Esercizio 1

Considerare il programma riportato nel file "esercizio1.s". Nella sezione .data è stato dichiarato l'array ***n***, in cui ogni elemento è di tipo word e occupa dunque 4 byte. Sono inoltre presenti due macro che permettono di stampare a video il contenuto del registro R1 in formato esadecimale: in particolare, nel programma, quando si fa uso del registro W1 si utilizza la macro print_32 mentre, quando si fa uso del registro X1 si utilizza la macro print_64. Nel main il programma riporta degli esempi, ciascuno generalmente composto da:

- una istruzione ldr che carica in R1 uno o più elementi dell'array ***n***;
- una invocazione delle macro suddette per visualizzare il contenuto del registro R1.

Considerando un esempio per volta:

a) indicare se l'istruzione ldr effettuata è corretta;
b) indicare il tipo di indirizzamento;
c) spiegare il significato dell'istruzione ldr;
d) decommentare soltanto le istruzioni dell'esempio in esame e utilizzare il seguente comando per assemblare il programma, indicare quindi eventuali errori segnalati dall'assemblatore dandone una spiegazione:
   `aarch64-linux-gnu-gcc -o esercizio1 -static esercizio1.s`
e) utilizzare infine il seguente comando per eseguire il programma e verificare quali dati sono stati caricati nel registro X1 o W1:
   `qemu-aarch64 esercizio1`

Soluzione

```
//Esempio 1
ldr w1, [x2]
print_32
```

Istruzione corretta. Register Address. Copia in W1 4 byte presi dall'indirizzo di memoria contenuto nel registro X2. Dal momento che X2 contiene l'indirizzo dell'array ***n***, in W1 che è un registro a 32 bit (ovvero, 4 byte), viene caricato il primo elemento dell'array ***n***.

```
//Esempio 2
ldr x1, [x2]
print_64
```

Istruzione corretta. Register Address. Copia in X1 8 byte presi dall'indirizzo di memoria contenuto nel registro X2. Dal momento che X2 contiene l'indirizzo dell'array ***n***, in X1 che è un registro a 64 bit (ovvero, 8 byte), vengono caricati 8 byte a partire da tale indirizzo e dunque, i primi due elementi dell'array ***n***.

```
//Esempio 3
ldr w1, [x2, #8]
print_32
```

Istruzione corretta. Signed Immediate Offset. Il registro X2 che contiene l'indirizzo dell'array ***n*** viene incrementato di 8 byte e dunque si ottiene l'indirizzo del terzo elemento in ***n***. Di conseguenza, in W1 vengono copiati 4 byte presi dall'indirizzo di memoria ottenuto da tale

incremento, ovvero il terzo elemento dell'array ***n***. Si noti che l'esecuzione dell'istruzione ldr non modifica il contenuto del registro X2, il quale rimane invariato.

```
//Esempio 4
mov x3, #4
ldr w1, [x2, x3]
print_32
```

Istruzione corretta. Register Offset. Il registro X2 che contiene l'indirizzo dell'array ***n*** viene incrementato del valore contenuto nel registro X3, ovvero 4 e si ottiene l'indirizzo del secondo elemento in ***n***. Di conseguenza, in W1 vengono copiati 4 byte presi dall'indirizzo di memoria ottenuto da tale incremento, ovvero il secondo elemento dell'array ***n***.

```
//Esempio 5
mov x3, #1 // Il registro X3 contiene il valore 1
ldr w1, [x2, x3, lsl #2]
print_32
```

Istruzione corretta. Register Offset con shift a sinistra. Il registro X2 viene incrementato del valore contenuto nel registro X3 dopo che ne è stato fatto lo shift a sinistra di 2 posizioni. Il registro X3 contiene inizialmente il valore 1 ed effettuando lo shift a sinistra è come se il suo contenuto venisse moltiplicato per 4. Infatti data una sequenza di bit, fare lo shift a sinistra di una posizione significa moltiplicare una volta per 2, e dunque fare lo shift di due posizioni a sinistra significa moltiplicare due volte per due, ovvero moltiplicare per 4. Dunque X2 viene incrementato di 4 e si ottiene l'indirizzo del secondo elemento in ***n***. Di conseguenza, in W1 vengono copiati 4 byte presi dall'indirizzo di memoria ottenuto da tale incremento, ovvero il secondo elemento dell'array ***n***.

```
//Esempio 6
ldr w1, [x2, #0xffff]
print_32
```

