Instituto Tecnológico de Costa Rica
Curso: IC-3101 Arquitectura de Computadores
I Semestre 2015

Tarea Programada #2:
Evaluador de expresiones matemáticas

Profesor:
Erick Hernández Bonilla

Alumnos:
Isaac Campos Mesén 2014004626
Melissa Molina Corrales 2013006074
Edwin Rees Sandí

Fecha de entrega:
18-05-2015

Índice

Propósito y Descripción

El objetivo de este proyecto es crear un software que permita evaluar expresiones matemáticas, con el fin de facilitar la manera en que se resuelven estas funciones al usuario, este software deberá leer una expresión matemática expresada con incógnitas, junto con los valores de esas incógnitas, para así poder sustituir los valores en la expresión matemática leída y mostrar paso a paso su resolución.

$2x + (6 - y) * 3x - ((x*z / 8) + 25)$,
$x = 50$,
$y = 5$,
$z = 20$

Figura 1. Expresión matemática junto con las incógnitas.

a. $2x + (6 - y) * 3x - ((x*z / 8) + 25)$
b. $(2x) + (6 - y) * (3x) - (((x*z) / 8) + 25)$
c. $(2*50) + (6 - 5) * (3*50) - (((50*20) / 8) + 25)$
d. $100 + 1 * 150 - ((1000 / 8) + 25)$
e. $100 + 1 * 150 - (125 + 25)$
f. $101 * 150 - 150$
g. $101 * 150 - 150$
h. $15150 - 150$
i. 15000

Figura 2. Sustitución y resolución paso a paso de la expresión matemática.

Este software será programado en YASM (Ensamblador) en la sintaxis de Intel x64.

El lenguaje ensamblador es el lenguaje de programación utilizado para escribir programas informáticos de bajo nivel, y constituye la representación más directa del Código máquina para cada arquitectura de computadoras legible por un programador.

Buffers a utilizar para el almacenamiento de los datos

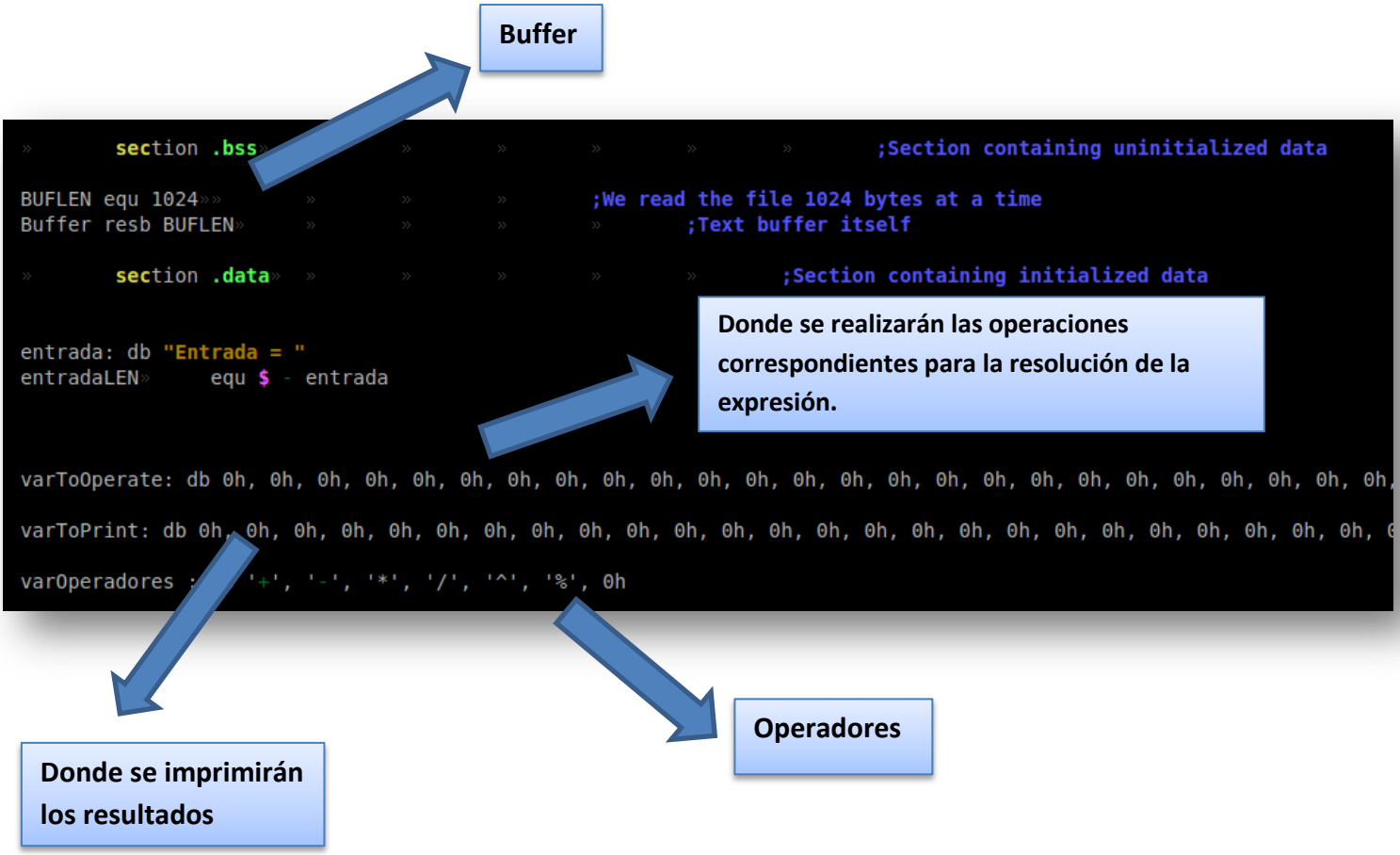
Para el almacenamiento de los datos, se define un buffer en donde leemos el archivo de 1024 bytes a la vez.

