

# Lab 5 - Data Addressing and Arithmetic

## 5.1 Addition and Subtraction

Arithmetic is a big and most important topic in assembly language. In this Lab we will focus on addition and subtraction as well as the addressing and retrieval of the data stored in the memory. In this Lab we will also see how the CPU status flags (Carry, Sign, Zero, etc.) are affected by arithmetic instructions.

### 5.1.1 INC and DEC Instructions

The **INC** (increment) and **DEC** (decrement) instructions, respectively, add 1 and subtract 1 from a register or memory operand. The syntax is:

```
INC reg/mem
DEC reg/mem
```

For example:

```
Include Irvine32.inc
.data
    myWord WORD 1000h
.code
main proc
    ;increment data
    inc myWord                ;myWord = 1001h
    mov bx,myWord
    dec bx                    ; BX = 1000h

    ;increment address
    mov EAX, OFFSET myWord    ; get address of myWord
    call WriteInt
    inc EAX                    ; address of myWord + 1
    call WriteInt

    invoke ExitProcess,0
main endp
end main
```

**Note:** INC and DEC instructions do not affect the Carry flag.

### 5.1.2 ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. The syntax is:

```
ADD dest,source
```

Source remains unchanged by the ADD operation, and the sum is stored in the destination operand. The set of possible operands is the same as for the MOV instruction.

```
.data
    var1 DWORD 10000h
    var2 DWORD 20000h
.code
    mov eax,var1            ; EAX = 10000h
    add eax,var2            ; EAX = 30000h
```

### 5.1.3 SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The set of possible operands is the same as for the ADD and MOV instructions. The syntax is:

```
SUB dest,source
```

For example:

```
.data
    var1 DWORD 30000h
    var2 DWORD 10000h
.code
    mov eax,var1 ; EAX = 30000h
    sub eax,var2 ; EAX = 20000h
```

### 5.1.4 NEG Instruction

The NEG (negate) instruction reverses the sign of a number by converting the number to its two's complement. The syntax is:

```
NEG reg
NEG mem
```

For example:

```
.data
    var1 DWORD 30000h
.code
    NEG var1          ; var1 = FFFFD000h (negative of 30000h)
    mov eax,var1      ; EAX = FFFFD000h
    NEG EAX           ; EAX = 30000h
    Call WriteInt
```

### 5.1.5 Implementing Arithmetic Expressions

Using the ADD, SUB, and NEG instructions, we can implement arithmetic expressions involving addition, subtraction, and negation in assembly language.

Let's solve this equation in assembly language:  $sum = -x + (y - z)$ , where  $x = 15$ ,  $y = 20$  and  $z = 25$ .

```
Include Irvine32.inc
```

```
.data
    sum SDWORD ?
    x1 SDWORD 15
    y1 SDWORD 20
    z1 SDWORD 25
.code
main proc
    ; first term: -x
    mov eax,x1
    neg eax          ; EAX = -15

    ; second term: (y - z)
    mov ebx,y1
    sub ebx,z1       ; EBX = -5

    ; add the terms and store:
```

```

    add eax,ebx
    mov sum,eax      ; -20
    call WriteInt

    invoke ExitProcess,0
main endp
end main

```

**Student Task:** solve the following equation by utilizing the appropriate datatypes and constants:

$$((-var1 + 4) * var2) - var3 \quad \text{where } var1 = 17, var2 = 19 \text{ and } var3 = 21$$

## 5.2 Flags Affected by Addition and Subtraction

### 5.2.1 Zero Flag

The Zero flag is set when the result of an arithmetic operation equals zero.

```

mov ecx,1
sub ecx,1      ; ECX = 0, ZF = 1
mov eax,0FFFFFFh
inc eax        ; EAX = 0, ZF = 1
inc eax        ; EAX = 1, ZF = 0
dec eax        ; EAX = 0, ZF = 1

```

### 5.2.2 Carry Flag

When adding two unsigned integers, the Carry flag is a copy of the carry out of the most significant bit of the destination operand. Intuitively, we can say CF = 1 when the sum exceeds the storage size of its destination operand.

