Universidad de Murcia

Facultad de Informática

Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

TÍTULO DE GRADO EN
CIENCIA E INGENIERÍA DE DATOS

Fundamentos de Computadores

Práctica 8: Lenguajes del computador:
alto nivel, ensamblador y máquina

Boletines de prácticas

CURSO 2022 / 23

Índice general

Boletines de prácticas

B8.1. Boletín 1. Generación de código, enlazado, carga en memoria y ejecución de programas.

B8.1.1. 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 bibliotecas 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.

B8.1.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento de los pasos expuestos en el ejemplo del boletín.
2. Realización de forma individual de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos.

B8.1.3. La aplicación web *Compiler Explorer*

Compiler Explorer es una aplicación web de código abierto disponible en `https://godbolt.org`, que permite escribir y compilar el código fuente de forma interactiva, todo desde la comodidad del navegador web. Permite elegir entre una amplia variedad de compiladores (gcc, clang, etc.), y tiene soporte para un gran número de lenguajes de programación. Además, *Compiler Explorer* también soporta lenguajes interpretados como Python, de manera que puedes observar cómo se compila a *bytecode* para ser ejecutado en la máquina virtual de Python. Una de sus características clave es que nos indica exactamente a qué instrucciones se traduce cada línea del código fuente, como podemos ver en la Figura I.1. En dicha figura, vemos el código fuente en C del típico programa "hello world" junto con su traducción a código ensamblador del ISA x86-64 por parte del compilador `gcc` (versión 12.2), donde mediante colores se nos indica la traducción correspondiente de cada sentencia en C a instrucciones en ensamblador.

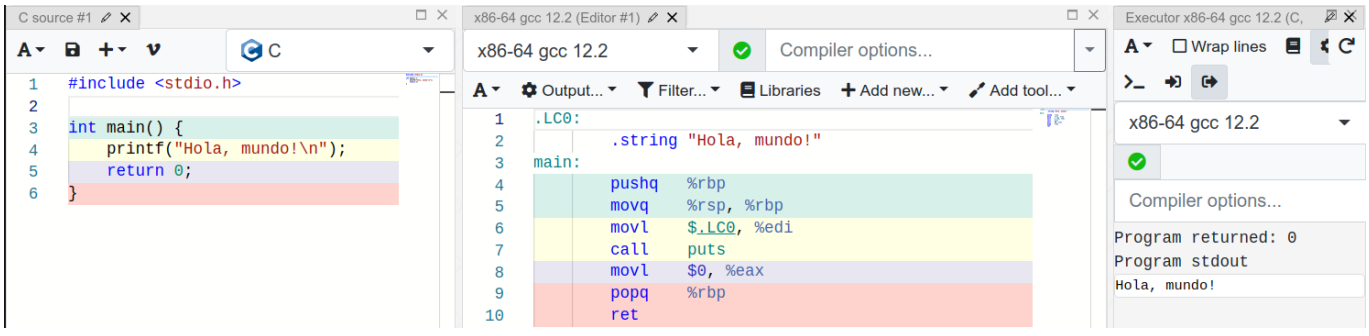

Figura I.1: Traducción de código fuente en C a ensamblador de x86-64 mediante GCC (*Compiler Explorer*).

B8.1.4. Traducción de código de alto nivel a lenguaje ensamblador y lenguaje máquina

Utilizaremos la aplicación web *Compiler Explorer* para mostrar la jerarquía de traducción y los distintos lenguajes del computador. Partiremos del fichero de código fuente en C `hola.c` disponible como recurso en el Aula Virtual. Una vez en el navegador, seleccionamos el lenguaje C en el panel izquierdo de código fuente (*source*) y a continuación copiamos y pegamos el código de `hola.c` en dicho panel. En el panel derecho podemos elegir el compilador a utilizar, si bien por ahora utilizaremos el compilador por defecto (`gcc` compilando para el ISA x86-64). En el botón *Add new* podemos añadir un nuevo panel del tipo *Executor from this* para observar el resultado de la ejecución del programa. En el botón de *Output* del panel del compilador podemos elegir el formato de la salida: en nuestro caso, marcamos las opciones para generar código binario, ejecutar el código, y desmarcamos la opción de usar la sintaxis de Intel. En este boletín, usaremos la sintaxis de AT&T, que es la que utiliza el compilador `gcc` por defecto, y en la cual el formato general de una instrucción en ensamblador x86-64 es: *mnemonic source, destination*.

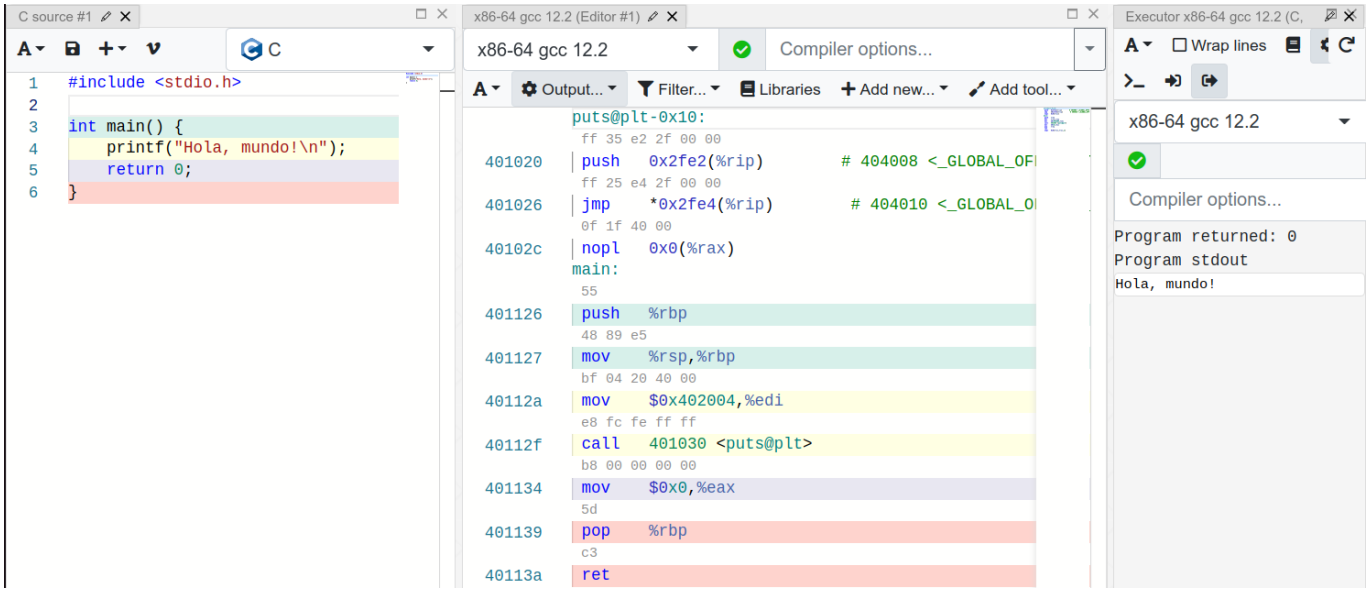

Figura I.2: Traducción de C a código ensamblador y código máquina de x86-64.

En la Figura I.2 podemos ver que en el repertorio de instrucciones del ISA x86-64 encontramos instrucciones `push` y `pop` (utilizadas para guardar/recuperar datos en/de la *pila*), instrucciones `mov` (para copiar datos de una ubicación a otra, p.ej., de un registro a otro), e instrucciones `call` y `ret` (para llamar/regresar de procedimientos). Encima de cada instrucción en ensamblador podemos ver su codificación en lenguaje máquina, mientras que a su izquierda tendríamos el contador de programa (su dirección en memoria). Vemos que en x86 hay instrucciones que se codifican en un único byte (p.ej., `0x55` codifica la instrucción `push %rbp`), mientras que otras instrucciones ocupan varios bytes. La mayoría de instrucciones utilizan como operando algún registro (p.ej., `%rbp`, `%eax`, `%edi`, etc.), mientras que algunas instrucciones tienen como operandos valores constantes que aparecen codificados en la propia instrucción (*inmediatos*). Por ejemplo, la instrucción `mov $0x0,%eax`, que establece el registro de 32 bits `EAX` con el valor 0, se codifica en cinco bytes (`b8 00 00 00 00`), de los cuales los cuatro últimos bytes codifican el valor entero al que se establecerá el registro. Por su parte, la instrucción `call puts` transfiere la ejecución a la función de la biblioteca de C `puts` (imprimir una cadena por pantalla). El convenio de llamadas que sigue Linux para x86-64[1] dictamina que el registro EDI se debe utilizar para pasar el primer parámetro a una función. Así, vemos que con anterioridad a la instrucción `call puts` se establece el registro EDI con el valor del parámetro pasado a la función, en este caso la dirección en memoria de la cadena de caracteres "Hola, mundo!" que queremos imprimir. El mismo convenio establece que las funciones utilizan el registro EAX para devolver un valor antes de ejecutar `ret`; como vemos, la línea 5 del código fuente en C se traduce (en parte) por el establecimiento del registro EAX al valor 0, que es el valor retornado por el programa, tal y como podemos ver en el panel "ejecutor" del lado derecho.

