

AD-A118 826 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG D--ETC F/G 9/2
MICROPROCESSOR SELF-TEST: SOFTWARE SELF-TEST FOR AN 8080-BASED --ETC(U)
JUN 82 J R ARMSTRONG, F G GRAY F30602-80-C-0200

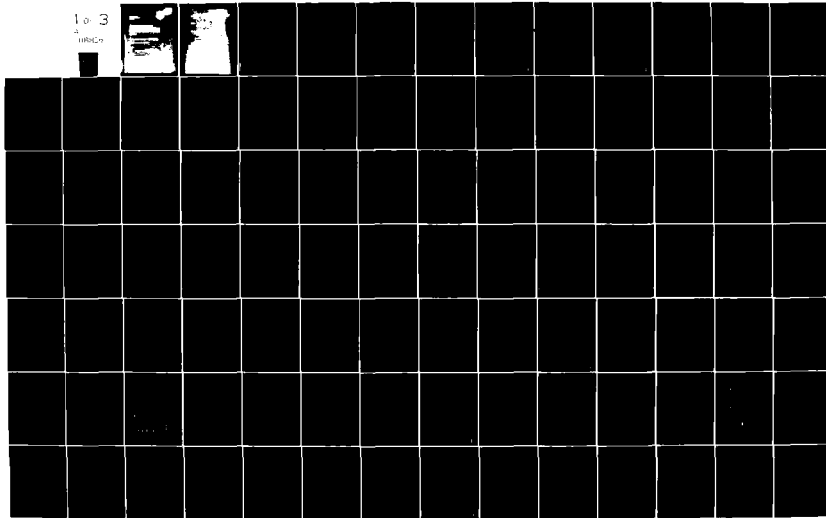
UNCLASSIFIED

RADC-TR-82-80

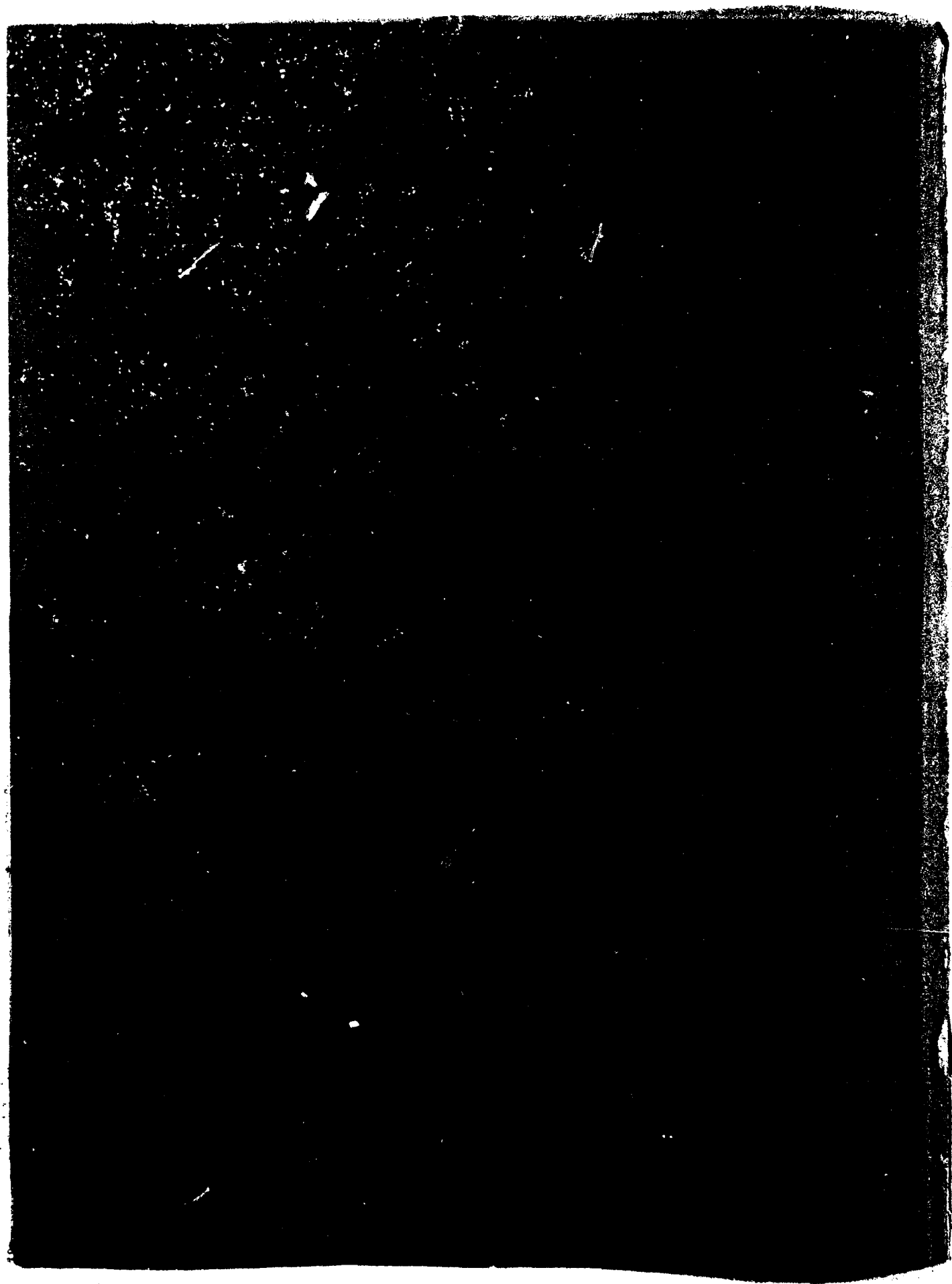
NL

1 of 3

AD-A118 826



AD A118826



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-80	2. GOVT ACCESSION NO. AD-A118836	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICROPROCESSOR SELF-TEST SOFTWARE SELF-TEST FOR AN 8080-BASED SYSTEM USING A MINIMUM OF ADDITIONAL HARDWARE		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 Sep 80 - 31 Aug 81
7. AUTHOR(s) James R. Armstrong F. Gail Gray		6. PERFORMING ORG. REPORT NUMBER N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS Virginia Polytechnic Institute & State Univ. Dept of Electrical Engineering Blacksburg VA 24061		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0200
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEA) Griffiss AFB NY 13441		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811722
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 243
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		16a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Lt Dean W. Gonzalez (COEA)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Microprocessor Self-Test Simulation		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A self-test program for an 8080-based microprocessor system is developed and verified using both high-level simulation and actual hardware components. The goals were minimum execution time, minimum added hardware, and minimum impact on applications software. Within these constraints, maximum fault coverage was obtained. A high-level simulation language (GSP) was used to verify the execution		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

of the self-test program and to determine the fault coverage. The self-test programs provided excellent diagnostic routines to test the simulation models and indeed were used to discover several faulty simulation models.

Finally, a complete self-testing hardware system was constructed to verify that the self-test program would run in the background of an applications program.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Report Summary

Introduction

This report describes the research efforts carried out under Air Force Contract F30602-80-C-0200. The purpose of the research was to develop techniques for the self-testing of microprocessors. These techniques were then implemented for a specific, 8080 based microcomputer system. The implementation took the form of a set of self-test routines and a small amount of added, self-test hardware. In order to assess the effectiveness of the self-test software, a "chip" level simulation model was developed and used to simulate faults in the systems and thus rate the effectiveness of the self-test software. Finally, a real 8080 system was built and the self-test software executed on it in order to demonstrate its compatibility with a real time computer system environment.

Self-Test Techniques

The report describes one program in a proposed library of self-test programs for microprocessor based systems. The library is to contain a set of programs with varying degrees of fault coverage and execution times. This report describes a self-test designed for minimum execution time, minimum use of added hardware, and minimum interference with the main system tasks. Within these constraints, maximum fault coverage is desired.

The basic approach was to partition the self-test program into segments that require from 2 to 4 milliseconds each to

execute. A timer is used to generate program interrupts at a frequency selected by the user (e.g., every two seconds). Each interrupt causes execution of the next self-test segment. The CPU test and parallel I/O port test both execute in the first segment. Memory tests posed the greatest demand upon execution time. Only 128 bytes of ROM or 32 bytes of RAM can be tested in the 2-4 ms window. Thus memory testing is carried out in a series of segments. Serial I/O port tests require 2 segments at 9600 Baud.

A second timer is used to insure that the interrupt request is acknowledged within a reasonable time. Once the interrupt is acknowledged, a timer is set for 2-4 ms, depending upon the program segment, to time execution of the self-test segment. If the self-test does not execute within the allotted time, an error condition is generated.

Two LED indicators are used to provide redundant error signals. One LED is normally ON. If the self-test software detects an error, if the interrupt acknowledge is not generated fast enough, or if a self-test program segment does not execute within the 2-4 ms window, then this LED is turned OFF to indicate an error. This provides a fail-safe indicator since the most prominent failure mode for an LED is to "burn-out" which indicates an error. The second LED provides a "heart-beat" status signal. This LED is toggled on and off at a fixed rate by the self-test program. This provides a redundant indication of failure. This system thus detects failure in both

the primary system hardware and in the self-test hardware.

A small amount of additional hardware is required to provide "wraparound" data paths for testing I/O ports. The added hardware required constitutes a small percentage of the total system hardware and all added hardware is covered by the self-test mechanism except the final isolation buffer that prevents external devices from corrupting the self-test data.

Fault Simulation

One of the difficulties in developing self-tests for LSI systems is trying to rate the effectiveness of the software. The reason for this is that, presently, the only known way of testing the effectiveness of self-test software is to conduct "fault injection experiments". One can either run these experiments with a real hardware system or through simulation. Using a hardware system is not feasible because obtaining LSI devices with known internal defects is much more difficult than obtaining good devices. Simulation does provide an answer, but there are problems here also. LSI devices contain thousands of gates, thus using traditional gate level simulation techniques can present great difficulties. The biggest problem is that accurate gate level models of LSI devices are usually known only by the manufacturer and in most cases they are unwilling to divulge this information. Secondly, even given a gate level model of an LSI system, the simulations require too much host CPU time, i.e. money, when validating self-test software. The only solution to this problem is to develop a simulation model

at a higher level.

On this contract a simulation language known as GSP (General Simulation Program) was used to develop a "chip" level model of the 8080 system under consideration. In modeling at the chip level, internal chip micro-operations and interface signal timing are modeled without resorting to detailed description of the internal gate structure. This allows accurate simulation of an LSI system in an efficient manner.

Once the simulation model was developed, it was used to conduct fault injection experiments. In these experiments; faults were injected into the simulation model, and the execution of the self-test software was simulated. The fault types simulated were (1) incorrect device micro-operations (2) stuck faults and (3) timing faults. Because of the high probability of interconnect failures between chips (vs internal defects), 43% of the defects simulated were interconnect faults.

The simulation experiments allowed us to calculate a "figure of merit" for the self-test routines, i.e. approximately 80% of faults injected were detected by self-test mechanisms.

Hardware System Checkout

In this effort an 8080 laboratory system was constructed and all self-test routines were executed on it. The purpose of this activity was to verify that the test routines would operate properly in a real system and that they would, when finished with their execution, leave the system in a state compatible with the operational program. Building of the hardware system

also allowed us to verify that the limited amount of added self-test hardware functioned as anticipated. Finally, experience with the hardware system provided the test program writer and simulation model developers with useful information about its characteristics.

Table of Contents

	<u>Page</u>
1. Introduction	1
2. Self-Test Methodology	3
2.1 Self-Test Environment	4
2.2 Alternative Approaches for a Short Periodic Test . . .	6
2.3 Constraints Imposed on System Design	7
2.4 Partitioning the System	8
2.5 General Statements about the Short Test Algorithms . .	9
2.5.1 Central Processing Unit (CPU)	11
2.5.2 Read Only Memory (ROM)	18
2.5.3 Random Access Memory (RAM)	20
2.5.4 Input/Output Ports	24
2.6 Implementation of the Short Test Algorithm for an 8080 System	28
2.6.1 Preliminary Considerations	28
2.6.2 Reporting Status	31
2.6.3 The CPU Test Program	32
2.6.4 ROM Test Program	38
2.6.5 RAM Test Program	40
2.6.6 Input/Output Ports Test Program	40
2.7 Performance Analysis	43
2.8 Hardcore Assumptions	44

3. Fault Simulation	46
3.1 Description of General Simulation Program (GSP) . . .	47
3.2 Simulation Model for an 8080 System	56
3.3 Modeling Process	56
3.4 Development of the System Model	59
3.5 Fault Injection Experiments	61
3.6 Analysis of Fault Coverage	65
4. Self-Test Hardware Experimentation and Documentation . . .	7
4.1 Experimental Verification of the Self-Test System . . .	"
4.2 Status Display	71
4.3 Timers: Interrupt and Timeout	71
4.4 Hardware Required to Self-Test I/O Ports	75
5. Conclusions	81
6. Recommendations for Further Research	84

Appendices

A. Self-Test Program Listings	
B. Simulation Model Flowcharts	
C. Module Assembly Language Descriptions	
D. Test System Data File	
E. Fault Experiments Summary	
F. MOVI test for RAM	
G. 8080 Micro-operations	
H. Checksum calculation program	
I. Details on the Test Board	

1. INTRODUCTION

The advent of LSI technology has presented computer system designers with a powerful design capability. However, along with this increased capability has come the attendant problem of trying to verify that the LSI devices in a system are operating correctly. The large number of logic gates on an LSI chip can make this testing process difficult. On the other hand, the LSI chips in a system tend to exhibit a degree of functional independence from each other and usually contain powerful logic capabilities. These features make possible the implementation of self-test mechanisms in LSI systems.

The purpose of the research carried out under Air Force Contract F30602-80-C-0200 was to develop self-test software for microprocessor systems and verify the effectiveness of this software through fault simulation. This document is the final report for this research.

The research efforts carried out under this contract can be divided into three major areas. The body of the report will treat each area in detail but we briefly summarize them here:

(1) Development of Self Test Software. Under this effort, a system self-test scheme was first developed which identified the major functions to be performed by software and a limited amount of self-test hardware to achieve the testing goals. Next, within the system scheme, self-test routines were designed and written to test the particular microprocessor chip set

chosen for the research: an 8080 microprocessor, semiconductor random access memory (RAM), read only memory (ROM) and 8228, 8251, and 8255 support chips. Details of this research area are given in section 2 of the report.

(2) Fault Simulation. In this part of the research a simulation model was developed for the microprocessor system. The modeling was done using the General Simulation Program (GSP) previously developed at VPI. Once developed and checked out, the simulation model was used for fault simulation. Functional faults were injected into the model and the execution of the self-test routines was simulated in order to test their effectiveness in detecting faults. Details of this research are given in section 3 of the report.

(3) Check out of the Self Test Scheme on a Hardware System. In this effort an 8080 laboratory system was constructed and all self-test routines were executed on it. The purpose of this activity was to verify that the test routines would operate properly in a real system and that they would, when finished with their execution, leave the system in a state compatible with the operational program. Building of the hardware system also allowed us to verify that the limited amount of added self-test hardware functioned as anticipated. Finally, experience with the hardware system provided the test program writer and simulation model developers with useful information about the characteristics of the chips. Details of this research are given in report section 4.

SELF-TEST METHODOLOGY

The motivation for this study is the development of a library of self-test software for microprocessor-based control systems. At one end of the scale would be a very fast executing self-test program that would provide as much fault coverage as possible using a minimal amount of extra hardware and a small amount of memory. The added cost for the self-test would be minimal. At the other end of the scale would be a comprehensive self-test that would provide the maximum possible fault coverage. It is anticipated that this test would require considerable execution time and possibly costly extra hardware. In between these two extremes would be a variety of self-test mechanisms that would provide a wide range of fault coverages with intermediate execution times, memory requirements, and hardware costs. If such a library existed, microprocessor system designers could select the library program that best matched their particular requirements.

This report describes one program in the library in detail. The primary goal of this project was to develop a short test using minimal extra hardware that would achieve the highest possible fault coverage. It was intended that successful completion of this project would establish credibility for the library concept in addition to being directly applicable in its own right.

2.1 Self-Test Environment

Of all the proposed library programs, this one would minimize the impact on system cost, on power requirements, on system programming and on system reliability. The extra hardware required can be classified into three categories.

Indicators of system status. Two LED's provide status information. One LED will be normally ON to indicate that the system is operational. This LED will be turned off by the self-test program if it detects an error condition or by a hardware time-out if the self-test program fails to respond within an appropriate time period. The normally ON condition makes the LED fail-safe since the most likely failure mode of an LED is to "burn out". However, it is possible for the electronic driving circuit to fail in such a way as to make the LED remain permanently ON. Therefore a second LED will act as a "heartbeat" for the system. It will be toggled off and on at a fixed visible rate by the self-test program. The "heartbeat" indicator will detect catastrophic type failures where the program is executing in an erratic manner that provides interrupt acknowledge and false pass signals to the fixed LED. This active redundant indicator will also detect stuck-at failures in the driving circuit of the first LED. The heartbeat then protects against failures in the self-test hardware and catastrophic failure of the CPU. For the system to be operating properly, the fixed LED must remain ON and the "heartbeat" must oscillate at a fixed visible rate.

Function Timers

Two timers are employed. One initiates the self-test program by periodically generating a program interrupt. The other times the response to the interrupt and the execution time of the self-test program. If a system failure prevents the microprocessor from responding to the interrupt request or prevents the microprocessor from executing the self-test program in the allotted time interval, the second timer will time out and indicate a system failure by turning off the fixed LED.

Wraparound and Isolation: Hardware for I/O Ports

Since testing 8255 and 8251 peripheral I/O chips was an important part of the self-test objective, a means for reading back the data written to the ports must be provided. In addition, we must isolate the peripheral device during testing to prevent distortion of the testing data by the external devices and to prevent test data from being transmitted to the external devices (when necessary). A device signal is provided to the external device during testing to indicate CPU busy status. The details of this logic can be found in Section 4.

This hardware represents the minimal amount necessary to implement complete system testing. The only burden placed on external hardware is to observe the busy signal and hold input data until the processor is ready to accept it. A complete description of the hardware is provided in section 4.

2.2 Alternative Approaches for a Short Periodic Test

A primary objective is to make the periodic test transparent to the users. This has two major ramifications: first, the user's registers and stack must be preserved; second, interrupts must be disabled during the test since execution of an interrupt service routine could lead to a hardware timeout (i.e., the test, once started, must run to completion uninterrupted); third, the test must execute in as short a time as possible so that its execution would not be noticed by the controller program. However, in general, a shorter test routine results in less fault coverage. In order to reduce execution time, one tries to design test algorithms with many operations between verifications; but too few verifications may allow a fault to escape detection. Thus a tradeoff between speed and fault coverage seems inevitable.

Three different approaches were considered for the short periodic test. The first is a simple, straightforward, quick test. A second approach uses a longer, more thorough (but slower) test and partitions it into a set of short segments that are executed one by one at consecutive test times. A third approach combines the first two methods. Again a series of tests is employed, but now a common 'core' test is executed each time. This core attempts to verify enough operations so that the housekeeping and dispatch functions required to decide which segment is to execute next can be expected to function reliably.

The primary advantage of the single comprehensive test is its simplicity. No overhead is required to schedule test segments. The major disadvantage is that fault coverage may not be adequate for a test that would execute in the available time window. The single test would certainly be preferable if the coverage is adequate. Our research indicates that such a test is practical for the 8080 CPU. However, we found that the execution time required for even a short memory test was excessive for this application. Since our objective is to test the whole system, we were forced to reject the single pass approach.

Experience with the test routine showed that the additional fault coverage gained by approach three was not worth the additional execution time. Therefore, we adopted approach two and partitioned the self-test into disjoint segments.

2.3 Constraints Imposed on System Design

A major objective of the self-test project was to provide an add-on package with minimal impact on the system design. This section describes the interaction required with the application system.

In the hardware area, the system must react to the self-test busy signal by holding input data until the self-test is completed. The required display isolation buffers must be provided. Three output port numbers must be reserved for

reporting the status and controlling the timers. About 1K bytes of ROM must be reserved for the self-test program.

In the software area, two vectored interrupts must be reserved for the self-test program. One is used to initiate the self-test execution and the other as an error exit. Sufficient additional stack depth must be provided to service the self-test program. The 8080 implementation requires 16 bytes of stack space to execute.

The application program must call the self-test initialization subroutine (INIT) whenever the system executes a cold start or system reset. See Section 2.6.1 for details. The programmer must also reserve 8 or 9 bytes of system RAM for the use of the self-test program.

In addition, the application program must operate with interrupts enabled most of the time. An extensive period with interrupts disabled would cause a hardware time-out. If the user has ROM to be included in the self-test, he must provide one checksum byte somewhere in every 128 byte block. See Section 2.6.4 for details.

2.4 Partitioning the System

Since we are doing functional testing, the most logical method to use in partitioning is based on function. In general, the CPU test, if possible, should be done in one segment. Memory will generally require many segments. As many I/O

devices as possible should be included in the remaining segments. Each segment should be approximately the same length in execution time.

For the 8080 system, we were able to test the 8080 CPU and the 8255 I/O port in the first segment. The ROM test was partitioned into 128 byte segments and the RAM test into 32 byte segments. The 8251 test required two segments. Execution time of each segment was approximately 4 milliseconds. The 8251 test execution time is practically clock independent since it depends mostly on the baud rate.

2.5 General Statements about the Short Test Algorithms

The self-test algorithms are designed to provide systemwide functional GO/NO-GO tests; they do not provide diagnostic information about what fault occurred. As such, they employ the 'start big' approach; i.e., they jump right into testing various functional elements rather than slowly building up from a small core. The tests are systemwide in that failures cannot be isolated to a single device; for example, a ROM or RAM fault could well cause the CPU test to fail. The algorithms were developed to cover single functional faults, although most multiple faults will also be detected.

Since there is virtually no failure mode data for microprocessors and their support chips, we don't know what faults are most likely to occur and cannot concentrate on testing for specific faults. Therefore the basic goal of the

self-test is to exercise all functional elements and data paths of the microprocessor system (excluding user peripherals, but including their I/O ports). Of course, exercising a faulty element or function does not guarantee detecting the fault; therefore we have made use of fault simulation results to determine how effective the self-tests are. (See Section 3.)

2.5.1 The CPU

The most complex component to test is the CPU itself; in general it consists of an ALU (arithmetic & logic unit), a (user) register array, other assorted registers and latches (accumulator(s), instruction register, etc.), some flags, instruction decoding logic (probably an internal ROM), timing and control circuitry, and the data paths connecting these elements.

The self-test exercises all functions of the ALU, thus testing the ALU control logic. The full adders that perform addition and subtraction are exercised by applying all input combinations to each adder (i.e. each bit position); since a full adder is a 3-input device (2 source operands and a carry in), 8 input combinations per adder are required. This tests for all detectable stuck-at faults and some shorts/opens in the adders. Similarly, the logic that performs AND, OR, and XOR is exercised by applying all four possible input combinations to each bit position. (That is, each bit position of each logic function is tested with inputs of 00, 01, 10, and 11.) Other functions (such as rotates) are tested in a similar manner, by applying all input combinations to each bit position for thorough coverage. A decimal (BCD) adjust function, if present, should be tested for no adjustment, adjustment due to flags, and adjustment due to a digit greater than nine.

The register array contains RAM (register memory), register select logic and multiplexers, and possibly increment/decrement, rotate, clear, and/or complement logic. The RAM must be tested for stuck-ats and shorts between adjacent bits; just which bits are adjacent depends on the RAM layout (which, in general, is unknown). Therefore the test requires loading (and verifying) at least three patterns into each register; the patterns apply both a 0 and a 1 to each bit position (stuck-at test), and the same and complement values to each pair of adjacent bits (test for shorts). These same patterns will also be used in testing the processor's other registers and latches and the data paths. This will be done by executing instructions that move the register data through the desired data paths to the other registers. When the internal layout of register memory and data paths is unknown (as usual), the most logical assumption is that logically adjacent bits are physically adjacent (this is certainly true for at least some of the registers, latches, and/or data paths). Under this assumption, some suitable patterns are (in hexadecimal): 00, 55, & AA; 33, 66, & CC; D9, 6C, & 36. The patterns are distributed in the registers so that register select faults may also be detected, assuming either logical ORing or logical ANDing (as appropriate for the technology employed) of register contents for a multiple select fault on reading. Thus the tests employ different patterns in the different user registers so that R1 OR R2 (or R1 AND R2, as

appropriate) equals neither R1 nor R2. This allows both erroneous select and multiple select faults to be detected. A fault resulting in no register being selected will be easily detected when using these patterns. The contents of all user registers are verified near the end of the test, so that an erroneous write or multiple write fault can be detected.

Increment/decrement logic would probably be centralized in one unit within the register array (as in the 8080); however, the exact gate implementation is most likely unknown. For this reason, the self-test applies only a few basic test vectors, such as incrementing -1 (all ones) and decrementing zero, which tests carry/borrow propagation through every bit. Carry/borrows through no bits (incrementing an even number and decrementing an odd number) and through an intermediate number of bits are also tested. It should be noted that this logic may also be used to increment the program counter (PC) and/or stack pointer (SP) (as is the case for the 8080), which provides some additional increment testing. Other possible register functions, such as clear or complement, are most likely built into each register if they are present at all. These are easily tested, but testing and verifying each function of each register will be time consuming, so that a tradeoff may be necessary.

Any microprocessor will contain, in addition to the register array, some internal registers and latches and possibly special accumulator registers. These registers may be tested

for stuck-ats and shorts between adjacent bits by using the same patterns discussed above, assuming they can be loaded by software (either directly or indirectly). The instruction register (IR) is of interest in that it is loaded, not with data, but with instruction opcodes; this means that a fault will result in the execution of some erroneous instructions, possibly causing loss of program control. This cannot be avoided; however, as long as the GO signal is not generated, a hardware timeout will provide the needed NO-GO result, so that the fault will be detected. Accumulator registers are also of interest in that they inevitably have special functions, such as complement and/or increment. The self-test must exercise each function of each accumulator with sufficient patterns to ensure the detection of any (detectable) stuck bits. Two complementary patterns will suffice to test complement logic; but note that two successive complements (with no verification in between) results in a poor test, since the correct final result will be obtained if each complement does nothing at all. The program counter (PC) and address latches/buffers are of special interest as well, because, for meaningful results, only valid memory and I/O addresses can be applied. However, these registers/latches are used for all memory reads and writes, including instruction fetches, and sometimes for I/O, so that enough patterns will be applied during the course of the test to detect most stuck-ats/shorts. Increment and decrement functions are tested as described above.

The condition flags make up another CPU element to test. The self-test simply applies and verifies (by conditional jump, add with carry, ...) both one and zero (true and false) for each flag. Shorts between the flag flip flops are conceivable, but cannot be efficiently tested for without knowing the internal layout. The logic driving the flags is exercised throughout the test by numerous arithmetic and logical instructions; but obviously the test can only verify the flags at strategic points (optimally where another function is also verified) to minimize execution time.

The instruction decoding and machine cycle encoding unit of the CPU is the most difficult component to test. To simplify LSI implementation, the decoder most likely employs ROM, the exact nature of which is unknown. Thus a fault in the ROM could conceivably be manifested for only a single instruction. Also, note that faults in instruction decoding, like faults in the instruction register, may cause erratic behavior or even loss of program control (hopefully resulting in a hardware timeout). Since the self-test cannot execute and verify every instruction in the short time available, it simply covers as many classes of instructions and as many micro-operations as possible. For example, only a few register to register moves are tested, instead of trying to move each register to each other register. All types of addressing and all types of parameters (registers, immediate data of all lengths, immediate addresses) are employed

through the course of the test. However some instructions (such as halt) cannot be self-tested without special hardware. The most commonly used instructions (such as adds, compares, branches, etc.) are tested most thoroughly. Conditional transfers (jumps, calls, and returns), for example, are tested for both transfer and no-transfer; note that this overlaps with flag testing and almost everything else, since the conditional transfers are used for verifications of other functions. This overlap is typical of the 'start big' approach, several functions being tested together.

The self-test is designed to exercise all possible micro-operations, thus testing some decoding and most of the machine cycle encoding. The micro-operations are determined from the processor's User's Manual based on the data paths, CPU elements, and functions employed by each instruction. All registers of the register array are considered equivalent since they use identical data paths external to the array (only register selection differs, and this was considered previously). The other registers (IR, accumulator, ...) are treated independently since their data paths differ. Conditional type instructions, which employ different micro-operations under different conditions, are considered to cover only those micro-operations common to both conditions. This prevents the illusion of covering micro-ops that may in fact not be performed during instruction execution. Software has been developed to

determine the micro-operation coverage of a given test algorithm, and to find a minimal set of instructions that cover any set of micro-operations. Fault simulation will be required to determine actual fault coverage, since exercising a faulty micro-op does not guarantee detecting the fault.

Since the micro-operations are derived based in part on the data paths they use, exercising all micro-operations also serves to exercise all data paths. Most of the data paths are exercised with the register test patterns previously described, thus testing for stuck-ats and shorts between adjacent bits. As mentioned earlier, this is performed by executing instructions that move data from the registers over the desired data paths (thus providing more overlap). Some data paths, however, cannot be directly exercised with these patterns. For example, consider the data paths leading to the instruction register and those leading to an address buffer. For meaningful results, only valid opcodes and valid addresses, respectively, can be applied to these data paths. However, during the course of the test, enough opcodes will pass into the instruction register to effectively test for any stuck-ats/shorts in IR or its data paths. Also, the RAM, ROM, and I/O tests will access all valid addresses so that most stuck-ats/shorts in the address circuitry can be detected. Thus although complete stuck-at/short coverage is not in general possible for all data paths, the self-test can still verify that valid data will not be distorted. What happens to invalid addresses is not important anyway.

The final part of the CPU is its timing and control circuitry; this cannot be directly self-tested. However, by exercising all micro-operations, the self-test will also exercise much of this control circuitry. Further, the hardware timeout feature guards against some possible major control faults that are totally transparent to the software, such as generation of numerous unneeded hold or wait states. Also, interrupt control is verified since the test is initiated by interrupt. Still, some control signals cannot be self-tested without considerable extra hardware because of their nature (e.g., the HOLD/HOLD Acknowledge circuitry of the 8080 is used for DMA by external devices and thus is completely transparent to software). This is a limitation that cannot be avoided without the addition of considerable extra hardware.

2.5.2 ROM

ROMs are the easiest system component to test. A checksum is stored in the ROM itself to produce a known result when the contents of all (or a certain piece of) ROM are summed. Several different methods of forming this sum have been considered. The first uses a modulo 2 sum, in effect an exclusive OR; each column (bit position) is independent of the others (there are no carries). However, two faults in the same column would go undetected; hence this method was rejected. The second method uses a modulo 256 sum, namely the ADD instruction; carries out of the most significant bit (MSB, bit 7) are lost. But two

faults in the MSB column would again go undetected, so this method was enhanced to form the third approach: the carry out of the MSB (from the ADD) is added back to the least significant bit (LSB, bit 0) of the sum; thus nothing is lost. Now to escape detection, the two faults must not only be in the same column, but they must also be complementary. That is, one must be a 0 turned 1, and the other a 1 turned 0. But due to the physical nature of a ROM, PROM, or EPROM, this is extremely unlikely; faults will normally occur in only one direction. Thus 0's may turn to 1's or 1's to 0's, but not both. Under this assumption, the third test method will prove quite effective.

Another consideration for the ROM test is how many checksums to use. The test algorithm is passed the start address of the ROM to test to make it address independent, but this also allows the ROM to be tested in pieces of any size desired, so long as each piece contains a checksum byte (to give the required sum); the checksum may be anywhere in the block of ROM under test. The ROM test could thus be broken up into a series of tests, so that periodic tests can execute in the short time available.

The final consideration is what number to use as the final known result of the sum. A sum to -1 (FF hex) was considered until we noticed that a 'dead' ROM (permanently deselected) would pass the test. (Since the bus would float when the ROM

should be active, FF hex would be read as the contents of each byte, resulting in the net sum of FF hex.) A sum to zero was similarly rejected in favor of a sum to AA hex, since the latter is a 'checkerboard' pattern (10101010 binary).

The test algorithm thus tests the ability of the ROM to access each location which verifies the address decoders, select logic, output drivers, and the ROM contents.

2.5.3 RAM

Many techniques for RAM testing have been developed over the past decade, each with its own advantages and disadvantages. But all thorough RAM tests share a common drawback: they take forever. Faster, specialized tests could be developed were the internal cell layout of the RAM known, since then a cell's true neighbors would be known. This would permit minimal tests of the decoders (access each row and column of the RAM only once) and allow true nearest neighbor (disturb) tests. But cell layouts vary from manufacturer to manufacturer and are almost never made available to users.

The Moving Inversions (MOVI) test technique is one example of a thorough test, and it is much faster than Walking or Galloping tests (1,2). A memory location is first read to verify it contains the previous pattern; then the current pattern is written and immediately read back for verification (to try and detect write recovery faults). This continues until the memory is filled with the current pattern. The patterns

employed are (in hexadecimal): 00, 01, 03, 07, ..., 7F, FF, FE, FC, F8, ..., 80, and back to 00; thus one bit is inverted each time. MOVI sequences through the RAM first moving forward (up) and then backward (down). In addition, MOVI steps through memory using each address bit as the LSB; that is, MOVI first moves from location N to N+1 (N-1 on down cycle), then from N to N+2 (N-2) on the next pass, then N to N+4 (N-4), etc. Thus all fundamental address transitions are tested (i.e. a change by a power of two, both forwards and backwards); this provides some testing for cell, row, and column disturb faults (despite the unknown layout). In order to test all of these basic address transitions, the test program tests all of (contiguous) RAM at once (testing smaller pieces would not test all basic transitions). Refer to reference [1] for complete details on MOVI. Note that MOVI does not test refresh for dynamic RAMs.

MOVI is a fairly good test for address decoder switching speed, cell, row, and column disturb faults, data sensitivity, and write recovery faults [2]. It is a very good test of address uniqueness, and a good general test of both functional and dynamic behavior [1].

The MOVI test requires $12 \times B \times n \times N$ memory cycles, where B = number of bits per word, n = number of address bits, and $N = 2^n$ = number of RAM locations. Naturally, a self-test program is much slower due to all the overhead it must perform. The MOVI test implemented for the 8080 requires about 20 seconds per

1K of RAM. Also, this test is, of necessity, destructive; that is, original RAM contents are lost. Thus MOVI is not at all suitable for a short, periodic test.

Therefore, two nondestructive tests were developed for the short periodic test, but they are, of necessity, not as thorough. Both algorithms make use of 'random' patterns, generated using a feedback shift register technique. This employs an irreducible polynomial of degree 8 to generate a sequence of 255 test patterns. The programs generate the next pattern by shifting the current pattern left and exclusive-ORing bits 2, 3, and 4 with the carry out from the MSB (bit 7). This carry out is also shifted into the LSB of the new pattern. Starting with 55 hex, the sequence is: 55, AA, 49, 92, 39, 72, E4, etc. Thus the desired 'randomness' is achieved. All 8-bit patterns except 00 will be generated before the sequence repeats. Note that test algorithms using this technique require one byte of RAM in which to store the current pattern. Since these random patterns change from test run to test run, numerous different patterns will be applied (over time), which provides some likelihood of detecting pattern sensitive faults. The algorithms are passed the start location of the RAM to test, so that RAM may be tested in segments.

The first algorithm tests RAM one location at a time, thus not testing address decoding (uniqueness) at all. It works as follows:

- (1) Read & save a RAM byte
- (2) Write/verify complement of original contents
- (3) Write/verify 'random' pattern
- (4) Write/verify complement of 'random' pattern
- (5) Restore/verify original contents

The second algorithm tests groups of two successive RAM locations, thus providing a minimal test for address uniqueness. It works as follows, where M and M+1 are the two locations under test:

- (1) Read & save both M & M+1
- (2) Write 'random' pattern to M+1; verify both M & M+1
- (3) Write complement of 'random' pattern to M;
verify both M & M+1
- (4) Restore & verify both M & M+1

Location M+1 then becomes the next M, and the test continues.

Both algorithms immediately follow each memory write with a read from the same location in an effort to detect write recovery faults. However, the algorithms test primarily for data errors in a RAM cell and the ability to read and write, with at best minimal testing of other RAM faults (such as cell, row, and column disturb faults, address uniqueness, and address decoder switching speed). Note that although the sequence of

'random' patterns does not include zero, a zero pattern will eventually be written since the complement of the 'random' pattern is also used.

2.5.4 I/O Ports

As mentioned in Section 2.1, self-testing I/O ports requires external wraparound hardware. Consider a serial I/O port chip (such as an 8251); three tri-state buffers can provide wraparound. Two are normally enabled to allow passage of user I/O data; the third, normally disabled, connects the serial output to the serial input. Then in test mode the first two buffers are disabled (tri-stated), isolating the user's external serial device, and the third buffer is enabled for wraparound. Thus serial data output will be received by the same chip's serial input, simultaneously exercising both transmit and receive logic if the serial chip is full duplex. Parallel I/O chips are tested similarly, using a single I/O chip if it has multiple I/O ports (as does the 8255) or using one output chip and one input chip if not (thus testing both at once).

Serial I/O chips (UARTs/USARTs) and some parallel I/O chips handshake with the CPU by means of status bits. In general, the processor must wait for proper status before performing input or output. This means that a fault in the I/O chip could leave the CPU in an infinite wait; this, however, will result in a hardware timeout so that the NO-GO test result will still be generated.

Some of the more recent LSI I/O chips (like the 8251 & 8255) are software programmable and can function in more than one operational mode. Unfortunately, the chip's current mode cannot, in general, be read back from the chip. This presents a severe problem to the self-test program. If it is to be nondestructive (as the periodic test must be), the chip's current mode must be restored before returning control to the application program. But, since the current mode cannot be read from hardware, the applications program would have to be constrained to store current mode data in RAM accessible to the self-test routine. Since most applications never change the mode of the I/O port, the best solution seems to be testing the chips thoroughly (in more than one mode) only after system RESET (or upon user request), and performing configuration dependent nondestructive tests periodically.

The serial port test algorithm consists of three parts: a transmit/receive test; a break send/receive test; and an overrun error detection test. Tests for framing or parity errors would require external hardware, and are therefore not performed. The transmit/receive test simply outputs test patterns and verifies that the same pattern is received (via the wraparound) with no errors. (Note that this also serves to test the wraparound.) The patterns used are (hexadecimal) 00, 55, and AA -- the same patterns that were used in the CPU register test. This tests

for stuck-at's or shorts between adjacent bits in the parallel data paths leading to the I/O port and in the chip's data registers, as well as testing the serial send/receive circuitry and portions of the status logic. The break test works in a similar fashion: a send break command is issued and then the CPU waits for the break detect (received) status bit to come active. In the overrun error test, the CPU outputs one character and waits until the UART has received it; then, without reading this character, the CPU outputs a second character and waits for it to be transmitted and received. Then the processor verifies that the overrun error status bit is set and that the second character can be read correctly (the first is lost). Thus most of the status and I/O circuitry has been tested. A more thorough destructive test would repeat the test for different baud rates, character lengths, and/or other programmable features.

A parallel I/O port test is much more chip dependent; simple I/O can be tested by applying the same patterns used in CPU register testing (00, 55, & AA hex will do). This tests the data paths, data registers (if any), output drivers, wraparound data paths, and input circuitry for stuck-at's or shorts between adjacent bits. (Note that strobe driven input may require some special hardware to generate the strobe signal.) Special functions (like the 8255's Port C single bit set/reset function)

are then tested for correct functional operation, provided they do not alter the chip's current operating mode (so that the test remains nondestructive).

2.6 Implementation of the Algorithm for the 8080 System

The preceding section considered test algorithms for microprocessor systems in general. That methodology has been applied to a system consisting of an 8080 CPU, RAM, ROM, 8251 serial I/O port, and 8255 parallel I/O port. This section of the report will discuss the self-tests developed for this 8080 system. It should be noted that the 8228 (system controller) and 8224 (clock generator) are considered part of the CPU element, along with the 8080 itself. This is reasonable since these chips are usually located on the same printed circuit board as the CPU.

2.6.1 Preliminary Considerations

This program is written in such a way as to be applicable to as wide a variety of user environments as possible. Also, the user responsibilities are reduced to a minimum. However, it is never possible to be completely transparent. The program must know certain parameters about the actual system. The user must provide these constants as shown in the configuration dependent assembly language equate statements on page A.1.

The first item that must be specified is the starting address of a 1.25K (1280) byte segment of ROM to be used for the self-test program. This is specified by defining the system parameter START as shown on page A.1.

Other items to be specified include the address ranges of RAM and ROM, the memory mapped addresses of the 8255 and 8251 data ports and the three memory mapped I/O addresses necessary to configure the wraparound logic and the self-test timers. (See Appendix A, page A.1.)

Also, 8 contiguous bytes of system RAM must be reserved for use by the self-test program. The user must specify the address of the first of these eight bytes by defining the value of TSTAD (page A.1).

The ROM and RAM test segment sizes may be changed by altering the ROMSS and RAMSS parameters. This segment size must always divide the ROM or RAM into an integral number of segments, and hence should usually be a power of 2. Note that changing the ROM segment size (ROMSS) will change the number of checksums required for the self-test program listed in Appendix A. Increasing ROMSS results in fewer checksums, while decreasing ROMSS increases the number of checksums and may require adding JMP's around the checksum byte (if the checksum must be placed within a block of program code). Naturally, increasing a segment size increases the time per test pass, while decreasing either segment size makes for faster test passes.

