# Bloque 1

## Programación en ensamblador de la arquitectura IA-32 bajo Win32

Prácticas de
Arquitectura de Computadores

SESIÓN 1

# Introducción a las herramientas de trabajo

## Objetivos

Esta práctica introduce al alumno en el uso de las herramientas de desarrollo en lenguaje ensamblador de la arquitectura IA-32. Durante su desarrollo, el alumno deberá obtener una versión ejecutable y que funcione correctamente de un programa sencillo.

## Conocimientos y materiales necesarios

Para el adecuado aprovechamiento de esta sesión de prácticas, el alumno necesita:

- Conocer los aspectos básicos de la arquitectura IA-32: parámetros de la arquitectura, modelo de memoria, tipos de datos, registros y formatos de las instrucciones.
- Conocer las directivas de ensamblador de uso común.
- Llevar los apuntes de la arquitectura IA-32 proporcionados en las clases de teoría

## Desarrollo de la práctica

## 1. El ciclo de desarrollo de un programa

La obtención de la versión ejecutable de un programa utilizando herramientas de desarrollo para lenguaje ensamblador requiere:

- Crear el *código fuente* en lenguaje ensamblador. Para su creación se utiliza un *editor de textos*, que para este bloque de prácticas será el editor integrado en el Visual Studio.
- Obtener el *código objeto*. Esta tarea es realizada por el compilador de ensamblador (`ml`) , que lo genera a partir del fichero fuente. Si se detectan errores en esta fase, es necesario volver al punto anterior para corregirlos.

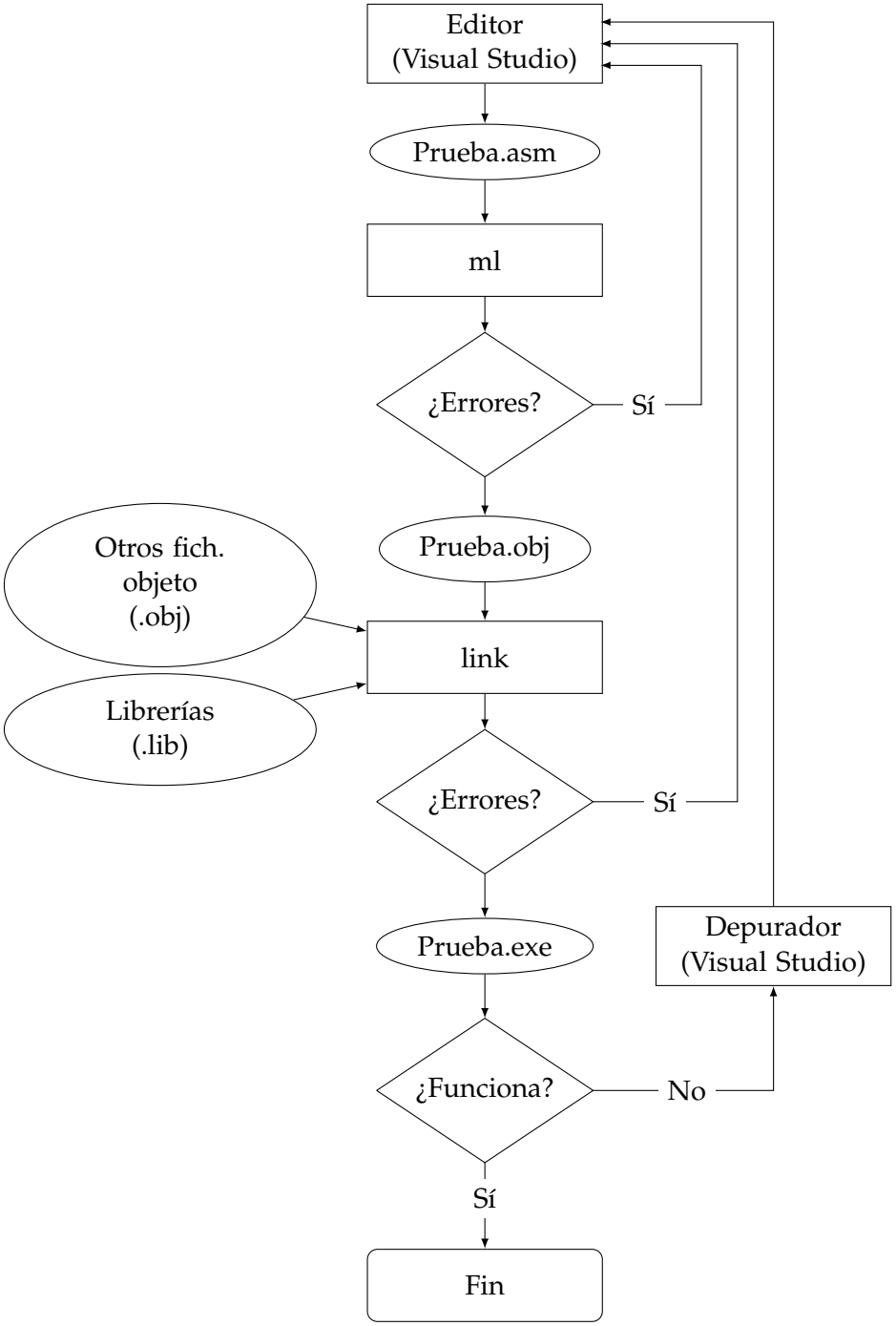


Figura 1.1: Ciclo de desarrollo de un programa en ensamblador

- Obtener el *archivo ejecutable,* enlazando el código objeto del programa con las librerías necesarias. Esto lo hace el *montador de enlaces* o *linker* (`link`). Si el linker fuese incapaz de hacerlo (porque quedan referencias sin resolver) sería necesario volver al primer punto para corregirlo.
- *Depurar* el ejecutable si no funcionase correctamente. Mediante el *depurador* se ejecuta paso a paso el programa para comprobar por qué falla. Una vez detectado el origen

del error, será necesario volver al primer punto para corregirlo.

Los pasos del ciclo de desarrollo se muestran gráficamente en la figura 1.1. Los nombres encerrados en rectángulos representan programas, y los nombres encerrados en elipses son ficheros almacenados en el disco.

## 2. Obtención de ficheros ejecutables

Para ilustrar todos estos conceptos, el alumno obtendrá a continuación la versión ejecutable de un programa que calcula la suma de tres variables, utilizando el lenguaje ensamblador de la arquitectura IA-32.

- ❏ Para que no pierdas tiempo, ya se ha creado la mayor parte del fichero fuente que utilizarás (`1-1prog1.asm`). Pregunta a tu profesor de prácticas dónde se encuentra y cópialo en tu directorio de trabajo.
- ❏ Ahora hay que arrancar una interfaz de comandos de Windows. Las herramientas del Visual Studio permiten poner en marcha una interfaz de comandos con una configuración especial. Dicha interfaz permitirá al usuario acceder al compilador (*ml*) y al linker (*link*) del Visual Studio. Para arrancar la interfaz de comandos abre el menú *inicio*, *programas*, *Microsoft Visual Studio 2005*, *Visual Studio Tools* y entonces elige la opción *Símbolo del sistema de Visual Studio 2005*.
- ❏ El listado contiene errores de sintaxis intencionados que deberás corregir. Te ayudará el compilador, ya que comprueba la sintaxis de tu código fuente y te informa de las líneas que contienen errores. Para compilar el programa teclea el comando

  ```
  ml /c /Cx /coff 1-1prog1.asm
  ```

  Los caracteres o cadenas precedidos del carácter / representan opciones de compilación necesarias que no se explican en este documento. La salida que obtendrás será:

  ```
  Microsoft (R) Macro Assembler Version 8.00.50727.42
  Copyright (C) Microsoft Corporation. All rights reserved.

   Assembling: 1-1prog1.asm
  1-1prog1.asm(15) : error A2008: syntax error :  in instruction
  1-1prog1.asm(20) : error A2008: syntax error :  in instruction
  ```

- ❏ El compilador indica que el código fuente tiene dos errores, uno en la línea 15 y otro en la 20, que debes corregir para poder continuar. (Nota: el editor del Visual Studio muestra el número de línea en el que está el cursor en una barra de estado en la parte inferior de la pantalla). Cuando creas que el código fuente no contiene errores, repite el proceso de compilación. Si de verdad no había errores, el compilador habrá generado el fichero objeto, `1-1prog1.obj`. Ejecuta en la interfaz de comandos el comando *DIR* para ver el fichero objeto que se acaba de crear.

Los ficheros `.obj` no contienen una versión totalmente terminada del código máquina de un programa. Esto es debido a que, normalmente, cuando escribimos un programa, no escribimos nosotros todo el código necesario para la correcta ejecución del programa. Así, en muchas ocasiones, dentro de nuestros programas llamamos a funciones que no están escritas por nosotros. Un ejemplo de esto lo tenemos en cualquier programa ensamblador desarrollado para el sistema operativo Windows. En estos programas siempre se llama al procedimiento `ExitProcess`, cuyo objetivo es devolver el control al sistema operativo. Sin embargo, en nuestros programas este procedimiento no está escrito, solamente lo llamamos. El código máquina de `ExitProcess` se encuentra en realidad en otro fichero denominado `kernel32.lib`, el cual pertenece a una categoría de ficheros conocidos como *librerías*. Entonces el código de nuestro programa (`1-1prog1.obj`) hay que completarlo con el código de `ExitProcess`, que es proporcionado por `kernel32.lib`. Para llevar a cabo esta operación se utiliza la herramienta *link*, a la cual se conoce como *linker* o *montador de enlaces*. Esta herramientas "enlaza" (une) el código escrito por nosotros con el código de `ExitProcess` y genera una versión completa y ejecutable de nuestro programa, que se denominará `1-1prog1.exe`

- ❏ Genera la versión ejecutable del programa tecleando el siguiente comando:
  `link /SUBSYSTEM:CONSOLE 1-1prog1.obj kernel32.lib`
  Tras su ejecución, comprueba que se ha generado el fichero `1-1prog1.exe`, ejecutando el comando *DIR*.
- ❏ Puedes ordenar al sistema operativo la ejecución de `1-1prog1.exe`; sin embargo, al ejecutarlo no notarás efecto alguno, ya que este programa (como la gran mayoría de los que harás en este bloque de prácticas) no hacen operaciones de Entrada/Salida. Comprueba esto, tecleando `1-1prog1` y pulsando ENTER a continuación.

## 3. Depuración de programas

Para ejecutar paso a paso las instrucciones de un programa y comprobar su funcionamiento se utiliza una herramienta denominada *depurador* o *debugger*. A la ejecución paso a paso de un programa se le denomina *depuración* del programa. Para depurar un programa es necesario que el código fuente haya sido compilado y enlazado de una manera especial, para que incluya dentro del fichero ejecutable los nombres (*etiquetas*) que el programador ha dado a las diferentes partes del programa (por ejemplo, `inicio`, `datos`, `bucle`, `fuera`), y no sólo el código máquina. Esto generará un archivo ejecutable que contiene información *simbólica*, es decir, nombres de etiquetas, variables, etc. pensada para ser utilizada por el depurador.

- ❏ Compila y enlaza de nuevo el programa para que incluya información de depuración (esto requerirá utilizar opciones adicionales en las llamadas al compilador y linker). En concreto ejecuta los siguientes comandos:
  ```
  ml /c /Cx /coff /Zi /Zd /Zf 1-1prog1.asm
  link /DEBUG /SUBSYSTEM:CONSOLE 1-1prog1.obj kernel32.lib
  ```
- ❏ Ejecuta el comando *DIR*. Observa que además de los ficheros 1-1prog1.obj y 1-1prog1.exe, también se ha generado el fichero 1-1prog1.pdb. Este fichero contiene una base de datos con información del programa ejecutable que será utilizada por el depurador para llevar a cabo el proceso de depuración.

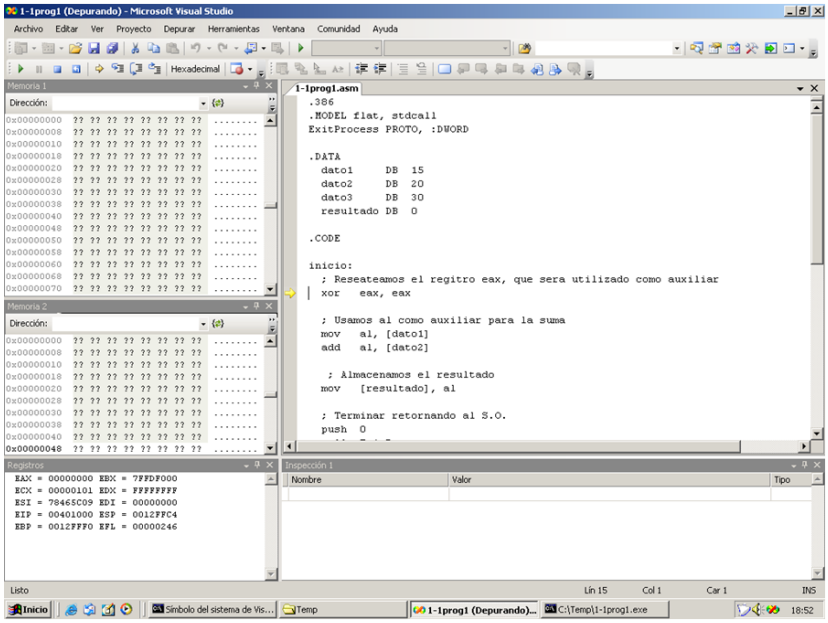


Figura 1.2: Entorno de depuración del Visual Studio

- ❏ Ahora vas a arrancar el entorno de desarrollo del Visual Studio, dentro del cuál se encuentra el depurador. Para ello, tienes que ejecutar en la línea de comandos el comando siguiente:

  `devenv /debugexe 1-1prog1.exe`

- ❏ Para comenzar la depuración, abre el menú *Depurar*, y elige la opción *Paso a paso por instrucciones*. Observa que la estructura de ventanas del entorno de desarrollo adquiere un nuevo aspecto. Esto se debe a que el entorno pasa a un nuevo estado que se llama *Depurando*, tal y como puedes observar en la barra de título de la ventana principal del entorno.

La Figura 1.2 muestra el entorno de depuración del Visual Studio. Como puedes observar en esta figura, este entorno se encuentra organizado en cinco ventanas. En la ventana superior derecha se muestra el código fuente del programa que estamos depurando. A la izquierda de esta ventana hay otras dos, denominadas *Memoria1* y *Memoria2*. Se trata de dos ventanas de exploración de memoria, que se utilizan para explorar las áreas de memoria usadas por las secciones de datos y pila del programa. En la zona inferior izquierda se encuentra la ventana de registros, que muestra el estado de los registros generales de la CPU durante la ejecución del programa, y en la zona inferior derecha hay una ventana, denominada *Inspección 1*, que se puede utilizar para inspeccionar el estado de las variables del programa.

El entorno de depuración del Visual Studio es configurable por el usuario. Así éste puede variar el número y el tipo de ventanas que aparecen en el entorno, así como su tamaño. Por favor, no realices modificaciones en el entorno, porque al abandonarlo, dichas modificaciones se registran automáticamente y, entonces, la siguiente persona que utilizase el ordenador se encontraría con el entorno modificado. Si te ocurre esto, pregúntale a tu profesor cómo restaurar la configuración del entorno de depuración según la estructura mostrada en la Figura 1.2.

A continuación se comenta cómo visualizar, mediante el entorno de depuración, las secciones de código, datos y pila del programa que se encuentra en ejecución.

- ❏ Para ver la sección de código del programa se utiliza la ventana superior derecha del entorno de depuración. Se trata de una ventana organizada mediante fichas. Cuando se comienza la depuración de un programa, se muestra una ficha con el código fuente del programa. En nuestro caso 1-1prog1.asm. Ver en el entorno el código fuente siempre es de utilidad, pero lo que queremos ver ahora es otra cosa diferente, queremos ver el código máquina correspondiente a la sección de código. Para ello debes hacer lo siguiente: abre el menú *Depurar*, *Ventanas*, *Desensamblador*. Entonces e abre una nueva ficha llamada *Desensamblador*. Selecciona esta ficha. En ella se muestra el código máquina de la sección de código, junto con las direcciones que ocupa.
- ❏ La ventana *desensamblador* cuenta con varias opciones de configuración que determinan cómo esta ventana visualiza la información. Para configurar estos parámetros, pincha con el botón derecho del ratón sobre un punto cualquiera de esta ventana. Las opciones de configuración se muestran en la parte inferior del menú que se abre. Son 6 y todas ellas empiezan por la palabra *mostrar*. De las 6 opciones debes tener seleccionadas las 4 siguientes: *Mostrar dirección*, *Mostrar bytes de código*, *Mostrar nombres de símbolos* y *Mostrar barra de herramientas*. Esta es, sin duda, la configuración más recomendable para la ventana *Desensamblador*. Asegúrate de que siempre esté así.

La ventana *Desensamblador* muestra un área de memoria de un tamaño determinado. No obstante, cuando estamos depurando un programa pequeño, como es nuestro caso, el código del programa no ocupa todo el área de memoria mostrado por la ventana. La zona de la memoria donde comienza el programa se marca con una línea roja en la que se escribe el nombre del fichero fuente precedido de la ruta en la que está ubicado.

Con la ventana *Desensamblador* configurada como se indicó anteriormente, la información queda organizada en esta ventana en tres columnas. En la columna de la izquierda se muestran las direcciones en que se encuentran almacenadas las instrucciones. Fíjate en que estas direcciones son de 32 bits y, por tanto, se representan con 8 dígitos hexadecimales. La segunda columna contiene el código máquina de las instrucciones. Observa cómo hay instrucciones de diversos tamaños. La tercera columna contiene el mnemónico de las instrucciones.

- ❏ A la vista de la información de la ventana *Desensamblador*, ¿sabrías decir cuál es la dirección de comienzo de la sección de código del programa? [1] Quédate con este valor, ya que la sección de código de todos los ejecutables que generemos para la plataforma Windows comenzarán en esta dirección. ¿Cuál es el contenido de la posición de memoria 00401006? [2]
- ❏ Ahora vamos a explorar el contenido de la sección de datos del programa. Para ello utilizaremos la ventana denominada *Memoria1*, que se ubica en la parte superior izquierda del entorno de depuración. En la parte superior de esta ventana puedes observar el campo *Dirección*. Mediante este campo podemos seleccionar la dirección del área de memoria que queremos visualizar. En nuestro caso, utilizaremos esta ventana para ver la sección de datos del programa. El montador de enlaces del sistema de desarrollo genera el ejecutable para que los datos se ubiquen a partir de la dirección 00404000 (hex), por tanto, vamos a visualizar esta zona de la memoria. Para ello tendrás que introducir el valor 00404000 (hex) en el campo *Dirección*. ¡Ojo! Para introducir valores hexadecimales en este campo debes precederlos de

1

2

'0x'. Entonces introduce el valor 0x00404000 en el campo *Dirección*. ¿Puedes identificar ahora los datos de tu programa en la ventana *Memoria1*? ¿Cuál es el contenido de la dirección de memoria 00404001? [3] ¿Coincide este valor con el segundo dato del programa? [4] ¿Por qué?

- ❏ La ventana *Memoria 2* será utilizada para explorar la sección de pila del programa. Aprenderás a utilizar esta ventana en la Sesión 5 de este bloque de prácticas.

Además de las ventanas orientadas a mostrar la memoria, en las que visualizamos la información contenida en las secciones de código, datos y pila del programa, hay otra ventana cuyo objetivo es mostrar el estado de los registros del procesador. Esta ventana recibe el nombre de *Registros*, y se muestra en el área inferior izquierda del entorno de depuración.

- ❏ Observa la ventana *registros* del procesador. En esta ventana puedes ver el estado de todos los registros de 32 bits del procesador (`eax`, `ebx`, etc.). Asimismo, esta ventana también te permite ver los bits del registro de estado. Para ello pulsa con el botón derecho sobre cualquier punto de esta ventana y selecciona la opción *Indicadores*. El Visual Studio utiliza una nomenclatura diferente que Intel para nombrar los bits del registro de estado. La equivalencia entre ambas nomenclaturas se indica a continuación: OV=OF, ZR=ZF, PL=SF y CY=CF.

Una vez descrita la información que proporcionan las diferentes ventanas del entorno de depuración, el alumno está preparado para comenzar la ejecución *paso a paso* del programa. Ten en cuenta que el programa que vas a trazar suma el contenido de las variables `dato1`, `dato2` y `dato3` de la sección de datos y almacena su suma en la variable `resultado`.

- ❏ Comenzarás ahora la traza del programa. En primer lugar, asegúrate de que tienes selecciona la ficha *Desensamblador*. Habitualmente, realizaremos la depuración desde esta ficha. Observa la flecha amarilla. Ésta apunta a la siguiente instrucción a ejecutar. Para ejecutar instrucciones paso a paso se utiliza la tecla F11. Antes de pulsar esta tecla por primera vez, contesta a la siguiente pregunta: ¿Cuál será el valor de `EIP` después de ejecutar la primera instrucción del programa? [5] ¿Por qué? Pulsa F11 y comprueba que tu respuesta es correcta. Fíjate como al ejecutar una instrucción, los registros que resultan modificados son resaltados en la pantalla.
- ❏ Continúa la ejecución paso a paso del programa pulsando F11, hasta que alcances la instrucción `mov [resultado], al` (no ejecutes esta instrucción todavía). Fíjate cómo en la ejecución paso a paso avanza la flecha amarilla que indica la siguiente instrucción a ejecutar, y como `EIP` se va incrementando para apuntar a las sucesivas instrucciones. Fíjate también en cómo se va realizando la suma de las variables sobre el registro `AL`.
- ❏ Ahora vamos a ejecutar la instrucción `mov [resultado], al`, la cual almacena el resultado de la suma en la variable `resultado`. ¿En qué dirección de memoria está situada esta variable? [6] Localiza esa dirección en la ventana *Memoria1*. Antes de ejecutar la instrucción `mov [resultado], al`, contesta: ¿Qué valor se almacenará en la variable resultado? [7] Ahora ejecuta la instrucción y comprueba que tu respuesta es correcta.

Queda ahora por ejecutar las instrucciones del programa que devuelven el control al sistema operativo Windows. Estas instrucciones son:

```
push 0
call ExitProcess
```

- ❏ Ejecuta la instrucción `push 0` pulsando F11.
- ❏ Ahora vas a ejecutar la última instrucción del programa (`call ExitProcess`), pero vas a hacerlo de una forma diferente. En lugar de utilizar la tecla F11, usarás la tecla F10. Esta tecla, cuando se aplica a una llamada a un procedimiento, hace que se ejecute el procedimiento completo. En nuestro caso, esto es muy conveniente, ya que no tenemos ningún interés en cómo funciona la función *ExitProcess*. Pulsa entonces F10. El programa termina, finalizándose también la depuración. La estructura de ventanas del entorno de desarrollo adquiere el mismo aspecto que tenía cuando se entró en el entorno. Ahora abre el menú *Archivo*, opción Salir para abandonar el entorno de desarrollo del Visual Studio. En este momento te preguntará si deseas guardar los cambios en el archivo 1-1prog1.sln. Contesta NO. Tras esto el Visual Studio se cerrará.