[1] https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

B8.1.5. Generación de código ensamblador

Vamos a ver ahora cómo llevar a cabo los diferentes pasos en la jerarquía de traducción en un entorno Linux, utilizando para ello el compilador GCC. Si bien sus siglas originalmente provienen de *GNU C Compiler*, hoy en día GCC es una colección de compiladores con soporte para múltiples lenguajes de programación. GCC es un componente clave de la cadena de herramientas de GNU y el compilador estándar para la mayoría de los proyectos relacionados con GNU y el núcleo de Linux.

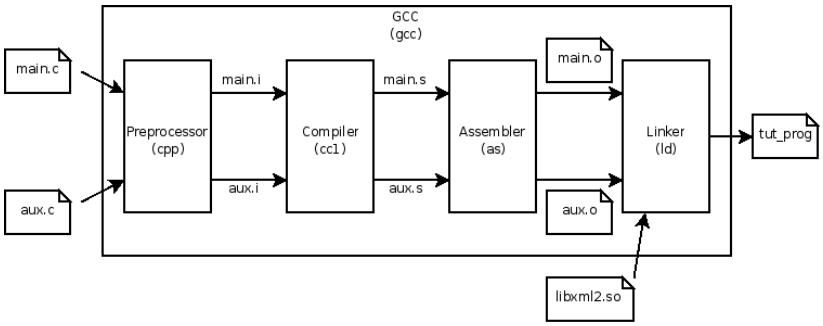

Figura I.3: Proceso de compilación con GCC, incluyendo programas traductores asociados.

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

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

La opción `-o` de `gcc` 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!
```

Al invocar `gcc` pasando como parámetros ficheros de código fuente en C y sin especificar ninguna opción adicional, se llevan a cabo por defecto todas las etapas del proceso de traducción hasta generar un programa ejecutable: compilación, ensamblado y enlazado. Sin embargo, es posible realizar mediante `gcc` las sucesivas etapas de traducción. Por ejemplo, con la opción `-S` podemos realizar únicamente la etapa de compilación, generando el correspondiente fichero con lenguaje ensamblador del Intel x86-64.

```
$ gcc hola.c -S
```

El resultado de la compilación es un nuevo fichero de texto ASCII llamado `hola.s`, cuyo contenido más relevante se muestra a continuación. Vemos que el contenido del fichero generado es similar al mostrado en la Figura I.1, salvo por algunas directivas adicionales (fácilmente identificadas ya que empiezan por el carácter "."). El compilador genera estas directivas para controlar el proceso de ensamblado; por ejemplo, la directiva `.text` marca el inicio de la sección de código ejecutable, en la que se encuentran las instrucciones.

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

        .text
        [...]
main:
        [...]
        movq    %rax, %rdi
        call    puts@PLT
        [...]
```

B8.1.6. Generación de código objeto

Ahora vamos a ensamblar el fichero de código ensamblador `hola.s` producido en el paso anterior, con el fin de generar el correspondiente fichero de código objeto `hola.o`. Aunque en realidad el programa ensamblador es independiente (llamado `as` en el caso del *toolkit* de `gcc`), resulta más cómodo utilizar `gcc` como *front-end* para hacer cualquiera de las etapas de la traducción. Así, con este sencillo comando estamos invocando al ensamblador:

```
$ gcc -c hola.s
```

Como podemos comprobar con el comando `less`, el fichero `hola.o` generado por el comando anterior ya no es un fichero de texto sino un fichero binario que contiene las instrucciones codificadas en lenguaje máquina. Para poder interpretar su contenido debemos utilizar una herramienta llamada *desensamblador*, que realiza la traducción desde código máquina a ensamblador, mediante el siguiente comando:

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

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

```
hola.o:     formato del fichero elf64-x86-64
Desensamblado de la sección .text:
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
```

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 su código máquina (mostrado como secuencia de bytes, en hexadecimal), y finalmente una tercera columna donde se muestra el código ensamblador correspondiente a dicha instrucción.

B8.1.7. Enlazado

Al igual que con el ensamblador, el programa `ld` del *toolkit* se encarga hacer el enlazado de código(s) objeto(s) y biblioteca(s) en un sólo fichero ejecutable. No obstante, el propio compilador `gcc` se puede encargar de llamarlo por nosotros, ya que el comando de enlazado usado internamente es más complicado. Así que, para generar el ejecutable `hola` a partir del anterior fichero `hola.o`, simplemente podemos llamar a `gcc` así:

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

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

```
$ ldd hola
```

La salida generada por `ldd` será 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 biblioteca dinámica correspondiente, y la dirección virtual de nuestro programa a la que las rutinas allí contenidas son mapeadas. En particular, la biblioteca que aquí más nos interesa es la biblioteca estándar de C (`libc`), que contiene, entre otras muchas utilidades, la función `puts` usada por nuestro programa. Las otras dos bibliotecas (`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 biblioteca que gestiona la posibilidad de carga dinámica de bibliotecas en memoria, para ser compartidas entre varios programas.

Puesto que el ejecutable generado enlaza con bibliotecas dinámicas, su tamaño tiende a ser bastante pequeño (en torno a los 15KB, dependiendo también de la versión concreta del `gcc` utilizada). Podemos comprobarlo con el comando `ls -l`.

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

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

Esta vez podemos comprobar con `ldd hola.static` que el ejecutable generado no enlaza con ninguna biblioteca dinámica, pero a cambio sí que se tiene que pagar un precio en el tamaño del ejecutable. En este caso, el fichero ocupa en torno a 900KB, dependiendo de las versiones tanto del compilador como de la biblioteca `libc` con la que haya enlazado. Para conocer de primera mano la razón por la que el ejecutable resultante tiene un tamaño mucho mayor, podemos llevar a cabo su desensamblado y contar el número de líneas (instrucciones) que contiene:

```
$ objdump -d hola.static | less
```

Si echamos un vistazo por encima al (¡enorme!) listado de código desensamblado generado, podemos incluso localizar las partes del código correspondientes a la función `puts` utilizada. Para ello, podemos buscar la cadena `_IO_puts` en `less` tecleando '`/`' seguido de la cadena a buscar (la tecla "n" pasa a la siguiente ocurrencia). También podemos usar el filtro `wc -l` para contar el número de líneas del desensamblado del ejecutable estático, y compararlo con lo que obtenemos al desensamblar el ejecutable dinámico.

B8.1.8. 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. Sin embargo, en este caso le añadimos la información necesaria para poder *tracearlo*, en nuestro caso, usando el depurador GDB[2]. El depurador GDB nos permitirá cargar nuestro programa en memoria para poder ejecutarlo dentro de un entorno controlado, donde podamos ir ejecutándolo paso a paso, observando los valores de las variables, y tengamos acceso tanto a los registros de la CPU como a las zonas de datos e instrucciones del programa. Para que el ejecutable cuente con la información necesaria para que GDB resulte plenamente útil, ha de ser generado usando la opción `-g` del compilador `gcc`:

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

Una vez generado el ejecutable "traceable", lanzamos el depurador:

```
$ 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 `break` sirve para establecer un punto de ruptura en una línea código o una determinada función, de manera que la ejecución del programa se detendrá en ese punto. Por su parte, el comando `run` sirve para lanzar el programa a ejecución. Una vez en la consola de GDB, podemos activar la interfaz textual (TUI) mediante el comando `layout split`, de forma que nos aparezcan dos paneles: en el superior, el código fuente del programa, y en el inferior, su traducción a código ensamblador, resultado de desensamblar el código máquina contenido en el fichero con el programa ejecutable. La Figura I.4 muestra el resultado de lanzar `gdb` con el programa `hola.static` y activar la interfaz TUI para ver tanto código de alto nivel como el de bajo nivel.

[2]El programa `gdb` es un potentísimo depurador de programas, con infinidad de potencialidades y opciones. En este documento simplemente utilizaremos unas pocas de ellas, con el fin de ilustrar los aspectos más relevantes del proceso de carga y ejecución de programas.

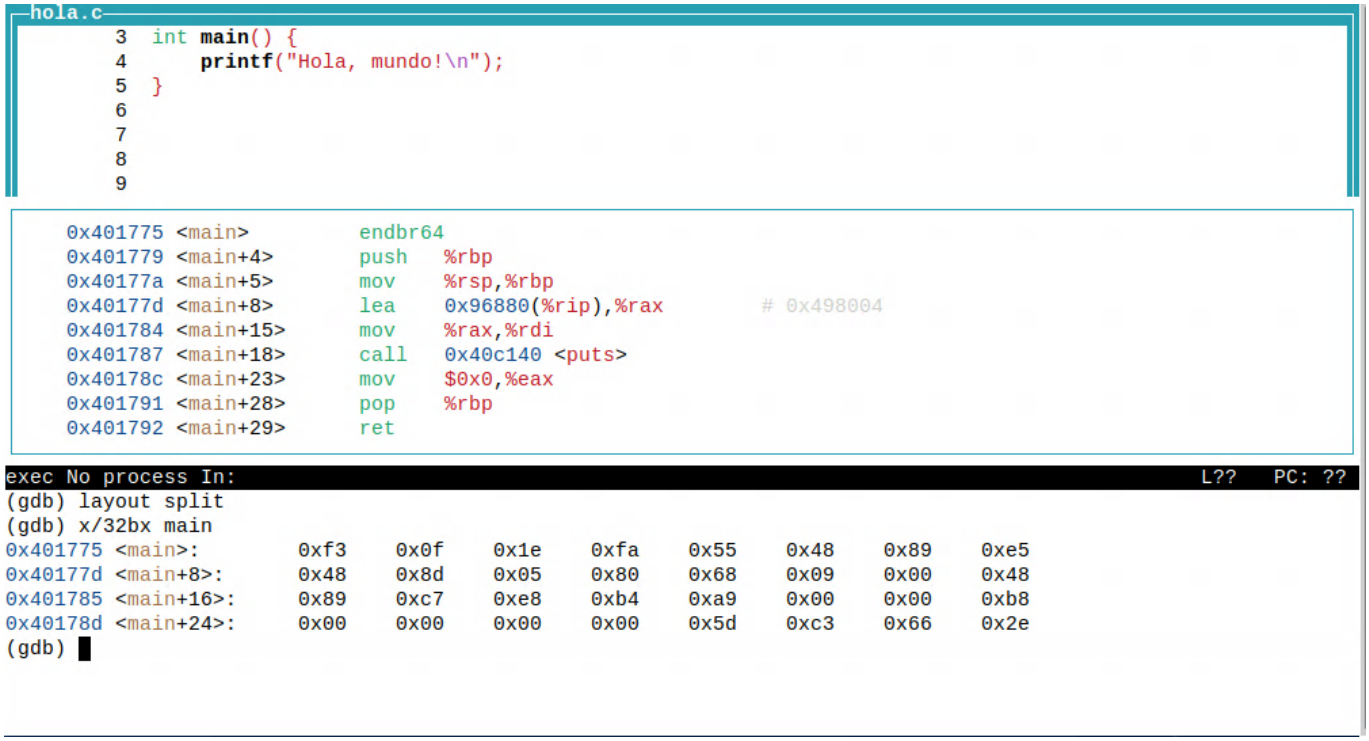

Figura I.4: Vista del depurador GDB en su interfaz TUI, listo para tracear el programa `hola`.

Ubicación del código: Se trata esencialmente del mismo código que se vio anteriormente cuando usamos el comando `objdump` sobre el fichero de código objeto, 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 `0x401775` 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 instrucción `call` en el desplazamiento `main+18`, que en el código objeto de `hola.o` (desplazamiento 0x12) se había codificado 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 `call 40c140 <puts>`. Como podemos comprobar con `disassemble puts`, la dirección memoria `0x40c140` es donde comienza la rutina `puts` una vez ubicada en memoria. Si examinamos el contenido de los 32 bytes en memoria a partir de la dirección donde se ubica `main` (`x/32bx main`), podemos ver las instrucciones en código máquina: en la dirección `0x401787` (`main+18`) empieza la secuencia de bytes `e8 b4 a9 00 00` correspondientes a la instrucción `call puts`. El valor entero inmediato `b4 a9 00 00` codificado en dicha instrucción (en *little endian*) indica la dirección de `puts` (procedimiento llamado), y se codifica de manera relativa a la instrucción siguiente al `call` (0x40178c+0xa9b4 = 0x40c140).

B8.1.9. Ejecución paso a paso de un programa y ubicación de los datos en memoria

Ahora vamos a ejecutar paso a paso el programa, observando cómo cambian los valores de determinados registros del procesador. Para ello, tal como se ilustra en la Figura I.5, en primer lugar ponemos un *breakpoint* (punto de ruptura) al comienzo del programa mediante el comando `break main` y a continuación lanzamos el programa a ejecución (comando `run`). Una vez detenida la ejecución en el *breakpoint*, podemos ejecutar paso a paso el programa. En este caso, lo haremos de instrucción en instrucción (a nivel de código ensamblador) mediante el comando `stepi`

(o `stepi`). Si avanzamos una instrucción ensamblador y mostramos el valor de ciertos registros, veremos que el el registro RIP contiene la dirección de la instrucción que se va a ejecutar a continuación (0x401784, es decir, `main+15`) mientras que el registro RAX, que acaba de ser escrito por la instrucción anterior, contiene el valor que se va a pasar como parámetro a la función `puts`, y que no es otro que la dirección en memoria de la cadena de caracteres que queremos imprimir por pantalla. Así, si examinamos el contenido de los 13 bytes en memoria a partir de la dirección *apuntada* por RAX (x/13bc 0x498004), veremos que efectivamente están almacenados los caracteres 'H', 'o', 'l', etc., correspondientes a la cadena "Hola, mundo!".

```
hola.c
        3  int main() {
B+>     4      printf("Hola, mundo!\n");
        5  }
        6
        7
        8
        9
       10
       11

    0x401775 <main>         endbr64
    0x401779 <main+4>       push   %rbp
    0x40177a <main+5>       mov    %rsp,%rbp
B+  0x40177d <main+8>       lea    0x96880(%rip),%rax        # 0x498004
  > 0x401784 <main+15>      mov    %rax,%rdi
    0x401787 <main+18>      call   0x40c140 <puts>
    0x40178c <main+23>      mov    $0x0,%eax
    0x401791 <main+28>      pop    %rbp
    0x401792 <main+29>      ret

native process 18108 In: main                                    L4    PC: 0x401784
(gdb) break main
Punto de interrupción 1 at 0x40177d: file hola.c, line 4.
(gdb) run
Starting program: /home/alumno/practicas/prac8/code/boletin1/hola/hola.static

Breakpoint 1, main () at hola.c:4
(gdb) stepi
(gdb) info registers rip rax
rip            0x401784            0x401784 <main+15>
rax            0x498004            4816900
(gdb) x/13bc 0x498004
0x498004:       72 'H'  111 'o' 108 'l' 97 'a'  44 ','  32 ' '  109 'm' 117 'u'
0x49800c:       110 'n' 100 'd' 111 'o' 33 '!'  0 '\000'
(gdb)
```

Figura I.5: Vista del depurador GDB, ejecutando paso a paso el programa `hola`.

B8.1.10. 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 = 0;
    while(i < ARRAY_SIZE) {
        array[i] = -1;
        ++i;
    }
}
```

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

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

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
0x4c5100 <array>:       0x0a    0x00    0x00    0x00    0x09    0x00    0x00    0x00
0x4c5108 <array+8>:     0x08    0x00    0x00    0x00    0x07    0x00    0x00    0x00
0x4c5110 <array+16>:    0x06    0x00    0x00    0x00    0x05    0x00    0x00    0x00
0x4c5118 <array+24>:    0x04    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0x4c5120 <array+32>:    0x02    0x00    0x00    0x00    0x01    0x00    0x00    0x00
```

Vemos que la variable `array` está ubicada a partir de la dirección `0x4c5100`[3], que cada elemento del array ocupa 4 bytes (32 bits) por cada entero, los cuales están almacenados en memoria según el esquema de almacenamiento usado por el ISA x86-64 (*little endian*). 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:

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

Ahora vamos a ejecutar el programa de forma controlada para observar el comportamiento dinámico del mismo y observar cómo cambian los valores de las variables en memoria. Para ello, listamos el código (comando `list`) y ponemos un punto de ruptura, por ejemplo, dentro del cuerpo del bucle *while* (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 = 0;
7     while(i < ARRAY_SIZE) {
8         array[i] = -1;
9         ++i;
10     }
(gdb) break 8
Punto de interrupción 1 at 0x4004b1: file globals.c, line 8.
(gdb) run
Starting program: [...]/globals.static
Breakpoint 1, main () at globals.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     while(i < ARRAY_SIZE) {
(gdb) print array
$3 = {-1, 9, 8, 7, 6, 5, 4, 3, 2, 1}
(gdb) x/40bx array
0x4c5100 <array>:       0xff    0xff    0xff    0xff    0x09    0x00    0x00    0x00
0x4c5108 <array+8>:     0x08    0x00    0x00    0x00    0x07    0x00    0x00    0x00
0x4c5110 <array+16>:    0x06    0x00    0x00    0x00    0x05    0x00    0x00    0x00
0x4c5118 <array+24>:    0x04    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0x4c5120 <array+32>:    0x02    0x00    0x00    0x00    0x01    0x00    0x00    0x00
```

[3] Completamente equivalente, pues, hubiese sido teclear el comando `x/40bx 0x4c5100`.

