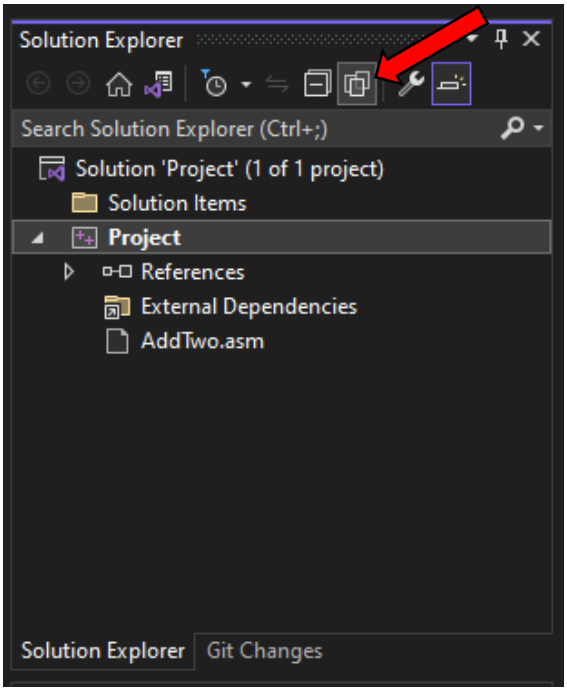
Lab 4 - Data Types and Data Transfers

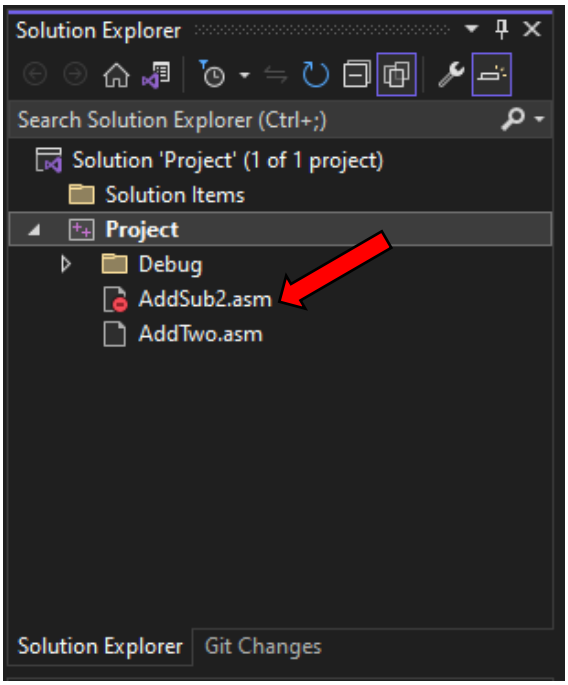
4.1 View the Registers in Console

This assembly program will display the general purpose and FLAGS registers in the console. We use ***DumpRegs*** procedure available in Irvine32.inc library to display the values of these registers in the console.

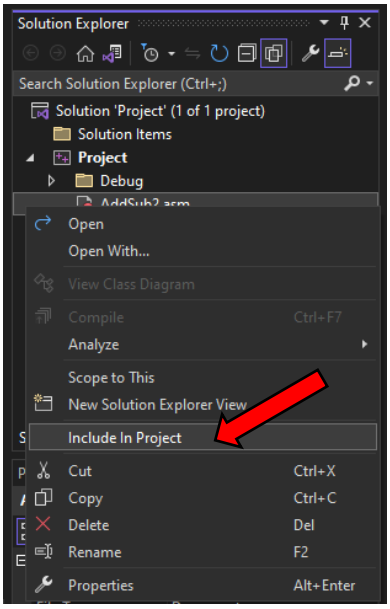
```
INCLUDE Irvine32.inc

.data
val1     dword  10000h
val2     dword  40000h
val3     dword  20000h
finalVal dword  ?

.code
main PROC

      mov    eax,val1          ; start with 10000h
      add    eax,val2          ; add 40000h
      sub    eax,val3          ; subtract 20000h
      mov    finalVal,eax      ; store the result (30000h)
      call   DumpRegs          ; display the registers

      exit
main ENDP
END main
```

This example program is also given in the Visual Studio Template Project we used in Lab 1. To include this file in the Visual Studio, go to “Solution Explorer” window then click on the “Show All Files” button to show all the available files in this project’s directory.



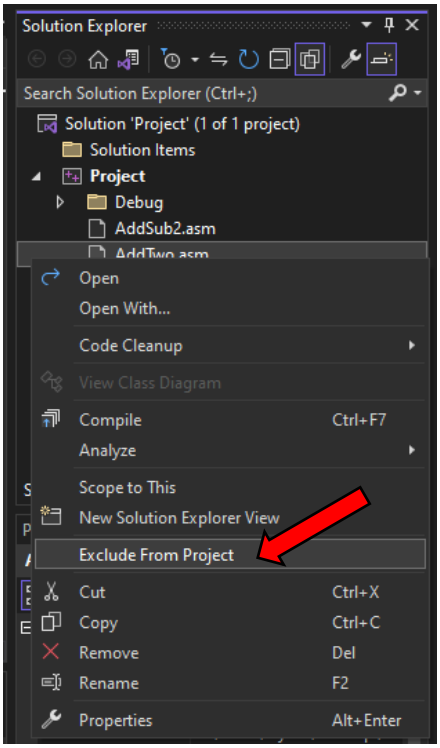


Now the Solution Explorer start showing all the files. Currently the “AddSub2.asm” is disabled it means it is not included in the project. To include this file in our project, right click on this file and clink on “Include In Project”.



