**Chapter 4 Notes**

**4.1 Data Transfer Instructions**

**4.1.1 Introduction**

* Assembly language is different from languages like Java or C++ because it doesn’t have strict syntax checking. This means programmers have more control but must be careful about machine-specific details, such as how data is stored.
* Understanding data transfer instructions is essential for writing effective assembly programs, especially on x86 processors which have many ways to perform tasks.

**4.1.2 Operand Types**

Operands are values used in assembly instructions, and they can be:

1. **Immediate**: A direct numeric value (like 5, 100).
2. **Register**: Refers to a CPU register (like AX, BX).
3. **Memory**: Refers to a location in memory where data is stored.

There are several operand types:

* **reg8**: 8-bit general-purpose registers (AH, AL, etc.)
* **reg16**: 16-bit general-purpose registers (AX, BX, etc.)
* **reg32**: 32-bit general-purpose registers (EAX, EBX, etc.)
* **imm**: Immediate value (8, 16, or 32-bit constants).
* **mem**: Refers to a memory operand (8, 16, or 32-bit).

**4.1.3 Direct Memory Operands**

* Variables in the data section refer to memory locations. For example, var1 BYTE 10h represents a byte variable var1 with the value 10h.
* You can access these values using memory operands in instructions, like mov al, var1 (copy var1 into register AL).
* **Alternative Notation**: Some prefer using brackets for memory operands like mov al, [var1].

**4.1.4 MOV Instruction**

* The **MOV** instruction copies data from a source to a destination.
  + **Format**: MOV destination, source
  + The destination is updated, but the source stays the same.
* The **MOV** instruction is versatile but follows these rules:
  + Both operands must be the same size.
  + You cannot move directly between two memory locations.
  + The instruction pointer (IP, EIP, RIP) cannot be the destination.
* **Memory to Memory Transfer**: You must first load data into a register before moving it to another memory location:

**Example:**

**mov ax, var1**

**mov var2, ax**

**4.1.5 Zero/Sign Extension of Integers**

* **Zero Extension**: When moving data from a smaller operand to a larger one (e.g., moving a 16-bit value into a 32-bit register), the value is padded with zeros.
  + **MOVZX**: A specific instruction for zero extension. It works with unsigned values.
    - Example: MOVZX ax, byteVal moves a byte into a 16-bit register, filling the higher bits with zeros.
* **Sign Extension**: When moving signed data, the highest bit of the source operand is copied to the upper bits of the destination operand, preserving the sign.
  + **MOVSX**: A specific instruction for sign extension. It works with signed values.
    - Example: MOVSX ax, byteVal moves a byte into a 16-bit register, filling the higher bits with the highest bit of the byte.

**4.1.6 LAHF and SAHF Instructions**

* **LAHF** (Load AH from Flags): Copies the lower 8 bits of the flags register (status flags) into the AH register.
* **SAHF** (Store AH to Flags): Copies the lower 8 bits of the AH register into the flags register.

**4.1.7 XCHG Instruction**

* The **XCHG** instruction is used to exchange (swap) data between two operands.
* It has three types:
  1. XCHG reg, reg – Swap two registers.
  2. XCHG reg, mem – Swap a register with memory.
  3. XCHG mem, reg – Swap memory with a register.
* **Important Note:** XCHG cannot work with immediate values (like constants).
* **Example:**
  1. xchg ax, bx swaps the values in AX and BX registers.
  2. xchg ax, al swaps the high byte (AH) and low byte (AL) in the AX register.
* **Memory Exchange**:
  1. To swap two memory locations, use a temporary register.

**Example:**

**mov ax, val1**

**xchg ax, val2**

**mov val1, ax**

**4.1.8 Direct-Offset Operands**

* You can access specific memory locations by adding a number (offset) to a variable's address.
* **Example**:
  + For an array of bytes: arrayB BYTE 10h, 20h, 30h, 40h, 50h
    - To get the first byte: mov al, arrayB (AL will be 10h).
    - To get the second byte: mov al, [arrayB + 1] (AL will be 20h).
* **Important**:
  + Use square brackets around the address to indicate memory access: [arrayB + 1].
  + Be careful with offsets! If you access an address beyond the array’s limit, it might cause unexpected results.
* **Word and Doubleword Arrays**:
  + **Word Arrays** (2 bytes per element): For example, in arrayW WORD 100h, 200h, 300h, each element is 2 bytes apart.
  + **Doubleword Arrays** (4 bytes per element): Example arrayD DWORD 10000h, 20000h has 4-byte elements.

