Assembly Programming

Chapter 4: Data Transfers, Addressing, and Arithmetic

CSE3030
Prof. Youngjae Kim
분산 처리 및 운영체제 연구실
(Distributed Computing and Operating Systems Laboratory)
https://discos.sogang.ac.kr

Chapter 4: Data Transfers, Addressing and Arithmetic

- Data Transfer Instructions
  - Operand Types, Direct Memory Operands, MOV Instruction, etc
- Addition and Subtraction
  - INC and DEC Instructions, ADD/SUB/NEG Instructions, etc
- Data-Related Operators and Directives
  - OFFSET Operator, ALIGN Directive, PTR Operator, etc
- Indirect Addressing
  - Indirect Operands, Arrays, Indexed Operands, etc
- JMP and LOOP Instructions
  - JMP Instruction, LOOP Instruction, etc

4.1 Data Transfer Instructions

- Compilers (C, C++, Java, etc)
  - Perform strict type checking to help programmers avoid possible errors such as mismatching variable and data

- Assemblers
  - Give programmers more freedom to allow them to do just anything they want, as along as the processor's instruction set can do what they ask
    - It means assembly language forces programmers to pay attention to data storage and machine-specific details.
    - Therefore, programmers must understand the processor's limitations when they write assembly language code.

4.1 Data Transfer Instructions

- Operand Types
  - Immediate – uses a numeric literal expression
    - a constant integer (8, 16, or 32 bits)
    - Value is encoded within the instruction.
  - Register – uses a named register in the CPU
    - Register name is converted to a number and encoded within the instruction.
  - Memory – references a memory location
    - Memory address is encoded within the instruction, or a register holds the address of a memory location.

Instruction Operand Notation, 32-Bit Mode

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	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
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Direct Memory Operands

Variable names are references to offsets within the data segment.

var1 says that its size attribute is byte and it contains the value 10 hexadecimal.

```
.data
var1 BYTE 10h
```

```
mov al var1
```

A0 00010400

Machine instruction produced after the above instruction is assembled

Operation code (opcode)

32-bit hexadecimal address of var1

4.1 Data Transfer Instructions

- Direct Memory Operands
  - A direct memory operand is a named reference to storage in memory.
  - The named reference (label) is automatically dereferenced by the assembler.

```
var1 BYTE 10h
. . .
mov al,var1                 ; AL = 10h
mov al,[var1]               ; AL = 10h
```

alternate format

4.1 Data Transfer Instructions

- MOV Instruction
  - Move from source to destination. Syntax:

    MOV *destination,source*
  - Rules
    - Both operands must be the same size.
    - Both operands cannot be memory operands.
    - IP, EIP, RIP, or CS cannot be a destination operand.

In C++ or Java

dest = source;

MOV instruction formats:

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

Memory to Memory

```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax, var1
mov var2, ax
```

Overlapping Values

```
.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
```

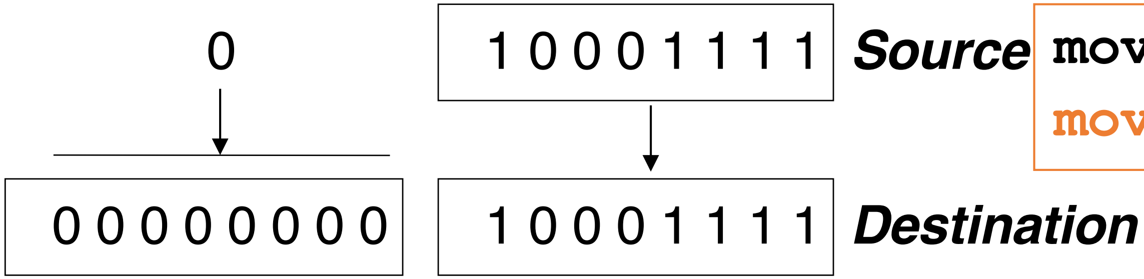
- Examples:

```
count BYTE 100
wVal  WORD 2
. . .
    mov bl,count
    mov ax,wVal
    mov count,al
    mov al,wVal                 ; error
    mov ax,count                ; error
    mov eax,count               ; error
```

- Explain why each of the following MOV statements are invalid:

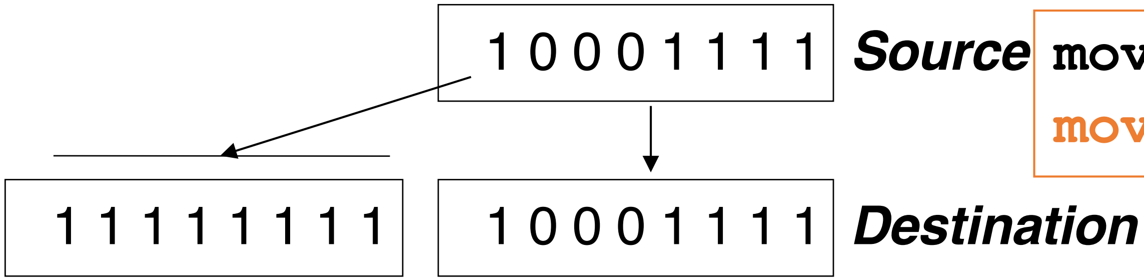
```
bVal   BYTE    100
bVal2  BYTE    ?
wVal   WORD    2
dVal   DWORD   5
. . .
    mov ds,45                ; a.
    mov esi,wVal             ; b.
    mov eip,dVal             ; c.
    mov 25,bVal              ; d.
    mov bVal2,bVal           ; e.
```

- Zero Extension (**MOVZX** instruction)




```
mov bl,10001111b
movzx ax,bl ; dest = reg
```

- Sign Extension (**MOVSX** instruction)




```
mov bl,10001111b
movsx ax,bl ; dest = reg
```

- XCHG Instruction

```
var1 WORD 1000h
var2 WORD 2000h
. . .
xchg ax,bx      ; exchange 16-bit regs
xchg var1,bx    ; exchange mem, reg
xchg var1,var2  ; error: two memory operands
```

- At least one operand must be a reg.
- No immediate operands are permitted.

4.1 Data Transfer Instructions

- Direct-Offset Operands
  - A constant offset is added to a data label to produce an effective address (EA).

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]          ; AX = 2000h
mov ax,[arrayW+4]          ; AX = 3000h
mov eax,[arrayD+4]         ; EAX = 00000002h
; Will the following statements assemble? Yes
mov ax,[arrayW-2]              ; Out of range.
mov eax,[arrayD+16]            ;
; What will happen when the above two run?
```

Some unkown values are loaded(usually 0)

- Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes          ; al= 80h
add al,[myBytes+1]      ; al=0E6h
add al,[myBytes+2]      ; al= 8Bh
```

ans
80h
0E6h
18Bh

- How about the following code?

```
mov ax,myBytes          ; assemble
add ax,[myBytes+1]      ; error
add ax,[myBytes+2]      ;    "
```

mov bx,0

- Correct code?

No!

bh may not be 0

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx   ; AX = sum
```

Correct code:

```
movzx ax,myBytes
movzx bx,[myBytes+1]
add   ax,bx
movzx bx,[myBytes+2]
add   ax,bx
```

4.2 Addition and Subtraction

- INC (increment) and DEC (decrement) Instructions
- ADD and SUB Instructions
- NEG (negate) Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow
- Multiplication and division later in Chapter 7
- Floating point arithmetic in Chapter 12

4.2 Addition and Subtraction

- INC and DEC Instructions
  - Add 1, subtract 1 from destination operand
    - operand may be register or memory
  - **INC *dest*** : $dest \leftarrow dest + 1$
  - **DEC *dest*** : $dest \leftarrow dest - 1$
  - Example

```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
 inc myWord  ; 1001h
 dec myWord  ; 1000h
 inc myDword ; 10000001h
 mov ax,00FFh
 inc ax      ; AX = 0100h
 mov ax,00FFh
 inc al      ; AX = 0000h
```

```
.data
myByte BYTE 0FFh, 0
.code
  mov al,myByte       ; AL = FFh
  mov ah,[myByte+1]   ; AH = 00h
  dec ah ; AH = FFh
  inc al ; AL = 00h
  dec ax ; AX = FEFFh
```

4.2 Addition and Subtraction

- ADD and SUB Instructions
  - **`ADD dest, src`** : *dest* ← *dest* + src
  - **`SUB dest, src`** : *dest* ← *dest* – src
  - The same operand rules as for the MOV instruction.
  - Example

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                          ; ---EAX---
      mov eax,var1             ; 00010000h
      add eax,var2             ; 00030000h
      add ax,0FFFFh            ; 0003FFFFh
      add eax,1                ; 00040000h
      sub ax,1                 ; 0004FFFFh
```

