Práctica 2.- Programación en ensamblador x86 Linux

1 Resumen de objetivos

Al finalizar esta práctica, se debería ser capaz de:

- Usar las herramientas `gcc`, `as`, `ld`, `objdump` y `nm` para compilar código C, ensamblar y enlazar código ensamblador, y localizar y examinar el código generado por el compilador.
- Reconocer la estructura del código generado por `gcc` para rutinas aritméticas muy sencillas, relacionando las instrucciones del procesador con la construcción C de la que provienen.
- Describir la estructura general de un programa ensamblador en `gas` (GNU assembler).
- Escribir un programa ensamblador sencillo.
- Usar interrupciones software para hacer llamadas al sistema operativo (*kernel* Linux).
- Enumerar los registros e instrucciones más usuales de los procesadores de la línea x86.
- Usar con efectividad un depurador como `gdb/ddd`, para ver los registros, ejecutar paso a paso y con puntos de ruptura, desensamblar el código, y volcar el contenido de la pila y los datos.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración, y reconocer cómo estas herramientas permiten familiarizarse con la arquitectura del computador.
- Explicar la gestión de pila en procesadores x86.
- Recordar y practicar en una plataforma de 32bits la representación de distintos tipos de datos (caracteres, números naturales, enteros con signo en complemento a dos), y el funcionamiento de diversas operaciones (incluyendo suma entera en doble precisión y división entera).

2 Herramientas de prácticas

Las prácticas se realizarán en Linux utilizando las herramientas GNU. Usaremos `gcc` para compilar (traducir fuente C a ensamblador, objeto o ejecutable), `as` para ensamblar (traducir fuente ensamblador a objeto), `ld` para enlazar (combinar varios ficheros objeto en un único fichero ejecutable), y para depurar usaremos `gdb`, a través del entorno gráfico `ddd`.

En la Figura 1 se pueden ver los programas que se deben ejecutar para obtener un fichero ejecutable a partir de un fichero fuente en lenguaje C: compilador, ensamblador y enlazador. En la práctica, el programador en lenguaje de alto nivel no tiene que ejecutar los tres programas por separado.

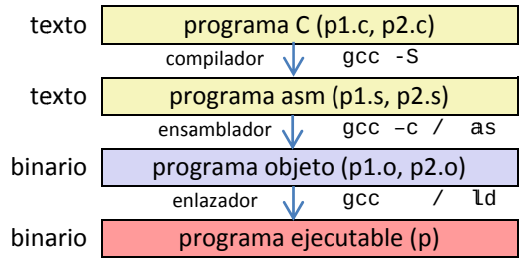

Figura 1: proceso de compilación

El código fuente se puede crear con cualquier editor de texto, como por ejemplo `gedit`, procurando que la extensión coincida con el tipo de lenguaje usado (.c/.s). El procesamiento del compilador `gcc` puede detenerse tras las etapas de traducción a ensamblador o a objeto con los modificadores (*switches*) `-S/-c`. El proceso puede continuarse con el propio `gcc` o con el ensamblador `as` y el enlazador `ld`.

Por ejemplo, con estos dos ficheros fuente en lenguaje C…

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Figura 2: programa p1.c

```
int main()
{
  return sum(1,3);
//printf("%d\n", sum(1,3));
}
```

Figura 3: programa p2.c

…usando `gcc` se podría producir el ejecutable `p` con una sola orden, o generar el código ensamblador u objeto para cada fichero, o retomar esos ficheros ensamblador u objeto para producir el ejecutable. Teniendo los ficheros intermedios, también se podría continuar con las otras herramientas `as/ld`.

