Bloque 6

Programación en ensamblador de la arquitectura x86-32 bajo Win32

Prácticas de
Fundamentos de Computadores
Curso 2009-2010

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 realización 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.
- Disponer de 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 una serie de pasos, mostrados 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.

- 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 entorno de desarrollo Microsoft Visual Studio.

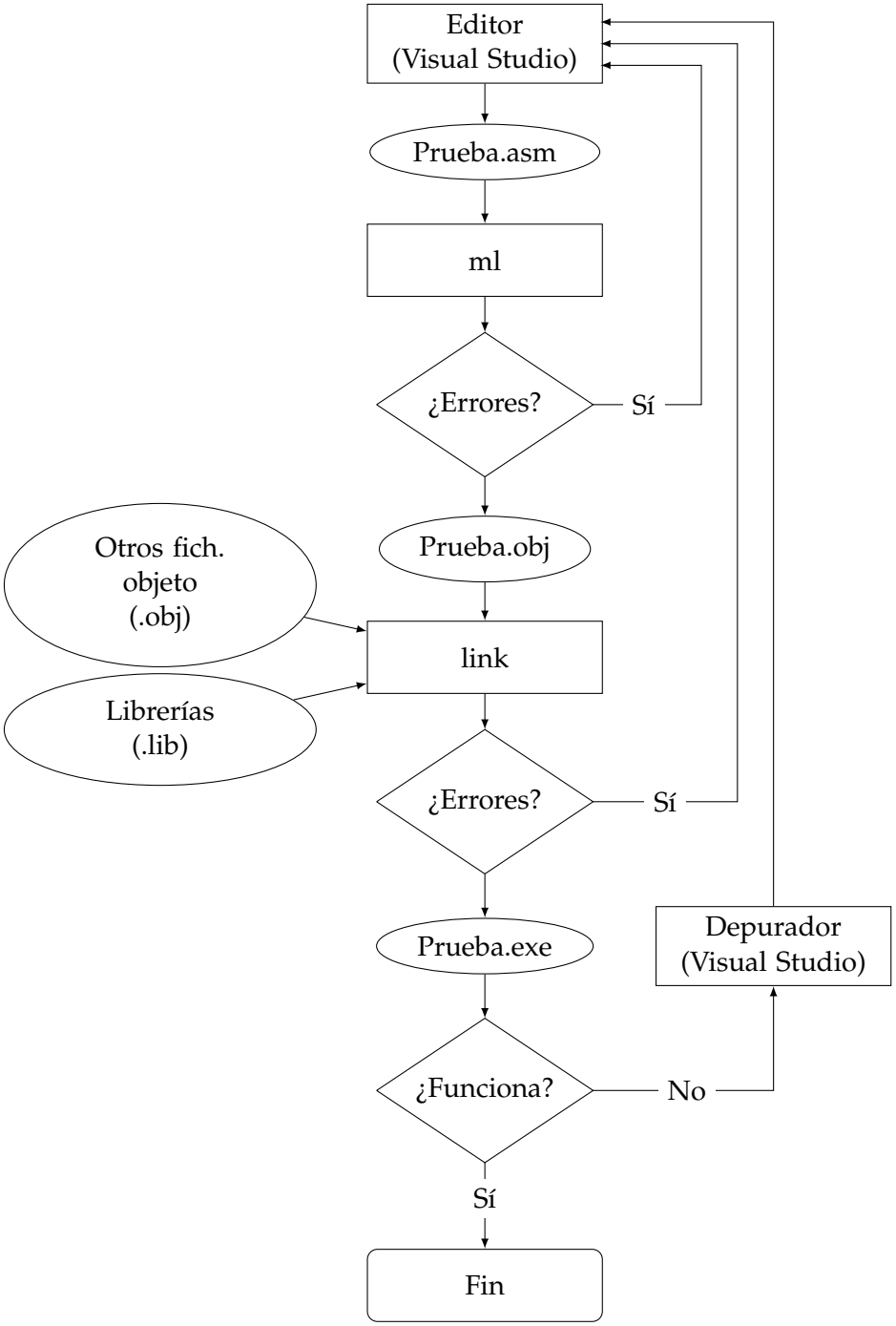

Figura 1.1: Ciclo de desarrollo de un programa en ensamblador

- 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.
- Obtener el *archivo ejecutable*, enlazando el código objeto del programa con las librerías necesarias. Esto lo hace el *montador de enlaces*, *enlazador* 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.