Para recibir la expresión matemática se define un buffer, llamado **entrada**, en esta se almacenará la expresión matemática junto con las variables a sustituir en la expresión.

Para almacenar la expresión matemática leída y realizar sobre esta las operaciones correspondientes, se utilizará el buffer **varToOperate**.

Para ir almacenando el resultado de las operaciones que se vayan realizando, se utilizará el buffer **varToPrint**, la cual también nos permitirá ir imprimiendo el resultado en pantalla. Asimismo para almacenar los operadores y poderlos utilizar en la resolución de la expresión matemática se usará el buffer **varOperadores**.

Importante mencionar que los buffers **varToOperate** y **varToPrint**, comenzarán con caracteres nulos.



Procesamiento de la entrada

Al decir procesamiento de la entrada, se refiere a la manera en que se almacenará la expresión que el software recibirá, una vez haya hecho las validaciones correspondientes. Para esto se utilizará un procedimiento llamado **procesarEntrada**, el cual recorre byte por byte la entrada hasta la primera coma, esto porque después de la coma siguen las variables a sustituir en la expresión y lo que necesitamos almacenar es solo la expresión para trabajar en ella, así este procedimiento almacena la expresión en el buffer **varToOperate.**

```
procesarEntrada:
        xor r14, r14                    ;indice
        .nextChar:
                ;Get a char from the buffer and put it in RAX
                mov al, byte [rsi + rcx]
                mov byte[varToOperate+r14], al
                inc r14
                inc rcx
                cmp byte[rsi + rcx], ','
                        jnz .nextChar
ret
```

Figura 3. Procedimiento para procesar la entrada

```
Entrada = a*b-23a+1
a*b-23*a+1
```

Figura 4. Forma en que la expresión es almacenada en varToOperate.

Validación de la expresión matemática ingresada

Antes de resolver la expresión se debe evaluar el estado de esta para comprobar que no se vaya a cometer algún error en la entrada.

Algunas validaciones:

- Validación que compruebe que la cantidad de variables ingresadas no sea superior a 20.

```
contarCantidadVariables:
    push r13
    xor r13,r13
   nextVariable:
»       mov al, byte [rsi + rcx]
»       call isVariable»
»       inc rcx
»       cmp r10, 1
»         jz nextVariable
»       inc r13
»       cmp byte [rsi + rcx],0h
»         jnz nextVariable
    pop r13
ret
```

```
validarCantidadVariables:
    call contarCantidadVariables
    mov rsi,varErrorCantidadVariables
    mov rdx,CantidadVariablesLEN
    cmp r13,20
      ja printError
ret
```

Validación de cantidad de variables ingresadas.

