Universidad de Costa Rica
Escuela de Ciencias de la Computación e Informática
CI0118 Lenguaje Ensamblador, grupo 01
Fecha: 09/05/2019, I ciclo lectivo 2019

Tarea Programada # 2

Integrantes del grupo

Iván Chavarría Vega, B72097

Filip Sobejko Bogusz, B77382

1. Introducción

Esta tarea programada tiene como objetivo desarrollar una máquina cuyo hardware no existe, en la cual vamos a simular la ejecución de diferentes instrucciones, entre ellas la suma y la resta. Todo esto se desarrollará en lenguaje ensamblador para arquitecturas x86-64, tomando como guia el manejo de funciones que se encuentra en el libro de Ed Jorgensen [1]. En esta se hizo uso primordialmente de la pila, esto porque se implementará una subrutina **instr** la cual es la que se encargará de realizar todo el manejo de hacer la suma o la resta, esto dependiendo de los operandos que le ingrese el usuario, los cuales se necesitan guardar en la pila para poder crear un *stackframe*. Las instrucciones de esta CPU tienen el siguiente formato: cod.Op, T1, Op1, T2, Op2, T3, Op3, vector de memoria de datos, vector de registros, en donde:

- cod. Op es el código de operación
- Ti es el tipo de este operando: operando de registro, de memoria o inmediato
- Opi es la dirección del operando.
- vector de memoria de datos es la dirección del inicio del arreglo de 1024 bytes
- vector de registros es la dirección del inicio de de una tabla que contiene el arreglo de registros.

Si fuera un lenguaje de alto nivel, se vería de la siguiente manera: **instr(ADD, R, R0, I, 100, I, 0, memarray, registers)**

2. Descripción del Algoritmo

El algoritmo funciona de manera similar a como lo haría un *switch* en un lenguaje de alto nivel. Esto debido a que se tienen diferentes casos posibles, ya que **Ti** puede ser tanto un operando de registro, de memoria o inmediato.

Una vez inicia el programa, lo primero que hace es guardar los operandos necesarios, y en los seis registros ya designados los primeros seis parámetros de la subrutina. Estas instrucciones y el estado inicial de la pila una vez guardado los operandos se puede observar en Figura 1. Seguidamente, se llama a la instrucción **instr** la cual es la que se encarga de realizar los cálculos deseados.

```
; seccion de codigo
section .text
global _start
_start:

mov r10, 0

push registers
push memarray
push r10

mov r9, 3
mov r8, 100
mov rcx, 3
mov rdx, 0
mov rsi, 1
mov rdi, 4
call instr
```

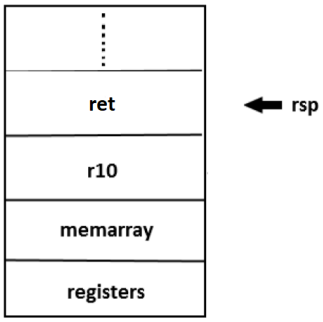

Figura 1: Estado inicial de la pila

El programa va a tener dos posibles casos iniciales:

- Suma: una vez llamada la subrutina **instr**, el prólogo de la subrutina va a hacer el **push** de los registros que se van a utilizar para hacer las operaciones, ver Figura 2. Los registros a usar serían los siguientes:
  - rbp: este registro será el encargado de establecer el stack frame. Para ello, se hará el **push** de este registro, y luego se moverá el registro **rsp** al registro **rbp**.
  - r11: este registro será el encargado de servir como puntero al **Op3**, el cual está en la pila.
  - r12: este registro será el encargado de almacenar el valor del **Op2**, para posteriormente utilizarlo en la operación de la suma.
  - r13: este registro será el encargado de almacenar el valor del **Op3**, para posteriormente utilizarlo en la operación de la suma.
  - r14: este registro será el encargado de servir como puntero al vector de registros.
  - r15: este registro será el encargado de servir como puntero al vector de memoria.
  - rbx: este registro será el encargado de almacenar valores, servirá como un temporal.

