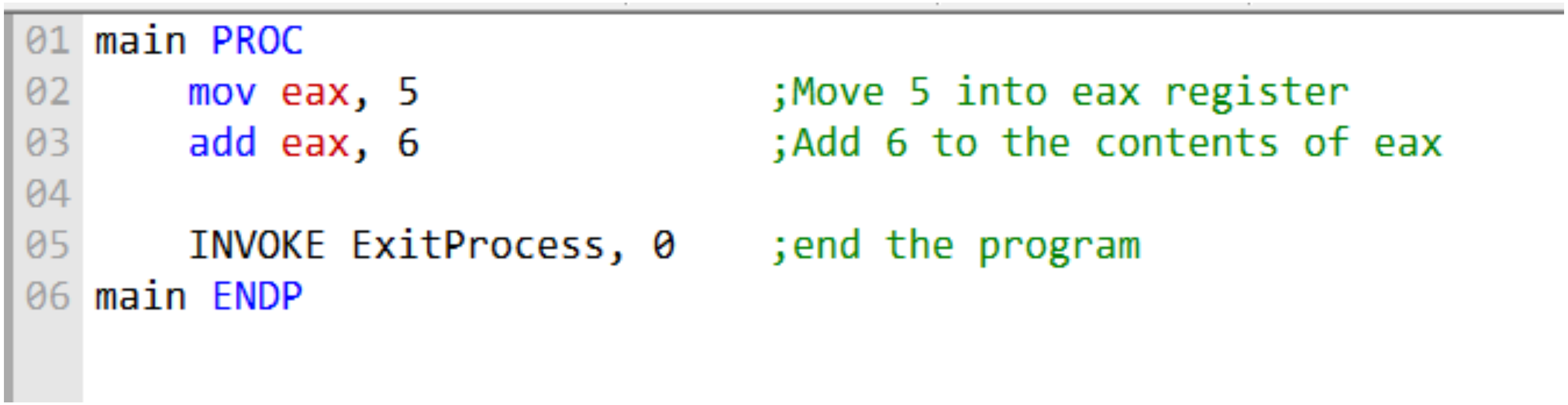
1ST ASSEMBLY PROGRAM

To see how this works, let’s look at a simple assembly language program that adds two numbers and saves the result in a register.



```
01 main PROC
02     mov eax, 5                  ;Move 5 into eax register
03     add eax, 6                  ;Add 6 to the contents of eax
04
05     INVOKE ExitProcess, 0       ;end the program
06 main ENDP
```

Also, don’t try to type in and run this program just yet—it’s missing some important declarations that we will include later on in this chapter.

Line 1 starts the main procedure, the entry point for the program.

Line 2 places the integer 5 in the eax register.

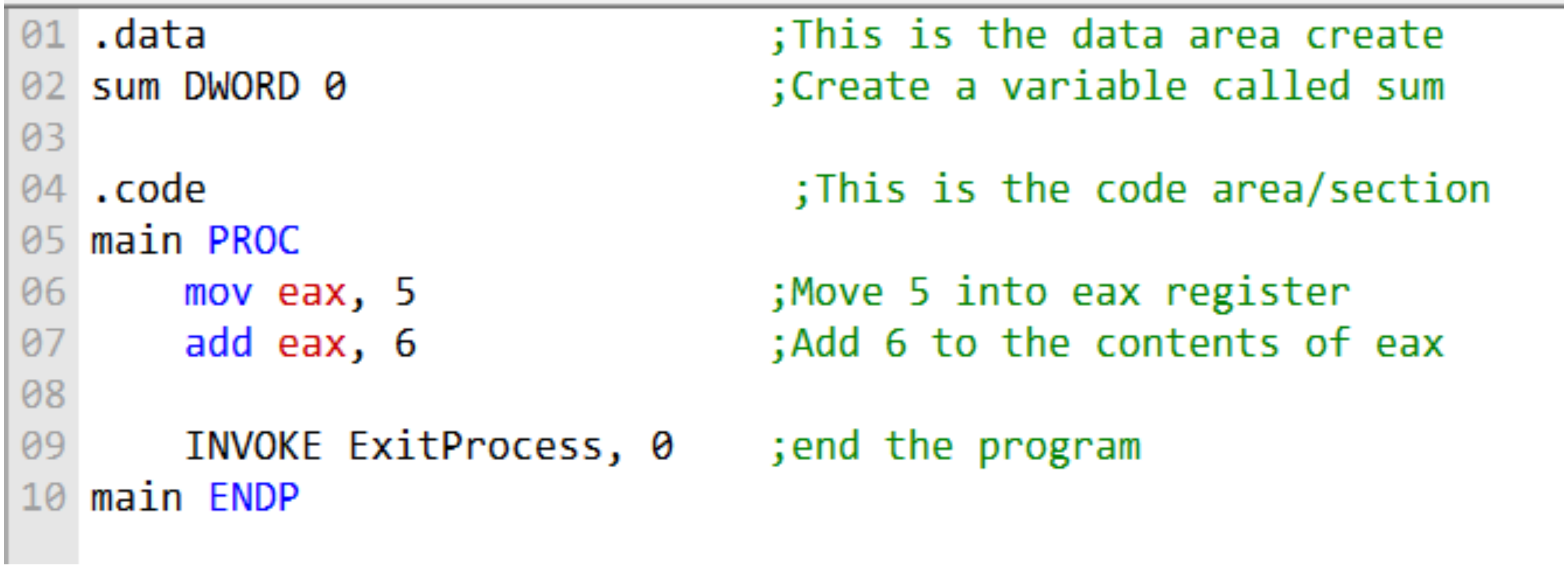
Line 3 adds 6 to the value in EAX, giving it a new value of 11.

Line 5 calls a Windows service (also known as a function) named ExitProcess that halts the program and returns control to the operating system.

Line 6 is the ending marker of the main procedure.

You probably noticed that we included comments, which always begin with a semicolon character.

We’ve left out a few declarations at the top of the program that we can show later, but essentially this is a working program.



```
01 .data                        ;This is the data area create
02 sum DWORD 0                  ;Create a variable called sum
03
04 .code                         ;This is the code area/section
05 main PROC
06     mov eax, 5               ;Move 5 into eax register
07     add eax, 6               ;Add 6 to the contents of eax
08
09     INVOKE ExitProcess, 0    ;end the program
10 main ENDP
```

The sum variable is declared on Line 2, where we give it a size of 32 bits, using the DWORD keyword.

There are a number of these **size keywords,** which work more or less like data types.

But they are not as specific as types you might be familiar with, such as int, double, float, and so on.

They **only specify a size**, but there’s no checking into what actually gets put inside the variable.

Remember, **you are in total control**.

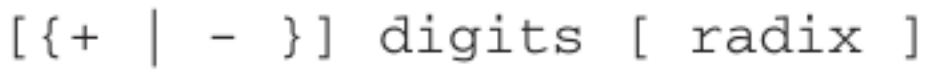
By the way, those code and data areas we mentioned, which were marked by the **.code** and **.data directives**, are called segments.

So you have the **code segment** and the **data segment**.

Later on, we will see a third segment named **stack.**

INTEGER LITERALS

An integer literal (also known as an integer constant) is made up of an **optional leading sign**, **one or more digits**, and an **optional radix character** that indicates the number’s base:



```
[{+ | - }] digits [ radix ]
```

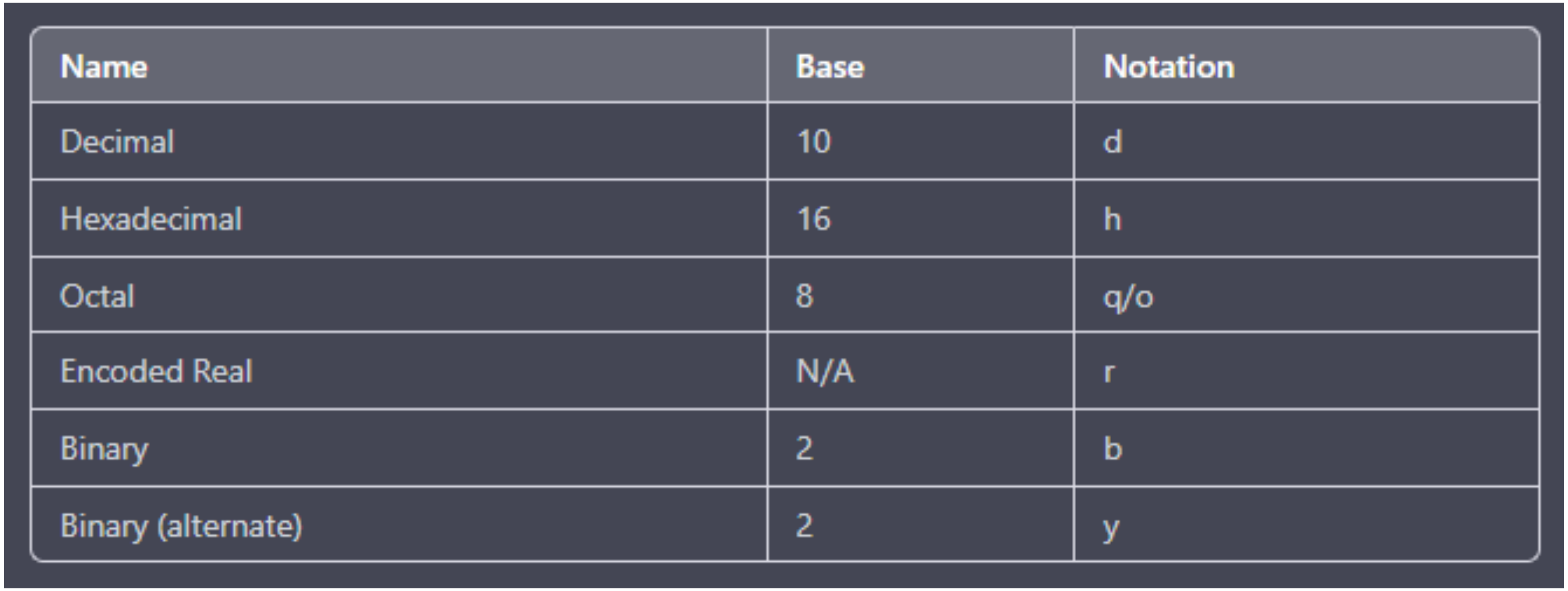
So, for example, 26 is a valid integer literal.

It doesn’t have a radix, so we assume it’s in decimal format.

If we wanted it to be 26 hexadecimal, we would have to write it as 26h.

Similarly, the number 1101 would be considered a decimal value until we added a “b” at the end to make it 1101b (binary).

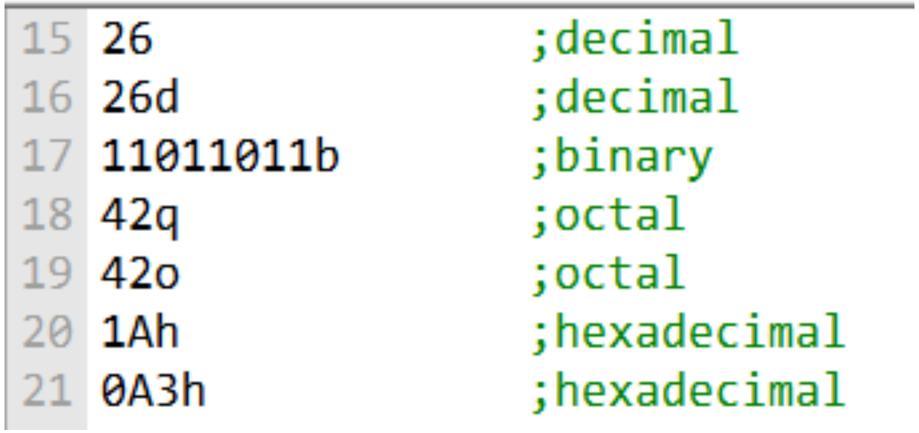
Here are the possible radix values:



Name	Base	Notation
Decimal	10	d
Hexadecimal	16	h
Octal	8	q/o
Encoded Real	N/A	r
Binary	2	b
Binary (alternate)	2	y

Note that **"Encoded Real"** does not have a specific base value since it is a format used to represent floating-point numbers in binary.

And here are some integer literals declared with various radixes. Each line contains a comment:

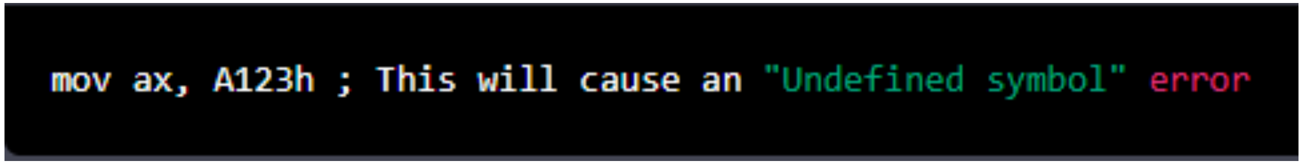


```
26                ;decimal
26d               ;decimal
11011011b         ;binary
42q               ;octal
42o               ;octal
1Ah               ;hexadecimal
0A3h              ;hexadecimal
```

HEXADECIMAL BEGINNING WITH A LETTER

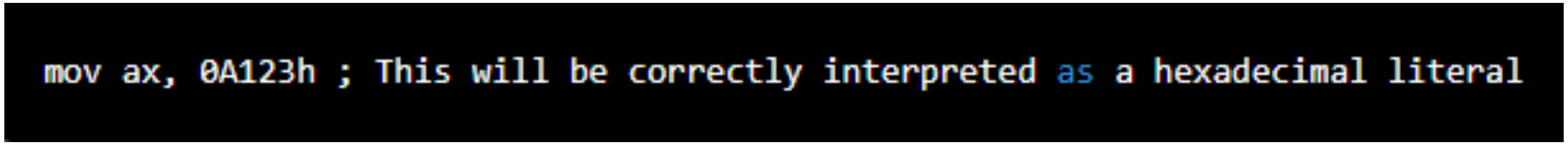
A hexadecimal literal beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

Here's an example of a hexadecimal literal beginning with a letter that requires a leading zero to prevent it from being interpreted as an identifier:



```
mov ax, A123h ; This will cause an "Undefined symbol" error
```

In the above example, the assembler will interpret A123h as an identifier because it begins with the letter "A". To prevent this, we need to add a leading zero to indicate that it is a hexadecimal literal:



```
mov ax, 0A123h ; This will be correctly interpreted as a hexadecimal literal
```

In the corrected example, the **leading zero** tells the assembler that **A123h** should be treated as a hexadecimal number, not an identifier.

CONSTANT INTEGER EXPRESSION

A constant integer expression is a mathematical expression involving integer literals and arithmetic operators.

Each expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh).

The arithmetic operators are listed in Table 3-1 according to their precedence order, from highest (1) to lowest (4).

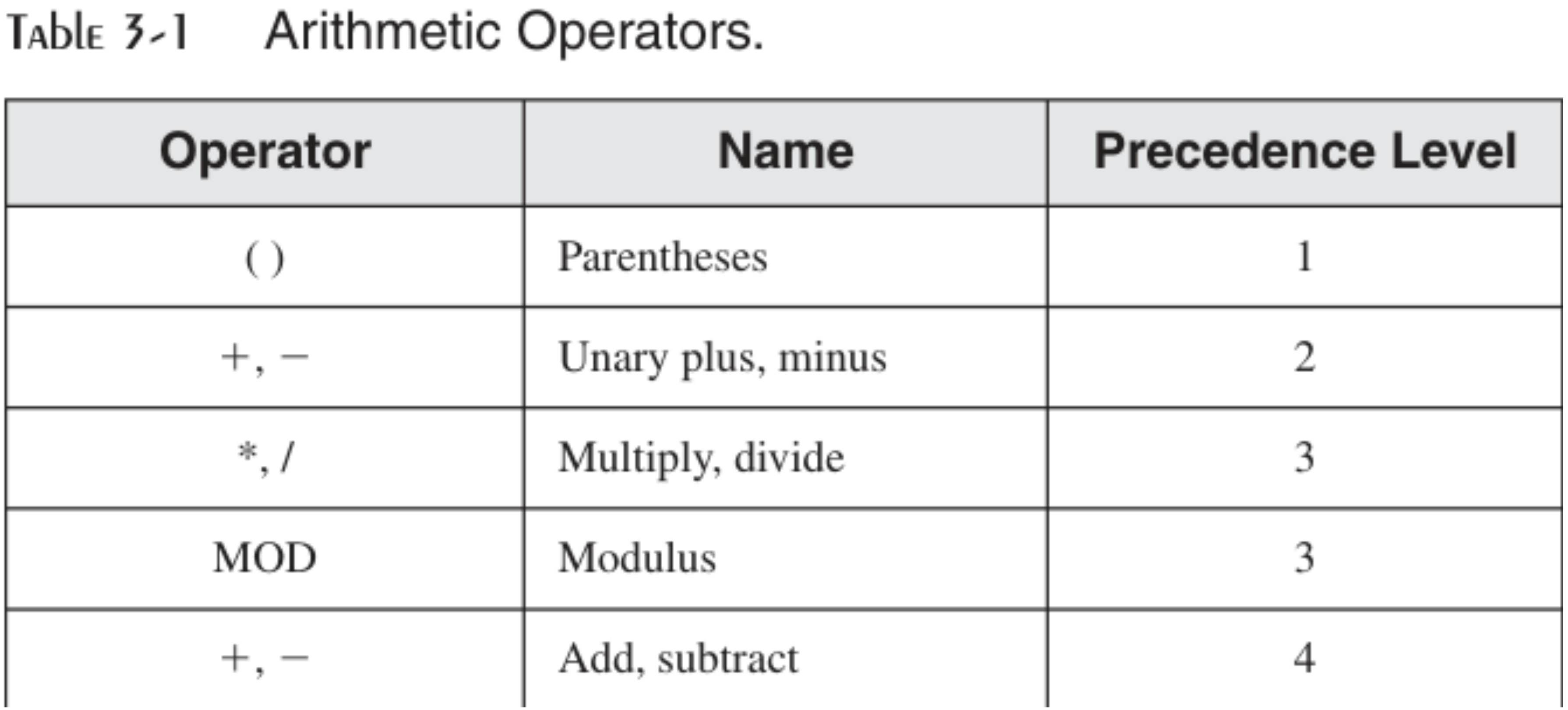


Table 3-1 Arithmetic Operators.

Operator	Name	Precedence Level
( )	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

The important thing to realize about constant integer expressions is that they can only be evaluated at assembly time.

From now on, we will just call them integer expressions.

In assembly language programming, **unary plus and minus** are operators that perform arithmetic operations on a single operand.

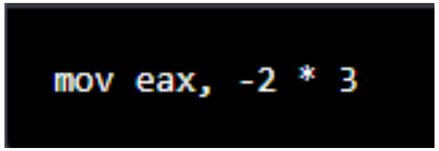
The **unary plus operator** simply returns the value of its operand, while the **unary minus operator** returns the negative value of its operand.

The reason that the **precedence of unary plus and minus is higher than that of division and multiplication** is due to the rules of operator precedence in most programming languages, including assembly language.

According to these rules, unary plus and minus have the highest precedence, followed by multiplication and division, and then addition and subtraction.

This means that when a mathematical expression contains both unary operators and multiplication or division operators, the unary operators are evaluated first, before the multiplication or division is performed.

For example, consider the following expression:



```
mov eax, -2 * 3
```

In this expression, the unary minus operator has a higher precedence than the multiplication operator.

This means that the expression -2 is evaluated first, resulting in the value -2.

The multiplication is then performed, resulting in the final value of -6.

If the multiplication operator had a higher precedence than the unary minus operator, the expression would be evaluated as (-2) * 3, resulting in a value of -6.

It's worth noting that operator precedence can be overridden using parentheses to group operations in the desired order.

Operator precedence refers to the implied order of operations when an expression contains two or more operators.

4 + 5 * 2	Multiply, add
12 -1 MOD 5	Modulus, subtract
-5 + 2	Unary minus, add
(4 + 2) * 6	Add, multiply

Use parentheses in expressions to clarify the order of operations so you don’t have to remember precedence rules.

In assembly language programming, the **modulus operator** is used to **calculate the remainder of a division operation** between two operands. It is denoted by the percent sign (%), and is also sometimes referred to as the "remainder" operator.

REAL NUMBER LITERALS

Real number literals (also known as floating-point literals) are represented as either decimal reals or encoded (hexadecimal) reals.

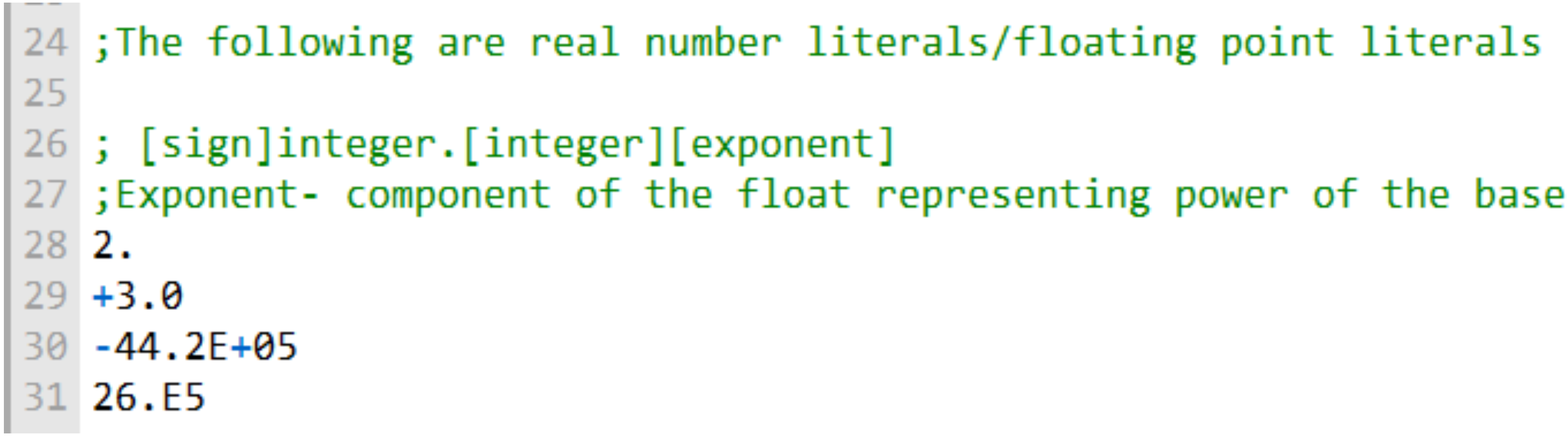
A decimal real contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

[sign]integer.[integer][exponent]

These are the formats for the sign and exponent:

sign {+,-} exponent **E[{+,-}]** integer

Following are examples of valid decimal reals:



```
;The following are real number literals/floating point literals

; [sign]integer.[integer][exponent]
;Exponent- component of the float representing power of the base
2.
+3.0
-44.2E+05
26.E5
```

Both 26.E5 and 44.2E05 are valid ways to represent a floating-point number in scientific notation. The notation [sign]integer.[integer][exponent] is just one convention for representing floating-point numbers in scientific notation, but it's not the only convention.

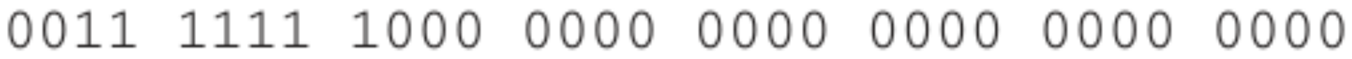
In the notation 26.E5, the integer component is assumed to be 0. This is because in scientific notation, the significant digits of a number are represented by the digits to the left of the decimal point, while the exponent represents the power of 10 that the number is being multiplied by. So in this case, the significant digits are 26, and the exponent is 5, meaning that the number is equal to 26 x 10^5, or 2,600,000.