B8.1.11. Ejercicios a realizar durante la sesión

NOTA IMPORTANTE:
Antes de realizar los ejercicios, asegúrate de grabar la sesión con `script -a typescript_prac8_bol1`. Debes empezar cada ejercicio escribiendo el comentario de documentación `### EJERCICIO N ###` . Finalmente, no olvides añadir dicho fichero a tu repositorio-bitácora, respetando la organización de la misma en directorios y subdirectorios. Recuerda responder a cada pregunta tecleando tu respuesta en el mismo terminal (a continuación de los comandos necesarios para cada apartado) precedida por el carácter #, que indica al *shell* que se trata de un comentario (el shell ignora los siguientes caracteres hasta el siguiente final de línea, cuando pulses INTRO). Puedes introducir tu respuesta en mútiples líneas si así lo deseas, empezando cada línea con #.

1. **Llamadas a bibliotecas y llamadas al sistema.**

   a) Vuelve a arrancar `gdb` con el programa `hola.static`, colocándole dos puntos de ruptura en las funciones `puts` y `write` de la biblioteca de C.

   b) Ejecuta el programa (comando `run`) hasta que la ejecución se detenga en la función `puts`. Muestra el valor del registro contador de programa en ese punto (registro RIP en x86-64) con `info registers rip`.

   c) A la vista del código ensamblador de `main` (visualiza con `disassemble main`), ¿cuál fue la última instrucción en ejecutarse antes de que el programa se detuviese en el *breakpoint*?

   d) Mediante el comando `x/10i puts`, examina el contenido de las 10 instrucciones en memoria a partir de la dirección donde comienza `puts`. ¿Qué tamaño tiene la instrucción que se ejecutará a continuación?

   e) ¿Cuál será el valor que tome el registro contador de programa (RIP) tras ejecutar la instrucción actual?

   f) Continúa la ejecución hasta el siguiente punto de ruptura en la función `write`. Muestra la pila actual de llamadas a funciones mediante el comando `backtrace`, ¿qué función es la que ha realizado la llamada a `write`?

   g) ¿Qué función invocada directamente por `puts` ha llevado a la ejecución de `write`?

   h) Mediante el comando `x/10i write`, examina el contenido de las 10 instrucciones en memoria a partir de la dirección donde comienza `write`. ¿Cuántas instrucciones hay antes de la instrucción `syscall`?

   i) Ejecuta ahora el comando `stepi 5`, el cual ejecuta 5 instrucciones máquina, y a continuación visualiza el contenido de los registros RDI, RSI y RDX. Estos tres registros contienen los valores de los tres parámetros que se pasan a la llamada al sistema `write(fichero,buffer,nbytes)`, la cual se usa para escribir en un determinado fichero (dado por su descriptor) el contenido de un buffer memoria, indicando el número de bytes a escribir en el fichero.

   j) Sabiendo que el registro RDI contiene el descriptor del fichero en el que se va a escribir, ¿a qué dispositivo de E/S crees que corresponde dicho descriptor de fichero?

   k) Muestra el contenido de la memoria en la dirección dada por el valor actual del registro RSI mediante `x/13bc DIRECCION`. ¿Qué dato se va a escribir en fichero? ¿Cuántos bytes se va a solicitar escribir?

   l) Ejecuta comando `stepi`, para ejecutar la instrucción `syscall`. ¿Qué ocurre con respecto al comportamiento percibido por el programa tras ejecutar dicha instrucción? ¿Qué código es el encargado de llevar a cabo las acciones que han producido dicho comportamiento?

2. **Variables globales y locales.**

   a) Arranca de nuevo con `gdb` el programa `globals.static`, colocándole un punto de ruptura justo al comienzo (comando `b main`).

   b) Usa el comando `print array` para mostrar el valor de la variable `array` antes de comenzar a ejecutar programa. ¿Qué ocurre si tratas de mostrar el valor de la variable `i`?

c) Lanza el programa a ejecución con `run`, y tras ello repite el apartado anterior. ¿Por qué es posible mostrar el valor de `array` antes incluso de empezar a ejecutar el programa, pero no ocurre lo mismo con la variable `i`?

d) Ejecuta paso a paso usando el comando `step` (de sentencia en sentencia en lenguaje C). Para cada iteración del bucle *while*, observa los sucesivos cambios en memoria tanto del vector `array` como de la variable `i`, con los comandos `print i`, `print array` y `x/40bx array`.

e) ¿Entre qué dos direcciones de memoria se almacena el último elemento del array? Da la respuesta como dirección inicial-final (ambas direcciones incluidas)

3. **Tipos de datos y su tamaño en memoria.**

a) Utiliza el comando `diff -u globals.c globals_long.c` para ver las diferencias entre ambos programas. ¿En qué se diferencian?

b) Compila el programa `globals_long.c` y arranca con `gdb` el ejecutable que has generado.

```
$ gcc -g globals_long.c -o globals_long; gdb globals_long
```

c) Observa la disposición de los datos del array en memoria (`x/80bx array`), prestando especial atención a la nueva disposición de los elementos. ¿Cuánto ocupa ahora cada elemento del array? ¿Cuál es ahora la dirección de memoria del último elemento del array?

d) Utiliza el comando `diff -u globals.c globals_float.c` para ver las diferencias entre ambos programas. ¿En qué se diferencian?

e) Compila el programa `globals_float.c` y arranca con `gdb` el ejecutable que has generado.

```
$ gcc -g globals_float.c -o globals_float; gdb globals_float
```

f) Observa la disposición de los datos del array en memoria (`x/40bx array`), prestando especial atención a la nueva disposición de los elementos. ¿Cuánto ocupa ahora cada elemento del array? ¿Cuál es ahora la dirección de memoria del último elemento del array?

g) Ejecuta el programa paso a paso durante varias iteraciones, mostrando cómo cambian los valores en las direcciones de memoria donde se ubica `array`. ¿En qué ristra de bytes se codifica el valor -1 en dicha representación?

4. **Enlazado y bibliotecas.**

a) Genera un fichero ejecutable `hola.dynamic`, enlazado dinámicamente, y compilado con símbolos de depuración.

```
$ gcc -g hola.c -o hola.dynamic
```

b) Arranca con `gdb` el ejecutable dinámicamente enlazado que acabas de generar, pon un punto de ruptura en la función `puts` y ejecuta (`run`) hasta llegar a dicho *breakpoint*.

c) Muestra con `x/10i main` las 10 primeras instrucciones de la función `main`, fijándote en la dirección de su primera instrucción.

d) Muestra con `x/10i puts` las 10 primeras instrucciones de la función `puts`, fijándote en la dirección de su primera instrucción.

e) Con `CTRL-Z`, detén la ejecución de `gdb`, averigua con `ps` el PID del proceso que está ejecutando el programa `hola.dynamic` y finalmente muestra su mapa de memoria con el comando `pmap PID`.

f) ¿A qué región del espacio de memoria de dicho proceso corresponde la dirección donde está ubicado el código de `main`? ¿Qué tamaño tiene dicha región? ¿Qué fichero corresponde a dicho área?

g) ¿A qué región del espacio de memoria de dicho proceso corresponde la dirección donde está ubicado el código de `puts`? ¿Qué tamaño tiene dicha región? ¿Qué fichero corresponde a dicho área?

h) Devuelve `gdb` a ejecución con el comando `fg` (pulsando INTRO para ver de nuevo el *prompt* de GDB) y sal del depurador (con `quit`).

i) Arranca con `gdb` el ejecutable `hola.static`, estáticamente enlazado, que hemos usado anteriormente. De nuevo, pon un punto de ruptura en la función `puts` y ejecuta (`run`) hasta llegar a dicho *breakpoint*.

j) Muestra con `x/10i puts` las 10 primeras instrucciones de la función `puts`, fijándote en la dirección de su primera instrucción.

k) Con `CTRL-Z`, detén la ejecución de gdb, averigua con `ps` el PID del proceso que está ejecutando el programa `hola.static` y finalmente muestra su mapa de memoria con el comando `pmap PID`.

l) ¿A qué región del espacio de memoria de dicho proceso corresponde la dirección donde está ubicado el código de `main`? ¿Y el de `puts`? ¿Qué tamaño tiene dicha región? ¿Qué fichero corresponde a dicho área?

