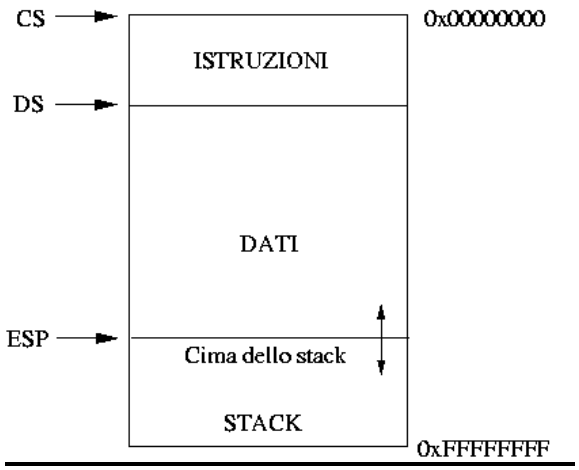
Funzioni e parametri

Gestione dello stack

Una parte della memoria principale a disposizione della CPU viene organizzata come una pila (STACK) di tipo LIFO (last-in-first-out). Il registro ESP (Extended Stack Pointer) tiene traccia dell'indirizzo dell'elemento che si trova sulla "cima" della pila mentre il registro EBP (Extended Base Pointer) punta alla base della pila (vedere lezione 1).

Comando	Descrizione
`push src`	Memorizza in cima allo stack il valore contenuto nell'operando `src`. Può essere seguito solo dalle lettere `w` o `l`. Il registro ESP viene decrementato di tanti byte quanti ne vengono messi sullo stack (4 con `l` e 2 con `w`).
`pop dest`	Estrae il contenuto della locazione di memoria in cima allo stack e lo memorizza nell'operando `dest`. Può essere seguito solo da `w` o `l`. Il registro ESP viene incrementato di tanti byte quanti ne vengono tolti dallo stack (4 con `l` e 2 con `w`).

NOTA: è possibile accedere direttamente e modificare il contenuto dei registri ESP ed EBP ma bisogna prestare particolare attenzione alle modifiche compiute.
NOTA: il recupero di informazioni dallo stack mediante più chiamate dell'istruzione `pop` deve avvenire nell'ordine inverso del loro inserimento nello stack con le istruzioni `push`.



Le funzioni

Una funzione in Assembly si definisce mediante il seguente costrutto:

```
.type nome_funzione, @function

istruzione 1
istruzione 2
… …
istruzione n

ret
```

La funzione può essere richiamata utilizzando l'istruzione:

```
call nome_funzione
```

Gli eventuali parametri della funzione devono essere caricati nei registri della CPU o nello stack. All'interno del corpo della funzione i parametri possono essere recuperati accedendo ai registri o allo stack che sono stati impostati prima della chiamata alla funzione.

Chiamata a funzione

Comando	Descrizione
`call etichetta`	Memorizza in cima allo stack l'indirizzo dell'istruzione successiva alla call e trasferisce l'esecuzione alla locazione di memoria corrispondente ad etichetta.
`Ret`	Estrae dalla cima dello stack un valore, lo interpreta come indirizzo di un'istruzione e fa saltare il processore a tale istruzione (il valore sullo stack corrisponde all'indirizzo dell'istruzione successiva all'istruzione call).

NOTA: al momento dell'esecuzione della `ret` in cima allo stack deve essere presente il valore che era stato messo dalla call. Quindi all'interno della funzione il numero di istruzioni `pushw` (e `pushl`) deve essere uguale al numero di istruzioni `popw` (e `popl`) affinché al momento dell'esecuzione della `ret` lo stack sia nelle stesse condizioni in cui si trovava all'inizio della funzione.

Organizzazione del programma su più file

Per aumentare la leggibilità del codice sorgente, è possibile organizzare il programma su più file, dedicando un file per il blocco di codice principale (quello che inizia con `_start`) e uno per ciascuna delle funzioni.

NOTA: affinché **gdb** riesca a visualizzare correttamente il contenuto delle variabili in memoria con i comandi `x/1w &nome_variabile` oppure `x/1b &nome_variabile` occorre che esse abbiano nomi diversi anche se dichiarate in file diversi.

Esempio

Il seguente programma è composto da due moduli contenuti in file sorgente separati:

1. `main.s` contiene il programma principale che chiama la funzione *itoa* per stampare un numero
2. `itoa.s` dichiara la funzione *itoa* per convertire un numero in stringa e stampare un numero decimale un carattere alla volta. Si noti il comando `.global itoa` all'inizio di `itoa.s` ; esso serve per rendere visibile all'esterno del file il nome della funzione in modo che il programma principale possa richiamarla. Tale comando va messo all'inizio di tutte le funzioni che devono essere chiamate da codice scritto in altri file.