**Example Program: Data Transfer**

* This example demonstrates using the MOV, MOVZX, MOVSX, XCHG, and direct-offset operands.

**.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**

**movzx edx, bl**

**; Demonstrating MOVSX instruction:**

**mov bx, 0A69Bh**

**movsx eax, bx**

**movsx edx, bl**

**; Memory-to-memory exchange:**

**mov ax, val1**

**xchg ax, val2**

**mov val1, ax**

**; Direct-Offset Addressing (byte array):**

**mov al, arrayB**

**mov al, [arrayB+1]**

**mov al, [arrayB+2]**

**; Direct-Offset Addressing (word array):**

**mov ax, arrayW**

**mov ax, [arrayW+2]**

**; Direct-Offset Addressing (doubleword array):**

**mov eax, arrayD**

**mov eax, [arrayD+4]**

**INVOKE ExitProcess, 0**

**main ENDP**

**END main**

* **What Happens**:
  + The MOVZX and MOVSX instructions are used to move data with zero-extend or sign-extend.
  + The XCHG instruction swaps values.
  + Direct-offset addressing retrieves data from specific memory positions.

**CPU Flags in Debugging**

* CPU flags help you monitor the processor's state while debugging. They include:
  1. **OV** (Overflow)
  2. **UP** (Direction flag)
  3. **EI** (Interrupt flag)
  4. **PL** (Sign flag)
  5. **ZR** (Zero flag)
  6. **AC** (Auxiliary Carry)
  7. **CY** (Carry flag)
* When debugging, you can view these flags in the debugger's **Registers** window. The flags change color when an instruction modifies their value.

**4.1.10 Section Review (Simplified Answers)**

1. **What are the three basic types of operands?**
   * The three basic types of operands are:
     1. **Register** – A small, fast storage in the CPU (like AX, BX).
     2. **Memory** – A place to store data in the computer's memory (like array, var1).
     3. **Immediate** – A constant value (like 5, 10h).
2. **(True/False): The destination operand of a MOV instruction cannot be a segment register.**
   * **False**. The destination operand in a MOV instruction can be a segment register (like CS, DS).
3. **(True/False): In a MOV instruction, the second operand is known as the destination operand.**
   * **True**. The second operand in a MOV instruction is the destination, where data is copied to.
4. **(True/False): The EIP register cannot be the destination operand of a MOV instruction.**
   * **True**. The **EIP** (Extended Instruction Pointer) register is used to point to the next instruction, so it cannot be the destination of a MOV instruction.
5. **In the operand notation used by Intel, what does reg/mem32 indicate?**
   * **reg/mem32** means the operand can be either a 32-bit register (like EAX, EBX) or a 32-bit memory address (like array[4]).
6. **In the operand notation used by Intel, what does imm16 indicate?**
   * **imm16** refers to a 16-bit immediate value, which is a constant number, like 10h or 5000.

**4.2 Addition and Subtraction in Assembly Language**

This chapter covers basic arithmetic operations like addition and subtraction in assembly language, including some details on how the CPU handles these operations using different instructions.

**4.2.1 INC and DEC Instructions**

* **INC**: Increments (adds 1) to a register or memory operand.
* **DEC**: Decrements (subtracts 1) from a register or memory operand.

**Syntax**:

* INC reg/mem
* DEC reg/mem

**Example**:

**.data**

**myWord WORD 1000h**

**.code**

**inc myWord ; myWord becomes 1001h**

**mov bx, myWord ; BX is now 1001h**

**dec bx ; BX becomes 1000h**

* **Flags**:
  + Changes **Overflow**, **Sign**, **Zero**, **Auxiliary Carry**, and **Parity** flags.
  + Does **NOT** affect the **Carry** flag.

