Esercizi di programmazione Assembler

Giovanni Stea

a.a. 2023/24

Ultima modifica: 04/10/2023

Sommario

1 Esercizi di base di programmazione Assembler

1.1 Esercizio – calcolo del fattoriale

Si scriva un programma che calcola il fattoriale di un numero naturale (da 0 a 9) contenuto nella variabile `dato`, di tipo byte. Il risultato deve essere inserito in una variabile `risultato`, di dimensione opportuna. Si controlli che `dato` non ecceda 9. Prestare attenzione al dimensionamento della moltiplicazione.

```
.GLOBAL _main

.DATA
numero:       .BYTE 9
risultato:    .LONG 1

.TEXT
_main:        NOP

              MOV $0, %ECX
              MOV $1, %EAX
              MOV numero, %CL
              CMP $9, %CL
              JA fine
              CMP $1, %CL
              JBE fine

ciclo_f:      MUL %ECX
              DEC %CL                    # anche LOOP ciclo_f
              JNZ ciclo_f

fine:         MOV %EAX, risultato
              RET
```

Commento: devo fare una moltiplicazione, e quindi devo dimensionare il numero di bit. La moltiplicazione più grande che devo fare ha un operando a 32 bit (9!/2 sta su 32 bit). Quindi conviene stare su moltiplicazioni a 32 bit. Avrei potuto fare il ciclo diversamente, cominciando da 2 e andando verso l'alto. In tal modo, l'ultima moltiplicazione (8! * 9) ha operandi a 16 bit. Però avrei avuto il risultato in `DX_AX`, che è leggermente più scomodo da gestire (avrei dovuto spendere più tempo a riportarlo in memoria). Per portare in un registro a 32 bit un risultato che sta su due registri (ad esempio, `DX_AX`), posso fare così:

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

1.2 Esercizio: test se una stringa di bit è palindroma

Scrivere un programma che si comporta come segue:

1. prende in ingresso un numero a 16 bit, contenuto in memoria nella variabile "numero".

2. controlla se "numero" è o meno una stringa di 16 bit palindroma (cioé se la sequenza di 16 bit letta da sx a dx è uguale alla sequenza letta da dx a sx).

3. Se X è (non è) palindromo, il programma inserisce 1 (0) nella variabile a 8 bit "palindromo", che si trova in memoria

```
.GLOBAL _main
.DATA
numero:       .WORD 0xF18F
palindromo:   .BYTE 1       # scommetto sul risultato positivo.

.TEXT

_main:        NOP

              MOV numero, %AX
              MOV $8, %CL
              MOV $0, %BL

ciclo:        RCL %AH       # metto i bit di AH in BL in ordine inverso
              RCR %BL       # usando il carry come appoggio
              DEC %CL       # anche LOOP ciclo
              JNZ ciclo     # (purché inizializzi %ECX invece di %CL)

              CMP %AL, %BL
              JE termina
              MOVB $0, palindromo

termina:      RET
```

1.3 Esempio: test di primalità

Scrivere un programma che si comporta come segue:

1. prende in ingresso un numero a 16 bit, contenuto in memoria nella variabile `numero`.

2. controlla se `numero` è o meno un numero primo. Se lo è, mette in `primo` il numero 1. Altrimenti mette in `primo` il numero 0.

```
.GLOBAL _main

.DATA
numero:          .WORD 39971
primo:           .BYTE 1

.TEXT
_main:        NOP
              MOV  numero, %AX
              CMP  $2,%AX
              JBE  termina
```

```
# AX contiene il numero N su 16 bit. BX contiene il divisore. BX va
# inizializzato a 2 e portato, al piu', fino a N-1.

            MOV  $2,%BX

ciclo:      MOV  $0,%DX       #[DX,AX] contiene il numero N su 32 bit
            PUSH %AX          # salvo AX viene sporcato dalla divisione
            DIV  %BX
            POP  %AX          # si ripristina AX

            CMP  $0,%DX       # DX contiene il resto della divisione
            JE   nonprimo     # il numero ha un divisore

            INC  %BX
            CMP  %AX,%BX
            JAE  termina

# Una finezza: visto che il numero da dividere sta su 16 bit, se non
# è primo ha un divisore che sta su 8 bit (teorema di Gauss).
# Quindi, quando BH è diverso da 0, posso terminare il ciclo.

            CMP  $0, %BH
            JNE  termina
            JMP  ciclo

nonprimo:   MOVB $0,primo

termina:    RET
```