## 4. Facilitando la compilación y enlazado de programas

Tal y como se ha visto en las secciones anteriores, los comandos utilizados para compilar y enlazar programas, así como para lanzar el depurador resultan un tanto engorrosos de escribir, debido a la gran cantidad de opciones (/c, /Cx, /coff, /DEBUG, /debugexe etc.) que requieren. Para simplificar esta labor se pueden utilizar los denominados *archivos de comandos*, los cuales tienen siempre la extensión *.bat*. Un archivo de comandos es un fichero de texto, en el que se escriben uno o varios comandos. Entonces, ejecutar el fichero de comandos es idéntico a ejecutar consecutivamente todos los comandos que hay en él.

- ❏ En la carpeta de la asignatura se encuentran los ficheros de comandos `compila.bat`, `enlaza.bat` y `depura.bat`. Dentro del fichero `compila.bat` se indica la ejecución del comando `ml` con todas las opciones necesarias. En el fichero `enlaza.bat` se hace otro tanto con `link` y lo mismo en el caso de `depura.bat`. Copia estos fichero en tu directorio de trabajo.
- ❏ Ahora volverás a obtener el mismo programa ejecutable que generaste usando `ml` y `link`, pero utilizando ahora `compila.bat` y `enlaza.bat`. Para comprobar que haremos esto correctamente, primero borra de tu carpeta de trabajo todos los ficheros excepto 1-1prog1.asm.
- ❏ Para volver a generarlos ejecuta
  `compila 1-1prog1.asm`
- ❏ Comprueba que se ha generado 1-1prog1.obj
- ❏ Ahora ejecuta
  `enlaza 1-1prog1.obj kernel32.lib`
- ❏ Comprueba que se ha generado 1-1prog1.exe

❏ Ahora ejecuta `depura 1-1prog1.exe` Pulsa entonces F11 para comenzar la depuración del programa. Termina la depuración.

En la siguiente sesión de estas prácticas utilizarás `compila.bat` y `enlaza.bat` para obtener los ejecutables de los ejemplos que debas realizar. Asimismo utilizarás `depura.bat` para abrir el Visual Studio y depurar el programa. El uso de esta sencillas herramientas te hará más cómodo el proceso de desarrollo.

## 5. Ejercicios adicionales

✏ Haz una copia del programa 1-1prog1.asm en otro fichero que se llame 1-1prog2.asm. En este nuevo fichero realizarás la siguiente modificación: introduce un cuarto dato negativo, que sumado con los otros datos de cero. Modifica la sección de código para que se sume también este dato. Obtén una versión ejecutable de este programa. Depura el programa. Busca en la ventana *Memoria1* el dato negativo. ¿cómo está codificado? En la ventana *Registros* haz que se muestren los bits del registro de estado. Abre la ventana *Desensamblador* y ejecuta el programa hasta que sume el registro `al` con el dato negativo. Ejecuta esta instrucción y comprueba que el bit ZR (bit de cero) se pone a '1'.

SESIÓN 2

# Tipos de datos y modos de direccionamiento

## Objetivos

- Comprender la representación de los diferentes tipos de datos manejados por la arquitectura IA-32: datos tipo byte, palabra y doble palabra.
- Aprender a usar los diferentes tipos de modos de direccionamiento proporcionados por la arquitectura IA-32 para acceder a los datos de los programas.

## Conocimientos y materiales necesarios

Para la correcta realización de la práctica, el alumno deberá:

- Conocer las directivas más comunes del lenguaje ensamblador.
- Conocer los diferentes tipos de datos proporcionados por la arquitectura IA-32, así como su representación en formato "little endian".
- Conocer los modos de direccionamiento proporcionados por la arquitectura IA-32, así como su codificación.
- Asistir a clase de prácticas con los apuntes proporcionados en las clases teóricas.

## Desarrollo de la práctica

## 1. Tipos de datos

En primer lugar investigaremos cómo se codifican los diferentes tipos de datos en la arquitectura IA-32.

❏ Haz una copia del fichero `1-1prog1.asm` (desarrollado en la práctica anterior) con el nombre `1-2prog1.asm` y almacénalo en tu carpeta de trabajo.

❏ Edita el fichero `1-2prog1.asm`. Borra en él todo lo necesario hasta dejar un esqueleto básico de programa.

- ❏ Ahora vamos a definir datos de 8, 16 y 32 bits en la sección de datos del programa. Copia las siguientes directivas en la sección de datos de tu programa.

```
datos_byte  DB  1, -1
datos_word  DW  2, 256, -2
datos_doble DD  3, -3
```

- ❏ Antes de visualizar este programa con el depurador del Visual Studio, vas a intentar contestar unas preguntas relativas a la codificación de los datos. Cuando este programa se cargue en la memoria, los datos se colocarán a partir de la dirección 00404000h. En realidad, cualquier programa hecho para Windows que tenga una sección de datos pequeña (pocos bytes), como es el caso de los programas que harás en estas prácticas, tendrá sus datos cargados siempre a partir de esta dirección. Recuerda que los datos se codifican en formato "Little Endian", y que se colocan unos a continuación de otros en la memoria, según son definidos en la sección de datos. Teniendo en cuenta toda esta información, contesta a las siguientes preguntas: ¿cuántos bytes ocupan los datos definidos en nuestro programa? [1] ¿Cuál será el contenido, expresado en hexadecimal, de las siguientes posiciones de la memoria: 00404003h [2], 00404006h [3] y 00404008h [4]? Para contestar, haz un esquema de la memoria y vete colocando en dicho esquema los datos del programa.
- ❏ Ahora debes comprobar si tus respuestas son correctas. Obtén el fichero ejecutable de tu programa (utiliza `compila.bat` y `enlaza.bat` para mayor simplicidad). Ahora ábrelo con el depurador del Visual Studio (usa para ello el comando `depura`). Ubica la ventana *Memoria1* sobre los datos de tu programa. ¿Coinciden tus respuestas con la información mostrada en la ventana *Memoria1*? Si no es así y tienes alguna duda, pregúntale a tu profesor.

1

2

3

4

## 2. Accesos a memoria tipo byte, palabra y doble palabra

En esta sección vamos a practicar los diferentes tipos de accesos a memoria. Debe recordarse que la arquitectura IA-32 define que las posiciones de memoria son de tipo byte, es decir, almacenan un byte de información. Sin embargo, esto no quiere decir que cuando una instrucción accede a memoria, deba hacerlo a un solo byte de información. Las instrucciones pueden acceder a datos de 16 y de 32 bits, lo cual significa que se accede a 2 ó 4 posiciones de memoria consecutivas. Vamos a practicar este concepto.

- ❏ Copia el fichero `1-2prog1.asm` a `1-2prog2.asm`
- ❏ Edita `1-2prog2.asm`. Elimina todos los datos definidos en la sección de datos, reemplazándolos por la siguiente definición:
  `datos DB 12h, 36h, 1Ah, 0FFh, 89h, 73h, 0AAh, 50h`
- ❏ Ahora en la sección de código, vamos a realizar accesos tipo byte, palabra y doble palabra sobre estos datos. Utilizaremos el registro `ESI` para apuntar a la zona de la memoria en la que se encuentran estos datos. Pondremos este registro apuntando al dato `FF`, y luego haremos tres accesos a memoria: uno tipo byte, copiando de memoria al registro AL; otro tipo palabra, copiando

de memoria a BX; y o otro tipo doble palabra, copiando de memoria a ECX. Fíjate en cómo el tipo de acceso a memoria viene determinado por el tamaño del registro usado como operando destino. Para hacer estas operaciones, incluye las siguientes instrucciones en la sección de código de tu programa:

```
; Reseteamos los registros usados como destino
xor eax, eax
xor ebx, ebx
xor ecx, ecx

; Hacemos que ESI apunte al dato FF
mov esi, OFFSET datos
(... 1 ...)

; Realizamos los accesos a memoria
mov al, [esi]
mov bx, [esi]
mov ecx, [esi]
```

Piensa la instrucción que tienes que poner en el hueco (...1...) para que el dato apuntado sea `FF`.

- ❏ Obtén el fichero ejecutable de tu programa y depúralo con el Visual Studio. Posiciona la ventana *Memoria1* sobre los datos de tu programa. Localiza en esta ventana el dato `FF`.
- ❏ Abre la ventana *Desensamblador* y ejecuta en ella instrucciones paso a paso hasta que alcances la instrucción `mov al, [esi]` (no ejecutes esta instrucción todavía).
- ❏ Si has completado correctamente el hueco (... 1 ...), cuando ejecutes `mov al, [esi]` deberá almacenarse en el registro `AL` el valor `FF`. Ejecuta esta instrucción y comprueba que ocurre la operación esperada. Acabas de realizar un acceso a memoria de tipo byte. El dato accedido es `FF`.
- ❏ Antes de ejecutar la instrucción `mov bx, [esi]`, contesta: ¿qué valor tendrá `EBX` justo después de ejecutar esta instrucción? [5] Recuerda el formato "Little Endian". Ejecuta la instrucción y verifica tu respuesta.
- ❏ Antes de ejecutar `mov ecx, [esi]`, contesta: ¿qué valor se almacenará en `ECX`? [6]

5

6

Ahora vamos a practicar el uso de los operadores BYTE PTR, WORD PTR y DWORD PTR. Estos operadores hay que utilizarlos en aquellas instrucciones que usan un modo de direccionamiento *memoria* en el operando destino, e *inmediato* en el operando fuente, como por ejemplo:

```
mov [ebx], 0
```

En este ejemplo estarían indeterminados el tamaño del dato inmediato (8, 16 o 32 bits) y el tipo de acceso a memoria (byte, palabra o doble palabra).

Primero vamos a comprobar qué hace el compilador cuando se encuentra con una instrucción como la que acabamos de comentar:

- ❏ Copia el fichero `1-2prog2.asm` a `1-2prog3.asm`
- ❏ Edita `1-2prog3.asm`. Borra todo lo necesario hasta dejar un esqueleto básico de programa.
- ❏ En la sección de datos, vamos a reservar 24 bytes, inicializándolos con el valor -1. Usa para ello la siguiente definición:

  ```
  datos DB 24 DUP(-1)
  ```

- ❏ En la sección de código, vamos a utilizar la instrucción comentada anteriormente para poner a cero el primer byte reservado en la sección de datos. Para ello introduce en la sección de código las siguientes instrucciones:

  ```
  mov ebx, OFFSET datos
  mov [ebx], 0
  ```

- ❏ Compila el programa que acabas de hacer. Observa los mensajes que genera el compilador. Este indica que se ha producido el error "*A2070: invalid instruction operands*". Comprueba con el editor que la línea del error se corresponde con la instrucción `mov [ebx], 0`.
- ❏ Para resolver este problema tendremos que utilizar cualquiera de los operadores BYTE PTR, WORD PTR o DWORD PTR, según el tipo de acceso a memoria que queramos llevar a cabo. Usando estos operadores vamos a probar a escribir el dato 0 en las posiciones 0, 8 y 16 de la zona de datos reservada. Para ello edita `1-2prog3.asm` y borra la instrucción `mov [ebx], 0`. Ahora, justo después de la instrucción `mov ebx, OFFSET datos`, introduce las siguientes instrucciones (no es necesario que copies los comentarios):

  ```
  mov [ebx], BYTE PTR 0
  add ebx, 8 ; para apuntar a la posición 8 de la zona reservada
  mov [ebx], WORD PTR 0
  add ebx, 8 ; para apuntar a la posición 16 de la zona reservada
  mov [ebx], DWORD PTR 0
  ```

- ❏ Obtén el ejecutable y ábrelo con el depurador del Visual Studio. Posiciona la ventana *Memoria1* en el área de datos del programa. Abre la ventana *Desensamblador* y en ella ejecuta el programa paso a paso fijándote muy bien en las instrucciones que escriben el dato `0` en memoria. ¿Ves la diferencia de comportamiento entre ellas?
- ❏ Mirando la ventana *Desensamblador* contesta: ¿cuántos bytes de código máquina ocupa la instrucción `mov [ebx], BYTE PTR 0`? [7] ¿Y la `mov [ebx], DWORD PTR 0`? [8] ¿A qué se debe esta diferencia? Si no lo ves pregúntale a tu profesor.

[7]

[8]

## 3. Uso de los modos de direccionamiento

En el fichero `1-2prog4.asm`, que se encuentra en la carpeta de la asignatura, tienes un programa en el que se ilustran las formas básicas de usar los modos de direccionamiento para acceder a los datos de los programas. El fichero es totalmente autoexplicativo. En él se comentan cuatro técnicas básicas que aparecen resaltadas entre filas de asteriscos. A continuación se muestra el listado correspondiente a este fichero, con objeto de que puedas ir

siguiendo muy cómodamente su ejecución, ya que tendrás que leer detenidamente todos los comentarios.

```
1
2  .386
3  .MODEL flat, stdcall
4  ExitProcess PROTO, :DWORD
5
6  .DATA
7    var     DB  5
8    datos1  DB  1, 3, 8, 12, 15, 18
9    datos2  DW  7, 34, 89, 10, 20, 25
10
11 .CODE
12
13 inicio:
14
15   xor  eax, eax
16
17   ; ******************************
18   ; Acceso a variables (etiquetas)
19   ; ******************************
20
21   ; queremos escribir un -1 en "var"
22
23   ; Aceso a "var" al estilo del ensamblador de la CPU elemental
24   mov esi, OFFSET var
25   mov [esi], BYTE PTR -1
26   mov [esi], BYTE PTR 5  ; la dejamos como estaba
27
28   ; En IA-32 podemos hacer esto con una sola instrucción
29   mov [var], -1
30
31   ; ************************************************************
32   ; Uso de constantes numéricas en los modos de direccionamiento
33   ; ************************************************************
34
35   xor eax, eax
36
37   ; Accedemos sucesivamente a los datos situados a partir de datos1
38   ; llevándolos al registro AL
39   mov al, [datos1]
40   mov al, [datos1+1]
41   mov al, [datos1+2]
42   mov al, [datos1+3]
43
44   xor ebx, ebx
45
46   ; Accedemos sucesivamente a los datos situados a partir de datos2
47   ; llevándolos al registro BX
48   ; Ojo! Los datos son de 16 bits
49   mov bx, [datos2]
50   mov bx, [datos2+2]
51   mov bx, [datos2+4]
52   mov bx, [datos2+6]
53
54   ; ***************************
55   ; Direccionamiento de listas
56   ; ***************************
57
58   ; Vamos a procesar la lista de datos etiquetada con la etiqueta "datos2"
59   ; Sencillamente accederemos a cada uno de sus valores colocandolo en
60   ; el registro AX
```

```
61
62    xor eax, eax
63
64    ; Hay dos formas típicas de direccionar listas
65
66    ; ....... PRIMERA FORMA .......
67
68    ; La primera de ellas ya la conoces. Cargamos un registro con la
69    ; dirección de la lista y lo vamos incrementado (usaremos ESI)
70
71    mov esi, OFFSET datos2
72    mov ecx, 6              ; bucle controlado por contador
73
74  bucle:
75    mov ax, [esi]
76    ; Ojo! como los datos son de 16 bits hay que incrementar ESI en 2
77    add esi, 2
78    loop bucle
79
80    ; ....... SEGUNDA FORMA .......
81
82    ; La segunda forma es utilizar un modo de direccionamiento
83    ; [etiqueta+REGISTRO]. La etiqueta proporciona la dirección del primer
84    ; elemento de la lista. El registro se inicializa con 0 y se va
85    ; incrementando para proporcionar un desplazamiento adicional que
86    ; permita ir direccionando los sucesivos elementos de la lista
87
88    xor esi, esi            ; registro usado en el direccionamiento
89    mov ecx, 6
90
91  bucle1:
92    mov ax, [datos2+esi]
93    add esi, 2
94    loop bucle1
95
96    ; *************************
97    ; Uso del factor de escala
98    ; *************************
99
100   ; El factor de escala multiplica por una constante (2, 4 u 8 ) el
101   ; desplazamiento proporcionado por un registro en un modo de
102   ; direccionamiento
103
104   ; Sirve para direccionar listas sin preocuparse del tamaño de los
105   ; datos que contienen. El registro siempre se incrementa en 1
106
107   ; Repetimos el algoritmo anterior usando factor de escala
108   ; Se puede aplicar sobre cualquiera de las dos versiones
109
110   xor esi, esi
111   mov ecx, 6
112
113 bucle2:
114   mov ax, [datos2+esi*2]
115   ; Ojo! Incrementar ESI solo en 1
116   inc esi
117   loop bucle2
118
119
120   ; Terminar retornando al S.O.
121   push  0
122   call  ExitProcess
123
```

```
124 END  inicio
125
```

- ❏ Copia a tu carpeta de trabajo desde la carpeta de la asignatura el fichero `1-2prog4.asm`.
- ❏ Obtén el ejecutable. Ábrelo con el depurador del Visual Studio. Comienza la depuración pulsando F11 una vez. Ubica la ventana *Memoria1* en el área de datos del programa. Ahora abre la ventana *Desensamblador*. Ahora vas a ir ejecutando ordenadamente el programa, pensando sobre los modos de direccionamiento que se utilizan.
- ❏ Comenzaremos con la técnica de acceso a variable. Una variable es una posición (de uno o más bytes, dependiendo de su tamaño) de la sección de datos a la que se hace referencia mediante una etiqueta. En la CPU elemental no se podía acceder directamente a una variable, había que hacerlo siguiendo el estilo de las instrucciones situadas en las líneas 24 y 25 de tu listado. Localiza en el panel de datos la posición correspondiente a la variable `var`. Ejecuta las instrucciones en las líneas 24 y 25 y comprueba que se ha almacenado el $-1$ en la variable.
- ❏ Ejecuta la instrucción de la línea 26 para dejar la variable como estaba, es decir, con el valor 5. Comprueba en la ventana *Memoria1* que `var` toma este valor.
- ❏ En la línea 29 está la instrucción que permite acceder directamente a una variable. Fíjate que para acceder a ella usamos la etiqueta que hace referencia a la variable entre corchetes. Ejecuta esta instrucción y comprueba en la ventana *Memoria1* que se almacena el $-1$ en la variable.
- ❏ Pasamos ahora a analizar el uso de constantes numéricas en los modos de direccionamiento. Ejecuta la instrucción en la línea 35. La siguiente instrucción a ejecutar será `mov al, [datos1]`. ¿Qué valor se almacenará en `AL`? [9] Ejecuta la instrucción y compruébalo. Hemos accedido a la posición a la que hace referencia la etiqueta `datos1`. La etiqueta `datos1` ha proporcionado la dirección a la que se ha accedido. En la instrucción siguiente (línea 40) se utiliza el modo de direccionamiento a memoria `[datos1+1]`. Esto significa que a la dirección proporcionada por la etiqueta se le suma 1, y por tanto, se accede a la posición siguiente. ¿Qué valor se almacenará entonces en `AL`? [10] Ejecuta esta instrucción y compruébalo. Ejecuta las instrucciones en las líneas 41 y 42, comprobando que vas accediendo a los datos siguientes en la memoria.

  [9]

  [10]
- ❏ Ejecuta la instrucción en la línea 44. Lee los comentarios. Ejecuta las instrucciones en las líneas 49, 50 y 51. Antes de ejecutar la instrucción en la línea 52 contesta: ¿qué valor tendrá el registro `BX` tras ejecutar esta instrucción? [11] Ejecuta la instrucción y comprueba tu respuesta.

  [11]
- ❏ Pasamos ahora al direccionamiento de listas. Observa en el listado la primera forma de direccionar listas. Comprende bien su código. Contesta: ¿qué valor se habrá almacenado en el registro `AX` justo después de que la instrucción en la línea 75 se haya ejecutado por tercera vez? [12] Ejecuta hasta ese instante (usando F7 ) y comprueba tu respuesta. Termina de ejecutar el bucle.

  [12]
- ❏ Observa la segunda forma de procesar listas. Compárala con la primera forma. Localiza el primer dato de la lista `datos2` en la ventana *Memoria1*. Ahora

vete ejecutando paso a paso las instrucciones del bucle, comprobando que vas accediendo sucesivamente a los datos de la lista.

❑ Pasamos a la parte del factor de escala. Lee detenidamente los comentarios del programa. Compara las instrucciones entre las líneas 113 y 117 con las que hay entre las líneas 91 y 94. Ambas realizan lo mismo. Ejecuta el último bucle comprobando su funcionamiento.

## 4. Ejercicios adicionales

✏ Escribe un programa en el que definas en su sección de código un array de 32 elementos de tipo byte inicializados con del valor $-1$. Escribe en la sección de código de este programa un bucle que cambie el valor $-1$ de todos los elementos del array por el valor $0$. Recorre el array de forma ascendente. Obtén el ejecutable del programa y depúralo, comprobando su correcto funcionamiento.

✏ Reescribe el programa anterior recorriendo el array de forma descendente.

SESIÓN 3

# Programación del control de flujo I

## Objetivos

Utilizar las instrucciones de control de flujo ofrecidas por la arquitectura IA-32 para programar las estructuras básicas de control (bucles y condiciones) usadas en el desarrollo de algoritmos. Para ello se programará un algoritmo sencillo: *cálculo del máximo de una lista de números*.

## Conocimientos y materiales necesarios

Para la correcta realización de la práctica, el alumno deberá:

- Conocer las directivas más comunes del lenguaje ensamblador.
- Conocer las diferentes instrucciones de control de flujo de la arquitectura IA-32: saltos incondicionales, saltos condicionales (con y sin signo), y la instrucción `loop` para la programación de bucles.
- Asistir a clase de prácticas con los apuntes proporcionados en las clases teóricas.
- Antes de realizar esta práctica se debe realizar el apéndice A.

## Desarrollo de la práctica

## 1. Búsqueda del máximo en una lista de números naturales

Para ejercitarse en la programación de estructuras de control, el alumno debe escribir un programa en lenguaje ensamblador que calcule el máximo de una lista de números naturales (positivos) de un byte. La lista de números entre los que debe buscarse el máximo es: 12, 45, 78, 75, 30, 135, 3, y 101. El valor máximo obtenido se dejará en una variable declarada a tal efecto.

Para facilitar su labor, se incluye a continuación el pseudocódigo de un algoritmo que busca el máximo de una lista:

```
Inicializar máximo
Inicializar el índice de número que estamos tratando
Inicializar contador

Repetir:
    Comparar el número actual de la lista con máximo
    Si número > máximo
       Actualizar máximo
    Pasar al siguiente número de la lista
    Decrementar contador
Hasta fin de números en la lista (contador = 0)

Almacenar máximo en memoria
```

Una vez planteado el problema y puesto que el pseudocódigo del algoritmo a implementar ya está disponible, la tarea del alumno consistirá en:

- Traducir el pseudocódigo a lenguaje ensamblador y generar el código fuente.
- Obtener el fichero ejecutable (procesos de compilación y enlace).
- Depurar el programa hasta que funcione correctamente.

