



Universidad de Murcia
Facultad de Informática

TÍTULO DE GRADO EN
INGENIERÍA INFORMÁTICA

Fundamentos de Computadores

Tema 5: Lenguajes del computador:
alto nivel, ensamblador y máquina

Boletines de prácticas

CURSO 2020 / 21

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores



Índice general

I. Boletines de prácticas	2
B5.1. Boletín 1. Generación de código, enlazado, carga en memoria y ejecución de programas.	2
B5.1.1. Objetivos	2
B5.1.2. Plan de trabajo	2
B5.1.3. Generación de código ensamblador	3
B5.1.4. Generación del código objeto	3
B5.1.5. Enlazado	3
B5.1.6. Carga en memoria de un fichero ejecutable	4
B5.1.7. Ejecución paso a paso de un programa y ubicación de las variables globales en memoria . . .	6
B5.1.8. Ejercicios a realizar durante la sesión	8
B5.1.9. Apéndice: Instalación del gcc-4.8 en Ubuntu 18.04	9
B5.2. Boletín 2: Traducción de C a ensamblador del Intel x86-64	10
B5.2.1. Objetivos	10
B5.2.2. Plan de trabajo	10
B5.2.3. EJEMPLO 1: Segmentos de datos y código. Instrucciones aritmético-lógicas y movimiento de datos.	10
B5.2.4. EJEMPLO 2: Instrucciones de salto condicional e incondicional. Acceso a arrays.	12
B5.2.5. EJEMPLO 3: Llamadas a subrutinas. Paso de parámetros y valor de retorno	13
B5.2.6. Ejercicios a realizar durante la sesión	14

Boletines de prácticas

B5.1. Boletín 1. Generación de código, enlazado, carga en memoria y ejecución de programas.

B5.1.1. Objetivos

En este boletín se ilustrarán la codificación de las instrucciones en lenguaje ensamblador y en lenguaje máquina de la arquitectura Intel x86-64, así como el proceso de enlazado de programas y su posterior carga en memoria para ejecución. Se hará énfasis en la comprensión de cómo es sobre el último nivel de la jerarquía de traducción (el código máquina en binario) sobre el que directamente trabaja la CPU ejecutando instrucciones, así como en aspectos clave en la generación final de programas ejecutables, como son la reubicación de direcciones al enlazar los programas y cargarlos en memoria para ejecución, y el uso de las librerías del sistema.

Para la realización de esta práctica se asume que el alumno posee unos conocimientos mínimos del manejo de Linux desde la línea de comandos, adquiridos en sesiones anteriores.

B5.1.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento, en grupos de hasta dos personas por PC, de los pasos expuestos en el ejemplo del boletín (simplemente replicándolos y observando los resultados).
2. Realización, en grupos de hasta dos personas por PC, de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos (bajo la supervisión del profesor).

Obtener el fichero fuente `hola.c` disponible como recurso en el Aula Virtual. Dicho fichero contiene el siguiente programa en C:

```
#include <stdio.h>

int main() {
    puts("Hola, mundo!"); // puts(s): Escribe la cadena s por la salida estándar
}
```

En primer lugar, simplemente compilaremos el programa, para obtener un fichero ejecutable llamado `hola`. Utilizamos para ello el siguiente comando:

```
$ gcc-4.8 hola.c -o hola
```

El comando `gcc` (del que usaremos en este caso la versión 4.8¹) es el compilador GNU de C, el más utilizado en entornos Linux. La opción `-o` sirve para indicar el nombre del fichero compilado generado. Comprobamos que, efectivamente, se ha generado un fichero `hola`, con los permisos de ejecución adecuados, y a continuación simplemente ejecutamos dicho programa:

```
$ ./hola
Hola, mundo!
```

¹Esta versión no es la utilizada por defecto en Ubuntu 18.04. Para instalarla se han de ejecutar los comandos contenidos en el apéndice que se encuentra al final de este boletín.

B5.1.3. Generación de código ensamblador

A continuación, vamos a volver a usar el `gcc`, pero en este caso con la opción `-S`, para generar no un ejecutable, sino el correspondiente fichero en lenguaje ensamblador del Intel x86-64²:

```
$ gcc-4.8 hola.c -fno-asynchronous-unwind-tables -S -o hola.s
```

El resultado de la compilación es un nuevo fichero de texto ASCII llamado `hola.s`, cuyo contenido más relevante se muestra a continuación:

```
[...]
.LC0:
.string "Hola, mundo!"
[...]
```

main:

```
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $.LC0, %edi
    call    puts
    popq    %rbp
    ret
[...]
```

B5.1.4. Generación del código objeto

Ahora vamos a compilar el programa `hola.c` para generar el correspondiente fichero objeto `hola.o` cuyo contenido ya son instrucciones codificadas en lenguaje máquina. El código objeto se puede directamente a partir del código fuente `hola.c`, con el comando siguiente:

```
$ gcc-4.8 -c hola.c -o hola.o
```

A continuación volcaremos en un fichero `hola.o.disassembled` el código objeto generado desensamblado, con el siguiente comando:

```
$ objdump -d hola.o > hola.o.disassembled
```

El volcado generado se parece más o menos a lo siguiente:

```
0000000000000000 <main>:
0: 55                      push    %rbp
1: 48 89 e5                mov     %rsp,%rbp
4: bf 00 00 00 00          mov     $0x0,%edi
9: e8 00 00 00 00          callq   e <main+0xe>
e: 5d                      pop     %rbp
f: c3                      retq
```

En él podemos comprobar cómo el código correspondiente a la función `main` del programa se muestra convenientemente formateado en tres columnas: para cada instrucción, la primera columna indica su desplazamiento relativo al comienzo del fichero objeto, la segunda el código máquina mostrado en bytes en hexadecimal (con longitud variable para las distintas instrucciones, algunas ocupando sólo un byte y otras ocupando hasta 5 bytes en el ejemplo), y finalmente una tercera columna donde se muestra el código ensamblador correspondiente a dicha instrucción.

