Soluzioni del Capitolo 3

Questo documento contiene le soluzioni ad un numero selezionato di esercizi del Capitolo 3 del libro "Calcolatori Elettronici - Architettura e organizzazione", Mc-Graw Hill 2017.
Coloro che avessero sviluppato soluzioni alternative a quelle qui proposte, o soluzioni a esercizi non compresi tra quelli qui trattati, sono invitati a trasmetterle all'indirizzo sotto riportato. Serviranno a migliorare e tenere aggiornati i contenuti di questo sito.
L'autore sarà grato nei confronti di coloro che segnaleranno errori di qualunque genere, sia nella parte che segue sia nel libro menzionato.
`giacomo.bucci@unifi.it`

Aggiornato il 18 aprile 2017

3.8 Affinché il codice resti sostanzialmente immutato nella rilocazione occorre che i campi dell istruzioni che contengono gli indirizzi non debbano essere modificati. A tal fine gli indirizzi non devono essere assoluti, bensì riferiti al contenuto di un registro (o di più registri) di CPU.

3.10 L'esercizio viene svolto per l'istruzione `MOV`. In forma generale l'istruzione `MOV` ha questo significato:

```
MOV     dest,source
```

e indica che il contenuto di `source` viene copiato in `dest`. Il precedente contenuto di `dest` viene perduto, mentre il contenuto di `source` resta invariato. Facendo riferimento al processore 8086/88, le combinazioni consentite sono le seguenti (`source`,`dest`):

- registro,memoria
- memoria,registro
- registro,registro
- memoria,immediato
- registro,immediato
- memoria,registro di segmento
- registro di segmento,memoria
- registro,registro di segmento
- registro di segmento,registro

Poiché sono previste più modalità di indirizzamento, il formato è variabile. Si riportano alcuni esempi di statement assembler.

- `MOV AX,VAR  ; EA= Offset(VAR) e AX<-M[EA]`
  indirizzamento diretto
- `MOV VAR(BX),CX  ; EA= Offset(VAR)+BX e M[EA]<-CX`
  indirizzamento relativo ai registri
  (Ovviamente per questo caso e il precedente l'indirizzo fisico è dato da DS;EA, sempre che non sia stato usato un prefisso che cambia il selettore di segmento).
- `MOV AL,BH  ; AL<-BH`
  indirizzamento di registro
- `MOV DX,2700  ;DX<-2700`
  indirizzamento immediato

Abbiamo considerato il caso del `MOV` per il processore 8086/88. Se si prendesse il manuale corrente "*Intel 64 and IA32 Architectures - Software Developer's Manual, vol. 2A: Instruction Set Reference, A-M*", si scoprirebbero ben oltre 30 codici di operazione per `MOV`, ai quali corrisponde una varietà molto ampia di formati, anche per soddisfare le versioni a 32 o 64 bit.

3.11 Si riporta la sequenza di istruzioni aggiungendo come commento l'effetto della loro esecuzione.

```
LD  R1, 10              ; R1 <- 10
LD  R2, 7               ; R2 <- 7
ADD R3, R1, R2          ; R3 <- 10 + 7 = 17
ADD R3, R3, R2          ; R3 <- 17 + 7 = 24
SUB R2, R2, R2          ; R2 <- 7 - 7 = 0
ADD R3, R3, R2          ; R3 <- 24 + 0 = 24
```

Dunque al termine `R3` contiene 24.

3.12 Al termine della sequenza AX conterrà il numero 55. Infatti, il ciclo contenuto nella sequenza non fa altro che sommare in AX i numeri da 1 a 10 (o meglio da 10 a 1). Dunque il contenuto di AX alla fine del ciclo è dato da:

$$10 + 9 + \ldots + 2 + 1 = \frac{10 \times 11}{2} = 55.$$

3.13 Si risolve l'esercizio con riferimento al caso in cui si abbia `R1`=10, `R2`=20, `R3`=30 e `R4`=40. Dopo i primi 3 `push` lo stato dello stack è quello di Figura 3.1 b).

