

Minitarea 6

Autor: Lukas Pavez
RUT: 19.401.577-1
Profesor: Pablo Guerrero P.
Auxiliar: Matías Torrealba A.
Ayudantes: Gabriel Chandía G.
Gaspar Ricci
Fecha de entrega: 26/11/2018

1. Solución del problema

El lenguaje utilizado en esta tarea fue c++.

Para resolver el problema, se modificó uno de los ejemplos que viene con sexpr: `simple_inter.c` (link: [simple_interp.c](#)).

Al principio del código, se crea la variable global `reg_num`, que parte en 0 e indica el próximo registro a utilizar, ya que en el programa los registros se utilizarán de forma incremental (desde el 0 en adelante), luego se escribe en el archivo de salida (con `fprintf`) el encabezado de el programa en assembler ARM:

```
.data
string:
    .ascii "%d\n"
.text
.global main
main:
    stmfd sp!, {fp,lr}
```

Donde en el encabezado se define `string`, que es donde se guardará el resultado para mostrarlo en la pantalla. También se define el comienzo de la función global `main`, y se hace un storage de los registros `fp` y `lr` en el registro `sp` (se hace push de los registros a la pila).

Luego con la librería `sexpr` (no se modifico nada de lo que había en el `main` del ejemplo con respecto a esta parte) se lee el archivo para obtener las instrucciones, y se llama a la función para escribirlas en arm, en esto la función que traduce tiene 3 casos:

1. `INT_CONST n`
2. `ADD`
3. `SUB`

1.1. `INT_CONST n`

En este caso, primero se crea el string con el registro a utilizar, que es concatenar una `r` y `reg_num`, luego se hace la operación de mover `n` al registro y apilarlo:

```
mov reg, #n
stmfd r13!, {reg}
```

Donde `reg` es el registro a utilizar y `n` el número recibido. Después se le suma 1 a `reg_num` (El próximo registro a utilizar).

1.2. ADD

Con esta instrucción se parte restando 1 a `reg_num` para tener acceso al último registro utilizado, y se comprueba si es 1 o mayor (en caso contrario solo habría un registro en la pila y no se podría realizar la operación), luego se hace pop y se guardan los valores en los 2 últimos registros, se hace un add y se hace push del resultado:

```
ldmfd r13!, {reg2}
ldmfd r13!, {reg1}
add reg1, reg2, reg1
stmfd r13!, {reg1}
```

Donde `reg2` es el último registro utilizado y `r1` el anterior a ese.

1.3. SUB

En esta instrucción se hace lo mismo que con ADD, solo que se cambia la operación add por sub:

```
ldmfd r13!, {reg2}
ldmfd r13!, {reg1}
sub reg1, reg2, reg1
stmfd r13!, {reg1}
```

Finalmente se agrega la parte de imprimir el resultado en pantalla, donde se desapila el resultado, se carga en el string y se llama a la función `printf`, y termina cargando `sp` en `fp` y `pc`:

```
ldmfd r13!, {r1}
ldr r0, =string
bl printf
ldmfd sp!, {fp, pc}
```

Esto imprime el resultado en pantalla y deja la pila vacía.

2. Resultados

Se hicieron 4 pruebas:

1. $((\text{INT_CONST } 3) (\text{INT_CONST } 8) (\text{INT_CONST } 6) (\text{ADD}) (\text{SUB})) = 6 + 8 - 3 = 11$
2. $((\text{INT_CONST } 8) (\text{INT_CONST } 1) (\text{SUB})) = 1 - 8 = -7$
3. $((\text{INT_CONST } 8) (\text{INT_CONST } 6) (\text{ADD})) = 6 + 8 = 14$
4. $((\text{INT_CONST } 3) (\text{INT_CONST } 8) (\text{ADD}) (\text{INT_CONST } 6) (\text{ADD})) = 6 + 8 + 3 = 17$

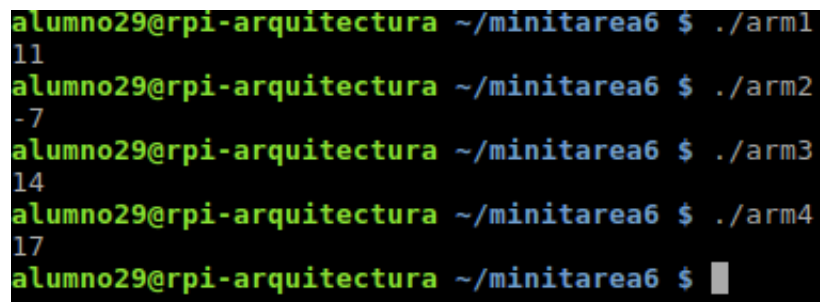
El primer test evalúa si se pueden apilar valores y después realizar suma y resta.

El segundo evalúa si se pueden obtener números negativos en los resultados.

El tercero evalúa si se puede realizar una suma simple.

El cuarto evalúa el funcionamiento de apilar números, realizar una operación, después volver a apilar y realizar otra operación.

Los resultados se pueden ver en la siguiente imagen:



```
alumno29@rpi-arquitectura ~/minitarea6 $ ./arm1
11
alumno29@rpi-arquitectura ~/minitarea6 $ ./arm2
-7
alumno29@rpi-arquitectura ~/minitarea6 $ ./arm3
14
alumno29@rpi-arquitectura ~/minitarea6 $ ./arm4
17
alumno29@rpi-arquitectura ~/minitarea6 $
```

Figura 1: Resultados minitarea 6

3. Instrucciones de ejecución y ensamblado

Primero se debe compilar `simple_interp.cpp` con la siguiente instrucción en linux:

```
g++ simple_interp.cpp sexprPath/libsexp.a -o readList
```

Para generar el código, se toma por default el archivo `test1.in`, pero si se quiere usar otro archivo, se debe agregar al ejecutar el programa:

```
./readList  
./readList nombreArchivo
```

Para ensamblar y ejecutar el programa, se debe mover el resultado obtenido (`arm.s`) a la raspberry, y ejecutar los siguientes comandos (también dentro de la raspberry):

```
as -o arm.o arm.s  
gcc -o arm arm.o  
./arm
```