**4.2.2 ADD Instruction**

The **ADD** instruction adds two operands of the same size. The result is stored in the destination operand, and the source operand remains unchanged.

**Syntax**:

* ADD dest, source

**Example**:

**.data**

**var1 DWORD 10000h**

**var2 DWORD 20000h**

**.code**

**mov eax, var1 ; EAX = 10000h**

**add eax, var2 ; EAX becomes 30000h (10000h + 20000h)**

* **Flags**: The flags (Carry, Zero, Sign, Overflow, etc.) are updated based on the result of the addition.

**4.2.3 SUB Instruction**

The **SUB** instruction subtracts the source operand from the destination operand.

**Syntax**:

* SUB dest, source

**Example**:

**.data**

**var1 DWORD 30000h**

**var2 DWORD 10000h**

**.code**

**mov eax, var1 ; EAX = 30000h**

**sub eax, var2 ; EAX becomes 20000h (30000h - 10000h)**

* **Flags**: The flags (Carry, Zero, Sign, Overflow, etc.) are updated based on the result of the subtraction.

**4.2.4 NEG Instruction**

The **NEG** instruction negates (reverses the sign of) a number by converting it to its two’s complement.

**Syntax**:

* NEG reg or NEG mem

**Explanation**:

* The two's complement is found by reversing all the bits and adding 1 to the result.
* **Flags**: The flags (Carry, Zero, Sign, Overflow, etc.) are updated based on the result.

**4.2.5 Implementing Arithmetic Expressions**

Using **ADD**, **SUB**, and **NEG**, you can implement complex arithmetic expressions in assembly.

**Example**: For the expression Rval = -Xval + (Yval - Zval):

1. **Negate Xval and store it in a register.**

**mov eax, Xval**

**neg eax ; EAX = -Xval**

1. **Subtract Zval from Yval.**

**mov ebx, Yval**

**sub ebx, Zval ; EBX = Yval - Zval**

1. **Add the two terms.**

**add eax, ebx ; EAX = -Xval + (Yval - Zval)**

**mov Rval, eax ; Store the result in Rval**

**4.2.6 Flags Affected by Addition and Subtraction**

When performing arithmetic, **status flags** help determine the outcome:

* **Carry Flag (CF)**: Indicates overflow in unsigned operations.
  + Set if the result exceeds the size of the destination operand.
* **Overflow Flag (OF)**: Indicates overflow in signed operations.
  + Set if the result is too large or small for the operand.
* **Zero Flag (ZF)**: Set if the result is zero.
* **Sign Flag (SF)**: Set if the result is negative (based on the most significant bit).
* **Parity Flag (PF)**: Set if there is an even number of 1s in the least significant byte of the result.
* **Auxiliary Carry Flag (AC)**: Set when there is a carry from bit 3 in the least significant byte.

**Examples of flags in action**:

* **Carry Flag**: Set when adding two numbers results in an overflow.

mov al, 0FFh

add al, 1 ; AL = 00, CF = 1

* **Zero Flag**: Set when the result is zero.