In the notation 44.2E05, the integer component is 44, and the decimal component is 2. The exponent is 5, meaning that the number is equal to 44.2 x 10^5, or 4,420,000.

So while the notation [sign]integer.[integer][exponent] is a common convention for representing floating-point numbers in scientific notation, it's not necessary to include an integer component before the decimal point if the significant digits of the number don't require it.

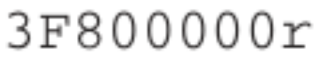
EXAMPLE 1:

An **encoded real** represents a real number in hexadecimal, using the IEEE floating-point format for short reals.



```
0011 1111 1000 0000 0000 0000 0000 0000
```

The same value would be encoded as a short real in assembly language as:



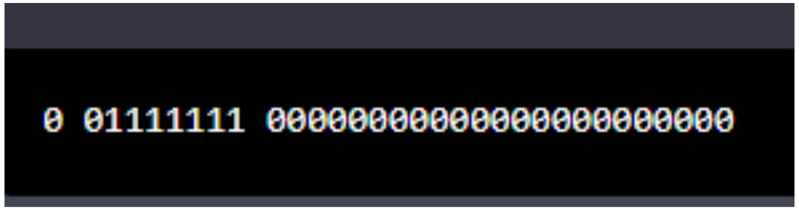
```
3F800000r
```

We will not be using real-number constants for a while, because most of the x86 instruction set is geared toward integer processing.

However, Chapter 12 will show how to do arithmetic with real numbers, also known as floating-point numbers. It’s very interesting, and very technical.

EXAMPLE 2:

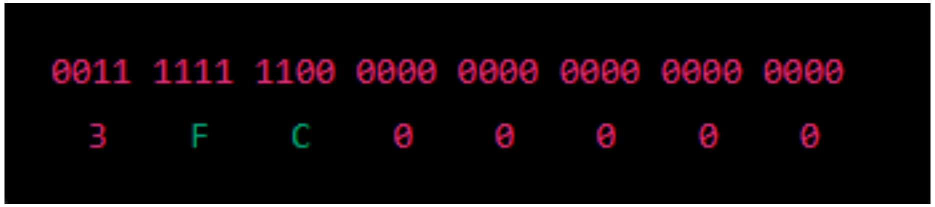
The binary representation of decimal +1.0 in IEEE floating-point format for short reals is:



```
0 01111111 00000000000000000000000
```

In this representation, the first bit is the sign bit (0 for positive), the next 8 bits represent the biased exponent (in this case, 01111111 is the biased exponent for +1.0), and the last 23 bits represent the mantissa (which is 1.0 for this number).

To convert this to hexadecimal, we can group the bits into groups of 4 and convert each group to its hexadecimal equivalent:

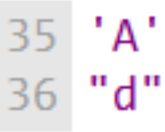


```
0011 1111 1100 0000 0000 0000 0000 0000
  3    F    C    0    0    0    0    0
```

So the hexadecimal representation of decimal +1.0 in IEEE floating-point format for short reals is 3FC00000.

CHARACTER LITERALS

A **character literal** is a single character enclosed in single or double quotes. The assembler stores the value in memory as the character's binary ASCII code. Examples are:



```
'A'
"d"
```

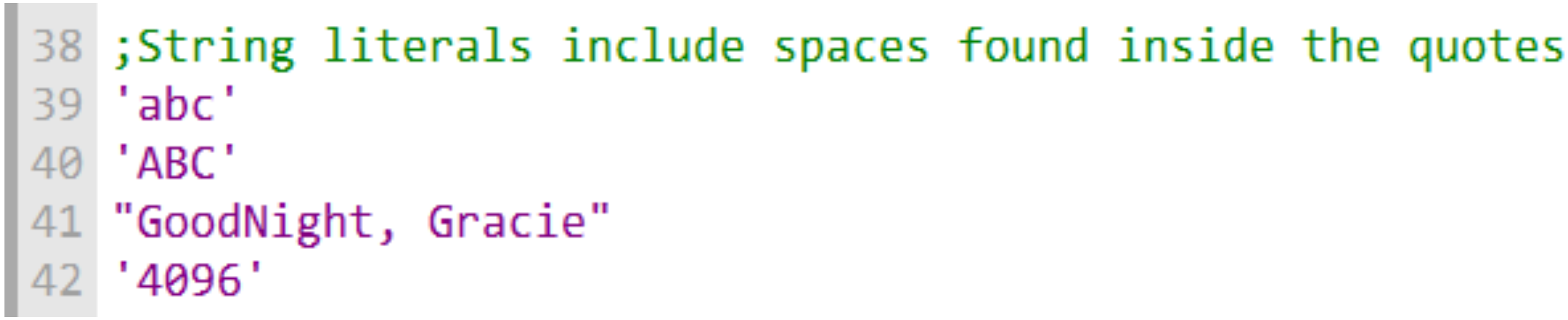
Recall that Chapter 1 showed that character literals are stored internally as integers, using the ASCII encoding sequence.

So, when you write the character constant "A," it's stored in memory as the number 65 (or 41 hex).

We have a complete table of ASCII codes on the inside front cover of this book, so be sure to look over them from time to time.

STRING LITERALS

A **string literal** is a sequence of characters (including spaces) enclosed in single or double quotes:



```
;String literals include spaces found inside the quotes
'abc'
'ABC'
"GoodNight, Gracie"
'4096'
```

Just as character constants are stored as integers, we can say that string literals are stored in memory as sequences of integer byte values.

So, for example, the string literal “ABCD” contains the four bytes 41h, 42h, 43h, and 44h.

WHY CHARACTERS, STRING LITERALS ETC ARE STORED AS INTEGERS IN A COMPUTER MEMORY?

Characters and string literals in assembly are stored in memory as integers because all data in a computer's memory is ultimately represented as a sequence of binary digits (bits). In other words, **everything in a computer's memory is represented as a number.**

In order to represent characters and strings in memory, **a standard encoding scheme is used, such as ASCII or Unicode.** These encoding schemes define a mapping between characters and their corresponding numerical codes, which are typically represented as integers.

For example, in ASCII, the letter "A" is represented by the integer value 65, the letter "B" is represented by 66, and so on. String literals in assembly are typically stored as a sequence of these integer values, with a null terminator (a value of 0) at the end to indicate the end of the string.

So while characters and string literals are conceptually different from integers, they are **stored in memory as integers because that is the fundamental representation of all data in a computer's memory.**

RESERVED WORDS

Reserved words have special meaning and can only be used in their correct context. Reserved works, by default, are not case-sensitive. For example, MOV is the same as mov and Mov. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL

- Register names.
- Directives, which tell the assembler how to assemble programs.
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD.
- Operators, used in constant expressions.
- Predefined symbols, such as @data, which return constant integer values at assembly time.

IDENTIFIERS

An **identifier** is a name that a programmer chooses to give to a variable, constant, procedure, or code label in their program. Identifiers can be used to make the code more readable and to help the programmer understand what the different parts of the program do.

Here are the rules for forming identifiers in assembly:

1. Length: An identifier can be between 1 and 247 characters long. This means that identifiers can be very long if needed, but it's usually best to keep them relatively short and descriptive.

2. Case sensitivity: In assembly, identifiers are not case sensitive. This means that the same identifier can be written using upper or lowercase letters (or a combination of both), and the assembler will treat them as the same identifier. For example, "myVariable" and "MYVARIABLE" are treated as the same identifier.

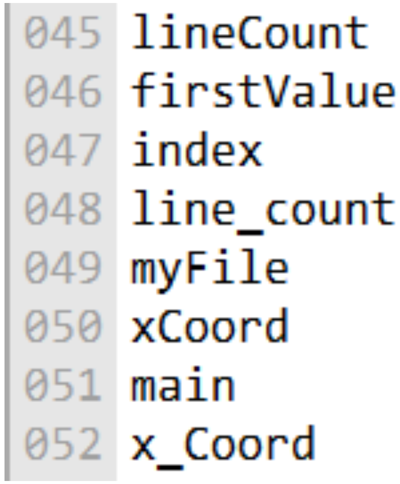
3. First character: The first character of an identifier must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. This means that an identifier cannot start with a number or any other character. For example, "_myVariable" and "$myVariable" are both valid identifiers.

4. Subsequent characters: After the first character, subsequent characters in an identifier may also be digits (0..9). This means that an identifier can include numbers, but it must start with a letter, underscore, @ , ?, or $.

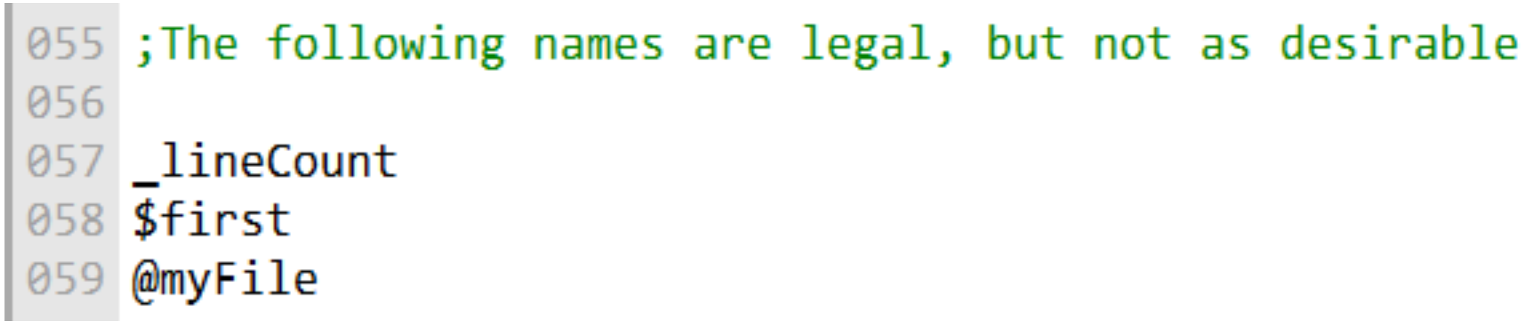
5. An identifier **cannot be** the same as an assembler **reserved word.** This means that certain words are reserved by the assembler for specific purposes, and cannot be used as identifiers. Example: mov, add, sub, and jmp.

Generally, you should avoid the @ symbol and underscore as leading characters, since they are used both by the assembler and by high-level language compilers.

Although assembly language instructions are short and cryptic, there’s no reason to make your identifiers hard to understand also! Here are some examples of well-formed names:



```
lineCount
firstValue
index
line_count
myFile
xCoord
main
x_Coord
```



```
;The following names are legal, but not as desirable

_lineCount
$first
@myFile
```

DIRECTIVES

Directives in assembly are special instructions that are used to provide information to the assembler, rather than to generate machine code instructions. Directives are typically used to specify data values, allocate memory for variables or arrays, and to control the layout and organization of the generated code.

A directive is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime, but they let you define variables, macros, and procedures.

They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive.

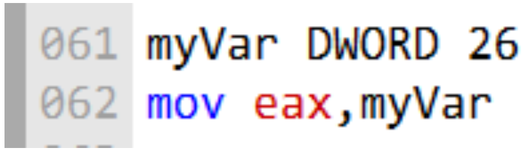
For example, **.data**, **.DATA,** and **.Data** are equivalent.

The following example helps to show the difference between **directives** and **instructions.**

The **DWORD directive** tells the assembler to reserve space in the program for a doubleword variable.

The **MOV instruction**, on the other hand, executes at runtime, copying the contents of myVar to the

EAX register:



```
myVar DWORD 26
mov eax,myVar
```

Although all assemblers for Intel processors share the same instruction set, they usually have different sets of directives. The **Microsoft assembler’s REPT directive**, for example, is not recognized by some other assemblers.

Defining Segments One important function of assembler directives is to define program sections, or segments. Segments are sections of a program that have different purposes.

For example, one segment can be used to define variables, and is identified by the .DATA directive:

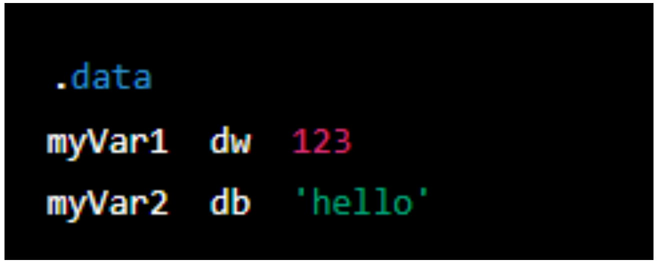
Here are some common directives used in assembly programming:

.data:

One segment can be used to **define variables**, and is identified by the .DATA directive.

This directive is used to specify that the following lines of code will **define data values that need to be stored in memory.**

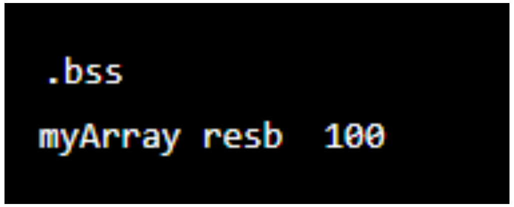
The .data directive is typically followed by one or more lines that specify the names and initial values of variables. Example:



```
.data
myVar1  dw  123
myVar2  db  'hello'
```

.bss:

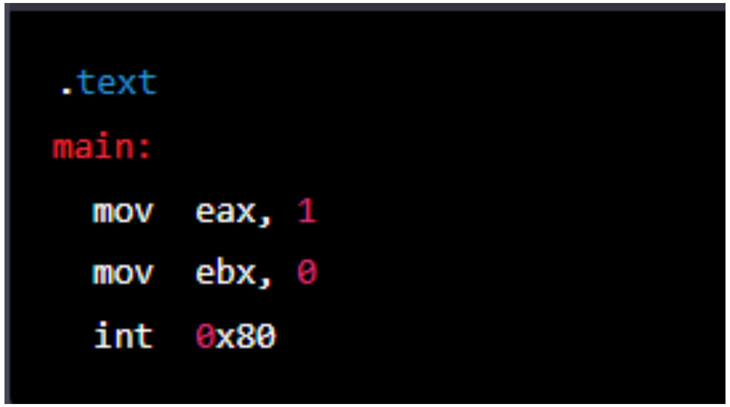
This directive is used to specify that the following lines of code will **define uninitialized data** that needs to be stored in memory. The .bss directive is typically followed by one or more lines that specify the names and sizes of variables. Example:



```
.bss
myArray resb  100
```

.text:

The .CODE or .TEXT directive identifies the area of a program **containing executable instructions.** This directive is used to specify that the following lines of code will define executable instructions that need to be loaded into memory. The .text directive is typically followed by one or more lines that specify the assembly language instructions to be executed. Example:



```
 .text
main:
  mov  eax, 1
  mov  ebx, 0
  int  0x80
```

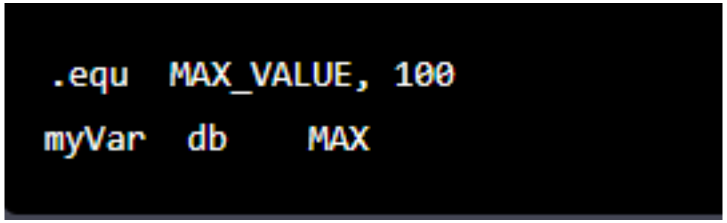
.equ:

In assembly language, a **symbol** is a label that is used to represent a memory location or a constant value. It can be used to represent a variety of things such as data, instructions, addresses, and offsets.

Symbols are often used to **make code more readable** and easier to understand. They allow programmers to use meaningful names instead of memory addresses or raw numbers.

Additionally, symbols can be used to make code more flexible and portable, as they can be easily modified without changing the underlying code.

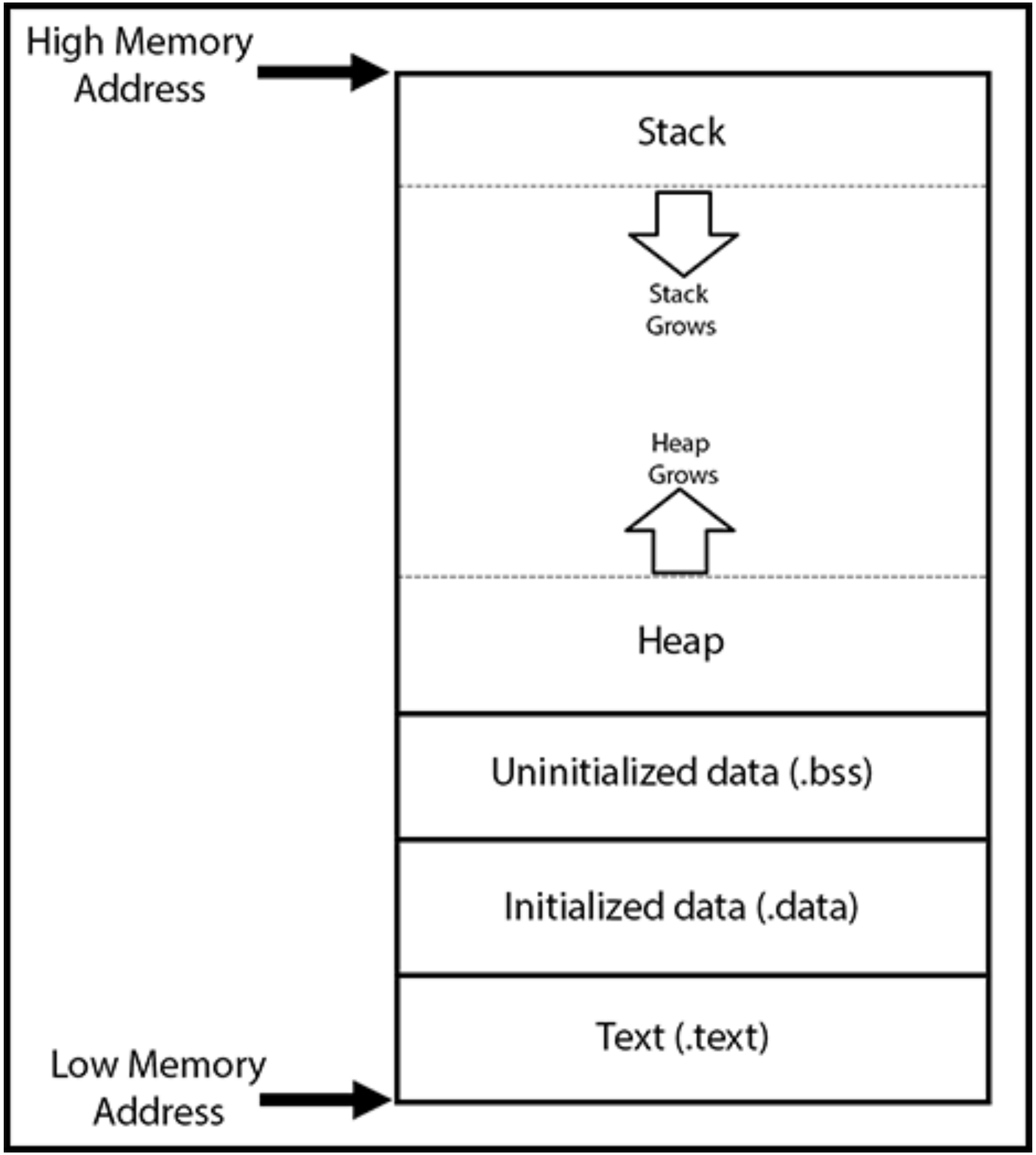
The **.equ directive** in assembly language is used to define a symbol with a constant value. Once defined, the symbol can be used throughout the code to represent that value. Example:



```
 .equ  MAX_VALUE, 100
myVar  db    MAX
```

.stack:

The .STACK directive identifies the area of a program holding the runtime stack, setting its size.

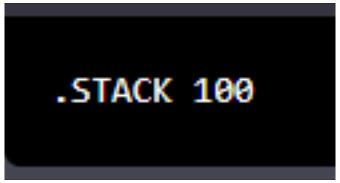

High Memory Address
Stack
Stack Grows
Heap Grows
Heap
Uninitialized data (.bss)
Initialized data (.data)
Text (.text)
Low Memory Address

The runtime stack is a data structure used by a program to manage subroutine calls and local variables.

It is a contiguous area of memory that is dynamically allocated and grows downwards, with the top of the stack being the most recently pushed data.

The .STACK directive in assembly language identifies the starting address and size of the runtime stack.

The syntax of the .STACK directive varies depending on the assembler, but generally it takes one argument, which is the size of the stack in bytes. For example, in NASM (Netwide Assembler), the syntax is as follows:

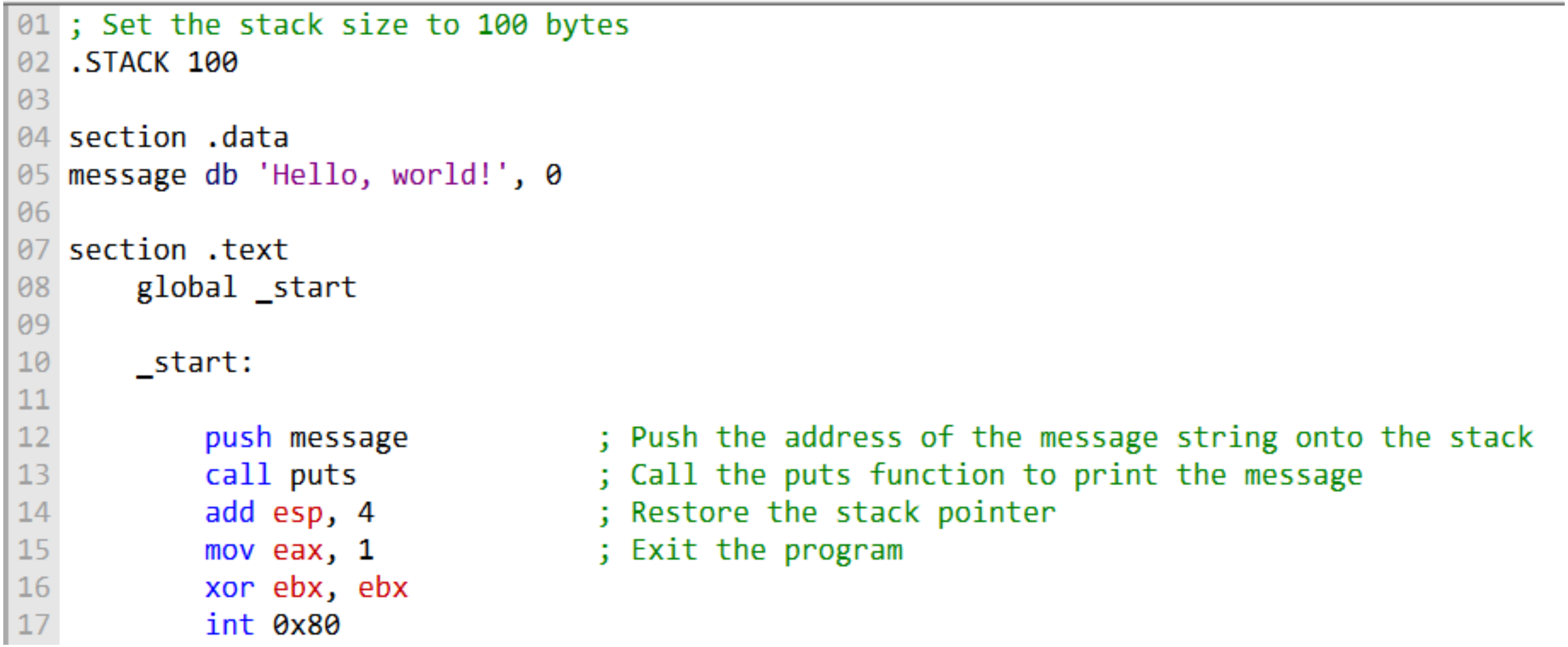


```
.STACK 100
```

This directive sets the size of the runtime stack to 100 bytes. The assembler will reserve this amount of memory for the stack at runtime, starting from the address specified by the operating system.

