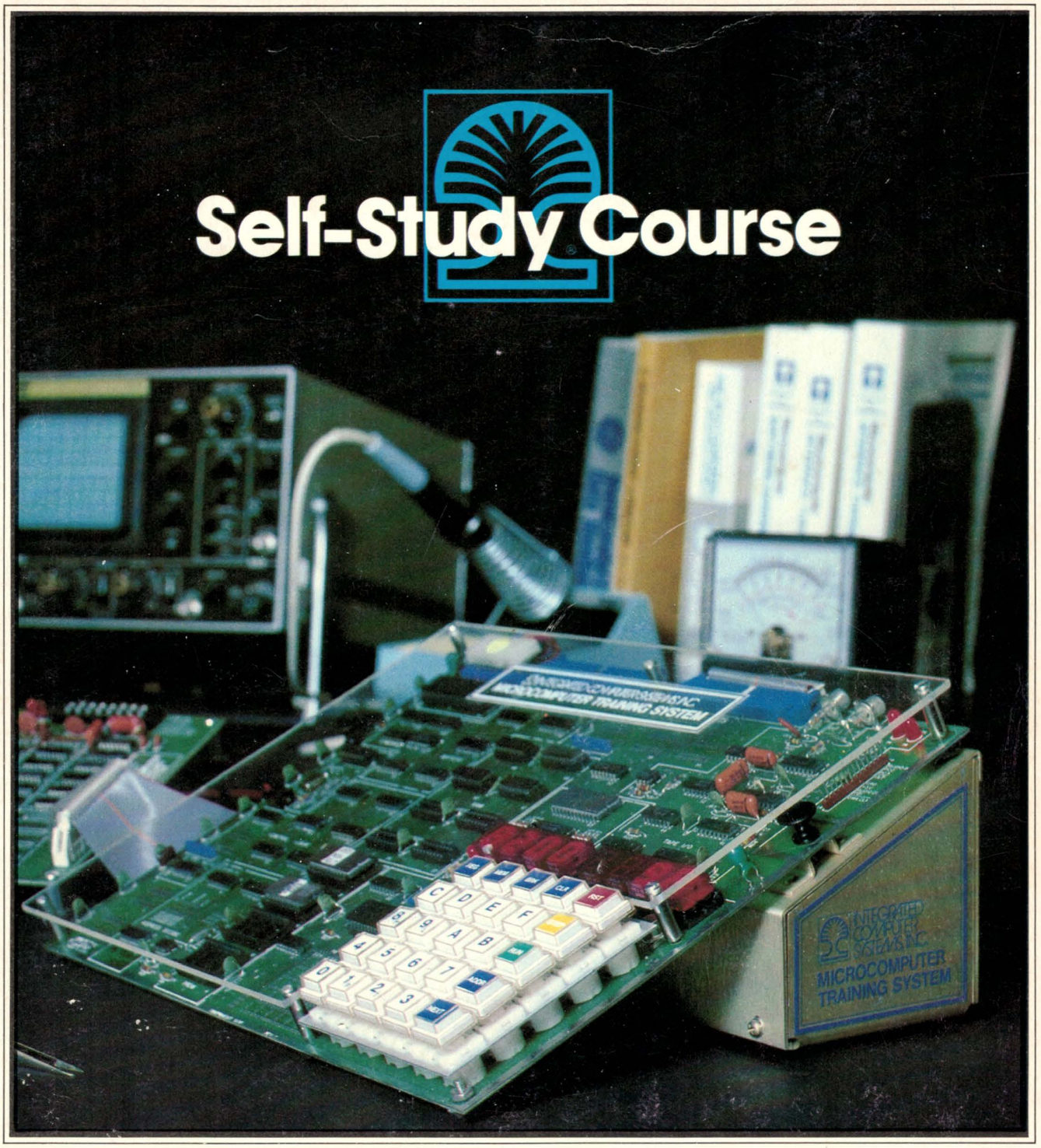
Self-Study Course



MICROPROCESSOR SOFTWARE & HARDWARE

Workbook/Text

Volume 2

Self-Study Course

Course 525A:

MICROPROCESSOR SOFTWARE & HARDWARE

Workbook/Text Volume II

DEVELOPED & PUBLISHED BY:
INTEGRATED COMPUTER SYSTEMS
Course Development Division

SENIOR AUTHOR:
Edward Dillingham, M.E., M.S.E.E.

ASSISTED BY:
Dr. Daniel M. Forsyth
Dr. Rudolf Hirschmann
Ms. Ruth H. Savoie
Dr. David C. Collins

EDUCATION IS OUR BUSINESS™

MICROPROCESSOR SOFTWARE & HARDWARE

Two Volumes
ISBN 0-89438-009-5

Volume I
ISBN 0-89438-010-9

Volume II
ISBN 0-89438-011-7

TABLE OF CONTENTS

VOLUME I

I INSTRUCTIONS - SYSTEM SETUP AND TEST PROCEDURE

I.1	RECEIVING INSPECTIONS	I-1
I.2	ASSEMBLY	I-1
I.3	POWER CONNECTION	I-2
I.4	INITIAL TEST	I-2
I.5	KEYBOARD TEST	I-3
I.6	PROGRAM LOADING TEST	I-4
I.7	SINGLE STEP TEST	I-5
I.8	PROM CHECKSUM TEST	I-6
I.9	READ-WRITE MEMORY TEST	I-7
I.10	SYSTEM EXPANSION	I-10

1 HARDWARE AND SOFTWARE FUNDAMENTALS

1.1	BASIC CONCEPTS	1-2
1.1.1	Definition of a Computer	1-2
1.1.2	Basic Hardware Structure of a Computer	1-2
1.1.3	Basic Software Concepts	1-6
1.1.4	The ICS Self-Study Microcomputer Training Course	1-9
1.2	NUMBER SYSTEMS AND REPRESENTATIONS	1-10
1.2.1	The Representation of Numbers	1-10
1.2.2	The Decimal Number System	1-12
1.2.3	The Binary Number System	1-14
1.2.4	Binary Addition and Counting	1-16
1.2.5	Hexadecimal Representation	1-19
1.3	THE ORGANIZATION OF MEMORY	1-22
1.3.1	Memory Words	1-22
1.3.2	Memory Module	1-24
1.3.3	Memory Access	1-26
1.3.4	Varieties of Memory	1-28
1.4	STRUCTURE OF THE CPU	1-31
1.4.1	Functional Units	1-31
1.4.2	The Execution of Instructions	1-33
1.4.3	Instruction Cycles	1-34
1.4.4	The Program Counter	1-35
1.4.5	The Instruction Register	1-37
1.4.6	The Accumulator	1-38
1.4.7	The Clock	1-38

TABLE OF CONTENTS

1.5	THE MTS MONITOR	1-41
1.5.1	Monitor Software	1-41
1.5.2	The MTS Keyboard and Display	1-43
1.5.3	Using the MTS	1-45
1.5.4	Inspecting Memory Contents	1-46
1.5.5	Changing Memory Contents	1-48
1.6	PREPARING A PROGRAM	1-50
1.6.1	Instructions to Be Used	1-51
1.6.2	Program Specification	1-53
1.6.3	Writing (Coding) the Program	1-53
1.6.4	Loading Your Program in the MTS	1-55
1.6.5	Verifying and Correcting the Stored Program	1-57
1.6.6	Executing Your Program	1-58
1.6.7	Instruction Execution: Detailed Examination	1-61
1.7	SUMMARY	1-65

2 TWO AND THREE BYTE INSTRUCTIONS

2.1	PROGRAM EXERCISE 2	2-1
2.1.1	The ADI Instruction	2-1
2.1.2	The STA Instruction	2-2
2.1.3	Instruction Execution Details	2-3
2.1.4	Writing the Program	2-10
2.1.5	Loading and Executing the Program	2-11
2.2	DATA STORAGE CONVENTIONS	2-15
2.3	PROGRAM EXERCISE 3	2-16
2.3.1	The LDA Instructions	2-16
2.3.2	The JMP Instruction	2-20
2.3.3	Writing the Program	2-23
2.4	SUMMARY OF INSTRUCTIONS	2-28
2.5	REVIEW OF COMMAND KEYS	2-29

3 PROGRAM LOOPS

3.1	PROGRAM LOOPS AND FLOW CHARTS	3-1
3.1.1	The Monitor RUN Command	3-1
3.1.2	The Conditional Jump	3-2
3.1.3	Flow Charts	3-7
3.2	PROGRAMMED MONITOR ENTRY	3-9
3.3	ADDITION BY COUNTING	3-13
3.4	EXERCISE	3-19
3.5	SUMMARY	3-20
3.6	SUMMARY OF INSTRUCTIONS	3-21

4	THE OTHER REGISTERS AND MEMORY ADDRESSING	4-1
4.1	THE MOV INSTRUCTION	4-2
4.2	THE ADD INSTRUCTIONS	4-4
4.3	THE CARRY AND ZERO FLAGS	4-6
4.3.1	Carry	4-7
4.3.2	Multiple Precision - The ADC Instruction	4-11
4.3.3	Exercise	4-16
4.3.4	Subtraction - SUB and SBB	4-18
4.3.5	Review and Self Test	4-23
4.4	IMMEDIATE INSTRUCTIONS	4-25
4.4.1	Move Immediate Instruction (MVI r)	4-25
4.4.2	Immediate Arithmetic Instructions	4-28
4.4.3	Multiplication by Repetitive Addition	4-30
4.4.4	Multiplication - Exercise	4-34
4.4.5	Table of Instructions	4-36
4.5	CONDITIONAL JUMPS	4-40
4.6	TRANSFER NOTATION	4-43
4.6.1	Instruction Definitions	4-44
4.6.2	Review and Self Test	4-48
4.7	THE MTS DISPLAY	4-53
4.7.1	Displaying a Bit Pattern	4-53
4.7.2	Display Digit Addresses	4-55
4.8	REGISTER PAIRS AND MEMORY ADDRESSING	4-57
4.8.1	The LDAX and STAX Instructions	4-59
4.8.2	Copy a List to Display - Exercise	4-63
4.8.3	Display of Eight Characters	4-67
4.8.4	Register Pair Loading - LXI	4-69
4.8.5	Register Pair Counting - INX, DCX	5-71
4.8.6	Delay Loops	4-73
4.8.7	Breakpoints	4-77
4.8.8	Review and Self Test	4-84
4.9	USE OF A MEMORY LOCATION AS A REGISTER	4-87
4.9.1	Memory Reference Instructions	4-88
4.9.2	Four Bye Addition Exercise	4-91
4.9.3	Counting in the Display - Exercise	4-95
4.10	INDIRECT ADDRESSING	4-96
4.10.1	Load and Store HL Direct	4-97
4.10.2	LHLD and SHLD - Example	4-99
4.10.3	Examining a Register Pair	4-103
4.10.4	Review and Self Test	4-106
4.11	COMPARISONS AND CONDITIONAL JUMPS	4-110
4.11.1	Comparison Instructions - CMP	4-111
4.11.2	Compare Immediate Instruction - CPI	4-112
4.11.3	Moving Message - Exercise	4-113
4.11.4	List of Intructions	4-118
4.12	SENSOR CORRECTION EXERCISE, VERSION 1	4-125
4.12.1	Sensor Characteristics	4-126
4.12.2	Organizing the Data Structure	4-130
4.12.3	Organizing the Program	4-131
4.12.4	Testing Sensor Correction	4-136
4.12.5	Review	4-139

TABLE OF CONTENTS

4.13	MULTIPLE TABLES WITH A DIRECTORY	4-140
4.13.1	Directory to Data Structures	4-141
4.13.2	Organizing the Program	4-142
4.13.3	Testing Sensor Numbers	4-145
4.13.4	Using the Directory	4-148
4.13.5	Testing Multiple Sensor Correction	4-153
4.14	SUMMARY	4-157
4.15	INSTRUCTION CHART	4-158

5 MEMORY AND CONTROL HARDWARE

5.1	SYSTEM CONTROLLER	5-3
5.1.1	Control Signals	5-3
5.1.2	Status Byte	5-5
5.1.3	Decoded Control Signals	5-6
5.1.4	MTS System Controller Logic	5-9
5.1.5	Intel 8228 System Controller	5-9
5.2	MEMORY TECHNOLOGY	5-11
5.3	CHIP SELECT LOGIC	5-17
5.3.1	Memory Enabling	5-19
5.3.2	RAM Chip Selection	5-19
5.3.3	ROM Chip Selection	5-20
5.3.4	Partial Decoding	5-23
5.3.5	Alternative Memory Addresssing	5-25
5.4	DATA BUS CONNECTIONS	5-26
5.4.1	Tri-State Circuits	5-26
5.4.2	Read-Write Control	5-27
5.4.3	DMA and Interrupts - Introduction	5-28
5.5	MEMORY SIGNALS AND TIMING	5-31
5.5.1	Machine States and Transitions	5-31
5.5.2	First State (T1)	5-31
5.5.3	Second State (T2) and Wait (TW)	5-32
5.5.4	States T3, T4 and T5	5-32

6 MODULES, SUBROUTINES AND THE STACK

6.1	PROGRAM MODULES	6-1
6.1.1	In-Line Programming	6-2
6.1.2	Creating Program Modules	6-3
6.1.3	Module Specification	6-6
6.2	SUBROUTINES	6-12
6.2.1	Subroutine Entry and Return	6-12
6.2.2	Tracing Subroutine Entry and Return	6-14
6.2.3	CALL Execution	6-16
6.2.4	Return Instructionn	6-20
6.2.5	Subroutine Nesting	6-24

6.3	SUBROUTINE SPECIFICATION	6-29
6.3.1	Program Development - Sensor Correction	6-29
6.3.2	Main Program	6-33
6.3.3	Input Subroutine	6-36
6.3.4	Conditional Calls	6-51
6.3.5	Subroutine DISPLAYRESULT	6-61
6.3.6	Subroutine SEARCHDIRECTORY	6-64
6.3.7	Program Data Initialization	6-67
6.3.8	Subroutine TABLELOOKUP	6-73
6.3.9	Stubs for Subroutines	6-75
6.3.10	Register Pair Addition	6-78
6.3.11	Program Integration	6-83
6.4	REVIEW AND SELF TEST	6-84
6.5	ADDITIONAL EXERCISES	6-88
6.5.1	Clear Result Display	6-97
6.5.2	Store and Recover Table Address	6-97
6.5.3	Two Byte Table Addresses	6-98
6.5.4	Empty Sensor Numbers	6-98
6.6	USING THE STACK FOR DATA	6-99
6.6.1	Testing Stack Usage	6-100
6.6.2	Using the Stack Inside a Subroutine	6-104
6.6.3	Processor Status Word (PSW)	6-105
6.6.4	Exchange Instructions	6-107
6.7	TEST DRIVER FOR MULTIPLY-EXERCISE	6-110
6.8	STACK POINTER INSTRUCTIONS AND RULES	6-116
6.8.1	Instructions that Affect Only the Stack Pointer	6-116
6.8.2	Stack Operation Rules	6-119
6.8.3	Monitor Usage of the Stack	6-120
6.8.4	The Growing Stack Problem	6-125
6.8.5	Review and Self Test	6-128
6.9	SUBROUTINE CLASSIFICATION	6-133
6.9.1	Global Subroutines	6-133
6.9.2	Local Subroutines	6-134
6.9.3	Re-Entrant Suroutines	6-134
6.9.4	Interrupt Service Routine	6-134
6.9.5	Subroutine Transparency	6-134
6.10	MONITOR SUBROUTINES	6-136
6.10.1	Monitor Keyboard Scan Subroutine (SCAN)	6-137
6.10.2	Monitor Key Entry Subroutine (GETKY)	6-138
6.10.3	Monitor Data Byte Input Subroutine (ENTBY)	6-140
6.10.4	Monitor Data Word Input Subroutine (ENTWD)	6-141
6.10.5	Monitor Display Digit Subroutine (DISPR)	6-142
6.10.6	Monitor Display Byte Subroutine - DMEM, DBYTE, DBY2	6-144
6.10.7	Monitor Display Word Subroutine - DWORD DWD2	6-146
6.10.8	Monitor Subroutine CLRGT, CLEAR, CLRLP	6-147
6.10.9	Monitor Subroutine DELAY, DELYA	6-148

TABLE OF CONTENTS

7	LOGIC AND BIT MANIPULATION	7-1
7.1	ROTATE COMMANDS	7-1
7.1.1	Rotate Exercise	7-3
7.1.2	Rotate Instructions for Control Functions	7-9
7.1.3	If-Then-Else Construct	7-11
7.1.4	Arithmetic Substitutes for RAL	7-17
7.1.5	Logical Rotate	7-18
7.2	BINARY ENTRY AND DISPLAY EXERCISE	7-22
7.3	LOGIC FUNCTIONS	7-29
7.3.1	Complement (CMA)	7-29
7.3.2	AND (ANA)	7-30
7.3.3	Inclusive OR (ORA)	7-31
7.3.4	Exclusive OR (XRA)	7-32
7.3.5	Immediate Logic Functions	7-33
7.3.6	Set and Complement Carry	7-34
7.4	LOGIC FUNCTIONS EXERCISE	7-35
7.4.1	Data Byte and Bit Marker	7-37
7.4.2	Keyboard Functions	7-39
7.4.3	Register Assignments	7-40
7.4.4	Subroutines for Logic Functions Exercise	7-40
7.4.5	Main Program for Logic Functions Exercise	7-43
7.4.6	Stubs for COMMAND and FUNCTION	7-45
7.4.7	Logic Functions DISPLAY Subroutine	7-49
7.4.8	Logic Functions DATA Subroutine	7-52
7.4.9	Additional Specifications for DATA	7-56
7.4.10	Logic Functions COMMAND Subroutine	7-60
7.4.11	Subroutine FUNCTION	7-65
7.4.12	Exercising Logic Functions	7-69
7.5	FLOW CONTROL TECHNIQUES	7-72
7.6	REVIEW AND ADDITIONAL EXERCISES	7-78
7.6.1	Traffic Control Exercise	7-79
7.6.2	Extended Traffic Control Exercises	7-85
7.6.3	Fire and Burglar Alarm	7-88
7.6.4	Model Railroad Simulator	7-88

TABLE OF CONTENTS

VOLUME II

8 INPUT/OUTPUT TECHNIQUES

8.1	ISOLATED INPUT/OUTPUT	8-2
8.1.1	I/O Ports	8-2
8.1.2	Programmable I/O Ports	8-9
8.1.3	Keyboard Input	8-15
8.1.4	Subroutine KYIN	8-16
8.1.5	Keyboard Display Exercise	8-26
8.1.6	Other I/O Interfaces	8-33
8.2	MEMORY MAPPED INPUT/OUTPUT	8-35
8.3	DIRECT MEMORY ACCESS	8-39
8.3.1	Repetitive Direct Memory Access	8-41
8.3.2	DMA Input and Output	8-45
8.4	I/O INITIATION	8-49
8.4.1	Programmed I/O	8-49
8.4.2	Interrupt Driven I/O	8-52
8.4.3	The MTS Interrupt System	8-66
8.5	INTERRUPT SERVICE ROUTINES	8-73
8.5.1	Preserving the Environment	8-73
8.5.2	Identifying the Source of the Interrupt	8-75
8.5.3	Vectored Interrupt Systems	8-75
8.5.4	Priority Interrupt Systems	8-76
8.5.5	Timed Interrupt Systems	8-76
8.6	USING INTERRUPTS WITH THE MTS	8-77
8.6.1	Interrupt Dispatch	8-77
8.6.2	Interrupt Service Routine Exercise	8-81
8.6.3	Interrupt Service Routine Test	8-83
8.6.4	Memory Change Breakpoints	8-88
8.6.5	Interrupt Service Operation	8-91
8.6.6	Combining Interrupt Service with monitor Functions	8-99
8.6.7	External Interrupt	8-100
8.6.8	Interrupt Handling -Summary	8-101

9 DATA FORMAT

9.1	PARALLEL INPUT/OUTPUT	9-3
9.1.1	Paper Tape Reader Example	9-3
9.1.2	Computer to Computer Interface	9-7
9.2	SERIAL INPUT/OUTPUT	9-14
9.2.1	Signal Coding	9-14
9.2.2	Synchronous Communication	9-16
9.2.3	Asynchronous Communication	9-17

TABLE OF CONTENTS

9.3	ASYNCHRONOUS TRANSMITTING AND RECEIVING	9-20
9.3.1	Serial Transmission Exercise	9-21
9.3.2	Character Data Pattern	9-23
9.3.3	Interrupt Service Routine	9-25
9.3.4	Main Program	9-27
9.4	ASYNCHRONOUS RECEIVING	9-33
9.4.1	Wait for Start Bit	9-35
9.4.2	Receive Data Bits	9-37
9.4.3	Receive Main Loop	9-39
9.5	MONITOR TAPE PROGRAMS AND SUBROUTINNES	9-44
9.5.1	Tape Recording Program	9-44
9.5.2	Tape Reading Program	9-45
9.5.3	Error Checking Character (LRC)	9-46
9.6	MONITOR SEND AND RECEIVE SUBROUTINES	9-47
9.6.1	SOTBT (0382)	9-47
9.6.2	Program Entry and Removal of Brekpoints	9-49
9.6.3	Subroutine BKMEM (01D3)	9-51
9.6.4	Subroutine SINWS (03CF)	9-52
9.6.5	Transmit/Receive with Monitor Subroutines	9-54
9.7	CALCULATING DELAY TIMES	9-61

10 BINARY AND DECIMAL ARITHMETIC

10.1	BINARY ADDITION	10-2
10.1.1	Multiple Precision	10-2
10.2	FOUR BYTE ADDITION	10-6
10.3	BINARY SUBTRACTION	10-13
10.4	DECIMAL ADDITION AND SUBTRACTION	10-25
10.5	BINARY MULTIPLICATION	10-33
10.6	DECIMAL MULTIPLICATION	10-39
10.7	OTHER REPRESENTATIONS OF NUMBERS	10-44
10.7.1	Negative Binary Numbers	10-45
10.7.2	Change Sign, Add, Subtract Exercise	10-53
10.7.3	Signed Decimal Numbers	10-59
10.7.4	Fractional Numbers	10-83

11 REVIEW

11.1	DATA TRANSFER	11-2
11.2	COUNTING INSTRUCTIONS	11-5
11.3	ACCUMULATOR/CARRY INSTRUCTIONS	11-7
11.4	ARITHMETIC AND LOGICAL INSTRUCTIONS	11-9
11.4.1	The Flags	11-10
11.5	BRANCH INSTRUCTIONS	11-13
11.6	INPUT/OUTPUT	11-15
11.7	UNDEFINED INSTRUCTIONS	11-16
11.8	OTHER MICROPROCESSORS	11-17
11.8.1	NEC 808A and NEC 8080AF	11-17
11.8.2	INTEL 8085	11-17
11.8.3	ZILOG Z-80	11-18

APPENDIX A THE ICS MONITOR

APPENDIX B BINARY/DECIMAL CONVERSIONS

APPENDIX C CALCULATING TRIGONOMETRIC FUNCTIONS

APPENDIX D THE S-100 ADAPTER CARD

APPENDIX E AMTS SCHEMATICS

APPENDIX F DIGITAL LOGIC

LIST OF ILLUSTRATIONS

VOLUME I

FIGURE	TITLE	PAGE
I-1	Read-Write Memory Test	I-8
1-1	MTS Board Layout	1-5
1-2	MTS Board Layout	1-30
1-3	MTS Board Layout	1-42
2-1	LDA Instruction Cycle	2-17
2-2	LDA Instruction Cycle (continued)	2-18
2-3	LDA Instruction Cycle (continued)	2-19
2-4	JMP Instruction Cycle	2-21
2-5	JMP Instruction Cycle (continued)	2-22
3-1	Conditional Jumps Flow Chart	3-10
3-2	Addition by Counting - Flow Chart	3-14
3-3	Addition by Counting - Program	3-15
4-1	Double Precision Addition	4-17
4-2	Double Precision Subtraction	4-22
4-3	MVI Instruction Cycle	4-27
4-4	Multiplication by Repetitive Addition	4-38
4-5	Bit Patterns for MTS Display	4-52
4-6	Instruction Cycle for STAX D Instruction	4-61
4-7	Hex Codes and Characters	4-62
4-8	Copy List to Display	4-66
4-9	Copy List to Display	4-72
4-10	Gradual Display with Clear	4-76
4-11	Four Byte Addition in Memory - Flow Chart	4-90
4-12	Four Byte Addition in Memory - Program	4-93
4-13	Counting in the Display	4-94
4-14	Moving Message - Flow Chart	4-116
4-15	Moving Message - Program	4-122
4-16	Sensor Calibration Curves	4-129
4-17	Sensor Correction	4-134
4-18	Multiple Sensor Correction - Flow Chart	4-144
4-19	Correcting Multiple Sensors - Program	4-150

5-1	Microcomputer Training System Configuration	5-2
5-2	MTS System Controller	5-8
5-3	Memory Addressing	5-12
5-4	Internal Address Decoding in a Memory Device	5-14
5-5	Chip Select Logic	5-18
5-6	MTS Memory Addresses	5-22
5-7	Minimum Chip Select	5-24
5-8	Memory Access Timing	5-30
6-1	Modular Sensor Correction - Flow Chart	6-5
6-2	Do Nothing Program with Do Nothing Module	6-9
6-3	Do Nothing Program	6-10
6-4	Call Instructions	6-17
6-5	Call Instructions (continued)	6-19
6-6	Return Instruction	6-21
6-7	Return Instruction (continued)	6-23
6-8	Nested Subroutines	6-25
6-9	Nested Do Nothing Subroutines	6-26
6-10	Sensor Correction with Subroutines	6-30
6-11	Sensor Correction - MAIN	6-34
6-12	Test GETKY and DBY2	6-40
6-13	Sensor Correction - INPUT (not complete)	6-49
6-14	Sensor Correction - INPUT (complete)	6-58
6-15	Sensor Correction - NEXTSENSOR	6-59
6-16	Sensor Correction - DIRECTORY AND DATA	6-60
6-17	Sensor Correction - DISPLAYRESULT	6-63
6-18	Sensor Correction - SEARCHDIRECTORY	6-66
6-19	Sensor Correction - MAIN and INITIALIZE	6-72
6-20	Sensor Correction - TABLELOOKUP	6-77
6-21	Sensor Correction - MULTIPLY	6-81
6-22	Complete Sensor Correction Program	6-89
6-23	Test Driver for MULTIPLY	6-111
6-24	Test Driver Program	6-112
7-1	Test Driver for SHIFT Subroutines	7-7
7-2	SHIFT Subroutines	7-8
7-3	Left and Right Shift Program	7-15
7-4	Sixteen Bit Logical Rotates	7-21
7-5	Binary Entry and Display Flow Diagram	7-24
7-6	Binary Entry and Display Program	7-27
7-7	Logic Functions - Main Program	7-46
7-8	Stubs for COMMAND and FUNCTION	7-47
7-9	Logic Functions DISPLAY Subroutine - Flow	7-48
7-10	Logic Functions - Subroutine DISPLAY	7-51
7-11	Logic Functions - Subroutine DATA	7-55
7-12	Logic Functions - Revised DATA	7-59
7-13	Logic Functions - Subroutine COMMAND	7-64
7-14	Logic Functions - Subroutine FUNCTION	7-66
7-15	Logic Functions - Self Test	7-71
7-16	Logic Functions with Dispatch Table	7-76
7-17	Traffic Control Program	7-83
7-18	Timer and Keyboard Scanner	7-87

LIST OF ILLUSTRATIONS

VOLUME II

FIGURE	TITLE	PAGE
8-1	From INTEL Manual	8-3
8-2	Array of Input/Output Ports	8-4
8-3	Isolated Input/Output with the 8255	8-8
8-4	8255 Mode 0 Combinations	8-10
8-5	MTS 8255 and Key Input Scanning Circuit	8-14
8-6	Subroutine KYIN	8-22
8-7	First test for KYIN	8-23
8-8	KPRG, KTST, KYIN with Debugging Features	8-24
8-9	KPRG, KTST, KYIN with Debugging Removed	8-25
8-10	Keyboard Display Program - Flow Chart	8-27
8-11	Keyboard Display Program	8-29
8-12	Keyboard Display Program	8-30
8-13	Typical I/O Interfaces	8-32
8-14	Memory Mapped Input/Output with the 8255	8-34
8-15	Memory Mapped Display	8-38
8-16	DMA Circuit	8-40
8-17	DMA timing	8-40
8-18	Display Circuit	8-42
8-19	Keyboard Testing in the Monitor	8-48
8-20	Programmed Input/Output	8-50
8-21	Coding and Effect of RST Instructions	8-56
8-22	Interrupt Processing	8-57
8-23	Interrupt Processing (continued)	8-58
8-24	Interrupt Processing (continued)	8-59
8-25	(From INTEL Manual)	8-60
8-26	Restart Port with 8212	8-62
8-27	Vectored Restart Port	8-63
8-28	Vectored Interrupt Using Resistors	8-64
8-29	MTS Interrupt Circuit and Timing	8-68
8-30	Interrupt Service Exercise - Main	8-80
8-31	Interrupt Service Routine	8-82
8-32	Test for Interrupt Service	8-84
8-33	Interrupt Service Exercise	8-93
9-1	8255 Mode 1 Input	9-2
9-2	High Speed Paper Tape Reader Interface	9-4
9-3	8255 Mode 2 - Bidirectional I/O	9-8
9-4	Interprocessor Communication Using 8255	9-10
9-5	Logic and Timing for Shared Memory	9-12
9-6	Serial Data Transmit Interrupt Service Routine	9-24
9-7	Serial Transmit - Main	9-26
9-8	Serial Transmit - Data Entry	9-29

9-9	Transmit - Receive Data Entry	9-32
9-10	Wait for Start Bit	9-34
9-11	Receive Data Bits	9-36
9-12	Receive Main Loop	9-38
9-13	Transmit - Receive	9-40
9-14	Transmit/Receive with Monitor Subroutines	9-53
9-15	Transmit Interrupt Service with SOTBT	9-55
9-16	Transmit Main Loop with Breakpoint Entry	9-56
9-17	Receive Main Loop with SINWS	9-58
9-18	Instruction Timing	9-60
10-1	Main Programs for Four Byte Add and Display	10-7
10-2	Multi-Byte Add Subroutine	10-8
10-3	Main Program for 4 Byte Add and Display	10-9
10-4	Multi-Byte Addition Subroutine	10-10
10-5	Modify Main to Display Halt	10-12
10-6	Multi-Byte Subtract Suroutine	10-17
10-7	Main Program for 4 Byte Subtract	10-18
10-8	Display Halt	10-19
10-9	Multi-Byte Subtraction Subroutine	10-20
10-10	Program Modify Module	10-22
10-11	Modify Subroutine by Key Input	10-23
10-12	Multi-Byte Add/Subtract Subroutine	10-24
10-13	Modify Subroutine by Key Input	10-26
10-14	Modify Subroutine by Key Input (continued)	10-27
10-15	For Experiment with DAA	10-32
10-16	Binary Multiplication	10-35
10-17	Binary Multiply - Two Byte Product	10-36
10-18	Decimal Multiply Subroutine	10-40
10-19	Data Entry and Display for Decimal Multiply	10-41
10-20	Change Sign of Number	10-47
10-21	Change Sign by CMA, INR A	10-50
10-22	Binary and Decimal Arithmetic	10-54
10-23	Change Sign, Add, Subtract Exercise	10-55
10-24	Change Sign Exercise - Data Entry and Command Interpretation	10-56
10-25	Command Execution	10-57
10-26	Change Sign Subroutine	10-58
10-27	Decimal Arithmetic	10-65
10-28	Two Byte Hundreds Complement	10-75
10-29	CHSIGN	10-78
10-30	SIGNMAG	10-82

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 8

INPUT/OUTPUT TECHNIQUES

8. INPUT/OUTPUT TECHNIQUES

Various techniques and peripheral devices may be used with the 8080 to provide input and output capabilities. This chapter describes the common methods of implementing I/O and provides exercises in the use of those that are readily carried out with the MTS.

The techniques differ from each other in three major respects: how the input or output device is addressed; what event initiates the transfer of information; and what form the data are in. (The latter will be treated in Chapter 9.)

Addressing

- Isolated Input/Output
- Memory Mapped Input/Output
- Direct Memory Access

Initiation

- Programmed Input/Output
- Interrupt Driven Input/Output
- Timed Input/Output
- Repetitive Direct Memory Access

The MTS includes facilities for all of these in one form or another, so you can learn each of the processes. For some, however, you must add external hardware.

INPUT/OUTPUT TECHNIQUES

8.1 ISOLATED INPUT/OUTPUT

The address and data buses are used to address input and output devices and transfer data between them and the CPU. The control bus from the system controller includes I/O Read and I/O Write commands in addition to the Memory Read and Memory Write commands. It is the use of these command signals, and the instructions that generate them, that distinguish I/O usage from memory usage of the buses.

8.1.1 I/O Ports

Any device with suitable electrical characteristics can be attached to the buses. In general such devices should have high impedance inputs from the bus and tri-state outputs to drive the bus. Intel and others provide the 8212 Input/Output Port for this purpose. The MTS includes one in the LED display circuit. A functional description is given in Figure 8-1; more detail is provided in the Intel 8080 Microcomputer System User's Manual. The principal features are low leakage currents of the inputs and outputs when the device is not selected, data latches, and control gating.

Functional Description

Data Latch

The 8 flip-flops that make up the data latch are of a "D" type design. The output (Q) of the flip-flop will follow the data input (D) while the clock input (C) is high. Latching will occur when the clock (C) returns low.

The data latch is cleared by an asynchronous reset input ($\overline{CLR}$). (Note: Clock (C) Overides Reset ($\overline{CLR}$).)

Output Buffer

The outputs of the data latch (Q) are connected to 3-state, non-inverting output buffers. These buffers have a common control line (EN); this control line either enables the buffer to transmit the data from the outputs of the data latch (Q) or disables the buffer, forcing the output into a high impedance state. (3-state)

This high-impedance state allows the designer to connect the 8212 directly onto the microprocessor bi-directional data bus.

Control Logic

The 8212 has control inputs $\overline{DS1}$, DS2, MD and STB. These inputs are used to control device selection, data latching, output buffer state and service request flip-flop.

$\overline{DS1}$, DS2 (Device Select)

These 2 inputs are used for device selection. When $\overline{DS1}$ is low and DS2 is high ($\overline{DS1} \cdot DS2$) the device is selected. In the selected state the output buffer is enabled and the service request flip-flop (SR) is asynchronously set.

MD (Mode)

This input is used to control the state of the output buffer and to determine the source of the clock input (C) to the data latch.

When MD is high (output mode) the output buffers are enabled and the source of clock (C) to the data latch is from the device selection logic ($\overline{DS1} \cdot DS2$).

When MD is low (input mode) the output buffer state is determined by the device selection logic ($\overline{DS1} \cdot DS2$) and the source of clock (C) to the data latch is the STB (Strobe) input.

STB (Strobe)

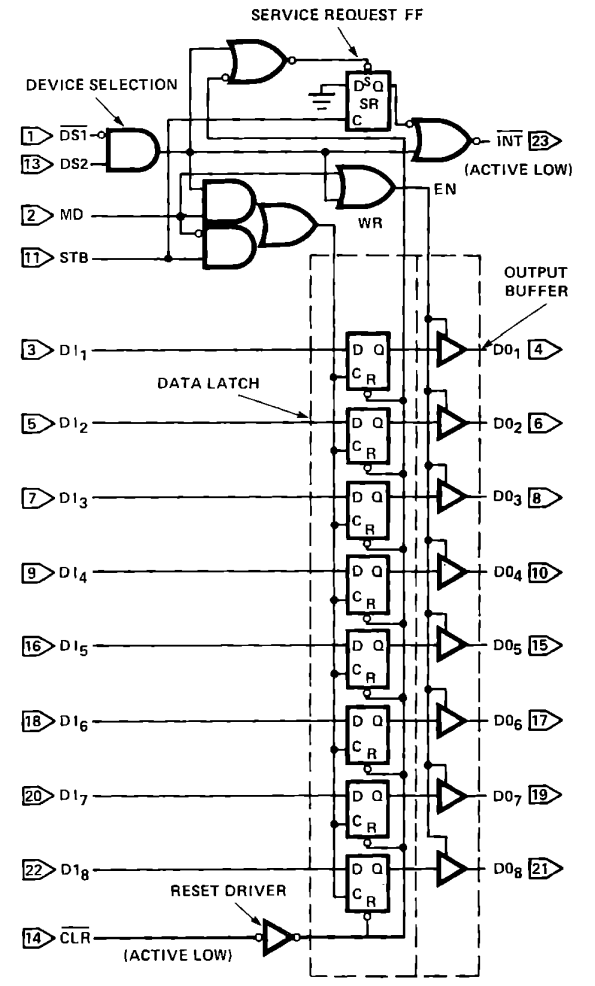
This input is used as the clock (C) to the data latch for the input mode MD = 0) and to synchronously reset the service request flip-flop (SR).

Note that the SR flip-flop is negative edge triggered.

Service Request Flip-Flop

The (SR) flip-flop is used to generate and control interrupts in microcomputer systems. It is asynchronously set by the $\overline{CLR}$ input (active low). When the (SR) flip-flop is set it is in the non-interrupting state.

The output of the (SR) flip-flop (Q) is connected to an inverting input of a "NOR" gate. The other input to the "NOR" gate is non-inverting and is connected to the device selection logic ($\overline{DS1} \cdot DS2$). The output of the "NOR" gate ($\overline{INT}$) is active low (interrupting state) for connection to active low input priority generating circuits.



STB	MD	$(\overline{DS_1} \cdot DS_2)$	DATA OUT EQUALS
0	0	0	3-STATE
1	0	0	3-STATE
0	1	0	DATA LATCH
1	1	0	DATA LATCH
0	0	1	DATA LATCH
1	0	1	DATA IN
0	1	1	DATA IN
1	1	1	DATA IN

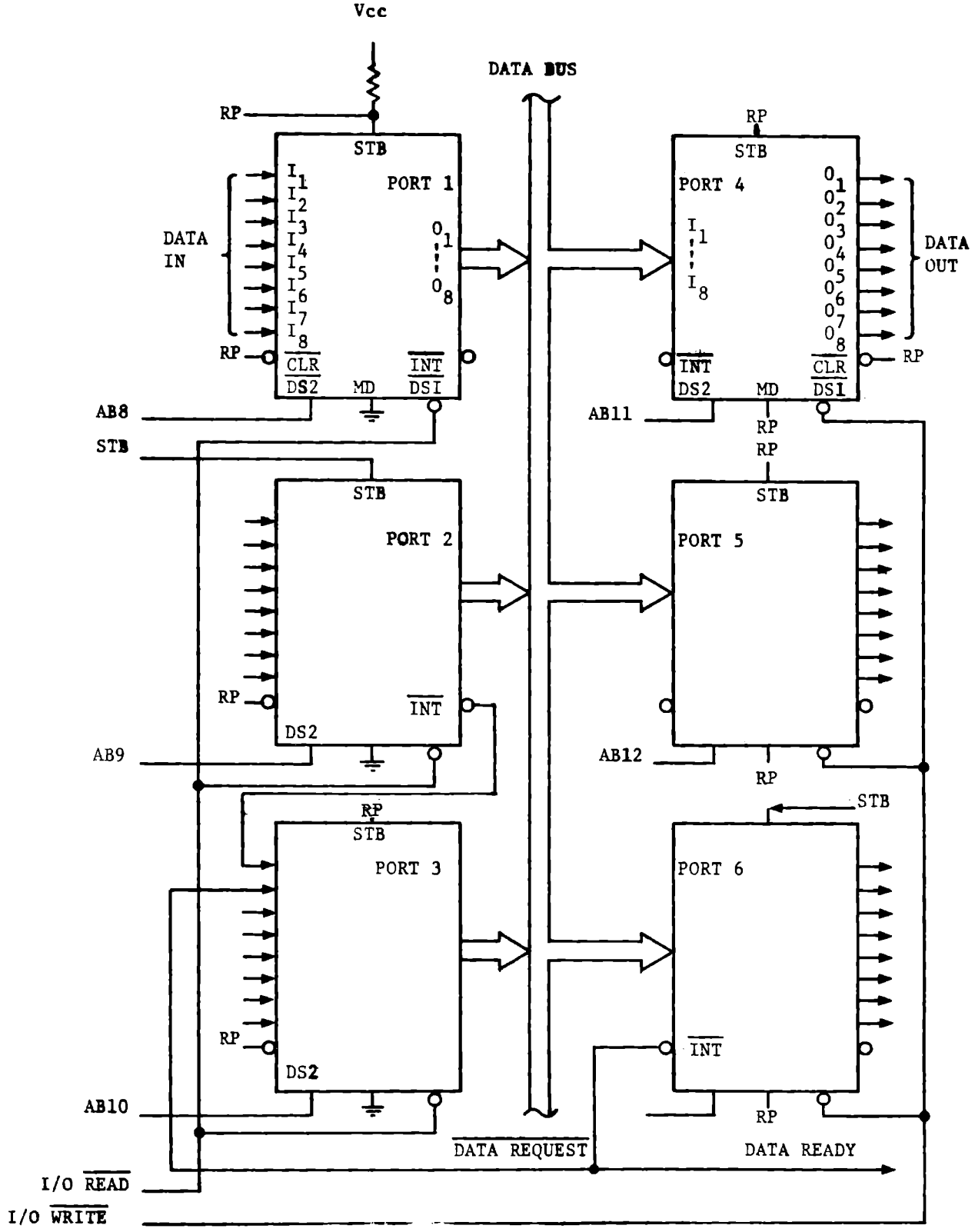
$\overline{CLR}$ – RESETS DATA LATCH
SETS SR FLIP-FLOP
(NO EFFECT ON OUTPUT BUFFER)

CLR	$(\overline{DS_1} \cdot DS_2)$	STB	*SR	INT
0	0	0	1	1
0	1	0	1	0
1	1	↓	0	0
1	1	0	1	0
1	0	0	1	1
1	1	↓	1	0

*INTERNAL SR FLIP-FLOP

Figure 8-1

From INTEL Manual



Array of Input/Output Ports

Figure 8-2

A suitable arrangement for using several 8212's as input and output ports is shown in Figure 8-2. Each is selected by a single bit of the high address bus to the non-inverting select input DS2, so no additional decoding is necessary. The input ports are enabled by the I/O $\overline{\text{READ}}$ command and the outputs by the I/O $\overline{\text{WRITE}}$ command, to the inverting select input $\overline{\text{DS1}}$. Output data from the CPU enters an output port when the device is selected by $\overline{\text{DS1}}$ and DS2, and latched by the 8212 when it is de-selected; the 8212 outputs are always enabled. This behavior is set by the MODE input being pulled high.

The STROBE input is unused for Output Ports 4 and 5. Output Port 6 receives a strobe from some external hardware to indicate a need for new data. With the MODE input high this has no effect on the data outputs, but it sets the INT output low, indicating a need for service. The diagram shows that signal being input to the processor through Input Port 3. When the CPU loads new data to Port 6 $\overline{\text{INT}}$ will be set high again to indicate that the requested data are ready.

Input Ports 1 and 3 are direct paths from their inputs onto the data bus when they are selected, because their strobe inputs are pulled high. This makes them suitable for stable data. Input Port 2 is designed to receive a fleeting input, which may be gone before the processor can service it. An external strobe is provided to latch the data in the 8212 and set $\overline{\text{INT}}$ low, requesting service from the CPU when it reads Port 3.

The CPU accesses these ports with the commands:

DB	IN	Input from port
xx	port address	to Register A
		High address <- (Byte 2)
		Low address <- (Byte 2)
		(A) <- (Data bus)
		No flags are affected
D3	OUT	Output to port
xx	port address	from Register A
		High Address <- (Byte 2)
		Low Address <- (Byte 2)
		(A) <- (Data Bus)
		No flags are affected.

These are the only instructions for isolated input and output. They alone create the I/O Read and I/O Write commands to the ports.

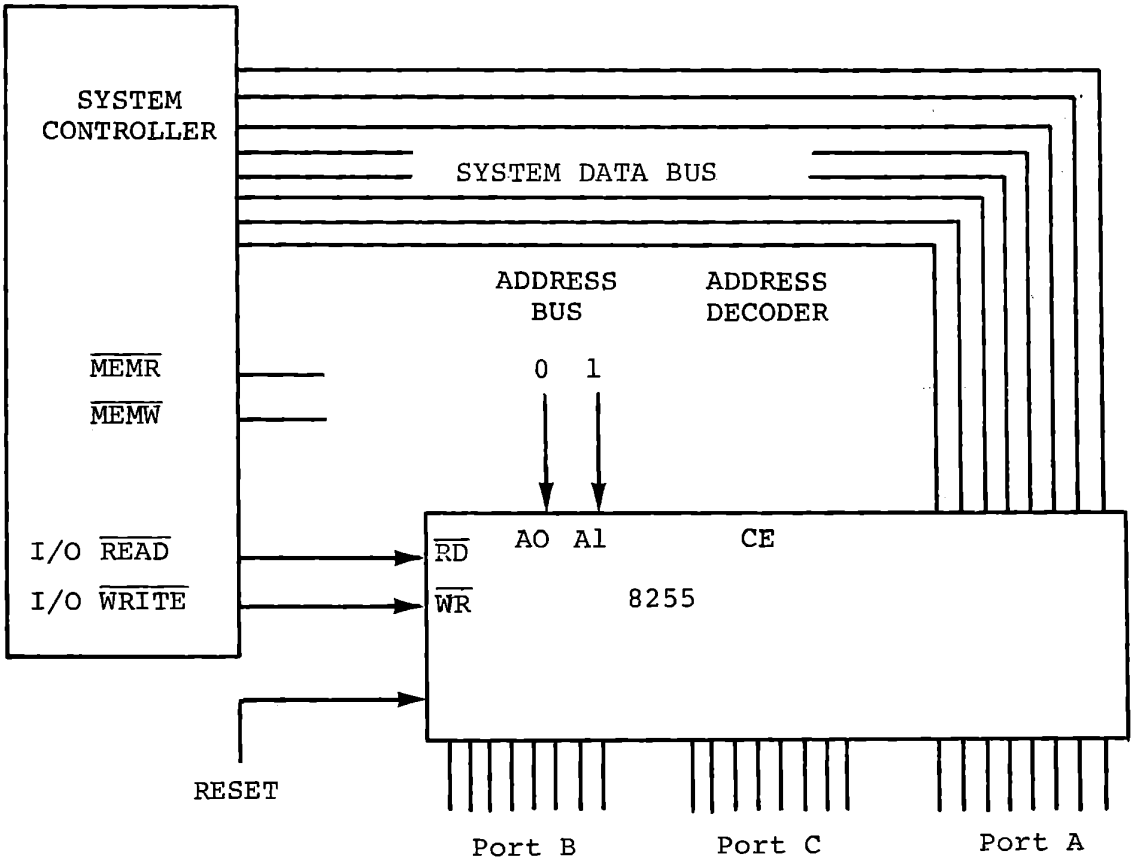
Note that the port address is only one byte, not two. In response to one of these instructions the CPU places that byte on the low eight bits of the address bus, and duplicates it on the high eight bits.

This duplication permits the I/O devices to be selected from the high address bus, which is typically less heavily loaded by memory devices than the low address bus.

The addressing shown here, where a single bit on the address bus selects a device, is called linear select. It is economical of hardware but restricts the system size. Port addresses for the devices in Figure 8-2 are:

Input Port 1	01	00000001
Input Port 2	02	00000010
Input Port 3	03	00000100
Output Port 4	08	00001000
Output Port 5	10	00010000
Output Port 6	20	00100000

For a larger system some decoding of the address is necessary.



Because the 8255 occupies four addresses, it receives and internally decodes the two low bits of the address bus. The I/O port address decoder examines only six bits of the address bus (AB2-AB7) to select the 8255.

Isolated Input/Output With the 8255

Figure 8-3

8.1.2 Programmable I/O Ports

The MTS includes one 8255 Programmable Peripheral Interface Adapter (Figure 8-3). It has 24 external connections which can be programmed as inputs or outputs in various combinations. It is connected to the microprocessor and system controller via the data bus, I/O Read, I/O Write, Reset, two address bits (AB0 and AB1) and a chip select input from the address decoder.

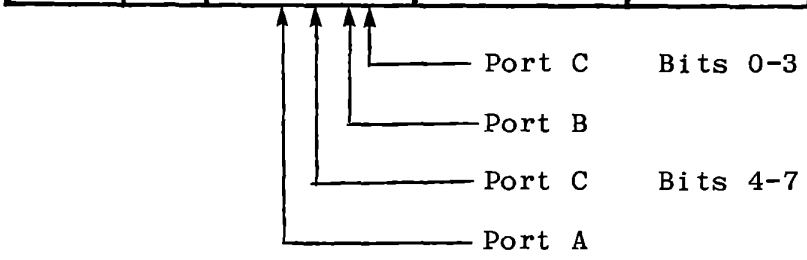
The 8255 accepts data from the data bus when its chip select input and I/O Write are both low. It delivers signals to the data bus when chip select and I/O Read are both low.

Four I/O port addresses are occupied by one 8255. Three of these correspond to the three groups of 8 bits provided by its 24 external connections. The fourth address is used to write control information to the 8255. For the 8255 on the MTS the addresses are:

00	Port A	(PORT0A)
01	Port B	(PORT0B)
02	Port C	(PORT0C)
03	Control	(CNT0)

The ICS Interface Training System contains two additional 8255's whose ports are referred to as PORT1A, PORT2C, etc. Therefore, all references to the MTS 8255 include 0 in the port name.

Notes	Control Byte		Port A	Port C	Port C	Port B
	Hex	Binary		Bits 4-7	Bits 0-3	
(3)	80	1000 0000	Out	Out	Out	Out
(3)	81	1000 0001	Out	Out	In	Out
(3)	82	1000 0010	Out	Out	Out	In
(3)	83	1000 0011	Out	Out	In	In
	88	1000 1000	Out	In	Out	Out
	89	1000 1001	Out	In	In	Out
	8A	1000 1010	Out	In	Out	In
	8B	1000 1011	Out	In	In	In
(1)	90	1001 0000	In	Out	Out	Out
(1)	91	1001 0001	In	Out	In	Out
(1,2)	92	1001 0010	In	Out	Out	In
(1)	93	1001 0011	In	Out	In	In
	98	1001 1000	In	In	Out	Out
	99	1001 1001	In	In	In	Out
	9A	1001 1010	In	In	Out	In
	9B	1001 1011	In	In	In	In



8255 Mode 0 Combinations
Figure 8-4

Notes: (1) Only the four combinations marked are suitable for use with the MTS if the keyboard is to be used. (2) This combination is set by the monitor whenever it controls the keyboard and display. (3) Port A and Port C (bits 4-7) should not both be programmed for output, since the keyboard would then short them together.

In addition to the three external ports, the 8255 has a "control port" addressed by 11 in the low bits of the address. This is used to program the external ports for input or output, and to select the mode of operation. The monitor programs the 8255 with the instructions:

```
3E   MVI  A,92      Write 10010010
92                  to the control port.
D3   OUT  CNTO
03
```

This sets Ports A and B for input and Port C for output. Ports A and B are each eight bit ports and can be programmed independently of each other. In the basic mode of operation (Mode 0) Port C is divided into two four-bit ports which can be independently programmed for input or output. Thus 16 different combinations of input and output assignments are available in Mode 0. The bits in the control byte are defined as follows:

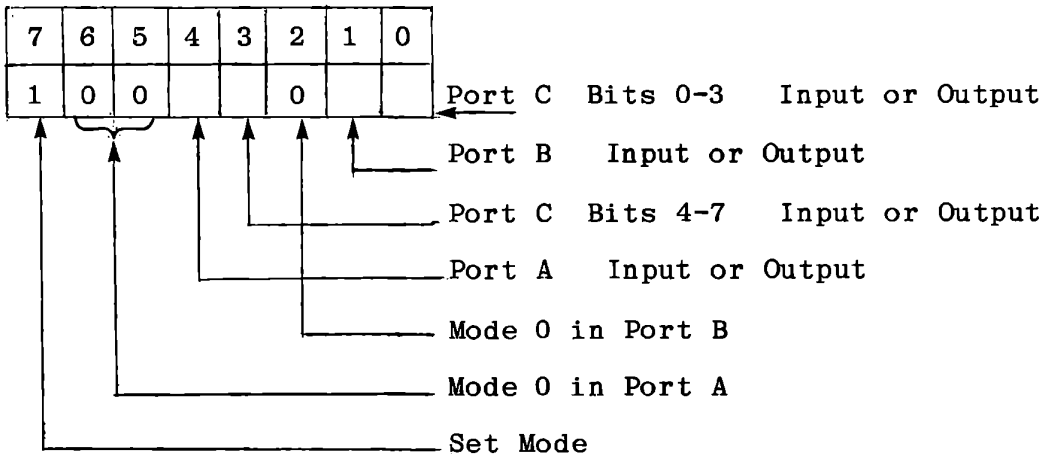


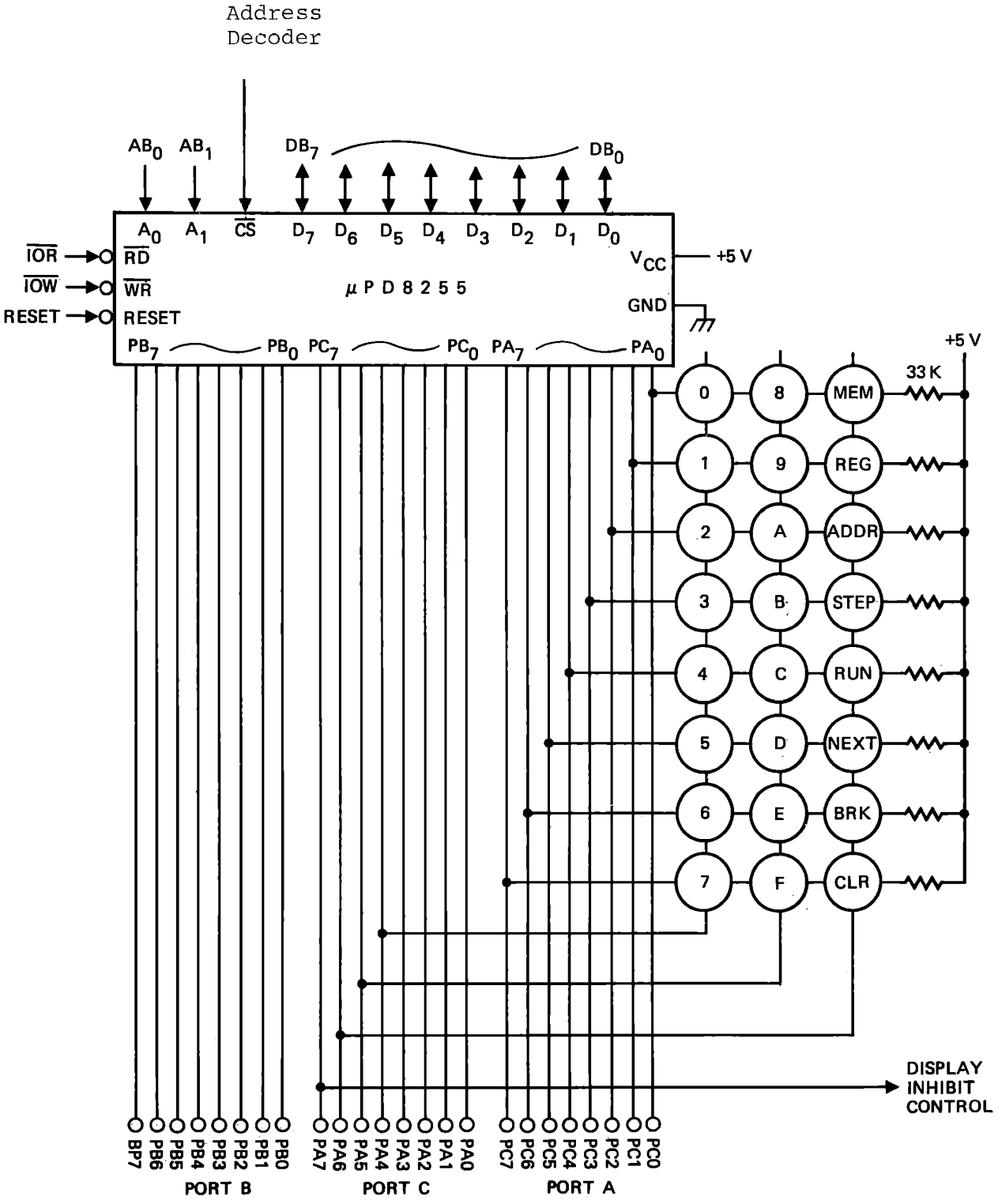
Figure 8-4 shows all 16 combinations.

This page intentionally left blank.

The 8255 provides a second mode of operation for Port A or Port B or both, in which certain bits of Port C are used for "handshaking" with external devices. For input in this mode the external device places its data at the input port and gives a strobe pulse to one bit of Port C. This stores the data in an eight bit latch associated with the eight bit input port, and generates other status bits in Port C which are accessible both to the CPU (by reading Port C) and to the external world at the Port C outputs. This allows transient signals to be input and read subsequently by the program at its convenience. For details the student is referred to the Intel 8080 Microcomputer System User's Manual.

In the basic input mode which we have been discussing, the data latches follow their inputs whenever the port is addressed. If a port is programmed for input the IN instruction will read the current state of the input. When a port is programmed for output, its data latch is cleared, setting all outputs low. Thereafter, the data latch is loaded by an OUT instruction, and the data remain stable until the next OUT. These data can be read back by the processor; IN will always read the content of the data latch. This does not apply to the control port, for which the IN instruction is not effective.

A third mode of operation is available for Port A only, in which it is both an input and an output port suitable for connection to a bi-directional data bus.



MTS 8255 and Key Input Scanning Circuit

Figure 8-5

8.1.3 Keyboard Input

To acquire familiarity with the 8255 we will develop a keyboard input program. You have been using the MTS monitor subroutines for this purpose. The subroutines to be developed here will be different in design.

Figure 8-5 shows the connections between the 8255 and the keyboard. The keyboard is a 3 x 8 matrix. Reset is not in the matrix but is directly connected to the reset input. The other keys form three columns: keys 0 through 7; 8 through F; and the command keys. Each row has three keys and a pullup resistor and is connected to an input bit of Port A. If no key in the row is pressed that bit of Port A will be 1 because of the resistor. If a key is pressed the input bit of Port A is connected through the key to one of three output bits of Port C. If that output is high the input to Port A will still be 1, but if it is low the input will be 0. Thus by setting one bit of Port C low and reading port A we can tell which, if any, key is pressed. We can make a quick test to see whether any key in the keyboard is pressed if we set all three outputs (C4, C5 and C6) low and read Port A; if the result is 1111 1111 no key is pressed.

There may be a circumstance where we are interested only in a particular key. This can be tested by setting the corresponding column low, reading the input, and masking to exclude all keys except the desired one. Subroutine KYIN is specified to permit any of these functions.

8.1.4 Subroutine KYIN

Function:

Test the keyboard for any desired key or keys being pressed. Set one or more of output bits C4, C5, C6 low (without affecting any other bits of Port C) according to a parameter passed in the call. Read the keyboard and mask with another byte passed as a parameter. Return with the Zero flag set if no desired key is pressed; otherwise with Zero cleared and the binary input data in Register C. Restore the column select bit (C4, C5, or C6) to 1 before returning.

Two alternate entries provide for setting the input parameters to test for any key, and for programming the 8255.

Call

```
CD    CALL KPRG
40    Program the 8255
82    and continue to KTST

CD    CALL KTST
44    Test for any key
82

CD    CALL KYIN
48    Test for specified key
82    or keys in specified
      column or columns
```

Inputs

```
KPRG: None
KTST: None
KYIN:
a)  Key column select in Register B
    contains 0 for each desired column.
    Bits 0, 1, 2, 3 and 7 must be 1
b)  Key mask in Register C
    contains 1 for each desired key
```

Outputs

Zero flag set if no desired key
Zero flag clear if desired key is pressed
Keyboard input (00 if no keys) in Register C
Key column select in Register B is preserved
(8F for KTST).

Registers

A, B, C, D are used.

Constraints

If KPRG is called, 8255 will be programmed as follows:

C0 - C3	Output
Port B	Output, mode 0
C4 - C7	Output
Port A	Input, mode 0

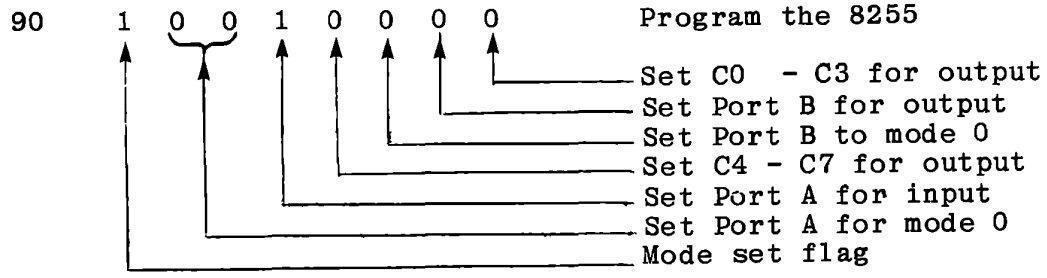
Outputs of all ports are cleared by KPRG.
If KTST or KYIN is called, C4 - C7 and Port A must be programmed as shown above.

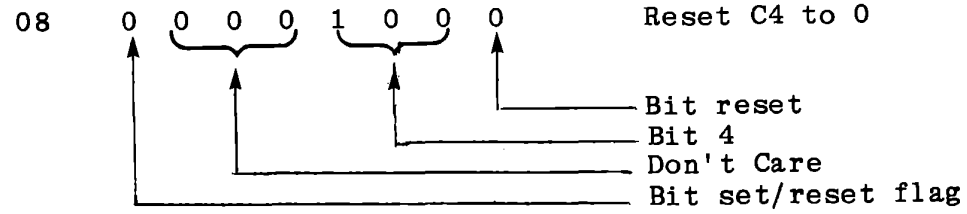
We have discussed programming the 8255 by writing to the control port. There is another function in the control port: you can set or reset any individual bit of Port C. This is done by writing a byte from Register A to the control port:

```
3E   MVI  A        (A) <- Selected command
xx
D3   OUT  CNTO
03
```

This sequence applies to both programming the 8255 and setting bits in Port C. The command bytes are distinguished by the high order bit as shown below:

Command Bytes to Control Port






```
0A   0 0 0 0 1 0 1 0    Reset C5 to 0
0C   0 0 0 0 1 1 0 0    Reset C6 to 0
```

This provides a technique for altering one output bit without changing others. Another technique is to read the content of the output data latch:

```
DB                              IN PORTC
02
```

will read the data latch of the port into register A even though the port is programmed for output. Then you can use "ORA r" or "ORI data" to set desired bits to 1; "ANA r" or "ANI data" to set desired bits to 0. For instance, to set C7, C6 and C5 to 1 and C4 to zero, use this program segment:

```
06   MVI  B,11101111     Set up for C4 low
EF
DB   IN   PORTC          Read old output data
02
F6   ORI  11110000       Set C7, C6, C5, C4 to 1
F0
A0   ANA  B              Set selected bit to 0
D3   OUT  PORTC          Write to Port C
02
```

Wherever several bits must be controlled this takes less program space than the individual bit set and reset instructions. Caution: Reading from an output port is not included in the manufacturer's specification for the 8255. That it will work is predictable from the design of the 8212, and proven by experiment with the 8255. A redesigned 8255 might not allow it.

Programs that write to the display or to Port C, or that program the 8255, are always difficult to debug because whenever the monitor actuates the keyboard and display it destroys whatever your program has done. Suggestion: at each point in the program when an output is written, first store the data in memory. When you read an input, immediately store the data. Being able to recover the data at a subsequent breakpoint makes debugging immensely easier. The additional instructions can be deleted when the program works properly.

Keyboard reading introduces another problem: at return from the monitor the keys are always released. You can simulate a key input by placing a breakpoint just after the IN instruction. When it is executed you can load some value other than FF in the A register to make sure that the rest of your program functions correctly.

If any peculiar condition arises while you have a key pressed, you can press RST while the other key is held down. The program counter will be saved. Press ADDR T MEM to see the program counter. This is the last value observed by the monitor (your program must have been running in Breakpoint mode). If your program was executing a subroutine when you pressed RESET, the return address can be found at (83DE, 83DF), provided no breakpoints had been entered.

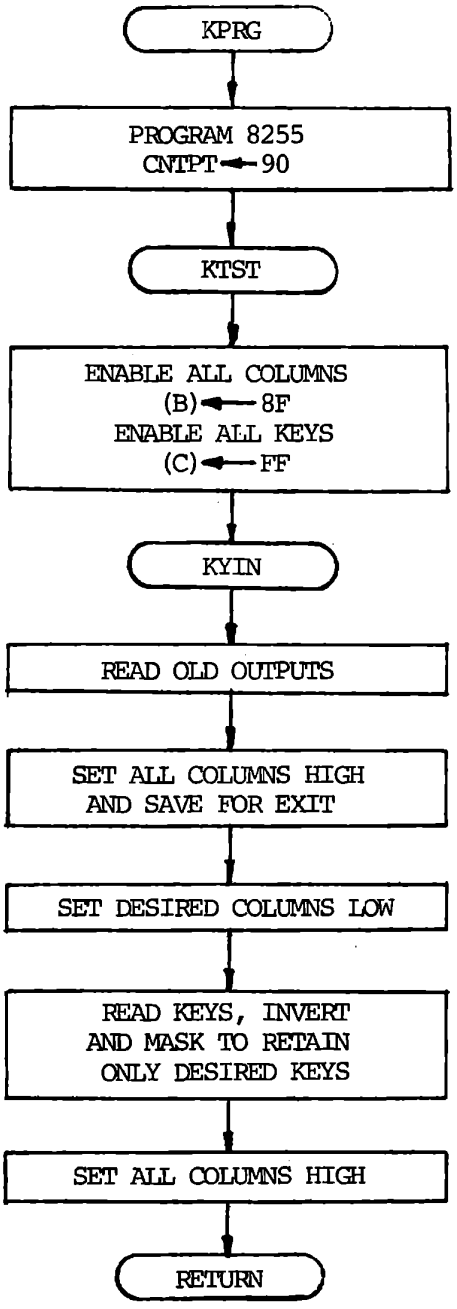
Draw the flow chart and write the program for KYIN. Test it initially with a very simple calling program. To ease debugging, call KYIN, not KTST. The monitor leaves the 8255 programmed with Port C for output and Port A for mode 0 input.

```
LXI     B,8FFF    Enable all keys
CALL    KYIN      Read keys
JZ                Repeat until
RST 4             key is found
JMP               Then call monitor
```

This will return to the monitor as soon as you press a key. Then you can look in the storage locations where you have saved the inputs and outputs to see if they are what you expect.

When you call the monitor with a key pressed, hold the key down until you see what you have. If you are displaying PC and the instruction, a numeric key will give the Err display as soon as you release it. If you are displaying a register, a numeric key will be entered into the register when you release it. You can retrieve the old value by pressing CLR, however.

Figures 8-6 to 8-9 provide a flow chart, test program, and two versions of KYIN, one with debugging code included.



Subroutine KYIN

Figure 8-6

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

FIRST TEST FOR KYIN

ADDR		CODE					
8 20	0	0	1		LXI	B, 8FFF	
	1	F	F				
	2	8	F				
	3	C	D		CALL	KYIN	
	4	4	8				
	5	8	2				
	6	C	A		JZ	8200	
	7	0	0				
	8	8	2				
	9	E	7		RST	4	
	A	C	3		JMP	8200	
	B	0	0				
	C	8	2				
	D						
	E						
	F						
8	0						
	1						
	2						
	3						
	4						
	5						
	6						
	7						
	8						
	9						
	A						
	B						
	C						
	D						
	E						
	F						
8	0						
	1						
	2						
	3						
	4						
	5						
	6						
	7						
	8						Figure 8-7

KTST, KYIN WITH DEBUGGING FEATURES

ADDR		CODE														
8 24	0	3	E		M	V	I		A	,	9	0			KPRG
	1	9	0												Program 8255
	2	D	3		O	U	T		C	N	T	O			Ports B,C Output
	3	0	3												Port A Input
	4	0	6		M	V	I		B	,	8	F			KTST
	5	8	F												(B) ← all columns
	6	0	E		M	V	I		C	,	F	F			(C) ← all keys
	7	F	F												
	8	D	B		I	N		P	O	R	T	O	C		KYIN
	9	0	2												Read old outputs
	A	3	2		S	T	A		8	3	0	0			Save for debugging
	B	0	0												
	C	8	3												
	D	F	6		O	R	I		F	O					Set all columns
	E	F	O												high and save
	F	5	7		M	O	V		D	,	A				in register D
8 25	0	A	0		A	N	A		B						Set selected
	1	3	2		S	T	A		8	3	0	1			columns low
	2	0	1												and save for
	3	8	3												debugging
	4	D	3		O	U	T		P	O	R	T	O	C	
	5	0	2												
	6	D	B		I	N			P	O	R	T	O	A	Read keys
	7	0	0												
	8	3	2		S	T	A		8	3	0	2			Save for debugging
	9	0	2												
	A	8	3												
	B	2	F		C	M	A								Invert and mask
	C	A	1		A	N	A		C						so desired key
	D	4	F		M	O	V		C	,	A				=1 if pressed
	E	7	A		M	O	V		A	,	D				Set all columns
	F	D	3		O	U	T		P	O	R	T	O	C	high
8 26	0	0	2												
	1	C	9		R	E	T								
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 8-8

KRPG, KTST, KYIN WITH DEBUGGING REMOVED

ADDR	CODE												
8 24 0	3E		MVI	A,	90							KPRG
1	90											Program 8255
2	D3		OUT	CNTO								Ports B,C out
3	03											Port A in
4	06		MVI	B,	8F							KTST
5	8F											(B) ← all columns
6	0E		MVI	C,	FF							(C) ← all keys
7	FF											
8	DB		IN	PORTOC								KYIN
9	02											Read old outputs
A	F6		ORI	F0								Set all columns
B	F0											high and save
C	57		MOV	D,	A							in reg D
D	A0		ANA	B								Set selected
E	D3		OUT	PORTOC								columns low
F	02											
8 25 0	DB		IN	PORTOA								Read keys and
1	00											invert so key
2	2F		CMA									pressed = 1
3	A1		ANA	C								Mask unwanted
4	4F		MOV	C,	A							keys and store
5	7A		MOV	A,	D							Set all columns
6	D3		OUT	PORTOC								high
7	02											
8	C9		RET									
9												
A												
B												
C												
D												
E												
F												
8 0												
1												
2												
3												
4												
5												
6												
7												
8												

Figure 8-9

8.1.5 Keyboard Display Exercise

Now we can make more interesting use of KYIN. The following program takes any key from 0 - 7 (which appears as a single bit = 1 in register C) and OR's it into a display location at the corresponding display segment bit. By pressing successive keys, you may "paint" a character. It also tests for CLR and NXT, either clearing the presently addressed display location or moving to the next location. This demonstrates one requirement of keyboard input: you must distinguish between a key being held down for a long time versus repetitive depressions of the same key. The numeric keys and CLR don't care in this program, but if you do not test for release of NXT it will step across the display many times before you can let go of the key.

Keyboard input programs normally provide for "debouncing". Many electrical switches do not change from closed to open perfectly, but "bounce" between the two states for some milliseconds. This can occur in the switch contact itself, or it can be created by a TTL circuit sensing the contact. To avoid seeing a single closure as multiple operations there is usually a time delay circuit or program used to require that the key be open for 10 to 30 milliseconds before it is accepted again. Such a provision is included in the MTS monitor subroutine GETKY, even though the MTS keys seem to be completely free of bounce. Before referring to Figures 8-10 through 8-12, try designing the program yourself, all the way from a specification and flow charts through the detailed coding.

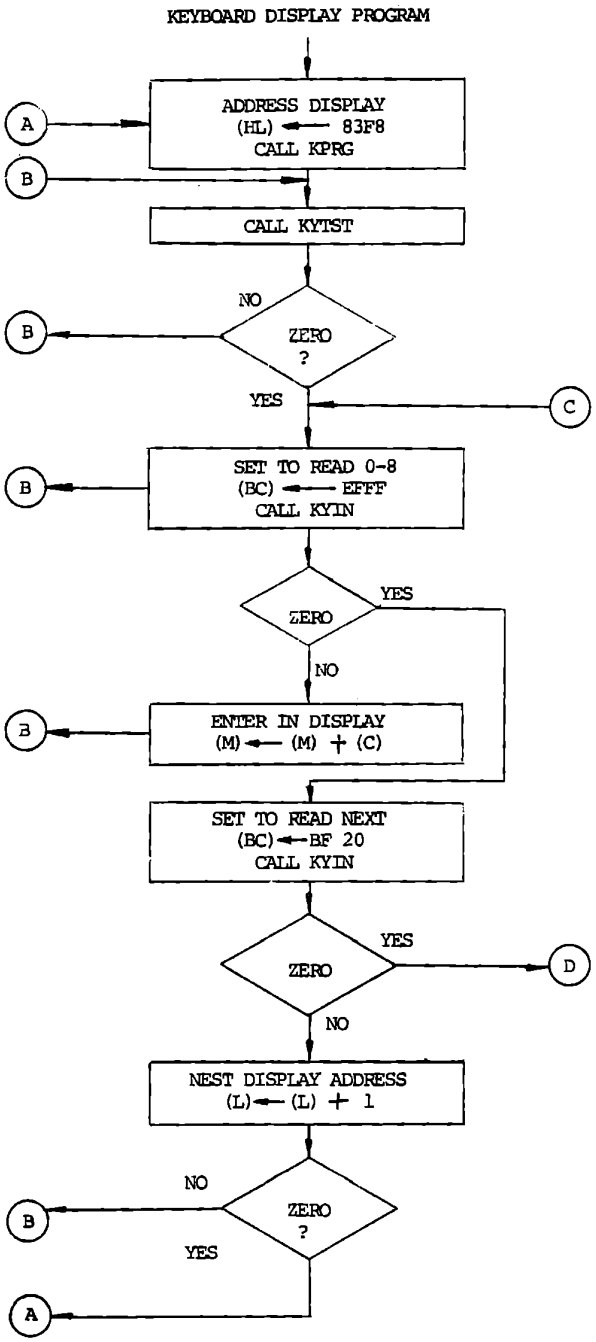

KEYBOARD DISPLAY PROGRAM
A
B
ADDRESS DISPLAY
(HL) ← 83F8
CALL KPRG
CALL KYTST
NO
B
ZERO
?
YES
C
B
SET TO READ 0-8
(BC) ← EFFF
CALL KYIN
ZERO
YES
NO
B
ENTER IN DISPLAY
(M) ← (M) + (C)
SET TO READ NEXT
(BC) ← BF 20
CALL KYIN
ZERO
YES
D
NO
NEST DISPLAY ADDRESS
(L) ← (L) + 1
NO
B
ZERO
?
YES
A

Figure 8-10a

KEYBOARD DISPLAY PROGRAM (CONT'D)

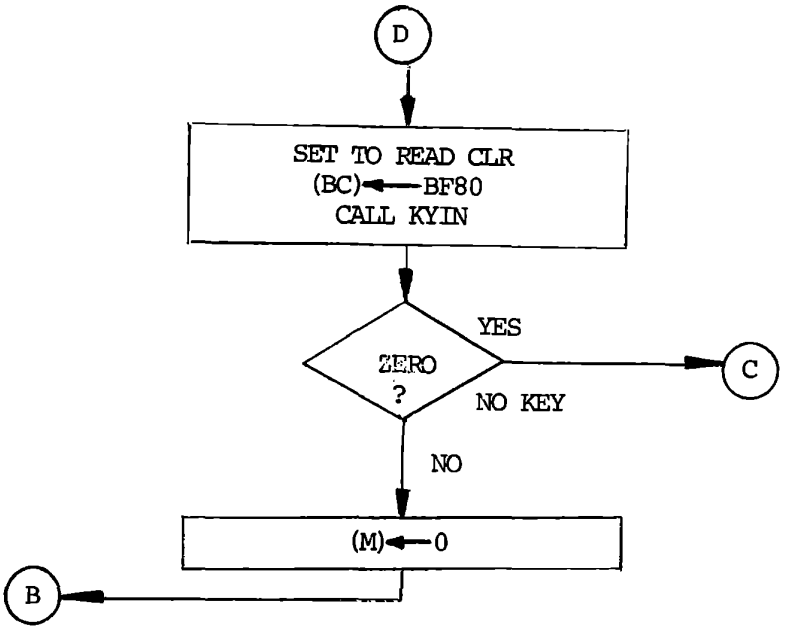


Figure 8-10b

KEYBOARD DISPLAY PROGRAM

ADDR		CODE														
8 20	0	00		NOP												
	1	00		NOP												
	2	00		NOP												
	3	21		LXI		H,	83	F8								Address display
	4	F8														
	5	83														
	6	CD		CALL		KPRG										Program 8255
	7	40														only once
	8	82														
	9	CD		CALL		KTST										Test for any key
	A	44														
	B	82														
	C	C2		JNZ		8209										Wait for keys
	D	09														to be released
	E	82														
	F	00		NOP												For breakpoint
8 21	0	01		LXI		B,	EF	FF								Set to read
	1	FF														keys 0-7
	2	EF														
	3	CD		CALL		KYIN										
	4	48														
	5	82														
	6	CA		JZ		8220										Jump if no key
	7	20														from 0-7
	8	82														
	9	7E		MOV		A,	M									OR key into
	A	B1		ORA		C										display
	B	77		MOV		M,	A									
	C	C3		JMP		8209										To wait for
	D	09														release
	E	82														
	F	00		NOP												
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 8-11

KEYBOARD DISPLAY PROGRAM

ADDR	CODE	Mnemonic	Operand	Comment
8220	01	LXI	B, BF20	Set to read NEXT
1	20			
2	BF			
3	CD	CALL	KYIN	
4	48			
5	82			
6	CA	JZ	8230	Jump if not NEXT
7	30			
8	82			
9	2C	INR	L	Next display address
A	C2	JNZ	8209	Go wait for release
B	09			
C	82			
D	C3	JMP	8200	At end of display go to start
E	00			
F	82			
8230	01	LXI	B, BF80	Set to read CLR
1	80			
2	BF			
3	CD	CALL	KYIN	
4	48			
5	82			
6	CA	JZ	8210	Jump if not CLR to read more keys
7	10			
8	82			
9	36	MVI	M, 00	Clear display digit
A	00			
B	00	NOP		
C	C3	JMP	8209	Go wait for release
D	09			
E	82			
F	00	NOP		
8 0				
1				
2				
3				
4				
5				
6				
7				
8				

Figure 8-12

This page intentionally left blank.

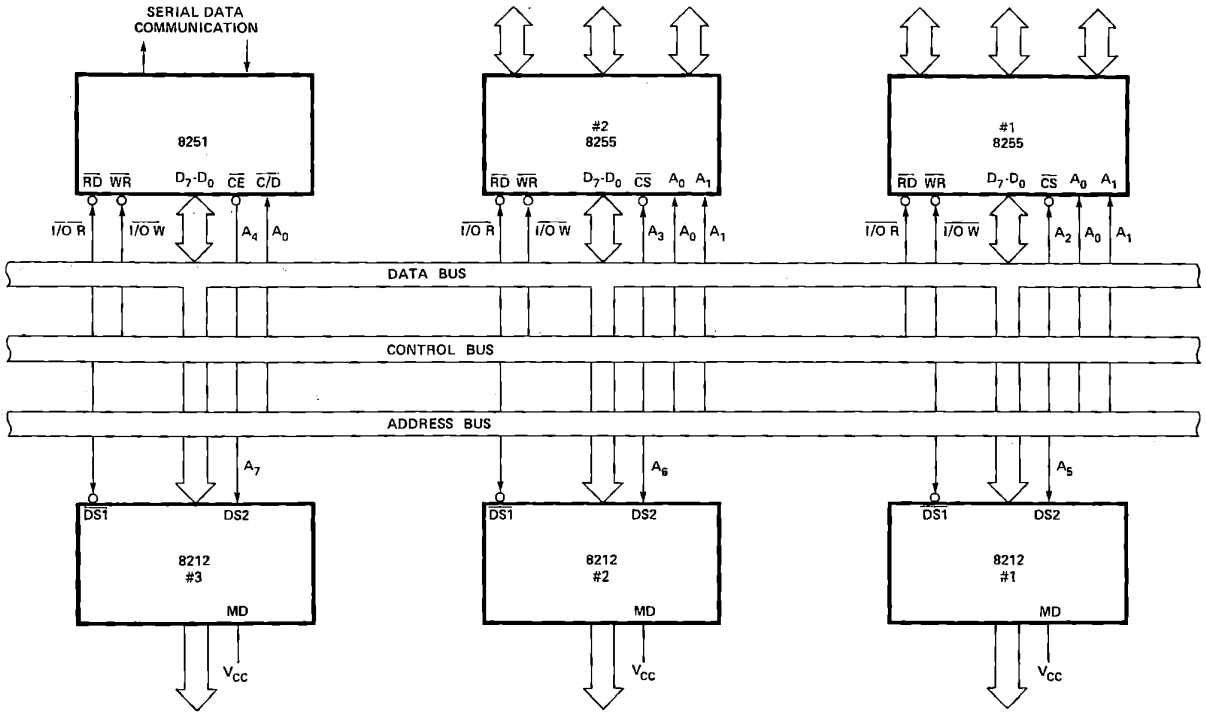


Figure 3-16. Typical I/O Interface.

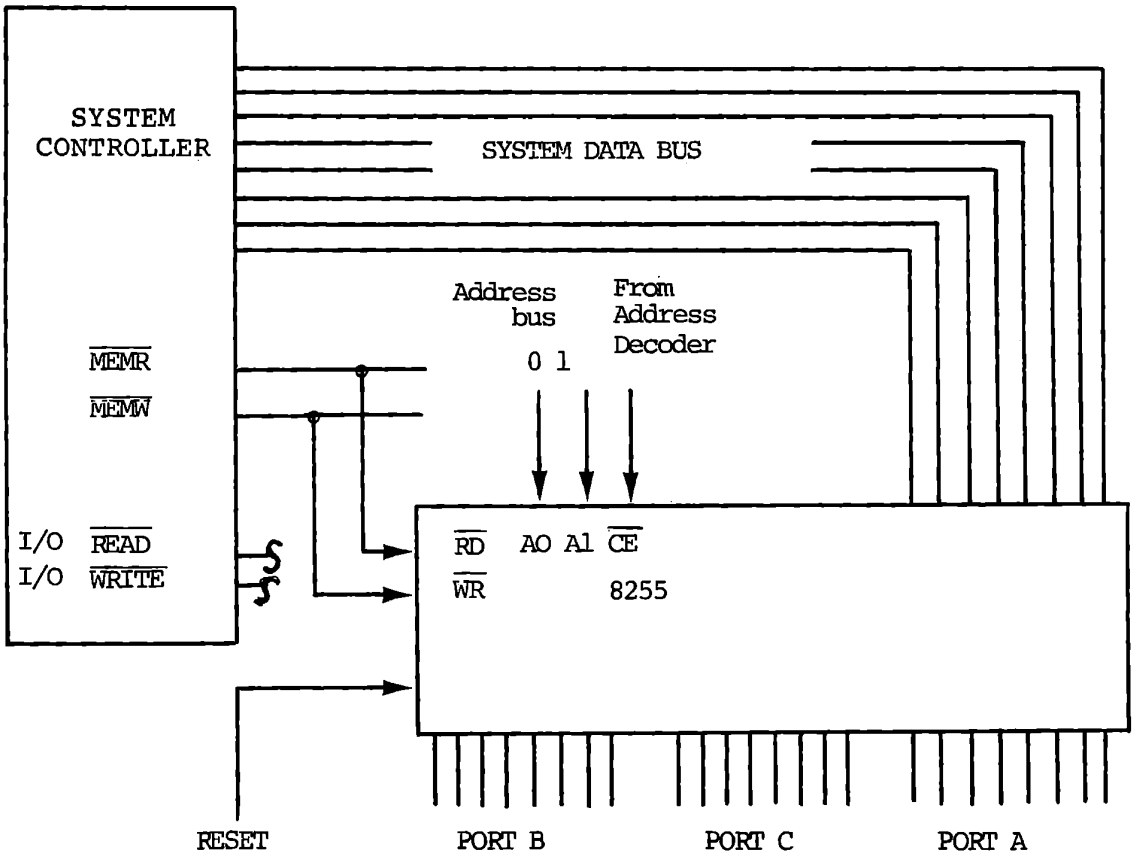
(From INTEL 8080 User's Manual)

Figure 8-13

8.1.6 Other I/O Interfaces

Isolated input/output is by no means restricted to the 8255; it is defined by the use of the IN and OUT instructions and the I/O Read and I/O Write commands. The necessary interface to the data bus, address bus and the command signals can be built with TTL and Tri-State circuits. Also, Intel and others offer several other devices made for this interface.