Istruzione non corretta. Unsigned Immediate Offset. L'istruzione non è valida perché l'offset deve essere compreso tra 0 e 7ff9 (in esadecimale). Si ottiene infatti l'errore:

```
Error: immediate offset out of range
```

```
//Esempio 7
ldr w1, [x2, #12]!
print_32
```

Istruzione corretta. Pre-indexed immediate Offset. Il registro X2 che inizialmente contiene l'indirizzo dell'array ***n*** viene incrementato di 12 byte e dunque X2 contiene l'indirizzo del quarto elemento in ***n***. Di conseguenza, in W1 vengono copiati 4 byte presi dall'indirizzo di memoria contenuto nel registro X2, ovvero il quarto elemento dell'array ***n***. Si noti che, in questo caso, l'esecuzione dell'istruzione ldr modifica il contenuto del registro X2.

```
//Esempio 8
ldr x1, [x2], #4
print_64
```

Istruzione corretta. Post-indexed immediate Offset. In X1 vengono copiati 8 byte presi dall'indirizzo di memoria contenuto nel registro X2. Dal momento che inizialmente X2 contiene l'indirizzo dell'array ***n***, in X1 vengono caricati il primo e il secondo elemento dell'array. Successivamente, il registro X2 viene incrementato di 4. Infatti, se dopo effettuassimo le seguenti istruzioni:

```
ldr x1, [x2]
print_64
```

vedremmo che in X1 sarebbero caricati il secondo e il terzo elemento dell'array. Dunque anche in questo caso, l'esecuzione dell'istruzione ldr modifica il contenuto del registro X2.

Esercizio 2

Considerare il programma riportato nel file "esercizio2.s". Nella sezione .data è stato dichiarato l'array ***n***, in cui ogni elemento è di tipo word e occupa dunque 4 byte. Sono inoltre presenti due macro che permettono di stampare a video un singolo elemento dell'array oppure tutti i suoi elementi in formato esadecimale. Nel main il programma riporta degli esempi, ciascuno generalmente composto da:

- una istruzione str che sostituisce uno o più elementi dell'array ***n*** con il contenuto del registro X3 che contiene il valore esadecimale ffff ffff ffff ffff;
- una invocazione della macro che stampa tutti gli elementi dell'array.

Considerando un esempio per volta:

a) indicare se l'istruzione str effettuata è corretta;
b) indicare il tipo di indirizzamento;
c) spiegare il significato dell'istruzione str;
d) decommentare soltanto le istruzioni dell'esempio in esame e utilizzare il seguente comando per assemblare il programma, indicare quindi eventuali errori segnalati dall'assemblatore dandone una spiegazione:
   `aarch64-linux-gnu-gcc -o esercizio2 -static esercizio2.s`
e) utilizzare infine il seguente comando per eseguire il programma e verificare come cambia l'array ***n***:
   `qemu-aarch64 esercizio2`

Soluzione

```
//Esempio 1
str x3, [x2]
print_all
```

Istruzione corretta. Register Address. Dal momento che X2 contiene l'indirizzo dell'array ***n*** il contenuto del registro X3 sostituisce i primi due elementi dell'array: ciò è dovuto al fatto che stiamo utilizzando X3 e dunque un registro a 64 bit.

```
//Esempio 2
str w3, [x2]
print_all
```

Istruzione corretta. Register Address. Dal momento che X2 contiene l'indirizzo dell'array ***n*** il contenuto del registro W3 sostituisce questa volta soltanto il primo elemento dell'array: ciò è dovuto al fatto che stiamo utilizzando W3 e dunque un registro a 32 bit.

```
//Esempio 3
str w3, [x2, #4]
print_all
```