B5.1.5. Enlazado

En principio, existe un programa en Linux, llamado `ld`, para hacer el enlazado de código(s) objeto(s) y librería(s) en un sólo ejecutable, pero el propio compilador `gcc` se puede encargar de llamarlo por nosotros. Así que, para generar el ejecutable `hola` a partir del anterior fichero `hola.o`, simplemente podemos llamar a `gcc` así:

```
$ gcc-4.8 hola.o -o hola
```

²La opción `-fno-asynchronous-unwind-tables` que aparece en el comando no sería estrictamente necesaria, pero al ponerla el código aparece bastante más limpio, sin ciertas directivas `{.cfi_}` que en nuestro caso no son necesarias, y cuya explicación en cualquier caso no es objetivo de esta práctica.

Librerías dinámicas: El ejecutable generado se puede probar directamente, tecleando el comando `./hola` como hicimos en el boletín anterior. Pero en este momento nos interesa más comprobar las librerías dinámicas con las que enlaza nuestro ejecutable generado. Para ello usamos el comando `ldd`:

```
$ ldd hola
```

Nos contestará con algo parecido a lo siguiente:

```
linux-vdso.so.1 (0x00007ffe89736000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9abaf57000)
/lib64/ld-linux-x86-64.so.2 (0x00007f9abb348000)
```

Cada línea nos dice el fichero donde se encuentra la librería dinámica correspondiente, y la dirección virtual de nuestro programa a la que las rutinas allí contenidas son mapeadas. En particular, la librería que aquí más nos interesa es la librería estándar de C (`libc`), que contiene, entre otras muchas utilidades, la función `puts` usada por nuestro programa. Las otras dos librerías (`linux-vdso` y `ld-linux-x86-64`) se corresponden, respectivamente, con las llamadas al sistema de Linux (que, como tal, están en el código del núcleo, siempre cargado desde el arranque en memoria, y por tanto no necesitan fichero para almacenarse), y la propia librería que gestiona la posibilidad de carga dinámica de librerías en memoria, para ser compartidas entre varios programas.

Puesto que el ejecutable generado enlaza con librerías dinámicas, su tamaño tiende a ser bastante pequeño (en torno a los 10KB, dependiendo también de la versión concreta del `gcc` utilizada). Podemos comprobarlo con el comando `ls -l`:

```
$ ls -l hola
```

Librerías estáticas: Sin embargo, tal vez podría interesarnos generar un ejecutable *estático*, que sea autocontenido, y por tanto no dependa de librerías dinámicas externas. Para ello, simplemente hay que compilar con la opción `-static` del `gcc`:

```
$ gcc-4.8 -static hola.o -o hola.static
```

Esta vez podemos comprobar con `ldd hola.static` que el ejecutable generado no enlaza con ninguna librería dinámica, pero a cambio sí que se tiene que pagar un precio en el tamaño del ejecutable generado (en torno a 800KB; comprobarlo con `ls -l hola.static`).

Otra prueba interesante de lo que está pasando la podéis hacer con el siguiente comando:

```
$ objdump -d hola.static > hola.static.disassembled
```

Y echando un vistazo por encima al (¡enorme!) listado de código desensamblado generado. Se pueden incluso localizar las partes del código correspondientes a la función `puts` utilizada (buscando la cadena `_IO_puts` en el fichero generado abierto con un editor de texto cualquiera).

B5.1.6. Carga en memoria de un fichero ejecutable

Vamos a generar de nuevo un fichero ejecutable `hola.static`, enlazado estáticamente, mediante compilación *directa* a partir del fuente en C original, pero en este caso le añadimos la información necesaria para poder *tracearlo* en tiempo de ejecución (es decir, cargarlo en memoria para ejecutarlo dentro de un entorno controlado, donde podamos ir ejecutándolo paso a paso y tengamos acceso tanto a los registros de la CPU como a las zonas de datos e instrucciones del programa). Para que el ejecutable pueda ser utilizado de esta manera, ha de ser generado usando la opción `-g` del compilador `gcc`:

```
$ gcc-4.8 -g -static hola.c -o hola.static
```

Una vez generado el ejecutable “traceable”, lo cargaremos en memoria usando el potente depurador de programas GNU, el `gdb`³:

```
$ gdb hola.static
```

Con ello se arranca el programa `gdb`, que tiene su propio intérprete, y donde podemos empezar a teclear una serie de comandos. Por ejemplo, el comando `list` sirve para listar el código fuente en C del programa original:

```
(gdb) l
1 #include <stdio.h>
2
3 int main() {
4     puts("Hola, mundo!\n");
5 }
(gdb)
```

