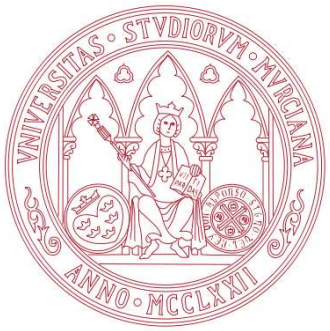

Universidad de Murcia

Facultad de Informática

TÍTULO DE GRADO EN
INGENIERÍA INFORMÁTICA

Fundamentos de Computadores

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

Boletín de autoevaluación de las sesiones prácticas

CURSO 2019 / 20

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

Índice general

Cuestiones de autoevaluación

A5.1. Objetivos

El objetivo de este boletín es comprobar el grado de aprovechamiento por parte del alumno de las sesiones prácticas del tema 5 de la asignatura. Todas las preguntas planteadas en este boletín deberían poder resolverse con los conceptos aprendidos y la documentación manejada durante dicha sesión.

A lo largo de este boletín de autoevaluación te encontrarás con fundamentalmente cuatro tipos de cuestiones diferentes:

1. Preguntas de tipo *test*: se ofrecen varias opciones, de las cuales tendrás que elegir cuál es la única correcta.
2. Preguntas de *emparejamiento*: se dan dos columnas con el mismo número de opciones en cada una, y hay que emparejar todos y cada uno de los términos de la columna primera con sus respectivos términos correspondientes en la segunda.
3. Preguntas de *rellenar huecos*: se enuncia un texto en el que existen algunos huecos, que tendrás que rellenar con el término correcto.
4. Preguntas de *razonamiento* breve: se realiza una pregunta a la que hay que contestar de modo preciso y conciso (menos de un párrafo).

Se trata de que, individualmente, en horario de trabajo en casa (fuera de las sesiones de teoría/prácticas semanales), y tras la finalización de las sesiones de prácticas dedicadas al tema 5, cada alumno intente resolver por su cuenta las preguntas del boletín, por supuesto SIN mirar previamente las soluciones, que se encuentran en el último apartado del documento. Una vez realizados los ejercicios, el propio alumno comprobará la corrección de los resultados, mirando las soluciones disponibles en dicho último apartado. Este proceso debería ayudarle a detectar sus propias carencias, en las que debe insistir en el trabajo de estudio en casa, repetición de los ejercicios propuestos en el boletín, planteamiento de dudas al profesor, etc.

Nota: Una cierta cantidad de las las preguntas planteadas en estos boletines han aparecido, planteadas de una u otra forma, en exámenes pasados de esta asignatura.

A5.2. Cuestiones

Observar el código C mostrado en la figura I.1, y contestar a las preguntas relacionadas:

```
#include<stdio.h>
int array[16] = {-4,0,-3,0,-2,0,-1,0,1,0,2,0,3,0,4,0};
int main() {
    int i = 0, j = 0;
    for(i=0;i<16;i=i+2) {
        j = i+1;
        array[j] = funcion(i);
    }
    for(i=0;i<16;i++)
        printf("%d ",array[i]);
    printf("\n");
}
int funcion(int parametro) {
    return 10*array[parametro];
}
```

Figura I.1: Código C de ejemplo.

1. **[*Razonar*]** Expresar en castellano, de modo conciso, lo que hace el código en C de dicha figura.
2. **[*Test*]** ¿Cuántos bytes ocupará en memoria el array completo `array` (sus 16 elementos), sabiendo que cada entero ocupa exactamente 32 bits?

   a) 16
   b) 32
   c) 64
   d) 128

3. **[*Emparejar*]** Si llamamos `array:` a la dirección donde comenzará el array una vez colocado en memoria, empareja las siguientes direcciones de memoria con las correspondientes posiciones de elementos del array:

`array: + 48`	`array[0]`
`array: + 4`	`array[1]`
`array: + 32`	`array[4]`
`array:`	`array[8]`
`array: + 16`	`array[12]`

