Subrutinas

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE
v1.0.0	2018 Octubre 3		CA

Índice

1. Introducción

1.1. Objetivos

1.1.1. Programación

- Concepto de Subrutina en el Lenguaje Ensamblador AT&T x86-32
- Instrucciones de llamada y retorno: call y ret
- Argumentos: Utilización de la pila
- Instrucciones de pila: push y pop
- Estructura de la pila: punteros al bottom y top de la pila: registros EBP y ESP.
- Anidamiento de llamadas: segmentación de la pila en segmentos "Frame".
- Convenio de llamada: Pase de los parámetros, Valor de retorno, Dirección de retorno, Pila, Frame de la pila, Punteros al stack Frame, Epílogo, Prólogo
- Directivas : .type sumMtoN, @function

1.1.2. Análisis

- Análisis de la pila mediante el depurador GDB
  - observar la generación de un nuevo frame
  - identificar los límites del frame a través de los registros puntero.
  - volcar los argumentos, dirección de retorno y valor de retorno de la subrutina en la pila.

2. Módulo Fuente

- El módulo fuente *sumMtoN.s* realiza una llamada desde la rutina principal *_start* a la subrutina *sumMtoN* pasándole dos argumentos y recibiendo el resultado de la suma.
- La subrutina sumMtoN realiza la suma desde el número entero M hasta el número entero N donde N>M.

3. Requisitos

- Conceptos básicos de estructura de computadores.
- Arquitectura básica intel x86-32.
- Programación en lenguaje ensamblador AT&T: práctica con datos, modos de direccionamiento e instrucciones básicas de transferencia, aritméticas y de saltos.

4. LEEME

- Lectura del guión de prácticas y del capítulo 3 del Libro Programming from the Ground-Up.
- Apuntes y Libro de Texto
- Documentación Memoria: Contenido y Formato de la Memoria
- Evaluación: sistema de evaluación
- Plataforma de Desarrollo : configuración de la computadora personal
- Programación : metodología

5. Cuestiones

- "Autoevaluación de Prácticas" opcional: Prácticas: Cuestionario

6. Tamaño de los datos y variables

6.1. Algoritmo

- Desarrollar un programa en lenguaje ensamblador de la arquitectura *i386* que realice la suma $\sum_{i=1}^{N} i$ cuyo resultado es $N(N+1)/2$ utilizando el método de programación de descripción inicial en lenguaje pseudocódigo y organigrama. El programa debe de contener dos módulos: uno principal referenciado con el nombre *_start* y una subrutina denominada *sumMtoN* que realiza la suma. El programa principal pasa los paramétros M y N a la subrutina para una vez realizada la suma se devuelva el resultado de la suma como valor de retorno de la subrutina.

6.2. Edición del Módulo fuente: sumMtoN.s

- Descargar el módulo fuente "sumMtoN.s" de miaulario y añadir los comentarios apropiados.

```
        /*
Programa: sumMtoN.s
Descripción: realiza la suma de números enteros de la serie M,M+1,M+2,M+3,...N
        función : sumMtoN(1° arg=M, 2° arg=N) donde M < N
Ejecución:    Editar los valores M y N y compilar el programa.
        Ejecutar $./sumMtoN
        El resultado de la suma se captura del  sistema operativo con el comando linux:  ↩
            echo $?

        gcc -nostartfiles -m32 -g  -o sumMtoN sumMtoN.s
        Ensamblaje as --32 --gstabs sumMtoN.s -o sumMtoN.o
        linker -> ld -melf_i386    -o sumMtoN  sumMtoN.o
        */

        ## MACROS
        .equ    SYS_EXIT,       1
        ## DATOS
        .section .data

        ## INSTRUCCIONES
        .section .text
        .globl _start
_start:
        ## Paso los dos argumentos M y N a la subrutina a través de la pila
        pushl $10       #push    second argument -> N
        pushl $5        #push    first argument  -> M

        ## Llamada a la subrutina sum1toN
        call  sumMtoN

        ## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
        mov  %eax, %ebx   # (%ebx is returned)
        ## Código de la llamada al sistema operativo
        movl  $SYS_EXIT, %eax       # llamada exit
        ## Interrumpo al S.O.
        int   $0x80

/*
Subrutina: sumMtoN
Descripción: calcula la suma de números enteros en secuencia desde el 1° sumando hasta el  ↩
    2° sumando
        Argumentos de entrada: 1° sumando y 2° sumando
        los argumentos los pasa la rutina principal a través de la pila:
        1° se apila el último argumento y finalmente se apila el 1° argumento.
        Argumento de salida: es el resultado de la suma y se pasa a la rutina principal a  ↩
            través del registro EAX.
```

```
        Variables locales: se implementa una variable local en la pila pero no se utiliza
*/
        .type sumMtoN, @function # declara la etiqueta sumMtoN
sumMtoN:
        ## PróÍogo: Crea el nuevo frame del stack
        pushl %ebp           #salvar el frame pointer antiguo
        movl  %esp, %ebp     #actualizar el frame pointer nuevo
        ## Reserva una palabra en la pila como variable local
        ## Variable local en memoria externa: suma
        subl  $4, %esp
        ## Captura de argumentos
        movl  8(%ebp), %ebx  #1º argumento copiado en %ebx
        movl  12(%ebp), %ecx #2º argumento copiado en %ecx


        ## suma la secuencia entre el valor del 1ºarg y el valor del 2ºarg
        ## 1º arg < 2ºarg
        ## utilizo como variable local EDX en lugar de la reserva externa para variable  ↩
            local: optimiza velocidad
        ## Inicializo  la variable local suma
        movl  $0,%edx

        ## Número de iteracciones
        mov %ecx,%eax
        sub %ebx,%eax

bucle:
        add %ebx,%edx
        inc %ebx
        sub $1,%eax
        jns bucle

        ## Salvo el resultado de la suma como el valor de retorno
        movl  %edx, %eax

        ## Epílogo: Recupera el frame antiguo
        movl  %ebp, %esp      #restauro el stack pointer
        popl  %ebp            #restauro el frame pointer

        ## Retorno a la rutina principal
        ret
        .end
```

