

## Esercizio 1

Considerare il programma riportato nel file “esercizio1.s” ed elencare quali sono i commenti, le etichette, le istruzioni e le direttive di cui si compone. Effettuare quindi l’assemblaggio e il linking del programma con il comando:

```
aarch64-linux-gnu-gcc -o esercizio1 -static esercizio1.s
```

Eseguire poi il programma con il seguente comando. Cosa si ottiene in output?

```
qemu-aarch64 esercizio1
```

**N.B.:** I comandi su riportati si riferiscono al caso in cui si utilizza l’emulatore suggerito a lezione (già preinstallato nella macchina virtuale fornita).

## Soluzione

Commenti:

```
/*
 * load e print msg1
 * load e print msg2
 * load e print msg1
 */
// return 0
```

Etichette: msg1, msg2, main

Direttive:

```
.global main
.section .rodata
.asciz "*****\n"
.asciz "*Architettura degli Elaboratori*\n"
.text
.size main, (. - main)
```

Istruzioni:

```
stp    x29, x30, [sp, #-16]!
adr     x0, msg1
bl      printf
adr     x0, msg2
bl      printf
adr     x0, msg1
bl      printf
mov     w0, #0
ldp     x29, x30, [sp], #16
ret
```

Eseguendo il programma otteniamo:

```
*****
*Architettura degli Elaboratori*
*****
```

## Esercizio 2

Considerare il seguente programma riportato nel file “esercizio2\_parte1.s” in cui vengono dichiarate alcune variabili.

```
.global main

.data
list: .word 0x00112233, 0x44556677, 0x8899aabb, 0xccddeeff
bool: .byte 0x99
num: .word 0x10203040
```

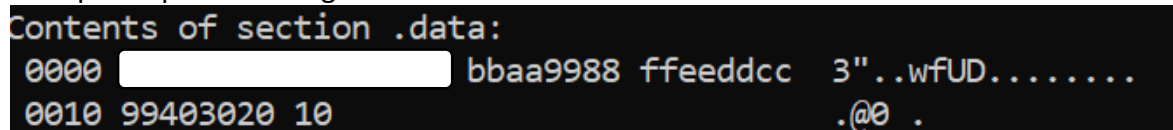
a. Utilizzando i comandi:

```
aarch64-linux-gnu-gcc -c esercizio2_parte1.s
aarch64-linux-gnu-objdump -s esercizio2_parte1.o
```

possiamo ispezionare la memoria.

**N.B.:** I comandi su riportati si riferiscono al caso in cui si utilizza l'emulatore suggerito a lezione (già preinstallato nella macchina virtuale fornita).

L'output è qualcosa del genere:



```
Contents of section .data:
0000 [redacted] bbaa9988 ffeeddcc 3"..wFUD.....
0010 99403020 10 .@0 .
```

Indicare il contenuto della parte coperta in bianco. Specificare inoltre, a quali variabili e a quanti byte corrisponde tale parte.

### Soluzione

La parte in bianco contiene: 33221100 77665544.

33221100 corrisponde al primo elemento dell'array `list`

77665544 corrisponde al secondo elemento dell'array `list`

Si noti che un array in C/C++ è una collezione di elementi disposti in modo consecutivo in memoria.

In totale la parte bianca corrisponde a 8 byte, perché ciascun elemento dell'array `list` è un word e occupa quindi 4 byte.

b. Perché nella sezione `.data` troviamo `bbaa9988` invece che `8899aabb`?

### Soluzione

Perché viene utilizzata la codifica little endian.

c. Quale sarebbe l'output dei comandi precedenti se usassimo la codifica **big endian**?

### Soluzione

```
0000 00112233 44556677 8899aabb ccddeeff
0001 99102030 40
```

d. Verificare come varia il contenuto della sezione `.data` modificando il programma come segue (ovvero aggiungendo `.align 2` tra le variabili `bool` e `num`).

```

.global main

.data
list: .word 0x00112233, 0x44556677, 0x8899aabb, 0xccddeeff
bool: .byte 0x99
.align 2
num: .word 0x10203040

```

Notiamo che ora la memoria risulta allineata. Quanti zero byte sono stati aggiunti?

### Soluzione

Sono stati aggiunti 3 zero byte. Infatti ora l'output è il seguente:

```

Contents of section .data:
0000 33221100 77665544 bb aa9988 ff ee ddcc  3" ..wfUD.....
0010 99000000 40302010                ....@0 .