In addition, any user ROM to be included in the self-test must include a checksum byte somewhere within each 128 byte

block. The checksum is chosen so that the modulo 256 sum (with end around carry) of all 128 bytes in the block is AA(Hex). A jump around this checksum may also be needed, so that 4 bytes out of each 128 bytes of user ROM must be dedicated to the self-test function. Only one byte is required if there is an unconditional branch in the block, since the checksum may then be placed immediately after this transfer. The 128 byte block size was chosen so that each ROM test segment will execute in approximately the same amount of time as the CPU/8255 test segment. The block size is small because the CPU self-test was made as short as possible.

The initialization routine is shown on page A.23. This routine (INIT) must be called by the applications program when processing a reset. This routine initializes all of the self-test variables in the self-test portion of RAM and initializes the programmable timers.

The entry point (START) must be reached from one of the 8 vectored interrupt locations selected by the system designer. All registers and flags from the main program are saved on the main program stack. The entry routine also initializes the timeout counter and reads the address of the next scheduled test segment from TSTAD. At the end of each routine, TSTAD is loaded with the address of the next segment to follow. Successive interrupts will then execute successive test segments. Variable

values that need to be retained are stored in the 8 reserved RAM locations.

2.6.2 Reporting Status

If the self-test discovers an error, the ERR routine (shown on page A.2) is executed. In our implementation, the LED that normally remains on is turned OFF to indicate an error condition. The processor is then halted. This routine was placed at the beginning of the self-test program to increase the probability of the HLT being executed when the CPU is bad. Three successive HLT statements take care of possible byte skew due to software failure.

If the current test segment executes properly, the GOEXIT routine is executed. This routine initializes the main interrupt counter to the desired interval and restarts the timeout counter to insure the next interrupt is processed. If the self-test program did not execute properly in the allotted time, then the timeout counter would turn off the error LED. This would fail only if the GOEXIT routine were accidentally executed properly by a faulty processor. To minimize this probability three HLT instructions precede this routine. Also, the software error routine immediately precedes this routine. We therefore minimize the probability of a faulty GO signal as much as possible.

2.6.3 The CPU Test

The 8080 CPU test is shown in A.4--A.11. The 8080 CPU self-test exercises all ALU functions, which overlaps with exercising all possible micro-operations. These ALU functions are: ADD and SUBtract both with and without carry/borrow (and for carry/borrow of both 0 and 1); XOR, OR, AND; rotates left and right both through the carry flag (RAL, RAR) and without the carry (RLC, RRC); decimal adjust (DAA); and no-operation (after INR/DCR). Since arithmetic is very common, the full adders used for addition and subtraction are tested with all input combinations for each adder (8 combinations for each of the 8 adders) during the course of the test. Note that the compare instructions are subtracts as far as the ALU is concerned. A self-contained subtest applies all input combinations to each of the ALU's XOR, OR, and AND logic function gates (4 combinations for each of the 8 gates for each function). (Users not concerned with the logic functions could omit this subtest.) The ALU's rotates are tested for functionality by performing each of the four different rotates once; this seems like a minimal test, but since the exact implementation of these functions is unknown, what is optimal? Also, the rotates are not one of the most commonly used ALU functions, so a longer test seems unjustified. The (BCD) decimal adjust function is tested for four cases: no-adjustment, adjustment due to the

carry flag CY=1, adjustment due to the auxiliary carry flag AC=1, and, finally, for adjustment due to a digit greater than nine. This test also verifies that the AC flag can be both one and zero (providing overlap with flag testing).

The 8080 register array consists of the user registers B&C, D&E, and H&L, plus the stack pointer SP, program counter PC, the W&Z internal registers, a 16-bit increment/decrement circuit, and a 16-bit address latch (plus multiplexers and demultiplexers for register select). Registers B,C,D,E,H,L, and SP are tested with the patterns described in Section 2.7 of this report. It is important to note that the 8080 XCHG instruction (exchange D&E with H&L) operates by switching the internal addressing of the DE and HL register pairs, and does not actually move any data at all [3]. Thus to test these registers' RAM cells, one must be wary of XCHG's, or the test coverage will not be what it seems. The register test patterns used for SP (hex AAAA, 5555, and 0000/FFFF) are also (while testing SP) applied to the array's address latch and inc/decrement circuit. This should detect any stuck bits or shorts between adjacent bits in these elements. The address buffer (which drives the address bus from the address latch), however, cannot be tested with these patterns, since (for meaningful results) only valid memory (and I/O) addresses can be applied (without adding more test hardware). The same restriction applies to the program counter

PC. In addition, the W&Z internal registers are not tested with these patterns either, despite the fact that they can be loaded via XTHL (exchange top of stack with H&L). This is because W&Z are used for all branching (jumps, calls, returns, and RSTs) except PCHL, which is unconditional. Thus though a stuck-at/short in W&Z could be detected, the conditional branch that must be used for verification would probably jump to the wrong address. (Also, the test would require a fair amount of time as XTHL is the slowest 8080 instruction.) Note that W&Z are implicitly tested to some extent since the test includes an XTHL and numerous branches; PC is similarly tested as it steps through the self-test programs.

The increment/decrement circuit is tested for worst case transitions (decrementing 0 and incrementing -1), for carry/borrow to/from the high order register of the pair, and for no carry/borrow propagation. Also, some additional testing is performed while testing stack operations and by the incrementing of PC after each fetch. Finally, register select logic is tested by using different and logically distinct patterns in the various user registers, as explained in Section 2.3.

The 8080 also contains several other 8-bit registers: the accumulator A, accumulator latch ACT, a TMP register, instruction register IR, and a data buffer/latch (driving the

data bus). The first three, A, ACT, and TMP, are used for almost all ALU operations and are tested (for stuck-ats and shorts between adjacent bits) with the register test patterns (described in Section 2.3) during the course of various arithmetic instructions (overlapped with ALU testing). Sufficient opcodes are applied to IR to verify that it too is free of stuck-ats and shorts between bits. The data bus buffer/latch is also tested for these faults by the opcodes and data bytes read from memory (latch) and by the patterns output by the CPU for testing memory and stack writes (buffer). The 8228 bidirectional data paths are also tested by these data transfers. The 8-bit increment/decrement function is tested in the same manner as the 16-bit inc/decrement circuit, with worst case transitions (incrementing -1 and decrementing 0) and various others. Finally, the accumulator's complement function (CMA) is also tested for any stuck-ats or shorts (between adjacent bits) while generating patterns for other uses (overlap).

The flags (CY, AC, even Parity, Sign, & Zero) are tested by verifying each as true (1) and false (0). This is simultaneous with the testing of conditional jumps: JZ, JNZ, JM, JP, JC, and JNC are all verified for both branch and no-branch. The remaining two, JPE and JPO, are tested for no-branch only since the Parity flag is not commonly used and the time seemed better

spent elsewhere. The AC flag is tested while testing the decimal adjust function (DAA). The CY and Z flags are considered most important and as such are tested most thoroughly. CY is the easiest to test, since it is used by adds with carry, subtracts with borrow, certain rotates, and decimal adjust. Some additional testing of all flags occurs while testing the PUSH/POP PSW instructions.

The 8080 instruction decoding, implemented by ROM, is tested by exercising all possible classes of instructions and all possible micro-operations. Also, the test includes instructions with zero, one, and two bytes of immediate data, and instructions with an immediate address (such as LDA). Likewise, all forms of addressing, direct, register, register indirect, and immediate, are exercised. Despite the fact that a decoder ROM fault may show up for only a single instruction, the self-test does not try to execute all instructions. Rather the test exercises classes of instructions, where, for example, ADD r, ADD M, and ADI xxx form a class, DCX rp form another, and XCHG is a class of its own. All classes of instructions are exercised except for DI, HLT, IN, and OUT (since extra hardware would be required to test these), and conditional calls/returns. IN and OUT would be tested in I/O port tests and/or status reporting if memory mapped I/O is not employed. A test containing conditional calls and returns has been implemented,

with all but the parity conditions tested for both call (return) and no-call (no-return). However, the extra time required is considerable (about 300 clock cycles) so that user priority will determine whether or not to use this version of the self-test. More importantly, the self-test exercises all micro-operations except those covered only by DI, HLT, IN, and OUT; this exercises most of the decoding and control logic of the CPU. Refer to Appendix G for a list of the 8080 micro-operations.

The data paths connecting the various elements of the CPU are tested (for stuck-at/shorts) "in the process of testing the elements themselves. Also, as was mentioned in Section 2.2, the micro-operations were developed with the data paths in mind; so all data paths are exercised to some degree. The weakest point is the path to the address buffer which, as stated before, is restricted to valid memory and I/O addresses (for meaningful results).

Tests were combined where possible. For example, on page A.4, the portion of the test between OK0 and OK1 executes the DCX, ADD, and ACI to set the carry and zero flags and EVEN parity. The JNZ, JNC, JPO instructions verify the correct operation of instructions, flags, and conditional jumps. Note that the JZ to OK1 is followed by a JMP error in case the JZ fails. The RST ERX is yet another branching mechanism that may work if the JMP fails. Finally, the HLT should hang up the

program if no transfer at all is executed. The RST ERX transfers control to one of 8 vectors in low memory, where a copy of the ERR routine is located.

Similarly, segments from OK1 to OK5 verify other combinations of operations. The segment OK5 verifies DAA, DAD, INX, and other register and ALU operations. As a final check, the contents of all registers are verified by adding them together modulo 256 with end around carry and checking for the expected sum. This tests for multiple register selects.

The other instruction tests are documented in appendix A, pages A.2 to A.10.

2.6.4 ROM Test Program

A ROM test using checksums formed by modulo 256 addition with carries added back to the LSB (as described in Section 2.5.2) has been implemented for the 8080 processor. The algorithm is passed the start address of the block of ROM to test and forms a sum of AA hex for a GO pass. The test routine object code occupies only 64 bytes of memory and runs in about $45N$ clock cycles, where N is the number of bytes of ROM to test. Thus 128 bytes of ROM can be tested in a single pass of just under 3 milliseconds (with a 2 MHz clock). See page A.16.

A sample program, written in Microsoft BASIC, for calculating the checksum needed for each ROM segment is listed in Appendix H. The program requests the segment size and

desired final sum for flexibility; these should be 128 and 170 decimal (AA Hex), respectively, to match the self-test program listed in Appendix A. The CHKSUM program accepts an INTEL ASCII format object file as input. This object file should have a zero byte where each checksum byte will be placed; the zeroes must then be changed to the appropriate checksums, and the file reassembled prior to burning the ROM.

2.6.5 RAM Test Program

A MOV_I RAM test and both quick, nondestructive RAM tests described in Section 2.5.3 have been coded for the 8080. The thorough MOV_I (Appendix F) test requires almost 20 seconds per 1K of RAM (for a 2 MHz clock) and, as stated earlier, tests all of contiguous RAM at once. The first quick test, which tests RAM location by location (A.17), requires about 150N clock cycles to test N locations, or about 75 milliseconds per 1K. The 32 byte segment used in the program (A.17) requires 2.3 milliseconds to execute. The second quick test, which tests two successive RAM bytes at a time, requires about 235N clock cycles for N locations, or about 120 milliseconds for 1K. Both of these tests are passed the start address of the RAM segment to be tested to allow segment testing. Each of the nondestructive test routines requires less than 100 bytes of object code, while MOV_I requires about 200 bytes. The first nondestructive test is included in the self-test program listed in Appendix A; it was chosen for its high speed.

2.6.6 I/O Ports Test Program

The 8080 has two software programmable I/O port chips: the 8251 USART for serial I/O, and the 8255 for parallel I/O. Nondestructive tests have been coded for both chips using memory mapped I/O so that the same test routine may be used to test several different 8251's (or several 8255's). However, the

routines may be easily converted to discrete I/O (using IN and OUT).

The asynchronous mode of the 8251 is tested and has been modelled for simulation. As described in Section 2.5.4, the test consists of an I/O test, a break send/detect test, and an overrun error detect test. Unfortunately, not all versions of the 8251 have the break detect capability [4,5]; the break is transmitted and wrapped around to the serial input, but a framing error is detected instead of the break. For these chips the break test has become a break send/framing error detect test. The 8251 test routine requires about 128 bytes of object code; its execution time depends upon the character length and baud rate selected. With one stop bit, 8 bit characters, no parity, and at 9600 baud, the 8251 test requires just under 8 milliseconds. Therefore, it was divided into two parts requiring about 4 ms each. (See p A.19)

An 8255 test has been coded to test Mode 0 operation (basic I/O with no strobes/handshaking) (See page A.11). Although the 8255 remains in Mode 0 throughout, it changes the port configurations (i.e., changes which ports are inputs and which are outputs). However, the test restores the 8255 to its original configuration, which must be known in advance. Before the test commences, all wraparound and isolation buffers are tri-stated (isolating the external device); then a pattern is

written to each port. This is a no-op if the port is defined as an input, while if it is an output, the pattern is latched and can be read back by reading that port. Each port is then read back; if it was an input, FF hex (all ones) is read (since the lines driving the inputs are tri-stated). If the port was an output, the pattern is read back; note that if neither value is read back, there is a fault and a NO-GO result is generated.

The first part of the test routine tests the 8255's three I/O ports (A, B & C) and the data paths involved for stuck bits or shorts between bits as well as testing the 8255's basic functionality. The test defines one port as output and the other two as input, and then outputs test patterns and verifies that they have been read back correctly through the two input ports. Each port has a turn as output port, as shown below:

Output port	Input ports	Patterns used (hex)
A	B, C	55, AA, 00
B	A, C	CC, 66, 33
C	A, B	D9, 6C, 36

The second part of the test checks the Port C single bit set/reset function. Each bit is set (verified) and reset; the test then verifies that setting a set bit and resetting a zero bit have no effect. The patterns in Port C are read back for verification through Port B. The total test routine requires approximately 300 bytes of object code and takes between 1700

and 1800 clock cycles to execute (0.85 to 0.90 milliseconds for a 2 MHz clock), depending upon the original configuration. Therefore, this test was combined with the CPU test to make a 2 ms segment.

A self-test methodology has been described for microprocessor systems in general and specific algorithms for an 8080 system have been discussed. The methodology has been developed under the constraints of minimum additional hardware, minimum impact on system users, and has been tailored to quick, periodic, transparent tests. Some elements/functions are untestable under these constraints (as was Halt), or can be given only a partial test (as for RAM). The test procedures proposed follow the 'start-big' approach and so attempt to overlap testing one element/function with testing others. Overlap is quite important under the above constraints in order to maximize fault coverage while minimizing testing time. Unfortunately, this overlap makes automatic generation of tests most difficult, but research toward this end is desirable.

2.7 Organization of Self-Test Segments

The overall system self-test is organized as a sequence of short periodic test segments. With a 2 MHz processor clock and fast memory that does not require WAIT states, the execution time of each segment is as follows:

where $m = (\text{ROM memory size in bytes})/128$
 $n = (\text{RAM memory size in bytes})/32$

Segment (1) Tests CPU and 8255 (2 ms)

Segment (2) Tests first 128 bytes of ROM (3 ms)

Segment (3) Tests next 128 bytes of ROM (3ms)

.

.

.

Segment (m+1) Tests last 128 bytes of ROM (3ms)

Segment (m+2) Tests first 32 bytes of RAM (2.5ms)

Segment (m+3) Tests next 32 bytes of RAM (2.5ms)

.

.

Segment (m+n+1) Tests last 32 bytes of RAM (2.5ms)

Segment (m+n+2) Tests 8251 (4ms at 9600 baud)

Segment (m+n+3) Tests 8251 (4ms at 9600 baud)

The total test time is $3m+2.5n+10$ milliseconds. For example, for a system with 16K of RAM and 2K of ROM, the total test time would be 1338 ms. The CPU test uses only 2ms of the total time. The 8251 uses 4 ms, the ROM requires 48 ms and the RAM requires the rest (1280 ms). The test is executed in $m+n+3 = 531$ sequential segments.

2.8 Hardcore Assumptions

The self-test, as mentioned previously, cannot test all CPU elements/functions completely without special extra hardware. Those elements not tested are: the HLT instruction; the DI (disable interrupts) instruction; the IN and OUT instructions (since memory mapped I/O is employed); conditional calls and returns (an enhanced CPU test verifying these calls/returns has been coded, but was not simulated); the program counter (PC) and address buffer (since only valid addresses may be used for meaningful results); and the WZ register pair. Faults in the

latter three elements may be detected by a hardware timeout should the program lose control due to faulty addresses. Finally, the 8080's timing and control circuitry cannot be self-tested directly (without special hardware).

3. FAULT SIMULATION

One of the difficulties in developing self-tests for LSI systems is trying to rate the effectiveness of the software. When one reads articles on self test development for LSI systems, the authors are usually mute on this point or make vague statements such as "the test routines were shown to be effective." The reason for this is that, presently, the only known way of testing the effectiveness of self-test software is to conduct "fault injection experiments." One can either run these experiments with a real hardware system or through simulation. Using a hardware system is not feasible because obtaining LSI devices with known internal defects is much more difficult than obtaining good devices. Simulation does provide an answer, but there are problems here also. LSI devices contain thousands of gates, thus using traditional gate level simulation techniques can present great difficulties. The biggest problem is that accurate gate level models of LSI devices are usually known only by the manufacturer and in most cases they are unwilling to divulge this information. Secondly, even given a gate level model of an LSI system, the simulations require too much host CPU time, i.e. money, when validating self-test software. The only solution to this problem is to develop a simulation model at a higher level. During the past several years at VPI, we've developed an approach to higher

level simulation known as chip level simulation. In chip level simulation, the internal microoperations of a device and the timing characteristics of the signals at its interface pins are simulated. This is done without modeling the detailed internal gate structure of the chip.

The simulation language that we employ is called GSP (General Simulation Program). It was developed under previous U. S. Navy research contracts. It has been used on this contract and we are also using it to do fault modeling for the NASA-Langley Research Center.

We have found GSP to be a very effective tool for the fault modeling required by this contract. As will be detailed, using GSP, we were successful in modeling the microprocessor system and conducting the required fault injection experiments.

3.1. A Description of the General Simulation Program (GSP)

The General Simulation Program (GSP) is a general purpose simulation program designed specifically to simulate LSI devices at the chip level, i.e. internal device micro-operations and detailed interface signal timing are simulated. The program is written in FORTRAN to insure portability. Presently the system runs in either a batch (MVS) or interactive (CMS) mode on an IBM 3032 Processor. An optimizing compiler (G1-HX(2)) is used to speed up the simulation.

When utilizing the GSP system one goes through the following three phases:

Phase 1 - Chip Description. The user models the device at the chip level (an example of this is given below) and codes its description using the GSP assembly language. This code description is then processed by the GSP assembler to produce an integer microcode file which can be used in any subsequent simulation requiring that device.

Phase 2 - Interconnect Description. The user edits an interconnect description file which is used to link the microcode descriptions of the individual modules into a total system microcode description. This description can be used for all subsequent simulations of the particular system.

Phase 3 - Simulation. External system inputs are specified and simulation is begun and repeated as necessary.

The process of modeling and coding a "sample" module is illustrated in Figures 3.1 through 3.4.

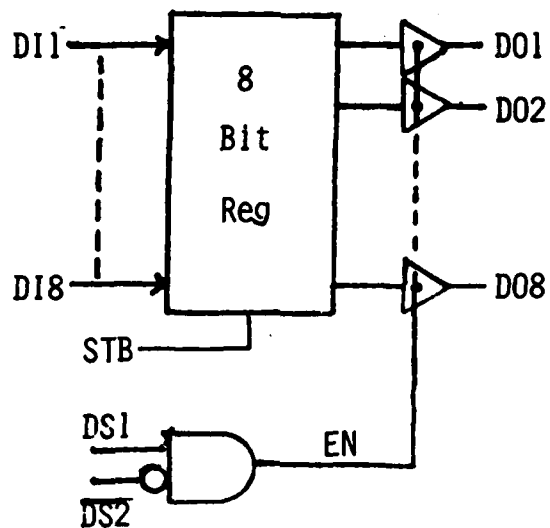
The sample module is an 8 bit register with buffered outputs. Data is clocked into the register on the fall of the strobe (STB). The output buffers are enabled (EN = 1) when the input select function (DS1 DS2) is true.

The first step in the modeling process is to examine the chip description and timing specifications to identify module events. As shown in Figure 3.1, the sample module has a 55 NS

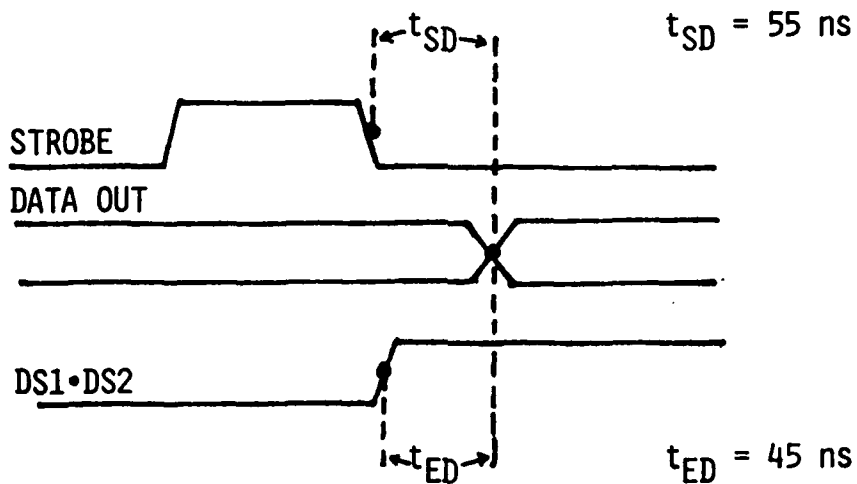
delay (t_{SD}) from the fall of the strobe until data appears on the outputs (assuming the buffers are enabled). Also, there is a 45 NS delay (t_{ED}) specified from the presence of the enabling input logic condition (DS1 DS2) until output data appears.

If the buffer delay is 10 NS, then three delay events can be identified: (1) negative strobe transition to register self call (45NS), (2) positive enable transition to enable self call (35NS), and (3) self calls to data output. The term "self calls" refers to the fact that after the module routine identifies either of the two external events (1) or (2), it causes an event to be placed in the time queue which will call the module routine in a specified number of nanoseconds. Once called, the routine will schedule the register content to appear in the outputs after 10 NS, provided that the buffers are enabled.

Figure 3.2 illustrates the second step in the modeling process: generation of the module flow chart. We have found this step in the modeling process to be very important in assuring an accurate model. A good deal of time can be fruitfully spent in this phase before proceeding to the coding phase.



Timing Specifications:



Events:

1. Neg strobe transition - register self call (45 ns)
2. Pos enable transition - enable self call (35 ns)
3. Self calls to output (10 ns)

Figure 3.1 Sample Module Specifications

For the example under discussion, the flowchart illustrates the logical structure of the model. After the enable (EN) signal is computed, a check is made to see if either the enable or the strobe events have occurred. Note that the events are detected by comparing the present value of a signal with the value of the signal that was stored the previous time the module was called (e.g. STB vs STBO). If either of the events has occurred, an appropriate self call is scheduled. Also, the value of the data or signal involved is "carried along" with the self call event for later storage.

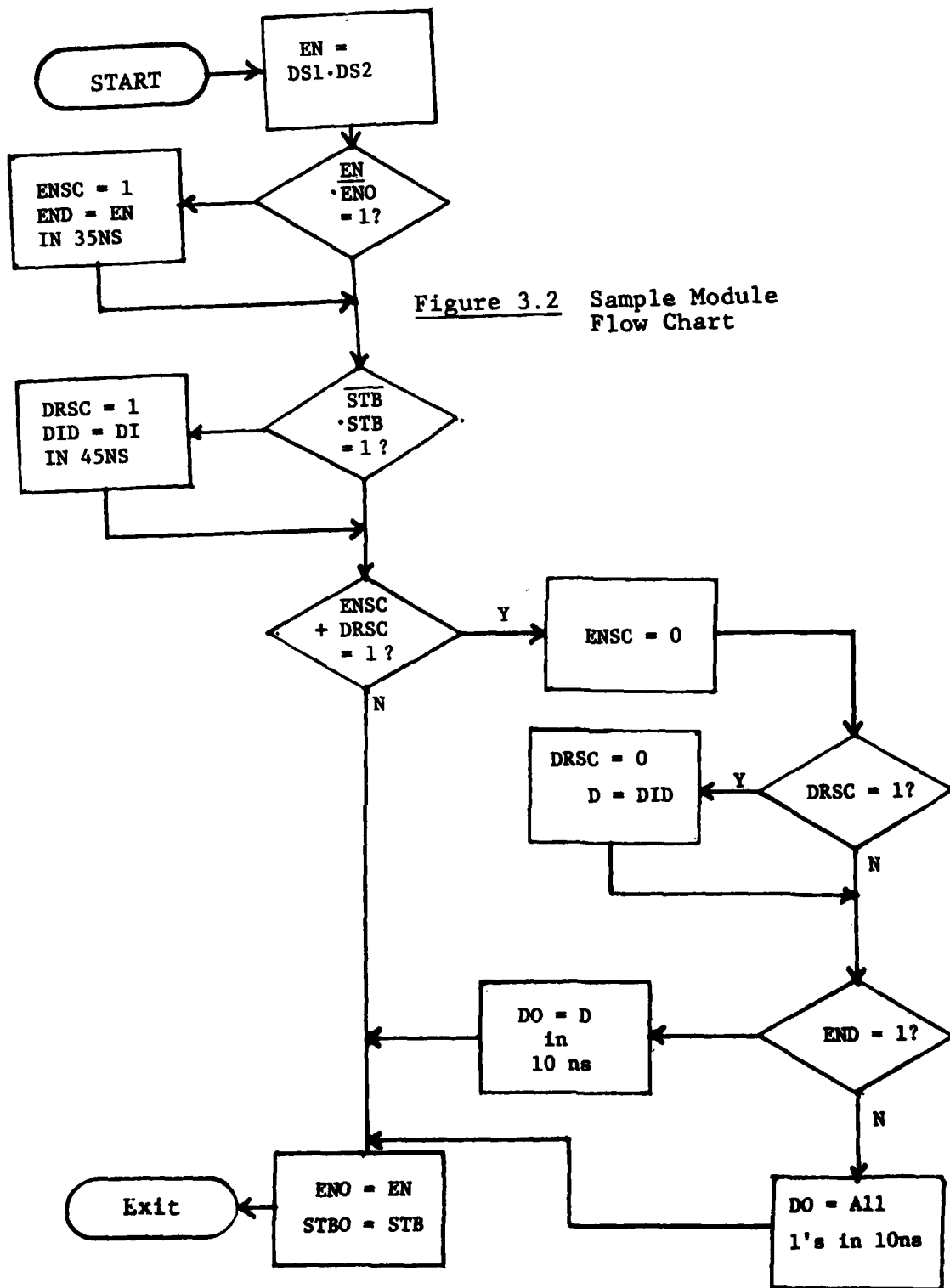
The next section of the flow chart checks for either an enable self call (ENSC) or a data register self call (DRSC). If DRSC = 1, the data register is updated using the previous value of the data input value that was "carried along" with the data register self call. Next, the value of the delayed enable (END) is checked. If END = 1, the data outputs are scheduled to take on the value of the data register in 10 NS. If END = 0, the data outputs will be forced to the all ones configuration simulating the high impedance state. The final activity in the flowchart is the updating of the "old values" of EN (ENO) and STB (STBO). After this, the module procedure is exited.

The final step is the coding of the module description using the GSP assembly language. This is illustrated in Figure 3.3.

Note that the description contains a declaration section and a section containing module micro-operations. The declaration section specifies registers (REG), pin connections (PIN) and events (EVW). Module micro-operations are specified using a rather normal looking assembly language except for instructions like: MOV(W10) D, DO. This instruction causes the contents of the D register to be moved to the output in 10 nanosJc.

Figure 3.4 shows the form of the integer microcode file for the sample module.

This section of the report has given only a cursory introduction to the General Simulation Program (GSP). For more information the reader is referred to reference 6.



DECLARATION SECTION:

```
REG(8) D ;SAMPLE MODULE, 8 BIT CLOCKED REGISTER
REG(1) TEMP1,EN,ENO,STBO ;TRI STATE OUTPUT WITH ENABLE CONTROL
PIN DI(1,8),DO(9,16),DS1(17),NDS2(18),STB(19) ;PIN SPECIFICATIONS
PIN DID(20,27),END(28),EX(61),ENSC(62),DRSC(63) ;PSEUDO PINS
EVW(500) W10(10),W35(35),W45(45)
```

MODULE MICROOPERATIONS:

```
MOV NDS2,TEMP1 ; COMPUTE EN=DS1.NDS2'
COM TEMP1
AND DS1,TEMP1,EN
ENC: XOR EN,ENO,TEMP1 ; CHANGE IN OUTPUT ENABLE?
BEQ TEMP1,STBC
MOV(W35) #1,ENSC ; SCHEDULE ENABLE SELF CALL IN 35NS
MOV(W35) EN,END ; CARRY ALONG CURRENT ENABLE VALUE
STBC: MOV STB,TEMP1 ; COMPUTE STB'.STBO
COM TEMP1
AND TEMP1,STBO,TEMP1
BEQ TEMP1,FLGCK ; NEG STROBE TRANSITION?
MOV(W45) #1,DRSC ; SCHEDULE REG SELF CALL IN 45NS
MOV(W45) DI,DID ; CARRY ALONG THE CURRENT INPUT VALUE
FLGCK: OR ENSC,DRSC,TEMP1
BEQ TEMP1,UPDATE ; EITHER SELF CALL FLAG SET?
MOV #0,ENSC ; RESET FLAG
BEQ DRSC,OUT ; CHECK FOR REGISTER SELF CALL
MOV #0,DRSC ; RESET FLAG
MOV DID,D ; UPDATE D REGISTER
OUT: BEQ END,HIZ ; CHECK OUTPUT ENABLE
MOV(W10) D,DO ; DO=D IN 10NS
BRU UPDATE
HIZ: MOV(W10) #225,DO ; DO=ALL 1'S IN 10NS
UPDATE: MOV EN,ENO ; UPDATE VARIABLES
MOV STB,STBO
MOV #0,EX ; EXIT MODULE
END
```

FIGURE 3.3 MODULE DESCRIPTION

Microcode

Module Microcode File

3.2. The Simulation Model for 8080 System

The system that was modeled consisted of the same chips that are in the hardware system: (1) 8080 microprocessor (2) 8228 System Controller and Bus Driver (3) Semiconductor Random Access Memory (RAM) (4) Read Only Memory (ROM) (5) 8251 Programmable Communication Interface (UART) and (6) 8255 Programmable Peripheral Interface (Parallel Port). The real system also contains an 8224 clock chip, however for simulation purposes this was considered to be part of the microprocessor. In addition to the above six models, the gating used to perform address selection was grouped together to form a Select Module. Finally the bus interconnect between the chips was also modeled as a module. Figure 3.5 below shows a diagram of the system model.

As pointed out in section 2, the system test scheme employs wrap around from I/O outputs to I/O inputs. It was not necessary to model this wrap around as separate modules in that we were able to use our regular method of interconnect specification.

3.3. The Modeling Process

The development of the simulation model for a computer system consists of steps which are analogous to hardware development. Models for the individual chips are first developed and checked out. This model development consists of

four steps: (1) examination of the manufacturer's specifications (2) development of a model flow chart (3) coding of the model (4) assembling the model code to produce a "micro code" file. This process was illustrated in section 3.1 for a sample module.

The development of the models for the test system followed the same four steps. Model development and model checkout for LSI chips is a sophisticated process, requiring that the modeler have a thorough understanding of the chip logic. We estimate that approximately 7 man months of effort were expended in model development. We do not discuss each model in detail in this report. However, in appendix B the model flow charts for the test system are given. Appendix C gives the assembly language description for each module. At the present time, a separate document discussing modeling techniques is in the writing process. We will forward this to RADC when completed.

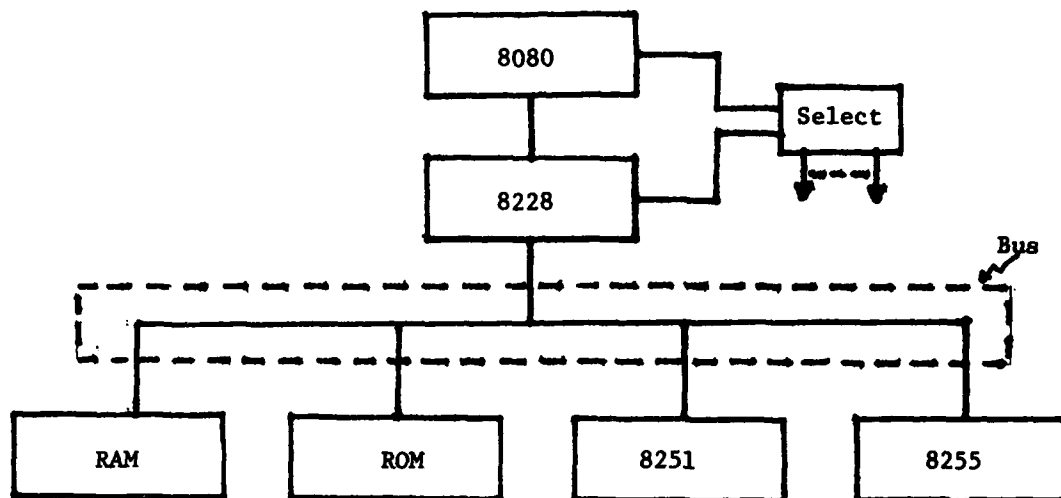


Figure 3.5
System Simulation Model

3.4. Development of the System Model

Once the individual models have been coded and checked out, they are merged to form a system microcode file. Figure 3.6 illustrates the process. It assumes that individual microcode files have been prepared. These files are then merged to form the LINK file. The LINK file holds the microcode for the entire system. In addition it also is used to maintain the state of all system signals initially and as simulation progresses. The merging of module descriptions to form the LINK file is performed automatically upon command.

Another file, the DATA file, contains necessary information on: (1) module interconnection (2) initial signal values (3) module input marking (4) input vector specification. The DATA file for the test system is given in Appendix D. Details of how to prepare the data file are given in reference 6.

Once the LINK file and the DATA file are in place the simulation can begin. As simulation progresses, the state of the system is maintained in the LINK file. Specified system outputs are routed to an output file for storage. The simulation output can also be simultaneously routed to the user's terminal and/or printer. The above discussion implies operation in an interactive computing system; however the simulator can be run in the batch mode as well. In fact, most of our longer simulation runs were made this way.

SIMULATOR STRUCTURE

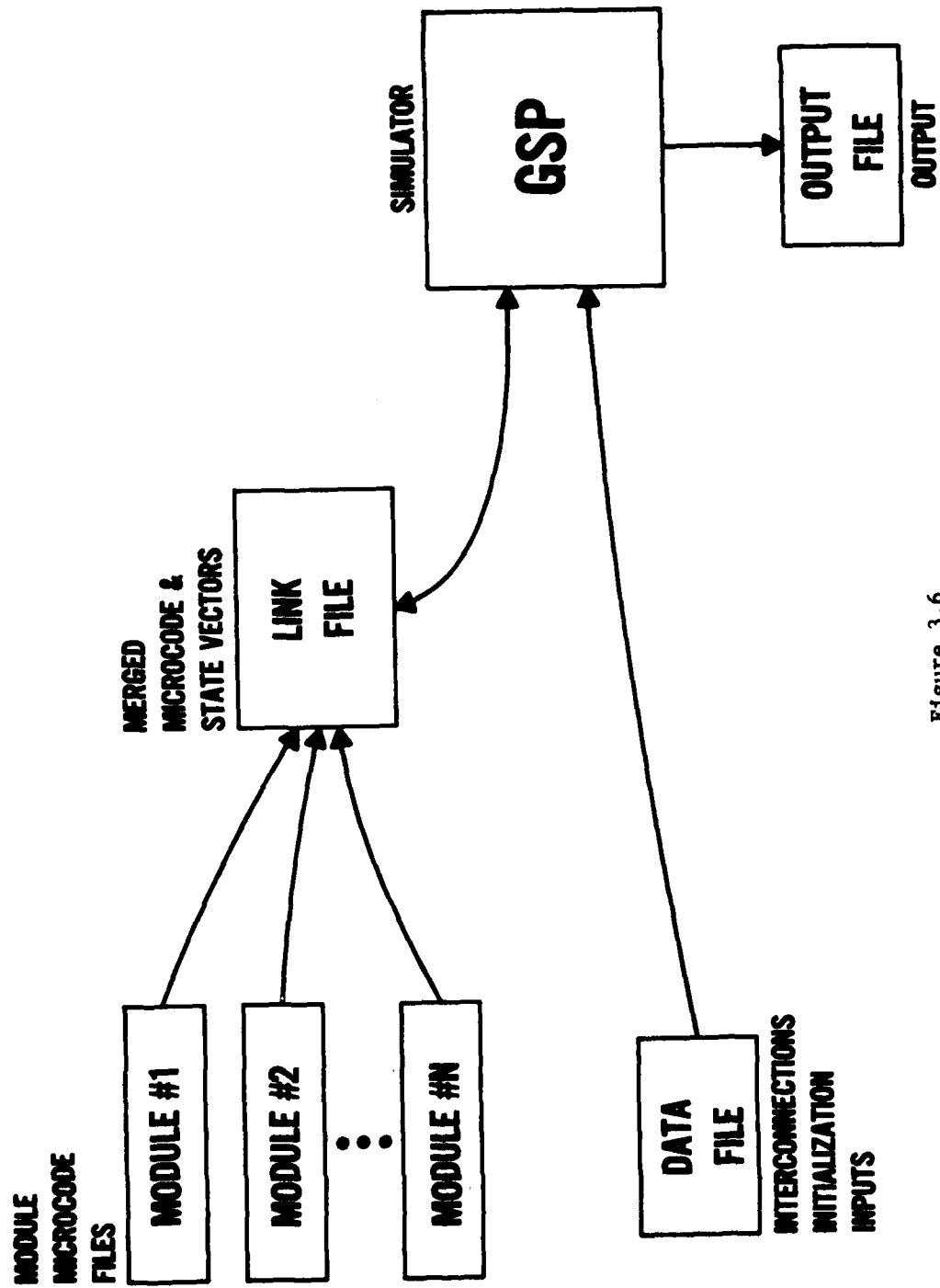


Figure 3.6

3.5. Fault Injection Experiments

The purpose of constructing the simulation model was to conduct "fault injection experiments" in order to assess the effectiveness of the test software. The first step in the process is the injection of the fault into the module microcode. This is done by assessing the effect the fault has on the module response and then incorporating this effect into the model. This incorporation is accomplished by: (1) deleting module code or (2) modifying module code or (3) adding additional module code or (4) some combination of (1), (2) and (3).

This contract was the first time we had ever done this on a large scale. Our approach to doing this was therefore "ad hoc", but we will study the overall results and attempt to derive general principles for fault insertion.

A full list of the faults injected for each chip are given in appendix E. However, they can loosely be divided into three categories:

(1) Incorrect microoperations - Examples: "Incorrect operation of carry flag for subtract instruction" (CPU), "multiple register select on read, selecting C also selects L" (CPU), "Bit set/reset command clears Port C output completely" (8255).

(2) stuck at faults

Examples: "Data bus line 2 from the 8228 stuck at zero", address line ADO stuck open (ROM)", and "status register stuck at all zeros" (8251).

(3) timing faults.

Examples: "Write pulse must be 500NS to write correctly instead of the specified 250 NS (8251)" or "Hold times for address and data had to be too large--300NS from the end of the write pulse" (8255).

A basic question that might be asked is how were the faults chosen. With the very high reliability of LSI devices [7], the most likely faults will be interconnect faults, e.g. cold solder joints and printed circuit board defects. Therefore, in our simulation of faults, a high priority was given to interface defects: 43 per cent of the faults injected were interface faults.

In selecting interior chip faults to simulate, the ideal approach would be to first compile a list of the most likely faults from literature data and information obtained directly from the manufacturer. The faults to inject could then be selected from this list. In the case of memory devices, failure modes are well documented [2] so that we were able to do this. For the other chips in the system, i.e. the 8080 processor and it's support chips, no such data is available. We contacted people involved in fault simulation at INTEL and their reply was

that they knew of no fault history for their chips. In light of this situation, we decided that the next best approach was to have the simulation modeler of each chip select the faults, e.g. the person who developed the model for the 8251 would compile a fault list for that chip. Also, to as great a degree as possible, the selector of the faults should not be aware of the characteristics of the test programs. We followed these guidelines as closely as possible given the limited number of people working on the project at one time (4-5) and believe we were able to select an unbiased set of faults to test the self test software.

In conducting the actual injection experiments we wanted to insure that the necessary fault information was collected. To insure this, a standard Fault Injection Experiment Record form was used for each experiment. Figure 3.7 gives an example of this. The Fault Description, as its name implies, describes the physical fault that is inserted, in this example: "data line D0 to 8251 shorted to ground." The System Configuration category lists the module files that were used for the run. The Initial Conditions and Input categories are self explanatory. This information is stored in a DATA file so the name of that file is specified here. The test routine that was being executed is recorded in the "Program Executed" category. A description of how the fault manifested itself is given in the Fault Syndrome category. Finally space is given for additional comments.