Per assemblare e linkare i moduli nell'eseguibile di nome `esempio` usare i comandi:

```
as -gstabs -o main.o main.s
as -gstabs -o itoa.o itoa.s
ld itoa.o main.o -o esempio
```

main.s

```
####################
# filename: main.s
####################

.section .text
      .global _start

_start:

  movl $100, %eax     # metto il numero da stampare in EAX

  call  itoa          # chiamata alla funzione itoa

  # syscall EXIT
  xorl  %eax, %eax    # azzera eax
  inc   %eax          # incr. eax di 1 (1 e' il codice della exit)
  xorl  %ebx, %ebx    # azzera ebx (alla exit viene passato 0)
  int   $0x80         # invoca la funzione exit
```

itoa.s

```
####################
# filename: itoa.s
####################

.section .data
car:
  .byte 0             # la variabile car è dichiarata di tipo byte

.section .text

.global itoa          # rende visibile il simbolo itoa al linker

.type itoa, @function # dichiarazione della funzione itoa
                      # la funzione converte un intero in una stringa
                      # il numero da convertire deve essere
                      # stato caricato nel registro eax
itoa:
  mov   $0, %ecx      # carica il numero 0 in ecx

continua_a_dividere:

  cmp   $10, %eax   # confronta 10 con il contenuto di eax
  jge   dividi      # salta all'etichetta dividi se eax è
                    # maggiore o uguale di 10
  pushl %eax        # salva nello stack il contenuto di eax
  inc   %ecx        # incrementa di 1 il valore di ecx per
                    # contare quante push eseguo
                    # ad ogni push salvo nello stack una cifra del
                    # numero (a partire da quella meno significativa)
  mov   %ecx, %ebx  # pone il valore di ecx in ebx
  jmp   stampa      # salta all'etichetta stampa

dividi:
  movl  $0, %edx   # carica 0 in edx
  movl  $10, %ebx  # carica 10 in ebx
  divl  %ebx       # divide per ebx (10) il numero ottenuto
                   # concatenando il contenuto di dx e ax (notare che
                   # in questo caso dx=0)
                   # il quoziente viene messo in eax, il resto in dx
  pushl %edx       # salva il resto nello stack
  inc   %ecx       # incrementa il contatore delle cifre da stampare
  jmp   continua_a_dividere

stampa:
  cmp   $0, %ebx  # controlla se ci sono (ancora) caratteri da
                  # stampare
  je    fine_itoa # se ebx=0 ho stampato tutto, quindi salto alla fine
  popl  %eax      # preleva l'elemento da stampare dallo stack
  movb  %al, car  # memorizza nella variabile car il valore contenuto
                  # negli 8 bit meno significativi del registro eax
                  # gli altri bit del registro non ci interessano
                  # visto che una cifra decimale e' contenuta in
                  # un solo byte
  addb  $48, car  # somma al valore car il codice ascii del carattere
                  # '0' (zero)
  dec   %ebx      # decrementa di 1 il numero di cifre da stampare
```

```
  pushw %bx       # salviamo il valore di bx nello stack poiché
                  # per effettuare la stampa dobbiamo modificare
                  # i valori dei registri come richiesto
                  # dalla funzione del sistema operativo WRITE
  movl  $4, %eax  # codice della funzione write
  movl  $1, %ebx  # la write scrive nello standard output
                  # identificato dal file descriptor 1
  leal  car, %ecx # il puntatore della stringa da stampare deve
                  # essere caricato in ecx
                  # l'istruzione lea carica l'indirizzo della
                  # locazione di memoria indicata dall'etichetta car,
                  # nel registro ecx
  mov   $1, %edx  # la lunghezza della stringa da stampare deve
                  # essere caricata in edx
  int   $0x80     # chiamata all'interrupt 0x80 per la stampa di car
  popw   %bx      # recupera il contatore dei caratteri da stampare
                  # salvato nello stack prima della chiamata alla
                  # funzione write
  jmp   stampa    # ritorna all'etichetta stampa per stampare il
                  # prossimo carattere. Notare che il blocco di
                  # istruzioni compreso tra l'etichetta stampa
                  # e l'istruzione jmp stampa e' un classico
                  # esempio di come creare un ciclo while in assembly

fine_itoa:
  movb  $10, car  # copia il codice ascii del carattere line feed
                  # (per andare a capo riga) nella variabile car

  movl  $4, %eax  # solito blocco di istruzioni per la stampa
  movl  $1, %ebx
  leal  car, %ecx
  mov   $1, %edx
  int   $0x80
  ret             # fine della funzione itoa
                  # l'esecuzione riprende dall'istruzione sucessiva
                  # alla call che ha invocato itoa
```