4.2 Addition and Subtraction

- NEG (negate) Instruction
  - Reverses the sign of an operand. Operand can be a register or memory operand (Internally, **`SUB 0,operand`**)
  - Any nonzero operand causes the Carry flag to be set.
  - Example:

```
.data
valB BYTE -1
valW WORD +32767
.code
 mov al,valB ; AL = -1
 neg al      ; AL = +1
 neg valW    ; valW = -32767
 mov ax, -32768 ; AX= 8000h
 neg ax         ; AX= 8000h
```

```
.data
valB BYTE 1,0
valC SBYTE -128
.code
 neg valB       ; CF = 1, OF = 0
 neg [valB + 1]; CF = 0, OF = 0
 neg valC       ; CF = 1, OF = 1
```

- Implementing Arithmetic Expressions
  - Example : **Rval = -Xval + (Yval – Zval)**

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
   mov eax,Xval
   neg eax                     ; EAX = -26
   mov ebx,Yval
   sub ebx,Zval                ; EBX = -10
   add eax,ebx
   mov Rval,eax                ; -36
```

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

```
   mov ebx,Yval
   neg ebx
   add ebx,Zval
   mov eax,Xval
   sub eax,ebx
   mov Rval,eax
```

4.2 Addition and Subtraction

- Flags Affected by Arithmetic
  - The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations.
    - based on the contents of the destination operand.
  - Essential flags:
    - Zero flag – destination equals zero.
    - Sign flag – destination is negative.
    - Carry flag – unsigned value out of range.
    - Overflow flag – signed value out of range.
  - The MOV instruction never affects the flags.

4.2 Addition and Subtraction

- Zero Flag (ZF)
  - Whenever the destination operand equals Zero, the Zero flag is set.

```
mov cx,1
sub cx,1        ; CX = 0, ZF = 1 (set)
mov ax,0FFFFh
inc ax          ; AX = 0, ZF = 1
inc ax          ; AX = 1, ZF = 0 (clear)
```

- Sign Flag (SF)
  - The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
; SF is a copy of the dest's msb.
mov cx,0
sub cx,1                     ; CX = -1, SF = 1
add cx,2                     ; CX = 1, SF = 0
mov al,0
sub al,1                 ; AL = 11111111b, SF = 1
add al,2                 ; AL = 00000001b, SF = 0
```

4.2 Addition and Subtraction

- Signed and Unsigned Integers : A Hardware View
  - All CPU instructions operate exactly the same on signed and unsigned integers.
  - The CPU cannot distinguish between signed and unsigned integers.
  - YOU, the programmer, are solely responsible for using the correct data type with each instruction.

4.2 Addition and Subtraction

- Carry Flag (CF)
  - The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1                     ; CF = 1, AL = 00
; Try to go below zero:
mov al,0
sub al,1                     ; CF = 1, AL = FF
```

```
mov ax,00FFh
add ax,1      ; AX= 0100h  SF= 0  ZF= 0  CF= 0
sub ax,1      ; AX= 00FFh  SF= 0  ZF= 0  CF= 0
add al,1      ; AL= 00h    SF= 0  ZF= 1  CF= 1
mov bh,6Ch
add bh,95h    ; BH= 01h    SF= 0  ZF= 0  CF= 1
mov al,2
sub al,3      ; AL= FFh    SF= 1  ZF= 0  CF= 1
```

Addition and Subtraction

- Overflow Flag (OF)
  - The Overflow flag is set when the signed result of an operation is invalid or out of range.
  - Example:

```
; Example 1
mov al,+127
add al,1          ; OF = 1,    AL = ??
; Example 2
mov al,7Fh
add al,1          ; OF=1,      AL = 80h
```

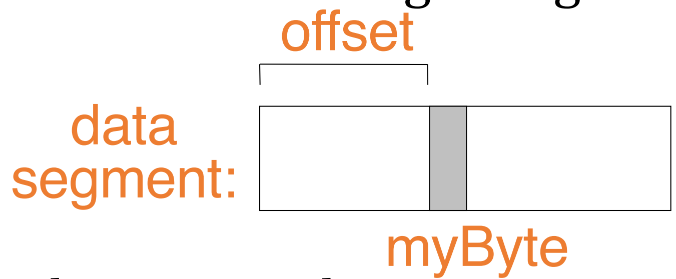
```
mov al,-128
neg al                  ; CF = 1    OF = 1
mov ax,8000h
add ax,2                ; CF = 0    OF = 0
mov ax,0
sub ax,2                ; CF = 1    OF = 0
```

4.3 Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

4.3 Data-Related Operators and Directives

- OFFSET Operator
  - OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
    - Protected mode: 32 bits
    - Real mode: 16 bits



  - The Protected-mode programs we write only have a single segment (we use the flat memory model).
  - Example:

Assume that the data segment begins at 00404000h.

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?
.code
mov esi,OFFSET bVal   ; ESI = 00404000
mov esi,OFFSET wVal   ; ESI = 00404001
mov esi,OFFSET dVal   ; ESI = 00404003
mov esi,OFFSET dVal2  ; ESI = 00404007
```

4.3 Data-Related Operators and Directives

- Relating to C/C++
  - The value returned by OFFSET is a pointer. Compare the following two codes written for both C++ and assembly language:

```
// C++ version:
char array[1000];
char * p = array;
```

```
; ASM version
.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array     ; ESI is p
```

4.3 Data-Related Operators and Directives

- PTR Operator
  - Overrides the default type of a label (variable). Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                   ; error – why?
mov ax,WORD PTR myDouble          ; loads 5678h
mov ax,WORD PTR [myDouble+2]      ; loads 1234h
mov WORD PTR myDouble, 9999h      ; saves 9999h
```

  - Little Endian Order
    - Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
    - For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

4.3 Data-Related Operators and Directives

- PTR Operator Examples

```
.data
myDouble DWORD 12345678h
```

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

```
mov al,BYTE PTR myDouble          ; AL = 78h
mov al,BYTE PTR [myDouble+1]      ; AL = 56h
mov al,BYTE PTR [myDouble+2]      ; AL = 34h
mov ax,WORD PTR [myDouble]        ; AX = 5678h
mov ax,WORD PTR [myDouble+2]      ; AX = 1234h
```

- PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.data
myBytes BYTE 12h,34h,56h,78h
.code
mov ax,WORD PTR [myBytes]    ; AX = 3412h
mov ax,WORD PTR [myBytes+2] ; AX = 7856h
mov eax,DWORD PTR myBytes    ; EAX = 78563412h
```

- Example:

```
.data
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h
.code
mov ax,WORD PTR [varB+2] ; a. 0502h
mov bl,BYTE PTR varD      ; b. 78h
mov bl,BYTE PTR [varW+2] ; c. 02h
mov ax,WORD PTR [varD+2] ; d. 1234h
mov eax,DWORD PTR varW    ; e. 12026543h
```

4.3 Data-Related Operators and Directives

- TYPE Operator returns the size, in bytes, of a single element of a data declaration.
- LENGTHOF Operator counts the number of elements in a single data declaration.
- SIZEOF Operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data
var1 BYTE ?
var2 WORD ?
array1 WORD 30 DUP(?),0,0   ; 32
var4 QWORD ?
.code
mov eax, TYPE var1          ; 1
mov eax, TYPE var2          ; 2
mov ecx, LENGTHOF array1    ; 32
mov ecx, SIZEOF array1      ; 64
```

4.3 Data-Related Operators and Directives

- Spanning Multiple Lines
  - A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration.

```
.data
array WORD 10,20,
   30,40
array1 WORD 50,60
       WORD 70, 80
.code
mov eax,LENGTHOF array          ; 4
mov ebx,SIZEOF array            ; 8
mov eax,LENGTHOF array1         ; 2
mov ebx,SIZEOF array1           ; 4
```

4.3 Data-Related Operators and Directives

- LABEL Directive
  - Assigns an alternate label name and type to an existing storage location
  - LABEL does not allocate any storage of its own
  - Removes the need for the PTR operator

```
.data
dwList   LABEL DWORD
wordList LABEL WORD
intList  BYTE 00h,10h,00h,20h
.code
 mov eax, dwList         ; 20001000h
 mov cx,  wordList       ; 1000h
 mov dl,  intList        ; 00h
```

4.4 Indirect Addressing

- Indirect Operands
- Array Sum Example
- Indexed Operands
- Pointers

4.4 Indirect Addressing

- Indirect Operands
  - An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer).
  - Use PTR when the size of a memory operand is ambiguous.