It is important to set the correct size of the stack to prevent stack overflow, which occurs when the stack grows beyond its allocated size and overwrites other memory. A stack overflow can cause a program to crash or behave unpredictably.

Here is an example of using the .STACK directive in a simple assembly program:



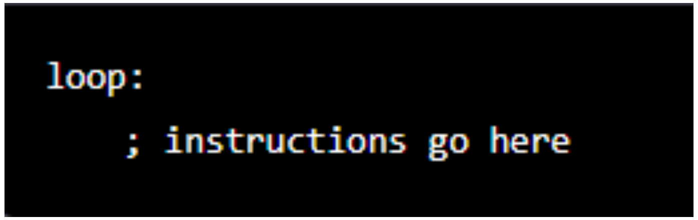
```
; Set the stack size to 100 bytes
.STACK 100

section .data
message db 'Hello, world!', 0

section .text
    global _start

    _start:

        push message           ; Push the address of the message string onto the stack
        call puts              ; Call the puts function to print the message
        add esp, 4             ; Restore the stack pointer
        mov eax, 1             ; Exit the program
        xor ebx, ebx
        int 0x80
```

INSTRUCTIONS

Instruction is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. Parts of an instruction:

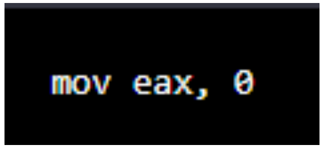
1. Label (optional):A label is an optional identifier that is used to mark a specific memory location or instruction in a program. It is usually followed by a colon (:) and must be unique within the program. Labels are used to provide reference points for jump and branch instructions.

Here is an example of a label in assembly language:



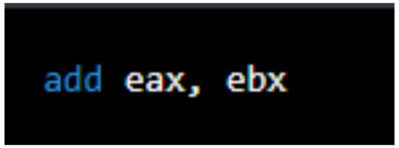
```
loop:
    ; instructions go here
```

2. Instruction mnemonic (required): The instruction mnemonic is the operation code (opcode) that tells the computer what operation to perform. It is a short abbreviation for the operation, such as ADD for addition or MOV for move. Here is an example of an instruction mnemonic in assembly language:



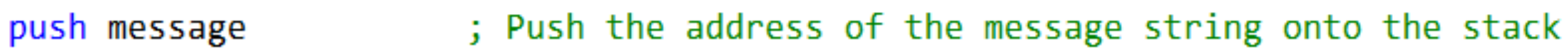
```
mov eax, 0
```

3. Operand(s) (usually required): An operand is a value or memory address that is used as an input or output for an instruction. Most instructions require one or more operands. The operands can be constants, registers, memory locations, or labels. Here is an example of an instruction with operands in assembly language:



```
add eax, ebx
```

4. Comment (optional):A comment is a note or explanation that is added to the program to provide information to the programmer or to document the code. Comments are ignored by the assembler and do not affect the program execution. Here is an example of a comment in assembly language:



```
push message            ; Push the address of the message string onto the stack
```

LABEL

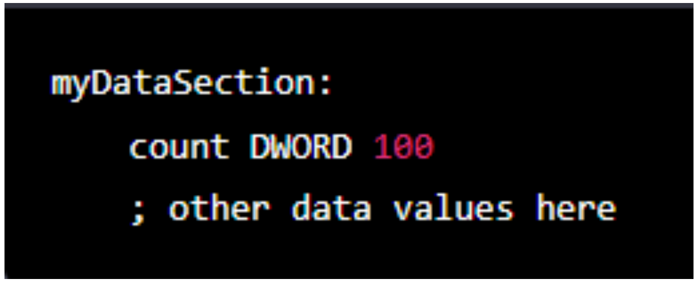
Labels in assembly language are identifiers that are used to mark specific locations in the program code. They are used to refer to specific memory addresses or instructions within the program. Labels are typically used to **mark the beginning of a subroutine or function**, to **indicate the location of a loop** or **conditional branch**, or to **mark a specific data location in memory.** A label in the code area of a program (where instructions are located) must end with a colon (:) character.

MAIN POINT: In assembly language, labels are typically defined using a colon (:) at the end of the label name, followed by the instruction or data that follows. For example, the label "loop_start:" might be used to mark the beginning of a loop in the code.

A label is an identifier used to mark a specific location in the program code, so there would need to be an additional line of code that defines a label. Labels are essential in assembly language programming, as they allow programmers to write code that is easy to read and understand, and to create more complex and powerful programs.

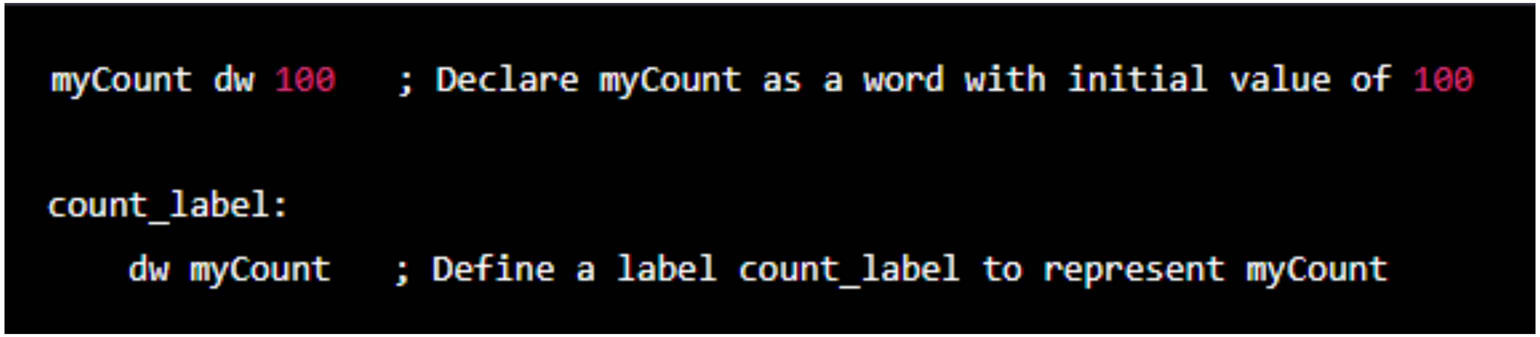
A **label** is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

There are two types of labels: **Data labels** and **Code labels.** A **data label identifies the location of a variable**, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:



```
myDataSection:
    count DWORD 100
    ; other data values here
```

Here's an example of using a label to represent the variable myCount declared as a word (2 bytes) with an initial value of 100 in assembly language.

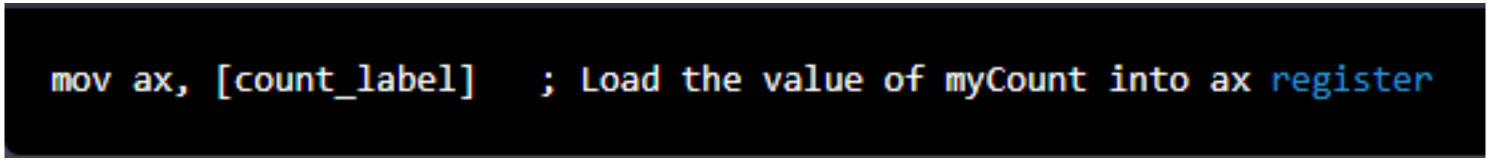


```
myCount dw 100   ; Declare myCount as a word with initial value of 100

count_label:
    dw myCount   ; Define a label count_label to represent myCount
```

In this example, the **myCount variable** is declared using the dw directive with an initial value of 100. Then, a label **count_label is defined to represent the memory address where myCount is stored.**

The **dw directive**
in the count_label assigns the value of myCount to the label count_label. The label **count_label can now be used in place of the variable myCount** in any instruction that requires its memory address.

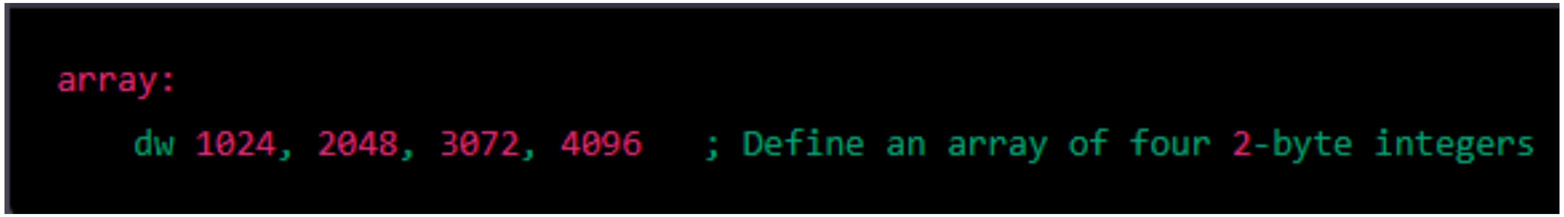
For example, to load the value of myCount into the eax register, you can use the following instruction:



```
mov ax, [count_label]   ; Load the value of myCount into ax register
```

Note that the square brackets around **count_label** indicate that the **value** at the memory address represented by count_label should be loaded, not the memory address itself.

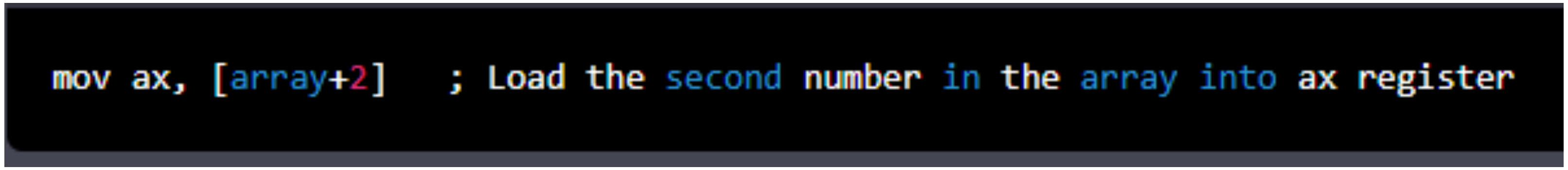
In assembly language programming, the assembler assigns a numeric address to each label. This address represents the memory location where the label is defined. It is also possible to define multiple data items following a label. The following example illustrates this:



```
array:
    dw 1024, 2048, 3072, 4096   ; Define an array of four 2-byte integers
```

In this example, **the label array defines the memory location of the first number in the array**, which is 1024. The other numbers (2048, 3072, and 4096) immediately follow in memory, each taking up 2 bytes.

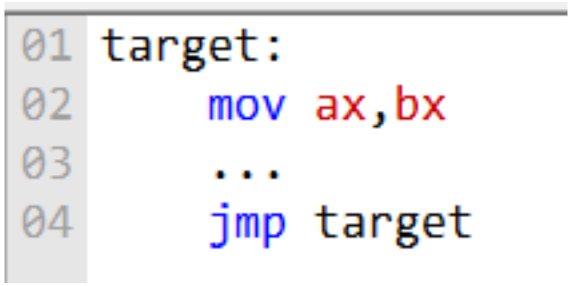
Using the label array, you can easily reference any of the numbers in the array by adding an offset to the label's address. For example, to load the second number (2048) into the eax register, you can use the following instruction:



```
mov ax, [array+2]   ; Load the second number in the array into ax register
```

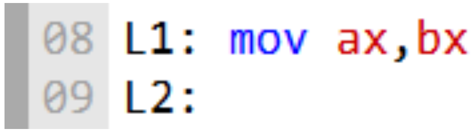
In this example, the +2 offset is added to the address of the array label to reference the memory location of the second number in the array.

Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named target, creating a loop:



```
target:
    mov ax,bx
    ...
    jmp target
```

A **code label** can share the same line with an instruction, or it can be on a line by itself:



```
L1: mov ax,bx
L2:
```

Label names follow the same rules we described for identifiers. You can use the same code label more than once in a program as long as each label is unique within its enclosing procedure.

INSTRUCTION MNEMONIC

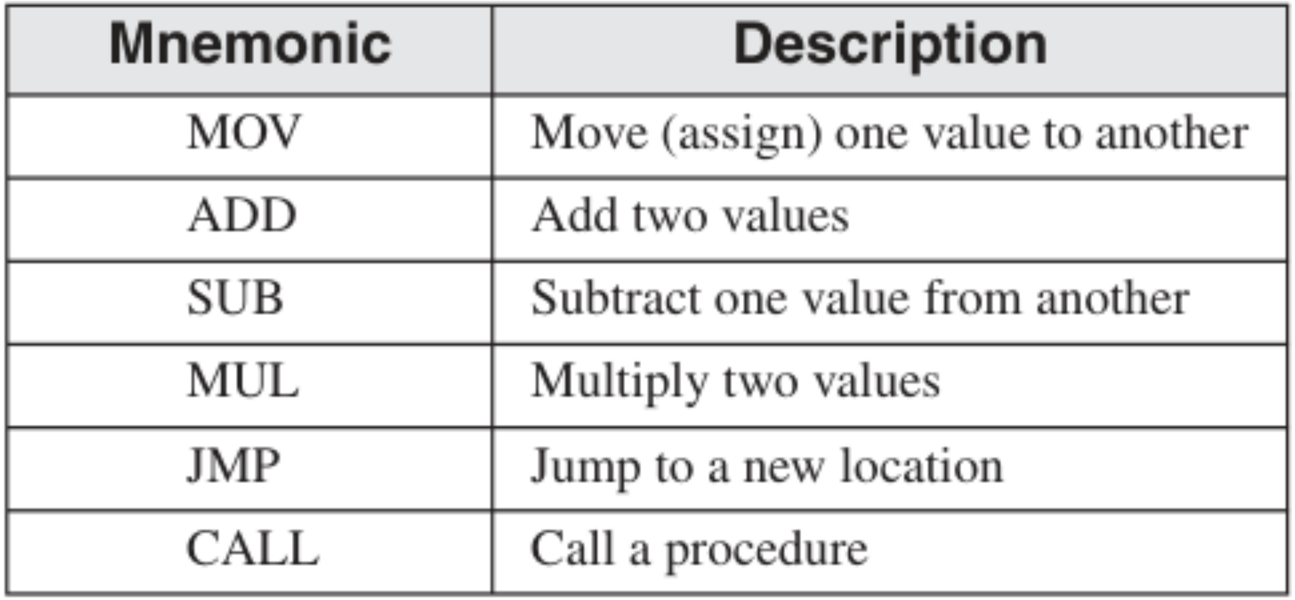
Instruction mnemonics are a **shorthand notation** used in assembly language programming **to represent machine instructions** in a more readable and human-friendly way.

Each instruction mnemonic corresponds to a specific operation that the processor can perform, such as adding two numbers together or branching to a different part of the program.

Similarly, assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform.

Here are some examples of common instruction mnemonics in assembly language:

- **MOV -** move data from one location to another
- **ADD -** add two values together
- **SUB -** subtract one value from another
- **CMP -** compare two values
- **JMP -** jump to a different part of the program
- **CALL -** call a subroutine
- **RET -** return from a subroutine
- **PUSH -** push a value onto the stack
- **POP -** pop a value off the stack



Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

In addition to the instruction mnemonic, each instruction in assembly language also includes one or more operands that specify the data to be operated on or the location of the data in memory.

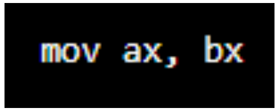
For example, the MOV instruction might have two operands that specify the source and destination of the data being moved.

It's important to note that different processors and architectures may have different sets of instruction mnemonics and operand formats.

As such, it's necessary to consult the documentation for your particular processor or assembly language to determine the correct syntax and usage of each instruction.

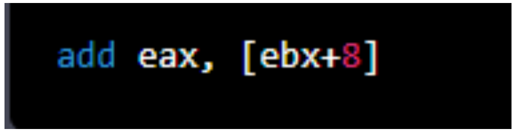
OPERANDS

The **opcode** is a short code that identifies the specific operation that the processor should perform, such as add, subtract, or move data. The **operand(s)** specify the data on which the operation should be performed, such as registers, memory addresses, or immediate values.



```
mov ax, bx
```

The **opcode** is **mov**, which instructs the processor to move data, and the **operands** are **ax** and **bx,** which specify the registers containing the data that should be moved. Here's another example:



```
add eax, [ebx+8]
```

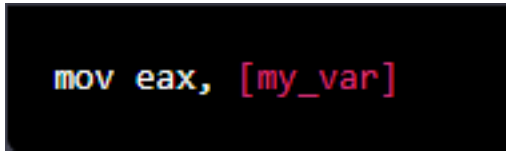
In this instruction, the **opcode is add**, which instructs the processor to add two values. The **operands are eax and [ebx+8].** eax specifies the register that will hold the result of the addition operation, while [ebx+8] specifies the memory location where the second value to be added is stored. The **+8 offset** specifies that the value is stored 8 bytes beyond the memory address contained in the ebx register.

In summary, the opcode specifies the operation to be performed, while the operand(s) provide the data to be used in that operation. Together, they form the complete instruction that is executed by the processor.

Assembly language instructions can have **between zero and three operands**, each of which can be a register, memory operand, integer expression, or input-output port. An **operand** is a value that an assembly language instruction uses as input or output.

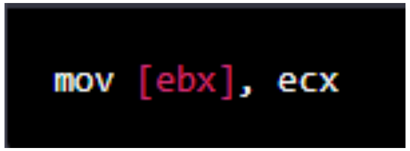
To create memory operands, you can use variable names or registers surrounded by brackets. When you use a variable name as an operand, it tells the computer to access the contents of memory at the specified address.

EG



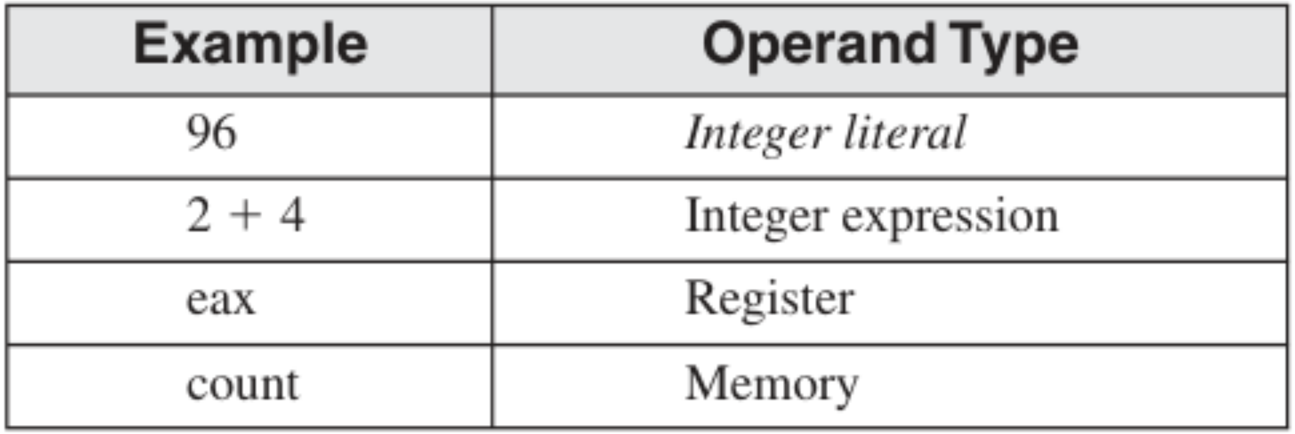
```
mov eax, [my_var]
```

In this instruction, my_var is a **variable name used as an operand.** The **brackets surrounding my_var indicate that** the contents of memory at the address specified by my_var should be loaded into the eax register. The assembler will replace my_var with the appropriate memory address during the assembly process. Similarly, you can use **registers surrounded by brackets** to create memory operands. For example, consider the following instruction:



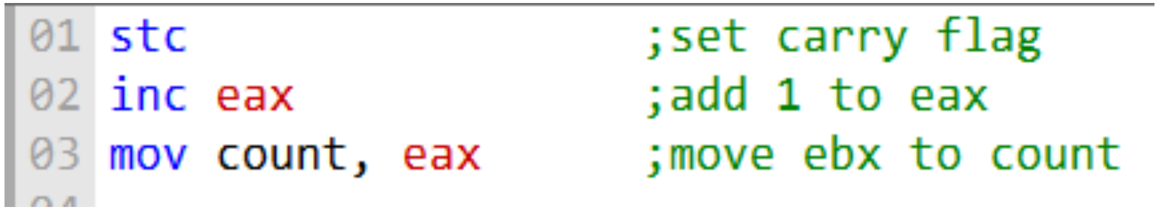
```
mov [ebx], ecx
```

In this instruction, [ebx] is a memory operand. The brackets indicate that the contents of memory at the address stored in the ebx register should be updated with the value in the ecx register. Using memory operands allows programmers to read from and write to specific memory locations, enabling the manipulation of data stored in memory.



Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Let’s look at examples of assembly language instructions having varying numbers of operands. The STC instruction, for example, has no operands:

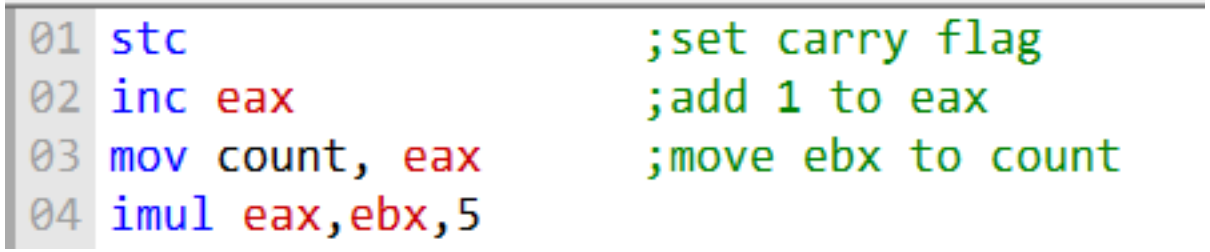


```
stc                 ;set carry flag
inc eax             ;add 1 to eax
mov count, eax      ;move ebx to count
```

There is a natural ordering of operands. When instructions have multiple operands, the first one is typically called the **destination operand**. The second operand is usually called the **source operand.**

In general, the contents of the destination operand are modified by the instruction. In a **MOV instruction**, for example, data is copied from the **source to the destination.**

The IMUL instruction has three operands, in which the first operand is the destination, and the following two operands are source operands, which are multiplied together:



```
stc                     ;set carry flag
inc eax                 ;add 1 to eax
mov count, eax          ;move ebx to count
imul eax,ebx,5
```

In this case, EBX is multiplied by 5, and the product is stored in the EAX register.

COMMENTS

Comments are an important way for the writer of a program to communicate information about the program’s design to a person reading the source code.