Istruzione corretta. Signed Immediate Offset. Il registro X2 viene incrementato X2 di 4 e dunque il contenuto del registro W3 sostituisce il secondo elemento dell'array. Infatti, inizialmente X2 contiene l'indirizzo dell'array ***n*** e viene poi incrementato di 4 arrivando così al secondo elemento. Si noti che dopo l'esecuzione dell'istruzione str, il contenuto di X2 rimane invariato.

```
//Esempio 4
mov x4, #8 // Il registro X4 contiene il valore 8
str x3, [x2, x4]
print_all
```

Istruzione corretta. Register Offset. Il registro X2 viene incrementato X2 di 8, ovvero del valore contenuto nel registro X4. Dunque il contenuto del registro X3 sostituisce il terzo e il quarto elemento dell'array. Infatti, inizialmente X2 contiene l'indirizzo dell'array ***n*** e viene poi incrementato di 8 arrivando così al terzo elemento. Si noti che dopo l'esecuzione dell'istruzione str, il contenuto di X2 rimane invariato.

```
//Esempio 5
mov x4, #2 // Il registro X4 contiene il valore 2
str w3, [x2, x4, lsl #2]
print_all
```

Istruzione corretta. Register Offset con shift a sinistra. Il registro X2 viene incrementato del valore contenuto nel registro X4 dopo che ne è stato fatto lo shift a sinistra di 2 posizioni. Il registro X4 contiene inizialmente il valore 2 ed effettuando lo shift a sinistra è come se il suo contenuto venisse moltiplicato per 4. Infatti data una sequenza di bit, fare lo shift a sinistra di una posizione significa moltiplicare una volta per 2, e dunque fare lo shift di due posizioni a sinistra significa moltiplicare due volte per due, ovvero moltiplicare per 4. Dunque X2 viene incrementato di 8 e di conseguenza il contenuto del registro W3 sostituisce il terzo elemento dell'array. Infatti, inizialmente X2 contiene l'indirizzo dell'array ***n*** e viene poi incrementato di 8 arrivando così al terzo elemento. Si noti che dopo l'esecuzione dell'istruzione str, il contenuto di X2 rimane invariato.

```
//Esempio 6
str w3, [x2, #16]!
print_all
```

Istruzione corretta. Pre-indexed immediate Offset. Il registro X2 prima viene incrementato di 16 e dunque il contenuto del registro W3 va a sostituire il quinto elemento dell'array. Inizialmente infatti, X2 contiene l'indirizzo dell'array ***n*** e viene poi incrementato di 16 arrivando così al quinto elemento. Si noti che, in questo caso, dopo l'esecuzione dell'istruzione str, il contenuto di X2 viene modificato in accordo all'incremento.

```
//Esempio 7
str w3, [x2], #4
print_all
```
Istruzione corretta. Post-indexed immediate Offset. Il contenuto del registro W3 va a sostituire il primo elemento dell'array poiché X2 contiene l'indirizzo dell'array ***n*** e soltanto successivamente, il registro X2 viene incrementato di 4. Infatti, se dopo effettuassimo le seguenti istruzioni:
```
str w3, [x2]
print_all
```
vedremmo che il contenuto del registro W3 sostituirebbe il secondo elemento dell'array. Dunque, anche in questo caso, dopo l'esecuzione dell'istruzione str, il contenuto di X2 viene modificato in accordo all'incremento.

```
//Esempio 8
str w3, =n
print_all
```
Istruzione non corretta. Pseudo-load. Tale tipologia di indirizzamento non è consentita con l'istruzione str. L'assemblatore restituisce infatti il seguente errore:
```
Error: invalid addressing mode at operand 2 -- `str w3,=n'
```

Esercizio 3

Considerare il programma riportato nel file "esercizio3.s":

a) Aggiungere delle opportune istruzioni ldr e str (utilizzando uno a scelta tra, post o pre indexed immediate offset) per impostare a 0 (utilizzando il registro W0) gli elementi che si trovano in posizioni pari nell'array ***n***. Assemblare ed eseguire poi il programma con i seguenti comandi per verificarne il corretto funzionamento.

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

Soluzione con post-indexed

```
ldr x1, =n
str w0, [x1], #8
str w0, [x1], #8
str w0, [x1]
```

Soluzione con pre-indexed

```
ldr x1, =n
str w0, [x1]
str w0, [x1, #8]!
str w0, [x1, #8]
```

b) Si noti che non è possibile utilizzare delle istruzioni **`stp`** per svolgere il punto a. Per quale ragione?
Gli elementi in posizioni pari non sono contigui in memoria.