Ubicación del código: El comando `disassemble`, por su parte, nos sirve en cambio para listar el código máquina convenientemente desensamblado:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x000000000400b4e <+0>: push    %rbp
0x000000000400b4f <+1>: mov     %rsp,%rbp
0x000000000400b52 <+4>: mov     $0x491d04,%edi
0x000000000400b57 <+9>: callq   0x410200 <puts>
0x000000000400b5c <+14>: pop     %rbp
0x000000000400b5d <+15>: retq
```

Se trata esencialmente del mismo código que se vio anteriormente en el volcado del código objeto (cuando usamos el comando `objdump`), pero son varias las diferencias claves a observar entre aquel y el código desensamblado correspondiente al programa ya cargado en memoria:

1. En primer lugar, se observa que el código cargado en el `gdb` está ya ubicado en direcciones virtuales concretas (a partir de la `0x400b4e` en nuestro ejemplo, correspondiente al comienzo de la función `main`), frente a las direcciones relativas a 0 del código objeto original.
2. En segundo lugar, y como consecuencia de lo anterior, las propias direcciones codificadas en algunas instrucciones (p.e., llamadas a subrutinas o accesos a variables en memoria) contienen ya direcciones definitivas. Por ejemplo, la llamada `callq` en el desplazamiento `0x9` del código objeto original, que se había codificado inicialmente dejando los 4 huecos de bytes para la dirección a cero (secuencia de código máquina `e8 00 00 00 00`), ha sido traducida en el código final, ya reubicado, a la instrucción `callq 0x410200 <puts>`, que como vemos ya hace referencia a la dirección virtual de memoria final `0x410200`, donde efectivamente comienza la rutina `puts` una vez ubicada en memoria⁴. Naturalmente, estas diferencias se pueden observar sólo cuando se mira el código máquina final, tal y como puede observarse volcando el mismo en pantalla (el comando `x/16bx` sirve para volcar en pantalla 16 bytes en formato hexadecimal, byte a byte). Se observa claramente como el volcado de los bytes es casi idéntico al del código objeto original, excepto en las direcciones reubicadas, como acaba de comentarse:

```
(gdb) x/16bx main
0x400b4e <main>:      0x55    0x48    0x89    0xe5    0xbf    0x04    0x1d    0x49
0x400b56 <main+8>:    0x00    0xe8    0xa4    0xf6    0x00    0x00    0x5d    0xc3
0x400b5e:           0x66    0x90    0x53    0x48    0x81    0xec    0x88    0x00
```

³El programa `gdb` es un potentísimo depurador de programas, con infinidad de potencialidades y opciones. En este documento simplemente utilizaremos unas pocas de ellas, de forma muy controlada, con el fin de ilustrar los aspectos más relevantes del proceso de carga y ejecución de programas vistos en la teoría.

⁴Esto último se puede comprobar fácilmente simplemente ejecutando en `gdb` el comando `disassemble puts`.

En este caso particular del `callq 0x410200 <puts>`, los bytes con el código máquina final son `0xe8 0xa4 0xf6 0x00 0x00`, situados a partir de la dirección virtual final `0x400b57 (<main+9>)`, puesto que la dirección de la rutina está almacenada de forma relativa a la dirección siguiente a la del `call`, en este caso `0x400b5c`, y $0x410200 - 0x400b5c = 0xf6a4$, que en little endian y extendido a 32 bits resulta en la secuencia `0xa4 0xf6 0x00 0x00`.

Un segundo ejemplo, quizá aún más claro, es la instrucción reubicada en la dirección final `0x400b52` (o sea, `<main+4>`). Esta instrucción es `mov $0x491d04, %edi`, y su código máquina final es `0xbf 0x04 0x1d 0x49 0x00`, comprendido entre las direcciones `0x400b52` y `0x400b56`, mientras que el código objeto original sin reubicar era `bf 00 00 00 00`, a partir del desplazamiento relativo 4. Esta instrucción se corresponde con la instrucción del código objeto `mov $0x0, %edi`, que aparecía en el código ensamblador original como `movl $.LC0, %edi`. Como vemos, la dirección del código objeto sin reubicar tuvo que fijarse a `0x0`, puesto que hasta que el código no fue debidamente enlazado y ubicado en memoria se desconocía completamente la dirección donde finalmente se ubicaría la etiqueta de la cadena de caracteres `.LC0` (*Hola, mundo!*). Dicha dirección final virtual acabaría siendo `0x491d04`, como se aprecia claramente en los cuatro últimos bytes de su código máquina reubicado final (codificado en little endian, `0x04 0x1d 0x49 0x00`). Con el comando `x/13bc 0x491d04` podemos ver los 13 bytes que hay a partir de dicha dirección de memoria, interpretados como caracteres: `'H', 'o', 'l', 'a',` etc.

B5.1.7. Ejecución paso a paso de un programa y ubicación de las variables globales en memoria

Obtener el fichero fuente `globals.c` disponible como recurso en el Aula Virtual. Dicho fichero contiene el siguiente programa en C, que recorre un array y establece el valor de todos los elementos a -1.

```
#define ARRAY_SIZE 10

int array[ARRAY_SIZE] = {10,9,8,7,6,5,4,3,2,1};

int main() {
    int i;
    for(i=0; i < ARRAY_SIZE; i++) {
        array[i] = -1;
    }
}
```

Generamos a partir del código fuente un fichero ejecutable `globals` enlazado dinámicamente, mediante compilación *directa*, incluyendo la información necesaria para poder tracearlo, y a continuación lo depuramos con `gdb`:

```
$ gcc-4.8 -g globals.c -o globals
$ gdb globals
```

Ubicación de los datos globales: Usando el comando `x` (*examinar*) podemos examinar cualquier zona de la memoria del programa, y en particular, la que contiene los datos globales del programa. Con el comando `x/40bx array` volcamos en pantalla los 40 bytes (mostrados en hexadecimal) que ocupa el vector `array`, y así ver qué direcciones de memoria ocupa:

```
(gdb) x/40bx array
0x601040 <array>:      0x0a    0x00    0x00    0x00    0x09    0x00    0x00    0x00
0x601048 <array+8>:    0x08    0x00    0x00    0x00    0x07    0x00    0x00    0x00
0x601050 <array+16>:   0x06    0x00    0x00    0x00    0x05    0x00    0x00    0x00
0x601058 <array+24>:   0x04    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0x601060 <array+32>:   0x02    0x00    0x00    0x00    0x01    0x00    0x00    0x00
```

Vemos que está ubicado a partir de la dirección `0x601040`⁵, y ocupando 4 bytes (32 bits) por cada entero, almacenados en memoria little endian (esquema de almacenamiento usado por el ISA x86-64). El depurador `gdb` tiene una forma más cómoda de imprimir los contenidos de un array, aprovechando que “conoce” el código en C que lo generó:

⁵Completamente equivalente, pues, hubiese sido teclear el comando `x/40bx 0x601040`.

```
(gdb) p array
$3 = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

Ejecución paso a paso del programa: Es interesante que ejecutemos ahora el programa de forma controlada para observar el comportamiento dinámico del mismo. Para ello, listamos el código (comando `list`) y ponemos un punto de ruptura, por ejemplo, dentro del cuerpo del bucle `for` (comando `break 8`), justo cuando ya se va a establecer a -1 el primer elemento del array (`array[0]`). Entonces comenzamos la ejecución (comando `run`):

```
(gdb) list main
1 #define ARRAY_SIZE 10
2
3 int array[ARRAY_SIZE] = {10,9,8,7,6,5,4,3,2,1};
4
5 int main() {
6     int i;
7     for(i=0; i < ARRAY_SIZE; i++) {
8         array[i] = -1;
9     }
10 }
(gdb) break 8
Punto de interrupción 1 at 0x4004b1: file arrayinit.c, line 8.
(gdb) run
Starting program: [...]arrayinit
Breakpoint 1, main () at arrayinit.c:8
8     array[i] = -1;
```

Una vez el programa se detiene en el *breakpoint*, podemos inspeccionar lo que queramos, tanto los datos (comandos `print array` o `x/40bx array`) como los registros (comando `info registers`) o el propio código ensamblador (comando `disassemble`). Si ejecutamos la sentencia en la línea 8 con el comando `next` y a continuación volvemos a visualizar el contenido del array en memoria, veremos que el primer elemento ahora vale -1, cuya representación en complemento es una ristra de bits de 32 unos.

```
(gdb) next
7     for(i=0; i < ARRAY_SIZE; i++) {
(gdb) print array
$3 = {-1, 9, 8, 7, 6, 5, 4, 3, 2, 1}
(gdb) x/40bx array
0x601040 <array>:      0xff  0xff  0xff  0xff  0x09  0x00  0x00  0x00
0x601048 <array+8>:      0x08  0x00  0x00  0x00  0x07  0x00  0x00  0x00
0x601050 <array+16>:     0x06  0x00  0x00  0x00  0x05  0x00  0x00  0x00
0x601058 <array+24>:     0x04  0x00  0x00  0x00  0x03  0x00  0x00  0x00
0x601060 <array+32>:     0x02  0x00  0x00  0x00  0x01  0x00  0x00  0x00
```

Es particularmente interesante observar mediante el comando `info registers rip` el valor actual del registro contador de programa (RIP en x86-64). En este punto vale `0x4004ca`, que marca la instrucción en ensamblador donde está parado el programa en este momento. Este punto, la ejecución del programa está detenida tras haber ejecutado la instrucción `movl $0xffffffff,0x601040(,%rax,4)` que escribe del valor entero -1 (32 bits) en la dirección de memoria correspondiente al *i*-ésimo elemento del array. Finalmente, el comando `disable` desactiva todos los *breakpoints* y con el comando `continue` continúa con el programa hasta termina la ejecución:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x00000000004004ad <+0>:      push    %rbp
0x00000000004004ae <+1>:      mov     %rsp,%rbp
0x00000000004004b1 <+4>:      movl    $0x0,-0x4(%rbp)
0x00000000004004b8 <+11>:     jmp     0x4004ce <main+33>
0x00000000004004ba <+13>:     mov     -0x4(%rbp),%eax
0x00000000004004bd <+16>:     cltq
0x00000000004004bf <+18>:     movl    $0xffffffff,0x601040(,%rax,4)
=> 0x00000000004004ca <+29>:     addl    $0x1,-0x4(%rbp)
0x00000000004004ce <+33>:     cmpl    $0x9,-0x4(%rbp)
0x00000000004004d2 <+37>:     jle     0x4004ba <main+13>
0x00000000004004d4 <+39>:     pop     %rbp
```



```
0x00000000004004d5 <+40>:    retq
End of assembler dump.
(gdb) info registers rip
rip             0x4004ca 0x4004ca <main+29>
(gdb) disable 1
(gdb) continue
Continuando.
[Inferior 1 (process 31098) exited with code 011]
```

Ojo, porque toda la memoria se *resetea* al terminar (vuelve al estado inicial). Si queremos ver el estado final, conviene poner un punto de ruptura justo antes de la finalización del programa, esto es, en el símbolo `}` con el que termina la función `main()`.

B5.1.8. Ejercicios a realizar durante la sesión

Por grupos de como máximo dos personas, y haciendo uso cada grupo de un PC con sistema operativo Linux, llevar a cabo los siguientes ejercicios:

1. Vuelve a arrancar `gdb` con el programa `globals`, colocándole un punto de ruptura justo al comienzo (comando `b main`). Ejecuta entonces el programa paso a paso (es decir, de sentencia en sentencia en lenguaje C; usar para ello el comando `step`). Para cada iteración del bucle `for`, observa los sucesivos cambios en memoria tanto del vector `array` como de la variable `i`, con los comandos `p i`, `p array` y `x/40bx array`. ¿Cuál es la dirección de memoria del último elemento del array?
2. Modifica el programa `globals.c` para calcular la suma de los elementos del array y mostrarla por pantalla antes de ejecutar el bucle existente. Para ello, incluye la librería de funciones de entrada/salida al comienzo del fichero (`#include <stdio.h>`), y añade el siguiente código justo a continuación de la declaración de la variable `i` (`int i`):