The following information is typically included at the top of a program listing:

- Description of the program’s purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program’s implementation Comments can be specified in two ways:
- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.
- Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. Here is an example:

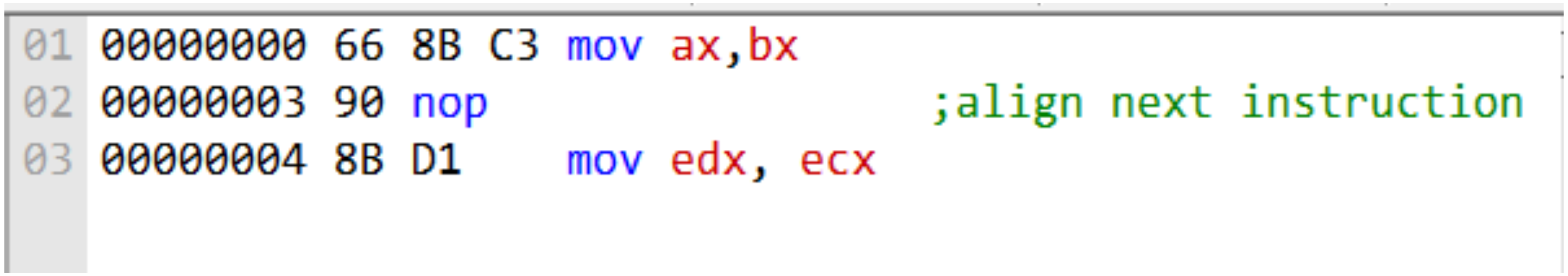
For block comments, just use semi-colon(;).

NOP OPERATION(NO OPERATION)

In assembly language programming, the NOP (no operation) instruction is an instruction that performs no operation when executed. It is a placeholder instruction that is often used for padding or alignment purposes, especially in cases where it is necessary to align code to specific memory boundaries.

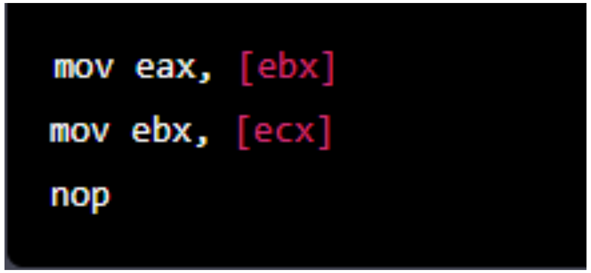
The NOP instruction takes up one byte of program storage and doesn't perform any work when executed. The instruction is useful in situations where it is necessary to maintain the length of the instruction stream, but no actual operation needs to be performed. This can happen, for example, when aligning code to specific memory boundaries that require instruction alignment.

In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):



```
00000000 66 8B C3 mov ax,bx
00000003 90 nop                        ;align next instruction
00000004 8B D1     mov edx, ecx
```

OR



```
mov eax, [ebx]
mov ebx, [ecx]
nop
```

In this example, the first two instructions move data between registers. The third instruction is a NOP instruction that does nothing, but it ensures that the address of the next instruction is aligned to a doubleword boundary. This alignment can improve the performance of the code by allowing the processor to access memory more efficiently.

In summary, the NOP instruction is a placeholder instruction that is often used for padding or alignment purposes. It takes up one byte of program storage and doesn't perform any work when executed. It is sometimes used by compilers and assemblers to align code to efficient memory boundaries, which can improve performance.

x86 PROCESSORS AND SPEED

x86 processors are designed to load code and data more quickly from even doubleword (4-byte) addresses. This is because the x86 architecture uses a memory bus that transfers data in units of 4 bytes, and loading data from even doubleword addresses ensures that the data can be transferred in a single bus cycle.

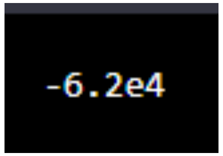
When data is loaded from an address that is not aligned to a doubleword boundary, the processor must perform two memory accesses to fetch the entire 4-byte value. This can slow down memory access and reduce the performance of the program.

To avoid this performance penalty, programmers can use techniques such as code padding or NOP instructions to ensure that code and data are aligned to doubleword boundaries. This can improve the performance of the program by allowing the processor to load and process data more efficiently.

In summary, x86 processors are designed to load code and data more quickly from even doubleword addresses, and programmers can improve performance by ensuring that code and data are aligned to these boundaries.

QUESTIONS AND ANSWERS

Write the real number –6.2 X 10^4 as a real number literal using MASM syntax.



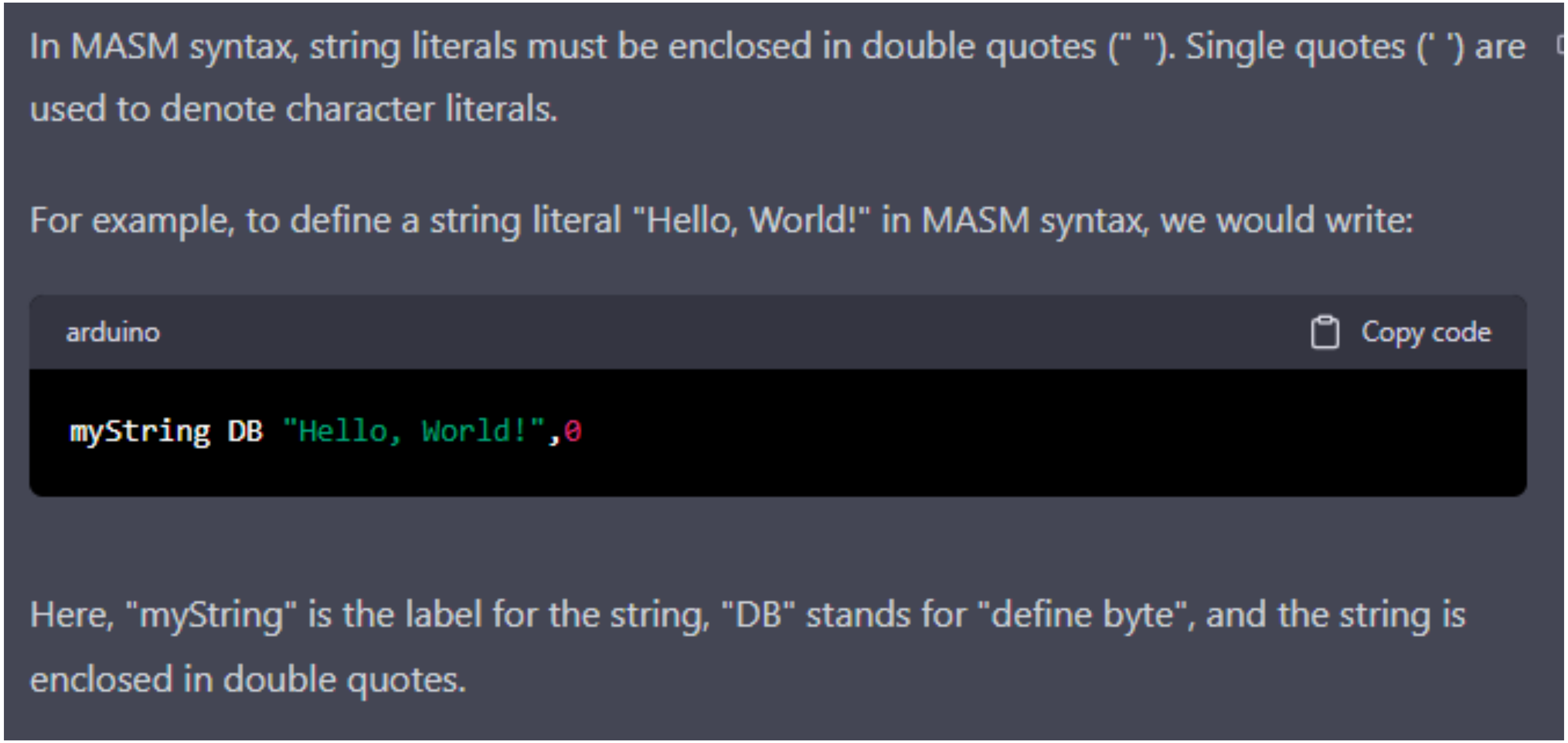
```
-6.2e4
```

****

NEXT QUESTION

****

Must string literals be enclosed in single quotes?

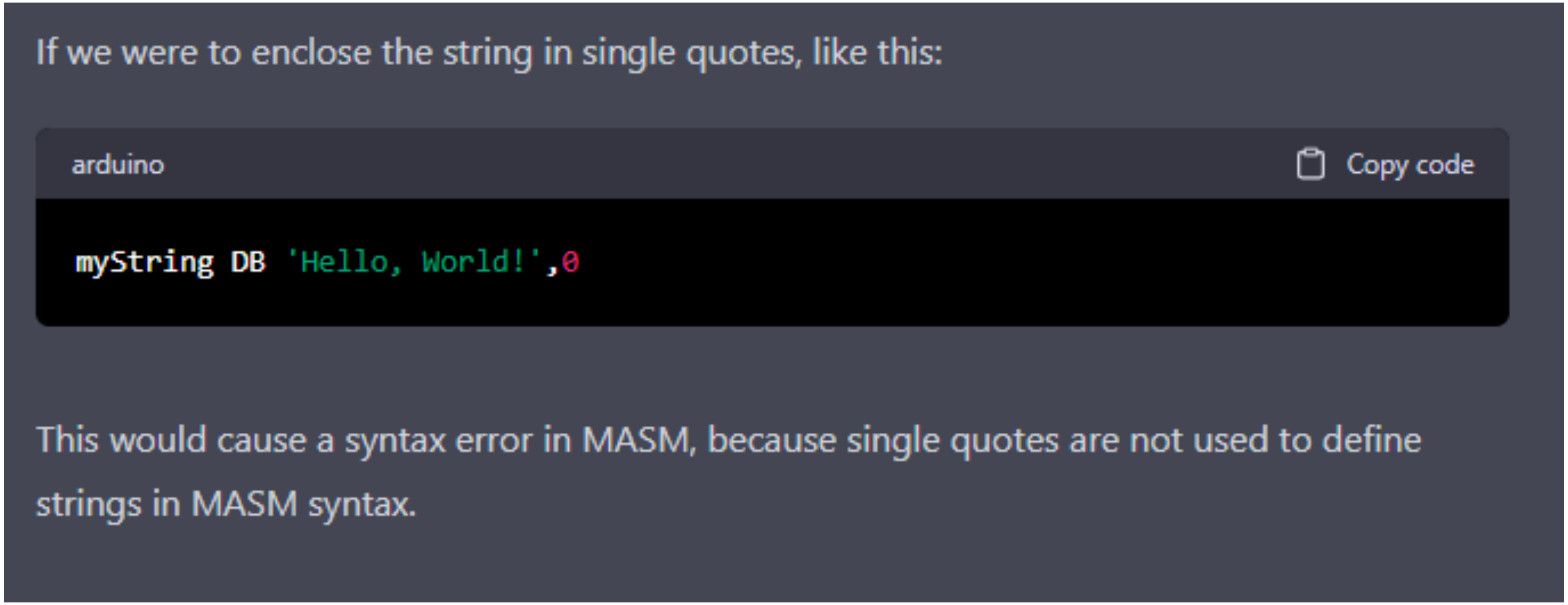


In MASM syntax, string literals must be enclosed in double quotes (" "). Single quotes (' ') are used to denote character literals.

For example, to define a string literal "Hello, World!" in MASM syntax, we would write:

```
myString DB "Hello, World!",0
```

Here, "myString" is the label for the string, "DB" stands for "define byte", and the string is enclosed in double quotes.



If we were to enclose the string in single quotes, like this:

```
myString DB 'Hello, World!',0
```

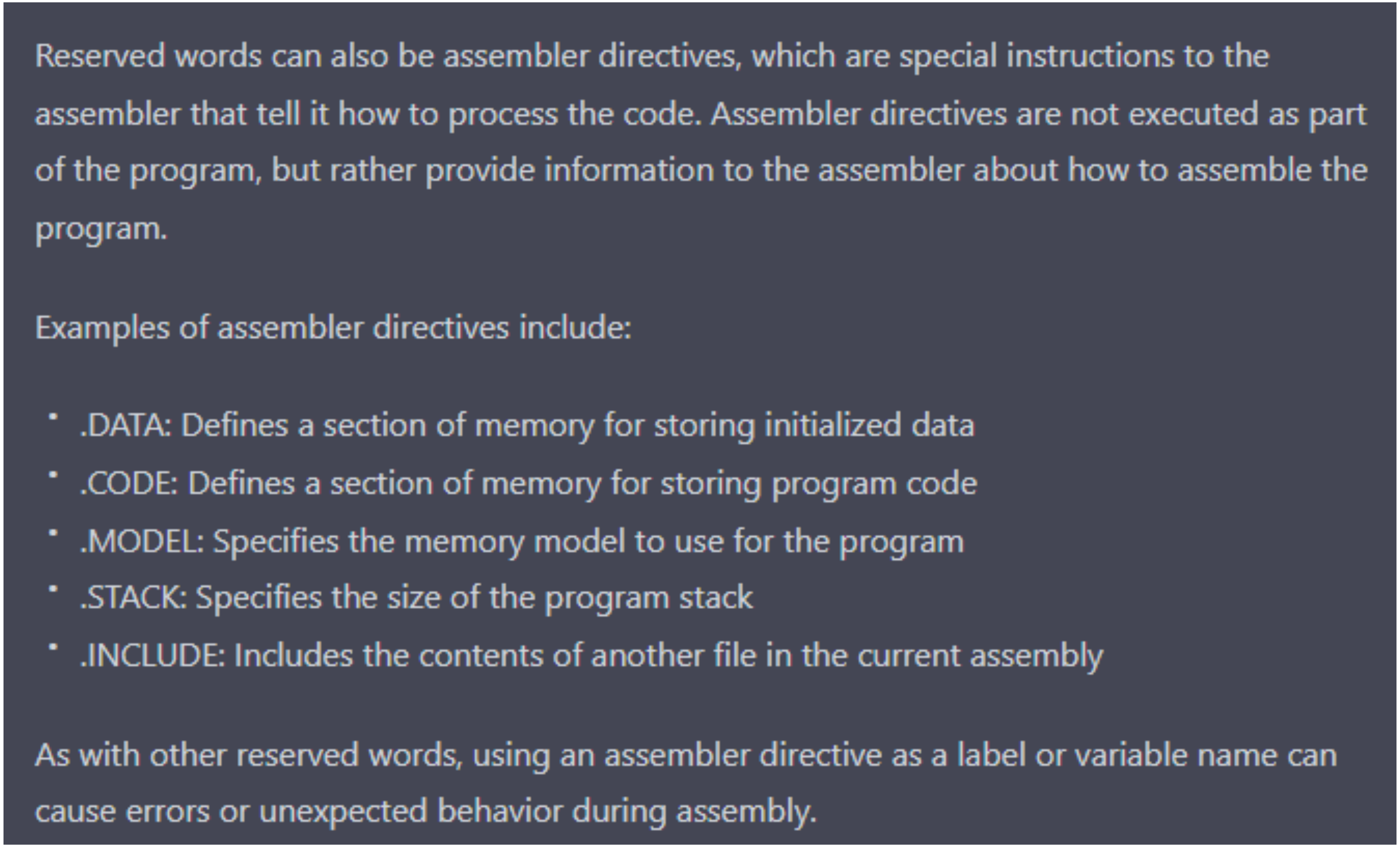
This would cause a syntax error in MASM, because single quotes are not used to define strings in MASM syntax.

****

NEXT QUESTION

****

Reserved words can be instruction mnemonics, attributes, operators, predefined symbols and ________



Reserved words can also be assembler directives, which are special instructions to the assembler that tell it how to process the code. Assembler directives are not executed as part of the program, but rather provide information to the assembler about how to assemble the program.

Examples of assembler directives include:

- .DATA: Defines a section of memory for storing initialized data
- .CODE: Defines a section of memory for storing program code
- .MODEL: Specifies the memory model to use for the program
- .STACK: Specifies the size of the program stack
- .INCLUDE: Includes the contents of another file in the current assembly

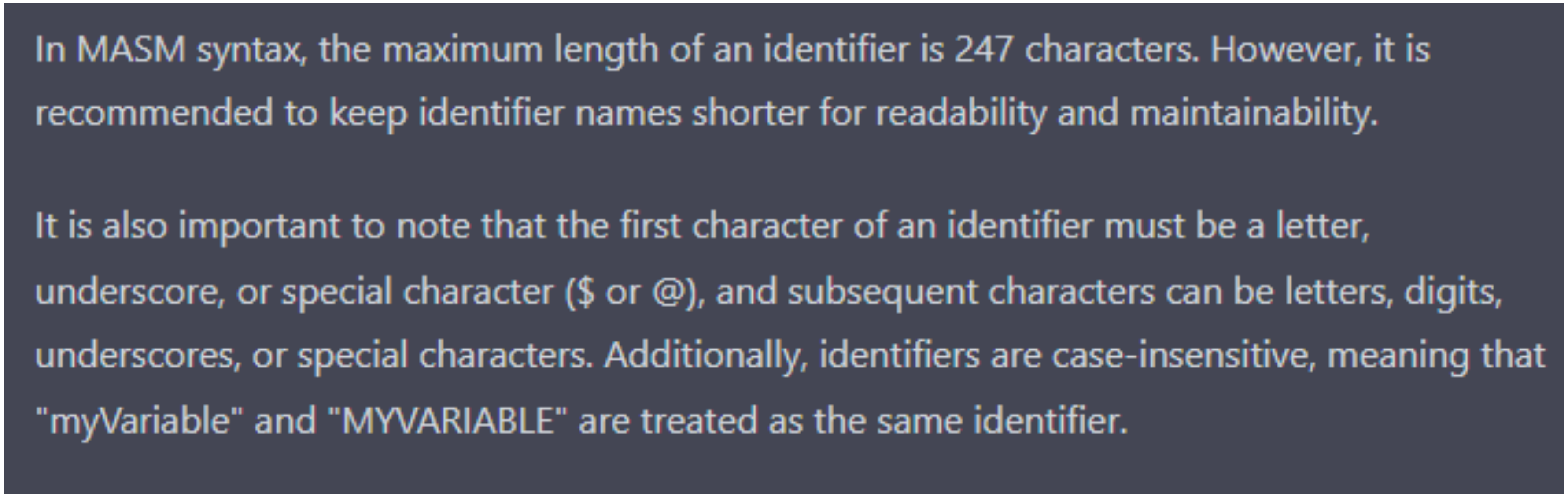
As with other reserved words, using an assembler directive as a label or variable name can cause errors or unexpected behavior during assembly.

****

NEXT QUESTION

****

What is the maximum length of an identifier?



In MASM syntax, the maximum length of an identifier is 247 characters. However, it is recommended to keep identifier names shorter for readability and maintainability.

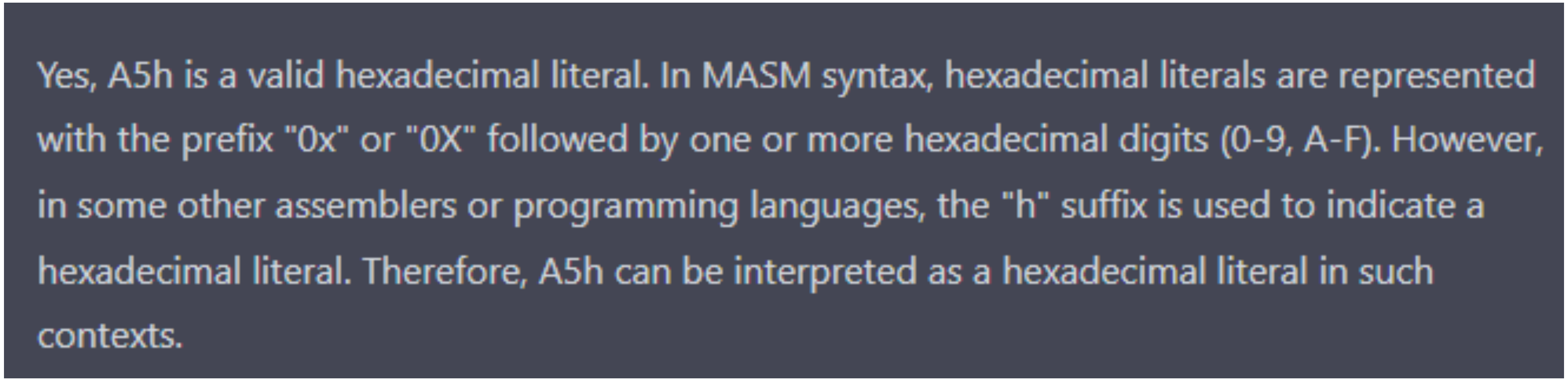
It is also important to note that the first character of an identifier must be a letter, underscore, or special character ($ or @), and subsequent characters can be letters, digits, underscores, or special characters. Additionally, identifiers are case-insensitive, meaning that "myVariable" and "MYVARIABLE" are treated as the same identifier.

****

NEXT QUESTION

****

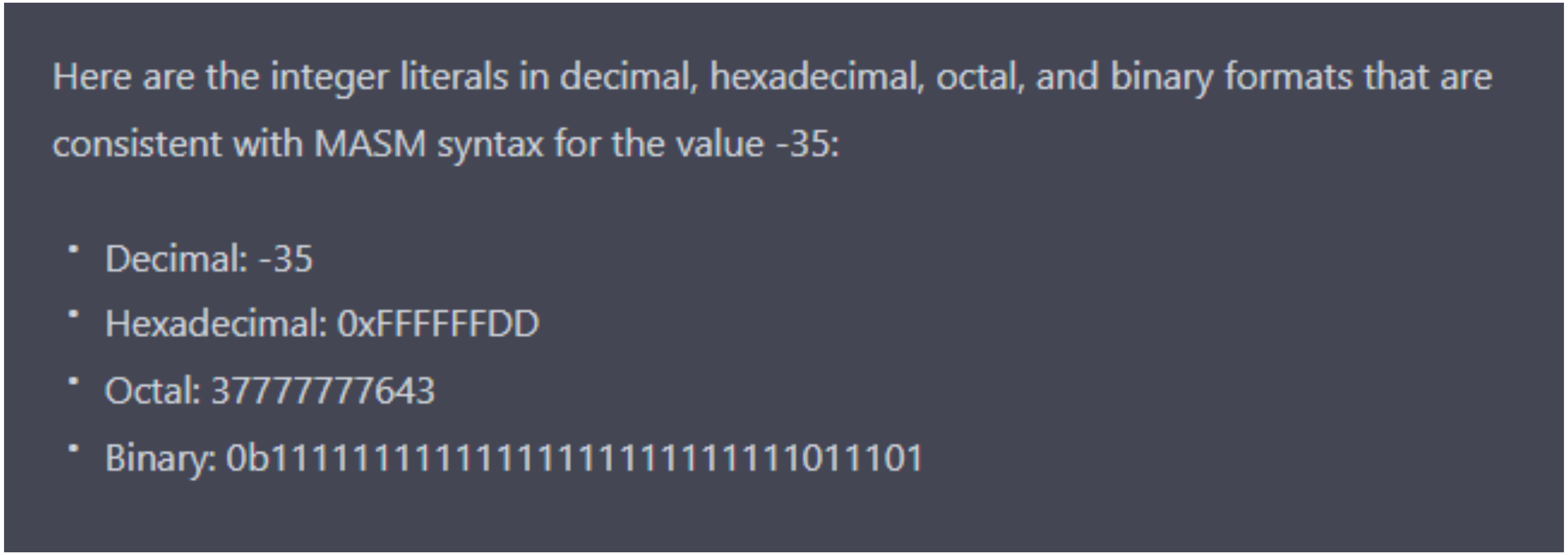
Is A5h a valid hexadecimal literal? NO, Use 0A5h



Yes, A5h is a valid hexadecimal literal. In MASM syntax, hexadecimal literals are represented with the prefix "0x" or "0X" followed by one or more hexadecimal digits (0-9, A-F). However, in some other assemblers or programming languages, the "h" suffix is used to indicate a hexadecimal literal. Therefore, A5h can be interpreted as a hexadecimal literal in such contexts.

```
***********************************************
                     NEXT QUESTION

***********************************************
```

Using the value –35, write it as an integer literal in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax.



