



Laboratorio 3: Programación en ensamblador MIPS

Planteamiento

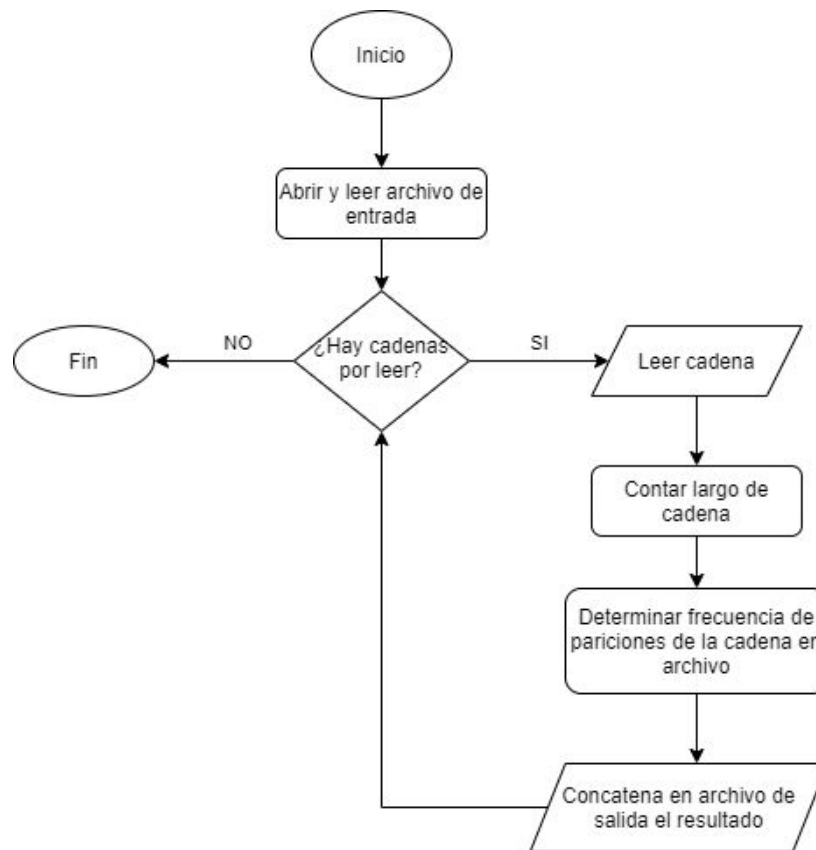
A continuación se encontrará de forma detallada la decisiones de diseño e implementaciones de los diferentes procedimientos que permiten la funcionalidad un un programa escrito en lenguaje ensamblador MIPS, que estará en capacidad de determinar la frecuencia de aparición de diferentes cadenas de caracteres en un archivo de entrada, el resultado estará copiado en un archivo de salida, ambos en texto plano.

Objetivos

- Estudiar la arquitectura del conjunto de instrucciones MIPS de 32 bits.
- Diseñar, codificar, ensamblar, simular y depurar programas escritos en lenguaje ensamblador MIPS.
- Familiarizarse con el uso de un entorno de desarrollo de software de bajo nivel.

Desarrollo

DIAGRAMA: FUNCIONAMIENTO BÁSICO DEL PROGRAMA



PROCEDIMIENTOS UTILIZADOS EN EL PROGRAMA

Procedimientos para el manejo del archivo de entrada:

```
abrirArchivoEntrada:
    # Abrimos archivo para lectura
    li $v0, 13
    la $a0, file_input
    li $a1, 0
    li $a2, 0
    syscall
    move $s0, $v0 # Guardamos el file descriptor
    jr $ra
```

El procedimiento **abrirArchivoEntrada** utiliza el **syscall** con el código 13 para el cual se necesita el string con el nombre del archivo que termine en nulo (**.asciiz**), que en este caso denominamos **file_input**, estará alojado en la sección del programa de datos (**.data**). El **syscall** nos devuelve en **\$v0** el descriptor del archivo.

```
leerArchivoEntrada:
    # Leemos el archivo de a 100 bytes en el buffer
    li    $v0, 14
    move  $a0, $s0
    la    $a1, buffer
    li    $a2, 100
    syscall
    move  $a2, $v0 # Guardamos cantidad de caracteres leídos
    jr    $ra
```

