Linux Assembly Language - Appunti

Edizioni ByteMan

Presentazione

Tutto il materiale qui presentato è frutto di mie personali ricerche. E' possibile, pertanto, che ci siano incompletezze ed errori. In tal caso, contattatemi. All information provided here derived from my own research. So, mistakes and some sort of incompleteness could exist. If you find any, please, contact me.

Non potevo, passando a Linux, abbandonare il mio linguaggio preferito. Pertanto i primi lavori di prova in questo S.O. me li scrivo in ASM. Si tratta di una raccolta di appunti, numerati progressivamente, per il momento senza alcun ordine logico nella sequenza, che è determinata solo da contingenti necessità di studio e lavoro. E' probabile che in un prossimo futuro sarà possibile una migliore consultazione, al momento ci si deve accontentare di un semplice indice progressivo. Gli esempi presentati hanno prevalentemente funzione didattica e non vengono mai resi complicati per non venir meno alla loro finalità. Mi auguro, comunque, che la fatica per la raccolta del materiale e per la riorganizzazione dello stesso possa tornare utile oltre che ai miei studenti a quanti altri volessero consultare questi appunti.
Per il momento mi atterrò alle seguenti convenzioni, fissate anche in funzione del software che uso:

- Assemblatore: nasm della **NetWide**
- Linker 1: ld
- Linker 2: quello incluso in gcc
- Nomi dei file sorgente: con suffisso .asm
- Nomi dei file oggetto: con suffisso .o
- Nomi dei file eseguibili: senza suffisso

Ecco alcune delle motivazioni:

- Come assemblatore ho scelto nasm per continuare ad usare la sintassi Intel, già utilizzata anche con altri sistemi operativi.
  Non ho intenzione, per il momento, di usare la sintassi AT&T, e quindi non considererò esempi da compilare con gas.
- Utilizzerò il linker gcc solo quando sarà necessario usare le funzioni della libreria libc. Il risultato finale sarà un po' più lungo in termini di byte, ma la semplicità con cui si importano le funzioni è, per me, didatticamente più importante. Da notare, ancora, che gcc utilizza, a sua volta, il linker ld.
- In tutti gli altri casi utilizzerò, invece, il linker ld per ottenere un eseguibile più compatto.

Introduzione ASM Linux

A partire da un file sorgente è necessario eseguire due operazioni per ottenere un file eseguibile: la compilazione e il collegamento (linking). Durante la fase di compilazione viene effettuata l'analisi necessaria per verificare la correttezza del codice e, in caso di esito positivo, si passa alla traduzione del sorgente in linguaggio macchina. Dopo la fase di compilazione è necessaria una fase di collegamento (linking) in cui si vanno a collegare più file compilati ed eventuali librerie statiche e/o dinamiche per ottenere un file eseguibile correttamente caricabile dal sistema operativo. L'intero procedimento si riduce, quindi, ai seguenti 2 passaggi:

```
nasm -f elf -o prova.o prova.asm
gcc -s -o prova prova.o
```

oppure ai seguenti 2:

```
nasm -f elf -o prova.o prova.asm
ld -s -o prova prova.o
```

In tutti e due i casi il parametro -o è seguito dal nome del file di output, il parametro -f elf specifica il formato del file oggetto, il parametro -s ordina di effettuare lo strip delle funzioni inutili dal file binario di uscita.

Per facilitare la compilazione si possono, in alternativa, usare i seguenti script, migliorabili sicuramente, ma è una cosa che farò in un secondo tempo:

```
#!/bin/sh                              #!/bin/sh
# ag.sh ::::::::::::::::::::::         # ag.sh ::::::::::::::::::::::::::::::::::::::
nasm -f elf -o $1.o $1.asm             nasm -f elf -o $1.o $1.asm && gcc -s -o $1 $1.o
gcc -s -o $1 $1.o             oppure   rm $1.o -f
rm $1.o -f                             #EOF ::::::::::::::::::::::::::::::::::::::::
#EOF ::::::::::::::::::::::::


#!/bin/sh                              #!/bin/sh
# al.sh ::::::::::::::::::::::         # al.sh ::::::::::::::::::::::::::::::::::::::
nasm -f elf -o $1.o $1.asm             nasm -f elf -o $1.o $1.asm && ld -s -o $1 $1.o
ld -s -o $1 $1.o              oppure   rm $1.o -f
rm $1.o -f                             #EOF ::::::::::::::::::::::::::::::::::::::::
#EOF ::::::::::::::::::::::::
```

La compilazione con l'uso di uno degli script si eseguirà quindi con:

```
./al.sh prova
```

Oltre alla compilazione, lo script provvederà a rimuovere il file oggetto, generalmente non utilizzato, con il comando rm.