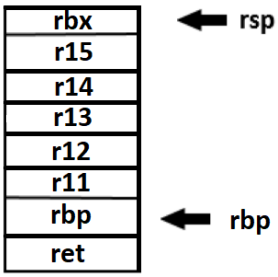

Figura 2: Estado de la pila después del prólogo de **instr**

Posteriormente se limpian los registros a usar, luego se mueven las direcciones del **Op3**, del vector de registros y del vector de memoria a los registros designados anteriormente. A continuación, inicia el cuerpo de la subrutina, en esta etapa se van a realizar una serie de comparaciones necesarias para determinar el tipo de cada operando. Algoritmo 1 muestra un fragmento del cuerpo de la subrutina que revisa los tipos de operandos.

Algoritmo 1 Fragmento del cuerpo de la subrutina.

```
esMemoria2:
        cmp rcx, 2
        jne esInmediato2
        mov r12, [r15+r8]
        jmp esRegistro3

esInmediato2:
        mov r12, r8

esRegistro3:
        cmp r9, 1
        jne esMemoria3
        mov rbx, [r14+r11*8]
        mov r13, rbx
        jmp sumarRegistro
```

Primero, se va a determinar el tipo del segundo operando, esto se va realizar mediantes una serie de comparaciones, usando el registro donde está guardado el tipo. Además, una vez obtenido el tipo (registro, memoria, inmediato), se va asignar el valor de ese operando al registro correspondiente establecido anteriormente. Segundo, se va a determinar el tipo del tercer operando de la misma manera que se hizo con el segundo, y se va a asignar el valor de la misma manera. Tercero, se va a obtener el tipo del primer operando, esto con el objetivo de saber donde vamos a guardar la suma del segundo operando con el tercero. Una vez obtenido el tipo, procedemos a realizar la suma y guardarla en el primer operando. Finalmente, la subrutina va a llegar a su epílogo, donde va a realizar el **pop** de los registros previamente insertados. Luego de remover los registros, la subrutina vuelve a su dirección de retorno, para volver al programa principal y seguir ejecutando las instrucciones restantes. Ver Figura 3.

```
final:
    pop rbx
    pop r15
    pop r14
    pop r13
    pop r12
    pop r11
    pop rbp
    ret
```

Figura 3: Epílogo de la subrutina.

- Resta: una vez llamada la subrutina **instr**, el prólogo de la subrutina va a hacer el **push** de los registros que se van a utilizar para hacer las operaciones. Los registros a usar serían los siguientes:
    - rbp: este registro será el encargado de establecer el stack frame. Para ello, se hará el **push** de este registro, y luego se moverá el registro **rsp** al registro **rbp**.

- r11: este registro será el encargado de servir como puntero al **Op3**, el cual está en la pila.
- r12: este registro será el encargado de almacenar el valor del **Op2**, para posteriormente utilizarlo en la operación de la suma.
- r13: este registro será el encargado de almacenar el valor del **Op3**, para posteriormente utilizarlo en la operación de la suma.
- r14: este registro será el encargado de servir como puntero al vector de registros.
- r15: este registro será el encargado de servir como puntero al vector de memoria.
- rbx: este registro será el encargado de almacenar valores, servirá como un temporal.

Posteriormente se limpian los registros a usar, luego se mueven las direcciones del **Op3**, del vector de registros y del vector de memoria a los registros designados anteriormente. A continuación, la subrutina va a entrar a una etiqueta que compara el registro **dil** con un ocho, esto con el fin de verificar si la instrucción es una resta. Si es un ocho, entonces se van a realizar una serie de comparaciones para verificar el tipo (registro, memoria, inmediato) del tercer operando. Esto con el fin de entrar a la etiqueta correcta, para luego negar ese tercer operando. Una vez hecho esto, se va a hacer un **push** del **r14**, **r15** y **11**. Ver Figura 4.

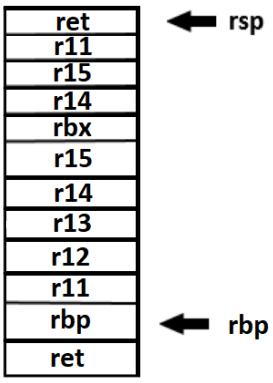

Figura 4: Estado de la pila antes del llamado recursivo

Esto con el fin de preservar esos registros para utilizarlos posteriormente en el siguiente llamado recursivo, ver Figura 5. Por último, se va a mover un cuatro al registro **dil**, para que a la hora de llamar **instr** otra vez, se haga la instrucción de suma con el tercer operando ya negado.