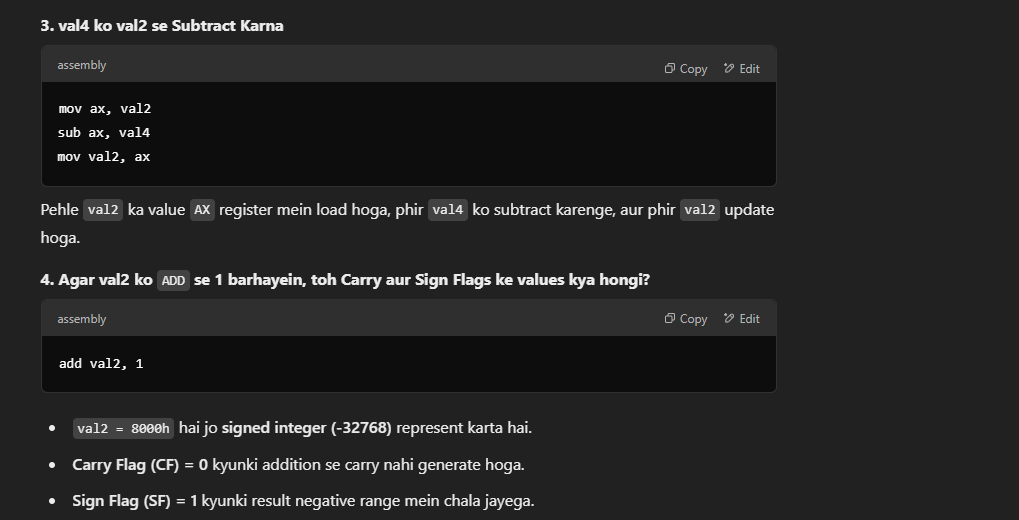
assembly

CopyEdit

mov ecx, 1

sub ecx, 1 ; ECX = 0, ZF = 1

**Summary:**

* **INC/DEC**: Used for adding/subtracting 1 from registers or memory.
* **ADD**: Adds two operands and stores the result in the destination.
* **SUB**: Subtracts the source operand from the destination operand.
* **NEG**: Negates the operand (reverses its sign).
* **Flags**: Help in detecting errors (carry, overflow, zero, etc.) and controlling program flow with conditional branching.
* A screenshot of a black and white screen

  AI-generated content may be incorrect.
* A screenshot of a computer

  AI-generated content may be incorrect.A screenshot of a computer program

  AI-generated content may be incorrect.

**4.3 Data-Related Operators and Directives**

Operators and directives are not instructions that the CPU executes. Instead, they are used by the assembler to gather information about data, its size, and address.

**Key Operators and Directives:**

1. **OFFSET Operator**:
   * It gives the offset (distance in bytes) of a variable from the start of its segment.
   * Example: OFFSET bVal returns the distance of bVal from the start of the data segment.
2. **PTR Operator**:
   * It overrides the default size of an operand. It allows you to specify a smaller or larger size for an operand.
   * Example: If myDouble is a DWORD, we use WORD PTR to access the lower 16 bits: mov ax, WORD PTR myDouble.
3. **TYPE Operator**:
   * It returns the size (in bytes) of an operand.
   * Example: TYPE var1 returns 1 (because a byte is 1 byte), TYPE var2 returns 2 (because a word is 2 bytes).
4. **LENGTHOF Operator**:
   * It counts the number of elements in an array.
   * Example: LENGTHOF array1 returns 30, which is the number of elements in array1.
5. **SIZEOF Operator**:
   * It calculates the size of an array in bytes, by multiplying the number of elements by the size of each element.
   * Example: SIZEOF intArray would return the total size of intArray.
6. **LABEL Directive**:
   * It defines an alias for a variable with a specific size, without allocating space for the variable.
   * Example: val16 LABEL WORD creates a label val16 with the size of a WORD.

**4.3.1 OFFSET Operator**

* The **OFFSET** operator gives the distance of a variable from the start of the segment in bytes.
* Example: If bVal is at address 00404000h, then OFFSET bVal will return 00404000h.

**4.3.2 ALIGN Directive**

* **ALIGN** is used to align data on a specific boundary, such as 1, 2, 4, 8, or 16 bytes. This ensures data is stored at the most efficient addresses for faster processing.
* Example: If you use ALIGN 2, the next variable will be aligned to an even-numbered address.

**4.3.3 PTR Operator**

* The **PTR** operator is used when you need to override the default size of an operand.
* Example: To move the lower 16 bits of a DWORD variable into a WORD register, you can use WORD PTR myDouble.

**4.3.4 TYPE Operator**

* The **TYPE** operator returns the size (in bytes) of a variable.
* Example: TYPE var1 returns 1 because var1 is a BYTE, TYPE var2 returns 2 because var2 is a WORD.

**4.3.5 LENGTHOF Operator**

* **LENGTHOF** calculates the number of elements in an array.
* Example: For an array defined as byte1 BYTE 10, 20, 30, LENGTHOF byte1 returns 3.

**4.3.6 SIZEOF Operator**

* **SIZEOF** calculates the total size (in bytes) of an array by multiplying the number of elements by the size of each element.
* Example: For intArray WORD 32 DUP(0), SIZEOF intArray would return 64 bytes.

