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)*

Ejemplo	`CALL procCubo`
	`CALL procMCD`
	`CALL procFactorial`

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.

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	Operación: *Restaura PC con la dirección de retorno almacenada en el tope de la pila (con ello reanuda la ejecución del programa principal).*
Ejemplo	RET

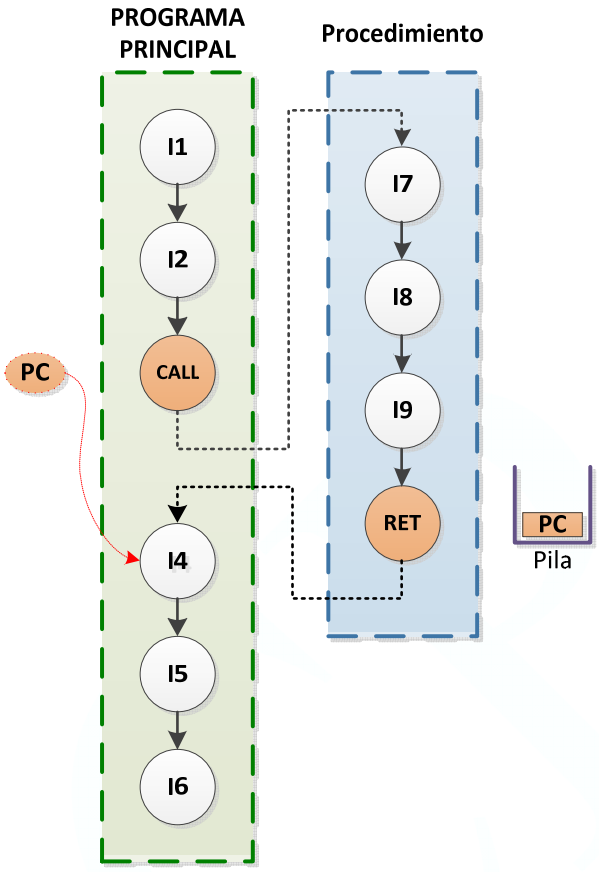
En esta instrucción no se especifica ningún parámetro, pues se asume que la dirección de retorno está en el tope de la pila.

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

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. Inicialmente nuestros programas utilizarán los registros de propósito general para el paso de parámetros a los procedimientos, así como para devolver valores desde los procedimientos.

VI. TRABAJO DE LABORATORIO.

1. Escriba un programa modular que calcule la potencia cuarta de un número entero N ingresado por el usuario.

Solución

```
;Nombre     :    potencia4.asm
;Proposito  :    calcula la potencia 4ta de un numero entero
;Autor      :    Edwin Carrasco
;FCreacion  :    02/29/2011
;FModific.  :    11/01/2023
;Compilar   :    nasm -f elf potencia.asm
;  Enlazar  :    ld -m elf_i386 -s -o potencia potencia.o io.o
;  Ejecutar :    ./potencia4

%include "io.mac"

section .data

   mensaje: db   "ESTE PROGRAMA CALCULA LA POTENCIA 4TA DE UN
                 NUMERO N",10,0
   pideN:   db   "INGRESE N: ",0
   salida:  db   "N^4 ES: ",0


section .text
    global  _start

_start:

   PutStr   mensaje  ;    indicar que hace el programa
   PutStr   pideN    ;    leer datos
   GetInt   ax       ;    datos en registro AX

   call potencia     ;    llamando al modulo potencia

   ;-- Mostrar resultados
   PutStr salida
   PutLInt eax
   nwln

   ;-- Salir del programa
   mov eax, 1
   int 80h

;-------------------------------------------------------------
;  Procedimiento potencia
;  long potencia (int A){return A*A*A*A}
;-------------------------------------------------------------

potencia:
   xor dx, dx    ;    dx <-- 0
```

```
    mov bx, ax
    mul bx          ;    ax <-- ax * ax
    mul bx          ;    ax <-- ax * ax * ax
    mul bx          ;    ax <-- ax * ax * ax * ax
    ret
```

PRUEBAS DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ #Compilar
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf potencia4.asm
ecp@ecp-OAC:~/codigo_ASM$ #Enlazar
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s -o potencia4 potencia4.o io.o
ecp@ecp-OAC:~/codigo_ASM$ #Ejecutar
ecp@ecp-OAC:~/codigo_ASM$ ./potencia4
ESTE PROGRAMA CALCULA LA POTENCIA 4TA DE UN NUMERO N
Ingrese N: 2
N^4 es: 16
ecp@ecp-OAC:~/codigo_ASM$ ./potencia4
ESTE PROGRAMA CALCULA LA POTENCIA 4TA DE UN NUMERO N
Ingrese N: 5
N^4 es: 625
```

2. Escriba un programa modular que muestre el mayor de tres números enteros ingresados por el usuario.

Solución