Los modificadores necesarios se pueden consultar en los manuales (`man gcc`, `man as`, `man ld`…) aunque los más habituales son:

gcc			
switch	**argumento**	**significado**	**explicación**
-c		Compile	Compila o ensambla los fuentes, pero no enlaza. Se obtiene un objeto por cada fuente. Por defecto, los objetos se llaman como el fuente.o
-S		aSsembly	Compila los fuentes pero no ensambla. Se obtiene un fuente ensamblador por cada fuente C. Por defecto, se llaman como el fuente.s
-o	fichero.ext	Output	Cambiar el nombre del fichero producido. Por defecto, ejecutable a.out, objeto fuente.o, ensamblador fuente.s
-g	1…3	debuG	Genera información de depuración (por defecto, nivel 2)
-O	0…3, s		Optimizar para velocidad (0 no optimización…3 agresivo) o tamaño (s). Poner –O = –O1. No poner nada = –O0.
-m32 **-m64**		Machine	Genera código para entorno de 32/64 bits, aunque no sea el configurado por defecto.
-L	dir	LibraryDir	Añadir dir a la lista de búsqueda de librerías (preferible sin espacio: -Ldir)
-l	name	LibraryName	Enlazar contra libname.so / libname.a (preferible usar sin espacio: -lname)
-print-file-name=lib			Imprime el pathname que se usaría para lib a la hora de enlazar
-v		Verbose	Imprime los comandos ejecutados para las diversas etapas de compilación
-###			Como –v, pero no ejecuta los comandos. Es más fácil de leer.
as			
-g		debuG	mismo sentido
-o	fichero.o	Output	mismo sentido
--32 **--64**			mismo sentido
ld			
-L	dir	LibraryDir	mismo sentido
-l	name	LibraryName	mismo sentido
-m	elf_i386 elf_x86_64	Machine	mismo sentido
-M		printImpMap	Imprimir mapa de enlazado, posición en memoria de objetos, símbolos, etc.

Tabla 1: modificadores para gcc, as, ld

Ejercicio 1: gcc

Crear los ficheros p1.c y p2.c mostrados anteriormente (Figura 2, Figura 3), y reproducir con ellos la siguiente sesión en línea de comandos Linux. Recordar que los modificadores `–m32`, `--32`, `-melf_i386`, son necesarios cuando se desea generar código de 32bit en una plataforma de 64bit.

```
gcc      p1.c p2.c -o p    # compilar de una vez
./p ; echo $?              # muestra cód. ret. 4
file p                     # ELF 64-bit LSB executable

gcc –m32 p1.c p2.c -o p    # compilar para 32bits
./p ; echo $?              # muestra cód. ret. 4
file p                     # ELF 32-bit LSB executable

ls                         # existen p1.c, p2.c, p
gcc –m32 -O -S p1.c        # crea p1.s (optimización básica –O1)
    cat p1.s               # observar aspecto código x86
gcc         -S p2.c        # crea p2.s
    cat p2.s               # observar aspecto código x86_64
    rm  p2.s               # no lo necesitaremos

gcc -m32 -c p1.s p2.c      # crea p1.o, p2.o, ambos 32bit
ls; file p1.o p2.o         # comprobar: ELF 32-bit LSB relocatable

gcc -m32 p1.o p2.o –o p    # crea ejecut.p (no a.out) a partir de objetos
./p; echo $?               # funciona
```

Figura 4: compilación, ensamblado y enlazado, usando gcc

Observar que el código ensamblador x86 menciona registros de 32bit como EBP (en instrucciones como `pushl %ebp` / `movl %esp, %ebp`), mientras que el código x86_64 menciona registros de 64bit (como en `pushq %rbp` / `movq %rsp, %rbp`). En el *shell bash*, $? representa el código de estado retornado por el último programa ejecutado, y la almohadilla # introduce un comentario hasta final de línea. El punto y coma ";" separa dos comandos en la misma línea.

Ejercicio 2: as y ld

El mismo resultado se puede obtener con las distintas herramientas separadamente. Reproducir la siguiente sesión de comandos Linux.

```
rm *.s *.o p; ls          # limpiar
gcc –m32 –O –S p1.c p2.c  # ASM para 32bits, optimización básica –O1
ls                        # aparecen p1.s y p2.s

as –32 p1.s –o p1.o       # ensamblar creando objeto p1.o (no a.out)
as –32 p2.s –o p2.o
ls; file *.o

ld p1.o p2.o              # 2 errs: x86_64 con objs 32bit, falta _start
ld –m elf_i386 p1.o p2.o  # elf_i386 para link OK pero aún falta _start
                          # _start está definido en crt1.o, verlo con nm
gcc –### -m32 p1.o p2.o   # una forma de ver cómo enlaza gcc
gcc –print-file-name=     # reproducir último paso collect2 usando ld
LDIR=`gcc –print-file-name=`
echo $LDIR
ld –m elf_i386 p1.o p2.o –o p \
      -dynamic-linker /lib/ld-linux.so.2 \
      /usr/lib32/crt1.o /usr/lib32/crti.o /usr/lib32/crtn.o \
      -lc             $LDIR/32/crtbegin.o $LDIR/32/crtend.o
./p; echo $?              #funciona
```

