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 regsitro.

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 verrano 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	`MOV 0x00002000, \%EDX`
Registro Generale, Memoria	`MOV \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`MOV \%AX, \%DX`
Immediato, Memoria	`MOVB \$0x5B, (\%EDI)`
Immediato, Registro generale	`MOV \$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	`LEA 0x00002000, \%EDX`
	`LEA 0x00213AB1 (\%EAX,\%EBX,4), \%ECX`

A differenza di MOV, LEA calcola l'indirizzo della locazione di memoria cercata come base + index × scala ± 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	`XCHG 0x00002000, \%DX`
Registro Generale, Memoria	`XCHG \%AL, 0x000A2003`
Registro Generale, Registro Generale	`XCHG \%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), \%Al` (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 \%Al, (\%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 (LIFO), 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 \$0x4871A000`
Registro Generale	`PUSH \%BX`

1.2.2 POP

- **Formato:** `POP destination`
- **Azione:** copia una word o un long dall'indirzzo 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	`POP \%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:

```
# sto usando %EAX, mi serve un dato da una porta
PUSH %EAX
IN 0x001A, %AL
...
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	`ADD 0x00002000, \%EDX`
Registro Generale, Memoria	`ADD \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`ADD \%AX, \%DX`
Immediato, Memoria	`ADDB \$0x5B, (\%EDI)`
Immediato, Registro Generale	`ADD \$0x54A3, \%AX`

Funzionamento della ADD

Il passo elementare di una somma consiste nel sommare le cifre degli addendi, x_i e y_i e un riporto entrante r_i per produrre:

- La possima cifra s_i del risultato;
- Un riporto uscente r_{i+1} (cioè il riporto entrante per il prossimo passo).

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

Possiamo facilmente ricavare il risultato che dà ogni tripla di argomenti su un singolo bit del risultato, e il riporto entrante generato:

$\mathbf{x_i}$	$\mathbf{y_i}$	$\mathbf{r_i}$	$(\mathbf{x_i}+\mathbf{y_i}+\mathbf{r_i})_{10}$	$\mathbf{s_i}$	$\mathbf{r_{i+1}}$
0	0	0	0	0	0
1	0	0	1	1	0
0	1	0	1	1	0
0	0	1	1	1	0
1	1	0	2	0	1
1	0	1	2	0	1
0	1	1	2	0	1
1	1	1	3	1	1

Questa tabella viene evaluta su ogni tripla di cifre x_i, y_i e riporto r_i incontrata, generando r_n riporti consecutivi.

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. Si ha infatti che, se X

e Y sono le rappresentazioni su n bit di due interi x e y, allora la rappresentazione di $s = x + y$ (se esprimibile su n bit) è data da $S = X + Y$.

Tolto il CF, il processore attiva i flag OF, ZF e SF secondo le modalità:

- **OF:** rappresenta l'overflow, ergo la non rappresentabilità, nel caso di somme intere, e guarda ai segni (il MSB):
  - **Segni discordi:** non c'é overflow;
  - **Segni concordi:** il risultato è corretto se è concorde con gli operandi.

  Alternativamente, si può pensare che l'OF viene attivato sulla base degli ultimi dure riporti. Se sono discordi (cioè $r_n \neq r_{n-1}$), si attiva;
- **ZF:** si attiva se la rappresentazione S finale è uguale a 0 (si ricorda che lo 0 è tale sia in base che in complemento a 2);
- **SF:** si attiva se il MSB è uguale a 1, che in complemento a due significa segno negativo.

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	`INCB (\%ESI)`
Registro Generale	`INC \%CX`

Quest'istruzione è più compatta di ADD, e (forse solo storicamente) è anche più veloce. Questo deriva dal fatto che la circuiteria che implementa l'incremento è (in teoria) 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 trabocamento.

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

Funzionamento della SUBTRACT

Il passo elementare della sottrazione potrebbe sembrare effettivamente il contrario di quello della somma: si sottraggono le cifre del sottraendo e del minuendo, x_i e y_i, e un riporto entrante r_i al minuendo, per produrre:

- La possima cifra d_i del risultato;
- Un riporto uscente r_{i+1} (cioè il riporto entrante per il prossimo passo).

In verità risulta più conveniente usare la stessa circuiteria della somma, e adottare quindi semplicemente il complemento a 2. Abbiamo che:

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

ergo possiamo sfruttare il carry bit (che prendiamo come già impostato), ed eseguire la somma fra X e il complemento di Y. Questo ci dà il risultato corretto sia che si parli di naturali che di interi, visto che equivale a $X + (-Y)$.

In generale, l'algoritmo esatto usato per la sottrazione (sottrazione manuale con prestiti, o addizione del complemento) non è poi così importante. Dobbiamo però fare attenzione ai flag, che vengono impostati in modo diverso rispetto alla somma, ovvero:

- **CF:** non viene impostato sul riporto effettivamente generato dalla somma (in quanto 1) sarebbe irrilevante; 2) non siamo nemmeno sicuri che il complemento a 2 sia il modo in cui viene effettivamente svolta la sottrazione a livello ALU), ma sul **prestito** che si sarebbe dovuto fare nel caso si avesse avuto $Y > X$. In altre parole, nel caso di numeri naturali, il CF rappresenta se la somma è un naturale (quando è 0) o se è un intero negativo (quando è 1).
- **OF:** rappresenta l'overflow, ergo la non rappresentabilità, nel caso di sottrazioni intere, e guarda come per le somme, ai segni (il MSB):
  - **Segni concordi:** non c'è overflow;
  - **Segni discordi:** il risultato è corretto se è concorde col minuendo.
- **ZF:** si attiva se la rappresentazione S finale è uguale a 0 (si ricorda che lo 0 è tale sia in base che in complemento a 2);
- **SF:** si attiva se il MSB è uguale a 1, che in complemento a due significa segno negativo.

Si ricorda un'ultima volta: la circuiteria per la somma (e per la sottrazione) non è diversa fra naturali ed interi: è controllando i flag giusti che si riesce ad ottenere informazioni riguardo all'esito della somma, e i flag giusti sono noti solo se lo è il tipo di rappresentazione degli operandi (nozione che conosce solo il programmatore).

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	`DECB (\%EDI)`
Registro Generale	`DEC \%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	`ADC 0x00002000, \%EDX`
Registro Generale, Memoria	`ADC \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`ADC \%AX, \%DX`
Immediato, Memoria	`ADCB \$0x5B, (\%EDI)`
Immediato, Registro Generale	`ADC \$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	`SBB 0x00002000, \%EDX`
Registro Generale, Memoria	`SBB \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`SBB \%AX, \%DX`
Immediato, Memoria	`SBBB \$0x255B, (\%EDI)`
Immediato, Registro Generale	`SBB \$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	`NEGB (\%EDI)`
Registro Generale	`NEG \%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	`CMP 0x00002000, \%EDX`
Registro Generale, Memoria	`CMP \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`CMP \%AX, \%DX`
Immediato, Memoria	`CMPB \$0x255B, (\%EDI)`
Immediato, Registro Generale	`CMP \$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	`MULB (\%ESI)`
Registro Generale	`MUL \%ECX`

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	`IMULB (\%ESI)`
Registro Generale	`IMUL \%ECX`

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 $\times$ source;
- Sorgente a 16 bit, si ha DX_AX $=$ AX $\times$ source;
- Sorgente a 32 bit, si ha EDX_EAX $=$ EAX $\times$ 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:

```
PUSH \%DX
PUSH \%AX
POP \%EAX
```