El procedimiento **leerArchivoEntrada** utiliza el **syscall** con el código 14 y el descriptor del archivo de entrada, además también se necesita el espacio de dirección de entrada del buffer con el que reservamos 100B para lectura, cuyo label será igualmente buffer, este estará alojado en la sección del programa de datos (**.data**). El **syscall** nos devuelve en **\$v0** el número de caracteres leídos.

Para nuestro programa será posible leer archivos de cualquier tamaño ya que se leerá de a 100 B en cada llamado al procedimiento (implementando sus funcionalidades) de manera sucesiva hasta que no hayan caracteres por leer, en otras palabras que **\$v0** devuelva cero.

```
cerrarArchivoEntrada:
    li    $v0, 16
    move  $a0, $s0
    syscall
    jr    $ra
```

El procedimiento **cerrarArchivoEntrada** utiliza el **syscall** con código 16 y el descriptor del archivo para finalizar el flujo de datos con el mismo.

Procedimiento para ¿Hay cadenas por leer?

En este punto del programa se tendrá la opción de cambiar (a través del código) el número de cadenas que se quieren buscar en el archivo.

En el registro de memoria **\$s6** se tendrá un especie de contador con el cual cada vez que lea una cadena y haga el ciclo completo con esta (busque la frecuencia de aparición en el archivo de entrada, y copie el resultado en el archivo de salida), le resta el entero 1, hasta convertirse en cero. Aquí se sabrá que no habrá más cadenas por leer, procesar y se finalizará el programa.

Procedimiento para determinar el largo de una cadena:

largoString:

Con este procedimiento se podrá calcular la cantidad de bytes ocupados por determinada cadena excluyendo caracteres que no aportan al resultado como lo son el salto de línea **'\n'** y el valor nulo **'\0'**. Para lograrlo se usan comparaciones **byte a byte** y con la ayuda de una variable contador retornar el valor deseado.

Pseudocódigo:

```
int largoString():
    cadena[]

    int contador = 0
    char salto = '\n'

    # Si llegamos a null significa que terminamos de
    leer el string.
    while( cadena[contador] != null ):

        # Si encontramos el salto de linea no contamos.
        if( cadena[contador] == salto ):
            continue
        end(if)

        contador++
    end(while)

    return contador
end(largoString)
```

Variable Alto Nivel	Variable MIPS	Definición
cadena[]	\$a0	Contiene la dirección donde comienza el contenido de la cadena.
contador	\$t0	Posee la cantidad de caracteres diferentes válidos leídos.

salto	\$t2	Almacena el carácter '\n' el cual debe ser omitido del conteo.
Auxiliar	\$t3	Es una variable que nos permite copiar el valor de cadena[] para poder movernos a través de ese espacio, dejando \$a0 intacto.

Codificación MIPS:

```

largoString:
    add $t0, $zero, $zero # inicializa $t1 sera contador = 0
    lb $t2, salto # Debemos omitir este caracter
    add $t3, $a0, $zero

Loop:    lb $t1, 0($t3) # cargamos byte inicial
        beq $t1, $zero, return
        beq $t1, $t2, suma # Si encontramos el salto de linea no contamos
        addi $t0, $t0, 1 # $t0 = $t0 + 1
suma:    addi $t3, $t3, 1 # $a0 = $a0 + 1 direccion byte siguiente
        j Loop

return:  add $v0, $t0, $zero # $v0 = contador
        jr $ra

```

Procedimiento para convertir de decimal a ASCII:

decToAscii:

Este procedimiento permite convertir un número en decimal a su equivalente en la codificación en ASCII simplemente sumándole 48. Hay que tener de cuenta que el número puede ser de varias cifras por lo tanto cuando guardemos los datos en memoria ya convertidos debemos empezar a guardar desde la última posición del vector para que el número no sea impreso al revés en el archivo.

