LABORATORIO DE PROGRAMACIÓN EN LENGUAJE ENSAMBLADOR x86-16bits

Programa DEBUG: ensamblado y trazado de instrucciones

Objetivo

El primer objetivo de esta práctica es familiarizarse con el potente programa DEBUG. Entre las numerosas facilidades que nos ofrece esta aplicación se encuentra la de ensamblar una secuencia de instrucciones y la posibilidad de ejecutar dicha secuencia paso a paso permitiéndonos examinar todo el *contexto* (registros, banderas de estado y memoria) en cada parada.

El segundo objetivo es comprender la diferencia entre un programa y una secuencia de instrucciones ensamblada. Efectivamente, el código máquina de una secuencia de instrucciones no constituye por sí solo una imagen válida de un ejecutable.

El programa DEBUG

El programa DEBUG es un *depurador*, una herramienta destinada a facilitarnos encontrar errores o ayudarnos a mejorar el funcionamiento de una aplicación durante su desarrollo. Para ello, un depurador muestra el contenido de la memoria y de los registros en cualquier instante de la ejecución de un programa. Además, nos permite modificar el contenido de secciones completas de memoria o dar valor a variables o registros.

El depurador DEBUG cuenta con la capacidad adicional de permitirnos ensamblar secuencias de instrucciones. Luego, ese código ensamblado, podemos depurarlo, ejecutarlo e incluso salvarlo en un fichero.

Las funciones que puede realizar DEBUG se enumeran a continuación:

- cargar la imagen de un ejecutable
- mostrar el código fuente de un programa junto con su código máquina y la dirección de cada instrucción
- mostrar el contenido de los registros y las banderas de estado
- ejecutar o trazar paso a paso programas o secuencias de instrucciones
- introducir valores en memoria o en registros
- buscar valores binarios o ASCII en memoria
- mover bloques de memoria de un lugar a otro
- rellenar bloques de memoria
- acceder en lectura y escritura tanto a ficheros como a sectores de disco
- ensamblar secuencias de instrucciones
- admite *scripts*

Depuración de un programa

Para depurar un programa existente, invocamos DEBUG pasándole como argumento el nombre del programa:

```
C:>debug programa.exe
```

Hay que indicar que DEBUG es un depurador a nivel de ensamblador. Esto significa que el código fuente que muestra es siempre lenguaje ensamblador independientemente del lenguaje con el que se haya escrito el código fuente original. Es decir, lo que hace es desensamblar el código de la imagen del ejecutable.

Para ver la memoria y los registros del procesador o para ensamblar una secuencia de instrucciones, invocamos DEBUG sin argumentos.

```
C:>debug
```

Comando ?

El comando ? (ayuda) muestra el listado completo de comandos disponibles y un recordatorio de los argumentos que admiten tal y como se puede ver a continuación.

```
-?
assemble     A [dirección]
compare      C intervalo de direcciones
dump         D [intervalo]
enter        E dirección [lista]
fill         F lista de rango
go           G [=dirección] [direcciones]
hex          H valor1 valor2
input        I puerto
load         L [dirección] [unidad] [primer sector] [número]
move         M intervalo de direcciones
name         N [ruta] [lista de argumentos]
output       O byte de puerto
proceed      P [=dirección] [número]
quit         Q
register     R [registro]
search       S lista de rango
trace        T [=dirección] [valor]
unassemble   U [intervalo]
write        W [dirección] [unidad] [primer sector] [número]
allocate expanded memory        XA [N.páginas]
deallocate expanded memory      XD [identificador]
map expanded memory pages       XM [páginaL] [páginaP] [identificador]
display expanded memory status  XS
```

Seguidamente, veremos algunos de ellos con más detalle.

Comando Q

El comando Q (*quit*) cierra el depurador DEBUG y devuelve el control al DOS.

Comando A

El comando A (*assemble*) sirve para ensamblar las instrucciones que se le pasen seguidamente. Opcionalmente se le puede pasar una dirección como argumento. La dirección se puede pasar como una base y un desplazamiento explícitos, un registro de segmento y un desplazamiento o sólo un desplazamiento que tomará como base implícita el valor de CS. Para dejar de ensamblar hay que pulsar 'enter' en una línea vacía.

Ejemplos:

```
-A                  ensambla a partir de la posición actual
-A 100              ensambla a partir de CS:0100h
-A DS:500           ensambla a partir de DS:0500h
-A 05FB:0100        ensambla a partir de 05FB:0100h
05FB:0100   mov ax, 10
05FB:0103   mov bx, 20
05FB:0106   add ax, bx
05FB:0108
-
```

Comando D

El comando D (*dump*) presenta el contenido de la memoria. Muestra los bytes de cada posición tanto en hexadecimal como en ASCII. La base por defecto, usada cuando no se especifica explicitamente, es DS.

