I. TEMA: PROGRAMACION EN ENSAMBLADOR – PROCEDIMIENTOS

II. OBJETIVO DE LA PRACTICA

Al finalizar la presente práctica, el estudiante:

1. Comprende y explica el mecanismo de llamada a procedimientos en el lenguaje ensamblador de Linux para microprocesadores de arquitectura x86.
2. Escribe programas en lenguaje Ensamblador para Linux utilizando procedimientos

III. TRABAJO PREPARATORIO.

Para un trabajo con mejores resultados, es imprescindible que el estudiante:

1. Revise los aspectos teóricos de la programación modular.
2. Conozca la arquitectura de los procesadores con arquitectura X86
3. Conozca el conjunto de instrucciones ensamblador de los microprocesadores con arquitectura x86.

IV. MATERIALES.

Para el desarrollo de la presente práctica es necesario contar con:

1. Computador con arquitectura x86.
2. Sistema operativo Linux instalado en el computador o en su defecto un liveCD o liveUSB con las herramientas de compilación y programación pre instalados.
3. Compilador Nasm
4. Librería io.mac

V. MARCO TEORICO

PROCEDIMIENTOS

El concepto de modularidad no es nuevo para nosotros, y las ventajas del uso de la misma en el desarrollo de programas tampoco. Sin embargo, en el caso del lenguaje de programación ensamblador, el uso de los procedimientos y funciones requiere de un mayor conocimiento de las particularidades del hardware del microprocesador y de la arquitectura del mismo; algo que es transparente al programador que utiliza lenguajes de más alto nivel, tales como el C, Pascal, Java o C# entre otros.

En la presente práctica veremos estas particularidades y aprenderemos a utilizar procedimientos para el desarrollo de programas en NASM para Linux

LLAMADA A PROCEDIMIENTOS

Para llamar a un procedimiento se utiliza la instrucción CALL. La sintaxis de esta instrucción se muestra a continuación:

CALL *nombreProc*	Operación: *Almacena el valor de PC en el tope de la pila y salta al procedimiento referenciado por* ***nombreProc*** *(Carga PC con la dirección de la primera instrucción del procedimiento llamado, la cual se obtiene a partir del nombre del mismo)*

Una instrucción CALL es equivalente a una instrucción de salto incondicional, en el sentido que altera el contenido del contador de programas (PC) sin depender de ninguna condición, pero a diferencia de las instrucciones de salto, realiza la tarea adicional de guardar el contador de programas en la pila del programa para hacer posible que, al terminar la ejecución del procedimiento, se pueda volver al programa principal.

A continuación, se muestra la estructura de un programa, que incluye un procedimiento. Observe que el procedimiento se escribe después del código de salida del programa principal:

```
section .text
        global   _start

_start:
        .
        .
        .
        call nombreProc           ;    llamar al procedimiento
nombreProc
        .
        .
        .
        ;final del programa, volver al sistema
        mov eax,1
        int 80h
```

```
;-------------------------------------------------
; Procedimiento nombreProc
;-------------------------------------------------
nombreProc:
        .       ;   instrucciones del procedimiento
        .
        .       ;   fin de procedimiento
```

RETORNO DE PROCEDIMIENTOS

Al terminar la ejecución de un procedimiento, el programa principal debe continuar desde la instrucción a continuación de la instrucción CALL. Puesto que la dirección de dicha instrucción se guardó en la memoria pila antes de la llamada al procedimiento, la recuperación de dicha dirección se hará desde la memoria pila.

La instrucción que realiza esta tarea es la instrucción RET, cuya sintaxis se indica a continuación:

RET N	Operación: *Restaura PC con la dirección de retorno almacenada en el tope de la pila (con ello reanuda la ejecución del programa principal). Cuando se indica N, al retornar al programa principal se retiran N bytes de la pila. N se indica para limpiar la pila de parámetros usados por el procedimiento.*

En esta instrucción no se especifica ningún parámetro, pues se asume que la dirección de retorno está en el tope de la pila.

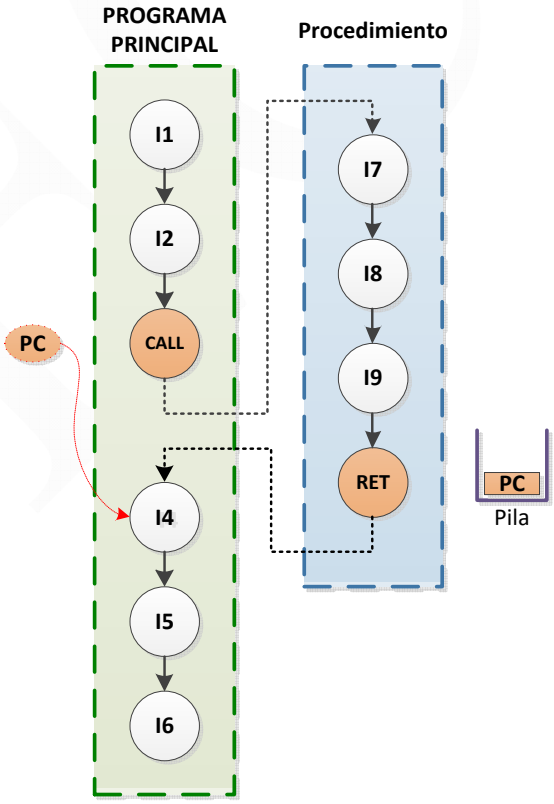
En el siguiente fragmento de programa se muestra la estructura de un programa que ilustra la llamada a un procedimiento, así como el uso de la instrucción de retorno de procedimientos:

```
section .text
        global   _start

_start:
        .
        .
        call nombreProc  ;   llamar al procedimiento nombreProc
        .                ;   punto de retorno del procedimiento
        .
        ;volver al sistema
        mov eax,1
        int 80h

;-------------------------------------------------
; Procedimiento suma
;-------------------------------------------------
nombreProc:
        .                    ;   instrucciones del procedimiento
        .
        .
        ret                  ;   volver del procedimientos
```

