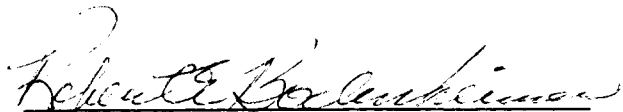
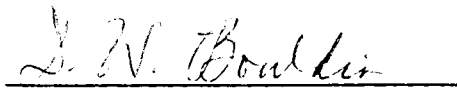


To the Graduate Council:

I am submitting herewith a thesis written by Seiko Harimoto entitled "An Advanced Microprocessor-Based System Design Course at UTK." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.


Robert E. Bodenheimer
Major Professor

We have read this thesis and
recommend its acceptance:




Accepted for the Council:


The Graduate School

AN ADVANCED MICROPROCESSOR-BASED
SYSTEM DESIGN COURSE AT UTK

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Seiko Harimoto
December 1983

ACKNOWLEDGMENTS

The author wishes to express his gratitude to Dr. Robert E. Bodenheimer for his suggestion on the thesis topic and guidance throughout the thesis preparation. His constant encouragement will always be remembered.

The author would also like to express his appreciation to Mr. Bobby Bodenheimer for proofreading the manuscript. Thanks are extended to Dr. D. W. Bouldin and Dr. R. W. Rochelle for serving on the committee and providing helpful comments on the thesis. The author would like to thank Mr. James Barnes, Mr. Lendon Adkins, and Mr. Jackson Watson in Technical Services for their help in providing and ordering parts for the project.

Finally, he would like to thank Dr. J. M. Googe and Dr. Walter L. Green for granting him a teaching assistantship that made it possible for him to complete the program. To his parents, brother, and sister, he is indebted for their unfailing support throughout his Master's program.

ABSTRACT

A microprocessor-based system design course at the chip level is needed to complement a presently available microcomputer interface course based on the TM 990/U89 microcomputer board. Projects in the microprocessor-based system are needed for the course. Using the TI 990/12 model 20 minicomputer facility and the AMPL (Advanced Microprocessor Prototype Laboratory) development system, six design projects are implemented and recommended for the course. Two of these projects are designed to familiarize the students with the AMPL development system. Each of the other four are microprocessor-based system projects that can be completed in a quarter's time. One of the microprocessor-based system projects is used as an example in the thesis to illustrate the design procedures using the AMPL development system. These six projects should provide an interesting laboratory for the microprocessor-based system design course.

TABLE OF CONTENTS

CHAPTER	PAGE
I. INTRODUCTION	1
II. MICROPROCESSOR EDUCATION	3
Microprocessor Education in U. S.	3
Future of Microprocessor Education	5
III. CURRENT MICROPROCESSOR EDUCATION AT UTK	7
Microcomputer Facility at UTK	7
Microprocessor Course Description	9
IV. AMPL DEVELOPMENT LABORATORY	13
Development System Concept	13
AMPL Development Laboratory Hardware Configuration . . .	19
AMPL Development Laboratory Command Structure	22
V. ADVANCED APPLICATIONS OF THE AMPL DEVELOPMENT LABORATORY	45
Six Different Operational Modes of the Emulator and the Trace Modules	45
Automated Memory Test Using AMPL	70
Real-Time Software Simulation	73
VI. MICROPROCESSOR-BASED SYSTEM DESIGN EXAMPLE	78
Design Process Overview	78
Project Definition	80
Front Panel Design	81
Hardware Block Diagram and System Flow Chart	82
Major Component Selections	89
Design and Test System Clock and Power-Up Circuit . . .	93
Design and Test Prototype Memory	95
System Interfacing Devices Design and Test	102
I/O Devices and Interfacing Design and Test	108
System Software Design	112
Software and Hardware Integration Phase	114
VII. CONCLUSION	120
LIST OF REFERENCES	122

PAGE

APPENDIXES	125
A. EMULATOR AND TRACE MODULE PRIMITIVE COMMAND SYNTAX	126
B. MICROPROCESSOR-BASED SYSTEM DESIGN PROJECT SPECIFICATIONS	128
C. MICROPROCESSOR-BASED SYSTEM DESIGN PROJECT SOLUTIONS . . .	138
VITA	155

LIST OF TABLES

TABLE	PAGE
3.1. Microprocessor Course Schedule at UTK	10
5.1. Emulator and Trace Module Operational Modes	46
5.2. Emulator Trace Buffer Content	47
6.1. Boundary Specifications of the Analyzer	81
6.2. Functions of the Front Panel Devices	84
6.3. Major Component Selections for the Analyzer	92
6.4. Order of Hardware Modules to be Designed	93
6.5. Loop Count Modifications	116

LIST OF FIGURES

FIGURE	PAGE
4.1. In-Circuit Emulation Concept	15
4.2. AMPL Development Laboratory Hardware Configuration . . .	20
4.3. Command Structure	23
4.4. Interactive Expression Evaluation	28
4.5. Memory Display and Modify	30
4.6. Help Command	39
4.7. Timing Diagram Display	41
5.1. Emulator Stand-Alone Mode	48
5.2. Trace Module Stand-Alone Mode	52
5.3. Emulator Control/Internal Data Interlocked Mode	58
5.4. Emulator Control/Internal Data Independent Mode	62
5.5. Emulator Control/External Data Interlocked Mode	64
5.6. Emulator Control/External Data Independent Mode	67
5.7. Microprocessor-Based System Debugging Process	69
5.8. Real-Time Simulation by Substitution	76
6.1. Microprocessor-Based System Design Phases	79
6.2. Histogram Analyzer Front Panel	83
6.3. Histogram Analyzer Block Diagram	85
6.4. General System Flow Chart	87
6.5. External Clock Circuit	94
6.6. Power-Up and Reset Circuit	94
6.7. TMS 9980 Memory Write Timing	97
6.8. TMS 9980 Memory Read Timing	98

FIGURE	PAGE
6.9. Memory Address Decoder	99
6.10. Memory Read/Write Timing Test	100
6.11. I/O and Interrupt Design	104
6.12. I/O Diagnostic Routine for the TMS 9901	105
6.13. Debounce Circuit Test	106
6.14. Interrupts and I/O Ports Diagnostic Routine	109
6.15. D/A, A/D, and Scope Interfacing Design	110
6.16. A/D and D/A Diagnostic Routines	113
6.17. Histogram Analyzer Results	119

CHAPTER I

INTRODUCTION

The microprocessor is a relatively new digital device which has revolutionized modern digital system design techniques. Since the introduction of the first 4-bit microprocessor, the 4004, by the Intel Corporation thirteen years ago, hundreds of varieties of microprocessors have appeared on the market ranging from 4-bits to 32-bits. The reasons for the rapid development are based in miniaturization and simplicity of usage. Since microprocessors are miniature computer CPUs, they have made computer technology available in many different areas of applications. Some of the applications included point of sale terminals, traffic controllers, intelligent terminals, and automobile fuel injection controllers. No math or physics background is required to use microprocessors. Design concepts are relatively simple and can be learned easily by hobbyists and engineers alike, in all fields of endeavor. Indeed the microprocessors have arrived like a tidal wave and changed the lives of many as one microprocessor educator has described [1].

To keep pace with this rapidly expanding field, an advanced microprocessor-based system design course at UTK is required. This course will be a continuation of a presently taught microcomputer course. The aim of this thesis is to examine the trend of microprocessor education, to develop hands-on microprocessor-based system design projects and to investigate how an AMPL development system can

be used by the students in their projects. The goal of this course is to teach students microprocessor-based system design at the chip level using an in-circuit emulator. This skill will increase their technical confidence and will enable them to design digital systems using the microprocessors like any other ICs with which they are familiar.

CHAPTER II

MICROPROCESSOR EDUCATION

Microprocessor Education in U. S.

The educational demand caused by the rapid development in the microprocessor is carried out in direct and indirect approaches. An indirect approach educates the general public through product advertisement and product usage. The direct approach, teaching through lectures and laboratories, is carried out by three main educational groups: schools, industry, and hobbyist organizations.

Because of the low price of today's personal computers, hobbyist organizations are proliferating in the country. These organizations provide substantial microprocessor education through exchange of information between hobbyists as well as providing basic microprocessor courses to a variety of age groups. Most of these are oriented toward high-level programming languages. A large number of these hobbyist are high school students who will eventually continue their interests in microcomputers when they enter colleges. The development of these hobbyist organizations and the courses that they offer is in its infancy. In the near future these courses will increase in number and sophistication. They will become one of the fundamental microprocessor educational tools for both the general public and the young computer scientists [1].

Another major driving force behind the microprocessor education is that of industry which realizes that many future products will be

microprocessor-based. Two major strategies are employed by industries to carry out the educational objectives. One of them is through an in-house learning center as employed by Texas Instruments. In this approach, the industry develops its own books, laboratories, video-tapes, and course lay-out. The other educational strategy is to develop its own course and hire consultants to teach the course. Intel employs this strategy. In either strategy, industry aims its microprocessor education at working engineers and the courses deal only with its own microprocessor-based products [1].

Finally, a majority of microprocessor education is carried out in colleges and universities. Initially, the microprocessor courses were taught in graduate classes in electrical engineering departments. Hands-on laboratories were few because microprocessor-based systems were expensive. As microprocessor development rapidly progressed and the products became cheaper, the universities began to incorporate microprocessor courses in the undergraduate curriculum. Some electrical engineering departments simply added the courses to the list of electives. Others required the courses of all students, while others required the courses of students in the computer options only. Still others modified existing classes in digital logic to incorporate microprocessor education.

In 1977 a survey showed most of the courses offered were still at the introductory level. A single microprocessor was taught in most electrical engineering departments. The most widely used processor was the 8080, followed by the 6800 and the 6502. Software and hardware

were equally emphasized. Hands-on laboratory experiences were widely practiced. Most of the projects for the courses were inspired by industrial problems. Availability of microprocessor systems to the students increased at a rapid rate. Each year the number of students taking the courses increased at a rate of 40% to 80%. Nevertheless, few universities had timesharing facilities for students to edit and assemble programs [2].

A survey three years later shows that over 95% of all electrical engineering departments, approximately 250 of them, offered microprocessor courses. Among the departments that offered microprocessor courses only about 3% lacked a hands-on laboratory. However, all of these departments expected to offer laboratory experiences in the near future. Eighty-five percent of the departments indicated that a majority of their students had a working knowledge of microprocessors. On the average, two courses had significant content on the subject of microprocessors. Each department offered an average of 50.4 fifty-minute lectures and 17.4 three-hour laboratory sessions. Approximately 50% of all the departments were expected to increase either undergraduate or graduate microprocessor lecture and laboratory hours [3].

Future of Microprocessor Education

Even though much change has occurred in the past few years in microprocessor education, the changes are expected to continue in the future at a slower rate. One of these changes may be courses in various aspects of microprocessor-based system and interface design. Another change may be better microprocessor development systems

available to colleges and universities. Courses in microprocessor-based system design at the component level will become more common. As the microprocessor industry reaches maturity, the courses and the textbooks will be standardized. The support from industry will be prevalent among all colleges and universities [3].

Another major change that is expected as the density of the integrated circuit reaches limits is the shift toward multiprocessor design. Courses in this area must be developed. Many colleges and universities already have included the topic of multiprocessor systems in their microprocessor courses. However, few have a hands-on laboratory in the design of multiprocessor systems. Hands-on experience in this area will increase as the multiprocessor-based system design technique reaches maturity and the development system can be purchased at a reasonable price.

Regardless of the future development in microprocessor, industry support will remain critical to the success of microprocessor education. Industries must remember that colleges and universities are providing the foundation for the microprocessor engineers and scientists of the future.

CHAPTER III

CURRENT MICROPROCESSOR EDUCATION AT UTK

Microcomputer Facility at UTK

The microprocessor course at UTK is organized around the use of microcomputer board. Only one logic design course in combinational and sequential circuits is a prerequisite to the microprocessor course. This course has used several different microcomputers. Initially the 6502 microcomputer board was used. There were only 4 units available in the laboratory for students to perform their assigned projects. Later the Motorola 6800 microcomputer board was used. The number of the microcomputer boards was also limited, so that the students had to do all their projects in the laboratory.

In the summer of 1981 Texas Instruments made a generous donation of 20 TMS 990/U89 microcomputer boards to UTK. With this large number of microcomputer boards available, students are able to check-out the microcomputer boards at the beginning of each quarter and return them at the end of the quarter. This allows students to have immediate hands-on experience with the microcomputer to experiment with what they have learned in the class lecture as well as providing a more flexible work schedule.

In addition to these microcomputer boards, a central time-sharing facility based around the TI990/12 Model 20 minicomputer became available. The minicomputer which supports multi-user and multi-task operations has 512-kilobytes of main memory and two disk

drives each with 80-megabyte. Eleven video display terminals, a 150 cps printer, and a 300 lines-per-minute printer provide the I/O interface to the students for program development. An AMPL development system with in-circuit emulation capability supports microprocessor-based system development. In addition the AMPL allows an assembled program to be loaded into a cassette for loading into the students' microcomputer boards. The detailed instructions on the use of the time-sharing facility are not covered in the lecture, but they are contained in three tutorials handed out in the class. Other tutorials on high-level language program development and the AMPL are available in the system for printout if the students are interested. The laboratory instructor is given the responsibility of operating and helping students in using the facility as well as checking-off completed projects.

The central time-sharing facility has greatly enhanced the productivity of the students. The students no longer need to waste time assembling their programs by hand or by using the assembler on the kit. More time can be spent on learning to write and debug an assembly language program. Program debugging is accomplished through the monitor program in the kit. Some of the debugging features and techniques are setting breakpoints, examining and modifying memory and registers, single stepping, and direct reading and writing to the computer I/O ports. When students have encountered difficulties in debugging their programs, the laboratory assistant can help them locate the problem by using the AMPL development system. This allows

the students to compare the debugging features of the kit and the AMPL. The AMPL development system is not fully utilized in the first course, but will be utilized extensively in the next microprocessor course.

Microprocessor Course Description

The present microprocessor course is project-oriented. A total of ten projects are available for the course, five of which are chosen by the instructor for implementation. These five projects represent 85% of the total grade. The flow charts and strategy of implementation of the other five projects are required but not the physical implementation. Depending upon the instructor, quizzes may or may not be given. More sophisticated aspects of the course such as interfacing A/Ds, D/As, and USART that are not covered in the projects are given as homework. Homework, quizzes, and flow charts organized into an engineering notebook represent the other 15% of the course grade.

The course is offered as both the undergraduate and graduate levels. The undergraduate course meets two times a week for 10 weeks a quarter. Each class session is 75 minutes. The graduate course meets one time each week for 2.5 hours. A typical undergraduate microprocessor course schedule is shown in Table 3.1. On the first day of the class, the course schedule, project specifications, homework assignments, and course grading policy are handed out. Projects are assigned early so that students have time to think ahead on choosing and developing algorithms.

Projects chosen for the course vary from one quarter to another. Only five typical projects will be discussed. A typical first project

Table 3.1. Microprocessor course schedule at UTK.

Session No.	Date	Activity	Comment
1	6/16/R	Introduction to Course	
2	6/21/T	Programming Model, Instruction Set, Addressing Modes	
3	6/23/R	Memory Map, I/O Initialization Switch to LED Example	
4	6/28/T	Programming Examples Software Delays, Debouncing Switches	
5	6/30/R	Interfacing Loudspeaker Example	
6	7/5/T	Interrupts Context Switching	
7	7/7/R	Linear Address Selection	
8	7/12/T	Software Subroutines	Project 11
9	7/14/R	SMA	
10	7/19/T	Priority Interrupt Encoding	Project
11	7/21/R	Homework	
12	7/26/T	Homework	Project
13	7/28/R	Peripheral Support Devices	
14	8/2/T	Microprocessor-based System Development	
15	8/4/R	Editing on a Development System	
16	8/9/T	Debugging on a Development System	Project
17	8/11/R	Real-Time Applications	
18	8/16/T	Signature Analysis	
19/20	8/18/R	Survey of Other Microprocessors	Project Engineering Notebook

is binary to BCD conversion, which is designed to familiarize the students with assembly language, I/O programming techniques, and the use of an on-board display unit. Eight toggle switches representing an 8-bit binary number is input and converted into BCD. Both the HEX and BCD conversions are displayed on the computer board.

A second project is typically a combinational and sequential circuit tester. This project reinforces the skills acquired in the first project. A combinational and a sequential circuit designed and constructed in a previous logic design class are used. Combinational logic such as signed-magnitude to two's complement is tested by the microcomputer. Outputs from the circuit are compared in a table to the expected values. If any output from the circuit is incorrect, the microcomputer displays BAD, otherwise GOOD. A sequential circuit such as an octal up-down counter is tested in a similar manner.

A third project is the stopwatch which teaches the use of the on-board interval timer and the interrupt handling procedures of the microcomputer. Two push buttons are required, one for start and stop and the other for freeze and reset. In this project the students also learn push button debouncing techniques.

The fourth project is a musical one. Using the microcomputer the students are asked to play a musical scale as well as a song of their choice. The tempo of the song should be easily changed. The specification also requires that provision be made to enter another composition easily during the project check-off. In addition an interface circuit for the external speaker is required. Both the

interval timer and the software approach are allowed. This is a fun project. Being able to program a microcomputer to play a song gives the student a feeling of accomplishment.

The fifth project, a ping-pong game, is a great challenge. Only a few students usually complete all aspects of the project. Several external devices must be interfaced to the microcomputer. These devices include 8 LEDs to indicate the position of the ball, two push buttons as paddles for two players, four rocker switches for speed setting and an external speaker for fanfare and a winning song for each player. Constant display of the score on the computer board is required. Being able to coordinate all these activities in a program is a complex task. Using software and hardware trade-off design techniques, the complexity of the software can be greatly reduced if an external clock and a counter are used to advance the ball. Overall this project serves nicely to conclude a first course in the design of microprocessor-based systems.

CHAPTER IV

AMPL DEVELOPMENT LABORATORY

Development System Concept

For a few years after microprocessors appeared on the market, microprocessor-based system engineers had only a few crude sets of tools for developing their system. These tools included oscilloscopes, digital probes, voltmeters, logic-timing analyzers, logic-state analyzers, and software simulators. Because these tools provided an inadequate means for controlling and monitoring the activities of a microprocessor-based system, microprocessor system design was a long, tedious, and complicated process. At that time a logic analyzer was the only tool capable of capturing a long string of logic states from a system under test.

Then the microprocessor analyzer appeared on the market. This analyzer was functionally the same as a logic analyzer except that it had the added capability of displaying captured data in mnemonic form so that the data could be easily interpreted by a designer. This capability alleviated the tedious aspect of design work. However, the microprocessor analyzer was still a passive debugging device lacking the capability to provide a needed stimulus and control for a microprocessor-based system [6,7]. With only these tools, software and hardware development was difficult not to mention the almost impossible task of software/hardware integration.

About eight years ago, the concept of in-circuit emulation was conceived. This concept as shown in Figure 4.1 involves replacing the microprocessor in the system under development, i.e., the prototype system or target system, with a mimic of the microprocessor that can be easily controlled externally [8]. This provided total control and monitoring capability of the microprocessor-based system.

An emulator can be implemented by random logic or a microprocessor-substituting approach. The ideal emulator is pin-to-pin and functionally compatible to that of the substituted microprocessor. Because of this, the random logic approach is more difficult to achieve [9]. The other approach is to replace the prototype microprocessor with the same microprocessor that has control logic built around it. The control logic provides an external means for single stepping, setting breakpoints, memory and register examination and modification, etc.

In addition, an emulator provides program tracing capability. Emulators are often hardwired to trace the address bus, data bus, and certain memory and I/O control lines of the microprocessor. Hardwired tracing eliminates the need to set up test points, a process which is tedious and prone to errors. However, hardwired tracing has the disadvantage of not being able to monitor test points other than those mentioned above. Therefore a logic analyzer is still an indispensable piece of equipment in a microprocessor development laboratory. Some manufacturers realize this and integrate logic analyzers into their emulators as a complete development system package. The AMPL

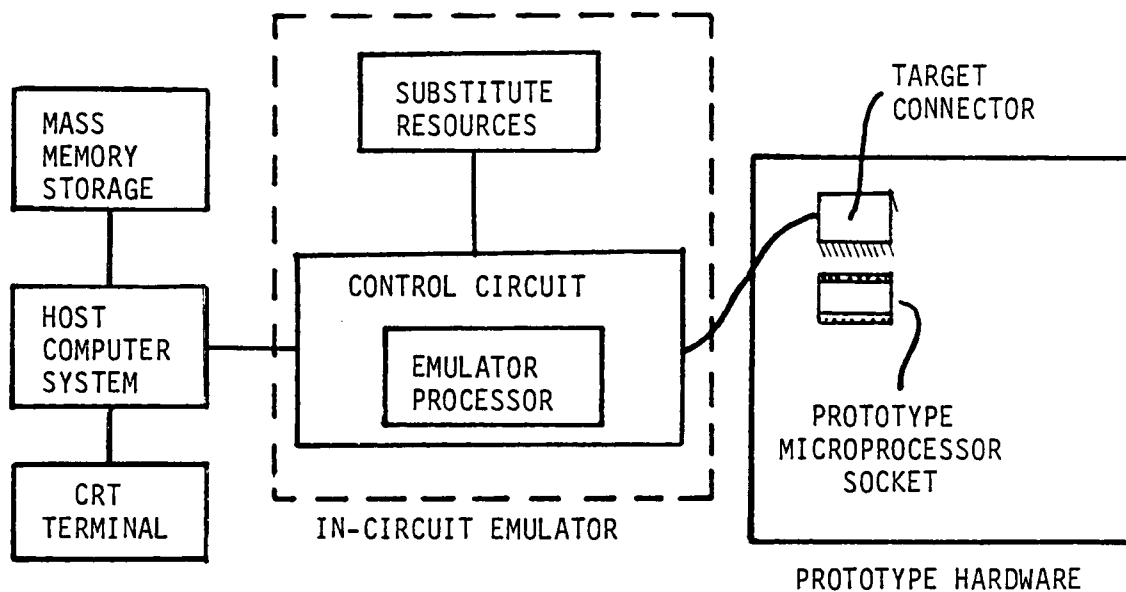


Figure 4.1. In-circuit emulation concept.

development system offered by the Texas Instrument Corporation provides such an integrated logic analyzer, to be discussed later.

In addition to microprocessor substitution, an emulator often provides other resource substitutions such as memory, I/O ports, and a system clock. This is also shown in Figure 4.1. Resource substitutions provide several functions to enhance the development process. For the first time, microprocessor-based system design does not need to begin with memory and I/O design based on estimates, which often require modifications during the design process.

The substitute memory on the emulator can be used as though it resides in the prototype system. To simulate the final memory configuration, the user can designate certain segments of the emulator memory to be write-protected in order to simulate ROM, while leaving other segments as RAM. This memory-mapping capability greatly reduces program modification time as compared to the old design method where a program resides in the EPROM of the prototype system during a debugging process. Clock substitution allows a designer to select the emulator clock for use by the prototype hardware. This allows system clock design to be carried out concurrently with other prototype hardware design, which reduces the development time. I/O substitution provides similar advantages. The user can easily add or modify I/O ports during a development process. I/O design can then be postponed until the final design stage [10].

Resource substitution also provides a unique method for isolating faults. Since substitute resources are known to work

properly, any fault which occurs can be assumed to come from software or hardware outside these substitute resources. When faults are corrected, prototype hardware replaces substitute resources. If any fault occurs after the switch, then the fault is isolated to the replaced prototype hardware. This fault isolation technique simplifies the debugging task.

The emulator, substitute resources, and trace module constitute three of five essential components of a complete development system. Two other essential components are the program development and EPROM programming facilities. The program development facility is made up of a file management utility, a text editor, an assembler, a compiler, a linker, and a loader. These program development tools are the "work horses" behind every development project. Therefore their capabilities and adaptability are essential for a smooth and efficient development process.

Since the program development tools are often available through existing minicomputer and microcomputer systems, an emulator is often integrated into an existing computer as a peripheral device. The user is interfaced to an emulator through the existing operating system and I/O devices such as video displays and TTY terminals. This type of development laboratory is very economical, because the existing facility is better utilized and less additional training is required. In addition, a laboratory of this type is usually more flexible because two separate processors are utilized. The host processor has the role of managing mass storage devices (such as disks

and tapes), controlling video display terminals, and setting up the emulator. The emulator processor's sole task is to mimic the prototype microprocessor as directed by the user through the host processor. This dedicated function allows real-time emulation.

A microprocessor development laboratory may be classified into a universal or a dedicated development laboratory, according to the expandability of the system to other microprocessor families. A universal development laboratory can support various microprocessors of different manufacturers by simply exchanging one emulator for another with the proper microprocessor in it. A universal development laboratory is often designed to support multi-users. This means that several microprocessor products can be developed concurrently by adding more emulator work stations to the host system. Another advantage of a universal development laboratory is that it will not become obsolete when new microprocessors are introduced into the market.

On the other hand, a dedicated development laboratory can only support one microprocessor or one family of microprocessors. A dedicated development laboratory may or may not support multi-users. Initially a universal development laboratory costs more than a dedicated development laboratory, but on a per station basis a universal development laboratory is a better choice for a long term product development effort [11].

The last essential tool for a complete microprocessor development laboratory is a EPROM programmer. Memory substitution

eliminates the need of a EPROM programmer until the completion of a project, at which time the debugged application program object code is loaded into the EPROM programmer through I/O interfaces. The EPROM programmer then programs the object code into EPROM devices. After the EPROM devices are installed in the prototype system, the emulator can be disconnected from the prototype hardware for stand-alone operation. The EPROM programmer also serves as a tool for program modification after the project is completed.

AMPL Development Laboratory Hardware Configuration

The AMPL development laboratory to be used in the microprocessor-based system design course is a dedicated multi-user minicomputer-based development system. Figure 4.2 provides an overview of the AMPL development laboratory hardware configuration. The development laboratory is composed of the five major hardware components discussed earlier: A time-sharing host system, an emulator module, a trace module, substitute memory, and a EPROM programmer. These five functional components are integrated to provide an extremely flexible and easy to use development system. The emulator board, memory expansion board, trace module, and EPROM programmer each plug into a standard CRU slot of the host TI 990 computer. The AMPL supports multi-development stations so that several different microprocessor-based systems can be developed concurrently. For each additional station, an emulator and a buffer board are required. A memory expansion board and trace module are optional.

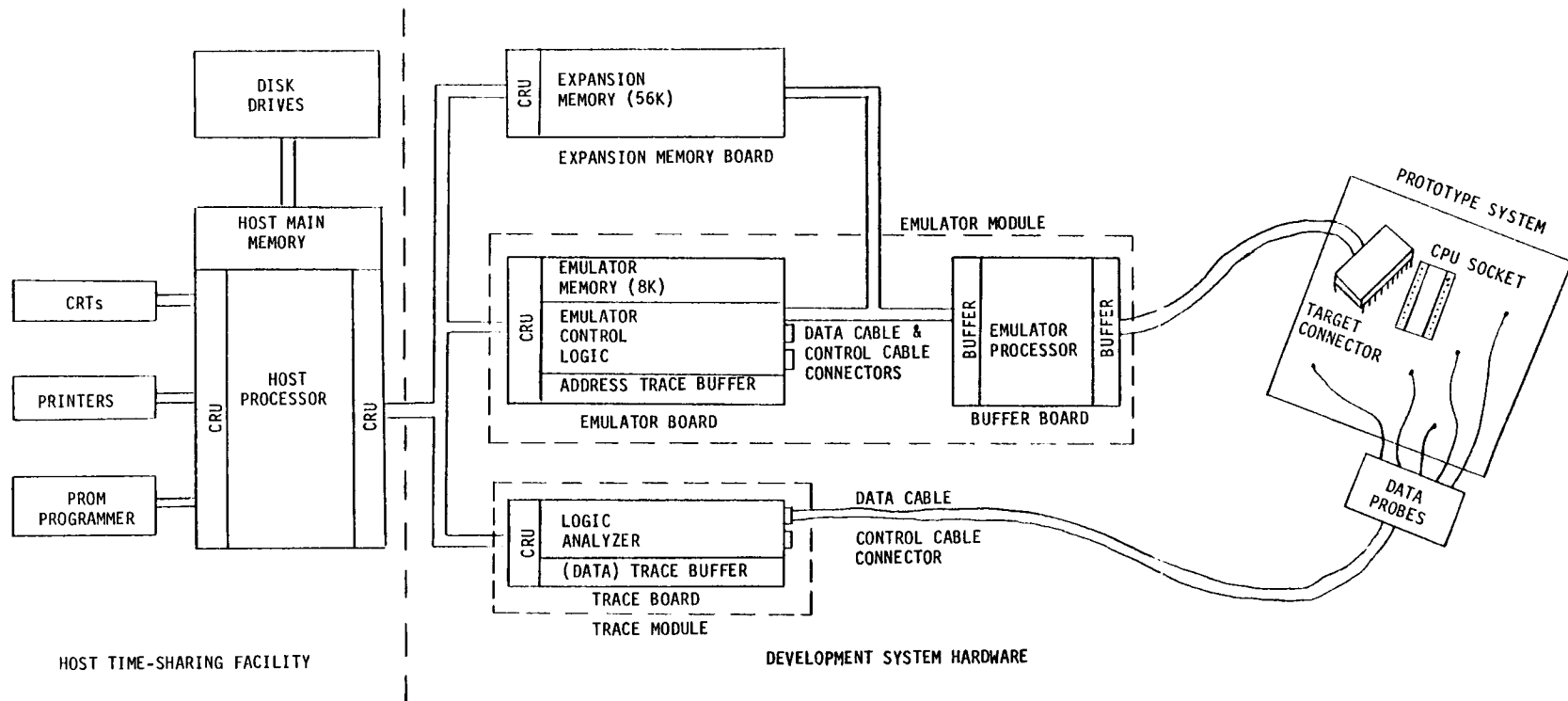


Figure 4.2. AMPL development laboratory hardware configuration.

The AMPL supports two types of emulator boards, the 9900/9980 and the 9940. The 9900/9980 emulator board contains trace and breakpoint logic as well as 8K bytes of substitute memory. An optional board of 56K bytes of memory expansion is available to increase the substitute memory capacity to 64K bytes. The emulator board also has a 256 word by 20-bit wide trace buffer. The trace buffer on the emulator board is hardwired to trace the address bus and some memory control lines of the emulator processor for monitoring memory activities such as memory read, memory write, and instruction acquisition of the processor.

