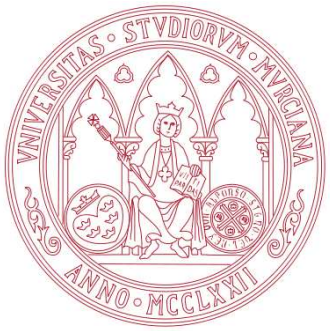

Universidad de Murcia

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 teoría / problemas

CURSO 2019 / 20

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

Índice general

Ejercicios resueltos

1. El contador de programa es un registro del procesador que:

   a) Cuenta el numero de instrucciones de programa ejecutadas hasta ahora.

   b) Nos dice el valor de la dirección de memoria principal donde se encuentra la siguiente instrucción a ejecutar.

   c) Se carga con la cantidad total de instrucciones que debe cargar el sistema operativo para ejecutar un proceso en sistemas operativos multitarea.

2. El registro de instrucción del procesador:

   a) Registra la cantidad de instrucciones del proceso activo que, hasta el momento, se han ejecutado.

   b) Contiene la dirección de memoria donde encontrar la próxima instrucción.

   c) Contiene el código máquina de la instrucción que actualmente se está ejecutando.

3. Sea una CPU un repertorio de 220 instrucciones distintas, 64 registros enteros de 32 bits de largo cada uno, y para la que se quiere implementar un cierto tipo de instrucciones aritmético-lógicas que tengan tres registros fuente y uno destino (p.e., `add Rd←Ra+Rb+Rc`). Si la longitud de las instrucciones codificadas es fija y de 32 bits, el formato más adecuado para esta instrucción será:

   a) 16 bits para el código de operación y 16 bits para codificar los cuatro registros `Ra`, `Rb`, `Rc` y `Rd`.

   b) 12 bits para el código de operación y 20 bits para codificar los cuatro registros `Ra`, `Rb`, `Rc` y `Rd`.

   c) 8 bits para el código de operación y 24 bits para codificar los cuatro registros `Ra`, `Rb`, `Rc` y `Rd`.

4. En una arquitectura del juego de instrucciones (ISA) de tipo RISC:

   a) Tenemos más riesgo de que algunas instrucciones de salto no se ejecuten, pero aún así sus procesadores suelen ser más rápidos.

   b) Suelen disponer de menos instrucciones más sencillas y ser éstas menos expresivas en cuanto a su capacidad de direccionamiento, con respecto a un ISA tipo CISC.

   c) Las instrucciones son muchas más y más variadas (como en el ISA de x86-64 visto en clase) que las de tipo CISC.

5. Las siglas RISC y CISC, referidas al diseño de computadores, significan exactamente, _____ y _____, respectivamente.

6. El lenguaje ensamblador se caracteriza por:

   a) Abarcar varios paradigmas de programación.

   b) Ser dependiente de la arquitectura de la máquina sobre la que se programa.

   c) Permitir al programador alejarse de las peculiaridades de la arquitectura del juego de instrucciones propia de la máquina sobre la que se traducirá el programa.

7. Un fichero objeto habitualmente contiene:

   a) Una cabecera, un segmento de código con instrucciones, otro de datos, información de reubicación, una tabla de símbolos e información de depuración.

   b) Imágenes, sonido o datos en cualquier tipo de formato que pueda ser utilizado por un programa de edición.

c) Las especificaciones formales del problema que ha de resolverse con el programa.

8. El enlazador (*linker*) es un programa que:

   a) Enlaza datos de memoria con los registros.

   b) Enlaza códigos objeto junto con funciones de librerías para generar el fichero ejecutable final.

   c) Enlaza en las posiciones de memoria que quedan libres, el código ejecutable y lo ejecuta.

9. Los ficheros que responden al comodín `/usr/lib/lib*.a` en Linux corresponden a _____, mientras que los que responden al comodín `/usr/lib/lib*.so` son _____. Indicar la principal ventaja de las segundas respecto a las primeras.

