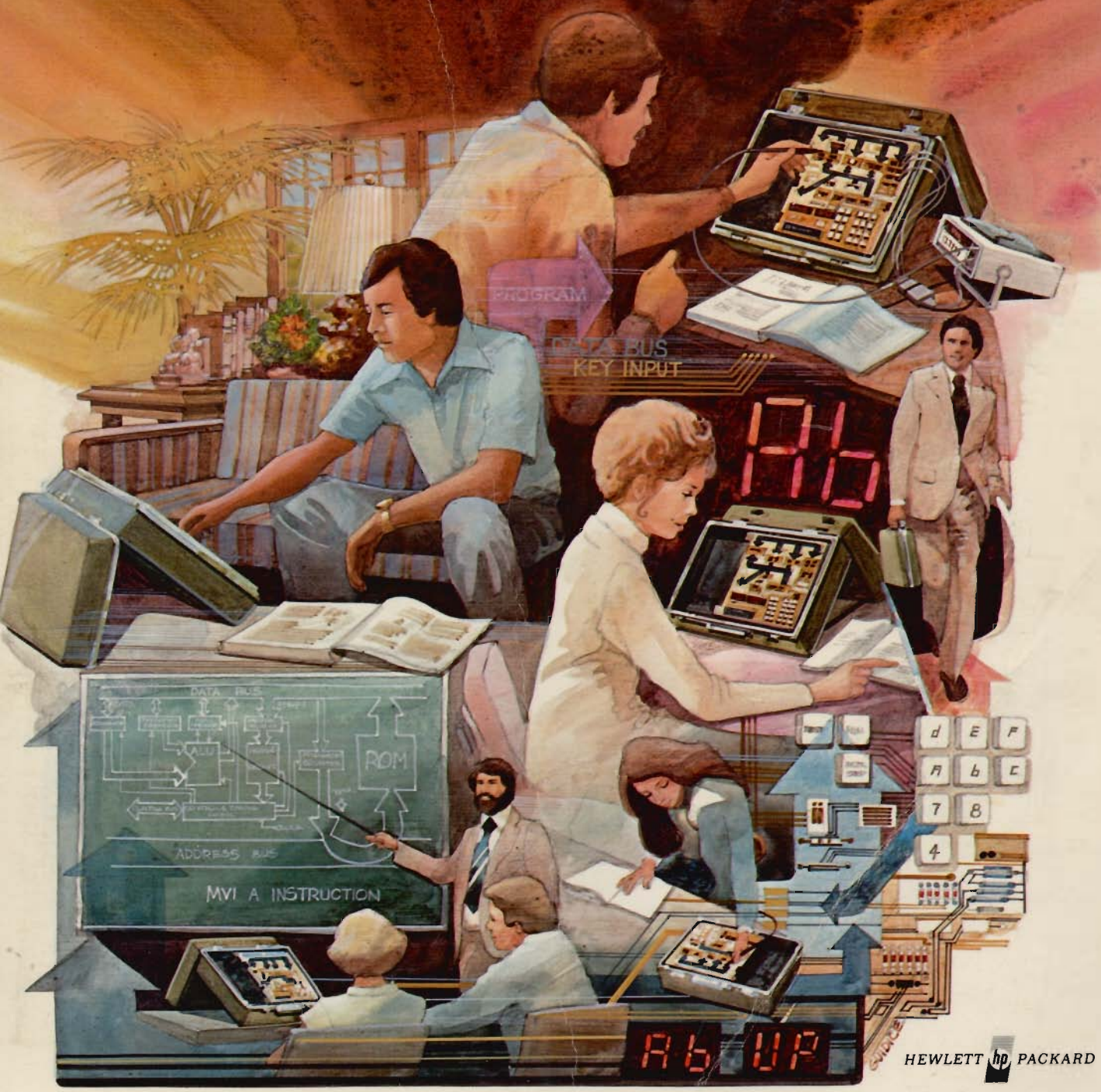
PRACTICAL MICROPROCESSORS



HEWLETT hp PACKARD

Hardware, Software, and Troubleshooting

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

PRACTICAL MICROPROCESSORS

Hardware, Software, and Troubleshooting

Michael Slater

Barry Bronson

First Printing — MARCH 1979
Second Edition, Revision — OCTOBER 1979
COPYRIGHT© 1979

by

HEWLETT-PACKARD COMPANY
5301 STEVENS CREEK BLVD.
SANTA CLARA, CALIF. 95050

TABLE OF CONTENTS

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

LIST OF FIGURES

LIST OF FIGURES

(Continued)

LIST OF FIGURES

(Continued)

LIST OF FIGURES

(Continued)

LIST OF FIGURES

(Continued)

LIST OF TABLES

LIST OF TABLES

(Continued)

PREFACE

PREFACE

INTRODUCTION TO THE MICROPROCESSOR LAB COURSE

You are about to invest approximately 50 hours of your time studying the HP5036A Microprocessor Lab Course. In return, you can expect to achieve the following objectives:

a. Acquire a practical knowledge of microprocessor system hardware.
b. Gain a basic understanding of the software that is used to control a microprocessor system.
c. Learn how the system uses this software to perform a wide variety of operations.
d. Use this information to learn practical troubleshooting techniques that are applicable to any microprocessor system.

One of the most significant benefits that you will derive from this course is the confidence that comes from understanding a total microprocessor system. This confidence removes the mystery associated with the concept of using programs to control the system hardware. This understanding will allow you to see how a tiny piece of silicon, called a microprocessor, controls the system. Once you have learned how the software and the microprocessor chip exercise this control, learning how any microprocessor system works is merely a matter of mastering the specific details of that system.

In addition, the new troubleshooting techniques that you will learn are also applicable to systems other than those based on the microprocessor. They can, in fact, be used to troubleshoot any type of digital equipment. Therefore, when you have completed this course, you should be able to understand and troubleshoot any digital system by applying the knowledge you have gained to any problem you may encounter.

WHAT THE MICROPROCESSOR LAB COURSE CONTAINS

Before beginning, a summary of the course contents and organization will help you understand the material that you will be studying. One of the most important characteristics of the course structure is that each section and lesson is modular. Since each student starts with a different background and set of objectives, this structure allows any of several methods of study to be used.

If you are studying this course in a formal class, your instructor will match the course materials to the class objectives. If you are using this material for self-study, read the following descriptions to learn what is covered in each lesson. Then, read the alternate methods of study that are recommended at the end of this introduction. By matching your background and objectives with these recommendations, you can obtain the maximum benefit from the specific material that you need to study.

The course is organized into six sections, each of which contains from one to five lessons, as required. Each lesson contains experiments and quizzes to reinforce the concepts presented and to show the practical applications of the information you have just learned.

SECTION I, MICROPROCESSOR FUNDAMENTALS, contains three lessons that provide a basic introduction to microprocessor systems.

Lesson 1, Introduction to Microprocessor Systems, supplies background information on the purposes, applications, and history of computing systems in general and microprocessor systems in particular.

Lesson 2, Number Systems, provides an introduction to number systems. The binary, octal, decimal, and hexadecimal number systems are covered.

Lesson 3, Software Fundamentals, introduces you to some of the basic concepts of programming. A brief explanation of the relationship between software and hardware is included.

SECTION II, INTRODUCTION TO PROGRAMMING, contains three lessons that provide an introduction to programming and instructions for using the Microprocessor Lab (μLab).

Lesson 4, Using the Microprocessor Lab, covers instructions for storing data, running programs on the μLab, and using the input/output ports.

Lesson 5, Software Concepts, discusses additional programming techniques and some of the more advanced features of the μLab.

Lesson 6, Inside the Microprocessor, takes a brief look inside the microprocessor to show how programs are executed. This insight will help you understand the system's operation.

SECTION III, MICROPROCESSOR SYSTEM HARDWARE, contains four lessons that describe microprocessor system hardware in detail.

Lesson 7, Basic Hardware Concepts, describes basic microprocessor system hardware. The emphasis is on helping you understand the fundamental parts of a typical system.

Lesson 8, Address Decoding, describes the features and characteristics of the μLab's address decoding circuits. Other types of address decoding circuits are also discussed.

Lesson 9, Memories and Peripherals, covers the memories and peripherals used in the μLab. In addition, it discusses other types of hardware that are used in a wide variety of systems.

Lesson 10, Control Circuits, deals with the control signals used in microprocessor systems. It covers the circuits that generate, transmit, and respond to these signals. Electrical circuit considerations are also presented.

SECTION IV, PROGRAMMING MICROPROCESSORS, contains five lessons that cover some of the more advanced concepts and techniques for programming microprocessors.

Lesson 11, Registers and Breakpoints, presents background material for the more detailed discussion of software that is contained in the rest of the section. The microprocessor's registers are described and the use of the breakpoint as a software debugging tool is covered.

Lesson 12, The 8085 Instruction Set, describes some additional instructions for programming the 8085. A representative group, including logical and arithmetic instructions, is presented.

Lesson 13, Software Design Techniques, introduces a set of techniques for designing a complex software system. Emphasis is placed on designing software that is not only efficient but also easy to debug and modify.

Lesson 14, Software Control of Peripherals, describes the software used to control the μLab's keyboard and display. The specific programs that are used to read from the keyboard and write to the display are described. Since most systems include a keyboard and a display, these concepts are applicable to a wide variety of microprocessor systems.

Lesson 15, Number Representations and Algorithms, discusses some of the techniques that are used to perform complex mathematical functions on a wide range of numbers.

SECTION V, TROUBLESHOOTING MICROPROCESSOR SYSTEMS, contains four lessons that deal with the theory of troubleshooting and the new tools and techniques that have been developed to troubleshoot microprocessor systems.

Lesson 16, Hand-Held Troubleshooting Tools, covers the use of Logic Probes, Logic Pulsers, and Current Tracers. These three instruments are easy to use and very effective in a broad range of digital troubleshooting situations. Practical examples show how they can be used, individually and together, to locate faults in microprocessor systems.

Lesson 17, Signature and Logic Analyzers, describes and shows how these two specialized instruments are used to troubleshoot microprocessor systems. Practical applications in field service, production, and product development are explained.

Lesson 18, Troubleshooting Microprocessor Systems, explains the troubleshooting philosophy and methods used for microprocessor systems. Specific problems and their solutions are discussed.

Lesson 19, Troubleshooting the Microprocessor Lab, describes the use of the troubleshooting flowchart for the μLab. First, a fault is found using this flowchart, and then you are shown how to insert and find various faults in the system.

SECTION VI, OTHER MICROPROCESSORS, contains only one lesson. It provides a survey of several currently available microprocessors.

Lesson 20, Microprocessor Survey, describes other types of microprocessors and demonstrates that although the details vary from one microprocessor to another, the concepts remain the same.

In addition to the information contained in the lessons, this course includes the following eight appendices found in Section VII which provide reference information for your use:

A. Solutions to Problems. Contains the answers to all the quiz questions and solutions to all the programming and troubleshooting problems.
B. 8085 Instruction Set Reference. Contains a complete set of 8085 instruction descriptions and hexadecimal codes.
C. Signature Tables. Contains troubleshooting signature information for all test points and IC pins in the μLab.
D. Reading Logic Diagrams. Contains a brief description of the significance of the logic level indicators used on logic symbols.
E. Demonstration and Utility Programs. Contains descriptions of these programs and directions for their use.
F. Monitor Listing. Contains a complete listing of the μLab's ROM contents.
G. Expanding the Microprocessor Lab. Contains a brief description of the signal inputs and outputs at the edge connector of the μLab, along with information on how to use these signals.

The reference material also includes an extensive glossary and bibliography that are oriented to microprocessor system requirements.

The last page of the manual is a foldout schematic of the μLab. The inside of the back cover contains the system error messages. The cover also folds out to present a quick reference for the μLab with brief definitions of the keys and switches.

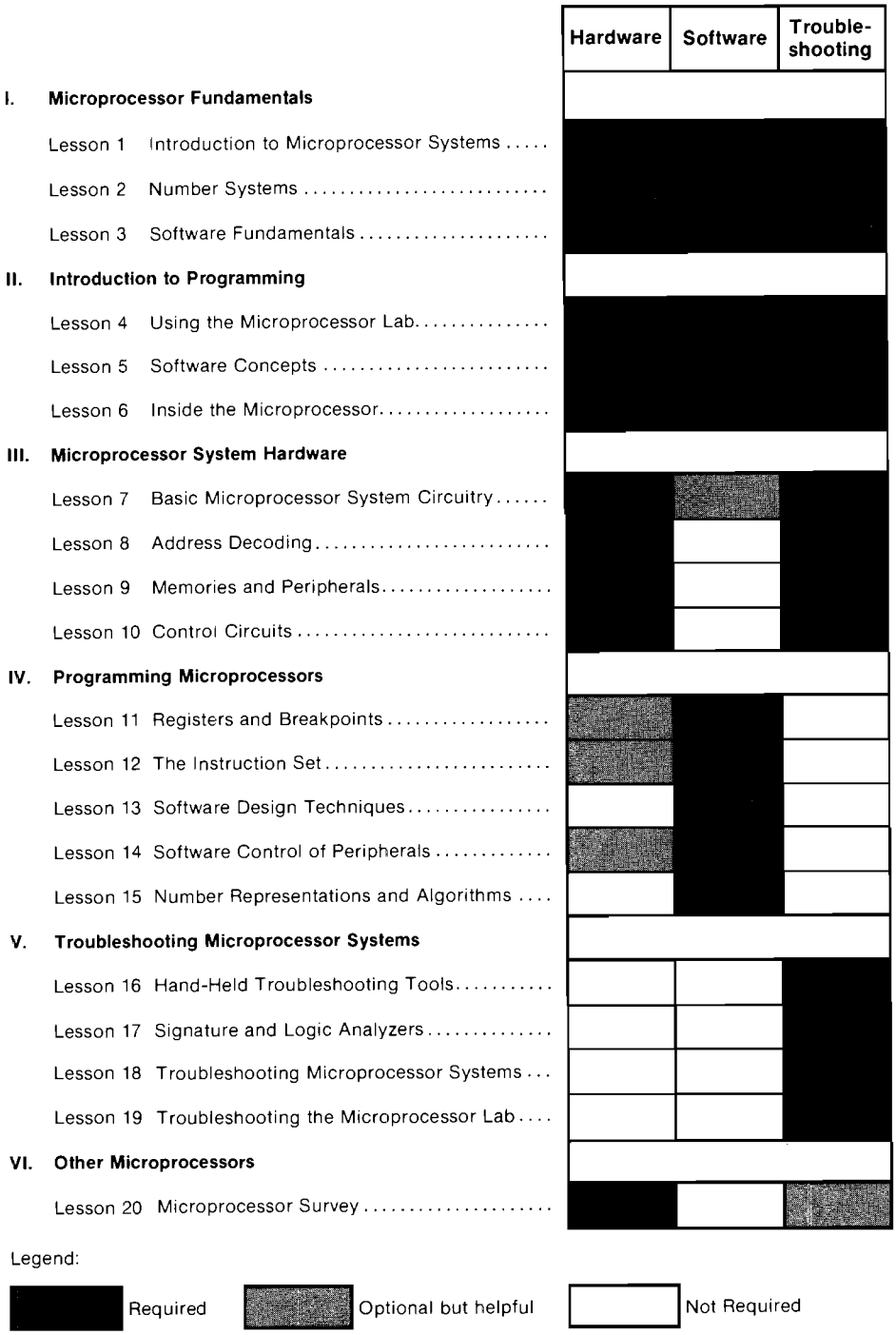
PREREQUISITES AND RECOMMENDED STUDY PROCEDURES

The minimum prerequisite for the μLab course is an understanding of basic digital circuits and symbology. If you need a complete review of this material, refer to HP's Practical Digital Electronics Course.

If you want an overall knowledge of software, hardware, and troubleshooting techniques, the best study procedure is to start at the beginning of the course and work each lesson sequentially. If you want to study only selected portions of the course, you can modify this sequence. The following figure shows which lessons should be studied for each of these three subjects:

a. Hardware
b. Software
c. Troubleshooting

The dark shading indicates mandatory lessons, and the light shading indicates optional lessons. For example, Lesson 7 provides an overview of microprocessor hardware. If you are only interested in learning software, studying Lesson 7 is not mandatory but it does provide excellent background information.

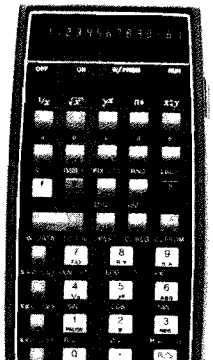


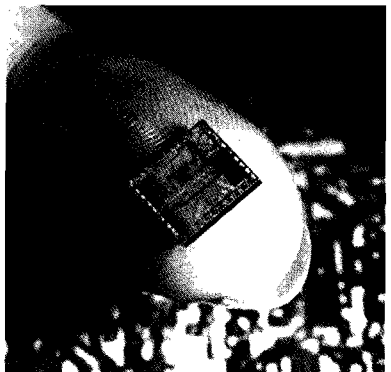
Microprocessor Lab Study Guide

In the 1960s smaller, more powerful computers were built using hundreds of gates, flip-flops, and other similar integrated circuits. These ICs are called *Small Scale Integration* (SSI) devices. As semiconductor technology developed, it became possible to put dozens of gates on a single IC. Examples of these *Medium Scale Integration* (MSI) ICs are counters, decoders, registers, and adders.

Programmable Desktop Calculator. *Introduced in 1968, it is more powerful than the much larger ENIAC and is built with discrete transistors.*

This miniaturization trend continued, and in 1971 the first microprocessor (the 4004) was introduced. Microprocessors contain the major computation and control sections of a computer, called the *Central Processing Unit* (CPU), on a single integrated circuit "chip." Microprocessors are often called *Microprocessing Units* (MPUs). A microprocessor chip contains thousands of gates and is called a *Large Scale Integration* (LSI) device. LSI memory devices were also developed that store thousands of bits of digital information on a single IC. These two LSI devices made it possible to drastically reduce the size and cost of small computers. Microprocessors have made it practical to build dedicated computers into many small, inexpensive products.



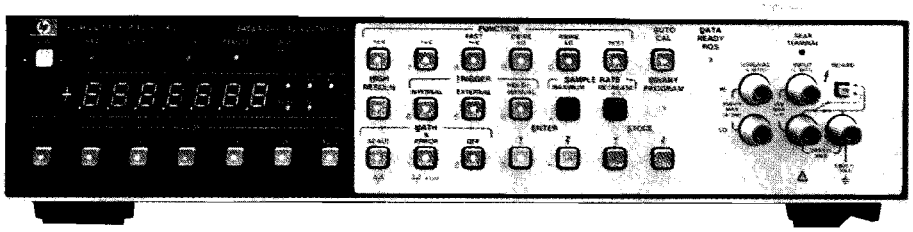


WHY MICROPROCESSORS ARE USED

Microprocessors are now being used in many products which were previously built with random logic. Microprocessor-based designs are usually less expensive and have many fewer components than the designs that they replace. Small microprocessor systems may be built with one or two ICs, at a cost of under ten dollars. These can often replace boards with dozens of simpler ICs. Because the number of discrete components and interconnections is greatly reduced, reliability is also improved.

This reduction in size is also possible using custom integrated circuits instead of microprocessors. However, the design of a custom IC can be an extremely complex and expensive process, often costing well over $100,000. This expense can be justified only for high volume products where the development cost can be spread over many thousands of units. The microprocessor allows standard ICs to be used to achieve the same miniaturization. The customizing takes the form of the program stored in the memory. Producing a standard memory with a custom program stored in it is a relatively inexpensive process.

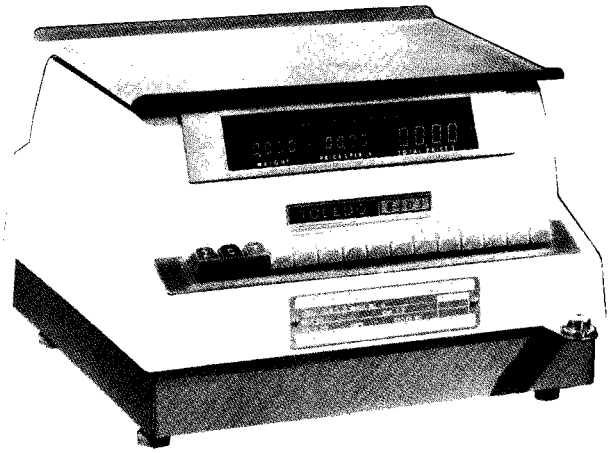
The flexibility and power of microprocessor-based systems makes many sophisticated features possible, which in the past were impractical. For example, microprocessor-based systems can often test themselves to a considerable extent and provide appropriate error messages. Instruments such as digital voltmeters can provide functions such as automatic averaging of a number of measurements, addition or subtraction of an offset value from each measurement, and self-calibration. The microprocessor also makes practical the use of a keyboard instead of multiposition switches on the front panel. Another feature is the capability for complete remote control.



Microprocessor-Based Digital Voltmeter. *The microprocessor provides functions such as self-calibration, averaging of a number of measurements, and automatic addition of an offset value.*

One of the most visible products to emerge from microprocessor technology is the electronic cash register or *Point of Sale* (POS) terminal. By replacing older electromechanical machines, reliability has been improved and many new features have become possible. By assigning each item to be sold an identification number, which the operator either types on the keyboard or reads from a bar code, inventory can be kept automatically. Sales tax can also be automatically added. It is even possible for the customer's bank account number to be entered, and the money automatically transferred from his bank account to the store's account.

Another example of the convenience which can be provided by microprocessor-based products is the computing scale. The operator enters the price per pound, and the scale weighs the item and displays the weight and the cost. The scale can also subtract the weight of the container.



Microprocessor-Based Scale. *The user enters the price per pound, and the scale computes the total price. The weight of the container can also be automatically subtracted. (Photo Courtesy Toledo Scale)*

All of these products may have been possible without microprocessors, but they would be so complex and expensive that they would be impractical. Microprocessor-based systems have these capabilities because so much complexity has been placed inside each IC. Product designers do not have to worry about the detailed construction of the ICs, and the size and reliability problems associated with complex systems are avoided. Furthermore, by placing the control in software, design changes are easy to make. The microprocessor has indeed revolutionized the design of many products.

A BASIC MICROPROCESSOR SYSTEM

Consider a system with a keyboard and a numeric display, as in a pocket calculator. When a key is pressed, the corresponding number should appear on the display. This system is a natural application for a microprocessor, and is in many ways similar to the Microprocessor Lab.

Figure 1-1 shows the block diagram of a system for doing this. The microprocessor (also called the *processor*) is the "brains" of the system. It contains all of the logic to recognize and execute the list of instructions *(program).* The *memory* stores the program, and may also store data. The fold-out inside the back cover shows these components on the Microprocessor Lab (μLab) board.

The microprocessor needs to exchange information with the keyboard and display. The *input port*, from which the processor can read data, connects the processor to the keyboard. The *output port*, to which the processor can send data, connects the processor to the display.

The blocks within the microcomputer are interconnected by three buses. A *bus* is a group of wires which connect the devices in the system in parallel. The microprocessor uses the *address bus* to select memory locations or input and output ports. You can think of the addresses as post office box numbers; they identify which locations to put information into or take information out of.

Once the microprocessor selects a particular location via the address bus, it transfers the data via the *data bus*. Information can travel from the processor to the memory or an output port, or from an input port or memory to the processor. Note that the microprocessor is involved in all data transfers. Data usually does not go directly from one port to another, or from the memory to a port.

The third bus is called the *control bus*. It is a group of signals which are used by the microprocessor to notify memory and I/O devices that it is ready to perform a data transfer. Some signals in the control bus allow I/O or memory devices to make special requests from the processor. The control bus is not apparent on the μLab board because it connects directly to the control logic, which generates control signals for each device in the system.

A single digit of binary information (1 or 0) is called a *bit* (a contraction of *binary digit*). One digital signal (high or low) carries one bit of information. Microprocessors handle data not as individual bits, but as groups of bits called *words*. The most common microprocessors today use eight-bit words, which are called *bytes*. These microprocessors are called eight-bit processors. For an eight-bit processor, *byte* and *word* are often used interchangeably. Be aware, however, that *word* is also used to mean a group of sixteen or more bits.

The address and data buses may also be seen on the μLab board. The data bus is a group of eight lines, and the address bus has sixteen lines.

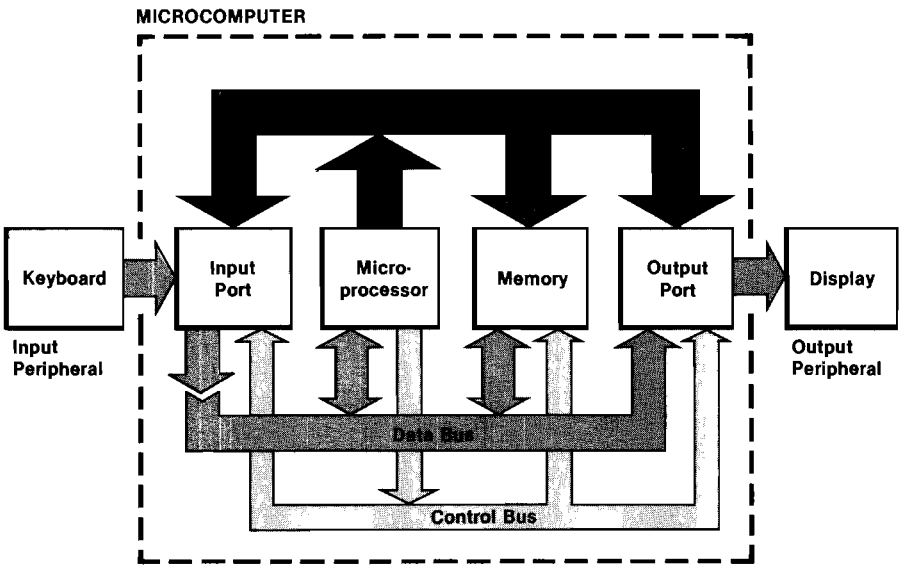


Figure 1-1. Basic Microprocessor System

PROGRAMS

To direct the system to perform the desired task, an appropriate list of instructions is required. For example:

1. Read data from the keyboard.
2. Write data to the display.
3. Repeat (go to step 1).

For the microprocessor to perform a task from a list of instructions, the instructions must be translated into a code that the microprocessor can understand. These codes are then stored in the system's memory. The microprocessor begins by reading the first coded instruction from the memory. The microprocessor decodes the meaning of the instruction and performs the indicated operation. The processor then reads the instruction from the next location in memory and performs the corresponding operation. This process is repeated, one memory location after another.

Certain instructions cause the microprocessor to *jump* out of sequence to another memory location for the next instruction. The program can therefore direct the microprocessor to return to a previous instruction in the program, creating a *loop* which is repeatedly executed. This enables operations which must be repeated many times to be performed by a relatively short program.

PERIPHERALS

A complete microprocessor system, including the microprocessor, memory, and input and output ports is called a *microcomputer*. The devices connected to the input and output ports (the keyboard and display for example) are called *peripherals*, or *Input/Output* (I/O) devices. The peripherals are the system's interface with the user. They may also connect the microcomputer to other equipment. Storage devices such as tape or disc drives are also referred to as peripherals.

An example of a microprocessor application from the instrumentation field is the microprocessor-based digital voltmeter (see Figure 1-2). Its input peripherals are an analog-to-digital converter and the range and function selector switches. The output peripheral is a digital display. The basic microcomputer is the same, whether the application is a calculator or a voltmeter; the difference is in the peripherals and the program.

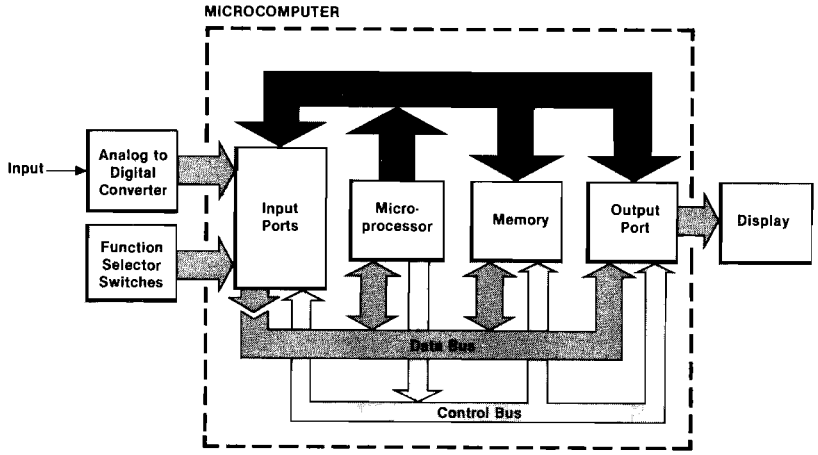
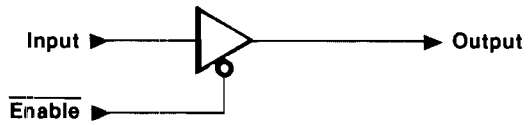


Figure 1-2. Microprocessor-Based Digital Voltmeter

THREE-STATE DRIVERS

All devices in the microprocessor system exchange information with the microprocessor over the *same* set of wires (the data bus). The microprocessor selects one device to place data on the data bus and disconnects the others. It is the *three-state* output capability of the devices on the bus that enables the processor to selectively turn devices on and off.



$\overline{\text{Enable}}$	Input	Output
0	0	0
0	1	1
1	0	floating
1	1	floating

0 = low 1 = high

Figure 1-3. Three-State Driver

Figure 1-3 shows the symbol and truth table for a three-state buffer (often called a *three-state driver*). The buffer has an output enable in addition to the usual input and output. When the enable is low, the buffer acts just as an ordinary buffer. The signal at the input is transferred to the output. When the enable is high, on the other hand, the output of the device is essentially disconnected.

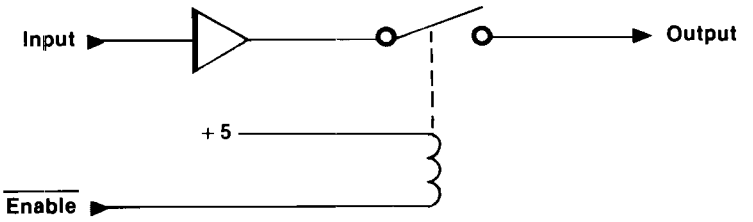


Figure 1-4. Conceptual Equivalent of Three-State Driver

Figure 1-4 shows a conceptual equivalent circuit which generates the open state using a relay. The disabled (open) output state is often called the *high impedance* state. Figure 1-5 shows a schematic of a typical three-state output.

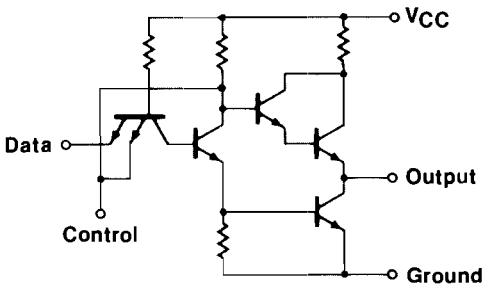


Figure 1-5. Schematic of Typical Three-State Output

Three-state drivers are important because they allow many devices to share a single data line. The circuit shown in Figure 1-6 allows any one of three different signals to drive one output. Only one driver's enable line may be low, and that device drives the output. If more than one driver were enabled, they would both try to drive the output. This condition is not allowed because the logic state of the output would be unpredictable.

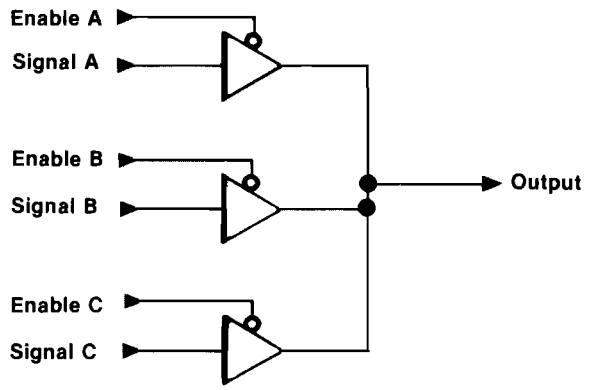


Figure 1-6. Circuit Showing Several Signals Sharing Single Data Line

Many devices, including microprocessors and memories, contain internal three-state drivers. These ICs have an output enable, often called *Chip Select* (CS) or *Chip Enable* (CE), which controls their output drivers.

Figure 1-7 shows how three-state drivers are used in microprocessor systems. All devices which put data on the data bus have three-state drivers on their outputs. The microprocessor generates control signals (part of the control bus) to enable the three-state drivers of the device from which it wants to read data. The three-state drivers of the other devices are disabled.

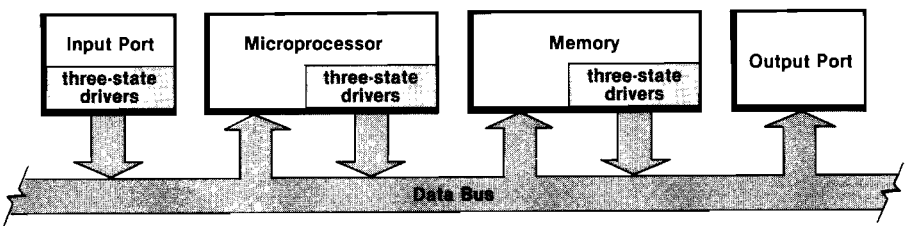
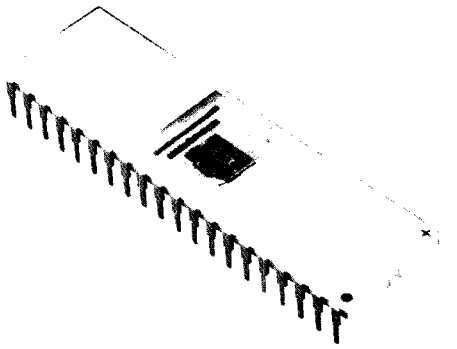


Figure 1-7. Three-State Drivers in Microprocessor System



8085 Microprocessor in Package with Lid Removed. *Forty thin wires connect pads on the chip to connection points on the package. (Photo Courtesy Intel Corp.)*

THE MICROPROCESSOR

Figure 1-8 shows the basic signals that connect to a typical microprocessor. There are sixteen address outputs which drive the address bus, and eight data pins which connect to the data bus. The data pins are *bidirectional*, which means that data may go into or out of them. $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ are the control signals that coordinate the movement of data on the data bus.

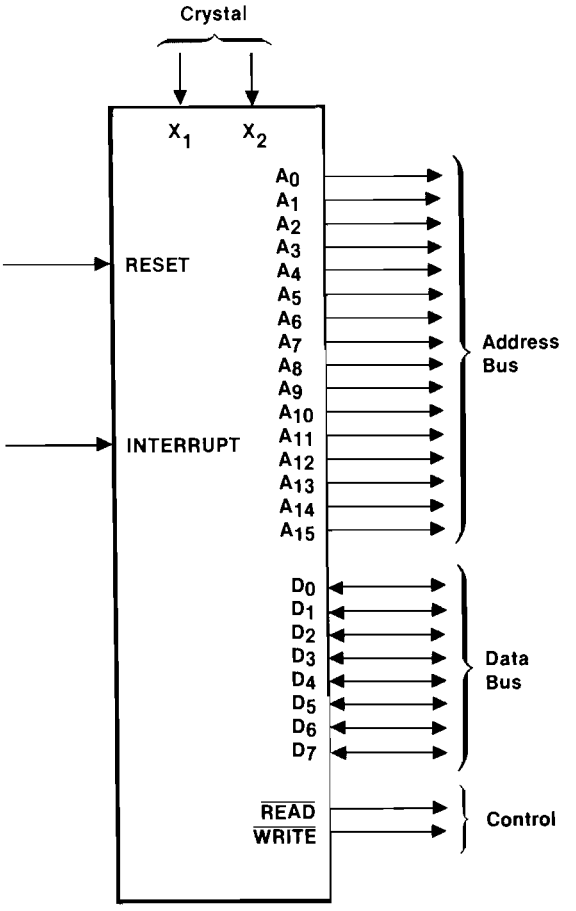


Figure 1-8. Basic Microprocessor Signals

The two signals shown on the left of the diagram provide additional control functions. The RESET input is used to initialize the microprocessor's internal circuitry. The INTERRUPT input allows the microprocessor to be diverted from its current task to another task which must be performed immediately. The use of these signals, plus others which have not been mentioned here, is described in section III, Microprocessor System Hardware.

The two connections at the top are for an external crystal, which is used to set the frequency of an oscillator in the microprocessor. The output of this oscillator is called the system *clock*. The clock synchronizes all devices in the system and sets the rate at which instructions are executed.

MEMORIES

Microprocessor systems usually use integrated circuit memories to store programs and data. They can store many bits of data in a single IC. Currently, devices are available with capacities of over 65,000 bits on one chip. A 65,536-bit memory can store over eight thousand alphanumeric characters, or about three pages of this text on a piece of silicon about a third of an inch square.

The simplest memory device is the *flip-flop*, which stores one bit of information. *Registers* contain up to eight flip-flops on a single IC, each with its own data in and data out pins but with a common clock line.

LSI technology made it possible to put thousands of flip-flops on a single IC, but a new problem was created. With thousands of flip-flops on an IC, there cannot be a separate data pin for each. The solution to this problem is to use address inputs to select the particular memory location (flip-flop) of interest. A *decoder* on the memory chip decodes the address and connects the selected memory location to the data pins.

Figure 1-9 shows a conceptual diagram of an eight-bit memory (most memories are much larger). Only the data output circuits are shown for simplicity. The decoder converts the binary address inputs to eight separate outputs, one for each possible combination of the three address lines. These signals control the three-state drivers at the output of each memory cell (flip-flop). The data from the addressed cell is placed on the data output line. This technique allows a single data pin to be used for all locations on the memory chip.

Each memory location can contain a group of bits rather than just one bit as in the example above. Each can hold one, four, or eight bits, depending upon the particular IC. If the IC has eight data pins, then each memory location stores eight bits of data. Note that while the memory may contain thousands of locations, only one may be accessed at a time.

The number of addressable locations depends upon the number of address lines. With one address line, two locations can be selected: address 0 and address 1. With two address lines, one of four locations can be selected: 00, 01, 10 and 11. The general rule is:

Number of locations = 2^N
Where N = number of address lines

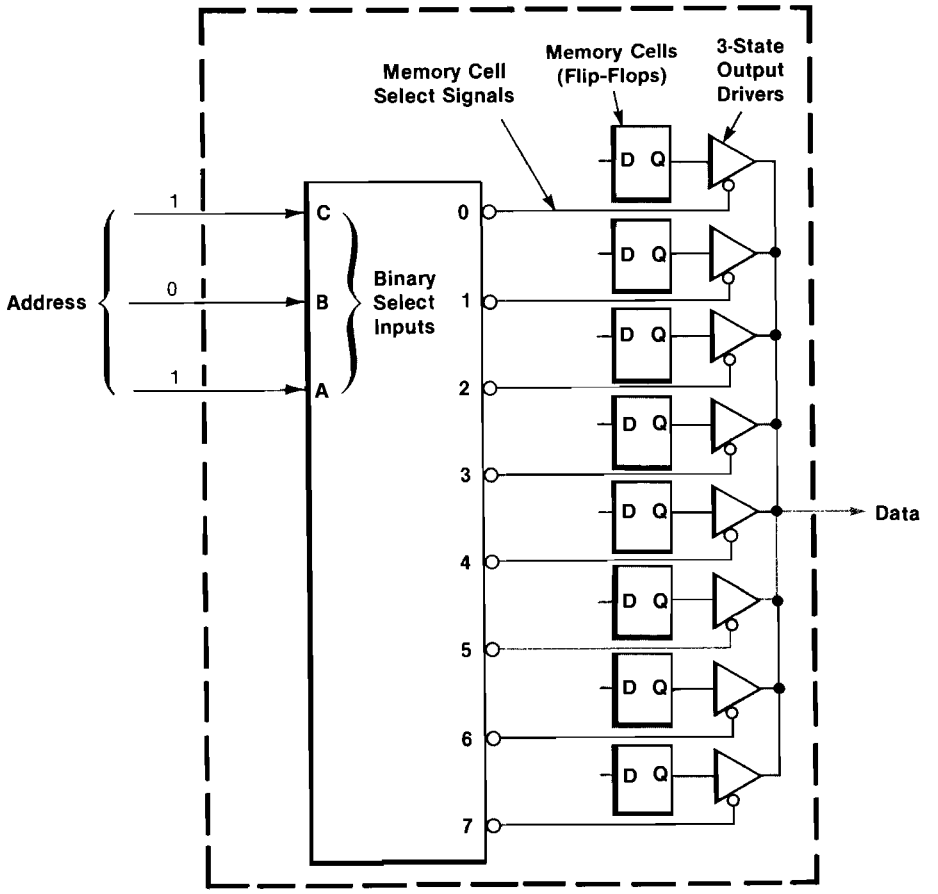
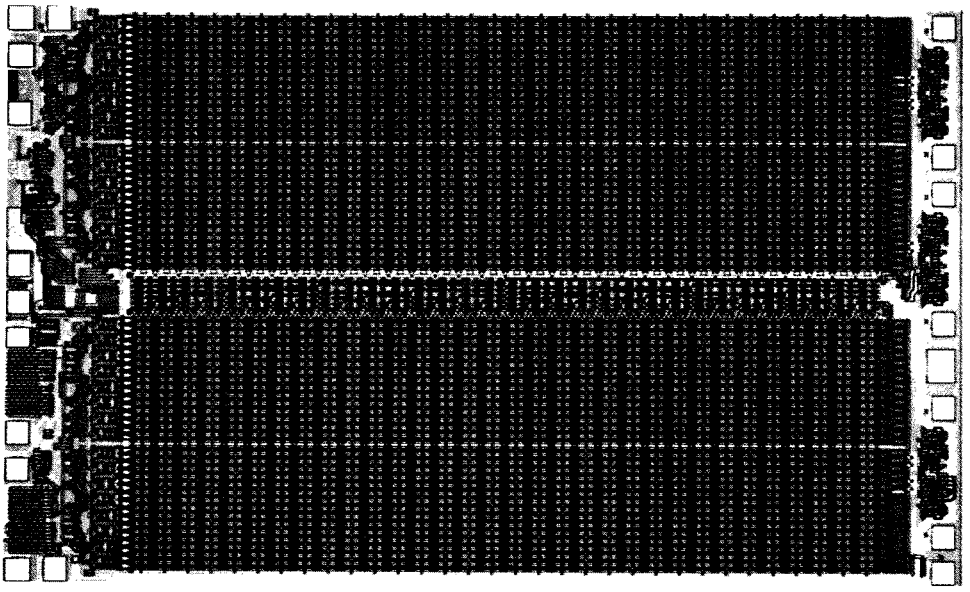


Figure 1-9. Conceptual Diagram of Eight-Bit Memory, Showing How All Memory Cells Share Single Data Line. Location 5 (101 binary) is shown selected.

ROMS AND RAMS

The memory ICs used with microprocessors fall into two broad categories: *ROMs* and *RAMs*. A ROM (Read Only Memory) is a memory which can only be read. The data is programmed into it at the time of manufacture, or by a special programming procedure prior to installation in the circuit. A program recorded into a ROM is often referred to as *firmware.*

A RAM (Random Access Memory) is a memory into which data can be stored and then retrieved. RAM is actually a misnomer; random access means that the time to access any memory location is the same, a characteristic also present in ROMs. Read/Write (R/W) memory is a more accurate term for what are usually called RAMs, but RAM is widely used to mean integrated circuit read/write memory. A digital tape recorder is an example of a memory which is not random access, since the time to access a particular location depends upon the position of the tape.



Photomicrograph of 4,096-Bit RAM. *The chip is 0.225 × 0.142 inches. (Photo Courtesy Intel Corp.)*

An important characteristic of semiconductor RAMs is that they are *volatile*: they lose their data when power is turned off, and when turned back on, they contain unknown data. ROMs do not have this problem, so they are used for permanent program and data storage. Since the contents of a ROM cannot be modified, RAMs must be used for temporary program and data storage.

Figure 1-10 shows a ROM containing 2,048 words of eight bits each, or 16,384 bits. When using large numbers that are powers of two, K is often used to mean 1,024 (2^{10}). Thus, this memory has 2K bytes or 16K bits. Since each location contains eight bits, it is called a 2K × 8 ROM.

When the Chip Select ($\overline{CS}$) input is low, the ROM's output drivers are enabled. When $\overline{CS}$ is high, the data outputs are in the high impedance state. The three-state outputs allow the data lines of many memory devices to be connected together, with one device selected by bringing its $\overline{CS}$ input low.

Figure 1-11 shows a 1K × 8 RAM. This RAM contains 1,024 locations of eight bits each. The data lines are bidirectional, since data can go into or out of the memory. RAMs have an additional control line called $\overline{WRITE}$. To store data in the RAM, an address is selected, the data is placed on the data lines, and the $\overline{WRITE}$ line is brought low. When the data and address are all set, the chip select is pulsed, and the data is stored in the memory.

The write line determines the direction of the data flow. The write line is usually active low, and is often called RD/$\overline{WR}$ (or R/$\overline{W}$). This notation indicates that if the signal is high, a read is performed, and if it is low, a write is performed. Note that this input has no effect unless the chip select is true.

ROMs and RAMs come in many different sizes (with different numbers of words and different numbers of bits per word) and many types. Lesson 9 contains additional information on ROMs and RAMs.

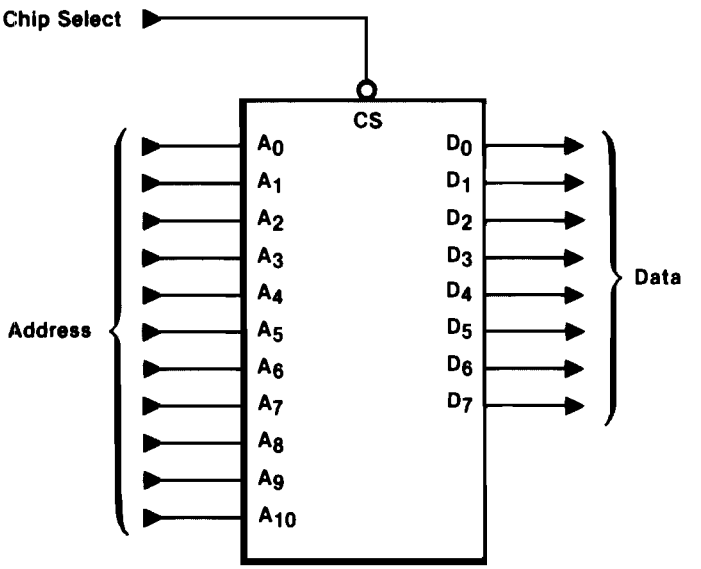


Figure 1-10. 2K × 8 ROM

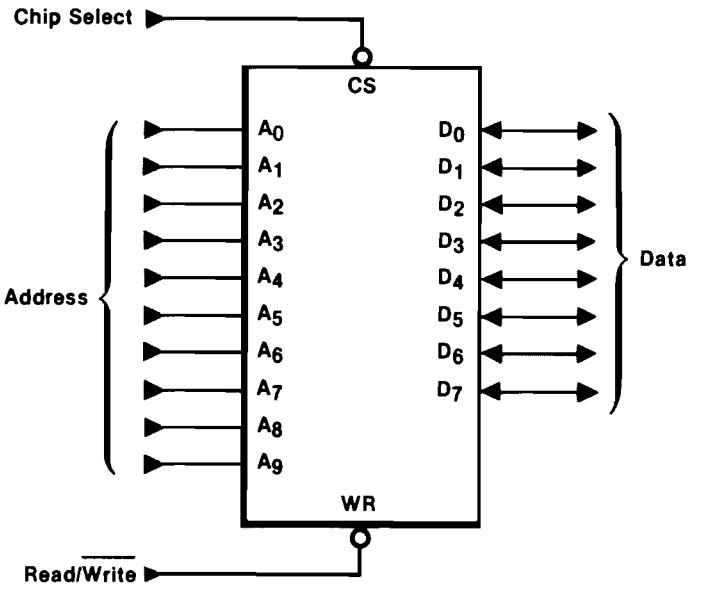
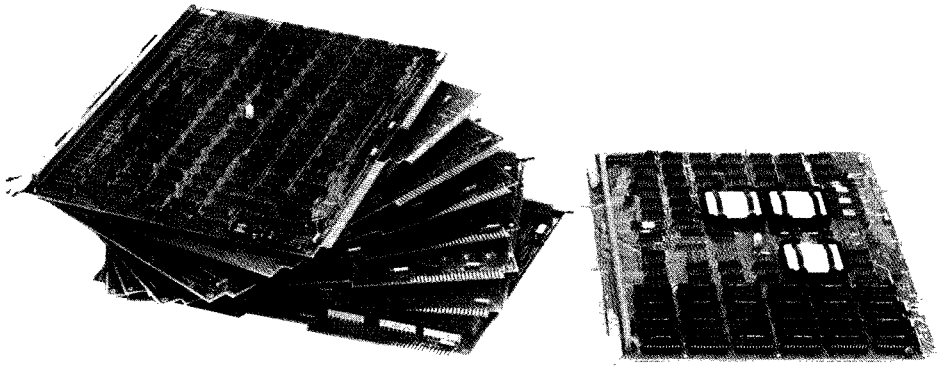


Figure 1-11. 1K × 8 RAM

MICROCOMPUTERS AND MINICOMPUTERS

A microcomputer is functionally similar to a minicomputer, and, in fact, the distinction between the two is becoming less and less clear. A microcomputer's CPU (Central Processing Unit) is the microprocessor. A minicomputer's CPU is usually a PC board with dozens of less complex, but faster integrated circuits on it. The main functional difference is that minicomputers are usually faster. They are also larger and more expensive. As microprocessors have increased in speed and power to compete with the older minicomputers, new minicomputers have been developed which are even faster and more powerful. These new minicomputers often use microprocessors internally. Thus, while the basic distinctions of speed, power, and size remain, the exact boundary is becoming vague. Microcomputers are now finding applications in systems where a minicomputer would be far too bulky and expensive.



One Board Replaces Eight. *On the left is the set of eight circuit boards which comprise the CPU for the HP 3000 Series II computer. The boards are built with standard small and medium scale TTL ICs. On the right is a complete CPU which performs exactly the same function with a single board. The three large squares are custom LSI ICs which make the size reduction possible. The single-board CPU is used in the Series 33 computer.*

EXPERIMENT 1-1

Introduction to the Microprocessor Lab

CONCEPT

This experiment introduces you to the μLab. Several demonstration programs which are stored in the ROM are run. A variety of tasks can be performed using the same hardware but different programs.

PROCEDURE

A) Open the μLab case and remove the lid. Turn the lid around so that the outside is facing you, and hook the hinges back together (see Figure 1-12). Fold the unit into an "A" shape, and connect the strap to the two snaps on the right-hand side of the case.

B) Connect the power cord. If the μLab has not been used before, check that the voltage selector switches are set correctly (see Figure 1-13).

C) Turn the μLab line switch to ON. The display and output LEDs light up for about a second, and then the speaker beeps. This indicates that the automatic power-up self-test has been successfully completed. The display shows μLAb UP indicating that the system is ready and waiting for a command. If at any point you press the wrong key and want to return to this state, just press [RESET].

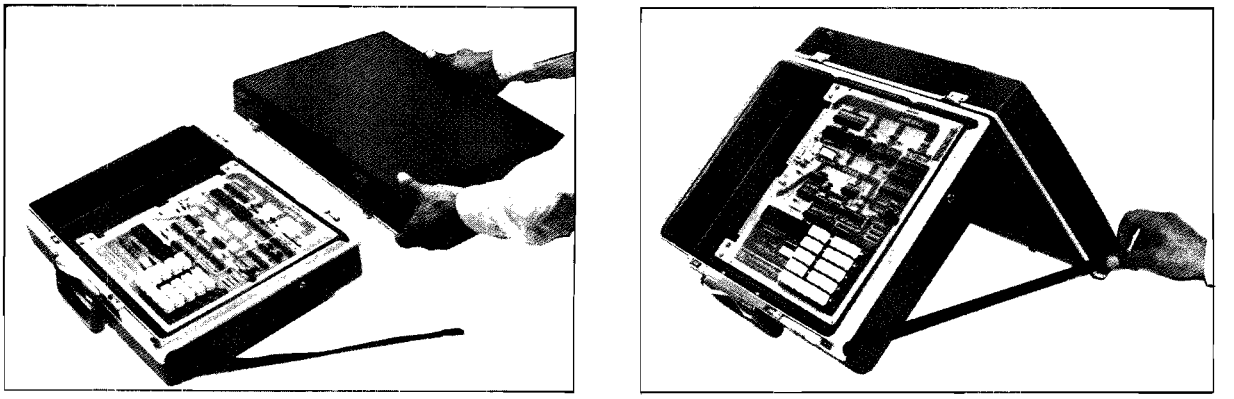


Figure 1-12. Assembling the Microprocessor Lab Case

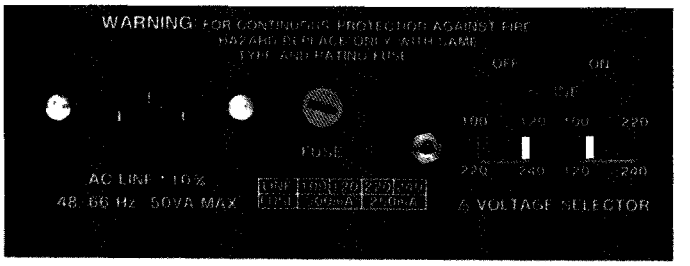


Figure 1-13. Setting the Line Voltage

EXPERIMENT 1-1

Continued

D) Press [FETCH ADRS]. The dashes on the display indicate that the μLab is waiting for you to enter the address.

E) Press [0] [5] [F] [9]. This number now appears in the left four digits of the display.

F) Press [RUN]. You have now started the "rocket blast-off" program stored in the μLab's ROM.

G) You will see some flashing lights and hear some noises, and then the program will stop. If you want to run it again, just press [RUN].

H) Press [FETCH ADRS] [0] [5] [3] [E] [RUN]. This is a "random" tone generation program.

I) Press [RESET] to halt the program.

J) Press [FETCH ADRS] [0] [5] [5] [A] [RUN]. This program creates a moving pattern on the display.

K) Press [RESET] to halt the program.

SUMMARY

In this experiment, you directed the μLab to execute several programs which are stored in the ROM. You entered an address that specified the starting location of the program. Pressing RUN caused the program to be executed. The "rocket" program has an instruction at the end which caused the microprocessor to return to the monitor program. The other two programs run continuously until you press RESET. These programs demonstrate the wide variety of functions that can be performed with the same hardware by using different programs.

These programs require a long series of relatively complex events. The microprocessor is well suited to these kinds of tasks because the control is provided by the software. Consider, on the other hand, a random logic circuit which could perform the "rocket blast-off." It would have to be very complex in order to generate the long sequence of events, including displaying numbers and generating tones. It would also probably require extensive circuit changes to perform, for example, the random tone generation. The microprocessor, however, allows all of this to be done with a general-purpose hardware system. Try running some of the other demonstration programs, which include a game, stopwatch, and organ program. These are described in Appendix E.

Lesson 1

Microprocessor systems are small computers which can exchange data with external devices. They have memories to store programs and data, and input and output ports to communicate with other devices. The basic system is called a microcomputer, and the devices connected to the I/O ports are called peripherals. The components are connected by address, data, and control buses.

ROMs are memories which are programmed once and cannot be modified afterwards. They are used for permanent program and data storage. RAMs are memories in which you can change the data, so they can be used for holding temporary data and programs. However, RAMs lose this information when power is removed.

Microcomputer circuits are called hardware. Program instructions, which are stored in the memory, are called software. When the software is permanently recorded into a ROM, it is called firmware.

A microprocessor can only read instructions from memory and execute them. The tremendous variety of tasks it can perform is a result of the infinite number of different instruction sequences it can execute. Most microcomputer applications consist of reading data from input ports, manipulating the data, making decisions based upon it, and writing data to output ports. All of this is controlled by the software. Software is much easier to change than hardware, and can perform very complex tasks. This makes microprocessor-based systems much more flexible and sophisticated than random logic systems.

QUIZ

Lesson 1

Note: Answers to the quizzes appear in Appendix A.

1. Microprocessor-based systems are more flexible than "random logic" designs because:

   a. they are faster.
   b. they use LSI devices.
   c. their operation is controlled by software.
   d. the hardware is specialized.

2. The "personality" of a microprocessor-based system is determined primarily by:

   a. the particular microprocessor used.
   b. the peripherals and the software.
   c. the number of lines in the data bus.
   d. the type of memory ICs used.

3. There are three groups of signals that interconnect the components of a microprocessor system: the _____ , _____ , and _____ buses.

4. Eight-bit microprocessors handle data in groups of bits called _____ .

5. The microprocessor system communicates with peripherals via:

   a. read only memories.
   b. input and output ports.
   c. the address bus.
   d. a keyboard.

6. The peripherals are:

   a. the memory devices.
   b. the microprocessor.
   c. the software.
   d. the I/O devices.

7. Three-state drivers are useful because:

   a. they allow many devices to be easily connected together.
   b. they have low power consumption.
   c. they are inexpensive.
   d. they provide three valid logic levels.

8. ROMs are used primarily for:

   a. data storage.
   b. temporary program and data storage.
   c. making things hard to change.
   d. permanent program and data storage.

9. RAMs are not usually used for long-term storage because:

   a. they lose their contents when power is removed.
   b. they are slow.
   c. they are expensive.
   d. the contents cannot be modified.

LESSON 2

Number Systems

INTRODUCTION

In this lesson, the decimal, binary, octal, and hexadecimal number systems are discussed. People work naturally in decimal, but digital computers work in binary. Octal and hexadecimal provide representations of binary that are easier to use. The Microprocessor Lab uses hexadecimal to represent binary programs and data.

DECIMAL AND BINARY

The number system most people are accustomed to is the *decimal,* or base 10, system. Base 10 means that with one digit, ten different numbers can be represented: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Digital computers use the *binary,* or base 2, number system. With one digit two different numbers can be represented: 0 and 1. This is the natural number system for digital systems because a digital signal has only two states, low and high, which can be interpreted as 0 and 1.

Figure 2-1 compares the decimal and binary number systems. In the decimal system, the digit positions have weights of $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, and so on. In the binary system the weights are $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and so on.

DECIMAL

Weight of each digit position	10^4	10^3	10^2	10^1	10^0
Example	1	9	6	0	1

$19{,}601 = 1\times10^4 + 9\times10^3 + 6\times10^2 + 0\times10^1 + 1\times10^0$
$= 10{,}000 + 9{,}000 + 600 + 1$

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1{,}000$
$10^4 = 10{,}000$

BINARY

2^4	2^3	2^2	2^1	2^0
1	0	1	1	0

$10110 = 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0$
$= 16 + 4 + 2$

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$

Figure 2-1. Binary and Decimal Number Systems

To convert from binary to decimal, the decimal values of the columns are simply added together. For example:

$11001 = (1 \times 2^0) + (0 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) + (1 \times 2^4) = 1 + 0 + 0 + 8 + 16 = 25.$

Table 2-1 shows the binary equivalents of the decimal numbers 0 through 31, along with the hexadecimal and octal representations.

Binary	Decimal	Octal	Hex
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	8	10	8
01001	9	11	9
01010	10	12	A
01011	11	13	B
01100	12	14	C
01101	13	15	D
01110	14	16	E
01111	15	17	F
10000	16	20	10
10001	17	21	11
10010	18	22	12
10011	19	23	13
10100	20	24	14
10101	21	25	15
10110	22	26	16
10111	23	27	17
11000	24	30	18
11001	25	31	19
11010	26	32	1A
11011	27	33	1B
11100	28	34	1C
11101	29	35	1D
11110	30	36	1E
11111	31	37	1F

Table 2-1. Number Systems

Figure 2-2 shows a technique for converting from decimal to binary. First the decimal number is divided by two. The remainder, which in this example is 1, becomes the least-significant (right-most) bit of the result. The result of the division (6 in this example) is then divided by two again. The remainder is the second bit of the result. This process is continued, with the remainder of each division contributing one bit to the binary number. When the result of the division is zero, the process is complete.

OCTAL

One problem with using binary numbers is that they contain more digits than the decimal equivalent and are therefore more difficult to deal with. For example, when copying the number "10111001," it is easy to make a mistake and write "10110001."

One way to reduce this problem is to use a more compact representation for the binary numbers. Decimal could be used, but the conversion between decimal and binary is awkward. Figure 2-3 shows the representation called *octal,* or base 8. The binary number is divided into groups of three bits starting at the right.

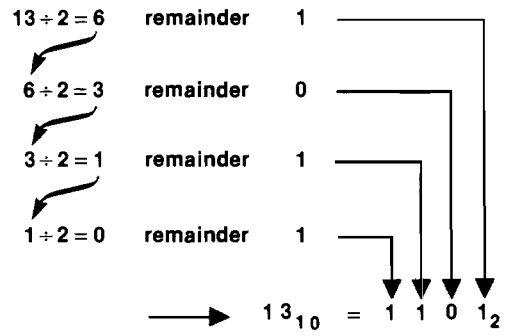


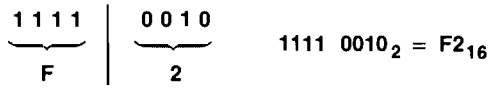
Figure 2-2. Decimal to Binary Conversion

Each group of three is replaced by its octal equivalent. The binary number 101 001 can therefore be represented by the octal number 51. Note that while octal numbers look like decimal numbers, they are not. The decimal equivalent of 101001 is $2^0 + 2^3 + 2^5 = 1 + 8 + 32 = 41$. It is much easier to convert back and forth between binary and octal than between binary and decimal.

Figure 2-3. Octal Number Representation

HEXADECIMAL

Another convenient representation for binary numbers is the *hexadecimal*, or base 16, number system. Each group of four bits is replaced by a single character (see Figure 2-4). Since four bits can have decimal values from 0 to 15, a way is needed to represent the decimal values 10 through 15 with a single character. The letters A through F are used for this purpose. In hexadecimal (as shown in Table 2-1), you count 0, 1, 2 . . . 8, 9, A, B, C, D, E, F, 10, 11, ...



Binary	Hex	Display
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	A
1011	B	B
1100	C	C
1101	D	D
1110	E	E
1111	F	F

Figure 2-4. Hexadecimal Number Representation

It is quite easy to convert between binary and hexadecimal (often abbreviated hex). The binary number is divided into groups of four bits, and each group is converted to the appropriate character. To go the other direction, each hex character is replaced by four bits. Once you are accustomed to using A for 10, B for 11, etc., you will find it very easy to do this conversion. A binary-to-hex table is printed in the upper right-hand corner of the μLab PC board.

Hexadecimal, the most widely used system for representing binary numbers, is used in this course. It is preferred over octal because an eight-bit binary number can be represented with two hex characters, but it takes three octal characters to do the same thing. The hex representation is therefore more compact.

BIT POSITION TERMINOLOGY

When referring to binary numbers, it is often necessary to refer to a particular bit or group of bits. The right-most bit is called the *Least Significant Bit* (LSB), and the left-most bit is called the *Most Significant Bit* (MSB).

When referring to a group of bits at one end of a word, the left-hand bits are called the *high-order* (or most significant) bits, and the right-hand bits are called the *low-order* (or least significant) bits.

REVIEW

The binary number system, which is used by all digital computers, is an awkward system for people to use. The numbers are long, and it is easy to make mistakes in copying a number. Decimal is the easiest system for people to use, but the conversion between decimal and binary is difficult. Octal and hexadecimal provide representations similar to decimal, but the conversion to binary is much simpler. In this course the hexadecimal system is used.

This lesson has considered only positive integer numbers. Lesson 15 describes ways of representing negative numbers, fractions, and very large or small numbers.

QUIZ

Lesson 2

1. A binary digit is either a _____ or a _____ .

2. The decimal equivalent of 10101100 is _____ .

3. The octal equivalent of 10101100 is _____ .

4. The hex equivalent of 10101100 is _____ .

5. The binary equivalent of B9 (hex) is _____ .

6. 11111111 (binary) = 255 (decimal) = _____ (hex) = _____ (octal).

7. Hexadecimal is preferred over decimal when working with binary numbers because:
   a. hex is easier for people to use.
   b. hex gives a less compact representation.
   c. hex has a simpler relationship to binary.
   d. decimal has a simpler relationship to binary.

LESSON 3

Software Fundamentals

INTRODUCTION

As described in the preceeding lessons, the microcomputer system consists of both hardware and software. The software (or programming) aspects of the system are discussed in this lesson.

Programs are first written in a way that is convenient for the person writing the program (the *programmer*). The program must then be rewritten and stored in the code that the microprocessor understands. The microprocessor then reads the codes from memory, one at a time, and performs the indicated operations.

It is important to understand what a microcomputer cannot do, as well as what it can do. The first part of this lesson discusses these limitations.

COMPUTERS DON'T THINK

When writing a computer program, the programmer must tell the computer what to do down to the most minute detail. Computers are not very smart when compared with people, as shown in Figure 3-1. An electric light is unintelligent: you turn on the switch and it goes on. A toaster is slightly smarter in that it ejects the toast when it is done. The smartest machine is the computer, which represents the ultimate in mechanized intelligence. Yet even a simple earthworm is more intelligent than a computer.

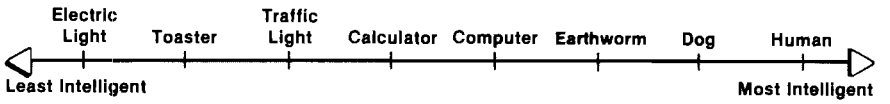


Figure 3-1. Levels of Intelligence

It is important to distinguish between intelligence and computational power. Computers can act with tremendous precision and speed for long periods of time, but they must be told exactly what to do. A computer can respond to a change in conditions, but only if it contains a program that says "If this condition exists, do this." A worm, on the other hand, operates not from a list of instructions, but by some general strategy for dealing with new situations. It possesses, even if only to a minor degree, those mysterious qualities that account for creativity and learning. Computers, on the other hand, operate in an entirely logical, predetermined manner.

The important point is that while computers are precise and fast, they are not creative. They can deal with contingencies, but only in ways for which they have been programmed. The apparent intelligence of computers is a function of the large number of programs that they contain. The use of a microprocessor allows some of this simulated intelligence to be incorporated into a product. On the other hand, this type of sophistication is very difficult to obtain using random logic.

THE MICROCOMPUTER AS A LOGIC DEVICE

Microprocessor systems are often used to replace circuits composed of standard logic devices. In order to illustrate the differences between a "programmed" logic device and a traditional logic device, consider using a microprocessor as a simple AND gate.

A microprocessor-based AND gate requires an input port for the gate's inputs and an output port for its output (see Figure 3-2). The microprocessor, using instructions stored in the memory, performs the AND function. Since an AND gate has only one output, only one bit of the output port is needed.

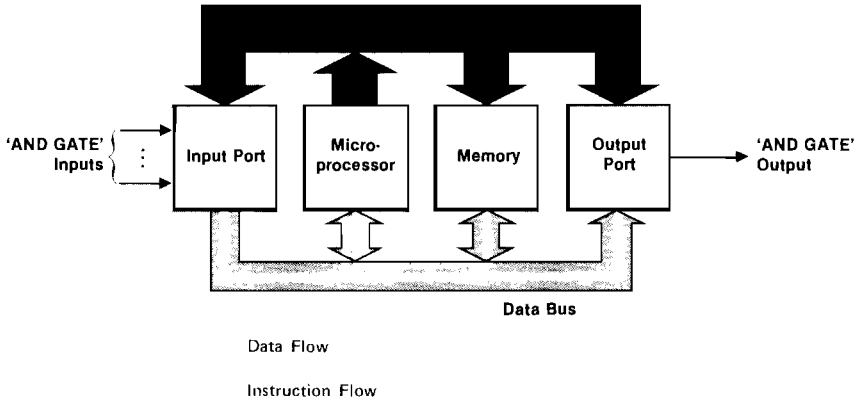


Figure 3-2. Microprocessor-Based AND Gate

An appropriate program is required for the processor-based AND gate. The following is a list of instructions that perform the AND gate function:

1. Read the input port.
2. Go to step 5 if all inputs are high; otherwise continue.
3. Set output low.
4. Go to step 1.
5. Set output high.
6. Go to step 1.

First, the input port is read. Then the inputs are examined to see if they are all high, since that is the function of an AND gate. If the inputs are all high, the output is set high; otherwise, it is set low. Once the procedure has been completed, the program jumps back to step 1 and repeats indefinitely, so the output continuously follows changes in the inputs.

FLOWCHARTS

Flowcharts are a graphic way of describing the operation of a program. They are composed of different types of blocks interconnected with lines. There are three principal types of blocks used for flowcharts, as shown in Figure 3-3. A rectangular block describes each action the program takes. A diamond-shaped block is used for each decision, such as testing the value of a variable. An oval marks the beginning of the flowchart, with the name of the program placed inside it. An oval can also be used to mark the end of the flowchart. There are many other specialized flowcharting symbols, but they will not be used in this course.

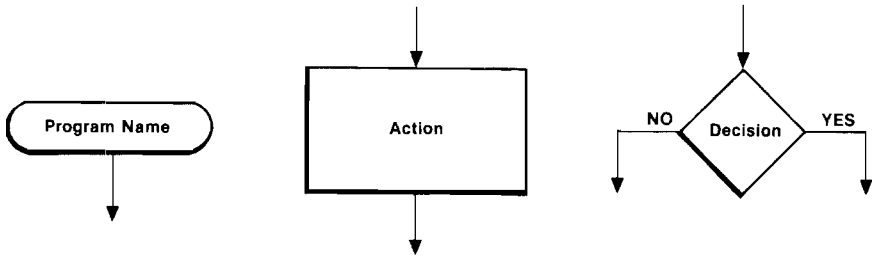


Figure 3-3. Flowcharting Symbols

Figure 3-4 shows a flowchart for the AND gate. For each line of the program there is a block, except for the two "Go to" instructions. These are represented simply by a line. The lines show the flow of the program from one block to another.

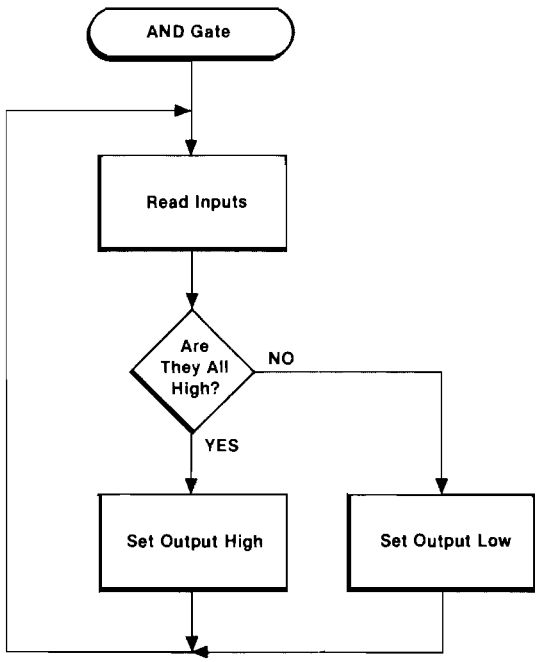


Figure 3-4. AND Gate Flowchart

While the flowchart contains the same information as the program list, it is in a more graphic form. When you first set out to write a program, a flowchart is a good way to organize your thoughts and document what the program must do. By going through the flowchart "by hand," you can check the logic. Then you can write the actual program from the flowchart. Flowcharts are also useful for going back to a program that has been written in the past and figuring out what the program does.

EXPERIMENT 3-1

The Microprocessor Lab as an AND Gate

CONCEPT

The μLab's ROM contains a program to perform the AND gate function just described. In this experiment you will run the program and observe its operation.

PROCEDURE

A) Press [RESET]. This forces the μLab to its normal "waiting for command" mode and is unnecessary if the display already shows ␣LAb UP.

B) Press [FETCH ADRS] [0] [4] [E] [0]. This specifies the starting address of the program.

C) Press [RUN]. The AND gate program is now running (the display is blank).

D) Find the eight-position switch labeled INPUT near the bottom of the board. These switches are connected to the input port and are the inputs to the AND gate.

E) Notice the row of LEDs, labeled OUTPUT LEDS, connected to the output port. The right-most LED is used for the output of the AND gate.

F) Set all the input switches to the *up* position. This puts a high level at each bit of the input port. The right-most output LED should now be on, since all the inputs are high. This LED connects to bit 0 of the output port, which is used as the AND gate's output.

G) Change the setting of any of the input switches. The output LED should go off, since the AND gate no longer has the required inputs for a high output.

H) Press [RESET]. Set the input switches to *up* again. Notice that the output LED no longer responds, since the program was stopped when [RESET] was pressed.

SUMMARY

The μLab was used to perform the function of an AND gate, using a program permanently stored in the μLab's ROM. The switches provided data to the input port, which was read by the microprocessor. The processor then decided if the "AND" condition was satisfied and set the output port to the appropriate level. The LED indicated the state of the output port.

Note that the μLab does not contain a special piece of hardware to act as an AND gate. Instead, it uses its general logical abilities to perform this function.

CHARACTERISTICS OF THE MICROPROCESSOR-BASED AND GATE

The microprocessor system can perform exactly the function of a normal AND gate, except that it is slow. It may appear instantaneous to you, but when compared to digital logic speeds, it is very slow. The AND gate program takes six instructions, and it takes the μLab approximately 2 microseconds to execute an instruction. Therefore, the microcomputer AND gate has a *propogation delay time* (the time between a change in the input and a change in the output) of about 2 microseconds × 6 = 12 microseconds. A standard TTL AND gate has a delay time of approximately 10 nanoseconds—over a thousand times faster! Even this simple example illustrates the slow speed of the microprocessor as compared to traditional logic. The difference is even greater for a more complex system. But in many applications this is of no consequence, because the speed of traditional logic is actually much more than needed. Consider, for example, an operator typing at a keyboard. If the operator types at the rather incredible speed of a hundred keys per second (over 1000 words per minute), the processor has 10 milliseconds to respond to each keystroke—enough time for the microprocessor to execute about 5,000 instructions.

You may be wondering why this complex system is used when a simple AND gate is sufficient. Indeed, if this is the only function a system is required to perform, an AND gate would be used. However, the microprocessor provides tremendous flexibility. It allows the function of the gate to be arbitrarily redefined just by changing the program. You could easily add more inputs, and the gate function could be extremely complex. Consider, for example, an eight-input logic device that functions as an electronic lock. The output goes on only if the inputs are turned on in a specific order. This requires a complex circuit using traditional logic, but it could be implemented using exactly the same microprocessor system as used for the simple AND gate. Of course, a new program is required, more complex than the simple AND gate program, but the hardware (except for the I/O devices) does not change. Additionally, the "combination" to this electronic lock could be changed just by modifying the program.

PROGRAMMING LANGUAGES

Writing programs in English is convenient since it is the language most people understand, but, unfortunately, it is meaningless to a microprocessor. The language understood by the microprocessor is called *machine language* (often referred to as *machine code*). Since microprocessors deal directly only with digital signals, machine language instructions are binary codes (e.g., "00111100"). The microprocessor is designed to recognize a specific group of codes called the *instruction set*.

Machine language is not easy for people to use, since "00111100" has no obvious meaning. It can be made easier to work with by using the hexadecimal representation: "0011 1100" is replaced by "3C." However, this still does not provide any clue to the meaning of the instruction.

The next step is to replace each instruction code with a short name called a *mnemonic*. The code "3C," for example, which for the 8085 microprocessor means "increment the A register," is represented by "INR A." The mnemonics are much easier to remember than the machine codes. By assigning a mnemonic to each instruction code, you can write programs using mnemonics instead of codes. The mnemonics can easily be converted to machine codes after the program is written. Therefore, you do not need to remember the machine codes, and the meaning of each instruction is easier to remember. Programs written using mnemonics are called *assembly language* programs.

The machine language is generally determined by the design of the microprocessor chip and cannot be modified. The assembly language mnemonics, however, are made up by the microprocessor's manufacturer as a convenience for programmers, not set by the processor design. For example, you could write INC A instead of INR A, as long as both were translated to the machine code 3C. In this course, Intel's 8085 assembly language conventions are followed.

While assembly language is a vast improvement over machine language, it is still difficult to use for writing complex programs. To make programming easier, *high-level languages* have been developed. These are similar to English and are generally independent of any particular microprocessor. A typical instruction might be "LET COUNT = 10" or "PRINT COUNT."

These instructions give a much more complicated command than those that the microprocessor can understand. Therefore, microcomputers on which high-level languages are used also contain long, complex programs (permanently stored in their memory) that translate the high-level language program into a machine language program. A single high-level instruction may translate into dozens of machine language instructions. Such translator programs are called *compilers*.

A PROGRAMMING EXAMPLE

A simple programming example will serve to illustrate these concepts. Figure 3-5 shows the flowchart for a program that counts to ten. There is no input or output in this program: the contents of a designated memory location simply count from zero to ten and repeat.

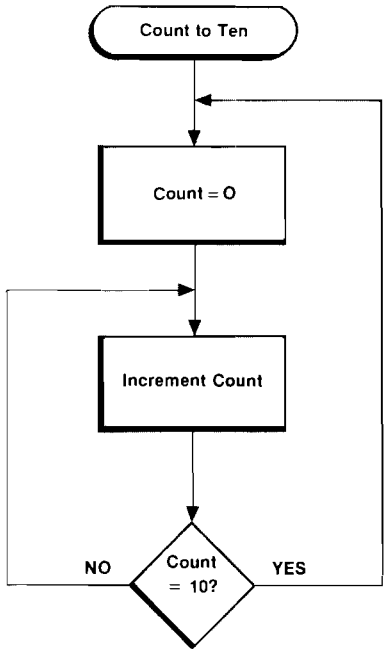


Figure 3-5. Count to Ten Flowchart

The programs that follow are intended to give you an idea of what the various types of languages look like. Do not worry about remembering the details. Programming is discussed in greater detail in the following lessons.

High-Level Language

The translation from the flowchart to a high-level language is fairly simple. The following example uses a variant of the high-level language called *BASIC*, which has the advantage of being simple and similar to English.

Line No.	Instruction	Description
1	LET COUNT = 0	Set Count to 0
2	LET COUNT = COUNT + 1	Increment Count
3	IF COUNT = 10 THEN 1	Go to 1 if Count = 10
4	GO TO 2	Otherwise go to 2

Table 3-1. Count to Ten Program in BASIC

Table 3-1 shows the program listing. The first two lines of the program correspond exactly to the first two action blocks of the flowchart. In the first line the memory location called COUNT is set to zero. The second line, LET COUNT = COUNT + 1, is simply a way of saying "increment count." Lines three and four perform the function of the decision block. Line three specifies that if COUNT = 10, then the next instruction executed should be line one. If COUNT $\neq$ 10, this instruction has no effect, and the program continues with line four. That instruction says "go to line two." Thus, these two instructions perform the actions required by the decision block in the flowchart. Try following the program step-by-step to see the flow as the count reaches ten.

Assembly Language

Assembly language is not one specific language, but a class of languages. Each microprocessor has its own machine language and therefore its own assembly language (as defined by the manufacturer). This example uses the assembly language for Intel's 8085 microprocessor, upon which the μLab is based. (Appendix B contains a complete description of the instruction set for the 8085.)

Label	Instruction	Comments
START:	MVI A,0	;Set A register to 0
LOOP:	INR A	;Increment A register
	CPI 10	;Compare A register to 10
	JZ START	;Go to beginning if A = 10
	JMP LOOP	;Repeat

Table 3-2. Count to Ten Program in 8085 Assembly Language

Table 3-2 shows the assembly language listing for the count to ten program. This program is certainly more cryptic than the BASIC language program, but it performs the same function. Remember that the characteristics of the assembly language are directly related to the characteristics of the microprocessor. This program is therefore different from the BASIC program, which is designed to be related to English rather than to the microprocessor's machine language.

The three columns are for *labels*, *instructions*, and *comments*. The label provides the same function as the line number. Instead of numbering every line, you simply make up a name (called a label) for each line to which you need to refer. A colon (:) is used to identify the label. A line needs a label only if there is another instruction in the program that refers to that line. The label allows you to easily identify a line that you want to jump to during the execution of a program.

The comments are an aid to understanding the program. A semicolon (;) identifies the beginning of a comment. High-level language programs do not need many comments because the instructions themselves are more descriptive. For assembly language programs, however, comments are an invaluable aid. They are useful for people other than the programmer who need to understand the program, as well as for the programmer, who may need to go back to the program after some time.

The first instruction is MVI A, 0 (move immediate to accumulator the data zero). The *accumulator* (also called the A register) is a storage location inside the microprocessor. This instruction is the equivalent of LET COUNT = 0, except that, instead of making up a name for the variable (COUNT), we used a preassigned name (A) for a register inside the microprocessor. Later you will see why this register is called the accumulator. For now, think of it simply as a general-purpose storage location in the microprocessor, which this instruction has loaded with the data zero.

The next instruction, INR A, means "increment the value in the accumulator." The accumulator contains the count, so this is the equivalent of LET COUNT = COUNT + 1.

The next three instructions together implement the decision function.The instruction CPI 10 (compare immediate) means "compare the value in the accumulator with the value ten." It does not directly cause any jumps, regardless of the outcome of the comparison. Instead it sets a special flip-flop (called a *flag*) in the microprocessor if the value in the A register is equal to ten. Then the next instruction, JZ START, tests this flag. If the values are equal, this instruction detects that the flag is set and causes a jump to the line with the label START. These two instructions together (CPI 10 and JZ START) perform the function of the BASIC statement IF COUNT = 0 THEN 1. The last instruction, JMP LOOP, simply causes a jump to the line with the label LOOP. It is the equivalent of the BASIC statement GO TO 2.

Machine Language

As the last step of this example, Table 3-3 shows a listing of the machine language that corresponds to the assembly language program just discussed. The function of this program has become thoroughly obscured, and the problems of dealing with machine language should now be apparent. However, the compelling reason to use it is that it is the only language that the microprocessor understands directly. Note that we began with an English program description and ended with a sequence of ones and zeros suitable for storing in the microcomputer's memory.

Memory Address (Hex)	Memory Contents (Hex)	(Binary)
07F0	3E	00111110
07F1	00	00000000
07F2	3C	00111100
07F3	FE	11111110
07F4	0A	00001010
07F5	CA	11001010
07F6	FO	11110000
07F7	07	00000111
07F8	C3	11000011
07F9	F2	11110010
07FA	07	00000111

Table 3-3. 8085 Machine Language Listing for Count to Ten Program

To understand the machine code, refer to Table 3-4, which compares the three programs. Each memory location in the μLab holds eight bits of data. (Eight bits can be represented by two hex characters.) Each instruction begins with an *opcode* (short for operation code). The opcode specifies the operation to be performed. All 8085 opcodes are eight bits (one byte) each, and therefore occupy one memory location. An opcode may be followed by zero, one, or two bytes of data, depending upon the instruction.

BASIC Language		8085 Assembly Language			8085 Machine Language		
Line No.	Instruction	Label	Instruction		Address	Contents	
1	LET COUNT = 0	START:	MVI	A,0	07F0	3E	*Opcode*
					07F1	00	*Data*
2	LET COUNT = COUNT + 1	LOOP:	INR	A	07F2	3C	*Opcode*
3	IF COUNT = 10 THEN 1		CPI	10_{10}	07F3	FE	*Opcode*
					07F4	0A	*Data*
			JZ	START	07F5	CA	*Opcode*
					07F6	F0	*Address*
					07F7	07	
4	GO TO 2		JMP	LOOP	07F8	C3	*Opcode*
					07F9	F2	*Address*
					07FA	07	

Table 3-4. Count to Ten Program in Three Languages

The first byte (3E) at address 07F0 is the opcode for the instruction MVI A. It is, however, only part of the complete instruction. It specifies that you want to move some data into the accumulator. Now you need another memory location to specify this data. Therefore, the next memory location (address 07F1) contains 00, the data to be moved to the accumulator.

The third location contains the opcode for the second instruction, INR A. This opcode (3C) tells the microprocessor to increment the accumulator. Since there is no additional data associated with this instruction, it occupies only one memory location.

The code FE is the opcode for the compare instruction, CPI. Just as with the MVI A, 0 instruction, the memory location that follows the opcode contains the data required by the instruction. Because the machine language is shown in hexadecimal notation, the data (10 decimal) appears as 0A (hex). This instruction compares the accumulator with the value 10 and sets a flag (as described earlier) if they are equal.

The JZ instruction has the opcode CA, which appears at address 07F5. This opcode tells the microprocessor to jump if the flag is set. The next two memory locations tell it what address to jump to. Since addresses in an 8085 system are 16 bits long, it takes two memory locations (8 bits each) to store an address. The two parts of the address are stored in the reverse of the order you might expect. The least significant half is stored first and then the most significant half. Thus, the address 07F0 is stored as F0 07. The assembly language instruction JZ START means that the processor should jump to the instruction labeled START. The machine code must then use the actual address that corresponds to the label START (07F0 in this case).

The last instruction, JMP LOOP, is coded in the same way. The only difference is that this jump is independent of any conditions. The code for this type of jump is C3. The jump address (07F2) is stored in the same way as for the JZ instruction.

Note that the machine language program consists of a series of bytes, each of which may have one of three meanings. Some are opcodes, some are data, and some are jump addresses. You must know the context of the information to know which type it is. Circuits within the microprocessor determine whether a particular opcode should be followed by data or an address, so it can keep track of the three types.

COMPARING THE DIFFERENT TYPES OF LANGUAGES

High-level languages are the easiest for programmers to use and can be independent of any particular microprocessor. However, lengthy translation programs must be stored in the microcomputer's memory to translate the programs to machine code. High-level languages are also less efficient in terms of speed of operation and memory usage. An equivalent program written in assembly language normally runs faster and occupies less memory.

Assembly language is widely used for programming microcomputers. It is more difficult to write programs in assembly language than in a high-level language. However, it is much easier to translate from assembly to machine language than from high-level to machine language. In applications in which the program must run as rapidly as possible or fit into as small a memory as possible, assembly language is usually the best choice. From an educational viewpoint, programming in assembly language gives you a much better idea of how the microprocessor system works.

Machine language is the only language directly understood by the microprocessor, but people have a hard time using it. It is difficult to program directly in machine language. Programs are usually written in assembly language and then translated to machine code. The translation may be performed by a special program (called an *assembler*).

PROGRAMMING THE μLAB

When programming the μLab, the assembly language is "hand assembled" by looking up the machine code (in hex) for each instruction in the instruction table in the *Assembly Language Reference Card* included with this book. Only machine language programs may be directly entered into the μLab. In general, you will write programs in assembly language and then translate them to machine code (in hex). The machine code is then entered into the μLab. The hexadecimal representation is used for the machine code, since it is easier to use than binary. The μLab contains software to perform the relatively simple translation from hexadecimal to binary.

EXPERIMENT 3-2

Interpreting Memory Contents

CONCEPT

In this experiment you will examine the contents of part of the μLab's memory and then translate the machine code into assembly language (the reverse of the usual assembly process).

PROCEDURE

A) Turn on the μLab if necessary.

B) The display should show μLAb UP. If not, press [RESET].

C) Press [FETCH ADRS] [0] [7] [F] [0]. This tells the μLab that you want to examine address 07F0. The data at this location (3E) is listed in the contents column of Table 3-5.

D) Press [STORE INCR] to examine the next memory location.

E) Copy the data shown in the display into the contents column of Table 3-5.

F) Repeat steps D and E until you have filled in all the contents entries in Table 3-5.

G) Look at the Assembly Language Reference Card, which lists all the opcodes and the corresponding mnemonics. Note that there are two lists: one grouped by function and one with the opcodes listed in numeric order (on the back of the card). The information on this card also appears in Table B-1 in Appendix B.

Address	Contents	Mnemonic/Data/Address
07F0	3E opcode	_____
07F1	_____ data	_____
07F2	_____ opcode	_____
07F3	_____ opcode	_____
07F4	_____ data	_____
07F5	_____ opcode	_____
07F6	_____ address	_____
07F7	_____ address	_____
07F8	_____ opcode	_____
07F9	_____ address	_____
07FA	_____ address	_____

Table 3-5. Instruction List for Experiment 3-2

H) Using the table that lists the opcodes in numeric order, fill in the mnemonics in Table 3-5. Remember that not every location contains an opcode. Many instructions consist of an opcode

Continued

followed by data or a jump address. To help you decode the machine code, Table 3-5 indicates which bytes are opcodes, which are data, and which are jump addresses. You need to look up only the opcodes. The data and jump address can simply be copied into the right-hand column.

l) Look back at the program listed in Table 3-4. The assembly language listed there should be the same as the one you just entered in Table 3-5.

SUMMARY

A listing was produced of information stored in the μLab's ROM. You then translated the codes into mnemonics. This is essentially a translation from machine code to assembly language, called a *reverse assembly* or *disassembly.* Information in the microprocessor's memory can be either opcodes, data, or jump addresses, as determined by the context of the information.

AN APPLICATION EXAMPLE

There are many different applications in which microprocessors can be used. For example, consider a conveyor belt that moves gears from the manufacturing area to to the shipping area of a factory. As the gears drop off the end of the belt into a shipping carton, an operator must count them as they drop into the box and replace the full box with an empty one when it contains ten gears. Several things can be done to automate this system.

First, some hardware is required (see Figure 3-6). An electric eye (the gear sensor) can be used to provide a pulse every time a gear goes by, so that they can be counted. A conveyor belt is needed for the boxes so that a full box can be automatically moved away and replaced by a new box. Finally, a controller is required to count the gears and move the boxes when necessary.

Using standard digital logic, a special-purpose controller could be built to count ten pulses and then send a pulse to the box conveyor. That is the random logic approach. The same task can be performed by a microcomputer, using a program to tell it what to do.

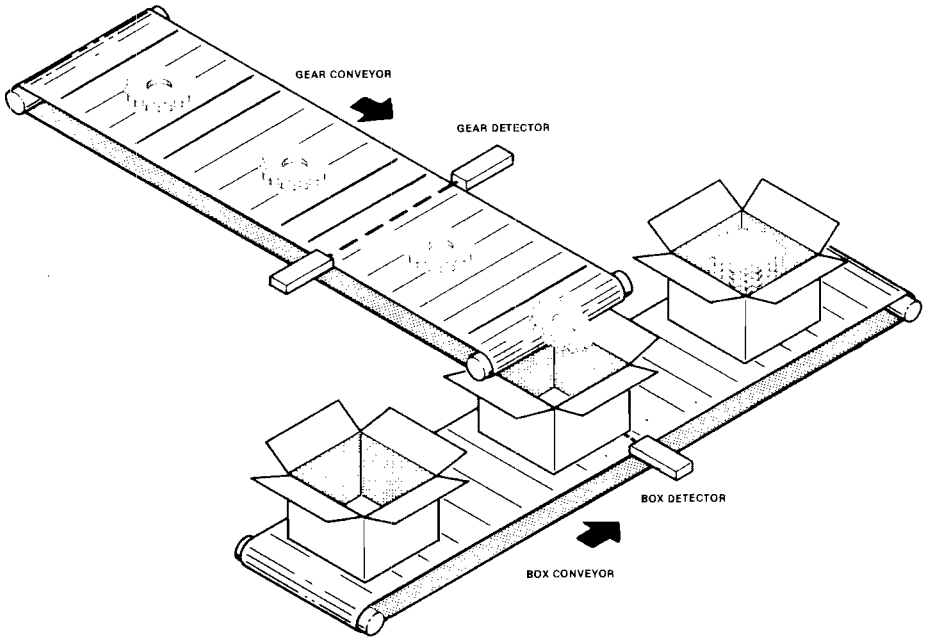


Figure 3-6. Conveyor Belt System

Figure 3-7 shows the flowchart for this program. First, the gear conveyor is stopped, and a new box is moved in. Then the gear conveyor is started, and the gears are counted. When ten have passed by, the process is repeated.

The word *count* in the flowchart does not refer to a physical counter. It is a name assigned to a memory location. "Set count to zero" means store the value zero in that memory location. "Increment count" means add one to the value in that memory location. These are very general types of operations (not specific to this application) and are typical of the instructions that microprocessors can execute.

This application demonstrates the use of software (programs) to replace hardware. The system contains no special-piece of hardware to act as a counter. Instead, it uses a general-purpose memory location along with the microprocessor's computation capabilities. This type of function is commonly performed by software in microprocessor systems, but requires hardware in a random logic system.

MODIFYING THE PROGRAM

There is a problem with the flowchart in Figure 3-7. What happens if no box is in place? The conveyor then dumps the gears onto the floor. A human operator would realize that something was wrong, but the microcomputer must be explicitly programmed to detect this situation and take corrective action. Figure 3-8

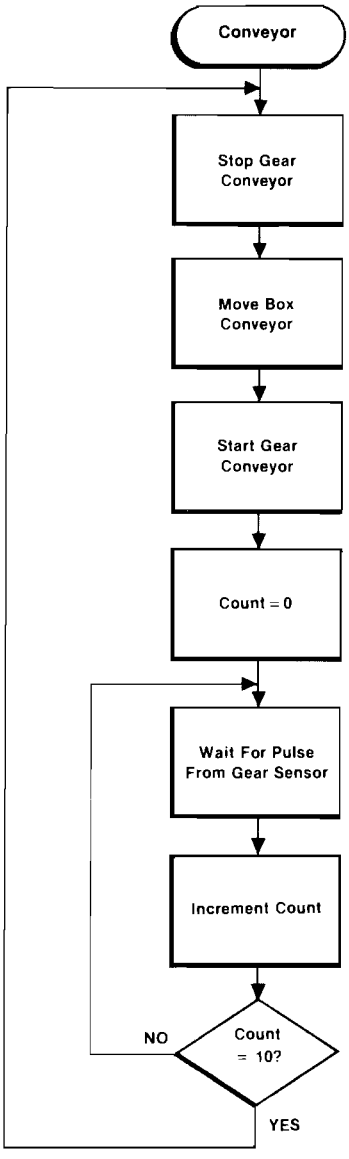


Figure 3-7. Conveyor Belt Controller Flowchart

shows a flowchart modified to do this. After the microprocessor signals the box conveyor to move, the gear conveyor is stopped until an empty box is in place (as detected by a second electric eye).

The flowchart makes it easier to detect and solve this type of problem, as compared with working on the actual program. Once you are satisfied with the flowchart, you can begin to write the program.

Note that if the controller had been built with random logic, these changes would have been hardware changes, which are more difficult to implement.

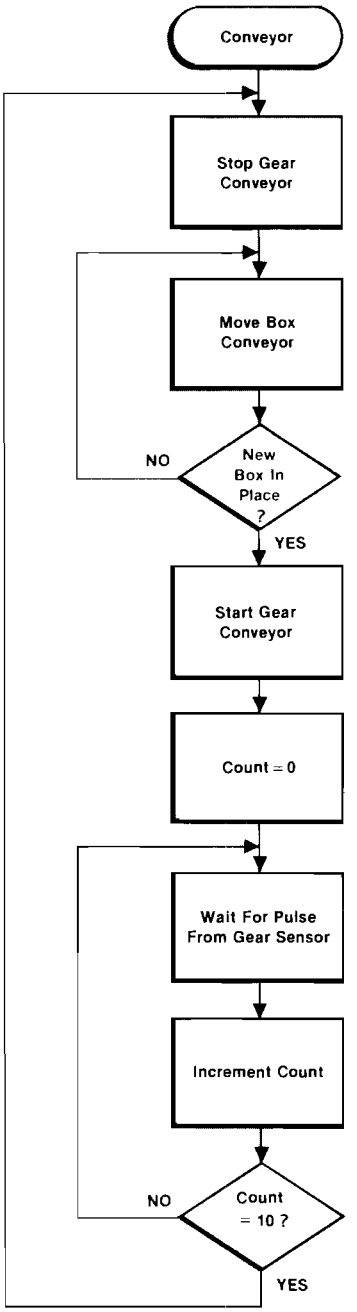


Figure 3-8. Modified Conveyor Belt Controller Flowchart

EXPERIMENT 3-3

Conveyor Belt Simulator

CONCEPT

In this experiment the μLab is used to simulate the conveyor belt controller just described. The μLab contains a program to perform this task following the logic of the flowchart in Figure 3–7. You simulate the "gear sensor" by pressing a button to indicate that a gear has passed.

PROCEDURE

A) Press [RESET] if the display does not show uLAb UP

B) Press [FETCH ADRS] [0] [4] [F] [8] [RUN]. You have now started the program beginning at address 04F8. The display shows 0, which is the number of gears that have passed the electric eye.

C) Press [0]. This simulates a pulse from the electric eye. The display changes accordingly.

D) Repeat step C until the display indicates that ten gears haved passed. The μLab's output LEDs then simulate the motion of the box conveyor.

E) Repeat steps C and D to watch the process repeat.

F) Press [RESET]. This halts the simulator program.

SUMMARY

In this experiment you ran a simulator program stored in the μLab's memory. Using the FETCH ADRS key, you specified the location of the program. Then you started the program by pressing RUN. The RESET key halted the program.

This experiment demonstrated that, by using only the basic μLab, it is possible to make the specialized controller just described. Only the peripherals (the electric eye and the conveyor belt controls) are lacking. Previous experiments showed that the μLab can also be used to simulate a rocket blast-off, create a moving pattern on the display, generate random tones, or function as an AND gate. These examples illustrate the general-purpose nature of the microprocessor.

REVIEW

Lesson 3

A program is a detailed list of instructions that tell the microprocessor what to do, step by step. The microcomputer system can respond to a change in conditions by having a program that detects the change and responds appropriately. It cannot, however, respond to a condition that the program does not expect. The programmer must anticipate all possible contigencies.

The first stage of a program is its conceptual design: deciding what it will do and how it will do it. A flowchart is a convenient way to document the design at this point. Once the flowchart is complete, the program can be written in a programming language.

The μLab is programmed in 8085 machine language by entering hexadecimal codes. Because machine language is difficult to work with, programs are first written in assembly language. This allows you to write the program using mnemonics instead of hexadecimal codes and to label locations with names instead of using actual addresses.

High-level languages such as BASIC are easier to use, but cannot be directly understood by the microprocessor. They are used on larger systems that have special programs to execute the high-level instructions.

The use of programs to control the microcomputer system allows many functions that are normally performed with hardware to be done with software. This can simplify the system's hardware considerably and provides tremendous flexibility. The experiments showed how the μLab could perform many different tasks by running various programs.

One disadvantage of using software to replace hardware is that the programmed version usually operates more slowly because a microprocessor executes instructions sequentially, doing one simple step at a time. At any given time, the entire microcomputer system is devoted to executing only one instruction. In a random logic design, on the other hand, different parts of the circuit can be doing different things at the same time. Another way of saying this is that microprocessors must operate sequentially, while random logic circuits can operate in parallel.

QUIZ

Lesson 3

1. The language that the microprocessor understands directly is called:
   a. assembly language.
   b. high-level language.
   c. machine language.
   d. BASIC language.

2. When writing a program for the μLab, you first write it in _____ language, and then translate it to _____ language.

3. One advantage of the microprocessor-based AND gate over a standard AND gate is that:
   a. it is faster.
   b. it is easier to modify.
   c. the hardware is simpler.
   d. it uses less power.

4. The purpose of the labels in an assembly language program is to:
   a. give the program a name.
   b. explain the purpose of the instruction.
   c. identify the start of the program.
   d. identify locations to which other instructions refer.

5. The μLab can perform many different functions because:
   a. its operation is controlled by software.
   b. it is a digital system.
   c. it uses a RAM.
   d. it is controlled by the keyboard.

II

INTRODUCTION TO PROGRAMMING

This section presents a series of programs and introduces several software concepts. Experiments will be used to familiarize you with the use of the μLab.

The μLab is a small microcomputer designed specifically for educational use. It uses an 8085 microprocessor with 2K bytes of ROM and 1K bytes of RAM. The system includes a keyboard from which you can enter programs, store data, and give commands to control the operation of the microcomputer and a display that allows you to view the contents of the memory and registers. There is an output port with LEDs on each of the eight output lines and an input port with a slide switch on each of the eight input lines. A speaker on the board is controlled by the processor. Finally, there are LEDs on the address bus, the data bus, and the major control lines so that you can monitor their activity.

The ROM in the μLab contains programs to read the keyboard, execute keyboard commands, and send data to the display. The entire operation of the system is governed by this software, called the *monitor.* The following experiments demonstrate the features of the monitor and the hardware that it controls.

LESSON 4

Using the Microprocessor Lab

INTRODUCTION

The basic operations performed by the Microprocessor Lab are storing data in memory, examining the contents of the memory, and running programs. Programs can be executed at full speed, as in the previous experiments, or they can be executed one step at a time to allow you to follow their operation in detail. This lesson illustrates these features.

THE MICROPROCESSOR LAB MEMORY MAP

Figure 4-1, the *Memory Map* for the μLab, shows which addresses are assigned to each device. The ROM occupies addresses 0000-07FF, and the RAM occupies 0800-0BFF. Note, however, that not all of the RAM is available for your programs.

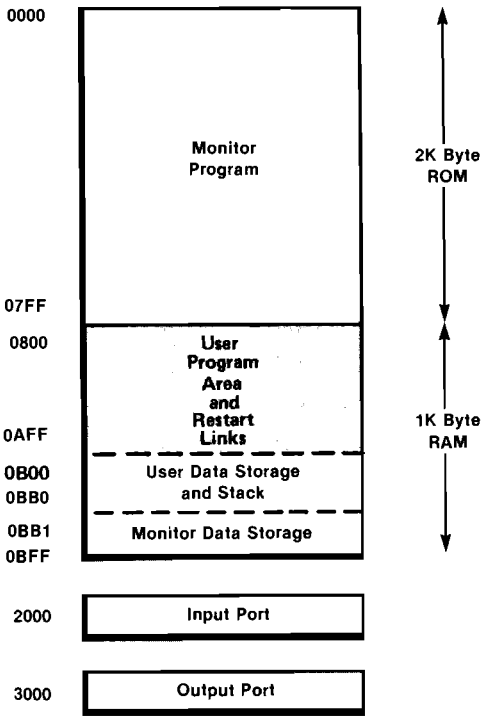


Figure 4-1. Microprocessor Lab Memory Map

The top section of the RAM, from 0BB1 to 0BFF, is used by the monitor program and should not be used by your programs; locations 0B00-0BB0 are used for other purposes to be described later. The area from 0800-0AEE is available for your programs.

The input port resides at address 2000, and the output port at address 3000. There are other ports used to control the keyboard, display, and single-step circuits, which are described in section III, Microprocessor System Hardware.

EXPERIMENT 4-1

Examining Memory and Storing Data

CONCEPT

This experiment uses three keys to examine and modify the contents of the memory in the μLab. The FETCH ADRS key allows you to examine the data stored at any address, and the STORE/INCR key is used for storing data or examining successive memory locations. These two keys were used in Lesson 3, and in this experiment, their operation is examined in greater detail. The DECR (Decrement) key, which allows you to step backwards through memory, is also introduced.

PROCEDURE

I. Examining the Contents of the Memory

A) The display should show μLAb UP indicating that the system is in its normal "waiting for command" mode. If it does not show this display, press [RESET].

B) Press [FETCH ADRS] [0] [0] [0] [0]. The display now shows the address (which you have just entered) in the left four digits of the display and the data (26) stored at that address in the right two digits of the display. Remember that addresses and data are displayed in hexadecimal.

C) Press [STORE INCR]. This directs the μLab to store the data shown in the display at the address shown in the display, and then to increment the address. Since you just fetched the data from location 0000 and then stored the same data in the same location, you did not change the contents of the memory in any way. In this situation, the [STORE INCR] key functions as an "increment address" key for examining successive locations in memory. By repeatedly pressing [STORE INCR], you can examine a section of memory. You used this feature previously in Experiment 3-2.

II. Storing Data in the Memory

A) Press [FETCH ADRS] [0] [8] [0] [0]. This specifies address 0800, which is the beginning of the RAM. The power-up program fills the RAM with zeros, so this location (and all others up through 0AEE) contains the data 00.

B) Press [STORE INCR]. This advances the address to 0801 and leaves 00 stored in 0800.

C) Press [C] [3]. Notice that this appears in the data section of the display, and the right-most decimal point is lighted. The decimal point indicates that you are in a "data-entering" mode.

(Continued)

D) Press [STORE INCR]. You have now stored the data C3 at address 0801, and the address has been incremented to 0802.

Note that the decimal point goes off, indicating that the data just entered has been stored and the μLab is no longer in a data-entering mode.

E) Press [STORE INCR]. This leaves 00 in 0802 and goes on to 0803.

F) Press [8] [STORE INCR]. Note that you do not need to press 0 first. If you only enter one digit, the μLab automatically inserts a leading zero. You have now stored the following data:

Address	Contents
0800	00
0801	C3
0802	00
0803	08

G) Press [DECR]. This decrements the address shown in the display and is useful for checking the data that you have just entered.

H) Repeat step G until you have verified that the above listing is contained in the memory. Note that [DECR] does not affect the contents of the memory in any way.

You could also have checked the data by pressing [FETCH ADRS] [0] [8] [0] [0] and then using [STORE INCR] to examine successive locations.

III. Correcting Mistakes

A) Press [FETCH ADRS] [0] [0] [0] [0]. This selects the first address in ROM.

B) Press [0] [STORE INCR]. This attempts to store 00 in this location.

C) What happened? You tried to store data in the ROM, which is impossible. The monitor detects this error, issues a warning beep, does *not* increment the address, and shows you the data that is stored there.

D) This procedure pointed out a mistake that the monitor detected. But what about mistakes such as entering the wrong address or data? When entering an address, you can always press [FETCH ADRS] again to start over.

To illustrate this, press [FETCH ADRS] [0] [7]. Suppose that at this point you realize that the address you want is 0900. Even though you are in the middle of entering the address, you can recover by pressing [FETCH ADRS] [0] [9] [0] [0].

E) Now that you have entered the correct address, you can enter some data. Press [7] [9]. This data now appears in the right two digits of the display.

F) Suppose this is not the data you intended to enter: what you wanted was 69. Simply enter the correct data, and the incorrect data will be shifted out. When you are entering data, you can keep entering digits indefinitely, and the μLab keeps only the last two. The decimal point indicates that it is still accepting data. Once you press [STORE INCR], however, the data is stored in memory, and the decimal point goes off.

G) Press [STORE INCR]. You just stored 69 at address 0900. Suppose you now want to change that to 68.

H) Press [DECR]. The address 0900 and the data 69 appear in the display. Key in the correct data, 68, and press [STORE INCR]. You have now changed the data.

SUMMARY

In this experiment you produced a listing of part of the ROM and then stored some data in RAM. The following keys were used:

[FETCH ADRS] allows you to specify an address and view the data stored there.

[STORE INCR] stores the data shown in the display at the address shown, and then displays the next location.

[DECR] decrements the address shown in the display without affecting the data and shows you the previous location.

These three keys enable you to easily store and examine data in the μLab's memory.

A SIMPLE PROGRAM

The data you stored in the RAM during the previous experiment is actually a program. The listing is shown in Table 4-1.

Address	Contents	Label	Instruction	Comments
0800	00	START:	NOP	;No operation
0801	C3		JMP START	;Jump to
0802	00			beginning
0803	08			

Table 4-1. Program Listing

The opcode 00 (stored at address 0800) has the mnemonic NOP, which stands for no operation (often abbreviated no-op). This instruction, as its name implies, does nothing. Then why does it exist? It is useful for two reasons. Because time is required to fetch it and decide which instruction it is, it can be used for a short time delay when no other operation is desired. Its most important use is as a space filler. If you later want to go back and add instructions to a program you have already written, you can just replace the NOPs with the desired instructions. If you had not put any NOPs into the program, you might have to move a large section of the program just to insert a single additional instruction.

The other instruction in the program is a jump. The code C3 (mnemonic JMP) means jump to the address specified in the next two bytes of memory. The jump address is stored with the *least-significant address byte first,* and the most-significant address byte second. Thus, the jump address 0800 is stored as 00 in 0802 and 08 in 0803.

This program, of course, does nothing but loop endlessly. Figure 4-2 shows the flowchart.

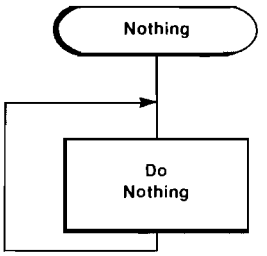


Figure 4-2. "Do Nothing" Flowchart

EXPERIMENT 4-2

Executing Programs

CONCEPT

The μLab has three modes for executing a program: the usual RUN function, which runs the program at full speed, and two different single-step modes, which enable you to execute the program one instruction (or part of an instruction) at a time. RESET is used to halt the execution of the program and to exit the hardware step mode.

PROCEDURE

I. Executing the Program Using Instruction Step

A) Press [FETCH ADRS] [0] [8] [0] [0]. Verify that the data at this location is the same as that shown in the first line of Table 4-1.

B) Press [STORE INCR]. Check the data at this location.

C) Repeat step B until you have verified that the program shown in Table 4-1 is stored in the memory. If it is not, repeat Experiment 4-1, Part II.

D) Press [FETCH ADRS] [0] [8] [0] [0] to set the address back to the beginning of the program (0800). (Alternatively, you can use [DECR] to get the display to show address 0800).

E) Press [INSTR STEP]. This causes the instruction shown in the display to be executed, and the display now shows the next instruction. Note that although this key may appear to act like [STORE INCR], its function is very different. When using [STORE INCR], you are simply examining the contents of memory. With [INSTR STEP] you are causing the contents of the displayed memory location to be interpreted as an instruction, which is executed by the microprocessor.

When instructing the μLab to execute an instruction (or a program) by pressing [INSTR STEP], [HDWR STEP], or [RUN], the display must show a location that contains an opcode. The location at which execution begins must not contain data or an address. For example, the program in Table 4-1 may begin execution at 0800 or 0801, but not at 0802 or 0803.

F) Now the jump instruction, C3, is shown in the display. Press [INSTR STEP]. Since this is a jump instruction, the address jumps to 0800. Note that the memory locations containing the jump address are never shown, because when you press [INSTR STEP], the μLab executes the entire instruction, which includes reading the two jump address bytes.

(Continued)

G) Press [INSTR STEP] repeatedly to watch the loop repeat.

II. Executing the Program Using Hardware Step

A) Press [FETCH ADRS] [0] [8] [0] [0] . This sets the address to the beginning of the program.

B) Press [HDWR STEP]. The display goes blank because the hardware step key completely stops the processor after an instruction is executed. [INSTR STEP], on the other hand, executes an instruction and then returns control to the monitor program.

C) Look at the sixteen LEDs labeled ADDRESS. These are connected directly to the address bus. They show 0000 1000 0000 0000. Using the binary-hex table on the PC board (to the right of the address LEDs), convert this to hex. It should be the address you specified in step A.

D) Look at the eight LEDs labeled DATA. These are connected to the data bus and show the data (00) stored at address 0800.

E) Look at the six LEDs labeled STATUS. These indicate whether a read or a write is being performed and whether it is to ROM, RAM, the input port, or the output port. The READ and RAM LEDs are on, indicating that information is being read from the RAM.

F) Press [HDWR STEP]. The address increments and the data and status LEDs show the information corresponding to the new address. Just as with [INSTR STEP], the instruction is executed.

G) The data LEDs now show the jump opcode C3 (1100 0011). Press [HDWR STEP]. Note that the address is incremented, and the jump is *not* performed. This points out a fundamental difference between the two step modes: [HDWR STEP] steps one memory location at a time, whereas [INSTR STEP] steps one instruction at a time (even if the instruction uses several memory locations). With [HDWR STEP], the instruction is not executed until all the parts have been read (e.g., a three-byte instruction does not begin execution until the third step).

H) Press [HDWR STEP] [HDWR STEP]. This completes the jump instruction. The address LEDs now show 0800. The jump instruction has been executed.

I) Press [HDWR STEP] repeatedly and watch the program repeat.

J) Press [RESET]. This brings the system back to normal operation. When [RESET] is pressed, the instruction waiting to be executed (as shown in the data LEDs) is executed. The display then shows the next instruction.

III. Running the Program

A) Press [FETCH ADRS] [0] [8] [0] [0] . This sets the address.

B) Press [RUN] . The program is now running at full speed (approximately 2 microseconds per instruction).

C) Look at the address LEDs. They appear to show 0803 (0000 1000 0000 0011). They are in fact counting from 0800 to 0803 and then going back to 0800. This is the same program sequence you stepped through previously. Now, however, the LEDs are changing so fast that they always appear to be on.

D) Look at the status LEDs. They indicate a read from RAM, the same as during the hardware step mode. Since all the instructions light the same status LEDs (this program reads only from the RAM), the fact that the program is running at full speed does not cause an ambiguous display (more than one set of LEDs lighted at a time).

E) Look at the data LEDs. For reasons that will be explained later, these all appear to be on when you are running a program. They are useful only in the hardware step mode.

F) Press [RESET] . This halts your program and returns control to the monitor. The display shows the instruction that was about to be executed when you pressed [RESET] .

G) The μLab monitor will not let you run a program if you are in the process of entering data. To observe this control, press [FETCH ADRS] [0] [8] [0] [0] and then key in the data [1] [2] . The decimal point goes on, indicating that the μLab is in a data-entering mode.

H) Press [RUN] . The μLab does not respond because it is in a data-entering mode.

I) Suppose you now realize that you did not want to change the data but wanted only to start the program running. Press [DECR] to get out of the data-entering mode, and then press [STORE INCR] to go back to address 0800. Alternatively, you could press [FETCH ADRS] [0] [8] [0] [0] . Note that the decimal point is now off.

J) Press [RUN] . Now the program runs.

K) Press [RESET] to return control to the monitor.

Continued

SUMMARY

This experiment demonstrated the three modes for executing a program: instruction step, hardware step, and run. The function of the keys is as follows:

INSTR STEP — causes the instruction shown in the display to be executed. The μLab then returns to the monitor program.

HDWR STEP — first causes the microprocessor to jump to the address shown in the display and then stop. Subsequent presses cause the byte shown in the binary bus LEDs to be read, and when an entire instruction has been read, it is executed. The microprocessor halts until HDWR STEP is pressed again (or RESET is pressed).

RUN — causes the microprocessor to start executing instructions at the address shown in the display and continue executing instructions in sequence.

RESET — causes the μLab to return to the "wait for command" mode. It is used for exiting the RUN or hardware step modes. It does not affect the contents of the memory or registers.

There are two important differences between the two step modes. First, in the hardware step mode, the microprocessor is stopped, the display is blank, and the binary LEDs on the address, data, and status lines indicate the state of the buses. They provide more information than the display, but are more difficult to interpret. They are an exact indication of what is going on in the μLab hardware. In the instruction step mode the display is on, but the binary address and data LEDs are meaningless.

The second difference is that each press of HDWR STEP fetches only a single byte, regardless of how many bytes it takes to complete the instruction. The instruction is executed only after all parts of the instruction have been stepped through. On the other hand, each time the INSTR STEP key is pressed, the instruction shown in the display is executed. If the instruction consists of multiple bytes, the entire instruction is executed (including reading the second and third bytes) and the second and third bytes never appear in the display. Instruction step is therefore more convenient for following general program flow, while hardware step allows the detailed operation of each instruction to be observed.

Note that the HDWR STEP key actually has two functions. When the μLab is not in the hardware step mode, the first press of the key puts it into that mode. Then, once it is in this mode, the key causes one step to occur. The RESET key is used to return to normal operation.

In the RUN mode it is difficult to obtain much information about what is going on. It is used for programs that do something, as opposed to this "do-nothing" program.

THE INPUT AND OUTPUT PORTS

The μLab has an input port with eight miniature slide switches to set the data at its inputs. The microprocessor can then read this data from the switches. The μLab also has an output port with an LED on each of the eight output lines. This enables the microprocessor to control the LEDs. These ports were used in the AND gate experiment in Lesson 3.

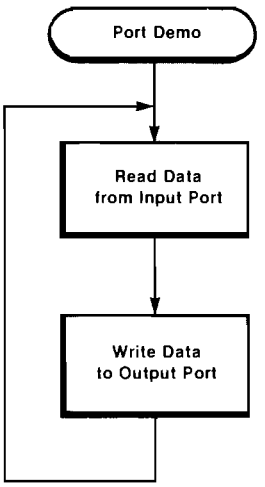


Figure 4-3. Flowchart for Program to Copy Data from Input to Output Port

Figure 4-3 shows the flowchart for a program that demonstrates the use of these ports. This program reads the data from the input port and writes it to the output port. The program listing is shown in Table 4-2.

Address	Contents	Label	Instruction	Comments
0900	3A	START:	LDA 2000	;Read input port
0901	00			
0902	20			
0903	32		STA 3000	;Write data to
0904	00			output port
0905	30			
0906	C3		JMP START	;Loop back
0907	00			
0908	09			

Table 4-2. Program to Copy Data from Input Port to Output Port

The first instruction is LDA 2000, which means load the accumulator with the contents of location 2000. Like memory locations, input and output ports also have addresses. The address of the μLab's input port is 2000 (as shown in the Memory Map in Figure 4-1). This instruction therefore loads the accumulator with the data from the input port.

The second instruction, STA 3000, means store the contents of the accumulator at address 3000. 3000 is the address of the output port. This instruction therefore takes the contents of the accumulator, which was just read from the input port, and sends it to the output port. The program thus transfers data from the input port to the output port.

The last instruction is a jump. It causes the program to jump back to the beginning and creates a loop that is executed indefinitely. The data at the output port will therefore follow the data at the input port while the program is running.

All three of these instructions have the same machine code format: an opcode followed by two bytes of address information. For example, 3A is the opcode for the LDA instruction. The next two bytes, 00 and 20, specify the address 2000.

EXPERIMENT 4-3

Using the Input and Output Ports

CONCEPT

The program to transfer data from the input port to the output port is entered and run. The switches at the input port may then be set to any desired position, and the same data will appear in the output port LEDs.

PROCEDURE

A) Press [FETCH ADRS] [0] [9] [0] [0].

B) Press [3] [A] [STORE INCR]. This stores the first byte of the program shown in Table 4-2.

C) Press [0] [STORE INCR]. This stores the second byte of the program, 00, in location 0901.

D) Continue storing the data from Table 4-2 until the entire program is entered.

E) Verify that the program is correctly stored.

F) Run the program by pressing [FETCH ADRS] [0] [9] [0] [0] [RUN].

G) Set the input port switches to any position. Up corresponds to a one, down to a zero.

H) Look at the output port LEDs. They show the same data as that indicated by the input port switches. Note that the output port LEDs indicate negative logic (i.e., they are on for a zero, and off for a one). They are connected this way because the TTL gate used to drive them cannot output very much current, but it can sink (input) a substantial amount of current. LEDs are commonly connected this way.

I) Change the input port switches. The output port data should change correspondingly.

J) Press [RESET]. The program stops, and control returns to the monitor.

K) Change the input port switches. Does the output port data change? Since the program you entered is no longer running, the input port is not being read.

L) The instruction CMA (Complement Accumulator, code 2F) causes the value in the accumulator to be complemented (inverted). Each one bit becomes a zero bit, and each zero bit becomes a one bit. Add this instruction at the appropriate place in the program so that the data read from the input port is complemented before it is written to the output port. Do not forget to move the necessary instructions to make room for the new one. Also note that each instruction in the program is three bytes long and that the three bytes must remain together. (The answer is in Table 4-3 at the end of the experiment.)

(Continued)

M) Run the modified program, and check to see that it performs as expected. An *up* switch should now correspond to an *on* LED.

N) Press RESET to halt the program.

SUMMARY

Data was read from the input port and sent to the output port. As long as the program was running, the data in the output port LEDs followed the setting of the input port switches. Since the output port LEDs indicate negative logic, an *up* switch corresponded to an *off* LED. To change this, an instruction was added to complement the contents of the accumulator. This is an example of the flexibility of a microprocessor-based system. If the switches were connected directly to the LEDs, eight inverter circuits would be required to make this change. But since they are connected via a microprocessor, a small change in the software is all that is needed to produce the desired effect.

Address	Contents	Labels	Instructions	Comments
0900	3A	START:	LDA 2000	;Read input port
0901	00			
0902	20			
0903	2F		CMA	;Complement data
0904	32		STA 3000	;Write data to output port
0905	00			
0906	30			
0907	C3		JMP START	;Loop back
0908	00			
0909	09			

Table 4-3. Program Modified to Complement Data

Lesson 4

The μLab has keys to allow you to easily examine and change the contents of the memory, run a program, or single-step through a program in either of two modes. The instruction step mode allows you to easily follow the program flow, while the hardware step mode allows you to see the detailed operation of the address and data buses.

Keep in mind that these key functions are defined by the monitor program in the ROM and the circuits of the μLab, and not by the microprocessor. The functions of the machine language instructions, on the other hand, are defined by the microprocessor itself.

QUIZ

Lesson 4

1. To examine the contents of memory location 0803, you would press _____ 0803.

2. Suppose the display shows address 0803. To see the contents of location 0802, you could press _____ or _____.

3. One difference between instruction step and hardware step is that:
   a. hardware step allows the state of the buses to be observed.
   b. hardware step allows the program to be followed on the display.
   c. instruction step steps one memory location at a time.
   d. instruction step causes the program to run at full speed.

4. Input and output ports are identified by _____, just like memory locations.

5. The functions of the keys on the μLab are determined by:
   a. the microprocessor design.
   b. the monitor program stored in the ROM.
   c. the contents of the RAM.
   d. all of the above.

LESSON 5

Basic Software Concepts

INTRODUCTION

The previous lesson described the basic functions of the μLab. Using the keys described in that lesson, you can store a program, check it, single-step through it, or run it. These are the essential features of the μLab. In this lesson, programming techniques and some of the more advanced features of the μLab are described.

THE MICROPROCESSOR'S REGISTERS

The 8085 microprocessor IC contains a number of internal registers that are used for various purposes. Some, such as the accumulator, are used for storing and manipulating data. For example, in Lesson 3 the accumulator was used to store the number of counts in the "count to ten" program. The contents of the accumulator can also be modified (in this case, it was incremented). These registers are different from regular memory locations in that they are inside the microprocessor and are not selected by the address bus, as shown in Figure 5-1.

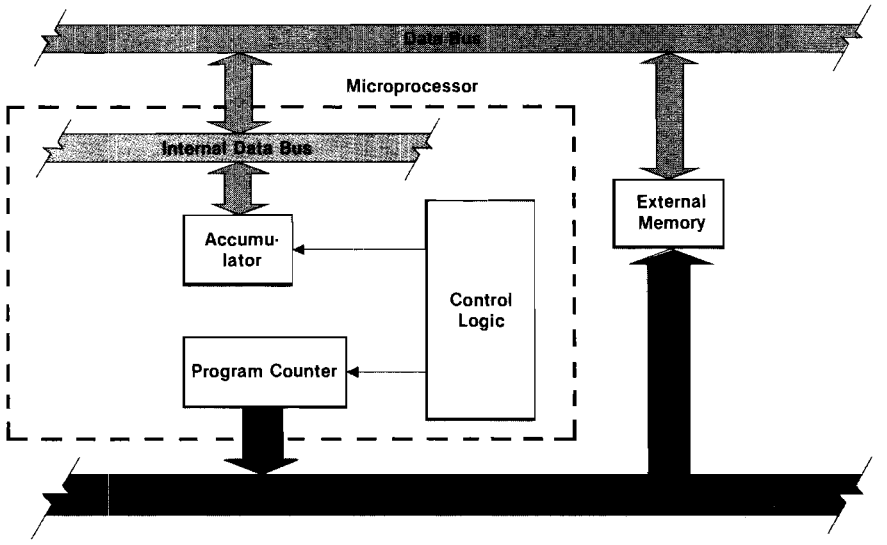


Figure 5-1. Registers and Memory. Registers such as the accumulator are selected directly by the control logic, whereas memory locations must be accessed via the address and data buses.

Instead, they are selected directly by a particular instruction (e.g., MVI A selects the accumulator). The control logic within the processor controls the registers directly (without using the buses). Thus, they are more convenient for temporary or intermediate data storage.

Some registers in the microprocessor are not used for general-purpose data storage but have a very specific function. The *Program Counter* (PC) is the most important of these. It is a sixteen-bit register that keeps track of which instruction is being executed. It always contains the address of the next instruction to be executed. When an instruction is read from the memory, the contents of the program counter are placed on the address bus. The addressed instruction will then appear on the data bus. After the microprocessor reads the instruction, the program counter is incremented. The contents of the PC are again placed on the address bus, and the next instruction is read. This process continues unless a jump instruction is executed. A jump instruction causes the PC to be loaded with the jump address.

A number of other registers in the microprocessor will be introduced as the course progresses.

MONITORING PROGRAM FLOW

To follow the step-by-step operation of a program using the μLab, it is useful to be able to look at the contents of the microprocessor's registers. Since they are internal to the microprocessor and have no addresses, you cannot use the FETCH ADRS key to access them. A special key, FETCH REG (Fetch Register) is used for this purpose. There is also a key called FETCH PC (Fetch Program Counter), which enables you to easily return to a program that has been halted.

THE μLAB's MONITOR PROGRAM

The μLab's ROM contains the monitor program, which reads the keyboard, performs the indicated operation, and controls the display. The μLab runs the monitor program at all times except when the μLab is executing a user program or is in the hardware step mode. Any time that the display is totally off, it can be assumed that the monitor is not running.

When the RUN key is pressed, the monitor program directs the microprocessor to jump to the address specified on the μLab display. When RESET is pressed, the μLab is forced to return to the monitor program.

The monitor program also allows you to examine the contents of the registers as your program left them. When you exit a program by pressing RESET (and also after each instruction step), the monitor program stores the contents of the registers in special RAM locations. Then when you press FETCH REG, the monitor displays the contents of the memory location in which it stored the contents of the register. Similarly, the monitor stores the contents of the PC in RAM and recalls this value when you press FETCH PC. Thus, although it appears that you are actually examining the contents of the registers, you are really examining the contents of the memory locations in which the monitor has stored the register contents. This is necessary because the monitor program also uses the registers.

A COUNTER PROGRAM

To demonstrate the use of these keys, another demonstration program is needed. Figure 5-2 shows the flowchart for a program that causes the output port to count, in binary, from 0 to 255 and then repeat. First, one register (the accumulator, or A register, in this case) is set to zero. The contents of the accumulator are then written to the output port, and the accumulator is incremented. Finally, the program jumps back to the "output" program step.

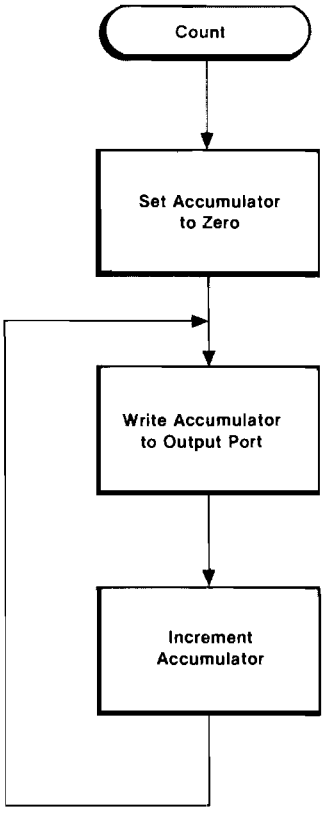


Figure 5-2. Program to Make Output Port Count in Binary

Table 5-1 shows the program listing. The program starts at address 0804 instead of 0800 so that you can later add some instructions at the beginning. (In the μLab, 0800 is the first location of the RAM, so user programs cannot go before it.) The first instruction is MVI A,0. It causes the accumulator to be loaded with the value zero. The next instruction, STA 3000 (store accumulator at location 3000), transfers the contents of the accumulator to the output port (address 3000). The opcode 32 at address 0806 specifies that it is an STA instruction. When the microprocessor sees this opcode, it knows that the next two bytes (addresses 0807 and 0808) contain the address at which the contents of the accumulator are to be stored (3000 in this case). Remember that the two address bytes are stored in reverse order. Note that this instruction does not change the contents of the accumulator; it simply copies the data to the port.