Many computer terminals use the 8251 Programmable Communication Interface for serial data communications. This has an interface to the 8080 system quite similar to that of the 8255, except that it needs the system clock. The student is again referred to the Intel 8080 User's Manual for detailed descriptions of these devices. Figure 8-13 shows how a number of devices can be connected to the system buses.



Memory Mapped Input/Output With the 8255

Figure 8-14

8.2 MEMORY MAPPED INPUT/OUTPUT

An alternative to isolated input/output is "memory mapped I/O". The input or output device is connected to the Memory Read and/or Memory Write command signals from the system controller, instead of the I/O Read and I/O write commands. Figure 8-14 shows such a connection. Here the IN and OUT instructions are not used, since the device is not connected to the command signals they generate. Instead any memory read or write command can be used. LDA may be used in place of IN, STA in place of OUT. All the convenience of register addressing and transfer becomes available. Suppose that Port A is addressed as memory location FFF8 and Port B as memory location FFF9, and both are programmed for inputs. Then they can be read by:

```
LXI     H,FFF8    Address Port A
MOV     E,M       (E) <- (Port A)
INX     H         Address Port B
MOV     D,M       (D) <- (Port B)
```

Alternately, they can both be ready by a single instruction:

```
LHLD    FFF8
```

The arithmetic and logic instructions become available for direct use with the input port. If you want to wait for a change in the input data you could use this:

```
      LXI       H,FFF8    Address Port A
      MOV       A,M       (A) <- (Port A)
  ┌-> CMP       M         (A) = (Port A)?
  └-- JZ                  Wait until equal
```

Or you can test for an input of 1111 1111:

```
      LXI       H,FFF8
      INR       M
  ┌-- JZ
  ↓
```

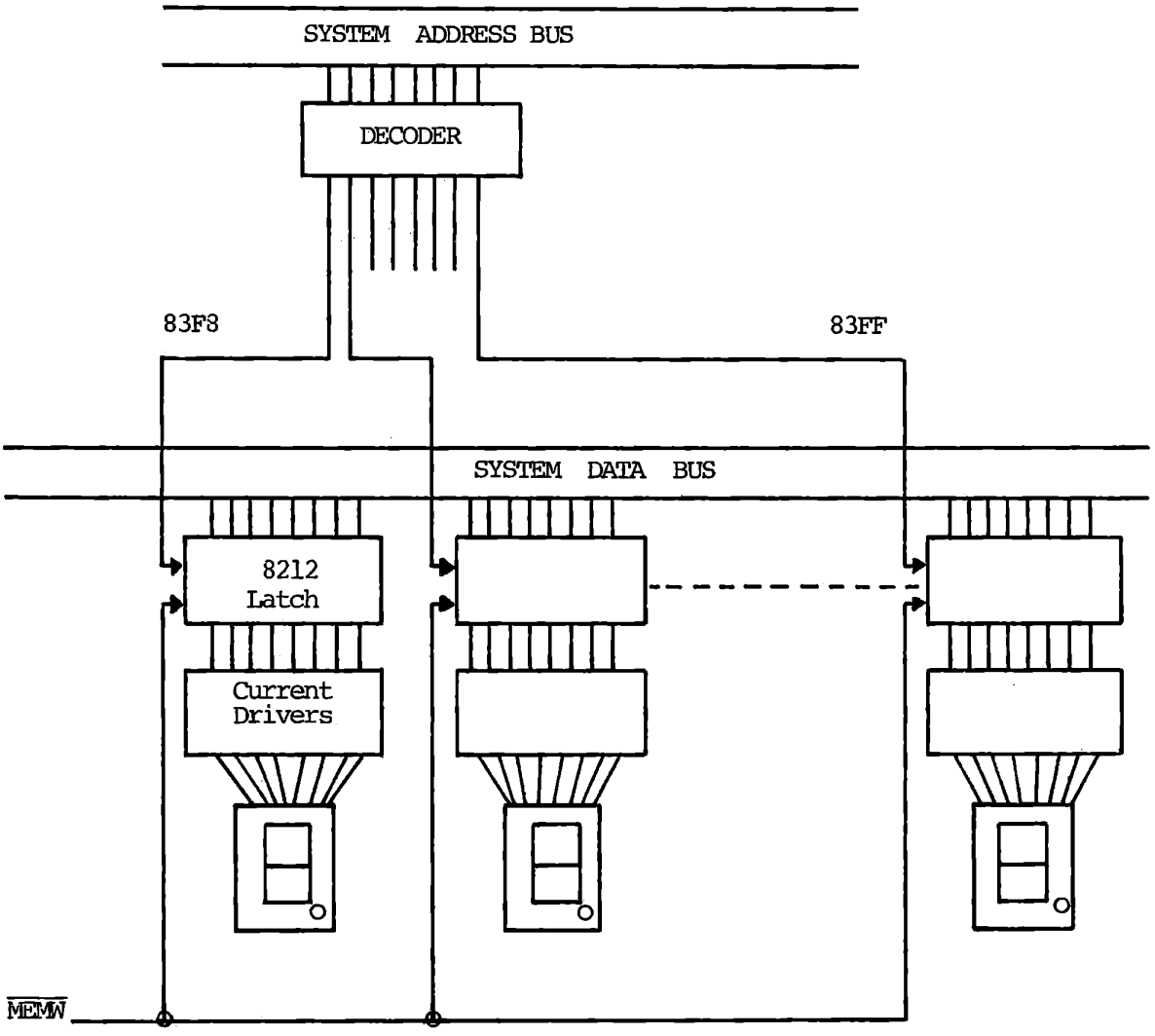
The INR M command is only partially effective. If Port A is programmed for input, you cannot effectively write to it. Nevertheless the flags will be set as though you incremented the data.

This sequence will set zero and jump if the input is 1111 1111, because the flags are set as if 1111 1111 is increased to 0000 0000. It is, of course, not possible to complete the INR M instruction, and store 0000 0000 because Port A is programmed for INPUT and not for OUTPUT.

While memory mapped I/O has some definite advantages, it sacrifices the two byte IN and OUT instructions. LDA and STA are three byte instructions; only by maintaining the I/O address in a register pair do you reduce the program length.

Note that with memory mapped I/O the 8255 must occupy addresses that will not conflict with any real memory. A typical scheme in small systems is to use all addresses from 8000 to FFFF for input/output, and 0000 to 7FFF for memory. Now address bus bit 15 indicates whether an I/O device is to be selected. If only a few I/O devices are to be used it is not necessary to fully decode the address. A single chip 1/8 decoder can select eight different devices.

Memory mapped I/O is probably overused in hardware design. For most applications isolated I/O is more efficient in both hardware and program space - but the difference is very small. The one severe restriction on isolated I/O is that the port address is fixed by the instruction, so it cannot be changed under program control if the instruction is in ROM.



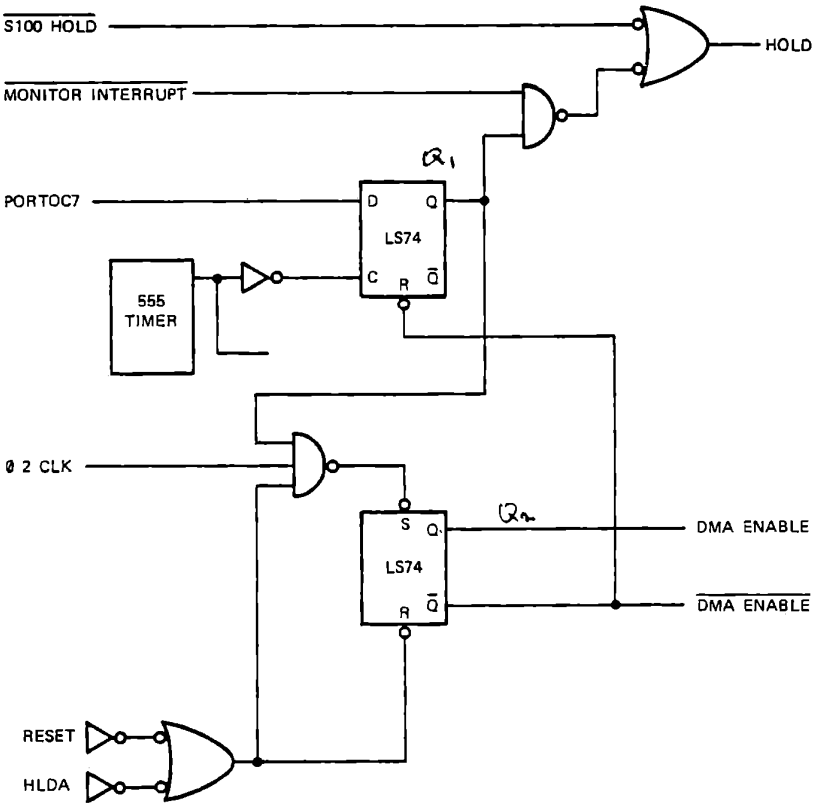
Memory Mapped Display

Figure 8-15

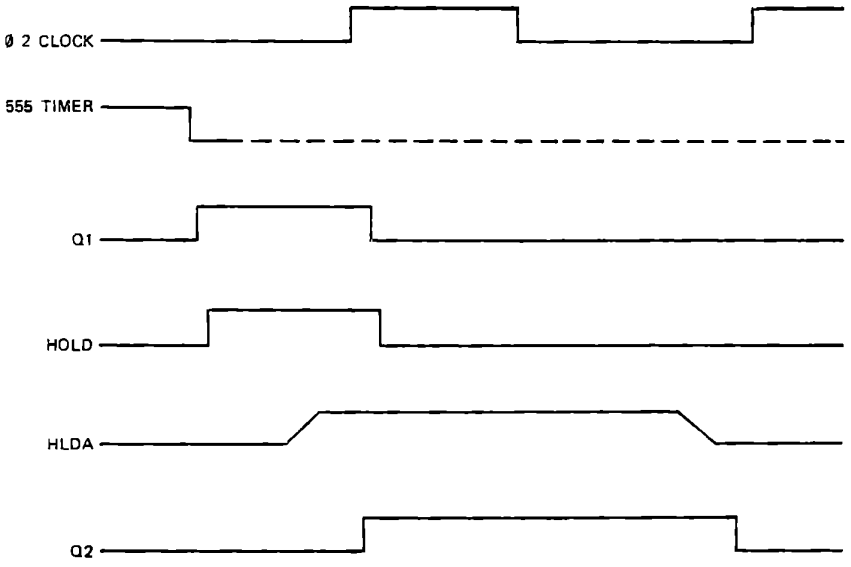
8.3 DIRECT MEMORY ACCESS

The third method of input and output is direct memory access, in which data are written to the processor's memory, or read from it, by external hardware as well as by the CPU. This is very efficient for the program, but typically it demands more external hardware than input and output ports require. In a Direct Memory Access system the DMA hardware ("channel") seizes control of the address and data buses whenever it is ready to read or write to the computer's memory. This briefly suspends the computer's operation, accomplishes the data transfer, and then allows normal computer operation to continue. We will describe in detail the DMA system used in the MTS for its display.

Let us suppose for a moment that we did not have memory devices at addresses 83F8 - 83FF in the MTS, but a set of 8212 output latches, as shown in Figure 8-15. Now to display a digit we would use memory mapped I/O, addressing 83F8, 83F9, etc. and write to those apparent memory locations. The data would be stored in the 8212 latches and would drive the LED displays. This demands eight latches and eight current drivers. Direct Memory Access provides an alternative which in this case takes less external hardware and appears almost identical to the program.



DMA Circuit
Figure 8-16



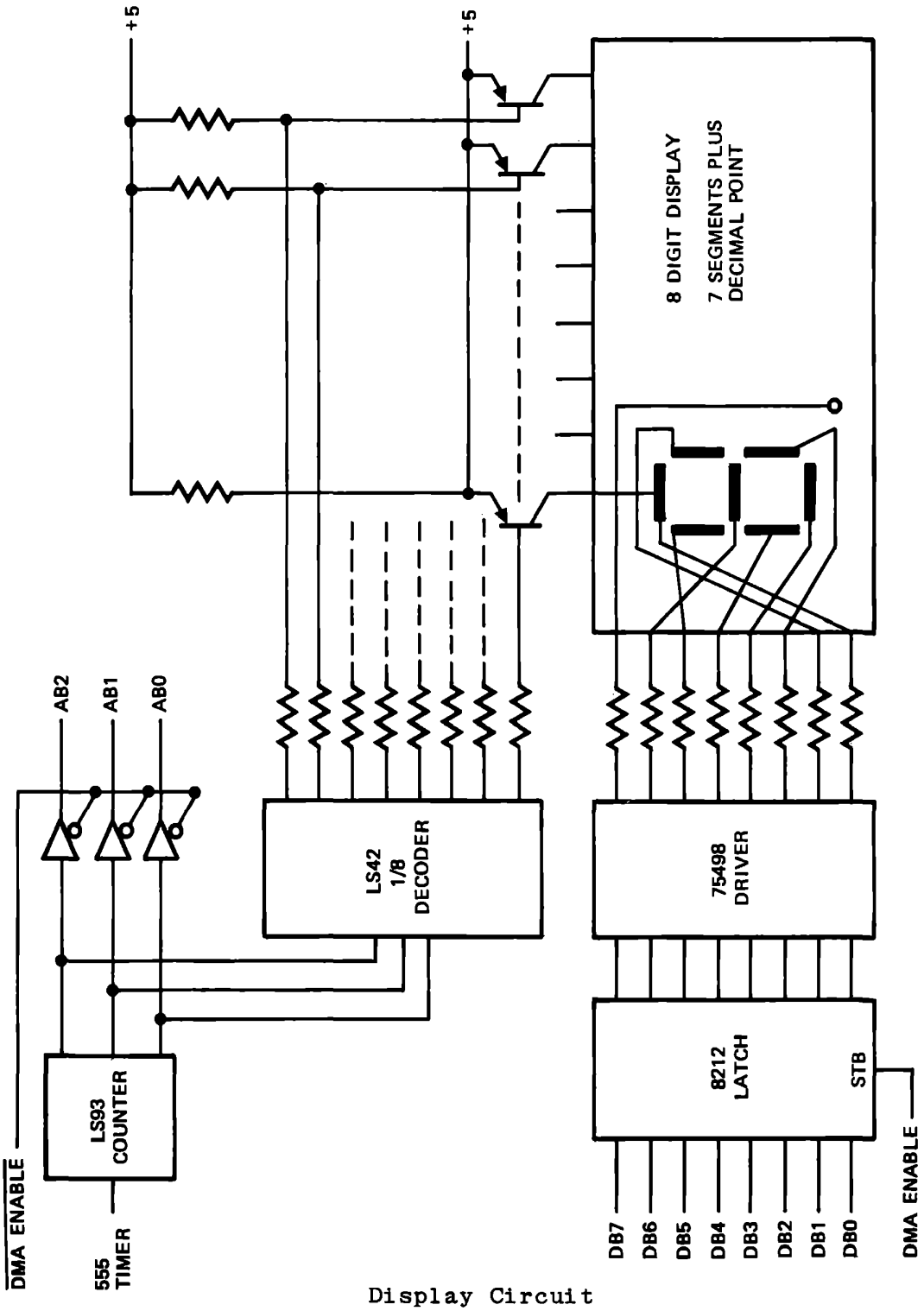
DMA Timing
Figure 8-17

8.3.1 Repetitive Direct Memory Access

In using the seven segment displays of the MTS you have been operating a repetitive direct memory access system. Data are written into a fixed set of addresses, and the DMA hardware periodically obtains data from these addresses and displays it. This is a very attractive scheme for displays of the kind used here, and also for video displays and some kinds of control systems. In each case the same data need to be accessed repetitively because very little external storage is provided. For the seven segment displays of the MTS, only one digit is stored externally, while that digit is illuminated. Then the DMA channel obtains the next digit and displays it.

Figure 8-16 shows the circuit connections to the 8080 that are involved in the DMA operation. The 555 timer periodically generates a pulse which sets the flip flop Q1, provided that the enabling signal from PORT0C7 is high. The output of this flip flop generates a HOLD request to the 8080. It is gated with the monitor interrupt signal to give priority to the interrupt, because internally the 8080 gives priority to the HOLD request, which is undesirable here.

At receipt of HOLD the 8080 suspends its operations, gives HLDA (hold acknowledge) and floats the address and data buses.



Display Circuit

Figure 8-18

HLDA becomes true just before a Ø2 (Phase 2) clock. (See Figure 8-17.) Q1, HLDA and Ø2 clock are gated to set Q2 when all are true. The inverted output of Q2 goes low and immediately resets Q1, terminating the HOLD request. The processor keeps HLDA high for one clock cycle, and then regains control of the buses.

During the single clock cycle (0.5 microsecond) that HLDA is high, the DMA channel controls the address bus. Figure 8-18 shows the connections to the display. A three-bit counter receives the timing pulse from the 555 timer, so at each DMA cycle it counts. Its output selects among the digits and memory locations. When the second flip flop in Figure 8-16 generates DMA ENABLE, indicating that the processor has released the buses, the counter data are connected to Bits 0, 1 and 2 of the address bus. Bits 3 through 9 are pulled high by resistors, and other logic (not shown) forces the chip select to the memory chips representing addresses 8000 - 83FF. Thus one of the data bytes 83F8 - 83FF is selected and is read onto the data bus.

DMA ENABLE strobes the data from the selected memory byte into an eight-bit latch. Thus the data byte from the selected memory locations has been copied into the single latch, and is ready to be displayed. The DMA channel no longer needs access to the memory, and the processor can resume its operation.

The outputs of the three-bit counter drive a 1/8 decoder, turning on one transistor to apply power to one of the eight digits. The data latch and an eight-bit power driver allow current to flow in those segments for which the data bit is 1. The selected digit is illuminated in the appropriate pattern to show the desired character. At the next DMA cycle this digit is turned off and the next digit is turned on, so that each digit operates for one eighth of the time. To make a display that does not flicker visibly, each digit must be turned on about 30 times per second or more. The MTS DMA channel actually operates at about 1400 cycles per second, so each digit is on 175 times per second.

8.3.2 DMA Input and Output

Direct memory access is commonly used in computer systems for both input and output if a high data rate is required. Reading or writing to magnetic disc memory is a typical example; Intel's Microcomputer Development System/Diskette Operating System operates at 250,000 bits per second or about 30 microseconds per byte. The 8080 could not keep up with such a data rate on a programmed or interrupt driven input system. In fact Intel uses their series 3000 Bipolar Microprocessor for the disc controller.

The disadvantage of DMA is the significant amount of external hardware required. It should seldom be used unless high data rates are mandatory, or in specialized situations such as repetitive DMA where the hardware is minimized. The hardware always includes the following:

a) Address counter to store and alter the memory address to be read or written (represented by the 74LS93 Counter in the MTS)

b) Address Bus buffer to isolate the DMA address from the system bus (the 74LS367 Tri-state Buffers)

c) Data Bus buffer to isolate the DMA data from the system bus (the 8212)

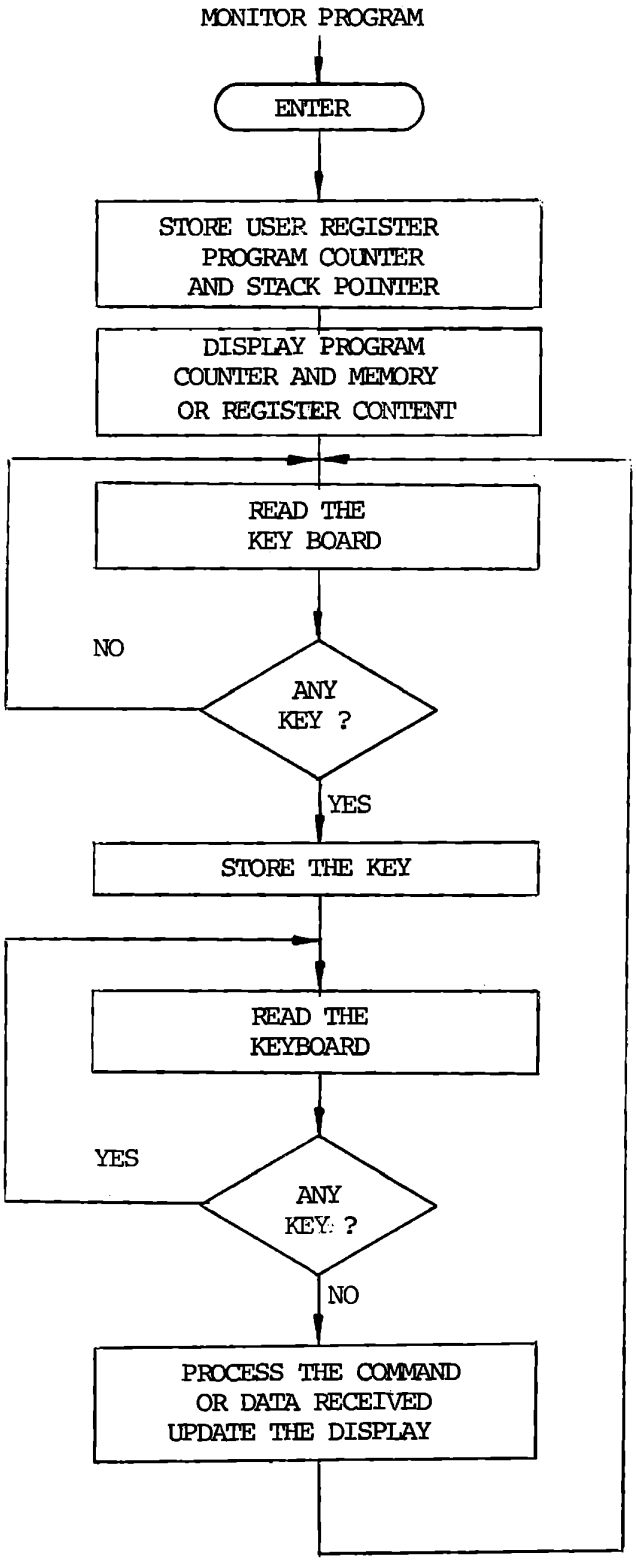
This page intentionally left blank.

d) Gating circuits to appropriately command memory read or memory write. The MTS is only concerned with memory read, for the DMA channel, so this feature is not required in this instance.

e) Timing or signal input to initiate the hold request (the 555).

In any DMA system other than a repetitive DMA there must be some means for the processor to inform the DMA channel that output data are ready, and for the DMA channel to inform the processor that input data have been stored or output data accepted. This can be handled as a separate programmed I/O, with the processor and channel exchanging discrete signals. If DMA input and output are both provided it can be done by writing a control byte into a specified memory location as the last operation in the DMA sequence; then the processor and channel both sample that location periodically. The most common practice, however, is to use a discrete output from the processor to initiate output and enable input, and an interrupt from the channel when data transfer is complete.

Sophisticated DMA systems generally provide for reading and writing to variable areas of memory. For output the processor will send a memory address and a byte count to the channel, which thereafter takes data from the given and succeeding addresses until the designated number of bytes have been read. For input the channel may interrupt to request a memory address where data are to be stored.



Keyboard Testing in the Monitor

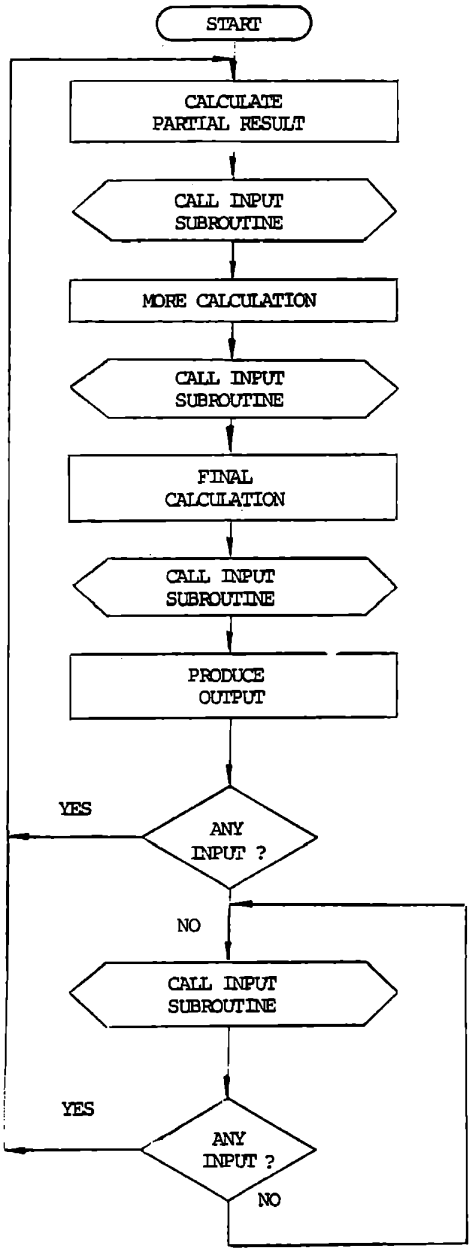
Figure 8-19

8.4 I/O INITIATION

8.4.1 Programmed I/O

Because a computer operates in sequential fashion, it is not always ready to receive an input or produce an output. If it is fast in comparison to the input device or the output requirement, which it often is, the computer can sample the input or produce the output at its own convenience. This is called "Programmed I/O". It is used in the MTS for the keyboard input. When the computer is slow compared to the input or output requirement, as in a magnetic disc system, we use direct memory access, but typically with either programmed or interrupt I/O to initiate and/or terminate the DMA operation. Interrupts are discussed in the following sections of this chapter.

Consider the MTS keyboard input. When the monitor is in control (running), almost all of its time is spent waiting for keyboard input. (See Figure 8-19.) The program has nothing better to do with its time. It can process any command you give it and get back to reading the keyboard long before you can press another key.



Programmed Input/Output

Figure 8-20

The processor can tell whether you have pressed a key because a unique state exists (all inputs high) when no key is pressed. It tests for this state after each new key input before processing the key, to avoid processing a single key stroke repetitively, and yet be able to react to multiple operations of the same key. In many input applications there is no special state which has a significance different from all others, and the processor must know by other means whether a particular input has been processed. There are, of course, applications where it does not matter; a digital voltmeter will process the input as fast as it can update its display whether the data has changed or not.

In some systems the processor has lengthy functions to perform, which must be interrupted to handle input or output. This can be done by repeatedly calling an input subroutine during the main processing, as suggested in Figure 8-20. This tends to be time wasting, and it demands that the programmer consider how long his processing will take in comparison to the input requirement.

We have seen that the strobed input feature of the 8212 enables fleeting and asynchronous data to be latched, until the program is ready to deal with it. This is very suitable for infrequent inputs such as may exist in control systems.

Sometimes, however, the system may demand a very prompt response to its occasional inputs, or it may give many inputs during the course of other calculations, each demanding some degree of processing or at least storage before the next input is delivered. It is for this kind of requirement that interrupt driven systems were invented.

8.4.2 Interrupt Driven I/O

When an external event occurs that demands the processor's immediate attention, hardware is used to cause a branch in the program. Instead of repeated calls to an input (or output) subroutine at predetermined intervals, as suggested in Figure 8-20, that call is created when and only when it is needed. The 8080 and most other microprocessors include interrupt handling capability.

We will discuss the internal and external logic required to create an interrupt; the MTS interrupt system; and the design of interrupt service subroutines.

8.4.2.1 Interrupt Logic

The following signals of the 8080 system are involved in the logic handling an interrupt:

INT Interrupt. Request input to the 8080. It is driven high by external hardware to request service.

INTE Interrupt Enable. A flip flop in the 8080 and also an external output, signifying that an interrupt will be accepted.

INT F/F Interrupt Accept. A flip flop in the 8080 signifying that an interrupt has been accepted.

INTA Interrupt Acknowledge. A signal passed in the status byte to the system controller, and also an output signal from the controller available to external hardware.

To create an interrupt the external logic must (in general) perform two functions: request an interrupt by raising INT, and respond to INTA by giving the 8080 an instruction. The instruction is usually one of the special one-byte restart calls: RST0, RST1, etc. These are essentially identical to the CALL instruction except that the address is implied by the op-code. Thereafter the processor executes an interrupt service subroutine just as it would any other subroutine.

Some systems have a requirement to test INTE to be sure that an interrupt will be accepted. In other systems it can be used as an indication that an interrupt has been accepted. It is not generally necessary to use this signal externally. It is internally gated with the interrupt request, so that interrupts will not be honored unless the interrupt system is enabled.

The interrupt system is enabled by a RESET, or by the instruction:

FB EI Enable Interrupt

This instruction sets the INTE flag high, but it is carefully arranged to be too late for the next instruction to be interrupted. It is guaranteed that one instruction (usually a RETurn from the interrupt service subroutine) will be executed before another interrupt is accepted.

The interrupt system is disabled by execution of an interrupt. This ensures that the interrupt service subroutine can accomplish its functions without itself being interrupted. It can also be disabled by the instruction:

F3 DI Disable Interrupt

This is commonly used when some time dependent task is to be executed and must not be delayed by interrupts, or when a process is being performed that will affect the results of the next interrupt.

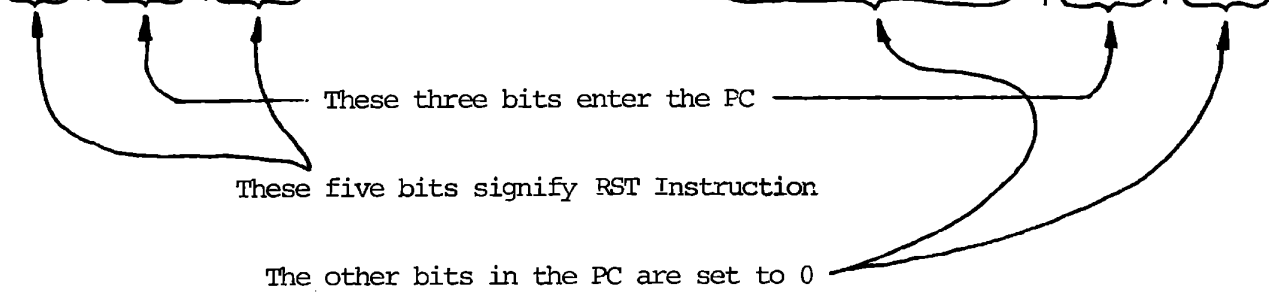
Provided that INTE is set, the INT input sets the internal INT Flip Flop at the end of the current instruction, which is completed before any other action occurs.

When the next instruction cycle starts with INT F/F set, some special events occur. The CPU starts its normal cycle, sending out the PC content and status data. The status includes INTA, a bit on the data bus during status strobe time which commands the system controller to issue the INTA command instead of the MEMR command. Then an instruction is placed on the data bus by external logic so that this is loaded into the instruction register in place of the next programmed instruction. During this cycle the 8080 does not increment the program counter, so the address of the instruction that has been interrupted is preserved. The 8080 clears the INT F/F and the Interrupt Enable Flag, so that the next instruction will not be interrupted.

8.4.2.2 Restart Instructions

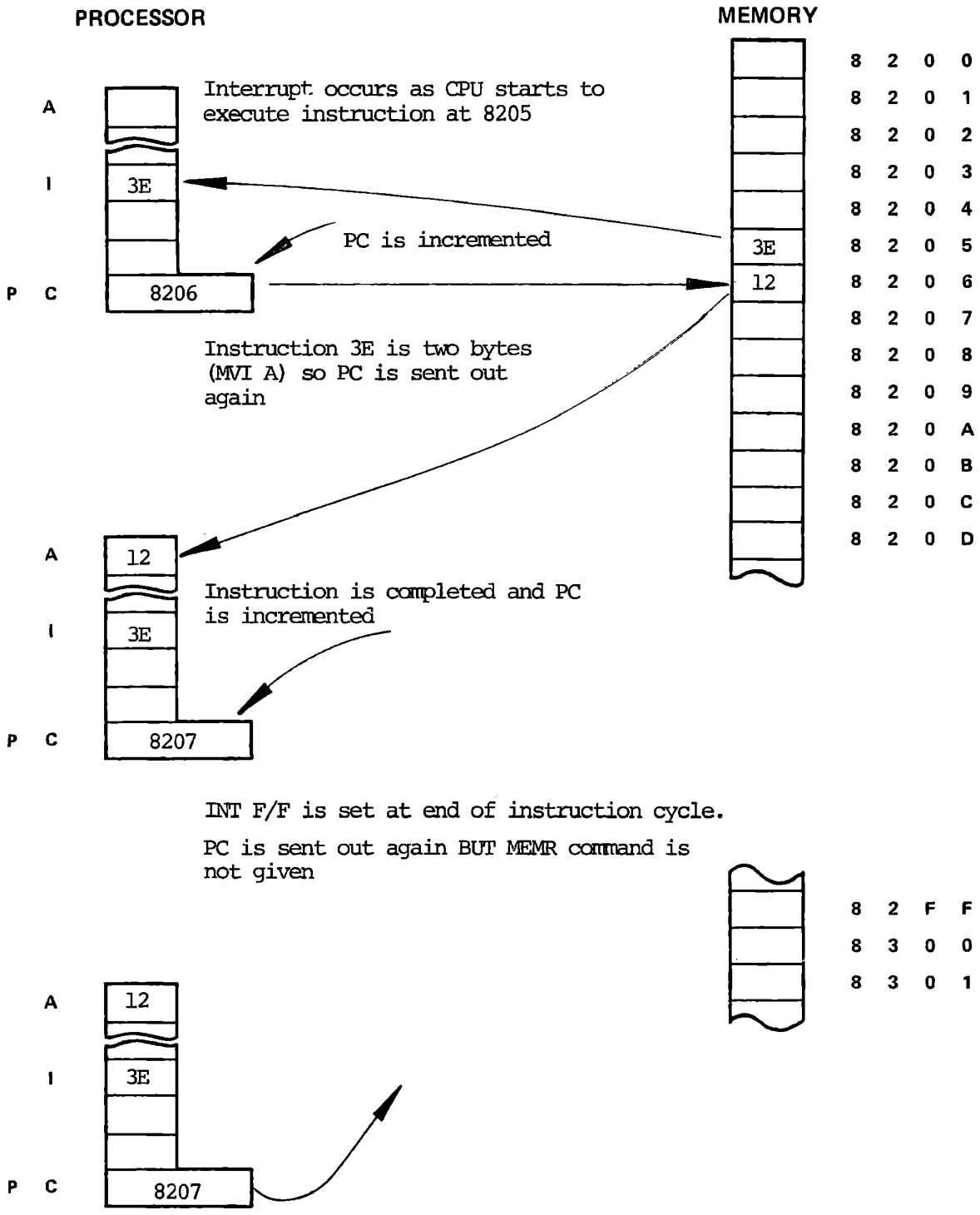
It is usual (but not necessary) that the instruction placed on the data bus in response to INTA is one of the special one-byte call instructions, RST0 to RST7. These are equivalent to normal CALL's except that the call address is implied by the op-code, as shown in Figure 8-21. The diagrams of Figure 8-22 through 8-24 show the process, and Figure 8-25 (from the INTEL 8080 User's Manual) shows the timing.

HEX CODE	INSTR	BINARY CODE			CORRESPONDS TO	NEW PROGRAM	COUNTER	
C7	RST 0	11	000	111	CALL 0000	0000000000000	000	000
CF	RST 1	11	001	111	CALL 0008	0000000000000	001	000
D7	RST 2	11	010	111	CALL 0010	0000000000000	010	000
DF	RST 3	11	011	111	CALL 0018	0000000000000	011	000
E7	RST 4	11	100	111	CALL 0020	0000000000000	100	000
EF	RST 5	11	101	111	CALL 0028	0000000000000	101	000
F7	RST 6	11	110	111	CALL 0030	0000000000000	110	000
FF	RST 7	11	111	111	CALL 0038	0000000000000	111	000



Coding and Effect of RST Instructions

Figure 8-21



Interrupt Processing

Figure 8-22

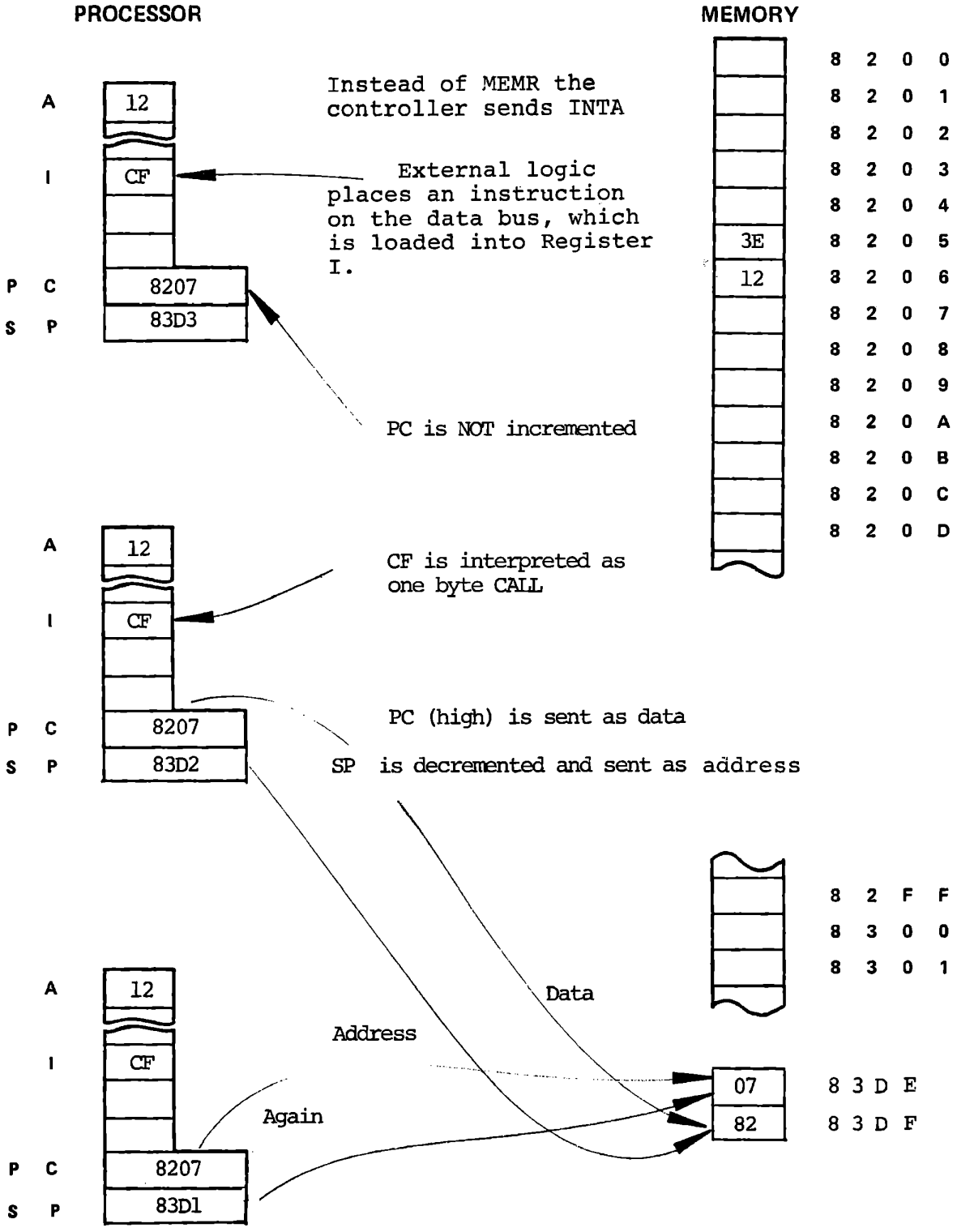

PROCESSOR
MEMORY
A
I
P C
S P
12
CF
8207
83D3
Instead of MEMR the controller sends INTA
External logic places an instruction on the data bus, which is loaded into Register I.
PC is NOT incremented
3E
12
8 2 0 0
8 2 0 1
8 2 0 2
8 2 0 3
8 2 0 4
8 2 0 5
8 2 0 6
8 2 0 7
8 2 0 8
8 2 0 9
8 2 0 A
8 2 0 B
8 2 0 C
8 2 0 D
CF is interpreted as one byte CALL
8207
83D2
PC (high) is sent as data
SP is decremented and sent as address
8 2 F F
8 3 0 0
8 3 0 1
Data
Address
Again
07
82
8 3 D E
8 3 D F
8207
83D1

Figure 8-23

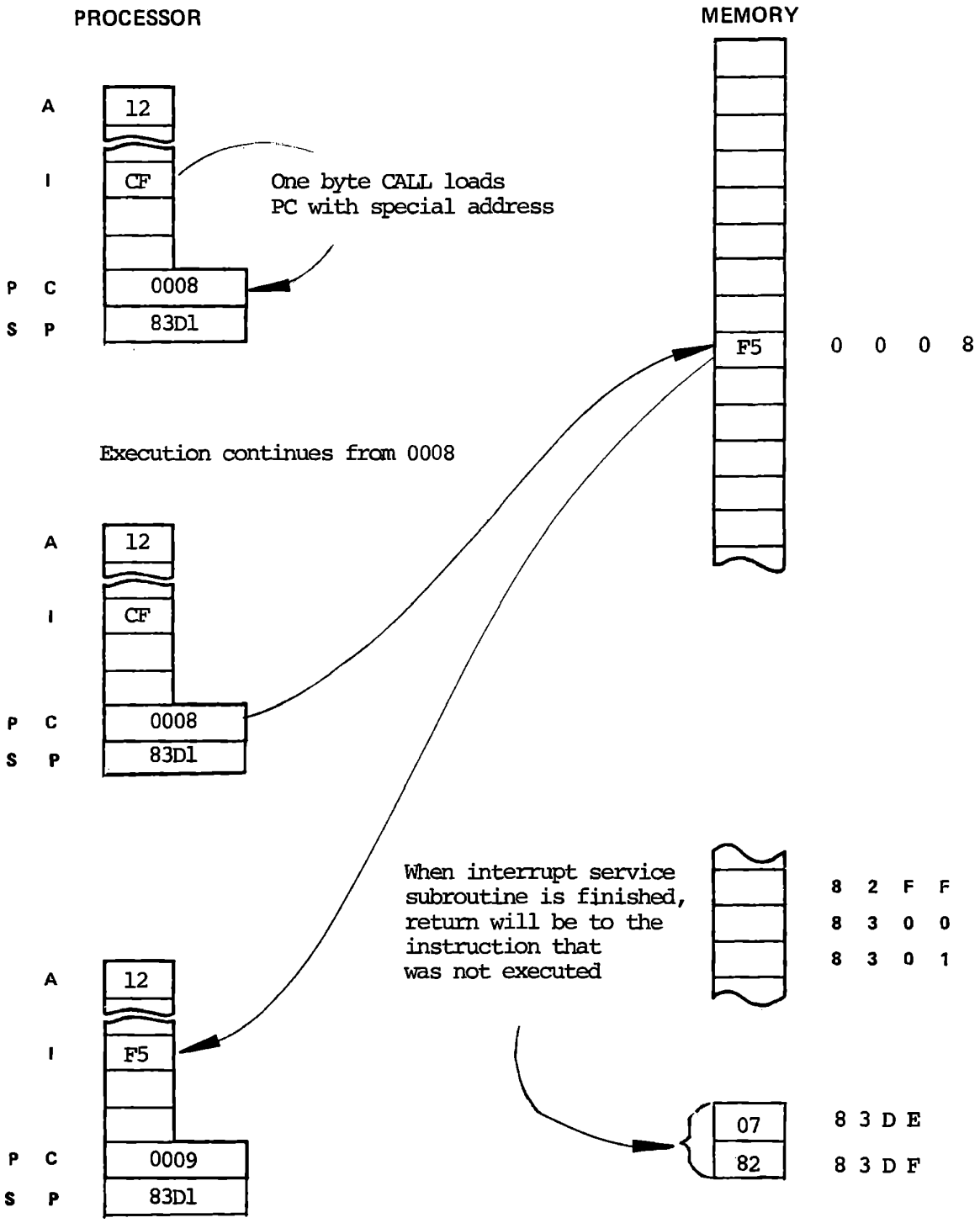

PROCESSOR
MEMORY
A
12
I
CF
One byte CALL loads
PC with special address
P C
0008
S P
83D1
F5
0 0 0 8
Execution continues from 0008
A
12
I
CF
P C
0008
S P
83D1
When interrupt service
subroutine is finished,
return will be to the
instruction that
was not executed
8 2 F F
8 3 0 0
8 3 0 1
A
12
I
F5
P C
0009
S P
83D1
07
82
8 3 D E
8 3 D F

Figure 8-24

INTERRUPT SEQUENCES

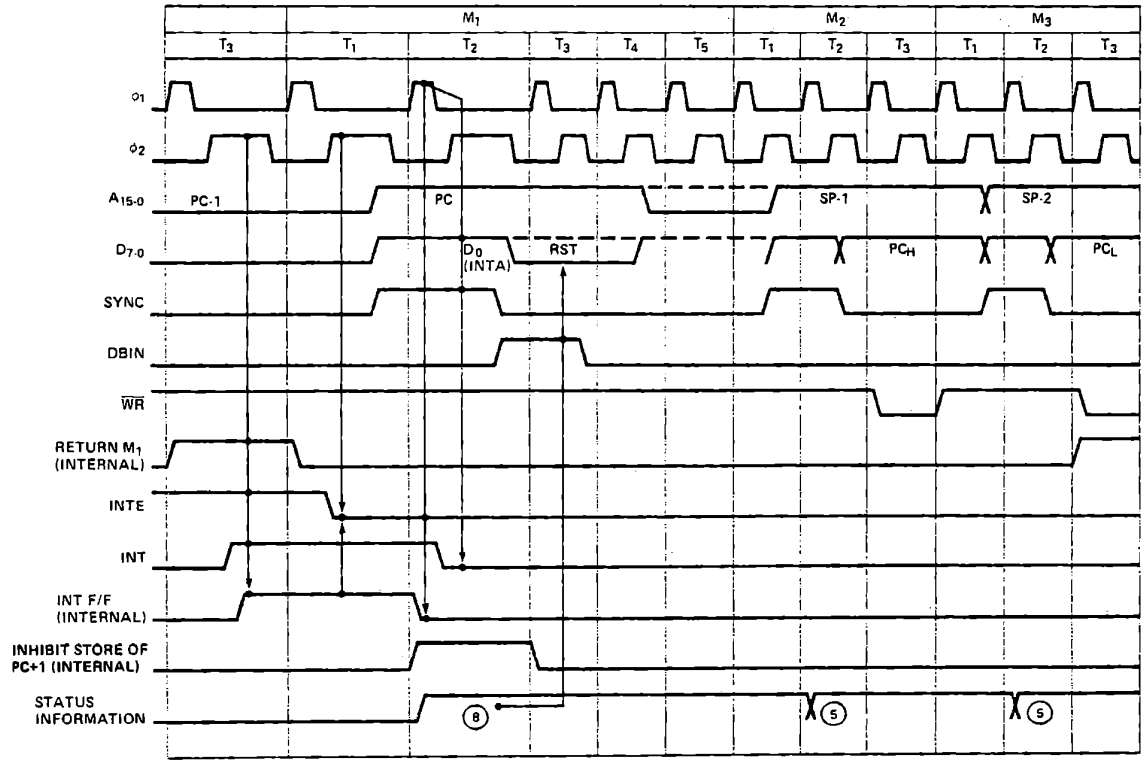
The 8080 has the built-in capacity to handle external interrupt requests. A peripheral device can initiate an interrupt simply by driving the processor's interrupt (INT) line high.

The interrupt (INT) input is asynchronous, and a request may therefore originate at any time during any instruction cycle. Internal logic re-clocks the external request, so that a proper correspondence with the driving clock is established. As Figure 2-8 shows, an interrupt request (INT) arriving during the time that the interrupt enable line (INTE) is high, acts in coincidence with the ϕ_2 clock to set the internal interrupt latch. This event takes place during the last state of the instruction cycle in which the request occurs, thus ensuring that any instruction in progress is completed before the interrupt can be processed.

The INTERRUPT machine cycle which follows the arrival of an enabled interrupt request resembles an ordinary FETCH machine cycle in most respects. The M_1 status bit is transmitted as usual during the SYNC interval. It is accompanied, however, by an INTA status bit (D_0) which acknowledges the external request. The contents of the program counter are latched onto the CPU's address lines during T_1, but the counter itself is not incremented during the INTERRUPT machine cycle, as it otherwise would be. In this way, the pre-interrupt status of the program counter is preserved, so that data in the counter may be restored by the interrupted program after the interrupt request has been processed.

The interrupt cycle is otherwise indistinguishable from an ordinary FETCH machine cycle. The processor itself takes no further special action. It is the responsibility of the peripheral logic to see that an eight-bit interrupt instruction is "jammed" onto the processor's data bus during state T_3. In a typical system, this means that the data-in bus from memory must be temporarily disconnected from the processor's main data bus, so that the interrupting device can command the main bus without interference.

The 8080's instruction set provides a special one-byte call which facilitates the processing of interrupts (the ordinary program Call takes three bytes). This is the RESTART instruction (RST). A variable three-bit field embedded in the eight-bit field of the RST enables the interrupting device to direct a Call to one of eight fixed memory locations. The decimal addresses of these dedicated locations are: 0, 8, 16, 24, 32, 40, 48, and 56. Any of these addresses may be used to store the first instruction(s) of a routine designed to service the requirements of an interrupting device. Since the (RST) is a call, completion of the instruction also stores the old program counter contents on the STACK.



NOTE: (N) Refer to Status Word Chart on Page 2-6.

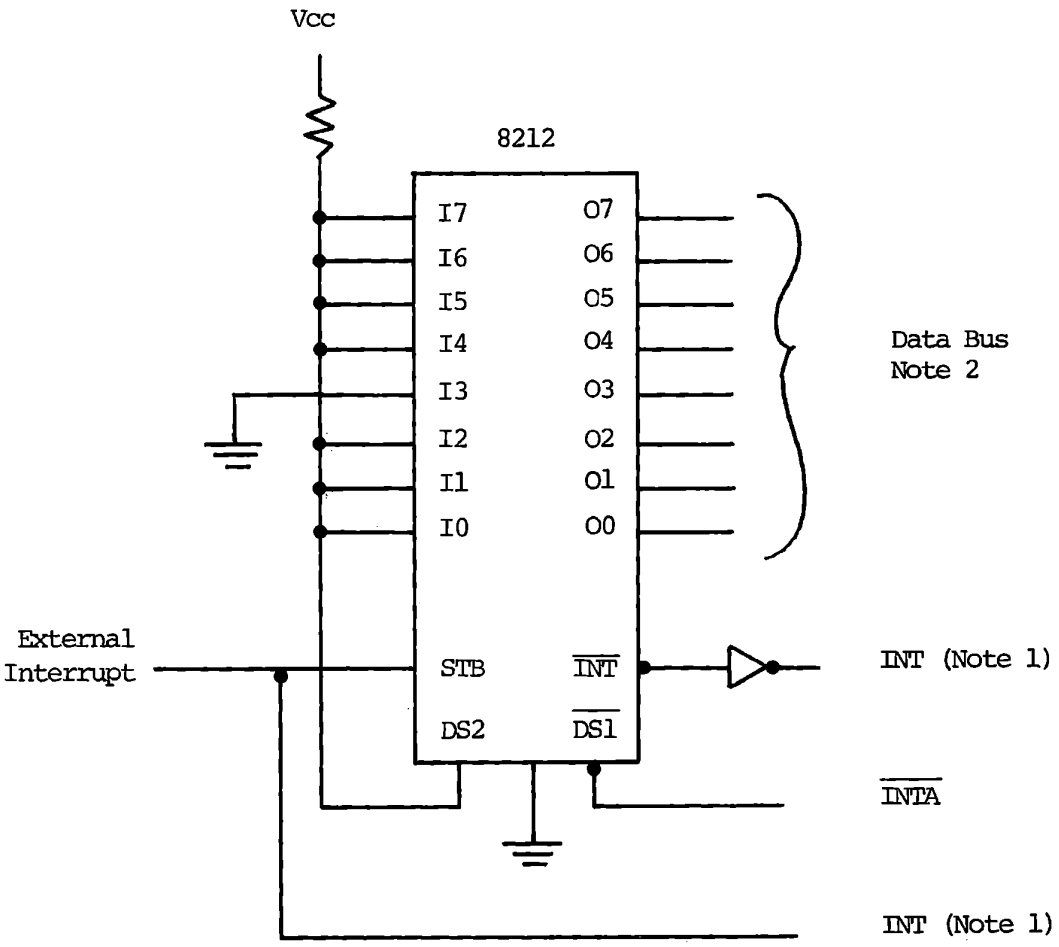
Figure 8-25
(From INTEL Manual)

8.4.2.3 Interfaces for RST Instruction

The restart (or other) instruction that is to be placed on the data bus during INTA must not interfere with the data bus at other times. It is best to buffer the data bus with a tri-state device such as the Intel or NEC 8212, or two 74125 Quad Buffers. Figure 8-26 shows an 8212 generating RST6 in response to an external interrupt.

When more than one device is to interrupt the 8080, it is often useful to use vectored interrupts. Each device creates a different RST instruction, thereby calling a different service routine. Figure 8-27 shows an arrangement with which two independent interrupts can create three different restarts: RST5 for INT1, RST6 for INT2, and RST4 for both at once.

In a small system, the data bus can tolerate some resistive pullup, and tri-state or open collector inverters or gates can be used to pull down specific bits. Figure 8-28 shows such a configuration.

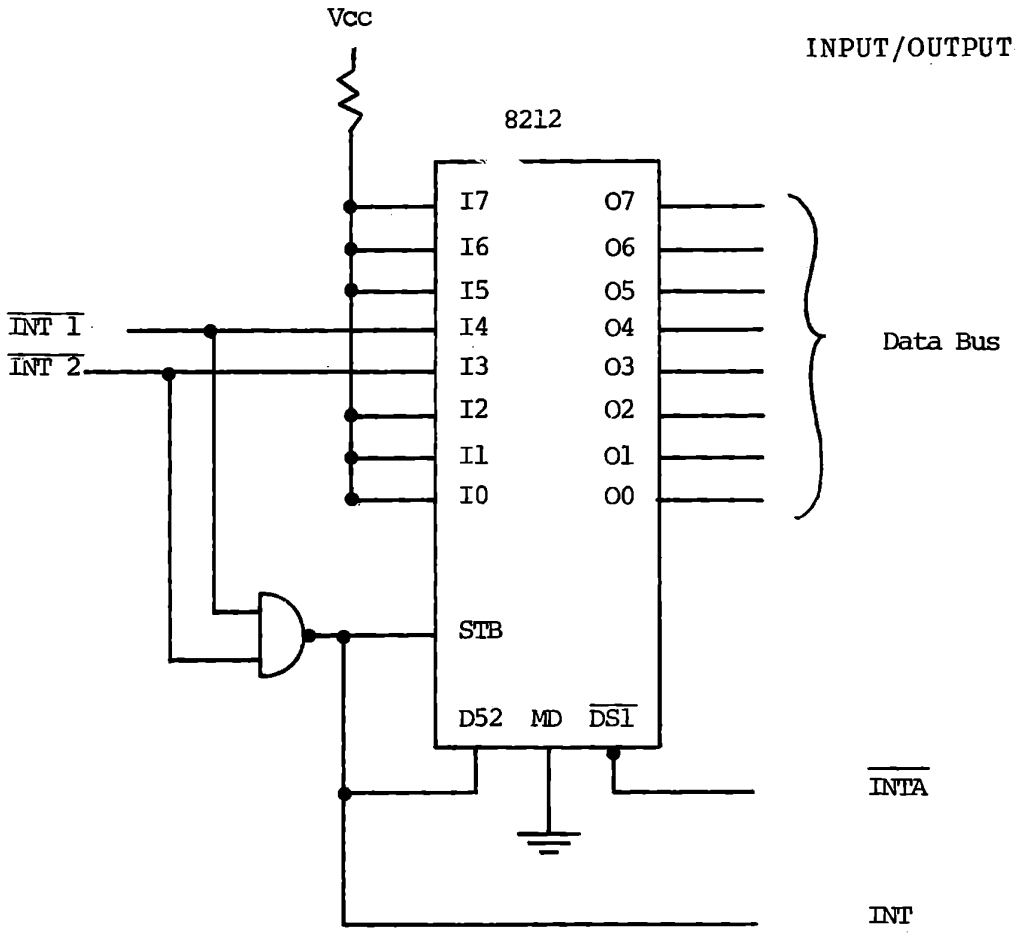


Note 1 : If the external interrupt is a continuous signal it should provide the interrupt to the 8080. If it is a pulse, the 8212 can store it and provide the interrupt request.

Note 2 : The configuration shown places RST 6 (F7) on the data bus during INTA.

Restart Port With 8212

Figure 8-26

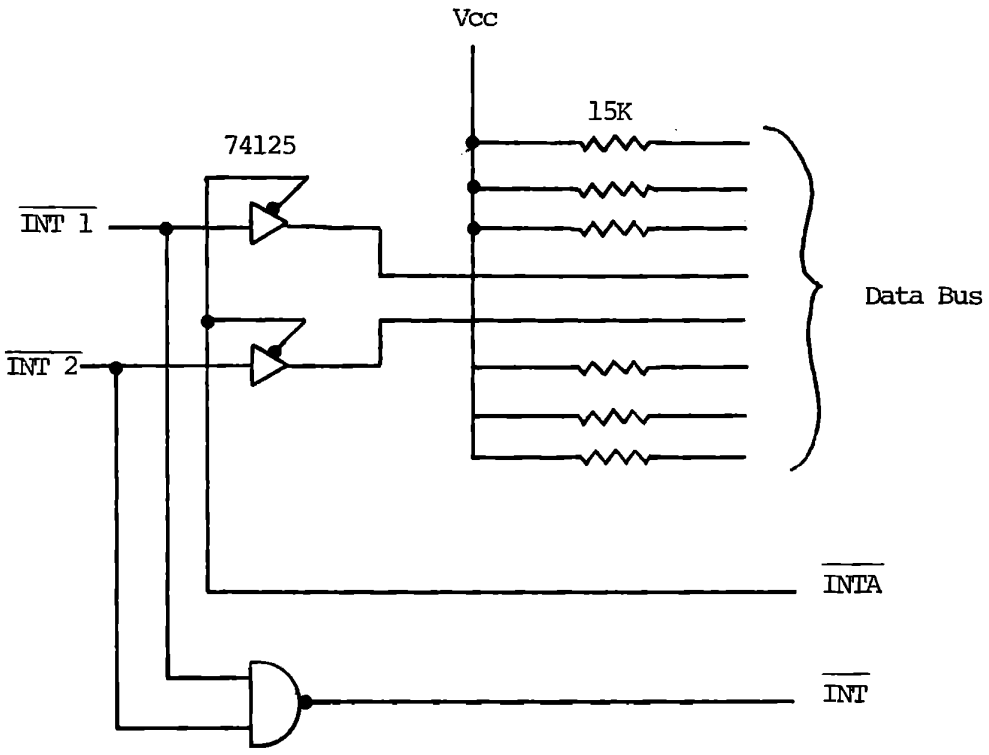


	FUNCTION							
INPUTS:	NONE	INT 1		INT 2		BOTH		OTHER
$\overline{\text{INT 1}}$	1	0	0	1	1	0	0	1
$\overline{\text{INT 2}}$	1	1	1	0	0	0	0	1
$\overline{\text{INT A}}$	1	1	0	1	0	1	0	0
OUTPUTS:								
INT	0	1	1	1	1	1	1	0
BUS	Z	Z	EF	Z	F7	Z	E7	Z

Z = High Impedance State
EF = RST 5
F7 = RST 6
E7 = RST 4

Vectored Restart Port

Figure 8-27



Vectored Interrupt Using Resistors

Figure 8-28

8.4.2.4 Generating RST7

For systems that need only one kind of interrupt, it is easiest to use RST7 (code FF). Simply pulling the data bus high with resistors will enter this instruction when the data bus is floated during INTA. No gates or buffers are needed provided that all other devices on the data bus are able to pull this resistive load down.

In systems that use the 8228 system controller the RST7 interrupt can be generated by pulling the INTA output of the 8228 to +12 volts through a 1K ohm resistor.

8.4.2.5 HALT Instruction

Many microprocessor based systems have no function to perform while they are waiting for input. The program can be made to cycle indefinitely in one place with:

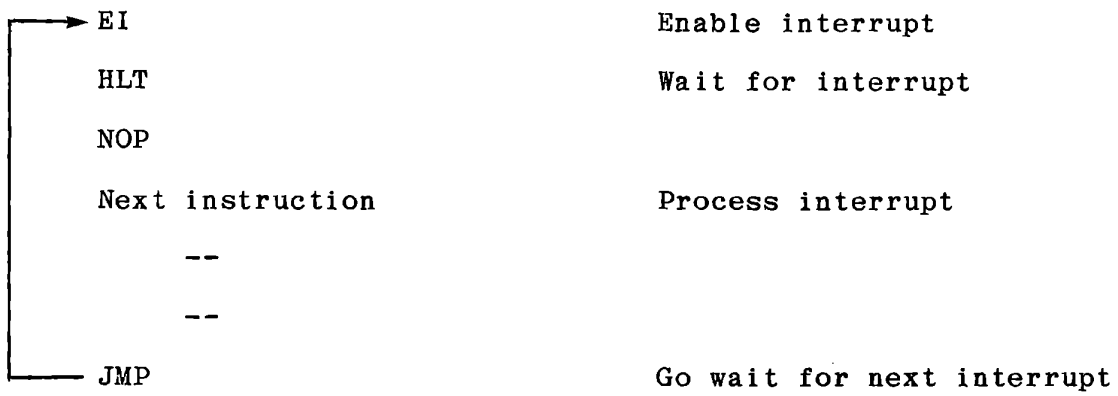
```
8200 C3        JMP 8200
     00
     82
```

Now an interrupt with an RST instruction will call an interrupt service routine which handles all of the processing, and the return will go back to 8200. An alternative is the instruction:

```
76   HLT       Halt at this address until
               an interrupt occurs.
```

When this instruction is executed the processor enters a WAIT state until an interrupt occurs. Now if $\overline{\text{INTA}}$ is OR'ed with $\overline{\text{MEMR}}$, the next instruction in the program will be read and program execution will continue:

Program Flow:



This avoids the need for placing a special instruction on the data bus. Note, however, that the byte following HLT will be read twice because the program counter is not incremented during INTA. Therefore, this instruction should be NOP.

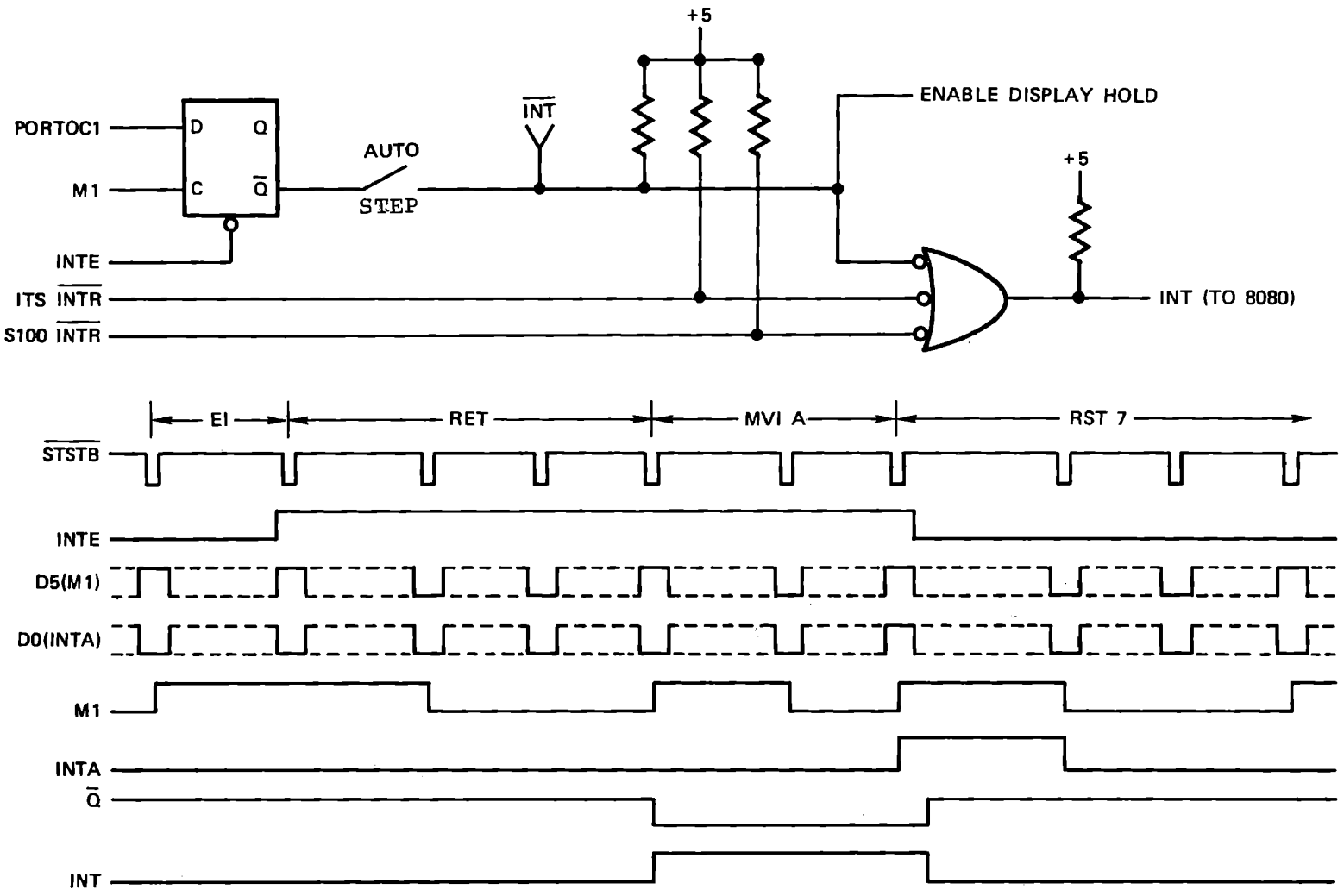
8.4.3 The MTS Interrupt System

When the MTS executes your program in STEP mode (whether it was started with the STEP or RUN key) an interrupt is generated by the MTS hardware as each of your instructions is executed, causing a RST7 that calls the monitor program. The monitor then operates as an interrupt service routine, which we will describe later. The hardware involved will show something of the timing relations of an interrupt system.

8.4.3.1 Interrupt Circuit Details

Figure 8-29 shows the interrupt circuit and timing. Recall that an 8080 instruction cycle comprises one to five machine cycles. Each machine cycle includes three to five clock periods, or states. The first state of each machine cycle is identified by a status strobe signal from the 8224; this is shown in the timing diagram as $\overline{\text{STSTB}}$. During the first state of each machine cycle the 8080 sends out signals on the data bus to identify the operations to be carried out. These are latched by the system controller and provide the information to generate all the control signals - MEMR, MEMW, I/O READ, I/O WRITE and INTA. Status strobe identifies the time at which the status data can be latched by the system controller.

Two of the status bits from the 8080 are used in the interrupt system. M1 identifies the first (or only) machine cycle of an instruction cycle, and appears on data bus bit 5 during status strobe. INTA signifies that the processor acknowledges an interrupt, and appears on data bus bit 0. These are latched by the system controller to be available as continuous signals, as shown in Figure 8-29. INTE (interrupt enable) is available as a continuous signal from the 8080 itself.



MTS Interrupt Circuit and Timing

Figure 8-29

The timing diagram of Figure 8-29 starts with the monitor in control. INTE has been set low at entry to the monitor. The STEP or RUN key has been pressed, so the monitor restores the registers, enables interrupts, and returns to the user's program. The timing diagram shows EI and RET of the monitor program, one user instruction (MVI A) and the RST7 that reenters the monitor.

At the end of the EI instruction the 8080 sets INTE high. Even if an interrupt were already present, however, the RET instruction would not be interrupted, because the 8080 demands that both INTE and INT be true before the end of an instruction to allow the next to be interrupted. This insures that EI, RET can always be executed at the exit from an interrupt service routine. The monitor has an additional requirement: one user instruction must also be executed. The flip flop provides for this.

The flip flop is held reset while INTE is low. When INTE goes high the flip flop is ready to set when M1 goes from low to high, provided that PORT0C1, which is the monitor enable signal, is also high. M1 does not change at the start of the RET instruction, however, because it was already set high by the single instruction EI. During the second and third machine cycles of RET, the M1 signal goes low. Then at the start of the user instruction (shown as MVI A), M1 goes high, clocks the flip flop, and generates an interrupt. This is too late to interrupt the MVI A instruction. At the end of MVI A the 8080 acknowledges the interrupt by sending INTA on data bus bit 0 during status strobe, and by setting INTE low. The flip flop is reset by INTE low, removing the INT input to the 8080, until once again the

monitor executes EI and RET. Thus every user instruction is interrupted before being executed.

Note that when the toggle switch AUTO/STEP is set to AUTO the monitor interrupt is not generated. This allows you to write instructions into experimental programs that enable breakpoint operation in parts that need debugging, but to operate other portions in automatic mode. This may be important in "real time" programs -- where program running time is critical. Section 8.6.4 describes how to disable and enable the monitor. Most of the monitor input subroutines disable monitor interrupts while they are running and enable monitor interrupts before they return.

Referring again to Figure 8-29, you will see that there are other ways that you can cause an interrupt. At the upper right corner of the MTS circuit board there are eight test points. One of these, INT, is connected to the interrupt circuit. We will use this in the following experiment.

8.4.3.2 External Interrupt Experiment

Enter this trivial program

```
8200    C3          JMP8200
8201    00
8202    82
```

Set the toggle switch to AUTO and press STEP (not RUN). Since the monitor interrupts do not reach the 8080, the program will run continuously, but an external interrupt will enter the monitor and

stop program execution. Connect a test lead to INT and touch it to ground. This will create the interrupt and stop your program.

8.4.3.3 Effect of DI and EI

Experiment with the DI and EI instructions. Enter this:

```
8200 F3        DI
  01 3C        INR A
  02 C2        JNZ 8201
  03 01
  04 82
  05 FB        EI
  06 00        NOP
  07 C3        JMP 8200
  08 00
  09 82
```

The DI instruction prevents the external interrupt from being effective until the EI at 8205 enables interrupts again. When you operate this, again using STEP to initiate it but in AUTO mode, your external interrupt with the test lead will always return you to the monitor at address 8207. The interrupt cannot affect the instruction immediately following the EI.

You will find that if you try to operate this program in STEP mode, the monitor will not interrupt it. It is a requirement of the MTS interrupt logic (not of the 8080) that the interrupt is not generated until a multi-cycle instruction has been completed and the next instruction has started. In normal operation this allows the

monitor's return and one user instruction to be executed before the monitor is called again. With this test program the single-cycle NOP does not create an interrupt. The JMP is executed, the monitor initiates the interrupt, but the instruction being processed at that time is Disable Interrupt, which makes the interrupt ineffective even though it had already been received. If you change the instruction at 8206 from NOP to:

```
8206 77          MOV M,A
```

or any other instruction requiring two memory cycles, then the interrupt will occur as the JMP is executed and the monitor will be called before DI is executed at 8200.

8.5 INTERRUPT SERVICE ROUTINES

When an interrupt occurs the interrupt instruction generally calls an interrupt service routine. This is a subroutine, but it has two special requirements. It must:

a) Preserve the environment.

b) Find out why it was called.

8.5.1 Preserving the Environment

An interrupt service routine does not use the registers to exchange data with a calling program. On the contrary, it must preserve the contents of all registers and flags, and restore those contents before returning to the instruction that was interrupted. The interrupted program module makes no special provisions for the interrupt, and except for the time taken by the interrupt service its functions must not be interfered with. It may be interrupted but not disrupted, and the service routine must be transparent.

The first several instructions in any interrupt service routine are almost invariably PUSH instructions to save the registers:

```
PUSH    PSW     Save A and flags
PUSH    B       Save B,C
PUSH    D       Save D,E
PUSH    H       Save H,L
```

The routine can now use all of the registers to perform its functions - typically input and/or output. When finished it restores the environment that existed before the interrupt by popping the registers in reverse order:

```
POP     H
POP     D
POP     B
POP     PSW
EI
RET
```

Remember that the interrupt itself disabled the interrupt system, so to restore the environment, allowing for another interrupt, there must be an EI in the service routine. If this is placed immediately before the return, it is guaranteed that the return will be executed. Placing it earlier in the interrupt routine will allow another interrupt to interrupt the interrupt routine! This is sometimes done, but usually with priority interrupt systems (which are discussed below), and requires special consideration. Many interrupt service routines cannot tolerate being interrupted. This is the case with the MTS monitor, for instance. Other program modules may also be intolerant of interrupts. They must be protected by a DI instructions, and at some point must also include EI.

8.5.2 Identifying the Source of the Interrupt

Commonly a system will have only one generalized interrupt service routine to handle a variety of interrupts. For instance an "intelligent" communications terminal might be interrupted by a transmit next character signal, or by an operator's keystroke. Hardware can be provided to call different interrupt service routines, as we showed earlier. This adds cost and introduces the problem of simultaneous interrupts from different sources. If there is not a severe time constraint it is usually less costly to use programmed I/O rather than providing for vectored priority interrupts. We will define these terms but otherwise in this course will not be concerned with them.

8.5.3 Vectored Interrupt Systems

This is a combination of hardware and software such that each different source of interrupt calls a service routine specific to the device that created the interrupt.

The prior discussion of RST instructions showed how vectored interrupts can be created by placing different instructions on the data bus in response to INTA. Other schemes are possible, for instance the program may store the address of a module to process the next interrupt, if a particular sequence is expected.

8.5.4 Priority Interrupt Systems

A priority interrupt system is a combination of hardware and software guaranteeing that an interrupt from one source is given priority over another; the higher priority can interrupt the lower, or, if they arrive simultaneously, will be handled first. This can be extended to many levels if necessary.

Specific hardware devices (LSI chips) are available to perform this function. In combination with software the 8255 can also create a priority interrupt system.

8.5.5 Timed Interrupt Systems

Systems that need to know the time of day often use a hardware counter, operating on the computer's crystal clock, to generate an interrupt once every millisecond (or any other desirable interval). An interrupt service routine increments a "clock" address in memory. The service routine may also conduct I/O operations at this time, checking each input port to see if any service is needed. This scheme provides frequent service to all I/O ports without requiring each I/O device to create interrupts, and is called "polling".

8.6 USING INTERRUPTS WITH THE MTS

The MTS provides for vectored interrupts using any of the RST instructions except RST0, which is the same as RESET The data bus is pulled high by resistors, so that if no external device places an instruction on the data bus, the 8080 will receive FF during interrupt acknowledge. This is the RST7 instruction, which normally enters the monitor program for a STEP or a breakpoint test. You can connect an external device to enter a different RST instruction, using any of the schemes described earlier.

8.6.1 Interrupt Dispatch

The RST instructions enter the monitor program at these locations:

RST1	0008
RST2	0010
RST3	0018
RST4	0020
RST5	0028
RST6	0030
RST7	0038

Since all of these locations are in Read Only Memory you cannot enter interrupt service routines here. The monitor provides for your interrupt service by loading an address from RAM and jumping to that location when an interrupt occurs.

The actual instruction sequence at the RST location (for RST7) is:

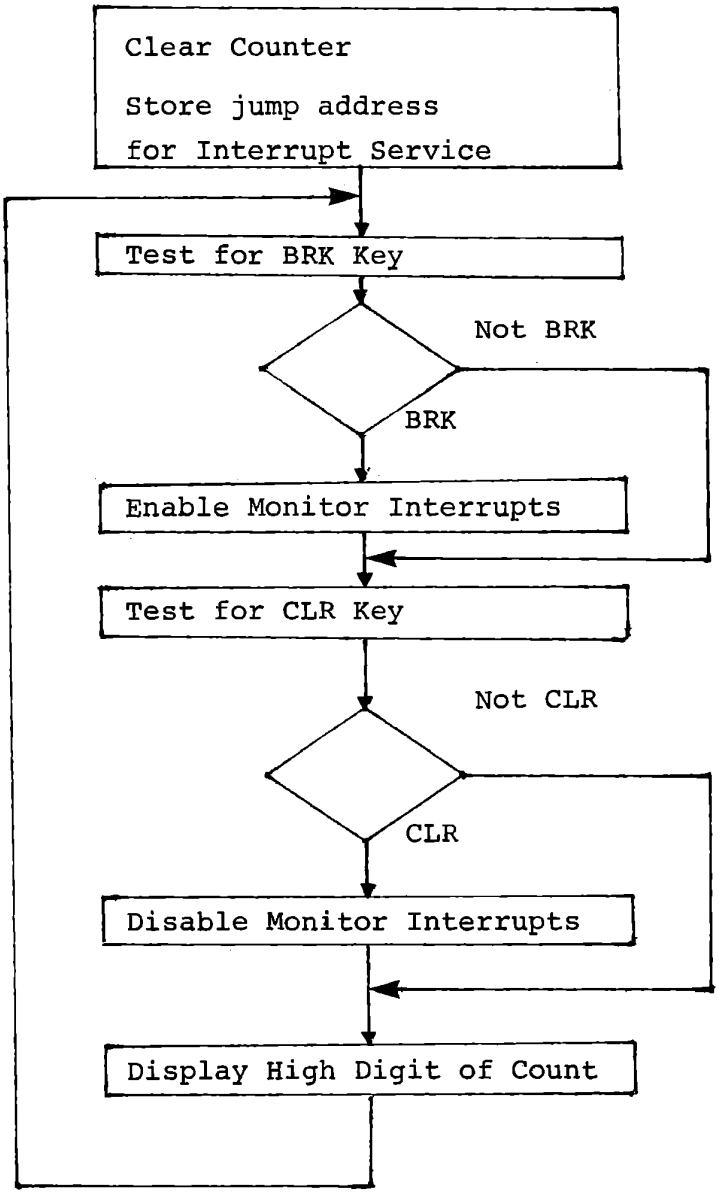
```
0038    E5      PUSH H
0039    2A      LHLD 83E8
003A    E8
003B    83
003C    E3      XTHL
003D    C9      RET
```

The RST instruction pushes the program counter into the stack. PUSH H places (HL) into the stack. Now the jump address is loaded into (HL), using LHLD, i.e. the content of location 83E8 is loaded into Register L and the content of location 83E9 into Register H. XTHL exchanges the content of (HL) with the top two bytes of the stack, so the original value of (HL) is restored and the stack now contains the jump address followed by the address of the interrupted instruction. RET pops the jump address into the program counter, so the program continues at the address that was stored at 83E8 and 83E9. The same instructions, except for different addresses in the LHLD instruction, exist for each of the seven RST instructions. Your program can store a jump address in the appropriate RAM location, and the RET instruction will then jump to that address. Note that it arrives there with the stack top containing the address of the interrupted instruction, and the registers unchanged, exactly as though the RST location had contained a JMP. (RST 4 also contains DI because it is used for programmed calls to the monitor, which must not be interrupted.)

At RESET the monitor loads all the jump addresses. In general your program must replace one or more of these to use interrupts. RST5 and RST6, however, are preloaded to jump into your program area. Refer to Appendix A, Section A.4.2, for the storage location and preset values.

In the following exercise we will develop an interrupt service routine which will be called instead of the monitor when the RST7 interrupt occurs. This can be exercised by operating in breakpoint mode, so that the MTS interrupt circuit will invoke your service routine after each instruction in your main program. We will demonstrate enabling and disabling the monitor interrupt system.

MAIN PROGRAM



Interrupt Service Exercise - Main

Figure 8-30

8.6.2 Interrupt Service Routine Exercise

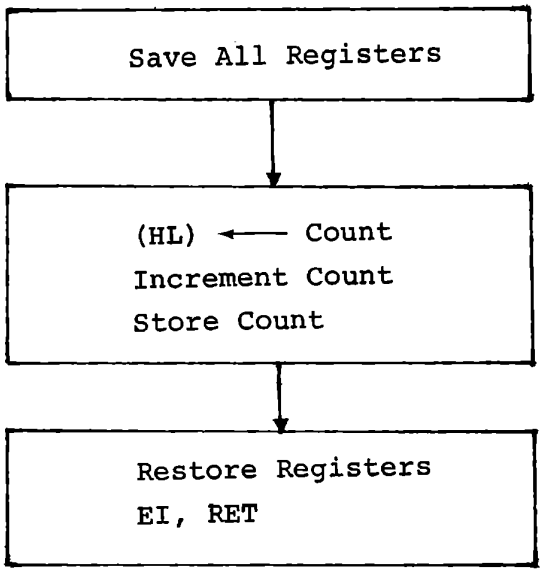
The program to be developed uses the monitor interrupt circuit to generate repeated interrupts, much like a timed interrupt system. The interrupt will call a service routine to increment a two-byte counter in memory. The main program will display the high digit of the count, and test the keyboard. In response to the BRK key it will enable the monitor interrupt system, and in response to CLR, it will disable the monitor interrupts.

When the monitor interrupt circuit is enabled, each instruction in your program will be interrupted, and your interrupt service routine will be called. The dispatch program in the monitor plus your interrupt service will take about 100 microseconds, so the fourth digit of the count (the high digit of the second byte) will count at about 0.4 second intervals.

The solution given used the following memory assignments:

8200 - 820F	Initialize
8210 - 823F	Main Loop
8248 - 8258	KYIN (Figure 8-9)
8260 - 8270	Interrupt Service
8300 - 8301	Counter
83E8 - 83E9	Store Interrupt Service Address

INTERRUPT SERVICE ROUTINE



Interrupt Service Routine

Figure 8-31

8.6.3 Interrupt Service Routine Test

Figure 8-31 shows the interrupt service routine. This is to be located at 8260, and the two byte counter will occupy addresses 8300, 8301. For a preliminary test of the service routine, use a trivial main program that calls this subroutine repeatedly. This will allow you to step through it and check the stack usage. Write the service routine and test program. (A solution is given in Figure 8-32.) Step through the program. After incrementing and storing the count, examine the stack. It is convenient to load the registers with some easily recognized data so you can identify the stack. The pages following the program solution show a testing procedure and the expected results.