**4.3.7 LABEL Directive**

* The **LABEL** directive is used to create an alias for a variable with a specific size.
* Example: val16 LABEL WORD creates a label val16 for a WORD variable, but does not allocate storage.

**Section Review Questions and Answers**

1. **Question**: Does the OFFSET operator always return a 16-bit value?  
   **Answer**: **False**. The OFFSET operator returns the **distance in bytes** from the start of the data segment, which could vary based on the system (16-bit, 32-bit, or more).
2. **Question**: Does the PTR operator return the 32-bit address of a variable?  
   **Answer**: **False**. The PTR operator does not return the **address** of a variable. It is used to **override the size** of an operand when accessing a variable in memory, like a byte, word, or doubleword.
3. **Question**: Does the TYPE operator return a value of 4 for doubleword operands?  
   **Answer**: **True**. The TYPE operator returns the size of a variable. For **doubleword operands**, it returns **4**, because a doubleword is 4 bytes.
4. **Question**: Does the LENGTHOF operator return the number of bytes in an operand?  
   **Answer**: **False**. The LENGTHOF operator returns the **number of elements** in an array, not the number of bytes.
5. **Question**: Does the SIZEOF operator return the number of bytes in an operand?  
   **Answer**: **True**. The SIZEOF operator returns the total number of bytes used by an operand, which is calculated by multiplying LENGTHOF by TYPE.

**4.4 Indirect Addressing**

Indirect addressing is used for array processing because **direct addressing** with constant offsets is impractical for larger arrays. Instead, we use a **register as a pointer** to point to memory locations. The value in the register is manipulated to access different data locations. When an operand uses indirect addressing, it's called an **indirect operand**.

**4.4.1 Indirect Operands**

* **Protected Mode**: Indirect operands can use **32-bit general-purpose registers** like **EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP**. These registers are surrounded by brackets, indicating that they hold the address of some data.
* **Example**: If the **ESI register** holds the address of byteVal, we can access the value at that address like this:

mov esi, OFFSET byteVal ; ESI points to byteVal

mov al, [esi] ; AL = byteVal (10h)

* If we need to modify the data at the address pointed to by the register, we can use an indirect operand in the destination:

mov [esi], bl ; Store the value of BL at the memory location pointed by ESI

* **PTR Operator**: Sometimes, the size of the operand may not be clear, causing errors. We use the **PTR operator** to explicitly specify the operand's size:

inc BYTE PTR [esi] ; Increment byte at the address pointed by ESI

**4.4.2 Arrays**

Indirect operands are perfect for processing **arrays**.

* **Example 1**: For an array of bytes (arrayB), as **ESI** is incremented, it points to each byte in sequence:

arrayB BYTE 10h, 20h, 30h

mov esi, OFFSET arrayB

mov al, [esi] ; AL = 10h

inc esi

mov al, [esi] ; AL = 20h

inc esi

mov al, [esi] ; AL = 30h

* **Example 2**: For an array of 16-bit integers (arrayW), we add 2 to **ESI** to move to the next 16-bit element:

arrayW WORD 1000h, 2000h, 3000h

mov esi, OFFSET arrayW

mov ax, [esi] ; AX = 1000h

add esi, 2

mov ax, [esi] ; AX = 2000h

add esi, 2

mov ax, [esi] ; AX = 3000h

* **Example 3**: For an array of **32-bit** integers (arrayD), we add 4 to **ESI** to move to the next 32-bit element:

arrayD DWORD 10000h, 20000h, 30000h

mov esi, OFFSET arrayD

mov eax, [esi] ; first number

add esi, 4

add eax, [esi] ; second number

add esi, 4

add eax, [esi] ; third number

**4.4.3 Indexed Operands**

Indexed addressing uses a **register** and **constant** to generate an effective address.