En el siguiente diagrama se muestra gráficamente el flujo de un programa que invoca un procedimiento. Cuando se alcanza la instrucción CALL, el PC apunta a la instrucción I4. Para no perder la referencia a dicha instrucción (que es la siguiente que debe ejecutarse al terminar de ejecutarse el procedimiento invocado por CALL), el contenido del PC se guarda en la memoria pila del programa, luego se carga el PC con la dirección del procedimiento (en el proceso de compilación, el nombre del procedimiento se convierte en la dirección de entrada al mismo).

Cuando concluye el procedimiento, la última instrucción en ejecutarse debe ser la instrucción RET, la cual toma un elemento del tope de la pila y carga con dicho valor el PC. Es responsabilidad del programador evitar que se almacene un valor diferente del PC en el tope de la pila, pues esto impediría que el procedimiento pueda volver al programa que lo invocó.

Al momento de ejecutarse la instrucción RET en el procedimiento, el PC apunta a la siguiente posición después de RET (la cual está fuera del espacio del procedimiento y probablemente fuera del espacio del programa principal, es decir, apunta a una posición de memoria no determinada y que podría pertenecer a otro programa). Puesto que RET modifica el PC con el valor almacenado en el tope de la pila, al finalizar la instrucción RET, el PC contendrá la dirección de I4, por lo tanto, cuando se extraiga la siguiente instrucción, esta será la instrucción I4, lo que significa que se ha retornado del procedimiento al programa principal y se continúa con la ejecución de la instrucción a continuación de la instrucción CALL en el programa principal. Justamente como se esperaba que funcionara una llamada a procedimiento



MARCO DE PILA O REGISTRO DE ACTIVACIÓN

Cuando un programa invoca un procedimiento mediante la instrucción CALL, la dirección de retorno se almacena en la memoria pila y es recuperada desde ahí por la instrucción RET para restablecer la dirección de retorno y volver al programa principal.

Si se desea utilizar la pila para pasar parámetros al procedimiento, los parámetros se deben almacenar en la memoria pila antes de llamar al procedimiento. Sin embargo, este mecanismo presenta un problema: al invocar la instrucción CALL, los datos en la memoria pila serán inaccesibles si nos limitamos a acceder a la memoria pila solo mediante las instrucciones PUSH y POP usando el registro ESP que siempre contiene la dirección del operando en el tope de la pila.

Para poder acceder a los parámetros se utiliza el registro EBP siguiendo los siguientes pasos:

1. Se guarda el valor de EBP en la pila.
2. Se hace una copia de ESP en EBP.
3. Se utiliza EBP para acceder a los parámetros almacenados en la pila.

Estas operaciones permiten proteger el valor de ESP, que apunta a la instrucción de retorno del procedimiento y al mismo tiempo acceder a los parámetros almacenados en la pila más allá del tope de pila.

En este caso, decimos que EBP apunta al registro de activación.

El registro de activación es el conjunto de datos asociados a la invocación al procedimiento, incluyendo los parámetros para la llamada, la dirección de retorno del procedimiento y el valor almacenado en pila de EBP.

Las operaciones descritas se pueden realizar con la secuencia de instrucciones:

```
push ebp
mov ebp, esp
```

o utilizarse la instrucción ENTER, cuya sintaxis describiremos luego.

Al terminar el procedimiento, se debe limpiar el registro de activación. Esto implica:

1. Copiar EBP en ESP para que ESP pueda ser utilizado para retornar al programa que invocó al procedimiento.
2. Restablecer el valor de EBP al valor que tenía antes de ser utilizado para acceder a los parámetros en la pila. Como resultado de esta operación ESP apunta a la dirección de retorno al programa que invocó al procedimiento.

Estas operaciones se consiguen con las instrucciones:

```
mov esp, ebp
pop ebp
```

Para realizar estas operaciones con mayor efectividad se implementó la instrucción LEAVE, la cual documentaremos a continuación.