❏ Siguiendo los pasos explicados en el apéndice A de estas prácticas, crea en tu carpeta de prácticas un proyecto nuevo llamado `1-3prog1`. Agrega al proyecto un nuevo fichero llamado `1-3prog1.asm`. Como el programa `ProgMin.asm` contiene el esqueleto básico de un programa, lo vamos a utilizar como punto de partida. Copia su contenido a `1-3prog1.asm`.

La idea básica del algoritmo es ir recorriendo todos los números, guardando el máximo que se haya encontrado hasta el momento y actualizándolo cuando se encuentre un número mayor que el que se había encontrado hasta ese instante.

A continuación vamos a pasar el algoritmo de pseudocódigo a ensamblador. Es muy conveniente que durante la explicación que sigue revises cada poco el código del algoritmo para saber en qué parte estás.

En primer lugar debes definir los datos del programa:

❏ Crea la sección `.DATA` antes de la `.CODE`.

❏ Define la lista de números. Deben ser de tipo byte. Utiliza para marcarla la etiqueta `lista`.

❏ Define una posición de memoria para guardar el máximo. Etiquétala con `maximo`.

Antes de comenzar a escribir instrucciones, se debe decidir dónde guardar el máximo. En la especificación del problema se dice que al final debe quedar en la variable `maximo` definida antes, es decir, en una zona de memoria. Se podría utilizar esa zona de memoria para ir almacenando el máximo temporalmente, pero habría que acceder a ella en cada iteración del bucle. Acceder a memoria es una tarea costosa frente acceder a un registro y, además, no

se pueden comparar dos zonas de memoria en la misma instrucción. Por estas razones, se va a almacenar temporalmente el máximo en un registro y cuando se haya acabado el bucle se va a guardar en memoria.

Por lo tanto, hay que decidir qué registro usar para el máximo temporal. En este caso vamos a usar `AL`.

- ❏ La primera instrucción que debes escribir debe inicializar `AL`. El valor inicial de `AL` debe ser uno que sea cual sea la lista de números siempre se calcule bien el máximo, es decir, tiene que ser el número más pequeño posible para que cualquier número mayor de la lista sea el nuevo máximo temporal. ¿Cuál es el número natural más pequeño que se puede poner en `AL`?[1] Escribe la instrucción correspondiente como primera instrucción del programa.

[1]

El siguiente paso del algoritmo es inicializar un registro que vamos a utilizar como índice del número de la lista que toca tratar en cada iteración del bucle. Vamos a utilizar el registro `EDI` para este cometido.

- ❏ Escribe la instrucción para inicializar el registro `EDI` que indique que se tiene que tratar el primer número de la lista, es decir, que al sumar a la etiqueta `lista` el valor de EDI se esté apuntando al primer número de la lista.

La última de las inicializaciones que se debe hacer es la correspondiente al contador. Como se ve en el algoritmo, el bucle está controlado por un contador. Para realizar este tipo de bucles, que aparecen de manera muy habitual en todo tipo de programas, Intel tiene una instrucción especial, `LOOP`. Esta instrucción utiliza como contador el registro `ECX` y cada vez que es llamada, decrementa en uno `ECX` y, si `ECX` no ha llegado a cero, salta a la etiqueta que se le diga. Por lo tanto, vamos a utilizar como contador `ECX`. Como debemos hacer el bucle tantas veces como números tengamos en la lista, habrá que inicializarlo con la cantidad de números de la lista.

- ❏ Escribe la instrucción necesaria para inicializar `ECX` con la cantidad de números de la lista.

Las instrucciones que se van a escribir a continuación forman parte del cuerpo del bucle. Como sabes, para marcar dónde comienza el conjunto de instrucciones que deben repetirse en ensamblador es necesario poner una etiqueta.

- ❏ Escribe la etiqueta `bucle` (seguida de dos puntos).

Lo primero que se debe hacer en el bucle es comparar el número que toca tratar en esta iteración, que será el que se encuentre en la posición apuntada por `lista` más el registro índice EDI, con el máximo que hayamos encontrado hasta el momento, que se está guardando en `AL`.

- ❏ Escribe la instrucción necesaria para comparar el número actual con el máximo hasta el momento.

A continuación se debe implementar la sentencia condicional de tal manera que si el número actual es menor que el máximo hasta el momento, no habrá que hacer nada; en cambio, si el número es mayor que el máximo hasta el momento, habrá que actualizar el máximo. Por lo tanto tendremos una instrucción de actualización del máximo que habrá que saltarse si el número es menor que el máximo: necesitaremos una instrucción de salto condicional antes de la actualización del máximo.

- ❏ Escribe la instrucción de salto condicional necesaria para que, si en la comparación anterior resultó que el número era menor que el máximo, se salte a una etiqueta llamada `sigue` que escribiremos más adelante después de la instrucción de actualización del máximo [1]. ¿Cuál es el mnemónico de la instrucción que has utilizado? 2
- ❏ Escribe la instrucción de actualización del máximo, que sólo se ejecutará si en la instrucción anterior no se produce el salto.
- ❏ Escribe la etiqueta `sigue` (seguida de dos puntos) para indicar el destino del salto condicional anterior.

2

Antes de acabar el bucle debemos dejar preparado el índice para la siguiente iteración. Como al principio de cada iteración del bucle suponemos que `lista` más el índice está apuntando a un número que todavía no hemos tratado, tendremos que incrementar el índice para apuntar al siguiente número.

- ❏ Escribe la instrucción necesaria para incrementar el índice.

Si repasas de nuevo el algoritmo, verás que para acabar el bucle nos queda decrementar el contador y luego realizar un salto condicional si el contador no ha llegado a cero. Precisamente estas dos tareas son las que lleva a cabo la instrucción `LOOP`.

- ❏ Escribe la instrucción `LOOP` necesaria para que si el contador no ha llegado a cero se salte al principio del bucle. Recuerda que el principio del bucle lo habíamos marcado con la etiqueta `bucle`.

Las siguiente instrucción ya está fuera del bucle. Cuando se llegue a ella será porque ya se han recorrido todos los números y se tiene en `AL` el máximo de todos ellos. Ya sólo queda copiar ese valor a la posición de memoria correspondiente.

- ❏ Escribe la instrucción necesaria para copiar el valor de `AL` a la posición de memoria apuntada por la etiqueta `maximo`.

Ahora debes comprobar que has hecho el programa bien:

[1] Los saltos en la arquitectura Intel se pueden codificar con 8 o con 32 bits. Por defecto el compilador codifica los saltos hacia adelante en el código con 32 bits, lo cual supone un incremento inútil del tamaño del código máquina, cuando se salta a distancias cortas (menores de 128 bytes). Para hacer que el compilador compile el salto con un tamaño de 8 bits se utiliza el operador SHORT. Es decir, si hubiera que saltar con la instrucción `JE` a la etiqueta `sigue`, escribiríamos la siguiente instrucción: `je SHORT sigue`

- ❏ Compila y enlaza tu programa para tener una versión ejecutable del mismo. Recuerda que te encuentras dentro del entorno de desarrollo. Utiliza la opción de menú *Generar→Generar solución* para llevar a cabo el proceso de compilación y enlazado. Si hay errores, corrígelos hasta que consigas que el programa compile y enlace correctamente.
- ❏ Ejecuta el programa paso a paso. Comprueba que en cada iteración del bucle el registro `AL` se actualiza cuando tiene que actualizarse. Fíjate muy bien en el funcionamiento de la instrucción `loop`. ¿Qué registro se modifica durante su ejecución? [3] ¿En cuánto se modifica? Comprueba que al final del programa en la posición de memoria correspondiente a la variable máximo está el valor adecuado. ¿Cuál tiene que ser (en hexadecimal)?[4]
- ❏ Una vez comprobado que tu programa funciona bien, vamos a "importunarle" para que funcione incorrectamente. Sustituye el número $135$ de la lista por el número $-1$. Obtén el nuevo fichero ejecutable y ejecútalo paso a paso. ¿Funciona tu programa correctamente? [5] ¿Sabrías explicar cuándo y cómo se produce el fallo?
- ❏ Vuelve a cambiar el $-1$ por el $135$ para dejar el programa correcto.

3

4

5

## 2. Búsqueda del máximo en una lista de números enteros

Vamos a hacer una nueva versión del programa que busque el máximo en una lista de números positivos y negativos, es decir, de números enteros.

- ❏ Crea un nuevo proyecto llamado `1-3prog2`. Agrega al proyecto un nuevo fichero llamado `1-3prog2.asm` y copia en él el contenido del fichero `1-3prog1.asm`.
- ❏ Cambia el $135$ por $-1$.
- ❏ Debes inicializar el registro utilizado para almacenar el máximo parcial con un nuevo valor. Este valor debe ser el mínimo de los que podamos encontrar en el rango de los procesables. ¿Qué valor es éste? [6] Si tienes duda pregúntale a tu profesor. Cambia la instrucción de inicialización de `AL` para que ponga este valor.
- ❏ Debes cambiar la instrucción de salto condicional, para que tenga en cuenta que está interpretando datos con signo. ¿Qué instrucción has elegido? [7]
- ❏ Obtén el fichero ejecutable. Ejecuta el programa paso a paso y determina si su funcionamiento es correcto. ¿Qué máximo ha calculado tu programa? Responde en hexadecimal. [8]

6

7

8

## 3. Ejercicios adicionales

✏ Haz una nueva copia del último programa que has desarrollado y modifica lo que sea necesario para que el programa procese la siguiente lista de números: -120, -128, -119, -121, -122, -118, -125, -126, -117, y -124. Obtén el ejecutable del programa, ejecútalo paso a paso y comprueba su correcto funcionamiento.

✏ Modificar el programa para que además de calcular el máximo calcule también el mínimo de la lista de números, interpretando los números con signo. Comprueba con diversas listas de números que tu programa funciona correctamente.

SESIÓN 4

# Programación del control de flujo II

## Objetivos

- Utilizar las instrucciones de control de flujo ofrecidas por la arquitectura IA-32 para programar las estructuras básicas de control (bucles y condiciones) usadas en el desarrollo de algoritmos.
- Realizar algoritmos simples que procesen cadenas de caracteres.

## Conocimientos y materiales necesarios

Para la correcta realización de la práctica, el alumno deberá:

- Conocer las directivas más comunes del lenguaje ensamblador.
- Conocer las diferentes instrucciones de control de flujo de la arquitectura IA-32: saltos incondicionales y saltos condicionales (con y sin signo)
- Conocer el uso de instrucciones lógicas para manejo de bits.
- Asistir a clase de prácticas con los apuntes proporcionados en las clases teóricas.

## Desarrollo de la práctica

En esta sesión, además de trabajar con las instrucciones de control de flujo, practicaremos también el procesamiento de cadenas de caracteres. Para ello se pedirá al alumno que realice un algoritmo que transforme una cadena de caracteres de minúsculas a mayúsculas. La cadena de caracteres a transformar estará definida en la sección de datos del programa, y estará integrada exclusivamente por letras minúsculas. La cadena generada como resultado del procesamiento se almacenará en otra zona de la sección de datos, reservada para tal efecto.

## 1. Transformación de minúsculas en mayúsculas

Como ya debes saber, cuando definimos una constante de tipo carácter en un programa, el compilador sustituye automáticamente esa constante por su código ASCII. Los códigos ASCII cumplen una propiedad que resultará fundamental para desarrollar el algoritmo que

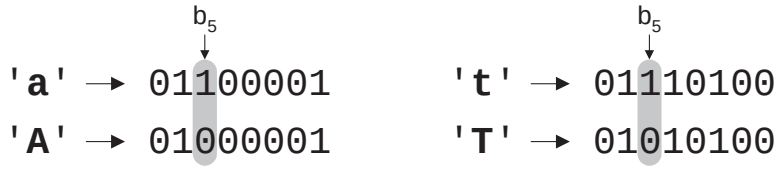


Figura 4.1: Diferencia de los códigos ASCII de mayúsculas y minúsculas en el bit de peso 5

convierte minúsculas a mayúsculas. Y es que los códigos correspondientes a minúsculas y mayúsculas sólo se diferencian en un bit, el de peso 5. A modo de ejemplo, esto puede observarse comparando los códigos ASCII mostrados en la Figura 4.1.

Gracias a la propiedad que se acaba de comentar, transformar una minúscula en mayúscula es poner a '0' el bit de peso 5 de la minúscula, dejando como están el resto de sus bits. Para conseguir esto, se realiza una operación lógica del código ASCII a transformar con una máscara (constante de 8 bits), que debe calcularse para forzar a '0' el bit de peso 5 de la letra que estamos transformando. ¿Qué operación lógica se debe realizar? [1] ¿Cuál es el valor de la máscara requerida expresado en hexadecimal? [2] Pregúntale a tu profesor de prácticas si tienes dudas.

1

2

## 2. Desarrollo del programa

Ahora iremos desarrollando por pasos este programa. Para ello realiza las siguientes operaciones:

- ❏ Siguiendo los pasos especificados en el Apéndice A de estas prácticas, crea en tu carpeta de prácticas un nuevo proyecto llamado `1-4prog1`. Agrega al proyecto un nuevo fichero llamado `1-4prog1.asm`. Abre alguno de los programas que hayas realizado anteriormente y cópialo en `1-4prog1.asm`. Entonces borra en este fichero todo lo necesario hasta dejar un esqueleto básico de programa.
- ❏ Ahora vamos a definir la sección de datos del programa. En esta sección habrá que definir la cadena de caracteres a procesar, y habrá que reservar un área de memoria para almacenar la cadena obtenida como resultado del procesamiento. Para definir cadenas de caracteres se utiliza la directiva `DB`, ya que cada carácter ocupa un byte de memoria. Los caracteres de la cadena se introducen entre comillas dobles (aunque el ensamblador también admite comillas simples). También suele ser práctica habitual terminar las cadenas de caracteres con un byte *terminador*, que indique el final de la cadena. Este byte suele ser el número 0. Supongamos que la cadena a procesar sea la palabra "neumann", en honor al inventor de los computadores. Para definir esta cadena en la sección de datos debes escribir lo siguiente:

  ```
  cadena_entrada DB "neumann", 0
  ```

  Ahora vamos a reservar un área en la sección de datos del programa para almacenar la cadena de resultado. Para ello utilizaremos la directiva `DUP`. El número de bytes a reservar debe ser ocho, siete para los caracteres de la cadena, más uno para el terminador. Lo normal, es que cuando se utiliza `DUP`, se inicialice la memoria reservada con el valor 0. No obstante, para que

observes mejor el funcionamiento de esta directiva durante la depuración del programa, inicializaremos los bytes reservados con el valor 42, que es el código ASCII del carácter '*'. Para llevar esto a cabo debes escribir en la sección de datos de tu programa lo siguiente:

```
cadena_salida DB 8 DUP(42)
```

El 8 indica que se reservan ocho bytes, y el 42 entre paréntesis es el valor con el que se inicializan los ocho bytes reservados.

- Una vez definida la sección de datos del programa, hay que escribir la sección de código. Para ello sólo se os va a dar una indicación acerca de cómo construir el bucle. El resto quedará del lado de vuestra creatividad. Para procesar la cadena utilizaremos un bucle, que calculará y almacenará una mayúscula en cada iteración. Sin embargo, vamos a construir el bucle de una forma diferente a la utilizada en el algoritmo del máximo, en la que se construyó un *bucle controlado por contador*. En esta ocasión, construiremos un *bucle controlado por condición*, es decir, el bucle se estará ejecutando mientras que se cumpla una determinada condición, que en este caso será que no hayamos alcanzado el carácter terminador de cadena (0). A continuación se muestra con pseudoinstrucciones la forma de construir este bucle:

```
bucle:
  Comparar carácter actual con el terminador
  saltar fuera del bucle si es el terminador


  ; Cuerpo del bucle


  jmp bucle  ; hay que volver siempre
fuera:
```

  Teniendo en cuenta estas indicaciones, completa toda la sección de código de este programa.

- Obtén la versión ejecutable de este programa.
- Empieza la depuración del programa. Entonces comenzaremos la ejecución paso a paso.
- En primer lugar, tienes que posicionar la ventana *Memoria1* en la zona de la memoria en la que se encuentran los datos del programa. Recuerda que los datos de nuestros programas se ubican a partir de la dirección 00404000h. En la Figura 4.2 se muestra cómo deben quedar ubicados los datos de nuestro programa. Fíjate cómo en la primera fila de la ventana *Memoria1* se observan los códigos ASCII de los caracteres de la cadena. En las ventanas de memoria se proporciona una columna en su parte derecha que muestra la interpretación ASCII de lo que hay en la memoria. Así en nuestro caso, como lo que hay en la memoria son los códigos correspondientes a la cadena "neumann", se visualiza esta cadena en la parte derecha de la ventana *Memoria1*. En la segunda fila puedes observar la zona reservada para la cadena de salida, que ha sido inicializada con el dato 42 (2Ah). A la derecha se muestra el carácter asociado a este valor, que es el '*'.

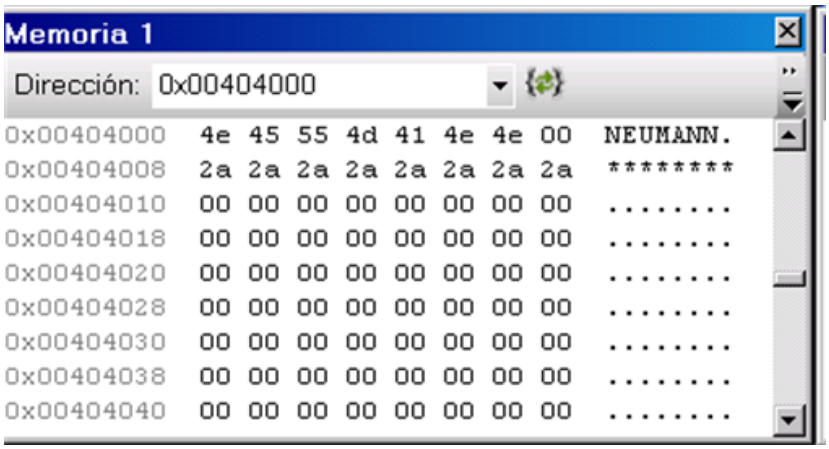


Figura 4.2: Estado de la sección de datos al comienzo del programa

❏ Utilizando la ventana *Desensamblador*, comienza la depuración paso a paso del programa. Acuérdate de trazar con F11, salvo cuando ejecutes la última instrucción `call ExitProcess`, que lo harás con F10. Durante la ejecución paso a paso, no pierdas de vista la ventana *Memoria1*. En cada iteración debes observar cómo se copia un carácter de la cadena, convertido a mayúscula, en la zona reservada para tal efecto. Si tu programa no funciona de esta forma, es posible que tengas algún error. Si no lo encuentras, pregúntale a tu profesor.

## 3. Ejercicios adicionales

✏ El algoritmo que hemos desarrollado es bastante incompleto, ya que, en principio, no tiene previsto que en la cadena a procesar haya caracteres que no sean letras minúsculas. Ahora vas a comprobar cómo se comporta el algoritmo cuando se dé este caso. Para ello, sustituye la cadena 'neumann' del programa que has hecho durante la práctica, por la cadena 'num = 3'. Como puedes observar ambas cadenas tienen el mismo número de caracteres. Obtén el ejecutable y ejecútalo paso a paso. ¿Qué ocurre con los caracteres que no son letras minúsculas?

✏ Se trata ahora de que realices una nueva versión del algoritmo, que compruebe el tipo de carácter que esté procesando. Entonces, si el carácter corresponde a una minúscula, que lo convierta a mayúscula, y si no, que no haga nada con él. Obtén el ejecutable y ejecútalo paso a paso, comprobando su correcto funcionamiento.

✏ Se pueden plantear infinidad de algoritmos que realicen diversos tipos de operaciones sobre cadenas de caracteres. Si has llegado hasta aquí, te proponemos uno más complejo que los que has hecho hasta ahora. Intenta hacer una algoritmo que cuente el número de palabras que hay en una cadena de caracteres. Supón que las palabras de la cadena están separadas por espacios en blanco, pero entre dos palabras puede haber uno o varios espacios. Define cadenas de prueba y procésalas con tu algoritmo.

SESIÓN 5

# Uso de la pila y llamadas a procedimientos

## Objetivos

Esta sesión pretende clarificar el funcionamiento de la pila en la arquitectura IA-32. Para ello se verá su uso como almacén de datos temporal. Asimismo en la parte final de la sesión se explicará el uso de la pila a través del mecanismo de llamada a procedimientos y retorno del mismo.

## Conocimientos y materiales necesarios

Para el buen desarrollo y aprovechamiento de la práctica sería necesario:

- Conocer el funcionamiento teórico de la estructura de pila. Debe repasarse los apuntes de clase sobre el tema.
- Conocer el juego de instrucciones resumido del lenguaje ensamblador para la arquitectura IA-32.

Se da por supuesto que el alumno ya está familiarizado con el entorno de desarrollo *Visual Studio*. Además se supone que se conoce la forma de depurar los programas, siendo capaz de visualizar el estado de los registros y de la memoria mientras se ejecutan las instrucciones.

## Desarrollo de la práctica

La pila es una zona de memoria que se utiliza en los programas para almacenar datos de forma temporal. Debido a que es una zona de memoria, la visualización de los datos que contiene la pila durante la depuración de los programas se realizará mediante una ventana de memoria. En concreto se utilizará la ventana *Memoria 2* para explicar la información contenida en la pila en cada momento.

La pila, como zona de almacenamiento de datos, sigue el criterio *"little endian"*, lo cual es imprescindible saber para poder interpretar su contenido.

# 1. Funcionamiento de la pila

A continuación se verá el uso de la pila como almacén de información. Se verá cómo se introducen datos en la pila y cómo se recuperan, así como las consideraciones especiales que hay que tener en función del tipo de datos a introducir y el orden de las operaciones de introducción/recuperación.

## 1.1. Almacenando operandos en la pila

❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-5prog1` y agrega un nuevo fichero que se llame `1-5prog1.asm`. En el fichero fuente se escribirá el siguiente fragmento de código:

```
1  .386
2  .MODEL FLAT, stdcall
3  ExitProcess PROTO, :DWORD
4
5  .DATA
6  valor1  DD 12345678h
7
8  .CODE
9  inicio:
10   mov eax, 14131211h
11   push eax         ; Apilar registro
12   push [valor1]    ; Apilar posición de memoria
13
14   push 4           ; Apilar dato inmediato
15
16   ; Retorno al Sistema Operativo
17   push 0
18   call ExitProcess
19 END inicio
```

❏ A continuación, compila el programa [CTRL-F7]. Si hay algún error arréglalo antes de continuar. Una vez que el programa haya compilado con éxito se debe realizar el proceso de enlazado [F7]. Por último se debe comenzar el proceso de depuración [F11]. Si se pulsa la tecla [F11] se realiza el proceso de compilación y enlazado antes del comienzo de la depuración, con lo que se puede usar como método rápido de comienzo de la depuración.