Esercizio 4

Considerare il programma riportato nel file “esercizio4.s”:

a) Aggiungere delle opportune istruzioni ldr e str (utilizzando uno a scelta tra, post oppure pre indexed immediate offset) per impostare al valore *x* (dichiarato nella sezione .data) tutti gli elementi nell’array ***n***. Assemblare ed eseguire poi il programma per verificarne il corretto funzionamento.

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

Soluzione con pre-indexed

```
ldr w0, x
ldr x1, =n
str w0, [x1]
str w0, [x1, #4]!
str w0, [x1, #4]!
str w0, [x1, #4]!
str w0, [x1, #4]!
```

Soluzione con post-indexed

```
ldr w0, x
ldr x1, =n
str w0, [x1], #4
str w0, [x1], #4
str w0, [x1], #4
str w0, [x1], #4
str w0, [x1]
```

b) Risolvere il punto a) utilizzando ora delle istruzioni **`stp`** con pre-indexed immediate offset.

Soluzione

```
ldr w0, x
ldr x1, =n
stp w0, w0, [x1]
stp w0, w0, [x1, #8]!
str w0, [x1, #8]
```

c) Modificare il programma in modo tale da impostare al valore dell’elemento in posizione 1 (ovvero al valore 13) tutti gli elementi nell’array ***n*** utilizzando delle istruzioni **`stp`** con post-indexed immediate offset.

Soluzione

```
ldr x1, =n
ldr w0, [x1, #4]
stp w0, w0, [x1], #8
stp w0, w0, [x1], #8
str w0, [x1]
```

Esercizio 5

Considerare il programma riportato nel file "esercizio5.s". Tale programma è una variante del programma "load_store_single_register.s" con register offset visto a lezione in cui sono state modificate le istruzioni add. In particolare, `add x3, x3, #4` è stato ovunque sostituito con `add x3, x3, #1` e dunque il contenuto del registro x3 che inizialmente è 0 viene ogni volta incrementato di 1.

Modificare opportunamente le istruzioni ldr usando register offset in modo che tale programma calcoli la somma degli elementi dell'array ***n***, ottenendo così lo stesso comportamento del programma originale visto a lezione.

Soluzione

Il registro 3 contiene inizialmente in valore 0 (`mov x3, #0`) e viene incrementato di 1 ogni volta che viene effettuata una istruzione `add x3, x3, #1`. Dal momento che n è un array di word, ciascun suo elemento occupa 4 byte. La soluzione è effettuare:

```
ldr w4, [x2, x3, lsl #2]
```

ovvero, load con register offset che carica nel registro w4 i 4 byte che si trovano all'indirizzo contenuto in x2 incrementato del valore contenuto in x3 dopo che su x3 è stato effettuato uno shift a sinistra di 2 posizioni. Tale shift fa sì che x3 sia incrementato di 4, infatti data una sequenza di bit, fare lo shift a sinistra di una posizione significa moltiplicare una volta per 2, e dunque fare lo shift di due posizioni a sinistra significa moltiplicare per 4. Ad esempio, consideriamo questa sequenza di bit 0000 0001 che in decimale corrisponde al numero 1. Facendo lo shift a sinistra di due posizioni otteniamo: 0000 0100 ovvero il numero 4. Di seguito si riporta il programma completo.

```
.section .rodata
fmt: .asciz "%d\n"
.align 2

.data
n: .word 10, 20, 30, 40, 50

.macro print i
    adr x0, fmt
    ldr x2, =n
    ldr w1, [x2, #\i * 4]
    bl printf
.endm

.text
.type main, %function
.global main
main:
    stp x29, x30, [sp, #-16]!

    print 0
```

```
print 1
print 2
print 3
print 4

// print the sum using register offset addressing: begin
mov w1, #0
ldr x2, =n

mov x3, #0
ldr w4, [x2, x3]
add w1, w1, w4

add x3, x3, #1
ldr w4, [x2, x3, lsl #2]
add w1, w1, w4

add x3, x3, #1
ldr w4, [x2, x3, lsl #2]
add w1, w1, w4

add x3, x3, #1
ldr w4, [x2, x3, lsl #2]
add w1, w1, w4

add x3, x3, #1
ldr w4, [x2, x3, lsl #2]
add w1, w1, w4

adr x0, fmt
bl printf
// print the sum using register offset addressing: end

mov w0, #0
ldp x29, x30, [sp], #16
ret
.size main, (. - main)
```