We did not include the Fault Experiment Record for each experiment in the report. (They are on file in the Department of Electrical Engineering at VPI&SU.) Instead we prepared for this report, as appendix E, a Fault Experiments Summary. The summary has an entry for each experiment. The entry contains a "description of the fault" and the test results. Test results are classified as detected, program control lost, or not detected. Under our system test concept, a fault can be detected in two ways: (1) the test routine detects the fault or (2) the test routine hangs up in an infinite loop due to lack of response from some section of the hardware. In this second case, a "watch dog timer" would time out indicating a system failure. Loss of program control means that the processor receives faulty instruction data from the ROM and therefore is no longer executing the test routine. It is highly probable, we believe, that a time out will occur in these cases also, but we list them separately since the probability of detection is not unity.

Fault Injection Experiment Record

Date: 8-25-1

Fault Description: Interconnect Fault #8251IB, Data line D₀ to 8251
shorted to ground

System Configuration: SYS51

RAM32RB

A8255V6

B8228

A8080N

CSL

BUS

TEST8251 (ROM)

A8251V5

Initial Conditions

Input

Program Executed

A8251IB DATA A1

TEST8251 SOR A1

Fault Syndrome: Mode word and command word to 8251 were modified to
4C and 14 respectively instead of 4D and 15.

Test incomplete. Processor hangs up. Hardware timer
should detect the fault.

Comments:

Figure 3.7

3.6 Analysis of Fault Coverage

Table 3.7 below gives a numerical summary of the results of the fault injection experiments.

<u>System Component</u>	<u>DET.</u>	<u>PCL</u>	<u>NOT DET.</u>	<u>Totals</u>
CPU	33	2	1	36
8228	4		2	6
BUS		3		3
ROM	1	3	1	5
RAM	2	2	3	7
8255	31		7	38
8251	21		2	23
Totals	92	10	16	118
Percent	78	8	14	100

Table 3.7

The data shows that of the 118 faults injected, 92, or 78 percent would definitely be detected, 102, or 86 percent would probably be detected, while 16, or 14 percent, would definitely go undetected. The table also shows, in particular, the sensitivity that the system has to faults in the data path involving the ROM, BUS and 8228. Of the total of 15 faults injected in these three modules, only 3 (20 percent) are definitely detected, 8 (53 percent) are probably detected, while 4 (27 percent) went definitely undetected. This is because these faults effect the instructions that the processor reads and thus would alter the program flow. The coverage of internal CPU faults, on the other hand, was excellent, with 92% definitely detected and 97% probably detected. This compares well with the data given in reference 8.

4. SELF-TEST HARDWARE EXPERIMENTATION AND DOCUMENTATION

The hardware added to the microprocessor system for self-testing was kept to a minimum. The hardware consists of three basic parts: wraparound, isolation, and control logic for self-testing I/O ports; timers and control for initiating the self-tests; and a display for reporting the test results. In addition, some system ROM (1.5K) is required to store the self-test routines and 8 bytes of system RAM must be dedicated to the self-test. Figure 4.1 shows a block diagram of a self-testing 8080 system.

4.1 Experimental Verification of Self-Test System

The self-test described in Appendix A has been successfully verified on the self-test board. A "user" program was run in the foreground to ensure that the self-testing did not interfere with user programs. The program read characters from the console terminal (through the 8251), echoed them, and printed back the whole line of text when a carriage return was typed. It had no problems executing--even through the 8251 tests--despite the fact that self-test passes were made every 75 ms instead of about once a second as would probably be the case in practice.

4.2 Status Display

The self-test displays system status on two LEDs and, optionally, on a 7-segment display. The first LED represents

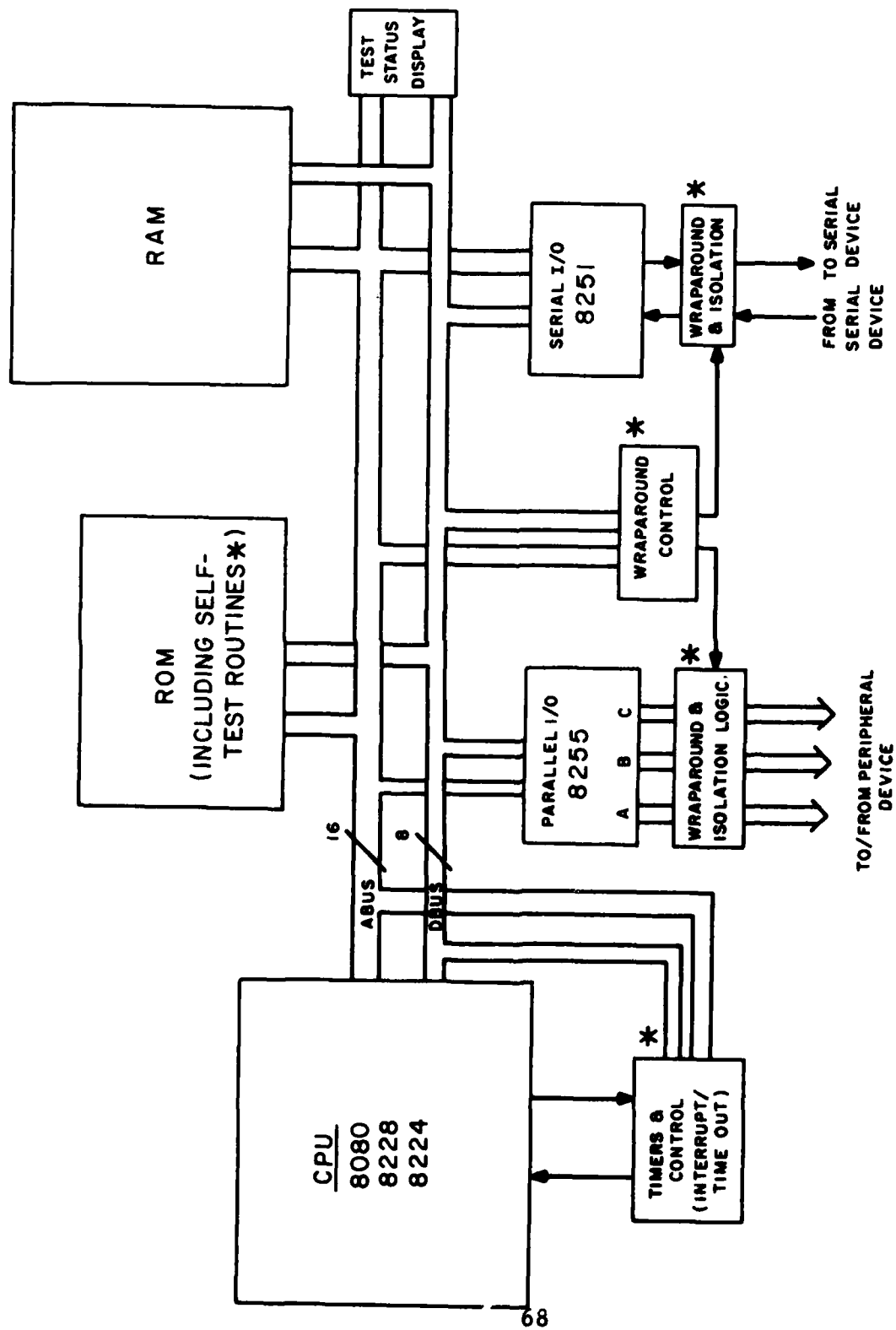


Figure 4.1: Functional Block Diagram of Self-Test System

* indicates logic added for self-testing.

NOT ERROR, it is normally on to indicate NO-ERROR and is extinguished to indicate ERROR. The second LED is a "heartbeat" indicator. It is toggled on and off by successive test passes (or by every nth test pass if n passes are made per second) so that the LED blinks on and off about once per second. This provides a visual indication that the system is successfully self-testing at (about) the proper rate. Note that should either LED burn out an error condition results. The LEDs do, of course, require monitoring. Figure 4.2, shows the LEDs and some of the driving circuitry. The 74373 8-bit latch is used as a memory mapped output port (at address ADDR1); a one or zero (alternately) is written to control bit B0 at the end of each successful test pass (or each n passes) to make the "heartbeat" blink. The ERROR signal driving the ERROR LED is generated by logic to be described in the next section of the report.

The 7-segment display, also shown in Figure 2, can be included to allow some fault location; that is, different codes are written to the display by different tests (CPU, 8255, ROM, RAM, 8251) so that a fault may be isolated to a specific element or card. Since our self-test was not designed for fault diagnosis, many faults in one module are detected by the self-test routine for a different module. For example, a memory fault could cause the CPU memory read/write test to fail, indicating a bad CPU when really the memory is at fault. The

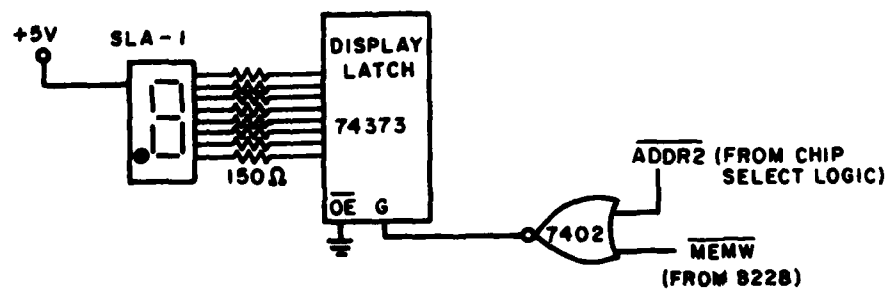
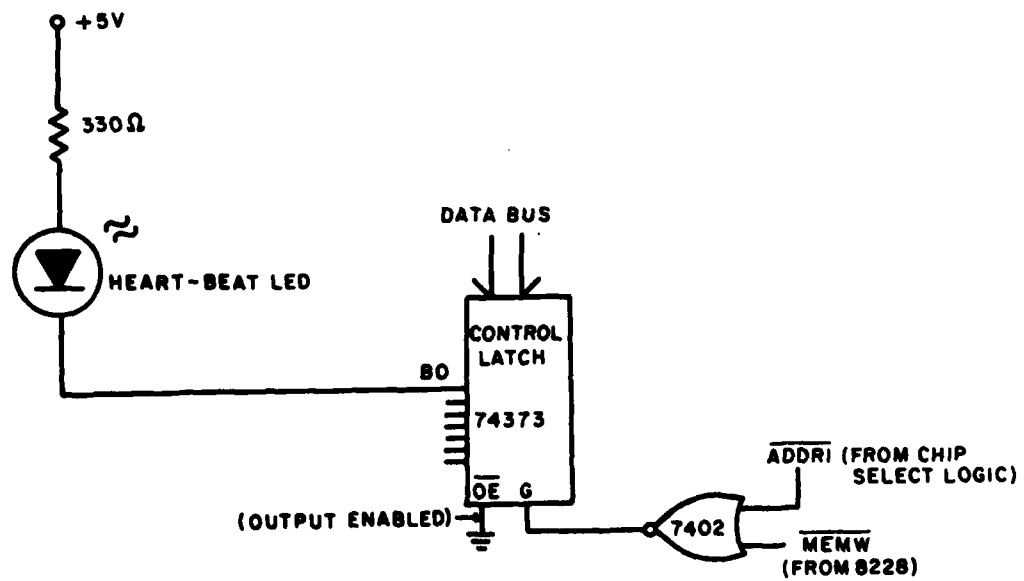
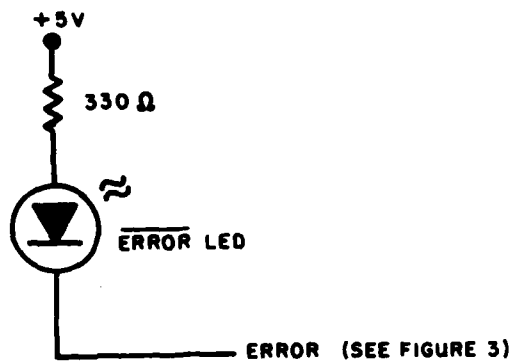


Figure 4.2: Test Status Displays.

display was included on the self-test prototype board mostly as a development aid, but can be incorporated into the system self-test if desired.

Of the three displays, the "heart-beat" provides the best indication that the system is up and running correctly because it must blink on and off (at approximately the right rate). However, certain failures in the self-test hardware could cause loss of the "heartbeat" while the ERROR LED would stay lit (indicating NO ERROR). The two indicators provide a measure of redundancy that allows detection of errors in the self-test hardware itself.

4.3 Timers: Interrupt and Timeout

Figure 4.3 shows the counters and control logic needed to generate the periodic self-test interrupt calls and the hardware timeout. Note that an 8253 Programmable Interval Timer provides the two counters (with one spare); the counters are software controlled. This has the advantage of allowing different test cycle times for different passes of the self-test (an 8251 test pass, for example, requires more time than a ROM test pass). The disadvantage is that the system must function correctly to initialize the counters; failure to do so, however, will be indicated by the loss of the "heartbeat". Both timers (0 & 1) are configured in Mode 0 so that the outputs, initially zero, will change to one and stay there upon the terminal count;

counting will then be suspended until a new start value is loaded.

Timer 0 is responsible for generating the interrupt request while Timer 1 provides the hardware timeout feature. The 8080 loads Timer 0 with the count value for the desired time between self-test passes, T_0 , and loads Timer 1 with a slightly larger value, $T_1 = T_0 + \delta$. The counters are started simultaneously by writing ones to control bits C0 and C1 of the CNTL latch (at memory mapped address ADDR3), thus enabling the Gate inputs on the 8253. When Timer 0 reaches T_0 , OUT 0 rises from zero to one, clocking a one (C0) through F/F #1 to generate the interrupt request; Timer 1 will still be counting. If the self-test has not begun within time δ after T_0 , Timer 1 will time out and generate the ERROR condition (latched by F/F #2). Time δ is the maximum time needed for the 8080 to process the interrupt, save user registers, and stop Timer 1. Thus Timer 1's first function is to ensure that the self-test is initiated within the allotted time.

Upon interrupt acknowledge, F/F #1 is cleared and an RST instruction is gated onto the Data Bus by the 74244. (The 74244 may be omitted if RST7 is used for the self-test, since that opcode is FF Hex.) The self-test has now been successfully initiated and zeroes are written to CNTL bits C0 and C1, disabling the timers. Timer 1 is now loaded with a new count

value corresponding to the length of the test to be performed on this pass (test times vary depending on what is being tested). A one is then written to CNTL bit C1 to start this timer. Upon successful completion of the test pass, Timer 1 is stopped (by writing a zero to CNTL bit C1), Timers 0 and 1 are reloaded with T_0 and T_1 , respectively, and the timers are started (by writing ones to C0 and C1) to await the next test pass. If, for any reason, the self-test does not finish within the allotted time interval, Timer 1 will time out and generate the ERROR signal. Thus Timer 1's second function is to ensure the self-test pass reaches a timely completion (or an error is generated).

During all previous write operations to the CNTL latch (actually writes to address ADDR3), bit C2 was a logic one. If the self-test detects a fault, a zero is written to C2, presetting F/F #2 to generate the ERROR signal; the processor then halts (since successful return of control to the user program is not guaranteed). Thus ERROR may be generated by either a hardware timeout (Timer 1) or by software. The ERROR signal turns off the ERROR LED and is available to the user for any other desired action. Note that upon power-up the ERROR signal may be true (indicating ERROR) for a short while before the software initializes the 8253 and CNTL latch. A one-shot could be used to eliminate this possibility, if necessary.

The hardware shown in Figure 4.3 allows complete software control of the self-testing and hardware timeout. This allows a user program to suspend self-testing during a time-critical operation. The user program simply writes zeroes to CNTL bits C0 and C1 (and a one to C2) to suspend testing, and then writes ones to C0, C1, and C2 to resume self-testing. Note that the "heartbeat" LED will be affected by a long suspension, so that suspensions for over 100 ms or so are not recommended. Note also that if the user program fails to resume self-testing, then the "heartbeat" will stop altogether, indicating a fault.

The "heartbeat" also minimizes the effect of a fault in the hardware of Figure 4.3 or in the software control. If tests are not successfully completed at (about) the proper rate, the "heartbeat" will indicate a fault even if the ERROR LED is still lit.

4.4 Hardware Required to Self-Test I/O Ports

Figures 4.4a and 4.4b illustrate the logic needed for self-testing parallel and serial I/O for an 8080 system. The 8255 parallel I/O port is especially easy to test in Mode 0 since each of the three ports can be configured as either input or output; this allows data output to one port to be input through another port of the same chip. Similarly, the 8251 (being a full duplex USART) has both serial input and serial output so that another UART is not required to self-test it.

As shown in Figure 4.4a, the self-test logic for the 8255 consists of three (8-bit) isolation buffers (I1-I3), two bidirectional (8-bit) wraparound buffers, and another control latch, memory mapped at address ADDR4. Buffers I1, I2, and I3 isolate the 8255 from its peripheral devices during self-testing of the 8255. Figure 4.5 shows the three possible implementations for an isolation buffer. In Figure 4.5a, the 8255 port is configured as an output port; the 74373 (octal D-type latch) outputs follow the inputs (from the 8255 port) during normal operation. During testing, the TESTING signal goes low and the 74373 latches its current outputs; thus the user signals are preserved during the test and the peripheral device never sees the test patterns. The 8255 output value is restored upon successful completion of the test, and TESTING is set high once again for normal operation. The TESTING signal is supplied by the CNTL latch bit X4 as shown in Figure 4.4.

In Figure 4.5b, the 8255 port is configured as an input port; now the 74373 outputs feed the peripheral data into the 8255 during normal operation. During testing, the TESTING signal goes high, tri-stating the 74373 outputs. Thus the peripheral input data is prevented from conflicting with test patterns. Figure 4.5c shows a special case; here the 8255 port is sometimes used as an input port and sometimes as an output port. The Y signal is controlled by the user (through a latch-

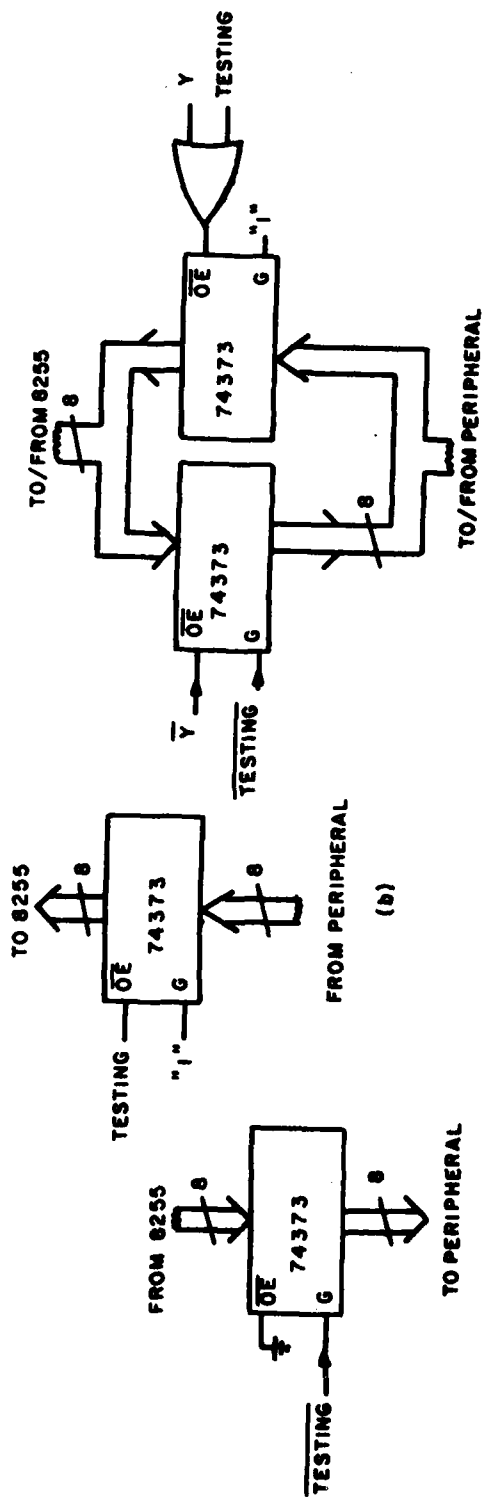


Figure 4.5: Details of the Parallel I/O Isolation.

DBUS

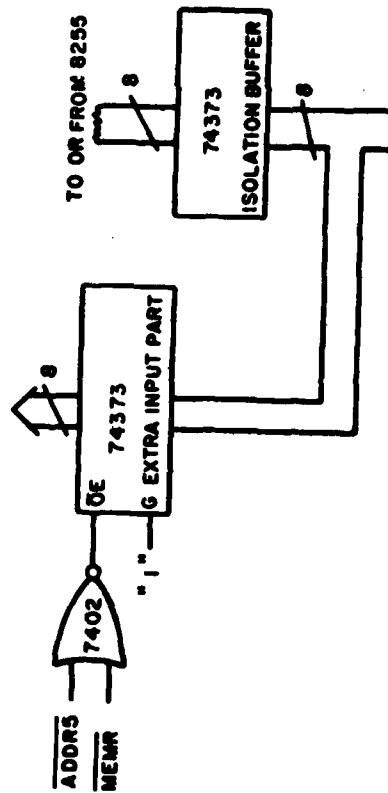


Figure 4.6: Testing Parallel Isolation Buffer.

like CNTL) to define the direction of data flow. This case is merely a combination of the first two cases. During testing, the lefthand 74373 latches its data (in case the port is an output, $Y = 1$) while the righthand 74373 is tri-stated (in case the port is an input, $Y = 0$).

The wraparound buffers, W1 and W2, are 74245's, octal bus transceivers with tri-state outputs. During normal operation, their EN inputs (X0 and X2) are high, and both directions have outputs tri-stated. During testing, CNTL bits X0 and X2 are set at zero and bits X1 and X3 are used to define the direction of data flow. Thus each 8255 port can be tested as both an input port and output port.

The serial I/O self-test logic, as shown in Figure 4.4b, requires two tri-state buffers to isolate the serial peripheral device and one more tri-state buffer to provide wraparound (requiring one 74125). Only one control bit (X5) is required to define either normal operation or self-test mode (wraparound). During testing, the serial input is connected to the serial output so that the 8251 receives what it transmits; the peripheral device is isolated by the tri-stated buffers. During normal operation, the wraparound buffer is tri-stated and the other two buffers enabled for normal serial I/O.

One important consideration here is the effect of a fault in the isolation/ wraparound hardware. A fault in a wraparound

buffer would be detected by the I/O port's self-test. However, a fault in an isolation buffer would not be detected. The isolation can be tested, but an additional input port is required and, more importantly, the testing must be done while writing to or reading from the peripheral device. This requires knowledge of the peripheral device. Figure 4.6 shows an example of testing parallel I/O isolation buffers. If the isolation buffers passes data out to the peripheral, data must be written to the 8255, input through the extra port (ADDR5), and compared. If the isolation buffer passes data in from the peripheral, data must be read through both the 8255 and the extra port, and compared. Thus data from both sides of the isolation buffers is compared. For serial I/O, an extra UART would replace the extra 74373 parallel input port. Again, since the peripheral will either receive or provide the test data, this testing must be part of the applications program.

5. CONCLUSIONS

The main conclusions that can be drawn from this report are:

(1) It is possible to develop efficient self-test routines for detecting faults in the processor, memory, and support chip areas of a microprocessor system. These routines comprise the essential element of a total self-test strategy for microprocessor systems. We found that the CPU and parallel I/O port tests required the least amount of execution time, in fact both of these tests are made in a single test pass in 2 ms, while RAM memory testing required by far the most time, approx. 80 ms for 1K. ROM testing also required a fair amount of time, approx. 24 ms for 1K. These figures are for a 2MHz clock rate. Serial I/O testing depends almost entirely upon the baud rate employed as opposed to the other tests which depend on the processor clock rate.

Fault injection experiments indicate that the fault coverage of the self-test strategy is approximately 80%. Failures detected by self-test mechanisms include not only those detected by the self-test routines directly, but also those uncovered by "watch dog" timer mechanisms. This second category of faults is characteristic of situations in which the system becomes totally unresponsive and when the self-test hardware is itself faulty.

(2) A chip level simulation model is an effective tool for evaluating self-test software. This model was used to construct fault injection experiments in order to assess the effectiveness of the self-test software. The results of these simulation experiments were used to calculate the 80% fault coverage figure mentioned above.

The fact that the model was constructed and was used to evaluate self-test software constitutes an important contribution to the state of the art in system validation. To date, accurate modeling and simulation of LSI devices has been prohibitively expensive in many validation situations. The work carried out in this contract has demonstrated the effectiveness of the GSP simulation language in solving this problem.

(3) An effective self-testing system requires only a small amount of self-test hardware. An actual 8080 hardware system was constructed and put into full working order. All self-test routines were run on this system to verify that: (a) the self-test routines will actually run on real equipment (b) the self-test routines, when finished with their execution, leave the system in a state compatible with an operational program (c) that the small amount of self-test hardware that was added, functioned as expected. Our laboratory system operation verified that all three of these requirements were satisfied.

In summary, then, we feel that we have met the three goals of the research contract. In doing so, we have developed an approach to the self-test of microprocessor systems which has been demonstrated as being effective. In addition we have accumulated a large base of concepts and ideas for further important research in the areas of system self-test and system modeling and simulation. We will present some of these ideas in the next section.

6. RECOMMENDATIONS FOR FURTHER RESEARCH

As was emphasized in the conclusion section, a large body of important knowledge has been accumulated in completing Air Force Contract F30602-80-C-0200. In light of this we make the following recommendations for further research:

Future Study

(1) Develop additional self-test library programs that provide greater fault coverage. The logical next step would be to develop a self-test that achieved coverage as close to 100% as possible without imposing any time constraints. This would provide an indication of the time required for such a test. This report describes such a test for RAM memory that requires approximately 20 seconds to test 1K of memory. It would be useful to know the time and the amount of added hardware required to achieve maximum coverage for the CPU as well. Next it would be useful to develop a set of programs with intermediate execution times and fault coverage. The system designer could then select the self-test that best suits his needs. The tradeoffs would be execution time and added hardware cost for additional fault coverage.

(2) Extension of the self-test techniques to other microprocessor technologies. The self-techniques that were developed were applied to an 8080 system. An important extension of this work could be made in two areas. First, self-

test techniques could be developed for microprocessor systems which possess a higher level of integration. A first step here might be the 8085 system which combines the functions of the 8080 processor and the 8228 and 8224 support chips into one chip. The 8085 has essentially the same instruction set as the 8080, so that this effort would constitute a rather straight forward extension of the present research results. Next, self-test techniques could be developed for a full microcomputer on a chip, such as the Motorola 6802. These chips extend the level of integration present in the 8085 by having both RAM and ROM memory in the chip. The obvious cost and reliability advantage of such chips will dictate their use in avionics systems and it is important that self-test techniques be developed for these chip types also. The second possible area of extension, could be to 16-bit microprocessors such as the 8086, MC68000, and the Z8000. The greater computational power and accuracy of these chips will no doubt result in their extensive use in avionics designs and self-test techniques should be developed for them.

(3) Abstract the features of the various library programs, possibly using a branch of formal mathematics, that would allow the tradeoff to be defined in general terms applicable to variety of microprocessor systems.

(4) Development of simulation models for other microprocessor technologies. This effort would support the work

AD-A118 826

VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG D--ETC F/G 9/2
MICROPROCESSOR SELF-TEST: SOFTWARE SELF-TEST FOR AN 8080-BASED --ETC(II)
JUN 82 J R ARMSTRONG, F G GRAY F30602-80-C-0200

UNCLASSIFIED

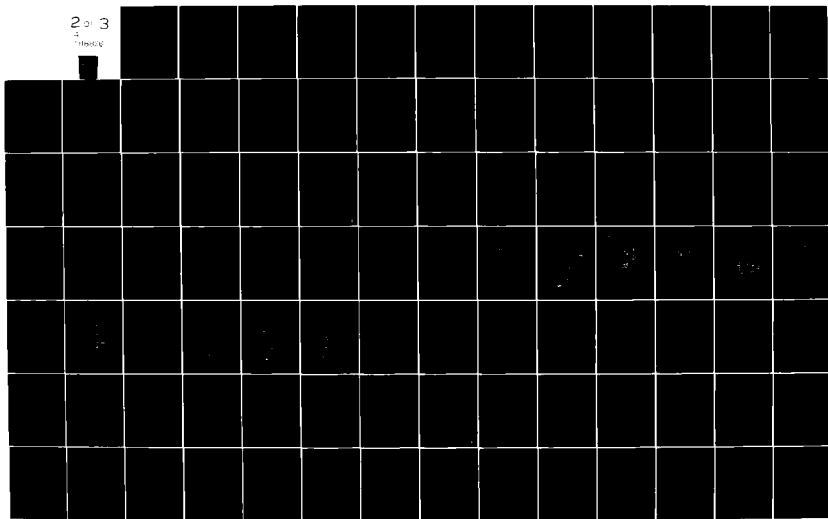
RADC-TR-82-80

NL

2 of 3

1

THINER



in (2) by allowing fault simulations to be done for these other microprocessor technologies as well.

(5) Development of systematic approaches to modeling of both good and faulty LSI devices. We now have in place, a complete, chip level model of a microprocessor system. Such a resource offers great opportunity for exploring various approaches to the modeling of good and faulty LSI devices. Information gained in such a study, plus that gained in the current contract F30602-80-C-0200 should allow us to develop systematic approaches to this modeling.

References

- [1] Jonge, J. Henk and A. J. Smulders, "Moving Inversions Test Pattern is Thorough, Yet Speedy," IEEE Tutorial: LSI Testing, IEEE Computer Society, Long Beach, CA 1978.

- [2] Fee, Warren G., "Memory Testing," IEEE Tutorial: LSI Testing, IEEE Computer Society, Long Beach, CA 1978.

- [3] Product Evaluation Group \RADC/RBRM] and Howard Dicken, MC8080A Microprocessor and Related Peripheral Devices--Product Evaluations, RADC, Griffiss AFB, NY 1978.

- [4] Intel Corporation, MCS-80 User's Manual, 1977.

- [5] Kane, Jerry with Adam Osborne, An Introduction to Microcomputers, Volume 3: Some Real Support Devices, Osborne & Associates, Inc., Berkeley, CA 1978.

- [6] Devlin, Donald E., "A Chip Level, Multimodule Logic Simulator," Masters Thesis, EE Department VPI&SU, May, 1981.

[7] Queyssac, D., "Projecting VLSI's Impact on Microprocessors," IEEE Spectrum, Vol. 16, No. 5, 1979, pp 38-41.

[8] Baruso, S. J., McGough J. J., and Sworn, F. "Latent Fault Modeling and Measurement Methodology for Application to Digital Flight Controls," Proceedings of the Advanced Flight Control Symposium USAF Academy, Colorado Springs, Colorado, Aug. 5-7, 1981.

Appendix A

Self Test Program Listings

```

*****
***      RADC MICROPROCESSOR SELF-TEST PROJECT      ***
***      *****                                     ***
*****
;
; START EQU 0A00H ; INTERRUPT ENTRY POINT FOR SELF-TEST PASS
;
; 'INIT' IS DEFINED BY ITS POSITION IN THE FILE; CHECK THE
; SYMBOL TABLE AFTER COMPILE FOR ITS VALUE.
;
; ERX EQU 6 ; RST NUMBER FOR ERROR EXIT (RST 6)
;
; ***** CONFIGURATION DEPENDENT SYMBOLS DEFINED: *****
;
; RAMBEG EQU 0F00H ; START ADDRESS OF RAM
; RAMTOP EQU 18H ; HIGH BYTE OF RAM END ADDR PLUS ONE!
;
; ROMBEG EQU 0A00H ; START ADDRESS OF ROM
; ROMTOP EQU 0FH ; HIGH BYTE OF ROM END ADDR PLUS ONE!
;
; PRT55 EQU 3C3CH ; MEMORY MAPPED ADDR. OF 8255
; PRT51 EQU 3C2CH ; MEMORY MAPPED ADDR. OF 8251
;
; CNTRL1 EQU 3C28H ; CONTROL FOR 8251 WRAPAROUND & BAUD RATES
; CNTRL2 EQU 3C30H ; 8255 WRAPAROUND & 8253 TIMER/TIMEOUT CONTROL
; CNTR EQU 3C36H ; 8253 TIMER MEM. MAPPED ADDR, COUNTER 0
;
; ***** STORAGE LOCATIONS: *****
;
; TSTAD EQU 1700H ; 2 BYTES FOR TEST ROUTINE ADDRESS
; RBEG EQU TSTAD+2 ; 2 BYTES FOR RAM/ROM TEST START ADDR
; RANDOM EQU RBEG+2 ; 1 BYTE FOR RAM TEST 'RANDOM' PATTERN
;
; RAM1 EQU RANDOM+1 ; 3 BYTES FOR MEMORY WRITE TESTING
; RAM2 EQU RAM1+1 ; (IN CPU TEST)
; RAM3 EQU RAM2+1
;
; BAUDS EQU 178EH ; 1 BYTE FOR BAUD RATE/8251 WRAPAROUND CNTRL
;
; BAUDS MUST BE INITIALIZED BY THE USER BEFORE CALLING
; INIT; BITS 1-3 & 5-7 ARE USER CONTROLLED BAUD RATE BITS;
; BIT 0 IS 1 FOR WRAPAROUND; BIT 4 IS THE HEARTBEAT.
;
; *****
;
; ROM AND RAM TEST SEGMENT SIZES DEFINED:
;
; ROMSS EQU 128D ; TEST ROM IN 128 BYTE SEGMENTS
; RAMSS EQU 32D ; AND RAM IN 32 BYTE SEGMENTS
;
0000
0006
0F00
0018
0A00
000F
3C3C
3C2C
3C28
3C30
3C36
1700
1702
1704
1705
1706
1707
178E
0080
0020

```


A.4

[illegible]

OPCODE	OPERANDS	OPERATION	REGISTER	STATUS
0AC5	G30D0A	JMP	ERR	
0AC8	F7	RST	ERX	
0AC9	76	HLT		
0ACA	AB	XRA	E	;A<-56
0ACB	2F	CMA		;A<-A9
0ACC	93	SUB	E	;A<-00; CY<-0
0ACD	G20D0A	JNZ	ERR	
0ADD	76	MOV	A,	;A<-CC
0AD1	1F	RAR		;A<-66
0AD2	0F	RAR		;A<-33
0AD3	57	MOV	D,	;D<-33
0AD4	78	MOV	A,	;A<-CC
0AD5	B2	ORA	D	;A<-FF
0AD6	FAE0A	JM	OK5	;VERIFY
0AD9	G30D0A	JMP	ERR	
0ADC	F7	RST	ERX	
0ADD	76	HLT		
0ADE	3C	INR	A	;A<-00
0ADF	FA0D0A	JM	ERR	;CHECK S FLAG RESET
0AE2	G20D0A	JNZ	ERR	;AND Z FLAG SET
0AE5	5F	MOV	E,	;E<-00
0AE6	1D	DCR	E,	;E<-FF
0AE7	93	SUB	E	;A<-01
0AE8	EA0D0A	JPE	ERR	
0AEB	27	DAA		;A<-61
0AEC	G605	AD1	05H	;A<-66
0AEE	27	DAA		;A=66 STILL
0AEF	BC	CHP	H	;VERIFY
0AF0	G20D0A	JNZ	ERR	
0AF3	27	DAA		;A<-6C
0AF4	27	DAA		;A<-72
0AF5	FE72	CPI	72H	;VERIFY
0AF7	G20D0A	JNZ	ERR	
0AFA	19	DAD	D	;HL<-99FF
0AFB	19	DAD	D	;HL<-CDFF
0AFC	23	INX	H	;HL<-CDFF
0AFD	23	INX	H	;HL<-CE00
0AFE	16CC	MVI	D,	;D<-CCH
0AF6				
0AF7				
0AF8				
0AF9				
0B00	80	ADD	B	;A<-3E; CY<-1
0B01	8C	ADC	H	;A<-00; CY<-1
0B02	83	ADD	E	;A<-0C; CY<-1
0B03	85	ADD	L	;A=0C STILL; CY<-0
0B04	89	ADC	C	;A<-61; CY<-0
0B05	82	ADC	D	;A<-2D; CY<-1
0B06	D20D0A	JNC	ERR	;VERIFY CY=1

ADDRESS	INSTR	OPERATION	COMMENTS
0809	E2D		
0809	C20D0A		
080E	118A0B		
0811	1A		
0812	EB		
0813	BE		
0814	C20D0A		
0817	17		
0818	DEE8		
081A	C20D0A		
081D	DA0D0A		
0820	C08E08		
0823	D20D0A		
0826	C637		
0828	07		
0829	DE6E		
082B	C20D0A		
082E	3EC6		
0830	C6CE		
0832	F5		
0833	AF		
0834	F1		
0835	D20D0A		
0836	CA0D0A		
083B	F20D0A		
083E	EA0D0A		
0841	27		
0842	FEFA		
0844	CA8B08		
0847	C30D0A		
084A	00		
084B	3E00		
0809	XRI JNZ	DH ERR	VERIFY RIGHT CONTENTS
080E	LXI D, ADDR1		POINT DE TO MEM. LOC
0811	LDAX D		READ CONTENTS TO A: A<-74
0812	XCHG		SWAP ADDR1 TO HL
0813	CMP M		VERIFY SAME ACCESS; CY<0
0814	JNZ ERR		
0817	RAL		A<-E8
0818	SBI 0E8H		NOW VERIFY RIGHT CONTENTS
081A	JNZ ERR		
081D	JC		CY SHOULD BE ZERO NOW
0820	CALL SPTST		GO TEST SP; SET CY BEFORE RETURN
0823	JNC ERR		VERIFY CALL WORKED
0826	ADI 37H		A=00 UPON RETURN
0828	RLC		A<-37
0829	SBI 6EH		A<-6E; CY<-0
082B	JNZ ERR		VERIFY
***** TEST OF PUSH/POP PSW INSTRUCTIONS *****			
082E	MVI A, 0C6H		A<-86
0830	ADI 0CEH		A<-94H
0832	PUSH PSW		FLAGS: AC,CY,S<-1; P,Z<-0
0833	XRA A		WRITE A & FLAGS TO STACK
0834	POP PSW		REVERSE ALL FLAGS & CLEAR A:
0835	JNC ERR		A<-00; AC,CY,S<-0; P,Z<-1
0836	JZ ERR		RESTORE A & FLAGS
083B	JP ERR		VERIFY FLAGS RESTORED
083E	JPE ERR		
0841	DAA		
0842	CPI 0FAH		VERIFY AC=1; A<-FAH
0844	JZ LOGIC		IS IT?
0847	JMP ERR		YES, O.K.
***** CHECKSUM FOR ROM SEGMENT 3 *****			
084A	DB 00H		CHECKSUM FOR ROM SEGMENT 3
***** LOGIC V1.1: TESTS THE LOGIC FUNCTIONS (XOR/OR/AND) OF THE ALU FOR STUCK-AT'S AND FUNCTIONAL ERRORS. ALL GATE INPUT COMBINATIONS ARE APPLIED (00,01,10, & 11) *****			
084B	MVI A, 0		CLEAR A

```

084D 47      MOV      A,      B      ; AND B
084E 0EFF     MVI      A,      0FFH  ; PUT ALL 1'S IN C

0850 AF      XRA      A              ; APPLY 00 TO XOR GATES: A=00 STILL
0851 87      ORA      A              ; APPLY 00 TO OR GATES: A=00
0852 A7      ANA      A              ; APPLY 00 TO AND GATES: A=00
0853 C20D0A  JNZ      ERR           ; VERIFY

0856 A1      ANA      C              ; APPLY 01 TO AND GATES: A=00 STILL
0857 C20D0A  JNZ      ERR           ; VERIFY

085A B1      ORA      C              ; APPLY 01 TO OR GATES: A<-FF
085B A9      XRA      C              ; APPLY 11 TO XOR GATES: A<-00
085C C20D0A  JNZ      ERR           ; VERIFY

085F A9      XRA      C              ; APPLY 01 TO XOR'S: A<-FF
0860 A1      ANA      C              ; APPLY 11 TO AND'S: A=FF
0861 B0      ORA      B              ; APPLY 10 TO OR'S: A=FF
0862 B9      CMP      C              ; VERIFY
0863 C20D0A  JNZ      ERR           ;

0866 B1      ORA      C              ; APPLY 11 TO OR'S: A=FF
0867 A8      XRA      B              ; APPLY 10 TO XOR'S: A=FF
0868 B9      CMP      C              ; VERIFY
0869 C20D0A  JNZ      ERR           ;

086C A0      ANA      B              ; APPLY 10 TO AND'S: A<-00
086D C20D0A  JNZ      ERR           ; VERIFY

*****
MWRITE V1.3: MEMORY WRITE TESTS (STA,STAX,SHLD,MOV M)
*****

0033      PATT1 EQU 33H
00CC      NPAT1 EQU 0FFH - PATT1
00C1      PATT2 EQU 0C1H
AA55      PAT16 EQU 0AA55H

0870 3E33     MVI      A,      PATT1 ; PUT TEST PATTERN IN A
0872 210517   LXI      H,      RAM1  ; POINT HL TO A RAM BYTE
0875 110617   LXI      D,      RAM2  ; AND DE TO NEXT BYTE
0878 010717   LXI      B,      RAM3  ; AND BC TO A 3RD BYTE
087C 02      STAX     B              ; WRITE A TO RAM3
087D 2F      CMA      C              ; AND COMPLEMENT PATTERN TO RAM2
087E 12      STAX     D              ; WRITE ANOTHER PATTERN TO RAM1
087F 36C1     MVI      M,      PATT2 ; READ PATT1 BACK FROM RAM3
0880 0A      LDAX     A              ; VERIFY
0881 EE33     XRI      A              ;
0883 C20D0A  JNZ      ERR           ;
0886 3A0617   LDA      RAM2         ; READBACK PATT1-BAR FROM RAM2
0889 FECC     CPI      NPAT1        ; VERIFY
088B C20D0A  JNZ      ERR           ;