The STA instruction is followed by a NOP to reserve space for later use. The next instruction is INR A, which increments the value in the accumulator. When the maximum count (11111111 binary, FF hex, or 255 decimal) is reached, the value "wraps around" and the next count is zero. This is the same way a standard binary counter IC, such as a 74163, behaves. A special flip-flop in the microprocessor called the *carry flag* is set to indicate that the count has overflowed. The carry flag is discussed in Lesson 12.

The last instruction is a jump. This instruction causes the program to jump back to the STA instruction, address 0806.

Address	Contents	Label	Instruction	Comments
0804	3E		MVI A,0	;Set A to zero
0805	00			
0806	32	LOOP:	STA 3000	;Output A to port
0807	00			
0808	30			
0809	00		NOP	;For later addition
080A	3C		INR A	;Increment A
080B	C3		JMP LOOP	;Loop back
080C	06			
080D	08			

Table 5-1. Counter Program Listing

EXPERIMENT 5-1

Running the Counter Program

CONCEPT

In this experiment you will enter and step through the program just described. FETCH REG (fetch register) is used to examine the registers used in the program. FETCH PC (fetch program counter) makes it easier to return to the program after it has been stopped.

PROCEDURE

I. Entering the Program

A) Press [FETCH ADRS] [0] [8] [0] [4].

B) Press [3] [E] [STORE INCR]. This stores the opcode 3E (the MVI A instruction) in location 0804.

C) Press [0] [STORE INCR]. This stores 00 in location 0805.

D) Continue entering the appropriate data (as shown in Table 5-1) until the entire program has been stored.

E) Press [DECR]. This allows you to check the previous location.

F) Repeat step E until you have verified that the entire program is correctly stored.

II. Stepping Through the Program Using Instruction Step

A) Press [FETCH ADRS] [0] [8] [0] [4] (you do not have to do this if the display already shows 0804). The MVI A instruction (opcode 3E) appears in the display.

B) Press [INSTR STEP] to execute the MVI A instruction (opcode 3E).

C) The opcode for the STA instruction (32) now appears in the display. Press [INSTR STEP]. What happened? The instruction sent all zeroes to the output port. The reason that the LEDs all came on is that they indicate negative logic. They are on for a zero and off for a one.

D) Press [INSTR STEP] three times. This executes the NOP, INR A and JMP instructions.

E) The STA opcode (32) now appears in the display again. The value in the accumulator has been incremented. The new value will be shown in the output port LEDs when this instruction is executed again. To see this happen, press [INSTR STEP].

EXPERIMENT 5-1

Continued

F) Press [INSTR STEP] repeatedly to see the loop repeat. If you remember that off means a one and on means a zero, you will see the output port LEDs count in binary.

G) Stop stepping when the NOP instruction (address 0809, opcode 00) appears in the display again. Press [FETCH REG]. This allows you to examine the processor's registers. The display shows an "A" and then a particular value. The data shown is the value that is in the accumulator (A-register) after the last program step. The data (when converted to binary) corresponds to the number shown in the output port LEDs. This is true because the data was just sent from the accumulator to the output port, and the display shows the contents of the accumulator. (When comparing the LEDs with the accumulator, do not forget that they indicate negative logic.)

H) Press [STORE INCR]. The display now shows the flag register (described in Lesson 12).

I) Press [STORE INCR] repeatedly until the display shows PCH. This is the high-order half of the program counter (08).

J) Press [STORE INCR]. This is the low-order half of the PC (09). The other registers you saw while stepping up to the PC are described in Lesson 11.

K) To continue stepping through the program, press [FETCH P C]. This recalls the last value of the program counter, which contains the address of the next instruction to be executed. The display now shows this address (0809). The monitor program saves (in the memory) the value of the program counter after each step, as described earlier in this lesson, and the [FETCH P C] key retrieves it.

L) Press [INSTR STEP]. Now you have resumed stepping through the program.

M) Step through the loop once more (using [INSTR STEP]) and stop at the same point (address 0809). Press [FETCH REG]. The value in the accumulator has incremented, as has the value shown in the output port LEDs.

N) Press [FETCH P C] and repeat step M to see the values increment again.

III. Stepping Through the Program Using Hardware Step

A) You can also step through the program and examine registers using [HDWR STEP] instead of [INSTR STEP]. Press [FETCH ADRS] [0] [8] [0] [4] [HDWR STEP]. The display goes blank since you are in the hardware step mode. The address 0804 now appears on the binary address LEDs. The data at that location, the opcode 3E, appears on the binary data LEDs. The opcode 3E specifies that this is an MVI A instruction. Remember that whenever you are using the hardware step mode, you must refer to the binary address and data LEDs.

B) Press [HDWR STEP]. This gets the second byte of the instruction, the data 00.

C) Press [HDWR STEP]. The instruction is now executed.

D) The STA opcode (32) now appears in the data LEDs. Press [HDWR STEP]. This gets the second byte of the instruction, the lower half of the port address (00).

E) Press [HDWR STEP]. The upper half of the port address (30) now appears in the data LEDs.

F) Press [HDWR STEP]. The entire instruction has been read from the memory, and now an extra step is needed to execute the instruction. The address LEDs show 3000, the port address, and the data LEDs show 00, the data to be sent to the port.

G) Press [HDWR STEP]. The instruction is now executed, and the output port LEDs are all on.

H) The program is now at the NOP instruction (00) which is the point at which you examined the accumulator in part II of this experiment. Press [RESET] to return to the monitor. The NOP instruction is executed, and the display shows the address of the next instruction (080A).

I) Press [FETCH REG] to view the contents of the accumulator.

J) To return to the program, press [FETCH P C] [HDWR STEP].

K) Press [HDWR STEP] repeatedly to go through the loop again, stopping at the same place (address 0809).

L) Repeat steps H, I, J, and K several times to see the accumulator increment. The value in the accumulator corresponds to that shown in the output port LEDs.

M) Press [RESET] to return the μLab to normal operation.

EXPERIMENT 5-1

Continued

SUMMARY

In this experiment the program to make the output port LEDs count in binary was loaded into the memory. Then you stepped through the program using both of the step modes. The FETCH REG key allowed you to examine the contents of the accumulator. The FETCH PC key was used to get back to the address at which stepping was interrupted. You could then continue stepping through the program. When using hardware step, you had to press RESET to get back to the monitor before the FETCH REG (or any other) key could be used.

Note that when using the hardware step mode, the STA instruction required an extra step to execute the instruction. One hardware step is required for each reference to memory or I/O ports. Thus, the STA instruction required three steps to read the three bytes of the instruction and a fourth step to write the data to the output port.

PROGRAM ORGANIZATION

Most microprocessor applications consist of a number of tasks. For example, a microprocessor-based voltmeter must read the input voltage and send data to the display. The processor needs to manipulate the data for ohms conversion and automatic zeroing. A sophisticated meter may have a keyboard for the user to enter specific requests, which the processor must read and interpret. For auto-ranging, the processor may have to set an attenuator. Each of these tasks is basically independent, and each can be performed by a relatively simple program. One main program can then coordinate all of the specialized programs without having to deal with the details of each task.

As an analogy, consider the structure of a large company. The president cannot do all the work, but he can make all of the major decisions. He does not need to worry about what is being served for lunch or what kind of mop the janitor should use, because other people can make these decisions. There may be several vice-presidents, each of whom has responsibility for a certain aspect of the company's operation. These people may have managers working for them, who in turn supervise the other employees.

Everyone in the company has a task to do, and the lower you are on the ladder, the more you are involved with the details. The president has the time to make the important decisions because he can call on other people to implement his decisions. They in turn call on others, who can call on additional people if required. Then each person reports to his or her supervisor, and eventually the president receives a response.

This is very similar to the operation of a microprocessor system. Because the tasks can be so varied and complex, there are usually many programs rather than just one. One program (often called the *executive*) acts as the president. The executive has other programs working for it, and they may in turn have other programs working for them.

Programs that work for other programs are called *subroutines* (or simply *routines*). Subroutines may have other subroutines working for them, like the multiple levels of a company.

AN EXAMPLE

Figure 5-3 shows a simple program that uses subroutines to flash the output port LEDs on and off repeatedly. One subroutine is used to turn them on, and another is used to turn them off.

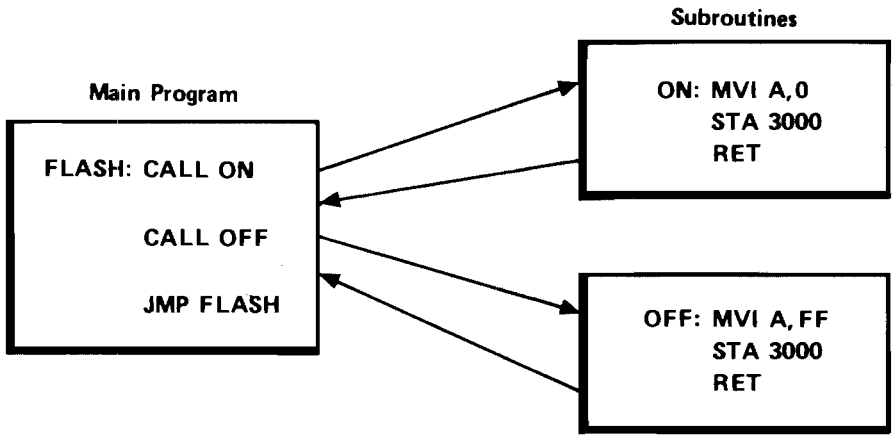


Figure 5-3. Using Subroutines to Flash the Output Port LEDs

The main program contains only three instructions: one to call the routine ON, one to call the routine OFF, and one to jump back to the beginning. The main program determines the overall function of the program, but it does not need to know that the port LEDs indicate negative logic, or even at which address the port is located. These details are handled by the subroutines.

The *CALL* instruction directs the microprocessor to jump to a subroutine. The label ON identifies the subroutine and is replaced with an actual address when the assembly language is converted to machine code. This program turns the LEDs on and then off and then repeats.

The first two instructions in each of the subroutines should be familiar to you. The last instruction, *RET* (Return), identifies the end of the subroutine and causes the microprocessor to jump back to the main program.

Table 5-2 shows the complete program with the machine code added. The CALL instruction consists of the opcode CD, followed by the subroutine address. The subroutine address is stored in the same way as a jump address. The RET instruction consists only of the opcode C9. No address is included in the instruction.

MAIN PROGRAM

Address	Contents	Labels	Instruction	Comments
0800	CD	FLASH:	CALL ON	;Turn on LEDs
0801	09			
0802	08			
0803	CD		CALL OFF	;Turn off LEDs
0804	0F			
0805	08			
0806	C3		JMP FLASH	;Repeat
0807	00			
0808	08			

SUBROUTINE TO TURN LEDS ON

0809	3E	ON:	MVI A,00	;Set accumulator to zero
080A	00			
080B	32		STA 3000	;Write accumulator con-
080C	00			tents to output port
080D	30			
080E	C9		RET	;Return

SUBROUTINE TO TURN LEDS OFF

080F	3E	OFF:	MVI A,FF	;Set accumulator to all
0810	FF			ones
0811	32		STA 3000	;Write accumulator con-
0812	00			tents to output port
0813	30			
0814	C9		RET	;Return

Table 5-2. LED Flasher Using Subroutines

You may be wondering how the microprocessor knows where to jump back to when it encounters the return instruction at the end of a subroutine. When a CALL instruction is executed, the return address (the address of the next instruction after the call) is stored in a special place in memory called the *stack*. When a return instruction is encountered at the end of the subroutine, the microprocessor gets the return address from the stack. The program flow then returns to the main program. The details of how the stack works and why it is used are described in Lesson 12. Fortunately, the operation of the stack is almost entirely automatic.

EXPERIMENT 5-2

Subroutines

CONCEPT

The LED flasher program just described is entered into the μLab and executed. You will see the program flow jump from the main program to the subroutines and then back to the main program.

PROCEDURE

A) Key in the program in Table 5-2.

B) Verify that the program is correctly stored.

C) Press [FETCH ADRS] [0] [8] [0] [0]. The display shows the first CALL instruction (opcode CD).

D) Press [INSTR STEP]. The CALL instruction is executed and the address jumps to 0809 (the start of subroutine ON).

E) Press [INSTR STEP] [INSTR STEP]. The output port LEDs go on, and the RET instruction (address 080E, opcode C9) appears in the display.

F) Press [INSTR STEP]. The return is performed, and the address jumps back to the main program. The second CALL instruction (address 0803, opcode CD) appears in the display.

G) Press [INSTR STEP]. The CALL instruction is executed and the display shows address 080F (the start of subroutine OFF).

H) Press [INSTR STEP] [INSTR STEP]. The LEDs go off and the RET instruction (address 0814, opcode C9) is shown in the display.

I) Press [INSTR STEP]. The return is performed and the display shows the last instruction of the main program (address 0806, opcode C3).

J) Press [INSTR STEP]. The program jumps back to the beginning (address 0800).

K) Press [INSTR STEP] repeatedly to see the program repeat.

SUMMARY

The three parts of the LED flasher program were entered into the μLab: the main program, the subroutine to turn the LEDs on, and the subroutine to turn the LEDs off. By stepping through the main program, you saw the CALL instruction cause the jump to the subroutine and the RET instruction cause the program flow to return to the main program. Even though the return instruction does not specify an address, the microprocessor knows that it must return to the instruction following the previous CALL.

INTERRUPTS

Sometimes the microprocessor system must react to events that are very infrequent and unpredictable. Returning for a moment to the company analogy, a company may expect certain events to happen, but the exact time that they will occur may be unknown. For example, everybody may be interrupted by a fire alarm and a planned course of action must be taken. After the fire is put out and the damage repaired, the people return to work and the old routine continues.

The same type of event may occur in a microprocessor system. For example, consider a microprocessor-based voltmeter with a "calibrate" button on the front panel. When this key is pressed, the microprocessor stops whatever it is doing and jumps to a calibration subroutine. After the calibration is performed, control returns to the program that was interrupted.

These types of actions are called *interrupts*. The microprocessor chip has an input that causes an interrupt. A high signal on this pin causes the microprocessor to stop whatever it is doing and jump to a special interrupt subroutine. This subroutine performs the task required by the interrupting device. A return instruction at the end of the interrupt routine causes the program flow to return to the interrupted program. There are five interrupt inputs on the 8085 to allow for many different interrupts. These are described in Lesson 10.

The INTRPT key on the μLab is connected to one of these interrupt inputs. Pressing this key causes the microprocessor to jump from whatever it is doing to a predefined location in RAM, where the interrupt subroutine (often called the *interrupt service routine*) is stored.

Because you sometimes want the microprocessor to ignore interrupts, they may be disabled by the program. For example, the interrupt generated by the INTRPT key is normally disabled by the system monitor program, since you would not want it to cause a jump to RAM if you had not stored a program there that tells it what to do.

USING THE μLAB'S INTERRUPT KEY

To illustrate this process, the counter program that appeared earlier in this lesson can be interrupted by the INTRPT key. To do so, instructions must be added to the beginning of the program to enable this interrupt input. Table 5-3 shows the listing of the interrupt enable routine.

Address	Contents	Instructions	Comments
0800	3E	MVI A,0D	;Put value for interrupt
0801	0D		mask in accumulator
0802	30	SIM	;Move accumulator to
			interrupt mask
0803	FB	EI	;Enable interrupts

Table 5-3. Interrupt Enable Routine

The *interrupt mask* is a special register in the microprocessor that determines which interrupts should be enabled. (The interrupt mask is described in Appendix B under the SIM instruction.) For now, you should realize that when the interrupt mask register is set to the hex value 0D, it causes the INTRPT key to be enabled. The program first puts this value into the accumulator, using the familiar MVI A instruction. Then the SIM (Set Interrupt Mask) instruction moves the contents of the accumulator into the interrupt mask register. The contents of the interrupt mask register tell the processor which interrupts should be enabled. Then, the next instruction, EI (Enable Interrupts), causes them to be enabled.

Once the interrupts have been enabled, an interrupt service routine is required. The monitor contains a subroutine that causes the speaker to generate a beep. Table 5-4 shows the listing of an interrupt service routine that calls the beep subroutine (located at address 0010).

Address	Contents	Instructions	Comments
0AFC	CD	CALL BEEP	;Jump to beep routine
0AFD	10		
0AFE	00		
0AFF	FB	EI	;Re-enable interrupts
0B00	C9	RET	;Return to counter program

Table 5-4. Interrupt Service Routine

The beep subroutine turns the speaker on and off at approximately 1 kHz, causing a beep to be generated. The beep routine ends with a return instruction, so that after a short beep, the program returns to the interrupt service routine. Figure 5-4 shows the sequence of events when an interrupt occurs. Figure 5-5 shows the detailed program flow.

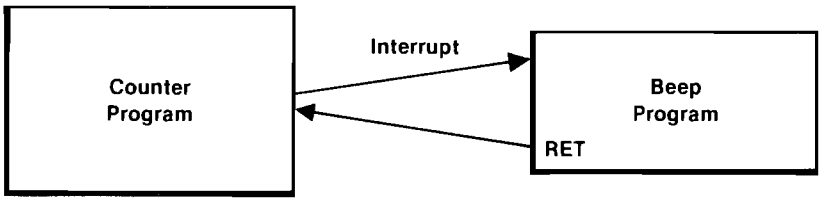


Figure 5-4 Simplified Sequence of Events When Counter Program is Interrupted

The EI instruction in this program is necessary because the microprocessor automatically disables interrupts when one is acknowledged. If the EI instruction were excluded, a beep would occur the very first time you pressed INTRPT, but after that the interrupt would be disabled.

The interrupt service routine ends with a return instruction. Therefore, after a beep is generated, control returns to the program that was interrupted.

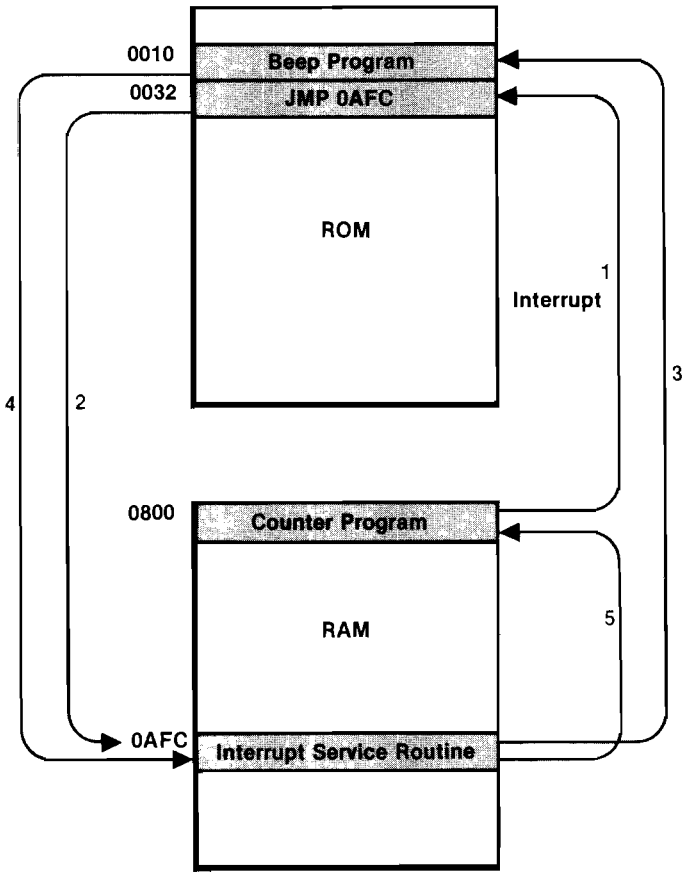


Figure 5-5. Detailed Program Flow When Counter Program is Interrupted

The address 0AFC (at which the interrupt service routine starts) is not arbitrary as are the locations of other user programs. The interrupt key forces the microprocessor to jump to address 0034. This address is specified by the design of the 8085 microprocessor and cannot be changed. This location (which is a ROM location in the μLab) contains the instruction JMP 0AFC. The address 0AFC is the RAM location at which the interrupt service routine is stored. You can verify that address 0034 contains this jump instruction by examining the contents of locations 0034, 0035, and 0036.

EXPERIMENT 5-3

Interrupts

CONCEPT

The counter program is run with the instructions added to enable the INTRPT key. An interrupt service routine is entered which calls the monitor's BEEP program. The counter program may then be interrupted to generate a beep by pressing the INTRPT key.

PROCEDURE

A) Key in the program in Table 5-1.

B) Key in the four instructions in Table 5-3 to enable the interrupts.

C) Key in the interrupt service routine in Table 5-4.

D) Verify that the routines are correctly stored.

E) Press [FETCH ADRS] [0] [8] [0] [0] [RUN]. This starts the counter program running. The output port is actually counting continuously, but it is going so fast that the output port LEDs all appear to be on.

F) Press [INTRPT]. The μLab leaves the counter program and executes the Beep routine as long as the [INTRPT] key remains pressed. The output port LEDs stay at their most recent state, since the counter program is not running (the interrupt service routine is). When [INTRPT] is released, the counter program continues.

G) Press [INTRPT] again. Since the counter program is stopped at a random point, the port LEDs should contain a different value. The value shown in the LEDs is whatever value the output port contained at the moment [INTRPT] was pressed.

H) Repeat step G a few times to see that it really is random.

I) Press [RESET] to stop the counter program.

SUMMARY

This experiment demonstrated the use of interrupts. When INTRPT was pressed, an interrupt was generated and the microprocessor jumped from the counter program to the interrupt service routine, which in turn called the beep routine in the ROM. As long as INTRPT was pressed, the beep routine was repeatedly executed. When the key was released, the microprocessor returned to the counter program.

REVIEW

Lesson 5

The microprocessor chip contains several registers. Some, such as the accumulator, are used for data storage. Others, such as the program counter (PC), provide specific control functions. The PC keeps track of which instruction is being executed.

The FETCH REG key on the μLab enables you to view the contents of the registers. Later in the course you will see how it can be used to change the contents of the registers.

The FETCH PC key restores the address in the display to the previous value of the program counter. It is useful for returning to a program after it has been stopped.

Subroutines are programs that are used by other programs. They allow the system to be divided into small, modular units. A CALL instruction is used to jump to a subroutine, and a RET instruction ends the subroutine. When a RET instruction is executed, the program flow returns to the program that called the subroutine. This allows the same subroutine to be used by different programs or at different places in one program.

Interrupts are used to service unpredictable events that require a quick response. When the interrupts are enabled, they allow hardware external to the microprocessor to request immediate action. When the microprocessor receives an interrupt signal, it stops whatever program it is executing and jumps to an interrupt service routine. When that routine is finished, it returns to the program that was interrupted.

QUIZ

Lesson 5

1. The accumulator's main purpose is:
   a. temporary data storage.
   b. keeping track of the next instruction to be executed.
   c. selecting which interrupts should be enabled.
   d. storing instructions.

2. The program counter is used for:
   a. data storage.
   b. instruction storage.
   c. storing the address of the instruction waiting to be executed.
   d. counting programs.

3. The FETCH REG key allows you to see the contents of the _____.

4. To jump to a subroutine, you use a _____ instruction.

5. To end a subroutine, you use a _____ instruction.

6. Interrupts are used primarily for:
   a. breaking a program into modular segments.
   b. responding quickly to unpredictable events.
   c. speeding up program execution.
   d. halting the system.

7. When an interrupt occurs, the microprocessor will:
   a. jump to the interrupt service routine.
   b. halt until another request is made.
   c. continue executing the main program.
   d. complete the current program and then stop.

LESSON 6

Inside the Microprocessor

INTRODUCTION

For most of this course, the microprocessor is treated as a *black box:* a device with known characteristics whose internal structure is of no concern. However, some knowledge of the internal operation of the microprocessor is helpful in obtaining a clear understanding of the system's operation. This lesson takes a brief look inside the microprocessor to see how programs are executed.

INSIDE THE 8085A

Figure 6-1 shows a simplified block diagram of the 8085 microprocessor. The accumulator connects to the data bus and the *Arithmetic and Logic Unit* (ALU). The ALU performs all data manipulation, such as incrementing a number or adding two numbers.

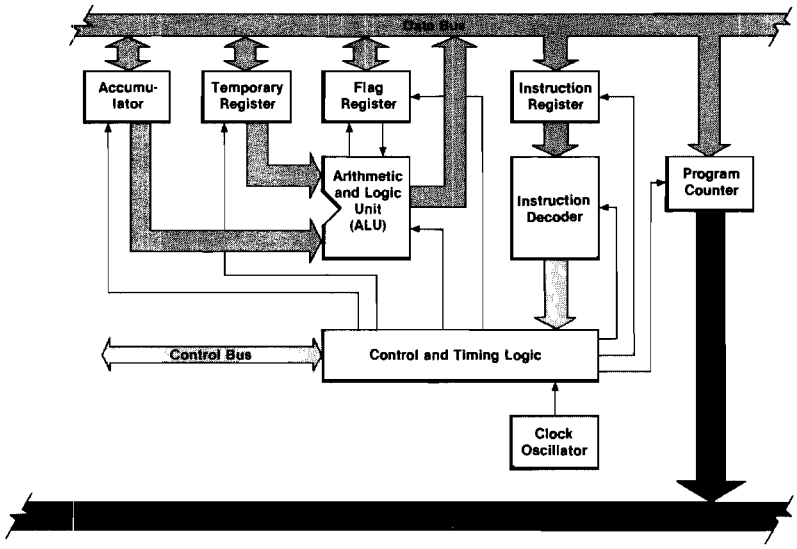


Figure 6-1. Simplified 8085 Block Diagram

The *temporary register* feeds the ALU's other input. This register is invisible to the programmer. It is controlled automatically by the microprocessor's control circuitry.

The *flags* are a collection of flip-flops that indicate certain characteristics of the result of the most recent operation performed by the ALU. For example, the *zero flag* is set if the result of an operation is zero. The zero flag is tested by the JZ instruction.

The *instruction register, instruction decoder, program counter* (PC), and *control and timing logic* are used for fetching instructions from memory and directing their execution. For example, suppose that an instruction is about to be read from location 0200. First the opcode must be read from memory; this is the instruction fetch, as shown in Figure 6-2. The PC, which contains the address 0200, is output to the address bus and causes memory location 0200 to be selected. The ROM will then place the contents of location 0200 (presumably an opcode) on the data bus, and the microprocessor will store the opcode in the Instruction Register.

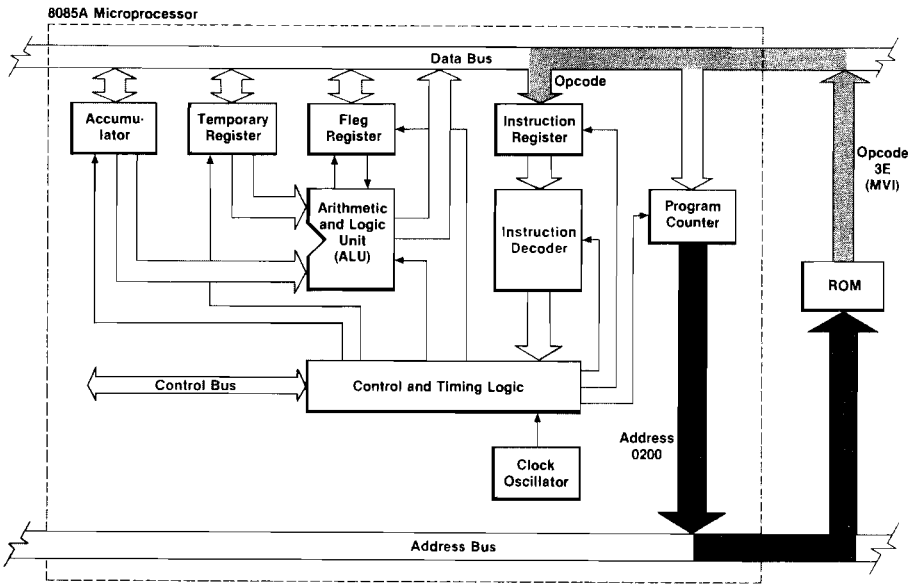


Figure 6-2. Reading the Opcode from Memory for a MVI A Instruction

THE INSTRUCTION CYCLE

The instruction register feeds the instruction decoder, which recognizes the opcode and provides control signals to the timing and control circuitry. The timing and control circuits are like a processor within the processor. A ROM within the microprocessor IC contains the *microcode* (or *microprogram*), which tells the processor exactly what to do to execute each machine language instruction. The microcode, which is part of the design of the microprocessor and generally cannot be changed, defines the microprocessor's machine language. Writing microcode (which is usually done by the microprocessor manufacturer) is called *microprogramming* and should not be confused with writing programs to be executed by the microprocessor.

For example, for a MVI A instruction, the control and timing logic first reads the opcode 3E, and then increments the address in the PC. The instruction decoder determines that this opcode is followed by a byte of data, so the contents of the memory location pointed to by the PC are read into the accumulator (see Figure 6-3).

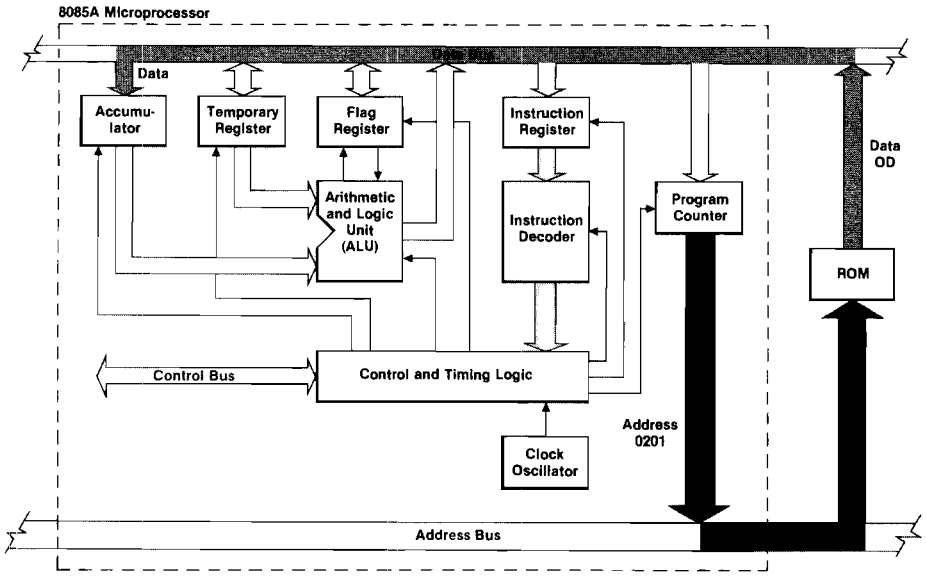


Figure 6-3. Reading the Data for the MVI A Instruction

The microprogram will now indicate to the control logic that the instruction is completed. The PC is incremented, and the next byte of the program (the next opcode) is read into the instruction register. The execution of this instruction then begins.

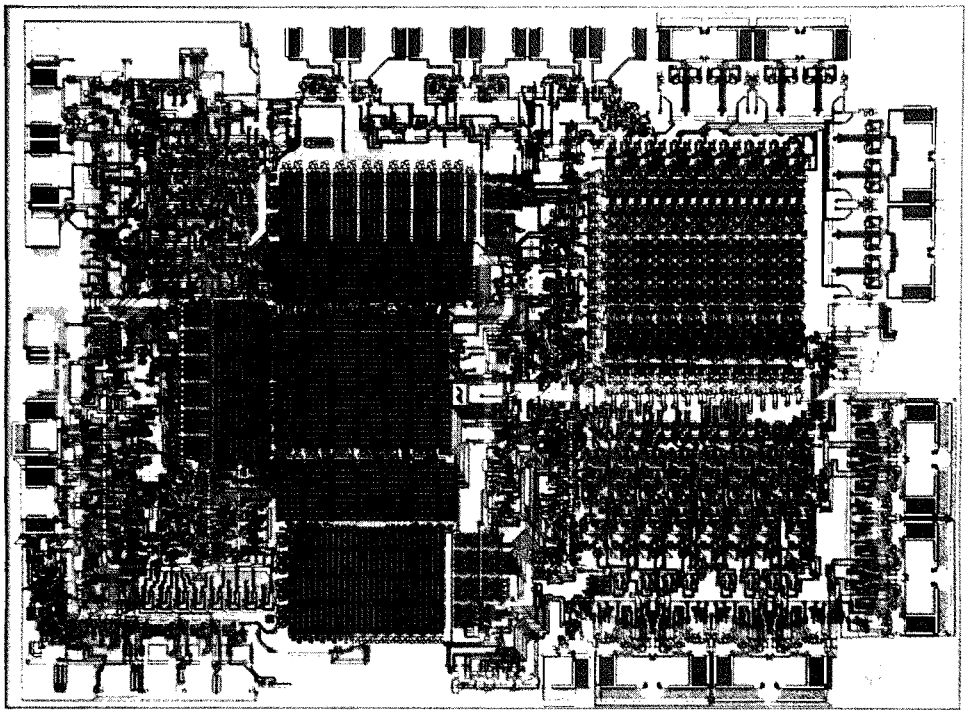
This repetitive sequence performed by the microprocessor is called the *fetch-execute cycle.*

INSTRUCTION EXECUTION

In the execute phase of the instruction the real work is done. There are four basic types of operations that can be performed by the 8085:

1. Read data from memory or an input port.
2. Write data to memory or an output port.
3. Do an operation internal to the microprocessor.
4. Transfer control to another memory location.

The first two types are self-explanatory. The third, internal operations, involves manipulating the registers (such as the accumulator) without accessing the memory or I/O ports. For example, the contents of one register may be moved to another register, or the contents of a register may be incremented or decremented. The fourth group includes instructions such as JMP, CALL, and RET.



Photomicrograph of 8085 Microprocessor. *The chip is approximately 0.2 inches square, but contains over 20,000 transistors.*

MACHINE CYCLES

The fetching and execution of instructions is divided into *machine cycles.* The first machine cycle of every instruction is the opcode fetch. An additional machine cycle is then required for each memory or I/O reference to provide time for the data transfer. A machine cycle consists of setting the address on the address bus and then transferring information over the data bus. Most operations internal to the microprocessor (such as incrementing the accumulator) are completed in the same machine cycle as the opcode fetch. A simple instruction such as INR A thus requires only one machine cycle, while STA requires four cycles: three to read the instruction and one to write the accumulator to memory.

The μLab's hardware step key causes one machine cycle to occur each time it is pressed. An extra step (after reading all bytes of an instruction) is required for executing instructions that access the memory or I/O ports. This step is needed by the microprocessor to set up the address bus and transfer data over the data bus. Internal operations, which are executed in the same step as the instruction fetch, do not require this extra step for the execute phase.

PROGRAM EXECUTION

In general, the microprocessor keeps reading sequentially through the memory, one location after another, performing the indicated operations. Exceptions to this occur when a jump, call, or return instruction is executed. Another exception is the occurrence of an interrupt. Any of these events will cause the microprocessor to interrupt the sequential flow and begin executing instructions from another address.

Note that opcodes and data are intermixed in memory. One address might contain an opcode, the next two a jump address, the next one an opcode, and the next one a piece of data. It is the programmer's responsibility to be sure that the memory contains a valid sequence of opcodes and data. The microprocessor can distinguish between them only by their context. The opcodes, jump addresses, and data are all simply bit patterns stored in the memory. All such information is read in exactly the same way, and it all travels over the same data bus. The microprocessor must always keep track of whether it is reading an opcode or data and treat each appropriately. The processor assumes that the first location it reads contains an opcode and goes from there. If the opcode requires a byte of data, the microprocessor "knows" (from the instruction decoder) that the next byte is data and treats it accordingly. It then assumes that the byte following the data is the next opcode. If data is misinterpreted as an opcode, the system usually goes completely out of control *(crashes)*.

EXPERIMENT 6-1

Bus Operation

CONCEPT

The program used in Experiment 4-3, which reads data from the input port and writes it to the output port, is entered and then stepped through using the hardware step mode. This enables you to follow the fetch-execute cycle for each instruction and see the operation of the buses using the individual LEDs on the address bus, data bus, and status lines. At each step, you will enter the information from the LEDs into a table. This table will then show the step-by-step execution of the program.

PROCEDURE

A) Key in the following program:

Address	Contents	Label	Instruction	Comments
0900	3A	START:	LDA 2000	;Read input port
0901	00			
0902	20			
0903	32		STA 3000	;Send data to
0904	00			output port
0905	30			
0906	C3		JMP START	;Loop back
0907	00			
0908	09			

B) Verify that the program is correctly stored.

C) Press [FETCH ADRS] [0] [9] [0] [0] [HDWR STEP]. This puts the μLab into the hardware step mode at the beginning of the program.

D) The address LEDs now indicate the first address of the program (0900), and the data LEDs show the first opcode (3A). The status LEDs indicate that a READ from the RAM is occurring. Note that the information from the three sets of LEDs appears in the first line of Table 6-1, and the following line contains the same information converted to hexadecimal.

E) Press [HDWR STEP]. The second byte of the LDA instruction (the low-order half of the port address, 00) is read from the RAM. Enter the information from the three sets of LEDs into the third line of Table 6-1. Convert the address and data to hex, and enter the hex numbers on the following line.

F) Press [HDWR STEP]. The last byte of the LDA instruction (the high order half of the port address, 20) is now read. Enter the information in Table 6-1.

	Address																Data								Read	Write	ROM	RAM	In Port	Out Port
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0						
Binary	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	1	0	0
Hex	0				9				0				0				3				A				Read from RAM					
Binary																														
Hex	0				9				0				1				0				0				R.F.RAM					
Binary																														
Hex	0				9				0				2				2				0				R.F.RAM					
Binary																														
Hex	0				9				0				3				3				2				R.F.RAM					
Binary																														
Hex	0				9				0				4				0				0				R.F.RAM					
Binary																														
Hex	0				9				0				5				3				0				R.F.RAM					
Binary																														
Hex	0				9				0				6				C				3									
Binary																														
Hex	0				9				0				7				0				0									
Binary																														
Hex	0				9				0				8				0				9									
Binary																														
Hex																														
Binary																														
Hex																														

Table 6-1. Program Sequence Table for Experiment 6-1

Continued

G) Press [HDWR STEP]. This is the execute phase of the LDA instruction. The address LEDs indicate the port address 2000. Set the input port switches to any position. Since the port is being read, the data from the switches appears on the data bus LEDs. Note that the status LEDs indicate that a READ from the INPUT port is occurring. Make the entry into Table 6-1.

H) Press [HDWR STEP]. Enter the address, data, and status into Table 6-1.

I) Repeat step H until the program jumps back to the beginning (address 0900). Follow the operation of the program in the binary LEDs, noting the execute phase of the STA instruction.

J) If you want to verify the information in the table, repeat the procedure. Study the information in Table 6-1 to identify the fetch-execute sequence for each instruction. Which instruction does not require an extra cycle for execution?

K) Press [RESET] [FETCH ADRS] [0] [9] [0] [0].

L) Now press [INSTR STEP] repeatedly to step through the program. Note that all parts of each instruction are read from memory and executed in one step. The binary address, data, and status LEDs are not meaningful in this mode. Compare the sequence of addresses shown in the display with those in Table 6-1.

SUMMARY

By following the address, data, and status LEDs while using the hardware step mode, you can see the detailed execution of the program. You should be able to correlate the data you entered into Table 6-1 with the program listing in step A. Each instruction requires a machine cycle for each byte of the instruction. The STA and LDA instructions also have a separate execute cycle. Other instructions, such as the JMP, are executed in the same cycle as the reading of the instruction. The hardware step mode allows you to see each of these cycles. The instruction step mode combines all the machine cycles of an instruction into one step. Also, note that the data bus is used for data transfer in both directions, and the address bus is used for both memory addresses and port addresses.

Lesson 6

Programs are stored in memory as a sequence of binary numbers. The microprocessor executes the program by reading the first memory location and interpreting it as an opcode. If the opcode implies that the next one or two bytes contain data or an address, then the microprocessor reads the information from those locations. After all the bytes of the instruction are read, the microprocessor executes the instruction. The byte in the next memory location is then read and interpreted as an opcode, and the process repeats. This is the instruction cycle, also called the fetch-execute cycle. The μLab's instruction step mode steps an entire instruction cycle at a time and is used for following the overall flow of the program. The hardware step mode is used for observing the operation of each instruction in detail and shows each machine cycle.

The operations that can be performed by the 8085 microprocessor consist of transferring data between the microprocessor's registers and the memory or I/O ports, or modifying the contents of the registers in the processor. Thus, data is generally brought into the microprocessor from memory or I/O, manipulated within the microprocessor, and then sent back to memory or I/O.

QUIZ

Lesson 6

1. Reading an opcode from memory is called the instruction _____ .

2. The microprocessor knows which bytes to interpret as opcodes because:
   a. every byte is an opcode.
   b. every third byte is an opcode.
   c. each opcode implies the number of information bytes that follow it.
   d. the programmer must specify which bytes are opcodes.

3. To see the detailed operation of each instruction, you would use the _____ step mode.

4. How many "hardware steps" (machine cycles) are required to execute an instruction?
   a. The number of bytes in the instruction.
   b. The number of bytes in the instruction plus three.
   c. The number of bytes in the instruction plus the number of memory or I/O references required to execute the instruction.
   d. Three steps are always required.

5. The purpose of the ALU is to:
   a. interpret the opcodes.
   b. perform arithmetic and logical operations.
   c. control the address bus.
   d. calculate the number of machine cycles required.

III

MICROPROCESSOR SYSTEM HARDWARE

This section discusses microprocessor hardware in some detail. The first lesson introduces basic microprocessor circuitry and is recommended reading for all students. Subsequent lessons elaborate on the hardware concepts of address decoding, memories, peripherals, system control, and electrical considerations. Although the μLab design is used as the center of discussion, alternative designs are also described.

LESSON 7

Basic Hardware Concepts

INTRODUCTION

This lesson describes basic microprocessor system hardware. Bus structures and address decoding are discussed. The emphasis is on understanding the fundamental parts of a typical system, rather than considering a variety of design possibilities. Subsequent lessons describe actual microprocessor system circuitry in greater detail.

THE BUS CONCEPT

Microprocessor systems are designed around buses, which are not usually found in traditional random logic designs. In a microprocessor system many devices must exchange data with the processor. Figure 7-1 shows how this can be done using traditional design techniques. The processor must have a set of data outputs for each device and a multiplexer to select a particular device for data input. This method very quickly gets unwieldly as more and more devices are added. The data paths commonly carry eight bits of data, so each path requires

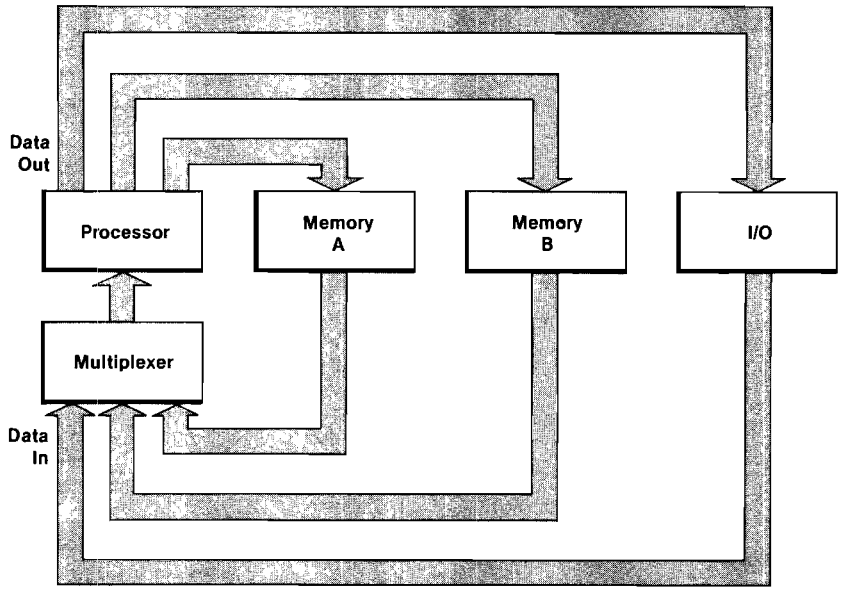


Figure 7-1. Data Exchange Using Traditional Design Techniques

eight lines. Therefore, for the simple three-device system shown, 48 lines are required: 24 for data input and 24 for data output. A more complicated system might have dozens of memory devices and I/O ports and require hundreds of interconnecting lines.

A solution to this interconnection problem is the use of a bus, as shown in Figure 7-2. Note how much simpler the interconnections are. A single set of eight lines is used to interconnect all the devices, and the same set of lines is used for data traveling into or out of the processor. This structure can be expanded indefinitely with little increase in interconnection complexity. A consequence of this technique is that, since all devices share the same data lines, only one may supply data at any given time. The address and control lines (driven by the microprocessor) provide the necessary control to select a particular device.

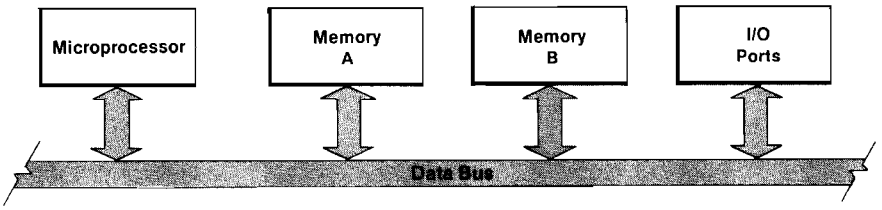


Figure 7-2. Data Exchange Using a Bus to Reduce the Number of Interconnecting Lines

THE THREE-STATE BUS

The *three-state driver* makes the shared data bus possible. For the sake of simplicity, a single-line bus is discussed first. However, the concept is exactly the same regardless of the number of lines in the bus. (A typical data bus has eight lines.)

The *three-state bus* is like a telephone party line. The bus can have many talkers and many listeners connected to it. Figure 7-3 shows a digital circuit bus with four talkers (three-state drivers) and two listeners (ordinary gates). The control logic

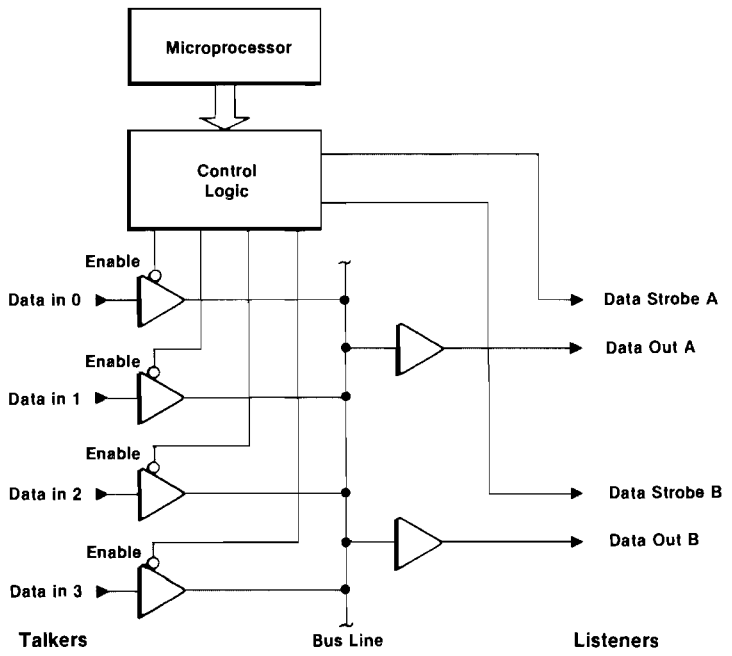


Figure 7-3. Three-State Single-Line Bus with Four Talkers and Two Listeners

chooses only one driver (talker) to be active at any given time. If more than one talker were enabled, the data on the bus would be meaningless. When a driver is enabled, the data at its input is placed on the bus. All of the other drivers are disabled. Their outputs are in a high-impedance (floating) state, so they have no effect on the logic state of the bus. (For an explanation of the symbology used in this course, refer to Appendix D.)

There can be many listeners on the bus. Since all they do is listen, more than one of them can be enabled at the same time. In general, however, the data on the bus is intended for one of them in particular. The control logic generates signals (data strobes) to tell selected listeners that the data on the bus is intended for them. A data strobe can be used, for example, to clock the data from the bus into a flip-flop. The inputs to the control logic are the address and control buses coming from the microprocessor, (see Figure 7-4.)

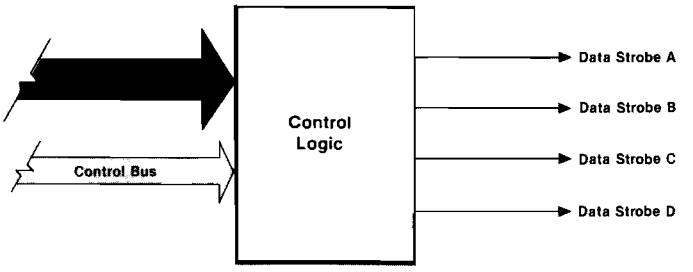


Figure 7-4. Control Logic Selects Device to be Involved in Data Transfer

The devices just described are *unidirectional.* They are either talkers or listeners, but not both. *Bidirectional* devices also exist, which are both talkers and listeners. Figure 7-5 shows a bus with two talker/listeners. For the sake of simplicity, only two have been shown, but there could be many more. Each talker/listener is provided with two control signals: the output enable signal for the three-state driver and the data strobe for the input. An example of a bidirectional device is a RAM, which can read and write data.

As an example of how this process works, suppose device A in Figure 7-5 must send a piece of data to device B. The control logic sets output enable A true

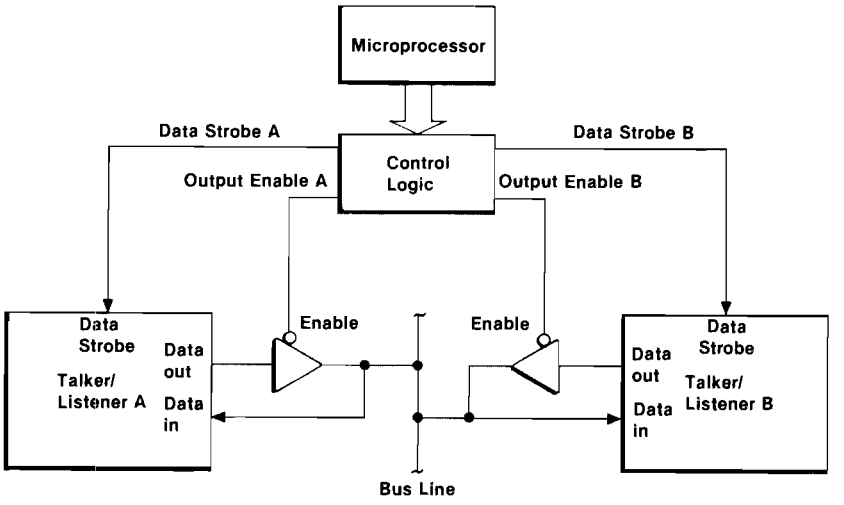


Figure 7-5. Bidirectional Talker/Listeners Connected to Bus Line

(enabled) and output enable B false (not enabled). Then, after enough *settling time* has elapsed for the data to reach device B's data input, the controller sends a pulse on the data strobe B line. This causes device B to read the data from the bus, which was supplied by device A. Note that many other devices can be connected to the bus. As long as their enables are false, they have no effect.

THE DATA BUS

The microprocessor system's data bus is a bidirectional, three-state bus. It is the same as a single-line bus except that there are eight lines instead of just one. To utilize all the data bus lines, each talker must have eight drivers (one for each line) and each listener must have eight inputs. The microprocessor and RAM are talker/listeners. Input ports are talkers that take inputs from outside the system and put them on the bus. Output ports are listeners that take data off the bus and send it outside the system. The ROM is only a talker.

Figure 7-6 shows how these devices communicate with the data bus. The microprocessor, ROM, RAM, and input ports contain three-state drivers on their outputs. The *Chip Select* (CS) inputs enable the drivers and cause the data from the selected device to appear on the data bus.

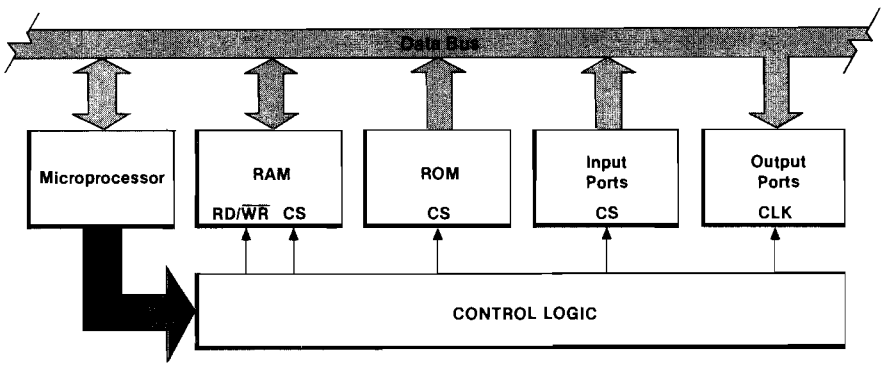


Figure 7-6. Devices with Three-State Outputs Communicate with Microprocessor Through Data Bus

The microprocessor acts as the controller for the system. It ensures that no more than one device is trying to use the bus at any given time. If the microprocessor wants to read data from the ROM, it first disables its own data outputs and then generates the control signals required to enable the ROM. The ROM's outputs then appear on the data bus, and the microprocessor reads the data. Reading the RAM or the input port is done in a similar manner.

To write data to a device (such as the RAM or output port), the microprocessor first places the data to be written onto the data bus. It then generates control signals that send a *write* pulse to the appropriate device. The write pulse causes that device to internally latch the data.

In general, data flows through the microprocessor. For example, to transfer data from the input port to the RAM, the microprocessor first reads the data from the input port and then writes it to the RAM. Because data cannot be transferred directly from the input port to the RAM, it must be temporarily stored within the microprocessor.

To summarize, the data bus is used for all transfers of data within the microprocessor system. All devices share the same bus. The control logic, operating from signals generated by the microprocessor, directs each device as to when it should place data on the bus or read data from the bus.

There are eight data bus lines in the μLab. The 8085 (the microprocessor in the μLab) is capable of handling eight bits of data at a time and is therefore called an eight-bit processor. Other microprocessors exist that handle more or less data. Many of the early microprocessors used a four-bit data bus and some of the newer devices use a sixteen-bit data bus.

When no devices are enabled, the data bus floats high and the data bus LEDs on the μLab will light up. The data bus LEDs appear to be on most of the time (whenever a program or the μLab monitor is running) because the data bus spends much of its time in the high impedance state.

THE ADDRESS BUS

You have seen how the data bus is used by many devices to exchange data. Now a method is needed by which the microprocessor can select the particular device that communicates with the data bus. The *address bus* (in conjunction with the control bus) provides this function.

Since the address bus is unidirectional, its operation is simpler than the data bus. Every memory location (and I/O port) has a unique address. Before any data transfer can take place (via the data bus), the microprocessor must output an address. The address specifies the exact memory location (or I/O port) which the processor wishes to access. In this way the microprocessor can select any part of the system with which it must communicate.

The 8085's address bus has sixteen lines, allowing direct addressing of 2^{16}=65,536 memory locations and I/O ports. These lines are referred to as A0, A1, A2, . . . A15, with A0 being the least significant bit.

ADDRESS DECODERS

The *address decoder* is a part of the control logic. It generates device select signals when a certain address (or range of addresses) is present on the address bus. For example, Figure 7-7 shows an address decoder for address 3000 Hex

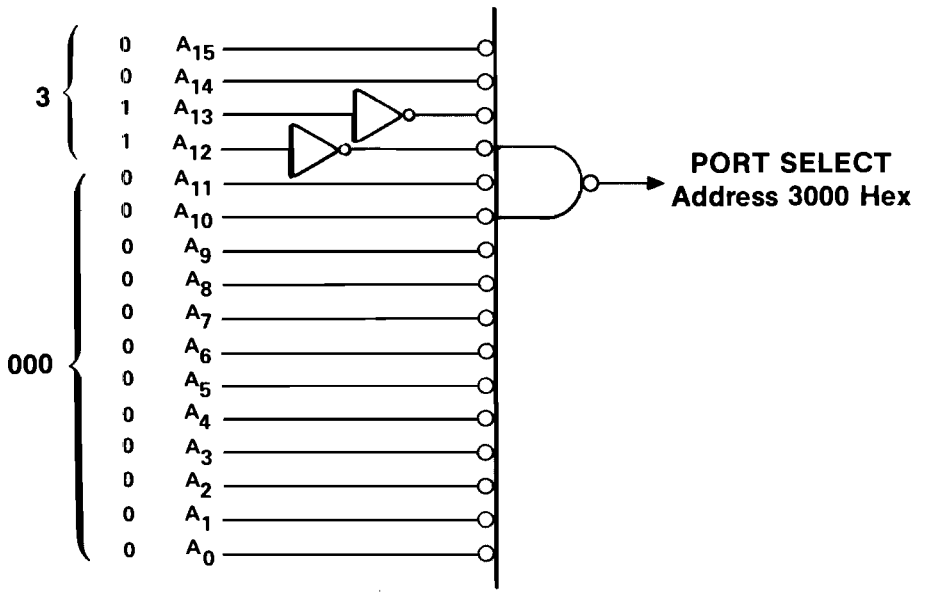


Figure 7-7. Address Decoder Configured to Control Port Assigned to Address 3000

(0011 0000 0000 0000 binary). The output of this decoder is true (logic 0) only when this exact address is present on the address bus. This output is then used to enable the port that is assigned address 3000.

THE CONTROL BUS

You have seen how the address bus is used to select a particular memory location or I/O port and how the data bus carries the data. The entire process is coordinated by the control bus, consisting of a number of control signals, most of which are generated by the microprocessor (a few are inputs to the processor). In this lesson only the signals that control the reading and writing of I/O ports and memory are discussed. Other control signals, used for accommodating interrupts, slow memories, and direct memory access, are discussed in Lesson 10.

The two main control signals generated by the 8085 are $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$. If $\overline{\text{READ}}$ is low, it indicates that a read operation is in progress, and the microprocessor signals the addressed device to place data on the data bus. If $\overline{\text{WRITE}}$ is low, then a write operation is in progress, and the microprocessor puts data on the data bus and signals the addressed device to store this data. The READ and WRITE status LEDs on the μLab are connected to these signals. The LEDs are on when the corresponding signal is true (low).

The major difference between the control bus and the address and data buses is that each wire in the control bus has a unique function. For the address and data buses, each line carries the same type of information (one bit of the address or data).

Keep in mind that we are describing the 8085's control signals and that other microprocessors may differ. The data transfers are the same, but they can be achieved in different ways. Refer to Lesson 20 for a description of other microprocessors' control signals.

OUTPUT PORTS

Figure 7-8 shows an output port latch with an assigned address of 3000. The latch is clocked whenever address 3000 is present on the address bus (as indicated by the address decoder) and a low-to-high transition occurs on the $\overline{\text{WRITE}}$ control signal. When the latch is clocked, the data from the data bus is stored in it. The microprocessor can therefore cause data specified by a program to appear at the output of the latch by writing the data to address 3000.

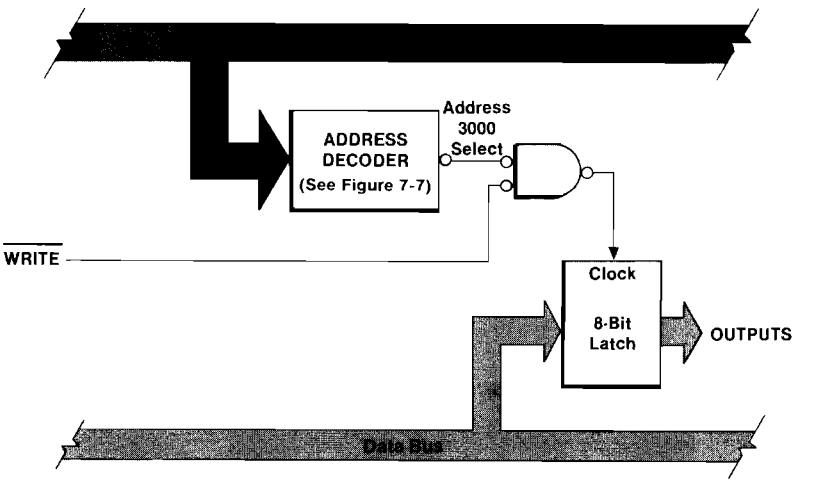


Figure 7-8. Data from Data Bus Stored in Latch Whenever Microprocessor Writes to Address 3000

INPUT PORTS

Input ports are connected in a similar manner, as shown in Figure 7-9. The output of the address decoder is ANDed with $\overline{\text{READ}}$ instead of $\overline{\text{WRITE}}$ to generate the port enable. The input port is an eight-line three-state driver which places the input signals on the data bus when enabled. The microprocessor can read the input signals by performing the read operation from the appropriate address. The processor then stores this data in one of its internal registers.

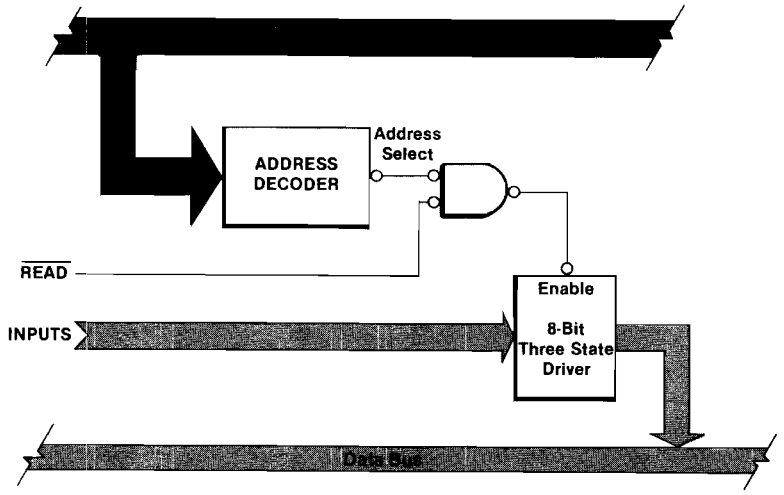
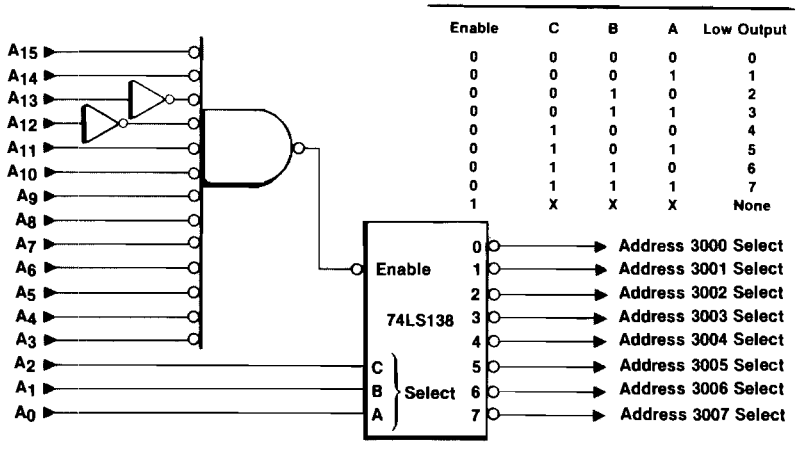


Figure 7-9. Input Data Placed on Data Bus Whenever Microprocessor Reads Address Assigned to Three-State Driver

ADDRESS DECODING FOR MULTIPLE DEVICES

Suppose that an address decoder is required to control eight I/O ports instead of just one. Eight address decoders similar to the one in Figure 7-7 could be used, but there is a simpler method. Figure 7-10 shows an address decoder which generates select signals for addresses 3000, 3001, 3002, . . . 3007. For these eight addresses, only the three low-order address bits (A0, A1, and A2) of the 16-bit address are changed. The upper thirteen bits can therefore be decoded by a common circuit similar to the one in Figure 7-7. The output of this circuit is used to enable a decoder such as a 74LS138. This decoder then generates eight separate outputs, one for each possible combination of A0, A1, and A2. The decoder is disabled (all outputs are false) if the upper thirteen address bits



Enable	C	B	A	Low Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	X	X	X	None

Figure 7-10. Decoder IC Provides Simple Way of Extending Number of Devices That Can Be Selected

are not of the specified value. In most cases, you do not need to decode the entire sixteen bits of the address to select a specific device. All sixteen bits are shown in the examples to emphasize that the system does have this amount of addressing capability if required.

ADDRESS DECODING FOR MEMORIES

Address decoding for memories is similar to that used for a group of I/O ports. For example, think of a ROM as a device with hundreds of eight-bit input ports on a single chip, with one port for each memory location. When the ROM is programmed, the ROM memory locations are permanently set to a desired pattern of ones and zeroes. For a RAM, each memory location can be thought of as having both an input and an output port tied together.

Suppose an address decoder is required for a small ROM containing 256 bytes. Figure 7-11 shows a circuit that accomplishes this. The low-order eight bits of the address bus connect directly to the ROM. The ROM has an internal address decoder that selects one of the 2^8=256 locations. The high-order eight bits of the address bus are decoded by an external address decoder to enable the ROM when its particular range of addresses is present on the upper half of the address bus. Notice that all sixteen address bits are decoded: half by the address decoder in the ROM and half by the external address decoder. The $\overline{\text{READ}}$ signal is ANDed with the address decoder output to generate the ROM enable. This is identical to the technique used for input ports.

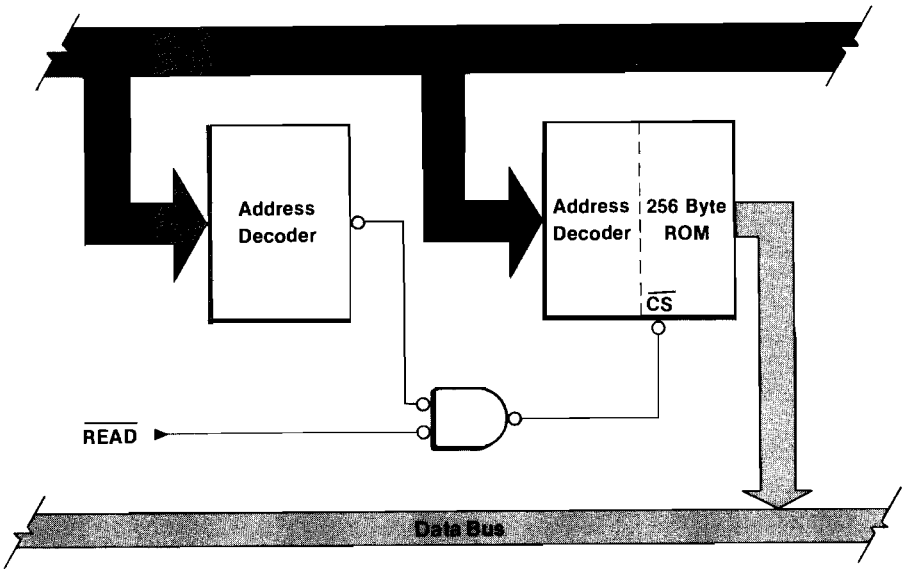
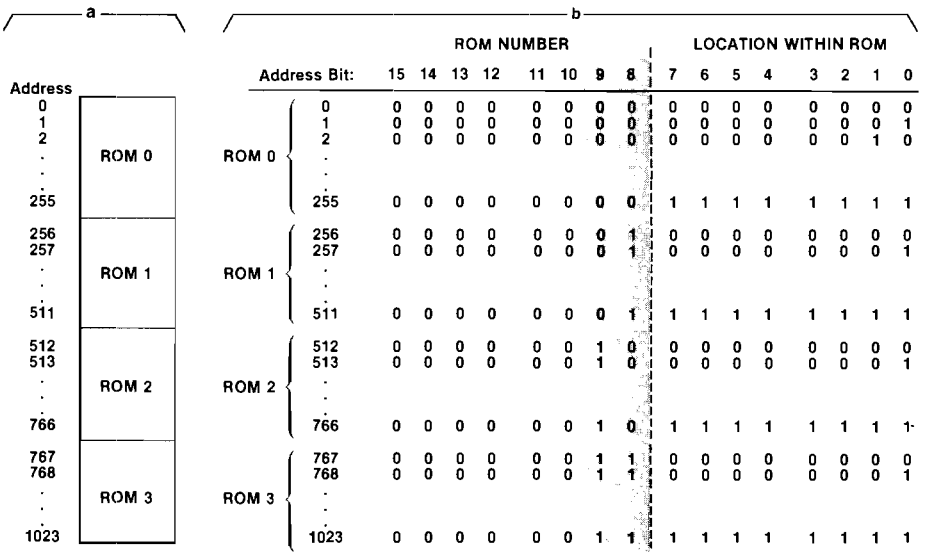


Figure 7-11. Internal Address Decoder in ROM Reduces Number of Address Lines Needed by External Address Decoder.

CONTROLLING MULTIPLE MEMORY DEVICES

Because most microprocessor systems use more than one memory chip, they need a more complex address decoder. Suppose that four of these 256 byte ROMs were connected with the addresses assigned to each chip as shown in Figure 7-12a.

The address lines must now indicate which memory chip should be selected and which word within that chip should be addressed. Figure 7-12b shows the addresses in binary, so that you can see how to deal with each individual bit. The lower eight bits of address specify the location within each chip, and the upper eight bits specify which chip is being addressed.



Figures 7-12 a and b. Addresses Assigned to Each of Four 256 Byte ROMs in a System

Observe that only bits A8 and A9 vary in decoding one chip from another. The reason that only these two address bits vary is that it takes exactly 10 bits to decode 1024 addresses (2^{10}=1024). A9 is the tenth bit. The four possible combinations of A8 and A9 therefore specify one of four blocks of 256 addresses each (the size of each ROM). The full 16-bit address bus is shown because the microprocessor has the capability of addressing up to 64K locations. However, this full capability is seldom used.

Figure 7-13 shows how this addressing is implemented. The lower eight bits of address go directly to the address lines of all four ROMs, since these bits specify the location within the chip. The address decoder then looks at the upper eight bits of address and generates the chip selects. The two least significant bits (of the upper half), A8 and A9, are used for the binary inputs to the decoder. The rest of the high-order address bits are used to enable the decoder only when they are

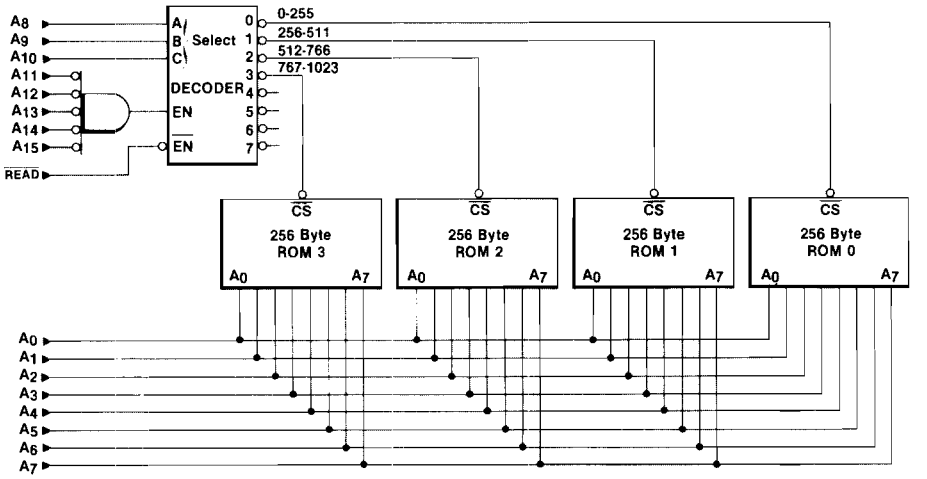


Figure 7-13. Address Decoding for Four 256 Byte ROM Example in Figure 7-12

all low. Study this diagram and the address-mapping table in Figure 7-12b; you should be able to see how they correspond. Notice that the $\overline{READ}$ control signal is used as a decoder enable. This is equivalent to ANDing this signal with each of the decoder's outputs.

Although there are numerous variations to this approach, this is a complete and straightforward address decoding technique. It serves to illustrate the basic principle: the low-order address bits are sent directly to the memory's address lines, and the high-order bits are decoded to generate the chip selects. No more than one chip can be selected at any given time. Other designs vary in the number of bits fed directly to the memory devices (a function of the number of words in each memory chip) and in the way the high-order bits are decoded to generate the chip selects.

RAM CONTROL

RAMs are decoded in a manner similar to ROMs, but with some extra control circuits to write (input to the RAM) as well as read (output from the RAM). RAMs have a $\overline{WRITE}$ input in addition to the $\overline{CS}$ (chip select) input. Figure 7-14 shows the truth table for the RAM control. $\overline{CS}$ must be low for either a read or write to take place. If $\overline{WRITE}$ is high (not true) when $\overline{CS}$ is low, the RAM outputs data to the data bus so that the processor can read it. To do this, $\overline{CS}$ enables the RAM's three-state output drivers. If $\overline{WRITE}$ is low, $\overline{CS}$ does not turn on the RAM's output drivers. Instead, the data on the data bus is stored in (written into) the memory at the location specified by the address bus.

$\overline{CS}$	$\overline{WR}$	FUNCTION
0	0	WRITE
0	1	READ
1	X	NO OPERATION

0 = LOW
1 = HIGH
X = DON'T CARE

Figure 7-14. Truth Table for Controlling RAM

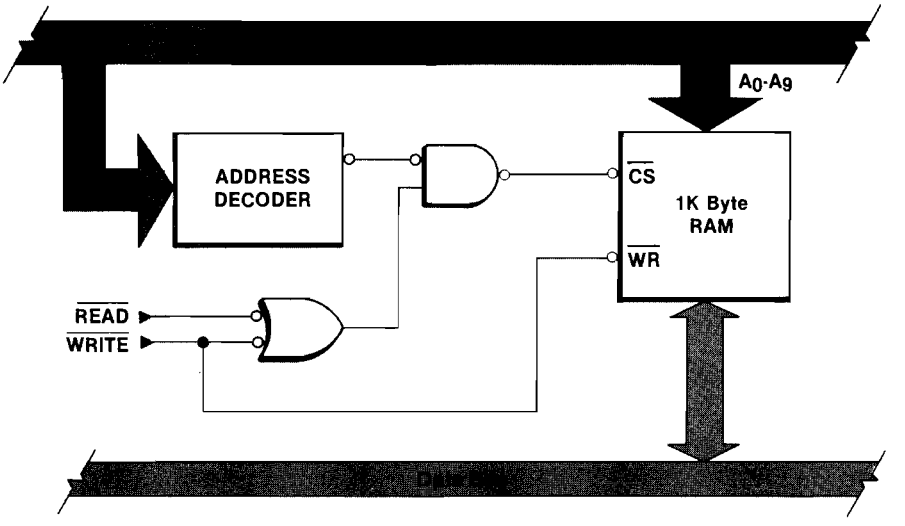


Figure 7-15. Address Decoding and Control for 1K Byte RAM Using Truth Table in Figure 7-14

A circuit to perform the desired gating is shown in Figure 7-15. $\overline{CS}$ is low if the $\overline{RAM}$ address select and either $\overline{READ}$ or $\overline{WRITE}$ are low. The $\overline{WRITE}$ line is connected directly to the RAM's $\overline{WRITE}$ input. The $\overline{WRITE}$ input is internally gated with the $\overline{CS}$ input, so that it is ignored unless $\overline{CS}$ is low.

Figure 7-16 is a block diagram of the μLab. It shows the major portions of the system and the communication paths between them that were discussed in this lesson.

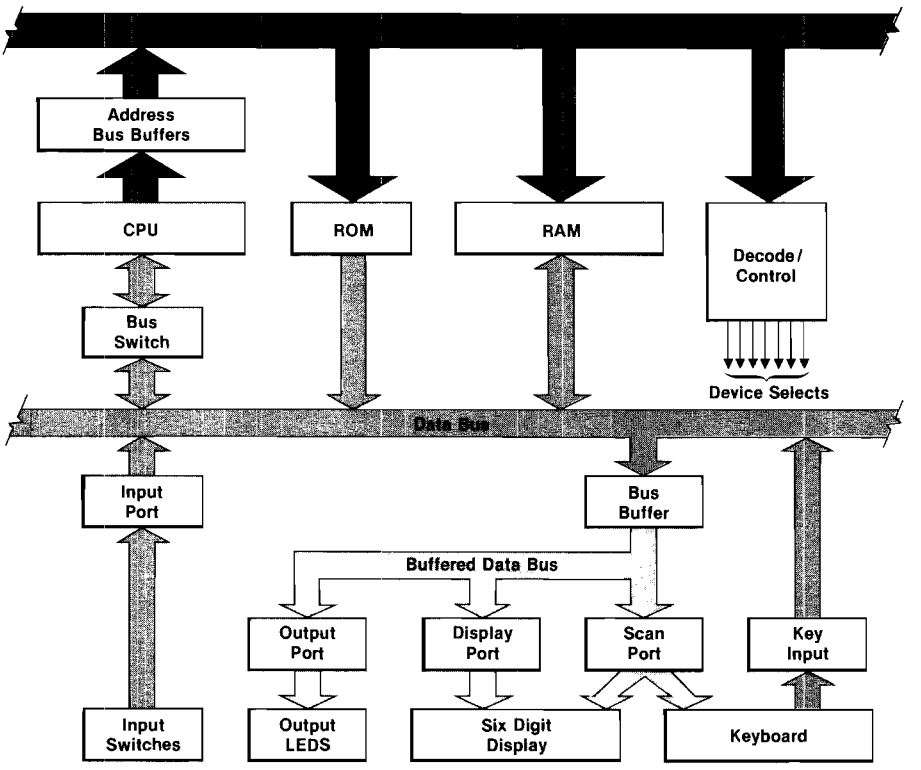


Figure 7-16. Block Diagram of μLab

REVIEW

Lesson 7

Microprocessor systems are designed around a bus structure. The microprocessor generates addresses and control signals. In addition, it connects to a bidirectional data bus on which it can send or receive data. ROMs, RAMs, and I/O ports all use the data bus for data transfers to or from the microprocessor. The address bus specifies the particular memory chip or I/O port that is selected, as well as the location within the memory chip. Address decoding circuits decode the state of the bus and generate an address-select signal for each device in the system. The $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ control signals determine whether a processor read or write operation is taking place and control the timing of the data transfer. In this way the microprocessor controls the operation of the system and allows all devices to share the same data and address buses.

QUIZ

Lesson 7

1. All data travels on the _____ bus, and the _____ and _____ buses select the device that the processor is accessing.

2. The data bus is:
   a. unidirectional and three-state.
   b. bidirectional and three-state.
   c. unidirectional and bidirectional.
   d. none of the above.

3. In a system using several 1K byte memory devices, which address lines are connected to the memory chip?
   a. A0—A9
   b. A10—A15
   c. A0—A15
   d. A0—A7

4. In a system with a 16-bit address bus, what is the maximum number of 1K byte memory devices it could contain?
   a. 16
   b. 64
   c. 256
   d. 65,536

5. When the $\overline{\text{READ}}$ line goes low (true):
   a. the microprocessor has just finished reading data.
   b. a memory or I/O device has just finished reading data.
   c. the microprocessor is reading data.
   d. a memory or I/O device is reading data.

6. The circuit in Figure 7-17 generates a chip select signal for a:
   a. ROM.
   b. RAM.
   c. input port.
   d. output port.

7. The device selected in Figure 7-17 is at address:
   a. A03C.
   b. 430A.
   c. 5FCC.
   d. A034.

8. To convert the circuit of Figure 7-17 to work with an output port, one of the 74LS138's enable inputs would connect to_____instead of_____.

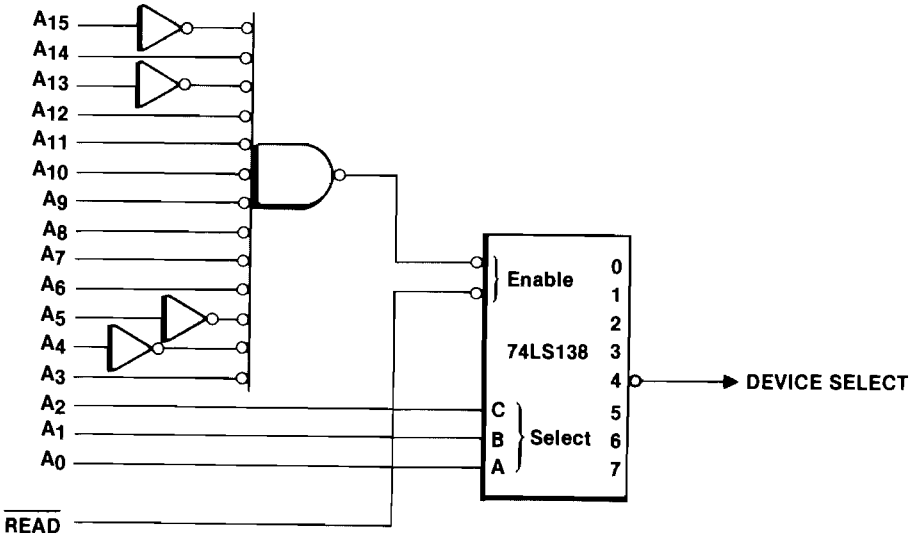


Figure 7-17. Address Decoder Circuit for Questions 6, 7, and 8

LESSON 8

Address Decoding

INTRODUCTION

All devices that communicate with the microprocessor have specific addresses assigned to them. The address decoding circuits ensure that the correct device is on the bus when it is addressed by the microprocessor.

This lesson describes the features and characteristics of the μLab's address decoding circuit. Other types of address decoding circuits are also discussed.

ADDRESSING STRUCTURES

In most microprocessor-based systems, the data and control buses are used in a straightforward manner with few alternatives available. The address bus, on the other hand, can be decoded in many different ways.

THE MICROPROCESSOR LAB'S ADDRESSING STRUCTURE

The μLab uses a technique called *memory mapped I/O*. The I/O ports are treated as addressable devices within the memory space. From a software viewpoint, this means that reading and writing the ports may be done using any of the memory reference instructions: "MOV A,M", "MOV M,A", "STA", "LDA", and a number of others. (These instructions are described in Lesson 11.) An alternative technique, called *I/O Mapped Decoding*, is discussed later in this lesson.

Figure 8-1 shows an *address map* for the μLab. Only the first one-fourth of the 64K byte address space is used. Therefore, address bits 14 and 15 are always zero. This address space is then divided into eight equal sections of 2K locations each (0800 hex × 2,048 decimal). The ROM occupies the first 2K addresses, and the RAM is assigned to the next 2K addresses. Even though the RAM contains only 1K bytes, it is allocated 2K bytes of address space. The extra 1K of available addresses are not used.

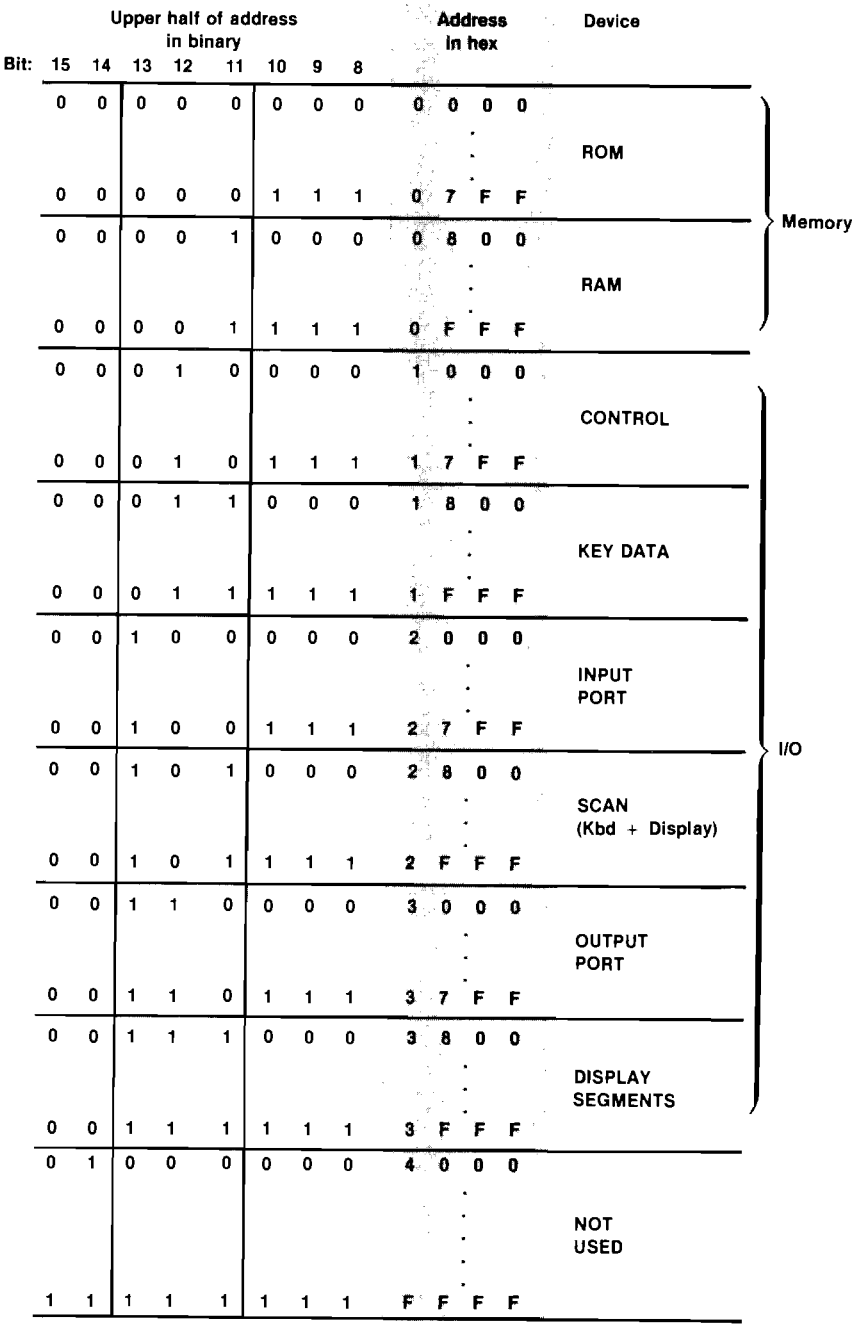


Figure 8-1. System Address Map for μLab

The six 2K byte address blocks following the RAM are used for I/O ports. The control port is used by the monitor program to provide some special functions, described in Lesson 10. The key input, scan, and display segment ports control the keyboard and display. The input and output ports are used for the switches and the LEDs.

Each of these ports sends or receives only a single byte of data and therefore requires only one address. Yet each is assigned 2,048 addresses. Each port responds to any one of its 2K addresses. This means that there are 2,047 redundant addresses allocated to each port.

Why use an addressing technique that appears to waste so many addresses? The reason is that it simplifies the hardware and there is much more address space available than the system needs. The μLab uses 16K addresses for its 2K ROM, 1K RAM, and six I/O ports. This leaves 48K addresses for system expansion. The μLab could have used only 2,048 + 1,024 + 6 = 3,078 addresses, but at the expense of a much more complicated address decoding circuit. This addressing philosophy is commonly practiced in small and medium-sized systems.

THE DECODING HARDWARE

An examination of the hardware used to implement the address decoding will show why this approach has simplified the circuits. Figure 8-2 shows the address decoding and control circuits used in the μLab. Look at the binary addresses listed in Figure 8-1. Notice that the A11, A12, and A13 lines specify which of eight sections is addressed. These three lines are used to provide the binary select inputs to the 74LS138 binary to one-of-eight decoder. This device provides eight separate outputs, one for each of the 2K blocks the system uses. This method results in a relatively simple address decoding circuit. The simplicity is a direct consequence of the fact that each device is assigned a block of addresses of equal length.

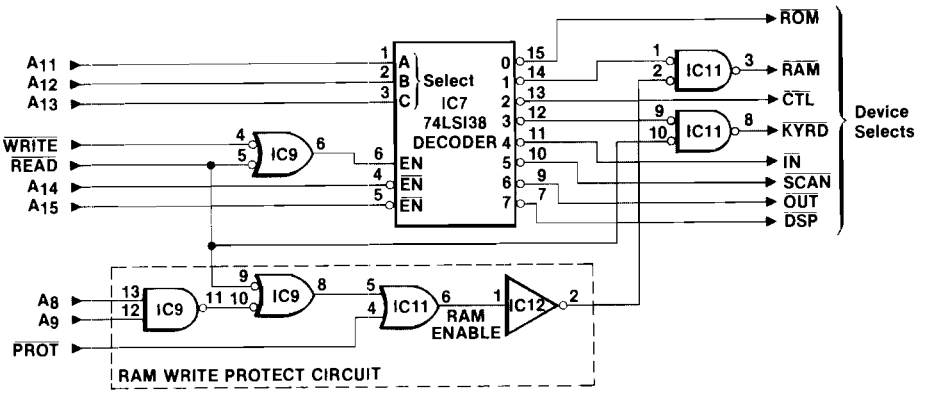


Figure 8-2. Address Decoding Circuit of μLab

The 74LS138 has three enable inputs: two active low and one active high. All three must be true to allow any of the outputs to be true. The A14 and A15 lines (connected to the two active low enables) prevent any of the outputs from being true unless both A14 and A15 are low. This restricts the μLab's devices to the lower 16K of the 64K address field.

Connecting the $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ lines to decoder enable inputs ensures that the bus devices can be enabled only during a read or write operation. This eliminates the need to gate the $\overline{\text{READ}}$ or $\overline{\text{WRITE}}$ lines directly into many of the device select signals. For this reason the decoder's third enable input is connected to a gate that generates the OR of $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$. This allows the device select outputs to be true only when either a read or a write is in progress. The address bus contains meaningful information only during these periods. Enabling the device select outputs only at these times prevents the devices from reading or writing data at the wrong time. Some microprocessor-based systems may not require this type of enabling to be done.

Additionally, the ROM and the input ports must be selected only if a read is being performed. If they responded to either a read or a write, a *bus conflict* could occur. For example, if a write to the ROM is performed, the microprocessor places data on the data bus. A write to a ROM is impossible. So if the ROM is enabled during the write operation, it will also try to put data on the data bus. This is an unacceptable situation which could even result in electrical circuit damage.

To solve this problem the $\overline{\text{READ}}$ signal must be ANDed with the ROM device select. This is implemented by connecting $\overline{\text{READ}}$ to one of the ROM's enables. The ROM and the input port chips in the μLab each have two enable pins. One is used for the device select and the other for $\overline{\text{READ}}$. Using the two enable pins in this way is equivalent to ANDing the $\overline{\text{READ}}$ signal and the device select. (See the complete schematic fold-out at the rear of the book.)

Figure 8-2 shows how this is done for the keyboard input port (KYRD). Since the keyboard port does not have an extra device select pin, IC11C is used to perform the logical AND function.

For the output ports, the situation is slightly different. Consider the $\overline{\text{OUT}}$ line in Figure 8-2. Since the output port controlled by this line can be enabled by either a $\overline{\text{READ}}$ or a $\overline{\text{WRITE}}$ signal, a write enable to the port would occur for either command. If an attempt is made to read the output port (which is not a meaningful operation), a write is performed instead of a read, loading the port with undefined data. This operation is acceptable because it does not cause a hardware conflict (i.e., it does not make any difference from an electrical standpoint.) Therefore, it is not necessary to AND the $\overline{\text{WRITE}}$ signal with the device selects for the output ports.

RAM WRITE PROTECT CIRCUIT

The RAM's device select is somewhat different. It must be true when either a read or a write to the RAM's address space is in progress. The gate (IC11A) on the RAM's device select line is for the write protect circuit, which is described in the following section.

The *write protect* circuit helps prevent the RAM's contents from being accidentally destroyed. Occasionally, a relatively simple programming error causes the microprocessor to run amuck (usually by interpreting data as instructions). This error often results in the storing of garbage data into the entire RAM, which could erase the program just entered.

To prevent this problem, the μLab contains a latch called *memory protect*. The output of this latch provides the $\overline{\text{PROT}}$ input to the circuit in Figure 8-2. When the latch is set, the RAM is protected. In this mode, the RAM can be read but not written to. The monitor automatically sets the protect latch whenever you run a program. Otherwise, it is reset to allow you to enter data or modify programs.

Because you may want to use the RAM to store data during program execution, only the first three-fourths of the RAM is protected. Address lines A8 and A9 determine which fourth of the RAM is addressed. When they are both high, the last quarter of RAM is addressed and therefore not protected. The A8 and A9 lines are ANDed together, and the result is then ORed with $\overline{\text{READ}}$ and ORed with $\overline{\text{PROT}}$. This output produces the RAM enable signal.

The following experiment (and the others in this section) uses an oscilloscope to show some of the signals in the μLab. The oscilloscope must have at least a 10 MHz bandwidth and be dual trace. All of the scope displays are shown as figures, so that you may verify that the set-up is correct. In addition, the figures enable you to understand the experiment even if you do not have an oscilloscope available.

In the discussions that follow, IC pin numbers are separated from the IC number by a hyphen. For example, pin 18 of IC13 is shown as IC13-18. IC numbers are printed on the circuit board.

EXPERIMENT 8-1

The Address Decoder

CONCEPT

In this experiment you will enter and then run a program that will sequentially enable each of the addressable devices in the μLab. You will observe the device select signals with an oscilloscope.

PROCEDURE

A) Key in the following program:

Address	Contents	Label	Instruction	Comments
0800	3A	LOOP:	LDA 0000	;ROM
0801	00			
0802	00			
0803	32		STA 1000	;CONTROL PORT
0804	00			
0805	10			
0806	3A		LDA 1800	;KEY PORT
0807	00			
0808	18			
0809	3A		LDA 2000	;INPUT PORT
080A	00			
080B	20			
080C	32		STA 2800	;SCAN PORT
080D	00			
080E	28			
080F	32		STA 3000	;OUTPUT PORT
0810	00			
0811	30			
0812	32		STA 3800	;DISPLAY PORT
0813	00			
0814	38			
0815	C3		JMP LOOP	;REPEAT
0816	00			
0817	08			

B) Connect the scope's channel A probe to the ROM select slot (labeled $\overline{\text{ROM}}$) below the address bus LEDs on the μLab.

C) Connect channel B to the RAM select line (IC7-14). Be careful not to short IC pins or your program may be destroyed. An IC test clip on IC7 is useful.

D) Set both inputs to 2 V/div. Set the sweep speed to 10 μs/div and trigger on channel A.

E) Run the program entered in step A. The display should be similar to Figure 8-3. The ROM is enabled once during each loop (loop time is about 50 μs). The RAM is enabled every time an instruction byte is read. Notice that the RAM is not enabled (RAM select not low) when the ROM select is low. Also notice that there are six other periods during the loop that the RAM Is not enabled. These correspond to periods when the other six devices are selected.

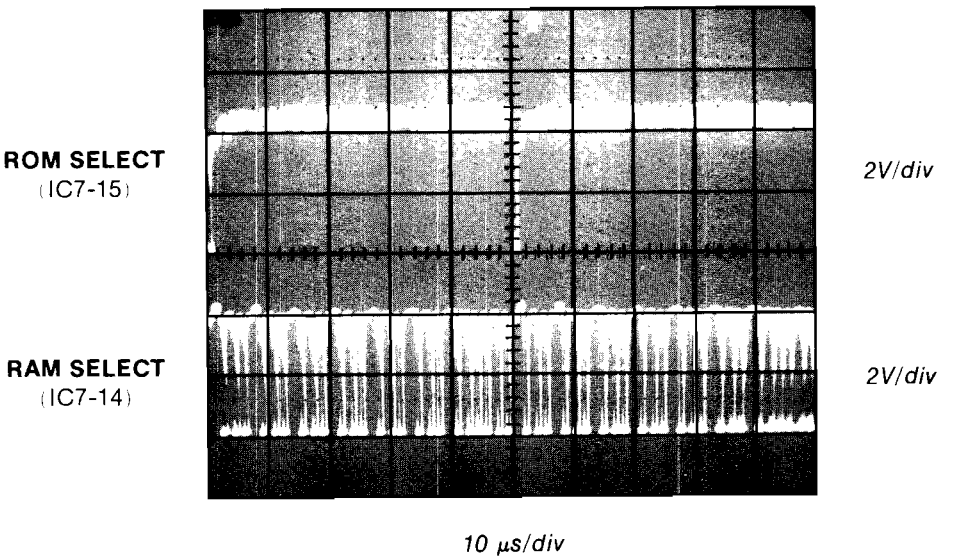
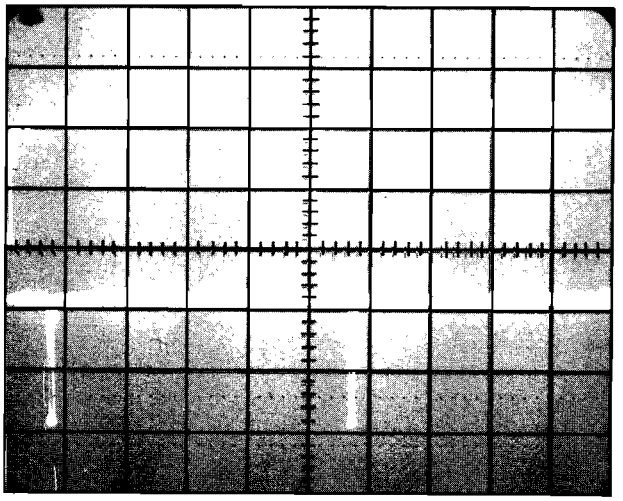


Figure 8-3. RAM Enabled (goes low) Every Time Instruction Byte Is Read for This Program Loop

F) Move channel B to the other enable pins on IC7 and observe from Figures 8-4 to 8-9 that the order in which they are enabled follows the instruction sequence of the program. At no time are any two devices (including the RAM) enabled at the same time.

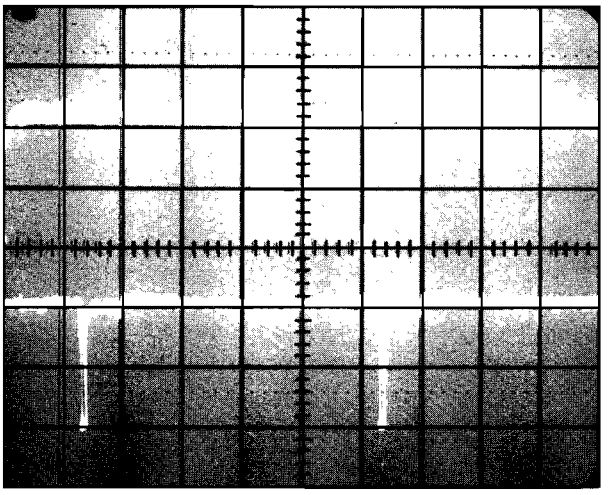
EXPERIMENT 8-1

(Continued)



10 μs/div

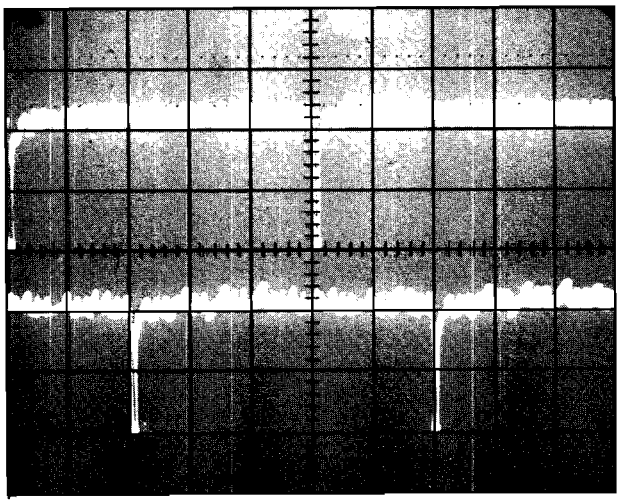
Figure 8-4.



10 μs/div

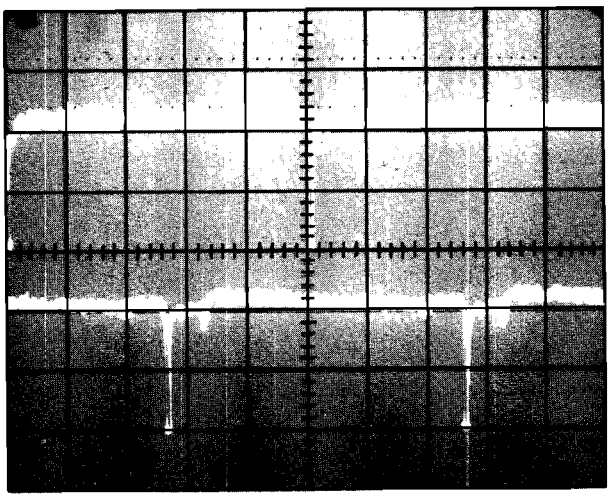
Figure 8-5.

Figures 8-4 to 8-9. Order of Device Enables Follows Instruction Sequence of Program



10 μs/div

Figure 8-6.



10 μs/div

Figure 8-7.

EXPERIMENT 8-1

(Continued)

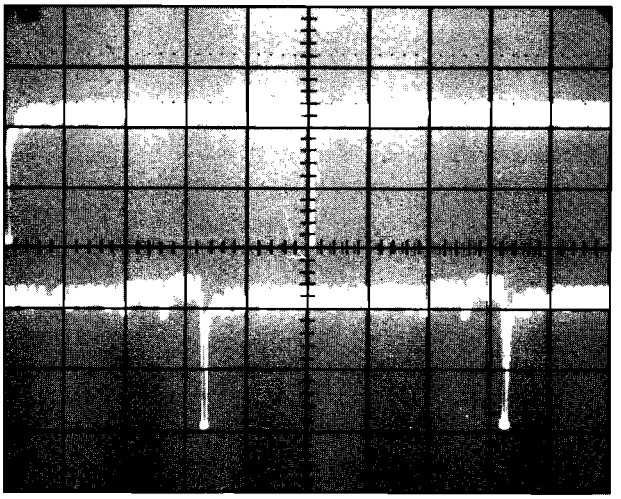


Figure 8-8.

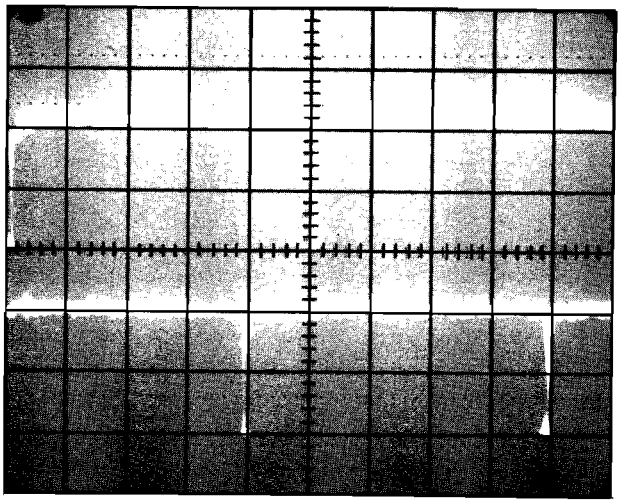


Figure 8-9.

SUMMARY

All eight device selects were examined while running a short exercise program. The control of the device enable lines by the program instruction sequence could be readily observed.

OTHER DECODING TECHNIQUES

There are a number of other techniques for performing address decoding. The technique used for a particular application depends on many factors, including the amount of memory, the number of peripherals, the need for expandability, and the types of memory and I/O devices used. In some cases these devices have enable pins on them that can be used as part of the address decoding.

LINEAR SELECT DECODING

Linear select, the simplest of all decoding techniques, uses no address decoding logic. The high-order address bits act directly as chip selects. Figure 8-10 shows an example of linear select decoding. The RAM is selected whenever A15 is high. This corresponds to all addresses from 8000 to FFFF. The ROM is selected whenever A14 is high. This is true for addresses 4000 to 7FFF.

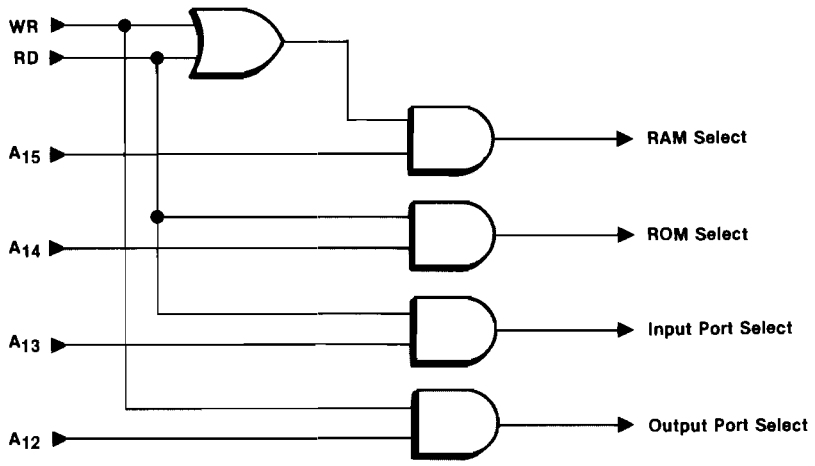


Figure 8-10. Linear Select Decoder

Notice that the ROM is also selected if A14 and A15 are high, corresponding to addresses C000 through FFFF. This overlaps the RAM's space. Both devices are enabled if an attempt is made to read from any of these addresses and will cause a bus conflict. This is one disadvantage of linear select decoding. Because of this potential problem, the software must never read any address in which more than one of the two most significant bits are true. Another disadvantage of this method is that it wastes a large amount of address space. The technique is therefore limited to small systems.

LOGIC COMPARATOR DECODING

One of the most straightforward and flexible techniques, *logic comparator decoding*, selects a single portion of 2^N possible address fields from N address inputs. Figure 8-11 shows a circuit that generates a single device select from the six high-order address bits of a system. Each comparator A input is compared to its respective B input. When they all match (all six input pairs are coincident), the comparator output goes low. The switches are used to set the logic level at the B inputs to the comparator. This technique is particularly useful on memory and peripheral boards where there are switches or jumpers that set the address of each board in a system. Exclusive-or gates can also be used to accomplish the comparator function.

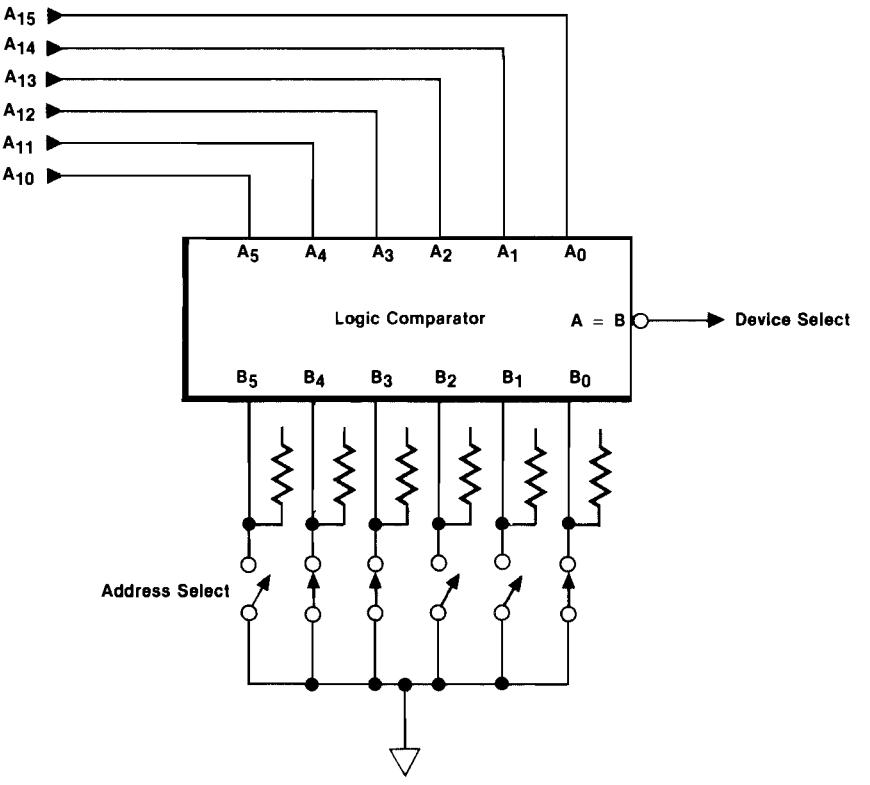


Figure 8-11. Logic Comparator Decoder

COMBINATIONAL LOGIC DECODING

In systems with very limited decoding requirements, standard logic gates are often used. Figure 8-12 shows a four-input NAND gate preceded by inverters which decodes addresses 9000 through 9FFF. The output goes low whenever the A15 to A12 address lines are in the state 1,0,0,1. By complementing or not complementing the address inputs to the gate, any one of sixteen (2^4) devices can be enabled.

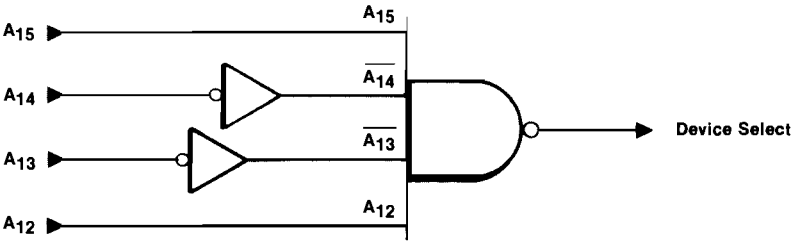


Figure 8-12. Logic Gate Decoder

I/O MAPPED DECODING

A few microprocessors (including the 8080 and 8085) use an extra control line to specify that the address is for either I/O or memory. In the 8085, this line is called IO/$\overline{M}$ (see Figure 8-13). The instruction being executed controls this line. During all memory transfers, IO/$\overline{M}$ is low. When one of the two I/O instructions (IN or OUT) is executed, IO/$\overline{M}$ goes high, enabling the I/O ports. When IO/$\overline{M}$ is low the memory is enabled. By using this method, memory and I/O have separate address spaces, increasing the total addressable space in a system (by $2^8 = 256$ bytes) and permitting a greater degree of decoder design flexibility. Also, IN and OUT are two-byte instructions (specifying one of 256 I/O ports). This saves one byte of program memory for each I/O transfer, as compared to using three-byte memory instructions such as "LDA" and "STA."

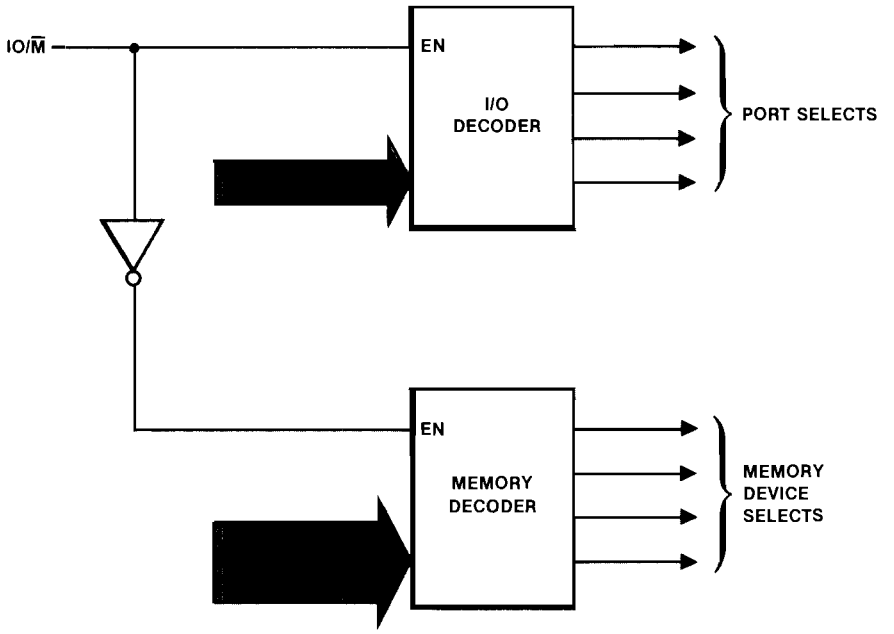


Figure 8-13. I/O Mapped Decoding Increases Total Address Space by 256 Bytes

Because the μLab does not use I/O mapping for its I/O ports, the IO/$\overline{M}$ line is not used. Instead, the μLab uses memory mapped I/O, as discussed previously in this lesson.

REVIEW

Lesson 8

Microprocessor-based systems employ many techniques for address decoding. Decoder ICs, comparators, and gates are often used for these circuits. In small systems, the address decoder tends to be simpler but more wasteful of the available address space. Memory mapped I/O is a commonly used decoding technique which addresses I/O ports as memory.

QUIZ

Lesson 8

1. Devices in a microprocessor system:
   a. have specific addresses assigned to them.
   b. always respond to only a single address.
   c. are always controlled by an address decoder circuit.
   d. are all of the above.

2. Memory mapped I/O:
   a. treats I/O ports as memory locations.
   b. tends to be more wasteful of address space than I/O mapped decoding.
   c. is used in the μLab.
   d. is all of the above.

3. A bus conflict will occur when:
   a. an output port is enabled by a $\overline{\text{READ}}$ signal.
   b. more than one device is reading from the data bus.
   c. more than one device is writing to the data bus.
   d. MOS and TTL devices are both used on the data bus.

4. The decoding technique that tends to be the most wasteful of available address space is:
   a. decoder ICs (as used in the μLab).
   b. linear select.
   c. logic comparator.
   d. combinatorial logic.

5. If devices are enabled when neither a $\overline{\text{READ}}$ nor a $\overline{\text{WRITE}}$ is in progress:
   a. bus conflicts may occur.
   b. data in the ROM may be lost.
   c. no problems occur because this is a normal state.
   d. none of these is true.

6. The greatest number of memory and I/O locations can be addressed by the following decoding technique:
   a. linear select.
   b. memory mapped I/O.
   c. I/O mapped decoding.
   d. logic comparator decoding.

LESSON 9

Memories and Peripherals

INTRODUCTION

If you think of the microprocessor as the heart of the system, then the data bus is the bloodstream, and the memories and peripherals are the organs. This lesson discusses the memories and peripherals used in the μLab and other systems as well.

MEMORIES

Semiconductor memories are available in two fundamentally different types: RAMs and ROMs. Within each of these categories, there are many varieties. Some of the more commonly used memory devices are discussed in this section.

RAMS

RAMs are used for the user "programmable" memory and the data storage in nearly all microcomputer systems. There are two different types of RAMs: static and dynamic. Static RAMs use a flip-flop for each memory element. A 1K RAM IC therefore has 1024 flip-flops in it. Each flip-flop can be set to store a one or reset to store a zero. Address decoding circuits inside the RAM chip select the particular flip-flop specified by the address lines. The state of the flip-flop does not change unless new data is stored in it or power to the RAM is interrupted.

Dynamic RAMs use an on-chip capacitor for each storage element. In general, a charge is stored on the capacitor to indicate a one; no charge indicates a zero. This technique simplifies the storage cell, permitting denser memory chips. There is a problem, however: the charge leaks off the capacitor and, after a few milliseconds, a one can become a zero. They must therefore be *refreshed*. Refreshing consists of reading a sequence of RAM address locations within a specified time. In the process of reading the data, the RAM chip automatically re-writes the same data back into the location read. As a result, all the one bits are restored to full charge and the zero bits to no charge. Dynamic memories are typically refreshed at least every two milliseconds.

Because dynamic memories must be continually refreshed, special circuits are added to do this, resulting in a more complex system. Small systems tend to use the simpler static memories because of this factor. However, dynamic memories have a number of advantages. They are less expensive than static RAMs of the same size and usually consume less power. The largest RAMs (greatest number of bits) are often available only in the dynamic type. Systems with large amounts of memory often use the cheaper, lower-power dynamic memories. Dynamic memories are almost always one bit wide. Some microprocessors contain on-chip refresh circuits to simplify the use of dynamic RAMs.

The Microprocessor Lab's RAM

The μLab uses 4K-bit static RAMs (2114 or 4045). They are organized 1K × 4. Two of them are used to get an eight-bit word. Two 1K × 4 chips therefore provide 1K bytes (1K × 8) of RAM.

Figure 9-1 shows the μLab's RAM circuit. The address and control pins of both chips are bused together. IC5 connects to data lines 0-3, and IC6 connects to lines 4-7. The two chips thus act as 1K bytes of memory.

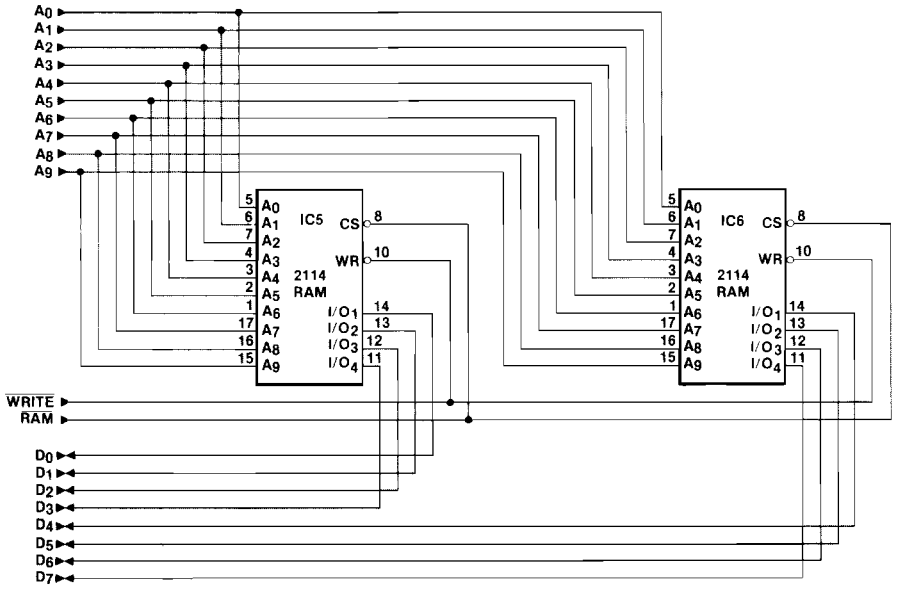


Figure 9-1. RAM Circuit for μLab

Other Memory Configurations

Many common memory chips are only one bit wide. The 2102 was the first inexpensive static memory and is organized 1K × 1. Since each chip reads or writes only one bit at a time, eight chips are required to read and write 1K bytes of data.

Figure 9-2 shows a 1K × 8 memory using 2102s. As with the 2114 circuit, all the address and control lines are bused together. Each chip takes care of one bit of the data bus. The 2102 has separate data in and data out pins which are useful in special systems that do not use a bidirectional data bus. For most microprocessor applications, these pins are simply tied together. This circuit is functionally equivalent to Figure 9-1, but requires eight ICs instead of two. This is representative of the ever-increasing density of memory chips.

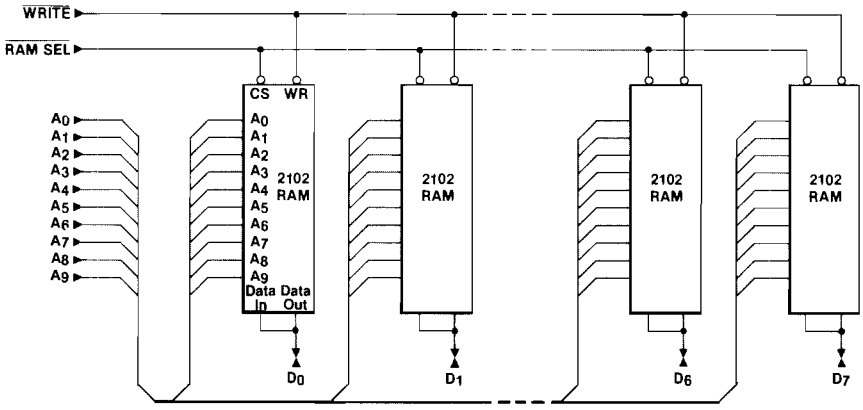


Figure 9-2. 1K x 8 Memory Using 2102 (1Kx1) RAMs

Other common memory chips are the 2141 (4K × 1 static), the 2104 (4K × 1 dynamic), and the 2116 (16K × 1 dynamic). Eight 2116s will provide 16K bytes of memory, which would require 128 2102s!

READ-ONLY MEMORIES

Read-only memories (ROMs) provide a means of permanently storing programs and data. The μLab's monitor program is stored in a ROM so that it will always be available. Since RAMs lose their contents when power is removed, they are not very useful for storing permanent programs. Most ROMs intended for use with microprocessors are eight bits wide.

There are four different types of ROMs. *Mask-programmed ROMs* are programmed by the IC manufacturer by customizing the actual chip. There is usually a one-time charge to generate the mask for a particular program, but thereafter, the ROMs are relatively inexpensive. Mask-programmed ROMs are often used in reasonably high-volume products because they are the least expensive and the highest in bit density.

The second type is the *Programmable Read Only Memory* (PROM). The user programs the ROMs electrically utilizing a special device called a PROM programmer. Once they are programmed, however, they cannot be changed.

The *Erasable Programmable Read-Only Memory* (EPROM) is similar to a PROM, except that it can be erased and reprogrammed. Programmed bits are stored as charge on a near-zero-leakage capacitor. Erasing is performed by shining ultraviolet light through a clear window in the IC package. These devices are most useful for prototypes or small-volume production runs. During the development of the μLab, an EPROM (2716) was used.

The newest ROM type is the *Electrically Alterable Read-Only Memory* (EAROM). The EAROM can be erased electrically while in the circuit. One advantage of this type over the EPROM is that small sections of the EAROM can be erased, whereas EPROMs must be completely erased. But EAROMs are not yet as easy to use as EPROMs and are more expensive. Many systems that require data to be stored for a long period of time, but which must change this data occasionally, use EAROMs. Typical applications include digital TV tuners, calibrated transducers, and automatic telephone dialers.

The Microprocessor Lab's ROM

The μLab uses the 2316E ROM, shown in Figure 9-3. It contains 2K bytes and is mask-programmed. The ROM drives the data bus only if the ROM select is true and the operation is a read. The two chip selects must both be true to enable the three-state output drivers. The low-order eleven address lines are used to specify the address.

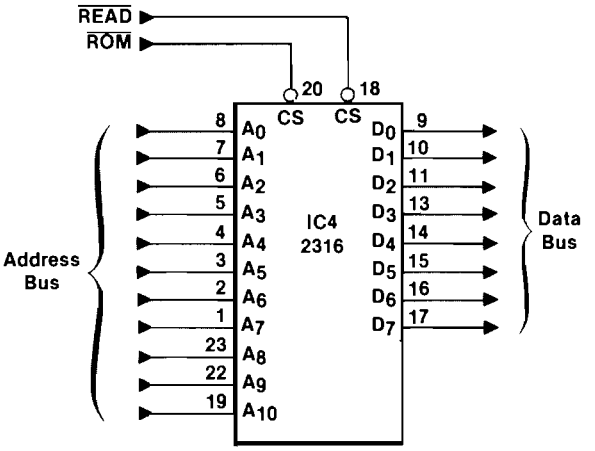


Figure 9-3. ROM Circuit for μLab

MICROCOMPUTER PERIPHERALS

For a microprocessor system to be of any use, there must be some way for it to interact with the outside world. It must have at least one input and one output device, which connect to the microprocessor system through I/O ports. Since they are not directly involved in the operation of the microprocessor, they are called *peripherals*.

INPUTS AND OUTPUTS

The μLab has two simple peripherals: the input port slide switches and the output port LEDs. Figures 9-4a and 9-4b show how they are connected. The ports and control circuits are exactly as described earlier. The switches at the inputs of the input port cause the corresponding input to be low if a switch is closed. If a switch is open, the resistor pulls the input high.

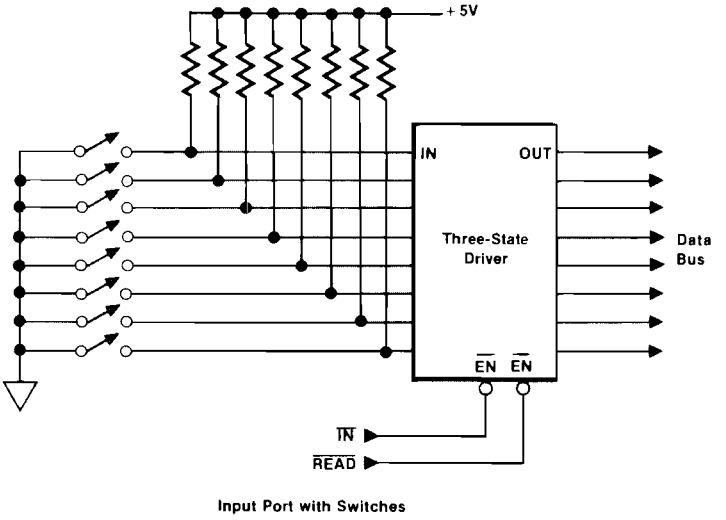


Figure 9-4a. μLab Input Port

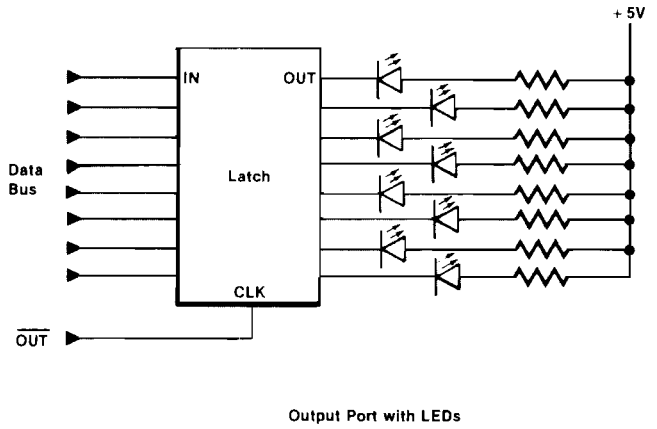


Figure 9-4b. μLab Output Port

Each output from the output port simply drives an LED. Resistors limit the current. When the output is low, current is drawn through the LED and it lights up.

KEYBOARD AND DISPLAY

Although the switches and LEDs are simple and functional, many applications require something easier for the operator to use. Two very common microprocessor system peripherals are keyboards and displays, both of which are used in the μLab. The following section describes the basic operation of these peripherals.

The Keyboard

Scanning, an important hardware technique, is used in both of these peripherals. The μLab keyboard has twenty-six keys. To connect each key to an input port bit would require four input ports (eight bits each). With the use of a scanning technique, however, up to 256 keys can be interfaced using only two eight-bit ports.

