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`

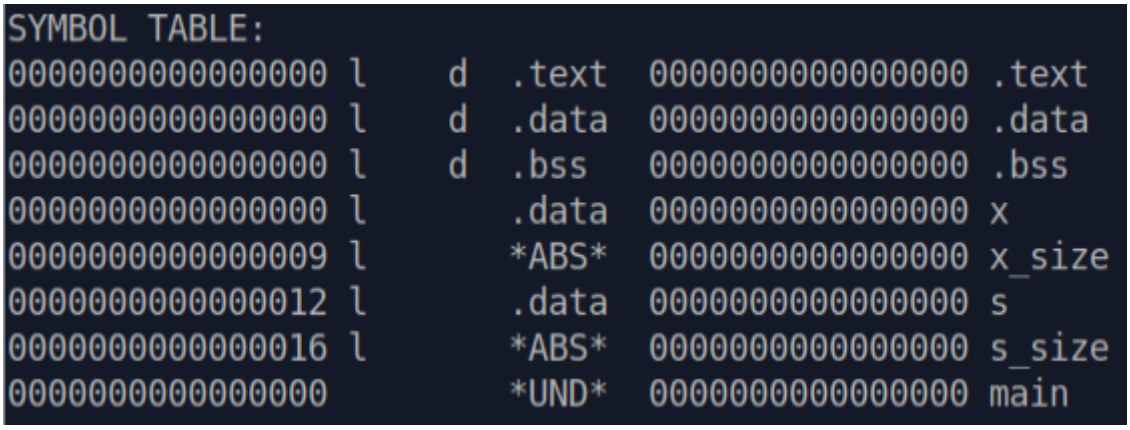
   Ritorniamo sul terminale. Verifichiamo il risultato ed analizziamo il contenuto della memoria con i comandi:

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

   Soluzione

   ```
   .global main
   .data
   x: .short 0, 1, 1, 2, 3, 5, 8, 13, 21
   // 2 indica il numero di byte per elemento (short corrisponde a 2 byte)
   .equiv  x_size, ( . - x )/2
   // 1 indica il numero di byte (essendo 1 si potrebbe omettere)
   s: .asciz "Calcola la lunghezza!"
   .equiv  s_size, ( . - s )/1
   ```

   Per calcolare `x_size`, dividiamo per 2, perché `x` è un array di short e ogni elemento occupa 2 byte (ovvero 16 bit). Per calcolare `s_size`, dividiamo per 1, perché `s` è una stringa (ovvero un array di caratteri terminato da NULL) e ogni carattere (NULL incluso) occupa un solo byte (8 bit).
   Osserviamo che `x_size` vale 9 e `s_size` vale 22. Infatti dopo l'esecuzione del comando objump vedremo in output una schermata simile a quella che segue, in cui i valori sono riportati in esadecimale (9 in esadecimale corrisponde a 9, 16 in esadecimale corrisponde a 22).



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.

   Soluzione

   Cambiando semplicemente la direttiva da `.equiv` a `.equi` il risultato resta invariato.

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.

   Soluzione

   La variabile `size` salverà solo l'ultimo valore calcolato. La lunghezza del primo array `x` viene sovrascritta dalla lunghezza dell'array `s` calcolata per ultima.

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.

   Soluzione

   Si ottiene l'errore: **symbol `size' is already defined**. Infatti, con la direttiva .equiv l'assemblatore segnala errore se si prova a ridefinire un simbolo già definito (concettualmente, è come se stessimo ridichiarando una variabile già dichiarata, operazione che in C/C++ non è consentita).

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**.**

Soluzione

Il programma stampa 0.