1.4 Numeri descritti in forma letterale

```
# Scrivere un programma che si comporta come segue:
# 1. emette ? e legge con eco da tastiera, facendo gli opportuni controlli,
#    le codifiche ASCII di due cifre di un numero naturale in base dieci;
# 2. se il numero è compreso fra 0 e 19:
#     a. stampa su una nuova riga il nome del numero;
#     b. torna al punto 1;
# 3. altrimenti, se il numero è maggiore di 19, termina.
#
# Esempio:
#13
#tredici
#0
#zero
#214
```

Se dovessi fare questa cosa in C++ scriverei un programma di 3 righe come segue

```
void main () {
char* stringhe[20]={“uno”; “due”; … ; “diciannove”;};
unsigned int i=0;
while (1) {
cin >> i;
if (i<20) cout << stringhe[i] << ‘\n’;
else break;
}
}
```

In cui ho una struttura **vettore a puntatori stringa** che mi consente di stabilire una corrispondenza tra un numero intero ed una serie di stringhe in memoria. In **Assembler** questa struttura dati me la devo fare a mano.
Visto che il programma richiede di **stampare** una stringa sullo schermo, vado a vedere cosa c'è tra i sottoprogrammi di utilità che possa fare al caso mio. Trovo **outline**, che stampa un certo numero di byte finché non trova un ritorno carrello (0DH). Questo è l'**unico** sottoprogramma che stampa un numero variabile di caratteri, che è quello che serve a me. Quindi è questo quello che devo usare. Vorrà dire che, nella parte dati, metterò tante stringhe di **byte** terminate da un byte 0DH.

```
str00: .ASCII "zero\n\r"
 […]
str19: .ASCII "diciannove\n\r"
```

A questo punto ci sono le stringhe in memoria, ma **non ho i puntatori**. Devo creare un **vettore di puntatori**. Il tipo **vettore** esiste in Assembler (basta definire variabili a più componenti). Il puntatore sarà l'**indirizzo (4 byte!)** della prima locazione di ciascuna stringa.

```
vett_nomi:  .LONG str00,str01,str02,str03,str04,str05,str06,str07,str08,str09
            .LONG str10,str11,str12,str13,str14,str15,str16,str17,str18,str19
```

In questo modo, per riferire la stringa di indice *i*, con *i*=0,…, 19, scriverò istruzioni del tipo:

```
MOV   vett_nomi(%ESI), %EAX
```

Avendo avuto cura di preparare SI in accordo alla seguente regola:

- se *i*=0, allora ESI=0
- se *i*=1, allora ESI=4
- se *i*=2, allora ESI=8
- se *i*=3, allora ESI=12

Quindi, SI dovrà contenere **il quadruplo del numero *i* che l'utente avrà digitato**, in quanto un long sono **quattro** byte e nell'indirizzamento con registro puntatore non si tiene conto del **tipo** della variabile, e quindi dovrò fare una delle due cose:

- sommare 4 ad ESI ogni volta che cambio elemento del vettore
- usare un fattore di scala pari a 4 per moltiplicare ESI

Scelgo la seconda, che è più leggibile.

Una volta che uno ha a disposizione la struttura dati sopra scritta, il resto del codice è relativamente semplice. Procediamo per punti:
1) devo leggere un numero di 2 cifre decimali da tastiera con eco. Ho un sottoprogramma che lo fa (`indecimal_byte`), che mi mette il numero in AL.

2) Posso controllare se è compreso tra 0 e 19, nel qual caso chiamerò la **outline** avendo cura di preparare come parametro di ingresso al sottoprogramma l'offset della stringa giusta, preparato secondo le regole viste sopra.

