1 Lezione del 02-10-24

1.1 Effetti collaterali

I sottoprogrammi non dovrebbero avere effetti collaterali, ergo dovrebbero lasciare i registri come li trovano. Per fare ciò, si sfrutta la pila per immagazzinare i loro valori precedenti:

```
sottoprog:  PUSH ... # fai push dei registri
            PUSH ...
            ... # esegui il sottoprogramma
            MOV ..., %CX

            POP ... # riprendi i resisti
            POP ...
            RET
```

Sono fondamentali due linee guida:

- Bisogna stare attenti ad operazioni come `IDIV` e `IMUL`, che sporcano registri come EDX implictamente;
- Bisogna far corrispondere una `POP` ad ogni `PUSH`, altrimenti si lascia la pila in uno stato inconsistente per il prossimo `RET`.

1.2 Sottoprogramma principale

Il `_main` va in esecuzione come un sottoprogramma, ergo deve terminare con una `RET` e lasciare in EAX un valore di ritorno (0 significa tutto ok, $\neq 0$ significa codice di errore). Per quanto ci riguarda, basterà scrivere `XOR %EAX, %EAX`.

1.3 Dichiarazione dello stack

Lo stack esiste se viene:

1. Dichiarato con una direttiva;
2. Inizializzato con il registro ESP.

Dichiarare significa allocare abbastanza memoria, e inizializzare significa impostare ESP alla cella successiva al fondo dello stack (si ricorda che lo stack si evolve verso sinistra). Ad esempio, potremo avere:

```
.DATA
...
mystack:  .FILL 1024, 4 #dichiarazione stack
.SET      initial_esp, (mystack + 1024*4)

.TEXT
_main:    NOP
          MOV $initial_esp, %ESP  # inizializzazione stack
```

Lo stack può essere grande a piacere del programmatore. Nel nostro ambiente (ma non in generale) possiamo omettere la dichiarazione.

La pila può essere anche usata per il passaggio dei documanti (è il metodo che usano i compilatori). Questo risulta difficile da fare a mano, e quindi è sconsigliato per programmi più semplici.

1.4 Sottoprogrammi di Input/Output

In assembler non esistono istruzioni di ingresso e uscita (tranne le `IN` e `OUT`, che però sappiamo essere privilegiate). Si usano quindi i servizi del sistema (DOS), ovvero sottoprogrammi scritti da altri che girano in modalità sistema. Questi servizi sono molto primitivi: permettono l'uscita di singoli caratteri. Esistono quindi sottoprogrammi (leggermente) più sofisticati per l'output di numeri, ecc...

1.4.1 I/O tastiera e video

Le informazioni che entrano ed escono da interfacce sono solo codifice ASCII di singoli caratteri. Infatti in assembler non esiste il concetto di I/O tipato di variabili.

Ricevere il numero 32 significa ottenere i caratteri '3' e '2', mentre stamparlo significa inviare i caratteri '3' e '2'. Questo chiaramente sui decimale si traduce in moltiplicazioni per 10 (in entrata) e divisioni per 10 con resto (in uscita) atte ad ottenere queste cifre.

1.4.2 I/O di caratteri e stringhe

Nel corso si userà il file di utilità `.INCLUDE "./files/utility.s"`. Questo file mette a disposizione alcuni sottoprogrammi fra cui:

- **inchar:** mette in AL la codifica ASCII del tasto premuto;
- **outchar:** mette sul video la codifica ascii contenuta in AL;
- **newline:** stampa `0x0D` (Carriage Return) e `0x0A` (Line Feed), ergo va a capo;
- **pauseN:** mette in pausa il programma e stampa a video:

```
Checkpoint number N. Press any key to continue
```

  dove N deve essere una cifra decimale.