Figura 5: compilación usando gcc, ensamblado y enlazado usando as y ld

Dependiendo de la version y configuración del compilador en la distribución que se use, el proceso de enlazado indicado por `gcc -###` será más o menos complicado. El *backslash* "\" continúa un comando que se desea prolongar a la siguiente línea. Las comillas inversas `…` (*backquote*) en bash sirven para reemplazar (*command substitution*) el comando entrecomillado por su salida estándar. En este caso, las hemos usado para definir la variable de entorno `LDIR`, usada posteriormente 2 veces con `ld`.

Si se desea, se puede modificar el programa p2.c para que el comentario "//" afecte a la primera sentencia en lugar de a la última. De esta forma, el programa imprime el resultado, en lugar de retornarlo como código de estado. La sintaxis del formato de printf se puede consultar en los manuales (`man 3 printf`).

2.1 Código ensamblador y código máquina

La traducción del ejemplo a lenguaje ensamblador ya la vimos en la sesión de la Figura 4:

```
int sum(int x, int
y)
{
  int t = x+y;
  return t;
}
```

Figura 6: programa p1.c

```
.file "p1.c"
 .text
.globl sum
 .type sum, @function
sum:
 pushl %ebp
 movl %esp, %ebp
 movl 12(%ebp), %eax
 addl 8(%ebp), %eax
 leave
 ret
 .size sum, .-sum
 .section .note.GNU-stack,"",@progbits
 .ident "GCC: (GNU) 3.4.6 20060404 (Red Hat 3.4.6-9)"
```

Figura 7: fichero ensamblador p1.s

La versión ensamblador es una representación legible de las instrucciones máquina en que se convierte el programa, como las comentadas anteriormente `pushl %ebp / movl %esp, %ebp`. Contiene también directivas, como `.text` (iniciar sección de código) y `.size` (tamaño de un objeto). En este caso, se define el tamaño del objeto `sum` (una función global, ver `.global` y `.type`) como "`.-sum`", esto es, la diferencia entre el contador de posiciones "`.`" y la propia etiqueta `sum`. El ensamblador *emite* código máquina conforme traduce ensamblador, ocupando posiciones (bytes) de memoria. El símbolo "`.`" es la posición por donde se va ensamblando, y cada etiqueta toma el valor del contador cuando se emite.

El fichero objeto p1.o no es legible, ya que contiene código máquina, pero se puede desensamblar y consultar los símbolos que define con otras utilidades del paquete GNU *binutils*, como `objdump` y `nm`.

Los modificadores necesarios se pueden consultar en los manuales (`man objdump`, `man nm`) aunque los más habituales son:

objdump	fich.obj.		
switch	**argumento**	**significado**	**explicación**
-d		Disassemble	Muestra los mnemotécnicos ensamblador correspondientes a las instrucciones máquina en las secciones de código del fichero objeto
-S		Source	Intercala código fuente con desensamblado. Implica –d. Requiere compilar con –g.
-h		Headers	Resumen de las cabeceras de sección presentes
-r		Reloc	Muestra las reubicaciones
-t		table	Muestra las entradas de la tabla de símbolos (similar a nm)
-T		Table	Muestra tabla de símbolos dinámica (similar a nm -D). Para librerías compartidas.
-j / --section=	name	Just	Seleccionar información sólo de la sección mencionada
-s / --full-contents			Mostrar contenidos completos de todas las secciones (o sólo de las indicadas con -j)
nm	fich.obj.		
-D		Dynamic	Mostrar símbolos dinámicos (p.ej. en librerías compartidas)