The 9900/9980 emulator board supports development of the following microprocessor-based systems: TMS 9900, TMS 9900-40, TMS 9980A, TMS 9981, and SBP 9900A. In order to develop each microprocessor type a unique buffer board is required. Each buffer board contains a microprocessor that is to be emulated and buffer circuits for the microprocessor, as well as a substitute clock. The buffer board provides the address bus, the data bus and control lines to the emulator board through two flat ribbon cables. A target connector from the buffer board connects directly into a prototype system for in-circuit emulation.

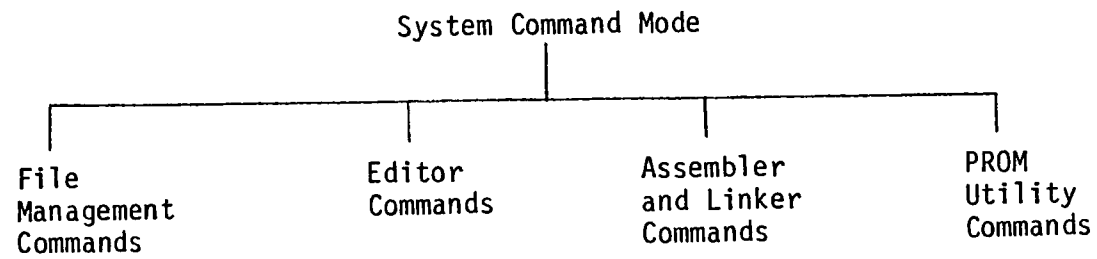
A 256 word by 20-bit wide of trace buffer is also available on the trace module. When the trace module is in stand-alone mode, it functions like a logic analyzer with twenty data channels which can be connected to various TTL test points on the prototype hardware through the data probes shown in Figure 4.2. However, the trace

module may also be hardwired to the emulator module in order to trace the data bus and to control the processor activities. The two modules can be connected through the control and data cable shown in Figure 4.2. A control cable allows the two modules to operate synchronously. When the data cable is connected between the two modules, the trace module is hardwired to trace the activities of the data bus. If connected differently, the data probes can provide selective monitoring of up to 20 TTL test points on the prototype hardware in synchronization with the emulator module. There are six different operational modes for the two modules, which will be discussed later [13].

AMPL Development Laboratory Command Structure

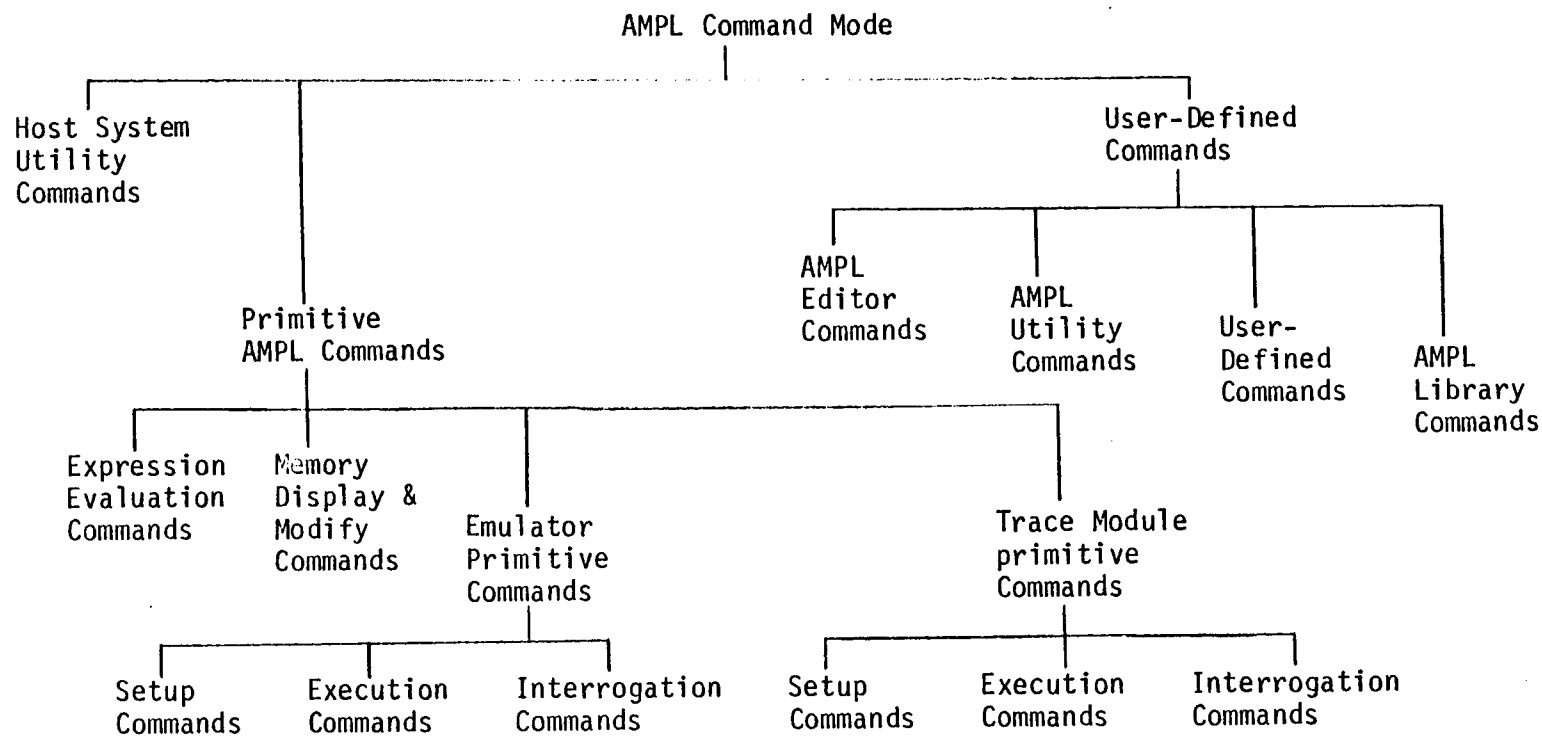
A microprocessor development effort is generally separated into two categories, program development and software and hardware testing. Program development is carried out in the system command mode, while software and hardware testing is implemented in the AMPL command mode. The AMPL supports multi-user operations in both modes.

In the system command mode the program development commands manage and control the devices that are grouped as the host time-sharing facility in Figure 4.2. These commands are classified into four groups, as shown in Figure 4.3(A). The first command group is composed of file management commands. They allow an operator to create, delete, and list directories, and to create, delete, append, show, and print files [14]. The second group of commands provide an operator a means to create and modify programs stored on a disk.



(A) System command structure.

Figure 4.3. Command structure.



(B) AMPL development laboratory command structure.

Figure 4.3. (continued)

The third command group is composed of the assembler and linker commands. An assembler assembles a source program and produces an object file and a list file. The object file, which contains only machine code, can later be loaded into the emulator for debugging and testing. The list file contains both the source program and the deassembled machine code and addresses. Linker commands allow a large program to be divided into modules and assembled separately. After each module is separately tested, they can be linked together into one large program. This feature allows the modular approach to software design which is absolutely essential for a large development effort.

The final command group is composed of the EPROM utility commands. These commands are not used until the final phase of a design process. During this time the working version of the application object file is programmed into the EPROM devices and placed into the prototype system for the final verification. Some EPROM programmer commands are: READ, SHOW, CHNG, PROG, and VRFY [17].

The AMPL command mode is entered from system command mode by the AMPLDX command. In this mode there are three general classes of commands as shown in Figure 4.3(B). The first class of commands are the host system utility commands. These commands serve as a communication link between the two sets of hardware, the host time-sharing facility, and the development system hardware, as shown in Figure 4.2. An assembled object file is loaded into the target memory through the command, LOAD. The target memory here refers to either the

emulator memory, prototype memory, or both, depending on how the user configures it.

A comparison between the target memory content and an object file on a disk is done by the VRFY command. This command is useful at the end of a debugging session. Any modification and patching that is done during debugging can be displayed at the user's terminal by the VRFY command. The displayed differences or any display on the user's terminal while in the AMPL command mode can be duplicated to a user designated file on a disk or to an output device such as a printer by using the utility command, LIST. This utility command is extremely useful when a hard copy of a traced program is desired or when a large amount of program modification is done during a debugging session.

Before exiting the AMPL command mode any segment of the target memory can be saved for later use in a file on the disk using the DUMP command. The memory content saved using the DUMP command is in an object file form and can be reloaded into the target memory using the LOAD command. There are several other input/output utility commands such as OPEN, CLSE, DPLY, REW, etc., that allow the user to open, close, or direct the AMPL output to a device or file. Finally, the utility command, EXIT, allows a user to leave the AMPL command mode and to return to the system command mode [13,18,19].

The second group of commands in the AMPL command mode is the primitive AMPL commands shown in Figure 4.3(B). These commands are called primitive commands because they are the lowest level commands

in the AMPL command mode. They provide direct control statements for the emulator and the trace modules. In addition, a set of expression-evaluation commands are also included for use in data reduction and analysis. All these primitive AMPL commands can be used interactively or as statements in a user-defined command. A user-defined command is a higher level of command which is written in the AMPL programming language. Both user-defined commands and the AMPL programming language will be discussed later.

AMPL has a very rich set of arithmetic and logical expression evaluations. Three types of symbols and variables are valid for expression-evaluations. They are system variables, user-defined variables, and load module symbols. System variables will be discussed later. Load module symbols are symbols loaded into the AMPL using the LOAD command. The word "symbol" will refer to all three types of symbols from now on.

Only integer values are allowed in an expression. Output of an expression can be in the following display formats: binary, octal, decimal, hexadecimal, or ASCII. Variable arrays of one or two dimensions are also available. Some examples of interactive expression evaluations are shown in Figure 4.4. One-line assembling and disassembling are also provided.

The next group of primitive commands is composed of memory display and modification commands. These primitive commands provide the user an easy access to the target memory whenever the emulation is halted. Display and modification of the target memory can be done one

<u>Expressions</u>	<u>Comments</u>
? 10+4/2*3-4: DH 12 >000C	..Display in decimal and hex
? A=12; B=3	..User-defined variables
? A OR B: B50 <01111 !000017	..in 5 binary digits and octal
? "AB"	..ASCII evaluation
>4142 ? ARRAY DATA(2,256)	..Define & initialize array to 0
? DATA(1,50)=>FF	..Set element(1,50) to >FF
? #MOV R0,R1# >C040	..One-line assembling & disassembling
? >C040: I MOV R0,R1	.."I" stands for instruction

Figure 4.4. Interactive expression evaluation.

"?" is the AMPL prompt mark.
 ".." is a comment mark.
 ">" is hex.
 "<" is binary.
 "!" is octal.
 ":" means display in.

location at a time or as a sequence of locations while in display/modify mode. Symbols can be used to specify memory locations. The target memory content can be displayed or entered in all display formats in addition to a mnemonic instruction format with or without symbolic arguments. Some of the target memory display and modify capabilities are shown in Figure 4.5.

The AMPL also provides simple display and modify commands for a program counter (PC), a workspace pointer (WP), a status register (ST), and registers (Rn). All display formats mentioned above apply here [13,18,19].

<u>Status Display and Modify</u>	<u>Comments</u>
? PC=>300; @PC >02E0	. . Set PC and display content of >300
? ST; WP; R0 ST = >C200 WP = >0100 R0 = >0100	. . Display ST, WP, R0

Direct interaction between the operator and the emulator and trace module is provided through the primitive commands of the two modules shown in Figure 4.3(B). The primitive commands for each module can be classified into three groups: setup, execution, and interrogation. These primitive commands take the form of function calls with keyword parameters. A list of these primitive commands and their syntax is included in Appendix A.

The host processor receives primitive commands for the emulator and the trace modules from the operator. From the parameters of the setup primitive commands the host processor programs the hardware on the emulator and the trace module as though they were peripheral devices. After the emulator and the trace modules are set up, the

<u>Memory Display and Modify</u>	<u>Comments</u>
? @>500=#CLR R12#	..Modify a memory location
? @>500: I	..Display a memory location
CLR R12	
?	..Modify a sequence of locations
? >334?:HJ	..While in display/modify mode
>0334 / >04C5 CLR R5	? #DEC R2#
>0334 / >0602 DEC R2	
>0336 / >0201 LI R1,>0500	? #JNE MUSIC. DELAY#
>0336 / >16FE JNE MUSIC. DELAY	
>0338 / >0202 LI R2,>0600	? ;
?	..";" is a terminator
? >334 TO >336?:HI	..Display a range of locations
>0334 / >0602 DEC R2	
>0336 / >16FE JNE MAIN. DELAY	

Figure 4.5. Memory display and modify.

"@" means content of.

operator can start running two modules using execution primitive commands. These primitive commands allow the operator to stop or start the two modules. At any time the operator can interrogate the running status of the two modules through the host processor by the interrogation commands. Traced data are also displayed using the interrogation primitive commands.

There are four primitive setup commands for the emulator module and its memory. These are EINT, MMAP, ETRC, and EBRK. An emulator module must be initialized by EINT primitive command before it can be accessed for use. An EINT command has three parameters. The first parameter specifies the name of the emulator module to be initialized and the second parameter specifies the emulator or the prototype clock that is to be used for the emulation. Emulation means running the emulator processor. The third parameter allows the operator to initialize the emulator and the trace module simultaneously. The trace module also has its own initialization command, TINT. However the effect of initializing the two modules separately using EINT and TINT is different from initializing the two modules together using EINT. The effects of the initialization process on the operation of the two modules will be discussed in greater detail in the next chapter. In any case, the initialization process is to be done only once, unless a different operation mode is desired. An example of the EINT command for initializing both modules using the emulator clock is:

```
? EINT('EM01',1,'TM01')
```

The next emulator primitive command, MMAP, selects whether the prototype or the emulator memory is to be used as the target memory. A 128-byte block is the increment of the memory mapping. The emulator memory can be designated as write-protected using this command. This specifies that the substituted memory is to emulate ROM. Below is an example of MMAP that specifies that the emulator memory ranging from >400 to >7FE is to be mapped into the target memory with write protection:

```
? MMAP("W",>400,>7FE)
```

Two emulator primitive commands, ETRC and EBRK, set up trace conditions and breakpoints. A trace condition is based on specified memory access type and memory range. Two types of breakpoints are available: the trace completion breakpoint and the address breakpoint. The syntax for ETRC and EBRK primitive commands is shown in Appendix A. A detailed discussion of these commands is included in the next chapter.

Two primitive execution commands, ERUN and EHLT, for starting and stopping the emulation, respectively. No parameter is required for either primitive command.

```
? ERUN
```

```
? EHLT          ..Emulator halted
EHLT = 0000      ..Emulator's status code returned
```

Primitive interrogation commands for the emulator consist of five system variables. The values of these variables can be examined

but cannot be changed. One of these variables indicates the status of the emulator (syntax is in Appendix A):

```
? EST          ..3 indicates emulator processor
EST = >0003    ..is running idle
```

Four other system variables are associated with the emulator trace buffer. These variables are used for accessing the contents of the trace buffer. The oldest sample traced during the last trace operation is pointed to by the variable ETB0, while the most recent is pointed to by the variable ETBN. Any integer between the values assigned to the ETB0 and ETBN variables at the end of a trace operation can be used as an index to access the contents of the trace buffer. This index value is the parameter for the remaining two variables, ETBH and ETB. The emulator trace buffer can be thought as consisting of two arrays, ETBH(N) and ETB(N). ETBH(N) contains the most significant 4 bits of each traced datum while ETB(N) contains the least significant 16 bits of each traced datum. Since the emulator is hardwired to trace the address bus and some memory control lines, ETB(N) contains the traced address while ETBH(N) contains the bus status during the trace operation. The command shown below displays the oldest and the newest emulator trace buffer contents [13,19].

```
? ETBH(ETB0):B4; ETB(ETB0); ETBH(ETBN):B4; ETB(ETBN)
ETBH= <0010 ETB = >31E8 ETBH= <0110 ETB = >0146
```

These primitive display commands are very tedious to use. A simpler and more eloquent way of processing and displaying the traced data by using the AMPL programming language will be discussed later.

Trace module primitive commands can also be classified into three groups: setup, execution and interrogation. The setup group consists of four primitive commands. The initialization command, TINT, initializes the trace module, and requires only one parameter, the trace module unit name. As mentioned previously, more than one trace or emulator module can be installed in the system, so the primitive command below would initialize the second trace module.

```
? TINT('TM02')
```

Before the trace module is set to run, there are three primitive commands that can preset the conditions upon which the data presented on the data lines of the trace module are to be stored as well as when to halt. These setup primitive commands are TTRC, TEVT, and TBRK. A glitch monitoring mode can also be specified by these primitive commands. The syntax for the three primitive commands is shown in Appendix A. A detailed discussion of these primitive commands is included in the next chapter.

The trace module primitive execution commands are TRUN and THLT. They are functionally identical to the counterparts for the emulator, so no example will be given here.

Seven trace module interrogation primitive commands are available. They are all system variables, five of which are functionally the same as the ones for the emulator modules. The TST system variable indicates the status of the trace module, such as event occurred, trace buffer full, and trace module running. The TTB,

TTBH, TTBN, and TTBO system variables for the trace module are functionally identical to the ETB, ETBH, ETBN, ETBO commands. In addition to these five, two other system variables, TNCE and TNC, are available for examining the number of events that have occurred. Each variable contains a 16 bit value. Together, they form a 32 bit event counter. TNC contains the least significant 16 bits of the events counted while TNCE contains the most significant 16 bits [13,19].

```
? TST
TST = >0004      ..Event occurred; trace module halted
? TNCE:D; TNE:D  ..10 events have occurred
TNCE = >0000     TNE = >000A
```

The next level of commands is the user-defined commands, which can be composed of the host system utility commands, primitive AMPL commands, and previously defined user commands. User-defined commands are written in the AMPL programming language, and must be in the form of either procedures or functions. Each of the commands has a unique name. Procedures or functions must be installed by the COPY or RSTR command before they can be used as user-defined commands. On the other hand, all other types of commands (shown in Figure 4.3(B), p. 23) are always available for use in the AMPL command mode without installation.

After functions and procedures are installed as user-defined commands, they can be called by another procedure or function to form even higher level commands. A function differs from a procedure in that it returns a value whereas procedure does not. Therefore a

function can also be used as a variable in an AMPL expression. When a procedure or function is called, the commands within it are executed in the order directed by the AMPL program control statements. These control statements include IF, THEN, ELSE, FOR, REPEAT, WHILE, BEGIN, and CASE. Control statements of these types make AMPL a structured programming language. In fact, the AMPL programming language is a special Pascal-like control programming language based on ALGOL 60, which is a popular international programming language. Programs written in the AMPL programming language are easy to understand and maintain.

In addition to procedures and functions, the AMPL programming language allows a user to define a FORM for each procedure or function created. A FORM is a tool by which the program interacts with the user by displaying messages, menus, and prompting the user for arguments. Arguments obtained by a FORM are assigned to separate local variables labeled ARG(n) and used by the procedure or the function with which the FORM is associated. Users are allowed to use local variables other than ARG(n). These local variables are labeled LOC(n). Local variables may be made available to other procedures or functions by assigning them to global variables, which are user-defined variables [13,19].

While in the AMPL command mode, there is a simple cursor oriented AMPL editor that does not have all the capabilities of the system editor. This editor is provided primarily for user to create and modify procedures and functions without the need to exit the AMPL command mode.

User-defined commands and edit buffers are main memory resident. Users are required to specify the memory requirement when they enter the AMPL command mode. When more memory space is required than was initially specified, the AMPL provides four utility commands for deleting edit buffers, user-defined commands, and symbol tables in order to make memory available for other uses. The DBUF command deletes an edit buffer; the DELE command deletes one or more user-defined commands; the CLR command clears all user-defined commands as well as load module symbol table and user symbol table; MDEL command is used to delete all load module symbols. A load module symbol table contains all labels and symbols defined in an assembly program that is loaded into the AMPL using the LOAD command. A user symbol table contains user-defined command names and user-defined variables. The load module symbol table and the user symbol table can be displayed by the MSYM and USYM command, respectively.

The procedures and functions created in an edit buffer can be saved in a disk file or installed as user-defined commands by using the AMPL utility commands shown in Figure 4.3(B), p. 23. The contents of an edit buffer containing procedures and functions are stored in a specified disk file by the KEEP utility command. The COPY utility command installs procedures and functions in an edit buffer or a disk file as user-defined commands. Two other useful AMPL utility commands are SAVE and RSTR (restore). The SAVE utility command stores all user-defined commands that are presently installed into a specified disk file. These user-defined commands can be reinstalled during the next AMPL session by the RSTR utility command [13,19].

User-defined commands provided by TI are called AMPL library commands, and must be installed by the COPY command before they can be used. They fit the general needs of users so that the AMPL development laboratory can be learned quickly and used more efficiently. On the other hand, each user can define his own commands that will be most useful to him. These high level commands can be modified from time to time and used over and over again.

The AMPL library commands can be classified into four groups. The first group is the emulator library commands. These commands organize all of the emulator primitive commands discussed earlier into a set of easier-to-use interactive commands. For example, instead of having to remember the syntax for the ETRC primitive command, a user can issue the TRACE command, an emulator library command, and the following prompt menu will be displayed on the user's terminal.

```
? ETRC(IAQX,50,>400,>600)    ..Equivalent to
? TRACE                      ..TRACE command
```

EMULATOR/TRACE MODULE TRACE PARAMETERS:

```
TRACE QUALIFIER(MA,MR,MW,IAQ,IAQX) = IAQX
TRACE COUNT(1-255,0-INFINITE)      = 50
LOW BOUNDARY OF TRACE RANGE         = >400
HIGH BOUNDARY OF TRACE RANGE        = >600
```

The same is true for the EBRK primitive command. Even a HELP command is provided for listing a short description for each of the emulator library commands as shown in Figure 4.6.

Among these library commands, the display command, DT, is one of the most useful. This command allows the data stored in the

? HELP

----- EMULATOR COMMANDS -----

ADDRESS TRACE PARAMETERS:

TRACE - SET ADDRESS TRACE PARAMETERS(ETRC).

BREAKPOINT PARAMETERS:

BP - SET BREAKPOINT PARAMETERS(EBRK).

CONTROL:

RUN[(PC[,WP])] - EXECUTE ERUN.

HALT - EXECUTE EHLT.

INIT - INITIALIZE EMULATOR, LOAD AN OBJECT FILE AT
SPECIFIED ADDRESS

SIE[(#0)] - SINGLE INSTRUCTION EXECUTION.

STATUS:

STAT - DISPLAY EMU STATUS(EST,ETYP,ECLK,EMEM).

DT[(FIRST[,LAST])] - DISPLAY THE ADDRESS TRACE BUFFER.

Figure 4.6. Help command.

emulator trace buffer to be displayed in a precise and easy-to-analyze format shown below. The IP indicates an external interrupt request is pending. A READ or WRITE indicates that a read or write operation has been executed to the corresponding address, respectively.

```
? DT
INDEX  IP  ADDRESS  /  CURRENT VALUE
-4                >0320      LWPI >0300
-3                >0322      READ
-2      IP      >0324      LI   R12,>043C
-1                >0326      READ
0                >0318      WRITE
```

Another very useful command provided in the emulator/trace module library commands is the single step command, SIE. This command allows single or multiple execution of instructions. After each instruction has been executed various processor status conditions as well as the instruction executed are displayed [16,18,19].

```
? SIE(3)
STEP  PC      WP      ST      WRITE ADDRESS  INSTRUCTION
1  >0014  >0800  >C000  WRITE @>0802  LI   R1,>0060
2  >0018  >0800  >C000  WRITE @>0440  MOV  R1,@HIST0.+>03A0
3  >001C  >0800  >C000  WRITE @>0818  LI   R12,>0020
NEXT >0020  >0800  >C000                MOV  R12,@HIST0.+>0380
```

The trace module library commands are the second group of AMPL library commands provided by TI. These commands are written to simplify the use of the trace module primitive commands. They are similar to the ones discussed previously. One unique display command provided is the LOGIC command, which displays the stored data in a timing diagram format shown in Figure 4.7.

? LOGIC

LOGIC ANALYZER DISPLAY:

FIRST SAMPLE TO DISPLAY = TTBO
 FIRST BIT TO DISPLAY = 0
 LAST BIT TO DISPLAY = 7
 SCALE FACTOR(POS: MARKS/SAMPLE = 1
 NEG: SAMPLES/MARK)

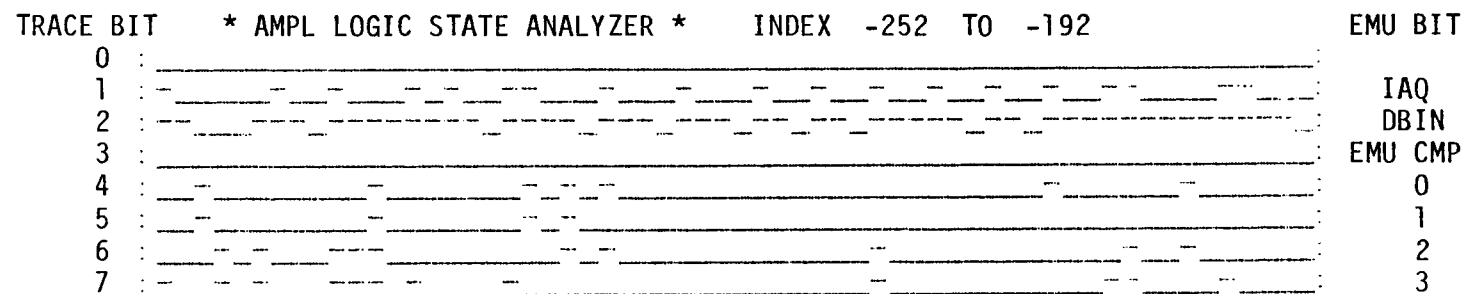


Figure 4.7. Timing diagram display.

Another nice feature that is often provided on a logic state analyzer is the ability to store a set of traced data for comparison to another set of traced data. This feature is available through two of the library commands, TSAVE and TVERFY [16,18,19].

```
? TSAVE      ..Store the traced data
? TVERFY     ..Compare and list any difference
```

COMPARE	BUFFER	CURRENT	BUFFER	DIFFERENCE
37	>2 >8600	-63 >2 >00FF	>0 >86FF	
54	>2 >2401	-46 >2 >2402	>0 >0003	
63	>6 >0100	-37 >2 >0200	>4 >0300	
64	>0 >0200	-36 >0 >0300	>0 >0100	
70	>2 >2402	-30 >0 >2403	>2 >0001	

The third group of AMPL library commands is called the emulator/trace module library commands. The emulator and trace modules should be connected in the emulator controlled internal data interlocked mode, i.e., the control and the data cables between the two modules are interconnected and the two modules are initialized together, so that a meaningful result can be obtained. The TRACE command in this group sets up both the TTRC and ETRC primitive commands to specify a trace condition and a trace completion breakpoint in both modules. The BP and BPA library commands combine the EBRK, TEVT, and TBRK primitive commands in order to set up an address breakpoint in the emulator and an event/delay breakpoint in the trace module. The combined use of the TRACE, BP, and BPA commands provides a rather comprehensive coverage of all possible breakpoints, which will be discussed in the next chapter. These breakpoints are based on the bit pattern occurring in the address and the data buses

as well as the instruction acquisition and the memory read and write control lines. The example below shows a detailed trace of the data and the address buses during a program execution [18-20].

```
? DT(-6)
INDEX  IP  EBRK ADDR  DATA  INTERPRETATION
-6                >0332 >C080  MOV  R0, R2
-5                >0300 >00FF  READ  >00FF
-4                >0304 >00C3  READ  >00C3
-3                >0304 >00FF  WRITE >00FF
-2                >0334 >0602  DEC  R2
-1                >0304 >00FF  READ  >00FF
0                >0304 >00FE  WRITE >00FE
```

Finally, there are AMPL utility library commands which supplement the user-defined commands and the primitive commands. Their functions are to reduce the tediousness involved in the development work as well as to enhance the overall capabilities of the system. For example, one of these commands is FNDWRD, find word, which searches a specified range of memory for a given word. Another one is SETMEN which initializes a given range of memory to any specific 16-bit value. The TIMER command can measure and display the execution time of the emulator by using the trace module event counters TNE and TNCE. The SNAP command will execute a given program until a specified breakpoint is encountered, at which time the status of the processor and the contents of the registers are displayed. The program execution is automatically restarted after the display. This process continues until the user terminates the examination by hitting the CMD key on the keyboard. A set of instruction simulation commands are also provided to simulate the BLWP, RTWP, BL, RT, and XOP instructions while the emulator is halted.

This concludes the overview of all the commands that are available to the users of the AMPL development laboratory [19,20].

CHAPTER V

ADVANCED APPLICATIONS OF THE AMPL DEVELOPMENT LABORATORY

Six Different Operational Modes of the Emulator and the Trace Modules

The AMPL development laboratory provides a user with six different operating modes for using the emulator and the trace modules [19]. These six modes can be classified into three groups according to their uses: software debugging, hardware trouble shooting, and software/hardware integration. A classification of these modes is shown in Table 5.1.

Emulator stand-alone and trace module stand-alone are the two basic modes of operation. The other four modes are based on variations of the data and control cable connections as well as the initialization methods of the two basic modes. Once the two basic modes of operation are well understood, the other modes of operation can be learned with few difficulties. Therefore the discussion of these operational modes will begin with the two basic ones and then extend to the other.

In the emulator stand-alone mode, memory control lines are recorded in the most significant 4 bits of the trace buffer. The least significant 16 bits store addresses. Table 5.2 shows the contents of the emulator trace buffer.

The address trace condition, the trace completion breakpoint, and the address breakpoint are specified by the primitive commands,

Table 5.1. Emulator and trace module operational modes.