```

- e. Consideriamo ora il seguente programma, riportato nel file "esercizio2\_parte2.s".

```

.global main

.data
list: .word 0x00112233, 0x44556677, 0x8899aabb, 0xccddeeff
bool: .byte 0x99, 0x88, 0x77, 0x66
.align 2
num: .word 0x10203040

```

Ispezioniamo la memoria occupata dalla sezione `.data` utilizzando i comandi:

```

aarch64-linux-gnu-gcc -c esercizio2_parte2.s
aarch64-linux-gnu-objdump -s esercizio2_parte2.o

```

Notiamo che la memoria risulta allineata e anche togliendo la direttiva `.align` e ispezionando nuovamente la memoria con i precedenti comandi, la memoria risulta comunque allineata. In altre parole, l'uso della direttiva `.align` non ha alcun effetto. Come mai?

### Soluzione

La direttiva `.align` stabilisce automaticamente quanti zero byte aggiungere affinché sia possibile lavorare su un numero di byte che è multiplo di 4. Affinché questo accada ogni variabile deve trovarsi su un indirizzo che sia multiplo di 4. La variabile `num` si trova all'indirizzo 0014: partiamo da 0010 e andiamo 4 byte avanti perché i primi 4 byte sono occupati dall'array `bool` composto da 4 elementi ed ogni elemento è un byte. L'indirizzo 0014 (in esadecimale) in decimale corrisponde a 20 che è multiplo di 4, per cui non c'è bisogno di aggiungere alcuno zero byte affinché la memoria risulti allineata a 4 byte.

### Esercizio 3

Considerare il seguente programma riportato nel file “esercizio3.s”, in cui vengono dichiarate alcune variabili.

```
.global main

.data
a: .byte 0, 1, 2
b: .hword 15
c: .byte 'A'
d: .word 5, 2
e: .quad 9, 4
f: .asciz "Ciao"
```

- a. Arricchire il programma delle direttive `.align` che si ritengono necessarie facendo in modo di limitare il più possibile il numero di byte 0 aggiunti.

#### Soluzione

```
.global main

.data
a: .byte 0, 1, 2
.align 1
b: .hword 15
c: .byte 'A'
.align 2
d: .word 5, 2
.align 3
e: .quad 9, 4
f: .asciz "Ciao"
```

**Nota:** `.align 3` è superfluo in questo caso in quanto la memoria risultava già allineata.

- b. Analizzare l'allineamento della memoria con i comandi:

```
aarch64-linux-gnu-gcc -c esercizio3.s
aarch64-linux-gnu-objdump -s esercizio3.o
```

**N.B.:** I comandi su riportati si riferiscono al caso in cui si utilizza l'emulatore suggerito a lezione (già preinstallato nella macchina virtuale fornita).

Quanti byte sono stati aggiunti? **2**

Quanti ne verrebbero aggiunti se, ogni volta che si passa da un tipo piccolo a uno più grande, si usassero soltanto direttive `.align 3`? **10**

- c. Riordinare le direttive in modo tale che non sia necessario aggiungere alcun `.align`.

#### Soluzione

```
e: .quad 9, 4
d: .word 5, 2
b: .hword 15
a: .byte 0, 1, 2
c: .byte 'A'
f: .asciz "Ciao"
```

**Nota:** l'ordine di a, c ed f è irrilevante.

## Esercizio 4

Considerare il seguente programma riportato nel file “esercizio4.s” in cui vengono dichiarate alcune variabili.

```
.global main

.bss
n: .word 0
m: .word 0

.data
x: .byte 'a'
.skip 2
y: .byte 1
.skip 2
arr: .byte 'A', 'B', 'C'
```

- a. Indicare il contenuto della memoria **senza utilizzare il terminale**.

### Soluzione

```
0000 61000001 00004142 43
```

- b. Verificare il punto precedente ed analizzare l’allineamento della memoria con i comandi:

```
aarch64-linux-gnu-gcc -c esercizio4.s
aarch64-linux-gnu-objdump -s esercizio4.o
```

**N.B.:** I comandi su riportati si riferiscono al caso in cui si utilizza l’emulatore suggerito a lezione (già preinstallato nella macchina virtuale fornita).

- c. Utilizzando la sola direttiva `.skip`, modificare il programma per ottenere:

```
0000 00610000 01000000 414243
```

Determinare ora la dimensione del file `esercizio4.o` con il comando:

```
ls -l esercizio4.o
```

Si noti che `ls -l` dà in output il cosiddetto “long list format” ovvero un formato di output più dettagliato rispetto al semplice `ls`. Tale output contiene diverse informazioni riportate in 7 colonne: la prima colonna indica tipo di file e permessi di lettura/scrittura/esecuzione, la seconda il numero di collegamenti, la terza il nome del proprietario, la quarta il nome del proprietario del gruppo, la quinta la dimensione del file in byte (è questa l’informazione che ci interessa), la sesta data e ora dell’ultima modifica e la settima il nome del file (<https://linuxhint.com/what-does-ls-l-command-do-in-linux/>).

