Cesar Alejandro Guayara, Felipe Jiménez

[[1]](#footnote-1)

Máquinas Digitales, parcial #2

*Abstract*— In the following article the application of assembly language "MIPS" is analyzed to create different algorithms than in any higher-level language could be made more easily. The first algorithm that was performed intended to convert a string to uppercase and lowercase letters alternately. The second algorithm calculated any position, given by the user, of the Fibonacci sequence. Implementing several tools that Assembler language provides, and using some basic programming knowledge ,such as mathematical operations, string management, cycles and functions, you can solve the two cases to be discussed in the article; concluding that this language helps the programmer to understand it's algorithms in a considerably easier way by treating them through simple instructions.

# INTRODUCCIÓN

L

OS lenguajes de programación de alto nivel como lo son Java o C++ facilitan mucho a los programadores su labor de programar, ya que proveen una gran cantidad de herramientas, constituidas por código ya escrito, que al final terminan hacienda el código personal mucho más corto y comprensible. A continuación se tratarán dos problemas (que realizados en código de alto nivel resultarían bastante sencillos), con instrucciones de código ensamblador, es decir, código entendible al procesador. Esto con el fin de entender cómo funcionan muchos de los comandos de código que se usan hoy en día en lenguajes de programación de alto nivel.

# PUNTO 1.

Para el primer punto de este parcial, se debía crear un código en lenguaje ensamblador para el procesador de arquitectura MIPS, el cual recibía un nombre y un apellido y se debía retornar el mismo nombre y apellido pero con la diferencia que cada cadena debía estar alternando cada carácter entre mayúscula y minúscula y también se debía retornar el valor de la función XOR entre la primera letra del nombre y la primera del apellido.

Ahora bien, al hablar del diseño previo para realizar este punto, se tomó en cuenta que por facilidad se debía recibir cada cadena por separado, así pues las operaciones serian mucha más sencilla de realizar y con mayor control sobre los resultados. También, se tuvo en cuenta la cantidad de caracteres que podrían entrar en cada campo, así que se tomó un valor predeterminado de 15 máximos de caracteres, pues según ciertas estadísticas se tiene que el promedio de los caracteres por campo ronda entre 8 y 12. Por último, se tuvo en cuenta que el usuario podría ingresar cualquiera de las cadenas en una combinación de mayúsculas y minúsculas, por lo cual se decidió realizar a cada campo una función que antes de ser procesadas, transformara toda la cadena a minúsculas. A continuación se encuentra el código utilizado para satisfacer la necesidad del algoritmo:

.data

buffer: .space 15

nom: .asciiz "Ingrese nombre: "

ape: .asciiz "Ingrese apellido: "

str: .ascii " "

nombre: .space 15

apellido: .space 15

fraFin: .ascii "El resultado es: "

todoFin: .space 45

xorFin: .ascii "El XOR entra la primera letra del Nombre y el Apellido es: "

.text

#-----------------------NOMBRE-------------------------

li $v0, 4

la $a0, nom

syscall

li $v0, 8

la $a0, buffer

la $a1, 10

move $s0, $a0

syscall

#Copiado a .data

la $t0, nombre

miniloop:

lb $t2, ($s0)

beqz $t2, finLoop

sb $t2, ($t0)

add $s0, $s0, 1

add $t0, $t0, 1

j miniloop

finLoop:

#------------TODO A MINUS-----------

la $t0, nombre

aMinus:

lb $t2, ($t0)

beqz $t2, fin1

li $t1, 'A'

bge $t2, $t1, pruebaMayus

j iter

pruebaMayus:

li $t1, 'Z'

ble $t2, $t1, esMayus

iter:

addi $t0, $t0, 1

j aMinus

esMayus:

addi $t2, $t2, 32

sb $t2, ($t0)

j iter

fin1:

#----------CAMBIO DE LETRAS--------------

la $s0, nombre #palabra

li $t0, 0 #contador letras

li $t1, 0 #contador loop

contarLetras:

lb $t2, 0($s0) #char actual

beqz $t2, preloop

add $s0, $s0, 1

add $t0, $t0, 1

j contarLetras

preloop:

sub $s0, $s0, $t0

sub $s0, $s0, 1

loop:

add $t3, $t1, $zero

li $t4, 2 #divisor

div $t3, $t4

mfhi $t5 #residuo

beq $t5, 0, aMayus

add $t1, $t1, 1

blt $t1, $t0, loop

j fin2

aMayus:

add $s0, $s0, $t1

lb $t2, 0($s0)

subi $t2, $t2, 32

sb $t2, ($s0)

sub $s0, $s0, $t1

add $t1, $t1, 1

j loop

fin2:

#----------------------APELLIDO-------------------------------

li $v0, 4

la $a0, ape

syscall

li $v0, 8

la $a0, buffer

la $a1, 10

move $s0, $a0

syscall

#Copiado a .data

la $t0, apellido

miniloop2:

lb $t2, ($s0)

beqz $t2, finLoop2

sb $t2, ($t0)

add $s0, $s0, 1

add $t0, $t0, 1

j miniloop2

finLoop2:

#------------TODO A MINUS-----------

la $t0, apellido

aMinus2:

lb $t2, ($t0)

beqz $t2, fin3

li $t1, 'A'

bge $t2, $t1, pruebaMayus2

j iter2

pruebaMayus2:

li $t1, 'Z'

ble $t2, $t1, esMayus2

iter2:

addi $t0, $t0, 1

j aMinus2

esMayus2:

addi $t2, $t2, 32

sb $t2, ($t0)

j iter2

fin3:

#------------------CAMBIO DE LETRAS--------------------

la $s0, apellido #palabra

li $t0, 0 #contador letras

li $t1, 0 #contador loop

contarLetras2:

lb $t2, 0($s0) #char actual

beqz $t2, preloop2

add $s0, $s0, 1

add $t0, $t0, 1

j contarLetras2

preloop2:

sub $s0, $s0, $t0

sub $s0, $s0, 1

loop2:

add $t3, $t1, $zero

li $t4, 2 #divisor

div $t3, $t4

mfhi $t5 #residuo

beq $t5, 0, aMayus2

add $t1, $t1, 1

blt $t1, $t0, loop2

j fin4

aMayus2:

add $s0, $s0, $t1

lb $t2, 0($s0)

subi $t2, $t2, 32

sb $t2, ($s0)

sub $s0, $s0, $t1

add $t1, $t1, 1

j loop2

fin4:

#--------------------IMPRESION FINAL----------------------

li $v0, 4

la $a0, fraFin

syscall

la $s0, todoFin

la $s1, nombre

la $s2, str

la $s3, apellido

copiaNombre:

lb $t0, ($s1)

beqz $t0, precopia

sb $t0, ($s0)

addi $s0, $s0, 1

addi $s1, $s1, 1

j copiaNombre

precopia:

sub $s0, $s0, 1

copiaStr:

lb $t0, ($s2)

beqz $t0, copiaApellido

sb $t0, ($s0)

addi $s0, $s0, 1

addi $s2, $s2, 1

j copiaStr

copiaApellido:

lb $t0, ($s3)

beqz $t0, fin5

sb $t0, ($s0)

addi $s0, $s0, 1

addi $s3, $s3, 1

j copiaApellido

fin5:

li $v0, 4

la $a0, todoFin

syscall

#-------------------------XOR------------------------------

la $s0, nombre

la $s1, apellido

lb $t0, 0($s0)

lb $t1, 0($s1)

xor $t2, $t0, $t1

li $v0, 4

la $a0, xorFin

syscall

li $v0, 1

la $a0, ($t2)

syscall

#----------------------------FIN------------------------------

li $v0, 10

syscall

Dentro de la sentencia “.data” se hace la separación de memoria para las constantes que se van a tener a lo largo de la ejecución de programa. “buffer” va a ser utilizado cada vez que se le pida al usuario datos de tipo cadena de caracteres, pues esta permite reservar memoria para los datos próximos a entrar. “nom”, “ape”, “str”, “fraFin”, “xorFin”, son cadenas que van a tener un valor predeterminado y que van a ser utilizadas como encabezados y conectores entre las cadenas ingresadas por el usuario. Por último, “nombre”, “apellido”, “todoFin”, son espacios de memoria asignados para los datos ingresados por el usuario que serán cambiados a lo largo de la ejecución del programa.