L'esecuzione del programma, compilato nella propria cartella di lavoro, si eseguirà digitando:

```
./prova
```

Quasi il classico "Hello world"

Il primo esempio leggerà il marchio di fabbrica della CPU e lo visualizzerà sul terminale. Il codice è scritto per macchine di classe Pentium e non esegue alcun controllo per i 486 o precedenti.

```
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; hello_1.asm
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; compilare con nasm+gcc

global  main
extern  printf

section .data
cls     db 1Bh,"[2J",0
msg     db 0Dh,0Ah,"Marchio del produttore della CPU: "
idx     dd 0,0,0
        db 0Dh,0Ah,0Ah,0

section .text
main:   push    dword cls       ;Clear del terminale
        call    printf
        pop     eax
        mov     eax,0           ;Identificazione CPU
        cpuid
        mov     [idx],ebx       ;Lettura prime 4 lettere
        mov     [idx+4],edx     ;Lettura 4 lettere centrali
        mov     [idx+8],ecx     ;Lettura ultime 4 lettere
        push    dword msg       ;Output Messaggio
        call    printf
        pop     eax
        ret                     ;Uscita programma
; EOF ::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

L'esempio appena illustrato esegue quattro operazioni: la pulizia del terminale, l'identificazione della CPU, l'emissione di un messaggio, l'uscita dal programma. L'uso della printf, appartenente alla libreria libc, semplifica le operazioni di output su terminale (clear e messaggio), si noti il passaggio del parametro (puntatore alla stringa) tramite una push con conseguente pop per ripristinare lo stato dello stack. Al posto della pop eax, che coinvolge l'uso di un registro, si potrebbe usare una add esp,4 che risolve il problema del riallineamento dello stack senza coinvolgere registri. L'uscita del programma realizzata con una semplicissima ret.

Ci proponiamo di risolvere lo stesso problema con l'uso di chiamate al kernel, ovvero con le cosiddette syscall. La soluzione è leggermente più complessa, ma è di particolare interesse il passaggio dei parametri tramite registri; si noti, in particolare, che il registro eax viene utilizzato per individuare il numero della syscall da usare, mentre i parametri (par1, par2, par3, ...) vengono passati ordinatamente tramite i registri ebx, ecx, edx, esi, edi, ebp, rispettivamente.
Il confronto con la soluzione precedente evidenzia alcune cose: l'assenza della dichiarazione extern, la mancanza del terminatore null alla fine delle stringhe, la presenza di una costante per l'individuazione della lunghezza delle stringhe.

```
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; hello_2.asm
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; compilare con nasm+ld

global  main

section .data
cls     db 1Bh,"[2J"
CLS     equ     $-cls
msg     db 0Dh,0Ah,"Marchio del produttore della CPU: "
idx     dd 0,0,0
        db 0Dh,0Ah,0Ah
MSG     equ     $-msg

section .text
main:                           ;Clear del terminale
        mov     edx,CLS         ;par3: lunghezza del messaggio
        mov     ecx,cls         ;par2: indirizzo del messaggio
```

```
        mov     ebx,1           ;par1: descrittore del file (stdout)
        mov     eax,4           ;numero della syscall write
        int     80h             ;chiamata kernel syscall
        mov     eax,0           ;Identificazione CPU
        cpuid
        mov     [idx],ebx       ;Lettura prime 4 lettere
        mov     [idx+4],edx     ;Lettura 4 lettere centrali
        mov     [idx+8],ecx     ;Lettura ultime 4 lettere
                                ;Output Messaggio
        mov     edx,MSG         ;par3: lunghezza del messaggio
        mov     ecx,msg         ;par2: indirizzo del messaggio
        mov     ebx,1           ;par1: descrittore del file (stdout)
        mov     eax,4           ;numero della syscall write
        int     80h             ;chiamata kernel syscall
                                ;Uscita programma
        mov     ebx,0           ;Par1: codice di ritorno
        mov     eax,1           ;numero della syscall exit
        int     80h             ;chiamata kernel syscall
; EOF ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Se effettuiamo un confronto sulla lunghezza dei due eseguibili ottenuti, anche se i sorgenti non sono perfettamente identici, possiamo notare l'evidente maggior compattezza di hello_2 rispetto ad hello_1.

```
-rwxr-xr-x  1 byteman users 2972 nov  1 19:02 hello_1*
-rwxr-xr-x  1 byteman users  576 nov  1 19:04 hello_2*
```