10. El cargador del sistema operativo lee un fichero ejecutable y:

    a) Lo carga en los registros del procesador para ejecutarlo a continuación.

    b) Lo carga en memoria principal, traduce las direcciones reubicables de algunas instrucciones dependiendo de su futura ubicación en memoria, y lo ejecuta.

    c) Carga el código del programa ejecutable en el procesador para que ejecute dicho programa, poniendo en memoria sólo los datos del programa.

11. De los siguientes programas, indicar cual es el único que toma como entrada código máquina:

    a) Compilador.

    b) Ensamblador.

    c) Enlazador.

12. El ISA de la arquitectura x86-64, en comparación con otros juegos de intrucciones, se caracteriza por:

    a) Disponer de una expresividad compleja, en cuanto al modo de direccionamiento, y una gran variedad de instrucciones.

    b) Tener todas las instrucciones el mismo tamaño de bytes.

    c) Ejecutar todas las instrucciones en el mismo número de ciclos del reloj.

13. Las instrucciones x86-64, una vez codificadas en memoria:

    a) Ocupan siempre 4 bytes de código objeto.

    b) Comienzan siempre en una dirección de memoria múltiplo de 4.

    c) Tienen una longitud variable en el código objeto (desde uno hasta varios bytes de longitud).

14. Si tenemos un array de datos de 32 bits en la variable array y queremos mover un elemento cuyo índice está ya cargado en el registro `%rax`, al registro `%ebx`, utilizaremos la instrucción del ensamblador del x86-64:

    a) `movl array(,%rax,4),%ebx`

    b) `movl 4(,%rax, array),%ebx`

    c) `movl %ebx,array(,%rax,4)`

15. Para apilar el contenido del registro `%rax` usaríamos el código ensamblador del x86-64 siguiente:

    a) `push %rax`, y ésta es la única forma de hacerlo.

    b) `sub $8, %rsp` primero y después `mov %rax, (%rsp)`, y ésta es la única forma de hacerlo.

    c) Las dos formas anteriores serían válidas para hacerlo.

16. En la arquitectura x86-64, los registros `%rax`, `%eax`, `%ax`, `%ah`, y `%al` tienen, respectivamente, de las siguientes longitudes en bits: _____, _____, _____, _____ y _____.

17. Mencionar los principales tipos de instrucciones de un repertorio (ISA) típico. Para cada tipo, dar un ejemplo de instrucción de ese tipo en el repertorio x86-64 de Intel (sólo el nombre -también llamado *mnemónico*- de la instrucción).

18. Considérese la siguiente sesión con el depurador `gdb` (los puntos suspensivos `[...]` indican simplemente que se ha suprimido la parte de la salida que no nos interesa para este ejercicio; aparecen tanto los comandos tecleados por el usuario como el resto de la salida producida por el terminal). Rellene todos los huecos del texto que va a continuación (indicando en el examen la correspondiente referencia al hueco para cada respuesta):

```
user@host:~/$ gdb ./main
[...]
(gdb) list
1       int array[6] = {-10,+20,-30,+40,-50,+60};
2       int main() {
[...]
(gdb) disassemble main
Dump of assembler code for function main:
[...]
0x00000000004005ac <+47>:    addl   $0x1,-0x4(%rbp)
0x00000000004005b0 <+51>:    cmpl   $0x5,-0x4(%rbp)
0x00000000004005b4 <+55>:    jle    0x40058e <main+17>
0x00000000004005b6 <+57>:    [...]
[...]
(gdb) x/10bx 0x4005ac
0x4005ac <main+47>:     0x83    0x45    0xfc    0x01    0x83    0x7d    0xfc    0x05
0x4005b4 <main+55>:     0x7e    0xd8
(gdb) x/24b array
0x601050 <array>:          A       B       C       D    0x14    0x00    0x00    0x00
0x601058 <array+8>:     0xe2    0xff    0xff    0xff    0x28    0x00    0x00    0x00
0x601060 <array+16>:    0xce    0xff    0xff    0xff    0x3c    0x00    0x00    0x00
```