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.

- ❏ En el fichero (`6-1prog1.asm`) está disponible la mayor parte del fichero fuente que utilizarás . 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 enlazador (*link*) del Visual Studio. Para arrancar la interfaz de comandos abre el menú *inicio*, *programas*, *Microsoft Visual Studio 2008*, *Visual Studio Tools* y entonces elige la opción *Símbolo del sistema de Visual Studio 2008*.
- ❏ 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 6-1prog1.asm
  ```

  La salida que obtendrás será:

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

  Assembling: 6-1prog1.asm
  6-1prog1.asm(15) : error A2008: syntax error : in instruction
  6-1prog1.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. 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, `6-1prog1.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 su correcta ejecución. 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, nosotros no hemos escrito el código del procedimiento `ExitProcess`, aunque se utilice. En realidad su código se encuentra en otro fichero, denominado `kernel32.lib`, el cual pertenece a una categoría de ficheros conocidos como *librerías*. Entonces el código de nuestro programa (`6-1prog1.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 *vinculador o enlazador*. Esta herramienta "enlaza" (une) el código escrito por nosotros con el código de `ExitProcess` disponible en la librería y genera una versión completa y ejecutable de nuestro programa, que se denominará `6-1prog1.exe`.

- ❏ Genera la versión ejecutable del programa tecleando el siguiente comando:
  `link /SUBSYSTEM:CONSOLE 6-1prog1.obj kernel32.lib`
  Tras su ejecución, comprueba que se ha generado el fichero `6-1prog1.exe`, ejecutando el comando *DIR*.
- ❏ Puedes ordenar al sistema operativo la ejecución de `6-1prog1.exe`; sin embargo, al ejecutarlo no notarás efecto alguno, salvo que se retorna de nuevo al sistema operativo dado que el código de nuestro programa invoca al procedimiento `ExitProcess`. ya que no se hacen operaciones de Entrada/Salida (como la gran mayoría de los que se harán en este bloque de prácticas). Comprueba esto, tecleando `6-1prog1` 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 al enlazador). En concreto ejecuta los siguientes comandos:
  `ml /c /Cx /coff /Zi /Zd /Zf 6-1prog1.asm`
  `link /DEBUG /SUBSYSTEM:CONSOLE 6-1prog1.obj kernel32.lib`
- ❏ Ejecuta el comando *DIR*. Observa que además de los ficheros 6-1prog1.obj y 6-1prog1.exe, también se ha generado el fichero 6-1prog1.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.
- ❏ 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 interfaz de comandos el comando siguiente:
  `devenv /debugexe 6-1prog1.exe`

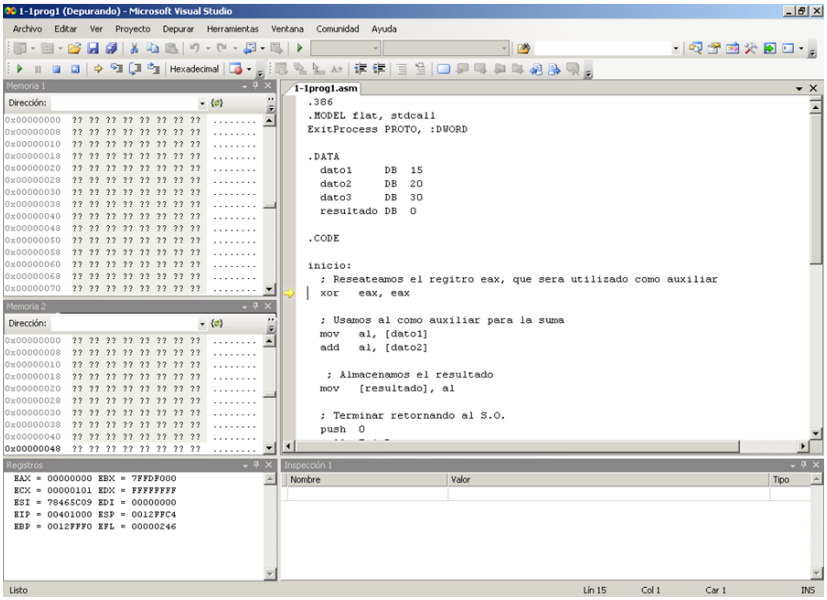

Figura 1.2: Entorno de depuración del Visual Studio

- ❏ Antes de continuar debes cargar una configuración que permite una depuración más cómoda de programas en ensamblador. Para ello debes realizar los pasos descritos en el apéndice A.
