

Lab 4: Basic Instructions and Addressing Modes

Contents

- 4.1. Data Transfer Instructions
- 4.2. Addition and Subtraction
- 4.3. Data Addressing Modes
- 4.4. LOOP Instruction
- 4.5. Copying a String
- 4.6. Summing an Array of Integers

4.1 Data Transfer Instructions

Data transfer instructions move data between registers or between registers and memory. These instructions are briefly described below. For more details, refer to the lecture notes or your textbook. The following program demonstrates the **MOV**, **MOVZX**, **MOVSX**, and **XCHG** instructions:

MOV	<i>destination, source</i>	Move <i>source</i> to <i>destination</i> .
MOVZX	<i>destination, source</i>	Move <i>source</i> to <i>destination</i> with zero extension.
MOVSX	<i>destination, source</i>	Move <i>source</i> to <i>destination</i> with sign extension.
XCHG	<i>destination, source</i>	Exchange <i>source</i> with <i>destination</i> .

TITLE Data Transfer Examples (File: moves.asm)
; Demonstration of MOV, MOVZX, MOVSX, and XCHG

```
.686
.MODEL flat, stdcall
.STACK
INCLUDE Irvine32.inc

.data
var1 WORD 1000h
var2 WORD 2000h

.code
main PROC
; Demonstrating MOV and MOVZX
    mov     ax, 0A69Bh
    movzx   bx, al
    movzx   ecx, ah
    movzx   edx, ax

; Demonstrating MOVSX
    movsx   bx, al
    movsx   ecx, ah
    movsx   edx, ax

; Demonstrating XCHG
    xchg    ax, var1
    xchg    ax, var2
    xchg    ax, var1

    exit
main ENDP
END main
```

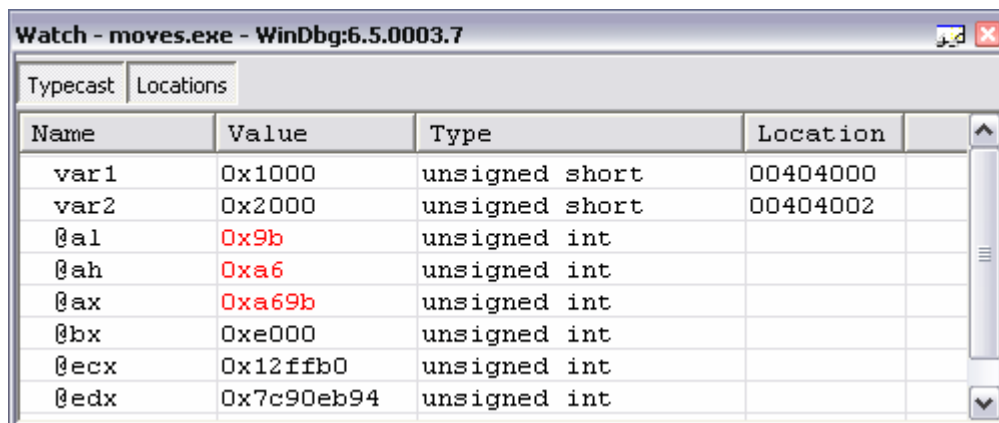
4.1.1 Lab Work: Assemble and Link moves.asm

Open file *moves.asm* and assemble and link the file. You can use the **make32** batch file from the command prompt or from the ConTEXT editor's **Tools** menu.

4.1.2 Lab Work: Trace the Execution of Program moves.exe

Now run the Windows debugger to trace the execution of the above program. You may open the debugger from ConTEXT **Tools** menu or by typing: **windbg –QY–G moves.exe**

Open the **Watch window** to view the **var1** and **var2** variables. You can also watch registers under the Watch window by typing their names preceded with the **@** symbol as shown below. Observe that the type of registers is *unsigned int* and the value is shown in hexadecimal. You can change the type of a register to *char*, *short*, or *int* (depending on its size), to see its value as a signed decimal.



Name	Value	Type	Location
var1	0x1000	unsigned short	00404000
var2	0x2000	unsigned short	00404002
@al	0x9b	unsigned int	
@ah	0xa6	unsigned int	
@ax	0xa69b	unsigned int	
@bx	0xe000	unsigned int	
@ecx	0x12ffb0	unsigned int	
@edx	0x7c90eb94	unsigned int	

Try first to guess the values of the registers and memory variables in the above program after the execution of each instruction. Write your answers in hexadecimal in the specified boxes. Place the cursor at the beginning of the *main* procedure and press **F7**. Now step through the program by pressing **F10** and watch the changes in registers and variables. Make the necessary corrections to your answers.