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:

```
00000000004003fc <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?

Soluzione

Questo comportamento è dovuto al fatto che tali direttive sono analizzate e valutate direttamente dall'assemblatore, ovvero quando effettuiamo il comando:
`aarch64-linux-gnu-gcc -o esercizio2 -static esercizio2.s`
in base alla definizione di `CONSTANT` nel file binario viene incluso il ramo `.ifned` (che sta per *if not defined*) oppure il ramo `.else`. In questo caso `CONSTANT` è definita e pertanto viene incluso il ramo `.else` che al suo interno invoca la macro `print` per stampare 0. Dal momento che una macro non è altro che un alias per un blocco di codice, tale blocco di codice viene ricopiato nel main.

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

Soluzione

Nulla, il programma continuerebbe ad avere lo stesso comportamento perché `CONSTANT` sarebbe comunque definita.

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?

Soluzione

Tale comportamento è dovuto alla direttiva `.ifndef` perché commentando la definizione della costante `CONSTANT` viene incluso il ramo `.ifndef` perché `CONSTANT` non sarebbe più definita. Infatti, nel main ottenuto dal dissassemblaggio otterremo qualcosa del genere:

```
00000000004003fc <main>:
  4003fc: a9bf7bfd  stp x29, x30, [sp, #-16]!
  400400: 102752c0  adr x0, 44ee58 <fmt>
  400404: 52800021  mov w1, #0x1                    // #1
  400408: 940018e6  bl  4067a0 <_IO_printf>
  40040c: 52800000  mov w0, #0x0                    // #0
  400410: a8c17bfd  ldp x29, x30, [sp], #16
  400414: d65f03c0  ret
```

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.

Soluzione

```
.macro print n
    adr x0, fmt
    mov w1, \n
    bl printf
.endm
```

b. Indicare la/le invocazioni della macro presenti nel programma.

Soluzione

```
print 1
print 2
print 3
print 4
print 5
```

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
```

Soluzione

Il programma stampa su singole linee: 1, 2, 3, 4 e 5.

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?

Soluzione

Una macro non è altro che un alias per un blocco di codice e in fase di assemblaggio tale codice viene ricopiato il codice e sostituito appropriatamente a ciascuna invocazione.

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?

Soluzione

È sufficiente utilizzare `print 15` e rimuovere tutte le altre invocazioni della macro. Nel main vedremo solo il codice della macro necessario per stampare 15.

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.

   Prima: $1100\ 1000_2 = 200_{10}$
   Dopo: $1100\ 1001_2 = 201_{10}$

b. Indicare quanto valgono i flag NZCV di Pstate, quindi assemblare ed eseguire il programma per verificare se la risposta data è corretta.

   N=0 Z=0 C=0 V=0

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`.

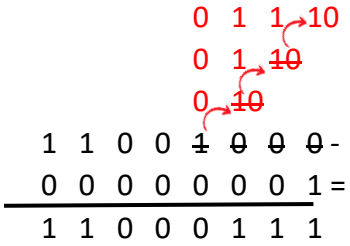
   $1100\ 0111_2 = 199_{10}$

d. Indicare quanto valgono i flag NZCV di PSTATE anche in questo caso, quindi assemblare ed eseguire il programma per verificare se la risposta data è corretta.

Soluzione

N=0 Z=0 C=1 V=0

C=1 poiché l'operazione effettuata è la seguente e come possiamo notare ha richiesto dei prestiti:



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.

Soluzione

Se al posto di 1 aggiungessimo 56_{10} ovvero 0011 1000_2 otterremmo il numero 256_{10} che non è rappresentabile con 8 bit, infatti 256_{10} corrisponde a 1 0000 0000_2. Si noti inoltre che aggiungendo 56, *n* diventa uguale a 0 perché il risultato della somma viene troncato a 8 bit. Inoltre, i flag NZCV valgono tutti 0:

- N=0 perché il risultato ottenuto non è un numero negativo
- Z=0 perché il risultato ottenuto non è realmente zero ma 1 0000 0000_2 troncato a 8 bit quando viene eseguita l'istruzione `strb` alla fine della macro `add_to_n`
- C=0 perché nel caso di addizione C diventa 1 solo se il risultato ottenuto eccede 32 bit (ovvero, la dimensione del registro w9 di cui la macro fa uso) e non è questo il caso, poiché 1 0000 0000_2 è memorizzabile in 9 bit.
- per lo stesso motivo, V=0 perché non si è verificato un overflow (il risultato dell'operazione è memorizzabile in meno di 32 bit).

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.

   Soluzione

   È sufficiente cambiare il LOG_LEVEL con un valore qualsiasi maggiore o uguale a 20 e minore 29.

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

   Soluzione

   È sufficiente cambiare il LOG_LEVEL con un valore qualsiasi maggiore o uguale a 30.