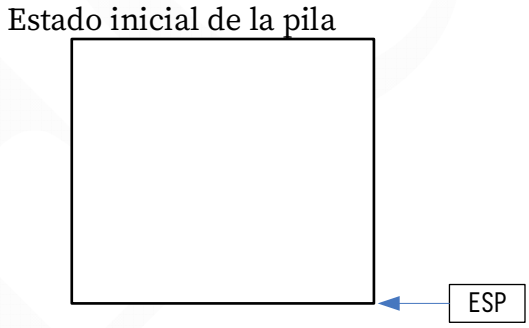
INSTRUCCIONES PARA EL MANEJO DE MARCO DE PILA

INSTRUCCIÓN ENTER:

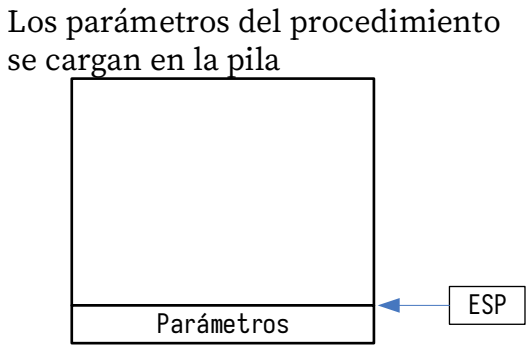
ENTER N, M	Operación: *Crea un marco de pila y reserva N bytes de espacio en memoria pila para variables locales. M indica el valor del atributo lexical nesting level (lexlevel) que se utiliza para indicar el número de apuntadores a marcos de pila que se almacenaran en la pila antes de establecer el apuntador de pila. Los apuntadores adicionales proporcionan al procedimiento invocado puntos de acceso a otros marcos anidados en la pila. El valor por defecto de M es 0.*
	
	El equivalente de la instrucción es:
	
	`push ebp`
	`mov ebp, esp`
	`sub esp, N`

La evolución del estado de la pila durante la ejecución de la instrucción ENTER se muestra en las siguientes figuras:

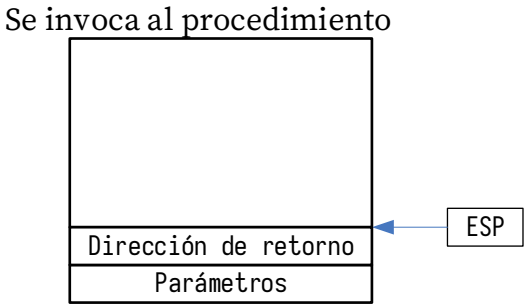
Estado inicial de la memoria pila:



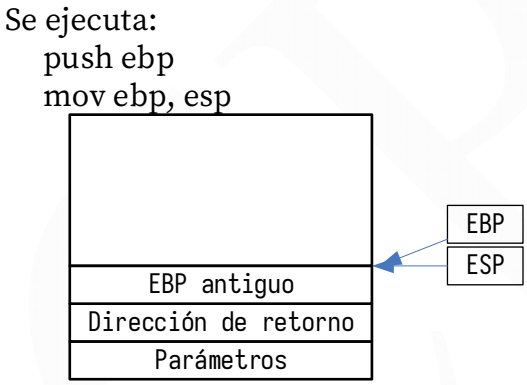
Se cargan los parámetros para invocar al procedimiento:



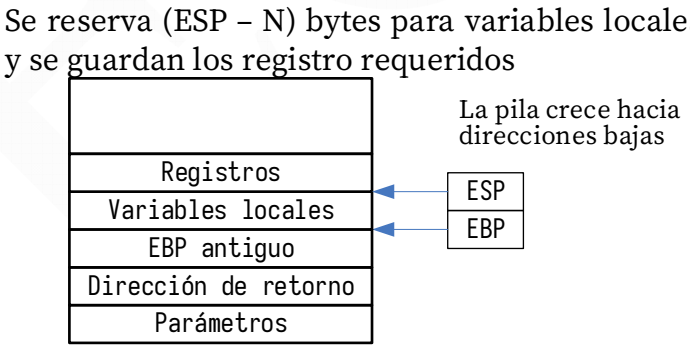
Se invoca al procedimiento con la instrucción CALL



Se ejecutan las instrucciones de creación del marco de pila:



Se reserva espacio para las variables locales y se almacenan los registros que sean necesarios o requeridos:



INSTRUCCIÓN LEAVE

LEAVE	Operación: *Libera el marco de pila creado por una instrucción ENTER. LEAVE copia el apuntador de marco (EBP) en el registro apuntador de pila (ESP) , lo que libera el espacio de pila asignado al marco de pila. El apuntador de marco previo se extrae de la pila al registro EBP, lo que restablece el marco de pila del programa que invoco el procedimiento.*
	
	El equivalente de esta instrucción es:
	
	`mov esp, ebp`
	`pop ebp`

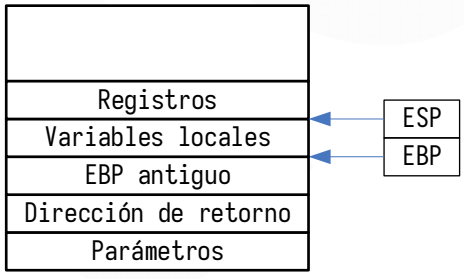
La evolución de la ejecución de la instrucción LEAVE se ilustra a continuación:

Iniciamos la descripción de este proceso, considerando que la instrucción LEAVE se invoca en correspondencia a la ejecución de la instrucción ENTER. Por lo tanto, asumimos que la memoria pila tiene el estado en el que quedo al finalizar la instrucción ENTER.