Figure 9-5 shows a keyboard interface. The keys are arranged in a matrix, interconnecting column lines and row lines. An output port drives the columns, and an

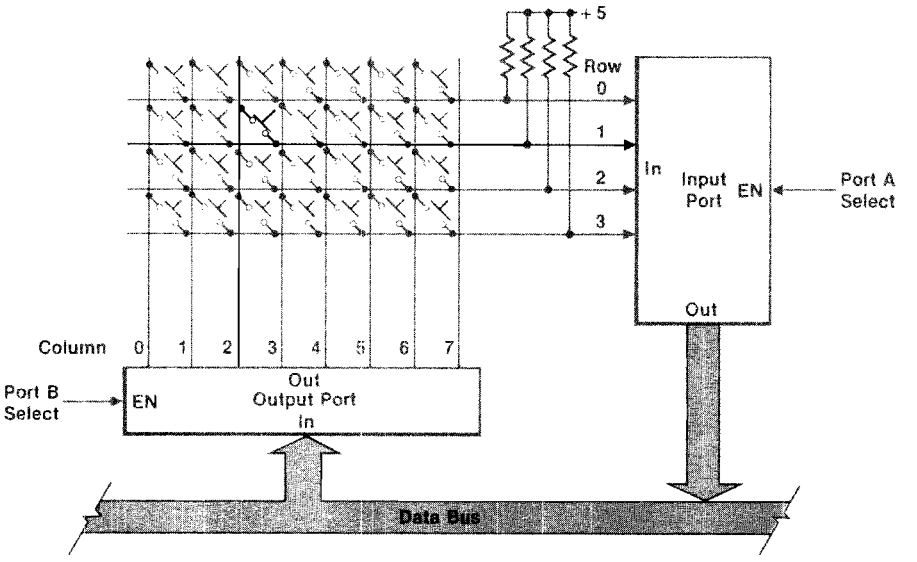


Figure 9-5. Keyboard Interface Similar to That Used in μLab

input port reads the rows. Figure 9-6 shows an example of how a key is detected. A program is required to scan the keyboard. The output port is set so that one line is low and all the others are high, and then the input port is read. If any of the keys in the column whose line is low are pressed, then the row line of that key is forced low causing that input port bit to go low. The program then knows which column the key is in (from which output port bit it set low) and which row it is in (from which input port bit is low). This uniquely identifies the key pressed.

State	Output Port Column								Input Port Row				
	0	1	2	3	4	5	6	7	0	1	2	3	
0	0	1	1	1	1	1	1	1	1	1	1	1	
1	1	0	1	1	1	1	1	1	1	1	1	1	
2	1	1	0	1	1	1	1	1	1	0	1	1	← Key Pressed is in Row 1, Column 2
3	1	1	1	0	1	1	1	1	1	1	1	1	
4	1	1	1	1	0	1	1	1	1	1	1	1	
5	1	1	1	1	1	0	1	1	1	1	1	1	
6	1	1	1	1	1	1	0	1	1	1	1	1	
7	1	1	1	1	1	1	1	0	1	1	1	1	

Figure 9-6. Scan Data for Pressed Key of Figure 9-5

In order to check all the keys, each output port bit is in turn set low and the keys are scanned. At each step only four keys are checked. The process is so fast, however, that the entire keyboard can be checked, four keys at a time, in much less time than the fastest possible rate of operator key depression.

The Display

The display also uses a scanning technique. Although the μLab display has six digits, only one is actually on at any instant. Each is turned on in sequence, but the process happens so rapidly that they all appear to be on at the same time.

Each display digit consists of seven LED segments plus a decimal point. There is a connection to each of these eight LEDs plus a common connection. A character is displayed by putting a low level on the common connection and a high level on the segment connections that correspond to the segments you wish to light. Current is limited by resistors in series with the segments.

To interface the displays without using an inordinate number of output port lines and current-limiting resistors, the segment connections for each display are bused together (see Figure 9-7). One eight-bit output port then supplies segment information to all of the displays. Another output port drives the commons of each display and specifies which digit to light.

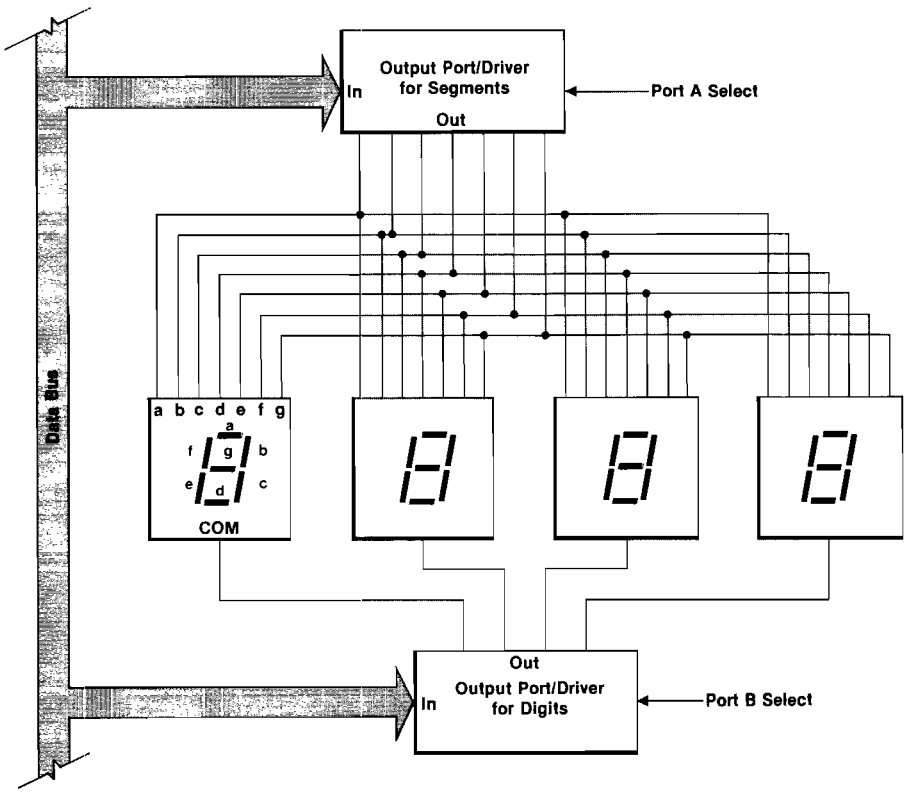


Figure 9-7. Display Interface Similar to That Used in μLab

Some software is, of course, required to run the display. First the segment information for digit zero is sent to the segment port where it is latched. Then the digit port's latch is set to activate only digit zero. After digit zero is on for a given amount of time (controlled by a timing loop in the program), it is turned off (via the digit port). Then the segment information for digit one is sent to the segment port, and digit one is enabled by the digit port. This process is repeated indefinitely, with each digit lighted in turn.

This provides an excellent example of how software is used to replace hardware. Traditionally, each digit had its own drive circuit requiring many more control and drive components. Another task performed by the software is to determine which of the segments to light to generate a given character. Previously, a seven-segment decoder was used for each digit. An advantage of the software technique (besides the elimination of the decoder) is that new characters can be made up at will. Anything you want to display that uses seven segments can be generated by the software in the system's memory. This capability makes possible the display patterns generated by some of the demonstration programs in the μLab's ROM.

These circuits are very similar to the ones actually used in the μLab. The main difference is that in the μLab's circuits the scan port is shared between the key column port and the display digit port (a technique often used to reduce hardware).

THE SERIAL OUTPUT PORT

The serial output port drives the μLab's speaker and is, in effect, a one-bit output port controlled by a special instruction in the microprocessor (SIM). Figure 9-8 shows how the speaker is connected. The 8085's SOD (serial output data) output is buffered and sent to the edge connector for use by external hardware. It is then buffered again to drive the speaker. The speaker draws so much current that the signal at the edge connector would not have valid logic levels if a separate speaker buffer were not used. A 100 ohm resistor in series with the speaker limits the current to a level that does not damage the buffer and permits suitable speaker volume. Notice that the other end of the speaker is connected to +5 V, not to ground, because the TTL buffer can sink more current than it can source.

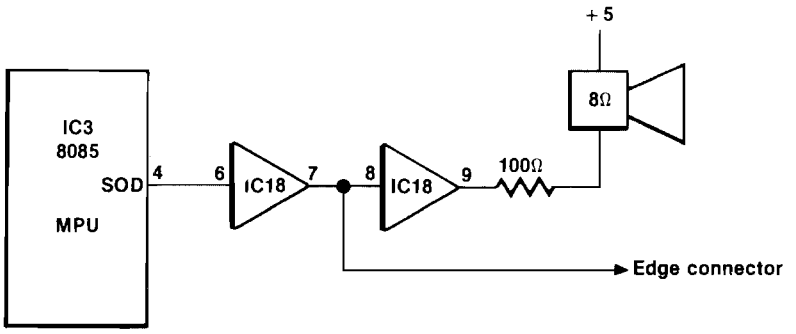


Figure 9-8. Serial Output Circuit of μLab

The software controls the tone generation. The "BEEP" program in the ROM turns the serial output on and off several hundred times a second, driving the speaker with a square wave.

EXPERIMENT 9-1

Program Execution

CONCEPT

In performing this experiment, you examine the outputs of the address decoder with an oscilloscope while the μLab is running a short program. This enables you to see the sequence of operations. You also examine these signals while a long program is running.

PROCEDURE

A) Key in the following program:

Address	Contents	Label	Instruction		Comments
0800	32	LOOP:	STA	3000	;Write Accumulator to Output Port
0801	00				
0802	30				
0803	C3		JMP	LOOP	;Repeat
0804	00				
0805	08				

B) Connect the oscilloscope's channel A probe to the output port select on the address decoder (IC7-9). Be careful not to short pins or your program may be destroyed.

C) Connect channel B to the RAM select (IC7-14).

D) Set both inputs to 2 V/div. Set the sweep speed to 2 μs/div and trigger off channel A.

E) Run the program entered in step A. Verify that the display is similar to Figure 9-9. Remember that both signals are active low. The RAM select goes low for each memory reference, which (for this program) is each time another byte of the program is read. Notice that the RAM is selected six times (corresponding to the six bytes in the program). Then the port is selected once (when the program writes the data to the port).

(Continued)

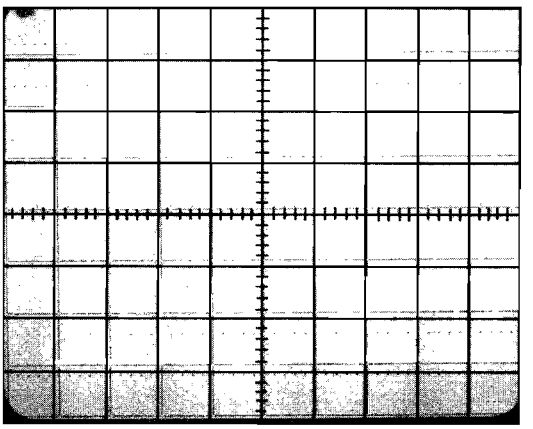


Figure 9-9. Chip Select Signals with Short Loop Program Running

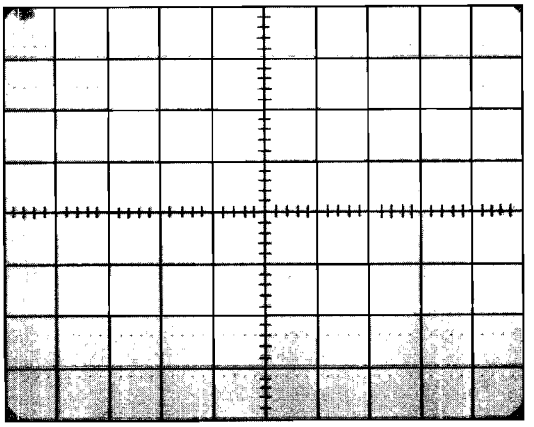
F) How long does it take for the program loop to be executed once? (Answer: 11.5 μs.)

G) Notice that the two devices (the RAM and the output port) are never selected at the same time. Examine the other device select pins on IC7. Since no other devices are used by the program, the other select signals are all high.

H) Press RESET to return the μLab to the monitor program.

I) Using only one channel of the oscilloscope, examine the ROM and RAM device select lines (IC7 pins 15 and 14). Set the input channel to 1 V/div and the sweep speed to 1 μs/div. Figures 9-10 and 9-11 show typical displays. It is difficult (and sometimes impossible) to sync on these waveforms because the program that is running (the monitor) is in a very long loop. Therefore, while the waveforms are periodic, the periods are very long. The trigger holdoff or delay control on the scope can be helpful.

ROM SELECT
(IC7-15)

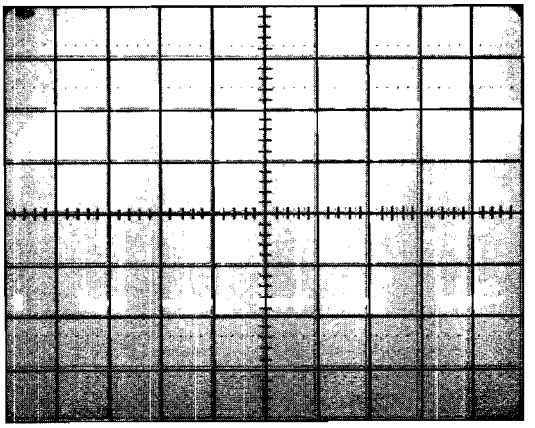


1V/div

1 μs/div

Figure 9-10. ROM Select Signal with Monitor Program Running

RAM SELECT
(IC7-14)



1V/div

1 μs/div

Figure 9-11. RAM Select Signal with Monitor Program Running

(Continued)

J) Examine the $\overline{\text{KYRD}}$ (keyread) select signal (IC7-12). Turn the sweep speed down to 50 μs/div. By carefully adjusting the trigger level, you should get a display like the one shown in Figure 9-12. The sequence of eight short pulses is the scanning of the eight rows of the keyboard. The frequency of this signal is lower than the memory selects because many instructions from memory are executed between each key read.

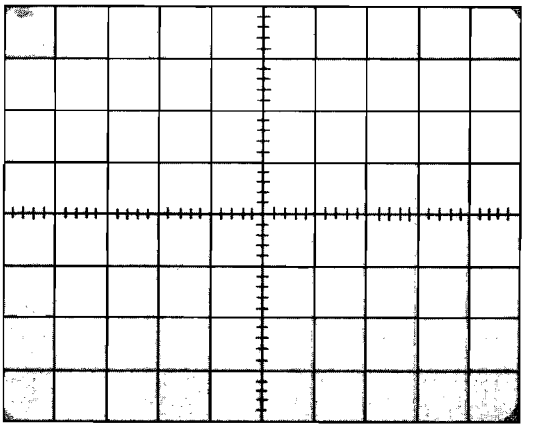
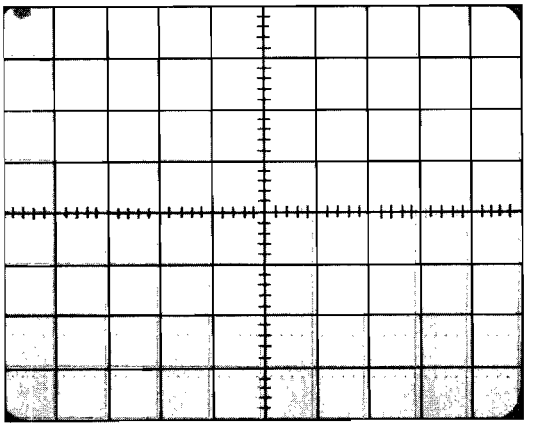


Figure 9-12. KYRD Select Signal with Monitor Program Running (Fast Sweep Speed)

K) Turn the sweep speed down to 2 ms/div. Verify that the display is similar to Figure 9-13. The groups of eight pulses are now blurred and appear as single pulses. Notice that they occur relatively infrequently. This happens because the monitor program reads the keyboard once and then refreshes the display (which takes much longer).

L) Examine the $\overline{\text{SCAN}}$ select signal (IC7-10). Try viewing it at several different sweep speeds. Figure 9-14 shows the display obtained at 200 μs/div. This signal is low each time the keyboard is read or data is sent to the display.

KYRD SELECT
(IC7-12)

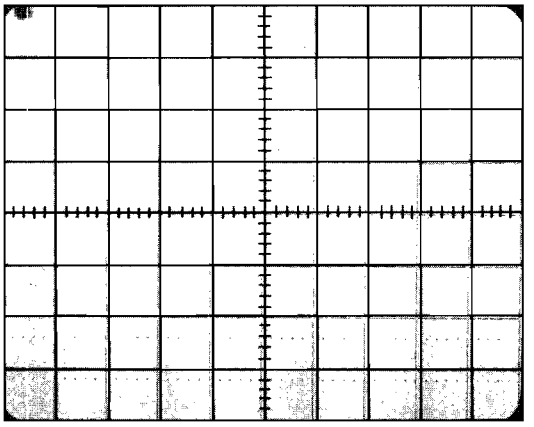


1V/div

2 ms/div

Figure 9-13. KYRD Select Signal with Monitor Program Running (Slow Sweep Speed)

SCAN SELECT
(IC7-10)



1V/div

200 μs/div

Figure 9-14. SCAN Select Signal with Monitor Program Running

(Continued)

M) Examine the $\overline{\text{DSP}}$ (display) select signal (IC7-7). This signal is low each time data is sent to the display. Figure 9-15 shows the display obtained at 200 μs/div.

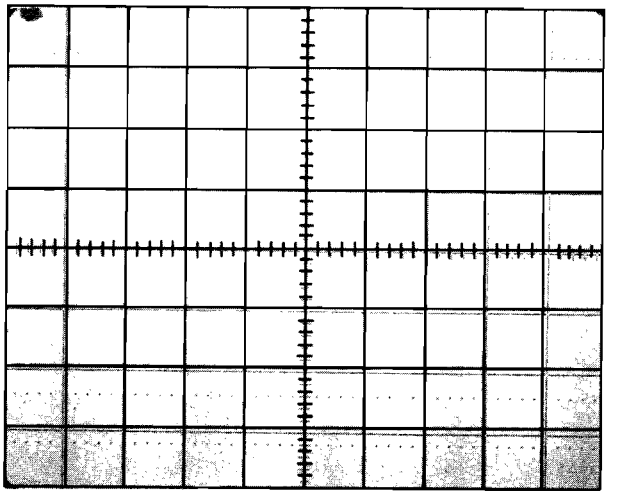


Figure 9-15. DSP Select Signal with Monitor Program Running

SUMMARY

Device select waveforms were examined, first while a short loop was running, and then when the relatively long monitor program was running. The short loop allowed easy examination of the waveforms and analysis of the timing. When the monitor program is running, however, the signals can be very difficult to view. Operations such as reading the keyboard take only about a microsecond but are performed only once every ten milliseconds. Short pulses, widely spaced, are common waveforms for device select signals.

PERIPHERAL INTERFACE CHIPS

All of the I/O functions on the μLab are performed using standard TTL parts. In other systems, a large variety of large-scale integration (LSI) I/O chips are commonly used to simplify many interfacing tasks, reduce component count, and increase system cost-effectiveness.

One of the simplest of these chips is the *Parallel Peripheral Interface*, or PPI (also called PIO, for Parallel Input Output). One version of such a part is a forty-pin IC that contains three I/O ports (see Figure 9-16). Each of the ports is used either as an input or output port. The direction of each port is controlled by a control register on the chip. An initialization program, contained in the system's ROM, sets this control register to select the desired combination of input and output ports.

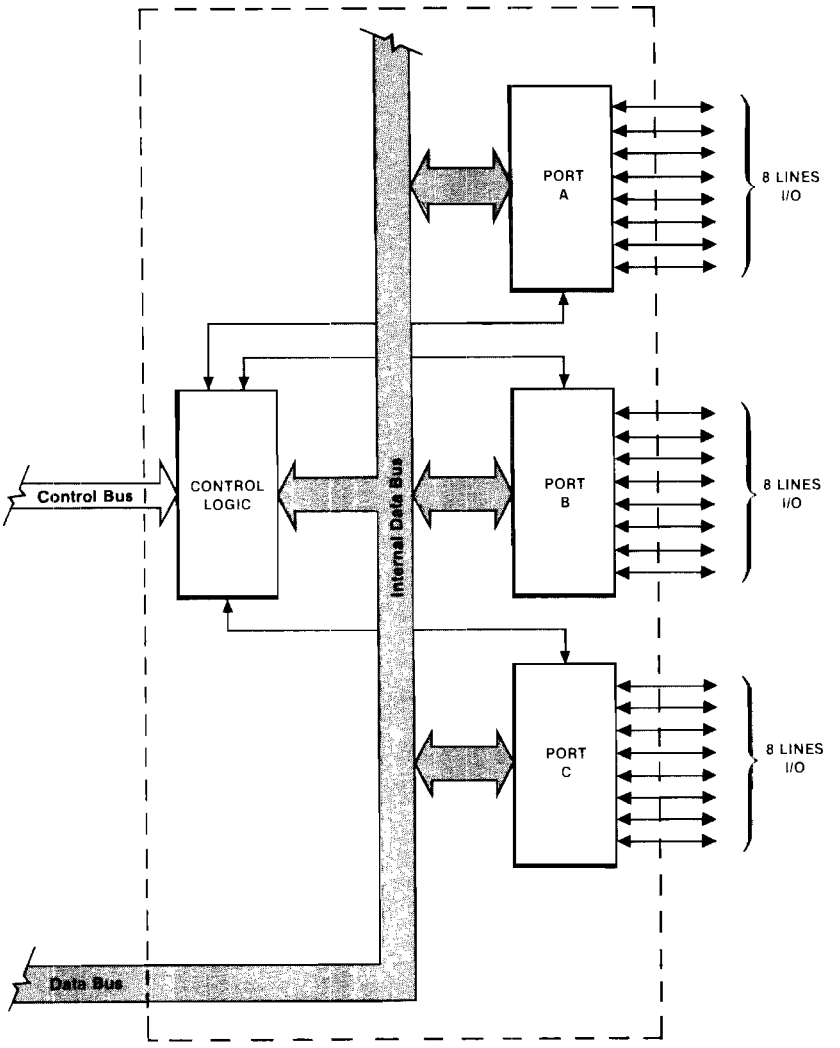


Figure 9-16. Peripheral Interface Device Containing Three 8-Bit I/O Ports

These chips have the advantage of providing several ports in one IC. They also are very flexible, since the nature of each port may be changed by the software. In addition, they usually include some control logic for synchronizing communication and interrupt control.

Another common type of interface chip provides serial inputs and outputs (see Figure 9-17). These chips are commonly referred to as UARTs (Universal Asynchronous Receiver and Transmitter). They accept a byte of data from the microprocessor and then output it one bit at a time. They operate much like a parallel-in/serial-out shift register. In addition, start, stop, and other synchronizing and control bits can be automatically inserted. The format is controlled by a control register similar to the one described for the PPI. UARTs can also handle data in the other direction, converting a serial bit stream into a parallel form suitable for direct use by the microprocessor. In this mode, they operate much like a serial-in/parallel-out shift register. These devices are also called SIOs (Serial Input Output) or ACIAs (Asynchronous Communications Interface Adapter).

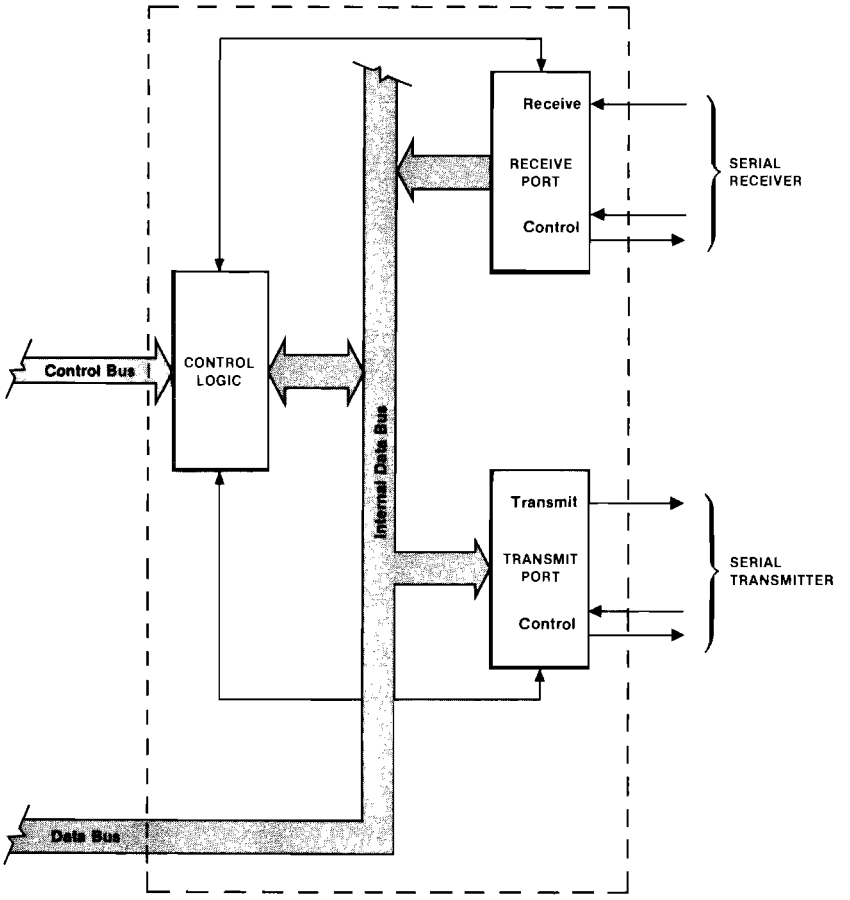


Figure 9-17. UARTS Provide Serial Communication Interface Between Two Systems

Serial I/O is most commonly used for communication between a microprocessor-based system and a peripheral, such as a CRT or teletype terminal. Since the information is in a serial format, only two wires are needed to interconnect the devices.

A wide variety of specialized interface chips is also available, including floppy disc controllers, CRT display controllers, direct memory access controllers, and keyboard and display controllers. Many of these LSI circuits are even more complex than a microprocessor. Some of them use a special-purpose internal microprocessor to control their function.

Lesson 9

There are two main classes of semiconductor memory: RAMs and ROMs. RAMs can both store and output data. Dynamic RAMs will lose stored data unless they are continually refreshed, but static RAMs do not need to be refreshed. ROMs can output only data that has been programmed into them. They are classified by the method in which they are programmed.

Microprocessor peripherals communicate through I/O ports to the data bus. Scanning is commonly used to reduce the amount of hardware required for keyboard and display circuits. LSI peripheral interface chips are often used to simplify system I/O.

QUIZ

Lesson 9

1. The main advantage of dynamic RAMs over static RAMs is that they are:
   a. non-volatile.
   b. easier to use.
   c. less expensive.
   d. all of the above.

2. Dynamic RAMs are best suited to:
   a. slow systems.
   b. small systems.
   c. large systems.
   d. one-bit systems.

3. The program ROM found in a high-volume microprocessor-based product is most likely to be a:
   a. mask ROM.
   b. PROM.
   c. EPROM.
   d. EAROM.

4. All microprocessor systems need peripherals to:
   a. interact with the outside world.
   b. supply data to the microprocessor.
   c. receive data from the microprocessor.
   d. do all of the above.

5. The main advantage of scanning is:
   a. faster I/O operations.
   b. a reduction in software cost.
   c. a reduction in hardware cost.
   d. all of the above.

6. When the keys in Figure 9-5 are scanned:
   a. one key is read at a time.
   b. one key row is read at a time.
   c. one key column is read at a time.
   d. all keys are read at once.

(Continued)

7. The speaker can be made to "beep" a tone by executing the:
   a. SIM instruction.
   b. OUT SOD instruction.
   c. BEEP instruction.
   d. BEEP subroutine.

8. LSI peripheral interface chips are used because they:
   a. simplify many interfaces.
   b. are flexible.
   c. are cost-effective.
   d. are all of the above.

LESSON 10

Control Circuits

INTRODUCTION

This lesson deals with the control signals used in microprocessor-based systems and the circuits that generate, transmit, and respond to them. Electrical circuit considerations are also presented. The discussions center on these topics as they relate to the μLab.

THE CONTROL PORT

There is an additional output port in the μLab, called the control port. You may not be aware of it because it is not obvious to the user. The microprocessor uses it to send signals to special circuits. The $\overline{\text{PROT}}$ bit of this port controls the memory protect circuit described earlier. If this bit is set, the first three-fourths of the RAM is "write protected." The other two bits control the HDWR and INSTR single-step circuits, which are described later in this lesson.

Figure 10-1 shows the control port register. It is clocked by the CTL (control port) select signal generated by the address decoder. This is similar to the other output ports. The unusual thing about the control port is that the data inputs are connected to the address bus instead of the data bus. Therefore, the data written to the port is independent of the state of the data on the data bus.

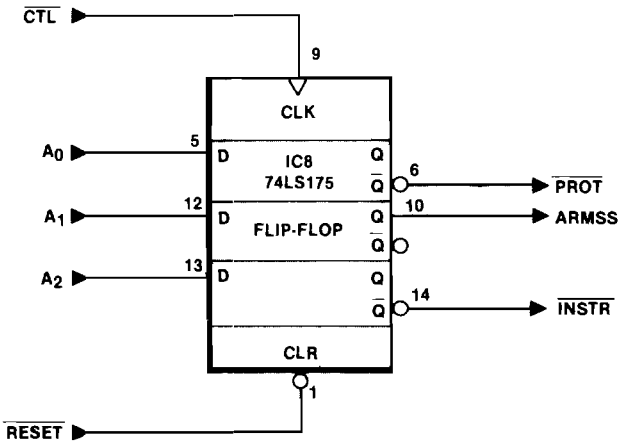


Figure 10-1. Control Port Register of μLab

How, then, does this port work? If you refer back to Figure 8-1, you can see that the control port is selected by any address from 1000 to 17FF. This allows the eleven low-order address lines to contain any value and still select this port. Notice that A0, A1, and A2 provide the data inputs to the register. The address used determines the data written to that port. For example, a write to address 1000 clears all the bits. A write to address 1001 sets the "PROT" bit, and a write to 1004 sets the "INSTR" bit.

This technique simplifies the control software. Since it doesn't matter what data is sent to the port (only the address matters), the software does not have to set up a data value before it writes to the port. The hardware is no more complicated than if the traditional arrangement were used.

Note that this control port is used for special features such as single-stepping and memory protection. It is not one of the basic control circuits required by the microprocessor system.

THE MULTIPLEXED BUS

Throughout this text, a 16-bit address bus and a separate 8-bit data bus are assumed. The 8085 microprocessor, however, multiplexes (shares) the data bus pins with the lower half of the address bus pins. The remaining eight bits of the address (the upper half) are on separate address pins. This technique reduces the pin count of the microprocessor.

The *Address Latch Enable* (ALE) signal indicates when the address/data bus contains an address. This signal is used to latch the bus contents to generate the lower half of the address bus (see Figure 10-2). IC2 is an eight-bit latch with

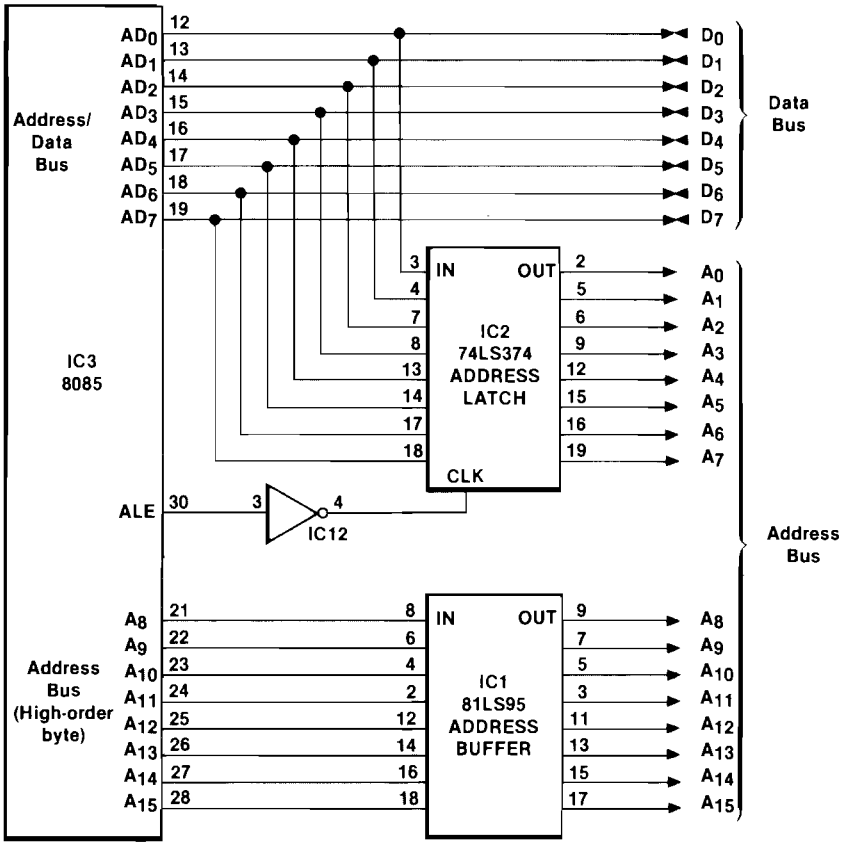


Figure 10-2. Address Demultiplexing Circuit Used for μLab

three-state outputs. It latches the address information from the address/data bus at the negative edge of ALE (the inverter IC12 is necessary to select this edge). IC1 is a simple three-state buffer and is not really part of the demultiplexing.

Figure 10-3 shows a generalized picture of the bus timing. The A8-A15 lines always contain the high-order address byte. At the beginning of each memory cycle, the low-order address byte is placed on the address/data bus. The trailing edge (high-to-low transition) of ALE indicates that the address is present and causes the demultiplexing latch (IC2) to store the low-order byte of address.

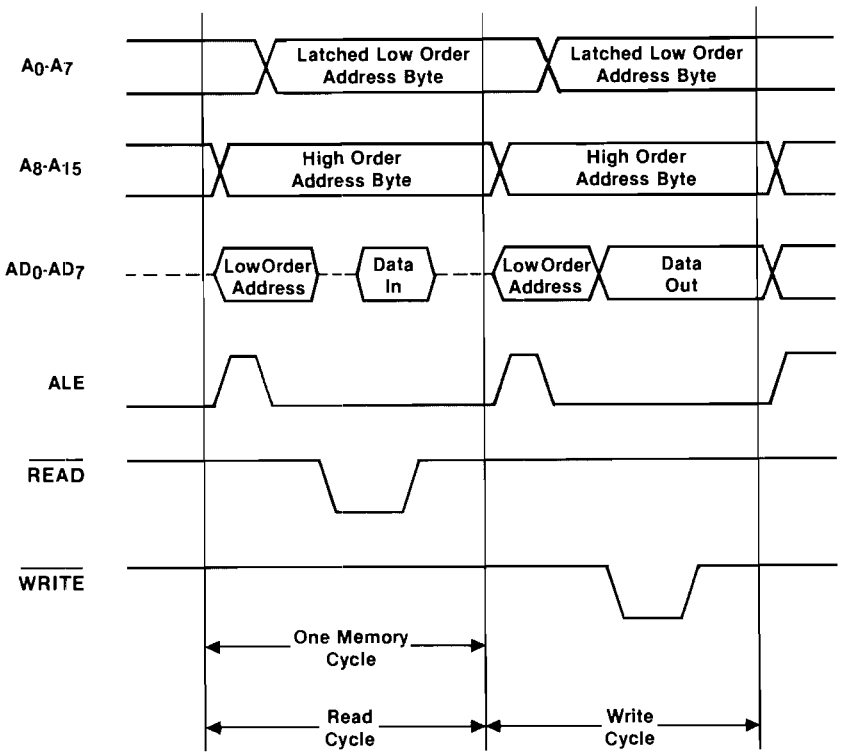


Figure 10-3. 8085 System Timing. The falling edge of ALE signals the rest of the system that the address/data bus contains a new address

The address information is then removed from the address/data bus to allow the data transfer to take place. If a read operation is in progress, the microprocessor issues a read signal, and the addressed memory or I/O device places the data on the address/data bus. At the rising edge of $\overline{\text{READ}}$, the microprocessor reads the data from the bus.

The write cycle is similar, except that the direction of the data transfer is reversed. At the beginning of the cycle, the low-order address byte is placed on the address/data bus and ALE is pulsed. Then the microprocessor issues a write pulse and places the data on the address/data bus. At the rising edge of $\overline{\text{WRITE}}$, the addressed memory device stores the data from the bus.

With the addition of the demultiplexing latch (shown in Figure 10-2), the function of the buses is identical to nonmultiplexed buses. The address/data bus simply becomes the data bus. Notice that this "data bus" contains address information early in each memory cycle. However, since the data bus is not in use at that time (neither $\overline{\text{READ}}$ nor $\overline{\text{WRITE}}$ are true), no conflict results.

THE 8085 FAMILY

The multiplexed bus frees 7 pins on the 40-pin microprocessor for other functions (16 address plus 8 data lines are replaced by 8 address, 8 address/data, and ALE). Standard memory and I/O devices are interfaced to the bus by use of a simple eight-bit demultiplexing latch.

Several memory and I/O ICs that were made specifically for use with the 8085 contain an address demultiplexing latch right on the chip. These chips have eight address/data inputs and an ALE input. They use 40-pin packages to allow I/O ports to be included on the chip. One such part (the 8155) contains a 256 byte RAM, three I/O ports, and a timer. The 8355 contains a 2K byte ROM and two eight-bit I/O ports. These chips make possible simple, powerful microcomputer systems. They are not used in the μLab because much of the bus activity is hidden inside them. They are therefore not as well suited for use in a teaching product.

EXPERIMENT 10-1

Bus Timing

CONCEPT

In this experiment, you can observe the operation of the address/data bus with an oscilloscope and see a real-life example of the multiplexing shown in Figure 10-3.

PROCEDURE

A) Connect the oscilloscope as follows:

1. Channel A to data bus line D0 (insert the probe tip into the D0 plated-through hole).
2. Channel B to ALE (IC12-3).
3. Trigger on channel A.
4. Set both input channels to 2 V/div and the sweep speed to 1 μs/div.

B) Key in the following program:

```
0800   C3     LOOP:   JMP   LOOP
0801   00
0802   08
```

This program is a single jump statement that jumps to itself. The microprocessor repeatedly executes this short loop, which enables you to get clear oscilloscope displays of the bus activty.

C) Run the program.

D) Adjust the trigger level until a stable display is obtained (see Figure 10–4). Channel A is connected

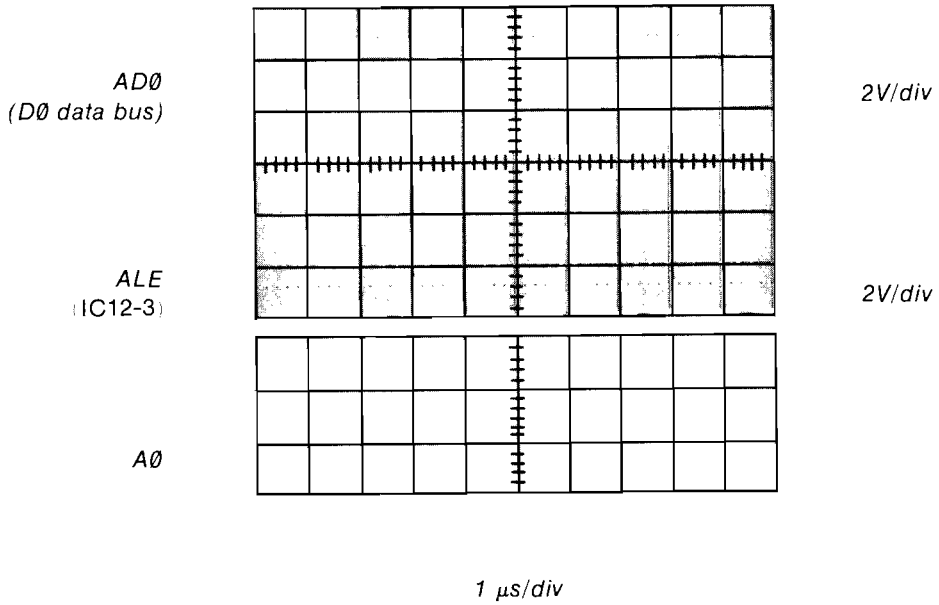


Figure 10-4. Falling Edges of ALE Indicate Stable Addresses on AD0

(Continued)

to the address/data bit 0 line (AD0). (See Figure 10-2 if this is not clear.) This waveform is somewhat messy since it contains levels between the defined one and zero levels. This condition exists because the line is sometimes floating (i.e., no device is driving it).

E) Remember that ALE signifies that an address is present on the address/data bus. At the falling edge of ALE, the AD0 line from the microprocessor is latched into the address latch to create A0. Notice that the AD0 signal is always at a stable, valid level at this edge of ALE.

F) Draw in the AØ address waveform in Figure 10-4. The waveform changes only at the falling edges of ALE, when it takes on the value of AD0.

G) Connect channel B to the A0 line (use the probe tip and the plated-through hole). Verify that the display is similar to Figure 10-5. The A0 waveform should look like the one you drew in Figure 10-4.

H) Connect channel A to A0 and channel B to ALE. Figure 10-6 shows the display. You can see that the A0 line does indeed change only at the falling edge of ALE.

I) Connect channel A back to AD0 and channel B to $\overline{\text{READ}}$ (use the $\overline{\text{READ}}$ signature analysis test point just below the row of address bus LEDs). Figure 10-7 shows the display. When $\overline{\text{READ}}$ is low, the memory puts data on the address/data bus. At the rising edge of $\overline{\text{READ}}$, the microprocessor inputs this data. Notice that the AD0 line is at a stable, valid logic level at this time.

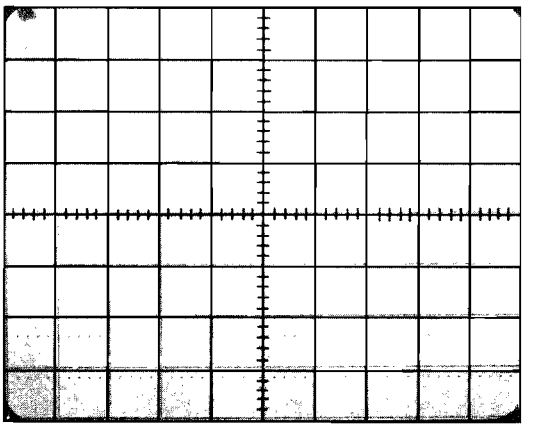


Figure 10-5. Demultiplexed AØ Address Line Generated from Multiplexed Address Line ADØ

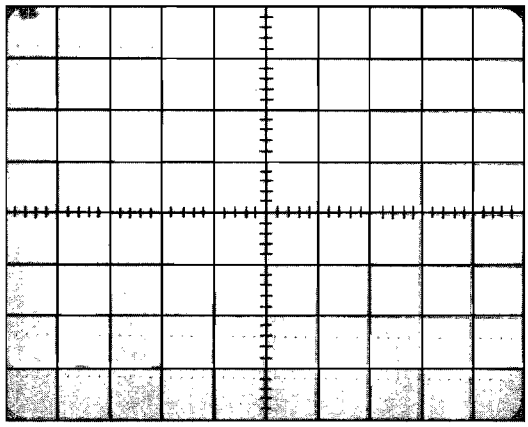


Figure 10-6. ALE Line Controls AØ

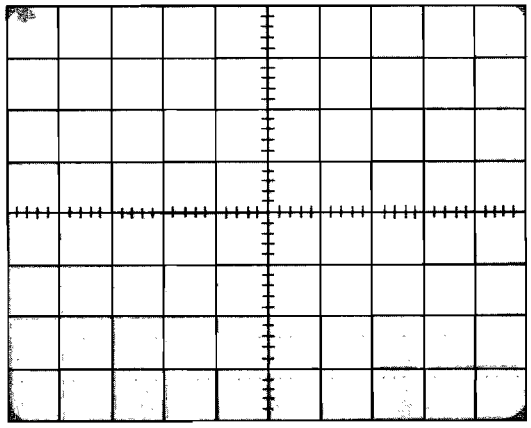


Figure 10-7. Stable Data from ADØ Read into Microprocessor at Rising Edge of $\overline{READ}$ Signal

J) Observe that there are times when $\overline{\text{READ}}$ is high and ALE is low (as shown in Figure 10-8). At these times, the address/data bus is not used, and it can be in a high impedance state. This accounts for the parts of the AD0 waveform which are not at valid logic levels.

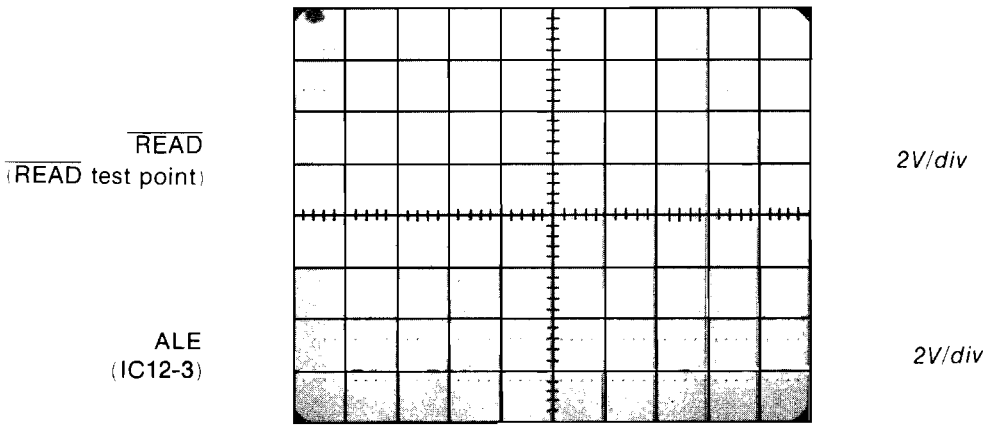


Figure 10-8. When $\overline{\text{READ}}$ is high and ALE is low, the AD0 line carries no logic information

SUMMARY

You examined the multiplexed bus signals while running a short program. The AD0 line contains valid address information at the falling edge of ALE and data information at the rising edge of $\overline{\text{READ}}$. At other times, the address/data bus is not used and may be at any level.

Clocks

All microprocessors are synchronous digital circuits and therefore require a clock. Most early processor ICs require external clock circuits to provide them with the proper clock waveforms. Voltage, phase, and timing requirements are often critical. The manufacturers of these processors (e.g., 6800, 8080) produce clock and timing ICs designed specifically to provide the clock to the processor. They are generally controlled by a quartz crystal.

Newer processors have internal clock circuits in which a crystal is connected directly to two pins of the IC to provide the clock (e.g., the 8085 used in the μLab). Alternately, when low cost is a concern and accuracy isn't important, the crystal is often replaced with an RC timing circuit. Many microprocessors can also be clocked by a TTL clock signal provided from a master system clock so that multiple processors can be synchronized.

In general, the processor operates at some fraction of the actual crystal frequency. For example, the 8085 used in the μLab has a 4 MHz crystal. The basic machine cycle, however, is 2 MHz and the Clock Out signal that the 8085 provides is also 2 MHz. Most MOS microprocessors operate in the 100 KHz to 10 MHz range. Bipolar bit-slice processors are able to run at higher clock rates.

Reset

The *Reset* pin on the 8085 is used for power-up initialization (see Figure 10-9). When a low level is applied to this pin, the microprocessor's internal circuits are cleared. The program counter is set to 0000, and program execution begins from that address. The power-up initialization routine in the ROM begins there. In a typical product (and the μLab), power-up reset performs system verification tests and then sets the peripherals to desired start-up conditions.

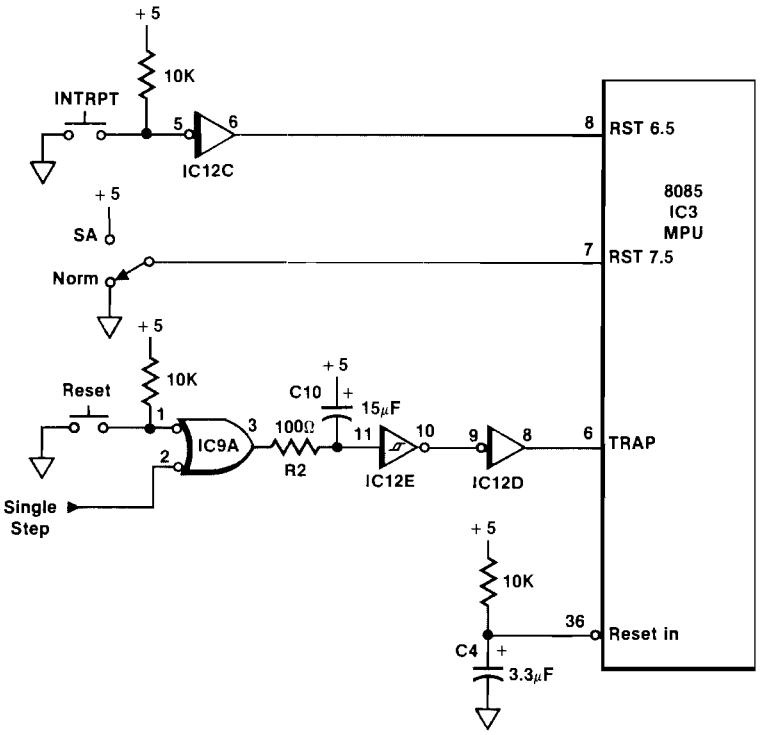


Figure 10-9. Interrupt Circuitry of μLab

A resistor is used to pull the line high, and a capacitor to ground provides an automatic power-on pulse. When power is first applied, the discharged capacitor is at a low logic level. This level is applied to the Reset input. As long as it remains low, the processor remains reset. When the pull-up resistor causes the capacitor to charge up to the threshold of the input (which has a Schmitt trigger to eliminate transition noise), the processor begins executing the program at address 0000. Notice that the μLab's RESET button does not go to the microprocessor's Reset input. If the RESET button were connected directly to the Reset input, then whenever RESET was pressed, the μLab would go through the power-up memory clear routine. This would destroy all of the programs stored in RAM. Instead, the RESET key goes to the *Trap* input line, described later in this lesson.

Status

The 8085 has two special status outputs, S0 and S1. They provide additional information about the machine cycle in progress. They also indicate whether the processor is in a HALT state. The μLab, however, does not use these signals, since they are used only for special applications.

Ready

When the Ready input goes low, it causes the microprocessor to enter a *wait* state. In this state the buses remain at their current logic state until Ready goes high again. The wait state allows slower memory and I/O devices to be used with the microprocessor. If, for example, a memory chip requires the address to be stable for 1 μs before producing valid data outputs, the microprocessor would read the memory's outputs before they were valid (it typically allows about 400 ns for the memory to respond). To avoid this error, the memory's address decoder must contain a special circuit to control the Ready line. When the slower memory is addressed, the decoder brings the Ready line low for 1 μs to ensure that the memory has sufficient time to respond.

The μLab uses the Ready line for a different purpose. In the hardware-step mode, the Ready line is pulled low immediately after each machine cycle. This freezes system activity indefinitely so that the user can observe bus and status information at each machine cycle.

Direct Memory Access

Direct Memory Access (DMA) is a method for transferring data directly between the memory and a peripheral device without going through the microprocessor. DMA provides data transfers at a much higher rate than "programmed" transfers, which use the microprocessor to pass each byte of data. CRT and disk controllers often use DMA to increase data transfer speeds.

The microprocessor's *Hold* control input is what makes this possible. When this line is high, the 8085 finishes the machine cycle it is currently executing and then stops to place a high level on the *Hold Acknowledge* (HLDA) line. All of the microprocessor's bus outputs (the address, data, and control buses) are put into the high impedance state. A peripheral device can now take control of these buses, and perform whatever data transfers are required. When the Hold line is set low by the peripheral, the microprocessor continues operation from where it was stopped.

The peripheral must have a DMA controller to issue the address and control signals. DMA controller chips that perform this function are available.

Notice that if a microcomputer system has DMA capability, all bus buffers must be three-state buffers. The HLDA line disables the buffer outputs and allows the DMA controller to use the buses.

Interrupts

Interrupts permit external hardware to request immediate action by the processor. They interrupt the usual program flow and transfer control to a special software routine. In this lesson, the hardware required to initiate an interrupt is described. The software aspects of interrupts are discussed in Lesson 6. It is assumed in this discussion that the interrupt in question is enabled by the software.

There are two groups of interrupts on the 8085. The first group (TRAP, RST 5.5, 6.5, and 7.5) is controlled by individual pins on the microprocessor. They are called *single pin interrupts*. The second group (RST 1, 2, 3, 4, 5, 6, and 7) is controlled by INTR and $\overline{\text{INTA}}$.

To initiate one of the interrupts in the first group, you simply apply a signal to the corresponding pin on the microprocessor. The interrupt service routine for that pin is then automatically addressed. Table 10–1 shows the address associated with each pin, as defined by the 8085 microprocessor design.

Interrupt Pin	Interrupt Address
TRAP	0024
RST 5.5	002C
RST 6.5	0034
RST 7.5	003C

The interrupt inputs each respond to a different logic level or edge. The RST 5.5 and 6.5 interrupts respond to a high level (logic 1). The RST 7.5 interrupt responds only to a positive edge, (i.e., a transition from low to high). The TRAP interrupt responds to a high level but does not acknowledge a second time until the logic level goes low and then high again.

The μLab uses the Trap input for the RESET button, the RST 6.5 input for the INTRPT button, and the RST 7.5 input for the "SA" switch (see Lesson 17 for a discussion of this switch). The circuits are straightforward, with the exception of the TRAP input (see Figure 10-9). An OR gate with active low inputs (actually a NAND gate) allows the single-step circuit to access the TRAP input, which is debounced by the hardware (R2, C10, and Schmitt input IC12E). The 100 ohm resistor and the capacitor debounce the RESET key to ensure that it causes only one interrupt each time it is pressed. It is not necessary to debounce the other interrupt inputs because the software can disable the interrupt as soon as it is acknowledged to prevent a second interrupt from occurring.

The use of the second group of interrupts is more complicated. The *Intr* (Interrupt Request) input initiates the interrupt. The circuit requesting the interrupt provides a code on the data bus indicating which interrupt service address to jump to. The $\overline{Inta}$ (Interrupt Acknowledge) pin on the microprocessor coordinates the timing of this operation. The Intr interrupt is not used in the μLab.

Interrupt Priorities

Allowance is made for the fact that more than one interrupt can be requested simultaneously. Each interrupt is assigned a priority, and the interrupt with the highest priority is acknowledged first. TRAP has the highest priority, followed by RST 7.5, 6.5, and 5.5, in that order. INTR has the lowest priority.

THE SINGLE-STEP CIRCUIT

The μLab contains a special control circuit to provide the single-step functions. Figure 10-10 shows the portion of the single-step circuit used to advance the

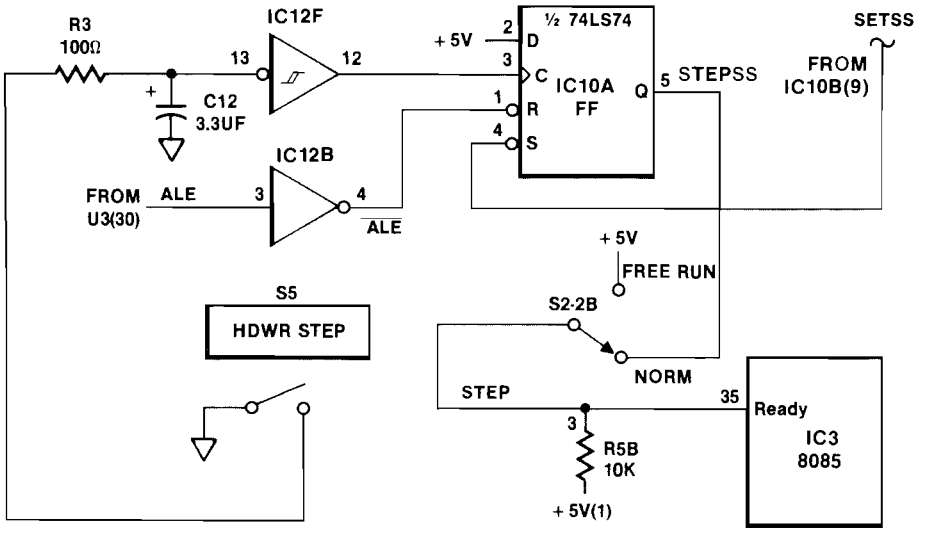


Figure 10-10. Single-Step Circuit of μLab Advances Microprocessor One Machine Cycle

microprocessor one machine cycle each time the HDWR STEP key is pressed. The sequence of events for this circuit is as follows:

1. When the monitor program is running, the SETSS (Set Single-Step) control signal (IC10-9) is low, forcing the STEPSS (Step Single-step) signal (IC10-5) to be high. This line is connected to the Ready input of the microprocessor (IC3-35) through switch S-2. The high logic level on the Ready line allows the system to run at full speed.

2. When a valid address appears in the μLab's display and HDWR STEP is pressed, the monitor program responds by transferring operation to the user program at the address specified and then setting the SETSS line high.

3. At the next machine cycle, the $\overline{\text{ALE}}$ line will pulse low and reset latch IC10A causing the STEPSS line to go low. This in turn forces the Ready line low.

4. With the Ready line low, all activity on the system stops. The μLab is now in the hardware single-step mode. The address previously on the display is now indicated on the address bus LEDs.

5. When the HDWR STEP key is pressed again, a rising edge appears at clock input pin IC10-3 causing the STEPSS line and the Ready line to go high. The HDWR STEP key is debounced by R3, C12, and IC12F.

6. With the Ready line high once again, the microprocessor resumes normal program execution, going on to the next machine cycle.

7. As soon as the next machine cycle occurs and a new address is latched, the $\overline{\text{ALE}}$ line pulses low. Again, this causes latch IC10A to reset and the STEPSS line to go low.

8. This sequence is repeated each time the HDWR STEP key is pressed.

9. When the RESET key is pressed, IC10B is reset causing the SETSS line to go low. This in turn forces the STEPSS signal high. The RESET signal is also sent to the Trap Interrupt of the microprocessor (IC3-6). This interrupt returns system operation to the monitor program.

PROGRAMMABLE TIMERS

One common use of interrupts is with a programmable interval timer. Basically, it is a counter, which is started by a command from the microprocessor and then counts at a known rate (usually crystal controlled). When it reaches the number of counts programmed, it interrupts the microprocessor. The microprocessor uses this counter interrupt to perform accurate timing functions without the need for timed program loops (as used by the μLab). The microprocessor can execute other programs while this timing function is operating. Other timing functions can also be performed with many of the timer ICs available.

In the μLab, the software timing loops used in the display scan routine could have been replaced with a timer circuit. In freeing-up the microprocessor from this time-consuming task, much longer programs could then have been executed between each display scan instruction.

ELECTRONIC CONSIDERATIONS

One of the attributes of digital systems arises from the fact that they can be analyzed from a logical viewpoint. You can think purely in terms of on and off, one and zero, or yes and no. Unlike analog circuits, they use very little mathematical or electronic theory. The ICs are often thought of as black boxes. However, some electronic factors must be taken into account to ensure that the logic operates logically.

Loading

One of the most important factors to consider is output loading. A gate's output can supply only a limited amount of current, and, if forced to supply more, the circuit may not work properly. If the circuit is severely overloaded, logic levels may change or a gate may overheat and burn out.

The two types of loading generally considered in microprocessor systems are static and dynamic. Static loading results from resistive and current components on logic nodes. Dynamic loading is primarily the result of node capacitance. Static loading of MOS outputs that drive multiple TTL inputs is usually the primary area of concern. However, when a large number of devices are on a bus and the bus extends through connectors to multiple boards, dynamic (capacitive) loading can become important. Whereas static loading generally affects logic voltage levels and noise margins, dynamic loading affects speed and timing by slowing down level transitions.

The static loading problem is particularly severe in microprocessor systems because many devices can be connected to the same bus. The microprocessor's address outputs must be able to drive all the devices that are connected to the address bus. Every device that drives the data bus must be capable of driving all the devices connected to the bus.

The μLab uses buffers on all the address lines because it does not have enough output drive current available to power the address bus LEDs. The data bus is

only partially buffered, as shown in Figure 10-11. To buffer the data bus right at the microprocessor would require a bidirectional buffer, since data passes in both directions on the data bus. Instead, the μLab uses a unidirectional buffer (IC14) to generate a buffered "data out" bus that drives the output ports and the data bus LEDs.

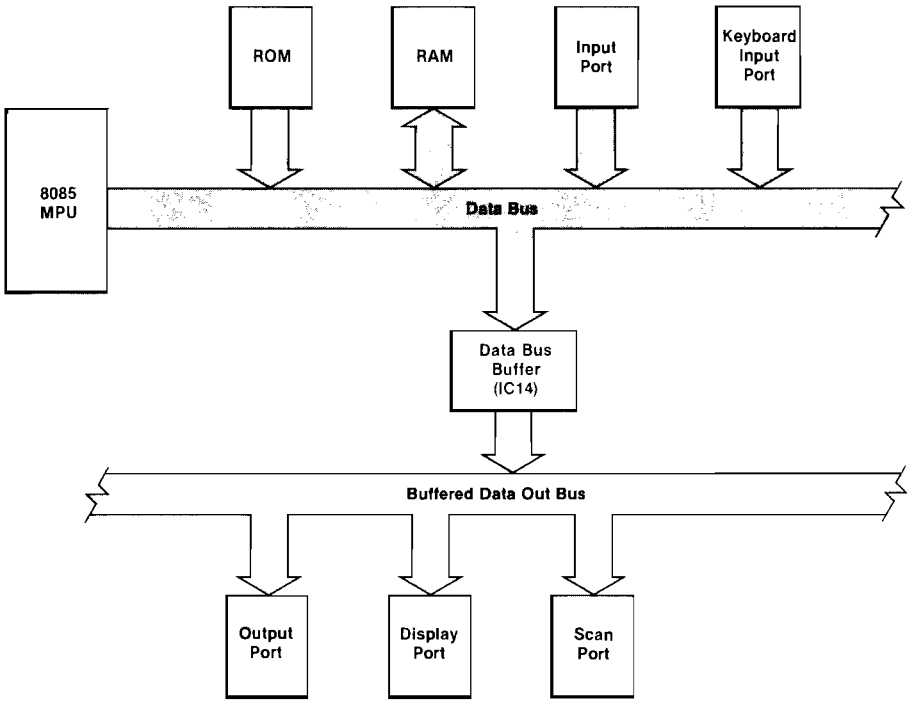


Figure 10-11. Data Bus Buffer of μLab

Although the μLab uses noninverting buffers for the address, data, and control buses, many systems use inverting buffers, which generate negative logic buses. Negative logic presents no problem so long as you are aware of it, as in the μLab's output port LEDs (the LEDs indicate negative logic). Inverting buffers are used because they are often cheaper, faster, or consume less power than non-inverting buffers.

Bypassing

As in all digital systems, care must be taken to keep the power supply and ground lines as noise-free as possible. Bypass capacitors (usually .01 or 0.1μF) are scattered over PC boards. In addition, a 10 μF or greater tantalum capacitor is placed on the circuit board for the 5 volt supplies.

Timing

There are many timing relationships that must be satisfied for a microprocessor-based system to operate correctly. Most of these are controlled internally by the microprocessor so that signal flow among the devices in the system can be coordinated. The address, data, and control buses all adhere to strict timing relationships.

Figure 10-12 shows the timing for a write operation. The address must be stable for a given period of time (called the *access time*) before any operation may be performed, to allow the memory's internal address decoders to select the specified memory cells. The data must then be stable for an additional period of time

(called the *set-up* time) before the write occurs. The data must also be stable for a period of time (called the *hold time*) after the write pulse. Finally, the write pulse must have a minimum duration.

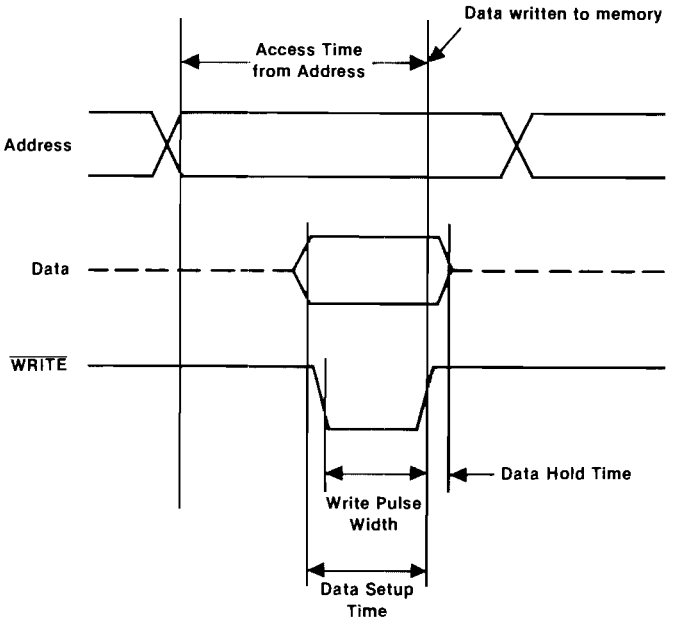


Figure 10-12 Data Stored in Memory on Rising Edge of WRITE Signal

Figure 10-13 shows the timing for a read operation. As with the write operation, the address must be stable long enough to allow the memory's internal decoders to settle. A read pulse is then generated, and, after a period of time (the *data access* time), the memory places the addressed data on the data bus. This data must be stable for the set-up time before the rising edge of READ, when the data is read into the microprocessor. The data must also remain stable for the data hold time.

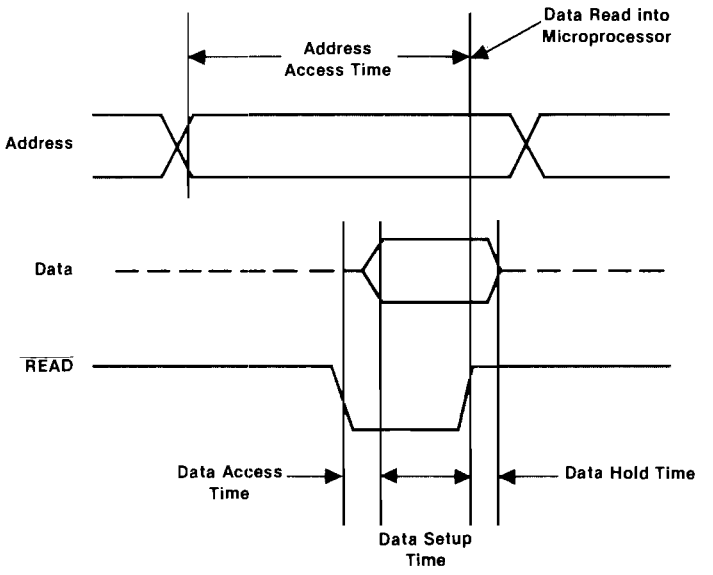
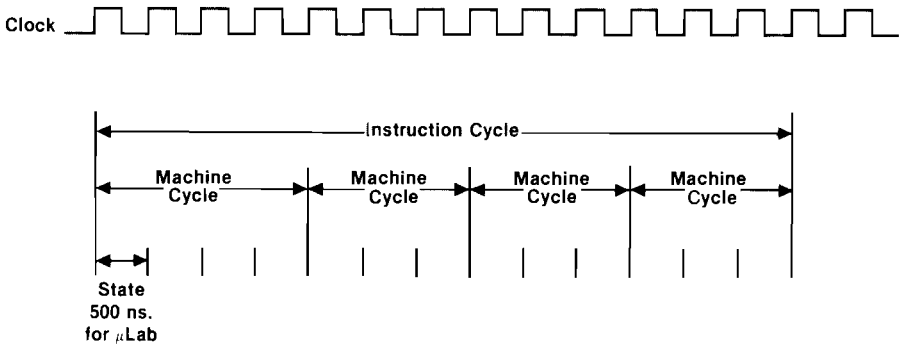


Figure 10-13. Data Read into Microprocessor on Rising Edge of READ signal

Figure 10-14 shows the CPU timing for a typical instruction. The basic unit of time is the state, which is one clock period. A machine cycle consists of from three to six states. Most simple operations (such as moving one register to another or reading a memory location) require one machine cycle. The instruction cycle is the time required to execute an entire instruction and consists of from one to five machine cycles.



Note: This is only a typical instruction cycle; there are many variations.
An instruction cycle may contain one to five machine cycles.
A machine cycle may contain three to six states.

Figure 10-14. Typical Instruction Cycle

Figure 10-15 gives the complete system timing for an OUT instruction. States are denoted by T1, T2, and so on, and machine cycles by M1, M2, and so on. This diagram combines all the timing discussed earlier: the multiplexed bus, reads, and writes.

In the first machine cycle (M1) the opcode is fetched from the memory. In M2 the second byte of the instruction (the port address) is read from the memory. In M3 the instruction is executed: data is written to the I/O port.

All instructions require one machine cycle to fetch the opcode. For simple instructions that do not use the memory or I/O (such as register transfers), the execution is performed during the first (and only) machine cycle. For multiple-byte instructions (such as MVI A,7 or STA 0837), one machine cycle is required to read each byte of the instruction. If the instruction execution requires a reference to memory or I/O, the execution requires an additional cycle. Some complex operations use an extra machine cycle to execute the instruction, even though the operation is internal to the microprocessor.

The μLab uses these cycles for its single-step modes. HDWR STEP steps one machine cycle at a time, and INSTR STEP steps one instruction cycle.

Although the details of timing may vary in any specific situation, these are the basic timing considerations. The microprocessor manufacturers have worked out all the timing details for families of parts in a system so that they are generally compatible in this respect. Care must be taken, however, when using general-purpose parts with a specific microprocessor.

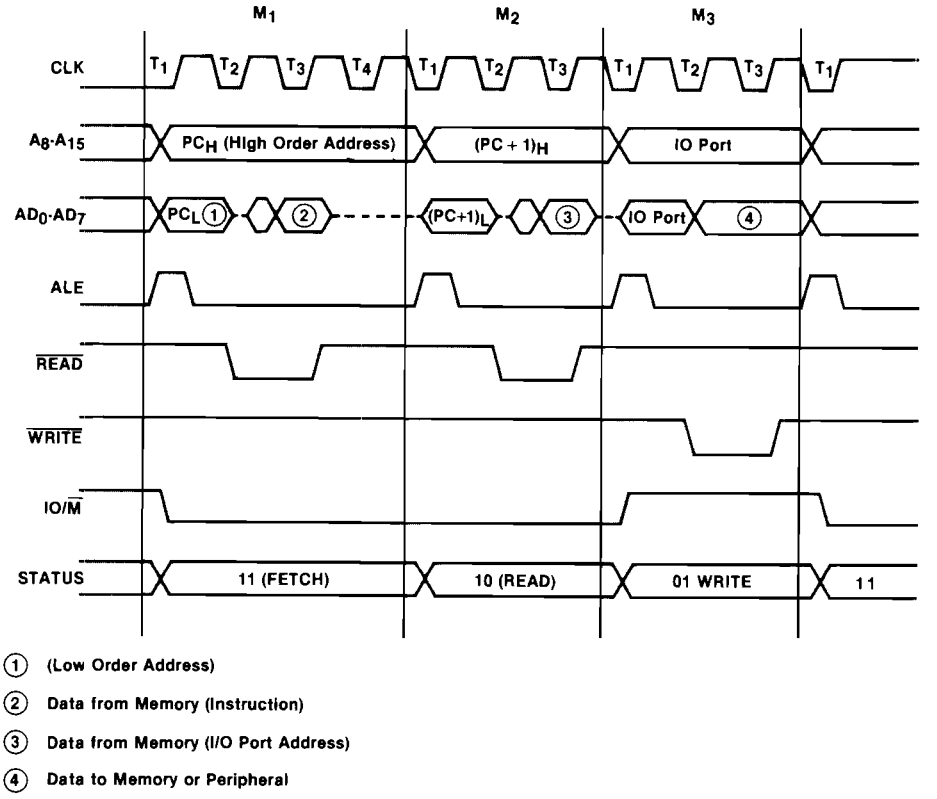


Figure 10-15. Timing for Fetching and Executing OUT Instruction

REVIEW

Lesson 10

The control port in the μLab is controlled by the microprocessor and memory protection and single-step circuits. The multiplexed bus shares eight pins of the microprocessor between the address and data bus. Another pin (ALE) indicates when address information is on these lines. The Reset pin is used for power-up initialization. The two status pins provide advance control information. The Ready line is used to force the microprocessor to wait for slow devices.

Direct memory access allows high-speed data transfers to occur between memory and an external circuit by bypassing the microprocessor. Interrupts allow the external hardware to request the microprocessor to perform special operations. Electrical loading, bypassing, and timing are important factors in microprocessor system design and troubleshooting.

QUIZ

Lesson 10

1. An advantage of using a multiplexed bus in the 8085 is:
   a. higher system speed.
   b. more pin functions are available on the processor.
   c. simpler control hardware.
   d. all of the above.

2. When the $\overline{\text{WRITE}}$ line goes from low to high, the address/data bus contains:
   a. valid data.
   b. valid address.
   c. valid control signals.
   d. unstable information.

3. When memory devices are slower than the microprocessor, the
   a. HOLD
   b. TRAP
   c. INTR
   d. READY

   pin is used to tell the microprocessor to wait.

4. An advantage of DMA is that:
   a. faster data transfers can occur.
   b. the microprocessor controls the data transfers.
   c. the data transfer circuits are less complex.
   d. slower memory can be used.

5. If the RESET key on the μLab were connected to the Reset pin on the microprocessor, it would cause:
   a. a hardware conflict
   b. the buses to disable
   c. the same effect as being connected to TRAP
   d. the memory to be cleared

   when RESET was pressed.

6. An advantage of the single pin interrupts is that:
   a. they reduce the pin count of the microprocessor.
   b. the interrupt circuit is simplified.
   c. they do not specify their own interrupt service routine address.
   d. all of the above are true.

QUIZ

(Continued)

7. The primary purpose of assigning priorities to interrupt lines is to:
   a. select the interrupt routine address.
   b. determine which interrupts are used most often.
   c. specify which interrupt is to be selected when more than one occurs.
   d. prevent the microprocessor from simultaneously executing more than one interrupt routine.

8. In a system with many MOS devices, the main bus loading factor is likely to be:
   a. capacitive.
   b. resistive.
   c. current.
   d. static charge.

IV
MICROPROCESSOR SOFTWARE

This section provides a more detailed look at microprocessor software. A representative selection of 8085 assembly language instructions is discussed, with examples and experiments to demonstrate their use. Lesson 13 describes the software development process, and an example is presented. The next lesson presents programs to control the keyboard and display. Finally, representation techniques for large and fractional numbers are described, along with techniques for calculating complex mathematical functions.

A detailed knowledge of software is not absolutely required for troubleshooting microprocessor systems. However, a good general knowledge of software is necessary to thoroughly understand the system. Therefore, while this section may be omitted without loss of continuity, it should be studied for a fuller understanding of microprocessor systems.

LESSON 11

Registers and Breakpoints

INTRODUCTION

This lesson provides background material for the more detailed discussion of microprocessor software presented later in this section. The instructions that have already been described are summarized, and a few new ones are introduced. The microprocessor's registers are described, and the μLab's FETCH REG key is used to examine and modify them. Finally, the use of the breakpoint as a software debugging tool is described.

INSTRUCTION REVIEW

Several of the 8085's instructions have already been discussed. In this lesson, these instructions are reviewed to provide a foundation from which to describe some new instructions. For a complete description of the 8085 instruction set, refer to Appendix B.

Some shorthand notation is useful in describing instructions. In the following text, the term *data* is used to indicate any eight-bit quantity, and *adrs* to indicate any sixteen-bit address.

Data Manipulation: MVI, INR, CMA

One of the most fundamental microprocessor operations is to load the accumulator with data. This is done by the *MVI A,data* instruction (move immediate to the accumulator). The data to be moved to the accumulator is stored in the byte following the opcode.

Once the data is in the accumulator, instructions are needed to manipulate it. The two instructions that have been used so far are *INR A* (increment accumulator) and *CMA* (complement accumulator).

Testing and Jumping: CPI, JMP, JZ

To test the value in the accumulator, the *CPI data* (compare immediate) instruction can be used. This compares the data specified in the second byte of the instruction with the contents of the accumulator and sets the processor flags accordingly. The only flag that you have used so far is the zero flag, which is set if the result of an operation is zero. The *JZ adrs* (jump if zero) instruction tests the zero flag (presumably set by a previous instruction, such as CPI) and causes a jump if the flag is set. There is also an unconditional jump instruction *JMP adrs* that causes a jump regardless of the state of the flags. The address for both jump instructions is stored in the two memory bytes following the opcode.

Memory and I/O: LDA, STA
The *LDA adrs* and *STA adrs* (load accumulator and store accumulator) instructions transfer data between the accumulator and memory or I/O ports. The address of the memory location or I/O port is specified in the two bytes following the opcode.

Subroutines: CALL, RET
To use subroutines, two more instructions are needed. *CALL adrs* is used to jump to a subroutine, and *RET* (Return) is used to end a subroutine. The CALL instruction specifies an address exactly like a jump instruction. The RET instruction does not specify an address, but causes a jump to the instruction that follows the previously executed CALL.

Interrupt Control: SIM, EI, DI
Control of the interrupts requires three instructions. *SIM* (set interrupt mask) is used to specify which interrupts should be enabled and which should not. It copies the contents of the accumulator into the processor's Interrupt Mask register. *EI* (enable interrupts) causes the selected interrupts to be enabled. *DI* (disable interrupts) disables all interrupts.

THE VARIETY OF INSTRUCTIONS

This relatively small set of instructions demonstrates most of the 8085's basic capabilities. As you become familiar with more instructions, it will become apparent that there are more instructions than are necessary. The variety of instructions available makes it easier to write programs, since you can choose from several alternatives. There is a direct parallel in hardware design: it is possible to build any logic circuit using only NAND gates. In fact, entire computers have been built in this way. However, the system is greatly simplified by using other devices such as NOR gates, flip-flops, multiplexers, counters, and adders.

THE GENERAL-PURPOSE REGISTERS

Up to this point, one major feature of the 8085 microprocessor has been ignored: its general-purpose registers. There are six eight-bit registers within the 8085, which can be used for temporary data storage. Figure 11-1 shows the 8085 block diagram including these registers. They are called the B, C, D, E, H, and L registers. The stack pointer is also shown.

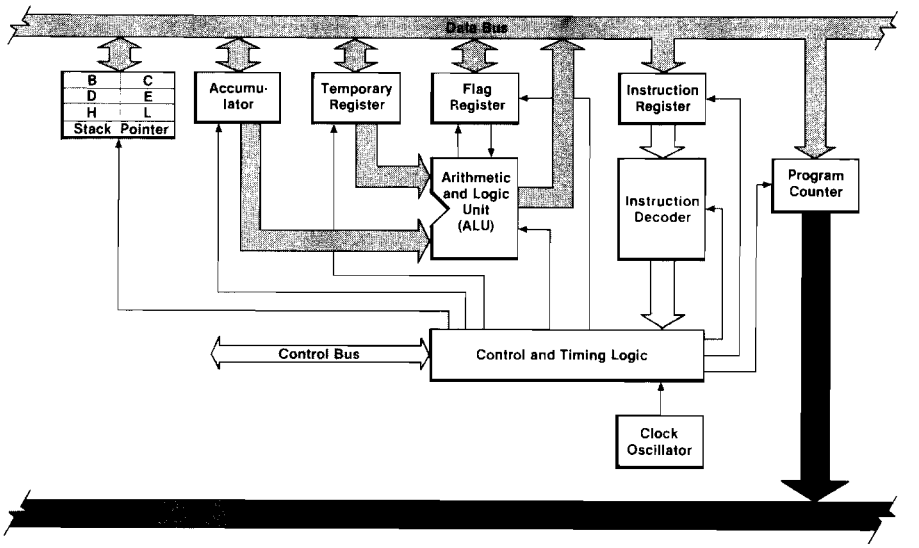


Figure 11-1. Simplified Block Diagram of 8085 Showing General-Purpose Registers

To use these registers, some new instructions are needed. The MVI instruction, which loads data into the accumulator, can in fact be used with any register. For example, *MVI D,data* causes the data to be moved to the D register. The general form of this instruction is *MVI r,data* where *r* indicates any of the registers (A, B, C, D, E, H, or L). Although the accumulator is special in that it is used for the results of computations, it may also be used as a general-purpose register.

The INR instruction can also be used on any register. The general form is *INR r.* For example, INR H increments the H register.

Now that all these registers are available, it is useful to have a way to move data from one register to another. The general form of the instruction that does this is *MOV r1,r2*. Register *r1* is the destination, and register *r2* is the source. For example, MOV A,H moves the contents of the H register into the accumulator, but the contents of H are not changed. Note that the source and destination are listed in the opposite order from what you might expect. You can think of the instruction MOV A,H as "move into the accumulator the contents of the H register".

The Assembly Language Reference Card lists all of the 8085's instructions in mnemonic and hexadecimal forms and shows all the various MOV instructions. This is your guide for translating assembly language mnemonics into hexadecimal machine code, and vice versa. It also provides a convenient list of all available instructions.

The general-purpose registers are useful when a program uses several different variables. Each register can be used for a different purpose. RAM locations are not needed for data storage as long as the six registers are sufficient. For example, a program that counts six different events can use one register for counting each event.

INDIRECT ADDRESSING

Notice that on the reference card there is an *M* "register" listed in the MOV instructions. This is not really a register, but refers to a memory location whose address is stored in the H and L registers. The H and L registers hold an address that points to a location in memory. This is called *indirect addressing*, which means that the instruction specifies where the address is stored (the H and L registers in this case) rather than the actual address.

For example, if H contains 12 and L contains 37 (see Figure 11-2), the instruction MOV A,M will load the accumulator with the contents of memory location 1237. The effect is exactly the same as the instruction LDA 1237.

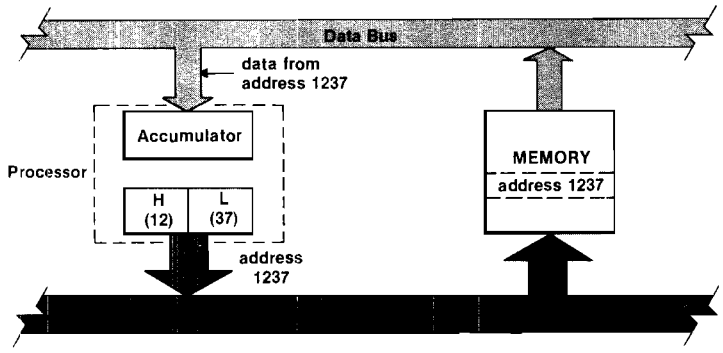


Figure 11-2. Indirect Addressing using H and L Registers

This is an example of how the same operation can be performed in two different ways. MOV A,M is a single-byte instruction, but requires that the H and L registers be previously set to the desired address. LDA 1237, on the other hand, is a three-byte instruction. However, it is often preferable because it does not require that the address be stored in the H and L registers. Indirect addressing is particularly useful for table-oriented operations, such as the table look-up described in Lesson 15.

EXPERIMENT 11-1

Using the Registers

CONCEPT

A short program is entered to demonstrate the use of the general-purpose registers and the operation of the MOV instruction. The FETCH REG key is used to follow the execution of the program.

PROCEDURE

A) Key in the program in Table 11-1. This program first sets the B register to 37 and then copies it into the H register. Finally, the H register is incremented.

Address	Contents	Instruction	Comments
0800	06	MVI B,37	;Set B to 37
0801	37		
0802	60	MOV H,B	;Move B to H
0803	24	INR H	;Increment H

Table 11-1. Register Demonstration Program

B) Verify that the program is correctly stored.

C) Press [FETCH ADRS] [0] [8] [0] [0] [INSTR STEP]. The MVI B,37 instruction is executed.

D) Press [FETCH REG]. The display shows the contents of the A register.

E) Press [STORE INCR] twice. The display shows the contents of the B register (37).

Pressing [FETCH REG] puts the μLab into the register mode and displays the contents of the A register. Then [STORE INCR] and [DECR] are used to select a particular register. Table 11-2 shows the order in which the registers are displayed.

Continued

Abbreviation	Description
A	Accumulator
FL	Flags
B, C, D, E, H, L	General-purpose Registers
SPH	Stack Pointer High-order byte
SPL	Stack Pointer Low-order byte
PCH	Program Counter High-order byte
PCL	Program Counter Low-order byte
I	Interrupt status

Table 11-2. 8085 Registers

F) Press [DECR] five times. The display shows PCH (Program Counter High-order byte). Since the PC is sixteen bits long, it must be displayed in two parts.

G) Press [STORE INCR] to see the lower half of the PC. The PC contains the address of the next instruction to be executed (0802).

H) To return to the execution of the program, press [FETCH PC] to set the address back to 0802, and then [INSTR STEP] to execute the MOV H,B instruction.

I) Press [FETCH REG] to put the μLab into the register mode. Then press [STORE INCR] twice to view the B register. It is unaffected by the MOV H,B instruction, so it still contains the data 37.

J) Press [STORE INCR] four more times to view the H register. It now contains 37, the data from register B.

K) Press [FETCH PC] to return to the program. Press [INSTR STEP] to execute the INR H instruction.

L) Press [FETCH REG] and then use [STORE INCR] to view the contents of the H register. Its contents have been incremented by the INR H instruction.

SUMMARY

This experiment demonstrated the use of the microprocessor's general-purpose registers. A program was entered that set the B register to 37, copied the data to the H register, and then incremented the H register. Then you stepped through the program and verified the operation of each instruction.

The FETCH REG key was used in combination with STORE/INCR and DECR to view the 8085's registers. This makes it possible to verify the operation of each instruction by stepping through the program and examining the registers of interest after each instruction is executed.

BREAKPOINTS

Single-stepping through a program allows you to stop between instructions and observe the effects of each instruction. However, in many cases it is more convenient if the program runs at full speed and then stops at a particular point of interest. This can be done by inserting an instruction in the program that causes the processor to jump from your program to the monitor program. This instruction is called a *breakpoint*. Once control has returned to the monitor, all of its facilities (such as examining or modifying registers or memory locations) are available to you.

The μLab uses the Restart 1 (RST 1) instruction for the breakpoint. It is similar to a CALL instruction except that it does not explicitly specify an address. RST 1 is equivalent to CALL 0008; the address 0008 is fixed by the design of the 8085 and cannot be changed. The subroutine beginning at this ROM location saves the contents of the registers in RAM and then returns to the monitor program.

USING BREAKPOINTS

Breakpoints are a valuable tool for debugging programs. They are inserted at key points in the program to facilitate program debugging. Breakpoints can be inserted after a program is written by replacing a program instruction with a breakpoint. The breakpoint stops the program at the desired point, so you don't need to single-step through all the preceeding instructions. You can then examine the registers and memory to see if the program operates as expected. Once the program operates correctly, the breakpoints can be replaced by no-ops (or by the instruction which was displaced by the breakpoint).

HARDWARE BREAKPOINTS

The breakpoint used in the μLab is a software breakpoint, implemented with a breakpoint instruction. There are also hardware breakpoints, which are implemented with a special logic circuit that monitors the address bus. When it senses the breakpoint address, it halts the processor, using its control inputs. Software breakpoints can be used only for programs in RAM, since the breakpoint instruction must be stored as part of the program. Hardware breakpoints, on the other hand, can be used for programs in ROM or RAM.

Using Breakpoints

CONCEPT

In this experiment you insert a breakpoint in the counter program and examine the accumulator each time through the loop. Instead of repeatedly pressing the step key (as in Experiment 5-1) to go through the loop, all you do is press RUN. When the μLab executes the breakpoint instruction, the program stops, and control returns to the monitor.

PROCEDURE

A) Key in the program in Table 11-3. This is the counter program from Experiment 5-1, with a breakpoint instruction added after the STA instruction.

Address	Contents	Label	Instruction	Comments
0804	3E		MVI A,0	;Set A to zero
0805	00			
0806	32	LOOP:	STA 3000	;Output A to port
0807	00			
0808	30			
0809	CF		RST 1	;Breakpoint
080A	3C		INR A	;Increment A
080B	C3		JMP LOOP	;Loop back
080C	06			
080D	08			

Table 11-3. Counter Program with Breakpoint

B) Verify that the program is correctly stored.

C) Press [FETCH ADRS] [0] [8] [0] [4] [RUN]. The microprocessor executes the program up to the breakpoint and then returns to the monitor. Since all the output port LEDs are on, you know that the instructions were executed. The display shows the address of the next instruction to be executed (080A).

D) Press [RUN]. The program continues until it reaches the breakpoint again. The output port LEDs indicate that the loop has been executed again.

E) Press [RUN] several times to see the LEDs count.

EXPERIMENT 11-2

Continued

F) Press [FETCH REG]. The contents of the accumulator correspond to the value shown in the output port LEDs.

G) Press [FETCH P C] [RUN]. The program executes the loop once more.

H) Press [FETCH REG]. You can now see the new value in the accumulator.

I) Repeat steps G and H several times and verify that the LEDs and the accumulator count together.

SUMMARY

This experiment illustrated how to use a breakpoint to observe the operation of the counter program. You inserted a breakpoint instruction at the point at which you wanted to stop the program. The breakpoint instruction allowed you to monitor the program's operation up to that point. You saw less step-by-step detail than with the single-step modes, but you pressed only one key (RUN) for each pass through the program.

Lesson 11

The basic set of instructions that has been presented allows data to be exchanged with memory and I/O and modified in several ways. The call and return instructions facilitate the use of subroutines. The large group of MOV instructions allows data to be moved among any of the 8085's general-purpose registers (A,B,C,D,E,H and L). These registers are convenient places for temporary data storage. The "M register" provides indirect addressing capability.

Breakpoints allow a program to be executed at full speed and then stopped at the desired point. The contents of the registers and the memory can then be examined or modified. This allows the correct operation of the program to be verified without stepping through each instruction individually.

QUIZ

Lesson 11

1. The B,C,D,E,H and L registers are used primarily for:

   a. specifying which interrupts should be enabled.
   b. temporary data storage.
   c. controlling the stack.
   d. specifying the next instruction.

2. The M "register" is:

   a. a general-purpose register.
   b. the instruction register.
   c. the program counter.
   d. the memory location pointed to by H and L.

3. To examine the contents of the C register, you would press FETCH REG and then _____.

4. Breakpoints are used for:

   a. stopping a program at a desired place.
   b. calling subroutines.
   c. manipulating the stack.
   d. executing each instruction individually.

LESSON 12

The 8085 Instruction Set

INTRODUCTION

This lesson describes some additional instructions for programming the 8085. The complete instruction set is not described, but a representative group is presented, including logical and arithmetic instructions. The μLab is used to observe the operation of the instructions. The use of the stack is also discussed.

LOGICAL INSTRUCTIONS

One of the most common building blocks for digital hardware is the logic gate. Four basic gate functions are NOT, AND, OR, and exclusive-OR. Each of these functions can also be performed by software.

The NOT function is performed by the *Complement Accumulator* (CMA) instruction, which was demonstrated in Experiment 4-3. Each bit of the accumulator is inverted.

The AND function is performed by the *And Accumulator* (ANA r) instruction. For example, ANA D causes the contents of the D register to be ANDed with the contents of the accumulator. The result is left in the accumulator. Note that the other register (D in this example) is not changed. Each variation (to operate on each register) has its own opcode.

Figure 12-1 shows a hardware equivalent of the ANA instruction. The AND function is performed individually on each bit of the accumulator. For example, if A = 1011 0110 and D = 0011 1100, then the result of the instruction ANA D is:

```
        0011 1100  (D register)
AND     1011 0110  (Accumulator)
        ---------
        0011 0100  (Accumulator)
```

The OR function is performed by the *OR Accumulator* (ORA r) instruction. Exclusive-OR is performed by XRA r. The operation of these instructions is similar to the ANA instruction, except that the logic function is different.

Since no address or data is specified, all of these instructions require only a single byte of code. Refer to the instruction set fold-out for the list of opcodes.

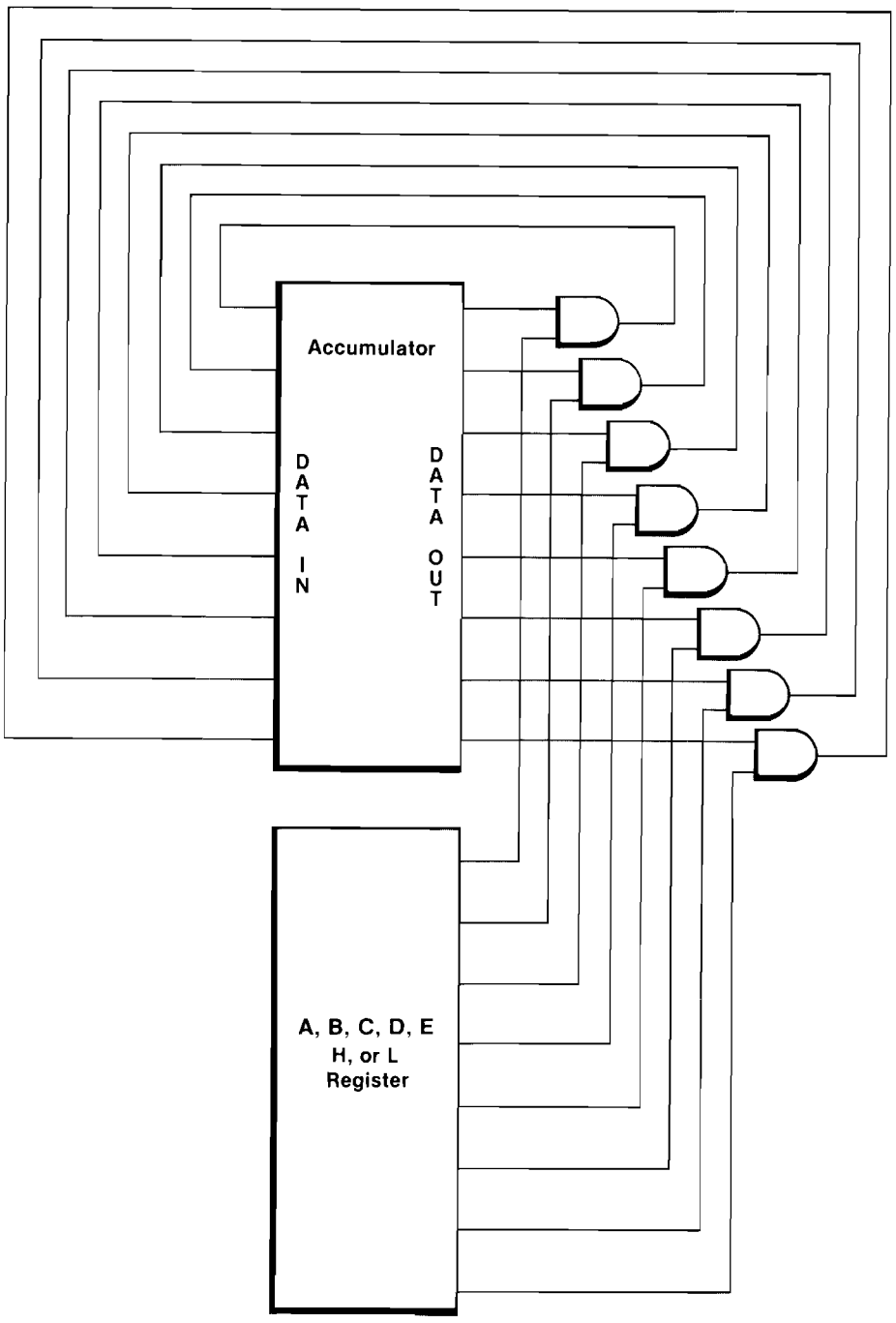


Figure 12-1. Hardware Equivalent of ANA Instruction

EXPERIMENT 12-1

Logical Instructions

CONCEPT

In this experiment, you enter and run a program to demonstrate three logical functions: AND, OR, and exclusive-OR. Data is input from the input port switches and then modified by a logical instruction. The output port LEDs display the results.

PROCEDURE

A) Translate the program in Table 12-1 to machine code and key it into the μLab. Each mnemonic must be converted to the appropriate opcode (from the reference card) and followed by the required data (if any). A completed program listing appears in Appendix A.

This program reads the data from the input port switches and then ANDs it with the value 0011 1100. The result is written to the output port LEDs. The program then loops back so that you can change the input and verify that the output responds.

Address	Contents	Label	Instruction	Comments
0800	_____	START:	LDA 2000	;Read input port
0801	_____			
0802	_____			
0803	_____		MVI B,3C	;Set B register to 0011 1100
0804	_____			
0805	_____		ANA B	;AND accumulator with B
0806	_____		STA 3000	;Send to output port
0807	_____			
0808	_____			
0809	_____		JMP START	;Repeat
080A	_____			
080B	_____			

Table 12-1. Program to Demonstrate Logical Instructions

B) Verify that the program is correctly stored.

C) Press [FETCH ADRS] [0] [8] [0] [0] [RUN]. The program is now running.

D) Set all the input switches to 1 (up). The output LEDs show the pattern 0011 1100 (0 = on, 1 = off).

E) Change the positions of the input switches and verify the output response. The bits that are ANDed with zeros will not be affected by the input switches.

F) Press [RESET] to return control to the monitor. Notice that the input switches no longer affect the output LEDs. Change the ANA B instruction (at address 0805) to ORA B (opcode B0).

G) Run the program.

H) Observe the relationship between various settings of the input switches and the pattern in the output LEDs. Note the difference between the AND and OR functions. The bits that are ORed with zeros are affected by the input switches.

I) Press [RESET] . Change the ORA B instruction to XRA B (opcode A8).

J) Repeat steps G and H. The bits that are exclusive-ORed with ones are inverted.

SUMMARY

In this experiment the μLab read various values from the input switches, modified them by several different logical instructions, and then displayed the results on the output LEDs. This allowed the operation of the logical instructions to be observed. When the AND function is used, selected bits can be forced to zero. With the OR function, selected bits can be forced to one. Using XOR, selected bits can be inverted.

MASKING

A common use for logical instructions is to select certain bits of a word. This is called *masking*. For example, suppose that eight switches are connected to an input port. To test only a single bit (a single switch), the processor must disregard the other bits. Figure 12-2 shows the flowchart for a program that tests the switch connected to bit 3 of the input port. If the switch is off, the output LEDs are turned on; otherwise, they are turned off.

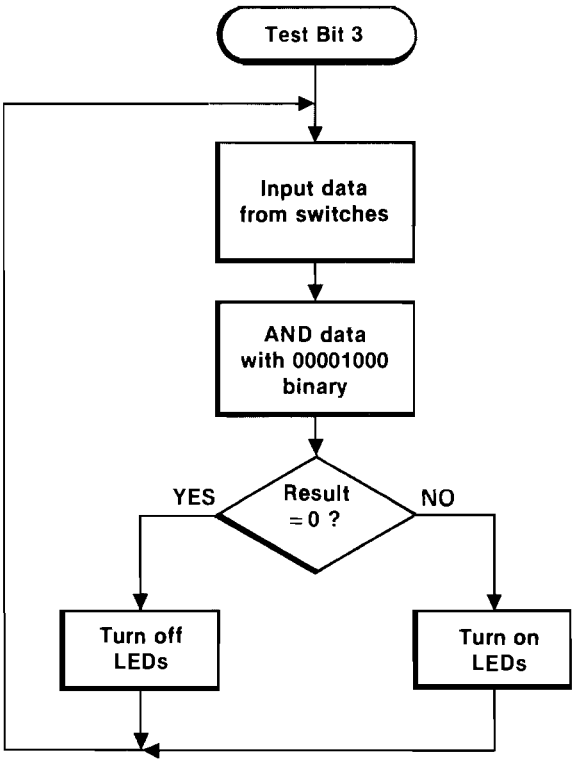


Figure 12-2. Flowchart for Program to Test Bit 3 of Input Port

Table 12-2 shows the program listing. The program first reads the data from the input port into the accumulator. It then sets the B register to the *mask* value and ANDs the B register with the accumulator. The result is that all bits except bit 3 are forced to zero. (Since 0 AND 1 = 0 and 1 AND 1 = 1, any bit ANDed with a 1 is unchanged.) Finally, the program uses the JZ instruction to jump if the zero flag is set. The zero flag indicates that the entire byte (and therefore bit 3) is zero.

PROGRAMMING EXERCISE 12-1: MASKING

Complete the program in Table 12-1 by adding the ON and OFF routines. Use the ON routine to turn on the output port LEDs and the OFF routine to turn them off.

Convert the program to machine code and key it into the μLab. Locate the program anywhere between 0800 and 0B00.

Check your program carefully and verify that it is correctly stored. Run the program and verify that the input switches properly control the output LEDs. All the LEDs should go on only if the switch on bit 3 is up. The other switches should have no effect.

Now modify the program to test a different bit of the port (a different switch). If it does not work correctly, recheck the logic of the program and the conversion to machine code. Next, verify that the program is correctly stored. As a last resort turn to Appendix A which lists the complete program.

Address	Contents	Label	Instruction	Comments
0800	_____	START:	LDA 2000	;Read input port to accumulator
0801	_____			
0802	_____			
0803	_____		MVI B,08	;Set B to mask value (0000
0804	_____			1000 binary)
0805	_____		ANA B	;Set all bits except no. 3 to zero
0806	_____		JZ OFF	;Test for accumulator = 0
0807	_____			
0808	_____			
0809	_____	ON:	_____	;Turn on LEDs
080A	_____			
080B	_____		_____	
080C	_____			
080D	_____			
080E	_____		JMP START	
080F	_____			
0810	_____			
0811	_____	OFF:	_____	;Turn off LEDs
0812	_____			
0813	_____		_____	
0814	_____			
0815	_____			
0816	_____		JMP START	
0817	_____			
0818	_____			

Table 12-2. Program to Test Bit 3

CLEARING THE ACCUMULATOR

The XRA A instruction exclusive-ORs the accumulator with itself. Since the exclusive-OR of anything with itself is zero, this instruction clears the accumulator using only one byte of memory. The alternative, MVI A,0, requires two bytes of memory.

SHIFTING

Another commonly required function is the shifting of data to the right or the left. This is performed in hardware using a shift register. The 8085 has instructions that shift the data in the accumulator. *Rotate Right Circular* (RRC) performs the right shift and *Rotate Left Circular* (RLC) performs the left shift. These instructions operate only on the accumlator. "Circular" refers to the fact that the LSB is shifted to the MSB (or vice versa) as shown in Figure 12-3.

For example, suppose that the accumulator contains the value 0010 0001. After an RRC is executed, it will contain 1001 0000. Note that the LSB is moved to the MSB. This happens because the data is rotated as if the bits were arranged in a circle, with the MSB and LSB adjacent to each other.

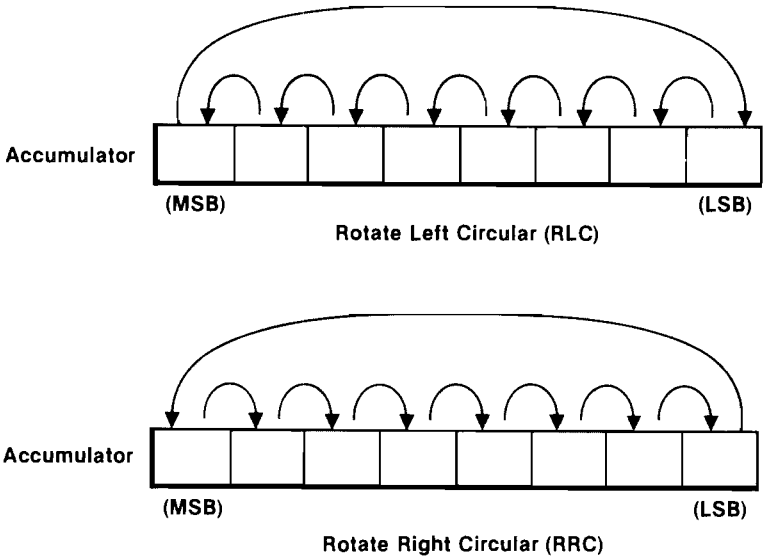


Figure 12-3. Rotate Instruction Function

PROGRAMMING EXERCISE 12-2: ROTATES

Write a program following the flowchart in Figure 12-4 to demonstrate the operation of the rotate instructions. Convert the program to machine code and key it into the µLab. Step through the program and verify that the output port LEDs show the operation of the rotate instruction.

Now add a breakpoint to the program so that you can use RUN instead of INSTR STEP. Each time RUN is pressed, the data should shift.

If you can not get your program to work, turn to Appendix A for the solution.

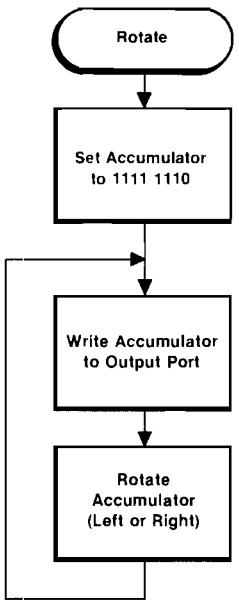


Figure 12-4. Flowchart for Programming Exercise 12-2

ADDITION

The most basic arithmetic function is the addition of two numbers. The ADD r instruction adds the contents of the specified register to the contents of the accumulator and stores the result in the accumulator.

For example, ADD D adds the contents of the D register to the contents of the accumulator. Suppose that the D register contains 1001 0011 and the accumulator contains 1010 1010. The result is:

	Decimal		Binary	
	147		1001 0011	(D register)
+	170	+	1010 1010	(Accumulator)
	317		1 0011 1101	(Accumulator)

THE CARRY FLAG

Note that the result is greater than 255 decimal, so it is more than eight bits long. To accomodate this overflow, the processor contains a *carry flag*. This flag acts as the ninth bit of the accumulator and can be tested by the *Jump If Carry* (JC) and *Jump If No Carry* (JNC) instructions. This test is similar to the test of the zero flag.

For example, the following instructions will add the D register to the accumulator and jump to location 0820 if a carry is generated:

```
AND  D
JC   0820
```

If no carry is generated (result < 256 decimal), then the program will continue with the next instruction.

The carry flag is also used when adding numbers longer than eight bits. Two (or more) registers are used to represent each number. The least significant bytes are added first. Then the most significant bytes are added. The carry from the least-significant byte (if any) is added to the most significant byte. The carry bit thus functions as a link between the two bytes.

You can examine the flags by pressing FETCH REG and then STORE/INCR. The flags are all combined into one byte, as shown in Figure 12-5. Only the carry and zero flags are used in this course. The sign flag is just a copy of the MSB and is used for two's complement arithmetic (described in Lesson 15). Not all instructions affect all the flags. MOV instructions, for example, do not affect any flags. The instruction descriptions in Appendix B indicate which flags are affected by each instruction.

SUBTRACTION

The *SUB r* instruction subtracts the contents of the specified register from the accumulator. For example, SUB B subtracts the contents of the B register from the accumulator.

The subtract instruction uses the carry flag as a borrow flag. If the carry flag is set after a SUB B instruction, it indicates that the value in the B register is greater than the value in the accumulator.

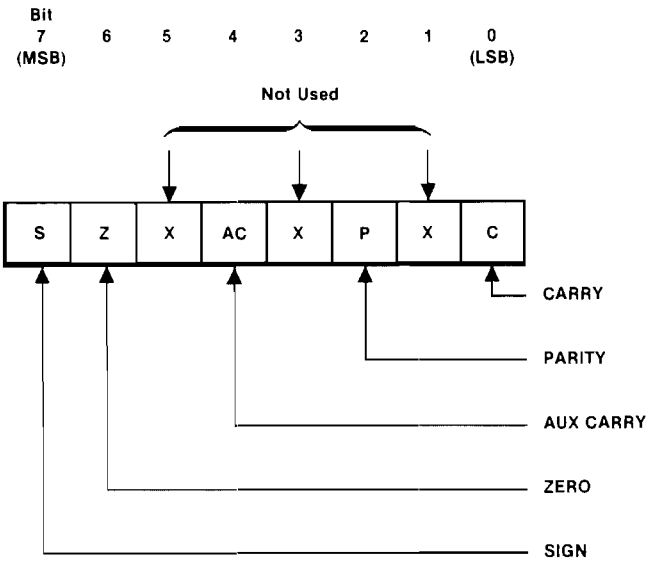


Figure 12-5. Microprocessor Flags

EXPERIMENT 12-2

Arithmetic Functions

CONCEPT

The INSTR STEP key is used to execute a program which demonstrates the functions of the arithmetic instructions. The FETCH REG key is used to set the registers before an instruction is executed and examine them afterwards.

PROCEDURE

A) Fill in the machine code for the program in Table 12-3. This program subtracts the B register from the accumulator, and then adds the C register to the accumulator.

Address	Contents	Instruction	Comments
0800	_____	SUB B	;Subtract B from accumulator
0801	_____	ADD C	;Add C to accumulator

Table 12-3. Arithmetic Instruction Demonstration Program

B) Key the program into the μLab.

C) Press [FETCH REG] to display the accumulator.

D) Key in the data A7 and press [STORE INCR]. This stores A7 in the accumulator. (Actually, the data is stored in a RAM location which the monitor uses to save the value of the accumulator. When RUN, INSTR STEP, or HDWR STEP is pressed, the accumulator is loaded from memory.)

E) Press [DECR] to verify that the accumulator contains the data A7.

F) Press [STORE INCR] twice to display the B register.

G) Key in the data 23. Press [STORE INCR] to store this data in the B register.

H) The C register is now displayed. Key in the data 93. Press [STORE INCR] to set the C register. The registers now contain the following data:

A = A7 B = 23 C = 93

I) All the inputs to the program are now set. Press [FETCH ADRS] [0] [8] [0] [0] [INSTR STEP] to execute the SUB B instruction.

J) Press [FETCH REG] to view the accumulator. Does it contain the expected value? Remember that these numbers are hexadecimal (A7 hex - 23 hex = 84 hex).

K) Press [STORE INCR] to view the flag register. Convert the value to binary and check the state of the carry and zero flags by referring to Figure 12-5. This subtraction generated no borrow (so the carry flag was not set), and the result was not zero.

L) Using [STORE INCR] to step through the registers, verify that B and C have not changed.

M) Press [FETCH PC] [INSTR STEP] to execute the ADD C instruction.

O) Press [FETCH REG]. The result of the addition is 84 hex + 93 hex = 117 hex. Since the result is greater than FF hex, it will not fit into the eight-bit accumulator. The least significant eight bits (17 hex) remain in the accumulator, and a carry is generated.

P) Press [STORE INCR] to view the flag register. Check the state of the carry flag. The flag is set, indicating that the result was greater than FF hex (255 decimal).

Q) Verify that the B and C registers did not change.

SUMMARY

This experiment demonstrated the operation of the addition and subtraction instructions. After each operation, the accumulator contained the result. The other register contained its original value. The carry flag indicated whether an overflow (or borrow) occurred.

You used the FETCH REG key to set the inputs to the program. You then examined the program's outputs. To provide a visual demonstration of the instructions.

SUBROUTINES AND THE STACK

Large programs can be simplified by using subroutines to perform repetative tasks. For example, suppose a program must do a series of calculations involving several multiplications. One general-purpose multiplication subroutine can be written, which is then used any time a multiplication must be performed. Subroutines are also useful for dividing a program into small modules, as described in Lesson 13.

The CALL instruction is used to access a subroutine. When a CALL instruction is executed, the microprocessor saves the contents of the program counter (the return address) in the *stack*, a specially-accessed part of memory. The microprocessor goes back to this return address when a RET instruction is executed at the end of the subroutine. The program then resumes execution at the instruction immediately following the CALL.

Suppose that a sixteen-bit register were used for storing the return address. Can you see the problem this causes? If the subroutine calls another subroutine (referred to as *nested* subroutines), then the first return address is lost.

The solution to this problem is to use a group of memory locations (called the stack). As one routine calls another, the return addresses are stored in sequential memory locations. Figure 12-6 shows the sequence of operations as routine A calls routine B which calls routine C. The contents of the stack and the PC at each step are shown in Figure 12-7. The sequence of events is as follows:

a. The main program is being executed.

b. The main program calls subroutine A. The return address is stored in the stack, and the PC is loaded with the starting address of routine A.

c. Routine A calls routine B. The return address for routine A is placed on the stack, and the PC is loaded with the starting address of routine B.

d. Routine B returns. The last entry in the stack is loaded into the PC, and control has returned to routine A.

e. Routine A returns control to the main program.

You can think of the stack as a pile of plates; any number of plates can be piled on, and they are removed in the reverse order. This is called a *push-down*, or *Last In First Out* (LIFO), stack. Since the 8085 uses the RAM for the stack, it can be arbitrarily large. A special register in the processor (the *stack pointer*) contains the address of the top of the stack. The location of the stack is established by setting the stack pointer to the desired address.

The μLab's power-up program sets the stack pointer to 0BB0. This allows you to use the stack without setting the stack pointer. It is possible for your program to set the stack pointer outside of the available RAM area. This will happen, for example, if a series of subroutine calls are executed without any returns, which will cause the program to halt. However, the next time you press RUN or either STEP key, the monitor checks the stack pointer. If it is at an undesirable value (<0B40 or >0BFF), it is set back to 0BB0. Normally the stack pointer does not go beyond the allowable range, and the monitor does not change it.

You can examine the stack pointer using the μLab's register mode. Since it is a sixteen-bit register, it is displayed in two parts: the high-order byte (SPH) and the low-order byte (SPL).

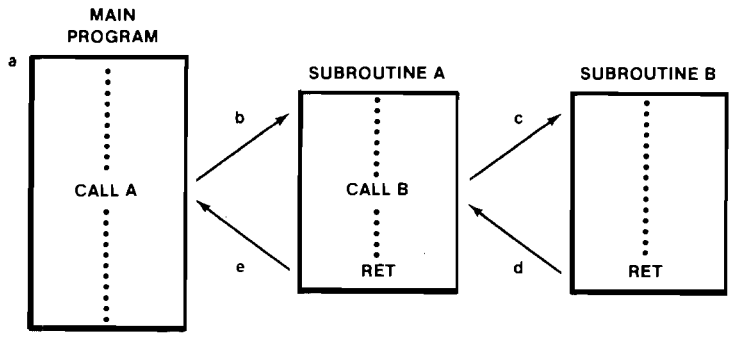


Figure 12-6. Sequence of Events as Main Program Calls Routine A Which Calls Routine B

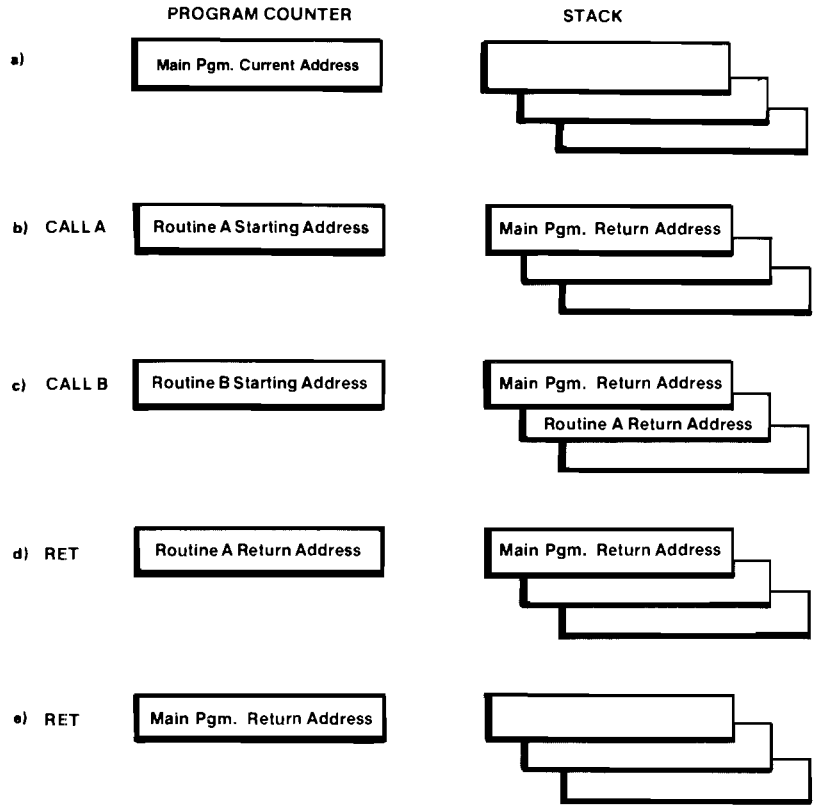


Figure 12-7. Operation of the Stack

PUSH AND POP INSTRUCTIONS

The CALL and RET instructions manipulate the stack pointer automatically. It is also possible to use the stack manually. The PUSH instruction stores a pair of registers in the stack, and the POP instruction reads the data back into the registers. Since the stack is designed for storing addresses, which are sixteen bits long, registers are stored two at time. PUSH B, for example, pushes the B and then the C register onto the stack. POP B restores both of these registers from the top of the stack.

These instructions are commonly used for temporarily saving the contents of registers. For example, suppose that a program uses the B and C registers to store some data. It then calls a subroutine which also must use these registers. The subroutine can save the original contents of these registers (as set by the calling program) by using PUSH B. The B and C registers can now be used by the subroutine. Then, near the end of the subroutine, a POP B instruction is used to restore the registers to their initial values. Alternatively, the saving and restoring can be done by the main program just before and after the CALL instruction.

Lesson 12

The logical instructions allow standard logic gate functions to be performed by software. The instructions operate on eight bits in parallel, providing the equivalent function of eight two-input logic gates. They are commonly used for selecting or modifying certain bits of a word.

The arithmetic instructions provide the basic addition and subtraction capabilities. With both the logical and arithmetic instructions, the result is always left in the accumulator. The carry flag is used by the arithmetic instructions to indicate an overflow or a borrow.

The rotate instructions are useful for manipulating the bits in a word. The data in the accumulator may be shifted left or right.

The stack is used for storing subroutine return addresses and allows the routines to be nested. It can also be used for storing and recovering the contents of the registers via the PUSH and POP instructions.

Lesson 12

1. Suppose the B register contains 37 hex, and the accumulator contains 14 hex. After executing the instruction ANA B, the accumulator will contain:

   a. 37 (hex).
   b. 14.
   c. 4B.
   d. 23.

2. In question 1, if the instruction were ORA B, the result would be:

   a. 37.
   b. 14.
   c. 4B.
   d. 23.

3. To leave the MSB unchanged and set all other bits to zero, the accumulator should be ANDed with:

   a. 01.
   b. 10.
   c. 80.
   d. F1.

4. If the accumulator initially contains F3 hex, the following program will jump to address _____ .

```
MVI  B,23
ADD  B
JC   0930
JMP  0900
```

5. The stack is used for storing subroutine return addresses because:

   a. the regular memory is too small.
   b. the registers cannot hold an address.
   c. it allows subroutine nesting.
   d. all of the above are true.

LESSON 13

Software Design Techniques

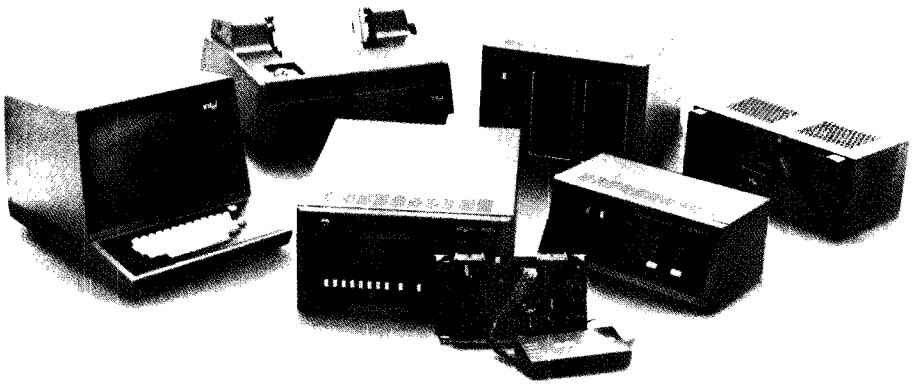
INTRODUCTION

Designing a complex piece of software is just as involved as designing a complex piece of hardware, but the problems are somewhat different. A new set of techniques is necessary for designing software that is easy to debug and modify. This lesson describes the software development process. As an example, a traffic light controller program is presented.

This lesson is concerned only with the design of software systems. If this topic is not of interest to you, this lesson may be skipped without loss of continuity.

DEVELOPMENT SYSTEMS

When writing programs using the μLab, the assembly language must be translated "by hand" to machine language. The machine code is then keyed into the μLab. While this is an adequate system for writing short programs, it is too cumbersome for a major software project.



Intel's Intellec MDS® Development System. *Clockwise from the left, the components are: CRT terminal, printer, dual floppy disk drives, paper tape reader PROM programmer, and microcomputer with ICE card. (Photo Courtesy Intel Corp.)*

Most microprocessor software is written using a *development system,* a microcomputer system especially designed for software and hardware development. A typical system includes 32-64K bytes of RAM, dual floppy disk drives for program storage, a CRT terminal with a keyboard, and a printer. The software usually includes an editor for entering and editing assembly language programs, an assembler for translating assembly to machine language, and a monitor, or debugger, for setting breakpoints, examining registers, and so forth. Such a system greatly eases the development process.

As an aid to hardware debugging, many development systems include an *In-Circuit Emulator* (ICE). The microprocessor IC is removed from the system under test, and a cable from the development system (with a 40-pin plug on the end) is plugged in its place. The development system can now completely control and monitor the operation of the system under test. ICE is one of the most sophisticated design and debugging tools available.

A SOFTWARE DESIGN PROCEDURE

When designing a complex program, you must follow an orderly sequence. The following is a suggested procedure:

- Define the problem.
- Design the solution—partition into functional blocks.
- Flowchart the programs.
- Write the programs.
- Test and debug each routine.
- Test and debug the entire program.

The first step is the most obvious, yet it is often overlooked. Before you can begin to design a solution, you must understand the problem. A set of specifications for the program should be created: what are its inputs, how must it process them, and what are its outputs?

Once these specifications have been determined, you can design the software. Before you start to write programs or even to draw flowcharts, think about the overall design of the system. It is best to separate the software into relatively small, modular sections. Make sure that each section has well-defined inputs and outputs. This approach allows you to attack the problem in manageable portions. It also allows you to test the program in small pieces, making it much easier to find errors in the program. The same approach is taken in hardware design by breaking up the system into functional blocks.

There are many ways to structure any given programming task. One approach, called *structured programming*, provides an excellent technique for program design. While it is not completely described in this course, its general philosophy is followed.

Once the problem is defined, an approach chosen, and the program divided up into modules, you can flowchart your logic. There is generally an overall flowchart that gives the "big picture" and shows each program module as a single box. There is also a flowchart for each module.

Now the actual program can be written. If the preceeding steps have been done carefully and thoroughly, this task is fairly straightforward. There is another parallel to hardware design: once the right approach to a problem has been determined, the actual implementation is usually relatively easy.

After the program modules are written, they must be tested. Even an experienced programmer does not expect all of the programs to work correctly the first time. The modularity of the program is very important at this stage. Each module can be tested separately or together with routines that have already been tested. The module is supplied with the required inputs, and its outputs are then examined to see if they are correct. It is much easier to debug an individual module than the entire software system. When all the modules work, then it is time to tie them all together and test the whole system.

A SOFTWARE DESIGN EXAMPLE

This section describes a traffic light controller program to illustrate the software design process. First the basic controller is developed, and then a number of improvements are suggested. The output port LEDs are used for the traffic lights.

Consider a simple two-street intersection. Each signal consists of a red, yellow, and green light. The controller must sequence the lights as follows:

1. Light A = red
2. Light B = green
3. Wait green time
4. Change light B to yellow
5. Wait yellow time
6. Change light B to red
7. Change light A to green
8. Wait green time
9. Change light A to yellow
10. Wait yellow time
11. Repeat entire process (go to step 1)

You could program this sequence in a straightforward manner, but that is not the best solution in terms of the program length. Notice that the program contains two repetitive segments: one to sequence signal A from green to yellow to red, and one to do the same for signal B. It is more efficient if one program is used to sequence both signals.

Figure 13-1 shows flowcharts to implement the controller in this way. The main program uses the sequencing routine to control each signal in turn. The sequencing routine controls both signals. The main program tells the sequencing routine which signal it wants changed by defining a register to be used as the "signal flag." The main program sets this register before calling the sequencing routine, which then tests the register. If the register is zero, signal A is sequenced. If the register is not zero, signal B is sequenced. This transfer of information between a calling program and a subroutine is called *parameter passing.*

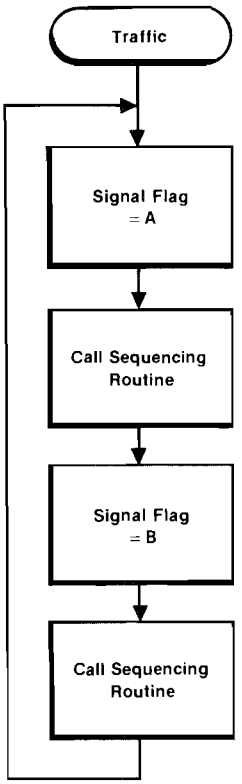


Figure 13-1a. Traffic Light Controller Main Program

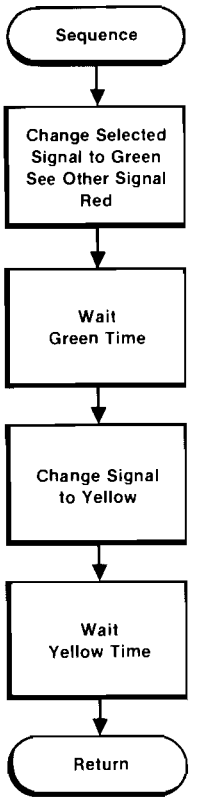


Figure 13-1b. Sequencing Routine

The basic architecture (structural design) of the traffic light controller is now defined. The main program is quite simple, since all the details are handled by the sequencing routine. A well-structured program will usually have a very simple main program.

THE SEQUENCING ROUTINE

If the signal flag is *not* set, the sequencing routine first sets signal A green and signal B red. Then signal A is changed to yellow, and signal B remains red. On the other hand, if the signal flag *is* set, then signal A is held red while signal B is cycled through green and yellow. Note that it is not necessary to set the signal red at the end of the sequence. This is done when the other signal is set to green at the beginning of the next cycle.

THE CHANGE ROUTINE

Since the sequence is the same regardless of the signal involved, the sequencing routine does not take the signal flag into consideration. The output routine (to be described shortly) checks this flag.

The signal lights (the output port LEDs) are all connected to a single output port, as shown in Figure 13-2. Two unused bits control extra LEDs, which are always off. Table 13-1 shows the bit patterns for the various signal settings. Notice that the difference between the first two patterns (signal A active) and the second two patterns (signal B active) is that the left-hand four bits are switched with the right-hand four bits. This is not surprising, since the same operation is performed on each signal, each of which uses half of the output port's eight bits.

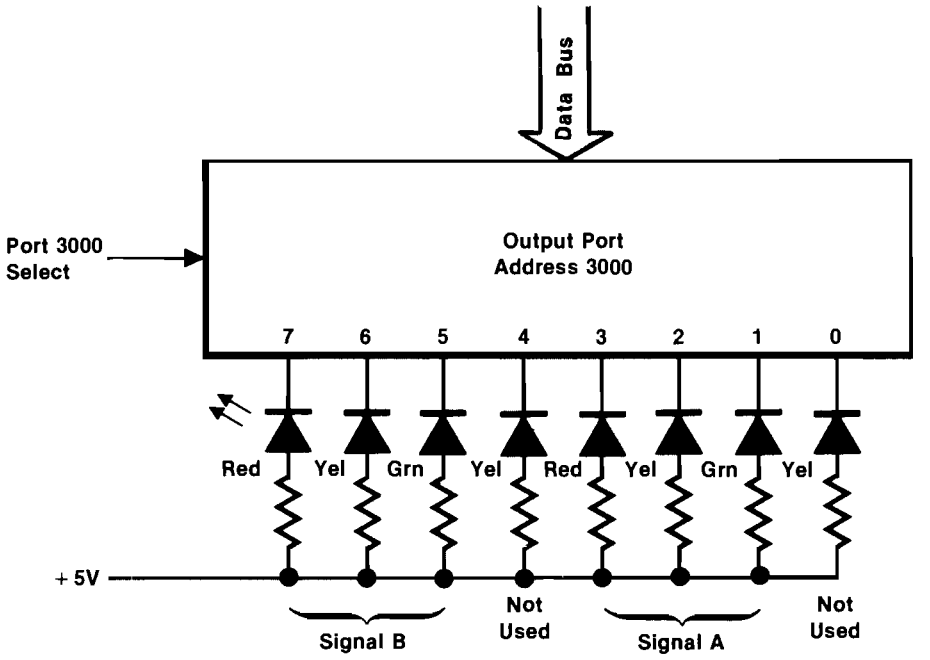


Figure 13-2. LED Connections for Traffic Light

	B				A				
	RED	YEL	GRN		RED	YEL	GRN		Hex
A Green, B Red	0	1	1	1	1	1	0	1	7D
A Yellow, B Red	0	1	1	1	1	0	1	1	7B
B Green, A Red	1	1	0	1	0	1	1	1	D7
B Yellow, A Red	1	0	1	1	0	1	1	1	B7

Table 13-1. Traffic Light Bit Patterns

With this background, the logic of the sequencing and change routines can be explained. The sequencing routine always sequences signal A (i.e., it always sequences the right-hand four bits). It calls the change routine to do the actual output operation. Figure 13-3 shows the flowchart for the change routine. If the signal flag is set (indicating that signal B should be sequenced), then the change routine simply switches the left and right halves of the data before it is output. This is accomplished by rotating the data four bit positions.