- ❏ 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 *Memoria 1* y *Memoria 2*. 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.[1]

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

[1]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.

mediante fichas. Cuando se comienza el proceso de depuración de un programa, se muestra una ficha con el código fuente del programa. En nuestro caso 6-1prog1.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 se 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 de memoria 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 seis y todas ellas empiezan por la palabra *mostrar*. De las seis opciones debes tener seleccionadas las cuatro 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 `00401006h`? [2]

1

2

- Ahora vamos a explorar el contenido de la sección de datos del programa. Para ello utilizaremos la ventana denominada *Memoria 1*, 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 enlazador del entorno de desarrollo genera el ejecutable para que los datos se ubiquen a partir de la dirección `00404000h`, por tanto, vamos a visualizar esta zona de la memoria. Para ello tendrás que introducir el valor `00404000h` en el campo *Dirección*. ¡Ojo! Para introducir valores hexadecimales en este campo debes precederlos de '`0x`'. Entonces introduce el valor `0x00404000` en el campo *Dirección*. ¿Puedes identificar ahora los

datos de tu programa en la ventana *Memoria 1*? ¿Cuál es el contenido de la dirección de memoria `00404001h`? [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 una sesión posterior 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 *Marcadores*. 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 seleccionada 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 del registro `EIP` después de ejecutar la primera instrucción del programa? [5] ¿Por qué? Pulsa F11 y comprueba que tu respuesta es correcta. Fíjate cómo 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 cómo el registro `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 *Memoria 1*. Antes de ejecutar la instrucción `mov [resultado], al`, contesta a la siguiente pregunta: ¿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 ejecuta 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 el procedimiento `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 6-1prog1.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 están disponibles 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 `compila.bat` y `enlaza.bat`. Para comprobar que hacemos esto correctamente, primero borra de tu carpeta de trabajo todos los ficheros excepto 6-1prog1.asm.
- ❏ Para volver a generarlos ejecuta:
  `compila 6-1prog1.asm`.
- ❏ Comprueba que se ha generado 6-1prog1.obj.
- ❏ Ahora ejecuta:
  `enlaza 6-1prog1.obj kernel32.lib`
- ❏ Comprueba que se ha generado 6-1prog1.exe.
- ❏ Ahora ejecuta `depura 6-1prog1.exe`. Pulsa entonces F11 para comenzar la depuración del programa. Termina la depuración.

En la siguiente sesión de este bloque de prácticas utilizarás los ficheros de comandos `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 6-1prog1.asm en otro fichero que se llame 6-1prog2.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 *Memoria 1* 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

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 B.

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 B, crea en tu carpeta de prácticas un proyecto nuevo llamado `6-2prog1`. Agrega al proyecto un nuevo fichero llamado `6-2prog1.asm`. Como el programa `ProgMin.asm` del apéndice B contiene el esqueleto básico de un programa, lo vamos a utilizar como punto de partida. Copia su contenido a `6-2prog1.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 datos almacenados en 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 una unidad el contenido del registro `ECX`. Además, si `ECX` no ha llegado a cero salta a la etiqueta que se le indique. 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 citada 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 colocar 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 la etiqueta lista `lista` más el contenido del 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 falta por decrementar en una unidad 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 se había 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. Genera 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 número -1 por el número 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 `6-2prog2`. Agrega al proyecto un nuevo fichero llamado `6-2prog2.asm` y copia en él el contenido del fichero `6-2prog1.asm`.
- ❏ Cambia el número 135 por el número -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]
- ❏ Genera 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. Genera el ejecutable del programa, ejecútalo paso a paso y comprueba su correcto funcionamiento.

✏ Modifica 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 3

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

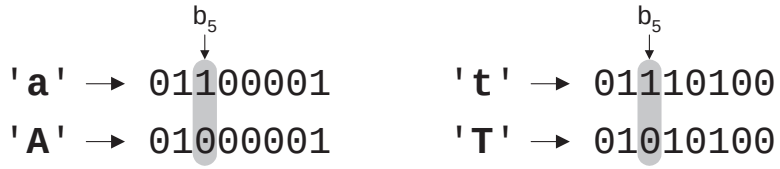

Figura 3.1: Diferencia de los códigos ASCII de mayúsculas y minúsculas en el bit de peso 5

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 3.1.