Si dentro del procedimiento se utilizó la memoria pila, esta debe restablecerse, para tal fin, se recomienda que a cada operación PUSH, le debe corresponder una operación POP.

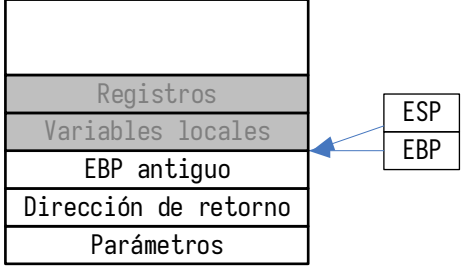
El estado de la memoria pila antes de invocar la instrucción LEAVE:

Estado de la memoria pila al terminar el procedimiento
y antes de invocar a la instrucción RET



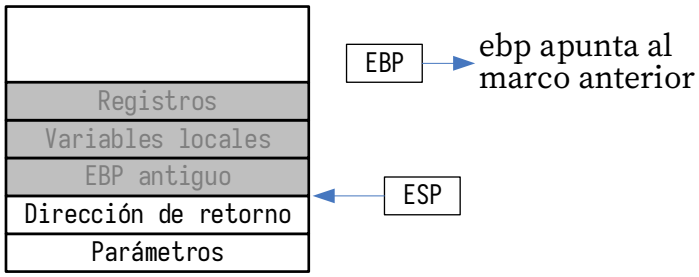
Se retiran las referencias a las variables locales:

Se ejecuta la instrucción mov esp, ebp



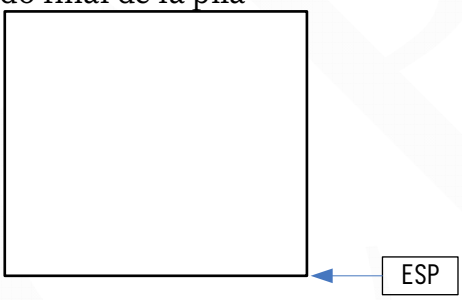
Se restablece la referencia al marco de pila anterior:

Se ejecuta la instrucción pop ebp



Cuando se ejecute la instrucción RET N para salir del procedimiento y volver al programa principal o al que invocó el procedimiento, el estado de la pila será:

Estado final de la pila



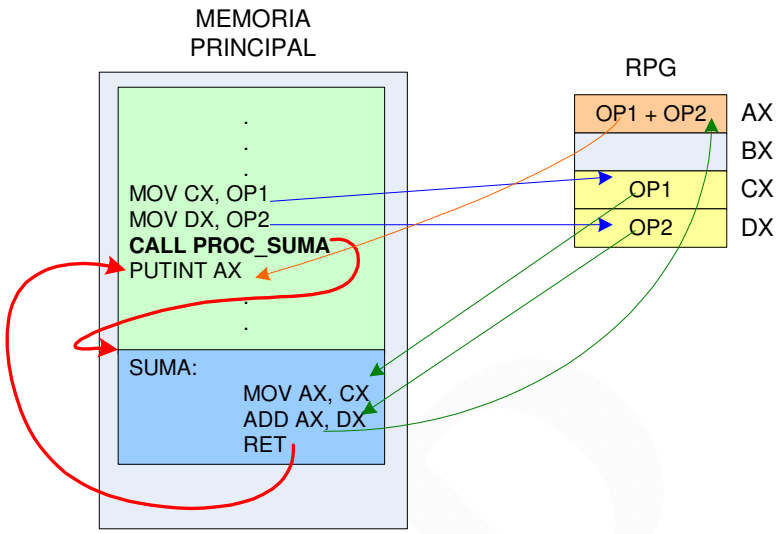
PASO DE PARAMETROS A PROCEDIMIENTOS

Si se requiere pasar uno o varios parámetros a un procedimiento, estos pueden pasarse a través de los registros de propósito general o a través de la pila. En los siguientes ejemplos se muestra los cuatro posibles mecanismos que se pueden utilizar para el paso de parámetros:

Ejemplo 1:

```
;Nombre    :    sumaValReg.asm
;Proposito:    calcula la suma de dos enteros ilustrando el paso de
               parámetros mediante llamada por valor utilizando
               registros
;Autor     :    Edwin Carrasco (adaptado de [4])
;FCreacion:    02/08/2008
;FModif.   :    ---
;Compilar :
;              nasm -f elf sumaValReg.asm
;              ld -m elf_i386 -s -o sumaValReg sumaValReg.o io.o

%include "io.mac"

section   .data
      msj_leer1:   db       “ingrese el primer numero: ",0
      msj_leer2:   db       “ingrese el segundo numero: ",0
      msj_mostrar: db       “La suma es: ",0

section   .text
      global   _start

_start:
      PutStr msj_leer1
      GetInt cx                        ;    cx = primer numero

      PutStr msj_leer2
      GetInt dx                        ;    dx = primer numero
      call suma                        ;    devuelve la suma en ax
      PutStr msj_mostrar
      PutInt ax
      nwln

      ;volver al sistema
      mov eax,1
      int 80h
;-------------------------------------------------
; Procedimiento suma
; int suma(int a, int b)
;-------------------------------------------------
suma:
      mov ax, cx                       ;    suma = primer numero
      add ax, dx                       ;    suma = suma + segundo numero
      ret
```