Antes de ejecutar ninguna instrucción, vas a configurar la ventana *Memoria2* para que puedas observar en ella la evolución de la pila de la forma más clara posible. Se entiende que en este instante ya te encuentras en el estado *Depurando*, es decir, ya has pulsado [F11] una vez y la flecha amarilla apunta a la instrucción `mov eax, 14131211h`.

❏ Observa la ventana *Memoria 2*. Esta ventana está configurada con un número de columnas *Automático*. Esto significa que el número de bytes de memoria que se muestran en cada fila de la ventana se ajusta al ancho de ésta. Sin embargo, como los datos que vamos a introducir y sacar de la pila van a ser siempre de 32 bits, nos interesa configurar la ventana *Memoria2* para que en cada una de sus filas muestre 4 bytes. Así en cada fila de esta ventana observaremos un dato de la pila. Para configurar el número de columnas, en la barra de herramientas de la ventana *Memoria 2* (es decir, en donde se encuentra el campo *Dirección*), pulsa sobre el botón situado en el

extremo derecho. Aparecerá entonces el campo *Columnas*. Selecciona para este campo el valor 4. La ventana *Memoria 2* cambia entonces su modo de visualización a 4 columnas.

- ❏ Ahora tienes que posicionar la ventana en la zona de memoria ocupada por la pila. Sabes que la cabecera de la pila se encuentra en la dirección apuntada por el registro ESP. ¿Qué valor tiene este registro? [1] Introdúcelo en el campo *Dirección* de la ventana. Entonces la venta te quedará posicionada en la zona de pila. Sin embargo, todavía hay que hacer algo más. Con la configuración que tienes ahora en la venta *Memoria 2*, tienes la cabecera de la pila en la parte superior de la ventana. Esto no es apropiado, ya que la pila crece hacia posiciones de memoria decrecientes, por lo que si ahora introduces un dato en la pila no lo verías en la ventana. Entonces, utilizando la barra de *scroll* de la ventana, debes bajar la cabecera de la pila a la parte inferior de la ventana, es decir, que la dirección que tienes en el registro ESP quede en el límite inferior de la ventana *Memoria2*. No obstante, si tienes dudas en esto, pregúntale a tu profesor de prácticas.

1

Ahora vamos a comenzar la ejecución paso a paso del programa.

- ❏ Pulsa F11, ¿qué valor se carga en el registro `EAX`?[2]
- ❏ Antes de ejecutar la nueva instrucción, ¿cuál crees que será el nuevo valor del registro `ESP`?[3] ¿Cuántos bytes de memoria serán necesarios para guardar el dato?[4] Pulsa F11 y comprueba tus respuestas. Observa la ventana *Memoria 2*. ¿Dónde se ha introducido el valor en la pila? Ese valor es el que tenía `EAX`, que ahora ha sido introducido en la pila.
- ❏ Ejecuta la siguiente instrucción. ¿Coincide el número de bytes reservados en la pila con el tipo de dato en memoria?[5]
- ❏ ¿Cuántos bytes crees que se reservarán en la pila para guardar el dato inmediato `4` con la instrucción `push 4`?[6] Ejecuta la instrucción y comprueba tu respuesta. A pesar de ser suficiente con un byte, todos los datos inmediatos se guardan en la pila con un tamaño de 32 bits.
- ❏ Abandona la depuración de este programa y cierra el Visual Studio.

2

3

4

5

6

Mediante este ejercicio hemos visto cómo se almacenan valores de 32 bits (4 bytes o tipo doble palabra) en la pila. Esto suele ser lo más habitual. No obstante debe indicarse que también es posible almacenar datos de 16 bits en la pila, aunque esta situación es menos común. Por ello, en los programas desarrollados en esta asignatura no se utilizará la pila para almacenar datos de dicho tamaño.

## 1.2. Orden de almacenamiento/recuperación

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-5prog2` y agrega un nuevo fichero que se llame `1-5prog2.asm`. En el fichero fuente se escribirá el siguiente fragmento de código:

```
1 .386
2 .MODEL FLAT, stdcall
3 ExitProcess PROTO, :DWORD
4
```

```
5  .DATA
6  valor1  DD 12345678h
7
8  .CODE
9  inicio:
10   mov eax, 14131211h
11   mov ecx, [valor1]
12   mov edx, 0ABCDEFh
13
14   push eax
15   push ecx
16   push edx
17
18   pop ecx
19   pop eax
20   pop edx
21
22   ; Retorno al Sistema Operativo
23   push 0
24   call ExitProcess
25 END inicio
```

- ❏ A continuación activa la ejecución paso a paso. Si ha habido algún error de compilación se debe solucionar antes de continuar.
- ❏ Antes de ejecutar ninguna instrucción trata de completar la siguiente tabla (los valores que hay que poner en las columnas de la derecha de la tabla son los que se obtienen tras la ejecución de las instrucciones):

| Instrucción | Valor del Registro | Valor de ESP | Dato en la cabecera de la pila |
|---|---|---|---|
| — | — | | |
| `push eax` | | | |
| `push ecx` | | | |
| `push edx` | | | |
| `pop ecx` | | | |
| `pop eax` | | | |
| `pop edx` | | | |

- ❏ Configura la ventana *Memoria 2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior, según se vio en el ejercicio anterior.
- ❏ Ejecuta las instrucciones necesarias para comprobar las respuestas que has obtenido para la tabla anterior.
- ❏ ¿Son los valores iniciales y finales de los registros los mismos?[7]
- ❏ A la vista de lo anterior, podrías explicar cómo funcionan las instrucciones `push` y `pop`.
- ❏ ¿Cuál debería ser el orden correcto de las instrucciones para que los registros recuperaran sus valores iniciales?[8]
- ❏ Abandona la depuración de este programa y cierra el Visual Studio.

[7]

[8]

## 1.3. Suma de un valor al registro ESP

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-5prog3` y agrega un nuevo fichero que se llame `1-5prog3.asm`. Copia en este fichero el ejercicio anterior, es decir, `1-5prog2.asm`.

- En `1-5prog3.asm` sustituye las tres instrucciones `pop` por las siguientes instrucciones:

```
1  ...
2  add  esp, 12
3
4  ; Se introduce un nuevo dato en la pila
5  push 44h
6  ...
```

- A continuación activa la ejecución paso a paso. Si ha habido algún error de compilación, debes solucionarlo antes de continuar.
- Antes de ejecutar ninguna instrucción, anota el valor del registro `ESP`[9].

9

- Configura la ventana *Memoria 2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior, según se vio en los ejercicios anteriores.
- Ejecuta las tres instrucciones `mov` que cargarán los registros con sus valores.
- A medida que ejecutes las instrucciones `push` completa la siguiente tabla:

| Instrucción | Valor de `ESP` | Dato en la cabecera de la pila |
|---|---|---|
| `push eax` | | |
| `push ecx` | | |
| `push edx` | | |

¿Cuál es el valor más bajo que alcanza el puntero de pila (registro `ESP`)?[10]

10

- Ejecuta la instrucción `add esp, 12`. ¿Cuál es ahora el valor del puntero de pila?[11]

11

Observa que el puntero de pila tiene ahora el mismo valor que al comienzo del programa. Los datos que se han apilado previamente permanecen en las posiciones de memoria en las que se ubicaron durante la apilación. Sin embargo, al incrementar el puntero de pila estos valores han quedado fuera de la pila.

- Ahora veremos cómo al hacer una nueva apilación se 'machacan' los valores anteriores. ¿Cuál es el primer valor que se introdujo en la pila durante la ejecución de este programa? [12]. Localiza este valor en la ventana *Memoria 2*. Ejecuta la instrucción `push 44h` y observa cómo se 'machaca' dicho valor.

12

- Abandona la depuración de este programa y cierra el Visual Studio.

Como has podido observar, sumar un valor al puntero de pila equivale a eliminar datos de la pila (los datos no se recuperan en ningún registro). Esta forma de actuar se usará en los procedimientos para eliminar los parámetros que se le pasan al procedimiento.

## 2. Llamadas a procedimientos

En esta sección se estudiará el uso de la pila como almacén de información temporal durante la ejecución de las instrucciones de llamada y retorno de procedimiento.

Cuando se realiza la llamada a un procedimiento, una vez ejecutado el código del procedimiento el programa debe continuar en la siguiente instrucción a la de llamada del procedimiento. Para conseguir esto, antes de llamar al procedimiento se guarda la dirección

donde se encuentra la siguiente instrucción del programa principal, después de la de llamada al procedimiento. A esta dirección se le denomina *dirección de retorno* y el lugar donde se guarda temporalmente (mientras se ejecuta el procedimiento) es la pila.

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-5prog4` y agrega un nuevo fichero que se llame `1-5prog4.asm`. En el fichero fuente se escribirá el siguiente fragmento de código:

```
1  .386
2  .MODEL FLAT, stdcall
3  ExitProcess PROTO, :DWORD
4
5  .DATA
6  valor1  DD 12345678h
7
8  .CODE
9  inicio:
10   mov eax, 14131211h
11   mov ecx, [valor1]
12
13   call minimo  ; llamada al procedimiento
14
15   mov edx, 0ABCDEFh
16
17   ; Retorno al S. O.
18   push 0
19   call ExitProcess
20
21 ; Codigo del procedimiento
22
23 minimo  PROC
24  ret         ; Instruccion de retorno
25 minimo  ENDP
26
27 END inicio
```

Este listado representa la llamada al procedimiento más sencillo que podemos tener, un procedimiento que no hace nada, se llama y retorna. Comienza la ejecución paso a paso.

Vamos a ver seguidamente la relación entre los procedimientos y la pila.

- ❏ Obtén el ejecutable del programa y comienza la depuración.
- ❏ Configura la ventana *Memoria 2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior, según se vio en los ejercicios anteriores.
- ❏ Abre la ventana *Desensamblador*. Debe estar configurada para que se vean las direcciones de memoria que ocupan las instrucciones y conviene que no se muestre el código fuente.
- ❏ Usando la ventana *Desensamblador*, ejecuta las dos primeras instrucciones del programa.
- ❏ Antes de continuar, observa lo siguiente: ¿qué valor tiene el registro `EIP`?[13] Busca esta dirección en la ventana *Desensamblador*. Como puedes ver corresponde a la dirección de la instrucción `call minimo`. ¿Cuál es el valor del registro `ESP`?[14]

13

14

- ❏ Intenta responder ahora a las siguientes preguntas: ¿qué valor tomará `EIP` después de ejecutar la instrucción `call minimo`? [15] ¿Y el registro `ESP`?[16]
- ❏ Ejecuta la instrucción pulsando F11 y comprueba tus respuestas.
- ❏ Busca en la ventana *Desensamblador* la dirección contenida en `EIP`, ¿a qué instrucción corresponde?[17] Efectivamente es la primera instrucción del procedimiento. El puntero de pila, `ESP`, se ha decrementado en 4 y en la pila se ha introducido una dirección, ¿cuál?[18] Busca esta dirección en la ventana *Desensamblador*. ¿A qué instrucción corresponde?[19] Efectivamente es la dirección de la instrucción que sigue a `call` en el programa principal.

  Hemos visto cómo se guarda temporalmente en la pila la dirección donde debe continuar la ejecución del programa una vez que haya concluido el procedimiento.
- ❏ Pulsa de nuevo F11 y se ejecutará la instrucción `ret`, observa la evolución de `EIP` y `ESP`. Se recupera la dirección de retorno de la pila y se carga de nuevo en el registro `EIP`, de esta forma el programa puede continuar ejecutando instrucciones.

15

16

17

18

19

## 3. Ejercicios adicionales

- ✏ En el programa `1-5prog1` no se ha vaciado la pila antes de salir del programa. ¿Qué instrucción se podría añadir después del último `push` para que el valor del puntero de pila, `ESP`, fuera el mismo al inicio del programa que después de ejecutar la instrucción propuesta? [20]. Modifica el programa y comprueba el correcto funcionamiento.
- ✏ Construye un pequeño programa que se aproveche del funcionamiento de la pila para intercambiar el valor de dos registros.

20

SESIÓN 6

# Procedimientos I

## Objetivos

El objetivo de esta práctica es:

- Comprender el funcionamiento de las instrucciones relacionadas con el uso de procedimientos (`call`, `ret`, `ret N`).
- Entender cómo se realiza el paso de parámetros a los procedimientos y practicar con ello.
- Aplicar todo lo anterior para construir programas utilizando procedimientos y comprobar la gran funcionalidad que estos aportan.

## Conocimientos y materiales necesarios

Para el buen desarrollo y aprovechamiento de la práctica sería necesario:

- Conocer el juego de instrucciones resumido del lenguaje ensamblador para la arquitectura IA-32.
- Haber comprendido el funcionamiento de la estructura de pila desarrollada en la sesión previa.
- Conocer a nivel teórico el concepto de procedimiento, el proceso de llamada y el mecanismo de paso de parámetros.

## Desarrollo de la práctica

A continuación se muestra el listado de un programa que contiene un programa principal y un procedimiento llamado `suma`.

```
1
2 .386
3 .MODEL flat, stdcall
4 ExitProcess PROTO, :DWORD
5
6 .DATA
7 datos       DD  1, 7, 45, 2, 43, -1
8 resultado   DD  0
9
```

```
10 .CODE
11
12 inicio:
13   ; Paso de parámetros (a través de la pila) al procedimiento 'suma'
14   ; Se escribe en la pila la dirección de la lista
15   ; (Utilizar sólo una instrucción)
16   ......
17
18   ; Se escribe en la pila el número de elementos de la lista 'datos'
19   ; (Utilizar sólo una instrucción)
20   ......
21
22   ; Se llama al procedimiento
23   ......
24
25   ; Se destruyen los parámetros
26   ......
27
28   ; Se almacena el resultado en la variable resultado
29   mov   [resultado], eax
30
31   ; Terminar retornando al S.O.
32   push  0
33   call  ExitProcess
34
35 suma  PROC
36   ; Instrucciones estándar de entrada en el procedimiento
37   push  ebp
38   mov   ebp, esp
39   ; Se salvan todos los registros a utilizar
40   ; (excepto EAX, donde se devuelve el resultado)
41   push  ecx
42   push  esi
43
44   ; El registro ESI se utiliza para apuntar a los elementos de 'datos'
45   ; Se carga el registro ESI con el parámetro adecuado
46   ......
47
48   ; El registro ECX se utiliza como contador del bucle
49   ; Se carga ECX con el parámetro adecuado
50   ......
51
52   ; El registro EAX se utiliza como acumulador de la suma
53   ; Se resetea el registro EAX
54   xor   eax, eax
55
56   ; Bucle de procesamiento
57 bucle:
58   add   eax, [esi]
59   ; Hacer que ESI apunte al siguiente elemento
60   ......
61   loop  bucle
62
63   ; restauramos los registros
64   pop   esi
65   pop   ecx
66   ......
67   ret
68 suma  ENDP
69
70 END  inicio
71
```

El objetivo del procedimiento `suma` es realizar la suma acumulada de una lista de números. Este procedimiento recibe dos parámetros a través de la pila. El primer parámetro (en orden de apilación) es la dirección de la lista que va a procesar y el segundo parámetro, el número de elementos de la lista. El procedimiento devuelve el resultado de la suma acumulada en el registro EAX.

El único objetivo del programa principal es llamar al procedimiento `suma`, pasándole los parámetros en la forma apropiada y recoger el valor devuelto por el procedimiento, almacenándolo en la variable `resultado`.

## 1. Paso de parámetros en la pila

En la primera parte de esta sesión vamos a practicar el paso de parámetros a través de la pila. Para ello utilizaremos el programa anterior, en el que tendrás que completar las instrucciones que faltan. Éstas están marcadas con una secuencia de puntos.

En la carpeta de prácticas de la asignatura encontrarás el fichero `1-6prog1(incompleto).asm`. Dicho fichero contiene el listado del programa.

- ❏ Copia el fichero `1-6prog1(incompleto).asm` en tu carpeta de prácticas.
- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-6prog1` y agrega a él un nuevo fichero que se llame `1-6prog1.asm`.
- ❏ Copia el contenido del fichero `1-6prog1(incompleto).asm` en el fichero `1-6prog1.asm`.
- ❏ Ahora tienes que completar todas las instrucciones que faltan en el programa. Justo antes de cada una de las instrucciones que faltan (que se indica mediante la secuencia .....) hay un comentario que indica el cometido de la instrucción. Lee el comentario detenidamente y luego sustituye los puntos por la instrucción necesaria.
- ❏ Una vez que hayas completado todas las instrucciones del programa, compílalo [CTRL-F7] y enlázalo [F7] corrigiendo los posibles errores que aparezcan en él hasta que obtengas el ejecutable.
- ❏ Ahora tienes que comprobar que tu programa funciona correctamente. Para ello puedes ejecutar usando [F11] las dos instrucciones que colocan los parámetros en la pila. Entonces ejecuta mediante [F10] la instrucción que llama al procedimiento `suma`. Esto hará que se ejecute todo el procedimiento y se retorne a la instrucción siguiente a la de llamada al procedimiento. Como el procedimiento retorna en el registro EAX el valor de la suma acumulada de la lista `datos`, en este momento de la ejecución, EAX debe contener el valor $00000061h$, es decir, $97$ (decimal). Si es así, es razonable pensar que tu programa es correcto. En el caso contrario, deberás depurarlo paso a paso hasta que encuentres el problema.

Una vez que dispongas de un diseño correcto del programa vas a hacer un trazado del mismo, para ver cómo evoluciona el puntero de instrucción y la pila durante la ejecución del procedimiento.

- ❏ Comienza la depuración del programa `1-6prog1`. Pulsa para ello F11. En este momento la flecha amarilla apunta a la instrucción que apila el primer parámetro, es decir, la dirección de `datos`. No ejecutes todavía esta instrucción, vamos a preparar las ventanas *Memoria 1* y *Memoria 2*.
- ❏ Haz que la ventana *Memoria 1* apunte a los datos del programa. Antes de observar el contenido de la memoria, contesta a la siguiente pregunta: ¿en qué dirección se ubica el dado $-1$ de la lista `datos`? [1]. Contrasta tu respuesta buscando este dato en la ventana *Memoria 1*.
- ❏ Configura la ventana *Memoria 2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior, según se vio en la sesión anterior.
- ❏ Abre la ventana *Desensamblador*. En este caso, vas a realizar el proceso de depuración utilizando esta ventana, ya que nos permite ver las direcciones en donde su ubican las instrucciones del programa.
- ❏ Anota el valor que tiene el registro `ESP` en este momento, es decir, al comienzo de la ejecución. [2]

  En la medida que vamos a ir ejecutando las instrucciones del programa, vas a realizar una traza en un hoja de la evolución de la pila, siguiendo el esquema usado habitualmente en las clases de teoría.
- ❏ Cuando se ejecute la primera instrucción del programa, que corresponde a la apilación del primer parámetro, ¿qué valor se almacenará en la cabecera de la pila? Contesta con 8 dígitos hexadecimales [3]. Ciertamente debe ser la dirección de comienzo de la cadena `datos`. Ejecuta esta instrucción y comprueba tu respuesta observando la ventana *memoria 2*. Actualiza el esquema de pila que estás haciendo en papel. No hace falta que escribas el valor numérico del dato, utiliza un símbolo que indique lo que acabas de introducir en la pila.
- ❏ Ejecuta la instrucción que introduce el segundo parámetro en la pila. Observa el parámetro en la ventana *Memoria 2*. Actualiza tu esquema de pila.
- ❏ En este momento, la siguiente instrucción a ejecutar debe ser la de llamada al procedimiento `suma`. Antes de que ejecutes esta instrucción responde a las siguientes preguntas: ¿cuál será el valor del registro `ESP` después de su ejecución? [4] ¿Qué valor aparecerá en la cima de la pila? [5] ¿Cuál será el nuevo valor del registro `EIP`? [6] Ejecuta la instrucción pulsando F11 y comprueba tus respuestas. Si tienes dudas, pregúntale a tu profesor. Ahora actualiza tu esquema de pila.
- ❏ Ahora, antes de ejecutar nuevas instrucciones, completa tu esquema de pila del programa, teniendo en cuenta que la pila alcanza su máximo desarrollo cuando se ejecuta la instrucción `push esi`. Una vez que has completado tu esquema de pila, contesta ¿cuál es el valor mínimo que alcanza `ESP` durante la ejecución del programa? [7] Ahora comprueba esta respuesta ejecutando hasta la instrucción `push esi` inclusive y observando en ese momento el valor de `ESP`.
- ❏ En este momento la siguiente instrucción a ejecutar debe ser `mov esi, [ebp+12]` ¿Qué valor se cargará en el registro `ESI` al ejecutarse esta instrucción? [8] Ciertamente se trata del primer parámetro apilado. Ejecuta la instrucción para comprobar tu respuesta.
- ❏ Ejecuta las instrucciones que siguen en el procedimiento, incluido el bucle, hasta que llegues a la instrucción `pop esi`, pero no ejecutes todavía esta

1

2

3

4

5

6

7

8

instrucción. Comprueba en este momento que el registro `EAX` tiene la suma acumulada de los números de la lista `datos`.

- ❏ Ahora vas a comenzar a eliminar los datos introducidos en la pila. Empezaremos ejecutando las tres instrucciones `pop`. Antes de ejecutar cada una de estas instrucciones haz lo siguiente: mira el valor de `ESP`. Busca en la ventana *Memoria 2* el dato apuntado por `ESP`. Entonces ejecuta la instrucción `pop` observando cómo dicho valor se restaura sobre el registro correspondiente. Observa también cómo se incrementa `ESP` al desapilar. Haz esto para cada una de las tres instrucciones `pop`.
- ❏ Ahora toca ejecutar la instrucción `ret`. Antes de que la ejecutes contesta: ¿que valor se cargará en `EIP` cuanto se ejecute esta instrucción? [9] Ejecuta la instrucción comprobando tu respuesta.
- ❏ Ejecuta la instrucción que elimina los parámetros de la pila y comprueba que tras su ejecución el registro `ESP` tiene el mismo valor que al comienzo del programa.
- ❏ Detén la depuración y cierra el Visual Studio.

[9]

El trazado detallado de este programa nos ha permitido analizar cómo se utiliza la pila del programa durante la ejecución de uno de sus procedimientos.

## 2. Paso de parámetros por valor y por referencia

Existen dos formas de pasar parámetros en la pila, conocidas como paso por valor y por referencia. El paso por valor significa que en la pila se coloca directamente el dato sobre el que el procedimiento tiene que trabajar. Por contra, en el paso por referencia en la pila se coloca una dirección que apunta al dato sobre el que hay que trabajar.

En realidad, en el programa anterior ya has utilizado los dos tipos de paso de parámetros. Has pasado la lista por referencia, ya que has pasado al procedimiento `suma` la dirección de comienzo de la lista. Sin embargo, el número de elementos de la lista lo has pasado por valor.