Sopra questi sottoprogrammi sono state scritte routine più complesse:

- **inline:**
  - **Descrizione:** porta una stringa di massimo 80 caratteri in un buffer di memoria, digitando con eco su video.
  - **Parametri di ingresso:**
    - EBX: indirizzo di memoria del buffer;
    - CX: numero di caratteri da leggere (massimo 80, una linea).

  Questo programma legge effettivamente 78 caratteri utili, in quanto gli ultimi 2 sono obbligatoriamente il nuova linea. Il programma inoltre gestisce la pressione dei tasti invio (finisci di ottenere caratteri) e backspace (cancella caratteri).

- **outline, outmess:**
  - **Descrizione:** stampa a video massimo 80 caratteri da un buffer di memoria. Si ferma prima se trova un carattere di ritorno carrello, andando anche a capo.
  - **Parametri di ingresso:**
    - EBX: indirizzo di memoria del buffer;

- **inbyte, inword, inlong:**
  - **Descrizione:** prelevano da tastiera (con eco sul video) 2, 4 o 8 caratteri. Interpretano tale sequenza di caratteri come un numero esadecimale a 2, 4 o 8 cifre. Ignorano tutti gli altri caratteri.
  - **Parametri di ingresso:**
    - AL, AX, o EAX: il numero esadecimale digitato.
- **outbyte, outword, outlong:**
  - **Descrizione:** stampano a video 2, 4 o 8 caratteri, corrispondenti a cifre esadecimali.
  - **Parametri di ingresso:**
    - AL, AX, o EAX: il numero esadecimale da stampare.
- **indecimal_byte, indecimal_word, indecimal_long:**
  - **Descrizione:** prelevano da tastiera (con eco sul video) fino a 3, 5 o 10 cifre decimali. Interpretano tale sequenza di caratteri come un numero decimale.
  - **Parametri di ingresso:**
    - AL, AX, o EAX: il numero decimale digitato.

  Se il numero decimale è troppo grande viene troncato. Inoltre si può usare invio per dare ingresso a meno cifre.
- **outdecimal_byte, outdecimal_word, outdecimal_long:**
  - **Descrizione:** stampano a video caratteri corrispondenti a cifre decimali.
  - **Parametri di ingresso:**
    - AL, AX, o EAX: il numero decimale da stampare.

1.5 Manipolazione di stringhe e vettori

In assembler non esistono tipi di dati né strutture dati. Si supporta però il concetto di vettore: si dichiarano vettori di variabili di una certa dimensione, e si indirizzano i loro elementi attraverso l'indirizzamento complesso (displacement + base + indice $*$ scala).

In verità esistono istruzioni stringa, che servono a copiare interi buffer di memoria, che sfruttano i registri ESI e EDI. Ad esempio, copiare un vettore a mano significherebbe:

```
1  vett_sorg:  .FILL 1000,4
2  vett_dest:  .FILL 1000,4
3
4              MOV $1000, %ECX
5              LEA vett_sorg, %ESI
6              LEA vett_dest, $EDI
7  ciclo:      MOV (%ESI), %EAX
8              MOV %EAX , (%EDI)
9              ADD $4, %ESI
10             ADD $4, %EDI
11             LOOP ciclo
```

ma abbiamo la possibilità di scrivere la stessa cosa come:

```
vett_sorg:  .FILL 1000,4
vett_dest:  .FILL 1000,4

MOV $1000, %ECX
          LEA vett_sorg, %ESI
          LEA vett_dest, %EDI
          REP MOVSL
```

dove l'istruzione `REP MOVSL` indica ripetizione (prefisso `REP`), di movimento da stringa a stringa su long (`MOVSL`) finché ECX $\neq 0$.