ILUSTRACIÓN DEL PROCEDIMIENTO



PRUEBAS DE FUNCIONAMIENTO

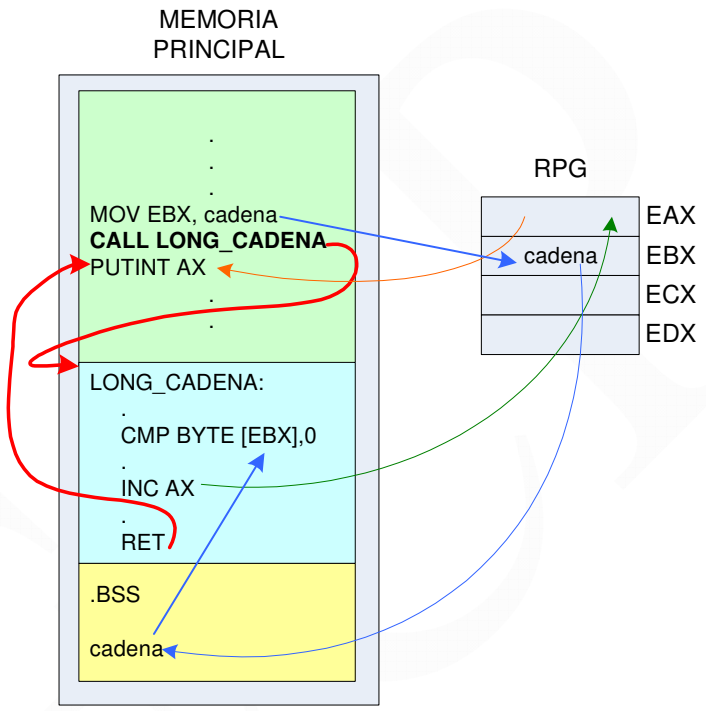
```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf sumaValReg.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s -o sumaValReg sumaValReg.o io.o
ecp@ecp-OAC:~/codigo_ASM$ ./sumaValReg
ingrese el primer numero: 9
ingrese el segundo numero: 13
La suma es: 22
```

Ejemplo 2:

```
;Nombre    :    longCadena.asm
;Proposito :    determina la longitud de una cadena ilustrando el paso
                de parámetros mediante llamada por referencia
                utilizando registros
;Autor     :    Edwin Carrasco (adaptado de [4])
;FCreacion :    02/08/2008
;FModif.   :    ---
;Compilar  :
;               nasm -f elf longCadena.asm
;               ld -s -o longCadena longCadena.o io.o


%include "io.mac"

BUF_LEN EQU 41      ;    longitud de buffer de cadena

section    .data
      msj_leer:    db       “ingrese una cadena: “,0
      msj_mostrar: db       “La longitud de la cadena es: “,0

section    .bss
      cadena resb  BUF_LEN

section    .text
      global   _start

_start:
      PutStr msj_leer
      GetStr cadena, BUF_LEN     ;    leer cadena de teclado

      mov ebx, cadena            ;    ebx = dirección de cadena
      call long_cadena           ;    devuelve longitud de la cadena
                                      en ax
      PutStr msj_mostrar         ;    mostrar longitud de cadena
      PutInt ax
      nwln

      ;volver al sistema
      mov eax,1
      int 80h

;-------------------------------------------------
; Procedimiento long_cadena
; int long_cadena (string cadena)
;-------------------------------------------------
long_cadena:
      push ebx
      xor ax,ax

repetir:
      cmp byte [ebx],0       ;    comparar con carácter NULL
```

```
        je terminado            ; si NULL, entoces terminamos
        inc ax                  ;    caso contrario, incrementar ax
        inc ebx                 ;    apuntar al siguiente carácter
        jmp repetir             ;    repetir el proceso

terminado:
        pop ebx
        ret
```