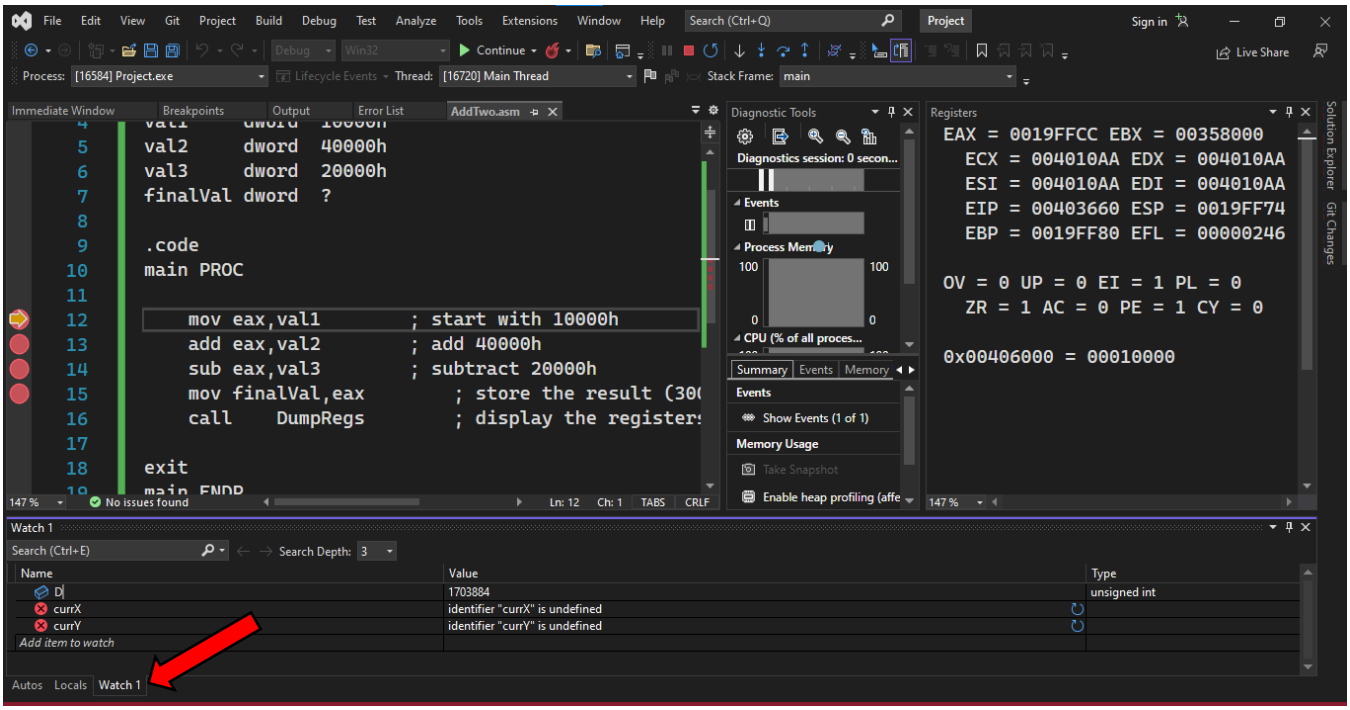
Now this “AddSub2.asm” can be executed. But if we try to execute this file Visual Studio will generate an error and this file will not be executed. The reason is that there are two .asm files, “AddSub2.asm” and “AddTwo.asm” and both files contain the “main” function. And we know that in a project there can’t be more than one “main” function. So, to execute this “AddSub2.asm” we have to exclude/disable the “AddTwo.asm” from the project.

To exclude the “AddTwo.asm” file from the project, right click on the file and then click on the “Exclude from Project”. The file will be excluded from the current project, and we can now successfully execute the “AddSub2.asm” file.

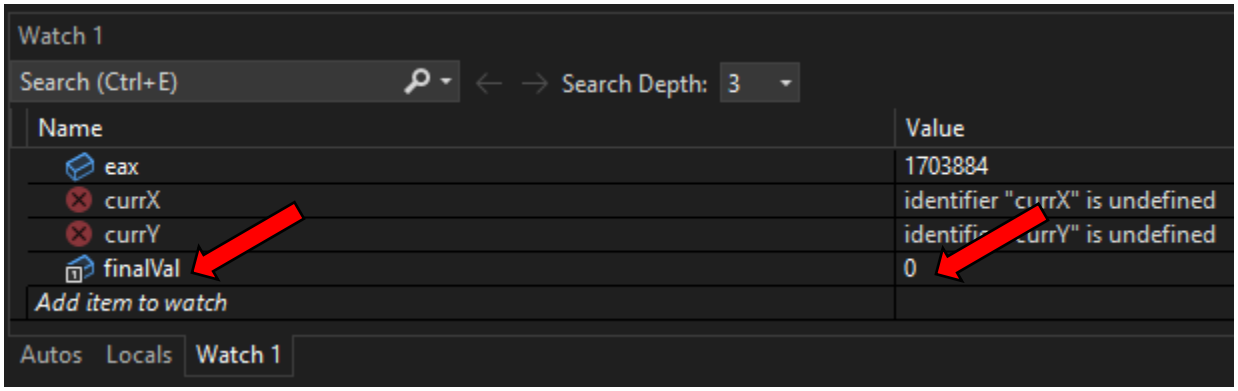


4.2 Watch a Variable in Debugging

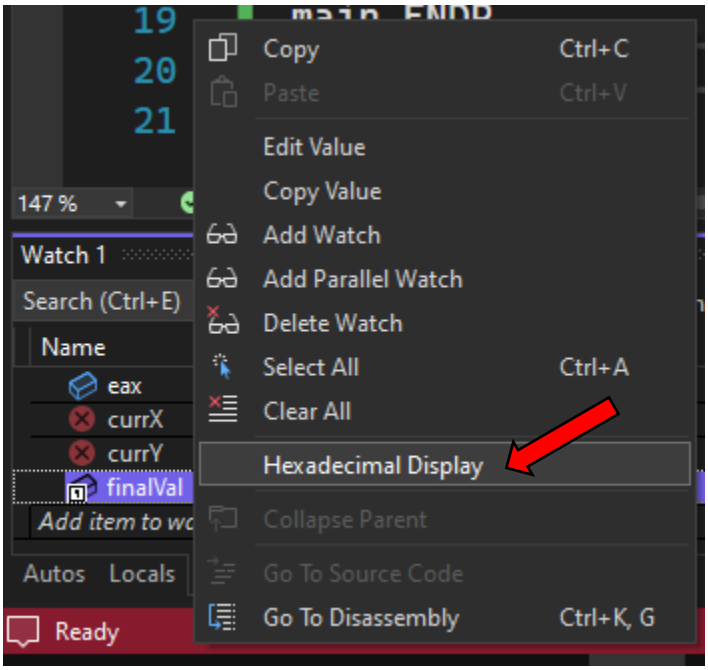
You can also view the value of variables during the debugging process like we watch the values of registers. In Visual Studio, to view the variable value during debugging you must add the variable in watch list and then start the Assembly program in debugging mode.