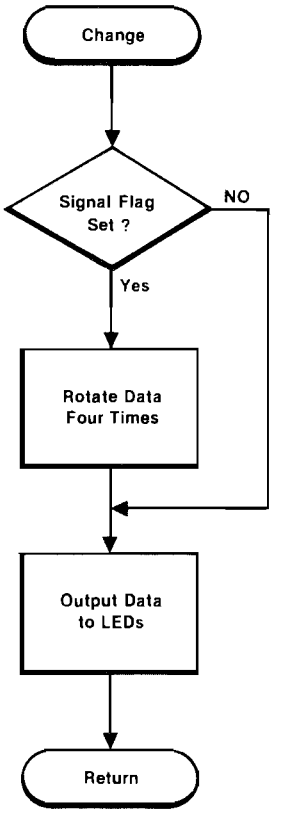


Figure 13-3. Change Signal Routine

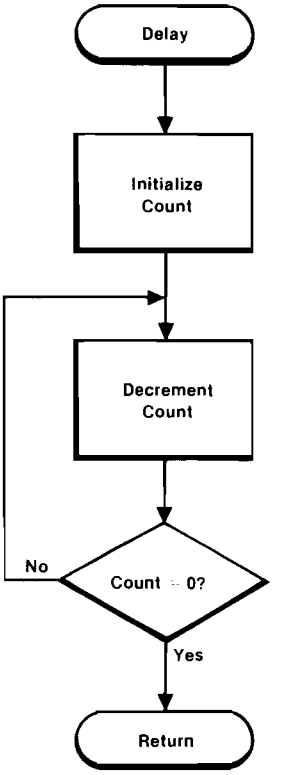


Figure 13-4. Basic Delay Routine

THE DELAY ROUTINE

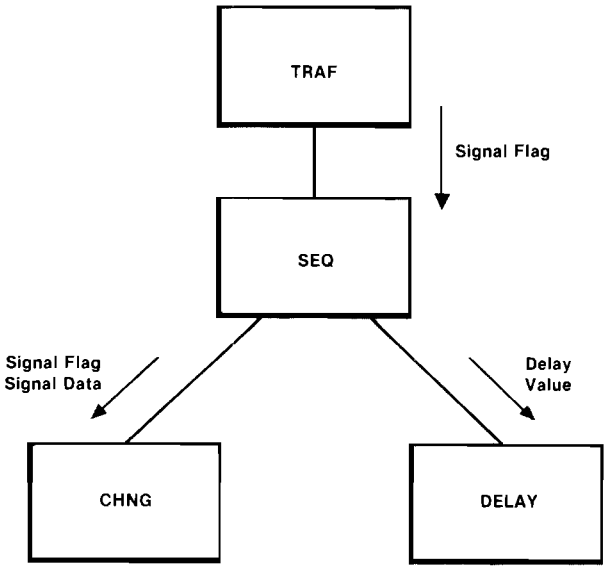
The system is now designed except for the delay routine. A routine is needed that causes a delay varying from a few seconds (yellow time) to as long as a minute or two (green time).

One way of generating time delays is shown in Figure 13-4. A register is initialized to some value and then decremented until it reaches zero. It takes approximately 7 μs to decrement and test the register and jump back to the beginning. Therefore, delays of from 7 μs to $7 \times 255 = 1{,}785$ μs (1.785 ms) can be set by initializing the counter to values from 1 to 255. To obtain a variable delay, the delay subroutine is written so that it does not initialize the count, but expects this information from the calling program. This is another instance of parameter passing.

The delay program for the traffic light controller works on this principle. However, there is a problem with the routine in Figure 13-4: it has a maximum delay of less than two milliseconds, and the traffic light needs tens of seconds. The solution is a straightforward extension of this technique. An additional register is used to extend the count value up to 65,536. The details of this process are presented later in this lesson.

STRUCTURE CHARTS

A *structure chart* is useful for showing how the program is partitioned. It shows which routines are called by which programs. Figure 13-5 shows a structure chart for the traffic light controller. The main routine, called TRAF, is represented by a box at the top of the chart. TRAF calls the subroutine SEQ, which in turn calls both CHNG and DLY. Note that this is not a flowchart and does not provide information about the actual program flow. Instead, it provides general information about the logical structure of the program. The parameters that are passed from one routine to another are also shown.



Computer Museum

Figure 13-5. Traffic Light Controller Structure Chart

THE CONTROLLER PROGRAMS

The specifications and flowcharts for each of the required programs are now complete. To write the programs, you must choose specific registers for each function. The E register is used for the signal flag (the choice is arbitrary). The main program is quite simple, as shown in Table 13-2.

```
TRAF: MVI  E,0      ;Sequence signal A
      CALL SEQ
      MVI  E,1      ;Sequence signal B
      CALL SEQ
      JMP  TRAF
```

Table 13-2. Traffic Light Main Program

This program closely follows the flowchart presented previously. The sequencing routine (SEQ) is shown in Table 13-3.

```
SEQ:  MVI   H,7D      ;Set signal green
      CALL  CHNG
      MVI   D,6       ;Wait green time
      CALL  DELAY
      MVI   H,7B      ;Set signal yellow
      CALL  CHNG
      MVI   D,2       ;Wait yellow time
      CALL  DELAY
      RET
```

Table 13-3. Traffic Light Sequencing Routine

The sequencing routine first sets the H register to the pattern for signal A red and signal B green, as shown in Table 13-1. Since the LEDs indicate negative logic, the pattern uses ones for off and zeros for on. It then calls the change routine to send the data to the signal lights. If the signal flag (register E) is set, the change routine switches the two halves of the data. This allows the routine to sequence either signal. The H register is arbitrarily chosen to pass the signal data between the two routines.

Now the delay must be generated. This is done by a delay routine (to be discussed shortly), which expects the delay variable to be in register D. Therefore, the D register is initialized and the delay routine is called. The sequence then repeats for the yellow light, with a different data value (to cause the yellow LED to be on) and a different delay value.

The traffic light change routine (Table 13-4) tests the signal flag in register E. The contents of register E are first moved to the accumulator. Then the contents of the accumulator are compared with the value zero. If the contents of the accumulator (the signal flag) are zero, the microprocessor's zero flag will be set. This flag is tested by the JZ instruction. Note that the MOV A,H instruction between the compare and the test operations does not interfere with the test because it does not affect the flags. The JZ instruction tests the zero flag, not the contents of the accumulator. If the signal flag is not set, the program jumps to the "output" step. If the signal flag is set, then the data is rotated four times to switch the left and right halves. Each rotate instruction shifts the data one bit position.

```
CHNG: MOV  A,E      ;Is signal flag set?
      CPI  0        ;Compare signal flag to zero
      MOV  A,H      ;Move signal data to accumulator
      JZ   OUTP     ;Not set—output data directly
      RLC           ;Set—Rotate data
      RLC
      RLC
      RLC
OUTP: STA  3000     ;Output data to signal
      RET
```

Table 13-4. Traffic Light Change Routine

DESIGNING THE DELAY ROUTINE

The delay routine for the traffic light controller is complicated slightly by the length of time delay required. To explain its design, some basic delay routines are explained first. Table 13-5 shows a simple, commonly used delay routine, which follows the flowchart in Figure 13-4.

```
DELAY: DCR A          ;Decrement counter    4 states
       JNZ  DELAY     ;Loop unless zero     7/10 states
       RET                                  10 states
```

Table 13-5. Simple Delay Routine

To use this routine, the accumulator is set to the desired delay value and the delay routine is called. The accumulator is decremented until it reaches zero, and then control returns to the calling program.

The delay time generated by this program can be analyzed as follows. Appendix B shows the number of clock periods (states) required to execute each instruction. Four states are required for the DCR A, and seven or ten states for the JNZ. The jump requires ten states if the condition is satisfied and the jump is performed, and seven states if it is not. The RET requires ten states.

The routine therefore requires 4 + 10 = 14 states for each pass through the loop, except for the last time through. When the accumulator reaches zero, the JNZ does not cause a jump and therefore requires only seven states. The RET instruction then uses ten states. This makes the total for the last pass 4 + 7 + 10 = 21. The equation for the total delay is therefore:

$$\text{DELAY} = (A-1)14 + 21 \text{ states}$$

Since the μLab uses a 2 MHz clock, a state is 0.5 μs long. The delay equation can therefore be rewritten as:

$$\text{DELAY} = (A-1)7\ \mu s + 10.5\ \mu s$$

"A" in this equation denotes the initial value in the accumulator. If it is 1, for example, then the delay is (1-1)7 + 10.5 = 10.5 μs. Notice that the first term goes to zero because the jump is never executed. The first time through the loop, the accumulator is decremented to zero, and the conditional jump is never performed.

The longest delay is caused by setting the accumulator to zero. The first time through the loop, it is decremented and set to 255. Since the accumulator is not tested for zero until after it is decremented, it gives the equivalent of A = 256 in the delay time equation. The maximum delay is then (256-1)7 + 10.5 = 1,795 μs. This is less than two thousandths of a second and is not very useful as a delay interval for the traffic light controller.

USING REGISTER PAIRS

The 8085 has a group of instructions that operate on *pairs* of registers instead of individual registers. While these instructions are not strictly necessary, they are often very convenient. The long delay routine in Table 13-6 provides a good example of the use of one of these instructions.

```
DELAY: DCX  B          ;Decrement counter
       MOV  A,B        ;Test for zero
       ORA  C
       JNZ  DELAY      ;Repeat until zero
       RET
```

Table 13-6. Delay Routine Using Register Pair

The DCX instruction decrements a register pair, treating it as a single sixteen-bit number. This allows the delay count to have values up to 2^{16}=65,536. DCX B specifies that the B and C registers should be decremented as a pair. In a register pair instruction, "B" refers to the B and C pair. If there is a borrow from the C register, it is automatically subtracted from the B register.

Unfortunately, the DCX instruction does not affect the zero flag. To set the zero flag, one byte of the count is moved to the accumulator, and then it is ORed with the other byte. The result is zero only if both bytes are zero. Now the zero flag is set, and the JNZ instruction can be used to test the count for zero.

The timing analysis for this routine is very similar to the one previously presented. The DCX B takes six states, the MOV A,B and ORA C each take four, and the other instructions are the same as in the previous example. The delay equation is therefore:

$$\begin{aligned} \text{DELAY} &= (N-1)24 + 31 \text{ states} \\ &= (N-1)12\ \mu s + 15.5\ \mu s \end{aligned}$$

"N" is the binary value in the B and C registers. The maximum delay is (65,536-1)12 μs + 15.5 μs = 0.786 seconds. This delay is closer to the values needed for the traffic light, but it is still too short. However, this routine can be used to generate a longer delay by combining it with a single register count loop. Figure 13-6 shows the flowchart. To generate the longer delay time, the delay routine in Table 13-5 is used with the delay value always set to zero (for the maximum 0.786 second delay). This routine is then placed inside another count loop. The delay value (passed to the delay routine in the D register) determines how many times this 0.786 second delay is executed. This can be done up to 256 times, so the maximum delay is 256 × 0.786 seconds = 201 seconds, an adequate value for the traffic light controller.

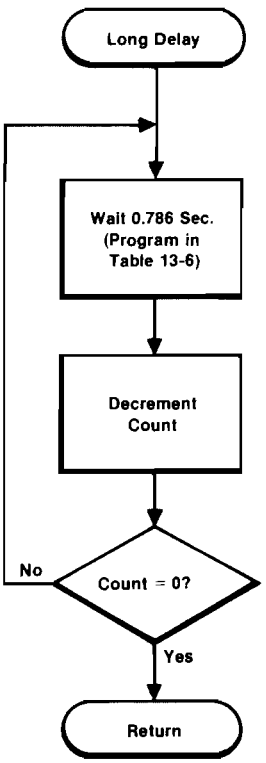


Figure 13-6. Long Delay Routine

Table 13-7 shows the listing. The 0.786 second delay is implemented with the same program used previously. The B and C registers are set to zero for maximum delay. This is accomplished with the *Load Register Pair Immediate* (LXI) instruction. It is similar to the MVI instruction, except that two registers are loaded at once. Two bytes of data follow the LXI opcode. The first data byte goes to the C register, and the second byte to the B register.

The D register is used for the main delay count. Note that this program has two loops. An "inner" loop generates the 0.786 second delay and an "outer" loop generates multiples of this delay. Arranging loops in this way is called *nesting.*

Delay routines are commonly used in a wide variety of microprocessor applications. They are used for programmed oscillators and pulse generators, keyboard debouncing (discussed in the following lesson), and many other applications.

```
DELAY: LXI  B,0       ;Initialize inner loop count
 LOOP: DCX  B         ;0.786 sec delay loop
       MOV  A,B
       ORA  C
       JNZ  LOOP
       DCR  D         ;Decrement main count
       JNZ  DELAY     ; and loop unless zero
       RET
```

Table 13-7. Traffic Light Delay Routine

TESTING TECHNIQUES

By testing the routines one at a time, you can isolate problems more easily. The general procedure is to use the keyboard to set the registers (from which the routine expects to get its inputs) to given test values, and then run the routine. Breakpoints are often used to stop the program at the desired spot. Alternatively, the single-step mode can be used. The routine's outputs are then examined to see if they are correct. This may involve examining registers or memory locations or observing the state of an output port.

The order in which the routines are tested is important. If you test a routine that calls an untested routine, then you may not know which routine is causing the problems. Therefore, the routines you should test first are those that do not call any other routines. The structure chart (Figure 13-5) conveniently identifies these routines (they are at the bottom of the chart). Next, you should test routines that call only previously tested routines. These appear on the next level of the structure chart.

The following experiments demonstrate the testing and use of the traffic light controller program. The entire program is listed, with the machine code added, in Table 13-8. NOPs are added to the main program so that new features can be readily added later. It is a good practice to do so in all your programs. To practice your coding skills, cover up the machine code and code the program yourself. Then check to see that the code you generated was the same as that shown in Table 13-8.

MAIN PROGRAM

Address	Contents	Label	Instruction		Comments
0810	1E	TRAF:	MVI	E,0	;Sequence signal A
0811	00				
0812	CD		CALL	SEQ	
0813	30				
0814	08				
0815	00		NOP		
0816	00		NOP		
0817	00		NOP		
0818	00		NOP		
0819	1E		MVI	E,1	;Sequence signal B
081A	01				
081B	CD		CALL	SEQ	
081C	30				
081D	08				
081E	C3		JMP	TRAF	
081F	10				
0820	08				

SEQUENCING ROUTINE

Address	Contents	Label	Instruction		Comments
0830	26	SEQ:	MVI	H,7D	;Set signal green
0831	7D				
0832	CD		CALL	CHNG	
0833	55				
0834	08				
0835	16		MVI	D,6	;Wait green time
0836	06				
0837	CD		CALL	DELAY	
0838	70				
0839	08				
083A	26		MVI	H,7B	;Set signal yellow
083B	7B				
083C	CD		CALL	CHNG	
083D	55				
083E	08				
083F	16		MVI	D,2	;Wait yellow time
0840	02				
0841	CD		CALL	DELAY	
0842	70				
0843	08				
0844	C9		RET		

Table 13-8. Traffic Light Controller Program

CHANGE ROUTINE

Address	Contents	Label	Instruction		Comments
0855	7B	CHNG:	MOV	A,E	;Signal flag set?
0856	FE		CPI	0	
0857	00				
0858	7C		MOV	A,H	
0859	CA		JZ	OUTP	;Flag not set—output
085A	60				data directly
085B	08				
085C	07		RLC		;Flag set—rotate
085D	07		RLC		data
085E	07		RLC		
085F	07		RLC		
0860	32	OUTP:	STA	3000	;Output data to
0861	00				signal
0862	30				
0863	C9		RET		

DELAY ROUTINE

Address	Contents	Label	Instruction		Comments
0870	01	DELAY:	LXI	B,0	;Initialize inner loop
0871	00				delay
0872	00				
0873	0B	LOOP:	DCX	B	;Inner loop 0.786
0874	78		MOV	A,B	sec delay
0875	B1		ORA	C	
0876	C2		JNZ	LOOP	
0877	73				
0878	08				
0879	15		DCR	D	;Main loop
087A	C2		JNZ	DELAY	
087B	70				
087C	08				
087D	C9		RET		

Table 13-8. Traffic Light Controller Program (Continued)

EXPERIMENT 13-1

Testing the Traffic Light Controller Program

CONCEPT

In this experiment you enter and test the traffic light controller routines. One routine is tested at a time. When they all work, the entire program is then tested. The register mode and breakpoint features of the μLab are used extensively.

PROCEDURE

A) Key in the four routines in Table 13-8. Note that there is space left between the routines. This leaves room for them to be modified.

B) Verify that they are correctly stored.

C) Test the delay routine first. When you test the routine, you want it to stop when it is finished. It should not try to execute the return instruction, since it was not called by another routine. Therefore, replace the RET instruction at the end of the delay routine (address 087D) with a breakpoint (RST 1, opcode CF).

D) Set the D register to 5. (Press [FETCH REG] and then [STORE INCR] four times to access the D register. Then press [5] [STORE INCR].) This sets the delay value to five.

E) Run the delay routine (starting at address 0870). The display should go blank for about four seconds (5 × 0.786) and then show the address following the breakpoint (087E). If it doesn't work, check the memory contents against Table 13-8. Be sure the RET has been changed to a RST 1. Check that you are performing step D correctly.

F) Set the D register to another value, and run the routine again. Calculate the expected delay, and measure the actual delay to see if it corresponds.

G) If the times correspond, the delay routine is working correctly. Change the RST 1 at address 087D back to a RET.

The delay routine has now been tested, and you are ready to test the CHNG routine.

H) The CHNG routine expects two inputs: the signal flag in the E register and the signal data in the H register. Set the E register to zero and the H register to 7D (0111 1101). This will set the left-hand signal to red and the right-hand signal to green when the routine is run.

I) Replace the RET at the end of the change routine (address 0863) with a breakpoint.

J) Run the change routine (starting at address 0855). The signals should be set as described in step H.

K) Set the E register to 1. This sets the signal flag and causes the two signals to reverse.

Continued

L) Run the change routine. The left-hand signal should now be green and the other one red.

M) Set the H register to the appropriate data to cause one signal to be red and the other yellow (refer to Table 13-1).

N) Run the change routine, and verify that the signals operate correctly.

O) If they do, the change routine is correct. If they do not, check whether the program is correctly stored and whether you performed the test procedure (beginning with step H) correctly.

P) Change the breakpoint at address 0863 back to a RET instruction.

The CHNG routine has now been tested, and you are ready to test the SEQ routine.

Q) The only input the sequencing routine expects is the signal flag (register E). Set register E to zero. This sets the signal flag to select the right-hand signal.

R) Change the RET at the end of the sequencing routine (address 0844) to a breakpoint.

S) Run the sequencing routine (address 0830). The right-hand signal should sequence through green and yellow. It does not change to red because that is done on the next cycle, when the other signal is changed to green. That part of the program is not being executed yet.

T) Set the E register to 1 to test the left-hand signal.

U) Run the sequencing routine again. The other signal should sequence from green to yellow.

V) If it does, the sequencing routine works. Change the breakpoint at address 0844 back to a RET.

All the modules have now been tested, and you are ready to test the entire system.

W) Run the main program (address 0810). Each light should sequence in turn. If they do not, check whether you have replaced the breakpoints with RET instructions and whether the main program is correctly stored.

X) The delay values were set unrealistically short to make the program's operation easier to observe. Try changing them to other values.

SUMMARY

This experiment tested and ran the traffic light controller. Each routine was tested individually, starting with the lowest level (those that call no other routines). By testing in this way, the number of possible problem sources is kept to a minimum, and faults can be more easily pinpointed.

IMPROVING THE TRAFFIC LIGHT CONTROLLER

There are many ways in which the basic traffic light controller can be improved. One factor to consider is that the two streets may have different amounts of traffic. If so, the green light time should be different for each.

To add this modification, the sequencing routine must accept another parameter in addition to the signal flag: the green light time. This can be accomplished quite simply. First, the MVI D,6 instruction is deleted from the sequencing routine (this instruction sets the green light time). Then instructions are added to the main program to set the D register (green light time). A listing of the modified main program is shown in Table 13-9.

Address	Contents	Label	Instruction	Comments
080E	16	TRAF:	MVI D,6	;Set Green Time A
080F	06			
0810	1E		MVI E,0	;Sequence signal A
0811	00			
0812	CD		CALL SEQ	
0813	30			
0814	08			
0815	00		NOP	
0816	00		NOP	
0817	16		MVI D,10	;Set Green Time B
0818	10			
0819	1E		MVI E,1	;Sequence signal B
081A	01			
081B	CD		CALL SEQ	
081C	30			
081D	08			
081E	C3		JMP TRAF	
081F	0E			
0820	08			

Table 13-9. Traffic Light Main Program with Variable Green Times

EXPERIMENT 13-2

Modifying the Traffic Light Controller

CONCEPT

In this experiment the traffic light controller is modified to provide different green light times for each signal.

PROCEDURE

A) If the traffic light controller program in Table 13-8 is not still in the μLab's memory, key it in (except for the main program).

B) Change the MVI D,6 in the sequencing routine (addresses 0835 and 0836) to NOPs.

C) Key in the modified main program shown in Table 13-9.

D) Run the traffic light controller (which now starts at 080E) and time the cycles to see if they are different.

E) Vary the green times for each signal to verify that the program performs as expected.

SUMMARY

This experiment demonstrated a simple modification of traffic light controller. The modular structure of the program made this change easy to implement.

PROGRAMMING EXERCISE 13-1: TRAFFIC LIGHT CONTROLLER MODIFICATIONS

There are many other possible ways to improve the traffic light controller. Here are some of them:

1. Provide different yellow times for each signal.
2. Make both signals red for a few seconds instead of always having one red and one green.
3. Use the two spare LEDs as left turn arrow indicators.
4. Use one of the input port switches to simulate a sensor in the road; do not allow the signal to change unless cars are waiting at the red light.

These are just a few of the possibilities. The first one requires minor changes to the main program and the sequencing routine. The second can be implemented with some additions to the sequencing routine, and the third is similar. The fourth, which is somewhat more involved, has been left for you to solve.

Modify the programs to implement one or more of these functions. If you add more than one, add them one at a time. By testing the program after each function is added, you will be able to find the problems more easily. Solutions to these exercises are in Appendix A.

Lesson 13

The first step in designing a program is the definition of the problem. A solution can then be designed, determining the general operation of the program. It is advantageous to partition the program into small, modular routines. The main program utilizes the routines, directing the flow without having to handle the details. Once the structure has been designed, the routines are flowcharted and coded. Finally, the main program and the subroutines are keyed in and tested.

Using a modular structure has many advantages. Dividing any problem into several smaller problems makes the solution easier, since the parts can be considered one at a time. Well-structured programs are easier to write, to debug, to modify, and to understand.

Subroutines communicate with each other by passing parameters. The traffic light controller used several parameters: the signal flag, the signal data, and the delay value.

Delay routines are used for providing a specified time interval between events. Since the time to execute an instruction is determined by the clock frequency, which is crystal-controlled, the time to execute a program loop can be precisely determined. By executing a loop a specified number of times, the desired delay can be obtained.

QUIZ

Lesson 13

1. The first step in designing a program is to define the _____.

2. A good way to write a complex program is as:
   a. one continuous program.
   b. a series of programs that are executed in sequence.
   c. one main program that calls a routine for each task.
   d. a main program and one subroutine.

3. A parameter is:
   a. a value passed from one routine to another.
   b. the contents of the program counter.
   c. the return address.
   d. the number of times a program is executed.

4. Delay routines:
   a. are limited to short time delays.
   b. are inaccurate.
   c. are rarely useful.
   d. are useful for a wide range of time delays.

LESSON 14

Software Control of Peripherals

INTRODUCTION

This lesson describes the software that controls the Microprocessor Lab's keyboard and display. Programs to read from the keyboard and write to the display are described. The programs are presented in two stages. First, the monitor's keyboard and display control subroutines are used. Then an independent program is described to show the software required to interface to a keyboard and display. The concepts described are applicable to a wide variety of microprocessor-based systems, since most systems include a keyboard and display.

THE KEYBOARD

The keyboard is arranged as a matrix of keys, with each row read separately (the hardware is described in Lesson 9). The data is read from each row and converted to the code for the pressed key by a software routine called KIND (Key Input and Decode), stored in the μLab's ROM. This routine and several others are described in Appendix E. Using the routine is quite simple: you call the routine, and it returns when a key is pressed. The accumulator then contains the code for the key pressed. (Table 14-1 shows the code for each key.) This monitor subroutine can be used in your programs to read the keyboard.

KEY	CODE
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	08
9	09
A	0A
B	0B
C	0C
D	0D
E	0E
F	0F
FETCH REG	80
DECR	81
FETCH ADRS	82
STORE/INCR	83
RUN	84
FETCH PC	85
INSTR STEP	86
HDWR STEP	F7

Table 14-1. Key Codes for KIND Routine

Using the Keyboard Read Routine

CONCEPT

The keyboard read routine, supplied as part of the monitor, is used to read the keyboard. The value of the pressed key is then compared to a specified value. A beep is generated if the specified key is pressed.

PROCEDURE

A) Code the program in Table 14-2 and key it in. This program illustrates a simple keyboard application. Two monitor subroutines are used: KIND (address 014B) and BEEP (address 0010).

Address	Contents	Label	Instruction	Comments
0800	_____	READ:	CALL KIND	;Read key
0801	_____			
0802	_____			
0803	_____		CPI 07	;Compare keycode
0804	_____			
0805	_____		JNZ READ	
0806	_____			
0807	_____			
0808	_____		CALL BEEP	;Generate beep
0809	_____			if key = 7
080A	_____			
080B	_____		JMP READ	
080C	_____			
080D	_____			

Table 14-2. Program to Read Keyboard and Generate Beep If "7" Is Pressed

The keyboard is read by the KIND routine, which waits until a key is pressed and then returns, leaving the key code in the accumulator. The CPI 07 instruction sets the processor's zero flag if the accumulator is equal to seven. The JNZ READ instruction then causes the program to jump back to the beginning if the zero flag is not set. If the zero flag is set, indicating that the key code was equal to seven, the JNZ instruction has no effect and the BEEP program is called. The process is then repeated. A beep is thus generated whenever the 7 key is pressed.

B) Verify that the program is correctly stored.

C) Run the program. Note that when [RUN] is pressed, nothing seems to happen. The program is running, but since the KIND routine scans the display while reading the keyboard, the display remains lit. The monitor program is *not* running.

Continued

D) Press [7]. The speaker will beep. Now press any other key. Only the 7 key generates a response.

E) Press [RESET] to return control to the monitor. Modify the program to detect a different key (refer to Table 14-1).

F) Test the modified program.

G) Press [RESET] to return control to the monitor.

SUMMARY

The monitor's key read routine (KIND) was used to read the keyboard. A beep was generated when a particular key was pressed. By changing the value that the key was compared with, any key could be detected. Using the monitor's KIND subroutine makes reading the keyboard a simple process.

PROGRAMMING EXERCISE 14-1: ELECTRONIC LOCK

SCANNING THE KEYBOARD

While the monitor's key read routine makes it easy to use the keyboard, it does not give you the chance to see what is involved in the reading process. To explain the technique used to read the keyboard, this section describes a program that reads the keyboard without using any monitor subroutines.

A diagram of the keyboard interface appears in Figure 14-1. As described in Lesson 9, the keyboard is scanned one row at a time. For simplicity, consider

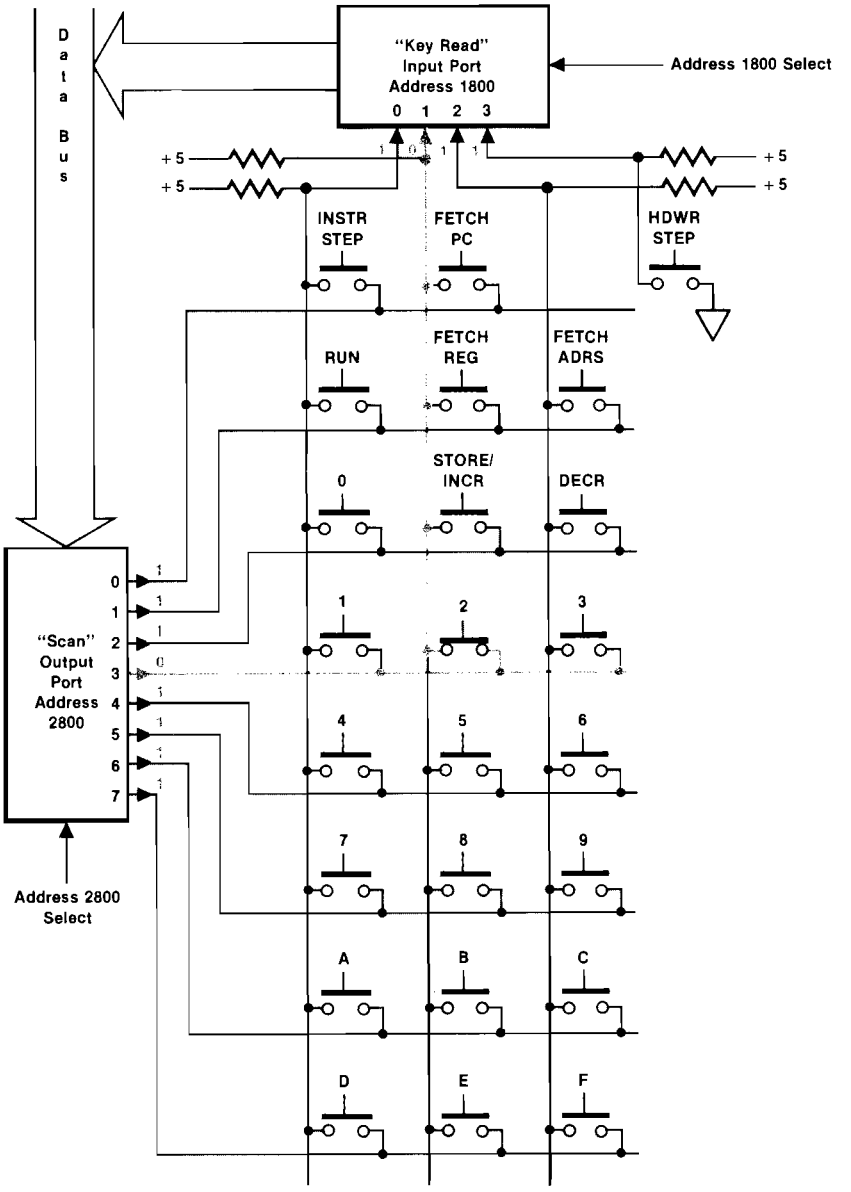


Figure 14-1. Keyboard Interface. The 2 key is shown pressed.

reading only a single row of keys (e.g., the "1," "2," and "3" keys). Reading a row of keys is a two-step process:

- Write data to the scan port to select the desired row.
- Read the column data from the key read port.

Each bit of the scan port is set high (logic one), except for the bit driving the row to be read, which is set low. Therefore, to read the 1, 2, and 3 keys the data 11110111 (F7 hex) is written to the scan port (refer to Figure 14-1). Then the key input port is read to get the column information. If no keys are pressed, bits 0-3 are all high (one) because of the pull-up resistors. If the 1 key is pressed, bit 0 is low. If the 2 key is pressed, bit 1 is low, and if the 3 key is pressed bit 2 is low (see Table 14-3). Note that none of the other keys affects the data read from the keyboard, since the scan port is set to "turn off" all but one row.

no keys pressed	XXXX X111
"1" key pressed	XXXX X110
"2" key pressed	XXXX X101
"3" key pressed	XXXX X011

Table 14-3. Key Column Data. The Xs in the most-significant five bits indicate that these bits will contain unknown data. The information of interest is contained in the least-significant three bits.

Note that the data read from the keyboard is a code that identifies the key, but it is not the actual key value. If the 2 key is pressed, for example, then the accumulator will contain 05 hex (assuming the five most-significant bits are zero). The KIND routine used in the previous experiment reads all the rows and converts these codes to actual key values.

Table 14-4 shows a program to perform the reading process just described. The scan port is set to select the desired row, and the column data is read into the accumulator.

```
MVI A,F7      ;Set scan port to 1111 0111 to select row
STA 2800
LDA 1800      ;Read key columns
```

Table 14-4. Program that Reads One Row of Keyboard.

Scanning the Keyboard

CONCEPT

In this experiment, the keyboard is read by setting the scan port and reading the key data port. This demonstrates the basic principle of the monitor's keyboard read program used in the previous experiment. The column data is compared with the expected value for a specified key, and a beep is generated if the values match.

PROCEDURE

A) Code and key in the program in Table 14-5. As previously described, this program reads a row of keys. The data is ANDed with a mask (0000 0111) to set all except the least-significant three bits to zero. These three bits contain the data from the keyboard, as shown in Table 14-3. The result is then compared with the value 0000 0101, which occurs if the 2 key is pressed. If the values are equal, a beep is generated by the monitor's BEEP routine.

Address	Contents	Label	Instruction	Comments
0800	_____		MVI A,F7	;Set scan port to
0801	_____			1111 0111
0802	_____		STA 2800	
0803	_____			
0804	_____			
0805	_____	READ:	LDA 1800	;Read columns
0806	_____			
0807	_____			
0808	_____		MVI B,07	;Mask off all bits except
0809	_____			three LSBs
080A	_____		ANA B	
080B	_____		CPI 05	;Is data 101 ("2" key)?
080C	_____			
080D	_____		JNZ READ	;If not, read again
080E	_____			
080F	_____			
0810	_____		CALL BEEP	;Yes-generate beep
0811	_____			
0812	_____			
0813	_____		JMP READ	;Read again
0814	_____			
0815	_____			

Table 14-5. Program to Test for "2" Key

B) Verify that the program is correctly stored.

C) Run the program, and try pressing different keys. Does it act as expected?

D) Press RESET to return control to the monitor. Modify the program to detect the "3" key instead of the "2" key (refer to Table 14-3).

E) Run the modified program and verify that it performs as expected. If you can not make it work, look at the solution in Appendix A.

SUMMARY

The keyboard was read by first setting the scan port to select a row and then reading the column data. The setting of the scan port disabled all but three of the keys. The exact key that was pressed was then determined by comparing the data read from the keyboard with the bit pattern generated by the desired key. To detect the "3" key, location 080C is changed to 03.

DEBOUNCING

Another factor that must be considered when reading a keyboard is debouncing. Switches do not, unfortunately, make one clean closure when pressed. They "bounce," as shown in Figure 14-2. When the two metal contacts first touch each other, they rebound and the switch is open again. Then they quickly touch again and bounce again. The same effect occurs when the switch is opened. This bouncing continues for 1-50 ms, depending upon the construction of the switch.

Contact bounce is important because it can cause the microprocessor to think that the key was pressed several times, when in fact it was only pressed once. The oscillations at the beginning and end of each pulse must be ignored. This can be done by adding debounce circuits to the input port, or it can be done entirely by software.

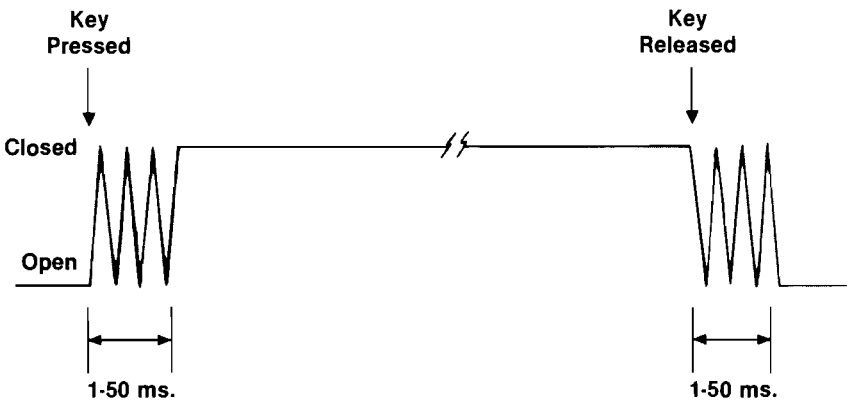


Figure 14-2. Switch Bounce

Figure 14-3 shows one way of implementing keyboard debouncing. First the keyboard is scanned, using the procedure discussed earlier. If no key is pressed, the process is repeated. When a key depression is detected, the function indicated by the key is performed. The keyboard is then ignored for 50 ms to ensure that the key is not read while it is bouncing. The keyboard is then scanned repeatedly, until the key is released. When the key is released, another 50 ms delay is inserted to avoid reading the key during the release bounce period.

THE DISPLAY

The μLab uses a six-digit, seven-segment LED display. Only one digit is on at any instant; the entire display is illuminated by writing repetitively to each digit in turn. This is referred to as *refreshing* or *scanning* the display. Each digit is on only one-sixth of the time, but each is flashing so fast that they all appear to be steadily lit. Multidigit displays are usually driven this way because it simplifies the hardware (as described in Lesson 9).

The μLab's monitor program contains a subroutine called DCD (Display Character Decoder), which controls the display. To use this program, the digits to be displayed are stored in six memory locations (one for each digit). The program reads the digits from the memory, converts the data to the code required by the display, and then refreshes the display.

There is another program that is helpful in creating a display. It is a routine called STDM (Store Display Message), which simply moves the message (the characters to be displayed) from your program to the location in memory where the display routine expects to find it. Using these two routines allows you to easily control the display.

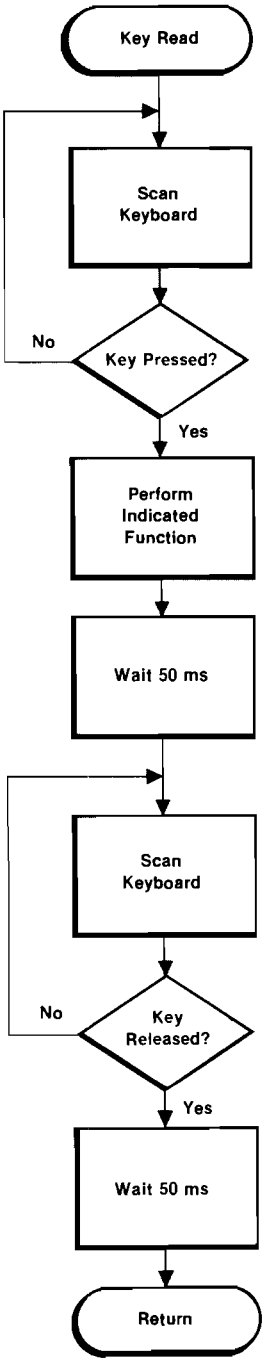


Figure 14-3. Keyboard Debounce Flowchart

Displaying a Message

CONCEPT

In this experiment, the routines contained in the μLab's ROM are used to display a message on the seven-segment LED display. The message is set by storing the desired character codes in the memory.

PROCEDURE

A) Code and key in the program in Table 14-6. The routine STDM starts at address 0018, and DCD starts at address 01E9.

Address	Contents	Label	Instruction	Comments
0800	_____		LXI D,0810	;Set message
0801	_____			address
0802	_____			
0803	_____		CALL STDM	;Move message
0804	_____			
0805	_____			
0806	_____	LOOP:	CALL DCD	;Display message
0807	_____			
0808	_____			
0809	_____		JMP LOOP	
080A	_____			
080B	_____			

Table 14-6. Message Display Program

This program first sets the D and E registers to the address where the message starts. The LXI D (load register pair immediate) instruction is used. This instruction loads the E register with the first byte after the opcode, and the D register with the second byte. Then the STDM routine is called, which moves the message starting at the address in D and E to the location where the DCD routine expects to find it. Finally, the DCD routine is repeatedly executed to refresh the display.

B) Key in the following data, which constitutes the message:

0810	06	(Right-hand digit)
0811	05	(Second digit)
0812	04	(Third digit)
0813	03	(Fourth digit)
0814	02	(Fifth digit)
0815	01	(Left-hand digit)

Continued

C) Run the program at 0800. The message 1234 56 appears in the display.

D) The refresh program can also generate a limited set of alphabetic characters. Table 14-7 shows the characters and the corresponding codes. Press RESET to return control to the monitor, and change the message data to the following:

0810	10
0811	10
0812	14
0813	12
0814	0E
0815	11

E) Run the program.

F) Now make up a message using the characters in Table 14-7. Press RESET to return to the monitor, store the appropriate codes in locations 0810–0815, and run the program.

Character	Hex Code	Character	Hex Code
0	0	F	F
1	1	(blank)	10
2	2	H	11
3	3	L	12
4	4	u	13
5	5	P	14
6	6	o	15
7	7	U	16
8	8	-	17
9	9	c	18
A	A	I	19
b	B	8.	1A
C	C	r	1B
d	D	-	1C
E	E		

Table 14-7. Character Codes for DCD Routine

EXPERIMENT 14-3

Continued

SUMMARY

This experiment used two monitor routines to place a message on the display. The first routine (STDM) moved the message to the RAM locations used by the display scan routine. The second routine (DCD) translated the character code to the seven-segment code and refreshed the display. Some alphabetic characters were also displayed.

PROGRAMMING EXERCISE 14-2: USING THE KEYBOARD AND DISPLAY

Write a program to display the value of the pressed key in the right-hand display digit. Use the routine KIND (as in Experiment 14-1) to read the keyboard and STDM and DCD (as in Experiment 14-3) to send the data to the display. Remember that if your program stores data in RAM it must use locations 0B00-0B90

CONTROLLING THE DISPLAY DIRECTLY

The display is controlled by two ports, as shown in Figure 14-4. Each bit of the scan port controls one digit, and each bit of the segment port controls one segment. Figure 14-4 shows the data format for these ports. To display a message, the characters are first converted to the seven-segment code. Each digit is then illuminated in turn (by setting the scan port), and the appropriate segments are turned on to display the desired character.

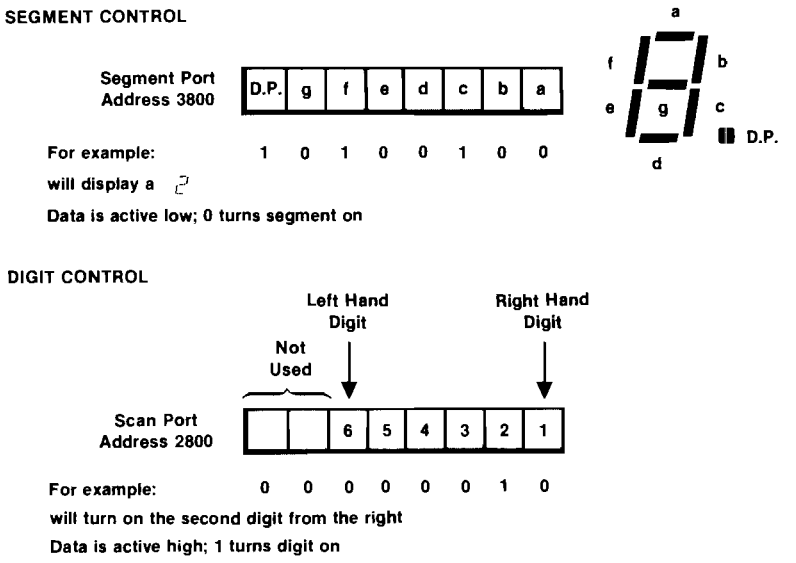


Figure 14-4. µLab Display Control

EXPERIMENT 14-4

Controlling the Display Directly

CONCEPT

In this experiment, a "2" is displayed in one digit of the display by setting the scan and segment ports. The data sent to the ports is then modified to change the character or the position in which it is displayed.

PROCEDURE

A) Code and key in the program in Table 14-8. This program first sets the scan port to turn on the third digit from the right. The data to generate a "2" is then sent to the segment port.

Address	Contents	Label	Instruction	Comments
0800	____	START:	MVI A,4	;Set scan port to select digit
0801	____			(4 hex=0000 0100 binary)
0802	____		STA 2800	
0803	____			
0804	____			
0805	____		MVI A,A4	;Set segment port to display
0806	____			character "2"
0807	____		STA 3800	
0808	____			
0809	____			
080A	____		JMP START	
080B	____			
080C	____			

Table 14-8. Program to Display a "2"

B) Run the program. The character "2" is displayed in the third digit from the right. Notice that it is brighter than the usual display. Normally all six digits are scanned, and each digit is on only one-sixth of the time. Now one digit is on all the time.

C) Press [RESET] to the return to the monitor. Change the data sent to the scan port to 8 hex (0000 1000 binary). This selects the fourth digit from the right.

D) Run the program. The character moves one position to the left.

E) Stop the program and change the data that is sent to the segment port to 9B. Referring to Figure 14-4, try to predict what character this will generate.

Continued

F) Run the program. A new character is displayed. Notice that since each segment is directly controlled, new characters can be made up at will.

G) Press [RESET]. Make up a new character (by referring to Figure 14-4), and change the program to generate it. Run the program to verify that your character is as expected.

SUMMARY

The μLab's seven-segment display was controlled directly, without the use of the monitor's subroutines. Data was sent to the scan port to select the digit and to the segment port to determine the character. Since each segment is individually controlled, new characters can be easily generated.

SCANNING ALL THE DIGITS

The next step towards a complete display control program is illuminating all six digits. As described earlier, this is accomplished by lighting each digit in turn.

Since writing to all six digits requires that you write new data to the segment and scan ports six times for each scan, a subroutine to perform this task makes the program much simpler. Table 14-9 shows a subroutine that writes the data in the B register to the scan port and the data in the C register to the segment port.

Address	Contents	Label	Instruction	Comments
0830	3E	DISP:	MVI A,FF	;Turn off segments
0831	FF			
0832	32		STA 3800	
0833	00			
0834	38			
0835	78		MOVA,B	;Set scan port
0836	32		STA 2800	
0837	00			
0838	28			
0839	79		MOVA,C	;Set segment port
083A	32		STA 3800	
083B	00			
083C	38			
083D	C9		RET	

Table 14-9. Display Subroutine

Note that the first step is to turn off all the segments, so that the data from the previous digit is not momentarily displayed. If this were not done, the scan port would be set to illuminate the next digit in the sequence, but the segment port would still contain data from the previous digit.

To light all six digits, a program to set the B and C registers and call the DISP routine once for each digit is required. Table 14-10 shows the listing for this program. The LXI instruction allows both the B and C registers to be set with one instruction, even though they are being used for different purposes. The first byte following the LXI opcode is loaded into the C register. It is the data that is sent to the segment port by the display routine. The second byte is loaded into the B register and is the digit select data (sent to the scan port). After the B and C registers are loaded, the DISP subroutine (Table 14-9) is called. This program controls all six digits of the display without using any monitor subroutines.

Address	Contents	Label	Instruction		Comments
0800	01	START:	LXI	B,018E	;Set data for right-hand
0801	8E				digit
0802	01				
0803	CD		CALL	DISP	;Display digit
0804	30				
0805	08				
0806	01		LXI	B,0286	;Set data for 2nd digit
0807	86				
0808	02				
0809	CD		CALL	DISP	;Display 2nd digit
080A	30				
080B	08				
080C	01		LXI	B,04A1	;Set data for 3rd digit
080D	A1				
080E	04				
080F	CD		CALL	DISP	;Display 3rd digit
0810	30				
0811	08				
0812	01		LXI	B,08C6	;Set data for 4th digit
0813	C6				
0814	08				
0815	CD		CALL	DISP	;Display 4th digit
0816	30				
0817	08				
0818	01		LXI	B,1083	;Set data for 5th digit
0819	83				
081A	10				
081B	CD		CALL	DISP	;Display 5th digit
081C	30				
081D	08				
081E	01		LXI	B,2088	;Set display for 6th digit
081F	88				
0820	20				
0821	CD		CALL	DISP	;Display 6th digit
0822	30				
0823	08				
0824	C3		JMP	START	;Repeat
0825	00				
0826	08				

Table 14-10. Display Scan Program

EXPERIMENT 14-5

Scanning the Display

CONCEPT

The program described above is entered and run. The entire display appears to be lit simultaneously, even though the digits are illuminated one at a time. This illustrates the basic operation of a multiplexed display.

PROCEDURE

A) Key in the display scan program in Table 14-10.

B) Key in the DISP subroutine in Table 14-9.

C) Verify that the programs are correctly stored.

D) Run the scan program. AbCd EF appears in the display.

E) Stop the program and change the message. You need to change the segment data for each digit (stored in locations 0801, 0807, 080D, 0813, 0819, and 081F). Figure 14-4 is helpful in determining the seven-segment code for the desired character.

F) Run the program again to see your message.

SUMMARY

A program that displays a message in all six digits was entered and run. While the program produces the same result as the monitor's display refresh program, it is less sophisticated. This program requires you to specify the message in seven-segment code, and it does not accept a message from any point in memory (the message is imbedded in the program). In spite of these differences, this program illustrates the techniques involved in display scanning.

Lesson 14

The μLab's keyboard and display are scanned by software. The display is driven one digit at a time, with each digit lit in turn. The keyboard is read one row at a time. This scanning process is acceptable because the microprocessor can scan the entire keyboard and display in much less time than it takes a person to respond.

The keyboard and display are easy to use by utilizing the monitor's routines. Writing a program to do all the scanning and code conversion without using the monitor's routines is more complicated, but it allows you to see the detailed operation of the keyboard and display scanning. It also permits new characters to be generated.

Most microprocessor-based products have a keyboard and a display. They are often interfaced using the techniques described in this lesson. Although a moderate amount of software is required, the hardware is very simple. This is an example of a function traditionally performed by hardware that can now be done with software.

QUIZ

Lesson 14

1. If both the 2 and 3 keys are pressed, the program in Table 14-5 will:
   a. beep as long as the 2 key is pressed.
   b. beep as long as the 3 key is pressed.
   c. not beep as long as the 3 key is pressed.
   d. beep as long as either key is pressed.

2. If a key switch that bounces for 75 ms is used with the debounce program of Figure 14-3, the system:
   a. will debounce the key correctly.
   b. might see one press of the key as two.
   c. might see one press as three or more.
   d. will think a different key has been pressed.

3. The display is illuminated:
   a. one segment at a time.
   b. one digit at a time.
   c. two digits at a time.
   d. all at once.

4. One function that is performed by the monitor's display scan routine DCD, but not by the program in Table 14-10, is:
   a. scanning of all six digits.
   b. control of all seven segments.
   c. conversion to seven-segment code.
   d. keyboard debouncing.

LESSON 15
Number Representations and Algorithms

INTRODUCTION

Microprocessor-based systems are often used to perform elaborate mathematical functions on a wide range of numbers, as in electronic calculators. However, the programs which have been used in this course are limited to positive integers between 0 and 255, and the functions are limited to simple logic, addition, and subtraction. This lesson discusses the techniques used to represent a wider range of numbers and perform complex mathematical operations.

NEGATIVE NUMBERS

Consider the problem of representing both negative and positive integers using an eight-bit word. Since there is no way to represent more than 256 different numbers using eight bits, the range is limited to about ±127. The first 128 numbers, zero through 127 (7F hex), are defined as positive numbers. Negative numbers are generated by counting "backwards" from zero. Like a hardware up/down counter, if a register is at 0000 0000 and is decremented, the next count is 1111 1111 (FF hex). FF hex is therefore the representation for -1. This representation is called *two's complement*. Table 15-1 shows the two's complement representation for -8 through +7.

Decimal	Two's Complement
7	0000 0111
6	0000 0110
5	0000 0101
4	0000 0100
3	0000 0011
2	0000 0010
1	0000 0001
0	0000 0000
-1	1111 1111
-2	1111 1110
-3	1111 1101
-4	1111 1100
-5	1111 1011
-6	1111 1010
-7	1111 1001
-8	1111 1000

Table 15-1. Two's Complement Representation of -8 through +7

Note that the most-significant bit indicates the sign. If the MSB is zero, then the number is positive. If the MSB is one, then the number is negative.

The procedure to calculate the two's complement representation is simple. For positive numbers, the two's complement and binary representations are the same (as shown by the first eight entries in Table 15-1). For negative numbers, the procedure for calculating the two's complement representation is as follows:

1. Write the binary representation of the absolute value (e.g., for -5 write 0000 0101).

2. Complement the binary number (this is called the *one's complement*, e.g., $\overline{0000\ 0101} = 1111\ 1010$).

3. Add one to form the two's complement (e.g., 1111 1010 + 1 = 1111 1011 = -5 two's complement).

The procedure to get the absolute value of a negative two's complement number is the same: complement the number and then add one.

For example, consider the two's complement number 1111 1011:

$\overline{1111\ 1011} = 0000\ 0100$ $\qquad$ $0000\ 0100 + 1 = 0000\ 0101 = 5$

Therefore, 1111 1011 is the two's complement representation of negative five.

Note that the number 1111 1011 could also be interpreted as 251 decimal, if it were considered to be straight binary rather than two's complement. It is therefore necessary to define the data as being two's complement and remember to treat it appropriately.

The two's complement representation is very convenient for arithmetic. Two's complement numbers, when added, subtracted, multiplied, or divided, yield results in two's complement. It is commonly used in microprocessor systems that must represent both positive and negative numbers.

LARGE AND SMALL NUMBERS

While two's complement provides a representation for negative numbers, the range is still limited to integers with absolute values of less than 129. This range can be extended in several different ways, depending upon how wide a range is required and the degree of precision needed.

Double Precision

The simplest technique for extending the range of numbers that can be represented is simply to increase the number of bits used to represent each number. This is often done by using pairs of words to represent a single number (see Figure 15-1). With the 8085, this can be done using the register pair instructions, which operate on sixteen bits at a time. Using two words for one number is called *double precision.* With an eight-bit processor, this extends the range to zero to 65,535, or ±32,767.

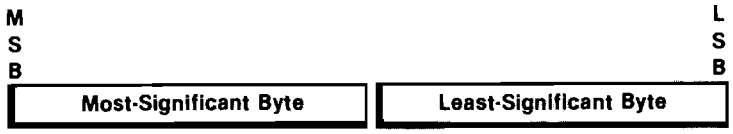


Figure 15-1. Double Precision

Fixed Point

Double precision extends the range of magnitudes, but what about numbers less than one, or between 3 and 4? Figure 15-2 shows a representation called *fixed point*. In this example, two bytes are used to store the number. The first byte is defined as being to the left of the decimal point (actually a binary point), and the second byte is the fractional part (to the right of the binary point). This allows numbers as small as $2^{-8} = 1/256$ to be represented, as well as fractional numbers such as 3.17. The resolution, however, is limited to 1/256th (about .004) and the range is limited to ± 127.

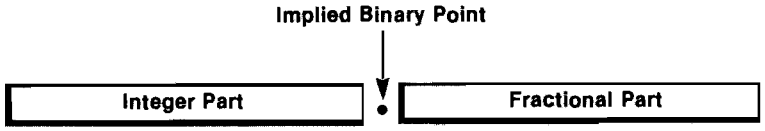


Figure 15-2. Fixed Point

Floating Point

Fixed point can be extended by using multiple bytes for each part of the number, but, unless a large amount of memory is dedicated to each number, it is still incapable of representing numbers such as 360,000,000,000 or 0.000000297. Note that these numbers contain many zeros, which are used as "place holders." These numbers can be easily represented by using "scientific notation," or *mantissa* and *exponent*. The mantissa is the magnitude of the number, adjusted to between zero and one. The number 360,000,000,000 for example, can be written as 0.36×10^{12} (0.36 is the mantissa and the exponent is 12), and .000000297 can be written 0.297×10^{-6} (0.297 is the mantissa and -6 is the exponent).

Suppose, then, that two bytes are used to represent each number as shown in Figure 15-3. One byte is the mantissa, and the other is the exponent. The range of values that can be represented, assuming that both the mantissa and exponent are stored in two's complement form, is about $\pm 10^{\pm 127}$. This is a very large range; 10^{127} is quite a large number, and 10^{-127} is very small.

Figure 15-3. Floating Point

This technique, called *floating point,* is commonly used for representing a wide range of numbers. More than two bytes are often used to obtain greater resolution (more digits in the mantissa).

Note that, as with all representations, the type of representation must be known in order to decipher the number. The same two bytes of data could be very different numbers if interpreted as a pair of two's complement numbers, a single fixed point number, or a floating point number. The software that operates on the numbers must know which representation is used.

DECIMAL NUMBER REPRESENTATION

Most microprocessor systems have decimal I/O devices, such as keyboards and displays. (The μLab is an exception, since it uses hexadecimal.) Since decimal is the natural form for most people, most microprocessor systems must accommodate it.

The problem is how to represent decimal numbers in the binary-oriented processor system. Suppose, for example, that the decimal number 28 is read from a keyboard. The number can be converted to its binary equivalent, 0001 1100 (1C hex). However, if this number is to be displayed on a decimal display, it must be converted back to the two decimal digits, 2 and 8.

An alternative method is to take each of the decimal digits, 2 and 8, and convert them independently to two four-bit binary numbers. The two four-bit numbers are then packed into one byte. Thus, 28 would be coded as 0010 1000. This is called *Binary Coded Decimal* (BCD). Note that the binary values 1010 through 1111 are never used in the BCD representation.

BCD is commonly used in systems that utilize decimal I/O, since it avoids the decimal-binary conversion process. One disadvantage is that it is inefficient in terms of storage space. The largest decimal number that can be stored in a byte using BCD is 99, whereas in pure binary it is 255. Arithmetic is also awkward in BCD, since it is not a "natural" number system. However, most microprocessors provide special instructions for accommodating BCD. (See the description of the DAA instruction in Appendix B.)

REPRESENTING ALPHANUMERICS

Many microprocessor systems must operate not only on numbers, but also on letters. For example, a computer terminal must read the characters from the keyboard and send them to the computer. Letters must somehow be represented by binary numbers.

The most common code for doing this, called *ASCII* (American Standard Code for Information Interchange), is shown in Table 15-2. Every character is assigned a binary value. Note that, as with all representations, the context of the information is important. For example, 0101 0100 may be the binary representation of the decimal number 84, the BCD representation of 54, or the ASCII character "T." The codes in the shaded areas are control codes, which provide special functions. The code "0A," for example, is used to cause a line feed on a printer or display.

00	NUL	20	SPACE	40	@	60	‘
01	SOH	21	!	41	A	61	a
02	STX	22	”	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	’	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6c	l
0D	CR	2D	—	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	72	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	—	7F	DELETE

SHADED CODES ARE CONTROL CODES

Table 15-2. ASCII Codes

The assignment of codes to characters is arbitrary, and there are many other possibilities. ASCII is currently the most widely used code, but another code called *BAUDOT* was very popular in the past. IBM machines use *EBCDIC* (Extended Binary Coded Decimal Interchange Code).

TABLE LOOK-UP

A common programming problem is the conversion of one number representation or code to another. For example, consider the problem of displaying a hexadecimal digit on a seven-segment display. It must somehow be determined which segments to turn on to display the appropriate character. A conversion from binary to seven-segment code is required.

This is done using a technique called *table look-up*. The segment patterns for each character are stored as a list in memory called a *table*. The first entry contains the segment pattern for the character "0," the next for the character "1," and so on. To translate a binary code to the corresponding seven-segment code, the code is simply "looked-up" in the table.

Figure 15-4 shows the flowchart for a program that converts binary data to seven-segment code. This program uses a table of seven-segment codes. The first entry in the table contains the seven-segment code for zero, the next entry the seven-segment code for one, and so forth. First, the binary number to be converted is added to the address of the first entry in the table. The result is the address of the table entry containing the desired seven-segment code. The contents of the addressed location are then read, and the conversion is complete.

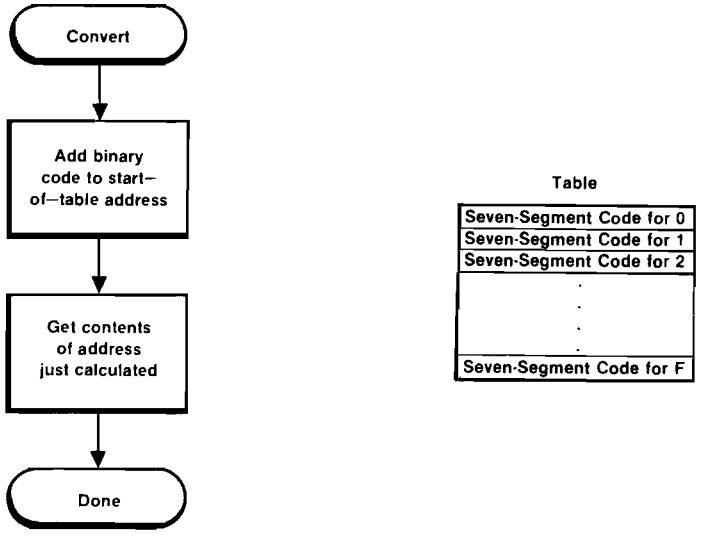


Figure 15-4. Table Look-Up for Binary to Seven-Segment Conversion

MATHEMATICAL ALGORITHMS

You may be wondering at this point how any really complicated functions can be performed with only this basic set of instructions. Calculators, which use microprocessors, perform division, multiplication, sine, cosine, and many other mathematical functions. Yet the direct arithmetic capabilities of the 8085 microprocessor (and most other eight-bit processors) are limited to addition and subtraction.

Complex mathematical functions are in fact performed (or approximated very closely) using only the simple add and subtract instructions. Multiplication can be performed very simply by a series of additions and shifts. To see how this is done, consider how you multiply "by hand." The multiplicand is multiplied by one digit of the multiplier at a time. The result from each successive digit of the multiplier is shifted left one place, and all the results are added. The same technique can be used for binary numbers. Multiplication by one digit is trivial: a number multiplied by one is the original number, and a number multiplied by zero is zero (see Figure 15-5). The entire multiplication can therefore be performed using only shifting and adding. A similar process performs division using shifting and subtraction. Techniques for performing a given operation are called *algorithms*.

```
        1 0 1 1
x       1 0 0 1
-----------------
        1 0 1 1
      0 0 0 0
    0 0 0 0
  1 0 1 1
-----------------
  1 1 0 0 0 1 1
```

Figure 15-5. Binary Multiplication

Calculating a function such as the sine of an angle is more complicated, but there are algorithms to approximate the sine function very closely using only simple functions. For example:

$$\sin X = X - \frac{X^3}{3\times2} + \frac{X^5}{5\times4\times3\times2} - \frac{X^7}{7\times6\times5\times4\times3\times2} + \frac{X^9}{9\times8\times7\times6\times5\times4\times3\times2} - \cdots$$

This is an infinite series, which will give an exact value for the sine of an angle if it is carried out for an infinite number of terms. Multiplication and division are used, but they can be performed using the addition and subtraction instructions.

In practice, of course, only a finite number of terms may be calculated. This causes the result to be an approximation, but it is very accurate if a large number of terms are calculated (which is no problem for a microprocessor).

The series described above provides an algorithm for calculating the sine of an angle. For every mathematical function, there is an algorithm that allows the function to be calculated using only the elementary operations that a microprocessor can perform.

REVIEW

Lesson 15

Using the standard binary number system, eight-bit processors are limited to integers between zero and 256. Negative numbers can be represented using two's complement, which assigns half the values to positive numbers and half to negative numbers.

Larger numbers can be represented in several ways. The simplest is double precision, in which two bytes are used for each word of data. This provides sixteen bits for each word, giving a range of zero to 65,535. For an even greater range, floating point is used. One byte is used to represent the mantissa and another for the exponent. Fixed point is another representation that uses one byte for the integer part and one for the fractional part.

BCD is a way of representing numbers in a form that is convenient for decimal input and output. Each decimal digit is converted separately to a four-bit binary number.

To represent alphanumerics, each character is assigned a binary code. ASCII is currently the most popular code.

Complex mathematical functions are performed using algorithms. Any function can be approximated (if not calculated exactly) by a series of additions and subtractions.

QUIZ

Lesson 15

1. 1111 0001 is the two's complement representation of:
   a. F1 (hex).
   b. -F1.
   c. 0E.
   d. -0F.

2. For a processor that uses four-bit words, the largest number that can be represented using double precision is:
   a. 15 (decimal).
   b. 255.
   c. 127.
   d. 65,535.

3. The greatest range of numbers is obtained using the _____ representation.

4. BCD is used because:
   a. decimal I/O is simplified.
   b. calculations are simplified.
   c. memory is used more efficiently.
   d. complex arithmetic functions can be performed.

5. Complex mathematical functions are performed by microprocessors:
   a. using special instructions designed for the function.
   b. using algorithms based on addition and subtraction.
   c. only by using special calculator chips.
   d. only if the processor is specially designed.

V

TROUBLESHOOTING MICROPROCESSOR SYSTEMS

In 1979 it was estimated that there were 10 billion digital ICs in nonconsumer products worldwide. If their average failure rate is .1% per year, then 10 million ICs die every year. If the average repair cost is $100 per failure, then $1 billion dollars is spent annually on field service.This amounts to 10 cents spent per IC per year on repairs. The maintenance cost over the life of the product can actually exceed the average selling price of the IC! Clearly, efficient testing and repair techniques can result in substantial cost savings.

Microprocessors can provide products with improved reliability, performance, features, and sophistication. But with these improvements come new service, troubleshooting, and repair problems. New tools and techniques are required to deal with the failures that can occur in these complex microprocessor systems.

In Lessons 16 and 17, the tools that are used to troubleshoot microprocessor-based systems are discussed. These include the logic probe, logic pulser, current tracer, signature analyzer, logic analyzer, and oscilloscope. Then in Lesson 18 microprocessor troubleshooting theory and the fault-finding process are described. Finally, in Lesson 19 these tools and techniques are used to troubleshoot faults in the μLab.

For the student already familiar with probes, pulsers, and current tracers, it may be desirable to read Lesson 16 lightly. Likewise, Lesson 17 can be read lightly if the student has had prior experience with signature and logic analyzers.

LESSON 16

Hand-Held Troubleshooting Tools

INTRODUCTION

Logic probes, logic pulsers, and *current tracers* are self-contained tools designed to stimulate and detect digital activity in logic circuits. Although internally complex, they are easy to use. These three instruments are very effective in a broad range of digital troubleshooting situations. The experiments that follow illustrate how they can be used individually and in combination to help locate faults in microprocessor-controlled digital circuits.

EXPERIMENT 16-1

Logic Probes

CONCEPT

Logic probes monitor in-circuit logic activity. By means of simple lamp indicators they tell you the logic state of a digital signal and allow brief pulses to be detected. In this experiment, the logic signals of the μLab are used to demonstrate the use of the HP 545A Logic Probe and show how it can be used for troubleshooting.

HP 545A Logic Probe

PROCEDURE

I. Setting-Up the Logic Probe

A) Be sure that the μLab is turned on and that the display shows μLAb UP. If not, check the power switch and press [RESET].

B) Connect the two power leads of the logic probe to a power supply having the same voltage as the logic circuit being tested. The logic power supply of the instrument under test can be used if it can supply the additional current required by the logic probe (70 mA). Power is connected by means of the spring-loaded connectors provided with the probe or by direct insertion onto the power pins of an IC test clip installed on an IC in the instrument under test (see Figure 16-1).

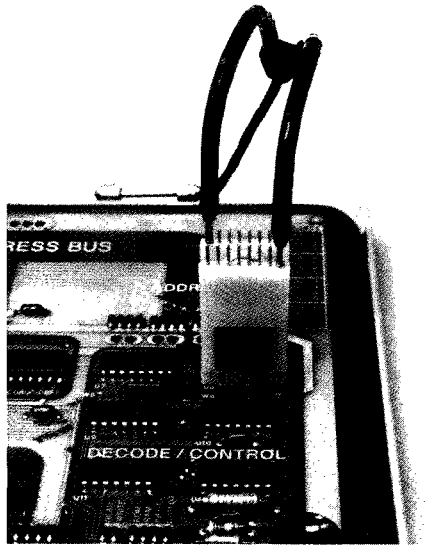


Figure 16-1. Probe Power Connection Using IC Test Clip

C) Figure 16-2 shows how the metal "grabbers" at the end of the spring-loaded connectors can be inserted through a pair of holes in the μLab power slots, one with a large hole and one with a small hole. These slots are located along the top edge of the Microprocessor Lab's circuit board. Connect the probe power cable with the red wire in it through a pair of holes in the right (+5) slot, and the cable with the black wire in it through a pair in the left (⏚) one. Notice that the lamp at the tip of the probe glows dimly. This is the "floating" or bad level indication.

Figure 16-2. Using "Grabbers" to Get Power from μLab's Power Slots

D) The slide switch on the probe sets the tip input logic thresholds for either CMOS or TTL levels. In the CMOS position these thresholds vary as a function of the power supply voltage. In the TTL position they are fixed. Be sure that the switch is in the TTL position when troubleshooting the μLab and most other 5 volt microprocessor systems. Even though they use MOS parts, most are designed to have TTL voltage thresholds at the pins.

II. Using the Probe

A) The lamp at the tip of the probe provides logic state information. Observe that with the metal probe tip not touching anything, the lamp is dimly lit, indicating the floating node or bad level condition. Each point in the circuit is called a *node.* All points that are wired directly together are part of the same node. If this indication (the dimly lit lamp) is present on an actual logic node, it means that a nonvalid logic level is present (between the "0" and "1" logic thresholds, as shown in Figure 16-3). If this node were a three-state bus line with all bus device outputs disabled (turned off) or perhaps the open input of a gate, a floating level might be acceptable. But if a floating level indication is present on a node where there should be a logic output, there is a problem.

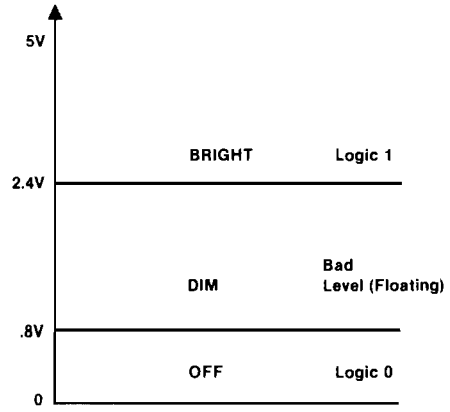


Figure 16-3. TTL Voltage Threshold for 545A Logic Probe

B) Check the probe by touching the tip to the ground slot. The light near the tip goes out, indicating a logic 0 level. Now touch the tip to the +5 slot. The lamp gets brighter, indicating a logic 1 level.

C) Use the probe to verify signal continuity between input port IC13-18 and the input port switch for the D7 line (located just below the right-most display module, as shown in Figure 16-4). Place the probe tip on IC13-18 and slide the left-most input switch (input 7) up and then down. Observe that the light follows the logic level change. Note that both the schematics and the PC board use the character "U" to designate ICs.

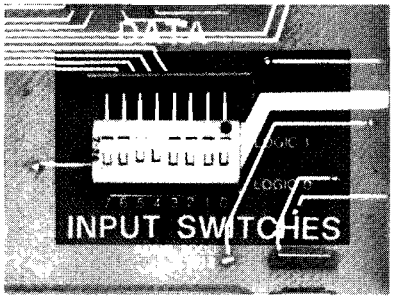


Figure 16-4. Input Port Switch for μLab

D) Referring to the schematic in Figure 16-5, probe IC18-12. This is a keyboard input line. Observe the constant bright light. It tells you that this signal is a steady logic 1. This is true because no keys in the 0-D key column (0,1,4,7,A,D) are being input to the microprocessor.

E) Now probe all lines on the output side of scan port IC17 (see Figure 16-5). The flashing light indicates that there is rapid logic activity on all of these lines. The logic probe slows down the lamp blink rate so that you can see it. If one of these lines is inactive, this condition will become readily apparent when you use the logic probe.

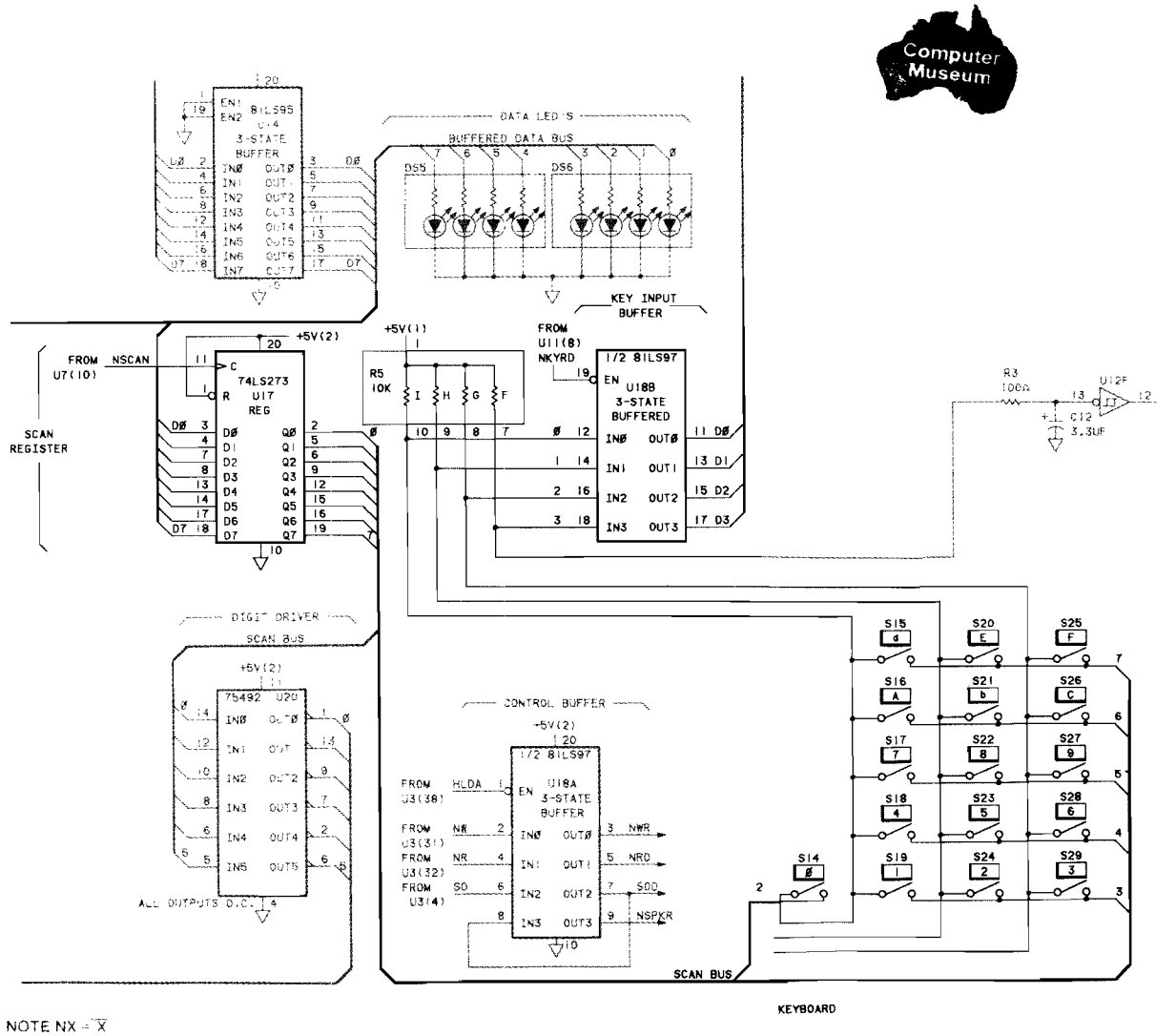


Figure 16-5. Key Input and Scan Circuit for Ø-D Key Column

EXPERIMENT 16-1

(Continued)

F) Now go back to IC18-12 again. While probing this pin, press any of the keys in the 0 through D column and observe the flashing light. This tells you that the signal is being switched through the keys. Pressing keys in the other columns does not cause activity on this line because they are not connected to it. Try a few.

G) Now probe IC18-19, the key input port select signal. Activity (a flashing light) means that this device is being enabled and is monitoring the keyboard. This is true because the monitor program is running and is reading the keyboard, looking for a new command.

CHECKPOINT

What does the logic probe tell us?

- Scan port IC17 is showing activity on its clock input and its outputs. The presence of activity on the outputs of IC17 indicates that it is probably functional. Activity on its clock input indicates that the circuits generating this signal are operating.
- Key input port buffer IC18 is enabled by the port select signal coming from the address decoding circuit.
- The keys in column 0-D are all operating properly as switches and are providing a signal to IC18-12.

How useful this knowledge is to you depends on the activity you observe at other nodes and how well you understand the system.

III. Probing Nodes

A) Using the schematic at the back of the book for reference, probe other nodes in the circuit. Compare the results to what you expected based on your knowledge of the μLab and the monitor program. Test the buses, chip select pins, input and output ports, and the control circuits. You will find that these circuits are very active.

B) Now probe IC10-9 (see Figure 16-6). This is the single-step control bit. A logic 0 level on this line indicates that the μLab is not in the hardware single-step mode.

C) Press the FETCH P C key on the μLab to fetch and display an address.

D) Press HDWR STEP and observe IC10-9 change to a logic 1. This level indicates that the μLab is now in the hardware single-step mode.

E) Now probe IC10-5. A low level on this line going to the Ready input (IC3-35) causes the activity on the buses to stop.

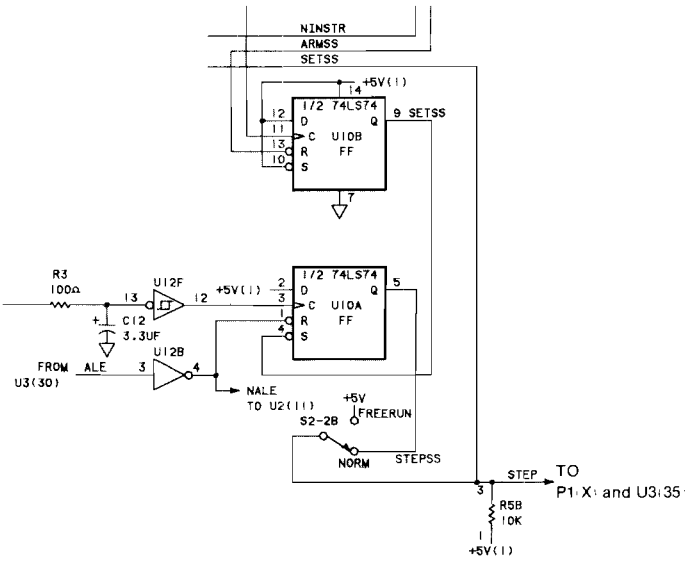


Figure 16-6. Single-Step Control Circuit

F) Re-probe some of the nodes examined before and verify that most of the activity of the μLab is frozen. The only active logic node in the system is the clock signal on pin 37 of IC3.

G) With the probe tip on IC10-5, press [HDWR STEP] and observe the probe blink once per key press. Although this pulse is only a few microseconds long, it appears much longer because the probe stretches the pulse to a length you can see.

H) If you press [HDWR STEP] and observe pins 1,3, and 5 on IC10, you can follow the logic signals for the single-step function (refer to Lesson 10).

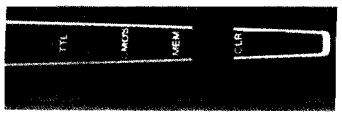
I) Now probe one of the data bus lines and press [HDWR STEP] . Observe that even when no logic state change occurs, the probe may blink. It blinks because address bus information is multiplexed on the data bus between machine steps. This happens so that the next address can be latched by IC2 when the $\overline{\text{ALE}}$ line goes high (refer to Lesson 10).

J) Probe IC2-11 (the clock input to the address latch). As you press [HDWR STEP] , verify that the inverted Address Latch Enable line ($\overline{\text{ALE}}$) pulses.

(Continued)

IV. Pulse Memory

The pulse memory feature of the logic probe is useful for capturing one or more pulses (or glitches) at times when it is not convenient for you to be looking at the probe tip. It is used when a pulse happens very infrequently or when the point you need to probe is hard to see.



Voltage Threshold Selector and Memory Display on Logic Probe

A) While still in the hardware single-step mode, probe IC10-5 and press the MEM CLR button on the logic probe. Observe that the memory light turns off. The pulse memory has been cleared. Keep the probe tip on pin 5.

B) Observe the MEM indicator as you press [HDWR STEP]. The indictor lights to show that a pulse has occurred. Additional pulses have no further effect. When using the memory, be sure that the probe tip remains in constant contact with the node so that the memory latch will not be set by electrical contact noise.

SUMMARY

The logic probe is a self-contained, easy-to-use tool for examining logic nodes. You used the probe to verify logic continuity, signal flow, address decoding, clock, switch, and bus device activity. You also used it to check the Microprocessor Lab's operating characteristics in the hardware single-step mode.

EXPERIMENT 16-2

The Logic Pulser

CONCEPT

In this experiment you use the logic pulser to inject signals into the μLab. The HP 546A Logic Pulser is an in-circuit stimulus device. It automatically outputs pulses of the required logic polarity, amplitude, current, and width to drive high nodes low and low nodes high. It also has several pulse burst and stream modes available.

PROCEDURE

I. Using the Logic Pulser

A) Connect the power leads of the logic pulser to the power slots of the μLab. (Leave the power to the logic probe connected).

B) To generate a single pulse, press and release the button on the pulser. Observe that the light at the tip blinks once.

C) Press and hold the button on the pulser. Verify that the light at the tip blinks once and then, about a second later, begins flashing rapidly. The pulser is now "programmed" into the 100 Hz pulse rate and can output brief high-energy pulses at a 100 Hz rate. Figure 16-7 illustrates this low duty-cycle pulse waveform. A low duty-cycle keeps the average energy delivered to an IC at a safe level to prevent potential damage.

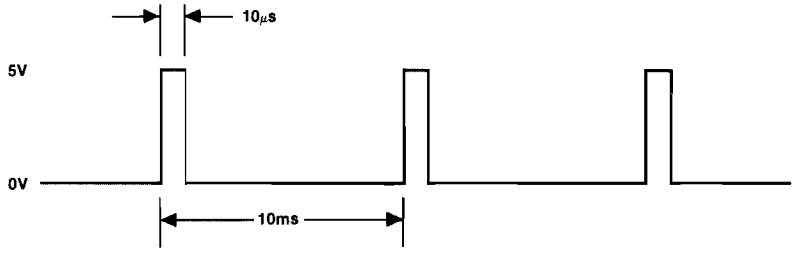


Figure 16-7. 546A Logic Pulser Output Waveform for Normally Low Logic Node

D) Slide the switch button forward to lock it. Notice that the pulser light continues to flash although your finger is not on the switch.

E) Read the label on the pulser. Other outputs can be programmed by pressing the button in rapid succession two, three, four or five times and then holding it down. Try these other output modes. Note that the 10 and 100 burst modes send out a burst of pulses and then pause before sending out another burst. Any of these modes can be locked on by sliding the switch button forward.

Programming Codes for Logic Pulser

EXPERIMENT 16-2

(Continued)

II. Pulsing a Node

A) Touch the pulser tip to IC18-9 (the output of the speaker buffer) and program it for the 100 Hz mode (see Figure 16-8). Listen carefully to hear a 100 Hz "crackle." The pulser is overdriving (forcing into the opposite logic state) the output of pin 9 and injecting a pulse stream into the speaker.

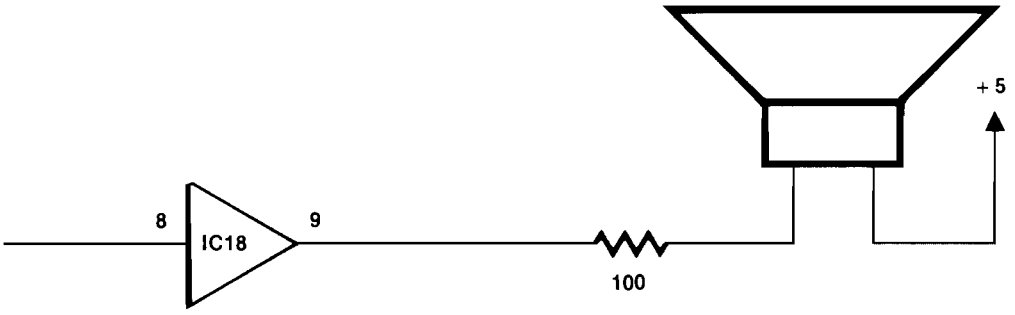


Figure 16-8. Speaker Drive Circuit for μLab

B) Touch the logic probe to this same point and verify that this output pin is indeed being overdriven by the pulser.

C) Try pulsing a V_{CC} or ground trace. Observe with the logic probe that the pulser cannot overdrive them. This is true because the power supply has a much lower output impedance than the logic devices that the pulser is designed to overdrive.

D) Now move the pulser tip to the right lead of the 100 ohm resistor located just below and to the right of IC18. This connection goes directly to the speaker and therefore increases the sound level. Now program the pulser for other output modes and listen to the difference.

III. Enabling a Device

A) If the display does not show μLAb UP press RESET to ensure that the monitor program is running.

B) Probe the clock input to the output port latch (IC15-11) with the logic probe (see Figure 16-9). No activity is occurring there at this time, indicating that the monitor program is not addressing the output port.

C) Now pulse this pin once with the pulser. Observe the μLab's output LEDs. Try it a few more times. Try it using other pulser output modes.

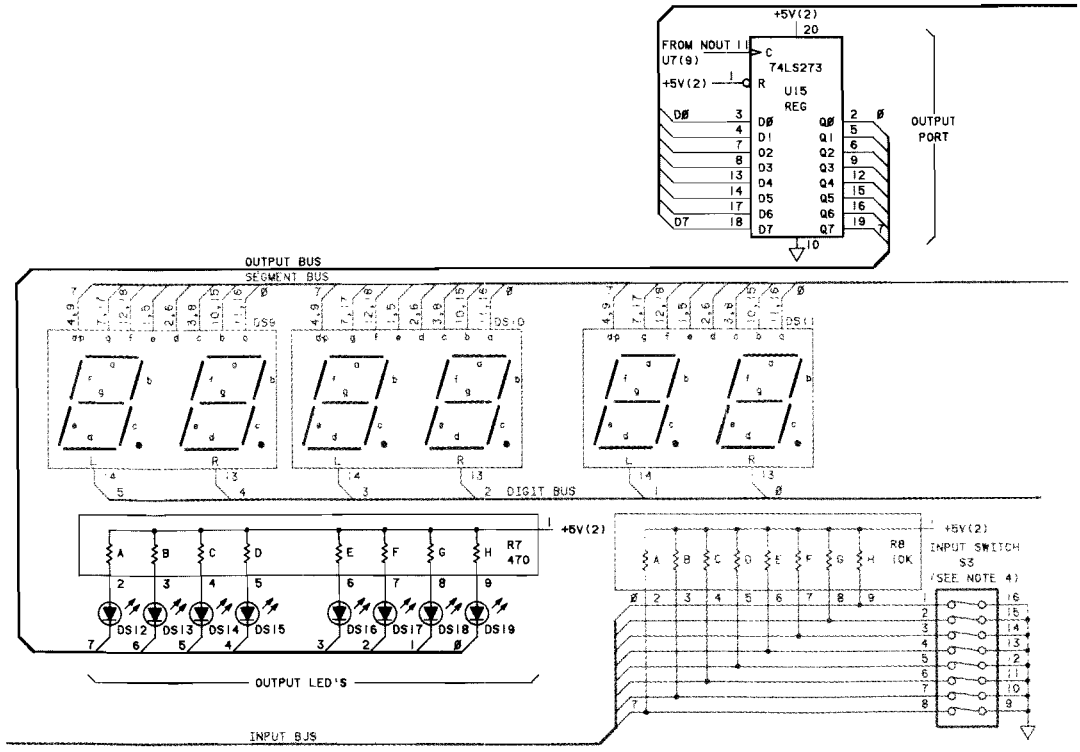


Figure 16-9. Output Port Latch Circuit

CHECKPOINT

Why did the LEDs behave so randomly? What have you shown?

- From the schematic, observe that the inputs of output latch IC15 are all connected to the data bus. The outputs go to the LEDs.
- Since the monitor program is running, there is activity on the data bus. Random bit patterns appear on the data bus at the moment when it is asynchronously sampled by the signal from the pulser.
- The pulser causes the latch to store data from the data bus with no regard to what that data is. It ignores the address bus and the control signals.
- This data storage tells you that IC15 can latch data when clocked. Therefore, its clock input (pin 11) is functional.
- You can also check the ability of all eight of the latch's bits to store and output both logic states. Pulse pin 11 a number of times and watch each of the LEDs change state. Had the output LEDs not been available to observe, a logic probe could have been used in their place.
- You have verified, to a fairly high level of confidence, that IC15 is good. Had it not performed in the manner that it did (given the activity present on the data bus lines and lack of it on pin 11), you could be fairly certain that something was faulty.

EXPERIMENT 16-2

(Continued)

SUMMARY

The logic pulser forces overriding pulses into logic nodes. It can be programmed to output single pulses, pulse streams, or bursts. The pulser can be used to force chips to be enabled or clocked. Logic circuit inputs can be pulsed while observing the effects of their outputs on the circuit under test. In this experiment you used the μLab's speaker to examine the various output modes of the logic pulser. Then the pulser was used to clock random data from the μLab's data bus into the output port latch. The output port LEDs verified that the latch was operating properly.

EXPERIMENT 16-3

Stimulus-Response Testing Using the Probe and the Pulser

CONCEPT

This experiment demonstrates how the pulser can stimulate circuits in the μLab and how it can be combined with the logic probe to perform stimulus-response testing. The logic pulser injects a stimulus signal into a node, and the logic probe monitors the response of other circuit nodes in the signal propagation path.

HP 546A Logic Pulser

PROCEDURE

I. Tracing Logic Flow

A) Clear the μLab memory by turning the power off and then on. Fetch the first RAM address (0800) by pressing [FETCH P C]. Press [HDWR STEP] to enter the hardware single-step mode at address 0800. Verify that the bus LEDs are reading RAM data 00 at address 0800. The display is blank since the system has stopped.

B) Using the logic probe, verify logic 0 levels on the D0 data bus line and the RAM enable line (IC5-8). This tells you that the RAM is enabled and that the data present on its D0 output is 0. Now insert the probe into the D0 test hole (see Figure 16-10) and leave it there.

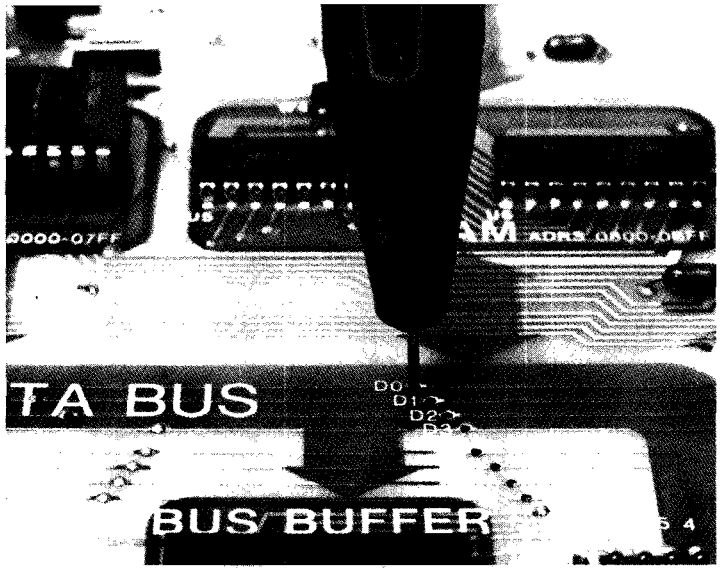


Figure 16-10. Logic Probe Inserted into DØ Test Hole

EXPERIMENT 16-3

(Continued)

C) Pulse D0 once with the pulser. Observe that the probe flashes once, indicating that the pulser has overdriven the data bus line and injected a logic 1 pulse. This demonstrates how simple it is to inject a pulse into a node.

D) Pulse the RAM enable (IC5-8) and observe the probe. It flashes on because the RAM chip is momentarily disabled by the logic 1 pulse supplied by the pulser. The RAM outputs turn off, and the data bus is pulled to a logic 1 level by the 10K pull-up resistors (just above the data bus switch). You have demonstrated that the RAM enable pin can cause the D0 output to disable.

E) Observe from Figure 16-11 that the $\overline{\text{RAM}}$ signal at IC5-8 comes from IC11-3 (the output of an OR gate). Use the logic probe to verify that this gate's inputs (IC11-1 and 2), as well as its output (IC11-3), are all low (logic 0).

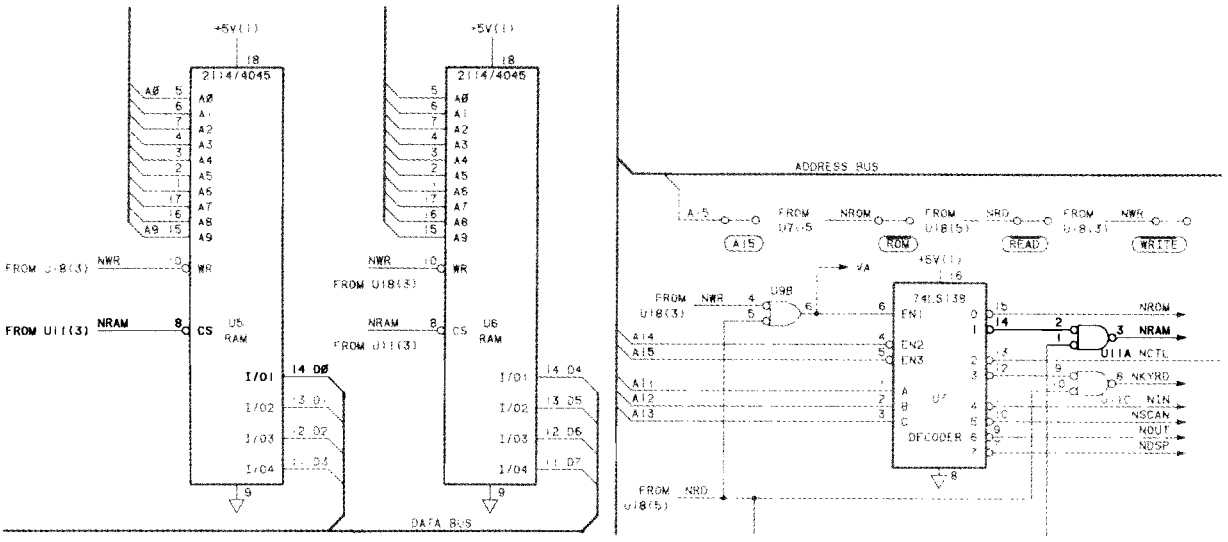


Figure 16-11. RAM Decoder Circuit

F) With the probe placed on the output of the gate (IC11-3), individually pulse its inputs (pins 1 and 2) to verify the correct logical operation of the gate. With either input pin pulsed (logic 1), a pulse appears at the output.

G) Move the probe back to data bus test point D0 and observe that pulses on the gate's inputs (IC11-1 and 2) propagate through the gate to pin 3. The pulse then continues to the RAM enable pin (IC5-8), through RAM IC5, and finally to the RAM output on the D0 line (IC5-14).

H) Back-trace the signal from IC11-2 to the address decoder (IC7) inputs. Pulse these inputs, one at a time, and observe that input pins 2, 3, 4, 5, and 6 all cause D0 to pulse. Note however, that IC7-1 (which connects to A11) does not affect D0. All of these inputs cause either another decoder output to be true (the select inputs, pins 1, 2, and 3) or all outputs to disable (the enable inputs, pins 4, 5, and 6). The reason that a pulse on IC7-1 doesn't cause a change of state on the D0 bus line is that the ROM is enabled when the A11 line goes low, and the D0 ROM data for this particular address (0000) is also a 0.

II. Pulse Burst Demonstration

The logic pulser can generate pulse bursts of 10 or 100 pulses. This is useful when a precise number of clock pulses are needed to preset a counter, shift register, or other sequential circuit.

A) Turn the μLab's power off and then on to clear the memory. Press [FETCH PC] to set the address to 0800 (the first address of RAM). Press [HDWR STEP] to stop the system. Observe that the address bus LEDs indicate address 0800.

B) Locate the clock input on the single-step flip-flop (IC10-3). Figure 16-12 shows that this pin is connected to a debounce circuit that goes to the [HDWR STEP] key. Press the [HDWR STEP] key a few times and observe that the address bus LED's increment.

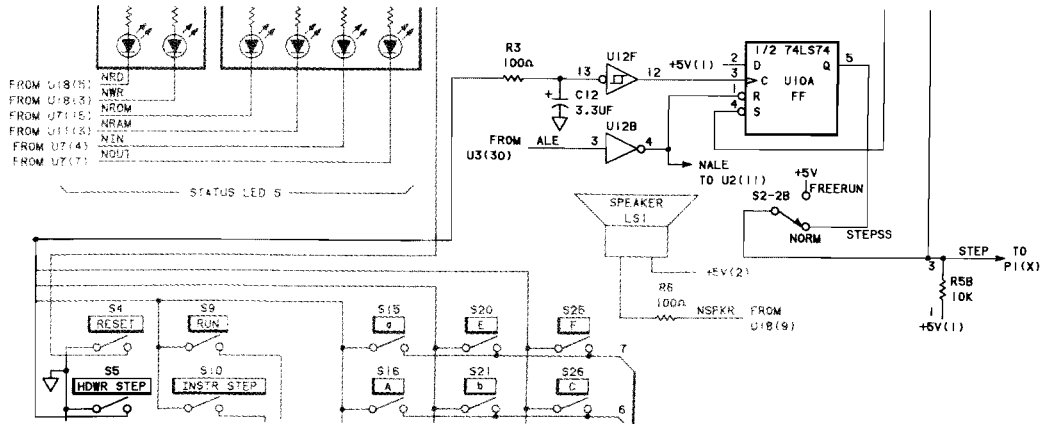


Figure 16-12. Pulser Used to Increment Single-Step Circuit

EXPERIMENT 16-3

(Continued)

C) Pulse the clock input (IC10-3) with the pulser and observe that these LEDs again increment.

D) Program the pulser to the 10 Hz and 1 Hz modes and repeat step C. The address bus LEDs increment at the pulse output rate.

E) Program the pulser to the 100 Hz mode and repeat step C. Observe that as the address increments from 0800 to OAEE, the data bus LEDs are off. During this sequence the μLab performs NOP instructions (00) that are stored below address OAEE in the RAM by the power-up program. After the μLab reaches address OAEE, it begins to execute a program and the bus LEDs go wild with activity. This activity results because each time a single-step cycle occurs, an instruction (or part of one) is executed. During the time that the nearly 1000 NOPs are executed, no address jumps or data transfers occur. But above address OAEE, there are stored programs and data instead of zeros.

F) Press [RESET] to return to the monitor. Fetch address 0800. Press [HDWR STEP] to return to the first NOP instruction in RAM.

G) Place the pulser on IC10-3. Program two 100-pulse bursts. Program five 10-pulse bursts. Output six single pulses. You will have output a total of 256 pulses, and the address bus LEDs should indicate address 0900 (0900 hex minus 0800 hex = 256 decimal).

H) Program a few other pulse counts (511 and 361 are interesting). Repeat step F when the RAM address goes beyond OAEE. Remember that there are less than 1024 RAM locations that are empty (00), so don't try to burst 1024 successive pulse steps and expect to see something predictable.

SUMMARY

Logic circuit inputs were pulsed with the logic pulser while their outputs were monitored with the logic probe. Signal propagation through the logic elements was then examined using the logic probe. In the hardware single-step mode, pulse bursts were injected while their response on the address bus LEDs was observed.

EXPERIMENT 16-4

The Current Tracer

CONCEPT

This experiment demonstrates how the logic pulser can be used with the current tracer. The HP 547A Current Tracer detects current activity on logic nodes by means of an inductive pick-up at its tip. By adjusting its sensitivity control and observing the internal lamp's intensity when placed on a pulsing logic signal line, you can identify current paths and relative magnitudes and locate a bad device on a node. In this experiment, you use the current tracer both alone and with the logic pulser to observe current activity in the μLab.

The troubleshooting attributes of the current tracer are less obvious than those of the probe or the pulser. This is true because the current tracer deals with currents rather than the voltages you have become used to. However, there is a great advantage once you gain confidence with it (and it really isn't that difficult). Frustrating troubleshooting experiences with problems such as stuck nodes and shorted power supply lines can be dealt with directly. Previously, these types of problems may have led you to find the fault by cutting traces, snipping pins, or worse.

HP 547A Current Tracer

PROCEDURE

I. Using the Current Tracer

A) With the logic probe and pulser power leads still connected, connect the power leads of the current tracer to the μLab's power slots.

B) Press [RESET] if the display doesn't already show μLAb UP

C) For proper current tracer operation, the tip must be correctly positioned on the circuit trace being checked. The tracer tip should always be perpendicular to the board and have the small holes on the sides of the tip aligned with (facing up and down) the current path (see Figure 16-13).

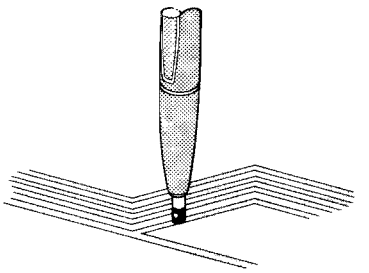


Figure 16-13. Proper Orientation of 547A Current Tracer on Circuit Trace

EXPERIMENT 16-4

(Continued)

D) Place the tip of the current tracer directly on any of the address bus lines on the top portion of the board. Since its tip is insulated, you can make direct contact with electrical conductors. Be sure the tip is correctly oriented.

E) Adjust the sensitivity knob and observe that the lamp near the tip of the tracer changes in intensity.

F) While still in contact with an address bus line, set this adjustment to about half of the maximum lamp brilliance.

G) Now, without changing the sensitivity, probe the other address lines and verify that they all produce about the same intensity. The current tracer is responding primarily to the current going from address bus drivers IC1 and IC2 to the address bus LEDs.

H) Rotate the tip of the tracer over an address line and observe that the lamp gets dimmer. It dims because the current does not travel across the traces but along them. This shows the importance of maintaining the correct orientation of the probe tip when tracing current along a trace.

I) Referring to Figure 16-14, probe other traces on the board. Notice that the power supply and display traces (which carry the large switching currents generated by many of the ICs) produce the brightest lamp readings. The lines with no activity produce no lamp indication.

J) The sensitivity adjustment allows the current tracer to detect currents from below 1 mA to more than 1 Amp. Readjust this control to compare the current levels on various lines of the board and, referring to the schematic, see if they agree with what you expect. Hint: It is easier to detect small differences in currents if the sensitivity is set to produce a very dim lamp indication.

II. Current Pulse Detection

A) Turn the μLab off and then on to clear the memory.

B) Adjust the current tracer to give a bright lamp indication when placed on an address line (high sensitivity).

C) Press [FETCH PC] [HDWR STEP] . Observe that the current activity (changing current magnitudes) on the address lines and all the other logic lines has stopped. Activity is still present on the clock lines, the microprocessor power pins, and some of the power traces. This activity indicates that the microprocessor is still running internally, waiting for its ready line to be released by the hardware single-step circuit.

D) While looking for current on address line A0, between the address bus probe hole and the

address bus LED, repeatedly press [HDWR STEP]. Verify that the tracer lamp flashes each time the key is pressed.

E) Move to the A1 line and watch it flash with alternate presses of the [HDWR STEP] button (coincident with the change in state of the A1 LED). The flashes illustrate two characteristics of the current tracer:

- It responds only to a change of current (not DC current).
- It has a current pulse "stretching" capability similar to that of the logic probe.

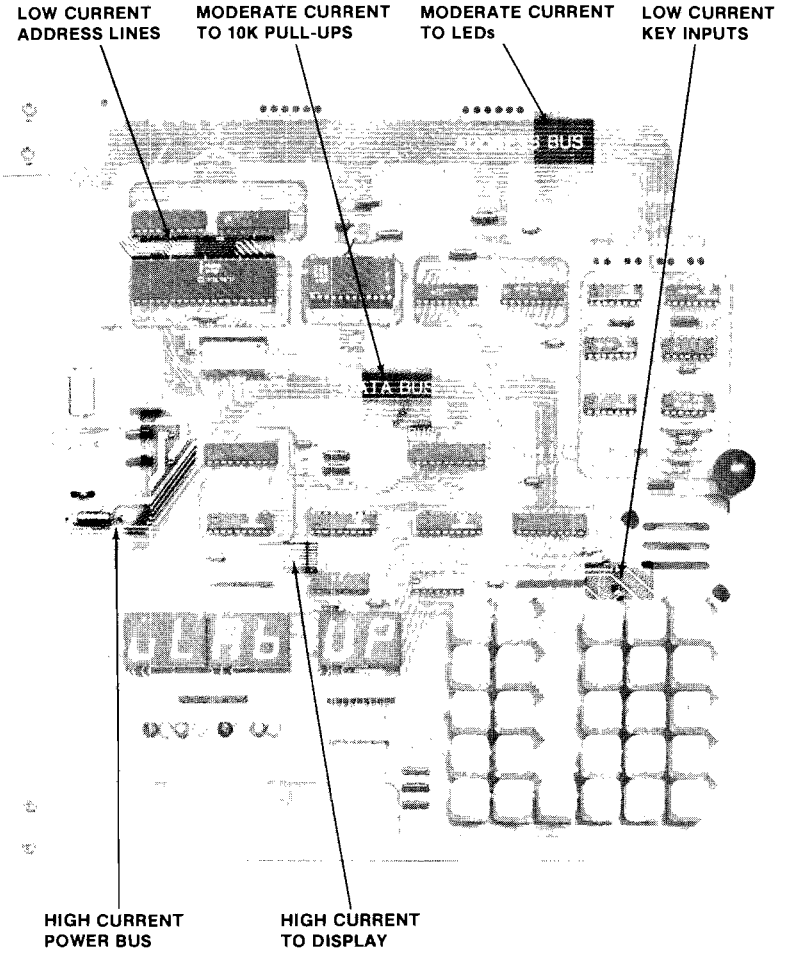


Figure 16-14. Current Activity for Various Circuits on μLab

EXPERIMENT 16-4

(Continued)

III. Stimulus-Response Testing Using the Pulser and the Tracer

The logic pulser can be used to provide pulse activity on an inactive node. The current tracer can then be used to follow the current flowing from that node.

A) Observe that all of the data bus lines are logic 0 (the data bus LEDs are all off). You can conclude that at least one bus device is enabled. When no bus devices are enabled, the bus lines are pulled high by the 10K pull-up resistors on the data bus. (Solder bridges or other shorts to ground on bus lines could also cause this condition.)

   Since the microprocessor is stopped (in the hardware step mode), only steady-state DC currents are present on the data bus lines. The current tracer does not respond to DC. However, if the logic pulser is used to inject current pulses into a node, then the current tracer can see where it goes. This will be done with a data bus line.

B) Place the logic pulser tip into the D0 test hole on the data bus. Program and lock it in the 100 Hz mode (press and slide the switch). The node is now being pulsed with as much current as it takes to overdrive (change the logic state of) the circuits on the D0 line. Use the logic probe to verify that the pulser is doing this.

C) Place the tip of the current tracer in line with the tip of the pulser so that the current going into the node can be determined (see Figure 16-15).

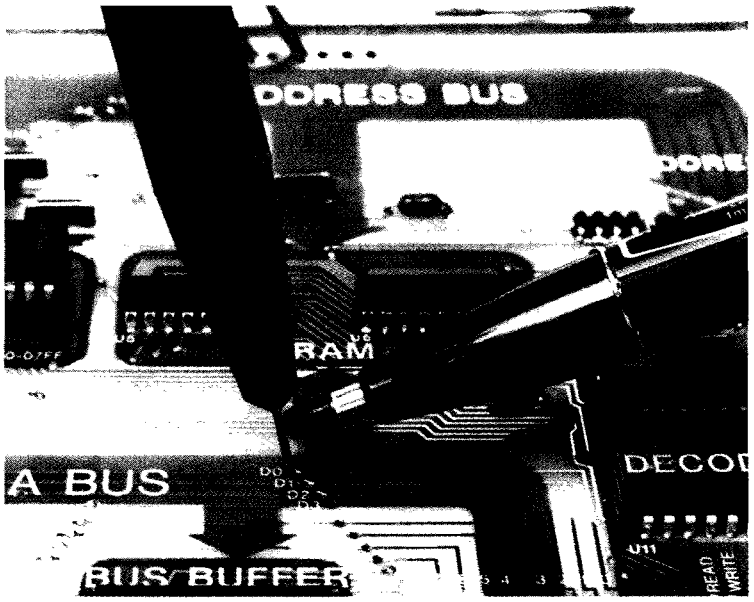


Figure 16-15. Measuring Pulser Node Current with Current Tracer

D) Adjust the sensitivity to produce a dim indication on the tracer lamp. Be sure to keep the current tracer tip in line with the pulser tip. Observe that the knob is roughly midway in its rotation. This position means that the current is probably between 10-100 mA, more current than 10K pull-ups or logic inputs will absorb. It is also much less than a hard short to ground since the pulser can output more than 1 Amp into a ground or V_{CC} short. It falls roughly into the range of an MOS or TTL output overdrive current. Comparisons with known circuit currents and experience with the current tracer will allow you to make educated guesses such as these.

E) Now, without further adjustment of the sensitivity, follow this current onto the board. Check each side (left and right) of trace D0 to see in which direction the current goes. Notice that it doesn't seem to go either way. Since the current doesn't just disappear, it must be going to the other side of the board. Pull firmly on the black knob at the middle of the right edge of the board to unlock it. Then tilt the board up on its hinges and check the current on the bottom side of the board near the D0 test point hole (see Figure 16-16). You should now be able to follow the current along the trace to IC5-14. You can actually follow the current right up this pin of the IC.

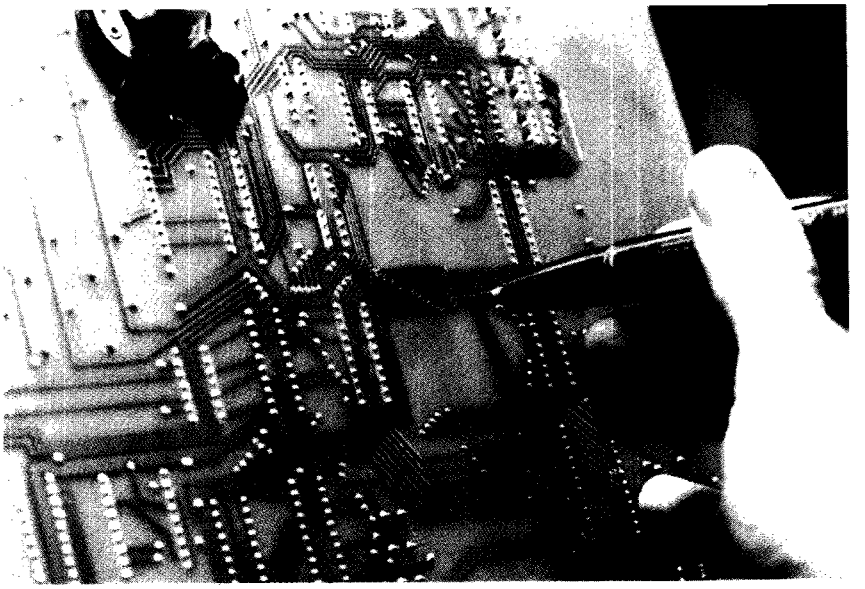


Figure 16-16. Following Current on Rear Side of Board with Current Tracer

This current path should come as no surprise to you since you can see from the status indicators that the μLab is performing a READ from the RAM. Had the RAM not been enabled, this particular current path would have indicated a faulty RAM output. With five other devices connected to this node, you are able to determine which one of the six is on. Had this RAM pin been stuck on, the current tracer would have led you to it. With voltage-based instruments it is difficult to isolate bus devices such as this.

EXPERIMENT 16-4

(Continued)

F) Turn off the pulser by sliding the switch back. Press [RESET] to return to the monitor. Press [FETCH ADRS] [0] [0] [0] [0] . Press [HDWR STEP] .

G) Set the pulser to the 100 Hz mode and trace the current path on the D0 line. (Refer to steps B through D if you have forgotten the procedure.) You should end up at the ROM, IC4-9 (because the ROM is now enabled). Try tracing current down other data lines.

H) Now place the pulser on a V_{cc} or ground trace and follow the current. The first difference you will notice is the magnitude of current put out by the pulser. You can get full tracer lamp intensity even with minimum sensitivity (1 Amp). The second difference is that the current path always leads back to the ground cable of the pulser since the pulser output current flows through this lead. When power bus shorts occur, the pulser and tracer can be useful for locating time.

SUMMARY

The current tracer is used to track down the cause of stuck nodes. It can tell you approximately how much pulse current is present and what path it takes. When a logic pulser is used to inject current into a node without pulse activity, its impedance and the general nature of the problem (e.g., gate output, hard short) can be estimated. Then, the actual low impedance point can be found by tracing the path of the current from the logic pulser to that point on the node. The current either goes someplace it should not (a shorted line), or it enters a component that is stuck, shorted, or turned on.

Lesson 16

The logic probe, logic pulser, and current tracer are convenient, handheld troubleshooting tools that are used individually and in combination to troubleshoot digital circuits. All three can be powered from the product under test.

The logic probe indicates the logic state of a node ("0", "1", or bad level) and shows you when a node has activity on it. It is particularly useful at spotting stuck nodes and single-event occurrences.

The logic pulser injects pulses into a node, forcing responses in the circuit. These pulses can then be monitored with the logic probe, the current tracer, or the response of the product.

The current tracer monitors current activity on signal lines (circuit traces, component leads, and wires). The path of current in a stuck node can be traced to the fault.

Used together, these tools can help you decide which IC to replace on a faulty node. If the logic probe indicates an "open" or "high" logic condition, the IC driving the node is probably faulty. If the probe indicates a low, check the current on the node with a current tracer. If no current is present on the faulty node, the driver IC is defective. If an excessive amount of current is present, the current tracer can locate the shorted trace or IC input. Additional techniques for locating the faulty circuit node are discussed in Lesson 17.

QUIZ

Lesson 16

1. A rapidly flashing logic probe tip tells you that the logic node being probed:
   a. is at an illegal logic level.
   b. has rapidly changing logic activity.
   c. has unstable logic activity.
   d. is a stuck node.

2. The response of the lamp near the tip of the logic probe to a single one-microsecond logic 1 pulse is:
   a. to flash on too briefly to see.
   b. to flash off too briefly to see.
   c. to flash on long enough to see.
   d. to go on and stay on.

3. The output pulses of the logic pulser:
   a. are too narrow for the logic probe to respond to.
   b. can damage logic circuits.
   c. can only force high nodes low.
   d. can be used to overdrive logic nodes high or low.

4. The amount of current output by the logic pulser:
   a. is always at least 1 Amp.
   b. depends on whether the logic probe or current tracer is being used.
   c. depends on which pulse output mode it is in.
   d. varies with the impedance of the node it is driving.

5. The logic pulser should be used with the current tracer when:
   a. there is no current activity on the test node.
   b. there is no current on the test node.
   c. the stuck node has not yet been found.
   d. the node is not stuck.

6. When tracing current with the current tracer, the sensitivity control should usually be set so that the lamp:
   a. is slightly less than maximum intensity (bright).
   b. is slightly more than minimum intensity (dim).
   c. just goes out.
   d. begins to flash.

LESSON 17

Signature and Logic Analyzers

INTRODUCTION

Signature Analyzers and *Logic Analyzers* are two specialized classes of test instruments used in product development, production, and field service. They can also be very effective in troubleshooting microprocessor systems. This lesson describes these instruments and demonstrates how they can be used to troubleshoot microprocessor-based products.

SIGNATURE ANALYSIS

Signature Analysis (SA) is an easy-to-use and highly accurate technique for identifying faulty logic nodes. The signature analyzer can convert the long, complex serial data streams present on microprocessor system logic nodes into four-digit "signatures." These signatures tell whether or not the node is acting properly.

The character set used for signature analysis contains sixteen characters (0-9,A, C,F,H,P,U). By using this character set, the confusion resulting from the seven-segment representation of the hex characters b and B is eliminated. All of the characters are also readily distinguished from one another.

In general, points in the suspected area of the fault are probed by a signature analyzer (see Figure 17-1) until a signature is found which does not agree with the one documented in the service manual. The signal path is then traced back until a correct signature is found, localizing the fault. Once the faulty node is found, the bad component on that node becomes apparent or can be located using the current tracer or other techniques. Signatures on digital nodes are used in much the same manner as the voltage and waveform information found on analog circuit service schematics.

Node signatures are meaningful because they are generated by a test stimulus program, provided by either the product under test or an external adapter. This stimulus program exercises specific portions of the circuit in a controlled, repeatable manner.

You may wonder how proper signatures are determined by the manufacturer. The normal procedure is to take a known good product and gather signatures from it, node by node. This approach is taken because it is very difficult to predict the tremendous amount of complex information contained in each signature. Signature gathering is a time-consuming process, and errors can and do occur (check for product errata sheets). If even a single word of ROM is changed because of a

Figure 17-1. 5004A Signature Analyzer

product revision, most or even all of the signatures may need to be respecified in the documentation. Be sure, therefore, that the product series number corresponds to the documentation you are using.

There is no such thing as a signature being "almost" right: it is either right or wrong. Because of the algorithm used by the signature analyzer, the signature 5H22 is no more related to 5H23 than to C7FP.

The signals required by the signature analyzer to generate a signature are DATA, START, STOP, and CLOCK. The DATA input receives the data from the node under test. The START signal, provided by the product under test, tells the signature analyzer when to begin looking at the data, and the STOP signal tells it when to stop. Between the START and STOP signals, data is processed every time a new CLOCK input occurs. The connection points for these inputs must be specified in the service documentation for the product under test.

In the following experiments, you will take signatures on nodes of the μLab in several signature analysis test modes. The relationship of the signatures to portions of the circuit will then be discussed.

SIGNATURE TABLES FOR THE MICROPROCESSOR LAB

The signature tables for the μLab list every IC pin plus the address and data bus lines. These tables can be found in appendix C. In most cases, a signature will be listed for each pin, but there are a number of ways in which the signature may be represented. If the pin is tied directly to ground or +5V, the table entry will simply indicate GND or Vcc rather than giving the signature. The Vcc signature is listed

at the beginning of the table, and the GND signature is always 0000. The table may also show a 1 or 0 signature, which means the same thing as Vcc or GND except that the signal is a gate output, not tied directly to Vcc or GND. Finally, the table may show a Vcc or GND signature with a B after it, which means the same thing as a 1 or 0 except that the light on the signature analyzer's probe tip should be blinking. The B means that although the signal is always at the same logic level when the clock edge arrives, at other times it is at a different level.

EXPERIMENT 17-1

Free-Running the Microprocessor Lab

CONCEPT

Free-running involves letting a minimum portion of a circuit (in this case, the microprocessor) exercise as much of the rest of the circuitry as possible. ROMs, RAMs, and I/O devices need not be working for this to occur. Any feedback path (such as the data bus) to the microprocessor must also be broken so that its operation will not be influenced by any erroneous or unpredictable signals. This technique provides a test stimulus for generating signatures. The advantage of using signature analysis in this mode is that it requires only a very small portion of the circuit to function correctly.

PROCEDURE

I. Setting Up the Free-Run Mode

A) Slide all eight of the data bus switches up (see Figure 17-2). The data bus signal path between the microprocessor and the rest of the system has now been opened. There is no feedback path from the memory to the microprocessor and, therefore, no instructions are sent to it. It is free to run "open-loop."

B) Slide the *free-run* switch (FR) up. It is located to the lower left of the data bus switches. This switch hardwires the MOV A, A instruction (code 7F) to the microprocessor whenever a read operation is performed (see Figure 17-3). The MOV A, A instruction is a "do nothing," or NOP, instruction. The microprocessor will read this instruction and then increment the address to read the next instruction. Since the data bus is disconnected, this instruction appears at every address, which causes the microprocessor to execute this instruction (and only this instruction) repeatedly. Therefore, it continuously increments the address bus lines through all 2^{16} possible addresses.

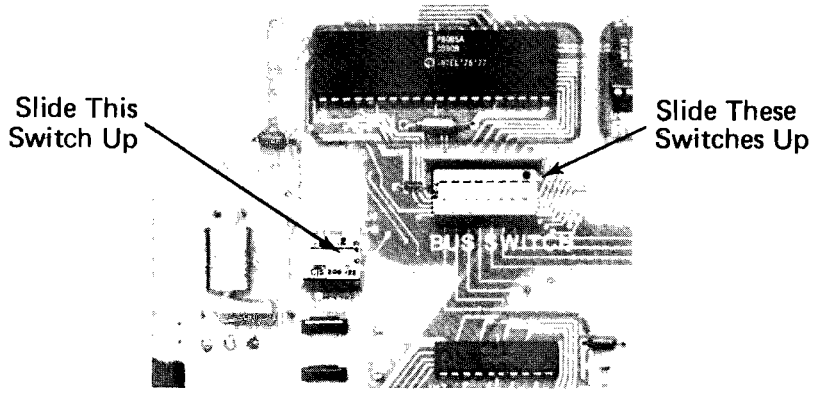


Figure 17-2. Location of Test Switches on μLab

Besides the microprocessor, only a few other circuits in the μLab could prevent the free-run mode from operating. They are the power supply, the crystal, the test switches, and the circuits connected directly to the microprocessor control pins. These circuits, often referred to as the *kernel*, can be a fundamental partitioning point for fault isolation in a microprocessor-based system. If the system does not free-run, the problem is narrowed down to this small number of components.

C) Observe the address bus LED indicators. The visibly flashing A15 and A14 LEDs and the dim A13 to A0 LEDs (flashing too fast to see) indicate that the μLab is continuously cycling through the full address field. Note that the "READ" status LED is bright. This brightness is consistent with the fact that the microprocessor is repeatedly reading the MOV A, A instruction being provided by the 10K pull-up resistors and diode D1. The status LEDs marked ROM, RAM, INPUT, and OUTPUT, which appear to be flashing together, are actually flashing in quick succession as the high order address lines increment through the address space assigned to each device.

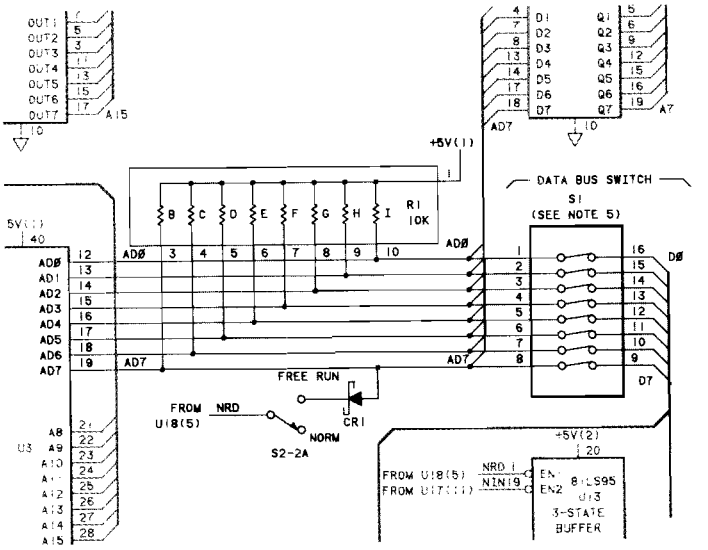


Figure 17-3. Circuit Used to Open Data Bus and Cause μLab to Free-Run

II. Taking Signatures

A) Connect the signature analyzer GND lead to the μLab's ground (⏚) slot, the START and STOP leads to the A15 slot, and the CLOCK lead to the $\overline{\text{READ}}$ slot (see Figure 17-4).

B) Set the START, STOP, and CLOCK switches on the signature analyzer to the rising edge position (push-buttons out). The GATE indicator flashes to indicate that a signature-gathering "window" is occurring. The window is from one rising edge of the A15 address bit to the next

EXPERIMENT 17-1

(Continued)

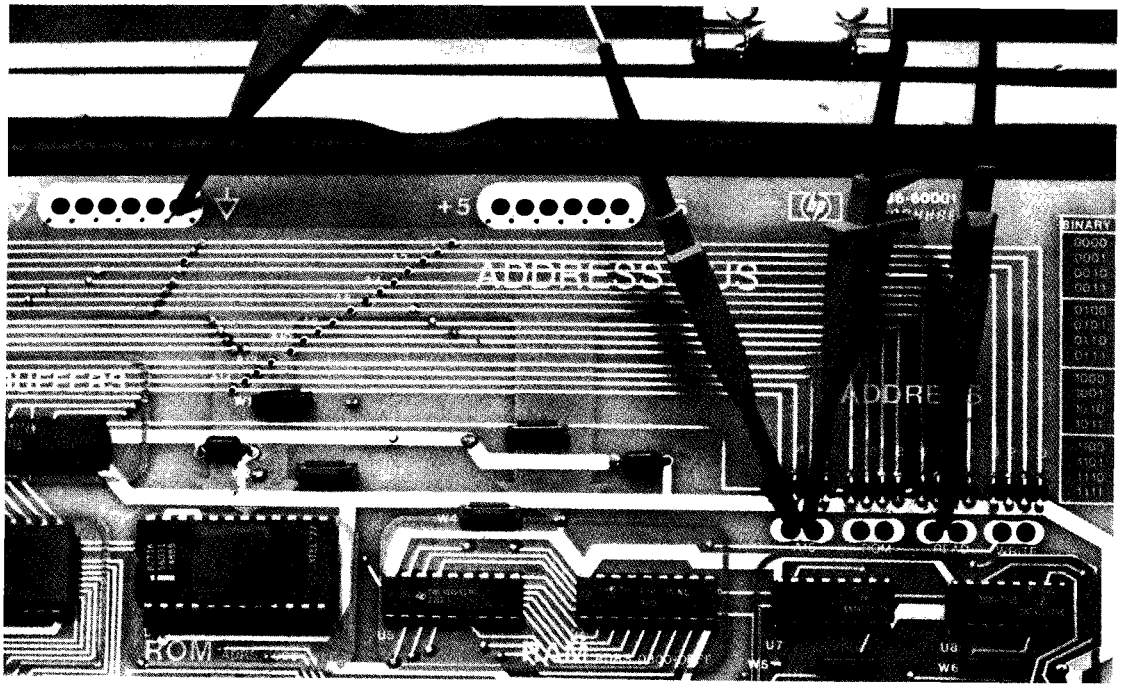


Figure 17-4. Signature Analyzer Connections to μLab for Free-Run Address Test Mode

rising edge (the full 64K address field). Signal data is input to the signature analyzer at every rising edge of the $\overline{\text{READ}}$ line, 64K times during each measurement window (see Figure 17-5).

C) The signature analyzer can now be used to verify a correct set-up for the free-run address mode. Touch the data probe (which also acts as a logic probe) to ground. Observe that the signature is 0000. This is the characteristic logic 0 signature for all test set-ups.

D) Touch the data probe to a Vcc line to take the Vcc signature. Observe the signature 0001. This is the characteristic logic 1 signature for only this set-up of the signature analyzer and the μLab. You can verify that this is true by changing the trigger edge switch on the signature analyzer START or STOP inputs and observing that the signature changes. The 0 0 0 1 display thus indicates that the test set-up is correct and that the kernel is free-running as expected. Had this not been the case, it could be assumed that the set-up was incorrect or that there was a fault in the kernel. This greatly reduces the number of circuits in the product that must be considered as possible fault sources.