MOV and MOVZX

1) al, ah (hex) =

2) bx (hex) =

3) ecx (hex) =

4) edx (hex) =

MOVSX

5) bx (hex) =

6) ecx (hex) =

7) edx (hex) =

XCHG

8) ax (hex) =

8) var1 (hex) =

9) ax (hex) =

9) var2 (hex) =

10) ax (hex) =

10) var1 (hex) =

4.2 Addition and Subtraction

Integer addition and subtraction are two of the most fundamental operations that a processor can perform. In the following Program, you will learn about the **INC**, **DEC**, **ADD**, **SUB**, and **NEG** instructions. You will also learn about the flags affected by these arithmetic instructions:

INC <i>destination</i>	Increment <i>destination</i> by 1.
DEC <i>destination</i>	Decrement <i>destination</i> by 1.
ADD <i>destination, source</i>	Add <i>source</i> to <i>destination</i> .
SUB <i>destination, source</i>	Subtract <i>source</i> from <i>destination</i> .
NEG <i>destination</i>	Negate <i>destination</i> by computing 2's complement.
CF	Carry Flag: set if unsigned result is out of range.
OF	Overflow Flag: set if signed result is out of range.
SF	Sign Flag: set if sign bit of result is negative (or 1).
ZF	Zero Flag: set if result is zero (all bits are zero).
PF	Parity Flag: set if low byte of result has even number of 1's.

The **ADD**, **SUB**, and **NEG** instructions affect all the above flags. The **INC** and **DEC** instructions also affect the above flags, except that the carry flag is not modified. For more details about these instructions and flags, refer to the lecture notes or your textbook.

```

TITLE   Simple Arithmetic           (SimpleArith.asm)
.686
.MODEL flat, stdcall
.STACK
INCLUDE Irvine32.inc
.data ; No data

.code
main PROC
    ; ADD
    mov eax, 91ab0748h
    mov ebx, 3f54f8f2h
    add eax, ebx

    ; SUB
    mov eax, 91ab0748h
    sub eax, ebx

    ; NEG
    mov eax, 91ab0748h
    neg eax

    ; INC
    cld ; clear carry flag to show that it is not affected
    mov eax, 7fffffffh
    inc eax

    ; DEC
    mov eax, 0
    dec eax
    exit
main ENDP
END main

```

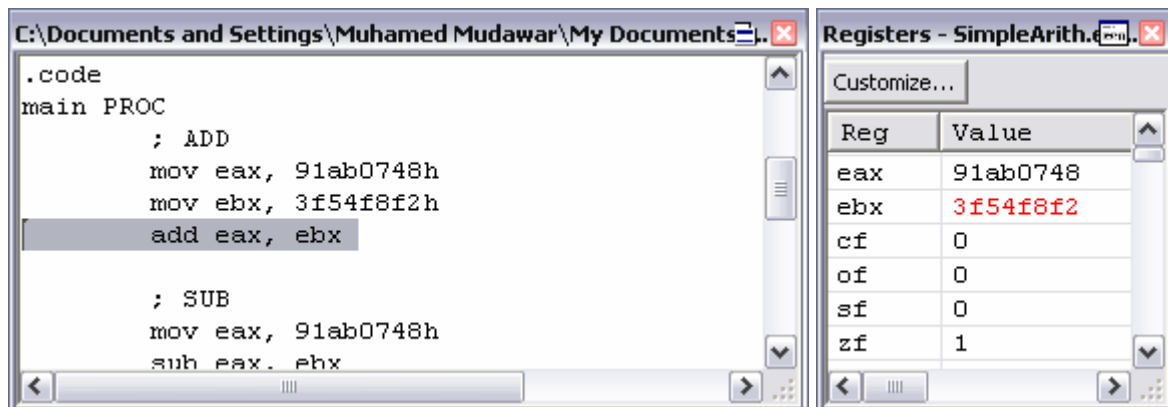
4.2.1 Lab Work: Assemble and Link SimpleArith.asm

4.2.2 Lab Work: Trace the Execution of SimpleArith.exe

First, guess the values of the **eax** register (in hexadecimal) and the **cf**, **of**, **sf**, **zf**, and **pf** flags after executing the **add**, **sub**, **neg**, **inc**, and **dec** instructions. Write these values below:

1) EAX after ADD (hex) =	CF=	OF=	SF=	ZF=	PF=
2) EAX after SUB (hex) =	CF=	OF=	SF=	ZF=	PF=
3) EAX after NEG (hex) =	CF=	OF=	SF=	ZF=	PF=
4) EAX after INC (hex) =	CF=	OF=	SF=	ZF=	PF=
5) EAX after DEC (hex) =	CF=	OF=	SF=	ZF=	PF=

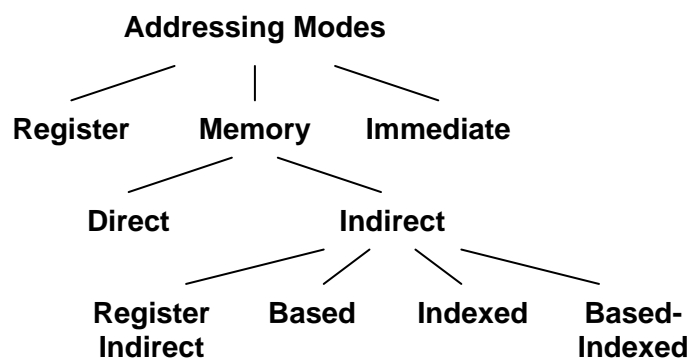
Run the 32-bit Windows Debugger. Open the source file *SimpleArith.asm* from the **File** menu if it is not already opened. Watch the registers by selecting **Registers** from the **View** menu. Have the registers **eax**, **ebx** and the flags **cf**, **of**, **sf**, **zf**, and **pf** on top of the list. Place the cursor at the beginning of *main* procedure and press **F7** to start debugging it. Press **F10** to step through the execution of the program. Watch the changes in the registers and flags.



Check the answers that you have guessed above. Make the necessary corrections and try to understand why your previous answer was incorrect.

4.3 Data Addressing Modes

The assembly language instructions require the specification of the location of data for source and destination operands. The specification of the location of data is called the **data addressing mode**. It can be classified as shown in the following diagram:



Register addressing is when a register is used to specify the source or destination of an operand. This is the most efficient addressing mode because registers are implemented inside the processor and their access is very fast.

Immediate addressing is when an immediate value (a constant) is used for a source operand. It cannot be used to specify a destination operand. The immediate constant is part of the instruction itself.

Memory addressing is used to specify the address of the source and destination operands located in memory. This is the most detailed and interesting addressing mode. It can be divided into **direct** and **indirect** memory addressing. **Direct** memory addressing is when the address of a memory operand is specified directly by name. For example:

`mov sum, eax ; sum is a variable in memory`

Direct memory addressing is useful for accessing simple variables in memory, but it is useless for addressing arrays or data structures. To address the elements of an array, we need to use a register as a pointer to the array elements. This is called **indirect memory addressing**. It can be further classified into **register indirect**, **based**, **indexed**, and **based-indexed**, depending on how the address of the memory operand is specified. In general, a memory address can be specified as follows:

Address = [BaseReg + IndexReg * Scale + Disp]

This is the most general indirect memory addressing called **based-indexed**, because it combines a **base register** with an **index register** and a **displacement**. The other indirect memory addressing modes: **register indirect**, **based**, and **indexed** are simpler.

Register Indirect Addressing: **Address = [Reg]**

Based Addressing: **Address = [BaseReg + Disp]**

Indexed Addressing: **Address = [IndexReg * Scale + Disp]**

The IA-32 processor architecture supports 32-bit addressing as well as 16-bit addressing. 16-bit addressing modes originated in the 8086 processor, which supports only 16-bit registers. The 16-bit addressing modes use BX and BP for the base register and SI and DI for the index register. No scale factor is allowed, and displacements are limited to 16 bits.

With the advent of the IA-32 processor architecture, registers were extended from 16 bits to 32 bits. This enabled 32-bit addressing modes in addition to 16-bit addressing. Any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, or ESP) can be used as a base register. Any general-purpose register, with the exception of ESP can be used as an index register. A **scale factor** of 1, 2, 4, or 8 is added to indexed-addressing, and displacements are extended to 32-bits.

The differences between 16-bit and 32-bit addressing modes are summarized below:

	16-bit Addressing	32-bit Addressing
Base Register	BX, BP	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
Index Register	SI, DI	EAX, EBX, ECX, EDX, ESI, EDI, EBP
Scale Factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

4.3.1 Examples on 32-bit Addressing Modes

The following program demonstrates 32-bit memory addressing modes and the LEA (Load Effective Address) instruction. The **LEA** instruction moves the **address**, rather than the value, of a source operand into destination.