TEST FOR INTERRUPT SERVICE

ADDR	CODE												
8 20 0	C	D		C	A	L	L		I	N	T	S	
1	6	0											
2	8	2											
3	C	3		J	M	P			8	2	0	0	
4	0	0											
5	8	2											
6													
7													
8													
9													
A													
B													
C													
D													
E													
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
A													
B													
C													
D													
E													
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													

Figure 8-32a

INTERRUPT SERVICE ROUTINE

ADDR		CODE														
8	26	0	F	5		P	U	S	H		P	S	W			Save Registers
		1	C	5		P	U	S	H		B					
		2	D	5		P	U	S	H		D					
		3	E	5		P	U	S	H		H					
		4	2	A		L	H	L	D		8	3	0	0		Load counter
		5	0	0												
		6	8	3												
		7	2	3		I	N	X		H						Increment and
		8	2	2		S	H	L	D		8	3	0	0		store counter
		9	0	0												
		A	8	3												
		B	E	1		P	O	P		H						Restore registers
		C	D	1		P	O	P		D						in reverse order
		D	C	1		P	O	P		B						
		E	F	1		P	O	P		P	S	W				
		F	F	B		E	I									Reenable interrupts
8		0	C	9		R	E	T								Return to
		1														interrupted instruction
		2														
		3														
		4														
		5														
		6														
		7														
		8														
		9														
		A														
		B														
		C														
		D														
		E														
		F														
8		0														
		1														
		2														
		3														
		4														
		5														
		6														
		7														
		8														

Figure 8-32b

Testing Procedure

REG A	A	8200 A-0A
NEXT	B	8200 B-0B
NEXT	C	8200 C-0C
NEXT	D	8200 D-0D
NEXT	E	8200 E-0E
NEXT	7	8200 F-07
NEXT	8	8200 H-08
NEXT	9	8200 L-09
ADDR, MEM		8200 .CD
STEP		8260 F5
STEP		8261 C5
STEP		8262 D5
STEP		8263 E5
STEP		8264 2A
STEP		8267 23
STEP		8268 22
STEP		826B E1

The displays above assume the coding shown in Figure 8-32. Examine the stack after storing the new count.

Testing Procedure (continued)

ADDR	1/P MEM	83D6 SP.09	Register L
NEXT		83D7 08	Register H
NEXT		83D8 0E	Register E
NEXT		83D9 0D	Register D
NEXT		83DA 0C	Register C
NEXT		83DB 0B	Register B
NEXT		83DC 07	Register F
NEXT		83DD 0A	Register A
NEXT		83DE 03	Return
NEXT		83DF 82	

If you have pushed the registers in some different order, their data will be in a different sequence in the stack.

Check the registers again. Note that only H and L have been changed, since the interrupt service routine used no others. Change the data in the registers.

REG A	826B	A-0A
0	826B	A-00
NEXT	826B	B-0B
0	826B	B-00

etc.

Now step to the return instructions.

ADDR, MEM	826B	E1
STEP	826C	D1
STEP	826D	C1
STEP	826E	F1
STEP	826F	FB
STEP	8270	C9

Check the registers again to be sure that they have been restored properly, and check the stack top to be sure it contains the return address.

ADDR 2/T MEM	8203	ST.C3
STEP	8203	C3

Whenever you write an interrupt service routine, it is a good idea to test it this way. Debugging an interrupt service routine in real time operation is difficult because the monitor is disabled when an interrupt has occurred.

8.6.4 Memory Change Breakpoints

Before going on to the main program, we will use this test program to demonstrate the memory change breakpoint system in the monitor.

RESET	8200	CD
ADDR 8300	8300	??
BRK	8300	BP

Since 8300 contains data rather than an instruction, the program counter should never reach this value. Your program will change the

data stored here, however, and the monitor breakpoint system will stop your program when that occurs.

```
ADDR            8200  CD
RUN             826B  E1
```

The SHLD instruction has changed the content of 8300, and your program is stopped after that has happened. Check the breakpoint.

```
BRK             8300  BP.00
```

Note that the 00 displayed here is not the content of 8300, but is the breakpoint count. Display the memory content by:

```
ADDR 8300 MEM   8300 .01
NEXT            8301  ??
0               8301  00
```

Press ADDR 0 BRK. This automatically enters a breakpoint at the current memory address. The display should show 8301 BP.

Now we have breakpoints at both data locations.

```
RUN             826B  E1
```

The program has stopped again when (8300) was changed. Remove this breakpoint.

```
BRK     8300  BP.00
CLR     8301  BP.00
RUN     836B  E1
```

Now your program has run until the data was changed at 8301.

```
BRK     8301  BP.00
MEM     8301  01
MEM     8300  00
```

The content of 8301 changed when the count went from 00FF to 0100.

The memory change breakpoint is very useful in debugging programs that use interrupts. Since most interrupt service routines store results in specified memory locations, you can enter a breakpoint at such a location. Now program execution will stop after return from your interrupt service routine.

Another use of the memory change breakpoint protects against unbalanced stack usage, which is one of the common errors in coding complicated programs. If you have more PUSH's than POP's in a repetitive loop, the program will fill the stack until it writes over the program and destroys itself. When you have loaded a lengthy program by hand, this can be extremely annoying. To protect against this, enter a breakpoint at the highest location in your program. Now if the stack destroys the data at that point, program execution will stop before the rest of the program is destroyed.

8.6.5 Interrupt Service Operation - Main Program

Write the main program to fulfill the design described in Figure 8-30. Use your KYIN program (Figure 8-9) to test for the keys that we want to recognize.

```
        LXI  B, BF40        Test for BRK
        CALL KYIN           Returns Not Zero if BRK
```

If not Zero enable monitor interrupts.

```
        LXI  B, BF80        Test for CLR
        CALL KYIN           Returns Not Zero if CLR
```

If not Zero disable monitor interrupts.

Since no other interrupts are being used, we could use EI and DI to enable or disable the interrupts. In many real systems, however, it is necessary to selectively enable or disable certain interrupts while allowing others to occur. The MTS hardware allows you to switch the monitor interrupt circuit on or off by setting PORTOC1 high or low.

```
        MVI  A, 03          Set PORTOC1 high
        OUT  CNTO           To Enable Monitor
```

or

```
        MVI  A, 02          Set PORTOC1 low
        OUT  CNTO           To Disable Monitor
```

Use this procedure here. Note that the solution given for KYIN (Figure 8-9) carefully preserves all of the low bits of PORTOC so that it will not affect the monitor enablement. THIS IS NOT TRUE OF THE MONITOR I/O SUBROUTINES.

To display the high digit of the count we will use the only monitor display subroutine that does not affect monitor interrupts:

DISPR (02A6)

DISPR displays the low digit of (A) at the display position addressed by (DE). Call with:

(A) = digit to be displayed, right justified
(DE) = display position

Copies (A) into (C). Loads (HL) with a table address (02B2) and adds (A) into (HL). Copies a seven-segment code from this table into (A) and ((DE)). Decrements (DE). Copies (C) into (A) to restore the entry value.

Since we want to display the high digit of (8301) we must right justify that digit. This can be done by:

```
LDA 8301          Load high byte
RRC               Move high digit
RRC               To low digit
RRC
RRC
CALL DISPR (02A6)
```

The program solution given in Figure 8-33 follows the flow chart of Figure 8-30.

INTERRUPT SERVICE EXERCISE - MAIN

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

ADDR		CODE				
820	0	21		LXI	H, 0000	Clear counter
	1	00				
	2	00				
	3	22		SHLD	8300	Store for
	4	00				interrupt service
	5	83				
	6	21		LXI	H, 8260	Address for
	7	60				interrupt service
	8	82				
	9	22		SHLD	83E8	Store as ENT 7
	A	E8				for dispatch at
	B	83				RST 7
	C	00		NOP		
	D	00		NOP		
	E	00		NOP		
	F	00		NOP		
821	0	01		LXI	B, BF40	Test for BRK
	1	40				
	2	BF				
	3	CD		CALL	KYIN	
	4	48				
	5	82				
	6	CA		JZ	8220	
	7	20				
	8	82				
	9	3E		MVI	A, 03	If BRK set
	A	03				PORTOC1 high
	B	D3		OUT	CNTO	to enable
	C	03				monitor interrupts
	D	00		NOP		
	E	00		NOP		
	F	00		NOP		
8	0					
	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					Figure 8-33a

INTERRUPT SERVICE EXERCISE - MAIN (continued)

ADDR	CODE											
8 22 0	01		LXI	B,	BF80							Test for CLR
1	80											
2	BF											
3	CD		CALL		KYIN							
4	48											
5	82											
6	CA		JZ		8230							
7	30											
8	82											
9	3E		MVI	A,	02							If CLR set
A	02											PORTOC1 low
B	D3		OUT		CNTO							to disable
C	03											monitor interrupts
D	00		NOP									
E	00		NOP									
F	00		NOP									
8 23 0	3A		LDA		8301							Load high byte
1	01											of counter
2	83											
3	0F		RRC									Right justify
4	0F		RRC									high digit
5	0F		RRC									
6	0F		RRC									
7	11		LXI	D,	83FF							Address right
8	FF											display digit
9	83											
A	CD		CALL		DISPR							Display digit
B	A7											
C	02											
D	C3		JMP		8210							Loop to test
E	10											keyboard
F	82											
8 0												
1												
2												
3												
4												
5												
6												
7												
8												

Figure 8-33b

ADDR CODE KPRG, KTST, KYIN WITH DEBUGGING REMOVED

ADDR	CODE												
824 0	3E		MVI		A,	90							KPRG
1	90												Program 8255
2	D3		OUT		CNTO								
3	03												
4	06		MVI		B,	8F							KTST
5	8F												(B) ← all columns
6	0E		MVI		C,	FF							(C) ← all keys
7	FF												
8	DB		IN		PORT	0C							KYIN
9	02												Read existing outputs
A	F6		ORI		F0								Disable all keys
B	F0												Preserve low bits
C	57		MOV		D,	A							of Port 0C
D	A0		ANA		B								Enable selected
E	D3		OUT		PORT	0C							key columns
F	02												
825 0	DB		IN		PORT	0A							Read keys and
1	00												invert so key
2	2F		CMA										pressed = 1
3	A1		ANA		C								Mask unwanted
4	4F		MOV		C,	A							keys and store
5	7A		MOV		A,	D							
6	D3		OUT		PORT	0C							Disable keys
7	02												Preserve low bits
8	C9		RET										of Port 0C
9													
A													
B													
C			IDENTICAL										TO FIGURE 8-9
D			8240-8247										NOT NEEDED
E			FOR		THIS								EXERCISE
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													Figure 8-33c

INTERRUPT SERVICE ROUTINE

ADDR	CODE													
8 26 0	F5		PUSH		PSW									Save registers
1	C5		PUSH		B									
2	D5		PUSH		D									
3	E5		PUSH		H									
4	2A		LHLD		8300									Load counter
5	00													
6	83													
7	23		INX		H									Increment and
8	22		SHLD		8300									store counter
9	00													
A	83													
B	E1		POP		H									Restore registers
C	D1		POP		D									in reverse order
D	C1		POP		B									
E	F1		POP		PSW									
F	FB		EI											Reenable interrupts
8 27 0	C9		RET											
1														
2														
3														
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 8-33d

For an initial test of your program, start after the initialization sequence by:

```
ADDR    8210    STEP
```

This permits normal monitor operation instead of using your interrupt service routine. You can step through to test program flow and the display function. Observe Register A as you step through 8230 - 823A and DISPR. Then RUN (still avoiding the initialization) and observe the display. Since your interrupt service routine is not called the count will not change.

Reset, and set a breakpoint at each of the OUT CNT0 instructions in the main program. Then ADDR 8210 RUN, again avoiding initialization.

```
ADDR    821B    BRK     821B  BP.
ADDR    822B    BRK     822B  BP.
ADDR    8210    RUN
```

The program will be stopped the first time you press BRK, and if you press BRK repeatedly it will stop each time. Because you will still be pressing BRK when the monitor keyboard functions are enabled, the breakpoint will be displayed.

```
BRK                     821B  BP.00
```

Now RUN and press CLR. The monitor will give the usual display of the program counter instruction.

```
CLR                     822B  D3
REG                     822B  A-02
```

When your program writes 02 to CNT0 at the next instruction, the monitor interrupt system will be disabled, just as if you had switched to AUTO mode. Press STEP, and observe that your program runs continuously. CLR will no longer stop at the breakpoint, because the monitor is not enabled to test for the breakpoint. BRK will reenable the monitor and your program will be stopped.

Now that both the main program with its monitor controls and the interrupt service routine have been tested, they can be operated together.

RESET

RUN

The interrupt system is enabled, your interrupt service routine is called to count in the memory locations 8300 and 8301, and the count is displayed. CLR will disable the interrupts and counting will stop. BRK will enable interrupts and counting will proceed.

8.6.6 Combining Interrupt Service with Monitor Functions

In the preceding exercise the monitor breakpoint functions are not available, since the monitor interrupt has called your interrupt service instead of calling the monitor. You can have both functions, however. Replace the EI, RET instructions in your interrupt service routine with JMP 006A. Now after your interrupt service has been processed and the registers have been restored, normal monitor functions will be resumed. You can step through your main program, or enter breakpoints in your main program (but not in your interrupt service routine), or enter breakpoints at data storage locations. Set a breakpoint at 8301. The program will be stopped each time the high byte of your counter is incremented. The monitor will show the address of the interrupted instruction as the program counter. The address will vary because the program is only stopped after 256 instructions have been executed. Clear the breakpoint and observe that counting is much slower than it was with only your own interrupt service. This illustrates the amount of time required by the monitor in checking for breakpoints.

8.6.7 External Interrupt

You can introduce an external interrupt instead of using the monitor interrupt source. Switch to AUTO mode and connect a clip lead to the INT pin at the upper right corner of the circuit board. Each time you ground it your interrupt service routine will be called. Modify your program to display the entire two-byte counter.

```
8230    2A      LHLD 8300
8231    00
8232    83
8233    CD      CALL DWORD
8234    D1
8235    02
8236    C3      JMP 8230
8237    30
8238    82
```

You will see a number of different counts each time you ground the clip lead. The grounding will not make a single clean connection, but will open and close many times.

8.6.8 Interrupt Handling - Summary

In this exercise you have written and tested an interrupt service routine, and used three important monitor features related to interrupt operations:

a) Storing a jump address for dispatch of an interrupt to your service routine. This is available for any of seven RST instructions.

b) Enabling and disabling monitor interrupts while leaving other interrupts enabled.

c) Entering a data change breakpoint. This permits stopping program execution whenever an interrupt has been serviced.

Other ICS courses deal much more extensively with interrupt systems, and are recommended for students concerned with more detailed treatment of interrupt hardware and programming.

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 9

DATA FORMAT

9. DATA FORMAT

In Chapter 8 you used only discrete inputs and outputs, each bit being essentially independent of all others. An output at C4, C5 or C6 selects a column of the keyboard; an input at any bit of Port A comes from one key. The timing of inputs and outputs, apart from their sequence, has no meaning. We will now consider parallel I/O, where a data byte representing a number is transferred, and serial I/O, where the timing of signals carries information.

Input Control Signal Definition

$\overline{STB}$ (Strobe Input)

A "low" on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A "high" on this output indicates that the data has been loaded into the input latch; in essence, an acknowledgement. IBF is set by the falling edge of the STB input and is reset by the rising edge of the RD input.

INTR (Interrupt Request)

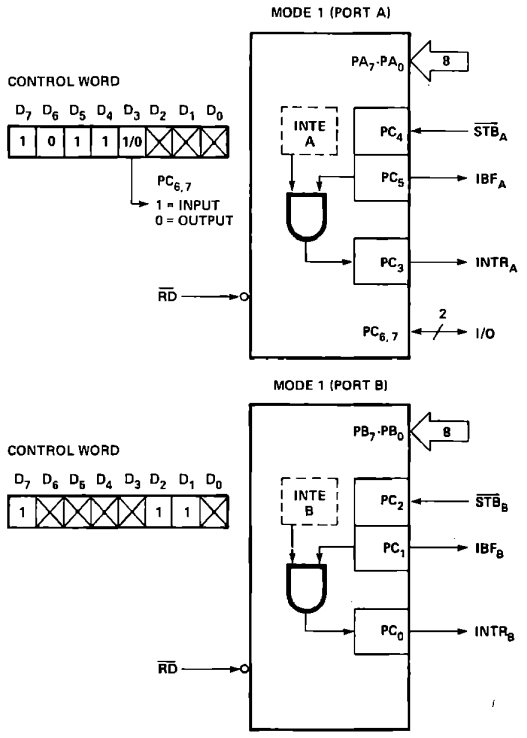
A "high" on this output can be used to interrupt the CPU when an input device is requesting service. INTR is set by the rising edge of $\overline{STB}$ if IBF is a "one" and INTE is a "one". It is reset by the falling edge of $\overline{RD}$. This procedure allows an input device to request service from the CPU by simply strobing its data into the port.

INTE A

Controlled by bit set/reset of PC_4.

INTE B

Controlled by bit set/reset of PC_2.



Mode 1 Input

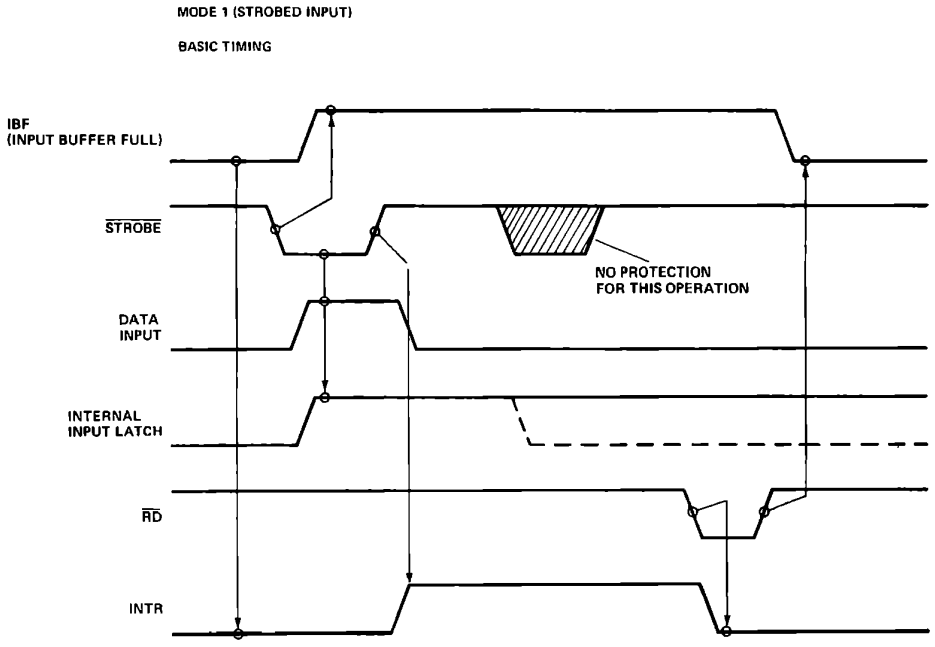


Figure 9-1

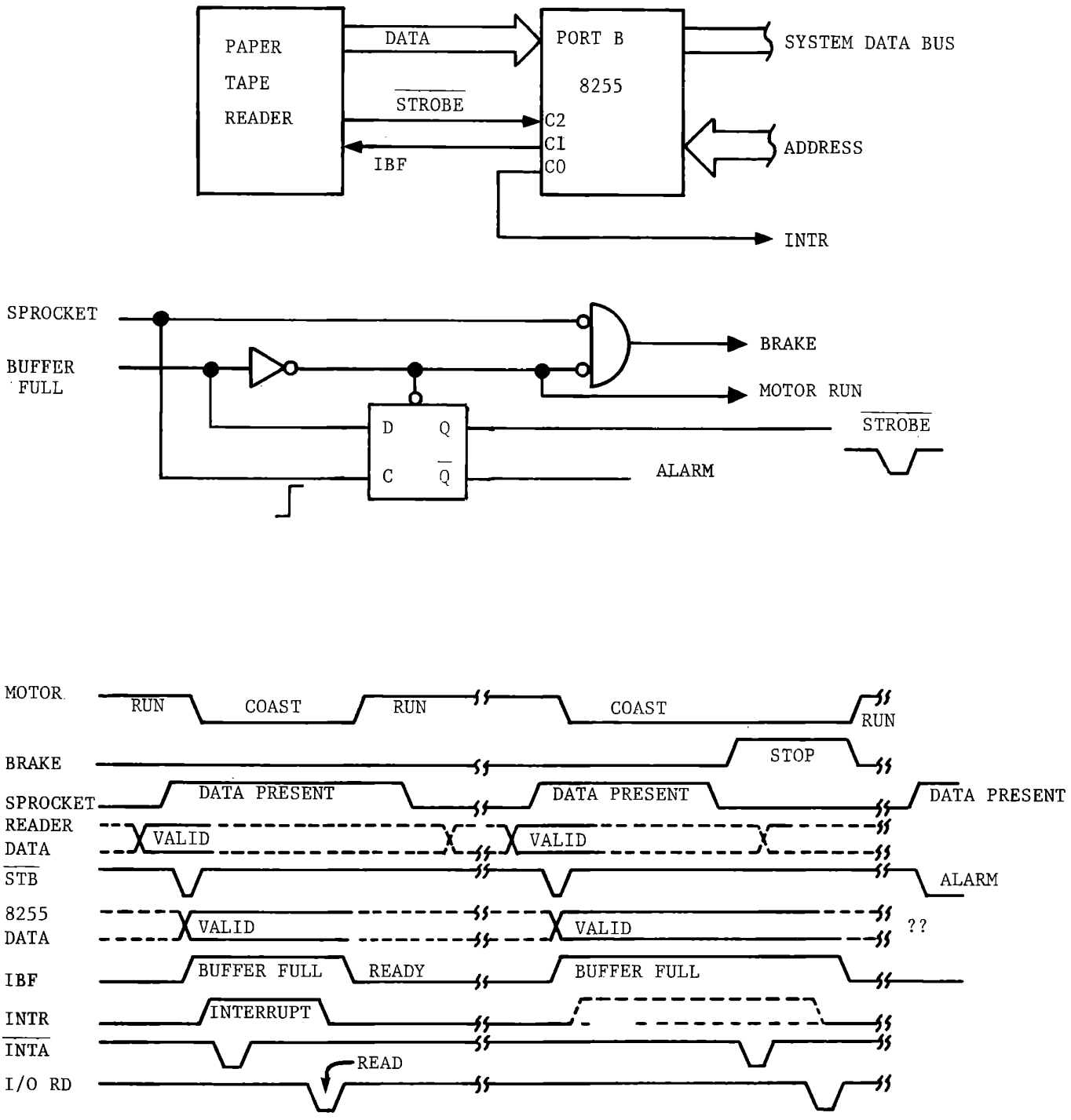
Basic Timing Input

9.1 PARALLEL INPUT/OUTPUT

Clearly the 8255 data ports are principally intended for 8-bit, parallel data transfer. Such data might come from a paper tape reader, an analog to digital converter, another computer, a keyboard that includes built-in scanning and decoding, or a communications device that includes serial to parallel conversion. A usual characteristic of such devices is that they generate a strobe signal indicating that an input byte is ready for the computer. When Port A or Port B of the 8255 is programmed to Input Mode 1, it uses some bits of Port C to handle the strobe and give an interrupt to the 8080, and responds with an acknowledgement to the input device when the computer has accepted the data. Some input devices are designed to demand such an acknowledgement before entering the next byte, or to recognize an error condition if it is not received.

9.1.1 Paper Tape Reader Example

Figure 9-1 shows bit assignments and timing for Mode 1 input through an 8255. Consider how this would be used with a high-speed paper tape reader.



High Speed Paper Tape Reader Interface

Figure 9-2

The photoelectric reader senses holes in the paper tape. The sprocket hole (which is present at every character position even though there may be no other holes) is sensed to indicate that the data holes are in position to be read. The sprocket hole signal provides the strobe to latch data into the 8255. The logic and timing diagram of Figure 9-2 shows the sprocket hole signal clocking a D flip-flop. The IBF signal is taken into the D input. Since it is (presumably) low, indicating that the buffer is ready to take data, the flip-flop is reset. Its output is the strobe signal; this enters the data into the 8255 data latch and sets IBF high. IBF high sets the D flip-flop through the asynchronous set input, ending the strobe pulse and latching the data. The end of strobe sets the 8255's interrupt request output. The 8080 acknowledges the interrupt, calls the interrupt service routine, and reads the data from the 8255.

The act of reading (I/O RD) resets IBF, indicating that the buffer is again available. All of this is normally accomplished while the sprocket hole is still visible to the reader. (At 1000 characters per second it lasts for about 200 microseconds, time enough for a reasonable interrupt service routine). While the IBF signal is high the reader's motor is allowed to coast; when IBF is reset it runs again.

This page intentionally left blank.

In the second segment of the timing diagram the CPU is not available to read the data promptly. Either it has disabled the 8255 interrupt, or its program has disabled all interrupts. The IBF signal stays high beyond the sprocket hole signal. This signals the paper tape reader that, although the 8255 has accepted and latched the present character, it may not be ready in time for the next. The mechanism now applies a brake to stop paper motion before the next character. When the data are finally accepted by the CPU by an I/O Read, the motor can run again.

The final segment of the timing diagram shows a failure: IBF is not set by the strobe (perhaps the 8255 has been reprogrammed). Strobe goes low but fails to rise again. This can generate a visible alarm signal to indicate a loss of data.

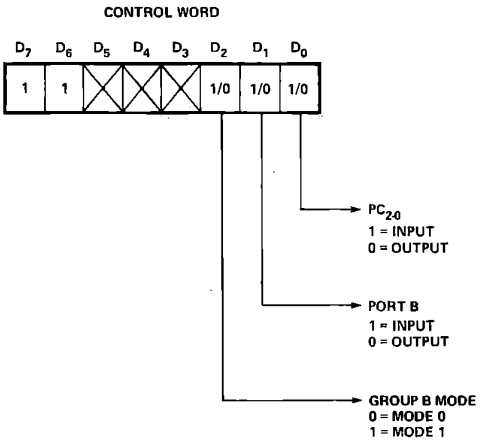
9.1.2 Computer to Computer Interface

Some applications overburden a microprocessor, particularly when two or more tasks require fast interrupt service response. One solution, of course, is to use a faster or more powerful computer such as a bipolar bit-slice machine, whose instruction time may be a small fraction of the 8080's. Often it is more economical to divide the task between two microprocessors. They will then need to communicate with each other. This can be handled in three ways:

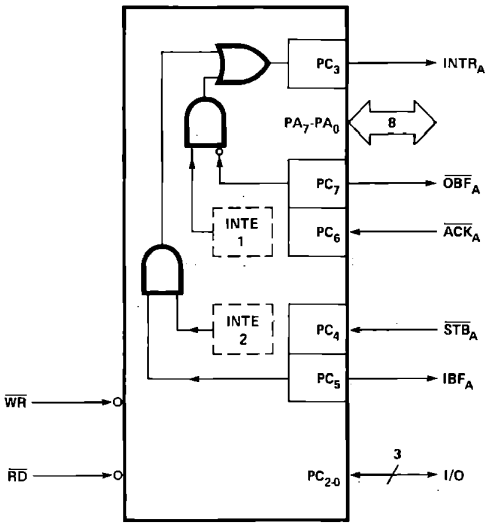
a) Through input/output ports

b) Direct memory access

c) Memory sharing



Mode 2 Control Word



Mode 2

Operating Modes

Mode 2 (Strobed Bi-Directional Bus I/O)

This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bi-directional bus I/O). "Handshaking" signals are provided to maintain proper bus flow discipline in a similar manner to Mode 1. Interrupt generation and enable/disable functions are also available.

Mode 2 Basic Functional Definitions:

- Used in Group A only.
- One 8-bit, bi-directional bus Port (Port A) and a 5-bit control Port (Port C).
- Both inputs and outputs are latched.
- The 5-bit control port (Port C) is used for control and status for the 8-bit, bi-directional bus port (Port A).

Bi-Directional Bus I/O Control Signal Definition

INTR (Interrupt Request)

A high on this output can be used to interrupt the CPU for both input or output operations.

Output Operations

$\overline{OBF}$ (Output Buffer Full)

The $\overline{OBF}$ output will go "low" to indicate that the CPU has written data out to Port A.

$\overline{ACK}$ (Acknowledge)

A "low" on this input enables the tri-state output buffer of Port A to send out the data. Otherwise, the output buffer will be in the high-impedance state.

INTE 1 (The INTE Flip-Flop associated with $\overline{OBF}$)

Controlled by bit set/reset of PC_6.

Input Operations

$\overline{STB}$ (Strobe Input)

A "low" on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A "high" on this output indicates that data has been loaded into the input latch.

INTE 2 (The INTE Flip-Flop associated with IBF)

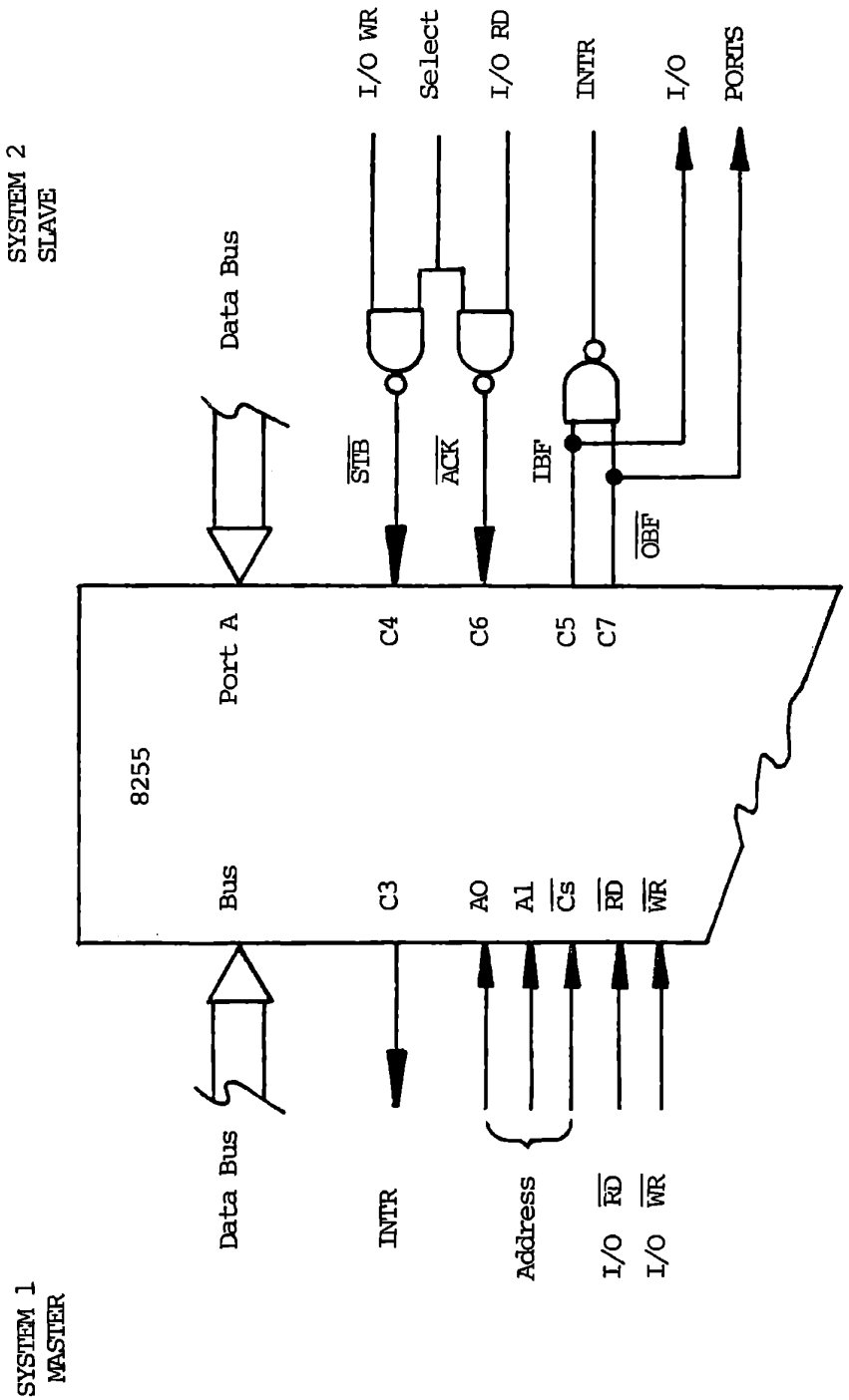
Controlled by bit set/reset of PC_4.

Figure 9-3

9.1.2.1 I/O Port Interface

One computer can write to a data latch (such as the 8212) and create an interrupt to another computer, which can read the data through a similar port or through a tri-state buffer. The 8255 can operate Port A as a tri-state, bi-directional bus interface, avoiding the need for a second device between the systems. The 8255 is connected as an I/O port to one 8080 (the master) and Port A is connected to the data bus of the other (the slave). Five bits of Port C are used for handshaking between the processors; the slave needs additional gating to enable Port A to interact with its bus.

Figure 9-3 defines Mode 2 of the 8255, and Figure 9-4 shows the connection between two processors through the 8255. The master writes and reads Ports A and B as in any other use of the device. The slave is connected to Port A. It can address the 8255 through an I/O Read or Write with a port address that gives the "Select" signal. I/O Write and Select generate an $\overline{\text{STB}}$ input to C4, latching the slave's data bus content into the Port A input latch, to be read by the master. I/O Read and Select generate an $\overline{\text{ACK}}$ input to C6, which places the output latch content onto the port A outputs and so onto the slave's data bus. Otherwise Port A is in the high impedance state. IBF (Port C5) goes low when the input buffer is empty. $\overline{\text{OBF}}$ (Port C7) goes low when the output buffer is full. Either of these will generate an interrupt to the slave CPU to indicate that the 8255 needs service. These two signals may also be taken to other input ports of the slave, so that it can determine which kind of service is needed.



Interprocessor Communication Using 8255

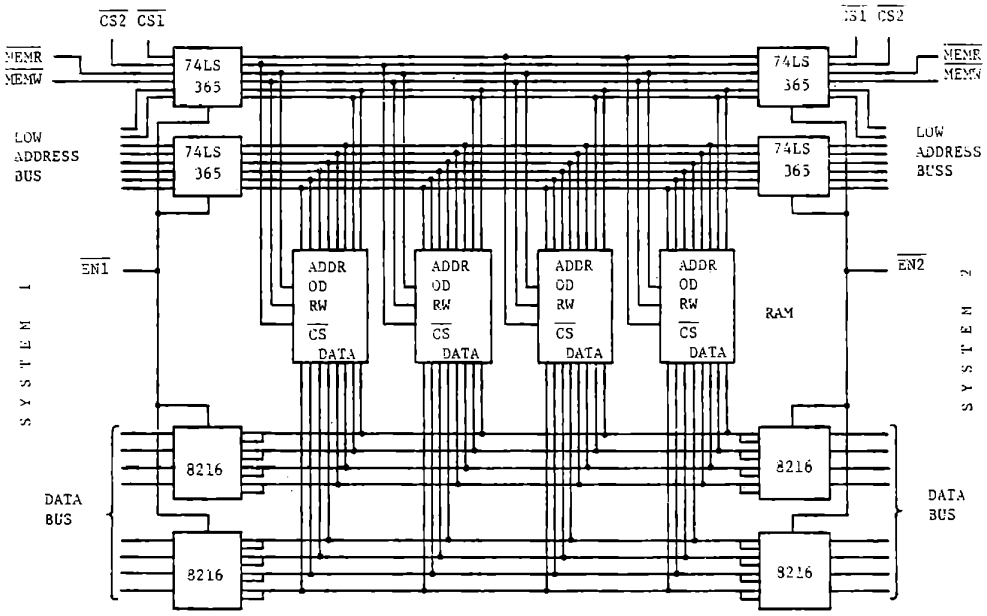
Figure 9-4

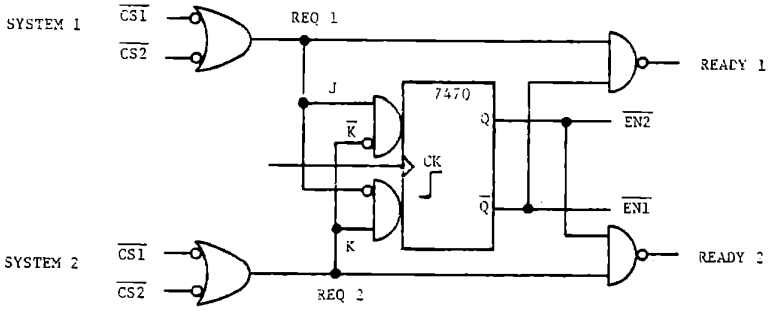
9.1.2.2 Direct Memory Access Interface

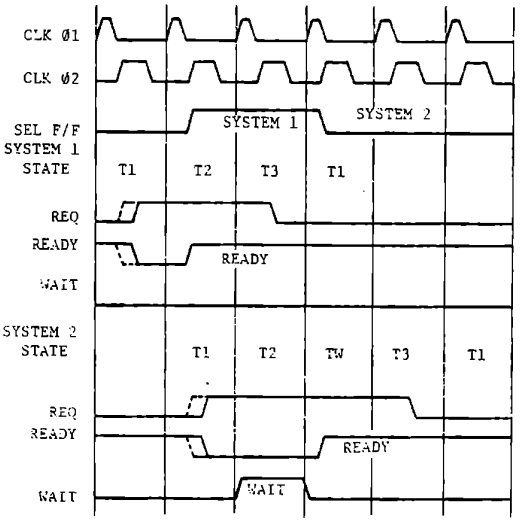
Clearly a DMA channel can be established between two processors. It may be handled by I/O ports with one processor given direct memory access to the other processor, or there may be separate hardware to operate DMA to both processors. This subject will not be covered further since DMA has been extensively discussed.

9.1.2.3 Shared Memory

A powerful but somewhat expensive technique for interfacing two processors is shown in Figure 9-5. Some part of memory is fully accessible to both processors, and either can address it at any time. As the figure shows, ten logic chips are needed to share 512 bytes (four chips) of RAM. The interesting point is the ready access each processor has to the data: it is simply addressed like any other part of memory. The timing diagram in Figure 9-5 shows what happens if both processors address the memory at the same time. Whoever gets there first has immediate access, while the other must enter a WAIT state for one clock period. If the first processor uses the memory for two consecutive reads or writes (with an INR M or SHLD instruction, for instance), the other must wait for two machine cycles. It is guaranteed access within one full instruction cycle, however, unless the other processor is executing a program stored in the shared memory. (That operation is not unreasonable. A master CPU might pass some lengthy task to a slave by loading a program into the shared memory).







Logic and Timing for Shared Memory

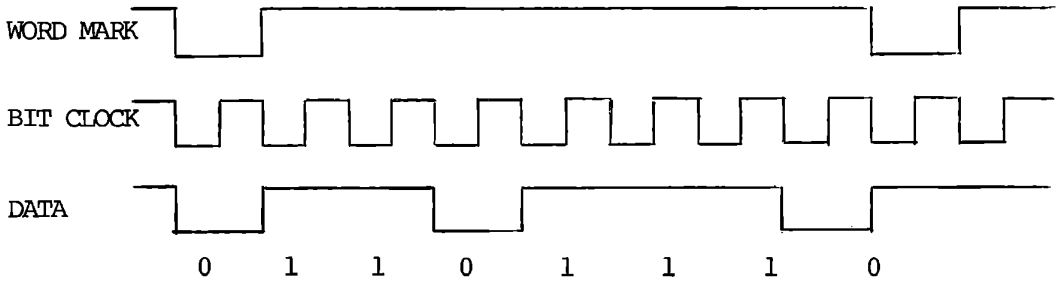
Figure 9-5

The shared memory is accessible to both processors, but it is vitally important that one does not alter data that the other is using, or unforeseen results are likely to occur. A typical convention is to reserve certain locations for flags and addresses to be passed between the two processors. System 1 writes (in a reserved location) a byte that indicates: "I have stored a message for you to process". The next two bytes identify the starting location of the message, and another two bytes indicate its length. Once having written these data, System 1 must not alter the data until System 2 responds by writing (in the same reserved location) a byte that indicates "I have finished processing this message". Thereafter System 2 must not alter the data in that message until it has been acknowledged by System 1.

In general one of the two systems must be responsible for allocating storage areas, or else each system must have some portion of the shared memory allocated to it. In a typical system, however, only one of the processors initiates messages, while the other always responds. In this case the initiator would generally be responsible for control of memory allocation.

9.2 SERIAL INPUT/OUTPUT

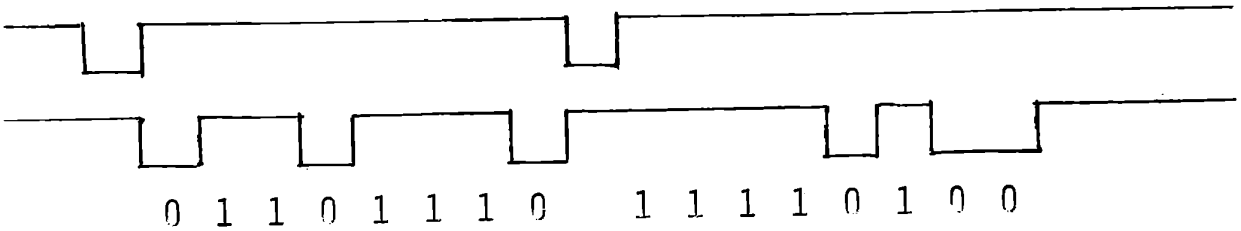
We can attach a meaning to the time of arrival of a data bit, just as we attach a meaning to its position in a binary number. To communicate an eight bit number from one machine to another, the sender outputs a discrete signal on one bit of a data port, thereafter sending successive bits at fixed time intervals. In the early days of computers it was common to send a data signal and two timing signals as discrete outputs.



9.2.1 Signal Coding

These signals are easy to generate and interpret. The sender switches the clock signal at some convenient time interval. Each time it is switched low, a new data bit is sent on a separate line. The receiver observes the clock and reads the data bit when the clock switches high. The first bit of each word is accompanied by a word mark. This delineates characters so that if an occasional bit is lost the entire message will not be garbled.

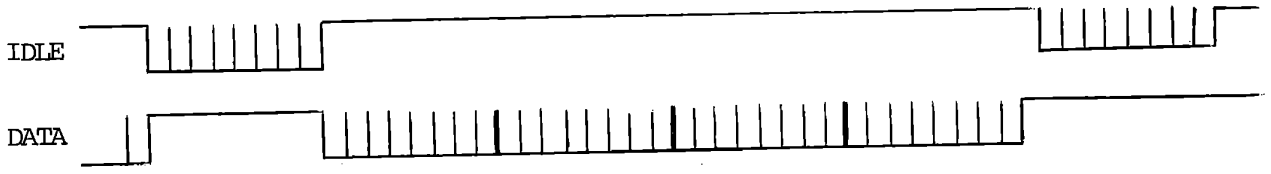
This scheme is simple, but transmitting it over long distances is extravagant as the timing signals carry very little information. If both transmitter and receiver have accurate timing sources, the bit clock is unnecessary. The receiver can recreate it, starting from the edge of the word mark. There are several ways of transmitting the word mark on the same wires with the data, thereby greatly reducing the cost.



We can put time intervals between words on the data line and fit the word marks into the intervals. If they can be distinguished from the data bits (by a narrower or wider pulse, or a different frequency, for instance) they will still serve the same function.

9.2.2 Synchronous Communication

A technique which is in common use is to send word marks only infrequently, maintaining a well synchronized clock over a long message. The word mark is now transmitted not as a single pulse for each word, but as a special, recognizable pattern called an Idle character.



This is merged into the normal data stream as though it were part of the message. It fulfills the role of a word mark in controlling synchronization of the bit clock and in marking the boundary of a character. When the receiver is seeking synchronization, it collects eight bits and compares the pattern with that of the known idle. If the pattern is wrong it discards the oldest bit and shifts in the next. This continues until the idle pattern is recognized, indicating that synchronization has been achieved and communication can begin. It is common in such systems to have at least some degree of reverse communication or feedback from the receiver to the sender, which is used to say "OK" or "HELP". This is called a supervisory channel and is only used to operate the communication system, not to transmit messages.

This method is referred to as "synchronous communication" because of the requirement for continuously synchronized send and receive signals. After the initial period of seeking synchronization, the receiver stays synchronized by observing signal transitions in the data stream. Its crystal clock is able to maintain sync even if long strings of data are all ones or all zeros, of if the signal is temporarily lost. Thus all the signals on the communications line are part of the message being sent. If there is a break in the message, the sender must fill the spaces with idle characters so that the time from the beginning of one word to the beginning of the next is always exactly one word time.

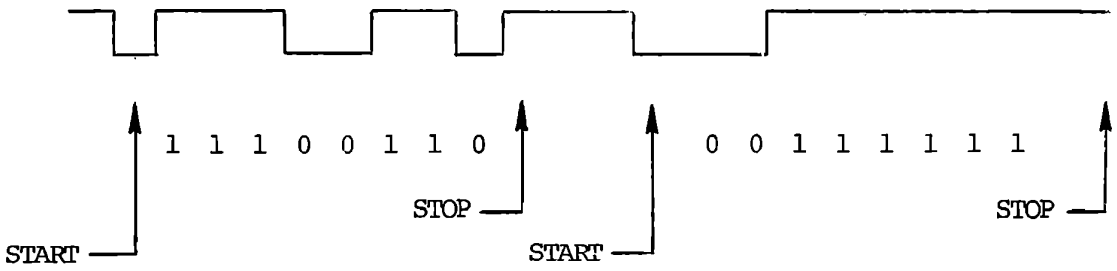
9.2.3 Asynchronous Communication

An alternative method is especially suited to devices such as the teletype, whose characters are transmitted and received asynchronously. There may be long pauses between characters, but occasionally one character will quickly follow another.

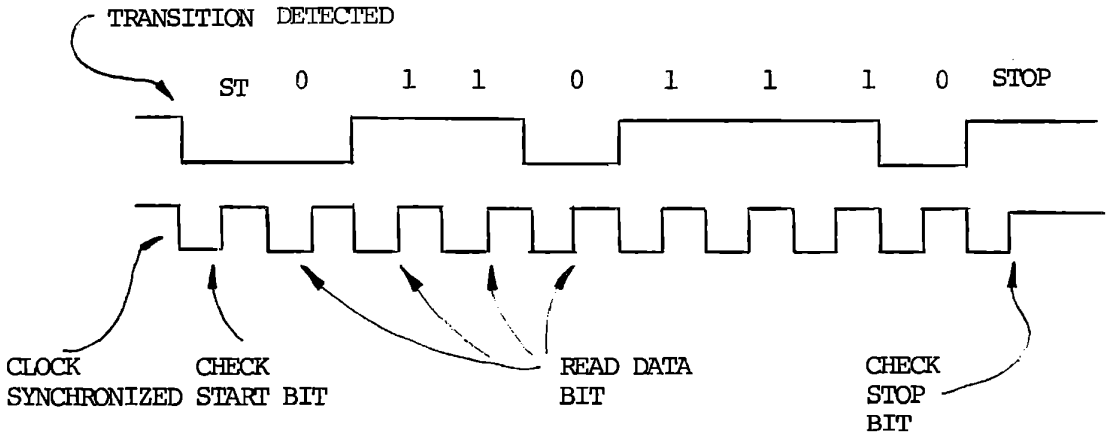
The transmission rate for a teletype is usually 10 characters per second or approximately 120 words per minute (a very fast typing speed). The same signal format has been adopted for faster electronic communication devices.

DATA FORMAT

In asynchronous communication each character is independent and carries its own word mark. The adopted convention is for each data character to be preceded by a zero, followed by one or more bit-times (intervals) of the "one" signal.



After some period of time with no data, (i.e. constant "one" signals) the receiver will see a transition to zero. This signals the start of a character, and the receiver synchronizes its clock.



One half bit-time later the receiver checks the start bit. If it is not zero, an error has been made. Thereafter the receiver accepts eight bits, reading them at one bit-time intervals, then tests the stop bit to see that it is a "one". Now the receiver waits until another transition to zero marks the start of next character.

Within a character the data are transmitted least significant bit first, so the sequence is:

Start Bit
Bit 0
--
--
Bit 7
Stop Bit
(optional addiitional stop bits)

This data format has been adopted for asynchronous communication by the American National Standards Institute and by CCITT. The data content is also coded in a standardized form. These standards were promulgated by the American Standards Committee on Information Interchange (ASCII).

9.3 ASYNCHRONOUS TRANSMITTING AND RECEIVING

A special purpose communications device, the 8251, is available as a peripheral to the 8080. This is a "Universal Synchronous - Asynchronous Receiver - Transmitter" (UART). It is a very capable device, and in any busy system its use is well justified. Often, however, the microprocessor has little enough to do that it can readily handle serial communications by "bit banging" - processing and timing each bit under program control. In this and the next exercises we will program the MTS to send and receive in the asynchronous format.

The tape cassette modem circuit on the MTS provides for recording and reading serial data. Connect an audio cable from the IN connector to the earphone connector of a cassette recorder, and from the OUT connector of the MTS to the microphone or auxiliary connection of the recorder. The programs to be developed will then record and read tapes in the same format used by the monitor read and record program.

The output from the writing program must be at PORT0C0 (the least significant bit of port address 02) and the input is received at PORT0B0 (the least significant bit of port address 01).

9.3.1 Serial Transmission Exercise

For practice in handling interrupts, we will write an interrupt service routine that will transmit one bit each time it is entered. If a timer were available we could use it to generate the interrupt at the appropriate intervals. In this program we will call a timing loop subroutine to generate the time delay, and use a programmed call to the interrupt service at the end of the delay period.

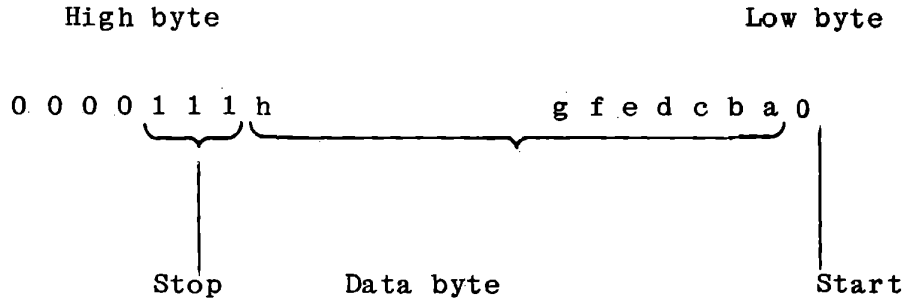
The main program loads successive bytes from memory and passes them to the interrupt service routine. Since in a real interrupt system the main program has no knowledge of when an interrupt occurs, the data must be passed through some fixed location in memory, and interrupt service must indicate when it has finished with one byte and needs another.

The interrupt service routine must send start and stop bits as well as data bits. A simple way of handling this is to store the full data pattern in 12 bits. This will be shifted out, one bit at a time, by the interrupt service routine. When the pattern is empty (0000), both the interrupt service routine and the main program will know that the character has been sent. No more bits will be sent until the main program stores a new pattern.

This page intentionally left blank.

9.3.2 Character Data Pattern

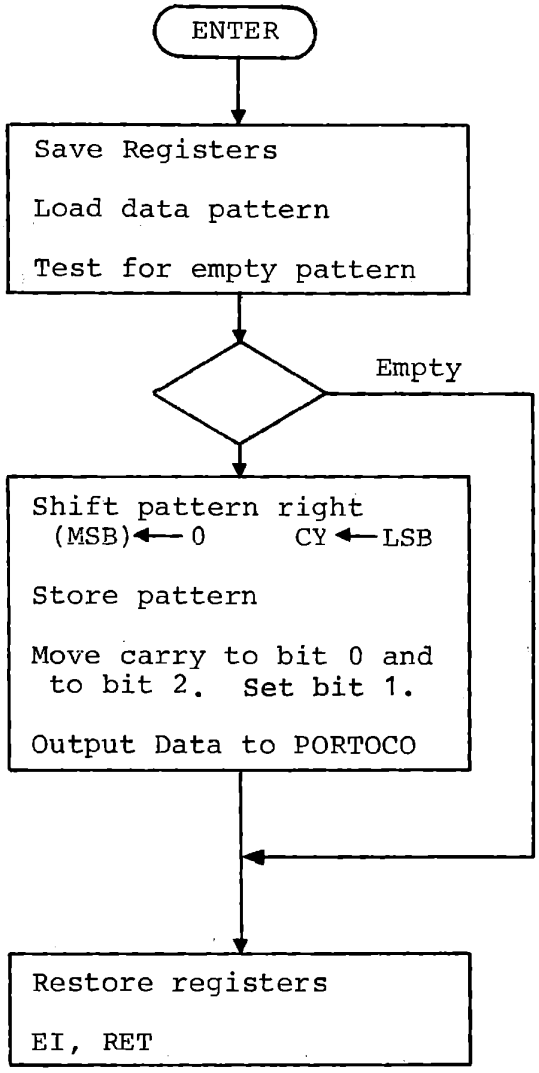
The data pattern is stored as two bytes, as follows:



This is loaded and stored by the main program by:

```
MOV  L,A          Data byte
MVI  H,07         Stop bits
DAD  H            Enters start bit
```

Interrupt service will shift the data pattern right and output each successive bit until the data pattern is empty (all zero).



Serial Data Transmit Interrupt Service Routine

Figure 9-6

9.3.3 Interrupt Service Routine

The interrupt service routine is shown in Figure 9-6. Write this routine and test it as described in Section 8.6.2. Note that the output is to be PORT0C0, which is connected to the tape cassette modem. Also copy the data bit to PORT0C2, which controls the Carry Indicator.

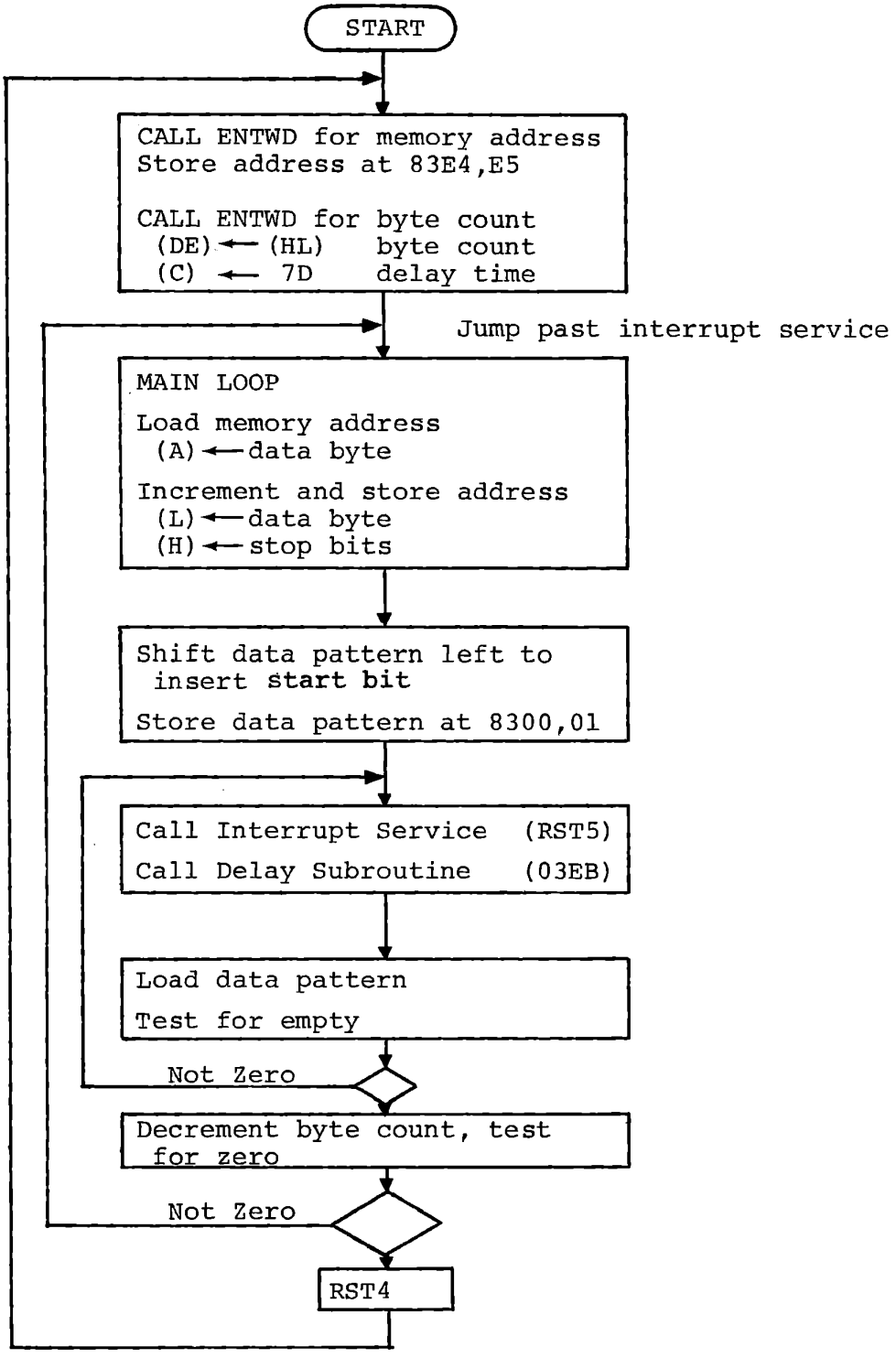
```
9F      SBB A           Copy CY to all bits
E6      ANI 05          Mask for bits 0,2
05
F6      ORI 02          Set Bit 1 to enable monitor
02
D3      OUT PORT0C
02
```

This will cause the carry indicator to display each bit as it is transmitted.

We will use RST5 to call the interrupt service routine. You can store a jump address at 83EC,ED.

```
LXI  H          (interrupt service address)
SHLD            83EC
```

Since the monitor automatically stores the address 8228 as a jump address for RST5, you can avoid the need for these instructions by locating your interrupt service routine at 8228.



Serial Transmit - Main

Figure 9-7

9.3.4 Main Program

After testing your interrupt service routine, write the main program shown in Figure 9-7. This will not fit in the space from 8200 - 8227, and the following locations are used by interrupt service. It is convenient to place initialization functions (in this case the two data entry calls) before interrupt service, and the main loop beyond it, say at 8250.

We can save both effort and program space by sharing a delay subroutine between this transmit program and the receive program of the next exercise. The monitor contains a suitable delay subroutine, which includes data input:

DELYT (03EB)

Delays for a time set by Register C.
106 + 148(C) system clocks, including CALL
All registers preserved. Zero Set.
Returns Carry = input bit from PORT0B0.

Load Register C with 7D to make this delay compatible with the tape cassette program of the monitor.

DATA FORMAT

The solution given in Figure 9-8 also uses another monitor subroutine:

SHLRT (022D)

This tests the content of (HL) and then shifts (HL) right one bit. It returns the least significant bit of (HC) in Carry. Zero is set if the entry value of (HL) was 0000.

You can use your program to record a tape, which the monitor tape reading program can then read back. In the next exercise we will create a serial input program which will read tapes recorded by this program or by the monitor.

SERIAL TRANSMIT - DATA ENTRY

ADDR	CODE																
8 20 0	C	D		C	A	L	L		E	N	T	W	D	Get starting address
1	4	6												
2	0	3												
3	2	2		S	H	L	D		8	3	E	4		Store starting address
4	E	4												
5	8	3												
6	C	D		C	A	L	L		E	N	T	W	D	Get byte count
7	4	6												
8	0	3												
9	E	B		X	C	H	G							(DE) ← byte count
A	0	E		M	V	I		C	,	7	D			(C) ← delay count
B	7	D												
C	C	3		J	M	P			8	2	5	0		Jump past interrupt service
D	5	0												
E	8	2												
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 9-8a

SERIAL TRANSMIT - INTERRUPT SERVICE

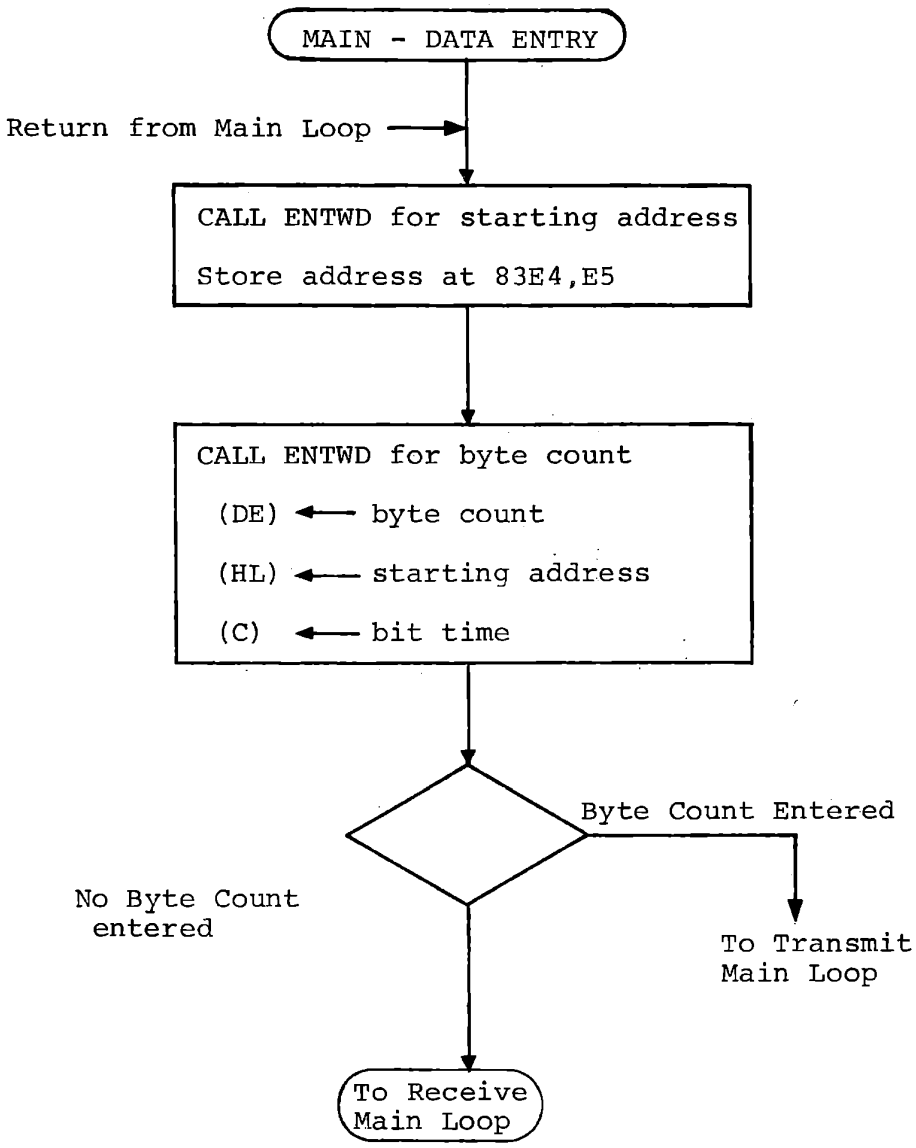
ADDR		CODE														
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
822	8	F	3		DI											NOP during test
	9	F	5		PUSH				PSW							Save registers
	A	E	5		PUSH				H							(B,C,D,E not used)
	B	2	A		LHLD				8300							Load data pattern
	C	0	0													
	D	8	3													
	E	C	D		CALL				SHLRT							Test for empty and shift
	F	2	D													
823	0	0	2													
	1	C	A		JZ				823C							No output if empty
	2	3	C													
	3	8	2													
	4	2	2		SHLD				8300							Store pattern
	5	0	0													
	6	8	3													
	7	9	F		SBB				A							Carry into all bits
	8	E	6		ANI				05							Mask for bits 0 and 2
	9	0	5													
	A	F	6		ORI				02							Set bit 1 to enable monitor
	B	0	2													
	C	D	3		OUT				PORTOC							Output to modem and CY indicator
	D	0	2													
	E	E	1		POP				H							Exit
	F	F	1		POP				PSW							
3	0	F	B		EI											
	1	C	9		RET											
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 9-8b

SERIAL TRANSMIT - MAIN LOOP

ADDR	CODE													Comments
8 25 0	2A		L	H	L	D		8	3	E	4			Load memory
1	E4													address
2	83													
3	7E		M	O	V		A	,	M					Data byte
4	23		I	N	X		H							Increment and
5	22		S	H	L	D		8	3	E	4			store address
6	E4													
7	83													
8	6F		M	O	V		L	,	A					Data byte
9	26		M	V	I		H	,	07					Stop bits
A	07													
B	29		D	A	D		H							Enter start bit
C	22		S	H	L	D		8	3	0	0			Store pattern for
D	00													interrupt service
E	83													
F	00		N	O	P									
8 26 0	EF		R	S	T	5								Call interrupt service
1	CD		C	A	L	L		D	E	L	Y	T		Delay one bit time
2	EB													
3	03													
4	2A		L	H	L	D		8	3	0	0			Load data pattern
5	00													
6	83													
7	7C		M	O	V		A	,	H					Test for empty
8	B5		O	R	A		L							
9	C2		J	N	Z			8	2	6	0			Send next bit
A	60													unless empty
B	82													
C	1B		D	C	X		D							Decrement byte count
D	7A		M	O	V		A	,	D					Test for zero
E	B3		O	R	A		E							
F	C2		J	N	Z			8	2	5	0			Get next data byte
8 27 0	50													unless finished
1	82													
2	E7		R	S	T	4								Enter monitor
3	C3		J	M	P		8	2	0	0				at completion
4	00													then back to start
5	82													
6														
7														
8														

Figure 9-8c



Transmit - Receive Data Entry

Figure 9-9

9.4 ASYNCHRONOUS RECEIVING

Modify the keyboard data entry section of the previous main program to select between recording and receiving according to whether a byte count is entered at the second call to ENTWD (Note that ENTWD returns the Zero flag set if no hex keys are entered.) Figure 9-9 shows this program segment. It will allow you to record data from memory to tape, then stop and rewind the tape and read it back.

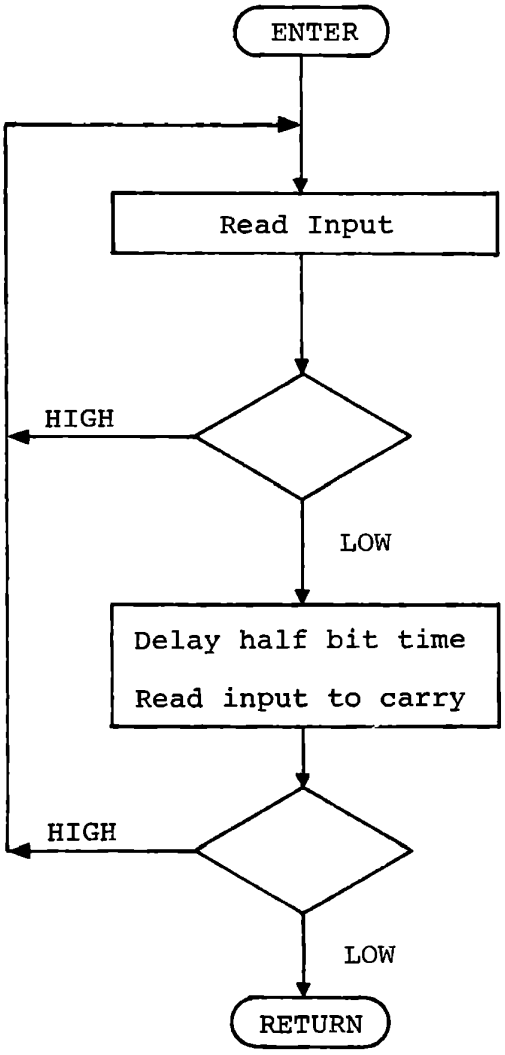
It is somewhat more complex to receive than to transmit in the serial format because two input functions are needed:

a) Wait for a start bit

b) Receive successive data bits

The start bit is initially detected when the received data bit changes from one to zero -- from a stop or no data condition to a start bit. This must be recognized promptly in order to obtain synchronization between the incoming data and the receiver's timing device or programmed timing loop.

To use interrupts for receiving data we would need to generate an interrupt at the leading edge of the start bit to obtain synchronization, delay one half bit time to the middle of the start bit, and then delay one bit time to the middle of each successive bit until the entire character has been received. In this exercise we will use a loop that repeatedly tests the input until it becomes low. Then a timing loop will be used to accept successive bits.

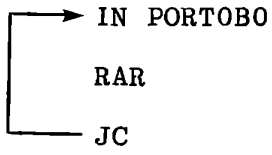


Wait for Start Bit

Figure 9-10

9.4.1 Wait for Start Bit

The subroutine shown in Figure 9-10 repeatedly reads the input from PORTOBO and waits for a start bit:



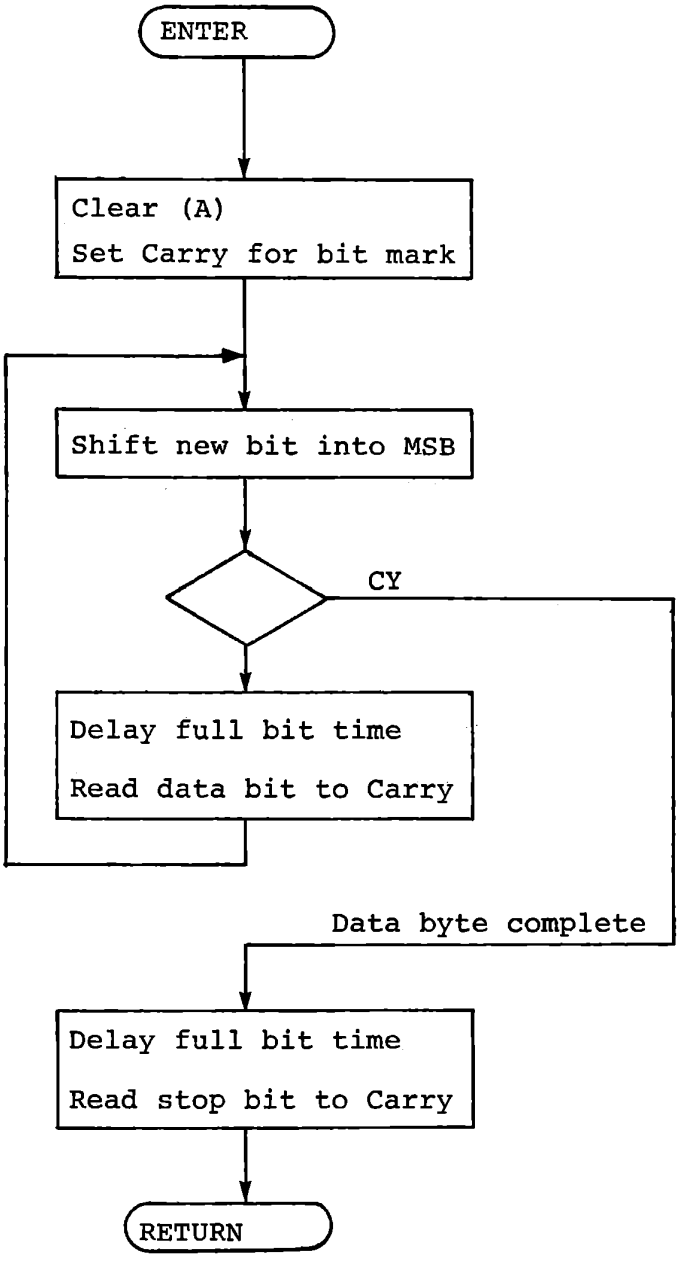
The three instructions above take 24 clock times or about 12 microseconds. This gives adequate precision for detecting the edge of the start bit at low data rates.

Now we delay one half bit time and check to be sure that the input is still low. This avoids synchronizing the receive timing to a momentary "glitch". There is good confidence that if the falling edge has been seen and the signal is low half a bit time later, the signal received is a legitimate start bit.

Recall from Section 9.3.4 that the delay subroutine DELYT delays for one bit time and returns with the input bit in Carry. An alternate entry delays half a bit time:

```
                DELYC           (03F0)
    Delays for a time set by Register C:
    48 + 74 (C) system clocks, including CALL.
    All registers preserved.  Zero set.
    Returns Carry = input bit from PORTOBO.
```

Using the DELYC entry avoids any need for changing the entry value in (C) between the half bit time and full bit time delays.



Receive Data Bits

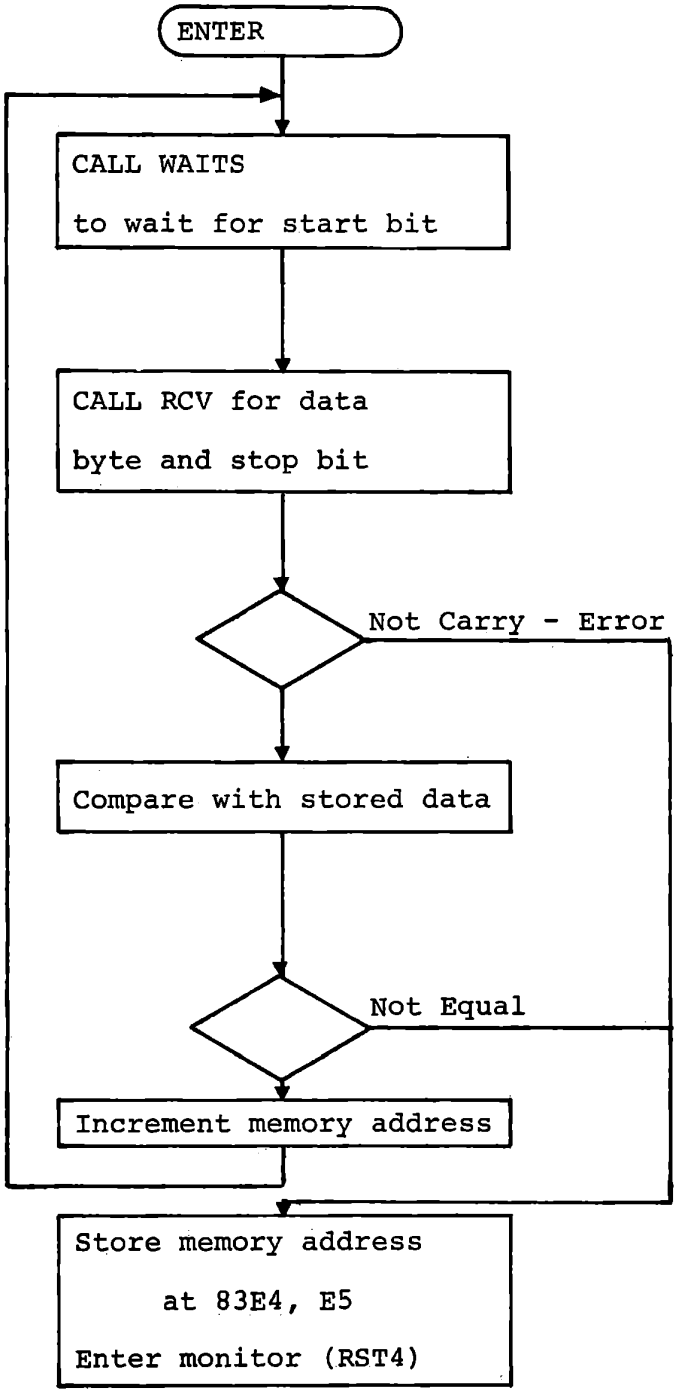
Figure 9-11

9.4.2 Receive Data Bits

After the start bit has been recognized we must receive and store eight successive data bits at full bit time intervals. Figure 9-11 shows a suitable process. Register A is used both for saving the received data and for counting bits. The start bit is marked by shifting a one into Bit 7. When that bit is shifted out from Bit 0 we know that the eight data bits have been shifted in.

It is possible to misread the start bit and synchronize improperly. If this happens, the data received will be garbage. Some protection against this is obtained by testing to make sure that a proper stop bit is received. If characters are received at lengthy intervals, as from a manually operated keyboard, this test has little, if any, value. When a continuous string of characters is being received, however, any timing error is likely to propagate to the following characters. Now synchronization will occur, not on a start bit but when a one- zero transition occurs in the data. Fairly soon this is likely to result in a zero data bit appearing when the program expects a stop bit. The stop bit test detects such an error.

The subroutine of Figure 9-11 returns with the data byte in (A) and the stop bit in Carry. Thus, if Carry is not set at return, an error has been detected. The Zero flag is set by DELYT, and so returned by the receive subroutine.



Receive Main Loop

Figure 9-12

9.4.3 Receive Main Loop

In the previous exercise we recorded a program on a tape cassette. Now we will read it back. Rather than storing the program, however, we will compare it with the program already in memory. Thus, if the recording or receiving program is bad, or if the two are not compatible, we will not have destroyed the programs and will quickly detect the problem.

If an error is detected, either by lack of a stop bit or by disagreement between the received data and the previously stored data, store the memory address and enter the monitor by RST4, as shown in Figure 9-12.

The monitor uses the content of 83E4, 83E5 as the memory address, so MEM will display the location and the data byte that was recorded or reread incorrectly. REG A displays the received data byte.

Since the delay subroutine returns with Zero set, if the error was detected by a missing stop bit the Zero indicator will be on. If the error was detected by the comparison, Zero will be off.

As long as data bytes or a continuous "one" input signal are received, there is no exit from the receive loop. When the tape reaches the place where you stopped it during recording, an error is bound to occur. MEM will display the address beyond the last byte that was recorded.

TRANSMIT - RECEIVE DATA ENTRY

ADDR	CODE													
8200	CD			C	A	L	L		E	N	T	W	D	Get starting address
1	46													
2	03													
3	22			S	H	L	D		8	3	E	4		Store starting address
4	E4													
5	83													
6	CD			C	A	L	L		E	N	T	W	D	Get byte count for transmit
7	46													
8	03													(Returns zero if no data)
9	EB			X	C	H	G							(DE) ← byte count
A	2A			L	H	L	D		8	3	E	4		(HL) ← start address
B	E4													
C	83													
D	01			L	X	I		B,	0	0	7	D		
E	7D													(C) ← Bit time
F	00													
8210	CA			J	Z			8	2	8	0			If no byte count
1	80													jump to receive
2	82													
3	C3			J	M	P		8	2	5	0			
4	50													
5	82													
6														
7														
8	NO		TE	:		A	L	S	O		R	E	Q	UIRES
9				I	N	T	E	R	R	U	P	T		SERVICE
A				A	N	D		T	R	A	N	S	M	IT LOOP
B				F	I	G	U	R	E	S		9	-	8b, 9-8c
C														
D														
E														
F														
80														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 9-13a

RECEIVE MAIN LOOP

ADDR		CODE														
8 28	0	CD		CALL			WAITS									Wait for start bit
	1	A0														
	2	82														
	3	CD		CALL			RCV									Accept data byte
	4	B0														
	5	82														
	6	D2		JNC			8291									Error if no stop bit
	7	91														
	8	82														
	9	BE		CMP		M										Compare data
	A	C2		JNZ			8291									Error if not equal
	B	91														
	C	82														
	D	23		INX		H										Next address
	E	C3		JMP			8280									Go to receive next byte
	F	80														
8 29	0	82														
829	1	22		SHLD			MADDR									Error
	2	E4														Store address
	3	83														
	4	E7		RST		4										Enter monitor
	5	C3		JMP			8200									Back to start
	6	00														
	7	82														
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 9-13b

WAIT FOR START BIT (WAITS)

ADDR		CODE													
8 2A	0	DB		IN			PO	RT	0B						
	1	01													
	2	1F		RAR											
	3	DA		JC			82A0								
	4	A0													
	5	82													
	6	CD		CALL			DELYC								
	7	EE													
	8	03													
	9	DA		JC			82A0								
	A	A0													
	B	82													
	C	C9		RET											
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 9-13c

RECEIVE DATA (RCV)

ADDR		CODE													
8	2B 0	A	F		X	R	A		A						Set mark to
	1	3	7		S	T	C								indicate last bit
	82B 2	1	F		R	A	R								Shift bit into (A)
	3	D	A		J	C				8	2	B	C		At end go to
	4	B	C												get stop bit
	5	8	2												
	6	C	D		C	A	L	L		D	E	L	Y	T	Delay and read
	7	E	B												data bit
	8	0	3												
	9	C	3		J	M	P			8	2	B	2		Loop for
	A	B	2												next bit
	B	8	2												
	82B C	C	D		C	A	L	L		D	E	L	Y	T	Delay and read
	D	E	B												stop bit
	E	0	3												
	F	C	9		R	E	T								
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 9-13d

9.5 MONITOR TAPE PROGRAMS AND SUBROUTINES

The modules you have developed in the preceding exercises are available in different form in the monitor. We have already used DELYT, DELYC, and SHLRT. Before defining the other subroutines we will describe the use of the monitor tape recording functions.

9.5.1 Tape Recording Program

The normal monitor function of recording a program on the tape is accomplished by:

Set toggle switch to AUTO.
Turn recorder on.
RESET
ADDR (starting address) MEM
ADDR (stopping address) BRK
ADDR 0371 RUN

The tape recording program uses the breakpoint system to terminate the transmission. It is important that no other breakpoints be entered while this is running.

While the program is running, the display is disabled. When the data have been recorded, the display will show: 0382 CD. Wait about two seconds before turning the recorder off. Note that the content of the stopping address is <u>not</u> recorded on the tape. Instead, an error check character is recorded. Therefore, the stopping address must be the next location past the end of the program you want to record. When you read the program back, the error check character will be

written to this location. When you have finished recording you can observe this character by pressing BRK. It will display the stopping address, the BP. symbol, and the error check character.

The tape is recorded at a data rate of 110 baud. This means 110 bit intervals per second. Since the tape program sends 12 bits per character, the data rate is 9.17 bytes per second. Each page (256 bytes) of memory recorded takes 28 seconds. If you wish, you may increase the data rate by loading a different delay count in Register C, and enter the program at 0373. A value of 2D in Register C generates 300 baud or 25 characters per second.

If you operate this program in STEP mode the tape will be recorded correctly but the final display of the breakpoint count will not show the recorded check character accurately.

9.5.2 Tape Reading Program

To read back and store in memory a program that has been recorded on tape, listen to the tape until you hear the steady tone that was recorded before you started the recording program. Stop the tape, connect the recorder earphone output to the modem input, set the toggle switch to AUTO, and enter:

```
ADDR (starting address) MEM
ADDR 03AE
Start the tape player
RUN
```

DATA FORMAT

It is important that the recorded tape have a few seconds of continuous tone before the data starts, to give you time enough to hear it, stop the recorder, and press RUN after starting it again. The tape must be giving the continuous tone when you press RUN. If data has already started before RUN, an error will be detected and the display will show Err.

While the tape is being read the display will be disabled. At the end of the tape the display will show 03CF C5. Register pair HL will contain the stopping address. ADDR 8/H MEM will display the error check character recorded when the tape was made. The reading program has also calculated the error check character, and if the two agree, Register A will contain 00. If it does not, an error has occurred.

9.5.3 Error Check Character (LRC)

The error check character, also called "longitudinal redundancy check" (LRC) character, is the exclusive OR of all data bytes sent or received. The reading program receives each data byte from a subroutine, stores it, and calculates the LRC.

```
MOV M, A        Store data byte
INX H           Address next byte
XRA E           Calculate LRC
MOV E, A        Save LRC
```

The last character recorded is the LRC of all preceding characters. This character is included in the LRC calculated in receiving, which therefore must be zero if no errors have occurred.

9.6 MONITOR SEND AND RECEIVE SUBROUTINES

The subroutine used by the monitor for sending and receiving are also available to your programs. We have already defined the delay subroutines DELYT and DELYC (see Sections 9.3.4 and 9.4.1). Additional subroutines are defined here. In the exercise of Section 9.6.5 we will use these monitor subroutines to record and read cassette tapes.

9.6.1 SOTBT (0382)

This monitor subroutine combines the testing and shifting function for the data pattern with the process of loading the next byte when the pattern is empty. You can call this subroutine from an interrupt service routine or at intervals set by a programmed timing loop. It must be entered with the data pattern in (HL). It returns with the data pattern shifted right, and the data bit to be output in bit 0 of Register A. Carry and Zero flags are both cleared. When the pattern is empty SOTBT loads the memory address from (83E4, 83E5) and tests whether this address has been entered as a breakpoint. If not, it loads the content of this address to the data pattern, stores the next memory address, and returns a start bit. This is marked by Zero, Not Carry.

SOTBT also calculates an LRC character (see Section 9.5.3) for the message being transmitted. When the breakpoint address is encountered, SOTBT loads the LRC instead of the data byte into the data pattern. When the LRC has been transmitted, SOTBT returns Carry, Not Zero, and (A) = FF to indicate the end of transmission.

DATA FORMAT

The starting address must be stored at (83E4, 83E5). The stopping address may be entered as a breakpoint either by monitor command or by calling a monitor subroutine. Section 9.6.2 describes entry and removal of breakpoints under program control.