```
int sum;
for(i=0; i < ARRAY_SIZE; i++) {
    sum += array[i];
}
printf("Suma de los elementos: %d\n", sum);
```

Una vez hecho esto, recompila `globals.c` con información de depuración y ejecútalo para ver el resultado impreso por pantalla.

3. Establece un punto de ruptura en el cuerpo del bucle `for` añadido en el paso anterior. Ejecuta paso a paso dos iteraciones de dicho bucle, mostrando el valor de la variable `sum` en cada iteración (`print sum`). A continuación, utiliza el comando `set array[5] = -2` para establecer al valor -2 el sexto elemento del array, y observa el resultado mirando la memoria con `x/40bx array`. Sabiendo la dirección en memoria del último elemento del array, modifica de nuevo la memoria usando `set {int} <direccion> = -2`. Finalmente, ejecuta el programa hasta su terminación. ¿Cual es el resultado de la suma impreso por pantalla?
4. Modifica de nuevo el programa `globals.c` para que el tipo de datos de los elementos del array sea entero corto (`short int` en lugar de `int`). ¿Cuál es el nuevo tamaño del array? Regenera adecuadamente el ejecutable `globals` y trácéalo de nuevo con `gdb`, poniendo un punto de ruptura al comienzo del procedimiento principal (`b main`) para después observar la disposición de los datos del array en memoria (`x/20bx array`), prestando especial atención a la nueva disposición de los elementos. ¿Cuál es ahora la dirección de memoria del último elemento del array? ¿Y la del elemento *i*-ésimo?
5. [Librerías] Modifica de nuevo el programa `globals.c` para llamar a una función de la librería matemática. Para ello, añade al comienzo del fichero el correspondiente `include` de la librería `math.h`, y añade un nuevo bucle al comienzo del `main` para establecer el elemento *i*-ésimo del array al valor 3^i :

```
for(i=0; i < ARRAY_SIZE; i++) {  
    array[i] += pow(3,i);  
}
```

Recompila entonces dos versiones del programa (estática y dinámica) con los respectivos comandos:

```
$ gcc-4.8 -g -static globals.c -o globals.static -lm  
$ gcc-4.8 -g globals.c -o globals.dynamic -lm
```

Compara los tamaños de los ficheros generados con `ls -l` y comprueba después con `ldd` sus respectivas naturalezas estática y dinámica, y, en este último caso, las librerías dinámicas con las que el nuevo ejecutable está enlazado.

B5.1.9. Apéndice: Instalación del gcc-4.8 en Ubuntu 18.04

La instalación es muy sencilla, se trata simplemente de abrir un terminal de comandos y en él ejecutar los siguientes comandos (se nos solicitará la palabra de paso como administrador):

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/ppa  
$ sudo apt-get update  
$ sudo apt-get install gcc-4.8 g++-4.8
```

B5.2. Boletín 2: Traducción de C a ensamblador del Intel x86-64

B5.2.1. Objetivos

Esta sesión estará dedicada al estudio del proceso de traducción de un lenguaje de alto nivel a un lenguaje ensamblador. En concreto, ilustraremos dicho proceso con un sencillo, pero a la vez completo ejemplo, consistente en la traducción de un pequeño programa escrito en C al ensamblador nativo de los PCs, basado en el ISA Intel x86-64.

Para la realización de esta práctica se asume que el alumno posee unos conocimientos mínimos del manejo de Linux desde la línea de comandos, adquiridos en sesiones anteriores.

B5.2.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento, en grupos máximos de dos personas por PC, de los pasos expuestos en el ejemplo del boletín (simplemente replicándolos y observando los resultados).
2. Realización, en grupos máximos de dos personas por PC, de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos.

B5.2.3. EJEMPLO 1: Segmentos de datos y código. Instrucciones aritmético-lógicas y movimiento de datos.

Compilación directa y ejecución del programa

Obtener el fichero fuente `aritmetica.c` del material proporcionado a través del Aula Virtual para este boletín. Dicho fichero contiene el siguiente programa en C:

```
int myVar = 3;
int main() {
    myVar += 5;           // Suma 5 (3 + 5 = 8)
    myVar = myVar << 2;    // Desplaza 2 bits a la izquierda (8 << 2 = 32)
    myVar = myVar * 2;     // Multiplica por 2 (32*2 = 64)
    --myVar;              // Resta 1 (64-1 = 63)
    return myVar;         // Valor devuelto como resultado: 63
}
```

En primer lugar, simplemente compilaremos el programa, para obtener un fichero ejecutable llamado `aritmetica`. Utilizamos para ello el siguiente comando:

```
$ gcc-4.8 aritmetica.c -o aritmetica
```

Comprobamos que, efectivamente, se ha generado un fichero `aritmetica`, con los permisos de ejecución adecuados, y a continuación simplemente ejecutamos dicho programa:

```
$ ./aritmetica
$ echo $?
63
```

Si bien dicho programa intencionadamente no imprime ningún mensaje por pantalla al ejecutarse, podemos comprobar que el valor devuelto por `main` es el esperado mostrando la variable de entorno `$?`⁶.

