AE 2020-21 Prova di ammissione all'orale – 2o appello, 3 Feb 2021

Si scrivano due funzioni in ASMv7:

- una `int convdigit(char c)`, che converte un carattere che rappresenta una cifra esadecimale (quindi un carattere in [0-9] o [a-f]) nel suo corrispondente valore. La rappresentazione ASCII dei caratteri in questione è riportata in tabella. Per le cifre "alfabetiche" si deve assumere di avere solo lettere minuscole. Il carattere passato come argomento è garantito essere nel range corretto;

Carattere	0	1	2	3	4	5	6	7	8	9		a	b	c	d	e	f
Codice ASCII	48	49	50	51	52	53	54	55	56	57		97	98	99	100	101	102

- una `int convstring(char * s)`, che converte una stringa di caratteri che rappresenta un numero in esadecimale in un intero (stesse assunzioni di prima, sui singoli caratteri, la stringa è una sequenza di caratteri terminata dal NULL (codice ASCII = 0)). La funzione fa uso della `convdigit` per convertire i singoli caratteri. L'algoritmo da seguire per la conversione è quello classico.

Si richiede che le due funzioni:

- rispettino tutte le convenzioni ARMv7 per parametri e utilizzo di registri;
- utilizzino, se possibile, i soli registri temporanei (r0-r3 ed eventualmente r8);
- siano ottimizzate per l'esecuzione sul processore single cycle.

Al termine si valuti quanti cicli di clock richiede la conversione di una stringa di *c* caratteri su un processore single cycle.

Quando si è terminato, le due funzioni e il numero dei cicli di clock devono essere copiati (in modo SOLO TESTO) nei box relativi nel compito google classroom.

Nella pagina che segue c'è un programma C che potete utilizzare per testare la correttezza delle vostre routine.

Per la conversione di una stringa si può utilizzare un codice che inizializza il risultato parziale a 0 e poi, una cifra alla volta, partendo dalla più significativa, finchè ce ne sono:

- moltiplica il risultato parziale per 16,
- calcola il valore della cifra che stiamo considerando (con la `convdigit`),
- somma il valore della cifra al parziale.

Per provare le due funzioni, si può utilizzare il codice C che segue:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int convstring(char *);

int check(char * s) {
  int i, n, ret;
  ret = (1==1);
  n = strlen(s);
  for(i=0; i<n; i++)
    if(!(s[i]>='0' && s[i]<='9') && !(s[i]>='a' && s[i]<='f'))
      ret = (1==0);
  return ret;
}

int main(int argc, char ** argv) {

  int i;
  if(argc == 1) {
    char * x[4] = {"00d","ff", "ff00", "1000"};
    for(int i=0; i<4; i++)
    printf("Stringa %s converte in %d\n",
   x[i], convstring(x[i]));
  } else {
    for(int i=1; i<argc; i++)
      if(check(argv[i]))
        printf("Stringa %s converte in %d\n",
       argv[i], convstring(argv[i]));
      else
        printf("La stringa %s non è stringa esadecimale valida\n", argv[i]);
  }
  return(0);
}
```

Compilando questo programma (**`main.c`**) con il vostro file dove sono definite le funzioni (**`fun.s`**) si ottiene un eseguibile che prende quanti parametri volete (>1) che siano stringhe esadecimali e ne stampa le conversioni eseguite con la routine assembler. Se non vengono passati parametri, si stampano le conversioni delle stringhe inserite nell'array x. Si ricorda che le funzioni da chiamare dal C dovrebbero essere dichiarate **`.type nomefunzione,%function`** nel file assembler.

Bozza di soluzione

Funzione per la conversione del singolo carattere

Nell'assunzione che sicuramente il carattere appartenga all'insieme [0-9,a-f], la funzione può essere scritta così:

```
        .text
        .global convdigit

        @ int conv(char c)
        @ restituisce il valore relativo a un carattere esadecimale (minuscolo)
        @ usa solo r0 (non occorre salvare niente prima di chiamarla)
        @ assume che il carattere sia sicuramente valido (in [0-9,a-z])

convdigit:
        cmp r0, #0x39           @ confronta con '9'
        bgt cont                @ se sono più grande è una lettera
        sub r0, r0, #0x30       @ altrimenti togli '0'
        mov pc, lr              @ e ritorna
cont:   sub r0, r0, #0x61       @ altrimenti -'a'+10
        add r0, r0, #10
        mov pc, lr              @ e ritorna
```

A regola le due `sub add` all'etichetta `cont`: possono essere compattate in una unica `add` di #87 (a è 97):

```
convdigit:
        cmp r0, #0x39           @ confronta con '9'
        bgt cont                @ se sono più grande è una lettera
        sub r0, r0, #0x30       @ altrimenti togli '0'
        mov pc, lr              @ e ritorna
cont:   addr0, r0, #87          @ altrimenti -'a'+10
        mov pc, lr              @ e ritorna
```

Dunque, a questo punto, visto che nei due casi dobbiamo solo fare una operativa (singola) diversa, potremmo utilizzare le istruzioni condizionali:

```
convdigit:
cmp r0, #0x39       @ confronta con '9'
suble r0, r0, #0x30 @ altrimenti togli '0'
subgt r0, r0, #87   @ altrimenti -'a'+10
mov pc, lr          @ e ritorna
```

Qualora di volesse controllare che il carattere sia nel range ricercato, andrebbero fatti più controlli:

```
.text
.global convdigit

@ int conv(char c)
@ restituisce il valore relativo a un carattere esadecimale (minuscolo)
@ usa solo r0 (non occorre salvare niente prima di chiamarla)

convdigit:
cmp r0, #0x30       @ confronta con '0'
```

```
blt cont            @ se sono più piccolo
cmp r0, #0x39       @ confronta con '9'
bgt cont            @ se sono più grande
sub r0, r0, #0x30   @ altrimenti togli '0'
mov pc, lr          @ e ritorna
cont:  cmp r0, #0x61       @ se sono più piccolo di 'a'
blt cont2
cmp r0, #0x7a       @ o più grande di 'z'
bgt cont2
sub r0, r0, #0x61   @ altrimenti -'a'+10
add r0, r0, #10
mov pc, lr          @ e ritorna
cont2: mov r0, #0          @ errore => -1
sub r0, r0, #1
mov pc, lr
```

Funzione per la conversione della stringa (NULL terminated)

Nell'ipotesi che la stringa sia una stringa esadecimale valida, il codice può essere scritto come segue:

```
        .text
        .global convstring
        .type convstring, %function
        @ function int convstring(char * str)
        @ r0 string pointer (null terminated)

convstring:
        push {r4-r5, lr}    @ salva i registri che servono per str ptr e sum
        mov r4, r0          @ indirizzo stringa (str ptr)
        mov r5, #0          @ risultato         (sum)
convloop:
        ldrb r0,[r4],#1     @ carica un byte e aggiorna punt al prossimo byte
        cmp  r0, #0         @ controlla se è un NULL
        beq  fineconv       @ in questo caso ritorna
        bl convdigit        @ converti il carattere corrente (già in r0)
        lsl r5, r5, #4      @ moltiplica il risultato fino ad ora per 16
        add  r5, r5, r0     @ e somma il contributo attuale al parziale
        b convloop          @ passa al prossimo caratter
fineconv:
        mov r0, r5          @ valore da restituire
        pop {r4-r5,pc}      @ ripristino registri e ritorno
```

Qualora dovessimo anche controllare la validità della stringa:

```
                .text
                .global convstring
                .type convstring, %function