Classes	Operational Mode Names	No.
Software Debug	emulator stand-alone mode	1
soft/ hard inte- gra- tion	emulator control/internal data interlocked mode	3
	emulator control/internal data independent mode	4
	emulator control/external data interlocked mode	5
	emulator control/external data independent mode	6
hardware trouble shoot	trace module stand-alone mode	2

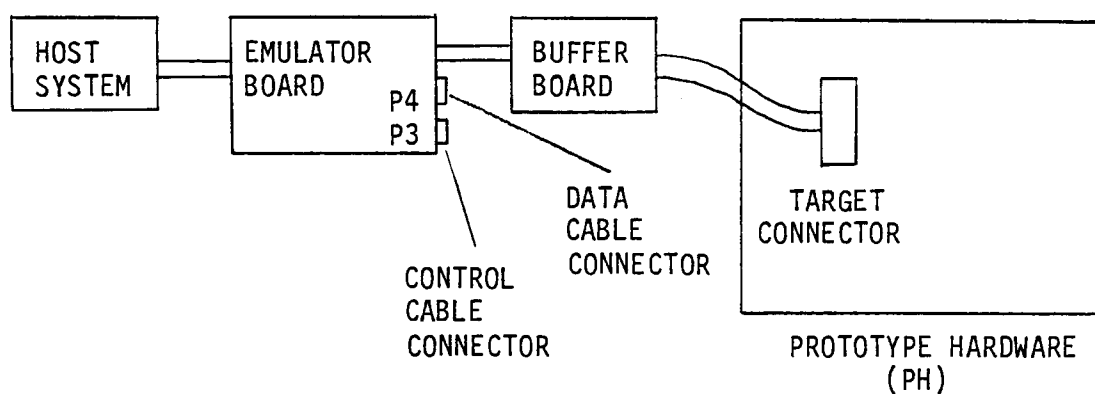
Mode NameMeaning

stand-alone	data and control cables not connected
emulator control	control cable connected
internal data	data cable connected
external data	data probes used
interlocked	modules initialized together
independent	modules initialized separately

Table 5.2. Emulator trace buffer content.

Array Variable	Buffer Bits	Emulator Trace Buffer Content
ETBH	D0 (MSb)	INT (interrupt request)
	D1	IAQ (instruction acquisition)
	D2	DBIN (read/write)
	D3	not used
ETB	D4-D19(LSb)	A0-A15

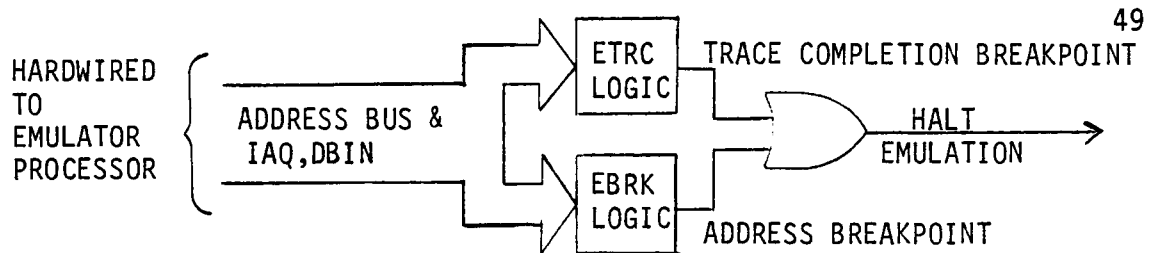
ETRC and EBRK, shown in Figure 5.1(A), or the library commands, TRACE and BP. The ETRC and the TRACE are the same command, which defines the address trace condition and trace completion breakpoint. This primitive command has four parameters: the memory access types, the trace count, and the address trace range both upper and lower bounds as shown in Figure 5.1(B). The memory access type and address trace range together define the address trace condition. The first parameter specifies the memory access types: MA, IAQ, and IAQX. MA refers to all memory accesses within the specified trace range. IAQ refers to all instruction acquisitions within the specified trace range and IAQX refers to all memory accesses for instructions within the specified trace range. The trace range is specified in the third and fourth parameters. A trace completion breakpoint is specified by the second parameter, which allows a user to specify the number of address to be traced before the trace module is halted. This number can have any value from 0 to 255. Zero disables the trace completion



Connections		No connection on the data and the control cable	
Initialization		EINT('EM01', ⁰ ₁)	INIT
Trace Condition/ Trace Completion Breakpoint		ETRC	TRACE
Address Breakpoints		EBRK	BP
Usage	with PH	Provide stimulus to PH	
	without PH	Verify program execution sequence	

(A) Setup and usage.

Figure 5.1. Emulator stand-alone mode.



$$\text{ETRC}(\left\{ \begin{array}{c} \text{MA} \\ \text{IAQ} \\ \text{IAQX} \end{array} \right\} [, \left\{ \begin{array}{c} \text{COUNT} \\ \text{OFF} \end{array} \right\} [, \text{LOW} [, \text{HIGH}]]])$$

$$\text{EBRK}(\left\{ \begin{array}{c} \text{MA} \\ \text{IAQ} \\ \text{MR} \\ \text{MW} \end{array} \right\} [, \text{ADDRESS}, \text{ADDRESS}])$$

Memory Cycle	ETBH Value	Bit value
Memory Access(MA)	0, 1, 4, 5, 6, 7	DBIN=0 or 1
Instruction Acquisition(IAQ)	6, 7	IAQ=1
Memory Read(MR)	4, 5, 6, 7	DBIN=1
Memory Write(MW)	0, 1	DBIN=0

(B) Operation diagram and command format.

Figure 5.1. (continued)

breakpoint. To specify an address trace condition of all memory accesses within the address range of >400 and >7FE, the operator enters the command below. This primitive command also specifies a trace completion breakpoint that halts the emulator after the trace buffer is full, 255 traced addresses.

```
? ETRC(MA,255,>400,>7FE)
```

The EBRK and BP are the same commands which specifies one or two address breakpoints. Up to three parameters can be specified. The second and third parameters specify address breakpoints while the first parameter specifies memory access types: MA, IAQ, MR (memory read), MW (memory write) for the address breakpoints. The example below specifies two address breakpoints which are any memory write to either register R11 or R12.

```
? EBRK(MW,R12,R11)
```

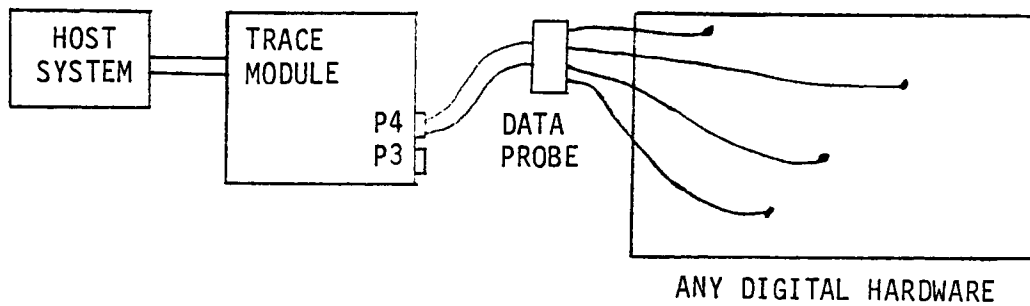
One advantage of the emulator-stand alone mode is that it frees the trace module for hardware trouble shooting by the hardware design group on another terminal. Since no data bus activity is traced, the user must stop the emulation process more frequently in order to verify the proper contents of the memory and registers.

Operation of the trace module stand-alone mode is similar to a logic analyzer, which can be used to trouble shoot any digital system. For trouble shooting the microprocessor prototype hardware, an external stimulus is essential. These stimulus can be provided by the target connector of the buffer module.

The logic analyzer inputs twenty external data lines, four clock qualifying lines, one clock line, and one event line to the trace modules as shown in Figure 5.2 (A), (B). Only the logic states of the data lines are stored in the trace buffer. The other lines provide trace enable signals and external event signals.

Several primitive commands, TTRC, TEVT, and TBRK or library commands, TRACE and EVENT, are available for set up trace conditions, trace completion breakpoints, and event/delay breakpoints. The TTRC or TRACE command specifies the trace condition. This is done using the four clock qualifying lines specified in the first parameter of the TTRC primitive command as either positive or negative enabling or "don't cares." The syntax for the TTRC primitive command is shown in Figure 5.2(B). Data are clocked into the trace buffer only when a clock pulse occurs while the qualifiers match the specified condition as shown in the timing diagram of Figure 5.2(C). Either the internal 10 MHz clock or the external clock from the data probe can be selected by the first parameter of the TTRC primitive command. A TTRC command can also be used to turn ON or OFF a glitch-latching mode provided for D0 to D3. Glitches as narrow as 10 nsecs can be detected. A timing diagram for the glitch-latching mode is shown in Figure 5.2(D) [21].

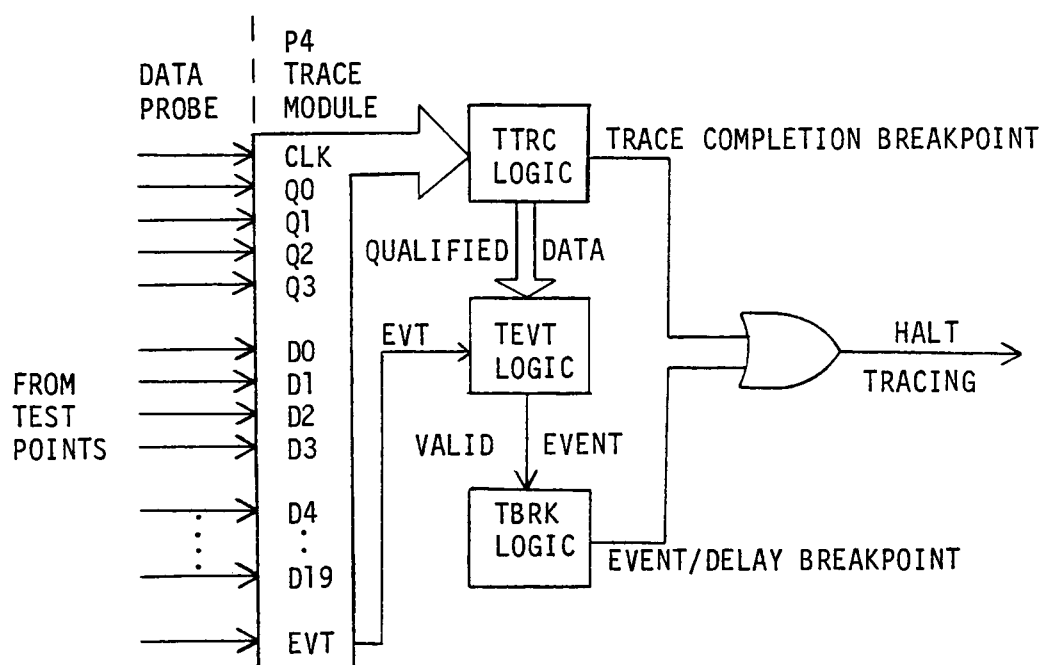
The trace completion and event/delay breakpoint can be specified to halt the trace module. A trace completion breakpoint is specified by the trace count parameter of the TTRC command as shown in Figure 5.2(B). The trace count can be any number between 0 and



Connection		Data probe to PH test points	
Initialization		TINT('TM01')	INIT
Trace Condition/ Trace Completion Breakpoint		TTRC	TRACE
Event/Delay Breakpoint		TEVT, TBRK	EVENT
Usage	with Data Probes	For any digital hardware trouble shooting	
	with Data Cable to Emulator	Monitor data bus activities	

(A) Setup and usage.

Figure 5.2. Trace module stand-alone mode.



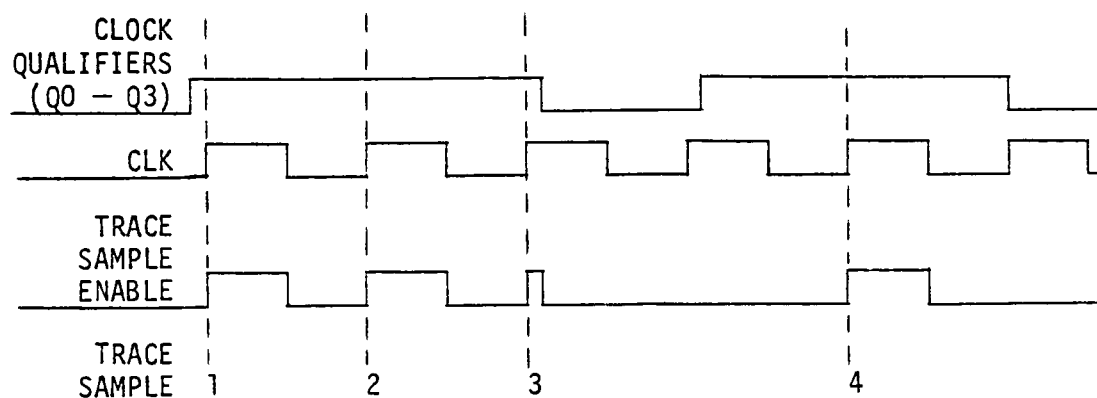
$$\text{TTRC}(\{ \text{INT}[\text{Q0}][\text{Q1}][\text{Q2}][\text{Q3}] \} [, \{ \text{COUNT} \} [, \{ \text{ON} \}]])$$

$$\text{TEVT}(\{ \text{OFF} \} [, \text{PATTERN}[, \text{MASK}]])$$

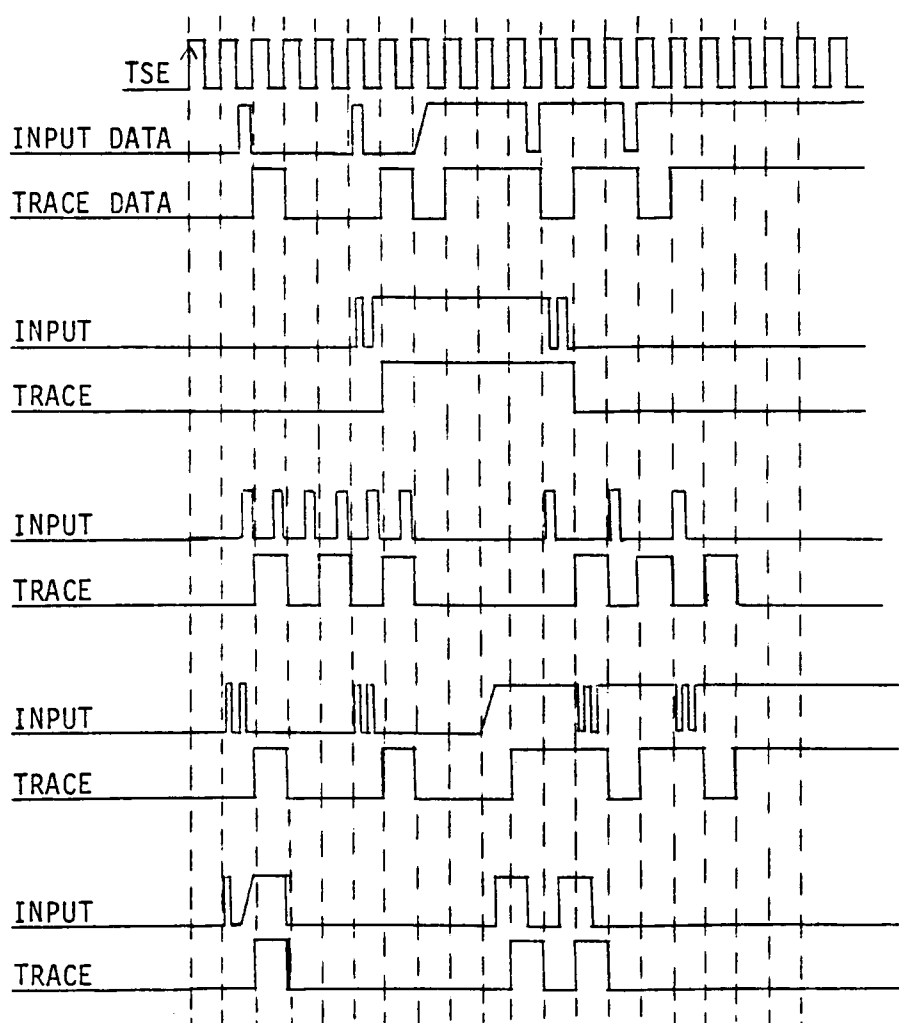
$$\text{TBRK}(\{ \text{COUNT} \} [, \{ \text{DELAY} \} [, \text{INV}][\text{EDGE}]])$$

(B) Operation diagram and command format.

Figure 5.2. (continued)

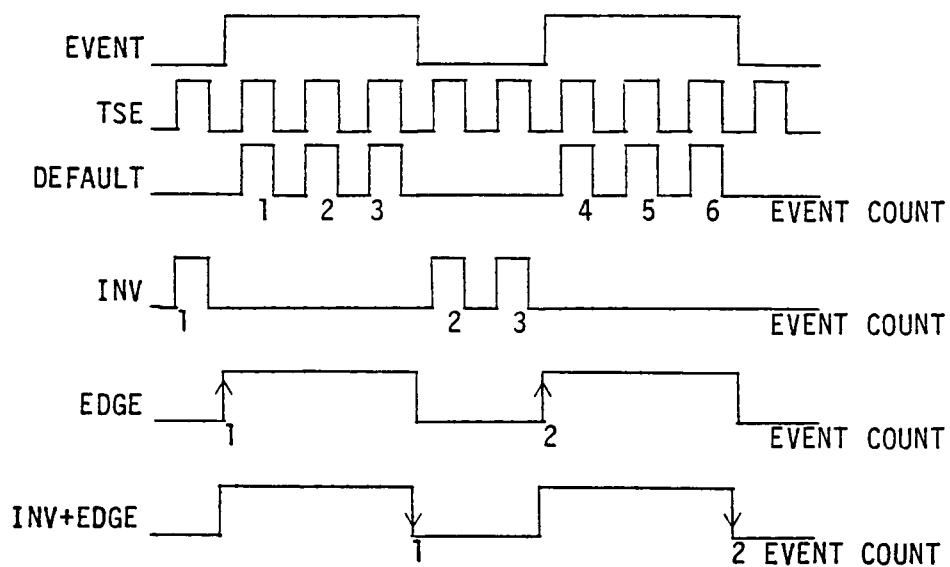


(C) Trace condition.



(D) Glitch mode timing diagram.

Figure 5.2. (continued)



(E) Event counting manners.

Figure 5.2. (continued)

255. ZERO or OFF disables the trace completion breakpoint. The example below specifies the use of the internal clock with -Q1, +Q2, and +Q3 as the clock qualifier. A trace count of 20 is specified and the glitch-latching mode is enabled.

```
? TTRC(INT-Q1+Q2+Q3,20,ON)
```

The primitive commands TEVT, TBRK, or library command EVENT specify event/delay breakpoint. TEVT defines an event and TBRK specifies the manner in which the events are counted as well as how many events are to be counted before halting the trace module. The TEVT command allows any pattern of 20-bit data to be specified as an event. Any of the 20 bit can be specified as "don't care" using a mask specification. Instead of defining an event through matching, the user has the option of specifying the external event line on the probe to be used for event counting. The event count can be any number between 0 and >FFFF as specified by the TBRK command. ZERO or OFF disables the event/delay breakpoint.

Events can be counted in four different manners as specified by the TBRK primitive command. They can be counted by each clock cycle while the event line is valid or invalid. Alternatively they can be counted only on the transition edge of an event becoming a non-event or vice versa. Figure 5.2(E) illustrates the four different methods for counting events.

The delay count is the number of data points to be traced after the last event has occurred. This parameter allows data

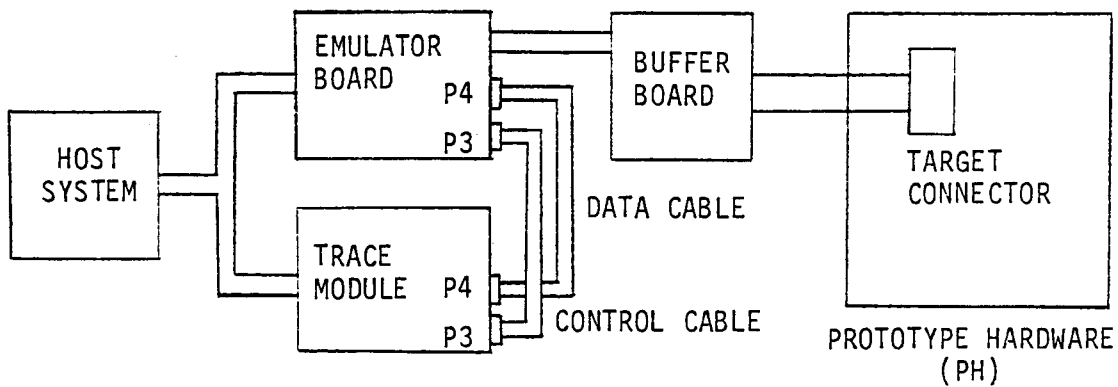
collection to be centered about the delay count specification. Any number between 0 and 255 can be specified as the delay count. After satisfying both the event and the delay count specification of the TBRK command, the trace module is halted. The TEVT primitive command depicted below specifies an event as +D2 and =D3 with D0 and D1 as "don't cares." Bit pattern field is specified as >AA on the least significant byte with the most significant byte as "don't care." The TBRK primitive command specifies an event count of 200 with a delay of 50 traced sampled

? TEVT(+D2-D3,>AA,>FF)

? TBRK(200,50)

The trace module stand-alone mode is useful in any logic design work. However the clock source must be properly chosen. Data traced must be properly interpreted according to the timing diagram shown in Figure 5.2 (C), (D), (E). This may not be easy for some cases. For example, to properly interpret glitches on D0-D3, one may have to connect these lines to other non-glitch monitoring lines D4-D19.

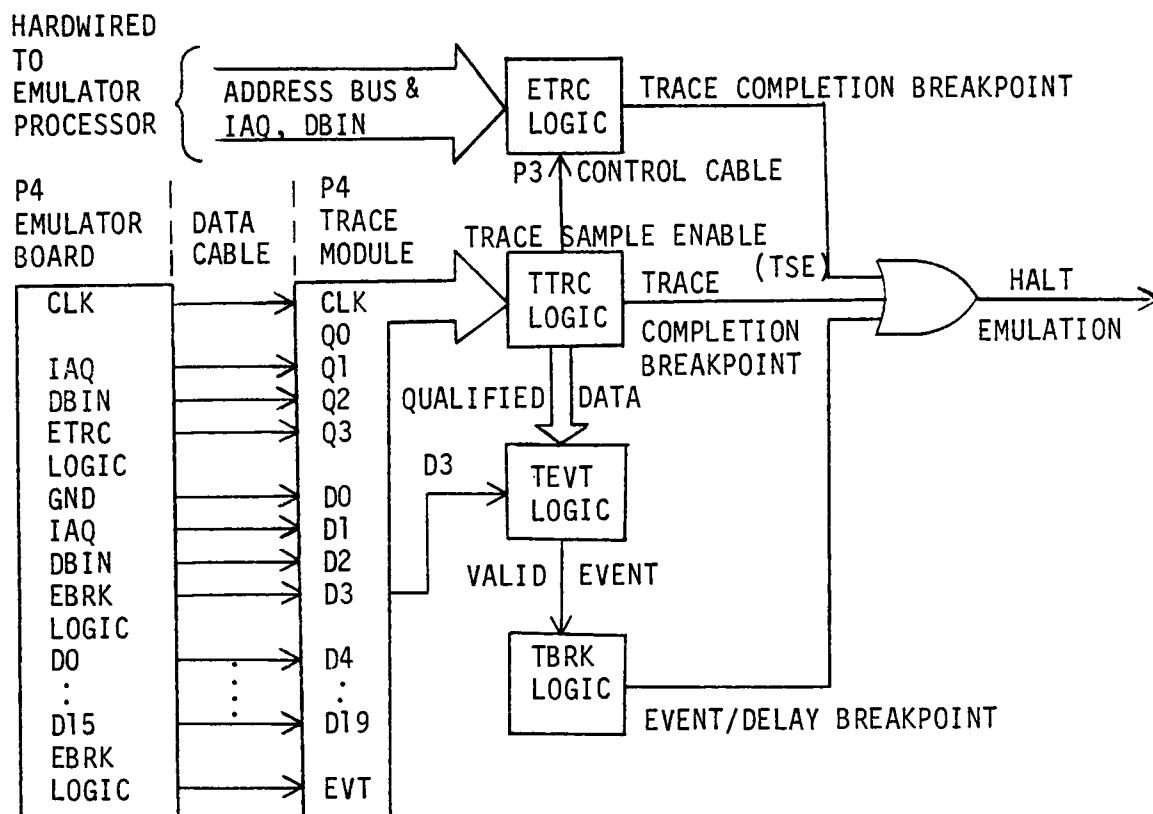
The next operational mode to be discussed is the emulator control/internal data interlocked mode. As the name implies, the two modules are interconnected by the control and the data cables and the two modules are initialized together. The control cable provides synchronized tracing between the two modules while data to the trace module are provided by the emulator as shown in Figure 5.3(A).



Connections		Both data and control cable are connected	
Initialization		EINT('EM01', ¹ ₀ , 'TM01')	INIT
Trace Condition/ Trace Completion Breakpoint		ETRC, TTRC	TRACE
Event/Delay Breakpoint		EBRK, TEVT, TBRK	BP, BPA
Usage	with PH	Software simulation	
	without PH	Hardware/software integration (Initial hardware fault detection)	

(A) Setup and usage.

Figure 5.3. Emulator control/internal data interlocked mode.



- (1) The TEVT logic receives only data that has been qualified by the TTRC logic.
- (2) Tracing of a sample in the emulator occurs only when a sample is traced in the trace module.
- (3) The TEVT logic may recognize either an external breakpoint as defined by satisfaction of the EBRK command or an internal event from qualified data.
- (4) When emulator trace completion, trace module trace completion, or event/delay breakpoint occurs, emulation and tracing in both devices are halted after executing the current instruction.

(B) Operation diagram and specifications.

Figure 5.3. (continued)

By using the specification below the trace module and the emulator module can be set to trace only memory write to register R0.

```
? ETRC(MA,R0,R0)
```

```
? TTRC(-Q2,Q3,255)          ..Q3=ETRC LOGIC; -Q2=MW
```

The resulting trace from the above specification is very useful for verifying the correct write pattern to register R0. This trace specification was not available in either stand-alone mode. Greater flexibility is due to the fact that whenever Q3 is specified as qualifier in the TTRC command, the ETRC trace qualifying logic capability is also available to the trace module. In addition, no data are stored in the trace buffer of the emulator module until it is enabled by the TTRC logic through control cable P3 as shown in Figure 5.3(B).

The breakpoint capability is also expanded in a similar manner by combining the use of the three primitive commands: EBRK, TEVT, and TBRK. This breakpoint is in effect whenever D3 is specified in the TEVT command. For example, the following command specifies that emulation should halt after the first interrupt level has occurred sixty times with a delay of 120 sampled data points.

```
? EBRK(MA,>0004)          ..INTERRUPT 1 VECTOR AT >0004
```

```
? TEVT(D3)                ..D3 = EBRK LOGIC
```

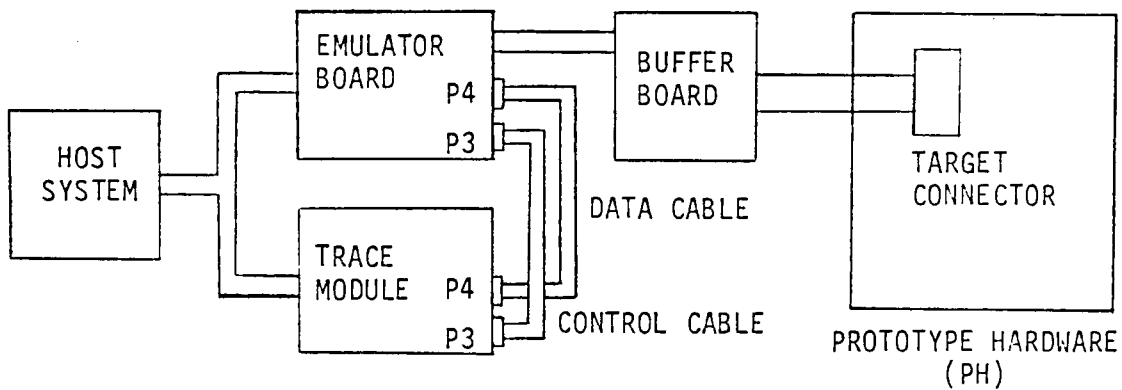
```
? TBRK(60,120)
```

The next operational mode to be discussed is the emulator control/internal data independent mode. The setup for this operational

mode is the same as the previous one except that the two modules are initialized separately as shown in Figure 5.4(A). Figure 5.4(B) shows the operation of this setup. This mode differs from the previous mode in that halt tracing in both devices does not necessarily halt emulation. However, halting emulation will halt tracing in both devices as well. Since data to the trace module are supplied by the emulator through the data cable, this mode provides a similar wide range of tracing capability. In addition, the trace buffer of the trace module can be displayed while the emulator is running. This feature is useful when it is undesirable to stop the emulation process during a debugging session. An example for this emulation is that DMA channel connecting to a disk drive.

Two other modes of operation to be discussed deal with hardware trouble shooting. One is emulator control/external data interlocked mode. The connection for this mode of operation is shown in Figure 5.5(A). A control cable allows the trace module to have direct control of the tracing in the emulator module. All tracing activities are directed by the clock and the clock qualifying lines from the prototype. Selection of these lines is important for obtaining meaningful results.