- Validación que compruebe que todas las variables a sustituir en la expresión existen.

```
validarVariablesExistentes:
»       .nextChar:
»       »       mov al, byte [rsi + rcx]
»       »       call isVariable»»       »
»       »       cmp r10, 1
»       »       »       jz validarVariables»

»       »       .continuar
»       »       inc rcx
»       »       cmp byte[rsi + rcx], 0h»»
»       »       »       jnz .nextChar
ret
»       »       »
```

```
validarVariables:
»       call buscarVariable»    »       »       »
»       cmp rdx, -1»    »       »       »       »
»       »       jz validarVariables.continuar»  »
»       mov rsi,varErrorVariableNoExistente
»       mov rdx,VariableNoExistenteLEN
»       jmp printError»
»
jmp validarVariables

buscarVariable:
»       push rbx
»       xor rdx, rdx
»       .nextChar:
»       »       mov bl, byte [varToOperate + rdx]
»       »       cmp al, bl
»       »       »       jz .exit
»       »       inc rdx
»       »       cmp byte[rsi + rdx], 0h»»       »
»       »       »       jnz .nextChar»  »       »
»       »       mov rdx, -1

»       .exit:
»         mov rdx,0
      pop rbx
ret
```

Validación de variables existentes.

En el caso de la operación multiplicación hay que tomar en cuenta, que esta puede estar tácita por ejemplo:

$$2x = 2 * x$$

Y en el caso de las variables que estén unidas **xy** la multiplicación se hace explicita **x * y**, sino esta se tomaría como una sola variable, para poder solucionar este problema se plantea un procedimiento el cual recorrerá el buffer **varToOperate**, donde estará almacenada la expresión y pondrá un asterisco en el lugar correspondiente.

```
ponerAsteriscos:
»       push rcx
»       xor rcx, rcx
»       .nextChar:
»       »       mov al, byte [varToOperate + rcx]»
»       »       call isOperador
»       »       cmp r10, 0»     »       »       »
»       »       »       jz validarPonerAsteriscos

»       »       .continuar:»    »       »       »
»       »       inc rcx»»       »       »       »
»       »       cmp byte [varToOperate + rcx], 0h»
»       »       »       jnz .nextChar
»       pop rcx
ret
```

Figura 5. Procedimiento para poner asteriscos en el caso de la operación multiplicación.

Como complemento a este procedimiento se utilizarán otros procedimientos: **isOperador**, el cual identifica si el carácter que está en **varToOperate** es un operador.

```
isOperador:
        push rcx

        xor rcx, rcx
        nextSymbol:
                cmp al, byte [varOperadores + rcx]
                        jz .isSymbol

                inc rcx
                cmp byte[varOperadores + rcx], 10
                        jnz nextSymbol

        .isNotSymbol:
                xor r10, r10
                jmp .exit

        .isSymbol:
                mov r10, 1

        .exit
                pop rcx
ret
```

Figura 6. Procedimiento para poner identificar si un carácter es un operador.

validarPonerAsteriscos, el cual se encarga de validar los casos en donde no es necesario poner asteriscos en la expresión y en caso de que se necesiten, este los agregará a la expresión matemática en el lugar correspondiente.

```
validarPonerAsteriscos:
        push r8
        push rax

        cmp al, '('
                jz .exit

        cmp al, 0h
                jz .exit

        call isDigit
        mov r8, r10
        mov al, byte [varToOperate + rcx+1]
        call isDigit
        and r8, r10
        cmp r8, 1
                jz .exit

        call isOperador
        cmp r10, 1
                jz .exit

        cmp al, 0h
                jz .exit
        cmp al, ')'
                jz .exit

        cmp al, ','
                jz .exit

        ;se procede a poner el asterisco
        mov rdx, rcx
        call liberarByteVarToOperate
        inc rcx
        mov byte [varToOperate + rcx], '*'

        .exit:
                pop rax
                pop r8
jmp ponerAsteriscos.continuar
```