Here are the integer literals in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax for the value -35:

- Decimal: -35
- Hexadecimal: 0xFFFFFFDD
- Octal: 37777777643
- Binary: 0b11111111111111111111111111011101

****

NEXT QUESTION

****

Explain why and what is hex with an example 05h, and why we also use ASCII?

IN ASSEMBLY LANGUAGE, I FIND IT HARD TO UNDERSTAND WHAT SOMETHING LIKE mov eax, 05H means. that hex part is really confusing. Like the same to ascii, eg) represented by a certain hex. I know ascii is a mapping of keyboard characters to respective numbers coz computers can only read numbers.

I can understand how hexadecimal notation can be confusing at first, especially if you're not used to working with it. Let me try to explain it in a way that might help.

In assembly language, **mov eax, 05h** means "move the value 5 (in hexadecimal notation) into the EAX register". The "h" at the end indicates that the number is written in hexadecimal notation. Hexadecimal notation is a base-16 number system, which means that it uses 16 digits instead of the 10 digits used in the decimal (base-10) number system. The digits in hexadecimal notation are 0-9 and A-F, where A represents the decimal value of 10, B represents 11, and so on up to F which represents the decimal value of 15.

So in the example **mov eax, 05h**, the value 5 in hexadecimal notation is equivalent to the decimal value 5. Similarly, the value FFh in hexadecimal notation is equivalent to the decimal value 255. In assembly language, hexadecimal notation is often used because it is more compact and easier to read than binary notation.

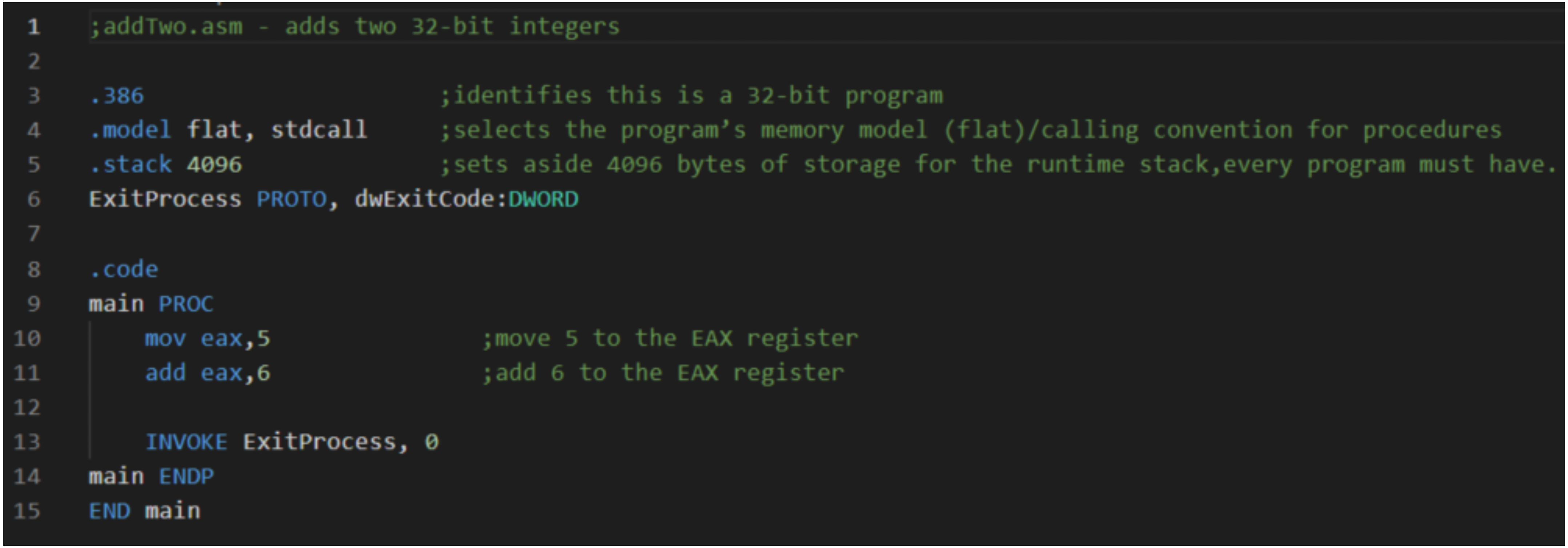
Regarding **ASCII,** you're correct that it is a mapping of keyboard characters to respective numbers. Each character in the ASCII table is assigned a unique number, or ASCII code, which is represented in hexadecimal notation. For example, the ASCII code for the character ")" is 29h in hexadecimal notation. When you want to display this character on a computer screen, the computer looks up the ASCII code for ")" and displays the corresponding character.

****

NEXT QUESTION

****

ADDING AND SUBTRACTING

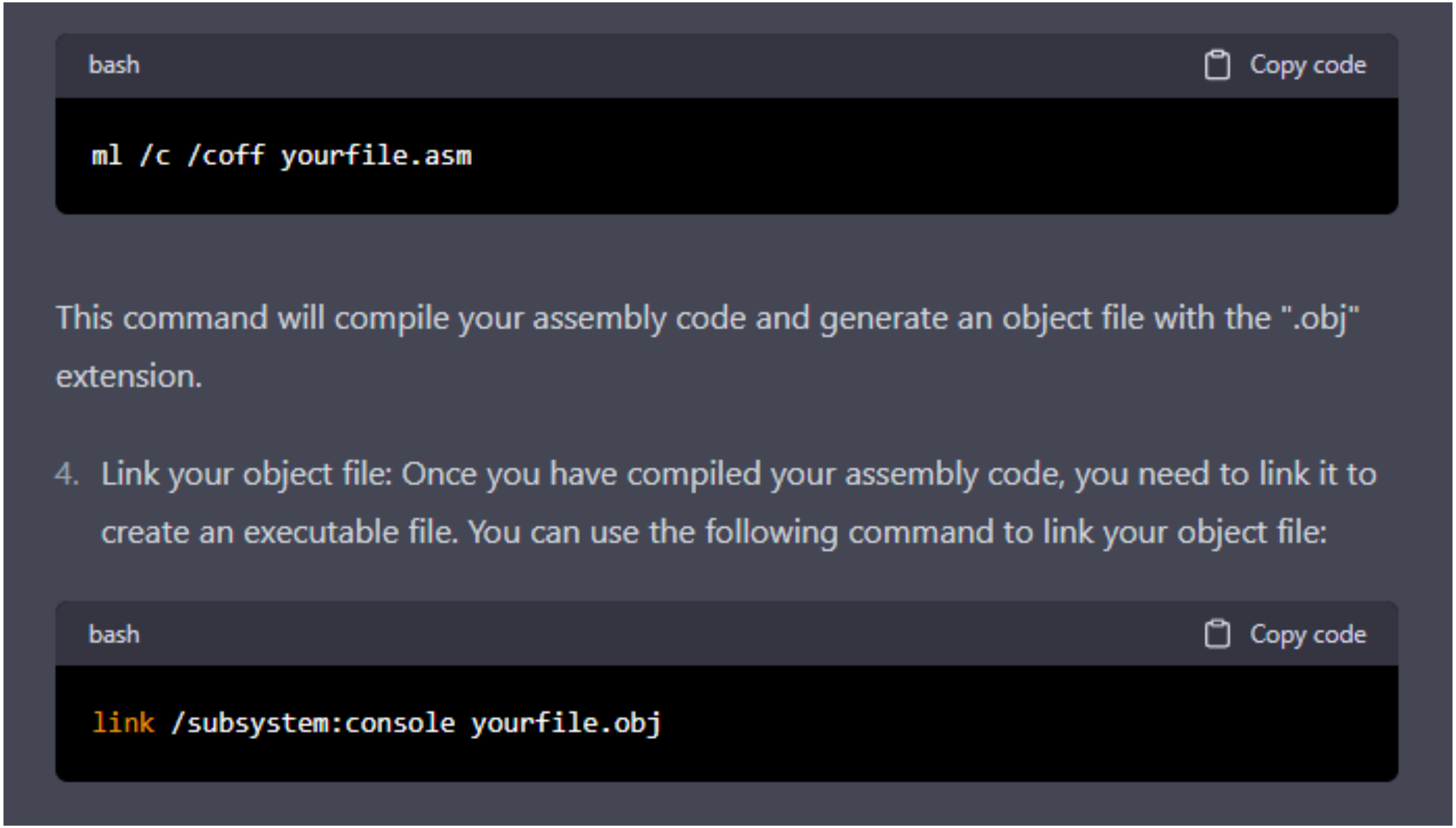


```
 1  ;addTwo.asm - adds two 32-bit integers
 2
 3  .386                     ;identifies this is a 32-bit program
 4  .model flat, stdcall     ;selects the program’s memory model (flat)/calling convention for procedures
 5  .stack 4096              ;sets aside 4096 bytes of storage for the runtime stack,every program must have.
 6  ExitProcess PROTO, dwExitCode:DWORD
 7
 8  .code
 9  main PROC
10      mov eax,5               ;move 5 to the EAX register
11      add eax,6               ;add 6 to the EAX register
12
13      INVOKE ExitProcess, 0
14  main ENDP
15  END main
```

Line 4 contains the **.386 directive,** which identifies this as a 32-bit program that can access 32-bit registers and addresses.

Line 5 **selects the program’s memory model (flat)**, and identifies the calling convention (named stdcall) for procedures. The **"flat" memory model** is a memory model where all memory is treated as a single, contiguous address space. In a flat memory model, the program sees all memory locations as a single, linear address space, regardless of the physical organization of the memory. This means that the program can access any memory location directly, without the need for special instructions or addressing modes.We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how stdcall works.)

Line 6 **sets aside 4096 bytes of storage** for the runtime stack, which every program must have.



```bash
ml /c /coff yourfile.asm
```

This command will compile your assembly code and generate an object file with the ".obj" extension.

4. Link your object file: Once you have compiled your assembly code, you need to link it to create an executable file. You can use the following command to link your object file:

```bash
link /subsystem:console yourfile.obj
```

In line 7 of the code, a **prototype** for the ExitProcess function is declared. This function is a standard Windows service that is used to exit a program. A **prototype** is a declaration of the function's name and input parameters, which allows the assembler to generate correct machine code for the function call.

The input parameter for **ExitProcess** is called **dwExitCode**, which is like a return value that is passed back to the operating system. A return value of zero indicates that the program was successful, while any other integer value generally indicates an error code number. This helps the operating system to understand the outcome of the program's execution.

You can think of your assembly programs as subroutines or processes that are called by the operating system. When your program is ready to finish, it calls the ExitProcess function and returns an integer that tells the operating system about the program's success or failure. This way, the operating system can manage the execution of multiple programs and handle any errors that might occur.

In the context of the previous explanation, system administrators often create script files that execute a series of programs in sequence. At each point in the script, they need to know if the most recently executed program has failed or succeeded, so they can determine the next step to take.

To achieve this, the operating system provides a mechanism for programs to indicate whether they completed successfully or not. This is typically done by returning an integer value from the program, where a value of 0 indicates success, and any non-zero value indicates an error code.

In a script file, system administrators can use the ErrorLevel variable to check the return code of the previous command or program. For example, if a program returns a non-zero value indicating an error, the ErrorLevel variable will be set to 1, and the script can be set up to exit or take appropriate action based on this value.

In summary, by using the return value mechanism provided by the operating system, programs can indicate whether they completed successfully or not, and system administrators can use this information to automate and manage the execution of a sequence of programs in a script file.

CODE EXPLAINED

.MODEL directive tells the assembler which memory model to use. In 32-bit programs, we always use the flat memory model, which is associated with the processor’s protected mode. The **stdcall keyword** tells the assembler how to manage the runtime stack when procedures are called. That’s a complicated subject that we will address in Chapter 8.

In Assembly language, the **ExitProcess symbol** is usually defined using the **PROTO directive,** which specifies the function prototype. The ExitProcess function is a Win32 API function that terminates the calling process and returns the specified exit code to the operating system.

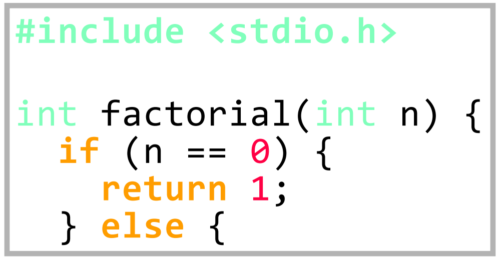
The **PROTO directive** is used to declare the function prototype and its parameter list. In this case, ExitProcess PROTO, dwExitCode:DWORD indicates that ExitProcess is a function that takes one parameter, dwExitCode, which is a 32-bit unsigned integer.

Next, the **.STACK directive** tells the assembler how many bytes of memory to reserve for the program’s runtime stack. The value 4096 is probably more than we will ever use, but it happens to correspond to the size of a memory page in the processor’s system for managing memory.

All modern programs use a **stack** when calling subroutines—first, to **hold passed parameters**, and second, to **hold the address of the code that called the function.**

****

Let's see the stack example in C:



```
#include <stdio.h>

int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}

int main() {
  int result = factorial(5);
  printf("The factorial of 5 is %d\n", result);
  return 0;
}
```

When the main() function calls the factorial() function, the following steps occur:

1. The stack pointer is decremented to make room for the return address.
2. The return address is pushed onto the stack.
3. The parameters to the factorial() function are pushed onto the stack.
4. The program counter is set to the address of the first instruction in the factorial() function.
5. The factorial() function executes.
6. When the factorial() function returns, the following steps occur:

- The parameters to the factorial() function are popped off the stack.
- The return address is popped off the stack and used to set the program counter.
- The stack pointer is incremented to restore the original stack frame.

The following diagram shows the stack before and after the main() function calls the factorial() function:

Stack before calling factorial():

```
| Return address |
| Parameter 1    |
| Parameter 2    |
| .............. |
```

Stack after calling factorial():

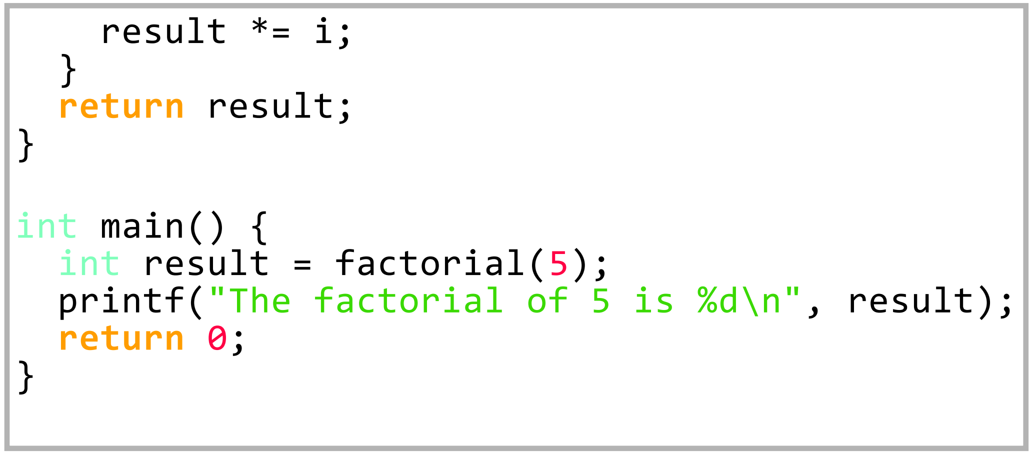
```
| Return address for factorial() |
| Parameter 1 for factorial()    |
| Parameter 2 for factorial()    |
| Return address for main()              |
| Parameter 1 for main()                 |
| Parameter 2 for main()                 |
| ...............................|
```

In addition to holding passed parameters and the return address, the stack can also be used to store local variables. Local variables are variables that are declared inside a function.

When a function is called, a new stack frame is created to store the local variables for that function call. When the function returns, the stack frame is popped off the stack and the local variables are destroyed.

Here is an example of how to use the stack to store local variables:

```
#include <stdio.h>

int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; i++) {
```



```
    result *= i;
  }
  return result;
}

int main() {
  int result = factorial(5);
  printf("The factorial of 5 is %d\n", result);
  return 0;
}
```

In this example, the factorial() function declares a local variable called result. This variable is stored on the stack while the factorial() function is executing.

When the factorial() function returns, the variable result is destroyed.

Stack usage is an important part of C programming. By understanding how the stack works, you can write more efficient and reliable code.

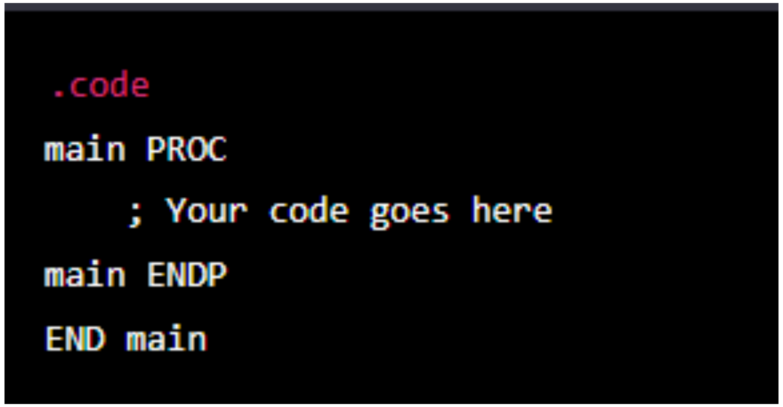
****

The CPU uses this address to return when the function call finishes, back to the spot where the function was called. In addition, the runtime stack can hold local variables, that is, variables declared inside a function.

The **.CODE directive** marks the beginning of the code area of a program, the area that contains executable instructions. Usually the next line after .CODE is the declaration of the program’s entry point, and by convention, it is usually a procedure named main. The entry point of a program is the location of the very first instruction the program will execute. We used the following lines to convey this information:

The **ENDP directive** marks the end of a procedure. Our program had a procedure named main, so the endp must use the same name.

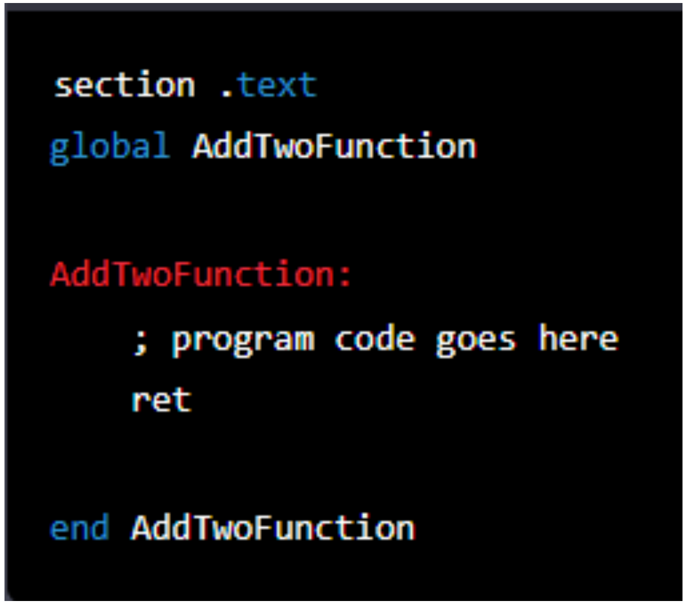
Finally, the **END directive** marks the end of the program, and references the program entry point:



```
 .code
main PROC
    ; Your code goes here
main ENDP
END main
```

The **"END" directive** in assembly language is a marker that indicates where the program ends, and the **"_start"** or **“main”** label specifies the entry point where the program execution begins.

The name used with the "END" directive can be any valid label or symbol in your assembly code. It does not have to be "main" or "_start". You can use any label or symbol that refers to the location of the last instruction in your code.



```
 section .text
global AddTwoFunction

AddTwoFunction:
    ; program code goes here
    ret

end AddTwoFunction
```

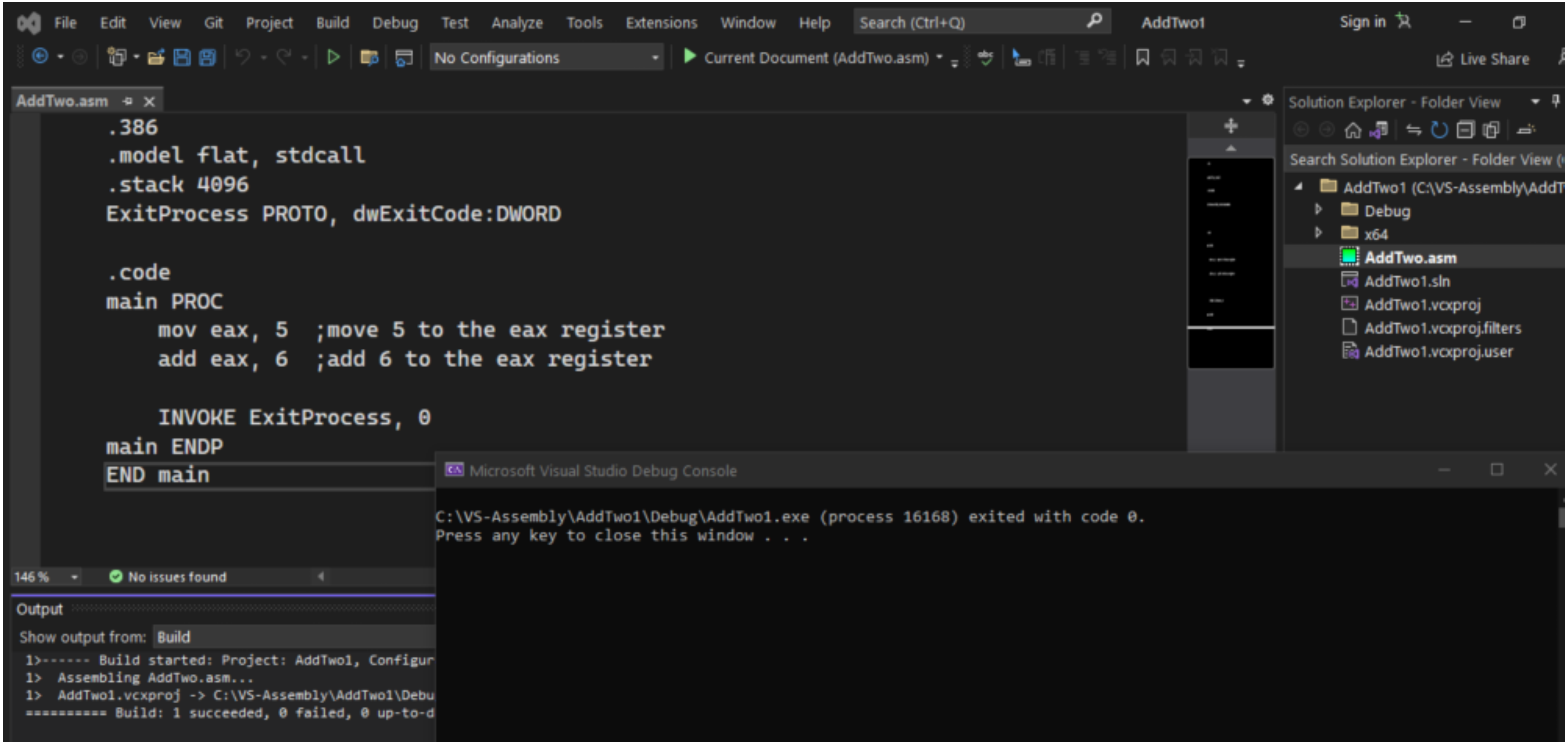
The END directive, on the other hand, marks the end of the entire program, not just the "main" procedure. It typically appears at the end of the source file and **references the program entry point**, which is typically the "main" function in a C or C++ program.