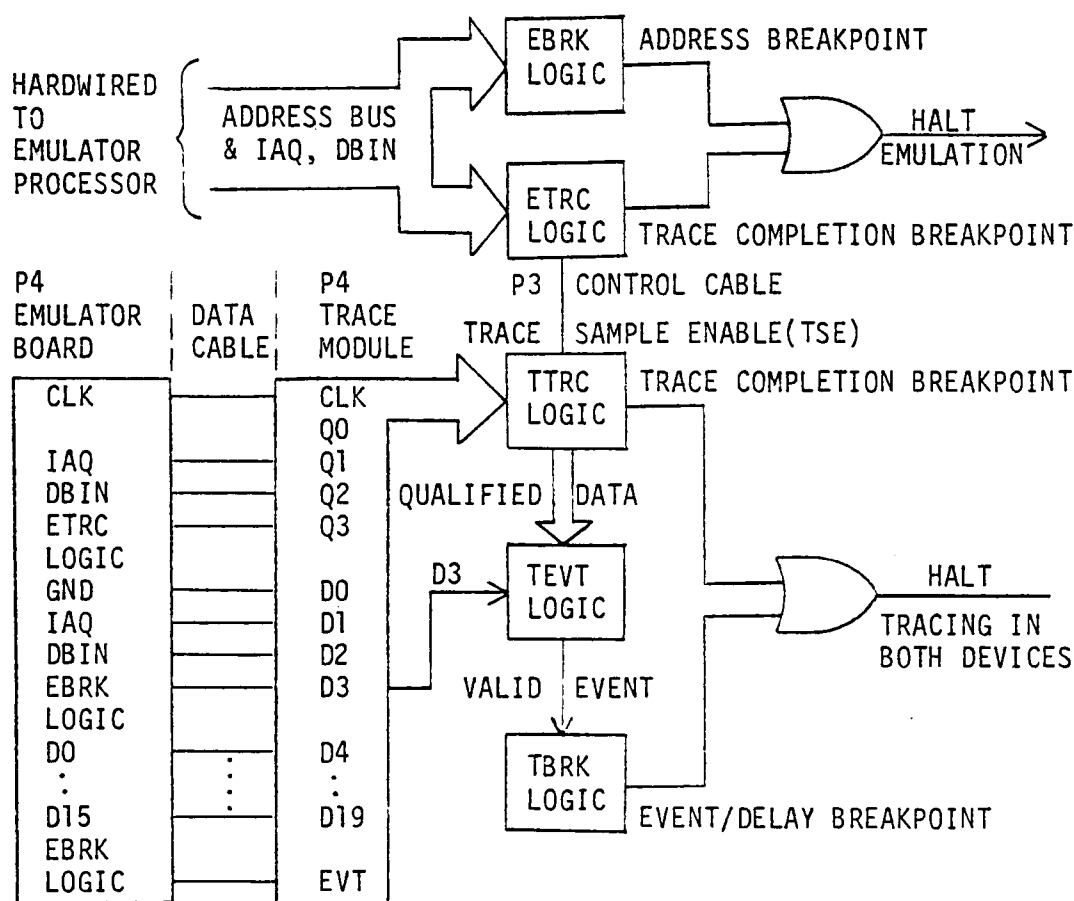
Lines D0 to D3 have glitch-latching capability that is important for detecting noise spikes as well as glitches resulting from race conditions. These types of hardware faults are usually intermittent and extremely difficult to locate by any other means. Another feature provided by this mode of operation is that activities external to



Connection	Both data and control cables are connected	
Initialization	EINT('EM01', ¹ ₀) TINT('TM01')	
Trace Condition/ Trace Completion Breakpoint	ETRC, TTRC	TRACE
Address Breakpoint	EBRK	BP, BPA
Event/Delay Breakpoint	EBRK, TEVT, TBRK	BP, BPA
Usage	Hardware/software integration; Trace data bus activities without halt emulation	

(A) Setup and usage.

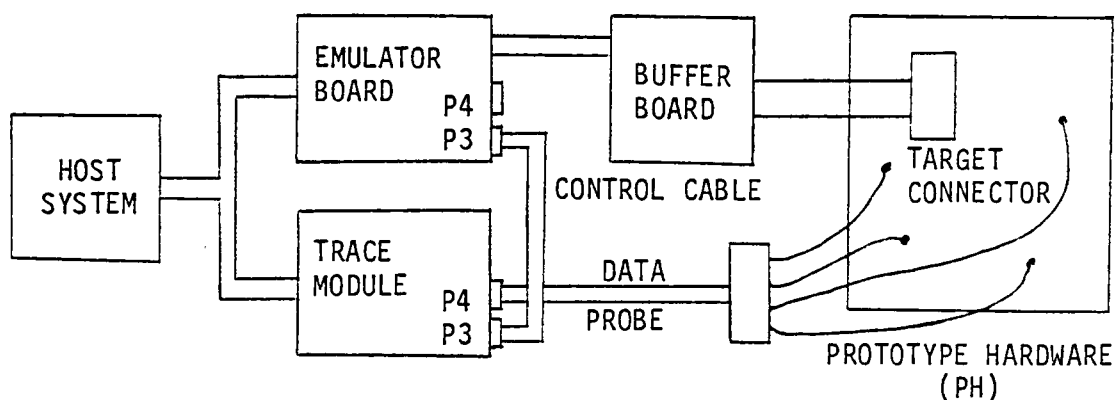
Figure 5.4. Emulator control/internal data independent mode.



- (1) The TEVT logic receives only data that has been qualified by the TTRC logic.
- (2) Tracing of a sample in the emulator occurs only when a sample is traced in the trace module.
- (3) The TEVT logic may recognize either an external breakpoint as defined by satisfaction of the EBRK command or an internal event from qualified data.
- (4) Tracing in both devices is halted when a breakpoint in the trace module occurs.
- (5) An emulator breakpoint will halt emulation at the end of the currently executing instruction, at which point tracing in both devices also halts.

(B) Operation diagram and specifications.

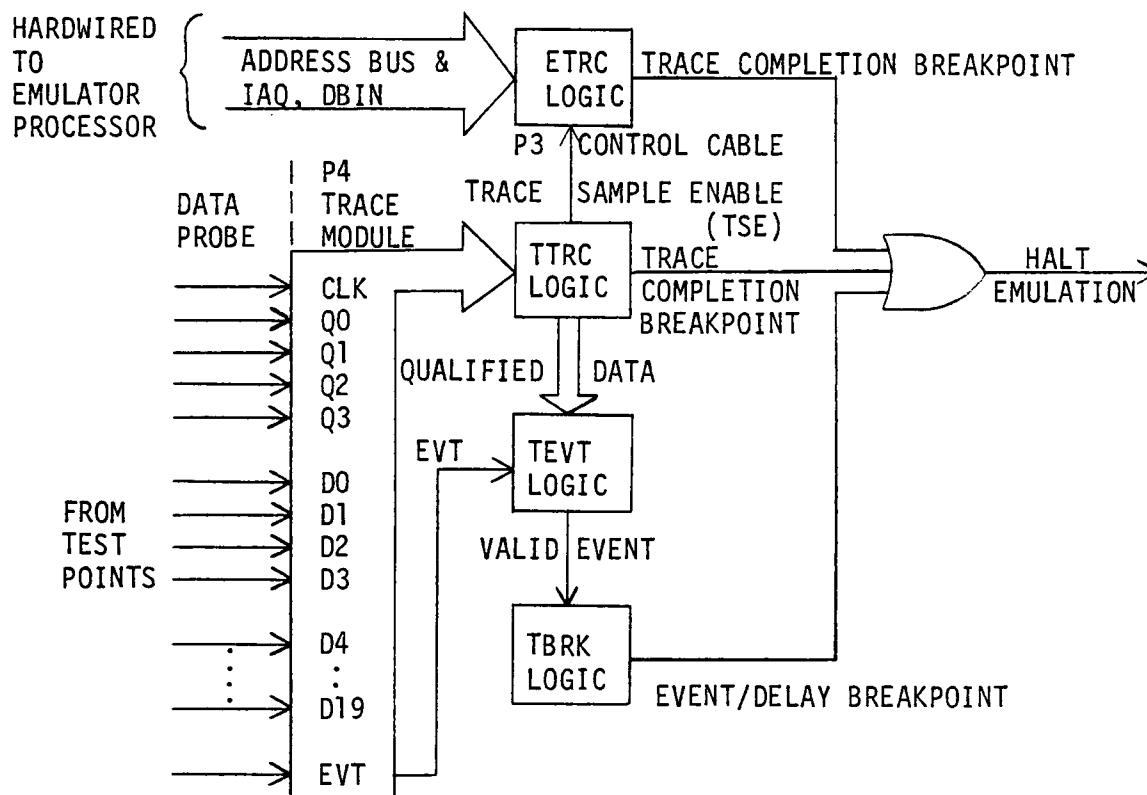
Figure 5.4. (continued)



Connections	Control cable connected; Data probe to test points
Initialization	<code>EINT('EM01',⁰₁, 'TM01')</code>
Trace Condition/ Trace Completion Breakpoint	ETRC, TTRC
Event/Delay Breakpoint	TEVT, TBRK
Usage	Hardware trouble shooting; Noise spike and race condition detection

(A) Setup and usage.

Figure 5.5. Emulator control/external data interlocked mode.



- (1) The TEVT logic receives only data that has been qualified by the TTRC logic.
- (2) Tracing of a sample in the emulator occurs only when a sample is traced in the trace module.
- (3) The TEVT logic may recognize either an external event via the EVENT data probe or an internal event from qualified data.
- (4) When emulator trace completion, trace module trace completion, or event/delay breakpoint occurs, emulation and tracing in both devices are halted after executing the current instruction.

(B) Operation diagram and specifications.

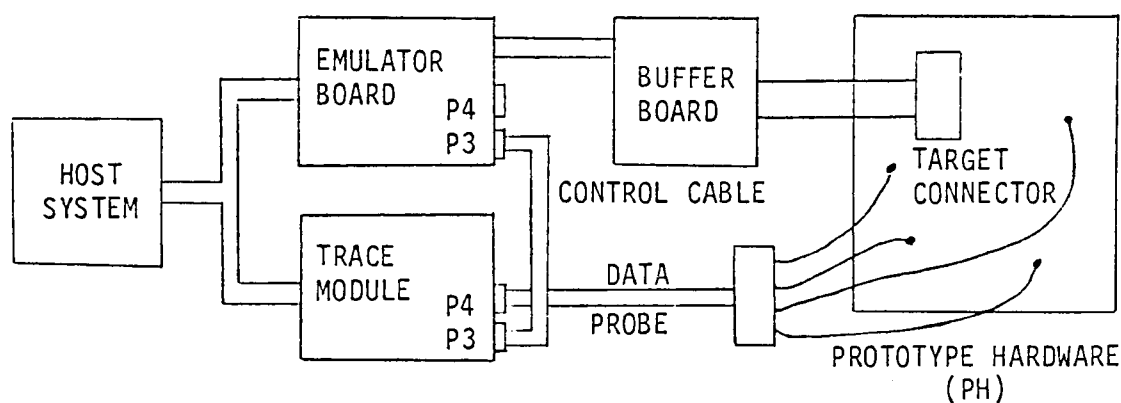
Figure 5.5. (continued)

the processor can be directly related to the processor activities traced in the emulator module. This requires that the CLK line on the data probe be connected to the memory cycle clock.

The emulator control/external data independent mode is connected in the same way as the previous mode except that the modules are initialized separately as depicted in Figure 5.6(A). This disables the trace module from halting the emulation when a breakpoint occurs. This operational mode is very similar to that of the two modules operating in the stand-alone mode. The difference is that the trace module is connected to the emulator module through the control cable. This allows the trace module to direct the tracing in the emulator module. Additionally, the emulator can halt the trace module through emulator trace completion and address breakpoints.

By initializing each module separately, the trace module is allowed to start and stop tracing without affecting the running of the emulator. This has the advantage of never needing to halt the emulator during a debugging session. Like the other external data mode, this mode is used in both hardware trouble-shooting and software/hardware integration.

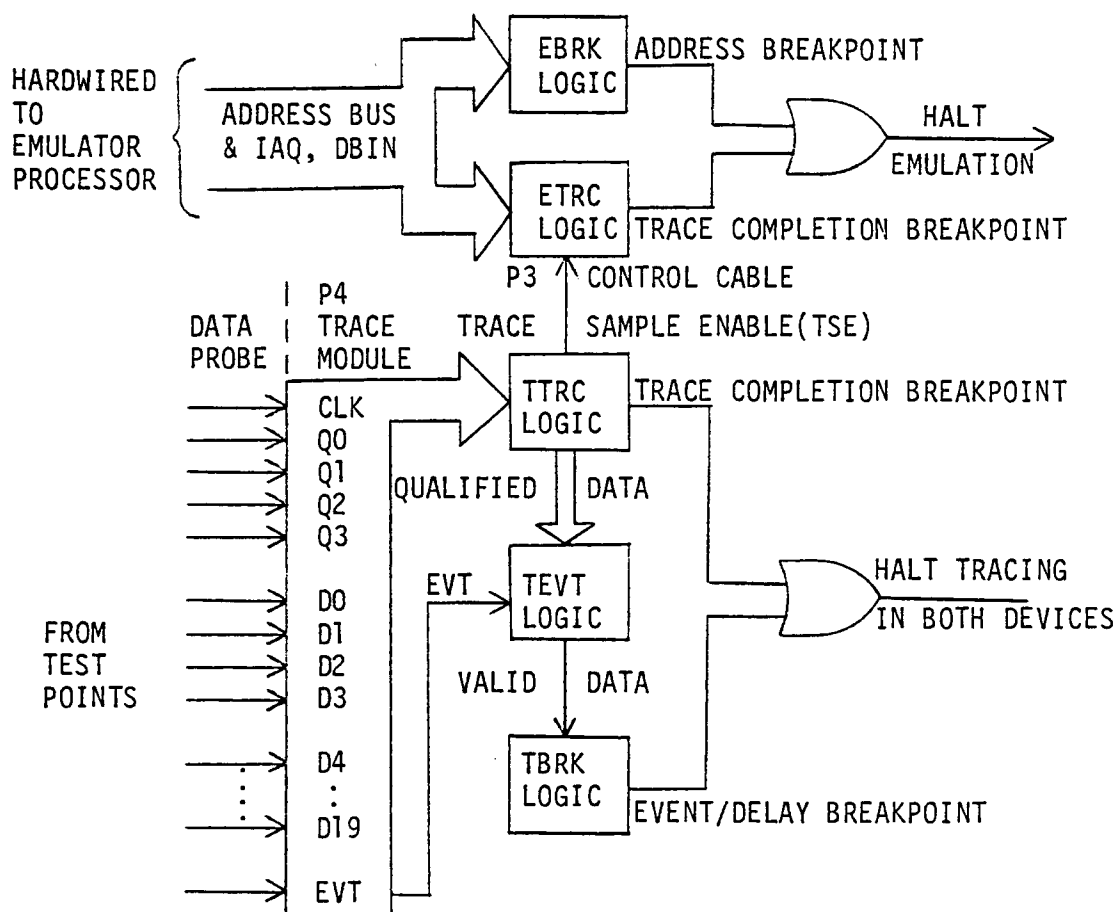
Figure 5.7 summarizes the uses of the various operational modes and depicts how they fit into the overall microprocessor-based system debugging process. The emulator stand-alone mode is included in hardware diagnostic testing to provide a stimulus to the prototype hardware. Software/hardware integration is usually the most



Connections	Control cable connected; Data probe to test points
Initialization	<code>EINT('EM01',⁰₁)</code>
Trace Condition/ Trace Completion Breakpoint	ETRC, TTRC
Event/Delay Breakpoint	EBRK, TEVT, TBRK
Usage	Hardware trouble shoot without halt emulation; Noise spike and race condition detection;

(A) Setup and usage.

Figure 5.6. Emulator control/external data independent mode.



- (1) The TEVT logic receives only data that has been qualified by the TTRC logic.
- (2) Tracing of an sample in the emulator occurs only when a sample is traced in the trace module.
- (3) The TEVT logic may recognize either an external event via the EVENT data probe or an internal event from qualified data.
- (4) Tracing in both devices is halted when a breakpoint in the trace module occurs.
- (5) An emulator breakpoint will halt emulation at the end of the currently executing instruction, at which point tracing in both devices also halts.

(B) Operation diagram and specifications.

Figure 5.6. (continued)

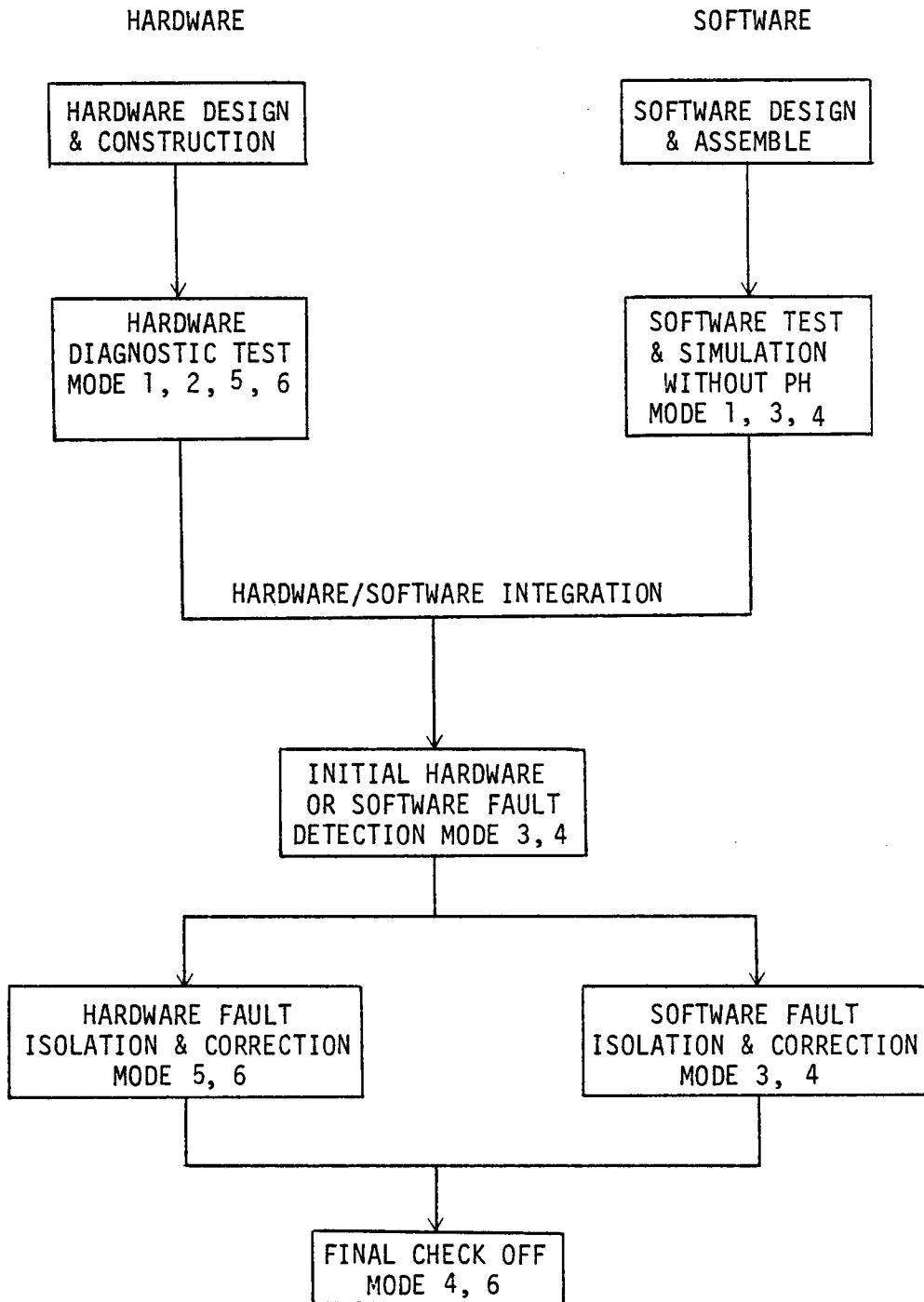


Figure 5.7. Microprocessor-based system debugging process.

difficult debugging phase. To reduce the time spent during this integration phase, it is essential that the software and the hardware are verified independently before integration. The final check-off phase employs mode 4 and 6, because in these modes the trace module can be stopped without halting the emulation.

Automated Memory Test Using AMPL

The AMPL development laboratory is made extremely flexible through the use of the AMPL language as previously demonstrated in the discussion of the library commands. There are other applications of the AMPL development laboratory such as automated testing and real-time software simulation. These two examples are good starting projects for the students of the new microprocessor course. They provide a better insight to the use of the AMPL development laboratory and, in particular, to the concept of in-circuit emulation.

A first example is the automatic memory test. Specifications of this project are described in Project No. 1 of Appendix B. The actual design of the specified memory system described in Appendix B is delayed until Chapter VI. A solution for implementing this memory test is provided in Appendix C.

The purpose of this memory test is to provide a quick and as thorough a memory test as possible. A list of all faulty locations and faulty bits within a specified range of memory is generated by this test. The list is displayed on the user's VDT. A hard copy is also available from a printer.

Three types of tests are conducted. They are the dead cell test, the data path test, and the 50 msec refresh test [18]. In the dead cell test all memory locations within a specified range are cleared to >0000 and tested for their proper contents. Then all memory locations are set to >FFFF and examined. These tests are sometimes called the write-all-zeroes test and the write-all-ones test. The purpose of the dead cell test is to check if all bits within the memory range can be set to zeroes and ones.

Each bit in the memory is tested individually in the data path test for the possibility that it is shorted to another bit. The test is implemented by writing >8000 to all memory locations and the results tested. Then the input data, >8000, is shifted right one bit, >4000, and is written to all memory locations and tested again. The process continues until the one is shifted out of the least significant bit position of the memory word. This is sometimes called a walking-one test. Using the same algorithm, a walking-zero test is performed. The test data are >7FFF, >BFFF, >DFFF, etc.

The 50 msec refresh test verifies memory retention. Initially all memory locations are written with >AAAA. There is a 50 msec delay between the memory write and the following memory read to verify the retention of the memory. Then all memory locations are written with >5555 and delayed 50 msec before verifying again.

To accomplish this memory test, an AMPL program is required. The first task of the program is to prompt the user for a memory range to be tested. This is done by using FORM associated with a

procedure discussed earlier. Of course, the entire memory test can be done simply by using an AMPL program alone. However the complete memory test is very time consuming and involves 36 write and read operations to every memory location within the specified memory range. To speed up the memory test, the test should be generated using assembly language routine as much as possible. In fact the result shows that the assembly language test routine executes four times faster than the AMPL test program. Nevertheless, the complete testing cannot be accomplished using an assembly language routine alone, because the routine itself resides in the target memory to be tested. The memory locations occupied by the assembly language routine must be tested first before the routine can be loaded for testing the rest of the specified memory range. To accomplish this, the AMPL test program is executed by the host processor to verify a portion of memory the first >100 locations to be occupied by the assembly language routine. The rest of the specified memory range is tested by the emulator processor itself.

The assembly language routine shown in Appendix C has several idle instructions located at various locations in the program. When a particular test condition has occurred, the emulator processor is trapped by a unique idle instruction. After detecting that the emulator is idling, the host processor takes the appropriate action depending on where the idle instruction is. These actions involve displaying test messages, restarting the emulator processor to execute the rest of the test routine, and, finally, halting the

emulator processor upon completion of the test. The test messages include test types, test values passed, test values failed, failed memory locations, expected values, actual values, and error bits. These messages are displayed on the user's terminal as well as listed to a user's file for a hard copy printout if needed.

The memory test discussed above is very useful in verifying the correct operation of a RAM memory. Any bad memory location on the target memory can be pinpointed and the use of that location is avoided. On the other hand, a EPROM device can only be verified by the proper contents of its memory. This is easily done with the VRFY command, in which the contents of a PROM device is verified against an object file on a disk. Usually a EPROM device is verified during a EPROM programming session.

Real-Time Software Simulation

A second example on the advanced application of the AMPL development laboratory is real-time software simulation. This is a good project for the students before any microprocessor-based system design project is assigned. By using the simulation program of this project, the students can easily and accurately test their software independent of any prototype hardware. Knowing that the software is working correctly before integration with the prototype hardware greatly reduces the difficulty of the software/hardware integration phase. This defines a good design practice, especially for a novice designer. The specification of this project is included in Appendix B, Project No. 2, and a solution is provided in Appendix C.

There are various solutions to this simulation project depending on the required accuracy of the simulation. Of course the running and the tracing of a program on the emulator can be considered as software simulation. But it lacks the capability to simulate I/O inputs and interrupts. I/O inputs and interrupts are essential because they affect the execution and the final result of a program. I/O output does not affect the execution of a program, but is required for verifying correct operation of a program. Therefore, the ability to examine the I/O outputs must be provided for a complete software simulation.

The following discussion is based on the software simulation design provided in Appendix C. The emulator and the trace modules need to be connected in the emulator controlled/internal data interlocked mode. After the simulation program is installed by the COPY or RSTR utility command, issuing a CSAP command allows the user to enter the simulation mode. The host processor prompts the user for a trace count and a trace range as well as the next expected encounter of an input instruction, STCR or TB. These parameters are used to set up the TTRC, TBRK, TEVT, ETRC, and EBRK primitive commands for the trace and the emulator modules. Because the TEVT command can specify only one data pattern for matching, the emulator can recognize only one specified input instruction, but not both. After the user returns the prompts, the simulation program starts the emulator and the trace modules.

When the two modules are halted by the specified input instruction, the host processor examines the input instruction and

prompts the user for an appropriate number of input bits. The input bits are loaded into the target memory location as specified by the input instruction. Before the host processor restarts the two modules, it prompts the user for another trace count, trace range and next expected input instruction. On the other hand, when the trace count specification is satisfied, the two modules are also halted. Again the host processor prompts the user for the next trace specification. The simulation can be aborted anytime for examining the traced data by hitting a CMD key on the keyboard.

In fact, I/O inputs to a program are simulated by halting the emulator processor on every input instruction encountered. This allows the user a means to direct the execution of a program. I/O outputs and program execution can be traced for verification. This I/O simulation certainly is not real-time, but it is very flexible.

Any I/D operation that requires real-time simulation can be done by substituting the I/O instructions with XOP instructions. For each XOP instruction an input or output table is provided. The values in the input table are set by the user while the output table is examined by the user for program verification. This technique is demonstrated in Figure 5.8.

All the XOPs can use the same workspace where R12 is a dedicated scratch-pad register and each of the R0 to R10 is a dedicated pointer to an I/O table for either the STCR or the LDCR instruction. Before a simulation is started, the XOP vectors as well as I/O table pointers are set to proper values. This technique is very close to real-time simulation. However the setup requirements for each

<u>I/O INSTR.</u>	<u>REPLACED XOP</u>	<u>XOP ROUTINE</u>
TBZ	XOP 14	ANDI R15,>DFFF RTWP
TBO	XOP 15	ORI R15,>2000 RTWP
STCR R5,8	XOP 0	MOV R13, R12 MOVB *R0+,10(R12) CI R0,ITABOE+1 JNE INOEND LI R2,ITABOE INOEND RTWP
LDCR R7,10	XOP 1	MOV R13, R12 MOV 14(R12),*R1+ CI R1,OTAB1E+2 JNE OTIEND LI R1,OTAB1E OTIEND RTWP

INPUT TABLE

ITABOB	:	_____
	:	_____
(R0 PTR.)	:	_____
	:	_____
ITABOE	:	_____

OUTPUT TABLE

OTAB1B	:	_____
	:	_____
(R1 PTR.)	:	_____
	:	_____
OTAB1E	:	_____

Figure 5.8. Real-time simulation by substitution.

simulation may be cumbersome and error prone for a program that contains many I/O operations. The setup can be done by a user-defined command written specially for simulation of a particular software routine. This command can be used over and over again with little or no modification for initialization before each simulation.

Interrupt simulation is simpler than I/O simulation and is an extremely useful tool. Based on the solution given in Appendix C, an interrupt is simulated by a user-defined command, INTR. The INTR command prompts the user for an interrupt level. Then it halts the emulation and compares the requested interrupt level with the interrupt mask of the processor. If the requested interrupt level is greater than the interrupt mask, no interrupt is simulated and the interrupt failed message is displayed on the user's terminal along with the status register value. Otherwise, the host processor initiates a context switch and restarts the emulator and the trace modules. The interrupt service routine is executed as if an interrupt has been generated. One discrepancy of this interrupt simulation is that the interrupt mask of an external I/O device, such as the TMS 9901, is not checked. However a simulation of this type is helpful in isolating possible faults and in verifying the correct operation of an interrupt routine.

CHAPTER VI

MICROPROCESSOR-BASED SYSTEM DESIGN EXAMPLE

The intent of this chapter is to actually design a microprocessor-based system to illustrate the design process involved using the AMPL development system. Various salient features of the AMPL development system discussed earlier will be used to show how they can simplify a microprocessor-based system design. The project chosen for this example is the histogram analyzer. Two other projects have been designed and tested and found suitable for the class projects. These two projects are the system alarm clock and piano keyboard synthesizer. The specifications and components used are included in Appendix B.

Design Process Overview

Figure 6.1 depicts the various design phases for a microprocessor-based system. The first phase is the project specification, which terminates with the hardware system block diagram and the software system flow chart preparation. Then the hardware and the software are divided into functional modules that can be designed and tested sequentially. Development of the hardware and the software are carried out concurrently starting with the most basic module such as the system clock and the power-up circuit. Each module should be tested independently whenever possible. This modular design approach helps to isolate faults. Both software and hardware are verified

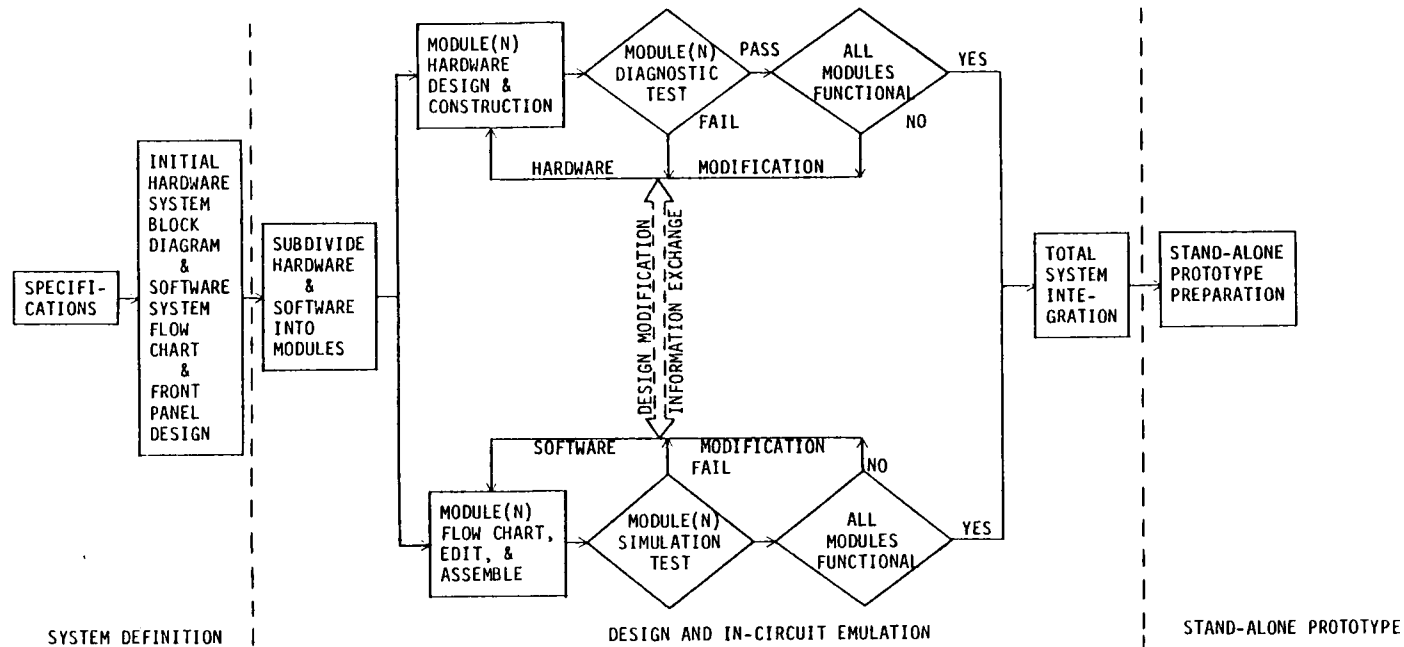


Figure 6.1. Microprocessor-based system design phases.

independently before the integration phase. Various operational modes of the in-circuit emulation are used as discussed earlier. Software is verified by simulation and hardware is tested by specially written short diagnostic routines. Software/hardware integration without assurance that the two are functional can be very frustrating. Finally the software is burned into a EPROM device and the micro-processor is placed into the prototype microprocessor socket for the stand-alone prototype operation.