Figura 7. Procedimiento validarPonerAsteriscos

Manejo de errores

Como se validan las entradas para comprobar que estas estén correctas, en el caso de que ocurra algún error en la entrada o en el cálculo de una operación, deberá lanzarse un error, los cuales se manejarán por medio de un procedimiento llamado **printError**, al cual se le pasará el mensaje de error correspondiente de acuerdo al error que se presente en el momento, se tendrán definidas variables con el error correspondiente y en los procedimientos en los casos que de error se llamará al procedimiento para que este lance el error.

```
printError:
»       push rax
»       push rdi
»       push rsi
»       push rdx
»       push rcx

»       mov rax, 1»     »
»       mov rdi, 1»     »
»       mov rsi,rsi»    »
»       mov rdx,rdx»    »
»       syscall»»       »

»       pop rcx
»       pop rdx
»       pop rsi
»       pop rdi
»       pop rax
ret
```

```
varErrorCalculo: db 'Error de calculo',10
ErrorCalculoLEN»equ $ - varErrorCalculo

dividir:
»       push rdx

»       xor rdx, rdx
»       div rbx ;RDX:RAX / RBX
»       cmp rbx,0
»         mov rsi,varErrorCalculo
»         mov rdx, ErrorCalculoLEN
»         jz printError
»       pop rdx
»       js ponerSigno
ret
```

```
varErrorCantidadVariables: db 'Error hay más de 20 variables',10
CantidadVariablesLEN»   equ $ - varErrorCantidadVariables
```

Manejo de errores.

Sustitución de variables

Para la sustitución de variables en la expresión matemática, se planteó de la siguiente manera:

Se utilizará un índice **(RCX)**, para recorrer el buffer de entrada después de la coma, y así se realizará una comparación con el buffer **varToOperate**, de esta manera se irá recorriendo y comparando las variables del buffer de entrada con las variables del buffer **varToOperate**, así en donde estas coincidan se sustituirá por su valor en la expresión.

Para su implementación se plantearon diversos procedimientos:

El procedimiento **cambiarVariables**, el cual irá recorriendo el buffer de entrada donde están las variables, hasta que haya llegado al fin de línea y ya no hayan variables por recorrer.

```
cambiarVariables:
        .nextChar:
                mov al, byte [rsi + rcx]
                call isVariable
                cmp r10, 1
                        jz cambiarVariable

                .continuar
                inc rcx
                cmp byte[rsi + rcx], 0h
                        jnz .nextChar
ret
```

Figura 8. Procedimiento cambiarVariables

Como complemento a este procedimiento se utilizará el procedimiento **isVariable**, el cual se encargará de identificar si un carácter en el buffer es una variable.

```
isVariable:
»       xor r10, r10

»       validarMenorAA:
»       »       cmp al, 'A'»    »
»       »       »       jb exit»»

»       validarMayorAz:
»       »       cmp al, 'z'»    »
»       »       »       ja exit»»

»       validarMenorAZ
»       »       cmp al, 'Z'»    »
»       »       »       jbe isVar»
»       validarMayorAA:
»       »       cmp al, 'A'»    »
»       »       »       jae isVar»
»       .exit:
»       ret

»       isVar:
»       inc r10
ret
```

Figura 9. Procedimiento isVariable

El procedimiento **cambiarVariable**, recorrerá el buffer **varToOperate** y sustituirá las variables con el valor correspondiente.

```
cambiarVariable:
»       call getPosVar» »       »       »
»       cmp rdx, -1»    »       »       »
»       »       jz cambiarVariables.continuar»

»       call getPosValue»       »       »
»       call movValue»  »       »       »
jmp cambiarVariable
```

Figura 10. Procedimiento cambiarVariable

Para obtener la posición en el buffer de entrada de la variable a sustituir en el buffer **varToOperate** se implementa el procedimiento **getPosVar**, este buscará la variable y en el caso de que esta no este retornará -1.

```
getPosVar:
        push rbx

        xor rdx, rdx
        .nextChar:
                mov bl, byte [varToOperate + rdx]
                cmp al, bl
                        jz .exit
                inc rdx
                cmp byte[rsi + rdx], 0h
                        jnz .nextChar
                mov rdx, -1

        .exit
        pop rbx
ret
```

Figura 11. Procedimiento getPosVar

Asimismo para obtener el valor de las variables a sustituir se usará el procedimiento **getPosValue**.

```
getPosValue:
        push rax
        mov r8, rcx
        .nextChar:

                mov al, byte [rsi + r8]
                call isDigit
                cmp r10, 1
                        jz .exit
                inc r8
                mov al, byte[rsi + r8]
                call isVariable
                        jnz .nextChar

        .exit:
        pop rax
ret
```

Figura 12. Procedimiento getPosValue

Operaciones

Para la implementación de las operaciones se manejará de la siguiente forma:

Se crean procedimientos para realizar las operaciones básicas suma, resta, multiplicación y división, además de las operaciones de módulo y exponente.

```
sumar:
	add rax, rbx
	js ponerSigno
ret
```

Figura 13. Procedimiento para sumar.

```
restar:
	cmp r8, 1
		jnz .op
	neg rbx
	.op
	add rax, rbx
	js ponerSigno
ret
```

Figura 14. Procedimiento para restar.

```
multiplicar:
	push rdx

	xor rdx, rdx
	imul rbx

	pop rdx
	js ponerSigno
ret
```

Figura 15. Procedimiento para multiplicar.

```
dividir:
	push rdx

	xor rdx, rdx
	div rbx ;RDX:RAX / RBX
	pop rdx
	js ponerSigno
ret
```

Figura 16. Procedimiento para dividir.

```
pow
	cmp rbx, 0
		jz casoPow0
	cmp rbx, 1
		jz casoPow1

	dec rbx
	mov rcx, rbx
	push rdx
	powloop:
		xor rdx,rdx
		imul rbx
		dec rcx
		jnz powloop
	pop rdx
	js ponerSigno

ret
```

Figura 17. Procedimiento para realizar la operación de exponente.

```
mod:
	push rdx

	xor rdx, rdx
	div rbx ;RDX:RAX / RBX
	mov rax, rdx
	pop rdx
	js ponerSigno

ret
```

Figura 16. Procedimiento para realizar la operación de módulo.

Para el manejo de los números negativos en las operaciones se implementó el procedimiento **ponerSigno**, el cual si el resultado de la operación es con signo hace complemento a 2 y pone el signo de menos en la variable, donde **neg** hace el complemento a 2.

```
ponerSigno:
»       neg rax
»       mov byte [varToPrint+r15], '-'
»       inc r15
ret
```

Figura 18. Procedimiento para ponerle signo al resultado de una operación.

Manejo de strings e integers

Una parte importante en la realización de este software es el manejo de los strings y de los integers, ya que al recibir la entrada, cada carácter ingresado la consola lo estará manejando como string y por tanto en el momento de almacenamiento necesitamos que los números ingresados se conviertan a integer para poder realizar operaciones sobre ellos para resolver la expresión matemática ingresada y poder mostrar el resultado en pantalla como string.

Para solucionar este problema se implementó dos algoritmos uno que convierte de string a integer y otro que convertirá el resultado de integer a string

Atoi

Este procedimiento se encarga de convertir todo lo que sea string a integer.

```
atoi:
        push rbx
        push r10
        push r11

        mov r11, r14

        xor rbx, rbx
        next_digit:

                mov al, byte [varToOperate + r14]
                sub al, '0'
                imul rbx, 10
                add rbx, rax
                inc r14
                mov al, byte[varToOperate + r14]
                call isDigit
                cmp r10, 1
                        jz next_digit

        cmp r11, 0
                jz .exit

        cmp r12, 1
                jz .exit
```

```
        cmp r12, 1
                jz .exit

        .validarNegarNumero:

                cmp byte [varToOperate + r11-1], '-'
                        jnz .exit
                neg rbx
                mov r10, 1
        .exit:
        mov rax, rbx

        pop r11
        pop r10
        pop rbx
ret
```

Figura 19. Atoi

Itoa

Este procedimiento se encarga de convertir todo lo que sea integer a string para poder mostrar el resultado en la pantalla.

```
xor rcx, rcx ;clear the rcx to 0;contiene el numero de digitos
xor rsi, rsi ;clear the rsi to 0

itoa_1:
        cmp rax, 0
                je itoa_2
        xor rdx, rdx
        mov rbx, 10
        div rbx
        push rdx
        inc rcx
        jmp itoa_1

itoa_2:
        cmp cx, 0
                ja itoa_3       ;if CX > 0
        mov ax, '0'                             ;En est
        mov byte [varToPrint+r15], al   ;no guarda valores en e
        ;put the char into varIntToString result
        inc rsi
        inc rcx
        jmp salirItoa

itoa_3:
        pop rax
        add rax, '0'
```

```
        itoa_3:
                pop rax
                add rax, '0'
                mov byte[varToPrint+r15+rsi], al
                inc rsi
                cmp rsi, rcx
                        je salirItoa
                jmp itoa_3
                salirItoa:

        pop rsi
        pop rdx
        pop rbx
ret
```