A partir del programa de la figura I.1, guardándolo en un fichero con nombre `main.c` y compilándolo con el comando `gcc-4.8 main.c -fno-asynchronous-unwind-tables -S -o main.s`, se puede generar el código ensamblador correspondiente al mismo en el fichero `main.s`. Parte de dicho código se muestra en la figura I.2. Observar atentamente dicho código, y contestar entonces a las preguntas relacionadas:

```
[...]
        .data
[...]
array:
        .long   -4
        .long   0
        .long   -3
[...]
.LC0:
        .string "%d "
        .text
[...]
main:
[...]
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        movl    $0, -8(%rbp)
        jmp     .L2
.L3:
        movl    -8(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -4(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, %edi
        movl    $0, %eax
        call    funcion
        movl    -4(%rbp), %edx
        movslq  %edx, %rdx
        movl    %eax, array(,%rdx,4)
        addl    $2, -8(%rbp)
.L2:
        cmpl    $15, -8(%rbp)
        jle     .L3
[...]
        ret
[...]
funcion:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        cltq
        movl    array(,%rax,4), %edx
        movl    %edx, %eax
        sall    $2, %eax
        addl    %edx, %eax
        addl    %eax, %eax
        popq    %rbp
        ret
[...]
```

Figura I.2: Parte del código ensamblador generado al compilar el código de la de la figura I.1.

4. **[*Rellenar*]** La parte correspondiente a los datos del fichero ensamblador generado comienza con la directiva _____, y la parte correspondiente al código en sí (instrucciones) comienza con la directiva _____.

5. **[*Rellenar*]** En el fichero ensamblador generado se hacen referencias a direcciones de código y datos mediante *etiquetas*, que son secuencias de caracteres terminadas con el carácter _____. Dichas etiquetas pueden venir de nombres de variables y/o funciones tomadas directamente del código C, o bien haber sido generadas automáticamente por el ensamblador, en cuyo caso siempre comienzan con el carácter _____. Finalmente, todas las constantes numéricas inmediatas a las que hacen referencia ciertas instrucciones se escriben con el número correspondiente, precedido por el carácter _____.

6. **[*Test*]** Las directivas `.long` que hay a continuación de la etiqueta `array:` significan:

   a) Que cada dato entero (`int`) correspondiente va a ocupar exactamente 8 bits en memoria.
   b) Que cada dato entero (`int`) correspondiente va a ocupar exactamente 16 bits en memoria.
   c) Que cada dato entero (`int`) correspondiente va a ocupar exactamente 32 bits en memoria.
   d) Que cada dato entero (`int`) correspondiente va a ocupar exactamente 64 bits en memoria.

7. **[*Test*]** Las instrucciones `movl $0, -4(%rbp)` y `movl $0, -8(%rbp)` sirven para:

   a) Almacenar la constante cero en sendas variables locales ubicadas en la zona de memoria de la pila.
   b) Almacenar la constante cero en sendos registros del procesador.
   c) Copiar un dato de una posición de memoria a un registro del procesador.
   d) Copiar un dato de un registro a otro del procesador.

8. **[*Emparejar*]** Emparejar las siguientes instrucciones con sus respectivas funcionalidades:

`jmp .L2`	Añadir el contenido de un registro a la cima de la pila
`movl -8(%rbp),%eax`	Movimiento de memoria a registro
`call funcion`	Salto condicional
`movl %eax,array(,%rdx,4)`	Suma de una constante a una variable en memoria
`addl $2,-8(%rbp)`	Llamada a subrutina
`ret`	Retorno de procedimiento
`cmpl $15,-8(%rbp)`	Comparación de constante y variable en memoria
`jle .L3`	Movimiento de registro a dirección de memoria calculada
`movslq %edx,%rdx`	Movimiento entre registros de distintos tamaños
`pushq %rbp`	Retirar el contenido de un registro desde la cima de la pila
`popq %rbp`	Multiplicar contenido de un registro por 4 usando desplazamiento de bits
`sall $2,%eax`	Salto incondicional

9. **[*Razonar*]** Expresar en castellano, de modo conciso, lo que hacen exactamente las instrucciones siguientes en el código ensamblador generado:

```
cmpl $15,-8(%rbp)
jle  .L3
```

10. **[*Razonar*]** ¿A qué acción en el programa en C original crees que se corresponderán las instrucciones de la anterior cuestión?