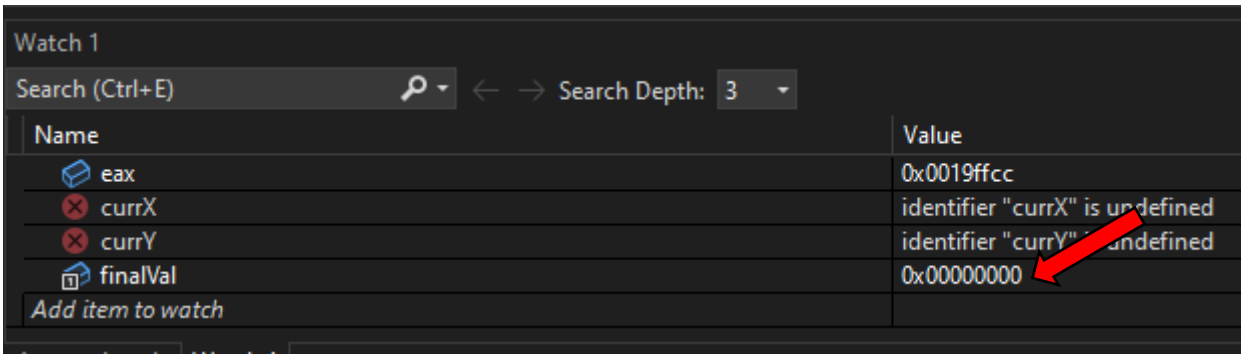


You will see a window named “Watch 1” showing some data. To view the value of a variable, we have to add it here in Watch 1 window. At the end of the list, click on “Add item to watch” and then type the name of that variable and press Enter. You can also add the variable by selecting the variable name in the code and drag & drop it to this list. The variable will be added in this list and its current value will be shown in “Value” column next to variable name.



Currently the value of this variable is showing in Decimal number system. To view the value in Hexadecimal, right click on the name of variable and click on “Hexadecimal Display”. The value will now start showing in Hexadecimal system. To view back in Decimal system, repeat the same process.





4.3 Operand Types

In Lab 3 we studied x86 instruction formats:

```
[label:] mnemonic [operands] [;comment ]
```

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

```
mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]
```

There are three basic types of operands:

- Immediate: uses a numeric literal expression (e.g., hard-coded constant value)
- Register: uses a named register in the CPU (EAX, AX, AL etc.)
- Memory: references a memory location (e.g., variable)

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP

reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8, 16, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8, 16, or 32-bit memory operand

4.4 Intrinsic Data Types

The assembler recognizes a basic set of *intrinsic data types*, which describe types in terms of their size (byte, word, doubleword, and so on), whether they are signed, and whether they are integers or reals. There's a fair amount of overlap in these types—for example, the DWORD type (32-bit, unsigned integer) is interchangeable with the SDWORD type (32-bit, signed integer).

The following table contains a list of all the intrinsic data types. The notation IEEE in some of the table entries refers to standard real number formats published by the IEEE Computer Society.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

4.4.1 Data Definition Statement

A *data definition statement* sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types. A data definition has the following syntax:

```
[name] directive initializer [,initializer]...
```

This is an example of a data definition statement:

```
count DWORD 12345
```

Directive: The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any of the types listed in above Table. In addition, it can be any of the legacy data definition directives shown in table below.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

Initializer: At least one initializer is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, initializer is an integer literal or integer expression matching the size of the variable's type, such as BYTE or WORD.

If you prefer to leave the variable uninitialized (assigned a random value), the "**?**" symbol can be used as the initializer. All initializers, regardless of their format, are converted to binary data by the assembler. Initializers such as 00110010b, 32h, and 50d all have the same binary value.

4.4.2 Defining BYTE and SBYTE Data

The BYTE (define byte) and SBYTE (define signed byte) directives allocate storage for one or more unsigned or signed values. Each initializer must fit into 8 bits of storage. For example:

```
value1 BYTE 'A'          ;character literal
value2 BYTE 0            ;smallest unsigned byte
value3 BYTE 255          ;largest unsigned byte
value4 SBYTE -128        ;smallest signed byte
value5 SBYTE +127        ;largest signed byte
```

A question mark (?) initializer leaves the variable uninitialized, implying that it will be assigned a value at runtime:

```
value6 BYTE ?
```

Name: The optional name is a label marking the variable's offset from the beginning of its enclosing segment. For example, if value1 is located at offset 0000 in the data segment and consumes one byte of storage, value2 is automatically located at offset 0001:

```
value1 BYTE 10h
value2 BYTE 20h
```

The DB directive can also define a signed or unsigned 8-bit variable:

```
val1 DB 255        ;unsigned byte
val2 DB -128       ;signed byte
```

4.4.3 Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. In the following example, assume list is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003. The following figure shows **list** as a sequence of bytes, each with its own offset.