So, in summary, the "main ENDP" directive marks the end of the "main" procedure, while the "END main" directive marks the end of the entire program and references the program entry point, which is typically the "main" function. While the names are the same, they refer to different entities in the program.

NB:
If you add any more lines to a program after the END directive, they will be ignored by the assembler. You can put anything there—program comments, copies of your code, etc.—it doesn’t matter.

RUNNING AND DEBUGGING

It ran successfully:



```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    mov eax, 5  ;move 5 to the eax register
    add eax, 6  ;add 6 to the eax register

    INVOKE ExitProcess, 0
main ENDP
END main
```

```
C:\VS-Assembly\AddTwo1\Debug\AddTwo1.exe (process 16168) exited with code 0.
Press any key to close this window . . .
```

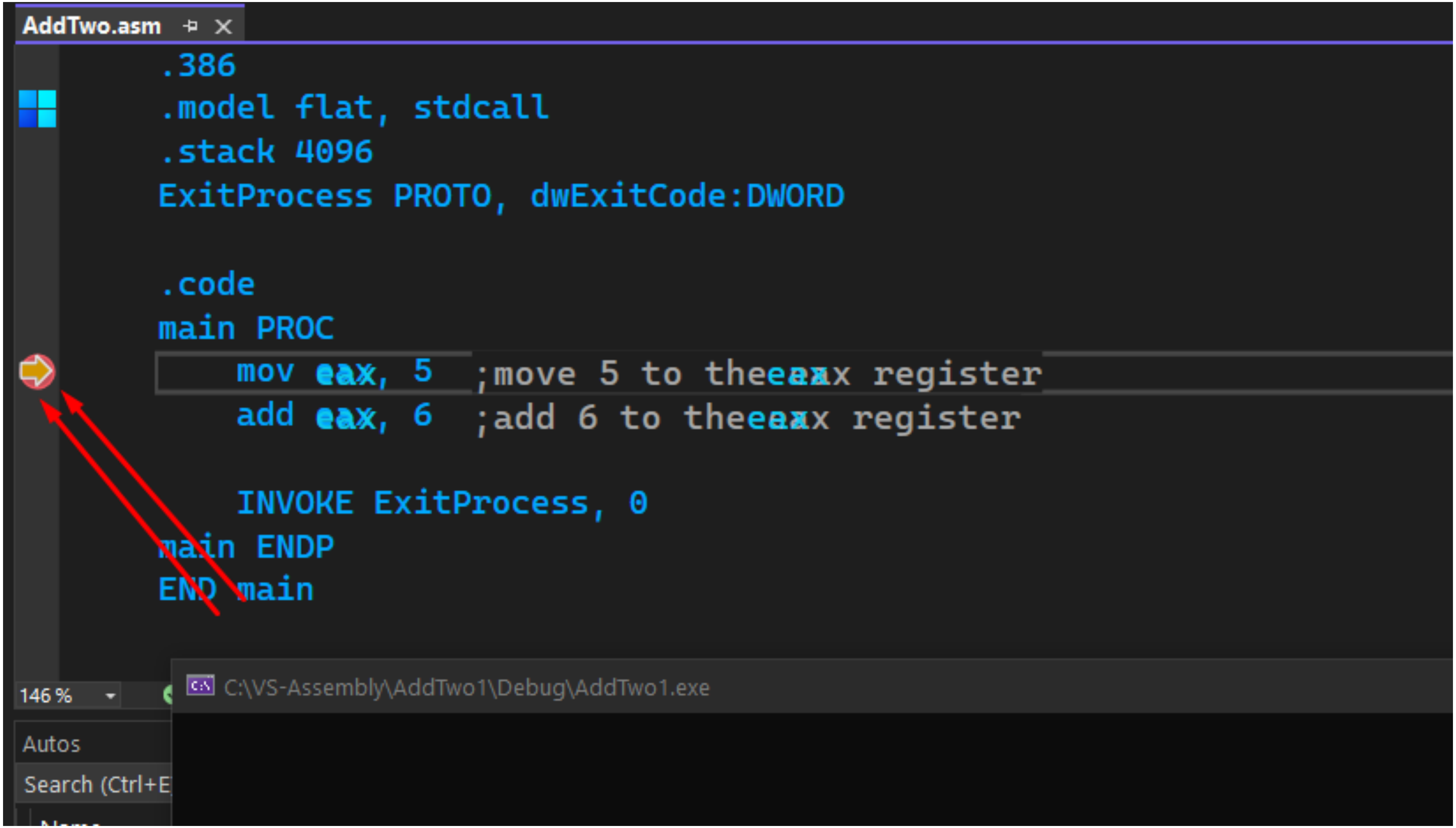
```
1>------ Build started: Project: AddTwo1, Configur
1>  Assembling AddTwo.asm...
1>  AddTwo1.vcxproj -> C:\VS-Assembly\AddTwo1\Debu
========== Build: 1 succeeded, 0 failed, 0 up-to-d
```

Just make sure the project is linked with masm and the files you run are in C drive together with masm.

Another way to start a debugging session is to **set a breakpoint** on a program statement by clicking the mouse in the vertical gray bar just to the left of the code window. A large red dot will mark the breakpoint location. Then you can run the program by selecting Start Debugging from the Debug menu.

Tip: If you try to set a breakpoint on a non-executable line, Visual Studio will just move the breakpoint forward to the next executable line when you run the program.

Memory window, registers etc have been removed in VS Community 2022.



```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    mov eax, 5    ;move 5 to the eax register
    add eax, 6    ;add 6 to the eax register

    INVOKE ExitProcess, 0
main ENDP
END main
```

Figure 3-3 shows the program at the start of a debugging session.

A breakpoint was set on Line 11, the first MOV instruction, and the debugger has paused on that line.

The line has not executed yet.

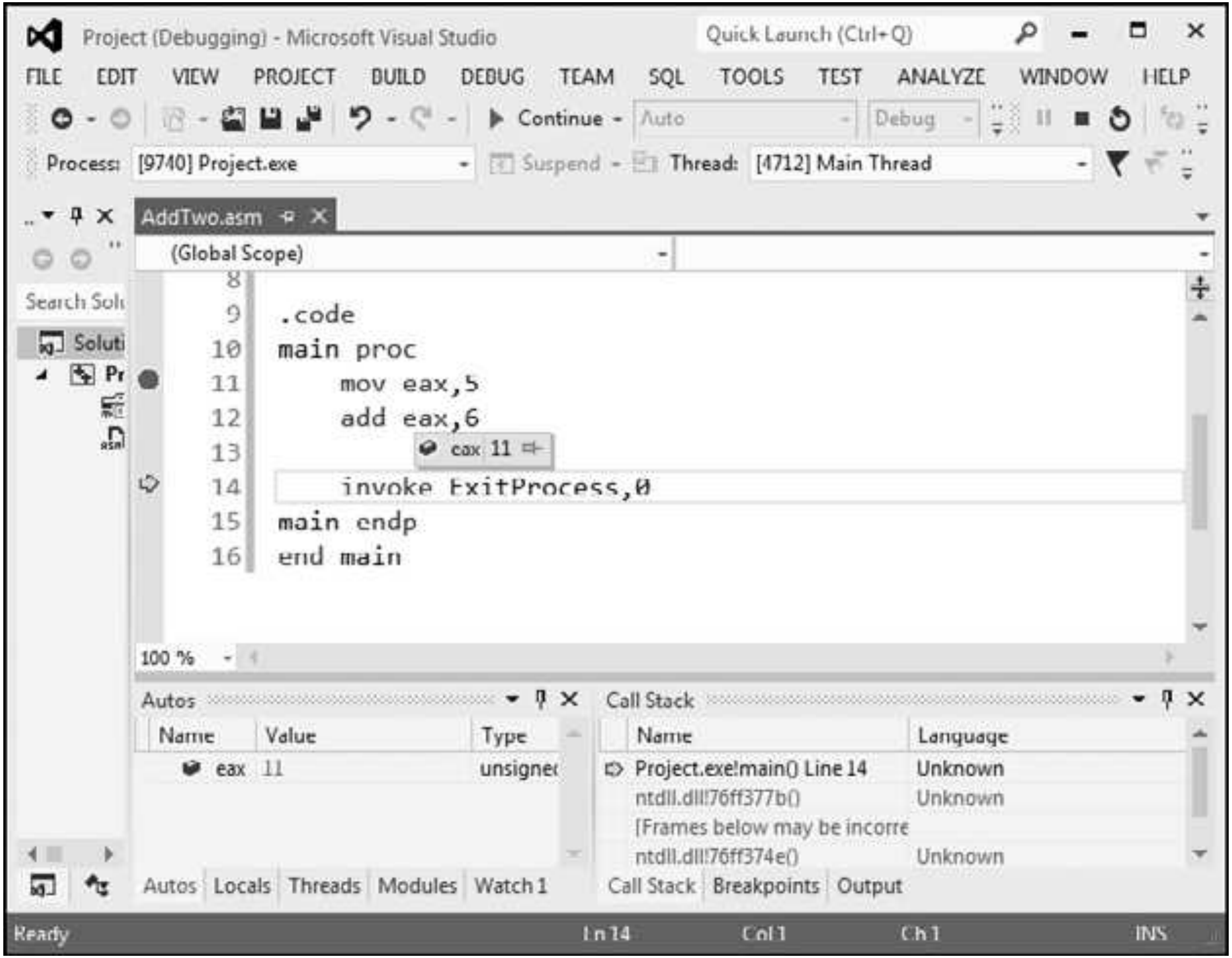
When the debugger is active, the bottom status line of the Visual Studio window turns orange.

When you stop the debugger and return to edit mode, the status line turns blue.

The visual cue is helpful because you cannot edit or save a program while the debugger is running.

Figure 3-4 shows the debugger after the user has stepped through lines 11 and 12, and is paused on line 14.

By hovering the mouse over the EAX register name, we can see its current contents (11).


Project (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Continue
Auto
Debug
Process: [9740] Project.exe
Suspend
Thread: [4712] Main Thread
AddTwo.asm
(Global Scope)
Search Solu
Soluti
Pr
cax 11
100 %
Autos Locals Threads Modules Watch 1
Call Stack Breakpoints Output
Ready
Ln 14
Col 1
Ch 1
INS

```
.code
main proc
    mov eax,5
    add eax,6

    invoke ExitProcess,0
main endp
end main
```

Autos

Name	Value	Type
eax	11	unsigne

Call Stack

Name	Language
Project.exe!main() Line 14	Unknown
ntdll.dll!76ff377b()	Unknown
[Frames below may be incorre	
ntdll.dll!76ff374e()	Unknown

We can then finish the program execution by clicking the Continue button on the toolbar, or by clicking the red Stop Debugging button (on the right side of the toolbar).

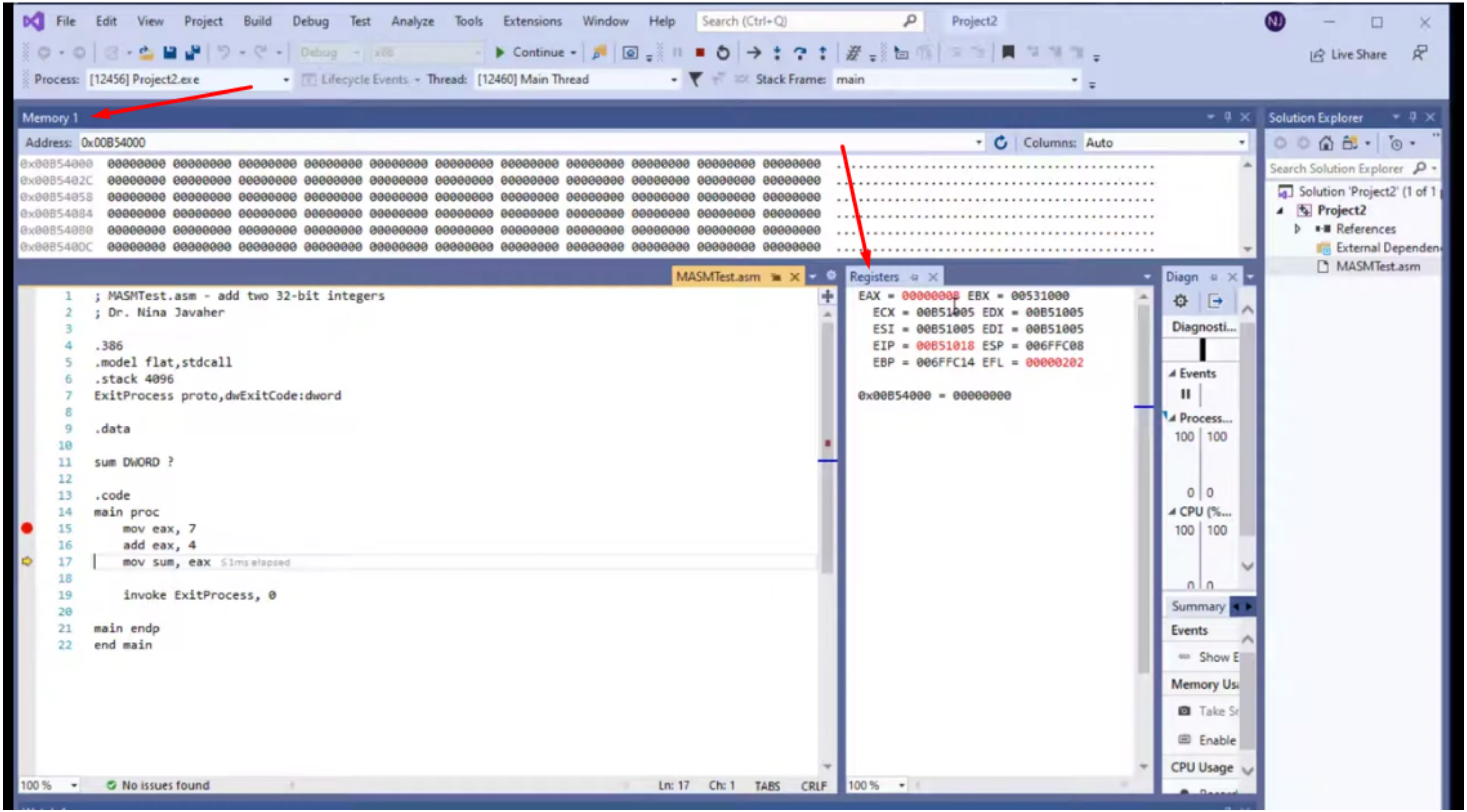
clicking the red Stop Debugging button (on the right side of the toolbar).

Customizing the Debugging

You can customize the debugging interface while it is running. For example, you might want to display the CPU registers; to do this, select Windows from the Debug menu, and then select Registers.

Figure 3-5 shows the same debugging session we used just now, with the Registers window visible. We also closed some other nonessential windows. The value shown in EAX, 0000000B, is the hexadecimal representation of 11 decimal.

Works in old VS Community old versions only.



Process: [12456] Project2.exe Thread: [12460] Main Thread Stack Frame: main

Memory 1

Address: 0x00B54000 Columns: Auto

```
0x00B54000  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
0x00B5402C  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
0x00B54058  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
0x00B54084  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
0x00B540B0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
0x00B540DC  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ............................................
```

MASMTest.asm

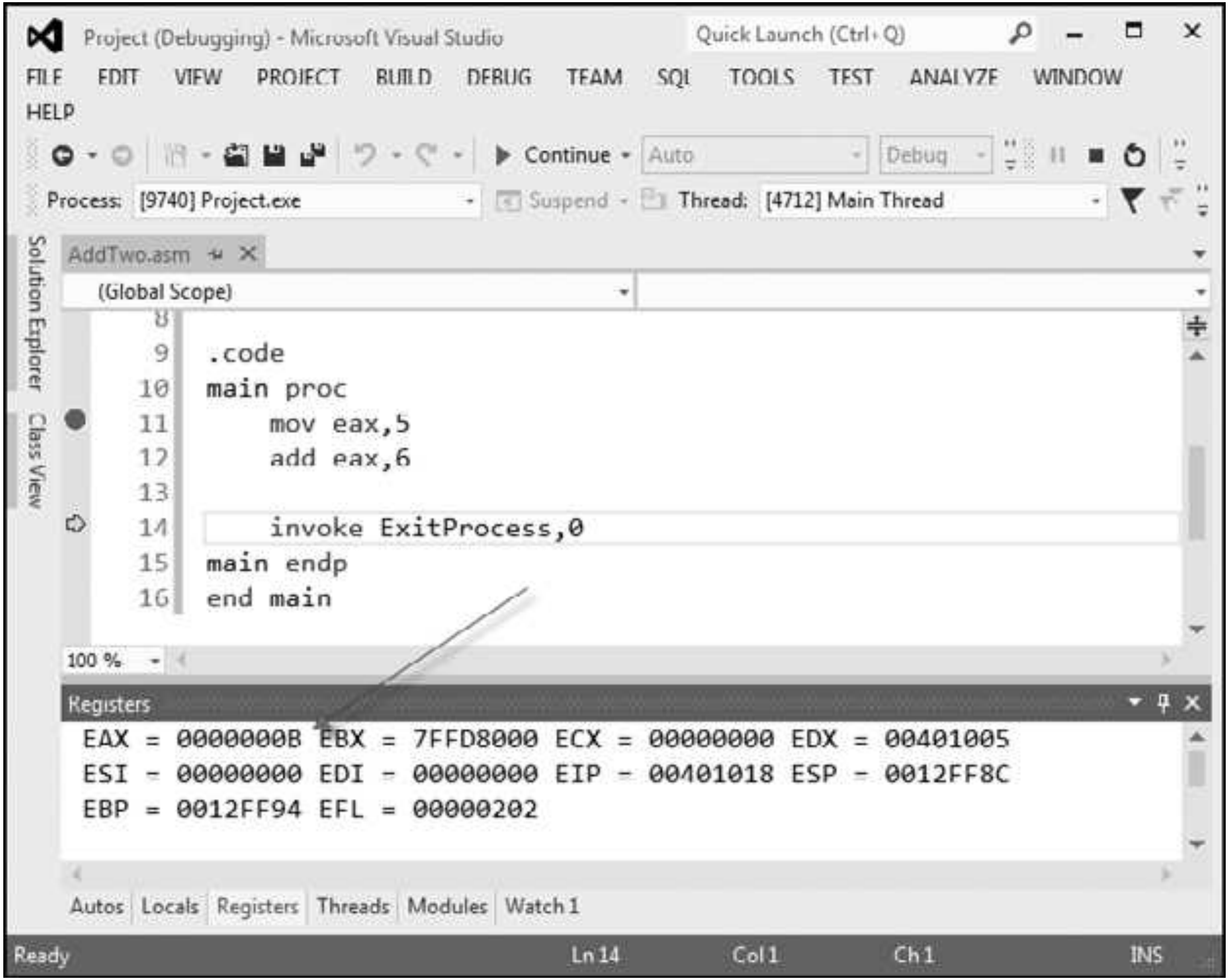
```
; MASMTest.asm - add two 32-bit integers
; Dr. Nina Javaher

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.data

sum DWORD ?

.code
main proc
    mov eax, 7
    add eax, 4
    mov sum, eax  ≤1ms elapsed

    invoke ExitProcess, 0

main endp
end main
```

Registers

```
EAX = 0000000B EBX = 00531000
  ECX = 00B51005 EDX = 00B51005
  ESI = 00B51005 EDI = 00B51005
  EIP = 00B51018 ESP = 006FFC08
  EBP = 006FFC14 EFL = 00000202

0x00B54000 = 00000000
```

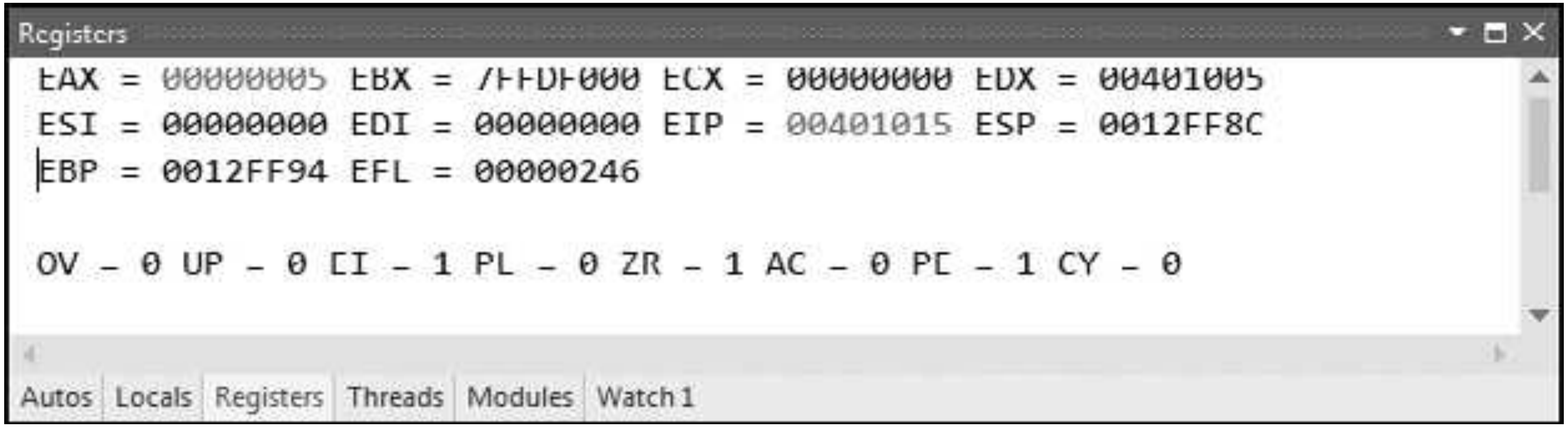
In the Registers window, the EFL register contains all the status flag settings (Zero, Carry, Overflow, etc.). If you right-click the Registers window and select Flags from the popup menu, the window will display the individual flag values. Figure 3-6 shows an example, where the flag values from left to right are: OV (overflow flag), UP (direction flag), EI (interrupt flag), PL (sign flag), ZR (zero flag), AC (auxiliary carry), PE (parity flag), and CY (carry flag).

The precise meaning of these flags will be explained in Chapter 4. One of the great things about the Registers window is that as you step through a program, any register whose value is changed by the current instruction will turn red. Although we cannot show it on the printed page (which is black and white), the red highlighting really jumps out at you, to let you know how your program is affecting the registers.