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 debe realizar 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 B de estas prácticas, crea en tu carpeta de prácticas un nuevo proyecto llamado `6-3prog1`. Agrega al proyecto un nuevo fichero llamado `6-3prog1.asm`. Abre alguno de los programas que hayas realizado anteriormente y cópialo en `6-3prog1.asm`. Entonces borra en este fichero todo lo necesario hasta dejar un esqueleto básico de programa. (También puedes utilizar el esqueleto del programa `ProgMin` creado al realizar el apéndice apéndice B.
- ❏ 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 3.2 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 (2*A*h). A la derecha se muestra el carácter asociado a este valor, que es el '*'.

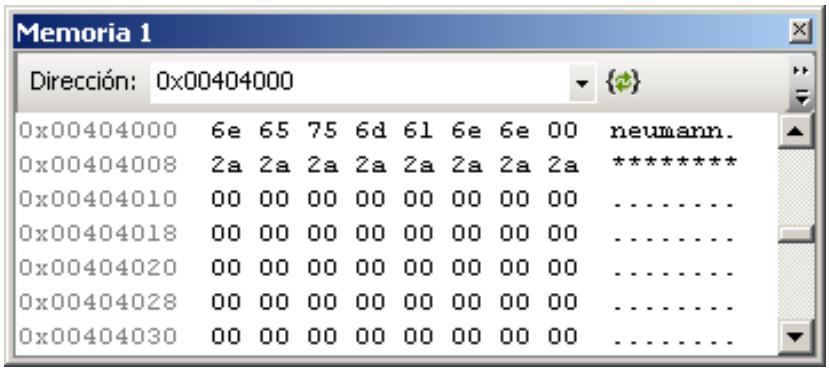

Figura 3.2: Estado de la sección de datos al comienzo del programa

❑ 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, lo debe convertir a mayúscula y, si no, no debe hacer 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 4

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 éstos aportan.

Conocimientos y materiales necesarios

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

- Conocer el juego de instrucciones resumido del lenguaje ensamblador para la arquitectura IA-32.
- Comprender el funcionamiento de la estructura de pila desarrollada.
- 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 introduce en la pila la dirección de la lista
15   ; (Utilizar sólo una instrucción)
16   ......
17
18   ; Se introduce 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 nú-

meros. 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 a través de 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 `6-4prog1.asm`. Dicho fichero contiene el listado del programa.

- ❏ Copia el fichero `6-4prog1.asm` en tu carpeta de prácticas.
- ❏ Crea un nuevo proyecto de la forma habitual con el nombre `6-4prog1` y agrega a él un nuevo fichero que se llame `6-4prog1.asm`.
- ❏ Copia el contenido del fichero `6-4prog1.asm` en tu fichero `6-4prog1.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 representa 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 `6-4prog1`. 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 *Memoria1* y *Memoria2*.

- ❏ Haz que la ventana *Memoria1* apunte a los datos del programa que se encuentra en la dirección 00404000h. Antes de observar el contenido de la memoria, contesta a la siguiente pregunta: ¿en qué dirección se ubica el dato -1 de la lista `datos`? [1]. Contrasta tu respuesta buscando este dato en la ventana *Memoria1*.

  1

- ❏ Configura la ventana *Memoria2* para que visualice los datos en 4 columnas y para que la cabecera de la pila quede en su límite inferior.
- ❏ 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]

  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 *Memoria2*. 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.

  3

- ❏ Ejecuta la instrucción que introduce el segundo parámetro en la pila. Observa el parámetro en la ventana *Memoria2*. Actualiza tu esquema de pila.
- ❏ 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.

  4

  5

  6

- ❏ 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`.

  7

- ❏ 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.

  8

- ❏ Ejecuta las instrucciones que siguen en el procedimiento, incluido el bucle, hasta que llegues a la instrucción `pop esi`, pero no ejecutes todavía esta 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 *Memoria2* 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 `6-4prog2` y agrégale un nuevo fichero llamado `6-4prog2.asm`. Copia en este fichero el programa anterior, es decir, `6-4prog1.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 `6-4prog1.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 se debe 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]
- ❏ 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 *Memoria2* 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 *Memoria2*.
- ❏ 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 *Memoria2* 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.