Pseudocódigo:

```
void decToAscii():
    # El vector numero es un espacio en memoria que posee
    10 bytes
    # los cuales van a permitir alojar un numero de 10
    digitos como
    # maximo convertido en ascii respectivamente.
    numero[10]
    contador = 10
    num

    while( num < 0 ):
        # Obtenemos el valor de la cifra menos
        significativa.
        # y lo guardamos en el vector donde se guardara
        convertido
        # en ASCII sumandole 48.
        numero[contador] = (num % 10) + 48
        num = num / 10
        contador--
    end(while)

    # Devolvemos el la resta de 10 - cantidadDeCifras del
    numero
    # que nos sera util para escribir el numero en el
    archivo.
    return contador

end(decToAscii)
```

Variable Alto Nivel	Variable MIPS	Definición
num	\$a0	Contiene el número a convertir a ASCII.
contador	\$a1	Dirección donde se almacenará luego de la conversión.

Codificación MIPS:

```
decToAscii:
    move $t2, $a1
    addi $t2, $t2, 9
    add $t1, $zero, 10    #Contador de número de dígitos

Loopdec:
    bltz $a0, ReturnAscii # Salta si $a0 < 0
    div $a0, $a0, 10 # $a0 = $a0 / 10
    mfhi $t0 # $t0 guarda el residuo de la división
    addi $t0, $t0, 48 # $t0 = $t0 + 48 (Convierte a ASCII)
    sb $t0, 0($t2) # Guardamos $t0 en la memoria
    addi $t2, $t2, -1 # Nos movemos a la siguiente posición de memoria
    addi $t1, $t1, -1
    beq $a0, $zero, ReturnAscii
    j Loopdec
ReturnAscii:
    move $v0, $t1
    jr $ra
```

Procedimiento para determinar frecuencia de apariciones de cadena en el archivo:

contarFrecuencia:

Con este procedimiento se podrá leer de manera simultánea el contenido almacenado actualmente en el espacio reservado para la lectura del archivo de entrada (**buffer**), el cual puede traer como máximo 100B en cada llamado, con el contenido de la cadena. El proceso consta de comparar **byte a byte** y detectar los caracteres que pertenezcan a la cadena para su posterior identificación la cual se logra cuando el número de caracteres identificados de manera adyacente sea igual al largo de la cadena. Cabe resaltar que el archivo puede ser más grande que 100B por lo que la lectura del archivo se realizará en trozos de esta cantidad hasta terminar.

Pseudocódigo:

```
int contarFrecuencia():
    buffer[]
    cadena[]
    datosBuffer = len(buffer)
    datosCadena = len(cadena)
    contadorBuffer = 0
    contadorCadena = 0
    contadorRepeticiones = 0

    # Si datosBuffer es cero es porque ya terminamos de leer el archivo.
    if( datosBuffer == 0 ) {
        return contadorRepeticiones
    }

    # Bucle que itera sobre los datos obtenidos al leer el archivo.
    while ( contadorBuffer < datosBuffer):

        # Si encontramos un carácter que coincida con la cadena sumamos al
        # contadorCadena.
        if ( buffer[contadorBuffer] == cadena[contadorCadena] ):
            contadorCadena++
        else if( buffer[contadorBuffer] == cadena[0] ):
            contadorCadena = 1
        else:
            contadorCadena = 0
        end(if)

        # Si el contadorCadena es igual al largo de la cadena
        # quiere decir que encontramos la cadena entera
        if( contadorCadena == datosCadena ):
            contadorRepeticiones++
            contadorCadena = 0
        end(if)

        # Nos movemos al siguiente elemento del buffer
        contadorBuffer++
    end(while)

    # Volvemos a leer el archivo.
    datosBuffer = leerBuffer()

    return contadorRepeticiones
end(contarFrecuencia)
```