"El código depurado en la sesión anterior manipula una tabla de __(a)__ elementos de tipo entero, que ocupará exactamente __(b)__ bytes en memoria, y que comienza en la dirección exacta __(c)__. La función principal `main`, por su parte, comienza exactamente en la dirección __(d)__. Los valores que aparecen sustituidos con las letras A, B, C y D eran originalmente, en realidad, los bytes __(e)__ (expresar los cuatro bytes en orden, y en formato 0xXX, con XX en hexadecimal). El código desensamblado que aparece en la figura se corresponde, probablemente, con un código de alto nivel que __(f)__ (dar aquí una explicación concisa pero completa de lo que hace dicho código). En particular, dentro de ese código la instrucción `cmpl` exactamente realiza lo siguiente: __(g)__. Un ejemplo de dirección concreta del programa depurado correspondiente con su segmento de datos sería la __(h)__, mientras que otra correspondiente a su segmento de texto sería la __(i)__. La instrucción ubicada en la dirección `0x4005ac` es de tipo __(j)__, mientras que la ubicada en la dirección `0x4005b4` es de tipo __(k)__. Podemos también afirmar que la instrucción `jle`, una vez ubicada en memoria, ocupa exactamente __(l)__ bytes, cuyos valores concretos son __(m)__ (expresar aquí de nuevo los bytes correspondientes en orden, y con formato 0xXX, con XX en hexadecimal). La instrucción `addl`, sin embargo, ocupa __(n)__ bytes, cuyos valores son __(ñ)__, y van a parar al rango de direcciones de byte comprendido entre la dirección __(o)__ y la __(p)__, ambas inclusive."

Nota: Los datos enteros de tipo int usados en el ejemplo ocupan 4 bytes cada uno, y están almacenados siguiendo el convenio little-endian.

19. Soporte a procedimientos en el x86-64 y pila:

    a) ¿Qué es y para qué sirve un procedimiento (también llamado subrutina, rutina o función)?

b) ¿Cón qué instrucción ensamblador del x86-64 se llama a un procedimiento desde el procedimiento llamador? ¿Y con qué instrucción acaba dicha rutina?

c) ¿Qué es la pila, y para qué funcionalidades importantes se utiliza en una llamada a procedimiento?

d) ¿Qué registros del procesador se emplean asociados al uso de la pila? ¿Cuál es la función de cada uno de ellos?

e) ¿Qué instrucciones emplean la pila explícitamente? ¿E implícitamente?

20. Dada la siguiente porción de código máquina de un programa llamado main.c responder, con respecto a él, las preguntas que seguidamente se formulan:

```
0000000000000000 <main>:
     [...]
  11:   8b 45 fc                mov    -0x4(%rbp),%eax
  14:   89 c7                   mov    %eax,%edi
  16:   b8 00 00 00 00          mov    $0x0,%eax
  1b:   e8 00 00 00 00          callq  20 <main+0x20>
  20:   8b 55 fc                mov    -0x4(%rbp),%edx
  23:   48 63 d2                movslq %edx,%rdx
  26:   89 04 95 00 00 00 00    mov    %eax,0x0(,%rdx,4)
     [...]
0000000000000073 <funcion>:
  73:   55                      push   %rbp
  74:   48 89 e5                mov    %rsp,%rbp
  77:   89 7d fc                mov    %edi,-0x4(%rbp)
  7a:   8b 45 fc                mov    -0x4(%rbp),%eax
  7d:   48 98                   cltq
  7f:   8b 04 85 00 00 00 00    mov    0x0(,%rax,4),%eax
  86:   0f af 45 fc             imul   -0x4(%rbp),%eax
  8a:   5d                      pop    %rbp
  8b:   c3                      retq
     [...]
```

a) ¿Qué niveles de la jerarquía de traducciones nos encontramos en este listado?

