LABORATORY 4:
IMPLEMENTING FUNCTIONS IN AN ASSEMBLY PROGRAM

NAME:                                STUDENT ID#:

Objectives
Learn how to:
- Describe a function that is to be performed with a program.
- Write a program to implement the function.
- Run the program to verify that it performs the function for which it was written.

Part 1: Description of the Problem – Sorting a Table of Data
Sort an array of 16-bit signed binary numbers so that they are arranged in ascending order. For instance, if the original array is
5,  1,  29,  15,  38,  3,  -8,  -32
after sorting, the array that results would be
-32,  -8,  1,  3,  5,  15,  29,  38
Assume that the array of numbers is stored at consecutive memory locations from addresses A40016 through A41E16. Write a sort program to carry out this task.

Part 2: Writing the Program
First, we will develop an algorithm that can be used to sort an array of element A(0), A(1), A(2), through A(N) into ascending order. One way of doing this is to take the first number in the array, which is A(0), and compare it to the second number, A(1). If A(0) is greater than A(1), the two number are swapped; otherwise, they are left alone. Next, A(0) is compared to A(2) and, based on the result of this comparison, they are either swapped or left alone. This sequence is repeated until A(0) has been compared with all numbers up through A(N). When this is complete, the smallest number will be in the A(0) position.

Now A(1) must be compared to A(2) through A(N) in the same way. After this is done, the second smallest number is in the A(1) position. Up to this point, just two of the N
numbers have been put in ascending order. Therefore, the procedure must be continued for A(2) through A(N-1) to complete the sort.

Figure L4.1(a) illustrates the use of this algorithm for an array with just four numbers. The numbers are A(0) = 5, A(1) = 1, A(2) = 29, and A(3) = -8. During the sort sequence, A(0) = 5 is first compared to A(1) = 1. Since 5 is greater than 1, A(0) and A(1) are swapped. Now A(0) = 1 is compared to A(2) = 29. This time 1 is less than 29; therefore, the numbers are not swapped and A(0) remains equal to 1. Next A(0) = 1 is compared with A(3) = -8. A(0) is greater than A(3). Thus A(0) and A(3) are swapped and A(0) becomes equal to -8. Notice in figure L4.1 (a) that the lowest of the four numbers now resides in A(0).

The sort sequence in figure L4.1(a) continues with A(1) = 5 being compared first to A(2) = 29 and then to A(3) = 1. In the first comparison, A(1) is less than A(2). For this reason, their values are not swapped. But in the second comparison, A(1) is greater than A(3); therefore, the two values are swapped. In this way, the second lowest number, which is 1, is sorted into A(1).

It just remains to sort A(2) and A(3). Comparing these two values, we see that 29 is greater than 5. This causes the two values to be swapped so that A(2) = 5 and A(3) = 29. As shown in figure L4.1(a), the sorting of the array is now complete.

Now we will implement the algorithm on the 80 x 86 microprocessor. The flowchart for its implementation is shown in figure L4.1(b).

<table>
<thead>
<tr>
<th>I</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(0)</td>
<td>5</td>
<td>1</td>
<td>29</td>
<td>-8</td>
<td>Original array</td>
</tr>
<tr>
<td>A(1)</td>
<td>1</td>
<td>5</td>
<td>29</td>
<td>-8</td>
<td>Array after comparing A(0) and A(1)</td>
</tr>
<tr>
<td>A(2)</td>
<td>1</td>
<td>5</td>
<td>29</td>
<td>-8</td>
<td>Array after comparing A(0) and A(2)</td>
</tr>
<tr>
<td>A(3)</td>
<td>-8</td>
<td>5</td>
<td>29</td>
<td>1</td>
<td>Array after comparing A(0) and A(3)</td>
</tr>
<tr>
<td>A(1)</td>
<td>-8</td>
<td>5</td>
<td>29</td>
<td>1</td>
<td>Array after comparing A(1) and A(2)</td>
</tr>
<tr>
<td>A(2)</td>
<td>-8</td>
<td>1</td>
<td>29</td>
<td>5</td>
<td>Array after comparing A(1) and A(3)</td>
</tr>
<tr>
<td>A(3)</td>
<td>-8</td>
<td>1</td>
<td>5</td>
<td>29</td>
<td>Array after comparing A(2) and A(3)</td>
</tr>
</tbody>
</table>

Figure L4.1 (a) Sort algorithm demonstration

The first block in the flowchart represents the initialization of data segment register DS and pointers PNTR1 and PNTR3. The DS register is initialized with the value DATA_SEG to define a data segment that contains the beginning and ending addresses of the array to be sorted. PNTR1 points to the first element of data in the array. It is register SI and is initialized to the value ARRAY_BEG. For pointer PNTR3, we use register BX and initialize it with value ARRAY_END. It points to the last element in the array. Next, PNTR2, the
moving pointer, is initialized so that it points to the second element in the array. Register DI is used to hold this pointer. This leads to the following instruction sequence for initialization.

```assembly
MOV AX, DATA_SEG
MOV DS, AX
MOV SI, ARRAY_BEG
MOV BX, ARRAY_END
AA: MOV DI, SI
ADD DI, 2
```

![Flowchart](image)

Figure L4.1 (b) Flowchart for the sort program. (c) Program.
Notice that DS was loaded via AX with DATA_SEG to define the data segment. SI and BX, which are PNTR1 and PNTR3, respectively, are loaded with immediate operands ARRAY_BEG and ARRAY_END. In this way they point to the first and last elements of the array, respectively. Finally, register DI, which is PNTR2, is loaded with $0400_{16}$ from SI and then increased by two with an ADD instruction so that it points to the second element in the array. This completes the initialization process.