10

2.2. Paso de parámetros por referencia

Nuestro objetivo ahora será modificar el programa anterior (`6-4prog2.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 `6-4prog3` y agrégale un nuevo fichero llamado `6-4prog3.asm`. Copia en este fichero el programa anterior, es decir, `6-4prog2.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 `6-4prog3.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 `6-4prog3`. 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. 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 `6-4prog4` y agrégale el fichero `6-4prog4.asm`. Copia en este fichero el código de `6-4prog1.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`.

APÉNDICE A

Configuración del Visual Studio

1. Carga del fichero de configuración

El Visual Studio es un entorno de desarrollo diseñado para diversos lenguajes de programación que posee múltiples ventanas para controlar distintos aspectos del desarrollo o de la depuración de los programas. Durante el desarrollo con el Visual Studio es relativamente fácil cerrar o mover accidentalmente alguna de estas ventanas, pudiendo ser difícil volver a la situación inicial. Para evitar este problema se va a proporcionar un fichero de configuración donde está almacenada información sobre qué ventanas deben estar abiertas y en qué posición.

Para cargar el fichero de configuración, se debe pulsar sobre el menú *Herramientas* la opción *Importar o exportar configuraciones*. En el diálogo que aparece se debe seleccionar *Importar la configuración del entorno seleccionada* y luego pulsar sobre el botón *Siguiente*. A continuación se debe seleccionar que no se quiere guardar la configuración actual, y de nuevo se pulsa en el botón *Siguiente*. En la siguiente pantalla se de pulsar sobre el botón *Examinar*, que abrirá un diálogo de selección de archivo, para luego seleccionar el archivo `MacroAssembler.vssettings` (consulta a tu profesor para que te diga cuál es su ubicación). Por ultimo pulsa *Siguiente*, *Finalizar*, y cuando haya terminado de cargar la configuración pulsa *Cerrar*.

La carga del fichero de configuración se debe realizar tras ejecutar por primera vez el Visual Studio, así como siempre que se quiera restaurar la configuración.

APÉNDICE B

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 entorno *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 B.1 pero con el nombre de proyecto y directorio adecuados. Una vez terminado este proceso se debe pulsar *Aceptar*.

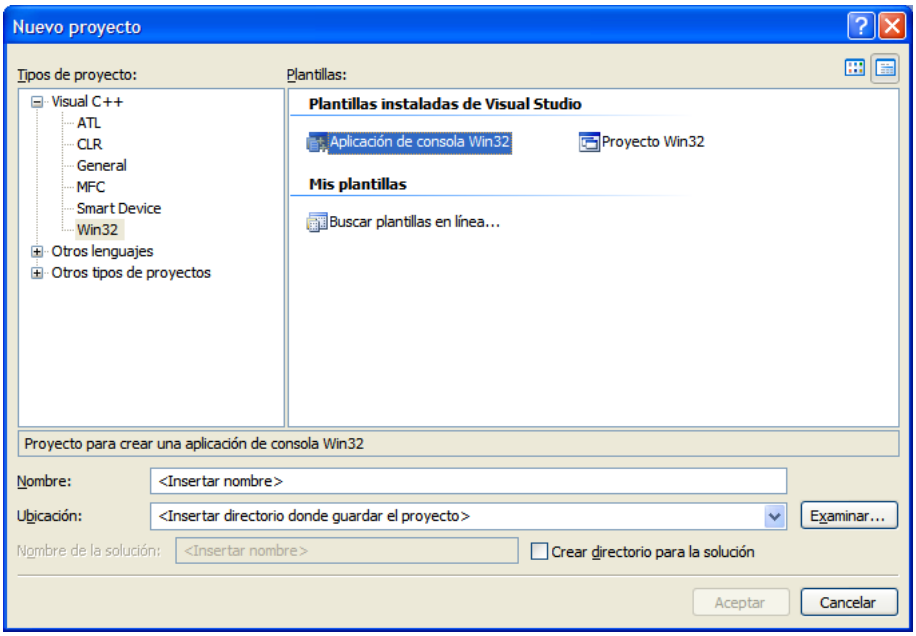

Figura B.1: Selección del tipo de proyecto

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 B.2. Una vez terminado este proceso se debe pulsar *Finalizar*.