Esempio di makefile:

```
EXE= eseguibile
AS= as --32
LD= ld -m elf_i386
FLAGS= -gstabs
OBJ= main.o itoa.o

$(EXE): $(OBJ)
      $(LD) -o $(EXE) $(OBJ)
main.o: main.s
      $(AS) $(FLAGS) -o main.o main.s
itoa.o: itoa.s
      $(AS) $(FLAGS) -o itoa.o itoa.s
clean:
      rm -f *.o $(EXE) core
```

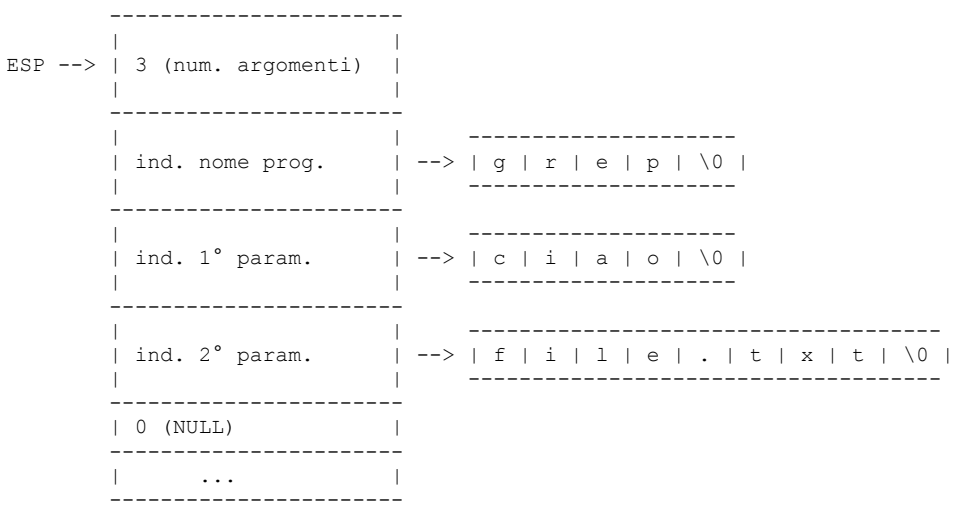
Accesso ai parametri del main

Quando si lancia un programma è possibile fornire un elenco di valori (parametri) che possono essere utilizzati dal programma stesso durante la sua esecuzione. Si pensi ad esempio al programma `grep`. Esso è un programma Unix che permette di cercare una stringa all'interno di un file di testo. La stringa da cercare e il file su cui eseguire la ricerca vengono forniti come parametri della riga di comando. Ad esempio, per cercare la stringa "ciao" dentro al file `file.txt` si deve scrivere la seguente riga di comando:

```
grep ciao file.txt
```

Come è possibile accedere dall'interno di un programma Assembly ai parametri forniti nella riga di comando? I parametri sono disponibili in locazioni di memoria i cui indirizzi sono memorizzati sullo stack. Le informazioni sono rappresentate su 32 bit (e quindi vanno recuperate con `popl` oppure saltando di 4 byte in 4 byte). La cima dello stack contiene il numero di parametri passati nella riga di comando (compreso il nome del programma), nella posizione successiva si trova l'indirizzo di memoria in cui è memorizzato il nome del programma, nella posizione successiva si trova l'indirizzo di memoria in cui è memorizzato il primo parametro, … e così via. L'elenco dei parametri termina con il valore speciale `NULL` (numero 0 su 32 bit).

Ecco come si presenta lo stack appena viene avviato `grep ciao file.txt`



Il nome del programma e tutti i parametri (anche quelli numerici) sono codificati come stringhe. Ciascuna stringa è un vettore di byte, uno per carattere, contenente il suo codice ASCII più un byte messo a 0 per indicare la fine della stringa. Nello stack si trovano gli indirizzi delle locazioni di memoria che contengono il primo carattere di ogni stringa.

