Esercizio 1

Dopo aver scaricato il file zip relativo al laboratorio odierno, estraiamolo sulla Scrivania. Apriamo la cartella estratta "lab03", al cui interno troviamo i file: "esercizio1.s", "esercizio2.s", "esercizio3.s", "esercizio4.s", "esercizio5.s". Facciamo click con il tasto destro del mouse su tale cartella e selezioniamo "Apri un terminale qui".

Considerare ora il seguente programma riportato nel file "esercizio1.s" in cui vengono dichiarati un array di short e una stringa terminata dal carattere NULL.

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

```
.global main

.data
x: .short 0, 1, 1, 2, 3, 5, 8, 13, 21
s: .asciz "Calcola la lunghezza!"
```

a. Determinare la lunghezza dell'array e della stringa salvandola, rispettivamente, nelle variabili `x_size` e `s_size` utilizzando opportunamente la direttiva `.equiv`

   Ritorniamo sul terminale. Verifichiamo il risultato ed analizziamo il contenuto della memoria con i comandi:

   ```
   aarch64-linux-gnu-gcc -c esercizio1.s
   aarch64-linux-gnu-objdump -t esercizio1.o
   ```

b. Determinare nuovamente la lunghezza dell'array e della stringa salvandola, rispettivamente, nelle variabili `x_size` e `s_size` ma **utilizzando questa volta la direttiva `.equ` al posto di `.equiv`**

   Verificare il risultato ed analizzare nuovamente il contenuto della memoria con gli stessi comandi del punto a.

c. Cambiare il nome delle variabili in modo che entrambe si chiamino `size`, ovvero rinominando come segue:

   `x_size` diventa `size`

   `s_size` diventa `size`

   Verificare il risultato ed analizzare nuovamente il contenuto della memoria con gli stessi comandi del punto a.

d. Mantenendo lo stesso nome per le due variabili, sostituire la direttiva `.equ` con `.equiv`

   Verificare il risultato ed analizzare nuovamente il contenuto della memoria con gli stessi comandi del punto a.

Esercizio 2

Come prima, apriamo un terminale nella cartella "lab03", facendo click con il tasto destro del mouse su tale cartella e selezionando "Apri un terminale qui". Se abbiamo un terminale già aperto in lab03 possiamo continuare ad utilizzarlo senza aprirne uno nuovo. Considerare ora il programma riportato nel file "esercizio2.s".

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

a. Effettuare quindi l'assemblaggio e il linking del programma con il comando:
`aarch64-linux-gnu-gcc -o esercizio2 -static esercizio2.s`
Eseguire poi il programma con il seguente comando. Cosa si ottiene in output?
`qemu-aarch64 esercizio2`
Indicare quindi cosa si ottiene in output**.**

b. Effettuiamo ora il disassemblaggio del file eseguibile `esercizio2` che abbiamo ottenuto, ovvero "traduciamo" questo file di nuovo in assembly, in modo da verificare l'effetto della direttiva `.ifndef`. Per fare ciò, ritornando sul terminare dobbiamo effettuare il seguente comando:
`aarch64-linux-gnu-objdump -d esercizio2 > esercizio2_dis.s`
Noteremo che all'interno della cartella "lab03" troveremo un nuovo file dal nome "esercizio2_dis.s". Infatti, il comando precedente salva la "traduzione" del file eseguibile "esercizio2" all'interno del file "esercizio2_dis.s". Apriamo ora Visual Studio Code, nel menu "File" scegliamo "Apri Cartella" e apriamo la cartella "lab03". Selezioniamo e apriamo il file "esercizio2_dis.s". Ora clicchiamo sul menù "Modifica" e poi su "Trova" (in alternativa, premiamo insieme Ctrl+F), in alto a destra ci verrà mostrata una casella di testo in cui possiamo inserire del testo da cercare nel file. In questo caso, vogliamo cercare `<main>` in modo tale da visualizzare come viene "tradotta" la funzione main e cosa contiene. Otterremo qualcosa di simile:

```
0000000000400fc <main>:
  4003fc: a9bf7bfd  stp x29, x30, [sp, #-16]!
  400400: 102752c0  adr x0, 44ee58 <fmt>
  400404: 52800001  mov w1, #0x0                    // #0
  400408: 940018e6  bl  4067a0 <_IO_printf>
  40040c: 52800000  mov w0, #0x0                    // #0
  400410: a8c17bfd  ldp x29, x30, [sp], #16
  400414: d65f03c0  ret
```

Notiamo che nel main non sono più presenti `.ifndef`, `.else` ed `.endif`. Inoltre, l'invocazione della macro è stata sostituita dalle istruzioni in essa contenute. A cosa è dovuto questo comportamento?

c. Cosa cambierebbe se `.equ CONSTANT, 1` fosse sostituito con
`.equ CONSTANT, 10` ?

d. Commentare `.equ CONSTANT, 1` anteponendo // quindi assemblare ed eseguire nuovamente il programma. Disassemblate ora nuovamente il programma come spiegato al punto b e analizzare nuovamente il main. Il programma si comporta in modo diverso: a cosa è dovuto questo comportamento?