Project Definition

Defining the project is the initial and most crucial step in the whole design process. Usually a project is only roughly defined when it is in the conceptual stage. A designer needs to refine the project definition. To do so, he may need to do research as to the functional purpose of the project. The next step is defining a boundary specification which set a limit to the complexity of the task by defining the capabilities and limitations of the system.

For the histogram analyzer project, the definition and boundary specifications are stated in Appendix B, Project No. 3, and restated in Table 6.1. For a more difficult project the specification can state a required minimum sampling speed and provision for selecting input sampling rate. Since the above additions would unnecessarily complicate the design example, the specification in Appendix B is used.

Definition of a histogram analyzer:

A device that samples an input signal and plots a graph of the frequency of occurrences of all possible sample values.

Table 6.1. Boundary specifications of the analyzer.

Boundary	Specification
Input	0 to 5 volts
No. of Samples	maximum no. of samples without overflow
Output	plot 256 dots on an oscilloscope
Other Display Features	ability to examine the value of any particular data point on the scope

Front Panel Design

The next step is the front panel design. A front panel should be as simple as possible so that the system functions can be easily understood and operated by the user. Yet it should satisfy all of the boundary specifications. A good design of the front panel uses fewer toggle switches, etc. Each I/O device on the front panel should be clearly labeled and its function well defined.

The front panel design at this stage is usually not the final version and will undergo modification as the design progresses. A designer should provide flexibility in the front panel design so that it can be modified easily. In addition, the designer should from time to time review the front panel design to see that it remains the best implementation.

Figure 6.2 shows the front panel of the histogram analyzer. The functions of the devices on the front panel are tabulated in Table 6.2. A histogram/sample display switch is an extra feature not required by the project specification. The reason for the implementation is explained later.

Hardware Block Diagram and System Flow Chart

After the front panel is designed, a hardware block diagram, a system flow chart, and component selections should be the next design step. The three processes are closely interrelated and usually occur simultaneously. However, a block diagram and a system flow chart is discussed first.

For a complex project, it is necessary to construct a general system block diagram showing the major hardware functions. From this general system block diagram, more detail system block diagrams are constructed. Then the detailed hardware design is completed based on the system block diagrams. In a simple project, the detail hardware design is completed directly from the general system block diagram, which is the case for the histogram analyzer project.

Each block is labeled with a name that accurately describes its function. The block name is often derived from the name of the major component within the block. In addition, the major component numbers can be included. Each block's inputs and outputs should also be labeled with their functional names. The number of lines in each data, address, and control bus should be indicated if it is known. Figure 6.3 shows the general system block diagram for the histogram

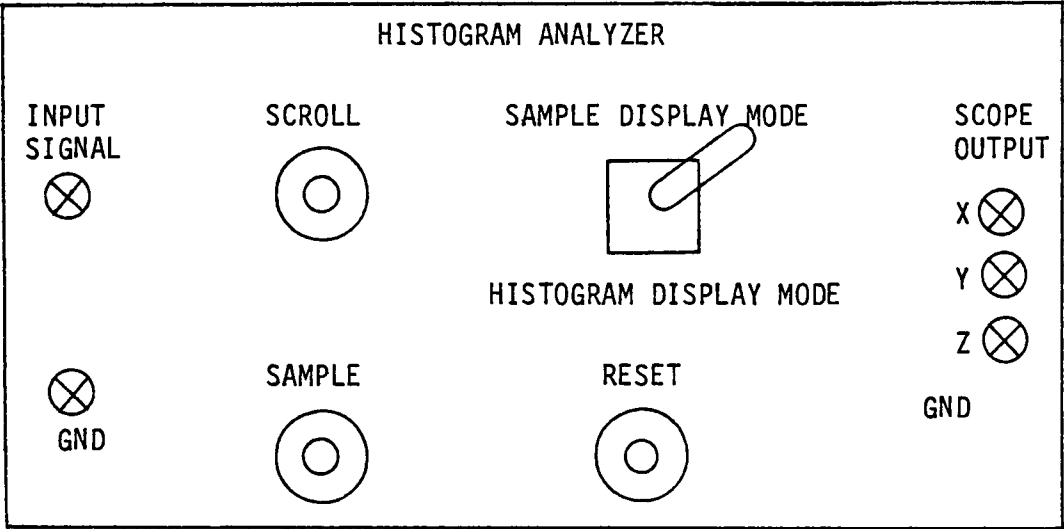


Figure 6.2 Histogram analyzer front panel.

Table 6.2. Functions of the front panel devices.

Button Names	Functions
RESET	This is a power-up reset button. When it is pushed, the histogram buffer is initialized to zero and the scope displays 256 data points at zero value.
SAMPLE	Every time the sample button is pushed, the A/D samples 256 values and tabulates them in the histogram buffer. These values are accumulated over the previous values.
SCROLL	The function of this button is similar to the delay function on an oscilloscope. Whenever it is pushed the displayed data points on the scope are shifted to the left by 20 data points. For example, initially all of the 256 points are displayed on the scope. The first data point represents 0 volts. After the first push on the scroll button, the scope displays only 236 data points. The first data point displayed is the 20th data point, which represents the sample voltage of $20 \times 5 \text{ volts} / 256 = .39 \text{ volts}$, in the histogram buffer. A scroll function enables a user to examine any specified point on an oscilloscope.
HISTOGRAM/ SAMPLE (added feature)	Display selection between a digitized signal and its histogram is provided by this toggle switch. This switch enables the analyzer to function as a simple digital scope as well as a histogram analyzer.

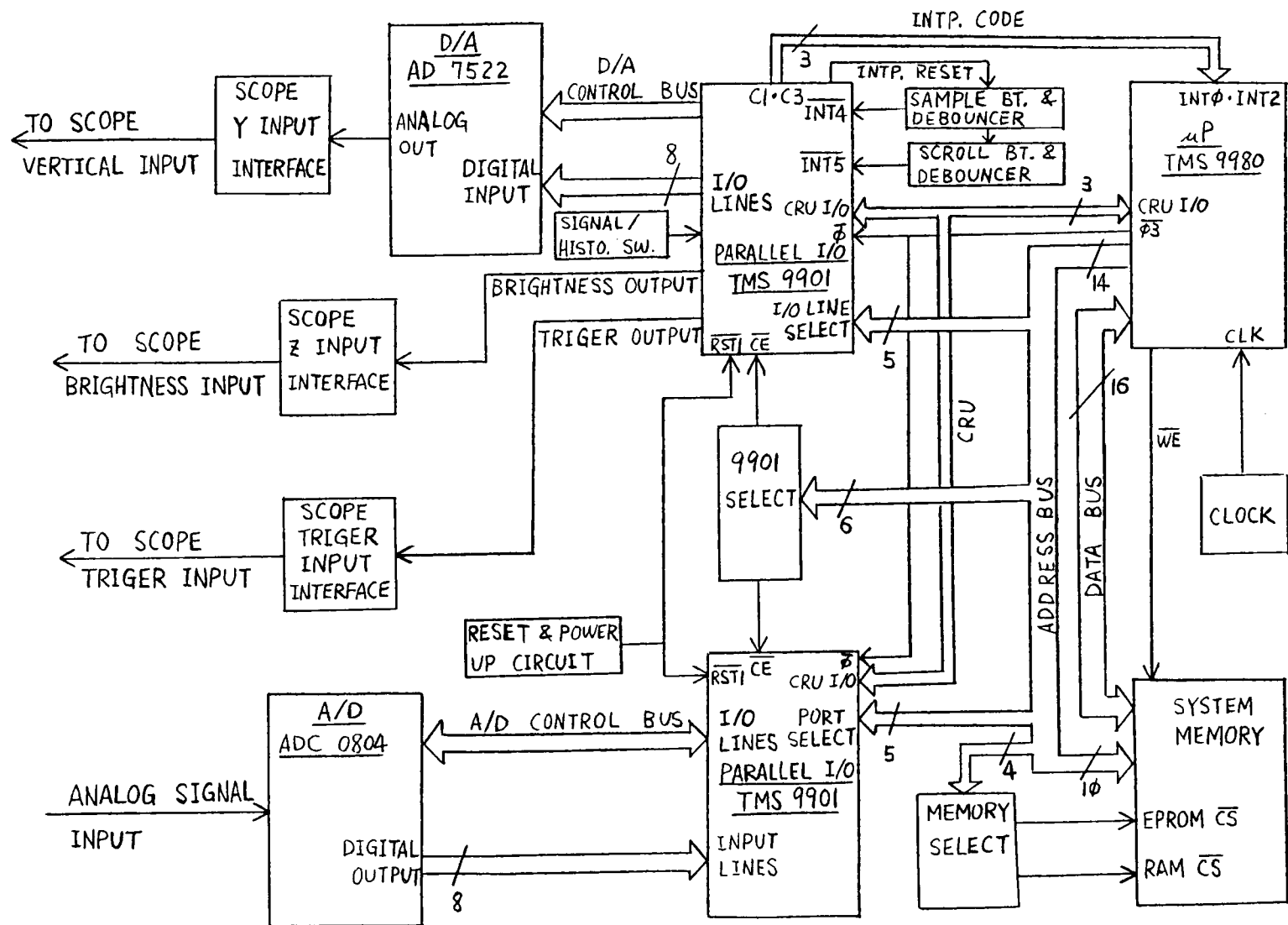


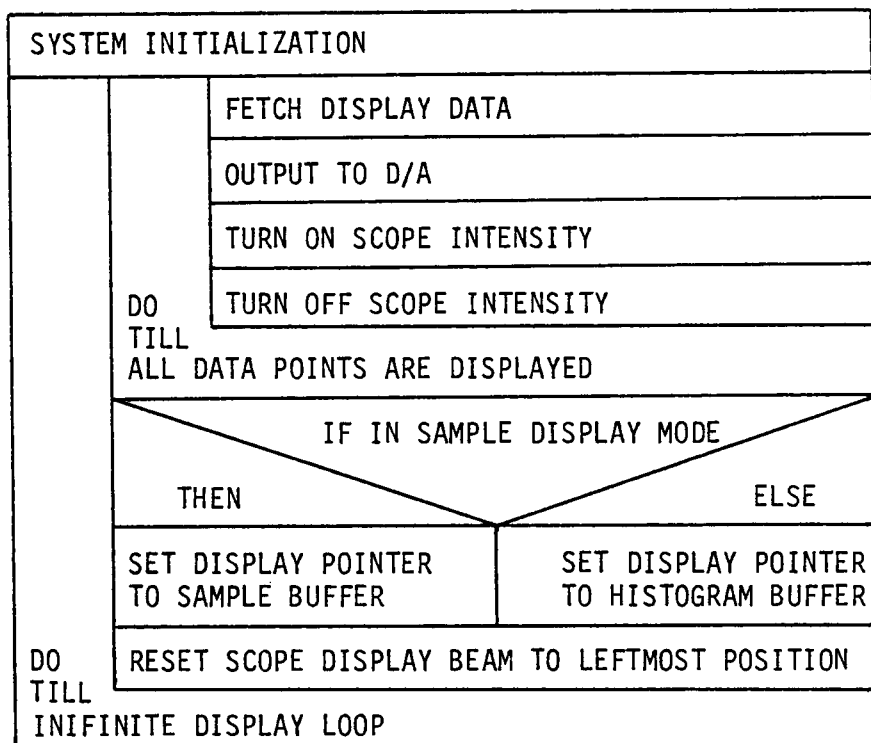
Figure 6.3. Histogram analyzer block diagram.

analyzer. The major interconnections between these blocks are shown and labeled. In addition, all front panel features are included in the block diagram.

The flow chart structure affects the readability of a program as well as providing for easy debugging. A traditional flow charting method is not used here, since it easily leads to unstructured programming, which is very difficult to debug. Rather, a structured flow charting technique is used. This technique restricts unordered branching so that only top-down program flow is allowed.

A general system flow chart defines the general software algorithm used (i.e., no details are included). Figure 6.4 shows the general system flow chart for the histogram analyzer. The main routine initializes the system and then stays in an infinite display loop until either the sample interrupt or the scroll interrupt occurs. The main routine can display either the previously sampled data in a sample buffer or the data in a histogram buffer, depending on the setting of the histogram/sample switch. A sampling interrupt service routine performs 256 A/D conversions on the analog input signal and stores them in the sample buffer. Then the histogram conversion subroutine is called to tabulate the sampled data into the histogram buffer. The scroll interrupt service routine modifies the display buffer starting address, so that the display of either buffers can begin at a predefined location, 20 samples apart.

Each of the two buffers, the sample buffer and the histogram buffer, stores 256 bytes of data. Even though only a histogram



SAMPLE SERVICE ROUTINE

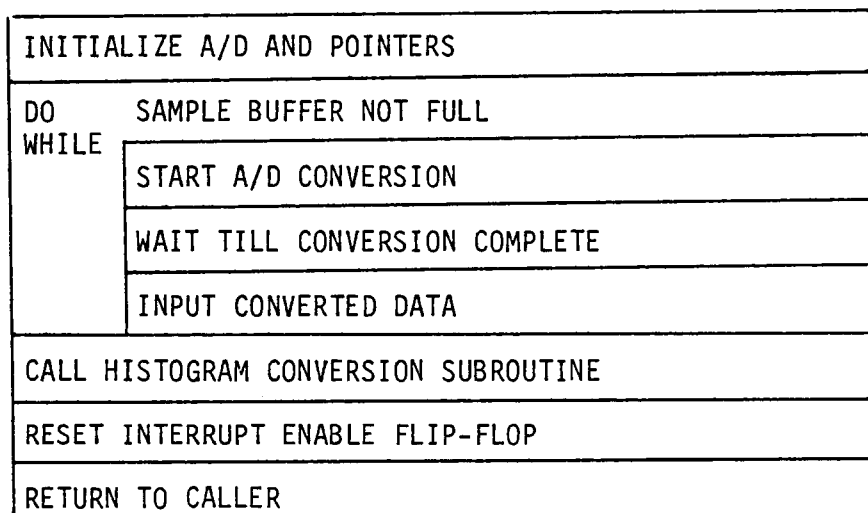
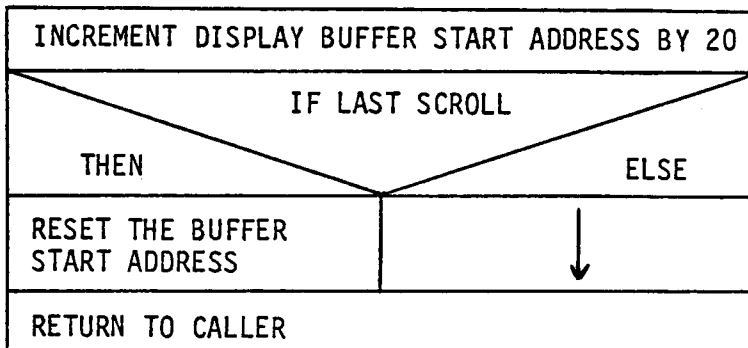


Figure 6.4. General system flow chart.

SCROLL SERVICE ROUTINE



HISTOGRAM CONVERSION SUBROUTINE

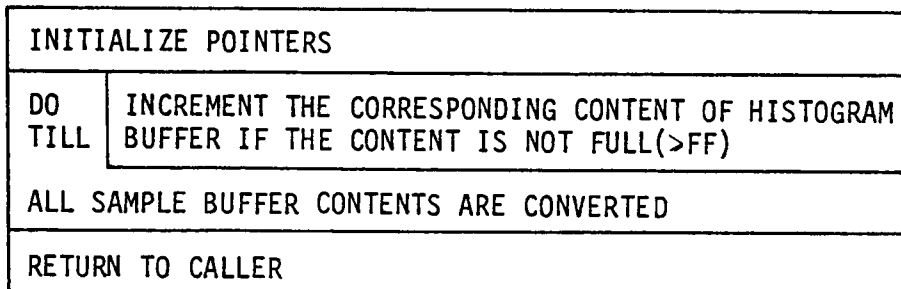


Figure 6.4. (continued)

buffer is required from the specification of the project, adding a sample buffer allows the analyzer to display the digitized signal as well as the histogram. This added feature extends the analytic capability of the analyzer with a slight increase in hardware, a two-position toggle switch, and 256 bytes of RAM, and software. For this reason the histogram/sample display switch is added on the front panel.

Major Component Selections

The availability of the AMPL development system is the single criteria for selecting TMS 9980 microprocessor for this project. Once a microprocessor is selected, the microprocessor supporting devices to be used are usually fixed to devices for that family.

Adapting a typical external clock rate of 8 MHZ for the TMS 9980, the system clock frequency is 2 MHZ. A CY 8.000 MHZ crystal is used. Five hundred twelve bytes of EPROM and 1K bytes of RAM would be enough for this project. Since 1K byte memory components are more available on the market than the smaller size memory components, a 1K byte EPROM and a 1K byte RAM will be used. This allows extra memory for possible unforeseen difficulties and design expansion.

The actual memory component selection will depend on the timing constraint imposed by the microprocessor, the system clock rate, and the bus loading. If the TMS 4045 (RAM) and TMS 2708 (EPROM) are used, the microprocessor can run at a system clock rate up to 3 MHZ without any timing conflict or requiring any wait states [6]. The bus

loading is minimal for this project, since only the microprocessor is connected to the data bus, and the microprocessor and two TMS 9901s are connected to the address bus.

The next major hardware components to be selected are peripheral devices. Since the project specifies that 256 unique sample points are to be displayed on an oscilloscope, 8-bit resolution A/D and D/A are needed. A National Semiconductor ADC 0804 A/D converter is chosen because it can be purchased at most electronic stores. The conversion time is 100 microseconds, which allows a maximum 5 KHZ input signal using the sampling theorem. The ADC 0804 inputs and outputs are TTL compatible and use single 5 volt supply. Interfacing it to the TMS 9980 can easily be implemented with one TMS 9901. The differential analog voltage input increases the common-mode rejection thus reducing noise input. 8-bit resolution can be obtained over any desired analog input voltage range through a special voltage reference input.

Analog Device AD 7522JN, a 10-bit D/A converter, is chosen for this project. Only 8 bits will be implemented in this project. The 10-bit capability allows system expansion in the future. Using 8 or 10 bits, the analyzer can accumulate up to 256 or 1024 samples, respectively, for each discretely sampled voltage. The histogram obtained using a 10-bit is closer to the ideal histogram than the result obtained by the 8-bit. Input data can be loaded in parallel or in serial mode. The AD 7522JN can also be easily interfaced to any microprocessor. Two power supply voltages, +5V and +15V, are

needed. Since a +15 volt supply is needed anyway for operating op-amps in oscilloscope interfacing circuits, the extra power supply requirement is not a drawback here.

It would be impractical to use another microprocessor system interface adapter, because the TMS 9980 microprocessor uses a rather unique serial I/O interface technique, communication register unit (CRU). The TMS 9901 is a programmable system interface adapter which has an interval timer, priority controller, and 16 parallel programmable I/O lines. The timer may be required to time the operation of the A/D or the D/A. The interrupt controller can be used to prioritize the three interrupt push buttons on the front panel. Other I/O interfaces can be implemented through the programmable I/O lines.

Interfacing the AD 7522JN and the ADC 0804 to the TMS 9980 requires two TMS 9901s, which provide 32 I/O lines. Among the 32 I/O lines, 8 lines are needed to input the data from the ADC 0804. Another 8 lines are needed to output the data to the AD 7522JN. Several other I/O lines are required to control the two converters. The remaining I/O lines are needed to drive the oscilloscope interfacing circuits and to clear the interrupt flip-flops. Multiplexing the I/O lines could reduce the number of TMS 9901 required to only one, but would unnecessarily complicate both hardware and software design. In addition, it would make the system extremely difficult to modify.

Another design approach that should be considered here is incorporating the use of a direct memory access (DMA) TMS 9911. This

approach would need only one TMS 9901. The hardware for the DMA approach would be little more complicated than the approach using two TMS 9901s, but the sampling speed of the analyzer would not necessarily be higher using DMA than the two TMS 9901s approach. The reason is that the microprocessor can execute several instructions during the conversion time of 100 microseconds. Therefore the ADC 0804 can operate at its maximum conversion speed using the two TMS 9901s approach as well as the DMA approach. For the above reasons, the approach using two TMS 9901s is adapted for this project. Table 6.3 lists the major components selected.

Table 6.3. Major component selections for the analyzer.

Major Components	Description	Quantity
TMS 9980	microprocessor	1
TMS 9901	programmable parallel I/O device	2
ADC 0804	A/D converter	1
AD 7522JN	D/A converter	1
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2
CY8.000	8.000 MHZ crystal	1

At this point the general concept and design approach to the project is established. The next step in design as shown in Figure 6.1 (p. 79) is to divide the hardware and the software into

modules. In Table 6.4 is listed the order of hardware modules to be designed and tested. Software is developed in the order presented in Figure 6.4.

Table 6.4. Order of hardware modules to be designed.

-
- | | |
|----|--|
| 1. | System clock and power-up circuit (CY8.000 MHZ) |
| 2. | System memory (TMS 4045 & TMS 2708) |
| 3. | System interfacing devices (TMS 9901) |
| 4. | I/O devices and interfaces (ADC 0804, AD 7522JN, & scope interfaces) |
-

Design and Test System Clock and Power-Up Circuit

For the analyzer project, the emulator clock of 10 MHZ may cause timing problems in software and hardware since the final prototype circuit uses an 8 MHZ clock. This is especially true if the prototype system uses the timer provided in the TMS 9901 as a timing source. A clock circuit is usually available from the manufacturer's microprocessor application notes or other reference books and can be constructed easily. The external clock circuit used for this analyzer project is shown in Figure 6.5 and can always be confirmed by measurement [23]. If the clock is operating properly, the emulator should be able to initialize with the external clock connected to the CKIN pin of the target connect.

```
? EINT('EM01',0)    ..Initialize with external clock
```

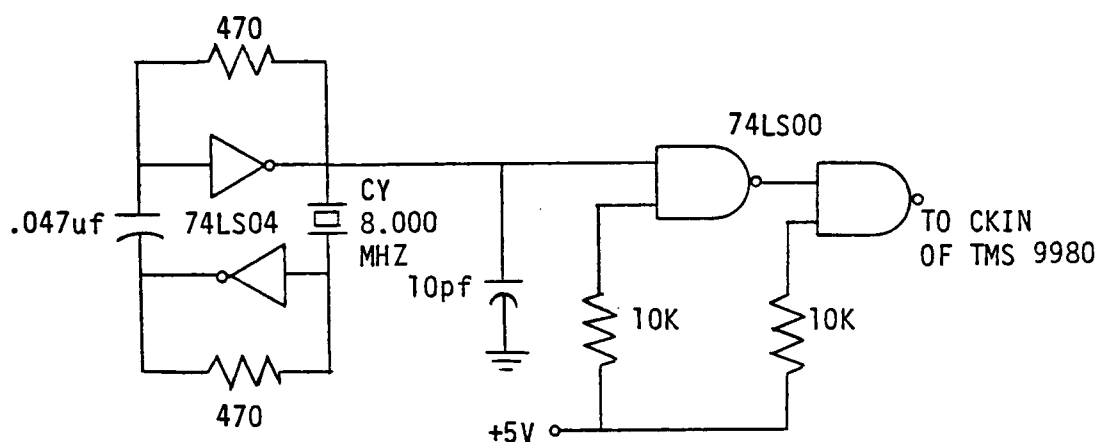


Figure 6.5. External clock circuit.

The power-up and reset circuit are implemented using a simple capacitor charging circuit shown in Figure 6.6. When the power is initially applied to the analyzer, the RC circuit holds the two TMS 9901s and the microprocessor in the reset state until the capacitor has time to charge above the threshold of the TTL logic one state (2.0 volts). This takes about 0.5 second for the circuit shown. By this time the power supply voltages should have reached their final

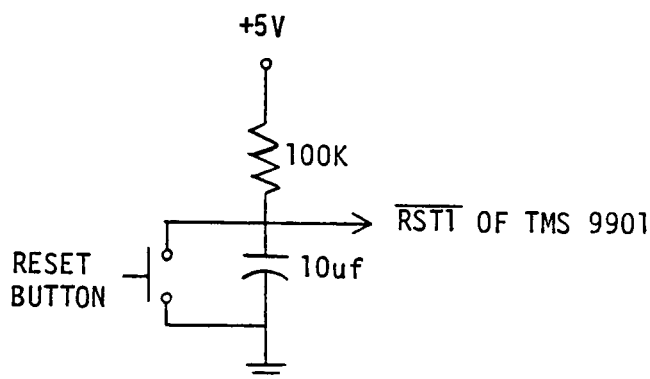


Figure 6.6. Power-up and reset circuit.

values. From location >0000 and >0002 the microprocessor fetches the reset vector and starts to execute the reset routine. The reset routine is also executed whenever the reset button is active. The output of the power-up circuit can be temporarily connected to INT0, INT1, and INT2 of the target connector for testing purpose. After the TMS 9901s are connected to the microprocessor, the output of the power-up circuit is connected to the reset pin, -RST1, of the TMS 9901.

To test the power-up circuit, a simple program shown below is loaded to the emulator memory. No breakpoint nor trace condition is needed. The two modules are set up in the emulator control/internal data interlocked mode and the PC is set to the first IDLE instruction in the main program. Then the reset button is pushed. The traced data should verify the context switching of the reset interrupt.

<u>ADDRESS</u>	<u>CONTENT</u>	<u>COMMENT</u>
>0000	>0400	reset vector
>0002	>0200	
>0100	IDLE	main program
>0102	IDLE	
>0200	RTWP	reset routine

Design and Test Prototype Memory

Even though designing a prototype memory is not necessary as the next designing step, it will be carried out first, since the memory size and configuration of the project is not likely to change. The memory address decoding circuit is designed first. Two criteria are considered in designing a memory address decoder. One of these criteria is meeting the timing requirements of the TMS 9980 and the memory devices. One should never select more than one device to

drive the bus, simultaneously. The TMS 9980 microprocessor memory read and write timing signals are shown in Figures 6.7 and 6.8, respectively [22]. Inputs to the decoder are -MEMEN, DBIN, -WE, and A0-A13 from the microprocessor. Outputs from the decoder to the memory are -CS and -WE. -WE from the microprocessor is connected directly to the -WE of the memory. -CS should not become active (low) until -WE is active (low) in a memory write cycle to avoid a data bus driving conflict.

The other criteria in designing a memory address decoder is the flexibility for future memory expansion. A decoder can be designed using minimum hardware to meet specification without any provision for future expansion. However, for the purpose of illustration, the analyzer memory system is designed for easy modification. Figure 6.9 shows one possible design that meets timing and expandability criteria. The first 1K bytes, >0000 to >03FF, is for an EPROM device and the second 1K bytes, >0400 to >07FF, is for an RAM device. The design allows expansion up to 8K bytes in increments of 1K bytes with minimum hardware required. For each 1K bytes of EPROM or RAM expansion, only one additional OR gate is required.

Two tests, the memory control timing test and the bad memory cell test, are needed to verify that the memory system is functional. The memory control timing test is implemented using the trace module in the emulator control/external data interlocked mode. The data probe connections and trace specification with glitch-latching mode turned on are shown in Figure 6.10(A).

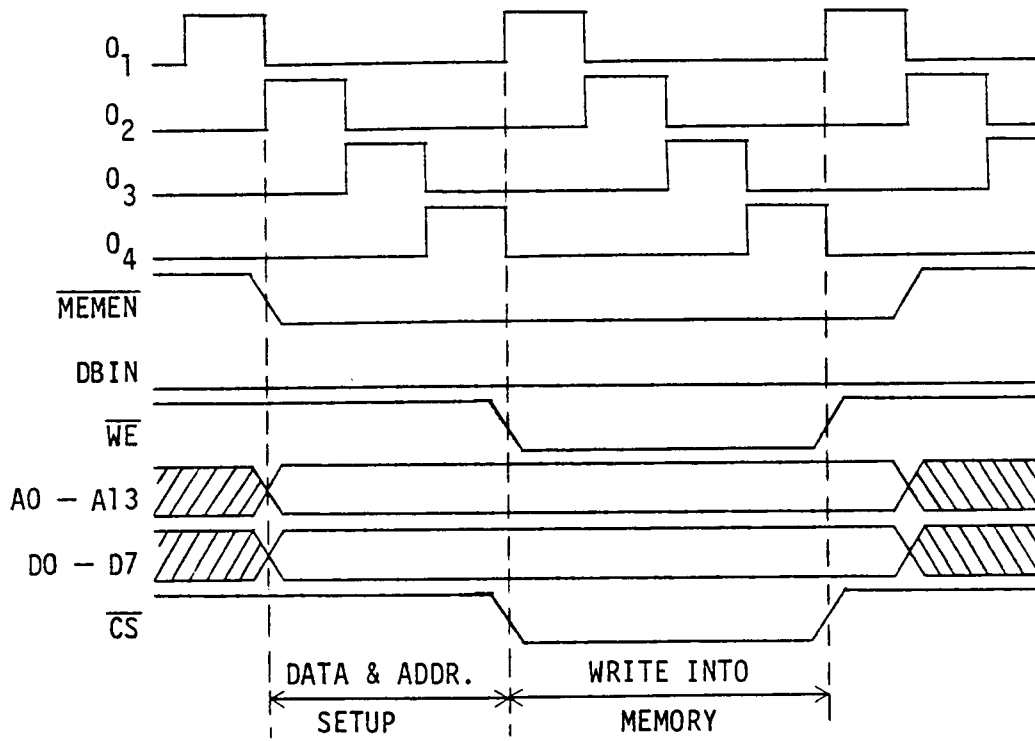


Figure 6.7. TMS 9980 memory write timing.

