

Lab 2: Introduction to Assembly Language Programming

Contents

- 2.1. Intel IA-32 Processor Architecture
- 2.2. Basic Program Execution Registers
- 2.3. FLAT Memory Model and Protected-Address Mode
- 2.4. FLAT Memory Program Template
- 2.5. Writing a Program using the FLAT Memory Model
- 2.6. Editing, Assembling, Linking, and Debugging Programs

2.1 Intel IA-32 Processor Architecture

Intel introduced the 8086 processor in 1979. It has a 20-bit address bus and a 16-bit data bus. The maximum physical memory that this processor can access is 2^{20} bytes or 1MB. This processor uses segmentation to address memory. The maximum size of a segment is restricted to 2^{16} bytes or 64 KB, since all registers were 16 bits at that time.

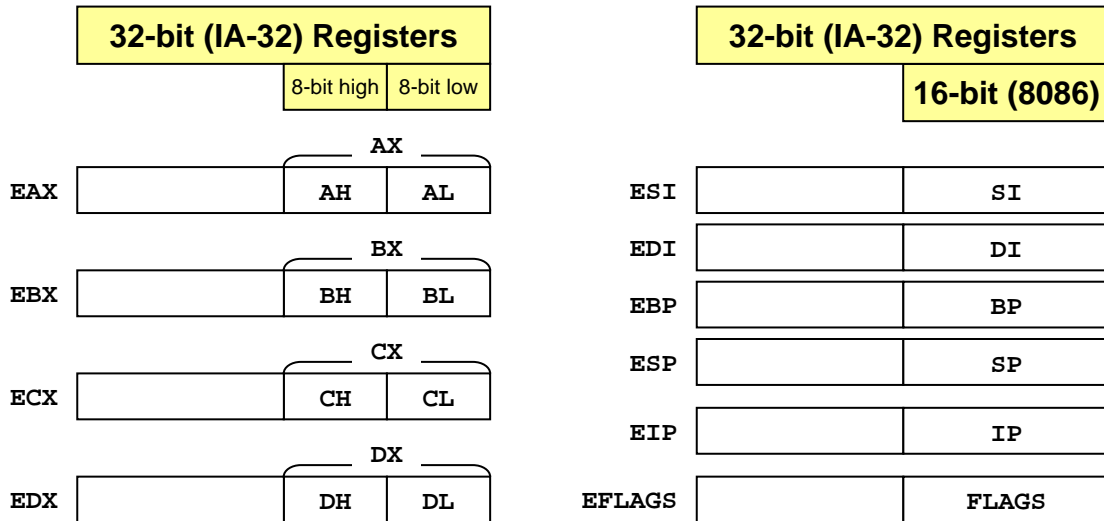
With the advent of 32-bit processors, starting with the 80386 in 1985, and continuing up with the latest Pentium family of processors, Intel has introduced a 32-bit architecture known as IA-32. This family of processors allows the use of 32-bit addresses that can address up to 4 Gigabytes of memory. These 32-bit processors remove the limitations of the earlier 16-bit 8086 processor.

2.2 Basic Program Execution Registers

There are eight 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), a 32-bit register that holds the processor flags (EFLAGS), and a 32-bit instruction pointer (EIP). These registers are shown below.

EAX	ESI
EBX	EDI
ECX	EBP
EDX	ESP
EFLAGS	EIP

The general-purpose registers are primarily used for arithmetic and data movement. Each register can be addressed as either a single 32-bit value or a 16-bit value. Some 16-bit registers (AX, BX, CX, and DX) can be also addressed as two separate 8-bit values. For example, AX can be addressed as AH and AL, as shown below.