11. **[*Test*]** Justo antes de realizar la llamada a la función `funcion`, hay que pasarle de alguna manera el parámetro `i`. Esto se hace moviendo dicho valor a un determinado registro, desde donde será luego leído el valor en la subrutina propiamente dicha. Dicho registro es, en este caso:

    a) `%eax`

b) `%edi`

c) `%rbp`

d) `%ebx`

12. **[*Test*]** Justo después de realizarse la llamada a la función `funcion`, ésta devuelve en `%eax` el valor a almacenar en la posición `i` de `array`. La instrucción en ensamblador que precisamente almacena este valor en dicho lugar (`array[i]`) es:

    a) `movl -4(%rbp),%edx`

    b) `movslq %edx,%rdx`

    c) `movl %eax,array(,%rdx,4)`

    d) `addl $2,-8(%rbp)`

    e) `cmpl $15,-8(%rbp)`

13. **[*Rellenar*]** La instrucción ensamblador que incrementa en dos unidades el índice del bucle es exactamente _____.

14. **[*Rellenar*]** Las variables locales `i` y `j` se encuentran en las posiciones de pila contiguas apuntadas por _____ y _____, respectivamente. La primera *instrucción de almacenamiento* en la variable `j` es, concretamente _____, mientras que la primera *instrucción de carga* sobre dicha variable es _____.

15. **[*Emparejar*]** Emparejar los siguientes registros de un procesador Intel x86-64 con sus correspondientes cometidos principales:

`%rip`	Cima de la pila
`%rax`	Registro de uso general
`%rbx`	Registro base para apuntar a la pila
`%rcx`	Registro de uso general
`%rdx`	Contador de programa
`%rsp`	Registro de uso general
`%rbp`	Registro de uso general

16. **[*Test*]** ¿Dónde se queda almacenada la dirección de retorno a la que hay que volver tras una instrucción del tipo `call` justo en el momento después de ejecutarse dicha instrucción?

    a) En la cima de la pila (apuntada por `%rsp`).

    b) En la cima de la pila (apuntada por `%rbp`).

    c) Depende de como se programe; en el ejemplo se guarda en el registro `%eax`.

    d) No es necesario guardar dicha dirección en ningún sitio.

17. **[*Rellenar*]** La instrucción con la que acaban las subrutinas en ensamblador del IA-32 es la instrucción _____.

18. **[*Razonar*]** Explicar en detalle qué hace exactamente dicha instrucción.

19. **[*Test*]** Exactamente, ¿qué instrucción ensamblador de la subrutina `funcion` accede por primera vez al valor del parámetro `i`?

    a) `pushq %rbp`

    b) `movq %rsp,%rbp`

c) `movl %edi,-4(%rbp)`

d) `movl -4(%rbp),%eax`

e) `cltq`

f) `movl array(,%rax,4),%edx`

20. **[*Rellenar*]** La instrucción que mete en la pila el valor de un determinado registro es _____, y la que lo recupera posteriormente es la instrucción _____. En ambos casos se decrementa/incrementa el registro que apunta a la cima de la pila, esto es, el registro _____.

En la figura I.3 se muestra parte del código máquina generado al compilar el programa C de la figura I.1 con el comando `gcc-4.8 -c main.c -o main.o`, posteriormente desensamblado con el correspondiente comando `objdump -d main.o > main.disassembled`. Observar atentamente dicho código y contestar a las preguntas relacionadas:

```
0000000000000000 <main>:
[...]
  1d:   eb 29                   jmp    48 <main+0x48>
  1f:   8b 45 f8                mov    -0x8(%rbp),%eax
  22:   83 c0 01                add    $0x1,%eax
  25:   89 45 fc                mov    %eax,-0x4(%rbp)
  28:   8b 45 f8                mov    -0x8(%rbp),%eax
  2b:   89 c7                   mov    %eax,%edi
  2d:   b8 00 00 00 00          mov    $0x0,%eax
  32:   e8 00 00 00 00          callq  37 <main+0x37>
  37:   8b 55 fc                mov    -0x4(%rbp),%edx
  3a:   48 63 d2                movslq %edx,%rdx
  3d:   89 04 95 00 00 00 00    mov    %eax,0x0(,%rdx,4)
  44:   83 45 f8 02             addl   $0x2,-0x8(%rbp)
  48:   83 7d f8 0f             cmpl   $0xf,-0x8(%rbp)
  4c:   7e d1                   jle    1f <main+0x1f>
[...]
000000000000008a <funcion>:
  8a:   55                      push   %rbp
  8b:   48 89 e5                mov    %rsp,%rbp
  8e:   89 7d fc                mov    %edi,-0x4(%rbp)
  91:   8b 45 fc                mov    -0x4(%rbp),%eax
  94:   48 98                   cltq
  96:   8b 14 85 00 00 00 00    mov    0x0(,%rax,4),%edx
  9d:   89 d0                   mov    %edx,%eax
  9f:   c1 e0 02                shl    $0x2,%eax
  a2:   01 d0                   add    %edx,%eax
  a4:   01 c0                   add    %eax,%eax
  a6:   5d                      pop    %rbp
  a7:   c3                      retq
[...]
```