Variable Alto Nivel	Variable MIPS	Definición
buffer[]	\$a1	Contiene la dirección donde comienza el contenido del buffer.
cadena[]	\$a0	Contiene la dirección donde comienza el contenido de la cadena.
datosBuffer	\$a2	Contiene la cantidad de bytes que fueron leídos del archivo, es decir el retorno del syscall leerArchivoEntrada .
datosCadena	\$a3	Contiene la cantidad de bytes que corresponden a la longitud de la cadena ingresada, obtenido del procedimiento largoString .
contadorBuffer	\$t0	Variable que permite iterar en el contenido del buffer en un rango [0, n-1] donde n es el valor de datosBuffer.
contadorCadena	\$t1	Variable que contabiliza las apariciones de caracteres que pertenecen a la cadena de forma adyacente.
contadorRepeticiones	\$t2	Variable que contabiliza la cantidad de veces que aparece la cadena si es identificada dentro del texto que se está analizando.
Auxiliares	\$t3, \$t4, \$t5, \$t6 y \$t7	Este conjunto de variables serán utilizadas como auxiliares para el correcto funcionamiento del procedimiento, debido al riguroso tratamiento de MIPS con las direcciones muchos de estas variables contendrán una copia de otras variables o valores para operar.

Codificación MIPS:

(Imagen anexada en la siguiente página)

Nota: Hacemos uso de la pila para preservar intactos algunos registros de vital importancia para el buen procesamiento, ya que hacemos constantes llamados al procedimiento **leerArchivoEntrada** que actualiza los datos almacenados en el **buffer**.

```

contarFrecuencia:
    add $t2, $zero, $zero # $t2 sera el contadorRepeticiones
    add $t1, $zero, $zero # $t1 sera el contadorCadena
    add $t3, $a0, $zero # direccion de la cadena

    asignaFreq:
    add $t0, $zero, $zero # $t0 sera el contadorBuffer
    add $t6, $a1, $zero # direccion del buffer

    beq $a2, $zero, returnFreq # Si ya leimos todos los datos del archivo retornamos

loopFreq:
    bge $t0, $a2, endLoopFreq # Si contadorBuffer es mayor o igual a datosBuffer salimos.
    lb $t4, 0($t6) # Almacenamos en $t3 el valor de buffer[contadorBuffer]
    lb $t5, 0($t3) # Almacenamos en $t4 el valor de cadena[contadorCadena]

    iffFreq:
        bne $t4, $t5, elifFreq
        addi $t3, $t3, 1
        addi $t1, $t1, 1
        j endIfFreq
    elifFreq:
        lb $t7, 0($a0)
        bne $t4, $t7, elseFreq
        addi $t3, $a0, 1
        addi $t1, $zero, 1
        j endIfFreq
    elseFreq:
        add $t3, $a0, $zero
        add $t1, $zero, $zero
    endIfFreq:

    bne $t1, $a3, elseFreq2 # Aca viene el segundo if contadorCadena == datosCadena
    addi $t2, $t2, 1
    add $t3, $a0, $zero
    add $t1, $zero, $zero

    elseFreq2:
    addi $t6, $t6, 1 # contadorBuffer++
    addi $t0, $t0, 1
    j loopFreq

endLoopFreq:
    # Push
    addi $sp, $sp, -28
    sw $ra, 0($sp) # Se almacena el registro de retorno $ra en la pila
    sw $a0, 4($sp) # Se almacena direccionCadena
    sw $a1, 8($sp) # Se almacena direccionBuffer
    sw $a3, 12($sp) # Se almacena datosCadena
    sw $t2, 16($sp) # Se almacena contadorRepeticiones
    sw $t1, 20($sp) # Se almacena contadorCadena
    sw $t3, 24($sp)

    jal leerArchivoEntrada # Leemos otros 100B del buffer

    # Pop
    lw $t3, 24($sp)
    lw $t1, 20($sp)
    lw $t2, 16($sp)
    lw $a3, 12($sp)
    lw $a1, 8($sp)
    lw $a0, 4($sp)
    lw $ra, 0($sp)
    addi $sp, $sp, 28

    j asignaFreq

returnFreq:
    add $v0, $t2, $zero # $v0 = contador
    jr $ra