                @ function int convstring(char * str)
                @ r0 string pointer (null terminated)

        convstring:
                push {r4-r5, lr}      @ salva i registri che servono per str ptr e sum
                mov r4, r0              @ indirizzo stringa     (str ptr)
                mov r5, #0              @ risultato             (sum)
        convloop:
                ldrb r0,[r4],#1      @ carica un byte e aggiorna punt al prossimo byte
                cmp  r0, #0             @ controlla se è un NULL
                beq  fineconv           @ in questo caso ritorna
                bl convdigit            @ converti il carattere corrente
                cmp r0, #0              @ controlla validità
                blt errore
                lsl r5, r5, #4
                add  r5, r5, r0         @ sommalo al risultato parziale
                b convloop              @ passa al prossimo
        fineconv:
                mov r0, r5              @ valore da restituire
                pop {r4-r5,pc}          @ ripristino registri e ritorno
        errore:
                mov r0, #-1             @ restituisce -1
                pop {r4-r5,pc}
```

Una possibile, minima ottimizzazione è relativa all'operazione di sommare il valore della cifra attuale alla somma parziale moltiplicata per 16, ovvero le due istruzioni LSL e ADD alla fine dell'iterazione del ciclo sui caratteri della stringa. L'espressione che calcoliamo è (R5 * 16) + R0. Per la proprietà commutativa della somma potremmo calcolare R0 + (R5 * 16). Il vantaggio è che a questo punto R5*16 diventa il Src 2 e quindi possiamo calcolare lo stesso risultato delle due istruzioni LSL e ADD con una ADD R5, R0, R5, LSL #4. Il ciclo principale della routine diventa quindi:

```
convloop:
        ldrb r0,[r4],#1         @ carica un byte e aggiorna punt
        cmp  r0, #0             @ controlla se è un NULL
        beq  fineconv           @ in questo caso ritorna
        bl convdigit            @ converti il carattere corrente
        add  r5,r0,r5, lsl #4   @ somma al parziale motliplicato per 16
        b convloop              @ passa al prossimo
```

Valutazioni sul processore single cycle

Ciascuna delle valutazioni della funzione che converte il singolo carattere richiede 4 istruzioni. Lo scorrimento della stringa richiede 3 istruzioni per il preambolo (push, mov, mov), 6 istruzioni per ciascun carattere, 3 istruzioni per l'ultima iterazione e infine 2 istruzioni per postambolo (mov e pop), per un totale di 3+6c+3+2 = 8+6c istruzioni. Sul single cicle questo significa altrettanti cicli di clock.

Ulteriore possibile ottimizzazione

Se consideriamo che la routing convdigit la scriviamo appositamente per essere chiamata nella convstring e non la prendiamo già preconfezionata da una libreria, possiamo anche fare affidamento sul fatto che sappiamo quali registri utilizza effettivamente. La routine, nell'ultima versione di sole 4 istruzioni macchina, utilizza in effetti il solo registro r0. Quindi per utilizzarla nella convstring, possiamo far affidamento sul fatto che non vengono utilizzati i registri r1-r3 e di conseguenza possono essere utilizzati per mantenere valori scritti prima della bl convdigit e letti successivamente. Questo significa che potremmo utilizzare nella convstring i soli registri temporanei e non avremmo bisogno di utilizzare r4 4d r5. Rimane il problema del link register LR. Questo deve essere salvato prima della bl convdigit in modo da poter restituire correttamente il controllo al codice che chiama la convstring. Dal momento che r0 ci serve per interagire con la convdigit, possiamo utilizzare r1 per mantenere l'indirizzo della stringa, r2 per il risultato e r3 per salvare il LR. Il codice della convstring diventa quindi:

```
convstring:
mov r1, r0          @ indirizzo stringa (str ptr)
mov r2, #0          @ risultato         (sum)
mov r3, LR          @ salvo LR per il ritorno al chiamante
convloop:
ldrb r0,[r1],#1     @ carica un byte e aggiorna punt al prossimo byte    cmp
r0, #0              @ controlla se è un NULL
beq  fineconv       @ in questo caso ritorna
bl convdigit        @ converti il carattere corrente
add  r2, r0, r2, lsl #4    @ sommalo al risultato parziale
b convloop          @ passa al prossimo
fineconv:
mov r0, r2          @ valore da restituire
```

```
mov PC, r3
```

Con la convdigit da 4 istruzioni, occorrono quindi 8+6k cicli di clock per calcolare la conversione di una stringa da k caratteri. Le istruzioni sono lo stesso numero del caso senza ottimizzazione ma non viene utilizzato lo stack.