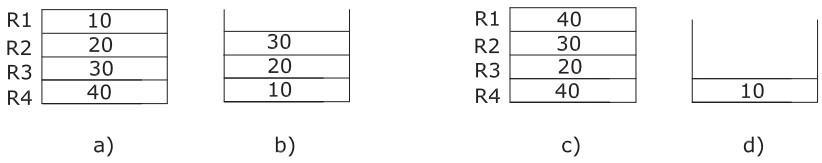

Figura 3.1 (Esercizio 3.13) a) contenuto iniziale dei registri; b) stato dello stack dopo i primi 3 push; c) stato dei registri al termine; d) stato dello stack al termine.

Il primo `POP` elimina 30 dalla testa dello stack, assegnandolo a `R2`. Il successivo `PUSH` (`PUSH R4`), deposita 40, per cui i due `POP` successivi assegnano rispettivamente i valori 40 e 20 a `R1` e `R3`, prelevandoli dallo stack. In conclusione la situazione finale dei registri e dello stack è rappresentato in Figura 3.1 c) e d) rispettivamente.

3.14 L'esercizio viene svolto in riferimento all'istruzione "MOVS".

L'istruzione `MOVS` viene utilizzata nello spostamento di stringhe (dove per stringa si intende una qualsiasi sequenza di byte, word, ecc.). L'istruzione MOVS presuppone che

- Il Flag di direzione, DF (parte della parola di stato) venga portato a 0 o 1 a seconda che la stringa venga processata secondo indirizzi crescenti o decrescenti. Il repertorio prevede le due istruzioni CLD (*CLear D*) e STD (*SeT D*).

- Il registro CX contenga il numero di iterazioni (elementi da spostare).
- L'indirizzo iniziale della stringa sorgente sia in DS:SI e quello della stringa di destinazione in ES:DI.

Limitandoci all'8086/88, l'istruzione è in diverse forme

```
MOVS <destinazione>, <sorgente>
MOVSB <destinazione>, <sorgente>     ;sposta byte
MOVSW <destinazione>, <sorgente>     ;sposta parole (16 bit)
```

La prima forma è generica e richiede che per altra via (che non stiamo a illustrare) venga indicato se si deve spostare byte o parole; il significato delle altre è evidente.

L'istruzione ha l'effetto di aggiornare i registri indice SI e DI (incrementando/decrementando a seconda di DF. Inoltre mettendo il prefisso REP (si faccia riferimento alla Figura 3.3 del teto) in fronte all'istruzione, la logica di CPU effettua automaticamente il numero di trasferimenti previsti in CX. Questo evita di costruire un loop per controllare la fine del trasferimento.

```
SOURCE  DB  'Ambarabà ciccì cuccò'      ;stringa sorgente
DEST    DB  30  DUP(?)                 ; 30 byte dove sarà spostata
        ::::::
        CLD                            ;incremento indirizzi
        MOV CX, LENGTH  SOURCE         ;CX <- n. byte della sorgente
        MOV SI, OFFSET  SOURCE         ;puntamento alla sorgente
        MOV DI, OFFSET  DEST           ;puntamento alla destinazione
        REP MOVSB                      ;trasferimento della stringa
```

Il primo statement definisce una stringa di caratteri (la stringa sorgente); il secondo semplicemente riserva uno spazio di 30 byte a partire dalla posizione `DEST`. L'operatore `LENGTH` calcola la lunghezza dell'oggetto alla sua destra. Infine il `REP` che precede `MOVSB` ha l'effetto di generare il prefisso che determina la ripetizione del `MOVSB` per quanto è il contenuto iniziale di `CX`.

3.15 Per la macchina registro-registro lo statement può essere tradotto come di seguito (si assume, ovviamente, un ampio numero di registri):

```
LD  R1, a              ; Carica le -
LD  R2, b              ; - variabili -
LD  R3, c              ; - nei singoli -
LD  R4, d              ; - registri -
LD  R5, e
MUL R6, R3, R4         ; R6 <- c * d
ADD R6, R6, R2         ; R6 <- R6 + b
SUB R6, R6, R5         ; R6 <- R6 - e
MUL R6, R6, R1         ; R6 <- R6 * a
ST  x, R6              ; Memorizza il risultato in x
```

Per una macchina registro-memoria (si assume un solo registro accumulatore implicato in tutte le operazioni.

```
LD  c
MUL d                  ; ACC <- c * d
```

```
ADD b                    ; ACC <- ACC + b
SUB e                    ; ACC <- ACC - e
MUL a                    ; ACC <- ACC * a
ST  x
```