```


A.10


```

0C31 3E02 MV1 A, 02H ; O.K., PORT B IS AN INPUT
0C33 B0 ORA B, A ; OR THIS INTO CMD WORD
0C34 47 MOV B, A ; STORE IN B

;
N2: 23 INX H ; POINT TO PORT C
0C35 56 MOV D, M ; SAVE (POSSIBLE) PORT C OUTPUT VALUE
0C36 71 MOV M, C ; WRITE PATTERN
0C37 7E MOV A, M ; READ BACK
0C38 B9 CMP C ; OUTPUT?
0C39 B9 JZ N3 ; YES
0C3A 90 MOV A, N3 ; NO, MAKE SURE WE READ BACK FF
0C3B 3C INR A ; O.K.
0C3D 3C JZ N2A ; PORT C SPLIT MODE WILL 'FAIL' HERE
0C3E 3C JMP ERR
0C41 C30D0A

; #####
;
; #####
;
N2A: 00 DB 000H ; CHECKSUM FOR SEGMENT 5
; #####
;
; #####
;
N3: 4A MOV C, D ; MOVE PORT C VALUE TO REG. C
0C49 C5 PUSH B ; SAVE CONFIG. WORD & (POSS) PORT C OUTPUT VAL
0C4A ;
0C4B 23 INX H ; POINT TO CMD PORT
; #####
; ***** MODE 0 PARALLEL I/O PORT TEST (8255A) *****
;
MOPIOT: 368B MV1 M, 8BH ; SET MODE 0; DEFINE A AS OUTPUT, B & C AS INPUT
0C4E 3E26 MV1 A, 26H ; WRAP A AROUND TO B & C...
0C50 32303C STA CNTL2
; *****
0C53 45 MOV B, L ; SAVE CMD. ADDR. IN B
0C54 7D MOV A, L ; PUT ADDR. IN A SO WE CAN...
0C55 D603 SUI 03H ; ADJUST TO PORT A ADDR.
0C57 6F MOV L, A ; MOVE BACK TO L
0C58 4F MOV C, A ; SAVE THIS IN C
;
0C59 3E55 MV1 A, 55H ; PUT CHECKERBOARD IN A AS 1ST PATTERN
0C5B CD2F0D CALL RVBC ; WRITE IT, READ & VERIFY B & C
0C5E 3EAA MV1 A, 0AAH ; 2ND PATTERN
0C60 CD2F0D CALL RVBC ; 3RD PATTERN
0C63 3E00 MV1 A, 00H
0C65 CD2F0D CALL RVBC
;
0C68 3E06 MV1 A, 06H ; WRAP B AROUND TO A & C
0C6A 32303C STA CNTL2
; *****

```

A.13

3090 MACRO ASSEMBLER, VER 2.4
ERRORS = 0 PAGE 14

Address	Operation	Comments
0CB5	EB	
0CB6	3604	
0CB8	3607	
0CBA	EB	
0CB8	7E	
0CB8	FEDA	
0CBE	C20D0A	
0CC1	EB	
0CC2	360C	
0CC4	3608	
0CC6	EB	
0CC7	7E	
0CC8	FE8A	
0CCA	C20D0A	
0CCD	EB	
0CCE	3601	
0CD0	3608	
0CD2	3605	
0CD4	EB	
0CD5	7E	
0CD6	FEAF	
0CD6	C20D0A	
0CD8	EB	
0CDC	3606	
0CDE	360E	
0CE0	360D	
0CE2	EB	
0CE3	7E	
0CE4	FE67	
0CE6	CAED0C	
0CE9	C30D0A	
0CEC	00	
0CED	EB	
0CEE	3609	
0CF0	360A	
0CF2	3602	
0CF4	EB	
0CF5	7E	
0CF6	FE55	
0CF8	C20D0A	

[illegible]

```

*****
**                                     **
**          R A M   T E S T          **
**                                     **
**      VERSION:  2.1      DATE:  7/25/81      **
**                                     **
**      DESCRIPTION:  NONDESTRUCTIVE QUICKIE RAM TEST.      **
**      TESTS RAM ONE LOCATION AT A TIME USING 'RANDOM'      **
**      PATTERNS.  DESIGNED FOR SERIES TEST, 32 BYTES/PASS  **
**                                     **
*****
THIS ROUTINE PERFORMS A QUICKIE RAM TEST BY PERFORMING
A WRITE/READ-VERIFY ON EACH RAM LOCATION IN THE BLOCK
WITH EACH OF THE FOLLOWING PATTERNS:
(1)  COMPLEMENT OF ORIGINAL CONTENTS
(2)  RANDOM PATTERN
(3)  COMPLEMENT OF (2)
(4)  ORIGINAL CONTENTS (RESTORATION/VERIFICATION)

** THE TEST DOES NOTHING TO CHECK ADDRESS DECODING. **

NOTE THAT THIS IS A NONDESTRUCTIVE TEST (ORIGINAL RAM CONTENTS
ARE RESTORED).
*****
RAMTS:  3599      99H      ; DISPLAY A '4' ON 7-SEGMENT DISPLAY
0096      STA      A, PRT55+1      ; POINT TO TIMEOUT COUNTER (CNTR 1)
0098      LXI      H, 123D      ; LOAD LSB OF TEST TIME (IN CLOCK CYCLES)
009E      MVI      M, 0      ; AND THEN MSS
00A0      MVI      A, 0F6H      ; START TIMEOUT COUNTER #
00A2      STA      C, CMTL2      ; ENABLE THE TIMEOUT
00A4      ;                                     ;
00A7      LHLD     RAMSS      ; PASS START ADDRESS IN HL
00AA      MVI      D, RAMSS      ; TEST RAMSS BYTES OF RAM PER PASS
00AC      LDA      RANDOM      ; LOAD RANDOM PATTERN
00AF      MOV      C, A      ; AND PASS IT IN C
00B0      ;                                     ;
00B1      READQ:  MOV      B, M      ; READ A RAM LOCATION; SAVE IN B
00B3      MOV      A, B      ; MOVE IT INTO A TOO
00B5      ;                                     ;
00B8      CHA      M, A      ; COMPLEMENT THE PATTERN
00BA      MOV      M, M      ; WRITE THIS
00BC      CHP      M, ERR      ; VERIFY
00BE      JNZ      C20D0A      ;
00C0      ;                                     ;
00C1      MOV      A, C      ; LOAD RANDOM PATTERN INTO A
00C3      MOV      M, A      ; WRITE IT
00C5      CHP      M, M      ; VERIFY
*****

```

0088	C20D0A		JNZ	ERR	
008E	2F		CMA		
008F	77	A	MOV	M,	
00C0	BE		CHP	M	
00C1	C20D0A		JNZ	ERR	
00C4	70	B	MOV	M,	
00C5	7E		MOV	A,	
00C6	86	M	CHP	B	
00C7	C20D0A		JNZ	ERR	
00CA	23		INX	H	
00CB	15		DCR	D	
00CC	C2B000		JNZ	READQ	
00CF	7C	H	MOV	A,	
00D0	FE18		CPI	RANTOP	
00D2	C2EC00		JNZ	RN1	
00D5	79	C	MOV	A,	
00D6	A7		ANA	A	
00D7	CA0D0A		JZ	ERR	
00DA	07		RLC		
00DB	D2E000		JNC	RNO	
00DE	EETC		XRI	1CH	
00E0	320417		STA	RANDOM	
00E3	21F30D		LXI	H,	
00E6	220017		SHLD	TSTAD	
00E9	C3150A		JMP	GOXIT	
00EC	220217		SHLD	RBEG	
00EF	C3150A		JMP	GOXIT	
00F2	00		DB	00H	

;COMPLEMENT THIS PATTERN
 ;WRITE IT
 ;VERIFY

;RESTORE ORIGINAL CONTENTS
 ;READ IT BACK
 ;VERIFY

;ADVANCE TO NEXT POSITION
 ;COUNT OFF A BYTE TESTED
 ;MORE TO DO, SO DO IT

;DONE ALL OF RAM YET?
 ;NOPE, STILL SOME LEFT

;YES, DONE ALL OF RAM; GET RANDOM PATTERN
 ;AND TEST IT TO MAKE SURE
 ;THAT IT'S NOT 0 *(ERROR IF SO)*
 ;SHIFT LEFT: MSB -> LSB, CY
 ;MSB WAS 0, SO WE HAVE THE NEW PATTERN
 ;MSB WAS 1, COMPLEMENT BITS 2,3,4
 ;SAVE NEW RANDOM PATTERN FOR NEXT PASS
 ;LOAD ADDR. OF NEXT TEST OF SERIES
 ;SO WE CAN STORE IT
 ;SAY 'GO' AND EXIT

;STORE CURRENT END ADDR AS NEXT START ADDR
 ;SAY GO AND EXIT

;CHECKSUM FOR ROM SEGMENT 8

A. 20

```

0024      7E      MOV      A,      M      ; READ STATUS
0025      E602     ANI      RXRDY      ; RECEIVED CHAR. YET?
0026      C240E    JZ       L2          ; LOOP TILL IT HAS
0027      7E      MOV      A,      M      ; READ STATUS AGAIN TO CHECK:
0028      E638     ANI      38H        ; ANY ERRORS?
0029      C200A    JNZ      ERR        ; YES, PROBLEMS => NO GO
0030      2B      DCX      H          ; NO, WANT TO READ DATA NOW
0031      7E      MOV      A,      M      ; SO DO SO
0032      B8      CMP      B          ; SAME AS WHAT WE SENT?
0033      C200A    JNZ      ERR        ; ERROR IF NOT
0034      23      INX      H          ; *ADJUST HL FOR CMD
0035      13      INX      D          ; *POINT DE TO NEXT PATTERN
0036      FE00     CPI      LSTPAT     ; WAS THAT THE LAST PATTERN?
0037      0E3A     JNZ      L0        ; NOPE, CONTINUE
0038      C2190E   ;
0039      ;
0040      ;
0041      ;
0042      ;
0043      ;
0044      ;
0045      ;
0046      ;
0047      ;
0048      ;
0049      ;
0050      ;
0051      ;
0052      ;
0053      ;
0054      ;
0055      ;
0056      ;
0057      ;
0058      ;
0059      ;
0060      ;
0061      ;
0062      ;
0063      ;
0064      ;
0065      ;
0066      ;
0067      ;
0068      ;
0069      ;
0070      ;
0071      ;
0072      ;
0073      ;
0074      ;
0075      ;
0076      ;
0077      ;
0078      ;
0079      ;
0080      ;
0081      ;
0082      ;
0083      ;
0084      ;
0085      ;
0086      ;
0087      ;
0088      ;
0089      ;
0090      ;
0091      ;
0092      ;
0093      ;
0094      ;
0095      ;
0096      ;
0097      ;
0098      ;
0099      ;
0100      ;
0101      ;
0102      ;
0103      ;
0104      ;
0105      ;
0106      ;
0107      ;
0108      ;
0109      ;
0110      ;
0111      ;
0112      ;
0113      ;
0114      ;
0115      ;
0116      ;
0117      ;
0118      ;
0119      ;
0120      ;
0121      ;
0122      ;
0123      ;
0124      ;
0125      ;
0126      ;
0127      ;
0128      ;
0129      ;
0130      ;
0131      ;
0132      ;
0133      ;
0134      ;
0135      ;
0136      ;
0137      ;
0138      ;
0139      ;
0140      ;
0141      ;
0142      ;
0143      ;
0144      ;
0145      ;
0146      ;
0147      ;
0148      ;
0149      ;
0150      ;
0151      ;
0152      ;
0153      ;
0154      ;
0155      ;
0156      ;
0157      ;
0158      ;
0159      ;
0160      ;
0161      ;
0162      ;
0163      ;
0164      ;
0165      ;
0166      ;
0167      ;
0168      ;
0169      ;
0170      ;
0171      ;
0172      ;
0173      ;
0174      ;
0175      ;
0176      ;
0177      ;
0178      ;
0179      ;
0180      ;
0181      ;
0182      ;
0183      ;
0184      ;
0185      ;
0186      ;
0187      ;
0188      ;
0189      ;
0190      ;
0191      ;
0192      ;
0193      ;
0194      ;
0195      ;
0196      ;
0197      ;
0198      ;
0199      ;
0200      ;
0201      ;
0202      ;
0203      ;
0204      ;
0205      ;
0206      ;
0207      ;
0208      ;
0209      ;
0210      ;
0211      ;
0212      ;
0213      ;
0214      ;
0215      ;
0216      ;
0217      ;
0218      ;
0219      ;
0220      ;
0221      ;
0222      ;
0223      ;
0224      ;
0225      ;
0226      ;
0227      ;
0228      ;
0229      ;
0230      ;
0231      ;
0232      ;
0233      ;
0234      ;
0235      ;
0236      ;
0237      ;
0238      ;
0239      ;
0240      ;
0241      ;
0242      ;
0243      ;
0244      ;
0245      ;
0246      ;
0247      ;
0248      ;
0249      ;
0250      ;
0251      ;
0252      ;
0253      ;
0254      ;
0255      ;
0256      ;
0257      ;
0258      ;
0259      ;
0260      ;
0261      ;
0262      ;
0263      ;
0264      ;
0265      ;
0266      ;
0267      ;
0268      ;
0269      ;
0270      ;
0271      ;
0272      ;
0273      ;
0274      ;
0275      ;
0276      ;
0277      ;
0278      ;
0279      ;
0280      ;
0281      ;
0282      ;
0283      ;
0284      ;
0285      ;
0286      ;
0287      ;
0288      ;
0289      ;
0290      ;
0291      ;
0292      ;
0293      ;
0294      ;
0295      ;
0296      ;
0297      ;
0298      ;
0299      ;
0300      ;
0301      ;
0302      ;
0303      ;
0304      ;
0305      ;
0306      ;
0307      ;
0308      ;
0309      ;
0310      ;
0311      ;
0312      ;
0313      ;
0314      ;
0315      ;
0316      ;
0317      ;
0318      ;
0319      ;
0320      ;
0321      ;
0322      ;
0323      ;
0324      ;
0325      ;
0326      ;
0327      ;
0328      ;
0329      ;
0330      ;
0331      ;
0332      ;
0333      ;
0334      ;
0335      ;
0336      ;
0337      ;
0338      ;
0339      ;
0340      ;
0341      ;
0342      ;
0343      ;
0344      ;
0345      ;
0346      ;
0347      ;
0348      ;
0349      ;
0350      ;
0351      ;
0352      ;
0353      ;
0354      ;
0355      ;
0356      ;
0357      ;
0358      ;
0359      ;
0360      ;
0361      ;
0362      ;
0363      ;
0364      ;
0365      ;
0366      ;
0367      ;
0368      ;
0369      ;
0370      ;
0371      ;
0372      ;
0373      ;
0374      ;
0375      ;
0376      ;
0377      ;
0378      ;
0379      ;
0380      ;
0381      ;
0382      ;
0383      ;
0384      ;
0385      ;
0386      ;
0387      ;
0388      ;
0389      ;
0390      ;
0391      ;
0392      ;
0393      ;
0394      ;
0395      ;
0396      ;
0397      ;
0398      ;
0399      ;
0400      ;
0401      ;
0402      ;
0403      ;
0404      ;
0405      ;
0406      ;
0407      ;
0408      ;
0409      ;
0410      ;
0411      ;
0412      ;
0413      ;
0414      ;
0415      ;
0416      ;
0417      ;
0418      ;
0419      ;
0420      ;
0421      ;
0422      ;
0423      ;
0424      ;
0425      ;
0426      ;
0427      ;
0428      ;
0429      ;
0430      ;
0431      ;
0432      ;
0433      ;
0434      ;
0435      ;
0436      ;
0437      ;
0438      ;
0439      ;
0440      ;
0441      ;
0442      ;
0443      ;
0444      ;
0445      ;
0446      ;
0447      ;
0448      ;
0449      ;
0450      ;
0451      ;
0452      ;
0453      ;
0454      ;
0455      ;
0456      ;
0457      ;
0458      ;
0459      ;
0460      ;
0461      ;
0462      ;
0463      ;
0464      ;
0465      ;
0466      ;
0467      ;
0468      ;
0469      ;
0470      ;
0471      ;
0472      ;
0473      ;
0474      ;
0475      ;
0476      ;
0477      ;
0478      ;
0479      ;
0480      ;
0481      ;
0482      ;
0483      ;
0484      ;
0485      ;
0486      ;
0487      ;
0488      ;
0489      ;
0490      ;
0491      ;
0492      ;
0493      ;
0494      ;
0495      ;
0496      ;
0497      ;
0498      ;
0499      ;
0500      ;
0501      ;
0502      ;
0503      ;
0504      ;
0505      ;
0506      ;
0507      ;
0508      ;
0509      ;
0510      ;
0511      ;
0512      ;
0513      ;
0514      ;
0515      ;
0516      ;
0517      ;
0518      ;
0519      ;
0520      ;
0521      ;
0522      ;
0523      ;
0524      ;
0525      ;
0526      ;
0527      ;
0528      ;
0529      ;
0530      ;
0531      ;
0532      ;
0533      ;
0534      ;
0535      ;
0536      ;
0537      ;
0538      ;
0539      ;
0540      ;
0541      ;
0542      ;
0543      ;
0544      ;
0545      ;
0546      ;
0547      ;
0548      ;
0549      ;
0550      ;
0551      ;
0552      ;
0553      ;
0554      ;
0555      ;
0556      ;
0557      ;
0558      ;
0559      ;
0560      ;
0561      ;
0562      ;
0563      ;
0564      ;
0565      ;
0566      ;
0567      ;
0568      ;
0569      ;
0570      ;
0571      ;
0572      ;
0573      ;
0574      ;
057
```

0E6B	7E	L3:	MOV	A,	M		; READ STATUS
		:	*	*	*	*	
0E6C	E620		ANI	20H			; DETECT *FRAMING ERROR* YET?
		:	*	*	*	*	
0E6E	CAG80E		JZ	L3			; LOOP IF NOT
0E71	3615		MV1	M,	15H		; STOP BREAK XMIT IF SO; CLEAR ERRS
		:					
0E73	7E	L4:	MOV	A,	M		; READ STATUS
0E74	E604		ANI	TXNTY			; READY FOR CHAR. TO SEND?
0E76	CAT730E		JZ	L4			; NO, LOOP
0E79	2B		DCX	H			; *DATA --
0E7A	3663		MV1	M,	63H		; WRITE PATTERN
0E7C	0E7C		INX	H			; *COMMAND
		:					
0E7D	7E	L5:	MOV	A,	M		; WAIT UNTIL CHAR. HAS BEEN SENT
0E7E	E604		ANI	TXNTY			
0E80	CAT7D0E		JZ	L5			
0E83	2B		DCX	H			; *DATA
0E84	36CC		MV1	M,	OCCH		; SEND ANOTHER CHAR. WITHOUT READING LAST ONE
0E86	23		INX	H			; *COMMAND
		:					
0E87	7E	L6:	MOV	A,	M		; READ STATUS
0E88	E614		ANI	14H			; WAIT UNTIL BOTH TXEMPTY & OVERRUN ERROR
0E8A	FE14		CPI	14H			; ;
0E8C	C2870E		JNZ	L6			; (OR UNTIL TIMEOUT)
		:					
0E8F	2B		DCX	H			; *DATA
0E90	7E		MOV	A,	M		; READ BACK CHAR.
0E91	FECC		CPI	OCCH			; VERIFY IT'S THE 2ND CHAR
0E93	C2DD0A		JNZ	ERR			
		:					
		:	***	8251	PASSED WHOLE TEST	***	
		:					
		:					
0E96	3ABE17		LDA	BAUDS			; LOAD BAUD RATE CONTROL WORD
0E99	32283C		STA	CNLT1			; SO WE CAN DISABLE WRAPAROUND
0E9C	21420A		LXI	H,	CPUTS		; REPEAT SERIES OF TESTS STARTING WITH
0E9F	220017		TSTAD	TSTAD			; THE CPU/8255 TEST
0EA2	C3150A		JMP	GXXIT			; SAY 'GO' AND EXIT

```

;CHECKSUM FOR ROM SEGMENT 10

```

```

;CHECKSUM FOR ROM SEGMENT 10

```

DB 000H

00 6030

O,O,O,O ,O,O,O,O MD

000000000	0EDA
000000000	0EDE
000000000	0EE2
000000000	0EE6
000000000	0EEA
000000000	0EEE
000000000	0EF2
000000000	0EF6
000000000	0EFA

DW 0'0"0" 0'0"0" MD

00000000
00000000
00000000
00000000
00000000
00000000
00000000

DW 0,0,0,0,0,0

0EFA 000000000

0EFA 000000000

8080 MACRO ASSEMBLER, VER 2.4
 ERRORS = 0 PAGE 24

0E0E 00000000
 0F02 00000000

 **
 ** D U M M Y U S E R P R O G R A M **
 **

** THIS PROGRAM IS LOADED INTO RAM TESTED BY THE SELFTEST **

READ FROM CONSOLE & PRINT BACK SAME

1000	ORG	1000H					
1000	CDAS0E		USER:	CALL	INIT		; INITIALIZE SELF-TESTING
1003	FB			EI			; ALLOW INTERRUPTS
1004	214E10		QRY:	LXI	H, PROMPT		; POINT HL TO PROMPT
1007	CD3C10			CALL	H, BUF		; POINT HL TO STORAGE BUFFER
100A	215C10			LXI	H, BUF		
100D	3A2D3C		TREAD:	LDA	PRT51+1		; READ 8251 STATUS
1010	E602			ANI	RXRDY		; ANY INPUT THERE?
1012	CA0D10			JZ	TREAD		; WAIT FOR SOME IF NOT
1015	3A2D3C			LDA	PRT51		; YES, READ THE CHAR
1018	47			MOV	B, A		; STORE IN B
1019	3A2D3C		XECHO:	LDA	PRT51+1		; READ STATUS AGAIN...
101C	E604			ANI	TXMTY		; READY TO OUTPUT?
101E	CA1910			JZ	XECHO		
1021	78			MOV	A, B		; YES, RESTORE CHAR TO A
1022	322C3C			STA	PRT51		; AND ECHO IT
1025	70			MOV	M, B		; STORE IT IN BUFFER
1026	23			INX	H		; ADVANCE POINTER
1027	E67F			ANI	7FH		; STRIP OFF PARITY
1029	FE0D			CFI	ODH		; <CR>?
102B	C20D10			JNZ	TREAD		; NOPE, KEEP READING
102E	360A			MVI	M, OAH		; YES, STORE A <LF> TOO
1030	23			INX	H		
1031	3600			MVI	M, OH		; STORE A ZERO TO MARK END
1033	215B10			LXI	H, LFBUF		; POINT TO <LF> BEFORE BUFFER
1036	CD3C10			CALL	PRINT		; PRINT <LF> THEN THE BUFFER CONTENTS
1039	C30A10			JMP	QRY		

*** PRINT: PRINT STRING POINTED TO BY HL & TERMINATED BY 00 ***

103C	3A2D3C		PRINT:	LDA	PRT51+1		; READ STATUS
103F	E604			ANI	TXMTY		; READY TO OUTPUT?
1041	CA3C10			JZ	PRINT		
1044	7E			MOV	A, M		; YES, READ CHAR FROM TEXT STRING

```

1045 A7 ANA
1046 C8 RZ
1047 322C3C STA
1048 23 INX
1049 C33C10 JMP
104E 454E5445 ; PROMPT: DB
1052 52205445 'ENTER TEXT: '
1056 58543A20 DB 0
105A 00
105B 0A LFBUF: DB OAH
105C 00000000 BUF: DB 0,0,0,0,0
1060 00
; END
PROGRAM ERRORS

```

NO PROGRAM ERRORS

ERRORS = 0 PAGE 26

SYMBOL TABLE

* 01	0007	ADDR1	0BBA	ALHLD	0BBB	AS10T	0E14
A	0000	BAUDS	17BE	BTC	0CED	BTEST	0C99
B	105C	C	0001	CNTL1	3C28	CNTL2	3C30
CNTR	3C38	CPUTS	0A42	D	0002	E	0003
ERR	0A0D	ERX	0006	GOX1T	0A15	H	0004
INIT	0EA5	INPAT	0B89	L	0005	L0	0E19
L1	0E18	L2	0E24	L3	0E68	L4	0E73
L5	0E7D	L6	0E87	LFBUF	1058	LOGIC	0B4B
LSTPA	0000	M	0006	M0PIO	0C4C	N1	0C24
N2	0C35	N2A	0C45	N3	0C49	NPAT1	00CC
N3	0A65	OK1	0A7A	OK2A	0A8C	OK2B	0AAD
OK3	0ABE	OK4	0ACA	OK5	0ADE	PAT16	AA55
PAT11	0033	PAT12	00C1	PATTS	0E9C	P10TS	0C0B
PRINT	103C	PROMP	104E	PRT51	32C2	PRT55	3C3C
PSW	0006	QRY	1004	RAM1	1705	RAM2	1706
RAM3	1707	RAMBE	0F00	RAMSS	0020	RAMTO	0018
RAMTS	0D96	RANDE	1704	RBEQ	1702	READQ	0D80
RNO	0DE0	RN1	0DEC	R0MBE	0A00	R0MSS	0080
R0MTO	000F	R0MTS	0D5D	RSEX1	0A3B	RVAB	0D4D
RVAC	0D3E	RVBC	0D2F	RXRDI	0002	SIOT1	0DF3
SP	0006	SP	0006	STES	0BBE	START	0A00
SIOT2	0E50	TREAD	100D	TSTAD	1700	TNXT	0004
STM	0A00	XECHO	1019	XLOOP	0D75	XR1	0D90
USER	1000						

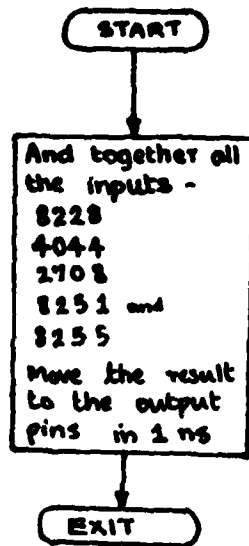
Interested qualified requesters may obtain copies of Appendix B from RADC (COEA), Griffiss AFB NY 13441.

Appendix B

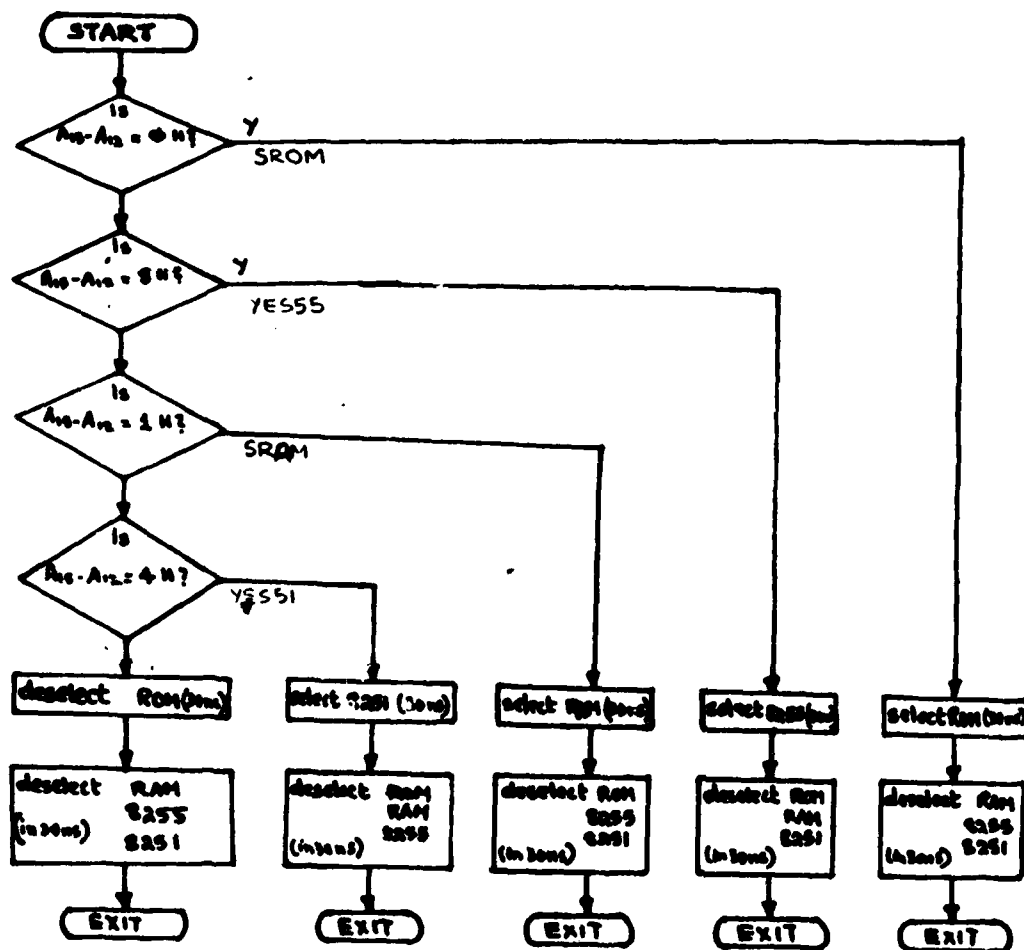
Simulation Model

Flowcharts

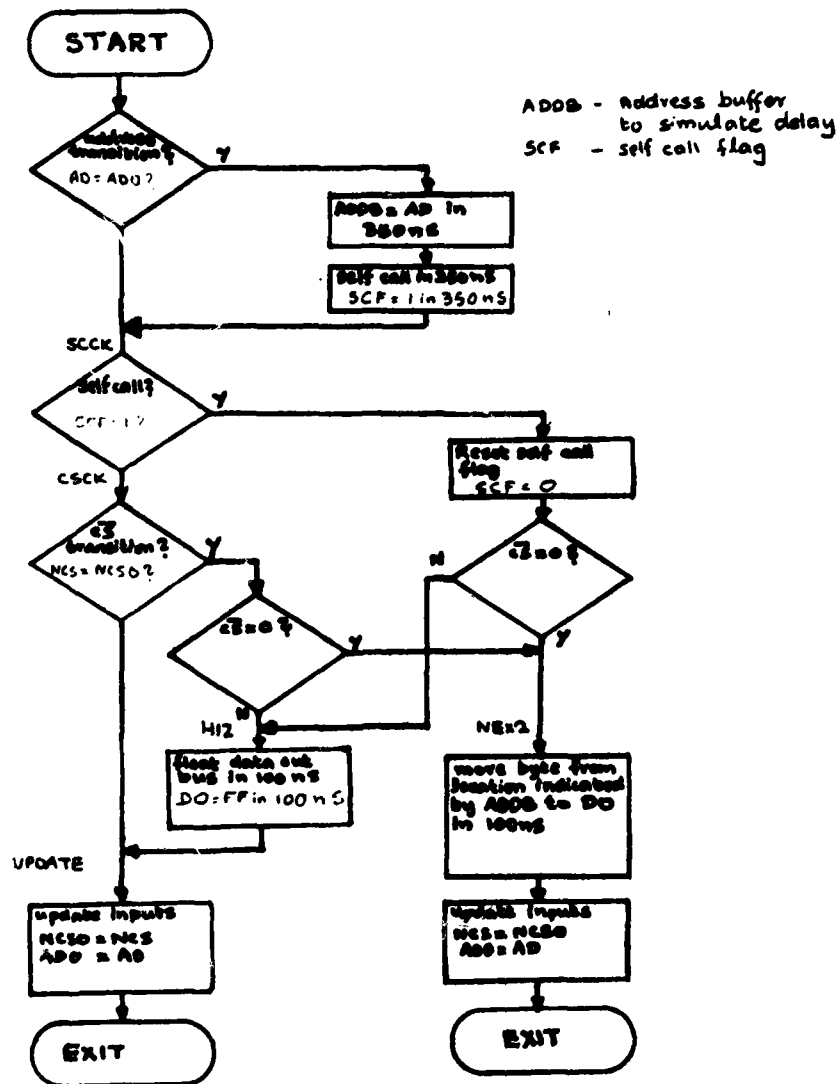
BUS



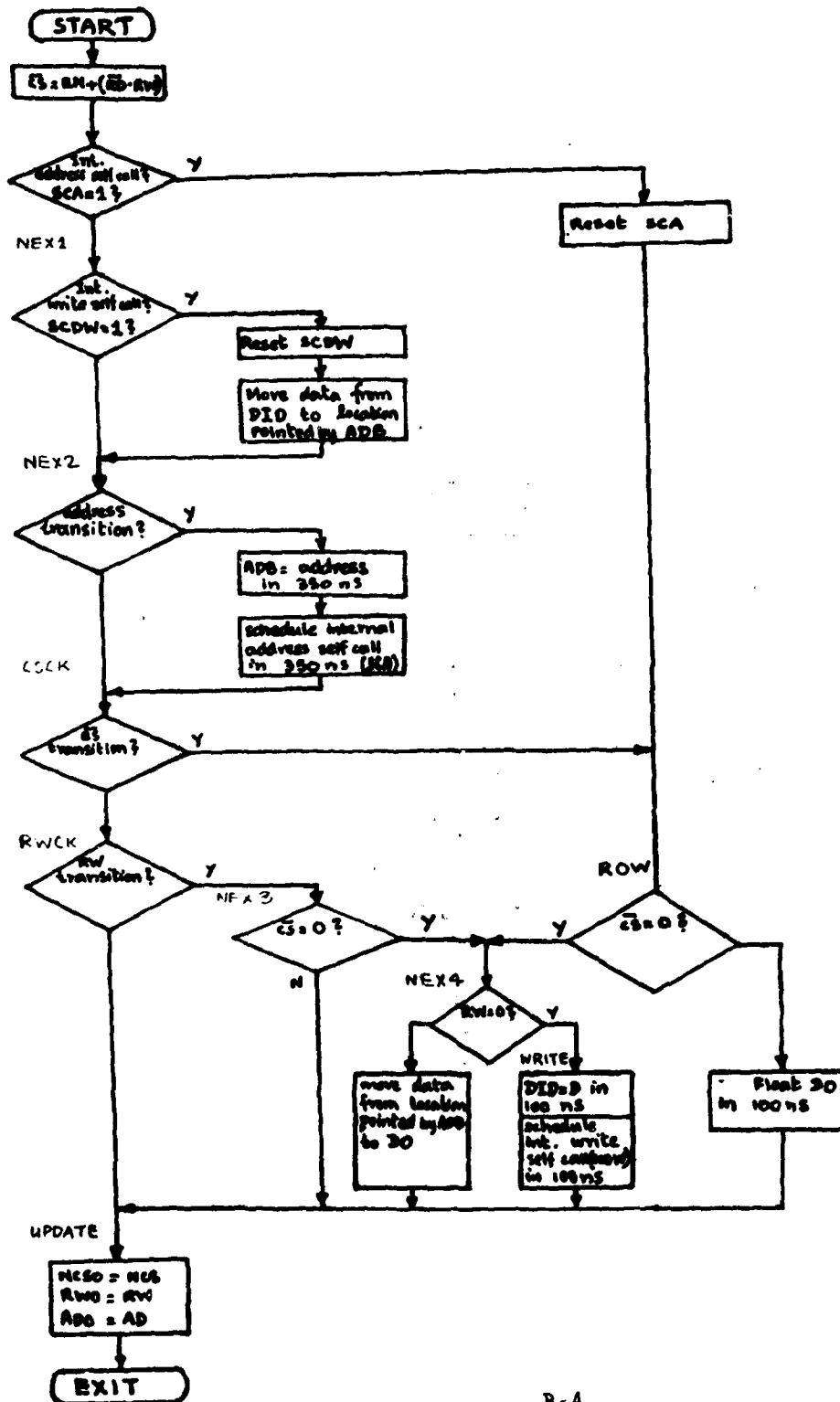
Chip Select Logic



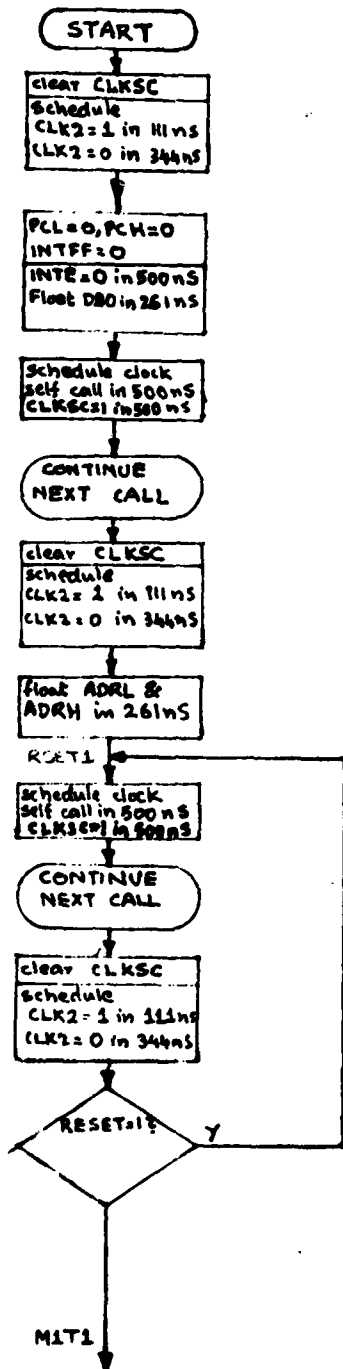
ROM



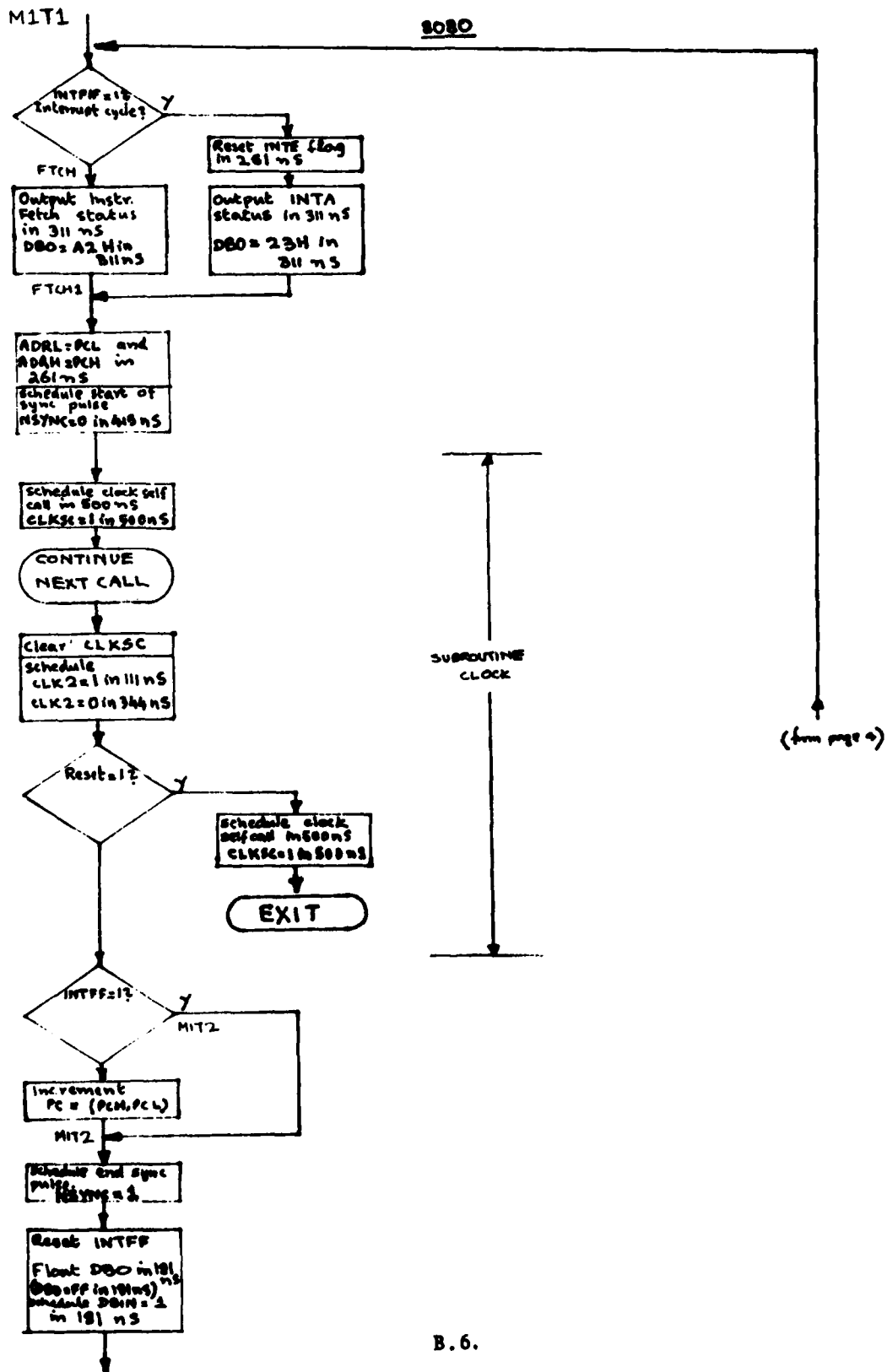
RAM



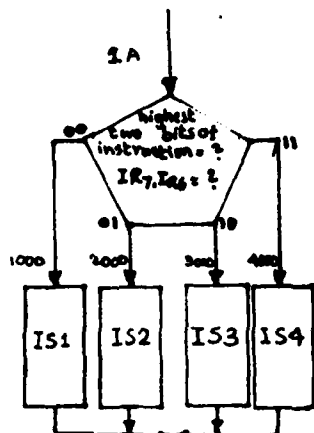
8080



CLKSC - clock self call
 PCL - Program Counter low byte
 PCH - " high byte
 INTFF - Interrupt Flip Flop
 INTE - Interrupt Enable flag
 ADRL - Address low byte
 ADRH - Address high byte



2010



DBSC - DEIN self call

DBSC = 1 in 261 ns

Schedule clock self call in 500 ns
CLKSC = 1 in 500 ns

CONTINUE
NEXT CALL

INTE = 1?

y

INT pin = 1?

y

INTPF = 1

MATX
Reset DBSC

CONTINUE
NEXT CALL

Reset CLKSC

Reset = 0?

y

Schedule clock self call in 500 ns
CLKSC = 1 in 500 ns

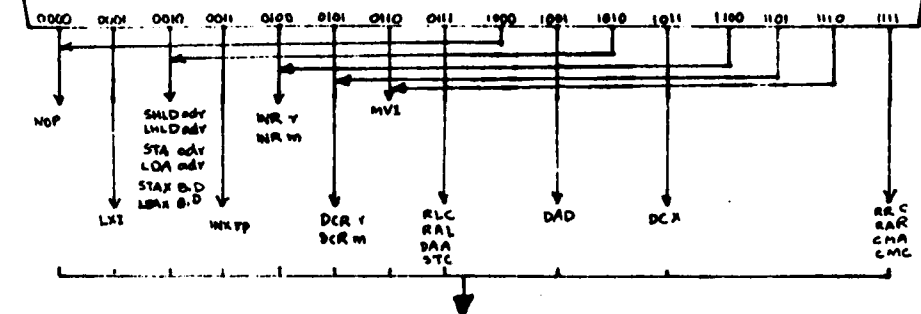
EXIT

M1T1 (beginning of page 2)

1000

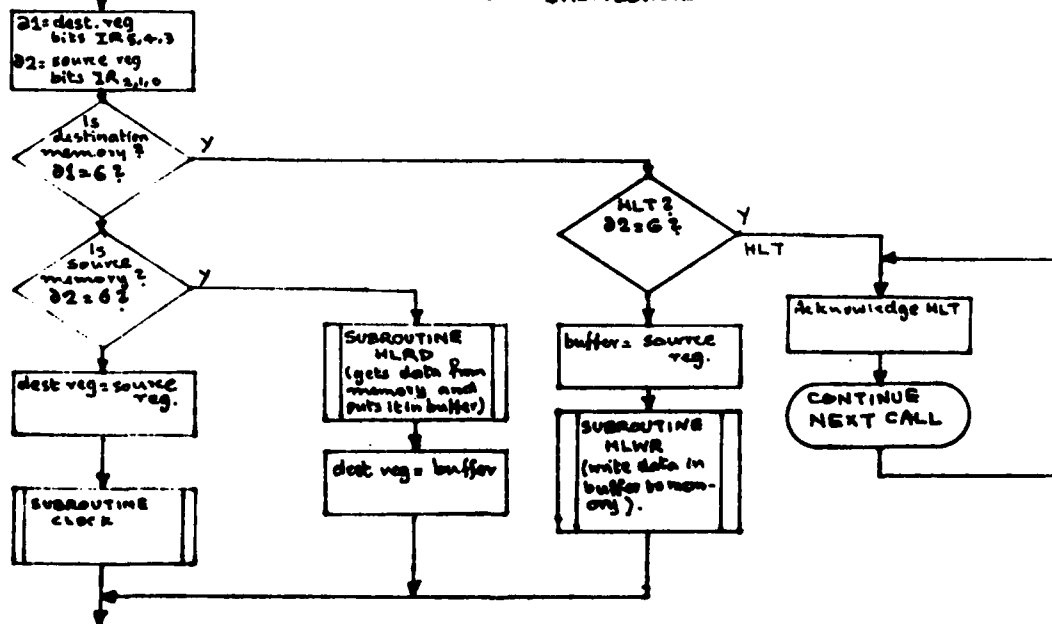
02 - Index register 2
03 - Index register 3

lowest four bits of instruction: ?