Tabla 2: modificadores para objdump, nm

Ejercicio 3: objdump y nm

Reproducir la siguiente sesión en línea de comandos Linux

```
objdump –d p1.o  # mostrado al lado->
objdump –t p1.o
nm         p1.o  # sum está en .text

objdump –S p1.o        # no hay debug
gcc –m32 –g –O –c p1.c # añadir -g
objdump –S p1.o        # ahora sí

objdump –t p1.o        # secciones -g
objdump –h p1.o        # ver.text=11B
gcc –m32    –O –c p1.c # quitar -g
objdump –S p1.o        # ahora no
objdump –h p1.o
```

Figura 8: sesión Linux

```
p1.o: formato del fichero elf32-i386

Desensamblado de la seccion .text:

00000000 <sum>:
   0: 55          push   %ebp
   1: 89 e5       mov    %esp,%ebp
   3: 8b 45 0c    mov    0xc(%ebp),%eax
   6: 03 45 08    add    0x8(%ebp),%eax
   9: c9          leave
   a: c3          ret
```

Figura 9: desensamblado de p1.o

Como vemos en la Figura 9, el ensamblador emitió 11 bytes, el contador iba por 0 cuando se definió `sum`, irá por 0xb tras emitir `ret`, y por tanto "`.size sum, .-sum`" calculará el tamaño de sum como 11. La primera instrucción, "`push %ebp`", se codifica en lenguaje máquina como 0x55 y ocupa 1B, la posición 0. La segunda instrucción, "`mov %esp,%ebp`", empieza en 0x1, ocupa 2B y acaba por tanto en 0x2, dejando el contador de posiciones en 0x3.

Se puede comprobar (con `nm`) que el símbolo `_start` que nos impedía enlazar nuestros dos objetos con `ld` a secas (Figura 5) está en uno de los objetos añadidos por `gcc`. En el Apéndice 2 hay un resumen de las instrucciones y modos de direccionamiento x86 y de las directivas del ensamblador GNU, que puede resultar útil para entender tanto este desensamblado como el siguiente programa completo.

3 Primer programa completo en ensamblador

Teclear (o copiar-pegar, o descargar del sitio web de la asignatura) el código de la Figura 10 en un fichero llamado `saludo.s`. Si se opta por reescribirlo, tener en cuenta que la almohadilla # indica que el resto de la línea es comentario, con lo cual no es necesario copiarlo.

En el código se pueden distinguir: instrucciones del procesador, directivas del ensamblador, etiquetas, expresiones y comentarios. Las instrucciones usadas en este caso han sido INT y MOV, para realizar las dos llamadas al sistema requeridas (escribir mensaje y terminar programa). Las directivas son comandos que entiende el ensamblador (no instrucciones del procesador), y se han usado para declarar las secciones de datos y código (`.data` y `.text`), para emitir un *string* y un entero (en `.data`) y para declarar como global el punto de entrada (en `.text`). Las etiquetas se han usado para nombrar esos tres elementos (`saludo`, `longsaludo` y `_start`), y poder referirse a ellos posteriormente (en *`WRITE`* o en `.global`), ya que representan su dirección de comienzo. Notar el uso del contador de posiciones y aritmética de etiquetas (`.-saludo`) para calcular la longitud del *string*. La otra expresión de inicialización es el valor del *string*. Los comentarios se indican con # ó `/**/`. Los valores inmediatos se prefijan con $, y los registros con %.

```
# saludo.s:  Imprimir por pantalla
#            Hola a todos!
#            Hello, World!
# retorna:   código retorno 0, programado en la penúltima línea
#            comprobar desde línea de comandos bash con echo $?

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
#     datos hex, octal, binario, decimal, char, string:
#            0x, 0,     0b,      díg<>0,  ',    ""
#     ejs: 0x41, 0101, 0b01000001,  65,  'A,   "AAA"

.section .data      # directivas comienzan por .
            # no son instrucciones máquina, son indicaciones para as
            # etiquetas recuerdan valor contador posiciones (bytes)
saludo:
      .ascii "Hola a todos!\nHello, World!\n"       # \n salto de línea

longsaludo:
      .int   .-saludo     # . = contador posic. Aritmética de etiquetas.

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)

.section .text      # cambiamos de sección, ahora emitimos código
.global _start      # muestra punto de entrada a ld (como main en C)

_start:                   # punto de entrada ASM (como main en C)

      #     Llamada al sistema WRITE, consultar "man 2 write"
      #     ssize_t write(int fd, const void *buf, size_t count);
      mov $4, %eax        # write: servicio 4 kernel Linux
      mov $1, %ebx        #    fd: descriptor de fichero para stdout
      mov     $saludo, %ecx     #   buf: dirección del texto a escribir
      mov  longsaludo, %edx     # count: número de bytes a escribir
      int $0x80           # llamar write(stdout, &saludo, longsaludo);

      #     Llamada al sistema EXIT, consultar "man 2 exit"
      #     void _exit(int status);
      mov $1, %eax        #   exit: servicio 1 kernel Linux
      mov $0, %ebx        # status: código a retornar (0=OK)
      int $0x80           # llamar exit(0);
```