Esercizio 3

Come prima, apriamo un terminale nella cartella "lab03", facendo click con il tasto destro del mouse su tale cartella e selezionando "Apri un terminale qui". Se abbiamo un terminale già aperto in lab03 possiamo continuare ad utilizzarlo senza aprirne uno nuovo. Considerare ora il programma riportato nel file "esercizio3.s".

a. Indicare la parte di programma che corrisponde alla dichiarazione di una macro.
b. Indicare la/le invocazioni della macro presenti nel programma.
c. Assemblare ed eseguire il programma con i seguenti comandi e verificare cosa si ottiene in output:

```
aarch64-linux-gnu-gcc -o esercizio3 -static esercizio3.s
qemu-aarch64 esercizio3
```

d. Come nell'esercizio 2, effettuiamo ora il disassemblaggio del file eseguibile esercizio3 che abbiamo ottenuto. Per fare ciò, ritornando sul terminare e assicurandoci di essere nella cartella "lab03" dobbiamo effettuare il seguente comando:
`aarch64-linux-gnu-objdump -d esercizio3 > esercizio3_dis.s`
Noteremo che all'interno della cartella "lab03" troveremo un nuovo file dal nome "esercizio3_dis.s". Ritorniamo ora su Visual Studio Code e cerchiamo `<main>` in modo tale da visualizzare come viene "tradotta" la funzione main e cosa contiene.
Notiamo che le invocazioni della macro sono state sostituite dalle istruzioni in essa contenute. A cosa è dovuto questo comportamento?
e. Modificare il programma in modo tale da stampare soltanto il numero 15. Dopo aver assemblato ed eseguito il programma come descritto nel punto c, verificare che venga stampato solo 15. Inoltre, disassemblare nuovamente il file binario e ispezionare il main come descritto nel punto d. Come è cambiato ora il contenuto del main?

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

Esercizio 4

Come prima, apriamo un terminale nella cartella "lab03", facendo click con il tasto destro del mouse su tale cartella e selezionando "Apri un terminale qui". Se abbiamo un terminale già aperto in lab03 possiamo continuare ad utilizzarlo senza aprirne uno nuovo. Considerare ora il programma riportato nel file "esercizio4.s".
Tale programma dichiara una variabile globale *n* di dimensione pari a un byte e avente come valore 0b11001000, ovvero 11001000: infatti, lo 0b iniziale indica proprio che il valore è riportato in rappresentazione binaria in complemento a due. Nella funzione main, *n* viene incrementata di una unità invocando la macro add_to_n con parametro 1, e successivamente, tramite la macro print, *n* viene stampata.

a. Senza eseguire il programma, indicare quanto vale *n* nel sistema decimale e nel sistema binario, prima e dopo l'esecuzione della macro nel main.
b. Indicare quanto valgono i flag NZCV di Pstate, quindi assemblare ed eseguire il programma per verificare se la risposta data è corretta.
c. Indicare quanto vale *n* nel sistema decimale e nel sistema binario dopo l'esecuzione della macro nel main, se al posto di `add_to_n 1` si avesse `add_to_n -1`.
d. Indicare quanto valgono i flag NZCV di Pstate anche in questo ultimo caso, quindi assemblare ed eseguire il programma per verificare se la risposta data è corretta.

e. Indicare nel sistema decimale e nel sistema binario in complemento a due, il più piccolo numero che se aggiunto ad *n* al posto di 1 nel main, farebbe ottenere un risultato che richiederebbe più di un byte.

Esercizio 5

Come prima, apriamo un terminale nella cartella "lab03", facendo click con il tasto destro del mouse su tale cartella e selezionando "Apri un terminale qui". Se abbiamo un terminale già aperto in lab03 possiamo continuare ad utilizzarlo senza aprirne uno nuovo. Considerare ora il programma riportato nel file "esercizio5.s".
Tale programma, in base al valore della costante LOG_LEVEL, consente di abilitare la stampa di alcuni messaggi utili al programmatore, che possono essere di tipo: debug (ovvero, messaggi di debugging), info (ovvero, messaggi generici informativi) e warning (ovvero, messaggi "di pericolo"). È tipico dei linguaggi di programmazione consentire diversi livelli di log, allo scopo di aiutare i programmatori a testare più agevolmente il loro codice. Ad esempio, in Python 3 è possibile abilitare i seguenti livelli di log: https://docs.python.org/3/library/logging.html#levels. A più basso livello, la realizzazione di tale meccanismo avviene tramite assembly condizionale.

a. Assemblare ed eseguire il programma con i due comandi riportati di seguito per verificare quali tipologie di messaggi sono abilitati:
```
aarch64-linux-gnu-gcc -o esercizio5 -static esercizio5.s
qemu-aarch64 esercizio5
```

b. Cambiare il valore di LOG_LEVEL affinché vengano abilitati solo debug e info. Quindi, assemblare ed eseguire nuovamente il programma.

c. Cambiare il valore di LOG_LEVEL affinché vengano abilitate tutte le stampe (debug, info e warning). Quindi, assemblare ed eseguire nuovamente il programma.