Andiamo al codice.

```
.GLOBAL _main
.DATA
# stringhe dei nomi dei numeri
str00: .ASCII "zero\n\r"
str01: .ASCII "uno\n\r"
str02: .ASCII "due\n\r"
str03: .ASCII "tre\n\r"
str04: .ASCII "quattro\n\r"
str05: .ASCII "cinque\n\r"
str06: .ASCII "sei\n\r"
str07: .ASCII "sette\n\r"
str08: .ASCII "otto\n\r"
str09: .ASCII "nove\n\r"
str10: .ASCII "dieci\n\r"
str11: .ASCII "undici\n\r"
str12: .ASCII "dodici\n\r"
str13: .ASCII "tredici\n\r"
str14: .ASCII "quattordici\n\r"
str15: .ASCII "quindici\n\r"
str16: .ASCII "sedici\n\r"
str17: .ASCII "diciassette\n\r"
str18: .ASCII "diciotto\n\r"
str19: .ASCII "diciannove\n\r"

# vettore contenente gli offset delle stringhe dei nomi
vett_nomi:  .LONG str00,str01,str02,str03,str04,str05,str06,str07,str08,str09
            .LONG str10,str11,str12,str13,str14,str15,str16,str17,str18,str19

.TEXT

_main:   NOP

inizio:  MOV  $'?',%AL
         CALL outchar
         CALL indecimal_byte

# verifico condizione di terminazione
         CMP  $19,%AL
         JA   fine

# immetto in EBX l'indirizzo della stringa con il nome del numero
         AND  $0x000000FF,%EAX
         MOV  %EAX,%ESI
         MOV  vett_nomi(,%ESI,4),%EBX
         MOV  $80, %CX
# stampo il nome
         CALL newline
         CALL outline
         JMP  inizio

fine:    RET
.INCLUDE "./files/utility.s"
```

1.5 Rettangolo di caratteri

```
#Scrivere un programma che si comporta come segue:
#
#1. legge da tastiera, EFFETTUANDO GLI OPPORTUNI CONTROLLI, la codifica ASCII di
#   una cifra in base dodici (0123456789AB), sia N il numero naturale
#   corrispondente;
#2. se N è uguale a zero, termina; altrimenti
#3. legge da tastiera la codifica ASCII di una lettera vocale minuscola;
#4. disegna un rettangolo pieno di larghezza 2*N caratteri ed altezza N
#   caratteri, utilizzando la vocale letta in precedenza, quindi torna al
#   punto 1.
#
#Esempio:
#A
#e
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee
#eeeeeeeeeeeeeeeeeeee

.GLOBAL _main
.TEXT

#Sottoprogramma in_12
#accetta un carattere corrispondente ad una cifra in base 12
#IN: nessun parametro
#OUT: mette in AL il numero in base 12

in_12:  CALL inchar
        CMP    $'0',%AL
        JB     in_12
        CMP    $'B',%AL
        JA     in_12
        CMP    $'9',%AL
        JBE    stampa
        CMP    $'A',%AL
        JB     in_12
stampa: CALL   outchar
        CMP    $'9',%AL
        JA     lettera
        SUB    $0x30,%AL
        RET
lettera:SUB    $0x37,%AL
        RET

#Sottoprogramma inVocale_eco
#accetta un carattere da tastiera, e se è una vocale lo stampa
#%AL:  parametro di uscita

inVocale_eco:
        CALL inchar
        CMP    $'a',%AL
        JE     dopo
        CMP    $'e',%AL
```

```
        JE      dopo
        CMP     $'i',%AL
        JE      dopo
        CMP     $'o',%AL
        JE      dopo
        CMP     $'u',%AL
        JNE     inVocale_eco
dopo:   CALL    outchar
        RET

#programma principale
_main:  NOP
start:  CALL    in_12
        CALL    newline
        CMP     $0,%AL
        JZ      stop
        MOV     %AL,%DL
        CALL    inVocale_eco
        CALL    newline

#Doppio ciclo per stampare il rettangolo
        MOV     %DL,%BL
rig:    MOV     %DL,%CL
        SHL     %CL             #moltiplicazione per due
col:    CALL    outchar
        DEC     %CL
        JNZ     col
        CALL    newline
        DEC     %BL
        JNZ     rig
        JMP     start
stop:   CALL    pause
        RET

.INCLUDE "./files/utility.s"
```