Ejemplos:

```
-D                  muestra a partir de la posición actual o DS:0 si es la primera vez
-D 100              muestra a partir de DS:0100h
-D ES:500           muestra a partir de ES:0500h
-D 05FB:0100        muestra a partir de 05FB:0100h
-D 100 110          muestra desde DS:0100h hasta DS:0110h (rango)
-D 100 L 20         muestra 20h bytes desde DS:0100h (lista)
```

Comando F

El comando F (*fill*) rellena un rango de memoria con un valor o con una lista de valores.

Ejemplos:

`-F 100 400, 0`	rellena el rango entre 100 y 400 con '0'
`-F 100 110, CF`	rellena el rango entre 100 y 110 con 0CFh
`-F 100 L20 'A'`	rellena 20 bytes a partir de 100 con el carácter 'A'
`-F 100 110 'AB'`	rellena el rango entre 100 y 110 con 'AB'

Comando G

El comando G (*go*) ejecuta el programa que se está depurando o bien ejecuta lo que hay en memoria a partir de una posición dada. También se pueden insertar hasta 10 puntos de ruptura.

Ejemplos:

`-G`	ejecuta el programa desde la posición en curso hasta el final
`-G 200`	ejecuta desde la posición en curso hasta la de desplazamiento 200
`-G=100`	ejecuta desde la posición CS:100
`-G=100 400`	ejecuta desde la posición CS:100 hasta la de desplazamiento 400

Comando L

El comando L (*load*) carga un fichero si ha sido declarado con el comando N o sectores del disco en caso contrario, en la posición de memoria indicada. Si no se indica dirección, se asume la CS:100. El número total de bytes leídos se expresa en BX:CX.

Ejemplos:

`-L`	carga el fichero declarado con N en la posición CS:100
`-L DS:100`	carga el fichero declarado con N en la posición DS:100
`-L 300 2 A 4`	carga 4 sectores del disco C a partir del 0Ah en la posición CS:300

Comando N

El comando N (*name*) declara el fichero que se va a utilizar en combinación con los comandos L y W.

Ejemplos:

`-N c:\codigo.bin`	declara el fichero de la ruta indicada

Comando R

El comando R (*register*) muestra el contenido de los registros así como la siguiente instrucción a ejecutar. También puede mostrar el contenido de un registro individual y, en ese caso, permite cambiar su valor.

Ejemplos:

`-R`	muestra el contenido de todos los registros
`-R BX`	muestra el contenido del registro BX y espera por un nuevo valor
`-R IP`	muestra el contenido del registro IP y espera por un nuevo valor
`-R F`	muestra el contenido de los *flags* y espera por un nuevo valor

La tabla siguiente muestra los códigos nominales que identifican los valores que adoptan los *flags*.

***flag* a 1**	***flag* a 0**
OV = desbordamiento	**NV** = no desbordamiento
DN = dirección decreciente	**UP** = dirección creciente
EI = interrupción habilitada	**DI** = interrupción deshabilitada
NG = signo negativo	**PL** = signo positivo
ZR = cero	**NZ** = no cero
AC = acarreo auxiliar	**NA** = no acarreo auxiliar
PO = paridad impar	**PE** = paridad par
CY = acarreo	**NC** = no acarreo

Comando T

El comando T (*trace*) ejecuta las instrucciones en modo paso a paso mostrando en pantalla el contenido de los registros después de la ejecución de cada una. A partir de la dirección en curso, puede ejecutar una sola instrucción o un conjunto de ellas.

Ejemplos:

`-T`	ejecuta la siguiente instrucción
`-T 10`	ejecuta las 16 instrucciones siguientes
`-T=200 10`	ejecuta las 16 instrucciones que se encuentran a partir de CS:200

Comando U

El comando U (*unassemble*) desensambla el contenido de memoria escribiendo los mnemónicos correspondientes. Si no se le pasa ninguna dirección, comienza en CS:100 o en la última posición en la que se usó.

Ejemplos:

`-U`	desensambla a partir de CS:100 (si es la primera vez que se invoca)
`-U 500`	desensambla a partir de CS:500
`-U 200 500`	desensambla entre CS:200 y CS:500

Comando W

El comando W (*write*) escribe un bloque de memoria en un fichero si ha sido declarado con el comando N o en sectores del disco en caso contrario. El tamaño del bloque de memoria se ha de expresar en BX:CX. La dirección de comienzo es CS:100 si no se expresa explícitamente.

Ejemplos:

`-N codigo.bin`	declara "codigo.bin" como fichero por defecto
`-R BX 0 R CX 100`	establece como tamaño 256 bytes (100h)
`-W`	escribe en "codigo.bin" los 256 bytes a partir de CS:100
`-W 300 2 A 4`	escribe 4 sectores del disco C a partir del sector 0Ah con el contenido de la memoria a partir de la posición CS:300

La escritura directa en sectores es extremadamente peligrosa si no se hace cuidadosamente ya que podría borrar el disco completo.