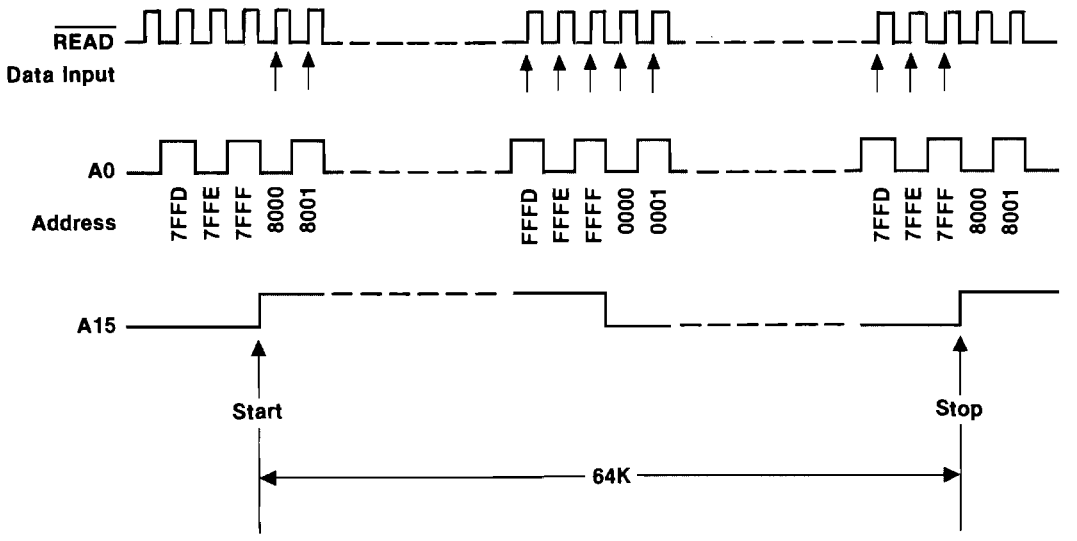


Figure 17-5. Free-Run Address Test Set-Up Samples Data at Each of the 64K Addresses of Measurement Window

E) Referring to Figure 17-6 and Table C-1 in Appendix C, take signatures on address bus lines A0 to A15 and verify that they agree with the entries in the table. This tells you that the microprocessor, the address buffer (IC1), and the address latch (IC2) are operating properly.

F) Referring to Figure 17-7, take signatures on the output pins of the address decoder (IC7 pins 7, 9, 10, 11, 12, 13, 14, 15) and compare them with the entries in the table. These signatures verify that the address decoder is operating properly.

G) All of the signatures in Table C-1 tell you something useful. The reason that a number of the devices have missing signature entries for this test set-up is that they are not "exercised" in the free-run mode. There is either no activity on their outputs (they need a write signal), or the data on them is not predictable (such as the RAM outputs).The signatures on these pins are therefore not useful. Observe these situations as you perform the next three steps.

H) Probe any data bus line and note its signature. This signature is the combined product of all the devices of the μLab that are output to the data bus during the 64K-bit address window.

I) Turn the μLab power switch off and then on. Observe that the signature on this data bus lines has probably changed. It changes because the RAM now has new (random) data stored in it as a result of the power-off cycle. (Since the data bus switches are open, the power-up program that normally fills the memory with zeros cannot run.) Try it again. The signature will probably change again.

(Continued)

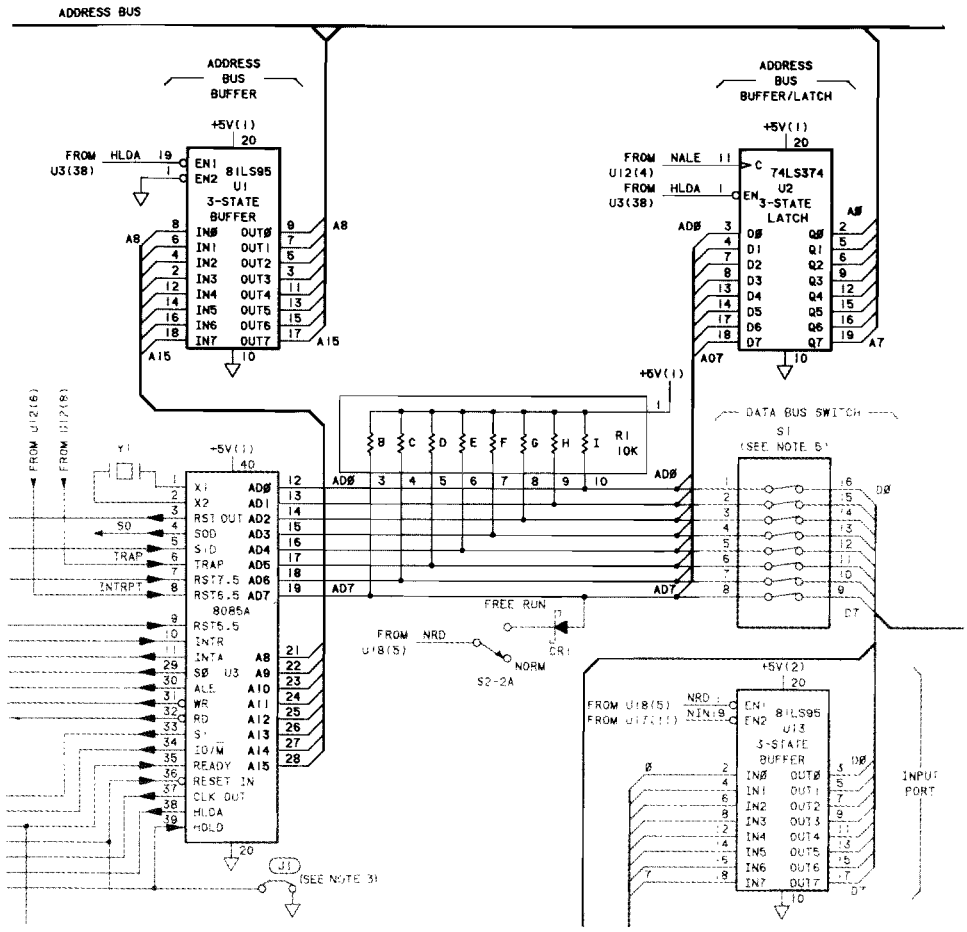


Figure 17-6. Address Demultiplexing and Drive Circuit

J) With the probe still on the data bus line, slide the input port switch (not the data bus switch) corresponding to that bus line to its opposite position. The input switch is located below and to the right of the display and is connected to input port IC13. Observe that the signature changes depending on the position of this switch. Try it on other bus lines as well. It is important to note that some circuit activity can be verified at this input port, though it is not necessarily the correct activity. Later, other test set-ups will provide more information.

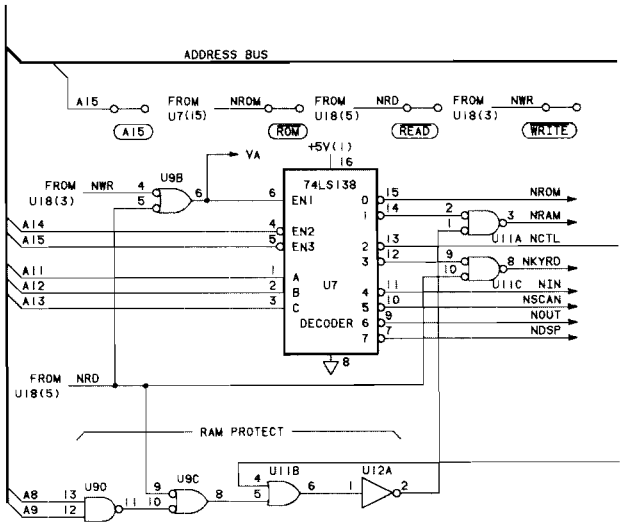


Figure 17-7. Address Decoder Circuit

SUMMARY

You placed the μLab in a free-run test mode in which the microprocessor scanned continuously through the entire 64K memory address field. The signature analyzer set-up was verified by the proper Vcc signature. The address buffers and decoders were checked by taking the signatures on their outputs.

In the free-run test mode, the path between the data bus and the microprocessor was broken. This isolation allowed all of the devices that talk to the microprocessor (ROM, RAM, and input ports) to be read, while the address bus was incrementing, without affecting the operation of the microprocessor. In this manner, the effects of some of these devices could be seen on the data bus signatures.

EXPERIMENT 17-2

Checking the ROM While Free-Running

CONCEPT

In the free-run mode, the microprocesor reads every possible address (all 64K). When the signature analyzer is gated to look at data on the data bus only while the ROM is addressed (address 0000 to 07FF), signatures are generated that depend only on the data in the ROM. The 2K words of the ROM are verified by checking the signature on each of the eight data bus lines.

PROCEDURE

A) Verify that the μLab is still in the free-run mode (refer to Experiment 17-1). Connect the signature analyzer's START and STOP connectors to the slot marked $\overline{\text{ROM}}$. Set the START input to the falling edge position (button in) and leave the STOP edge rising (button out).The measurement window now begins the first time the ROM is read (when the ROM enable line goes low at address 0000) and ends just after the last ROM address is read (when the ROM enable goes high at address 0800). Although this measurement window is only 2K addresses long, the microprocessor still runs in the 64K long free-run loop. The signature analyzer, however, is now looking at data for only the 2K cycles of the loop during which the ROM is enabled (see Figure 17-8).

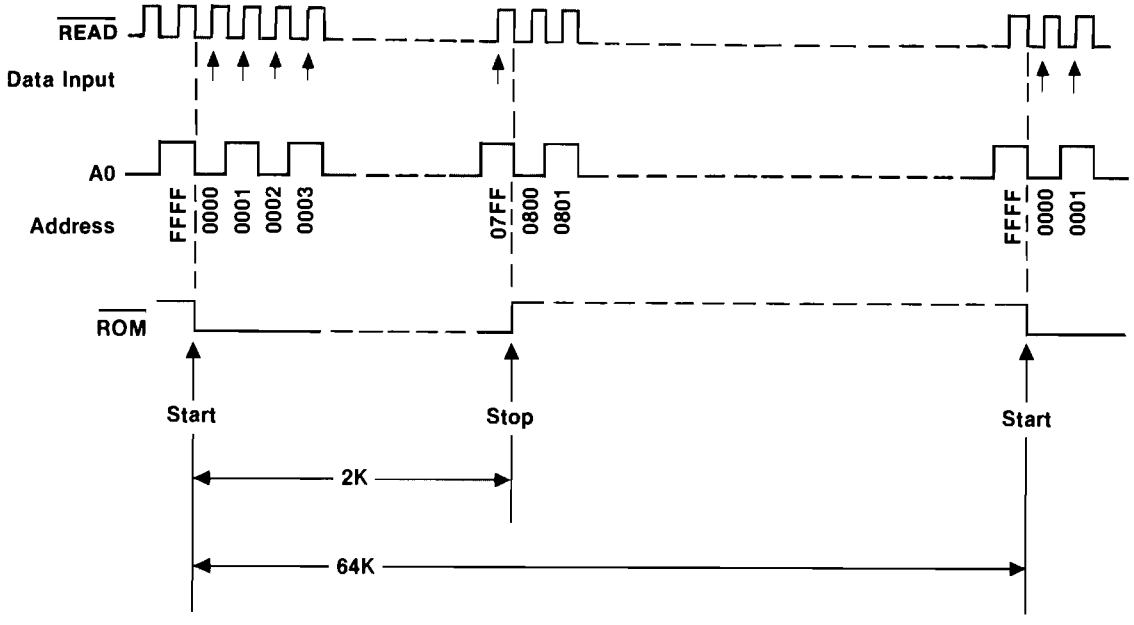


Figure 17-8. Free-Run ROM Test Set-Up Samples Data Only During 2K Addresses that ROM is Being Read

B) Touch the data probe to Vcc and verify from Table C-2 that the proper signature (7A70) is present for the free-run ROM test mode.

C) Verify that signatures on each of the eight data bus lines agree with the ones in the table. This test verifies all 16K (2K × 8) bits of the ROM.

D) Try changing the input switches and turning the power off and then on to scramble the RAM data. The data bus signatures do not change. The other devices talking to the data bus are still talking, but they are now outside the measurement window (as defined by START and STOP). This is in contrast with the previous experiment in which the measurement window included all addresses and therefore all devices.

SUMMARY

This test provides a great deal of information. You have verified the ROM and can assume it is good to a very high level of confidence. It is also fairly certain that no bus conflicts (two or more devices talking to the bus at the same time) occurred while the ROM was accessed (wrong signatures would have resulted). The data bus is not stuck or shorted and has no serious defects (solder or gold bridges, device outputs stuck on, etc.).

All of the diagnostic information observed thus far was obtained by free-running the system. No ROM code was executed (i.e., no program was run). The μLab used only a small portion of its circuits (the kernel) running open-loop to provide the test stimulus (incrementing address lines and the pulsing read line) for many other devices. This information (in the form of correct signatures) allows you to continue to the next stage of testing with confidence in these circuits.

EXPERIMENT 17-3

The SA Test Loop

CONCEPT

The SA test loop is a program permanently stored in the μLab's ROM for the sole purpose of providing a controlled stimulus for signature analysis. The signature analysis switch (SA) is connected to an interrupt line on the microprocessor. The interrupt service routine is the SA test loop. The SA switch, therefore, puts the μLab into this exercise mode, and the RESET key returns it to normal operation. This test mode also turns on all display segments and output port LEDs so that they can be visually tested. They can also be used to indicate that the μLab is running the SA test loop program.

This loop provides stimulus signals to devices that the microprocessor talks to (the output port, the scan and display ports, the speaker, and the control logic.) The loop also reads the devices that can talk to the microprocessor (the input port switches and the keyboard). Circuit responses to the stimulus program (a repeating sequence of simple programs in the ROM to exercise the individual devices) is monitored by the signature analyzer.

A simple program will serve to illustrate the nature of this SA test loop stimulus. The following is the sequence of events performed during the output port test:

1. Turn on all LEDs but D0.
2. Turn on all LEDs but D1.

•
•
•

8. Turn on all LEDs but D7.
9. Turn on all LEDs.
10. Continue with the next stimulus program in the SA test loop.

As you can see from this simple program (which is typical of most of the stimulus routines), it merely exercises one device at a time (in this case, the output port IC15).

PROCEDURE

I. Setting Up the SA Test Loop

A) Slide the free-run switch (FR) down to its normal position to remove the forced free-run instruction from the microprocessor.

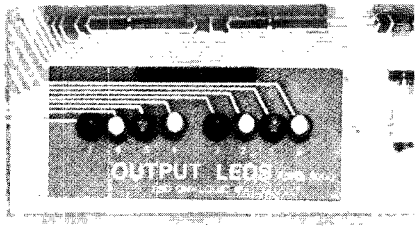
B) Slide all eight data bus switches down to reconnect the microprocessor data bus to the rest of the system (closed-loop). The μLab has now been restored to its normal operating configuration.

C) Press RESET twice. The display shows μLAb UP . The system is now running. The output LEDs are in unpredictable logic states at this point because of the opening and closing of the data bus.

D) Slide the SA switch (located to the left of the FR switch) up and then down. Before doing so, be sure that the display shows μLAb UP to insure that the SA switch interrupt line is enabled by the microprocessor. A beep sounds as you move it to the *up* position. Observe that the display segments and output LEDs are lit and a faint crackle can be heard coming from the speaker. Display segment intensity may not be uniform in this mode. The μLab is now in the SA test loop.

II. Taking Signatures in the SA Write Test Mode

A) Touch the signature analyzer's data probe to any one of the output LED signal lines (see Figure 17-9). The blinking tip shows pulse activity even though the LEDs appear to be fully on. The reason for this activity is that the SA test routine is driving the LEDs on most of the time and off occasionally. This routine allows both output logic states of IC15 to be checked with the signature analyzer.



Computer Museum

Figure 17-9. Output LEDs Exercised During SA Test Loop

B) Connect both the START and STOP leads of the signature analyzer to the A15 test slot. Connect the CLOCK lead to the $\overline{\text{WRITE}}$ test slot. Be sure that the GND lead is still connected to ground.

C) Set the START, STOP, and CLOCK inputs on the signature analyzer to rising edges (button out). The A15 bit is controlled by the SA test loop program and pulses high once during each loop cycle. The program generates this pulse by writing to address 8000 once at the beginning of this test loop. The signature analyzer is clocked off the μLab $\overline{\text{WRITE}}$ line and thus looks at data every time a write operation occurs.

D) Verify the SA write test set-up by checking the Vcc signature (refer to Table C-3).

(Continued)

E) Using Table C-3, check signatures on any nodes of interest. Notice that input devices (which are enabled by the $\overline{\text{READ}}$ signal) are not checked in this test mode.

F) While probing the various data bus lines, slide any of the input port switches and press any of the keyboard switches (except RESET). Verify that they have no effect on the signature. These switches have no effect because they are enabled by the $\overline{\text{READ}}$ signal, but the signature analyzer is clocked by the $\overline{\text{WRITE}}$ signal.

III. The SA Read Test Mode

The SA test loop provides two types of stimuli. In the previous part of this experiment, the test loop was used to write data from the microprocessor to the memory and I/O devices. The SA test loop also reads data from these devices into the microprocessor. The signature analyzer's CLOCK input connection determines whether the signatures taken are dependent on read or write data. The $\overline{\text{READ}}$ and the $\overline{\text{WRITE}}$ test slots are used for this purpose.

A) Move the signature analyzer's CLOCK lead from the $\overline{\text{WRITE}}$ slot to the $\overline{\text{READ}}$ slot on the μLab. Verify that the START, STOP, and CLOCK inputs on the signature analyzer are still set to rising edges and that the START and STOP leads are still connected to the A15 test slot. Now you can take signatures on the devices that the microprocessor reads.

B) Touch the data probe tip to Vcc and check the Vcc signature in Table C-4. This set-up may seem familiar to you because it has the same set of connections and edge settings as the free-run address mode. The Vcc signatures are different in this mode because there is a difference in the program cycle length. In the free-run case, A15 defined a cycle length of 64K successive addresses. In the SA test loop, the SA stimulus program controls the A15 line for use as a START and STOP control. The window length in this case is much less than 64K cycles because the test loop is much shorter.

C) Slide all of the input switches down so that logic 0's are input to all data bus lines when the input port is read. Check the signatures on the data bus lines against the entries in Table C-4.

D) Slide all of the input switches back up. The signatures on each data bus line will change as the corresponding input switch is changed. These signatures will agree with those in the table for this set-up and verify that the input switches are being read correctly onto the data bus.

E) Refer to Figure 17-10 and Table C-4. Verify that the keyboard switches in the table work (avoid the RESET key) by pressing them and observing the signatures on the corresponding data line inputs on IC18 and the data bus lines.

F) Press INTRPT and verify that the speaker beeps repeatedly. The SA stimulus program provides this audible test function.

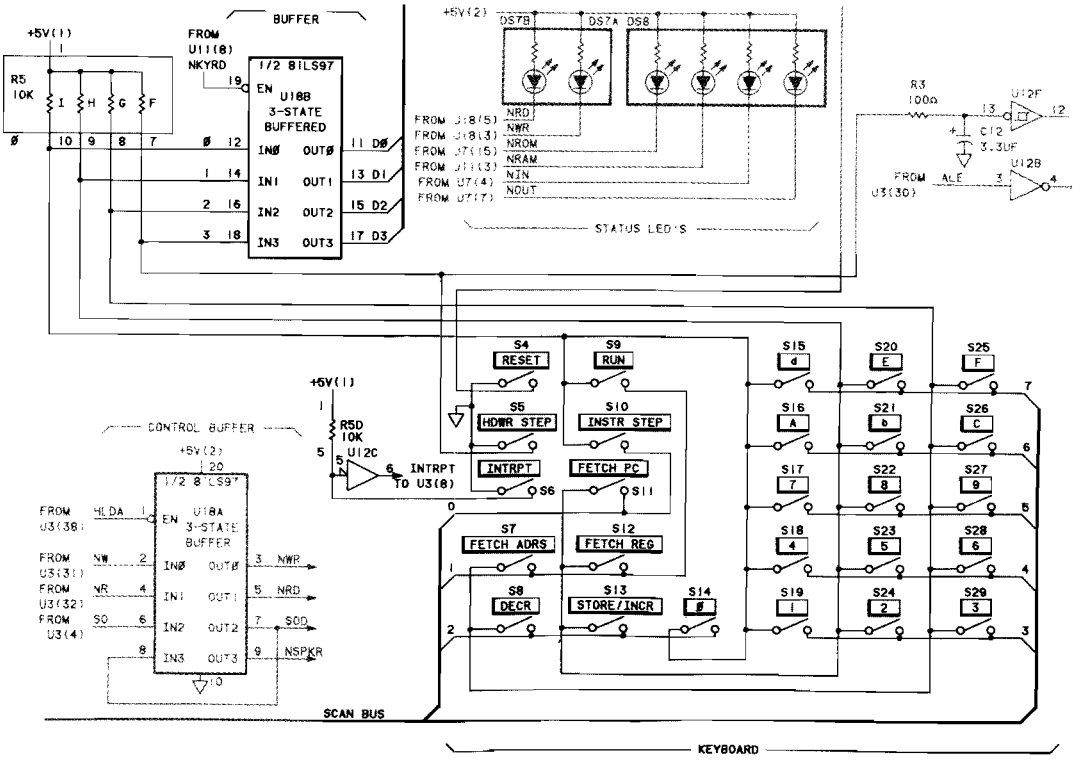


Figure 17-10. Keyboard Input Circuit

IV. Finding a Simple Fault

A) Locate fault jumper 10 (marked W10) near the center of the μLab board (see Figure 17-11). Remove the jumper and observe that output LED 4 goes out.

B) Referring to Figure 17-12 and signature Table C-3, check the signature on LED 4. (Be sure to change the test set-up as specified in Table C-3.) The bad signature at this point means that the LED itself is probably not at fault but that it is simply not receiving the correct signal.

C) Trace the signal on the schematic back to output port IC15-12. Observe that this output pin also has the same bad signature. You have not yet found the fault.

EXPERIMENT 17-3

(Continued)

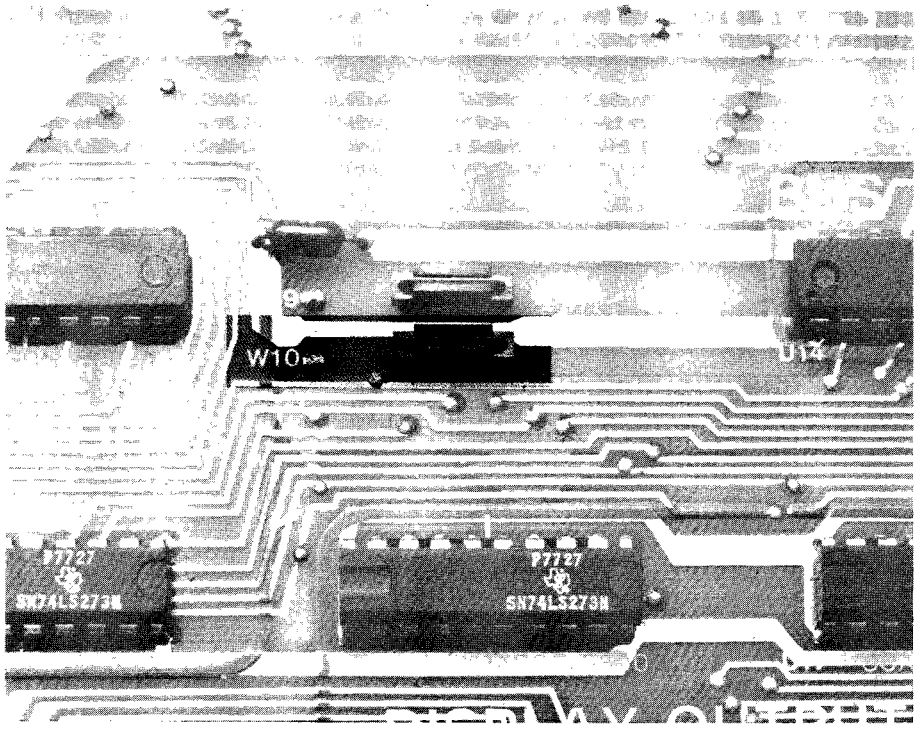


Figure 17-11. Location of Fault Jumper W10

D) Probe the D4 input to the output port (IC15-13). This signature is also bad.

E) Now probe the bus buffer output for D4 (IC14-11). This signature is good. Therefore the fault must lie between this output and IC15-3. You have now isolated the fault to one PC trace.

F) Probe the signal along the trace going from IC15-13 toward IC14-11. You have now found the fault.

G) Restore fault jumper 10 between the right two holes of fault location W10.

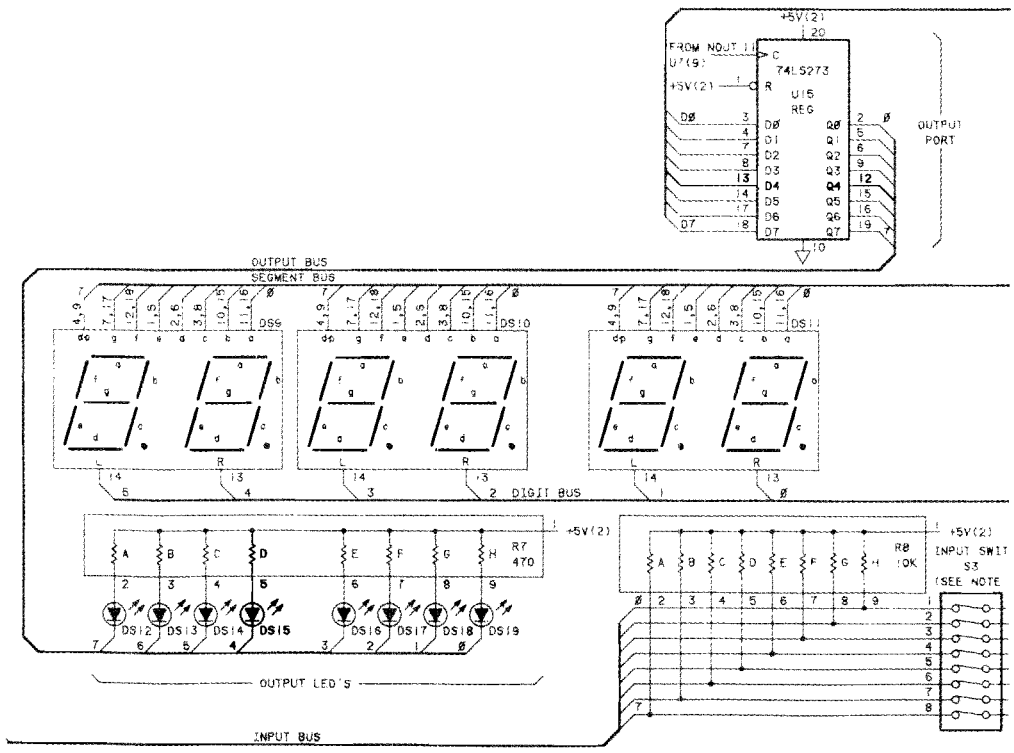


Figure 17-12. Output Port Circuit

SUMMARY

In this experiment you learned how to use the signature analyzer to verify microprocessor system performance. Fault isolation techniques incorporating a free-run test mode and a special SA test loop program were discussed. With these techniques, only a small portion of the overall circuit (the kernel) is required to function in order to test some of the other key system circuits (including the ROM). Once the kernel was verified, the SA test loop program was used to exercise devices not tested by free-running. You saw how the signature analyzer, combined with a general understanding of the operation of the μLab's devices (and some troubleshooting common sense), could lead you to a faulty node with a minimum of hardware manipulation and no software manipulation whatsoever.

LOGIC ANALYZERS

Logic analyzers are specialied instruments used for examining a number of logic signals (usually 16 or 32) during very specific portions of the long, complex data sequences present in microprocessor systems. This section covers, in general terms, what logic analyzers are and how they are used. Although they are primarily considered to be microprocessor system development and system-level troubleshooting instruments, applications do exist in product service situations.

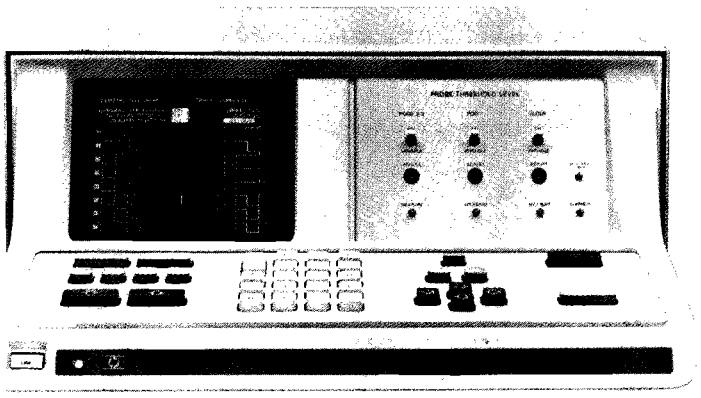


Figure 17-13a shows a logic analyzer display that represents address (column A) and data (column B) information in a tabular form. It allows you to follow the sequence of events beginning with the instruction at address 06A2. Use the following table to follow the program flow of this sequence:

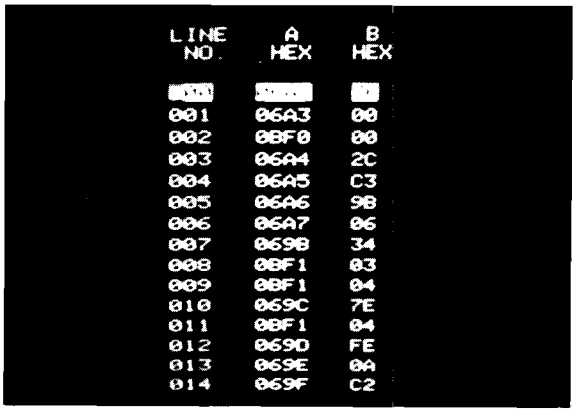
Line Number	Operation		Comments
000,1	MVI M,00	Instruction	
002	M←00	Execution	M=location 0BF0
003	INR L	Instruction	point to next M
004,5,6	JMP 069B	Instruction	
007	INR M	Instruction	
008	μP←M	Execution	M=location 0BF1
009	M←M+1	Execution	
010	MOV A,M	Instruction	
011	A←M	Execution	M=location 0BF1
012, 13	CPI 0A	Instruction	
014	JNZ	Instruction	

These 16 lines on the display represent a small portion of a long program sequence. The logic analyzer "captures" this sequence, stores it in its internal memory, and can display it indefinitely. The logic analyzer also allows you to look at activity before or after the operations shown.

A logic analyzer is similar to an oscilloscope in many ways: it has signal inputs, a trigger, timing circuitry, and a CRT (cathode ray tube) display. Its signal inputs differ from those of an oscilloscope in that it has 16 to 32 logic threshold-sensitive inputs that detect logic levels (1's and 0's).

The triggering circuit is much more sophisticated than that of an oscilloscope. Analyzers can trigger on a particular address, on the Nth occurrence of that address (for software loops), after N clock cycles past that address (for delay), or

after a specified sequence of trigger addresses (to detect a particular program path). The trigger words need not be addresses. They can be data, control, or any combination of logic signals.



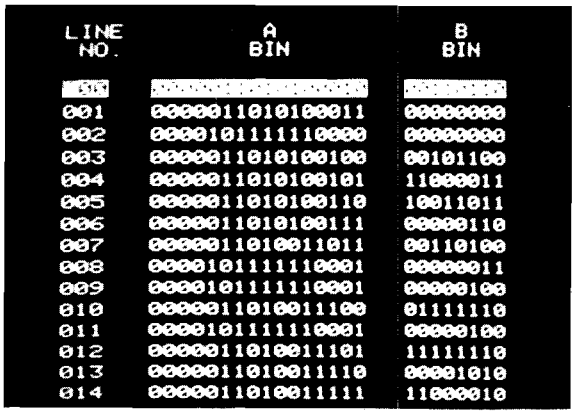
LINE NO.	A HEX	B HEX
001	06A3	00
002	0BF0	00
003	06A4	2C
004	06A5	C3
005	06A6	9B
006	06A7	06
007	069B	34
008	0BF1	03
009	0BF1	04
010	069C	7E
011	0BF1	04
012	069D	FE
013	069E	0A
014	069F	C2

Figure 17-13a. Logic Analyzer Used to Display Hex Address and Data in Tabular Form So That Execution of Program Sequence Can Be Followed

Like oscilloscopes, logic analyzers can display the data that occurs immediately after the trigger event. Unlike most scopes, logic analyzers can also display data prior to the trigger event. This capability, called *negative time* recording, can be used for troubleshooting by choosing a faulty system operation to be the trigger event and then observing the events that led up to it.

For timing, logic state analyzers use the clock of the circuit they are connected to. A new line of display information is generated for each clock input. While oscilloscopes are referred to as "time domain" instruments, logic analyzers are often called "data domain" instruments.

In an oscilloscope, time advances from left to right across the display, using an internal time base. In a logic analyzer, time advances from top to bottom, one line per clock input. The display of the logic analyzer is a tabular listing of digital data. In Figure 17-13a it is formatted in hex numerals, but it can also be displayed in binary, octal, or decimal. Figure 17-13b shows how the program sequence of



LINE NO.	A BIN	B BIN
001	0000011010100011	00000000
002	0000101111110000	00000000
003	0000011010100100	00101100
004	0000011010100101	11000011
005	0000011010100110	10011011
006	0000011010100111	00000110
007	0000011010011011	00110100
008	0000101111110001	00000011
009	0000101111110001	00000100
010	0000011010011100	01111110
011	0000101111110001	00000100
012	0000011010011101	11111110
013	0000011010011110	00001010
014	0000011010011111	11000010

Figure 17-13b. Figure 17-13a Formatted in Binary

Figure 17-13a looks in binary. No analog voltage waveforms or values are displayed.

Some logic analyzers have a feature called *timing analysis*. With the use of an internal time base, timing diagrams of the logic signal inputs can be displayed on the CRT in a manner similar to an oscilloscope (see Figure 17-14). The difference is that the display is "digitized" (or rounded-off) to discrete time and voltage limits to give a clear graphic display without specific waveshape information. Also, glitches (narrow pulses) can be detected by means of internal detection and stretching circuits. Many inputs can be observed at once. By taking full advantage of the triggering features, you can observe the timing of a small, specific portion of a long program sequence. The elaborate triggering is made possible by the fact that the waveforms are stored in digital form in an internal RAM.

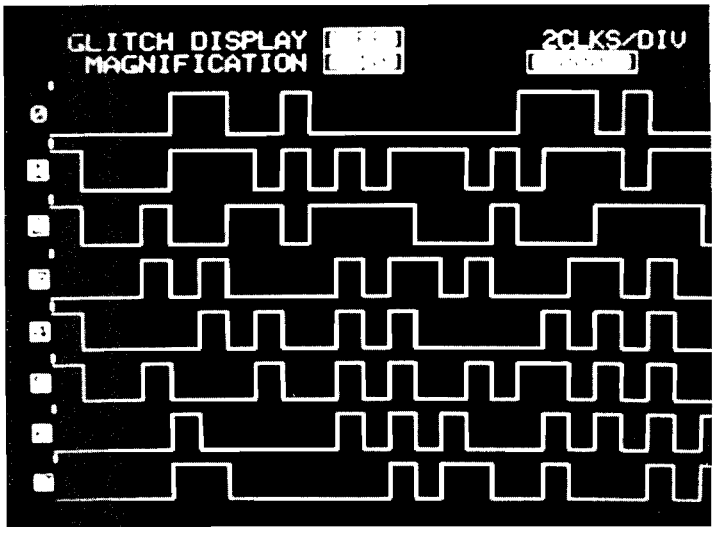


Figure 17-14. Eight Data Bus Lines Displayed in Timing Analysis Mode

Another feature of many logic analyzers is the *map mode*, a useful feature for product troubleshooting. In this mode, the CRT is used as an X-Y grid of up to 256 × 256 (65,536 or 2^{16}) points. One point can be used to represent each address in a 64K memory space. When a microprocessor system is running a program, one of the 64K points on the CRT is lit each time the microprocessor specifies an address. The more often it points to that address, the brighter the point becomes. As the program executes sequential instructions, successive points are illuminated. When jumps occur in the program, they can be observed on the CRT as a jump from one cluster of points to another. To improve resolution, the display can be expanded to magnify a region of interest. Figure 17-15 shows an 8085 program loop displayed in the expanded mode. It begins at address 0800. When it gets to address 0808, it stores a byte of data at address 3000. At address 0810 it jumps to address 0F20 to continue the program. Address 0F20 contains an instruction that causes a byte of data to be stored at address 3838. Finally, a jump instruction at 0F24 causes the program to go back to address 0800 and repeat the loop.

The map display shows where the system is spending its time (i.e., which subroutine and peripheral addresses it is using). There are several variations and enhancements to this basic map mode that can be found on specific logic analyzers.

Contrast Figure 17-15 with Figure 17-16 (free-running). In this free-running mode, the microprocessor spends equal time at all addressses and eventually causes all address points on the CRT to light. Figure 17-15 shows clear program flow, whereas Figure 17-16 shows none. The map is useful for checking a microprocessor system that shows bus activity but is not functioning correctly. The map of a good product can also be compared with that of a bad one as an aid in identifying possible problem areas.

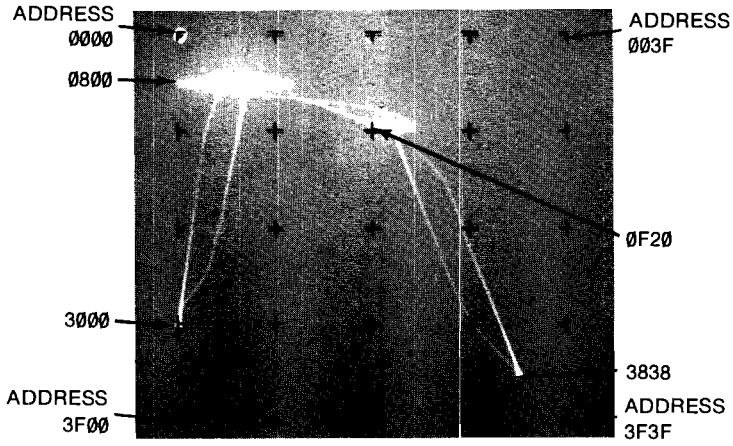


Figure 17-15. Logic Analyzer in Expanded Map Mode Used to Indicate Address Activity

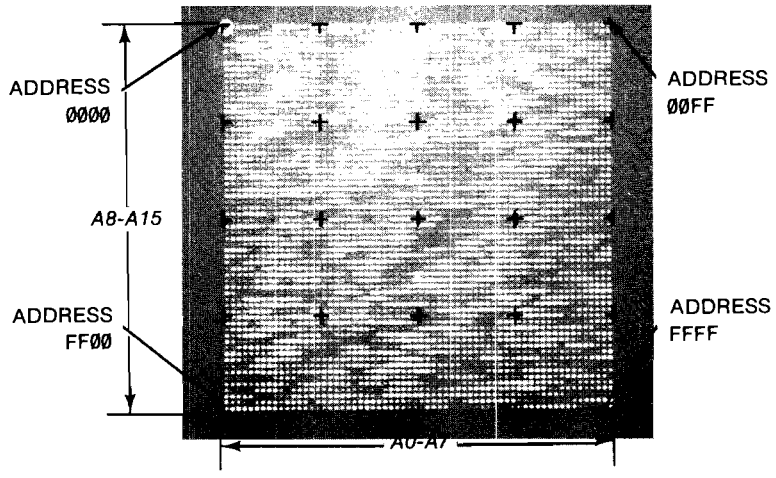


Figure 17-16. Map Display for Free-Run Mode Showing Equal Activity at All 64K Addresses

Logic analyzers are useful for system software development because of their ability to trigger on and then follow specific program and timing sequences in an operating breadboard. They can provide a map of where the microprocessor system spends its time. These same capabilities can be used to analyze existing products in production. When properly set up, program sequences can be observed in the product being tested. Good service documentation is generally required to describe proper set-ups, display outputs, and courses of action to be taken when incorrect program flow activity is observed.

REVIEW

Lesson 17

The constant flow of signals over the buses of a microprocessor system generates a tremendous amount of information. This information must either be reduced or compressed (e.g., signature analysis) or broken up into smaller portions (e.g., logic analysis) if it is to make sense.

Signature analyzers reduce the complex bit streams present in microprocessor systems into an easy-to-read "signature." These signatures then tell you whether or not the node examined is operating as it should. Once a bad node is found, back-tracing the circuits leads to the fault source. Signature analysis uses the product under test to provide its own test stimulus. For this reason, the signature analyzer can best be used on products that have this test feature designed in (or added on). It is very important to realize that signature analysis cannot be used with all microprocessor systems.

Logic analyzers can be thought of as very specialized digital oscilloscopes. They allow you to examine specific portions of the hardware and software sequences that occur in microprocessor systems. They can be especially useful as design aids for troubleshooting new product designs. The map mode feature lends itself well to troubleshooting existing products. Logic analyzers require a greater understanding of hardware and software program flow than do signature analyzers. Unlike signature analyzers, however, they do not need to be designed into a product to be useful, and they provide much more detailed information about the system.

QUIZ

Lesson 17

1. In the free-run test mode, circuit stimulus patterns are generated by:
   a. a program stored in the ROM.
   b. a program stored in the RAM.
   c. random action of the circuits.
   d. sequential cycling of the microprocessor.

2. Signatures can:
   a. provide information about the nature of the fault.
   b. tell whether a node is operating correctly or not.
   c. verify the test set-up.
   d. do all of the above.

3. A correct Vcc signature tells you that:
   a. the signature analyzer test set-up is probably correct.
   b. the test loop is running correctly.
   c. there is a logic "1" level on the Vcc line.
   d. all of the above are true.

4. Data is processed by the signature analyzer:
   a. at every microprocessor clock cycle.
   b. at every signature analyzer input clock cycle.
   c. only between START and STOP inputs.
   d. at all of the above times.

5. A bad signature on a logic node always means that a fault is:
   a. on that node.
   b. on that node or before it.
   c. on that node or after it.
   d. a stuck node.

6. In a product that has SA designed in, test modes can be selected by:
   a. switches and jumpers
   b. signature analyzer input corrections.
   c. signature analyzer switch settings.
   d. all of the above.

7. The reason that logic analyzers can capture and display selected program sequences better than oscilloscopes is that they have internal memory and:
   a. multiline sequential trigger circuits.
   b. fast internal clocks.
   c. high resolution CRTs.
   d. all of the above.

8. The map mode lets you see:
   a. physical component location.
   b. at what addresses the system spends its time.
   c. what data is located at a particular address.
   d. a partial listing of the program.

LESSON 18

Troubleshooting Microprocessor Systems

INTRODUCTION

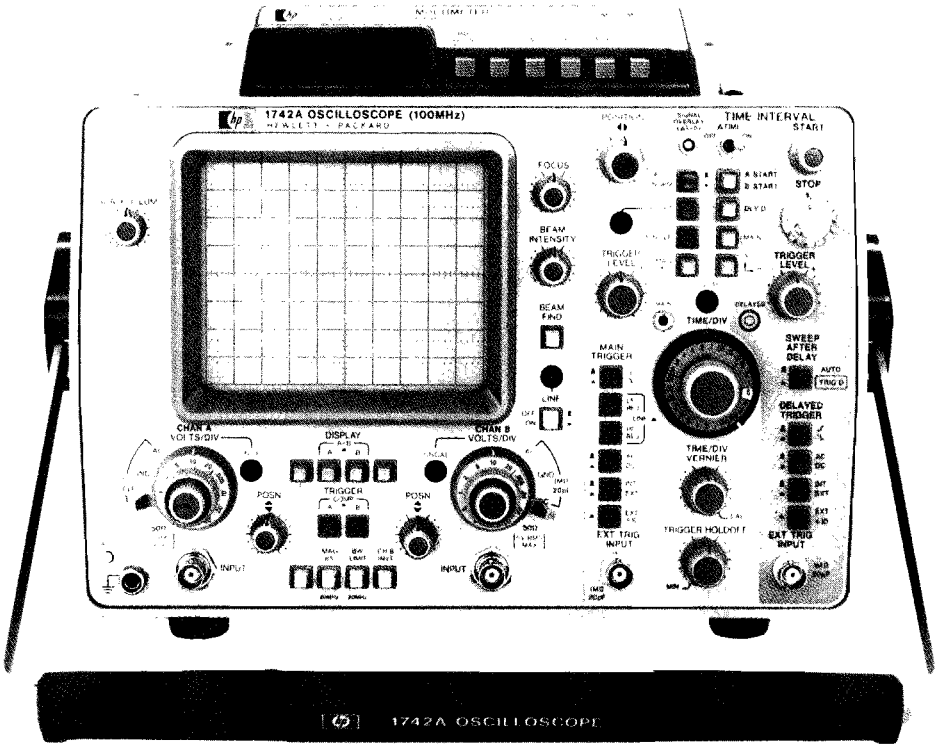
The troubleshooting philosophy for microprocessor-based products is fundamentally no different than for standard digital designs. As with any circuit you are trying to analyze or troubleshoot, it is helpful to first become familiar with the circuit. Studying the theory of operation, the block diagram, and the schematic provides a base of knowledge from which to work. In this lesson, problems relating to microprocessor systems and the troubleshooting techniques for dealing with them are discussed.

MICROPROCESSOR TROUBLESHOOTING PROBLEMS

There are a number of testing problems somewhat unique to microprocessor systems. For one thing, most of the control is in the software, so that signal flow is hard to trace. Another difficulty is that everything happens too rapidly to see in real time. In most cases, a microprocessor system, unlike many logic circuits, cannot be stopped and manipulated. Measurements must be taken while the microprocessor is running. This requirement reduces the effectiveness of the logic probe and pulser but enhances the usefulness of the current tracer, oscilloscope, signature analyzer, and logic analyzer because these instruments rely on circuit activity for their measurements.

Microprocessor bus structures pose additional difficulties. Data on these buses is often unstable or meaningless because of three-state outputs, multiplexing, and switching transients. These conditions cause no problems for the system itself, since it is synchronous and knows when the bus lines contain stable signals. The signature analyzer and the logic analyzer also know when these lines are valid, because of clock signals provided to them. The oscilloscope does not have this capability. It provides little quantitative information, but is useful for examining qualitative factors, such as general activity, logic levels, waveform timing, and bus conflicts.

Since bus structures also make it possible for many devices to be connected together on a single node, finding the one bad device on such a node can be difficult. The current tracer is useful for this purpose. The data bus also acts as a digital signal feedback path and tends to propagate errors through good



Oscilloscope Helps Identify Problems in Microprocessor Systems

circuits and then back to the fault source. The best way to deal with this problem is to open the feedback path when possible. Techniques for doing so are discussed in this lesson.

Complex devices are often connected to the microprocessor buses. It is difficult to test these devices using simple stimulus-response testing. The correct operation of these devices can be verified by swapping them with a known good chip, or by observing that the function they perform for the system is being performed correctly.

Microprocessors are sequential machines. Program flow depends on a long sequence of instructions and events. If even a single bit of information is incorrect, the whole system can go awry. Noise glitches and bad memory bits are the most common sources of single-bit errors. Others are also discussed in this lesson. These failures are difficult to pinpoint because the entire system may appear to be operating incorrectly.

Experience gained from doing the Microprocessor Lab troubleshooting experiments in Lesson 19 will provide you with a good foundation for troubleshooting other microprocessor-based products. Such experience can prevent the really difficult troubleshooting problems from being thrown under your bench (or worse). New things always seem more difficult at first, and the same is true of microprocessors. Designed-in serviceability and good documentation by the manufacturer can make troubleshooting much easier. The use of signature analysis and other high-level servicing aids can greatly reduce the task of troubleshooting.

Dozens of different microprocessors exist, and hundreds of people design products and service procedures for them. Since the μLab is specifically designed for educational purposes and for teaching troubleshooting, the concepts developed using the μLab should be applicable to many classes of microprocessor systems. It is as close to a typical (but small) system as is practical.

PROBLEMS SPECIFIC TO MICROPROCESSOR SYSTEMS

Clocks

Bad clocks can cause fouled, but "running," systems. There are a number of malfunctions that can result in system clocking problems. Clock problems can show up as a failure of the system to function at all (no activity), the ability to function only open-loop (free-running), or semifunctional activity (a meaningless and undefined program sequence). Some microprocessors are sensitive to clock speed. Since many systems run "at spec," even a small variation in clock rate (too fast) can cause system failures. If the system runs too slow, dynamic storage cells on ICs in the system may fail. Both of these problems are more likely to occur when resistor and capacitor (RC) clock circuits are used instead of the more accurate and stable crystal-controlled circuits. However, crystals can sometimes break into their third overtone oscillation mode, causing a much higher than expected clock rate. In addition, some processors require multiphase and nonoverlapping clocks with very stringent timing requirements. Also, clock voltage levels are not necessarily TTL compatible, but may be much wider in voltage swing. Microprocessor clock specs can be found on the device data sheets and can be checked using conventional frequency counters and oscilloscopes.

Power-Up Reset

The microprocessor's power-up reset circuit can also cause fouled, but running, system operation. A reset pulse that is nonexistent, too short, too noisy, or too slow in transition can start everything off on the wrong foot, resulting in out-of-sequence, partial, or no reset activity. Problems can also occur in reset circuits that are susceptible to power supply glitches. Even when Schmitt input circuits are used, slow edges can cause reset timing skew from one device to another within some systems. This will cause some of the devices to power-up before the others, resulting in erroneous behavior. A too rapid ON-OFF-ON system power sequence will fail to restart many systems (e.g., the μLab). It may then be necessary to increase the OFF time to allow the power supplies and restart circuits to discharge.

None of these reset failures will necessarily prevent the system from running. It may run for a short time and then stop, or lock up in a meaningless program loop, or even perform most of its normal operations. The key point to remember is that the system must complete the power-up reset sequence to insure that all of the test, control, and initialization operations necessary to bring the system up have been performed.

Power-up reset circuits are normally operative only when the system is initially powered-up. They can be monitored at that time with storage oscilloscopes, logic analyzers, and in some cases, signature analyzers. They can also be manually overdriven and controlled externally for testing purposes.

Interrupts

Stuck or noisy interrupt lines can cause faulty system operation. The system may work with a stuck line but it will do so very slowly (spending most of its time servicing the "phantom" interrupt). Noisy interrupt lines can cause sporadic system changes to occur, or peripheral inputs or outputs may take place at improper times. Sometimes the system will not respond at all to certain I/O devices, which can occur when a higher priority interrupt has disabled the lower ones.

Interrupt line activity can be monitored with a logic probe, logic analyzer, or oscilloscope. Interrupts are asynchronous in nature and can often be manually controlled (enabled or disabled) for testing purposes.

Signal Degradation

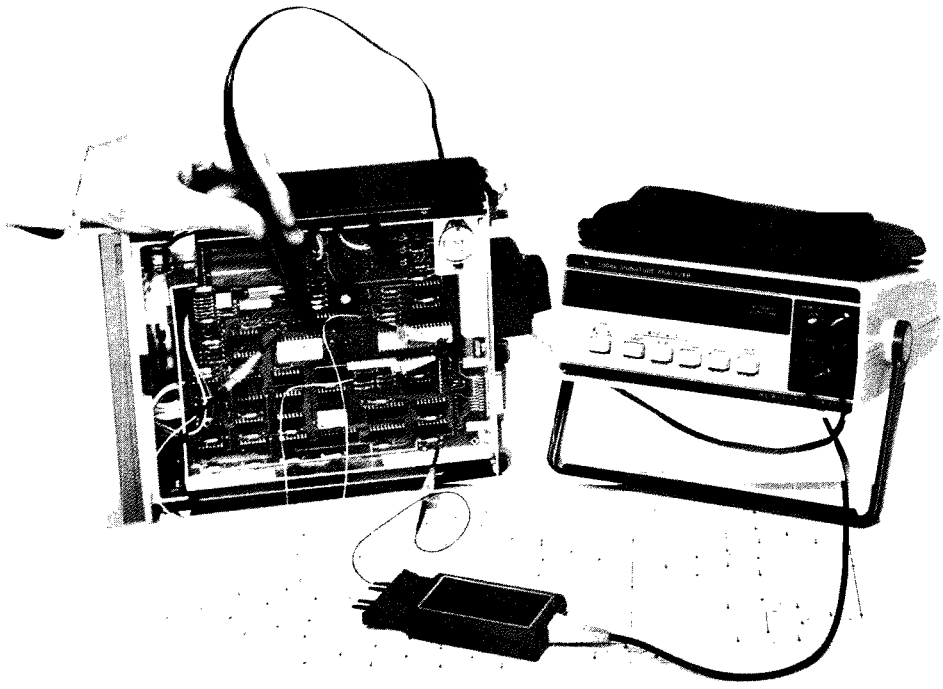
The long parallel bus and control lines present in medium-to-large microprocessor systems are sometimes susceptible to crosstalk and transmission line problems on critical lines (such as clocks and enables). These problems can show up as glitches on adjacent signal lines or ringing on the driving line (causing multiple transitions through a logic threshold). Either of these situations can inject faulty data or control signals that are very difficult to detect. This problem is most common when signal lines are long and already taxing the timing and noise margins of the system. When extender cards are added to these systems or high-humidity conditions exist, failures may occur. Cross-coupling of lines on extender cards can be a problem when fast signal transition lines (such as Schottky gate outputs) run alongside other signal lines, even when they are on opposite sides of a PC board.

Memories

Memory failures in microprocessor systems can produce deviant system behavior in a number of ways. Anything from a total system failure to a single faulty bit of stored data can occur. Most memory failures can be found during the power-up self-test program, unless the memory failure prevents this program from running. If the system doesn't do a RAM verification test and no RAM test service fixtures or procedures are provided, it is nearly impossible to test the RAM. You will probably need to resort to substitution techniques when a RAM becomes suspect.

RAM failures occuring in the area of the memory used for the stack will usually cause the system to crash, even for a single-bit error. Otherwise, RAM failures may cause soft errors that result in unreliable system operation. Faulty dynamic RAM refresh circuitry is another factor to consider in diagnosing apparent RAM failures.

ROMs can also fail. Such failures are more frequent when nonmask programmable types are used. A single bad bit could crash the system or, even worse, 99 percent of it could work and 1 percent could produce erroneous results. ROMs can be effectively tested during power-up self-test, if such tests are



Using Signature Analyzer to Troubleshoot Microprocessor-Based Product

designed in. But, unlike RAMs, ROMs can also be tested by other techniques if no self-test is provided. One such technique involves free-running the system and then using the signature analyzer to either verify documented signatures or compare the outputs of a suspected ROM with that of a ROM in a known good system (see Experiment 17-2).

SELF-TEST PROGRAMS

The programmability of microprocessor-based systems can be used to great advantage in assisting system testing. Programs stored in the system's ROM can test ROMs, RAMs, and the processor itself. Often the I/O can be tested to some extent. Software can also be used to provide stimulus for an external test instrument, such as a signature analyzer.

ROM Testing

The most common technique for testing ROMs uses a *checksum.* When the ROM is programmed, all of its words are added together, ignoring any carries that result. This number is complemented and stored in the last (or sometimes the first) word of the ROM, so that when all the words are added together (including the checksum stored in the last byte), the result is zero. If the total is not zero at the end of the test sequence, then something is wrong with the ROM. (In actual practice, the checksum is usually calculated to make the total a specific number other than zero.)

Unfortunately, the checksum is not totally reliable. It detects any single-bit error and most multiple-bit errors; however, there are many combinations of two or more errors that still produce the correct checksum. Thus, a ROM that passes a checksum test is probably good. If the test fails, something is definitely wrong (though it might not be the ROM itself).

RAM Testing

RAMs are tested by writing a pattern into the memory, reading it back, and then verifying that has changed. Of the many different patterns that can be used, a common one is the checkerboard. In this pattern, all the bits are set to alternating ones and zeros. Once all memory locations have been tested, the pattern is repeated with each bit reversed, verifying that each bit of the RAM can store a one and a zero. Many other patterns used to test RAMS are specifically aimed at detecting various failure mechanisms within the RAM.

No memory test can guarantee 100 percent accuracy, even though it may show that each bit can store a one or a zero. RAMs can be pattern sensitive. For example, one location might correctly store 01010101 and 10101010 but fail when 01111000 is stored. Even for a small RAM, it would take an extremely long time to test every possible pattern sequence. For this reason, RAM test credibility is generally much lower than that of ROMs. As with the checksum test, if a RAM passes the system self-test program, it is probably good. If it fails the test, something is definitely wrong.

MULTIPLEXED I/O

Multiplexed keyboards and displays often share some of the same scanning circuits (as does the μLab). In these situations a stuck key can appear to make the display fail. Likewise, a bad display driver input could cause a keyboard error. The interaction between common scan circuits must be considered in making a diagnosis.

INTERFACES

Many microprocessor systems interface with other systems through external communication lines (e.g., IEEE-488, RS-232C, telephone modem). These lines are frequently long and are often exposed to sources of electrical interference, such as relays, transformers, motors, solenoids, and even lightning. Electromagnetic interference (EMI) emanating from these sources can cause the transmission of faulty data, overstressing of interface circuits, and, especially in the case of lightning, gross component failures. Generally, output line driver circuits tend to have higher-than-average failure rates, due both to EMI stressing and to the high transition currents that result from driving capacitive interfacing cables.

TROUBLESHOOTING TREES

A *troubleshooting tree* is a graphical means of showing the sequence of tests performed on a product under test. These trees are often drawn as flowcharts in which the results of each test determine what step is taken next. The use of troubleshooting trees for repairing microprocessor-based products can save considerable time and effort.

Figure 18-1 shows a portion of the troubleshooting tree for the HP 3455A Digital Voltmeter. Theoretically, it should lead you to the product's fault by means of the actions taken and decisions made along the tree. Unfortunately, such is not always the case. A perfect troubleshooting tree must consider all possible failures, a difficult criterion for the person writing the troubleshooting tree to meet. Also, troubleshooting trees tend to be fairly generalized, lacking the specifics desired for making tests and decisions. Few troubleshooting trees provide practical information about how a specified test or measurement relates to what the circuit does or is supposed to do. If the troubleshooting tree fails to direct you to the actual fault, you may be left at a dead-end, with no idea of where to go next. However, the troubleshooting tree will often be your best guide (at least to begin with).

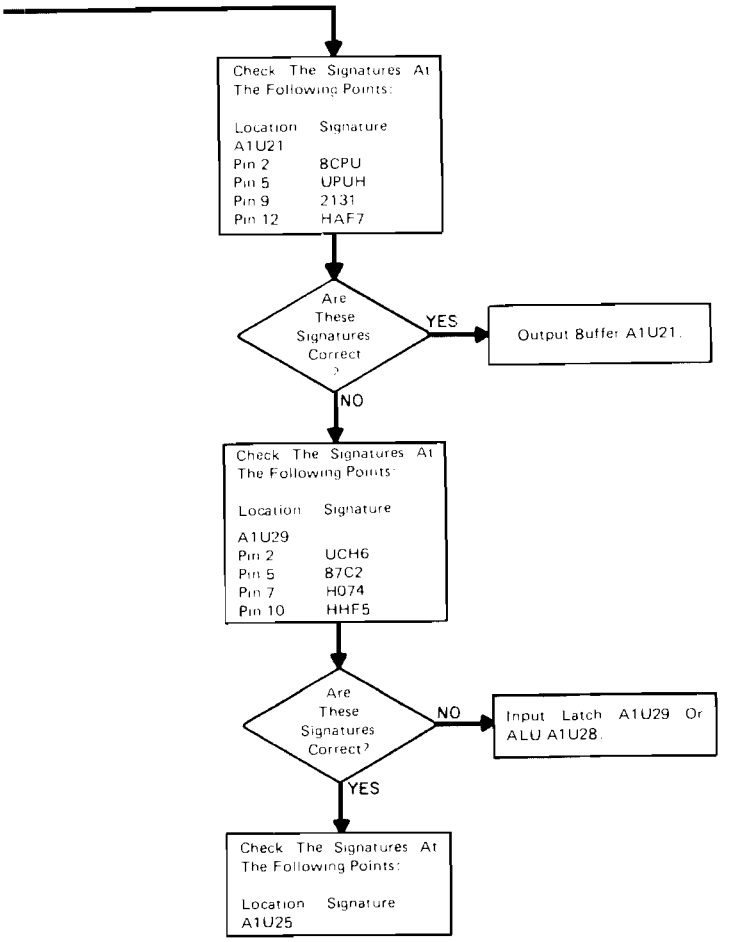


Figure 18-1. Typical Troubleshooting Tree for Product Incorporating Signature Analysis

There are good troubleshooting trees and there are bad troubleshooting trees. The good ones seldom lead to a dead-end and provide a logical, well-directed sequence of tests and measurements, requiring a minimum level of understanding of the circuit under test. Often they include advanced techniques such as signature analysis to simplify the procedure. In troubleshooting a product, even the poorer troubleshooting trees can be useful for localizing a failure area in the system and can save considerable time and effort.

OTHER DOCUMENTATION

For many experienced troubleshooters, working from product *block diagrams* can supply the right amount of information to understand how the different parts of the circuit work together. A product's theory of operation and its troubleshooting trees do not relate as closely to the hardware. The schematics often provide too much detailed information, making it difficult to see the "big picture."

The remaining portion of this lesson outlines a loose sequence of general steps that you can take to troubleshoot a microprocessor-based product. Numerous servicing techniques and "tricks of the trade" are interspersed with the descriptions.

IS THERE REALLY A PROBLEM?

It is important to have a general understanding of the defective product so that you can be sure that a problem really exists. To some degree, you should know what it does and how it operates. Microprocessors allow designers to design products that are not only complex in function, but sometimes complex to operate as well. Be sure the apparent problem is not a user error, but a real product malfunction. Few things are more frustrating than trying to fix something that is not broken. In some situations, it appears that a product should do something it was not actually designed to do. For example, a DVM AC select switch may work on VOLTS but not on AMPS. This "design limitation" can usually be verified in the operating manual and does not constitute a product malfunction; it is only a shortcoming.

Design "bugs" in the firmware (ROM) can sometimes cause failures when used under operating conditions that were not anticipated during the product design. These are more likely to occur in early production runs and can best be verified (if suspected) by contacting the manufacturer. At the other extreme, a problem may actually exist but not show up because the product is not adequately exercised. These kinds of problems are often very simple to detect (e.g., observing a burnt out OHMs LED indicator when pushing the OHMs button on a DVM). They can also be complex problems. For example, errors can occur when an unusual sequence of operations is performed. Because the complex problems are much more difficult to test for, extensive test procedures are used to test products at the factory. The customer, bringing in a product for repair, has no trouble pointing out a problem. It is up to the troubleshooter to solve it.

WHAT CAN BE LEARNED FROM THE FRONT PANEL?

A great deal of diagnostic information can often be obtained without even removing the product's covers. Most microprocessor-based products have some sort of front panel. On it there may be switches and indicators, inputs and outputs. *Milking* the front panel is a process in which the switches, buttons, and other inputs are used to solicit responses from the product that can be observed using its indicators and other outputs. For instance, if the indicators are all dead when the power is turned on, you might suspect a bad switch, fuse, power cord, battery connection or power supply. If one segment of a display is dead, the problem is probably the display itself or the circuit that drives it. If the only failure of a DVM is in the 1-10 VOLT range, the problem area can be narrowed down to a relatively small portion of the circuit (the attenuator).

Always take advantage of any designed-in performance verification or power-up test modes and diagnostic messages that are available. These are specified in the product manual.

At this point you may have some idea of where the problem is or you may have even fixed it. But in all likelihood, neither has taken place.

WHAT DOES THE MANUAL SAY?

"If all else fails, look at the manual." This rather poor (but prevalent) attitude makes even less sense for microprocessor-based products than for conventional ones. There may be a bonanza of service aids and procedures in the manual just waiting for you to try out. Special service switches, jumpers, test fixtures, indicators, and test techniques can make the job much easier.

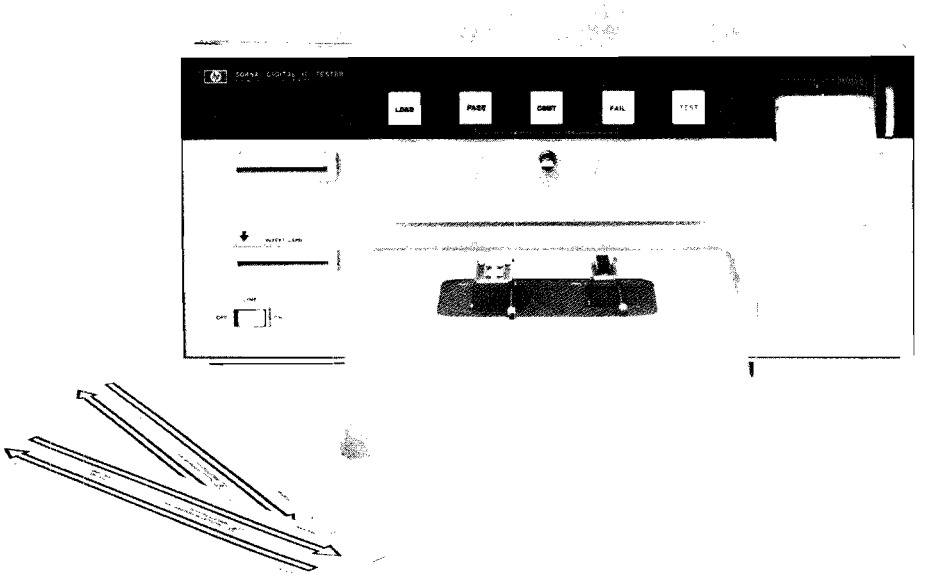
Try to understand the circuits and figure out where things are. Check out the manual's theory of operation section, the block diagrams, and the schematics.

You do not have to do this in great detail but just enough to have some idea of what is going on. Identify the microprocessor, ROM, RAM, I/O, address decoder, clock, bus, control, and interrupt portions of the system.

PRODUCTION VERSUS FIELD FAILURES

The life of an IC is generally a sequence of predictable events. It is born in the IC factory and is sent to a product manufacturer. There it is inserted into a circuit board, which in turn is inserted into a product. Then the product goes into service, and the IC remains there for the rest of its useful life. Needless to say, not all ICs live a long and healthy life.

Product manufacturers estimate that approximately 2 percent of all incoming ICs are defective. Testing incoming ICs on an IC tester will detect most of these. The effective cost of finding a defective IC at this point is about 10 cents. Once ICs are loaded into circuit boards, the bad ones cost about $1 to find. If they are not detected until the boards are assembled into the end product, this "in-situ" troubleshooting and repair costs about $10 at the factory. Replacing a bad IC in the field is even more expensive: a typical bill for finding and replacing a faulty IC in a customer's product is about $100. Clearly, it makes sense to find and eliminate the defective ICs as early in the cycle as possible.



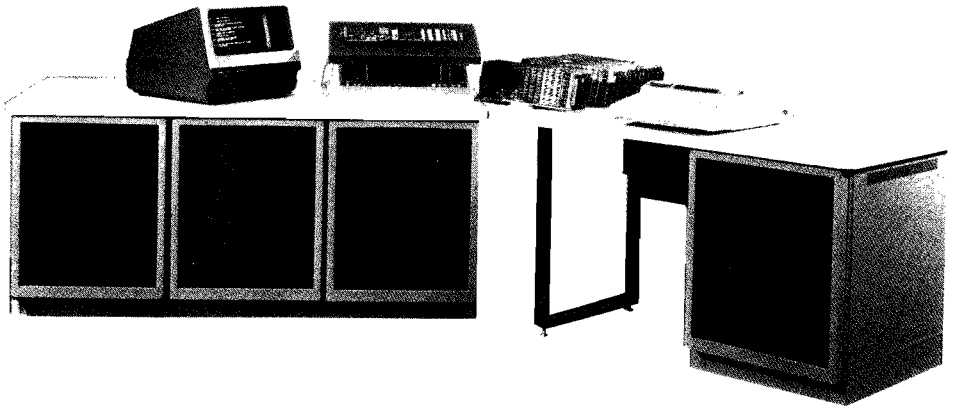
HP 5045A Digital IC Tester Used to Perform Incoming Inspection

Types of Failures

Common fault sources and the best troubleshooting techniques for finding them depend on the history of the product and the environment in which it is tested. When a new product is first turned on at the factory, almost anything might be wrong with it. Products that fail in the field have all worked at one time. Assembly errors, such as misloaded components and miswired circuits, generally need not be considered in field failures. Also, the likelihood of solder shorts and multiple faults is much greater on the production line than in the field. Field failures are usually caused by components or connections that have failed.

Automatic Testers

Because of the volume of identical products tested at the factory, specialized testing and troubleshooting equipment and techniques can be justified. Automatic board testers and test fixtures are often used to minimize the time it takes to locate faults. In general, they provide fast, economic, and accurate verification and fault diagnosis. Do not, however, fall victim to overconfidence in computer-controlled automatic board testers. Occasionally, boards passed by a production board tester are actually defective as a result of deficiencies (timing, loading, or component exercising) in the tester or the test program being run. However, newer board testers that perform more sophisticated dynamic, functional, and parametric tests have greatly increased credibility.



HP 3060A Board Test System Performs Fast, Thorough, and Efficient Testing in Production Environment

WHAT ARE THE EASY THINGS TO TEST?

It makes sense to look first at the things that can be tested and repaired easily. The simple things are as likely to fail as the complicated ones. A case in point is the power supply: it is actually one of the more failure-prone portions of most products. It is also one of the easiest to test and is usually simple to troubleshoot. An out-of-spec voltage can cause erratic circuit performance. If the voltages are not checked first, it could take considerable time to find the problem.

A mechanical inspection can also be fruitful. Poor PC board and cable connectors, broken wires, and loose parts can usually be found either visually or by touch.

COMMON PRODUCTION-LINE TROUBLESHOOTING PROBLEMS

A number of common sources of failure in a manufacturing environment can be found through careful visual inspection of a product's circuit assemblies. It is easy to check for improperly set switches and jumpers, misloaded components (wrong ones and backward ones), and cold solder joints. Backward resistor packs can be particularly hard to diagnose electrically because they can cause interaction between unrelated logic nodes, but they are easy to check visually.

Two of the more common failures in production are solder and gold (copper) shorts on printed circuit boards. These can usually be removed with a sharp knife. When the precise location of the short is not known, there is a rather novel technique for removing it that often works. It is also useful for situations in which the location of the short is not accessible (such as inner layer shorts on multilayer boards). The procedure involves charging a 100,000 μF (or larger) capacitor

to five volts (a safe voltage for logic circuits). Then, with cables solidly connected to the two shorted nodes and proper polarity observed, discharge the capacitor into them and listen for a snapping sound on the board. Check continuity to see if the short has been opened and, if not, try again. This technique should be used with caution since it will open the weakest link of the current path, which may not always be the fault source, but may be a fine trace or a plate-through. The current tracer provides a much safer means for finding shorts, as demonstrated in Experiment 16-4.

A relatively new problem in production is the occurrence of bent-under IC pins caused by automatic component insertion equipment. These can result in an open electrical connection between the IC and the PC board, an intermittent connection, or shorts to traces near or under the IC. The bent-under pin is often difficult to spot visually because it may look as though it is properly soldered in place. The best way to tell is to look at the bottom of the board for the ends of any IC pins or along the plane of the board to see under the ICs.

PC board edge connectors are commonly used. They may cause problems in production when their borders are cut off center or when they are accidentally covered with solder resist or board sealing spray. Visual inspection can reveal such problems.

Multilayer PC boards suffer from all of the problems of regular boards plus some of their own. Misregistration and contamination of inner layers (which can cause high frequency or leakage problems) can often be observed by holding the board up to the light. Since repair of the inner layers is often impossible, the entire board may have to be scrapped.

Wire-wrap boards are prone to bent posts that cause shorting. Other common production problems include 14-pin ICs loaded into the wrong end of a 16-pin socket, miswiring, wire shorts between pins, and signal coupling (crosstalk) due to closely bundled wires.

MECHANICAL FIELD FAILURES

Visual inspection of a product that fails in the field can reveal such things as loose wires, broken traces, cracked ceramic ICs and resistor packs, bent wire-wrap posts, and dirty connectors. A "calibrated fist" on the side of the cabinet can often be used to detect loose or intermittent connections and stuck relays. Mechanically stressing boards and connectors (by twisting and flexing) can often help to locate some of these problems. You might suspect the PC board edge connectors when a product is "D.O.A." (dead on arrival) or fails in a hostile physical environment. You may want to try reseating all of the assemblies and circuit board connections to determine if the problem is poor connector contact. A pencil eraser is useful for cleaning dirty edge connectors.

GENERAL TROUBLESHOOTING TECHNIQUES

Board Swapping

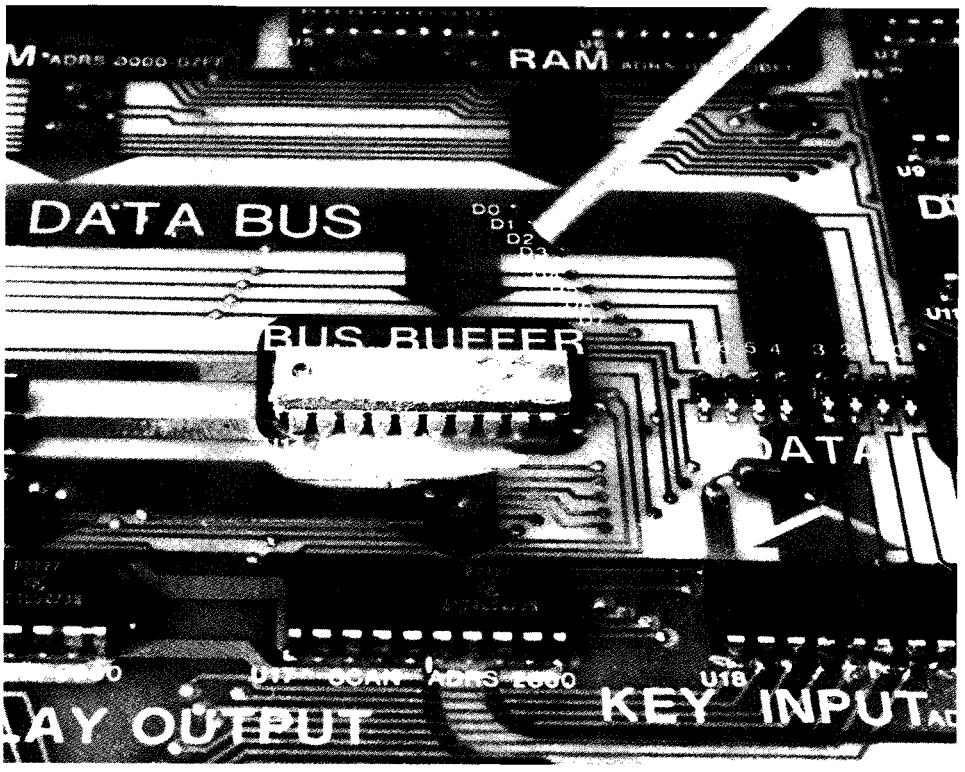
If any of the PC boards are easy to remove and replace and known good ones are at hand, you can try swapping them. When duplicates of the same board or assembly are used in one product, they can be swapped with each other. The risk involved in board swapping is that you could damage a good board because of the same electrical overload that damaged the bad one when it was installed. In any case, power to the product should be turned off when removing or installing boards or assemblies.

If an identical product is available, functional comparisons can sometimes be informative. This comparison can be especially useful in situations in which it is not clear that there is actually a hardware problem (it may be a product idiosyncrasy or design limitation).

If a device in a socket is suspect, try tapping it first to see if there is a loose connection and then try substituting a known good one. Note, however, that one of the last devices you should suspect, but that is most often the first to be replaced, is the microprocessor. The actual failure rate for microprocessors is very low. However, because they are complex and their correct operation is difficult to verify, they are often the first to be plucked from a PC board. This is also true of the LSI chips used with them.

Stress Testing

A technique referred to as *stress testing* can be very effective in dealing with marginal or intermittent failures. Stress testing can often cause these types of failures to temporarily improve or deteriorate; either case is beneficial in locating a fault. Boards are stressed physically by tapping or twisting them, thermally by heat (air gun or hair dryer) or by cooling them (from an aerosol freeze can), and electrically by varying the supply voltage. Thermal stressing can be used to isolate a fault in a specific device on a board more precisely than the other methods because heat or cold can be applied directly to a single component. Intermittents can result from marginal chips, lead bonds, solder joints, connections, and drive and timing circuits.



Cold Spray Helps Identify Faulty and Marginal Devices

Briefly touching each device on a circuit board can pinpoint a component that is running hot (much hotter than the others). When a particular device runs significantly hotter than others of the same type, a problem may exist. A faulty device

can sometimes be hot enough to burn your finger, so use this technique with caution. Be aware also that some good devices may run hotter than you expect during normal operation, and that temperatures may vary widely from one device to another.

Power Supply Shorts

There are some effective ways of dealing with shorts across the power supply. The first thing to do in a multiboard system is to try to localize the short to a single board. This can be done by removing one board at a time until the power supply is no longer shorted. The last board to be removed is the shorted one.

One technique for finding the short on a faulty board is to inject current through the two shorted lines with the logic pulser. The current tracer is then used to follow this current to the short. Keep in mind that capacitors (especially electrolytics) will have some current going into them because of the pulsing current. Shorted capacitors can be found by using the current tracer to compare the current levels going into identical capacitors on the same board. The capacitor that shows a much higher level than the others is likely to be shorted. This technique is particularly useful for finding shorted ceramic bypass capacitors.

Another technique for locating power bus shorts is to supply a relatively high current (about 3-5 Amps) into the short. Be sure to maintain the same voltage polarity and not to exceed the supply voltage normally present. The current path to the short can often be determined by using a DVM with high resolution (.01 mV) to look at voltage drops on the power bus traces. Voltages are developed across the traces that are in the path going to the short, and not elsewhere (see Figure 18-2).

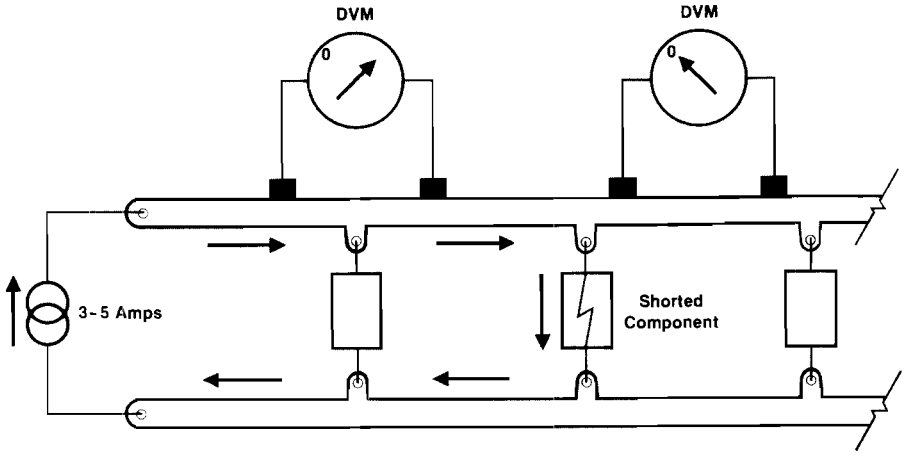


Figure 18-2. Using Sensitive Voltmeter for Locating Power Bus Short

A less scientific, but much more dramatic, technique for finding power bus shorts is to freeze the entire board (to about -10 degrees C), allow moisture to condense on it, and then power it up with a 3-5 Amp supply. As it warms up and defrosts, the current path becomes visible and, in many cases, will pinpoint the short.

HOW CAN THE FAULT BE ISOLATED?

Once the easy things have been tried unsuccessfully, it is time to get down to business. At this point individual troubleshooting skills, intuition, and knowledge of the product really make a difference.

First, be sure to take advantage of any designed-in and documented circuit isolation features, such as selected board removal, service jumpers, and special test modes and procedures. It can be very useful to separate the microprocessor system from the peripheral circuits to allow you to diagnose each portion independently.

An important troubleshooting concept is *half-splitting*. Although the term may be new to you, you've probably been using the process for years without even knowing it. Half-splitting involves choosing a point roughly in the middle of the circuit. It is just as likely that a fault exists before as after that point. If the performance is correct up to that point, the fault lies after it. If not, then the fault is before that point. This process works best in circuits that have clear, unidirectional signal paths without large feedback loops. Even with microprocessor-based systems, this approach can be effective because the circuits outside the microprocessor portion often fit these guidelines.

In a typical product, the first half-split is generally done at the digital-to-analog interface, if possible. Analog circuits often have higher failure rates (due to higher demands made on speed, power, temperature, sensitivity, accuracy, adjustment, external overloads, and reduced component safety margins). The contribution of a product often relates to its analog circuits. These are often the circuits that represent the "high-technology" contribution and that may be operating near their limits. They may also outnumber the digital ones. Be aware also of the possibility of the electrical interaction of clock and TTL power bus lines with analog circuits, which can cause serious system noise problems.

DIGITAL FAILURE MODES

When suspicion falls on the digital portion, the first thing to look for is signal activity. With a logic probe you can examine activity on the clock signals, bus lines, chip enables, and control lines. Absence of activity on any of these nodes indicates a possible problem. You may wish to refer back to Experiment 16-1 to refresh your memory about troubleshooting with the logic probe.

The most common failure mode for digital ICs is open lead bonds inside the package. There are thin wires connecting the package pins to the IC chip. If an output lead bond opens, the output pin floats and the logic probe will probably indicate a constant floating logic level because of other device inputs connected to that node. If an input lead bond opens, one or more of that IC's outputs will usually appear to malfunction (stuck high, low, or executing its logic function incorrectly). If any of these outputs goes to a three-state bus, it can cause bus conflicts (more than one output on at a time), and the current tracer can be used to find these. Bus conflicts are often observed on an oscilloscope by the presence of bad, but solid, logic levels on bus lines, but the scope provides no information as to the source of the fault (see Figure 18-3). Good bus lines can also appear to have solid, bad levels present when all devices on the bus are off.

Another common digital IC failure is a shorted input pin to ground. This fault is often caused by a bad input protection diode on the chip. It usually appears as a stuck low level, which can be seen with a logic probe. An oscilloscope connected to a node with this type of problem shows a voltage level near ground being pulled up, perhaps a few hundred millivolts, whenever a logic 1 output on that node turns on (see Figure 18-4). The current tracer provides an excellent means of pinpointing shorted input pins.

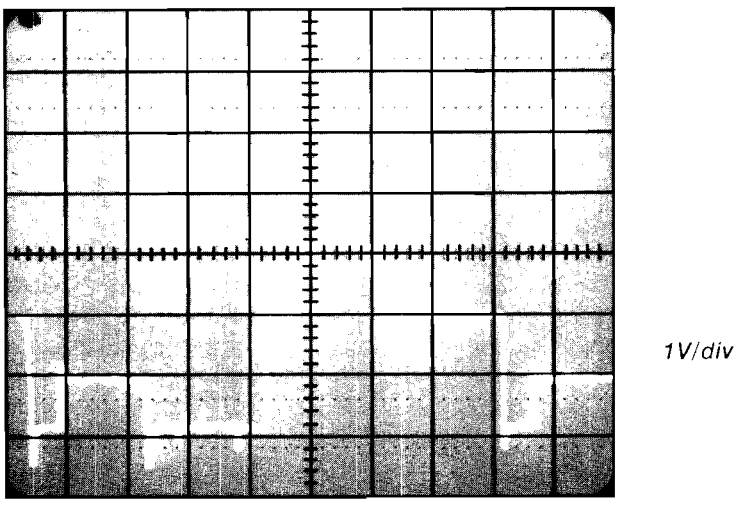


Figure 18-3. Bus Conflicts Cause Bad, But Solid, Logic Levels

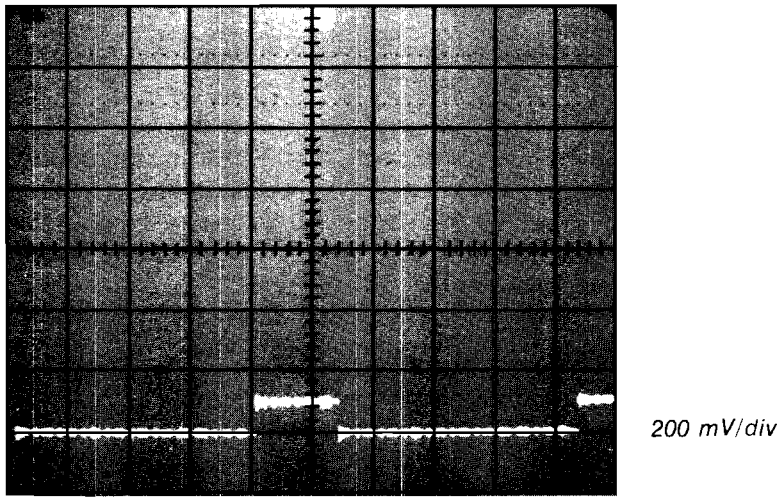


Figure 18-4. Shorted Substrate Diode on Gate Input Pin Clamps Node to Ground

If a current tracer is not available, another means for locating stuck inputs and outputs involves the use of a sensitive (high resolution) DVM and a can of cold spray. Connect the DVM to the stuck node and select the most sensitive DC voltage range available. Then, while monitoring the voltage, spray each IC connected to the stuck node, one at a time, to change its temperature. Any noticeable change in voltage (more than 10 mV) on the node indicates that the IC being sprayed is drawing current. If a freeze can is not available, a heat source can be used instead. This technique relies on the properties of the semiconductor material used in the IC that relates voltage to temperature.

ISOLATION TECHNIQUES

Once a particular input or output pin is suspected, it is useful to isolate it from the rest of the circuit. A quick, nondestructive way to do so is to suck the solder away from the area between the pin and the PC board pad, using a vacuum desoldering tool or solder wicking braid. Then bend the pin so that it is centered in the pad's hole, not touching it at any point. Use a continuity tester to verify that the pin is no longer in electrical contact with the board.

The techniques that you can use to isolate the digital blocks of a microprocessor-based product are entirely dependent upon its electrical and mechanical architecture. If some of the digital boards can be removed and still allow the kernel to operate, this procedure can be useful. If the kernel can be allowed to run open-loop (no feedback from the data bus), a free-run mode can sometimes be used to check the kernel and address bus activity.

An extender board with switches on bus and signal lines can be used to break selected signals between a PC board and the rest of the system. In this manner, feedback paths and stuck buses can be removed from the main system.

An even simpler way to open selected signals going through a board edge connector is to place a piece of tape or stiff paper on the PC board edge fingers that you wish to isolate. Be careful to note to which board(s) you have done this to so that you will remember to remove the tape or paper later.

A somewhat unconventional, but often effective means of detecting bus line problems is to measure the resistance to ground (with the power off) of each of the bus lines in a particular bus (e.g., data bus, address bus). The resistance of each of these lines is usually the same. If any one differs substantially, you may suspect a problem on this line. If two lines show the same (lower) resistance, the two lines may be shorted together. In either case, check the schematic to see if the arrangement of circuits connected to these lines could explain the differences before going further.

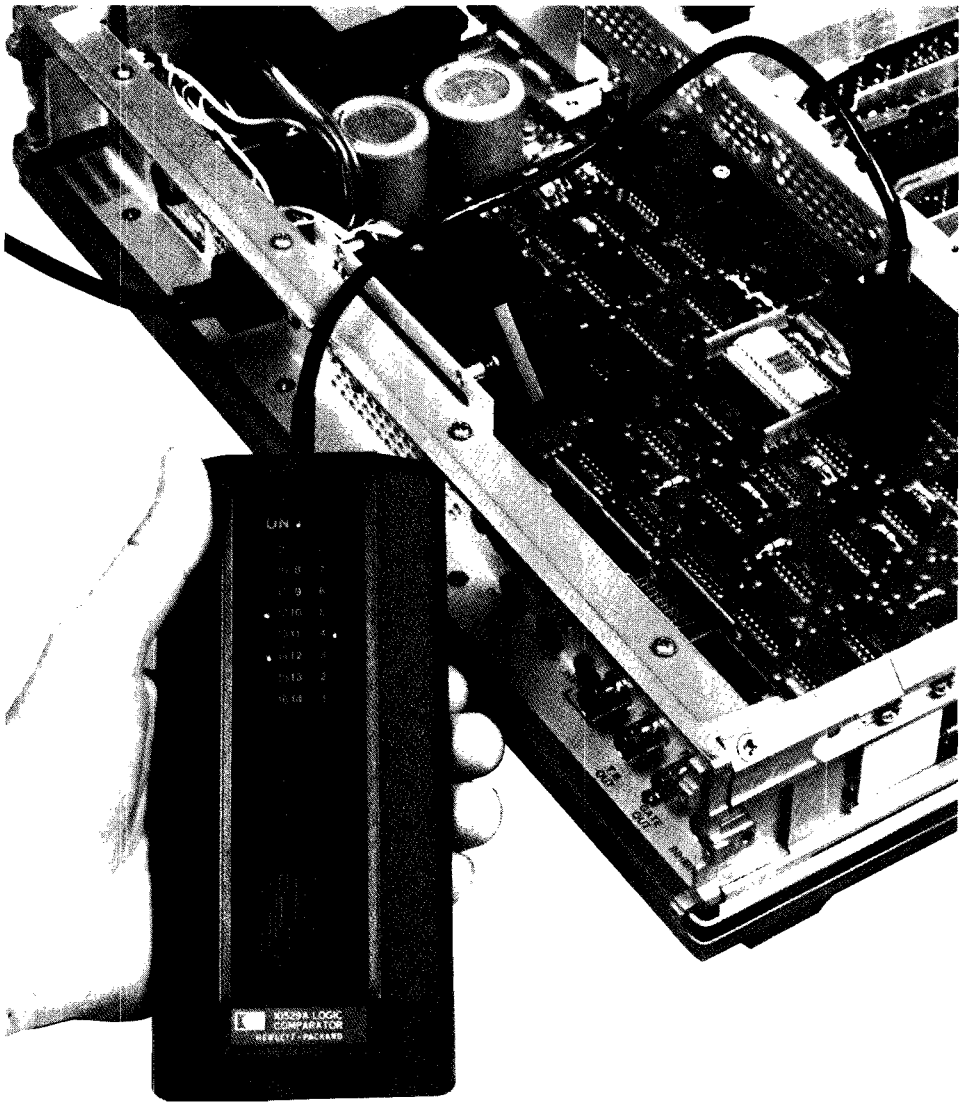
Overriding interrupt lines and chip enable pins on suspected devices can be used to verify that the IC is functioning correctly. This can be done by momentarily shorting the appropriate pin high or low, or by using a logic pulser (refer to Experiment 16-2).

FEEDBACK LOOPS

Digital feedback loops are often difficult to troubleshoot because errors propagate around and around. A feedback loop with a faulty output signal sends this signal back to the input to produce more bad outputs. Opening this feedback path prevents the faulty output signals from going back to the input. Then, if controlled inputs to the loop can be generated, the signal flow from the input to the output can be observed. Often, however, it is not easy to provide this input (many lines may need to be controlled). It may also be difficult to predict correct circuit operation. If another working product (or board with the same circuitry) is available, it is sometimes practical to allow the output of the good circuit to

control the inputs of both circuits. In this manner, you know that the circuit under test is getting the correct input signal. It is then a matter of comparing the nodes of the two circuits and looking for differences. A signature analyzer can be useful for doing this comparison.

Piggy-backing ICs is a technique that can sometimes be used to locate defective ICs. It involves looking at suspicious IC outputs with an oscilloscope or signature analyzer and then placing an identical IC package directly on top of it. The pins should be bent slightly, if necessary, so they are all in contact. A signal change can indicate problems with that device. If no change is observed and the output is not stuck, it can generally be assumed that the IC is not the problem. Be cautious of sequential circuits (such as counters and shift registers) that may cause output differences because of start-up conditions. A better way of performing this test is to use an IC comparator, such as the HP 10529A Logic Comparator.



HP 10529A Logic Comparator Performs In-Circuit Logic Device Comparisons to Known Good Reference

CONCLUSION

No amount of knowledge and experience can totally compensate for inadequate service documentation. In some cases, shotgunning (replacing components until the problem disappears) may be the only solution. Most microprocessor-based products, fortunately, do not fall into this class. Future products will probably incorporate advanced service techniques, such as signature analysis, as more designers realize that the old troubleshooting methods and tools used for random logic are not very effective in dealing with microprocessors.

Occasionally Shotgunning Produces Unfavorable Results

REVIEW

Lesson 18

Microprocessor systems can be thought of as an extension of traditional digital logic. Many of the components, circuit designs, and troubleshooting tools and techniques are the same. However, there are some differences. Microprocessor systems are bus structured, and many of the devices on the bus are complex LSI devices. The signal activity between the devices on the buses is constant and complex. It is often useful to break the data bus, which is the system's main feedback path, to help isolate a fault that causes the entire system to malfunction.

Although troubleshooting trees provide an orderly approach for locating system faults, they are not always adequate. There are numerous techniques, procedures, and tricks that can be effective in diagnosing, isolating, and locating faults in microprocessor-based products. Many of these were discussed.

QUIZ

Lesson 18

1. In microprocessor buses, oscilloscopes are least effective when looking for:
   a. improper data.
   b. faulty logic levels.
   c. timing problems.
   d. bus conflicts.

2. The most effective tool for finding the defective device on a stuck bus is the:
   a. signature analyzer.
   b. logic analyzer.
   c. oscilloscope.
   d. current tracer.

3. A potential problem with troubleshooting trees is that:
   a. they are hard to follow.
   b. they have termites.
   c. they require too much knowledge of the product.
   d. they may lead you to a dead-end.

4. The first troubleshooting step should be to:
   a. read the product service manual.
   b. check the fuse and power cord.
   c. shake the product and listen for rattles.
   d. determine the nature of the problem.

5. A key requirement for *half-splitting* is:
   a. unidirectional signal paths.
   b. SA test modes.
   c. board swapping.
   d. having a good comparison product available.

6. The most common failure mode for digital ICs is:
   a. a wrong chip in the package.
   b. a shorted input diode.
   c. an open lead bond.
   d. a bad output logic level.

LESSON 19

Troubleshooting the Microprocessor Lab

INTRODUCTION

This lesson describes the troubleshooting flowchart for the μLab and shows how to find a fault using this flowchart. You will then insert and find faults on your own. Because the μLab is a complete microprocessor system, most of the types of failures discussed in the previous lesson can occur in it. The troubleshooting strategy for the μLab is therefore similar to that for many other microprocessor-based systems.

THE MICROPROCESSOR LAB TROUBLESHOOTING FLOWCHART

Figure 19-1 shows the troubleshooting flowchart for the μLab. It makes use of the designed-in self-test and signature analysis features. This flowchart provides a useful guide for directing the tests and measurements necessary for locating a circuit fault.

Is There a Problem?

This question is not as trivial as it may seem. There are several faults in the μLab that do not cause obvious malfunctions. Some examples are power-up failures, the inability to store data in memory, the inability to properly execute the ECHO demo program that is stored at address 04D7, and partial keyboard failures.

Does It Power-Up?

The μLab contains a power-up performance verification program that is run whenever power is first turned on. The most obvious occurrences during this test are the momentary lighting of all output LEDs and display segments, the beeping of the speaker, and, finally, the μLAb UP message in the display. If you obtain all these results, you can assume that all is (reasonably) well. In addition, tests are run on the ROM and the two RAM chips during this program. If a problem is detected, a display message may result, giving the IC designation number (IC4, 5 or 6) of a possibly bad ROM or RAM. The mere fact that the μLab is able to go through this power-up program sequence and control the display tells you that the basic microprocessor system (microprocessor, ROMs, RAMs, decoding and control circuits, and buses) is working to a substantial degree.

Lights On?

If the μLab fails to power-up and none of the LED indicators light, the power supply or primary power source should be suspected.

Bus Activity?

If the power supply is good, then the next logical test is to determine the presence of activity on the data and address bus lines, using a logic probe or signature analyzer probe. Lack of activity would explain the failure to power-up and could be caused by a faulty clock. Or, an incorrect logic state could be present on the Reset, Hold, or Ready inputs to the microprocessor. If these lines are good, or if one of them is bad but you cannot easily tell why, continue on to the next step. A faulty ROM could also cause a HALT instruction to be executed, causing bus activity to stop.

SA Test Loop Working?

This point in the procedure is basically a half-split (described in Lesson 18). The signature analysis test loop requires more of the circuit to operate than the free-run modes need. Try to get the μLab into the SA test loop by pressing the RESET key and then sliding the SA switch up and down once. If it is working, all of the output LEDs and display segments light and the speaker beeps once. Pressing the INTRPT key also causes the speaker to beep repeatedly. If most of these actions occur, the SA loop is probably running. The best way to be sure is to connect the signature analyzer to the μLab and check the Vcc set-up signature specified in Table C-3 or C-4 (the write or read SA test loop).

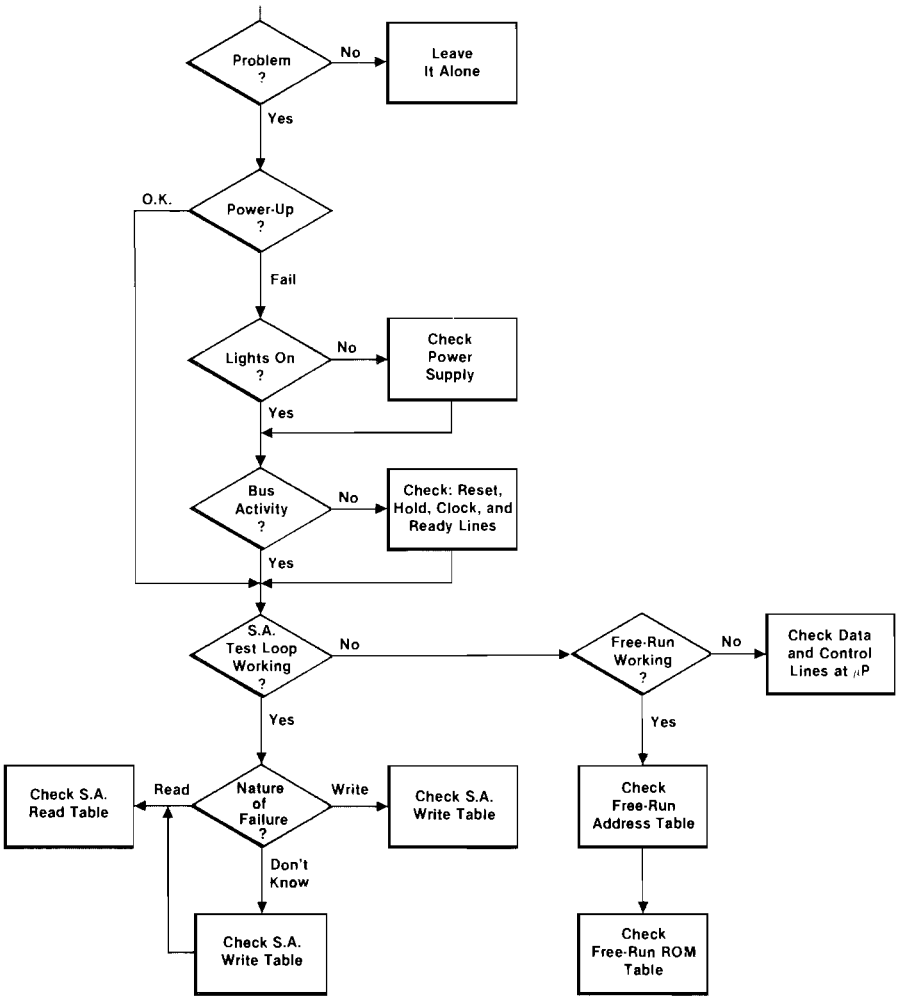


Figure 19-1. Microprocessor Lab Troubleshooting Diagram

If the SA test loop runs, you know that the essential portions of the system are operating satisfactorily. The microprocessor is addressing the ROM, is receiving instructions from the ROM through the data bus, and is executing those instructions.

There are two stimulus programs in the SA test loop. One of them writes stimulus patterns to the devices that the microprocessor talks to, and the other reads data from the devices that the microprocessor listens to. Both of these programs run alternately while in the SA test loop. By changing the connections to the signature analyzer, either the talking or listening devices can be checked.

The symptoms of the fault often point to a general portion of the circuit and are classified as a read or a write type of problem. For example, a faulty display is most likely to be a write problem. The signature analyzer is then connected according to the set-up specified in the SA write loop table (Table C-3). Signatures on nodes in the display circuit are then checked against entries in that table.

Free-Run Working?

When the SA test loop will not run, you need to test smaller portions of the circuit. Opening the data bus lines to the microprocessor (by sliding the eight BUS SWITCHES up) and inserting a free-run instruction (by sliding the FR switch up) causes the microprocessor to cycle through the address space on its own. Opening the data bus lines allows you to isolate the microprocessor from the data bus and the rest of the system. In the free-run mode, the microprocessor stimulates other portions of the circuit through the address bus. This address bus stimulus is unsophisticated compared to the well-controlled data patterns used in the SA test loop. However, the free-run mode does exercise portions of the address bus drive and the decoding and control circuits, as well as the ROM. The advantage of this test mode is that, in order to use it, little more than the microprocessor chip has to work.

The free-run test mode can be identified by the action of the bus and status LEDs on the μLab. The fourteen least-significant (right-most) address bus LEDs should be flashing so rapidly that they appear to be steadily lit. The A15 and A14 LEDs should appear to flash rapidly. The status LEDs should behave as follows: READ on, WRITE off, and ROM, RAM, INPUT, and OUTPUT flashing.

If the microprocessor will not free-run, check the control lines going to it (Reset, Hold, Ready, and Interrupt) and the clock to see if one of these is the cause. You can check the power-on reset circuit by forcing its output on (low). You might also check the data bus pins at the microprocessor to see whether it is getting the free-run instruction. If these signals all appear to be correct, the microprocessor is probably defective. There are few other factors that could keep a properly powered and controlled microprocessor from free-running.

The Free-Run Address Test Modes

The free-run address test mode allows you to exercise most of the address and control circuits in the μLab. Table C-1 shows the set-up and signatures. A correct Vcc signature verifies the proper test set-up and free-running of the microprocessor.

When the μLab is free-running correctly, the address bus and much of the decoding and control circuits can be tested. The first signatures to look at are those on the address bus lines. They should all agree with the signatures in

Table C-1. Signatures on the device chip select pins, the address decoder (IC7), the other control circuits, and the microprocessor (IC3) control pins can then be taken. If all of these signatures are correct, you can assume that the address, decode, and control portions of the system are probably good. However, if you have suspected other problems as well, you could check the remainder of the signatures in Table C-1. If all these signatures are correct, you should suspect a bad device on the data bus and go on to the free-run ROM test mode.

Free-Run ROM Test Mode

The free-run ROM test mode requires a different set-up from the free-run address mode so that it samples data only when the ROM is being addressed. In this way the contents of the ROM can be verified and faulty data bus lines can be detected. Either of these conditions could prevent the μLab from running the SA test program (or any other program).

The procedure for testing the ROM is to first connect the signature analyzer to the μLab according to Table C-2 (while still in the free-run test mode). Verify the set-up by checking the Vcc signature. Then check each of the eight data bus lines. A faulty signature on any one of these lines should be checked at the corresponding ROM output pin to determine whether the problem is a ROM output or a board trace. If bad signatures are found on all of the bus lines, the ROM enable and address signals should be checked at the ROM pins. If these signatures are good, then another device may be erroneously enabled onto the data bus, creating a bus conflict. Signatures on the other bus device enable pins can identify this problem. If these signatures are good, the system can be stopped while the ROM is enabled. The logic pulser and current tracer can then be used to find the offending bus device (refer to Experiment 16-4).

EXPERIMENT 19-1

Familiarization with the Fault-Finding Process

CONCEPT

In this experiment, you will insert a fault into the μLab and then follow the troubleshooting tree, making observations, measurements, and decisions until the fault is found. In the process of finding this fault, you will gain familiarity with the tools and techniques used for troubleshooting the μLab.

PROCEDURE

I. Setting the Fault

A) Verify that the μLab is fault-free prior to inserting the fault. To do this, simply turn the power off, wait a second, and then turn it back on again. The display shows μLAb UP, and you know that no major faults are present.

B) Locate fault jumper W1 in the upper center portion of the board (see Figure 19-2). Notice that it is plugged into the left two holes of a series of three holes. Remove and then re-insert it into the right two holes. Observe that the display goes blank or contains stuck digits. For the rest of this experiment forget, having ever touched jumper W1.

C) Now you've done it. You have broken it and cannot remember what you did, right? Read on to see what to do next.

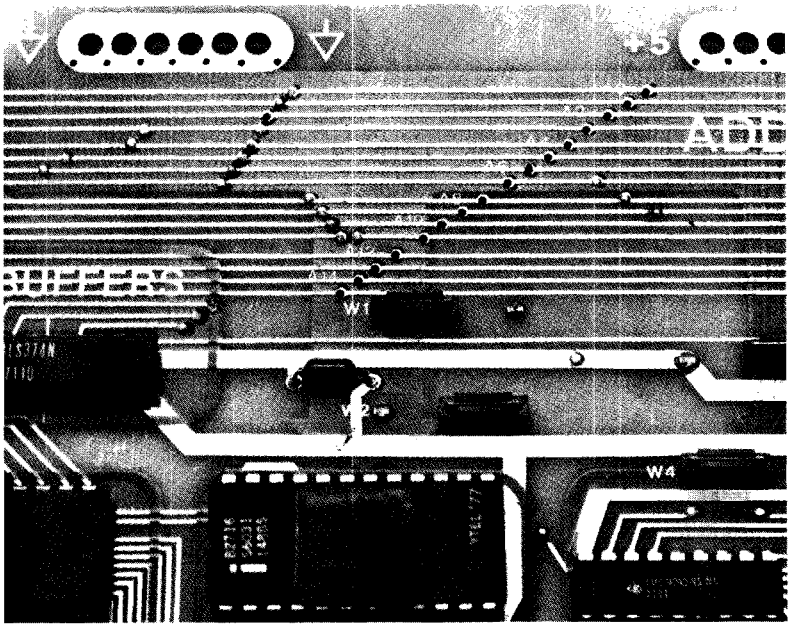


Figure 19-2. Location of Fault Jumper W1 and Address Bus Test Probe Holes

(Continued)

II. Diagnosing the Fault

A) The μLab has power-up, self-test, and diagnostic capabilities designed in, so try turning it off and then on again. Maybe it will tell you what is wrong by displaying a message.

B) No such luck. Press [RESET] . Still nothing. Press some other keys. Now try to see how bad it is. Are any lights on? Yes. That means the power supply is probably all right. Are the address and data buses doing anything? Yes. Do not always rely on the LEDs on these buses to show you activity. You cannot see them blinking if the frequency is very far above 30 Hz, a pretty slow rate for a microprocessor.

C) Now is the time to reach for the logic probe. Connect its power leads to the power slots at the top edge of the board (see Experiment 16-1). The probe tip should light. Set the slide switch on the probe to the TTL position.

D) Place the probe tip into test probe hole A0 (refer back to Figure 19-2). The flashing tip tells you there is activity on the A0 line. The probe converts the fast activity (data stream) at its tip to a rate slow enough for you to see.

E) To determine whether there is bus activity, probe the other 15 address lines, as well as the data lines D0 through D7.

F) Since there is activity on all of these lines, you know that none of them is stuck high or low. This tells you that the microprocessor is trying to do something.

G) It is apparent that the microprocessor cannot make the whole system operate, but maybe the internal signature analysis test loop program available on the μLab will run. Try running this program by sliding the μLab's SA slide switch up and then down. If all the display segments and output LEDs had lighted and the speaker had beeped, then the SA loop would have been engaged. But not a flicker or a sound was generated. Read on to see what this means.

CHECKPOINT 1

Problems:

- The whole system does not run.
- The display does not do anything useful, nor does it appear to respond to the keys.
- The SA test program will not run, even though it requires even fewer system circuits to work.

On the bright side:

- Because there is some bus activity, you know that the microprocessor is running (to some extent).

- The power supply appears to be good. A quick check of its actual voltage would confirm this.

Conclusion: The problem appears to be somewhere between the microprocessor and something connected to the bus. Unfortunately, this still involves most of the circuitry.

III. Partitioning the System

A) Start by isolating the microprocessor. Open the data bus. It is the only major feedback path in the system. To open the bus, slide the eight bus switches in the DIP package, located just below the microprocessor (labeled BUS SWITCH), to the "up" (FREE-RUN) position. This disconnects the microprocessor from all of the other devices on the data bus and allows it to run "open-loop."

B) Now that the data bus is open, the microprocessor should be put into a free-running mode, which allows it to cycle continuously through the full address field. Slide the switch marked FR (next to the SA switch) up to hardwire in an instruction to the microprocessor that forces free-running to occur.

C) Probe several of the data lines and observe the visibly flashing high-order address line LEDs to verify free-run activity. This bus activity should be no surprise since even less of the system is being exercised than before.

CHECKPOINT 2

What have you done?

- Given up trying to get the system to run a program.
- Isolated the microprocessor kernel from most of the rest of the system and allowed it to run in a free-running open-loop mode.

Now what can you do?

- You can examine the microprocessor by itself.
- You can see it cycle through the address field.
- You can evaluate what effect the address bus has on the ROM, the address decoder, and other circuits.
- You can look at the data bus without regard for its effect on the microprocessor, since the processor no longer has any instruction or data path going back to it (feedback).

What tools are needed to accomplish the above? Remember, you need to analyze long, complex data streams that are present on the logic nodes of these devices. This free-run cycle is 2^{16}, or 65,536 cycles, long!

EXPERIMENT 19-1

(Continued)

IV. Examining the Logic Nodes

A) The signature analyzer discussed in Lesson 17 is one such tool. By compressing a long, complex serial data stream into a four-digit "signature," the signature analyzer makes it easy to tell a good node from a bad one.

B) Connect the signature analyzer as shown in Figure 19-3 and turn it on. Set its front panel switches as shown.

C) The signature analyzers' probe can be used as a logic probe by observing the lamp in its tip. Touch the probe to the ground (⏚) slot. Observe that the display reads 0000, as it always should when the probe is connected to ground. Now touch the probe to Vcc. The signature on the display should read 0001. If it doesn't, check the set-up and the switch positions on the μLab (DATA BUS and FR switches up) and the signature analyzer. Press RESET if there appears to be no bus activity.

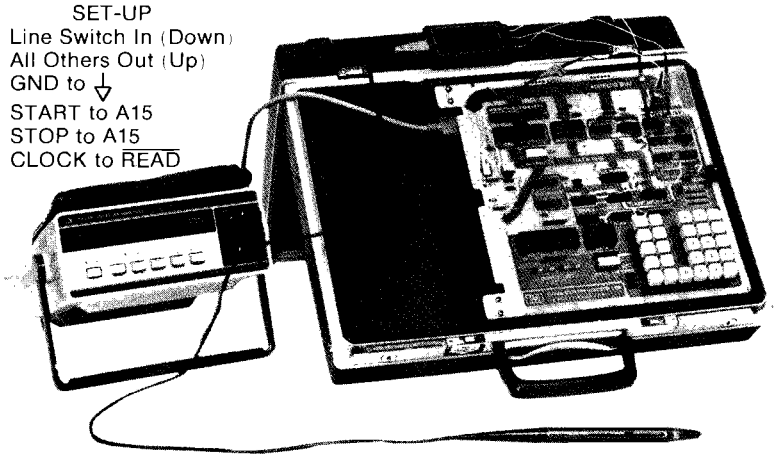


Figure 19-3. Signature Analysis Set-Up for Free-Run Address Test Mode

CHECKPOINT 3

What does this signature tell us?

- The signature analyzer is properly set-up and connected.
- The microprocessor is free-running correctly in an open-loop mode because the data bus is no longer connected to it. The kernel is running.
- We can now use the signature analyzer to check signatures on logic nodes stimulated by the free-running microprocessor.

V. Taking Signatures

A) Consult Table C-1 and take signatures on all of the address lines using the probe holes provided. They should all agree with those in the table.

B) It appears that the A11 address line is faulty because it produces the same signature as A10. What does this mean? Could they be shorted together?

C) If you have not yet opened up the μLab's schematic, do so now. Notice that the schematic is laid out similarly to the μLab's circuit board. Now locate the points connected to A10 and A11 on the schematic.

D) Observe that A10 and A11 are not connected directly to the microprocessor but go through buffer U1. A short between A10 and A11 on either side of buffer U1 could be causing the bad signature.

E) Now check the signatures on U1 pins 4 and 2 (the inputs to the A10 and A11 buffers) against their proper values from Table C-1. They are correct.

F) Check signatures on U1 pins 5 and 3 (the outputs of the buffer) to see if U1 is at fault. They are also correct. The fault must be between the U1 outputs (pins 5 and 3) and the address bus test points A10 and A11.

G) Follow the trace with the signature analyzer probe going from U1-3 toward the A11 test point. This should lead you to fault jumper W1. For the time being, leave this fault set.

H) The fault has been found and no additional signatures need to be taken. However, to satisfy your own curiosity, look at the signatures on other nodes specified in Table C-1 and see which ones are affected by the A11 fault.

Before restoring fault jumper W1 to its proper position, you can take advantage of the A10 to A11 bus line short to demonstrate the use of the current tracer with the logic pulser.

VI. Current Tracing

A) Connect the power leads of the current tracer and the logic pulser to the power slots on the μLab board.

B) Press the logic pulser switch down and slide it forward. The tip should be flashing rapidly indicating that it is in the 100 Hz mode. If it is not, slide the switch back, release and repeat. The pulser is now set to inject a pulsing current into a node.

(Continued)

C) Touch the pulser tip to the A11 line at the location shown in Figure 19-4. The pulser is now injecting current pulses into this address line.

Figure 19-4. Pulsing Current into A11 Address Line

D) With the pulser still on the A11 node, place the tip of the current tracer at right angles to and touching the tip of the pulser (see Figure 19-5). Make sure that the adjustment knob on the current tracer is in line with (parallel to) the pulser tip. Now adjust the knob on the current tracer to cause its lamp to glow dimly.

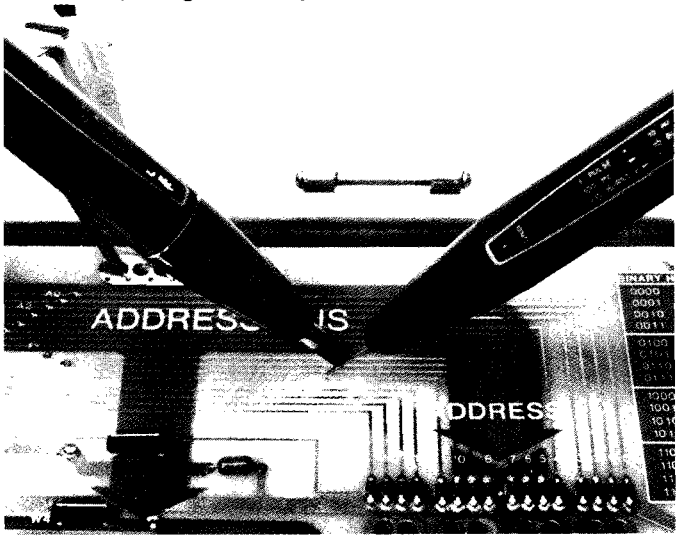


Figure 19-5. Adjusting Current Tracer Sensitivity

E) The current tracer is now adjusted to detect the current injected into the node by the pulser. Follow the current from the pulser tip onto the A11 line by moving the current tracer. Determine in which direction the current goes (left or right of the pulser tip) once the tracer contacts the circuit board. Remember that the brighter the lamp, the stronger the current. Be sure to keep the two holes on the side of the tip in line with the A11 circuit trace and the tip at right angles to the board.

F) Hopefully you chose to go toward the left and found that the lamp mysteriously dimmed once you came to the point shown in Figure 19-6. As you know, current never disappears into thin air nor does it fall through a hole in the board. It must therefore be flowing through the plated-through hole to the other side of the board.

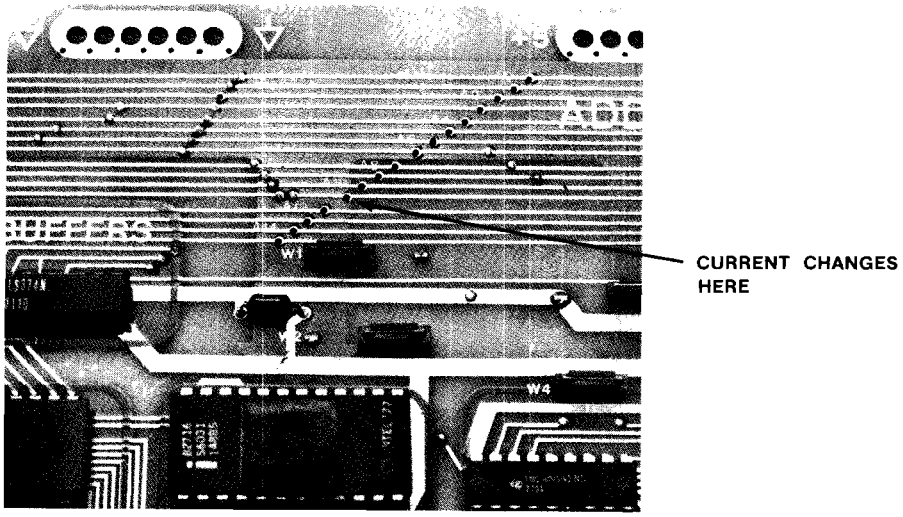


Figure 19-6. Point at Which Current Changes

G) Check this by lifting the board and continuing to trace the current on the rear side. A firm tug on the black knob located near the center of the right edge of the μLab PC board will release the board and allow it to swing up. If you allow the board to swing past the vertical position, it will hold itself up.

H) The current tracer should lead you through fault jumper W1 to the A10 line. You have now found the problem.

I) Restore fault jumper W1 to its original position (left two holes). You may wish to verify the signatures on the A10 and A11 lines.

J) Slide the free-run (FR) switch down to its normal mode and do the same for the data bus switches.

EXPERIMENT 19-1

(Continued)

K) Press RESET once or twice and observe the display message µLAb UP.

L) Turn the power off and then on to verify that the power-on self-test operates properly and produces the same message.

SUMMARY

In this experiment a fault was inserted by means of a jumper. Visual observations of the µLab indicators, internal self-test features, and the logic probe were used to eliminate some of the possible fault sources. Since no program would run, the major feedback path (the data bus) was opened. The microprocessor was then set to the free-run mode, isolating the microprocessor from the rest of the system and preventing the components from interacting with each other. The signature analyzer was used to identify good and bad logic nodes in the free-run test loop. Once the faulty lines were found and the nature of the fault determined (A10 and A11 shorted together), the signature analyzer led you to the actual fault. Even though the fault had been found, the current tracer and the logic pulser were used to illustrate how they could also have led you to it. The pulser injected current pulses into one of the nodes while the current tracer followed that current to the other node. The path joining these two nodes was the fault. On a typical product, this type of fault is often caused by a solder short or gold bridge and can sometimes be very difficult to find visually or by means of voltage tracing techniques.

TROUBLESHOOTING WITH AN OSCILLOSCOPE

A conventional oscilloscope can also be used to locate this type of fault. The procedure is similar to that used with the signature analyzer. The fundamental difference is in the interpretation of the measurements.

In tracing the fault in this experiment, the oscilloscope can be used to verify the general presence of activity on the address, data, and control buses and the device chip enable pins. In the process of doing so, you could also detect identical waveforms on the A10 and A11 bus lines. However, unless these lines were already suspect, it is unlikely that this coincidence would be noticed.

If the system is made to free-run, the oscilloscope can display each of the sixteen address bus lines in turn. Since the frequency of these incrementing address lines should be either double or half that of an adjacent address line, the A10 to A11 short would be obvious on the scope display (see Figure 19-7).

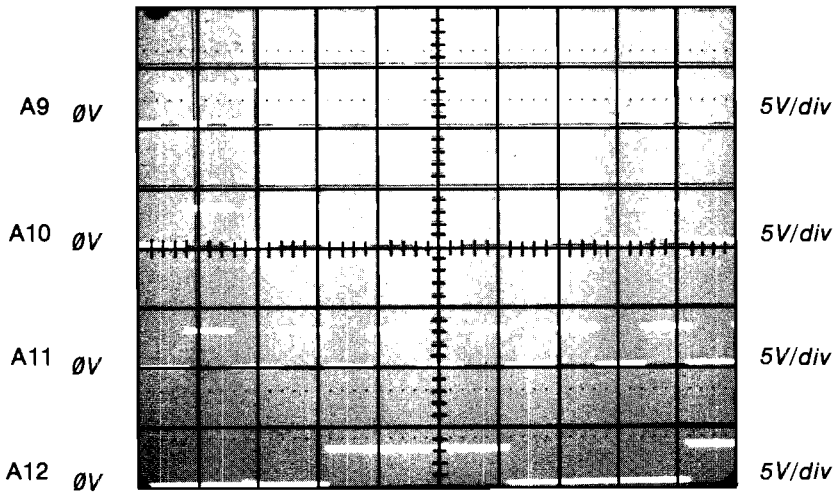


Figure 19-7. Scope Display of A10 to A11 Short

Whereas the signature analyzer presents precise go/no-go information, the oscilloscope's waveform must be analyzed against expected results. It is very difficult to evaluate the long, complex bit streams that are present in microprocessor-based systems on an oscilloscope. Drive, timing, and gross functional failures (e.g., stuck nodes), can however, be easily observed.

TROUBLESHOOTING THE FAULTS

The next step is to find faults on your own. All twelve fault locations have three socket pins with a jumper plug connecting the middle pin to one of the end ones (see Figure 19-8). One jumper position is normal, and the other introduces a circuit fault. The jumpers do not appear on the schematic. Jumpers should always be installed in one of the two hole pairs. A small dot appears on the underside of the circuit board beneath the normal (fault-free) jumper position. Use these dots as a "last resort" means for restoring the μLab to its normal operating mode.

These fault jumpers simulate such real-world circuit malfunctions as shorted and open traces, stuck and open outputs, and functional failures within the ICs. In some cases, a fault jumper actually causes more than one change to occur. For example, some faults open one trace and short it to another to prevent potentially

destructive component overloads from occurring. While faults that do so are not as realistic as the others, they are just as instructive for learning how to troubleshoot.

You can either introduce a fault jumper yourself or have someone else do it for you (so that you won't know which one it is). For those who wish to increase the challenge, multiple faults can be set. It is helpful to refer to the board trace diagram of the μLab at the back of the book so that circuit traces, covered by components, can be followed during the course of troubleshooting.

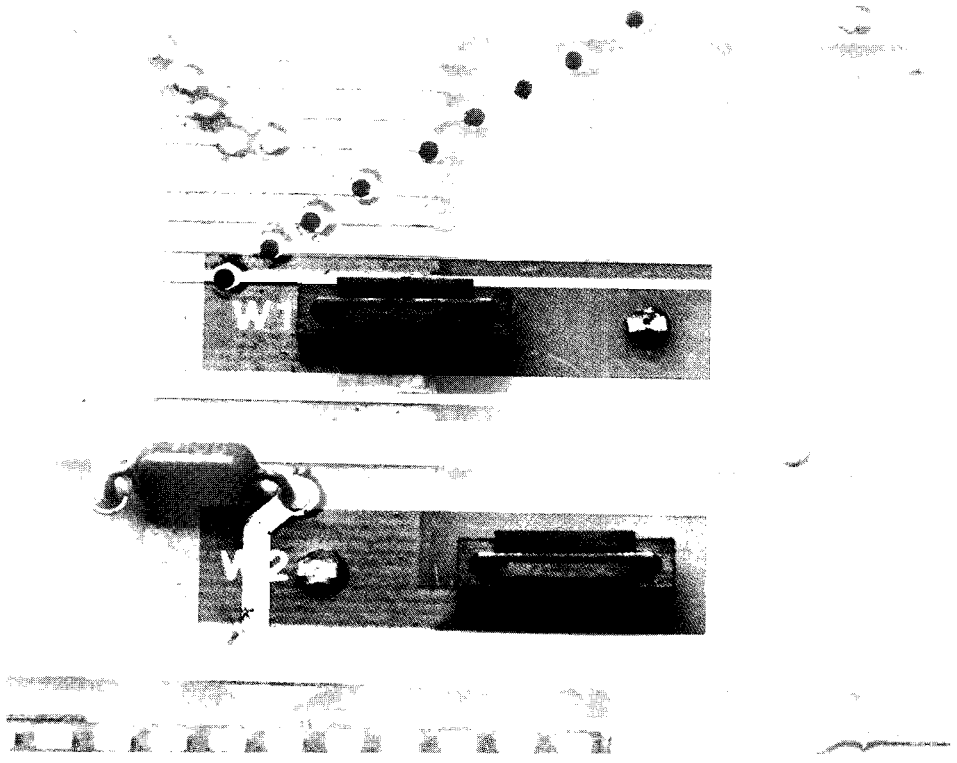


Figure 19-8. All Twelve Fault Jumpers Have Two Positions: Normal and Faulty. A dot on the rear of the board indicates the normal (fault-free) position.

As with real circuit failures, some faults are easier to troubleshoot than others. If you have difficulty with a particular fault, you may wish to skip it and return to it after you have found others and gained more confidence in your technique. Do not be afraid to check the solutions if you are really stumped. You can learn more from the solutions than from floundering around pointlessly. The solutions can show you where you may have taken a wrong turn, made an incorrect observation, or missed taking a measurement. After you have successfully found a fault, you may wish to compare your solution with the one documented in Appendix A.

There are two types of fault solution descriptions:

- The simplified path taken on the Microprocessor Lab Troubleshooting Diagram.
- A more detailed description of the troubleshooting process, using the Microprocessor Lab Troubleshooting Diagram.

Both solutions make use of signature analysis.

ATYPICAL FEATURES OF THE μLAB

There are some relevant differences in the nature of the faults on the μLab and those that might occur in other microprocessor products. None of μLab faults are marginal, are caused by bad components, connectors, or power supply, or are hard to spot visually. All possible faults are located on a single PC board. The μLab contains a relatively small number of devices and no analog circuits whatsoever.

Keep in mind that the μLab is a teaching tool. A number of the features designed into it for this purpose are not present in most microprocessor-based products. The liberal use of LEDs to indicate bus and control functions on the μLab cannot be expected on other products. In addition, the ability to manually enter and step through programs is very uncommon in dedicated applications. The easy-to-follow placement of devices and bus lines and the liberal use of graphics on the PC board of the μLab is not consistent with the economic and packaging constraints of larger, noneducational products.

In the μLab, the microprocessor is the essence of the product. In most other applications, the peripherals define the products' identity; the microprocessor acts merely as the controller.

REVIEW

Lesson 19

This lesson related entirely to troubleshooting the μLab. Its troubleshooting tree was discussed, and you were directed through an actual fault solution. The fault jumpers were described, and you were given the opportunity to troubleshoot faults on your own.

QUIZ

Lesson 19

1. If the free-run test loop will not run:
   a. the SA loop will not run either.
   b. a microprocessor control line may be stuck.
   c. the microprocessor may be faulty.
   d. all of the above may be true.

2. The main advantage of the SA test loop over the free-run test mode is that it:
   a. is easier to run.
   b. requires less of the circuitry to run.
   c. takes signatures faster.
   d. exercises more of the circuitry.

3. If the μLab successfully passes the power-up self-test, it can be assumed that:
   a. the entire board is fault-free.
   b. the address and data buses are fault-free.
   c. the kernel and the keyboard are fault-free.
   d. the keyboard and the display are fault-free.