Esempio

Il seguente programma recupera dalla riga di comando l'elenco dei parametri e li stampa a video.

```
.section .data
new_line_char:
  .byte 10

.section .text
.align 4

.global _start

_start:
  movl  %esp, %ebp      # Salva una copia di ESP in EBP per poter
                        # modificare ESP senza problemi.
                        # In questo punto dell'esecuzione ESP contiene
                        # l'indirizzo di memoria della locazione in cui
                        # si trova il numero di argomenti passati alla
                        # riga di comando del programma.

ancora:
  addl  $4, %esp        # Somma 4 al valore di ESP. In tal modo ESP
                        # punta al prossimo elemento sulla cima dello
                        # stack, che contiene l'indirizzo di memoria
                        # del prossimo parametro della riga di comando.
                        # Alla prima iterazione, dopo questa
                        # istruzione, ESP punta all'elemento dello
                        # stack che contiene l'indirizzo della
                        # locazione di memoria che contiene il nome del
                        # programma.

  movl (%esp), %eax     # Copia in EAX il contenuto della locazione
                        # di memoria puntata da ESP, cioè copia in EAX
                        # il puntatore al prossimo parametro della riga
                        # di comando (oppure NULL se non ci sono altri
                        # parametri).

  testl %eax, %eax      # Controlla se EAX contiene NULL (cioè 0). In
                        # tal caso significa che ho già recuperato
                        # tutti i parametri.

  jz    fine_ancora     # Esce dal ciclo se non ci sono altri parametri
                        # da recuperare.

  call  stampa_parametro  # Richiama la funzione per stampare il
                        #parametro. ESP punta alla locazione di memoria
                        # che contiene l'indirizzo del parametro da
                        # considerare. Al posto di tale funzione si
                        # potrebbe inserire il codice che elabora il
                        # dato, invece di stamparlo.

  jmp   ancora          # Ricomincia il ciclo per recuperare gli altri
                        # parametri.

fine_ancora:
```

```
  movl  $1, %eax        # Solito blocco di codice per la chiamata alla
  movl  $0, %ebx        # system call exit per uscire dal programma.
  int   $0x80


.type stampa_parametro, @function # Definizione della funzione
                        # stampa_parametro per la stampa del parametro.
stampa_parametro:
  pushl %ebp            # Salva il contenuto di EBP sullo stack per
                        # poter rendere disponibile il registro EBP.

  movl  %esp, %ebp      # Salva su EBP il valore di ESP per poter
                        # rendere disponibile il registro ESP.

  movl  8(%ebp), %ecx   # Carica ECX con il valore contenuto alla
                        # locazione di memoria il cui indirizzo si
                        # ottiene sommando 8 al valore contenuto in
                        # EBP. Ora ECX contiene il puntatore alla
                        # stringa da stampare. Bisogna sommare 8 poiché
                        # sono state eseguite 2 operazioni sullo stack
                        # che hanno incrementato il valore di ESP di 8:
                        # una CALL e una PUSH!!!


  xorl  %edx, %edx      # azzera EDX


conteggio_caratteri:    # Vengono contati quanti sono i caratteri della
                        # stringa corrispondente al parametro da
                        # stampare.

  movb  (%ecx,%edx), %al  # Carica in AL il carattere puntato da
                        # ECX+EDX.

  testb %al, %al        # Controlla se la stringa è finita (tutte le
                        # stringhe terminano con 0).

  jz    finito_conteggio  # Se la stringa è finita salta alla parte di
                        # codice che esegue la stampa

  incl  %edx            # Se la stringa non è finita incrementa il
                        # contatore dei caratteri.

  jmp   conteggio_caratteri # Prosegui con il conteggio dei caratteri.

finito_conteggio:       # Blocco di codice che usa la funzione di
                        # sistema write
  movl  $4, %eax        # EAX=4
  movl  $1, %ebx        # EBX=1 (cioé video)
                        # ECX punta già alla stringa
                        # EDX contiene già il numero di caratteri da
  int   $0x80           # stampare

  movl  $4, %eax        # Blocco di codice che stampa il carattere di
                        # ritorno a capo
  movl  $1, %ebx        # EBX=1 (cioé video)

  leal  new_line_char, %ecx # ECX contiene l'indirizzo della variabile
                        # che contiene il carattere "a capo"
```

```
movl   $1, %edx         # stampo un solo carattere
int    $0x80

movl   %ebp, %esp       # Riporta ESP al valore che aveva appena
                        # entrati nella funzione.

popl   %ebp             # Riporta EBP al valore che aveva appena
                        # entrati nella funzione.

Ret                     # Ritorna dalla chiamata a funzione. Se prima
                        # della ret non si riporta ESP e EBP al valore
                        # originario, sono guai!!! Ricordare che
                        # l'istruzione CALL salva nello stack il valore
                        # di ritorno per il program counter!!
```

Esercizi

Esercizio 1

Scrivere un programma che recupera e stampa i parametri della riga di comando. Usare l'istruzione POP per estrarre i parametri dallo stack, invece di manipolare il valore dei registri ESP ed EBP manualmente come effettuato nell'esempio precedente.

Esercizio 2

Scrivere un programma Assembly che esegue la divisione intera tra due numeri decimali forniti come parametri della riga di comando e stampa il risultato a video.

Esercizio 3

Raffinare il programma dell'esercizio 2 affinché stampi un messaggio di errore se viene fornito un numero di parametri errato, oppure se i parametri contengono caratteri che non corrispondono a cifre decimali.