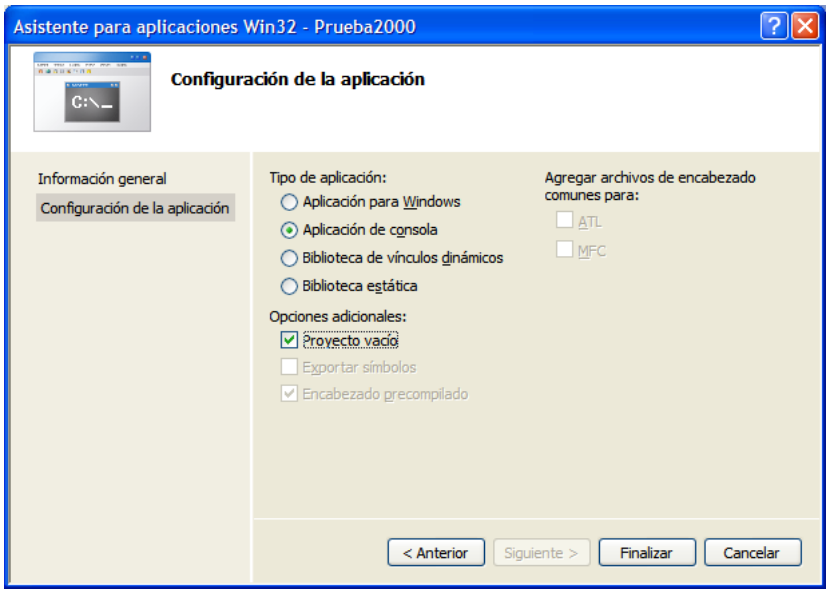

Figura B.2: Configuración del proyecto

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...*, como se muestra en la figura B.3.

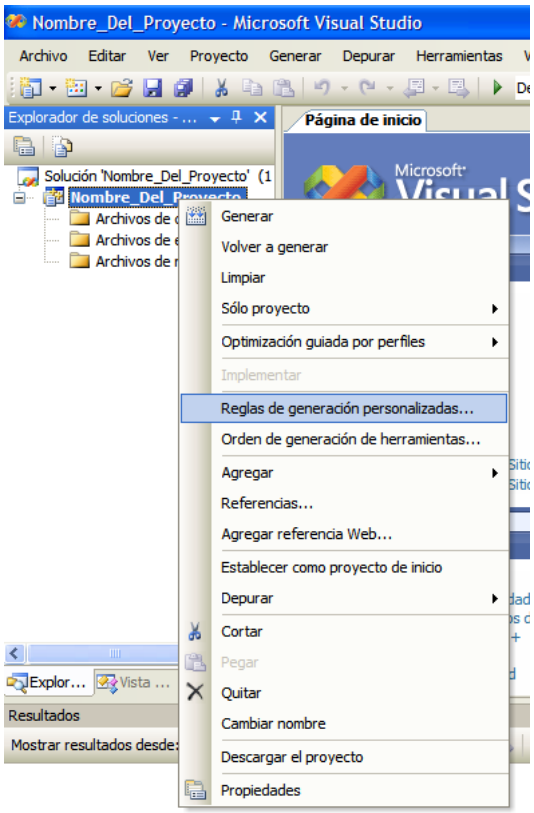

Figura B.3: Reglas de generación

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 B.4

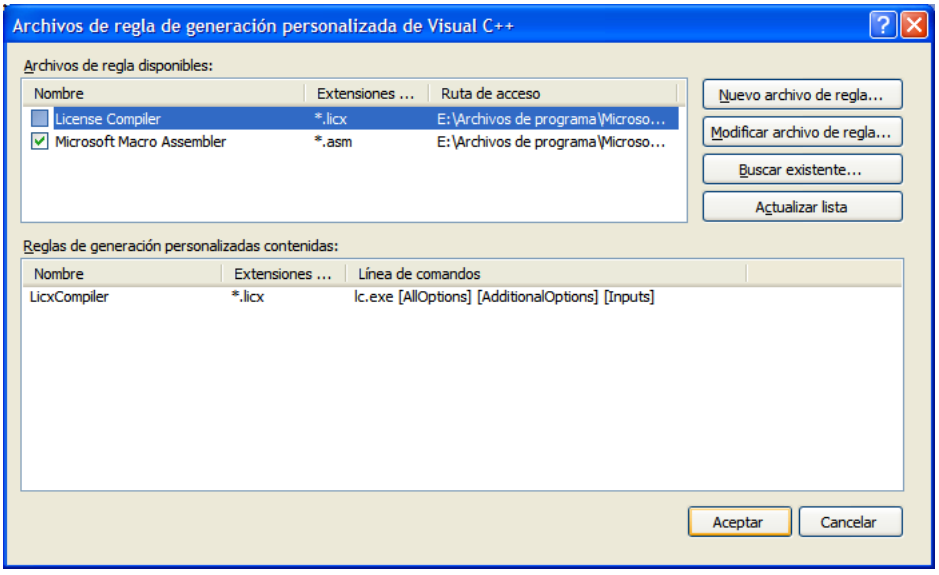

Figura B.4: Activación de la regla MASM

7. En este momento el proyecto ya está totalmente configurado, pero aún es necesario agregarle un nuevo fichero fuente. Para agregar un nuevo fichero fuente al proyecto se debe pulsar 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 B.5.