Figura I.3: Parte del código máquina generado al compilar el programa de ejemplo, una vez desensamblado.

21. **[*Test*]** Exactamente, ¿qué representan los números de la primera columna de la figura (`1d:`, `1f:`, etc.)?

a) Se trata de las direcciones absolutas de memoria en las que se ubicarán finalmente las correspondientes instrucciones.

b) Se trata de desplazamientos relativos al comienzo del código (etiqueta `main:`), que aún no han sido reubicados.

c) Se trata de la longitud en bytes de las respectivas instrucciones.

d) se trata del código máquina de las respectivas instrucciones, que ocupa siempre un byte.

22. **[*Test*]** ¿Cuántos bytes de memoria ocupa cada instrucción máquina codificada del ISA Intel x86-64?

a) 1 byte.

b) 2 bytes.

c) 4 bytes.

d) Cada instrucción ocupa un número variable de bytes.

23. **[*Test*]** A la vista del código máquina correspondiente, indica cuál es la única de entre las siguientes instrucciones en la que hay un valor inmediato codificado directamente en el código máquina de la instrucción:

a) `movslq %edx,%rdx` `(48 63 d2)`

b) `add %eax,%eax` `(01 c0)`

c) `pop %rbp` `(5d)`

d) `retq` `(c3)`

e) `cmpl $0xf,-0x8(%rbp)` `(83 7d f8 0f)`

24. **[*Razonar*]** Muchos bytes del código máquina generado (segunda columna de la figura) son ceros (`00`). ¿Puedes explicar por qué se generan estos ceros, y qué ocurrirá con ellos cuando se genere el ejecutable final y se cargue en memoria?

25. **[*Emparejar*]** Ordenar, a la vista de la figura I.3, las siguientes instrucciones de menor a mayor longitud en bytes, una vez codificadas:

`mov 0x0(,%rax,4),%edx`	Más corta
`push %rbp`	Segunda más corta
`addl $0x2,-0x8(%rbp)`	Tercera más corta
`cltq`	Segunda más larga
`mov 0x4(%rbp),%eax`	Más larga

26. **[*Razonar*]** ¿Qué programa ejecutable crees que ocupará más espacio en disco, uno enlazado estáticamente o uno enlazado dinámicamente? Razona brevemente tu respuesta.

27. **[*Emparejar*]** Emparejar los siguientes comandos de linux con el uso que se les ha dado a lo largo de las prácticas del tema 5:

`gcc`	Cargar un programa en memoria y ejecutarlo paso a paso.
`ldd`	Compilar un programa fuente en C / ensamblador x86-64.
`objdump`	Comprobar las librerías con las que está enlazado un ejecutable dinámico.
`gdb`	Volcar un código objeto en pantalla, desensamblándolo.