```
;Nombre     :    mayorDeTres.asm
;Proposito  :    Muestra el mayor de tres numeros
;Autor      :    Edwin Carrasco
;FCreacion  :    02/09/2011
;FModific.  :    11/01/2023
;Compilar   :    nasm -f elf mayorDeTres.asm
;  Enlazar  :    ld -m elf_i386 -s -o mayorDeTres mayorDeTres.o
                 io.o
;  Ejecutar :    ./mayorDeTres


%include "io.mac"    ;libreria de macros de entrada/salida

section .data
   proposito: db     "ESTE PROGRAMA MUESTRA EL MAYOR DE TRES
                     NUMEROS INGRESADOS POR EL USUARIO",10,0
   nro1:    db       "INGRESE EL PRIMER NUMERO: ",0,10
   nro2:    db       "INGRESE EL SEGUNDO NUMERO: ",0,10
   nro3:    db       "INGRESE EL TERCER NUMERO: ",0,10
   resultado:   db   "EL MAYOR ES: ",0,10

section .text

    global _start

_start:
    ;--inicializar registros
    xor ax, ax
    xor bx, bx
    xor cx, cx
    xor dx, dx

    ;--Indicar el proposito del programa
    PutStr proposito

    ;--Leer numeros
    PutStr nro1
    GetInt ax

    PutStr nro2
    GetInt bx

    PutStr nro3
    GetInt cx

    ;--Procesar
    call mayor
```

```
    mov bx, cx

    call mayor

    ;--Mostrar resultados
    PutStr resultado
    PutInt ax
    nwln

    ;--Salir del programa
    mov ax,1
    int 80h

    ;-------------------------------------------------------------
    ; Devuelve el mayor de dos numeros
    ;   int mayor(int A, int B)
    ;       { if (A>B) return A; else return B}
    ;-------------------------------------------------------------
mayor:
    ; Datos en ax y bx
    ; Devuelve mayor en ax
    cmp ax, bx
    js segundoEsMayor

    ret

segundoEsMayor:
    mov ax, bx

    ret
```

PRUEBA DE FUNCIONAMIENTO

```
ecp@ecp-OAC:~/codigo_ASM$ #Compilar
ecp@ecp-OAC:~/codigo_ASM$ nasm -f elf mayorDeTres.asm
ecp@ecp-OAC:~/codigo_ASM$ #Enlazar
ecp@ecp-OAC:~/codigo_ASM$ ld -m elf_i386 -s -o mayorDeTres mayorDeTres.o io.o
ecp@ecp-OAC:~/codigo_ASM$ #Ejecutar
ecp@ecp-OAC:~/codigo_ASM$ ./mayorDeTres
Este programa muestra el mayor de tres numeros ingresados por el usuario
Ingrese el primer numero: 34
Ingrese el segundo numero: 17
Ingrese el tercer numero: 55
El mayor es: 55
```

VII. PRACTICAS DE LABORATORIO

1. Escriba un programa modular que muestre el N ésimo número primo. Para este ejercicio, implemente un módulo que determine si un número es primo.

   El prototipo del módulo debe ser:

   ```
   bool esPrimo(int N)
   ```

2. Escriba un programa modular que determine los N primeros números amigos. N es ingresado por el usuario.

   El prototipo del módulo debe ser:

   ```
   bool sonAmigos(int a, int b)
   ```

   También puede implementar otro módulo que calcule la suma de los divisores propios de un número, cuyo prototipo sería:

   ```
   int sumaDivisores(int a)
   ```

3. Escriba un programa modular que muestre todos los números abundantes de un arreglo MxN. M y N se deben definir en tiempo de ejecución.

   Un número entero positivo se denomina abundante, si este es menor a la suma de sus divisores propios. Por ejemplo:

   12 : Es un número abundante, porque 12 < 1 + 2 + 3 + 4 + 6.
   15 : No es un número abundante, porque 15 > 1 + 3 + 5.

   El prototipo del módulo debe ser:

   ```
   bool esAbundante(int a)
   ```

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	--	04
Resolución del ejercicio propuesto 02	--	06
Resolución del ejercicio propuesto 02	--	10
TOTAL	**--**	**20**

IX. BIBLIOGRAFIA

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. Carrasco E. Sitio web de la asignatura http://dais.unsaac.edu.pe/~ecarrasco/
6. Dandamudi. "*Guide To Assembly Language Programming In Linux*". Springer 2005
7. Hyde Randall. "*Art of Assembly Language Programming*". Nostarch Press 1Ed.
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. Swanepoel D. http://docs.cs.up.ac.za/programming/asm/derick_tut/. Tutorial de ensamblador para Linux. (01/01/2012).
11. Toal R. http://www.cs.lmu.edu/~ray/notes/nasmexamples/. Ejemplos de código en ensamblador. (01/01/2012)
12. Toal R. *"NASM Tutorial"*. https://cs.lmu.edu/~ray/notes/nasmtutorial/ (30/07/2019)