Next, the array element pointed to by PNTR1 is to be compared to the element pointed to by PNTR2. If the element corresponding to PNTR1 is arithmetically less than the element corresponding to PNTR2, the two elements are already in ascending order. But if this is not the case, the two elements must be interchanged. Both of these elements are in memory. However, the MPU cannot directly compare two values in memory. For this reason, one of the two elements must be moved to a register within the microprocessor. We use AX for this purpose. The resulting code is as follows:

```
BB:  MOV AX, [SI]
CMP AX, [DI]
JLE CC
MOV DX, [DI]
MOV [SI], DX
MOV [DI], AX
```

The first instruction moves the element pointed to by PNTR1 into AX. The second instruction compares the value in AX with the element pointed to by PNTR2. The result of this comparison is reflected in the status flags. The jump on less-than or equal-to instruction that follows checks if the first element is arithmetically less than or equal to the second element. If the result of this check is yes, control is transferred to CC. CC is a label to be used in the segment of program that will follow. If the check fails, the two elements must be interchanged. In this case, the instructions executed next move the element pointed to by PNTR2 into the location pointed to by PNTR1. Then the copy of the original value pointed to by PNTR1, which is saved in AX, is moved to the location pointed to by PNTR2.

To continue sorting through the rest of the elements in the array, we update PNTR2 so that it points to the next element. The comparison is repeated until the first element has been compared to each of the other elements in the array. This condition is satisfied when PNTR2 points to the last element in the array. That is, PNTR2 equals PNTR3. This part of the program can be done with the code that follows:

```
CC: INC DI
INC DI
CMP DI, BX
JBE BB
```
The first two instructions update PNTR2 so it points to the next element. The third instruction compares PNTR2 to PNTR3 to determine whether or not they are equal. If they are equal to each other, the first element has been compared to the last element and we are ready to continue with the second element. Otherwise, we must repeat from the label BB. This test is done with the jump on below or equal instruction. Notice that label BB corresponds to the beginning of the part of the program that compares the elements of the array. Once we fall through the JBE instruction, we have placed the smallest number in the array into the position pointed to by PNTR1.

To process the rest of the elements in the array in a similar way, PNTR1 must be moved over the entire range of elements and the foregoing procedure must be repeated. This can be done by implementing the code that follows:

```
INC SI
INC SI
CMP SI, BX
JB AA
NOP
```

The first two instructions increment PNTR1 so that it points to the next element in the array. The third instruction checks if all the elements have been sorted. The fourth instruction passes control back to the sorting sequence of instructions if PNTR1 does not point to the last element. However, if all elements of the array have been sorted, we come to a halt at the end of the program. The entire program appears in figure L4.1(c).

**Part 3: Running the Program**

The sort algorithm we just developed is implemented by the source program LAB04.ASM. This program was assembled and linked to produce a run module in file LAB04.EXE. The source listing produced by the assembler during the assembly process is LAB04.LST. Now we will run the program on the PC for an arbitrary set of data points. *Check off each step as it is completed.*

<table>
<thead>
<tr>
<th>Check</th>
<th>Step</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.</td>
<td>Load the program LAB04.EXE with the command</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>C:\&gt;DEBUG LAB04.EXE (-)</code></td>
</tr>
<tr>
<td></td>
<td>2.</td>
<td>Verify loading of the program by disassembling the contents of memory starting at the current code segment.</td>
</tr>
</tbody>
</table>
3. Execute the program according to the instructions that follow:
   a. Enter the following 16-decimal data values as 16-bit numbers starting at
      the address defined by ARRAY_BEG.
      5, 0, -3, 1, 12, -20, 77, 2, 9, -2, 53, -5, 1, 28, 15, 19.
   b. Verify the 16 data values by dumping them.
   c. Execute the program from the start till CS:2D.
   d. Dump the 16 data values starting from the address ARRAY_BEG. Are
      the numbers sorted? ________________________________________
   e. Execute the program to completion.

Part 4: Description of the Problem – Generating a Fibonacci Series

Write a program to generate the first ten elements of a Fibonacci series. In this series, the
first and second elements are zero and one, respectively. Each element that follows is
obtained by adding the previous two elements. Use a subroutine to generate the next element
from the previous two elements. Store the elements of the series starting at address FIBSER.