Esercizio 6

Considerare il programma riportato nel file "esercizio6.s". Tale programma riprende il programma "jump.s" visto a lezione. In particolare, nella funzione main viene memorizzato nel registro 0 il valore 0 e nel registro 1 il valore 1. Successivamente tramite l'istruzione `tst` si effettua l'AND bit a bit tra il contenuto del registro X0 e il contenuto del registro X1, ovvero tra 0 e 1 e vengono impostati in base al risultato ottenuto i flag N e Z. In base al valore del flag Z viene effettuato un salto condizionale, che fa sì che il programma stampi `true` oppure `false`: `true` se entra nel `true_case`, ovvero Z è impostato perché il risultato dell'operazione effettuata è 0; `false`, se entra nel `false_case` ovvero Z non viene impostato perché il risultato dell'operazione effettuata non è 0.

a) Assemblando ed eseguendo il programma con i due comandi riportati di seguito osserviamo che in output si ottiene `true`. Per quale motivo?

```
aarch64-linux-gnu-gcc -o esercizio6 -static esercizio6.s
qemu-aarch64 esercizio6
```

Soluzione

Viene effettuato l'AND tra i valori 0 e 1 rappresentati tramite 64 bit poiché entrambe i registri utilizzati sono a 64 bit. In particolare il registro x0 conterrà dunque 64 bit impostati tutti a 0, mentre il registro X1 conterrà 1 nel bit più a sinistra (per via della rappresentazione little endian, assumendo di essere sulla macchina virtuale) e 0 nei restanti 63 bit. Il risultato dell'AND bit a bit tra i due registri sono 64 bit tutti uguali a zero perché, ricordiamo che:

0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1

Di conseguenza il flag Z è impostato perché il risultato è 0, quindi si salta nel `true_case` viene stampato `true`.

b) Modificare ora il programma sostituendo a `tst x0, x1` con `tst x0, x0` che effettua l'AND tra 0 e 0. Rieseguire i comandi riportati al punto 1 e spiegare il motivo del risultato ottenuto.

Soluzione

In modo simile a prima, ora viene effettuato l'AND tra i valori 0 e 0 rappresentati tramite 64 bit poiché il registro utilizzato è a 64 bit. Il risultato dell'AND bit a bit tra X0 e X0 sono 64 bit tutti uguali a zero. Di conseguenza il flag Z è impostato perché il risultato è 0, quindi si salta nel `true_case` viene stampato nuovamente `true`.

c) Modificare ora il programma sostituendo a `tst x0, x0` con `tst x1, x1` che effettua l'AND tra 1 e 1. Rieseguire i comandi riportati al punto 1 e spiegare il motivo del risultato ottenuto.

Soluzione

In questo caso viene effettuato l'AND tra i valori 1 e 1 rappresentati tramite 64 bit poiché il registro utilizzato è a 64 bit. Il risultato dell'AND bit a bit tra X1 e X1 sono 64 bit: il bit più a sinistra vale 1, i restanti 63 bit valgono 0. Di conseguenza il flag Z non è impostato perché il risultato non è 0, quindi si salta nel `false_case` viene stampato `false`.

d) Modificare ora il programma sostituendo a `tst x1, x1` con `tst x1, x0` che effettua l'AND tra 1 e 0. Rieseguire i comandi riportati al punto 1 e spiegare il motivo del risultato ottenuto.

Soluzione

In quest'ultimo caso, viene effettuato l'AND tra i valori 1 e 0 rappresentati tramite 64 bit poiché entrambe i registri utilizzati sono a 64 bit. Il risultato dell'AND bit a bit tra

X1 e X0 sono 64 bit tutti uguali a zero. Di conseguenza il flag Z è impostato perché il risultato è 0, quindi si salta nel `true_case` viene stampato `true`.