```

Procedimiento para concatenar en archivo de salida el resultado:

Para este procedimiento se realizó una macro en donde se ingresarán los parámetros de registro donde está **%descriptor**, **%buffer**, **%offset**, **%int**. El buffer es una etiqueta y el offset el desplazamiento.

```
.macro escribirarchivosalida(%descriptor, %buffer, %offset, %int)

    li $v0, 15
    move $a0, %descriptor
    la $a1, %buffer(%offset)
    move $a2, %int
    syscall

.end macro
```

Como salida en el archivo se pondrá:

CadenaABuscar : 67 repeticiones .

```
# Macro escribirArchivoSalida(descriptor, buffer, offset, cantidadCaracteres)
escribirarchivosalida( $s1, cadena_M0, $zero, $a3) # Escribimos la cadena
addi $t9, $zero, 3
escribirarchivosalida( $s1, separador, $zero, $t9) # Escribimos el separador

move $a0, $s5
la $a1, numero
jal decToAscii # Convertimos el numero resultado de contadorFrecuencia en ASCII

move $t8, $v0 # Guardamos el resultado del procedimiento decToAscii en $t8
addi $t9, $zero, 10
sub $t9, $t9, $t8

escribirarchivosalida( $s1, numero, $t8, $t9) # Escribimos el numero en ASCII
addi $t9, $zero, 16
escribirarchivosalida( $s1, repeticiones, $zero, $t9) # Escribimos el final
```

Prueba

Utilizamos un software para contar la frecuencia de las palabras **antonio**, **vanessa**, **freddy**. y esto nos arrojó:

Buscar	Reemplazar	Buscar en archivos	Marcar
Buscar: <input type="text" value="antonio"/>			
<input type="checkbox"/> Hacia atrás <input type="checkbox"/> Solo palabras completas <input type="checkbox"/> Coincidir MAYÚSCULAS/minúsculas <input checked="" type="checkbox"/> Buscar en todo el documento			
Modo de búsqueda <input checked="" type="radio"/> Normal <input type="radio"/> Extendido (\n, \r, \t, \0, \x...) <input type="radio"/> Expresión regular <input type="checkbox"/> . se ajusta a línea			
Contar: 6 coincidencias en todo el archivo			

Buscar	Reemplazar	Buscar en archivos	Marcar
Buscar: <input type="text" value="vanessa"/>			
<input type="checkbox"/> Hacia atrás <input type="checkbox"/> Solo palabras completas <input type="checkbox"/> Coincidir MAYÚSCULAS/minúsculas <input checked="" type="checkbox"/> Buscar en todo el documento			
Modo de búsqueda <input checked="" type="radio"/> Normal <input type="radio"/> Extendido (\n, \r, \t, \0, \x...) <input type="radio"/> Expresión regular <input type="checkbox"/> . se ajusta a línea			
Contar: 9 coincidencias en todo el archivo			

Buscar	Reemplazar	Buscar en archivos	Marcar
Buscar: <input type="text" value="freddy"/>			
<input type="checkbox"/> Hacia atrás <input type="checkbox"/> Solo palabras completas <input type="checkbox"/> Coincidir MAYÚSCULAS/minúsculas <input checked="" type="checkbox"/> Buscar en todo el documento			
Modo de búsqueda <input checked="" type="radio"/> Normal <input type="radio"/> Extendido (\n, \r, \t, \0, \x...) <input type="radio"/> Expresión regular <input type="checkbox"/> . se ajusta a línea			
Contar: 5 coincidencias en todo el archivo			