6.3. Compilación

- Seguir los pasos del proceso de compilación común a todas las sesiones.
  - `gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s`

6.4. Ejecución

- `./sumMtoN`
- `echo $?`
- Comprobar que funciona correctamente cambiando los valores de los parámetros: 1º valor de la suma y 2º valor de la suma.

6.5. Análisis del módulo fuente

- Leer en las hojas de referencia rápida el Programa Ejemplo Minimalista

6.5.1. Estructura

- La estructura del programa esta formada por los siguientes elementos:
  - Cabecera
  - Definición de Macros
  - Sección de Datos
  - Sección de Instrucciones

6.5.2. Ejecución modo paso a paso mediante el depurador GDB

- Compilar el programa con la opción de generación de la tabla de símbolos requerida por el depurador y generar el módulo binario ejecutable:
  - `gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s`
- Abrir el depurador GDB, cargar el módulo binario ejecutable y comprobar que se carga la tabla de símbolos junto al módulo binario ejecutable.
  - `gdb`
  - `file modulo_ejecutable`
  - `info sources`
- Configurar el fichero para el logging histórico de los comandos.
  - `set trace-commands on`
  - `set logging file sumMtoN_gdb_asm.txt`
  - `set logging on`
  - `shell ls -l sumMtoN_gdb_asm.txt`
- Activar un punto de ruptura en la instrucción de entrada al programa.
  - `b _start`
- Ejecutar el programa deteniéndolo en la primera instrucción del programa.
  - `run`
- Sin ejecutar ninguna instrucción del programa
  - Estado de la pila
    - Top del stack: `x $esp` ó `x $sp` : stack pointer
    - Bottom del frame: `x $ebp` ó `x $fp` : frame pointer
    - Contenido del top de la pila (dirección sp): argc: número de argumentos string de la línea de comandos en ejecución
      - `x /xw $sp`
    - Contenido una posición anterior al top de la pila (dirección sp+4): argv[0]: dirección del 1º string de la línea de comandos en ejecución
      - `p /s *(char **)($sp+4)`
- Ejecutar las líneas necesarias hasta entrar en la subrutina:

  - Comando step: `s` ya que el comando `n` no entra en la subrutina sino que la ejecuta completamente.
  - ¿A dónde apunta el stack pointer sp?¿Qué información contiene a donde apunta el sp?
    - `x /i *(int *)$sp` : ¿qué instrucción es?
- Ejecutar el prólogo de la subrutina
  - Nuevo frame
    - Nuevo valor del frame pointer: `p $fp`
    - Valor del stack pointer: `p $sp`
    - Acceso a la dirección de retorno tomando como refefencia el nuevo frame pointer: `x /i *(int *)($fp+4)`
- Ejecutar la subrutina hasta obtener el valor de retorno
  - Imprimir el valor de retorno: `p $eax`
- Ejecutar el epílogo de la subrutina
  - Valor del frame pointer: `p $fp`
  - Valor del stack pointer: `p $sp`
  - Dirección de retorno: `x *(int *)$sp`
- Ejecutar la instrucción de retorno
  - Dirección del stack pointer: `p $sp`
    - ¿Por qué ha cambiado la dirección del stack pointer?