SOTBT Data Entry and Return

Enter with

(HL) = data pattern
(83DE-83E3) loaded by entering stopping address as a breakpoint.
(83E4-83E5) memory address for data byte.

Return

(a) If the data pattern (HL) was not empty (zero) at entry, SOTBT returns:
(A) 0 0 0 0 0 1 0 X where X is the data bit to be output.
(HL) shifted right
Not Carry, Not Zero.

(b) If the data pattern (HL) was empty at entry, and more data remains to be sent, SOTBT returns:
(A) = 0 0 0 0 0 0 0 0 to send start bit
(L) data byte
(H) = 0 0 0 0 0 1 1 1 to send 3 stop bits
Not Carry, Zero

(c) If the data pattern (HL) was empty and the stopping address has been reached, return as in (b) except that
(L) = LRC for message
(83E0) changed to mark end of message

(d) After the LRC has been transmitted SOTBT returns
(A) = FF
(L) = FF
(H) = 07
Carry, Not Zero

In all cases SOTBT preserves (BC).

Note that during data transmission the data returned in (A) by SOTBT, which will be written to PORTOC, disables monitor interrupts. At the end of transmission SOTBT returns (A) = FF which enables monitor interrupts when written to PORTOC. If the AUTO/STEP toggle switch is set to STEP a RST7 interrupt will occur.

9.6.2 Program Entry and Removal of Breakpoints

The breakpoint system can be controlled by user programs. This might be used in debugging a complicated program, to enter a breakpoint within some subroutine only when it is called by one of several modules. Another use is to let the breakpoint system terminate a memory search, as is done in the serial output module of the monitor.

Three monitor subroutines are used in such a process. BKLOC finds a breakpoint that exists in the table. BKENT enters a breakpoint into the table, and BKRMV deletes it from the table. It is important that BKLOC be used in conjunction with the other two subroutines, and that monitor interrupts be disabled while these subroutines are in use.

Their addresses are:

BKLOC	01C3
BKENT	01A3
BKRMV	0186

DATA FORMAT

Given an address in (HL), it can be entered as a breakpoint by:

```
        DI
        XRA   A
        CALL  BKLOC
        CNC   BKENT
        EI
```

The address can be removed by:

```
        DI
        XRA   A
        CALL  BKLOC
        CC    BKRMV
        EI
```

Subroutine BKLOC finds the location in the breakpoint table of the given address, and returns Carry set if the address exists. Then it can be entered or removed. The conditional calls to BKENT and BKRMV prevent duplicating an existing breakpoint or removing a non-existing breakpoint. Since these subroutines lengthen or shorten the stack it is vital that they not be used improperly. The procedures shown above protect against stack errors. BKLOC must be entered with Carry clear.

9.6.3 Subroutine BKMEM (01D5)

This is the subroutine used by the monitor breakpoint system to test for a change in the data stored at a memory location that has been entered as a breakpoint. SOTBT marks the end of transmission of the message by changing the data stored in the breakpoint memory table (rather than that in the main memory), which allows BKMEM to detect the end of transmission.

To be effective, BKMEM must be entered with Carry cleared.

If no breakpoint data has changed, BKMEM returns:

Not Carry, Not Zero
(A) = Data byte from oldest breakpoint
(BC) = Address of oldest breakpoint
(DE) Preserved
(HL) = Address of count byte of oldest breakpoint

If breakpoint data has been changed, BKMEM returns:

Carry, Not Zero
(A) = Data byte that has changed
(BC) = Address of data byte that has changed
(DE) Preserved
(HL) = Address of count byte for breakpoint whose data has changed

9.6.4 Subroutine SINWS (03CF)

This subroutine waits for a start bit and then for successive data bits. It includes calls to the delay subroutine. SINWS returns with the received data byte in (A), stop bit in Carry, and Not Zero if a character has been received. If a long delay expires without a start bit SINWS returns with Zero set.

The delay times must be loaded to Registers C and B before the call to SINWS.

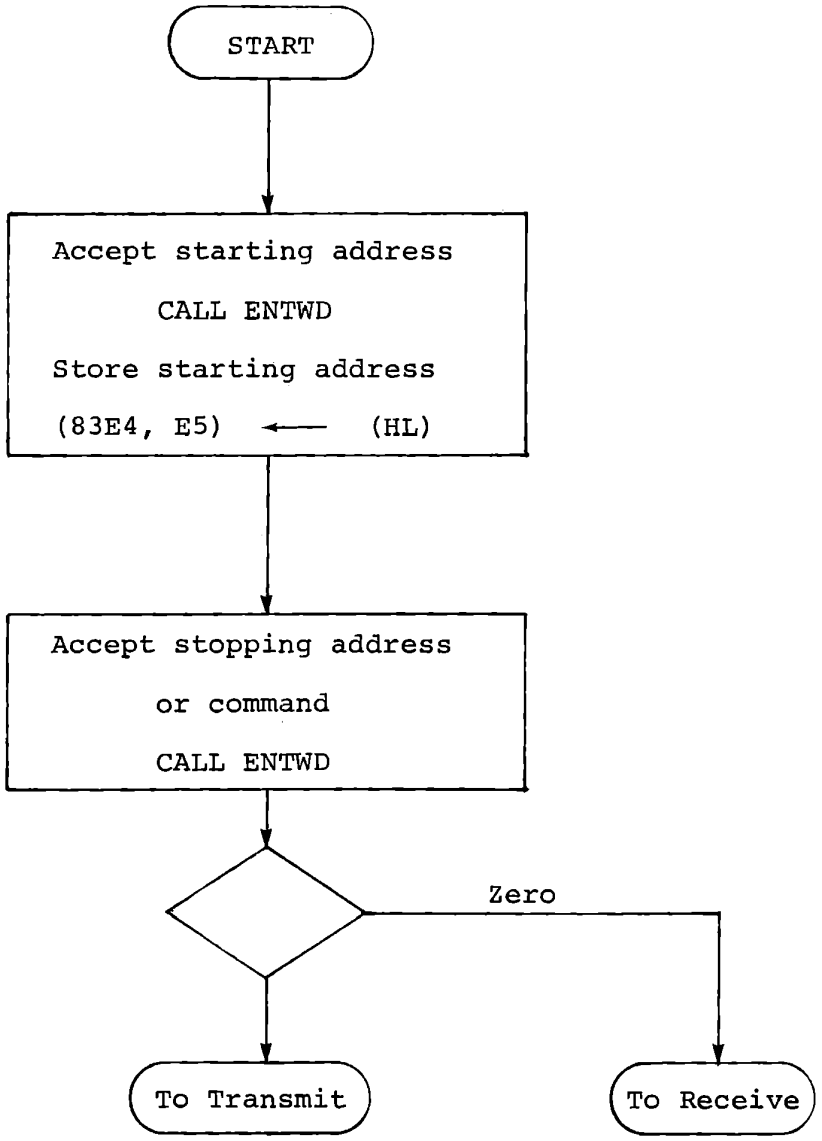
(C) = Bit time delay count

Bit time = 106 + 148 (C) system clocks

(B) = Delay to wait for start bit

Delay 9.375 milliseconds for each count in (B).

All registers except (A) are preserved.



Transmit/Receive With Monitor Subroutines

Figure 9-14

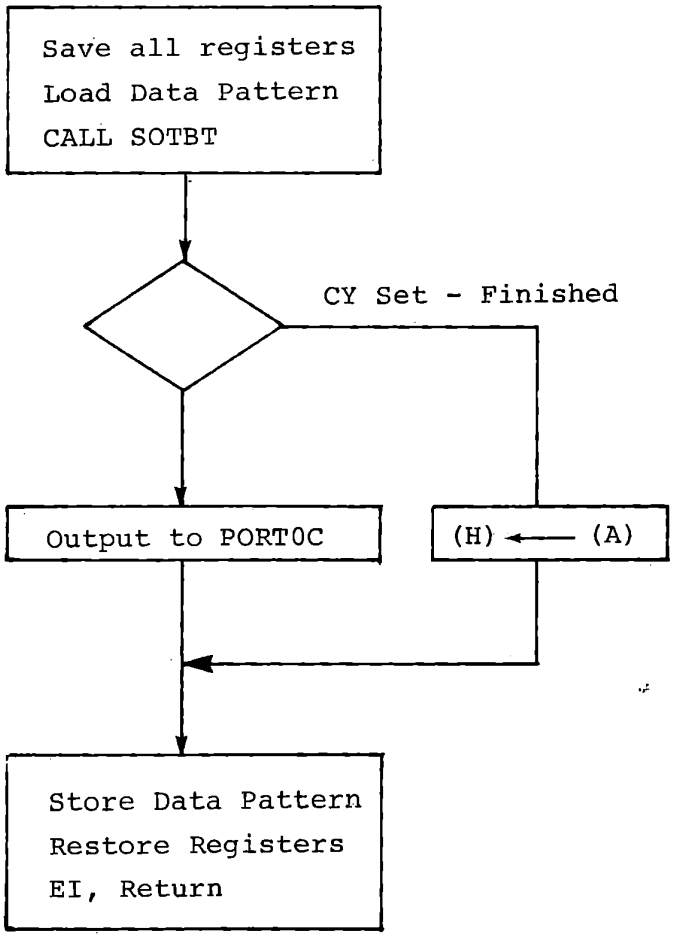
9.6.5 Transmit/Receive with Monitor Subroutines

This exercise uses the subroutine described above to record data on tape and read it back, comparing the received data with that recorded.

A starting address must be entered via ENTWD. (See Figure 9-14.) Another call to ENTWD accepts a stopping address for transmission, but if none is entered the receive function is performed. ENTWD returns Zero set after a command.

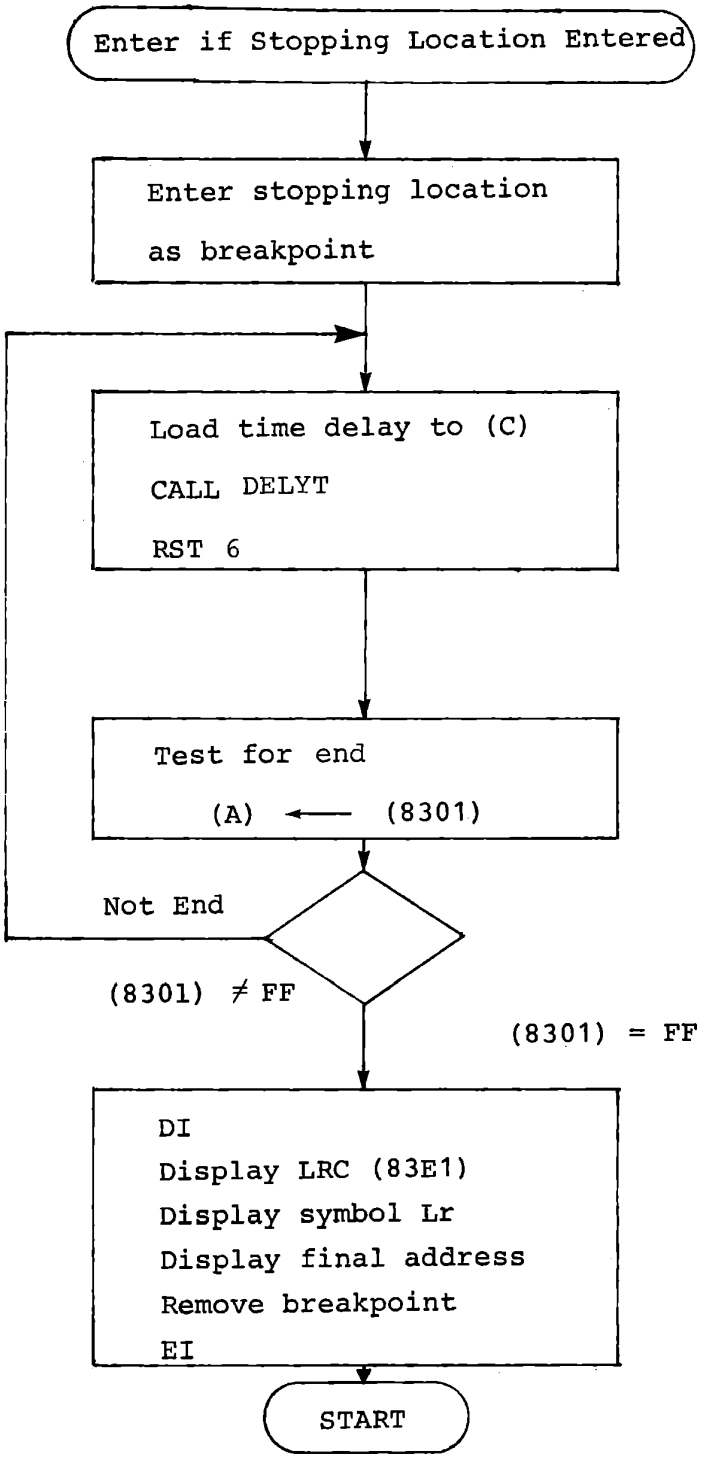
9.6.5.1 Transmission

Monitor subroutine SOTBT is called by an interrupt service routine, activated by a programmed RST6 after the time delay subroutine DELYT. At the end of the transmission the service routine stores FF as the high byte of the data pattern, indicating completion as a signal to the main loop. Interrupt service and the transmit loop are shown in Figures 9-15 and 9-16.



Transmit Interrupt Service With SOTBT

Figure 9-15



Transmit Main Loop With Breakpoint Entry

Figure 9-16

The main transmit loop uses the procedures of Section 9.6.2 to enter the stopping address as a breakpoint and remove it when transmission is finished (Figure 9-16). It recognizes the end of the message by finding FF at address 8301, which otherwise contains stop bits or zeros. The final address and the LRC generated by SOTBT are displayed, along with a symbol, Lr. Since the display subroutines enable monitor interrupts, it is necessary here to disable interrupts until the breakpoint has been removed. Otherwise the monitor will detect a data change at the breakpoint, because SOTBT changes the data in the breakpoint table to indicate that the LRC has been sent.

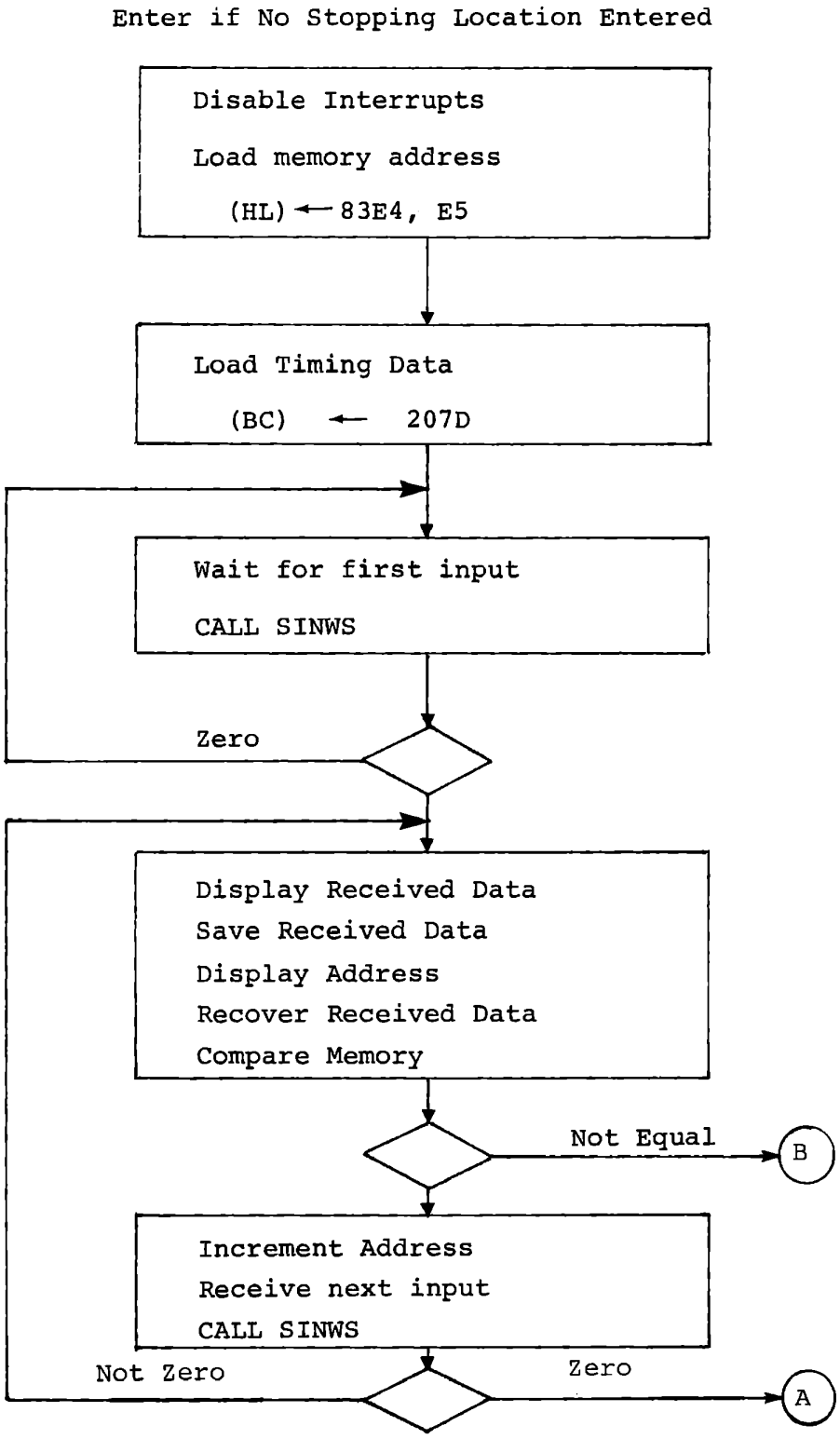
9.6.5.2 Receiving

The receive loop calls SINWS in three different places (Figure 9-17). The first call is repeated indefinitely until a data byte is received. Data bytes are compared with successive data locations in memory, and the address and data are displayed until one of two possible events terminates the operation.

a) SINWS returns Zero set, to indicate the end of the recording.

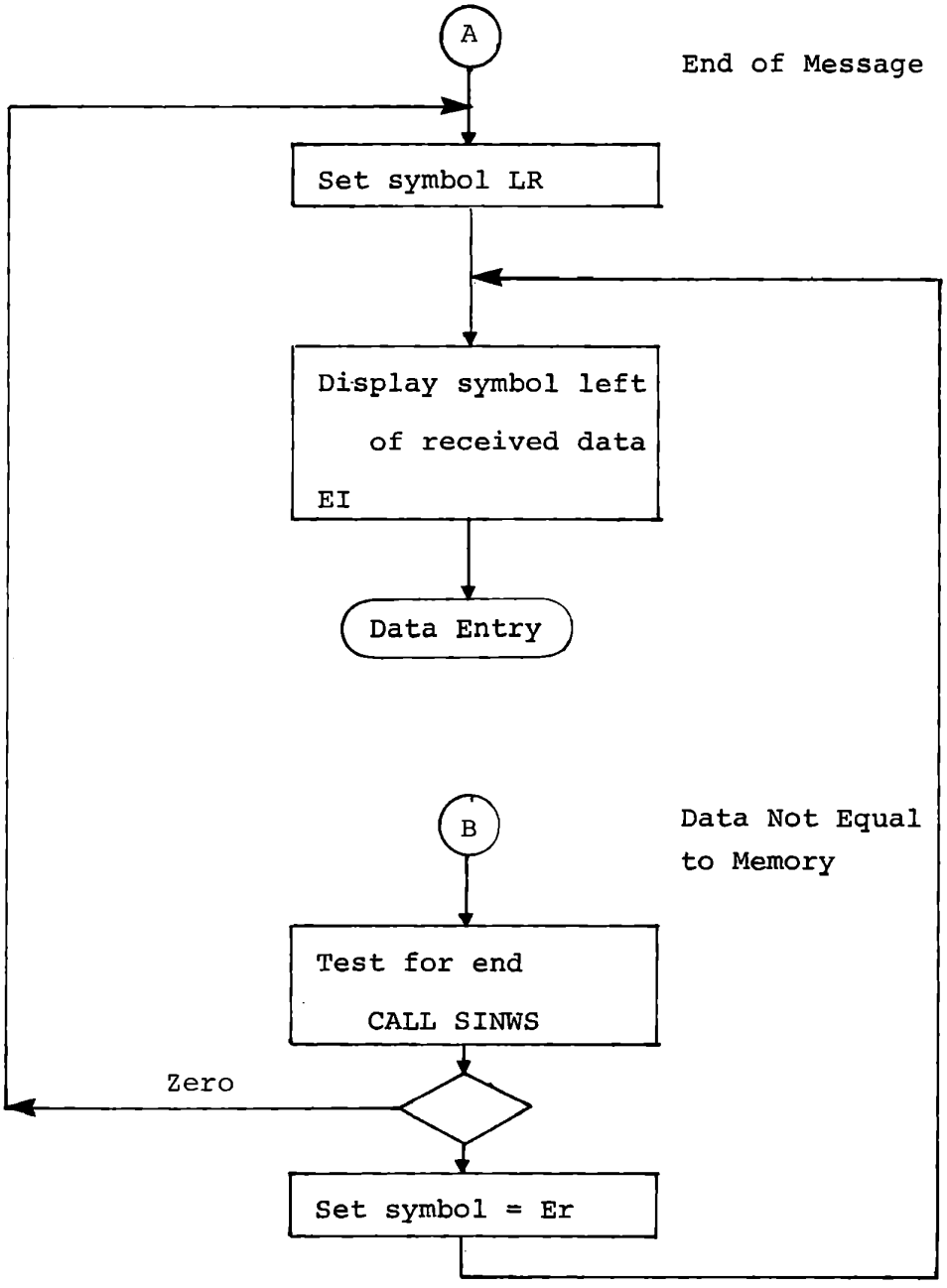
b) Received data is different from the memory data.

In the latter case, there may be an error, or the LRC recorded on tape may have been received. If an error has occurred, more data will be received, but if the LRC has been received SINWS should return Zero set. This is tested by the final call to SINWS. If there is an error, Err is displayed. If the end of message is found, Lr is displayed to signify the LRC character.



Receive Main Loop With SINWS

Figure 9-17a



Receive Main Loop With SINWS

Figure 9-17b

INSTRUCTION TIMING

	Clock Periods
MOV r,r	5
MOV r,M; MOV M,r	7
MVI r	7
MVI M	10
LXI rp	10
LDA; STA	13
LDAX; STAX	7
LHLD; SHLD	16
SPHL; PCHL	5
XCHG	4
XTHL	18
POP	10
PUSH	11
INR r; DCR 4	5
INR M; DCR M	10
INX rp; DCX rp	5
DAD rp	10
ADD r; ADC r; SUB r; SBB r	4
ANA r; XRA r; ORA r, CMP r	
ADD M, etc	7
ADI etc	7
RLC; RRC; RAL; RAR	4
DAA; CMA; STC; CMC	
JMP; JNZ; etc	10
CALL	17
CNZ etc - executed	17
- not executed	11
RET	10
RNZ etc - executed	11
- not executed	5
HLT (if interrupted immediately)	7
NOP	4
IN; OUT	10
EI; DI	4
RST	11

Figure 9-18

9.7 CALCULATING DELAY TIMES

In the previous exercises we have used the monitor subroutine DELYT and DELYC. When you design delay loops with critical time requirements, it is necessary to calculate the timing. Figure 9-15 lists the number of clocks for each 8080 instruction. As an exercise design a delay subroutine to replace DELYT in the transmit program. Calculate the timing and the necessary delay value.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 10

BINARY AND DECIMAL ARITHMETIC

10. BINARY AND DECIMAL ARITHMETIC

A number of the exercises presented in earlier chapters have included some arithmetic functions, including (in Chapter 4) addition, subtraction and multiplication. In this chapter we review the basic concepts of binary arithmetic and the arithmetic instructions. We shall write programs for decimal arithmetic and signed numbers. The multiplication by repetitive addition technique used in Chapter 4 would be very slow for multi-byte numbers; we shall write a fast multiplication program using shifting. Fractions and floating point numbers are also discussed.

10.1 BINARY ADDITION

The rules for binary addition were presented in Chapter 1, Section 1.2.4, and a quick review of that material is suggested. The complete addition table for binary arithmetic is:

```
0 + 0 =  0
0 + 1 =  1
1 + 0 =  1
1 + 1 = 10
```

Addition of two bit numbers produces carries into the third position. This extends to full eight bit addition:

```
   1111 1111
+  1111 1111
= 11111 1110
```

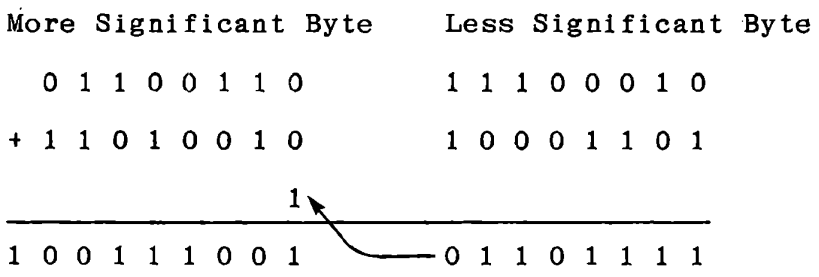
Eight bit addition can generate a carry into the ninth position. The addition of two numbers of any size may produce a carry into the next bit position. When a carry is generated, however, the sum never has ones in all positions. The example above shows the addition of the two largest possible eight bit numbers. A carry is generated but the least significant bit is zero. This is of fundamental importance for multiple precision addition.

10.1.1 Multiple Precision:

The use of more than one word to represent a number is termed multiple precision. If the number is an integer, this permits a

greater value than can be represented in a single word. If the number is a fraction it permits greater precision than can be represented in a single word. The number of words used often serves to describe the operation. Thus, double precision refers to arithmetic operations using two words, triple precision to three words, etc.

Consider a double precision addition in which each number is represented by two memory words (or bytes in an eight bit machine):



We add the two less significant bytes, and if a carry is generated, as above, it must be added in with the more significant bytes. Even if every bit in all four bytes was one, only a single carry bit is generated from the complete addition. This permits a multiple precision addition to proceed as follows:

a) Add the two less significant bytes.

b) Add the next two bytes, and if a carry resulted from the preceding addition add it into the sum.

c) Repeat (b) for as many bytes as are required.

The ADC instruction is used for multiple precision arithmetic. As with the other arithmetic and logical instructions there is a version of ADC using each of the registers as a source:

```
8F   ADC  A        Add the content of the
88   ADC  B        named register and the
89   ADC  C        Carry flag to the content
8A   ADC  D        of Register A, and place
8B   ADC  E        the result in Register A.
8C   ADC  H
8D   ADC  L        All flags are set or reset
8E   ADC  M        according to the result.
```

A double precision add of the content of register pairs B,C and D,E could be done by:

```
MOV    A,C      (A) <- Less significant byte
ADD    E        Ignore previous carry on first addition
MOV    E,A      Store less significant byte
MOV    A,B      (A) <- More significant byte
ADC    D        Add with carry
MOV    D,A      Store more significant byte
```

The 8080 includes a separate double precision add function, however, allowing two register pairs to be added directly. The above could have been performed by:

```
XCHG          Move (D,E) into (H,L)
DAD   B       Add (B,C) to (H,L)
XCHG          Put the result in (D,E)
```

Of course if one number had been in HL originally and we wanted the result in HL, a single DAD instruction would do the job. Therefore, double precision is usually done with DAD rather than ADC.

For convenience in discussing these functions we will refer to the augend (a number to which another will be added to generate a sum) and the addend (a number to be added to an augend to generate a sum).

10.2 FOUR BYTE ADDITION

We will use the following specification for this exercise:

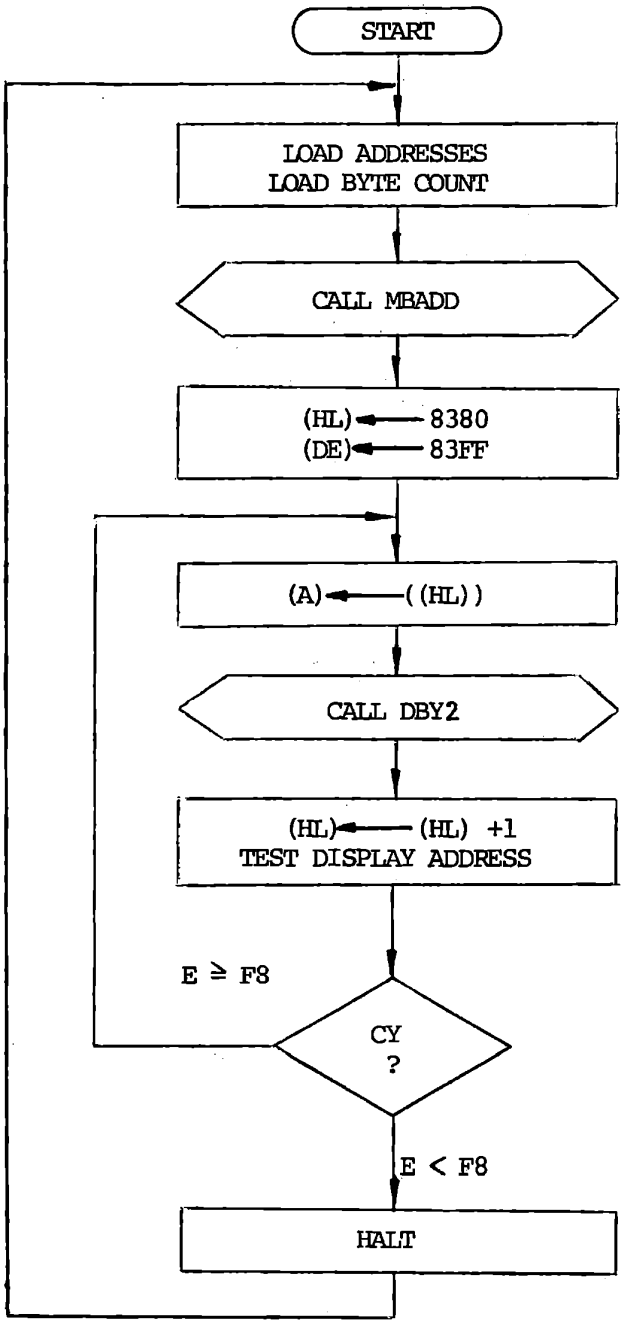
a) To a four byte number in memory locations 8380 - 8383 add the four byte number in 8390 - 8393.

b) Place the result in 8380 - 8383 and clear 8390 - 8393.

c) Display the result.

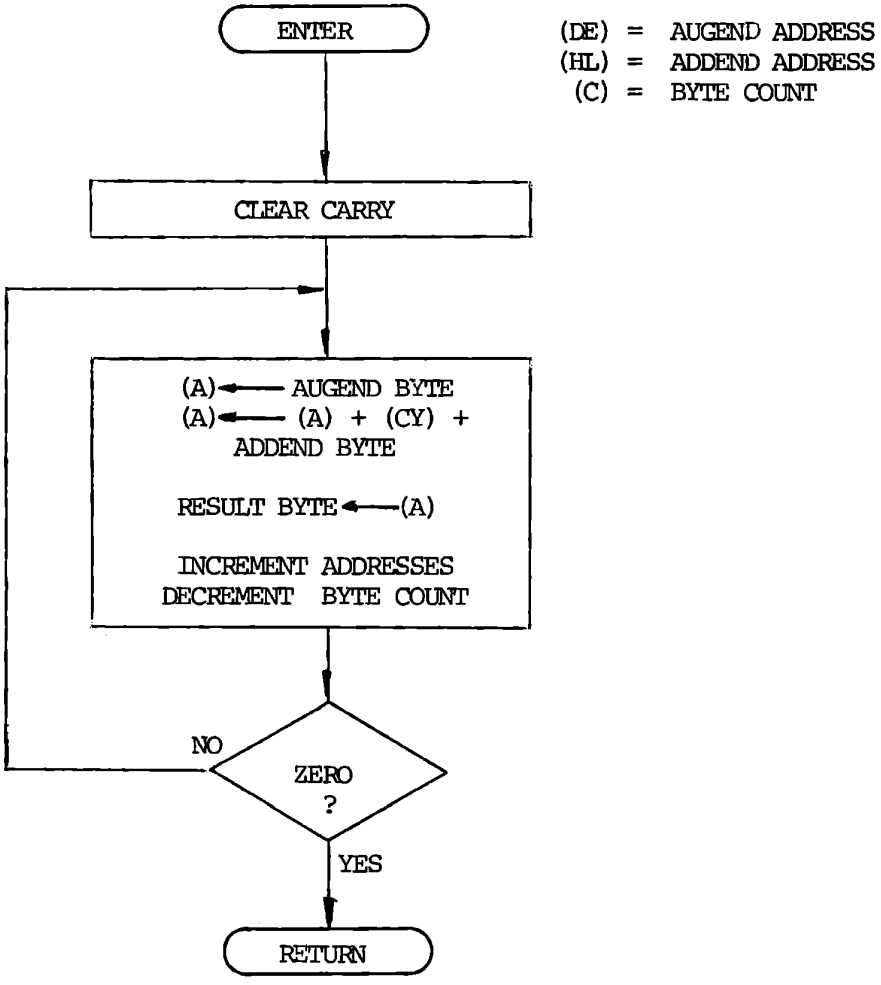
Write a subroutine for the addition, to be called with addresses and byte count already loaded. Note that you can modify addresses and count bytes without affecting the Carry flag, because INR and DCR affect all flags except carry.

Figures 10-1 through 10-4 present flow charts and coding sheets for this exercise.



Main Programs for Four Byte Add and Display

Figure 10-1



Multi Byte Add Subroutine

Figure 10-2

MAIN PROGRAM FOR 4 BYTE ADD AND DISPLAY

ADDR		CODE													
8220	0	00		NOP											
	1	00		NOP											
	2	00		NOP											
	3	11		LXI		D,	8	3	8	0	Address for
	4	80													Augend and result
	5	83													
	6	D5		PUSH		D									Save for display
	7	21		LXI		H,	8	3	9	0	Address for
	8	90													Addend, to be
	9	83													cleared
	A	0E		MVI		C,	0	4			Byte count
	B	04													for addition
	C	CD		CALL		MBADD					Multi Byte Add
	D	F0													subroutine
	E	82													
	F	E1		POP		H					(HL) ← Data Address
8210	0	11		LXI		D,	8	3	F	F	(DE) ← Display Address
	1	FF													
	2	83													
	3	7E		MOV		A,	M				(A) ← Byte
	4	CD		CALL		DBY2					Display byte
	5	98													subroutine
	6	02													
	7	23		INX		H					Address next byte
	8	7B		MOV		A,	E				Test display address
	9	FE		CPI		F8					to see if all have
	A	F8													been displayed
	B	D2		JNC		8213					Continue until
	C	13													left digit displayed
	D	82													
	E	00		NOP											
	F	00		NOP											
8220	0	76		HLT											
	1	C3		JMP		8200									
	2	00													
	3	82													
	4														
	5														
	6														
	7														
	8														

Figure 10-3

MULTI-BYTE ADDITION SUBROUTINE

ADDR		CODE													
8 2F	0	B7			ORA		A								Clear carry
	1	1A			LDAX		D								(A) ← Augend
	2	8E			ADC		M								Add Addend
	3	12			STAX		D								((DE)) ← Sum
	4	36			MVI		M,	00							Clear Addend
	5	00													
	6	13			INX		D								Next augend address
	7	23			INX		H								Next addend address
	8	0D			DCR		C								Count bytes
	9	C2			JNZ		82F1								Loop until finished
	A	F1													
	B	82													
	C	FB			EI										Enable Interrupt
	D	C9			RET										if used with
	E														interrupt exercise
	F														
8	0				NOTE:				ENTER WITH						
	1				(DE) =				ADDRESS FOR AUGEND						
	2								AND						RESULT
	3				(HL) =				ADDRESS FOR ADDEND						
	4								TO				BE CLEARED		
	5				(C) =				NUMBER OF BYTES						
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-4

The calling program uses a feature that is seldom convenient with the monitor - the HLT instruction. After displaying the result, your taskis finished until you load new data, so it is reasonable to HLT until an interrupt occurs. As long as the STEP/AUTO toggle switch is in the STEP position, however, the monitor interrupts at every instruction, so you cannot really halt. You will be interrupted, go back to the start and do the addition and display again. Since the augend now contains the result and the addend is cleared, the result will be the same and the display will be fixed, as though the halt had been effective. Now if you turn the switch to AUTO, the processor will indeed halt until you press RST or introduce an interrupt some other way. The difference is not visible unless you watch with an oscilloscope. The modification shown in Figure 10-5 uses a trick to make it visible. We turn on the decimal point at the right hand digit just before the halt, and turn it off immediately afterward, so it is only illuminated during the halt. Try it in both STEP and AUTO modes.

MODIFY MAIN TO DISPLAY HALT

ADDR	CODE												
8 22 0	2	1		L	X	I		H	,	8	3	F	F	Address - right hand
1	F	F												display digit
2	8	3												
3	7	E		M	O	V		A	,	M				Fetch display
4	E	E		X	R	I		8	0					Turn decimal
5	8	0												point on
6	7	7		M	O	V		M	,	A				Display
7	7	6		H	L	T								Halt
8	7	E		M	O	V		A	,	M				Fetch display
9	E	E		X	R	I		8	0					Turn decimal
A	8	0												point off
B	7	7		M	O	V		M	,	A				Display
C	C	3		J	M	P		8	2	0	0			Jump back
D	0	0												
E	8	2												
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 10-5

10.3 BINARY SUBTRACTION

The process of subtraction is defined by these equations:

```
If      A  =  B  + C
then    A  -  B  =  C
and     A  -  C  =  B
```

This can be expressed in terms of 8080 instructions:

```
MOV     A,B
ADD     C          (A) <- (B) + (C)
SUB     B          (A) <- (A) - (B)      result is equal to C
```

Successive ADD and SUB of the same values cancel each other, except that flags may be affected. The subtract instruction is again one of a set which includes one for each register:

```
97   SUB  A        Subtract the content of the named
90   SUB  B        register from the content of
91   SUB  C        Register A.  If the content of the
92   SUB  D        named register was greater than
93   SUB  E        Register A, set the carry flag.  Set
94   SUB  H        or clear the other flags according
95   SUB  L        to the results of the subtraction.
96   SUB  M
```

Like ADD, SUB ignores and destroys the previous content of the Carry flag. Another set of instructions SBB r, includes the Carry flag:

		SBB r	(A)	(A)	<-	(r)	-	(CY)
Example:	98	SBB B	(A)	(A)	<-	(B)	-	(CY)

The result of SUB or SBB sets or clears the Carry flag, which is meant to be passed to the next more significant byte. In subtraction, it becomes a borrow flag. It is set if the subtrahend ((B, in the example) is greater than the minuend (A), and in multi-byte subtraction the borrow is subtracted from A when the next byte is processed. This is done by the subtract with borrow instruction:

9F	SBB	A	Subtract from the content
98	SBB	B	of the Register A
99	SBB	C	content of the Carry
9A	SBB	D	flag and the content
9B	SBB	E	of the named register.
9C	SBB	H	Place the result in
9D	SBB	L	Register A. Set or clear
9E	SBB	M	all flags according to the result.

A double precision subtraction can be done by:

```
MOV   A,C
SUB   L
MOV   E,A       (E) <- (C) - (L)
MOV   A,B
SBB   H
MOV   D,A       (D) <- (B) - (H) - (CY)
```

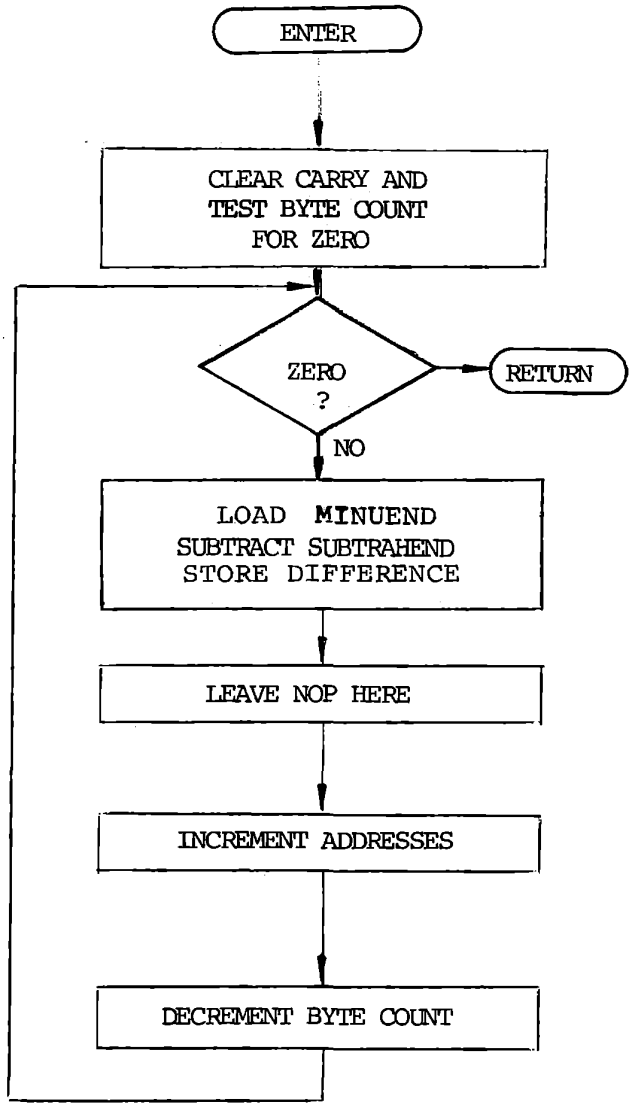
The result in (DE) is (BC) - (HL). Multiple precision subtraction would use the SBB M instruction:

```
LDAX B
SBB M
STAX D          ((DE)) <- ((BC)) - ((HL)) - (CY)
INX B  \
INX D  }        next addresses
INX H  /
```

Note that we have used three register pairs for addresses, and Register A for the subtraction, leaving no register available to count bytes. We can keep a byte counter in a fixed memory location and use LDA, DCR A, STA to count, or we can use the stack. But be careful: POP PSW to bring a counter into Register A will destroy the Carry flag, which is needed. This is a place where the XTHL instruction is very useful. Write a subroutine for a general purpose multi-byte subtraction, entering with:

(A) = number of bytes
(B,C) = address for minuend
(D,E) = address for difference
(H,L) = address for subtrahend

We can use the same calling program as for the addition, except that we must load an address to (B,C) and initialize a byte counter in A, and the call will be to the subtract subroutine at 82D0. Place the minuend (from which the subtrahend will be subtracted) at 8370 - 73; the difference at 8380 - 83, and the subtrahend at 8390 - 93. Since they are to be kept separate, do not clear any of these areas during the operation. For convenience in an exercise of the following section, leave a NOP immediately after the SBB M instruction.



Multi-Byte Subtract Subroutine
Figure 10-6

MAIN PROGRAM FOR 4 BYTE SUBTRACT

ADDR		CODE													
8 20	0	01		LXI		B,	8370								Address for minuend
	1	70													
	2	83													
	3	11		LXI		D,	8380								address for difference
	4	80													
	5	83													
	6	D5		PUSH		D									Save for display
	7	21		LXI		H,	8390								Address for subtrahend
	8	90													
	9	83													
	A	3E		MVI		A,	04								Byte count in A
	B	04													
	C	CD		CALL		MBSUB									
	D	D0													
	E	82													
	F	E1		POP		H									(HL) ← Result Addr
8 21	0	11		LXI		D,	83FF								(DE) ← Displ Addr
	1	FF													
	2	83													
	3	7E		MOV		A,	M								(A) ← Byte
	4	CD		CALL		DBY2									Display Byte Subroutine
	5	98													
	6	02													
	7	23		INX		H									Address next byte
	8	7B		MOV		A,	E								Test display address
	9	FE		CPI		F8									
	A	F8													
	B	D2		JNC		8213									
	C	13													
	D	82													
	E	00		NOP											
	F	00		NOP											
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-7

ADDR		CODE													
8	22 0	21		LXI	H,	83	FF	This segment
	1	FF													is not needed
	2	83													
	3	7E		MOV	A,	M									
	4	EE		XRI	80										
	5	80													
	6	77		MOV	M,	A									
	7	76		HLT											
	8	7E		MOV	A,	M									
	9	EE		XRI	80										
	A	80													
	B	77		MOV	M,	A									
	C	C3		JMP	8200										Go back to start
	D	00													
	E	82													
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-8

MULTI-BYTE SUBTRACTION SUBROUTINE

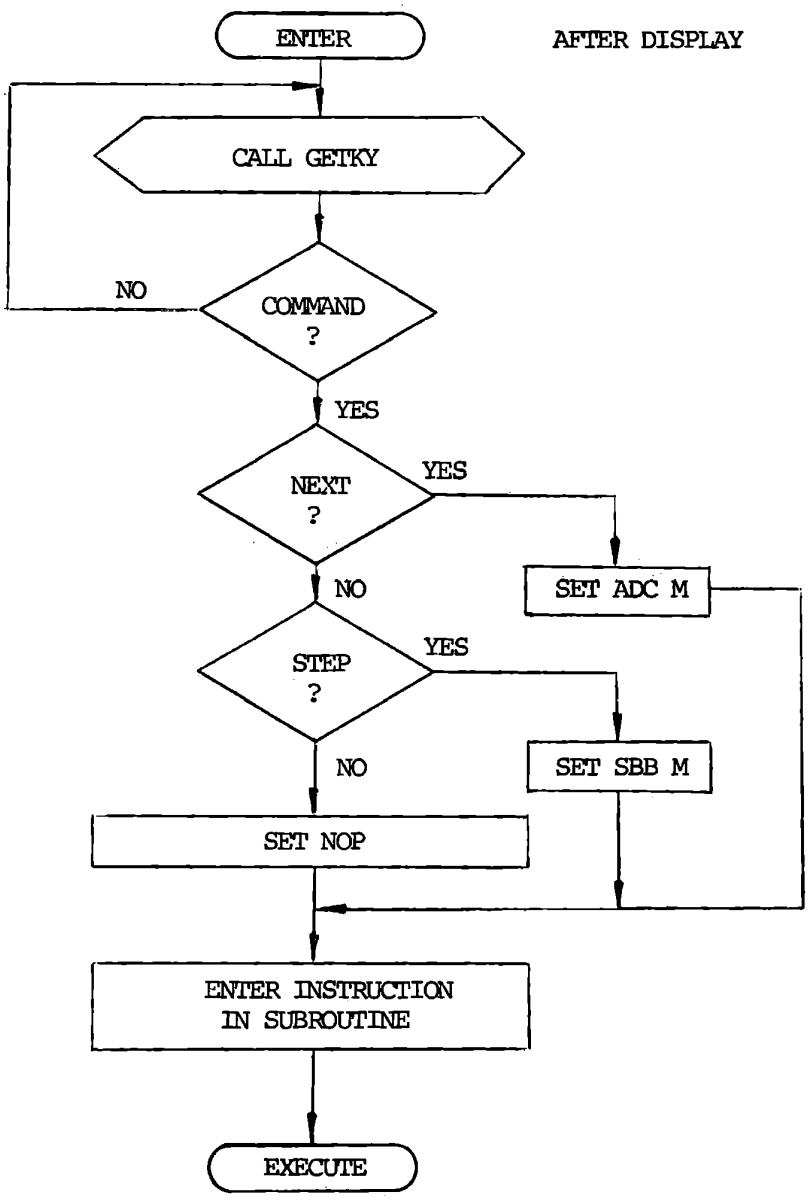
ADDR		CODE														
8 2D	0	B	7		O	R	A		A							Clear CY and test for
	1	C	8		R	Z										zero byte count
	2	F	5		P	U	S	H		P	S	W				Save byte count
	3	0	A		L	D	A	X		B						Load Minuend
	4	9	E		S	B	B		M							Subtract Subtrahend
	5	0	0		N	O	P									to be used later
	6	1	2		S	T	A	X		D						Store difference
	7	0	3		I	N	X		B							Next addresses
	8	1	3		I	N	X		D							
	9	2	3		I	N	X		H							
	A	E	3		X	T	H	L								(ST) ← (HL);
	B	7	C		M	O	V		A	,	H					(A) ← Byte count
	C	E	1		P	O	P		H							Restore HL and stack
	D	3	D		D	C	R		A							Decrement count
	E	C	3		J	M	P		8	2	D	1				Go to test for
	F	D	1													zero and return
8	0	8	2													when finished
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 10-9

The subroutine can be changed from subtraction to addition by altering one instruction (at 82D4):

```
9E   SBB  M      to subtract
8E   ADC  M      to add
```

We now introduce a scheme that is not available to programs stored in ROM but can be very convenient for programs in RAM. The program can modify itself by altering the instruction in response to an input. After the display, and before jumping back to the start, take a key input for a command to add or subtract. Use NEXT (=15) for add; STEP (=13) for subtract. For any undefined key enter NOP instead of either ADC or SBB. Use the monitor subroutine GETKY, which waits for a key to be entered. Figures 10-10 through 10-12 show a coding example.



Program Modify Module

Figure 10-10

ADDR		CODE														
8 22	0	C	D		C	A	L	L		G	E	T	K	Y	Accept key.
	1	3	D												
	2	0	2												
	3	D	A		J	C				8	2	2	0		Reject hex keys
	4	2	0												
	5	8	2												
	6	2	1		L	X	I			H	,	8	2	D	4	Address of add
	7	D	4													or subtract
	8	8	2													instruction
	9	0	6		M	V	I			B	,	8	E			Enter ADC M
	A	8	E													
	B	F	E		C	P	I			1	5					Is key NEXT?
	C	1	5													
	D	C	A		J	Z				8	2	3	A			If so, go store
	E	3	A													ADC M in
	F	8	2													subroutine
8 23	0	0	6		M	V	I			B	,	9	E			Enter SBB M
	1	9	E													
	2	F	E		C	P	I			1	3					Is key STEP?
	3	1	3													
	4	C	A		J	Z				8	2	3	A			If so, go store
	5	3	A													SBB M in
	6	8	2													subroutine
	7	0	6		M	V	I			B	,	0	0			Enter NOP
	8	0	0													for any other key
	9	0	0		N	O	P									
	A	7	0		M	O	V			M	,	B				Enter instruction
	B	C	3		J	M	P			8	2	0	0			in subroutine
	C	0	0													and execute
	D	8	2													
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 10-11

MULTI-BYTE ADD/SUBTRACT SUBROUTINE

ADDR		CODE													
8 2D	0	B	7		ORA		A								Clear CY and test for
	1	C	8		RZ										zero byte count
	2	F	5		PUSH		PSW								Save byte count
	3	0	A		LDAX		B								Load augend/minuend
	4	0	0		NOP										to be replaced
	5	0	0		NOP										
	6	1	2		STAX		D								Store sum/difference
	7	0	3		INX		B								Next addresses
	8	1	3		INX		D								
	9	2	3		INX		H								
	A	E	3		XTHL										(ST)←(HL);
	B	7	C		MOV		A, H								(A)←Byte Count
	C	E	1		POP		H								Restore HL and stack
	D	3	D		DCR		A								Decrement count
	E	C	3		JMP		82D1								Go to test for
	F	D	1												zero and return
8	0	8	2												when finished
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-12

10.4 DECIMAL ADDITION AND SUBTRACTION

Often the microprocessor will have a human interface for its arithmetic results, and decimal input and output will be required. The 8080 provides an instruction to convert a binary result to a decimal result:

27 DAA Decimal Adjust Accumulator

This tests the result of an arithmetic instruction and corrects the content of the accumulator to create a "packed decimal" result, in the form of two decimal digits. Before exploring the operation in detail we will insert the instruction into the subroutine of the previous exercise. To compare results of decimal versus binary arithmetic, we will provide for inserting or deleting this instruction under keyboard control as we did the ADC and SBB instructions. Use the key RUN to invoke binary and ADDR to invoke decimal results, and interpret them as you did NEXT or STEP . Insert NOP after ADC or SBB for binary, DAA for decimal. As before, any undefined key should place a NOP in place of the ADC or SBB.

If the numbers used generate no carries, the binary and decimal results are alike. Try putting 33 33 33 33 at 8370 - 73 for the augend or minuend and 22 22 22 22 at 8390-93 for the addend or subtrahend. Then addition will produce 55 55 55 55; subtraction, 11 11 11 11. Try your program with those numbers to make sure it works. Coding examples are shown in Figures 10-13 and 10-14.

CODING SHEET

MODIFY SUBROUTINE BY KEY INPUT

ADDR		CODE													
8 22	0	C	D		C	A	L	L		G	E	T	K	Y	Accept key
	1	3	D												
	2	0	2												
	3	D	A		J	C			8	2	2	0			Reject key keys
	4	2	0												
	5	8	2												
	6	2	1		L	X	I		H	,	8	2	D	4	Address of add
	7	D	4												or subtract
	8	8	2												instruction
	9	0	6		M	V	I		B	,	8	E			Enter ADC M
	A	8	E												
	B	F	E		C	P	I		1	5					Is key NEXT?
	C	1	5												
	D	C	A		J	Z			8	2	3	A			If so, go store
	E	3	A												ADC M in
	F	8	2												subroutine
8 23	0	0	6		M	V	I		B	,	9	E			Enter SBB M
	1	9	E												
	2	F	E		C	P	I		1	3					Is key STEP?
	3	1	3												
	4	C	A		J	Z			8	2	3	A			If so, go store
	5	3	A												SBB M in
	6	8	2												subroutine
	7	C	3		J	M	P		8	2	4	0			If process other
	8	4	0												key commands
	9	8	2												
	A	7	0		M	O	V		M	,	B				
	B	C	3		J	M	P		8	2	0	0			
	C	0	0												
	D	8	2												
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-13

ADDR		CODE												
8 24	0	2	3		I	N	X		H					Address 82D5
	1	0	6		M	V	I		B	,	0	0		Enter NOP
	2	0	0											
	3	F	E		C	P	I		1	4				Is key RUN?
	4	1	4											(means HEX)
	5	C	A		J	Z			8	2	3	A		Yes enter NOP
	6	3	A											
	7	8	2											
	8	0	6		M	V	I		B	,	2	7		Enter DAA
	9	2	7											
	A	F	E		C	P	I		1	2				Is key ADDR?
	B	1	2											(means decimal)
	C	C	A		J	Z			8	2	3	A		Yes enter DAA
	D	3	A											
	E	8	2											
	F	2	B		D	C	X		H					Address 82D4
8 25	0	0	6		M	V	I		B	,	0	0		Enter NOP in
	1	0	0											place of ADC or SBB
	2	C	3		J	M	P		8	2	3	A		
	3	3	A											
	4	8	2											
	5													
	6													
	7													
	8													
	9													
	A													
	B													
	C													
	D													
	E													
	F													
8	0													
	1													
	2													
	3													
	4													
	5													
	6													
	7													
	8													

Figure 10-14

BINARY AND DECIMAL ARITHMETIC

Now compare the binary and decimal operations. Enter these data:

```
8370 43   low byte  \
  71 65              |        Augend or
  72 87              |        Minuend
  73 09   high byte  /
8390 78   low byte  \
  91 77              |        Addend or
  92 77              |        Subtrahend
  93 07   high byte  /
```

Run your program using the steps shown below:

RUN,NEXT	Augend	0 9 8 7 6 5 4 3
(binary add)	Addend	0 7 7 7 7 7 7 8
	Sum	1 0 F E D C B B

No carries have occurred except for 09 + 07.

ADDR,NEXT	Augend	0 9 8 7 6 5 4 3
(decimal add)	Addend	0 7 7 7 7 7 7 8
	Sum	1 7 6 5 4 3 2 1

Carries have occurred from all digits.

RUN, STEP	Minuend	0 9 8 7 6 5 4 3
(binary subtract)	Subtrahend	0 7 7 7 7 7 7 8
	Difference	0 2 0 F E D C B

Borrows have occurred from the first and second bytes.

ADDR, STEP	Minuend	0 9 8 7 6 5 4 3
(decimal subtract)	Subtrahend	0 7 7 7 7 7 7 8
	Correct difference	0 2 0 9 8 7 6 5

The computer generats an incorrect result! Decimal Adjust only works for addition, not for subtraction. We will see what is necessary for decimal subtraction in Section 10.7.3.

The binary to decimal correction process for addition works as follows: the addition is performed, and a flag called Auxiliary Carry is set if a carry occurs from Bit 3 to Bit 4 - that is, from the first digit to the second. When DAA is executed, the content of the Accumulator and both Carry (CY) and Auxiliary Carry (AC) flags are tested. Then the following is done:

If the value of the low four bits exceeds 9, or if the AC is set, add 06 to the Accumulator. These corrections occur:

```
ADC     07 + 08 -> 0F          no carry
DAA     0F + 06 -> 15

ADC     08 + 08 -> 10          AC set
DAA     10 + 06 -> 16
```

After this correction to the low digit, if the value of the high four bits exceeds 9 or if CY is set, add 60 to the Accumulator. These corrections are made:

```
ADC     70 + 80 -> F0          no carry
DAA     FA + 60 -> 50          CY set

ADC     80 + 80 -> 00          CY set
DAA     00 + 60 -> 60          CY still set
```

Note that when 60 is added it may set the CY but will not clear it. The following examples taken from the experiment with the program show the correction process in operation:

```
ADC     43 + 78 -> BB          no carry
DAA     BB + 06 -> C1
        C1 + 60 -> 21          sets CY

ADC     65 + 77 +  CY - DD     no carry
DAA     DD + 06 -> E3          sets AC
        E3 + 60 -> 43          sets CY

ADC     87 + 77 +  CY - FF     no carry
DAA     FF + 06 -> 05          sets CY
        05 + 60 -> 65          CY still set

ADC     09 + 07 +  CY - 11     sets AC
DAA     11 + 06 -> 17
```

Caution: The DAA instruction only works correctly while the CY and AC flags are still set or cleared in response to the arithmetic instruction that produced the binary result. Any intervening arithmetic or logical instruction, or INR or DCR, affects its operation. The safe procedure is always to place DAA immediately after the instruction whose result is to be corrected.

Note that DAA corrects the result of addition of decimal numbers to give a decimal result. It does NOT convert a binary number to a decimal equivalent.

If you want to investigate the DAA command further, the program shown in Figure 10-15 will let you try different instructions and view the results.

FOR EXPERIMENT WITH DAA

ADDR	CODE															
8 20 0	7D		MOV		A	,	L									Recover previous value
1	3C		INR		A											Experiment with other instructions
2	00		NOP													
3	00		NOP													
4	00		NOP													
5	27		DAA													Decimal adjust
6	6F		MOV		L	,	A									Save result
7	CD		CALL				DBYTE									Display result
8	45															
9	02															
A	CD		CALL				GETKY									Wait for a key
B	3D															
C	02															
D	D2		JNC		8200											Jump if command
E	00															
F	82															
8 21 0	6F		MOV		L	,	A									Save new key
1	AF		XRA		A											Clear the flags
2	7D		MOV		A	,	L									Recover the key
3	C3		JMP		8205											Go to DAA
4	05															
5	82															
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																
3 0																
1																
2																
3																
4																
5																
6																
7																
8																

Figure 10-15

10.5 BINARY MULTIPLICATION

Multiplication of integers is a process of repeated addition, or a substitute process that gives the same results.

$$3 + 3 + 3 + 3 = 12$$

$$4 \times 3 = 12$$

We have previously performed multiplication by repetitive addition. This is the easiest way, and the required program can be very short and easy to write, but it is very slow when the multiplier is large. The usual computer multiplication process is similar to what we do by hand.

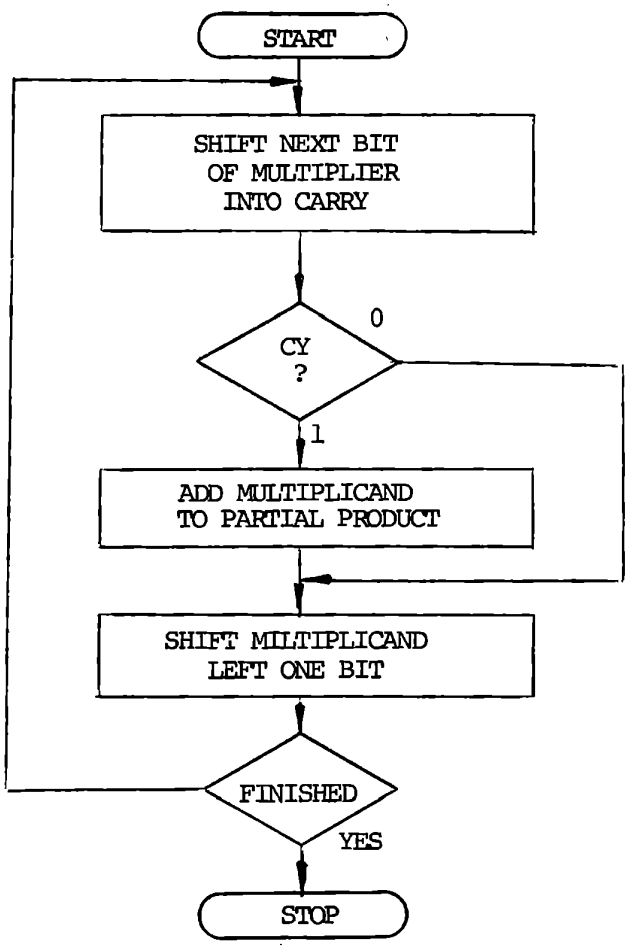
Multiplicand		362				
Multiplier	x	426				
		1972	=	6	x	362
		7240	=	20	x	362
		144800	=	400	x	362
Product		154012	=	426	x	362

In our familiar multiplication process we simply multiply the multiplicand by each component of the multiplier and add the individual products. Multiplication becomes trivially easy if the multiplier happens to comprise only ones and zeros:

```
      362
x     101
      362        1  x 362
        0        0  x 362
    36200      100  x 362
    36562
```

With binary numbers, of course, multiplication is that easy. According to whether each bit in the multiplier is zero or one, the multiplicand, appropriately shifted, is added into a partial product. Figure 10-16 shows the process, with an example of two 8-bit numbers. The flow chart shows one approprite procedure. The only difference from paper and pencil multiplication is that the addition is performed after each shift, instead of writing the numbers down and adding the column later. Write a program to implement the process. A solution is provided in Figure 10-17a.

```
MULTIPLICAND        0 1 1 0 0 0 1 0
MULTIPLIER          0 0 1 0 0 1 1 0 <-
                                    0   0
                              0 1 1 0 0 0 1 0   1
                            0 1 1 0 0 0 1 0     1
                                      0         0
                                    0           0
                        0 1 1 0 0 0 1 0         1
                                  0             0
                                0               0
                    0 0 0 1 1 1 0 1 0 0 0 1 1 0 0
```



Binary Multiplication
Figure 10-16

BINARY MULTIPLY TWO BYTE PRODUCT

ADDR		CODE														
8 20	0	C	D		C	A	L	L		E	N	T	B	Y	Get multiplicand
	1	3	6												
	2	0	3												
	3	E	5		P	U	S	H		H					Save it
	4	C	D		C	A	L	L		E	N	T	B	Y	Get Multiplier
	5	3	6												
	6	0	3												
	7	7	D		M	O	V		A	,	L				(A) ← Multiplier
	8	D	1		P	O	P		D						(E) ← Multiplicand
	9	2	1		L	X	I		H	,	0	0	0	0	Clear product
	A	0	0												and high byte
	B	0	0												of multiplicand
	C	5	4		M	O	V		D	,	H				
	D	1	F		R	A	R								Shift multiplier
	E	D	2		J	N	C		8	2	1	2			Skip add if bit = 0
	F	1	2												
8 21	0	8	2												
	1	1	9		D	A	D		D						Add multiplicand
	2	E	B		X	C	H	G							to product
	3	2	9		D	A	D		H						Double multiplicand
	4	E	B		X	C	H	G							
	5	B	7		O	R	A		A						Test multiplier
	6	C	2		J	N	Z		8	2	0	D			for zero
	7	0	D												
	8	8	2												
	9	7	D		M	O	V		A	,	L				Display product
	A	C	D		C	A	L	L		D	B	Y	T	E	low byte
	B	9	5												
	C	0	2												
	D	7	C		M	O	V		A	,	H				Display product
	E	C	D		C	A	L	L		D	B	Y	2		high byte
	F	9	8												
8 22	0	0	2												
	1	C	3		J	M	P		8	2	0	0			Go back for
	2	0	0												input
	3	8	2												
	4														
	5														
	6														
	7														
	8														

Figure 10-17a

There is an alternate scheme, sometimes more convenient, in which the multiplication is done backwards:

```
DAD  H      Double Product
ADD  A      Next MSB to CY
JNC         Skip add if bit = 0
DAD  D      Add multiplicand
DCR  C      Count bits
JNZ
```

The product is developed from most significant position toward least significant, and instead of shifting the multiplicand we shift the product. The result is identical. This requires a bit counter, since the product must be shifted eight times, whereas the previous program can stop as soon as the multiplier reaches a value of zero. Figure 10-17b shows the program.

ALTERNATE BINARY MULTIPLY

ADDR		CODE														
8 20	0	C	D		C	A	L	L		E	N	T	B	Y	Get multiplicand
	1	3	6												
	2	0	3												
	3	E	5		P	U	S	H		H					Save it
	4	C	D		C	A	L	L		E	N	T	B	Y	Get multiplier
	5	3	6												
	6	0	3												
	7	7	D		M	O	V		A	,	L				(A) ← Multiplier
	8	D	1		P	O	P		D						(E) ← Multiplicand
	9	2	1		L	X	I		H	,	0	0	0	0	Clear product
	A	0	0												and high byte
	B	0	0												of multiplicand
	C	5	4		M	O	V		D	,	H				
	D	0	E		M	V	I		C	,	0	8			(C) ← Bit count
	E	0	8												
820	F	2	9		D	A	D		H						Shift product
8 21	0	8	7		A	D	D		A						Shift multiplier
	1	D	2		J	N	C		8	2	1	5			Skip add if
	2	1	5												multiplier bit = 0
	3	8	2												
	4	1	9		D	A	D		D						Add multiplicand
821	5	0	D		D	C	R		C						to product
	6	C	2		J	N	Z		8	2	0	F			Loop until 8 bits
	7	0	F												completed
	8	8	2												
	9	7	D		M	O	V		A	,	L				Display product
	A	C	D		C	A	L	L		D	B	Y	T	E	low byte
	B	9	5												
	C	0	2												
	D	7	C		M	O	V		A	,	H				Display product
	E	C	D		C	A	L	L		D	B	Y	2		high byte
	F	9	8												
3 22	0	0	2												
	1	C	3		J	M	P		8	2	0	0			Go back for
	2	0	0												input
	3	8	2												
	4														
	5														
	6														
	7														
	8														

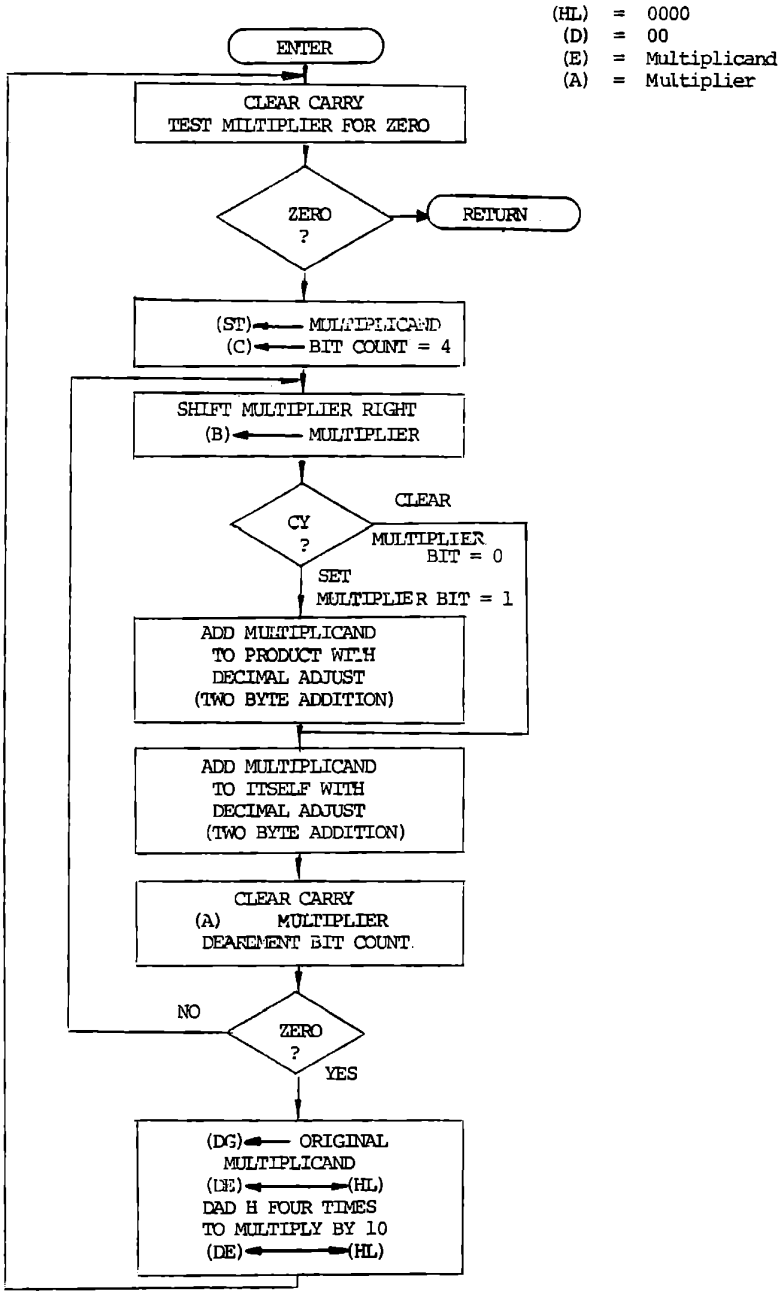
Figure 10-17b

10.6 DECIMAL MULTIPLICATION

Basically the same procedure is used for decimal multiplication, but it must be done digit by digit instead of a byte at a time, and since decimal adjustment is necessary the additions must take place in the Accumulator. It is common, but not necessary, to use unpacked decimal arithmetic (one decimal digit per byte) if multiplication and division are to be done, because it is more efficient. The decimal multiplication subroutine developed here is for packed decimal, with two digit multiplier and multiplicand and four digit result. This is the largest value that can be handled without storing data in the memory.

Figure 10-18 shows a flowchart of the subroutine, and Figure 10-19 the code. Like the first binary multiplication method, this shifts the multiplier right and doubles the multiplicand for each bit, stopping when the multiplier reaches zero. It also requires a bit counter, initialized to four bits, because after the first digit of the multiplier has been handled the original multiplicand must be recovered and multiplied by ten for the second digit.

The program used for the binary multiplication provides the input and display functions, calling this subroutine instead of doing the arithmetic itself.



Decimal Multiply Subroutine

Figure 10-18

DATA ENTRY AND DISPLAY FOR DECIMAL MULTIPLY

ADDR		CODE														
8 20	0	C	D		C	A	L	L		E	N	T	B	Y	Get multiplicand
	1	3	6												
	2	0	3												
	3	E	5		P	U	S	H		H					Save it
	4	C	D		C	A	L	L		E	N	T	B	Y	Get multiplier
	5	3	6												
	6	0	3												
	7	7	D		M	O	V		A	,	L				(A)←Multiplier
	8	D	1		P	O	P		D						(E)←Multiplicand
	9	2	1		L	X	I		H	,	0	0	0	0	Clear product
	A	0	0												and high byte
	B	0	0												of multiplicand
	C	5	4		M	O	V		D	,	H				
	D	C	D		C	A	L	L		D	E	C	M	U	Decimal multiply
	E	4	0												subroutine
	F	8	2												
8 21	0	C	3		J	M	P		8	2	1	9			
	1	1	9												
	2	8	2												
	3														
	4														
	5														
	6														
	7														
	8														
	9	7	D		M	O	V		A	,	L				Display product
	A	C	D		C	A	L	L		D	B	Y	T	E	low byte
	B	9	5												
	C	0	2												
	D	7	C		M	O	V		A	,	H				Display product
	E	C	D		C	A	L	L		D	B	Y	2		high byte
	F	9	8												
3 22	0	0	2												
	1	C	3		J	M	P		8	2	0	0			Go back for
	2	0	0												input
	3	8	2												
	4														
	5														
	6														
	7														
	8														

Figure 10-19a

Packed Decimal Multiply Sub

ADDR	CODE													
8 24 0	B	7		O	R	A		A						Start Digit Loop
1	C	8		R	Z									Return if multiplier 0
2	D	5		P	U	S	H		D					(ST) ← multiplicand
3	0	E		M	V	I		C	,	0	4			(C) ← digit bit count
4	0	4												
5	1	F		R	A	R								Start bit loop
6	4	7		M	O	V		B	,	A				(B) ← multiplier
7	D	2		J	N	C		8	2	5	2			Skip add if
8	5	2												multiplier bit = 0
9	8	2												
A	7	D		M	O	V		A	,	L				Add multiplicand
B	8	3		A	D	D		E						to product
C	2	7		D	A	A								
D	6	F		M	O	V		L	,	A				
E	7	C		M	O	V		A	,	H				
F	8	A		A	D	C		D						
8 25 0	2	7		D	A	A								
1	6	7		M	O	V		H	,	A				
825 2	7	B		M	O	V		A	,	E				Double multiplicand
3	8	3		A	D	D		E						
4	2	7		D	A	A								
5	5	F		M	O	V		E	,	A				
6	7	A		M	O	V		A	,	D				
7	8	A		A	D	C		D						
8	2	7		D	A	A								
9	5	7		M	O	V		D	,	A				
A	A	F		X	R	A		A						Clear carry
B	7	8		M	O	V		A	,	B				(A) ← Multiplier
C	0	D		D	C	R		C						Decr bit count
D	C	2		J	N	Z		8	2	4	5			End of bit loop
E	4	5												
F	8	2												
8 0				C	O	N	T	I	N	U	E	D		NEXT PAGE
1														
2														
3														
4														
5														
6														
7														
8														

Figure 10-19b

PACKED DECIMAL MULTIPLY SUBROUTINE (continued)

ADDR	CODE													
826 0	D	1		P	O	P		D						Restore multiplicand
1	E	B		X	C	H	G							
2	2	9		D	A	D		H						Multiply by 10
3	2	9		D	A	D		H						for high digit
4	2	9		D	A	D		H						
5	2	9		D	A	D		H						
6	E	B		X	C	H	G							
7	C	3		J	M	P		8	2	4	0			
8	4	0												
9	8	2												
A														
B														
C														
D														
E														
F														
830														
1														
2														
3														
4														
5														
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
840														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 10-19c

10.7 OTHER REPRESENTATIONS OF NUMBERS

There are many ways of storing numeric values in a computer, and we have used only two: binary unsigned integer and packed decimal unsigned integer. There are numerous others, including:

Binary Number Representations

- Unsigned integer
- Twos complement (signed binary)
- Fractional, fixed binary point
- Floating point

Decimal Number Representation

- Packed, unsigned integer
- Unpacked, unsigned integer
- Sign and magnitude (packed or unpacked)
- Hundreds complement (signed decimal)
- Tens complement (signed, unpacked)
- Fractional, fixed decimal point
- Floating Point

We will discuss the representation of signed numbers using twos or hundreds complement, and both fixed and floating point fractions.

10.7.1 Negative Binary Numbers

When we represent negative numbers on paper, we use a separate sign indicator attached to the corresponding positive value: e.g. - 232. This procedure is sometimes used in computers. It is called "sign and magnitude" representation. For integer arithmetic it is more efficient to use a different representation, called "twos complement".

Consider the sequence of hexadecimal values generated by decrementing a register, and the corresponding signed values that would be generated by repeatedly subtracting 1 from a number:

Count	Signed Value
03	+3
02	+2
01	+1
00	0
FF	-1
FE	-2

Here we can see that a hexadecimal value with 0 in the high digit can be considered positive, and an F in the high digit somehow represents a negative number.

If we add two numbers in this representation according to the rules of binary addition, we will obtain a correct result:

```
  0 0 0 0 0 0 1 1          03        +3
  1 1 1 1 1 1 1 0     +    FE   + (-2)
= 0 0 0 0 0 0 0 1     =    01   =  +1
```

This is the advantage of "twos complement" representation of signed numbers: They can be added (or subtracted) without separately considering their signs.

10.7.1.1 Changing the Sign

A positive number is changed to a negative number by subtracting it from zero:

00 - 02 = FE which represents -2

Similarly:

00 - FE = 02

The following program accepts a number and changes its sign. Try it, and find the representations for various negative numbers.

CHANGE SIGN OF NUMBER

ADDR		CODE														
8 20	0	CD			C	A	L	L		E	N	T	B	Y	Accept number	
	1	36														
	2	03														
	3	3E			M	V	I		A	,	0	0			Subtract from 0	
	4	00														
	5	95			S	U	B		L							
	6	CD			C	A	L	L		D	B	Y	T	E	Display number	
	7	95													with sign changed	
	8	02														
	9	C3			J	M	P			8	2	0	0			
	A	00														
	B	82														
	C															
	D															
	E															
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															
	9															
	A															
	B															
	C															
	D															
	E															
	F															
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 10-20

10.7.1.2 Range of Signed Numbers

Since we only need a single bit to indicate the sign of a number, we can define the most significant bit to represent the sign, and the other seven bits to represent the magnitude.

0 1 1 1 1 1 1 1	+7F (+127 decimal)
0 0 0 0 0 0 0 1	+01
0 0 0 0 0 0 0 0	0
1 1 1 1 1 1 1 1	-01
1 0 0 0 0 0 0 1	-7F (-127 decimal)
1 0 0 0 0 0 0 0	-80 (-128 decimal)

The list above demonstrates that a single byte can represent a signed number from -80 through +7F.

Note that the twos complement of zero is still zero. Also, the twos complement of 80 (hex) is still 80, but all other values change from positive to negative (or vice versa) when subtracted from zero. Numbers expressed in this form can be added or subtracted provided that the result can be expressed in seven bits.

10.7.1.3 Changing Sign by Complementing

A different procedure for changing the sign of a number is more convenient in many instances: Complement the number and increment the result:

0 0 0 0 0 0 0 1	+1
1 1 1 1 1 1 1 0	Complement
1 1 1 1 1 1 1 1	Increment

The result is the same as obtained by subtracting from zero. The advantage appears when the value was already in the Accumulator, since no other register needs to be used. Also, the Carry flag is not affected. Satisfy yourself that the following program gives the same result as the preceding program.

INTEGRATED COMPUTER SYSTEMS MICROCOMPUTER TRAINING SYSTEM CODING SHEET

CHANGE SIGN BY CMA, INR A

ADDR	CODE			
8 20 0	CD		CALL ENTBY	Accept number
1	36			
2	03			
3	7D		MOV A, L	
4	2F		CMA	Complement
5	3C		INR A	Increment
6	CD		CALL DBYTE	Display number
7	95			with sign changed
8	02			
9	C3		JMP 8200	
A	00			
B	82			
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
A				
B				
C				
D				
E				
F				
8 0				
1				
2				
3				
4				
5				
6				
7				
8				Figure 10-21

10.7.1.4 Sign Flag

Using twos complement representation, negative and positive numbers can be added and subtracted to obtain a signed result in twos complement notation. The sign of the result is also available in the Sign flag. This is set if the high bit of the result of an arithmetic, logical or counting operation is 1, reset if the result is zero. Like the Zero flag and the Carry flag, it will control the action of several conditional instructions.

```
F2      JP          Jump if Plus
xx      low address (if high bit is 0)
yy      high address

FA      JM          Jump if Minus
xx      low address (if high bit is 1)
yy      high address

F4      CP          Call if Plus
xx      low address
yy      high address

FC      CM          Call if Minus
xx      low address
yy      high address

F0      RP          Return if Plus

F8      RM          Return if Minus
```

Like the other conditional instructions, these respond to a flag set

by one of the arithmetic or logical instructions (also DAA, INR and DCR), not to the present content of the Accumulator.

10.7.1.5 Overflow

Twos complement representation permits addition, subtraction, multiplication and division of signed numbers, giving correct results in twos complement form, correctly signed, provided that the magnitude of the result does not exceed the allowed range for the number of bits used (-128 to +127 for one byte). In many applications the programmer can be certain that the limits will not be exceeded. If results reach the limits, however, an "arithmetic overflow" will occur.

```
  40      0 1 0 0   0 0 0 0
+ 40      0 1 0 0   0 0 0 0
 -80      1 0 0 0   0 0 0 0
          ↑
          └── negative
```

There are two ways of treating this problem. One is simply to provide additional capacity. If two byte numbers are used, only the highest bit of the high byte represents the sign, and values from 0 to + 32767 and - 1 to - 32768 can be represented.

```
  40      0 0 0 0   0 0 0 0   0 1 0 0   0 0 0 0
+ 40      0 0 0 0   0 0 0 0   0 1 0 0   0 0 0 0
  80      0 0 0 0   0 0 0 0   1 0 0 0   0 0 0 0
          ↑
          └── still positive
```

With multiple precision arithmetic this can be carried to as many bytes as are necessary.

Another way of handling arithmetic overflow is to test for it. If two positive numbers are added and the result is negative, an overflow has occurred. If two negative numbers are added producing a positive result, an overflow has occurred. Subtraction of numbers with the same sign or addition of numbers with different signs cannot produce overflow. In most cases where only addition and subtraction are required, it is easier to provide additional storage capacity so that overflow cannot occur, but for multiplication and division the test for overflow is likely to be necessary.

10.7.2 Change Sign, Add, Subtract Exercise

Write a program that will accept a binary number of two bytes, and on command do one of the following:

NEXT key: Store the number as entered.
STEP key: Change the sign of the number and store it.
RUN key: Subtract the number from the previously stored value.
ADDR key: Add the number to the previously stored value.
CLR key: Clear the stored value.

After each entry display the result. If the result is negative, display its twos complement with a minus sign. A flow chart and coding sheets are presented in Figure 10-22 through 10-26.