### Soluzione

```
.global main

.bss
n: .word 0
m: .word 0
```

```

.data
.skip 1
x: .byte 'a'
.skip 2
y: .byte 1
.skip 3
arr: .byte 'A', 'B', 'C'

```

- f. A partire dalla soluzione del punto precedente, sostituire il primo `.skip` nella sezione `.data` con:

```
.skip 1024
```

Ripetere i comandi riportati nei punti b e c per assemblare e determinare nuovamente la dimensione del file `esercizio4.o`.

- g. Modificare ora il programma in questo modo:

```

.global main

.bss
n: .word 0
m: .word 0

.data
.skip 1*1024
x: .byte 'a'
.skip 2*1024
y: .byte 1
.skip 3*1024
arr: .byte 'A', 'B', 'C'

```

Ripetere i comandi riportati nei punti b e c per assemblare e determinare nuovamente la dimensione del file `esercizio4.o`.

Come cambia la dimensione del file `esercizio4.o`?

- h. Arricchire ora la sezione `.bss` come riportato di seguito:

```

.bss
n: .word 0
.skip 1
m: .word 0

```

- 1) Ripetere i comandi riportati nei punti b e c per assemblare e determinare nuovamente la dimensione del file `esercizio4.o`.
- 2) Ritorniamo nella sezione `.bss` e modifichiamola in modo da fare uno skip di 2048 (ovvero come segue):

```

.bss
n: .word 0
.skip 2048
m: .word 0

```

Ripetere i comandi riportati nei punti b e c per assemblare e determinare nuovamente la dimensione del file `esercizio4.o`.

Come cambia la dimensione del file `esercizio4.o`?

### Soluzione

Il piccolo incremento della dimensione in 1) è dovuto all'aggiunta della direttiva `.skip 1` che deve essere memorizzata all'interno del file `esercizio4.o`

Le variabili dichiarate nella sezione `.bss` vengono inizializzate a 0 solo durante l'esecuzione, ed il file `esercizio4.o` deve memorizzare solo la dimensione delle variabili in essa dichiarate. Ecco perché la dimensione del file oggetto non cambia tra 1) e 2).

Quindi, in `.bss` il numero di byte argomento della direttiva `.skip` non ha effetto sul numero di byte salvati nel file oggetto.

### Esercizio 5

La sequenza di bit 10010110 in complemento a due, a quale intero in base 10 corrisponde?

Per svolgere l'esercizio, effettuare prima il complemento a 1 della sequenza di bit e poi sommare 1.

A tale scopo, ricordiamo che per sommare due numeri binari bisogna incolonnarli e che 1 corrisponde a 00000001. Inoltre, ricordiamo che:

$0+0=0$

$1+0=1$

$0+1=1$

$1+1=0$  con riporto di 1 alla colonna immediatamente a sinistra

### Soluzione

1	0	0	1	0	1	1	0	Sequenza di bit
0	1	1	0	1	0	0	1 + 1 =	Complemento a 1 Sommo 1
0	1	1	0	1	0	1	0	
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
	64	32		8		2		$64+32+8+2 = 106$

Dal momento che il bit più a sinistra del byte originale è 1, si tratta del numero negativo: **-106**.

Come verifica del procedimento, possiamo effettuare la somma della rappresentazione in complemento a 2 di -106 e +106. Ci aspettiamo che esca 0, esattamente come se sommassimo -106 e +106. Infatti, otteniamo:

1	0	0	1	0	1	1	0 +	-106
0	1	1	0	1	0	1	0 =	+106
(1)0	0	0	0	0	0	0	0	Il bit in eccesso (ovvero, l'overflow) lo ignoriamo perché stiamo lavorando con 8 bit.



## Esercizio 6

La sequenza di bit 1101 0101 in complemento a due, a quale numero intero in base 10 corrisponde? Questa volta, per svolgere l'esercizio, sottrarre prima 1 alla sequenza di bit e poi effettuare il complemento a 1.

