT

Le uniche due operazioni che gestiscono l'input e l'output possono trasferire solo dai o nei registri AL e AX, e indirizzare indirettamente la memoria puntando col registro DX. Questo rende le operazioni non ortogonali: non si possono usare altri registri, ed eventuali operazioni vanno fatte nel processore,

## 1.2 Pila

La pila, o **stack**, è una regione di memoria gestita con politica Last In First Out (LI-FO), essenziale al funzionamento del calcolatore. Permette di annidare sottoprogrammi, funzionalità per cui l'assembler è organizzato.

Generalmente, la pila viene usata come segue per eseguire i sottoprogrammi:

- Prima di saltare al sottoprogramma, si fa **PUSH** sulla pila dell'indirizzo di ritorno (e.g. l'indirizzo della prossima istruzione);
- Si esegue il sottoprogramma;
- Al termine del sottoprogramma, si fa **POP** dalla pila del prossimo indirizzo.

Più sottoprogrammi possono chiamarsi a vicenda (annidarsi), ponendosi su livelli via via superiori della pila. Al termine della sua esecuzione, ogni sottoprogramma tornerà all'indirizzo di ripresa del sottoprogramma precedente, finché tutti i sottoprogrammi non termineranno l'esecuzione.

Il registro **ESP** punta al top della pila, ergo non va usato per altri scopi. Va però inizializzato prima che parta il programma. Si deve inoltre notare che la pila in assembler si estende *verso il basso*: aggiungere alla pila significa decrementare ESP, e rimuovere dalla pila significa incrementare ESP. I frame successivi della pila si vanno a disporre via via sotto (o "a sinistra") del frame corrente.

Per lavorare sulla pila si usano le istruzioni:

### 1.2.1 PUSH

- **Formato:** **PUSH** source
- **Azione:** decrementa ESP e copia il sorgente nell'indirizzo puntato da ESP. Il sorgente deve essere a 16 bit o a 32 bit. Nello specifico, compie le seguenti azioni:
  - Decrementa l'indirizzo contenuto nel registro ESP di 2 o 4;
  - Memorizza una copia dell'operando sorgente nella word o long il cui indirizzo è contenuto in ESP.
- **Flag:** nessuno.

Operandi	Esempi
Memoria	PUSHW 0x3214200A
Immediato	PUSHL \x0x4871A000
Registro Generale	<b>PUSH</b> %BX

### 1.2.2 POP

- **Formato:** **POP** destination
- **Azione:** copia una word o un long dall'indirizzo puntato dall'ESP nel destinatario e incrementa ESP. Nello specifico compie le seguenti azioni:
  - Sostituisce all'operando destinatario una copia del contenuto nella word o long il cui indirizzo è contenuto in ESP;
  - Incrementa di due o quattro l'indirizzo contenuto in ESP, rimuovendo la word o il long copiato.
- **Flag:** nessuno.

Operandi	Esempi
Memoria	POPW 0x02AB2000
Registro Generale	<b>POP</b> \%BX

#### Dati temporanei nella pila

Solitamente la pila viene usata per memorizzare dati temporanei, visto che i registri sono pochi e spesso hanno scopi diversi in momenti diversi. Ad esempio:

```

1 # sto usando %EAX, mi serve un dato da una porta
2 PUSH %EAX
3 IN 0x001A, %AL
4 ...
5 POP %EAX # ritorno da dove ero

```

### 1.2.3 PUSHAD

- **Formato:** **PUSHAD**
- **Azione::** salva nella pila corrente una copia degli 8 registri generali a 32 bit, nell'ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **Flag:** nessuno.

### 1.2.4 POPAD

- **Formato:** **POPAD**
- **Azione::** copia dalla pila corrente gli 8 registri generali a 32 bit, nell'ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **Flag:** nessuno.

## 1.3 Istruzioni aritmetiche

Molte operazioni aritmetiche di base non distinguono numeri naturali e numeri interi, distinzione che viene fatta solo per moltiplicazioni e divisioni.