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

Not all data definitions require labels. To continue the array of bytes begun with list, for example, we can define additional bytes on the next lines:

```
List   BYTE 10,20,30,40
       BYTE 50,60,70,80
       BYTE 81,82,83,84
```

Within a single data definition, its initializers can use different radixes. Character and string literals can be freely mixed. In the following example, **list1** and **list2** have the same contents:

```
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h
```

4.4.4 Defining Strings

To define a string of characters, enclose them in single or double quotation marks. The most common type of string ends with a null byte (containing 0). Called a null-terminated string, strings of this type are used in many programming languages:

```
greeting1 BYTE "Good afternoon",0
greeting2 BYTE 'Good night',0
```

Each character uses a byte of storage. Strings are an exception to the rule that byte values must be separated by commas. Without that exception, greeting1 would have to be defined as:

```
greeting1 BYTE 'G','o','o','d'
```

A string can be divided between multiple lines without having to supply a label for each line:

```
greeting1    BYTE "Welcome to the Encryption Demo program "
             BYTE "created by Kip Irvine.",0dh,0ah
             BYTE "If you wish to modify this program, please "
             BYTE "send me a copy.",0dh,0ah,0
```

The hexadecimal codes **0Dh** and **0Ah** are alternately called CR/LF (carriage-return/line-feed) or end-of-line characters. When written to standard output, they move the cursor to the left column of the line following the current line.

The line continuation character (\) concatenates two source code lines into a single statement. It must be the last character on the line.

The following statements are equivalent:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
```

Or:

```
greeting1 \
BYTE "Welcome to the Encryption Demo program "
```

4.4.5 DUP Operator

The DUP operator allocates storage for multiple data items, using an integer expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

```
BYTE 20 DUP(0)          ;20 bytes, all equal to zero
BYTE 20 DUP(?)          ;20 bytes, uninitialized
BYTE 20 DUP("STACK")    ;20 bytes: "STACKSTACKSTACKSTACK"
```

4.4.6 Defining WORD and SWORD Data

The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit (2 bytes) integers:

```
word1 WORD 65535        ;largest unsigned value
word2 SWORD -32768      ;smallest signed value
word3 WORD ?
```

The legacy DW directive can also be used:

```
val1 DW 65535       ;unsigned
val2 DW -32768      ;signed
```

The following instruction creates an array of 16-Bit Words by listing the elements or using the DUP operator. The following array contains a list of values:

```
myList WORD 1,2,3,4,5
```

If **myList** starts at offset 0000. The addresses increment by 2 because each value occupies 2 bytes.

Offset	Value
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

The DUP operator provides a convenient way to declare an array:

```
array WORD 5 DUP(?)     ;5 values, uninitialized
```

4.4.7 Defining DWORD and SDWORD Data

The DWORD directive (define doubleword) and SDWORD directive (define signed doubleword) allocate storage for one or more 32-bit integers:

```
val1 DWORD 12345678h            ;unsigned
val2 SDWORD -2147483648         ;signed
val3 DWORD 20 DUP(?)            ;unsigned array
```

The legacy DD directive can also be used to define doubleword data.

```
val1 DD 12345678h               ;unsigned
val2 DD -2147483648             ;signed
```

The DWORD can be used to declare a variable that contains the 32-bit offset of another variable. Below, **pVal** contains the offset of **val3**:

```
pVal DWORD val3
```

The following instruction creates an array of 32-Bit Doublewords by explicitly initializing each value:

```
myList DWORD 1,2,3,4,5
```

The following figure shows a diagram of this array in memory, assuming **myList** starts at offset 0000. The offsets increment by 4.

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

4.4.8 Defining QWORD Data

The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values:

```
quad1 QWORD 1234567812345678h
```

The legacy DQ directive can also be used to define quadword data:

```
quad1 DQ 1234567812345678h
```

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

4.4.9 Defining Packed BCD (TBYTE) Data

Intel stores a packed binary coded decimal (BCD) integers in a 10-byte package. Each byte (except the highest) contains two decimal digits. In the lower 9 storage bytes, each half-byte holds a single decimal digit. In the highest byte, the highest bit indicates the number's sign.

If the highest byte equals 80h, the number is negative; if the highest byte equals 00h, the number is positive. The integer range is -999,999,999,999,999,999 to +999,999,999,999,999,999.

For example, the hexadecimal storage bytes for positive and negative decimal 1234 are shown in the following table, from the least significant byte to the most significant byte:

Decimal Value	Storage Bytes
+1234	34 12 00 00 00 00 00 00 00 **00**
-1234	34 12 00 00 00 00 00 00 00 **80**

MASM uses the TBYTE directive to declare packed BCD variables. Constant initializers must be in hexadecimal because the assembler does not automatically translate decimal initializers to BCD. The following two examples demonstrate both valid and invalid ways of representing decimal -1234:

```
intVal TBYTE 80000000000001234h          ;valid
intVal TBYTE -1234                       ;invalid
```

The reason the second example is invalid is that MASM encodes the constant as a binary integer rather than a packed BCD integer.

4.4.10 Defining Floating-Point Types

REAL4 defines a 4-byte single-precision floating-point variable. REAL8 defines an 8-byte double-precision value, and REAL10 defines a 10-byte extended-precision value. Each requires one or more real constant initializers:

```
rVal1 REAL4 -1.2
rVal2 REAL8 3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

The following table describes each of the standard real types in terms of their minimum number of significant digits and approximate range:

Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} *to* 3.40×10^{38}
Long real	15	2.23×10^{-308} *to* 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} *to* 1.18×10^{4932}

The DD, DQ, and DT directives can define also real numbers:

```
rVal1 DD -1.2              ;short real
rVal2 DQ 3.2E-260          ;long real
rVal3 DT 4.6E+4096         ;extended-precision real
```

4.4.11 A Program That Adds Variables

The sample programs shown so far in the Lab 1 added integers stored in registers. Now we will revise the same program by making it add the contents of three integer variables and store the sum in a fourth variable.

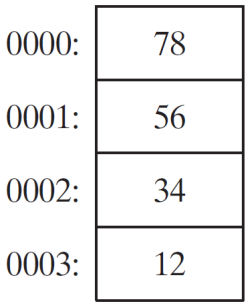
```
Include Irvine32.inc
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
	firstval DWORD 20002000h
	secondval DWORD 11111111h
	thirdval DD 22222222h
	sum DWORD 0
.code
main PROC
	mov eax,firstval
	add eax,secondval
	add eax,thirdval
	mov sum,eax
	call WriteInt
	call DumpRegs

	INVOKE ExitProcess,0
main ENDP
END main
```

Debug this code and watch the variables as well as the registers.