A tale scopo, ricordiamo che per sottrarre due numeri binari bisogna incolonnarli e che 1 corrisponde a 00000001. Inoltre, ricordiamo che:

0-0=0

1-0=1

0-1=1 prestando 1 dalla colonna a sinistra

1-1=0

## Soluzione

1	1	0	1	0	1	0	1 -	Sequenza originale
							1 =	Sottraggo 1
1	1	0	1	0	1	0	0	
0	0	1	0	1	0	1	1	Complemento a 1
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
		32		8		2	1	$32+8+2+1 = 43$

Notiamo che se invece prima facciamo il complemento a 1 e poi sommiamo 1 otteniamo lo stesso risultato.

1	1	0	1	0	1	0	1	Sequenza originale
0	0	1	0	1	0	1	0 +	Complemento a uno
							1 =	Sommo 1
0	0	1	0	1	0	1	1	
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
		32		8		2	1	$32+8+2+1 = 43$

Dal momento che il bit più a sinistra nel byte originale è 1, si tratta di un numero negativo, quindi otteniamo: **-43**.

## Esercizio 7

Avendo **8 bit a disposizione** e dato il numero intero -35 in base 10, calcolare la rappresentazione in complemento a 2.

## Soluzione

Divisione	Quoziente	Resto
35:2	17	1 (meno significativo)
17:2	8	1
8:2	4	0
4:2	2	0
2:2	1	0
1:2	0	1 (più significativo)

Il numero 35 con 8 bit corrisponde a **00 100011** (primi due bit aggiunti per arrivare a 8 bit).

0	0	1	0	0	0	1	1	
1	1	0	1	1	1	0	0 + 1 =	Complemento a uno Sommo 1
1	1	0	1	1	1	0	1	

Quindi, -35 in complemento a 2 è **11011101**.

### Esercizio 8

Si converta in complemento a due, utilizzando il minor numero di bit possibile, il numero -38 in base 10.

### Soluzione

Per rappresentare -38 abbiamo bisogno di 7 bit. Con 7 bit possiamo rappresentare in complemento a 2, 128 numeri, ovvero i numeri interi nell'intervallo [-64,+63].

Divisione	Quoziente	Resto
38:2	19	0 (meno significativo) ↑
19:2	9	1
9:2	4	1
4:2	2	0
2:2	1	0
1:2	0	1 (più significativo)

Si ottiene **100 110**. Lo 0 come bit iniziale si aggiunge per ottenere la rappresentazione in complemento a 2 di +38 con 7 bit -> **0100110**

Dunque, per ottenere -38:

0	1	0	0	1	1	0	Byte originale
1	0	1	1	0	0	1 + 1 =	Complemento a uno Sommo 1
1	0	1	1	0	1	0	

Quindi in complemento a due, -38 corrisponde a **1011010**

### Esercizio 9

Convertire la stringa costituita dal proprio nome seguito da uno spazio, dal proprio cognome e dal carattere NUL nel formato ASCII binario, esadecimale e decimale.

**Suggerimento:** per svolgere questo esercizio è sufficiente fare riferimento alla tabella sulle slide del corso sullo standard ASCII (vedi "caratteri stampabili").

### Soluzione

Come esempio, supponendo di dover convertire "Maria Rossi" otteniamo:

Carattere	In binario	In esadecimale	In decimale
M	01001101	4D	77
a	01100001	61	97

r	01110010	72	114
i	01101001	69	105
a	01100001	61	97
	00100000	20	32
R	01010010	52	82
o	01101111	6F	111
s	01110011	73	115
s	01110011	73	115
i	01101001	69	105
NUL	00000000	00	0

### Esercizio 10

Convertire la seguente sequenza di bit in UTF-8 e determinare la frase da essa codificata:

01001111 01100111 01100111 01101001 00100000 01101110 01101111 01101110 00100000  
11000011 10101000 00100000 01101101 01100001 01110010 01110100 01100101 01100100  
11000011 10101100 00100001 00000000

**Suggerimento:** aiutatevi con questa tabella di conversione <https://www.utf8-chartable.de/unicode-utf8-table.pl?utf8=bin>

### Soluzione

La frase codificata è “Oggi non è martedì!” seguita dal carattere NUL. Si noti che i caratteri accentati occupano più di un byte. Infatti:

01001111	<b>O</b>
01100111	<b>g</b>
01100111	<b>g</b>
01101001	<b>i</b>
00100000	
01101110	<b>n</b>
01101111	<b>o</b>
01101110	<b>n</b>
00100000	
11000011 10101000	<b>è</b>
00100000	
01101101	<b>m</b>
01100001	<b>a</b>
01110010	<b>r</b>
01110100	<b>t</b>
01100101	<b>e</b>
01100100	<b>d</b>
11000011 10101100	<b>ì</b>
10101100	<b>!</b>
00000000	<b>NUL</b>