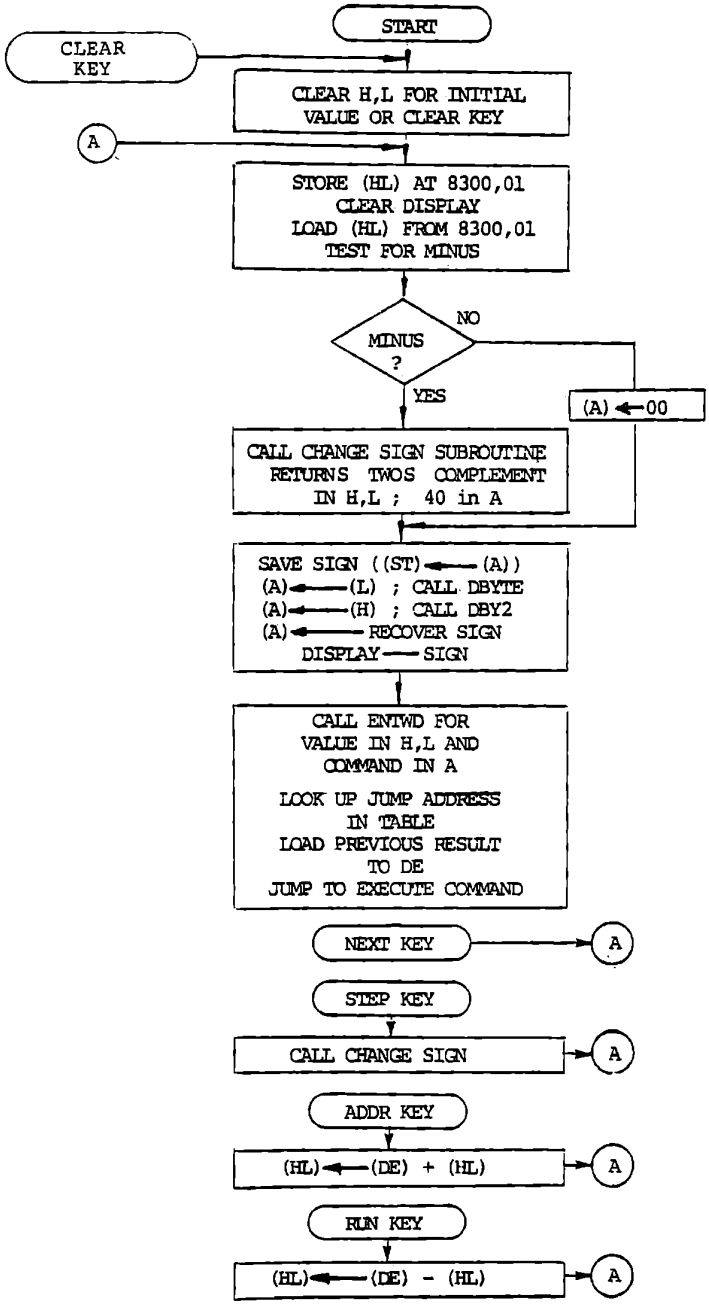

START
CLEAR KEY
CLEAR H,L FOR INITIAL VALUE OR CLEAR KEY
A
STORE (HL) AT 8300,01
CLEAR DISPLAY
LOAD (HL) FROM 8300,01
TEST FOR MINUS
MINUS ?
NO
YES
(A) ← 00
CALL CHANGE SIGN SUBROUTINE RETURNS TWOS COMPLEMENT IN H,L ; 40 in A
SAVE SIGN ((ST) ← (A))
(A) ← (L) ; CALL DBYTE
(A) ← (H) ; CALL DBY2
(A) ← RECOVER SIGN
DISPLAY ← SIGN
CALL ENTWD FOR VALUE IN H,L AND COMMAND IN A
LOOK UP JUMP ADDRESS IN TABLE
LOAD PREVIOUS RESULT TO DE
JUMP TO EXECUTE COMMAND
NEXT KEY
A
STEP KEY
CALL CHANGE SIGN
A
ADDR KEY
(HL) ← (DE) + (HL)
A
RUN KEY
(HL) ← (DE) - (HL)
A

Figure 10-22

ADDR		CODE															
8 20	0	2	1		L	X	I		H	,	0	0	0	0		To clear initial
	1	0	0													value
	2	0	0													
	3	2	2		S	H	L	D			8	3	0	0		Store result
	4	0	0													
	5	8	3													
	6	C	D		C	A	L	L			C	L	E	A	R	Clear display
	7	8	7													
	8	0	2													
	9	2	A		L	H	L	D			8	3	0	0		Load result
	A	0	0													
	B	8	3													
	C	A	F		X	R	A		A							Test for minus
	D	8	4		A	D	D		H							
	E	3	E		M	V	I		A	,	0	0				Blank if positive
	F	0	0													
8 21	0	0	0		N	O	P									
	1	F	C		C	M		C	H	S	I	G	N			Returns two's
	2	9	0													complement of HL
	3	8	2													and (A) = 4B
	4	F	5		P	U	S	H		P	S	W				Save symbol for sign
	5	7	D		M	O	V		A	,	L					(A) ← low byte
	6	C	D		C	A	L	L		D	B	Y	T	E		Display low byte
	7	9	5													
	8	0	2													
	9	7	C		M	O	V		A	,	H					(A) ← high byte
	A	C	D		C	A	L	L		D	B	Y	2			Display high byte
	B	9	8													
	C	0	2													
	D	F	1		P	O	P		P	S	W					Recover sign
	E	1	2		S	T	A	X		D						Display sign
	F	0	0		N	O	P									
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 10-23

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

CHANGE SIGN EXERCISE - DATA ENTRY AND COMMAND INTERPRETATION

ADDR		CODE		LABEL	MNEMONIC	OPERANDS	COMMENTS
8 22	0	C	D		CALL	ENTWD	
	1	4	6				
	2	0	3				
	3	E	B		XCHG		(DE) ← Data
	4	2	1		LXI	H, 8220	Jump table address
	5	2	0				-10
	6	8	2				
	7	8	5		ADD	L	Add key value
	8	6	F		MOV	L, A	(10-17)
	9	6	E		MOV	L, M	Get jump address
	A	E	5		PUSH	H	(ST) ← jump addr
	B	2	A		LHLD	8300	(HL) ← old data
	C	0	0				
	D	8	3				(DE) ← old data
	E	E	B		XCHG		(HL) ← new data
	F	C	9		RET		Jump to execute
8 23	0	0	6		MEM		Ignore entry
	1	0	6		REG		Ignore entry
	2	7	6		ADDR	(ADD)	
	3	7	0		STEP	(CHANGE SIGN)	
	4	8	2		RUN	(SUBTRACT)	
	5	0	3		NEXT	(STORE)	
	6	0	6		BRK		Ignore entry
	7	0	0		CLR	(CLEAR)	
	8						
	9						
	A						
	B						
	C						
	D						
	E						
	F						
8	0						
	1						
	2						
	3						
	4						
	5						
	6						
	7						
	8						Figure 10-24

ADDR	CODE			LABEL	OPERATION	OPERAND	COMMENT
8 27 0	C	D		CALL		CHSIGN	STEP key
1	9	0					
2	8	2					
3	C	3			JMP	8203	to store result
4	0	3					
5	8	2					
6	7	B			MOV	A, E	ADD
7	8	5			ADD	L	NOTE: DAD D
8	0	0			NOP		COULD DO ALL
9	6	F			MOV	L, A	THIS INSTEAD
A	7	A			MOV	A, D	
B	8	C			ADC	H	
C	0	0			NOP		
D	6	7			MOV	H, A	
E	C	3			JMP	8203	
F	0	3					
8 28 0	8	2					
1	0	0			NOP		
828 2	7	B			MOV	A, E	SUBTRACT
3	9	5			SUB	L	
4	0	0			NOP		
5	6	F			MOV	L, A	
6	7	A			MOV	A, D	
7	9	C			SBB	H	
8	0	0			NOP		
9	6	7			MOV	H, A	
A	C	3			JMP	8203	
B	0	3					
C	8	2					
D	0	0			NOP		
E	0	0			NOP		
F	0	0			NOP		
3 0							
1							
2							
3							
4							
5							
6							
7							
8							Figure 10-25

CHANGE SIGN SUBROUTINE

ADDR		CODE													
8 29	0	7	D		M	O	V		A	,	L				CHSIGN
	1	2	F		C	M	A								Complement
	2	C	6		A	D	I		0	1					Add 1 for less
	3	0	1												significant byte
	4	6	F		M	O	V		L	,	A				(L) ← LSB
	5	7	C		M	O	V		A	,	H				(A) ← MSB
	6	2	F		C	M	A								
	7	C	E		A	C	I		0	0					Add CY
	8	0	0												
	9	6	7		M	O	V		H	,	A				(H) ← MSB
	A	3	E		M	V	I		A	,	4	0			(A) ← minus sign
	B	4	0												
	C	C	9		R	E	T								
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-26

10.7.3 Signed Decimal Numbers

Decimal numbers are commonly represented in "sign and magnitude" form in computers as well as on paper. Often one bit in a number occupying two or more bytes is used to designate the sign:

```
+ 1327     0  001  0011     0010  0111
- 1327     1  001  0011     0010  0111
```

A signed two byte packed decimal number then has a possible range of + 7999 to -7999.

This representation is convenient for several reasons. A number expressed in sign and magnitude form is easily displayed, with the sign bit controlling only the display of a minus sign. Similarly at data entry, if a minus key or change sign key is pressed it is easy to complement the high bit. Also, multiplicaton is easy, since the magnitudes can be multiplied without regard to sign and the signs combined by the exclusive OR function.

Unfortunately, this representation is not convenient for addition and subtraction of signed decimal numbers. The Decimal Adjust Accumulator (DAA) instruction of the 8080 does not work correctly for the result of a subtraction. Therefore, it is necessary to convert signed decimal numbers into "hundreds complement" form before addition. Subtraction is performed by changing the sign of the number to be subtracted, and adding the negative number.

"Hundreds complement" properly should refer only to a two digit number: It is the sum of 100 (decimal) plus or minus the number,

with the carry discarded:

```
100  +  16   = 16
100  -  16   = 84
```

In this form signed numbers can be added, giving correct results in hundreds complement form.

```
  36  = 100 + 36  =     36
- 14  = 100 - 14  =     86
= 22  = 100 + 22  = (1) 22
```

This is also effective when the result is negative:

```
  - 36  = 100 - 36  =     64
  + 14  = 100 + 14  =     14
= - 22  = 100 - 22  =     78
```

Provided that 78 is recognized as a negative value, this answer is correct. Because of the limited range of a one byte signed number, decimal values almost always require more than one byte; typically at least three and often as many as seven bytes are used.

A two byte decimal number in "hundreds complement" form is really 10000 plus or minus the number. A three byte number in hundreds complement is 1,000,000 plus or minus the number.

```
   1327  = 1000000  +  1327  =  00  00  13  27
 - 6564  = 1000000  -  6564  =  99  99  33  36
= - 5237 = 1000000  -  5237  =  99  99  46  63
```

When enough bytes are allocated for the number it becomes reasonable

to define as negative any number whose high bit is one. This gives an unbalanced range of possible values:

+ 799999 to - 199999

In general, hundreds complement numbers must occupy one more byte than their sign and magnitude representation. Sometimes this byte is assigned in memory and used for the sign; then the other bytes can have a full range of decimal values:

+ 999999 to - 999999

10.7.3.1 Signed Decimal Arithmetic Exercise

We shall develop a program to accept a six digit decimal number from the keyboard, change its sign if desired, add it to a previously stored value, and display the result.

The new data and the result will be stored in memory in this form:

Result	New Data	
8300	8304	Sign Byte
8301	8305	High Byte
8302	8306	Mid Byte
8303	8307	Low Byte

Note that we have reversed the usual storage sequence here. The only disadvantage to this is that LHLD and SHLD would not load these data into HL in the normal high-low sequence; with multi-byte numbers we would be unlikely to load variable data that way. The sequence given is sometimes more convenient than the normal sequence.

We shall define a group of subroutines to handle the data process requirements. Most of these operate on only one set of data; they are to be entered with (HL) addressing the sign byte of the number to be processed and must return that address unchanged.

CLRMEM: Clear the four bytes of memory from (HL) through (HL) + 3.

SIXKEY: Accept and display six decimal keys. Pack the entered data into three bytes from (HL) + 1 through (HL) + 3. Ignore keys A-F. Return when a command is entered with (A) = command key.

DISPLAY: Display a three byte packed decimal number. If the sign byte is negative display a minus sign.

CHSIGN: Change the sign of a number in sign and magnitude form by complementing the high bit of its sign byte.

HUNCP: Convert a three byte magnitude with a one byte sign into a four byte hundreds complement.

DECADD: Add two decimal numbers in four byte hundreds complement form, replacing the augend with the sum:

(HL) addresses the augend
(DE) addresses the addend

SIGNMAG: Convert a four byte hundreds complement numer to three byte sign with one byte magnitude.

The main program will provide the addresses to be used by the subroutines, exchanging (HL) with the stack top as required. The main program is shown in a functional form below.

```
START:    CALL CLRMEM (addressing result)
LOOP:     CALL SIXKEY (addressing new data)
          IF COMMAND = STEP
                CALL CHSIGN (addressing new data)
          IF COMMAND = CLR
                CALL CLRMEM (addressing result)
          CALL HUNCP (addressing result)
          CALL HUNCP (addressing new data)
          CALL DECADD (addressing both)
          CALL SIGNMAG (addressing result)
          CALL DISPLAY (addressing result)
          JMP  LOOP
```

Use the top down approach, coding and testing the main program first, and then each of the subroutines. It is suggested that DISPLAY be the first subroutine, since it can be used by SIXKEY to display the new data.

We have not yet described the process for generating a hundreds complement; this is covered in Section 10.7.3.2. The data entry, display and addition subroutines require only reasonably familiar programming techniques. Two hints that may be useful in your development:

Entering keys: Use GETKY (023D) to obtain a key. This returns Not

Carry if a command is entered; use RNC or JNC after GETKY. The key value is returned in Register A and in C; Register B is cleared. Shift the old data left four bits and enter the new key. You can keep the data in registers while it is being entered, and store it only at the command key, but the given solution always keeps the data in memory.

Displaying data: Address the low byte and the right display digit. Load the byte, decrement HL, and call DBY2 (0298) three times to display the three data bytes. Register B can be used as a counter.

At exit from this loop, register pair DE addresses the next blank digit, and HL again addresses the sign byte. Its high bit is 1 if the number is negative. This will display a minus sign or a blank:

```
MOV  A,M
RAR
ANI  40
STAX D
```

Without subroutine CHSIGN, HUNCP and SIGNMAG this program will work for positive numbers. Develop the program using stubs for these three subroutines. A solution is given on the following pages.

CODING SHEET — MICROCOMPUTER TRAINING SYSTEM — INTEGRATED COMPUTER SYSTEMS

ADDR		CODE														
8 20	0	21			L	X	I		H	,	8	3	0	0	Clear Result
	1	00													
	2	83													
	3	CD			C	A	L	L		C	L	R	M	E	M
	4	30													
	5	82													
820	6	E5			P	U	S	H		H					
	7	21			L	X	I		H	,	8	3	0	4	(Address) new data
	8	04													
	9	83													
	A	CD			C	A	L	L		S	I	X	K	E	Y
	B	60													
	C	82													
	D	FE			C	P	I		S	T	E	P			
	E	13													
	F	CC			C	Z			C	H	S	I	G	N	
8 21	0	B0													
	1	82													
	2	E3			X	T	H	L							(HL) ← 8300
	3	FE			C	P	I		C	L	R				
	4	17													
	5	CC			C	Z			C	L	R	M	E	M	Clear old result
	6	30													
	7	82													
	8	CD			C	A	L	L		H	U	N	C	P	
	9	C0													
	A	82													
	B	E3			X	T	H	L							(HL) ← 8304
	C	CD			C	A	L	L		H	U	N	C	P	
	D	C0													
	E	82													
	F	EB			X	C	H	G							(DE) ← 8304
8 22	0	E1			P	O	P		H						(HL) ← 8300
	1														
	2				(	C	O	N	T	I	N	U	E	D	)
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-27a

ADDR	CODE															
8221	CD			C	A	L	L		D	E	C	A	D	D		
2	90															
3	82															
4	CD			C	A	L	L		S	I	G	N	M	A	G	
5	E0															
6	82															
7	CD			C	A	L	L		D	I	S	P	L	A	Y	
8	40															
9	82															
A	C3			J	M	P			8	2	0	6				
B	06															
C	82															
D																
E																
F																
80																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																
80																
1																
2																
3																
4																
5																
6																
7																
8																

Figure 10-27b

CLRMEM

ADDR	CODE											
8230	C5		PUSH		B							
1	01		LXI		B,	0004						Clear 4 Bytes
2	04											
3	00											
4	09		DAD		B							Address 8304 or 8308
8233	2B		DCX		H							Next lower byte
6	36		MVI		M,	00						Store 00
7	00											
8	0D		DCR		C							Count bytes
9	C2		JNZ			8235						
A	35											
B	82											
C	C1		POP		B							
D	C9		RET									
E												
F												
8 0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
A												
B												
C												
D												
E												
F												
8 0												
1												
2												
3												
4												
5												
6												
7												
8												

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

Figure 10-27c

ADDR		CODE	DISPLAY	
8 24	0	F5	PUSH PSW	
	1	D5	PUSH D	
	2	C5	PUSH B	
	3	01	LXI B, 0003	
	4	03		
	5	00		
	6	09	DAD B	Address low byte
	7	41	MOV B, C	(B) ← byte count
	8	11	LXI D, 83FF	Address right
	9	FF		digit
	A	83		
824	B	7E	MOV A, M	Display 3 bytes
	C	CD	CALL DBY2	
	D	98		
	E	02		
	F	2B	DCX H	
8 25	0	05	DCR B	
	1	C2	JNZ 824B	
	2	4B		
	3	82		
	4	7E	MOV A, M	Sign byte
	5	1F	RAR	Sign to Bit 6
	6	E6	ANI 40	Clear other bits
	7	40		and display
	8	12	STAX D	blank or minus)
	9	C1	POP B	
	A	D1	POP D	
	B	F1	POP PSW	
	C	C9	RET	
	D			
	E			
	F			
8	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			
	8			

Figure 10-27d

ADDR		CODE														
8 26	0	C	D		C	A	L	L		C	L	R	M	E	M	
	1	3	0												Clear 8304-8307	
	2	8	2													
826	3	C	D		C	A	L	L		G	E	T	K	V		
	4	3	D													
	5	0	2													
	6	D	0		R	N	C									
	7	F	E		C	P	I		0	A						
	8	0	A													
	9	D	2		J	N	C			8	2	6	3			
	A	6	3													
	B	8	2													
	C	2	3		I	N	X		H						Skip sign byte	
	D	7	E		M	O	V		A	,	M				(A) ← high byte	
	E	2	3		I	N	X		H							
	F	5	6		M	O	V		D	,	M				(D) ← mid byte	
827	0	2	3		I	N	X		H							
	1	5	E		M	O	V		E	,	M				(E) ← low byte	
	2	E	B		X	C	H	G							(HL) ← mid and low	
	3	2	9		D	A	D		H						Shift (AHL)	
	4	1	7		R	A	L								left 4 bits	
	5	2	9		D	A	D		H							
	6	1	7		R	A	L									
	7	2	9		D	A	D		H							
	8	1	7		R	A	L									
	9	2	9		D	A	D		H							
	A	1	7		R	A	L									
	B	0	9		D	A	D		B						Adding new digit	
	C	E	B		X	C	H	G								
	D	7	3		M	O	V		M	,	E				(8307) ← low byte	
	E	2	B		D	C	X		H							
	F	7	2		M	O	V		M	,	D				(8306) ← mid byte	
828	0	2	B		D	C	X		H							
	1	7	7		M	O	V		M	,	A				(8305) ← high byte	
	2	2	B		D	C	X		H						Address sign	
	3	C	D		C	A	L	L		D	I	S	P	L	AY	
	4	4	0													
	5	8	2													
	6	C	3		J	M	P			8	2	6	3			
	7	6	3													
	8	8	2													

Figure 10-27e

INTEGRATED COMPUTER SYSTEMS — MICROCOMPUTER TRAINING SYSTEM — CODING SHEET

DECADD

ADDR		CODE	Instruction	Comment
8 29	0	01	LXI B, 0004	
	1	04		(C)← byte count
	2	00		
	3	09	DAD B	Address beyond
	4	EB	XCHG	low byte of
	5	09	DAD B	each number
	6	EB	XCHG	
	7	AF	XRA A	Clear Carry
829	8	1B	DCX D	Loop - Address
	9	2B	DCX H	next lower bytes
	A	1A	LDAX D	((HL))←((HL))+((DE))
	B	8E	ADC M	for four bytes
	C	27	DAA	
	D	77	MOV M, A	
	E	0D	DCR C	
	F	C2	JNZ 8298	
8 2A	0	98		
	1	82		
	2	C9	RET	
	3			
	4			
	5			
	6			
	7			
	8			
	9			
	A			
	B			
	C			
	D			
	E			
	F			
8	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			
	8			

Figure 10-27f

STUBS

ADDR	CODE														
82B 0	C	9		R	E	T									CHSIGN
1															
2															
3															
4															
5															
6															
7															
8															
9															
A															
B															
C															
D															
E															
F															
82C 0	C	9		R	E	T									HUNCP
1															
2															
3															
4															
5															
6															
7															
8															
9															
A															
B															
C															
D															
E															
F															
82E 0	C	9		R	E	T									SIGNMAG
1															
2															
3															
4															
5															
6															
7															
8															

Figure 10-27g

This page intentionally left blank.

10.7.3.2 Hundreds Complement

The decimal arithmetic program as developed so far is useful for demonstrating hundreds complement arithmetic. Run the program and enter a number:

```
RUN
1   2   2   NEXT                          00  0122
```

Add 1 to this number:

```
1   NEXT                                  00  0123
```

Now add 999999 to this result:

```
9   9   9   9   9   9   NEXT              00  0122
```

We can see here that 999999 is equivalent to -1. Try other values close to this. You can see that a negative number is equivalent to 1,000,000 minus the magnitude of the number.

The reason for using hundreds complement is in part the fact that the 8080 cannot do a decimal subtraction. Therefore we cannot perform the conversion in the simple way that we did a twos complement.

The conversion of a negative decimal number to hundreds complement form involves a series of steps. First the decimal number is subtracted from decimal 100, represented as 9A (= 90 + 10). Now to make the following DAA correct we must add zero to this value, because the subtraction has improperly set the Auxiliary Carry flag, which controls the DAA operation. Adding zero clears the flag and DAA generates a correct hundreds complement decimal value. If the value

is greater than 99 a carry is generated.

For successive bytes the carry is added to 99 (without DAA) to give 99 or 9A. Then the next byte is subtracted from this value; once again zero is added and DAA is executed. The program segment below makes the conversion for a four digit (two byte) decimal number in (HL).

```
MVI  A,9A        (A) < - 100 decimal
SUB  L           Subtract low byte
ADI  00          Correct flags
DAA              Decimal adjust
MOV  L,A         (L) < - low byte
MVI  A,99        (A) < - 99
ACI  00          (A) < - 9A if Carry
SUB  H           Subtract high byte
ADI  00          Correct flags
DAA              Decimal adjust
MOV  H,A         (H) < - high byte
```

This is coded, with calls to data entry and display subroutines, in Figure 10-28. You may want to experiment with this before going on to a multi-byte subroutine for the decimal arithmetic program.

TWO BYTE HUNDREDS COMPLEMENT

ADDR		CODE														
8 00	0	CD		CALL				ENTWZ								(HL) ← Keys
	1	49														If no key keys
	2	03														(HL) is preserved
	3	CD		CALL				DWORD								
	4	D1														Display at left
	5	02														if nothing entered
	6	3E		MVI				A, 9A								= 100 decimal
	7	9A														
	8	95		SUB				L								Subtract low byte
	9	C6		ADI				00								Correct flags
	A	00														
	B	27		DAA												Get decimal value
	C	6F		MOV				L, A								
	D	3E		MVI				A, 99								
	E	99														99 if no carry
	F	CE		ACI				00								9A if carry
8 01	0	00														
	1	94		SUB				H								Subtract high byte
	2	C6		ADI				00								Correct flags
	3	00														
	4	27		DAA												Get decimal value
	5	67		MOV				H, A								
	6	11		LXI				D, 83FF								Display hundreds
	7	FF														complement at
	8	83														right
	9	CD		CALL				DWD2								
	A	D4														
	B	02														
	C	C3		JMP				8000								
	D	00														
	E	80														
	F															
8	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure 10-28

10.7.3.3 Subroutine HUNCP and CHSIGN

Now let us see how this technique will be applied for the data structure we have adopted for the decimal arithmetic program. First we shall test the sign byte; if it is positive no conversion is needed except that the sign byte must be set to zero. If the number is marked negative we must make the conversion; here also the sign byte should be set to zero, to be converted to 99 in the four byte hundreds complement number.

```
MOV  A,M        (A) < - Sign byte
MVI  M,00       Clear for highest byte
RAL             Sign bit to Carry
RNC             Exit if positive
```

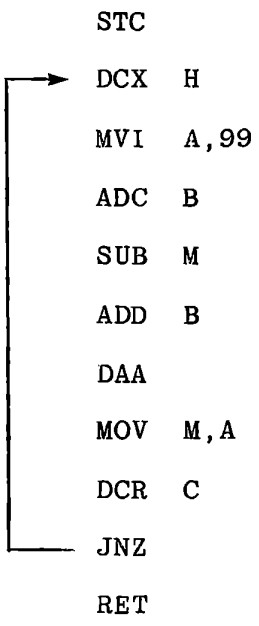
We shall use a loop to make the conversion, so we must load a counter. We also must address the least significant byte.

```
LXI B,0004
DAD B
```

Now Register C contains the count to convert four bytes, and (HL) addresses the byte beyond the least significant. We shall decrement the address as the first step in the loop, so the conversion will be done for the appropriate four bytes and at the end the original content of HL will be restored.

Since we have loaded Register B with zero for the DAD B we can ADD B and ADC B instead of ADI 00 and ACI 00. By setting Carry before entering the loop we can use MVI A,99; ADC B for the least

significant byte as well as for the higher bytes.



To use negative numbers we must also provide subroutine CHSIGN. This is entered with HL addressing the sign byte; we are to complement the most significant bit. Recall that the main program makes another test on the command in Register A after calling CHSIGN, so this subroutine must preserve Register A.

These two subroutines are given in Figure 10-29. HUNCP also performs the function defined for SIGNMAG, properly converting a four byte hundreds complement number to sign and magnitude. Figure 10-29 shows a jump from 82E0 to 82C0 to use the same subroutine.

Experiment with the program to show that it correctly adds and subtracts decimal numbers, provided that the sum or difference has a magnitude no greater than 999,999.

CHSIGN

ADDR		CODE													
8 2B	0	F	5		PUSH		PSW								
	1	7	E		MOV		A,	M							
	2	E	E		XRI		80								
	3	8	0												
	4	7	7		MOV		M,	A							
	5	F	1		POP		PSW								
	6	C	9		RET										
	7														
	8														
	9														
	A				CHANGE		SIGN		SUBROUTINE						
	B				FOR		DECIMAL		ARITHMETIC						
	C				PROGRAM		OF		FIGURE 10-27						
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure 10-29a

HUNCP

ADDR		CODE												
8 2C	0	7	E		MOV		A,	M						
	1	3	6		MVI		M,	0	0					
	2	0	0											
	3	1	7		RAL									
	4	D	0		RNC									
	5	0	1		LXI		B,	0	0	0	4			
	6	0	4											
	7	0	0											
	8	0	9		DAD		B							
	9	3	7		STC									
82C	A	2	B		DCX		H							
	B	3	E		MVI		A,	9	9					
	C	9	9											
	D	8	8		ADC		B							
	E	9	6		SUB		M							
	F	8	0		ADD		B							
8 2D	0	2	7		DAA									
	1	7	7		MOV		M,	A						
	2	0	D		DCR		C							
	3	C	2		JNZ			8	2	C	A			
	4	C	A											
	5	8	2											
	6	C	9		RET									
	7													
	8				HUNDREDS									COMPLEMENT
	9				SUBROUTINE									FOR
	A				DECIMAL									ARITHMETIC
	B				PROGRAM									OF FIGURE 10-27
	C													
	D													
	E													
	F													
8 2E	0	C	3		JMP			8	2	C	0			SIGNMAG
	1	C	0											Jump to HUNCP
	2	8	2											for SIGNMAG
	3													
	4													
	5													
	6													
	7													
	8													

Figure 10-29b

10.7.3.4 Decimal Overflow

The possibility of overflow was discussed in Section 10.7.1.5. The same problem obviously exists in decimal arithmetic, since we still have a limit to the range of a number. Since we are using one more byte for the arithmetic than for storing numbers, the highest byte will always have a value of either 00 or 99 unless overflow has occurred.

If the sum (or difference) is within the three byte range HUNCP will not change the sign byte. If the sum is between zero and +999999 the highest byte of the sum is zero; HUNCP inserts zero. If the sum is negative, but not more negative than -999999, then the highest byte is 99. HUNCP inserts zero but then converts it back to 99.

Now consider three cases where overflow occurs.

a) Addition of two positive numbers:

```
  00  50  00  00
+ 00  50  00  00
= 01  00  00  00
```

Here the highest byte is 1; HUNCP makes it 0.

b) Addition of two negative numbers; sum 0:

```
  99  50  00  00          (-500000)
+ 99  50  00  00          (-500000)
= 99  00  00  00          (-0)
```

Here the highest byte is 99. HUNCP makes it zero and converts this number to all zeros.

c) Addition of two negative numbers; sum > 0:

```
  99  50  00  00          (-500000)
+ 99  49  99  99          (-500001)
= 98  99  99  99         (-1000001)
```

Now the highest byte is 98. HUNCP inserts zero and converts this number to -1, stored as 99 00 00 01.

For each case of overflow, HUNCP has changed the content of the highest byte in converting the number to sign and magnitude. Figure 10-30 shows a test for this change. If there is no overflow a blank is displayed in the left digit, but if a change occurs a symbol (E.) is displayed to indicate "Error."

ADDR	CODE													
82E 0	7	E		MOV			A,	M						Save highest byte
1	F	5		PUSH				PSW						
2	C	D		CALL				HUNCP						If negative change to sign and magnitude
3	C	0												
4	8	2												
5	F	1		POP			PSW							HUNCP does not
6	9	6		SUB			M							change highest
7	C	A		JZ				82EC						byte unless
8	E	C												overflow has
9	8	2												occurred
A	3	E		MVI			A,	F9						If overflow
B	F	9												display E
82E C	3	2		STA				83F8						Display blank
D	F	8												or E. at left
E	8	3												
F	C	9		RET										
8 0														
1														
2				CONVERT				HUNDREDS						
3				COMPLEMENT				TO SIGN						
4				AND			MAGNITUDE,							WITH
5				TEST			FOR		OVERFLOW					
6														
7														
8														
9														
A														
B														
C														
D														
E														
F														
8 0														
1														
2														
3														
4														
5														
6														
7														
8														

Figure 10-30

10.7.4 Fractional Numbers

A fractional value in the decimal number system is expressed by digits to the right of a decimal point:

```
0.1   =  1/10
0.01  =  1/100
0.11  =  1/10 + 1/100  =  11/100
```

In the binary number system fractional values are also expressed by digits to the right of a binary point:

```
0.1   =  1/2  =  1/10
0.01  =  1/4  =  1/100
0.11  =  1/2 + 1/4  =  3/4  =  11  /100
```

The beauty of this representation is that all the arithmetic operations of integer numbers apply equally to fractional numbers and mixed numbers:

```
  3  10/16              0 0 1 1 . 1 0 1 0
+ 4   7/16            + 0 1 0 0 . 0 1 1 1
= 8   1/16            = 1 0 0 0 . 0 0 0 1
```

Twos complement still works with fractional values:

```
- 3  10/16              1 1 0 0 . 0 1 1 0
+ 4   7/16              0 1 0 0 . 0 1 1 1
= 0  13/16              0 0 0 0 . 1 1 0 1
```

Computers use two binary point systems, fixed point and floating point. The examples above are fixed point. Each number has its

binary point in the same place. Generally multi-byte precision is needed in real problems, and the binary point lies between two of the bytes. A four byte number can represent any value from - 32768.0 to 32767.9999847 with a precision of .0000152 (one part in 65536).

For many purposes floating point numbers are much more satisfactory. This is equivalent to scientific notation with the number represented as a fraction times the number system base raised to a power:

$$0.9876 \times 10^{4} = 9876$$

To avoid the difficulties of showing exponents in print this is often shown as:

0.9876 E04

where E represents "10 with exponent".

Scientific notation is very convenient for multiplication and division. The two fractions are multiplied (or divided) and the exponents are added (or subtracted):

```
   0.9000 E04
x  0.2000 E02
=  0.1800 E06
```

For addition and subtraction, however, the numbers must be converted to fixed point format:

```
  0.9000 E04 = 9000.0000
+ 0.2000 E02 = 0020.0000
= 0.9020 E04 = 9020.0000
```

These same techniques apply to binary numbers in the computer. In a computer as on paper, the fraction (or mantissa) must be stored separately from the exponent. Each can be positive or negative, and expressed in twos complement or sign and magnitude form. Generally a computing system that is doing floating point arithmetic will operate in binary form, converting from decimal at input and to decimal at output. Decimal/binary/decimal conversions are treated in Appendix B.

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

CHAPTER 11

REVIEW

11. REVIEW

You have now met all of the instructions of the 8080, and actually used most of them. We will review the instruction set and look at the code structure and flags. The instructions can be divided into several categories:

a) Data Transfer Instructions

b) Counting Instructions

c) Accumulator/Carry Instructions

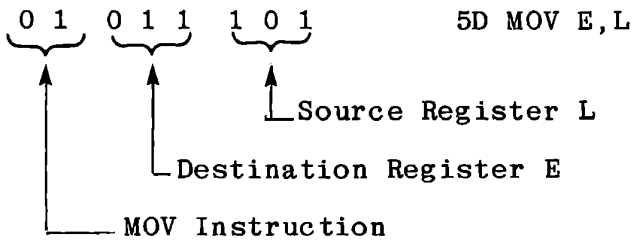
d) Arithmetic and Logical Instructions

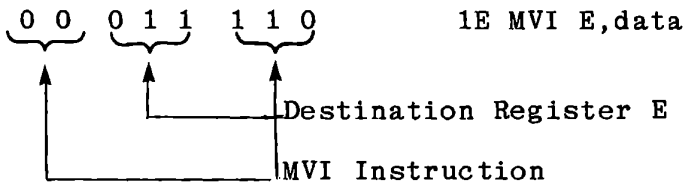
e) Branch Instructions

f) Input/Output Instructions

11.1 DATA TRANSFER

Data transfer instructions include MOV, MVI, STA, etc. All register reference instructions in the 8080 conform to a pattern in which three bits identify a source, or else a different three bits identify a destination, or both.





Other data transfer instructions are the eight instructions that load and store the accumulator and register pair H,L:

3A	LDA	yyxx	32	STA	yyxx
0A	LDAX	B	02	STAX	B
1A	LDAX	D	12	STAX	D
2A	LHLD	yyxx	22	SHLD	yyxx

The four LXI instructions:

01	LXI	B
11	LXI	D
21	LXI	H
31	LXI	SP

The stack instructions:

C5	PUSH	B	C1	POP	B
D5	PUSH	D	D1	POP	D
E5	PUSH	H	E1	POP	H
F5	PUSH	PSW	F1	POP	PSW

The register pair transfer instructions:

EB	XCHG	(DE) <-> (HL)
E3	XTHL	(ST) <-> (HL)
F9	SPHL	(SP) <- (HL)
E9	PCHL	(PC) <- (HL)

REVIEW

The 8080 has an abundance of data transfer instructions, yet is lacking three needed functions that therefore require multiple instructions:

a) Exchange BC with HL

```
PUSH    B              (BC) <-> (HL)
PUSH    H
POP     B
POP     H
```

b) Initialize the stack to a new location and push the old stack pointer into the new stack.

```
LXI     H,0000
DAD     SP
LXI     SP,new location
PUSH    H
```

It is easier to restore the old value:

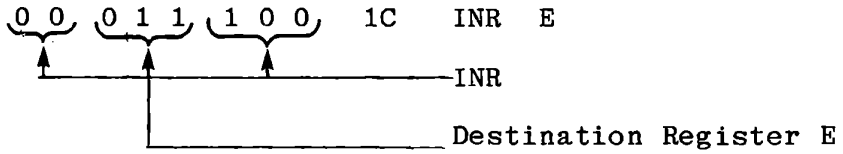
```
POP     H
SPHL
```

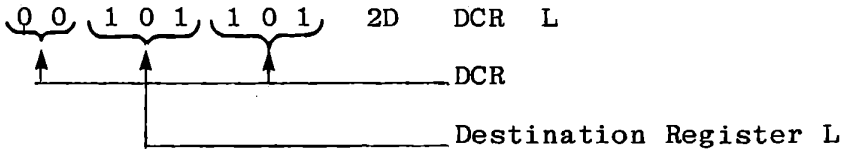
c) Save all registers and flags.

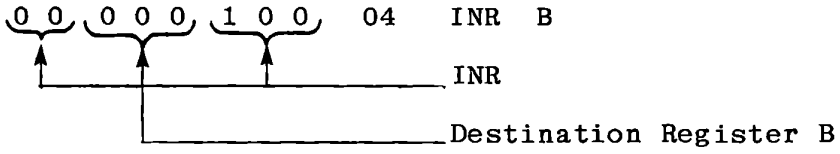
Some microprocessors have a single command that pushes all registers into the stack; others, such as the Intel 8048 have a duplicate set of registers. In the 8080 four instructions are needed. Data instructions do not affect any flags (Except POP PSW, which restores the flags to the state when PUSH PSW was executed).

11.2 COUNTING INSTRUCTIONS

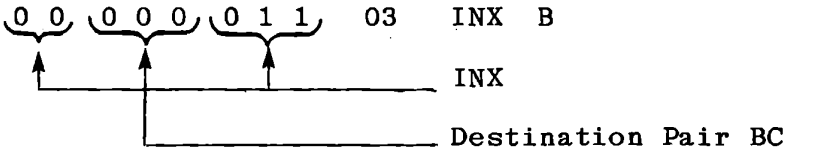
The INR and DCR instructions use the same register identification that appears in MOV.

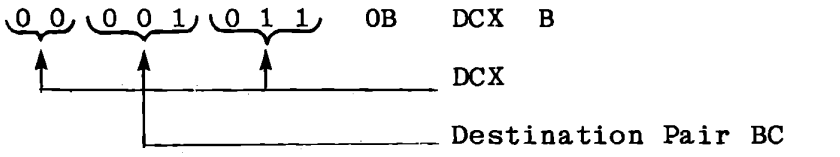






The structure is modified for register pair instruction





REVIEW

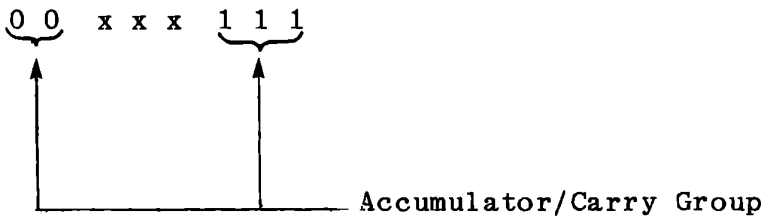
The counting instructions affect flags as follows:

INX:	No flags
DCX:	No flags
INR:	Set or clear zero, sign, parity Does not affect carry Set or clear auxiliary carry
DCR:	Set or clear zero, sign, parity Does not affect carry Set or clear auxiliary carry

Zero, sign and parity flags may be used to cause a conditional branch as a result of INR or DCR. INR or DCR may be used in a loop with ADC or SBB instructions, since carry is preserved.

11.3 ACCUMULATOR/CARRY INSTRUCTIONS

These instructions affect only the accumulator and flags. The instruction format is:



0	0	0	0	0	1	1	1	07	RLC
0	0	0	0	1	1	1	1	0F	RRC
0	0	0	1	0	1	1	1	17	RAL
0	0	0	1	1	1	1	1	1F	RAR
0	0	1	0	0	1	1	1	27	DAA
0	0	1	0	1	1	1	1	2F	CMA
0	0	1	1	0	1	1	1	37	STC
0	0	1	1	1	1	1	1	3F	CMC

REVIEW

The rotate instructions shift the accumulator left or right.

RLC Copies bit 7 to bit 0 and CY and shifts other bits left.

RRC Copies bit 0 to bit 7 and CY and shifts other bits right. Previous carry is lost.

RAL Copies bit 7 to CY, CY to bit 0 and shifts other bits left

RAR Copies bit 0 to CY, CY to bit 7 and shifts other bits right

STC Sets carry

CMC Complements carry

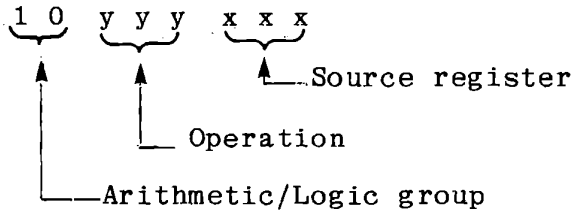
These instructions do not affect any flags except carry, even though execution may result in the accumulator containing zero or having a different sign or parity condition. To set or clear the flags to correspond to the content of the accumulator you must execute a logical or arithmetic instruction.

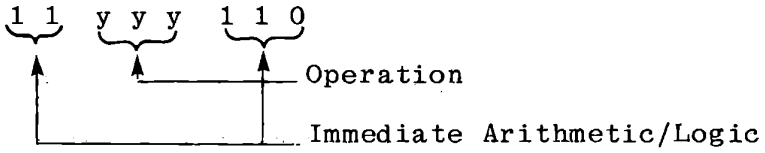
CMA complements the accumulator but affects no flags.

DAA corrects the result of an addition to decimal. It affects sign, zero, parity and carry flags. It may set carry but never clears carry.

11.4 ARITHMETIC AND LOGICAL INSTRUCTIONS

There are eight types of instructions and each has nine possible sources: the seven registers, the memory location addressed by (HL), and the program memory (the immediate instructions). As in the MOV instructions the three low bits designate the source. The next three bits specify which of the instructions is intended:





The operations designated by bits 5, 4, 3, are:

1 0	0 0 0	x x x	ADD	(A) <- (A) + (r)
1 0	0 0 1	x x x	ADC	(A) <- (A) + (r) + (CY)
1 0	0 1 0	x x x	SUB	(A) <- (A) - (r)
1 0	0 1 1	x x x	SBB	(A) <- (A) - (r) - (CY)
1 0	1 0 0	x x x	ANA	(A) <- (A) AND (r)
1 0	1 0 1	x x x	XRA	(A) <- (A) XOR (r)
1 0	1 1 0	x x x	ORA	(A) <- (A) OR (r)
1 0	1 1 1	x x x	CMP	(see below)

The same coding for the operation applies to the immediate instructions.

CMP r (or CMP M) performs a subtract operation and sets or clears the flags appropriately, but discards the result instead of storing it in the accumulator.

The four DAD instructions are also included in the arithmetic group. They are:

```
09   DAD  B        (HL) <- (HL) + (BC)
19   DAD  D        (HL) <- (HL) + (DE)
29   DAD  H        (HL) <- (HL) + (HL)
39   DAD  SP       (HL) <- (HL) + (SP)
```

These instructions affect only the carry flag. They can be used both for double precision arithmetic and to index a memory address. The latter is especially useful when operations are to be performed on bytes that are spaced from each other by some fixed or variable distance.

11.4.1 The Flags

The flag register (Processor Status Word, PSW) contains 5 bits. These are arranged as indicated below.

Bit	7	6	5	4	3	2	1	0
Flag	Sign	Zero	0	AC	0	Par	1	CY

The following list summarizes how these are affected by the various instructions:

Sign: Set if the high bit of the result is 1, cleared if 0, by the following instructions:

INR, DCR, DAA
Any arithmetic or logical instruction (except DAD).
Not affected by shift or complement instructions.

Zero: Set if the result is zero, cleared if not, by:

INR, DCR, DAA
Any arithmetic or logical instruction (except DAD).
Not affected by shift or complement instructions.

Parity: Set if parity of the result is even, cleared if odd, by:

INR, DCR, DAA
Any arithmetic or logical instruction (except DAD).
Not affected by shift or complement instructions.

Auxiliary Carry: Set if a carry or borrow occurs from bit 3 as a result of:

ADD, ADC, ADI, INR; cleared if the digit carry does not occur.

Also, set if no borrow occurs from bit 3 to bit 4 as a result of SUB, SBB, SUI, SBI, CMP, CPI, DCR; cleared if borrow occurs.

It is cleared by logical instructions (ANA, XRA, ORA, ANI, XRI, ORI).

Not affected by shift instructions.

Carry: Set or cleared by any shift or arithmetic operation, including:

CMP, DAD and DAA. Cleared by any of the logical instructions ANA, ORA, XRA.

Set by STC; complemented by CMC.

Not affected by count instructions.

11.5 BRANCH INSTRUCTIONS

Jump, Call, Return, Restart and PCHL are the branch instructions.

1 1 0 0 0 0 1 1	C3 JMP
1 1 0 0 1 0 0 1	C9 RET
1 1 0 0 1 1 0 1	CD CALL
1 1 1 0 1 0 0 1	E9 PCHL

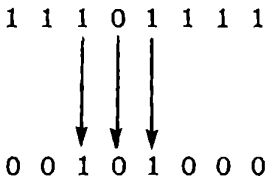
All of the branch instructions include 11 as the two high bits (bits 7 and 6) of the instruction. The three low bits distinguish among the branch, conditional branch, and various non-branching instructions. The conditional branches use bits 5 and 4 to determine which flag is to be tested and bit 3 to indicate whether the jump is to be executed when the flag is set or when it is clear.

1 1 x x x 0 1 0	Conditional Jump
1 1 x x x 1 0 0	Conditional Call
1 1 x x x 0 0 0	Conditional Return
0 0 0	If Not Zero
0 0 1	If Zero
0 1 0	If not Carry
0 1 1	If Carry
1 0 0	If Parity Odd
1 0 1	If Parity Even
1 1 0	If Plus
1 1 1	If Minus

REVIEW

The Restart instructions use the three bits 5, 4 and 3 as part of the address for the single byte CALL. They are copied into the corresponding three bits of the program counter while the remaining bits are all set to zero. For instance, RST 5 jumps to 0028:

EF RST 5



Note that a programmed RST instruction does not affect the interrupt enable flip flop. Interrupts are disabled by acknowledgement of an external interrupt, not by the instruction.

No flags are affected by any branch instruction.

11.6 INPUT/OUTPUT

```
DB    IN
xx    port address

D3    OUT
xx    port address
```

The port address is copied to both the high eight bits and the low eight bits of the address bus. I/O Read or I/O Write is activated. The CPU copies the data bus to (A) on input; copies (A) to the data bus on output. No flags are affected.

```
FB    Enable Interrupt
F3    Disable Interrupt
```

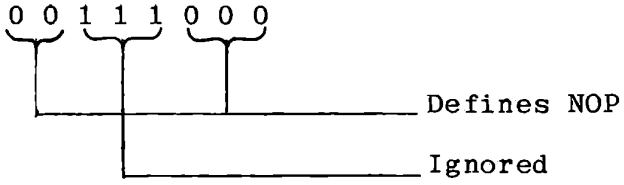
Set or clear the internal interrupt enabled flip-flop. EI is not effective until one instruction following EI has been executed. No flags are affected.

11.7 UNDEFINED INSTRUCTIONS

Twelve operation codes are "undefined" in the sense that Intel has not specified meanings for them. In fact they all translate into defined instructions because the instruction decoder ignores certain bits.

The following are all treated as NOP.

0 0	0 0 0 0 0 0 0 0	Defined as NOP
0 8	0 0 0 0 1 0 0 0	
1 0	0 0 0 1 0 0 0 0	
1 8	0 0 0 1 1 0 0 0	
2 0	0 0 1 0 0 0 0 0	(See 11.8.2)
2 8	0 0 1 0 1 0 0 0	
3 0	0 0 1 1 0 0 0 0	(See 11.8.2)
3 8	0 0 1 1 1 0 0 0	



CB acts as C3, JMP (Bit 3 is ignored).

D9 acts as C9, RET (Bit 4 is ignored).

DD, ED and FD all act as CD, CALL. (Bits 4 and 5 are ignored.)

11.8 OTHER MICROPROCESSORS

Various manufacturers other than Intel make exact equivalents of the 8080. In addition, there are several microprocessors that use the 8080 instruction set but are not equivalent.

11.8.1 NEC 8080A and NEC 8080AF

Nippon Electric manufactures the NEC 8080AF as an exact equivalent to the Intel 8080A. The NEC 8080A has a useful added feature that permits DAA to be effective for decimal subtraction as well as addition. It also executes the MOV instructions in four clock periods instead of five. Although these are desirable features the designer must recognize that if he uses them he will have no alternate source of supply for the microprocessor.

11.8.2 INTEL 8085

The 8085 uses the 8080 instruction set, and any program developed for the 8080 can be used without change. The 8085 includes the functions of the clock generator (8224) and system controller (8228) in the microprocessor chip. Hardware interfaces are different. In the 8085 the data bus is time multiplexed with the low eight bits of the address bus. This requires either special memory chips or additional interface hardware.

The 8085 has five separate interrupt inputs. One is exactly equivalent to the 8080 interrupt input. The other four are directly vectored, and have a defined priority sequence, so RST instructions need not be used. The interrupts can be masked independently of each other. Two of the undefined instructions in the 8080 set (op codes 20 and 30) are used to read and set the masks, and also to read and transmit serial data through separate pins.

The 8085 operates with a 1.3 microsecond clock, the same speed as the fastest version of the 8080A, but does not require high speed memory devices. It requires only a single power supply (+ 5 volts.)

All of these features make the 8085 very attractive, especially for small systems. Having learned to program the 8080, you are fully prepared to use the 8085. Only the two additional instructions mentioned above, and the use of the additional interrupt inputs, will be new.

11.8.3 ZILOG Z-80

This is a very sophisticated microprocessor that greatly extends the addressing capability of the 8080 instruction set. It is software compatible, in that it will execute programs written for the 8080. The undefined 8080 op-codes are used in the Z-80, in some cases for four byte instructions. Although there are a number of features that will be new to the 8080 programmer, use of the extended instruction set greatly simplifies programming of arithmetic functions. One of the most popular uses of the Z-80 is in personal computers with BASIC interpreters.

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX A

THE ICS MONITOR

A.1 ICS ADVANCED MICROCOMPUTER TRAINING SYSTEM DESCRIPTION

1) The ICS Microcomputer Training System uses the INTEL 8080A microprocessor, or an exact equivalent.

2) There are 1K bytes of ROM (addresses 0000 to 03FF), expandable on the board to 4K or 8K bytes using 1K by 8 or 2K by 8 chips, and 2K bytes of RAM (addresses 8000 to 87FF), expandable on the board to 4K bytes. With the optional S-100 bus, memory may be expanded to 64K bytes (any combination of RAM or ROM chips).

3) An 8255 Programmable Peripheral Interface chip is provided for Input/Output.

4) A Keyboard is provided with 25 keys. RESET gives a reset signal to the 8080A. Other switches provide input to the 8255.

5) A display is provided with eight digit positions. This is driven by DMA using the contents of addresses 83F8 through 83FF for digit positions 1 through 8.

6) Two LED's display the user's carry and zero flags.

7) The complete instruction set for the 8080A is given in the 8080 Microcomputer System User's Manual, together with detailed specifications of the machine's internal state during instruction execution and a description of all registers.

8) The MTS board layout is shown in Figure A-1. A block diagram is presented in Figure A-2.

INTEGRATED COMPUTER SYSTEMS, INC.

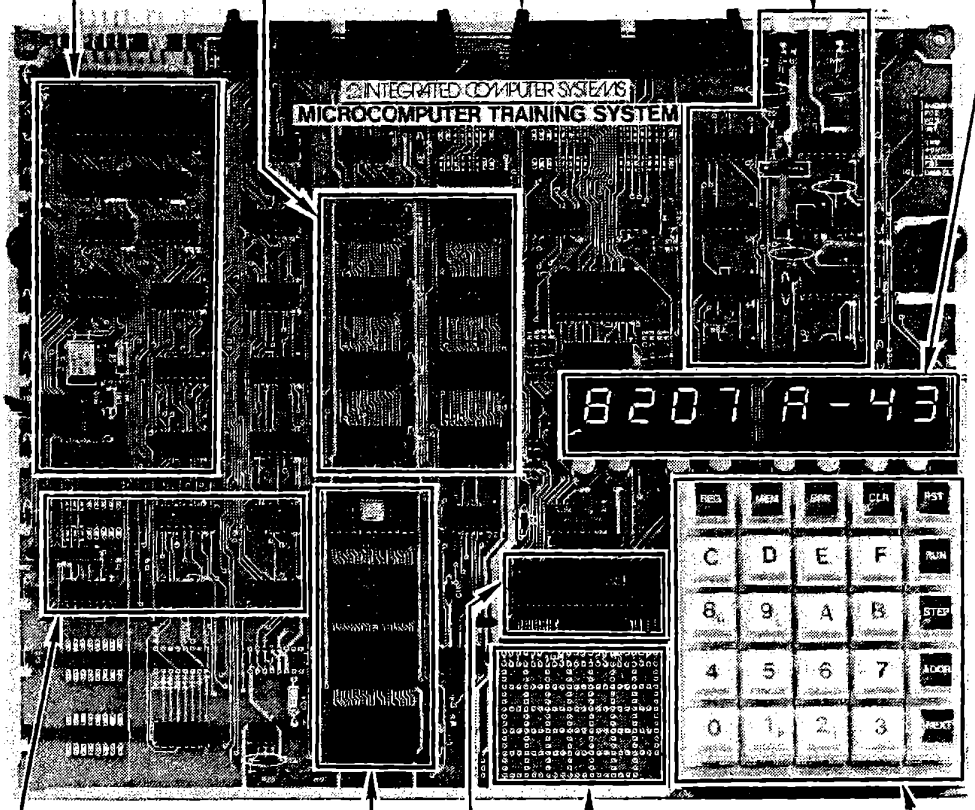
FULLY ASSEMBLED AND TESTED MICROCOMPUTER AND POWER SUPPLY

PROCESSOR HARDWARE
8080A Microprocessor and Control Logic

RAM MEMORY
2048 Bytes of RAM Memory for Programs and Data. Expandable On-Board to 4K Bytes.

AUDIO CASSETTE INTERFACE
Audio Cassette Interface and Associated Software for Easy Program Storage and Retrieval

DISPLAY
On-Board 8-Digit LED Display



DMA
Direct Memory Access (DMA) Channel

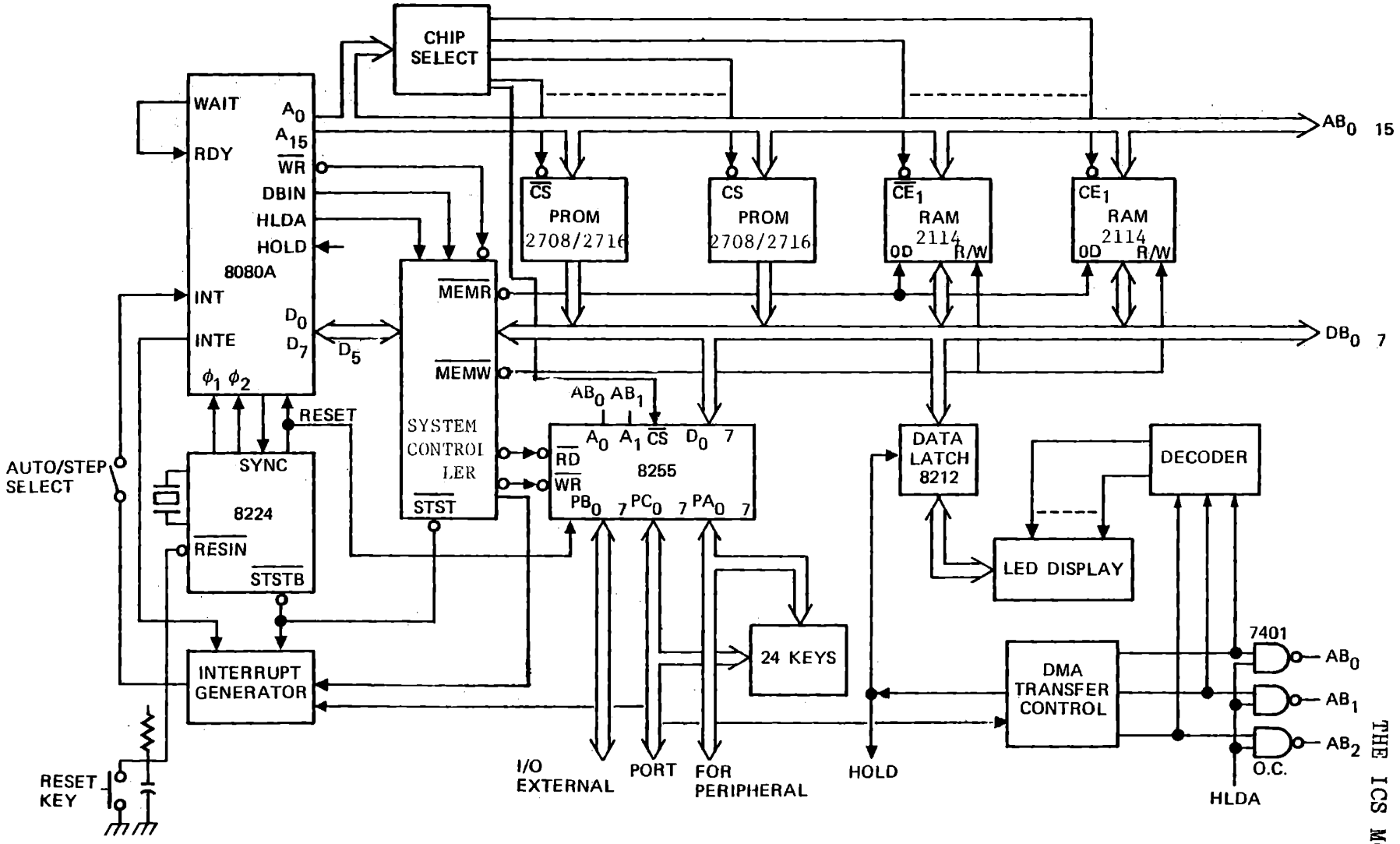
PROM MEMORY
Eraseable PROM Memory (containing the Educational Monitor Program) - 1024 Bytes. Expandable On-Board to 8K Bytes.

FREE AREA
Space for User's Hardware Additions

KEYBOARD
On-Board Keyboard with 25 Keys for Program and Data Entry.

PROGRAMMABLE PERIPHERAL INTERFACE
Programmable I/O Device Including Three 8-Bit Ports.

MTS Board Layout
Figure A-1



Microcomputer Training System Configuration
Figure A-2

A.2 GENERAL MONITOR FUNCTIONS

The monitor provides five general functions:

Load memory from keyboard

Store program on tape

Load program from tape

Operate program in debug mode

Run user program

A.2.1 Load Memory from keyboard

A.2.1.1 To select a memory address, press

ADDR n n n n

MEM

(where nnnn is the address: e.g. ADDR 8300 MEM)

The address will appear in the left four digits, and its present contents will appear in the right two digits preceded by a decimal point.

A.2.1.2 To enter data to memory after pressing MEM, key in one or two digits. They will replace the contents of the memory location whose address is displayed, and the new data will appear on the right.

A.2.1.3 If an error is made, it can be corrected by pressing additional digits. The last two digits keyed in will be stored and displayed. Pressing CLR will restore the original value that was stored, provided that no other command key has been pressed.

A.2.1.4 To proceed to the next higher memory location, press NEXT. It is not necessary to press MEM again.

A.2.1.5 Press MEM again to access the next lower memory location.

A.2.1.6 A decimal point to the left of the two right hand digits indicates that MEM has been pressed and data entry is enabled. If no symbol is displayed in this position data will not be accepted.

A.2.1.7 If data entry is attempted without being enabled, or if the memory location is in ROM or does not exist, the display will show Err. Press MEM to restore the address and enable entry.

A.2.2 Store program on tape.

The monitor program SEROT copies binary data from memory to a serial recording medium such as an audio tape cassette. The MTS circuit board includes an oscillator and modulator which are driven by this program. Data are recorded in serial synchronous format as described in Chapter 9.

A typical program will occupy 30 seconds to two minutes of tape. Several programs may be recorded on one tape; it is advisable to identify each program with a voice message preceding it. Then connect a cable from the MTS audio connector labeled with an outward arrow to the recorder Microphone or Auxiliary Input.

A.2.2.1 Operating Procedure

Press RESET. Now the modem will record a continuous tone. Let this continue for five to ten seconds while you do the following.

Set the STEP/AUTO toggle switch to AUTO.
Store the starting address by:

ADDR (starting address) MEM

If the starting address is 8200, this step is not necessary because RESET sets the memory address to 8200.

Then store the stopping address by:

ADDR (stopping address) BRK

(If this step is omitted, the serial output program will continue forever.) Note that the content of the stopping address is not recorded, so it must be the location of the next byte beyond the end of the program.

Start the program by:

ADDR 0371 RUN

The display will be blank. The Carry indicator will flicker, showing that data are being transmitted. This indicator is on during all data and stop bits; off during the start bit only. When the block of memory specified has all been transmitted, an error check character is recorded on the tape. Then the program reenters the monitor and the display will show: 0382 CD. Let the recorder run for another two or three seconds, then turn it off.

You can observe the error check character by pressing BRK. The stopping address and the error check character will be displayed. There is no need to observe this character, however.

A.2.2.2 Data Rate for Recording

The monitor program records data at 110 baud: That is, 110 bit intervals per second. (This rate is compatible with Teletype paper tape punches and readers.) Since the program records 12 bits per character, the resulting data rate is 9.17 characters per second. A 256 byte program will occupy 28 seconds of tape.

You can record at a higher data rate by entering the recording program with a delay count in register C. A value of 2D (hex) generates 300 baud. The procedure to use this feature is:

Turn recorder on
RESET
REG C (delay count)
ADDR (starting address) MEM
ADDR (stopping address) BRK
ADDR 0373 RUN

Note that the entry address is different, to avoid the monitor instruction that loads register C with 7D for 110 baud.

A.2.3 Load Program from Tape

The program SERIN loads binary data from a serial recording medium into memory. It is complementary to SEROT: it receives data in the format described above. A demodulator circuit is provided for reading from an audio tape cassette.

A.2.3.1 Operating Procedure

Before connecting the tape player to the MTS, listen to the tape and wait for the continuous tone. Now stop it promptly, so that when you turn it on again the continuous tone will appear again. Connect the recorder Earphone output to the MTS audio connector labeled with an inward arrow. Enter the starting address by:

ADDR (starting address) MEM

(This step may be replaced by RESET if the program is to start at 8200.)

Now load the program address:

ADDR 03AE

Turn the recorder on and press RUN. The display will go blank. Indicators labeled AUDIO and DATA will flicker while data is being received. If the continuous tone was not present the moment you pressed RUN, the display will show Err. Then you must repeat the procedure.

When the tape has been read successfully the program will reenter the monitor and display 03CF C5. Press MEM to display the stopping address. The error check character resorded on the tape is stored here. Now press REG A. The display will show 03CF A-00.

If the content of register A is not zero, there is a discrepancy between the error check character calculated and recorded by the output program and that calculated by the reading program. This means that an error exists somewhere in the data read back, but there is no indication of which byte is incorrect.

A.2.3.2 Alternate Data Rate

If you have recorded a tape at a different data rate than 110 baud, as described in A.2.2.2, you must again load register C for that data rate before entering the reading program.

It is also necessary to load register B with a constant. This determines the delay time before the reading program recognizes the end of the recording. A value of 20 H gives a 0.3 second delay.

The procedure is:

REG B 20
NEXT (data rate constant)
ADDR 03B1
turn tape player on
RUN

A.2.4 Operating in Debug Mode

The monitor provides for tracing the flow and results of a user's program. The STEP/AUTO toggle switch must be set to STEP; after each user instruction is executed a hardware interrupt is generated. This causes an entry to the monitor.

A.2.4.1 Step and Run

Operation of the user's program is initiated by the STEP command or the RUN command. A flag byte (SFLAG) is stored by the monitor when the STEP or the RUN key is operated. This flag determines the procedure to be followed at the next interrupt entry to the monitor. With the Mode toggle switch at STEP, either command will result in the user's program being interrupted at each instruction, but in RUN the return to the user program is automatic unless a Breakpoint is encountered. With the Mode toggle switch at STEP, the STEP key results in the monitor activating the keyboard and display after each user instruction is executed. With the Mode toggle switch at AUTO, the user's program runs without interruption, with either command.

A.2.4.2 Breakpoint Operation

If the initiating command was RUN, the monitor tests any breakpoints entered by the user. Two tests are made:

a) Is the user's program counter equal to any breakpoint entered?
b) Has the data changed at any memory location entered as a breakpoint?

If neither condition is true, the monitor returns control to the user's program. If either test is true, the monitor tests a counter associated with that breakpoint. If non-zero, it decrements the counter and returns to the user's program, but if the counter is zero, the keyboard and display are activated.

Note that a breakpoint stops the user's program before executing an instruction whose address is entered as a breakpoint, but after data has been changed at a breakpoint. If any breakpoint is encountered, no other breakpoint will be tested until the next instruction in the user's program has been executed.

The breakpoint system can also be used to stop execution after a specified number of instructions have been executed, rather than at a specific instruction. The process of entering and removing breakpoints is covered in Section A.3.3.8 of this appendix.

A.2.4.3 Monitor Display

When the display is active under monitor control, it shows an address in display positions 1-4 (the left four digits) and a data byte in positions 7 and 8 (the right two digits). At entry to the monitor the address displayed is the program counter, and the data are either the next instruction or the contents of a register. The latter is identified in digits 5 and 6.

The user may request many other displays, such as another register, another address in memory, a register pair and the contents of the addressed location, the stack pointer, or the user's stack top.

A.3 MONITOR COMMANDS

The major sections of the monitor operate as an interrupt service routine entered by a hardware interrupt automatically generated as each user instruction is executed, provided that the AUTO/STEP switch is in the STEP position.

The user may program entry to the monitor by including the RST4 instruction (E7) in his program. He may alter addresses and flags used by the monitor through his own program, thereby affecting monitor functions. Various monitor subroutines are accessible to the user by normal subroutine calls.

A.3.1 Monitor Entry

When the monitor is entered by interrupt (RST7) or by programmed call (RST4) the user's registers, program counter, and stack pointer are saved in memory and may be accessed by monitor commands.

The RESET key causes a hardware reset to the 8080. In general, the user's register contents and stack are lost. The user's memory address and program counter are set to 8200, and the user's stack pointer is set to 83E0. All interrupt entry addresses are initialized and all breakpoints are cleared. Some data, including the user's program counter, may be recovered as described in Section A.3.4.

A.3.2 Monitor Data Storage

At entry to the monitor, the user's program counter is popped from the stack and stored at PCADR. The registers are pushed onto the stack. If the STEP Key was used or a breakpoint is encountered, the Carry and Zero flags are shown in two LED's. Neither these nor the hexadecimal display are changed unless STEP was used or a breakpoint is encountered.

The monitor stores the following data in fixed memory locations:

BKLOW (83E2) is a one byte pointer to the oldest breakpoint in the breakpoint table. If no breakpoints have been entered, it contains E2, pointing to itself.

BKPOS (83E3) is a one byte pointer to the breakpoint most recently encountered.

MADDR (83E4, E5) The address of the last memory location accessed via the MEM or NEXT command.

PCADR (83E6, E7) The user's program counter.

SFLAG (83F6) indicates whether the user's program was last initiated by RUN or STEP. If it contains zero, breakpoints are tested; otherwise, the monitor keyboard and display functions are enabled at entry to the monitor.

RGNAM (83F7) The name of the register displayed by REG command, or zero if MEM command has been used since the last REG command. If this value is non-zero when the monitor keyboard and display functions are activated, the named register is displayed.

When the monitor is awaiting a command or data, register pair H,L generally contains a display address, which points to either the memory address, the user's program counter, a breakpoint, or an address just keyed in by the user following the ADDR command.

Operation of the monitor commands can be described in large part by reference to these addresses:

PCADR (the user's program counter)

MADDR (the memory location most recently addressed with MEM or NEXT)

and the display address in (HL).

A.3.3 Monitor Commands

Monitor commands are issued by pressing one of the eight command keys: ADDR, MEM, NEXT, CLR, REG, STEP, RUN, BRK. These are discussed below in the order listed.

A.3.3.1 ADDR

Recalls the user's program counter and makes it the display address. The PC is displayed in the left four digits, and the content of memory at that address in the right two digits.

If ADDR is followed by hexadecimal keys, the display address is cleared and the hex characters are entered as the display address. In general four characters must be entered, but this depends on the command which follows ADDR. A count of the number of keys is complemented and stored in register D for use by the monitor in executing the next command.

Contents of D:

00	ADDR not used
FF	ADDR used, no hex keys
FE	one hex key
FD	two hex keys
FC	three hex keys
FB	four hex keys

When the first hex key is pressed, the left display digit shows that key with three leading zeros, and the right hand display is blanked. Additional keys are shifted into the left hand display. When four keys have been entered, they represent an address, and the memory content of that address is displayed at the right.

When another command key is pressed, the displayed address is passed to the appropriate command processing module. In some cases fewer than four hex keys will be accepted as an address or converted to an address. See the sections describing MEM, BRK, STEP and RUN for details.

A.3.3.2 MEM

Calls for display of a memory address and its contents. If the preceding command was not ADDR, the previously stored memory address is used. If ADDR was used, the address in H,L becomes the memory address. This may be the user's program counter or a newly keyed address. If exactly one hex key followed ADDR, that is taken as the name of a register pair, the stack pointer, or the stack top, and the two bytes referred to thereby become the memory address.

Key	Register Pair
1/SP	Stack pointer
2/ST	Stack top
8/H	H,L
B	B,C
D	D,E

Other single key entries are errors.

With a memory address determined, it is displayed in the four left digits and the contents of that location are displayed in the right two digits. If the address was derived from a register pair, a label identifying that pair is displayed.

After the MEM command has been issued, the contents of the displayed location can be altered by keying in one or two (or more) hex digits. A decimal point in digit 6 indicates that data can be altered.

The NEXT command increments the memory address and displays the new address and contents. Again, the contents can be altered.

Note that ADDR causes display of a memory address, but the contents cannot be altered until the MEM command has been given. Assume that the present memory address is 8300:

MEM			8300	.AF

Recalls and displays previous memory address and contents with a decimal point. Contents can be altered by hex keys.

	4	4	8300	.44
ADDR			8200	01

Recalls and displays user's PC and instruction. Contents cannot be altered. No decimal point is shown.

```
ADDR    MEM                                          8200      .01
```

Now contents can be altered. 8200 is now the stored memory address.

```
NEXT                                                 8201      .80
```

Displays the next byte in memory. 8201 is the stored memory address. Contents can be altered.

```
ADDR    8       3       0       0                    8300      44
```

Displays 8300 again and its contents again, but contents are protected.

```
MEM                                                  8300      .44
```

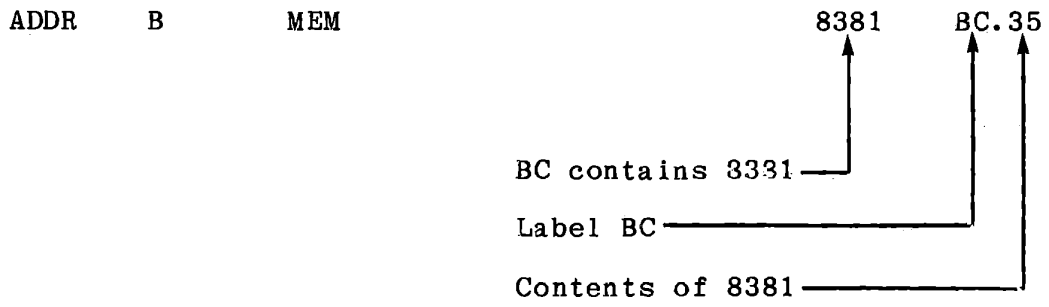
Now 8300 is the stored memory address and its contents can be altered, as indicated by the decimal point:

```
            3       2                                8300      .32

MEM                                                  82FF      .00
```

Repeated use of MEM with no other command intervening displays the address and content of the next lower memory location. By using NEXT or MEM, the memory content can be reviewed (and altered if desired) in either ascending or descending order.

Register pair display



When a register pair is displayed, its contents are stored as the memory address, and the data addressed can be altered. The address can be incremented or decremented by NEXT or MEM, but this changes only the memory address, not the content of the register pair. The register pair symbol is removed when the address is changed.

A.3.3.3 NEXT

This increments the memory address if a memory location is being displayed. If MEM has not previously been pressed, NEXT increments the display address and stores it as the current memory address, but does not enable data entry.

If a register pair is displayed, the content of that pair is the memory address. NEXT will display the next higher address, and remove the register pair symbol.

When a register is displayed NEXT selects the next register in sequence: A, B, C, D, E, F, H, L, A - - - (See A.3.3.5)

When a breakpoint is displayed NEXT calls for display of the next breakpoint in the list. If there is only one breakpoint in the table, NEXT has no effect. (See A.3.3.8)

A.3.3.4 CLR

If CLR follows entry of data (to a register or a memory location) the previous data are restored and displayed.

If a memory location was being displayed and MEM had been pressed, the address, pair label (if any) and data are still displayed. Data entry is still enabled.

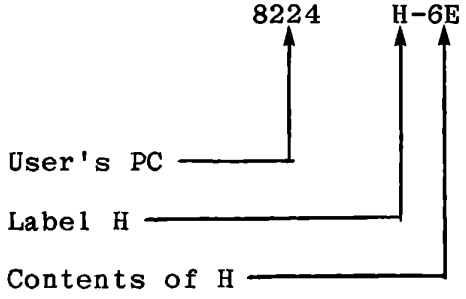
If a memory location was being displayed without MEM having been pressed, CLR restores the display of the program counter. The memory location can be recovered by MEM.

If an address was being entered, CLR restores the display of the program counter and permits an address to be entered. (ADDR need not be pressed again.) MEM will recover the previous memory address. CLR is also used to remove a breakpoint. See A.3.3.8.

A.3.3.5 REG

REG is followed by a hex key naming the register desired. REG n displays the current contents of the user's program counter and the contents of register n, with a label.

```
REG     8/H                                    8224    H-6E
```



Legal register names are A, B, C, D, E, F, H (Key 8) and L (Key 9).

If followed by any hexadecimal key or keys the contents of the displayed register are altered:

```
REG     8/H     3       2                      8224    H-32
```

If followed by NEXT, the next register (alphabetically) is displayed:

```
NEXT                                           8224    L-13
```

The name of the register selected for display is retained, and at subsequent entry to the monitor the selected register will be displayed. When the MEM key is used, the register name is cleared; then further entries to the monitor will display the contents of the current address. A register name is stored (as one byte at RGNAM) when a register is selected by REG n or by NEXT while a register is being displayed.

If REG follows an ADDR command the effect of the ADDR command is lost. REG always shows the program counter in the left hand four digits.

A.3.3.6 STEP

STEP sets (SFLAG) = 1 to indicate that the monitor keyboard and display functions are to be activated at the next entry to the monitor. All user registers are restored, the interrupt system is enabled, and control is returned to the user's program at the location stored in PCADR. The user's program is interrupted upon execution of the next instruction and the monitor is reactivated.

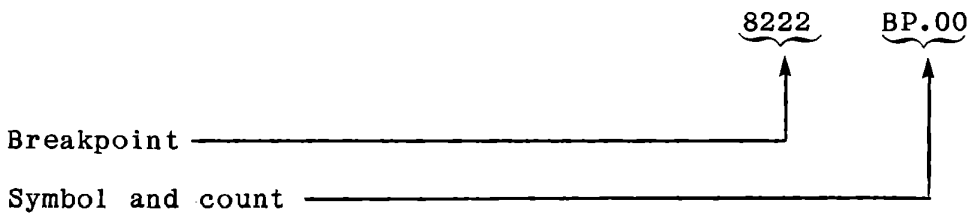
If the STEP (or RUN) command immediately follows an ADDR command with four (or more) hexadecimal keys, then the address entered becomes the user's program counter, and control is passed to that location.

A.3.3.7 RUN

RUN sets (SFLAG) = 0 to indicate that the RUN command was issued and then returns to the user's program exactly as in STEP. The user's program is interrupted at each instruction to test for breakpoints, but the keyboard and display are not activated unless a breakpoint is encountered and its count reaches zero. When this occurs the monitor behaves as though STEP had been used.

A.3.3.8 Breakpoints

BRK displays the address of the current breakpoint. If a breakpoint has been encountered during execution of the user's program, it will be displayed in response to BRK.



A breakpoint is "encountered" either when the user's program counter is equal to the breakpoint, or when the user's program changes the content of a memory location whose address has been set as a breakpoint.

If no breakpoints have been entered, the BP. symbol will be displayed with all other digits blank.

A breakpoint is entered by:

```
ADDR    8       2       1       0               8210        3C

BRK                                             8210    BP.00

        4       (optional count)                8210    BP.04
```

When RUN is pressed, this address will be encountered and executed four times, stopping on the fifth. Then the display shows the program counter and instruction:

```
                                                8210        3C

BRK                                             8210    BP.00
```

BRK shows the breakpoint, now counted down to zero. It may be left with a zero count, or a new count may be entered:

```
        2       4                               8210    BP.24
```

Certain addresses can be entered as breakpoints by abbreviated entries, as listed below.

User's program counter	ADDR	BRK	
Current memory address	ADDR	0	BRK
User's stack pointer	ADDR	1/P	BRK
User's stack top	ADDR	2/T	BRK
Content of BC	ADDR	B	BRK
Content of DE	ADDR	D	BRK
Content of HL	ADDR	8/H	BRK

When a breakpoint is displayed it can be cleared.

```
                                                    8210     BP.24

        CLR                                                  BP.
```

The blank display shows that no breakpoints exist. If other breakpoints are still stored, the next one in the list would now be displayed. If more than one breakpoint is stored NEXT will display each in turn. Whenever a breakpoint is displayed it may have a new count entered or it may be cleared. RST clears all breakpoints.

The breakpoint system can also be used to stop execution after a specified number of instructions have been executed, rather than at a specific instruction:

```
ADDR    83E6    BRK     (count)
```

This sets a breakpoint at the location where the monitor stores the user's program counter. Since the content of this location is changed at each monitor entry, the breakpoint will be encountered after every instruction. When the count is decremented to zero, program execution will stop.

Each breakpoint entered occupies four bytes of memory. These data are entered above the user's stack, which is pushed down in memory. The data are:

Breakpoint address (two bytes)
Breakpoint data
Breakpoint count

The oldest breakpoint is stored immediately above the user's stack area, and BKLOW points to the low byte of its address. The breakpoint data is a copy of the data stored in the memory location addressed by the breakpoint; it is updated each time the breakpoint is tested. The count is zero unless another value is entered when the breakpoint is displayed.

The number of breakpoints that can be entered is limited only by the space available in page 8300 of memory. However, it is seldom useful to have more than four breakpoints in operation at one time.

Note that when a breakpoint is entered or removed, the entire stack from 83E1 down is moved. If the user's program reinitializes the stack pointer, no breakpoints should be added or removed subsequently.

Breakpoints can be entered or removed by the user's program. See Section 9.6.2 in the text for a description of this process.

A.3.3.9 Error Display

Err may be displayed for any of the reasons listed below. CLR or ADDR will restore the display of program counter and instruction. MEM will recover and display the memory address and its content. Error conditions are:

a) Attempt to write to a non-existent or ROM memory location; or to enter data when data entry is not enabled.

b) REG x where x is not a register name (A, B, C, D, E, F, H, L).

c) ADDR x MEM where x is not a register pair name (B, D, H, P, T).

d) ADDR x BRK where x is not a register pair name nor 0.

e) Attempt to enter a breakpoint count when no breakpoint exists.

f) Numeric key pressed without a preceding legitimate command key (ADDR, MEM, REG, BRK).

g) Attempt to STEP or RUN after entering a one, two or three byte address.

The error display can be generated by the user's program using a call to monitor subroutine ERRDS. See A.5.4.6.

A.3.4 Recovering Data after RESET

Occasionally a program which has not been fully debugged will enter an endless loop, with no input or output accomplished. Control can be restored to the monitor by pressing RESET.

The fixed data locations 83E8 through 83F7 are initialized with the interrupt dispatch addresses. MADDR and PCADR are loaded with 8200. RGNAM and SFLAG are cleared, and all breakpoints are cleared. Provided that the program was running in breakpoint mode (toggle switch at STEP), some useful data can be recovered.

Before PCADR is initialized, its content is copied to 83E0, E1. During RESET initialization, this is treated as a fixed address, but it is subsequently treated as part of the stack. It can be viewed by pressing ADDR 2/T MEM after a reset.

Eight bytes of the stack (83D8 - 83DF) are also preserved. If no breakpoint had been entered and the user's stack was empty, these contain the user's registers as last stored while the user's program was running, and can be observed as register contents by REG A, NEXT, etc. If the user's program was executing a subroutine when RESET was pressed, its return address will be found at 83DE, DF. If a breakpoint had been entered, then four bytes of the preserved stack area will be occupied by useless data, but four bytes of the user's stack or registers will be preserved. If two breakpoints had been entered, no useful data except the program counter can be recovered. The following table shows the content of this area under several conditions.

	NO BREAKPOINTS		ONE BREAKPOINT	
	User Stack Empty	User Subroutine	User Stack Empty	User Subroutine
83D8	F	E	C	L
D9	A	D	B	H
DA	E	C	L	Return
DB	D	B	H	Address
DC	C	L	Previous last	
DD	B	H	program counter	
DE	L	Return	Breakpoint Address	
DF	H	Address		
83EO	Last observed user program counter			
E1				

A.4 PROGRAM CONTROL OF MONITOR FUNCTIONS

A.4.1 Output Port 0C

The keyboard, display, modem output, and the monitor interrupts are controlled by outputs at Port C of the 8255. (This is referred to as PORT0C to distinguish it from ports of the ICS Interface Training System.) The bit assignments and effects are listed below.

PORT0C	Zero	One
Bit 0	Modem frequency low	High
Bit 1	Monitor disabled	Enabled
Bit 2	Carry LED off	On
Bit 3	Zero LED off	On
Bit 4	Keys 0-7 enabled	Disabled
Bit 5	Keys 8-F enabled	Disabled
Bit 6	Command keys enabled	Disabled
Bit 7	Display disabled	Enabled

These can be controlled by the user's program, either by writing to PORT0C:

```
3Exx        MVI   A,xx
D302        OUT   PORT0C
```

or by setting or resetting a single bit:

```
3Eyy        MVI   A,yy
D303        OUT   CNT0
```

The table below gives values to be loaded to (A) for the bit set/reset function (using MVI A, xx; OUT CNTO).

00	Set modem frequency low
01	Set modem frequency high
02	Disable monitor
03	Enable monitor
04	Carry LED off
05	Carry LED on
06	Zero LED off
07	Zero LED on
08	Enable keys 0-7
09	Disable keys 0-7
0A	Enable keys 8-F
0B	Disable keys 8-F
0C	Enable command keys
0D	Disable command keys
0E	Disable display
0F	Enable display

All of these bits are controlled by the monitor when its keyboard and display functions are enabled. They are also affected by certain monitor subroutines which may be called by the user.

The monitor sets the modem frequency high, enables monitor interrupts, shows the user's carry and zero flags in the LED's, enables all rows of the keyboard, and enables the display.

Monitor subroutines DBYTE, DWORD, ENTBY, ENTWD, ENMEM, KEYS and GETKY enable the monitor, the display and all rows of the keyboard. The modem and LED's are not affected. During execution of all keyboard input subroutines, the monitor interrupts are disabled to avoid slowing the debounce delay by repeated interrupts.

Monitor subroutine SCAN reads the keyboard once. If any key is pressed, it returns with the row containing that key enabled. If no key is pressed, all keys are disabled. Monitor interrupts are disabled.

A.4.2 Interrupt Entry Points

The MTS provides for external interrupts as well as the monitor interrupt. All of the 8080 RST instructions except RST0 (which is also RESET) are available, although RST7 is required for monitor debug operation.

RST1 through RST7 are treated identically when they are detected. Each loads a dispatch address from a different location in memory and jumps to that address with all registers and the stack intact. The stack top contains the address of the interrupted instruction. The instruction sequence is:

```
RSTX:   PUSH H
        LHLD ENTX
        XTHL
        RET
```

Except for timing, this is equivalent to JMP xxxx. RST4 precedes the above sequence with DI, because it is used for programmed calls to the monitor which must not be interrupted. The other RST entry points do not disable interrupts, but if invoked by an external interrupt, that action will have disabled interrupts.

The dispatch address locations and initial values set by the monitor at RESET are listed below.

Interrupt Vector	Dispatch Location	Initial Value
RST1	83F4, F5	0000
RST2	83F2, F3	0000
RST3	83F0, F1	0000
RST4	83EE, EF	0063
RST5	83EC, ED	8228
RST6	83EA, EB	8230
RST7	83E8, E9	006A

The user may change these values either by monitor commands or by program instructions. The latter is recommended since RESET will restore the values listed above. If it is possible for an interrupt to occur before initialization, the user's program should disable interrupts as its first instruction.

Note that changing the dispatch address for RST7 allows the user to write his own monitor functions (See Section 8.6.2 in the text).

A.5 INPUT/OUTPUT AND MONITOR SUBROUTINES

The hexadecimal and command keys of the MTS and the eight digit display are accessible to the user either by direct input and output or memory operations or through monitor subroutines. Refer to Chapter 8 for a discussion of input and output techniques and exercises.