b) ¿Qué representan los números de la primera columna del código (`11:`, `14:`, `16:`, `1b:`, etc.)?

c) ¿Qué representan los números de la segunda columna del código (`8b 45 fc`, `89 c7`, etc.)?

d) ¿Cuántos bytes ocupan en memoria (en total) todas las instrucciones de la función `funcion`?

e) Indica cuáles de las instrucciones de todo el código (tanto de `funcion` como de `main`) modifican de un modo u otro la pila, explicando con una breve frase para cada una lo que hacen.

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

Ejercicios propuestos

1. Una etiqueta en un código fuente en ensamblador (por ejemplo, `etiq:`), representa:

   a) Una dirección del segmento de código (también llamado segmento de texto), siempre.

   b) Una dirección del segmento de datos, siempre.

   c) Una dirección que puede pertenecer al segmento de datos o al segmento de código (o texto).

2. Sólo una de las siguientes instrucciones del ensamblador x86-64 modifica la pila ¿Cuál?

   a) `retq`

   b) `movl %rbp,%rax`

   c) `jmp label`

3. La instrucción `call rutina` del repertorio del x86-64:

   a) Salta a la dirección de código apuntada por la etiqueta `rutina:`, y apila el `%rip` actual para poder luego volver a la instrucción siguiente al `call`.

   b) Es funcionalmente idéntica a una instrucción `jmp rutina`.

   c) Sirve para provocar la carga en memoria de la librería que contiene a la función `rutina`.

4. Define los siguientes conceptos de una jerarquía de traducción, explicando brevemente la interrelación entre ellos:

   a) Lenguaje de alto nivel.

   b) Compilador.

   c) Ensamblador.

   d) Fichero objeto.

   e) Librerías.

   f) Enlazador (linker).

   g) Cargador (loader).

5. Considere la siguiente sesión interactiva con el depurador `gdb` (los puntos suspensivos `[...]` indican que se ha suprimido la parte de la salida que no nos interesa para este ejercicio; aparecen tanto los comandos tecleados por el usuario como el resto de la salida producida por el terminal; la salida de los volcados de bytes utiliza almacenamiento de tipo little-endian):

```
user@host:~/$ gdb ./main
[...]
(gdb) l main
1       #include<stdio.h>
2       int array[10] = {10,20,30,40,50};
3       int main() {
4           int i;
5           for(i=0;i<5;i++)
6               array[i] = funcion(i);
[...]
(gdb) disassemble main
Dump of assembler code for function main:
[...]
```

```
0x000000000040058e <+17>:    mov    -0x4(%rbp),%eax
0x0000000000400591 <+20>:    mov    %eax,%edi
0x0000000000400593 <+22>:    mov    $0x0,%eax
0x0000000000400598 <+27>:    callq  0x4005f0 <funcion>
0x000000000040059d <+32>:    mov    -0x4(%rbp),%edx
0x00000000004005a0 <+35>:    movslq %edx,%rdx
0x00000000004005a3 <+38>:    mov    %eax,0x601060(,%rdx,4)
0x00000000004005aa <+45>:    [...]
[...]
(gdb) x/28bx 0x40058e
0x40058e <main+17>:     0x8b    0x45    0xfc    0x89    0xc7    0xb8    0x00    0x00
0x400596 <main+25>:     0x00    0x00    0xe8    0x53    0x00    0x00    0x00    0x8b
0x40059e <main+33>:     0x55    0xfc    0x48    0x63    0xd2    0x89    0x04    0x95
0x4005a6 <main+41>:     0x60    0x10    0x60    0x00
(gdb) x/20b 0x601060
0x601060 <array>:          A    0x00    0x00    0x00       B    0x00    0x00    0x00
0x601068 <array+8>:     0x1e    0x00    0x00    0x00    0x28    0x00    0x00    0x00
0x601070 <array+16>:    0x32    0x00    0x00    0x00
```

a) Indique para qué sirven cada uno de los cuatro comandos de `gdb` utilizados:

- `l main`
- `disassemble main`
- `x/28bx 0x40058e`
- `x/20b 0x601060`

b) ¿Qué significado tienen los bytes impresos por el comando `x/28bx 0x40058e`?

c) ¿Cuál es la instrucción máquina más corta (en bytes) de las 7 instrucciones en ensamblador mostradas? Indique su código máquina completo (secuencia de bytes en hexadecimal).

d) Repita el apartado anterior, en este caso para la instrucción más larga.

e) Indique la longitud en bytes y el código máquina completo correspondiente a la instrucción `callq`.

f) ¿A qué se corresponden los bytes impresos por el comando `x/20b 0x601060`?

g) En dicho volcado han sido suprimidos 2 bytes (Marcados con A y B). Indica exactamente los valores de los bytes que han sido suprimidos.

h) Explique, en términos de para lo que sirven en el contexto del programa en lenguaje de alto nivel, el cometido que cumplen *(1)* las tres instrucciones antes del `callq`, *(2)* el `callq` en sí, y *(3)* las tras instrucciones posteriores al `callq`.

6. **[Experimentación]** Partiendo del mismo código `main.c` usado en las prácticas de este tema, ir sustituyendo sucesivamente la instrucción `return array[parametro]*parametro;` de la función `funcion` por las instrucciones que se detallan a continuación, generando el código ensamblador resultante con el comando `gcc-4.8 -S`, y detallando la instrucción aritmético-lógica generada en cada caso:

a) Suma: `return array[parametro]+parametro;`

b) Resta: `return array[parametro]-parametro;`

c) AND lógico: `return array[parametro]&parametro;`

d) OR lógico: `return array[parametro]|parametro;`

e) XOR lógico: `return array[parametro]^parametro;`

f) NOT lógico: `return ~parametro;`

g) División entera: `return array[parametro]/parametro;`

h) Módulo de la división entera: `return array[parametro]%parametro;`

7. **[Experimentación]** Generar el código ensamblador correspondiente al sencillo programa en C siguiente (que trabaja con el tipo de datos `short int`, de 16 bits, en lugar del `int`, de 32 bits), y comprobar si en algún lugar se utiliza algún registro de tamaño inferior a 32 bits.

```
int main() {
    short int i=5, j=6, k=0;
    k = i-j;
}
```

8. **[Experimentación]** Repetir el ejercicio anterior con el siguiente código alternativo, en el que ahora se utiliza el tipo de datos `char`, correspondiente a enteros tipo byte (8 bits):

```
int main() {
   char i=5, j=6, k=0;
   k = i-j;
}
```

9. **[Investigación]** Dado el siguiente código (fichero `main-simple.c`), generar el código ensamblador resultante e investigar e intentar razonar sobre las dos cuestiones indicadas:

```
#include<stdio.h>
float array[10] = {10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0};
float funcion(int parametro) {
    return array[parametro]*parametro;
}
int main() {
    int i = 0;
    for(i=0;i<10;i++)
        array[i] = funcion(i);
    for(i=0;i<10;i++)
        printf("%f ",array[i]);
    printf("\n");
}
```

   a) ¿Qué crees que pueden significar las constantes numéricas que aparecen en el array en la zona del segmento de datos? ¿Serías capaz de decodificarlas, con lo aprendido sobre la representación en punto flotante en el tema 2 de la asignatura?

   b) ¿Observas algunas instrucciones y/o registros distintos a los vistos hasta ahora, en relación a la implementación en ensamblador de las operaciones en punto flotante?[1]

10. **[Investigación]** Dado el mismo programa del ejercicio anterior, generar un ejecutable depurable con el comando `gcc-4.8 -g main.c -o main`, e intentar repetir la secuencia de pasos descrita en el segundo boletín de prácticas (carga en memoria con `gdb`, ejecución paso a paso, acceso a la memoria de datos correspondiente al array, etc.). En particular, intentar relacionar con lo aprendido sobre representación de datos en punto flotante en el tema dos los resultados obtenidos al volcar el vector `array` de `floats` con los comandos *a)* `p array` y *(*b) `x/10w array`.