1.5.1 Direction Flag

Esiste un'altro bit utile nel registro dei flag: il Direction Flag, o DF. Si imposta con le istruzioni:

- **STD**: SET DIRECTION FLAG, la imposta ad 1;
- **CLD**: CLEAR DIRECTION FLAG, la imposta a 0;

Si usa questo flag per dare indicazioni alla prossima istruzione:

1.5.2 MOVE DATA FROM STRING TO STRING (with REPEAT)

- **Formato:** `MOVSsuf`, `REP MOVSsuf`
- **Azione:** copia il numero di byte indicato dal suffisso suf dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI. Successivamente, SE DF è 1, sottrae da ESI e EDI il numero di byte indicati da suf, altrimenti li somma.

  Se si include il prefisso, le operazioni vengono ripetute decrementando ECX (come per `LOOP`).
- **Flag:** nessuno.

Esistono poi altre istruzioni di stringa, fra cui:

1.5.3 LOAD DATA FROM STRING

- **Formato:** `LODSsuf`
- **Azione:** copia in AL, AX, oppure EAX, il contenuto della memoria all'indirizzo puntato da ESI. Successivamente incrementa o decrementa ESI di 1, 2 o 4 a seconda di DF.
- **Flag:** nessuno.

1.5.4 STORE DATA TO STRING

- **Formato:** `LODSsuf`
- **Azione:** copia il registro AL, AX, oppure EAX, in memoria all'indirizzo puntato da EDI. Successivamente incrementa o decrementa EDI di 1, 2 o 4 a seconda di DF.
- **Flag:** nessuno.

Si dovrebbe essere notato che ESI sta per sorgente, ed EDI per destinatario. Vediamo quindi degli esempi:

Copia un vettore da una parte all'altra, eseguendo un'operazione su tutti i suoi elementi:

```
        MOV $1000, %CX
        LEA buffer_src, %ESI
        LEA buffer_dst, %EDI
        CLD
ciclo:  LODSL
        ... #modifica %EAX
        STOSL
        LOOP ciclo
```

Riempi un buffer in memoria di zeri:

```
MOV $1000, %ECX
LEA buffer, %EDI
XOR %EAX, %EAX
CLD
REP STOSL
```

1.5.5 Istruzioni stringa per l'I/O

Esistono delle istruzioni stringa di ingresso e uscita:

1.5.6 INSERT STRING

- **Formato:** `INSsuf`
- **Azione:** fa ingresso di 1, 2 o 4 byte dalla porta di I/O il cui offset è contenuto in DX. L'operando viene inserito in memoria a partire dall'indirizzo contenuto in EDI. Successivamente incrementa o decrementa EDI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

1.5.7 OUTPUT STRING

- **Formato:** `INSsuf`
- **Azione:** fa uscita di 1, 2 o 4 byte dall'indirizzo di memoria contenuto in EDI. L'operando viene inserito nella porta di I/O il cui offset è contenuto in DX. Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

1.5.8 Istruzioni di confronto su stringhe

Vediamo infine alcune istruzioni per effettuare confronti su e fra stringhe:

1.5.9 COMPARE STRINGS

- **Formato:** `CMPSsuf`
- **Azione:** confronta il valore delle locazioni (singole, doppie o quadruple) indicate da ESI (sorgente) ed EDI (destinatario). Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

1.5.10 SCAN STRING

- **Formato:** `SCASsuf`
- **Azione:** confronta il contenuto del registro AL, AX o EAX con la locazione (singola, doppia o quadrupla) di memoria indirizzata da EDI. L'algoritmo di confronto è lo stesso di CMP. Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

Quest'espressione si usa per trovare valori noti dentro un vettore con, $DF = 0$ che cerca la prima occorrenza, e $DF = 1$ che cerca l'ultima. Ad esempio, poniamo di voler trovare il primo elemento differente fra due vettori:

```
array1: .WORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
array2: .WORD 1, 2, 3, 4, 7, 6, 7, 8, 9, 10

CLD
LEA array 1, %ESI
LEA array2, %EDI
MOV $10, %ECX
REPE CMPSW
```

dove si noti che alla fine del ciclo EDI e ESI puntano all'elemento successivo.