For addition:

```

mov al,0FFh
add al,1      ; AL = 00, CF = 1
mov ax,00FFh
add ax,1      ; AX = 0100h, CF = 0

```

For subtraction:

```

mov al,1
sub al,2      ; AL = FFh, CF = 1

```

### 5.2.3 Auxiliary Carry Flag

The Auxiliary Carry (AC) flag indicates a carry or borrow out of bit 3 in the destination operand.

```

      0 0 0 0 1 1 1 1
+     0 0 0 0 0 0 0 1
-----
      0 0 0 1 0 0 0 0

```

↓

```

mov al,0Fh
add al,1      ; AC = 1

```

### 5.2.4 Parity Flag

The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits.

```
mov al,10001100b
add al,00000010b      ; AL = 10001110, PF = 1
sub al,10000000b      ; AL = 00001110, PF = 0
```

### 5.2.5 Sign Flag

The Sign flag is set when the result of a signed arithmetic operation is negative. For example, subtracting integer 5 from 4:

```
mov eax,4
sub eax,5              ; EAX = -1, SF = 1
```

### 5.2.6 Overflow Flag

The Overflow flag is set when the result of a signed arithmetic operation overflows or underflows the destination operand.

```
mov al,+126
add al,1              ; OF = 0
add al,1              ; OF = 1
```

### 5.2.7 Example Program

The following example program implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how status flags are affected:

```
Include Irvine32.inc

.code
main proc
    ; Zero flag example:
    mov cx,1
    add cx,1      ; ZF = 0
    sub cx,2      ; ZF = 1
    mov ax,0FFFFh
    inc ax        ; ZF = 1

    ; Carry flag example:
    ;For addition:
    mov al,0FFh
    add al,1      ; AL = 00, CF = 1
    mov ax,00FFh
    add ax,1      ; AX = 0100h, CF = 0
    ;For subtraction:
    mov al,1
    sub al,2      ; AL = FFh, CF = 1

    ; Auxiliary Carry flag example:
    mov al,00001111b
    add al,00000001b      ; AC = 1
    mov al,00000111b
    add al,00000001b      ; AC = 0

    ; Parity flag example:
    mov al,10001100b
```

```

add al,00000010b      ; AL = 10001110, PF = 1
sub al,10000000b      ; AL = 00001110, PF = 0

; Sign flag example:
mov eax,4
sub eax,5              ; EAX = -1, SF = 1
mov eax,+5
add eax,+1             ; EAX = 6, SF = 0

; Overflow flag example:
mov al,+126
add al,1               ; OF = 0
add al,1               ; OF = 1

invoke ExitProcess,0
main endp
end main

```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of flags registers.

## 5.3 Data-Related Operators and Directives

Operators and directives are not executable instructions. They are interpreted by the assembler. You can use these directives to get information about the addresses and size characteristics of data.

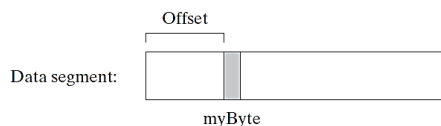
These are some important data operators:

[address kahan sai start ho raha hi](#)

- **OFFSET:** returns the distance of a variable from the beginning of its enclosing segment.
- **PTR:** lets you override an operand's default size.
- **TYPE:** returns the size (in bytes) of an operand or of each element in an array.
- **LENGTHOF:** returns the number of elements in an array.
- **SIZEOF:** returns the number of bytes used by an array initializer.
- **LABEL:** provides a way to redefine the same variable with different size attributes.
- **ALIGN:** aligns a variable on a byte, word, doubleword, or paragraph boundary.

### 5.3.1 OFFSET Operator

The OFFSET operator returns the offset of a data label. The offset represents the distance, in bytes, of the label from the beginning of the data segment.



```

.data
var1 BYTE 135
var2 WORD 3AF5h
var3 DWORD 12345678h
var4 BYTE 5

.code
mov ESI,OFFSET var1      ; ESI = 00404000h
mov ESI,OFFSET var2      ; ESI = 00404001h
mov ESI,OFFSET var3      ; ESI = 00404003h
mov ESI,OFFSET var4      ; ESI = 00404007h

```

Variables	Addresses	Memory
var1	00404000h	135
var2	00404001h	F5
	00404002h	3A
var3	00404003h	78
	00404004h	56
	00404005h	34
	00404006h	12
var4	00404007h	5

You can initialize a doubleword variable with the offset of another variable, effectively creating a pointer.

```
.data
    bigArray DWORD 500 DUP(?)
    pArray DWORD bigArray
.code
    mov esi, pArray          ; ESI = 00404000h
```

### 5.3.2 PTR Operator

PTR operator is used to override the declared size of an operand. This is only necessary when you're trying to access the operand using a size attribute that is different from the one assumed by the assembler.