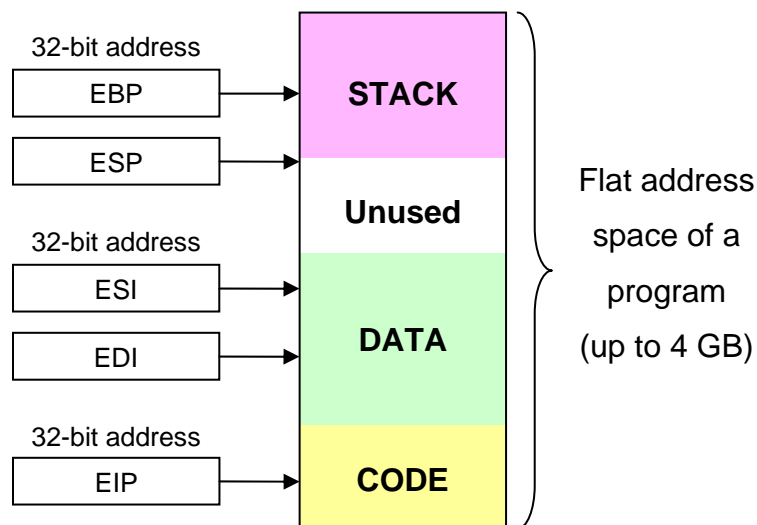


Some general-purpose registers have specialized uses. EAX is called the *extended accumulator* and is used by multiplication and division instructions. ECX is the *extended counter register* and is used as a loop counter. ESP is the *extended stack pointer* and is used to point to the top of the stack. EBP is the *extended base pointer* and is used to access function parameters and local variables on the stack. ESI is called the *extended source index* and EDI is called the *extended destination index*. They are used by string instructions. We will learn more about these instructions later during the semester.

The EIP register is called the *extended instruction pointer* and contains the address of the next instruction to be executed. EFLAGS is the *extended flags register* that consists of individual bits that either control the operation of the CPU or reflect the outcome of some CPU operations. We will learn more about these flags later.

2.3 FLAT Memory Model and Protected-Address Mode

The FLAT memory model is used in 32-bit operating systems (like Windows XP) running on a 32-bit processor. Each program addresses up to 4GB of memory. All code and data go into a single 32-bit (4-GigaByte) address space. Linear 32-bit addresses are used in each program to access the program instructions and data in memory.



Rather than using **real memory addresses**, a program uses **virtual addresses**. These **virtual** addresses are mapped onto **real (physical)** addresses by the operating system through a scheme called **paging**. The processor translates **virtual** addresses into **real** addresses as the program is running. With **virtual memory**, the processor runs in **protected mode**. This means that each program can access only the memory that was assigned to it by the operating system and cannot access the memory of other programs.

2.4 FLAT Memory Model Program Template

Writing an assembly language program is a complicated task, particularly for a beginner. We will simplify this task by hiding those details that are irrelevant. We will use the following template for writing FLAT memory programs. This template consists of three types of statements: executable instructions, assembler directives, and macros. Executable instructions generate machine code for the processor to execute at runtime. Assembler directives provide information to the assembler while translating the program. Macros are shorthand for a sequence of instructions, directives, or even other macros. We will learn more about instructions, directives, and macros throughout the semester.

```
TITLE FLAT Memory Program Template      (template.asm)

; Program Description:
; Author:
; Date Created:
; Last Modified:

.686
.MODEL FLAT, STDCALL
.STACK

INCLUDE Irvine32.inc

; (insert symbol definitions here)

.DATA

; (insert variables here)

.CODE
main PROC

; (insert executable instructions here)

    exit      ; exit to operating system
main ENDP

; (insert additional procedures here)

END main
```

The first line of an assembly language program is the **TITLE** line. This line is optional. It contains a brief heading of the program and the disk file name. The next few lines are line comments. They begin with a semicolon (;) and terminate with the end of the line. They are ignored and not processed by the assembler. However, they are used to document a program and are of prime importance to the assembly language programmer, because assembly language code is not easy to read or understand. Insert comments at the beginning of a program to describe the program, its author, the date when it was first written and the date when it was last modified. You need also comments to document your data and your code.

The **.MODEL** is a directive that specifies the memory configuration for the assembly language program. For our purposes, the **FLAT** memory model will be used. The **.686** is a processor directive used before the **.MODEL FLAT** directive to provide access to the 32-bit instructions and registers available in the Pentium Processor. The **STDCALL** directive tells the assembler to use standard conventions for names and procedure calls.

The **.STACK** is a directive that tells the assembler to define a stack for the program. The size of the stack can be optionally specified by this directive. The stack is required for procedure calls. We will learn more about the stack and procedures later during the semester.

The **.DATA** is a directive that defines an area in memory for the program data. The program's variables should be defined under this directive. The assembler will allocate storage for these variables and initialize their locations in memory.

The **.CODE** is a directive defines the code section of a program. The code is a sequence of assembly language instructions. These instructions are translated by the assembler into machine code and placed in the code area in memory.

