[T02] Esercitazione

Istruzioni per l'esercitazione:

- Aprite il form di consegna in un browser e loggatevi con le vostre credenziali `uniroma1`.
- Scaricate e decomprimete sulla scrivania il codice dell'esercitazione. Vi sarà una sotto-directory separata per ciascun esercizio di programmazione. Non modificate in alcun modo i programmi di test `*_main.c`.
- Rinominare la directory chiamandola `cognome.nome`. Sulle postazioni del laboratorio sarà `/home/studente/Desktop/cognome.nome/`.
- È possibile consultare appunti/libri e il materiale didattico online.
- Rispondete alle domande online sul modulo di consegna.
- **Finiti gli esercizi**, e non più tardi della fine della lezione:
  - **zippate la directory di lavoro** in `cognome.nome.zip` (`zip -r cognome.nome.zip cognome.nome/`).
- **Per consegnare**:
  - inserite nel form di consegna come autovalutazione il punteggio di ciascuno dei test forniti (inserite zero se l'esercizio non è stato svolto, non compila, o dà errore di esecuzione).
  - fate **upload** del file `cognome.nome.zip`.
  - **importante**: verificate di aver ricevuto mail di conferma per la sottomissione del form
- Se siete in laboratorio, prima di uscire:
  - **importante**: fate logout dal vostro account Google!
  - eliminate dal desktop la directory creata (`rm -rf cognome.nome`).
  - rimettete a posto eventuali **sedie** prese all'ingresso dell'aula!

Per maggiori informazioni fate riferimento al regolamento delle esercitazioni.

Esercizio 1 (Calcolo di espressioni)

Tradurre nel file `E1/e1.s` la seguente funzione C contenuta in `E1/e1.c`, **senza semplificare l'espressione manualmente**:

```
int f() {
    int x = ((2+3)*(4-2)-(2+3))*3;
    return x + 1;
}
```

Usare il main di prova nella directory di lavoro `E1` compilando con `gcc -m32 e1_main.c e1.s -o e1`

Esercizio 2 (Calcolo di espressioni con un parametro)

Tradurre nel file `E2/e2.s` la seguente funzione C contenuta in `E2/e2.c` che calcola un polinomio a valori interi:

```
int f(int x) {
    return 2*x*x-7*x+1;
}
```

Usare il main di prova nella directory di lavoro `E2` compilando con `gcc -m32 e2_main.c e2.s -o e2`

Esercizio 3 (Calcolo di espressioni con due parametri)

Tradurre nel file `E3/e3.s` la seguente funzione C contenuta in `E3/e3.c` che calcola un polinomio a valori interi:

```
int f(int x, int y) {
    return (x+y)*(x-y);
}
```

Usare il main di prova nella directory di lavoro E3 compilando con `gcc -m32 e3_main.c e3.s -o e3`

Esercizio 4 (Palestra C)

Scrivere nel file `E4/e4.c` la vostra versione personale della funzione della libreria standard libc strcat che appende la stringa `src` alla stringa nel buffer `dest` e restituisce `dest`. Il prototipo della funzione da realizzare è il seguente:

```
char *my_strcat(char *dest, const char *src);
```

Usare il main di prova nella directory di lavoro E4 compilando con `gcc e4_main.c e4.c -o e4`

Esercizio 5 (Debugging)

PREMESSA: Per poter analizzare correttamente un eseguibile 32 bit con Valgrind con la VM BIAR è necessario eseguire i seguenti due comandi:

```
sudo apt update
sudo apt install libc-dbg:i386
```

Se si sta utilizzando WSL su Windows (invece delle VM BIAR), occorre eseguire PRIMA dei due comandi indicati sopra il comando:

```
sudo dpkg --add-architecture i386
```

FINE PREMESSA.

Anche i programmi generati da codice assembly scritto a mano possono essere analizzati con Valgrind e GDB. Si consideri il seguente programma C:

```
int f(int x, int y){
    int r = x * (y - 1) - 4;
    return y + r;
}
```

Uno studente ha scritto la relativa traduzione ASM nel file `E5/e5.s` ma il binario generabile con:

```
$ gcc -g -m32 -o e5 e5_main.c e5.s g.s
```

Non ha un esecuzione corretta:

```
$ ./e5
Segmentation fault (core dumped)
```

Per risolvere questo esercizio, occorre:

1. Analizzare l'esecuzione con Valgrind ed identificare (ove possibile) il punto che porta al segmentation fault

2. Debuggare step by step la funzione `f` con GDB per identificare la causa del segmentation fault e di altri errori logici che non permettono al programma di generare il risultato corretto

Suggerimento #1: Con GDB è possibile stampare il contenuto di un registro utilizzando il comando `print`, ad esempio: `print $ecx` (notare che viene usato `$` prima del nome del registro e non `%` come nella sintassi GAS) o `p $ax`.