For example, we move the lower 16-bits of a doubleword variable into AX.

```
.data
    var1 DWORD 12345678h
.code
    mov ax, var1              ; error
    mov ax, WORD PTR var1    ; successfully moves first 2 bytes to AX, AX = 5678h
```

Why wasn't 1234h moved into AX? x86 processors use the *little-endian* storage format in which the low-order byte is stored at the variable's starting address.

Variable	Addresses	Memory
var1	00404000h	78
	00404001h	56
	00404002h	34
	00404003h	12

We can access more data using the direct-offset method:

```
mov ax, WORD PTR [var1+2]    ; 1234h
mov al, BYTE PTR [var1+3]    ; 12h
mov ax, WORD PTR [var1+1]    ; 3456h
```

### 5.3.3 TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable. For example, the TYPE of a byte equals 1, the TYPE of a word equals 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8.

```
.data
    var1 BYTE ?
    var2 WORD ?
    var3 DWORD ?
    var4 QWORD ?

.code
    mov al, TYPE var1      ; AL = 1
    mov al, TYPE var2      ; AL = 2
    mov al, TYPE var3      ; AL = 4
    mov al, TYPE var4      ; AL = 8
```

### 5.3.4 LENGTHOF Operator

The LENGTHOF operator counts the number of elements in an array, defined by the values appearing on the same line as its label.

```
.data
    var1 BYTE 10,20,30
    var2 WORD 30 DUP(?),0,0
    var3 WORD 5 DUP(3 DUP(?))
    var4 DWORD 1,2,3,4
    var5 BYTE "12345678",0

.code
    mov al, LENGTHOF var1   ; AL = 3
    mov al, LENGTHOF var2   ; AL = 32
    mov al, LENGTHOF var3   ; AL = 15
    mov al, LENGTHOF var4   ; AL = 4
    mov al, LENGTHOF var5   ; AL = 9
```

If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array.

```
.data
    myArray BYTE 10,20,30,40,50
             BYTE 60,70,80,90,100

.code
    mov al, LENGTHOF myArray ; AL = 5
```

But if we end the first line with a comma and continue the list of initializers onto the next line then LENGTHOF would return the correct length:

```
.data
    myArray BYTE 10,20,30,40,50,
             60,70,80,90,100

.code
    mov al, LENGTHOF myArray ; AL = 10
```

### 5.3.5 SIZEOF Operator

The SIZEOF operator returns the total bytes consumed by the variable. It is a value that is equivalent to multiplying LENGTHOF by TYPE of the variable.

```
.data
var1 WORD ?
var2 DWORD 0000FFFFh
var3 QWORD ?
var4 BYTE 10,20,30
var5 WORD 10,20,30
intArray WORD 32 DUP(0)

.code
mov eax,SIZEOF var1      ; EAX = 2
mov eax,SIZEOF var2      ; EAX = 4
mov eax,SIZEOF var3      ; EAX = 8
mov eax,SIZEOF var4      ; EAX = 3
mov eax,SIZEOF var5      ; EAX = 6
mov eax,SIZEOF intArray  ; EAX = 64
```

### 5.3.6 LABEL Operator

The LABEL directive lets you insert a label and give it a size attribute without allocating any storage. That LABEL will point toward the memory location where the label is declared. The LABEL directive itself allocates no storage.

All standard size attributes can be used with LABEL, such as BYTE, WORD, DWORD, QWORD or TBYTE. A common use of LABEL is to provide an alternative name and size attribute for the variable declared next in the data segment.

```
.data
var1 LABEL WORD
var2 DWORD 12345678h
var3 LABEL BYTE
var4 WORD 0Fh

.code
mov eax, var2      ; EAX = 12345678h
mov ax, var1       ; EAX = 5678h
mov ax, [var1+2]   ; EAX = 1234h
mov al, var3       ; EAX = Fh
```

### 5.3.7 ALIGN Operator

The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary. The syntax is

**ALIGN** *bound*

*Bound* value can be 1, 2, 4, 8, or 16.