En esta parte de la práctica vamos a trabajar un poco más este concepto, introduciendo una nueva variable en el programa anterior que pasaremos al procedimiento `suma` de las dos formas, por valor y por referencia.

### 2.1. Paso de parámetros por valor

- ❏ Crea de la forma habitual un nuevo proyecto que se llame `1-6prog2` y agrégale un nuevo fichero llamado `1-6prog2.asm`. Copia en este fichero el programa anterior, es decir, `1-6prog1.asm`
- ❏ En este nuevo programa, vamos a introducir una pequeña diferencia en la lista a procesar. Añade un 5 entre el 43 y el -1.
- ❏ Define una nueva variable de tipo doble palabra, que se llame `num_datos` y que esté inicializada con el valor 7. Define esta variable entre la lista `datos` y la variable `resultado`. El objetivo de esta variable es indicar el número de datos que hay en la lista.

En el programa `1-6prog1.asm` pasabas al procedimiento `suma` el número de datos de la lita apilando el dato inmediato 6, es decir, ejecutando la instrucción `push 6`. Ahora imagina que la lista pudiera tener cualquier número de elementos, y que dicho número está definido en la variable `num_datos`. Entonces es obligado apilar el contenido de esta variable. Fíjate que esto sigue siendo un paso de parámetros por valor. Lo único que varía en este programa respecto al programa anterior es que el parámetro proviene de una variable en vez de ser un dato inmediato.

- ❏ Cambia la instrucción que apila el parámetro *número de datos de la lista* por otra que apile el contenido de la variable `num_datos`.
- ❏ ¿Será necesario que hagas algún cambio en el código del procedimiento `suma` para que éste siga funcionando correctamente? [10]

10

- ❏ Compila y enlaza el programa y comienza la depuración. Vamos a centrarnos nada más que en la apilación del parámetro *número de datos de la lista*.
- ❏ Configura la ventana *Memoria 2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior.
- ❏ Ejecuta la instrucción que coloca en la pila la dirección de `datos`. Observa cómo aparece dicha dirección en la ventana *Memoria 2*
- ❏ Ahora ejecuta la instrucción que coloca en la pila el contenido de la variable `num_datos`. Este valor debe ser un 7. Observa en la ventana *Memoria 2* la ubicación de este valor en la cabecera de la pila.

  Esto es el paso de parámetros por valor. Lo que se ubica en la pila es el valor del parámetro sobre el que trabaja el procedimiento. En este caso un 7.
- ❏ Ejecuta la instrucción `call suma` pulsando F10. Recuerda que esto hace que se ejecute todo el procedimiento. Si todo ha ido bien, en el registro EAX estará el resultado de la suma acumulada de la lista `datos`. Comprueba que el resultado es correcto.
- ❏ Detén la depuración y abandona el Visual Studio.

## 2.2. Paso de parámetros por referencia

Nuestro objetivo ahora será modificar el programa anterior (`1-6prog2.asm`), para que el parámetro *número de datos de la lista* sea pasado por referencia al procedimiento `suma`.

- ❏ Crea de la forma habitual un nuevo proyecto que se llame `1-6prog3` y agrégale un nuevo fichero llamado `1-6prog3.asm`. Copia en este fichero el programa anterior, es decir, `1-6prog2.asm`
- ❏ Cambia la instrucción que apila el parámetro *número de datos de la lista*, de modo que lo que se apila no sea el contenido de la variable `num_datos`, sino la dirección de la misma. Esto es el paso de parámetros por referencia.

  Ahora sí tendrás que hacer modificaciones en el procedimiento `suma`. Dentro de este procedimiento vas a necesitar un registro para direccionar la variable `num_datos`. Este registro tendrás que cargarlo con la dirección de esta variable obtenida de la pila. Supongamos que elegimos para este cometido el registro `EDI`.
- ❏ Agrega al procedimiento `suma` las instrucciones necesarias para salvaguardar y restaurar el registro EDI.

- ❏ Haz el resto de modificaciones que consideres oportunas para que el procedimiento funciones apropiadamente, teniendo en cuenta que lo que se recibe a través de la pila es la dirección de `num_datos` y no su valor.
- ❏ Obtén el ejecutable del programa y depúralo comprobando su correcto funcionamiento. Recuerda que al ejecutar mediante F10 la instrucción `call suma` se ejecuta el procedimiento completo y en `EAX` debe reflejarse la suma acumulada de la lista.
- ❏ Cuando todo funcione bien, detén la depuración.

## 3. Mecanismos de destrucción de los parámetros

Hay dos posibilidades para la destrucción de los parámetros: que ésta sea realizada por el procedimiento llamador, o bien que sea realizada por el procedimiento llamado. En los ejemplos que hemos realizado hasta ahora en esta sesión, la destrucción de los parámetros es realizada por el programa principal que actúa de procedimiento llamador. La idea ahora es modificar uno de estos programas, por ejemplo el `1-6prog3.asm`, para que la destrucción de los parámetros la haga el procedimiento llamado, es decir, el procedimiento `suma`. Para esto tendrás que utilizar la instrucción `ret N`. Además, deberás eliminar del programa principal la instrucción `add esp, 8`.

- ❏ Abre el proyecto `1-6prog3`. Realiza las modificaciones necesarias en el programa para que la destrucción de los parámetros sea realizada por el procedimiento `suma`.
- ❏ Obtén el ejecutable del programa e inicia la depuración pulsando F11. Ahora toma nota del valor que tiene el registro `ESP` al comienzo del programa. [11]
- ❏ Vete ejecutando paso a paso todas las instrucciones del programa hasta que llegues a la instrucción `ret 8`. ¿Qué valor tiene el puntero de pila antes de ejecutar esta instrucción? [12] ¿Qué valor tendrá después de ejecutarla? [13] Ejecuta la instrucción y comprueba tu respuesta. Si todo ha ido bien, en este momento el puntero de pila debe tener el mismo valor que al comienzo del programa. Compruébalo.
- ❏ Detén la depuración y abandona el Visual Studio.

11

12

13

## 4. Ejercicios adicionales

✏ Gracias al paso de parámetros a través de la pila, el procedimiento `suma` puede procesar cualquier lista que se encuentre en la sección de datos del programa. Para probar esto, vas a utilizar el mismo procedimiento para procesar dos listas diferentes. Crea un nuevo proyecto llamado `1-6prog4` y agrégale el fichero `1-6prog4.asm`. Copia en este fichero al código de `1-6prog1.asm`. A continuación de `resultado` define otra lista que se llame `datos_2` que contenga 8 números cualesquiera. A continuación de `datos_2` define la variable `resultado_2` inicializada con 0. Desde el programa principal se llamará dos veces al procedimiento `suma` para procesar las dos listas. El resultado de procesar la lista `datos` se dejará en la variable `resultado` y el resultado de procesar la lista `datos_2` se dejará en la variable `resultado_2` Obtén el ejecutable de este programa y depúralo, comprobando su correcto funcionamiento.

- ✏ Hacer un procedimiento que cuente la cantidad de números negativos que hay en una lista. El procedimiento debe recibir dos parámetros. La dirección de la lista (paso por referencia) y la cantidad de elementos de ésta (paso por valor). El procedimiento devuelve el resultado en el registro `EAX`.
- ✏ Hacer un procedimiento que cuente la cantidad de letras mayúsculas que hay en una cadena de caracteres. La cadena debe terminar con el número 0. El procedimiento recibe como parámetro la dirección de la cadena y devuelve el resultado en el registro `EAX`.

SESIÓN 7

# Procedimientos II

## Objetivos

El objetivo de esta práctica es:

- Profundizar en el conocimiento y comprensión del funcionamiento de los procedimientos.
- Practicar el paso de parámetros a los procedimientos, tanto por valor como por referencia.
- Generalizar el uso de los procedimientos y ver cómo se pueden llamar procedimientos de forma anidada (un procedimiento que llama a otro procedimiento).

## Conocimientos y materiales necesarios

Aparte de los conocimientos teóricos comunes a las dos sesiones anteriores, en esta práctica es necesario:

- El fichero `1-6proc1.asm` que contiene el procedimiento `suma`, cuyo objetivo es sumar los números de una lista.

## Desarrollo de la práctica

El primer objetivo de esta sesión es escribir un programa, cuyo programa principal calcule el cuadrado de la suma de una lista de números definida en su sección de datos. Para ello reutilizaremos el procedimiento `suma`, escrito en la sesión anterior, y escribiremos otro procedimiento llamado `cuadrado`, que calcula el cuadrado de un número que se le pasa como parámetro. Utilizando estos dos procedimientos en secuencia, podremos calcular el cuadrado de la suma. En este ejemplo no se produce anidamiento de procedimientos.

El segundo objetivo de la sesión es hacer un programa que calcule la suma de cuadrados de los números de una lista definida en la sección de datos. Para esto, escribiremos el procedimiento `suma_cuad`, que recibe como parámetros la dirección y el número de elementos de la lista a procesar. `suma_cuad` procesará cada elemento de la lista elevándolo primero al cuadrado, mediante una llamada al procedimiento `cuadrado` y luego agregándolo a la suma acumulada. En este segundo ejemplo el procedimiento `cuadrado` se ejecuta anidado en el procedimiento `suma_cuad`, mostrándose así la técnica del anidamiento de procedimientos.

# 1. Uso de procedimientos sin anidamiento

Comenzaremos reutilizando el procedimiento `suma`, que escribiste en el fichero `1-6prog1.asm`. Recuerda que este procedimiento realiza la suma acumulada de los datos de una lista. El procedimiento recibe dos parámetros a través de la pila: el número de elementos de la lista (pasado por valor) y la dirección de la lista (pasada por referencia), y devuelve el resultado en el registro `EAX`.

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-7prog1` y agrégale un nuevo fichero que se llame `1-7prog1.asm`.
- ❏ Copia el contenido del fichero `1-6prog1.asm` al fichero `1-7prog1.asm`.
- ❏ En el programa `1-7prog1.asm`, modifica la lista de valores que se suman. Los nuevos valores serán ahora:

```
datos   DD   1, 2, 3, 4, 5
```

- ❏ Al cambiar el número de elementos de la lista debes modificar en el programa principal la instrucción que pasa al procedimiento `suma` el número de elementos de la lista.

Tenemos así ya como punto de partida el programa principal y el procedimiento `suma`. Ahora hay que definir el procedimiento `cuadrado`.

**El procedimiento `cuadrado` recibirá a través de la pila el número cuyo cuadrado debe calcular y devolverá el resultado en la registro `EAX`. Para calcular el cuadrado de un número cualquiera `X`, el procedimiento utiliza la técnica de sumar `X` consigo mismo `X` veces. Así por ejemplo, para calcular el cuadrado de 5, sumará 5 veces 5. El propio procedimiento se encargará de destruir el parámetro**.

El procedimiento `cuadrado` será ubicado a continuación del procedimiento `suma`.

- ❏ Edita el fichero `1-7prog1.asm`. Colócate al final del fichero, entre las directivas `suma ENDP` y `END inicio`. Aquí comenzarás a escribir el código del procedimiento `cuadrado`. Sigue el siguiente orden:
  1. Directiva de definición del procedimiento.
  2. Salvaguarda del registro `EBP`.
  3. Actualización de `EBP` con el valor del puntero de pila.
  4. Salvaguarda de los registros que se vayan a utilizar en el procedimiento, excepto `EAX` que se usará para devolver el resultado (con los registros `EDX` y `ECX` sería suficiente.)
  5. Leer el parámetro desde la pila y asignarlo al contador de bucle, `ECX`.
  6. Inicializar el registro `EAX` a cero y `EDX` con el valor del parámetro, que ya tienes en `ECX`.
  7. Sumar `EAX` con `EDX` tantas veces como indique `ECX`, acumulando la suma sobre `EAX`.
  8. Restaurar los registros previamente guardados en la pila.
  9. Retornar destruyendo el parámetro.
  10. Directiva de fin del procedimiento.

Ahora hay que modificar el programa principal. En este momento el programa principal hace una llamada al procedimiento `suma`, el cual devuelve el resultado de la suma acumulada en el registro `EAX`. Entonces, después de la instrucción que destruye los parámetros de `suma` se puede hacer una llamada al procedimiento `cuadrado`, introduciendo antes en la pila el valor sobre el que debe operar. El valor retornado por `cuadrado` será el resultado final y habrá que almacenarlo en la variable `resultado`. Vamos a hacer entonces estas modificaciones.

- ❏ Haz que el resultado del procedimiento `suma` se pase como parámetro desde el programa principal al procedimiento `cuadrado`.
- ❏ El valor devuelto por el procedimiento `cuadrado` en `EAX` será el que se guarde en la variable `resultado`. En realidad, observa que la instrucción que hace esto ya está escrita en el programa principal.

  Esquemáticamente el programa principal debe quedar de la siguiente forma:

```
; Apila parámetros para el procedimiento suma
      ....

; Llama al procedimiento suma
      call suma

; Destruye los parámetros de suma
      add  esp, 8

; Pasa el resultado de suma como parámetro a cuadrado
      ....

; Llama al procedimiento cuadrado
      call cuadrado

; Guarda el valor devuelto por cuadrado en resultado
      ....
```

- ❏ Comprueba con el depurador el correcto funcionamiento del programa y los procedimientos. Para ello recuerda que puedes ejecutar los procedimientos con la tecla F10. Esta tecla hace que el procedimiento se ejecute completamente y retorne. El procedimiento `suma` debe retronar 15 ('F' en hexadecimal) y el procedimiento `cuadrado`, 215 ('E1' en hexadecimal).

## 2. Uso de procedimientos anidados

Vamos a modificar el programa anterior para que en lugar de calcular el cuadrado de la suma, calcule la suma de cuadrados.

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-7prog2` y agrégale un nuevo fichero que se llame `1-7prog2.asm`.
- ❏ Copia el contenido del fichero `1-7prog1.asm` en el fichero `1-7prog2.asm`.

Ahora realizaremos las modificaciones necesarias en el fichero `1-7prog2.asm` para que este programa calcule la suma de cuadrados.

- ❏ En primer lugar, cambia el nombre al procedimiento `suma`. Dale el nombre de `suma_cuad`. Tienes que hacer este cambio en las directivas de comienzo y finalización del procedimiento.
- ❏ El procedimiento `suma` utilizaba el registros `EAX` para acumular la suma de los datos de la lista. Sin embargo en `suma_cuad` no se va a poder utilizar este registro, debido a que dicho registro será utilizado por `cuadrado` para devolver el resultado. Entonces debemos utilizar otro registro para acumular la suma. Elijamos por ejemplo el registro `EDX`. Debes entonces cambiar la instrucción que pone a 0 el registro `EAX` por otra que pone a 0 el registro `EDX`. Además debes añadir al procedimiento las instrucciones necesarias para salvar y restaurar este registro.
- ❏ Ahora deberás hacer cambios en el bucle de procesamiento, es decir, en la parte del procedimiento correspondiente a las siguientes instrucciones:

  ```
  bucle:
    add  eax, [esi]
    add  esi, 4
    loop bucle
  ```

  Este bucle es el que genera la suma acumulada de la lista en `EAX`. Sin embargo, ahora, aparte de que hemos cambiado `EAX` por `EDX`, no queremos sumar los números, sino su cuadrado. Por tanto, tendremos que obtener el cuadrado de cada número antes de acumular su suma sobre `EDX`. Para ello, tendrás que llamar al procedimiento `cuadrado`, pasándole previamente en la pila el número de la lista que se está procesando. El resultado devuelto por `cuadrado` (en `EAX`) habrá que sumárselo a `EDX`.
- ❏ Haz todos los cambios que se acaban de indicar en el bucle de `suma_cuad`
- ❏ El bucle acumula la suma de los cuadrados en el registro `EDX`. Debes añadir una instrucción para llevar este valor al registro `EAX` que es donde el procedimiento devuelve el resultado.

Será necesario hacer también cambios en el programa principal. Los cambios a realizar se indican a continuación:

- ❏ Debes cambiar la llamada al procedimiento `suma` por una llamada al procedimiento `suma_cuad`, ya que este procedimiento ha cambiado de nombre.
- ❏ Debes eliminar del programa la llamada al procedimiento `cuadrado`, ya que ahora este procedimiento es llamado dentro de `suma_cuad`.
- ❏ Obtén el ejecutable del programa y depúralo, observando su correcto funcionamiento. Ten en cuenta que el resultado de la suma de cuadrados debe ser 55 ('37' hexadecimal).
- ❏ Una vez que tengas una versión correcta de tu programa, dibuja un esquema de la pila y después intenta contestar a las siguientes preguntas: ¿qué valor tiene el registro `EBP` en el ámbito de ejecución del procedimiento `suma_cuad`? [1] ¿Qué valor tiene `EBP` en el ámbito de ejecución del procedimiento `cuadrado`? [2] ¿Cuál es el valor mínimo que alcanza el puntero de pila durante la ejecución del programa? [3] Ahora ejecutando el programa paso a paso comprueba tus respuestas.

1

2

3

# 3. Ejercicios adicionales

✏ El procedimiento `cuadrado` no funciona correctamente cuando se le pasa como parámetro un número negativo. Esto es debido a la forma en que se representan los datos negativos. Para solucionar este problema sería necesario introducir las siguientes modificaciones en el procedimiento `cuadrado`:

1. Después de leer el parámetro, comprobar si es negativo.
2. Si es negativo cambiarlo de signo, en el caso contrario no hacer nada.

realiza estos cambios y comprueba ahora que el procedimiento funciona correctamente, tanto con números positivos como negativos.

✏ Crea un programa que cuente el número de letras que aparecen en una cadena de caracteres. Dicha cadena estará terminada mediente el número 0.

El programa principal le pasará como parámetro al procedimiento la dirección donde comienza la cadena de caracteres. El procedimiento va leyendo la cadena carácter a carácter y averiguando si es letra (mayúscula o minúscula) o no. Para ello llama a otro procedimiento al que pasa el carácter como parámetro. Este segundo procedimiento comprueba si el código ASCII del carácter que recibe como parámetro está situado dentro del rango de las letras mayúsculas o minúsculas, en cuyo caso devolverá en el registro `EAX` el valor `1`, devolviendo el valor `0` en el caso contrario. El primer procedimiento contará el número de unos y obtendrá de esa forma el número de letras presentes en la cadena, retornando este valor en el registro `EAX`. El programa principal almacena el resultado del procesamiento de la cadena en una variable de la sección de datos definida para tal efecto. La destrucción de los parámetros es realizada por los propios procedimientos.

SESIÓN 8

# Excepciones

## Objetivos

Los objetivos de esta práctica son:

- Observar cómo se comporta el sistema ante la ocurrencia de excepciones generadas durante la ejecución de los programas.
- Comprender cómo las excepciones sirven para la señalización de errores.
- Comprender el papel crucial que juegan las excepciones en el soporte a los mecanismos de protección.

## Conocimientos y materiales necesarios

- Conocer los conceptos teóricos relativos a las excepciones en la arquitectura IA-32.

## Desarrollo de la práctica

Uno de los objetivos básicos de las excepciones es la señalización de errores ocurridos durante la ejecución de los programas. Así cuando se alcanza un estado de excepción, la CPU interrumpe el mecanismo normal de ejecución de instrucciones y realiza una transferencia de control. Si el software del sistema está bien diseñado, la transferencia se hará hacia una rutina que realice el tratamiento de la excepción ocurrida.

En el caso de un computador de propósito general, como por ejemplo un PC, el sistema se prepara para que las excepciones provoquen siempre una transferencia de control al sistema operativo. Éste será entonces el encargado de intentar buscar una solución al problema causante de la excepción. Algunas excepciones son tratadas internamente por el sistema operativo de una forma totalmente transparente al usuario. Un ejemplo típico de estas excepciones es el fallo de página que estudiarás en el tema de la gestión de memoria. Sin embargo, hay otras excepciones que no tienen asignadas en el sistema operativo una rutina de tratamiento que pueda solventar el problema que las causó. Cuando ocurren estas excepciones, lo único que puede hacer el sistema operativo es señalizar el problema y terminar la ejecución del programa causante de la excepción.

En esta práctica vas realizar programas en los que crearás problemas a propósito para que se generen excepciones, entonces comprobarás cómo se comporta el sistema. Primero

probarás excepciones que señalizan errores, como el de división por 0 y, después, probarás las excepciones que se producen cuando se viola algún mecanismo de protección. Todas estas pruebas las harás en modo depuración, es decir, mientras el programa que genera la excepción está siendo depurado con el Visual Studio. Después, comprobaremos cómo se comporta el sistema cuando un programa en ejecución normal (es decir, fuera del entorno de depuración) genera una excepción.

# 1. Errores

Comenzaremos por analizar ejemplos de excepciones que señalizan errores. Ejemplos de estas excepciones son la de división por 0, y el código de operación inválido. En esta sección probaremos la excepción de división por 0. Después, en la sección de ejercicios adicionales se propone un ejercicio para probar el código de operación inválido.

Para probar la excepción de división por 0 deberemos ejecutar una instrucción de división, por ejemplo, `idiv`. Esta instrucción realiza la división de un número de 64 bits entre un número de 32 bits y genera como resultados un cociente y un resto, ambos de 32 bits. Los operandos sobre los que trabaja esta instrucción son:

- Dividendo (64 bits) -> EDX:EAX -> operando implícito
- Divisor (32 bits) -> r32/m32 -> operando explícito

Con relación al resultado, la instrucción deja el cociente en el registro `EAX` y el resto en el `EDX`.

Vamos entonces a introducir esta instrucción en un programa simple y ejecutarla, primero sin que genere ningún problema y después haciendo que genere una excepción.

❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-8prog1` y agrégale un nuevo fichero que se llame `1-8prog1.asm`.

❏ Copia en este fichero el listado del siguiente programa.

```
1
2  .386
3  .MODEL flat, stdcall
4  ExitProcess PROTO, :DWORD
5
6  .DATA
7
8  .CODE
9
10 inicio:
11
12   ; Carga del dividendo (EDX:EAX)
13   xor  edx, edx
14   mov  eax, 17
15
16   ; Carga del divisor (r32/m32), usamos ESI
17   mov  esi, 4
18
19   ; División (cociente -> EAX; resto -> EDX)
20   idiv esi
21
```

```
22    ; Terminar retornando al S.O.
23    push  0
24    call  ExitProcess
25
26  END  inicio
27
```

❏ En este programa se utiliza la instrucción `idiv` para dividir 17 entre 4. Compila, enlaza y depura este programa. Fíjate al ejecutar la instrucción `idiv` en cómo se actualizan `EAX` y `EDX` con el cociente y el resto de la división, respectivamente.

Hasta aquí no ha habido ningún problema, ya que no debe surgir ninguna dificultad al dividir 17 entre 4. Sin embargo, ahora vas a cargar un 0 en el operando que hace de divisor. En este caso, la CPU no puede calcular el resultado. Entonces señaliza el problema generando una excepción de división por 0. Vamos a comprobar esto.