LEA destination, source Load Effective Address of *source* into *destination*

This is similar to moving the OFFSET of a variable. For example,

lea eax, variable is similar to **mov** eax, OFFSET variable

The **LEA** instruction provides more flexibility in terms of computing complex addresses at runtime. However, the **OFFSET** operator is used to determine the address of named variables at assembly-time.

TITLE Memory Addressing Examples (File: addressing.asm)

```
.686
.MODEL flat, stdcall
.STACK
INCLUDE Irvine32.inc
.data
arrayB BYTE "COE 205",0
arrayW WORD 100h,200h,300h, 400h
arrayD DWORD 01234567h,89ABCDEFh

.code
main PROC
; Direct Memory Addressing
    mov al, arrayB           ; same as [arrayB]
    mov ah, arrayB[5]        ; same as [arrayB+5]
    mov bx, arrayW[2]         ; same as [arrayW+2]
    mov ecx, [arrayD]         ; same as arrayD
    mov edx, [arrayD+2]       ; same as arrayD[2]

; Register Indirect Addressing
    mov ecx, OFFSET arrayB + 3
    mov edx, OFFSET arrayW + 1
    mov bx, [ecx]             ; address in [ecx]
    mov al, [edx]             ; address in [edx]

; Based Addressing
    mov edx, 4
    mov al, arrayB[edx]
    mov bx, arrayW[edx]
    mov ecx, arrayD[edx]

; Scaled Indexed Addressing
    mov esi, 1
    mov arrayB[esi*2], 'S'
    mov arrayW[esi*2], 102h
    mov arrayD[esi*4], 0

; Load Effective Address (LEA)
    lea eax, arrayB
    lea ebx, [eax + LENGTHOF arrayB]
    lea ecx, [ebx + esi*8]
    lea edx, arrayD
    exit
main ENDP
END main
```

4.3.2 Lab Work: Assemble and Link 'addressing.asm'

4.3.3 Lab Work: Trace the Execution of Program 'addressing.exe'

First, guess the values of the registers and memory variables in program *addressing.asm*. Write your answers in hexadecimal in the specified boxes after the execution of each instruction. For characters, show the character symbol as well as its code in hexadecimal.

Run the Windows Debugger. Open the source file *addressing.asm* from the **File** menu if it is not already opened. Watch the registers and memory by selecting them in the **View** menu. In the Memory window, write the name of the first variable *arrayB* in the Virtual address box. You may resize the Memory window so that exactly 16 bytes are displayed on each line.

Place the cursor at the beginning of *main* procedure and press **F7**. Press **F10** to step through the execution of the program. Watch the changes in the registers and memory.

Direct Memory Addressing

1) al =	2) ah =	3) bx =
4) ecx =	5) edx =	

Register Indirect Addressing

6) ecx =	7) edx =
8) bx =	9) al =

Based Addressing

11) al =	12) bx =	13) ecx =
----------	----------	-----------

Scaled Indexed Addressing – show the address and byte values in memory

15) arrayB address	byte	byte	byte	byte	byte	byte	byte	byte
16) arrayW address	byte	byte	byte	byte	byte	byte	byte	byte
17) arrayD address	byte	byte	byte	byte	byte	byte	byte	byte

Load Effective Address (LEA)

18) eax =	19) ebx =
20) ecx =	21) edx =

4.4 LOOP Instruction

The LOOP instruction provides a simple way to repeat a block of statements a specific number of times. It uses ECX as a counter, which is decremented each time the loop repeats. The execution of the LOOP instruction involves two steps: First, it subtracts 1 from ECX. If ECX is not equal to zero, a jump is taken to the *label* (start of the loop). Otherwise, the loop terminates and the next instruction is executed.

LOOP label Decrement ECX and Jump to *label* if ECX \neq 0

4.5 Copying a String

Study the following program:

```
TITLE Copying a String                      (File: CopyStr.asm)
; Demonstrates LOOP instruction and array indexing

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

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

.code
main PROC
    mov esi, 0                ; used to index source and target
    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                   ; increment index
    loop L1                   ; repeat for entire string

    exit
main ENDP
END main
```

What is the value of the target string in memory after finishing the execution of loop L1?

.....

What is the number of iterations for loop L1?

4.5.1 Lab Work: Assemble and Link CopyStr.asm

4.5.2 Lab Work: Trace the Execution of CopyStr.exe

Run the 32-bit Windows Debugger. View the registers **esi**, **ecx**, and **al**, as well as the *source* and *target* variables in memory. You can open two Memory windows to view separately the *source* and *target* strings.