- **bound = 1:** aligns the next variable on a 1-byte boundary (the default).
- **bound = 2:** the next variable is aligned on an even-numbered address.
- **bound = 4:** the next address is a multiple of 4.
- **bound = 16:** the next address is a multiple of 16, a paragraph boundary.
- The assembler can insert one or more empty bytes before the variable to fix the alignment.
- Why aligning data? Because the CPU can process data stored at even numbered addresses more quickly than those at odd-numbered addresses.

```
.data
var1 BYTE 105      ; 00404000h
ALIGN 2
var2 WORD 4BF9h    ; 00404002h
```



```
var3 BYTE 9           ; 00404004h
ALIGN 4
var4 DWORD 12345678h   ; 00404008h
```

Variables	Addresses	Memory
<b>var1</b>	<b>00404000h</b>	<b>105</b>
-	00404001h	0
<b>var2</b>	<b>00404002h</b>	<b>F9</b>
	00404003h	<b>4B</b>
<b>var3</b>	<b>00404004h</b>	<b>9</b>
-	00404005h	0
-	00404006h	0
-	00404007h	0
<b>var4</b>	<b>00404008h</b>	<b>78</b>

## 5.4 Types of Operands while Accessing Data from Memory

### 5.4.1 Direct Operand

A direct operand is the name of a variable and represents the variable's address.

```
.data
var1 DWORD 12345678h
.code
mov eax, var1      ; var1 is Direct Operand
mov eax, [var1]    ; alternate method
```

### 5.4.2 Direct-Offset Operand

A direct-offset operand adds a displacement to the name of a variable, generating a new offset. This new offset can be used to access data in memory.

```
.data
var1 BYTE 5,6,7
var2 BYTE 9
.code
mov al, var1      ; var1 is Direct Operand (AL = 5)
mov al, [var1+1]  ; var1+1 is Direct-Offset Operand (AL = 6)
mov al, [var1+3]  ; (AL = 9)
```

### 5.4.3 Indirect Operand

An indirect operand is a register containing the address of data. By surrounding the register with brackets (as in [esi]), a program dereferences the address and retrieves the memory data.

```
.data
var1 BYTE 10h
.code
mov esi, OFFSET var1 ; ESI = 00404000h
mov al, [esi]        ; AL = 10h
```

If the destination operand uses indirect addressing, a new value is placed in memory at the location pointed to by the register. In the following example, the contents of the BL register are copied to the memory location addressed by ESI.

```
.data
    var1 BYTE 10h
.code
    mov esi, 00000000h    ; ESI = 00000000h
    mov al, 0FFh         ; AL = FFh
    mov [esi], al         ; FFh will store at var1 (var1 = FFh)
```

The size of an operand cannot be guessed by just the context of an instruction. For example, the following instruction causes the assembler to generate an “operand must have size” error message:

```
inc [esi]                ; error: operand must have size
```

The assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

#### 5.4.4 Indexed Operands

An indexed operand combines a constant with an indirect operand. The constant and register value are added, and the resulting offset is dereferenced. For example, [array+esi] and array[esi] are indexed operands.

Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable’s offset.

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

```
.data
    arrayB BYTE 10h,20h,30h
.code
    mov esi,0
    mov al, [arrayB+esi]    ; AL = 10h
    mov al, arrayB[esi]    ; alternate method
    inc esi
    mov al, [arrayB+esi]    ; AL = 20h
    inc esi
    mov al, [arrayB+esi]    ; AL = 30h
```

The second type of indexed addressing combines a register with a constant offset. The index register holds the base address of an array or structure, and the constant identifies offsets of various array elements.

```
.data
    arrayW WORD 1000h,2000h,3000h
.code
    mov esi,OFFSET arrayW    ; ESI = 00000000h
```

```

mov ax,[esi]      ; AX = 1000h
mov ax,[esi+2]    ; AX = 2000h
mov ax,[esi+4]    ; AX = 3000h

```

#### 5.4.4.1 Scale Factors in Indexed Operands

The *scale factor* is the size of the array component (word = 2, doubleword = 4, or quadword = 8).