En la figura I.4 se muestra parte de una sesión de `gdb` al cargar en memoria el ejecutable generado al compilar el programa C de la figura I.1 con el comando `gcc-4.8 -g main.c -o main`. Observar atentamente dicha sesión y contestar a las preguntas relacionadas:

```
(gdb) disassemble main
Dump of assembler code for function main:
[...]
   0x000000000040059a <+29>:    jmp    0x4005c5 <main+72>
   0x000000000040059c <+31>:    mov    -0x8(%rbp),%eax
   0x000000000040059f <+34>:    add    $0x1,%eax
   0x00000000004005a2 <+37>:    mov    %eax,-0x4(%rbp)
   0x00000000004005a5 <+40>:    mov    -0x8(%rbp),%eax
   0x00000000004005a8 <+43>:    mov    %eax,%edi
   0x00000000004005aa <+45>:    mov    $0x0,%eax
   0x00000000004005af <+50>:    callq  0x400607 <funcion>
   0x00000000004005b4 <+55>:    mov    -0x4(%rbp),%edx
   0x00000000004005b7 <+58>:    movslq %edx,%rdx
   0x00000000004005ba <+61>:    mov    %eax,0x601060(,%rdx,4)
   0x00000000004005c1 <+68>:    addl   $0x2,-0x8(%rbp)
   0x00000000004005c5 <+72>:    cmpl   $0xf,-0x8(%rbp)
   0x00000000004005c9 <+76>:    jle    0x40059c <main+31>
[...]
   (gdb) disassemble funcion
Dump of assembler code for function funcion:
   0x0000000000400607 <+0>:     push   %rbp
   0x0000000000400608 <+1>:     mov    %rsp,%rbp
   0x000000000040060b <+4>:     mov    %edi,-0x4(%rbp)
   0x000000000040060e <+7>:     mov    -0x4(%rbp),%eax
   0x0000000000400611 <+10>:    cltq
   0x0000000000400613 <+12>:    mov    0x601060(,%rax,4),%edx
   0x000000000040061a <+19>:    mov    %edx,%eax
   0x000000000040061c <+21>:    shl    $0x2,%eax
   0x000000000040061f <+24>:    add    %edx,%eax
   0x0000000000400621 <+26>:    add    %eax,%eax
   0x0000000000400623 <+28>:    pop    %rbp
   0x0000000000400624 <+29>:    retq
End of assembler dump.
```

Figura I.4: Sesión de `gdb` al cargar en memoria el ejecutable correspondiente al código C de la figura I.1.

28. **[*Rellenar*]** A la vista de dicho desensamblado, pueden deducirse con mayor o menor facilidad las siguientes direcciones de 64 bits (expresarlas todas en hexadecimal): la función `main` comienza exactamente en la dirección _____, mientras que la función `funcion` comienza en la dirección _____; por su parte, el array `array` comienza en la dirección _____, mientras que el primer bucle `for` (que se cierra con la instrucción `jle`) comienza en la dirección _____ (es decir, justo a donde se dirije dicho salto condicional).

29. **[*Rellenar*]** La sesión de `gdb` continúa en la figura I.5. En dicha figura se observa el volcado hexadecimal de los bytes de código máquina a partir de la dirección en la que finalmente se ha ubicado el código de la función `main`. Comparando dicha figura con la anterior figura I.3, podemos deducir que la dirección final con la que se han rellenado los ceros que había en la instrucción situada en el desplazamiento `3d` (es decir, `mov %eax,0x0(,%rdx,4)`, codificada inicialmente como `89 04 95 00 00 00 00` en la figura I.3) es exactamente _____.

30. **[*Rellenar*]** Comparando de nuevo las figuras I.5 y I.3, el rango de direcciones finales en los que se han ubicado

```
(gdb) x/80b main
0x40057d <main>:        0x55    0x48    0x89    0xe5    0x48    0x83    0xec    0x10
0x400585 <main+8>:      0xc7    0x45    0xf8    0x00    0x00    0x00    0x00    0xc7
0x40058d <main+16>:     0x45    0xfc    0x00    0x00    0x00    0x00    0xc7    0x45
0x400595 <main+24>:     0xf8    0x00    0x00    0x00    0x00    0xeb    0x29    0x8b
0x40059d <main+32>:     0x45    0xf8    0x83    0xc0    0x01    0x89    0x45    0xfc
0x4005a5 <main+40>:     0x8b    0x45    0xf8    0x89    0xc7    0xb8    0x00    0x00
0x4005ad <main+48>:     0x00    0x00    0xe8    0x53    0x00    0x00    0x00    0x8b
0x4005b5 <main+56>:     0x55    0xfc    0x48    0x63    0xd2    0x89    0x04    0x95
0x4005bd <main+64>:     0x60    0x10    0x60    0x00    0x83    0x45    0xf8    0x02
0x4005c5 <main+72>:     0x83    0x7d    0xf8    0x0f    0x7e    0xd1    0xc7    0x45
```