Suggerimento #2: L'interfaccia TUI lanciata con il comando `go` mostra una vista (parte alta) su tutti i registri.

NOTA: Ignorare il contenuto di `g.s`.

Domande

Rispondere ai quiz riportati nel form di consegna di consegna.

Soluzioni

Esercizio 1 (Calcolo di espressioni)

Versione C equivalente, creato a partire dall'alberto sintattico dell'espressione (come visto a lezione):

```
int f() {  // codice C equivalente
    int a = 2;
    a = a + 3;
    int c = 4;
    c = c - 2;
    a = a * c;
    c = 2;
    c = c + 3;
    a = a - c;
    a = a * 3;
    a = a + 1;
    return a;
}
```

Versione IA32:

```
.globl f

f:
    movl $2, %eax    #     int a = 2;
    addl $3, %eax    #     a = a + 3;
    movl $4, %ecx    #     int c = 4;
    subl $2, %ecx    #     c = c - 2;
    imull %ecx, %eax #     a = a * c;
    movl $2, %ecx    #     c = 2;
    addl $3, %ecx    #     c = c + 3;
    subl %ecx, %eax  #     a = a - c;
    imull $3, %eax   #     a = a * 3;
    incl %eax        #     a = a + 1;
    ret
```

Esercizio 2 (Calcolo di espressioni con un parametro)

Versione C equivalente, creato a partire dall'alberto sintattico dell'espressione (come visto a lezione):

```
int f(int x) {
    int a = x;
    a = a * a;
    a = a * 2;
    int c = x;
    c = c * 7;
    a = a - c;
    a = a + 1;
    return a;
}
```

Versione IA32:

```
.globl f

f:
    movl 4(%esp), %eax  # int a = x;
    imull %eax, %eax    # a = a * a;
    imull $2, %eax      # a = a * 2;
    movl 4(%esp), %ecx  # int c = x;
    imull $7, %ecx      # c = c * 7;
    subl %ecx, %eax     # a = a - c;
    incl %eax           # a = a + 1;
    ret                 # return a;
```

Esercizio 3 (Calcolo di espressioni con due parametri)

Versione C equivalente, creato a partire dall'alberto sintattico dell'espressione (come visto a lezione):

```
int f(int x, int y) { // x <-> c, y <-> d
    int c = x;
    int d = y;
    int a = c;
    a = a + d;
    c = c - d;
    a = a * c;
    return a;
}
```

Versione IA32:

```
.globl f

f:
    movl 4(%esp), %ecx  #     int c = x;
    movl 8(%esp), %edx  #     int d = y;
    movl %ecx, %eax     #     int a = c;
    addl %edx, %eax     #     a = a + d;
    subl %edx, %ecx     #     c = c - d;
    imull %ecx, %eax    #     a = a * c;
    ret                 #     return a;
```

Esercizio 4 (Palestra C)

```c
#include <string.h>

char *my_strcat(char *dest, const char *src) {
    char* d = dest;
    while (*dest) dest++;
    while (*src) *dest++ = *src++;
    *dest = 0;
    return d;
}
```

Esercizio 5 (Debugging GDB e Valgrind)

Eseguendo il programma compilato (usando le flag `-m32 -g`) otteniamo:

```
[...]
==11860== Invalid read of size 4
==11860==    at 0x10860C: ??? (e5.s:4)
==11860==    by 0x4877FA0: (below main) (libc-start.c:310)
==11860==  Address 0x4 is not stack'd, malloc'd or (recently) free'd
```

L'output è molto prezioso perchè ci indica un accesso invalido alla memoria alla riga 4:

```
    movl 4(%ecx), %ecx                # int c = x;
```

Questa riga effettivamente contiene un errore: il parametro x si trova a `4(%esp)` e non `4(%ecx)`. Correggendo l'errore, se ripetiamo l'analisi con Valgrind, notiamo un errore analogo a riga 5. Dopo aver sistemato entrambi gli errori:

```
    movl 4(%esp), %ecx                # int c = x;
    movl 8(%esp), %edx                # int d = y
```

Se analizziamo nuovamente con Valgrind, sebbene il codice contenga ancora degli errori (logici), non otteniamo nessuna indicazione utile:

```
==13727== Memcheck, a memory error detector
==13727== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13727== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13727== Command: ./e5
==13727==
Valore: atteso=21 ottenuto=0 => KO
Risultato: 0/1
==13727==
==13727== HEAP SUMMARY:
==13727==     in use at exit: 0 bytes in 0 blocks
==13727==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==13727==
==13727== All heap blocks were freed -- no leaks are possible
==13727==
==13727== For counts of detected and suppressed errors, rerun with: -v
==13727== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Notiamo che il test di correttezza effettuato dal `main` non viene superato. Per poter quindi comprendere perchè questo test fallisce, dobbiamo procedere con GDB, analizzando step by step l'esecuzione della funzione `f`.

Una possibile strategia di debug con GDB potrebbe essere:

1) Lanciamo GDB:

```
$ gdb ./e5
```

2) Avvio:

```
(gdb) go
```

3) Settiamo breakpoint su `f` e portiamo avanti esecuzione fino a quel punto:

```
(gdb) break f
(gdb) continue
```

4) Portiamo avanti esecuzione step by step usando il comando `step`.

5) Arrivati alla riga 8, notiamo che l'esecuzione della riga 7 non ha aggiornato ECX ma EDX. Questo sembra non essere consistente con il C equivalente mostrato nel commento della riga. Questo vuol dire che gli operandi dell'istruzione sono invertiti!

6) Sistemiamo l'errore, ricompiliamo e rilanciamo GDB. Arrivati a riga 10 possiamo notare che viene usato EDX, assumendo che questo contenga ancora il valore della variabile `y`. Tuttavia, EDX è stato manipolato a riga 6, quindi occorre rileggere `y` dalla stack usando l'operando `8(%esp)`.

7) Sistemiamo l'errore, ricompiliamo e rilanciamo GDB. Arrivati a riga 14 della funzione `main`, ossia subito dopo l'esecuzione dell'istruzione `ret` di `f`, notiamo con `print r` che il valore di ritorno di `f` non è quello atteso. Questo perchè il codice non rispetta l'ABI: essa prescrive che, prima di eseguire la `ret`, il valore di ritorno deve essere messo nel registro A (EAX se il valore di ritorno è a 32 bit). Quindi aggiungiamo prima della `ret`:

```
movl %ecx, %eax
```

Quindi il codice corretto di `f` risulta essere:

```
.globl f

f:                                  # int f(int x, int y){
    movl 4(%esp), %ecx              # int c = x;
    movl 8(%esp), %edx              # int d = y
    decl %edx                       # d = d - 1
    imull %edx, %ecx                # c = c * d;
    subl $4, %ecx                   # c = c - 4;
    addl 8(%esp), %ecx              # c = c + y;
    movl %ecx, %eax
    ret                             # return c;
```

Domande

Domanda 1) Quale dei seguenti frammenti di codice potrebbe essere scritto in linguaggio macchina?

- `55 23 C3 D3 00 00 00 C3` [corretto, il linguaggio macchina è una sequenza di numeri]
- `movl $2, %eax`
- `while(i<y) i++;`
- nessuno dei precedenti

Domanda 2) Quanti bit servono per rappresentare il numero esadecimale 0xDEADBEEF?

- 32 [corretto, ogni cifra esadecimale corrisponde a 4 bit]

- 8
- 24
- 16

Domanda 3) Il comando `gcc hello.s -o hello` attiva i seguenti stadi della toolchain di compilazione:

- assemblatore e linker [corretto, hello.s va prima assemblato per creare il file oggetto `.o` e poi il tutto va linkato per generare il file eseguibile `hello`]
- assemblatore
- linker
- preprocessore, compilatore e assemblatore

Domanda 4) Il comando `gcc hello.o -o hello` attiva i seguenti stadi della toolchain di compilazione:

- linker [corretto, si parte già da un modulo oggetto che è stato già assemblato, rimane solo da creare l'eseguibile `hello` con il linker]
- preprocessore
- assemblatore e linker`
- preprocessore, compilatore e assemblatore

Domanda 5) Fra i seguenti, qual è il tipo primitivo C con la sizeof minore che consente di rappresentare il numero 256?

- `short` [corretto, 256 è 2^8, quindi sevono > 8 bit per rappresentare il valore]
- `char`
- `unsigned char`
- `nessuno dei precedenti`

Domanda 6) Per quale operatore bit a bit OP si ha che `0x13 OP 0x21 == 0x32?

- `^ (XOR)` [corretto, tradotto in binario si ha 0001 0011 xor 0010 0001 = 0011 0010]
- `~ (NOT)`
- `& (AND)`
- `| (OR)`

Domanda 7) Dati: char s[]="hello"; int a=sizeof(s), b=strlen(s), c=sizeof("hello"); quale delle seguenti affermazioni è vera? Assumere puntatori a 64 bit.

- `a=6, b=5, c=6` [corretto, sia la variabile `s` che il letterale `"hello"` denotano un array di 6 byte che include il terminatore zero della stringa. Invece `strlen` restituisce il numero di caratteri escluso il terminatore]
- `a=6, b=5, c=5`
- `a=5, b=5, c=5`
- `a=6, b=5, c=8`