Práctica 8 - Lenguaje Ensamblador

Tecnología Digital II

Los siguientes ejercicios están ideados para ser realizados en lenguaje ensamblador de x86-64 o AMD64. Ya sea que se realicen en papel o en computadora, se proveen archivos complementarios para ejecutar y resolver cada uno de los ejercicios. En el caso de tener que imprimir texto, se puede utilizar el servicio del sistema para imprimir visto en clase. Para imprimir el valor numérico de un registro se provee la función `printHex` en los archivos complementarios. La función toma un número entero de 64 bits desde el registro `rdi` y lo imprime en pantalla.
Para ensamblar los archivos complementarios se puede utilizar el comando `make` que construye los ejecutables de todos los ejercicios.

Ejercicio 1

a. Realizar una programa en ASM que imprima su nombre en pantalla.
Ejemplo:
resultado:
```
Pepe
```

b. Modificar la sección `.data` del código anterior para imprimir una letra debajo de la otra.
Ejemplo:
resultado:
```
P
e
p
e
```

c. Considerando la solución de los ejercicios anteriores, ¿cuántas veces se llama al sistema para imprimir en pantalla en cada caso?

Ejercicio 2

Realizar un programa en ASM que tome dos números enteros sin signo de 16 bits y los multiplique utilizando sumas sucesivas. Imprimir en pantalla el valor resultado en hexadecimal.

Ejemplo:
números: `[10, 22]`, resultado: `0xDC` (220d)

Recordatorio: Las instrucciones reciben operandos del mismo tamaño:
`ADD eax, ebx` es una instrucción válida y suma los números de 32 bits almacenados en `eax` y en `ebx`.
`ADD eax, bx` no es una instrucción válida porque el operando `eax` tiene 32 bits y el operando `bx` tiene 16 bits.

Ejercicio 3

Dado un arreglo de 10 números de 32 bits en complemento a dos. Realizar la suma de todos los valores e imprimir en pantalla si la suma total resultó en un número positivo o negativo.

Ejemplo:
arreglo: `[-10, 2, -8, 17, -1, 20, -9, 52, 4, 12]`, resultado: `Positivo`
arreglo: `[-10, 2, -8, -7, -1, -1, -9, -2, 4, 12]`, resultado: `Negativo`

Tip: Repasar el último ejemplo de las diapositivas de la clase teórica de Lenguaje Ensamblador (contar la cantidad de números pares).

Ejercicio 4

a. Recorrer un arreglo de 16 números de 16 bits sin signo y determinar cuál es el mayor. Imprimir en pantalla el valor resultante en hexadecimal.

Ejemplo:

arreglo: `[10, 0, 39, 2, 8, 17, 51, 28, 2, 3, 20, 39, 52, 34, 12, 1, 8]`
resultado: `0x34` (52d)

b. ¿Qué modifcaciones debe realizarse sobre el ejercicio anterior para soportar números con signo de 16 bits?

Ejercicio 5

a. Dados 3 números enteros de 8 bits, 16 bits y 32 bits respectivamente en complemento a 2, determinar si los tres números son positivos, o son negativos, o si hay positivos y negativos entre los tres números. Imprimir en pantalla el resultado.

Ejemplo:

números: `[10, -200, 3233]`, resultado: `Hay positivos y negativos`
números: `[-21, -21, -300]`, resultado: `Son todos negativos`
números: `[111, 411, 8934]`, resultado: `Son todos positivos`

b. Si el primer número fuera de 64 bits, ¿qué modificaciones le tendría que hacer a su programa para soportar este cambio?

Ejercicio 6

Dado un arreglo de 10 números de 32 bits en complemento a 2, determinar si existe un par de números del arreglo que sumen un valor objetivo. En caso de existir al menos un par de números que sumen el valor objetivo, imprimir `verdadero`, en caso contrario `falso`.

Ejemplo:

arreglo: `[1, -2, 3, 0, -9, 2, 4, 2, 1, 9]`, objetivo: `5`, resultado: `verdadero`
arreglo: `[1, -2, 3, 0, -9, 2, 4, 2, 1, 9]`, objetivo: `8`, resultado: `falso`

Recomendación: Pensar cómo lo resolvería en Python. Luego, mirando el programa escrito en Python reflexione de qué manera recorre el arreglo y cuáles son las variables que necesitó para recorrerlo de ese modo.

Ejercicio 7

Implementar en lenguaje ensamblador de `x86-64` dos sabores de la función `mergeLow`. Ambas toman un par de valores de 32 bits en los registros `ESI` y `EDI` y los combinan según muestra la figura. Las funciones retornan el resultado en `EAX`.

a. `mergeLow1`

Ejemplo:

`ESI=0xFF0081A1, EDI=0x00ABBA0F → EAX=0x81BAA10F`

`ESI=0xFFFFFFFF, EDI=0x00000000 → EAX=0xFF00FF00`

b. `mergeLow2`

Ejemplo:

`ESI=0xFF0081AA, EDI=0x00AAAA0F → EAX=0x81AAAA0F`

`ESI=0xFFFFFFFF, EDI=0x00000000 → EAX=0xFFFF0000`

Ejercicio 8

Suponer el siguiente código ensamblador de una función que suma un arreglo de elementos de 32 bits a partir de la dirección `RSI`. La longitud del arreglo se pasa como parámetro en el registro `EDI`. El resultado de la sumatoria se presenta en `EAX`.

```
sum:
   PUSH RSI
   PUSH RDI
   MOV EAX, 0
   ciclo:
       ADD EAX, [RSI]
       ADD RSI, 4
       SUB EDI, 1
       CMP EDI,0
       JNZ ciclo
   POP RDI
   POP RSI
   RET
```

a. Usando de referencia el código anterior, implementar la función `sumNotIf` que suma todos los valores de un arreglo pero ignora los que sean iguales al un valor pasado por parámetro. La función utilizará los mismos parámetros que `sum`, agregando el registro `ECX` como el valor a ignorar.

Ejemplo:

`RSI=[2,3,4,5,6], EDI=5, ECX=1000 → EAX=20`

`RSI=[-2,-3,4,-5,6], EDI=5, ECX=3 → EAX=0`

`RSI=[22,3,4,22,6], EDI=5, ECX=22 → EAX=13`

b. Usando de referencia el código anterior, implementar la función `sumSorted` que suma todos los valores de un arreglo solo si están ordenados en orden estrictamente creciente. Es decir, sí se considero el `A[i]` en la suma, el próximo a sumar debe ser mayor a este. La función utilizará los mismos parámetros que `sum`.

Ejemplo:

`RSI=[2,3,4,5,6], EDI=5 → EAX=20`

`RSI=[-2,-3,4,-5,6], EDI=5 → EAX=8`

`RSI=[22,3,4,22,6], EDI=5 → EAX=22`