Figura I.5: Sesión de `gdb` al cargar en memoria el ejecutable correspondiente al código C de la figura I.1 (cont.).

los siete bytes en código máquina (los bytes `89 04 95 00 00 00 00`, correspondientes a la instrucción `mov %eax,0x0(,%rdx,4)`, antes de cambiar los ceros por direcciones finales reubicadas), van de la dirección _____ a la dirección _____, ambas inclusive.

31. **[*Razonar*]** La sesión de `gdb` sigue en la figura I.6. Explicar detalladamente qué es exactamente lo que se está mostrando en dicha figura.

```
(gdb) x/64b array
0x601060 <array>:       0xfc    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x601068 <array+8>:     0xfd    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x601070 <array+16>:    0xfe    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x601078 <array+24>:    0xff    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x601080 <array+32>:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x601088 <array+40>:    0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x601090 <array+48>:    0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x601098 <array+56>:    0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figura I.6: Sesión de `gdb` al cargar en memoria el ejecutable correspondiente al código C de la figura I.1 (III).

32. **[*Rellenar*]** Observando atentamente la figura I.6, podemos comprobar que el comienzo del array ha sido ubicado definitivamente en la dirección _____; así pues, en las direcciones `0x601070` a `0x601073` se encuentra la posición `array[k]` del array, con `k` = _____ (contestar con el índice entero correspondiente); dicha posición contiene el valor de 32 bits _____ (expresarlo en código hexadecimal de 8 dígitos), que se corresponde con el entero decimal _____ (expresarlo como el entero correspondiente en decimal, sabiendo que, como entero con signo, está representado en complemento a 2); se observa claramente que los enteros de 32 bits están almacenados siguiendo el criterio _____ *endian*.

33. **[*Rellenar*]** Sin necesidad de ejecutar el programa, y simplemente a partir de la interpretación del código en C original (figura I.1), deducir los valores de los bytes que habrá almacenados finalmente en las direcciones `0x601064` a `0x601067` (omitidos en la figura I.7), sabiendo que dicho volcado de los contenidos de memoria correspondientes ha sido obtenido justo después de detener el programa tras la finalización del primer bucle.

34. **[*Emparejar*]** Una vez que sabemos la ubicación definitiva de la etiqueta `array:`, empareja las siguientes direcciones de memoria finales con las correspondientes posiciones de elementos del array:

```
(gdb) x/64b array
0x601060 <array>:       0xfc    0xff    0xff    0xff    ____    ____    ____    ____
0x601068 <array+8>:     0xfd    0xff    0xff    0xff    0xe2    0xff    0xff    0xff
0x601070 <array+16>:    0xfe    0xff    0xff    0xff    0xec    0xff    0xff    0xff
0x601078 <array+24>:    0xff    0xff    0xff    0xff    0xf6    0xff    0xff    0xff
0x601080 <array+32>:    0x01    0x00    0x00    0x00    0x0a    0x00    0x00    0x00
0x601088 <array+40>:    0x02    0x00    0x00    0x00    0x14    0x00    0x00    0x00
0x601090 <array+48>:    0x03    0x00    0x00    0x00    0x1e    0x00    0x00    0x00
0x601098 <array+56>:    0x04    0x00    0x00    0x00    0x28    0x00    0x00    0x00
```

Figura I.7: Sesión de `gdb` al cargar en memoria el ejecutable correspondiente al código C de la figura, habiendo detenido el programa justo después de la terminación del primer bucle `for` de la función `main` (IV).

0x601070	`array[0]`
0x60106c	`array[3]`
0x60109c	`array[4]`
0x601090	`array[12]`
0x601060	`array[15]`

35. **[*Rellenar*]** Si, en el momento indicado en las preguntas anteriores (justo al terminar el primer bucle `for`, y antes de comenzar el segundo), se tecleara en `gdb` el comando `info registers`, sabemos que se obtendrían los valores de los registros del procesador. El valor del registro `%rip` en ese momento sería exactamente _____. (Pista: Es un problema para pensar un poco, e interrelacionar varias figuras: en la figura I.4 podemos observar la dirección de la última instrucción del bucle for, es decir, el salto condicional; por otro lado, en la figura I.3 podemos saber el tamaño exacto que dicha instrucción tiene una vez codificada; de ambas informaciones se puede deducir el siguiente valor del contador de programa, que es exactamente lo que pide el ejercicio).

Soluciones a las cuestiones de autoevaluación

S5.1. Soluciones

1. *Solución:* Recorre un array llamado `array`, de 16 posiciones, y en cada posición impar (1, 3, 5, 7, etc.) va escribiendo el valor de la posición par anterior (0, 2, 4, 6, etc.) multiplicado por 10.

2. *Solución:* c)

3. *Solución:*