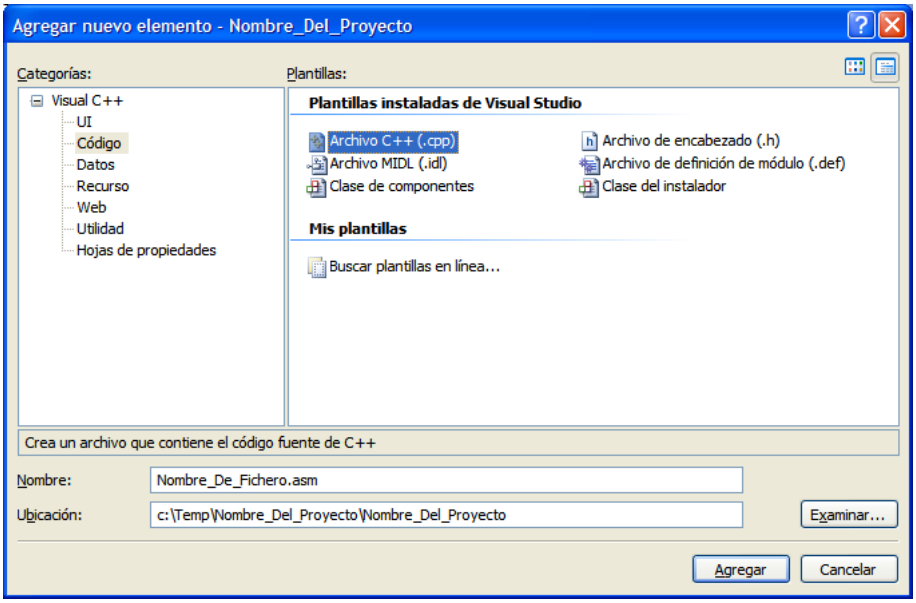

Figura B.5: Agregar un nuevo elemento al proyecto

9. Por último, aparecerá el editor para 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 entorno *Visual Studio* para desarrollar los programa es que todas las herramientas necesarias se encuentran integradas dentro del mismo. 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 B.6.

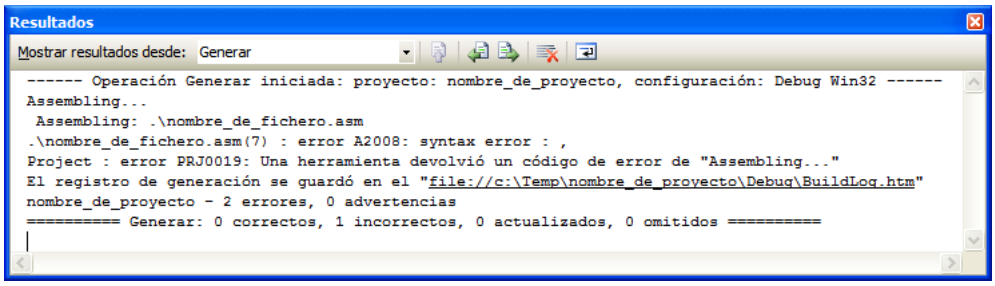

Figura B.6: Ventana de resultados con errores de compilación

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. El proceso de depuración se detiene 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 extensión *.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. Como se ha comentado anteriormente, 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 B.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 B.8. Como se puede observar, en el la línea de comandos de enlazado se incluyen las librerías más comunmente utilizadas, como la `kernel32.lib`.