B8.2. Boletín 2: El lenguaje ensamblador del ISA Intel x86-64

B8.2.1. Objetivos

Esta sesión estará dedicada al estudio de los aspectos básicos del lenguaje ensamblador de nativo de los PCs, basado en el ISA Intel x86-64. Por un lado, se pretende hacer ver el papel que juega el compilador a la hora de generar código ensamblador, y cómo mediante los distintos niveles de optimización podemos conseguir código más eficiente. Por otro lado, se ilustrará el ISA x86-64 mediante la ejecución controlada de sencillos programas en ensamblador, con el objetivo de que el alumno observe de primera mano aspectos fundamentales acerca de la arquitectura del computador, tales como el uso de los registros del procesador, las forma en que se accede a los datos en memoria, etc.

B8.2.2. Plan de trabajo

El plan de trabajo de esta sesión será el siguiente:

1. Lectura y seguimiento de los pasos expuestos en el ejemplo del boletín.
2. Realización de forma individual de los ejercicios propuestos en el boletín. Estos consistirán en ligeras modificaciones sobre los pasos replicados anteriormente, y el estudio de sus efectos.

B8.2.3. Aspectos fundamentales del ISA x86-64.

La arquitectura x86-64 posee registros de tamaño 64 bits, cada uno de los cuales puede ser accedido en su totalidad, o bien por los 32, 16 u 8 bits inferiores. La Figura I.6 muestra los 8 registros de propósito general que se direccionan mediante su nombre (RAX, RBX, etc.), y que son extensión de los registros de 16 bit existentes en la arquitectura primitiva del Intel 8086 (año 1978). De esta forma, los procesadores de Intel que implementan el ISA x86 han mantenido la compatibilidad hacia atrás, de forma que cada evolución de este ISA es compatible con la anterior, lo cual permite que el software existente (compilado para una especificación del ISA más antigua) siga funcionando en procesadores que implementan evoluciones del ISA más recientes.

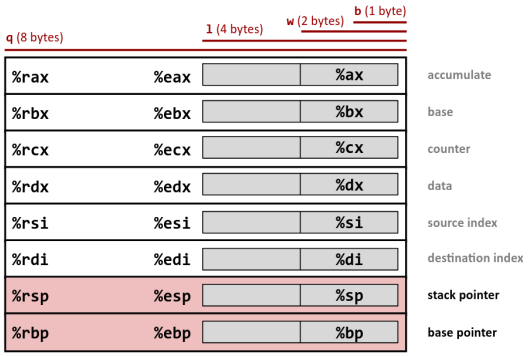

Figura I.6: Registros en el ISA x86-64.[4]

Así, por ejemplo, si nuestro programa en lenguaje C opera con una variable de tipo entero (cuya representación interna utiliza 32 bits), la traducción correspondiente a ensamblador x86-64 utilizará instrucciones que operen únicamente con la mitad inferior de un registro de 64 bits, donde se almacena dicho dato. Por ejemplo, dicho entero podría guardarse en EAX, que son los 32 bits inferiores del registro RAX, como vemos a continuación.

```
|----------------------------rax-------------------------------|
|                             |--------------eax-------------|
|                             |               |------ax------|
|                             |               |       |--al--|
0000000000000000000000000000000000000000000000000000000000000000
```

[4]Fuente: `https://hackmd.io/@paolieri/x86_64`

Repertorio básico de instrucciones en x86-64.

Recuerda que en la sintaxis que utiliza por defecto el compilador `gcc`, el formato general de una instrucción ensamblador x86-64 es: `mnemonic source, destination`. Por lo general, el operando a la derecha de la coma es el que se modifica como resultado de la ejecución de la instrucción.

- **Instrucciones aritmético-lógicas**: Sirven para hacer operaciones aritméticas (suma, resta, multiplicación, etc.) y/o lógicas (and, or, xor, desplazamiento de bits, etc.) con los operandos. Ejemplos:
  ```
  add %rbx, %rax   # Suma RBX a RAX, y deja el resultado en RAX
  ```
  También pueden operar con constantes (siempre precedidas por $):
  ```
  sub $1234, %rax # Resta 1234 a RAX, y deja el resultado en RAX
  ```

- **Instrucciones de movimiento de datos**: Sirven para copiar datos de un registro a otro, de la memoria a un registro (y viceversa) y para cargar valores constantes en registros.
  ```
  mov %rbx, %rax    # Copia el valor de RBX a RAX
  mov $1234, %rax   # Establece RAX con el valor 1234
  ```
  En particular, para leer y escribir datos en memoria es muy habitual establecer un registro con la dirección de memoria a la que queremos acceder, y posteriormente utilizar dicho registro como "apuntador" para leer/escribir de dicha dirección. Por ejemplo, si previamente hemos establecido RBX con la dirección de memoria de un entero largo (64 bits), la siguiente instrucción lo leería de memoria y lo copiaría al registro RAX:
  ```
  mov (%rbx), %rax  # Usa RBX para leer un dato de memoria y copiarlo a RAX
  ```
  A veces, queremos acceder a posiciones de memoria que están "cerca" de la dirección que tenemos guardada en el registro "puntador". Por ejemplo, la siguiente instrucción escribe el valor de RAX en memoria, en la dirección resultante de restar 4 al valor del registro RBP:
  ```
  mov %rax, -4(%rbp) # Usa RBP para escribir en memoria el valor de RAX
  ```

- **Instrucciones de salto incondicional**: Rompen el flujo secuencial de ejecución del programa (una instrucción tras otra), ya que establecen el registro contador de programa (RIP en x86-64) a una dirección de código fija, indicada por una etiqueta (hacia atrás o adelante en el código). El programa sigue ejecutándose a partir de la instrucción destino del salto:
  ```
        jmp .L1
  [...]
  L1:  mov %rax, %rbx
  ```

- **Instrucciones de salto condicional**: Sólo saltan a la etiqueta si se cumple una determinada condición. Se utilizan en la traducción de bucles (for, while, ...) y condiciones (if, switch, ...) de los lenguajes de alto nivel como C. La condición se comprueba en una instrucción `cmp` anterior.
  ```
        cmp $5, %rax    # Compara RAX con 5
        jge .L1         # Salta a .L1 si RAX mayor o igual que 5
        [...]
  .L1:  mov %rax, %rbx
  ```

- **Instrucciones de soporte de procedimientos**: Las instrucciones `call` y `ret` sirven, respectivamente, para llamar y regresar de procedimientos. Al llamar a un procedimiento, se guarda la dirección de retorno (dirección de la siguiente instrucción tras el `call`), que es recuperada por la instrucción `ret`. Por ejemplo, para llamar a la función `puts` desde `main` y regresar posteriormente al main (en la instrucción `cmp`), se usarían estas instrucciones:
  ```
  puts: [...]
        ret             # Regresa al invocador (último call)
  main: [...]
        call puts       # Salta a la etiqueta 'puts'
        [...]           # Regresa aquí mediante 'ret'
  ```

B8.2.4. Optimizaciones del compilador.

Utilizaremos de nuevo la aplicación web *Compiler Explorer*[5] para mostrar el lenguaje ensamblador generado por el compilador `gcc`. Partiremos del fichero de código fuente en C `aritmetica.c` disponible como recurso en el Aula Virtual. Una vez en el navegador, seleccionamos el lenguaje C en el panel izquierdo de código fuente (*source*) y a continuación copiamos y pegamos el código de `aritmetica.c` en dicho panel. En el panel derecho podemos elegir el compilador a utilizar, si bien por ahora utilizaremos el compilador por defecto (`gcc` compilando para el ISA x86-64). Al igual que en el boletín anterior, en el botón de *Output* del panel del compilador elegimos no usar la sintaxis de Intel. El resultado se muestra en la Figura I.7. En el botón *Add new* podemos añadir un nuevo panel del tipo *Clone compiler* para observar el resultado de compilar el programa con el mismo compilador, pero usando dos niveles distintos de optimización del código: para ello, pasamos las opciones `-O0` (sin optimizar) y `-O1`, respectivamente.