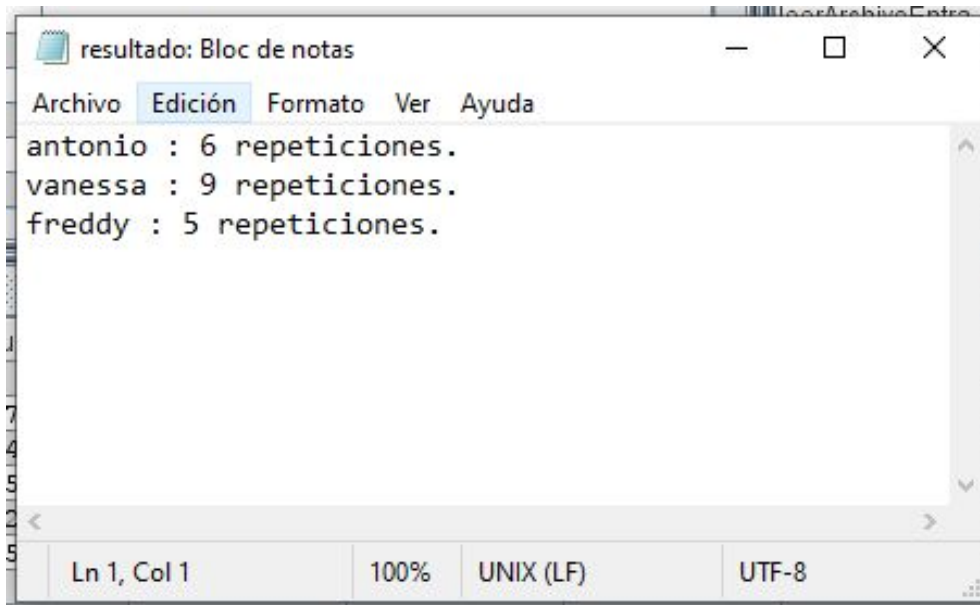
Utilizando el programa desarrollado:

Entrada
<input type="text" value="antonio"/>
Aceptar Cancelar

Entrada
<input type="text" value="vanessa"/>
Aceptar Cancelar

Entrada
<input type="text" value="freddy"/>
Aceptar Cancelar

En el archivo de salida sale:



```
resultado: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
antonio : 6 repeticiones.
vanessa : 9 repeticiones.
freddy : 5 repeticiones.
Ln 1, Col 1    100%    UNIX (LF)    UTF-8
```

Observaciones

- Es apreciable ver como una sola sentencia de alto nivel se puede desempaquetar en múltiples de bajo nivel, las cuales realizan instrucciones muy básicas pero que en conjunto forman una lógica mucho más grande.
- Como se está en una capa de abstracción muy baja se puede evidenciar que el programa corre mucho más rápido el código ya que son menos etapas para llegar al código máquina.
- A pesar de ser una arquitectura de conjunto de instrucciones muy básica, cuenta con mucha variedad de documentación, además de implementar .macros y syscall que son bastante importantes a la hora de escribir un programa.

Conclusiones

- A la hora de codificar, se nos olvida hacer nuestros algoritmos lo más óptimos posibles y sobre todo al principio llegamos a creer que no tiene mucha importancia, que sólo es el tiempo lo que se ve reflejado, pero ahora que nos llegamos a enfrentar directamente a los procedimientos más básicos y a un manejo de memoria más directo nos damos cuenta de la importancia de hacer una buena administración de todos los recursos del computador.
- Es raro enfrentarse al paradigma de programación estructurada después de todo, es algo fácil de ver en lenguajes de alto nivel. Este paradigma nos lleva a tener ciertas consideraciones a la hora de programar, como por ejemplo la ubicación de ciertas etiquetas, la estructura en general del algoritmo, la forma de hacer las preguntas, etc, ya que una mala ubicación de estos puede llevar a malos resultados.
- Reconocer el esfuerzo de todos aquellos que han puesto su conocimiento para hacer cada vez más amigable la sintaxis, poniendo capas y capas de abstracción que van desde lo más bajo hasta un nivel mucho más alto, cosa que no es tarea fácil.