4. The primary purpose of opening the data bus lines between the microprocessor and the rest of the system is:
   a. to reduce bus loading on the microprocessor outputs.
   b. to prevent the microprocessor from being sent instructions from the rest of the system.
   c. to prevent the microprocessor from sending data to the rest of the system.
   d. to do all of the above.

5. Two lines that have the same incorrect signature on them may be:
   a. inputs and outputs of a buffer IC.
   b. shorted to ground.
   c. shorted together.
   d. all of the above.

VI
OTHER MICROPROCESSORS

The Microprocessor Lab uses Intel's 8085 microprocessor. This lesson describes some other widely used microprocessors. While the details may vary greatly from one microprocessor to another, the basic concepts remain the same. Learning about another microprocessor will be relatively easy now that you understand these concepts.

LESSON 20

Microprocessor Survey

THE 8085 IN PERSPECTIVE

The world's first commercial eight-bit microprocessor was the Intel 8008, introduced in 1971. By today's standards it is very slow and difficult to use, but at the time of its introduction, it was a major breakthrough. Two years later the 8080 was introduced and represented a significant improvement in both speed and ease of implementation. The 8080 rapidly became an industry standard. The 8085 is the third generation of this series. It is faster than the 8080 and requires fewer components for a complete system.

Intel did not remain alone in the microprocessor field for long. The 8080's most significant early competitor was Motorola's 6800. The 6800 has a number of features lacking in the 8080 but has not become quite as popular.

Zilog's Z80 is another outgrowth of the 8080. It is also a substantial improvement over the 8080, but is quite different from the 8085. The 8080, Z80, 6800, and some other important processors are described later in this lesson.

FOUR-BIT MICROPROCESSORS

Eight-bit microprocessors are more powerful than is needed for many simple control applications. Four-bit microprocessors, being simpler and less expensive, are often more suitable. Early four-bit microprocessors required external ROM, RAM, and I/O, just as the eight-bit processors. More recent devices include the processor, RAM, ROM, and I/O on a single integrated circuit.

The most significant architectural difference between four-bit and eight-bit microprocessors is that the four-bit machines use separate program and data memory. The program memory (usually ROM) is generally eight bits wide, while the data memory is four bits wide. Eight-bit words are used for the instructions, since a four-bit word permits only sixteen different instructions.

SIXTEEN-BIT MICROPROCESSORS

At the other end of the applications spectrum, there are some situations in which eight-bit processors are inadequate. Sixteen-bit microprocessors, which are architecturally similar to minicomputers, provide increased power for such applications. They provide greatly improved arithmetic capabilities and are well suited to "number crunching" applications. More recent sixteen-bit microprocessors provide many sophisticated features to facilitate the implementation of operating systems and high-level languages.

Sixteen-bit microprocessors can do computations involving large numbers much faster than eight-bit processors because they handle sixteen bits at a time. Eight-bit processors can handle equally large numbers, but they must break them up into eight-bit sections, which significantly slows the computation.

SINGLE-CHIP MICROCOMPUTERS

The next advance after the integration of an entire CPU was the inclusion of ROM, RAM, and I/O on the same chip. This has been done for four, eight, and sixteen-bit processors. These processors typically provide 1K to 4K or more bytes of ROM for program storage, and 64 to 256 or more bytes of RAM for temporary data storage. The I/O facilities range from 16 to 32 lines. Some of the more advanced devices also include a UART (serial I/O controller) on the chip. These chips provide a complete microcomputer system on a single IC and are a simple, inexpensive solution to many control applications. Often an entire board with dozens of SSI and MSI ICs can be replaced with a single IC.

Some single-chip microcomputers are available with a UV EPROM instead of the ROM. This facilitates system prototyping, since software changes can be made by reprogramming the EPROM. When the design is finalized, a less expensive mask ROM version can be used.

BIT-SLICE PROCESSORS

Bit-slice processors differ substantially from other microprocessor types. Each bit-slice processor chip has a relatively small word size (usually four bits), but they can be stacked together to make as long a word as desired. For example, four four-bit "slices" can be used to form a sixteen-bit processor.

Another distinguishing characteristic of bit-slice processors is that they do not have a fixed instruction set. The user must write the *microprogram* that defines the instruction set. This allows the instruction set to be customized for each application, resulting in fast, efficient programs. However, it greatly increases the design effort required to build a system.

Bit-slice processors are bipolar devices, while general-purpose processors are MOS devices. Bit-slice processors are therefore much faster and consume more power. They also require faster memories to keep up with them.

As this brief discussion suggests, bit-slice systems are very powerful and also quite complicated. The CPU generally contains dozens of ICs, and the programming is quite complex. They are used in applications that demand this speed and power, such as digital signal processing, high-speed arithmetic processors, high-speed control systems, and minicomputers.

MICROPROCESSOR DESCRIPTIONS

More than thirty general-purpose microprocessors and about the same number of single-chip microcomputers are now available. (Refer to the bibliography for more information on the available devices.) In this section a few of those that have been used in high volume are described: the 8080A, the Z80, the 6800, the 6500 series, the F8, and the TMS 1000.

The Intel 8080

Figure 20-1 shows the basic 8080 CPU, the predecessor to the 8085. The 8080 requires three chips to provide the functions supplied by the 8085 itself: a clock generator (8224), CPU (8080), and system controller (8228). The 8080A does not multiplex the address and data lines, but it does multiplex the control signals and the data. The 8228 controller IC demultiplexes the control information. The 8080's maximum clock rate is 2 MHz as compared to the 8085's 3 MHz. (As with most processors, selected higher speed versions of the 8080 and 8085 are available.) The 8080 requires three power supplies (+5, -5 and +12 V), whereas the 8085 requires only +5 V.

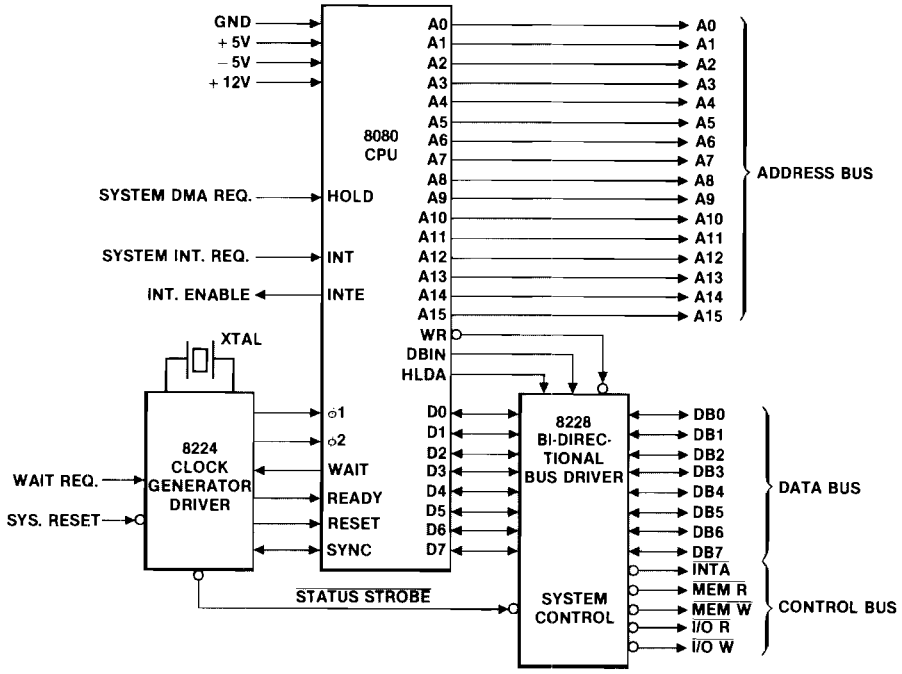


Figure 20-1. Three-Chip 8080-Based CPU

The 8080 lacks the RST 5.5, 6.5, and 7.5 interrupts, as well as the TRAP input. It also does not have any serial I/O pins. The instruction set is identical to the 8085's, except that the 8085 includes two additional instructions to control the interrupts and the serial I/O. All 8080 programs will run on the 8085, but the timing is different. Nearly all new designs use the 8085 rather than the 8080.

The Zilog Z80

Zilog's Z80, like the 8085, is a descendant of the 8080. However, it has many features not found in either the 8080 or the 8085. Like the 8085, it requires no clock generator or system controller chip and operates from a single +5 V supply. Unlike the 8085, its instruction set has been greatly expanded. It includes all of the 8080 instructions (so that it can run 8080 programs), but it also includes a large group of powerful new instructions. For example, a block of data of any size can be moved from one part of memory to another with a single instruction. Any bit of any register can be tested, set, or cleared with a single instruction. These are just two of the many additional instruction types which can ease the programmer's task considerably. Relative and indexed addressing (described below in the 6800 description) were also added. The Z80 includes all of the 8080's registers (B, C, D, E, etc.) but has two complete sets of them.

This is far from a complete description of the Z80's features, but it illustrates the basic concept: the Z80 is an 8080-type processor with a number of hardware simplifications and greatly improved software capabilities.

The Motorola 6800

The 6800 microprocessor was the 8080's first direct competitor and is widely used. However, it differs from the 8080 and its descendants in a number of important ways.

The 6800 operates from a single + 5V supply and does not require a system controller chip. It does, however, require a non-TTL compatible, two-phase clock and therefore requires a special clock generator circuit. Its standard maximum clock frequency is 1 MHz (faster versions are available), but this cannot be directly compared with the 8085's 3 MHz clock. While the 8085 requires from three to six clock cycles for each memory reference, the 6800 requires only one. A 1 MHz 6800 may therefore actually operate faster than a 3 MHz 8085.

The 6800's read and write control signals are quite different from the 8085's. Figure 20-2 shows the timing for a read operation. The clock consists of two nonoverlapping signals, Φ1 and Φ2 (this is called a *two-phase clock*). Shortly after the rising edge of Φ1, the address and R/$\overline{W}$ lines are set up. Valid Memory Address (VMA) goes high at this time, indicating that the address bus contains valid information. Φ2 signals the addressed device to place the data on the bus, and, at the falling edge of Φ2, this data is stored in the microprocessor.

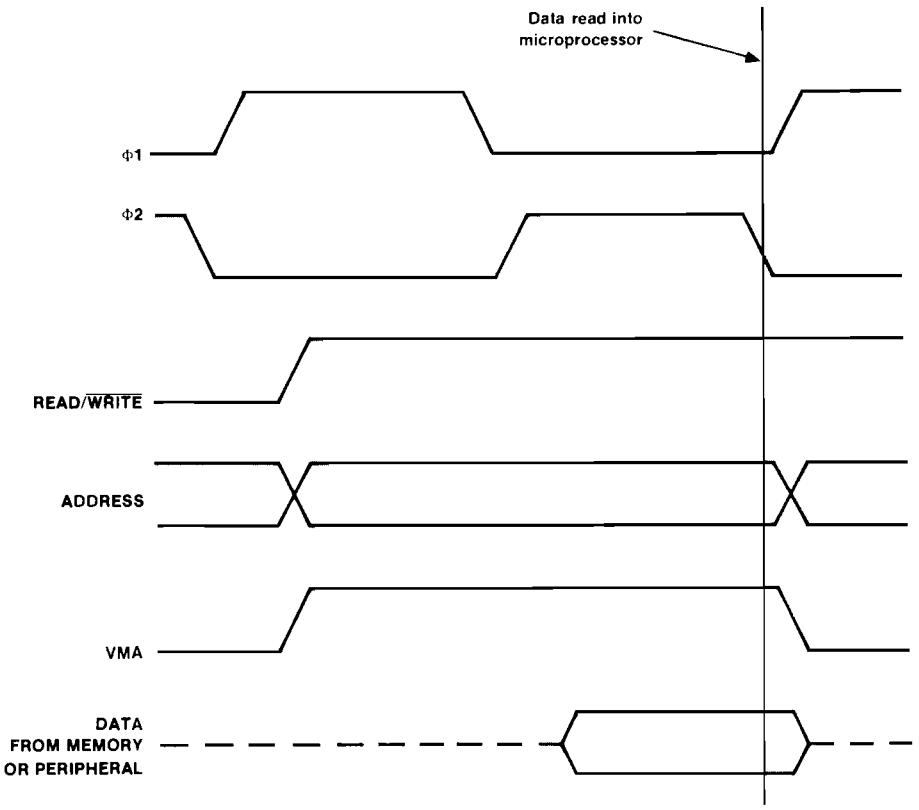


Figure 20-2. 6800 Control Signals, Showing a Read Operation

Note that the meaning of the R/$\overline{W}$ signal is different from the 8085's $\overline{READ}$ and $\overline{WRITE}$ signals. The R/$\overline{W}$ signal does not indicate the exact time at which the transfer occurs, but only specifies the direction. The timing for the transfer is provided by the Φ2 clock. This is quite different from the 8085's timing, in which the $\overline{READ}$ and $\overline{WRITE}$ signals specify both the direction and the timing, and the clock rate is faster than the data transfer rate.

Figure 20-3 shows the 6800's write timing. It is identical to the read timing except that R/$\overline{W}$ is low instead of high. The microprocessor places data on the bus when Φ2 goes high, and, at the falling edge of Φ2, the addressed device stores this data.

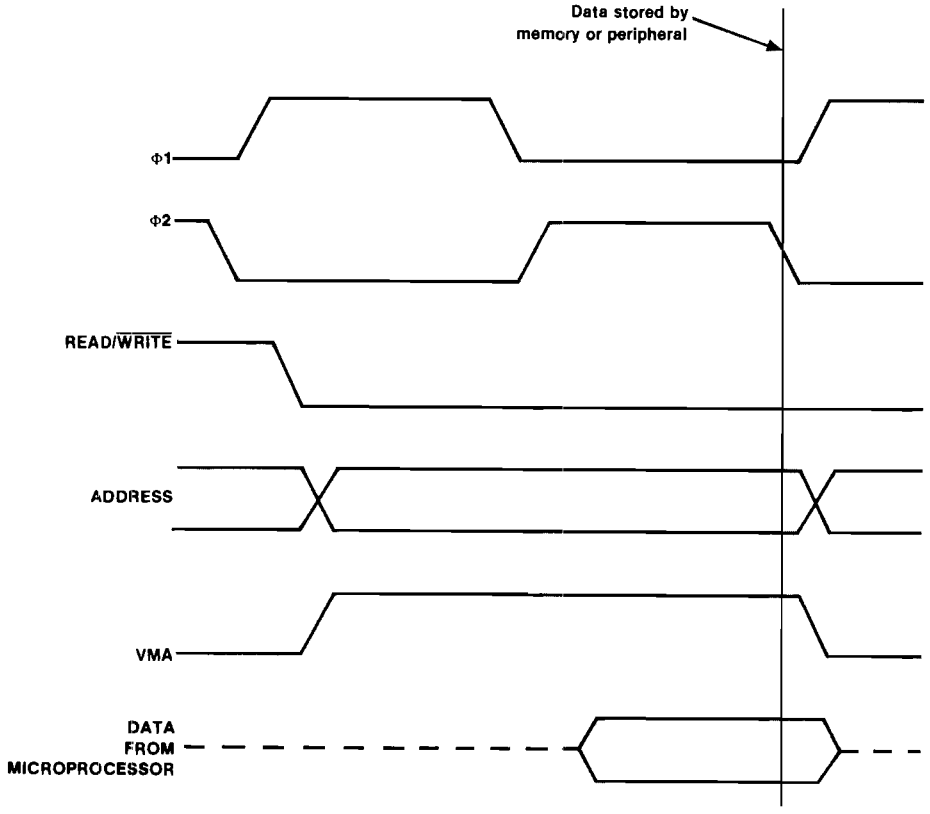


Figure 20-3. 6800 Write Operation

6800 Software

The 6800's instruction set differs from the 8085's in several important ways. The 6800's registers are shown in Figure 20-4. Like the 8085, it has a stack pointer and a program counter. It also has a sixteen-bit register, called the index register, and two eight-bit accumulators. There are no general-purpose registers (other than the accumulators).

Because of the lack of general-purpose registers, external RAM must be used for most temporary data storage. This simplifies the instruction set, since the variations of each instruction needed to refer to each register are unnecessary. The second accumulator can be used for operands or results.

The greatest difference between the software power of the 8085 and that of the 6800 is the use of three additional addressing modes in the 6800: indexed, relative, and direct. *Indexed addressing* uses the index register just described. In this

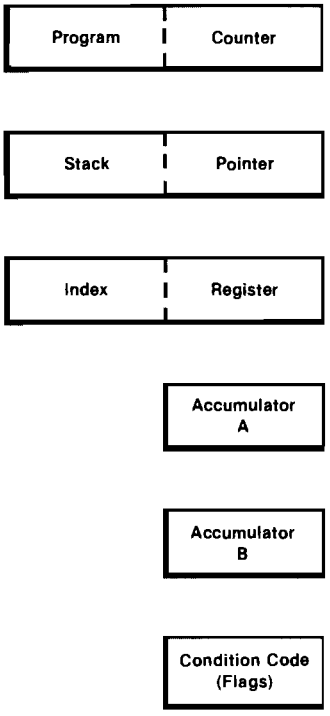


Figure 20-4. 6800 CPU Registers

mode, the effective address (the address sent to the memory) is the sum of the offset specified in the instruction and the contents of the index register. The index register can be set, incremented, or decremented just like the other registers. This addressing technique is useful for handling blocks of data and performing table-oriented operations.

Relative addressing allows you to specify an address relative to the current instruction. You can, for example, write an instruction that causes the processor to jump ahead three addresses or jump backwards seven addresses. The 8085, on the other hand, always requires you to specify the actual address (called *absolute addressing*).

The *direct addressing* mode allows you to specify only the low-order byte of the address, with the upper byte assumed to be zero. This shortens by one byte any instruction that references the first 256 memory locations.

The 6802 is a newer version of the 6800. The 6802 includes 128 bytes of RAM and a clock generator on the CPU chip. The instruction set is identical to the 6800's. Several other descendants of the 6800 also exist.

The 6800 has been only briefly described here, but this discussion should give you an idea of the differences between the 6800 and the 8085. Complete information may be obtained from the Motorola literature.

The MOS Technology 6500

The 6500 series is widely used, particularly in high-volume consumer applications. The 6502 is the standard model.

The 6502's instruction set is similar to the 6800's, but it has only one accumulator and a few additional addressing modes. A significant hardware difference is the inclusion of an on-chip clock generator.

The 6502 microprocessor is available in several versions in 28-pin (instead of 40-pin) packages, which reduces the cost. The difference between the versions is the selection of deleted pin functions, since twelve pins must be eliminated. Some eliminate the interrupts and a few address lines, while others eliminate mostly address lines. This allows small systems to be implemented with an inexpensive microprocessor that has only the functions and addressing range required for the application.

The Fairchild F8

The F8 microprocessor is a two-chip system. The 3850 CPU and 3851 Program Storage Unit (PSU) combine to make a complete microcomputer system with 64 bytes of RAM, 1K bytes of ROM, 32 I/O lines, and a timer.

The most unusual aspect of the F8 system is the division of functions between the 3850 CPU and the 3851 PSU. Figure 20-5 shows the block diagram. The 3850 CPU does not have an address bus; the program counter (PC) and memory addressing logic are part of the 3851 memory device. The CPU generates five

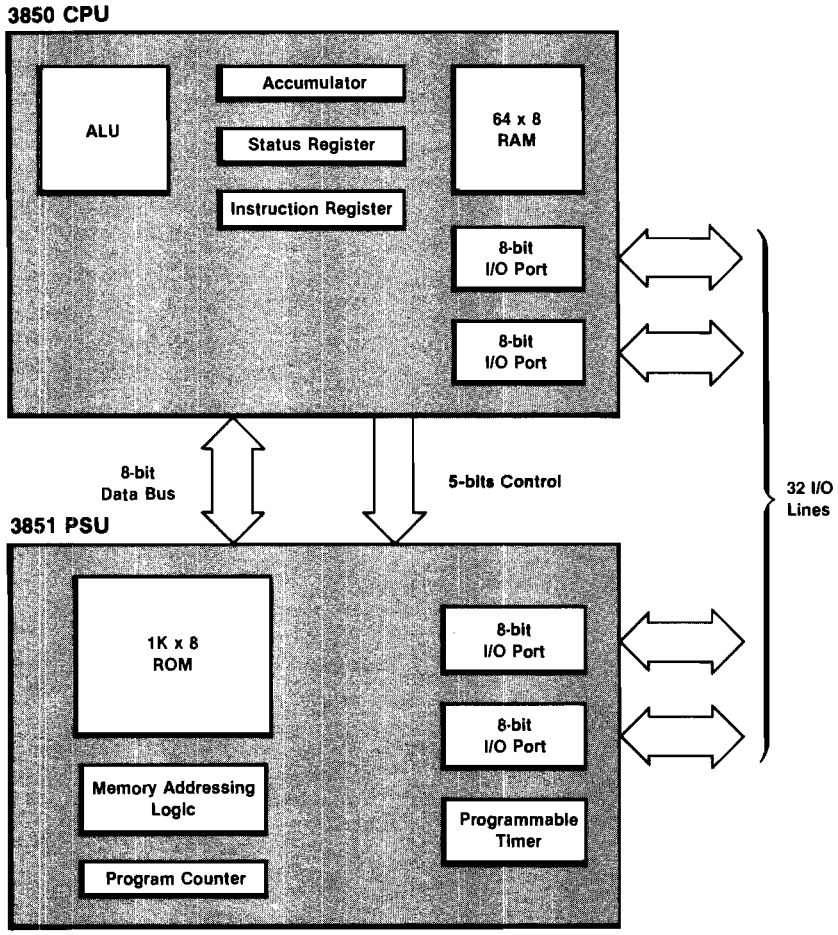


Figure 20-5. Simplified F8 System Block Diagram

special control signals to control the program counter in the 3851. The CPU can set the address in the PC by sending the address in two parts over the data bus, and instructing the PSU to load that data into the PC. It can also instruct the PSU to increment the PC. If there is more than one PSU in the system, each maintains its own PC. All the PCs are synchronized by the control signals from the CPU. The elimination of the address bus makes sixteen pins available on both the CPU and PSU packages, which are used to provide 32 bits of I/O.

There are a number of other members of the F8 family. The 3856 is a 2K byte version of the 3851. There are also special interface chips for using standard static or dynamic memories with the 3850. The 3870 is a single-chip version of the F8 that incorporates the functions of the 3850 and 3856, providing the processor, 2K bytes of ROM, and 64 bytes of RAM on the same chip. The 3871 is a single-chip processor with 4K bytes of ROM and 132 bytes of RAM.

The F8 is widely used in control applications, including games, appliances, and instruments. The single-chip version provides a very powerful system with a single IC.

The Texas Instruments TMS 1000

The TMS 1000 is a family of single-chip four-bit microcomputers. Each device contains the CPU, ROM, RAM, and I/O on a single IC and is very inexpensive in large quantities. There are about 35 varieties available, with varying amounts of ROM, RAM, and I/O. The ROM is 1K or 2K bytes, and the RAM is either 64 or 128 four-bit words. Up to sixteen I/O lines are available. These devices are widely used in high-volume, low-end applications.

REVIEW

Lesson 20

A wide variety of microprocessors is now available. The general-purpose eight-bit types, such as the 8085 and 6800, provide a great deal of flexibility with a moderate amount of complexity. For small systems, single-chip microcomputers such as the 3870 or TMS 1000 provide an extremely simple and inexpensive system. At the high end, the sixteen-bit microprocessors provide minicomputer-like performance for more demanding applications. For very high-speed applications, bit-slice processors are used.

All microprocessors share a common basic design: a CPU to fetch and execute instructions, memory to store instructions and data, and I/O ports to communicate with other devices. The various microprocessors have different instruction sets, using different registers and addressing modes. Most applications can be implemented using any of several different processors, though the amount of ROM and RAM required may vary considerably. There is always a data bus and an address bus, though they are sometimes hidden within an IC. The control signals may vary somewhat, but they always provide the same functions: telling the memory and I/O when they should read or write data and allowing other devices to interrupt or halt the processor.

When learning about a new microprocessor, look for the features that have been discussed. How many bits wide is the data bus? What are the registers in the CPU? What addressing modes are available? Are the address (or control) and data multiplexed? How many ICs are required to make a complete system? What kinds of support chips are available?

The variety and sophistication of microprocessors is rapidly increasing. Once you understand the basic concepts and a few of the current microprocessors, you will be able to put the new devices in perspective and keep up with this rapidly advancing field.

QUIZ

Lesson 20

1. The 8085 and Z80 evolved from the _____.

2. The 6502 and 6802 evolved from the _____.

3. Four-bit processors are used (instead of eight-bit processors) because:
   a. they are faster.
   b. they can access more memory.
   c. they are easier to program.
   d. they are less expensive.

4. Sixteen-bit processors are used (instead of eight-bit processors) because:
   a. they are less expensive.
   b. they consume less power.
   c. they can handle a greater range of values.
   d. they can perform calculations with large numbers more quickly.

5. Single-chip microcomputers generally include:
   a. ROM, RAM, and I/O ports.
   b. ROM and RAM, but require I/O ports.
   c. only I/O ports.
   d. RAM and I/O but no ROM.

6. Bit-slice microprocessors are:
   a. easy to use.
   b. inexpensive.
   c. very flexible and powerful.
   d. very similar to standard processors.

7. The memory control signals of different microprocessors:
   a. are all the same.
   b. vary in details, but always perform the same function.
   c. can perform widely varying functions.
   d. are always identical to 8085 control signals.

APPENDICES

APPENDIX A

Solutions to Problems

This appendix contains solutions to all the problems in the text. There are four types of solutions:

- Answers to quizzes
- Solutions to programming exercises
- Machine code for programs used in experiments
- Solutions to troubleshooting faults

The fault solutions beginning on page A-7 assume that only one fault is present in the μLab. They outline the steps taken to locate the faults using the μLab Generalized Troubleshooting Flowchart (Figure 19-1). The troubleshooting path taken for each of the faults is shown on this flowchart. You may wish to refer to them before reading the actual fault solutions. In these solutions, specific ICs or circuit areas are often called out within a signature test table based on a modest level of circuit insight. These "educated guesses" are done to save time probing all signature points in a table. Such a procedure is not, however, necessary to pinpoint bad signatures.

ANSWERS TO QUIZZES

Lesson 1

1. c
2. b
3. address, data, and control
4. bytes
5. b
6. d
7. a
8. d
9. a

Lesson 2

1. 1 or 0
2. 172
3. 254
4. AC
5. 1011 1001
6. FF (hex) 377 (octal)
7. c

Lesson 3

1. c
2. assembly, machine
3. b
4. d
5. a

Lesson 4

1. FETCH ADRS
2. DECR or FETCH ADRS 0 8 0 2
3. a
4. addresses
5. b

Lesson 5

1. a
2. c
3. accumulator (registers)
4. CALL
5. RET
6. b
7. a

Lesson 6

1. fetch
2. c
3. hardware
4. c
5. b

Lesson 7

1. data, address/control
2. b
3. a
4. b
5. c
6. c
7. d
8. $\overline{\text{WRITE}}$, $\overline{\text{READ}}$

Lesson 8

1. a
2. d
3. c
4. b
5. a
6. c

Lesson 9

1. c
2. c
3. a
4. d
5. c
6. c
7. d
8. d

Lesson 10

1. b
2. a
3. d
4. a
5. d
6. b
7. c
8. a

Lesson 11

1. b
2. d
3. STORE/INCR three times
4. a

Lesson 12

1. b
2. a
3. c
4. 0930
5. c

Lesson 13

1. problem
2. c
3. a
4. d

Lesson 14

1. c
2. b
3. b
4. c

Lesson 15

1. d
2. b
3. floating point
4. a
5. b

Lesson 16

1. b
2. c
3. d
4. d
5. a
6. b

Lesson 17

1. d
2. d
3. d
4. c
5. b
6. d
7. a
8. b

Lesson 18

1. a
2. d
3. d
4. d
5. a
6. c

Lesson 19

1. d
2. d
3. b
4. b
5. d

Lesson 20

1. 8080
2. 6800
3. d
4. d
5. a
6. c
7. b

PROGRAMMING EXERCISE SOLUTIONS

Note: These solutions are for the programming exercises only.
For programs used in experiments, see the following section.

Programming Exercise 12-1: Masking

Address	Contents	Label	Instruction		Comments
0800	3A	START	LDA	2000	;Read input port to accumulator
0801	00				
0802	20				
0803	06		MVI	B,08	;Set B to mask value (0000 1000 binary)
0804	08				
0805	A0		ANA	B	;Set all bits except no. 3 to zero
0806	CA		JZ	OFF	;Test for accumulator = 0
0807	11				
0808	08				
0809	3E	ON	MVI	A,0	;Turn on LEDs
080A	00				
080B	32		STA	3000	
080C	00				
080D	30				
080E	C3		JMP	START	
080F	00				
0810	08				
0811	3E	OFF	MVI	A,FF	;Turn off LEDs
0812	FF				
0813	32		STA	3000	
0814	00				
0815	30				
0816	C3		JMP	START	
0817	00				
0818	08				

Programming Exercise 12-2: Rotates

Address	Contents	Label	Instruction		Comments
0800	3E		MVI	A,FE	;Set accumulator to 1111 1110
0801	FE				
0802	32	LOOP:	STA	3000	;Write to output port
0803	00				
0804	30				
0805	0F		RRC		;Rotate data
0806	C3		JMP	LOOP	;Repeat
0807	02				
0808	08				

Using Breakpoint in Exercise 12-2

Address	Contents	Label	Instruction		Comments
0800	3E		MVI	A,FE	;Set accumulator to 11111110
0801	FE				
0802	32	LOOP:	STA	3000	;Write to output port
0803	00				
0804	30				
0805	CF		RST	1	;Breakpoint
0806	0F		RRC		;Rotate data
0807	C3		JMP	LOOP	;Repeat
0808	02				
0809	08				

Programming Exercise 13-1a: Traffic Light Controller with Individually Variable Green and Yellow Times

Main Program

Address	Contents	Label	Instruction		Comments
080C	2E	TRAF:	MVI	L,1	;Set yellow time A
080D	01				
080E	16		MVI	D,6	;Set green time A
080F	06				
0810	1E		MVI	E,0	;Sequence signal A
0811	00				
0812	CD		CALL	SEQ	
0813	30				
0814	08				
0815	2E		MVI	L,3	;Set yellow time A
0816	03				
0817	16		MVI	D,10	;Set green time B
0818	10				
0819	1E		MVI	E,1	;Sequence signal B
081A	01				
081B	CD		CALL	SEQ	
081C	30				
081D	08				
081E	C3		JMP	TRAF	
081F	0C				
0820	08				

Sequencing Routine

0830	26	SEQ:	MVI	H,7D	;Set signal green
0831	7D				
0832	CD		CALL	CHNG	
0833	55				
0834	08				
0835	00		NOP		;Wait green time
0836	00		NOP		
0837	CD		CALL	DELAY	;Wait green time
0838	70				
0839	08				
083A	26		MVI	H,7B	;Set signal yellow
083B	7B				
083C	CD		CALL	CHNG	
083D	55				
083E	08				
083F	55		MOV	D,L	;Wait yellow time
0840	00				
0841	CD		CALL	DELAY	
0842	70				
0843	08				
0844	C9		RET		

(CHANGE and DELAY routines same as in Table 13-8

PROGRAMMING EXERCISE SOLUTIONS
(Continued)

Programming Exercise 13-1b: Traffic Light Controller with Both Signals Red Between Cycles

Main Program

Address	Contents	Label	Instruction	Comments
080E	16	TRAF:	MVI D,6	Set Green Time A
080F	06			
0810	1E		MVI E,0	;Sequence signal A
0811	00			
0812	CD		CALL SEQ	
0813	30			
0814	08			
0815	00		NOP	
0816	00		NOP	
0817	16		MVI D,10	Set Green Time B
0818	10			
0819	1E		MVI E,1	;Sequence signal B
081A	01			
081B	CD		CALL SEQ	
081C	30			
081D	08			
081E	C3		JMP TRAF	
081F	0E			
0820	08			

Sequencing Routine

0830	26	SEQ	MVI H,7D	;Set signal green
0831	7D			
0832	CD		CALL CHNG	
0833	55			
0834	08			
0835	00		NOP	Wait green time
0836	00		NOP	
0837	CD		CALL DELAY	
0838	70			
0839	08			
083A	26		MVI H,7B	Set signal yellow
083B	7B			
083C	CD		CALL CHNG	
083D	55			
083E	08			
083F	16		MVI D,2	Wait yellow time
0840	02			
0841	CD		CALL DELAY	
0842	70			
0843	08			
0844	26		MVI H,77	;Set both sigs red
0845	77			
0846	CD		CALL CHNG	
0847	55			
0848	08			
0849	16		MVI D,1	;Wait both red time
084A	01			
084B	CD		CALL DELAY	
084C	70			
084D	08			
084E	C9		RET	

(CHNG and DELAY routines same as in Table 13-8

Programming Exercise 13-1c: Traffic Light Controller with Left-Turn Arrows

Main Program

Address	Contents	Label	Instruction	Comments
080E	2E	TRAF	MVI L,6	;Set Green Time A
080F	06			
0810	1E		MVI E,0	Sequence signal A
0811	00			
0812	CD		CALL SEQ	
0813	26			
0814	08			
0815	00		NOP	
0816	00		NOP	
0817	2E		MVI L,10	Set Green Time B
0818	10			
0819	1E		MVI E,1	Sequence signal B
081A	01			
081B	CD		CALL SEQ	
081C	26			
081D	08			
081E	C3		JMP TRAF	
081F	0E			
0820	08			

Sequencing Routine

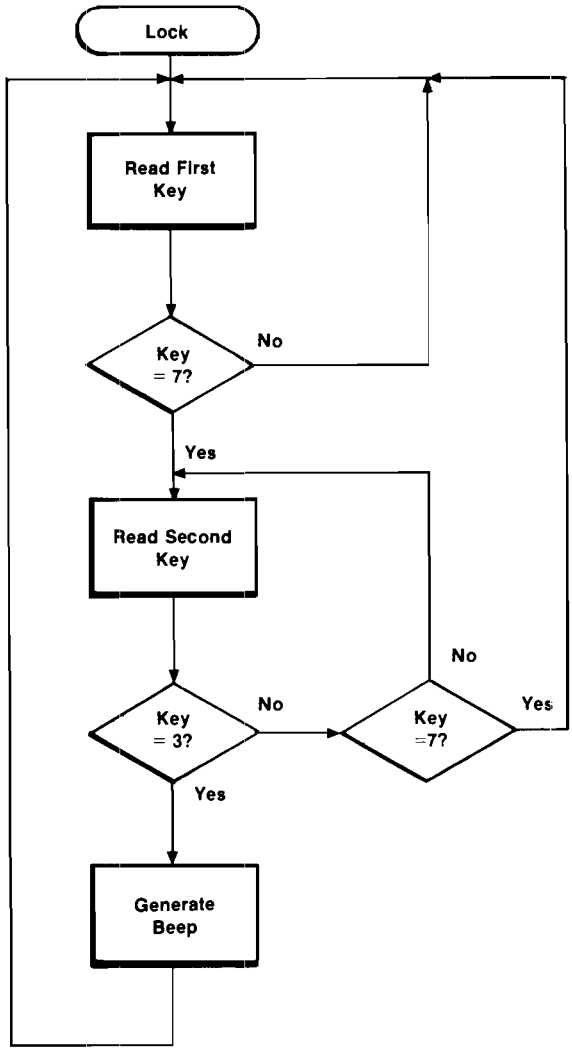
0826	26	SEQ:	MVI H,7E	;Set left-turn indicator
0827	7E			
0828	CD		CALL CHNG	
0829	55			
082A	08			
082B	16		MVI D,3	Wait left-turn time
082C	03			
082D	CD		CALL DELAY	
082E	70			
082F	08			
0830	26		MVI H,7D	Set signal green
0831	7D			
0832	CD		CALL CHNG	
0833	55			
0834	08			
0835	55		MOV D,L	Wait green time
0836	00		NOP	
0837	CD		CALL DELAY	
0838	70			
0839	08			
083A	26		MVI H,7B	Set signal yellow
083B	7B			
083C	CD		CALL CHNG	
083D	55			
083E	08			
083F	16		MVI D,2	Wait yellow time
0840	02			
0841	CD		CALL DELAY	
0842	70			
0843	08			
0844	C9		RET	

(CHNG and DELAY routines same as in Table 13-8)

PROGRAMMING EXERCISE SOLUTIONS
(Continued)

Programming Exercise 14-1: Electronic Lock

Flowchart



Program for Exercise 14-1

Address	Contents	Label	Instruction	Comments
0800	CD	FIRST:	CALL KIND	;Read first key
0801	4B			
0802	01			
0803	FE		CPI 07	;Key = 7?
0804	07			
0805	C2		JNZ FIRST	;Read again if not
0806	00			
0807	08			
0808	CD	SECOND:	CALL KIND	;Read second key
0809	4B			
080A	01			
080B	FE		CPI 03	;Key = 3?
080C	03			
080D	CA		JZ OPEN	;Generate beep if yes
080E	18			
080F	08			
0810	FE		CPI 07	;Key = 7?
0811	07			
0812	CA		JZ SECOND	;Read second key if yes
0813	08			
0814	08			
0815	C3		JMP FIRST	;Read first key if no
0816	00			
0817	08			
0818	CD	OPEN:	CALL BEEP	;Correct combination — generate beep
0819	10			
081A	00			
081B	C3		JMP FIRST	;Repeat
081C	00			
081D	08			

Programming Exercise 14-2: Using the Keyboard and Display

Address	Contents	Label	Instruction	Comments
0800	CD	LOOP	CALL KIND	;Read key
0801	4B			
0802	01			
0803	32		STA 0B00	;Store key in RAM
0804	00			
0805	0B			
0806	11		LXI D,MESS	;Set message address
0807	00			
0808	0B			
0809	CD		CALL SDM	;Move message
080A	18			
080B	00			
080C	CD		CALL DCD	;Display message
080D	E9			
080E	01			
080F	C3		JMP LOOP	;Repeat
0810	00			
0811	08			
0B00	00	MESS:	(Reserved for key data)	
0B01	10		} Blanks	
0B02	10			
0B03	10			
0B04	10			
0B05	10			

PROGRAM FOR EXPERIMENTS

Experiment 12-1:
Program to Demonstrate Logical Instructions

Address	Contents	Label	Instruction	Comments
0800	3A	START:	LDA 2000	;Read input port
0801	00			
0802	20			
0803	06		MVI B,3C	;Set B register to 0011 1100
0804	3C			
0805	A0		ANA B	;AND accumulator with B
0806	32		STA 3000	;Send to output port
0807	00			
0808	30			
0809	C3		JMP START	;Repeat
080A	00			
080B	08			

Experiment 12-2:
Arithmetic Instruction Demonstration Program

Address	Contents	Instruction	Comments
0800	90	SUB B	;Subtract B from accumulator
0801	81	ADD C	;Add C to accumulator

Experiment 14-1:
Program to Read Keyboard and Beep if 7 Is Pressed

Address	Contents	Label	Instruction	Comments
0800	CD	READ:	CALL KIND	;Read key
0801	4B			
0802	01			
0803	FE		CPI 07	;Compare keycode
0804	07			
0805	C2		JNZ READ	
0806	00			
0807	08			
0808	CD		CALL BEEP	;Generate beep if key = 7
0809	10			
080A	00			
080B	C3		JMP READ	
080C	00			
080D	08			

To detect the E key, change location 0804 to 0E.

Experiment 14-2:
Program to Test for 2 Key

Address	Contents	Label	Instruction	Comments
0800	3E		MVI A,F7	;Set scan port to 1111 0111
0801	F7			
0802	32		STA 2800	
0803	00			
0804	28			
0805	3A	READ	LDA 1800	;Read columns
0806	00			
0807	18			
0808	06		MVI B,07	;Mask off all bits except three LSBs
0809	07			
080A	A0		ANA B	
080B	FE		CPI 05	;Is data 101 ("2 key")?
080C	05			
080D	C2		JNZ READ	;If not, read again
080E	05			
080F	08			
0810	CD		CALL BEEP	;Yes-generate beep
0811	10			
0812	00			
0813	C3		JMP READ	;Read again
0814	05			
0815	08			

To detect the 3 key, change location 080C to 3.

Experiment 14-3:
Message Display Program

Address	Contents	Label	Instruction	Comments
0800	11		LXI D,0810	;Set message address
0801	10			
0802	08			
0803	CD		CALL STDM	;Move message
0804	18			
0805	00			
0806	CD	LOOP:	CALL DCD	;Display message
0807	E9			
0808	01			
0809	C3		JMP LOOP	
080A	06			
080B	08			

Experiment 14-4:
Program to Display a "2"

Address	Contents	Label	Instruction	Comments
0800	3E	START:	MVI A,4	;Set scan port to select digit (4 hex = 0000 0100 binary
0801	04			
0802	32		STA 2800	
0803	00			
0804	28			
0805	3E		MVI A,A4	;Set segment port to display character "2"
0806	A4			
0807	32		STA 3800	
0808	00			
0809	38			
080A	C3		JMP START	
080B	00			
080C	08			

SOLUTION TO FAULT 1

Problem:

A) Display: Stuck, random information
B) Output LEDs: Stuck, random information
C) Keyboard: No response

Power-up: Fail

Lights on: Yes
Bus activity: Yes

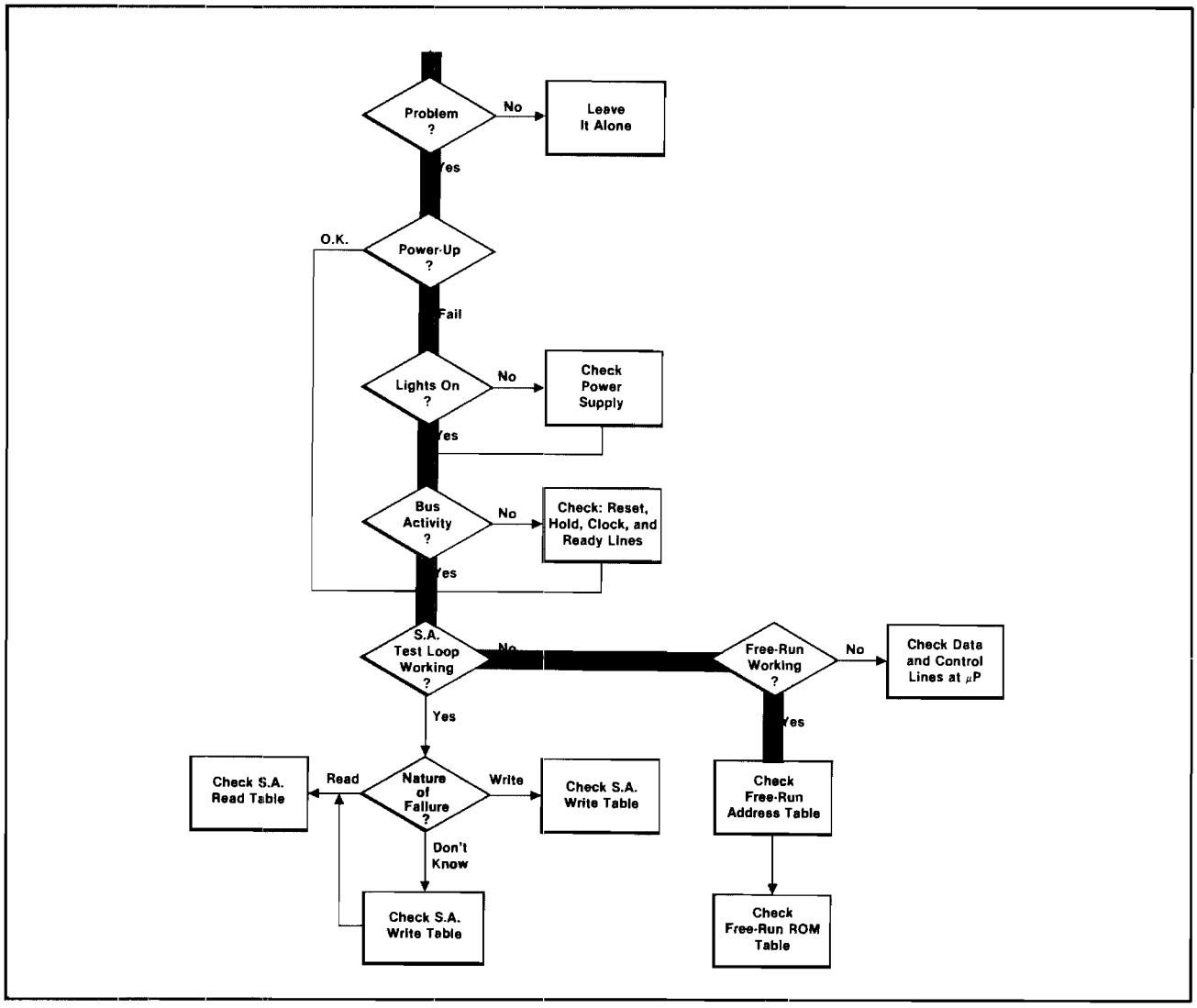
SA test loop working: No

Free-run working: Yes

Table C-1 A0-A15: One bad signature found

A) A11 signature is incorrect (HPPO).
B) A11 signature at IC1-3 (address buffer output) is correct (1293).
C) Trace along the signal path with the signature analyzer between IC1-3 and the A11 bus test point.
D) Locate Fault 1, a short between A10 and A11 bus lines.

Troubleshooting Path for Fault 1



SOLUTION TO FAULT 2

Problem:

A) Display: Stuck
B) Output LEDs: Stuck
C) Keyboard: Inoperative

Power-up: Fail

Lights on: Yes

Bus activity: Yes

SA test loop working: No

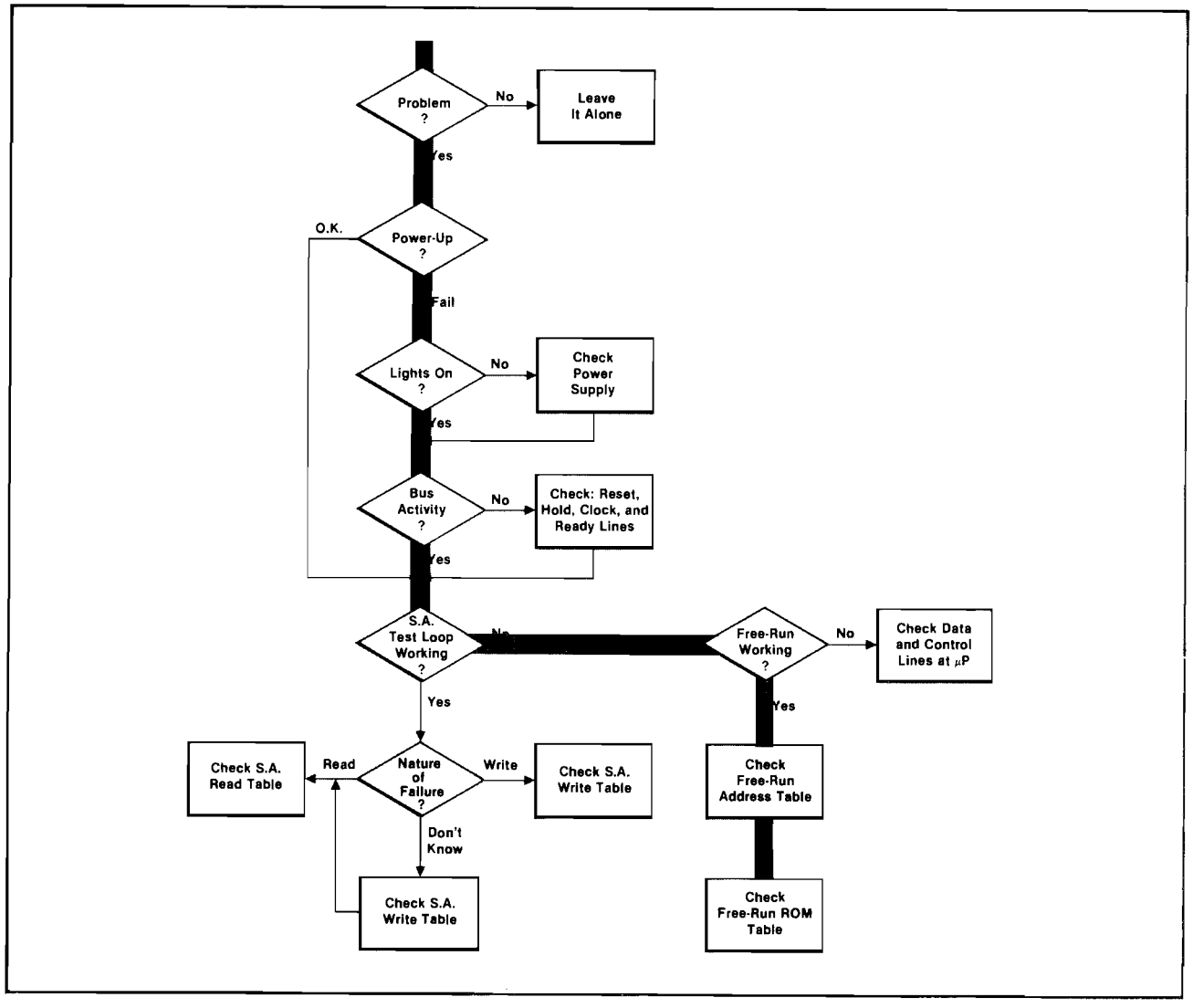
Free-run working: Yes

Table C-1 A0-A15: All address signatures good

Table C-2: One bad signature found

A) D5 signature is unstable.
B) D5 signature at IC4-15 (ROM output) is correct (0HF1).
C) Trace signal path with the signature analyzer between IC4-15 and the D5 test point.
D) Locate Fault 2, an open trace to the D5 ROM output.

Troubleshooting Path for Fault 2



SOLUTION TO FAULT 3

Problem:
A) Display: Stuck
B) Output LEDs: Stuck
C) Keyboard: Inoperative

Power-up: Fail

Lights on: Yes

Bus activity: Yes

SA test loop working: No

Free-run working: Yes. No activity appears on A0-A7 lines

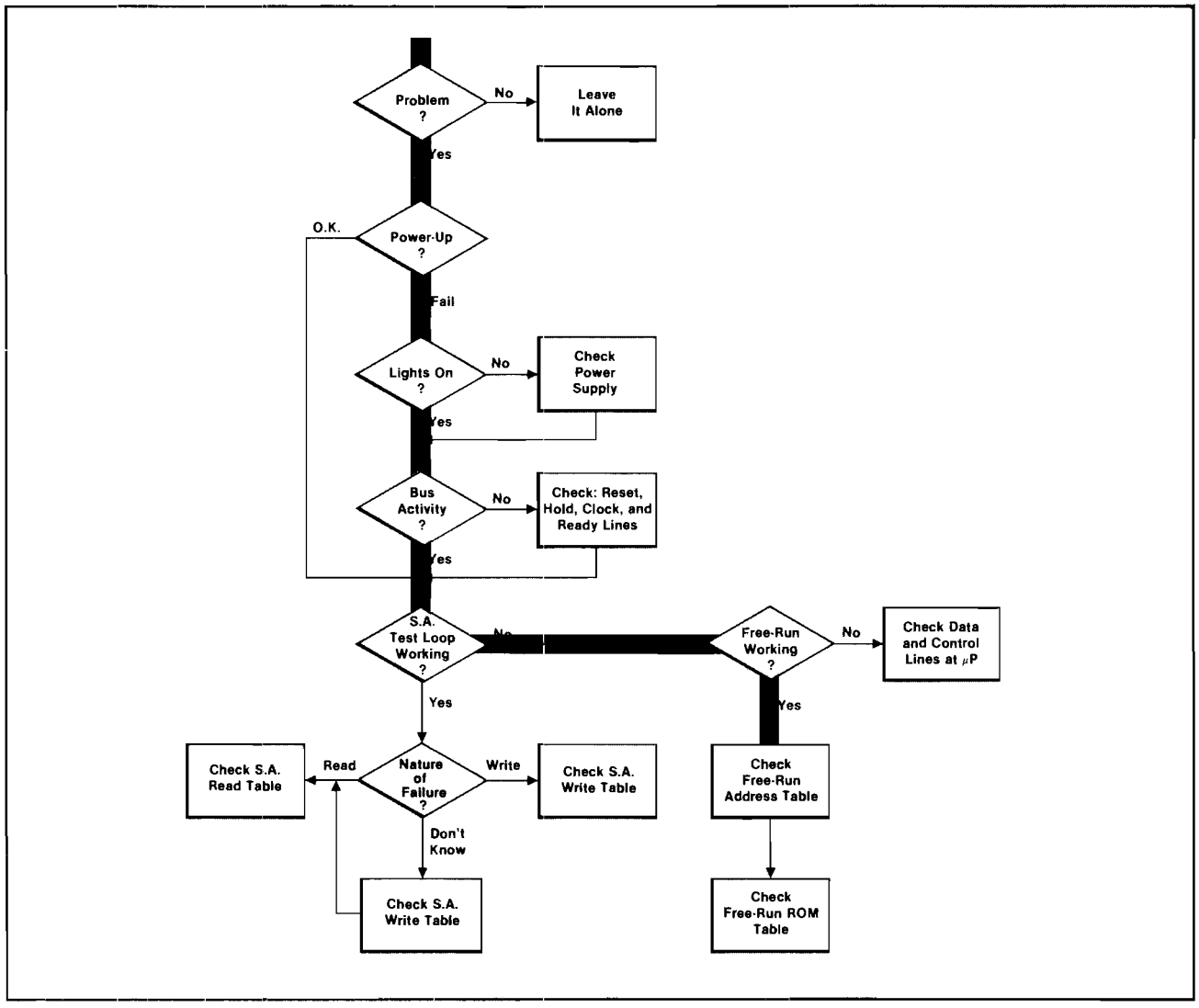
Table C-1 A0-A15: Bad signatures found

A) A0-A7 lines are stuck (no activity).
B) A0-A7 are stuck at IC2 outputs.
C) AD0-AD7 are not stuck at IC2 inputs.
D) Suspect IC2 or $\overline{ALE}$ signal.

Table C-1 IC2: Bad $\overline{ALE}$ signature found

A) $\overline{ALE}$ at IC2-11 is stuck high.
B) $\overline{ALE}$ at IC12-4 is stuck high.
C) $\overline{ALE}$ at IC12-3 has good signature.
D) $\overline{ALE}$ is the faulty signal. The signal is IC12-4, IC10-1, IC2-11, or a trace on the board.
E) Use the current tracer to verify that there is current on IC12-4. Place the current tracer tip right on IC12-4 and adjust the sensitivity to about midway. The level of current on this pin appears to be fairly high, suggesting that IC8 is trying to drive the node. The fault must be elsewhere.
F) Trace the current path along the trace that runs toward IC10-1 and then past it. Observe that the high current is not going into IC10-1.
G) Continue tracing the current toward IC2-11.
H) The current disappears once you pass Fault 3. Fault 3 shorts the $\overline{ALE}$ line to ground.

Troubleshooting Path for Fault 3



SOLUTION TO FAULT 4

Problem:

A) Display: IC6 failure message

Power-up: Fail

Lights on: Yes

Bus activity: Yes

SA test loop working: Yes

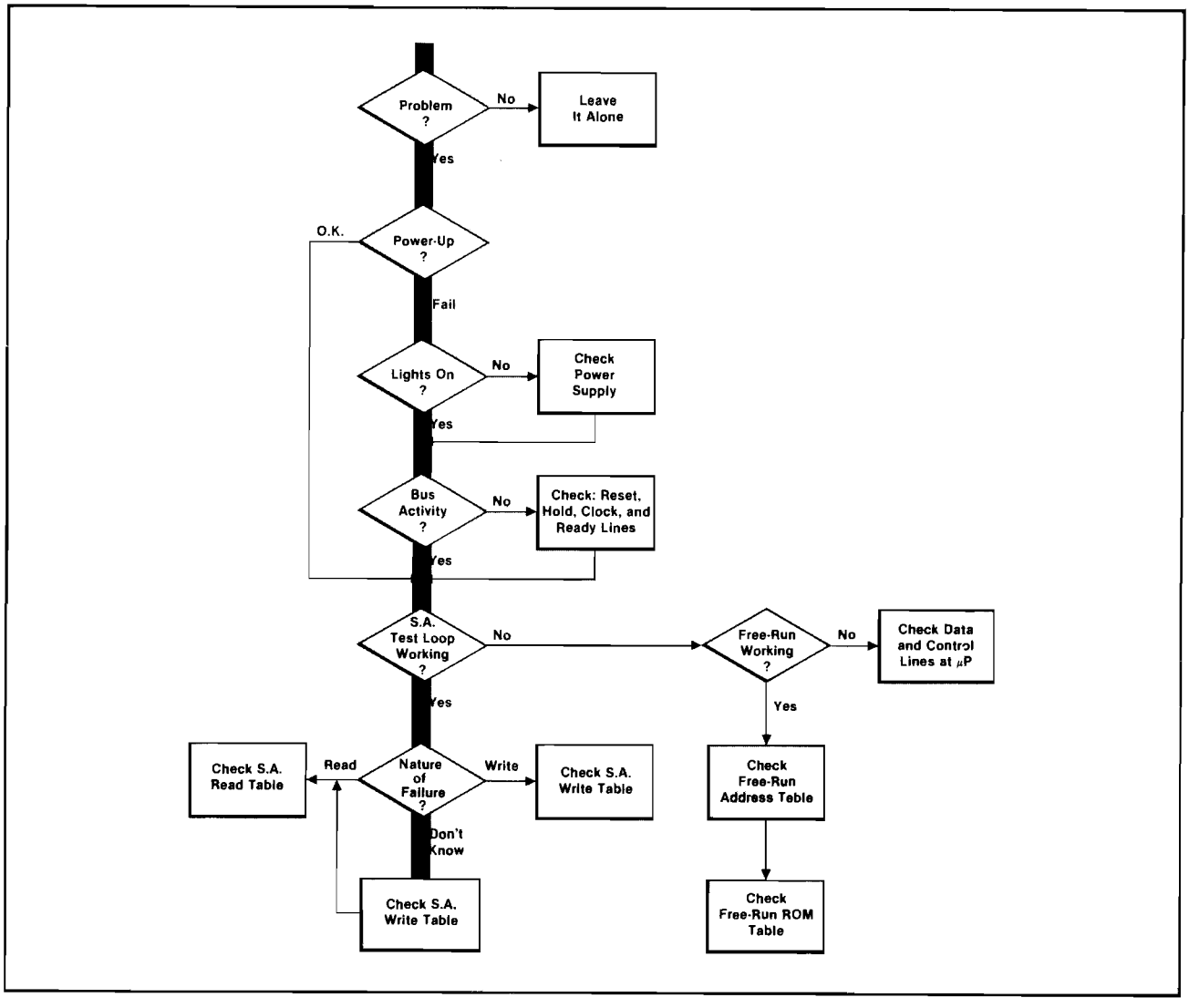
Either RAM IC6 or the control signals going to it are bad. If the control signals turn out to be good, then part of RAM IC6 is bad. At least some of it is good or else the keyboard and the display would not be operable.

Table C-3 IC6: One bad signature found

A) A7 on IC6-17 appears to be stuck high.
B) A7 on address bus has good signature.
C) Trace signal path with the signature analyzer between IC6-17 and the A7 test point.
D) Locate Fault 4, RAM address line A7 pulled up to Vcc.

A faulty address line input can be caused by an open trace or plate-through causing the RAM address pin to float high. An internally open RAM input pin will cause a similar failure without producing a bad signature on the input pin of the device.

Troubleshooting Path for Fault 4



SOLUTION TO FAULT 5

Problem:

A) Display: Stuck
B) Output LEDs: Flickering
C) Keyboard: Inoperative

Power-up: Fail

Lights on: Yes

Bus activity: Yes

SA test loop working: No

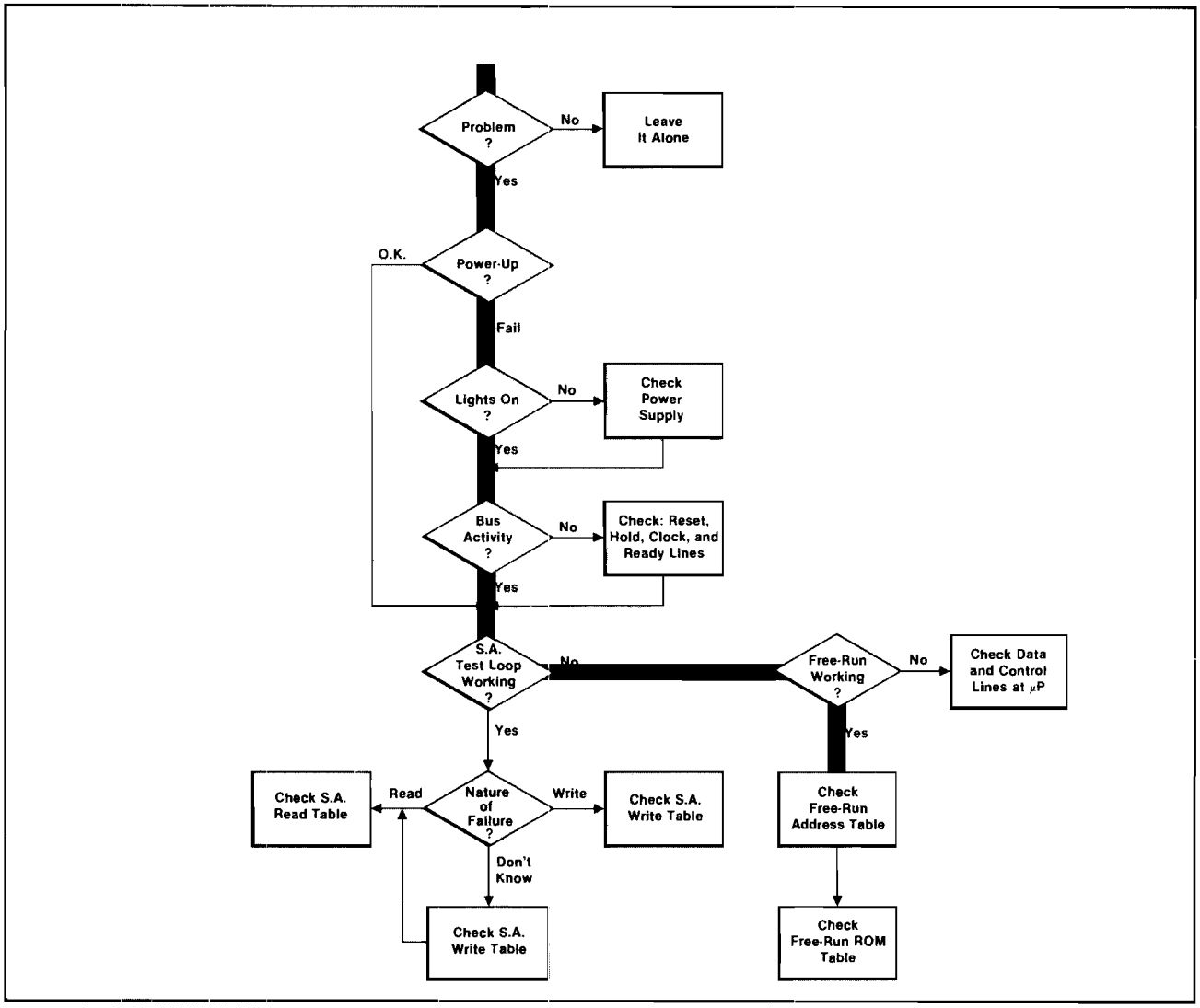
Free-run working: Yes

Table C-1 A0-A15: All signatures good

Table C-1 IC7 : Bad signatures found

A) Bad signatures on IC7 pins 3,7,9,10,11, 12,13,14,15.
B) The single bad input on pin 3 could explain all the other bad output signatures.
C) Pin 3 shows no activity and a bad logic level on the signature analyzer probe tip. Suspect an open trace to A13 line.
D) Trace the signal path between IC7-3 and the A13 test point.
E) Locate Fault 5, an open connection between A13 and IC7-3.

Troubleshooting Path for Fault 5



SOLUTION TO FAULT 6

Problem:

A) Display: SP error message.
B) Data cannot be modified in the protectable portion of RAM (0800-0AFF). Data can be modified in the unprotected portion of RAM (0800-OBFF).

Power-up: Fail

Lights on: Yes

Bus activity: Yes

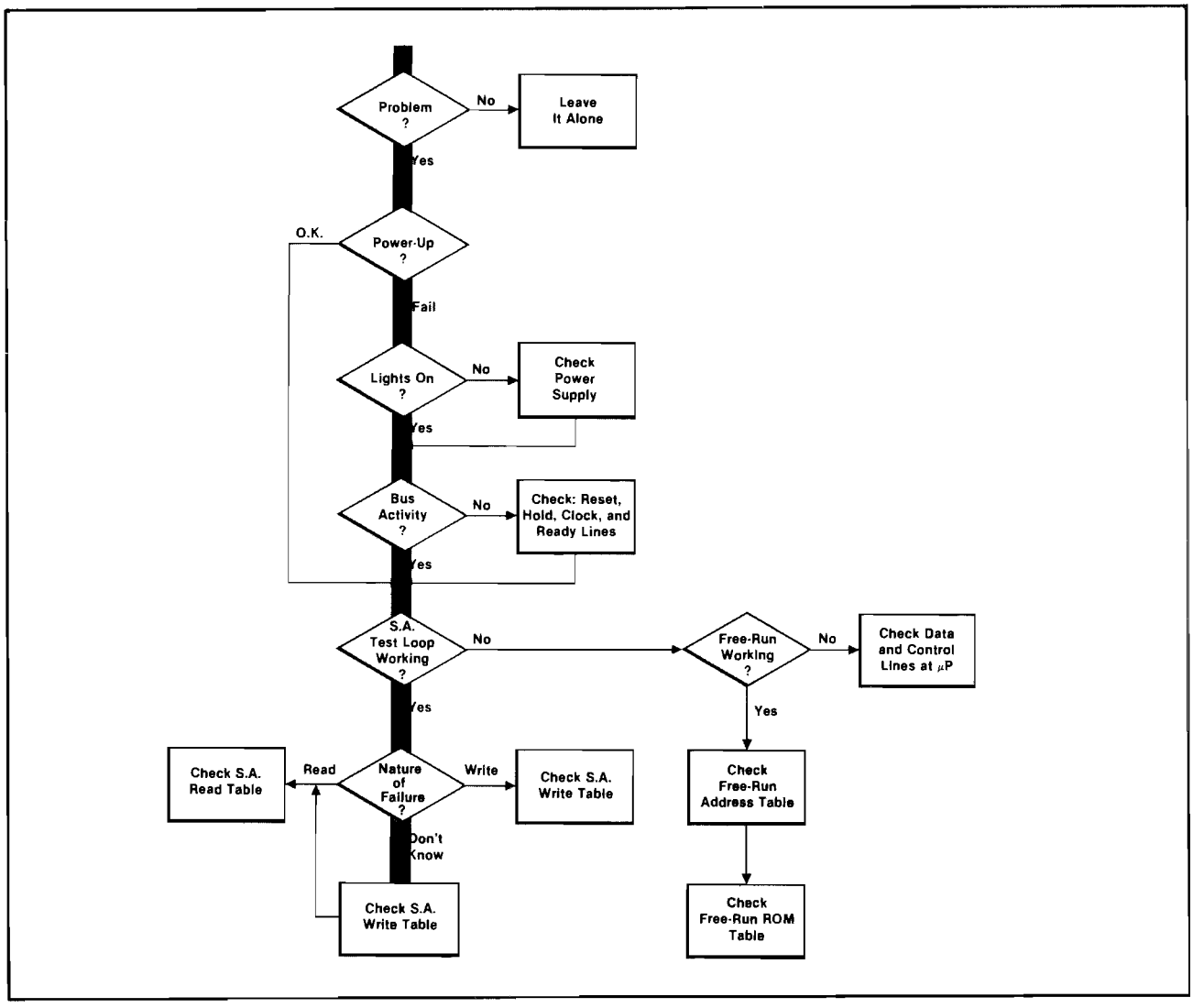
SA test loop working: Yes

The protectable portion of RAM appears to be protected all the time. The circuits involved in generating RAM write control signals should be looked at first.

Table C-3 ICs 11, 12, 5, 6: Bad signatures found

A) A bad WR signature on IC5 or 6 pin 10 can be traced back through a string of bad signatures on IC11 pins 3 and 2, IC12 pins 2 and 1, IC11 pins 6 and 4 to a good signature on IC8-6.
B) Since IC11-4 and IC8-6 are supposed to be connected (and have the same signature), a break between them can be assumed.
C) Trace the signal path between IC11-4 and IC8-6.
D) Locate Fault 6, a short to ground on IC11-4.

Troubleshooting Path for Fault 6



SOLUTION TO FAULT 7

Problem:

A) Display: Stuck
B) Output LEDs: Stuck
C) Keyboard: Inoperative

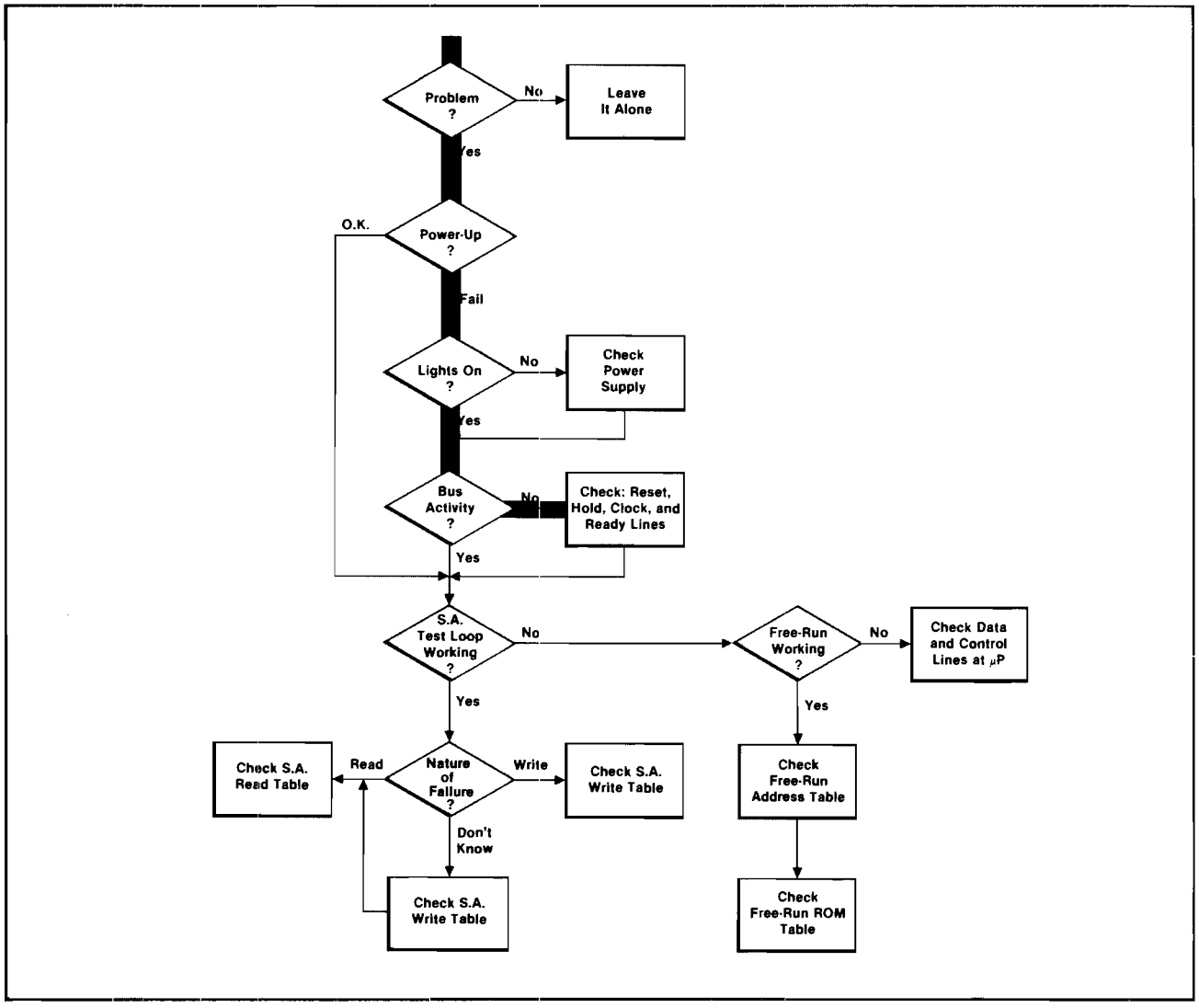
Power-up: Fail

Lights on: Yes

Bus activity: No

A) Ready line is stuck low at IC3-35.
B) The step signal (IC10-5), which goes to the Ready line, is high.
C) Trace the signal path between IC10-5 and IC3-35 (Microprocessor Ready input).
D) Locate Fault 7, a short to ground on the Ready input.

Troubleshooting Path for Fault 7



SOLUTION TO FAULT 8

Problem:

A) Display: Stuck
B) Output LEDs: Stuck
C) Keyboard: Inoperative

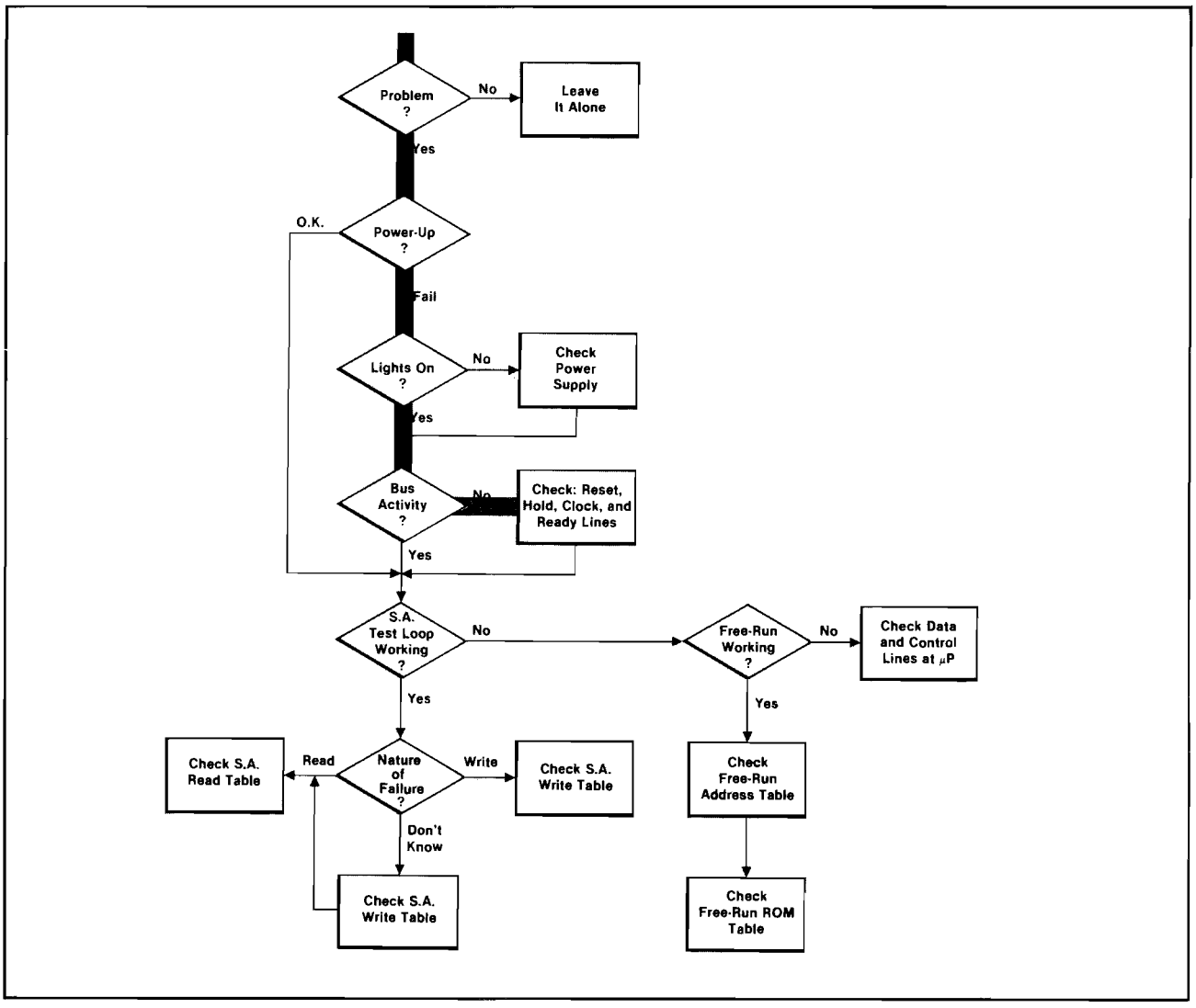
Power-up: Fail

Lights on: Yes

Bus activity: No

A) Hold line is stuck high at IC3-39.
B) Trace the signal from IC3-39 to Fault 8, a short to Vcc.

Troubleshooting Path for Fault 8



SOLUTION TO FAULT 9

Problem:

A) Display: Center segment on all digits fail to light.
B) Output LEDs: D6 LED fails to light on power-up.

Power-up: Completes power-up sequence, but fails visually

Lights on: Yes

Bus activity: Yes

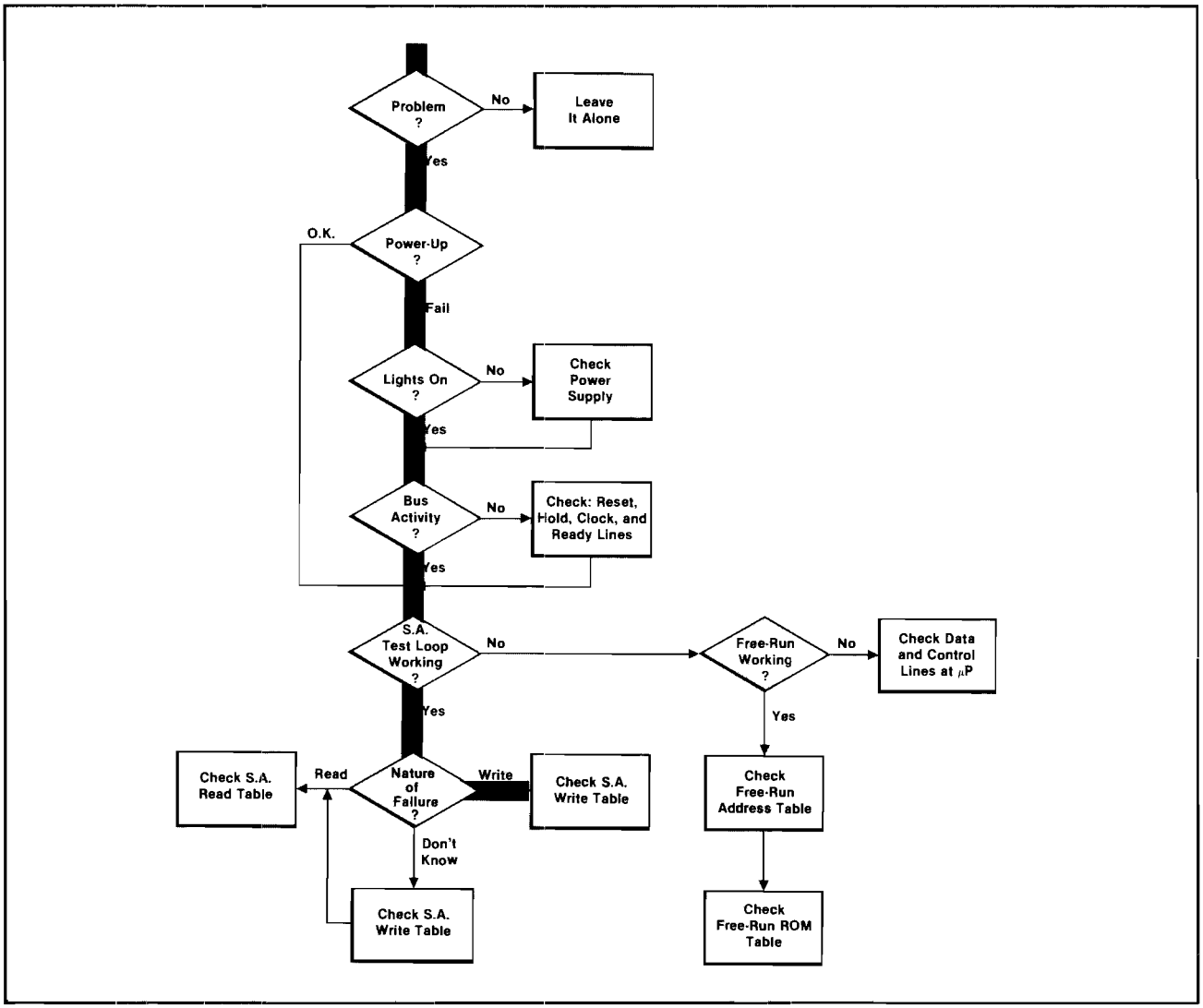
SA test loop working: Yes

The center segments of the display and the D6 output LED have the D6 data line in common and are devices that the microprocessor writes to.

Table C-3 IC14, 15, 16, 19, display and output LEDs: Bad signatures found

A) A lack of activity on the D6 output LED can be traced back to IC15-16, to IC15-17, and finally to a good signature on IC14-15.
B) A lack of activity on "g" segment (display pins 7 and 17) can be traced back to IC19-11, to IC19-7, to IC16-16, to IC16-17, and finally to a good signature on IC14-15.
C) Trace the signal path between IC14-15 and IC16-17.
D) Locate Fault 9, an open trace on the D6 buffered data line.

Troubleshooting Path for Fault 9



SOLUTION TO FAULT 10

Problem:

When the ECHO demo program is run (address 04D7), output LEDs 4 and 5 are both controlled by input switch 5. Input switch 4 has no effect.

Power-up: Yes

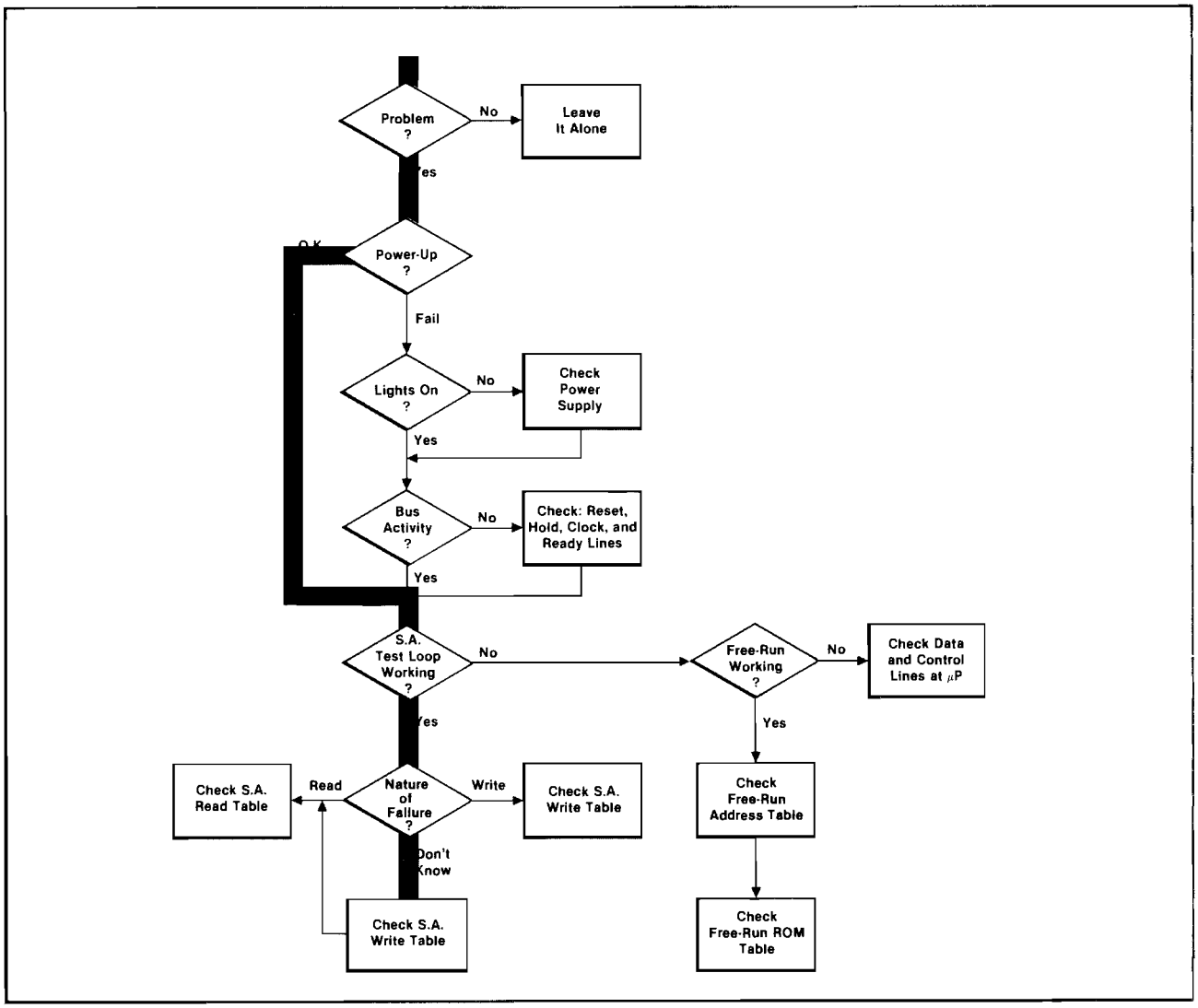
SA test loop working: Yes

This could be a read fault from the input switches or a write fault to the output LEDs.

Table C-3 ICs 14, 15, output LEDs: Bad signatures found

A) Identical signatures on output LEDs 4 and 5 (correct output 5 signature) indicate a probable short between them.
B) Trace signatures back from output LED 4 to IC15-12 to IC15-13 to a good signature on IC14-11.
C) Trace the signal path between IC14-11 and IC15-13.
D) Locate Fault 10, a short between the D4 and D5 buffered data bus lines.

Troubleshooting Path for Fault 10



SOLUTION TO FAULT 11

Problem:

Keyboard: No response to FETCH ADRS, DECR, 3, 6, 9, C, F keys

Power-up: Yes

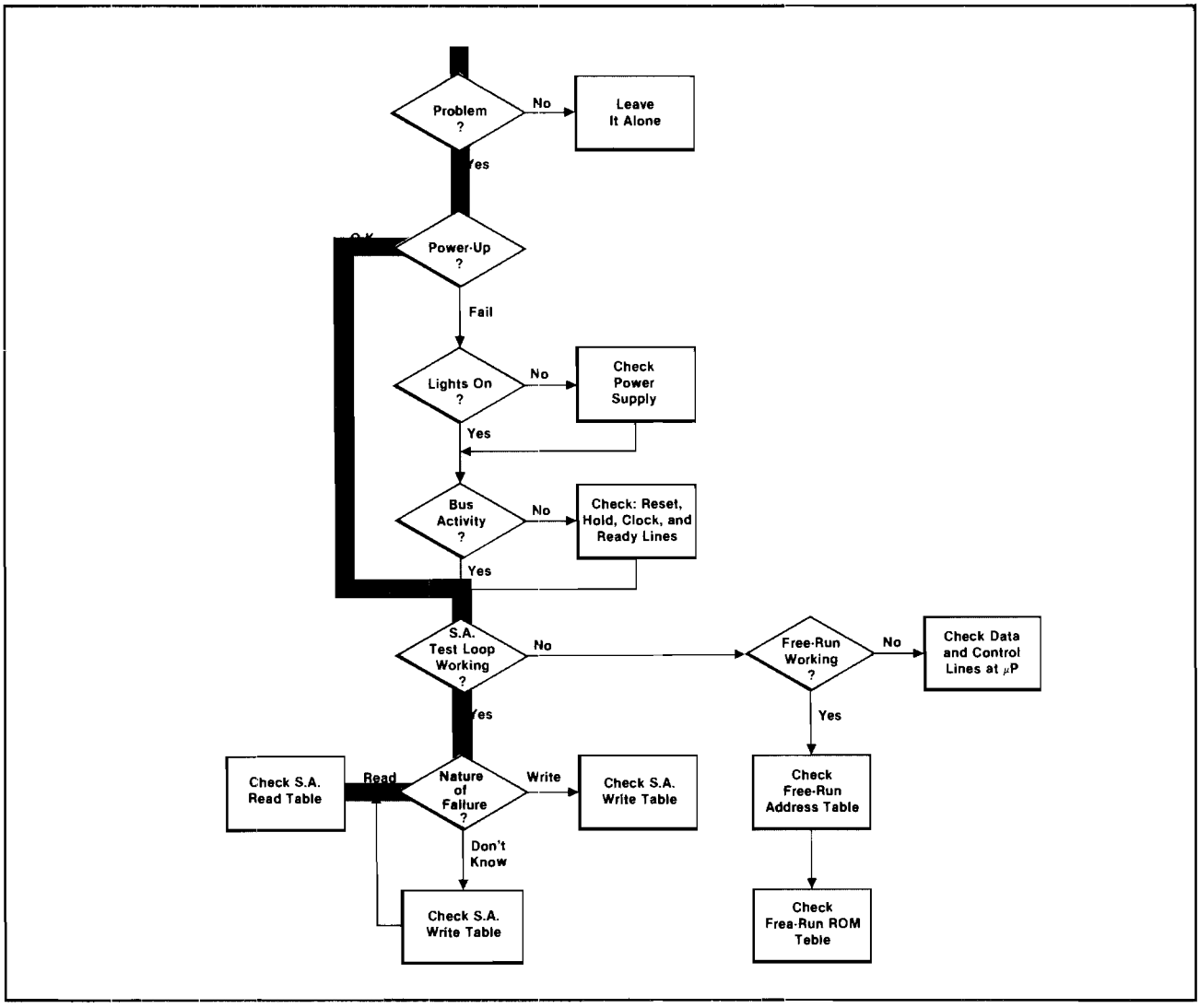
SA test loop working: Yes

These nonreading keys have the D2 line in common. This fault appears to be read-oriented.

Table C-4 IC 18: Bad signatures found when keys pushed

A) Signatures do not change to proper values on IC18-15 (D2) when any of these keys is pressed.
B) IC18-16 remains high when any of these keys is pressed.
C) The correct signature appears near the keys (try to trace below the 3 key on the top side of the board).
D) Trace the signal path between the key column line and IC18-16.
E) Locate Fault 11 , an open trace between the 3-F key column and IC18-16.

Troubleshooting Path for Fault 11



SOLUTION TO FAULT 12

Problem:

A) Display: Left two digits blank at all times
B) Keyboard: Keys 7, 8, 9 input values 4, 5, 6, respectively

Power-up: Completes power-up sequence, but fails visually

Lights on: Yes

Bus activity: Yes

SA test loop working: Yes. Left two display digits now come on.

Scan latch lines D4 and D5 are common to the the bad display and keyboard sections. This is probably a write fault at the scan latch IC17.

Table C-3 IC17, 20, and keyboard: Bad signatures found

A) Identical bad signatures on IC20-3 and IC20-5, on key scan rows, 4-6 and 7-9, and on IC17-12 and IC17-15 are found.

B) Good signatures on IC17-13 and IC17-14 indicate a probable short between D4 and D5 lines on the output side of the scan port (IC17).

C) A short between pins 12 and 15 will cause relatively high currents to occur periodically when these output pins are in opposite logic states. The path of current should flow from one output pin to the board, along the trace, toward the short, through the short, and then into the other trace leading back to the other IC output pin.

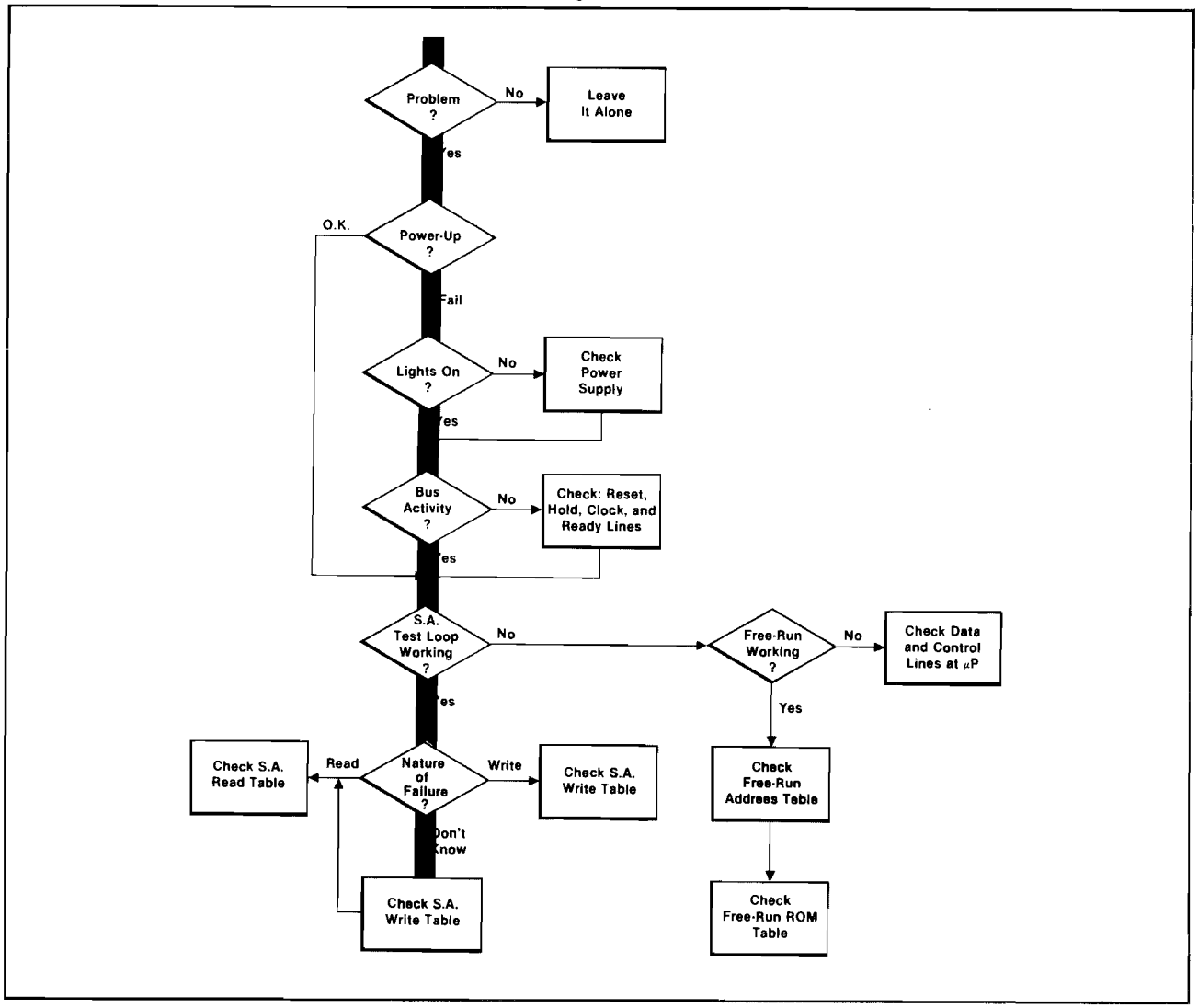
D) Using the current tracer, follow the current on the underside of the board from IC17-12 to IC20-3 to the plate-through to the left of the INTRPT key.

E) Observe that the current goes away as the current tracer moves past the plate-through (still on the underside of the board) toward the keyboard.

F) Follow the current through the plate-through to the top of the board.

G) Locate Fault 12, a short between scan lines 4 and 5.

Troubleshooting Path for Fault 12



APPENDIX B

8085A Instruction Set Reference

GENERAL

The following information is extracted from Intel's MCS 85™ User's Manual*. For additional information on the capability of the 8085 Microprocessor refer to the User's Manual. In the explanations of the individual instructions, the hexadecimal equivalent of each instruction is given in addition to the binary format.

The 8085 instruction set includes five different types of instructions:

- **Data Transfer Group**—move data between registers or between memory and registers
- **Arithmetic Group**—add, subtract, increment or decrement data in registers or in memory
- **Logical Group**—AND, OR, EXCLUSIVE-OR, compare, rotate or complement data in registers or in memory
- **Branch Group**—conditional and unconditional jump instructions, subroutine call instructions and return instructions
- **Stack, I/O and Machine Control Group**— includes I/O instructions, as well as instructions for maintaining the stack and internal control flags.

The individual instruction groups and Table B-1 are divided into these types of instructions and are presented in the same sequential order.

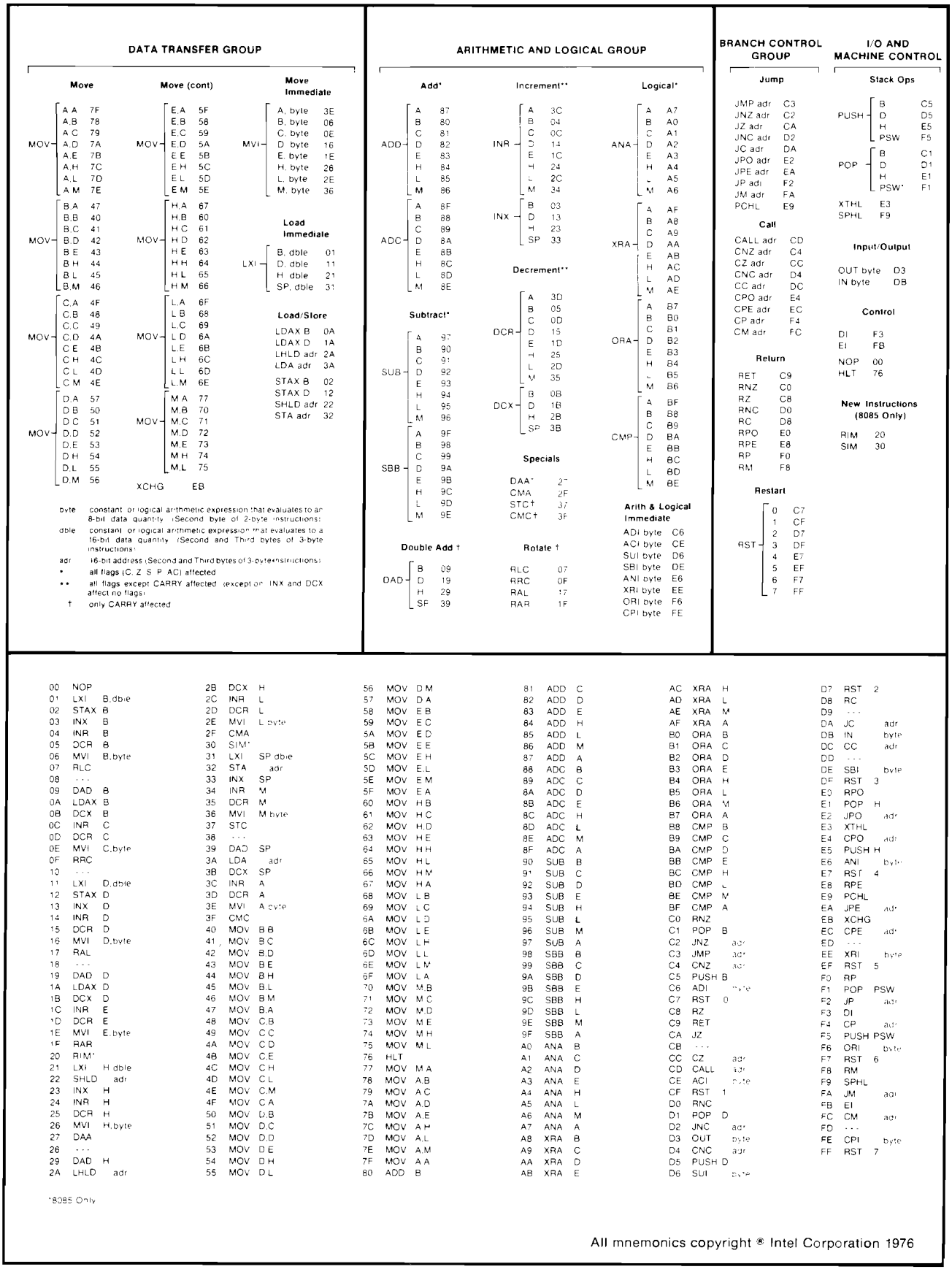
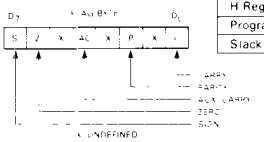


Table B-1. 8085/8080 Assembly Language Reference

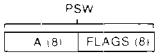
INTEL® 8080/8085
INSTRUCTION SET REFERENCE TABLES

INTERNAL REGISTER ORGANIZATION

A Reg (8)	
B Reg (8)	C Reg (8)
D Reg (8)	E Reg (8)
H Reg (8)	L Reg (8)
Program Counter (16)	
Stack Pointer (16)	



REGISTER-PAIR ORGANIZATION



NOTE: Leftmost Byte is high-order byte for arithmetic operations and addressing. Left byte is pushed or stack first. Right byte is popped first.

B (B C) (16)	
D (D E) (16)	
H (H L) (16)	
Prog Ctr (16)	
Stack Ptr (16)	

BRANCH CONTROL INSTRUCTIONS

Flag Condition	Jump		Call		Return	
Zero=True	JZ	CA	CZ	CC	RZ	C8
Zero=False	JNZ	C2	CNZ	C4	RNZ	C0
Carry=True	JC	DA	CC	DC	RC	D8
Carry=False	JNC	D2	CNC	D4	RNC	D0
Sign=Positive	JP	F2	CP	F4	RP	F0
Sign=Negative	JM	FA	CM	FC	RM	F8
Parity=Even	JPE	EA	CPE	EC	RPE	E8
Parity=Odd	JPO	E2	CPO	E4	RPO	E0
Unconditional	JMP	C3	CALL	CD	RET	C9

ACCUMULATOR OPERATIONS

	Code	Function
XRA A	AF	Clear A and Clear Carry
ORA A	B7	Clear Carry
CMC	3F	Complement Carry
CMA	2F	Complement Accumulator
STC	37	Set Carry
RLC	07	Rotate Left
RRC	0F	Rotate Right
RAL	17	Rotate Left Thru Carry
RAR	1F	Rotate Right Thru Carry
DAA	27	Decimal Adjust Accum

RESTART TABLE

Name	Code	Restart Address
RST 0	C7	0000_{16}
RST 1	CF	0008_{16}
RST 2	D7	0010_{16}
RST 3	DF	0018_{16}
RST 4	E7	0020_{16}
TRAP	Hardware* Function	0024_{16}
RST 5	EF	0028_{16}
RST 5.5	Hardware* Function	$002C_{16}$
RST 6	F7	0030_{16}
RST 6.5	Hardware* Function	0034_{16}
RST 7	FF	0038_{16}
RST 7.5	Hardware* Function	$003C_{16}$

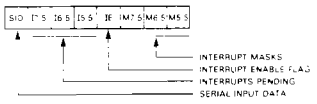
*NOTE: The hardware functions refer to the on-chip interrupt feature of the 8085 only.

HEX-ASCII TABLE

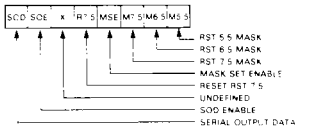
00	NUL		21	!	42	B	63	c
01	SOH		22	"	43	C	64	d
02	STX		23	#	44	D	65	e
03	ETX		24	$	45	E	66	f
04	EOT		25	%	46	F	67	g
05	ENQ		26	&	47	G	68	h
06	ACK		27	'	48	H	69	i
07	BEL		28	(	49	I	6A	j
08	BS		29	)	4A	J	6B	k
09	HT		2A	*	4B	K	6C	l
0A	LF		2B	+	4C	L	6D	m
0B	VT		2C	,	4D	M	6E	n
0C	FF		2D	–	4E	N	6F	o
0D	CR		2E	.	4F	O	70	p
0E	SQ		2F	/	50	P	71	q
0F	SI		30	0	51	Q	72	r
10	DLE		31	1	52	R	73	s
11	DC1	(X-ON)	32	2	53	S	74	t
12	DC2	(TAPE)	33	3	54	T	75	u
13	DC3	(X-OFF)	34	4	55	U	76	v
14	DC4	~~(TAPE)~~	35	5	56	V	77	w
15	NAK		36	6	57	W	78	x
16	SYN		37	7	58	X	79	y
17	ETB		38	8	59	Y	7A	z
18	CAN		39	9	5A	Z	7B	{
19	EM		3A	:	5B	[	7C	|
1A	SUB		3B	;	5C	\	7D	}
1B	ESC		3C	<	5D	]		(ALT MODE)
1C	FS		3D	=	5E	↑	7E	~
1D	GS		3E	>	5F	←	7F	DEL
1E	RS		3F	?	60	`		(RUB OUT)
1F	US		40	@	61	a		
20	SP		41	A	62	b		

USE OF THE A REGISTER BY RIM AND SIM INSTRUCTIONS (8085 ONLY)

A REGISTER AFTER EXECUTING RIM



A REGISTER BEFORE EXECUTING SIM



REGISTER PAIR AND STACK OPERATIONS

		Register Pair					
	PSW (A F)	B (B C)	D (D E)	H (H L)	SP	PC	Function
INX		03	13	23	33		Increment Register Pair
DCX		0B	1B	2B	33		Decrement Register Pair
LDAX		0A	1A	7E(1)			Load A Indirect (Reg Pair holds Adrs)
STAX		02	12	77(2)			Store A Indirect (Reg Pair holds Adrs)
LHLD				2A			Load H L Direct (Bytes 2 and 3 hold Adrs)
SHLD				22			Store H L Direct (Bytes 2 and 3 hold Adrs)
LXI		01	11	21	31	C3(3)	Load Reg Pair Immediate (Bytes 2 and 3 hold immediate data)
PCHL						E9	Load PC with H L (Branch to Adrs in H L)
XCHG			EB				Exchange Reg Pairs D E and H L
DAD		09	19	29	39		Add Reg Pair to H L
PUSH	F5	C5	D5	E5			Push Reg Pair on Stack
POP	F1	C1	D1	E1			Pop Reg Pair off Stack
XTHL				E3			Exchange H L with Top of Stack
SPHL					F9		Load SP with H L

Notes: 1. This is MOV A,M. 2. This is MOV M,A. 3. This is JMP.

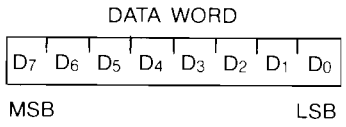
All mnemonics copyright ® Intel Corporation 1976

Table B-1. 8085/8080 Assembly Language Reference (Continued)

Instruction and Data Formats:

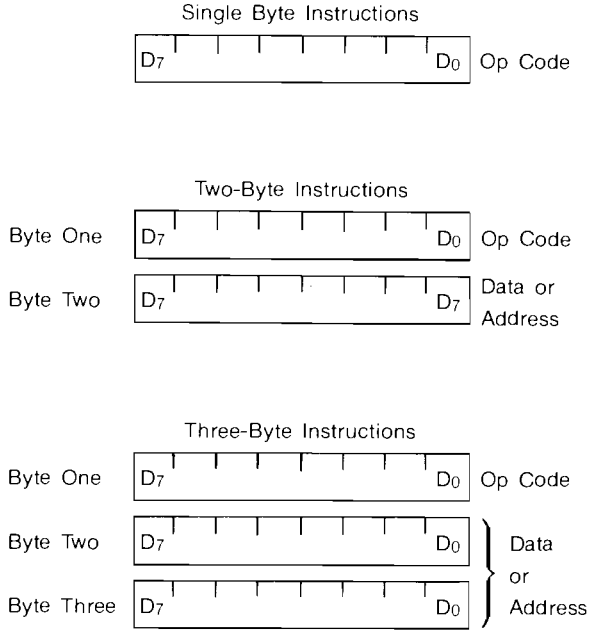
Memory for the 8085 is organized into 8-bit quantities, called bytes. Each byte has a unique 16-bit binary address corresponding to its sequential position in memory. The 8085 can directly address up to 65,536 bytes of memory, which may consist of both read-only memory (ROM) elements and random-access memory (RAM) elements (read/write memory).

Data in the 8085 is stored in the form of 8-bit binary integers:



When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written. In the Intel 8085, BIT 0 is referred to as the **Least Significant Bit (LSB)**, and BIT 7 (of an 8 bit number) is referred to as the **Most Significant Bit (MSB)**.

The 8085 program instructions may be one, two or three bytes in length. Multiple byte instructions must be stored in successive memory locations. The address of the first byte is always used as the address of the instructions. The exact instruction format depends on the particular operation to be executed.



Addressing Modes:

Often the data to be operated on is stored in memory. When multi-byte numeric data is used, the data, like instructions, is stored in successive memory locations. The least significant byte is stored first, followed by increasingly significant bytes. The 8085 has four different modes for addressing data stored in memory or in registers:

- **Direct**—Bytes 2 and 3 of the instruction contain the exact memory address of the data item. The low-order bits of the address are in byte 2, the high-order bits in byte 3.
- **Register**—The instruction specifies the register or register-pair in which the data is located.
- **Register Indirect**—The instruction specifies a register-pair which contains the memory address where the data is located. The high-order bits of the address are in the first register of the pair, the low-order bits in the second.
- **Immediate**—The instruction contains the data itself. This is either an 8-bit quantity or a 16-bit quantity (least significant byte first, most significant byte second).

Unless directed by an interrupt or branch instruction, the execution of instructions proceeds through consecutively increasing memory locations. A branch instruction can specify the address of the next instruction to be executed in one of two ways:

- **Direct**—The branch instruction contains the address of the next instruction to be executed. Except for the 'RST' instruction, byte 2 contains the low-order address and byte 3 the high-order address.
- **Register Indirect**—The branch instruction indicates a register-pair which contains the address of the next instruction to be executed. The high-order bits of the address are in the first register of the pair, the low-order bits in the second.

The RST instruction is a special one-byte call instruction (usually used during interrupt sequences). RST includes a three-bit field. Program control is transferred to the instruction whose address is eight times the contents of this three-bit field.

All mnemonics copyright ® Intel Corporation 1976

Condition Flags:

There are five condition flags associated with the execution of instructions on the 8085. They are Zero, Sign, Parity, Carry, and Auxiliary Carry, and are each represented by a 1-bit register in the CPU. A flag is "set" by forcing the bit to 1. It is "reset" by forcing the bit to 0.

Unless otherwise indicated, when an instruction affects a flag, it affects it in the following manner:

- **Zero**—If the result of an instruction has the value 0, this flag is set. Otherwise, it is reset.
- **Sign**—If the most significant bit of the result of the operation has the value 1, this flag is set. Otherwise, it is reset.
- **Parity**—If the modulo 2 sum of the bits of the result of the operation is 0, (i.e., if the result has even parity), this flag is set. Otherwise, it is reset (i.e., if the result has odd parity).
- **Carry**—If the instruction resulted in a carry (from addition), or a borrow (from subtraction or a comparison) out of the high-order bit, this flag is set. Otherwise, it is reset.
- **Auxiliary Carry**—If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, the auxiliary carry is set. Otherwise, it is reset. This flag is affected by single precision additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used with additions and increments preceding a DAA (Decimal Adjust Accumulator) instruction.

All mnemonics copyright ® Intel Corporation 1976

Symbols and Abbreviations:

The following symbols and abbreviations are used in the subsequent description of the 8085 instructions:

SYMBOLS	MEANING
accumulator	Register A
addr	16-bit address quantity
byte	8-bit data quantity
dble	16-bit data quantity
byte 2	The second byte of the instruction
byte 3	The third byte of the instruction
port	8-bit address of an I/O device
r, r1, r2	One of the registers A, B, C, D, E, H, L
DDD, SSS	The bit pattern designating one of the registers A, B, C, D, E, H, L (DDD = destination, SSS = source):

DDD or SSS	REGISTER NAME
111	A
000	B
001	C
010	D
011	E
100	H
101	L

rp	One of the register pairs: B represents the B, C pair with B as the high-order register and C as the low-order register. D represents the D, E pair with D as the high-order register and E as the low-order register. H represents the H, L pair with H as the high-order register and L as the low-order register. SP represents the 16-bit stack pointer register.

SYMBOLS	MEANING (Continued)
RP	The bit pattern designating one of the register pairs B, D, H, SP:

RP	REGISTER PAIR
00	B-C
01	D-E
10	H-L
11	SP

rh	The first (high-order) register of a designated register pair.
rl	The second (low-order) register of a designated register pair.
PC	16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order 8 bits respectively).
SP	16-bit stack pointer register (SPH and SPL are used to refer to the high-order and low-order 8 bits respectively).
r_m	Bit m of the register r (bits are number 7 through 0 from left to right).
Z, S, P, CY, AC	The condition flags: Zero, Sign, Parity, Carry, and Auxiliary Carry, respectively.
( )	The contents of the memory location or registers enclosed in the parentheses.
←	"Is transferred to"
∧	Logical AND
⊻	Exclusive OR
∨	Inclusive OR
+	Addition
−	Two's complement subtraction
*	Multiplication
↔	"Is exchanged with"
—	The one's complement (e.g., $(\overline{A})$)
n	The restart number 0 through 7
NNN	The binary representation 000 through 111 for restart number 0 through 7 respectively.

Description Format:

The following pages provide a detailed description of the instruction set of the 8085. Each instruction is described in the following manner:

1. The instruction format consists of the instruction mnemonic and operand fields, and is printed in **BOLDFACE** on the left side of the first line. Note that each entry must be converted to hexadecimal before you can enter it into the microlab.

2. The name of the instruction is enclosed in parenthesis on the right side of the first line.

3. The next line(s) contain a symbolic description of the operation of the instruction.

4. This is followed by a narrative description of the operation of the instruction.

5. The following line(s) contain the binary fields and patterns that comprise the machine instruction.

6. The following lines contain the hexadecimal format op code highlighted in color.

7. The last four lines contain incidental information about the execution of the instruction. The number of machine cycles and states required to execute the instruction are listed first. If the instruction has two possible execution times, as in a Conditional Jump, both times will be listed, separated by a slash. Next, any significant data addressing modes are listed. The last line lists any of the five Flags that are affected by the execution of the instruction.

All mnemonics copyright ® Intel Corporation 1976

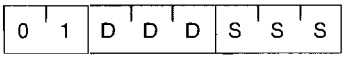
Data Transfer Group:

This group of instructions transfers data to and from registers and memory. Condition flags are not affected by any instruction in this group.

MOV r1, r2 (Move Register)

(r1) ← (r2)

The content of register r2 is moved to register r1.



Hexadecimal Format

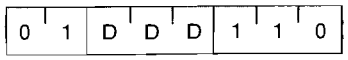
MOV	A, A	7F
MOV	A, B	78
MOV	A, C	79
MOV	A, D	7A
MOV	A, E	7B
MOV	A, H	7C
MOV	A, L	7D
MOV	B, A	47
MOV	B, B	40
MOV	B, C	41
MOV	B, D	42
MOV	B, E	43
MOV	B, H	44
MOV	B, L	45
MOV	C, A	4F
MOV	C, B	48
MOV	C, C	49
MOV	C, D	4A
MOV	C, E	4B
MOV	C, H	4C
MOV	C, L	4D
MOV	D, A	57
MOV	D, B	50
MOV	D, C	51
MOV	D, D	52
MOV	D, E	53
MOV	D, H	54
MOV	D, L	55
MOV	E, A	5F
MOV	E, B	58
MOV	E, C	59
MOV	E, D	5A
MOV	E, E	5B
MOV	E, H	5C
MOV	E, L	5D
MOV	H, A	67
MOV	H, B	60
MOV	H, C	61
MOV	H, D	62
MOV	H, E	63
MOV	H, H	64
MOV	H, L	65
MOV	L, A	6F
MOV	L, B	68
MOV	L, C	69
MOV	L, D	6A
MOV	L, E	6B
MOV	L, H	6C
MOV	L, L	6D

Cycles: 1
States: 4
Addressing: register
Flags: none

MOV r, M (Move from memory)

(r) ← ((H) (L))

The content of the memory location, whose address is in registers H and L, is moved to register r.



Hexadecimal Format

MOV	A, M	7E
MOV	B, M	46
MOV	C, M	4E
MOV	D, M	56
MOV	E, M	5E
MOV	H, M	66
MOV	L, M	6E

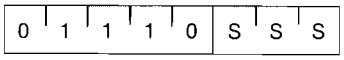
Cycles: 2
States: 7
Addressing: register indirect
Flags: none

All mnemonics copyright ®Intel Corporation 1976

MOV M, r (Move to memory)

((H) (L)) ← (r)

The content of register r is moved to the memory location whose address is in registers H and L.



Hexadecimal Format

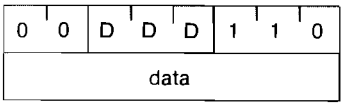
MOV	M, A	77
MOV	M, B	70
MOV	M, C	71
MOV	M, D	72
MOV	M, E	73
MOV	M, H	74
MOV	M, L	75

Cycles: 2
States: 7
Addressing: register indirect
Flags: none

MVI r, byte (Move Immediate)

(r) ← (byte 2)

The content of byte 2 of the instruction is moved to register r.



Hexadecimal Format

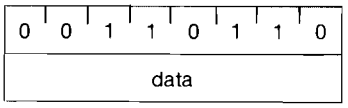
MVI	A, byte	3E
MVI	B, byte	06
MVI	C, byte	0E
MVI	D, byte	16
MVI	E, byte	1E
MVI	H, byte	26
MVI	L, byte	2E

Cycles: 2
States: 7
Addressing: immediate
Flags: none

MVI M, byte (Move to memory immediate)

((H) (L)) ← (byte 2)

The content of byte 2 of the instruction is moved to the memory location whose address is in registers H and L.



Hexadecimal Format

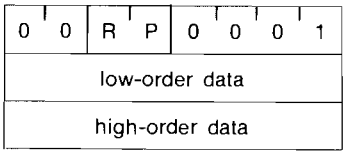
MVI M, byte 36

Cycles: 3
States: 10
Addressing: immediate/reg. indirect
Flags: none

LXI rp, dble (Load register pair immediate)

(rh) ← (byte 3),
(rl) ← (byte 2)

Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.



Hexadecimal Format

LXI	B, dble	01	(load immediate register pair B and C)
LXI	D, dble	11	(load immediate register pair D and E)
LXI	H, dble	21	(load immediate register pair H and L)
LXI	SP, dble	31	(load immediate stack pointer)

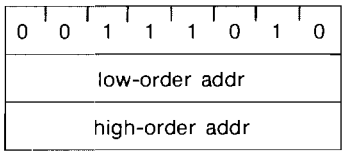
Cycles: 3
States: 10
Addressing: immediate
Flags: none

All mnemonics copyright ® Intel Corporation 1976

LDA addr (Load Accumulator direct)

(A) ← ((byte 3)(byte 2))

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.



Hexadecimal Format

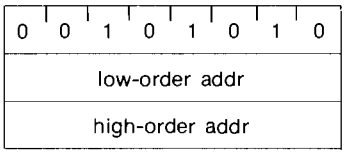
LDA addr 3A

Cycles: 4
States: 13
Addressing: direct
Flags: none

LHLD addr (Load H and L direct)

(L) ← ((byte 3)(byte 2))
(H) ← ((byte 3)(byte 2) + 1)

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register L. The content of the memory location at the succeeding address is moved to register H.



Hexadecimal Format

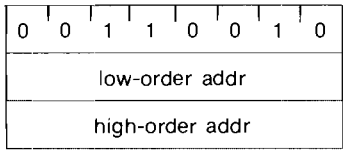
LHLD addr 2A

Cycles: 5
States: 16
Addressing: direct
Flags: none

STA addr (Store Accumulator direct)

((byte 3)(byte 2)) ← (A)

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.



Hexadecimal Format

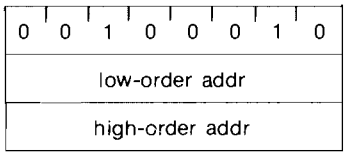
STA addr 32

Cycles: 4
States: 13
Addressing: direct
Flags: none

SHLD addr (Store H and L direct)

((byte 3)(byte 2)) ← (L)
((byte 3)(byte 2) + 1) ← (H)

The content of register L is moved to the memory location whose address is speci fied in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.



Hexadecimal Format

SHLD addr 22

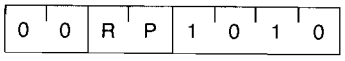
Cycles: 5
States: 16
Addressing: direct
Flags: none

All mnemonics copyright ® Intel Corporation 1976

LDAX rp (Load accumulator indirect)

(A) ← ((rp))

The content of the memory location, whose address is in the register pair rp, is moved to register A. Note: only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.



Hexadecimal Format

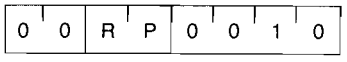
LDAX B 0A
LDAX D 1A

Cycles: 2
States: 7
Addressing: register indirect
Flags: none

STAX rp (Store accumulator indirect)

((rp)) ← (A)

The content of register A is moved to the memory location whose address is in the register pair rp. Note: only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.



Hexadecimal Format

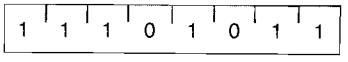
STAX B 02
STAX D 12

Cycles: 2
States: 7
Addressing: register indirect
Flags: none

XCHG (Exchange H and L with D and E)

(H) ↔ (D)
(L) ↔ (E)

The contents of registers H and L are exchanged with the contents of registers D and E.



Hexadecimal Format

XCHG EB

Cycles: 1
States: 4
Addressing: register
Flags: none

All mnemonics copyright ® Intel Corporation 1976

Computer Museum

Arithmetic Group:

This group of instructions performs arithmetic operations on data in registers and memory.

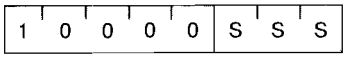
Unless indicated otherwise, all instructions in this group affect the Zero, Parity, Carry, and Auxiliary Carry flags according to the standard rules.

All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow.

ADD r (Add Register)

(A) ← (A) + (r)

The content of register r is added to the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

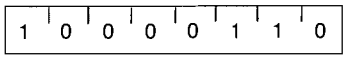
ADD	A	87
ADD	B	80
ADD	C	81
ADD	D	82
ADD	E	83
ADD	H	84
ADD	L	85

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

ADD M (Add memory)

(A) ← (A) + ((H)(L))

The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

ADD M 86

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

ADI byte (Add immediate)

(A) ← (A) + (byte 2)

The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.

Hexadecimal Format

ADI byte C6

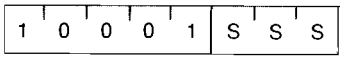
Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

ADC r (Add Register with carry)

(A) ← (A) + (r) + (CY)

The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

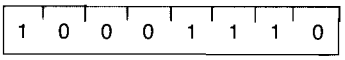
ADC	A	8F
ADC	B	88
ADC	C	89
ADC	D	8A
ADC	E	8B
ADC	H	8C
ADC	L	8D

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

ADC M (Add memory with carry)

(A) ← (A) + ((H)(L)) + (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.



Hexadecimal Format

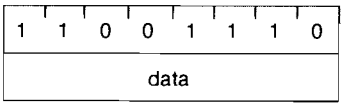
ADC M 8E

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

ACI byte (Add immediate with carry)

(A) ← (A) + (byte 2) + (CY)

The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

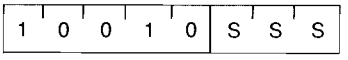
ACI byte CE

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

SUB r (Subtract Register)

(A) ← (A) − (r)

The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

SUB	A	97
SUB	B	90
SUB	C	91
SUB	D	92
SUB	E	93
SUB	H	94
SUB	L	95

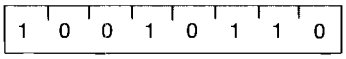
Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

SUB M (Subtract memory)

(A) ← (A) − ((H)(L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

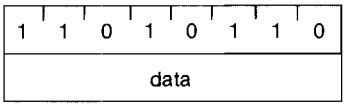
SUB M 96

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

SUI byte (Subtract immediate)

(A) ← (A) − (byte 2)

The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.



Hexadecimal Format

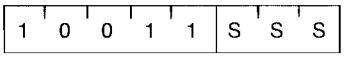
SUI byte D6

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

SBB r (Subtract Register with borrow)

(A) ← (A) − (r) − (CY)

The content of register r and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.



Hexadecimal Format

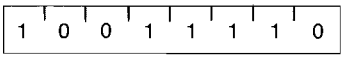
SBB A 9F
SBB B 98
SBB C 99
SBB D 9A
SBB E 9B
SBB H 9C
SBB L 9D

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

SBB M (Subtract memory with borrow)

(A) ← (A) − ((H)(L)) − (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.



Hexadecimal Format

SBB M 9E

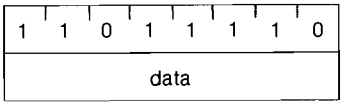
Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

SBI byte (Subtract immediate with borrow)

(A) ← (A) − (byte 2) − (CY)

The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.



Hexadecimal Format

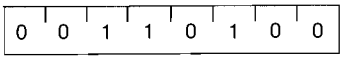
SBI byte DE

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

INR M (Increment memory)

((H)(L)) ← ((H)(L)) + 1

The content of the memory location whose address is contained in the H and L registers is incremented by one. Note: All condition flags except CY are affected.



Hexadecimal Format

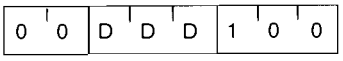
INR M 34

Cycles: 3
States: 10
Addressing: register indirect
Flags: Z, S, P, AC

INR r (Increment Register)

(r) ← (r) + 1

The content of register r is incremented by one. Note: All condition flags except CY are affected.



Hexadecimal Format

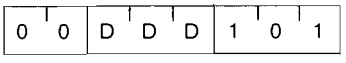
INR A 3C
INR B 04
INR C 0C
INR D 14
INR E 1C
INR H 24
INR L 2C

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, AC

DCR r (Decrement Register)

(r) ← (r) − 1

The content of register r is decremented by one. Note: All condition flags except CY are affected.



Hexadecimal Format

DCR A 3D
DCR B 05
DCR C 0D
DCR D 15
DCR E 1D
DCR H 25
DCR L 2D

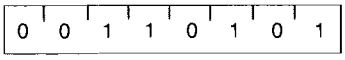
Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, AC

All mnemonics copyright ® Intel Corporation 1976

DCR M (Decrement memory)

((H)(L)) ← ((H)(L)) −1

The content of the memory location whose address is contained in the H and L registers is decremented by one. Note: All condition flags except CY are affected.



Hexadecimal Format

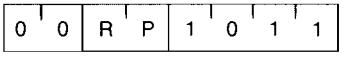
DCR M 35

Cycles: 3
States: 10
Addressing: register indirect
Flags: Z, S, P, AC

DCX rp (Decrement register pair)

(rh)(rl) ← (rh)(rl) −1

The content of the register pair rp is decremented by one. Note: No condition flags are affected.



Hexadecimal Format

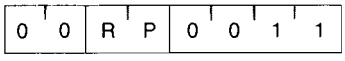
DCX B 0B (register pair B/C)
DCX D 1B (register pair D/E)
DCX H 2B (register pair H/L)
DCX SP 3B (stack pointer)

Cycles: 1
States: 6
Addressing: register
Flags: none

INX rp (Increment register pair)

(rh)(rl) ← (rh)(rl) + 1

The content of the reigster pair rp is incremented by one. Note: No condition flags are affected.



Hexadecimal Format

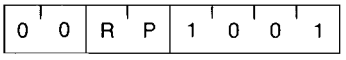
INX B 03 (register pair B/C)
INX D 13 (register pair D/E)
INX H 23 (register pair H/L)
INX SP 33 (stack pointer)

Cycles: 1
States: 6
Addressing: register
Flags: none

DAD rp (Add register pair to H and L)

(H)(L) ← (H)(L) + (rh)(rl)

The content of the register pair rp is added to the content of the register pair H and L. The result is placed in the register pair H and L. Note: Only the CY flag is affected. It is set if there is a carry out of the double precision add; otherwise it is reset.