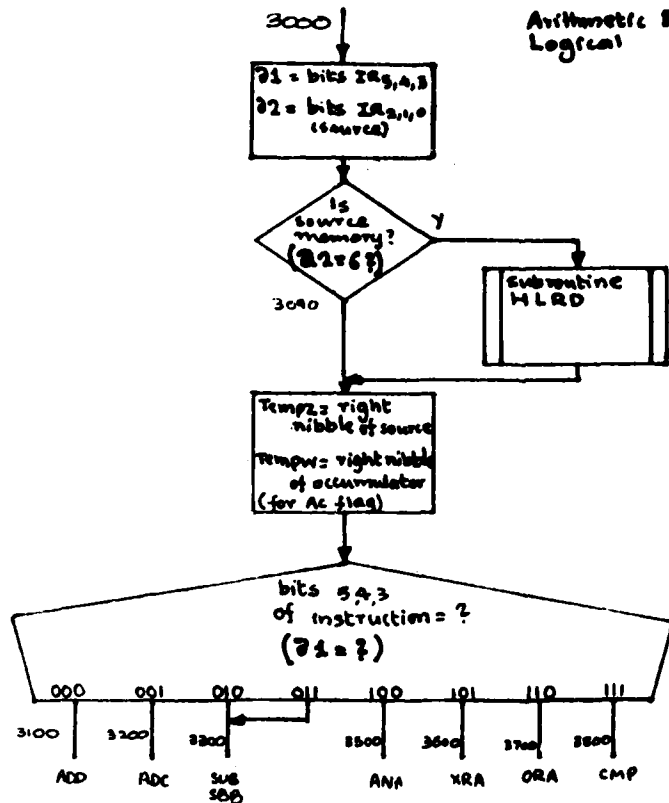


2000

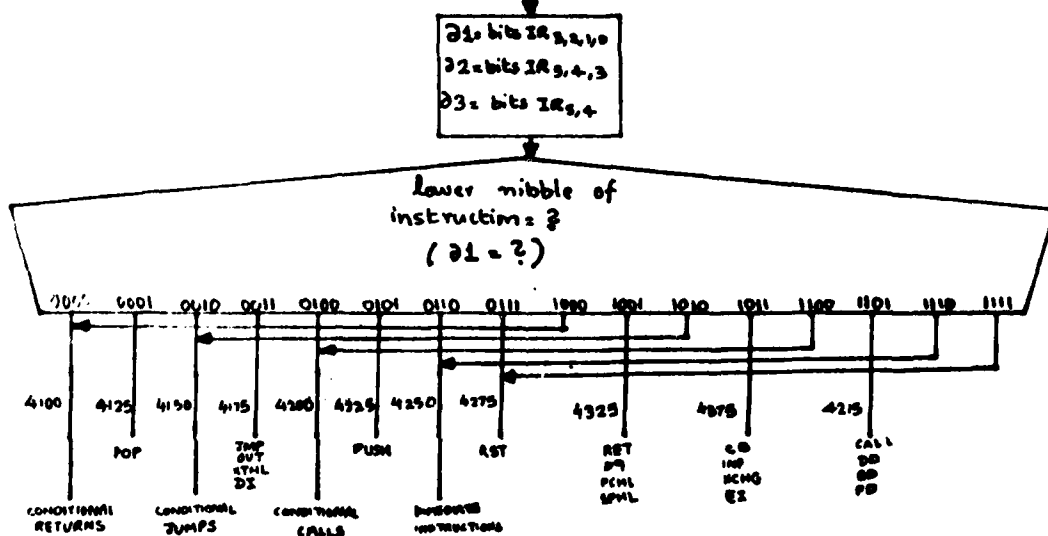
MOV Instructions



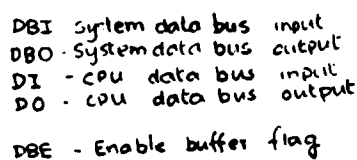
3080



4000



8228



L - status latch

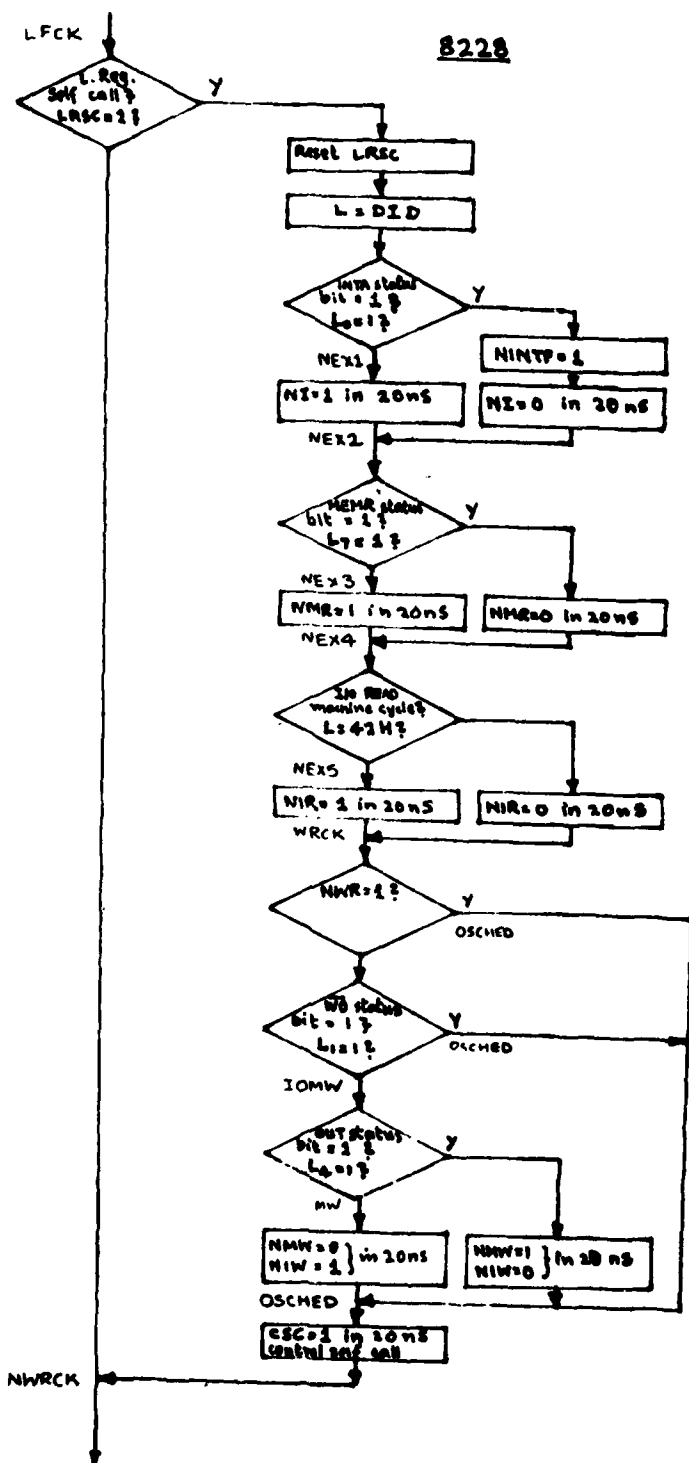
NI <=> NINTA = INTA
 NNR <=> NNMNR = NMNR
 NNN <=> NNMNN = NMNN
 NIR <=> NIOIR = IOIR
 NIW <=> NIOIW = IOIW

NOTES - ~~STATUS STAGE~~

DID - Internal reg. to simulate propagation delay

SP3. Special control Input

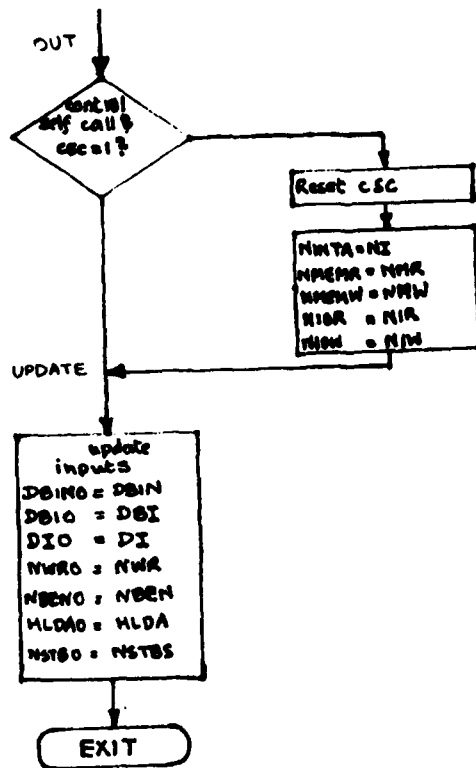
DBSC - DBIN self call flag
LRSC - L Reg. self call flag
CSC - control self call flag