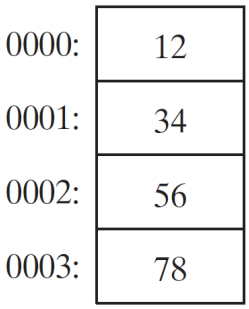
Note: The x86 instruction set does not let us add one variable directly to another, but it does allow a variable to be added to a register. That is why we used **EAX** as an accumulator.

4.5 Little-Endian Order

x86 processors store and retrieve data from memory using *little-endian* order (low to high). The least significant byte is stored at the first memory address allocated for the data. The remaining bytes are stored in the next consecutive memory positions. Consider the doubleword 12345678h. If placed in memory at offset 0000, 78h would be stored in the first byte, 56h would be stored in the second byte, and the remaining bytes would be at offsets 0002 and 0003, as shown in following figure:



Some other computer systems use *big-endian* order (high to low). Following figure shows an example of 12345678h stored in big-endian order at offset 0:



Note: Perform debugging on the code and then in "Disassembly" window view the instruction bytes. There you will see that the data (immediate constants) are showing in little-endian order.

4.6 Declaring Uninitialized Data

The .DATA? directive declares uninitialized data. When defining a large block of uninitialized data, the .DATA? directive reduces the size of a compiled program. For example, the following code is declared efficiently:

```
.data
      smallArray DWORD 10 DUP(0)            ;40 bytes
.data?
      bigArray DWORD 5000 DUP(?)            ;20,000 bytes, not initialized
```

The following code, on the other hand, produces a compiled program 20,000 bytes larger:

```
.data
      smallArray DWORD 10 DUP(0)            ;40 bytes
      bigArray DWORD 5000 DUP(?)            ;20,000 bytes
```

4.7 Mixing Code and Data

The assembler lets you switch back and forth between code and data in your programs. You might, for example, want to declare a variable used only within a localized area of a program. The following example inserts a variable named temp between two code statements:

```
.code
      mov eax,ebx
.data
```

```
        temp DWORD ?
.code
        mov temp,eax
. . .
```

Although the declaration of **temp** appears to interrupt the flow of executable instructions, MASM places temp in the data segment, separate from the segment holding compiled code. At the same time, intermixing *.code* and *.data* directives can cause a program to become **hard to read**.

----------Half Time----------

4.8 Symbolic Constants

A symbolic constant (or symbol definition) is created by associating an identifier (a symbol) with an integer expression or some text. Symbols do not reserve storage. They are used only by the assembler when scanning a program, and they cannot change at runtime. The following table summarizes their differences:

	Symbol	Variable
Uses storage?	No	Yes
Value changes at runtime?	No	Yes

These are some important symbolic constants:

- **Equal-Sign (=)** directive: to create symbols representing only integer expressions.
- **EQU** directives: to create symbols representing integer expressions, floating point expressions, or arbitrary text.
- **TEXTEQU** directives: to create symbols representing integer expressions, arbitrary text, or text macros.
- **Current Location Counter ($)**: to represent the address of the current location in code or data segment.

4.8.1 Equal-Sign Directive

The *equal-sign* directive associates a symbol name with an integer expression. The syntax is:

```
name = expression
```

The result of an expression is a 32-bit integer value. When a program is assembled, all occurrences of name are replaced by expression during the assembler's preprocessor step. Suppose the following statement occurs near the beginning of a source code file:

```
COUNT = 500
mov eax, COUNT
```

When the file is assembled, MASM will scan the source file and produce the corresponding code lines:

```
mov eax, 500
```

We might have skipped the COUNT symbol entirely and simply coded the MOV instruction with the literal 500, but experience has shown that programs are easier to read and maintain if symbols are used. Suppose COUNT were used many times throughout a program. At a later time, we could easily redefine its value.

A symbol defined with = can be redefined within the same program. The following example shows how the assembler evaluates COUNT as it changes value:

```
COUNT = 5
mov al,COUNT              ;AL = 5
COUNT = 10
mov al,COUNT              ;AL = 10
COUNT = 100
mov al,COUNT              ;AL = 100
```

4.8.2 EQU Directive

The EQU directive associates a symbolic name with an integer expression, floating point expression or some arbitrary text. There are three formats:

```
name EQU expression
name EQU symbol
name EQU <text>
```

- In the first format, expression must be a valid integer expression.
- In the second format, symbol is an existing symbol name, already defined with = or EQU.
- In the third format, any text may appear within the brackets <. . .>. When the assembler encounters **name** later in the program, it substitutes the integer value or text for the symbol.

EQU can be useful when defining a value that does not evaluate to an integer. A real number constant, for example, can be defined using EQU:

```
PI EQU <3.1416>
```

The following example associates a symbol with a character string. Then a variable can be created using the symbol:

```
pressKey EQU <"Press any key to continue...",0>
.
.
.data
prompt BYTE pressKey
```

Suppose we would like to define a symbol that counts the number of cells in a 10-by-10 integer matrix. We will define symbols two different ways, first as an integer expression and second as a text expression. The two symbols are then used in data definitions:

```
matrix1 EQU 10 * 10
matrix2 EQU <10 * 10>
.data
      M1 WORD matrix1
      M2 WORD matrix2
```

The assembler produces different data definitions for **M1** and **M2**. The integer expression in **matrix1** is evaluated and assigned to **M1**. On the other hand, the text in **matrix2** is copied directly into the data definition for **M2**:

```
M1 WORD 100
M2 WORD 10 * 10
```

Note: Unlike the = directive, a symbol defined with EQU cannot be redefined in the same source code file. This restriction prevents an existing symbol from being inadvertently assigned a new value.