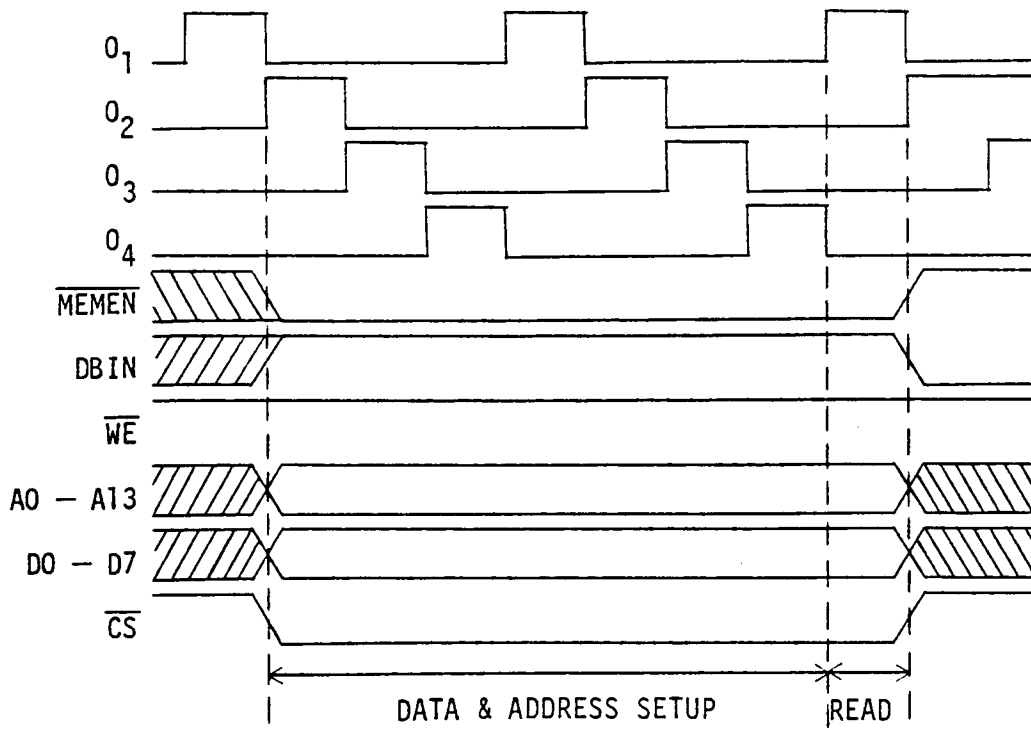


Figure 6.8. TMS 9980 memory read timing.

<u>TRACE PROBES</u>	<u>MEMORY CONTROL LINE</u>	<u>TRACE SPECIFICATION</u>
CLK	CKIN	TTRC(OFF,255,ON)
D0	-MEMEN	
D1	DBIN	
D2	-WE	
D3	-CS	
D4 - D11	D0 - D7	
D12 - D19	A6 - A13	

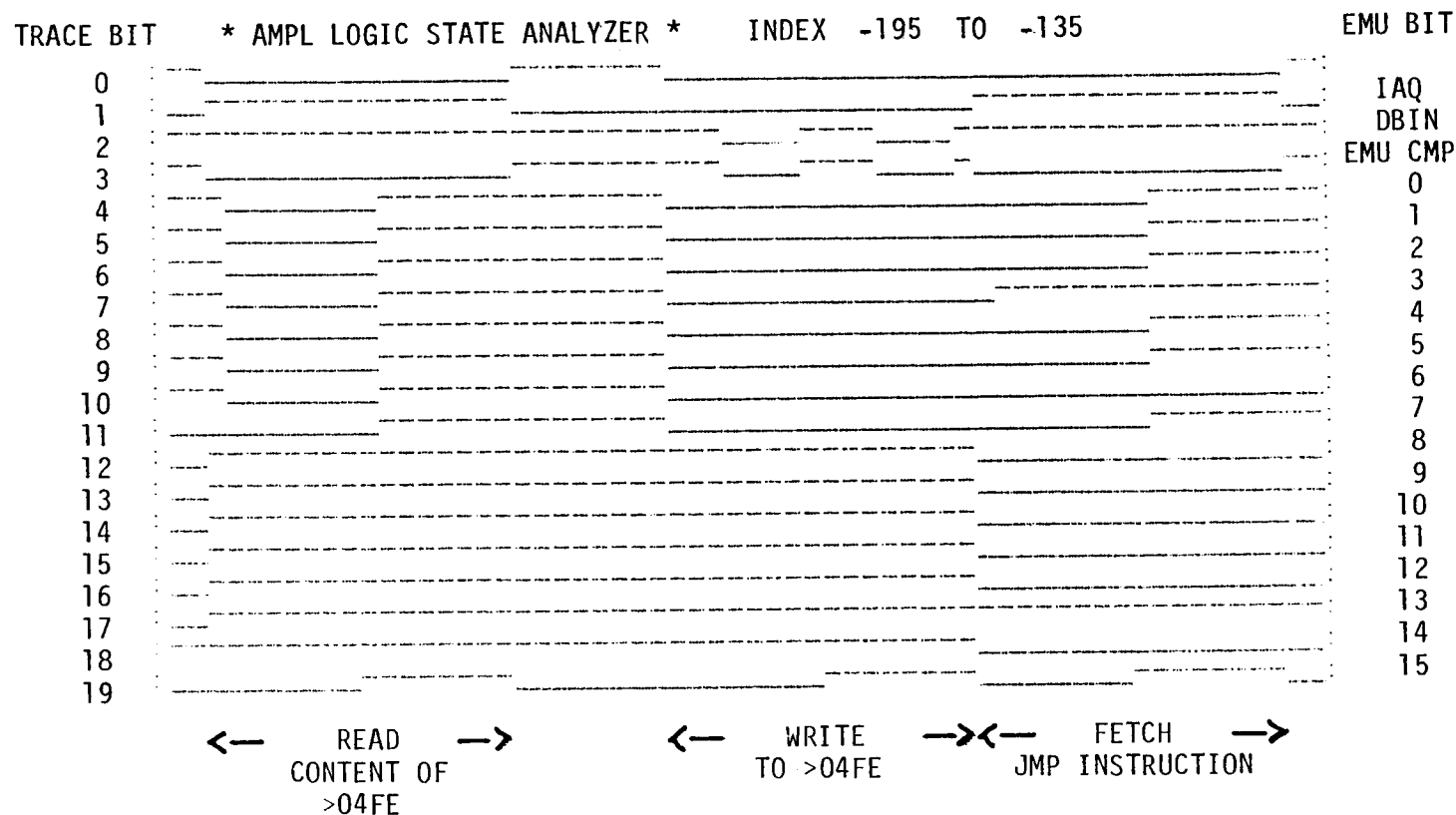
(A) Memory read/write timing test setup

Figure 6.10. Memory read/write timing test.

? LOGIC

LOGIC ANALYZER DISPLAY:

FIRST SAMPLE TO DISPLAY = -195
 FIRST BIT TO DISPLAY = 0
 LAST BIT TO DISPLAY = 19
 SCALE FACTOR(POS: MARKS/SAMPLE = 1
 NEG: SAMPLES/MARK)



(B) Memory read/write timing diagram

Figure 6.10. (continued)

The instructions traced are:

<u>Address</u>	<u>Instructions</u>
>0400	CLR @>4FE
>0404	JMP >0400

A portion of the traced data is displayed in a timing diagram format in Figure 6.10(B). Two read cycles and one write cycle are labeled. Each memory word is accessed by two separate byte operations. No timing conflict exists when compared with the timing diagram of Figures 6.7 and 6.8.

The next test to be conducted is the bad memory cell test. This test was discussed in the previous chapter. After the memory is verified to function correctly, the RAM on the prototype is selected as the target memory. However the memory locations from 0 to >3FF are still mapped into the emulator memory as write-protected simulated ROM. This allows the program to be modified easily.

System Interfacing Devices Design and Test

The decoder for TMS 9901 is designed and tested first. Then the interrupt push buttons and debounce circuits are interfaced to the processor through the TMS 9901. Finally the I/O and the interrupt tests are conducted to verify that these circuits function correctly.

The decoder for TMS 9901 is simpler to design than the memory decoder. These are no -WE and DBIN lines. The functions of these lines are provided by the CRU lines: CRUIN, CRUOUT, and CRUCLK. For a CRU operation, the address lines A0 and A1 must be low and the

-MEMEN line must be inactive (high). A0, A1, and the -MEMEN lines are connected to chip selects of the decoder as shown in Figure 6.11. In this design only two additional 7500 gates are needed for each TMS 9901 added. Up to eight TMS 9901s can be added to the system. This means that up to four histogram analyzer channels can be implemented. Buffers are needed to avoid overloading the microprocessor's bus lines for such an expansion.

A short diagnostic routine shown in Figure 6.12 checks the I/O ports. If an error occurs, the processor is trapped by the IDLE instruction. The result of the test can be checked by the status of the emulator.

If the input and the output are not equal, hardware troubleshooting is required. The development system is set up in emulator control/external data interlocked or independent mode. The trace module data probe's clock line is connected to CRUCLK and a qualifier line is connected to -CE. D0-D19 are connected to suspected faulty data and control lines. The same diagnostic routine is executed. By inspecting the traced data, any unexpected value requires further analysis until the fault is detected. With this test setup the traced data for A0, A1, and A2 should always be low and -MEMEN should always be in the inactive (high) state. For an input operation the test setup would require the external clock line be connected to -03 and the qualifier line be connected to -CE.

Each interrupt debouncing circuit uses two D flipflops. Once an interrupt button is depressed, the corresponding interrupt line

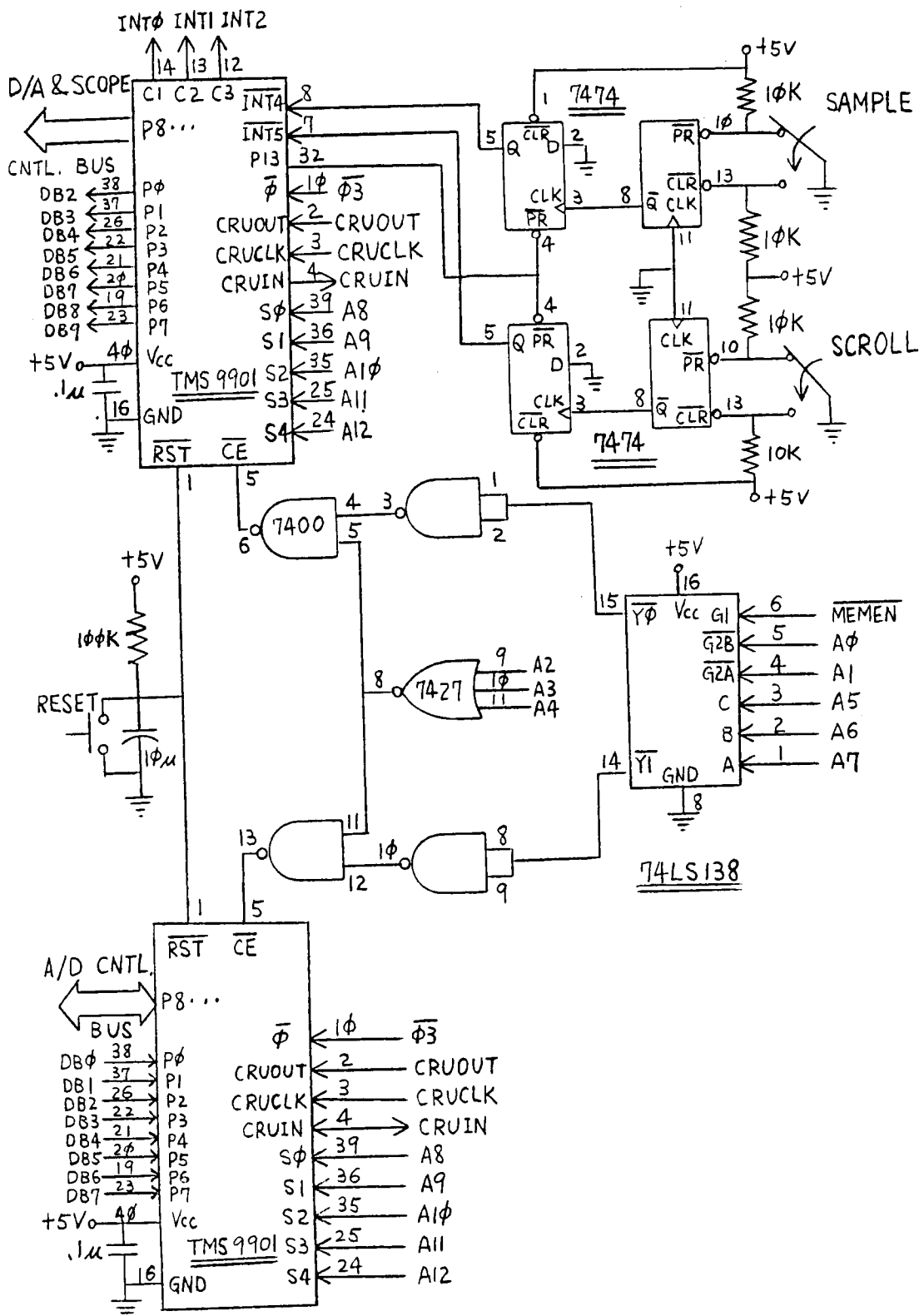


Figure 6.11. I/O and interrupt design.

```

                CLR  R0
                LI   R12,>20
OUTPUT  LDCR  R0, 0
        STCR  R1, 0
        C     R0, R1
        JNE   ERROR
        INC   R0
        JMP   OUTPUT
ERROR   IDLE
                                ERROR TRAP

```

Figure 6.12. I/O diagnostic routine for the TMS 9901.

is pulled low and stays low until software resets it. This ensures proper service of an interrupt.

A digital probe would be most appropriate for trouble shooting the debouncer. However the trace module can be used to provide a timing diagram shown in Figure 6.13 (B) (C). The following routine is used to generate clock pulses for the CLK line of the data probes.

```

CLKPLS  SBO  0
        LI   R1,>8
LOOP    DEC  R1          GENERATE PERIOD OF 134 USEC
        JNE  LOOP
        JMP  CLKPLS

```

Figure 6.11 shows the debounce circuit while the data probe connections and trace specifications are shown below. The first set of the trace specifications in Figure 6/13(A) captures the moment when the button is pushed and the second set of the trace specification captures the moment when the button is released. Before the interrupt button is depressed the trace and the emulator modules are set to running mode.

<u>DATA PROBES</u>	<u>DEBOUNCER LINES</u>	<u>TRACE SPECIFICATIONS</u>
CLK	CRUCLK of TMS 9980	TTRC(EXT,0,ON) TEVT(EXT)
EVT	-PR	TBRK(1,250,+EDGE)
D0	-PR	
D1	-CLR	
D2	-Q	TTRC(EXT,0,ON) TEVT(EXT)
D3	Q	TBRK(1,230,-EDGE)

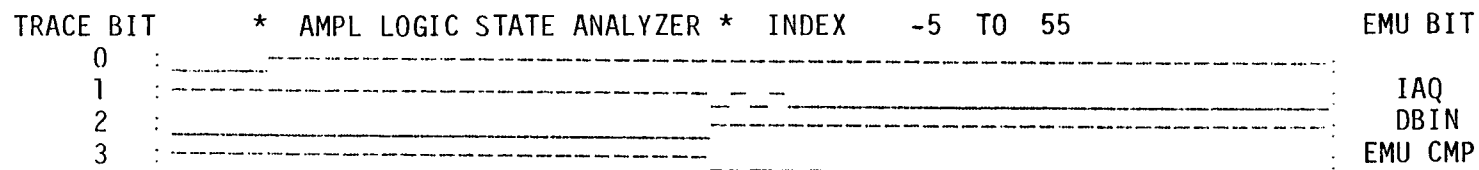
(A) Debounce circuit timing test setup

Figure 6.13. Debounce circuit test.

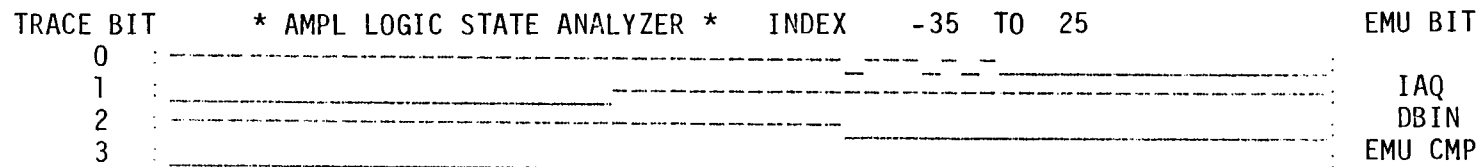
? LOGIC

LOGIC ANALYZER DISPLAY:

FIRST SAMPLE TO DISPLAY = TTBO
 FIRST BIT TO DISPLAY = 0
 LAST BIT TO DISPLAY = 7
 SCALE FACTOR(POS: MARKS/SAMPLE = 1
 NEG: SAMPLES/MARK)



(B) Timing diagram when button pushed



(C) Timing diagram when button released

Figure 6.13. (continued)

Finally a complete hardware and software test of both the interrupt and the I/O ports is accomplished using the diagnostic routine shown in Figure 6.14. Before running the test, the corresponding I/O lines of the two TMS 9901s are connected together. The main test routine sets up the interrupt vectors, enables the interrupt, and then waits for an interrupt. The function of the interrupt routine is to output all possible data through a TMS 9901 and input this data back through the other TMS 9901. A comparison is made between each output and input. Any discrepancy between the data causes the microprocessor to trap in an idle instruction within the interrupt routine. If the test is successful, the microprocessor returns to the main routine and waits for the next interrupt. The result of the test can be checked by identifying the location of the PC.

I/O Devices and Interfacing Design and Test

The next design task is to interface the A/D, the D/A, and the oscilloscope to the microcomputer system. An ADC 0804 has four control lines, -CS, -RD, -WR, and -INTR. The -CS line is tied low to enable the device. -WR, -INTR, and -RD are tied to P8, P9, and P10 of the A/D's TMS 9901, respectively, as shown in Figure 6.15.

A transition from low to high on the write enable line, -WR, starts the A/D. Upon completion of the conversion, the interrupt request line, -INTR, is pulled low by the A/D. The -INTR stays low until the converted data are read. Pulling -RD (the read enable line)

```

ID      'IOTEST'
AORG >400
START  LWPI >500
      LI    R0, >520
      MOV   R0, @>4          LOAD INTERRUPT VECTORS
      MOV   R0, @>8
      LI    R0, SAMPBT
      MOV   R0, @>6
      LI    R0, SCRLBT
      MOV   R0, @>A
      CLR   R12
      SBZ   29              RESET INTERRUPT FF
      SBO   29
      SBO   4
      SBO   5              ENABLE INTERRUPT
      LIMB  3
WAIT   IDLE
      JMP   WAIT          WAIT FOR INTERRUPT
*****
***** INTERRUPT ROUTINE *****
SAMPBT LI    R2,>20        PO OF D/A's TMS 9901
      LI    R3,>60        PO OF A/D's TMS 9901
      JMP   SCRLBT+8
SCRLBT LI    R2,>60
      LI    R3,>20
      CLR   R12
      SBO   0              INIT. D/A's TMS 9901 I/O PINS
      SBZ   15
      LI    R12,>60
      SBO   0              INIT. A/D's TMS 9901 I/O PINS
      SBZ   15
      CLR   R0              INIT. COUNTER & R12
LOOP   MOV   R2,R12
      LDCR  R0,0           OUTPUT 16 BITS
      MOV   R3,R12
      STCR  R1,0           INPUT THE SAME 16 BITS
      C     R0,R1          COMPARE IF SAME
      JNE   ERROR          NOT SAME, ERROR
      INC   R0             GET NEXT OUTPUT
      JNE   LOOP           CONTINUE
      CLR   R12
      SBZ   29              RESET INTERRUPT FF
      SBO   29
      RTWP
ERROR  IDLE              ERROR TRAP

```

Figure 6.14. Interrupts and I/O ports diagnostic routine.

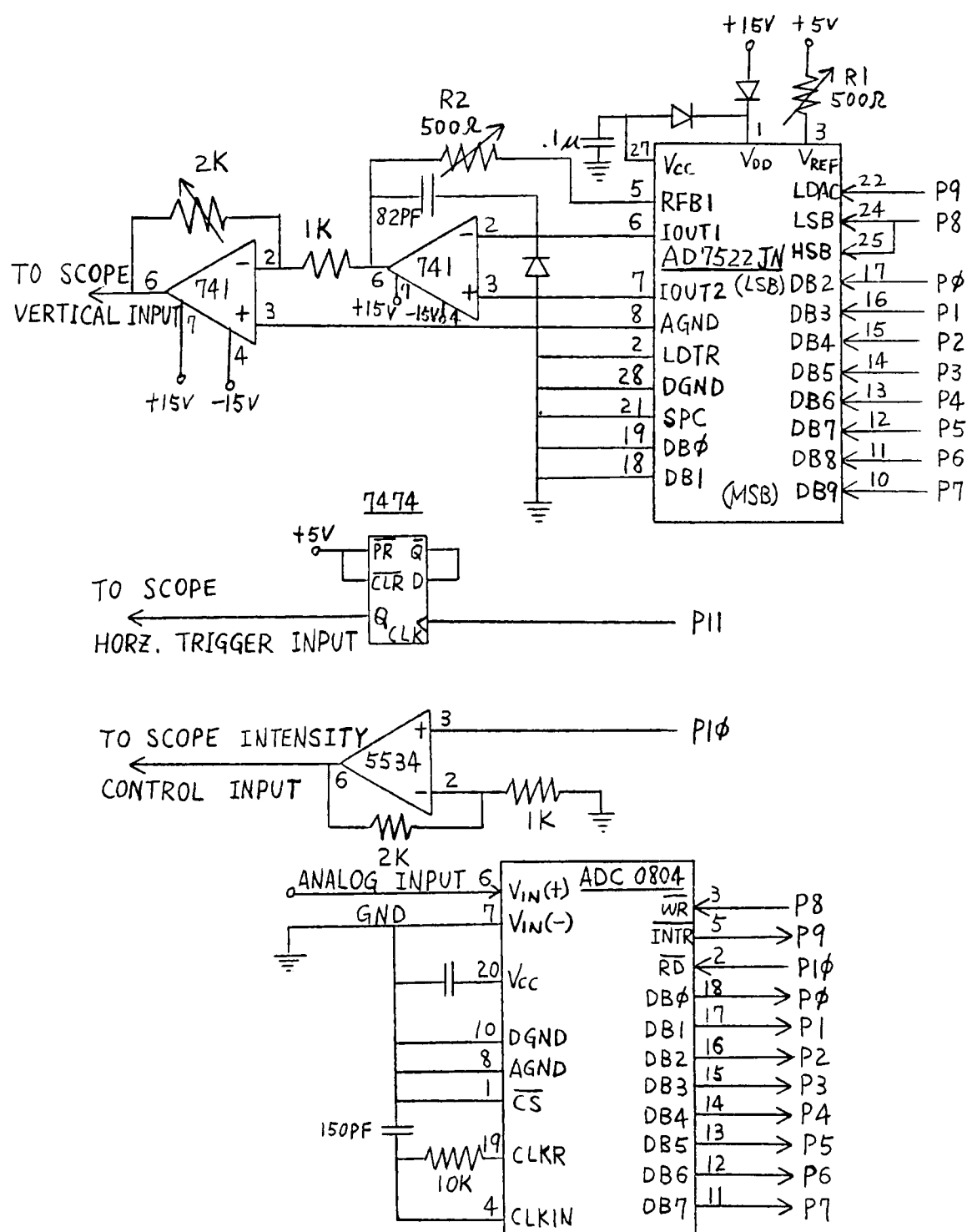


Figure 6.15. D/A, A/D, and scope interfacing design.

high resets -INTR. While -RD is low, the output is enabled. The converted data are read and stored by the microprocessor.

The D/A converter, an AD 7522JN, provides several external control lines but only three lines, LSB, HSB, and LDAC, are essential to this application. The LSB and the HSB are data strobes and are tied together so that all 10 data bits are strobed into the A/D's input buffer on the positive transition of the strobe. The LDAC, an output update line, loads the output register and updates the analog output after 0.5 microsecond. All other control lines are tied low. The LDTR and the SPC lines are tied low for selecting unipolar operation and parallel mode, respectively. The two least significant inputs, DB0 and DB1, are tied low since only 8 out of 10 bits are used here. Zero offset and gain adjustments are implemented by adjusting R1 and R2 according to the manufacturer's specification sheet. Since the output of the D/A is 0 to -5 volt for the unipolar operation, a unity gain inverting op-amp is used to interface the D/A to the scope.

The oscilloscope selected for this project is a Telequipment Oscilloscope D61, which has a Z input for controlling the intensity. This allows an external circuit to plot points on the scope by intensifying the display beam only at the desired screen positions. Applying 10V or 0V to the Z input intensities or blanks the display beam, respectively. A simple noninverting op-amp circuit with a gain of 3 interfaces a TTL signal from the TMS 9901 output line to the scope's Z input.

The horizontal trigger input to the oscilloscope is driven by an I/O line from a T flip-flop. When all of the 256 data points are output, the I/O line toggles the T flip-flop. This resets the horizontal position of the oscilloscope beam and prepares for the next display cycle.

The diagnostic routines for the A/D and the D/A are shown in Figure 6.16. A known DC voltage is connected to the A/D analog input. With the AMPL set in the emulator control/internal data interlocked mode, the routine is single stepped using the SIE library command and content of R0 is examined for the correct value. Using the same single step procedure the D/A is tested. The oscilloscope is used to check for the correct output voltage.

The last two circuits to be checked are the scope's intensity and trigger input interfaces. Outputting 1's and 0's to P10 and P11 of D/A's TMS 9901 and measuring the output voltage of the circuits provide sufficient tests for the interface circuits.

System Software Design

A detailed software flow chart is developed concurrently with the hardware development. Symbols are used where the hardware I/O and memory signals are still undefined. These symbols are assigned temporary values by using the assembler directive, EQU. When the hardware signals are defined, the symbols are assigned the correct values.

A/D diagnostic routine

INIT	LI	R12,>60	SET CRU TO PO OF A/D's TMS 9901
	SBO	10	DISABLE ADC OUTPUT
START	SBZ	8	START A/D CONVERSION
	SBO	8	
LOOP	TB	9	WAIT TILL CONVERSION COMPLETE
	JEQ	LOOP	
	SBZ	10	ENABLE A/D OUTPUT
	STCR	R0,8	INPUT CONVERTED DATA
	SBO	10	DISABLE A/D OUTPUT

D/A diagnostic routine

DACTST	LI	R12,>20	SET CRU TO PO OF D/A's TMS 9901
	LDCR	R0,8	OUTPUT 8 BITS
	SBO	8	STROB DATA INTO D/A
	SBZ	8	
	SBO	9	UPDATE D/A OUTPUT
	SBO	10	TURN ON CRT INTENSITY
	SBZ	10	TURN OFF CRT INTENSITY
	SBZ	9	DISABLE OUTPUT

Figure 6.16. A/D and D/A diagnostic routines.

A detailed flow chart and program for the analyzer project is included in Appendix C. Each step in the flow chart corresponds to assembly language instructions.

Structured programming reduces debugging time by 50% over the traditional unstructured approach. Considering the time required to debug a program often exceeds the time to write the program, 50% saving in time at the debugging stage is significant. In addition, the time required to write a structured program is reduced significantly as one becomes more familiar with the structured approach. An unstructured program is difficult to follow even with modest complexity.

After each software routine is written, it is run and verified using the tracing capability of the emulator or the simulation methods discussed in the previous chapter. If the hardware required for testing the routine is ready, the routine can serve to verify the hardware as well. In such cases the outputs from the routine and inputs received from the hardware would provide useful diagnostic information for detecting faults. By verifying that each routine works correctly with the prototype hardware, the final software and hardware integration phase is simplified.

Software and Hardware Integration Phase

The software and hardware integration phase is the process for bringing together the hardware and software to obtain an intended function. This is an extremely difficult step if the hardware was not designed and tested modularly. The difficulty arises from the

fact that the two untested systems are working simultaneously. Most likely several hardware and software errors occur simultaneously making it extremely difficult to identify and isolate errors even with a development system. Having hardware and software verified before the integration process greatly reduces unnecessary frustration and complexity in the integration phase. Even then, the chance of a successful integration on the first trial is almost zero. Therefore the software should be tested modularly with the prototype hardware. This testing process isolates possible fault areas and simplifies the debugging process.

The first program module to be tested is usually the main routine and its associated subroutines. After the program is loaded into the emulator memory and the write-protect is set up, the program counter is set to the entry point, >14. Program flow is checked first by setting the emulator and trace modules in emulator control/internal data interlocked or independent mode and setting the trace completion breakpoint for 255 samples. Before running the emulator, loop counts in the program should be reduced, so that the resulting trace is not filled with unnecessary data from the loop executions. By examining the program in Appendix C, four instructions need to be modified as listed in Table 6.5.

Several traces are usually required to completely trace a program flow. After the program flow is verified to be correct, the trace completion breakpoint is removed and the PC is reset. The program is executed in a free running mode. Oscilloscope should

display three data points at zero volt. Any errors should be debugged before proceeding further. Possible fault sources are isolated by analyzing the conditions under which errors occur. For example, if the scope is properly adjusted but no display results, the software may not be outputting a signal to the intensity control input of the scope.

Table 6.5. Loop count modifications.

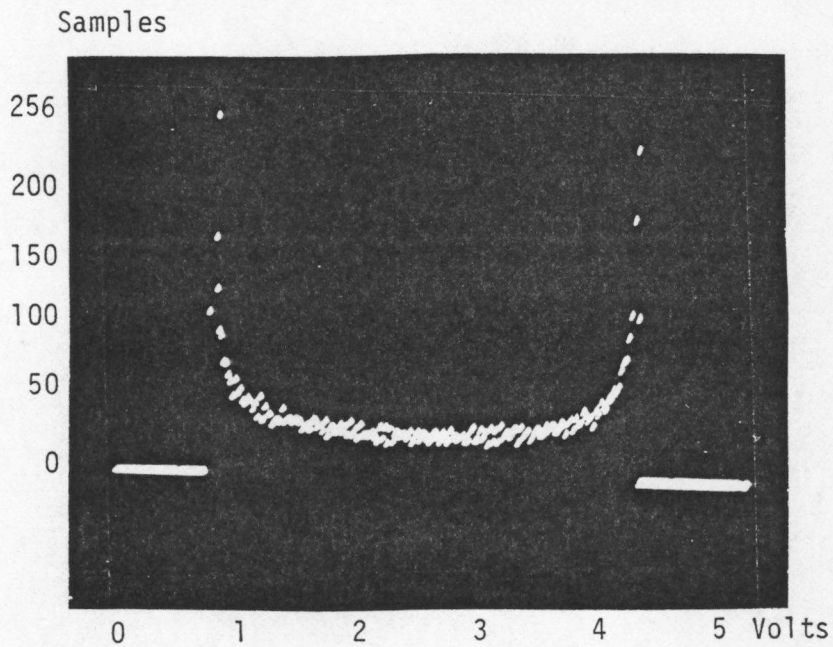
Address	Original Content	Modified Content
>56	>600	>6FD
>6A	>500	>5FD
>F8	>500	>6FD
>106	>500	>5FD

To test the interrupt service routine related to the sample function, the loop count located at address >B8 is set to >5FD. The analog input voltage is adjusted to a known value and the sample/histogram display switch is set to sample mode. Program execution is initiated and the sample interrupt button is depressed to initiate the sampling process. The sample buffer is examined for the actual sampled values. A linearity error may cause the sampled values to be off by a bit. Setting the sample/histogram display switch to histogram mode should display the histogram of the three sampled voltages. The histogram should consist of a single data point if no linearity error is present.