⁶La variable de entorno `$?` muestra el *exit status* del último comando ejecutado por el shell. Para los programas que terminan mediante la sentencia `return` en el procedimiento `main`, el *exit status* contiene los 8 bits menos significativos del valor devuelto por `main`.

Generación de código ensamblador mediante gcc

A continuación, vamos a volver a usar el `gcc`, pero en este caso con la opción `-S`, para generar no un ejecutable, sino el correspondiente fichero en lenguaje ensamblador del Intel x86-64. Además, vamos a generar dos versiones del código ensamblador, una sin optimizar (`-O0`) y otra optimizado (`-O1`), y después ver las diferencias más notables.

```
$ gcc-4.8 aritmetica.c -O0 -fno-asynchronous-unwind-tables -S -o aritmetica_gcc-no-opt.s
$ gcc-4.8 aritmetica.c -O1 -fno-asynchronous-unwind-tables -S -o aritmetica_gcc-opt.s
```

En cada caso, el resultado de la compilación es un nuevo fichero de texto ASCII, cuyo contenido puede consultarse directamente con un editor de texto. En ambos ficheros vemos que el código ensamblador generado por el compilador puede variar ostensiblemente en función del nivel de optimización indicado con la opción `-O0, 1, 2, 3`. Sin necesidad de entrar en los detalles, podemos observar a simple vista la gran diferencia en el número de instrucciones del ensamblador generado sin optimizaciones `aritmetica_gcc-no-opt.s` (17) y con ellas `aritmetica_gcc-opt.s` (4). Mientras que la versión sin optimizar lee/escrive la variable `myVar` de/a memoria en cuatro ocasiones (una por cada operación aritmético-lógica), la versión optimizada no sólo lo reduce a una lectura y una escritura, sino que además traduce las cuatro sentencias en C con operaciones aritméticas a una única instrucción en ensamblador.

Independientemente del nivel de optimización, el código ensamblador generado automáticamente por un compilador es por lo general mas difícilmente comprensible que la traducción que pueda llevar a cabo un programador. Por esta razón, para este boletín de introducción al lenguaje ensamblador vamos a hacer uso de traducciones *manuales* a ensamblador manual de programas en lenguaje C muy sencillos, en lugar de apoyarnos el código generado automáticamente por el compilador.

Traducción manual de C a ensamblador: Segmento de código y segmento de datos. Instrucciones aritmético-lógicas y de movimiento de datos.

El código fuente en C del fichero `aritmetica.c` ha sido traducido de forma manual al código ensamblador que vemos en fichero `aritmetica_manual.s`, con el fin de mejorar la legibilidad y facilitar el aprendizaje de la tarea de traducción. Su contenido es el siguiente:

```
#### Segmento de datos (variables globales del programa)
.data
myVar:
    .long    3          # Variable de tipo entero (tamaño: 4 bytes)
                        # con valor inicial 3

#### Segmento de código (instrucciones del programa)
.text
.globl main

# Procedimiento principal, llamado por el cargador del SO (loader)
main:
    movl     myVar, %eax    # Lee la variable myVar de memoria y
                          # pone su valor en el registro EAX
    addl     $5, %eax       # Suma 5 a EAX
    sall     $2, %eax       # Desplaza EAX 2 bits a la izquierda
    movl     $2, %edx       # Carga la constante 2 en el registro EDX
    imull    %edx           # Multiplica EDX*EAX, producto en EDX:EAX
    decl     %eax           # Resta uno al valor de EAX
    movl     %eax, myVar    # Escribe el valor de EAX en memoria
    ret                  # Termina el procedimiento main,
                          # regresa al invocador
```

En primer lugar, se observa la declaración de una primera parte del programa dedicada al segmento de datos (que comienza con la directiva `.data`). En ella podemos distinguir claramente la variable `myVar` (referida simbólicamente por la etiqueta `myVar:`), de tamaño 4 bytes (directiva `.long`) e inicializada con el valor 3.

A continuación viene el segmento con el código en ensamblador correspondiente al programa, comenzando con la directiva `.text`. En dicho segmento distinguimos en primer lugar la función principal `main`, que comienza en

la etiqueta `main:` y acaba en la primera instrucción `ret`. En el caso particular de este programa, todas las instrucciones que preceden a `ret` son de tipo aritmético-lógico (suma, multiplicación, desplazamiento de bits, decremento) y de movimiento de datos (llevar datos desde memoria a los registros del procesador, y viceversa, o cargar valores constantes en dichos registros).

B5.2.4. EJEMPLO 2: Instrucciones de salto condicional e incondicional. Acceso a arrays.

Compilación directa y ejecución del programa

Obtener el fichero fuente `array.c` del material proporcionado a través del Aula Virtual para este boletín. Dicho fichero contiene el siguiente programa en C, que devuelve la suma de los elementos de un array de 5 enteros con ciertos valores iniciales:

```
#define ARRAY_SIZE 5

int array[ARRAY_SIZE] = {10,20,30,40,50};

int main() {
    int i, sum = 0;
    for(i=0; i < ARRAY_SIZE; i++) {
        sum += array[i];
    }
    return sum;
}
```

Al igual que antes, compilamos el programa para obtener un fichero ejecutable llamado `array`, ejecutamos dicho programa y comprobamos el valor devuelto mediante la variable de entorno `$?`.

```
$ gcc-4.8 array.c -o array
$ ./array
$ echo $?
150
```

Traducción manual de C a ensamblador: bucles 'for' y acceso a arrays.