Le operazioni possono modificare i flag, e in questo caso i flag da controllare dipenderanno dal tipo di numeri su cui si è fatta l'operazione (informazione nota soltanto al programmatore).

Abbiamo quindi che un'operazione aritmetica si svolge seguendo i passi:

- Si esegue l'operazione;
- Si controllano i flag interessati (OF, SF e ZF sugli interi, CF e ZF sui naturali) per verificarne l'esito.

Vediamo quindi le operazioni aritmetiche:

### 1.3.1 ADD

- **Formato:** **ADD** source, destination
- **Azione:** modifica l'operando destinatario sommandovi l'operando sorgente. Il risultato è consistente sia che si interpretino i numeri come naturali, che come interi.
- **Flag:** attiva CF se, interpretando i numeri come naturali, si è verificato un riporto; attiva OF se, interpretando gli operandi come interi, si è verificato un traboccamento. Inoltre attiva opportunamente ZF e SF se il numero è rispettivamente zero o negativo (in complemento a 2).

Operandi	Esempi
Memoria, Registro Generale	<b>ADD</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>ADD</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>ADD</b> %AX, %DX
Immediato, Memoria	<b>ADD</b> \$0x5B, (%EDI)
Immediato, Registro Generale	<b>ADD</b> \$0x54A3, %AX

#### Funzionamento della ADD

Il passo elementare di una somma consiste nel sommare due addendi (propriamente due cifre degli addendi) e un riporto entrante per produrre:

- Una cifra;
- Un riporto uscente (cioè il riporto entrante per il prossimo passo).

L'ultimo riporto, se non entra in memoria, attiva il carry flag (CF).

L'operazione di somma ha lo stesso effetto sia su naturali che su interi in complemento a 2: la differenza sta nel controllo dell'attivazione dei flag. Il carry flag non ha infatti alcun significato nella somma fra interi: dobbiamo controllare l'OF.

In generale, si ha overflow (OF) quando il risultato esce dall'intervallo di rappresentabilità. Si può capire se si è verificato un overflow controllando i segni degli operandi:

- **Segni discordi:** non c'è overflow;
- **Segni concordi:** il risultato è concorde se è concorde con gli operandi.

La ADD imposta quindi OF secondo queste regole. Il ZF viene poi impostato se il risultato è fatto da tutti zeri, e il SF viene impostato se il MSB è uno.

### 1.3.2 INCREMENT

- **Formato:** **INC** destination
- **Azione:** equivale all'istruzione **ADD** `\$1, destination`.
- **Flag:** modifica tutti i flag di ADD tranne CF (il riporto).

Operandi	Esempi
Memoria	<b>INCB</b> ( <code>\%ESI</code> )
Registro Generale	<b>INC</b> <code>\%CX</code>

Quest'istruzione è più compatta di ADD, e storicamente era anche più veloce. Questo deriva dal fatto che la circuiteria che implementava l'incremento era più efficiente di quella che implementa le somme.

### 1.3.3 SUBTRACT

- **Formato:** **SUB** source, destination
- **Azione:** modifica l'operando destinatario sottraendovi l'operando sorgente. Il risultato è consistente sia che si interpretino i numeri come naturali, che come interi.
- **Flag:** attiva CF se, interpretando i numeri come naturali, si è verificato un riporto; attiva OF se, interpretando gli operandi come interi, si è verificato un traboccamento.

Operandi	Esempi
Memoria, Registro Generale	<b>SUB</b> <code>0x00002000, \%EDX</code>
Registro Generale, Memoria	<b>SUB</b> <code>\%CL, 0x12AB1024</code>
Registro Generale, Registro Generale	<b>SUB</b> <code>\%AX, \%DX</code>
Immediato, Memoria	<b>SUBB</b> <code>\\$0x5B, (\%EDI)</code>
Immediato, Registro Generale	<b>SUB</b> <code>\\$0x54A3, \%AX</code>

#### Funzionamento della SUBTRACT

Il passo elementare della sottrazione è effettivamente il contrario di quello della somma: si sottraggono il sottraendo e un prestito entrante al minuendo, producendo:

- Una cifra;
- Un prestito uscente.