A.5.1 Keyboard Input

The keyboard is connected as a 3 x 8 matrix. A row of eight keys is enabled by setting one bit at output port 0C low.

Port 0C4 Low enables keys 0-7
Port 0C5 Low enables keys 8-F
Port 0C6 Low enables command keys.

If one row is enabled and the other two rows are disabled, the eight keys in that row can be sensed at input port 0A.

Input Bit	Key		
0A0	0	8	MEM
0A1	1	9	REG
0A2	2	A	ADDR
0A3	3	B	STEP
0A4	4	C	RUN
0A5	5	D	NEXT
0A6	6	E	BRK
0A7	7	F	CLR

If a key is pressed and its row is enabled, the corresponding input bit will be low. Otherwise, the input bit will be high.

When the monitor returns control to the user's program after STEP or RUN, all three rows of the keyboard are enabled. Several of the monitor I/O subroutines also enable all three rows. In this state no single key can be recognized, but it is easy to test whether the keyboard is idle or in use.

```
DB00        IN PORT0A
3C          INR A
```

sets the zero flag if no key is pressed. This function can be followed by a conditional jump or a conditional call to an input subroutine.

A.5.2 Monitor Input Subroutines

Six monitor subroutines are provided for keyboard input. These are described below. Any of them can be called by user programs. Each is identified by a name and by its address. The descriptions specify the data returned by the subroutine, and entry data where required.

A.5.2.1 SCAN (0257)

Disables monitor interrupts. Tests each row of the keyboard in turn, until all have been tested or until an active key is found. If no key is pressed, all keyboard rows are disabled. If a key is pressed, its row is left enabled, and the others are disabled. It tests command keys first, then 8-F, and finally 0-7. If two keys in different rows are pressed, SCAN detects the key from the higher valued row. If two keys in the same row are pressed, SCAN returns the lower valued key. If more than two keys are pressed, an erroneous value may be returned, but it will still lie in the range of allowed values, 00-17.

If no key is pressed, SCAN returns (A) = 00, Zero Set, No Carry.

If a key is pressed, SCAN returns (A) = key value (00-17), Carry Set. Zero is set if the key is 0.

All registers except A are preserved.

No entry data are required.

The stack is used for one level in addition to the return address.

Timing depends on the input detected:

No keys	457 clocks
Maximum (Key 7)	553 clocks
Minimum (MEM key)	200 clocks

Add 5432 clocks if the monitor was enabled at entry.

A.5.2.2 GETKY (023D)

GETKY waits indefinitely for a key to be pressed, and then waits until all keys are released for a time of 26 milliseconds to protect against contact bounces.

GETKY calls SCAN to read the keyboard. Comments under SCAN regarding multiple key depressions apply to GETKY. The first key detected is returned by GETKY.

Returns (A) = (C) = Key
(B) = 00
All other registers preserved

Carry is cleared for command keys, set for hexadecimal keys.

Zero is set for MEM, cleared for all others.

At exit monitor interrupts, display, and all rows of the keyboard are enabled.

No entry data are required.

A.5.2.3 KEYS (0365)

Subroutine KEYS accepts successive hexadecimal keys by calling GETKY and shifting the hexadecimal values into register pair HL, to return the last four hexadecimal digits packed into two bytes. It increments register D and returns to the calling program after each key. KEYS does not demand any entry data, but is usually called with (HL) containing previous data and (D) containing the key count. Usually (HL) is cleared and (D) = FF at the first call. The following assumes that initialization:

Return (A) = (C) = last key pressed

(B) = 00

(D) = count of hexadecimal keys, provided (D) was initialized to FF before the first call

(E) preserved

(HL) last four hexadecimal keys pressed, with leading zeros if cleared before the first call.

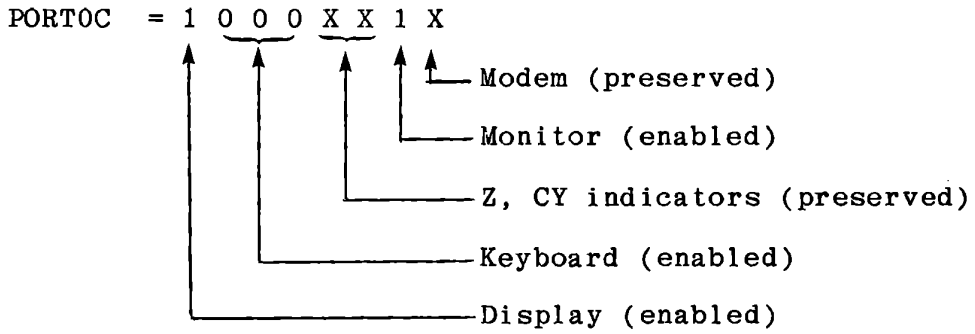
Carry set if the last key is hexadecimal, clear if command.

Zero is cleared, except zero is set after the first key if D was initialized to FF.

A.5.2.4 MENAB (0222)

Enables the monitor, display, and all rows of the keyboard, by writing to PORT0C. Zero indicator, carry indicator, and modem control bits are preserved. At return:

All registers and flags are preserved.



The following monitor subroutines exit via MENAB:

All display subroutines except DISPR

GETKY, which is called by all input subroutines except SCAN

A.5.2.5 ENTBY (0336)

ENTBY is intended for entry of one byte. It initializes (D) = FF and (HL) = 0000. It then calls KEYS repeatedly until Not Carry is returned indicating that a command key has been entered. After each hexadecimal key it calls DBYTE to display the last two keys.

At return:

(A) = (C) = command
(B) = 00
(D) = count of hexadecimal keys
(E) preserved
(HL) = last four hex keys
Carry clear
Zero clear, except if no hex keys entered then Zero is set.

A.5.2.6 ENTWD (0346)

ENTW2 (0349)

ENTWD is intended for entry of two bytes of data as a memory address. AT ENTWD, (HL) is cleared. Alternate entry ENTW2 preserves (HL) if no hex keys are entered. Register D is initialized to FF.

ENTWD calls KEYS to accept keyboard entry, and returns when a command is entered.

When the first hex key is entered, it is placed in (HL) as the low digit, with higher digits cleared. The right hand display digits are cleared.

Subsequent hex keys are shifted into (HL) by KEYS. ENTWD calls DWORD to display (HL) at the left after each hex key. After four or more hex keys have been entered, representing a memory address, ENTWD calls DMEM to display the content of the memory location.

At return:

(A) = (C) = command

(B) = 00

(D) = count of hex keys

(E) preserved

(HL) = keyed data. If no hex keys entered, ENTWD entry clears (HL), but ENTW2 preserves (HL). If one, two three or four hex keys entered, (HL) contains leading zeros.

Carry and Zero clear, except Zero set if no hex keys.

Return from ENTW2 if no hex keys entered:

(A) = (C) = command

(B) = 00

(D) = 00

(E) = preserved

(HL) preserved

Carry clear, Zero set

A.5.2.7 ENMEM (01FF)

ENME2 (0200)

ENMEM accepts keyboard entry and stores the last two hexadecimal keys entered into the memory location addressed by (HL). After each hexadecimal key is entered it calls DMEM to display the data.

If entry is at ENMEM (01FF) the existing data at (HL) are displayed before any key is entered. If entry is at ENME2 (0200) and the carry is set, this initial display is omitted.

ENMEM returns when a command is entered. If no hexadecimal keys precede the command, the data byte at (HL) is preserved. Also, if RESET is pressed with no hex keys having been entered, the data have not been changed.

When the first hex key is pressed, it is stored and displayed as the low digit of the data byte with the high digit set to zero. Successive keys are shifted into the data byte. The data byte is stored in memory as each hex key is entered, so RESET following a hex key leaves the data entered in the designated memory location.

When a command key is pressed, it is tested for CLR. If the command is CLR, the original data found at (HL) is restored before return.

Entry Data:

(HL) = memory address

For entry at ENME2, carry set to suppress initial display.

Return Data, after command key:

(A) = (C) = command

(B) = 00

(D) = new data entered

(E) = original data from ((HL))

(HL) preserved

If command key is not CLR:

((HL)) = new data if entered, otherwise ((HL)) preserved

Not Zero. Carry set, except Not Carry if command is BRK.

If command key is CLR:

((HL)) = original data

Zero Set, Not Carry

Error Detection: See next page.

If data entry is attempted to a ROM or non-existent memory location, returns via ERRDS to display Err, with:

(A) = (B) = 00

(C) = (D) = hex key

(E) = F7

(HL) preserved

Zero Set, Not Carry

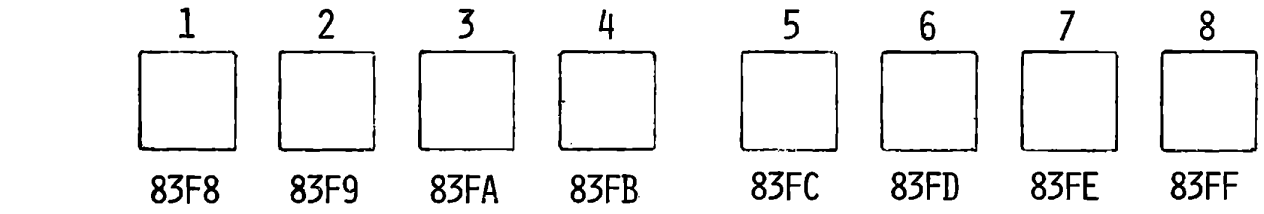
Note that the flags do not distinguish the error return from the CLEAR return. The content of A can be tested for the error condition: (A) = 00 only if an error return has been made. The memory address which could not be written to is preserved in (HL).

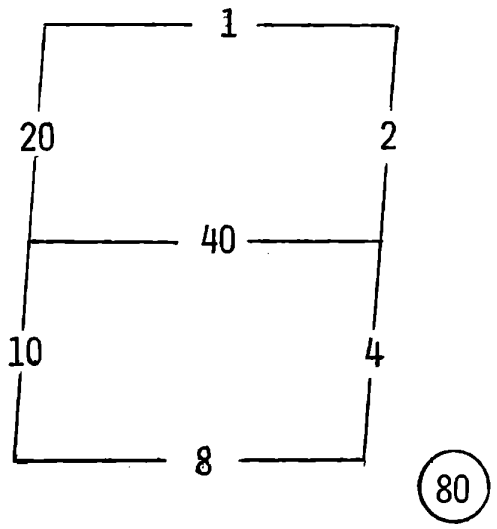
A.5.3 Display

Data stored in memory locations 83F8 - 83FF are displayed by the DMA channel, provided the display is enabled by a high output at Port 0C7. The content of 83F8 controls the left digit, 83FF controls the right digit. The data must be stored in seven segment code, with each bit controlling one segment. Bit assignments, as shown in Figure A-3, are:

Bit 0	Top horizontal
Bit 1	Upper right
Bit 2	Lower right
Bit 3	Bottom horizontal
Bit 4	Lower left
Bit 5	Upper left
Bit 6	Middle horizontal
Bit 7	Decimal point

Monitor subroutines are available for displaying hexadecimal data. A table of hexadecimal symbols is in ROM at 02B3.





Hexadecimal Codes for LED Segments

Figure A-3

A.5.4 Monitor Display Subroutines

Four display subroutines are described:

DISPR	displays one digit
DBYTE	displays one byte
DWORD	displays two bytes
DMWD	displays a memory address and its content.

Several alternate entries are also identified. In addition, a subroutine is provided to right justify the high digit of a byte to make it ready for display by DISPR. Another subroutine clears all or part of the display.

A.5.4.1 DISPR (02A6)

Displays one digit at a specified location. It masks register A to obtain the low digit, adds this to a table address to find a corresponding seven segment code, and stores the result at the memory location specified by (DE). It then decrements (DE) to address the next leftward display position.

Entry (A) = digit to be displayed, right justified
(High bits need not be clear)
(DE) = display digit address

Return (A) = (C) = byte entered
(B) preserved
(DE) decremented
(HL) = table address for seven segment code

A.5.4.1.1 DIGHI (02C3)

DIGSW (02C5)

DIGHI clears the low digit of the byte entered in (A) and right justifies the high digit. DIGSW switches the two digits of the byte entered in (A).

All other registers are preserved.

A.5.4.2 DBYTE (0295)

DMEM (0294)

DBY2 (0298)

DBYTE displays the content of register A, after loading (DE) with the address of the right display digit, so that the data are displayed in the two right positions.

DMEM loads the data stored at the memory location addressed by (HL) to be displayed.

DBY2 displays the content of register A in the digit addressed by (DE) and the next digit to the left. The translation and memory write are done by:

```
CALL DISPR          Display low digit
CALL DIGHI          Get high digit
CALL DISP2          Display high digit
```

After the second digit is displayed, the subroutine exits via MENAB to enable the monitor, display, and all rows of the keyboard.

Entry data

DMEM	(HL)	addressing memory location
DBYTE	(A)	contains byte to be displayed
DBY2	(A)	contains byte to be displayed
	(DE)	addresses display digit for low digit

Return (via MENAB)

(A) = (C) = digit displayed

(B) preserved

(DE) addressing next digit to left of high digit displayed; for DMEM and DBYTE (DE) = 83FD.

(HL) preserved

Not zero, Not carry

Display, monitor and all rows of the keyboard are enabled.

A.5.4.3 DWORD (02D1)

DWD2 (02D4)

DWORD displays the content of register pair HL as four digits at the left, by loading pair DE with 83FB, the address of the fourth display location. DWD2 displays the content of register pair HL as four digits, placing the low digit at the location addressed by (DE).

The procedure is:

```
DWORD:    LXI D, 83FB
DWD2:     MOV A, L
          CALL DBY2
          MOV A, H
          JMP DBY2
```

Note that the exit is from DBY2 via MENAB. See the summary of entry and return data in Section A.5.4.4.1.

A.5.4.4 DMWD (02CE)

DMWD displays a memory address and its content. It comprises a single instruction, CALL DMEM, immediately preceding DWORD. The address contained in (HL) is displayed at the left and the memory content, ((HL)) is displayed at the right. See A.5.4.4.1 for entry and return data.

A.5.4.4.1 DYPC (02CB)

DYPC loads (HL) with the data stored at 83E6, E7 and enters DMWD. These data represent the last recorded user program counter when operating in breakpoint mode with interrupts enabled, so DYPC would always display its own address and instruction. It is useful as a debugging and analysis tool if called by an interrupt service routine, because it will then display the interrupted instruction and its address.

The table below summarizes entry and return data for DYPC, DMWD, DWORD and DWD2.

Entry Data

	DYPC	DMWD	DWORD	DWD2
(DE)	--	--	--	Digit Address
(HL)	--	——— Memory Address ———		

Return Data

(A)	=(H)	=(H)	=(H)	=(H)
(B)	——— Preserved ———			
(C)	=(H)	=(H)	=(H)	=(H)
(DE)	83F7	83F7	83F7	(DE)-2
(HL)	(PCADR)	——— Preserved ———		

A.5.4.5 Clear Display and Memory

CLRGT (0282)

CLEAR (0287)

CLRHI (0289)

CLRLP (028C)

CLRGT clears the right hand four digits of the display; CLEAR clears the entire display. Neither of these needs any entry data.

CLRHI starts at 83FF, the right hand display digit, and clears the number of bytes specified by the content of register B, working downward.

CLRLP starts at the memory location addressed by (HL) and clears the number of bytes specified by (B), working downward. CLRLP is not restricted to the display, but can be used to clear any desired part of memory, up to 256 bytes.

All return with (B) = 00 and (HL) addressing the byte below the lowest byte cleared. For CLEAR this addresses RGNAM.

Zero is set. Carry is preserved.

Registers A, C, D and E are preserved.

A.5.4.6 ERRDS (00BC)

Displays Err in three left hand digits of the displays, the clears the remaining digits. No entry data required.

Returns

Zero, Not Carry

(A) = (B) = 00

(E) = F7

C, D, H, L Preserved

A.5.5 Time Delay Subroutines

A.5.5.1 DELAY (0236)

DELYA (0238)

DELYA counts down in register A and returns at Zero. DELAY loads register A with 83H to delay slightly less than one millisecond. Return (A) = 00, Zero set. Carry and all other registers preserved.

Each count in (A) represents 15 clocks. Including CALL and RET, the delay generated is given by:

$$t = (15\ (A) + 27) \quad \text{system clocks}$$

Some sample values calculated from the system clock frequency of 2048000 clocks per second:

A	Time (milliseconds)
00	1.888
87H	1.002
83H	0.973
42H	0.497
20H	0.248
0CH	0.101

A.5.5.2 DELYT (03EB)

DELYC (03EE)

Subroutine DELYC delays for a time period set by the content of (C) and returns with the input from PORT0B0 in the carry.

DELYT calls DELYC and then exits into DELYC, thereby doubling the delay period. These subroutines are used in serial data transmission and receiving, where DELYC gives a half bit time delay and DELYT a full bit time. They are also useful for giving longer delay times than are available from DELYA. Some delay times are tabulated below.

Enter with (C) = delay count

Return Carry = PORT0B0

Zero set

All registers preserved

The delay time is given by:

DELYC 48 + 74 X (C) system clocks

DELYT 106 + 148 X (C) system clocks

These counts include 17 clocks for the user's call to the subroutine.

Some typical values are:

	DELYC		DELYT	
(C)	Clocks	Time (msec)	Clocks	Time (msec)
01	122	.06	254	.12
1B	2046	1.00	4102	2.00
37	4118	2.01	8246	4.03
53	6190	3.02	12390	6.05
6E	8118	4.00	16386	8.00
7D	9298	4.54	18606	9.08
8A	10260	5.01	20530	10.02
FF	18918	9.24	37846	18.48

A.5.6 Shift HL Subroutine

SHLRT (022D)

SHLRZ (022E)

SHLRC (022F)

Shift the content of register pair HL right one bit, returning the previous least significant bit in carry. SHLRZ shifts zero into the high bit; SHLRC enters carry into the high bit. SHLRT tests the content of (HL) before shifting, and returns Zero set if the value was zero. These subroutines are useful in double precision multiplication and division.

Enter (HL) = two byte value to be shifted
Return (HL) shifted right one bit
(A) = (L)
Carry = previous LSB of (HL)

A.5.7 Breakpoint System Subroutines

BKLOC (01C3)

Tests whether a memory location exists in the breakpoint table. See Section 9.6.2

BKENT (01A3)

Enters a breakpoint. See Section 9.6.2.

BKRMV (0186)

Removes a breakpoint. See Section 9.6.2.

A.5.8 Move Memory

A monitor function allows copying the contents of a memory area either to a higher overlapping area or to a non-overlapping area.

Enter COPYL (0059) with:

(C) = number of bytes to be moved

(DE) = address of the highest byte to be moved

(HL) = new address for the highest byte.

Return is to the monitor at NORUN. Press RESET when finished. If more than 256 bytes are to be moved the registers must be loaded again and a new entry made.

A.5.9 Serial Data Communications

SOTBT (0382)

Returns successive bits for serial data transmission. Refer to Chapter 9, Section 9.6.2 for details

SINWS (03CF)

Receives serial data, returning after each character. Refer to Chapter 9, Section 9.6.3 for details.

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX B

BINARY/DECIMAL CONVERSIONS

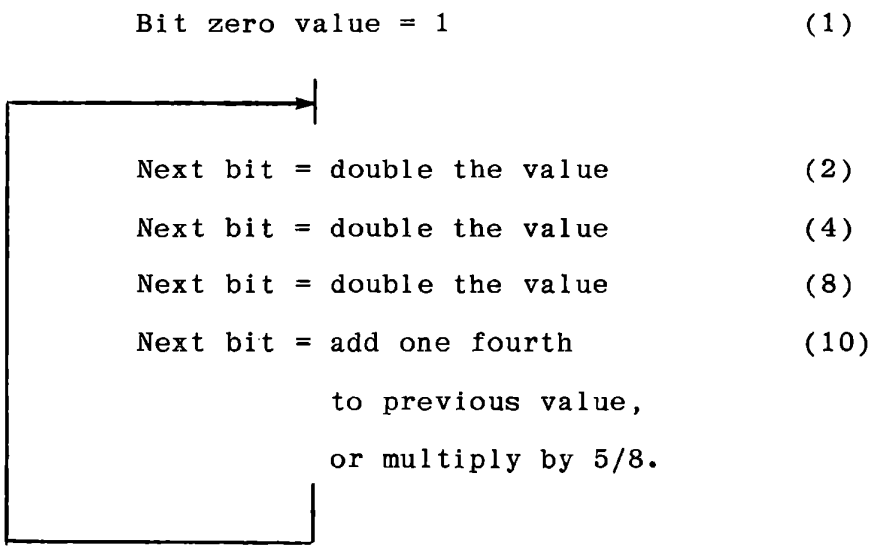
Appendix B

BINARY/DECIMAL CONVERSIONS

Several programs are presented for conversion of decimal data to binary data. All of these are written as subroutines; generally the data to be converted (or a memory address for the data) are entered in register pair HL and the result is returned in the same, with all other registers preserved.

B.1 DECIMAL TO BINARY INTEGER

The conversion from decimal data to binary can be done by calculating and summing the values of the successive bits. Figure B-1 lists the values of the bits. These can be calculated by the following procedure:



Bit	Decimal Value	Hex Value
0	1	0001
1	2	0002
2	4	0004
3	8	0008
4	10	000A
5	20	0014
6	40	0028
7	80	0050
8	100	0064
9	200	00C8
10	400	0190
11	800	0320
12	1000	03E8
13	2000	07D0
14	4000	0FA0
15	8000	1F40
16	10,000	2710
20	100,000	186A0
24	1,000,000	F4240
28	10,000,000	989680

Values of Bits in a Decimal Number

Figure B-1

The bit value can be calculated and added into the sum representing the binary value as each bit of the decimal value is processed, or they can be pre-calculated and stored. It is faster and simpler to store a table of the bit values, but this requires memory for the storage, as shown in the program of Figure B-2. The procedure of Figure B-3 calculates the values and pushes them into the stack; then recovers each bit value as the decimal value is shifted. Thus no memory is permanently allocated to the bit value. The stack is used for 38 bytes - six to save registers and 32 for bit values. Either subroutine meets the same specification, except for length.

DECBN Convert four digit packed decimal value to two byte binary.

Enter with decimal value in (HL)

Return with binary value in (HL)

All other registers are preserved.

The programs of Figure B-4 can be used to test either of these programs.

DECIMAL TO BINARY WITH TABLE

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE		COMMENTS
8 22 0	F5	PUSH PSW	
1	D5	PUSH D	
2	C5	PUSH B	
3	11	LXI D, 0000	
4	00		
5	00		
6	01	LXI B, 8240	Table address for
7	40		low byte of most
8	82		significant bit
822 9	0A	LDAX B	(A) ← low byte value
A	03	INX B	for MSB bit
B	29	DAD H	Shift decimal value
C	D2	JNC 8234	Skip add if
D	34		decimal bit = 0
E	82		
F	83	ADD E	Add table value
8 23 0	5F	MOV E, A	to decimal value
1	0A	LDAX B	in (DE)
2	8A	ADC D	
3	57	MOV D, A	
823 4	03	INX B	Address value of
5	7D	MOV A, L	next lower bit. Test
6	B4	ORA H	binary value for 0
7	C2	JNZ 8229	Loop until all
8	29		non-zero bits
9	82		have been converted
A	EB	XCHG	(HL) ← result
B	C1	POP B	Restore registers
C	D1	POP D	
D	F1	POP PSW	
E	C9	RET	
F			
8 0		ENTER WITH	
1		(HL) = PACKED DECIMAL	
2		4 DIGITS	
3		RETURN	
4		(HL) = BINARY EQUIVALENT	
5		ALL OTHER REGISTERS	
6		PRESERVED	
7			
8			Figure B-2a

TABLE OF BIT VALUES

ADDR		CODE		
8 24	0	40		Value of bit 15
	1	1F		
	2	A0		Bit 14
	3	0F		
	4	D0		Bit 13
	5	07		
	6	E8		Bit 12
	7	03		
	8	20		Bit 11
	9	03		
	A	90		Bit 10
	B	01		
	C	C8		Bit 9
	D	00		
	E	64		Bit 8
	F	00		
8 25	0	50		Bit 7
	1	00		
	2	28		Bit 6
	3	00		
	4	14		Bit 5
	5	00		
	6	0A		Bit 4
	7	00		
	8	08		Bit 3
	9	00		
	A	04		Bit 2
	B	00		
	C	02		Bit 1
	D	00		
	E	01		Bit 0
	F	00		
8	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			
	8			

Figure B-2b

DECIMAL TO BINARY WITH STACK

ADDR	CODE			
8 22 0	F5		PUSH PSW	
1	D5		PUSH D	
2	C5		PUSH B	
3	EB		XCHG	(DE) ← decimal value
4	3E		MVI A, 04	To count 4 digits
5	04			
6	21		LXI H, 0001	
7	01			
8	00			Store values
822 9	E5		PUSH H	1 10 100 1000
A	29		DAD H	2 20 200 2000
B	E5		PUSH H	
C	4D		MOV C, L	
D	44		MOV B, H	
E	29		DAD H	
F	E5		PUSH H	4 40 400 4000
8 23 0	29		DAD H	
1	E5		PUSH H	8 80 800 8000
2	09		DAD B	
3	3D		DCR A	
4	C2		JNZ 8229	
5	29			
6	82			
7	EB		XCHG	(HL) ← decimal value
8	11		LXI D, 0000	Clear binary value
9	00			
A	00			
B	3E		MVI A, 10	To count 16 bits
C	10			
823 D	C1		POP B	(BC) ← bit value
E	29		DAD H	(CY) ← high bit
F	D2		JNC 8245	Skip add if
3 24 0	45			decimal bit = 0
824 1	82			
2				
3			ENTER WITH	
4			(HL) = PACKED DECIMAL	
5			4 DIGITS	
6			RETURN	
7			(HL) = BINARY EQUIVALENT	
8				

Figure B-3a

ADDR		CODE													
8242		E	B		X	C	H	G							(HL)← binary value
	3	0	9		D	A	D		B						Add bit value
	4	E	B		X	C	H	G							(HL)← decimal value
824	5	3	D		D	C	R		A						Count bits
	6	C	2		J	N	Z		8	2	3	D			Loop until stack
	7	3	D												cleared of all
	8	8	2												bit values)
	9	E	B		X	C	H	G							
	A	C	1		P	O	P		B						Restore BC
	B	D	1		P	O	P		D						Restore DE
	C	F	1		P	O	P		P	S	W				Restore AF
	D	C	9		R	E	T								
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure B-3b

TEST FOR DECIMAL TO BINARY

ADDR	CODE												
8 20 0	CD		CALL		ENTWD								
1	46												
2	03												
3	CD		CALL		DECBN								
4	20												
5	82												
6	11		LXI		D, 83FF								
7	FF												
8	83												
9	CD		CALL		DWD2								
A	D4												
B	02												
C	C3		JMP		8200								
D	00												
E	82												
F													
8 0													
1													
2													
3													
4													
5													
6													
7													
8													
9													
A													
B													
C													
D													
E													
F													
3 0													
1													
2													
3													
4													
5													
6													
7													
8													

Figure B-4

A single byte conversion can use either of the foregoing procedures, but a simpler method results from separating the two decimal digits. The low digit, with a value from 0 to 9, is already in binary as well as binary coded decimal form. The high digit, 00 to 90, can be converted by a binary multiplication by 5/8, which only takes five steps.

```
RAR          (A)   X/2
MOV E,A      (C)   X/2
RAR          (A)   X/4
RAR          (A)   X/8
ADD E        (A)   X/2 + X/8
```

Figure B-5 shows the complete subroutine, which accepts the two digit decimal number in (L) and returns the binary equivalent in (L).

DECBI - DECIMAL TO BINARY INTEGER - ONE BYTE

ADDR		CODE													
822	0	F	5		PUSH		PSW								
	1	7	D		MOV		A, L								
	2	E	6		ANI		0F								(A) ← low digit
	3	0	F												
	4	6	7		MOV		H, A								(H) ← low digit
	5	A	D		XRA		L								(A) ← high digit
	6	1	F		RAR										
	7	6	F		MOV		L, A								(L) ← 1/2 high digit
	8	1	F		RAR										
	9	1	F		RAR										(A) ← 1/8 high digit
	A	8	5		ADD		L								(A) ← 5/8 high digit
	B	8	4		ADD		H								(A) ← binary value
	C	6	F		MOV		L, A								(L) ← binary value
	D	F	1		POP		PSW								
	E	C	9		RET										
	F														
8	0														
	1														
	2														
	3														
	4				ENTER		WITH								
	5				(L)		= PACKED DECIMAL								
	6						2 DIGITS								
	7				RETURN										
	8				(L)		= BINARY EQUIVALENT								
	9				USES		REGISTER H								
	A				ALL		OTHERS PRESERVED								
	B														
	C														
	D														
	E														
	F														
8	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure B-5

The procedure of Figure B-5 can also be used with multi-byte values. Almost any realistic program that requires decimal to binary conversion will also have a binary multiplication subroutine, which can be used to multiply the value of the two digit number by an appropriate power of 10 expressed in binary. These values can be stored in a table, or they can also be calculated by binary multiplication. This scheme is by far the best when more than four digits are involved.

B.2 DECIMAL FRACTION TO BINARY FRACTION

Surprisingly, the conversion of a decimal fraction to a binary fraction is significantly simpler than the conversion of integers. The decimal fraction is repeatedly doubled: if a carry out of the fraction results, a one is shifted into the binary value; if no carry occurs, a zero is shifted in. Figure B-6 shows a 16 bit conversion program. For larger numbers of bits, the data would be kept in memory, and the procedure can then be extended to any desired precision.

DCFBF - DECIMAL FRACTION TO BINARY FRACTION

ADDR	CODE															
8 826 0	F	5		PUSH		PSW										Save registers
1	D	5		PUSH		D										
2	1	1		LXI		D,	0	0	0	1						Clear binary fraction
3	0	1														with marker in
4	0	0														low bit
826 5	7	D	→	MOV		A,	L									Double decimal
6	8	7		ADD		A										fraction in (HL)
7	2	7		DAA												
8	6	F		MOV		L,	A									
9	7	C		MOV		A,	H									
A	8	F		ADC		A										
B	2	7		DAA												
C	6	7		MOV		H,	A									
D	7	B		MOV		A,	E									Shift carry from
E	1	7		RAL												decimal fraction
F	5	F		MOV		E,	A									into binary fraction
8 27 0	7	A		MOV		A,	D									
1	1	7		RAL												
2	5	7		MOV		D,	A									
3	D	2		JNC		8	2	6	5							Loop until marker
4	6	5														bit shifts out
5	8	2														
6	E	B		XCHG												(HL) ← binary fraction
7	D	1		POP		D										Restore other
8	F	1		POP		PSW										registers
827 9	C	9		RET												
A																
B																
C																
D																
E																
F																
3 0																
1																
2																
3																
4																
5																
6																
7																
8																

Figure B-6

B.3 BINARY TO DECIMAL CONVERSION

Since each bit in a binary number, either integer or fraction, has twice the value of the preceding bit, this conversion starts with a decimal value for the least significant bit and repeatedly doubles that value for succeeding bits. The successive bits of the binary value are tested, and each time a one is encountered, the bit value is summed into the decimal value.

The program of Figure B-7 operates in memory rather than in registers, and allows conversion of any number of bytes. It demonstrates passing parameters to a subroutine through memory with a command and address table. Five areas in memory are required:

Binary Data
Decimal Result
Temporary Bit Value
Value of Least Significant Bit
Command and Address Table

The conversion subroutine is entered with (HL) = address of the command and address table, which contains (in this order):

Number of binary bytes to be converted
Number of decimal bytes
Binary data address
Result address
Temporary bit value address
LSB value address

} address for least significant byte

The conversion program alters only the result and the temporary bit value. None of the other data are changed, so the binary value remains available for further processing and the other data could be stored in ROM.

A subroutine, RECAD, recovers these addresses and places them in registers for use in initialization and in the repetitive conversion loop. In the initialization, the least significant bit value is copied from its permanent location to the temporary bit value area, and the result area is closed.

In the loop, RECAD is called with a byte count in register C (initially set to 00), and RECAD adds this value to the binary data address from the table, returning the address of the binary data byte now being processed. The data byte addressed is masked by the content of register B (initially set to 01 and subsequently shifted left), giving the value of the current bit.

If the current bit is one, another subroutine, DCADM, is called to add the decimal value of the bit (addressed by BC) to the decimal result (addressed by HL). Then the bit value address is duplicated in HL and another call to DCADM adds the bit value to itself, giving the value of the next higher bit.

At the end of the loop, the bit mask and byte count are recovered, and the bit mask in register B is rotated left before repeating the loop. When it shifts from bit 7 back to bit 0, the byte count is incremented and compared with the number of bytes to be converted.

The command table shown is suitable for conversion of a four byte binary value with 16 integer bits and 16 fractional bits. The coding given is for locations 8280 to 82F4, with the command table, LSB value and scratch pad in 8300-831F; binary data and decimal result are in 8320-832E.

BINARY TO DECIMAL CONVERSION - INITIALIZE

ADDR		CODE					COMMENTS
8 28	0	F5		PUSH PSW			Save registers
	1	C5		PUSH B			
	2	D5		PUSH D			
	3	E5		PUSH H			Save command addr
	4	23		INX H			
	5	46		MOV B, M			(B) ← decimal bytes
	6	0E		MVI C, 00			Set 0 offset
	7	00					for RECAD
	8	23		INX H			Skip address for
	9	23		INX H			binary data
	A	23		INX H			Address result address
	B	CD		CALL RECAD			(HL) ← result address
	C	CC					(DE) ← bit value addr
	D	82					(BC) ← LSB value addr
828	E	77	→	MOV M, A			(M) ← decimal bytes
	F	0A		LDAX B			Copy LSB value
8 29	0	12		STAX D			into bit value
	1	7E		MOV A, M			(A) ← decimal bytes
	2	36		MVI M, 00			Clear decimal
	3	00					result
	4	23		INX H			
	5	13		INX D			
	6	03		INX B			
	7	3D		DCR A			
	8	C2	↳	JNZ 828E			
	9	8E					
	A	82					
	B	E1		POP H			Restore command addr
	C	01		LXI B, 0100			
	D	00					(C) ← byte count = 00
	E	01					(B) ← bit mask = 01
	F	00		NOP			
3	0						
	1						
	2						
	3						
	4						
	5						
	6						
	7						
	8						

Figure B-7a

BINARY TO DECIMAL CONVERSION - LOOP

ADDR		CODE														
8 2A	0	E	5		P	U	S	H		H						Save command address
	1	C	5		P	U	S	H		B						Save mask count
	2	2	3		I	N	X		H							Address decimal bytes
	3	E	5		P	U	S	H		H						
	4	2	3		I	N	X		H							
	5	C	D		C	A	L	L		R	E	C	A	D		(HL) ← decimal data addr
	6	C	C													(DE) ← result address
	7	8	2													(BC) ← bit value address
	8	A	6		A	N	A		M							Mask for current bit
	9	E	1		P	O	P		H							Address decimal bytes
	A	7	E		M	O	V		A	,	M					(A) ← decimal bytes
	B	E	B		X	C	H	G								(HL) ← result addr
	C	C	4		C	N	Z		D	C	A	D	M			If bit from binary
	D	E	0													data = 1 add bit
	E	8	2													value to result
	F	6	9		M	O	V		L	,	C					Duplicate bit value
8 2B	0	6	0		M	O	V		H	,	B					address in (HL)
	1	C	D		C	A	L	L		D	C	A	D	M		Add bit value to
	2	E	0													itself
	3	8	2													
	4	C	1		P	O	P		B							Recover mask, count
	5	E	1		P	O	P		H							Recover command addr
	6	7	8		M	O	V		A	,	B					
	7	0	7		R	L	C									Rotate bit mask
	8	4	7		M	O	V		B	,	A					left and restore
	9	D	2		J	N	C		8	2	A	0				Loop unless
	A	A	0													mask restored
	B	8	2													to right bit
	C	0	C		I	N	R		C							Count bytes
	D	7	E		M	O	V		A	,	M					Compare with
	E	B	9		C	M	P		C							final byte count
	F	D	2		J	N	C		8	2	A	0				Loop unless
3 2C	0	A	0													final byte finished
	1	8	2													
	2	D	1		P	O	P		D							EXIT - restore
	3	C	1		P	O	P		B							registers saved
	4	F	1		P	O	P		P	S	W					at Initialize
82C	5	C	9		R	E	T									
	6															
	7															
	8															

Figure B-7b

RECAD - RECOVER ADDRESS FROM MEMORY

ADDR		CODE														
8	0				ENTER				AT		82CC WITH					
	1					(HL)	=	ADDR	ESS OF ADDRESSES							
	2					(C)	=	OFFS	ET FROM FIRST							
	3															
	4				RETURN											
	5					(HL)	=	FIRS	T ADDRESS							
	6							PLUS	OFFSET (C)							
	7					(DE)	=	SECO	ND ADDRESS							
	8					(BC)	=	THIR	D ADDRESS							
	9					(A)	=	ENTR	Y CONTENT							
	A							OF	REGISTER B							
	B															
82C	C	7	E		MOV	A	,	M								
	D	8	1		ADD	C										
	E	5	F		MOV	E	,	A								Add offset in (C) to first address
	F	2	3		INX	H										
82D	0	7	E		MOV	A	,	M								
	1	C	E		ACI	0	0									
	2	0	0													
	3	5	7		MOV	D	,	A								
	4	D	5		PUSH	D										Save in Stack
	5	7	8		MOV	A	,	B								Returns (B) in (A)
	6	2	3		INX	H										
	7	5	E		MOV	E	,	M								(DE) ← second address
	8	2	3		INX	H										
	9	5	6		MOV	D	,	M								
	A	2	3		INX	H										
	B	4	E		MOV	C	,	M								(BC) ← third address
	C	2	3		INX	H										
	D	4	6		MOV	B	,	M								
	E	E	1		POP	H										(HL) ← first address + offset
	F	C	9		RET											
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure B-7c

DCADM - DECIMAL ADD IN MEMORY

ADDR		CODE													
82E	0	E5		PUSH		H									
	1	D5		PUSH		D									
	2	C5		PUSH		B									
	3	5F		MOV		E	,	A						(E) ← byte count	
	4	57		MOV		D	,	A						(D) ← byte count	
	5	AF		XRA		A									
82E	6	0A		LDAX		B									
	7	8E		ADC		M									
	8	27		DAA											
	9	77		MOV		M	,	A							
	A	03		INX		B									
	B	23		INX		H									
	C	1D		DCR		E									
	D	C2		JNZ		82E6									
	E	E6													
	F	82													
82F	0	7A		MOV		A	,	D						(A) ← byte count	
	1	C1		POP		B									
	2	D1		POP		D									
	3	E1		POP		H									
	4	C9		RET											
	5														
	6			ENTER		AND								RETURN WITH	
	7			(HL)	=	AUGEND		ADDRESS							
	8			(BC)	=	ADDEND		ADDRESS							
	9			(A)	=	BYTE		COUNT							
	A														
	B			ADDEND		PRESERVED									
	C			AUGEND		REPLACED BY SUM									
	D														
	E			ALL		REGISTERS		PRESERVED							
	F			EXCEPT		FLAGS									
3	0														
	1			CY		SET		IF						CARRY FROM	
	2			HIGH		BYTE		ADDITION							
	3														
	4														
	5														
	6														
	7														
	8														

Figure B-7d

COMMAND TABLE AND LSB VALUE

ADDR		CODE	
8 30	0	04	4 BINARY BYTES
	1	0B	11 DECIMAL BYTES
	2	20	BINARY DATA ADDRESS
	3	83	
	4	24	DECIMAL RESULT ADDRESS
	5	83	
	6	15	SCRATCH PAD ADDRESS
	7	83	
	8	0A	ADDRESS FOR LSB VALUE
	9	83	
830	A	25	VALUE OF LSB (11 BYTES)
	B	06	
	C	89	
	D	87	
	E	25	
	F	15	
8 31	0	00	
	1	00	
	2	00	
	3	00	
	4	00	
831	5	00	SCRATCH PAD (11 BYTES)
	6	00	
	7	00	
	8	00	
	9	00	
	A	00	
	B	00	
	C	00	
	D	00	
	E	00	
	F	00	
8 32	0	00	BINARY DATA
	1	00	
	2	00	
	3	00	
832	4	00	DECIMAL RESULT
	5	00	
	6	00	
	7	00	
	8		

Figure B-7e

B.4 BINARY FRACTION TO DECIMAL FRACTION

The program of Figure B-8 is a shortened version of the binary to decimal conversion, taking a two byte binary fraction in (HL) and returning the two byte decimal equivalent in (HL). For economy of program space it does not save the other registers, and returns only the two high bytes of the result in (HL). The other bytes of the conversion are stored in memory, with the least significant at 8308 and most significant at 830F. It requires that its scratch pad and result area occupy the lowest 16 bytes of the page immediately following the least significant bit value, which is stored at 82F8-82FF. The program would work for integers or mixed integer/fraction values if a different LSB value were stored in that location.

BFDCF - BINARY FRACTION TO DECIMAL FRACTION

ADDR		CODE													
8 2C	0	E	B		X	C	H	G							(DE)←binary fraction
	1	2	1		L	X	I		H	,	8	3	1	0	Address beyond
	2	1	0												scratch pad and
	3	8	3												result area
	4	A	F		X	R	A		A						Clear scratch pad
	5	2	D		D	C	R		L						and result
	6	7	7		M	O	V		M	,	A				(8300-830F)
	7	C	2		J	N	Z		8	2	C	5			End with
	8	C	5												(HL)=8300
	9	8	2												
	A	0	1		L	X	I		B	,	8	2	F	8	Address least
	B	F	8												significant byte
	C	8	2												of LSB value
82C	D	C	D		C	A	L	L		8	2	E	5		((HL))←((HL))+((BC))
	E	E	5												for 8 bytes
	F	8	2												Returns (BC)=8300
8 2D	0	7	A		M	O	V		A	,	D				
	1	1	F		R	A	R								Shift binary
	2	5	7		M	O	V		D	,	A				value right
	3	7	B		M	O	V		A	,	E				one bit
	4	1	F		R	A	R								
	5	5	F		M	O	V		E	,	A				
	6	D	C		C	C			8	2	E	5			If LSB=1 add
	7	E	5												bit value (8300-8307)
	8	8	2												to result (8308-830F)
	9	6	9		M	O	V		L	,	C				(HL)← 8300
	A	7	A		M	O	V		A	,	D				Test binary value
	B	B	3		O	R	A		E						for zero
	C	C	2		J	N	Z		8	2	C	D			Loop until zero
	D	C	D												
	E	8	2												
	F	2	A		L	H	L	D		8	3	0	E		(HL)←result
3 2E	0	0	E												(two high bytes)
	1	8	3												Complete result
	2	C	9		R	E	T								is in (8308-830F)
	3	0	0		N	O	P								
	4	0	0		N	O	P								
	5				E	N	T	E	R		(	H	L	)	=BINARY FRACTION
	6				R	E	T	U	R	N	(	H	L	)	=DECIMAL FRACTION
	7				A	L	L		R	E	G	I	S	T	ERS USED
	8														

Figure B-8a

DECIMAL ADD IN MEMORY FOR BEDCF

ADDR		CODE														
82E	5	D	5		P	U	S	H		D						Save binary value
	6	1	E		M	V	I		E	,	0	8				Count 8 bytes
	7	0	8													
	8	A	F		X	R	A		A							Clear CY for first byte
82E	9	0	A		L	D	A	X		B						(A) ← value of bit
	A	8	E		A	D	C		M							Add in decimal
	B	2	7		D	A	A									to bit value or
	C	7	7		M	O	V		M	,	A					result
	D	2	3		I	N	X		H							
	E	0	3		I	N	X		B							
	F	1	D		D	C	R		E							Repeat for all bytes
82F	0	C	2		J	N	Z		8	2	E	9				
	1	E	9													
	2	8	2													
	3	4	4		M	O	V		B	,	H					Returns (BC) = 8300
	4	4	B		M	O	V		C	,	E					(HL) incr. by 8
	5	D	1		P	O	P		D							Restore binary value
	6	C	9		R	E	T									
	7	0	0		N	O	P									
	8	2	5													Value of LSB
	9	0	6													
	A	8	9													
	B	8	7													
	C	2	5													
	D	1	5													
	E	0	0													
	F	0	0													
30	0															Scratch-pad and
	1															result area
	2															Must be
	3															8300 - 830F
	4															
	5															
	6															
	7															
	8															

Figure B-8b

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX C

CALCULATING TRIGONOMETRIC FUNCTIONS

Appendix C

CALCULATING TRIGONOMETRIC FUNCTIONS

The sine of an angle (in radians) is calculated from:

$$\text{(a)} \quad \sin x = x - \frac{x^3}{3!} + \frac{x^5}{4!} - \frac{x^7}{7!} + \text{---}$$

The cosine is generated by a similar series:

$$\text{(b)} \quad \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \text{---}$$

The exponential function e^x is:

$$\text{(c)} \quad e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{---}$$

All three functions can be generated simultaneously by a procedure that calculates each successive term in the series for e^x ; adds the terms into a sum for e^x ; and adds or subtracts each term to a sine or cosine sum. Each term is calculated from the preceding term, the term number, and the value of x.

$$t_i = x t_{i-1} / i$$

Starting with t = 1, this gives:

Term	Value	Disposition
0	1	Enter to cosine
1	x/1	Enter to sine
2	x /2	Subtract from cosine
3	x /3.2	Subtract from sine
4	x /4.3.2	Add to cosine
5	x /5.4.3.2	Add to sine
6	x /6.5.4.3.2	Subtract from cosine

The value of x must be expressed in radians, and for reasonably rapid convergence of the series large values of x should be avoided. Since the sine of an angle is equal to the cosine of its complement:

$$\sin x = \cos \left(\frac{\pi}{2} - x\right)$$

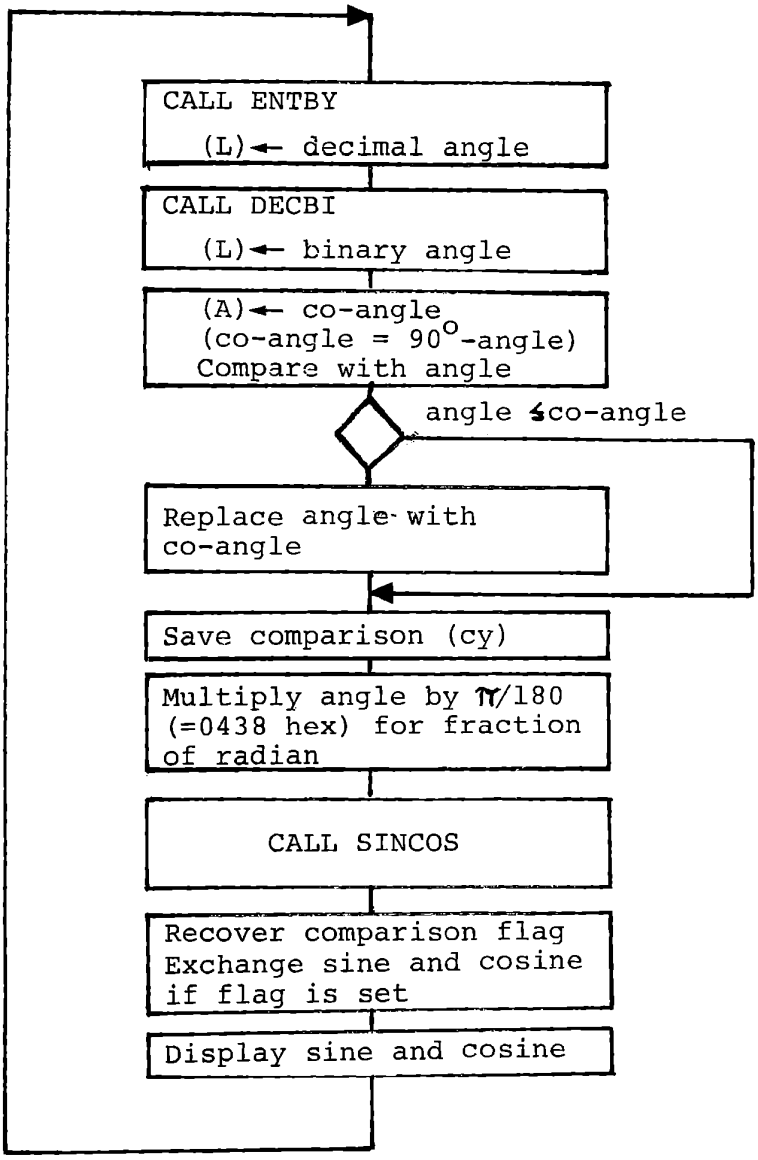
it is easy to restrict the angle to less than 45 , or 0.785 radians. With this limit terms beyond 6 are not needed for 16 bit precision.

In this Appendix, we present a subroutine to calculate the sine and cosine, given x as a value between 0.0 and 0.785 radians. A main program (Figure C-1) will accept an angle in decimal degrees and convert it to binary radians, call SINCOS, and display the results in decimal.

The program also uses a binary multiplication subroutine and a twos complement subroutine, presented in the following pages; the single byte decimal to binary integer conversion of Figure B-5 and the two byte binary fraction to decimal fraction conversion of Figure B-8, in Appendix B. These are also duplicated here.

Memory assignments for the program are:

MAIN	8200-823F
SINCOS	8250-827F
TERM	8280-82AF
DECBI	82B0-82BF
BFDCF	82C0-82FF
Variable Data	8300-830F
BMULT	8310-8330
TWOSC	8336-833F



Main Program

Figure C-1

MAIN - ACCEPT DECIMAL ANGLE, DISPLAY SIN, COS

ADDR		CODE														
8 20	0	CD		CALL				ENTBY								(L)← angle in
	1	36														decimal degrees
	2	03														
	3	CD		CALL				DECBI								(L)← angle in
	4	B0														binary degrees
	5	82														
	6	3E		MVI		A,	5A									(A)← 90° binary
	7	5A														
	8	95		SUB		L										(A)← co-angle
	9	BD		CMP		L										Set CY if angle >
	A	F5		PUSH		PSW										Save CY
	B	D2		JNC		820F										Jump if
	C	0F														angle ≤ co-angle
	D	82														Else replace
	E	6F		MOV		L,	A									angle with co-angle
820	F	26		MVI		H,	00									Clear high byte
8 21	0	00														
	1	01		LXI		B,	0478									Multiply by π/180
	2	78														for angle in radians
	3	04														
	4	CD		CALL		BMULT										(DE)← angle as
	5	10														binary fraction
	6	83														of one radian
	7	EB		XCHG												
	8	CD		CALL		SINCOS										(HL)← sine
	9	50														(DE)← cosine
	A	82														Recover co-angle
	B	F1		POP		PSW										flag. If set
	C	D2		JNC		8220										exchange sine
	D	20														and cosine of
	E	82														co-angle
821	F	EB		XCHG												(HL)← cosine
3	0															
	1															
	2															
	3															
	4															
	5															
	6															
	7															
	8															

Figure C-2a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

A D D	R	CODE				
8 22	0	D	5		PUSH D	Save sine
	1	C	D		CALL BFDCF	Convert cosine
	2	C	0			to decimal
	3	8	2			
	4	C	D		CALL DWORD	Display at left
	5	D	1			
	6	0	2			
	7	E	1		POP H	
	8	C	D		CALL BFDCF	Convert sine
	9	C	0			to decimal
	A	8	2			
	B	1	1		LXI D,83FF	Display at right
	C	F	F			
	D	8	3			
	E	C	D		CALL DWDZ	
	F	D	4			
8	0	0	2			
	1	C	3		JMP 8200	
	2	0	0			
	3	8	2			
	4					
	5					
	6					
	7					
	8					
	9					
	A					
	B					
	C					
	D					
	E					
	F					
8	0					
	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					Figure C-2b

Subroutines SINCOS and TERM are defined in the text below and depicted in Figures C-3 and C-4. SINCOS adds or subtracts successive terms, as discussed early in this Appendix. TERM generates the terms, addressing a table of coefficients according to the term number. These coefficients are nominally 1/2, 1/3, 1/4, 1/5, etc. Adjustments to the coefficients for terms 5 and 6 are made as shown in the table of Figure C-5 to correct for rounding errors and absent higher order terms.

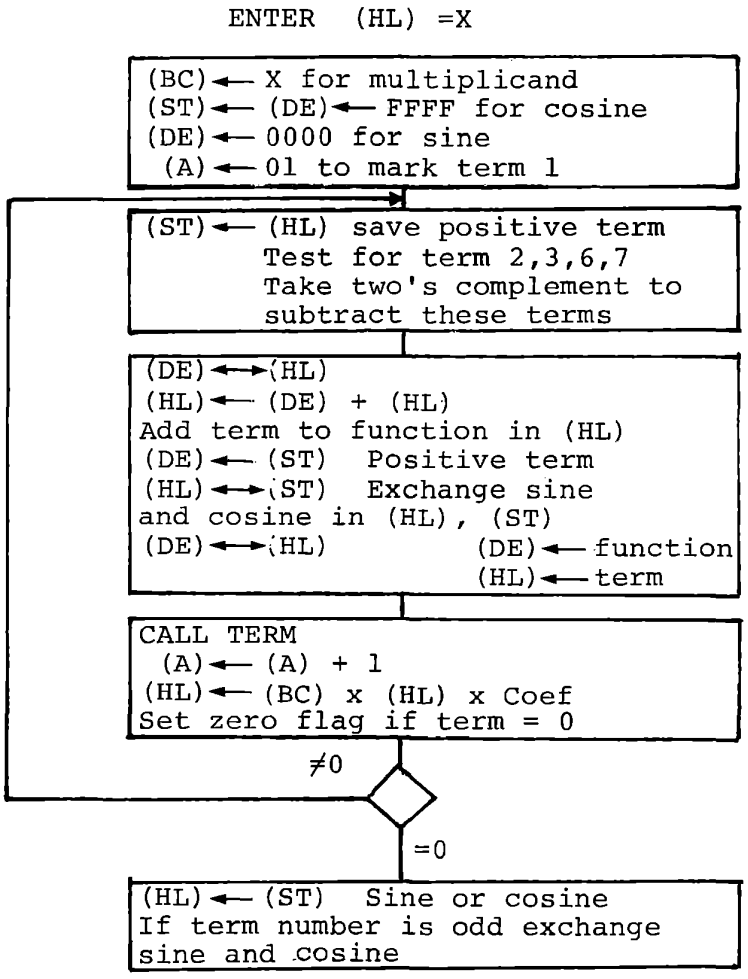
The table of Figure C-5 shows the results returned by this program. Note that the adjusted coefficients affect only the least significant digit, for angles between 40 and 50 degrees. The adjustment may be important in some instances, to make sin 45^{O} = cos 45^{O}.

SINCOS	Find the sine and cosine of X
Enter with	(HL) = X
Return with	(BC) = X
	(DE) = sin X
	(HL) = cos X

Constraints: X must be a fractional value (i.e. less than 1). The cosine of zero is returned as FFFF.

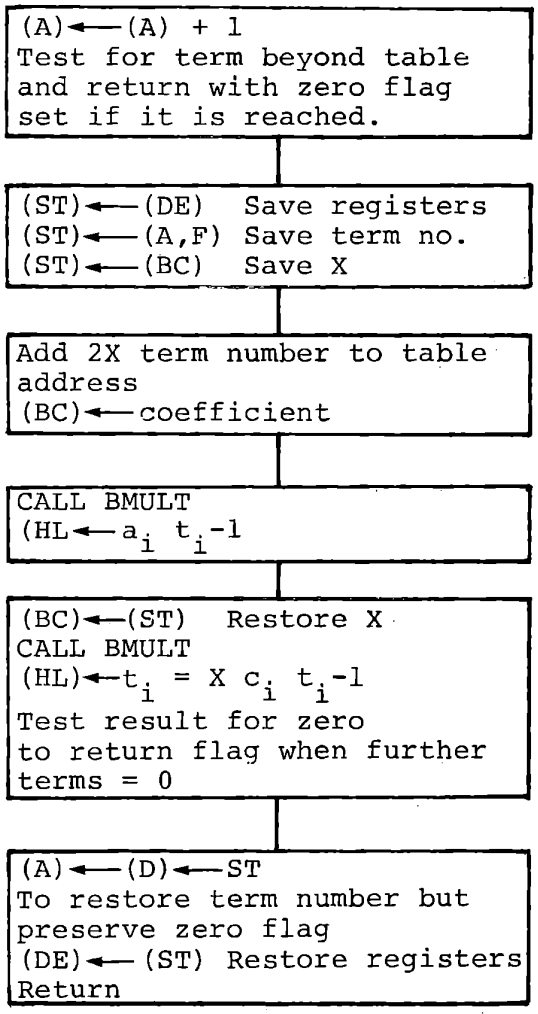
TERM	Find successive terms of e^x
Enter with	(A) = term number 1 to 8
	(BC) = X
	(HL) = previous term
Return with	(A) = next term number
	(BC) = X
	(HL) = new term

Requires a table of values of 1/(A). Term is always positive.



Subroutine SINCOS

Figure C-3



Subroutine Term

Figure C-4

Coefficients for Successive Terms

Term	Nominal Value		Adjusted	
	Decimal	Hex	Decimal	Hex
a_1	1.0000	-	1.0000	-
a_2	0.5000	8000	0.5000	8000
a_3	0.3333	5555	0.3333	5555
a_4	0.2500	4000	0.2500	4000
a_5	0.2000	3344	0.1953	3200
a_6	0.1667	2AAD	0.0937	1800
a_7	0.1429	2498	0	0
a_8	0.1250	2000	0	0
a_9	0.1111	1C72	0	0

Results of Sine/Cosine Calculation

Angle	Cosine		Sine	
0	0.9999		0.0000	
1	.9998		.0174	
2	.9993		.0349	
3	.9986		.0523	
4	.9975		.0697	
5	.9961		.0871	
10	.9847		.1736	
15	.9658		.2588	
20	.9396	With	.3420	With
25	.9062	adjusted	.4266	adjusted
30	.8659	coefficients	.5000	coefficients
35	.8191		.5736	
40	.7661*	.7660	.6428	
44	.7195*	.7193	.6949*	.6947
45	.7073*	.7071	.7072*	.7071
46	.6949*	.6947	.7195*	.7193
50	.6428		.7661*	.7660
60	.5000		.8659	
75	.2588		.9658	
90	0.0000		0.9999	

*Values with error least significant digit

Results in Sine/Cosine Calculation

Figure C-5

SINCOS - SINE AND COSINE OF X (X < 1 RADIAN)

ADDR		CODE															
8 25	0	4	D		M	O	V		C	,	L				(BC) ← x
	1	4	4		M	O	V		B	,	H				
	2	1	1		L	X	I		D	,	F	F	F	F	maximum value
	3	F	F												for cosine
	4	F	F												
	5	D	5		P	U	S	H		D					(ST) ← cosine
	6	1	3		I	N	X		D						(DE) ← sine = 00
	7	3	E		M	V	I		A	,	0	1			Mark term 1
	8	0	1												
825	9	E	5		P	U	S	H		H					Save positive term
	A	0	F		R	R	C								Test bit 1 of term
	B	0	F		R	R	C								number.
	C	D	C		C	C		T	W	O	S	C			If term is 2, 3, 6, 7
	D	3	6												take two's complement
	E	8	3												of term to subtract
	F	0	7		R	L	C								Restore term
826	0	0	7		R	L	C								number.
	1	E	B		X	C	H	G							(DE) ← term or comp
	2	1	9		D	A	D		D						(HL) ← function
	3	D	1		P	O	P		D						(DE) ← positive term
	4	E	3		X	T	H	L							Exchange sine/cosine
	5	E	B		X	C	H	G							(HL) ← term
	6	C	D		C	A	L	L		T	E	R	M		(HL) ← next term
	7	8	0												(A) ← next number
	8	8	2												
	9	C	2		J	N	Z		8	2	5	9			Term sets zero flag
	A	5	9												after last term
	B	8	2												or if term = 0
	C	E	1		P	O	P		H						(HL) ← sine or cosine
	D	0	F		R	R	C								If term number
	E	D	8		R	C									even, return
	F	E	B		X	C	H	G							Else exchange
827	0	C	9		R	E	T								sine and cosine
	1				E	N	T	E	R		W	I	T	H	
	2					(	H	L	)	=		x			
	3				R	E	T	U	R	N					
	4					(	B	C	)	=		x			
	5					(	D	E	)	=		S	I	N	x
	6					(	H	L	)	=		C	O	S	x
	7														
	8														

Figure C-6

TERM - GENERATE SUCCESSIVE TERMS OF e

ADDR		CODE													
8 28	0	3C		INR			A								Next term number
	1	FE		CPI			09								Test for past
	2	09													last term
	3	C8		RZ											Return zero set
	4	D5		PUSH			D								Save function
	5	F5		PUSH			PSW								Save term number
	6	C5		PUSH			B								Save value of x
	7	87		ADD			A								Double term number
	8	11		LXI			D,	829C							Address -4 for
	9	9C													table of coefficients
	A	82													a_0 and a_1 not
	B	83		ADD			E								stored in table
	C	5F		MOV			E,	A							Address a_i
	D	1A		LDAX			D								}
	E	4F		MOV			C,	A							
	F	13		INX			D								} $(BC) \leftarrow a_i$
8 29	0	1A		LDAX			D								
	1	47		MOV			B,	A							}
	2	CD		CALL			BMULT								$(HL) \leftarrow a_i t_{i-1}$
	3	10													
	4	83													
	5	C1		POP			B								$(BC) \leftarrow x$
	6	CD		CALL			BMULT								$(HL) \leftarrow t_i$
	7	10													$= a_i t_{i-1} x$
	8	83													
	9	7C		MOV			A,	H							Test for
	A	B5		ORA			L								zero result
	B	D1		POP			D								(D) ← term number
	C	7A		MOV			A,	D							
	D	D1		POP			D								(DE) ← function
	E	C9		RET											
	F	00		NOP											
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														
	8														

Figure C-7a

COEFFICIENTS FOR TERM

ADDR	CODE				
82A 0	0	0			$a_2 = 1/2$
1	8	0			
2	5	5			$a_3 = 1/3$
3	5	5			
4	0	0			$a_4 = 1/4$
5	4	0			
6	4	4	00		$a_5 = 1/5$
7	3	3	32		
8	A	D	00		$a_6 = 1/6$
9	2	A	18		
A	9	8	00		$a_7 = 1/7$
B	2	4	00		
C	0	0	00		$a_8 = 1/8$
D	2	0	00		
E	7	2	00		$a_9 = 1/9$
F	1	C	00		
80			↑	ADJUSTED COEFFICIENTS	
1					
2		↑			
3		└		EXACT COEFFICIENTS	
4					
5					
6					
7					
8					
9					
A					
B					
C					
D					
E					
F					
30					
1					
2					
3					
4					
5					
6					
7					
8					

Figure C-7b

DECBI - DECIMAL TO BINARY INTEGER - ONE BYTE

ADDR		CODE													
8	2B 0	F	5		PUSH		PSW								
	1	7	D		MOV		A,	L							
	2	E	6		ANI		0F								(A) ← low digit
	3	0	F												
	4	6	7		MOV		H,	A							(H) ← low digit
	5	A	D		XRA		L								(A) ← high digit
	6	1	F		RAR										
	7	6	F		MOV		L,	A							(L) ← ½ high digit
	8	1	F		RAR										
	9	1	F		RAR										(A) ← ¼ high digit
	A	8	5		ADD		L								(A) ← ⅝ high digit
	B	8	4		ADD		H								(A) ← binary value
	C	6	F		MOV		L,	A							
	D	F	1		POP		PSW								
	E	C	9		RET										
	F	0	0												
8	0														
	1														
	2														
	3				ENTER		WITH								
	4				(L)	=	PACKED DECIMAL BYTE								
	5				RETURN		WITH								
	6				(L)	=	BINARY INTEGER								
	7				REGISTER		H	USED							
	8				OTHER		REGISTER PRESERVED								
	9														
	A														
	B														
	C														
	D														
	E														
	F														
3	0														
	1														
	2														
	3														
	4														
	5														
	6														
	7														Duplicate
	8														Figuer B-5

BFDCF - BINARY FRACTION TO DECIMAL FRACTION

A D D R		CODE														
8 2C	0	E	B		X	C	H	G								
	1	2	1		L	X	I		H	,	8	3	1	0		
	2	1	0													
	3	8	3													
	4	A	F		X	R	A		A							
	5	2	D		D	C	R		L							
	6	7	7		M	O	V		M	,	A					
	7	C	2		J	N	Z		8	2	C	5				
	8	C	5													
	9	8	2													
	A	0	1		L	X	I		B	,	8	2	F	8		
	B	F	8													
	C	8	2													
82C	D	C	D		C	A	L	L		8	2	E	5			
	E	E	5													
	F	8	2													
8 2D	0	7	A		M	O	V		A	,	D					
	1	1	F		R	A	R									
	2	5	7		M	O	V		D	,	A					
	3	7	B		M	O	V		A	,	E					
	4	1	F		R	A	R									
	5	5	F		M	O	V		E	,	A					
	6	D	C		C	C			8	2	E	5				
	7	E	5													
	8	8	2													
	9	6	9		M	O	V		L	,	C					(HL) ← 8300
	A	7	A		M	O	V		A	,	D					
	B	B	3		O	R	A		E							
	C	C	2		J	N	Z		8	2	C	D				
	D	C	D													
	E	8	2													
	F	2	A		L	H	L	D		8	3	0	E			
8 2E	0	0	E													
	1	8	3													
	2	C	9		R	E	T									
	3	0	0		N	O	P									
	4	0	0		N	O	P									
	5															
	6															
	7															Duplicate
	8															Figure B-8a

CODING SHEET

MICROCOMPUTER TRAINING SYSTEM

INTEGRATED COMPUTER SYSTEMS

ADDR	CODE														
82E5	D5		P	U	S	H		D							
6	1E		M	V	I		E	,	0	8					
7	08														
8	AF		X	R	A		A								
82E9	0A		L	D	A	X		B							
A	8E		A	D	C		M								
B	27		D	A	A										
C	77		M	O	V		M	,	A						
D	23		I	N	X		H								
E	03		I	N	X		B								
F	1D		D	C	R		E								
82F0	C2		J	N	Z		8	2	E	9					
1	E9														
2	82														
3	44		M	O	V		B	,	H						
4	4B		M	O	V		C	,	E						
5	D1		P	O	P		D								
6	C9		R	E	T										
7	00		N	O	P										
8	25														
9	06														
A	89														
B	87														
C	25														
D	15														
E	00														
F	00														
30															
1															
2															
3															
4															
5															
6															
7															Duplicate
8															Figure B-8b

BMULT (HL) x (BC) --> (H,L,D,E)

ADDR		CODE														
8 31	0	E	B		X	C	H	G								(DE) ← multiplier
	1	A	F		X	R	A		A							Clear carry
	2	6	F		M	O	V		L	,	A					and product
	3	6	7		M	O	V		H	,	A					
	4	E	5		P	U	S	H		H						
	5	2	E		M	V	I		L	,	1	1				(L) ← count
	6	1	1													
831	7	E	3		X	T	H	L								(HL) ← product
	8	D	2		J	N	C		8	3	1	C				Jump if multiplier
	9	1	C													bit = 0
	A	8	3													
	B	0	9		D	A	D		B							Add multiplicand
831	C	C	D		C	A	L	L		8	3	2	9			Shift product
	D	2	9													Returns in (DE)
	E	8	3													
	F	C	D		C	A	L	L		8	3	2	9			Shift multiplier
8 32	0	2	9													Return in (DE)
	1	8	3													
	2	E	3		X	T	H	L								
	3	2	D		D	C	R		L							
	4	C	2		J	N	Z		8	3	1	7				
	5	1	7													
	6	8	3													
	7	E	1		P	O	P		H							(HL) ← high product
	8	C	9		R	E	T									
832	9	7	C		M	O	V		A	,	H					
	A	1	F		R	A	R									
	B	6	7		M	O	V		H	,	A					
	C	7	D		M	O	V		A	,	L					
	D	1	F		R	A	R									
	E	6	F		M	O	V		L	,	A					
	F	E	B		X	C	H	G								
8 33	0	C	9		R	E	T									
	1															
	2				E	N	T	E	R		W	I	T	H		
	3					(	B	C	)	=	M	U	L	T		IPLICAND
	4					(	H	L	)	=	M	U	L	T		IPLIER
	5				R	E	T	U	R	N						
	6					(	H	L	)	=	H	I	G	H		PRODUCT
	7					(	D	E	)	=	L	O	W			PRODUCT
	8					(	B	C	)		P	R	E	S		ERVED

Figure C-8

TWOSC - TWOS COMPLEMENT OF (HL)

ADDR	CODE			
8000				
8001				
8002				
8003				
8004				
8005				
8006	F5		PUSH PSW	
8007	7C		MOV A,H	
8008	2F		CMA	
8009	67		MOV H,A	
800A	7D		MOV A,L	
800B	2F		CMA	
800C	6F		MOV L,A	
800D	23		INX H	
800E	F1		POP PSW	
800F	C9		RET	
8010				
8011				
8012				ENTER WITH
8013				(HL) = BINARY NUMBER
8014				RETURN
8015				(HL) = TWOS COMPLEMENT
8016				OTHER REGISTERS
8017				PRESERVED
8018				
8019				
801A				
801B				
801C				
801D				
801E				
801F				
8020				
8021				
8022				
8023				
8024				
8025				
8026				
8027				
8028				

Figure C-9

This page intentionally left blank.

MICROCOMPUTER TRAINING WORKBOOK

ADDENDIX D

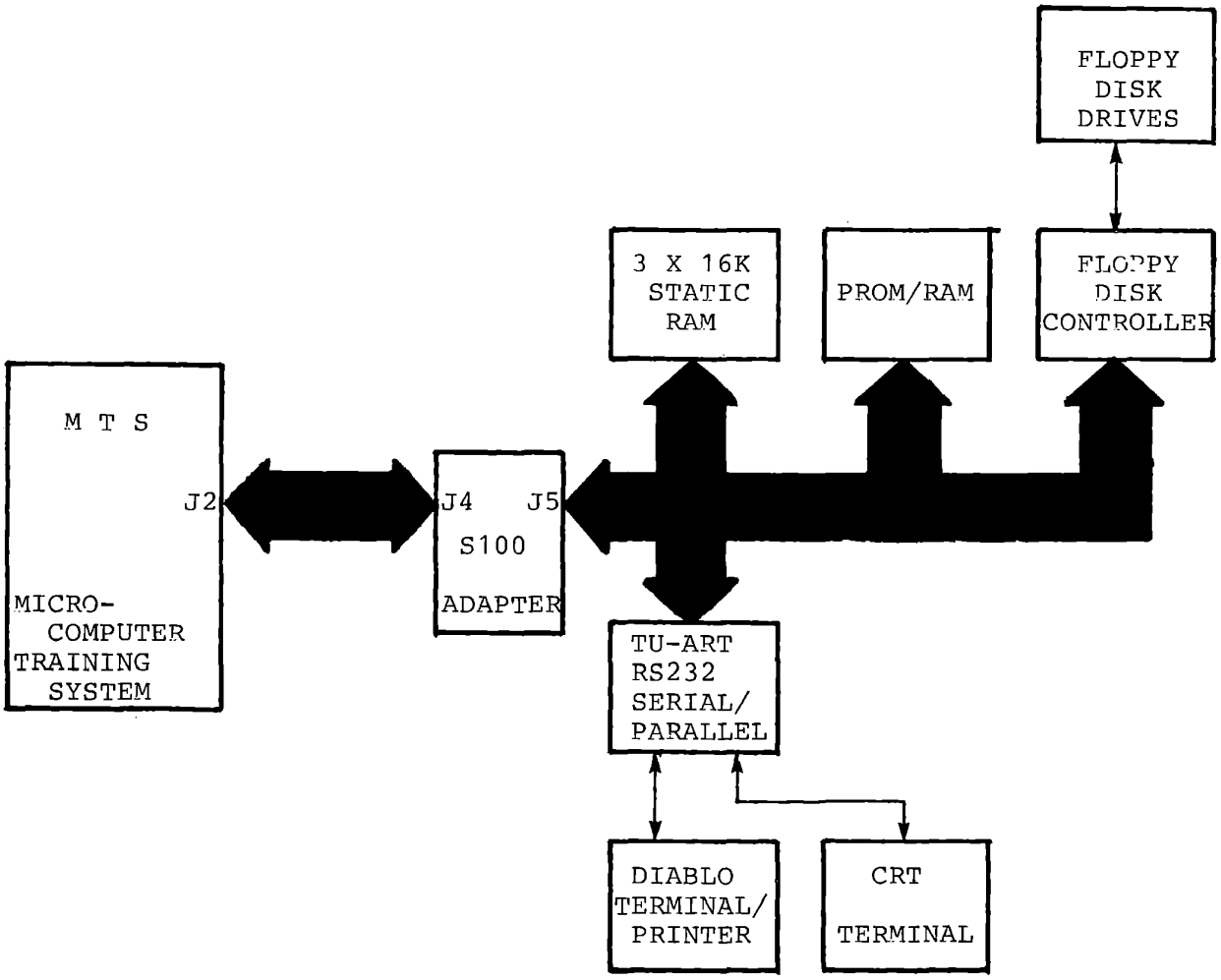
EXPANDING YOUR MICROCOMPUTER TRAINING SYSTEM

WITH THE S-100 BUS ADAPTER

EXPANDING YOUR MICROCOMPUTER TRAINING SYSTEM

Your Microcomputer Training System (MTS) can be easily expanded to a more complete system with floppy disk drives, printers, CRT terminals and other peripherals as illustrated in Figure D-1. This is easily accomplished via the S-100 Adapter Card which interfaces the MTS to any standard S-100 mainframe, into which you can insert other S-100 compatible circuit cards. The S-100 bus has rapidly become the industry standard for personal computers and as a result, has nurtured the explosion of many compatible peripherals.

To expand your system, you will need an S-100 bus mainframe to house the S-100 compatible cards. The mainframe should contain at least 12 - 16 slots to allow ample room for expansion. Each slot will hold one card, for example a 16K RAM card, a disk controller card, or a serial interface card. Note also that in addition to allowing one slot for each card you have planned, it is wise to leave one empty slot between each of the RAM cards for thermal stability. This is especially true for static RAM cards, which dissipate considerable heat.



Typical Expanded MTS-S100 System Configuration

Figure D-1

Generally the mainframe will contain a built-in power supply. Note that the S-100 power supplies furnish unregulated DC power on the bus and that each individual card in the system, including the MTS S-100 Bus Adapter, has its own on-board regulators to create the necessary regulated voltages for that board. You should be certain that the mainframe has a large enough power supply to power the S-100 cards. Typical current and voltage outputs are: +8V @ 30A, +18V @ 15A, -18V @ 15A.

TEI manufactures a mainframe which has been used in testing the MTS S-100 adapter. (For list of distributor addresses, see Table I.)
What S-100 compatible cards and peripherals should you buy? The answer of course is dependent on your application requirements, which will determine the extent of memory and type of peripherals necessary. However, almost any expanded system will involve adding more than the 4K bytes of RAM and 8K bytes of PROM available on the MTS itself. Thus, the first boards you will probably buy are RAM memory cards. Most operating systems that run Basic, Fortran, Editors, and Assemblers, require at least 32K bytes of RAM, so plan on buying at least two 16K RAM boards.

RAM cards from many different manufacturers can be used. However, you should select one that responds to the S-100 bus control signal named 'PHANTOM.' As described in detail later, the PHANTOM signal is issued by the S-100 Adapter card whenever a RESET occurs. The purpose of the PHANTOM signal is to disable all memory cards and thereby allow the S-100 adapter to force a 'JMP XXXX' instruction onto the bus as the first instruction to be executed after the RESET.

You can specify the jump address, XXXX, in switch settings on the S-100 adapter card as described later in this section.

The PHANTOM signal thus allows the S-100 adapter card to override the RAM memory at address 0 after a RESET, so that the processor can automatically jump to a specified address (other than 0000) to begin program execution. This is especially important since most S-100 disk operating systems have a 'Bootstrap Loader' program in a ROM located at a high memory address. Such Bootstrap Loaders are designed to be executed automatically following RESET. They read the Disk Operating System itself into the memory from the disk. To implement this function, the RAM cards you select must respond to the PHANTOM signal.

You will probably want to communicate with the microcomputer using an ASCII keyboard entry device and a CRT (Cathode Ray Tube) display (also commonly referred to as a VDU - Video Display Unit). Most terminals communicate through the standard EIA RS232C serial convention; therefore you will need a serial/parallel I/O card. Cromemco manufactures an S-100 card called the TU-ART which has two RS232C serial ports and two parallel I/O ports as well. You can connect any RS232C compatible VDU terminal, printer or modem to the TU-ART since it has software programmable communication rates. The TU-ART has been tested with a DIABLO printer/terminal and a Lear Siegler ADM-3A CRT terminal. Newbury Labs, a manufacturer in England, also manufactures a low cost VDU suitable for connection.

To efficiently develop and execute programs using higher-level languages, you will ultimately need a Disk Operating System and therefore, disk drives. Micropolis Corporation has a diskette drive system which includes an S-100 compatible controller card and a disk operating system called MDOS. This system runs well with the MTS. The MDOS software is rather powerful, and includes utility programs to read and write files on the disk, keep a directory and other 'housekeeping' functions. The system also includes a text editor, an assembler and a reasonably powerful version of BASIC.

Finally, you might want to include video monitors with controllers employing memory mapped techniques for use in word processing and limited graphics displays. You may even add speech recognition and synthesizing capabilities are S-100 compatible Digital to Analog/Analog to Digital converter cards, multi-channel data acquisition systems, musical synthesizers, relay controllers and other interesting new devices constantly appearing in the Personal Computer market. You are limited only by your imagination, since you can even design your 'better mouse-trap' onto blank S-100 card-size printed circuit boards.

To keep up-to-date on these developments, you may want to subscribe to one of the many magazines now devoted to covering the personal computer field. Two of the leading magazines are:

BYTE MAGAZINE
BYTE Subscriptions
P.O. Box 590
Martinsville, New Jersey 08836

INTERFACE AGE MAGAZINE
13913 Artesia Blvd.
Cerritos, California 90701

S-100 Module/Hardware	U.S. Distributor	European Distributor
Micro System Main-Frame MCs - 112	TEI, Inc. 5636 Etheridge Houston, Texas 77087 (713) 738-2300 TWX: 1-910-881-3639	(Sub-Distributor) CMC Marketing (Dealer) Abacus Computers 62 New Cavendish Street London, England WIM7LD
TU-ART Digital Interface Contains: 2 channels of duplex data exchange. 2 channels of parallel data exchange. 10 interval timers.	CROMEMCO, Inc. 2400 Charleston Rd. Mountain View, CA. 94043 (415) 964-7400	Hiltrup 4400, Munster, W. Germany BASIS MICROCOMPUTER VERTRIEB Von-Flotow-Strabe 5 02501-4800
16K Dynamic RAM Memory	DYNABYTE, Inc. 4020 Fabian Way Palo Alto, CA. 94303 (415) 965-1010	BELVEDERE COMPUTERS 9 Belvedere Place Scarborough North Yorkshire England YO1122X
16K Static RAM Memory Module	PROBLEM SOLVER SYSTEMS 20834 Lassen Street Chatsworth, CA. 91311 (213) 998-5100	FIMECA FA Avenue Albert 125 1060 Brussels, Belgium Phone: (02) 345-98-37
VDM-1 Video Display Module	PROCESSOR TECHNOLOGY 7100 Johnson Industrial Pleasanton, CA. 94566 (415) 829-2600	SPA TALTEC Square Larousse, 5 1060 Bruxelis Phone: (02) 345-98-95 WERNOR ELECTRONICS Torsvagen 61 Box 72, S-133 01 Saltsjobaden Stockholm, Sweden Phone: 08717-6288
Disk Storage Module - Model 1023 11 and Controller	MICROPOLIS CORP. 7959 Deering Avenue Canoga Park, CA. 91304 (213) 703-1121	TEKELEC-AIRTRONIC Cite des Bruyeres rue Carle Vernet 92310 Sevres, France Phone: (1) 027-75-35
CRT Terminal	----------	NEWBURY LABORATORIES LTD. King Street Odiham Hampshire RG25 INN, UK Telex: 858815

Manufacturers and Distributors of

Representative S-100 Compatible Equipment

Table I

THE S-100 ADAPTER CARD

S-100 ADAPTER DESIGN

The S-100 bus was originally specified by MITS, a company in Albuquerque, New Mexico, when they designed the first computer aimed at hobbyists - the Altair 8800A. Soon after the introduction of the 8800A, other manufacturers began producing compatible memory and peripheral boards which would operate on the 8800A bus. Various unused lines in the 100 line structure soon began to be defined by those manufacturers, sometimes in conflicting ways. MITS themselves defined new signals in their 8800B, the successor to the model A. Despite these occasional conflicts, the S-100 is so close to a universal standard that literally dozens of manufacturer's equipment and software can be used together.

To achieve optimum compatibility, the S-100 Bus Adapter Card has been designed not only to emulate the Altair 8800B definition of the S-100 bus, but also to incorporate additional signals used by other manufacturers (such as the PHANTOM), where these are compatible with the 8800B and are generally useful. While some manufacturer's equipment may not meet these standards, most will. You should have little difficulty if you follow the guidelines presented in this section.

The basic difference between the MTS microcomputer bus and the S-100 bus is that the MTS uses a single bi-directional data bus, as do most 8080 systems. The S-100 bus involves two separate 8-bit data buses; one for outbound and one for inbound data. The other key difference in the S-100 bus involves the status byte, placed on the data bus by the 8080 during the first clock cycle of each bus cycle. In most

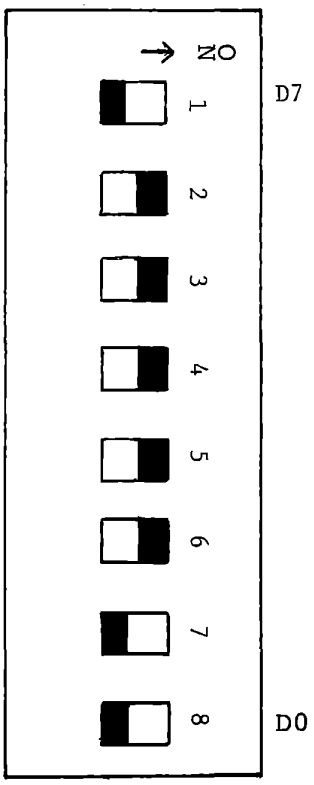
8080 systems, including the MTS, this status byte is decoded either with an 8228 or with discrete logic, and this logic then produces MEMR, MEMW, IOR, and IOW signals. Therefore, in most systems the status byte itself is 'discarded' following decoding. However, in the S-100 bus definition, this status byte must be stored in a register and made available on eight separate S-100 control lines, so that each external card can decode it. Your S-100 Bus Adapter Card does produce these status signals even though most external cards do not actually require them.

Table III at the conclusion of this section describes the S-100 signals emulated by the MTS S-100 Adapter. You will notice on the 'comments' heading that some signals are delayed. These signals are actually double-buffered. They are buffered once before they leave the MTS board, via the ribbon cable, and again are buffered on the S-100 Bus Adapter. This is to minimize the length of any single-driven signal path to eliminate noise problems. While the double buffering results in signal delays of 20 - 40 nano-seconds compared to the standard S-100 definition, this minimal delay should present no timing problems.

A few signals are not emulated by the Adapter. These were unique to the original Altair design. In particular, they were used with front panel switch operation of the Altair computer. Since most S-100 systems today are operated through a CRT or printing terminal, they have no front panel switches at all, except for RESET. Therefore, those front panel related signals have not been emulated.