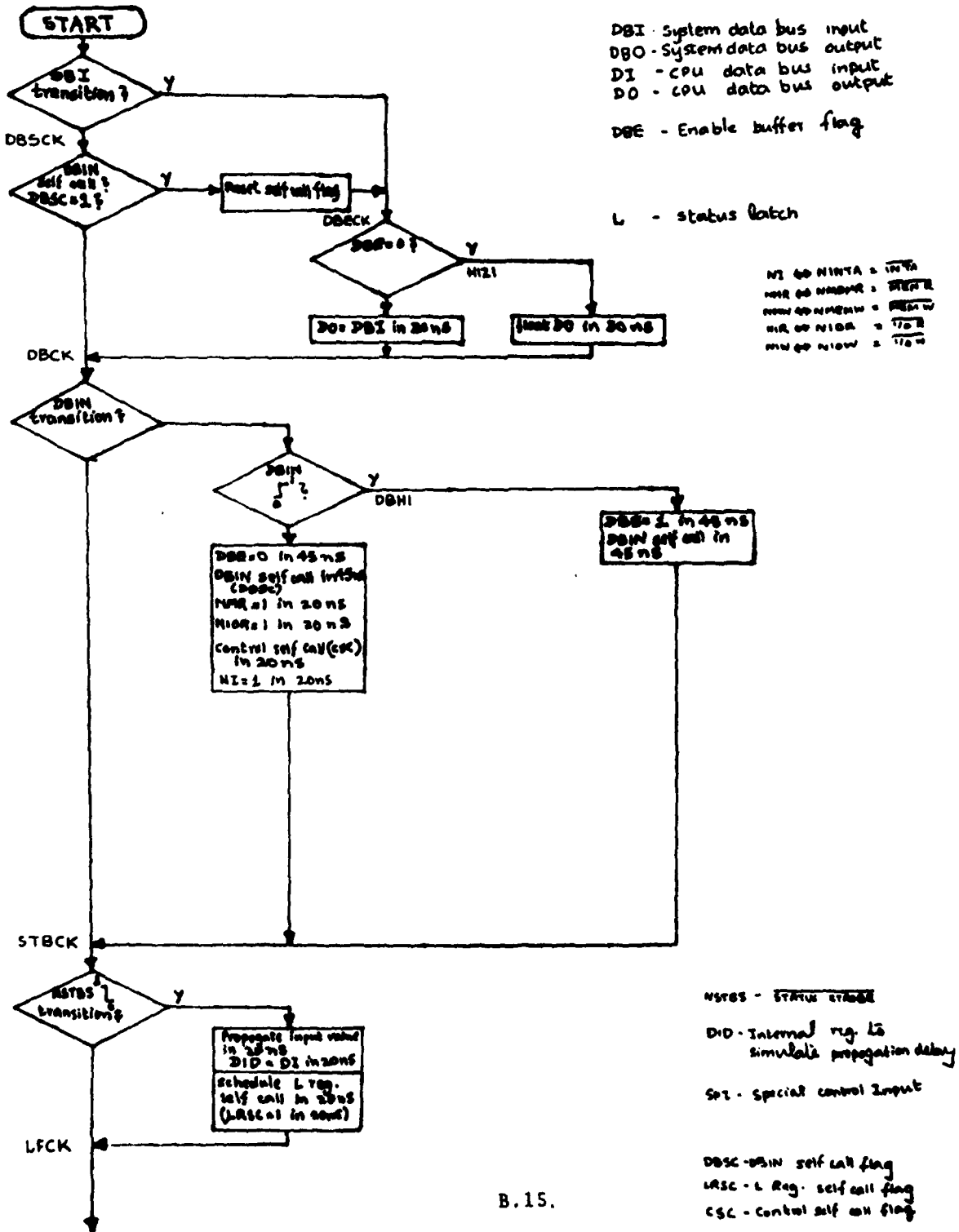
NRBN - ~~BUTEN~~

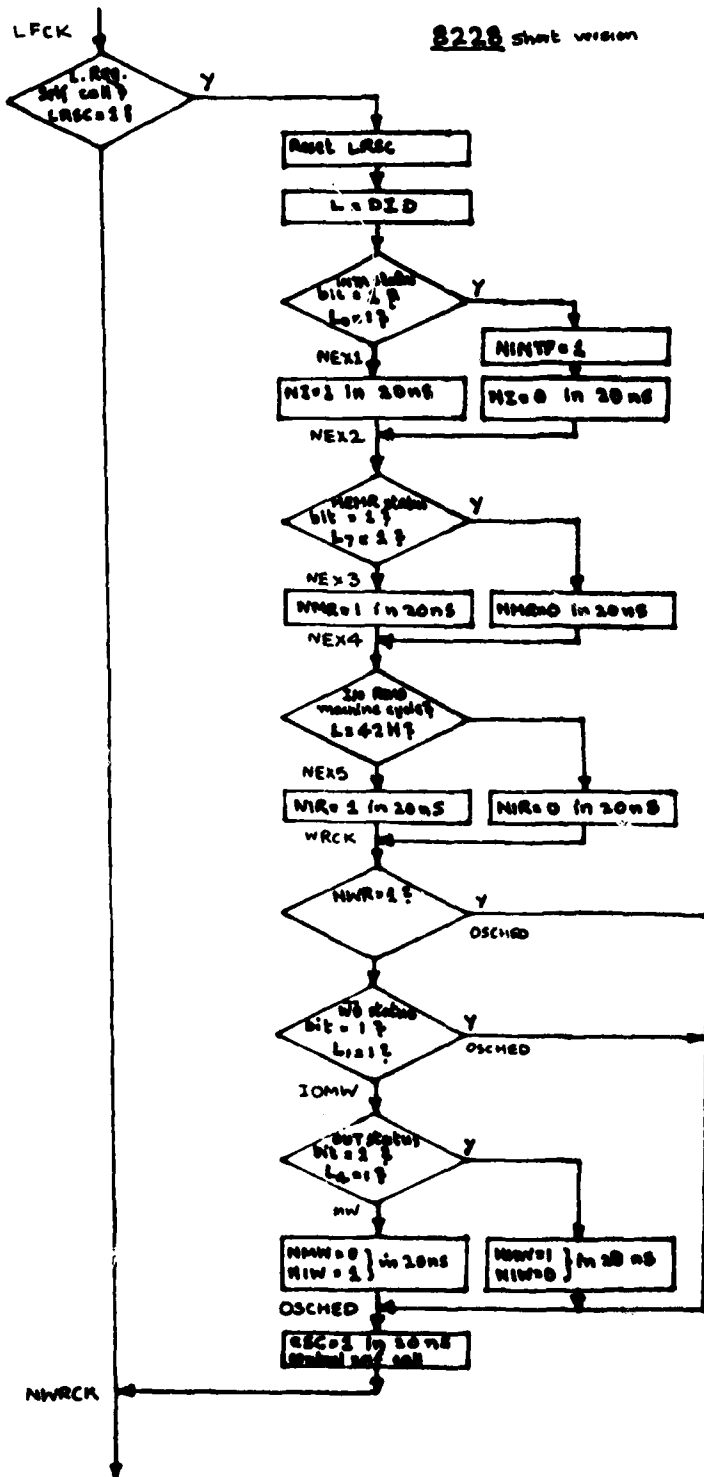


1228

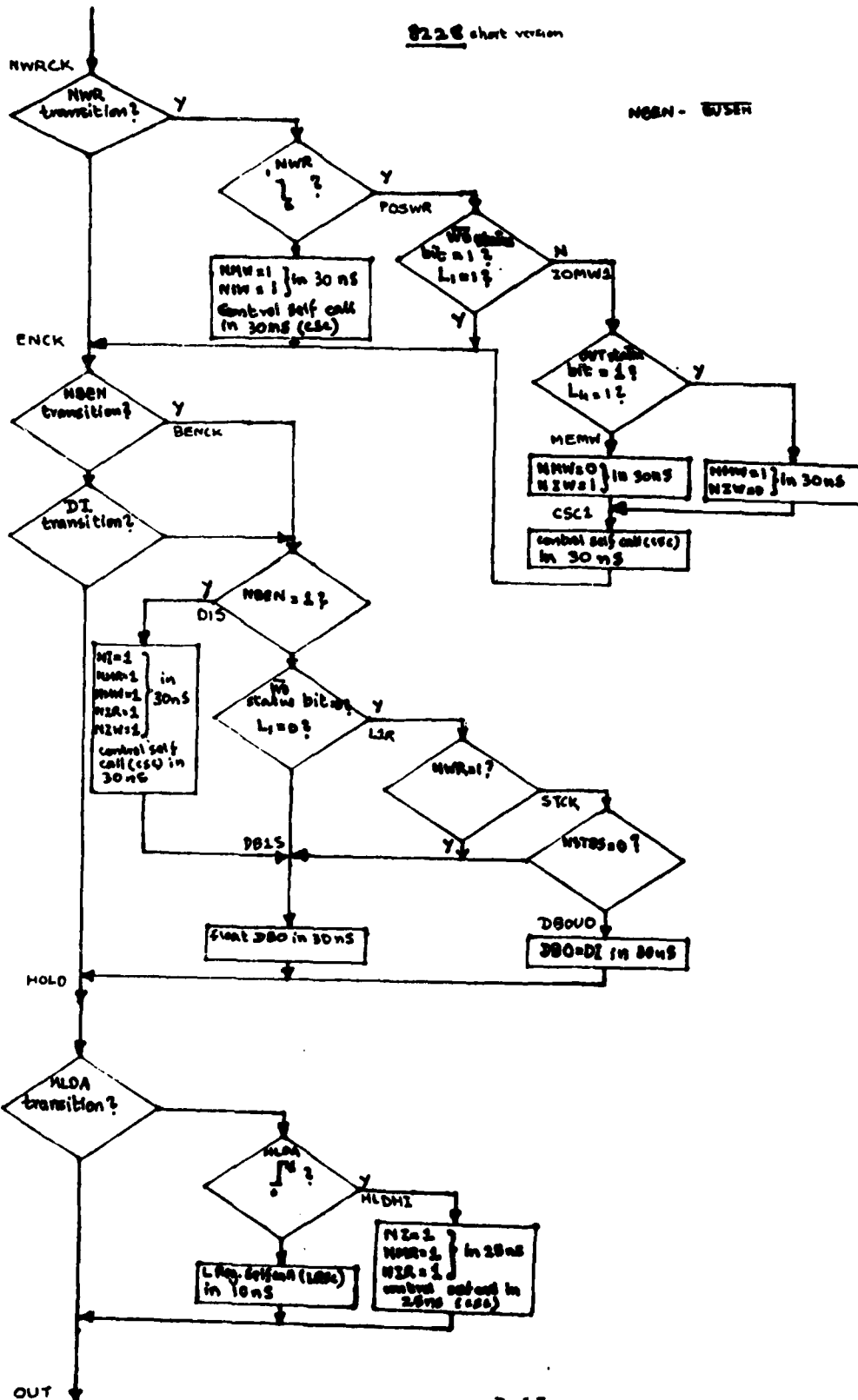


8228 SHORT VERSION

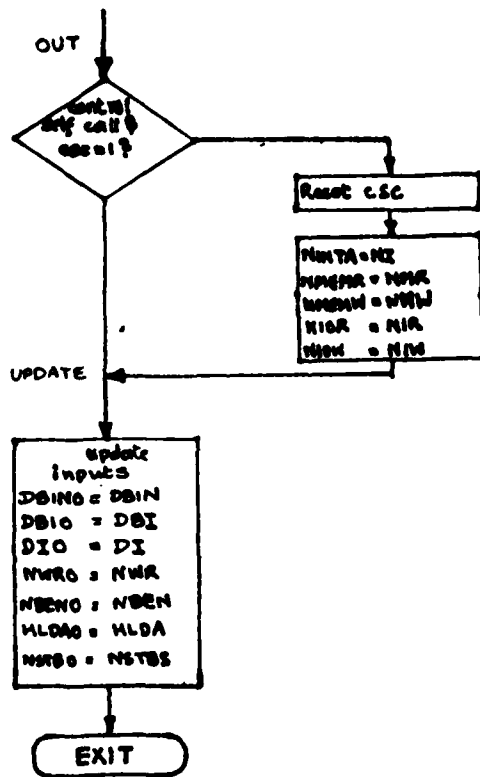


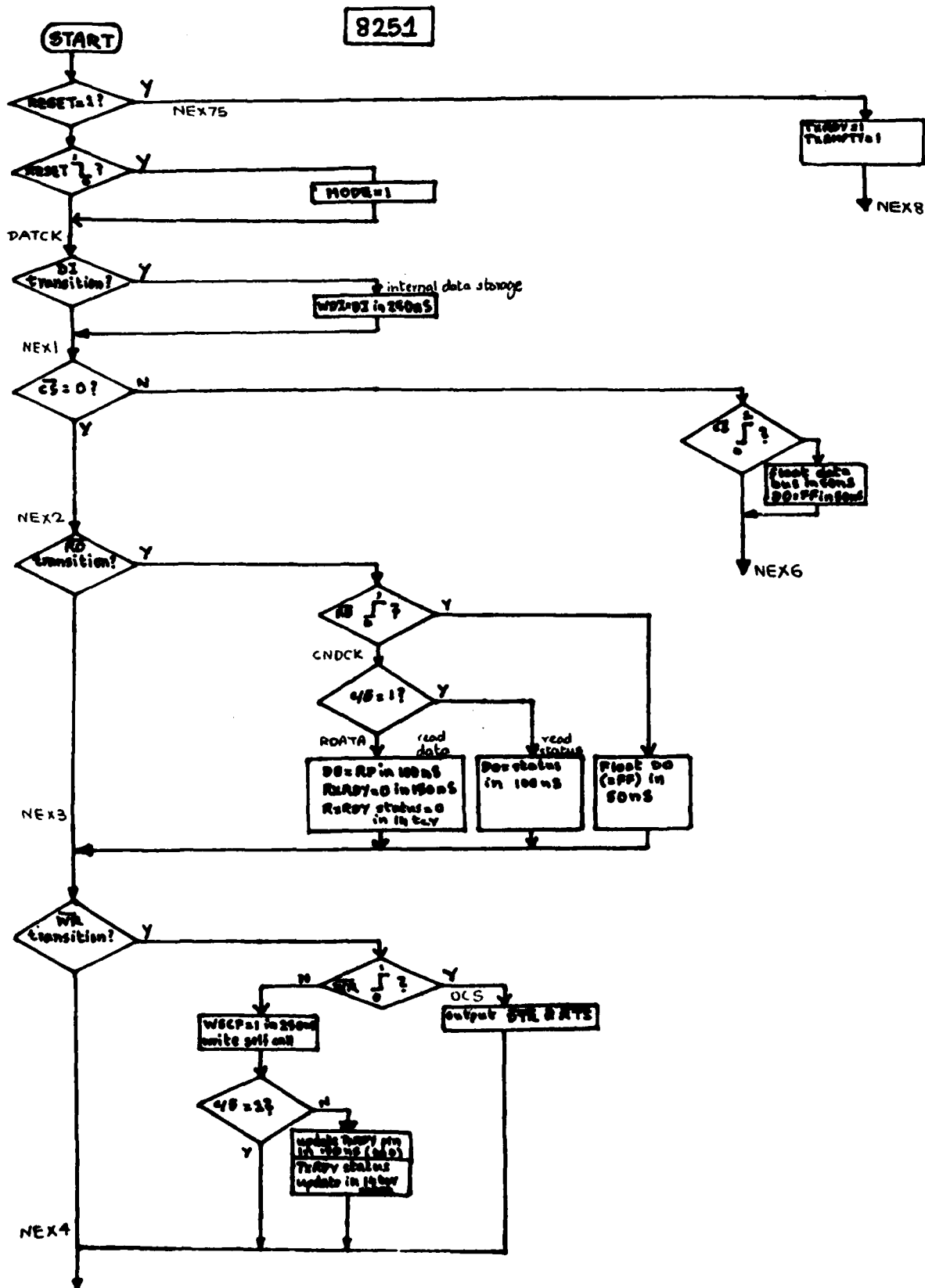


8228 short version

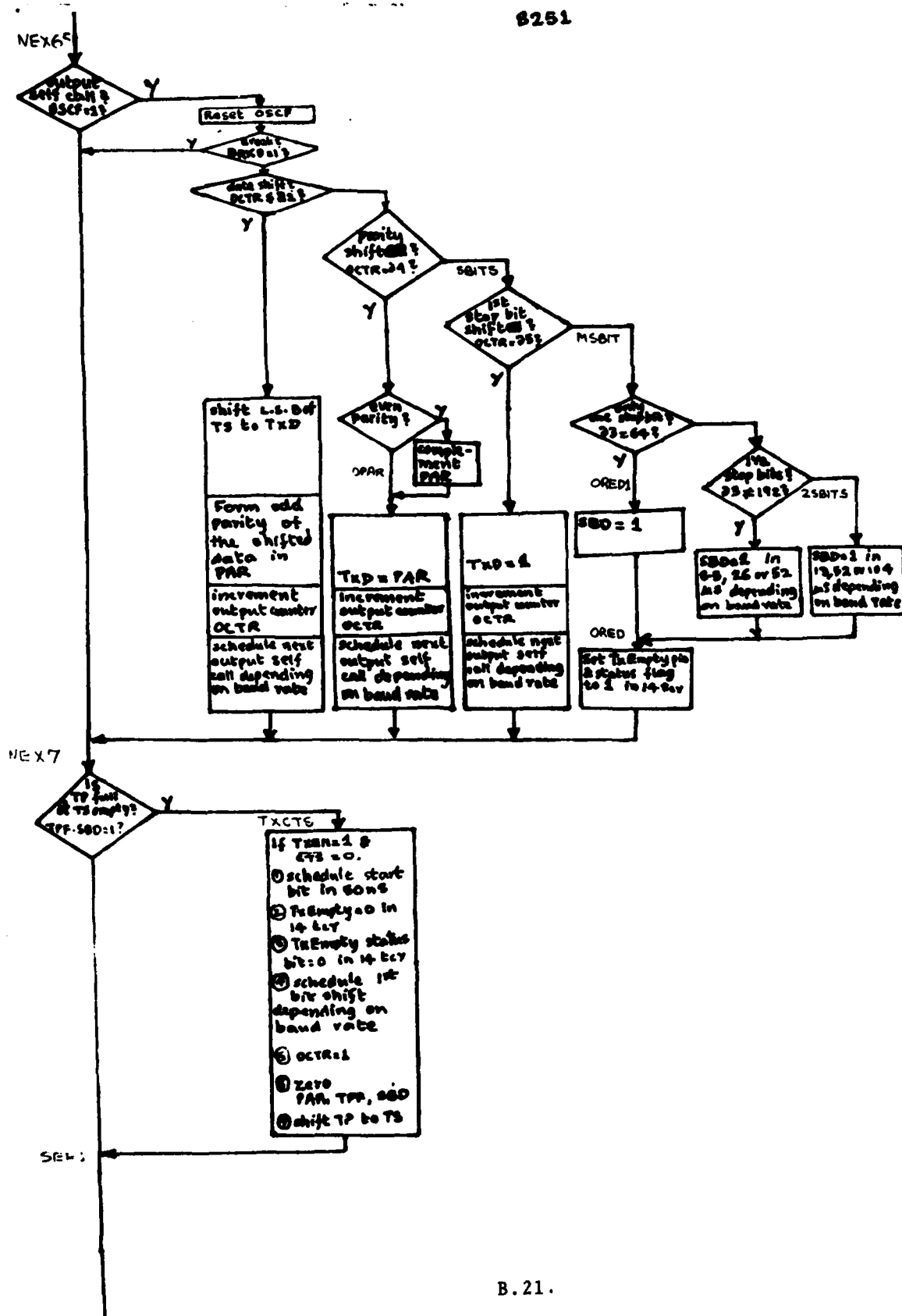


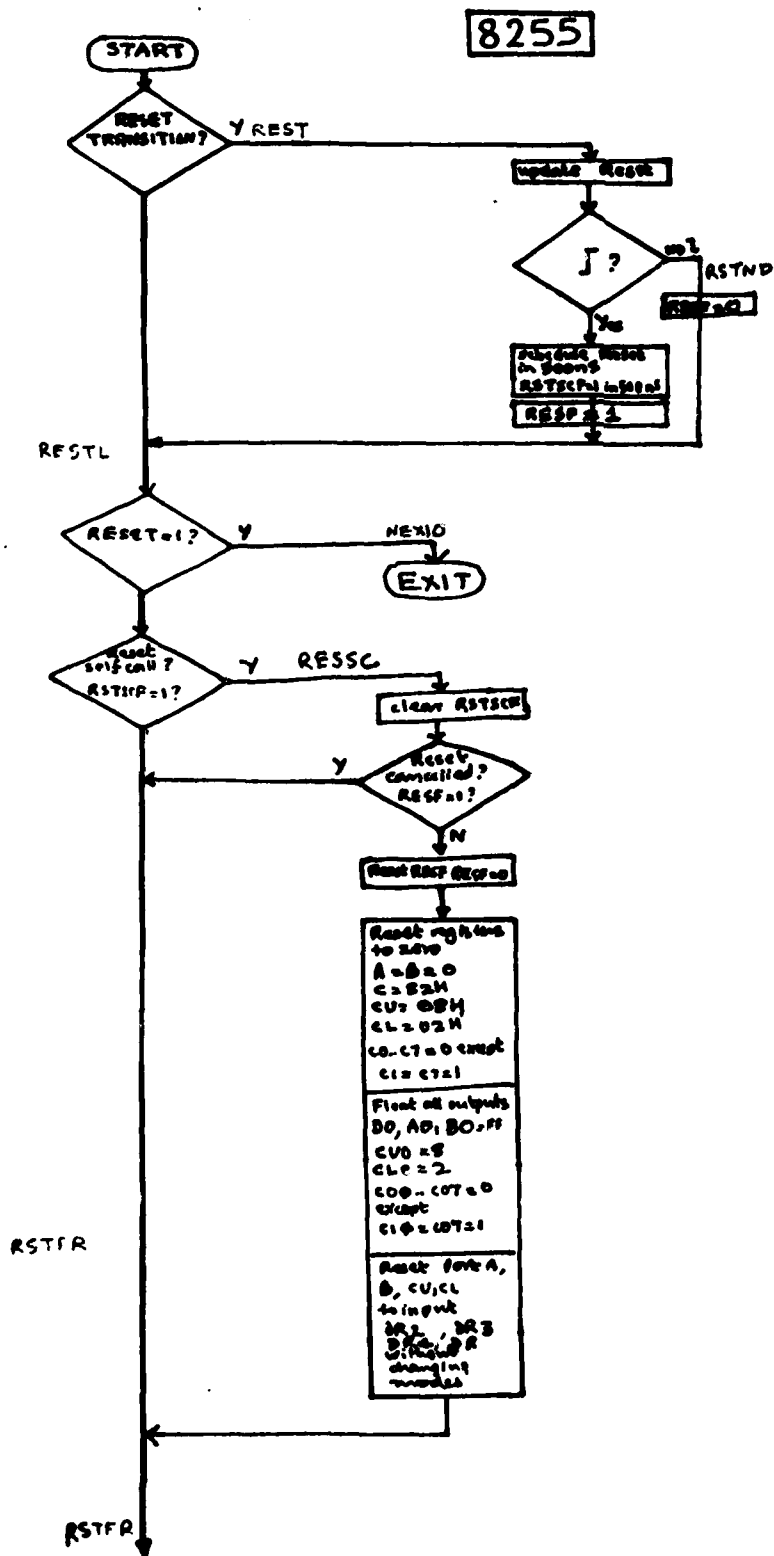
B236 short version

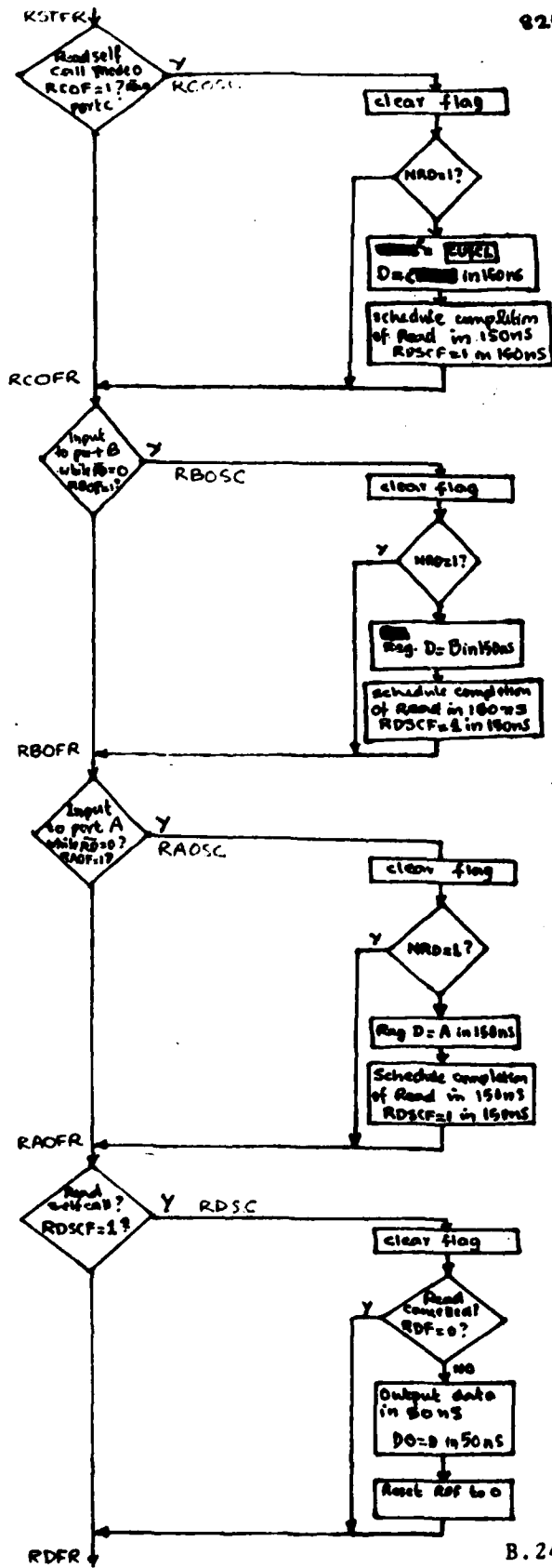


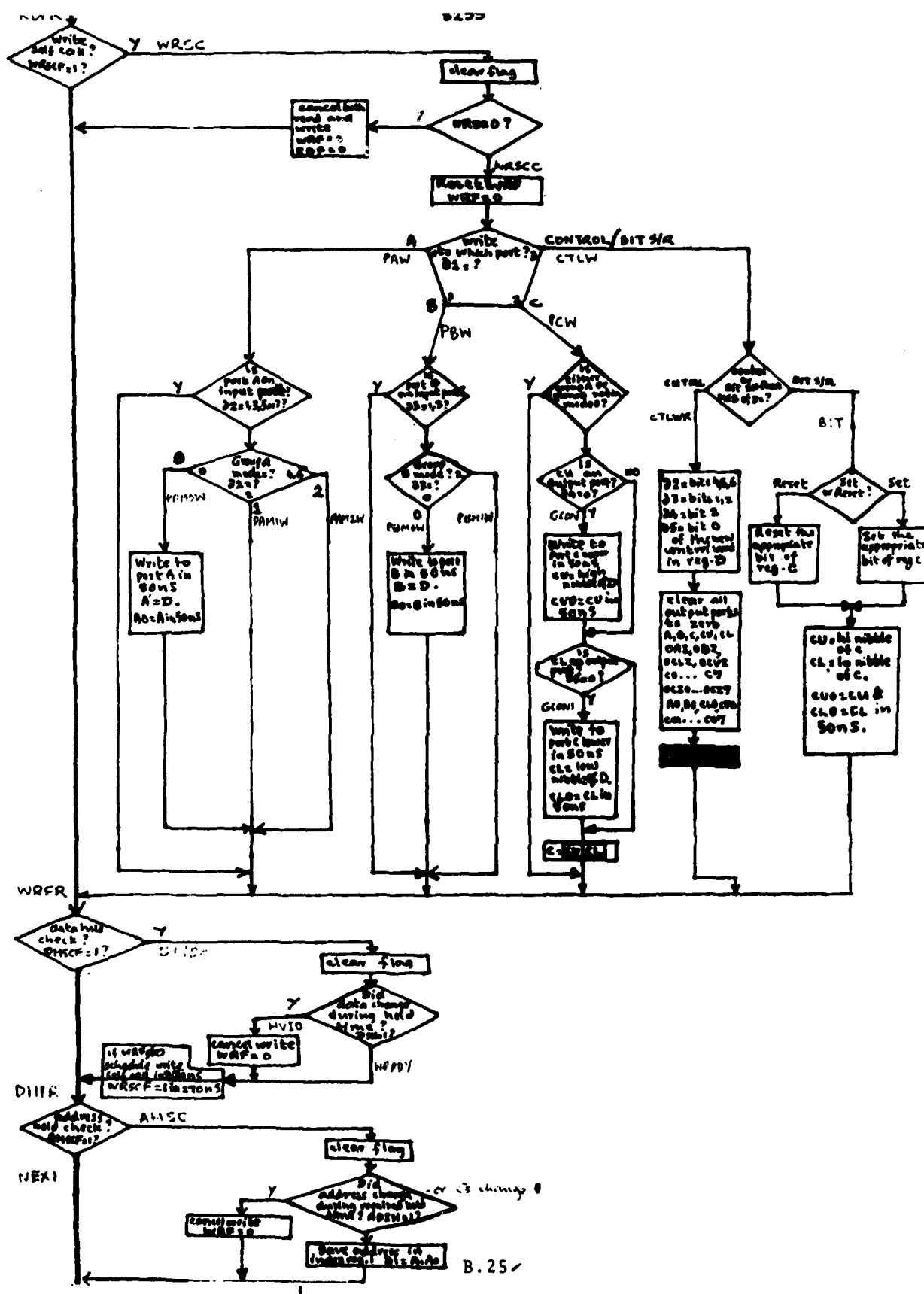


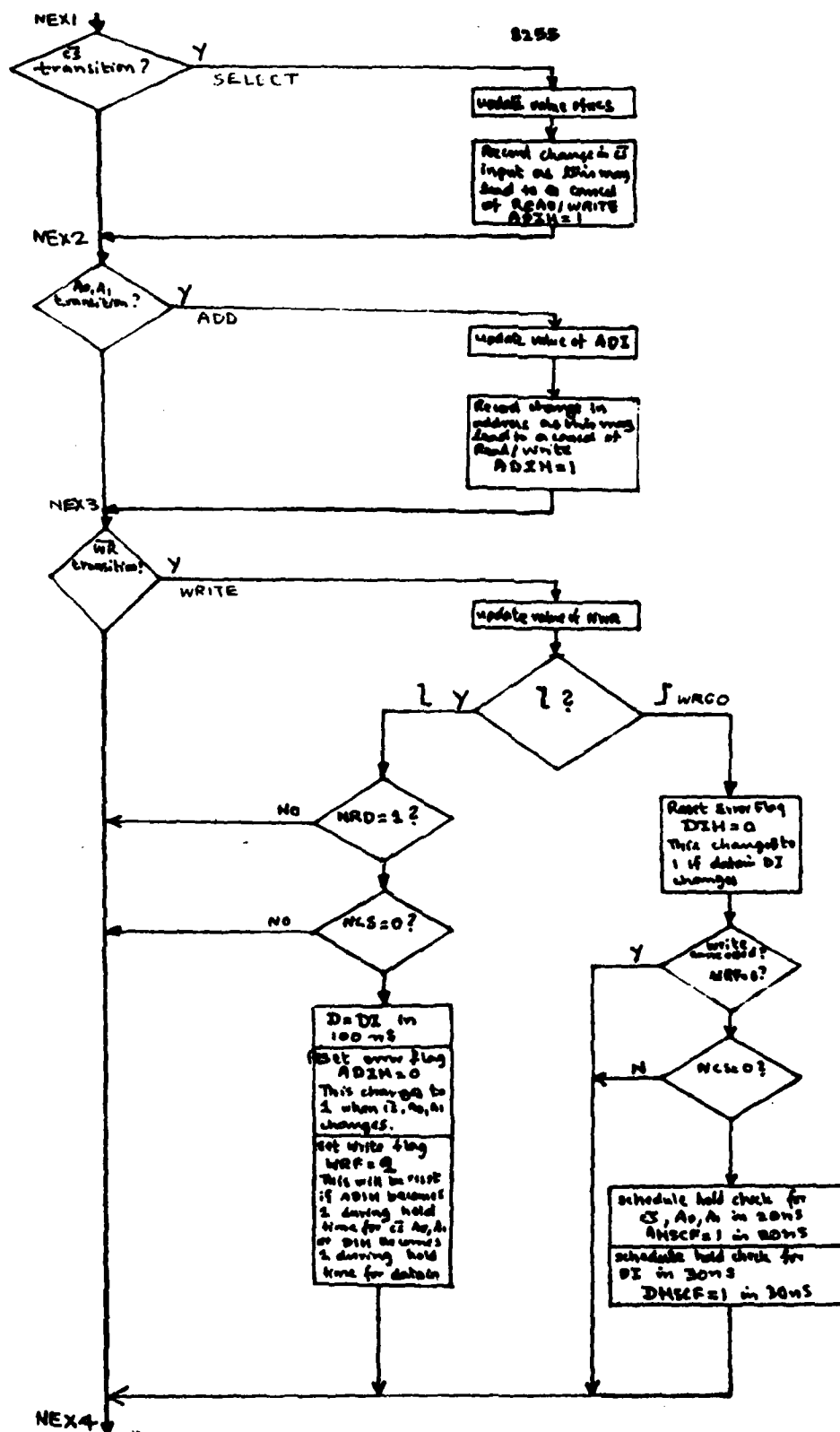


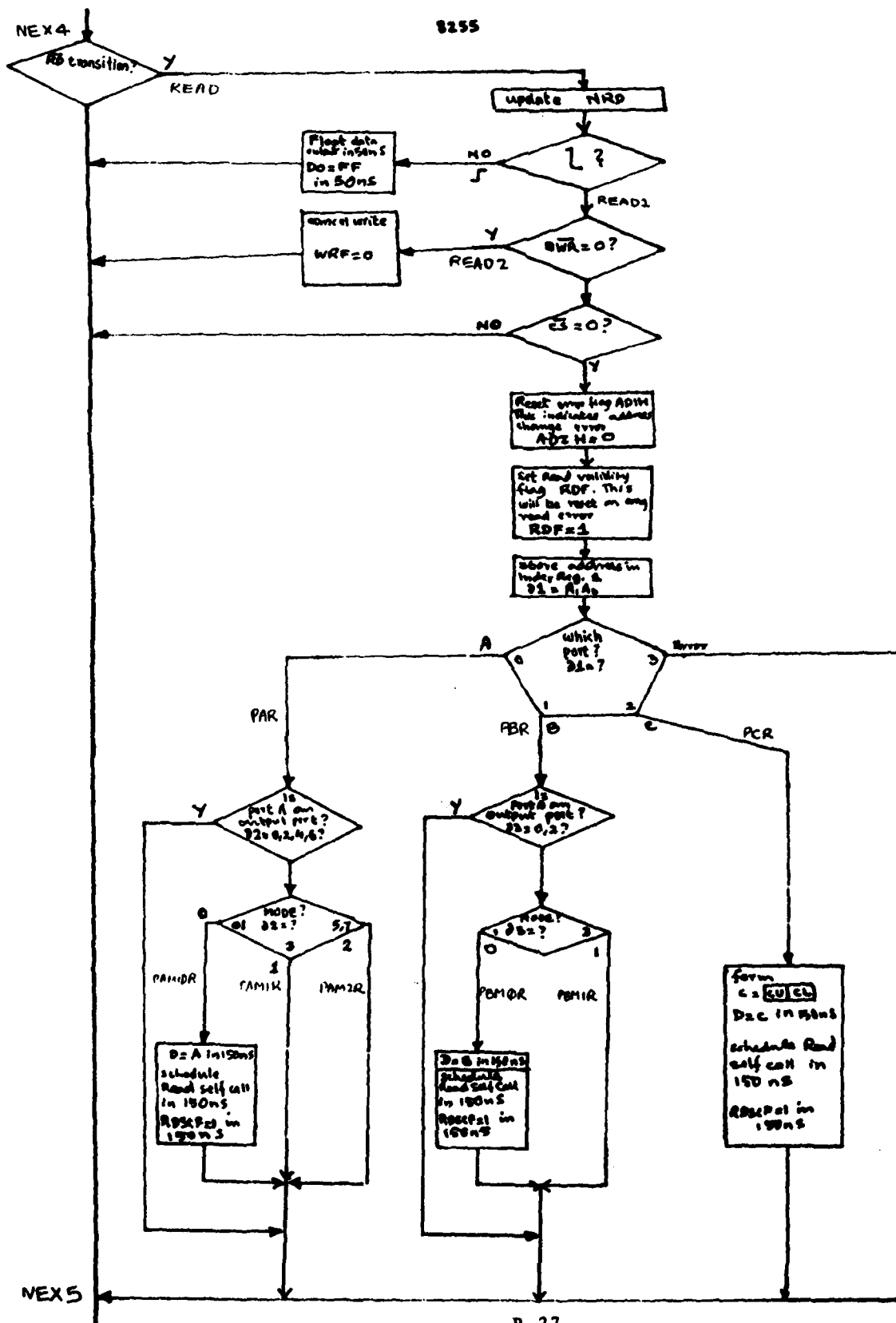


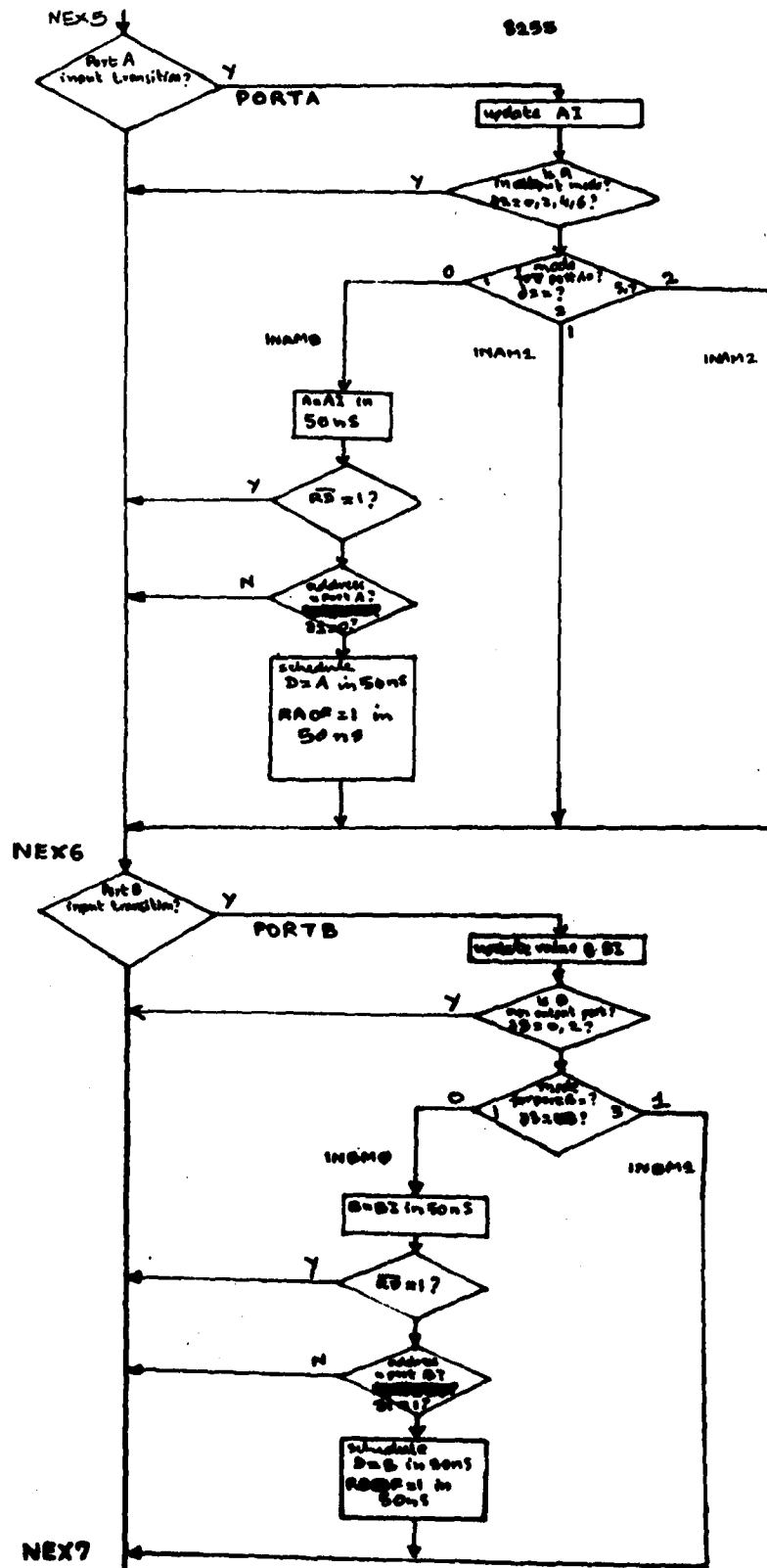


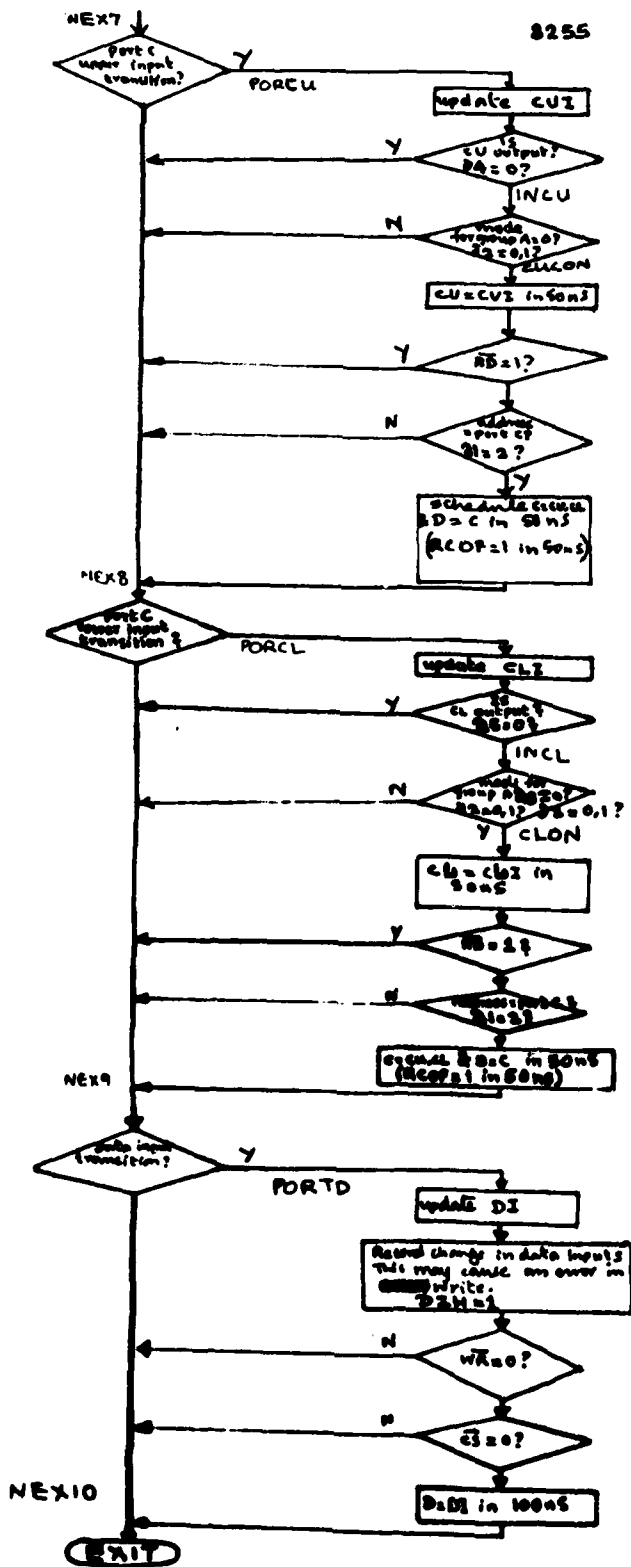












Appendix C

Module Assembly Language Descriptions


```

M1T2: INC PCH
      MOV(RSYNC) #1, NSYNC
      MOV(RDBIN) #255, DB0
      MOV #0, INTFF
      MOV(RDBIN) #1, DBIN
      JSR WAIT1
      MOV DB1, IR
      MOV(FDBIN) #0, DBIN
      JSR CLOCK
      BEQ MONON, 1A
      MON MONON, 1A
      IR(6).2, 1
      TBL101

```

1A:

```

; AT THIS POINT WE SHOULD BE IN THE MACHINE CYCLE PRIOR
; TO M1T1. THIS IS NECESSARY SO I CAN CHECK IF AN
; INTERRUPT HAS OCCURRED.

```

```

      MOV(FINTE) #1, DBSC
      MOV(CP) #1, CLKSC
      MOV #1, EX
      BEQ INTE, M1TX
      BEQ INT, M1TX
      MOV #1, INTFF
      MOV #0, DBSC
      MOV #1, EX
      MOV #0, CLKSC
      BEQ RESET, M1T1
      MOV(CP) #1, CLKSC
      MOV #0, EX

```

M1TX:

C. 2.

```

      TBL1: BYT 1000, 2000, 3000, 4000

```

```

      WAIT1: MOV(CP) #1, CLKSC
      MOV(RRDY) #1, RDYSC
      MOV #1, EX
      MOV #0, RDYSC
      BNE READY, ENDT3
      MOV #1, EX
      MOV #0, CLKSC
      MOV(RISE2) #1, CLK2
      MOV(FALL2) #0, CLK2
      BEQ RESET, WT2
      MOV(CP) #1, CLKSC
      MOV #0, EX
      MOV(RWAIT) #1, WAIT1
      BRU WAIT1
      MOV #1, EX
      ENDT3: MOV #0, CLKSC
      MOV(RISE2) #1, CLK2
      MOV(FALL2) #0, CLK2
      RTS

```

```

; ASSUMPTIONS:
;

```

```

PC => ADDRESS
MEM READ STATUS => DB0

```

```

; IF SO THEN INC PCH
; TRAILLING EDGE OF SYNC PULSE
; TRISTATE DB0 SO BUS MODULE WORKS
; RESET INTERNAL INTERRUPT FLAG
; SET DBIN SO THAT INSTRUCTION CAN BE
; FETCHED, CHECK FOR WAIT STATES
; INPUT THE INSTRUCTION INTO THE REG.
; FALLING EDGE OF DBIN
; CLOCK PULSE
; EXECUTE ROUTINE.
; THIS MON CALL IS TOGGLED ON AND
; OFF BY THE MONON PIN.
; THIS DOES THE EXECUTE PART.

; WAIT STATE (TW) GENERATOR SUBROUTINE
; SETUP FOR NEXT CLOCK PULSE
; DELAY SO THAT READY CAN BE CHECKED
; EXIT AND WAIT FOR READY SIGNAL
; RESET RDYSC AND CHECK READY SIG.
; THIS CHECKS IF WAIT STATES ARE NEEDED
; COMPLETE CLOCK CYCLE
; RESET CLOCK SELF CALL
; RISING EDGE OF CLOCK
; FALLING EDGE OF CLOCK
; CHECK FOR RESET
; SET UP FOR RESET SELF CALL
; EXIT TO RESET
; ENTER WAIT STATE
; GO BACK AND CHECK FOR MORE STATES
; FINISH CLOCK
; RESET THE CLOCK SELF CALL
; CLOCK PULSE
; RETURN TO CALLING PROGRAM

```

```

; PREPARE SELF CALL FOR INT TEST
; SELF CALL FOR CLOCK
; DO EXIT
; IS INT SET?
; IS INT SET?
; YES, THEN SET INTERNAL INT FF
; RESET FLAG
; FINISH CLOCK PULSE
; RESET FLAG
; GO BACK AND FETCH NEW INSTR.
; SET UP FOR RESET CLOCK
; GOTO RESET

```

```

; WAIT STATE (TW) GENERATOR SUBROUTINE
; SETUP FOR NEXT CLOCK PULSE
; DELAY SO THAT READY CAN BE CHECKED
; EXIT AND WAIT FOR READY SIGNAL
; RESET RDYSC AND CHECK READY SIG.
; THIS CHECKS IF WAIT STATES ARE NEEDED
; COMPLETE CLOCK CYCLE
; RESET CLOCK SELF CALL
; RISING EDGE OF CLOCK
; FALLING EDGE OF CLOCK
; CHECK FOR RESET
; SET UP FOR RESET SELF CALL
; EXIT TO RESET
; ENTER WAIT STATE
; GO BACK AND CHECK FOR MORE STATES
; FINISH CLOCK
; RESET THE CLOCK SELF CALL
; CLOCK PULSE
; RETURN TO CALLING PROGRAM

```

```

PCRD: JSR      ; CLOCK
      MOV(RSYDB) #130,DB0
      MOV(RADR) PCL,ADRL
      MOV(RADR) PCH,ADRH
      INC      PCL
      BEQ      C,MEMRD
      INC      PCH

      ; ASSUMPTIONS:
      ; PC IS INCREMENTED
      ; COMPLETE CURRENT CLOCK CYCLE
      ; STATUS BYTE IS PUT OUT ON THE DBUS
      ; PUT LOW BYTE OF PC ON ADDRESS BUS
      ; MOV THE HIGH BYTE OF PC ONTO BUS
      ; INCREMENT PROGRAM COUNTER
      ; INC UPPER 8 BITS IF THERE IS A CARRY
      ; INC UPPER 8 BITS

MEMRD: MOV(FSYNC) #0,MSYNC
      JSR      ; CLOCK PULSE
      MOV(RSYNC) #1,MSYNC
      MOV(RDBIN) #255,DB0
      MOV(RDBIN) #1,DBIN
      JSR      ; TRAILLING EDGE OF SYNC PULSE
      WAIT1   ; TRISTATE DB0 SO THAT BUS MODULE WORKS
      DBI,BUF ; CHECK FOR WAIT STATES
      MOV(FDBIN) #0,DBIN ; INPUT DATA BYTE
      RTS     ; TRAILLING EDGE OF DBIN PULSE

      ; ASSUMPTIONS:
      ; ADDRESS IS ALREADY SET
      ; DB0 HAS SYNC BYTE
      ; BUF HAS BYTE TO OUTPUT
      ; SYNC PULSE
      ; CLOCK PULSE
      ; END SYNC PULSE
      ; OUTPUT DATA BYTE
      ; SET WRITE PULSE
      ; CHECK FOR WAIT STATES
      ; REMOVE WRITE PULSE

      ; CLOCK SUBROUTINE, SET SELF CALL
      ; STOP EXECUTION UNTIL TIME IS REACHED
      ; RESET CLKSC FLAG
      ; OUTPUT CLOCK PULSE
      ; BRANCH IF NOT RESET
      ; SET UP FOR RESET SELF CALL
      ; GOTO RESET
      ; RETURN FOR CLOCK AND ESCAPE FOR HLRD

CLOCK: MOV(CP) #1,CLKSC
      MOV      #1,EX
      MOV      #0,CLKSC
      MOV(RISE2) #1,CLK2
      MOV(FALL2) #0,CLK2
      BEQ      RESET,ESCHL
      MOV(CP) #1,CLKSC
      MOV      #0,EX

      ; THIS PART GETS THE SSS BYTE FROM A REGISTER (A1-L1) OR
      ; FROM MEMORY USING THE HL PAIR. TO IMPLEMENT USE TABLE 2
      ; EXAMPLE: JSR HLRD
      ; MOVE TBL205,A1
      ; CLOCK PULSE BEG. M1T5 OR M2T1
      ; CHECK IF REGISTER OR HL PAIR USED
      ; STATUS BYTE => MEMORY READ
      ; OUTPUT LOW ADDRESS ( L REGISTER )
      ; OUTPUT HIGH ADDRESS ( H REGISTER )
      ; USE MEMRD TO COMPLETE CYCLE

      ; THIS SETS UP THE ADDRESS FOR A MEMORY READ USING THE

```

HLWR: JSR ; HL PAIR.

MOV(RSYDB)
MOV(RADR)
MOV(RADR)
BRU

DECSP: XOR TEMP1,TEMP1
COM TEMP1
ADD TEMP1,SPL
BNE C,5
ADD TEMP1,SPH
RTS

5: C,CARRY
Z,ZERO
N,SIGN
TEMPZ,TEMPW
TEMPW(4),1,6
C6,AC
TEMP1(0),4,7
PARIT07,PAR
TEMP1(4),4,7
PARIT07,TEMP
PAR,TEMP,FLAG2
#0,PAR
#1,PAR

FLAG2: MOV
RTS

FLAG1: MOV
RTS

C ACFLG: BYT
4 PARIT: BYT

#0,#0,#0,#0,#0,#0,#0,#0,#0,#0,#0,#0,#0,#0,
#0,#1,#1,#0,#0,#1,#0,#0,#1,#1,#1,#1,#1,#1

0 => BIT IS 0
* => BIT IS 1
X => BIT IS NOT TESTED YET
1-8 => INDEX REG USED FOR BITS

MSB LSB
00XX XXXX ALL INST. WITH 00 IN 6,7
1111 DECODE LOWER 4 BITS
22 2 DECODE REGISTER
33 DECODE REGISTER PAIR

1000: IDX IR(0),4,1
IDX IR(3),3,2
IDX IR(4),2,3
BRU TBL401

TBL4: BYT 1100,1125,1150,1175
BYT 1200,1225,1250,1275
BYT 1100,1325,1150,1375
BYT 1200,1225,1250,1475

1100: RTS
1125: JSR PCRD
MOV BUF,TBL603

;; STATUS BYTE => MEMORY WRITE
;; OUTPUT ADDRESS

;; LET MEMR COMPLETE CYCLE

;; THIS SUBROUTINE IS USED ANYTIME
;; SOMETHING IS PUSHED ONTO THE STACK
;; -1 IS ADD TO THE STACK POINTER
;; PAIR

;; SET 8080 CARRY FLAG TO NEW VALUE
;; CHANGE ZERO FLAG
;; CHANGE SIGN FLAG
;; ADD LOW ORDER BITS
;; GET AUX. CARRY BIT
;; STORE FLAG
;; CONVERT 4 LOW ORDER BIT TO 07
;; FIND PARITY OF LOWER BITS
;; 4 UPPER BITS TO 07
;; STORE PARITY OF UPPER BITS
;; BRANCH ON EVEN PARITY
;; THEREFORE WE HAVE ODD PARITY
;; DONE
;; EVEN PARITY
;; DONE

00XX XXXX 00H IS NOP, ALL OTHERS
ARE NOT DEFINED
00XX 000* LXI INSTRUCTIONS
33 BYTE IS PUT IN PROP. REG

```

1150: JSR      PCRD
      MOV     BUF,TBL503
      RTS

      BEQ     #2,03,1165
      BEQ     #3,03,1165
      JSR     CLOCK
      MOV(RADR) TBL603,ADRL
      MOV(RADR) TBL503,ADRH
      BEQ     #2,01,1151
      MOV(RSYDB) #130,DB0
      JSR     MEMRD
      MOV     BUF,A1
      RTS

1151: MOV     A1,BUF
      MOV(RSYDB) #0,DB0
      JSR     MEMMR
      RTS

1165: JSR      PCRD
      MOV     BUF,TEMPZ
      JSR     PCRD
      MOV     BUF,TEMPW
      JSR     CLOCK
      MOV(RADR) TEMPZ,ADRL
      MOV(RADR) TEMPW,ADRH
      BNE     #2,01,1170
      MOV(RSYDB) #0,DB0
      BEQ     #2,03,1166
      MOV     A1,BUF
      BRU     1167

1166: MOV     L1,BUF
      JSR     MEMMR
      BEQ     #3,03,1176
      INC     TEMPZ
      BEQ     C,1168
      INC     TEMPW
      JSR     CLOCK
      MOV(RADR) TEMPZ,ADRL
      MOV(RADR) TEMPW,ADRH
      BNE     #2,01,1173
      MOV     H1,BUF
      MOV(RSYDB) #0,DB0
      BRU     MEMMR

1170: MOV(RSYDB) #130,DB0
      BEQ     #2,03,1171
      JSR     MEMRD
      MOV     BUF,A1
      RTS

1171: JSR     MEMRD

```

C. 5.

```

MOV      BUF, L1
BRU      1172
1173:    JSR      MEMRD
MOV      BUF, H1
RTS
1175:    INC      TBL603
BEQ      C, 1176
INC      TBL503
RTS
1176:
1200:    JSR      HLRD
INC      TBL202
1201:    MOV      TBL202, TEMP1
MOV      #1, TEMPZ
AND      #15, TEMP1, TEMPW
JSR      FLAG1
BNE      #6, 02, 1176
BRU      HLRW
1225:    JSR      HLRD
XOR      TEMPZ, TEMPZ
COM      TEMPZ
ADD      TEMPZ, TBL202
MOV      TBL202, TEMP1
JSR      FLAG1
BNE      #6, 02, 1252
BRU      HLRW
1250:    JSR      PCRD
BNE      #6, 02, 1251
BRU      HLRW
1251:    MOV      BUF, TBL202
1252:    RTS
1275:    BRU      TBL703
TBL7:    BYT      1276, 1280, 1285, 1290
1276:    ADD      A1, A1
MOV      C, CARRY
BEQ      C, 1176
OR      #1, A1
RTS
1280:    MOV      CARRY, TEMP
ADD      A1, A1
MOV      C, CARRY
BEQ      TEMP, 1282
OR      #1, A1
RTS
1282:

```

C. 6.

```

; STORE IT IN L1
; GO BACK AND INCREMENT TEMPW-Z PAIR
; READ SECOND BYTE FROM MEMORY
; AND PUT IT IN H REGISTER
;
; O0XX 00** INX RP (INDEX 3)
; CHECK IF CARRY
; INC UPPER BYTE
;
; O0XX X*00 INR DDD
; INC THAT BYTE
; SET UP FOR FLAGS
; *** SET UP AC FLAG STUFF
; ***
; SET CERTAIN FLAGS
; MEMORY?
; YES, STORE BYTE
;
; O0XX X*0* DCR
; ZERO TEMPZ
; SET TO -1
; ADD -1 TO REGISTER
; SET UP FOR FLAGS
; GOTO FLAGS ROUTINE
; WAS IT A MEMORY TYPE INSTR.?
; STORE BYTE INTO MEMORY
;
; O0XX X**0 MVI
; MEMORY?
; YES, PUT BYTE INTO MEMORY
; PUT BYTE INTO REGISTER
;
; O0XX 0*** RLC, RAL, DAA, STC
;
; O000 0*** RLC
;
; SHIFT BIT INTO CARRY FLAG
; IS CARRY SET?
; YES SO SET LSB OF REGISTER
;
; O00* 0*** RAL
; STORE CARRY
; SHIFT
; STORE CARRY
; TEMP SET *****
; MOVE BIT INTO LSB

```

AB080N SOR

```

1285: BNE AC,1286          ; 00*0 0*** DAA
      JDX A1(0),4,7      ; SELECT 4 LOWER BITS
      MOV ACFLG87,TEMP    ; GET AC FLAG
      BEQ TEMP,1287       ; IS IT SET?
      ; LOWER NIBBLE NOW CORRECTED
      ; IS CARRY FLAG SET?
      JDX CARRY,1288      ; SELECT TOP 4 BITS
      ; GET AC FLAG FOR UPPER BITS
      JDX A1(4),4,7      ; FLAG SET?
      BEQ ACFLG87,TEMP    ; CORRECT UPPER BITS
      ; SET UP FOR FLAGS
      JDX #96,A1          ; GO SET FLAGS
      MOV A1,TEMP1
      BRU FLAGS
      ; 00** 0*** STC SET CARRY BIT

1290: MOV #1,CARRY
      RTS

1325: ADD TBL603,L1
      BEQ C,1326
      INC H1
      ; TBL503,H1
      ; C,CARRY
      ; CLOCK
      ; JSR
      ; JSR
      ; JSR
      ; JSR
      ; JSR
      ; JSR
      ; JSR
      ; RTS

1326: MOV TBL503,H1
      JSR C,CARRY
      JSR CLOCK
      JSR CLOCK
      JSR CLOCK
      JSR CLOCK
      JSR CLOCK
      ; 1 NEED TO WASTE 6 CLOCK PULSES

1375: JSR CLOCK
      XOR TEMPZ,TEMPZ
      COM TEMPZ
      ADD TEMPZ,TBL603
      BNE C,1176
      ADD TEMPZ,TBL503
      RTS

1475: BRU TBL803
      ; 00XX **** RRC, RAR, CMA, CMC

TBL8: BYT 1400,1425,1485,1490
      ; 0000 **** RRC

1400: ROR A1
      BRU 1440
      ; 000* **** RAR
      ; STORE CARRY IN 7
      ; SHIFT REST OF BYTE
      ; STORE NEW CARRY

1425: MOV CARRY,07
      SHR A1
      MOV C,CARRY
      ; #6,02,1176
      ; HLWR
      ; BNE
      ; BRU

1440: MOV CARRY,07
      SHR A1
      MOV C,CARRY
      ; #6,02,1176
      ; HLWR
      ; BNE
      ; BRU

1485: COM A1
      ; 00*0 **** CMA

```

C.7.

A8080H SOR

```

3500: AND      TBL202,A1      ; *0*0 0XXX  ANA
3511: MOV      #0,AC          ; CLEAR AC FLAG
      MOV      #0,CARRY      ; CLEAR CARRY
      MOV      #,SIGN        ; SET SIGN FLAG
      MOV      Z,ZERO        ; SET ZERO FLAG
      MOV      A1,TEMP1
      BRU      FLGPA

3600: XOR      TBL202,A1      ; *0*0 *XXX  XRA
      BRU      3511

3700: OR       TBL202,A1      ; *0** 0XXX  ORA
      BRU      3511

3800: COM      TEMPZ          ; ***
      AND      #15,TEMPZ     ; ***
      INC      TEMPZ         ; ***
      SUB      A1,TBL202,TEMP1 ; *0** *XXX  CMP
      BRU      FLAGS

4000: IDX      IR(0),4,1      ; **XX XXXX  INSTRUCTIONS BEGIN WITH **
      IDX      IR(3),3,2      ; DECODE INSTRUCTION TYPE
      IDX      IR(4),2,3      ; CONDITIONS
      BRU      TBL901         ; 22 2
                                   ; 33

TBL9:  BYT      4100,4125,4150,4175
      BYT      4200,4225,4250,4275
      BYT      4100,4325,4150,4375
      BYT      4200,4215,4250,4275

      IR(3),1,4
      CLOCK
      TBL1703

      4101,4102,4103,4104
      04,ZERO,4105
      04,CARRY,4105
      04,PAR,4105
      04,SIGN,4105

      #0,IR
      CLOCK
      SPL,ADRL
      SPH,ADRH
      #134,DB0
      MEMRD

4100: IDX      IR(3),1,4      ; **XX X000  CONDITIONAL RETURNS
      JSR      CLOCK         ; ALL HAVE 5 CYCLES IN M1
      BRU      TBL1703

TBL17: BYT      4101,4102,4103,4104
      BEQ      4101:         ; **00 4000  RZ,RNZ
      RTS
      BEQ      4102:         ; **0* 4000  RC,RNC
      RTS
      BEQ      4103:         ; ***0 4000  RPO,RPE
      RTS
      BEQ      4104:         ; **** 4000  RP,RM
      RTS
      MOV      4105:         ; **00 *00*  RET
      JSR      4106:         ; THIS SECTION IS USED TWICE (IR COUNTS)
      MOV(RADR) SPL,ADRL     ; OUTPUT STACK POINTER
      MOV(RADR) SPH,ADRH     ; STATUS BYTE => STACK READ
      MOV(RSYDB) #134,DB0    ; GET BYTE
      JSR      MEMRD

```



```

4152: BEQ      CARRY,4155      ; JC,JNC
      BRU      4156
4153: BEQ      PAR,4155       ; JPO,JPE
      BRU      4156
4154: BEQ      SIGN,4155      ; JP,JM
4155: RTS
      ; CONDITION NOT MET
4156: MOV      TEMPZ,PCH
      RTS
      ; DO BRANCH BY REPLACING PROGRAM COUNTER
4175: BRU      TBL1403
      ; **XX 00** JMP, OUT, XTHL, DI
TBL14: BYT      4156,4176,4180,4185
4156: JSR      PCRD
      MOV      BUF,TEMPZ
      JSR      PCRD
      MOV      BUF,TEMPW
      BRU      4155
      ; **00 00** JMP UNCOND
      ; SAVE LOW ADDRESS
      ; GET HIGH BYTE
      ; SAVE HIGH BYTE
      ; GOTO TRUE PART OF CONDITIONAL
4176: JSR      PCRD
      JSR      CLOCK
      MOV      BUF,ADRL
      MOV      BUF,ADRH
      MOV      BUF,DB0
      MOV      A1,BUF
      BRU      MEMWR
      ; **0 00** OUT
      ; OUTPUT AND INPUT INST USE BOTH UP&LOW
      ; STATUS WORD => OUTPUT
      ; SET UP FOR WRITE
      ; DO IT
      ; ***0 00** XTHL
      ; OUTPUT STACK POINTER TO READ TOP
4180: JSR      CLOCK
      MOV      SPL,ADRL
      MOV      SPL,ADRH
      MOV      SPL,DB0
      JSR      MEMRD
      MOV      BUF,TEMPZ
      INC      SPL
      BEQ      C,4186
      INC      SPL
      JSR      CLOCK
      MOV      SPL,ADRL
      MOV      SPL,ADRH
      MOV      SPL,DB0
      JSR      MEMRD
      MOV      BUF,TEMPW
      JSR      CLOCK
      MOV      BUF,DB0
      MOV      H1,BUF
      JSR      MEMWR
      JSR      DECSP
      JSR      CLOCK
      MOV      SPL,ADRL

```

C.11.

```

MOV(RADR)    SPH,ADRH
MOV(RSYDB)   #4,D80
MOV          L1,BUF
JSR          MEMMR
MOV          TEMPW,H1
MOV          TEMPZ,L1
RTS

4185: MOV     #0,INTE
RTS

4200: IR(3),1,4
      CLOCK
      PCRD
      BUF,TEMPZ
      PCRD
      BUF,TEMPW
      TBL1603

TBL16: BYT   4201,4202,4203,4204
4201: BEQ    CZ,CNZ
RTS

4202: BEQ    CC,CNC
RTS

4203: BEQ    CPO,CPE
RTS

4204: BEQ    CP,CH
RTS

4205: JSR    CLOCK
      JSR    PCRD
      MOV    BUF,TEMPZ
      JSR    PCRD
      MOV    BUF,TEMPW
      MOV    #0,IR
      MOV    PCH,BUF
      JSR    DECSP
      JSR    SPL,ADRL
      MOV(RADR) SPH,ADRH
      MOV(RSYDB) #4,D80
      JSR    MEMMR
      BNE    IR,4210
      MOV    #1,IR
      MOV    PCL,BUF
      BRU    4208

      TEMPZ,PCL
      TEMPW,PCH
      RTS

      ; STATUS => STACK WRITE
      ; GET READY TO STORE L1 REGISTER
      ; STORE IT
      ; SAVE HL VALUES

      ; **** 00** DI DISABLE INT
      ; **XX X*00 CONDITIONAL CALL
      ;
      ; GET LOW BYTE OF ADDRESS
      ; STORE IT IN TEMP REGISTER
      ; GET HIGH BYTE OF ADDRESS
      ; STORE IN TEMP REGISTER

      ; CZ,CNZ
      ; CC,CNC
      ; CPO,CPE
      ; CP,CH

      ; **00 **0* CALL
      ; GET LOW BYTE
      ; STORE IT
      ; GET HIGH BYTE
      ; STORE IT
      ; DO CALL INSTRUCTION
      ; SET UP TO STORE PC
      ; TWO PASS, DEC STACK POINTER
      ; ADVANCE TO NEXT CLOCK PULSE
      ; OUTPUT STACK POINTER
      ; STATUS => STACK WRITE
      ; STORE IT
      ; CHECK WHICH PASS WE ARE ON
      ; SET UP FOR NEXT PASS
      ; SET UP PC TO BE PUT ON STACK

      ; THIS DOES THE BRANCH
      ;

```

C.12.