[1] Obviamente, aquí se trata sólo de detectar las instrucciones y los registros utilizados, no de intentar entender el código generado, que corresponde al uso de la unidad funcional de punto flotante de la arquitectura x86-64, que tiene cierta complejidad y que no hemos estudiado en clase.

11. **[Investigación]** Con unos cuantos programas ejecutables del sistema (localizados principalmente en los directorios `/bin` y `/usr/bin`), utilizar el comando `ldd` para saber si están enlazados dinámicamente o no. Localizar algún ejecutable enlazado estáticamente (no valen guiones de shell, han de ser ejecutables "puros")[2], y otros tres que estén enlazados dinámicamente, en concreto con la librería matemática básica del sistema (`libm.so`)[3].

[2]Es posible que sea difícil encontrar alguno, o incluso que no haya ninguno, dependiendo de la distribución e instalación concreta del sistema, ya que en general, como es lógico, hay una mayoría abrumadora de ejecutables enlazados dinámicamente, frente a los ejecutables estáticos, en general mucho mayores en tamaño.

[3]Pistas: utilizar el comando `ls | xargs ldd | less` una vez dentro del directorio `/bin` o `/usr/bin`, para localizar rápidamente tanto ejecutables estáticos como dinámicos, y comprobar también las librerías utilizadas por éstos últimos; y usar después el comando `file nombre_fichero` para comprobar si el fichero en cuestión es un ejecutable "puro" (*ELF executable*) o no.

Soluciones a los ejercicios resueltos

1. *Solución:* b)
2. *Solución:* c)
3. *Solución:* c)
4. *Solución:* b)
5. *Solución:* Reduced Instruction Set Computers (Computadores de conjunto de instrucciones reducido). Complex Instruction Set Computers (Computadores de conjunto de instrucciones complejo).
6. *Solución:* b)
7. *Solución:* a)
8. *Solución:* b)
9. *Solución:* Librerías estáticas. Librerías dinámicas. En las librerías dinámicas, el código de las funciones de biblioteca no se incluye en el ejecutable final,sino que éste simplemente almacena la información necesaria para cargar dicho código en memoria desde el fichero de la librería en el momento de la ejecución. Así, dichas funciones pueden ser compartidas por varios ejecutables simultáneamente, sin duplicar espacio necesario en memoria, con el consiguiente ahorro de espacio tanto en disco como en memoria.
10. *Solución:* b)
11. *Solución:* c)
12. *Solución:* a)
13. *Solución:* c)
14. *Solución:* a)
15. *Solución:* c)
16. *Solución:* 64, 32, 16, 8 y 8 bits.
17. *Solución:* Aritmético-lógicas (`add, sub, ...`), movimiento de datos (`mov, ...`), saltos condicionales (`jle, jge, ...`), saltos incondicionales (`jmp`), soporte para procedimientos (`call, ret, ...`). Otros tipos: punto flotante, entrada salida, interrupciones...
18. *a*) 6.
    b) 24.
    c) `0x601050`.
    d) `0x4005ac - 47 = 0x4005ac - 0x2f = 0x40057d`.
    e) `0xf6`, `0xff`, `0xff` y `0xff`.
    f) Incrementa una variable local en memoria (almacenada en la posición de pila apuntada por `-4(%rbp)`), y comprueba si es menor o igual que 5 (esto es, estrictamente menor que 6), en cuyo caso vuelve a ejecutar un bucle que comienza en la dirección `0x40058e`.
    g) Compara el valor contenido en la variable local apuntada por `-4(%rbp)` con la constante 5.

h) Por ejemplo, la `0x601050` (datos), aunque valen muchas otras, claro.

i) Por ejemplo, la `0x4005ac` (texto), aunque valen muchas otras, claro.

j) Aritmético-lógica.

k) Salto condicional.

l) Dos bytes.

m) `0x7e` y `0xd8`.

n) Cuatro bytes.