Per una macchina a stack (TS sta per *Top of Stack*):

```
PUSH c                   ; Inserisce le due -
PUSH d                   ; - variabili
MUL                      ; TS <- c * d
PUSH b
ADD                      ; TS <- TS + b
PUSH e
SUB                      ; TS <- TS - e
PUSH a
MUL                      ; TS <- TS + a
POP  x                   ; Estrae il risultato
```

3.16 Faremo uso della terminologia relativa all' architettura dell'8086, assumendo però che lo stack si sviluppi nel senso degli indirizzi crescenti. Il primo parametro della funzione è un intero passato per valore, il secondo e il terzo sono puntatori a interi. Per passare i due puntatori a `c` e `b` si può ricorrere all'istruzione `LEA` (*Load Effective Address*).

```
LEA     AX,c          ;AX <- c
PUSH    AX            ;mette il valore nello stack
LEA     AX,b          ;AX <- b
PUSH    AX            ;mette il valore nello stack
LD      AX,a          ;carica in R1 il valore a
PUSH    AX            ;mette il valore nello stack
CALL    F             ;Chiama la subroutine F
```

Assumendo che gli interi siano a 16 bit, e che pure l'indirizzo di ritorno sia a 16 bit, dopo l'esecuzione dello statement `CALL F`, lo stato dello stack è quello di sinistra in Figura 3.2, dove BP0 e SP0 sono i valori contenuti in BP e SP prima della sequenza di chiamata.

All'ingresso della routine BP deve essere aggiornato in modo da costituire un riferimento per i parametri. Ciò richiede che prima BP venga salvato. Dunque le prime istruzioni della routine saranno:

```
F:   PUSH    BP           ;Inizializza BP
     MOV     BP,SP        ;Carica in BP il valore di SP
```

Lo stato risultante dello stack è allora quello a destra in Figura 3.2.

I parametri vengono indirizzati in modo relativo a BP. Lo statement `b=a+c` viene tradotto come:

```
MOV     AX,BP-4             ;AX <-  a
MOV     BX,BP-8             ;BX <- offset di c
ADD     AX,[BX]             ;AX <- a + c
MOV     BX,BP-6             ;BX <- offset di b
MOV     [BX],AX             ;b <- AX     (ovvero b<-a+c)
```

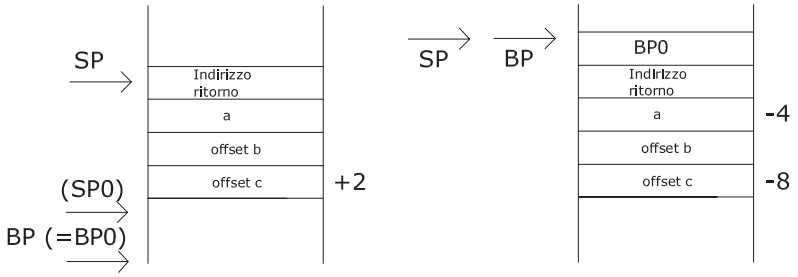

Figura 3.2 (Esercizio 3.16) A sinistra: stato dello stack all'atto dell'esecuzione di CALL F. A destra: stato dello stack dopo l'aggiornamento di BP .

3.17 Poiché si ipotizza che codice e dati siano in segmenti distinti, il passaggio dei parametri richiede la costruzione di uno pseudo stack.

Si indichi con psSTK la prima posizione dell'area che funge da pseudo stack e con x1, x2, ..., xn le variabili (che supponiamo a 32 bit). La sequenza di chiamata richiede il trasferimento di parametri in psSTK. Abbiamo ipotizzato che sia disponibile l'istruzione LDADD che carica in un registro l'indirizzo (non il contenuto) della posizione riferita. La chiamata si effettua, ad esempio, nel modo seguente:

```
psSTK   BSS     100             ;100 byte riservati (pseudo stack)
        :::
        :::
        LDADD   R1,psSTK        ;R1<- Indirizzo di psSTK
        LD      R2,Xn           ;R2<- valore parametro Xn
        ST      (R1),R2         ;--- queste 2 equivalgono
        ADD     R1,4            ;--- a push(Xn)
        :::                     ;
        :::                     ;si salva
        ADD     R1,4            ;..fino
        LD      R2,X1           ;...al primo
        ST      (R1),R2         ;....parametro
        JAL     f               ;chiamata di f
```

Entrando in `f`, il registro XFR contiene l'indirizzo di ritorno, mentre R1 punta alla testa dello stack che contiene X1. La routine `f` può prelevare i parametri indirizzando in modo relativo. Al esempio, il parametro X2 sarà preso in questo modo `LD Rx,-4(R1)`.