* There are two ways to write indexed operands:
  + constant[reg] — Combines the variable with a register.
  + [constant + reg] — Adds a constant offset to a register.
* **Example**: Accessing an array of bytes using **ESI** as an index:

mov esi, 0

mov al, arrayB[esi] ; AL = 10h

* **Example**: Accessing an array of 16-bit words:

mov esi, OFFSET arrayW

mov ax, [esi] ; First element (1000h)

mov ax, [esi+2] ; Second element (2000h)

mov ax, [esi+4] ; Third element (3000h)

* **Scaling**: When dealing with arrays, the index must account for the size of each element. For **32-bit** elements (doublewords), you multiply the subscript by 4:

mov esi, 3

mov eax, arrayD[esi\*4] ; EAX = 400h (4th element)

**4.4.4 Pointers**

A **pointer** is a variable that holds the **address of another variable**. Pointers are helpful for manipulating arrays and data structures.

* **Example**: Declaring a pointer to an array (arrayB):

ptrB DWORD arrayB ; ptrB contains the address of arrayB

* You can use the **OFFSET operator** to clarify the pointer’s purpose:

ptrB DWORD OFFSET arrayB

* **Pointer Types**: You can create new types using the TYPEDEF operator, which is especially useful for pointer variables:

PBYTE TYPEDEF PTR BYTE ; Defines a pointer to a byte

* **Example Program Using Pointers**:

TITLE Pointers

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

PBYTE TYPEDEF PTR BYTE

PWORD TYPEDEF PTR WORD

PDWORD TYPEDEF PTR DWORD

.data

arrayB BYTE 10h,20h,30h

arrayW WORD 1,2,3

arrayD DWORD 4,5,6

ptr1 PBYTE arrayB

ptr2 PWORD arrayW

ptr3 PDWORD arrayD

.code

main PROC

mov esi, ptr1

mov al, [esi] ; AL = 10h

mov esi, ptr2

mov ax, [esi] ; AX = 1

mov esi, ptr3

mov eax, [esi] ; EAX = 4

invoke ExitProcess, 0

main ENDP

END main

Here are the answers to the questions, explained in simple terms:

**1. (True/False): Any 32-bit general-purpose register can be used as an indirect operand.**

* **Answer**: **True**
  + In assembly, you can use any 32-bit general-purpose register (like EAX, EBX, ECX, etc.) as an indirect operand to refer to memory locations.

**2. (True/False): The EBX register is usually reserved for addressing the stack.**

* **Answer**: **False**
  + EBX is a general-purpose register and is not specifically reserved for addressing the stack. The ESP register is typically used for stack operations.

**3. (True/False): The following instruction is invalid: inc [esi]**

* **Answer**: **False**
  + The instruction inc [esi] is valid. It increments the value at the memory location pointed to by the register ESI.

**4. (True/False): The following is an indexed operand: array[esi]**

* **Answer**: **True**
  + In this expression, esi is used as an index to access the value in the array. This is called an indexed operand.

**5. Register values after each instruction sequence:**

We have the following data:

myBytes BYTE 10h, 20h, 30h, 40h

myWords WORD 8Ah, 3Bh, 72h, 44h, 66h

myDoubles DWORD 1, 2, 3, 4, 5

myPointer DWORD myDoubles

**Instruction sequence**:

1. mov esi, OFFSET myBytes
   * ESI points to the beginning of myBytes, which is the address of the first byte.
2. mov al, [esi]
   * AL gets the value at the address esi points to. myBytes[0] = 10h.
   * **AL = 10h**
3. mov al, [esi+3]
   * AL gets the value at the address esi + 3. myBytes[3] = 40h.
   * **AL = 40h**
4. mov esi, OFFSET myWords + 2
   * ESI points to the address of myWords + 2. It now points to the third word (72h).
5. mov ax, [esi]
   * AX gets the value at the address esi points to. myWords[1] = 3Bh (because esi now points to the second word).
   * **AX = 3Bh**
6. mov edi, 8
   * EDI gets the value 8.
7. mov edx, [myDoubles + edi]
   * EDX gets the value at the address myDoubles + 8. Since myDoubles[2] = 3, it loads 3 into EDX.
   * **EDX = 3**
8. mov edx, myDoubles[edi]
   * EDX is again set to myDoubles[8], which would be 3.
   * **EDX = 3**
9. mov ebx, myPointer
   * EBX gets the address of myDoubles.
10. mov eax, [ebx+4] - EAX gets the value at the address myDoubles + 4. myDoubles[1] = 2. - **EAX = 2**