To traverse an array, we can use the indexed operands method. But to get each item of the array we have to move the address according to the array datatype (BYTE, WORD or DWORD etc.). For example, to next 2nd value from WORD array we have to add 2 in the first address and so on. And to get the 2nd value of a DWORS array we have to add 4 in the first address.

```

.data
    arrayD DWORD 1,2,3,4
.code
    mov esi, 3                ; we want to access 4rd index of array
    mov eax, arrayD[esi*4]    ; EAX = 4

```

The TYPE operator can make the indexing more flexible should **arrayD** be redefined as another type in the future:

```

    mov esi, 3                ; we want to access 4rd index of array
    mov eax, arrayD[esi*TYPE arrayD] ; EAX = 4

```

## 5.5 Pointers

A variable containing the address of another variable is called a *pointer*. Pointers are a great tool for manipulating arrays and data structures because the address they hold can be modified at runtime. A pointer's size is affected by the processor's current mode (32-bit or 64-bit).

```

.data
    arrayB byte 10h,20h,30h,40h
    ptrB dword arrayB

```

Optionally, you can declare **ptrB** with the OFFSET operator to make the relationship clearer:

```
ptrB dword OFFSET arrayB
```

### 5.5.1 TYPEDEF Operator

The TYPEDEF operator lets you create a user-defined type that has all the status of a built-in type when defining variables. TYPEDEF is ideal for creating pointer variables.

For example, the following declaration creates a new data type PBYTE that is a pointer to bytes:

```
PBYTE TYPEDEF PTR BYTE
```

This declaration would be placed near the beginning of a program before the data segment. Then, variables could be defined using PBYTE:

```

PBYTE TYPEDEF PTR BYTE
.data
    arrayB BYTE 10h,20h,30h,40h
    ptr1 PBYTE ? ; uninitialized
    ptr2 PBYTE arrayB ; points to an array

```

### 5.5.2 Example Program: Pointers

The following program uses `TYPDEF` to create three pointer types (`PBYTE`, `PWORD`, `PDWORD`). It creates several pointers, assigns several array offsets, and dereferences the pointers:

```
Include Irvine32.inc

; Create user-defined types.
PBYTE TYPEDEF PTR BYTE ; pointer to bytes
PWORD TYPEDEF PTR WORD ; pointer to words
PDWORD TYPEDEF PTR DWORD ; pointer to doublewords

.data
    arrayB BYTE 10h,20h,30h
    arrayW WORD 1,2,3
    arrayD DWORD 4,5,6
    ; Create some pointer variables.
    ptr1 PBYTE arrayB
    ptr2 PWORD arrayW
    ptr3 PDWORD arrayD

.code
main PROC
    ; Use the pointers to access data.
    mov esi,ptr1
    mov al,[esi] ; 10h
    mov esi,ptr2
    mov ax,[esi] ; 1
    mov esi,ptr3
    mov eax,[esi] ; 4

    invoke ExitProcess,0
main ENDP
END main
```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.

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

## 5.6 JMP and LOOP Instructions

A transfer of control, or branch, is a way of altering the order in which statements are executed. There are two basic types of transfers:

- **Unconditional Transfer:** Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The **JMP** instruction does this.
- **Conditional Transfer:** The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

### 5.6.1 JMP Instruction

The **JMP** instruction causes an unconditional transfer to a destination, identified by a code label that is translated by the assembler into an offset. The syntax is:

**JMP** *destination*

The JMP instruction provides an easy way to create a loop by jumping to a label at the top of the loop:

```
label:
    .
    .
    jmp label        ; repeat the endless loop
```

JMP is unconditional, so a loop like this will continue endlessly unless another way is found to exit the loop.

### 5.6.2 Loop Instruction

The LOOP instruction, formally known as **Loop According to ECX Counter**, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decremented each time the loop repeats. Its syntax is:

```
LOOP destination
```

The loop destination must be within -128 to +127 bytes of the current location counter. The execution of the LOOP instruction involves two steps:

- First, it subtracts 1 from ECX.
- Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by destination. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruction following the loop.

```
.code
    mov ax,0
    mov ecx,5
L1:
    inc ax
    loop L1
```

A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.

```
.code
    mov ax,0
    mov ecx,0        ; logical error
L1:
    inc ax
    loop L1          ; it will decrement ECX until FFFFFFFFh
```

The ECX register is used by default by the LOOP instruction that is why we should not change the ECX register inside the loop body because it will alter the loop behavior.

```
L1:
    .
    .
    inc ecx          ;logical error
    loop L1
```

If you need to modify ECX inside a loop, you can save it in a variable at the beginning of the loop and restore it just before the LOOP instruction:

```

.data
    count DWORD ?
.code
    mov ecx,100          ; set loop count
top:
    mov count,ecx        ; save the count

    mov ecx,20           ; modify ECX
    ; use the ECX for your tasks

    mov ecx,count        ; and before LOOP instruction, restore loop count
    loop top