Il carry flag (CF) memorizza il prestito. Se alla fine dell'operazione il CF è impostato, significa che il risultato è un numero intero.

Questo funziona anche sugli interi: in questo caso, come prima, non si controlla il CF, ma l'OF, che conterrà la seguente informazione:

- La differenza di numeri concordi è sempre rappresentabile;
- La differenza di numeri discordi è rappresentabile solo se il risultato ha il segno del minuendo.

Il ZF e il SF vengono attivati secondo le regole già note.

### 1.3.4 DECREMENT

- **Formato:** **DEC** destination
- **Azione:** equivale all'istruzione **SUB** %\$1, destination.
- **Flag:** modifica tutti i flag di SUBTRACT tranne CF (il prestito).

Operandi	Esempi
Memoria	<b>DECB</b> (%EDI)
Registro Generale	<b>DEC</b> %CX

### 1.3.5 ADD WITH CARRY

- **Formato:** **ADC** source, destination
- **Azione:** modifica l'operando destinatario sommandovi sia l'operando sorgente sia il contenuto del flag CF.
- **Flag:** modifica tutti i flag come ADD.

Operandi	Esempi
Memoria, Registro Generale	<b>ADC</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>ADC</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>ADC</b> %AX, %DX
Immediato, Memoria	<b>ADCB</b> %0x5B, (%EDI)
Immediato, Registro Generale	<b>ADC</b> %0x54A3, %AX

Quest'istruzione è utile per effettuare somme di numeri più grandi di 32 bit. In questo caso si:

- Effettua la somma dei 32 bit meno significativi con ADD;
- Sommano i successivi 32 bit con ADC portandosi quindi dietro il carry.

### 1.3.6 SUBTRACT WITH BORROW

- **Formato:** **SBB** source, destination
- **Azione:** modifica l'operando destinatario sottraendovi sia l'operando sorgente sia il contenuto del flag CF.
- **Flag:** modifica tutti i flag come SUBTRACT.

Operandi	Esempi
Memoria, Registro Generale	<b>SBB</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>SBB</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>SBB</b> %AX, %DX
Immediato, Memoria	<b>SBBB</b> %0x255B, (%EDI)
Immediato, Registro Generale	<b>SBB</b> %0x54A3, %AX

Come ormai dovrebbe essere chiaro, è la duale dell'ADC, e si usa per effettuare sottrazioni di numeri più grandi di 32 bit.

### 1.3.7 NEGATE

- **Formato:** **NEG** destination
- **Azione:** interpreta l'operando destinatario come un numero intero e lo sostituisce con il suo opposto in complemento a 2.
- **Flag:** quando l'operazione non è possibile (l'intervallo di rappresentabilità degli interi in complemento a 2 non è simmetrico) imposta il flag OF. Imposta inoltre il flag CF quando l'operando è diverso da zero, e tutti gli altri flag in base a nullità e segno del risultato.

Operandi	Esempi
Memoria	<b>NEGB</b> (%EDI)
Registro Generale	<b>NEG</b> %CX

#### Funzionamento della NEGATE

L'opposto di un numero  $X$  in complemento a due è:

$$-X = \bar{X} + 1$$

Si ricordi che questo ha senso *solamente* se il numero è rappresentato in complemento a due.

### 1.3.8 COMPARE

- **Formato:** **CMP** source, destination
- **Azione:** verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando gli operandi come naturali che come interi, e aggiorna i flag di conseguenza. Più propriamente, la compare si comporta come la SUB, ma senza sovrascrivere nessuno degli operandi.
- **Flag:** come la SUB.