So, the answers are:

* a. **AL = 10h**
* b. **AL = 40h**
* c. **AX = 3Bh**
* d. **EDX = 3**
* e. **EDX = 3**
* f. **EAX = 2**

**6. Register values after the second instruction sequence:**

**Instruction sequence**:

1. mov esi, OFFSET myBytes
   * ESI points to the start of myBytes.
2. mov ax, [esi]
   * AX gets the value at the address esi points to. myBytes[0] = 10h.
   * **AX = 10h**
3. mov eax, DWORD PTR myWords
   * EAX gets the value at the address myWords. This will load the first word (8Ah).
   * **EAX = 8Ah**
4. mov esi, myPointer
   * ESI points to myPointer, which is the address of myDoubles.
5. mov ax, [esi+2]
   * AX gets the value at esi + 2. myDoubles[2] = 3.
   * **AX = 3**
6. mov ax, [esi+6]
   * AX gets the value at esi + 6. myDoubles[1] = 2.
   * **AX = 2**
7. mov ax, [esi-4]
   * AX gets the value at esi - 4. myDoubles[0] = 1.
   * **AX = 1**

So, the answers are:

* a. **AX = 10h**
* b. **EAX = 8Ah**
* c. **AX = 3**
* d. **AX = 2**
* e. **AX = 1**

**Notes on JMP and LOOP Instructions in Assembly Language:**

**Overview of Branching (Control Transfer)**:

* **Control transfer**: This means altering the usual sequential execution of instructions and jumping to a different part of the program.
* **Two types of transfers**:
  + **Unconditional transfer**: The control is transferred without any condition (e.g., the JMP instruction).
  + **Conditional transfer**: Control is transferred based on certain conditions, such as the value of CPU flags (e.g., JE for "jump if equal").

**1. JMP Instruction (Unconditional Transfer)**

* **Purpose**: The JMP instruction is used to jump to a new location in the program unconditionally.
* **Syntax**: JMP destination
  + The CPU will load the destination address into the instruction pointer, and execution will continue from there.
* **Example of an Endless Loop**:
  + The JMP instruction can be used to create a loop:

top:

; some code here

JMP top ; This will jump back to the 'top' label endlessly.

**2. LOOP Instruction (Conditional Transfer)**

* **Purpose**: The LOOP instruction repeats a block of code a certain number of times.
* **How it works**:
  + Uses **ECX** as a counter, which is automatically decremented by 1 after each loop.
  + When ECX reaches 0, the loop stops.
* **Syntax**: LOOP destination
  + The destination must be within -128 to +127 bytes of the current instruction.
* **Example**: To add 1 to AX five times:

mov ax, 0

mov ecx, 5

L1:

inc ax

LOOP L1 ; Repeat 5 times

After the loop, AX will be 5, and ECX will be 0.

* **Important Notes**:
  + **Common mistake**: Initializing ECX to 0 before the loop can cause an infinite loop.
  + **Jump range**: The loop has a limited jump range (128 to +127 bytes).
  + **Changing ECX in a loop**: Modifying ECX inside the loop can cause unexpected results, so it's better to save and restore ECX if needed.

**3. Nested Loops:**

* You can create loops inside loops, but be careful with the outer loop's counter (ECX).
* **Save ECX** before an inner loop if you need to modify it.
* **Example** of nested loops:

mov ecx, 100 ; Outer loop counter

L1:

mov count, ecx ; Save outer loop counter

mov ecx, 20 ; Inner loop counter

L2:

; inner loop code

loop L2 ; Repeat inner loop

mov ecx, count ; Restore outer loop counter

loop L1 ; Repeat outer loop

**4. Debugging: Displaying an Array:**

* **Memory Window in Debugger**: In Visual Studio, you can view an array's contents during debugging by opening the **Memory window** and entering the array name (e.g., &myArray).
* You can change the format of displayed data to hexadecimal or decimal.