A8080M SOR

```

4215: BRU TBL2203 ; **XX **0* CALL AND ILLEGAL INSTR
TBL22: BYT 4205,1100,1100,1100 ; CALL, DD, ED, FD
4225: JSR GLOCK ; M175
MOV #0,IR ; **XX 0*0* PUSH
4226: JSR TBL1003,BUF ; GET REGISTER
; OUTPUT STACK POINTER
; STATUS => STACK WRITE
; IF DONE THEN RETURN
; IS IT PUSH PSW?
; YES, PUT FLAGS INTO TEMP1
; STORE CARRY FLAG
; SHIFT BYTE TWICE
; STORE PARITY
; STORE AUX. CARRY
; DUMMY BIT
; STORE ZERO FLAG
; STORE SIGN FLAG
; SET UP FOR SECOND PASS
; THIS PART DOES IMMEDIATE TYPE 8080
; INSTRUCTIONS. DATA IS READ AND STORED
; IN BUF. TABLE 12 THEN DECODES THE
; PROPER INSTRUCTION.
; ADI
; SET ALL OF THE FLAGS
; ACI
; ***
; ***
; ***
; SUI
TBL12: BYT 4251,4252,4253,4254
4250: JSR PCRD ;
AND #15,BUF,TEMPZ ;
AND #15,A1,TEMPW ;
BRU TBL1202 ;
4251: ADD BUF,A1 ;
MOV A1,TEMP1 ;
BRU FLAGS ;
4252: BEQ CARRY,4251 ;
INC TEMPW ;
INC A1 ;
BRU 4251 ;
4253: COM TEMPZ ;
AND #15,TEMPZ ;
INC TEMPZ ;
4259: SUB A1,BUF,A1 ;
MOV A1,TEMP1 ;

```

C.13.

```

BRU
4254: DEQ CAPTY, A253
      ADD #255, A1
      CMPZ #15, TEMPZ
      BRU 4299
4255: AND BUF, A1
4256: AND #0, CARRY
      AND #0, AC
      Z, ZERO
      N, SIGN
      MOV A1, TEMP1
      BRU FLGPA
4256: XOR BUF, A1
      BRU 4296
4257: OR BUF, A1
      BRU 4296
4258: COM TEMPZ
      AND #15, TEMPZ
      INC TEMPZ
      SUB A1, BUF, TEMP1
      BRU FLAGS
4275: MOV IR, TEMPZ
      AND #56, TEMPZ
      XOR TEMPW, TEMPW
      BRU 4296
4325: BRU TBL1903
TBL19: BYT 4105, 1100, 4326, 4330
4326: MOV L1, PCL
      MOV RT1, PCH
      RTS
4330: MOV L1, SPL
      MOV RT1, SPH
4335: RTS
4375: BRU TBL1503
TBL15: BYT 1100, 4380, 4390, 4395
4380: JSR PCRD
      JSR CLOCK
      MOV(RADR) BUF, ADRL
      MOV(RADR) BUF, ADRH
      MOV(RSYD8) #66, D80
      ; SET FLAGS
      ; SBI
      ; ***
      ; ***
      ; ANI
      ; SET PARITY
      ; XRI
      ; SET FLAGS
      ; ORI
      ; SET FLAGS
      ; ***
      ; ***
      ; ***
      ; CPI
      ; SET FLAGS
      ; ***X *** RST
      ; INSTRUCTION IS THE LOCATION FOR CALL
      ; STRIP OFF WRONG BITS *****
      ; CLEAR UPPER BYTE
      ; GOTO THE CALL ROUTINE
      ; ***X *00*
      ; RET, D9, PCHL, SPHL
      ; PCHL
      ; SPHL
      ; ***X *0** SEE TBL15
      ; CB, INPUT, XCHG, EI
      ; IN
      ; OUTPUT LACATION
      ; STATUS => INPUT

```

; GET BYTE

MEMRD
BUF,A1

;XCHG

L1,E1
E1,L1
L1,E1
H1,D1
D1,H1
H1,D1

;E1
;

#1, INTE

MOV(CF)
RTS

B1,C1,D1,E1
H1,L1,BUF,A1
B1,D1,H1,SPH
C1,E1,L1,SPL
B1,D1,H1,A1
C1,E1,L1,TEMP1

JSR
MOV
RTS

XOR
XOR
XOR
XOR
XOR
RTS

TBL2: BYT
BYT
TBL5: BYT
BYT
TBL6: BYT
BYT
TBL10: BYT
BYT
TBL11: END

4390:

4395:

```

L, TEMP8, DB10, D10 ; 8228 SIMULATOR
REG(8)
REG(3)
REG(1)
PIN
PIN
PIN
PIN
PIN
PIN
EVI
BEQ
BRU
D8SCK: BEQ
MOV
D8ECK: BEQ
MOV(W30)
BRU
HIZI: MOV(W30)
BEQ
D8CK: BEQ
MOV(W45)
MOV(W45)
MOV(W45)
MOV(W45)
MOV(W45)
MOV(W45)
BEQ
MOV
XOR
BEQ
BRU
ICT: MOV
BRU
ICCK: CMP
BEQ
INC
BRU
INTC: MOV(W30)
MOV(W30)
MOV
BRU
DBHI: MOV(W45)
MOV(W45)
XOR
BEQ
BEQ
BEQ
MOV(W45)
MOV
COM
AND
BEQ
L, TEMP8, DB10, D10 ; 8228 SIMULATOR
NSTBO, DB1NO, WPRO, NBENO, NINTF, TEMP1, HILDAO
DI(1,8), DO(9,16), DB1(17,24), DBO(25,32)
NSTBS(33), DB1IN(34), NMR(35), HILDA(36), NBEN(37), SPI(38)
NIO(39), NIOW(40), NMEMW(41), NMEMR(42), NINTA(43), DBE(44)
DI(46,53), NI(54), NMR(55), NMR(56), NIR(57), NIW(58)
EX(150), D8SC(151), CSC(152), LRSC(153)
W30(30), W25(25), W10(10), W20(20), W45(45), WZ(0)
DB1, DB10, D8SCK
D8ECK
D8SC, D8CK
#0, D8SC
DBE, HIZI
DB1, DO
D8CK
#255, DO
DB1IN, DB1NO, STBCK
#1, DB1IN, DBHI
#1, D8SC
#0, DBE
#1, NMR
#1, NIOW
#1, CSC
NINTF, ICCK
#0, NINTF
#205, DB1, TEMP8
TEMP8, ICT
STBCK
#0, INTCT
STBCK
#2, INTCT
Z, NINTC
INTCT
STBCK
#1, CSC
#1, NI
#0, INTCT
STBCK
#1, D8SC
#1, DBE
LOM, L, TEMP8
TEMP8, STBCK
SPI, STBCK
#255, DO
NSTBS, TEMP1
TEMP1
TEMP1, NSTBO, TEMP1
TEMP1, LFCK
; RESET SELF CALL FLAG
; CHECK DBE
; DO = DB1 IN 30NS
; DO = ALL 1'S IN 30NS
; CHECK DBIN FOR CHANGE
; DBIN SELF CALL IN 45NS
; DBE = 0 THEN
; RESET NOT MEN, READ
; RESET NOT IO READ
; SET CONTROL SELF CALL
; CHECK NINTF
; SELF CALL INST
; COUNT = 27
; CALL CONT SIGNAL ROUTINE
; NINTA = 1 IN 30NS
; DBIN SELF CALL IN 45NS
; DBE = 1 THEN
; LO = 17
; CHECK THE SPECIAL CONTROL INPUT
; DO = ALL 1'S IN 45NS
; CHECK FOR STROBE CHANGE

```

```

MOV(W20)      ; SCHEDULE L REG SELF CALL
BEQ(W20)      ; PROPAGATED INPUT VALUE
LFRCK:        ; CHECK L REG SELF CALL FLAG
MOV          ; RESET SELF CALL FLAG
MOV          ; UPDATE L
AND          ; INT ACK?
BEQ          ;
MOV(W20)      ;
BRU          ;

; NEX1:
; NEX2:
MOV(W20)      ;
AND          ;
BEQ          ;
MOV(W20)      ;
BRU          ;

; NEX3:
; NEX4:
MOV(W20)      ;
XOR          ;
BNE          ;
MOV(W20)      ;
BRU          ;

; NEX5:
MOV(W20)      ;
BRU          ;

; WRCK:
BEQ          ;
AND          ;
BEQ          ;
BRU          ;

; IOW:
AND          ;
BEQ          ;
MOV(W20)      ;
BRU          ;

; IW:
MOV(W20)      ;
MOV(W20)      ;
OSCHED:MOV(W20)
; WRCK: BEQ
BEQ          ;
MOV(W45)      ;
MOV(W45)      ;
MOV(W45)      ;
BRU          ;

; POSWR:
AND          ;
BEQ          ;
BEQ          ;
BRU          ;

; IOW1:
AND          ;
BEQ          ;
MOV(W45)      ;
MOV(W45)      ;

```

```

BRU          CSC1
MOV(W45)     #1, N1W
MOV(W45)     #0, N1W
MOV(W45)     #1, CSC
BRU          STCK
;CHANGE IN BUS ENABLE?
;CHANGE IN PROCESSOR DATA?
;OUT?
BNE          NBEN, NBEN0, BENCK
BEQ          DI, DIO, HOLD
BNE          NBEN, DIS
AND          L1M, L, TEMPS
BEQ          TEMP8, L1R
BRU          DB1S
;
L1R:         #1, N1R, DB1S
STCK:        BEQ      NSTBS, DB1S
DEMOD:       BRU      DI, DB0
;DB0=DI IN 30 NS
;
DIS:         #1, N1
MOV(W30)     #1, N1R
MOV(W30)     #1, N1W
MOV(W30)     #1, N1R
MOV(W30)     #1, N1W
MOV(W30)     #1, CSC
MOV(W30)     #255, DB0
MOV(W30)     HLDA, HLD0, OUT
HOLD:        BEQ      HLDA, HLDHI
BNE          #1, LRSC
BRU          OUT
;CONTROL SIGNALS OFF
;
DB1S:        #255, DB0
HOLD:        BEQ      HLDA, HLD0, OUT
BNE          HLDA, HLDHI
MOV(W10)     #1, LRSC
BRU          OUT
;DB0 = ALL 1'S
;REESTABLISH CONTROL OUTPUTS
;
HLDHI:       MOV(W25) #1, N1
MOV(W25)     #1, N1R
MOV(W25)     #1, N1R
MOV(W25)     #1, CSC
OUT:         BEQ      CSC, UPDATE
MOV          #0, CSC
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
MOV(W2)      #1, N1TA
UPDATE:      MOV      DB1M, DB1NO
MOV          DB1, DB10
MOV          DI, DIO
MOV          N1R, N1RO
MOV          NBEN, NBEN0
MOV          HLDA, HLD0
MOV          NSTBS, NSTBO
MOV          #0, EX
L1M:         BYT      #1
L1M:         BYT      #2
L1M:         BYT      #16
L1M:         BYT      #64
L1M:         BYT      #128
;RESET SELF CALL FLAG
;UPDATE CONTROL OUTPUTS

```

B0228 SOR A1 05/05/81 16:16 HP21 F 80 167 RECS VA TECH PRINTED 09/12/81 20:02 PAGE 004

L16M: BYT #66
END

C.19.


```

MOV(W1) #1, RSTSCF
MOV #1, RESF

```

RSTND:

```

BRU RSTL
MOV #0, RESF
BRU RSTL
MOV #0, RSTSCF
BEQ RESF, RSTFR
MOV #0, RESF

```

RESCC:

```

MOV #0, A
MOV #0, B
MOV #0, CU
MOV #2, CL
MOV #130, C
MOV #0, C0
MOV #1, C1
MOV #0, C2
MOV #0, C3
MOV #0, C4
MOV #0, C5
MOV #0, C6
MOV #1, C7
MOV(W0) #255, D0
MOV(W0) #255, A0
MOV(W0) #255, B0
MOV(W0) #6, C10
MOV(W0) #2, C11
MOV(W0) #0, C00
MOV(W0) #1, C01
MOV(W0) #0, C02
MOV(W0) #0, C03
MOV(W0) #0, C04
MOV(W0) #0, C05
MOV(W0) #0, C06
MOV(W0) #1, C07

```

```

EXCEPT C7
FLOAT ALL OUTPUTS.

```

```

RESET C PORT OUTPUTS AS IN OSBORNE?

```

```

RESET PORT A TO INPUT WITHOUT CHANGING MODE.
RESET PORT B TO INPUT WITHOUT CHANGING MODE.
RESET PORT C UPPER TO INPUT.
RESET PORT C LOWER TO INPUT.
RETURN TO MAIN PROGRAM.

```

```

*****
** THIS SECTION OF CODE PROCESSES A CHANGE ON THE CHIP SELECT INPUT PIN. **
**

```

```

*****
**
** SELECT: MOV WCS, ONCS
**           MOV #1, ADIH
**           BRU NEX2
*****

```

```

*****
** THIS SECTION OF CODE PROCESSES A CHANGE ON THE ADDRESS INPUT PINS. **
**

```

```

*****
ADD:  MOV ADI, QADI
      MOV #1, ADIH
      BRU NEX3
*****
;
; * THIS SECTION OF CODE PROCESSES A CHANGE ON THE WRITE INPUT PIN (NWR). *
; *
WRITE:
      MOV NMR, ONMR
      BNE NMR, WRGO
      BEQ NRD, NEX4
      BNE NCS, NEX4

      MOV(W9) DI.D
      MOV #0, ADIH

      MOV #1, WRF

      BRU NEX4
      MOV #0, DIH

      BEQ WRF, NEX4
      BNE NCS, NEX4
      MOV(W5) #1, DHSCEF
      MOV(W8) #1, AHSCEF
      BRU NEX4
*****
;
; * THIS SECTION PROCESSES THE SELF CALLS ASSOCIATED WITH THE WRITE OPERATION. *
; *
*****
;
; * THIS SECTION CHECKS FOR THE REQUIRED HOLD TIME ON THE DATA INPUTS *
; * FOLLOWING THE RISING EDGE OF NWR. *
; *
*****
DHSCEF:
      MOV #0, DHSCEF
      BNE DIH, HVIO

      BRU WRDDY
      MOV #0, WRF
      BEQ WRF, DHSCEF
      MOV(W11) #1, WRSCEF

```

```

BRU DHFR
*****
; SELF CALL IN 270 NS.
; RETURN TO THE MAIN PROGRAM.
*****
; THIS SECTION CHECKS FOR CONSTANT ADDRESS INPUTS AND CHIP SELECT INPUTS
; DURING THE WRITE OPERATION.
*****
; *****
; ANSC: MOV #0, ANSCF
; BNE ADI, AVIO
; MOV ADI, 01
; BRU NEX1
; MOV #0, WRF
; BRU NEX1
; CLEAR THE SELF CALL FLAG.
; IF THE ADDRESS INPUTS CHANGED DURING THE WRITE
; OPERATION, CANCEL THE WRITE OPERATION.
; SAVE THE ADDRESS INPUT CODE IN INDEX REGISTER.
; OTHERWISE, PROCEED.
; CLEAR A FLAG TO INDICATE CANCELLATION OF THE
; WRITE OPERATION.
; RETURN TO THE MAIN PROGRAM.
*****
; *****
; THIS SECTION PROCESSES THE WRITE OPERATION IF ALL REQUIRED CONDITIONS ARE
; MET.
*****
; *****
; WRSC: MOV #0, WRSCF
; BNE NRD, WRSCC
; MOV #0, WRF
; MOV #0, REF
; BRU WRFR
; MOV #0, WRF
; BRU ATAB01
; CLEAR SELF CALL FLAG.
; IF NRD IS 1, THEN CONTINUE.
; OTHERWISE, CANCEL BOTH THE READ AND THE WRITE.
; RETURN TO MAIN PROGRAM.
; TURN OFF WRITE FLAG SINCE THE WRITE WILL NOW GO
; TO COMPLETION.
; INDEXED BRANCH. SEE ATAB DECLARATION BELOW.
*****
; *****
; ATAB: BYT PAV
; BYT PBW
; BYT PCW
; BYT CTLW
; WRITE TO PORT A.
; WRITE TO PORT B.
; WRITE TO PORT C.
; WRITE IN CONTROL MODE OR BIT RESET MODE.
*****
; THIS SECTION SELECTS THE MODE FOR THE PORT A WRITE OPERATION.
; *****
; PAV: BRU WATAB02
; USE INDEX REG TO SELECT THE MODE OF PORT A.
*****
; *****
; WATAB: BYT PAV0W
; BYT WRF0
; BYT PAV1W
; BYT WRF1
; BYT PAV2W
; BYT WRF2
; WRITE TO PORT A IN MODE 0.
; ATTEMPT TO WRITE TO PORT A WHEN IN INPUT MODE.
; WRITE TO PORT A IN MODE 1.
; ATTEMPT TO WRITE TO PORT A WHEN IN INPUT MODE.
; WRITE TO PORT A IN MODE 2.

```



```

BYT PAMOR          ;READ FROM PORT A IN MODE 0.
BYT NEX5          ;READ ERROR. PORT A IS DEFINED AS OUTPUT.
BYT PAM1R         ;READ FROM PORT A IN MODE 1.
BYT NEX5          ;READ ERROR. PORT A IS DEFINED AS OUTPUT.
BYT PAM2R         ;READ FROM PORT A IN MODE 2.
BYT PAM2R
BYT PAM2R

*****
* THIS SECTION PROCESSES A READ FROM PORT A IN MODE 0.
*
*****
PAMOR:  MOV(W10) A,D      ;READ PORT A TO DATA BUS.
        MOV(W10) #1,RDSCF ;COMPLETE THE READ IN 150 NS.
        BRU NEX5         ;RETURN TO MAIN PROGRAM.
*****
* THIS SECTION PROCESSES A READ FROM PORT A IN MODE 1.
*
*****
PAM1R:  BRU NEX6         ;TEMPORARY EXIT.
*****
* THIS SECTION PROCESSES A READ FROM PORT A IN MODE 2.
*
*****
PAM2R:  BRU NEX6         ;TEMPORARY EXIT.
*****
* THIS SECTION PROCESSES A READ FROM PORT B.
*
*****
PBR:    BRU RBTAB03      ;BRANCH TO THE APPROPRIATE ROUTINE.
RBTAB:  BYT NEX5          ;READ ERROR. PORT B IS DEFINED AS OUTPUT.
        BYT PBMOR        ;READ FROM PORT B IN MODE 0.
        BYT NEX5          ;READ ERROR. PORT B IS DEFINED AS OUTPUT.
        BYT PBM1R        ;READ FROM PORT B IN MODE 1.
        MOV(W10) B,D      ;SCHEDULE TRANSFER FROM B TO D IN 150 NS.
        MOV(W10) #1,RDSCF ;SCHEDULE THE READ SELF CALL IN 150 NS.
        BRU NEX5         ;RETURN TO MAIN PROGRAM.
        BRU NEX5         ;TEMPORARY EXIT.
*****
* THIS SECTION PROCESSES A READ FROM PORT C.
*
*****

```

```

;PCR:
IDX CU(0),4,7
MOV #7,C
ROR C
ROR C
ROR C
ROR C
IDX CL(0),4,7
MOV #7,TEMP
OR TEMP,C,C
MOV(W10) C,D
MOV(W10) #1,RDSCF
BRU NEX6

;*****
; THIS SECTION PROCESSES A READ SELF CALL.
;*****
RDSC:
MOV #0,RDSCF
SEQ R0F,RDSCF
MOV(W2) D,D
MOV #0,RDSCF
BRU RDSCF

;*****
; THIS SECTION PROCESSES A CHANGE IN PORT A INPUTS.
;*****
PORTA:
MOV A1,DA1
BRU INATB02
BYT NEX6
INATB:
BYT INAM0
BYT NEX6
BYT INAM1
BYT NEX6
BYT INAM2
BYT NEX6
BYT INAM2

;*****
; INPUT TO PORT A IN MODE 0.
;*****
INAM0:
MOV(W2) A1,A
BNE NRD,NEX6
MOV #1,PBMTM
BNE PBMTM,NEX6
MOV #1,RAOF
BRU NEX6

;*****
; INPUT TO PORT A WHILE RD=0.
;*****
RD0SC:
MOV #0,RAOF
;CLEAR FLAG.

```

```

BNE WRD,RAOFR          ;IF WRD=1, DO NOTHING.
MOV(W10),A,D           ;SCHEDULE TRANSFER FROM A TO D IN 150 NS.
MOV(W10),#1,RDSCF      ;SCHEDULE READ SEFL CALL IN 150 NS.
BRU RAOFR              ;RETURN TO MAIN PROGRAM.
BRU NEX6               ;TEMPORARY EXIT.
INAM1: BRU NEX6         ;TEMPORARY EXIT.
INAM2: BRU NEX6         ;TEMPORARY EXIT.
;*****
; * THIS SECTION PROCESSES A CHANGE ON PORT B NUTS.
; *
;*****
PORTB: MOV B1,OB1       ;UPDATE VALUE OF PORT B.
BRU INB7B3             ;BRANCH TO THE APPROPRIATE ROUTINE.
INB7B: BYT NEX7         ;INPUT ERROR. PORT B IS OUTPUT.
BYT INBMO              ;INPUT TRANSFER TO PORT B IN MODE 0.
BYT NEX7               ;INPUT ERROR. PORT B IS OUTPUT.
BYT INB1               ;INPUT TRANSFER TO PORT B IN MODE 1.
;*****
; * INPUT TRANSFER TO PORT B IN MODE 0.
;*****
INBMO: MOV(W2),B1,B     ;SCHEDULE THE TRANSFER TO PORT B IN 50 NS.
BNE WRD,NEX7           ;IF WRD=1, TERMINATE THE TRANSFER.
MOV(W2),#1,OB1,NEX7    ;IF PORT B IS NOT SELECTED, TERMINATE PROCESS.
BRU NEX7               ;SCHEDULE A TRANSFER FROM B TO D IN 50 NS.
;*****
; * INPUT TRANSFER TO PORT B IN MODE 1.
;*****
INB1: BRU NEX7          ;TEMPORARY EXIT.
;*****
; * TRANSFER TO PORT B IN MODE 0 WHILE WRD=0.
;*****
RBOFC: MOV #0,RBOF      ;CLEAR FLAG.
BNE WRD,RBOFR          ;IF WRD=1, DO NOTHING.
MOV(W10),B,D           ;TRANSFER PORT B TO PORT D IN 150 NS.
BRU RBOFR              ;SCHEDULE THE OUTPUT TO DATA BUS IN 150 NS.
;*****
; * THIS SECTION PROCESSES A CHANGE IN PORT C UPPER INPUTS.
; *
;*****
PORCU: MOV CUI,OCUI     ;UPDATE PORT C UPPER.
BRU INCUT4             ;BRANCH TO THE APPROPRIATE ROUTINE.
INCUT: BYT NEX8         ;INPUT ERROR. PORT C UPPER IS OUTPUT.
BYT INCU              ;INPUT PORT C UPPER.
;*****
; * INPUT FORM PORT C CONTINUES.
;*****
INCUC: BEQ #0,CUCON      ;IF PORT A IS MODE 0, CONTINUE PROCESSING.
BEQ #1,CUCON            ;OTHERWISE, TERMINATE TRANSFER TO PORT C UPPER.
BRU NEX8               ;SCHEDULE A TRANSFER TO PORT C UPPER IN 50 NS.
CUCON: MOV(W2),CUI,CU    ;IF RD=1, TERMINATE THE DATA TRANSFER.
BNE WRD,NEX8

```


PAGE 013

PRINTED 09/12/81 20:02

VA TECH

661 RECS

F 80

A1 07/21/81 17:45 HP21

A8255V6 BOR

END

C. 32.

```

REG(8) RP,RS,TP,TS,MR,CR,STATINP,STATUS,WDI,DIO,TEMP8,TEMPA
REG(4) OCTR,ICTR,TEMP4
REG(1) MODE,ORESET,TEMP1,NDTRO,PAR,RXDO,TPF,ISHE,I PAR
REG(1) IBRK,BRKO,NCTSO,NRDO,NCSO,NMRO,SYNBDQ,SBD
PIN DI(1,8),DO(9,16),RESET(17),CLK(18),NMR(19),NRD(20)
PIN NCS(21),CND(22),NRXC(23),NRXC(24),RXD(25),TXD(26)
PIN RXRDY(27),TXRDY(28),TXEMPTY(29),NDR(30),NRTS(31)
PIN NDSR(32),NCTS(33)
PIN EX(150),WSCF(151),ISSCF(152),ISCF(153),OSCF(154)
EVM W0(0),W1(150),W2(50),W3(100),W5(250),W6(2000),W7(7000)
EVR R0(0)
EVM W8(13020),W9(6510),W10(26040),W11(52080),W12(104160)
BNE RESET,NEXT5
BEQ ORESET,DATCK
MOV #1,MODE
DATCK:BEQ DI,DIO,NEX1
MOV(W5) DI,WDI
NEX1:BEQ NCS,NEX2
BEQ NCS,NCSO,NEX6
BRU NEX6
NEX2:BEQ NRDO,NRDO,NEX3
BEQ NRDO,CNDCK
MOV(W2) #255,DO
BRU NEX3
CNDCK:BEQ CND,RDATA
MOV(W3) STATUS,DO
BRU NEX3
RDATA:MOV(W3) RP,DO
MOV(W1) #0,RXRDY
BIR #1,STATINP
MOV(W7) STATINP,STATUS
NEX3:BEQ NMR,NMRO,NEX4
BNE NMR,OCS
MOV(W5) #1,WSCF
BNE CND,NEX4
BIR #0,STATINP
MOV(W7) STATINP,STATUS
JSR CTXRD
BRU NEX4
OCS: AND R1M,CR,TEMP8
BEQ TEMP8,DTR
MOV(W6) #0,NDR
BRU CRTS
DTR:MOV(W6) #1,NDR
CRTS:AND R5M,CR,TEMP8
BEQ TEMP8,RTSR
MOV(W6) #0,NRTS
BRU NEX4
RTSR:MOV(W6) #1,NRTS
NEX4:BEQ WSCF,NEX5
MOV #0,WSCF
BNE NMR,NEX5
BNE CND,CNDW
MOV WDI,TP

```

```

MOV #1, TPF
BIS #0, STATINP
MOV(W7) STATINP, STATUS
JSR CTXRD
BRU NEX5
CHDM: BNE MODE, MODE1
MOV W01, TP
MOV TP, CR
AND RAN, CR, TEMPS
BEQ TEMPS, INRES
BIR #3, STATINP
BIR #4, STATINP
BIR #5, STATINP
MOV(W7) STATINP, STATUS
INRES: AND RAN, CR, TEMPS
BEQ TEMPS, BRK
MOV #1, MODE
AND RAN, CR, TEMPS
BEQ TEMPS, NOBR
MOV #1, BRKO
MOV(W0) #0, TXD
MOV #0, TPF
BRU TXR11
NOBR: MOV #0, BRKO
MOV(W0) #1, TXD
TXR11: JSR CTXRD
BRU NEX5
MODE1: MOV #0, MODE
MOV W01, MR
MOV W01, TP
LDX MR(0), 2, 1
AND R23R, MR, TEMPS
MOV #0, #7
SHR TEMPS
SHR TEMPS
ADD #5, TEMPS
MOV TEMPS, #2
AND RAN, MR, TEMPS
BEQ TEMPS, PARNO
MOV #2, TEMPS
ADD #1, TEMPS, TEMPS
MOV TEMPS, #4
BRU STOPB
PARNO: MOV #2, #4
MOV #4, TEMPS
STOPB: ADD #1, TEMPS, TEMPS
MOV TEMPS, #5
AND R67M, MR, TEMPS
MOV TEMPS, #3
NEX5: BEQ NDSR, NDSRO, NEX6
BEQ NDSR, RB1T
BIS #7, STATINP
BRU STATUS
RB1T: BIR #7, STATINP
STATUS: MOV(W7) STATINP, STATUS; UPDATE STATUS IN 14TCY
; SET TP FLAG TO 1
; SET TXRDY STATUS BIT.
; UPDATE TXRDY.
; CHECK MODE FLAG
; TP IS EFFECTED TOO
; SET UP CONTROL REG
; CHECK ERROR RESET FLAG
; RESET PARITY, OVERRUN AND FRAMING ERRORS
; CHECK INTERNAL RESET FLAG
; NEXT COMMAND WILL BE A MODE COMMAND
; BREAK CHAR?
; SET UP BREAK
; START OUTPUTTING BREAK.
; KILL PREVIOUSLY WRITTEN DATA
; SET UP NO BREAK
; UPDATE TXRDY.
; RESET MODE FLAG
; STORE MODE WORD
; TP EFFECTED TOO
; INDEX REG 1 GETS BAUD RATE FACTOR
; INDEX REG 2 GETS CHAR LENGTH
; FORM CHAR LENGTH COUNT
; PARITY ENABLED?
; SKIP NEXT INST FOR NO PAR
; IR4 =S CHAR LENGTH +PAR
; IR5 =S CHAR LENGTH +PAR +1 STOP BIT
; INDEX REG 3 GETS # OF STOP BITS
; CHANGE IN NDSR?
; RESET NDSR STATUS
; RESET NDSR STATUS
; UPDATE STATUS IN 14TCY

```

```

NEX6: BEQ NCTS, NCTSO, NEX65
      MOV NCTS, NCTSO
      JSR CTXRD
NEX65: BEQ OSCF, NEX7
      MOV #0, OSCF
      BNE BRKO, NEX7
      MOV #2, TEMP4
      CMP TEMP4, OCTR
      BNE C, PARTY
      SHR TS
      MOV (W0) C, TXD
      XOR PAR, C, PAR
      INC OCTR
      JSR SHFTSKED
      BRU NEX7
      PARTY: MOV #A, TEMP4
      CMP TEMP4, OCTR
      BEQ Z, SBITS
      AND R5M, MR, TEMP8
      BEQ TEMP8, OPAR
      COM PAR
      OPAR: MOV (W0) PAR, TXD
      OCTR: INC OCTR
      JSR SHFTSKED
      BRU NEX7
      SBITS: MOV #5, TEMP4
      CMP TEMP4, OCTR
      BEQ Z, MSBIT
      MOV (W0) #1, TXD
      INC OCTR
      JSR SHFTSKED
      BRU NEX7
      MSBIT: MOV #3, TEMP8
      CMP #64, TEMP8
      BNE Z, ORED1
      CMP #192, TEMP8
      BNE Z, 2SBITS
      BRU TAB101
      100: MOV #0, EX
      101: MOV(W9) #1, SBD
      BRU ORED
      102: MOV(W10) #1, SBD
      BRU ORED
      103: MOV(W11) #1, SBD
      BRU ORED
      2SBITS: BRU TAB201
      200: MOV #0, EX
      201: MOV(W8) #1, SBD
      BRU ORED
      202: MOV(W11) #1, SBD
      BRU ORED
      203: MOV(W12) #1, SBD
      BRU ORED
      ORED1: MOV #1, SBD
      ORED: B'5 #2, STATINP
      ; NCTS TRANSITION?
      ; UPDATE NCTS.
      ; UPDATE TXRDY PIN.
      ; CHECK OUTPUT SELFCALL FLAG
      ; RESET SELFCALL FLAG
      ; IF BREAK IS SET, QUIT OUTPUTTING.
      ; END OF DATA SHIFT?
      ; SHIFT TRANSMITT REG
      ; OUTPUT DATA
      ; FORM ODD PARITY
      ; SCHEDULE SHIFT
      ; END OF PARITY SHIFT?
      ; EVEN OR ODD PAR?
      ; OUTPUT EVEN PARITY OR BREAK CHAR
      ; OUTPUT PARITY
      ; 1ST STOP BIT SHIFTED?
      ;
      ; OUTPUT 1ST STOPBIT OR BREAK CHAR
      ; SCHEDULE SHIFT
      ; ONLY ONE STOP BIT?
      ; 1 AND 1 HALF OR 2 STOP BITS?
      ; ILLEGAL ,EXIT
      ; TXD LOW IN 6.51USEC
      ; TXD LOW IN 26.04 USEC
      ; TXD LOW IN 52.08 USEC
      ; ILLEGAL, EXIT
      ; TXD LOW IN 13.02 USEC
      ; TXD LOW IN 52.08 USEC
      ; TXD LOW IN 104.16 USEC
      ; SET TXEMPTY STATUS IN 14 TCY

```

AD-A118 826

VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG D--ETC F/G 9/2
MICROPROCESSOR SELF-TEST: SOFTWARE SELF-TEST FOR AN 8080-BASED --ETC(11)
JUN 82 J R ARMSTRONG, F G GRAY F30602-80-C-0200

UNCLASSIFIED

RADC-TR-82-80

NL

3 3

12500

END
DATE
FEB 82
DTIC

```

MOV(W7) STATINP,STATUS
MOV(W7) #1, TXEMPTY
NEXT:AND TPF, SBD, TEMP1
BEQ TEMP1, SER1
JSR TXCTS
SER1:BNE LSHF, FLCK1
COM RXD, TEMP1
AND TEMP1, RXD, TEMP1
BEQ TEMP1, FLCK
BRU TAB301
300: MOV #0, EX
301: MOV(W9) #1, ISSCF
302: MOV(W10) #1, ISSCF
303: MOV(W11) #1, ISSCF
FLCK:BEQ ISSCF, FLCK1
BNE RXD, FLCK1
MOV #1, LSHF
MOV #1, ICTR
MOV #1, IPAR
MOV #0, IBRK
JSR LSKED
BRU FLCK1
FLCK1:BEQ LSCF, NEX8
MOV #0, LSCF
MOV #2, TEMP4
CMP TEMP4, ICTR
BNE C, IPARC
MOV RXD, #7
SHR RS
XOR RXD, IPAR, IPAR
INC ICTR
JSR LSKED
BRU NEX8
IPARC:MOV #4, TEMP4
CMP TEMP4, ICTR
BNE C, RXDCK
INC ICTR
AND NNN, NR, TEMP8
BEQ RXDCK
MOV RXD, IBRK
AND RSM, NR, TEMP8
BEQ TEMP8, ODD
COM IPAR, IPAR
ODD:BNE IPAR, RXD, IPER
BRU LSK
IPER:BIS #3, STATINP
MOV(W7) STATINP,STATUS
LSK: JSR LSKED
BRU NEX8
RXDCK:BNE RXD, RDXSET
MOV RXD, IBRK

```

```

BNE IBRK, FRE
MOV #1, SYNEDO
BIS #0, STATINP
FRE: BIS #0, STATINP
RDYSET: BNE RXRDY, OER
MOV(W6) #1, RXRDY
BIS #1, STATINP
BRU RPUP
OER: BIS #4, STATINP
RPUP: MOV RS, RP
MOV #0, ISHF
MOV(W7) STATINP, STATUS
BRU HEX8
HEX75: BIS #0, STATINP
BIS #2, STATINP
MOV STATINP, STATUS
JSR CTXRD
MOV #1, TXEMPTY
MOV (W0) #1, TXD
MOV #1, S8D
HEX8: MOV D1, D10
MOV RESET, ORESET
MOV NDSR, NDSRO
MOV MCS, MCSO
MOV RXD, RXJO
MOV MRD, MRDO
MOV MAR, MARO
MOV #0, EX
TXCTS: AND ROM, CR, TEMP8
BEQ TEMP8, RET
BNE MCTS, RET
MOV(W2) #0, TXD
BIR #2, STATINP
MOV(W7) STATINP, STATUS
JSR SHFTSRED
MOV #1, OCTR
MOV #0, PAR
MOV #0, TPF
MOV #0, S8D
MOV TP, TS
RET: RTS
SHFTSRED: BRU TAB801
400: MOV #0, EX
401: MOV(W6) #1, O8CF
RTS
402: MOV(W11) #1, O8CF
RTS
403: MOV(W12) #1, O8CF
RTS
18KED: BRU TAB591
500: MOV #0, EX
501: MOV(W6) #1, 18CF
RTS
502: MOV(W11) #1, 18CF

```

```

;BREAK CHAR?
;SET BREAK DETECT PIN.
;UPDATE STATUS
;SET FRAMING ERROR STATUS
;CHECK FOR OVERRUN ERROR
;RXRDY = 1 IN 4TCY
;SET RXRDY STATUS
;SET OVERRUN STATUS BIT
;RESET ISHF
;UPDATE STATUS IN 14 TCY
;SET TXRDY & TXEMPTY STATUS BITS.
;UPDATE TXRDY.
;TXEMPTY GET SET TO 1.
;UPDATE VARIABLES
;EXIT MODULE
;CHECK TXEN
;CHECK NCTS
;SCHEDULE START BIT IN 50NS(ASSUMED)
;RESET TXEMPTY STATUS IN 14TCY
;RESET TXEMPTY PIN ALSO
;SCHEDULE FIRST BIT SHIFT
;INITIALISE OUTPUT COUNTER
;ZERO PARITY
;ZERO TP FULL FLAG
;PARA T REG TO SER TREG
;WHICH BAUDRATE?
;SELFCALL IN 13.02 USEC
;SELFCALL IN 52.08 USEC
;SELFCALL IN 104.16 USEC
;WHICH BAUDRATE?
;ILLEGAL ,EXIT

```

```

RTS
503:MOV(W12) #1,ISCF
RTS

CTXRD:BNE NCTS,RTXRD
      AND ROM,CR,TEMP8
      BEQ TEMP8,RTXRD
      AND ROM,STATINP,TEMP8
      BEQ TEMP8,RTXRD
      MOV (W1) #1,TXRDY
RTS

RTXRD:MOV (W1) #0,TXRDY
RTS

TAB1:BYT 100,101,102,103
TAB2:BYT 200,201,202,203
TAB3:BYT 300,301,302,303
TAB4:BYT 400,401,402,403
TAB5:BYT 500,501,502,503
ROM:  BYT #1
R1M:  BYT #2
R2M:  BYT #4
R3M:  BYT #8
R4M:  BYT #16
R5M:  BYT #32
R6M:  BYT #64
R7M:  BYT #128
R01M: BYT #3
R23M: BYT #12
R67M: BYT #192
      END

      ; IF NCTS=1, SCHEDULE TXRDY=0.
      ; IF TXEN=0, SCHEDULE TXRDY=0.
      ; IF D.B. BUFFER IS NOT EMPTY,
      ; SCHEDULE TXRDY=0.

```

```

REG(8)  TEMP1,TEMP2
PIN      8228(1,6),4044(9,16),2708(17,24),8251(25,32)
PIN      8255(33,40),BUSOUT(41,46),EX(150)
EW       W10(1)
STR: AND  8228,4044,TEMP1
AND      2708,8251,TEMP2
AND      8255,TEMP1,TEMP1
AND      TEMP1,TEMP2,TEMP2
MOV(W10) TEMP2,BUSOUT
MOV      #0,EX
BRU      STR
END

```

```

THIS CHIP SIMULATES CHIP SELECT LOGIC.
SELECTION LOGIC IS AS FOLLOWS.
IF ADRH = 00H      SELECT ROM
IF ADRH = 10H      SELECT RAM32RB
IF ADRH = 80H      SELECT A8255
IF ADRH = 40H      SELECT A8251
IF NONE OF THE ABOVE , DESELECT ALL FOUR CHIPS.
-----

```

```

REG(6) TEMP8, CAT
PIN ADRL(1,8),ADRH(9,16)
PIN ROM(17),RAM(18),A8251(19),A8255(20)
PIN EX(150)
EVM V30(30)

```

```

AND #240,ADRH,TEMP8
BEQ TEMP8,SRAM
MOV #128,CAT
BEQ CAT,TEMP8,YES55
MOV #16,CAT
BEQ CAT,TEMP8,SRAM
MOV #64,CAT
BEQ CAT,TEMP8,YES51
MOV(W30) #1,ROM
MOV(W30) #1,RAM
MOV(W30) #1,A8255
MOV(W30) #1,A8251
MOV #0,EX

```

```

YES55: MOV(W30) #0,A8255
        MOV(W30) #1,ROM
        MOV(W30) #1,RAM
        MOV(W30) #1,A8251
        MOV #0,EX

```

```

YES51: MOV(W30) #0,A8251
        MOV(W30) #1,ROM
        MOV(W30) #1,RAM
        MOV(W30) #1,A8255
        MOV #0,EX

```

```

SRAM:  MOV(W30) #0,ROM
        MOV(W30) #1,RAM
        MOV(W30) #1,A8255
        MOV(W30) #1,A8251
        MOV #0,EX

```

```

SRAM:  MOV(W30) #0,RAM
        MOV(W30) #1,ROM
        MOV(W30) #1,A8255
        MOV(W30) #1,A8251
        MOV #0,EX

```

```

END

```

```

REG(5) ADO,TEMP5 ;4044 RAM TIMING,32 LOC
REG(1) NCSO,TEMP1,TEMP2,RWO ;SIMULATES ADDRESS RIPPLE
PIN DI(1,8),DO(9,16),NCS(17),RW(18)
PIN AD(19,23),ADB(24,28),DID(29,36),EN(37),ARD(38),NWR(39)
PIN EX(150),SCA(151),SCDW(152)
EVA V35(350),W10(100)
MOV EN,TEMP1 ;SIN EXT SEL LOGIC
COM TEMP1
AND NRD,RW,TEMP2 ;NCS=(EN.(NRD'+WR'))'=EN'+NRD.WR
OR TEMP1,TEMP2,NCS
BEQ SCA,NEX1 ;INTERNAL ADDR SELF CALL?
MOV R0,SCA ;ADDR CHANGE,READ OR WRITE?
BRU R0V ;INTERNAL WRITE SELF CALL?
NEX1:BEQ SCDW,NEX2 ;WRITE CURRENT DATA AT OLD ADDR
IDX ADB(0),5,1 ;ADDR CHANGE?
MOV R0,SCDW ;SCHED INT ADDR CHANGE
NEX2:BEQ AD,ADO,CSCK ;CHANGE IN NCS?
MOV(W35) AD,ADB
MOV(W35) R1,SCA
CSCK:BEQ NCS,NCSO,RNCK
R0V: BEQ NCS,NEX4
MOV(W10) R255,DO ;DO =ALL 1'S
BRU UPDATE ;CHANGE IN WR?
RNCK:BNR RW,RND,NEX3
UPDATE:MOV NCS,NCSO
MOV RW,RND
MOV AD,ADO
MOV R0,EX ;EXIT
NEX3:BEQ R1,NCS,UPDATE ;READ OR WRITE?
NEX4:BEQ RW,WRITE
IDX ADB(0),5,1
MOV(W10) MEM01,DO ;OUTPUT DATA IN 100NS
BRU UPDATE
WRITE:MOV(W10) R1,SCDW
MOV(W10) DI,DID ;PROPAGATE INPUT DATA
BRU UPDATE
MEM: BYT R0,R0,R0,R0,R0,R0,R0,R0
BYT R0,R0,R0,R0,R0,R0,R0,R0
BYT R0,R0,R0,R0,R0,R0,R0,R0
END

```

```

REG(9) TEMPS,ADO
REG(1) NCSO,TEMP1
PIN DO(1,8),NCS(9) ;INITIALIZED TO QCPU2LS V2.6.2
PIN AD(10,18),ADDR(19,27),EX(150),SCF(151)
EVA W350(350),W100(100)
XOR AD,ADO,TEMP5 ; ADDR CHANGE?
BEQ TEMPS,SCCK
MOV(W350) AD,ADDR
MOV(W350) #1,SCF
SCCK:BEQ SCF,CSCK
BEQ #0,SCF
BEQ NCS,NEX2
BRU NIZ
CSCK:XOR NCS,NCSO,TEMP1 ;CHANGE IN NCS?
BEQ TEMP1,UPDATE
BEQ NCS,NEX2 ;CHECK NCS
NIZ: MOV(W100) #255,DO ;DO = ALL 1'S
UPDATE:MOV NCS,NCSO ;UPDATE
MOV AD,ADO
MOV #0,EX ;EXIT
NEX2:LDX ADDR(0),9,1
MOV(W100) MEM#1,DO ;OUTPUT DATA IN 100NS
MOV NCS,NCSO ;UPDATE
MOV AD,ADO
MOV #0,EX ;EXIT
MEM:
BYT #195, #56, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #175, #211, #2, #118, #0, #0, #0, #0, #0, #0
BYT #89, #31, #16, #58, #152, #1, #6, #51
BYT #17, #65, #204, #42, #154, #1, #235, #75
BYT #195, #77, #0, #247, #118, #27, #130, #206
BYT #215, #194, #48, #0, #210, #48, #0, #226
BYT #88, #0, #202, #97, #0, #195, #48, #0
BYT #247, #180, #47, #4, #144, #242, #48, #0
BYT #202, #48, #0, #194, #114, #0, #195, #48
BYT #0, #247, #153, #250, #48, #0, #45, #189
BYT #194, #48, #0, #218, #48, #0, #66, #161
BYT #179, #197, #35, #227, #140, #170, #157, #194
BYT #48, #0, #210, #145, #0, #195, #48, #0
BYT #247, #11, #128, #145, #193, #130, #184, #194
BYT #48, #0, #242, #161, #0, #195, #48, #0
BYT #195, #48, #0, #247, #171, #47, #147, #194
BYT #48, #0, #120, #31, #15, #67, #120, #178
BYT #250, #191, #0, #195, #48, #0, #247, #60
BYT #250, #48, #0, #194, #48, #0, #95, #29
BYT #147, #234, #48, #0, #39, #198, #5, #39
BYT #188, #194, #48, #0, #39, #39, #254, #114
BYT #194, #48, #0, #25, #25, #35, #22
BYT #204, #128, #140, #131, #133, #137, #130, #210
BYT #48, #0, #238, #45, #194, #48, #0, #17

```

BYT #133, #1, #26, #235, #190, #194, #48, #0
BYT #23, #225, #232, #194, #48, #0, #218, #48
BYT #0, #205, #156, #1, #210, #48, #0, #198
BYT #35, #7, #222, #110, #194, #48, #0, #62
BYT #198, #198, #206, #245, #175, #241, #210, #48
BYT #0, #202, #48, #0, #242, #48, #0, #234
BYT #48, #0, #39, #254, #250, #194, #48, #0
BYT #62, #0, #71, #14, #255, #175, #183, #167
BYT #194, #48, #0, #161, #194, #48, #0, #177
BYT #169, #194, #48, #0, #169, #161, #176, #185
BYT #194, #48, #0, #177, #168, #185, #194, #48
BYT #0, #169, #194, #48, #0, #62, #51, #33
BYT #0, #16, #17, #1, #16, #1, #2, #16
BYT #2, #47, #18, #54, #193, #10, #238, #51
BYT #194, #48, #0, #58, #1, #16, #254, #204
BYT #194, #48, #0, #62, #193, #78, #185, #194
BYT #48, #0, #47, #50, #0, #16, #190, #194
BYT #48, #0, #33, #85, #170, #34, #1, #16
BYT #235, #126, #187, #194, #48, #0, #35, #126
BYT #186, #194, #48, #0, #198, #1, #52, #190
BYT #194, #48, #0, #62, #255, #211, #2, #118
BYT #194, #116, #170, #102, #33, #0, #0, #57
BYT #235, #33, #170, #170, #249, #33, #0, #0
BYT #62, #170, #57, #188, #194, #48, #0, #189
BYT #194, #48, #0, #47, #33, #85, #85, #249
BYT #33, #0, #0, #57, #188, #194, #48, #0
BYT #189, #194, #48, #0, #33, #0, #0, #249
BYT #59, #62, #255, #57, #188, #194, #48, #0
BYT #189, #194, #48, #0, #35, #47, #188, #194
BYT #48, #0, #189, #194, #48, #0, #235, #249
BYT #192, #248, #55, #201, #195, #48, #0, #247
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
END

TESTROM SOR A1 07/15/81 12:17 HP21 F 80 58 RECS VA TECH PRINTED 09/12/81 20:02 PAGE 002

BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
END

C.45./C.46.