Figura 10: saludo.s: ejemplo de llamadas al sistema WRITE y EXIT

En arquitectura x86 (y x86_64) cada posición de memoria es un byte. Ya vimos en la Figura 7 cómo se usó aritmética de etiquetas para calcular el tamaño ocupado por la función `sum`. En este caso, podríamos modificar (alargar o acortar) el *string* en el código fuente ensamblador, y la variable `longsaludo` tomaría automáticamente el valor correcto (longitud) para la posterior llamada a *`WRITE`*.

Un mecanismo frecuentemente usado por los Sistemas Operativos para permitir que se les realicen llamadas al sistema (*syscalls*) son las interrupciones software (que son instrucciones programadas, en oposición a las interrupciones hardware o las excepciones). El mecanismo de interrupción permite el cambio a espacio kernel (en oposición al espacio de usuario). El programador utiliza un vector concreto (`0x80` en el caso de Linux) cuando desea realizar la llamada. La subrutina de servicio espera encontrar el número de servicio en EAX y sus argumentos en sucesivos registros EBX, ECX, EDX, ESI, EDI, EBP (hasta 6) de manera que el programador debe fijar estos valores antes de realizar la interrupción `int 0x80`. Si la llamada al sistema produce un valor de retorno, lo devuelve en EAX (es decir, que se encuentra disponible en EAX tras ejecutarse `int 0x80`).

Los números de servicio (llamada) pueden encontrarse en `/usr/include/asm/unistd_32.h`, y los argumentos de cada llamada pueden conocerse leyendo la correspondiente página de manual de la sección 2 (Llamadas al Sistema). En nuestro caso, nos interesa consultar `man 2 write` y `man 2 exit` (o `info write`, `info exit`) para saber que tienen 3 (EBX…EDX) y 1 argumentos, respectivamente. El argumento de `exit(status)` es un código de retorno (se puede probar a cambiarlo en el fuente y comprobarlo con `echo $?`) mientras que `write(fd, buf, count)` escribe `count` bytes a partir de `buffer` en el descriptor de fichero `fd`. En concreto, el descriptor para la salida estándar (STDOUT_FILENO) está definido en `/usr/include/unistd.h` (también se puede comprobar listando `ls -la /dev/stdout`).