Part 5: Writing the Program

Our plan for the solution of this problem is shown in figure L4.2(a). This flowchart shows
the use of a subroutine to generate an element of the series, store it in memory, and prepare
for generation of the next element.

The first step in the solution is initialization. It involves setting up a data segment,
generating the first two numbers of the series, and storing them at memory locations with
offset addresses FIBSER and FIBSER+1. Then a pointer must be established to address the
locations for other terms of the series. This address will be held in the DI register. Finally, a
counter with initial value equal to 8 can be set up in CX to keep track of how many numbers
remain to be generated. The instructions needed for initialization are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX, DATA_SEG</td>
<td>Set up data segment address</td>
</tr>
<tr>
<td>MOV DS, AX</td>
<td>Set up data segment base</td>
</tr>
<tr>
<td>MOV NUM1, 0</td>
<td>Set first Fibonacci number</td>
</tr>
<tr>
<td>MOV NUM2, 1</td>
<td>Set second Fibonacci number</td>
</tr>
<tr>
<td>MOV FIBSER, 0</td>
<td>Store first Fibonacci number</td>
</tr>
<tr>
<td>MOV FIBSER+1, 1</td>
<td>Store second Fibonacci number</td>
</tr>
</tbody>
</table>
LEA DI, FIBSER+2
MOV CX, 8

Notice that the data segment address is defined as DATA_SEG. It is first moved into AX, and then DS is loaded from AX with another MOV operation. Next the memory locations assigned to NUM1 and NUM2 are loaded with immediate data $0000_{16}$ and $0001_{16}$, respectively. The same values are then copied into the storage locations for the first two series elements, FIBSER and FIBSER+1. Now DI is loaded with the address of FIBSER+2, which is a pointer to the storage location of the third element of the series. Finally, CX is loaded with 8.

To generate the next term in the series, we call a subroutine. This subroutine generates and stores the elements. Before returning to the main program, it also updates memory locations NUM1 and NUM2 with the values of the immediate past two elements. After this, the counter in CX is decremented to record that a series element has been generated and stored. This process must be repeated until the counter becomes equal to zero. This leads to

Figure L4.2 (a) Flowchart for generation of a Fibonacci series. (b) Program.
the following assembly language code:

```
NXTNM:   CALL   SBRTF
       DEC    CX
       JNZ    NXTNM
DONE:    NOP
```

The call is to the subroutine labeled SBRTF. After the subroutine runs to completion, program control returns to the DEC CX statement. This statement causes the count in CX to be decremented by one. Next, a conditional jump instruction tests the zero flag to determine if the result after decrementing CX is zero. If CX is not zero, control is returned to the CALL instruction at NXTNUM. If it is zero, the program is complete and execution halts.

The subroutine itself follows:

```
SBRTF:  PUSH   AX
       PUSH   BX
       MOV    AL, NUM1
       MOV    BL, NUM2
       ADD    AL, BL
       MOV    [DI], AL
       MOV    NUM1, BL
       MOV    NUM2, AL
       INC    DI
       POP    BX
       POP    AX
       RET
```

First, we save the contents of AX and BX on the stack. Then NUM1 and NUM2 are copied into AL and BL, respectively. They are then added together to form the next element. The resulting sum is produced in AL. Now the new element is stored in memory indirectly through DI. Remember that DI holds a pointer to the storage location of the next element of the series in memory. Then the second element, which is held in BL, becomes the new first element by copying it into NUM1. The sum, which is in AL, becomes the new second term by copying it into NUM2. Finally, DI is incremented by one so that it points to the next element of the series. The registers saved on the stack are restored and then we return to the main program.

Notice that both the subroutine call and its return have Near-proc operands. The entire program is presented in figure L4.2 (b).
Part 6: Running the Program

The source program that implements the Fibonacci series generation routine written as a procedure is in the file LAB04_2.ASM. This program was assembled and linked to produce a run module called LAB04_2.EXE. The source listing produced by the assembler is LAB04_2.LST. Now we will verify the operation of the program by generating the first ten numbers of the series by executing it on the PC. *Check off each step as it is completed.*

<table>
<thead>
<tr>
<th>Check</th>
<th>Step</th>
<th>Procedure</th>
</tr>
</thead>
</table>
|       | 1.   | Load the program LAB04_2.EXE with the command  
|       |      | C:\>DEBUG LAB04_2.EXE (J) |
|       | 2.   | Verify loading of the program by disassembling the contents of memory starting at the current code segment. |
|       | 3.   | Execute the program according to the instructions that follow:  
|       |      | a. GO from CS:0 to CS:2C.  
|       |      | b. DUMP the data memory from DS:0 to DS:C.  
|       |      | c. Verify that the series of numbers starting at location DS:2 and ending at DS:B satisfies the rules of Fibonacci series.  
|       |      | d. What is contained in locations DS:0 and DS:1? ________________  
|       |      | e. Execute the program to completion. |