4.8.3 TEXTEQU Directive

The TEXTEQU directive, like EQU, creates what is known as a text macro. There are three different formats: the first assigns text, the second assigns the contents of an existing text macro, and the third assigns a constant integer expression:

```
name TEXTEQU <text>
name TEXTEQU textmacro
name TEXTEQU %constExpr
```

For example, the **prompt1** variable uses the **continueMsg** text macro:

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
.data
prompt1 BYTE continueMsg
```

Text macros can build on each other. In the next example, **count** is set to the value of an integer expression involving **rowSize**. Then the symbol **move** is defined as **mov**. Finally, **setupAL** is built from **move** and **count**:

```
rowSize = 5
count TEXTEQU %(rowSize * 2)
move TEXTEQU <mov>
setupAL TEXTEQU <move EAX,count>
```

Therefore, the statement

```
setupAL
```

would be assembled as

```
mov EAX,10
```

A symbol defined by TEXTEQU can be redefined at any time.

4.8.4 Current Location Counter Symbol

One of the most important symbols of all, shown as **$,** is called the *current location counter*. For example, the following declaration declares a variable named **selfPtr** and initializes it with the variable's offset value:

```
selfPtr DWORD $
```

4.8.4.1 Calculating the Sizes of Arrays and Strings

When using an array, we usually like to know its size. The following example uses a constant named **ListSize** to declare the size of list:

```
list BYTE 10,20,30,40
ListSize = 4
```

Explicitly stating an array's size can lead to a programming error, particularly if you should later insert or remove array elements. A better way to declare an array size is to let the assembler calculate its value for you. The **$** operator (*current location counter*) returns the offset associated with the current program statement.

In the following example, **ListSize** is calculated by subtracting the offset of **list** from the **current location counter** ($):

```
INCLUDE Irvine32.inc
.data
	list BYTE 10,20,30,40
	ListSize = ($ - list)
.code
main proc
	mov EAX, ListSize
	call WriteInt
exit
main endp
end main
```

ListSize must follow immediately after **list**. The following, for example, produces too large a value (24) for **ListSize** because the storage used by **var2** affects the distance between the current location counter and the offset of **list**:

```
INCLUDE Irvine32.inc
.data
	list BYTE 10,20,30,40
	var2 BYTE 20 DUP(?)
	ListSize = ($ - list)
.code
main proc
	mov EAX, ListSize
	call WriteInt
exit
main endp
end main
```

String size can also be calculated using same method. The end-line and null-terminating characters will be counted too:

```
myString     BYTE "This is a long string, containing"
                  BYTE "any number of characters",0dh,0ah,0
myString_len = ($ - myString)
```

4.8.4.2 Arrays of Words and DoubleWords

When calculating the number of elements in an array containing values other than bytes, you should always divide the total array size (in bytes) by the size of the individual array elements. The following code, for example, divides the address range by 2 because each **word** in the array occupies 2 bytes (16 bits):

```
list WORD 1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

Similarly, each element of an array of **doublewords** is 4 bytes long, so its overall length must be divided by four to produce the number of array elements:

```
list DWORD 10000000h,20000000h,30000000h,40000000h
ListSize = ($ - list) / 4
```

Run the following Assembly code and explain its behavior also run it in debugging mode and watch the EAX:

```
INCLUDE Irvine32.inc
.data
	val SDWORD -150
	adrs DWORD ?
.code
main proc
mov adrs,$
	mov EAX, val
	call WriteInt
	JMP adrs
exit
main endp
end main
```

4.9 64-Bit Programming

To run 64-bit Assembly code, first you must download the 64-bit Visual Studio pre-configured template project from the following link:

- For Visual Studio 2022: Link
- For Visual Studio 2019: Link

Extract the project from the zip file then open the project solution file. There is a demo code to add two integers using 64-bit register RAX in the file "AddTwoSum_64.asm".

```
ExitProcess proto

.data
sum qword 0		;8 byte variable

.code
main proc
	mov RAX,5	; moving 5 to RAX register
	add RAX,6	; adding 6 to RAX
	mov sum,rax	; moving the result (11) to sum variable

	mov ecx,0	; setting ECX register to 0 to assign a process return code
	call ExitProcess
main endp
end
```

The following 3 lines are not used while working in 64-bit mode:

```
.386
.model flat,stdcall
.stack 4096
```

Statements using the PROTO keyword do not have parameters in 64-bit programs. This is from Line 1:

```
ExitProcess proto
```

This was our earlier 32-bit version:

```
ExitProcess PROTO,dwExitCode:DWORD
```

The 64-bit version of MASM does not support the INVOKE directive. So, we have to use CALL instruction instead:

```
call ExitProcess
```

The END directive does not specify a program entry point.

```
end
```

This was our earlier 32-bit version using END directive:

```
End main
```

4.10 Data Transfer

- Compilers of high-level languages perform strict type checking to avoid possible errors such as mismatching variables and data.
- But Assemblers let you do just about anything you want as long as the processor's instruction set can do what you ask.
- Assembly language forces you to pay attention to data storage and machine-specific details. You must understand the processor's limitations when you write assembly language code.
- x86 processors has a "complex instruction set" we discussed in Lab 3, so they offer a lot of ways of doing things.
- If you thoroughly learn the material presented in the Lab 4 and Lab 5, the rest of the Labs will go a lot more smoothly.
- In next Labs the example programs will become more complicated, so you will rely on mastery of fundamental tools presented in these Lab 4 and Lab 5.