Complementi

Per quanto riguarda l'identificazione della CPU, ecco quale stringa viene restituita a seconda dei principali produttori:

```
                        AMD     "AuthenticAMD"
                        Centaur "CentaurHauls"
                        Cyrix   "CyrixInstead"
                        Intel   "GenuineIntel"
                        NexGen  "NexGenDriven"
                        Rise    "RiseRiseRise"
                        UMC     "UMC UMC UMC "
```

Input/Output di testo

Nell'esempio seguente vogliamo coinvolgere alcune funzioni base necessarie in quasi tutti i programmi: l'input da tastiera, l'elaborazione, l'output su video e l'output di errori. Prenderemo in considerazione la trasformazione in maiuscolo di una stringa immessa da tastiera. Utilizzeremo solo syscall e pertanto potremo eseguire il link direttamente con ld. Creeremo le condizioni per generare un errore, nel caso di input nullo, in modo da utilizzare tre descrittori di file: STDIN per l'input, STDOUT per l'output normale, STDERR per l'output d'errore.

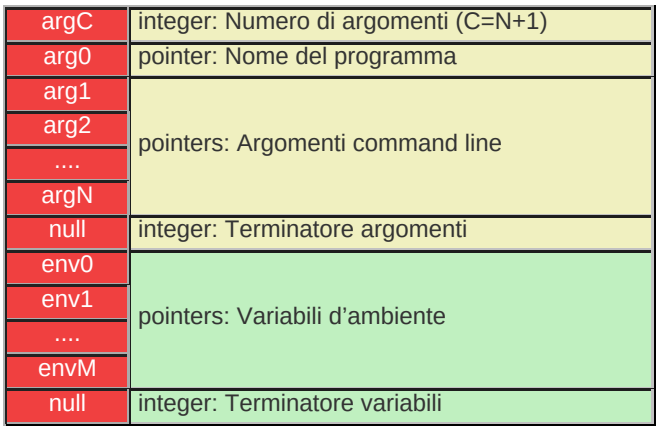
```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; upper_1.asm
; Converte in maiuscolo l'input dell'utente.
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; compilare con nasm+ld

%define STDIN 0
%define STDOUT 1
%define STDERR 2
%define SYSCALL_EXIT  1
%define SYSCALL_READ  3
%define SYSCALL_WRITE 4
%define BUFLEN 256


        section .data                   ; Sezione dati inizializzati
msg1:   db "Digita una stringa: "       ; prompt
MSG1:   equ $-msg1                      ; lunghezza di msg1
msg2:   db "Originale:  "               ;
MSG2:   equ $-msg2                      ; lunghezza di msg2
msg3:   db "Convertita: "               ;
MSG3:   equ $-msg3                      ; lunghezza di msg3
msg4:   db 10, "ERRORE: input nullo.", 10    ; messaggio d'errore
MSG4:   equ $-msg4                      ;lunghezza di msg4

        section .bss                    ; Sezione dati non inizializzati
buf:    resb BUFLEN                     ; buffer di input da tastiera
newstr: resb BUFLEN                     ; buffer di output conversione
rlen:   resb 4                          ; lunghezza stringa in input (incluso l'invio)

        section .text                   ; Sezione codice
        global  _start                  ; let loader see entry point

_start:
;::::::::::::::::::::::::::::::::::::: Output Prompt
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg1               ; Arg2: addr of message
        mov     edx, MSG1               ; Arg3: length of message
        int     080h                    ; syscall kernel to write

;::::::::::::::::::::::::::::::::::::: Input da tastiera
        mov     eax, SYSCALL_READ       ; read function
        mov     ebx, STDIN              ; Arg 1: file descriptor
        mov     ecx, buf                ; Arg 2: address of buffer
        mov     edx, BUFLEN             ; Arg 3: buffer length
        int     080h                    ; syscall kernel to read

;::::::::::::::::::::::::::::::::::::: Test Errore
        mov     [rlen], eax             ; salva la lunghezza della stringa
        dec     eax                     ; decrementa per escludere l'invio
        cmp     eax, 0                  ; Test su zero
        jg      read_OK                 ; Uscita read_OK se Test>0

;::::::::::::::::::::::::::::::::::::: Output messaggio d'errore
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDERR             ; Arg1: file descriptor
        mov     ecx, msg4               ; Arg2: addr of message