```

### 5.6.3 Nested Loops

When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. You can save it in a variable:

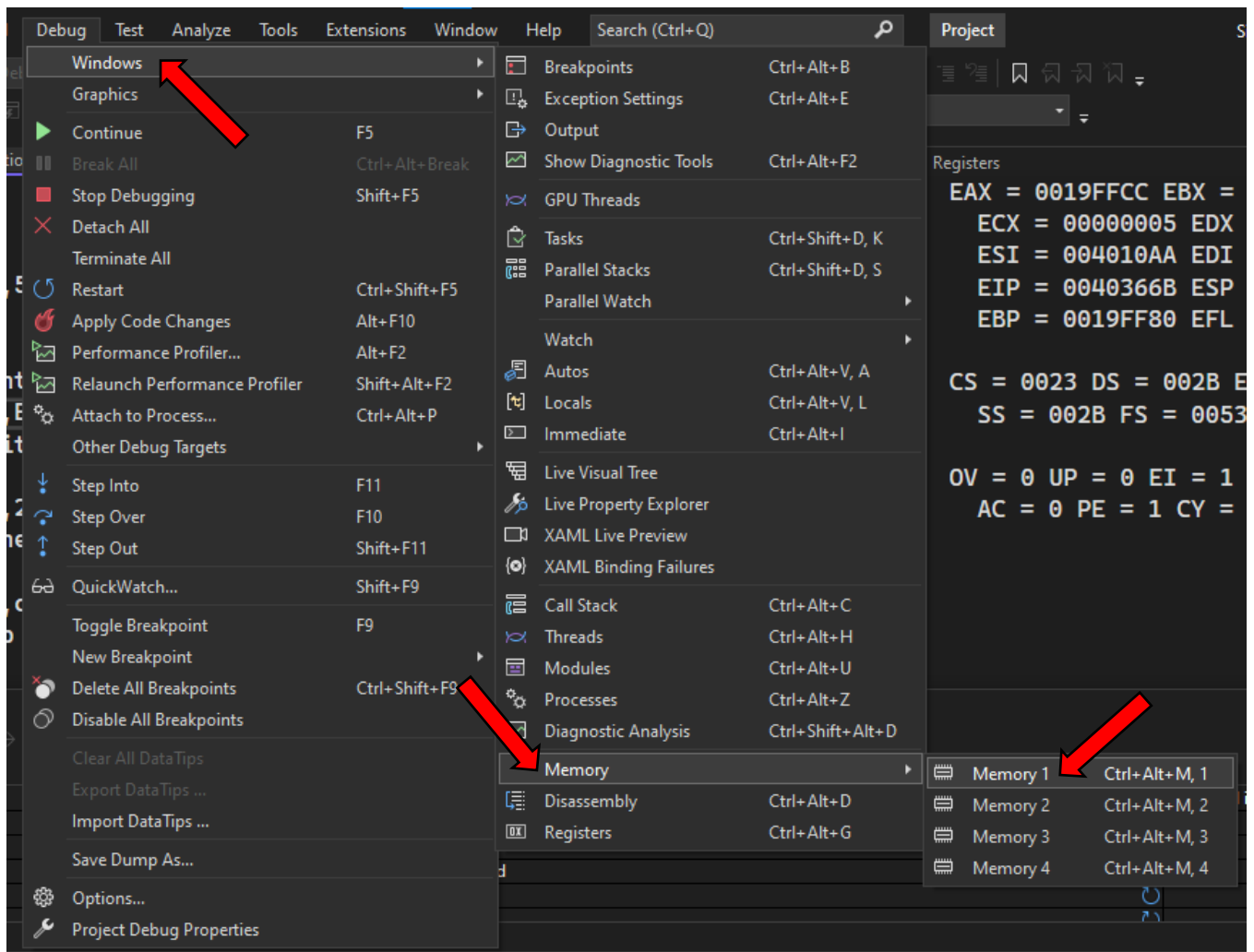
```

.data
    count DWORD ?
.code
    mov ecx,100          ; set outer loop count
L1:
    mov count,ecx        ; save outer loop count
    mov ecx,20           ; set inner loop count
L2:
    .
    .
    loop L2              ; repeat the inner loop
    mov ecx,count        ; restore outer loop count
    loop L1              ; repeat the outer loop

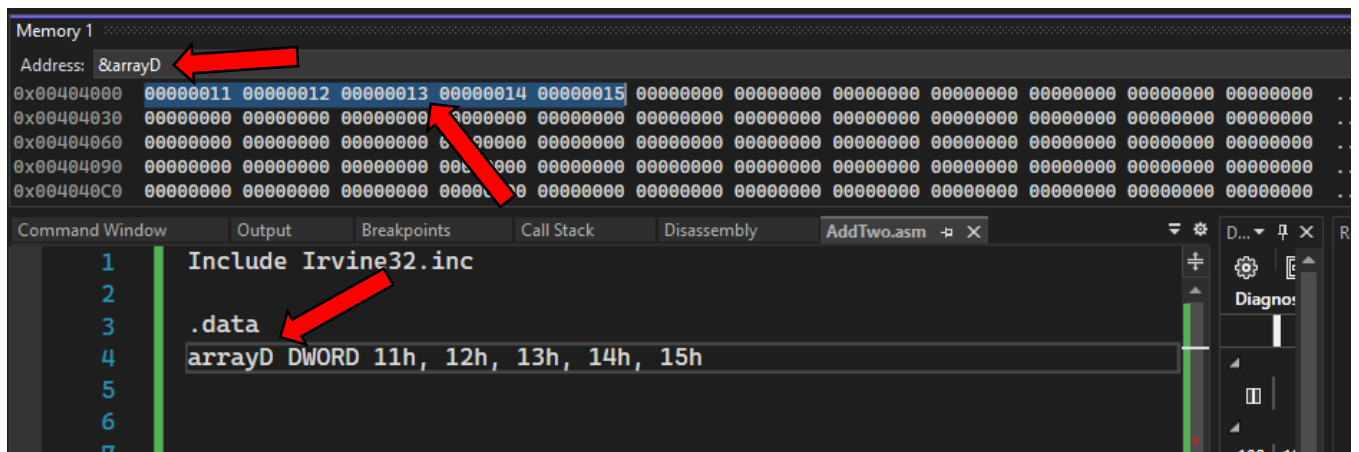
```