ñ) `0x83`, `0x45`, `0xfc` y `0x01`.

o) `0x4005ac`.

p) `0x4005af`.

19. *Solución:*

a) Una secuencia de instrucciones que ejecuta una determinada subtarea, actuando sobre unos parámetros de entrada (opcionales) y devolviendo, también de forma opcional, unos resultados. Su utilidad principal es que sirve para estructurar el programa en módulos más o menos independientes y reutilizables.

b) La instrucción ensamblador para llamar a un procedimiento desde el procedimiento principal es la instrucción `call`. Y la instrucción para volver, la `ret`.

c) La pila es una zona de memoria gestionada de forma especial por los procedimientos, usada principalmente para almacenar variables locales, la dirección de vuelta de la subrutina, salvar posibles registros que van a ser temporalmente machacados, y pasar posibles parámetros adicionales en el caso de que el uso de los registros no sea suficiente.

d) Los registros del procesador x86-64 que se emplean asociados al uso de la pila son el `%rsp` y el `%rbp`. El primero es el *stack pointer*, que marca en todo momento la cima de la pila. El segundo es el *base pointer*, cuyo objetivo es usarse como base para poder acceder a todos las posiciones de interés dentro de cada marco de pila correspondiente a cada subrutina.

e) Las instrucciones que usan explícitamente la pila son `push` y `pop`, para meter y sacar valores de la misma, respectivamente. Otras instrucciones que la usan implícitamente son, por ejemplo, `call` y `ret`, que la usan para guardar y recuperar, respectivamente, la dirección de retorno en una llamada a procedimiento. También la usan instrucciones de movimiento de datos que involucran a los registros `%rbp` o `%rbp`, como por ejemplo `mov %rax,(%rsp)`.

20. *a*) Código objeto (segunda columna) y código ensamblador (tercera columna).

b) Desplazamientos, en cantidad de bytes, y expresados en hexadecimal, del comienzo de cada instrucción máquina respecto al comienzo del programa (etiqueta `main:`).

c) Es el código máquina correspondiente a cada instrucción en ensamblador.

d) 25 bytes en total (1+3+3+3+2+7+4+1+1=25).

e) Instrucciones que modifican la pila:

- `callq 20 <main+0x20>`
  Llama a la rutina función, poniendo la dirección de retorno (el valor actual del registro `%rip`) en la cima de la pila.
- `push %rbp`
  Pone registro `%rbp` en cima de la pila.
- `mov %edi,-0x4(%rbp)`
  Pone el registro `%edi` (que contenía el parámetro pasado a la función) en la variable local (contenida en la pila) apuntada por `-0x4(%rbp)`.

- `pop %rbp`
  Retira registro `%rbp` de la cima de la pila.
- `retq`
  Vuelve al lugar posterior al `callq`, retirando el registro `%rip` de la cima de la pila.
- `mov %rsp, %rbp`
  Esta instrucción, finalmente, no modifica el contenido de la pila en sí, aunque sí afecta a uno de los punteros importantes a la misma, el `%rbp`, que a partir de este momento apuntará al nuevo marco de pila (correspondiente a la función `funcion`).

f) Son espacios reservados para poner direcciones de memoria, que como aún no se saben hasta que el programa esté completamente enlazado y cargado en memoria, se dejan a `00` por el momento. Es el caso de todos los `00` que aparecen en el código mostrado, excepto para la instrucción `b8 00 00 00 00` (`mov $0x0, %eax`), donde estos ceros significan, simplemente, la constante `0x00000000` de 32 bits.