Universidad de Costa Rica
Escuela de Ciencias de la Computación e Informática
CI0118 Lenguaje Ensamblador, grupo 01
Fecha: 24/04/2019, I ciclo lectivo 2019

Tarea Programada # 1

Integrantes del grupo

Iván Chavarría Vega, B72097

Filip Sobejko Bogusz, B77382

1. Introducción

Esta tarea programada tiene como objetivo desarrollar un algoritmo en lenguaje ensamblador, para arquitecturas x86-64, el cual sea capaz de recorrer un arreglo de palabras dado y retornar cuales de estas son palíndromo o no. Para esto, vamos a tomar como 1 ser palíndromo y 0, no ser palíndromo. Por ende, la salida del programa será devolver una hilera binaria con el resultado de cada palabra. Todo esto se realizará tomando como guía el libro de Ed Jorgensen [1]

2. Descripción del Algoritmo

El algoritmo consta de tres bloques principales: el encargado de recorrer el arreglo principal de palabras, el encargado de obtener el tamaño de la palabra actual, y, finalmente, el que retorna si la palabra actual es palíndromo o no.

- **Recorrer arreglo principal:** en este bloque no se realizan tantas instrucciones como en los otros dos, solamente llenamos diferentes registros, necesarios para tareas futuras, con ceros (esto para evitar posible basura almacenada), seguidamente, obtenemos la primera hilera del arreglo, y saltamos a la etiqueta encargada de obtener el tamaño de la palabra actual. Una vez todas las instrucciones de dicha etiqueta hayan terminado, aumentamos un registro contador en 1, y volvemos a empezar el ciclo. Esto nos permite poder movernos a la segunda hilera en el arreglo y así sucesivamente.

- **Obtener tamaño de palabra:** este bloque va estar a cargo de obtener el tamaño de cada hilera del arreglo con el fin de establecer un puntero a la última posición de dicha hilera. Este puntero se designará en su respectiva etiqueta para evitar errores de asignación. Además, el puntero se usará posteriormente en las comparaciones realizadas para la determinación del palíndromo. En cuanto a la obtención del tamaño de la hilera, cada hilera será inicializada de tal manera que terminará con un carácter nulo. Esto nos beneficia a la hora de realizar la comparación, ya que de esta manera podemos comparar desde el inicio de la hilera hasta encontrarnos con un byte 0, y ahí sabemos que hemos llegado al final de la hilera. Esto se ejecutará dentro de un ciclo, y tendrá sus respectivos contadores, tanto para avanzar dentro del registro que contiene la hilera, como para mantener la cuenta de cuantos caracteres tiene la hilera. En el momento que se active la bandera del cero, se realizará un salto al ciclo que determinará si la palabra es un palíndromo. Este ciclo se explicará en el siguiente bloque.

- **Es palíndromo:** bloque principal del algoritmo. Determina si la hilera actual es palíndromo o no. Este bloque hará uso de tres registros esenciales: *rsi, rdi* y *al*, ; donde *rsi* representa el primer caracter de la hilera actual, por ende se inicializa en cero, *rdi* representa el último caracter en la hilera actual (para esto se inicializa con el tamaño de la hilera actual, el cual se obtuvo en el bloque anterior), y *al* se inicializa con el número 128 (esto para poder sumar el número correspondiente a cada iteración para que el resultado sea el esperado). Seguidamente, empieza el ciclo en el cual se realizan todas las comparaciones. Este ciclo tendrá tres posibles maneras de detenerse: la primera es que el caracter direccionado desde *rsi* sea diferente al caracter direccionado desde *rdi*, la segunda será cuando *rdi* sea menor a *rsi* (esto es por si la hilera actual es impar), y la tercera es cuando terminan todas las comparaciones entre caracteres posibles, lo cual se sabe cuando la dirección de *rsi* sea igual a la dirección de *rdi*. En caso de que suceda lo primero, nos movemos a una etiqueta en la cual le realizamos un *shr (shift right)* a nuestro registro *al*, esto es equivalente a dividir entre dos, y realizamos un *jump* hasta el primer bloque para que realice de nuevo todas las instrucciones con la hilera siguiente en el arreglo. En caso contrario, este bloque se vuelve a ejecutar después de incrementar en uno a rsi (movernos al caracter siguiente en la hilera) y decrementar en uno a rdi (movernos al caracter anterior en la hilera). Si después de todas estas comparaciones, el caracter direccionado desde rsi nunca fue diferente al de rdi, llegará un momento en que la dirección de ambos sea la misma, o la de rdi sea menor, cuando esto suceda, nos movemos a una etiqueta en la cual se le suma el dato actual de *al* a *r11b* (el cual es nuestro registro resultado), y se realiza un *shr* a *al*. Finalmente, realizamos un *jump* hacia el primer bloque para que vuelva a suceder todas las instrucciones con la hilera siguiente. Estos bloques se muestran en el diagrama de la Figura 1.