```
.data
val1 BYTE 10h,20h,30h
myCount WORD 0
.code
mov esi,OFFSET val1
mov al,[esi]        ; dereference ESI (AL = 10h)
inc esi
mov al,[esi]        ; AL = 20h
inc esi
mov al,[esi]        ; AL = 30h
mov esi,OFFSET myCount
inc [esi]                ; error: ambiguous
inc WORD PTR [esi]       ; ok (use PTR)
```

4.4 Indirect Addressing

- Array Sum Example
  - Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
arrayW WORD 1000h,2000h,3000h
.code
   mov esi, OFFSET arrayW
   mov ax, [esi]
   add esi, 2      ; or: add esi, TYPE arrayW
   add ax, [esi]
   add esi, 2
   add ax, [esi]   ; AX = sum of the array
```

4.4 Indirect Addressing

- Indexed Operands
  - An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:

[*label* + *reg*] or ***label*[*reg*]**

```
.data
arrayW WORD 1000h,2000h,3000h
.code
   mov esi,0
   mov ax, [arrayW + esi]   ; AX = 1000h
   mov ax, arrayW[esi]      ; alternate format
   add esi, 2
   add ax, [arrayW + esi]
   etc.
```

4.4 Indirect Addressing

- Pointers
  - You can declare a pointer variable that contains the offset of another variable.

```
.data
arrayW WORD  1000h,2000h,3000h
ptrW   DWORD arrayW
.code
   mov esi,ptrW
   mov ax,[esi]         ; AX = 1000h
```

Alternate format:

```
ptrW DWORD OFFSET arrayW
```

4.5 JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String

4.5 JMP and LOOP Instructions

- JMP Instruction
  - JMP is an unconditional jump to a label that is usually within the same procedure.
  - Syntax: ***JMP target***
  - Logic: **EIP ← *target***
  - Example:

```
top:
    .
    .
    jmp top
```

  - A jump outside the current procedure must be to a special type of label called a global label (see Section 5.5.2.3 for details).

4.5 JMP and LOOP Instructions

- LOOP Instruction
  - The LOOP instruction creates a counting loop
  - Syntax: **LOOP *target***
  - Logic: **ECX ← ECX – 1**
    if ECX != 0, jump to *target*
  - Implementation:
    - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset.
    - The relative offset is added to EIP.

4.5 JMP and LOOP Instructions

- LOOP Example
- The following loop calculates the sum of the integers 5 + 4 + 3 +2 + 1:

offset	machine code		source code
00000000	66 B8 0000		mov ax,0
00000004	B9 00000005		mov ecx,5
00000009	66 03 C1	L1:	add ax,cx
0000000C	E2 FB		loop L1
0000000E			

When LOOP is assembled, the current location = `0000000E` (offset of the next instruction). `–5` (`FBh`) is added to the the current location, causing a jump to location `00000009`:

`00000009 ← 0000000E + FB`

4.5 JMP and LOOP Instructions

- If the relative offset is encoded in a single signed byte,
  - what is the largest possible backward jump? -128
  - what is the largest possible forward jump? +127
- What will be the final value of AX?

```
    mov ax,6
    mov ecx,4
L1:
    inc ax
    loop L1
```

10

- How many times will the loop execute?

4,294,967,296

```
    mov ecx,0
X2:
    inc ax
    loop X2
```

4.5 JMP and LOOP Instructions

- Nested Loop
  - If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.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
```

4.5 JMP and LOOP Instructions

- Summing an Integer Array
  - The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
   mov edi,OFFSET intarray ; address of intarray
   mov ecx,LENGTHOF intarray  ; loop counter
   mov ax,0                   ; zero the accumulator
L1:
   add ax,[edi]             ; add an integer
   add edi,TYPE intarray ; point to next integer
   loop L1                  ; repeat until ECX = 0
```

4.5 JMP and LOOP Instructions

- Copying a String
  - The following code copies a string from source to target:

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

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

  - Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.