Ejercicio 4: ddd

Ensamblar y enlazar el programa `saludo.s`, incluyendo información de depuración, y reproducir la siguiente sesión ddd. Aunque usemos el front-end ddd en modo gráfico, es conveniente aprender también los comandos gdb en modo texto. Hay una lista de comandos en el enlace [6].

```
as --32 –g     saludo.s –o saludo.o     # ensamblar 32b / info.depuración
ld –m elf_i386 saludo.o –o saludo       # enlazar p/32bits
ddd saludo &                            # sesión ddd en modo gráfico

   list             # localizar línea “mov $1,%ebx” y ponerle breakpoint
   break 30         # equiv. en modo gráfico: cursor a izq. línea y stop
   info break       # equiv.gráf: Source->Brkpts. Notar address 080480xx

   run              # eq.gr: Program->Run /(View->CmdTool->)botonera->Run
   disassemble      # eq.gr: View->MachCodeWin
   print $eip             # Notar dirección break=EIP
   print $eax             # Notar EAX=4, pero EBX=0 aún
   info registers         # eq.gr: Status->Registers
   stepi                  #
   p $eip                 # EIP sigue avanzando (p=print)
   p $ebx                 # Notar EBX=1 ahora
   si                     #                   (si=stepi)
   p/x $ecx               # Notar ECX=080490xx > EIP

   x/32cb &saludo   # eq.gr: Data->Memory->Examine 32 char bytes &saludo
   x/32xb &saludo   # Print p/probar (cambiar a hex bytes)/Display p/fijo
   x/1dw &longsaludo      # Print para ver dirección de inicio y valor
   x/1xw &longsaludo      # Ver correspondencia tras final saludo
   x/4xb &longsaludo
   si               # Comprobar regs EAX,EBX,ECX,EDX = 4,1,0x080490…,28
   info reg         #                (write,stdout,&saludo,longsaludo)
   disas            # $saludo=$0x0804…(inm), longsl…=0x0804…(dir), %edx=28(reg)
   si               # Se escribe mensaje en pantalla (View->GDB Console)

2x si / cont              # Pulsar cont o clickar 3 stepi para exit(0)
   set $ebx=1             # o parar justo antes del final
   stepi                  # y cambiar código de retorno sobre la marcha

   run                          # volver a empezar programa
   print      saludo            # interpreta string como entero 4B
   p   (char) saludo            # typecast a char 1B ‘H’
   p    (int) saludo            # interpreta 4B “Hola” (4 códigos ASCII)
   p  /x(int) saludo            # como un entero 0x616c6f48
```

```
p         &saludo             # dirección en memoria
p  (char*)&saludo             # typecast a char*
p  (char*)&saludo+13          # saltarse 13 letras, localizar \n
p*((char*)&saludo+13)         # cambiarlo por '-'
set var *((char*)&saludo+13)='-'
print     (char*)&saludo      # comprobar cambio en memoria
cont                          # comprobar cambio en ejecución
```

Figura 11: ensamblado, enlazado, y sesión de depuración usando gbd/ddd

Como vemos, la forma general de usar un depurador es escoger un punto de parada (o varios), lanzar la ejecución, comprobar valores de variables y registros cuando el programa se detenga (al encontrar algún punto de parada), y seguir ejecutando (paso a paso, continuar ejecución normal, o volver a empezar desde el principio). En el Apéndice 3, Tabla 10, se ofrecen unas preguntas de autocomprobación por si se desea poner a prueba la comprensión que se ha obtenido de este apartado.

4 Llamadas a funciones

Es conveniente dividir el código de un programa entre varias funciones, de manera que al ser éstas más cortas y estar centradas en una tarea concreta, se facilita su legibilidad y comprensión, además de poder ser reutilizadas si hacen falta en otras partes del programa. En la Figura 12 se muestra un programa ensamblador con una función (subrutina) que calcula la suma de una lista de enteros de 32bits. La dirección de inicio de la lista y su tamaño se le pasa a la función a través de registros. El resultado se devuelve al programa principal a través del registro EAX.

Conocer el funcionamiento de la pila (*stack*) es fundamental para comprender cómo se implementan a bajo nivel las funciones. La pila se utiliza (en llamadas a subrutinas) para guardar la dirección de retorno, para almacenar las variables locales, y para pasar argumentos (según la convención de llamada). Las instrucciones PUSH y POP (Apéndice 2, Tabla 3) y las llamadas y retornos de subrutinas (CALL, RET, INT, IRET, Tabla 7) utilizan de forma implícita la pila, que es la zona de memoria adonde apunta el puntero de pila ESP (RSP en x86_64). La pila crece hacia direcciones inferiores de memoria, y ESP apunta al último elemento insertado (tope de pila), de manera que PUSH primero decrementa ESP en el número de posiciones de memoria que ocupe el dato a insertar, y luego escribe ese dato en las posiciones reservadas, a partir de donde apunta ESP ahora. Similarmente POP primero lee del tope de pila, guardando el valor en donde indique su argumento, y luego incrementa ESP. Por su parte, CALL guarda la dirección de retorno en pila antes de saltar a la subrutina indicada como argumento, y RET recupera de pila la dirección de retorno.