❏ En el programa anterior, sustituye la instrucción "`mov esi, 4`" por "`mov esi, 0`". Esto hace que el divisor de la instrucción `idiv` sea 0.

❏ Compila, enlaza y depura el programa. Cuando ejecutes la instrucción `idiv`, la CPU generará la excepción de división por 0. Como estamos ejecutando el programa desde el entorno de depuración, la excepción será recogida por el entorno, el cual muestra una ventana que proporciona información de la excepción ocurrida. La figura 8.1 muestra esta ventana. Pulsa en el botón *Interrumpir*, puesto que va a ser imposible continuar la ejecución de este programa.

❏ Vamos a analizar la información que proporciona esta ventana. En primer lugar se indica la dirección en la que se encuentra la instrucción que ha generado la excepción. Abre la ventana *Desensamblador* y comprueba que la instrucción `idiv` se encuentra en la dirección indicada en la ventana. Después se muestra el nombre del programa que ha generado la excepción, que en este caso es `1-8prog1.exe`. Finalmente se proporciona un identificador y un texto explicativo de la excepción generada. ¿Cuál es el identificador asociado por Windows a la excepción ocurrida? [1]

1

❏ Los identificadores de excepción utilizados por Windows requieren una aclaración. En la carpeta de la asignatura hay un fichero llamado Codigos-de-Excepcion-de-Windows, en el que se indica el código que asigna Windows a cada tipo de excepción. Mira en dicho fichero el código correspondiente a la excepción de división por 0, y comprueba que se corresponde con la excepción que ha generado el programa `1-8prog1.exe`.

❏ Ten en cuenta que los identificadores de excepciones utilizados por Windows son identificadores internos del sistema operativo y que no coinciden con los números de excepción asignados por la CPU. Escribe a continuación el número de excepción que asocia la CPU IA-32 a la excepción de división por 0 [2]. Si tienes dudas míralo en la transparencia "Uso de la IDT en la plataforma PC/Windows".

2

❏ Finalmente abandona la depuración del programa.

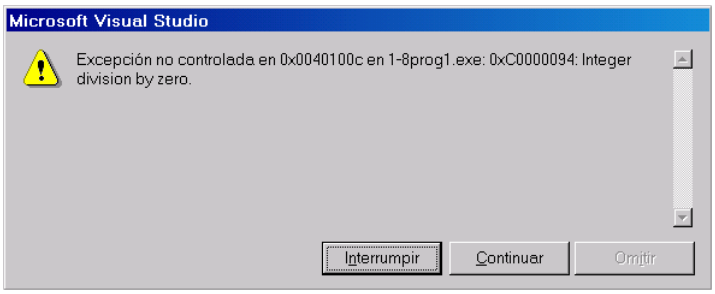


Figura 8.1: Ventana mostrada por el depurador al generarse una excepción de división por 0

## 2. Violaciones de protección

Otro de los objetivos fundamentales de las excepciones es señalizar las violaciones de los mecanismos de protección llevadas a cabo por los programas. Vamos a plantear ejemplos de tres casos diferentes de violación de protección. Comprobarás cómo los mecanismos de protección restringen las operaciones que los programas pueden llevar cabo, evitando así que un programa mal intencionado o con errores de programación pueda dañar la integridad del sistema.

Con objeto de agilizar el desarrollo de la práctica, en vez de hacer un programa diferente para probar cada caso de violación de protección, haremos las tres pruebas sobre el mismo programa.

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-8prog2` y agrégale un nuevo fichero que se llame `1-8prog2.asm`.
- ❏ Copia en este fichero el listado del siguiente programa.

```
1
2  .386
3  .MODEL flat, stdcall
4  ExitProcess PROTO, :DWORD
5
6  .DATA
7
8  .CODE
9
10 inicio:
11
12   ; Viaolación de protección por acceso a un registro de sistema
13   ; Poner a 0 el bit IF del registro de estado
14   ; .....
15
16   ; Violación de protección por acceso a un dispositivo de E/S
17   ; Escribir el dato 0FFh en el registro IMR del PIC
18   ; .....
19   ; .....
20
21   ; Violación de protección en el acceso a memoria
22   ; Escribir un 0 en la primera posición de la IDT
23   ; .....
24   ; .....
25
26   ; Terminar retornando al S.O.
27   push  0
```

```
28    call  ExitProcess
29
30  END  inicio
31
```

## 2.1. Violación de protección por acceso a un registro de sistema

Los programas de usuario no deben tener acceso a los registros de sistema. Un ejemplo de registro de sistema es el registro de estado. Hay diversas instrucciones que trabajan con este registro. Por ejemplo, las instrucciones `sti` y `cli` se utilizan, respectivamente, para poner a 1 o a 0 el bit IF (Interrupt Flag) del registro de estado. Este bit es muy importante porque es el que determina si la CPU acepta o no interrupciones. Como puedes comprender no se puede permitir que un programa de usuario tenga permiso para modificar el estado de este bit. Vamos a comprobar qué ocurre cuando intentamos modificarlo.

- ❏ Escribe en el lugar apropiado del programa `1-8prog2.asm` la instrucción que pone a 0 el bit IF del registro de estado. Compila y enlaza el programa y comienza la depuración. Ahora ejecuta la instrucción `cli`. Entonces se produce una excepción. Anota el identificador asignado por Windows a dicha excepción [3]. El depurador indica que se trata de instrucción privilegiada. Busca en el fichero Codigos-de-Excepcion-de-Windows este indentificador y comprueba que se corresponde con el de instrucción privilegiada.
- ❏ Abandona la depuración y comenta la instrucción `cli`. Si no lo haces, no se podrían ejecutar las instrucciones que escribirás a continuación para probar el segundo tipo de violación de protección.

[3]

## 2.2. Violación de protección por acceso a un dispositivo de E/S

Los programas de usuario tampoco deben tener acceso a los dispositivos de E/S. Para analizar qué ocurre cuando un programa quiere acceder a uno de estos dispositivos, utilizaremos el PIC, que es el circuito que gestiona las interrupciones del computador. Este circuito contiene un registro, el IMR, cuyos bits determinan las interrupciones que serán tratadas o rechazadas. Este registro se ubica en la posición 21h del espacio de direcciones de E/S. Si envías el dato 0FFh al registro IMR, el sistema deja de aceptar interrupciones, lo que llevaría al "cuelgue" del ordenador. Como puedes comprender, no se puede permitir a los programas de usuario que lleven a cabo este tipo de operaciones. Vamos a probar qué ocurre cuando un programa viola esta norma.

- ❏ Escribe en el lugar apropiado del programa `1-8prog2.asm` las instrucciones necesarias para enviar el dato 0FFh al registro IMR del PIC. Necesitarás dos instrucciones. Recuerda que para enviar un dato a una dirección de E/S deberás usar la instrucción "`out num_puerto, al`". Compila y enlaza el programa y comienza la depuración. Cuando se ejecute la instrucción `out` se producirá un excepción que es del mismo tipo que la del ejemplo anterior, es decir, instrucción privilegiada.
- ❏ Abandona la depuración y comenta las instrucciones anteriores.

Las instrucciones de E/S (`in` y `out`) son también instrucciones privilegiadas y no pueden ser ejecutadas por los programas de usuario.

### 2.3. Violación de protección en el acceso a memoria

Ahora vamos a comprobar que los programas no pueden acceder libremente a todas las direcciones del espacio de direcciones de memoria. Más bien, ocurre todo lo contrario. A modo de ejemplo, vamos a ver cómo se comporta el sistema cuando un programa intenta acceder a una estructura de datos del sistema operativo, por ejemplo, la IDT. Este estructura se ubica a partir de la dirección 80036400h. Para llevar a cabo una operación de lectura o escritura sobre una posición del la IDT, necesitas cargar un registro con la dirección en donde se encuentra ubicada esta estructura y, después, leer de la posición apuntada por el registro o escribir en ella.

- ❏ Escribe en el lugar apropiado del programa `1-8prog2.asm` las instrucciones necesarias para escribir un valor cualquiera en la dirección de comienzo de la IDT. Necesitarás dos instrucciones. Compila y enlaza el programa y comienza la depuración. Cuando se ejecute la instrucción que escribe en memoria se producirá una excepción. Anota el identificador asignado por Windows a dicha excepción [4]. El depurador indica que se trata de una infracción de acceso. Además indica la dirección de memoria a la que se está intentado acceder y causa la infracción de acceso. Busca en el fichero Codigos-de-Excepcion-de-Windows este indentificador y comprueba que se corresponde con el de violación de acceso. Finalmente abandona la depuración.

  La infracción de acceso a memoria es la excepción más común generada por los programas. Cuando las direcciones de memoria no se manejan correctamente se generan excepciones de este tipo.

  4
- ❏ Haz los cambios necesarios en el programa anterior para analizar si puedes leer de la IDT en vez de escribir en ella. Ejecuta la nueva versión del programa y contesta: ¿puedes leer del área de memoria en donde se encuentra la IDT? [5]

  5
- ❏ Abandona el programa.

La idea es que un programa no puede utilizar la memoria sin restricciones, sino más bien todo lo contrario. De esta forma los errores en el manejo de direcciones cometidos en un programa no dañarán a otros programas o al sistema operativo.

## 3. Manejo de excepciones ocurridas en ejecución normal

Hasta ahora hemos visto cómo el depurador maneja las excepciones que ocurren en un programa mientras éste está siendo depurado. Ahora vamos a ver cómo se gestionan las excepciones cuando el programa es ejecutado en modo normal. Cuando esto sucede, los pasos dados por el sistema operativo son los siguientes:

1. Señalizar la ocurrencia de la excepción. Esto se lleva a cabo mostrando una ventana que indica mediante un mensaje la ocurrencia de la excepción. Entonces se proporcionan al usuario dos posibilidades: abandonar el programa, o bien transferir el control a un depurador que nos permita depurar el programa y encontrar la causa de la excepción.

2. Si se decide abandonar el programa, se saca el programa de ejecución y se devuelve el control al sistema operativo.

3. Si se decide llamar al depurador, se transfiere el control a la herramienta de depuración que haya sido establecida como depurador por defecto en el entorno de configuración del sistema. Cuando se instala el Visual Studio, su entorno de depuración se convierte en el depurador por defecto. Este es precisamente nuestro caso.

Para probar todo esto, vamos a utilizar el primer programa de esta sesión, es decir, `1-8prog1.asm`. En este momento este programa genera una excepción de división por 0. Primero modificaremos este programa para que no genere la excepción y lo ejecutaremos en modo normal, observando su comportamiento. Después, volveremos a modificar el programa para que genere la excepción, observando cómo la excepción es gestionada por el sistema.

- ❏ Abre el proyecto `1-8prog1`. En su programa asociado cambia la instrucción "`mov esi, 0`" por "`mov esi, 4`". Así el divisor de la instrucción `idiv` no será cero y no generará una excepción de división por cero al ejecutarse. Compila y enlaza el programa. Ahora vas a ejecutar el programa, pero sin depurarlo, es decir, vas a ejecutarlo como cualquier otro programa normal. Abre una consola ejecutando el comando `CMD` desde el menú *inicio*. Ahora debes ubicar esta consola en la carpeta en la que se encuentra el programa ejecutable (`1-8prog1.exe`) que acabas de generar. Recuerda que el Visual genera los ejecutables en una carpeta llamada `Debug`, que está dentro de la carpeta del proyecto. Una vez que hayas ubicado la consola en esta carpeta, ejecuta `1-8prog1.exe`. Observarás que el programa se ejecuta sin que ocurra nada aparente. Esto es así porque el programa no hace ninguna operación de E/S y, por consiguiente, no genera ningún efecto visible. No obstante, el programa se ejecuta sin generar problemas.
- ❏ Ahora vas a modificar `1-8prog1.asm` para que genere una excepción de división por 0. Para ello, cambia la instrucción "`mov esi, 4`" por "`mov esi, 0`". Compila y enlaza el programa. Cierra el Visual Studio. Vuelve a ejecutar en la consola `1-8prog1.exe`. Ahora este programa genera una excepción de división por 0, lo que provoca una transferencia de control al sistema operativo. Entonces el sistema operativo muestra una ventana indicando que se ha producido un error en la aplicación y proporciona al usuario el identificador de la excepción ocurrida. En la figura 8.2 se muestra la ventana de aviso generada por el sistema operativo al ejecutarse `1-8prog1.exe`, cuando éste genera la excepción de división por 0.

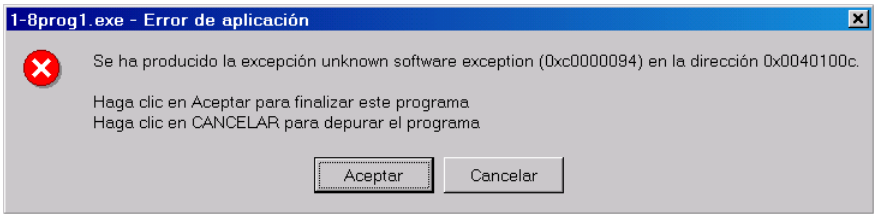


Figura 8.2: Ventana mostrada por el sistema operativo al generarse una excepción de división por 0

En la ventana de aviso proporcionada por el sistema operativo señalizando la excepción se dan dos opciones al usuario, *Aceptar* y *Cancelar*. Vamos a probar cada una de ellas.

- *Aceptar* provoca que se cancele la ejecución del programa. Pulsa *Aceptar*. Entonces el programa se elimina de ejecución y se devuelve el control a la consola del sistema.
- Ahora vamos a probar la opción *Cancelar*. Esta opción provoca que el sistema operativo busque un depurador que permita depurar el programa causante de la excepción. Entonces el sistema muestra la ventana representada en la figura 8.3. Esta ventana indica que hay una excepción no manejada en el programa `1-8prog1.exe`, e informa de los depuradores que están disponibles para depurarla. En nuestro caso sólo está disponible el proporcionado por el Visual Studio 2005. Pulsa *Sí* para comenzar la depuración. Esto hace que se abra el Visual Studio en modo depuración, abriendo éste a su vez el programa causante de la excepción, y señalizando la instrucción que ha provocado la excepción, que como puedes observar es "`idiv esi`".

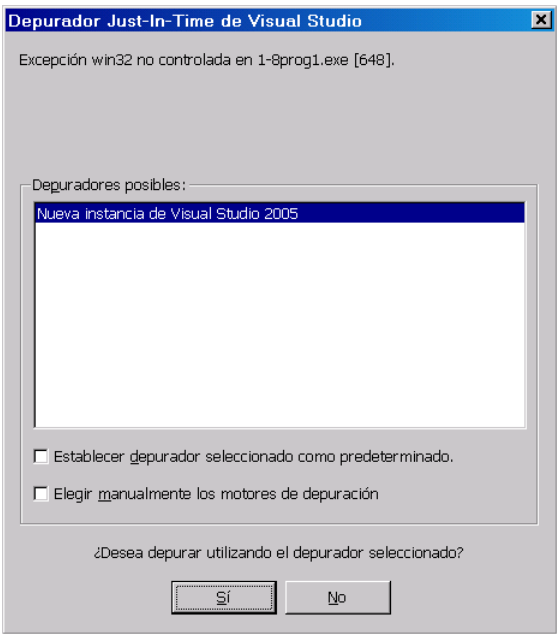


Figura 8.3: Ventana mostrada por el sistema operativo para que el usuario seleccione el depurador que se utilizará para depurar una excepción

Los sistemas de depuración nos proporcionan un mecanismo indispensable para buscar errores debidos a excepciones, que en algunos casos son extraordinariamente difíciles de encontrar.

## 4. Ejercicios adicionales

✏ El objetivo de este ejercicio es probar la excepción de código de operación inválido. Esta excepción se produce cuando el `EIP` apunta a una dirección de memoria en la que no hay ningún código de instrucción interpretable por la CPU. Entonces al tratar la CPU de acceder a la siguiente instrucción se produce la excepción. Esto puede ocurrir si debido a algún error, el `EIP` se carga con una dirección incorrecta.

Para generar esta excepción tendremos que conseguir introducir en una área de código unos bytes que no sean interpretables como instrucciones. Entonces al ser alcanzados

por el EIP se producirá la excepción. Esto parece difícil de conseguir en un programa normal, ya que en la sección de código debemos escribir instrucciones correctas y el compilador traducirá siempre estas instrucciones a código máquina correcto. Sin embargo, podemos utilizar un truco, que si bien no debe utilizarse nunca cuando se siguen unas prácticas correctas de programación, en este caso nos servirá para alcanzar el objetivo perseguido. Se trata de utilizar directivas del tipo DB, DW o DD en la sección de código. Así por ejemplo, utilizando una directiva DB podremos poner en cualquier punto de la sección de código el byte que queramos. **Debe resaltase otra vez que esta técnica no debe utilizarse nunca en la práctica habitual de la programación en ensamblador**.

Antes de probar esto, debemos encontrar un byte o una secuencia de bytes que no sean interpretables como instrucción por parte de la CPU. Esto no es tan sencillo porque están casi todos los bytes utilizados. El código de operación de las instrucciones de la arquitectura IA-32 puede ser de 1 o de 2 bytes. Si el código es de 2 bytes, el primer byte comienza por 0F. Todos los códigos de 1 byte son reconocidos como instrucciones, por tanto tendremos que elegir un código de dos bytes. Por ejemplo, el código 0F 38 no se corresponde con ninguna instrucción.

- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `1-8prog3` y agrégale un nuevo fichero que se llame `1-8prog3.asm`.
- ❏ Copia en este fichero el listado del siguiente programa.

```
1
2   .386
3   .MODEL flat, stdcall
4   ExitProcess PROTO, :DWORD
5
6   .DATA
7
8   .CODE
9
10  inicio:
11    push eax
12    pop  eax
13    push eax
14    pop  eax
15
16    DB   0Fh
17    DB   38h
18
19    push eax
20    pop  eax
21    push eax
22    pop  eax
23
24    ; Terminar retornando al S.O.
25    push  0
26    call  ExitProcess
27
28  END  inicio
29
```

- ❏ Compila y enlaza el programa y comienza la depuración pulsando F11 una vez.
- ❏ Abre la ventana *Desensamblador* y configúrala para que no muestre el código fuente. Ahora observa la interpretación que hace el Visual Studio del código máquina. Cuando alcanza el byte $0F$ muestra una interrogación, ya este $0F$ junto con el $38$ que le sigue no sabe interpretarlos.

Después junta el 38 con los bytes siguientes apareciendo una instrucción `cmp`. Claramente el código del programa es incorrecto.

- ❑ Ejecuta paso a paso el programa hasta que alcances el byte "0F" y se produzca la excepción. Anota el identificador asociado por Windows a la excepción ocurrida [6]. El depurador indica que se trata de una instrucción ilegal. Busca en el fichero Codigos-de-Excepcion-de-Windows este indentificador y comprueba que se corresponde con el de instrucción ilegal.
- ❑ Finalmente abandona la depuración de esta programa.

6

SESIÓN 9

# Estructura en memoria de un programa C

## Objetivos

Los objetivos de esta práctica son:

- Comprender la estructura que adquiere un programa C cuando se encuentra en ejecución.
- Observar las estrategias que utiliza el compilador de C para transformar las sentencias de un programa en lenguaje máquina, poniendo especial atención en los modos de direccionamiento que utiliza para acceder tanto a variables globales como locales.

## Conocimientos y materiales necesarios

Aparte de los conocimientos teóricos comunes a las dos sesiones anteriores, en esta práctica es necesario:

- Conocer el mecanismo estandarizado de creación y destrucción del marco de pila.
- Conocer las estrategias básicas utilizadas por un compilador de C para transformar las sentencias de un programa en lenguaje máquina.
- Conocer las estrategias básicas utilizadas por un compilador de C para transformar a lenguaje máquina el cuerpo de las funciones, así como las llamadas a las mismas.
- Manejar las tablas de codificación de instrucciones, poniendo especial énfasis en la codificación del direccionamiento directo a memoria.
- Antes de realizar esta práctica se debe hacer el Apéndice B.

# Desarrollo de la práctica

## 1. Variables globales: la sección de datos

Las variables globales de un programa C son las que dan lugar a la sección de datos del programa. Vamos a observar esto mediante un ejemplo simple de programa.

- ❏ Siguiendo los pasos indicados en el Apéndice B, crea un nuevo proyecto para un programa C llamado `1-9prog1` y agrégale un nuevo fichero que se llame `1-9prog1.c`.
- ❏ En el *Explorador de soluciones*, pulsa con el botón derecho del ratón sobre el fichero `1-9prog1.c` y elige la opción *Propiedades*. Se abre entonces la *Página de propiedades del fichero*. En el árbol que se muestra a la izquierda, denominado *Propiedades de configuración*, despliega la rama *C/C++*. En esta rama se pueden configurar múltiples aspectos acerca de cómo será utilizado el compilador de C/C++ para la compilación del fichero. En nuestro caso debemos indicar al compilador que no introduzca *comprobaciones básicas de tiempo de ejecución*, ya que esto generaría un código más "farragoso", que dificultaría nuestra comprensión del lenguaje máquina del programa. Para hacer esto, selecciona *Generación de código* y en las opciones que se muestran a la izquierda, pulsa sobre *Comprobaciones básicas en tiempo de ejecución*. Entre las opciones posibles, debes seleccionar *Predeterminado*. Esto hace que se deshabiliten las *Comprobaciones básicas en tiempo de ejecución*, que es lo que queremos.
- ❏ En el fichero `1-9prog1.c`, copia el código del siguiente programa:

```
1
2  int A=1, B=-1;
3
4  main()
5  {
6          A=B;
7  }
8
```

- ❏ Compila y enlaza este programa y comienza su depuración, pulsando F11. Observa que la flecha amarilla queda justo en la llave de apertura de la función `main`.
- ❏ Lo primero que queremos saber es dónde se encuentran las variables globales del programa. Para ello vamos a utilizar la ventana *Inspección 1*, que se encuentra en el área inferior derecha del entorno. Es la primera vez que utilizamos esta ventana, pero su nombre indica perfectamente su cometido. Mediante esta ventana, el entorno proporciona información, entre otras cosas, sobre valores y direcciones de variables. En el campo *Nombre* de esta ventana escribe `A` y pulsa Enter. Entonces se muestra en el campo *Valor* el contenido de la variable. Haz lo mismo para la variable `B`. Sin embargo, lo que nos interesa conocer ahora es la dirección de estas variables. Para indicarle al entorno que lo que se desea es la dirección de una variable, se precede el nombre de la variable con el símbolo '`&`'. Teniendo en cuenta esta indicación, obtén mediante la ventana *Inspección 1* las direcciones de las variables `A` y `B`. Toma nota de la dirección de `A` [1] y también de `B` [2].