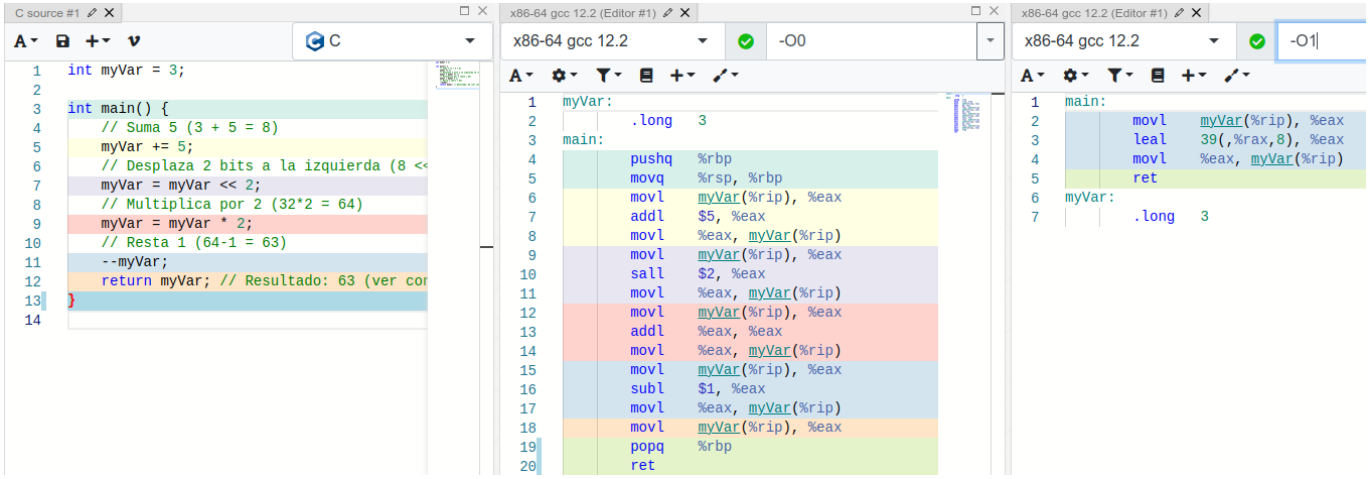

Figura I.7: Comparativa del código ensamblador de x86-64 generado mediante GCC para un mismo programa, sin y con optimización por parte del compilador (*Compiler Explorer*).

Vemos que el código ensamblador generado varía ostensiblemente en función del nivel de optimización que use el compilador. 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 y con ellas (17 frente a 4 instrucciones), en gran parte porque el código sin optimizar lee de memoria la variable `myVar` en cuatro ocasiones y la escribe otras tantas (una por cada sentencia del código C: suma, desplazamiento, multiplicación y resta). Como se puede apreciar fácilmente gracias al empleo de diferentes colores en la Figura I.7, en el código sin optimizar cada sentencia C que opera sobre la variable se traduce en: 1) cargar el valor de la variable en memoria al registro EAX (32 bits menos significativos del registro EAX); 2) operación aritmético-lógica que lee el valor del registro EAX y escribe el resultado de nuevo en EAX; y 3) guardar el valor del registro EAX en la memoria asignada a `myVar`.

Por su parte, al activar las optimizaciones no sólo se reduce a una lectura y una escritura de la variable en cuestión, sino que además el compilador traduce las cuatro sentencias en C con operaciones artiméticas a una única instrucción (línea 3 en la Figura I.7). Esta instrucción computa directamente el resultado que debe devolver el programa (myVar*8+39), ya que compilador es capaz determinar que el resultado del programa siempre será el mismo (63), pues ningún dato de entrada varía.

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, en adelante en este boletín vamos a hacer uso de traducciones *manuales* a ensamblador, en lugar de apoyarnos el código generado automáticamente por el compilador.

[5] `https://godbolt.org`

B8.2.5. Ejecución de programas en ensamblador x86-64 con GDB

Con el fin de mejorar la legibilidad e ilustrar más fácilmente el lenguaje ensamblador de x86-64, el código fuente del fichero `aritmetica.c` ha sido traducido de forma manual al código ensamblador que vemos en fichero `aritmetica_manual.s`, cuyo contenido es el siguiente (disponible como recurso en el Aula Virtual):

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

En primer lugar, se observa la declaración de una primera parte del programa dedicada al segmento de datos en la que vemos la variable global `myVar` (referida simbólicamente por la etiqueta `myVar:`), de tamaño 4 bytes (directiva `.long`) e inicializada con el valor 3. A continuación vienen las instrucciones del programa (segmento de código). Distinguimos en primer lugar la función principal `main`, que comienza en la etiqueta `main:` y acaba con la 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, etc.) y de movimiento de datos.

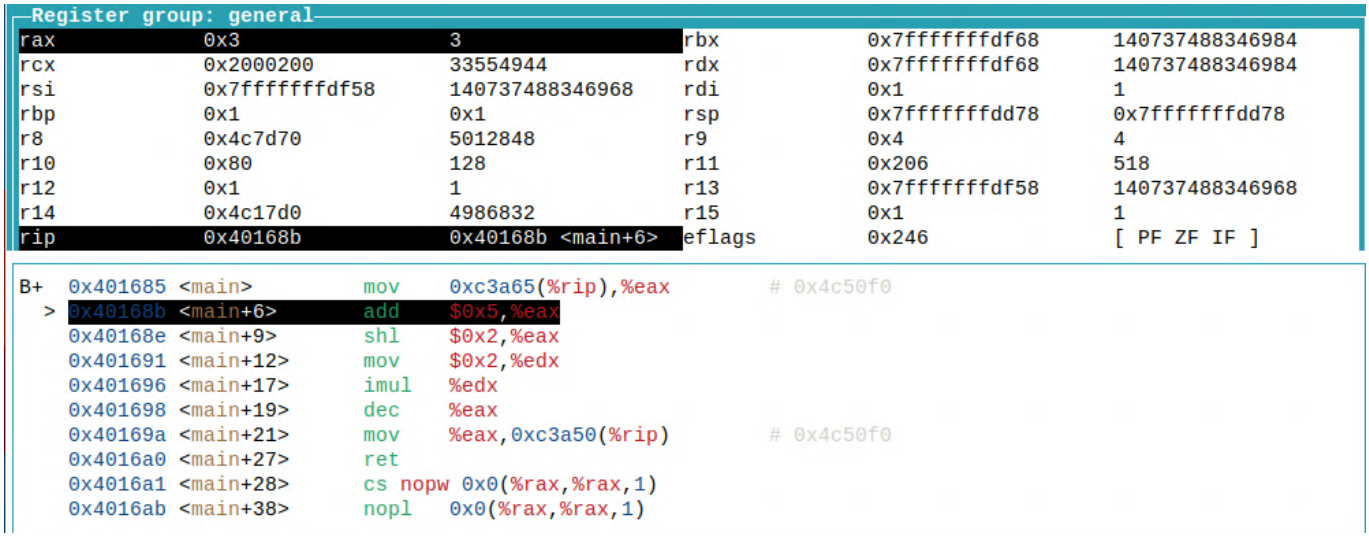

Figura I.8: Vista de los registros de la CPU durante la ejecución controlada de `aritmetica_manual` con GDB.

Por último, podemos ensamblar y enlazar el código del fichero `aritmetica_manual.s` con el comando `gcc -g -static aritmetica_manual.s -o aritmetica_manual`. Una vez generado el ejecutable estático, procedemos a tracearlo con GDB. Activaremos de nuevo la interfaz textual (TUI) con `layout split`, y a continuación escribiremos el comando `layout regs`, para mostrar los valores de los registros del procesador (panel superior) y el código ensamblador del programa (panel inferior). Establecemos un punto de ruptura en `main`. Después, ejecutamos el programa (`run`) hasta que se detiene en la primera instrucción de `main`. Ahora, podemos ejecutar de instrucción en instrucción con el comando `stepi`. En la Figura I.8 vemos que tras ejecutar la primera instrucción, el registro RAX toma el valor leído de memoria que en ese momento tiene la variable `myVar` (inicialmente, 3). Los registros modificados por la última instrucción ejecutada aparecen resaltados en esta perspectiva: además del contador de programa (RIP), que pasa a apuntar a la siguiente instrucción, también se ha modificado RAX.

B8.2.6. Traducción de bucles y acceso a arrays en ensamblador.

Utilizaremos de nuevo la aplicación web *Compiler Explorer* para observar cómo el compilador lleva a cabo la traducción a lenguaje ensamblador de x86-64 de cada una de las sentencias del programa en C `array.c` (disponible como recurso en el Aula Virtual), cuyo código vemos en la Figura I.9.