## 5.7 Displaying an Array in the Visual Studio Debugger

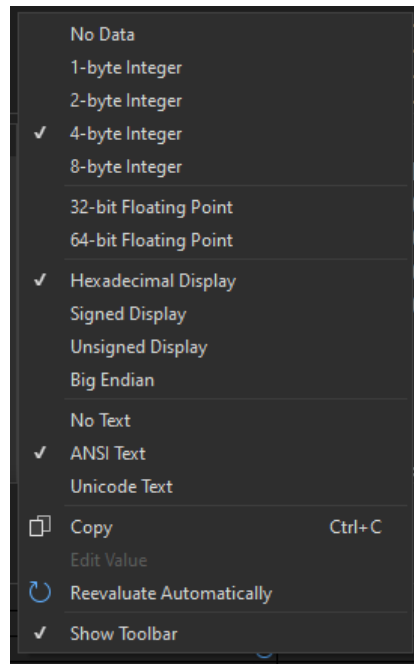
In a debugging session, if you want to display the contents of an array, go to “Debug” menu, select “Windows”, select “Memory”, then select “Memory 1”. A memory window will appear.



To view the array, in the Address field at the top of the memory window, type the “&” character, followed by the name of the array, and press Enter. For example, **&arrayD**. The memory window will display a block of memory starting at the array’s address.



If your array values are doublewords, you can right-click inside the memory window and select “4-byte integer” from the popup menu. You can also select from different formats, including Hexadecimal Display, signed decimal integer (called Signed Display), or unsigned decimal integer (called Unsigned Display) formats.



## 5.8 Summing an Integer Array

In assembly language, you would follow these 7 steps to add the elements of an array:

1. Assign the array's address to a register that will serve as an indexed operand.
2. Initialize the loop counter to the length of the array.
3. Assign zero to the register that accumulates the sum.
4. Create a label to mark the beginning of the loop.
5. In the loop body, add a single array element to the sum.
6. Point to the next array element.
7. Use a LOOP instruction to repeat the loop.