4.10.1 Dereference Operation

For the following code:

```
.data
	var1 BYTE 10h
.code
	mov al,var1
```

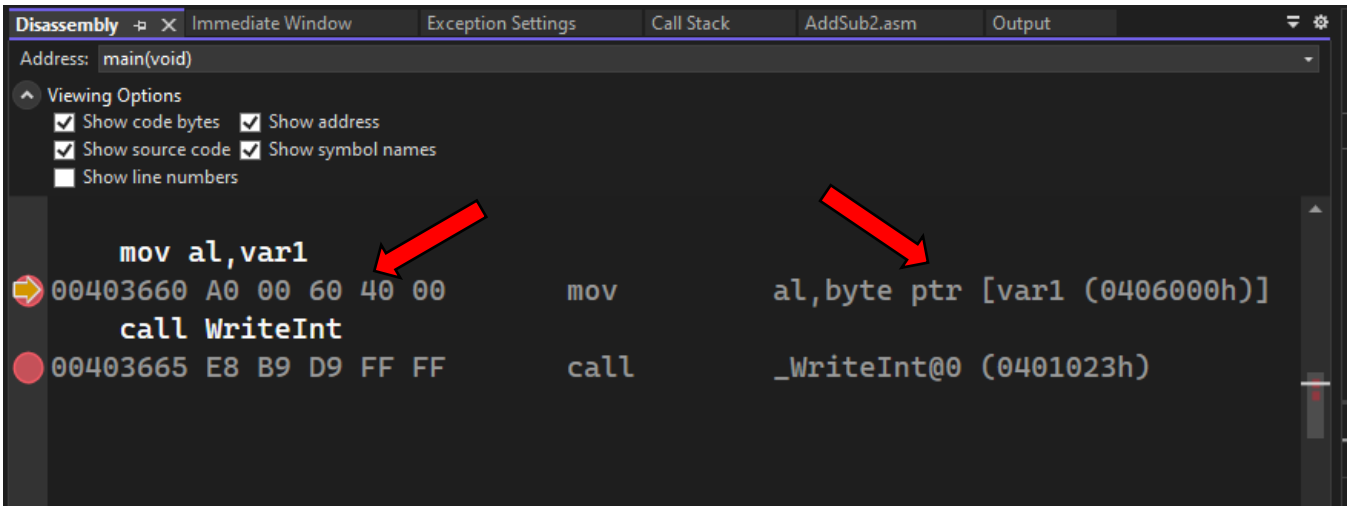
It would be assembled into the following machine instruction:

```
A0 00010400
```

The first byte in the machine instruction is the operation code (known as the *opcode*). The remaining part is the 32-bit hexadecimal address of **var1**. Although it might be possible to write programs using only numeric addresses, symbolic names such as **var1** make it easier to reference memory.

But what we see in disassembly or in listing file is different value of this operand. The reason is that MASM automatically dereference the operand and show the actual value of the operand in the byte code in the listing file.

So instead of the address of the variable **var1** (00010400) we see the data present at that address (00604000).



Assembly uses **[ ]** brackets as dereference operator(e.g., to get the data at location **var1**):

```
mov al,[var1]
```

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs are printed without the brackets.

Run the following Assembly code and explain its behavior also run it in debugging mode:

```
INCLUDE Irvine32.inc
.data
	var1 DB 16
	var2 DB	19
.code
main proc
	mov EAX,0

	mov EAX, OFFSET var1		; address of var1
	call WriteInt

	mov EAX, [OFFSET var1]		; value present at var1
	call WriteInt
	mov EAX, [OFFSET var1+1]	; value present at next byte to var1
	call WriteInt


exit
main endp
end main
```

4.10.2 MOV Instruction

The MOV instruction copies data from a source operand to a destination operand. Known as a *data transfer* instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination, and the second operand is the source:

```
MOV destination,source
```

The destination operand's contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

```
dest = source;
```

MOV follow these rules:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer registers (IP, EIP, or RIP) cannot be a destination operand.

These are MOV instruction formats:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

MOV instruction can not move data from memory to memory:

```
.data
	var1 WORD ?
	var2 WORD ?
.code
	mov var1,var2		;Error
	mov ax,var1		;Successful
	mov var2,ax
```

The MOV instruction can access individual parts of register like (EAX, AX, AL, AH) and hence we can have overlapping behavior:

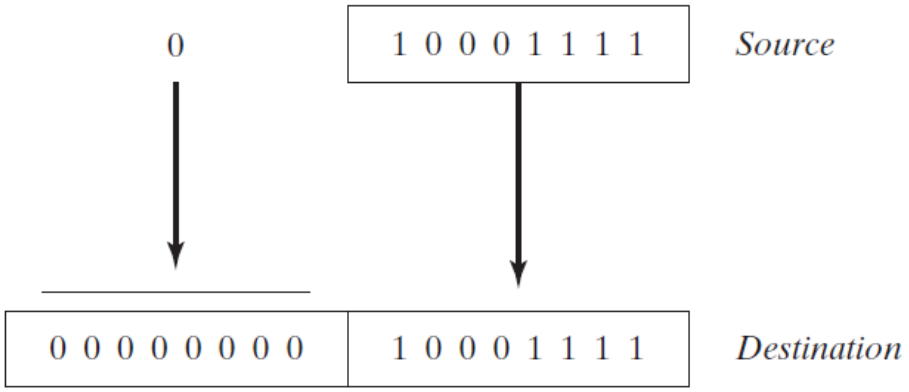
```
.data
	oneByte BYTE 78h
	oneWord WORD 1234h
	oneDword DWORD 12345678h
.code
	mov eax,0		; EAX = 00000000h
	mov al,oneByte		; EAX = 00000078h
	mov ax,oneWord		; EAX = 00001234h
	mov eax,oneDword	; EAX = 12345678h
	mov ax,0		; EAX = 12340000h
```

4.10.3 Copying Smaller Values to Larger Ones

4.10.3.3 MOVZX Instruction

The MOVZX instruction (move with zero-extend) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants:

```
MOVZX reg32,reg/mem8
MOVZX reg32,reg/mem16
MOVZX reg16,reg/mem8
```



Example 1:

```
.data
      byteVal BYTE 10001111b
.code
      movzx ax,byteVal    ; AX = 0000000010001111b
```