The **INCLUDE** directive causes the assembler to include code from another file. We will include **Irvine32.inc** that specifies basic input and output procedures provided by the book author Kip Irvine, and that can be used to simplify programming. These procedures are defined in the **Irvine32.lib** library that we will link to the programs that we will write.

Under the code segment, we can define any number of procedures. As a convention, we will define the first procedure to be the *main* procedure. This procedure is defined using the **PROC** and **ENDP** directives:

```
main PROC
    . . .
main ENDP
```

The **exit** at the end of the main procedure is used to terminate the execution of the program and exit to the operating system. Note that **exit** is a macro. It is defined in **Irvine32.inc** and provides a simple way to terminate a program.

The **END** is a directive that marks the last line of the program. It also identifies the name (**main**) of the program's startup procedure, where program execution begins.

2.5 Writing a Program using the FLAT Memory Model

The following program adds and subtracts integers. You may open it using any text editor.

```
TITLE Add and Subtract    (addsub.asm)
; This program adds and subtracts integers

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.code
main PROC
    mov eax, 60000h        ; EAX = 60000h
    add eax, 80000h        ; EAX = EAX + 80000h
    sub eax, 20000h        ; EAX = EAX - 20000h
    exit
main ENDP
END main
```

2.5.1 Lab Work: Guessing the Value of the EAX Register

The constant values **60000h**, **80000h**, and **20000h** are in hexadecimal. Guess and write the values of the EAX register in the above program after the **add** and **sub** (subtract) instructions:

Value of EAX in hexadecimal after add =

Value of EAX in hexadecimal after sub =

2.5.2 Instructions

The above program uses 3 instructions. The **mov** instruction is used for moving data. The **add** instruction is used for adding two value, and the **sub** instruction is used for subtraction.

An instruction is a statement executed by the processor at runtime after the program has been loaded into memory and started. An instruction contains four basic parts:

Label: (optional)	Instruction Mnemonic	Operand(s)	; Comment
----------------------	-------------------------	------------	-----------

A *label* is an identifier that acts as a place marker for an instruction. It must end with a colon (:). The assembler assigns an address to each instruction. A *label* placed just before an instruction implies the instruction address. Labels are often used as targets of jump instructions.

An *instruction mnemonic* is a short word that identifies the operation carried out by the instruction when it is executed at runtime. Instruction mnemonics have useful names, such as **mov**, **add**, **sub**, **jmp** (jumping to a target instruction), and **call** (calling a procedure).

An instruction can have between zero and three *operands*, each of which can be a *register*, *memory operand*, *constant expression*, or an *I/O port*. In the above program (*addsub.asm*), the **mov**, **add**, and **sub** instructions used two operands. The first operand specified the destination, which was the **eax** register. The second operand was a constant integer value.

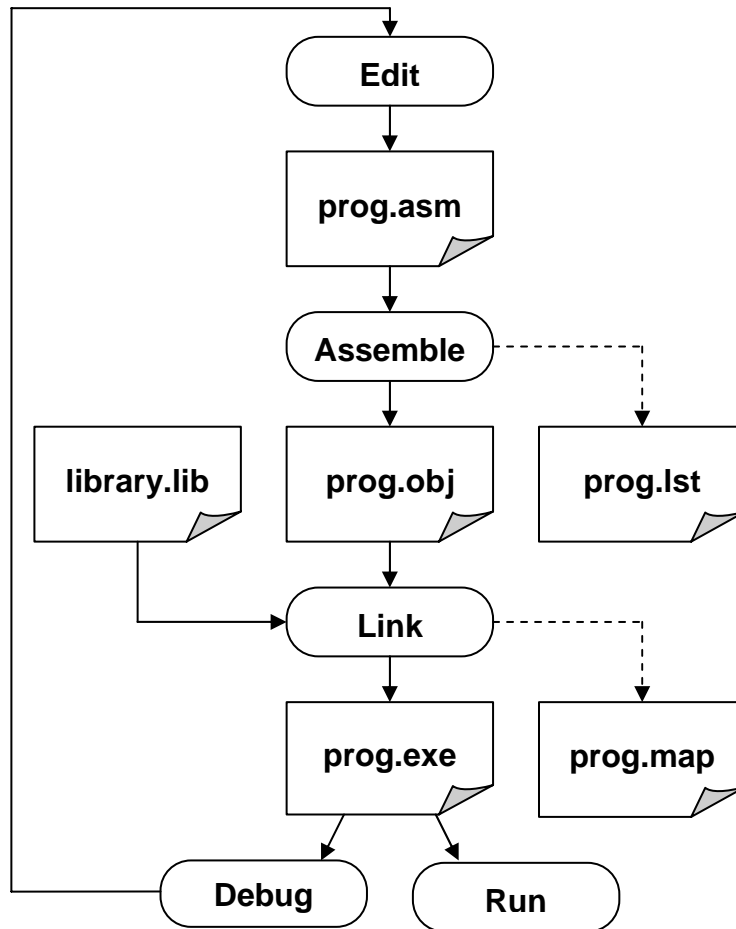
An instruction can be terminated with an optional comment. The comment starts with a semicolon (;) and terminates with the end of line. A comment at the end of an instruction can be used as a short description and clarification for the use of that instruction.