        mov     edx, MSG4               ; Arg3: length of message
        int     080h                    ; syscall kernel to write
        jmp     exit1                   ; salta all'uscita: errore =1
```

```
;:::::::::::::::::::::::::::::::::::::::: Conversione stringa
read_OK:
        mov     ecx, [rlen]             ; inizializza il contatore
        mov     esi, buf                ; puntatore al buffer di input
        mov     edi, newstr             ; puntatore al buffer di conversione
nextch: mov     al, [esi]               ; legge un carattere
        inc     esi                     ; aggiorna il puntatore sorgente
        cmp     al, 'a'                 ; verifica che sia minuscolo
        jb      store                   ; Se NO: Salta la conversione
        cmp     al, 'z'
        ja      store                   ; Se NO: Salta la conversione
        and     al,0DFh                 ; Se SI: Converte in maiuscolo
store:  mov     [edi], al               ; memorizza il carattere
        inc     edi                     ; aggiorna il puntatore destinazione
        dec     ecx                     ; aggiorna il contatore
        jnz     nextch                  ; ricicla

;:::::::::::::::::::::::::::::::::::::::: Output messaggio 2
        mov     eax, SYSCALL_WRITE      ; write message
        mov     ebx, STDOUT
        mov     ecx, msg2
        mov     edx, MSG2
        int     080h

;:::::::::::::::::::::::::::::::::::::::: Output stringa originale
        mov     eax, SYSCALL_WRITE      ; write user input
        mov     ebx, STDOUT
        mov     ecx, buf
        mov     edx, [rlen]
        int     080h

;:::::::::::::::::::::::::::::::::::::::: Output messaggio 3
        mov     EAX, SYSCALL_WRITE      ; write message
        mov     EBX, STDOUT
        mov     ECX, msg3
        mov     EDX, MSG3
        int     080h

;:::::::::::::::::::::::::::::::::::::::: Output stringa convertita
        mov     EAX, SYSCALL_WRITE      ; write out string
        mov     EBX, STDOUT
        mov     ECX, newstr
        mov     EDX, [rlen]
        int     080h

;:::::::::::::::::::::::::::::::::::::::: Uscita
        mov     EBX, 0                  ; exit code, normal=0
        jmp     exit
exit1:  mov     EBX, 1                  ; exit code, error=1
exit:   mov     EAX, SYSCALL_EXIT       ; exit function
        int     080h                    ; syscall kernel to take over
; EOF ::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Tra le cose da notare, l'inserimento della sezione .bss per la definizione di variabili non inizializzate (i buffer: buf e newstr), in contrapposizione alla sezione .data; ed ancora le pseudoistruzioni %define per introdurre delle costanti. Infine la funzione exit riporta in ebx un codice di uscita che vale "0" se tutto si è concluso regolarmente, oppure "1" in caso di uscita su STDERR.

Recuperare la command line, e non solo

Quando un programma elf viene lanciato eredita una serie di informazioni depositate come dword sullo stack. Queste informazioni riguardano sia la command line, completa di tutti gli argomenti, sia le variabili d'ambiente associate. La situazione dello stack, all'inizio dell'esecuzione, è schematizzata nella figura seguente.



L'esempio seguente estrae dallo stack tutto il contenuto schematizzato nella tabella precedente e lo visualizza sul terminale. La conversione del contatore è effettuata per semplicità, nell'ipotesi che il suo valore sia minore di 10, aggiungendo semplicemente una costante opportuna.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; commline_1.asm