Put the cursor at the beginning of *main* procedure and press **F7** to start debugging it. Press **F8** to trace the execution of the loop, iteration by iteration. If you press **F10** on the LOOP instruction, it will execute ALL the loop iterations and will terminate the loop. This is why it

is better here to use **F8** to trace the execution of the loop. View how the **esi**, **ecx**, and **ax** registers change, as well as memory for the *target* variable. Check your previous answers, make the necessary corrections, and try to understand your mistakes.

4.6 Summing an Array of Integers

Program *SumArray.asm* uses register **esi** as a **pointer** to *intarray*. Register-indirect addressing is used to access the elements of *intarray*.

```
TITLE Summing an Array of Signed Words    (File: SumArray.asm)
; Register-Indirect memory addressing is used

.686
.MODEL flat, stdcall
.STACK

INCLUDE Irvine32.inc

.data
intarray SWORD 5,7,-3,100,0,-9, 10 DUP(-999)
sum      SWORD ?

.code
main PROC

    mov     esi,OFFSET intarray    ; esi = pointer to intarray
    mov     ecx,LENGTHOF intarray ; ecx = loop counter
    mov     ax,0                  ; zero the accumulator
L1:
    add     ax,[esi]               ; register esi is a pointer
    add     esi,TYPE intarray      ; point to next integer
    loop    L1                    ; repeat until ECX = 0

    mov     sum, ax
    exit
main ENDP
END main
```

What is the value of *sum* (in decimal) at the end of the program?

How many iterations was loop L1 repeated?

Assuming the initial value of ESI is 404000h before starting loop L1, what is the final value of ESI (in hex) after finishing loop L1?

4.6.1 Lab Work: Assemble, Link, and Trace Program Execution

Open file *SumArray.asm* and assemble and link the file. Now run the Windows debugger to trace the execution of the above program. You need to view the registers **esi**, **ecx**, and **ax**. Add also a watch for the *sum* variable.

Put the cursor at the beginning of *main* procedure and press **F7** to start debugging this procedure. Press **F8** to trace the execution of the loop, iteration by iteration. View how the **esi**, **ecx**, and **ax** registers change. Press **F10** to exit the program. Check your answers and make the necessary corrections.

Review Questions

- For each of the following, write the destination register value (in hexadecimal) if the instruction is valid. Otherwise, indicate that the instruction is invalid. Assume that **var1** is at virtual address **404000h**.

```
var1 SBYTE -4, 2
var2 WORD 1000h, 2000h, 3000h
var3 DWORD 1, 2, 3, 4, 5
```

- | | | |
|----------|---------------|-------|
| a. mov | ax, var1 | |
| b. movzx | ax, var1 | |
| c. movsx | eax, var1 | |
| d. mov | ax, var2[2] | |
| e. mov | bx, var3 | |
| f. mov | edx, [var3+4] | |
| g. lea | esi, var2 | |
| h. mov | al, [esi] | |
| i. mov | ax, [esi] | |
| j. mov | eax, [esi] | |
| k. inc | [esi] | |

- (Yes/No) Is it possible to set the *Overflow flag* if you add a positive to a negative integer?
- (Yes/No) Is it possible for the **NEG** instruction to set the *Overflow flag*?
- (Yes/No) Is it possible for both the *Sign* and *Zero flags* to be set at the same time?
- (Yes/No) Can any 16-bit general-purpose register be used for indirect addressing?
- (Yes/No) Can any 32-bit general-purpose register be used for indirect addressing?

Programming Exercises

- Write a program that does the following:
 - Use the **ADD** and **SUB** instructions to set and clear the *Carry flag*.
 - Use the **ADD** and **SUB** instructions to set and clear the *Zero* and *Sign flags*.
 - Use the **ADD** and **SUB** instruction to set and clear the *Overflow flag*.
 - Use the **ADD** instruction to set and clear both the *Carry* and *Overflow flags*.

Trace program execution and Explain why the flags are affected by each instruction.
- Write a program that uses a loop to calculate the first ten values in the *Fibonacci* number sequence {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}. Place each value in the EAX register inside the loop, and trace it with the Windows debugger.
- Modify Program *SumArray.asm* to use the register **esi** as an **index** to *intarray* with a scale factor of 2, instead of a pointer. Trace it with the Windows debugger.
- Modify Program *CopyStr.asm* to use the register **esi** as a **pointer** (indirect addressing), instead of an **index**, to copy the characters from *source* to *target*, but in reverse order.