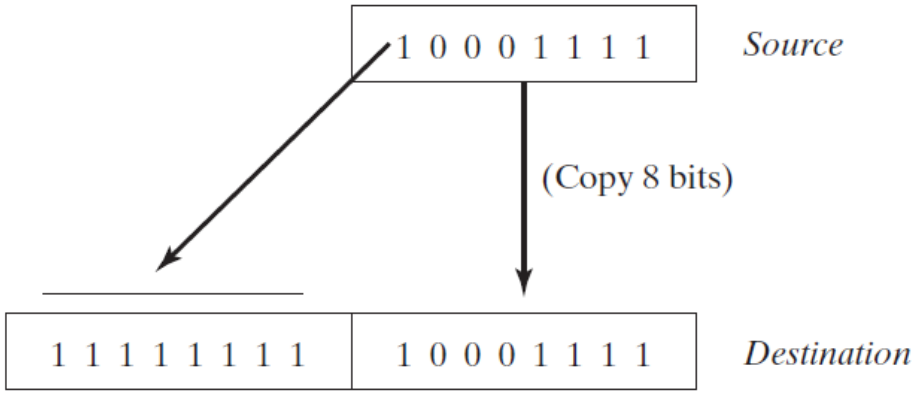
Example 2:

```
.data
byte1 BYTE 9Bh
word1 WORD 0A69Bh
.code
movzx eax,word1           ; EAX = 0000A69Bh
movzx edx,byte1           ; EDX = 0000009Bh
movzx cx,byte1            ; CX = 009Bh
```

4.10.3.4 MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

```
MOVSX reg32,reg/mem8
MOVSX reg32,reg/mem16
MOVSX reg16,reg/mem8
```



Example 1:

```
.data
      byteVal BYTE 10001111b
.code
      movsx ax,byteVal          ; AX = 1111111110001111b
```

Example 2:

```
mov bx,0A69Bh
movsx eax,bx        ; EAX = FFFFA69Bh
movsx edx,bl        ; EDX = FFFFFF9Bh
movsx cx,bl         ; CX = FF9Bh
```

4.11 LAHF and SAHF Instructions

The **LAHF** (load status flags into AH) instruction copies the low byte of the EFLAGS register into AH. Using this instruction, you can easily save a copy of the flags in a variable for safekeeping.

The following flags are copied:

- Sign
- Zero
- Auxiliary Carry
- Parity
- Carry

```
.data
   saveflags BYTE ?
.code
   lahf                     ; load flags into AH
   mov saveflags,ah         ; save them in a variable
```

The **SAHF** (store AH into status flags) instruction copies AH into the low byte of the EFLAGS (or RFLAGS) register. For example, you can retrieve the values of flags saved earlier in a variable:

```
mov ah,saveflags          ; load saved flags into AH
sahf                      ; copy into Flags register
```

4.12 XCHG Instruction

The **XCHG** (exchange data) instruction exchanges the contents of two operands. There are three variants:

```
XCHG reg,reg
XCHG reg,mem
XCHG mem,reg
```

XCHG do not allow memory to memory as well as immediate operands.

Example 1:

```
xchg ax,bx          ; exchange 16-bit regs
xchg ah,al          ; exchange 8-bit regs
xchg var1,bx        ; exchange 16-bit mem op with BX
xchg eax,ebx        ; exchange 32-bit regs
```

Example 2:

To exchange two memory operands, use a register as a temporary container and combine MOV with XCHG:

```
mov ax,val1
xchg ax,val2
mov val1,ax
```

4.13 Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels. Let's begin with an array of bytes named **arrayB**:

```
.data
      arrayB BYTE 10h,20h,30h,40h,50h
.code
      mov al,arrayB             ; First index, AL = 10h
      mov al,[arrayB+1]         ; Second index, AL = 20h
      mov al,[arrayB+2]         ; Third index, AL = 30h
```

An expression such as **arrayB+1** produces what is called an **"effective address"** by adding a constant to the variable's offset. Surrounding an effective address with brackets makes it clear that the expression is dereferenced to obtain the contents of memory at the address. The assembler does not require you to surround address expressions with brackets, but we highly recommend their use for clarity.

What will be result of the following instruction? Explain.

```
mov al,[arrayB+20]        ; AL = ??
```

For Word Arrays:

```
.data
   arrayW WORD 100h,200h,300h
.code
   mov ax,arrayW             ; AX = 100h
   mov ax,[arrayW+2]         ; AX = 200h
```

For Doubleword Arrays:

```
.data
      arrayD DWORD 10000h,20000h
.code
      mov eax,arrayD            ; EAX = 10000h
      mov eax,[arrayD+4]        ; EAX = 20000h
```

4.14 Example Program (Moves)

In this example we combined all the instructions we've covered so far, including MOV, XCHG, MOVSX, and MOVDX, to show how bytes, words, and doublewords are affected. We will also include some direct-offset operands.

Students should run this program and perform debugging and watch registers and variables at each step.

```
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD
.data
	val1 WORD 1000h
	val2 WORD 2000h
	arrayB BYTE 10h,20h,30h,40h,50h
	arrayW WORD 100h,200h,300h
	arrayD DWORD 10000h,20000h
.code
main PROC

; Demonstrating MOVZX instruction:
	mov bx,0A69Bh
	movzx eax,bx		; EAX = 0000A69Bh
	movzx edx,bl		; EDX = 0000009Bh
	movzx cx,bl		; CX = 009Bh

; Demonstrating MOVSX instruction:
	mov bx,0A69Bh
	movsx eax,bx		; EAX = FFFFA69Bh
	movsx edx,bl		; EDX = FFFFFF9Bh
	mov bl,7Bh
	movsx cx,bl		; CX = 007Bh

; Memory-to-memory exchange:
	mov ax,val1		; AX = 1000h
	xchg ax,val2		; AX=2000h, val2=1000h
	mov val1,ax		; val1 = 2000h

; Direct-Offset Addressing (byte array):
	mov al,arrayB		; AL = 10h
	mov al,[arrayB+1]	; AL = 20h
	mov al,[arrayB+2]	; AL = 30h

; Direct-Offset Addressing (word array):
	mov ax,arrayW		; AX = 100h
	mov ax,[arrayW+2]	; AX = 200h

; Direct-Offset Addressing (doubleword array):
	mov eax,arrayD		; EAX = 10000h
	mov eax,[arrayD+4]	; EAX = 20000h
	mov eax,[arrayD+4]	; EAX = 20000h

	INVOKE ExitProcess,0
main ENDP
END main
```