2.6 Editing, Assembling, Linking, and Debugging Programs

The process of editing, assembling, linking, and debugging programs is shown below. You will learn a lot about this cycle during this semester. The editor is used to write new programs and to make changes to existing ones.

Once a program is written, it can be assembled and linked using the **ML.exe** program. Alternatively, it can be assembled only using the **ML.exe** program and linked using the **LINK32.exe** program. The assembler translates the source assembly (**.asm**) file into an equivalent object (**.obj**) file in machine language. As an option, the assembler can also produce a listing (**.lst**) file. This file shows the work of the assembler and contains the assembly source code, offset addresses, translated machine code, and a symbol table.

The linker combines one or more object (**.obj**) files produced by the assembler with one or more link library (**.lib**) files to produce the executable program (**.exe**) file. In addition, the linker can produce an optional (**.map**) file. A map file contains information about the program being linked. It contains a list of segment groups in the program, a list of public symbols, and the address of the program's entry point.



Once the executable program is generated, it can be executed and/or debugged. A debugger allows you to trace the execution of a program and examine and/or modify the content of registers and memory. With a debugger, you will be able to discover your errors in the program. Once these errors are discovered, you will make the necessary changes in the source assembly program. You will go back to the editor to make these changes, assemble, link, and debug the program again. This cycle repeats until the program starts functioning properly, and correct results are produced.

2.6.1 Lab Work: Using ML, LINK32, and MAKE32 Commands

We will use the Command Prompt to assemble and link a 32-bit program. Type the following commands after changing the directory to one containing the *addsub.asm* program. Make sure the environment variables are set properly (as explained in Lab 1).

```

C:\COE205\Lab\02>ml -c -Zi -F1 -coff addsub.asm
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.

Assembling: addsub.asm

C:\COE205\Lab\02>link32 addsub.obj kernel32.lib /subsystem:console /debug /map
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

C:\COE205\Lab\02>
  
```

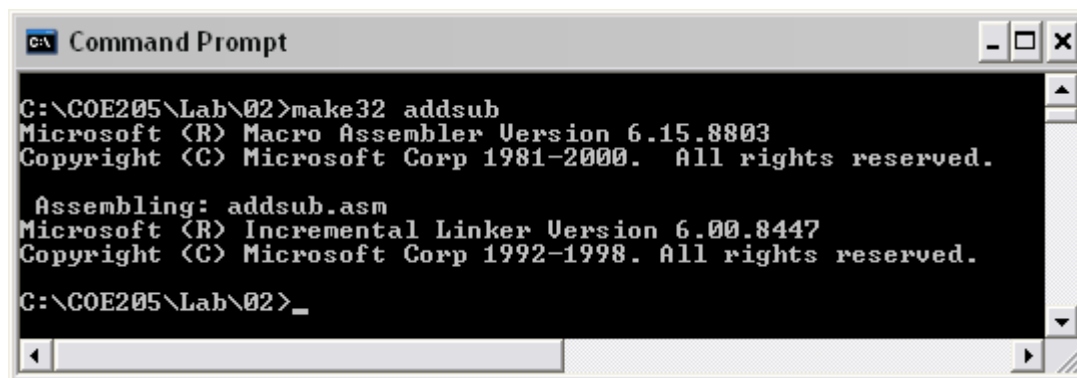
```
ml -c -Zi -Fl -coff addsub.asm
```

```
link32 addsub.obj kernel32.lib /subsystem:console /debug /map
```

The **ml** command assembles *addsub.asm*. The **-c** option is used to assemble only without linking. This will generate *addsub.obj* object file. The **-Zi** option will add debugging information and the **-Fl** option will generate the listing file *addsub.lst*. You may examine the *addsub.lst* file to understand how the assembler has translated the source assembly language file into an object file. The **-coff** option tells the assembler to generate a COFF object file (Common Object File Format) used for 32-bit programs.