Ejecutar el programa de la Figura 12 paso a paso con ddd y comprobar que el funcionamiento de CALL y RET se corresponde con lo explicado. En el Apéndice 3, Tabla 11, se ofrecen unas preguntas de autocomprobación por si se desea poner a prueba la comprensión que se ha obtenido de este apartado.

```
# suma.s:     Sumar los elementos de una lista
#             llamando a función, pasando argumentos mediante registros
# retorna:    código retorno 0, comprobar suma en %eax mediante gdb/ddd

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
.section .data
lista:
       .int 1,2,10, 1,2,0b10, 1,2,0x10  # ejemplos binario 0b / hex 0x
longlista:
       .int   (.-lista)/4  # . = contador posiciones. Aritm.etiquetas
resultado:
       .int 0x01234567           # para ver cuándo se modifica cada byte
                                 # 0x0123456789ABCDEF cuando sea 64b
# formato:
#      .ascii "suma = %u = %x hex\n\0"  # formato para printf() libC
# el string "formato" sirve como argumento a la llamada printf opcional
# si se usa printf, compilar el programa con gcc en lugar de ensamblar
# en la siguiente práctica aprenderemos cómo ensamblar y linkar con -lc
```

```
# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)

.section .text                  # PROGRAMA PRINCIPAL
_start: .global _start          # se puede abreviar de esta forma
# main: .global main            # Programa principal si se usa gcc

       mov     $lista, %ebx     # dirección del array lista
       mov  longlista, %ecx     # número de elementos a sumar
       call suma                # llamar suma(&lista, longlista);
       mov  %eax, resultado     # salvar resultado

#      # si se usa este bloque, usar también la línea que define formato,
#      # cambiar la línea _start por main, y compilar con gcc
#      push  resultado          # versión libC de syscall __NR_write
#      push  resultado     # ventaja: printf() con formato "%d" / "%x"
#      push $formato       #      traduce resultado a ASCII decimal/hex
#      call  printf        # == printf(formato,resultado,resultado)
#      add  $12, %esp
                                #   void _exit(int status);
       mov  $1, %eax            #   exit: servicio 1 kernel Linux
       mov  $0, %ebx            # status: código a retornar (0=OK)
       int  $0x80               # llamar _exit(0);

# SUBRUTINA: suma(int* lista, int longlista);
# entrada:   1) %ebx = dirección inicio array
#            2) %ecx = número de elementos a sumar
# salida:       %eax = resultado de la suma

suma:
       push     %edx            # preservar %edx (se usa como índice)
       mov  $0, %eax            # poner a 0 acumulador
       mov  $0, %edx            # poner a 0 índice
bucle:
       add  (%ebx,%edx,4), %eax # acumular i-ésimo elemento
       inc   %edx               # incrementar índice
       cmp   %edx,%ecx          # comparar con longitud
       jne    bucle             # si no iguales, seguir acumulando

       pop   %edx               # recuperar %edx antiguo
       ret
```

Figura 12: suma.s: ejemplo de llamada a subrutina (paso de parámetros por registros)

Desarrollo de las Prácticas en Windows + VirtualBox + Ubuntu 10.04.LTS

Como ya se ha comentado en clase de Teoría [1] (Presentación p.33 y Tema 1 p.64), en el laboratorio estamos usando Ubuntu 10.04.LTS 64bit. En un portátil con Windows, uno puede instalar Ubuntu en una partición separada, pero hay opciones más cómodas: Wubi y VirtualBox. Para realizar las prácticas de la asignatura en un portátil con Windows, recomendamos instalar VirtualBox, y crear una máquina virtual con Ubuntu 10.04.LTS 64bit idéntico al del laboratorio de prácticas.

Por defecto Ubuntu no instala los paquetes de compatibilidad para compilar en entorno de 32bits ni el entorno gráfico ddd, así que se deben añadir los paquetes *ia32-libs*, *gcc-multilib* y *ddd*, posiblemente con la aplicación gráfica *System→Administration→Synaptic Package Manager*. Tampoco está activado por defecto el *firewall*; se puede activar con el comando "`sudo ufw enable`".

De esta forma se pueden repetir los tutoriales de prácticas, como éste que acabamos de completar, de forma independiente. Al estar las sesiones de tutorial transcritas en su totalidad (incluso se han transcrito los equivalentes en modo texto de los comandos gráficos ejecutados en ddd), se pueden probar antes de venir al laboratorio, se pueden repetir después de haber asistido al tutorial, y se pueden repasar en cualquier momento, independientemente de las sesiones de laboratorio.