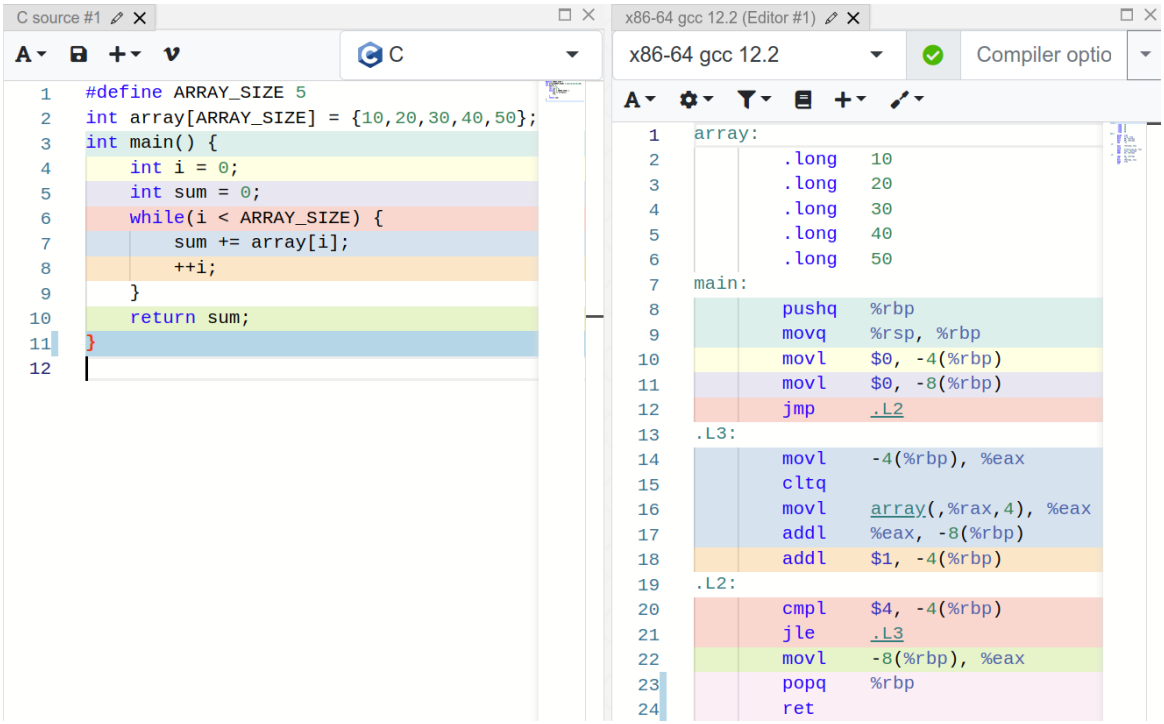

Figura I.9: Traducción a ensamblador de x86-64 del programa `array.c` mediante GCC (*Compiler Explorer*).

Se trata de un sencillo programa que recorre un array de enteros sumando sus valores. Vemos como la variable global `array` se traduce en las líneas 1 al 6, empezando por una etiqueta homónima que representa la dirección en memoria en la que se ubicará dicha variable (ya que, al ser global, es accesible en cualquier punto del programa). A continuación de la etiqueta, se observa la declaración de cada uno de los 5 elementos que conforman el array, inicializados con los valores `{10,20,30,40,50}`. Cada directiva `.long` indica al ensamblador que debe reservar 4 bytes (tamaño de un entero) con un determinado valor.

Al contario de lo que ocurre con las variables globales como `array`, las variables locales `i` y `sum` no tienen un espacio en memoria permanentemente asignado ya que una variable local sólo es accesible cuando la función en la que se declara está en ejecución. Las variables locales se almacenan en una zona de la memoria denominada *pila*, que se utiliza para el soporte de procedimientos (llamadas a funciones, variables locales, paso de parámetros, etc.). Durante la ejecución de una función, el valor del registro RBP suele fijarse a una dirección fija de la pila, lo cual permite acceder a las variables que están en la pila usando dicho registro como *apuntador*. Así, por ejemplo, las instrucciones en las líneas 10 y 11 de la Figura I.9 se encargan, respectivamente, de escribir el valor 0 en las variables locales `i` y `sum`, y cuyas direcciones en memoria se obtienen restando 4 y 8 al valor del registro RBP.

En la línea 12 tenemos un salto incondicional a la etiqueta .L2, lugar en el que se encuentran las instrucciones que comprueban la condición del bucle *while*: se compara el valor de la variable `i` con 4, y se salta a la etiqueta .L3 (inicio del cuerpo del bucle) si en la comparación anterior `i` resultó ser menor o igual que 4 (`jle`, *jump if less or equal than*).

El cuerpo del bucle se traduce en dos partes: lectura del elemento i-ésimo del array (líneas 14-17) para sumarlo a la variable local `sum`, e incremento de la variable local `i`. La instrucción en la línea 16 realiza el acceso al elemento i-ésimo del array usando el registro EAX, en el cual se ha copiado previamente el valor de `i` leído de la pila (línea 14). Así, la instrucción `mov` de la línea 16 lee de memoria en la dirección `array+rax*4` (cada entero son 4 bytes) y copia el valor a EAX, mientras que la línea 17 suma el valor de EAX a la variable `sum` ubicada en la pila.

B8.2.7. Traducción manual a ensamblador.

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 (se han añadido números de línea por claridad):

```
 1          #### Segmento de datos (variables globales del programa)
 2          .data
 3 array:   .long   10
 4          .long   20
 5          .long   30
 6          .long   40
 7          .long   50

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

11 main:    # Procedimiento principal, llamado por el cargador del SO (loader)
12          # Variables locales a main:
13          # i -> registro ESI
14          # sum -> registro EAX
15          mov    $0, %esi        # Inicializa i a 0
16          mov    $0, %eax        # Inicializa sum a 0
17          lea    array(%rip), %rbx  # RBX: dirección de memoria de array

18          ##### Bucle while:
19 inicio_while:
20          # Comprobación de la condición
21          cmp    $5, %esi        # Compara ESI con 5 (tamaño del array)
22          jge     fin_while      # Salta a fin_while si cmp anterior fue mayor o igual
23          mov    (%rbx), %ecx    # Lee variable en la dirección de memoria
24                                 # array+4*i y lo guarda en el registro ECX
25          add    %ecx, %eax      # sum += array[i]
26          # Incremento de la variable de control
27          inc     %esi            # i++
28          add    $4, %rbx         # EDX: dirección del siguiente elemento del array
29          # Regresa al inicio del bucle
30          jmp     inicio_while
31 fin_while:
32          # EAX contiene sum (valor retornado por main)
33          ret                     # Termina el procedimiento main
```

Esta traducción manual a ensamblador es diferente al código que vimos en la Figura I.9, por dos razones. Primero, utiliza dos registros para mantener las variables locales `sum` (EAX) e `i` (ESI), en vez de la pila. En segundo lugar, mantiene en cada iteración del bucle la dirección del elemento i-ésimo del array en el registro RBX. En el segmento de código, tras la inicialización de los registros donde se guarda el valor de `sum` e `i`, tenemos la instrucción que coloca en RBX la dirección de memoria donde comienza el array. Después, tenemos el bucle while, entre las etiquetas `inicio_while` y `fin_while`), que se divide en:

1. Comprobación de la condición de continuación del bucle, mediante una instrucción de comparación (línea 21) seguida de un salto condicional `jge` a la etiqueta `fin_while`. El salto se producirá sólo cuando el registro que alberga la variable `i` (ESI) sea mayor o igual que 5 (o sea, cuando se deba salir del bucle)

2. Cuerpo del bucle: Lectura del elemento i-ésimo del vector `array` situado en memoria (en el segmento de datos), mediante la instrucción `mov (%rbx), %ecx`, que lee el entero (4 bytes) en la dirección de memoria dada por el valor actual del registro RBX, y lo guarda en el registro ECX. El elemento del array leído de memoria se suma a la variable local `sum` (línea 25)

3. Incremento de la variable `i` que dirige el bucle `while`, mediante la instrucción `inc %esi`.

4. Sumar 4 a la dirección de memoria contenida en RBX, para apuntar al siguiente elemento del array en memoria (puesto que cada elemento ocupa 4 bytes).

5. Vuelta al comienzo del bucle mediante la instrucción de salto incondicional `jmp inicio_while`, para comprobar nuevamente la condición con el nuevo valor de la variable i.

B8.2.8. Ejercicios a realizar durante la sesión

NOTA IMPORTANTE:
Para responder a los ejercicios de este boletín, simplemente edita con el editor de texto de tu elección (`gedit`, `nano`, etc.), un fichero de texto llamado `ejercicios_prac8_bol2`, y escribe ahí tus respuestas. Puedes copiar y pegar cualquier información mostrada por GDB, que consideres útil para completar o explicar tu respuesta. Debes empezar cada ejercicio escribiendo el comentario de documentación `### EJERCICIO N ###` , y en su caso especificar el apartado dentro del ejercicio correspondiente a cada pregunta. Finalmente, no olvides añadir dicho fichero a tu repositorio-bitácora, respetando la organización de la misma en directorios y subdirectorios.
EN EL MODO TUI DE GDB, ES POSIBLE QUE TENGAS QUE PULSAR LA TECLA MAYÚSCULA PARA PODER SELECCIONAR, COPIAR Y PEGAR CON EL RATÓN

1. **Instrucciones aritmético lógicas. Registros del procesador.** Reproduce fielmente los pasos indicados en el boletín para compilar y ejecutar de manera controlada el programa `aritmetica_manual.s`. Sigue estos pasos:

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

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

   b) Carga el ejecutable obtenido con `gdb aritmetica_manual`

   c) Usa el comando `layout split` para activar el modo TUI.

   d) Usa el comando `layout regs` para mostrar los registros del procesador.

   e) Usa el comando `print (int)myVar` para mostrar el valor inicial de la variable.

   f) Usa el comando `print (int*)&myVar` para mostrar la dirección de memoria donde se ubica la variable. ¿Entre qué dos direcciones de memoria se almacena `myVar`? Da la respuesta como dirección inicial-final (ambas direcciones incluidas)

   g) Usa el comando `x/4bx DIRECCION` para examinar el contenido de la memoria en la dirección ocupada por la variable (obtenida del apartado anterior). ¿De qué valor se trata?

   h) Pon un breakpoint al comienzo (`b main`) y lanza el programa (`run`). ¿Cuál es la dirección de memoria donde está la primera instrucción del programa?

   i) 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, hasta ejecutar la instrucción `imul %edx`.

   j) ¿Qué valores tiene el registro RAX antes y después ejecutar la instrucción `shl`?

   k) ¿Qué registros se modifican al ejecutar la instrucción `imul`?

   l) ¿Qué valores tiene el registro RAX antes y después ejecutar la instrucción `dec`?

   m) Ejecuta paso a paso hasta llegar al `ret`. En ese punto, muestra de nuevo el contenido de la memoria ocupada por `myVar`.

   n) Vuelve a mostrar el contenido de la memoria ocupada por la variable (`x/4bx DIRECCION`) justo antes de ejecutar la instrucción `ret`.