The **link32** command is a 32-bit linker used to link *addsub.obj* file to the *kernel32.lib* library. This generates *addsub.exe*, a 32-bit executable file. The **/subsystem:console** option specifies the console as being the subsystem. The **/debug** option tells the linker to generate debugging information in the executable file, and the **/map** options generates a **.map** file.

To save typing, we can write a batch file to assemble and link a 32-bit program. This batch file is already written for you and exits under the MASM installation directory and is called **make32.bat**. Type **make32 addsub** to assemble and link the *addsub.asm* program.



```

C:\COE205\Lab\02>make32 addsub
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.

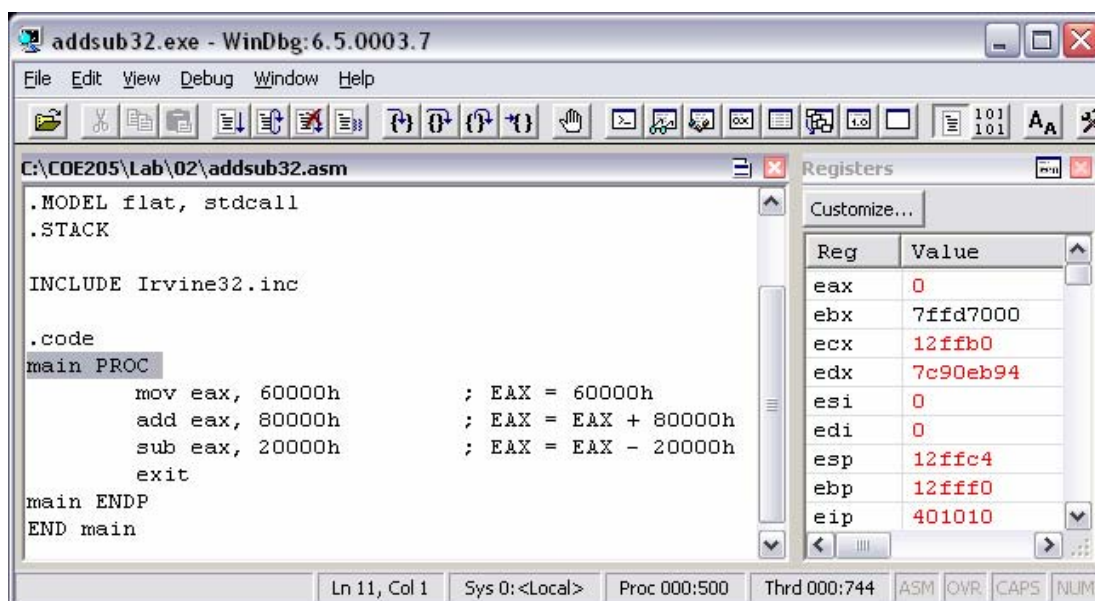
Assembling: addsub.asm
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

C:\COE205\Lab\02>_

```

2.6.2 Lab Work: Using the Windows Debugger

At the Command Prompt, type: **windbg -QY -G addsub.exe** to run the Windows Debugger. Make sure that the installation directory for **windbg.exe**, which is typically **C:\Program Files\Debugging Tools for Windows**, exists in the **path** variable.



```

C:\COE205\Lab\02\addsub32.asm
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.code
main PROC
    mov eax, 60000h      ; EAX = 60000h
    add eax, 80000h     ; EAX = EAX + 80000h
    sub eax, 20000h     ; EAX = EAX - 20000h
    exit
main ENDP
END main

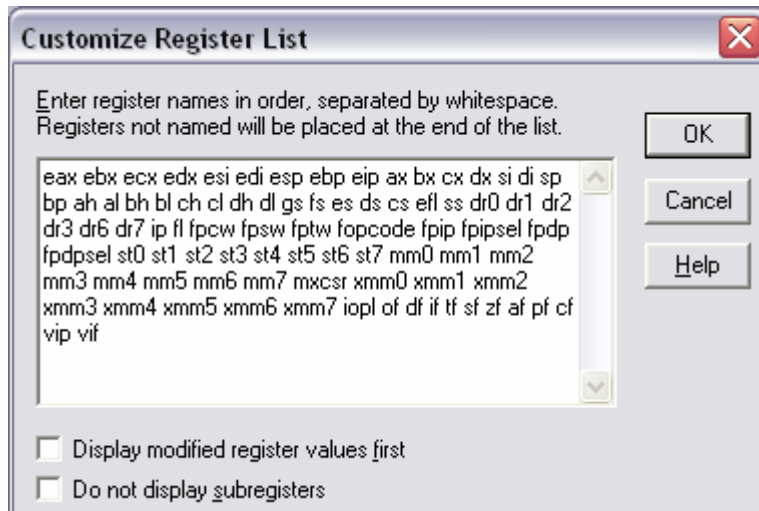
```