Include Irvine32.inc

```
.data
    intarray DWORD 10000h,20000h,30000h,40000h
.code
main PROC
    mov edi,OFFSET intarray      ; 1: EDI = address of intarray
    mov ecx,LENGTHOF intarray    ; 2: initialize loop counter
    mov eax,0                    ; 3: sum = 0
L1:                               ; 4: mark beginning of loop
    add eax,[edi]                ; 5: add an integer
    add edi,TYPE intarray        ; 6: point to next element
    loop L1                      ; 7: repeat until ECX = 0

    call WriteInt

    invoke ExitProcess,0
main ENDP
END main
```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.



## 5.9 Copying an Array

Include Irvine32.inc

```
.data
    source BYTE "This is the source string",0
    target BYTE SIZEOF source DUP(0)

.code
main PROC
    mov esi,0                ; index register
    mov ecx,SIZEOF source    ; loop counter
L1:
    mov al,source[esi]       ; get a character from source
    mov target[esi],al       ; store it in the target
    inc esi                  ; move to next character
    loop L1                  ; repeat for entire string

    mov edx,OFFSET target
    call WriteString         ; writes a null-terminated string to the console

    invoke ExitProcess,0
main ENDP
END main
```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.

## 5.10 64-Bit Programming

### 5.10.1 Summing an Array

This is the example program to sum the elements of array in 64-bit mode.

```
ExitProcess proto
.data
intarray QWORD 10000000000000h,20000000000000h
           QWORD 30000000000000h,40000000000000h

.code
main proc

    mov rdi,OFFSET intarray    ; 1: RDI = address of intarray
    mov rcx,LENGTHOF intarray ; 2: initialize loop counter
    mov rax,0                  ; 3: sum = 0
L1:                             ; 4: mark beginning of loop
    add rax,[rdi]              ; 5: add an integer
    add rdi,TYPE intarray      ; 6: point to next element
    loop L1                    ; 7: repeat until RCX = 0

    mov ecx,0                  ; ExitProcess return value
    call ExitProcess
main endp
end
```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.

## 5.10.2 Testing the 64-bit Move Operations

## 5.10.2.1 Example 1

ExitProcess proto

```

.data
    initval qword 0
    myByte byte 55h
    myWord word 6666h
    myDword dword 80000000h

.code
main proc
    ; Moving immediate values:
    mov rax,0FFFFFFFFh
    add rax,1                ; RAX = 100000000h

    mov rax,0FFFFh
    mov bx,1
    add ax,bx                ; RAX = 0

    mov rax,0
    mov ebx,1
    sub eax,ebx              ; RAX = 00000000FFFFFFFF

    mov rax,0
    mov bx,1
    sub ax,bx                ; RAX = 000000000000FFFF

    mov rax,0FFh
    mov bl,1
    add al,bl                ; RAX = 0

    mov rcx,OFFSET myByte
    inc BYTE PTR [rcx]        ; requires BYTE PTR
    dec BYTE PTR [rcx]

    mov ecx,0
    call ExitProcess
main endp
end

```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.

## 5.10.2.2 Example 2

ExitProcess proto

```

.data
    message BYTE "Welcome to my 64-bit Library!",0
    maxuint qword 0FFFFFFFFFFFFFFFFh
    myByte byte 55h
    myWord word 6666h
    myDword dword 80000000h

.code
main proc
    ; Moving immediate values:
    mov rax,maxuint                ; fill all bits in RAX
    mov rax,81111111h              ; clears bits 32-63 (no sign extension)
    mov rax,06666h                 ; clears bits 16-63
    mov rax,055h                   ; clears bits 8-63

    ; Moving memory operands:
    mov rax,maxuint                ; fill all bits in RAX
    mov eax,myDword                ; clears bits 32-63 (no sign extension)
    mov ax,myWord                  ; affects only bits 0-15
    mov al,myByte                  ; affects only bits 0-7

    ; 32-bit sign extension works like this
    mov myWord,8111h               ; make it negative
    movsx eax,myWord               ; EAX = FFFF8111h

    ; The MOVSXD instruction (move with sign-extension) permits the
    ; source operand to be a 32-bit register or memory operand:
    mov ebx,0FFFFFFFFh
    movsxd rax,ebx                 ; rax = FFFFFFFFFFFFFFFFFF

    mov ecx,0
    call ExitProcess
main endp
end

```

**Student Task:** Execute the above assembly program in debugging and watch the behavior of relevant registers.