In addition to creating the control signals specified in the S-100

bus definition and splitting the data bus into separate in/out buses, the Adapter Card performs several other necessary functions. First, the Adapter Card has on it eight switches which emulate the eight 'sense' switches on the front panel of an Altair 8800B. These are attached to an input port with device address FF. Some programs available for S-100 systems read these switch settings to learn the configuration of the peripherals (i.e. whether a CRT, printer, etc. is available for I/O). If you want to use the switches, they are located in position 22 on the Adapter Card. The topmost switch (labeled '1' on the switch itself) is data bit 7 (the most significant bit) and the bottom switch (labeled '8' on the switch itself) is data bit 0 (the least significant bit). These are marked on the PCB itself as D7 and D0 respectively. Moving a switch to the 'on' position grounds the associated bit and creates a logic 0. Switching it to 'off' (to the left) creates a logic 1. Thus, to create a Hexadecimal 83, the switch positions would be set as in Figure D-2.

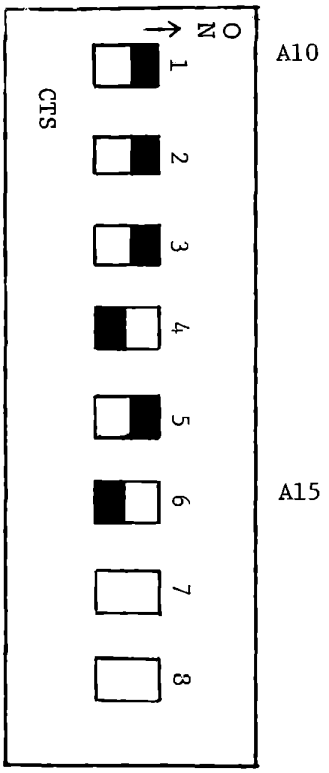


Sense Switch Positions to Create
Hexadecimal 83 on Input Port FF

Figure D-2

The Adapter Card also has on it a circuit to cause the microprocessor to jump to a selected address upon RESET. This address is selectable by setting six switches located in position 31 on the Adapter Card. You can specify a starting address at the first location of any 1K page, i.e. memory locations 0, 1024, 2048, 3072, etc. (decimal), or in hexadecimal, 000, 0400, 0800, 0C00, 1000, etc.

The jump start address is selected using switches 1 through 6. Switches 7 and 8 are unused. Switch 1 corresponds to address bit 10, switch 2 corresponds to address bit 11, etc. (Address bits 0-9 are always set to 0). To specify an address bit as 0, move the switch to on. To specify an address as 1, move the switch to off. As an example, for a jump start address of A000, the binary equivalent is 1010 0000 0000 0000. Thus the switch settings should be 101000 as shown in Figure D-3.



Switch Settings

for

Jump Start Address A000

Figure D-3

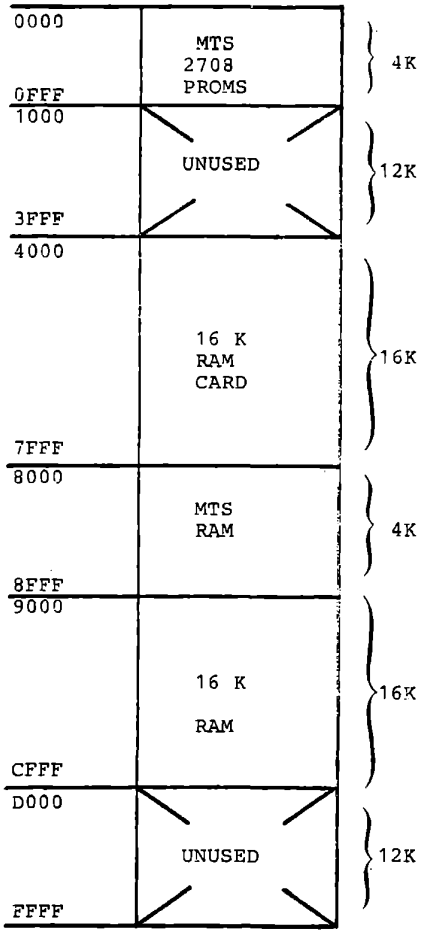
HOW TO IMPLEMENT AN EXPANDED SYSTEM

USING THE S-100 BUS ADAPTER

The first step in implementing an expanded system is to plan the memory and I/O device addresses. We assume there are two general categories of expanded systems: those that incorporate disks and use a disk operating system as the fundamental software for interfacing with the user, and those that do not have a disk and rely on the Microcomputer Training System's Monitor program.

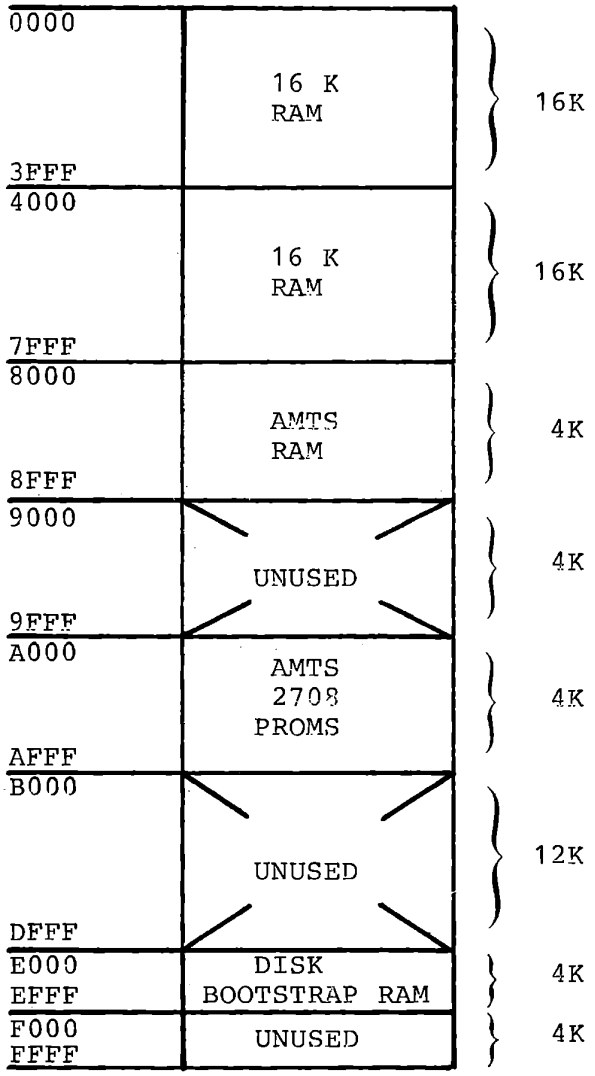
Most monitor-based expanded systems simply involve additional RAM memory. In these cases the memory on the training system can be left in the original addresses assigned to it, and the expansion memory assigned to other unused locations as illustrated in Figure D-4.

Most disk-based systems, however, require RAM memory at address 0. Therefore, the MTS PROMs must be relocated, and in some cases, the MTS RAM should be relocated as well. Instructions for relocating these memories are given in STEP 10 below. Note that if you move either the MTS PROM or RAM, the monitor will no longer function since it relies on being located in address 0 and having the display RAM at locaton 83F8-83FF. A typical arrangement of memory for a disk-based system is shown in Figure D-5.



Typical Expanded Memory Configuration with a total of 36K RAM plus 4K PROM. To be used with the MTS MONITOR in PROM at location 0

Figure D-4



A typical disk operating system memory organization containing 36K RAM plus 4K PROM on MTS board. The bootstrap PROM's for disk operating system begin at E000 and are physically located on the disk controller card itself.

Figure D-5

After designing the memory space, you can start installing the hardware. We suggest the following steps:

STEP 1

Do not yet change the memory addresses on the MTS. Leave the monitor in location 0 and the RAM at 8000. Remove all cards from the S-100 chassis. Turn on the S-100 power and check the voltages on bus lines 1, 2, 51 and 52, relative to line 50 (ground). They should be within a few volts of +8, +18, -8 and -18V respectively (Remember, this is an unregulated supply). Turn the S-100 power off. Turn the MTS power off. BEFORE CONTINUING, HEED THE FOLLOWING WARNING:

> CAUTION: NEVER REMOVE OR INSERT CHIPS, S-100 CARDS, OR THE RIBBON CABLES WHILE THE POWER IS ON. FIRST TURN POWER OFF, WAIT 5 - 10 SECONDS FOR THE CAPACITORS TO DISCHARGE, THEN REMOVE OR INSERT THE DEVICE.

STEP 2

Set the 'Jump Start' address to 0000 by switching all the A - A address select switches (in position 31 of the Adapter Card) to 'ON', which represents logic 0.

STEP 3

Plug the ribbon connector into socket J2 on the MTS. This connector is labeled 'S-100'. Both this connector and the connector labeled 'ITS' are keyed to prevent accidentally plugging the cable into the wrong connector, or plugging it in backwards.

Insert the Adapter Card into the first (front) slot in your S-100 mainframe. Turn on power for both the S-100 and MTS. Verify that the monitor functions normally. Try entering and executing the following program to verify that the MTS can read the sense switches. If you have set the sense switches to '55', you should see a pattern of in the leftmost display on the MTS.

8000	D3		IN FF
8001	FF		
8002	32		STA 83F8
8003	F8		
8004	83		
8005	C3		JMP 8000
8006	00		
8007	80		

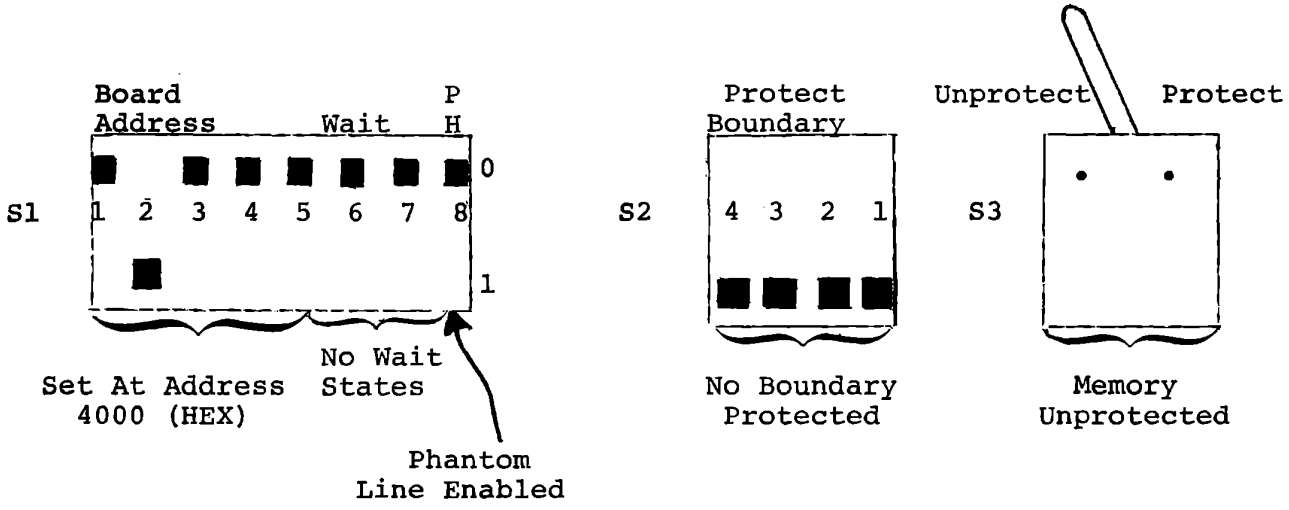
Next, try pressing the RESET switch on the front of the S-100. It should cause the monitor to RESET just as if you had pressed the RST switch on the MTS keyboard.

STEP 4

Set the address on an Expansion RAM card to 4000 by following the directions supplied by the manufacturer of the card. Also, set the control switch on the RAM card to enable it to recognize the PHANTOM signal (Not all cards have this switch). Finally, set the control switch(es) to allow you to write into the RAM, i.e., set the WRITE

PROTECT switch to the 'unprotect' position. Again, some cards don't have this switch either.

An example of setting up a typical RAM card is shown in Figure D-6 for the 16K Static RAM manufactured by Problem Solvers, Inc. Referring to the manufacturer's documentation, the following switch settings would activate the RAM at address 4000.



Switch positions to enable the PROBLEM SOLVER 16K RAM at address 4000

NOTE: Refer to manufacturer's documentation for details

Figure D-6

STEP 5

Turn off all power (both MTS and S-100). Insert the RAM card into any open slots; however, in most S-100 mainframes, it is a good idea to leave a vacant slot on each side of a RAM card for better cooling.

STEP 6

Turn on power for both the S-100 and the MTS. Using the monitor, store data in the S-100 RAM at address 4000, 5000, 6000 and 7000, and then check that the data is there. First, store 55, then repeat with AA. Turn off the MTS and S-100 system power. Remove the RAM card.

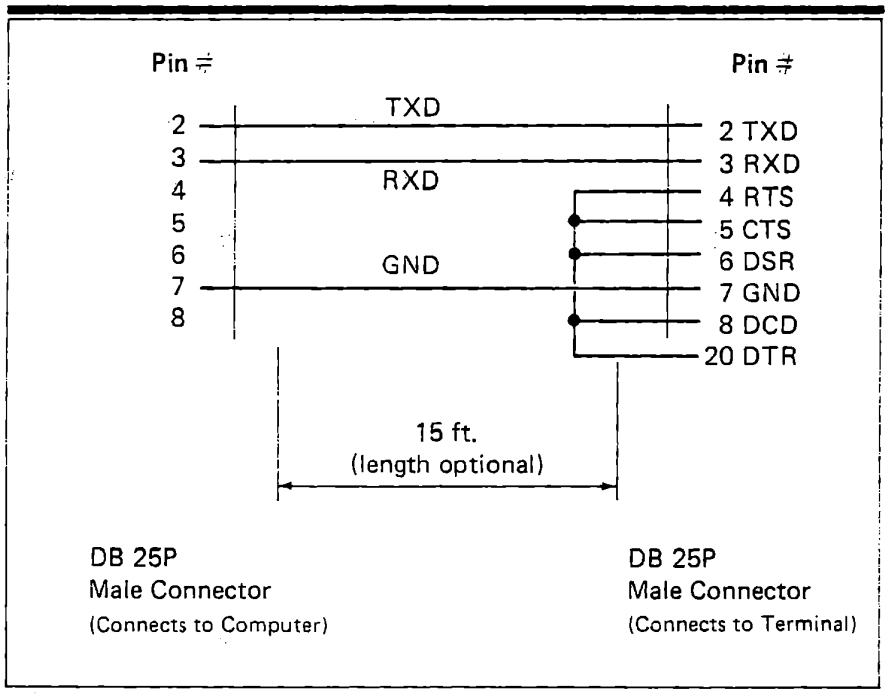
STEP 7

Repeat steps 4, 5 and 6 for each RAM card you have purchased to verify that you can store and retrieve data in it.

STEP 8

If you have purchased an RS232C serial interface card such as the TU-ART, read the manufacturer's directions carefully. Set up the switches on the card as directed. Turn off all power. Insert the TU-ART card in the S-100 frame and connect the cable to the 25-pin 'D' shaped connector, commonly used for RS232C cables (Figure D-7a). The TU-ART to 'D' connector cable is available from Cromemco (the TU-ART manufacturer). Now write a simple program to output characters to the terminal. An example program is shown in Figure D-7b.

TERMINAL TO TU-ART CABLE



This is a diagram of the cable required to connect a serial RS-232 I/O device (such as a CRT terminal) from the DB 25-S socket of the TU-ART cable (model TRT-CBL) to the DB 25-S connector of the RS-232 device.

The jumper connection between pins 4, 5, 6, 8 and 20 may not be required since some terminals have internal pullups on these lines.

Terminal to TU-ART Cable

Figure D-7a

```
ASM80 :F1:TUART.SRC DEBUG SYMBOLS PAGEWIDTH(100)

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0          TUART    PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1 ; SAMPLE PROGRAM TO UTILIZE CROMENCO TU-ART SERIAL/PARALLEL S100 CARD
                      2 ; THIS PROGRAM WILL CONFIGURE A SERIAL OUPUT PORT DEVICE A FOR
                      3 ; TRANSMITTING A SEQUENCE OF "A"S ONTO THE RS232C DEVICE.
                      4 ;
                      5 NAME TUART
  8000                6          ORG 8000H
  8000 3E84           7 TUART:   MVI     A,10000100B     ;1 STOP BIT, 300 BAUD
  8002 D310           8          OUT     10H             ;DEVICE A BAUD RATE REGISTER
  8004 3E20           9          MVI     A,00100000B     ;ALLOW ONLY DEVICE A, TBE
  8006 D313          10          OUT     13H             ;TO INTERRUPT
  8008 3E41          11          MVI     A,'A'           ;CHARACTER "A"
  800A 3E08          12          MVI     A,08H           ;ENABLE INTA
  800C D312          13          OUT     12H             ;COMMAND REGISTER
  800E 3E41          14          MVI     A,'A'           ;CHARACTER "A"
  8010 D311          15          OUT     11H             ;TRANSMITTER DATA REGISTER
  8012 FB            16          EI
  8013 C31380        17 SELF:    JMP     SELF            ;WAIT FOR INTERRUPT
  0028               18          ORG     028H
  0028 C32882        19 INT5:    JMP     IRST5           ;RST5 VECTOR
  8228               20          ORG     8228H
  8228 3E41          21 IRST5:   MVI     A,'A'           ;OUTPUT A 'A'
  822A D311          22          OUT     11H             ;DEVICE A; SERIAL TRANS DATA REGISTER
  822C FB            23          EI
  822D C9            24          RET
                     25 ;
  8000               26          END     TUART

PUBLIC SYMBOLS

EXTERNAL SYMBOLS

USER SYMBOLS
INT5   A 0028    IRST5  A 8228    SELF   A 8013    TUART  A 8000

ASSEMBLY COMPLETE,  NO ERRORS
```

Figure 7b: An example program for TU-ART card.

An Example Program for the TU-ART Card

Figure D-7b

STEP 9

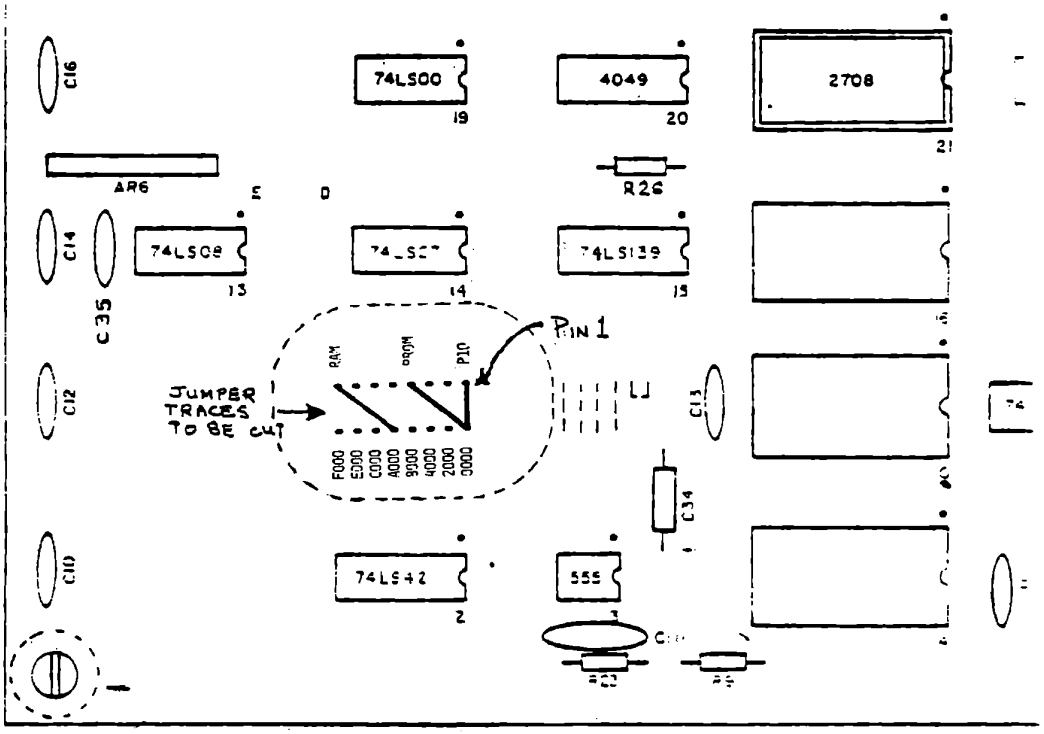
Now you're ready to integrate the system. Set up the S-100 RAM card(s) to the addresses you initially designed. Turn off all power. Insert the cards. If you're not planning to add a disk, the system is ready to use. If you are going to add a disk system, you should now thoroughly study the manufacturer's hardware and software manuals.

STEP 10

Should you want to locate an S-100 RAM in memory address 0000, as required with most disk systems, you will need to relocate the ICS monitor and any other PROMs which currently occupy addresses 0000 - 03FF. To do this you will need to modify certain jumpers on the decoding circuitry of the MTS. If you do, the MTS monitor will not execute properly due to the non-relocatable code.

The MTS has the facility to locate the PROM and the RAM at any 8K page of memory in the 64K address space, as well as to locate the 8255 at any 32 block increment of I/O in the 256 I/O address range from 00 - FF (Refer to Figures D-8 and D-9 and Table II).

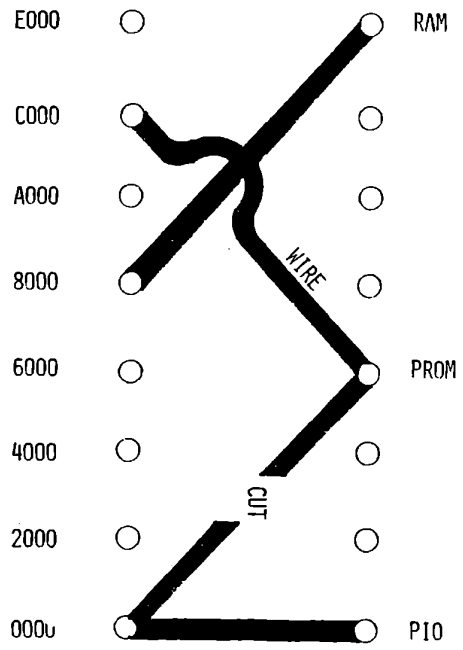
For example, in order to relocate the address of the MTS PROMs to C000 - CFFF, you will need to cut the existing jumper trace between pin 16 marked '0000' on the silk-screen and pin 4 marked 'PROM'. Next, insert a jumper wire from pin 11 marked 'C000' to pin 4 marked 'PROM' (as shown in Figure D-9). You can also perform the same type of operation for moving the RAM or I/O devices, if necessary for your memory system design.



MTS STANDARD MEMORY AND I/O CONFIGURATION

MTS Standard Memory and I/O Configuration

Figure D-8



Address Decoder Jumper Tracess

Figure D-9

MTS ADDRESS DECODER JUMPER SETTINGS

MTS DEVICE	ADDRESS	JUMPERS
EPROM	0000 - 1FFF	"PROM" to pin 16
	2000 - 3FFF	" to pin 15
	4000 - 5FFF	" to pin 14
	6000 - 7FFF	" to pin 13
	8000 - 9FFF	" to pin 12
	A000 - BFFF	" to pin 11
	C000 - DFFF	" to pin 10
	E000 - FFFF	" to pin 9
RAM	0000 - 1FFF	"RAM" to pin 16
	2000 - 3FFF	" to pin 15
	4000 - 5FFF	" to pin 14
	6000 - 7FFF	" to pin 13
	8000 - 9FFF	" to pin 12
	A000 - BFFF	" to pin 11
	C000 - DFFF	" to pin 10
	E000 - FFFF	" to pin 9
8255A	00 - 03	"PIO" to pin 16
	20 - 23	" to pin 15
Control register = X0	40 - 43	" to pin 14
Port A = X1	60 - 63	" to pin 13
Port B = X2	80 - 83	" to pin 12
Port C = X3	A0 - A3	" to pin 11
	C0 - C3	" to pin 10
	E0 - E3	" to pin 9

TABLE II

STEP 11

Set the Jump Start address on the S-100 Adapter Card to the address of the Disk System bootstrap PROM.

STEP 12

Set up the disk controller card according to the manufacturer's directions. Turn off all power. Insert the card, connect the cables, and the hardware is configured. Note, however, that you will probably have to configure the disk operating system software, i.e., to create a version of the disk operating system that uses your addresses for inputs and outputs to CRT terminals and printers. This procedure will be described in the manual for your disk system.

The key elements in bringing up a complete system are:

(1) to test only one new card at a time,

(2) to read the manufacturer's literature prior to using their card, and

(3) to carefully follow the step-by-step procedure given above.

SUMMARY FOR CONFIGURING A SYSTEM

(1) Connect the MTS to the S-100 mainframe using the adapter card and run the ICS monitor.

(2) Draw a memory and I/O map of the proposed system.

(3) Remove any addressing conflicts via the jumper connections n the decoding circuit of the MTS.

(4) Verify that the low address 0000 memory cards can recognize the PHANTOM LINE. If so, set the appropriate switches by referring to the manufacturer's documentation. IF NOT, DO NOT USE.

(5) Configure switch 31 'Jump Address' to vector to the appropriate address upon power-up or a master reset.

(6) Connect J4 of the S-100 Adapter to socket J2 on the MTS.

(7) Insert the 100 pin edge connector J5 into the S-100 system.

(8) Power-up the S-100 system.

(9) Power-up the MTS.

IMPORTANT CONSIDERATIONS

(1) If the MTS monitor is moved from starting location 0000 to another 8K page, the monitor program will not execute properly.

(2) If RAM locations 8000 - 8FFF are moved and the MTS monitor is not, the MTS monitor will still not function because the display locations from 83F8 to 83FF will follow the 8000 - 8FFF block whenever it is moved; i.e., if 8000 - 8FFF is moved to A000 - AFFF, display locations will now be at locations A3F8 to A3FF. The monitor assumes they will be at 83F8 - 83FF.

(3) To use the MTS S-100 Adapter with any S-100 memory card, the automatic Jump Address Vector switches U31 must be configured. In addition, the lower address 0000 of any S-100 RAM card must be capable of disabling itself should the PHANTOM signal be generated upon a master RESET or power-up.

(4) If any memory or I/O address conflicts occur, the MTS memory will have highest priority and therefore will be selected over S-100 memory with the same address.

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
1	+8v	+8 VOLTS	UNREGULATED VOLTAGE ON BUS, SUPPLIED TO PC BOARDS AND REGULATED TO 5v.	YES	VOLTAGE NOT SUPPLIED
2	+18v	+18 VOLTS	POSITIVE PRE-REGULATED VOLTAGE.	YES	VOLTAGE NOT SUPPLIED
3	XRDY	EXTERNAL READY	EXTERNAL READY INPUT TO CPU BOARD'S READY CIRCUITRY.	YES	DELAYED
4	VI0	VECTORED INTERRUPT LINE #0		NO	THESE SIGNALS ARE GENERATED BY I/O BOARDS AND ARE TO BE PROCESSED BY A SEPARATE PRIORITY INTERRRUPT BOARD WHICH THEN GENERATES A SINGLE INTERRUPT REQUEST (ON LINE 73) TO THE ADAPTER CARD
5	VI1	VECTORED INTERRUPT LINE #1		NO	
6	VI2	VECTORED INTERRUPT LINE #2		NO	
7	VI3	VECTORED INTERRUPT LINE #3		NO	
8	VI4	VECTORED INTERRUPT LINE #4		NO	
9	VI5	VECTORED INTERRUPT LINE #5		NO	
10	VI6	VECTORED INTERRUPT LINE #6		NO	

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
11	VI7	VECTORED INTERRUPT LINE #7		NO	"
12	*XRDY2	EXTERNAL READY #2	A SECOND EXTERNAL READY LINE SIMILAR TO XRDY	YES	DELAYED
13 TO 17	TO BE DEFINED				
18	$\overline{\text{STAT DSB}}$	$\overline{\text{STATUS DISABLE}}$	ALLOWS THE BUFFERS FOR THE 8 STATUS LINES TO BE TRI-STATED	YES	-
19	$\overline{\text{C/C DSB}}$	$\overline{\text{COMMAND/CONTROL}}$	ALLOWS THE BUFFERS FOR THE 6 OUTPUT COMMAND/CONTROL LINES TO BE TRI-STATED	YES	-
20	UNPROT	UNPROTECT	INPUT TO THE MEMORY PROTECT FLIP-FLOP ON A GIVEN MEMORY BOARD	NO	MOST MEMORY BOARDS HAVE SOFTWARE CONTROLLABLE FLIP-FLOPS
21	SS	SINGLE STEP	INDICATES THAT THE MACHINE IS IN THE PROCESS OF PERFORMING A SINGLE STEP (i.e. THAT SS FLIP-FLOP ON D/C IS SET)	NO	NOT USED
22	$\overline{\text{ADD DSB}}$	$\overline{\text{ADDRESS DISABLE}}$	ALLOWS THE BUFFERS FOR THE 16 ADDRESS LINES TO BE TRI-STATED	YES	-
23	$\overline{\text{DO DBS}}$	$\overline{\text{DATA OUT DISABLE}}$	ALLOWS THE BUFFERS FOR THE 8 DATA OUTPUT LINES TO BE TRI-STATED	YES	-

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
24	Ø2	PHASE 2 CLOCK		YES	DELAYED
25	Ø1	PHASE 1 CLOCK		YES	DELAYED
26	PHLDA	HOLD ACKNOWLEDGE	PROCESSOR COMMAND/CONTROL OUTPUT SIGNAL THAT APPEARS IN RESPONSE TO THE HOLD SIGNAL; INDICATES THAT THE DATA AND ADDRESS BUS WILL GO TO THE HIGH IMPEDANCE STATE AND PROCESSOR WILL ENTER HOLD STATE AFTER COMPLETION OF THE CURRENT MACHINE CYCLE.	YES	DELAYED
27	PWAIT	WAIT	PROCESSOR COMMAND/CONTROL SIGNAL THAT APPEARS IN RESPONSE TO THE READY SIGNAL GOING LOW; INDICATES PROCESSOR WILL ENTER A SERIES OF .5 MICROSECOND WAIT STATES UNTIL READY AGAIN GOES HIGH.	YES	DELAYED
28	PINTE	INTERRUPT ENABLE	PROCESSOR COMMAND/CONTROL OUTPUT SIGNAL; INDICATES INTERRUPTS ARE ENABLED, AS DETERMINED BY THE CONTENTS OF THE CPU INTERNAL INTERRUPT FLIP-FLOP. WHEN THE FLIP-FLOP IS SET (ENABLE INTERRUPT INSTRUCTION), INTERRUPTS ARE ACCEPTED BY THE CPU; WHEN IT IS RESET (DISABLE INTERRUPT INSTRUCTION), INTERRUPTS ARE INHIBITED.	YES	DELAYED

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
29	A5	ADDRESS LINE #5		YES	DELAYED
30	A4	ADDRESS LINE #4		YES	DELAYED
31	A3	ADDRESS LINE #3		YES	DELAYED
32	A15	ADDRESS LINE #15	(MSB)	YES	DELAYED
33	A12	ADDRESS LINE #12		YES	DELAYED
34	A9	ADDRESS LINE #9		YES	DELAYED
35	DO1	DATA OUT LINE #1		YES	-
36	DO0	DATA OUT LINE #0	(LSB)	YES	-
37	A10	ADDRESS LINE #10		YES	DELAYED
38	DO4	DATA OUT LINE #4		YES	-
39	DO5	DATA OUT LINE #5		YES	-
40	DO6	DATA OUT LINE #6		YES	-
41	DI2	DATA IN LINE #2		YES	-
42	DI3	DATA IN LINE #3		YES	-
43	DI7	DATA IN LINE #7	(MSB)	YES	-

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
44	SM1	MACHINE CYCLE 1	STATUS OUTPUT SIGNAL THAT INDICATES THAT THE PROCESSOR IS IN THE FETCH CYCLE FOR THE FIRST BYTE OF AN INSTRUCTION.	YES	DELAYED
45	SOUT	OUTPUT	STATUS OUTPUT SIGNAL THAT INDICATES THE ADDRESS BUS CONTAINS THE ADDRESS OF AN OUTPUT DEVICE AND THE DATA BUS WILL CONTAIN THE OUTPUT DATA WHEN PWR IS ACTIVE.	YES	DELAYED
46	SINP	INPUT	STATUS OUTPUT SIGNAL THAT INDICATES THE ADDRESS BUS CONTAINS THE ADDRESS OF AN INPUT DEVICE AND THE INPUT DATA SHOULD BE PLACED ON THE DATA BUS WHEN PDBIN IS ACTIVE.	YES	DELAYED
47	SMEMR	MEMORY READ	STATUS OUTPUT SIGNAL THAT INDICATES THE DATA BUS WILL BE USED TO READ MEMORY DATA.	YES	DELAYED
48	SHLTA	HALT	STATUS OUTPUT SIGNAL THAT ACKNOWLEDGES A HALT INSTRUCTION.	YES	DELAYED
49	$\overline{\text{CLOCK}}$	$\overline{\text{CLOCK}}$	INVERTED OUTPUT OF THE Ø2 CLOCK	YES	DELAYED

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
50	GND	GROUND		YES	
51	+8v	+8 VOLTS	UNREGULATED INPUT TO 5 VOLT REGULATORS	YES	VOLTAGE NOT SUPPLIED BY AMTS
52	-18v	-18 VOLTS	NEGATIVE PRE-REGULATED VOLTAGE	YES	VOLTAGE NOT SUPPLIED BY AMTS
53	$\overline{\text{SSWI}}$	SENSE SWITCH INPUT	INDICATES THAT AN INPUT DATA TRANSFER FROM THE SENSE SWITCHES IS TO TAKE PLACE. THIS SIGNAL IS USED BY THE DISPLAY/CONTROL LOGIC TO: A) ENABLE SENSE SWITCH DRIVERS B) ENABLE THE DISPLAY/ CONTROL BOARD DRIVERS DATA INPUT (FDIØ-FDI7 C) DISABLE THE CPU BOARD DATA INPUT DRIVERS (DIØ-DI7)	NO	FRONT PANEL RELATED FUNCTION
54	$\overline{\text{EXT CLR}}$	EXTERNAL CLEAR	CLEAR SIGNAL FOR I/O DEVICES (FRONT PANEL SWITCH CLOSURE TO GROUND)	NO	MOST S100 CARDS NOW USE MASTER SYSTEM RESET

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
55	RTC	REAL TIME CLOCK	60Hz SIGNAL USED AS TIMING	NO	NOT NEEDED
56	$\overline{\text{STSTB}}$	$\overline{\text{STATUS STROBE}}$	OUTPUT STROBE SIGNAL SUPPLIED BY THE 8224 CLOCK GENERATOR. PRIMARY PURPOSE IS TO STROBE THE 8212 STATUS LATCH SO THAT STATUS IS SET UP AS SOON IN THE MACHINE CYCLE AS POSSIBLE. THIS SIGNAL IS ALSO USED BY DISPLAY/CONTROL LOGIC.	YES	DELAYED
57	DIG1	DATA INPUT GATE #1	OUTPUT SIGNAL FROM THE DISPLAY/ CONTROL LOGIC THAT DETERMINES WHICH SET OF DATA INPUT DRIVERS HAVE CONTROL OF THE CPU BOARD'S BIDIRECTIONAL DATA BUS. IF DIG1 IS HIGH, THE CPU DRIVERS HAVE CONTROL; IF IT IS LOW THE DISPLAY/CONTROL LOGIC DRIVERS HAVE CONTROL.	YES	
58	FRDY	FRONT PANEL READY	OUTPUT SIGNAL FROM D/C LOGIC THAT ALLOWS THE FRONT PANEL TO CONTROL THE READY LINE TO THE CPU.	YES	DELAYED
59 - 66	TO BE DEFINED		-	-	-
67	$\overline{\text{PHANTOM}}$	$\overline{\text{PHANTOM}}$	SIGNAL FROM ADAPTER CARD TO DISABLE MEMORIES AFTER RESET TO ALLOW JUMP START TO ADDRESS SPECIFIED ON ADAPTER CARD.	YES	-

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
68	MWRITE	MEMORY WRITE	INDICATES THAT THE DATA PRESENT ON THE DATA OUT BUS IS TO BE WRITTEN INTO THE MEMORY LOCATION CURRENTLY ON THE ADDRESS BUS.	YES	DELAYED
69	$\overline{PS}$	PROTECT STATUS	INDICATES THE STATUS OF THE MEMORY PROTECT FLIP-FLOP ON THE MEMORY BOARD.	NO	NO FRONT PANEL LAMPS ON WHICH TO INDICATE STATUS
70	PROT	PROTECT	INPUT TO THE MEMORY PROTECT FLIP-FLOP ON THE MEMORY BOARD CURRENTLY ADDRESSED.	NO	MEMORY PROTECT USUALLY IS SWITCH SELECTABLE ON THE RAM CARD ITSELF
71	RUN	RUN	INDICATES THAT THE STOP/RUN FLIP-FLOP IS RESET; i.e. MACHINE IS IN RUN MODE.	NO	NOT USED
72	PRDY	PROCESSOR READY	MEMORY AND I/O INPUT TO THE CPU BOARD WAIT CIRCUITRY	YES	DELAYED
73	PINT	INTERRUPT REQUEST	THE PROCESSOR RECOGNIZES AN INTERRUPT REQUEST ON THIS LINE AT THE END OF THE CURRENT INSTRUCTION OR WHILE HALTED. IF THE PROCESSOR IS IN THE HOLD STATE OR THE INTERRUPT ENABLE FLIP-FLOP IS RESET, IT WILL NOT HONOR THE REQUEST.	YES	DELAYED

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
74	$\overline{\text{PHOLD}}$	$\overline{\text{HOLD}}$	PROCESSOR COMMAND/CONTROL INPUT SIGNAL THAT REQUESTS THE PROCESSOR ENTER THE HOLD STATE; ALLOWS AN EXTERNAL DEVICE TO GAIN CONTROL OF ADDRESS AND DATA BUSES AS SOON AS THE PROCESSOR HAS COMPLETED ITS USE OF THESE BUSES FOR THE CURRENT MACHINE CYCLE.	YES	DELAYED
75	$\overline{\text{PRESET}}$	$\overline{\text{RESET}}$	PROCESSOR COMMAND/CONTROL INPUT; WHILE ACTIVATED, THE CONTENT OF THE PROGRAM COUNTER IS CLEARED AND THE INSTRUCTION REGISTER IS SET TO 0.	YES	-
76	PSYNC	SYNC	PROCESSOR COMMAND/CONTROL OUTPUT; PROVIDES A SIGNAL TO INDICATE THE BEGINNING OF EACH MACHINE CYCLE.	YES	DELAYED
77	$\overline{\text{PWR}}$	$\overline{\text{WRITE}}$	PROCESSOR COMMAND/CONTROL OUTPUT; USED FOR MEMORY WRITE OR I/O OUTPUT CONTROL. DATA ON THE DATA BUS IS STABLE WHILE THE PWR IS ACTIVE.	YES	DELAYED
78	PDBIN	DATA BUS IN	PROCESSOR COMMAND/CONTROL OUTPUT; INDICATES TO EXTERNAL CIRCUITS THAT THE DATA BUS IS IN THE INPUT MODE.	YES	DELAYED

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
79	A0	ADDRESS LINE #0	(LSB)	YES	DELAYED
80	A1	ADDRESS LINE #1		YES	DELAYED
81	A2	ADDRESS LINE #2		YES	DELAYED
82	A6	ADDRESS LINE #6		YES	DELAYED
83	A7	ADDRESS LINE #7		YES	DELAYED
84	A8	ADDRESS LINE #8		YES	DELAYED
85	A13	ADDRESS LINE #13		YES	DELAYED
86	A14	ADDRESS LINE #14		YES	DELAYED
87	A11	ADDRESS LINE #11		YES	DELAYED
88	DO2	DATA OUT LINE #2		YES	-
89	DO3	DATA OUT LINE #3		YES	-
90	DO7	DATA OUT LINE #7		YES	-
91	DI4	DATA IN LINE #4		YES	-
92	DI5	DATA IN LINE #5		YES	-
93	DI6	DATA IN LINE #6		YES	-
94	DI7	DATA IN LINE #1		YES	-

TABLE III

S100 BUS SPECIFICATION AS DEFINED BY MITS FOR THEIR ALTAIR 8800B

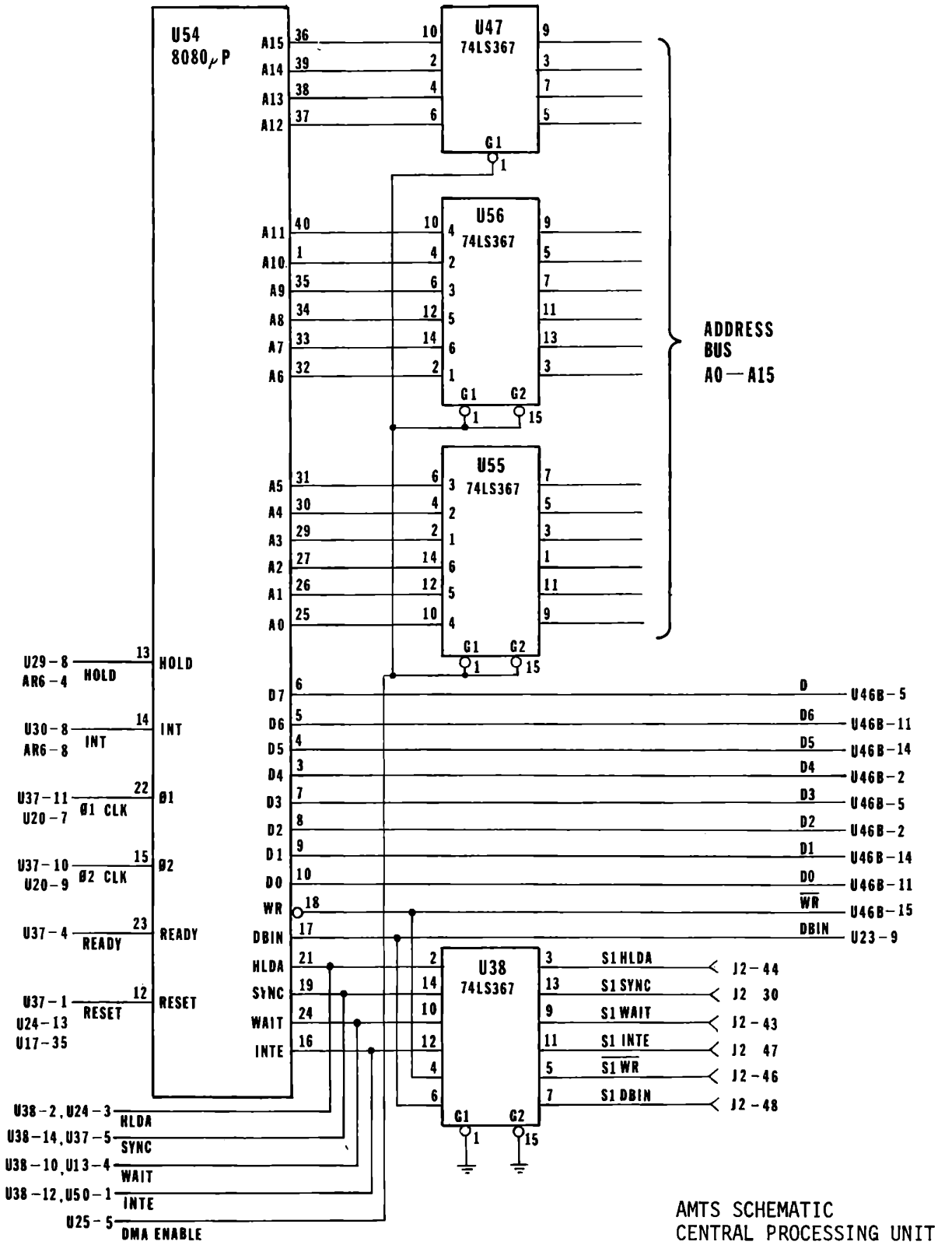
& EMULATED BY THE S100 BUS ADAPTER

PIN NUMBER	SYMBOL	NAME	FUNCTION	AMTS S100 ADAPTER IMPLEMENTATION	COMMENTS
95	DIO	DATA IN LINE #0	(LSB)	YES	
96	SINTA	INTERRUPT ACKNOWL-EDGE	STATUS OUTPUT SIGNAL; ACKNOWLEDGES SIGNAL FOR INTERRUPT REQUEST.	YES	DELAYED
97	$\overline{\text{SWO}}$	WRITE OUT	STATUS OUTPUT SIGNAL; INDICATES THAT THE OPER-ATION IN THE CURRENT MACHINE CYCLE WILL BE A WRITE MEMORY OR OUTPUT FUNCTION.	YES	DELAYED
98	SSTACK	STACK	STATUS OUTPUT SIGNAL INDICATES THAT THE ADDRESS BUS HOLDS THE PUSHDOWN STACK ADDRESS FROM THE STACK POINTER.	YES	DELAYED
99	$\overline{\text{POC}}$	POWER-ON CLEAR		YES	-
100	GND	GROUND		YES	-

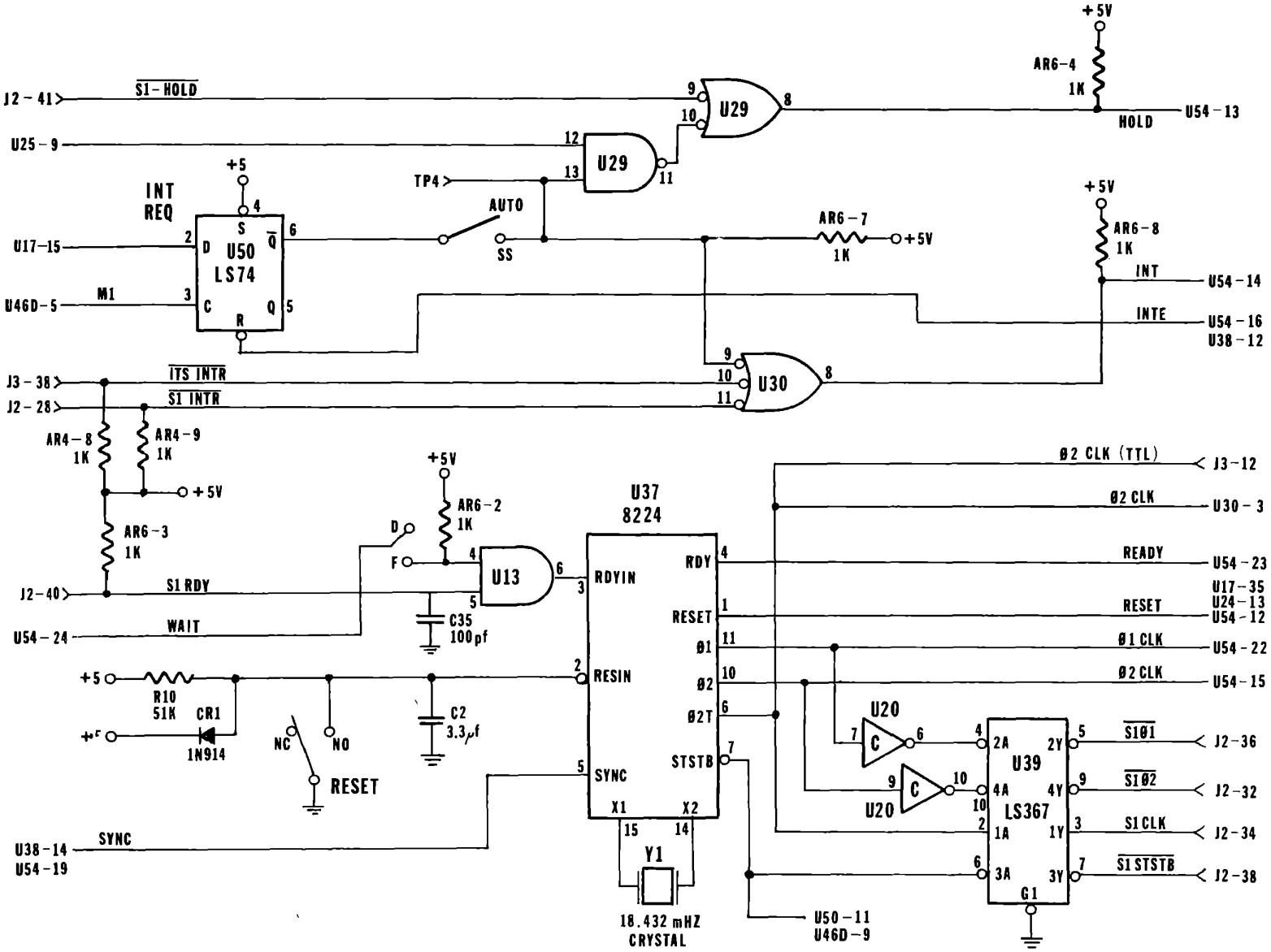
MICROCOMPUTER TRAINING WORKBOOK

APPENDIX E

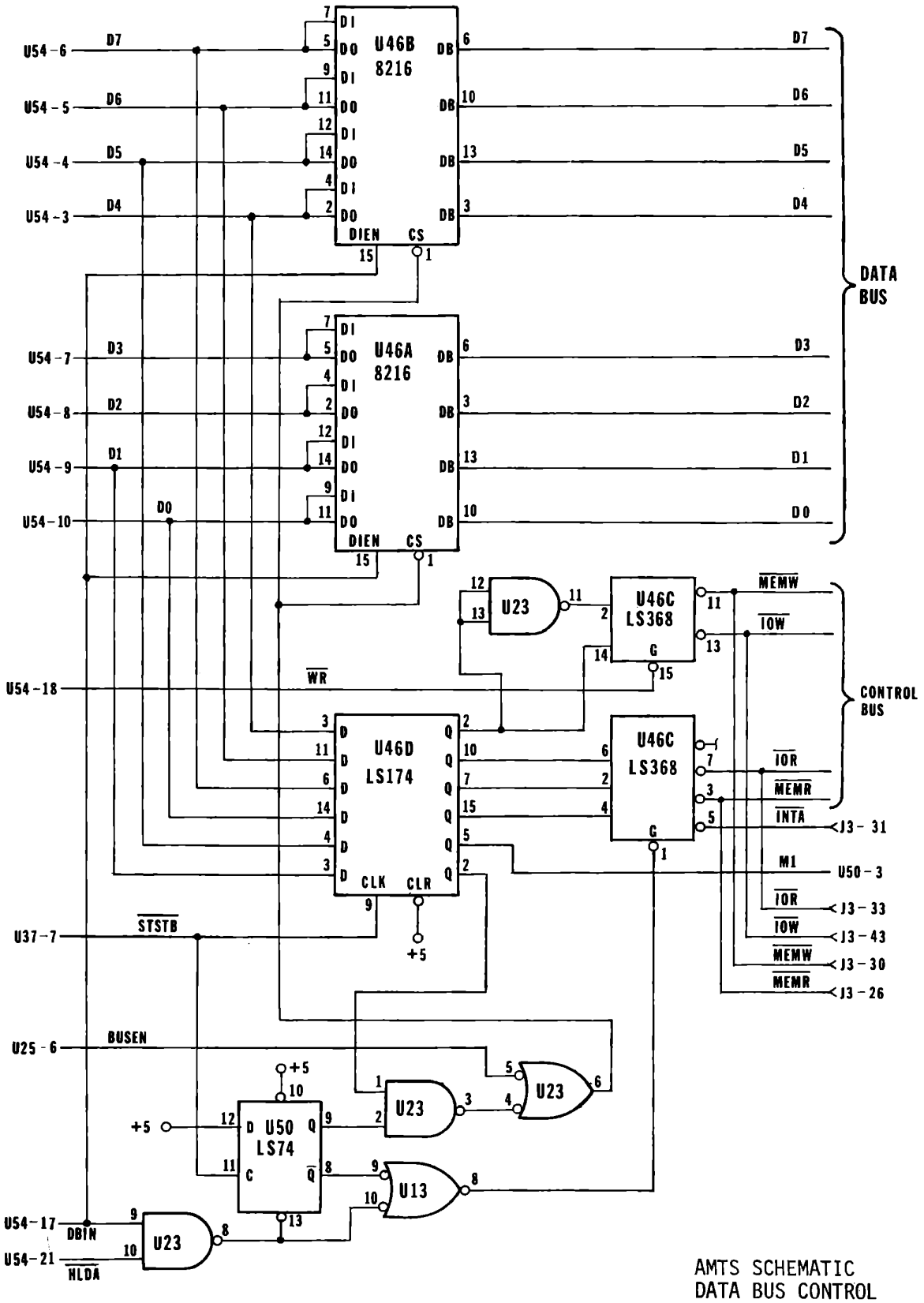
AMTS SCHEMATICS



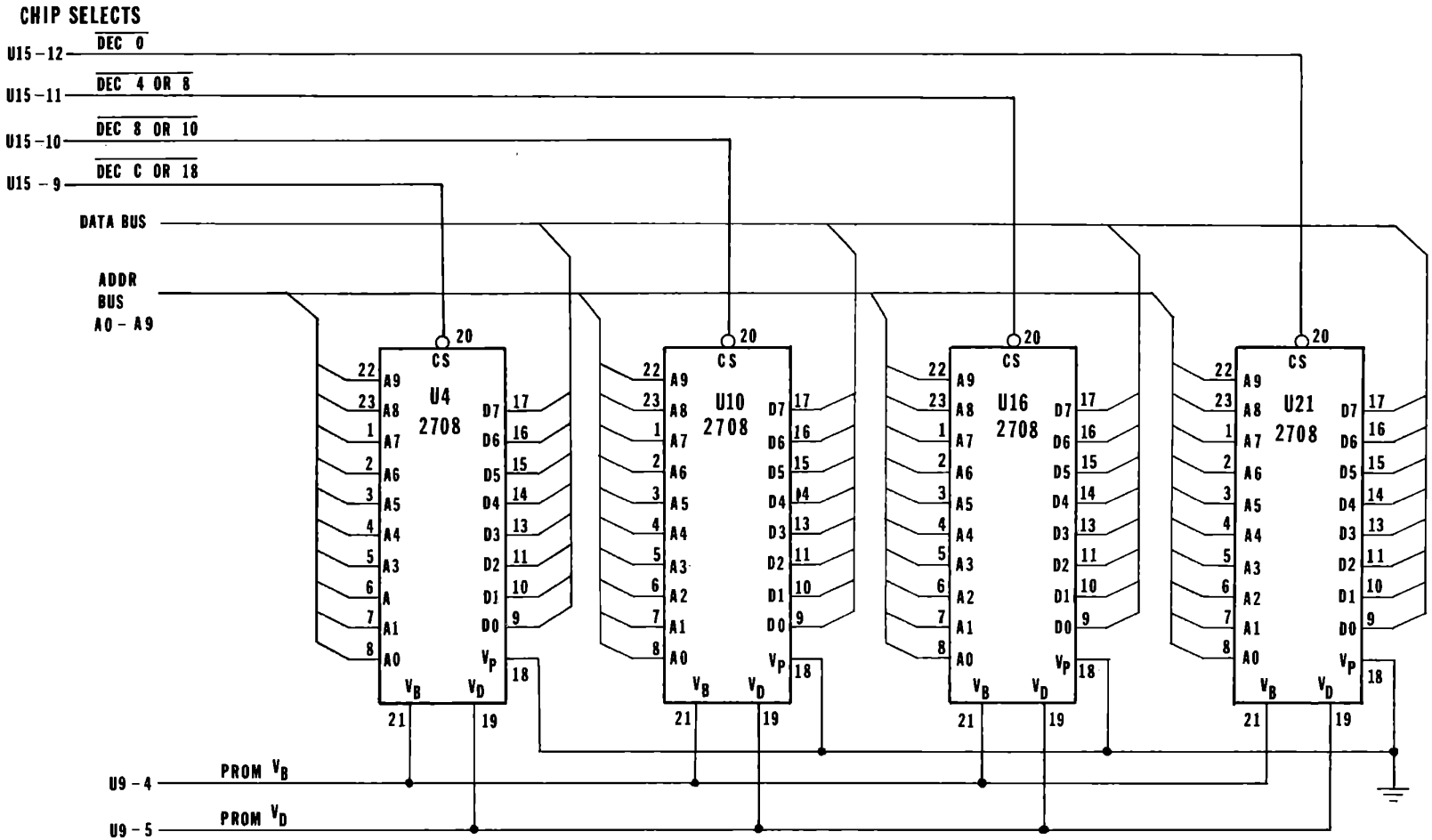
AMTS SCHEMATIC
CENTRAL PROCESSING UNIT
PAGE 1 OF 10 05/79/0132



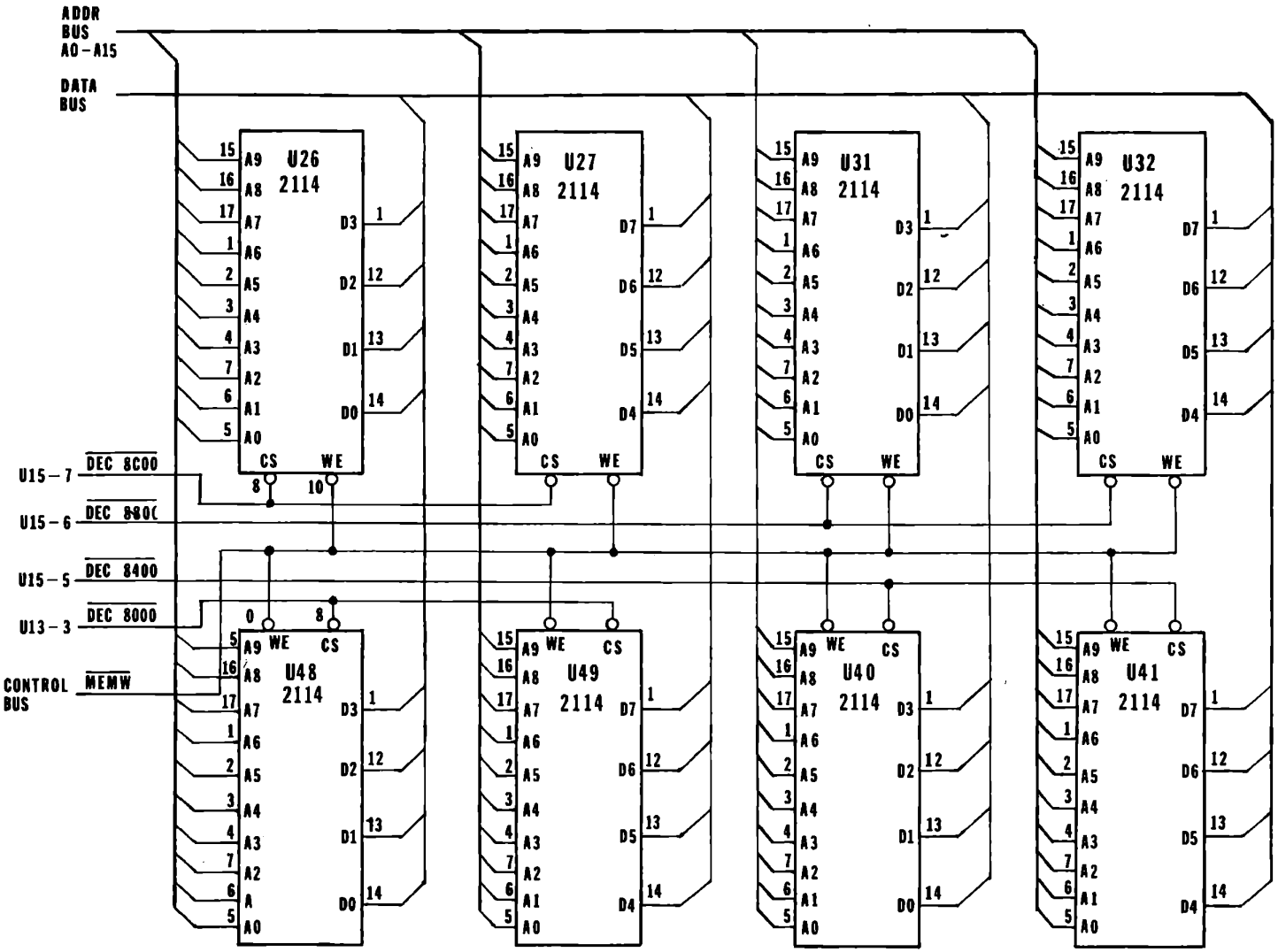
AMTS SCHEMATIC
CLOCK AND INTERRUPT CIRCUIT
PAGE 2 OF 10 05/79/0132



AMTS SCHEMATIC
DATA BUS CONTROL
PAGE 3 OF 10 05/79/0132

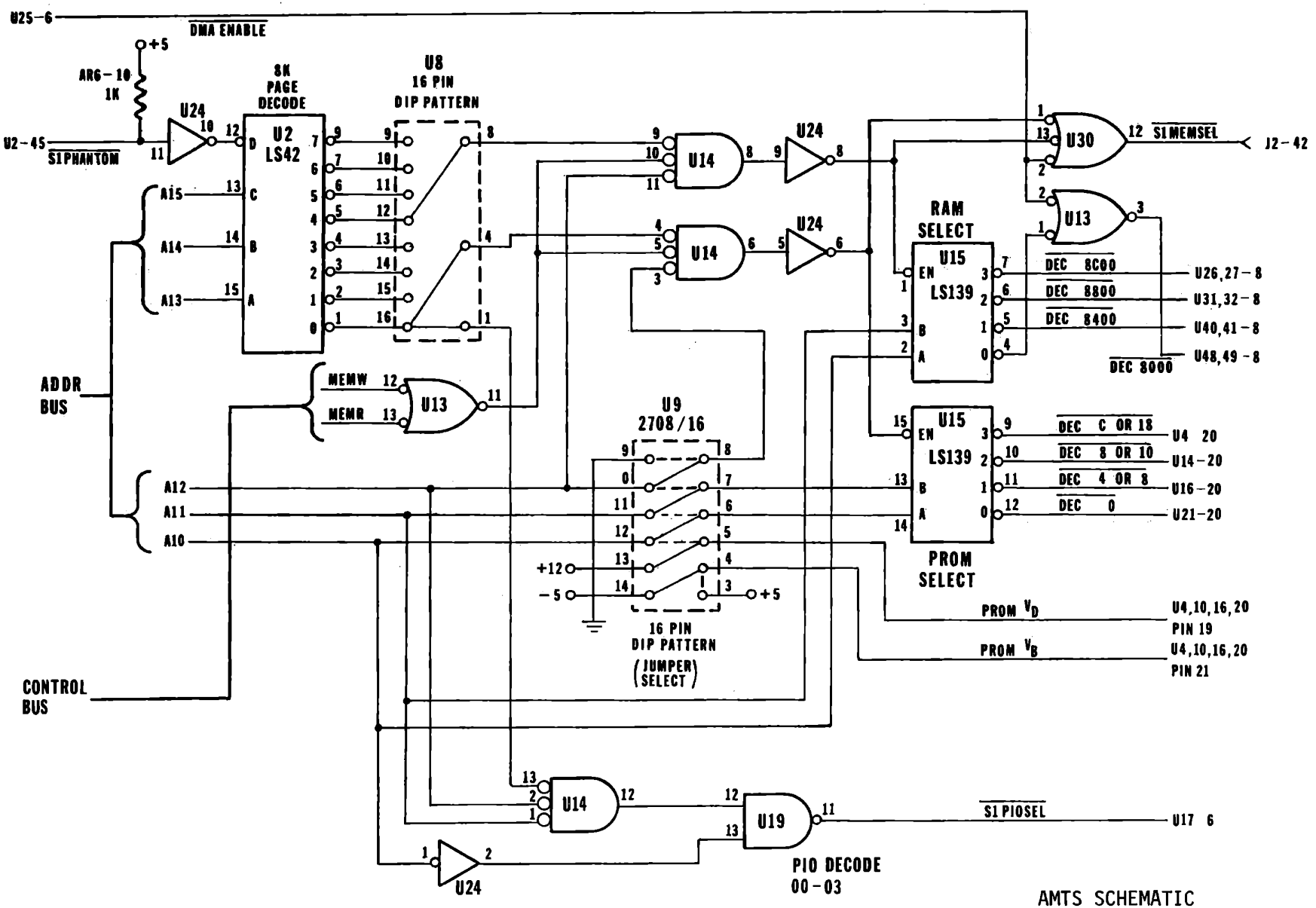


AMTS SCHEMATIC
ROM MEMORY
PAGE 4 OF 10 05/79/0132

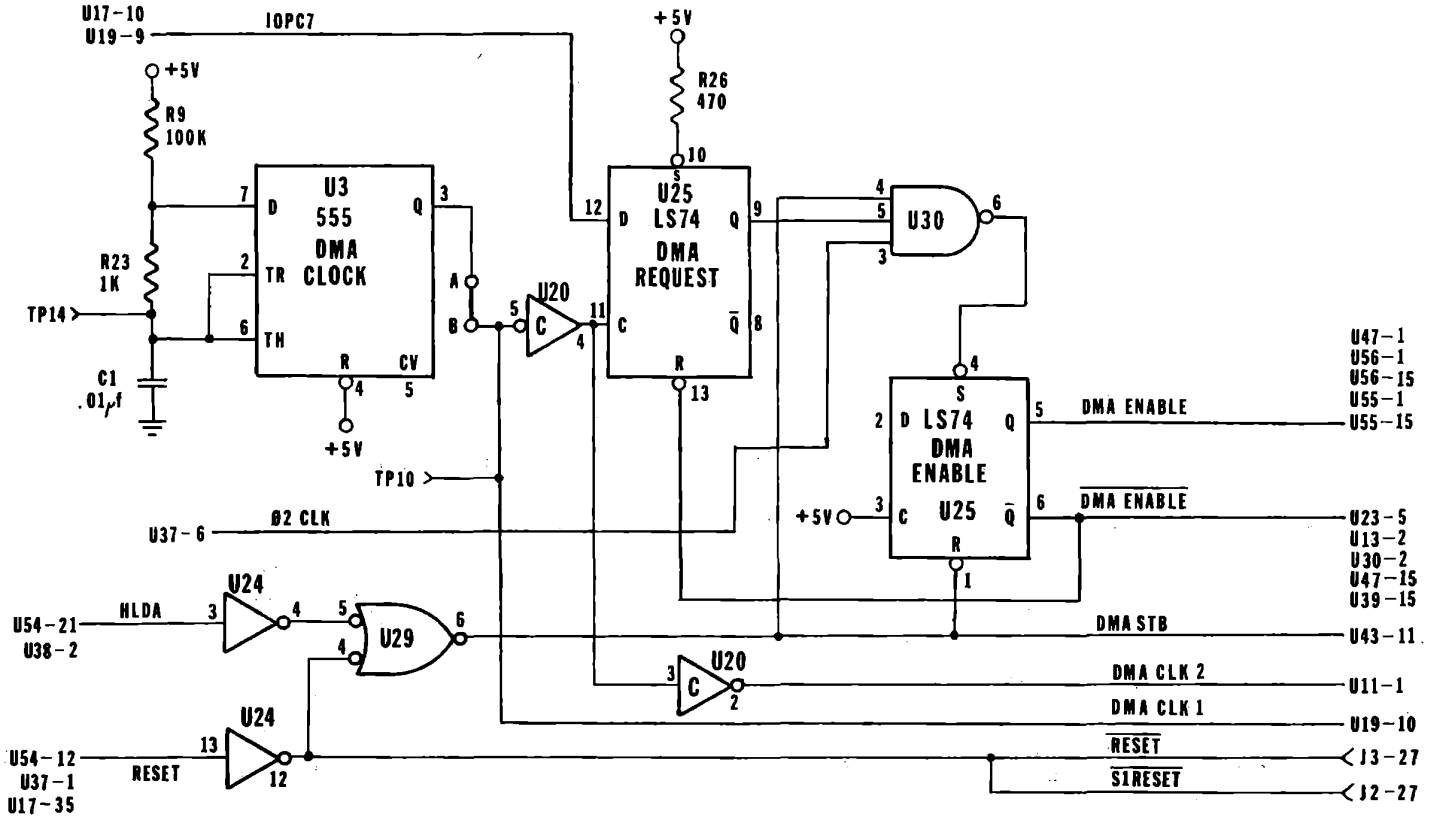


AMTS SCHEMATIC
RAM MEMORY
PAGE 5 OF 10 05/79/0132

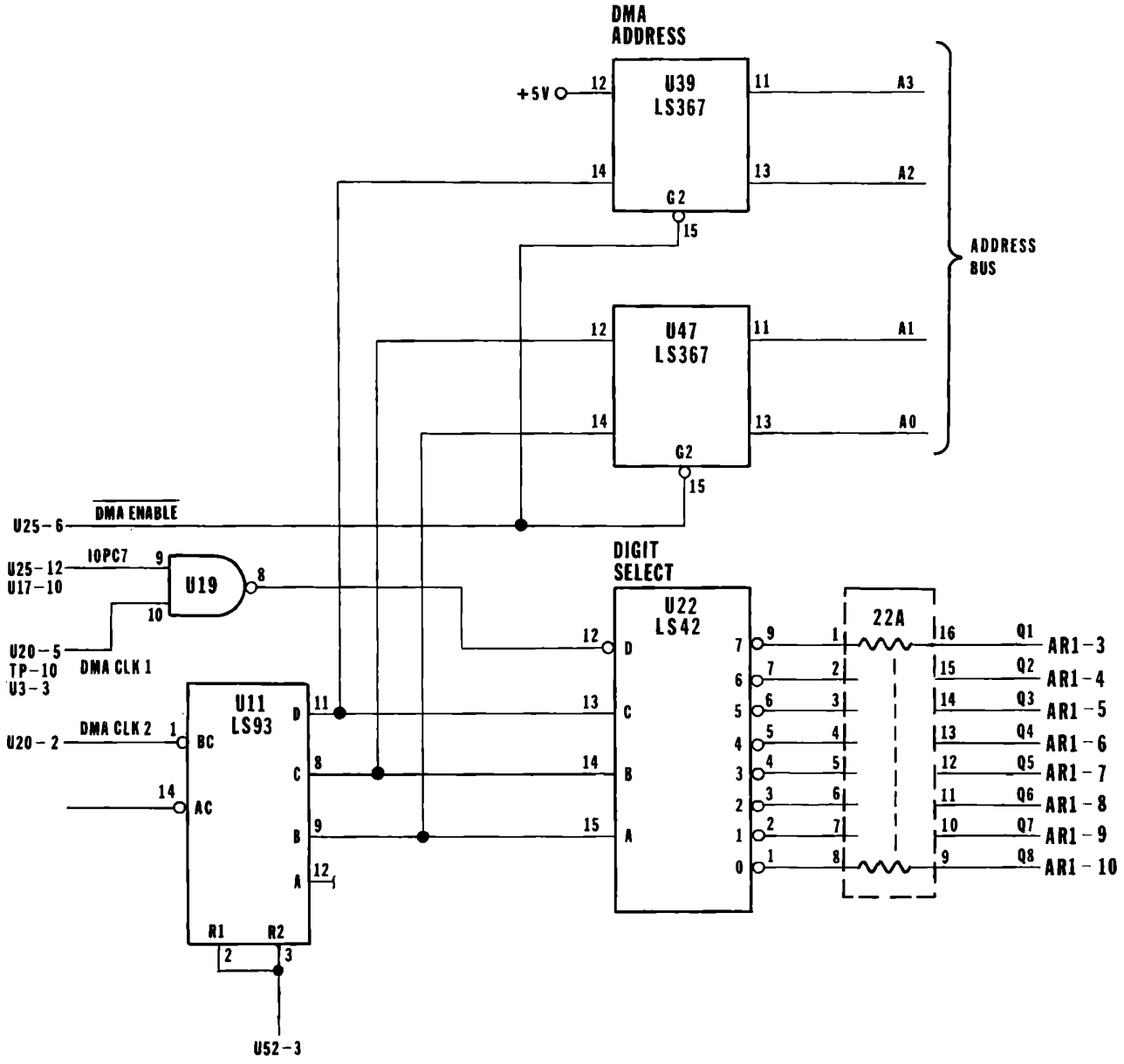
E-5



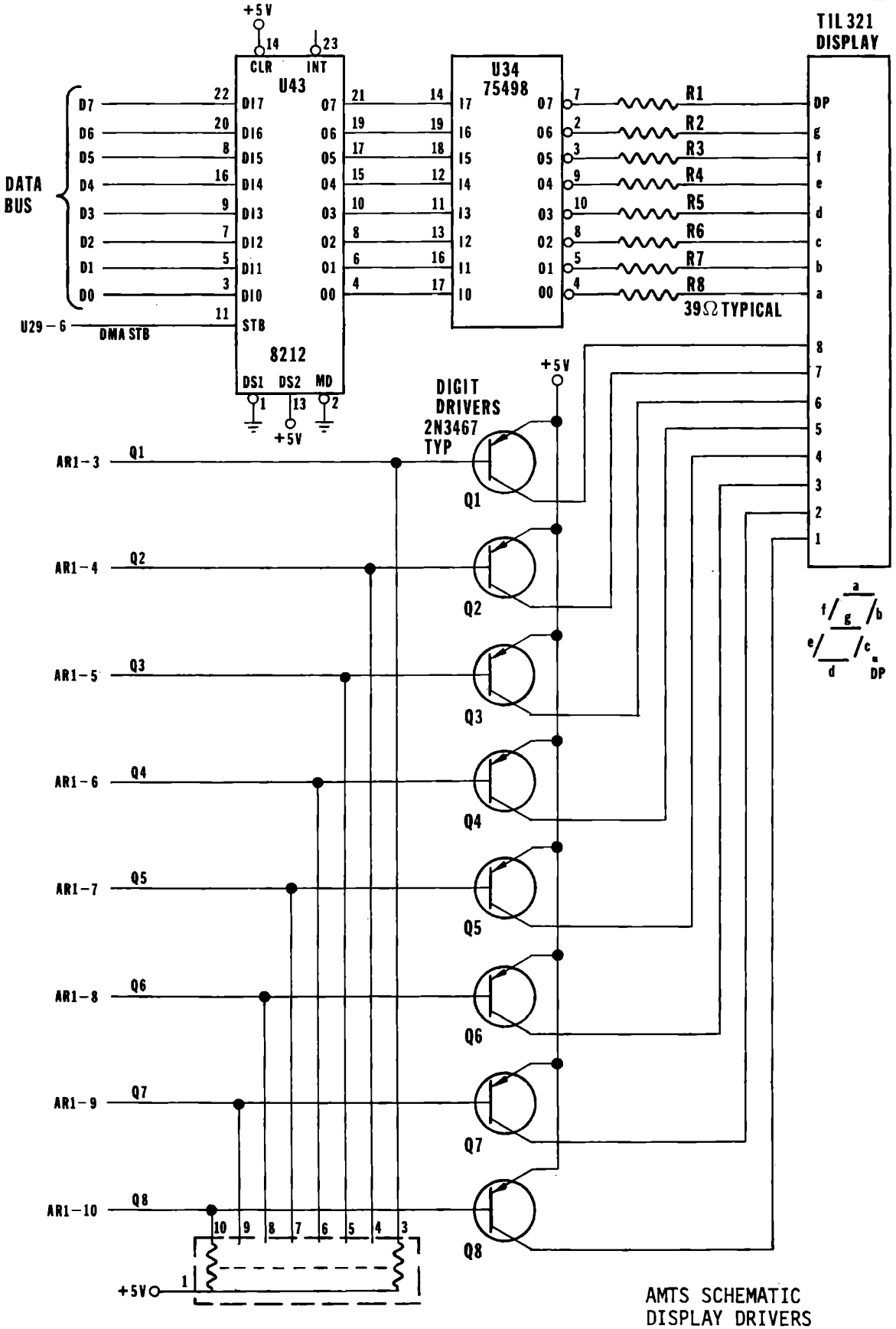
AMTS SCHEMATIC
ADDRESS DECODING
PAGE 6 OF 10 05/79/0132



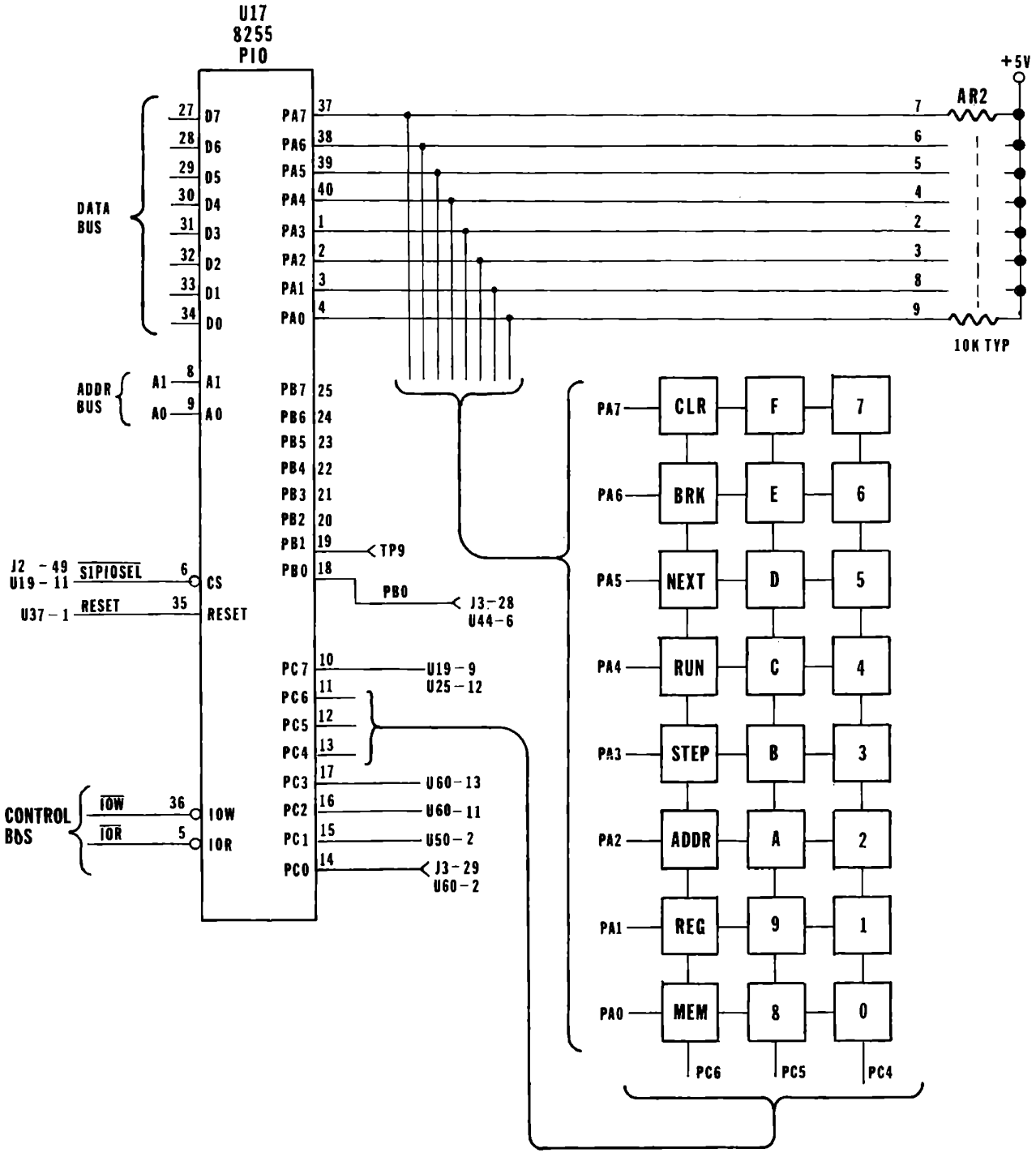
AMTS SCHEMATIC
DMA CLOCK
PAGE 7 OF 10 05/79/0132



AMTS SCHEMATIC
DMA SELECT
PAGE 8 OF 10 05/79/0132



AMTS SCHEMATIC
DISPLAY DRIVERS
PAGE 9 OF 10 05/79/0132



AMTS SCHEMATIC
KEYBOARD CONTROLLER
PAGE 10 OF 10 05/79/0132

MICROCOMPUTER TRAINING WORKBOOK

APPENDIX F

A PRIMER TO DIGITAL LOGIC

WHAT IS THE MOST BASIC LOGIC DEVICE?

Answer: a wire

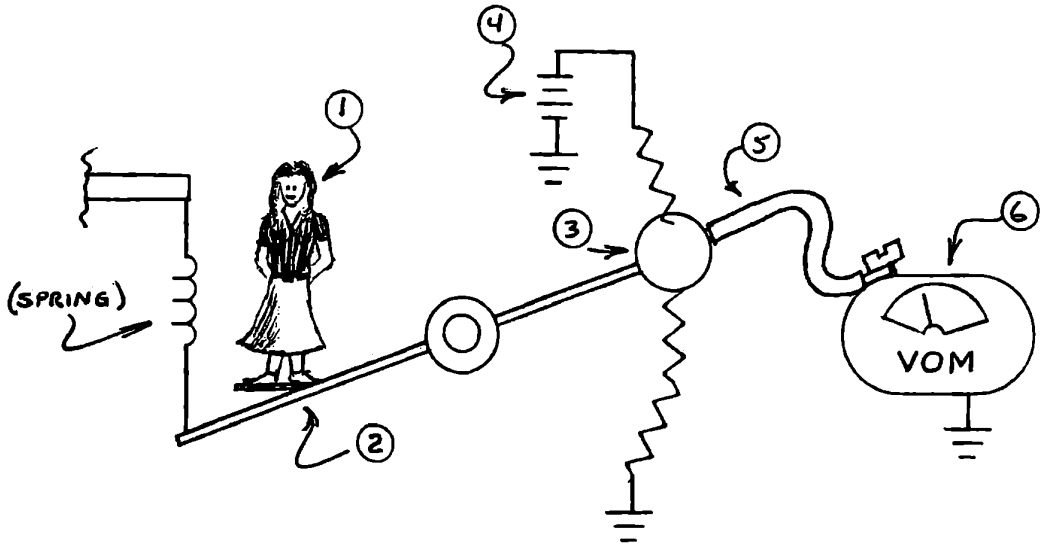
Logical property of a wire:

Any voltage potential that is placed on the "A" input of the wire will soon (very soon) appear at the "B" output. In other words, a wire can be used to transmit voltage states from any node "A" to any node "B".

WHAT INFORMATION MAY BE TRANSMITTED ON A WIRE?

First Answer: A voltage representing a number (an analog signal):

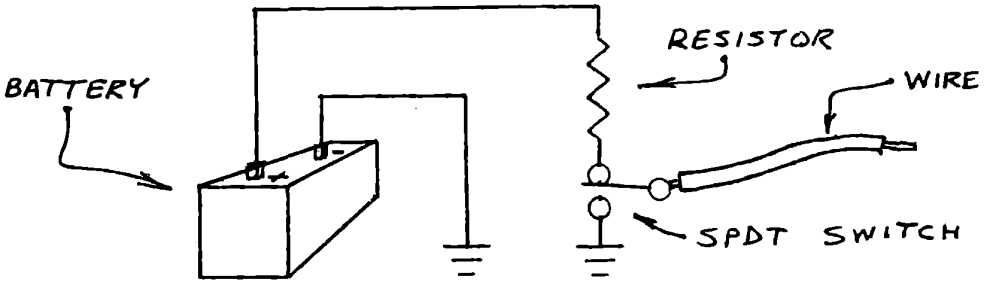
Here's an example,



An electrical reading scale:

1. Person steps on scale
2. Platform sags in response to person standing on scale
3. Causing metal wiper to change position along a resistive material
4. That is connected as a potentiometer in a voltage divider network
5. Whose wiper output is connected via a wire to cause a voltage that is proportional to the weight of the person:
6. To be transmitted to a voltmeter that indicates the weight of the person

Second Answer: A wire can also transmit a voltage representing one of two conditions:



If switch is up, voltage at end of wire will be equal to battery voltage

If switch is down, voltage at end of wire will be zero (or nearly so)

Let's look more at this setup in which the wire must carry a voltage that represents one of two conditions:

For many kinds of electronic devices that are classified as digital devices, the two possible conditions are defined as follows:

1st condition:

Wire is defined to be **ON** when voltage between wire and ground is between +2.5v and +5.25v.