[1]

[2]

- Las variables `A` y `B` forman la sección de datos. Una vez conocidas sus direcciones podremos ubicarlas en la memoria. Para ello utilizamos, al igual que hemos hecho en las prácticas de ensamblador, la ventana *Memoria 1*. Posiciona esta ventana en la dirección de la variable `A` y observa su contenido en la memoria. A continuación de `A`, debes ver en la memoria también el contenido de `B`. Ahora, mientras observas en la memoria el contenido de `A`, ejecuta la sentencia "`A = B`" (como siempre pulsando F11). Ciertamente ocurre lo esperado, la variable `B` se copia en `A`.
- Ahora vamos a ver los modos de direccionamiento que se utilizan en el código del programa para acceder a las variables `A` y `B`. Para ello, abre la ventana *Desensamblador*. Pulsa con el botón derecho del ratón en un punto cualquiera de esta ventana y comprueba que se encuentran activadas las siguientes opciones: *Mostrar dirección*, *Mostrar código fuente*, *Mostrar nombre de símbolos* y *Mostrar barra de herramientas*. Esta es la mejor configuración para esta ventana. Como la ventana está configurada para que muestre el código fuente, observarás en color negro las sentencias de código fuente del programa y en un color gris más atenuado, la traducción a lenguaje ensamblador de dichas sentencias. Ubícate en la sentencia "`A = B`". Observa que la traducción a lenguaje ensamblador de dicha sentencia es

  ```
  mov eax, dword ptr [_B (415004h)]
  mov dword ptr [_A (415000h)],eax
  ```

  ¿Entiendes cómo se ha traducido la sentencia "`A = B`" a lenguaje ensamblador? Si tienes alguna duda pregúntale a tu profesor. En las instrucciones anteriores se muestra el modo de direccionamiento que utiliza el compilador para acceder a las variables globales, se trata del modo de direccionamiento conocido como *direccionamiento directo a memoria*. Para asegurarte de que comprendes bien este modo de direccionamiento, vas a codificar la primera de estas instrucciones, es decir, la que copia en `EAX` el contenido de la variable `B`. Para codificar esta instrucción necesitas las transparencias de codificación de instrucciones, a las que puedes acceder utilizando el enlace correspondiente en la página web de la asignatura. Codifica la instrucción teniendo en cuenta que se trata de uno de los casos especiales que se indican en la transparencia *Códigos especiales para instrucciones que usan AL o EAX*. Si tienes dudas, pregúntale a tu profesor. Indica el código que has calculado. 3

  3

  Ahora debes comprobar que has codificado correctamente la instrucción. Para ello, pulsa con el botón derecho del ratón en la ventana *Desensamblador* y elige la opción *Mostrar bytes de código*. Esto muestra la codificación elegida por el compilador para compilar las instrucciones. ¿Coincide la codificación que calculaste con la mostrada en la ventana *Desensamblador*? Si es así, todo ha ido bien, si no debes detectar donde está el problema.

  Vuelve a eliminar la selección de la opción *Mostrar bytes de código* en la ventana *Desensamblador*.

## 2. Funciones: la sección de código

Los cuerpos de las funciones de un programa C son los que dan lugar a la sección de código del programa. Observaremos mediante un programa simple, formado por dos funciones, cómo el compilador genera código para dichas funciones.

- ❏ Siguiendo los pasos indicados en el Apéndice B, crea un nuevo proyecto para un programa C llamado `1-9prog2` y agrégale un nuevo fichero que se llame `1-9prog2.c`.
- ❏ En el *Explorador de soluciones*, pulsa con el botón derecho del ratón sobre el fichero `1-9prog2.c` y elige la opción *Propiedades*. Al igual que hiciste en el programa anterior, debes configurar en el apartado de *Generación de código* la opción *Comprobaciones básicas en tiempo de ejecución* con el valor *Predeterminado*.
- ❏ En el fichero `1-9prog2.c`, copia el código del siguiente programa:

```
1
2  int cuadrado ( int );  //prototipo
3
4  int A=0;
5
6  main()
7  {
8          A=cuadrado(4);
9  }
10
11 int cuadrado( int par )
12 {
13   int cuad=0;
14
15   cuad = par*par;
16
17   return cuad;
18 }
19
```

- ❏ Compila y enlaza este programa y comienza su depuración, pulsando F11. Observa que la flecha amarilla queda justo en la llave de apertura de la función `main`.
- ❏ Vamos a observar el código generado por el compilador de C para este sencillo programa. Para ello abre la ventana *Desensamblador*. Asegúrate de que en esta ventana está activada la opción *Mostrar código fuente*. Comenzaremos analizando el código de la función `main`. Para ello, a continuación se indica un subconjunto representativo de las acciones básicas llevadas a cabo por `main`, y debes identificar las instrucciones que llevan a cabo a cada acción. Entonces escribe en los cuadros de la derecha, la instrucción o instrucciones que corresponden a cada acción. Estas acciones son las siguientes:

  Salvaguardar el registro `EBP` [4]

  Hacer que `EBP` apunte al marco de pila de `main` [5]

  Salvaguardar registros (el compilador de C siempre salvaguarda tres registros, además de `EBP`, de forma estándar) [6]

  Apilar el parámetro para el procedimiento `cuadrado` [7]

  Destruir los parámetros del procedimiento `cuadrado` [8]

  Restaurar los registros salvaguardados [9]

  Retornar [10]

- ❏ Teniendo en cuenta el código que has observado, ¿qué convenio utiliza el compilador de C para llevar a cabo la destrucción de los parámetros, es decir, qué procedimiento destruye los parámetros, el llamador o el llamado? [11]

4

5

6

7

8

9

10

11

En el análisis de la función `main` se ha omitido, a propósito, un aspecto importante, la reserva de espacio para variables locales. En las clases de teoría se ha indicado que la instrucción estándar usada por los procedimientos para llevar a cabo dicha reserva es "`sub esp, x`", siendo `x` el tamaño ocupado por las variables locales. Esta instrucción abre un hueco en la pila para las variables locales. Después, el acceso a estas variables se realiza utilizando desplazamientos negativos respecto a `EBP`. En la figura 9.1(A) se muestra la estructura que adquiere la pila de forma estándar durante la ejecución de un procedimiento. En la figura debe resaltarse la posición que ocupan las variables locales dentro de la pila.

El compilador de C/C++ del Visual Studio introduce un espacio adicional en la pila. Se trata de un *buffer* para almacenamiento de datos temporales, que no se encuentra documentado en la ayuda del compilador [1]. Se sospecha que este *buffer* es utilizado por el depurador. Dicho *buffer*, que tiene un tamaño de 64 bytes (40h), se ubica entre las variables locales y el área de la pila utilizada para salvaguardar los registros, tal y como se muestra en la figura 9.1(B). La forma de generar este *buffer* es muy sencilla para el compilador, así en vez de generar la instrucción "`sub esp, x`" para reservar espacio para las variables locales (siendo `x` el tamaño requerido por estas variables), el compilador genera la instrucción "`sub esp, y`", donde "`y=x+40h`". Así el espacio total reservado por esta instrucción es `x` bytes para las variables locales más 40h bytes para el *buffer*.

- ❏ Si has entendido la explicación anterior, ¿sabrías explicar por qué la función `main` utiliza la instrucción "`sub esp, 40h`", dentro del grupo de instrucciones estándar de entrada a la función?
- ❏ Siguiendo con este mismo tema, busca en la ventana *Desensamblador* el código de la función `cuadrado`. Está un poco después del código de `main`. ¿Por qué en esta función se utiliza la instrucción "`sub esp, 44h`", dentro del grupo de instrucciones estándar de entrada a la función? Si tienes dudas sobre estas dos últimas cuestiones, pregúntale a tu profesor.
- ❏ ¿Qué instrucción es utilizada tanto por la función `main` como por la función `cuadrado` para eliminar de la pila el *buffer* y las variables locales? 12

12

Finalmente vamos a ver los modos de direccionamiento que utiliza el procedimiento `cuadrado` para acceder a su parámetro (`par`) y a su variable local (`cuad`). La sentencia de la función `cuadrado` en la que se accede a estos dos elementos es "`cuad = par*par`".

- ❏ Utilizando la ventana *Desensamblador* comprueba que las instrucciones utilizadas por el compilador para compilar esta sentencia son las siguientes:

```
1  mov   eax,dword ptr [par]
2  imul  eax,dword ptr [par]
3  mov   dword ptr [cuad],eax
```

Como puedes observar, para llevar a cabo la multiplicación del parámetro por sí mismo, el compilador utiliza la instrucción `imul`, que no ha sido vista en las clase de teoría. Se trata de la instrucción de multiplicación entera. Su funcionamiento es muy simple, multiplica sus dos operandos, considerados con signo, y deja el resultado en el operando destino.

[1] El uso de este *buffer* no sigue el estándar de compilación de funciones del lenguaje C (es un mecanismo utilizado solo por el compilador del Visual). Debido a esto no se ha explicado en las clases de teoría y se introduce en esta práctica con el único objetivo de comprender correctamente el código generado por el compilador de C/C++ del Visual Studio.

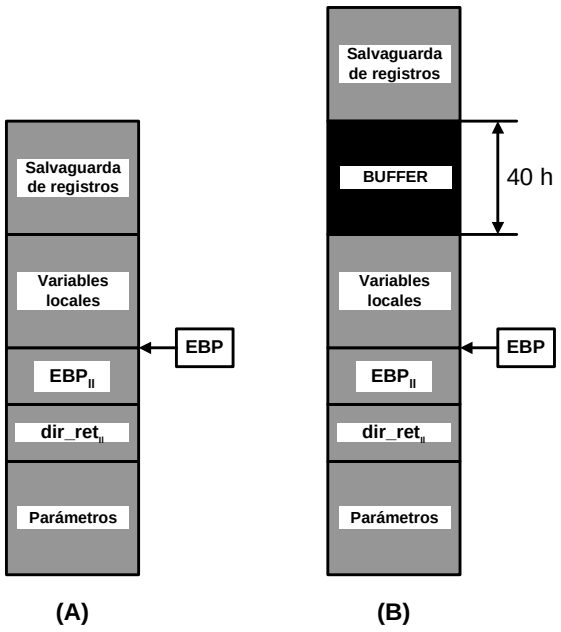


Figura 9.1: Estructura de la pila durante la ejecución de de un procedimiento: (A) sin buffer para datos temporales y (B) con buffer

Sin embargo, el punto en el que queremos centrar la atención respecto a las instrucciones anteriores es en los modos de direccionamiento utilizados por el compilador para acceder al parámetro y a la variable local. El compilador, con objeto de facilitar al programador la comprensión del código, introduce las etiquetas `par` y `cuad` en la representación ensamblador de las instrucciones, pero hay que insistir en que esto es solo un recurso usado por el entorno de trabajo para facilitar las cosas el programador. Nosotros sabemos, según se ha explicado en las clases de teoría, que a los parámetros y a las variables locales se accede mediante el registro `EBP`.

- ❏ Reescribe las tres instrucciones anteriores sustituyendo las etiquetas `par` y `cuad` por los correspondientes modos de direccionamiento mediante el registro `EBP`. Recuerda que a los parámetros se accede mediante desplazamientos positivos y a las variables locales, mediante desplazamientos negativos. Si tienes alguna duda respecto a esto, pregúntale a tu profesor. Escribe tu respuesta en el siguiente cuadro:

```
1
2
3
```

Finalmente vamos a comprobar que el lenguaje máquina generado por el compilador corresponde a las instrucciones que acabas de escribir, en las que se utilizan modos de direccionamiento mediante el registro `EBP`.

- ❏ Utilizando la tabla de codificación de instrucciones, codifica la instrucción "`mov eax, dword ptr [ebp+x]`", en donde `x` es el desplazamiento que has calculado antes que te permite acceder al parámetro. [13]
- ❏ Utilizando la tabla de codificación de instrucciones, codifica la instrucción "`mov dword ptr [ebp-y], eax`" en donde `y` es el desplazamiento que has calculado antes que te permite acceder a la variable local. [14]

13

14

❏ Ubícate en la ventana *Desensamblador*, activa en ella la opción *Mostrar bytes de código* y comprueba que la codificación que has calculado para las instrucciones anteriores coincide con la utilizada por el compilador.

## 3. Ejercicios adicionales

✏ En este ejercicio se propone realizar una modificación en el programa `1-9prog1.c`. Se trata de introducir en la sección de datos de este programa un *array* de enteros. Para ello, justo después de la definición de las variables `A` y `B`, realiza la definición de un *array* de 4 enteros que se llame `vector`, y en la misma definición inicializa sus elementos con los valores 10, 11, 12 y 13. Después, dentro de `main` y a continuación de la sentencia "A = B", introduce una pareja de sentencias que carguen el elemento 0 del *array* con el valor -10, y el elemento 2, con el valor -12.

Obtén el ejecutable de esta nueva versión del programa y comienza la depuración pulsando F11. Limpia la ventana *Inspección 1* pulsando sobre ella con el botón derecho del ratón y eligiendo la opción *Borrar todo*. Vamos a localizar de nuevo la posición de las variables de la sección de datos. Esta posición puede variar de una compilación a otra de un programa. Obtén utilizando la ventana *Inspección 1* la dirección de la variable `A`. Toma nota de ella [15]. Posiciona la ventana *Memoria 1* en la dirección de esta variable. ¿Ves en la memoria las variables `A` y `B` y los cuatro elementos del *array*? Si tienes alguna duda, pregúntale a tu profesor. Ejecuta las sentencias que actualizan los elementos 0 y 2 del *array* con valores negativos, observando en la ventana *Memoria 2* cómo se modifican las posiciones de memoria correspondientes.

[15]

Veamos ahora qué modos de direccionamiento ha utilizado el compilador para acceder a los elementos del *array*. Abre la ventana *Desensamblador*. Busca la sentencia que carga el valor -12 en el elemento 2 del *array*. ¿Qué instrucción ha utilizado el compilador para compilar esta sentencia? [16] Utilizando las transparencias de codificación, codifica esta instrucción [17]. Después, selecciona en la ventana *Desensamblador* la opción *Mostrar bytes de código* y comprueba tu respuesta. Fíjate cómo en las instrucciones que utilizan el direccionamiento directo a memoria, la dirección a la que accede la instrucción va codificada en la propia instrucción (en el campo `Desplazamiento` del código de la instrucción.)

[16]

[17]

✏ En este ejercicio se propone la ejecución del programa `1-9prog2.c` con objeto de observar cómo evoluciona la pila durante su ejecución. Para ello, primero abre el proyecto correspondiente a este programa. La depuración la vamos a llevar a cabo en la ventana en la que se muestra el código fuente del programa.

Comienza la depuración pulsando F11. Configura la ventana *Memoria 2* para que muestre cuatro columnas. Recuerda que esta ventana es la que utilizamos para observar la pila. La cabecera de la pila está en la dirección apuntada por `ESP`. Posiciona la ventana *Memoria 2* en la dirección de `ESP` haciendo que dicha dirección quede en el límite inferior de la ventana. Así tendremos la ventana *Memoria 2* correctamente configurada para observar la pila. Toma nota del valor del registro `ESP` [18]. Pulsa ahora F11. Esto hace que se ejecuten todas las instrucciones estándar de entrada en la función `main`. Toma nota del nuevo valor que alcanza el registro `ESP` [19]. Observarás que este registro se ha desplazado sustancialmente. Ten en cuenta que, entre otras cosas, se reserva espacio para el *buffer* de datos temporales comentado anteriormente, y que

[18]

[19]

solo este *buffer* ocupa 40h bytes en la pila. La ventana *Memoria 2* ha quedado desfasada, reubícala para que de nuevo la cabecera de la pila quede en el límite inferior de la ventana.

Ahora vamos a la parte más interesante. En este punto toca ejecutar la sentencia "`A=cuadrado(4)`". Esto significa que hay que llamar a la función `cuadrado` pasándole el parámetro 4. Para ello el programa pone en la pila el parámetro y, después, hace la llamada, lo que implica que se coloca también en la pila la dirección de retorno. El parámetro que se colocará en la pila es claramente un 4. Para ver la dirección de retorno, debes abrir la ventana *Desensamblador* y en la función `main` buscar la instrucción siguiente a la que provoca la llamada a `cuadrado`. Dicha instrucción es la que destruye el parámetro. La dirección de esta instrucción es la dirección de retorno de `cuadrado`, anótala [20]. Ahora vuelve a la ventana de código fuente y mientras observas la ventana *Memoria 2*, ejecuta la sentencia "`A=cuadrado(4)`". Debes observar cómo se introducen en la pila el parámetro y la dirección de retorno.

20

Ahora la flecha amarilla se encuentra dentro de la función `cuadrado`. Ejecuta paso a paso hasta que llegues a la sentencia "`cuad = par*par`", pero no ejecutes esta sentencia todavía. Esta sentencia carga en la variable local `cuad` el cuadrado del parámetro. ¿En qué dirección se encuentra esta variable local? [21] Recuerda que entre la dirección de retorno y las variables locales solo se encuentra la salvaguarda del registro `EBP`, tal y como se representa en la figura 9.1. Localiza en la ventana *Memoria 2* la posición de la variable local, y entonces ejecuta la sentencia "`cuad = par*par`". Observa cómo se actualiza en la pila el contenido de la variable. Finalmente, detén la depuración.

21

Este ejercicio te ha permitido analizar cómo las funciones de C utilizan la pila para recibir parámetros y para almacenar variables locales.

APÉNDICE A

# Introducción al desarrollo de proyectos con Visual Studio

## 1. Desarrollo de programas ensamblador desde el entorno de programación

El ciclo de desarrollo de un programa consiste en un conjunto de fases: edición, compilación, enlazado y depuración. Estas fases se pueden llevar a cabo desde la línea de comandos. Sin embargo, el proceso de desarrollo de un programa también se puede realizar integramente desde el entorno de desarrollo (*Visual Studio*) sin tener que usar la línea de comandos, lo que resulta más simple y más ágil.

El desarrollo de programas desde el *Visual Studio* se realiza a través de proyectos. Un proyecto no es más que un contenedor de ficheros fuente donde se almacena información sobre la forma de compilar y enlazar, así como todos los ficheros que se generan durante el desarrollo.

A continuación se van a describir los pasos necesarios para crear un proyecto que permita agregar un fichero fuente en ensamblador.

Sigue los pasos que aparecen a continuación para crear un nuevo proyecto que se llame `ProgMin` que contenga un fichero fuente denominado `ProgMin.asm`. Este fichero debe contener el siguiente código:

```
1 .386
2 .MODEL FLAT, stdcall
3 ExitProcess PROTO, :DWORD
4
5 .CODE
6 inicio:
7   push 0
8   call ExitProcess
9 END inicio
```

### 1.1. Pasos para crear un proyecto y agregar un fichero fuente

1. Ejecutar el *Visual Studio*. Debería ser accesible a través del menú de programas instalados.

2. Dentro del *Visual Studio* se debe crear un nuevo proyecto, para lo cual se debe pulsar en el menú *Archivo->Nuevo->Proyecto*.

3. En el dialogo que aparece se debe seleccionar *Win32* como tipo de proyecto y *Aplicación de consola Win32* como plantilla. Además, se debe indicar el nombre del proyecto y el directorio donde se va a guardar. Por último hay que desactivar la casilla *Crear directorio para la solución*. La configuración debe ser la que se muestra en la figura A.1 pero con el nombre de proyecto y directorio adecuados. Una vez terminado este proceso se debe pulsar *Aceptar*.

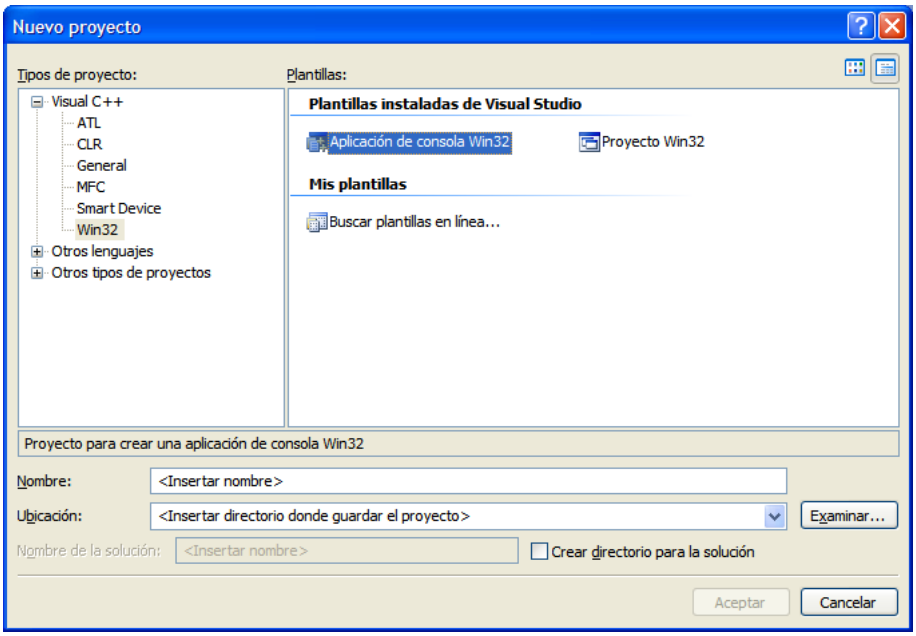


Figura A.1: Selección del tipo de proyecto

4. A continuación aparece la ventana del *Asistente*, el cual permite configurar ciertos aspectos de las aplicaciones que se van a desarrollar. Se debe pulsar *Siguiente* para que aparezca la *Configuración de la aplicación*. Sobre esta ventana, y dentro de las *Opciones adicionales*, se debe seleccionar *Proyecto vacío*. Esta ventana debe ser similar a la que se muestra en la figura A.2. Una vez terminado este proceso se debe pulsar *Finalizar*.

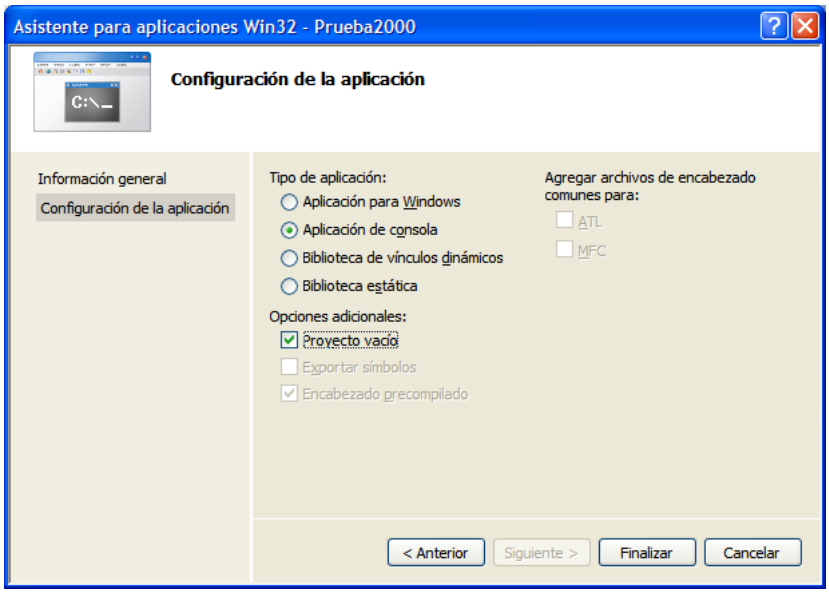


Figura A.2: Configuración del proyecto

5. Una vez que se ha creado el proyecto, aparecerá de nuevo la pantalla principal donde se podrá observar el *Explorador de soluciones*. El siguiente paso es pulsar con el botón derecho del ratón sobre el nombre del proyecto dentro del *Explorador de soluciones* y seleccionar *Reglas de generación personalizadas...*, tal y como aparece en la figura A.3.