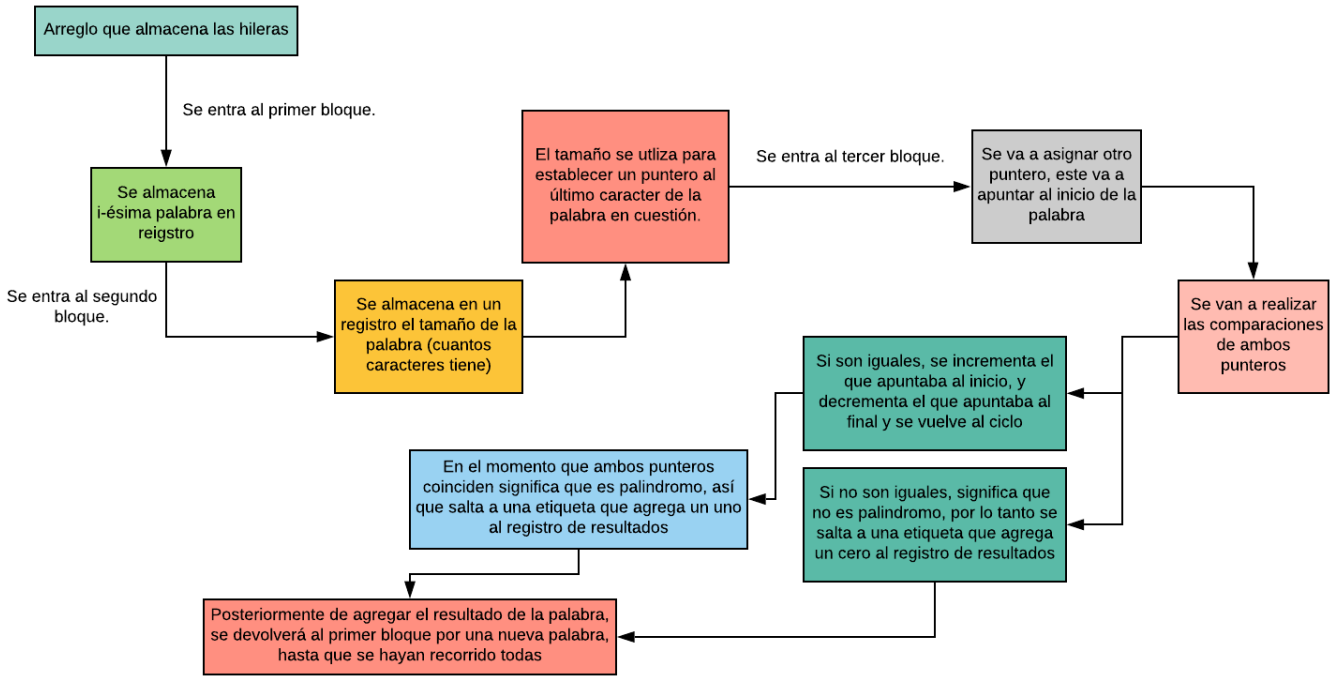

Figura 1: Bloques del algoritmo.

3. Ejecución del Algoritmo

Las hileras utilizadas en la primera parte fueron las siguientes: 'notardesysedraton', 'hasoidoalaodiosahadamariajose', 'a', 'ab', 'aa', 'aba', 'abc', y 'abba'.

Si se observa Figura 2, el resultado obtenido fue 173, que en binario es equivalente a **10101101**, donde 1 representa ser palíndromo, y 0 no serlo, es decir, si se lee de izquierda a derecha el número binario se obtiene lo siguiente: hilera1 es palíndromo, hilera2 no lo es, hilera3 si, y así sucesivamente.

```
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ yasm -g dwarf2 -f elf64 primeraTarea.asm -l primeraTarea.lst
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ ld -g -o primeraTarea primeraTarea.o
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ ./primeraTarea
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ echo $?
173
```

Figura 2: Ejecución de la primera parte de hileras

Por otro lado, las hileras utilizadas en la segunda parte fueron las siguientes: 'adannocallaconnada', 'anitalagordalagartonanotragaladrogalatina', 'davidanadaeresmiamadanavidad', 'secortasaritaatirasatroces', 'asimaloirasorrosariolamisa', 'saritasosaesidoneaenodiseasosatiras', 'sorrebecahiceberror', y 'olamoromoramalo'.

Si se observa Figura 3, el resultado obtenido fue 220, que en binario es equivalente a **11011100**, y de la misma manera a la primera parte, 1 representa ser palíndromo, y 0 no serlo. Leyendo el número binario de izquierda a derecha se obtiene lo siguiente: hilera1 es palíndromo, hilera2 también, hilera3 no lo es, y así sucesivamente.

```
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ yasm -g dwarf2 -f elf64 primeraTarea2.asm -l primeraTarea2.lst
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ ld -g -o primeraTarea2 primeraTarea2.o
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ ./primeraTarea2
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$ echo $?
220
ivanj@IvanJ:~/Desktop/Tareas/Ensamblador/Filip$
```

Figura 3: Ejecución de la segunda parte de hileras

4. Conclusiones

Al realizar este algoritmo, pudimos entender la flexibilidad del lenguaje ensamblador para explotar al máximo los recursos de la máquina. Este lenguaje permite tener un control muy preciso de las tareas realizadas por el microprocesador, por lo que se pueden crear subrutinas bastante eficientes. Sin embargo, al ser un lenguaje de tan bajo nivel, se requieren más instrucciones para realizar procesos simples, por lo tanto requiere de más cuidado por parte del programador. Esto a su vez implica tener un manejo sólido de la teoría presentada en el libro de Jorgensen [1], porque, como se mencionó anteriormente, al ser un lenguaje de tan bajo nivel, el encontrar errores se vuelve más complicado comparado con otros lenguajes de programación, especialmente porque no se cuenta con IDEs tan avanzados que faciliten este proceso.

Referencias

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