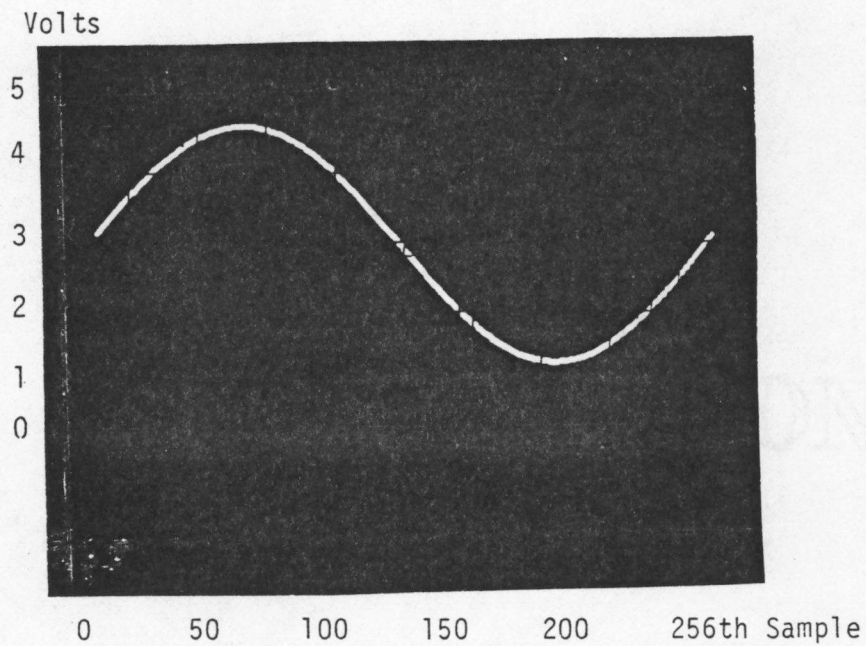
Testing of the scroll interrupt service routine is simpler since no I/O operation is involved. Any error in the routine can be debugged through tracing with the AMPL in the emulator control/internal data interlocked or independent mode. However, before the scroll routine can be tested, the previously modified loop counts should be set to their original values.

After the software/hardware integration phase is verified, the software is programmed into an EPROM device using the EPROM programmer. The EPROM is installed and mapped into the target memory. With the emulator still connected to the prototype hardware, the analyzer is tested again. Upon proper verification, the target connector is unplugged from the prototype system and the TMS 9980 microprocessor is plugged into the microprocessor socket for prototype stand-alone operation. The analyzer should run the first time. If the analyzer fails to run, the power supplies to the microprocessor should be checked, since while in the in-circuit emulation mode the emulator processor uses its own power supplies and is not connected to the prototype power supplies. A high at the HOLD and the READY lines should also be checked, since the emulator has internal pull-up resistors at these pins and the prototype system may not have the resistors connected. Another possible cause of failure to run at this stage is that different clock speeds are used in the in-circuit emulation and the stand-alone operation. Since the 8 MHZ prototype clock is used throughout this example, a failure due to different clock speeds is ruled out.

The histogram analyzer is tested with a generator connected to its input. DC, square, triangular, and sawtooth waveforms are used for testing the analyzer. Figure 6.17(B) shows an input sine wave that was sampled at a frequency of 6 KHZ. The histogram of the sine wave is shown in Figure 6.17(A). As more samples are accumulated the histogram does converges toward the ideal shape.



(A) Histogram of sine wave after 7680 samples are taken.



(B) Digitized sine wave.

Figure 6.17. Histogram analyzer results.

CHAPTER VII

CONCLUSION

Due to the fact that a large percentage of today's digital systems are based around microprocessors, the skill to use them like any other ICs is becoming essential for the digital major electrical engineering students. Therefore a microprocessor-based system design course is essential as a continuation to the presently available microcomputer class at UTK. In addition, during the preparation of this thesis many students in the microcomputer class have expressed their interests in learning to build microprocessor-based systems at chip level.

The AMPL development laboratory made available by the TI is a very powerful tool for developing microprocessor-based systems. A variety of operating modes allows the designer to configure the development system for the hardware trouble shooting, the software debugging, and the hardware/software integration. With a built-in high level language for user-defined commands, the AMPL has a wide variety of applications. Some of these applications are demonstrated in the first two projects of this thesis. These projects familiarize the students with the AMPL development system capabilities.

A microprocessor-based system design example is presented to illustrate the design process involved in using a development system. This modular design and test approach, which reduces time wasted and unnecessary frustration in locating faults, should be used as a guide

to design projects. In addition to the first two projects, four other microprocessor-based system design projects are provided. Any one of the four projects can be completed in a quarter's time yet provide sufficient challenge for a good learning experience.

In conclusion, with the TI minicomputer facility already available, a microprocessor-based system design course and the existing microcomputer course can provide a complete microprocessor education for the students at UTK.

LIST OF REFERENCES

LIST OF REFERENCES

1. M. E. Sloan, "Microcomputer Education in United States," EURO-MICRO NEWSLETTER, Vol. 3, No. 2, pp. 5-7, April 1977.
2. J. D. Nicoud, "Microprocessor Education," EUROMICRO NEWSLETTER, Vol. 3, No. 2, pp. 3-4, April 1977.
3. Jens J. Jonsson, "Microprocessors (uP's) in Electrical Engineering Education Survey," IEEE TRANS. ON EDUC., Vol. E-24, No. 1, pp. 3-8, February 1981.
4. Martha Sloan, R. A. Rohrer, ..., "When Should We Teach Microprocessors?" IEEE TRANS. ON EDUC., Vol. E-22, No. 2, pp. 56-59, May 1979.
5. Russell J. Niederjohn, Gary W. Larsen, Daniel M. Ritt, "A Microcomputer Interfacing Course," IEEE TRANS. ON EDUC., pp. 114-115, May 1977.
6. S. E. Lee, "Microprocessor Analyzers in the Development and Support of Microprocessor Based Systems," Electronics & Power, Vol. 25, No. 3, pp. 193-195, March 1979.
7. Larry Krummel, Gaymond Schultz, "Advances in Microcomputer Development System," Computer, Vol. 10, No. 2, pp. 13-19, February 1977.
8. B. Kline, M. Maerz, P. Rosenfield, "The In-Circuit to the Development of Microcomputer-Based Product," Proceeding of IEEE, Vol. 64, No. 6, pp. 937-942, June 1976.
9. S. Dum, D. Philpott, "In-Circuit Emulation Aids Microprocessor System Design," Electronic Engineering, Vol. 50, No. 615, pp. 87, 89-90, November 1978.
10. M. Conrad, D. King, R. Price, "The Integration of Microcomputer Hardware and Software Development Tools and Techniques," 15th IEEE Computer Society International Conference, Washington, D.C. USA, pp. 201-208, September 6-9, 1977.
11. M. J. Weisberg, "Development-Phase Tools Unravel the Hard/Software Knot," EDN, Vol. 25, No. 6, pp. 187-193, March 20, 1980.
12. B. Barlow, B. Marshall, "Complexities of Capturing and Analyzing Logic Data in a Real-Time Environment," 1978 MIDCON Technical Papers, Dallas, Texas USA, December 12-14, 1978, pp. 31/2/1-4.

13. Model 990 Computer AMPL Microprocessor Prototyping Laboratory Operation Guide, Texas Instrument Inc., 1980.
14. Model 990 Computer DX10 Operating System Production Operation Manual, Vol. 2, Texas Instrument Inc., 1981.
15. Model 990 Computer DX10 Operating System Developmental Operation Manual, Vol. 4, Texas Instrument Inc., 1981.
16. Model 990 Computer 990/10 and 990/12 Assembly Language Reference Manual, Texas Instrument Inc., 1981.
17. Model 990 Computer AMPL PROM Programming Utility User's Guide, Texas Instrument Inc., 1979.
18. Model 990 Computer AMPL Microprocessor Prototyping Laboratory Tutorial, Texas Instrument Inc., 1980.
19. AMPL Development System Student Guide, Semiconductor Group Technology Center, Texas Instrument Inc., 1979.
20. Model 990 Computer AMPL Microprocessor Prototyping Laboratory Command Library User's Guide, Texas Instrument Inc., 1980.
21. Model 990 Computer Logic-State Trace Data Module Installation and Operation Manual, Texas Instrument Inc., 1978.
22. Don L. Cannon, "Fundamentals of Microcomputer Design System Hardware and Software," Texas Instrument Learning Center, 1982.
23. TMS 990/U89 Microcomputer User's Guide, Texas Instrument Inc., 1981.

APPENDIXES

APPENDIX A

EMULATOR AND TRACE MODULE PRIMITIVE COMMAND SYNTAX

EMULATOR COMMANDS/VARIABLES

Syntax	Definition
$EINT(EM0n \left[\begin{matrix} 1 \\ 0 \end{matrix} \right] [1:TM0n])$	Initialize emulator EM0n and optionally trace module TM0n. Set clock: 1 = emulator clock 0 = prototype Default clock value is 1.
ECLK	Contains processor clock value: 1 = emulator clock 0 = prototype
ETYP	Contains processor type: -1 = TMS9940 0 = SBP 9900 1 = TMS 9900 2 = TMS 9900
$ETRC(\begin{matrix} MA \\ IAQ \\ IAQX \end{matrix} [count], low[, high])$	Set trace conditions for addresses within trace range <low> to <high>: MA — all memory accesses IAQ — instruction acquisitions IAQX — all memory cycles for IAQ Set breakpoint after <count> samples.
$EBRK(\begin{matrix} MA \\ IAQ \\ MR \\ MW \\ ILLA \end{matrix} [addr], [addr]...)$	Set breakpoint at <addr> for specified memory access: MA — any memory access IAQ — instruction acquisition MR — memory read MW — memory write ILLA — illegal address (9940 only).
ERUN	Start emulation at current PC, WP, ST.

EMULATOR COMMANDS/VARIABLES (Continued)

EST	Contains emulation status (3 LSBits): hold, idle, running
EHLT	Halt emulation, return status.
ETBO	Contains oldest trace sample index.
ETBN	Contains newest trace sample index.
ETBindex	Display address bus of <index>ed trace sample
$ETBHindex(\begin{matrix} MR \\ MW \\ IAQ \end{matrix} [i])$	Display bus control lines of <index>ed trace sample. Return nonzero value if sample type specified TRUE.

TRACE MODULE COMMANDS/VARIABLES

Syntax	Definition
$TINT(TM0n)$	Initialize trace module TM0n.
$TTRC(\begin{matrix} INT \\ 00 \\ 01 \\ 02 \\ 03 \end{matrix} [count], \begin{matrix} ON \\ OFF \end{matrix} [i])$	Set qualifiers (00-03) for lines D0-D19: + On — On high - On — On low INT — use trace module clock OFF — 00-03 are don't cares. Set breakpoint after <count> samples (0-255). Set latch mode for D0-D3: ON — enable latch mode OFF — disable latch mode.
$TBRK(\begin{matrix} count \\ OFF \end{matrix} [delay], \begin{matrix} INV \\ OFF \end{matrix} [EDGE])$	Define event/delay breakpoints. Set event counter to <count> (1->FFFF); 0 or OFF - disable breakpoints. Set delay counter to <delay> (1-255); 0 or OFF - disable delays. Set count mode: INV — inverted EDGE — transitions on low to high INV + EDGE — transitions on high to low
$TEVT(\begin{matrix} 00 \\ 01 \\ 02 \\ 03 \end{matrix} [pattern], [mask])$	Define event. Set conditions for lines D0-D3:

TRACE MODULE COMMANDS/VARIABLES (Continued)

TRUN	Start trace module tracing.
TST	Contains trace module status (3 LSBits): event occurred, trace module full, tracing
THLT	Halt trace module, return status.
TNE	Contains number of events since last TRUN
TNCE	Contains number of event count overflows.
TTBO	Contains oldest trace sample index.
TTBN	Contains newest trace sample index.
$TTBHindex(\begin{matrix} 00 \\ 01 \\ 02 \\ 03 \end{matrix} [i])$	Display D0-D3 of <index>ed trace sample
TTBindex	Display data bus (D4-D19) of <index>ed sample.

APPENDIX B

MICROPROCESSOR-BASED SYSTEM DESIGN

PROJECT SPECIFICATIONS

PROJECT #1: MEMORY TEST

Design and test a microprocessor memory system for the TMS 9980 Microprocessor. The memory system should be easily expandable in increments of 1K bytes. One 1K byte of the memory is to be ROM (TMS 2708) and another 1K byte of the memory is to be RAM (TMS 4045).

The memory system is to be tested under control of the AMPL development system. Three types of test are required: a dead cell test, a data path test, and a 50 msec refresh test. Even though the memory test can be implemented using only the AMPL language, it will not be permitted here, since an excessive amount of time is required for any significant amount of memory to be tested. An assembly language memory test routine should be written to supplement the AMPL procedure to reduce the memory test time. The AMPL memory test procedure should prompt the user for the starting and the ending memory addresses to be tested. The portion of memory where the assembly language memory test routine will reside should be first tested by the AMPL memory test procedure. The memory test should complete without any further intervention from the operator. Results of the memory test should be displayed on the terminal as well as listed to a file for printout purpose. The result should be labeled according to the type of memory test. Memory test value passed as well as failed should be included in the result. Beside each bad memory location, display the input value, the expected value, and the bits that are bad.

The following parts are provided for this project:

<u>Parts</u>	<u>Description</u>	<u>Quantity</u>
TMS 9980	microprocessor	1
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2

PROJECT #2: ASSEMBLY PROGRAM SIMULATION

Write an assembly language simulation program using the AMPL language. The purpose of this project is to check the functionality of software easily and independently of any prototype hardware. Since only the inputs and the interrupts affect the flow of the software, the simulation should prompt the user for inputs, when required, from the terminal and be able to simulate and trace an interrupt through the terminal. The simulation should allow the user to specify trace count and trace range of the simulation.

In a subsequent project the simulator is useful for the initial testing of software. An example of the simulation program is available by using the following COPY command.

```
COPY(^.S$SYSLIB.AMPL$CMD.CSAP^)
```

The use of the simulation program is explained in the CSAP, computer simulated assembly program, tutorial.

PROJECT #3: HISTOGRAM ANALYZER

Using the tested memory of Project No. 1, design a TMS 9980 based histogram analyzer. The input to the histogram analyzer is a signal from 0 to +5 volts. Output of the histogram analyzer is plotted on an oscilloscope. The voltages and the frequency of occurrences of the corresponding voltage are plotted on the horizontal and the vertical axes, respectively. The graph will consist of 256 dots. Each dot represents a possible discrete sampled voltage and its amplitude is proportional to the number of times that voltage was sampled. Since the accuracy of a histogram is proportional to the total number of samples taken, the analyzer should provide an accumulative result of the maximum possible samples without overflowing. Also the analyzer should provide a means for locating and examining any particular data point. The following parts are provided for this project:

<u>Parts</u>	<u>Description</u>	<u>Quantity</u>
TMS 9980	microprocessor	1
TMS 9901	programmable parallel I/O device	2
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2
ADC 0804	A/D converter	1
AD 7522JN	D/A converter	1
CY 8.000	8.000 MHZ crystal	1

PROJECT #4: DMA HISTOGRAM ANALYZER

Implement Project No. 3 using a DMA controller. The following parts are provided for this project:

<u>Parts</u>	<u>Description</u>	<u>Quantity</u>
TMS 9980	microprocessor	1
TMS 9901	programmable parallel I/O device	2
TMS 9911	DMA controller	1
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2
ADC 0804	A/D converter	1
AD 7522JN	D/A converter	1
CY 8.000	8.000 MHZ crystal	1

PROJECT #5: SYSTEM ALARM CLOCK

Using the tested memory of Project No. 1 design a TMS 9980 based computer system that interfaces with a data terminal through the EIA RS232 standard port. A monitor program should be written to accept and interpret a command string of variable length. The system command repertory should be easily expandable to include new commands and to modify the old ones. Since the system clock command is a very basic command that exists in most computer systems, a system clock with the alarm feature should be implemented in this project. The computer should keep track of the time after it has been turned on. The system clock time should be easily reset. When an invalid command string or value is entered, an error message should be displayed. The following alarm capabilities should be provided:

1. The alarm can be turned on or off.
2. The alarm time and message should be easily entered.
3. When requested the alarm time and a message is displayed.
4. The alarm should be implemented through the speaker on the terminal.

The following parts are provided for this project:

<u>Parts</u>	<u>Description</u>	<u>Quantity</u>
TMS 9980	microprocessor	1
TMS 9902	asynchronous communication controller	1
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2
SN 75188	line driver	1
SN 75189	line receiver	1
CY 8.000	8.000 MHZ crystal	1

PROJECT #6: PIANO KEYBOARD SYNTHESIZER

The purpose of this project is to implement a piano keyboard synthesizer on a data terminal. A synthesizer should be capable of playing five octaves. As long as a key is depressed, a note is played. When the key is released the note should cease within 0.5 second. Terminal key should be provided if a note is to be terminated immediately.

A synthesizer should also be able to record and replay the music as it is entered. The current mode of operation (piano, record, or replay) of the synthesizer should be indicated on the terminal. The mode of the synthesizer can be altered at anytime by pressing the designated keys on the terminal. When the record buffer is full, a message should be displayed on the terminal.

The following parts are provided for this project:

<u>Parts</u>	<u>Description</u>	<u>Quantity</u>
TMS 9980	microprocessor	1
TMS 9901	programmable parallel I/O device	1
TMS 9902	asynchronous communication controller	1
TMS 2708	1K X 8 EPROM	1
TMS 4045	1K X 4 RAM	2
SN 75188	line driver	1
SN 75189	line receiver	1
CY 8.000	8.000 MHZ crystal	1

APPENDIX C

MICROPROCESSOR-BASED SYSTEM DESIGN PROJECT SOLUTIONS

AUTOMATED MEMORY TEST PROJECT SOLUTION

```

PROC  WRITE(3,1)          ..WRITE SPECIFIED TEST CONTENT TO MEMORY RANGE
FOR LOC 1=ARG 1 TO ARG 2 BY 2 DO @LOC 1=ARG 3
END
PROC  CHECK(3,2)          ..LOC 1 CONTAIN NO. OF ERRORS FOUND
LOC 1=0
FOR LOC 2=ARG 1 TO ARG 2 BY 2 DO ..CHECK COMPLETE MEMORY RANGE
    IF @LOC 2 NE ARG 3 THEN BEGIN .. IF ERROR OCCUR
        LOC 1=LOC 1+1
        'MEMORY ERROR ADDRESS: ';LOC 2:H;
        'EXPECTED: ';ARG 3:H;
        'ACTUAL: '; @LOC 2:H;
        'ERROR BIT: ';@LOC 2 XOR ARG 3:H;NL
    END
IF LOC 1 EQ 0          .. IF TEST PASSED FOR SPECIFIED MEMORY RANGE
    THEN BEGIN
        'MEMORY TEST - PASS,VALUE: '; ARG 3:H
    END
    ELSE BEGIN ..TEST FAILED
        'MEMORY TEST - FAIL - ';
        'NUMBER OF ERRORS: ';LOC 1:D
    END
NL
END
PROC  AMPLMT(2,2)
'MEMORY TEST(FIRST >100 LOCATIONS): FROM ';ARG 1:H; 'TO ';ARG 2:H;NL
'WRITE ALL ZERO TEST: ';NL
WRITE(ARG 1,ARG 2,0)          ..WRITE ZERO'S
CHECK(ARG 1,ARG 2,0)          ..CHECK
'WRITE ALL ONE TEST: ';NL
WRITE(ARG 1,ARG 2,>FFFF)      ..WRITE ONE'S
CHECK(ARG 1,ARG 2,>FFFF)      ..CHECK
'WALKING ONE TEST: ';NL
LOC 1=>8000
FOR LOC 2=0 TO 15 DO BEGIN .. WALKING ONE'S
    WRITE(ARG 1,ARG 2,LOC 1)
    CHECK(ARG 1,ARG 2,LOC 1)
    LOC 1 =LOC 1 >>1
END
'WALKING ZERO TEST: ';NL
LOC 1=>7FFF
FOR LOC 2=0 TO 15 DO BEGIN .. WALKING ZERO'S
    WRITE(ARG 1,ARG 2,LOC 1)
    CHECK(ARG 1,ARG 2,LOC 1)
    LOC 1=LOC 1<>1
END
'REFRESH >AAAA TEST: ';NL
WRITE(ARG 1,ARG 2,>AAAA)      ..REFRESH TEST(>AAAA)
WAIT(1)                      ..WAIT FOR 50 MSEC
CHECK(ARG 1,ARG 2,>AAAA)
'REFRESH >5555 TEST: ';NL
WRITE(ARG 1,ARG 2,>5555)      ..REFRESH TEST(>5555)
WAIT(1)                      ..WAIT FOR 50 MSEC
CHECK(ARG 1,ARG 2,>5555)
END
PROC  ALMT(2,0)

```

```

NL;NL      ... SKIP TWO SPACE
'MEMORY TEST(REST OF MEMORY RANGE): FROM ' ; ARG 1: H; 'TO ' ; ARG 2: H; NL
'WRITE ALL ZERO TEST: ' ; NL
R0=ARG 1      ... SET STARTING ADDRESS
R1=ARG 2      ... SET END ADDRESS
REPEAT BEGIN
TRUN;        ... START TRACER AND EMULATOR
ERUN;
REPEAT NULL      ... WAIT TILL EMULATOR IDLE
UNTIL EST EQ 3;
MDR=EHLT;      ... HALT EMULATOR
IF PC EQ ARG 1->8B OR PC EQ ARG 1->76 THEN WAIT(1) ... REFRESH DELAY
  ELSE IF PC NE ARG 1->52      ... IDLE AT TEST COMPLETE
    THEN BEGIN IF R2 EQ 0      ... ERROR HAS NOT OCCUR
      THEN BEGIN
        'MEMORY TEST - PASS, VALUE: ' ; R4: H; NL;
      END
      ELSE BEGIN      ... ERROR HAS OCCUR
        'MEMORY TEST - FAIL - ' ;
        'NUMBER OF ERRORS: ' ; R2: D; NL;
      END
      CASE R11 OF      ... DISPLAY TYPE OF TEST
        ... BY CHECKING RETURN ADDRESS
        ARG 1->D6::BEGIN 'WRITE ALL ONE TEST: ' ; NL;
        END
        ARG 1->C8::BEGIN 'WALKING ONE TEST: ' ; NL;
        END
        ARG 1->B8::BEGIN
        IF R4 EQ 1 THEN BEGIN ... IF THE TEST COMPLETE
        'WALKING ZERO TEST: ' ; NL;
        END
        END
        ARG 1->A0::BEGIN      ... IF THE TEST COMPLETE
        IF R4 EQ >FFFF THEN BEGIN
        'REFRESH >AAAA TEST: ' ; NL;
        END
        END
        ARG 1->84::BEGIN 'REFRESH >5555 TEST: ' ; NL;
        END
        END
      END
    ELSE IF (@R3 XOR R4) NE 0 THEN BEGIN ... IDLE AT ERROR
      ... DOUBLE CHECK IF ERROR REALY OCCUR
      ... VACANT MEMORY LOCATION MAY CAUSE WRONG ERROR DETECT
      'MEMORY ERROR ADDRESS: ' ; R3: H;
      'EXPECTED: ' ; R4: H;
      'ACTUAL: ' ; @R3: H;
      'ERROR BIT: ' ; @R3 XOR R4: H; NL;
      END
      ELSE R2=R2-1 ... NO ERROR OCCUR DEC. ERROR COUNT
    END UNTIL PC EQ ARG 1->70      ... REPEAT UNTIL END OF TEST
  END
END
... MEMORY TEST PROCEDURE
PROC MEMTST(2,1)
IF ARG 1 MOD 2 NE 0
  THEN ARG 1 =ARG 1 - 1 ... SET START ADDRESS TO WORD BOUNDARY
IF ARG 2 MOD 2 NE 0
  THEN ARG 2 =ARG 2 - 1 ... SET END ADDRESS TO WORD BOUNDARY
IF ARG 2 -ARG 1 LT 0
  THEN 'BOUNDARY SPECIFICATION ERROR'
  ELSE BEGIN
    LIST('LIST.LIST'); ... OPEN A LISTING FILE FOR ERROR MESSAGE
  
```

```

'MEMORY TEST RANGE: FROM ' ; ARG 1: H; 'TO ' ; ARG 2: H; NL; NL; NL
IF ARG 2 -ARG 1 LE >100
  THEN AMPLMT(ARG 1, ARG 2) .. USE AMPL LANGUAGE TO TEST MEMORY
  ELSE BEGIN
    LOC 1=ARG 1 +>100; .. SET END ADDRESS FOR AMPLMT PROC
    AMPLMT(ARG 1, LOC 1); .. TEST FIRST SIZE WORDS USE AMPL LANGUA
    LOAD('OBJ. ALMT', ARG 1); .. LOAD ASSEMBLY LANGUAGE
                                .. MEMORY TEST PROGRAM
    PC=ARG 1+>20; .. SET PC
    WP=ARG 1; .. SET WP
    ARG 1 =LOC 1 +2; .. SET NEW START ADDRESS
    ALMT(ARG 1, ARG 2); .. CALL ASSEMBLY LANGUAGE MEMORY TEST PROG
  END

```

```

LIST(EOF) .. CLOSE LISTING FILE
END

```

```

END
FORM

```

```

MEMTST
'; MEMORY TEST';
'Starting Address' =;
'Ending Address' =;

```

```

END

```

```

*****
***** ASSEMBLY TEST ROUTINE FOR AMPL MEMORY TEST PROCEDURE *****
*****
IDT,ALMT,
OPTION SYMT
PROVIDES SYMBLE TABLE
*****
ALMTWP BSS 32
*****
WRITE ALL ZERO'S TO MEMORY RANGE *****
*****
CLEAR ERROR NO. REGISTER TO ALL 0
SET TEST PATTEN REGISTER TO ALL 0
CALL WRITE SUBROUTINE
CALL CHECK SUBROUTINE
WAIT FOR AMPL TO SHOW TEST MESSAGE
*****
WRITE ALL ONE'S TO MEMORY RANGE *****
*****
CLR R2
SETO R4
BL @WRITE
BL @CHECK
IDLE
WAIT FOR AMPL TO SHOW TEST MESSAGE
*****
CLR R2
MOV R4,R4
SRL R4,1
MOV R4,R4
JNE NXTONE
IF R4 NE 0 CONTINUE WALK ONE'S TEST
*****
WALKING ONE'S TO MEMORY RANGE *****
*****
CLR R2
LI R4,>8000
SET TEST PATTEN REGISTER TO >8000
BL @WRITE
BL @CHECK
IDLE
WAIT FOR AMPL TO SHOW TEST MESSAGE
*****
CLR R2
LI R4,>8000
SET TEST PATTEN REGISTER TO >8000
BL @WRITE
BL @CHECK
IDLE
NEXTONE BL @WRITE
BL @CHECK
WAIT FOR AMPL TO SHOW TEST MESSAGE
CLEAR ERROR COUNT REGISTER FOR NEXT TEST
OBTAIN NEXT TEST CONTENT
JNE NXTZER
IF R4 NE >7FFF CONTINUE WALK ZERO'S TEST
*****
>AAAA REFRESH TEST *****
*****
CLR R2
LI R4,>AAAA
SET TEST PATTEN REGISTER TO >AAAA
BL @WRITE
BL @CHECK
IDLE
WAIT FOR AMPL TO TIME OUT 50 MSEC
REFSHA BL @CHECK
IDLE
WAIT FOR AMPL TO SHOW TEST MESSAGE
*****
>5555 REFRESH TEST *****
*****
CLR R2
LI R4,>5555
SET TEST PATTEN REGISTER TO >5555
*****

```

```

      BL      @WRITE
      IDLE
REFSH5 BL      @CHECK      WAIT FOR AMPL TO TIME OUT 50 MSEC
END      IDLE      WAIT FOR AMPL TO SHOW TEST MESSAGE
*****
*****      WRITE SUBROUTINE      *****
*****
WRITE MOV      R0,R5      R5 IS TEMPORARY COPY OF R0
NXTWRT MOV     R4,*R5+    WRITE TEST CONTENT TO MEMORY RANGE
      C        R5,R1
      JLE     NXTWRT      CONTINUE IF NOT END OF MEMORY RANGE
      B        *R11
*****
*****      CHECK SUBROUTINE      *****
*****
CHECK MOV      R0,R5
NXTCHK C        R4,*R5+    CHECK IF THERE IS ERROR OR NOT
      JNE     ERROR
ENDTST C        R5,R1      CHECK IF END OF TEST OR NOT
      JLE     NXTCHK
      B        *R11
ERROR MOV      R5,R3      ERROR ROUTINE
      DECT   R3      LOAD MEMORY FAILURE ADDRESS REGISTER
      INC    R2      INCREMENT ERROR COUNT
IDLER  IDLE
      JMP     ENDTST      WAIT FOR AMPL TO SHOW ERROR MESSAGE