; Visualizza command line e variabili d'ambiente.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; compilare con nasm+ld

%define STDOUT 1
%define SYSCALL_EXIT  1
%define SYSCALL_WRITE 4

global  _start                          ;necessario per il linker ld

        section .data                   ; Sezione dati inizializzati
msg1    db      0Dh,0Ah,"Numero di argomenti: "
argc    dd      0
        db      " incluso il nome del programma.",0Dh,0Ah
MSG1    equ     $-msg1
msg2    db      0Dh,0Ah,"Elenco argomenti: ",0Dh,0Ah
MSG2    equ     $-msg2
msg3    db      0Dh,0Ah,"Elenco variabili d'ambiente: ",0Dh,0Ah
MSG3    equ     $-msg3
crlf    db      0Dh,0Ah
xflag   dd      0

        section .text
_start:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Visualizza Numero degli Argomenti
        pop     ecx                     ; Estrae il contatore degli argomenti
        add     ecx,20202030h           ; Conversione in ASCII (Ipotesi N<10)
        mov     [argc],ecx              ; Inserimento nel corpo di msg1
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg1
        mov     edx, MSG1
        int     80h                     ; syscall kernel to write
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Visualizza Messaggio Primo Elenco
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg2
        mov     edx, MSG2
```

```
        int     80h                     ; syscall kernel to write
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Ciclo principale estrazione/visualizzazione
                                        ;    utilizzato sia per gli argomenti sia per
                                        ;    le variabili d'ambiente
xloop:  pop     ecx                     ; Estrae un puntatore
        or      ecx,ecx                 ; Verifica se null
        jz      exit1                   ; Se SI: va a exit1
                                        ; Se NO: prosegue
        mov     esi,ecx                 ; Utilizza Puntatore esi per usare la lodsb
        xor     edx,edx                 ; Azzera contatore lunghezza (edx)
        dec     edx
strlen: inc     edx                     ; calcola la lunghezza di msg
        lodsb
        or      al,al
        jnz     strlen
                                        ; Visualizza msg
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        int     80h                     ; syscall kernel to write
        call    newln                   ; Aggiunge un newline
        jmp     short xloop             ; Ricicla
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Controllo per il secondo giro
exit1:  mov     eax,[xflag]
        inc     eax
        mov     [xflag],eax
        cmp     eax,2
        je      exit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Visualizza Messaggio Secondo Elenco
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg3
        mov     edx, MSG3
        int     80h                     ; syscall kernel to write
        jmp     short xloop             ; Rientra Secondo giro
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Uscita
exit:   mov     EBX, 0                  ; exit code, normal=0
        mov     EAX, SYSCALL_EXIT       ; exit function
        int     80h                     ; syscall kernel to take over

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Routine NewLine
newln:  mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, crlf
        mov     edx, 2
        int     80h                     ; syscall kernel to write
        ret
; EOF ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Poichè le variabili d'ambiente sono molte è opportuno avviare il programma con:

```
./commline_1 ... ... ... |more
```

in modo da potere scorrere agevolmente tutto l'output. Si noterà, tra l'altro, che l'ultima voce delle variabili d'ambiente contiene ancora il nome del programma, che era già presente come arg0 nella lista degli argomenti.

Talvolta è necessario, invece, avere dei puntatori diretti ad alcuni punti speciali dello stack, senza dovere essere costretti a scaricarlo tutto, ecco alcuni suggerimenti:

```
        pop     eax                     ; Lettura contatore degli argomenti
        pop     esi                     ; Il reg. esi punta il primo degli argomenti
oppure
        pop     eax                     ; Lettura contatore degli argomenti
        mov     esi,[esp+eax*4]         ; Il reg. esi punta l'ultimo degli argomenti
oppure
        pop     eax                     ; Lettura contatore degli argomenti
        mov     esi,[esp+(eax+1)*4]     ; Il reg. esi punta la prima var. d'ambiente
```

E' evidente che queste istruzioni devono essere usate proprio all'inizio del programma, prima di avere alterato lo stato dello stack.

L’esempio precedente ha, secondo me, l’inconveniente di alterare la situazione dello stack in quanto c’è una palese quantità di istruzioni di pop non equilibrate o da altrettante istruzioni di push o da una opportuna modifica del valore del registro esp. A tal proposito, c’è una scuola di pensiero che vuole che il programmatore, all’uscita del programma, lasci il puntatore di stack con lo stesso valore trovato all’inizio.
Quella che segue è una variante breve dell’esercizio precedente che illustra questo secondo modo di agire, per semplicità tratta soltanto gli argomenti della command line.

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; commline_2.asm
; Visualizza la command line
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; compilare con nasm+gcc

%define STDOUT 1
%define SYSCALL_WRITE 4

        global  main
        extern  printf

        section .data
        msg1    db      0Dh,0Ah,"Numero di argomenti: "
argc    dd      0
        db      " incluso il nome del programma.",0Dh,0Ah
MSG1    equ     $-msg1
format: db      ’%s’, 10, 0             ; Stringa di formato

        section .text
main:
;:::::::::::::::::::::::::::::::::::::
        mov     ecx, [esp+4]            ; Lettura di argC
        push    ecx                     ; Salvataggio ecx
        add     ecx,20202030h           ; Conversione in ASCII (Ipotesi N<10)
        mov     [argc],ecx              ; Inserimento nel corpo di msg1
        mov     eax, SYSCALL_WRITE      ; write function
        mov     ebx, STDOUT             ; Arg1: file descriptor
        mov     ecx, msg1
        mov     edx, MSG1
        int     80h                     ; syscall kernel to write
        pop     ecx                     ; Ripristino ecx
        mov     edx, [esp+8]            ; Lettura puntatore di arg0
;:::::::::::::::::::::::::::::::::::::: Ciclo di lavoro
xloop:  push    ecx                     ; Salvataggio registri usati da printf
        push    edx                     ;
;.....................................
        push    dword [edx]             ;    Passa puntatore stringa
        push    dword format            ;    Passa stringa formato
        call    printf                  ;    Visualizza argomento
        add     esp, 8                  ;    Rimozione parametri dallo stack
;.....................................
        pop     edx                     ; Ripristino registri
        pop     ecx                     ;
        add     edx, 4                  ; Modifica puntatore ad arg successivo
        dec     ecx                     ; Decrementa contatore
        jnz     xloop                   ; Ricicla
        ret
;EOF ::::::::::::::::::::::::::::::::::::::::::::::::
```

Abbiamo usato il linker gcc in quanto si è preferito, per semplicità, effettuare l’output tramite la printf; si noti, a questo proposito, l’uso della stringa di formato dentro il segmento .data per concorrere al formato della printf. Il puntatore di stack esp, alla fine del programma, rimane inalterato.

I colori sul terminale e le macro

Gli esercizi di questa sezione hanno un duplice scopo:

- realizzare l'output su terminale di un testo a colori
- utilizzare le macro in un programma nasm.

Il primo problema lo risolviamo con le sequenze di escape. In sostanza inviando una stringa opportuna al terminale è possibile reimpostare gli attributi colore. Questa stringa, in particolare, che chiameremo foba (foreground background) è composta da 9 byte ed ha la seguente struttura:

1Bh	5Bh	33h	----	3Bh	34h	----	6Dh	00h
esc	[	3		;	4		m	null
0	1	2	3	4	5	6	7	8

I byte nelle posizioni 3 e 6 rappresentano i colori di foreground e di background rispettivamente, e vanno impostati secondo la seguente tabella:

30h	0	nero
31h	1	rosso
32h	2	verde
33h	3	ocra
34h	4	blu
35h	5	fucsia
36h	6	celeste
37h	7	bianco

Ad esempio per impostare i colori rosso/blu (rosso su fondo blu) occorrerà definire una stringa che contenga in posizione 3 il byte 31h ed in posizione 6 il byte 34h, come illustrato di seguito:

```
        foba    db 1Bh,5Bh,33h,31h,3Bh,34h,34h,6Dh,00h
oppure
        foba    db 1Bh,"[31;44m",0
```

Tornando al problema del testo colorato sul terminale, una prima soluzione viene proposta nel listato seguente nel quale sono presenti delle macro sviluppate nell'ultimo listato. Per mia convenzione, i nomi delle macro asm iniziano con il carattere @ in modo che sia facile individuarle nel listato del programma. Si noti la presenza della direttiva %include che consente l'aggancio del file con le macro.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; txtcol_1.asm
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;compilare con nasm+gcc

%include "macrobase.mac"

global  main
extern  printf

section .data
foba    db 1Bh,"[37;40m",0
cls     db 1Bh,"[2J",1Bh,"[1;1f",0
crlf    db 0Dh,0Ah,0
msg20   db "Scritta in verde su fondo nero",0
msg02   db "Scritta in nero su fondo verde",0
msg71   db "Scritta in bianco su fondo rosso",0

section .text
main:   @outterm   cls                  ; Clear del terminale

        @fixcol    2,0                  ; Imposta Verde/Nero
        @outterm   msg20                ; Emette Messaggio
```