ILUSTRACIÓN DEL PROCEDIMIENTO



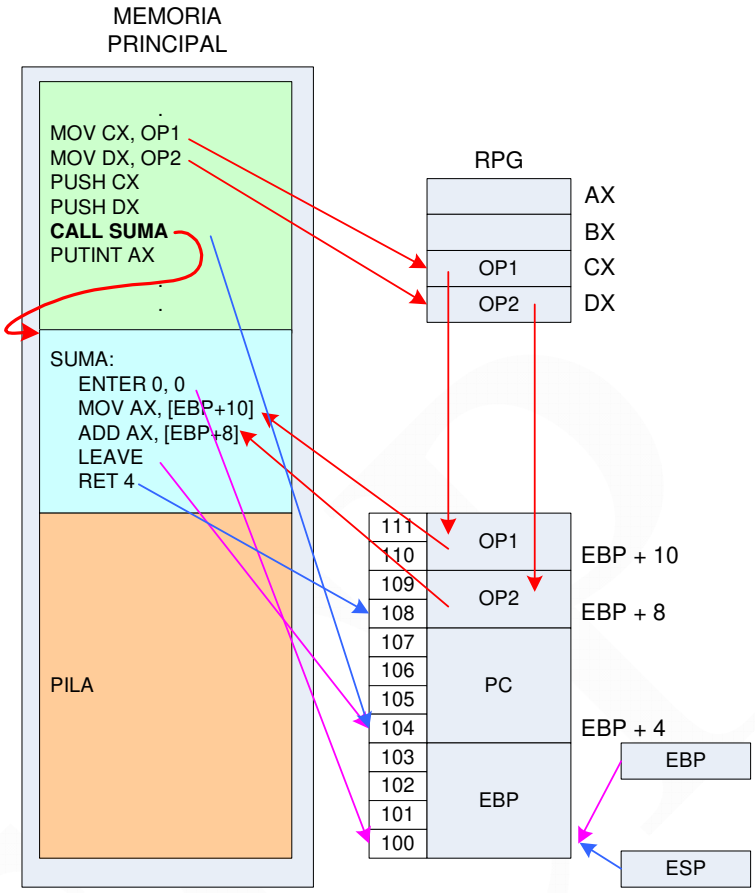
PRUEBAS DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf longCadena.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s -o longCadena longCadena.o io.o
ecp@ecp-OAC:~/codigo_ASM$ ./longCadena
ingrese una cadena: El mundo es ancho y ajeno
La longitud de la cadena es: 25
```

Ejemplo 3:

```
;Nombre    :    sumaValStack.asm
;Proposito :    suma dos enteros, ilustrando el paso de parámetros
                mediante llamada por valor utilizando la pila
;Autor     :    Edwin Carrasco (adaptado de [4])
;FCreacion :    02/08/2008
;FModif.   :    ---
;Compilar  :
;               nasm -f elf sumaValStack.asm
;               ld -s -o sumaValStack sumaValStack.o io.o

%include "io.mac"

section    .data
      msj_leer1:    db       “ingrese el primer numero: “,0
      msj_leer2:    db       “ingrese el segundo numero: “,0
      msj_mostrar:  db       “La suma es: “,0

section    .text
      global   _start

_start:
      PutStr msj_leer1
      GetInt cx              ;    cx = primer número

      PutStr msj_leer2
      GetInt dx              ;    dx = segundo número

      push cx                ;    primer numero a la pila
      push dx                ;    segundo numero a la pila

      call suma              ;    devuelve la suma en ax

      PutStr msj_mostrar
      PutInt ax
      nwln

      ;volver al sistema
      mov eax,1
      int 80h

;-------------------------------------------------
; Procedimiento suma
; int suma (int a, int b)
;-------------------------------------------------
suma:
      enter 0,0                ;    salvar ebp
      mov ax, [ebp + 10]       ;    suma = primer numero
      add ax, [ebp + 8]     ;    suma = suma + segundo numero
      leave                    ;    restaurar ebp
      ret 4                    ;    regresar y limpiar parámetros
```

ILUSTRACIÓN DEL PROCEDIMIENTO



PRUEBAS DE FUNCIONAMIENTO

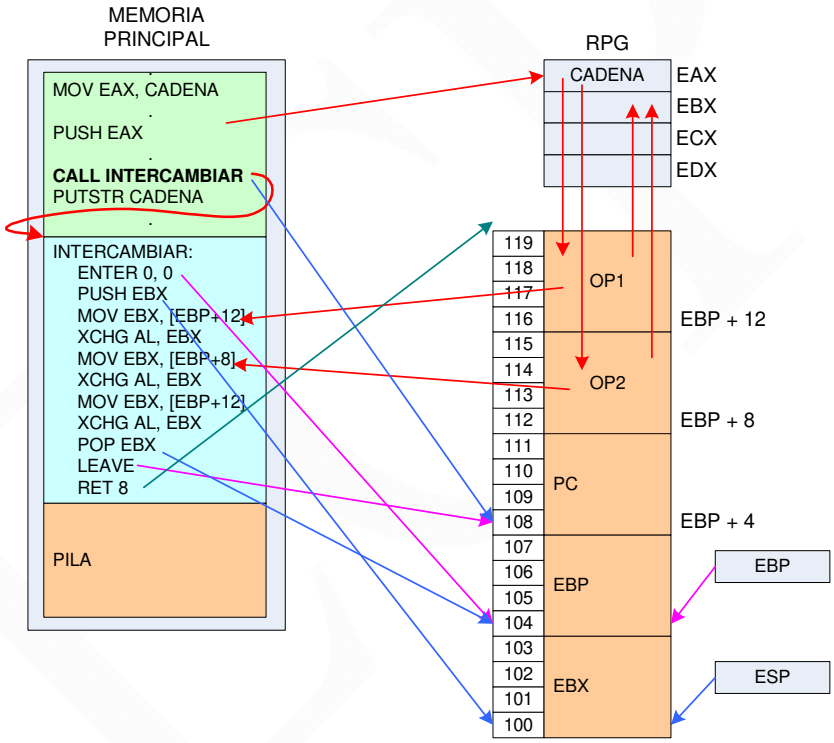
```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf sumaValStack.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s io.o sumaValStack.o -o sumaValStack
ecp@ecp-OAC:~/codigo_ASM$ ./sumaValStack
ingrese el primer numero: 34
ingrese el segundo numero: 76
La suma es: 110
```

Ejemplo 4:

```
;Nombre    :    swapCadena.asm
;Proposito :    intercambia los dos primeros caracteres de una cadena
                e ilustra el paso de parámetros mediante llamada por
                referencia usando la pila
;Autor     :    Edwin Carrasco (adaptado de [4])
;FCreacion :    02/08/2008
;FModif.   :    ---
;Compilar  :    nasm -f elf swapCadena.asm
;Enlazar   :    ld -m elf_i386 -s -o swapCadena swapCadena.o io.o

%include "io.mac"

BUF_LEN EQU 41      ;    longitud de buffer de cadena

section    .data
      msj_leer:     db       “ingrese una cadena: “,0
      msj_mostrar:  db       “La cadena invertida es: “,0

section    .bss
      cadena resb BUF_LEN

section    .text
      global   _start

_start:

      PutStr msj_leer
      GetStr cadena, BUF_LEN;    leer cadena

      mov eax, cadena        ;    eax = apuntador a cadena[0]
      push eax
      inc eax                ;    eax = apuntador a cadena[1]
      push eax
      call intercambiar ;    intercambiar las primeras 2
                             letras
      PutStr msj_mostrar     ;    mostrar la cadena intercambiada
      PutStr cadena
      nwln

      ;volver al sistema
      mov eax,1
      int 80h

;-------------------------------------------------
; Procedimiento intercambiar
; void intercambiar (char a, char b)
;-------------------------------------------------
intercambiar:
      enter 0,0              ;    salvar ebp
      push ebx               ;    salvar ebx

      ;    el intercambio comienza aquí. Debido a xchg, se preserva AL
```

```
mov ebx, [ebp + 12]    ;    ebx = apuntador al 1er carácter
xchg al, [ebx]
mov ebx, [ebp + 8]     ;    ebx = apuntador al 2do carácter
xchg al, [ebx]
mov ebx, [ebp + 12]    ;    ebx = apuntador al 1er carácter
xchg al, [ebx]

;    el intercambio termina aquí

pop ebx        ;    restaurar registros
leave
ret 8          ;    regresar y limpiar parámetros de la pila
```

ILUSTRACIÓN DEL PROCEDIMIENTO



PRUEBAS DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf swapCadena.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s io.o swapCadena.o -o swapCadena
ecp@ecp-OAC:~/codigo_ASM$ ./swapCadena
ingrese una cadena: solo
La cadena invertida es: oslo
```

RETORNO DE VALORES DESDE PROCEDIMIENTOS

Cuando se debe devolver un valor desde un procedimiento, puede hacerse utilizando los registros de propósito general o a través de la memoria pila

VI. TRABAJO DE LABORATORIO.

1. Escriba un programa modular que calcule el MCD de dos números enteros ingresados por el usuario

Solución

```
;Nombre      :    mcd.asm
;Proposito   :    calcula el mcd de dos numeros enteros
;Autor       :    Edwin Carrasco
;FCreacion   :    11/08/2008
;FModif.     :    ---
;compilacion:
;                 nasm -f elf mcd.asm
;                 ld -m elf_i386 io.o mcd.o -o mcd

%include "io.mac"

section .data
   mensaje: db "ESTE PROGRAMA CALCULA EL MCD DE DOS NUMEROS
ENTEROS",10,0
   pideNro1:    db "Ingrese el primer numero: ",0
   pideNro2:    db "Ingrese el primer numero: ",0
   salida:      db "El mcd es: ",0

section .text
    global  _start

_start:
   PutStr   mensaje      ;   indicar que hace el programa
   PutStr   pideNro1     ;   leer datos
   GetInt   ax           ;   datos en registro AX
   PutStr   pideNro2
   GetInt   bx

;--PROCESAR
bucle:                   ; while (bx != 0){
   cmp      bx, 0        ;
   jz       fin
   mov      cx, bx       ;       resto = bx
   call     modulo       ;       bx = ax % bx
   mov      ax, cx       ;       ax = resto
   jmp      bucle        ;   }

fin:
;--MOSTRAR RESULTADOS
   PutStr   salida
   PutInt   ax
   nwln

;--SALIR DEL PROGRAMA
   mov      eax,1
   int      80h
```

```
;-------------------------------------------------------------
;  Procedimiento modulo
;  int modulo (int A, int B){ return A % B;}
;-------------------------------------------------------------
modulo:
    xor     dx, dx              ; dx <-- 0
    div     bx                  ;    AX = Q = dx:ax / bx, R = dx
    mov     bx, dx
    ret
```

PRUEBAS DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf mcd.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 io.o mcd.o -o mcd
ecp@ecp-OAC:~/codigo_ASM$ ./mcd
ESTE PROGRAMA CALCULA EL MCD DE DOS NUMEROS ENTEROS
Ingrese el primer numero: 42
Ingrese el primer numero: 35
El mcd es: 7
```

2. Escriba un programa recursivo que calcule el factorial de un número entero.

Solución

```
;Nombre     :    factRec.asm
;Proposito  :    calcula el factorial de un numero
;Autor      :    Edwin Carrasco
;FCreacion  :    27/08/2008
;FModif.    :    ---
;compilacion:
;                nasm -f elf factRec.asm
;                ld -m elf_i386 io.o factRec.o -o factRec