El código fuente en C del fichero `array.c` ha sido traducido de forma manual al código ensamblador que vemos en fichero `array_manual.s`, cuyo contenido es:

```
#### Segmento de datos (variables globales del programa)
.data
array:
    .long    10
    .long    20
    .long    30
    .long    40
    .long    50

#### Segmento de código (instrucciones del programa)
.text
.globl  main

# Procedimiento principal, llamado por el cargador del SO (loader)
main:
    # Variables locales a main:
    # i -> registro ESI
    # sum -> registro EAX

    movl    $0, %eax        # Inicializa sum a 0

    ##### Bucle for:

    movl    $0, %esi        # Inicializa i a 0
inicio_for:
    # Comprobación de la condición
    cmpl    $5, %esi        # Compara ESI con 5 (tamaño del array)
```

```

    jge     fin_for      # Salta a fin_for si cmp anterior fue mayor o igual

    movl    array(,%esi,4), %ecx    # Lee variable en la dirección de memoria
                                    # array+4*i y lo guarda en el registro ECX
    addl    %ecx, %eax    # sum += array[i]

    # Incremento de la variable de control
    inc     %esi         # i++
    # Regresa al inicio del bucle
    jmp     inicio_for

fin_for:
    # EAX contiene sum (valor retornado por main)
    ret                     # Termina el procedimiento main,
                          # regresa al invocador

```

En primer lugar, se observa en el segmento de datos la declaración del array de 5 posiciones inicializado con los valores {10, 20, 30, 40, 50} (directivas `.long` seguidas del valor adecuado), con la etiqueta `array`:

A continuación, en el segmento de código encontramos la traducción de un bucle “for”: En primer lugar, tenemos la inicialización de la variable de control del bucle `i`, que se pone a 0 antes de comprobar por primera vez la condición del bucle. Tras ello tenemos el bucle en sí (entre las etiquetas `inicio_for` y `fin_for`), que se divide en:

1. Comprobación de la condición de continuación del bucle, mediante una instrucción de comparación seguida de un salto condicional a la etiqueta `fin_for`, que se producirá cuando el registro que alberga la variable `i` (ESI) no sea menor que 5.
2. Cuerpo del bucle: Lectura del elemento `i`-ésimo del vector `array` situado en memoria (en el segmento de datos), mediante la instrucción `movl array(,%esi,4), %ecx`, que lee el entero (4 bytes) en la dirección de memoria `array+esi*4 = array+i*4`, y lo guarda en el registro `%ecx`. Nótese que la dirección `array` marca el comienzo de dicho vector, y que cada entero ocupa exactamente 32 bits = 4 bytes. El elemento del array leído en cada iteración se acumula en la variable local `sum` (registro EAX).
3. Incremento en una unidad de la variable de control `i` que dirige el bucle `for`, mediante la instrucción `inc %esi`.
4. Vuelta al comienzo del bucle mediante la instrucción de salto incondicional `jmp inicio_for`, para comprobar nuevamente la condición con el nuevo valor de la variable de control.

B5.2.5. EJEMPLO 3: Llamadas a subrutinas. Paso de parámetros y valor de retorno

Compilación directa y ejecución del programa

Obtener el fichero fuente `funcion.c` del material proporcionado a través del Aula Virtual para este boletín. Dicho fichero contiene el siguiente programa en C, que devuelve la suma de los elementos de un función de 5 enteros con ciertos valores iniciales:

```

int funcion_resta(int minuendo, int sustraendo) {
    int resta = minuendo - sustraendo;
    return resta;
}

int main() {
    int resultado = 100;
    resultado += funcion_resta(50, 30); // 100+(50-30)
    ++resultado; // 120 + 1
    return resultado; // 121
}

```

Compilamos el programa para obtener un fichero ejecutable llamado `funcion`, ejecutamos dicho programa y comprobamos el valor devuelto mediante la variable de entorno `$?`.

```

$ gcc-4.8 funcion.c -o funcion
$ ./funcion
$ echo $?
121

```

Traducción manual de C a ensamblador: Llamadas a subrutinas, paso de parámetros y retorno de valores.

El código de `funcion.c` ha sido traducido manualmente al siguiente código ensamblador (`funcion_manual.s`):

```
#### Segmento de código (instrucciones del programa)
.text
.globl main

funcion_resta:
# Función: int funcion_resta(int minuendo, int sustraendo):
# Recibe como parámetros dos enteros vía registros:
# - 1er argumento (en EDI): Minuendo
# - 2º argumento (en ESI): Sustraendo
# Valor retornado (en EAX): La resta (minuendo-sustraendo)

# Variable local 'resta' en EAX
movl    %edi, %eax    # Copio minuendo a EAX
subl    %esi, %eax    # EAX = EAX - ESI
ret     # EAX contiene el valor a devolver

# Procedimiento principal, llamado por el cargador del SO (loader)
main:
# Variables locales a main:
# resultado: EBX
movl    $100, %ebx    # Inicializa 'resultado'

movl    $50, %edi     # Establece primer argumento
movl    $30, %esi     # Establece segundo argumento
call    funcion_resta # Llama a funcion_resta
# EAX: Valor devuelto por el procedimiento
addl    %ebx, %eax    # EAX = EAX + EBX
inc     %eax          # ++resultado
# EAX contiene sum (valor retornado por main)
ret     # Termina el procedimiento main,
        # regresa al invocador
```

Los sistemas operativos tipo Unix (como Linux) que corren en arquitecturas Intel x86-64 siguen la convención de llamadas del ABI System V AMD64, según la cual los primeros seis parámetros de tipo entero (o puntero) se pasan a la subrutina invocada mediante los registros RDI⁷, RSI, RDX, RCX, R8 y R9, en ese preciso orden. Las subrutinas que exceden dicho número de parámetros utilizan la pila para pasar el resto de parámetros.