Reg	Value
eax	0
ebx	7ffd7000
ecx	12ffb0
edx	7c90eb94
esi	0
edi	0
esp	12ffc4
ebp	12fff0
eip	401010

Ln 11, Col 1 Sys 0: <Local> Proc 000:500 Thrd 000:744 ASM OVR CAPS NUM

Open the source file *addsub.asm* from the **File** menu if it is not already opened. Watch the registers by selecting **Registers** in the **View** menu or by pressing **Alt+4**. You can make the Registers window floating or you can dock it as shown above.

You can customize the order of registers. Click on the **Customize...** button and type **eax** at the beginning to show the **eax** register on top of the list. You can customize the rest as shown.



Place the cursor at the beginning of the main procedure and press **F7** to start the execution of the main procedure as shown above. Press **F10** to step through the execution of the main procedure. Observe the changes to the EAX register after executing each instruction. Make the necessary corrections to the values of EAX that you have guessed in Section 2.5.1.

2.6.3 Lab Work: Understanding Program Termination

The **exit** at the end of the main procedure is a macro. It is expanded into a system call to the **ExitProcess** MS-Windows function that terminates the program. This function is defined in the **kernel32** library. To have a better understanding of program termination, remove the **exit** at the end of the main procedure and replace it with the following two instructions:

```
exit           =>      push 0
                  call ExitProcess
```

The **push 0** pushes the number 0 on the stack and the **call** instruction is used to call the function **ExitProcess**. You can also replace **exit** with **INVOKE ExitProcess, 0**. The assembler will translate into: **push 0** and **call ExitProcess** as shown above.

There is no need to include the **Irvine32.inc** file, since the **exit** macro is no longer used. However, there is a need to declare that **ExitProcess** is an external function defined outside the *addsub.asm* program. We use the **PROTO** directive for this purpose. Therefore, remove the **INCLUDE** directive and replace it with the **PROTO** directive as shown below:

```
INCLUDE Irvine32.inc  =>      ExitProcess PROTO, ExitCode:DWORD
```

The **PROTO** directive declares functions used by a program and defined outside the program file. This directive also specifies the parameters and types of a given function.

Now that you have made these changes, assemble and link the program using the **ML** and **LINK32** programs as explained in Section 2.6.1. Notice that *addsub.obj* is linked to the *kernel32.lib* because the **ExitProcess** function is defined in the **kernel32** library. Open the debugger and trace the execution of the program.

Review Questions

1. Name all eight 32-bit general-purpose registers.
2. Name all eight 16-bit general-purpose registers.
3. Name all eight 8-bit general-purpose registers.
4. What special purpose does the EAX register serve?
5. What is the purpose of the EIP register?
6. What is the purpose of the ESP register?
7. In the FLAT memory model, how many bits are used to hold a memory address?
8. What is the meaning of the **INCLUDE** directive?
9. What does the **.CODE** directive identify?
10. Which directive begins a procedure and which directive ends it?
11. What is the purpose of the **END** directive?
12. What does the **PROTO** directive do?
13. What types of files are produced by the assembler?
14. What types of files are produced by the linker?

Programming Exercises

1. Using the *addsub* program as a reference, write a program that moves four integers into the EAX, EBX, ECX, and EDX registers and then accumulates their sum into the EAX register. Trace the execution of the program and view the registers using the windows debugger.
2. Rewrite the above program using the 16-bit registers: AX, BX, CX, and DX.