Para una mayor facilidad de desarrollo y entendimiento del código para terceros, se realizó una separación por secciones y bloques de sentencias que tienen en común el dato sobre el cual se está trabajando actualmente, en otras palabras, dentro de la sección “#-..-NOMBRE-..-“, se encontrara todo lo relacionado con el ingreso de datos por parte del usuario, su cambio a minúsculas y el procesamiento de la cadena para realizar la alternación de mayúsculas dentro de la cadena. Mas específicamente, se tienen 2 bloques posteriores a la separación, los cuales son los encargados de imprimir en pantalla la instrucción para el usuario y recibir la cadena de caracteres entregada por el usuario correspondiente al nombre. Posteriormente, sigue un comentario que marca el inicio del bloque que carga lo que se recibió del usuario al espacio de memoria anteriormente mencionado y de nombre “nombre”, este bloque realiza un ciclo que itera a través de cada carácter ingresado y es guardado en cada posición directamente proporcional del espacio reservado. Más tarde encontramos el comentario que marca el inicio de la función que revisa el estado de carácter y de estar en mayúscula, lo cambia automáticamente a minúscula. Este bloque consta de un ciclo que itera por cada carácter de la cadena, y realiza dos condicionales por cada iteración, el primero revisa si ya se ha llegado al final de la cadena lo cual llevara a que se continúe con el flujo del programa y el segundo condicional es divido en dos, así se revisa que este en el rango ASCII entre ‘A’ y ‘Z’ para saber si es mayúscula y de ser positivo lo anterior, se le es sumado 32 y guardado a su valor en ASCII para volverlo minúscula. Más tarde, se encuentra el comentario que marca el comienzo de la sección que realiza el cambio alternado de los caracteres a mayúscula y minúscula, para comenzar se tiene el bloque que asigna todas las variables iniciales para ser usadas posteriormente. Luego, se tiene un bloque que realiza un ciclo corto para realizar el conteo total de los caracteres dentro de la palabra el cual continua con una sentencia previa al ciclo final el cual reinicia la posición de la variable que contiene a la palabra, de esta manera puede ser utilizada una vez más. Finalmente, los últimos dos bloques son un gran ciclo el cual en cada iteración divide el contador actual del ciclo en dos y revisa su residuo para saber si la posición del carácter es par o impar y de esta manera saber si se le debe restar 32 a su valor en ASCII y volver mayúscula. Al terminar este gran bloque se continúa con la sección del apellido, la cual realiza exactamente las mismas operaciones que con el nombre pero posiblemente con valores diferentes.

Después se encuentra la sección de la impresión del resultado final de cada cadena y la sección que realiza la operación XOR con el primer carácter de cada palabra y luego imprime el resultado en pantalla.

# PUNTO 2.

En este punto, se debe crear en código ensamblador “MIPS” un algoritmo que reciba por consola un número entero, y que a partir de ese número calcule tal valor de la serie de Fibonacci, mostrando todos los valores anteriores en pantalla, y validando que el número ingresado por consola sea mayor o igual a cero y menor a treinta. Considerando que el programa requiere verificar algunas condiciones para validar el número entrado, y que hay ciertas partes del código que deben repetirse algunas veces para hacer la suma de los números, se hace necesario el uso de algunas etiquetas en el código. Se deben cargar algunos valores por defecto al principio del código para referenciar lo que se imprime y para comparar el número ingresado. A continuación se encuentra el código utilizado para satisfacer la necesidad del algoritmo:

.data

nom: .asciiz "Ingrese numero: "

res: .asciiz "Resultado: "

finStr: .asciiz "\n" #Cambio de linea

.text

comienzo:

li $s5, 30

li $v0, 4

la $a0, nom

syscall

li $v0, 5

syscall

add $s1, $v0, $zero

add $s2, $s5, $zero

bgez $s1,menor #Evalua que el número sea positivo

j comienzo

menor:

sub $s3,$s2,$s1 #Evalua que el número sea menor a 30

bgtz $s3,fibonacci

j comienzo

fibonacci:

add $s3, $s1, $zero

add $s1, $zero, $zero #Establece valores iniciales

li $s2, 1

beqz, $s3,fin

j ciclo

ciclo:

li $v0, 1

add $a0, $s1, $zero

syscall

sub $s3, $s3, 1

add $s4, $s1, $s2

add $s1, $s2, $zero #Realiza un ciclo para sumar

add $s2, $s4, $zero

li $v0, 4

la $a0, finStr

syscall

bgtz $s3,ciclo

beqz $s3,fin

fin:

li $v0, 4

la $a0, res

syscall

li $v0, 1

add $a0, $s1, $zero

syscall

li $v0, 10 #Fin del programa

syscall

Al principio del código definimos las cadenas “nom”, “finStr” y “res” las cuales se utilizan más adelante para imprimir los resultados. Luego nos encontramos con la etiqueta comienzo, en donde cargamos en una variable el valor 30; luego imprimimos la cadena “Ingrese número” y esperamos a que entre un número por consola. Una vez se ingresa el número, este se mueve a la variable s1, y luego a la variable s2 damos el valor constante de 30 cargados previamente. Se evalúa que el número ingresado sea positivo, si no es así se volverá a ejecutar desde la etiqueta comienzo. Si el número es positivo se pasa a la etiqueta menor, en donde se resta a 30 el número ingresado, para verificar si este es menor que 30. Si el número no es menor que 30 se vuelve a ejecutar todo desde “comienzo”, si el número es menor que 30 se pasará a la etiqueta fibonacci. En la etiqueta fibonacci se establecen los valores iniciales de 0 a s1 y de 1 a s2, además de mover a s3 el numero ingresado por el usuario; si el número ingresado es 0 se finalizará la ejecución del programa, imprimiendo el resultado de 0; si el número es mayor a 0 se pasará a la etiqueta ciclo. En la etiqueta ciclo imprimimos el valor de s1, luego restamos una unidad al entero guardado en s3, guardamos en s4 la suma entre s1 y s2, movemos s2 a s1 y guardamos en s2 el valor de s4, imprimimos un salto de linea y nos queda guardado en s0 el valor de la sucesión para la primera iteración. El ciclo se repite hasta que s3, a quién cada vez se le resta 1 llegue a ser 0, en este caso se salta a la etiqueta fin, en donde se imprime “El resultado es: “seguido del valor guardado en s0, y luego el programa finaliza su ejecución. De esta manera se satisfice el enunciado del algoritmo y se hace posible hallar todos los valores de la serie de Fibonacci desde el valor 0 hasta el valor 29.

# CONCLUSIONES

* El manejo de código de ensamblador requiere un análisis considerablemente más amplio que el que requiere un lenguaje de más alto nivel para programar.
* En código de ensamblador se le dan instrucciones directamente al procesador o a la memoria para reservar espacio para nuestros datos, por lo tanto nos encontramos con un espacio limitado que debemos saber controlar.
* El código en lenguaje Ensamblador suele resultar varias veces más amplio que un código con la misma funcionalidad en un lenguaje de programación de alto nivel.
* Al manejar tan detalladamente las direcciones de memoria y las pocas variables con las que se cuenta se debe tener cuidado con no interferir con valores de retorno a funciones o con datos para llamadas al sistema al momento de guardar nuestros datos.
* Al manejar sentencias tan repetitivas, estas pueden ser mejoradas con el uso de funciones o macros para evitar la inconsistencia, redundancia y la dificultad de lectura del código. Lastimosamente por errores que aún no comprendemos, no pudimos hacer uso de alguna de esas estructuras.

References

1. Sanderson, Pete and Vollmar, Kennethr, Mars Mips Assembler Help Documentation, 2003.

1. [↑](#footnote-ref-1)