Hexadecimal Format

DAD B 09 (register pair B/C)
DAD D 19 (register pair D/E)
DAD H 29 (register pair H/L)
DAD SP 39 (stack pointer)

Cycles: 3
States: 10
Addressing: register
Flags: CY

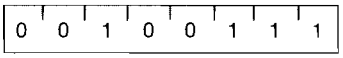
All mnemonics copyright ® Intel Corporation 1976

DAA (Decimal Adjust Accumulator)

The eight-bit number in the accumulator is adjusted to form two four-bit Binary-Coded-Decimal digits by the following process:

1. If the value of the least significant 4 bits of the accumulator is greater than 9 or if the AC flag is set, 6 is added to the accumulator.
2. If the value of the most significant 4 bits of the accumulator is now greater than 9, or if the CY flag is set, 6 is added to the most significant 4 bits of the accumulator.

Note: All flags are affected.



Hexadecimal Format

DAA 27

Cycles: 1
States: 4
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

Logical Group:

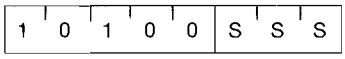
This group of instructions performs logical (Boolean) operations on data in registers and memory and on condition flags.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules.

ANA r (AND Register)

(A) ← (A) ∧ (r)

The content of register r is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set.**



Hexadecimal Format

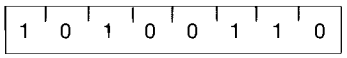
ANA	A	A7
ANA	B	A0
ANA	C	A1
ANA	D	A2
ANA	E	A3
ANA	H	A4
ANA	L	A5

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

ANA M (AND memory)

(A) ← (A) ∧ ((H)(L))

The content of the memory location whose address is contained in the H and L registers is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set.**



Hexadecimal Format

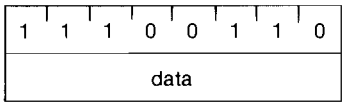
ANA M A6

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

ANI byte (AND immediate)

(A) ← (A) ∧ (byte 2)

The content of the second byte of the instruction is logically anded with the contents of the accumulator. The result is placed in the accumulator. **The CY flag is cleared and AC is set.**



Hexadecimal Format

ANI byte E6

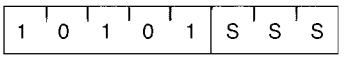
Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

XRA r (Exclusive OR Register)

(A) ← (A) ⊻ (r)

The content of register r is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

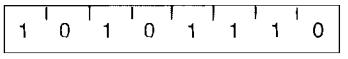
XRA	A	AF
XRA	B	A8
XRA	C	A9
XRA	D	AA
XRA	E	AB
XRA	H	AC
XRA	L	AD

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

XRA M (Exclusive OR Memory)

(A) ← (A) ⊻ ((H)(L))

The content of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

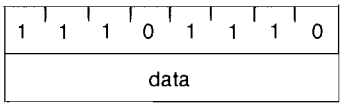
XRA	M	AE

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

XRI byte (Exclusive OR immediate)

(A) ← (A) ⊻ (byte 2)

The content of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

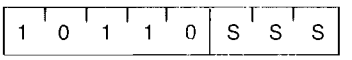
XRI	byte	EE

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

ORA r (OR Register)

(A) ← (A) ∨ (r)

The content of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

ORA	A	B7
ORA	B	B0
ORA	C	B1
ORA	D	B2
ORA	E	B3
ORA	H	B4
ORA	L	B5

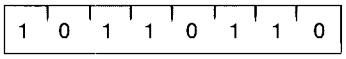
Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

ORA M (OR memory)

(A) ← (A) ∨ ((H)(L))

The content of the memory location whose address is contained in the H and L registers is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

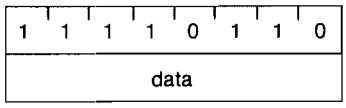
ORA M B6

Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

ORI byte (OR immediate)

(A) ← (A) ∨ (byte 2)

The content of the second byte of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**



Hexadecimal Format

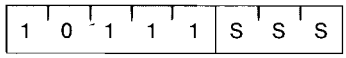
ORI byte F6

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

CMP r (Compare Register)

(A) − (r)

The content of register r is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = (r). The CY flag is set to 1 if (A) < (r).**



Hexadecimal Format

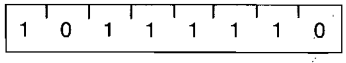
CMP A BF
CMP B B8
CMP C B9
CMP D BA
CMP E BB
CMP H BC
CMP L BD

Cycles: 1
States: 4
Addressing: register
Flags: Z, S, P, CY, AC

CMP M (Compare memory)

(A) − ((H)(L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = ((H)(L)). The CY flag is set to 1 if (A) < ((H)(L)).**



Hexadecimal Format

CMP M BE

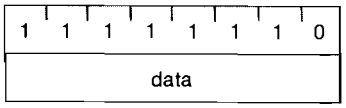
Cycles: 2
States: 7
Addressing: register indirect
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

CPI byte (Compare immediate)

(A) — (byte 2)

The content of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. **The Z flag is set to 1 if (A) = (byte 2). The CY flag is set to 1 if (A) < (byte 2).**



Hexadecimal Format

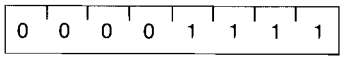
CPI byte FE

Cycles: 2
States: 7
Addressing: immediate
Flags: Z, S, P, CY, AC

RRC (Rotate right)

$(A_n) \leftarrow (A_{n+1})$; $(A_7) \leftarrow (A_0)$
$(CY) \leftarrow (A_0)$

The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. **Only the CY flag is affected.**



Hexadecimal Format

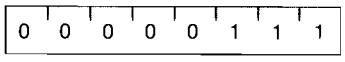
RRC 0F

Cycles: 1
States: 4
Flags: CY

RLC (Rotate left)

$(A_{n+1}) \leftarrow (A_n)$; $(A_0) \leftarrow (A_7)$
$(CY) \leftarrow (A_7)$

The content of the accumulator is rotated left one position. The low order bit and the CY flag are both set to the value shifted out of the high order bit position. **Only the CY flag is affected.**



Hexadecimal Format

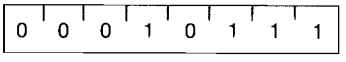
RLC 07

Cycles: 1
States: 4
Flags: CY

RAL (Rotate left through carry)

$(A_{n+1}) \leftarrow (A_n)$; $(CY) \leftarrow (A_7)$
$(A_0) \leftarrow (CY)$

The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. **Only the CY flag is affected.**



Hexadecimal Format

RAL 17

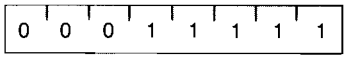
Cycles: 1
States: 4
Flags: CY

All mnemonics copyright ® Intel Corporation 1976

RAR (Rotate right through carry)

$(A_n) \leftarrow (A_{n+1})$; $(CY) \leftarrow (A_0)$
$(A_7) \leftarrow (CY)$

The content of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit. **Only the CY flag is affected.**



Hexadecimal Format

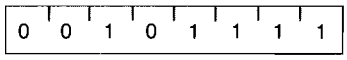
RAR 1F

Cycles: 1
States: 4
Flags: CY

CMA (Complement accumulator)

$(A) \leftarrow (\overline{A})$

The contents of the accumulator are complemented (zero bits become 1, one bits become 0). **No flags are affected.**



Hexadecimal Format

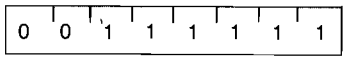
CMA 2F

Cycles: 1
States: 4
Flags: none

CMC (Complement carry)

$(CY) \leftarrow (\overline{CY})$

The CY flag is complemented. **No other flags are affected.**



Hexadecimal Format

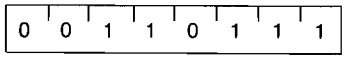
CMC 3F

Cycles: 1
States: 4
Flags: CY

STC (Set carry)

$(CY) \leftarrow 1$

The CY flag is set to 1. **No other flags are affected.**



Hexadecimal Format

STC 37

Cycles: 1
States: 4
Flags: CY

All mnemonics copyright ® Intel Corporation 1976

Branch Group:

This group of instructions alter normal sequential program flow.

Condition flags are not affected by any instruction in this group.

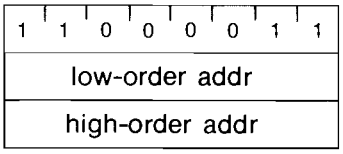
The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The conditions that may be specified are as follows:

CONDITION	CCC
NZ — not zero (Z = 0)	000
Z — zero (Z = 1)	001
NC — no carry (CY = 0)	010
C — carry (CY = 1)	011
PO — parity odd (P = 0)	100
PE — parity even (P = 1)	101
P — plus (S = 0)	110
M — minus (S = 1)	111

JMP addr (Jump)

(PC) ← (byte 3)(byte 2)

Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.



Hexadecimal Format

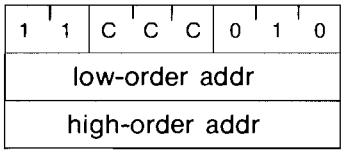
JMP addr C3

Cycles: 3
States: 10
Addressing: immediate
Flags: none

Jcondition addr (Conditional jump)

If (CCC),
(PC) ← (byte 3)(byte 2)

If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.



Hexadecimal Format

JNZ	addr	C2	(jump on no zero)
JZ	addr	CA	(jump on zero)
JNC	addr	D2	(jump on no carry)
JC	addr	DA	(jump on carry)
JPO	addr	E2	(jump on parity odd)
JPE	addr	EA	(jump on parity even)
JP	addr	F2	(jump on positive)
JM	addr	FA	(jump on minus)

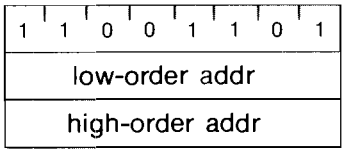
Cycles: 2/3
States: 7/10
Addressing: immediate
Flags: none

All mnemonics copyright ® Intel Corporation 1976

CALL addr (Call)

((SP) − 1) ← (PCH)
((SP) − 2) ← (PCL)
(SP) ← (SP) − 2
(PC) ← (byte 3)(byte 2)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.



Hexadecimal Format

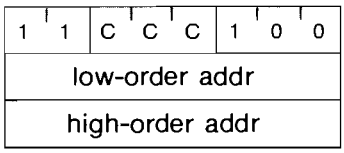
CALL addr CD

Cycles: 5
States: 18
Addressing: immediate/reg. indirect
Flags: none

Ccondition addr (Condition call)

If (CCC),
((SP) − 1) ← (PCH)
((SP) − 2) ← (PCL)
(SP) ← (SP) − 2
(PC) ← (byte 3)(byte 2)

If the specified condition is true, the actions specified in the CALL instruction (see above) are performed; otherwise, control continues sequentially.



Hexadecimal Format

CNZ	addr	C4	(call on no zero)
CZ	addr	CC	(call on zero)
CNC	addr	D4	(call on no carry)
CC	addr	DC	(call on carry)
CPO	addr	E4	(call on parity odd)
CPE	addr	EC	(call on parity even)
CP	addr	F4	(call on positive)
CM	addr	FC	(call on minus)

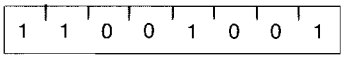
Cycles: 2/5
States: 9/18
Addressing: immediate/reg. indirect
Flags: none

All mnemonics copyright ® Intel Corporation 1976

RET (Return)

(PCL) ← ((SP));
(PCH) ← ((SP) + 1);
(SP) ← (SP) + 2

The content of the memory location whose address is specified in register SP is moved to the low-order eight bits of register PC. The content of the memory location whose address is one more than the content of register SP is moved to the high-order eight bits of register PC. The content of register SP is incremented by 2.



Hexadecimal Format

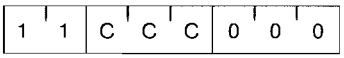
RET C9

Cycles: 3
States: 10
Addressing: register indirect
Flags: none

Rcondition (Conditional return)

If (CCC),
(PCL) ← ((SP))
(PCH) ← ((SP) + 1)
(SP) ← (SP) + 2

If the specified condition is true, the actions specified in the RET instruction (see above) are performed; otherwise, control continues sequentially.



Hexadecimal Format

RNZ	C0	(return on no zero)
RZ	C8	(return on zero)
RNC	D0	(return on no carry)
RC	D8	(return on carry)
RPO	E0	(return on parity odd)
RPE	E8	(return on parity even)
RP	F0	(return on positive)
RM	F8	(return on minus)

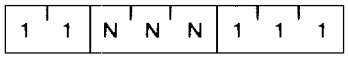
Cycles: 1/3
States: 6/12
Addressing: register indirect
Flags: none

All mnemonics copyright ® Intel Corporation 1976

RST n (Restart)

((SP) — 1) ← (PCH)
((SP) — 2) ← (PCL)
(SP) ← (SP) - 2
(PC) ← 8 * (NNN)

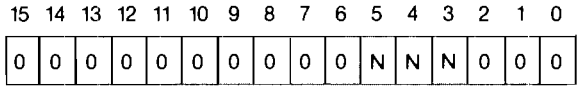
The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the content of NNN.



Hexadecimal Format

RST 0	C7
RST 1	CF
RST 2	D7
RST 3	DF
RST 4	E7
RST 5	EF
RST 6	F7
RST 7	FF

Cycles: 3
States: 12
Addressing: register indirect
Flags: none

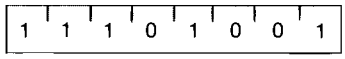


Program Counter After Restart

PCHL (Jump H and L indirect —move H and L to PC)

(PCH) ← (H)
(PCL) ← (L)

The content of register H is moved to the high-order eight bits of register PC. The content of register L is moved to the low-order eight bits of register PC.



Hexadecimal Format

PCHL E9

Cycles: 1
States: 6
Addressing: register
Flags: none

All mnemonics copyright ® Intel Corporation 1976

Stack, I/O, and Machine Control Group:

This group of instructions performs I/O, manipulates the Stack, and alters internal control flags.

Unless otherwise specified, **condition flags are not affected by any instructions in this group.**

FLAG WORD

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	X	AC	X	P	X	CY

X: undefined

PUSH rp (Push)

((SP) − 1) ← (rh)
((SP) − 2) ← (rl)
(SP) ← (SP) − 2

The content of the high-order register of register pair rp is moved to the memory location whose address is one less than the content of register SP. The content of the low-order register of register pair rp is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. **Note: Register pair rp = SP may not be specified.**

1	1	R	P	0	1	0	1

Hexadecimal Format

PUSH	B	C5	(push register pair B and C)
PUSH	D	D5	(push register pair D and E)
PUSH	H	E5	(push register pair H and L)

Cycles: 3
States: 12
Addressing: register indirect
Flags: none

PUSH PSW (Push processor status word)

((SP) − 1) ← (A)
((SP) − 2)$_0$ ← (CY), ((SP) − 2)$_1$ ← X
((SP) − 2)$_2$ ← (P), ((SP) − 2)$_3$ ← X
((SP) − 2)$_4$ ← (AC), ((SP) − 2)$_5$ ← X
((SP) − 2)$_6$ ← (Z), ((SP) − 2)$_7$ ← (S)
(SP) ← (SP) − 2 X: Undefined

The content of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two.

1	1	1	1	0	1	0	1

Hexadecimal Format

PUSH PSW F5

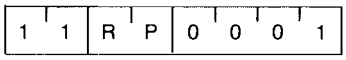
Cycles: 3
States: 12
Addressing: register indirect
Flags: none

All mnemonics copyright ® Intel Corporation 1976

POP rp (Pop)

(rl) ← ((SP))
(rh) ← ((SP) + 1)
(SP) ← (SP) + 2

The content of the memory location whose address is specified by the content of register SP is moved to the low-order register of register pair rp. The content of the rnemory location whose address is one rnore than the content of register SP is rnoved to the high-order register of register pair rp. The content of register SP is incremented by 2. **Note: Register pair rp = SP may not be specified.**



Hexadecimal Format

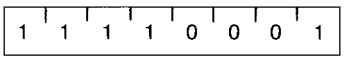
POP B C1 (pop register pair B and C)
POP D D1 (pop register pair D and E)
POP H E1 (pop register pair H and L)

Cycles: 3
States: 10
Addressing: register indirect
Flags: none

POP PSW (Pop processor status word)

(CY) ← ((SP))$_0$
(P) ← ((SP))$_2$
(AC) ← ((SP))$_4$
(Z) ← ((SP))$_6$
(S) ← ((SP))$_7$
(A) ← ((SP) + 1)
(SP) ← (SP) + 2

The content of the memory location whose address is specified by the content of register SP is used to restore the condition flags. The content of the memory location whose address is one more than the content of register SP is moved to register A. The content of register SP is incremented by 2.



Hexadecimal Format

POP PSW F1

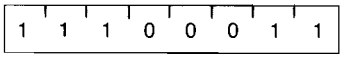
Cycles: 3
States: 10
Addressing: register indirect
Flags: Z, S, P, CY, AC

All mnemonics copyright ® Intel Corporation 1976

XTHL (Exchange stack top with H and L)

(L) ↔ ((SP))
(H) ↔ ((SP) + 1)

The content of the L register is exchanged with the content of the memory location whose address is specified by the content of register SP. The content of the H register is exchanged with the content of the memory location whose address is one more than the content of register SP.



Hexadecimal Format

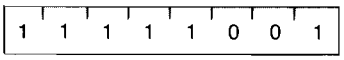
XTHL E3

Cycles: 5
States: 16
Addressing: register indirect
Flags: none

SPHL (Move HL to SP)

(SP) ← (H)(L)

The contents of registers H and L (16 bits) are moved to register SP.



Hexadecimal Format

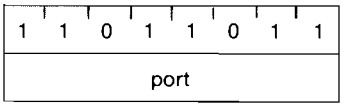
SPHL F9

Cycles: 1
States: 6
Addressing: register
Flags: none

IN port (Input)

(A) ← (data)

The data placed on the eight bit bi-directional data bus by the specified port is moved to register A.



Hexadecimal Format

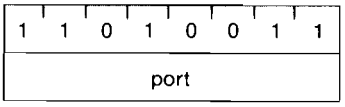
IN port DB

Cycles: 3
States: 10
Addressing: direct
Flags: none

OUT port (Output)

(data) ← (A)

The content of register A is placed on the eight bit bi-directional data bus for transmission to the specified port.



Hexadecimal Format

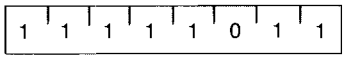
OUT port D3

Cycles: 3
States: 10
Addressing: direct
Flags: none

All mnemonics copyright ® Intel Corporation 1976

EI (Enable interrupts)

The interrupt system is enabled **following the execution of the next instruction.**



Hexadecimal Format

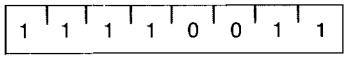
EI FB

Cycles: 1
States: 4
Flags: none

Note: Interrupts are not recognized during the EI instruction.

DI (Disable interrupts)

The interrupt system is disabled **immediately following the execution of the DI instruction.**



Hexadecimal Format

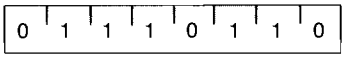
DI F3

Cycles: 1
States: 4
Flags: none

Note: Interrupts are not recognized during the DI instruction.

HLT (Halt)

The processor is stopped. The registers and flags are unaffected.



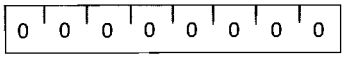
Hexadecimal Format

HLT 76

Cycles: 1
States: 5
Flags: none

NOP (No op)

No operation is performed. The registers and flags are unaffected.



Hexadecimal Format

NOP 00

Cycles: 1
States: 4
Flags: none

All mnemonics copyright ® Intel Corporation 1976

RIM (Read Interrupt Mask)

After the execution of the RIM instruction, the accumulator is loaded with the restart interrupt masks, the Interrupt Enable Flag, any pending interrupts, and the contents of the serial input data line (SID).

When the first RIM instruction is executed following a TRAP interrupt, the IE bit in the accumulator reflects the *previous* interrupt enable status before the TRAP occurred. The first RIM after a TRAP also has the action of releasing the IE status bit such that all subsequent RIM's provide current interrupt enable status.

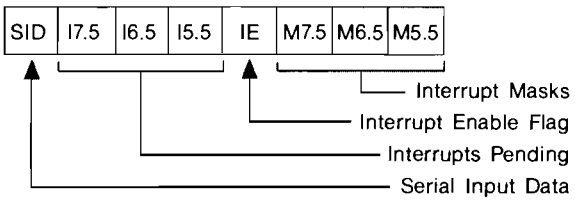
IMPORTANT: The RIM instruction must be executed following a TRAP interrupt in order to restore the correct Interrupt Enable Status.

0	0	1	0	0	0	0	0

Hexadecimal Format

RIM 20

ACCUMULATOR CONTENT AFTER RIM



Cycles: 1
States: 4
Flags: none

SIM (Set Interrupt Masks)

During execution of the SIM instruction, the contents of the accumulator will be used in programming the restart interrupt masks. Bits 0-2 will set/reset the mask bit for RST 5.5, 6.5, 7.5 of the interrupt mask register, if bit 3 is 1 ("set"). Bit 3 is a "Mask Set Enable" control.

Setting the mask (i.e., mask bit = 1) **disables** the corresponding interrupt.

	Set	Reset
RST 5.5 MASK	if bit 0 = 1	if bit 0 = 0
RST 6.5 MASK	bit 1 = 1	bit 1 = 0
RST 7.5 MASK	bit 2 = 1	bit 2 = 0

The RST 7.5 (edge triggered) internal request filp flop will be reset if bit 4 of the accumulator = 1; regardless of whether RST 7.5 is masked or not.

A hardware RESET of the 8085A will set all RST MASKs, and reset/disable all interrupts.

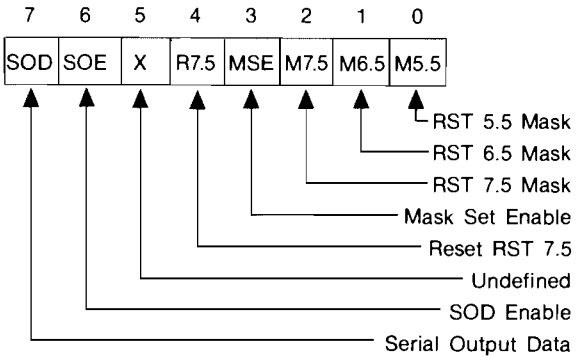
SIM can also load the SOD output latch. Accumulator bit 7 is loaded into the SOD latch if bit 6 is set. The latch is unaffected if bit 6 is a zero. $\overline{\text{RESET IN}}$ input sets the SOD latch to zero.

0	0	1	1	0	0	0	0

Hexadecimal Format

SIM 30

ACCUMULATOR CONTENT BEFORE SIM



Cycles: 1
States: 4
Flags: none

All mnemonics copyright ® Intel Corporation 1976

APPENDIX C

Signature Tables

INTRODUCTION

This appendix contains the signature tables for the μLab. These tables list every IC pin plus the address and data bus lines.

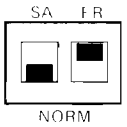
In most cases a signature will be listed for each pin, but there are a number of different ways in which the signature may be represented. If the pin is tied directly to ground or +5V, the table entry will simply indicate GND or Vcc rather than giving the signature. The Vcc signature is listed at the beginning of the table, and the GND signature is always 0000. The table may also show a 1 or 0 signature, which means the same thing as Vcc or GND except that the signal is a gate output, not tied directly to Vcc or GND. Finally, the table may show a Vcc or a GND signature with a B after it, which means the same thing as a 1 or 0 except that the light on the signature analyzer's probe tip should be blinking. For example, in Table C-2 on IC2, pin 3, the signature is 7A70B. This means that although the signal is always at the same logic level when the clock edge arrives, at other times it is at a different level. Refer to Lesson 17 for a complete explanation of how to use the signature tables when troubleshooting the μLab.

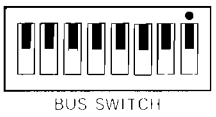
TABLE C-1. FREERUN ADDRESS SIGNATURES

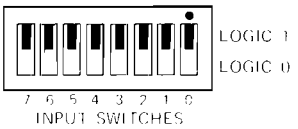
5004A SWITCHES	5036A CONNECTIONS
START ⎍	A15
STOP ⎍	A15
CLOCK ⎍	$\overline{\text{READ}}$

VCC SIGNATURE: 0001

5036A SWITCHES







Data Lines	Address Lines			
D0 X	A0	UUUU	A8	HC89
D1 X	A1	5555	A9	2H70
D2 X	A2	CCCC	A10	HPP0
D3 X	A3	7F7F	A11	1293
D4 X	A4	5H21	A12	HAP7
D5 X	A5	0AFA	A13	3C96
D6 X	A6	UPFH	A14	3827
D7 X	A7	52F8	A15	755P

IC1

GND	1	20	VCC
1293	2	19	0000
1293	3	18	755P
HPP0	4	17	755P
HPP0	5	16	3827
2H70	6	15	3827
2H70	7	14	3C96
HC89	8	13	3C96
HC89	9	12	HAP7
GND	10	11	HAP7

IC2

0000	1	20	VCC
UUUU	2	19	52F8
0001-B	3	18	0000-B
0001-B	4	17	0001-B
5555	5	16	UPFH
CCCC	6	15	0AFA
0001-B	7	14	0001-B
0001-B	8	13	0001-B
7F7F	9	12	5H21
GND	10	11	0001-B

IC3

X	1	40	VCC
X	2	39	0000
0000	3	38	0000
0000 or 0001	4	37	0001-B
0000	5	36	0001
0000	6	35	0001
0000	7	34	0000
0000	8	33	0001
0000	9	32	0001-B
0000	10	31	0001
0001	11	30	0000-B
0001-B	12	29	0001
0001-B	13	28	755P
0001-B	14	27	3827
0001-B	15	26	3C96
0001-B	16	25	HAP7
0001-B	17	24	1293
0001-B	18	23	HPP0
0000-B	19	22	2H70
GND	20	21	HC89

IC4

52F8	1	24	VCC
UPFH	2	23	HC89
0AFA	3	22	2H70
5H21	4	21	0001
7F7F	5	20	3PCF
CCCC	6	19	HPP0
5555	7	18	0000-B
UUUU	8	17	X
X	9	16	X
X	10	15	X
X	11	14	X
GND	12	13	X

IC5

UPFH	1	18	VCC
0AFA	2	17	52F8
5H21	3	16	HC89
7F7F	4	15	2H70
UUUU	5	14	X
5555	6	13	X
CCCC	7	12	X
84AF	8	11	X
GND	9	10	0001

IC6

UPFH	1	18	VCC
0AFA	2	17	52F8
5H21	3	16	HC89
7F7F	4	15	2H70
UUUU	5	14	X
5555	6	13	X
CCCC	7	12	X
84AF	8	11	X
GND	9	10	0001

IC7

1293	1	16	VCC
HAP7	2	15	3PCF
3C96	3	14	84AF
3827	4	13	960F
755P	5	12	4154
0001-B	6	11	UA87
1920	7	10	597C
GND	8	9	C34C

IC8

0001	1	16	VCC
0000	2	15	F770
0001	3	14	F771
GND	4	13	CCCC
UUUU	5	12	5555
8HUC	6	11	U6AH
8HUA	7	10	U6AF
GND	8	9	960F

X = Don't Care Signature B = Blinking GND or VCC Signature

TABLE C-1. FREERUN ADDRESS SIGNATURES (Continued)

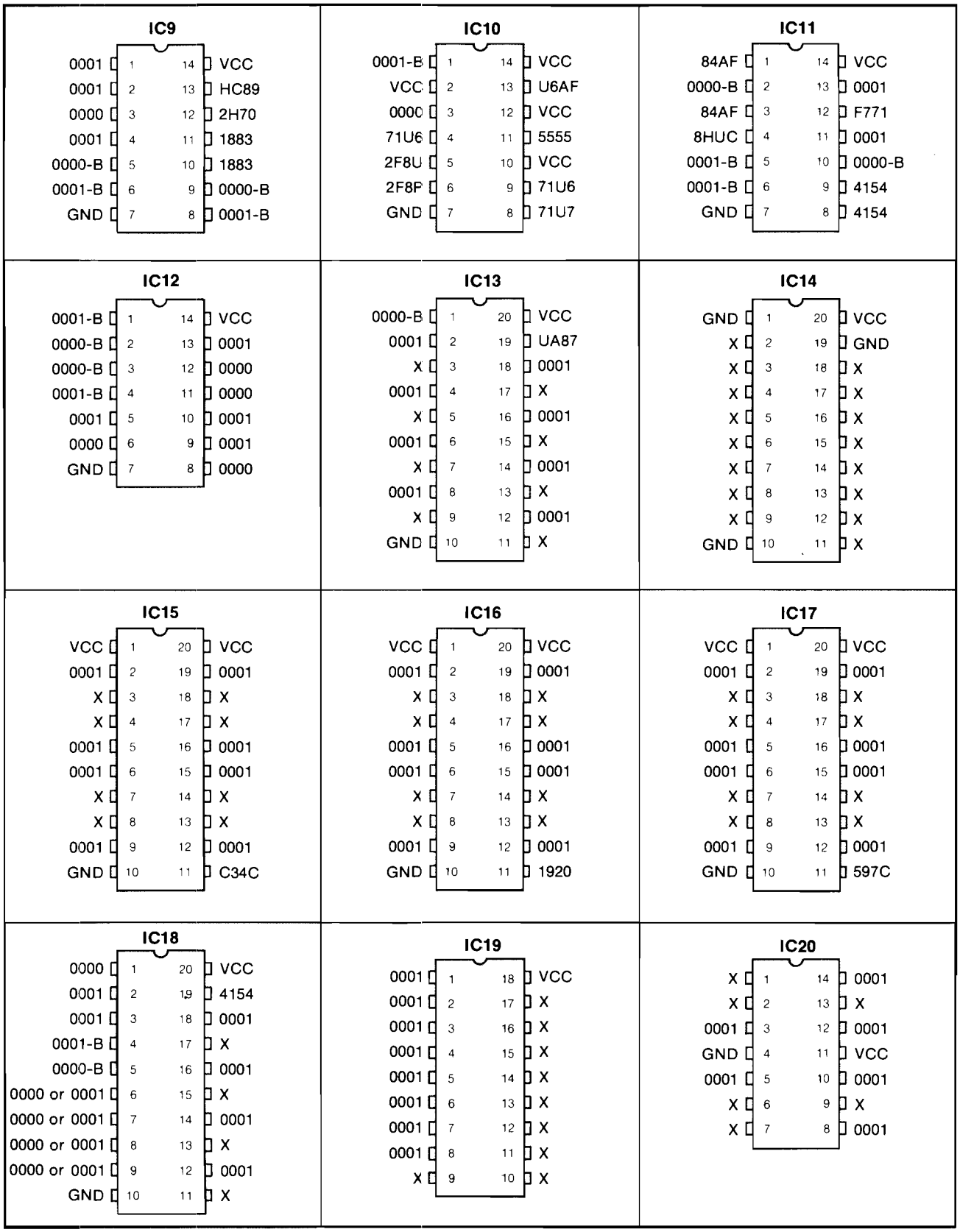
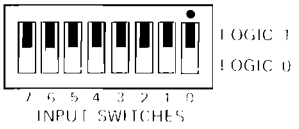


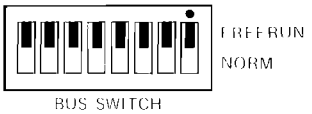
TABLE C-2. FREERUN ROM SIGNATURES

5004A SWITCHES	5036A CONNECTIONS
START ⊓	$\overline{ROM}$
STOP ⊔	$\overline{ROM}$
CLOCK ⊔	$\overline{READ}$

VCC SIGNATURE: 7A70

5036A SWITCHES





Data Lines	Address Lines	
D0 A66U	A0 H62U	A8 9635
D1 H278	A1 C21A	A9 1734
D2 1P32	A2 HA07	A10 8P54
D3 14FH	A3 H0AA	A11 0000-B
D4 A314	A4 P030	A12 0000-B
D5 0HF1	A5 4442	A13 0000-B
D6 1HHA	A6 4U2A	A14 0000-B
D7 AC52	A7 0772	A15 0000-B

IC1

GND	1	20	VCC
0000-B	2	19	0000
0000-B	3	18	0000-B
8P54	4	17	0000-B
8P54	5	16	0000-B
1734	6	15	0000-B
1734	7	14	0000-B
9635	8	13	0000-B
9635	9	12	0000-B
GND	10	11	0000-B

IC2

0000	1	20	VCC
H62U	2	19	0772
7A70-B	3	18	0000-B
7A70-B	4	17	7A70-B
C21A	5	16	4U2A
HA07	6	15	4442
7A70-B	7	14	7A70-B
7A70-B	8	13	7A70-B
H0AA	9	12	P030
GND	10	11	7A70-B

IC3

X	1	40	VCC
X	2	39	0000
0000	3	38	0000
0000 or 7A70	4	37	7A70-B
0000	5	36	7A70
0000	6	35	7A70
0000	7	34	0000
0000	8	33	7A70
0000	9	32	7A70-B
0000	10	31	7A70
7A70	11	30	0000-B
7A70-B	12	29	7A70
7A70-B	13	28	0000-B
7A70-B	14	27	0000-B
7A70-B	15	26	0000-B
7A70-B	16	25	0000-B
7A70-B	17	24	0000-B
7A70-B	18	23	8P54
0000-B	19	22	1734
GND	20	21	9635

IC4

0772	1	24	VCC
4U2A	2	23	9635
4442	3	22	1734
P030	4	21	7A70
H0AA	5	20	0000-B
HA07	6	19	8P54
C21A	7	18	0000-B
H62U	8	17	AC52
A66U	9	16	1HHA
H278	10	15	0HF1
1P32	11	14	A314
GND	12	13	14FH

IC5

4U2A	1	18	VCC
4442	2	17	0772
P030	3	16	9635
H0AA	4	15	1734
H62U	5	14	A66U
C21A	6	13	H278
HA07	7	12	1P32
7A70-B	8	11	14FH
GND	9	10	7A70

IC6

4U2A	1	18	VCC
4442	2	17	0772
P030	3	16	9635
H0AA	4	15	1734
H62U	5	14	A314
C21A	6	13	0HF1
HA07	7	12	1HHA
7A70-B	8	11	AC52
GND	9	10	7A70

IC7

0000-B	1	16	VCC
0000-B	2	15	0000-B
0000-B	3	14	7A70-B
0000-B	4	13	7A70-B
0000-B	5	12	7A70-B
7A70-B	6	11	7A70-B
7A70-B	7	10	7A70-B
GND	8	9	7A70-B

IC8

7A70	1	16	VCC
0000	2	15	7A70-B
7A70	3	14	0000-B
GND	4	13	HA07
H62U	5	12	C21A
0000-B	6	11	0000-B
7A70-B	7	10	7A70-B
GND	8	9	7A70-B

X = Don't Care Signature B = Blinking GND or VCC Signature

TABLE C-2. FREERUN ROM SIGNATURES (Continued)

IC9

7A70	1	14	VCC
7A70	2	13	9635
0000	3	12	1734
7A70	4	11	P5AP
0000-B	5	10	P5AP
7A70-B	6	9	0000-B
GND	7	8	7A70-B

IC10

7A70-B	1	14	VCC
VCC	2	13	7A70-B
0000	3	12	VCC
7A70-B	4	11	C21A
0000-B	5	10	VCC
7A70-B	6	9	7A70-B
GND	7	8	0000-B

IC11

7A70-B	1	14	VCC
0000-B	2	13	7A70
7A70-B	3	12	0000-B
0000-B	4	11	7A70
7A70-B	5	10	0000-B
7A70-B	6	9	7A70-B
GND	7	8	7A70-B

IC12

7A70-B	1	14	VCC
0000-B	2	13	7A70
0000-B	3	12	0000
7A70-B	4	11	0000
7A70	5	10	7A70
0000	6	9	7A70
GND	7	8	0000

IC13

0000-B	1	20	VCC
7A70	2	19	7A70-B
A66U	3	18	7A70
7A70	4	17	AC52
H278	5	16	7A70
7A70	6	15	1HHA
1P32	7	14	7A70
7A70	8	13	0HF1
14FH	9	12	7A70
GND	10	11	A314

IC14

GND	1	20	VCC
A66U	2	19	GND
A66U	3	18	AC52
H278	4	17	AC52
H278	5	16	1HHA
1P32	6	15	1HHA
1P32	7	14	0HF1
14FH	8	13	0HF1
14FH	9	12	A314
GND	10	11	A314

IC15

VCC	1	20	VCC
7A70	2	19	7A70
A66U	3	18	AC52
H278	4	17	1HHA
7A70	5	16	7A70
7A70	6	15	7A70
1P32	7	14	0HF1
14FH	8	13	A314
7A70	9	12	7A70
GND	10	11	7A70-B

IC16

VCC	1	20	VCC
7A70	2	19	7A70
A66U	3	18	AC52
H278	4	17	1HHA
7A70	5	16	7A70
7A70	6	15	7A70
1P32	7	14	0HF1
14FH	8	13	A314
7A70	9	12	7A70
GND	10	11	7A70-B

IC17

VCC	1	20	VCC
7A70	2	19	7A70
A66U	3	18	AC52
H278	4	17	1HHA
7A70	5	16	7A70
7A70	6	15	7A70
1P32	7	14	0HF1
14FH	8	13	A314
7A70	9	12	7A70
GND	10	11	7A70-B

IC18

0000	1	20	VCC
7A70	2	19	7A70-B
7A70	3	18	7A70
7A70-B	4	17	14FH
0000-B	5	16	7A70
0000 or 7A70	6	15	1P32
0000 or 7A70	7	14	7A70
0000 or 7A70	8	13	H278
0000 or 7A70	9	12	7A70
GND	10	11	A66U

IC19

7A70	1	18	VCC
7A70	2	17	X
7A70	3	16	X
7A70	4	15	X
7A70	5	14	X
7A70	6	13	X
7A70	7	12	X
7A70	8	11	X
X	9	10	X

IC20

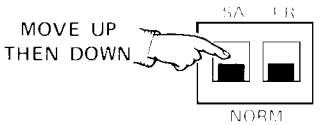
X	1	14	7A70
X	2	13	X
7A70	3	12	7A70
GND	4	11	VCC
7A70	5	10	7A70
X	6	9	X
X	7	8	7A70

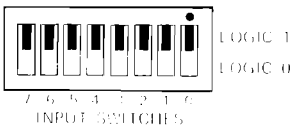
TABLE C-3. S.A. WRITE SIGNATURES

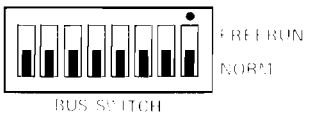
5004A SWITCHES	5036A CONNECTIONS
START ⎍	A15
STOP ⎍	A15
CLOCK ⎍	$\overline{\text{WRITE}}$

VCC SIGNATURE: ICAU

5036A SWITCHES







Data Lines		Address Lines			
D0	937H	A0	UU3P	A8	9P91
D1	C072	A1	209C	A9	0765
D2	F5A5	A2	HHC0	A10	0000-B
D3	UA0P	A3	85A9	A11	1P3H
D4	51H6	A4	6F64	A12	1C32
D5	AU8U	A5	7A21	A13	7A26
D6	2PPH	A6	A2UC	A14	0000-B
D7	H011	A7	3483	A15	9678

IC1

GND	1	20	VCC
1P3H	2	19	0000
1P3H	3	18	9678
0000-B	4	17	9678
0000-B	5	16	0000-B
0765	6	15	0000-B
0765	7	14	7A26
9P91	8	13	7A26
9P91	9	12	1C32
GND	10	11	1C32

IC2

0000	1	20	VCC
UU3P	2	19	3483
937H	3	18	H011
C072	4	17	2PPH
209C	5	16	A2UC
HHC0	6	15	7A21
F5A5	7	14	AU8U
UA0P	8	13	51H6
85A9	9	12	6F64
GND	10	11	1CAU-B

IC3

X	1	40	VCC
X	2	39	0000
0000	3	38	0000
1CAU-B	4	37	ICAU-B
0000	5	36	1CAU
0000	6	35	1CAU
0000	7	34	FP05
0000	8	33	0000-B
0000	9	32	1CAU-B
0000	10	31	1CAU-B
1CAU	11	30	0000-B
937H	12	29	1CAU-B
C072	13	28	9678
F5A5	14	27	0000-B
UA0P	15	26	7A26
51H6	16	25	1C32
AU8U	17	24	1P3H
2PPH	18	23	0000-B
H011	19	22	0765
GND	20	21	9P91

IC4

3483	1	24	VCC
A2UC	2	23	9P91
7A21	3	22	0765
6F64	4	21	1CAU
85A9	5	20	1CAU-B
HHC0	6	19	0000-B
209C	7	18	1CAU-B
UU3P	8	17	H011
937H	9	16	2PPH
C072	10	15	AU8U
F5A5	11	14	51H6
GND	12	13	UA0P

IC5

A2UC	1	18	VCC
7A21	2	17	3483
6F64	3	16	9P91
85A9	4	15	0765
UU3P	5	14	937H
209C	6	13	C072
HHC0	7	12	F5A5
1FFA	8	11	UA0P
GND	9	10	0000-B

IC6

A2UC	1	18	VCC
7A21	2	17	3483
6F64	3	16	9P91
85A9	4	15	0765
UU3P	5	14	51H6
209C	6	13	AU8U
HHC0	7	12	2PPH
1FFA	8	11	H011
GND	9	10	0000-B

IC7

1P3H	1	16	VCC
1C32	2	15	1CAU-B
7A26	3	14	FP05
0000-B	4	13	39U4
9678	5	12	1CAU-B
1CAU-B	6	11	1CAU-B
9377	7	10	58P0
GND	8	9	AA1P

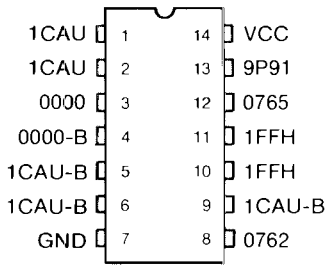
IC8

1CAU	1	16	VCC
0000	2	15	0000
1CAU	3	14	1CAU
GND	4	13	HHC0
UU3P	5	12	209C
0599	6	11	1CAU
1P36	7	10	0000
GND	8	9	39U4

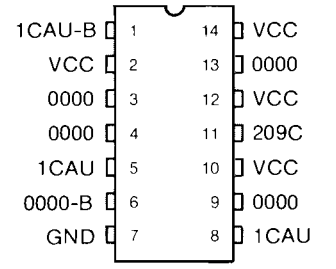
X = Don't Care Signature B = Blinking GND or VCC Signature

TABLE C-3. S.A. WRITE SIGNATURES (Continued)

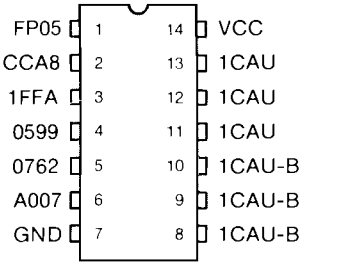
IC9



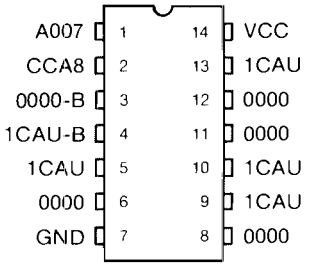
IC10



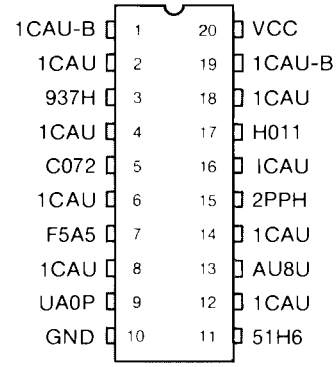
IC11



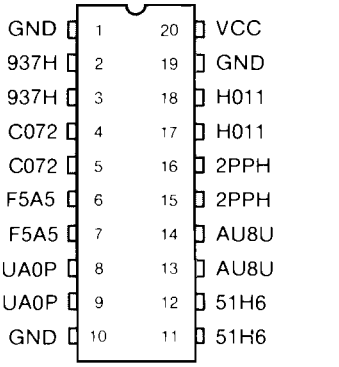
IC12



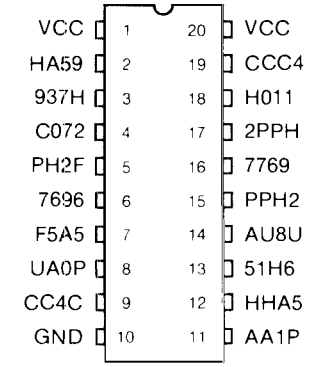
IC13



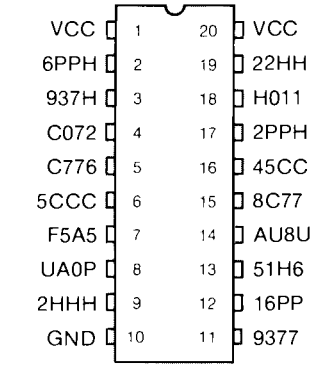
IC14



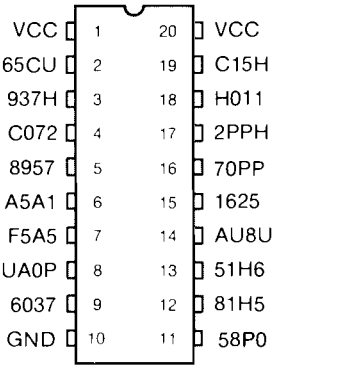
IC15



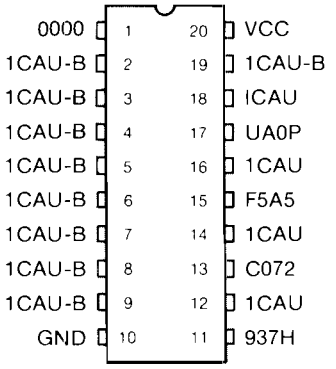
IC16



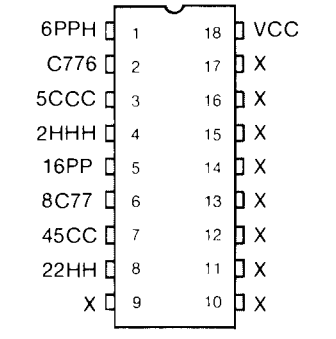
IC17



IC18



IC19



IC20

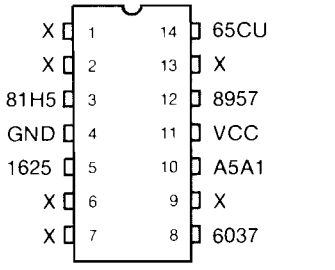
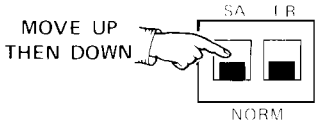


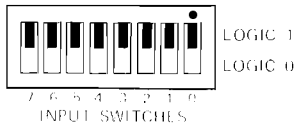
TABLE C-4. S.A. READ SIGNATURES

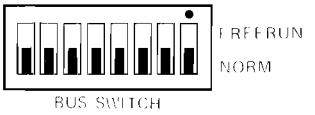
5004A SWITCHES	5036A CONNECTIONS
START ⎍	A15
STOP ⎍	A15
CLOCK ⎍	$\overline{\text{READ}}$

VCC SIGNATURE: AU35

5036A SWITCHES







Data Lines		Address Lines			
D0	0122	A0	773F	A8	3U98
D1	A7FP	A1	4H2F	A9	3U98
D2	8863	A2	6087	A10	2UP0
D3	H3A4	A3	5HA7	A11	AAH3
D4	F616	A4	6757	A12	954C
D5	5C74	A5	HUC6	A13	52A9
D6	P165	A6	1C96	A14	0000-B
D7	36A9	A7	PAU5	A15	78AU

IC1

GND	1	20	VCC
AAH3	2	19	0000
AAH3	3	18	78AU
2UP0	4	17	78AU
2UP0	5	16	0000-B
3U98	6	15	0000-B
3U98	7	14	52A9
3U98	8	13	52A9
3U98	9	12	954C
GND	10	11	954C

IC2

0000	1	20	VCC
773F	2	19	PAU5
0122	3	18	36A9
A7FP	4	17	P165
4H2F	5	16	1C96
6087	6	15	HUC6
8863	7	14	5C74
H3A4	8	13	F616
5HA7	9	12	6757
GND	10	11	AU35-B

IC3

X	1	40	VCC
X	2	39	0000
0000	3	38	0000
13F3	4	37	AU35-B
0000	5	36	AU35
0000	6	35	AU35
0000	7	34	CU4H
0000	8	33	AU35-B
0000	9	32	AU35-B
0000	10	31	AU35-B
AU35	11	30	0000-B
0122	12	29	2F69
A7FP	13	28	78AU
8863	14	27	0000-B
H3A4	15	26	52A9
F616	16	25	954C
5C74	17	24	AAH3
P165	18	23	2UP0
36A9	19	22	3U98
GND	20	21	3U98

IC4

PAU5	1	24	VCC
1C96	2	23	3U98
HUC6	3	22	3U98
6757	4	21	AU35
5HA7	5	20	80H5
6087	6	19	2UP0
4H2F	7	18	0000-B
773F	8	17	36A9
0122	9	16	P165
A7FP	10	15	5C74
8863	11	14	F616
GND	12	13	H3A4

IC5

1C96	1	18	VCC
HUC6	2	17	PAU5
6757	3	16	3U98
5HA7	4	15	3U98
773F	5	14	0122
4H2F	6	13	A7FP
6087	7	12	8863
90AH	8	11	H3A4
GND	9	10	AU35-B

IC6

1C96	1	18	VCC
HUC6	2	17	PAU5
6757	3	16	3U98
5HA7	4	15	3U98
773F	5	14	F616
4H2F	6	13	5C74
6087	7	12	P165
90AH	8	11	36A9
GND	9	10	AU35-B

IC7

AAH3	1	16	VCC
954C	2	15	80H5
52A9	3	14	90AH
0000-B	4	13	AU35
78AU	5	12	3A7P
AU35-B	6	11	UH9F
AU35-B	7	10	AU35-B
GND	8	9	AU35-B

IC8

AU35	1	16	VCC
0000	2	15	0000
AU35	3	14	AU35
GND	4	13	6087
773F	5	12	4H2F
H443	6	11	AU35
7C76	7	10	0000
GND	8	9	AU35-B

X = Don't Care Signature B = Blinking GND or VCC Signature

TABLE C-4. S.A. READ SIGNATURES (Continued)

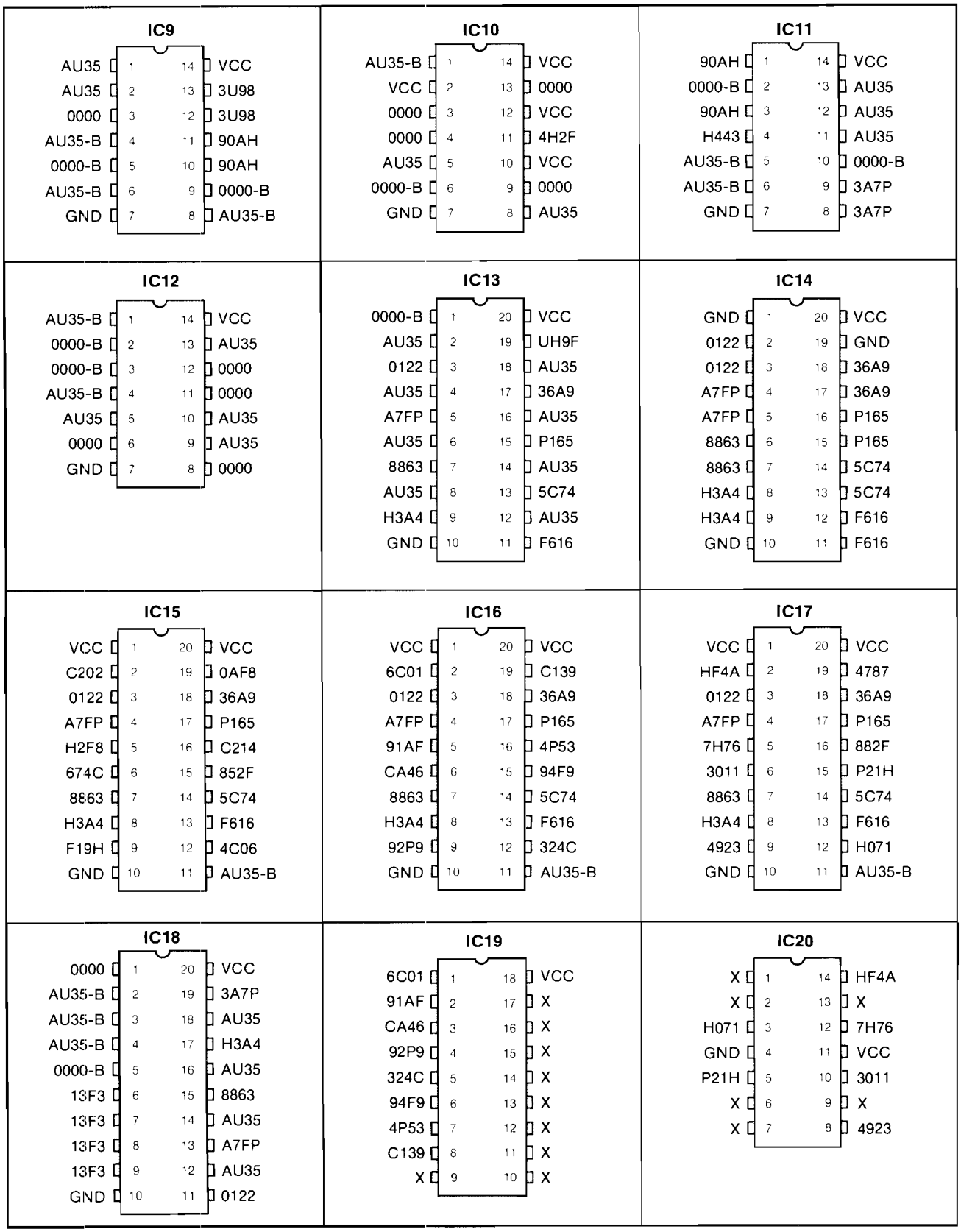
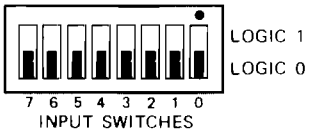


TABLE C-4. S.A. READ SIGNATURES (Continued)

NOTE

Place all INPUT SWITCHES down for the Input Port Test and Keyboard Test and check data line signatures as shown:



Input Port Test

D0	538C
D1	U567
D2	HAFA
D3	810H
D4	94CU
D5	09HH
D6	C3FF
D7	6400

Keyboard Test

D0	F6F0	when 0, 1, 4, 7, A or d key is pressed
D1	602F	when 2, 5, 8, b or E key is pressed
D2	4U81	when 3, 6, 9, C or F key is pressed
D3	1446	when HDWR STEP key is pressed

APPENDIX D

Reading Logic Diagrams

It is absolutely necessary that you be able to read logic diagrams in order to understand and troubleshoot microprocessor-based systems. Should you need a complete review of logic symbology, refer to HP's Practical Digital Electronics Course.

One aspect of logic symbology that seems to cause more confusion than is necessary is centered around the use of the logic low level indicators (small bubbles). These logic low level indicators can be used to establish two different symbols to represent the same physical device. To clarify this important concept, the relationship between the basic AND gate and OR gate is shown in Table D-1. By comparing the symbol changes to the truth tables, you can see that both symbols in each example satisfy the same truth table. In formal Boolean logic this relationship is defined by De Morgan's theorem ($\overline{AB}=\overline{A}+\overline{B}$). Note that while it is not necessary to understand this theorem to understand (and troubleshoot) microprocessor systems, the use of Boolean logic does make some of the more advanced concepts simpler to learn.

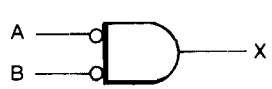
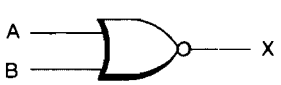
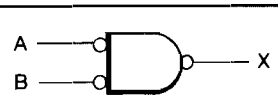
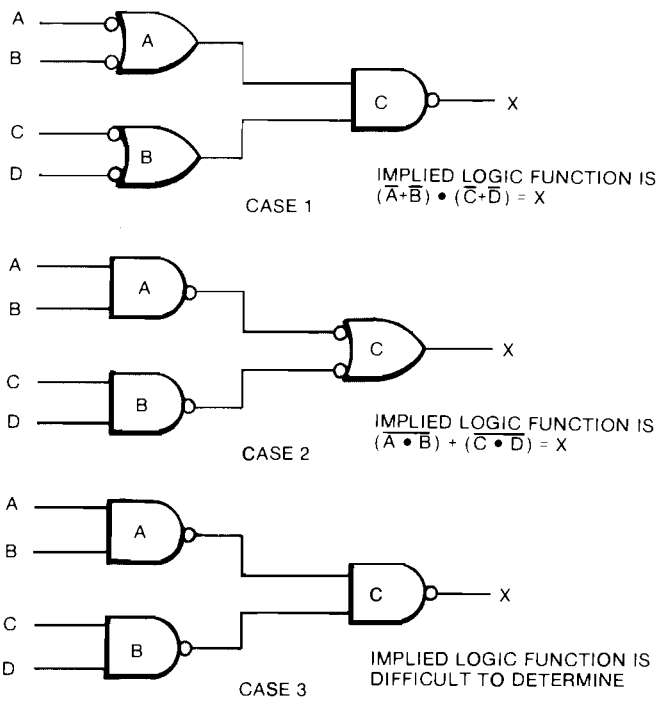
EQUIVALENT SYMBOLS		TRUTH TABLE		
(N)AND	(N)OR	A	B	X
		1 1 0 0	1 0 1 0	1 0 0 0
		1 1 0 0	1 0 1 0	0 1 1 1
		1 1 0 0	1 0 1 0	0 0 0 1
		1 1 0 0	1 0 1 0	1 1 1 0

Table D-1. Basic Gate Relationships

When you read logic diagrams you may notice that the use of these two symbols is sometimes misleading. The reason for this confusion is that the person who drew the diagram did not conform to the original conventions for the use of the symbols. The purpose of having two symbols (for identical gates) is to allow the use of the symbol that shows the functional application of the gate in a specific circuit. Unfortunately, since the designer already knows what the gate does, he sometimes does not bother to differentiate between the same physical gate used in different ways. He shows all of the gates the way they are shown in the manufacturer's specification. To avoid this problem, the gates shown in this course are drawn to indicate the specific functional application in the circuit shown.

As an example of the difference that the use of proper symbology can make, see Figure D-1. All of these circuits are identical and could be implemented using a 7400 gate. The choice of symbols is determined by the information that you wish to convey to the reader. Note the differences in the three cases illustrated. In case 1 you can be fairly confident that low level signals are intended to activate gates A and B and that a low level signal is the active output from gate C. With this knowledge it is easier to check circuit operation. Case 2 is similar but has active high level signals being implied as the inputs and the output of the combined gates. However, in case 3, the same symbol is used for all three gates and it is difficult to determine exactly how the circuit is intended to function. An additional complication is added because the implied active low level outputs of gates A and B are connected to the implied active high level inputs of gate C. The most important point to remember is that circuit operation is identical in all three cases. The only reason to use different symbols is to help the user understand the purpose of the circuits in a specific application. Unfortunately more circuits are drawn using the standard symbol in the manufacturer's specification than are drawn using functional symbols. If you keep in mind the distinctions between the different styles of logic diagrams, it is easier to determine how a specific circuit operates.



TRUTH TABLE

A	B	C	D	X
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure D-1. Functional Logic Diagrams

APPENDIX E

Demonstration and Utility Programs

This appendix contains the procedures required to run the demonstration and utility programs. A complete listing of all of these programs is contained in Appendix F.

DEMONSTRATION PROGRAMS

There are nine demonstration programs and each is run by loading the starting address and pressing RUN. For your convenience, each demonstration program name and starting address is listed in Table E-1. Once you are familiar with each program, this information is all you will need to run the programs. In addition, detailed procedures are provided for your use when required.

Mnemonic	Starting Address	Description
ECHO	04D7	Presents input switch data on output LEDs.
ANDGT	04E0	Causes input switches and output port to function as an AND gate.
CONV	04F8	Conveyor belt controller.
WTM	053E	Well-tempered microprocessor generates random tones from the ROM.
SQRL	055A	Squirrel feedback shift register display.
ORGAN	0599	Keyboard tone generation.
ROCT	05F9	Rocket blast-off.
STW	0662	Stopwatch.
SNAKE	06C2	Snake paddle game.

Table E-1. Demonstration Programs

ECHO Procedure (presents input switch data on output LEDs):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH ADRS key and enter 04D7 on the hexadecimal keys.
c. Press RUN to start the program.
d. Move the INPUT SWITCHES to any random combination of LOGIC 1 and LOGIC 0. Note that the corresponding LEDs reflect whatever is set on the INPUT SWITCHES. Keep in mind that the LED lights when the corresponding INPUT SWITCH is set to LOGIC 0.
e. Press RESET to exit the program.

ANDGT Procedure (causes input switches and output port to function as an AND gate):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH APRS key and enter 04E0 on the hexadecimal keys.
c. Press RUN to start the program.
d. Move all of the INPUT SWITCHES to LOGIC 1 and verify that OUTPUT LED 0 lights.
e. Move any combination of INPUT SWITCHES to LOGIC 0 and verify that OUTPUT LED 0 goes off.
f. Press RESET to exit the program.

CONV Procedure (conveyor belt controller discussed in Lesson 3):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH ADRS key and enter 04F8 on the hexadecimal keys.
c. Press RUN to start the program. The display shows 0, which indicates the number of gears that have passed the electric eye.
d. Press the 0 key. This simulates a pulse from the electric eye. The display changes accordingly.
e. Repeat step d until the display indicates that ten gears have passed. The OUTPUT LEDs then simulate the motion of the box conveyor and an audio tone is generated.
f. Repeat step e to see the process repeat.
g. Press RESET to exit the program.

WTM Procedure (well-tempered microprocessor generates pseudo random tones from the ROM):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH ADRS key and enter 053E on the hexadecimal keys.
c. Press RUN to start the program.
d. Press RESET to exit the program.

SQRL Procedure (squirrel feedback shift register display):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH ADRS key and enter 055A on the hexadecimal keys.
c. Press RUN to start the program.
d. Observe the shifting pattern on the ADDRESS/REGISTER display.

Note

This pattern is generated by an algorithm similar to the one used to compress data in the signature analyzer.

e. Press RESET to exit the program.

ORGAN Procedure (keyboard tone generation):

a. Press the RESET key if the display does not show ␣LAb UP
b. Press the FETCH ADRS key and enter 0599 on the hexadecimal keys.
c. Press RUN to start the program.

Note

The hexadecimal keys now control individual tone generators. The lowest frequency tone is controlled by 0 and the highest frequency tone is controlled by F. Only one key should be pressed at a time. Table E-2 lists the keys and their corresponding notes.

Note	Key
D	0
E	1
F	2
A	3
B	4
middle C	5
D	6
E	7
F	8
G	9
A	A
B	b
C	C
D	d
E	E
F	F

Table E-2. Musical notes for Organ Program

d. Press any sequence of hexadecimal keys. If you have sufficient musical talent you can play simple tunes.
e. Press RESET to exit the program.

ROCT Procedure (rocket blast-off):

a. Press the RESET key if the display does not show ␣L Ab UP
b. Press the FETCH ADRS key and enter 05F9 on the hexadecimal keys.
c. Press RUN to start the program.
d. Observe the countdown on the display followed by the simulated sound and visual effect of a rocket taking off.
e. Press RESET to exit the program or RUN to repeat it.

STW Procedure (stopwatch):

a. Press the RESET key if the display does not show ␣L Ab UP
b. Press the FETCH ADRS key and enter 0662 on the hexadecimal keys.
c. Press RUN to start the program.
d. Verify that the display shows 0 00 00
e. Press the 3 key and verify that the display starts to count in 1/100 of a second increments. Press the 3 key again to halt the timer. Pressing the 3 key will alternately start and stop the timer. The timer can count to 9 minutes 59 seconds 99 one-hundreds of a second before starting again.
f. Press the 0 key to reset the timer.
g. Press RESET to exit the program.

SNAKE Procedure (Snake paddle game):

a. Press the RESET key if the display does not show ␣L Ab UP
b. Press the FETCH ADRS key and enter 06C2 on the hexadecimal keys.
c. Press RUN to start the program.

Note

The rules of the game are simple. The DECR key controls the left paddle and the 3 key controls the right paddle. The left paddle (DECR) starts the game by serving the little red dash to the opposing player (you may play against yourself). Now lets start a game and the remaining rules will be explained as you progress.

d. The player using the left paddle should press the DECR key. Note that the red dash begins to "snake" across the display.
e. The player using the right paddle should wait until the red dash reaches its final position (upper right hand corner of the ADDRESS/REGISTER display) before pressing the 3 key.

Note

The most important skill to develop in the snake paddle ball game is to learn to correctly judge the amount of time the red dash remains in each position as it travels across the display. This time judgement is important because the amount of time the dash remains in the home (final) position determines the speed of the return (see Table E-3). The objective of timing the return is to be able to press the key during period 3. If you time your response correctly, the resulting faster return is more difficult for your opponent to handle. An audible trill signals the change in return speed (faster or slower).

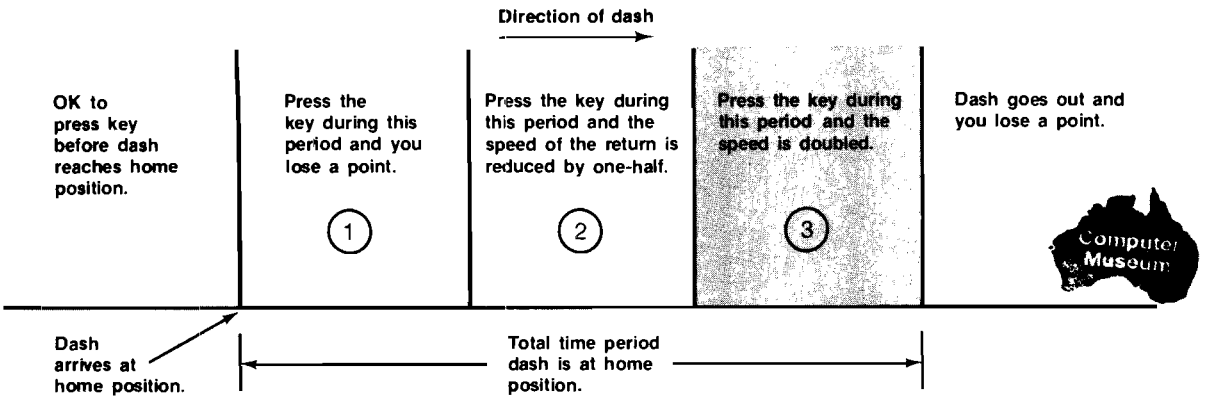


Table E-3. Timing the Return

f. Now, whoever has the red dash stationary on his side has the serve and should start the volley by pressing the control key (DECR or 3).
g. Continue to volley and try to return the red dash faster by correctly timing the return.
h. Whenever either player misjudges the timing for the return, his opponent scores a point. The speaker sounds to indicate a point has been scored. The score for each player is shown on the appropriate side of the DATA display. Each time you score a point you either become the serving player (if your opponent had the serve) or retain the serve if you had it on the previous serve. The position of the dash on the display indicates which player has the serve. The game is set for 10 points. Therefore, the first player to show the hexadecimal character A on his side is the winner. Note that pressing the control keys has no effect once a player has won a game.
i. To start another game press the 0 key. This resets the scoreboard and enables the left paddle player to begin the game by serving the red dash to his opponent.
j. Press RESET to exit the program.

UTILITY PROGRAMS

The utility programs are subroutines that you can use when writing your own programs. Table E-4 lists the utility programs and gives the starting address, the registers that are modified by the subroutine, the required input data, and a brief description of the program. For some examples of the use of these utility programs, refer to Experiments 14-1 and 14-3.

Program Name	Registers Modified	Data Required	Starting Address	Comments
BEEP	All	None	0010*	Produces a single beep of fixed frequency and duration. Can also be called by coding a D7 into your program. D7 is a Restart 2 command (RST 2).
BEEP 1	All except B	Set B for freq	0012	Produces a single beep of fixed duration at the frequency established by register B. When B is set to a small value, the beep tone is a high frequency (the minimum value allowed is 01). When B is set to a larger value, the tone is lower in frequency.
BEEP 2	All except B and D	Set B for freq. and D for duration	0447	Produces a single beep with the frequency established by register B (see BEEP 1). The duration of the beep is established by register D. The larger the value in D the longer the duration of the beep (the minimum value allowed is 01).
KIND (key input and Decode)	A	keyboard inputs	014B	Updates the display while scanning, debouncing, and decoding the keyboard inputs. The value of the pressed key (0-F) is placed in the A register.
KPU (Key Pushed)	A, H, and L	keyboard input	0185	Scans keyboard and clears the zero flag if any key is pushed.
SDS (Scan Display Segments)	None	display segments	01C8	The segment data stored in RAM locations 0BFA to 0BFF are sent to display digits 0 to 5. Each digit is on for 1 ms during the display scan (see Table E-5).

Table E-4. Utility Routines

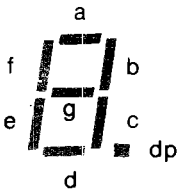
Program Name	Registers Modified	Data Required	Starting Address	Comments
DCD (Display Character Decoder)	None	display digits	01E9	Takes the display digit character codes(00 to IC) stored in RAM display locations 0BF0 to 0BF5 and codes them into the seven segment format. It then stores these codes in RAM display locations 0BFA to 0BFF and scans the display one time. The decoding is done using the DCC table at address 0218 (see Tables E-6 and E-7).
STDM (Store Display Message)	All	DE register pair	0018*	Takes the six character message beginning at the address specified in the DE register pair and stores it in UDSP display RAM locations 0BF0 to 0BF5. The first character of the message appears in the far right position of the display. This routine is normally followed by the DCD subroutine. The STDM routine can also be called by coding a DF into your program. DF is a Restart 3 command (RST 3).
DELA (Fixed Delay)	None	None	0429	Causes a fixed 1 ms delay to occur.
DELB (Variable Delay)	None	BC register pair	0430	Causes a variable delay to occur. The delay is equal to approximately 1 ms times the numerical value in the BC register pair.
RS1	None	None	0008*	Software breakpoint for return to monitor. CF is the RST 1 command.

*RST can be used (see comments)

Table E-4. Utility Routines (Continued)

DDSP 0-5, Adrs 0BFA — 0BFF

Active low, a zero bit turns on the segment



data byte

7	6	5	4	3	2	1	0
dp	g	f	e	d	c	b	a

RAM Address	Label		Decoded Display Digit (in segment code)
0BFA	DDSP0	0	(rightmost digit)
0BFB	DDSP1	1	
0BFC	DDSP2	2	
0BFD	DDSP3	3	
0BFE	DDSP4	4	
0BFF	DDSP5	5	(leftmost digit)

Table E-5. Decoded Display Digits

Display Character	Character Code in 0BF0-0BF5 Addresses
0	00
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	08
9	09
A	0A
b	0B
C	0C
d	0D
E	0E
F	0F
(blank)	10
H	11
L	12
n	13
P	14
o	15
U	16
-	17
r	18
t	19
Y	1A
'	1B
-	1C

Table E-6. Display Character Decoder

UDSP 0-5, Adrs 0BF0 — 0BF5

Hex character codes are decoded using the Display Code Converter table (DCC) starting at address 0218 in the program listing.

RAM Address	Label	Undecoded Display Digit (in hex code)
0BF0	UDSP0	0 (rightmost digit)
0BF1	UDSP1	1
0BF2	UDSP2	2
0BF3	UDSP3	3
0BF4	UDSP4	4
0BF5	UDSP5	5 (leftmost digit)

Table E-7. Undecoded Display Digits

APPENDIX F

Microprocessor Lab ROM Listing

This appendix contains the complete listing of the monitor, utility and demonstration programs stored in the ROM. The procedures for using the demonstration and utility programs is contained in Appendix E.

Table F-1 contains an alphabetical list of the symbols and labels and the hexadecimal addresses at which they are used in the program. Table F-2 is the listing of the ROM program. The first column is the hexadecimal address for the instruction. Note that when an instruction opcode requires more than one location, the list is discontinuous. For example, the JMP opcode at location 0005 requires three memory locations. The address sequence at that point jumps from 0005 to 0008 indicating that the JMP opcode used three memory locations. The second column is the hexadecimal contents of that address. Keep in mind that address information is entered least significant byte first. Therefore, in location 0005, the C34000 code causes a jump to location 0040. In the third column the user symbols that are used as labels are shown. The fourth column contains the instruction mnemonic. A complete explanation of these codes and mnemonics are contained in Appendix B. One important point to remember is that the 8085 assembler expects all numerical data to be in decimal. When hexadecimal data is used, an H is inserted after the number. The last column contains brief programmer comments. General comments and descriptions starting at the beginning of the fourth column and extending through the fifth column are placed at the start of programs, subroutines, or modules.

ALL	A 0281	ANDGT	A 04E0	BEEP	A 0010	BEEP1	A 0012	BEEP2	A 044
BEEP5	A 0464	BEEP6	A 046F	BEEP7	A 047E	BEEP8	A 0480	BEEP9	A 048
CDCRM	A 02B3	CDCRR	A 02CD	CFETA	A 02B8	CFETP	A 02BD	CFETR	A 020
CINSS	A 02A4	CODE	A 05BB	CONV	A 04F8	CRUN	A 02A9	CSTRM	A 02A
DCD	A 01E9	DCD1	A 01F3	DCD2	A 0210	DCRM	A 03AA	DCRR	A 03F
DDSP5	A 0BFF	DEL1	A 0431	DEL2	A 0434	DEL3	A 0436	DELA	A 042
DLY	A 05AE	DMT	A 0241	DPS	A 0319	DSP	A 0038	DTIME	A 005
ENDGM	A 06F7	FETA	A 0328	FETA1	A 032F	FETA2	A 034D	FETA3	A 036
FETA6	A 0393	FETA7	A 039C	FETAR	A 0329	FETCH	A 0247	FETR	A 02D
FETR3	A 02F1	FETR4	A 02FE	FETR5	A 0300	FETR6	A 0310	FLG	A 025
FSTTN	A 0001	HDSS	A 03F7	ICX	A 0287	IM	A 027D	INCR	A 001
KIND	A 014B	KIND1	A 014D	KIND2	A 0156	KIND3	A 0164	KIND4	A 017
KPU	A 0185	KPU1	A 0190	KRD	A 019A	KRD1	A 01A3	LEFT	A 00F
LOSTN	A 0028	LOUT	A 0030	LSM	A 07D4	MA	A 024D	MB	A 025
ME	A 0261	MERR	A 00DC	MERR1	A 00E5	MERR2	A 00ED	MH	A 026
MLP	A 0518	MOVE	A 0514	MSP	A 0BCE	NCTL	A 034A	NOKEY	A 05D
ORGAN	A 0599	ORGAN1	A 059B	PADL	A 07A2	PASS	A 0738	PASS1	A 073
PCL	A 0279	PLAY	A 075A	PLAY1	A 0767	PNTLS	A 00FF	PPER	A 00C
RAML2	A 0BD7	RBND	A 070C	RBND1	A 0727	READ	A 05C0	RESET	A 000
ROCM	A 065C	ROCT	A 05F9	ROCT1	A 0601	ROCT2	A 0603	ROCT3	A 061
RS	A 0BD6	RS1	A 0008	RS4	A 0020	RS4C	A 0AF0	RS5	A 002
RS5C	A 0AF3	RS6	A 0030	RS65	A 0034	RS65C	A 0AFC	RS6C	A 0AF
RSM	A 07E2	RSRV	A 0719	RSTGM	A 0799	RUN	A 03C2	RUN1	A 03C
SATL2	A 04A1	SATL3	A 04A7	SATL4	A 04AF	SAVA	A 0BE7	SAVE	A 0BE
SAVPC	A 0BDC	SAVSH	A 0BDF	SAVSL	A 0BDE	SCAN	A 0028	SDM	A 023
SDS1	A 01D0	SERV	A 06C8	SERV1	A 06EB	SERV2	A 0700	SHIFT	A 05C
SNAKE	A 06C2	SPEED	A 0B00	SPH	A 026D	SPKR	A 05B5	SPL	A 027
SQRL2	A 057B	SQRL3	A 0586	STDM	A 0018	STRM	A 03AF	STRR	A 041
STRT2	A 0064	STRT3	A 0073	STRT4	A 008A	STRT5	A 009C	STRT6	A 00A
STW1	A 0669	STW2	A 0693	STW3	A 069B	STW4	A 06A4	STW5	A 06A
STWM	A 06BC	TABLE	A 05E2	TIME	A 008C	TIME1	A 07B0	TIMER	A 07A
TRAP	A 0024	TRIL	A 07B9	TRIL1	A 07BF	TRIL2	A 07CD	TRP	A 00F
TRP3	A 0111	TRP4	A 0115	TRP5	A 011B	TRP6	A 0124	TRP7	A 014
UDKY	A 0BE8	UDSP0	A 0BF0	UDSP1	A 0BF1	UDSP2	A 0BF2	UDSP3	A 0BF
UR	A 0AEF	USP	A 0BB0	WTM	A 053E	WTM1	A 0541	XLOOP	A 07F

Table F-1. User Symbols

```
Hex
Adrs Contents  Label    Instruction       Comments

0000                    ORG   0
0000 [illegible]03 RLTE:  MVI  H,[illegible]  ;[illegible] ADRS OF [illegible] TEST RAM
0002 [illegible]        MOV   A,M         ;FIRST RAM CELL [illegible]
0003 2F                 CMA               ;COMPLEMENT IT
0004 77                 MOV   M,A         ;STORE IT BACK [illegible]
0005 C34300             JMP   STRT        ;CONTINUE
                        ;
                        ;RST1 IS MAIN ENTRY POINT TO [illegible]
                        ;
0008                    ORG   8
0008 22[illegible]08 RST1:  SHLD TSAVH    ;[illegible] USER HL [illegible] IN RAM
000B D313               OUT   13H         ;[illegible]PROTECT RAM
000D C3F300             JMP   TRP         ;CONTINUE
                        ;
                        ;BEEP PRODUCE [illegible] TONE [illegible] DURATION
                        ;
0010                    ORG   10H
0010 0605       BEEP:   MVI   B,FREQ      ;DEFAULT FREQUENCY
0012 1604       BEEP1:  MVI   D,DURA      ;DEFAULT DURATION
0014 C34704             JMP   BEEP2       ;CONTINUE
0017 00                 NOP
                        ;
                        ;STDM STORES [illegible] MESSAGE [illegible]
                        ;
0018                    ORG   18H
0018 E5         STDM:   PUSH  H
0019 21F00B             LXI   H,[illegible]  ;ADRS OF [illegible]
001C C33503             JMP   STD         ;CONTINUE
001F 00                 NOP
0020                    ORG   20H
0020 C3F30B RS4:        JMP   RS4C        ;[illegible] ROUTINE
0023 00                 NOP
0024                    ORG   24H
0024 C3[illegible]300 TRAP:  JMP  RS1     ;[illegible] USER REG [illegible]
0027 00                 NOP
0028                    ORG   28H
0028 C3F30B RS5:        JMP   RS51        ;[illegible] ROUTINE
002B 00                 NOP
002C                    ORG   2CH
002C C3F50B RS55:       JMP   RS55C       ;[illegible] ROUTINE
002F 00                 NOP
0030                    ORG   30H
0030 C3F30B RS6:        JMP   RS6C        ;[illegible] ROUTINE
0033 00                 NOP
0034                    ORG   34H
0034 C3F10B RS65:       JMP   RS65C       ;[illegible] INTERRUPT [illegible]
0037 00                 NOP
0038                    ORG   38H
0038 C3A600 RS7:        JMP   STRT6       ;[illegible] TO [illegible]
003B 00                 NOP
003C                    ORG   3CH
003C [illegible]  RS75:  RET  [illegible]
003D C[illegible]3004   JMP   FAIL1       ;[illegible] LOOP
                        ;
                        ;POWER UP SELF [illegible] AND INITIALIZE
                        ;
0040                    ORG   40H
0040 BE         STRT:   CMP   M           ;[illegible] DATA [illegible] IN RAM
0041 C2[illegible]300   JNZ   PPER        ;IF RAM WAS [illegible]
0044 31[illegible]0B    LXI   SP,USP      ;INITIALIZE MONITOR SP
0047 AF                 XRA   A           ;CLEAR A
0048 67                 MOV   H,A         ;CLEAR H
0049 6F                 MOV   L,A         ;CLEAR L
004A D3[illegible]      OUT   LOUT        ;[illegible] ON OUTPUT LEDS
                        ;
                        ;ROM SELF TEST
                        ;
004C 86         STRT1:  ADD   M           ;ADD ROM DATA TO A
004D 23                 INX   H           ;POINT TO NEXT ROM ADRS
004E 4F                 MOV   C,A         ;SAVE A RESIDUE [illegible]
004F 3E03               MVI   A,[illegible]  ;LAST ADRS OF ROM [illegible]
0051 BC                 CMP   H           ;COMPARE IT TO H
0052 79                 MOV   A,C         ;RESTORE RESIDUE [illegible]
0053 C24C00             JNZ   STRT1       ;[illegible] LAST ROM ADRS [illegible]
0056 2B                 DCX   H           ;POINT HL TO [illegible] CHECKSUM VALUE
0057 96                 SUB   M           ;[illegible] CHECK [illegible] FOR A RESIDUE
0058 BE                 CMP   M           ;COMPARE RESIDUE TO THE [illegible]
0059 0604               MVI   B,04        ;[illegible] (ROM) MESSAGE
005B C2E500             JNZ   MERR1       ;IF NOT A MATCH
                        ;
                        ;RAM SELF TEST
                        ;
005E AF                 XRA   A           ;CLEAR A
005F 210008             LXI   H,0800H     ;FIRST RAM ADDR
0062 0603               MVI   B,3         ;ADD CONSTANT
0064 77         STRT2:  MOV   M,A         ;STORE DATA IN RAM
0065 80                 ADD   B           ;ADD 3 TO A
0066 23                 INX   H           ;POINT TO NEXT RAM ADRS
0067 4F                 MOV   C,A         ;SAVE A
0068 7C                 MOV   A,H         ;GET MSBYTE ADRS
0069 FE0C               CPI   0CH         ;LAST RAM ADRS+1
006B 79                 MOV   A,C         ;RESTORE A
006C C26400             JNZ   STRT2       ;IF NOT LAST RAM ADRS
006F AF                 XRA   A           ;CLEAR A
0070 210008             LXI   H,0800H     ;1ST RAM ADRS
0073 BE         STRT3:  CMP   M           ;DID DATA GET STORED IN RAM?
0074 C2D100             JNZ   MERR        ;IF DATA NOT SAME
0077 2F                 CMA
0078 77                 MOV   M,A         ;STORE COMPLEMENT BACK IN RAM
0079 BE                 CMP   M           ;DID IT STORE?
007A C2D100             JNZ   MERR        ;IF NOT
007D 2F                 CMA               ;RECOMPLEMENT A
007E 80                 ADD   B           ;ADD 3 TO A
007F 23                 INX   H           ;NEXT RAM ADDR
0080 4F                 MOV   C,A         ;SAVE A
0081 7C                 MOV   A,H         ;GET MSBYTE ADRS
0082 FE0C               CPI   0CH         ;LAST RAM ADRS+1
0084 79                 MOV   A,C         ;RESTORE A

0085 C27300             JNZ   STRT3       ;IF [illegible] NEXT [illegible]
                        ;
                        ;DISPLAY TEST
                        ;
0088 06[illegible]      MVI   B,[illegible]  ;[illegible] LOOP [illegible]
008A 119102 STRT4:      LXI   D,[illegible]  ;[illegible] MESSAGE [illegible]
008D DF                 RST   3           ;GET MESSAGE
008E CDE501             CALL  DCT         ;[illegible] DISPLAY
0091 05                 DCR   B           ;DEC LOOP [illegible]
0092 C2[illegible]00    JNZ   STRT4       ;IF NOT DONE
                        ;
                        ;CLEAR RAM [illegible]
                        ;
0095 0E00               MVI   C,0         ;CLEAR A
0097 3E0C               MVI   A,0CH       ;[illegible] ADRS [illegible]
0099 210008             LXI   H,0800H     ;[illegible] ADRS OF RAM
009C 71         STRT5:  MOV   M,C         ;CLEAR RAM LOCATION
009D 23                 INX   H           ;POINT TO NEXT [illegible]
009E BC                 CMP   H           ;[illegible] RAM ADRS
009F C29C00             JNZ   STRT5       ;IF NOT DONE [illegible]
00A2 3EFF               MVI   A,[illegible]  ;[illegible]
00A4 D330               OUT   [illegible]    ;[illegible]
00A6 D7         STRT6:  RST   2           ;SIGNAL START [illegible]
                        ;
                        ;INITIALIZE REGISTERS
                        ;
00A7 2[illegible]B00E   LXI   H,[illegible]  ;[illegible] DEFAULT [illegible]
00AA 22DE08             SHLD  SAVSP       ;[illegible] IT IN RAM
00AD 3[illegible]D60B   MVI   A,[illegible]  ;[illegible] STATUS WORD [illegible]
00B0 3600               MVI   M,0         ;[illegible] STATUS TO [illegible]
00B2 3E0E               MVI   A,[illegible]  ;DEFAULT INTERRUPT [illegible]
00B4 32DB08             STA   SAVIM       ;STORE IT IN RAM
00B7 210008 STRT7:      LXI   H,[illegible]  ;DEFAULT PC
00BA 22DC08             SHLD  SAVPC       ;STORE IT IN RAM
00BD 3EFF               MVI   A,0FFH      ;[illegible] INSTR [illegible]
00BF 32FF08             STA   TPS         ;STORE IT IN [illegible]
00C2 32EF08             STA   [illegible]    ;[illegible] IT IN [illegible]
00C5 C31101             JMP   TRP3        ;JUMP TO MONITOR
                        ;
                        ;PUSH-BTN ERROR [illegible]
                        ;
00C8 [illegible]E0E PPER:  LXI  [illegible]  ;[illegible] REGISTER [illegible]
00CB 210008             LXI   H,[illegible]  ;DEFAULT PC
00CE 22DC08             SHLD  SAVPC       ;STORE IT IN [illegible]
00D1 D7                 RST   2           ;SIGNAL AN ERROR
00D2 AF                 XRA   A           ;CLEAR A
00D3 32E[illegible]0B   STA   [illegible]    ;[illegible]
00D6 115D02             LXI   D,PPM       ;[illegible] ERROR [illegible]
00D9 C3150[illegible]   JMP   TRP4
                        ;
                        ;MEMORY ERROR [illegible]
                        ;
00DC 0602       MERR:   MVI   B,2         ;[illegible] FAIL [illegible]
00DE AE                 XRA   M           ;[illegible] DIFFERENCE [illegible]
00DF E60F               ANI   0FH         ;[illegible]
00E1 CAE500             JZ    MERR1       ;IF PROBLEM [illegible]
00E4 05                 DCR   B           ;[illegible] TO [illegible]
00E5 119701 MERR1:      LXI   D,[illegible]  ;[illegible] MESSAGE [illegible]
00E8 DF                 RST   3           ;[illegible] MESSAGE
00E9 21F208             LXI   H,[illegible]  ;[illegible]
00EC 70                 MOV   M,B         ;[illegible] NUMBER [illegible]
00ED CDE901 MERR2:      CALL  DCT         ;DISPLAY MESSAGE
00F0 C3ED00             JMP   MERR2       ;LOOP MESSAGE
                        ;
                        ;RST1 IS MAIN ENTRY POINT [illegible]
                        ;
00F3 210200 TRP:        LXI   H,[illegible]  ;CLEAR H,L
00F6 DCFA00             [illegible]  TRP1   ;[illegible]
00F9 2E                 [illegible]  H      ;[illegible] TO [illegible]
00FA 39         TRP1:   DAD   SP          ;[illegible] VALUE INTO [illegible]
00FB D2FF00             JNC   TRP2        ;[illegible] TO TRP1 [illegible]
00FE 23                 INX   H           ;[illegible] OCCURRED [illegible]
00FF 22E108 TRP2:       SHLD  TSPSP       ;[illegible] USER SP IN RAM
0102 31D10B             LXI   SP,SAVSP    ;[illegible] ADRS
0105 F5                 PUSH  PSW         ;SAVE PSW AND [illegible]
0106 21D60B             LXI   H,[illegible]  ;[illegible] STATUS ADR
0109 AF                 XRA   A           ;CLEAR A
010A BE                 CMP   M           ;[illegible] STATUS [illegible]
010B 32F60B             STA   [illegible]    ;CLEAR DATA [illegible]
010E C22401             JNZ   TRP6        ;IF CAME FROM USER [illegible]
0111 114102 TRP3:       LXI   D,IMT       ;[illegible] MESSAGE ADRS
0114 FB                 EI
0115 3E0B       TRP4:   MVI   A,TIM       ;DEFAULT INTERRUPT [illegible]
0117 30                 SIM               ;[illegible] TO ENABLE [illegible]
0118 D310               OUT   10H         ;UNPROTECT RAM
011A DF                 RST   3           ;GET MESSAGE
011B CD4B01 TRP5:       CALL  [illegible]    ;INPUT KEYS
011E CD8202             CALL  [illegible]    ;[illegible] FOR [illegible]
0121 C31B01             JMP   TRP5        ;[illegible] AGAIN
0124 77         TRP6:   MOV   M,A         ;[illegible]
                        ;
                        ;SAVE REGISTERS
                        ;
0125 F1                 POP   PSW         ;[illegible] SAVPSW IN [illegible]
0126 E1                 POP   H           ;GET USER SP [illegible]
0127 23                 INX   H           ;USER SP
0128 23                 INX   H           ;USER SP
0129 22DE08             SHLD  SAVSP       ;SAVE SP IN RAM
012C 2B                 DCX   H           ;USER SP
012D 2B                 DCX   H           ;USER SP
012E F9                 SPHL              ;RESTORE SP
012F E1                 POP   H           ;GET RETURN ADRS [illegible] PROGRAM
0130 22DC08             SHLD  SAVPC       ;STORE IT IN RAM
0133 31E80B             LXI   SP,0BE8H    ;ADRS OF SAVA+1
0136 2AD30B             LHLD  TSAVH       ;RESTORE H,L
0139 F5                 PUSH  PSW         ;INTO SAVPSW
013A C5                 PUSH  B           ;INTO SAVB
013B D5                 PUSH  D           ;INTO SAVD
```

Table F-2. ROM Listing

```
Hex
Adrs Contents  Label   Instruction          Comments

013C E5                PUSH H          ;INTO SAVH
013D 20                RIM             ;GET IN
013E 32D800            STA  SAVIM      ;STORE IT IN RAM
0141 21DC00  TRP7:     LXI  SP,SAVPSW  ;POINT IT TO USER SP
0144 C1                POP  B          ;AND POP IT IN B&T
0145 21CE00            LXI  SP,INSP    ;RESTORE MONITOR SP
0148 C36402            JMP  FETH3
                       ;
                       ;KEY INPUT AND DECODE
                       ;
014B D5      KIND:     PUSH D
014C E5                PUSH H
014D CDE901  KIND1:    CALL DCD        ;UPDATE DISPLAY AND WAIT
0150 CD8501            CALL KPU        ;CHK FOR PUSHED KEY
0153 C24D01            JNZ  KIND1      ;IF KEY STILL PUSHED
0156 CDE901  KIND2:    CALL DCI        ;UPDATE DISP AND WAIT
0159 CD9501            CALL KPU        ;CHK FOR PUSHED KEY
015C CA5601            JZ   KIND2      ;IF KEY NOT PUSHED
015F 21E80B            LXI  H,LDKY     ;ADRS OF FIRST KEY FOR SCAN
0162 1EFF              MVI  D,0FFH     ;LOAD ROW COUNTER TO 0-1
0164 7E      KIND3:    MOV  A,M        ;GET ROW N KEY DATA
0165 FEF7              CPI  0F7H       ;IS IT THE HOME KEY FLAG
0167 CA8201            JZ   KIND5      ;YES JUMPS
016A 2F                CMA             ;INVERT KEY DATA
016B 0C                INR  L          ;NEXT ROW
016C 14                INR  D          ;NEXT ROW N FLAG
016D A7                ANA  A          ;TEST ROW N FOR 0
016E CA6401            JZ   KIND3      ;JUMP IF KEY NOT PUSHED
0171 FE04              CPI  4          ;SET IF D3=1
0173 D27701            JNC  KIND4      ;IF SO
0176 3D                DCR  A          ;ELSE SET A=D
0177 82      KIND4:    ADD  D          ;ADD 3X THE ROW NUMBER
0178 82                ADD  D          ;GET THE TABLE OFFSET
0179 82                ADD  D
017A 5F                MOV  E,A        ;STORE TABLE IN E
017B 1600              MVI  D,0        ;CLEAR MS BYTE OF DE
017D 21AF01            LXI  H,KTBL-1   ;ADRS OF KEY CODE TABLE
0180 19                DAD  D          ;ADD INDEX TO TABLE ADRS
0181 7E                MOV  A,M        ;PUT KEY CODE IN A
0182 E1      KIND5:    POP  H
0183 D1                POP  D
0184 C9                RET
                       ;
                       ;DETERMINES IF ANY KEY IS PUSHED
                       ;
0185 C5      KPU:      PUSH B
0186 CD9A01            CALL KPI        ;READ THE KEYBOARD
0189 0600              MVI  B,0        ;SET THE LOOP COUNTER
018B 21E80B            LXI  H,LDKY     ;ADDRESS OF UNPROCESSED KEY FLAG
018E 3EFF              MVI  A,0FFH     ;UNPUSHED KEY CODE
0190 BE      KPU1:     CMP  M          ;ANY KEY PUSHED (IF SO CODE CHANGED)
0191 20                INR  L          ;NEXT RAM KEY FLAG
0192 05                DCR  B          ;LOOP COUNTER
0193 C29001            JNZ  KPU1       ;IF ALL KEY ROWS NOT READ WITH A
0196 FEFF              CPI  0FFH       ;SET FLAG IF ALL FF'S (NOT TOUCHED)
0198 C1                POP  B
0199 C9                RET
                       ;
                       ;READS KEYS AND LOOPES THEM UNTIL ONLY
                       ;
019A 21E80B  KRD:      LXI  H,LDKY     ;ADRS OF UNDECODED KEY SCAN
019D 3EFF              MVI  A,0FFH     ;BLANK DISPLAY CODE
019F D338              OUT  DSP        ;CLEAR DISPLAY
01A1 3D                DCR  A          ;SET A TO 'FE' (1ST SCAN POINTER
01A2 37                STC             ;TO PRESET END OF SCAN LOOP FLAG
01A3 D328    KRD1:     OUT  SCAN       ;SCAN ONE KEY ROW
01A5 47                MOV  B,A        ;SAVE SCAN POINTER
01A6 DB18              IN   KEY        ;INPUT A KEY ROW
01A8 77                MOV  M,A        ;STORE IT IN RAM
01A9 7B                MOV  A,E        ;RESTORE SCAN POINTER
01AA 2C                INR  L          ;POINT TO NEXT RAM ADRS
01AB 1F                RAR             ;MOVE SCAN POINTER TO NEXT KEY ROW
01AC DAA301            JC   KRD1       ;IF LAST KEY ROW NOT SCANNED
01AF C9                RET
                       ;
                       ;KEY INPUT DECODE TABLE (RANGE FROM 1 TO F7
                       ;
01B0 8E      KTT:      DB   8EH        ;INRTR STEP   IF  CODE
01B1 85                DB   85H        ;FETCH PC     IF  CODE
01B2 00                DB   0          ;UNDEFINED    KE  CODE
01B3 84                DB   84H        ;RUN          KE  CODE
01B4 8A                DB   80H        ;FETCH REG    IF  CODE
01B5 82                DB   82H        ;FETCH ADRS   IF  CODE
01B6 0A                DB    0         ;    0        IF  CODE
01B7 83                DB   83H        ;STORE/INCR   IF  CODE
01B8 81                DB   81H        ;RFF          KE  CODE
01B9 01                DB    1         ;    1        KE  CODE
01BA 02                DB    2         ;    2        KE  CODE
01BB 03                DB    3         ;    3        KE  CODE
01BC 04                DB    4         ;    4        KE  CODE
01BD 05                DB    5         ;    5        KE  CODE
01BE 06                DB    6         ;    6        KE  CODE
01BF 07                DB    7         ;    7        KE  CODE
01C0 08                DB    8         ;    8        KE  CODE
01C1 09                DB    9         ;    9        KE  CODE
01C2 0A                DB   0AH        ;    A        KE  CODE
01C3 0B                DB   0BH        ;    B        KE  CODE
01C4 0C                DB   0CH        ;    C        KE  CODE
01C5 0D                DB   0DH        ;    D        KE  CODE
01C6 0E                DB   0EH        ;    E        KE  CODE
01C7 0F                DB   0FH        ;    F        KE  CODE
                       ;
                       ;SCAN DISPLAY SEGMENTS
                       ;
01C8 F5      SDS:      PUSH PSW
01C9 E5                PUSH H
01CA C5                PUSH B
01CB 21FF0B            LXI  H,IDSP5    ;ADRS OF DECODED DISP DIGIT 5
01CE 0620              MVI  B,20H      ;DISP DIGIT 5 SCAN POINTER

Hex
Adrs Contents  Label   Instruction          Comments

01D0 AF      SDS1:     XRA  A          ;CLEAR A
01D1 D320              OUT  SCAN       ;TURN DISP DIGIT OFF
01D3 7E                MOV  A,M        ;GET SEGMENT DATA
01D4 D338              OUT  DSP        ;STORE IT IN DISP LATCH
01D6 78                MOV  A,B        ;GET SCAN DIGIT POINTER
01D7 D328              OUT  SCAN       ;TURN ON DISP DIGIT
01D9 CD2904            CALL DELA       ;STRETCH DIGIT TIME
01DC 2D                DCR  L          ;ADRS OF NEXT DIGIT IN RAM
01DD 1F                RAR             ;NEXT SCAN DIGIT POINTER
01DE 47                MOV  B,A        ;SAVE IT IN B
01DF D2D001            JNC  SDS1       ;IF NOT LAST SCANED DIGIT (LSD)
01E2 2F                CMA             ;SET A=FF BLANK DISP CODE
01E3 D338              OUT  DSP        ;TURN OFF DISP DIGIT
01E5 C1                POP  B
01E6 E1                POP  H
01E7 F1                POP  PSW
01E8 C9                RET
                       ;
                       ;DISPLAY CHARACTER DECODER
                       ;
01E9 F5      DCD:      PUSH PSW
01EA C5                PUSH B
01EB D5                PUSH D
01EC E5                PUSH H
01ED 01FA0B            LXI  B,IDSP0    ;DECODED DIGIT BIN COUNTER
01F0 11F00B            LXI  D,LDSP0    ;UNDECODED DIGIT BCD ADRS
01F3 211802  DCD1:     LXI  H,TCC      ;DISPLAY CODE CONVERTER TABLE ADRS
01F6 1A                LDAX D          ;GET UNDECODED DATA
01F7 D5                PUSH D          ;SAVE ITS ADRS
01F8 5F                MOV  E,A        ;TABLE OFFSET VALUE
01F9 1600              MVI  D,0        ;CLEAR D
01FB 19                DAD  D          ;ADD OFFSET VALUE TO TABLE ADRS
01FC 7E                MOV  A,M        ;GET DECODED DATA FROM TABLE ADRS
01FD 02                STAX B          ;STORE IT IN DECODED RAM
01FE D1                POP  D          ;RESTORE UDSP0 ADRS
01FF 1C                INR  E          ;POINT TO NEXT UNSP ADRS
0200 0C                INR  C          ;POINT TO NEXT DISP ADRS
0201 C2F301            JNZ  DCD1       ;IF NOT LAST DIGIT
0204 21FA0B            LXI  H,IDSP0    ;DECODED DIGIT 0
0207 1A                LDAX D          ;UPDATE DATA MODIFY FLAG
0208 A7                ANA  A          ;CHECK FOR SET FLAG
0209 CA1002            JZ   DCI2       ;IF DATA NOT BEING MODIFIED
020C 7E                MOV  A,M        ;GET DDSP0 DATA
020D E67F              ANI  7FH        ;SET ITS DECIMAL POINT OFF BIT
020F 77                MOV  M,A        ;STORE IT IN DDSP0
0210 E1      DCD2:     POP  H          ;
0211 D1                POP  D          ;
0212 C1                POP  B          ;
0213 F1                POP  PSW        ;
0214 CDC801            CALL SDS        ;OPERATE DISPLAY
0217 C9                RET             ;
                       ;
                       ;DISPLAY CODE CONVERTER TABLE
                       ;
0218 C0      TCC:      DB   0C0H       ;0
0219 F9                DB   0F9H       ;1
021A A4                DB   0A4H       ;2
021B B0                DB   0B0H       ;3
021C 99                DB   099H       ;4
021D 92                DB   092H       ;5
021E 82                DB   082H       ;6
021F F8                DB   0F8H       ;7
0220 80                DB   080H       ;8
0221 90                DB   090H       ;9
0222 88                DB   088H       ;A
0223 83                DB   083H       ;B
0224 C6                DB   0C6H       ;C
0225 A1                DB   0A1H       ;D
0226 86                DB   086H       ;E
0227 8E                DB   08EH       ;F
0228 FF                DB   0FFH       ;BLANK
0229 89                DB   089H       ;H
022A C7                DB   0C7H       ;L
022B E3                DB   0E3H       ;U SMALL
022C 8C                DB   08CH       ;P
022D A3                DB   0A3H       ;O
022E C1                DB   0C1H       ;U LARGE
022F F7                DB   0F7H       ;-
0230 A7                DB   0A7H       ;C SMALL
0231 CF                DB   0CFH       ;I LEFT
0232 00                DB   00H        ;ALL SEGS
0233 AF                DB   0AFH       ;R SMALL
0234 BF                DB   0BFH       ;
                       ;
                       ;STDM STORES DISP MESSAGE AT DE ADRS IN UDSP RAM
                       ;
0235 0606    SDM:      MVI  B,6        ;LOOP COUNTER FOR 6 FLAT DIGITS
0237 1A      SDM1:     LDAX D          ;DISPLAY CHARACTER
0238 77                MOV  M,A        ;STORE IT IN UDSP5 - 0 RAM
0239 2C                INR  L          ;NEXT UDSP ADRS
023A 13                INX  D          ;NEXT MESSAGE TABLE ADRS
023B 05                DCR  B          ;LOOP COUNTER
023C C23702            JNZ  SDM1       ;IF NOT LAST DIGIT
023F C1                POP  B          ;
0240 C9                RET
                       ;
                       ;DISPLAY MESSAGE TABLES
                       ;
0241 14      DNT:      DB   14H        ;P
0242 16                DB   16H        ;U
0243 0E                DB   0EH        ;E
0244 0A                DB   0AH        ;A
0245 12                DB   12H        ;L
0246 13                DB   13H        ;U SMALL
0247 10      FETCH:    DB   10H        ;SP
0248 10                DB   10H        ;SP
0249 17                DB   17H        ;.
024A 17                DB   17H        ;.
024B 17                DB   17H        ;.
```

Table F-2. ROM Listing (Continued)

```
Hex
Adrs Contents  Label    Instruction        Comments

024C 17                 DB    17H          ;
024D 0A        MA:      DB    0AH          ;A
024E 10                 DB    10H          ;
024F 10                 DB    10H          ;
0250 10                 DB    10H          ;
0251 12        FLG:     DB    12H          ;L
0252 0F                 DB    0FH          ;F
0253 10                 DB    10H          ;
0254 10                 DB    10H          ;
0255 0B        MB:      DB    0BH          ;B
0256 10                 DB    10H          ;
0257 10                 DB    10H          ;
0258 10                 DB    10H          ;
0259 0C        MC:      DB    0CH          ;C
025A 10                 DB    10H          ;
025B 10                 DB    10H          ;
025C 10                 DB    10H          ;
025D 0D        MD:      DB    0DH          ;D
025E 10                 DB    10H          ;
025F 10                 DB    10H          ;
0260 10                 DB    10H          ;
0261 0E        ME:      DB    0EH          ;E
0262 10                 DB    10H          ;
0263 10                 DB    10H          ;
0264 10                 DB    10H          ;
0265 11        MH:      DB    11H          ;H
0266 10                 DB    10H          ;
0267 10                 DB    10H          ;
0268 10                 DB    10H          ;
0269 12        ML:      DB    12H          ;L
026A 10                 DB    10H          ;
026B 10                 DB    10H          ;
026C 10                 DB    10H          ;
026D 11        SPH:     DB    11H          ;H
026E 14                 DB    14H          ;P
026F 05                 DB    05H          ;S
0270 10                 DB    10H          ;
0271 12        SPL:     DB    12H          ;L
0272 14                 DB    14H          ;P
0273 05                 DB    05H          ;S
0274 10                 DB    10H          ;
0275 11        PCH:     DB    11H          ;H
0276 0C                 DB    0CH          ;C
0277 14                 DB    14H          ;P
0278 10                 DB    10H          ;
0279 12        PCL:     DB    12H          ;L
027A 0C                 DB    0CH          ;C
027B 14                 DB    14H          ;P
027C 10                 DB    10H          ;
027D 15        IM:      DB    15H          ;I
027E 10                 DB    10H          ;
027F 10                 DB    10H          ;
0280 10                 DB    10H          ;
0281 1A        ALL:     DB    1AH          ;A
0282 1A                 DB    1AH          ;A
0283 1A                 DB    1AH          ;
0284 1A                 DB    1AH          ;
0285 1A                 DB    1AH          ;
0286 1A                 DB    1AH          ;ALL
0287 10        ICC:     DB    10H          ;
0288 10                 DB    10H          ;
0289 10                 DB    10H          ;
028A 0C                 DB    0CH          ;C
028B 01                 DB    01H          ;1
028C 10                 DB    10H          ;
028D 1B        PRM:     DB    1BH          ;     CALL
028E 0E                 DB    0EH          ;E
028F 14                 DB    14H          ;P
0290 05                 DB    05H          ;S
0291 10        BLNKC:   DB    10H          ;     BLANK CHARACTER CODE
0292 10                 DB    10H          ;
0293 10                 DB    10H          ;
0294 10                 DB    10H          ;
0295 10                 DB    10H          ;
0296 10                 DB    10H          ;
                        ;
                        ;BLANK THE DISPLAY
                        ;
0297 119102 BLNK:       LXI   D,BLNKM      ;START OF BLANK [illegible]
029A DF                 RST   3            ;TO MESSAGE
029B CDE?03             CALL  DCS          ;[illegible] TO DISPLAY
029E C9                 RET
                        ;
                        ;CONTROL KEY [illegible] HANDLE
                        ;
029F FEF7      CHISS:   CPI   0F7H         ;HARDWARE SINGLE [illegible]
02A1 CAF702             JZ    HDSS         ;HARDWARE SINGLE [illegible]
02A4 FE86      CINSS:   CPI   86H          ;SOFTWARE SINGLE [illegible]
02A6 CAF103             JZ    INSS         ;SOFTWARE SINGLE STEP ROUTINE
02A9 FE84      CRUN:    CPI   84H          ;RUN PUSHED?
02AB CA?203             JZ    RUN          ;RUN ROUTINE
02AE FE83      CSTRM:   CPI   83H          ;STORE MEMORY PUSHED?
02B0 CAAF03             JZ    STRM         ;STORE MEMORY ROUTINE
02B3 FE81      CDCRM:   CPI   81H          ;DECREMENT MEMORY PUSHED?
02B5 CAAA03             JZ    DCRM         ;DECREMENT MEMORY ROUTINE
02B8 FE82      CFETA:   CPI   82H          ;FETCH ADRS PUSHED?
02BA CA2903             JZ    FETA         ;FETCH ADRS ROUTINE
02BD FE85      CFETP:   CPI   85H          ;FETCH PROGRAM COUNTER PUSHED?
02BF CA4101             JZ    TRFT         ;FETCH PROGRAM COUNTER ROUTINE
02C2 FE80      CFETR:   CPI   80H          ;FETCH REGISTER PUSHED?
02C4 CAD502             JZ    FETR         ;FETCH REGISTER ROUTINE
02C7 C9                 RET                ;IF NO LEGAL [illegible] WAS PUSHED
02C8 FE83      CSTRR:   CPI   83H          ;STORE REGISTER PUSHED?
02CA CA1304             JZ    STRR         ;STORE REGISTER ROUTINE
02CD FE81      CDCRR:   CPI   81H          ;DECREMENT REGISTER PUSHED?
02CF CA1D03             JZ    DCRR         ;DECREMENT REGISTER ROUTINE
02D2 C3B802             JMP   CFETA        ;
                        ;
                        ;FETCH REGISTER CODE
```

```
Hex
Adrs Contents  Label    Instruction        Comments

                        ;
02D5 E1        FETR:    POP   H            ;[illegible] STACK CELL
02D6 214D02             LXI   H,MA         ;[illegible] MESSAGE ADR
02D9 01C70B             LXI   B,AVM        ;[illegible] USER [illegible]
02DC 2C        FETR1:   INX   H            ;[illegible]
02DD 2C                 INX   H            ;
02DE 22F80B    FETR2:   SHLD  ??S          ;STORE [illegible]
02E1 EB                 XCHG               ;[illegible] MESSAGE [illegible]
02E2 DF                 RST   3            ;[illegible] MESSAGE [illegible]
02E3 0A                 LDAX  B            ;[illegible] REG CONTENTS
02E4 5F                 MOV   E,A          ;[illegible] IT [illegible]
02E5 21F10B             LXI   H,UDSP1      ;[illegible] WHERE [illegible]
02E8 CD3C03             CALL  FETA7        ;[illegible] REG BY [illegible]
02EB 3EDB               MVI   A,0DBH       ;[illegible]
02ED E9                 CMP   ?            ;[illegible] REG EXAMINE [illegible]
02EE CA1003             JZ    FETR6        ;[illegible]
02F1 CD4B01    FETR3:   CALL  KIND         ;[illegible] KEYS
02F4 CDC802             CALL  CSTRR        ;[illegible] FOR CONTROL
02F7 D2F102             JNC   FETR3        ;[illegible] A HEX [illegible]
02FA 5F                 MOV   E,A          ;[illegible] HEX [illegible]
02FB 2C                 INR   L            ;POINT TO UDSP1
02FC 3600               MVI   M,0          ;CLEAR IT TO [illegible]
02FE 2D        FETR4:   DCR   L            ;POINT BACK [illegible]
02FF 73                 MOV   M,E          ;STORE NEW [illegible]
0300 CD1903    FETR5:   CALL  DPS          ;[illegible] SET UP AND [illegible]
0303 CDC802             CALL  CSTRR        ;[illegible] FOR CONTROL
0306 D20003             JNC   FETR5        ;[illegible] A HEX [illegible]
0309 2C                 INR   L            ;POINT TO UDSP
030A 5?                 MOV   D,E          ;[illegible]
030B 5F                 MOV   E,A          ;[illegible] NEW [illegible]
030C 72                 MOV   M,D          ;STORE OLD [illegible]
030D C3FE02             JMP   FETR4        ;CONTINUE
0310 CD4B01    FETR6:   CALL  KIND         ;[illegible] KEYS AND [illegible]
0313 CDC802             CALL  CSTRR        ;LOOK FOR CONTROL [illegible]
0316 C31003             JMP   FETR6        ;KEEP LOOKING
                        ;
                        ;DECIMAL POINT SET
                        ;
0319 3E01      DPS:     MVI   A,1          ;[illegible] SET [illegible]
031B 32F60B             STA   UDSPC        ;[illegible] DATA MUST [illegible]
031E CD4B01             CALL  KIND         ;[illegible] ANOTHER [illegible]
0321 F5                 PUSH  PSW          ;[illegible] KEY CODE
0322 AF                 XRA   A            ;[illegible] A
0323 32F60B             STA   UDSPC        ;CLEAR DATA [illegible]
0326 F1                 POP   PSW          ;[illegible]
0327 C9                 RET
                        ;
                        ;FETCH [illegible] ROUTINE
                        ;
0328 D1        FETA:    POP   D            ;[illegible] STACK CELL
0329 114703    FETAP:   LXI   D,FETCH      ;[illegible] MESSAGE [illegible]
032C DF                 RST   3            ;[illegible] MESSAGE [illegible]
032D 0E04               MVI   C,4          ;[illegible] DIGIT [illegible]
032F CD4B01    FETA1:   CALL  KIND         ;[illegible] KEYS AND [illegible]
0332 CDB802             CALL  CFETA        ;[illegible] FOR CONTROL
0335 D22F03             JNC   FETA1        ;[illegible] A HEX [illegible]
0338 21F60B             LXI   H,UDSP6      ;[illegible] OF DISPLAY [illegible]
033B 47                 MOV   B,A          ;[illegible] KEY [illegible]
033C FE01               CPI   1            ;[illegible] PUSHED
033E C24A03             JNZ   NOT1         ;IF NOT
0341 3E04               MVI   A,4          ;[illegible] POSIT[illegible]
0343 B9                 CMP   C            ;[illegible] POSITION
0344 C24A03             JNZ   NOT1         ;IF [illegible] NOT [illegible]
0347 C3290?             JMP   FETAP        ;[illegible] FOR [illegible]
034A 78        NOT1:    MOV   A,B          ;[illegible] NOW [illegible]
034B 0604               MVI   B,4          ;[illegible] POSITION [illegible]
034D 2D        FETA2:   DCR   L            ;POINT TO DISP [illegible]
034E 2D                 DCR   L            ;POINT TO DISP [illegible]
034F 56                 MOV   D,M          ;[illegible] THIS CHARACTER [illegible]
0350 2C                 INR   L            ;POINT TO DISP [illegible]
0351 72                 MOV   M,D          ;STORE THE DIGIT [illegible]
0352 05                 DCR   B            ;[illegible] POSITION COUNTER
0353 C24D03             JNZ   FETA2        ;[illegible] NOT DONE [illegible]
0356 77                 MOV   M,A          ;[illegible] CODE TO [illegible]
0357 0D                 DCR   C            ;[illegible] COUNTER
0358 C22F03             JNZ   FETA1        ;[illegible] DONE
035B CD8303             CALL  FETA6        ;[illegible] ADRS [illegible]
035E 4F                 MOV   C,A          ;STORE MERGED [illegible]
035F 2C                 INR   L            ;POINT TO MS [illegible]
0360 CD9302             CALL  FETA6        ;MERGE IT [illegible]
0363 47                 MOV   B,A          ;STORE IT IN [illegible]
0364 21F50B    FETA3:   LXI   H,UDSP5      ;[illegible] OF DISP [illegible]
0367 58                 MOV   E,B          ;[illegible] ADRS BYTE [illegible]
0368 CD3C02             CALL  FETA7        ;[illegible] AND [illegible]
036B 2D                 DCR   L            ;POINT TO UDSP?
036C 59                 MOV   E,C          ;[illegible] ADRS [illegible]
036D CD3C01             CALL  FETA7        ;SEPARATE AND [illegible]
0370 2D                 DCR   L            ;POINT TO UDSP?
0371 0A                 LDAX  B            ;GET DATA AT [illegible]
0372 5F                 MOV   E,A          ;PUT IT IN E
0373 CD3C03             CALL  FETA7        ;SEPARATE IT AND [illegible] DATA
                        ;
                        ;MODIFY THE DATA
                        ;
0376 CD4B01             CALL  KIND         ;GET A KEY
0379 CD9F02             CALL  CHISS        ;LOOK FOR ANY [illegible]
037C 5F                 MOV   E,A          ;STORE HEX KEY [illegible]
037D 77                 MOV   M,A          ;[illegible] UDSP0
037E 2C                 INR   L            ;POINT TO UDSP1
037F 3600               MVI   M,0          ;CLEAR IT
0381 2D        FETA4:   DCR   L            ;POINT TO UDSP0
0382 73                 MOV   M,E          ;[illegible] THE LATEST [illegible]
0383 CD1903    FETA5:   CALL  DPS          ;[illegible] SET DP AND [illegible]
0386 CDAE02             CALL  CSTRM        ;LOOK FOR CONTROL [illegible]
0389 D28303             JNC   FETA5        ;IF NOT A VALUE [illegible]
038C 2C                 INR   L            ;POINT TO UDSP1
038D 53                 MOV   D,E          ;[illegible] KEY ENTRY
038E 5F                 MOV   E,A          ;NEW KEY ENTRY
038F 72                 MOV   M,D          ;[illegible] ENTRY SHIFTS [illegible]
0390 C38103             JMP   FETA4        ;UPDATE NEW KEY [illegible]
```

Table F-2. ROM Listing (Continued)

```
Hex
Adrs Contents  Label   Instruction          Comments

                       ;
                       ;MERGES 2 HEX NUMBERS INTO A REG
                       ;
0392 5E      FETH4:    MOV  E,M             ;LS HEX CHAR
0393 23                INX  H               ;POINT TO MS HEX CHAR
0394 7E                MOV  A,M             ;MS HEX CHAR
0395 07                RLC                  ;MOVE IT TO 4 MS BITS IN A
0396 07                RLC                  ;AND CLEAR  4 LS BITS IN A
0397 07                RLC                  ;
0398 07                RLC                  ;
0399 83                ORA  E               ;MERGE MS AND LS HEX CHARS IN A
039A C9                RET
                       ;
                       ;SEPARATES 2 HEX CHARACTERS IN A REG STORES IN M
                       ;
039B 7A      FETH3:    MOV  A,D             ;PUT MERGED HEX CHARS INTO A
039C 0F                RRC                  ;MOVE MS HEX CHAR TO 4 LS BITS O
039D 0F                RRC                  ;
039E 0F                RRC                  ;
039F 0F                RRC                  ;
03A0 0F                RRC                  ;
03A1 E60F              MVI  D,0FH           ;MASK TO CLEAR 4 MS BITS OF A
03A3 A2                ANA  D               ;DO IT
03A4 77                MOV  M,A             ;STORE MS HEX CHAR
03A5 2D                DCR  L               ;LS CHAR DESTINATION ADDR
03A6 7B                MOV  A,E             ;GET MERGED HEX CHARS
03A7 A2                ANA  D               ;CLEAR MS HEX CHAR
03A8 77                MOV  M,A             ;STORE LS HEX CHAR
03A9 C9                RET
                       ;
                       ;DECREMENTS MEMORY ADDRS
                       ;
03AA 0B      DCRM:     DCX  B               ;POINT TO NEXT LOWER ADDR
03AB E1                POP  H               ;UNDO STACK CALL
03AC C36403            JMP  FETH3           ;TO FETCH NEW MEMORY DATA
                       ;
                       ;STORES DATA IN MEM AND INCREMENTS ADDRS
                       ;
03AF D1      STRM:     POP  D               ;UNDO STACK CALL
03B0 CD9303            CALL FETH4           ;MERGE 2 HEX DATA VALUES INTO A
03B3 5A                MOV  E,D             ;SAVE IN E
03B4 02                STAX B               ;STORE DATA BYTE IN RAM ADDRS IN B
03B5 0A                LDAX B               ;READ IT BACK
03B6 BB                CMP  E               ;IS IT THE SAME BIT IF STORED
03B7 0F                INX  B               ;POINT TO NEXT RAM ADDRS
03B8 CA6403            JZ   FETH3           ;FETCH NEW ADDRS IF LAST STORE OK
03BB 0B                DCX  B               ;ELSE POINT BACK TO LAST RAM ADDRS
03BC C5                PUSH B               ;SAVE THIS ADDRS
03BD D7                RST  2               ;BEEP A FAIL TONE ERROR
03BE F1                POP  B               ;RESTORE RAM ADDRS
03BF C36403            JMP  FETH3           ;AND FETCH IT AGAIN
                       ;
                       ;RUNS THE PROGRAM STARTING AT ADDRS IN DISPLAY
                       ;
03C2 213201  RUN:      LXI  H,0132H         ;INSTR CODES FOR JMP 0100H
03C5 22050B  RUN1:     SHLD RAM14           ;JUMP LINK TO MON
03C8 21106C            LXI  H,6C31H         ;INSTR CODES FOR STA AND JMP TO
03CB 22D70B            SHLD RAM12           ;JUMP LINK IN RAM
03CE 31DB0B            LXI  SP,SAV14        ;USER PROG START ADDRS LINK
03D1 C5                PUSH B               ;STORE USER START ADDRS IN RAM'S TOP
03D2 21DF0B            LXI  H,SAV5H         ;RAM ADDRS OF USER'S E
03D5 3600              MVI  M,00H           ;N=BYTE USER SP
03D7 2B                DCX  H               ;RAM ADDRS OF USER'S D
03D8 7E                MOV  A,M             ;L=BYTE USER SP
03D9 FE40              CPI  40H             ;>= 40
03DB D2E003            JNC  RUN2            ;IF AVAIL STACK, JMP OFF
03DE 3680              MVI  M,80H           ;IF NOT, RESET INTERP
03E0 31E20B  RUN2:     LXI  SP,SAVE         ;PREPARE TO RESTORE USER REGS
03E3 D1                POP  D               ;RESTORE D,E
03E4 C1                POP  B               ;RESTORE B,C
03E5 F1                POP  PSW             ;RESTORE PSW,A
03E6 31DE0B            LXI  SP,SAVSL        ;RAM ADDRS OF USER SP
03E9 E1                POP  H               ;PUT SPH,L IN H,L
03EA F9                SPHL                 ;TRANSFER SP VAL TO MPU SP
03EB 2AE00B            LHLD SAVL            ;RESTORE H,L
03EE C2D50B            JMP  RAM13           ;JUMP TO USER PROGRAM
                       ;
                       ;INSTRUCTION SINGLE STEP AND RETURN TO MON IF
                       ;
03F1 213206  INSS:     LXI  H,0632H         ;INSTR CODES FOR MOV M,A JMP
03F4 C30C03            JMP  RUN1            ;SET LINKS,RESTORE REGS,USER PROG
                       ;
                       ;HARDWARE SINGLE STEP ONE LINE OF CODE
                       ;
03F7 213203  HDSS:     LXI  H,F032H         ;INSTR CODES FOR STA 08XX, RST 6
03FA C30C03            JMP  RUN1            ;SET LINKS,RESTORE REGS,USER PROG
                       ;
                       ;DECREMENTS REGISTER DISPLAY
                       ;
03FD D1      DCRR:     POP  D               ;UNDO STACK CALL
03FE 03                INX  B               ;POINT TO LAST SAVED REG RAM ADDRS
03FF 2AF80B            LHLD RMP             ;REGISTER MESSAGE POINTER
0402 2B                DCX  H               ;SUBTRACT 2 FROM IT
0403 2B                DCX  H               ;
0404 3EE8              MVI  A,0E8H          ;LS BYTE OF SAVED REGS IN RAM
0406 B9                CMP  C               ;SEE IF IT'S IN REG
0407 C2DC02            JNZ  FETR1           ;FETCH NEW NON-PC REG+DISP IF NOT
040A 010B0B            LXI  B,SAV1M         ;ADRS OF IN VALUE IN RAM
040D 21D702            LXI  H,1M            ;ADRS OF IM DISP MESSAGE
0410 C3DC02            JMP  FETR1           ;FETCH IM REG + DISP IF IT IS IM
                       ;
                       ;STORES REGISTER DATA AND INCREMENTS
                       ;
0413 D1      STRR:     POP  D               ;UNDO STACK CALL
0414 CD9303            CALL FETH4           ;MERGE 2 HEX DATA VALUES IN A
0417 02                STAX B               ;STORE DATA BYTE IN SAVED REG ADRS
0418 0B                DCX  B               ;POINT TO NEXT SAVED REG ADRS
0419 2AF80B            LHLD RMP             ;REGISTER MESSAGE POINTER
041C 23                INX  H               ;POINT TO NEXT REG MESSAGE
041D 23                INX  H               ;

Hex
Adrs Contents  Label   Instruction          Comments

041E 23                INX  H               ;
041F 23                INX  H               ;
0420 3EDA              MVI  A,0DAH          ;LS BYTE OF SAVED REGS ADRS IN RAM
0422 B9                CMP  C               ;SEE IF IT'S H REG
0423 CAD502            JZ   FETR            ;FETCH A REG + DISP IF IT IS
0426 C3DE02            JMP  FETR2           ;FETCH NEXT REG + DISP IF NOT
                       ;
                       ;DELAYS APPROX 1MS
                       ;
0429 C5      DELA:     PUSH B
042A 010100            LXI  B,0001H         ;FIXED 1MS VALUE
042D C33104            JMP  DEL1            ;START TIMING LOOP
                       ;
                       ;DELAYS APPROX 1MS TIMES VALUE IN REG
                       ;
0430 C5      DELB:     PUSH B
0431 F5      DEL1:     PUSH PSW
0432 AF                XRA  A               ;CLEAR A
0433 B5                PUSH B               ;
0434 1E8C      DEL2:   MVI  D,TIME          ;INIT SMALL LOOP TIME CONSTANT
0436 15        DEL3:   DCR  D               ;INNER LOOP
0437 C23604            JNZ  DEL3            ;IF NOT 1MS WORTH OF COUNTS
043A 0B                DCX  B               ;LARGE LOOP COUNTER
043B B8                CMP  B               ;B= 0?
043C C23404            JNZ  DEL2            ;IF NOT, LOOP FOR 1 MORE MS
043F B9                CMP  C               ;C= 0?
0440 C23404            JNZ  DEL2            ;IF NOT, LOOP
0443 D1                POP  D               ;WHEN DONE
0444 F1                POP  PSW
0445 C1                POP  B
0446 C9                RET
                       ;
                       ;BEEP PRODUCES A FIXED TONE FREQ + DURATION
                       ;
0447 2EFF    BEEP2:    MVI  L,0FFH          ;DURATION MULTIPLIER
0449 2600              MVI  H,0             ;SPEAKER FLAG
044B 48                MOV  C,B             ;SET COUNT REG 1
044C 5A                MOV  E,D             ;SET COUNT REG 1
044D E5                PUSH H               ;DUR, DONE FLAG
044E 0F      BEEP3:    DCR  C               ;OFF FREQ
044F C25004            JNZ  BEEP3           ;
0452 48                MOV  C,B             ;RESTORE FREQ
0453 7C                MOV  A,H             ;CHG SPKR FLAG
0454 2F                CMA                  ;
0455 67                MOV  H,A             ;
0456 57                MOV  D,A             ;
0457 C26004            JNZ  BEEP4           ;TEST SPKR FLAG
045A 3E08              MVI  A,08H           ;TURN SPKR OFF
045C 30                SIM                  ;
045D C36404            JMP  BEEP5           ;
0460 3E40    BEEP4:    MVI  A,40H           ;TURN SPKR ON
0462 30                SIM                  ;
0463 6E                MOV  L,M             ;TIME DELAY IN L
0464 F1      BEEP5:    POP  PSW             ;GET DONE FLAG
0465 F5                PUSH PSW             ;
0466 B7                ORA  A               ;
0467 CA3F04            JZ   BEEP6           ;CONTINUE IF NOT DONE
046A 7A                MOV  A,D             ;RETURN IF SPKR OFF
046B B7                ORA  A               ;
046C CA7E04            JZ   BEEP7           ;
046F 1D      BEEP6:    DCR  E               ;DURATION CNTR
0470 C23804            JNZ  BEEP9           ;
0473 5A                MOV  E,D             ;RELOAD DURATION
0474 2D                DCR  L               ;TIME COUNT
0475 C24E04            JNZ  BEEP8           ;
0478 F1                POP  PSW             ;GET DONE FLAG
0479 2F                CMA                  ;
047A F5                PUSH PSW             ;SET DONE FLAG
047B C34E04            JMP  BEEP8           ;
047E F1      BEEP7:    POP  PSW             ;
047F C9                RET                  ;
0480 E3      BEEP8:    XTHL                 ;DELAY 91 CYCLES
0481 E3                XTHL                 ;
0482 E3                XTHL                 ;
0483 E3                XTHL                 ;
0484 BE                CMP  M               ;
0485 C32F04            JMP  BEEP6           ;
0488 00      BEEP9:    NOP                  ;DELAY 14 CYCLES
0489 00                NOP                  ;
048A C34E04            JMP  BEEP8           ;
                       ;
                       ;SIGNATURE ANALYSIS TEST LOOP
                       ;
048D F3      SATL1:    DI                   ;TURN OFF INTERRUPTS
048E DB80              IN   80H             ;PULSE A15 READY LINE = STOP LINE
0490 D380              OUT  80H             ;PULSE A15 READY LINE = START LINE
0492 31E00B            LXI  SP,0BE0H        ;SET TO MONITOR RAM
                       ;
                       ;RAM PROTECT TEST
                       ;
0495 D311              OUT  11H             ;SET RAM PROTECT
0497 3E110B            STA  0B11H           ;WRITE TO UNPROTECTED RAM
049A 3E1109            STA  0911H           ;WRITE TO PROTECTED RAM
049D D310              OUT  10H             ;UNPROTECT RAM
                       ;
                       ;OUTPUT PORT TEST
                       ;
049F AF                XRA  A               ;CLEAR A
04A0 37                STC                  ;SET CARRY BIT = 1 FOR S LOOP
04A1 17      SATL2:    RAL                  ;MOVE 1 BIT TO LSB
04A2 D330              OUT  LOL1            ;OUTPUT PORT LSI
04A4 D2A104            JNC  SATL2           ;IF NOT DONE
                       ;
                       ;DISPLAY LATCH TEST
                       ;
04A7 17      SATL3:    RAL                  ;MOVE 1 BIT TO LSB
04A8 D338              OUT  DSP             ;OUTPUT DISPLAY SEGMENTS
04AA D2A704            JNC  SATL3           ;IF NOT DONE
                       ;
```