**5. Summing an Integer Array:**

* To sum elements of an array:
  1. Load the array's address into a register.
  2. Set up a loop counter for the array's length.
  3. Initialize a register to accumulate the sum (e.g., EAX).
  4. Loop through the array, adding each element to the sum.
  5. Update the register to point to the next array element.
  6. Repeat the loop until all elements are processed.
* **Example** of summing an array:

**.data**

**intarray DWORD 10000h, 20000h, 30000h, 40000h**

**.code**

**mov edi, OFFSET intarray ; Load array address**

**mov ecx, LENGTHOF intarray ; Set loop counter**

**mov eax, 0 ; Set sum to 0**

**L1:**

**add eax, [edi] ; Add array element to sum**

**add edi, TYPE intarray ; Move to next element**

**loop L1 ; Repeat loop**

**6. Copying a String:**

* Strings in assembly are often copied byte-by-byte using a loop.
* **Procedure**:
  1. Load the source string's address.
  2. Load the target string's address.
  3. Use a loop to copy each character from source to target.
  4. Continue until the null terminator (0) is reached.
* **Example of copying a string**:

assembly

CopyEdit

.data

source BYTE "This is the source string", 0

target BYTE SIZEOF source DUP(0)

.code

mov esi, 0 ; Index for source string

mov ecx, SIZEOF source ; Length of the string

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 until the whole string is copied

**Summary**:

* **JMP** is used for unconditional jumps (e.g., creating infinite loops).
* **LOOP** is used for repeating instructions a specific number of times, based on the ECX register.
* Always ensure that ECX is properly initialized to avoid infinite loops or excessive repetitions.
* Nested loops require careful handling of outer loop counters.
* Assembly can be used to perform common tasks like summing arrays or copying strings, but these tasks need proper loop structures and register management.

**Section Review**

1. **(True/False): A JMP instruction can only jump to a label inside the current procedure.**
   * **False.** The JMP instruction can jump to any label, not just inside the current procedure.
2. **(True/False): JMP is a conditional transfer instruction.**
   * **False.** JMP is an unconditional transfer instruction, meaning it jumps without any condition.
3. **If ECX is initialized to zero before beginning a loop, how many times will the LOOP instruction repeat? (Assume ECX is not modified by any other instructions inside the loop.)**
   * The LOOP instruction will **repeat 4,294,967,296 times** (because ECX will be decremented from 0, and it will wrap around to the maximum value of 0xFFFFFFFF).
4. **(True/False): The LOOP instruction first checks to see whether ECX is not equal to zero; then LOOP decrements ECX and jumps to the destination label.**
   * **False.** The LOOP instruction first decrements ECX, then checks if it’s zero to decide whether to jump.
5. **(True/False): The LOOP instruction does the following: It decrements ECX; then, if ECX is not equal to zero, LOOP jumps to the destination label.**
   * **True.** This is how the LOOP instruction works.
6. **In real-address mode, which register is used as the counter by the LOOP instruction?**
   * In real-address mode, **CX** is used as the loop counter.
7. **In real-address mode, which register is used as the counter by the LOOPD instruction?**
   * In real-address mode, **ECX** is used as the loop counter by the LOOPD instruction.
8. **(True/False): The target of a LOOP instruction must be within 256 bytes of the current location.**
   * **False.** The target of the LOOP instruction must be within 128 to +127 bytes of the current location.
9. **(Challenge): What will be the final value of EAX in this example?**

mov eax,0

mov ecx,10

L1:

L2:

mov eax,3

mov ecx,5

add eax,5

loop L2

loop L1

* + The final value of **EAX** will be **8**. Here’s how it works:
    - The outer loop runs 10 times.
    - In each iteration, the inner loop runs 5 times, but the inner loop keeps setting **EAX to 3** and then adds 5 to it, making **EAX = 8** after the inner loop finishes each time. The inner loop doesn't change EAX permanently because it’s reset on each iteration.

1. **Revise the code from the preceding question so the outer loop counter is not erased when the inner loop starts.**

* Here’s a revised version:

mov eax,0

mov ecx,10

L1:

push ecx ; Save outer loop counter

mov ecx,5

L2:

mov eax,3

add eax,5

loop L2

pop ecx ; Restore outer loop counter

loop L1

* This version saves the outer loop counter in memory before starting the inner loop and restores it afterward. This prevents the outer loop counter from being overwritten by the inner loop.