B5.2.6. Ejercicios a realizar durante la sesión

Por grupos de como máximo dos personas, y haciendo uso cada grupo de un PC con sistema operativo Linux, llevar a cabo los siguientes ejercicios:

1. Ensambla el programa `aritmetica_manual.s` para generar un binario ejecutable con información de depuración:

```
gcc-4.8 -g aritmetica_manual.s -o aritmetica_manual
```

Carga el ejecutable obtenido con `gdb` y usa el comando `layout regs` para, respectivamente, pasar a la vista de ensamblador y mostrar el contenido de los registros del procesador. Averigua a partir del código ensamblador la dirección de memoria donde está `myVar`, y visualiza su valor inicial con `x/4bx <direccion>` y luego con `print (int)myVar`. Después, pon un breakpoint al comienzo (`b main`) y lanza el programa (`run`). Ve ejecutando paso a paso con `stepi` para avanzar de instrucción en instrucción ensamblador, visualizando en cada paso el contenido de los registros afectados tras cada instrucción. Vuelve a mostrar el contenido de la variable justo antes de ejecutar la instrucción `ret`.

⁷Los programas de ejemplo de este boletín manejan variables de tipo entero (32 bits) y por tanto operan con los 32 bits de menos peso de cada registro Intel x86-64 (64 bits), accesibles con el prefijo “E”: EDI, ESI, EDX, etc. El prefijo “R” indica que se accede al registro completo.

2. Repite los pasos del ejercicio anterior, pero ahora usando el programa `array_manual`. Visualiza el contenido de la memoria ocupada por el array con `x/20bx <direccion>` y observa cómo en cada iteración del bucle, la instrucción `mov` el *i*-ésimo elemento del array y lo guarda en el registro ECX. Comprueba los registros que se modifican al ejecutar cada instrucción, ¿en cuál se almacena el resultado de la comparación previa al salto condicional?
3. Repite los pasos del ejercicio anterior, pero ahora usando el programa `funcion_manual`, visualizando el contenido de los registros EDI, ESI y EAX, tras cada instrucción.
4. [Pila] Repite los pasos del ejercicio anterior, pero ahora usando el programa `llamadas`. Ejecuta paso a paso, visualizando el contenido de los registros RIP y RSP, y de la dirección de memoria apuntada por RSP (`x/1qx $rsp`), inmediatamente antes y después de cada instrucción `call` y `ret`. ¿Qué acciones realizan cada una de estas dos instrucciones con respecto a RIP y RSP? ¿Dónde se guarda la dirección de retorno al ejecutar un `call`? ¿De dónde obtiene la instrucción `ret` la dirección a la que debe saltar? `x/1qx $rsp` ¿Qué ocurriría si la dirección de retorno (es decir, la dirección de la instrucción inmediatamente posterior al `call`) se guardase en un registro en vez de la pila?
5. [Pila] Compila el programa `stackoverflow.c` con información de depuración, y lánzalo con `gdb stackoverflow`. Después, pon un breakpoint al comienzo (`b main`) y lanza el programa (`run`). Ve ejecutando paso a paso con `step` para avanzar de sentencia C en sentencia C. Tras cada llamada a una función, usa el comando `backtrace` para ver la pila de llamadas. Cuando la pila de llamadas llegue a 10, utiliza el `continue` para dejar que el programa continúe su ejecución hasta que se detenga, y entonces vuelve a visualizar la pila de llamadas. A la vista de la salida del comando `backtrace -1`, ¿qué crees que ha ocurrido? Averigua el PID del proceso `stackoverflow` con `ps a` y con el comando `pmap` averigua el tamaño de su pila. Finalmente, comprueba que coincide con el límite establecido con `ulimit -a`.
6. [Pila] [Avanzado] Compila el programa `locales.c` para obtener un fichero ejecutable llamado `locales`. Ejecútalo en un terminal introduciendo cualquier número de hasta tres cifras, hasta llegar a ver el mensaje “Pulsa INTRO para continuar”. En este momento, en otro terminal, lanza el comando `pmap` seguido del PID del proceso que está ejecutando el programa `locales`. El comando `pmap` permite ver el mapa de memoria de un proceso, incluyendo el tamaño en memoria ocupado por la pila. ¿Qué tamaño tiene el área de memoria asignada a la pila?
Ejecuta de nuevo el programa, pero esta vez introduce el número *mágico* 1234 cuando el programa `locales` lo solicite. ¿Qué diferencias observas en la cantidad de memoria utilizada por la pila del proceso? Mirando el código fuente del `main`, ¿qué funciones se ejecutan dependiendo del número introducido por teclado? ¿En qué zona de memoria se almacenan las variables `un_array` y `otro_array` que aparecen en el programa?

Nota importante:

En la distribución de Ubuntu instalada en los laboratorios de la Facultad (y por tanto también en la de *eva.um.es*) el comando `gcc-4.8` está renombrado como `gcc48`. Tener esto en cuenta de cara a poder probar en dichas distribuciones todos los ejemplos contenidos tanto en los videotutoriales como en este boletín que hagan referencia a dicho comando.