2nd condition:

Wire is defined to be **OFF** when voltage between wire and ground is between - Ø.5v and +Ø.5v.

BUT WHAT ABOUT VOLTAGES THAT ARE NOT WITHIN ABOVE RANGES?

Answer: they are not allowed. Voltages between +Ø.5v and +2.5v are unclear and therefore not to be trusted. Voltages below -Ø.5v or above +5.25 volts are hazardous (may cause smoke and fire!) and are therefore expressly forbidden !

CAN WE SIMPLIFY THE DEFINITION OF "ON" AND "OFF" WHEN TALKING ABOUT VOLTAGES ON WIRES?

Answer: certainly. For sake of discussion we will simply define the **ON** state as +5 volts between the wire and ground.

The **OFF** state we will define as Ø volts between the wire and ground.

Definition: Digital electronics: That branch of electrionics specializing in circuits whose signals are defined as "on" and "off" voltages.

Actually, let's go further and define what 5 volts and Ø volts are really equivalent to:

5 volts = **ON** = True = One = High = "1"

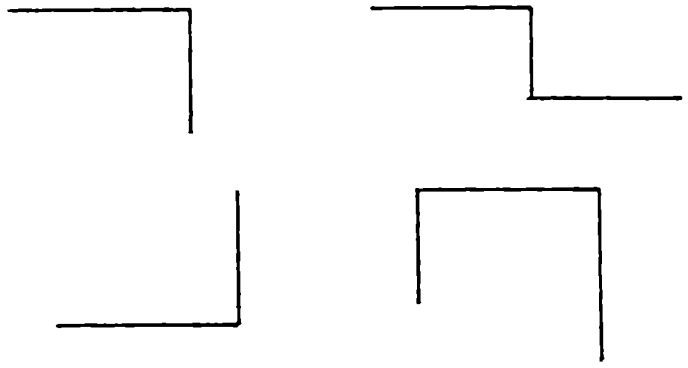
Ø volts = **OFF** = False = Zero = Low = "Ø"

So now, when we talk about the signal on a wire being **ON**, true, one, or high we mean that there is approximately 5 volts on it. And when the signal is **OFF**, false, zero, or low we mean the voltage is zero. (Actually, there is an exception to these assignments but we'll talk about that later).

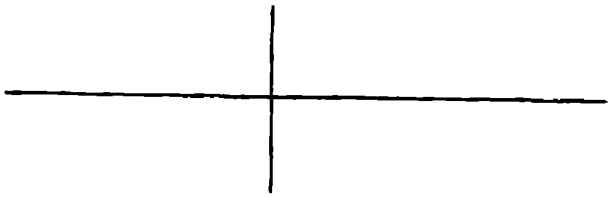
Definition: the schematic symbol for a wire is as follows:

(schematic symbol for a wire)

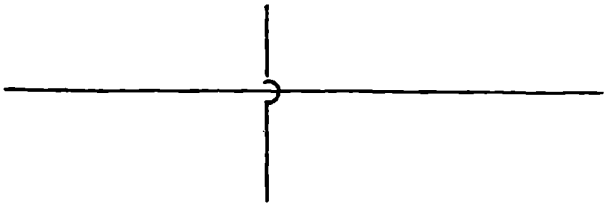
Variations on the schematic symbol of a wire:



Other variations on the schematic symbols for a wire:



a. (2 wires crossing but not touching each other)



b. same as a. (less commonly seen)

c. 2 wires connected together

d. a wire in hiding

OKAY. BUT WHAT'S SO GREAT ABOUT A WIRE? AFTER ALL, IT'S JUST A WIRE.

Answer: yes, true. What's exciting is what we can do with more than one of them.

WHAT?

Answer: well, take one wire.

Quiz (1): WHAT CAN Ø VOLTS REPRESENT?

Answer (1): (you fill in): _____, _____, _____, or _____

AND WHAT CAN 5 VOLTS REPRESENT?

Answer (2): _____, _____, _____, or _____.

Let's use the terms "one and "zero" to define 5 volts and Ø volts respectively in conjunction with the voltage on a wire. On the following schematic drawing of a wire, how many different combinations of "one" and "zero" can be true over a period of time?

Answer (3): _____

Answers: 1. OFF, False, Zero, or Low
2. ON, True, One, or High
3. Two

Now, add another wire:

_____ (Wire A)

_____ (Wire B)

NOW, HOW MANY COMBINATIONS OF VOLTAGES CAN BE APPLIED TO TWO WIRES?

Answer (4): _____

WHAT ARE THE COMBINATIONS?

Answer (5): (fill in "1" or "0" in the table:)

	Wire A	Wire B
1st combination		
2nd combination		
3rd combination		
4th combination		

4. Four

5.

0	0
0	1
1	0
1	1

or

0	0
0	1
1	0
1	1

or

0	0
1	0
0	1
1	1

Above show 3 out of 16 possible correct answers.

You need to show 00, 01, 10, and 11 in any order.

Let's assign numeric values to each combination:

Combination	Number
0 0	0
0 1	1
1 0	2
1 1	3

So now, our two wires:

_____ (Wire A)

_____ (Wire B)

can define any number between 0 and 3.

HOW ABOUT 3 WIRES? HOW MANY COMBINATIONS OF "1" AND "0" CAN THERE BE FOR 3 WIRES?

Answer (6): _____

WHAT ARE THEY?(7)

_____, _____, _____, _____, _____, _____, _____

(Hint: one answer is 000)

6. Eight

7. Answer is 000, 001, 010, 011, 100, 101, 110, 111

Let's again assign numeric values to the combinations. As a quiz, fill in the combinations for 3 and 6:

Combination	Number
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1 (8)	3
1 0 0	4
1 0 1	5
1 1 0 (9)	6
1 1 1	7

Definition: a bundle of wires bunched together to define a number is called a bus.

_____ 0 This bus has the combination 011

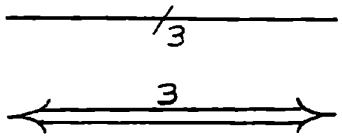
_____ 1 or the number 3.

_____ 1

8. Answer is 011

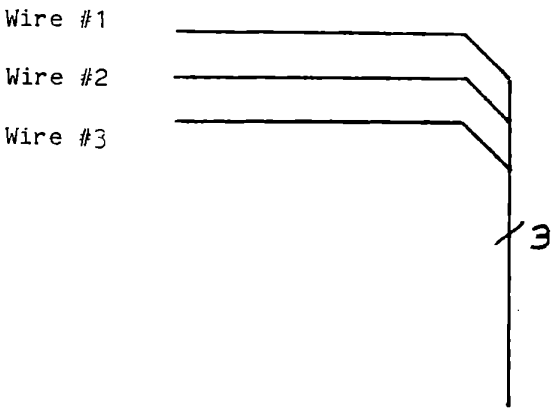
9. Answer is 110

Other schematic symbols for a bus:



Schematic often found in block diagrams

Indicates bus has 3 wires



How to divide the schematic symbol

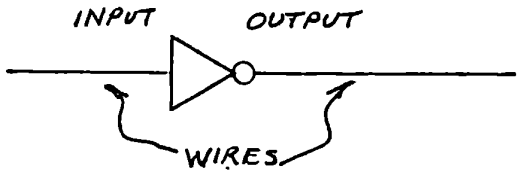
for a bus into its individual wires

WHAT OTHER THINGS CAN YOU DO WITH WIRES?

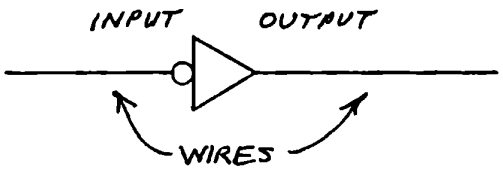
Answer: hook electronic circuits to them

LIKE WHAT KIND OF ELECTRONIC CIRCUITS?

Answer: well, one of the simplest is an inverter. An inverter has 2 wires connected to it: one wire to its input and one to its output. The schematic drawing is as follows;



They can also be drawn with the little circle or bubble at the left like this:



OKAY. WHAT'S AN INVERTER DO FOR A LIVING ANYWAYS?

Answer: simple. A "1" at the input causes an "0" at the output and a "0" at the input causes a "1" at the output.

Quiz question: fill in the table for the output of an inverter:

(Called a "truth table")

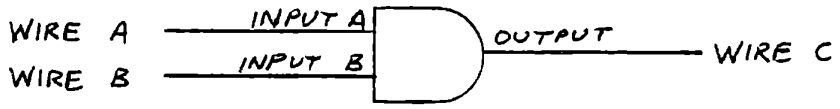
Input	Output	
0		(10)
1		(11)

HOW EXCITING! WHAT DO YOU USE INVERTERS FOR, FOR PETE'S SAKES?

Answer: lots of things. Some digital circuits need signals to be inverted. Those cases will be shown later.

WHAT OTHER CIRCUITS ARE USED?

Answer: an AND gate for one. Two wires are connected to its input and it has one output. Here's its schematic.



This circuit has the following properties:

A. When both A and B are "1", the output is one.

B. If either A or B is "0", or if they are both "0", then the output is "0".

Quiz (12): Fill in the truth table for the AND circuit:

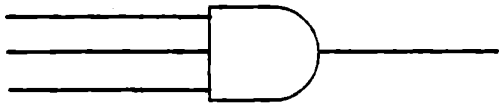
A	B	Output
0	0	
0	1	
1	0	
1	1	

12. Answer:

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

CAN THE AND CIRCUIT HAVE MORE THAN TWO INPUTS?

Answer: yes. Here is the schematic for an AND gate with three inputs:



The property of the circuit is as follows:

A. If all three inputs are "1", the output is "1"

B. If any one or more inputs are low, the output is "0"

Quiz (13): Fill in the truth table for a 3-input AND circuit:

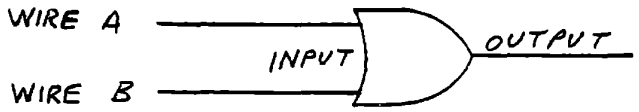
A	0	1	0	1	0	1	0	1
B	0	0	1	1	0	0	1	1
C	0	0	0	0	1	1	1	1
Output								

13. Answer:

A	0	1	0	1	0	1	0	1
B	0	0	1	1	0	0	1	1
C	0	0	0	0	1	1	1	1
Output	0	0	0	0	0	0	0	1

HOW ABOUT OTHER CIRCUITS?

Answer: sure, the OR circuit. It's schematic for two inputs and one output is as follows:



Its electrical property is as follows:

A. If both A and B are "0", the output is "0"

B. If either A or B is "1", or if both A and B are "1", then the output is "1"

Quiz (14): Fill in the truth table for an OR gate:

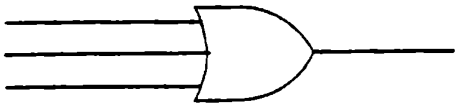
A	0	1	0	1
B	0	0	1	1
Output				

14. Answer:

A	0	1	0	1
B	0	0	1	1
Output	0	1	1	1

CAN AN "OR" CIRCUIT HAVE MORE THAN TWO INPUTS?

Answer: yes, it can. Here is a schematic symbol of a 3-input OR circuit:



Its electrical property is similar to a 2-input OR circuit:

A. If all three inputs are "0", then the output is "0"

B. In all other cases, the output is "1"

Quiz (15): Fill in the truth table for a 3-input OR gate:

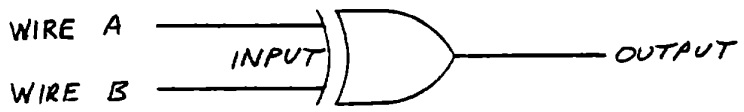
A	0	1	0	1	0	1	0	1
B	0	0	1	1	0	0	1	0
C	0	0	0	0	1	1	1	1
Output								

15. Answer:

A	0	1	0	1	0	1	0	1
B	0	0	1	1	0	0	1	0
C	0	0	0	0	1	1	1	1
Output	0	1	1	1	1	1	1	1

ANY OTHER CIRCUITS WE HAVEN'T SEEN?

Answer: one more. It's called an EXCLUSIVE-OR circuit, sometimes called an XOR circuit. Its schematic symbol is as follows:



Its characters are as follows:

A. If A and B are different; that is, if A is "0" when B is "1", or A is "1" when B is "0", then the output is "1"

B. If A and B are the same; that is, A and B are both "0" or A and B are both "1", then the output is "0"

Quiz (16): Fill in the truth table for the XOR circuit:

A	0	1	0	1
B	0	0	1	1
Output				

IS THERE A 3-INPUT XOR CIRCUIT?

Answer: yes, but you don't see them very often and we won't go into them here.

16. Answer:

A	0	1	0	1
B	0	0	1	1
Output	0	1	1	0

Definition: All of the circuits discussed so far, that is, the AND, OR, and XOR circuits are called "gates."

AS OPPOSED TO WHAT?

Answer: as opposed to another circuit called a "flip-flop".

WHAT IS A FLIP-FLOP? IS THAT A MANEUVER EXECUTED BY AN ACROBAT?

Answer: No. It is a device that stores the logic condition of a wire ("1" or "Ø").

WHY WOULD YOU WANT TO STORE ANYTHING?

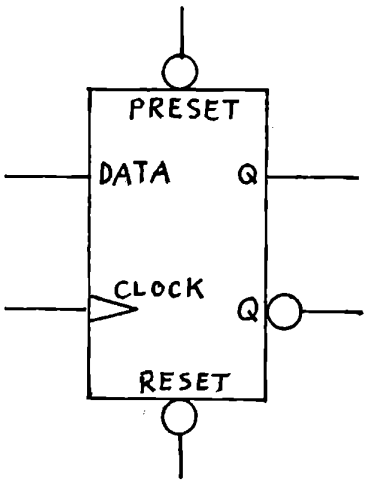
Answer: two reasons:

A. To remember the condition of a wire at a specified time.

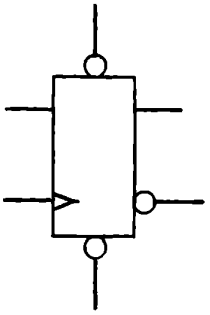
B. So the wire can be used for something else.

ALL RIGHT THEN, WHAT IS A FLIP-FLOP?

Answer: there are several different kinds of flip-flops. The commonest one is the D-flip-flop and its schematic symbol is as follows:



Most of the time it is drawn without the words as follows:



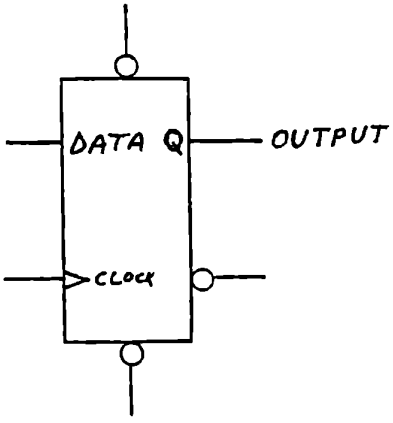
CAN YOU BREAK IT DOWN SOMEWHAT TO SHOW WHAT IT DOES?

Answer: sure. The idea of a flip-flop is to be able to capture the "1" or "Ø" state of a wire at any time electronically. There are three components needed to do this:

A. The data line. This input line is attached to the wire that you want to capture the "1" or "Ø" state on.

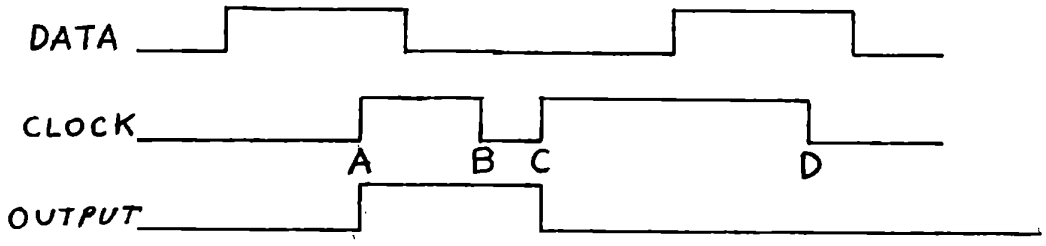
B. The clock line. This input line tells you <u>when</u> to capture the state of the data line.

C. The output. This output line holds the state "captured" on the data line when "told" to by the clock line.

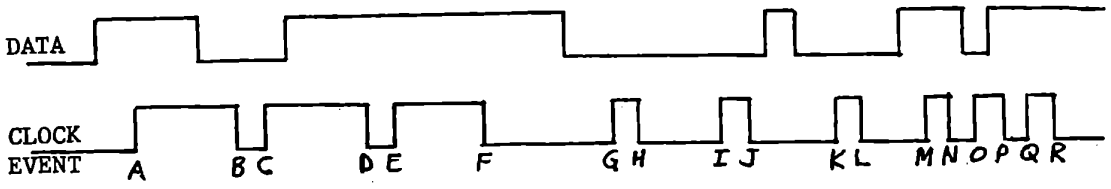


In the following illustration, you will see that the output of the flip-flop depends on what the data line was doing at the time the clock changes from a zero to a one. At event A, the clock changes (transitions) from "0" to "1", and at that time the data line happens to be "1", so the output becomes "1" also. Note: The output is <u>not</u> affected by a "1" to "0" transition in the clock; the output remains the same. So the output does not change at event B, the clock's "1" to "0" transition. Event C is a "0" to "1" clock transition when the data is "0", hence the output becomes "0" also. Finally, note that the output does <u>not</u> change at event D even though the data line is a "1". This is again because of the fact that clock "1" to "0" transitions do <u>not</u> affect the output.

In summary, the flip-flop's clock feature is simple; the output changes to become the same as the data line whenever the clock transitions from a "0" to a "1".



Quiz (17): Fill in the table below for whether the output is "0" or "1" based on the behavior of the data line and the transtions of the clock line. The first two time points are done for you.



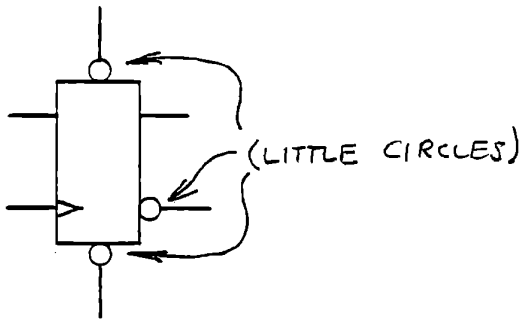
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	1																

17. Answer: EVENT

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	1	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1

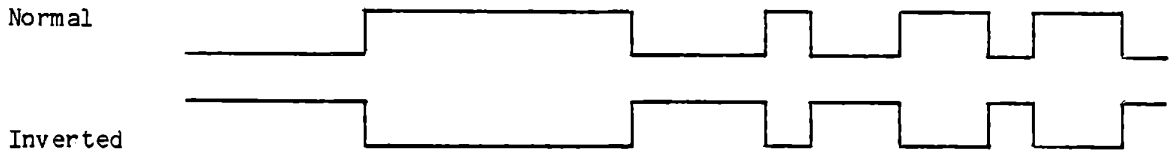
OUTPUT

WHAT ARE THE LITTLE CIRCLES (i.e. THE BUBBLES) FOR?



Answer: the little bubbles indicate that the lines connected to the bubbles are inverted - that is, instead of a "1", a "0" either appears or causes something to happen. For example: there are usually two outputs to a flip-flop: one is a "normal" output while the other is connected to a bubble and is therefore the inverted output, the opposite of the "normal" output.

Example of normal vs. inverted output



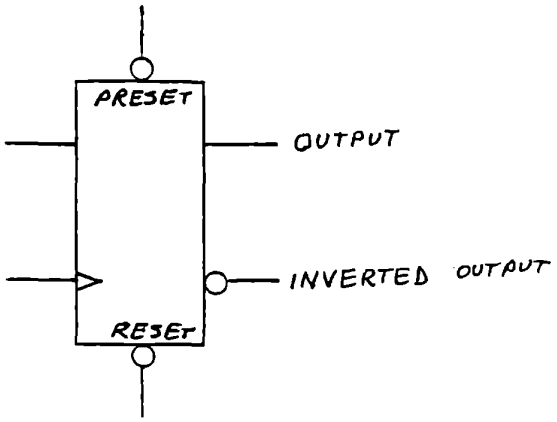
Quiz (18): Fill in the values for the inverted output for a flip-flop whose output is given. The first two are done for you.

Time interval	1	2	3	4	5	6	7	8
Output	1	1	0	0	1	0	1	1
Inverted Output	0	0						

18. Answer:

Time interval	1	2	3	4	5	6	7	8
Output	1	1	0	0	1	0	1	1
Inverted Output	0	0	1	1	0	1	0	0

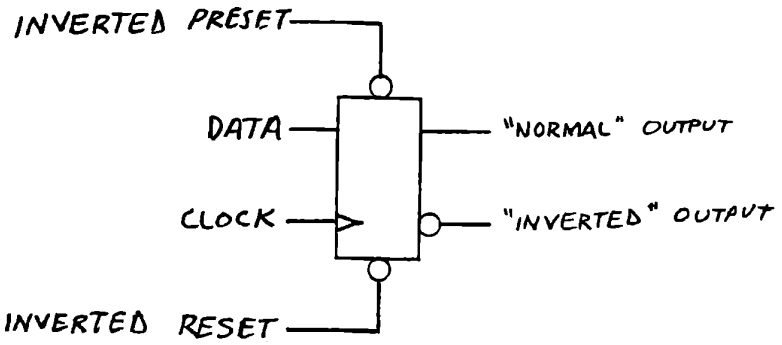
I NOTICED THAT THE PRESET AND RESET LINES WERE ALSO CONNECTED TO BUBBLES. WHAT DO THEY DO?

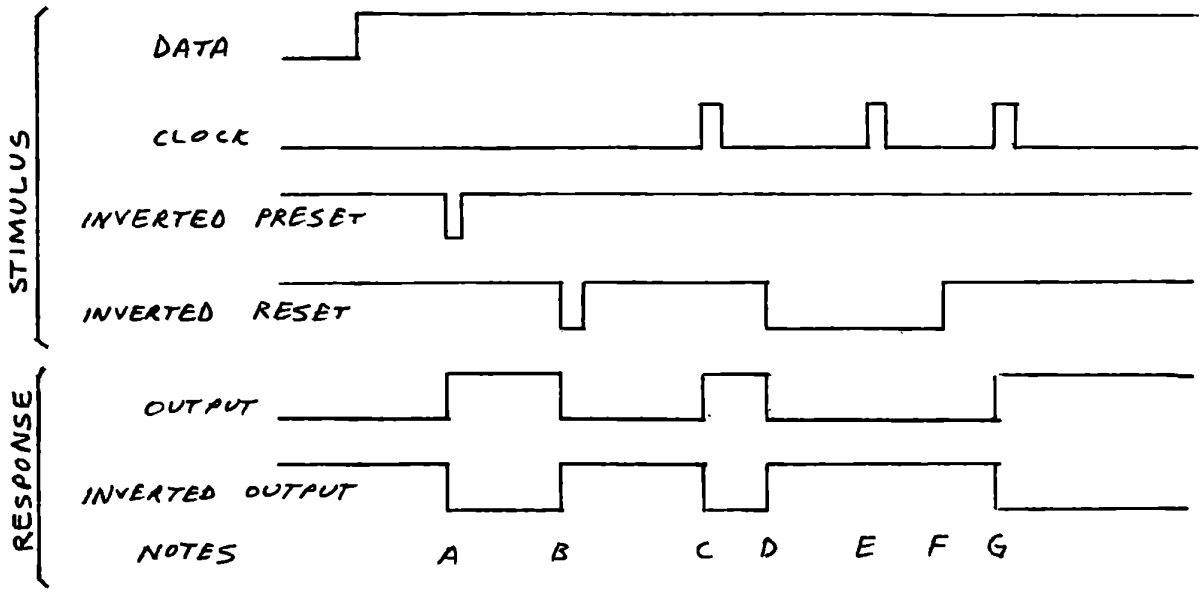


Answer: "Preset" causes the output to immediately change to a "1", regardless of its previous state or what the data and clock lines do. In the same manner, "Reset" causes the output to immediately change to a "zero".

The output is left in that condition when "Preset" or "Reset" is removed and remains there until a clock or another preset or reset occurs. Note that in this flip-flop the "Preset" and "Reset" signals are inverted; that is, the bubble indicates that the "preset" or "reset" function occurs when the line is low or "Ø".

Now note the response of the flip-flop to the following stimulus:





NOTES:

A. "Preset" = 0 causes output to go high.

B. "Reset" = 0 causes output to go low.

C. "Data" = 1 and "Clock" cause output to go high.

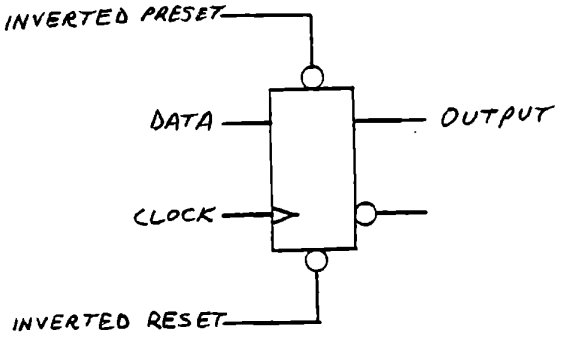
D. "Reset" causes output to go low.

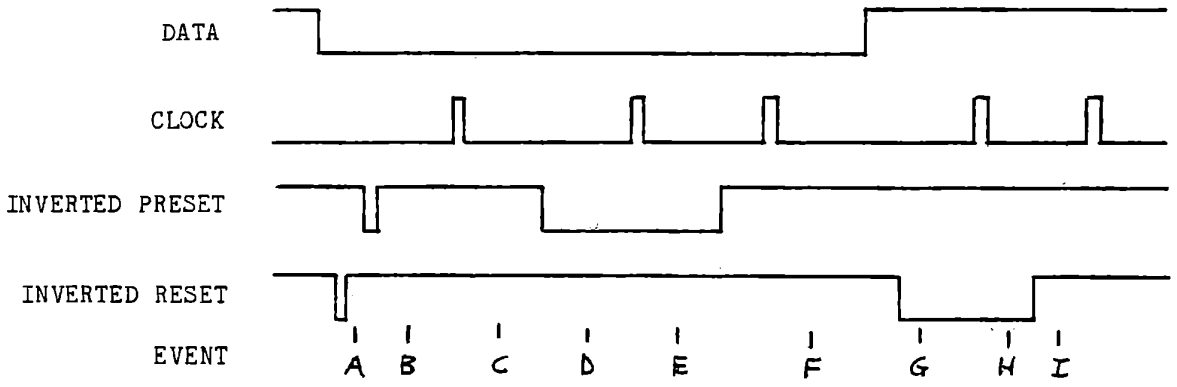
E. " Data" = 1 and "Clock" have no effect because "Reset" is still low.

F. "Reset" is released ("Reset" = 1).

G. Now "Data" = 1 and "Clock" cause output to go high.

Quiz (19): Fill in the values for the output of a flip-flop from the following diagram (called a "timing diagram"). (The first 2 are done for you).





Event	A	B	C	D	E	F	G	H	I
Output	0	1							

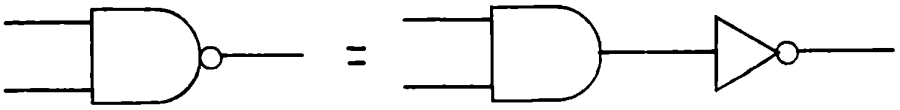
19. Answer:

Event	A	B	C	D	E	F	G	H	I
Output	0	1	0	1	1	0	0	0	0

ARE BUBBLES EVER USED WITH GATED LOGIC? THAT IS, "AND" GATES, "OR" GATES, ETC?

Answer: yes, as follows:

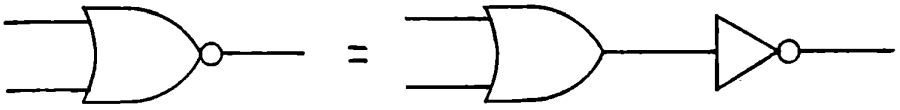
Definition: A "NAND" gate is equivalent to an "AND" gate followed by an inverter in series. Schematically:



The truth table for a "NAND" gate yields the opposite state for the output as for an "AND" gates:

A	0	1	0	1
B	0	0	1	1
Output	1	1	1	0

Definition: A "NOR" gate is equivalent to an "OR" gate followed by an inverter in series. Schematically:



Quiz (20): Fill in the truth table for the output of a "NOR" gate as follows:

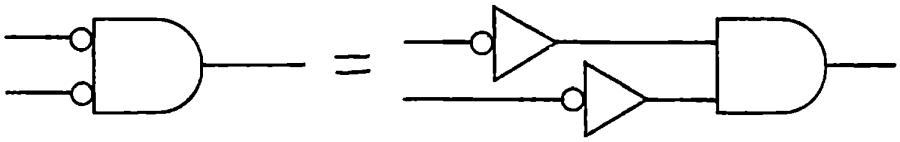
A	0	1	0	1
B	0	0	1	1
Output				

20. Answer:

A	0	1	0	1
B	0	0	1	1
Output	1	0	0	0

I SEE THAT A BUBBLE PLACED ON THE OUTPUT OF A GATE CAUSES THE "NORMAL" OUTPUT OF THAT GATE TO BE INVERTED. IS A BUBBLE EVER PLACED ON THE <u>INPUT</u> OF A GATE?

Answer: yes. As one would guess, it causes the input to be inverted going into the gate. For example:

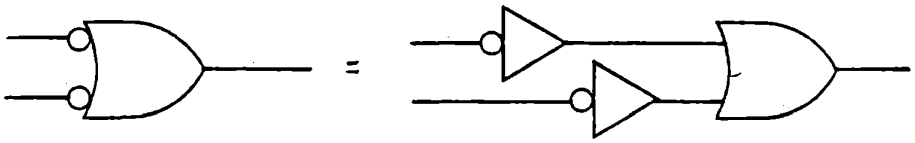


Its truth table looks as follows:

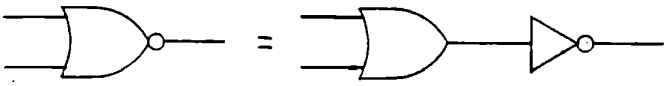
A	0	1	0	1
B	0	0	1	1
Output	1	0	0	0

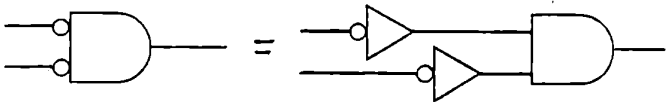
which is the same as that for a "NOR" gate.

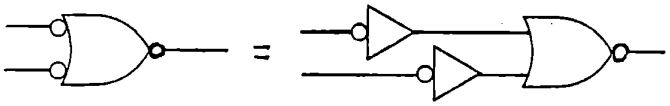
Quiz (21): Examine the following schematic symbol:

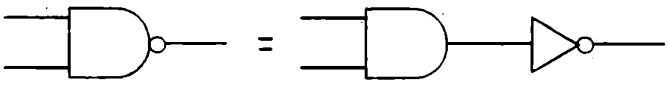


One of the following schematic diagrams has the same circuit behavior as the above symbol: (Hint: draw a truth table for each circuit)

a. 

b. 

c. 

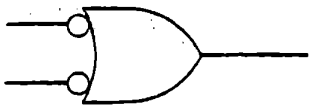
d. 

WHICH ONE IS IT?

Answer: ____

21. Answer: d

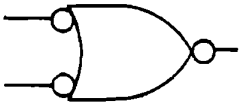
Quiz (22): Fill in the truth table for the output of the symbol whose schematic is as follows:



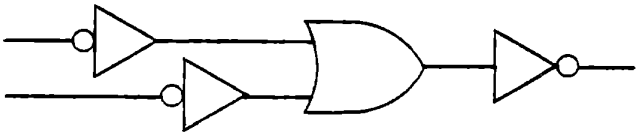
A	0	1	0	1
B	0	0	1	1
Output				

DO SCHEMATIC SYMBOLS EVER USE BUBBLES ON BOTH THE INPUTS AND THE OUTPUTS?

Answer: yes. For example, the following schematic



is equivalent to



Quiz (23): Fill in the truth table for the previous schematic symbol:

A	0	1	0	1
B	0	0	1	1
Output				

Quiz (24): What previous logic gate is equivalent to the above symbol?

(multiple choice)

a. An "OR" gate

b. An "AND" gate

c. A "NOR" gate

d. A "NAND" gate

22.

A	0	1	0	1
B	0	0	1	1
Output	1	1	1	0

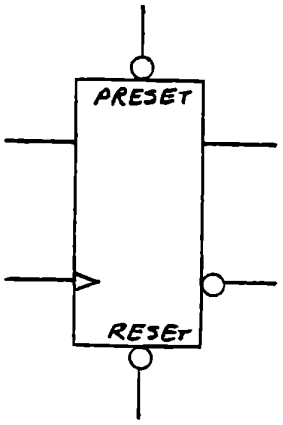
23.

A	0	1	0	1
B	0	0	1	1
Output	0	0	0	1

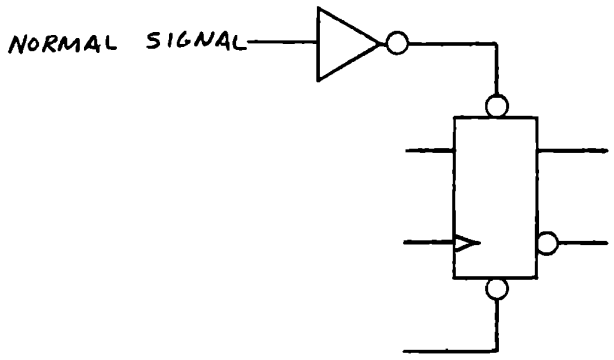
24. Answer: b

WHY WOULD ANYONE DRAW A SYMBOL WITH BUBBLES ON ALL INPUTS AND OUTPUTS WHEN THERE IS AN EQUIVALENT SYMBOL THAT DOESN'T REQUIRE ANY BUBBLES AT ALL?

Answer: to clarify the intention of the circuit. For example, on the flip-flop, "preset" and "reset" are inverted inputs.



If a normal signal is intended to preset the flip-flop when it is high, it must have an inverter placed between it and the preset input.

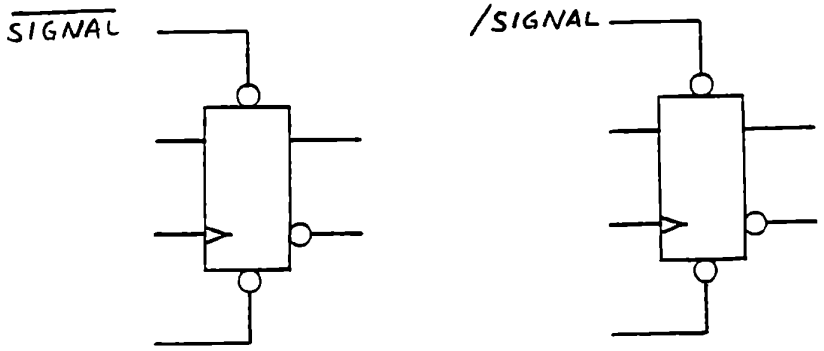


Such a signal is called an "active high" signal because it causes an action when its state is "1".

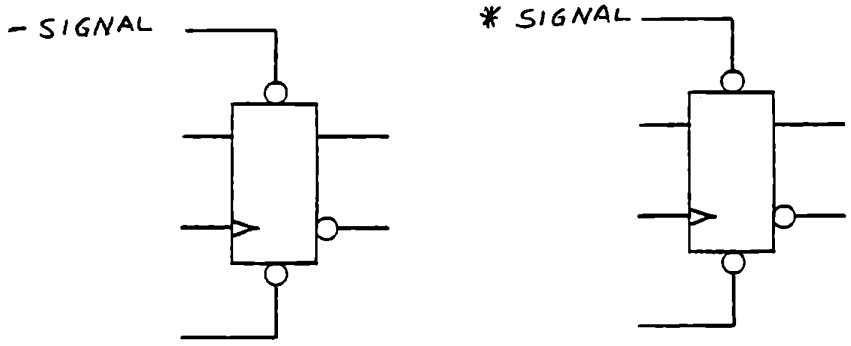
BUT WHAT IF THE SIGNAL CAUSES AN EFFECT WHEN IT IS "Ø"?

Answer: such a signal is called "active low" and is denoted on a schematic diagram in any of the following ways (presume the signal name is "SIGNAL") .

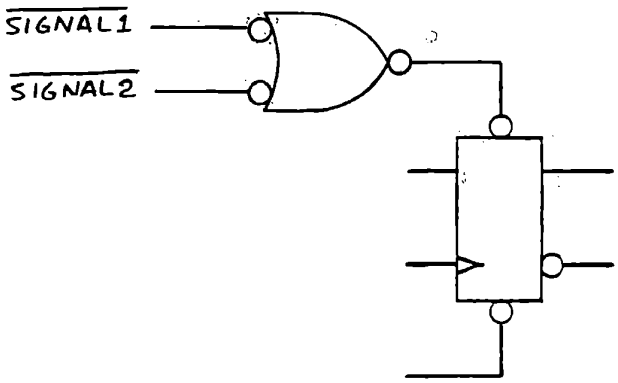
a) With a bar over it or a slash in front of it



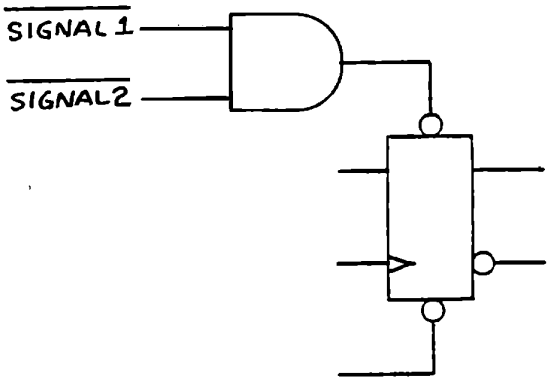
b) With a "-" or "*" before it



Now, say that either of two active low signals can preset our flip-flop. This is an "OR" function of two active low signals as follows:



The above schematic directly states that if either $\overline{\text{SIGNAL1}}$ or $\overline{\text{SIGNAL2}}$ is "0" ("true" in an active low sense), then the flip-flop will be preset. This is more straightforward than using the active high symbol.

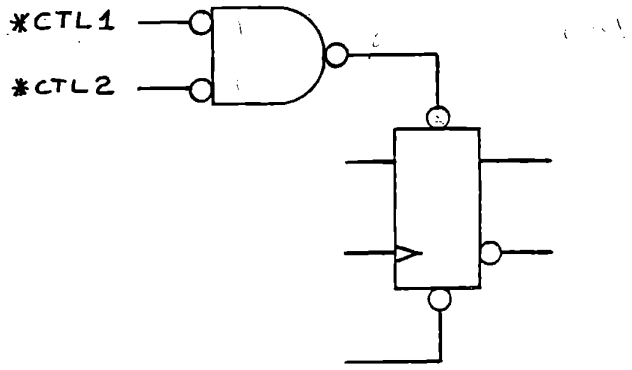


This schematic is functionally identical to the previous one, but its meaning is obscured by having to switch from active low signals to an active high device back to an active low preset. In this case, the AND gate indicates that the preset is not activated when both SIGNAL1 and SIGNAL2 are "1", which is logically accurate but does not really communicate the intention of the designer.

Quiz (24): (multiple choice) - Given the following truth table for the inverted NAND:

A	0	1	0	1
B	0	0	1	1
Output	0	1	1	1

what is the design intention of the following schematic?



Choose one:

[ ] a) The flip-flop is preset if either *CTL1 or *CTL2 is "1".

[ ] b) The flip-flop is preset if either *CTL1 or *CTL2 is "0".

[ ] c) The flip-flop is preset if both *CTL1 and *CTL2 are "1".

[ ] d) The flip-flop is preset if both *CTL1 and *CTL2 are "0".

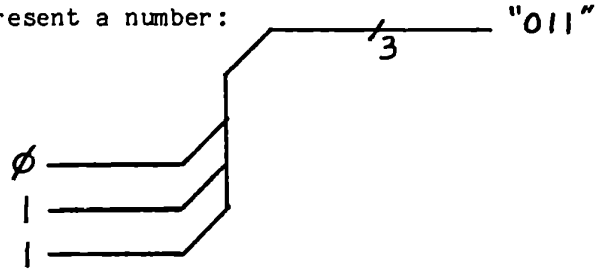
24. Answer is d)

SPECIAL INSTRUCTIONS TO STUDENT:

Take a breather. In the next section we are going to look at devices that connect to busses.

COULD YOU REFRESH ME ON WHAT A BUS IS?

Answer: certainly. A bus is simply a bundle of wires upon which information may be transmitted. This information is implemented as "1" and "0" voltages on the wire that represent a number:



Above Bus has number "011" on it.

ARE THERE DEVICES THAT CONNECT TO BUSSES?

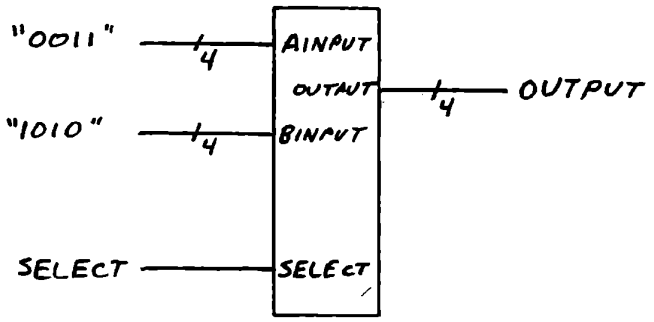
Answer: you bet. One of the most common devices is the multiplexer.

WHAT'S THAT?

Definition: A multiplexer is a digital electronic circuit that has the ability to choose one of several bus inputs.

GIVE ME AN EXAMPLE OF A MULTIPLEXER

Okay: Here is a schematic that chooses one of two 4-wire busses:



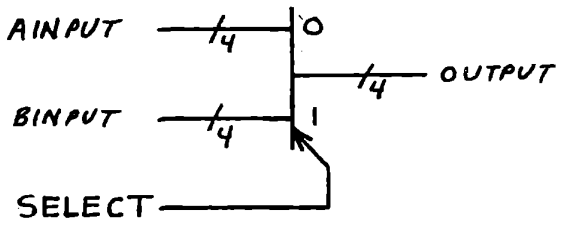
If SELECT = "0", the output will contain the A input, in this example "0011".

If SELECT = "1", the output will contain the B input, in this example "1010".

Notice that multiplexers are more complex than gates or flip-flops. The above multiplexer, not including power connections, requires 13 wires total connected to it as follows:

SELECT	1 line
AINPUT	4 lines
BINPUT	4 lines
OUTPUT	4 lines
TOTAL	13 lines

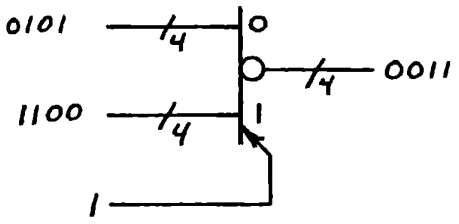
There are several ways to draw a multiplexer in a block diagram. For instance, the previous example could be drawn as follows:



Quiz (25): If AINPUT = "1010" and BINPUT = "0010", and SELECT = "0", what will OUTPUT be?

Answer:_____

Bubbles may be used on multiplexers just as in gates and flip-flops. For example, a bubble placed on the output means that all 4 output lines are inverted:



Quiz (26): If the select line in the above example is equal to 0, then what is the output?

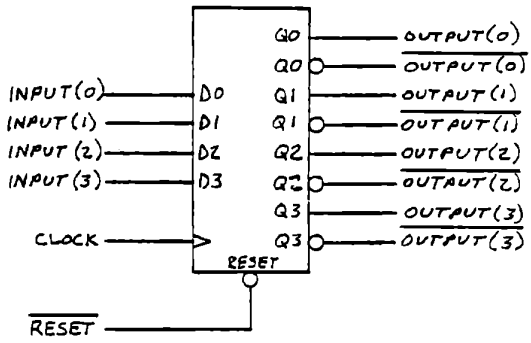
Answer: _____

25. Answer is "1010"

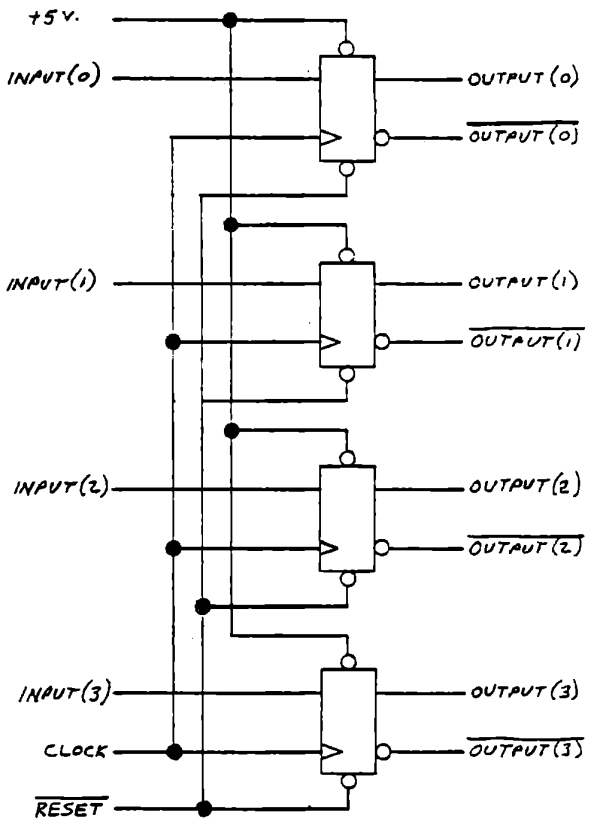
26. Answer is 1010

WHAT ABOUT FLIP-FLOPS? ARE ANY OF THEM EVER CONNECTED TO A BUS IN SUCH A WAY AS TO BE ABLE TO STORE A NUMBER ON THE BUS?

Answer: you bet. As a matter of fact, flip-flops connected in such a fashion form a circuit called a register. A register schematic symbol can look like the following:



which internally is an array of flip-flops hooked up as follows:



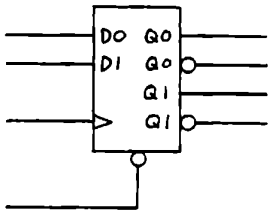
Note the following about the flip-flops connected as a register:

1) All of the clock inputs are connected together

2) All of the reset inputs are connected together

3) The preset inputs are internally connected to a logic "1" so that they are deactivated. Therefore, the user does not have access to the preset lines, they are always disabled.

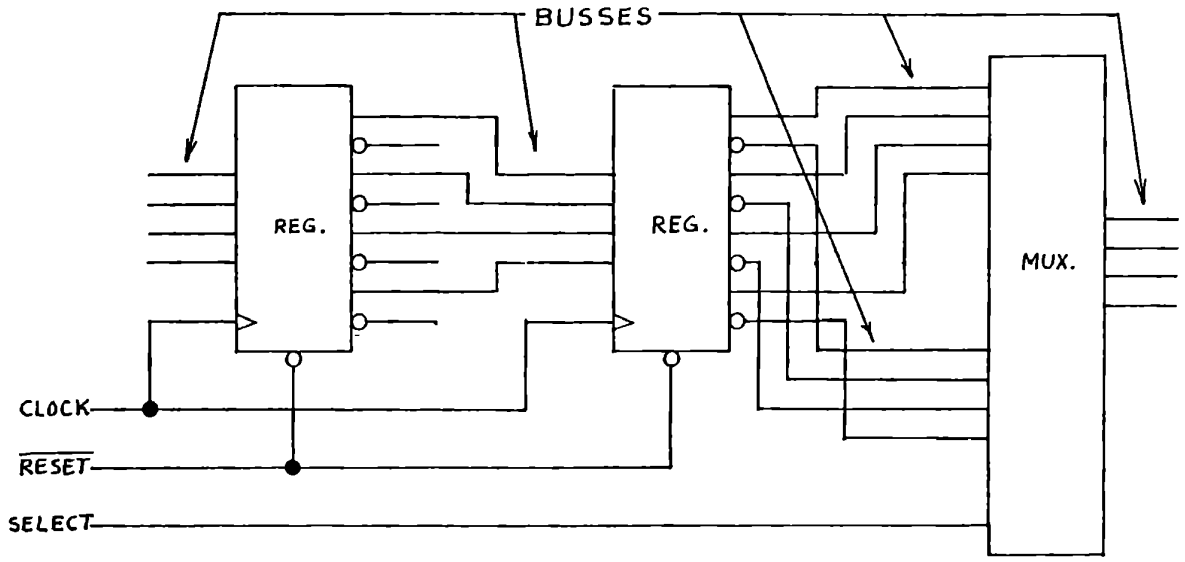
Quiz (27) : How many flip-flops would the register in the schematic circuit be likely to contain?



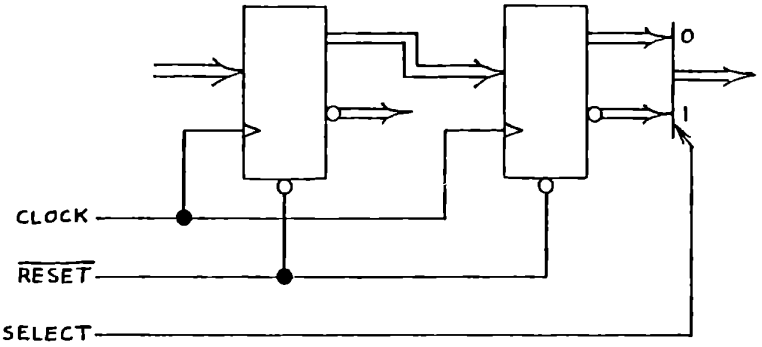
Answer: 2

27. Answer is 2.

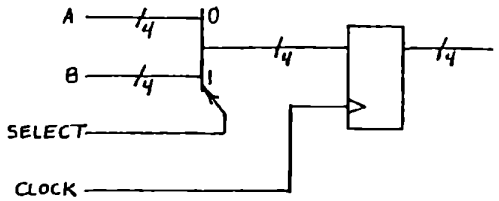
Registers, multiplexers and busses can be shown interconnected as in the following schematic



On a block diagram or simplified schematic, the above schematic would be drawn as:

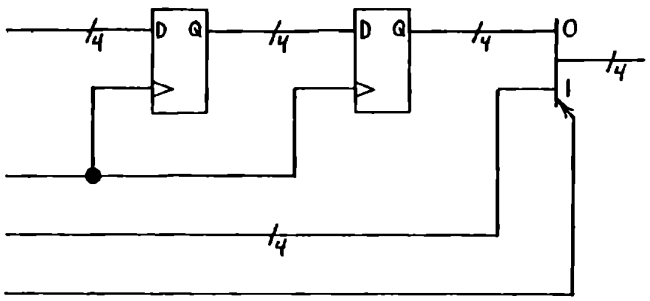


There are many kinds of digital circuits that are used with busses: registers, multiplexers, counters, and arithmetic logic units to name a few. For example, a circuit to route one of two 4-bit busses (A and B to a register might be drawn as follows:

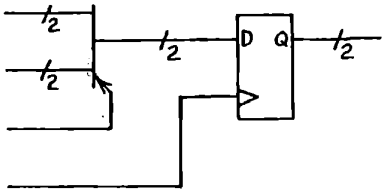


Quiz (28) : One of the following circuits selects one of two register outputs. Which is it?

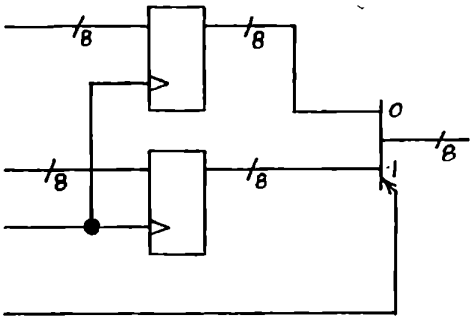
a



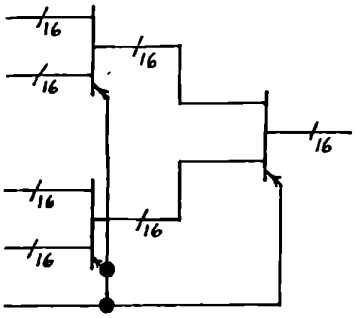
b



c



d



Answer: ____

That completes the preparation for the course in which you are enrolled. Thank you for reading it, and for working out the quizzes. Your knowledge of digital fundamentals is now up to speed for the course. Please bring this material with you when you come to the course.

28. Answer is c.

INDEX

8085 Microprocessor	11-17
8212 I/O Port	8-2
8251 USART	8-33, 9-20
8255 Mode 1	9-3
8255 Mode 2	9-9
8255 PIA	5-28, 8-9
Accumulator	1-38, 1-51
Accumulator/Carry Instructions	11-7
ACI Instruction	4-28, 11-9
ADC A as Shift Left	7-17
ADC Instruction	4-12, 10-4
ADD Instruction	4-4, 4-46, 11-9
Addend	10-5
Addition by Counting	3-13
ADDR Command	1-59, 2-11, 2-29, A-16
Address Bus	1-27
Address Size	1-23
Addresses, Input/Output Ports	8-9
ADI Instruction	2-1, 2-28, 4-29
Alternate Subroutine Entries	6-68, 6-70
Alternative Memory Addressing	5-25
AMTS Description	A-1
AMTS Setup	I-1
AMTS Test	I-2
ANA Instruction	7-30
AND Function (ANA)	7-30
ANI Instruction	7-33
A-Register	1-51
Arithmetic and Logic Unit	1-31
Arithmetic Intructions	4-121, 11-9
Arithmetic Overflow	10-52, 10-80
Arithmetic Shift	7-2, 7-17
Array of Input/Output Ports	8-5, 8-33
ASCII	9-19
Asynchronous Communication	9-17
Asynchronous Receiving Exercise	9-33
Augend	10-5
Auxiliary Carry Flag	10-29, 11-12

Base	1-12
Basic Concepts	1-2
Binary Addition	1-16, 10-2
Binary Entry and Display Exercise	7-22
Binary Multiplication	10-33, 10-37
Binary Number System	1-10, 1-14, 10-44
Binary Subtraction	10-13
Binary to Decimal Conversion	B-1
Bit	1-18
Bit Manipulation	7-1
Bit Masking	7-34
Bit Position	1-18
Bit String	1-18
BKENT Subroutine	9-49
BKLOC Subroutine	9-49
BKMEM Subroutine	9-51
BKRMV Subroutine	9-49
Branch Instructions	11-13
Breakpoints	A-24
Clearing	4-82, A-26
Data Change	6-125, 8-88, A-11, A-24
Entering	4-77, A-11, A-24
Program Counter	A-11
Program Entry and Removal	9-49
Protection from Growing Stack	6-125
Repetition Count	A-25
Stack Usage	6-120
To Count Instructions	A-26
BRK Command	4-77, A-24
Bus	1-27
Byte	1-22

Calculating Delay Times	9-61
Call if Minus Instruction	10-51
Call if Plus Instruction	10-51
CALL Instruction	6-13, 6-16, 11-13
Calling Program	6-12
Carry	4-6
Carry and Zero Flags	4-6, 11-10
Carry Flag	4-6, 7-1, 7-34, 11-7, 11-12
CC Instruction	6-52, 11-13
Change Sign, Add, Subtract Exercise	10-53
Change Sign Exercise	10-45, 10-49, 10-53
Chip Select Logic	5-17
CLEAR Subroutine	6-147, A-53
Clear Carry	4-47, 7-32, 7-34
Clearing Breakpoints	4-82, A-26
Clock	1-38
CLR Command	4-82, A-21
CLRGT Subroutine	6-147, A-53
CLRHI Subroutine	A-53
CLRLP Subroutine	6-147, A-53
CM Instruction	10-51
CMA Instruction	7-29
CMC Instruction	7-34
CMP Instruction	4-111
CNC Instruction	6-52, 11-13
CNZ Instruction	6-52, 11-13
Codes, Hex	4-159, 11-1
Coding	1-53
Command Keys	2-29, A-11
Comments	1-53
Communication, Serial	9-14
Asynchronous	9-17, 9-54
Synchronous	9-16
Word Mark in Serial Communication	9-14
Comparison Instructions	4-110
Complement	7-29, 7-34
Computer, Definition of	1-2
Computer to Computer Interface	9-7
Conditional CALL and RET	6-52
Conditional Jump	3-2, 4-40
Control of Monitor Functions	A-31
Control Ports, 8255	8-18
Control Signals	5-3
Cosine	C-1
Count Instructions by Breakpoint	A-26
Counting Instructions	11-5
CP Instruction	10-51
CPE Instruction	11-13
CPO Instruction	11-13
CPU	1-2, 1-31
Cycle, Machine	2-2
CZ Instruction	6-52, 11-13

DAA Instruction 10-25, 11-8
DAD Intruction 6-78, 10-5
DAD SP Instruction 6-116
Data Bus 1-27, 5-26
Data Change Breakpoints 6-125, 8-88, A-24
Data Format 9-1, 9-18
Data Rate for Recording A-8
Data Structure 4-130
Data Transfer Instructions 11-2, 4-119
DBIN Signal 5-4
DBYTE and DBY2 Subroutines 6-38, 6-144, A-49
DCR Instruction 3-13, 3-21, 4-121, 11-5
DCX Instruction 4-71
DCX SP Instruction 6-116
Debouncing 8-26
Debug Mode A-11
Debugging I/O Programs 8-20
Decimal Addition and Subtraction 10-25
Decimal Multiplication 10-39
Decimal Number System 1-12
Decimal, Packed 10-25
Decimal to Binary Conversion B-1
Decoded Control Signals 5-6
Decoder 1-26
Decrement 3-13, 4-71, 4-121
Delay Loops 4-73
DELAY Subroutine 6-148, A-54
Delay Times, Calculation of 9-61
DELYA Subroutine 6-148, A-54
DELYC Subroutine A-55
DELYT Subroutine A-55
DI Instruction 8-54, 8-71
DIGHI Subroutine A-49
Digit 1-12
Digital Logic F-1
DIGSW Subroutine A-49
Direct Memory Access 5-20, 5-28, 8-39, 8-45
Directory 4-141
Disabling Monitor Interrupts 8-91, A-31
Dispatch Table 7-73
Display Codes for Characters 4-62
Display MTS 4-53, 8-41, A-12, A-46, A-48
Display Register Pair 4-103
DISPR Subroutine 6-142, 8-92, A-48
DMA and Interrupts 5-28
DMA Enable Signal 5-20, 8-43
DMEM Subroutine 6-144, A-49
DMWD Subroutine A-51
Do Nothing Program 6-8
Double Precision 4-11, 4-16, 6-78, 10-2
Double Precision Multiply 6-48, 6-56, 10-35
Drivers 6-83, 6-110
DWORD Subroutine 6-146, A-51

Effect of DI and EI 8-71
EI Instruction 8-53
Enabling Monitor Interrupts 8-91, A-31
ENMEM Subroutine A-43
ENTBY Subroutine 6-140, A-41
Entering Breakpoints 6-34, 6-63, A-25
ENTWD Subroutine 6-141, A-42
ERRDS Subroutine A-54
Error Checking Character 9-46
Error Display A-28
Exchange Instructions 6-107
Exchange Stack Top 6-108
Exclusive or (XRA) 1-52, 7-32
Execution 1-58
Exercise 1 1-53
Exercise 2 2-1, 2-10
Exercise 3 2-16, 2-23
Exercises (see sequential listing, page i-15 of index)
Experiment with DAA 10-31
Exponential Function C-1
External Interrupt Exercise 8-100
External Interrupt Experiment 8-70

Fetch 1-35
Fetch Cycle 5-31
Fixed Point 10-69
Flag Register 11-10
Flags 3-2, 4-6, 4-44, 11-10
Flags Affected by Instructions 11-11
Floating Point 10-69
Flow Charts 3-7
Flow Control Techniques 7-72
Format, Data 9-1
Four Byte Addition Exercise 10-6
Four Byte Subtraction Exercise 10-16
Fractional Numbers 10-68

GETKY Subroutine 6-37, 6-138, A-38
Global Subroutines 6-133
Growing Stack Problem 6-125

HALT Instruction 8-65
Handshaking, 8255 8-13
Hardware 1-2
Hex Codes for 8080 Instructions 4-159
Hexadecimal 1-19
HLDA Signal 5-4, 8-43
HLT Instruction 8-65, 10-11
HLTA Signal 5-6
HOLD Signal 5-29, 8-41
Hundreds Complement 10-59

If-Then-Else Construct 7-11
Immediate Instructions 4-25, 4-28
Immediate Instructions - Arithmetic 2-1, 4-28, 4-46, 4-47, 11-9
Immediate Instructions - Logical 7-33
Immediate Instructions - LXI 4-69
Immediate Instructions - MVI 4-25, 4-46, 11-2
IN Instruction 8-6
Inclusive Or (ORA) 7-32
Increment 1-52, 4-71, 4-121
Indirect Addressing 4-96
Initiation of I/O 8-1, 8-49
In-Line Programming 6-2
INP Signal 5-6
Input/Output 1-3, 8-1, A-35
Input/Output Instructions 8-6, 11-15
Input/Output Ports 8-2
  8212 I/O Port 8-2
  8251 USART 8-33, 9-20
  8255 Mode 1 9-3
  8255 Mode 2 9-9
  8255 PIA 5-28, 8-9
  Addresses 8-9
  Array of Input/Output Ports 8-5, 8-33
Input/Output Techniques 8-1
  Direct Memory Access 5-20, 5-28, 8-39, 8-45
  Initiation if I/O 8-1, 8-49
  Interrupt Driven I/O 8-52
  Isolated I/O 8-2, 8-8
  Memory Mapped I/O 8-35
  Parallel I/O 9-3
  Programmed I/O 8-49
  Repetitive DMA 8-41
  Serial Input/Output 9-14
  Timed Input/Output 8-76

INR Instruction	1-52, 2-28, 4-121, 11-5
Instruction	1-34
Instruction Codes	1-50, 4-159
Instruction Cycle	1-34
Instruction Effects on Flags	11-10
Instruction Execution	1-58, 2-3
Instruction Register	1-37
Instruction Timing	9-61
Instructions (see page i-18 of index for	alphabetic listing)
Instructions, Hex Codes	4-159
Instructions, Summary	11-1
INT Signal	8-52
INTA Signal	5-6, 5-7, 8-52
INTE Signal	8-52, 8-67
Intel 8085 Microprocessor	11-17
Intel 8228 System Controller	5-9
Internal Address Decoding, Figure 5-4	5-14
Interrupt Driven I/O	8-52
Interrupt Entry Points	A-33
Interrupt Logic	8-52
Interrupt Service - Combined with Monitor	8-99
Interrupt Service - Main Program	8-91
Interrupt Service Routine Exercise	8-81
Interrupt Service Routines	6-134, 8-73, 9-21
Interrupt Service Testing	8-83
Interrupt System, MTS	8-67
Interrupts	5-28, 8-52
Interrupts and DMA - Introduction	5-28
Interrupts, Priority	8-76
Interrupts, Timed	8-76
Interrupts, Vectored	8-75
Interrupts, with MTS	8-77
INX Instruction	4-71
INX SP Instruction	6-116
I/O Devices	1-3, 8-2, 8-9
I/O Instructions	8-6
I/O Ports (see also Input/Output Ports)	8-2
I/O Techniques (see Input/Output)	8-1
IOR Signal	5-7
IOW signal	5-7
Isolated I/O	8-2, 8-8
JC Instruction	4-40, 4-45
JM Instruction	10-51
JMP Instruction	2-20, 2-28
JNC Instruction	4-40, 4-45
JNZ Instruction	3-3, 3-22
JP Instruction	10-51
JPE Instruction	11-13
JPO Instruction	11-13
JZ Instruction	4-40, 4-45

Keyboard Display Exercise 8-26
Keyboard Input 8-15, A-35
Keyboard Input Exercise 8-15
KEYS Subroutine A-39

LDA Instruction 2-16, 2-28
LDAX Instruction 4-59
Least Significant Bit 1-18
LHLD Instruction 4-97
Linear Select 8-7
Load Memory from Keyboard A-4
Load Program from Tape A-9
Loading a Program 1-55, A-
Local Subroutines 6-134
Logic Functions 7-1, 7-29
Logic Functions Exercise 7-35
Logic Functions - Immediate 7-33
Logic Instructions 7-29, 11-9
Logical Rotate 7-18
Longitudinal Redundancy Check 9-46
Loops 3-1
LRC Character 9-46
LXI Instruction 4-69
LXI SP Instruction 6-116

M1 Signal 5-6, 5-7, 8-67
Machine Cycle 2-2
Machine States and Transitions 5-31
Masking 7-34
MEM Command 1-48, 2-29, A-17
Memory 1-2, 1-20, 5-1
Memory Access 1-26
Memory Access Timing, Figure 5-8 5-30
Memory Address 1-23, A-14, A-15, A-17
Memory Addressing 4-57, 4-87, 4-88, 5-12
Memory Change Breakpoints 8-88, A-24
Memory Contents, Changing 1-48, A-17
Memory Contents, Inspecting 1-46, A-17
Memory Enabling 5-19, A-17
Memory Location 1-23
Memory Location, Use as a Register 4-87
Memory Mapped I/O 8-35
Memory Reference Instructions 4-88
Memory, Shared 9-11
Memory Signals and Timing 5-31
Memory Size 1-23
Memory Technology 5-11
Memory Write Cycle 5-33
MEMR Signal 5-6, 5-7
MEMW Signal 5-7
MENAB Subroutine A-40
Minimum Chip Select, Figure 5-7 5-24

Minuend 10-16
MINUS - see Sign Flag 10-51
Mnemonic 1-50
Module Specification 6-6
Modules, Program 6-1
Monitor 1-41, A-4
Monitor Breakpoints 4-77, 6-124, A-11, A-25
Monitor Commands A-13
Monitor Data Storage 6-124, A-14
Monitor Effect on Program Speed 4-83
Monitor Enable/Disable 8-91, A-31
Monitor Entry 3-9
Monitor Functions A-4
  ADDR Command 1-59, 2-11, 2-29, A-16
  Breakpoint Entry 4-77, 9-49, A-25
  Breakpoint Operation A-11
  Breakpoint Removal 4-82, 9-49, A-26
  BRK Command 4-77, A-24
  CLR Command 4-82, A-21
  Control of Monitor Functions A-31
  Data Rate for Recording A-8
  Debug Mode A-11
  Display 1-43, A-46, A-48
  Error Display A-28
  Interrupt Entry Points A-33
  Load Memory from Keyboard A-4
  Load Program from Tape A-9
  MEM Command 1-48, 2-29, A-17
  Monitor Commands A-13
  Monitor Data Storage 6-124, A-14
  NEXT Command 1-46, 2-30, A-20
  REG Command 1-58, 2-29, A-22
  Register Pair Display 4-103, A-20
  RESET, Data Recovery A-29
  RUN Command 2-12, 3-1, A-11, A-24
  STEP Command 1-58, 3-1, A-11, A-23
  Store Program on Tape A-6
Monitor Subroutines 6-136
    (see also alphabetic listing, page i-17 of index)
Monitor Tape Programs 9-44
Most Significant Bit 1-18
MOV Instructions 4-2, 4-119
M-Register 4-87
MTS Configuration, Figure 5-1 5-2
MTS Display 4-53, 8-41, A-46
MTS Interrupt System 8-67
MTS System Controller Logic 5-9
Multiple Precision 4-11, 4-91, 10-2
Multiplicand 4-30
Multiplication and Division by 2 7-3
Multiplication by Addition 4-30
Multiplier 4-30
MVI Instruction 4-25, 4-46, 11-2

NEC 8080 Microprocessor 11-17
Negative Numbers 10-45, 10-59
Nested Do Nothing Subroutines 6-24
NEXT Command 1-46, 2-30, A-20
NOP Instruction 1-51
Number Representations 10-44
  Decimal, Packed 10-25
  Fixed Point 10-69
  Floating Point 10-69
  Fractional Numbers 10-68
  Hundreds Complement 10-59
  Negative Numbers 10-45, 10-59
  Packed Decimal 10-25, 10-59
  Sign and Magnitude 10-45
  Signed Binary Numbers 10-45
  Signed Decimal Numbers 10-59
  Twos Complement 10-45
Number System 1-10, 1-21

Operands 4-1
OR, Exclusive 7-32
OR, Inclusive 7-31
ORA Instruction 7-32
ORI Instruction 7-33
OUT Instruction 8-6
OUT Signal 5-6
Overflow, Arithmetic 10-52

Packed Decimal 10-25, 10-59
Paper Tape 9-3
Parallel Input/Output 9-3
Parity Flag 11-11
Partial Decoding 5-23
PCHL Instruction 6-8
Peripheral Interface Adaptor 5-27
PLUS - see Sign Flag 10-51
Polling 8-76
POP Instruction 6-99
POP PSW Instruction 6-105
Port Addresses 8-9
Power 1-12
Priority Interrupt Systems 8-76
Processor Status Word 6-105
Program 1-7
Program Counter 1-35
Program Exercises (see sequential listing, page i-15 of index)
Program Loops 3-1
Program Modification by Input Data 10-21
Program Modules 6-21

Program Specification 1-53
Program Timing 9-61
Programmed I/O 8-49
Programmed Monitor Entry 3-9
PSW 6-105
PUSH Instruction 6-99
PUSH PSW Instruction 6-105

RAL Instruction 7-1
RAM 1-28
RAM Chip Selection 5-19
Range of Signed Numbers 10-48
RAR Instruction 7-2
RC Instruction 6-52
Read-Write Control 5-27
READY Signal 5-4
Receiving, Asynchronous 9-20, 9-33, 9:57
Reentrant Subroutines 6-134
REG Command 1-58, 2-29, A-22
Register 1-33, 4-1
Register M 4-87
Register Pair 4-57, 4-69
Register Pair Addition 6-78
Register Pair Display 4-103, A-20
Register Pair HL 4-87, 4-97
Register Pair Instructions 4-69, 4-87, 4-97, 6-78, 6-107
Repetition Count, Breakpoints A-26
Repetitive DMA 8-4
Representations of Numbers 10-44
RESET Command 1-46
RESET, Data Recovery A-29
RET Instruction 6-13, 6-20
Return if Minus Instruction 10-51
Return if Plus Instruction 10-51
RLC Instruction 7-18
RM Instruction 10-51
RNC Instruction 6-52
RNZ Instruction 6-52
ROM 1-28
ROM Chip Selection 5-20
Rotate Exercise 7-3
Rotate Instructions 7-1, 7-18, 11-8
RP Instruction 10-51
RPE Instruction 11-13
RPO Instruction 11-13
RRC Instruction 7-18
RST 4 Instruction 3-9, 3-22, 4-47, 4-118, A-33, A-34
RST 7 Generation of 8-65
RST Command 1-46
RST Instructions 8-55
RST Interfaces 8-61
RUN Command 2-12, 3-1, A-11, A-24
RZ Instruction 6-52

S-100 Bus D-1
SBB Instruction 4-18, 4-21, 4-47, 10-14
SBI Instruction 4-29, 11-9
SCAN Subroutine 6-137, A-37
Sensor Correction, Version 1 4-125
Sensor Correction, Version 2 4-140
Sensor Correction, Version 3 6-29, 6-89
Serial Communication 9-14
Serial Transmission Exercise 9-21
Set Carry 7-34
Shared Memory 9-11
Shift Instructions 7-2, 11-8
SHLD Instruction 4-97
SHLRT Subroutine A-57
Sign and Magnitude 10-45
Sign Flag 10-51, 11-11
Signals, 8080
  DBIN Signal 5-4
  DMA Enable Signal 5-20, 8-43
  HLDA Signal 5-4, 8-43
  HLTA Signal 5-6
  HOLD Signal 5-29, 8-41
  INP Signal 5-6
  INT Signal 8-52
  INTA Signal 5-6, 5-7, 8-52
  INTE Signal 8-52, 8-67
  IOR Signal 5-7
  IOW Signal 5-7
  M1 Signal 5-6, 5-7, 8-69
  MEMR Signal 5-6, 5-7
  MEMW Signal 5-7
  OUT Signal 5-6
  READY Signal 5-4
  STACK Signal 5-6
  STSTB Signal 5-5, 8-67
  SYNC Signal 5-4
  WAIT Signal 5-4
  WO Signal 5-6
  WR Signal 5-4
Signed Binary Numbers 10-45
Signed Decimal Numbers 10-59
Signed Decimal Arithmetic Exercise 10-59
Sine C-1
SINWS Subroutine 9-52
Software 1-7
SOTBT Subroutine 9-47
Specification, Module 6-6
SPHL Instruction 6-116
STA Instruction 2-2, 2-28, 11-2

Stack 6-13, 6-99
Stack Operation Rules 6-119, 6-125
Stack, Monitor Usage of 6-120
Stack Pointer 6-13, 6-116
Stack Pointer Display 6-14
Stack Pointer Instructions 6-116
STACK Signal 5-6
Stack Top 6-15, 6-108
Stack, Using for Data 6-99
Start Bit 9-19, 9-33
Status Byte 5-5
STAX Instruction 4-60
STC Instruction 7-34
STEP Command 1-58, 2-29, A-11, A-23
Store Program on Tape A-6
Structure, Data 4-130
STSTB Signal 5-5, 8-67
SUB Instruction 4-18, 4-21, 4-47, 10-13
Subroutines 6-1, 6-12
  Alternate Entry 6-68, 6-70
  Classification 6-133
  Global 6-133
  Interrupt 6-134
  Local 6-134
  Monitor 6-136, A-35
    (see alphabetic listing, page i-17 of index)
  Reentrant 6-134
  Transparent 6-134
Subtraction 4-18
Subtrahend 10-16
SUI Instruction 4-29, 11-9
SYNC Signal 5-4
Synchronous Communication 9-16
System Controller 5-3, 5-9

Table Lookup 4-130, 6-73
Tape Programs and Subroutines 9-44, 9-47, 9-52
Tape Recording 9-44
Tens Complement 10-60
Timed Input/Output 8-76
Timed Interrupt Systems 8-76
Top Down Programming 6-33
Transfer Notation 4-43
Transmit/Receive with Monitor Subroutines 9-47
Transmitting, Asynchronous 9-20, 9-54
Transparency of Subroutine 6-134
Tri State Cicuits 5-26
Trigonometric Functions C-1
Twos Complement 10-45

Unbalanced Usage of the Stack 6-119
Undefined Instructions 11-16
Unpacked Decimal 10-59
Unsigned Integer 10-44

Vectored Interrupt Systems 8-75, 8-77

W, Z Registers 2-5
WAIT Signal 5-4
Wait State 5-32
WO Signal 5-6
Word 1-22
Word Mark in Serial Communication 9-14
WR Signal 5-4

XCHG Instruction 6-107
XRA Instruction 1-52, 2-28, 4-47, 7-33, 11-9
XRI Instruction 7-33
XTHL Instruction 6-108

Zero Flag 3-2, 11-10
ZILOG Z-80 Microprocessor 11-18

PROGRAM EXERCISES - LISTED IN SEQUENCE

Exercise 1	1-50
Exercise 2	2-1, 2-10
Exercise 3	2-16, 2-23
Addition by Counting	3-13
Double Precision Addition	4-11, 4-16
Review and Self Test	4-23
Multiplication by Repetitive Addition	4-34
Review and Self Test	4-48
Display Bit Pattern	4-56
Copy List to Display	4-63
Display Eight Characters	4-67
Delay Loops	4-73
Review and Self Test	4-84
Four Byte Addition	4-91
Counting in the Display	4-95
Review and Self Test	4-106
Moving Message	4-113
Do Nothing Program with PCHL	6-9
Nested Subroutines	6-24
Sensor Correction, Version 3	6-29
Input Subroutine	6-36, 6-44
Nextsensor Subroutine	6-54
Displayresult Subroutine	6-61
Searchdirectory Subroutine	6-64
Tablelookup Subroutine	6-73
Multiply Subroutine	6-79
Clear Result Display	6-97
Store and Recover Table Address	6-97
Two Byte Table Address	6-98
Empty Sensor Numbers	6-98
Testing Stack Usage	6-100
Test Driver for Multiply	6-110
Rotate (Arithmetic Shift)	7-3
Logical Rotate	7-19
Sixteen Bit Rotate	7-19
Binary Entry and Display	7-22
Logic Functions	7-35
Display	7-49
Data	7-52, 7-56
Command	7-60
Function	7-65
Logic Functions Self Test	7-69
Dispatch Table	7-75
Traffic Control	7-79
Extended Traffic Control	7-85
Fire and Burglar Alarm	7-88
Model Railroad Simulator	7-88

PROGRAM EXERCISES - Continued

Keyboard Input Exercise	8-15
Keyboard Display Exercise	8-26
External Interrupt Experiment	8-70
Effect of DI and EI	8-71
Interrupt Service Routine Exercise	8-81
Interrupt Service Testing	8-83
Interrupt Service - Main Program	8-91
Interrupt Service - Combined with Monitor	8-99
External Interrupt Exercise	8-100
Serial Transmission Exercise	9-21
Asynchronous Receiving Exercise	9-33
Transmit/Receive with Monitor Subroutines	9-47
Four Byte Addition Exercise	10-6
Four Byte Subtraction Exercise	10-16
Program Modification by Input Data	10-21
Decimal Addition and Subtraction	10-25
Experiment with DAA	10-31
Binary Multiplicaton Exercise	10-35
Binary Multiplication - Reversed	10-37
Change Sign Exercise	10-46
Change Sign by Complementing	10-49
Change Sign, Add, Subtract Exercise	10-53
Signed Decimal Arithmetic Exercise	10-59

MONITOR SUBROUTINES

BKENT Subroutine 9-49
BKLOC Subroutine 9-49
BKMEM Subroutine 9-51
BKRMV Subroutine 9-49
CLEAR Subroutine 6-147, A-53
CLRGT Subroutine 6-147, A-53
CLRHI Subroutine A-53
CLRLP Subroutine 6-147, A-53
DBYTE Subroutine 6-144, A-49
DELAY Subroutine 6-148, A-54
DELYA Subroutine 6-148, A-54
DELYC Subroutine A-55
DELYT Subroutine A-55
DIGHI Subroutine A-49
DIGSW Suroutine A-49
DISPR Subroutine 6-142, 8-92, A-48
DMEM Subroutine 6-144, A-49
DMWD Subroutine A-51
DWORD Subroutine 6-146, A-51
ENMEM Subroutine A-43
ENTBY Subroutine 6-140, A-41
ENTWD Subroutine 6-141, A-42
ERRDS Subroutine A-54
GETKY Subroutine 6-138, A-38
KEYS Subroutine A-39
MENAB Suroutine A-40
SCAN Subroutine 6-137, A-37
SHLRT Subroutine A-57
SINWS Subroutine 9-52
SOTBT Subroutine 9-47

MACHINE INSTRUCTIONS

ACI Instruction 4-28, 4-121, 11-9
ADC Instruction 4-12, 4-121, 10-4, 11-9
ADD Instruction 4-4, 4-46, 4-121, 11-9
ADI Instruction 2-1, 2-28, 4-29, 4-121
ANA Instruction 7-30, 11-9
ANI Instruction 7-33, 11-9
CALL Instruction 6-13, 6-16, 11-13
CC Intruction 6-52, 11-13
CM Instruction 10-51, 11-13
CMA Instruction 7-29, 11-7
CMC Instruction 7-34, 11-7
CMP Instruction 4-111, 11-9
CNC Instruction 6-52, 11-13
CNZ Instruction 6-52, 11-13
CP Instruction 10-51, 11-13
CPE Instruction 11-13
CPI Instruction 4-112, 11-9
CPO Instruction 11-13
CZ Instructiion 6-52, 11-13
DAA Instruction 10-25, 11-7
DAD Instruction 6-78, 10-5
DAD SP Instruction 6-116
DCR Instruction 3-13, 3-21, 4-121, 11-5
DCX Instruction 4-79, 4-120, 6-116, 11-5
DI Instruction 8-54, 8-71
EI Instruction 8-53, 8-71
HLT Instruction 8-65, 10-11
IN Instruction 8-6, 11-15
INR Instruction 1-52, 2-28, 4-121, 11-5
INX Instruction 4-71, 4-120, 6-116, 11-5
JC Instruction 4-40, 4-45, 11-13
JM Instruction 10-51, 11-13
JMP Instruction 2-20, 2-28, 11-13
JNC Instruction 4-40, 4-45, 11-13
JNZ Instruction 3-3, 3-22, 11-13
JP Instruction 10-51, 11-13
JPE Instruction 11-13
JPO Instruction 11-13
JZ Instruction 4-40, 4-45, 11-13
LDA Instruction 2-16, 2-28, 4-119, 11-2
LDAX Instruction 4-59, 4-119, 11-2
LHLD Instruction 4-97, 4-120, 11-2
LXI Instruction 4-69, 4-120, 11-3
LXI SP Instruction 6-116
MOV Instruction 4-2, 4-119, 11-2
MVI Instruction 4-25, 4-46, 4-119, 11-2
NOP Instruction 1-51, 11-15
ORA Instruction 7-32, 11-9
ORI Instruction 7-33, 11-9
OUT Instruction 8-6, 11-15

MACHINE INSTRUCTIONS - continued

PCHL Instruction 6-8, 11-3
POP Instruction 6-99, 6-105, 11-3
PUSH Instruction 6-99, 6-105, 11-3
RAL Instruction 7-1, 11-7
RAR Instruction 7-2, 11-7
RC Instruction 6-52, 11-13
RET Instruction 6-13, 6-16, 11-13
RLC Instruction 7-18, 11-7
RM Instruction 10-51, 11-13
RNC Instruction 6-52, 11-13
RNZ Instruction 6-52, 11-13
RP Instruction 10-51, 11-13
RPE Instruction 11-13
RPO Instruction 11-13
RRC Instruction 7-18, 11-7
RST 4 Instruction 3-9, 3-22, 4-47, 4-118, A-33, A-34
RST Instruction 3-9, 8-55, 11-14, A-33
RZ Instruction 6-52, 11-13
SBB Instruction 4-18, 4-21, 4-47, 4-121, 10-14, 11-9
SBI Instruction 4-29, 4-121, 11-9
SHLD Instruction 4-97, 4-120, 11-2
SPHL Instruction 6-116, 11-3
STA Instruction 2-2, 2-28, 4-119, 11-2
STAX Instruction 4-50, 4-119, 11-2
STC Instruction 7-34, 11-7
SUB Instruction 4-18, 4-21, 4-47, 4-121, 10-13, 11-9
SUI Instruction 4-29, 4-121, 11-9
Undefined Instructions 11-16
XCHG Instruction 6-107, 11-3
XRA Instruction 1-52, 2-28, 4-47, 7-33, 11-9
XRI Instruction 7-33, 11-9
XTHL Instruction 6-108, 11-3

INTEGRATED COMPUTER SYSTEMS

EDUCATION IS OUR BUSINESS™

NORTH AMERICAN HEADQUARTERS
Integrated Computer Systems, Inc.
3304 Pico Boulevard
P.O. Box 5339
Santa Monica, California 90405 USA
Telephone: (213) 450-2060
TWX: 910-343-6965

NORTH AMERICA – EASTERN REGION
Integrated Computer Systems, Inc.
300 North Washington Street
Suite 103
Alexandria, Virginia 22314 USA
Telephone: (703) 548-1333
TWX: 710-832-0045

EUROPEAN HEADQUARTERS
ICSP – U.K.
Pebblecoombe, Tadworth
Surrey KT20 7PA
England
Telephone: Leatherhead (03723) 79211
Telex: 915133

FRANCE
ICS France
90 Ave Albert 1er
92500 Rueil-Malmaison
France
Telephone: (01) 749 40 37
Telex: 204593

GERMANY
ICSD GmbH
Leonrodstrabe 54
8000 Munich 19
West Germany
Telephone: (089) 19 80 66
Telex: 5215508

SCANDINAVIA
ICSP Inc. – Scandinavia
Utbildningshuset AB
Box 1719
S-221 01 Lund, Sweden
Telephone: (046) 30 70 70
Telex: 33345