Operandi	Esempi
Memoria, Registro Generale	<b>CMP</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>CMP</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>CMP</b> %AX, %DX
Immediato, Memoria	<b>CMPB</b> \$0x255B, (%EDI)
Immediato, Registro Generale	<b>CMP</b> \$0x54A3, %AX

### 1.3.9 Funzionamento della COMPARE

Solitamente la CMP si usa nei salti condizionati come:

```
1 CMP %AX, %BX
2 JCOND # salto condizionato
```

Ciò che fa la CMP è effettivamente creare un'oggetto temporaneo:

$$\text{tmp} = \text{dest} - \text{source}$$

che viene poi rimosso.

I flag restano però aggiornati, e questo valore può essere interpretato correttamente dalla JE per effettuare un salto condizionale.



## 1.4 Moltiplicazioni

Le moltiplicazioni, a differenza delle somme e delle differenze, sono diverse fra naturali ed interi. Bisogna inoltre notare che le dimensioni il risultato della somma di un numero a  $n$  cifre sta su  $n$  o  $n + 1$  cifre, mentre il prodotto di due numeri a  $n$  cifre sta su  $2n$  cifre. In altre parole, il numero di bit necessari a memorizzare il risultato non è più confrontabile con quello degli operatori.

### 1.4.1 MULTIPLY

- **Formato:** `MUL source`
- **Azione:** considera l'operando sorgente come un moltiplicando, l'operando destinatario (implicito) come un moltiplicatore, e effettua la moltiplicazione assumendo i numeri naturali. Nello specifico:
  - Sorgente a 8 bit, si ha  $AX = AL \times source$ ;
  - Sorgente a 16 bit, si ha  $DX\_AX = AX \times source$ ;
  - Sorgente a 32 bit, si ha  $EDX\_EAX = EAX \times source$ .
- **Flag:** imposta CF e OF se il risultato non sta nel numero di bit di source. SF e ZF sono indefiniti.

Operandi	Esempi
Memoria	<code>MULB (%ESI)</code>
Registro Generale	<code>MUL %ECX</code>

### 1.4.2 INTEGER MULTIPLY

- **Formato:** `MUL source`
- **Azione:** considera l'operando sorgente come un moltiplicando, l'operando destinatario (implicito) come un moltiplicatore, e effettua la moltiplicazione assumendo i numeri interi. Nello specifico:
  - Sorgente a 8 bit, si ha  $AX = AL \times source$ ;
  - Sorgente a 16 bit, si ha  $DX\_AX = AX \times source$ ;
  - Sorgente a 32 bit, si ha  $EDX\_EAX = EAX \times source$ .
- **Flag:** li imposta tutti, ma non è attendibile.

Operandi	Esempi
Memoria	<code>IMULB (%ESI)</code>
Registro Generale	<code>IMUL %ECX</code>

### Funzionamento delle MULTIPLY e INTEGER MULTIPLY

Queste operazioni hanno sia un operando che il destinatario impliciti, in base al tipo dell'operando fornito. Questo deriva dal fatto che il risultato di una moltiplicazione raramente sta nello stesso numero di bit dei fattori. Di preciso, abbiamo visto i 3 tipi di moltiplicazione concessi:

- Sorgente a 8 bit, si ha  $AX = AL \times \text{source}$ ;
- Sorgente a 16 bit, si ha  $DX\_AX = AX \times \text{source}$ ;
- Sorgente a 32 bit, si ha  $EDX\_EAX = EAX \times \text{source}$ .

La differenza fra le prime due operazioni e l'ultima, in particolare con sorgente a 16 bit, che usa una due registri da 16 bit separati, ha principalmente motivi storici (il registro EAX è stato introdotto dopo).

Si può rimettere il valore dai due registri a 16 bit in un registro a 32 bit attraverso la pila:

```
1 PUSH  \%DX
2 PUSH  \%AX
3 POP   \%EAX
```