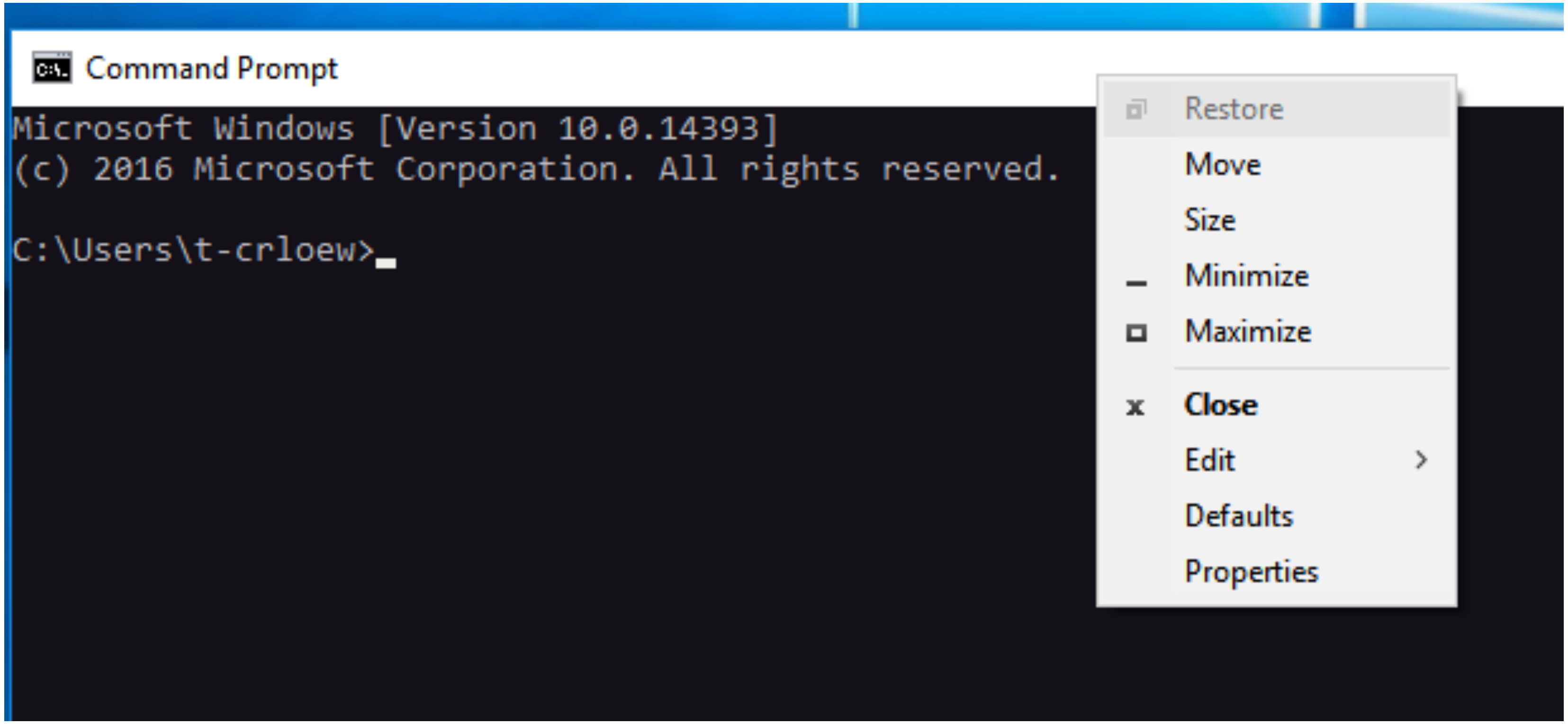
```
.code
main proc
    mov eax,5
    add eax,6

    invoke ExitProcess,0
main endp
end main
```

```
EAX = 0000000B EBX = 7FFD8000 ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000 EIP = 00401018 ESP = 0012FF8C
EBP = 0012FF94 EFL = 00000202
```



```
EAX = 00000005 EBX = 7FFDF000 ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000 EIP = 00401015 ESP = 0012FF8C
EBP = 0012FF94 EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0
```

When you run an assembly language program inside Visual Studio, it launches inside a console window.



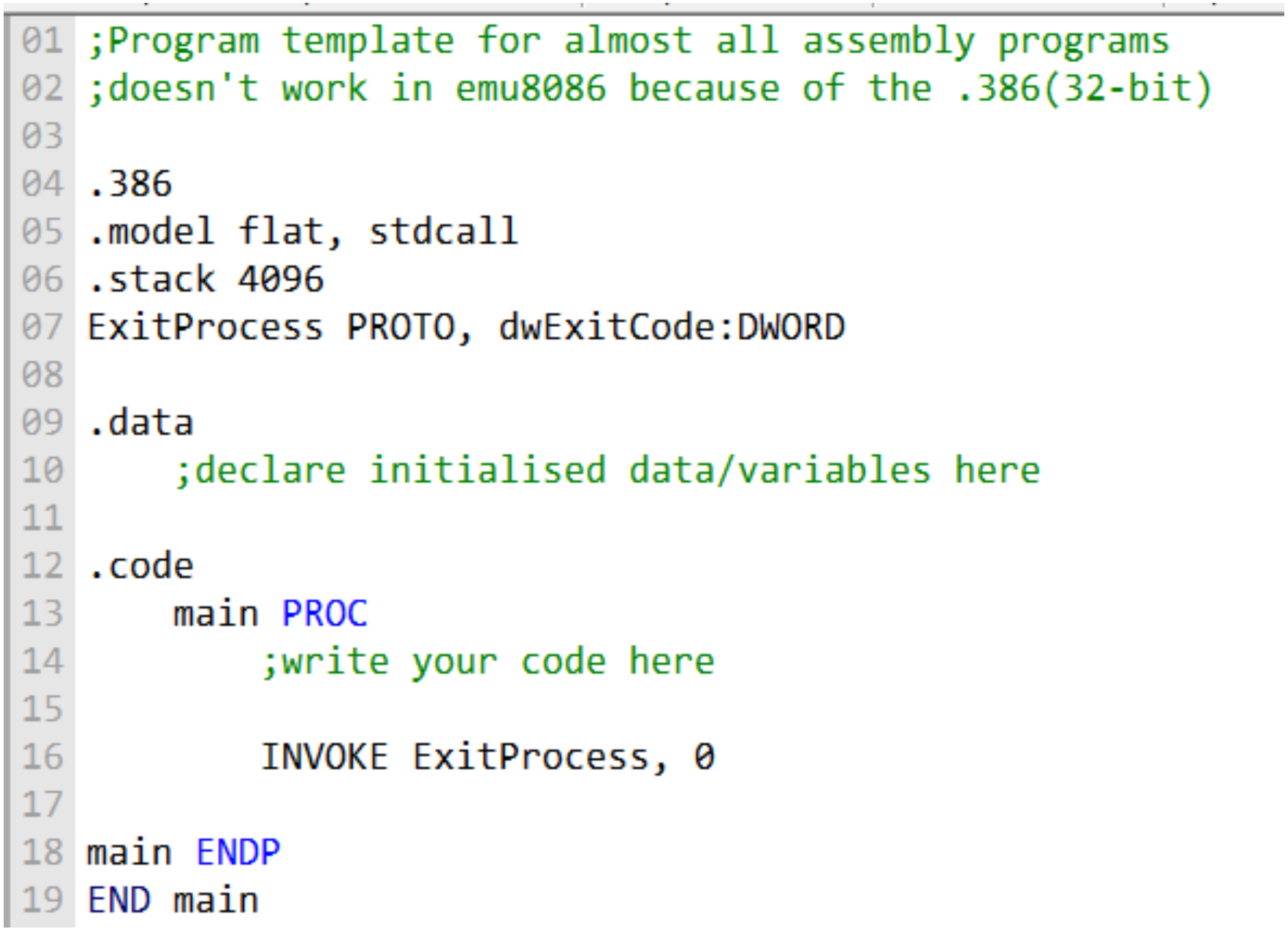
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\t-crloew>_
```

This is the same window you see when you run the program named cmd.exe from the Windows Start menu. Alternatively, you could open up a command prompt in the project's Debug\Bin folder and run the application directly from the command line.

If you did this, you would only see the program's output, which consists of text written to the console window. Look for an executable filename having the same name as your Visual Studio project. 3.2.3 Program Template Assembly language programs have a simple structure, with small variations.

When you begin a new program, it helps to start with an empty shell program with all basic elements in place. You can avoid redundant typing by filling in the missing parts and saving the file under a new name.

The following program (Template.asm) can easily be customized. Note that comments have been inserted, marking the points where your own code should be added. Capitalization of keywords is optional:

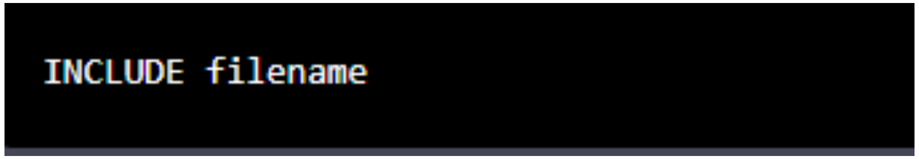


```
;Program template for almost all assembly programs
;doesn't work in emu8086 because of the .386(32-bit)

.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
    ;declare initialised data/variables here

.code
    main PROC
        ;write your code here

        INVOKE ExitProcess, 0

main ENDP
END main
```

Use Comments - It’s a very good idea to include a program description, the name of the program’s author, creation date, and information about subsequent modifications. Documentation of this kind is useful to anyone who reads the program listing (including you, months or years from now).

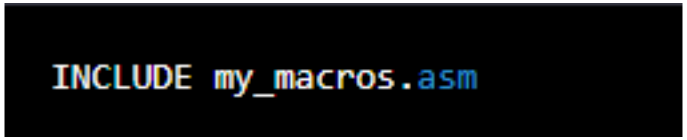
Many programmers have discovered, years after writing a program, that they must become reacquainted with their own code before they can modify it. If you're taking a programming course, your instructor may insist on additional information.

In assembly language, the **INCLUDE directive** is used to include the contents of an external file into the source code being assembled.



```
INCLUDE filename
```

Here, filename is the name of the file to be included. The file may have an extension specific to the assembler being used, such as **.inc** or **.asm.** For example, suppose you have a file named **my_macros .asm** that contains some commonly used macros. You can include this file in your main assembly file using the INCLUDE directive like this:



```
INCLUDE my_macros.asm
```

ASSEMBLE, LINKING AND RUNNING PROGRAMS

- A source program written in assembly language cannot be executed directly on its target computer.

- It must be translated or assembled into executable code using an assembler.

- The assembler produces an **object file, which contains machine language.**

- The object file must be passed to a linker which produces an executable file that is ready to execute from the operating system's command prompt.

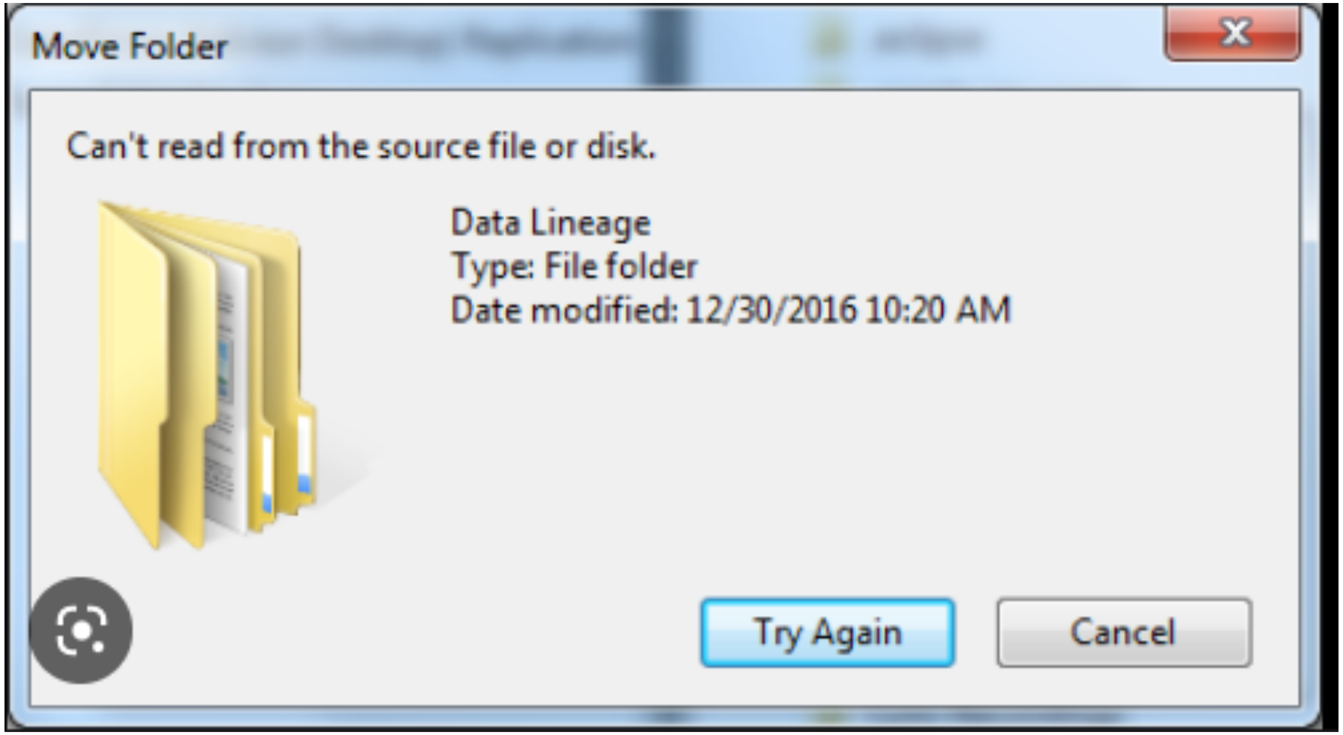
- The process of editing, assembling, linking, and executing assembly language programs involves the following steps:

Step 1: The programmer uses a text editor to create an ASCII text file named the source file.

```
## What is Svelte?

Svelte is a new way to build web applications. It's

Learn more at the [Svelte website](https://svelte.d

## Supporting Svelte

Svelte is an MIT-licensed open source project with
```

```
# Contributing to Svelte

Svelte is a new way to build web applications.
It's a compiler that takes your declarative
components and converts them into efficient
JavaScript that surgically updates the DOM.

The [Open Source Guides](https://opensource.guide/
) website has a collection of resources for
individuals, communities, and companies. These
resources help people who want to learn how to
run and contribute to open source projects.
Contributors and people new to open source alike
will find the following guides especially useful:

* [How to Contribute to Open Source](
https://opensource.guide/how-to-contribute/)
* [Building Welcoming Communities](
https://opensource.guide/building-community/)

## Get involved

There are many ways to contribute to Svelte, and
many of them do not involve writing any code.
Here's a few ideas to get started:

- Simply start using Svelte. Go through the [
```

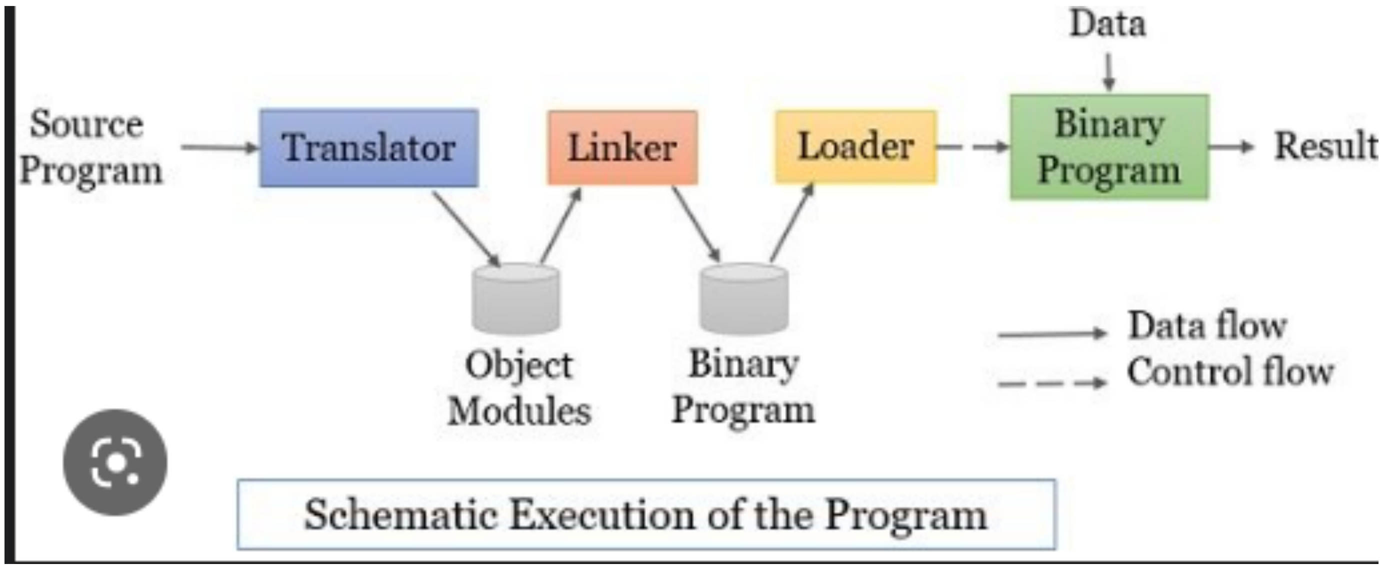
Step 2: The assembler reads the source file and produces an object file, optionally producing a listing file. If any errors occur, the programmer must return to Step 1 and fix the program.

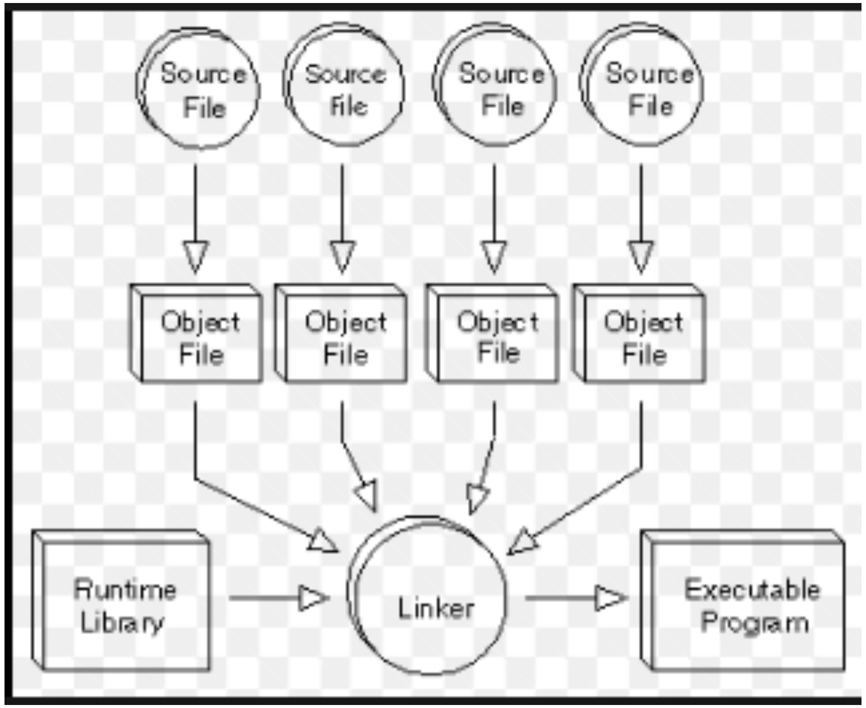


Can't read from the source file or disk.

Data Lineage
Type: File folder
Date modified: 12/30/2016 10:20 AM

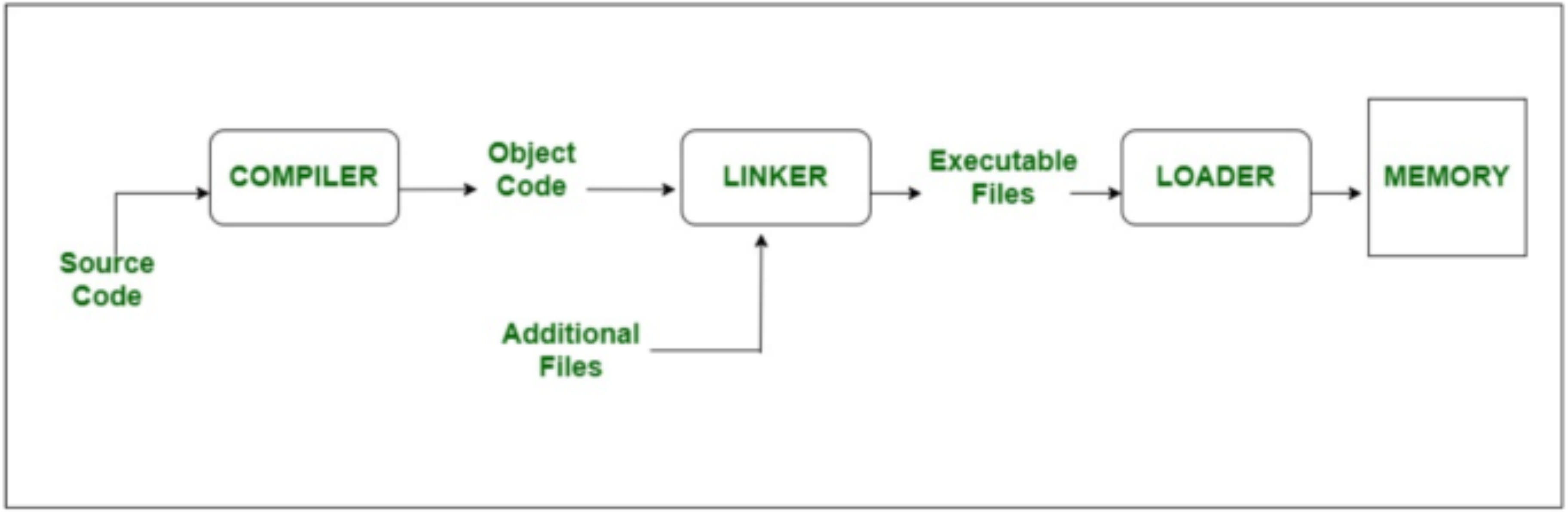
Step 3: The linker reads the object file, checks for calls to procedures in a link library, copies required procedures from the library, combines them with the object file, and produces the executable file.





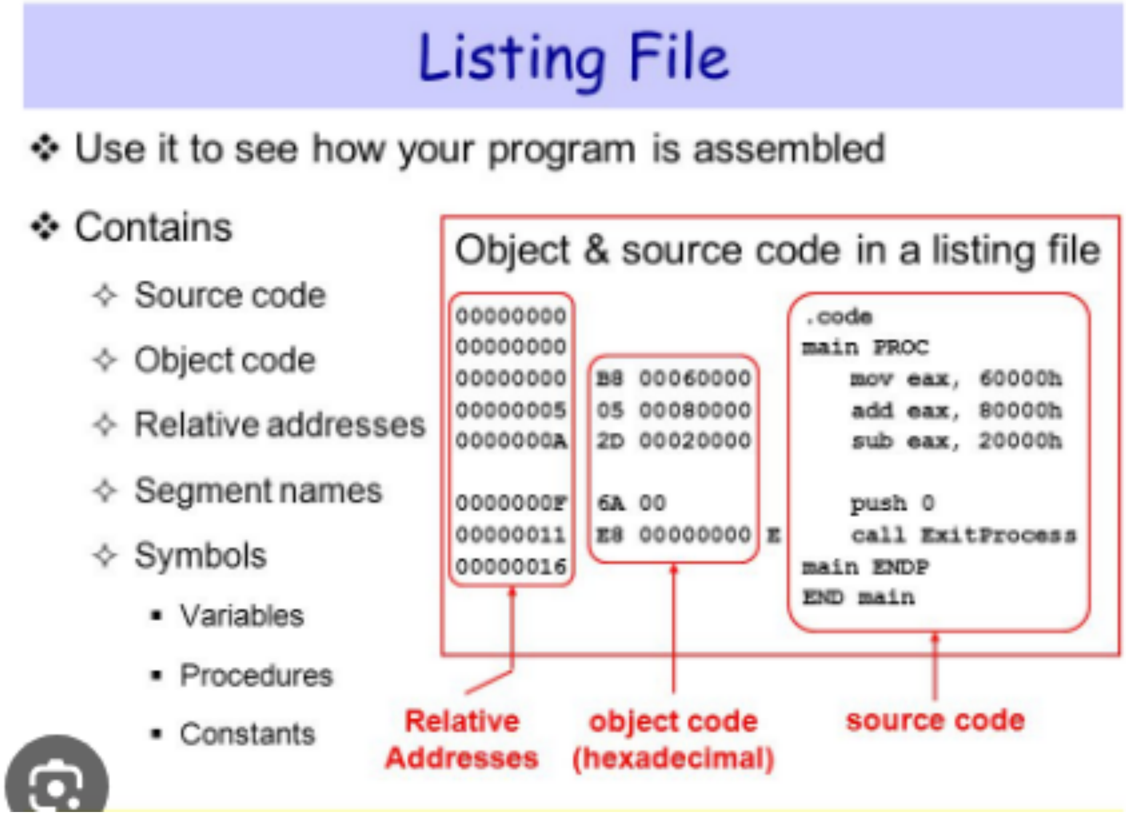
Step 4: The operating system loader utility reads the executable file into memory, branches the CPU

to the program's starting address, and the program begins to execute.



LISTING FILES AND SYMBOL TABLES

A **listing file** contains a copy of the program’s source code, with line numbers, the numeric address of each instruction, the machine code bytes of each instruction (in hexadecimal), and a symbol table.



Listing File

- Use it to see how your program is assembled
- Contains
  - Source code
  - Object code
  - Relative addresses
  - Segment names
  - Symbols
    - Variables
    - Procedures
    - Constants

Object & source code in a listing file