   ñ) Finalmente, continúa la ejecución hasta terminar el programa.

2. **Instrucciones de movimiento de datos e instrucciones de salto. Acceso a memoria.** Reproduce fielmente los pasos indicados en el boletín para compilar y ejecutar de manera controlada el programa `array_manual.s`. Sigue estos pasos:

a) Ensambla el programa `array_manual.s` para generar un binario ejecutable con información de depuración:

```
gcc -static -g array_manual.s -o array_manual
```

b) Carga el ejecutable obtenido con `gdb array_manual`

c) Usa el comando `layout split` para activar el modo TUI.

d) Usa el comando `layout regs` para mostrar los registros del procesador.

e) Pon un breakpoint al comienzo (`b main`) y lanza el programa (`run`). ¿Cuál es la dirección de memoria donde está la primera instrucción del programa?

f) Ejecuta paso a paso con `stepi` hasta llegar a la instrucción `cmp`. ¿Cuál es el valor del registro RBX?

g) Muestra con `x/20bx DIRECCION` el contenido de la memoria a partir de la dirección indicada por el registro RBX. ¿Qué variable se almacena en esa zona de memoria? ¿Entre qué dos direcciones de memoria se almacena `array`? Da la respuesta como dirección inicial-final (ambas direcciones incluidas)

h) Visualiza la dirección de la variable `array` con `print (int[4]*)&array`. Comprueba que se trata del mismo valor que contiene RBX antes de entrar en el bucle.

i) Usa el comando `stepi` para ejecutar por primera vez la instrucción `cmp`. ¿Qué registro se ha modificado? ¿Qué tamaño (en bytes) ocupa en memoria dicha instrucción?

j) Usa el comando `stepi` para ejecutar por primera vez la instrucción `jge`. ¿Se ha cumplido la condición del salto?

k) Usa el comando `stepi` para ejecutar por primera vez la instrucción `mov (%rbx),%ecx`. ¿Qué valor tiene el registro ECX tras su ejecución? ¿De dónde procede dicho valor? Muéstralo con `x/4bx DIRECCION`, sustituyendo *DIRECCION* por el valor del registro RBX en ese instante.

l) ¿Qué valor tiene el registro EAX en este momento? Usa el comando `stepi` para ejecutar por primera vez la instrucción `add %ecx,%eax`. ¿Qué valor tiene EAX tras la suma? ¿Qué variable del programa `array.c` está siendo almacenada en EAX?

m) ¿Cuál es el valor del registro RIP en este punto? ¿Cuánto valdrá RIP justo después de ejecutar la instrucción `inc %esi`?

n) Usa el comando `stepi` para ejecutar hasta llegar a la instrucción `jmp`. ¿Cuál es el valor del registro RIP en este punto? ¿Cuánto valdrá RIP justo después de ejecutar la instrucción `jmp`?

ñ) Ejecuta paso a paso la segunda iteración del bucle (NOTA: En GDB Puedes simplemente pulsar INTRO para repetir la ejecución del último comando tecleado). ¿A qué dirección de memoria accede esta vez la instrucción `mov %rbx,%ecx`? ¿Qué valor se lee esta vez?

o) Continúa ejecutando paso a paso hasta que la instrucción `inc %esi` haga que dicho registro valga 5, fijándote en el valor del registro EFLAGS tras cada ejecución de la instrucción `cmp`. ¿Se activa el flag de *zero* (ZF)?

p) Cuando ejecutes la instrucción `cmp` una vez RSI valga 5, ¿qué flag se ha activado que no aparecía antes? ¿Qué operación crees que realiza el procesador al ejecutar la instrucción de comparación para establecer el registro EFLAGS?

3. **Instrucciones de soporte de procedimientos.** Reproduce fielmente los pasos indicados en el boletín para compilar y ejecutar de manera controlada el programa `funcion_manual.s`. Sigue estos pasos:

a) Observa el código del programa `funcion.c` y explica qué hace este programa. ¿Cuántas funciones contiene?

b) Ensambla el programa `funcion_manual.s`, el cual contiene una traducción manual a lenguajge ensamblador del programa C `funcion.c`, para generar un binario ejecutable con información de depuración:

```
gcc -static -g funcion_manual.s -o funcion_manual
```

c) Carga el ejecutable obtenido con `gdb funcion_manual`

d) Usa el comando `layout split` para activar el modo TUI.

e) Usa el comando `layout regs` para mostrar los registros del procesador.

f) Pon un breakpoint al comienzo (`b main`) y lanza el programa (`run`).

g) Usa el comando `stepi` para ejecutar paso a paso hasta llegar a la instrucción `call`. ¿Cuál es valor del registro RSP en este punto?

h) Usa el comando `x/1qx DIRECCION-8`, sustituyendo *DIRECCION* por el valor del registro RSP en ese instante. Fíjate en el valor que hay almacenado en memoria.

i) Usa el comando `stepi` para ejecutar la instrucción `call`. ¿Qué registros se han modificado? ¿Cuál es valor del registro RSP ahora? Fíjate que ahora apunta a la dirección de memoria cuyo contenido hemos mostrado en el apartado anterior.

j) Repite el comando anterior `x/1qx DIRECCION` para mostrar el contenido de la dirección de memoria apuntada por el registro RSP, sustituyendo de nuevo *DIRECCION* por el valor del registro RSP. ¿Qué valor contiene?

k) Sabiendo que el valor mostrado en el apartado anterior (*VALOR*) es a su vez una dirección de memoria, muestra el contenido de la memoria en dicha dirección con `x/2i VALOR`. ¿Qué contiene? A la vista de lo anterior, indica para qué crees que la instrucción `call` ha guardado dicho valor en la zona de memoria apuntada por el registro RSP (pila).

l) Usa el comando `stepi` para ejecutar paso a paso el código de la función `funcion_resta` hasta llegar a la instrucción `ret`. ¿Cuál es valor del registro RSP en este punto? ¿Y el valor del registro RIP?

m) Usa el comando `stepi` para ejecutar la instrucción `ret`. ¿Cuál es valor del registro RSP ahora? ¿Cuál es el valor del contador de programa (registro RIP)? Explica de dónde crees que proviene el valor que ha tomado ahora el registro RIP.

n) Termina la ejecución del programa y de GDB.

4. **Un error de programación archiconocido.**

a) Modifica el código ensamblador de `funcion_manual.s` de forma que la instrucción `call` invoque la ejecución de la función `main` en vez de invocar `funcion_resta`. Únicamente has de cambiar la etiqueta que sirve de operando a "call", de esta forma:

```
-       call    funcion_resta   # Llama a funcion_resta
+       call    main            # Llama a main
```

b) Compila de nuevo el programa, repite los apartados iniciales y ejecuta con `stepi` hasta llegar a `call`. ¿Qué ocurre tras ejecutar esta instrucción?

c) Continúa la ejecución del programa con el comando `continue`, fijándote en el valor del registro RSP. Repite este comando en 10 ocasiones.

d) Usa el comando `ignore 1 100` para ignorar las próximas 100 ocurrencias del punto de ruptura número 1 (en la primera instrucción de `main`).

e) Continúa la ejecución del programa con el comando `continue`.

f) Muestra con el comando `x/100qx DIRECCION` el contenido de la dirección de memoria apuntada por el registro RSP. ¿Qué valores contiene?

g) Elimina el punto de ruptura 1 con el comando `delete 1` y continúa la ejecución del programa. ¿Qué le ocurre al programa?[6]

[6] Acabas de ver en primera persona una situación anómala ampliamente conocida por cualquier programador, que da nombre al sitio web de preguntas y respuestas para programadores profesionales y aficionados más popular de Internet.