```
array:                  array[0]
array: + 4              array[1]
array: + 16             array[4]
array: + 32             array[8]
array: + 48             array[12]
```

4. *Solución:* `.data` y `.text`

5. *Solución:* Etiquetas: acaban con `:` (p.e. `array:` o `funcion:`); etiquetas generadas automáticamente: comienzan por `.` (p.e. .LC0: o .L3); constantes: comienzan con el carácter `$`.

6. *Solución:* c)

7. *Solución:* a)

8. *Solución:*

`jmp .L2`	Salto incondicional
`movl -8(%rbp),%eax`	Movimiento de memoria a registro
`call funcion`	Llamada a subrutina
`movl %eax,array(,%rdx,4)`	Movimiento de registro a dirección de memoria calculada
`addl $2,-8(%rbp)`	Suma de una constante a una variable en memoria
`ret`	Retorno de procedimiento
`cmpl $15,-8(%rbp)`	Comparación de constante y variable en memoria
`jle .L3`	Salto condicional
`movslq %edx,%rdx`	Movimiento entre registros de distintos tamaños
`pushq %rbp`	Añadir el contenido de un registro a la cima de la pila
`popq %rbp`	Retirar el contenido de un registro desde la cima de la pila
`sall $2,%eax`	Multiplicar contenido de un registro por 4 usando desplazamiento de bits

9. *Solución:* Se compara el valor almacenado en la variable local `i`, contenida en la dirección de pila apuntada por `-8(%rbp)`, con la constante numérica 15 y, en caso de que dicho valor sea menor o igual que 15, se salta al lugar del código apuntado por la etiqueta `.L3`. En caso contrario, se sigue normalmente por la siguiente instrucción.

10. *Solución:* Se corresponde con la comprobación del final del bucle ($i < 16 \Leftrightarrow i <= 15$).

11. *Solución:* b)

12. *Solución:* c)

13. *Solución:* `addl $2,-8(%rbp)`

14. *Solución:* `-8(%rbp)`; `-4(%rbp)`; `movl $0,-4(%rbp)`; `movl -4(%rbp),%edx`.

15. *Solución:*

`%rip`	Contador de programa
`%rax`	Registro de uso general
`%rbx`	Registro de uso general
`%rcx`	Registro de uso general
`%rdx`	Registro de uso general
`%rsp`	Cima de la pila
`%rbp`	Registro base para apuntar a la pila

16. *Solución:* a)

17. *Solución:* `ret`

18. *Solución:* La instrucción `ret` recupera de la pila el contador de programa (`%rip`) que se almacenó en el momento de la última llamada a función (instrucción `call`). Dicha dirección apunta a la siguiente instrucción que seguía al `call`, y al ser asignada al registro `%rip` provoca que se realice el salto de vuelta de la función.

19. *Solución:* c)

20. *Solución:* `pushq`, `popq`, y `%rsp`

21. *Solución:* b)

22. *Solución:* d)

23. *Solución:* e)

24. *Solución:* Dichos ceros corresponden a direcciones de datos que aún no se conocen, al no estar el programa aún ubicado en unas direcciones concretas de memoria. Así, el compilador los deja a ceros mientras tanto. Cuando se genere el ejecutable final, y éste se cargue en memoria, entonces dichas direcciones sí que se conocerán definitivamente, con lo que en el código máquina definitivo dichos ceros serán sustituidos por direcciones reales.

25. *Solución:*

`push %rbp`	Más corta
`cltq`	Segunda más corta
`mov 0x4(%rbp),%eax`	Tercera más corta
`addl $0x2,-0x8(%rbp)`	Segunda más larga
`mov 0x0(,%rax,4),%edx`	Más larga