%include "io.mac"

section .data
   mensaje: db "ESTE PROGRAMA CALCULA EL FACTORIAL DE UN
            NUMERO ENTERO",10,0
   pideNro: db "Ingrese el numero: ",0
   salida:  db "El factorial es: ",0

section .text
    global  _start

_start:

;--VARIABLES
       xor  eax, eax

;--INDICAR QUE HACE EL PROGRAMA
       PutStr   mensaje

;--LEER DATOS
       PutStr   pideNro       ;    leer datos
       GetLInt  eax           ;    datos en registro AX

;--PROCESAR
       push     eax           ;    pasar parametro a traves de la
                                    pila
       call     factorial     ;    el resultado se devuelve en
                                    eax

;--MOSTRAR RESULTADOS
       PutStr   salida
       PutLInt  eax
       nwln

;--SALIR DEL PROGRAMA
       mov eax,1
       int 80h
```

```
;-------------------------------------------------------------
;  Procedimiento factorial
;  int factorial (int A)
;-------------------------------------------------------------
factorial:
        enter 0,0
        mov     eax, [ebp + 8]    ;    eax <-- N
        cmp     eax, 1            ;    if (N == 1)
        je      fin_fact          ;        factorial = 1 //caso
                                  base
        dec     eax               ;    else
        push    eax               ;        factorial = N *
                                  factorial(N-1)
        call    factorial
        mov     ebx, [ebp + 8]    ;    ebx <-- N
        imul    ebx               ;    eax <-- factorial(N-1) *
                                  ebx

fin_fact:
        leave
        ret     4
```

PRUEBAS DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf factorial.asm
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 io.o factorial.o -o factorial
ecp@ecp-OAC:~/codigo_ASM$ ./factorial
ESTE PROGRAMA CALCULA EL FACTORIAL DE UN NUMERO ENTERO
Ingrese el numero: 7
El factorial es: 5040
```

VII. PRACTICAS DE LABORATORIO

1. Escriba un programa modular que calcule la suma de los elementos primos de un arreglo de 20 números enteros naturales. Utilice paso de parámetros por referencia.

2. Se dice que un número natural es un número feliz, si al sumar los cuadrados de sus dígitos obtenemos un número al que al aplicársele el mismo procedimiento una y otra vez converge en la unidad. Si se entra en un bucle que no incluye el 1, se dice que el número es infeliz (https://es.wikipedia.org/wiki/Número_feliz)

   Por ejemplo, el 7 es un número feliz porque:

   $7^2 = 49;$
   $4^2 + 9^2 = 97;$
   $9^2 + 7^2 = 130;$
   $1^2 + 3^2 + 0^2 = 10;$
   $1^2 + 0^2 = 1;$

   Escriba un módulo que determine si un número es feliz y utilícelo para mostrar los números felices encontrados en un arreglo de MxN. M, N y los elementos del arreglo se definen en tiempo de ejecución.

   El módulo devuelve un valor lógico (booleano) y recibe como parámetro por valor el entero que debe verificarse:

```
bool esFeliz (int Nro)
```

VIII. EVALUACION

La evaluación de las actividades realizadas en la presente guía de práctica se hará en función de la siguiente tabla:

ACTIVIDAD	Procedimental	
	Sesión 01	Sesión 02
Resolución del ejercicio propuesto 01	--	12
Resolución del ejercicio propuesto 02	--	08
TOTAL	**--**	**20**

IX. REFERENCIAS

1. Anvin P. http://nasm.sourceforge.net/. Sitio web del compilador NASM. (01/01/2012).
2. Bartlett Jonathan. "*Programming From The Ground Up*". Bartlett Publishing, 2003
3. Bendersky E. *"Stack frame layout on x86-64"*. https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/ (30/07/2019)
4. Brey Barry. "*Los Microprocesadores Intel. Arquitectura, Programación e Interfaces*". Prentice Hall 3Ed.
5. Dandamudi. "*Guide To Assembly Language Programming In Linux*". Springer 2005
6. Hyde Randall. "*Art of Assembly Language Programming*". Nostarch Press 1Ed.
7. Johnson P. "*Mixing Assembly And C*". http://courses.engr.illinois.edu/ece390/books/labmanual/c-prog-mixing.html. (01/01/2012)
8. Leto J. "*Writing a useful Program With NASM*". http://leto.net/writing/nasm.php. (01/01/2012)
9. Smith B. E., Johnson M. T. "*Programming The Intel 80386*" Editorial Scott, Foresman And Company, 1987.
10. Stack overflow. gcc x86-32 stack alignment and calling printf. https://stackoverflow.com/questions/52287214/gcc-x86-32-stack-alignment-and-calling-printf (04/08/2019)
11. Swanepoel D. http://docs.cs.up.ac.za/programming/asm/derick_tut/. Tutorial de ensamblador para Linux. (01/01/2012).
12. Toal R. http://www.cs.lmu.edu/~ray/notes/nasmexamples/. Ejemplos de código en ensamblador. (01/01/2012)
13. Toal R. *"NASM Tutorial"*. https://cs.lmu.edu/~ray/notes/nasmtutorial/ (30/07/2019)