Procesando *scripts*

El programa DEBUG admite secuencias de comandos escritos en ficheros de texto (ficheros de *script*) que se le pasan mediante comandos de redireccionamiento (<). Por ejemplo, supongamos que queremos ensamblar a partir de la dirección CS:100 una serie de instrucciones. Escribiríamos el fichero de *script* `miscript.txt` de esta manera:

```
A 100
mov ax, 10
mov bx, 20
add ax, bx

Q
```

Obsérvese como finalizamos la secuencia de comandos con el comando Q (*quit*) para devolver el control al DOS. Véase también como, tras la serie de instrucciones, se añade un retorno de carro para que finalice el ensamblado. Para pasar el *script* al DEBUG haremos lo siguiente:

```
C:>debug < miscript.txt
```

Prácticas

A) Creación de una secuencia de código con DEBUG

Utilizando el depurador DEBUG vamos a crear una secuencia de código. La secuencia de código dispondrá de una pequeña área de datos en la cabecera (posiciones bajas de memoria) y ejecutará algunas operaciones de proceso y de transferencia. La secuencia se salvará como un fichero de nombre 'codigo.bin'. Hay que recordar que ese fichero no es un ejecutable ya que no cuenta con una cabecera reconocible por parte del sistema operativo.

Realizaremos las siguientes tareas:

- Comenzamos invocando el comando **R** y observando el estado de los registros.

```
-R
```

- Declaramos un fichero por defecto para las transferencias con el sistema de ficheros utilizando el comando **N**

```
-N c:\codigo.bin
```

- Volcamos el contenido de 16 bytes de memoria a partir de DS:100 invocando el comando **D**.

```
-D 100 l 10
```

- Rellenamos los 16 bytes de memoria a partir de DS:100 con 0 invocando el comando **F**. Estos 16 bytes serán la cabecera de la secuencia de código.

```
-F 100 l 10 0
```

- Comprobamos que la memoria se ha rellenado como queríamos usando el comando **D** de nuevo.

```
-D 100 l 10
```

- Ensamblamos con el comando **A** el siguiente código a partir de CS:110:

```
-A 110
CS:0110 mov ax, cs
CS:0112 add ax, 10
CS:0115 mov ds, ax
CS:0117 mov al, 11
CS:0119 mov bl, 22
CS:011B xchg bh, bl
CS:011D add ax, bx
CS:011F mov [0000], al
CS:0122 mov [0001], ax
CS:0125 neg cx
CS:0127 sub cx, [0003]
CS:012B push cx
CS:012C pop dx
CS:012D mov si, 4
CS:0130 inc si
CS:0131 mov [si], ch
CS:0133 dec dx
CS:0134 mov di, 6161
CS:0137 mov [si+1], di
CS:013A dec si
CS:013B mov ax, [si]
CS:013D
```

- Cargamos BX con 0 usando el comando **R**.

```
-R BX
BX 0000
:0
```

- Cargamos CX con 3D (tamaño del código y la cabecera de datos) usando el comando **R**.

```
-R CX
CX 0000
:3D
```

- Escribimos el bloque de memoria definido como secuencia de código en el fichero 'codigo.bin' utilizando el comando **W**.

```
-W
```

- Desensamblamos el contenido de memoria a partir de CS:100 y a partir de CS:110 con el comando **U**. Veremos que el área de código, rellena con 0, también admite una interpretación como código.

```
-U 100
…
-U 110
```

La secuencia de comandos se puede escribir en un fichero de *script* para pasársela a DEBUG posteriormente. La salida del depurador se puede salvar en un fichero de texto. Este sistema puede ser más cómodo que usar DEBUG directamente ya que, en caso de error, no hay que volver a empezar el ensamblado completo del código.

B) Trazado de la secuencia de código generada

Utilizando nuevamente DEBUG, se pide trazar la secuencia de código generada anteriormente. Entendemos por trazar el hecho de ejecutar paso a paso una serie de instrucciones teniendo acceso en cada detención a los registros del procesador, a su estado y a la memoria del proceso.

La ejecución paso a paso nos permite depurar los programas así como alcanzar una comprensión más profunda del funcionamiento de una determinada tarea con el objetivo, por ejemplo, de conseguir un mayor rendimiento del código.

Para trazar el código con DEBUG utilizaremos el comando T. Cargaremos, primero, la secuencia de código utilizando los comandos N y L. Una vez en memoria, pasaremos a ejecutarla paso a paso observando cada vez las modificaciones que se hayan llevado a cabo en registros, estado y memoria.

```
-n codigo.bin
-l
-r ip
IP 0100
:0110
-r
AX=0000  BX=0000  CX=003D  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CB8  ES=0CB8  SS=0CB8  CS=0CB8  IP=0110   NV UP EI PL NZ NA PO NC
0CB8:0110 8CC8          MOV     AX,CS
-t
```

Hago que IP apunte a la primera instrucción de la secuencia de código.