Direct-Offset operands

Direct-offset operands are used to access memory locations with specific offsets from a variable's base address.

Direct-offset operands allow you to access memory locations in an array or structure by adding a displacement (offset) to the base variable's address.

```
arrayB BYTE 10h, 20h, 30h, 40h, 50h
mov al, arrayB                      ;al = 10h
mov al, [arrayB+1]                  ;al = 20h
mov al, [arrayB+2]                  ;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.

MASM has no built-in range checking for effective addresses.

In the following example, assuming arrayB holds five bytes, the instruction retrieves a byte of memory outside the array. The result is a sneaky logic bug, so be extra careful when checking array references:

```
mov al, [arrayB+20]           ;al = retrieving memory outside the array
```

Word and Doubleword Arrays

When accessing word and doubleword arrays, you need to take into account the size of the array elements. For example, the following code shows how to access the second element in an array of 16-bit words:

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

The [arrayW+2] operand tells the assembler to add 2 bytes to the offset of the arrayW variable and then access the memory location at that address. This is because each element in the arrayW array is 2 bytes long.

Similarly, the following code shows how to access the second element in an array of doublewords:

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

The [arrayD+4] operand tells the assembler to add 4 bytes to the offset of the arrayD variable and then access the memory location at that address. This is because each element in the arrayD array is 4 bytes long.

It is important to note that MASM does not perform range checking on direct-offset operands. This means that it is possible to **accidentally access memory locations outside of the program's address space.** It is important to be careful when using direct-offset operands and to make sure that you are always accessing valid memory locations.

```
.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
```

Repeat of the code:

```
.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
```

Of course, here's a detailed explanation of the key concepts and examples from the provided assembly code:

MOVZX Instruction (Zero Extension):

- The MOVZX instruction is used to zero-extend values from smaller registers or memory locations into larger ones.
- For example, it can extend a byte-sized value to a word or dword-sized value, filling the higher bits with zeros.

MOVSX Instruction (Sign Extension):

- The MOVSX instruction is used to sign-extend values from smaller registers or memory locations into larger ones.

registers or memory locations into larger ones.
• It extends a byte or word-sized value to a larger size while preserving the sign bit. If the original value is positive, it fills the higher bits with zeros; if negative, it fills them with ones.

Memory-to-Memory Exchange (XCHG):
• The XCHG instruction swaps the contents of two operands, whether they are registers or memory locations.
• In the provided code, it's used to swap the values of two variables, val1 and val2.

Direct-Offset Addressing:
• Direct-offset addressing allows you to access specific elements in arrays or data structures by adding an offset to the base address.
• It's particularly useful for accessing individual elements within arrays.
• In the code, it's demonstrated for byte, word, and doubleword arrays. For example, mov al, [arrayB+1] accesses the second byte in arrayB.

ExitProcess Function:
• The program concludes by invoking the ExitProcess function to exit the application. This function is part of the Windows API and is used to terminate the program gracefully.

The code serves as a practical demonstration of these assembly language concepts, illustrating how data transfers, memory operations, and type extensions work in x86 assembly. It's essential to understand these concepts when working with assembly language for tasks like data manipulation and memory management.