26. *Solución:* Uno enlazado estáticamente, ya que en él se "copia y pega" el código de las funciones/rutinas de librería utilizadas, mientras que en uno enlazado dinámicamente dicho código no se "pega" en el ejecutable, sino que es dinámicamente cargado en memoria desde el archivo con la librería dinámica correspondiente, en el momento de la ejecución.

27. *Solución:*

`gcc`	Compilar un programa fuente en C / ensamblador x86-64.
`ldd`	Comprobar las librerías con las que está enlazado un ejecutable dinámico.
`objdump`	Volcar un código objeto en pantalla, desensamblándolo.
`gdb`	Cargar un programa en memoria y ejecutarlo paso a paso.

28. *Solución:* `main` comienza en la dirección de 64 bits `0x40057d (0x40059a - 29)`, `funcion` en la dirección `0x400607`, `array` comienza en la dirección `0x601060` (se puede saber mirando las instrucciones que hacen referencia a él, por ejemplo la `mov 0x601060(,%rax,4),%edx`), y por último el bucle se cierra en la dirección 0x40059c (se sabe por la instrucción `jle 0x40059c`).

29. *Solución:* `0x00601060` (`0x60 0x10 0x60 0x00` en *little endian*).

30. *Solución:* Direcciones `0x4005ba` a `0x4005c0`.

31. *Solución:* Se trata del volcado hexadecimal de las direcciones de memoria en las que está ubicado el array `array`, antes de la ejecución del programa. Puesto que éste tiene 16 posiciones, cada una conteniendo un entero con signo de 32 bits (4 bytes), se muestran un total de $4 \times 16 = 64$ bytes.

32. *Solución:* El array comienza en la dirección `0x601060`; la posición del array indicada se corresponde a `k=4` (5ª posición del array, dado que empieza en `array[0]`); dicha posición contiene el valor de 32 bits `0xfffffffe`, que se corresponde exactamente con el valor entero -2, en C2 de 32 bits. El criterio utilizado es *little-endian*.

33. *Solución:* Dichas posiciones de memoria se corresponden a la segunda posición del array, esto es, `array[1]`. Dado que el programa en C consiste en un bucle que en cada posición de índice impar almacena justamente el valor almacenado en la posición par precedente, multiplicada por 10, y `array[0]` valía inicialmente `0xfffffffc` (`0xfc 0xff 0xff 0xff` en little-endian), es decir, -4 al ser interpretado en complemento a 2, se puede deducir que el valor final almacenado en la posición `array[1]` tendrá que ser $-4 \times 10 = -40$. Esto se escribe en complemento a dos de 32 bits como `0xffffffd8`, que hay que invertir byte a byte para ser almacenado en little endian, así que los bytes de las posiciones ocultas serán exactamente `0xd8`, `0xff`, `0xff` y `0xff`.

34. *Solución:*

0x601060	array[0]
0x60106c	array[3]
0x601070	array[4]
0x601090	array[12]
0x60109c	array[15]

35. *Solución:* La respuesta es `%rip=0x4005cb`. La explicación es que el registro `%rip`, al ser el contador de programa, tendrá que apuntar en el momento indicado a la instrucción siguiente a la de cierre de bucle, ésto es, el `jle` ubicado en la dirección `0x4005c9` (figura I.4). Puesto que dicha instrucción ocupa exactamente dos bytes (más concretamente, los bytes `7e d1`, como se puede comprobar en la figura I.3), el valor del `%rip` en dicho momento deberá ser exactamente `0x4005c9 + 2 = 0x4005cb`. Es un buen ejercicio práctico, que se recomienda al alumno, reproducir en el `gdb` la situación indicada, poniendo un punto de ruptura al comienzo del segundo bucle `for`, ejecutando el programa hasta ahí, y comprobando el valor del registro `%rip` en dicho momento.