Vediamo ora cosa occorre prevedere per la chiamata sottoprogramma `f1` dall'interno di `f` (trascurando il passaggio dei parametri). È necessario che prima di effettuare la chiamata, `f` salvi il contenuto di XFR, in modo che al ritorno dal sottoprogramma `f1`, XFR possa essere ripristinato in previsione del ritorno al chiamante.

```
f       :::
        LDX     R2              ;R2<- XFR
        ST      saveXFR,R2      ;salvataggio indir. di ritorno
        :::
        JAL     f1              ;chiamata di f1
```

```
LD      R2,saveXFR
STX     R2              ;XFR<- indirizzo di ritorno
:::
JRX                     ;ritorno al chiamante
```

Nel precedente schema si suppone che R2 torni invariato da `f1`.

Si osservi che la routine `f` non è rientrante (salva XFR in una posizione predefinita). Saprebbe il lettore renderla rientrante?

3.19 Per il passaggio dei parametri si può fare riferimento ai metodi adottati nell' esercizio 3.17. La sequenza di chiamata sarà:

```
        JSB     F
_A      DW      A   ;parametri
_B      DW      B   ;   passati in questo
_C      DW      C   ;     ordine
```

Assumendo che la macchina abbia un numero ragionevole di registri e che questi possano essere usati per indirizzare, la routine F sarà come qui sotto. Si illustra solo la parte riferita allo statement `c=a+b` (Attenzione: non è lo statement richiesto dal testo dell'esercizio), ipotizzando che sia l'unico statement.

```
F       NOP
        LD      R2,F        ;R2 punta alle posizioni _A
        LD      R1,(R2)     ;R1 <- indirizzo di A
        LD      R0,(R1)     ;R0 <- A
        ADD     R2,4        ;parole di 4 bit
        LD      R1,(R2)     ;R2 <- indirizzo di B
        ADD     R0,(R1)     ;R0 <- A + B
        ADD     R2,4        ;
        LD      R1,(R2)
        ST      (R2),R0
        ADD     R2,4
        JR      R2          ;salto relativo a R2
```

3.20 Supponiamo che i registri siano 32 e che l'istruzione di chiamata (JAL) depositi l'indirizzo di ritorno in R31.

Se il chiamante deve salvare il contenuto dei registri in modo da poterlo ristabilire al ritorno della subroutine chiamata e se l'area deve essere determinata al tempo di esecuzione, occorre sacrificare un ulteriore registro, assumiamo R30, che dovrà comunque tornare non modificato dalla routine chiamata, in funzione di puntatore all'area in cui vengono salvati i registri.

Tralasciando il modo in cui l'area di memoria viene allocata e assumendo che in R30 ci sia il puntatore a tale area, in modo da poterlo utilizzare con una istruzione di Store con indirizzamento indiretto con indice, nel chiamante, prima della chiamata, dovrà essere presente un tratto di codice come quello che segue.

```
:::  assunzione:  R30 contiene il puntatore all'area allocata
SUB     R2,R2,R2      ;R2 <- 0  (R2 fa da indice)
ST      (R30)[R2],Ri  ;salvataggio Ri
ADD     R2,R2,4       ;scostamento della posizione successiva
:::  salvataggio degli altri registri come sopra
```

dove l'istruzione di ST viene ripetuta per tutti i registri Ri da salvare. Tra questi ci dovrà essere necessariamente R31 che contiene l'eventuale indirizzo di ritorno di chi ha chiamato il chiamante.

Al ritorno dal chiamato, R30 è non modificato e quindi si può ripetere il processo precedente in senso inverso, con l'istruzione LD, ripristinando i registri salvati.

L'area in cui vengono passati i parametri è allocata di volta in volta. Ciò richiede una funzione del tipo `malloc` del C.