Table F-2. ROM Listing (Continued)

```
Hex
Adrs Contents Label     Instruction          Comments

                        ;SCAN LATCH TEST
                        ;
04AD [illegible]        [illegible]          [illegible]
04AE [illegible]        CMC                  [illegible] CARRY
04AF 1[illegible] SATL[illegible] [illegible]  [illegible] BIT [illegible]
04B0 D[illegible]       OUT  SCAN            [illegible]OUT SCAN [illegible]
04B2 C[illegible]04     CALL DEL[illegible]  [illegible]TCH EACH [illegible]
04B5 DAAF04             JC   SATL[illegible] [illegible] BIT DONE
                        ;
                        ;[illegible] TEST
                        ;
04B[illegible] [illegible]  MVI  A,4[illegible]H  [illegible] ON [illegible]
04BA [illegible]        SIM                  [illegible]
04BB [illegible]        MVI  A,C0[illegible]H [illegible] OFF [illegible]
04BD [illegible]        SIM                  [illegible]
                        ;
                        ;[illegible] INPUT TEST
                        ;
04BE [illegible]        [illegible]          [illegible]
04BF D[illegible]       OUT  [illegible]     [illegible] LINES [illegible]
04C1 DB[illegible]      IN   [illegible]     [illegible] TO [illegible]
                        ;
                        ;INPUT PORT TEST
                        ;
04C[illegible] DB[illegible]  [illegible]    [illegible] THE [illegible]
                        ;
                        ;INTERRUPT [illegible]
                        ;
04C[illegible] [illegible]  [illegible] [illegible]  [illegible] BEEP [illegible]
04C[illegible] [illegible]  [illegible] [illegible]  [illegible]
04CB [illegible]        MVI  A,C[illegible]H [illegible] CODE
04CD [illegible]        STA  [illegible]     [illegible]
04D0 [illegible]        MVI  A,1[illegible]H [illegible] MASK [illegible]
04D2 [illegible]        SIM                  [illegible]
04D3 FB                 EI                   [illegible] INTERRUPT [illegible]
04D4 C[illegible]04     JMP  SATL[illegible] [illegible] BACK [illegible]

                        ;[illegible] DATA [illegible] LEDS
                        ;
04D[illegible] [illegible] ECHO: LDA [illegible]  [illegible] INPUT [illegible]
04DA [illegible]        STA  [illegible]     [illegible] OUTPUT [illegible]
04DD [illegible]        JMP  ECHO            [illegible]
                        ;
                        ;AND [illegible]
                        ;
04E0 [illegible] ANLG[illegible] LDA [illegible]  [illegible] INPUT [illegible]
04E3 [illegible]        CPI  [illegible]     [illegible]
04E5 [illegible]04      JZ   [illegible]     [illegible] ALL [illegible]
04E8 [illegible]        MVI  A,[illegible]   [illegible]
04EA [illegible]        STA  [illegible]     [illegible]
04ED [illegible]04      JMP  ANLGT           
04F0 [illegible] ON:    MVI  A,[illegible]H  [illegible] LED [illegible]
04F2 [illegible]        STA  [illegible]     
04F5 [illegible]04      JMP  ANLGT
                        ;
                        ;[illegible] BUFFER [illegible]
                        ;
04F8 [illegible] CONV:  CALL [illegible]     [illegible] THE [illegible]
04FB [illegible]        [illegible] M        [illegible] A
04FC [illegible]        STA  [illegible]     [illegible]
04FF [illegible] LOOP:  CALL [illegible]     [illegible]
0502 FE[illegible]      CPI  [illegible]     [illegible] FOR [illegible]
0504 [illegible]04      JNZ  LOOP
0507 [illegible]        LXI  H,[illegible]   [illegible]
050A [illegible]        INR  M
050B [illegible]        MOV  A,M             [illegible] FOR [illegible]
050C [illegible]        CPI  10              
050E [illegible]05      JZ   MOVE
0511 [illegible]04      JMP  LOOP
0514 [illegible] MOVE:  MVI  A,[illegible]   
0516 [illegible]        MVI  [illegible]     [illegible] FREQ [illegible]
051[illegible] [illegible] MLP: STA [illegible]  [illegible] PATH [illegible]
051B [illegible]        LXI  B,[illegible]   [illegible] DELAY [illegible]
051E D5                 PUSH D
051F [illegible]04      CALL DELB
052[illegible] [illegible]  POP  D
052[illegible] [illegible]05  CALL [illegible]  [illegible] BEEP
052[illegible] [illegible]  STC
052[illegible] [illegible]  RAR              [illegible] PATTERN
052[illegible] [illegible]  [illegible]
052[illegible] [illegible]  STA  [illegible]  [illegible]
052[illegible] [illegible]04  JMP  CONV
053[illegible] [illegible] TONE: MOV [illegible]  [illegible] H
053[illegible] D[illegible]  PUSH D
053[illegible] 4[illegible]  MOV  B,[illegible]
053[illegible] [illegible]  CALL BEEP        [illegible] BEEP
053[illegible] D[illegible]  POP  D
053[illegible] [illegible]  MOV  A,[illegible]  [illegible] FREQ [illegible]
053[illegible] [illegible]  [illegible] INCR
053[illegible] [illegible]  MOV  [illegible]
053[illegible] [illegible]  MOV  A,[illegible]  [illegible] A
05[illegible] [illegible]   RET
08F0        NUM         EQU  08F0H           [illegible] BUFFER [illegible]
0010        INCR        EQU  10H             [illegible] INCREMENT
00F0        MIN         EQU  0F0H            [illegible] FREQUENCY [illegible]
0050        DTIME       EQU  50              [illegible] BETWEEN [illegible]
                        ;
                        ;[illegible] TEMPERA[illegible] [illegible]PROCESS[illegible]
                        ;GENERATES [illegible] RANDOM [illegible]
                        ;
053E [illegible] NTM:   LXI  H,[illegible]H  [illegible]
054[illegible] [illegible] NTM1: MOV A,M     [illegible] CONTENTS
054[illegible] E[illegible]  ANI  7EH        [illegible] BITS
0544 4[illegible]       MOV  B,A             [illegible] BEEP [illegible]
0545 E5                 PUSH H               [illegible] ADRS [illegible]
0546 [illegible]        CALL BEEP[illegible] [illegible] BEEP
0549 E1                 POP  H               [illegible] ADRS [illegible]
054A [illegible]        LXI  B,[illegible]H  [illegible] DELAY [illegible]
054D C[illegible]04     CALL DELB            ;PAUSE
0550 2[illegible]       INX  H               ;[illegible] ADRS OF ROM
0551 3E[illegible]      MVI  A,[illegible]H  ;[illegible] OF LOOP
0553 BC                 CMP  H               ;[illegible] ADRS
0554 [illegible]05      JC   NTM             ;[illegible] ADRS
0557 C[illegible]05     JMP  NTM1            ;[illegible] ADR[illegible]
                        ;
                        ;[illegible]
                        ;
055A CD[illegible] SOPL: CALL [illegible]    ;[illegible]
055D AF                 XRA  A               ;[illegible] A AND [illegible]
055E [illegible]01      MVI  [illegible]     ;[illegible]
0560 1[illegible] SOPL[illegible]: RAL       ;[illegible]
0561 5[illegible]       MOV  D,[illegible]   ;[illegible]
0562 7[illegible]       [illegible] A,[illegible]  ;[illegible]
0563 1[illegible]       RAR                  ;[illegible] BITA [illegible]
0564 4[illegible]       MOV  [illegible]     ;[illegible]
0565 [illegible]        [illegible] 2        ;[illegible]
0566 E[illegible]01     ANI  1               ;[illegible]
0568 [illegible]        ORA  [illegible]     ;[illegible]
0569 4[illegible]       MOV  [illegible]     ;[illegible] NEW [illegible]
056A 2[illegible]       LXI  [illegible]     ;[illegible]
056D F5                 PUSH PSW             ;[illegible]
056E 0F                 RRC                  ;[illegible]
056F 0F                 RRC                  ;
0570 C[illegible]05     CALL [illegible]     ;[illegible]
0573 2[illegible]       INX  [illegible]     ;[illegible]
0574 7[illegible]       MOV  [illegible]     ;[illegible]
0575 0[illegible]       RLC                  ;[illegible]
0576 C[illegible]05     CALL [illegible]     ;[illegible]
0579 [illegible]        MVI  [illegible]     ;[illegible] TIMES [illegible]
057B C[illegible] SORL[illegible]: CALL [illegible]  ;[illegible]
057E [illegible]        DCR  [illegible]     ;[illegible]
057F C[illegible]05     JNZ  [illegible]     ;[illegible]
0582 F[illegible]       POP  PSW             ;[illegible]
0583 C[illegible]       [illegible]          ;[illegible]
0586 4F   [illegible]URL[illegible]: MOV [illegible]  ;[illegible]
0587 [illegible]        CPI  [illegible]     ;[illegible]
0589 [illegible]        [illegible]          ;[illegible]
058A 7[illegible]       MOV  [illegible]     ;[illegible]
058B [illegible]        [illegible]          ;[illegible]
058C [illegible]        RLC                  ;[illegible]
058D E[illegible]01     ANI  [illegible]     ;[illegible]
058F [illegible]        [illegible]          ;[illegible]
0590 [illegible]        MOV  [illegible]     ;[illegible]
0591 0[illegible]       RRC                  ;[illegible]
0592 E[illegible]       [illegible]          ;[illegible]
0594 B[illegible]       ORA  [illegible]     ;[illegible]
0595 F[illegible]       CPI  [illegible]     ;[illegible]
0597 [illegible]        [illegible]          ;[illegible]
0598 [illegible]        RET
                        ;
                        ;[illegible] GENERATES [illegible]
                        ;
0599 [illegible] URGAN: LXI  [illegible]     ;[illegible]
059C [illegible] URGAN[illegible]: [illegible]  ;[illegible]
059E C[illegible]05     CALL [illegible]     ;[illegible]
05A1 [illegible]        [illegible]          ;[illegible]
05A2 C[illegible]05     [illegible]          
05A5 C[illegible]05     [illegible]          ;[illegible]
05A8 C[illegible]05     CALL [illegible]     ;[illegible]
05AB [illegible]05      JMP  [illegible]     ;[illegible]
                        ;
                        ;[illegible]
                        ;
05AE [illegible] [illegible]: [illegible]    ;[illegible]
05AF 0[illegible]       [illegible]
05B0 0[illegible]       [illegible]
05B1 [illegible]05      [illegible]
05B4 C[illegible]       RET
                        ;
                        ;[illegible] TO [illegible]
                        ;
05B5 7[illegible] SPKR: MOV  [illegible]     ;[illegible]
05B6 EE[illegible]      XRI  [illegible]     ;[illegible] BIT
05B8 5[illegible]       MOV  [illegible]     ;[illegible]
05B9 [illegible]        SIM                  ;[illegible]
05BA C[illegible]       RET
                        ;
                        ;[illegible] DELAY [illegible]
                        ;
05BB [illegible] CODE:  [illegible]          ;[illegible]
05BD 2[illegible]       [illegible]          ;[illegible]
05C0 7E   READ:         MOV  A,M             ;[illegible]
05C1 2[illegible]       [illegible]          
05C2 B[illegible]       ORA  [illegible]     ;[illegible]
05C3 CA[illegible]05    JZ   [illegible]     ;[illegible]
05C6 FE[illegible]      CPI  [illegible]     ;[illegible]
05C8 C[illegible]05     JNZ  [illegible]     ;[illegible]
05CB 3D                 DCR  A
05CC 4F   SHIFT:        MOV  C,A             ;[illegible]
05CD 7[illegible]       MOV  A,B             ;[illegible]
05CE 07                 RLC
05CF 07                 RLC
05D0 E[illegible]       [illegible]          ;[illegible]
05D1 [illegible]05      [illegible]          ;[illegible]
05D4 [illegible]        [illegible] L
05D5 [illegible]        MOV  [illegible]
05D6 7E                 MOV  [illegible]     ;[illegible]
05D7 C9                 RET
05D8 0[illegible] NOKEY: DCR [illegible]     ;[illegible]
05D9 [illegible]        [illegible] H
05DA 7[illegible]       MOV  [illegible]     ;[illegible]
05DB FE[illegible]      CPI  [illegible]
05DD C[illegible]05     JNZ  [illegible]     ;[illegible] READ [illegible]
05E0 AF                 XRA  A               ;[illegible]
05E1 C9                 RET
                        ;
```

Table F-2. *ROM Listing (Continued)*

```
Hex
Adrs Contents Label    Instruction          Comments

                       ;LOOK-UP TABLE FOR ORGNN OF  VALUES
                       ;
05E2 E6      TABLE:    DB   0E6H      ;E4
05E3 00                DB   0
05E4 00                DB   0
05E5 00                DB   0
05E6 DA                DB   0DAH      ;F5
05E7 BD                DB   0BDH      ;G5
05E8 A3                DB   0A3H      ;A5
05E9 00                DB   0
05EA 0F                DB   0FH       ;B5
05EB 85                DB   85H       ;C4
05EC 72                DB   72H       ;D4
05ED 00                DB   0
05EE E4                DB   0E4H      ;E4
05EF 5C                DB   5CH       ;F4
05F0 47                DB   47H       ;G4
05F1 00                DB   0
05F2 43                DB   43H       ;A4
05F3 38                DB   38H       ;B4
05F4 33                DB   33H       ;C5
05F5 00                DB   0
05F6 2D                DB   2DH       ;D5
05F7 24                DB   24H       ;E5
05F8 20                DB   20H       ;F5
                       ;
                       ;ROCKET FLAG  UFO DEMO PROGRAM
                       ;
05F9 3E8A    ROCT:     MVI  A,8AH     ;ALL AMBER LED  ON
05FB D320              OUT  LOUT      ;TURN ON AMBER  LED
05FD 11C80E            LXI  D,ROCD    ;ROCKET DISPLAY ADDRESS
0600 DF                RST  3         ;STORE MESSAGE  DISPLAY
0601 0E80    ROCT1:    MVI  B,80H     ;1 SECOND DELAY  LOOP COUNTER
0603 CDE901  ROCT2:    CALL DCI       ;UPDATE DISPLAY
0606 05                DCR  B         ;COUNTER
0607 C20306            JNZ  ROCT2     ;REPEAT IF  1 SECOND
060A DF                RST  3         ;BEEP
060B 21F00F            LXI  H,LOSER   ;COUNT DOWN IS  OVER LIGHT
060E 35                DCR  M         ;DECREMENT IF
060F C20106            JNZ  ROCT1     ;IF NOT LAST  IF YES SCORE
0612 0600              MVI  B,6       ;START BLAST-OFF  SEQUENCE
0614 1602              MVI  D,2       ;START BLAST-OFF  DURATION
0616 3EE7    ROCT3:    MVI  A,0E7H    ;LED PATTERN
0618 CD5506            CALL ROCT5     ;SEQUENCE
061B 3EDB              MVI  A,0DBH    ;LED PATTERN
061D CD5506            CALL ROCT5     ;SEQUENCE
0620 3EBD              MVI  A,0BDH    ;LED PATTERN
0622 CD550E            CALL ROCT5     ;SEQUENCE
0625 3E7E              MVI  A,7EH     ;LED PATTERN
0627 CD5506            CALL ROCT5     ;SEQUENCE
062A C2160E            JNZ  ROCT3     ;REPEAT IF  FOR SMAL
062D AF                XRA  A         ;CLEAR A
062E 2F                CMA            ;SET A TO FF
062F D320              OUT  LOUT      ;TURN OFF ALL  LEDS
0631 010005            LXI  B,0500H   ;1 SECOND DELAY  VALUE
0634 CD3004            CALL DELB      ;FOR 1 SECOND  DELAY
0637 3E03              MVI  A,3       ;NOTE COUNTER
0639 0634    ROCT4:    MVI  B,34H     ;1ST 3 NOTE FREQ
063B F5                PUSH PSW       ;SAVE NOTE COUNT
063C CD1200            CALL BEEP1     ;PLAY NOTE
063F F1                POP  PSW       ;RESTORE NOTE COUNT
0640 010000            LXI  B,0       ;PAUSE VALUE
0643 CD3004            CALL DELB      ;PAUSE BETWEEN NOTES
0646 3D                DCR  A         ;NOTE COUNTER
0647 C23906            JNZ  ROCT4     ;IF NOT LAST  IF YES SI NOTES
064A 0685              MVI  B,85H     ;LAST NOTE FREQ
064C 1640              MVI  D,40H     ;LAST NOTE DURATION
064E CD4704            CALL BEEP2     ;PLAY LAST NOTE
0651 CF                RST  1         ;RETURN TO MON  OR
0652 C3F805            JMP  ROCT      ;RUN PROGRAM AGAIN
0655 D628    ROCT5:    OUT  LOUT      ;UPDATE LEDS
0657 CD4704            CALL BEEP2     ;BEEP
065A 05                DCR  B         ;DECREASE FREQ  BY  1
065B C9                RET            ;
                       ;
065C 09      ROCM:     DB   09H       ;B
065D 1C                DB   1CH       ;A
065E 18                DB   18H       ;SMALL C
065F 15                DB   15H       ;SMALL D
0660 1B                DB   1BH       ;SMALL E
0661 10                DB   10H       ;SPACE
                       ;
                       ;STOPWATCH
                       ;
0662 118C06  STW:      LXI  D,STWN    ;ADDRESS OF DISPLAY  DATA
0665 DF                RST  3         ;ZERO DISPLAY
0666 11FF00            LXI  D,00FFH   ;RUN AND S-S-S  FLAG
0669 AF      STW1:     XRA  A         ;CLEAR A
066A 32F60B            STA  UDSPS     ;TURN OFF DECIMAL POINTS
066D CDE901            CALL DCI       ;UPDATE DISPLAY
0670 0EF5              MVI  B,0F5H    ;DELAY CONSTANT
0672 05      STW2:     DCR  B         ;DELAY COUNTER
0673 23                INX  H         ;DELAY INSTRUCTION
0674 C27206            JNZ  STW2      ;IF NOT DONE
0677 3E0F              MVI  A,0FH     ;0-KEY SCAN INSTR
0679 D328              OUT  SCAN      ;0-KEY ROW
067B DB18              IN   KEY       ;INPUT
067D FEFE              CPI  0FEH      ;CHECK FOR 0 KEY
067F CA8206            JZ   STA       ;IF PUSHED
0682 3EF7              MVI  A,0F7H    ;5-KEY SCAN INSTR
0684 D328              OUT  SCAN      ;5-KEY ROW
0686 DB18              IN   KEY       ;INPUT
0688 FCFB              CPI  0FBH      ;CHECK FOR 5 KEY
068A C27B0E            JNZ  STW       ;IF NOT PUSHED
068D B8                CMP  E         ;WAS IT PUSHED  PREVIOUSLY
068E CA930E            JZ   STW2      ;IF IT WAS
0691 5F                MOV  E,A       ;PUSHED STATUS  CODE
0692 14                INR  D         ;CHANGE RUN MODE  START STOP
0693 7A      STW3:     MOV  A,D       ;CURRENT RUN MODE

Hex
Adrs Contents Label    Instruction          Comments

0694 1F                RAR            ;FLAG MODE BIT  TO CARRY
0695 D26906            JNC  STW1      ;IF IN STOP MODE
0698 21F00B            LXI  H,LDSPA   ;20 SEC MEM LOCATION
069B 34      STW3:     INR  M         ;INCREMENT DIGIT  DISPLAY
069C 7E                MOV  A,M       ;LOAD DIGIT IN A
069D FE0A              CPI  0AH       ;OVERFLOW
069F C2A806            JNZ  STW5      ;IF DIGIT 9 UP  LEVEL
06A2 3600              MVI  M,0       ;SEC. DIGIT TO 0
06A4 2C      STW4:     INR  L         ;NEXT HIGHER DIGIT  ADDR
06A5 C39B0F            JMP  STW3      ;REPEAT SEQUENCE  FOR MIN. DIGIT
06A8 21F30B  STW5:     LXI  H,LDSPS   ;10 SEC MEM LOCATION
06AB 7E                MOV  A,M       ;LOAD DIGIT IN A
06AC FE06              CPI  6         ;OVERFLOW
06AE C2B906            JNZ  STW1      ;IF DIGIT 5 OR LESS
06B1 3600              MVI  M,0       ;SEC DIGIT TO 0
06B3 2D                INR  L         ;MIN DIGIT ADDR
06B4 C3A406            JMP  STW4      ;REPEAT SEQUENCE  FOR MIN DIGIT
06B7 1EFF    STW6:     MVI  E,0FFH    ;KEY NOT PUSHED CODE
06B9 C3930E            JMP  STW2      ;CONTINUE IN CONFUSED RUN MODE
06BC 0A      STNM:     DB   0         ;ZERO DISPLAY  BCD CODE TABLE
06BD 00                DB   0
06BE 00                DB   0
06BF 00                DB   0
06C0 10                DB   10H
06C1 00                DB   0
                       ;
                       ;SNAKE PADDLE GAME
                       ;
06C2 21A0FF  SNAKE:    LXI  H,0FFA0H  ;INITIALIZE STACK  POINTER
06C5 0E47              MVI  B,RIGHT   ;RIGHT PLAYER KEY
06C7 E5                PUSH H         ;SAVE SCORE
06C8 3E40    SERV:     MVI  A,40H     ;SERVE SPEED
06CA 320008            STA  SPEED     ;CURRENT BALL  SPEED
06CD CD9702            CALL BLNK      ;CLEAR THE DISPLAY
06D0 E1                POP  H         ;GET SCORE
06D1 0E1C              MVI  C,1CH     ;SETS FOR  TONE
06D3 79                MOV  A,E       ;PLAYER OK CODE
06D4 FEF7              CPI  RIGHT     ;RIGHT PLAYER  LOST
06D6 79                MOV  A,C       ;DOESN'T CHANGE  CODE
06D7 C2E406            JNZ  LLOSE     ;IF LEFT PLAYER  LOST
06DA 32F500            STA  UDSP5     ;LEFT SERVE  INDICATOR
06DD 2E                INR  H         ;LEFT PLAYER  GETS POINT
06DE 7C                MOV  A,H       ;SCORE
06DF 0EFB              MVI  B,LEFT    ;LEFT PLAYER  KEY CODE
06E1 C3EB0E            JMP  SERV1     ;CONTINUE
06E4 32F00B  LLOSE:    STA  UDSP2     ;RIGHT SERVE  INDICATOR
06E7 2D                INR  L         ;RIGHT PLAYER  SCORE
06E8 7D                MOV  A,L       ;RIGHT SCORE
06E9 0EF7              MVI  B,RIGHT   ;RIGHT PLAYER  KEY
06EB FE0A    SERV1:    CPI  0AH       ;LAST POINT
06ED E5                PUSH H         ;SAVE SCORE
06EE 21F00B            LXI  H,UDSP4   ;DISPLAY SCORE
06F1 CDE901            CALL DCI       ;UPDATE DISPLAY
06F4 C20007            JNZ  SERV2     ;IF NOT LAST  GAME REQUEST
06F7 CD9907  ENDGM:    CALL RSTGM     ;NEW GAME REQUEST
06FA C4C801            CALL SDS       ;UPDATE DISPLAY
06FD C2F706            JNZ  ENDGM     ;WAIT FOR NEW  GAME SIGNAL
0700 CD9907  SERV2:    CALL RSTGM     ;NEW GAME REQUEST
0703 CAC206            JZ   SNAKE     ;IF NEW GAME  REQUEST
0706 CDA207            CALL PADL      ;CHECK PADDLE
0709 C30007            JMP  SERV2     ;IF NOT PUSHED
070C C5      RBND:     PUSH B         ;SAVE HITTING PLAYER
070D 7A                MOV  A,D       ;HITTING PLAYER  KEY
070E FEFB              CPI  LEFT      ;LEFT PLAYER  HIT
0710 01D407            LXI  B,LSM     ;LEFT PLAYER  HIT COUNTER
0713 CA1907            JZ   RSRV      ;IF LEFT PLAYER  WAS HIT
0716 01E207            LXI  B,RSM     ;RIGHT PLAYER  HIT COUNTER
0719 CD3807  RSRV:     CALL PASS      ;START BALL MOVING
071C C1                POP  B         ;RESTORE LAST  TONE CODE
071D 7B                MOV  A,E       ;HITTING PLAYER  KEY
071E FEFB              CPI  LEFT      ;LEFT PLAYER  HIT
0720 0EF7              MVI  B,RIGHT   ;RIGHT PLAYER  KEY
0722 CA2707            JZ   RBND1     ;IF LEFT CAME  FROM LEFT PLAYER
0725 0EFB              MVI  B,LEFT    ;SET  PLAYER  KEY
0727 CD5A07  RBND1:    CALL PLAY      ;CHECK PADDLE
072A 7A                MOV  A,I       ;POINT LOSS  REGISTER
072B FEFF              CPI  PNTLS     ;POINT LOSS  CODE
072D C20C07            JNZ  RBND      ;IF NO LOST  POINT GO TO SAME
0730 0E28              MVI  C,LOSTN   ;LOST POINT  TONE CODE
0732 CD8907            CALL TRIL      ;PLAY TRILL
0735 C3C806            JMP  SERV      ;NEW SERVE
                       ;
                       ;PASSES BALL FROM ONE PLAYER  TO THE OTHER
                       ;
0738 CD9702  PASS:     CALL BLNK      ;CLEAR THE DISPLAY
073B 3608              MVI  M,08H     ;WRITE DISPLAY  MASK
073D 160E              MVI  D,0EH     ;BALL PATTERN  LOCATION
073F 0A      PASS1:    LDAX B         ;BALL POSITION  DISPLAY ADDR
0740 5F                MOV  E,A       ;SAVE IT
0741 E612              ANI  12H       ;BIT POSITION
0743 0F                RRC            ;SHIFT
0744 4F                MOV  L,A       ;SAVE
0745 7D                MOV  A,L       ;POSITION
0746 0F                RRC            ;POSITION
0747 B5                ORA  L         ;MASK
0748 E603              ANI  3         ;MASK
074A C4FC              CNZ  0FCH      ;OFFSET FOR R-H
074C 6F                MOV  L,A       ;LOW IC  DISPLAY
074D 2F                MOV  A,E       ;RESTORE MESSAGE
074E F640              ORI  40H       ;SET DP+E+B SEGMENTS
0750 77                MOV  M,A       ;SEND TO UDSP2
0751 CDAD07            CALL TIMER     ;DISPLAY IT
0754 05                DCR  B         ;NEXT MESSAGE
0755 15                DCR  D         ;MESSAGE COUNTER
0756 C23F07            JNZ  PASS1     ;IF LAST BALL  NOT
0759 C9                RET            ;
                       ;PLAY ROUTINE WHEN PASS IS  NOT
```

Table F-2. ROM Listing (Continued)

```
Hex
Adrs Contents  Label    Instruction          Comments

                        ;
075A 16FF      PLAY:    MVI  D,FNTLS  ;POINT LOSS CODE
075C CDA207             CALL PADL     ;CHECK PADDLE
075F C8                 RZ            ;IF PUSHED TOO LATE
0760 3A0308             LDA  SPEED    ;PRESENT BALL SPEED
0763 5F                 MOV  E,A      ;SAVE IT
0764 57                 MOV  D,A      ;SET REACTION SPEED COUNTER VALUE
0765 0F                 RRC           ;HALVE BALL SPEED
0766 4F                 MOV  C,A      ;SAVE IT
0767 7A        PLAY1:   MOV  A,I      ;RESTORE COUNTER VALUE
0768 3D                 DCR  A        ;COUNTER
0769 B9                 CMP  C        ;HALF TIME
076A CA8207             JZ   FSTR     ;IF PADL NOT PUSHED BY THIS TIME
076D 57                 MOV  D,A      ;SAVE COUNT
076E CDA207             CALL PADL     ;CHECK PADDLE
0771 C26707             JNZ  PLAY1    ;IF NOT PUSHED
0774 0E08               MVI  C,SLOTN  ;SLOWER TRILL CODE
0776 CDB907             CALL TRIL     ;PLAY TRILL
0779 7B                 MOV  A,E      ;ORIGINAL BALL SPEED
077A FE40               CPI  40H      ;SLOWEST SPEED
077C C8                 RZ            ;IF ALLREADY AT SLOWEST SPEED
077D 07                 RLC           ;HALVE BALL SPEED
077E 32000B             STA  SPEED    ;STORE IT
0781 C9                 RET
                        ;
                        ;FASTER INCREASES RETURN SPEED
                        ;
0782 7A        FSTR:    MOV  A,I      ;RESTORE COUNTER VALUE
0783 16FF               MVI  D,FNTLS  ;LOSE SIGNAL CODE
0785 3D                 DCR  A        ;COUNTER
0786 C8                 RZ            ;IF PADDLE NOT PUSHED IN TIME
0787 57                 MOV  D,A      ;SAVE A
0788 CDA207             CALL PADL     ;CHECK PADDLE
078B C28207             JNZ  FSTR     ;IF NOT PUSHED
078E 0E01               MVI  C,FSTTN  ;FASTER SIGNAL CODE
0790 CDB907             CALL TRIL     ;PLAY TRILL
0793 7B                 MOV  A,E      ;ORIGINAL BALL SPEED
0794 0F                 RRC           ;DOUBLE BALL SPEED
0795 32000B             STA  SPEED    ;STORE IT
0798 C9                 RET
                        ;
                        ;RESET GAME BUTTON PUSHED ?
                        ;
0799 3EFB      RSTGM:   MVI  A,0FBH   ;0 KEY INPUT CODE
079B D320               OUT  SCAN     ;SET SCAN LATCH
079D DB10               IN   KEY      ;INPUT KEYS
079F FEFE               CPI  0FEH     ;0 KEY INPUT CODE
07A1 C9                 RET
                        ;
                        ;PADDLE PUSHED?
                        ;
07A2 CDC801 PADL:    CALL SDS      ;UPDATE DISPLAY
07A5 78                 MOV  A,E      ;PADDLE KEY SCAN MASK
07A6 D320               OUT  SCAN     ;SET SCAN LATCH
07A8 DB10               IN   KEY      ;INPUT KEYS
07AA FEFB               CPI  0FBH     ;ACTIVE-TABLE KEY INPUT CODE
07AC C9                 RET
                        ;
                        ;TIME CONTROLS BALL SPEED
                        ;
07AD 3A000B TIMER:   LDA  SPEED    ;PRESENT BALL SPEED
07B0 CDC801 TIME1:   CALL SDS      ;UPDATE DISPLAY - DELAY
07B3 1E02               MVI  E,2      ;COUNTER
07B5 C2B007             JNZ  TIME1    ;IF COUNTER NOT ZERO
07B8 C9                 RET
                        ;
                        ;TRILL MAKES SPEAKER SOUNDS
                        ;
07B9 D5        TRIL:    PUSH D        ;
07BA C5                 PUSH B        ;
07BB 0606               MVI  B,6      ;START FREQ
07BD 1601               MVI  D,1      ;TONE DURATION
07BF C5        TRIL1:   PUSH B        ;SAVE START FREQ
07C0 CD4704             CALL BEEP2    ;TONE
07C3 C1                 POP  B        ;START FREQ
07C4 04                 INR  B        ;DECREASE FREQ
07C5 79                 MOV  A,C      ;TRILL CODE
07C6 FE01               CPI  FSTTN    ;FASTER CODE
07C8 C2CD07             JNZ  TRIL2    ;IF NOT
07CB 05                 DCR  B        ;RESTORE FREQ
07CC 05                 DCR  B        ;INCREASE FREQ
07CD B8        TRIL2:   CMP  B        ;LAST BEEP
07CE C2BF07             JNZ  TRIL1    ;IF NOT DONE
07D1 C1                 POP  B        ;
07D2 D1                 POP  D        ;
07D3 C9                 RET
                        ;
                        ;LEFT SERVE MESSAGE TABLE
                        ;
07D4 F7        LSM:     DB   0F7H     ;DISPLAY SEGMENT CODES
07D5 F5                 DB   0F5H
07D6 E7                 DB   0E7H
07D7 E5                 DB   0E5H
```

```
Hex
Adrs Contents  Label    Instruction          Comments

07D8 E1                 DB   0E1H
07D9 A1                 DB   0A1H
07DA A7                 DB   0A7H
07DB B5                 DB   0B5H
07DC B7                 DB   0B7H
07DD 97                 DB   097H
07DE 96                 DB   096H
07DF B4                 DB   0B4H
07E0 A6                 DB   0A6H
07E1 A0                 DB   0A0H
                        ;
                        ;RIGHT SERVE MESSAGE TABLE
                        ;
07E2 EC        RSM:     DB   0ECH     ;DISPLAY SEGMENT CODES
07E3 EE                 DB   0EEH
07E4 FC                 DB   0FCH
07E5 FE                 DB   0FEH
07E6 DE                 DB   0DEH
07E7 9E                 DB   09EH
07E8 BC                 DB   0BCH
07E9 AE                 DB   0AEH
07EA AC                 DB   0ACH
07EB A5                 DB   0A5H
07EC A0                 DB   0A0H
07ED A6                 DB   0A6H
07EE B4                 DB   0B4H
07EF 96                 DB   096H
0800           SPEED    EQU  0800H    ;
00FF           PNTLS    EQU  0FFH     ;
00FB           LEFT     EQU  0FBH     ;
00F7           RIGHT    EQU  0F7H     ;
0028           LOSTN    EQU  028H     ;
000B           SLOTN    EQU  00BH     ;
0001           FSTTN    EQU  001H     ;
                        ;LESSON 2  COUNTER PROGRAM
                        ;
07F0 3E00      XSTART:  MVI  A,0       ;SET A REGISTER TO 0
07F2 3C        XLOOP:   INR  A         ;INCREMENT A REGISTER
07F3 FE0A               CPI  10       ;COMPARE A REGISTER TO 10
07F5 CAF007             JZ   XSTART   ;GO TO BEGINNING IF A=10
07F8 C3F207             JMP  XLOOP    ;INCREMENT AGAIN
07FB 00                 NOP
07FC 00                 NOP
07FD 00                 NOP
07FE 00                 NOP
07FF DE        CHKSM:   DB   0DEH     ;
                        ;EQUATES
                        ;
0018           KEY      EQU  18H      ;KEY INPUT PORT ADRS
0020           SIN      EQU  20H      ;KEY INPUT PORT ADRS
0028           SCAN     EQU  28H      ;KEY SCAN PORT ADRS
0030           LOUT     EQU  30H      ;KEY OUTPUT PORT ADRS
0038           DSP      EQU  38H      ;KEY DISPLAY PORT ADRS
000B           DIM      EQU  0BH      ;DEFAULT INTERRUPT MASK
0800           PC       EQU  0800H    ;DEFAULT PROGRAM LOCATION
0B80           USP      EQU  0BE0H    ;DEFAULT USER STACK POINTER
0BCE           MSP      EQU  0BCEH    ;MONITOR STACK LOCATION
003C           TIME     EQU  3CH      ;10MS TIME CONSTANT FOR DELAY LOOP
0006           FREQ     EQU  06H      ;BEEP DEFAULT FREQUENCY CONSTANT
0004           DURA     EQU  04H      ;BEEP DEFAULT DURATION CONSTANT
0AF0           RS4C     EQU  0AF0H    ;RST4   LINK
0AF3           RS5C     EQU  0AF3H    ;RST5   LINK
0AF6           RS55C    EQU  0AF6H    ;RST5.5 LINK
0AF9           RS6C     EQU  0AF9H    ;RST6   LINK
0AFC           RS65C    EQU  0AFCH    ;RST6.5 LINK FOR INTER. KEY
0AFF           TPP      EQU  0AFFH    ;TOP OF PROTECTED RAM
0AEF           UR       EQU  0AEFH    ;UPPER RAM BELOW RST LINKS
0BD1           TSAVSP   EQU  0BD1H    ;TEMPORARY STACK POINTER SAVE ADRS
0BD3           TSAVH    EQU  0BD3H    ;TEMPORARY H&L REG SAVE ADRS
0BD5           RAML1    EQU  0BD5H    ;RAM LINK TO USER PROG
0BD6           RS       EQU  0BD6H    ;RUN STATUS ADRS
0BD7           RAML2    EQU  0BD7H    ;RAM LINK TO USER PROG
0BDB           SAVIM    EQU  0BDBH    ;SAVE INTERRUPT MASK ADRS
0BDC           SAVPC    EQU  0BDCH    ;SAVE PROGRAM COUNTER ADRS
0BDE           SAVSL    EQU  0BDEH    ;SAVE STACK POINTER LOW ADRS
0BDF           SAVSH    EQU  0BDFH    ;SAVE STACK POINTER HIGH ADRS
0BE0           SAVL     EQU  0BE0H    ;SAVE L REG ADRS
0BE2           SAVE     EQU  0BE2H    ;SAVE E REG ADRS
0BE7           SAVA     EQU  0BE7H    ;SAVE A REG ADRS
0BE8           UDKY     EQU  0BE8H    ;UNDECODED KEY SCAN #
0BF0           UDSP0    EQU  0BF0H    ;UNDECODED DISPLAY DIGIT 0 ADRS
0BF1           UDSP1    EQU  0BF1H    ;UNDECODED DISPLAY DIGIT 1 ADRS
0BF2           UDSP2    EQU  0BF2H    ;UNDECODED DISPLAY DIGIT 2 ADRS
0BF3           UDSP3    EQU  0BF3H    ;UNDECODED DISPLAY DIGIT 3 ADRS
0BF5           UDSP5    EQU  0BF5H    ;UNDECODED DISPLAY DIGIT 5 ADRS
0BF6           UDSP6    EQU  0BF6H    ;UNDECODED DISPLAY DIGIT 6 ADRS
0BF8           RNP      EQU  0BF8H    ;REGISTER MESSAGE POINTER
0BFA           DDSP0    EQU  0BFAH    ;DECODED DISPLAY DIGIT 0 ADRS
0BFC           DDSP2    EQU  0BFCH    ;DECODED DISPLAY DIGIT 3 ADRS
0BFF           DDSP5    EQU  0BFFH    ;DECODED DISPLAY DIGIT 5 ADRS
0000                    END  0H
```

Table F-2. ROM Listing (Continued)

APPENDIX G

Expanding the Microprocessor Lab

INTRODUCTION

This appendix contains the information necessary to expand the Microprocessor Lab using peripheral equipment. The use and general description of all bus and special signals are provided in Table G-1. Refer to the schematic diagram when reading signal descriptions. For a detailed explanation of all 8085 microprocessor signals refer to the data sheet in Appendix H and the applicable Intel manuals.

The 48K of addresses beginning at address 4000 and ending with address FFFF are completely available for memory expansion. Likewise, I/O addresses 40 through FF are available for I/O expansion using the IO/$\overline{M}$ line on P1-S.

Edge connector signals may be accessed through a pair of 44 pin connectors. TRW Cinch manufactures 44 pin connectors that accommodate the 3/32" board thickness of the μLab under the part number 251-22-30 341. A minimum order quantity may be required. These connectors are also available individually from Hewlett-Packard under the HP Part Number 1251-2680.

SIGNAL	LOCATION	USE	GENERAL DESCRIPTION
ADDRESS BUS A0-A15	(pins [P1] 1-16)	System expansion, activity monitor (Logic Analyzer), and DMA (direct memory access) of on-board memory and I/O.	All address lines are demultiplexed and buffered. HLDA signal causes bus to float.
DATA BUS D0-D7	(pins [P2] 1-8)	System expansion, activity monitor (Logic Analyzer), and DMA of on-board memory and I/O.	Direct connection to the system data bus. Requires external buffering near the edge connector if more than 1 LS TTL load is connected, or if appreciable cable or board capacitance is added (>100 pf). HLDA signal causes bus to float (with 10K pull-up resistors).
DATA INPUT BUS D0-D7	(pins [P2] K-T)	Allows 8 external input signals to be connected to the μLab input port (address 2000) for system expansion.	Inputs go directly to input port IC13 and to input switch S3. Because this switch ties these lines directly to ground when placed in the down switch position, individual input data switches must be in the up (logic 1) position for any externally controlled lines. All switches can be placed in the up position if desired. 10K pull-up resistors are on each line. The IN 20 instruction will also access this port.
DATA OUTPUT BUS D0-D7	(pins [P2] 9-16)	Provides 8 latched output signals for system expansion using the μLab's output port (address 3000).	The output pins of output port IC15 go directly to output LEDs DS12 to DS19 and the edge connector. The OUT 30 instruction will also access this port.
INTERRUPTS RST5.5 INTR	 (P1-L) (P1-K)	System expansion.	RST5.5 and INTR provide additional interrupt capability for the μLab. To use these interrupts, jumpers J3 and J2, respectively, must first be opened on the μLab PC board.*
$\overline{\text{INTA}}$ (Interrupt Acknowledge)	(P1-J)	Handling of an INTR interrupt.	This output from the microprocessor responds to the device that caused the INTR interrupt by asking for an instruction.
CLOCK OUT	(P1-D)	System expansion and synchronization with external circuits.	A 2 MHz TTL level square wave signal controlled by the crystal oscillator on the μLab.
VA (Valid Address)	(P1-V)	System expansion, address decoding and activity monitor (Logic Analyzer).	When this signal is high, a valid address is present on the address bus and either a read or a write is being performed on memory or I/O.

Table G-1. Edge Connector Signals

SIGNAL	LOCATION	USE	GENERAL DESCRIPTION
ALE (Address Latch Enable) $\overline{ALE}$	(P1-A) (P1-U)	System expansion using multiplexed address/data bus chips, and activity monitor (Logic Analyzer).	Microprocessor generated signals that indicate valid address data is present on the data bus. Both true and complement signals are provided for user convenience.
$\overline{WR}$ (Write)	(P1-T)	System expansion, activity monitor (Logic Analyzer), and DMA of on-board RAM and output ports.	Buffered output. HLDA signal causes it to float with DS7A pulling it up. An external circuit must be able to provide 4 ma in the logic 0 state.
$\overline{RD}$ (Read)	(P1-R)	System expansion, activity monitor (Logic Analyzer), and DMA of on-board memory and input ports.	Buffered output. HLDA signal causes it to float with DS7B pulling it up. An external circuit must be able to provide 4 ma in the logic 0 state.
IO/$\overline{M}$ (Input-output/ memory)	(P1-S)	System expansion of memory or I/O.	Microprocessor signal indicating either I/O or memory operation.
SID (Serial Input Data)	(P1-N)	System expansion.	A single bit input port to the microprocessor controlled by the RIM instruction. To use this input, jumper J4 must first be opened on the μLab PC board.*
SOD (Serial Output Data)	(P1-W)	System expansion.	One bit of serial output data identical to the data sent to the speaker, but separately buffered. Controlled by the SIM instruction. HLDA signal causes this line to float.
READY	(P1-B)	System expansion with slow bus devices.	Normally used by the μLab to perform the single-step functions. A low logic level on this line forces the microprocessor to wait for slow bus devices. To use this input, jumper J5 must first be opened on the μLab PC board.* If the single-step feature is also desired, the step signal must be externally gated back into the READY line (see the STEP line description that follows).
STEP	(P1-X)	Retains the μLab's single-step functions for internal and external memory and I/O when an external READY input is used.	When jumper J5 is opened to allow external control of the READY input, the μLab single-step circuits are no longer gated into the microprocessor. To re-insert the single-step features, the circuit shown in Figure G-1 can be used.

Table G-1. Edge Connector Signals (Continued)

SIGNAL	LOCATION	USE	GENERAL DESCRIPTION
$\overline{\text{RESET IN}}$	(P1-C)	External power-on reset control of μLab.	A low logic level on this line causes the μLab to execute the same power-on reset cycle that it does when first turned on.
RESET OUT	(P1-M)	Initializing external circuits.	This line goes to a logic 1 whenever the microprocessor is reset (during power-up or when the $\overline{\text{RESET IN}}$ line is pulled low).
HOLD	(P1-E)	DMA of μLab memory and I/O.	A high logic level on this input causes the microprocessor to enter the hold state. In this state the HLDA line goes high, forcing the address, data, and control lines into their high impedance state (see HLDA output). To use this input, jumper J1 must first be opened on the μLab PC board.*
HLDA (Hold Acknowledge)	(P1-P)	Disables all system memory and I/O control lines so that an external controller can perform DMA.	When this line is high, the μLab's microprocessor is in the hold state and the address, data, and control bus lines ($\overline{\text{RD}}$, $\overline{\text{WR}}$, IO/$\overline{\text{M}}$) are in their high impedance state. In this mode an external controller can transfer data directly to or from the system bus devices. HLDA results from a HOLD input signal to the microprocessor.
S0 (State 0) S1 (State 1)	(P1-H) (P1-F)	System expansion.	Output signals directly from the microprocessor that provide advanced state and system information.

Table G-1. Edge Connector Signals (Continued)

*NOTE: Circuit jumpers (J1 to J5) are provided on the μLab for system expansion. All five are shorted at the factory using traces on the PC board. To open a jumper, a sharp knife can be used to break the narrow trace that connects the pair of plated-through holes at the jumper position. To restore the short, a solder bridge or piece of wire connecting the holes can be used.

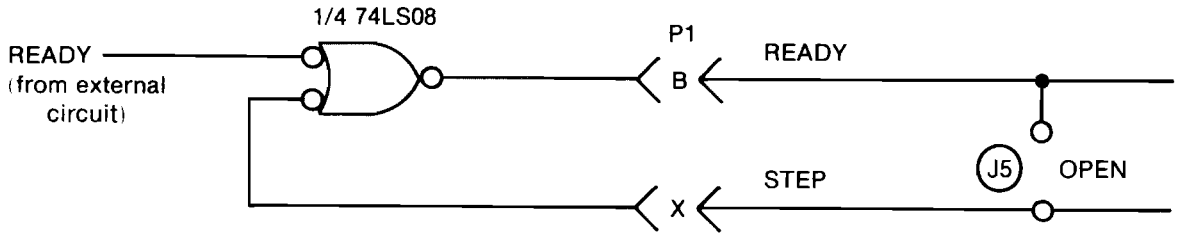


Figure G-1. A Circuit for Retaining the Single-Step Feature When Using an External READY Input

RESTART LINKS

Since the restart (RST) and interrupt entry points are fixed by the 8085 to be in a portion of memory occupied by the μLab's ROM, restart links are used to relocate the user available entry points to specific addresses of the RAM. For example, the RST5 restart (code EF) causes a call to address 0028. The instruction at 0028 is a jump to address 0AF3 in the μLab's RAM. The user would store a jump instruction beginning at 0AF3 that would cause the microprocessor to jump to another address. This address would contain the subroutine desired and would end with a return instruction.

RAM Address	Label	Restart Links
0AF0 0AF1 0AF2	RST4C	User RST4 routine jump
0AF3 0AF4 0AF5	RST5C	User RST5 routine jump
0AF6 0AF7 0AF8	RS55C	User RST5.5 external interrupt routine jump
0AF9 0AFA 0AFB	RST6C	User RST6 routine jump
0AFC 0AFD 0AFE	RS65C	User RST6.5 INTRPT key interrupt routine jump

Table G-2. Restart Links

APPENDIX H

IC Data Sheets

This appendix contains functional specifications for the microprocessor, ROM, and RAM, excerpted from the manufacturer's data sheets. Data sheets for the other devices in the Microprocessor Lab may be found in the following data books:

74LS00, 74LS14, 74LS32, 74LS74, 74LS138, 74LS175, 74LS273, 74LS374	Texas Instruments TTL Data Book
81LS95, 81LS97	National Semiconductor TTL Data Book
8871, 75492	National Semiconductor Interface Data Book

8085A

SINGLE CHIP 8-BIT N-CHANNEL MICROPROCESSOR

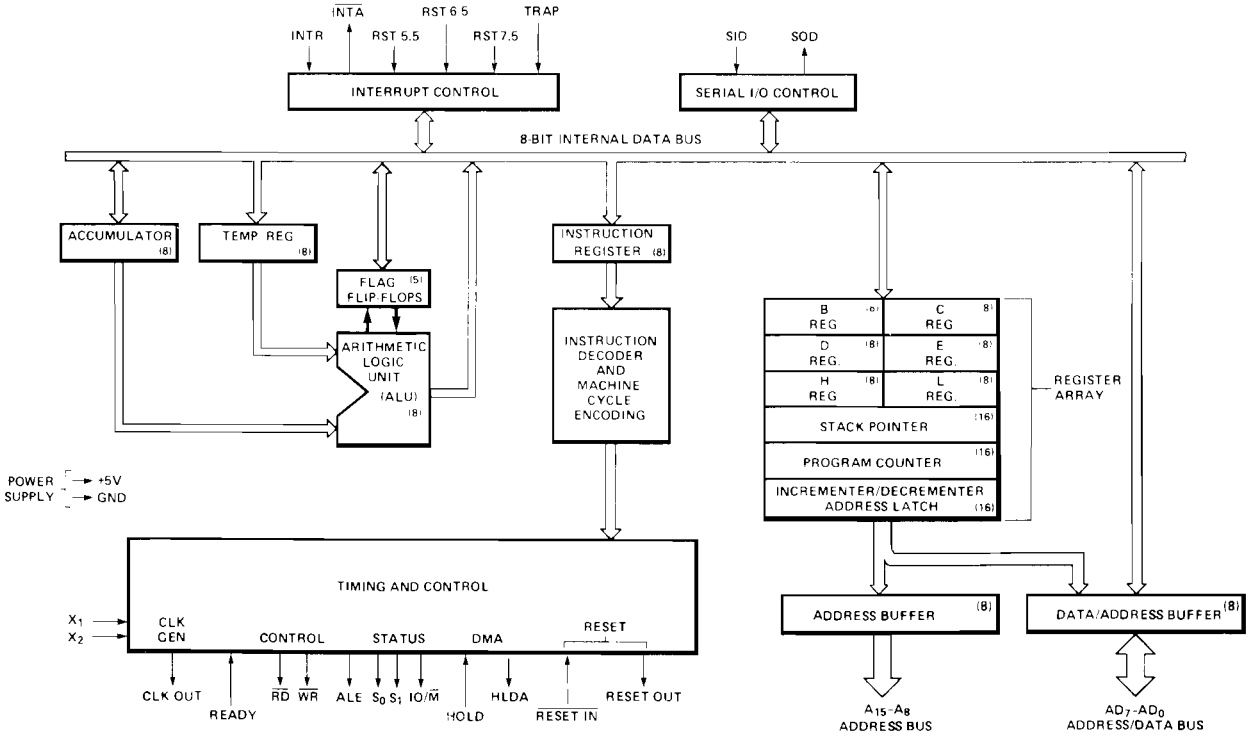
- **Single +5V Supply**
- **100% Software Compatible with 8080A**
- **1.3 μs Instruction Cycle**
- **On-Chip Clock Generator (with External Crystal or RC Network)**
- **On-Chip System Controller; Advanced Cycle Status Information Available for Large System Control**
- **4 Vectored Interrupts (One is Non-Maskable)**
- **Serial In/Serial Out Port**
- **Decimal, Binary, and Double Precision Arithmetic**
- **Direct Addressing Capability to 64K Bytes of Memory**

The Intel® 8085A is a new generation, complete 8 bit parallel central processing unit (CPU). Its instruction set is 100% software compatible with the 8080A microprocessor, and it is designed to improve the present 8080's performance by higher system speed. Its high level of system integration allows a minimum system of 3 IC's: 8085A (CPU), 8156 (RAM), and 8355/8755A (ROM/PROM).

The 8085A incorporates all of the features that the 8224 (clock generator) and 8228 (system controller) provided for the 8080, thereby offering a high level of system integration.

The 8085A uses a multiplexed data bus. The address is split between the 8-bit address bus and the 8-bit data bus. The on-chip address latches of 8155/8156/8355/8755A memory products allows a direct interface with the 8085A.

BLOCK DIAGRAM



Reprinted by Permission of Intel Corporation
Copyright 1977, 1978

PIN DESCRIPTION

The following describes the function of each pin:

A_8-A_{15} (Output 3-State)

Address Bus; The most significant 8-bits of the memory address or the 8-bits of the I/O address, 3-stated during Hold and Halt modes.

AD_{0-7} (Input/Output 3-state)

Multiplexed Address/Data Bus; Lower 8-bits of the memory address (or I/O address) appear on the bus during the first clock cycle of a machine state. It then becomes the data bus during the second and third clock cycles.

3-stated during Hold and Halt modes.

ALE (Output)

Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on-chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information. ALE can also be used to strobe the status information. ALE is never 3-stated.

S_0, S_1 (Output)

Data Bus Status. Encoded status of the bus cycle:

S_1	S_0	
0	0	HALT
0	1	WRITE
1	0	READ
1	1	FETCH

S_1 can be used as an advanced R/$\overline{W}$ status.

$\overline{RD}$ (Output 3-state)

READ; indicates the selected memory or I/O device is to be read and that the Data Bus is available for the data transfer. 3-stated during Hold and Halt.

$\overline{WR}$ (Output 3-state)

WRITE; indicates the data on the Data Bus is to be written into the selected memory or I/O location. Data is set up at the trailing edge of $\overline{WR}$. 3-stated during Hold and Halt modes.

READY (Input)

If Ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

HOLD (Input)

HOLD; indicates that another Master is requesting the use of the Address and Data Buses. The CPU, upon receiving the Hold request, will relinquish the use of buses as soon as the completion of the current machine cycle. Internal processing can continue. The processor can regain the buses only after the Hold is removed. When the Hold is acknowledged, the Address, Data, $\overline{RD}$, $\overline{WR}$, and IO/$\overline{M}$ lines are 3-stated.

HLDA (Output)

HOLD ACKNOWLEDGE; indicates that the CPU has received the Hold request and that it will relinquish the buses in the next clock cycle. HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycle after HLDA goes low.

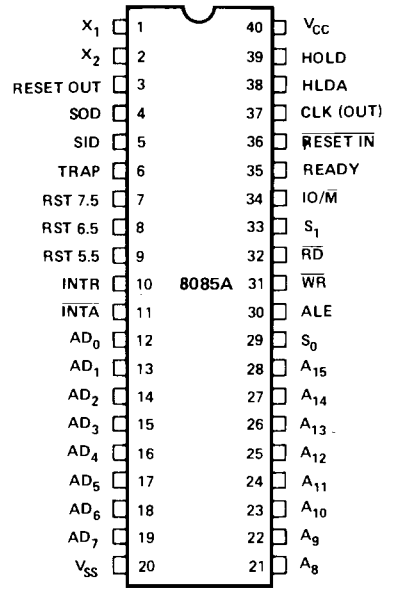


Figure 1. Pin Configuration

INTR (Input)

INTERRUPT REQUEST; is used as a general purpose interrupt. It is sampled only during the next to the last clock cycle of the instruction. If it is active, the Program Counter (PC) will be inhibited from incrementing and an $\overline{INTA}$ will be issued. During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.

$\overline{INTA}$ (Output)

INTERRUPT ACKNOWLEDGE; is used instead of (and has the same timing as) $\overline{RD}$ during the Instruction cycle after an INTR is accepted. It can be used to activate the 8259 Interrupt chip or some other interrupt port.

RST 5.5
RST 6.5
RST 7.5 } (Inputs)

RESTART INTERRUPTS; These three inputs have the same timing as INTR except they cause an internal RESTART to be automatically inserted.

RST 7.5 → Highest Priority
RST 6.5
RST 5.5 → Lowest Priority

The priority of these interrupts is ordered as shown above. These interrupts have a higher priority than the INTR.

Reprinted by Permission of Intel Corporation
Copyright 1977, 1978

TRAP (Input)

Trap interrupt is a nonmaskable restart interrupt. It is recognized at the same time as INTR. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt.

$\overline{\text{RESET IN}}$ (Input)

Reset sets the Program Counter to zero and resets the Interrupt Enable and HLDA flip-flops. None of the other flags or registers (except the instruction register) are affected. The CPU is held in the reset condition as long as Reset is applied.

RESET OUT (Output)

Indicates CPU is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.

X_1, X_2 (Input)

Crystal or R/C network connections to set the internal clock generator. X1 can also be an external clock input instead of a crystal. The input frequency is divided by 2 to give the internal operating frequency.

CLK (Output)

Clock Output for use as a system clock when a crystal or R/C network is used as an input to the CPU. The period of CLK is twice the X_1, X_2 input period.

IO/$\overline{\text{M}}$ (Output)

IO/$\overline{\text{M}}$ indicates whether the Read or Write is to memory or I/O. Tri-stated during Hold and Halt modes.

SID (Input)

Serial input data line. The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.

SOD (output)

Serial output data line. The output SOD is set or reset as specified by the SIM instruction.

V_{CC}

+5 volt supply.

V_{SS}

Ground Reference.

Reprinted by Permission of Intel Corporation
Copyright 1977, 1978

2048x8 Static Read Only Memory

SY2316A
SY2316B

MEMORY PRODUCTS

Synertek®

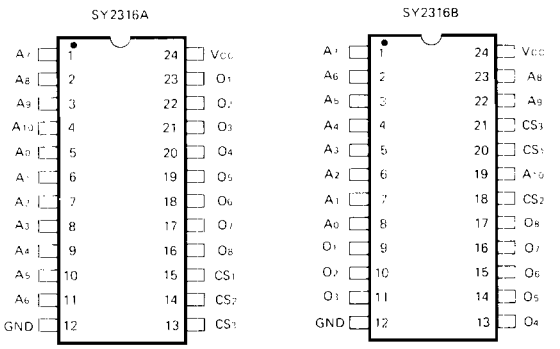
- 2048x8 Bit Organization
- Single +5 Volt Supply
- Metal Mask Programming
- Two Week Prototype Turnaround
- Access Time—550ns /450ns (max.)
- Totally Static Operation
- Completely TTL Compatible
- Three-State Outputs for Wire-OR Expansion
- Three Programmable Chip Selects
- SY2316A – Replacement for Intel 2316A
- SY2316B – Pin Compatible with 2708 EPROM
  – Replacement for Two 2708s

The SY2316A and SY2316B high performance read only memories are organized 2048 words by 8 bits with access times of less than 550 ns and 450 ns. These ROMs are designed to be compatible with all microprocessor and similar applications where high performance, large bit storage and simple interfacing are important design considerations. These devices offer TTL input and output levels with a minimum of 0.4 Volt noise immunity in conjunction with a +5 Volt power supply.

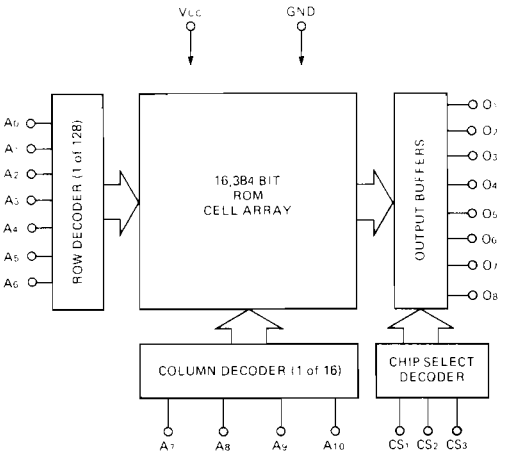
The SY2316A/B operate totally asynchronously. No clock input is required. The three programmable Chip Select inputs allow eight 16K ROMs to be OR-tied without external decoding. Both devices offer three-state output buffers for memory expansion.

Designed to replace two 2708 8K EPROMs, the SY2316B can eliminate the need to redesign printed circuit boards for volume mask programmed ROMs after prototyping with EPROMs.

PIN CONFIGURATION



BLOCK DIAGRAM



ORDERING INFORMATION

Order Number	Package Type	Access Time	Temperature Range
SYC2316A	Ceramic	550ns	0°C to +70°C
SYP2316A	Plastic	550ns	0°C to +70°C
SYC2316B	Ceramic	450ns	0°C to +70°C
SYP2316B	Plastic	450ns	0°C to +70°C

A custom number will be assigned by Synertek.

Reprinted by Permission of Synertek Incorporated
Copyright 1976.

2114
1024 X 4 BIT STATIC RAM

	2114-2	2114-3	2114	2114L2	2114L3	2114L
Max. Access Time (ns)	200	300	450	200	300	450
Max. Power Dissipation (mw)	525	525	525	370	370	370

- **High Density 18 Pin Package**
- **Identical Cycle and Access Times**
- **Single +5V Supply**
- **No Clock or Timing Strobe Required**
- **Completely Static Memory**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**
- **Pin-Out Compatible with 3605 and 3625 Bipolar PROMs**

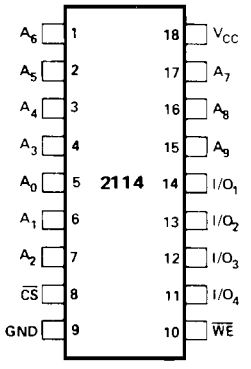
The Intel® 2114 is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using N-channel Silicon-Gate MOS technology. It uses fully DC stable (static) circuitry throughout — in both the array and the decoding — and therefore requires no clocks or refreshing to operate. Data access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

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

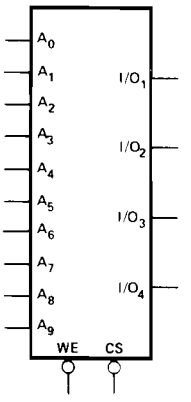
It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. A separate Chip Select ($\overline{CS}$) lead allows easy selection of an individual package when outputs are or-tied.

The 2114 is fabricated with Intel's N-channel Silicon-Gate technology — a technology providing excellent protection against contamination permitting the use of low cost plastic packaging.

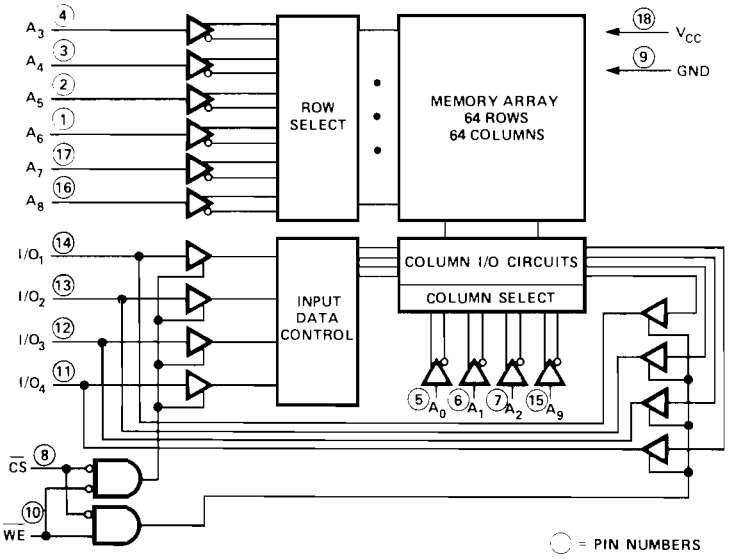
PIN CONFIGURATION



LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

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

Reprinted by Permission of Intel Corporation
Copyright 1977, 1978

GLOSSARY

GLOSSARY

A

Abort Halts the program and returns control to the operator or operating system. Performed in the μLab by the RESET key.

Access Time The time required to receive valid data from a memory device following a read signal.

Accumulator One or more registers associated with the Arithmetic and Logic Unit (ALU), which temporarily store sums and other arithmetical and logical results of the ALU.

A/D See Analog to Digital Converter.

ADC See Analog to Digital Converter.

Adder Device that forms, as output, the sum of two or more numbers presented as inputs.

Address Number that indicates the position of a word in the memory. Typically, addresses are sixteen bits long and therefore can range from 0 to 64K.

Address Bus Set of wires (typically 16) used to transmit addresses, usually from the microprocessor to a memory or I/O device.

Address Decoding Process of selecting a specific address or field of addresses to enable unique devices.

Addressing Modes Various methods of specifying an address as part of an instruction. See Direct Addressing, Indirect Addressing, Immediate Addressing, and Indexed Addressing.

Algorithm Step-by-step procedure for the solution to a problem. First the problem is stated and then an algorithm is devised for its solution.

Alphanumeric Set of all alphabetic and numeric characters.

ALU See Arithmetic and Logic Unit.

Analog Continuous range of voltage or current values.

Analog to Digital Converter Converts analog voltages and currents to the digital representation used by computer systems. This enables the computer to sense real-world signals.

Architecture Logical structure of a computer system.

Arithmetic and Logic Unit (ALU) One of three essential components of a microprocessor. The other two are the registers and the control block. The ALU performs various forms of addition, subtraction, and logic operations, such as ANDing the contents of two registers or masking the contents of a register.

ASCII American Standard Code for Information Interchange. Character code used for representing information in most computer systems.

Assembler Program Translates assembly language statements (mnemonics) into machine lanugage.

Assembly Language Machine-oriented language. A program is normally written as a series of statements using mnemonic symbols that suggest the definition of the instruction. It is then translated into machine language by an assembler program.

Asychronous Any circuit or system that is not synchronized by a common clock signal.

B

Backplane The circuit board that other boards in a system plug into. Usually contains the system buses. Sometimes called a *motherboard.*

Base See Radix.

BASIC An easy-to-learn, easy-to-use language, which is available on most microcomputer systems.

Baud Rate Measure of data flow: the number of signal elements per second. When each element carries one bit, the Baud rate is numerically equal to bits per second (bps). For example, teletypes transmit at 110 baud. Each character is 11 bits, and the TTY transmits 10 characters per second.

BCD Binary Coded Decimal. A 4-bit representation of the 10 decimal digits "0" through "9." Six of the sixteen possible codes are unused. Two BCD digits are usually packed into one byte.

Benchmark Method used to measure performance of a computer in a well-defined situation.

Bidirectional Indicates that signal flow may be in either direction. Common bidirectional buses are three-state or open collector TTL.

Binary A system of numbers using 2 as a base, in contrast to the decimal system which uses 10 as a base. The binary system requires only two symbols, 0 and 1. Two is expressed in binary by the number 10.

Binary Search Technique in which the search interval is divided by two at every iteration.

Bit Contraction of *binary digit*. A single digit in a binary number.

Bit-Slice Method that implements an n-bit slice of the CPU, usually n=4. A bit-slice processor chip implements a complete data path across the CPU. A thirty-two-bit processor could be constructed by using eight 4-bit CPU slices.

Board Tester Product programmed to automatically stimulate the circuits on a PC board and check the responses. Electrical failures can be detected and diagnosed to facilitate board repair.

Boolean Logic Named after George Boole, who defined binary arithmetic and logical operations such as AND, OR, NOT, and XOR.

Bootstrap Program used to initialize the computer. Usually clears memory, sets up I/O devices, and loads the operating system.

Bounce Oscillations and noise generated when a mechanical switch is opened or closed. See Debounce.

Branch See Jump.

Breakpoint Software or hardware device that stops the program and saves the current machine status, under user-specified conditions.

Bubble Memory Memory that utilizes microscopic magnetic domains in an aluminum garnet substrate. Important because they are nonvolatile, solid-state memories.

Buffer An IC that is used to restore the logic drive level.

Bug An error. Eliminating errors is known as *debugging*.

Burn-In Component testing method used to screen out early failures by running the circuit for a specified length of time.

Bus Path for signals that have a common function. Most microprocessors use three buses: the data bus, address bus, and control bus.

Bus Conflict Conflict that occurs when two or more device outputs of opposite logic states are placed on a three-state bus at the same time.

Bus Controller Generates bus commands and control signals.

Bus Driver An IC that is added to a bus to provide sufficient drive between the CPU and the other devices that are tied to the bus. These are necessary because of capacitive loading, which slows down the data rate and prevents proper time sequencing of microprocessor operation.

Bus Termination Method of preventing reflections at the end of a bus. Necessary only in high-speed systems.

Byte Group of 8 bits. Can be used to represent a character. Microcomputer instructions require one, two, or three bytes. A word can be one or more bytes.

C

Call Jump to a subroutine. A jump to the specified address is performed, but the contents of the program counter are saved (usually in the stack) so that the calling program flow can resume when the subroutine is finished.

Carry Flag Flag bit in the microprocessor's status register, which is used to indicate the overflow of an operation by the arithmetic logic unit.

CCD Charge Coupled Device. Serial storage technology that uses MOS capacitors.

Character Generator Circuit that forms the letters or numbers on a display or printer.

Checkerboard Memory test pattern in which alternate 1's and 0's are stored in the cells of the memory array.

Checksum Method used to verify the integrity of data loaded into the computer.

Central Processing Unit Computer module in charge of fetching, decoding, and executing instructions. It incorporates a control unit, an ALU, and related facilities (registers, clock, drivers).

Chip Common name for all ICs.

Chip Enable (CE) See Chip Select.

Chip Select (CS) Usually enables three-state drivers on the chip's output lines. Most LSI chips have one or more chip selects. The CS line is used to select one chip among many.

Clear Set a circuit to a known state, usually zero.

Clock Reference timing source in a system. A clock provides regular pulses that trigger or synchronize events.

Closed-Loop Circuit operating with feedback, whose inputs are a function of its outputs.

Code The machine language itself, or the process of converting from one language to another.

Combinational Logic Circuit arrangement in which the output state is determined only by the present state of the input. Also called *combinatorial logic*.

Comment Field Field within an instruction that is reserved for comments. Ignored by the compiler or the assembler when the program is converted to machine code.

Compiler Translation program that converts high-level instructions into a set of binary instructions (machine code) for execution. Each high-level language requires a compiler or an interpreter. A compiler translates the complete program, which is then executed.

Complement Process of changing each 1 to a 0 and each 0 to a 1.

Computer General-purpose computing system incorporating a CPU, memory, I/O facilities, and power supply.

Condition Code Refers to a limited group of program conditions, such as carry, borrow, overflow, etc., that are pertinent to the execution of instructions. The codes are contained in a condition code register. Same as *flag register.*

Conditional Jump or Call Instruction that when reached in a program will cause the computer either to continue with the next instruction in the original sequence or to transfer control to another instruction, depending on a predetermined condition.

Constant A fixed value.

Control Block Circuits that perform the control functions of the CPU. They are responsible for decoding instructions and then generating the internal control signals that perform the operations requested.

Control Bus Set of control lines in a computer system. Provides the synchronization and control information necessary to run the system.

Control Program Sequence of instructions that guide the CPU through the various operations it must perform. This program is stored permanently in ROM memory where it can be accessed by the CPU during operation. Usually this ROM is located within the microprocessor chip. Same as *microprogram* or *microcode.*

Core Small magnetic toruses of ferrite that are used to store a bit of information. These can be strung on wires so that large memory arrays can be formed. The main advantage of core memory is that it is nonvolatile.

CPS Characters Per Second.

CPU See Central Processing Unit.

Crash Hardware or software malfunction that causes the system to halt or become lost in a loop.

CRC See Cyclic Redundancy Check.

Cross-Assembler Assembler that runs on a processor whose assembly language is different from the language being assembled.

Crosstalk Interference between two signals.

CRT Terminal Computer terminal using a CRT display and a keyboard, usually connected to the computer by a serial link.

Current Tracer Hand-held troubleshooting tool used to detect current flow in logic circuits.

Cycle Time Total time required by a memory device to complete a read or write cycle and become available again.

Cyclic Redundancy Check (CRC) Binary polynomial. Used to generate check information on blocks of data. Similar to a checksum, but is harder to generate and more reliable.

D

D/A See Digital to Analog Converter.

DAC See Digital to Analog Converter.

Daisy Chain Bus line that is interconnected with units so that the signal passes from one unit to the next in serial fashion.

Data General term denoting any or all facts, numbers, letters, and symbols, or facts that refer to or describe an object, idea, condition, situation, or other factors. Connotes basic elements of information that can be processed or produced by a computer. Sometimes data is considered to be expressible only in numerical form, but information is not so limited.

Data Acquisition Collection of data from external sensors, usually in analog form.

Data Base Systematic organization of data files for easy access, retrieval, and updating.

Data Bus Set of lines carrying data. The data bus is usually bidirectional and three-state.

Data Domain Analysis or display of signals in which only their digital value is considered and not their precise voltage or timing. A logic state analyzer displays information in the data domain.

Debouncing Elimination of the bounce signals characteristic of mechanical switches. Debouncing can be performed by either hardware or software.

Debugger Program designed to facilitate software debugging. In general, it provides breakpoints, dump facilities, and the ability to examine and modify registers and memory.

Debugging Process of eliminating hardware or software errors in a system.

Decoder Logic device that decodes binary inputs. A 3-bit decoder (e.g. 74138) will have $2^3=8$ outputs because a 3-bit number can have 8 different values.

Decrement Programming instruction that decreases the contents of a storage location.

Dedicated Set apart for some special use. A dedicated microprocessor is one that has been specially programmed for a single application such as weight measurement, traffic light control, etc. ROMs by their very nature are dedicated memories.

Development System Microcomputer system with all the facilities required for hardware and software development for a given microprocessor. Generally consists of a microcomputer system, CRT display, printer, mass-storage (usually dual floppy-disk drives), PROM programmer, and in-circuit emulator.

Digit Sign or symbol used to convey a specific quantity of information either by itself or with other numbers of its set; 2, 3, 4, and 5 are digits. The base or radix must be specified and each digit's value assigned.

Digital Having discrete states. Most digital logic is binary, with two states (on or off).

Digitize Process of converting an analog quantity into a digital quantity.

Digital to Analog Converter Converts from the digital representation used in computers to the analog signals used in the real world.

DIP Dual In-line Package. Standard IC package with two parallel rows of pins.

Direct Addressing Standard addressing mode, characterized by the ability to reach any point in main storage directly. The address is specified as part of the instruction.

Direct Memory Access (DMA) Method of gaining direct access to main storage in order to perform data transfers without involving the CPU.

Disable Process of inhibiting a device function.

DMA See Direct Memory Access.

DOS Disk Operating System.

Dot Matrix Method of forming characters by using many small dots.

Double Precision Arithmetic Uses two words to represent each number.

Dump Transferring the contents of major blocks of memory.

Dynamic Memory Memory devices whose stored data must be continually refreshed to avoid degradation. Each bit is stored as a charge on a single MOS capacitor. Because of charge leakage in the transistors, dynamic memory must be refreshed every 2 ms by rewriting its entire contents. Normally, this does not slow down the system but does require additional memory refresh logic.

E

EAROM Electrically Alterable Read Only Memory. A ROM that can be modified in-circuit by electrical means.

Echo Action of sending a character input from a keyboard to the printer or display.

Electromagnetic Interference (EMI) Interference caused by electrical fields.

Enable Input signal that allows the device function to occur.

EPROM Erasable Programmable Read Only Memory. A PROM that can be reused. Most EPROMs can be erased by exposing them to ultraviolet light.

Error Correcting Code Code using extra bits to automatically detect and correct errors.

Execute (Cycle) Last cycle of instruction execution. During this time, the instruction operation is performed.

Execution Time Time required for the execution of an instruction.

Exponent Power of ten by which a number is multiplied, used in floating point representation. For example, the exponent in the decimal number 0.9873×10^7 is 7.

F

Falling Edge High-to-low logic transition.

Fan-In Electrical load presented by an input. Usually expressed as the number of equivalent standard input loads.

Fan-Out Electrical load that an output can drive. Usually expressed as the number of inputs that can be driven.

Feedback Information from one or more outputs to be used as inputs in a control loop.

Fetch Reading an instruction from memory.

FIFO First-In-First-Out memory structure. Data is entered at one end and removed from the other. A FIFO is used as a buffer to connect two devices that operate asynchronously.

Firmware Program stored in ROM. Normally, firmware designates any ROM-implemented program.

Fixed-Point Representation Number representation in which the decimal point is assumed to be in a fixed position.

Flag Information bit that indicates some form of demarcation has been reached, such as overflow or carry. Also an indicator of special conditions such as interrupts.

Floating Logic node that has no active outputs. Three-state bus lines, such as data bus lines, float when no devices are enabled.

Floating-Point Representation Technique used to represent a large range of numbers, using a mantissa and an exponent. The precision of the representation is limited by the number of bits allocated to the mantissa. See Mantissa and Exponent.

Floppy Disk Mass-storage device that uses a flexible (floppy) diskette to record information.

Flowchart or Flow Diagram Graphical representation of program logic. Flowcharts enable the designer to visualize the procedure necessary for each item in the program. A complete flowchart leads directly to the final code.

Free-Run Process of allowing a digital circuit (typically a microprocessor) to run without feedback (open-loop). This is done to stimulate other devices in the circuit in a recurring and predictable manner.

G

Glitch Pulse or burst of noise. Also used to indicate any unexplained system failure.

H

Half Splitting Troubleshooting technique used for fault isolation. It involves the examination of circuit nodes approximately midway through a circuit. Once the operational state of these nodes has been determined, the source of the fault can be isolated to the circuits either before or after this point. This process can then be continued.

Halt Command to stop the computer.

Hand Assemble Translate a program from assembly language to machine code without the assistance of an assembler program.

Handshake Control signals at an interface in which the sending device generates a signal indicating that new information is available, and the receiving device then responds with another signal indicating that the data has been received.

Hardware Individual components of a circuit, both passive and active, have long been characterized as hardware in the jargon of the engineer. Today, any piece of data processing equipment is informally called hardware.

Hard-Wired Logic See Random Logic.

Hexadecimal Base 16 number system. Since there are 16 hexadecimal digits (0 through 15) and only ten numerical digits (0 through 9), six additional digits are needed to represent 10 through 15. The first six letters of the alphabet are used for this purpose. Hence, the hexadecimal digits read: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The decimal number 16 becomes the hexadecimal number 10. The decimal number 26 becomes the hexadecimal number 1A.

High-Level Language Problem-oriented programming language, as distinguished from a machine-oriented programming language. A high-level language is closer to the needs of the problem to be handled than to the language of the machine on which it is to be implemented.

High-Order Most significant bits of a word. Typically, bits 8 through 15 of a 16-bit word.

Hold Time The time data must be stable following the completion of a write signal.

I

Immediate Addressing In this mode of addressing, the operand contains the value to be operated on, and no address reference is required.

In-Circuit Emulator (ICE) Debugging aid that connects to the system under test by plugging into the microprocessor's socket. This allows the ICE to gain full control over the system. Typical features include the ability to set breakpoints, single-step a program, examine and modify registers and memory, and divide memory and I/O between the system under test and the ICE system.

Increment Adding the value one to the contents of a register or memory location.

Indexed Addressing Mode in which the actual address is obtained by adding a displacement to a base address.

Index Register Contains address information used for indexed addressing.

Indirect Addressing Addressing a memory location that contains the address of data rather than the data itself.

Initialization Setting a system to a known state.

Input/Output Lines or devices used to transfer information outside the system.

Input Port Circuit that connects signals from external devices as inputs to the microprocessor system.

Instruction Single command within a program. Instructions may be arithmetic or logical, may operate on registers, memory, or I/O devices, or may specify control operations. A sequence of instructions is a program.

Instruction Cycle All of the machine states necessary to fully execute an instruction.

Instruction Decoder Unit that interprets the program instructions into control signals for the rest of the system.

Instruction Register Register inside the microprocessor that contains the opcode for the instruction being executed.

Instruction Set Total group of instructions that can be executed by a given microprocessor. Supplied to the user to provide the basic information necessary to assemble a program.

Interface Indicates a boundary between adjacent components, circuits, or systems that enables the devices to exchange information. Also used to describe the circuit that enables the microprocessor to communicate with a peripheral device.

Interrupt Involves suspension of the normal program that the microprocessor is executing in order to handle a sudden request for service (interrupt). The processor then jumps from the program it was executing to the interrupt service routine. When the interrupt service routine is completed, control returns to the interrupted program.

Interrupt Mask Register that has one bit to control each interrupt. Used to selectively disable specific interrupts.

Interrupt Service Routine Program that is executed when an interrupt occurs.

Interrupt Vectoring Providing a device ID number or an actual branching address in response to the interrupt acknowledge signal. Allows each interrupt to automatically be serviced by a different routine.

Interval Timer Programmable device used to perform timing, counting, or delay functions. Usually treated as a peripheral.

I/O Mapped I/O I/O devices that are accessed by using instructions and control signals that differ from those of the memory devices in a system. Assigns I/O devices to a separate address space.

Iterative Procedure or process that repeatedly executes a series of operations until some condition is satisfied. Usually implemented by a loop in a program.

J

Jump Instruction that results in a change of sequence.

K

K Symbol for 1000 (10^3). When referring to bits or words, K=1024 (2^{10}).

Kernel Minimum circuitry required to allow the microprocessor to function. Usually consists of the microprocessor, clock circuit, interrupt and DMA control lines, and power supply.

Keyboard Group of push buttons used for inputting information to a system.

L

Label Name assigned to a memory location. When an assembly language program is written, a label is assigned to an instruction or memory location that must be referred to by another instruction. Then when the program is converted to machine code, an actual address is assigned to the label.

Large Scale Integration (LSI) Technology by which thousands of semiconductor devices are fabricated on a single chip.

Latch Hardware device that captures information and holds it (e.g., a group of flip-flops).

Least Significant Bit (LSB) Rightmost bit in a number, which has the least numerical weight.

LED Light Emitting Diode. Semiconductor device that emits light when current is passed through it.

LIFO Last-In-First-Out buffer. Same as *push-down stack*. See Stack.

Linear Select Decoding Address decoding technique that uses the most significant address bits to directly enable devices in the system.

Listener Device that inputs data from a data bus. An output port is a listener.

Logic Analyzer Test system capable of displaying 0's and 1's, as well as performing complex test functions. Logic analyzers typically have 16 to 32 input lines and can store sequences of sixteen or more bits on each of the input lines.

Logic Comparator Test product that compares pin-for-pin operation of an IC operating in-circuit with a known good reference IC.

Logic Probe Hand-held troubleshooting tool that detects logic state and activity on digital circuit nodes.

Logic Pulser Hand-held troubleshooting tool that injects controlled digital signals into logic nodes.

Loop Part of a program that is repeatedly executed.

Low-Order Pertaining to the weight or significance assigned to the digits of a number. In the number 123456, the lower order digit is six. The three low-order bits of the binary word 11100101 are 101.

LSB See Least Significant Bit.

LSI See Large Scale Integration.

LSTTL Low power Schottky TTL. Digital integrated circuits that employ Schottky transistors for improved speed/power performance over standard TTL.

M

Machine Code See Machine Language.

Machine Cycle Basic period of time required to manipulate data in a system.

Machine Language Binary language (often represented in hex) that is directly understood by the processor. All other programming languages must be translated into binary code before they can be entered into the processor.

Mantissa Fractional value used as part of a floating point number. For example, the mantissa in the number 0.9873×10^7 is 0.9873.

Mask Pattern used to selectively set certain bits of a word to 1 or 0. Usually ANDed or ORed with the data.

Mask Programmed An IC that is programmed by generating a unique photomask used in the fabrication of the IC.

Mass Storage Secondary, slower memory for large files. Usually floppy disk or cassette.

Medium Scale Integration (MSI) Technology by which a dozen or more gate functions are included on one chip.

Memory Part of a computer system into which information can be inserted and held for future use. Storage and memory are interchangeable terms. Digital memories accept and hold binary numbers only. Common memory types are core, disk, tape, and semiconductor (which includes ROM and RAM).

Memory Map Shows the address assignments for each device in the system.

Memory-Mapped I/O I/O devices that are accessed by using the same group of instructions and control signals used for the memory devices in a system. The memory and I/O devices share the same address space.

Microcode See Microprogram.

Microcomputer Complete system, including CPU, memory, and I/O interfaces.

Microprocessor Central processing unit fabricated on one or two chips. The processor consists of the arithmetic and logic unit, control block, and registers.

Microprogram Program that defines the instruction set. The microprogram (also called microcode) tells the CPU what to do to execute each machine language instruction. It is even more detailed than machine language and is not generally accessible to the user.

Mnemonic Code Codes designed to assist the human memory. The microprocessor language consists of binary words, which are a series of 0's and 1's, making it difficult for the programmer to remember the instructions corresponding to a given operation. To assist the human memory, the binary numbered codes are assigned groups of letters (or mnemonic symbols) that suggest the definition of the instruction. For example, the 8085 code 32 means load accumulator and is represented by the mnemonic LDA.

Modem Modulator-demodulator. Usually used to interface a digital device to a telephone line. Encodes and decodes serial bits into frequencies.

Monitor Program that controls the operation of a microcomputer system and allows user to run programs, examine and modify memory, etc.

MOS Metal Oxide Semiconductor. Integrated circuits made of field effect transistors. All MOS devices originally used metal gate technology, but the term is used to describe silicon gate circuits as well.

Most Significant Bit (MSB) Bit in the left-most position of a word, which has the greatest numerical weight.

Motherboard See Backplane.

MPU Microprocessing Unit. See Microprocessor.

MSB See Most Significant Bit.

MSI See Medium Scale Integration.

MTBF Mean Time Between Failures.

MTTR Mean Time to Repair.

Multiplexing Process of transmitting more than one signal via a single link. The most common technique used in microprocessor systems is time division multiplexing, in which one signal line is used for different information at different times.

N

Negative Logic The logic false state is represented by the more positive voltage in the system, and the logic true state is represented by the more negative voltage in the system. For TTL, 0 becomes +2.4 volts or greater, and 1 becomes +.4 volts or less.

Nested Subroutine that is called by another subroutine or a loop within a larger loop is said to be nested.

Node Any signal line connected to two or more circuit elements. All logic inputs and outputs electrically connected together are part of the same node.

Number Crunching Action of performing complex numerical operations.

O

Object Program End result of the source language program (assembly or high-level) after it has been translated into machine language.

Octal Base 8 number system. Often used to represent binary numbers, since each octal digit corresponds directly to three binary digits.

One's Complement Number representation system used for signed binary integers in which the negative of a number is obtained by complementing it. The left-most bit becomes the sign bit, with 0 for plus, 1 for minus.

Opcode See Operation Code.

Open-Loop Circuit operating without feedback.

Operation Code (Opcode) Segment of the machine-language instruction that specifies the operation to be performed. The other segments specify the data, address, or port. For the 8085, the first byte of each instruction is the opcode.

Output Port Circuit that allows the microprocessor system to output signals to other devices.

Overflow Results when an arithmetic operation generates a quantity beyond the capacity of the register. An overflow status bit (the carry) in the flag register is set if an operation causes an overflow.

P

Page Usually a block of 256 addresses. The lower eight bits of an address therefore specify the location within the page, while the upper eight bits specify the page.

Parameter Value passed from one routine to another, either in a register or a memory location.

Parity Number of 1's in a word, which may be even or odd. When parity is used, an extra bit is used to force the number of 1's in the word (including the parity bit) to be even (even parity) or odd (odd parity). Parity is one of the simplest error detection techniques and will detect a single-bit failure.

Patch Section of coding inserted into a routine to correct a mistake or alter the routine. It is usually not inserted into the actual sequence of the routine being corrected, but placed somewhere else. A jump to the patch and a return to the routine are then provided.

PC Printed Circuit or Program Counter.

PCB Printed Circuit Board.

Peripheral Any interface device connected to a computer. Also, a mass storage or communications device connected to a computer.

Polling One method used to identify the source of interrupt request. The CPU must poll (read) the devices to determine which one caused the interrupt.

Pop Operation of reading a word from the stack. Same as *pull*.

Port Point at which the I/O devices are connected to the computer.

Positive Logic True level is the more positive voltage level in the system.

Power-Up Reset Initialization process whereby storage elements within a system are preset to defined conditions whenever power is first applied.

Priority Number assigned to an event or device that determines the order in which it will receive service if more than one request is made simultaneously.

Processor Same as *microprocessor*.

Program Procedure for solving a problem, coded into a form suitable for use by a computer. Frequently referred to as *software*.

Program Counter (PC) Register in the CPU that holds the address of the next program byte to be read. Branching requires loading of the jump address into the program counter. Otherwise, the PC is incremented after each byte is read.

Programming Language Language used to write a program. May be machine, assembly, or high-level.

PROM Programmable Read-Only Memory. Integrated circuit memory that is manufactured with a pattern of all logical zero or ones and has a specific pattern written into it by a special hardware programmer.

Propagation Delay Time required for a signal to propagate through a device.

Protocol Set of rules for exchange of information.

Pseudo-Instruction Instruction that is used in an assembly language program but is an instruction for the assembler. Pseudo-instructions have no direct correspondence to machine language.

Pull-Up Resistor Used to provide the source current for open-collector and three-state logic gates or a termination for unused inputs. Pulls the voltage level up when no other device is driving the line.

Pulser See Logic Pulser.

Push Operation of adding a word to the stack.

Push-Down Stack See Stack.

R

Radix Total number of distinct characters or numbers used in a numbering system. Same as *base*.

RAM (Random Access Memory) Usually used to mean semiconductor read/write memory. Strictly speaking, ROMs are also RAMs (see also Random Access).

Random Access Access method in which each word can be retrieved in the same amount of time (i.e., the memory locations can be accessed in any [random] order).

Random Logic Hard-wired (or random) logic design solutions require interconnection of numerous integrated circuits representing the logic elements. The function of the circuit is determined by the functional blocks and their interconnections, rather than by a program.

Refresh Process of restoring the charge in a dynamic memory. Refresh logic must rewrite the contents of the complete RAM periodically (typically 2 ms), called refreshing the memory (see Dynamic Memory).

Register Single word of memory. Registers within the CPU are more readily accessible than external memory locations. Registers external to the CPU are simply a group of flip-flops.

Relative Addressing Specifying an address as a distance from the current address (e.g., three bytes ahead or four bytes backwards).

Restore To return a register or other computer word to its initial or preselected value.

Return Mechanism providing for a return in the usual sense. In particular, an instruction at the end of a subroutine that causes control to return to the proper point in the main routine.

Rising Edge Low-to-high logic transition.

ROM (Read-Only Memory) Permanently programmed memory. Mask-programmed ROMs are programmed by the chip manufacturer. PROMs (Programmable ROMs) can be programmed by the user. EPROMs (Erasable PROMs) can be erased with ultraviolet light.

S

Scanning Process of sequentially accessing individual signal lines in a group.

Schmitt Trigger Circuit with hysteresis used for input signals that are noisy or have slow transition times.

Scratchpad Memory containing intermediate data needed for final results.

Self-Test Test performed by a product on itself.

Sequential Logic Circuit arrangement in which the output state is determined by the previous state and the current inputs. (Compare with *combinational logic.*)

Serial Transmitting data bits one at a time over a single wire, instead of using one wire for each bit.

Set-Up Time Time that data must be stable prior to a write signal.

Shift To move the characters of a unit of information right or left. For a binary number, this is equivalent to multiplying or dividing by two for each shift.

Signature Four-digit value generated by a signature analyzer, which is used to characterize data activity present on a logic node during a specific period of time.

Signature Analysis Technique used to facilitate the troubleshooting of digital circuits. Nodes of the circuit, stimulated during a test mode, produce "signatures" as the result of the data compression process performed by the signature analyzer. When node signatures are compared to known good documented signatures, faulty nodes can be identified.

Signature Analyzer Instrument used to convert the long, complex serial data streams present on microprocessor system nodes into four-digit signatures.

Simulator Special program that simulates the logical operation of the microprocessor. It is designed to execute machine language programs on a machine other than the one for which the program is written. This allows programs for one microprocessor to be debugged on a system that uses another processor.

Single-Step Process of executing a program one instruction or machine cycle at a time.

Sink Current Current input capability of a device.

Small Scale Integration (SSI) Technology of less complexity than medium scale integration. Usually means less than ten gate functions in the IC.

Software See Programs.

Solder Bridge Glob of excess solder that shorts two conductors. A common problem on production PC boards.

Source Code Program written in other than machine language. May be assembly language or a high-level language.

Source Current Current output capability of a device.

SSI See Small Scale Integration.

Stack Block of successive memory locations that is accessible from one end on a last-in-first-out basis (LIFO). For most processors, the stack may be any block of successive locations in the read/write memory.

Stack Pointer Contains the address of the top of the stack. In general, the stack pointer is decremented immediately following the storage in the stack of each byte of information. Conversely, the stack pointer is incremented immediately before retrieving each byte of information from the stack.

Static Memory Memory devices that do not need clocks or refreshing.

Status Present condition of the device. Usually indicated by flag flip-flops or special registers. See Flag.

Storage See Memory.

Stress Testing Introducing mechanical, electrical, or thermal stress on electrical devices so as to modify their operation and allow intermittent problems to be observed.

Subroutine Self-contained portion of a program that performs a well-defined task. May be used at different places in the same program.

T

Table Collection of data in a form suitable for ready reference, frequently stored in sequential memory locations.

Table Look-Up Obtaining a value from a table of values stored in the computer.

Talker Device that outputs data to a data bus. A ROM is a talker.

Three-State Logic device whose output can be placed into a high-impedance (off) state, in addition to the usual high and low states. This feature allows more than one device output to be connected to the same logic node. Three-state operation is a fundamental requirement for devices used on microprocessor data buses. Same as tri-state (registered trademark).

Throughput Speed with which problems or segments of problems are performed. Throughput will vary from one application to another.

Time Domain Information that is a direct function of time. An oscilloscope displays information in the time domain.

Tracer See Current Tracer.

Troubleshoot To seek the cause of a malfunction or erroneous program behavior in order to remove the malfunction.

Troubleshooting Tree Flow diagram consisting of tests and measurements used to diagnose and locate faults in a product.

TTL Transistor Transistor Logic. Family of digital integrated circuits that have bipolar transistor inputs and outputs.

TTY Teletype.

Two's Complement Numbers Numbering system commonly used to represent both positive and negative numbers. The positive numbers in 2's complement representation are identical to the positive numbers in standard binary. However, the 2's complement representation of a negative number is the complement of the absolute binary value plus 1. Note that the eighth or most significant bit indicates the sign: 0 = plus, 1 = minus.

U

UART Universal Asynchronous Receiver Transmitter. A serial to parallel and parallel to serial converter.

μC Microcomputer.

Unidirectional Wire or group of wires in which data flows in only one direction. Each device connected to a unidirectional bus is either a transmitter, or a receiver, but not both.

μP Microprocessor.

V

Vector Interrupt See Interrupt Vectoring.

Very Large Scale Integration (VLSI) Technology by which hundreds of thousands of semiconductor devices are fabricated on a single chip.

Volatile Memory Memory devices whose stored data changes when power is removed. RAMs can be made to appear nonvolatile by providing them with back-up power sources.

VLSI See Very Large Scale Integration.

W

Walking-Ones Memory test pattern in which a single one bit is shifted through each location of a memory filled with 0's. A walking-zero pattern is the converse.

Word Set of characters that occupies one storage location and is treated by the computer circuits as a unit. Ordinarily a word is treated by the control unit as an instruction and by the arithmetic unit as a quantity.

Write To transfer information, usually from main storage, to an output device; to record data in a register, location, or other storage device.

BIBLIOGRAPHY

BIBLIOGRAPHY

BOOKS

Blakeslee, Thomas R. *Digital Design with Standard MSI and LSI,* 2nd Ed. Somerset, New Jersey: Wiley-Interscience, 1979. Describes digital design using standard logic and microprocessors.

Hilburn, John L., and Julich, Paul M. *Microcomputers/Microprocessors: Hardware, Software, and Applications.* Englewood Cliffs, New Jersey: Prentice-Hall, 1976. A general book on microcomputer principles.

Klingman, Edwin E. *Microprocessor Systems Design.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977. An advanced text on microprocessor systems design.

Lesea, Ausin, and Zaks, Rodnay. *Microprocessor Interfacing Techniques.* Berkeley: Sybex, Inc., 1977. Describes many common microprocessor peripherals and the circuits used to interface them to a microprocessor system.

Leventhal, Lance A. *8080A/8085 Assembly Language Programming.* Berkeley: Osborne & Associates, Inc., 1978. Describes the 8080/8085 instruction set with many example programs.

Osborne, Adam. *An Introduction to Microcomputers: Volume I—Basic Concepts.* Berkeley: Osborne & Associates, Inc., 1976. A general introductory text on microprocessor concepts.

Osborne, Adam. *An Introduction to Microcomputers: Volume II—Some Real Products.* Berkeley: Osborne & Associates, Inc., 1978. Describes many of the available microprocessors and support chips.

Peatman, John B. *Microcomputer-Based Design.* New York: McGraw-Hill, 1977. Describes design techniques for microprocessor-based instruments.

Yourdon, Edward. *Techniques of Program Structure and Design.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975. Presents techniques for writing well-structured programs.

Zaks, Rodnay. *Microprocessors from Chips to Systems.* Berkeley: Sybex Inc., 1977. An introduction to microprocessor systems with practical examples.

PERIODICALS

"Microcomputer Systems Reference Issue." *EDN.* November 20, 1978. (Repeated annually.) Describes all of the available microprocessors and their support chips.

"Microprocessor Data Manual." *Electronic Design.* October 11, 1978. (Repeated annually.) Describes all of the available microprocessors.

Beckwith, John F. "Current Tracer: A New Way to Find Low Impedance Logic-Circuit Faults." *Hewlett-Packard Journal,* December 1976, pp. 2–8.

Bronson, Barry, and Chan, Anthony Y. "A Multifunction, Multifamily Logic Pulser." *Hewlett-Packard Journal,* December 1976, pp. 12–15.

Chan, Anthony Y. "Easy-to-Use Signature Analyzer Accurately Troubleshoots Complex Logic Circuits." *Hewlett Packard Journal,* May 1977, pp. 9–14.

Gordon, Gary, and Nadig, Hans. "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems." *Electronics,* March 3, 1977, pp. 89–96.

Nadig, Hans J. "Signature Analysis—Concepts, Examples, and Guidelines." *Hewlett-Packard Journal,* May 1977, pp. 15–21.

Quenelle, Robert C. "New Logic Probe Troubleshoots Many Logic Families." *Hewlett-Packard Journal,* December 1976, pp. 9–11.

MANUFACTURERS' LITERATURE

MCS-85 Microcomputer Systems User's Manual. Santa Clara, California: Intel Corp., 1978. Hardware reference manual for the 8085.

MCS-80 Microcomputer Systems User's Manual. Santa Clara, California: Intel Corp., 1977. Hardware reference manual for the 8080.

8080/8085 Assembly Language Programming Manual. Santa Clara, California: Intel Corp., 1978. Software reference manual for the 8080 and 8085.

Z80 Microcomputer Data Book. Carrollton, Texas: Mostek Corp., 1978.

M6800 Microprocessor Applications Manual. Phoenix: Motorola Semiconductor Products, Inc., 1975. A description of 6800 hardware and a complete design example.

M6800 Programming Reference Manual. Phoenix: Motorola Semiconductor, 1976.

R6500 Microcomputer System Hardware Manual. Anaheim, California: Rockwell International, 1978.

3870/F8 Microcomputer Data Book. Carrollton, Texas: Mostek Corp., 1978.

TMS 1000 Series Data Manual. Houston: Texas Instruments, Inc., 1976.

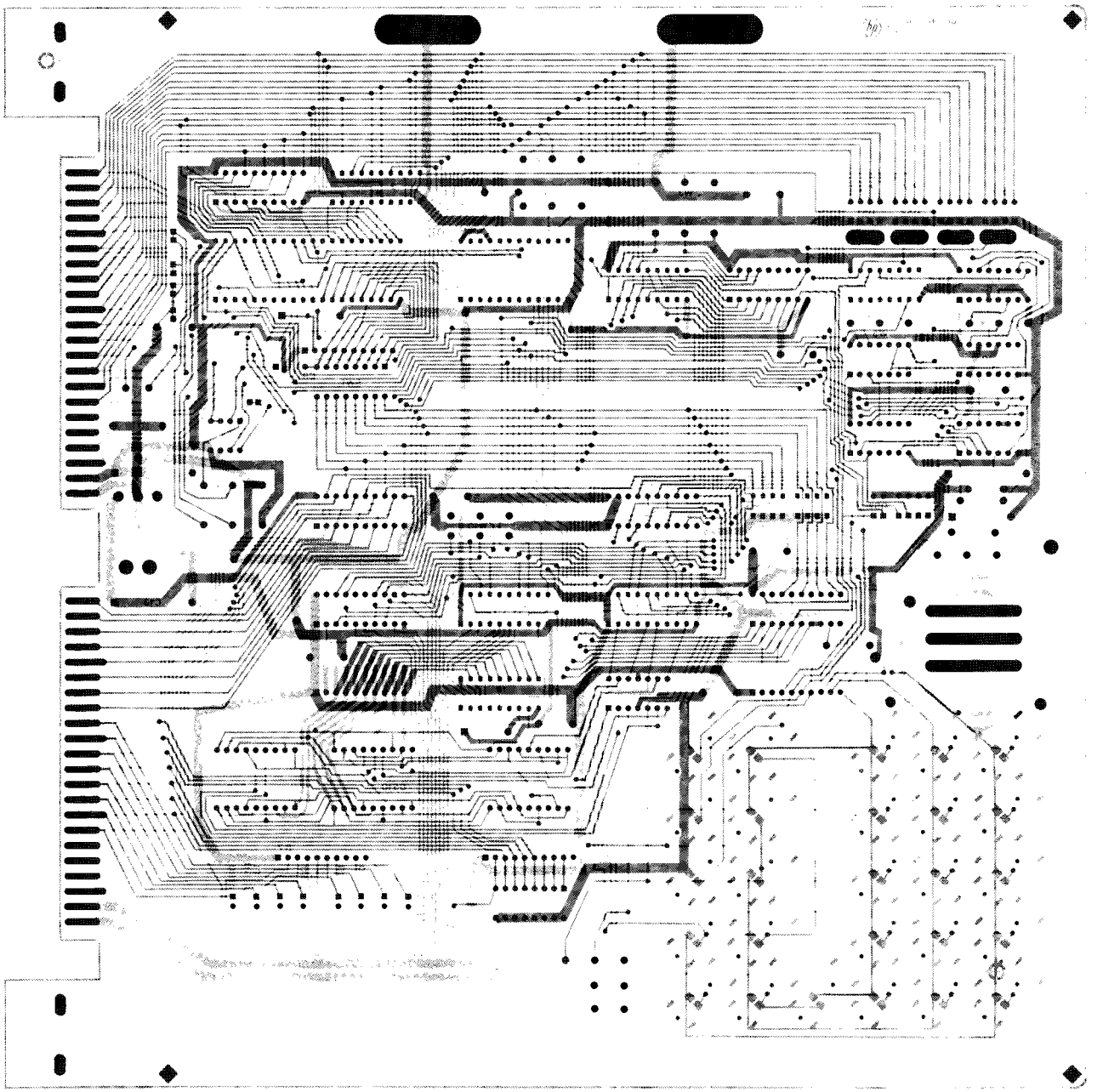
Am 2900 Bipolar Microprocessor Family Data Book. Sunnyvale, California: Advanced Micro Devices, Inc., 1976.

APPLICATION NOTES

A Designer's Guide to Signature Analysis. Hewlett-Packard Application Note 222, 1977.

Techniques of Digital Troubleshooting. Hewlett-Packard Application Note 163-1, 1973.

HARDWARE REFERENCE DIAGRAMS



5036A Board Trace Diagram

TABLE OF INTEGRATED CIRCUITS		
REFERENCE DESIGNATION	HP PART NUMBER	GENERIC PART NUMBER
U1, U13, U14	1820-1794	81LS95
U2	1820-1997	74LS374
U3	1820-2074	8085A
U4	1818-0773	1818-0773
U5, U6	1818-0438	2114
U7	1820-1216	74LS138
U8	1820-1195	74LS175
U9	1820-1197	74LS00
U10	1820-1112	74LS74
U11	1820-1208	74LS32
U12	1820-1416	74LS14
U15, U16, U17	1820-1730	74LS273
U18	1820-1759	87LS97
U19	1820-2138	8871
U20	1820-1231	75492

NOTES

1. REFERENCE DESIGNATIONS WITHIN THIS ASSEMBLY ARE ABBREVIATED. ADD ASSEMBLY NUMBER TO ABBREVIATION FOR COMPLETE DESCRIPTION.

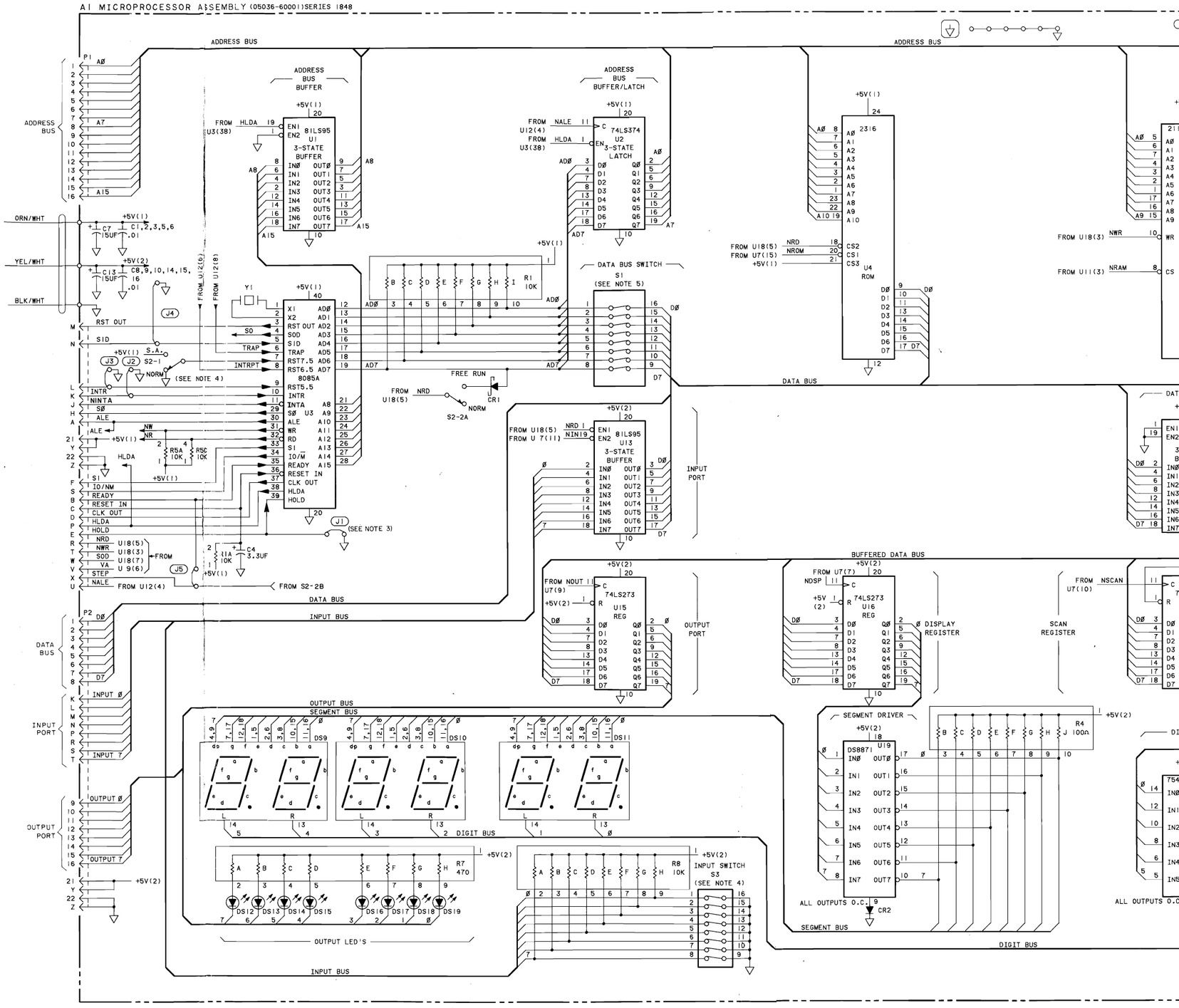
2. UNLESS OTHERWISE INDICATED:
   RESISTANCE IN OHMS;
   CAPACITANCE IN MICROFARADS

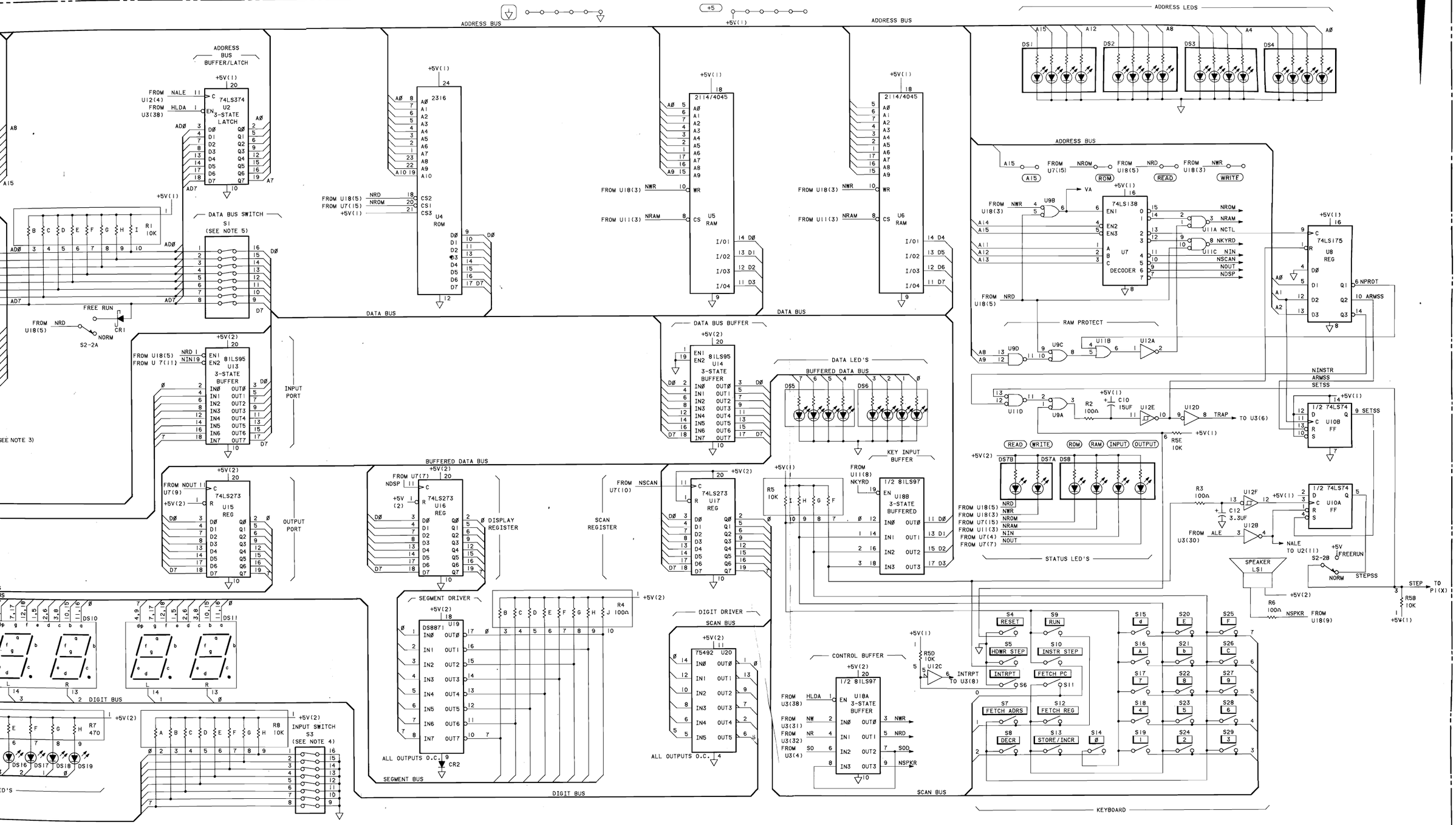
3. THE TRACES BETWEEN THE TERMINALS AT J1, J2, J3, AND J4 MUST BE CUT IF THESE LINES ARE TO BE USED WITH PERIPHERALS.

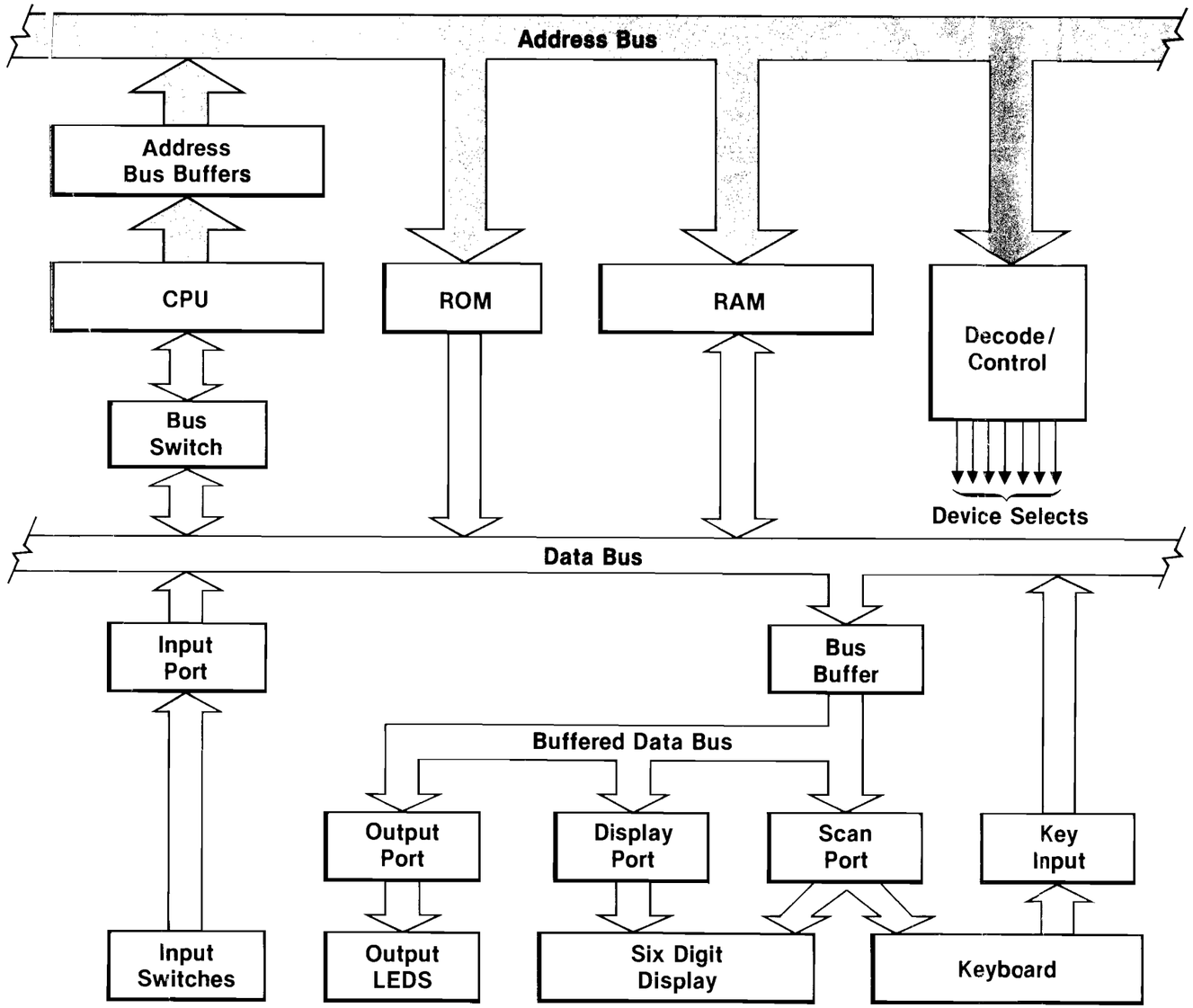
4. S3 SWITCHES ARE SHOWN IN CLOSED (LOGIC Ø) POSITION.

5. S1 SWITCHES ARE SHOWN IN CLOSED (NORM) POSITION. ALL SECTIONS OF S1 MUST BE OPEN FOR FREERUN MODE.

6. AN N DENOTES AN ACTIVE LOW SIGNAL LINE (E.G. NRAM=$\overline{\text{RAM}}$).




ADDRESS BUS
+5V(1)
ADDRESS BUS BUFFER/LATCH
74LS374 U2 3-STATE LATCH
DATA BUS SWITCH S1 (SEE NOTE 5)
FREE RUN
CR1
NORM
S2-2A
2316 U4 ROM
2114/4045 U5 RAM
2114/4045 U6 RAM
DATA BUS
ADDRESS LEDS
DS1
DS2
DS3
DS4
74LS138 U7 DECODER
74LS175 U8 REG
RAM PROTECT
U11A NCTL
U11C NIN
NSCAN
NOUT
NDSP
NROM
NRAM
NKYRD
NPROT
ARMSS
NINSTR
SETSS
TRAP
TO U3(6)
R2 100Ω
C10 15UF
R5E 10K
U12E
U12D
U10B FF
READ
WRITE
ROM
RAM
INPUT
OUTPUT
DS7B
DS7A
DS8
STATUS LED'S
R3 100Ω
C12 3.3UF
U12F
U12B
U10A FF
FROM ALE U3(30)
NALE TO U2(11)
SPEAKER LS1
S2-2B
FREERUN
STEPSS
STEP TO PI(X)
R5B 10K
R6 100Ω
NSPKR FROM U18(9)
INPUT PORT
74LS273 U15 REG
OUTPUT PORT
74LS273 U16 REG
DISPLAY REGISTER
74LS273 U17 REG
SCAN REGISTER
DATA BUS BUFFER
8ILS95 U14 3-STATE BUFFER
BUFFERED DATA BUS
DATA LED'S
KEY INPUT BUFFER
1/2 8ILS97 U18B 3-STATE BUFFERED
CONTROL BUFFER
1/2 8ILS97 U18A 3-STATE BUFFER
INTRPT TO U3(8)
R50 10K
U12C
S4 RESET
S5 HDWR STEP
S6 INTRPT
S7 FETCH ADRS
S8 DECR
S9 RUN
S10 INSTR STEP
S11 FETCH PC
S12 FETCH REG
S13 STORE/INCR
S14 Ø
S15 d
S16 A
S17 7
S18 4
S19 1
S20 E
S21 b
S22 8
S23 5
S24 2
S25 F
S26 C
S27 9
S28 6
S29 3
KEYBOARD
SEGMENT DRIVER
DS8871 U19
SEGMENT BUS
ALL OUTPUTS O.C.
CR2
DIGIT DRIVER
75492 U20
DIGIT BUS
SCAN BUS
R4 100Ω
R7 470
R8 10K
INPUT SWITCH S3 (SEE NOTE 4)
DS10
DS11
DS16
DS17
DS18
DS19
DIGIT BUS
SEE NOTE 3)



5036A Microprocessor Lab Block Diagram

5036A MICROPROCESSOR LAB SCHEMATIC

See Page 453

5036A MICROPROCESSOR LAB BLOCK DIAGRAM

See Page 454

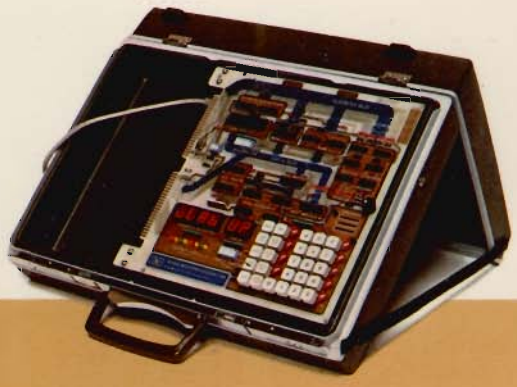
Error Messages

Hardware Errors: "IC4", "IC5", "IC6" — indicate a hardware problem with the displayed IC, the circuits used with that IC, or a misplaced fault jumper.

Software Errors: "SP Er" — stack pointer error message occurs when your program tries to run the power-up initialize sequence at address 0000. This error usually happens when more returns than calls are executed by the program. Program bugs of this nature are fairly common. Had the monitor program not detected this condition, the μLab would have cleared memory and lost the program.

BEEP after pushing STORE/INCR — means that the new data in the display failed to store in the address specified. This error usually results when trying to store data in a ROM location (0000-07FF) or at an address outside of the RAM (>0BFF). It also occurs when trying to store data in a defective RAM location.

BEEP after pushing RUN — means that your program is either executing a RST7 instruction or, more likely, not running in a loop. If the program does not end in a jump back to a point in the loop, it will continue executing instructions in memory (the 00 NOP instructions stored during power-up) until encountering the RST7 instruction at address 0AEF. At this point, the μLab beeps, sets the PC to 0800, and returns to the monitor program.



OPEN FOR KEY DESCRIPTIONS

Quick Reference Chart

RESET — Forces the μLab into the "waiting for command" mode. Used for exiting user programs. If pressed while in the hardware single-step mode, the address of the next instruction is displayed. Does not affect the contents of the memory or the user registers (see page 17).

HDWR STEP — Sets the μLab to the hardware single-step mode. Once in this mode, one byte of information is passed between the microprocessor and the rest of the system each time HDWR STEP is pressed. With the system stopped, the instruction and data byte transfers can then be observed directly on the μLab's binary bus and status LEDs (see page 56).

INTRPT — Interrupts the μLab (if this interrupt has been enabled). Used primarily for demonstrating the use of interrupts (see pages 77 and 80).

FETCH ADRS — Specifies that you want to enter an address. The μLab then expects a four digit address to be entered. Used for examining memory and for specifying the starting location of a program (see page 18).

DECR — Decrements the displayed address. Used to examine preceding memory locations. Does not affect the contents of the memory. Also used for examining registers (see page 52).

RUN — Causes the program starting at the displayed address to begin execution. The standard way of starting a program (see page 57).

INSTR STEP — Causes the displayed instruction to be executed and the next instruction to be displayed. Used for analyzing and debugging program flow (see page 55).

FETCH P C — Sets the displayed address to the last value of the user's program counter. Most often used for returning to a program after a breakpoint. Also used when single-stepping to return to a program after examining registers or memory (see pages 66 and 70).

FETCH REG — Sets the μLab to the "register" mode. Once in this mode, the registers can be examined and changed by using the STORE/INCR and DECR keys (see pages 66, 70, and 171).

STORE /INCR — Stores the displayed data at the displayed address and then increments the address. Used for entering data and examining successive memory locations. Also used for examining and modifying registers (see page 51).

0 — F — Used for entering hexadecimal addresses and data (see page 51).

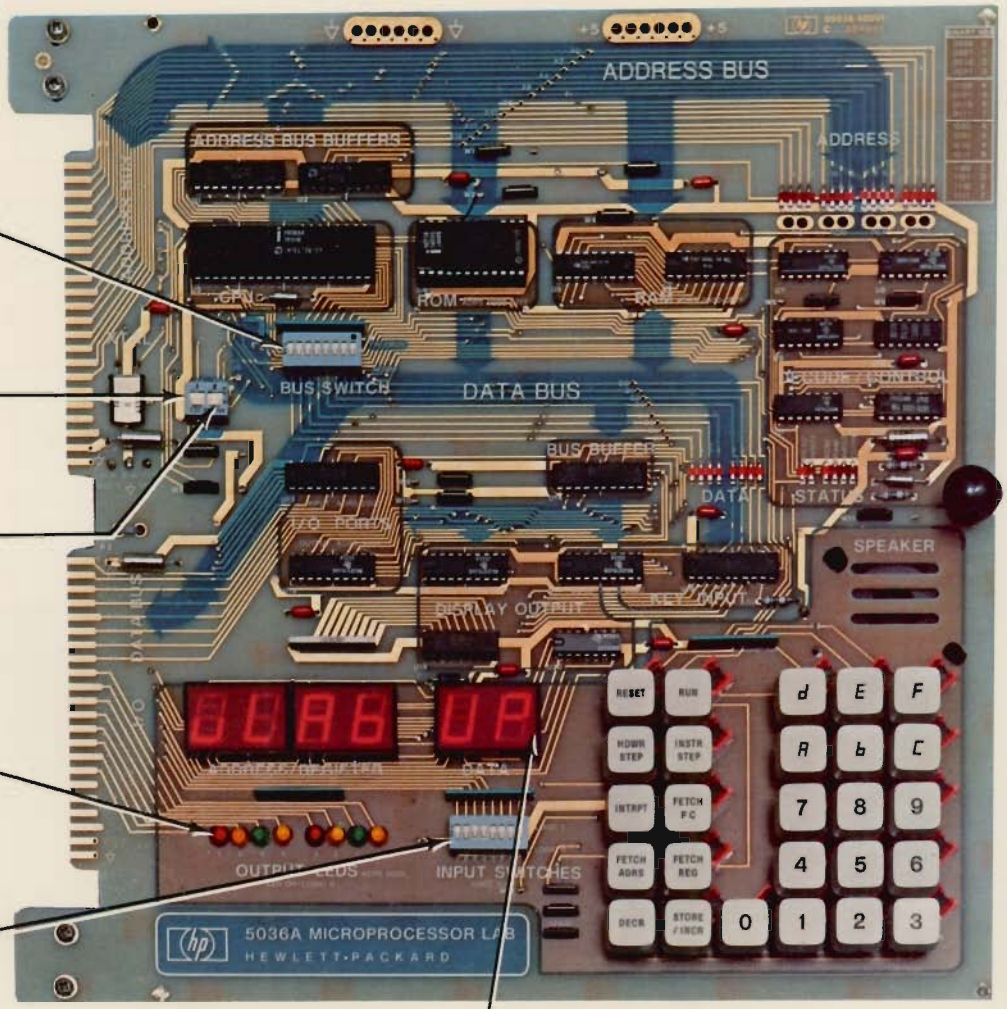
Data bus switches. All in up position for freerun mode, all in down position at all other times (see page 274).

S.A. switch. Move up and then down to initiate Signature Analysis test mode (see page 283).

Freerun switch. Up for freerun mode (see page 274).

Output LEDs, address 3000 (see page 59).

Input switches, address 2000 (see page 59).



Note: If the decimal point is lit, the μLab is in data entering mode and RUN, INSTR STEP, and HDWR STEP are disabled (see page 57).

05036–90003

PRINTED IN U.S.A.