```
00000000                     .code
00000000                     main PROC
00000000   B8 00060000           mov eax, 60000h
00000005   05 00080000           add eax, 80000h
0000000A   2D 00020000           sub eax, 20000h

0000000F   6A 00                 push 0
00000011   E8 00000000 E         call ExitProcess
00000016                     main ENDP
                             END main
```

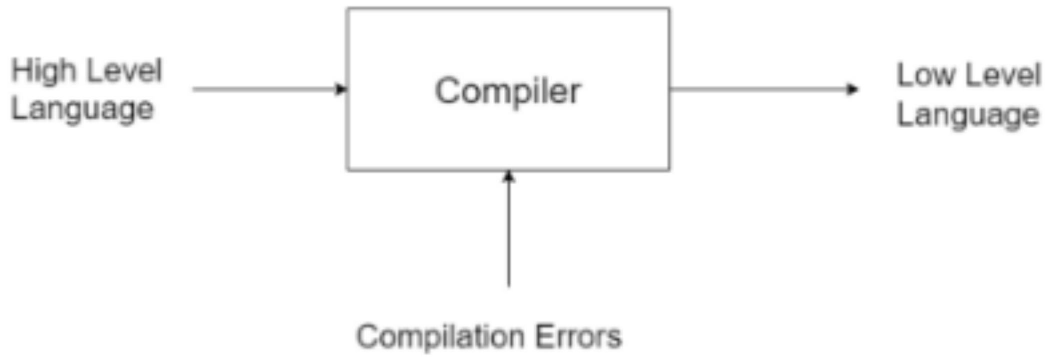
Relative Addresses

object code (hexadecimal)

source code

The **symbol table** contains the names of all program identifiers, segments, and related information. Advanced programmers sometimes use the listing file to get detailed information about the program.

Symbol Tables in Compiler Design



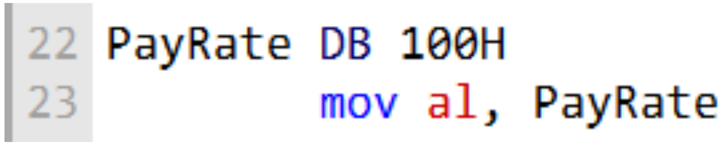
MOV AL,0001H means load the AL register with the contents of memory location 0001H. That is, if you stored the **pay rate in memory location 0001H**, then the next time you make use of the pay rate it should be fetched from memory location 0001H, but if you had used memory location 0002H instead of 0001H then that would be fine also – as long as you always used memory location 0002H when you wanted to use the pay rate.

For most tasks the exact memory location that you use to store some data is irrelevant - as long as it is always used consistently. In the early days programmers would have to start their programs by performing manual memory allocation. that is they first assigned uses to particular memory locations - 001 will be the total, 002 the running count, 003 a temporary result and so on.

Then as they programmed the action of the program they used the addresses that had been assigned in instructions. Clearly trying to remember where you stored everything isn’t fun and it’s very error prone.

The assembler idea, and its symbol table, can help again. Instead of using absolute memory addresses why not use symbolic addresses?

That is use symbols in your programs that the assembler replaces with consistent addresses that it assigns. For example, you might write something like:



```
PayRate DB 100H
        mov al, PayRate
```

Here the first line isn’t a program instruction – it is an assembler declaration. It says that the symbol “PayRate” is to be considered to be a particular byte in memory, i.e. DB = Declare Byte. When the programmer writes in the next line MOV AL,PayRate it means replace PayRate with the actual address that it was allocated.

Notice now that the assembler has taken on a new role in life – it is allocating memory! When the assembler translates the program to machine code it has to keep a table that tells it what address “PayRate” corresponds to and it has to allocate an address to “PayRate” when it meets its declaration for the first time.

In fact this idea is too good to leave there and not only can the “DB” assembler directive tell it to allocate memory it also can tell it what to initialise that memory to - 100H in this example. So when the MOV AL,PayRate instruction is obeyed not only does this load AL from an address allocated by the assembler, the contents of the memory location have already been initialised to 100H.

To implement this use of symbolic addressing the assembler has to have a slightly more sophisticated symbol table and back in the early days many a computer science course would spend hours on hash tables and other techniques for fast lookup of symbols. Today lookup tables are a solved problem – unless you’re discussing millions and millions of symbols!

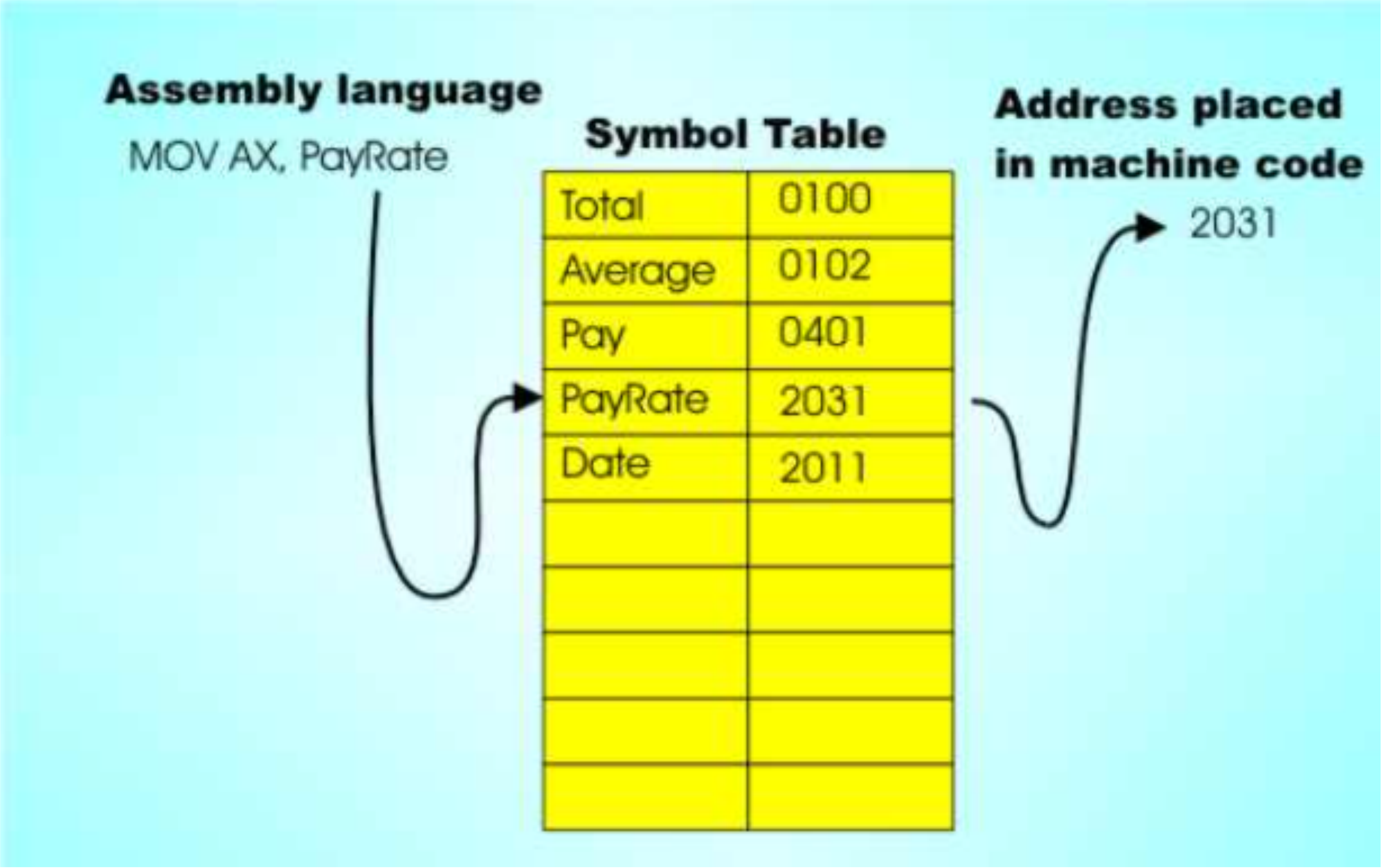
Yes, that is exactly how the assembler works. It keeps a symbol table of all the symbols in the program and their corresponding memory addresses. When the assembler encounters a symbol in an instruction, it looks up the symbol in the symbol table and replaces it with the corresponding memory address.

This allows programmers to use symbolic names instead of absolute memory addresses in their code. This makes the code more readable and easier to maintain.

In the example you provided, the assembler would create a symbol table entry for the symbol PayRate and assign it the memory address 100H. When the assembler encounters the instruction MOV AL,PayRate, it would look up the symbol PayRate in the symbol table and replace it with the memory address 100H. The assembler would then generate the machine code for the instruction MOV AL,100H.

The fact that the assembler can allocate memory and initialize it to specific values is also very useful. This allows programmers to declare variables and other data structures in their assembly language programs.

The use of symbolic addressing and the ability to allocate and initialize memory are two of the most important features of assemblers. These features make assembly language programming much more powerful and flexible.



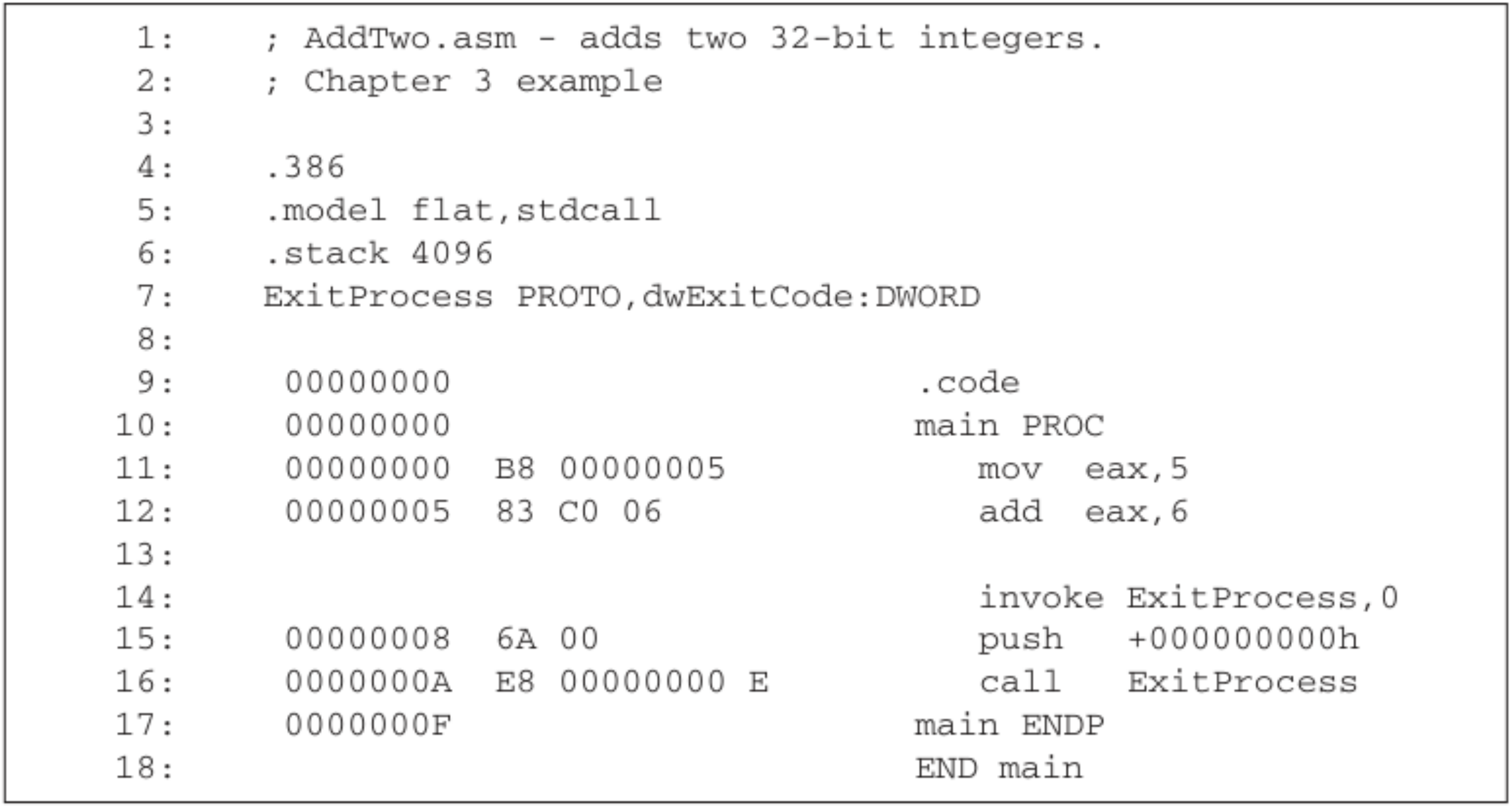
Total	0100
Average	0102
Pay	0401
PayRate	2031
Date	2011

Symbol tables store information about identifiers that occur in the (source) program. An identifier can be:

- **Program** (when having multiple input files, that we will not have for our compiler).
- **variables** (type, value, lineno's, scope, location etc.)
- **subprograms/subroutines** (we store the parameters, return type etc. to make it easier for the code generation)

- **subroutine parameters** (need to be separated from the variables, cause we need information for the way of passing etc.)
- **instruction labels** (when using goto or something else to go to a label, that we will not have)
- **constants** (sometimes storing the constants might also become handy, but we will store it as a AST Node)
- **data types** (like structs, unions etc. that we will add later on)

Figure below shows a partial listing file for the AddTwo program. Let’s examine it in more detail.



```
 1:    ; AddTwo.asm - adds two 32-bit integers.
 2:    ; Chapter 3 example
 3:
 4:    .386
 5:    .model flat,stdcall
 6:    .stack 4096
 7:    ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                        .code
10:     00000000                        main PROC
11:     00000000  B8 00000005              mov  eax,5
12:     00000005  83 C0 06                 add  eax,6
13:
14:                                        invoke ExitProcess,0
15:     00000008  6A 00                    push   +000000000h
16:     0000000A  E8 00000000 E            call   ExitProcess
17:     0000000F                        main ENDP
18:                                     END main
```

Lines 1–7 contain no executable code, so they are copied directly from the source file without changes.

Line 9 shows that the beginning of the code segment is located at address 00000000 (in a 32-bit program, addresses display as 8 hexadecimal digits). This address is relative to the beginning of the program's memory footprint, but it will be converted into an **absolute memory address** when the program is loaded into memory. When that happens, the program might start at an address such as 00040000h.

Lines 10 and 11 show the same starting address of 00000000, because the first executable statement is the MOV instruction on line 11. The bytes (B8 00000005) on line 11 represent the machine code instruction (B8) and the constant 32-bit value (00000005) that is assigned to EAX by the instruction. The value B8 is an opcode that represents the specific machine instruction to move a 32-bit integer into the EAX register.

Line 12 contains an executable instruction, starting at offset 00000005, which is a distance of 5 bytes from the beginning of the program.

Line 14 contains the invoke directive, which causes the assembler to generate the PUSH and CALL statements shown on lines 15 and 16.

The **sample listing file** in Figure above shows that the machine instructions are loaded into memory as a sequence of integer values expressed in hexadecimal. The number of digits in each number indicates the number of bits, and the machine instructions are exactly 15 bytes long.

The listing file is a useful resource to check if the assembler is generating the correct machine code bytes based on the program. Let's expand on the above listing file, with a simpler explanation:

.386 | This tells the assembler that the program is targeted for the 386 or higher processor

.model flat,stdcall | This tells the assembler that the program uses the flat memory model and the standard calling convention

.stack 4096 | This tells the assembler to reserve 4096 bytes of memory for the stack

ExitProcess PROTO,dwExitCode:DWORD | This declares a prototype for the ExitProcess function. The ExitProcess function terminates the program and returns the specified exit code.

.code | This tells the assembler that the following code is machine code.

main PROC | This is the beginning of the main() function.

mov eax,5 | This instruction moves the value 5 into the eax register.

add eax,6 | This instruction adds the value 6 to the eax register.

invoke ExitProcess,0 | This instruction calls the ExitProcess function with the exit code 0.

push +000000000h | This instruction pushes the value 0 onto the stack.

call ExitProcess | This instruction calls the ExitProcess function with the exit code on the stack.

main ENDP | This is the end of the main() function.

END main | This is the end of the program.

Let's break down the information you provided about assembly language and the given code excerpt:

Listing File and Executable Code: In assembly language programming, a listing file is a text document that displays the assembly code, machine code, and additional information about the program. In Figure 3-8, you have a partial listing file for the "AddTwo" program.

Memory Addresses:
Line 9 indicates that the beginning of the code segment starts at address 00000000. This address is relative to the program's memory footprint.
In a 32-bit program, addresses are typically displayed as 8 hexadecimal digits.
When the program is loaded into memory, this relative address may be converted into an absolute memory address (e.g., 00040000h).

Machine Code:
Line 11 contains machine code bytes (B8 00000005) followed by the assembly instruction mov eax,5.
The value B8 is an opcode that represents the specific machine instruction to move a 32-bit integer into the EAX register.

Offsets:
Line 12 contains an executable instruction starting at offset 00000005, which is a distance of 5 bytes from the beginning of the program.
Offsets are used to specify the location of instructions or data within the program's memory space.

INVOKE Directive:
Line 14 contains the invoke directive, which generates PUSH and CALL statements on lines 15 and 16.
The PUSH and CALL instructions are used for function calls and parameter passing.

Machine Code Values:
The listing file displays machine instructions as a sequence of integer values, expressed in hexadecimal.
For example, B8 represents an opcode, and 00000005 represents a 32-bit value assigned to EAX by the mov instruction.

Number of Digits and Bits:
The number of digits in each value indicates the number of bits. For instance, a 2-digit number is 8 bits, a 4-digit number is 16 bits, and an 8-digit number is 32 bits.

Purpose of Listing Files:
Listing files are helpful for verifying that the assembler generates the correct machine code based on your program.
They also serve as teaching tools for understanding how machine code instructions are generated.

Configuring Visual Studio for Listing Files:
Figure 3–9 provides guidance on configuring Visual Studio to generate a listing file, which can be useful for debugging and learning purposes.

The 80386, also known as the 386, is a 32-bit microprocessor that was introduced by Intel in 1985.
It was the successor to the 80286, and it was the first microprocessor to achieve widespread success in personal computers.
The 386 was a significant improvement over the 286, with a number of new features, including:

- 32-bit internal architecture
- Protected mode, which allowed multiple operating systems to run on the same computer
- Virtual memory, which allowed the computer to use more memory than was physically installed
- On-chip floating-point unit (FPU), which improved the performance of floating-point operations

The 386 was a major turning point in the history of personal computers, and it helped to usher in the era of 32-bit computing.
The term "386 or higher processor" means a processor that is at least as powerful as the 386. This

includes all modern processors, as they are all significantly more powerful than the 386.
If you see a software requirement that states that a 386 or higher processor is required, it means that the software will not run on a processor that is less powerful than the 386.
For example, if you are trying to install a piece of software on a computer that has a 286 processor, you will not be able to do so because the 286 is not as powerful as the 386.
If you are unsure whether your computer has a 386 or higher processor, you can check the specifications of your computer. The specifications will usually list the type of processor that is installed in your computer.
You can also use a program such as CPU-Z to check the specifications of your processor. CPU-Z is a free program that can be downloaded from the internet.

----

main ENDP
The main ENDP directive is used in assembly language to mark the end of the main() function. The main() function is the entry point for all assembly language programs. It is where the program execution begins.

The ENDP directive tells the assembler that the function is over. It also tells the assembler to generate the machine code for the function and to store it in the program's memory.
Without the main ENDP directive, the assembler will not be able to generate the correct machine code for the program.

END main
The END main directive is used in assembly language to mark the end of the program. It tells the assembler to stop generating machine code and to write the program to memory.
Without the END main directive, the assembler will not be able to finish generating the machine code for the program.

----

1. **main ENDP**: In assembly language, "main ENDP" is a directive used to mark the end of a procedure or subroutine named "main." It stands for "End Procedure." This directive tells the assembler that the code for the "main" procedure is complete, and the assembler should now handle any necessary cleanup tasks or bookkeeping related to the procedure. Essentially, it's a way to signify the end of the code block for the "main" procedure, allowing the assembler to move on to other parts of the program or handle subsequent procedures. It's an essential part of structuring assembly code and helps maintain program organization and readability.

2. **END main**: "END main" is another directive used in assembly language, but it serves a different purpose. This directive marks the end of the entire assembly program or source file and indicates that there is no more code to process. It is typically followed by the name of the entry point or the starting procedure, in this case, "main." This directive signifies the conclusion of the assembly program and is often used to generate an executable file. It tells the assembler to stop processing code and wrap up the compilation process, producing the final output file that can be run on a computer. In essence, "END main" is like the period at the end of a sentence, indicating the end of the program's source code.

Question: Is "END main" in assembly similar to how, in C programming, once the "main" function ends, everything is done?

Answer: Yes, that's correct! "END main" in assembly language and the termination of the "main" function in C both mark the end of program execution. Just as in C, where the "main" function serves as the entry point and the program terminates when it finishes, "END main" in assembly indicates the conclusion of code execution. This analogy illustrates that both in C and assembly, the respective markers signify the program's endpoint.

NB: To generate a listing file in Visual Studio, select Properties from the Project menu, then select Microsoft Macro Assembler, and set Generate Preprocessed Source Listing to Yes and List All Available Information to Yes in the Listing File dialog window.

QUESTIONS

What types of files are produced by the assembler?

The assembler produces object files, which contain machine code in a binary format that can be executed by the computer's processor. Additionally, the assembler may produce listing files, which display the original source code alongside the corresponding machine code instructions, and can be useful for debugging and understanding how the code is translated into machine language. Some assemblers may also produce debug files, which contain additional information about the code, such as symbol tables and line numbers, that can be used by a debugger to assist in program analysis and troubleshooting.

The linker extracts assembled procedures from the link library and inserts them in the executable program.

During the linking phase, the linker checks the object file for any unresolved external symbols, which are references to functions or variables defined in other object files or libraries. The linker then searches through the specified libraries and extracts the necessary procedures (functions) to resolve these external symbols. These extracted procedures are then inserted into the final executable program.

When a program’s source code is modified, it must be assembled and linked again before it can be executed with the changes.

executed with the changes.

Yes, that is true. When changes are made to a program's source code, the modified source code must be assembled into object code and linked with other necessary object files and libraries to create an executable program that incorporates those changes. Only then can the updated program be executed with the new changes.

Which operating system component reads and executes programs?

The operating system component responsible for reading and executing programs is the loader.

What types of files are produced by the linker?

The linker produces an executable file, also known as an executable program or application, which can be executed directly by the operating system. This file contains all the necessary machine code from the object files, as well as any required libraries or resources, and is ready to be loaded into memory and executed by the operating system's loader.