```

REAL-TIME SOFTWARE SIMULATION PROJECT SOLUTION

```

PROC TB(1,0)
  ST = ST AND >DFFF      .. SET ST EQ. BIT TO ZERO
  IF ARG 1 NE 0 THEN ST = ST XOR >2000      .. SET ST EQ. BIT TO ONE
END
FORM TB
  ':TB INSTRUCTION: ';
  'INPUT '='0';
END
FUNC STCR(1,3)
  NL
  LOC 1=(@VALUE2 AND >03C0)/>40  .. DETERMINE NO. OF INPUT BITS REQUIRED
  IF LOC 1 NE 0 THEN BEGIN
    LOC 2=ARG 1
    FOR LOC 3=1 TO 16 DO BEGIN .. DETERMINE NO. OF INPUT BITS INPUT
      IF LOC 2 LT 0 THEN BEGIN
        LOC 3=16;
        ESCAPE;
      END
      LOC 2=LOC 2/2
      IF LOC 2 EQ 0 THEN ESCAPE
    END
    IF LOC 3 GT LOC 1 THEN BEGIN .. ERR IF INPUT BITS IS GT. REQUIRED
      'ERROR: PLEASE ENTER ONLY ' LOC 1:D3 ' BITS !';NL;
      RETURN 1;  .. RETURN 1 TO INDICATE INCORRECT INPUT DATA
    END
  END
  IF (LOC 1 NE 0) AND (LOC 1 LE 8) THEN      .. CORRECT INPUT DATA
    @ETB(ETBN)=(@ETB(ETBN) AND >00FF) OR (ARG 1 * >0100)
    ELSE @ETB(ETBN)=ARG 1
  RETURN 0  .. RETURN 0 TO INDICATE CORRECT INPUT DATA
END
FORM STCR
  ':STCR INSTRUCTION: ';
  'INPUT '='0';
END
PROC CSAP(4,3)
  IF ARG 1 EQ 0 THEN BEGIN
    LOC 2 = >3400;  .. STORE STCR OP CODE
    LOC 3 = >FC00;  .. STORE STCR OP CODE BIT MASK
  END
  ELSE BEGIN
    .. STORE TB OP CODE
    LOC 2 = >1F00;  .. STORE TB OP CODE BIT MASK
    LOC 3 = >FF00;
  END
  TTRC(Q3)      .. TRACE DATA QUALIFIED BY ETRC
  TBRK(1)      .. BREAK ON ONE EVENT COUNT
  TEVT(D1,LOC 2,LOC 3)  .. SPECIFY EVENT AS STCR OR TB
  ETRC(IAQX,ARG 2,ARG 3,ARG 4)  .. SPECIFY TRACE COUNT &TRACE BOUNDARY
  EBRK(MA)      .. DISABLE EMULATOR BREAKPOINT
  TRUN      .. START TRACER AND EMULATOR
  ERUN
  REPEAT NULL UNTIL EST XOR 1  .. WAIT FOR PROCESSOR HALT OR IDLE
  IF EST AND 2 THEN BEGIN
    NL      .. PROCESSOR IDLE
    'IDLE INSTRUCTION ENCOUNTERED';NL;
    'PROCESSOR HALTED';NL;
  
```

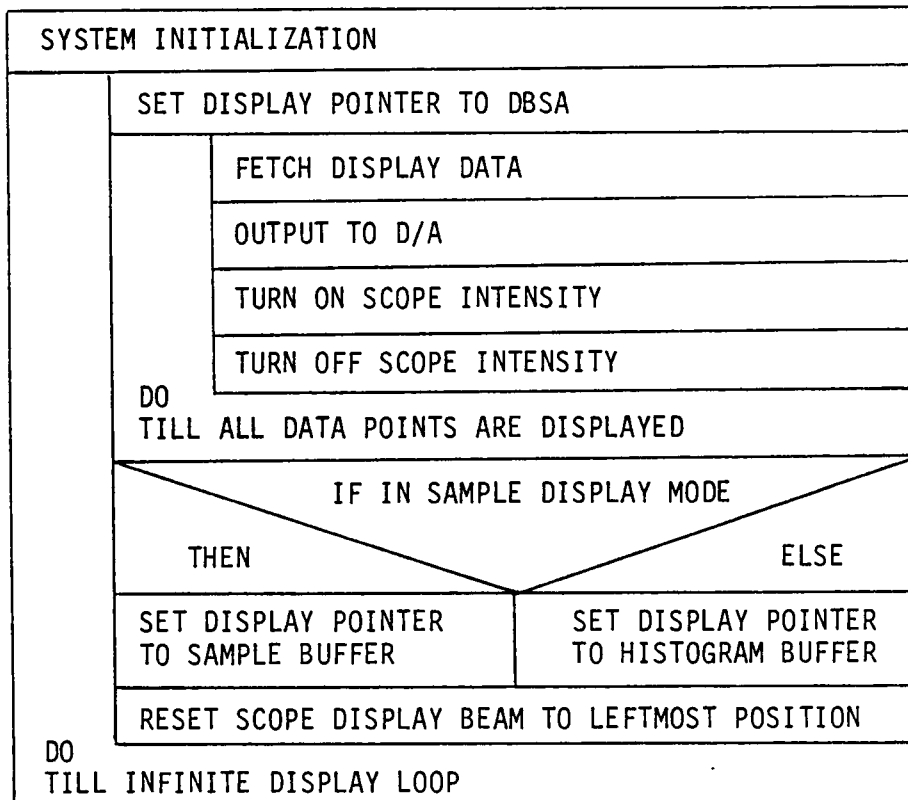
```

MDR=EHLT;
RETURN;
END
ELSE BEGIN
LOC 1=ETBN
WHILE ETBH(LOC 1, IAG) EQ 0 DO LOC 1=LOC 1 - 1
VALUE2 = ETB(LOC 1)    .. FETCH STCR OR TB FROM BUFFER
IF (@VALUE2 AND >FC00) EQ >3400 THEN BEGIN
    NL                .. STCR IS ENCOUNTERED
    'STCR INSTRUCTION ENCOUNTERED AT 'VALUE2: H4; NL
    'PLEASE ENTER ONLY '(@VALUE2 AND >03C0)/>40: D3' BITS'; NL
    WHILE STCR DO MDR=0 .. CALL STCR FUNC
    END                .. AND REQUEST INPUT FROM TERMINAL
    ELSE IF (@VALUE2 AND >FF00) EQ >1F00 THEN TB
    END                .. CALL TB PROC TO REQUEST INPUT
CSAP
END
FORM CSAP
': INPUT INSTRUCTION AND TRACE RANGE: ';
':      ';
'STCR OR TB INSTRUCTION(STCR=0; TB=1) '='0';
'TRACE COUNT(1-255, 0=INFINITE) '='0';
'LOW BOUNDARY OF TRACE RANGE '='0';
'HIGH BOUNDARY OF TRACE RANGE '='>FFFF';
END
PROC INTR(1, 12)
MDR=EHLT
IF (ST AND 15) GE ARG 1
    THEN BEGIN
        LOC 1 = WP;                .. SAVE OLD WORKSPACE POINTER
        WP = @(ARG 1 * 4);          .. NEW WP
        R13 = LOC 1;               .. OLD WP
        R14 = PC;                  .. OLD PC
        R15 = ST;                  .. OLD ST
        PC = @(ARG 1 * 4 + 2);      .. NEW PC
        LOC 1 = ARG 1 - 1;          .. NEW PRIORITY
        IF LOC 1 LT 0
            THEN LOC 1 = 0;         .. PATCH FOR LEVEL ZERO
        ST = (ST AND OFFFO) + LOC 1; .. NEW ST
        CSAP;                       .. INTERRUPT SUCCEEDED
    END ELSE BEGIN
        'INTERRUPT FAILED'; NL      .. INTERRUPT FAILED
        ST: EH;                    .. DISPLAY STATUS REGISTER
        RETURN;
    END
END
FORM INTR
': SIMULATION OF INTERRUPT: ';
'INTERRUPT LEVEL '='15';
END

```

HISTOGRAM ANALYZER PROJECT SOLUTION

MAIN ROUTINE



DBSA = DISPLAY BUFFER START ADDRESS

SYSTEM INITIALIZATION

INITIALIZE R12 OF SAMPLE ROUTINE TO P0 OF A/D TMS 9901			
RESET A/D TMS 9901 I/O PINS			
INITIALIZE A/D CONVERTER			
INITIALIZE R12 OF MAIN ROUTINE TO P0 OF D/A TMS 9901			
INITIALIZE HISTOGRAM BUFFER			
ENABLE INTERRUPT FLIP-FLOP AND MASK			
INITIALIZE D/A CONVERTER AND SCOPE			
IF IN HISTOGRAM DISPLAY MODE			
THEN		ELSE	
IF DBSA= HBSA		IF DBSA=SBSA	
THEN	ELSE	THEN	ELSE
CLEAR HISTOGRAM & SAMPLE BUFFER	↓	CLEAR HISTOGRAM & SAMPLE BUFFER	↓
SET DBSA TO HBSA SET DBEA TO HBEA		SET DBSA TO SBSA SET DBEA TO SBEA	

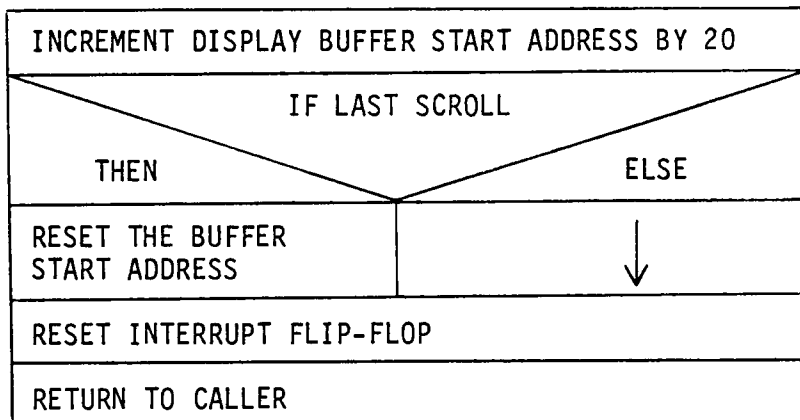
DBSA = DISPLAY BUFFER START ADDRESS
 DBEA = DISPLAY BUFFER END ADDRESS
 SBSA = SAMPLE BUFFER START ADDRESS
 SBEA = SAMPLE BUFFER END ADDRESS
 HBSA = HISTOGRAM BUFFER START ADDRESS
 HBEA = HISTOGRAM BUFFER END ADDRESS

SAMPLE SERVICE ROUTINE

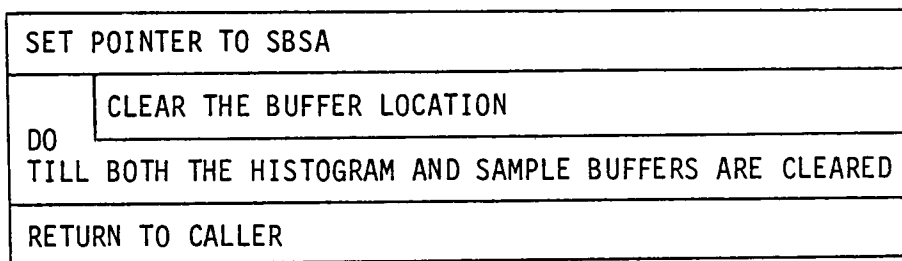
DISABLE ALL INTERRUPT
SET POINTER TO SBSA
START A/D CONVERSION
DO WHILE SAMPLE BUFFER NOT FULL
WAIT TILL CONVERSION COMPLETE
RESTART A/D CONVERSION
INPUT CONVERTED DATA
CALL HISTOGRAM CONVERSION SUBROUTINE
RESET INTERRUPT FLIP-FLOP
RETURN TO CALLER

SBSA = SAMPLE BUFFER START ADDRESS

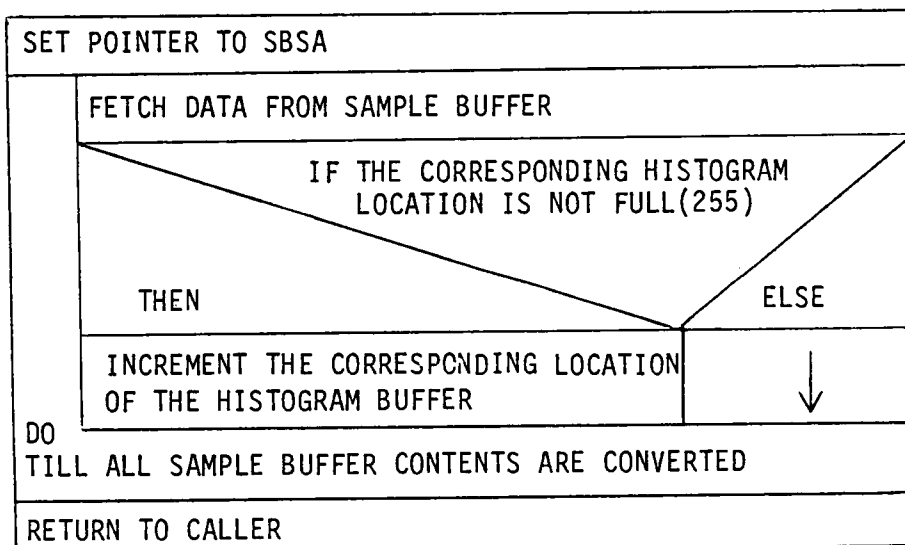
SCROLL SERVICE ROUTINE



CLEAR BUFFER SUBROUTINE



HISTOGRAM CONVERSION SUBROUTINE



HISTO SDSMAC 3.4.0 81.117 17:07:47 TUESDAY, SEP 06, 1983.

PAGE 0002

```

0001          IDT 'HISTO'
0002          *****
0003          ** THIS PROGRAM IS WRITTEN FOR TMS 9980 MICROPROCESSOR BASED HISTO-**
0004          ** GRAM ANALYZER. THE ANALYZER SAMPLES AN ANALOG INPUT VOLTAGE **
0005          ** (256 SAMPLES) AND STORES THEM IN A SAMPLE BUFFER. FROM THIS **
0006          ** BUFFER THE FREQUENCY OF OCCURANCES OF EACH VOLTAGE IS TABULATED **
0007          ** IN A HISTOGRAM BUFFER. EITHER THE SAMPLE BUFFER OR THE HISTO- **
0008          ** GRAM BUFFER CAN BE DISPLAYED ON THE SCOPE BY SETTING A HISTO- **
0009          ** GRAM/SAMPLE DISPLAY SWITCH. **
0010          *****
0011          OPTION SYMT          *GENERATE SYMBLE TABLE
0012          0400 MRWP EQU >400    *DISPLAY ROUTINE WP=>400
0013          0428 SAMWP EQU >428    *SAMPLE ROUTINE WP=>420
0014          0408 SCLWP EQU >408    *SCROLL ROUTINE WP=>408
0015          0440 SAMR12 EQU SAMWP+24 *R12 OF SAMPLE INTP. ROUTINE
0016          0420 SCLR12 EQU SCLWP+24 *R12 OF SCROLL INTP. ROUTINE
0017          0020 DACPO EQU >20     *DAC TMS 9901 PO
0018          0008 LSB EQU 8        *LSB & HSB OF DAC
0019          0009 LDAC EQU 9        *LDAC OF DAC
0020          000A INTSY EQU 10     *INTENSITY CONTROL OF SCOPE
0021          000B HZTRG EQU 11     *HORIZONTAL TRIGGER OF SCOPE
0022          000C SW1 EQU 12       *SAMPLE/HISTOGRAM SWITCH
0023          000D INTPFF EQU 13    *INTERRUPT FLIP-FLOP
0024          0060 ADCPO EQU >60    *ADC TMS 9901
0025          0008 WR EQU 8        *-WR OF ADC
0026          0009 INTR EQU 9      *-INTR OF ADC
0027          000A RD EQU 10       *-RD OF ADC
0028          0300 SBSA EQU >300    *SAMPLE BUFFER START ADDRESS
0029          0600 SBEA EQU >600    *SAMPLE BUFFER END ADDRESS
0030          0600 HBSA EQU >600    *HISTO. BUFFER START ADDRESS
0031          0700 HBEA EQU >700    *HISTO. BUFFER END ADDRESS
0032          0000 ADRG >0
0033          0000 0400 DATA MRWP  *POWER UP VECTOR
0034          0002 0014 DATA MR    *(MAIN ROUTINE WP & PC)
0035          0004 BSS 4            *RESERVED FOR TIMER
0036          0008 0428 DATA SAMWP *SAMPLE INTP. ROUTINE VECTOR
0037          000A 0080 DATA SAMSR
0038          000C 0408 DATA SCLWP *SCROLL INTP. ROUTINE VECTOR
0039          000E 00DE DATA SCLSR
0040          *****
0041          ***** MAIN ROUTINE ***** 3/23/83 *****
0042          *****
0043          ** THE FUNCTION OF THE MAIN ROUTINE IS TO PROVIDE A DISPLAY OF **
0044          ** CONTENTS OF THE SAMPLE BUFFER OR THE HISTOGRAM BUFFER ON AN **
0045          ** OSCILLOSCOPE AS A SERIES OF POINTS. HISTOGRAM/SAMPLE DISPLAY **
0046          ** SWITCH(SW1) DETERMINES WHETHER HISTOGRAM OR SAMPLE BUFFER IS **
0047          ** TO BE DISPLAYED. **
0048          ** R3 CONTAINS DISPLAY POINTER (DSPTR) **
0049          ** R4 CONTAINS DISPLAY BUFFER START ADDRESS (DBSA) **
0050          ** R5 CONTAINS DISPLAY BUFFER END ADDRESS (DBEA) **
0051          ** DISPLAY BUFFER IS CONTAINED WITHIN EITHER HISTO. OR SAM. BUFFER. **
0052          ** THE ROUTINE INITIALIZES THE BUFFERS, ADC, DAC, AND TMS 9901'S. **
0053          ** THEN IT SETS UP R3, R4, AND R5 ACCORDING TO THE POSITION OF **
0054          ** THE SWITCH #1. NEXT THE ROUTINE OUTPUTS THE DATA IN DISPLAY **
0055          ** BUFFER TO THE DAC, WHICH DRIVES THE SCOPE. AFTER ALL BUFFER **
0056          ** DATA ARE OUTPUTED, SWITCH #1 IS TESTED AGAIN AND R3, R4, AND R5 **
0057          ** ARE RESET ACCORDINGLY. THE ROUTINE RETURNS TO OUTPUT DATA AGAIN. **
0058          ** THIS CONTINUES UNTIL AN INTERRUPT OCCURS. **
0059          *****
0060          0010

```

HISTO SDSMAC 3.4.0 B1.117 17:07:47 TUESDAY, SEP 06, 1983.

PAGE 0003

```

0061          ***** SYSTEM INITIALIZATION *****
0062 0014          AORG >14
0063 0014 020C MR    LI    R12,ADCP0    *INIT. R12 OF SAMPLE ROUT.
          0016 0060
0064 0018 C80C      MOV    R12,@SAMR12  *TO PO OF ADC TMS 9901
          001A 0440
0065 001C 1DF0      SBO    -16        *RESET ADC TMS 9901 I/O PINS
          001E 1EFF      SBZ    -1
0066 0020 1E08      SBZ    WR        *SET ADC IN HOLD MODE
0067 0022 1D0A      SBO    RD        *DISABLE ADC OUTPUT
0068 0024 020C      LI     R12,DACPO   *INIT. R12 OF MAIN ROUT.&SCL
          0026 0020
0070 0028 C80C      MOV    R12,@SCLR12 *ROUT. TO PO OF DAC TMS 9901
          002A 0420
0071 002C 1DF0      SBO    -16        *RESET DAC TMS9901 I/O PINS
0072 002E 1EFF      SBZ    -1
0073 0030 06A0      BL     @HSTCNV     *INIT. HISTOGRAM BUFFER
          0032 0104
0074 0034 1E0D      SBZ    INTPFF     *CLEAR INTP. FF
0075 0036 1EF0      SBZ    -16        *UNMASK INT4 & INT5
0076 0038 1DF4      SBO    -12
0077 003A 1DF5      SBO    -11
0078 003C 1D0D      SBO    INTPFF     *ENABLE INTP. FF
0079 003E 0300      LIM1  3           *ENABLE INTERRUPT
          0040 0003
0080 0042 04C1      CLR     R1         *INIT. DAC AND SCOPE
0081 0044 3301      LDCR   R1,12
0082 0046 1F0C      TB     SW1        *IS SW1 IN HISTO. DSP. MODE?
0083 0048 130A      IF1    JEQ    ELSE1 *IF SW1 IN HISTO. DSP. MODE
0084 004A 0284      THEN1  CI     R4,HBSA *IS DBSA = HBSA ?
          004C 0600
0085 004E          *****
0086 004E 1602      IF5     JNE     FI5    *IF DBSA = HBSA
0087 0050 06A0      THEN5   BL     @CLRBUF *CLEAR HISTO. AND SAM. BUF.
          0052 00F6
0088 0054 0204      FI5     LI     R4,HBSA *SET DBSA TO HBSA
          0056 0600
0089 0058 0205      LI     R5,HBEA *SET DBEA TO HBEA
          005A 0700
0090 005C          *****
0091 005C 1009      ELSE1   JMP     FI1
0092 005E 0284      CI     R4,SBSA    *IS DBSA = SBSA ?
          0060 0500
0093 0062          *****
0094 0062 1602      IF6     JNE     FI6    *IF DBSA = SBSA
0095 0064 06A0      THEN6   BL     @CLRBUF *CLEAR HISTO. AND SAM. BUF.
          0066 00F6
0096 0068 0204      FI6     LI     R4,SBSA *SET DBSA TO SBSA
          006A 0500
0097 006C 0205      LI     R5,SBEA *SET DBEA TO SBEA
          006E 0600
0098          ***** INFINITE DISPLAY LOOP *****
0099 0070 1000      FI1     NOP
0100 0072 C0C4      DOTL2   MOV    R4,R3    *SET DSPTR = DBSA
0101 0074 3233      DOTL3   LDCR   *R3+,8  *OUTPUT TO DAC
0102 0076 1D08      SBO    LSB        *LOAD INPUT REG.
0103 0078 1E08      SBZ    LSB
0104 007A 1D09      SBO    LDAC        *UPDATE OUTPUT
0105 007C 1E09      SBZ    LDAC
0106 007E 1D0A      SBO    INTSY      *TURN ON INTENSITY

```

HISTO SDSMAC 3.4.0 81.117 17:07:47 TUESDAY, SEP 06, 1983.

PAGE 0004

```

0107 0080 1E0A      SBZ  INTSY      *TURN OFF INTENSITY
0108 0082 8143      C    R3,R5      *END OF DISPLAY?
0109 0084 16F7      ODTL3 JNE  DOTL3  *NO, CONTINUE DISP.
0110 0086 0300      LIM1  0          *DISABLE INTERRUPTS
      0088 0000
0111 008A 6103      S    R3,R4      *CAL. DIST. BETW. DBSA & DSPTR
0112 008C 1F0C      TB    SW1        *IS SW1 IN HISTO. DSP. MODE?
0113 008E 1304      IF3   JEQ  ELSE3  *IF SW1 IS IN HISTO. DSP. MODE
0114 0090 0205      THEN3 LI    R5,HBEA *SET DBEA = HBEA
      0092 0700
0115 0094 A105      A     R5,R4      *SET DBSA
0116 0096 1003      JMP   FI3        *
0117 0098 0205      ELSE3 LI    R5,SBEA *SET DBEA = SBEA
      009A 0600
0118 009C A105      A     R5,R4      *SET DBSA
0119 009E 0300      FI3   LIM1  3      *REENABLE THE INTERRUPT
      00A0 0003
0120 00A2 0202      LI    R2,>100
      00A4 0100
0121 00A6 0602      DOTL4 DEC  R2      *BLANK DSP. DELAY LOOP
0122 00A8 16FE      ODTL4 JNE  DOTL4
0123 00AA 1D0B      SBO  HZTRQ      *TOGGLE HORTZ. TRIGER
0124 00AC 1E0B      SBZ  HZTRQ
0125 00AE 10E1      ODTL2 JMP  DOTL2  *INFINITE DSP. LOOP
0126 *****
0127 ***** SAMPLE SERVICE ROUTINE ***** 3/23/83 *****
0128 *****
0129 ** THIS IS BUTTON #2 (INT 4) INTERRUPT SERVICE ROUTINE. IT ACTI- **
0130 ** VATES THE ADC TO SAMPLE 256 DATA FROM AN ANALOG INPUT VOLTAGE **
0131 ** AND STORES THEM IN THE SAMPLE BUFFER. AFTER THE SAMPLING IS **
0132 ** COMPLETED, A HISTOGRAM IS TABULATED. **
0133 *****
0134 00B0 0300      SAMSR LIM1  0      *ADC. MAIN ROUTINE
      00B2 0000
0135 00B4 1D0A      SBO  RD          *DISABLE OUTPUT & ADC INTP.
0136 *              *              *FROM CLEAR ITSELF
0137 00B6 0202      LI    R2,SBSA      *SET POINTER TO SBSA
      00B8 0500
0138 00BA 1E0B      SBZ  WR          *START ADC CONVERSION
0139 00BC 1D0B      SBO  WR
0140 00BE 02B2      DOWH11 C1  R2,SBEA *DO WHILE BUFFER NOT FULL
      00C0 0600
0141 00C2 130B      JEQ  ODWH11
0142 00C4 1F09      DOTL12 TB  INTR    *WAIT CONVERSION COMPLETE
0143 00C6 13FE      ODTL12 JEQ  DOTL12
0144 00C8 1E0B      SBZ  WR          *START CONVERSION AGAIN
0145 00CA 1D0B      SBO  WR
0146 00CC 1E0A      SBZ  RD          *ENABLE ADC OUTPUT
0147 00CE 3632      STCR  *R2+,B      *INPUT FROM ADC
0148 00D0 1D0A      SBO  RD          *DISABLE OUTPUT & ADC INTP.
0149 *              *              *FROM CLEAR ITSELF
0150 00D2 10F3      JMP  DOWH11
0151 00D4 06A0      ODWH11 BL  @HSTCNV *CALL HISTO. CONV. SUBRT.
      00D6 0104
0152 00D8 1EED      SBZ  INTPFF-32   *RESET INTP. FF
0153 00DA 1DED      SBO  INTPFF-32   *(P13 OF DAC TMS 9901)
0154 00DC 0380      RTWP

```

HISTO SDSMAC 3.4.0 81.117 17:07:47 TUESDAY, SEP 06, 1983.

PAGE 0005

```

0156 *****
0157 ***** SCROLL SERVICE ROUTINE ***** 3/23/83 *****
0158 *****
0159 ** THIS IS THE INTERRUPT SERVICE ROUTINE FOR BUTTON #3 (INT 5) **
0160 ** THIS ROUTINE SCROLLS THE DISPLAYED DATA ON THE SCOPE TO LEFT AT **
0161 ** INCREMENT OF 20. THIS ALLOWS CLOSE UP LOOK AT ANY PARTICULAR **
0162 ** SAMPLED DATA. WORKSPACES OF THIS ROUTINE AND THE MAIN ROUTINE **
0163 ** ARE OVERLAPED, SO R0 AND R1 OF THIS ROUTINE ARE R4 AND R5 OF **
0164 ** THE MAIN ROUTINE, RESPECTIVELY. **
0165 *****
0166 00DE 0220 SCLSR AI R0,20 *INC. DBSA BY 20
0167 00E0 0014
0167 00E2 8040 C R0,R1 *IS (DBSA-DBEA) > 0 ?
0168 00E4 1105 IF21 JLT FI21 *IF (DBSA-DBEA) > 0
0169 00E6 0221 THEN21 AI R1,->100 *CALC. DBSA
0169 00E8 FF00
0170 00EA C001 MOV R1,R0 *SET DBSA
0171 00EC 0221 AI R1,>100 *SET DBEA
0171 00EE 0100
0172 00F0 1E0D FI21 SBZ INTPFF *RESET INTP. FF
0173 00F2 1D0D SBO INTPFF *(P13 OF DAC TMS 9901)
0174 00F4 0380 RTWP
0175 *****
0176 ***** CLEAR BUFFER SUBROUTINE ***** 3/27/83 *****
0177 *****
0178 00F6 0205 CLRBUF LI R5,SBSA *LOAD SBSA
0178 00FB 0500
0179 00FA 04F5 DOTL31 CLR *R5+ *CLEAR BOTH BUF.
0180 00FC 02B5 CI R5,HBEA *(HISTO. & SAMPLE BUFFERS)
0180 00FE 0700
0181 0100 16FC ODTL31 JNE DOTL31
0182 0102 045B B *R11
0183 *****
0184 ***** HISTOGRAM CONVERSION SUBROUTINE *** 3/23/83 *****
0185 *****
0186 ** THIS SUBROUTINE TABULATES THE HISTOGRAM FROM THE SAMPLE BUFFER. **
0187 ** HISTOGRAM IS TABULATED BY ACCUMULATING UPON PREVIOUS VALUES IN **
0188 ** THE HISTOGRAM BUFFER. NO OVERFLOW IS ALLOWED. **
0189 *****
0190 0104 0202 HSTCNV LI R2,SBSA *SET POINTER TO SBSA
0190 0106 0500
0191 0108 04C3 CLR R3
0192 010A 04C1 DOTL42 CLR R1
0193 010C D072 MOV *R2+,R1 *FETCH DATA FROM SAM. BUF.
0194 010E 06C1 SWPB R1 *INTO LSB OF R1
0195 0110 D0E1 MOV *HBSA(R1),R3 *FETCH HISTO. BUF. LOC.
0195 0112 0600
0196 0114 02B3 CI R3,>FF00 *CHECK IF THE LOCATION IS FULL
0196 0116 FF00
0197 0118 1304 IF42 JEG FI42 *IF HISTO. BUF. LOC. IS NOT FULL
0198 011A 0223 THEN42 AI R3,>100 *INC. THE HISTO. BUF. LOC.
0198 011C 0100
0199 011E D843 MOV R3,@HBSA(R1) *PUT BACK INTO HISTO. BUF
0199 0120 0600
0200 0122 02B2 FI42 CI R2,SBEA *DO TILL CONVERSION COMPLETED
0200 0124 0600
0201 0126 16F1 ODTL42 JNE DOTL42
0202 0128 045B B *R11
NO ERRORS, NO WARNINGS

```



VITA

Seiko Harimoto was born in Taiwan on February 23, 1955. After completing seven years of Chinese education, he moved with his family to Okinawa, Japan, where he attended an American Catholic school for four years. In August 1972 he came to the United States and graduated in May 1974 from Washington College Academy in Limestone, Tennessee.

The following September he entered Middle Tennessee State University. After studying there one year he transferred to The University of Tennessee, Knoxville, and graduated in December 1978 with a Bachelor of Science degree in Electrical Engineering. Having accepted a teaching assistantship at UTK in the spring of 1979, he began study toward his Master's degree in Electrical Engineering. In the summer of 1981 he went home to Okinawa for two years and obtained a Japanese citizenship. He returned to UTK in the summer of 1982 to complete his study for the Master's degree.