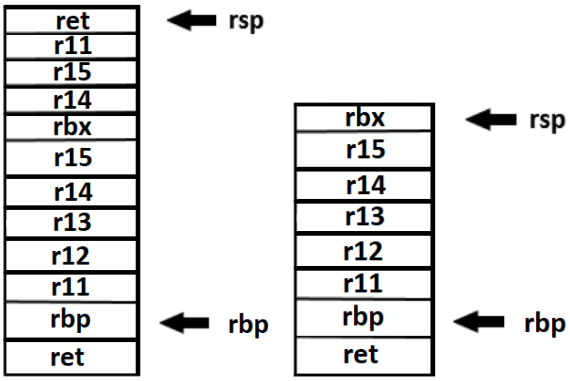

Figura 5: Estado de la pila después del llamado recursivo

Después de realizar la instrucción de suma, la subrutina va a retornar al lugar donde se hizo el llamado en la instrucción de resta, y va a empezar el epílogo de la subrutina, se va a hacer

pop de los registros insertados anteriormente (r14,r15,r11) y por último se va a hacer **pop** de los registros insertados al inicio de la subrutina. Luego de remover los registros, la subrutina vuelve a su dirección de retorno, para volver al programa principal y seguir ejecutando las instrucciones restantes.

3. Ejecución del Programa

Para este programa se tienen tres instrucciones a ejecutar:

- **Primera instrucción:** Se realiza una suma entre 100 y 0, la cual, por supuesto, da 100 como resultado.
- **Segunda instrucción:** Se realiza una suma entre el resultado de la instrucción anterior, el cual es 100, más 50, dando como resultado 150.
- **Tercera instrucción:** Se realiza una resta entre 1000 y el resultado de la instrucción anterior, el cual fue 150, dando como resultado 850.

Accediendo a los datos de **registers** con el comando **x/8dh registers** desde el debugger GDB se puede obtener los datos actuales del registro. Este comando se ejecutó una vez se ejecutó la primera instrucción del programa, la cual sucedió como se esperaba, guarda como resultado 100. Ver Figura 6

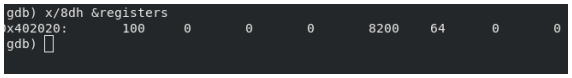

Figura 6: Resultado de la primera instrucción

De la misma manera, el resultado de la segunda ejecución dio lo esperado, se sumó lo obtenido en la primera instrucción con 50 para un total de 150. Para este fue necesario acceder a los datos del arreglo de memoria, **memarray**, con el comando de GDB **x/200dh memarray**. Ver Figura 7.

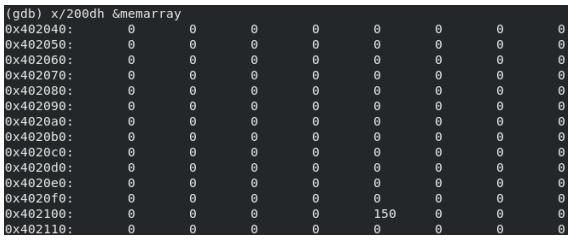

Figura 7: Resultado de la segunda instrucción **Ti**

Finalmente, se llega a la tercera instrucción, en la cual se espera que el programa le resta a 1000 lo obtenido en la segunda instrucción, lo cual es 150, para un total de 850. Lo cual sucede perfectamente. Para esta instrucción fue necesario acceder a el arreglo de memoria nuevamente con el comando **x/40 memarray**. Ver Figura 8.

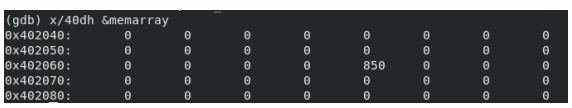

Figura 8: Resultado de la tercera instrucción

Referencias

[1] E. Jorgensen, *x86-64 Assembly Language Programming with Ubuntu*. 2018.