1.5.11 Prefissi di ripetizione

Vediamo nel dettaglio il prefisso `REP`, e le sue varianti `REPE` e `REPNE`. Bisogna ricordare che questi prefissi si applicano ad istruzioni, non a blocchi di codice. Nel dettaglio sono.

- `REP`: si può usare con `MOVS`, `LODS`, `STOS`, `INS` e `OUTS`, anche se l'utilizzo con `LODS` è privo di senso (almeno che non si voglia ottenere l'ultimo elemento...).
- `REPE` e `REPNE`: si può usare con `CMPS` e `SCAS`, ed effettua al massimo ECX ripetizioni, finché la condizione specificata è vera.

1.5.12 Perchè due direzioni?

L'uso di due direzioni di scorrimento di stringhe attraverso il flag DF è utile, sopratutto nel caso si debbano fare traslazioni del vettore (copia di buffer **parzialmente sovrapposti**). Infatti, cercando si spostare il vettore a destra spostandoci verso destra, finiremo per copiare sempre gli stessi dati.

1.6 Note sull'efficienza

Un compilatore ottimizza il codice in alto livello per il sistema su cui quel codice dovrà girare. Un assemblatore, invece, traduce le istruzioni una per una.

1.6.1 Tempo di esecuzione di un processo

Un processo è un programma in esecuzione con dei dati. In questo, dipende dai dati, dallo stato del sistema, e da cosa sta facendo il processore (chi lo sta usando?). Questo rende il calcolatore una macchina poco prevedibile, e il tempo di esecuzione del processo difficile da calcolare a priori. Di base, infatti:

- Il clock non va a velocità costante;
- Il vostro processo non necessariamente gira su un solo core;
- Altri meccanismi introducono variabilità considerevoli:
  - Memorie cache;
  - Code di prefetch;
  - Esecuzione in pipeline: eseguire un'istruzione significa fare fetch dell'istruzione, recuperare l'OPCODE, il sorgente, scrivere sul destinatario, ecc... conviene eseguire queste operazioni in pipeline, cioè eseguendo in parallelo più istruzioni possibili contemporaneamente;
  - Esecuzione non sequenziale: il processore non esegue necessariamente il codice nell'ordine in cui è scritto: se possibile, modifica l'ordine in modo dal caricare in modo più efficiente possibile la pipeline;
  - Branch prediction: quando si esegue in pipeline, le istruzioni condizionali creano forti bottleneck di prestazioni. Per ovviare a questo problema, il processore cerca di predire il tipo della prossima istruzione, pagando un prezzo nel caso si sbagli, ma ottenendo un significativo incremento di velocità nel caso abbia successo.

1.6.2 Lunghezza delle istruzioni e tempo di fetch

Il numero di byte occupati da un'istruzione dipende dall'OPCODE e dal tipo di indirizzamento. Se gli operandi sono **registri**, le istruzioni stanno normalmente su 1 byte; gli operandi **immediati** devono essere codificati (in 1, 2 o 4 byte); i **displacement** occupano 4 byte.

La lunghezza delle istruzioni, oltre alle dimensioni dei file binari, influenza anche il tempo di fetch delle stesse, e va quindi tenuto in considerazione.

1.6.3 Tempo di esecuzione delle istruzioni

Il tempo di esecuzione delle istruzioni dipende molto dall'architettura specifica del processore (anche in processori della stessa famiglia).

Abbiamo che le istruzioni ALU (escluse MUL e DIV) costano poco, su O(1) cicli di clock. Le MUL e DIV costano sui O(10) cicli di clock, e per questo vengono tradotte in procedure alternative (attraverso LEA o le istruzioni di shift) dai compilatori attraverso apposite tablle di corrispondenza.

Le operazioni più costose sono quelle dellA FPU (Floating Point Unit), che richiedono sulle O(100) istruzioni.

Anche le istruzioni condizionali sono molto costose, ma per i motivi visti prima che causano alla pipeline.