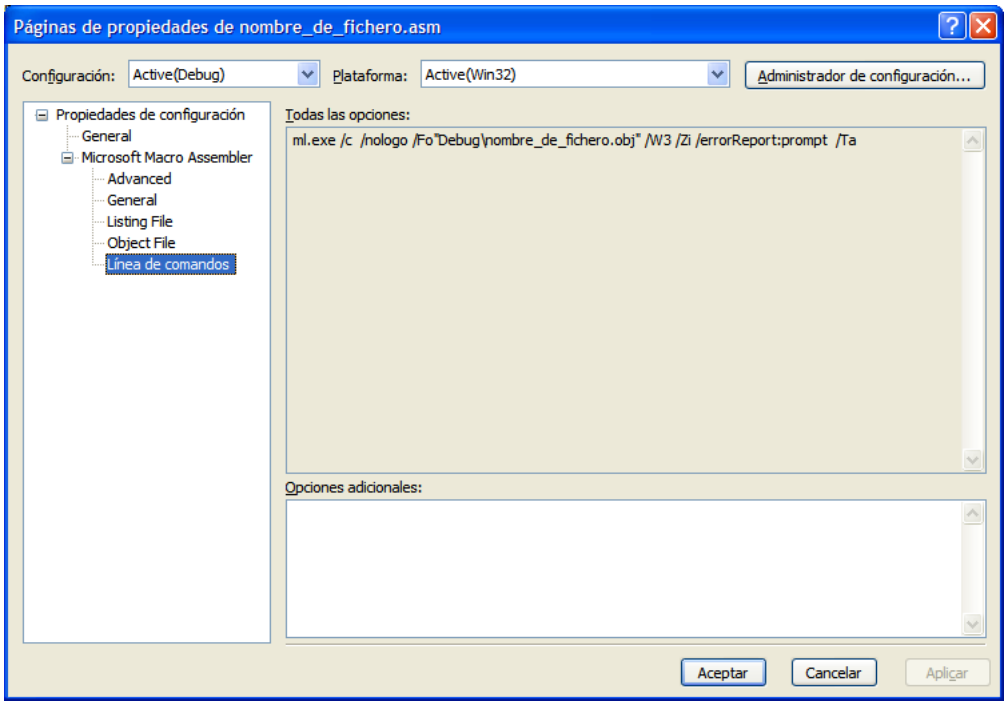

Figura B.7: Linea de comandos usada por el *Visual Studio* para compilar

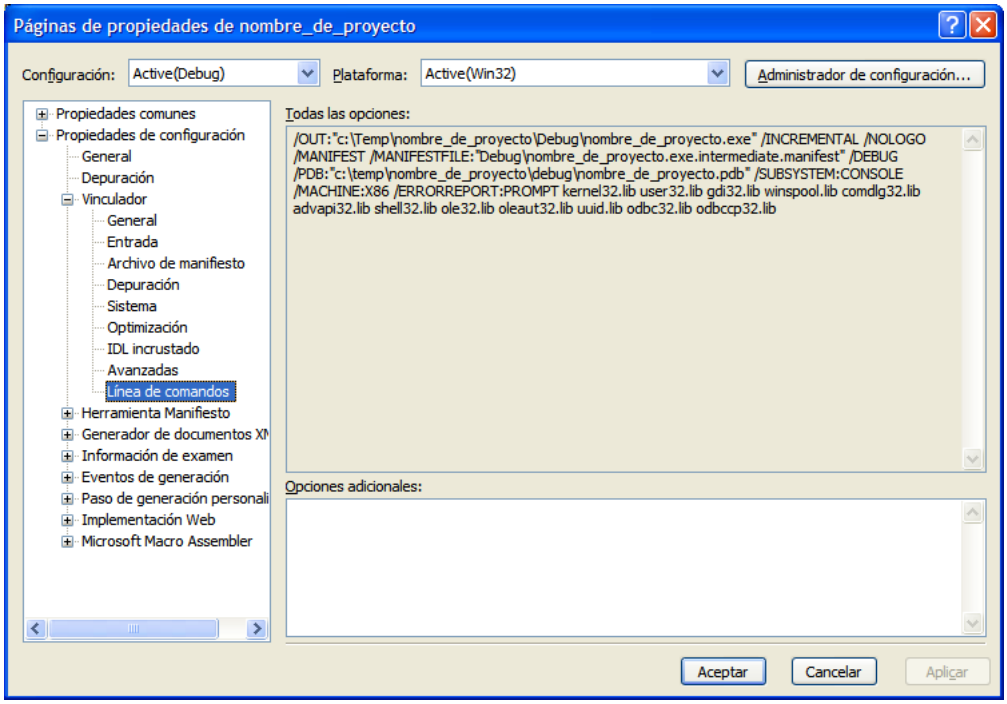

Figura B.8: Linea de comandos usada por el *Visual Studio* para enlazar