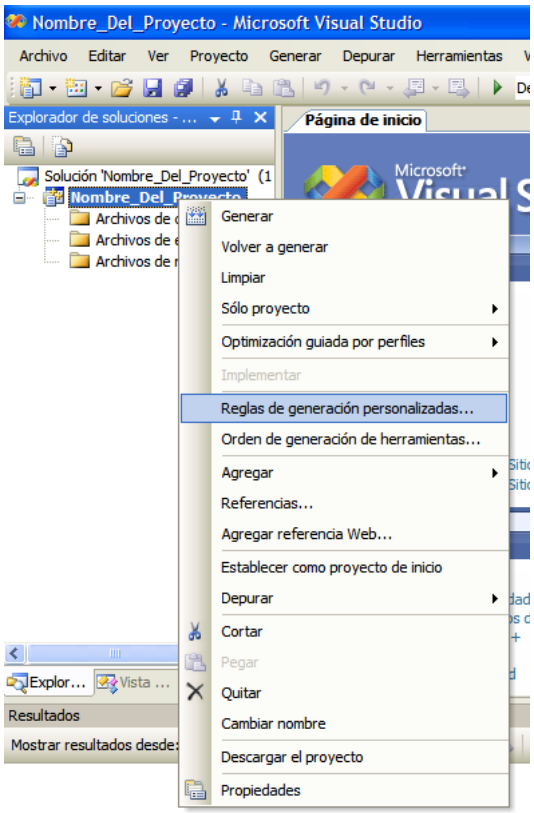


Figura A.3: Reglas de generación

6. En el diálogo que aparece se debe seleccionar *Microsoft Macro Assembler* y luego se debe pulsar *Aceptar*. El diálogo se muestra en la figura A.4

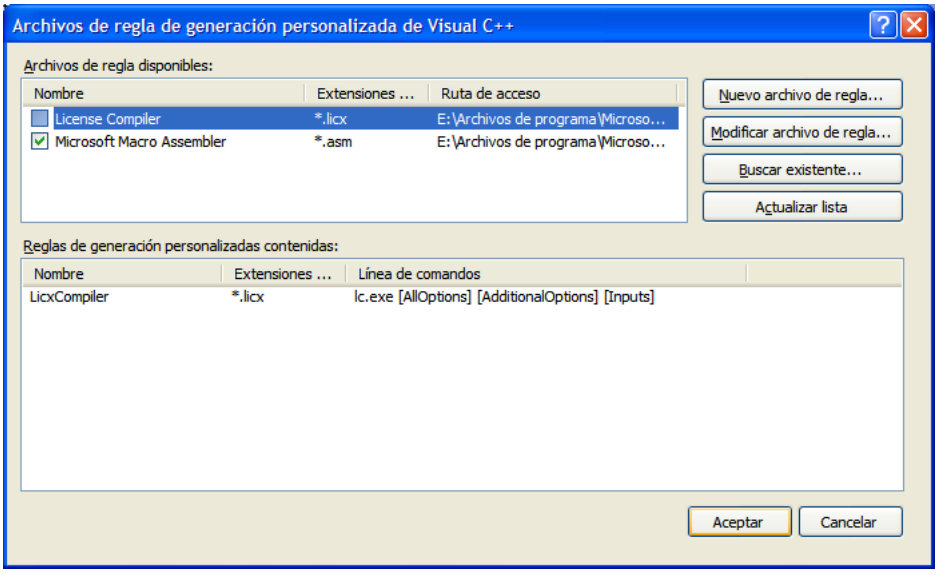


Figura A.4: Activación de la regla MASM

7. En este momento el proyecto ya está totalmente configurado, pero aún es necesario agregar un nuevo fichero fuente al proyecto. Para agregar un nuevo fichero fuente al proyecto se debe pulsar de nuevo con el botón derecho del ratón sobre el nombre del proyecto, y seleccionar *Agregar -> Nuevo elemento*.

8. En esta ventana se debe seleccionar *Código* dentro de las categorías y *Archivo C++ (.cpp)* como plantilla (no hay plantillas específicas para ensamblador, pero las de C++ funcionan). Además se debe indicar el nombre del fichero que se desea agregar al proyecto. Es importante acordarse de añadir la extensión .asm al fichero que se agregue. La ventana debe ser similar a la figura A.5.

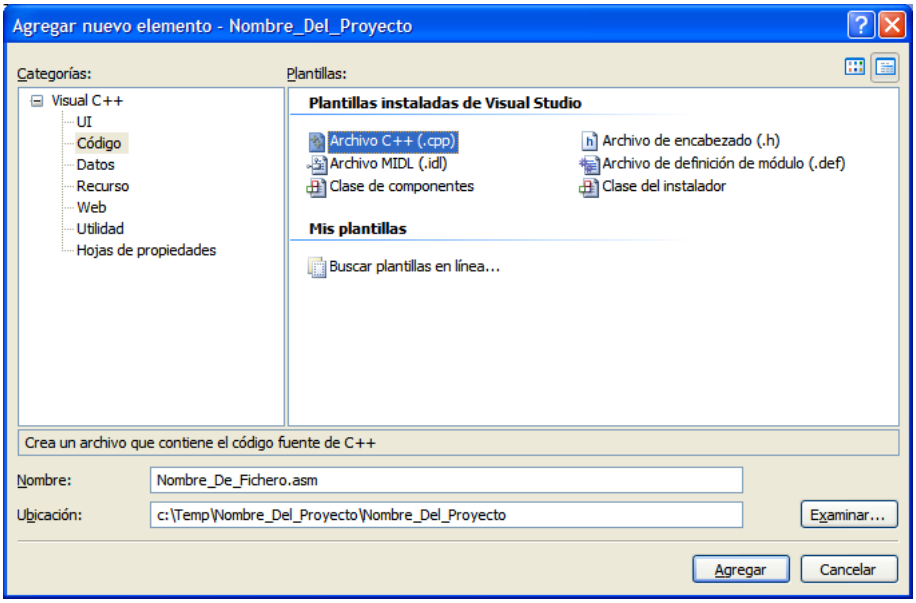


Figura A.5: Agregar un nuevo elemento al proyecto

9. Por último aparecerá el editor donde se puede empezar a escribir el código fuente del programa ensamblador.

10. Una vez escrito el programa se puede compilar, enlazar y depurar. Estas acciones pueden ser realizadas desde el menú, desde la barra de herramientas o través de teclas, siendo esta última la forma más rápida y por tanto la recomendada. Se puede compilar con CTRL-F7. Si hay errores de compilación aparecerán en la ventana inferior. Haciendo doble clic sobre el error, el editor se posicionará sobre la línea en la que dicho error se ha producido. Una vez solucionados los posibles errores, se puede enlazar el programa con F7 lo que creará el programa ejecutable. La depuración comenzará pulsando F11. Para simplificar este proceso se puede simplemente pulsar F11 lo cual provoca que se compile el programa, se enlace y finalmente comience la depuración, todo con la pulsación de una sola tecla. Para detener la depuración se puede pulsar SHIFT-F5.

Los ficheros del proyecto se almacenan en un directorio cuyo nombre coincide con el nombre del proyecto. Dentro de este directorio se crea un subdirectorio denominado *Debug* donde se almacenarán los ficheros objeto, el programa ejecutable y la información de depuración. El directorio *Debug* se puede eliminar para que no ocupe espacio tras terminar la realización de un programa ya que es generado cada vez que se compila.

## 1.2. Ciclo de edición, compilación, enlazado y depuración desde el entorno de desarrollo

La gran ventaja de utilizar el *Visual Studio* para desarrollar los programa es que todas las herramientas necesarias se encuentran integradas dentro del entorno. Esto hace que las repetidas fases de edición y compilación ante la presencia de errores se vean enormemente simplificadas.

A modo de ejemplo modifica el programa anterior añadiendo las siguientes instrucciones con errores de sintaxis a continuación de la etiqueta *inicio*:

```
xor eax,
mov al, 123h
```

Además sustituye `ExitProcess` por `ExitProcessX` en los dos sitios donde aparece en el programa.

Una vez modificado, intenta compilar el programa pulsando CTRL-F7. En la parte inferior de la pantalla aparecerá el resultado de la compilación, que debe ser similar al que aparece en la figura A.6.

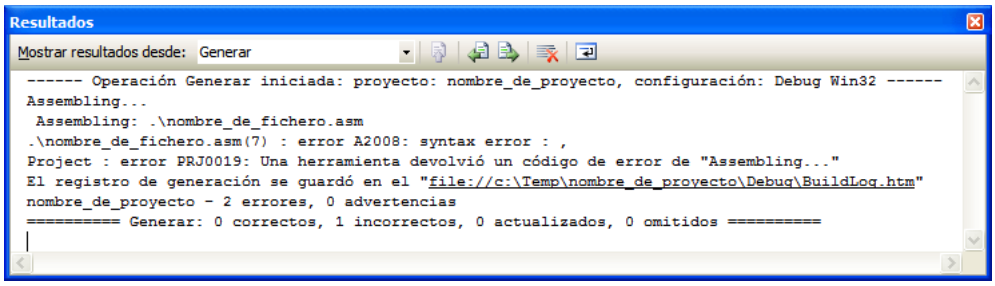


Figura A.6: Ventana de resultados con errores de compilación

Para solucionar el error se debe hacer doble click sobre la línea de la ventana de resultados donde aparece el error. Esto situará el editor justo en la línea del fichero fuente donde se produjo el error. Es importante leer el error que se muestra en la ventana de resultados, ya que dará pistas sobre cómo solucionarlo.

En este caso, el error se produce porque la instrucción `xor` necesita dos operandos. Para solucionar el error añade `eax` tras la coma de la instrucción errónea.

Una vez que se considera que se ha solucionado el error se debe proceder a compilar de nuevo con CTRL-F7. A continuación se debe revisar la ventana de resultados para ver si ha habido nuevos errores. En este caso debe aparecer otro error ya que no se puede mover el valor `123h` a un registro de tamaño byte. Para solucionarlo cambia el número por el `12h`.

Compila de nuevo el programa. Revisa la ventana de resultados, ahora no deberían aparecer errores.

Una vez que la compilación es correcta se puede proceder a enlazar el código objeto del fichero fuente, obtenido como resultado de la compilación, con las librerías. El enlazado desde el entorno se hace pulsando F7. De nuevo en la ventana de resultados aparecerán los posibles errores. Se puede observar cómo el proceso de enlazado ha producido un error ya que la función `ExitProcessX` no ha sido encontrada en ninguna de la librerías con las que

se ha enlazado. Soluciónalo eliminando la `X` del nombre de la función, compila y enlaza de nuevo.

Si todo hay ido bien y no han aparecido errores se puede comenzar el proceso de depuración con F11. Finalmente detén la depuración con SHIFT-F5.

Elimina las dos instrucciones añadidas para dejar al programa en su estado original y abandona el entorno de desarrollo.

A continuación abre el *Explorador de windows* y selecciona el directorio donde has almacenado el proyecto. Podrás observar una carpeta con el nombre del proyecto, si entras aparecerán un conjunto de archivos y una carpeta denominada *Debug*. El fichero principal donde se almacena la información con el proyecto es el que tiene extensión *.sln*. Haciendo doble click sobre ese fichero se abrirá el proyecto. También se puede observar el fichero fuente *.asm*. Es importante darse cuenta de que para poder comenzar nuevamente el ciclo de desarrollo sobre un programa hay que abrir el proyecto, no es suficiente con abrir el fichero fuente, siendo por tanto obligatorio abrir el fichero con exténsión *.sln*, preferiblemente haciendo doble click. Entrando en la carpeta *Debug* se pueden observar los ficheros objeto (extensión *.obj*) y el fichero ejecutable (extensión *.exe*) entre otros. La carpeta *Debug* se puede eliminar para ahorrar espacio ya que se regenera cada vez que se compila y enlaza.

## 1.3. Línea de comandos usada en la compilación y en el enlazado

La ventaja del entorno de desarrollo es que tiene memorizadas las líneas de comandos necesarias para compilar y enlazar, las cuales son ejecutadas cuando se pulsan las teclas correspondientes. Esto implica que cuando se trabaja desde el entorno de programación no es necesario abrir el interprete de comandos.

Para observar la línea de comandos que se utiliza al compilar, en el explorador de soluciones pulsa con el botón derecho sobre el fichero fuente y abre sus propiedades. La ventana que aparece se puede observar en la figura A.7.

De igual forma, se puede observar la línea de comandos que se utiliza para enlazar pulsando con el botón derecho sobre el proyecto (dentro del explorador de soluciones) y abriendo sus propiedades. La ventana que aparece se puede observar en la figura A.8. Como se puede observar, en el la línea de comandos de enlazado se incluyen las librerías más comunmente utilizadas, como la `kernel32.lib`.

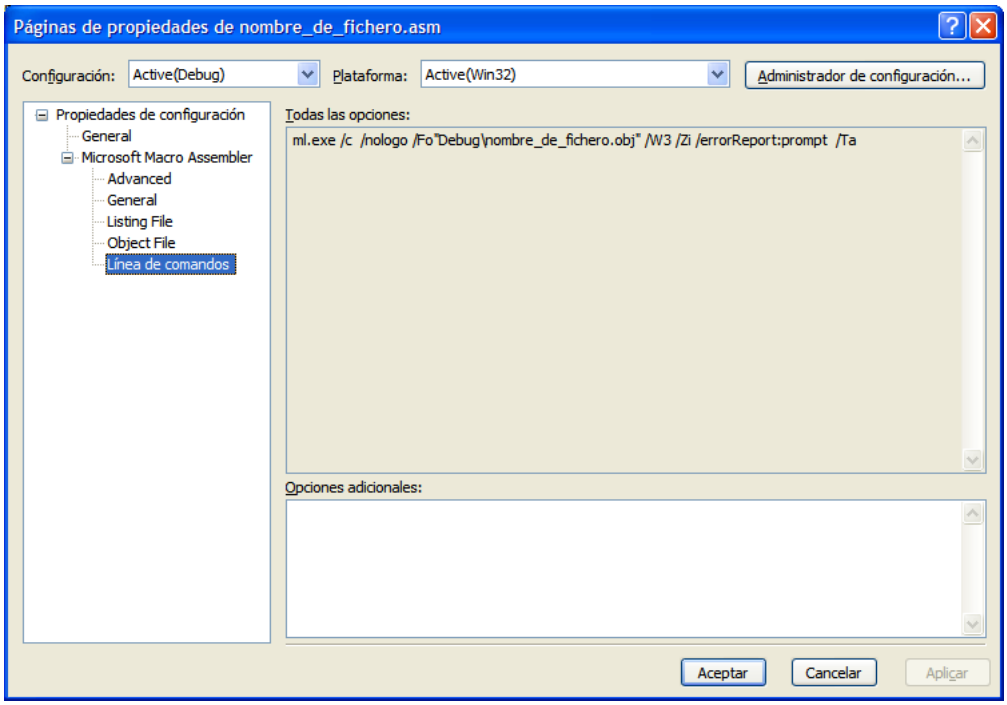


Figura A.7: Linea de comandos usada por el *Visual Studio* para compilar

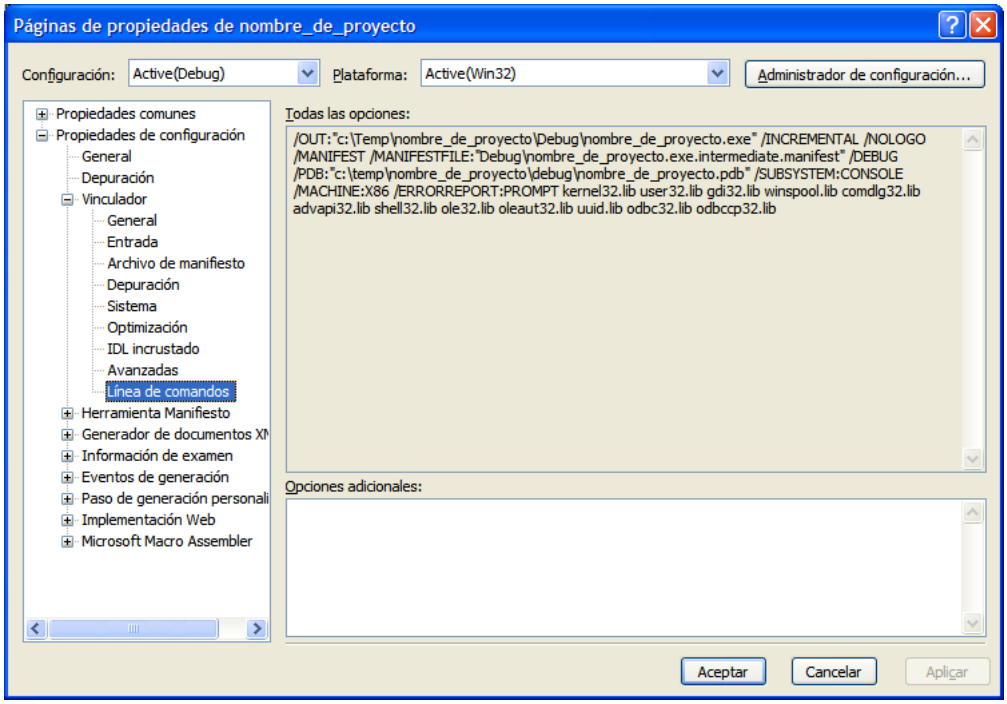


Figura A.8: Linea de comandos usada por el *Visual Studio* para enlazar

APÉNDICE B

# Desarrollo en C con Visual Studio

## 1. Diferencias en el desarrollo en C frente a ensamblador

Se pueden crear muchos tipos de proyectos diferentes orientados al desarrollo en lenguaje C. Dependiendo del tipo de proyecto que elijamos, el entorno de desarrollo generará automáticamente ciertos fragmentos de código, agregará determinados componentes al proyecto, etc. En nuestro caso, queremos utilizar el entorno de desarrollo para algo más sencillo, concretamente para editar y compilar programas C simples, sin ningún tipo de código ni componente añadido. Para hacer esto, los pasos a seguir son muy similares a los indicados en el apéndice A, en el que se muestra cómo editar y compilar programas escritos en lenguaje ensamblador. En este apéndice veremos los pasos a seguir para editar y compilar un programa C, indicándose las diferencias que existen con el desarrollo de programas escritos en lenguaje ensamblador.

Para ilustrar los pasos a seguir, utilizaremos un ejemplo de programa C mínimo. El listado de este programa se indica a continuación:

```
1
2 int A=1;  // Variable global
3
4 main()
5 {
6         A++;
7 }
8
```

Para editar este programa y obtener su versión ejecutable con el Visual Studio debes seguir los siguientes pasos.

- ❏ Abre el Visual Studio.
- ❏ Ahora debes crear el proyecto. Para ello en el menú *Archivo* elige las opciones *Nuevo->Proyecto*. Se muestra entonces la ventana de diálogo *Nuevo proyecto*. En esta ventana debes seleccionar *Win32* como tipo de proyecto y *Aplicación de consola Win32* como plantilla. Además debes indicar el nombre del proyecto y el directorio donde se va a guardar. Como nombre de proyecto utiliza `ProgMinC`. Por último hay que desactivar la casilla *Crear directorio para la solución*.

- ❏ A continuación se abre el *Asistente para aplicaciones Win32*. Debes pulsar *Siguiente* para que aparezca la *Configuración de la aplicación*. En *Tipo de aplicación* debes elegir *Aplicación de consola* y en *Opciones adicionales*, selecciona *Proyecto vacío*. Tras pulsar en *Finalizar*, se generará el proyecto, que podrás observar en el *Explorador de soluciones*.

Hasta aquí todo igual que lo que llevabas a cabo para desarrollar programas en lenguaje ensamblador. Ahora debe señalarse la primera diferencia. Cuando el proyecto iba orientado a lenguaje ensamblador había que indicarle al Visual Studio que utilizase el compilador de ensamblador para compilar los ficheros del proyecto. Esto se hacía con la opción *Reglas de generación personalizadas*, a la que se accede pulsando con el botón derecho del ratón sobre el nombre del proyecto. Esto ahora no debe hacerse, ya que los proyectos que creamos pertenecen a la categoría de proyectos *Visual C++*. El Visual utiliza por defecto el compilador de C/C++ para compilar los ficheros de estos proyectos, que es lo que queremos cuando desarrollamos programas C. Por consiguiente en este punto, el proyecto ya está totalmente configurado.

- ❏ Ahora hay que agregar al proyecto el fichero de código fuente que contendrá el código del programa C que queremos generar. Para agregar un fichero fuente al proyecto se debe pulsar de nuevo con el botón derecho del ratón sobre el nombre del proyecto y seleccionar *Agregar->Nuevo elemento*. En la ventana que se abre, seleccionar *Código* dentro de las categorías y *Archivo C++(.cpp)* como plantilla. Además se debe indicar el nombre del fichero que se desea agregar al proyecto. Aquí está la otra diferencia importante respecto al desarrollo en lenguaje ensamblador. Ahora tienes que indicar que el fichero es C, por consiguiente debes añadir la extensión `.c` al fichero. Es importante no olvidarse de añadir la extensión, ya que si no, el entorno crearía un fichero de tipo `.cpp`, que es parecido a un fichero C, pero no es lo mismo. Entonces agrega al proyecto el fichero `ProgMin.C`.
- ❏ Se abre el editor de código fuente. Escribe entonces el código del programa C mínimo indicado anteriormente.
- ❏ Una vez escrito el código fuente del programa C, éste se compila, enlaza y depura exactamente igual que los programas ensamblador que has realizado hasta este momento. Compila pulsando CTRL-F7. Aquí debe indicarse que aunque el usuario compila de la misma forma, sea el programa C o ensamblador, el entorno utilizará el compilador adecuado, en función del tipo de programa que se esté compilando. En este caso utilizará el compilador de C, porque está compilando un archivo que tiene código fuente C. Ahora enlaza mediante F7, y comienza la depuración pulsando F11. Si hubiese errores, se corregirían de la misma forma que cuando desarrollabas en lenguaje ensamblador.
- ❏ Detén la depuración y abandona el Visual Studio.

Mediante este apéndice habrás podido observar que el desarrollo de programas C simples mediante el Visual Studio resulta muy similar al desarrollo de programas ensamblador. No obstante, deben tenerse en cuenta las diferencias indicadas en este apéndice respecto al desarrollo en ambos lenguajes.