Figura 20. Itoa

Resolución de la expresión matemática

Para la resolución de la expresión una vez implementados todos los procedimientos anteriores el algoritmo se efectuará de la siguiente forma:

1. En el registro **(R9)** , se pasará la operación que se desee efectuar, así ira iterando **varToOperate** hasta el final de este buffer e irá haciendo las operaciones en este orden:
    - División
    - Multiplicación
    - Suma
    - Resta
2. Esto estará funcionando como un ciclo, dentro del cual se estará llamando al procedimiento **quitarParentesis**, el funcionamiento de este ciclo sería de la siguiente forma:
    - Limpia el buffer **varToPrint**, que se utilizará para guardar el resultado de las operaciones y la que nos permitirá imprimir en pantalla.
    - Busca la operación a realizar.
    - Realiza la operación.
    - Ajusta la variable **varToPrint**, es decir se le agregan los bytes del buffer **varToOperate** siguientes a la operación.
    - Se quitan los paréntesis innecesarios
    - Se imprime lo que hay en **varToPrint**
    - Se pasan todos los datos que este en **varToPrint** a **varToOperate**

    Este ciclo se irá repitiendo por cada operador que haya en la expresión ingresada hasta que ya no quede ninguna operación por resolver.

Obtener Operandos

Para poder realizar las operaciones básicas e ir resolviendo paso a paso la expresión, se necesita obtener los operandos y saber el operador para poder realizar la operación deseada. Para esto se implementan procedimientos que nos permiten obtener los operandos y dependiendo del operador llamar a los procedimientos de las operaciones para efectuar dicha operación.

Lo que se llevará a cabo para obtener los operandos es un procedimiento donde podamos recorrer la variable **varToOperate** y obtener el inicio del primer operando y el inicio del segundo operando, con el fin de guardar estos operandos en los registros **RAX** y **RBX** para luego pasarle estos operandos a los procedimientos de las operaciones.

¿Cómo hacemos esto?

Bien creamos un procedimiento llamado **buscarOperacion**, el cual busca el inicio del primer operando y va incrementándose hasta encontrarse con un operador para luego llamar a otro procedimiento **ObtenerOperandos** y decrementar el índice hasta encontrar el inicio del primer operando.

```
buscarOperacion:
        xor r14, r14
        xor rax, rax
        cmp r11, 0
                jz .exit
        .nextOperando:
                mov al, byte[varToOperate +r14]
                call addCharVarToPrint
                cmp al, r9b
                        jz ObtenerOperandos
                .continuar:
                inc r14
                cmp byte[varToOperate + r14], 0h
                        jnz .nextOperando
        .exit:
        mov r10, -1
ret
```

Figura 21. buscarOperacion

```
ObtenerOperandos:
        cmp byte[varToOperate +r14+1], '-'
                jz IraInicio

        mov al, byte[varToOperate +r14+1]
        call isDigit
        cmp r10, 0
                jz buscarOperacion.continuar

        mov al, byte[varToOperate +r14-1]
        call isDigit
        cmp r10, 0
                jz buscarOperacion.continuar
```

Figura 22. ObtenerOperandos

Así una vez tengamos los operandos, el procedimiento **realizar Operación** determinará cual operación es la que se desea realizar y llamará al procedimiento correspondiente para resolver la operación.

```
realizarOperacion:
»
»       cmp r9b, '+'
»       »       jz sumar
»       »
»       cmp r9b, '-'
»       »       jz restar
»       »
»       cmp r9b, '*'
»       »       jz multiplicar
»       »
»       cmp r9b, '/'
»       »       jz dividir

»       cmp r9, '^'
»       »       jz pow

»       cmp r9b, '%'
»       »       jz mod
ret
```

Figura 23. realizarOperacion

Para finalmente obtener la resolución de la expresión matemática paso a paso mostrada en pantalla:

```
Entrada = 2*4(1+2)
2*4*(1+2)
2*4*(1+2)
2*4*(3)
2*4*3
8*3
24
```

Bibliografía

Orenga, M. A. (05 de 09 de 2011). *Progamacion en ensamblador (x86-64).* Obtenido de http://www.cartagena99.com/recursos/alumnos/temarios/M6.%20Programacion%20en%20ensamblador%20(x86-64).pdf