[illegible]

```

REG(9) TEMP5,ADO
REG(1) NCSC,TEMP1
PIN DO(1,8),NCS(9) ;INITIALIZED TO QCPU3S V3.3
PIN AD(10,18),ADDB(19,27),EX(150),SCF(151)
EVA W350(350),W100(100)
XOR AD,ADO,TEMP5 ; ADDR CHANGE?
BEQ TEMP5,SCCK
MOV(W350) AD,ADDB ;SCHEDULE INT ADDR CHANGE
MOV(W350) #1,SCF
SOCK:BEQ SCF,CSCK
MOV #0,SCF
BEQ NCS,NEX2
BRU HIZ
CSCK:XOR NCS,NCSC,TEMP1 ;CHANGE IN NCS?
BEQ TEMP1,UPDATE
BEQ NCS,NEX2 ;CHECK NCS
HIZ: MOV(W100) #255,DO ;DO = ALL 1'S
; UPDATE:MOV NCS,NCSC ;UPDATE
MOV AD,ADO
MOV #0,EX
NEX2:IDX ADDB(0),9,1
MOV(W100) MEM#1,DO ;OUTPUT DATA IN 100NS
MOV NCS,NCSC ;UPDATE
MOV AD,ADO
MOV #0,EX
MEN:
BYT #195, #56, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0
BYT #175, #211, #2, #118, #0, #0, #0, #0
BYT #49, #31, #16, #58, #23, #1, #6, #51
BYT #17, #95, #204, #42, #25, #1, #235, #75
BYT #195, #77, #0, #247, #118, #27, #130, #206
BYT #215, #194, #48, #0, #210, #48, #0, #226
BYT #48, #0, #202, #97, #0, #195, #48, #0
BYT #247, #220, #89, #1, #144, #242, #48, #0
BYT #202, #48, #0, #194, #114, #0, #195, #48
BYT #0, #247, #153, #250, #48, #0, #196, #98
BYT #1, #194, #48, #0, #218, #48, #0, #66
BYT #161, #220, #48, #0, #79, #197, #35, #227
BYT #204, #104, #1, #170, #157, #196, #48, #0
BYT #210, #151, #0, #195, #48, #0, #247, #11
BYT #128, #204, #48, #0, #145, #193, #130, #184
BYT #194, #48, #0, #242, #170, #0, #195, #48
BYT #0, #247, #145, #161, #63, #212, #48, #0
BYT #153, #218, #184, #0, #195, #48, #0, #247
BYT #171, #212, #109, #1, #178, #244, #48, #0
BYT #250, #199, #0, #195, #48, #0, #247, #60
BYT #250, #48, #0, #194, #48, #0, #95, #29
BYT #252, #124, #1, #198, #5, #39, #188, #194
BYT #48, #0, #252, #48, #0, #39, #39, #244
BYT #135, #1, #22, #204, #128, #140, #131, #133
BYT #137, #130, #210, #48, #0, #238, #45, #194

```

BVT #48, #0, #17, #24, #1, #26, #235, #190
 BVT #194, #48, #0, #23, #222, #232, #194, #48
 BVT #0, #218, #48, #0, #205, #27, #1, #210
 BVT #48, #0, #198, #55, #7, #222, #110, #194
 BVT #48, #0, #62, #255, #211, #2, #118, #194
 BVT #116, #170, #102, #33, #0, #0, #17, #0
 BVT #0, #57, #235, #49, #170, #170, #62, #170
 BVT #57, #188, #194, #48, #0, #189, #194, #48
 BVT #0, #47, #49, #85, #33, #0, #0
 BVT #57, #188, #194, #48, #0, #189, #194, #48
 BVT #0, #33, #0, #0, #249, #175, #57, #188
 BVT #194, #48, #0, #189, #194, #48, #0, #235
 BVT #249, #192, #248, #55, #201, #195, #48, #0
 BVT #247, #180, #216, #200, #47, #4, #192, #195
 BVT #48, #0, #45, #189, #200, #195, #48, #0
 BVT #140, #208, #195, #48, #0, #47, #147, #194
 BVT #48, #0, #224, #120, #31, #15, #87, #120
 BVT #240, #195, #48, #0, #240, #147, #208, #234
 BVT #48, #0, #39, #216, #195, #48, #0, #254
 BVT #114, #194, #48, #0, #25, #25, #35, #35
 BVT #184, #248, #195, #48, #0

END

```

REG(9) TEMP5, ADO
REG(1) NCSC, TEMP1
PIN DO(1,8), NCS(9) ;INITIALIZED TO QCPU4S V4.3
PIN AD(10,18), ADD8(19,27), EX(150), SCF(151)
EVA W350(350), W100(100)
XOR AD, ADO, TEMP5 ; ADDR CHANGE?
BEQ TEMP5, SOCK
MOV(W350) AD, ADD8 ; SCHEDULE INT ADDR CHANGE
MOV(W350) #1, SCF
SOCK: BEQ SCF, CSCK
MOV #0, SCF
BEQ NCS, NEX2
BRU HIZ
CSCK: XOR NCS, NCSC, TEMP1 ;CHANGE IN NCS?
BEQ TEMP1, UPDATE
BEQ NCS, NEX2 ;CHECK NCS
HIZ: MOV(W100) #255, DO ;DO = ALL 1'S
UPDATE: MOV NCS, NCSC ;UPDATE
MOV AD, ADO
MOV #0, EX
NEX2: IDX ADD8(0), 9, 1
MOV(W100) MEM#1, DO ;OUTPUT DATA IN 100NS
MOV NCS, NCSC ;UPDATE
MOV AD, ADO
MOV #0, EX
MEM:
BYT #255, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #0, #0, #0, #0, #0, #0, #0, #0, #0, #0
BYT #175, #211, #2, #118, #0, #0, #0, #0, #0, #0
BYT #19, #31, #16, #58, #27, #1, #6, #51
BYT #17, #85, #204, #42, #29, #1, #235, #75
BYT #195, #77, #0, #247, #118, #27, #147, #156
BYT #218, #48, #0, #238, #33, #61, #194, #48
BYT #0, #226, #48, #0, #202, #99, #0, #195
BYT #48, #0, #247, #140, #161, #144, #189, #194
BYT #48, #0, #250, #48, #0, #47, #141, #242
BYT #195, #48, #0, #202, #48, #0, #194, #124, #0
BYT #0, #178, #66, #23, #148, #194, #48, #0
BYT #242, #143, #0, #195, #48, #0, #247, #230
BYT #255, #79, #197, #227, #170, #11, #137, #210
BYT #48, #0, #132, #234, #48, #0, #210, #165
BYT #0, #195, #48, #0, #247, #152, #193, #7
BYT #184, #194, #48, #0, #173, #250, #180, #0
BYT #195, #48, #0, #247, #63, #31, #218, #48
BYT #0, #214, #179, #87, #120, #178, #242, #48
BYT #0, #60, #194, #48, #0, #250, #48, #0
BYT #95, #29, #147, #218, #210, #0, #195, #48
BYT #0, #247, #39, #198, #5, #39, #188, #194
BYT #48, #0, #39, #39, #254, #114, #194, #48
BYT #0, #25, #25, #35, #22, #204, #20
BYT #128, #140, #139, #141, #137, #138, #210, #48

```

BYT #0, #238, #48, #194, #48, #0, #17, #28
BYT #1, #26, #235, #190, #194, #48, #0, #23
BYT #222, #232, #194, #48, #0, #218, #48, #0
BYT #205, #31, #1, #210, #48, #0, #198, #55
BYT #7, #222, #110, #194, #48, #0, #62, #255
BYT #211, #2, #118, #150, #116, #170, #102, #33
BYT #0, #0, #17, #0, #0, #57, #235, #49
BYT #170, #170, #62, #170, #57, #188, #194, #48
BYT #0, #189, #194, #48, #0, #47, #49, #85
BYT #85, #33, #0, #0, #57, #188, #194, #48
BYT #0, #189, #194, #48, #0, #33, #0, #0
BYT #249, #175, #57, #188, #194, #48, #0, #189
BYT #194, #48, #0, #235, #249, #192, #248, #55
BYT #201, #195, #48, #0, #247
END

Appendix D

Test System

Data File

```

SYSCPU DATA A1 09/11/81 16:58 HP21 F 80 197 RECS VA TECH
;MIGNA ADDRESS DATA MREAD MWR ROM RAM & 51 8255
Y 1-7, 11-18 21-28 30 33 35 37-38 40
; 7000000
; #1 -> RAM32RB, #2 -> CSL, #3 -> BUS, #4 -> B8228
; #5 -> A825V6, #6 -> A825V5, #7 -> TESTCPU*, #8 -> A8080N
;=====
; BUS MODULE
; 3,41 6,1 4,17 1,1 5,7 0,21 ;BUSOUT (LSB - M
; 3,42 6,2 4,18 1,2 5,8 0,22 ;FRONT PANEL DATA
; 3,43 6,3 4,19 1,3 5,9 0,23
; 3,44 6,4 4,20 1,4 5,10 0,24
; 3,45 6,5 4,21 1,5 5,11 0,25
; 3,46 6,6 4,22 1,6 5,12 0,26
; 3,47 6,7 4,23 1,7 5,13 0,27
; 3,48 6,8 4,24 1,8 5,14 0,28
; 3,49 6A
; A C 3 41 255
; A C 3 25 255
; A C 3 33 255
;=====
; ROMCPU*
; 7,1 3,17 ; DO => DATA OUT CONNECTED TO BUS MODULE
; 7,2 3,18
; 7,3 3,19
; 7,4 3,20
; 7,5 3,21
; 7,6 3,22
; 7,7 3,23
; 7,8 3,24
; 7,9 17
; 7,10 151
; 7,11 151
; A C 7 1 255
; A C 7 168 1
;=====
; PRESET INTERNAL FLAG (NCSO)
;=====
; RAM32RA
; 1,9 3,9 ; DATA OUT CONNECTED TO BUS MODULE
; 1,10 3,10
; 1,11 3,11
; 1,12 3,12
; 1,13 3,13
; 1,14 3,14
; 1,15 3,15
; 1,16 3,16
; 1,17 8
; 1,18 23
; 1,19 39
; 1,20 152
; 1,21 152
; A C 1 9 255
;=====
; A8228
; 4,9 8,1 ; DATA OUT TO 8080
; 4,10 8,2
; 4,11 8,3

```

D.1.

```

      1.36      5.2      0.30      ; MREAD
      1.18      5.3      0.33      ; MWRITE
                                     ; DATA OUT TO BUS MODULE
  
```

```

      8.4
      8.5
      8.6
      8.7
      8.8
      8.9
      8.10
      8.11
      8.12
      8.13
      8.14
      8.15
      8.16
      8.17
      8.18
      8.19
      8.20
      8.21
      8.22
      8.23
      8.24
      8.25
      8.26
      8.27
      8.28
      8.29
      8.30
      8.31
      8.32
      8.33
      8.34
      8.35
      8.36
      8.37
      8.38
      8.39
      8.40
      8.41
      8.42
      8.43
      8.44
      8.45
      8.46
      8.47
      8.48
      8.49
      8.50
      8.51
      8.52
      8.53
      8.54
      8.55
      8.56
      8.57
      8.58
      8.59
      8.60
      8.61
      8.62
      8.63
      8.64
      8.65
      8.66
      8.67
      8.68
      8.69
      8.70
      8.71
      8.72
      8.73
      8.74
      8.75
      8.76
      8.77
      8.78
      8.79
      8.80
      8.81
      8.82
      8.83
      8.84
      8.85
      8.86
      8.87
      8.88
      8.89
      8.90
      8.91
      8.92
      8.93
      8.94
      8.95
      8.96
      8.97
      8.98
      8.99
      9.00
      9.01
      9.02
      9.03
      9.04
      9.05
      9.06
      9.07
      9.08
      9.09
      9.10
      9.11
      9.12
      9.13
      9.14
      9.15
      9.16
      9.17
      9.18
      9.19
      9.20
      9.21
      9.22
      9.23
      9.24
      9.25
      9.26
      9.27
      9.28
      9.29
      9.30
      9.31
      9.32
      9.33
      9.34
      9.35
      9.36
      9.37
      9.38
      9.39
      9.40
      9.41
      9.42
      9.43
      9.44
      9.45
      9.46
      9.47
      9.48
      9.49
      9.50
      9.51
      9.52
      9.53
      9.54
      9.55
      9.56
      9.57
      9.58
      9.59
      9.60
      9.61
      9.62
      9.63
      9.64
      9.65
      9.66
      9.67
      9.68
      9.69
      9.70
      9.71
      9.72
      9.73
      9.74
      9.75
      9.76
      9.77
      9.78
      9.79
      9.80
      9.81
      9.82
      9.83
      9.84
      9.85
      9.86
      9.87
      9.88
      9.89
      9.90
      9.91
      9.92
      9.93
      9.94
      9.95
      9.96
      9.97
      9.98
      9.99
  
```

D.2.

; DATA BUS OUT TO 8228

```

      8.1
      8.2
      8.3
      8.4
      8.5
      8.6
      8.7
      8.8
      8.9
      8.10
      8.11
      8.12
      8.13
      8.14
      8.15
      8.16
      8.17
      8.18
      8.19
      8.20
      8.21
      8.22
      8.23
      8.24
      8.25
      8.26
      8.27
      8.28
      8.29
      8.30
      8.31
      8.32
      8.33
      8.34
      8.35
      8.36
      8.37
      8.38
      8.39
      8.40
      8.41
      8.42
      8.43
      8.44
      8.45
      8.46
      8.47
      8.48
      8.49
      8.50
      8.51
      8.52
      8.53
      8.54
      8.55
      8.56
      8.57
      8.58
      8.59
      8.60
      8.61
      8.62
      8.63
      8.64
      8.65
      8.66
      8.67
      8.68
      8.69
      8.70
      8.71
      8.72
      8.73
      8.74
      8.75
      8.76
      8.77
      8.78
      8.79
      8.80
      8.81
      8.82
      8.83
      8.84
      8.85
      8.86
      8.87
      8.88
      8.89
      8.90
      8.91
      8.92
      8.93
      8.94
      8.95
      8.96
      8.97
      8.98
      8.99
      9.00
      9.01
      9.02
      9.03
      9.04
      9.05
      9.06
      9.07
      9.08
      9.09
      9.10
      9.11
      9.12
      9.13
      9.14
      9.15
      9.16
      9.17
      9.18
      9.19
      9.20
      9.21
      9.22
      9.23
      9.24
      9.25
      9.26
      9.27
      9.28
      9.29
      9.30
      9.31
      9.32
      9.33
      9.34
      9.35
      9.36
      9.37
      9.38
      9.39
      9.40
      9.41
      9.42
      9.43
      9.44
      9.45
      9.46
      9.47
      9.48
      9.49
      9.50
      9.51
      9.52
      9.53
      9.54
      9.55
      9.56
      9.57
      9.58
      9.59
      9.60
      9.61
      9.62
      9.63
      9.64
      9.65
      9.66
      9.67
      9.68
      9.69
      9.70
      9.71
      9.72
      9.73
      9.74
      9.75
      9.76
      9.77
      9.78
      9.79
      9.80
      9.81
      9.82
      9.83
      9.84
      9.85
      9.86
      9.87
      9.88
      9.89
      9.90
      9.91
      9.92
      9.93
      9.94
      9.95
      9.96
      9.97
      9.98
      9.99
  
```

CONNECTION FROM ADDR BUS.

```

      0.11      2.1      5.5      6.22
      0.12      2.2      5.6
      0.13      2.3
      0.14      2.4
      0.15      2.5
      0.16      2.6
      0.17      2.7
      0.18      2.8
  
```

SYSCPU DATA

8.25 7.18 0.1 2.9
 8.26 0.2 2.10
 8.27 0.3 2.11
 8.28 0.4 2.12
 8.29 0.5 2.13
 8.30 0.6 2.14
 8.31 0.7 2.15
 8.32 0.8 2.16
 N 8 150 153
 A C 8 34 1
 A C 8 41 1
 A C 8 9 255

CSL (CHIP SELECT LOGIC)

2.17 7.9 0.35
 2.18 1.37 0.37
 2.19 6.21 0.38
 2.20 5.4 0.40

2 1 8
 N 2 9 16
 N 2 150 150

8251 3.25
 6.9 3.26
 6.10 3.27
 6.11 3.28
 6.12 3.29
 6.13 3.30
 6.14 3.31
 6.15 3.32
 6.16 6.25

; CONNECTION FOR 'TEST51'

6 1 8
 N 6 17 21
 N 6 23 25
 N 6 33 33
 N 6 151 154
 C 6 9 255
 A C 6 29 1
 A C 6 35 4
 A C 6 202 4
 A C 6 19 1
 A C 6 20 1
 A C 6 21 1
 A C 6 30 1
 A C 6 31 1
 A C 6 32 1
 A C 6 43 1
 A C 6 44 1
 A C 6 45 1
 A 6 17 10 1
 A 6 17 40 0

A8255 3.33
 5.15

D.3.

5.16 3.34
5.17 3.35
5.18 3.36
5.19 3.37
5.20 3.38
5.21 3.39
5.22 3.40

5 1 14
5 150 155
5 23 255
5 39 255
5 35 1
5 36 1
5 57 1
5 58 1
5 59 1
5 60 1
5 61 1
5 62 1
5 71 15
5 75 15

=====

1 0
1 200
1 300
1 400
1 500
1 600
1 700
1 800
1 900
1 1000

D.4. : FIRST CALL TO START UP

APPENDIX E

Fault Experiments Summary

CHIP: 8080 CPU

FAULT LIST

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
Push PSW results in A & PSW pushed onto stack in reverse order. PSW value on stack wrong	*			Detected by CPI 4H at 75H
WR appeared before data	*			detected by CPI 4H at 75H
XTHL writes H twice and L not at all	*			
DCR C decremented 80Hz got 00H	*			detected by CMPH/JNZ ERR at 8CH
ADD did not set AC flag correctly	*			
Incorrect CY flag after subtracting zero CY = 1 instead of 0	*			
Improper RAL execution LSB wrong	*			
SBI subtracted CY instead of CY	*			Detected by CMPH/JNZ ERR at 8CH
Incorrect CY flag after DAA operation	*			
Incorrect parity flag	*			detected by JPO at 52H
Incorrect CY flag after subtract and compare instructions	*			detected by SBB C and JH at 62H
Multiple register select on READ. Select B also selects D	*			
Multiple register select on READ. Select C also selects L.	*			
Multiple register select on READ. Select H also selects B.	*			

CHIP: 8080 CPU

FAULT LIST

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
Multiple register select on READ. Select D also select E	*			
Register C bit 3 stuck-at-0	*			
Register C bit 5 stuck-at-1	*			
CHA always sets bit 2 of of Accum.	*			
WR pulse duration too short (about half of nom. value)	*			
Parity flag determination ignores bit 3. Treats it as stuck-at-1	*			
Parity flag determination ignores bit 3. Treats it as stuck-at-0	*			
SYNC pulse is delayed till about the falling edge of $\phi 2$ clock			*	Fault had no effect on the system performance.
Data Line D3 stuck-at-1		*		Hardware timer would Probably detect the fault
Data line stuck-at-0	*			
Reset pin open (stuck-at-1)	*			8080 CPU does not function hardware timer should detect the fault
Interrupt pin open (stuck-at-1)	*			Results in RST7 which causes a call to address 0038H. The test program incidentally, starts at the same location. Timer detected
Incorrect register select on READ & WRITE (select E selects B)	*			
Incorrect register select on READ & WRITE (select E selects C)	*			
Multiple register select on write (Write E also writes to C)	*			

CHIP: 8080 CPU

E. 3.

CHIP: 8228

E.4.

FAULT LIST

CHIP: BUS

[illegible]

FAULT LIST

CHIP: ROM

[illegible]

CHIP: RAN

E.7.

FAULT LIST

CHIP: 8251

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
TxEMPTY & TxRDY status bits do not get set to '1' at the end of Reset mode word & command word	*			Test program gets stuck in a status check loop. Detected by hardware timer
Data line D ₀ open (s-a-l)	*			
Data line D ₀ stuck-at-0	*			hardware timer should detect the fault
WR shorted to ground	*			hardware timer should detect the fault
Reset line open (stuck-at-1)	*			
No stop bit transmitted	*			
Status register s-a-0	*			Test program gets stuck in a status check loop. Detected by hardware timer
Output parity always even			*	Chip was configured for NO parity-hence no parity checking
short between C/D & RD inputs	*			
short between WR & CS inputs			*	
short between CS & C/D inputs	*			hardware timer should detect the fault
RD stuck-at-0	*			
RD input open (s-a-l)	*			
C/D input stuck-at-0	*			hardware timer should detect the fault
C/D input open (stuck-at-1)	*			hardware timer should detect the fault

CHIP: 8251

FAULT LIST

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
CS stuck-at-0	*			
WR input open (stuck-at-1)	*			
CS input open (stuck-at-1)	*			
8251 does not respond to read or write commands. Data bus to Hi-Z state	*			
On a data read, data remains stable for only 100NS after it is gated onto the bus	*			
Write pulse requires to be 500NS long instead of 250NS	*			
Both mode word and command word are loaded into the command register.	*			hardware timer should detect the fault
Output counter OCTR stuck-at-all 1's	*			

FAULT LIST

CHIP: 8255

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
Short between adjacent data input lines (D ₂ and D ₃)	*			Data read back is different from data output to the 8255
data line shorted to ground (D ₇)	*			"
data line open (D ₈)	*			
delay for data appearing at an output port excessive (delay increase of 100NS)			*	(chip not meeting time specifications)
wrong bit is SET/RESET stuck-at-0 on bit select line #0	*			
wrong bit is SET/RESET stuck-at-1 on bit select line #2	*			
Bit SET/RESET operation is reversed	*			
BIT reset does not work set works ok	*			
Bit set does not work Reset works ok	*			
Address pin open (A ₁)				
Address pin open (A ₈)	*			
Port C does not work in split mode			*	Test routine does not test port C in split mode.
Output drivers in the the 8255 fail (bit 5, port stuck at 0)	*			
incorrect register selection (writing to port A results in writing to port B too)			*	not detected since data is same at both ports due to wraparound
incorrect register selection (read from port port A results in data from port A & B			*	"

FAULT LIST

CHIP: 8255

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
output driver in the 8255 fails. Bit stuck-at-1	*			
incorrect register selection (write to port A results in writing to port B)	*			
Port C lower (4 bits) stuck at 0	*			
Hold times for address and data had to be too (300NS from end of write pulse)	*			chip not meeting timing specifications
Bit set/reset command clears port C output completely	*			
Reset fails to clear A & B			*	
Port A (input) bit 1 open	*			
Port B (input) bit 7 stuck-at-0	*			
Read port A operation fails (Data buffer does not get loaded from port A)			*	Data buffer contains data which was previously written to it.
A read operation always fails (Data bus tri-stated all the time)	*			
Pattern 0FFH fails to be read correctly (pattern sensitive fault)			*	test program does not test using the pattern FF H
A write operation fails completely	*			
short between RD and CS of 8255	*			
short between RESET & WR of 8255	*			
Port A (input) bit 1 stuck-at-0				

CHIP: 8255

FAULT LIST

FAULT DESCRIPTION	TEST RESULT			COMMENTS
	Detected	Program Control Lost	Not Detected	
Port B (input) bits 4 & 5 shorted	*			
Address pin stuck-at-0 (A ₀)	*			
Address pin stuck-at-0 (A ₁)	*			
short between address pins A ₀ & A ₁	*			
WR stuck-at-0	*			
WR open	*			
CS open	*			
RD open	*			

Appendix F

MOVI RAM Test Program Listing

```

*****
***
***      RAMMOVI ASM80          VERSION 1.2
***
***      MOVI RAM TEST FOR 8080/8085
***
***      MOVING INVERSIONS TEST PATTERN
***
***      DAVE HAILSLETT      7/7/81
***
*****
***
***      SYMBOLS:
***      RAMBEG = START ADDR OF RAM (XX00H)
***      RAMBH  = HIGH BYTE OF RAMBEG
***      RAMEND = (END ADDR OF RAM)+1 (XX00H)
***      RAMEH  = HIGH BYTE OF RAMEND
***      RAMSIZ = RAMEND - RAMBEG (= #BYTES OF RAM)
***      RSIZH  = HIGH BYTE OF RAMSIZ
***
***      ** NOTE THAT RAMBEG, RAMEND, AND RAMSIZ MUST
***      ALL HAVE A LOW BYTE OF ZERO (IE., THEY MUST
***      BE OF THE FORM XX00 HEX.)
***
*****
***      REGISTERS:
***      A      COMPARISONS
***      B      NEW PATTERN
***      C      OLD PATTERN
***      D,E     ADDRESS INCREMENT/DECREMENT
***      H,L     MEMORY POINTER
***      SP      WRAP AROUND OFFSET
***
*****
***      NOTE:  SINCE THE STACK POINTER IS USED AS A
***      16 BIT REGISTER, INTERRUPTS MUST BE
***      DISABLED. (IF STACK IS USED DURING
***      THE RAM TEST, PATTERN DATA WILL BE
***      DESTROYED AND BOGUS ERRORS 'DETECTED'.)
***
*****
***
***      ORG      1000H
***      RAMBEG   EQU      1400H
***      RAMBH    EQU      RAMBEG/100H
***      RAMEND   EQU      1800H
***      RAMEH    EQU      RAMEND/100H
***      RAMSIZ   EQU      RAMEND - RAMBEG
***      RSIZH    EQU      RAMSIZ/100H
***
1000
1400
0014
1800
0018
0400
0004

```

F. 2.


```

10C2 320F11      ; NAMEC: STA MERR      ; STORE BAD MEMORY CONTENTS
10C5 79         ;      MOV A, C        ; AND NOW...
10C6 321011     ;      STA EXPC1      ; STORE THE EXPECTED CONTENTS
10C9 C30310     ; NAMEB: JMP MERR      ; STORE BAD MEMORY CONTENTS
10CC 320F11     ;      STA A, B        ; AND THEN...
10CF 78         ;      MOV A, EXPC1    ; STORE EXPECTED CONTENTS
10D0 321011     ; NAMEC: STA HLERR    ; SAVE H & L
10D3 220D11     ;      SHLD H, ENSGR   ; POINT TO ERROR MESSAGE
10D6 21EE10     ;      LXI H, ENSGR   ;
;
; PORT EQU 3C2CH ; CONSOLE TERMINAL PORT MEMORY MAPPED ADDR.
;
; PRINT SUBRT: PRINT STRING POINTED TO BY HL ON CONSOLE
;
; PRINT: MOV A, M      ; READ A CHAR FROM MEMORY
;      ORA A          ; TEST IT
;      JZ EXIT        ; WE'RE DONE IF ZERO
;      LDA PORT+1     ; READ STATUS
;      ANI 0AH        ; TRANSMIT BUFFER EMPTY?
;      JZ TEST        ; NO, WAIT
;      MOV A, M        ; GET CHAR AGAIN
;      INX H          ; POINT TO NEXT CHAR
;      STA PORT       ; WRITE THIS CHAR
;      JMP PRINT      ; BACK FOR NEXT ONE
;
; ENSGR: DB 0DH,0AH,'?RAM ERROR',0DH,0AH,00H ; ERROR MESSAGE
;
; MEMOK: DB 0DH,0AH,'RAM PASSED!',0DH,0AH,00H
;
; HLERR: DB 0, 0      ; STORAGE FOR H & L ON ERROR
; MERR:  DB 0         ; STORAGE FOR BAD MEMORY CONTENTS
; EXPC1: DB 0         ; STORAGE FOR EXPECTED MEMORY CONTENTS
;
; NO PROGRAM ERRORS
;
END

```

3080 MACRO ASSEMBLER, VER 2.4
ERRORS = 0 PAGE 6

SYMBOL TABLE

*** 01**

A	0007	B	0000	C	0001	CLOOP	1008
CNTF	102E	CNTR	107E	D	0002	DATO	1054
DATRO	10A4	E	0003	EMSGR	10EE	EXIT	108E
EXPT	1110	FTL1	1011	FTLOO	100E	H	0004
HLERR	110D	L	0005	LASTF	1040	LASTR	1090
M	0006	MEMOK	10FD	MERR	110F	NXTAD	10A9
OVR	1038	PORT	3C2C	PRINT	10D9	PSM	0006
RANBE	1400	RANBH	0014	RAMEB	10CC	RAMEC	10C2
RANWH	1408	RANEN	1800	RAMER	10D3	RAMES	0400
RANWTS	1000 *	RL2	1068	RLOOP	1017	RSIZH	0004
RTL1	1062	RVR5	1059	RX1	1050	RX2	10A0
SP	0006	TEST	10DE	UNDR	1088		

F. 6.

APPENDIX G

The following is a list of the micro-operations determined for the 8080 CPU based on the clock cycle by clock cycle breakdown of each instruction <4 , pp. 2-16 to 2-20) and the CPU functional block diagram <4 , p. 2-2) .

Key:

r8	Any 8-bit register of the register array
r16	Any 16-bit register pair of the register array
ACT	Accumulator Latch
A	Accumulator
CY	Carry flag
TMP	Temp. register
Dbus	8-bit data bus
Abus	16-bit address bus

8080 Micro-operations:

- 1) r16 = Abus
- 2) r16a + 1 = r16b
- 3) Dbus = TMP and IR
- 4) r8 = TMP
- 5) A = TMP
- 6) TMP = r8
- 7) TMP = A
- 8) Dbus = r8
- 9) Dbus = A
- 10) Dbus = TMP
- 11) TMP = Dbus
- 12) A = Dbus
- 13) r8 = Dbus
- 14) (HL) = (DE) [XCHG]
- 15) r16a = r16b
- 16) A = ACT
- 17) r8 = ACT
- 18) ACT + TMP = ALU [i.e. ALU outputs]
- 19) ACT + TMP + CY = ALU
- 20) ACT - TMP = ALU
- 21) ACT - TMP - CY = ALU
- 22) TMP + 1 = ALU
- 23) TMP - 1 = ALU
- 24) ALU = r8
- 25) ALU = A
- 26) ALU = Dbus
- 27) r16a - 1 = r16b
- 28) DAA = A, flags [decimal adjust]
- 29) ACT AND TMP = ALU
- 30) ACT OR TMP = ALU
- 31) ACT XOR TMP = ALU
- 32) ALU = flags S, Z, P & AC
- 33) ALU = CY
- 34) CY = ALU
- 35) A = ALU

36) Rotate Right
37) Rotate Right through CY
38) Rotate Left
39) Rotate Left through CY
40) NOT A = A
41) NOT CY = CY
42) 1 = CY
43) Judge Condition
44) 00 = W reg.
45) Flags = Dbus
46) Dbus = Flags
47) Set INTE F/F
48) Reset INTE F/F
49) Halt Mode
50) Status: Instruction Fetch
51) Status: Memory Read
52) Status: Memory Write
53) Status: Stack Read
54) Status: Stack Write
55) Status: IN Read
56) Status: OUT write
57) Status: Interrupt Acknowledge
58) Status: Halt Acknowledge
59) Status: Interrupt Ack. while Halt
60) Dbus = r16high and r16low

Appendix H

Checksum Calculation Program

APPENDIX H

```

100  /      CHKSUM -- CALCULATE CHECKSUMS FOR AN OBJECT FILE
      /      IN SEGMENTS OF A DESIRED LENGTH.
      /      THE SUM IS FORMED USING A MODULO 256 ADD WITH CARRY
      /      OUTS ADDED BACK INTO LSB.

110  /      THE SEGMENT LENGTHS AND DESIRED FINAL CHECKSUM
      /      ARE SELECTED BY THE USER.

120  /      DAVID HATSLETT      MICROPROCESSOR SELF-TEST PROJECT
      /

400  HEXO$ = "0123456789ABCDEF"
500  DEF FNM$(D$) = (INSTR(1,HEXO$,LEFT$(D$,1))-1)*16 +
      INSTR(1,HEXO$,MID$(D$,2,1))-1
      /      RETURN INTEGER VALUE OF 2 DIGIT HEX STRING
      /

1000 PRINT CHR$(10);"CHKSUM - V01.01";CHR$(10)
1010 PRINT "INPUT FILE"; : INPUT FIL$ :
      IF INSTR(1,FIL$,".")=0 THEN FIL$=FIL$+".HEX"
1020 OPEN "I", #1, FIL$
1030 PRINT "ROM SEGMENT SIZE"; : INPUT ROMSS$ :
      IF ROMSS$<3 THEN PRINT "?INVALID SIZE: ";ROMSS$ : GOTO 1030
1040 PRINT "DESIRED FINAL SUM"; : INPUT DSUM$ :
      IF DSUM$<0 OR DSUM$>255 THEN PRINT "?OUT OF RANGE, ENTER 0-255"
      GOTO 1040
1050 HLINE$ = "" : CSUM$ = 0
2000 FOR B$ = 1 TO ROMSS$ : GOSUB 10000 :
      IF OP$ = -1 THEN IF B$ = 1 THEN 2100 ELSE 2030
2005 ADDR1$=ADDR$ : IF B$=1 THEN ADDR1$=ADDR$
2010 CSUM$=CSUM$ + OP$ :
      Z$=CSUM$ AND (NOT 255) : CSUM$=CSUM$ AND 255:
      IF Z$ THEN CSUM$=CSUM$+1
2020 NEXT B$
2030 RSUM$=DSUM$-CSUM$ :
      IF RSUM$<0 THEN RSUM$=RSUM$-1
2040 RSUM$=RSUM$ AND 255
2050 PRINT "CHECKSUM FOR SEGMENT ";RIGHT$("0000"+HEX$(ADDR1$),4);
      "H TO "; RIGHT$("0000"+HEX$(ADDR1$),4);"H IS ";
      RIGHT$("0000"+HEX$(RSUM$),4);"H"
2060 CSUM$=0
2070 IF OP$<>-1 THEN 2000
2100 CLOSE #1 : GOTO 32767
9999 /
      /      *****
      /      READ AN OBJECT CODE BYTE FROM HEX FILE &
      /      RETURN IT IN OP$
10000 IF HLINE$="" THEN LINE INPUT #1, H$ : H$=MID$(H$,2) :
      IF MID$(H$,1,1)<>"":
      THEN 10000 ELSE HBYTES$=FNM$(MID$(H$,2,2)) :
      ADDR$=FNM$(MID$(H$,4,2))*256 + FNM$(MID$(H$,6,2)) :
      FOR$=-1 : HLINE$=MID$(H$,10,2*HBYTES$) : IF HBYTES$=0 THEN OP$=-1: RETURN
10010 OP$=FNM$(HLINE$) : OP$=LEFT$(HLINE$,2) :
      HLINE$=MID$(HLINE$,3)
10020 IF FOR$ THEN FOR$=0 ELSE ADDR$=ADDR$+1

```

Appendix I

RADC Microprocessor Self-Test Project

Hardware System Documentation

The self-test system consists of an 8080 CPU, 8228 system controller, 2 8251 serial I/O ports, an 8255 parallel I/O port, 8253 programmable interval timer (with 3 independent timers), 2 8-bit data latches for various control functions, 2K of ROM containing the system monitor, and 4K of RAM. The system employs memory mapped I/O and thus does not use IN or OUT instructions. The system memory map is shown below:

<u>Address (Hex)</u>	<u>Assignment</u>
0000 - 07FF	ROM (monitor)
0800 - 17FF	System RAM
3C24	Down-line load 8251, data port
3C25	Down-line load 8251, command port
3C28	CNTL1: 8-bit latch; see below
3C2C	Console 8251, data port
3C2D	Console 8251, command port
3C30	CNTL2: 8-bit latch; see below
3C38	8253 Timer 0 (interrupt)
3C39	8253 Timer 1 (timeout)
3C3A	8253 Timer 2 (unused)
3C3B	8253 Control
3C3C	8255 Port A
3C3D	8255 Port B
3C3E	8255 Port C
3C3F	8255 Control

The 8253 Timer 0 is used to generate the interrupt that initiates the periodic self-test. This timer should be configured for Mode 0 since the interrupt is generated on the rising edge of Output 0. A RST 3 is executed upon interrupt acknowledge, and control is passed to RAM location 0A00H. Timer 1 is used to generate a hardware timeout; this occurs if the self-test is not initiated or is not completed in its allotted time. Timer 1 should be configured for Mode 0, since its output must go high and stay high for the timeout. This causes the ERR-bar LED to go out (indicating error). Timer 2 is unused and thus available to the user. The Timer 0 and 1 Gate inputs are controlled by the CNTL2 latch (3C30H); this latch is also responsible for enabling or disabling the hardware timeout (see below). Both Timers 0 and 1 are driven by the processor clock (.89 MHz).

The CNTL1 8-bit latch (IC 22) is responsible for controlling the baud rates of both 8251's; it also controls the console 8251 wraparound (self-test) logic and drives the 'heartbeat' LED. Its bits have the following meanings:

<u>Bit(s)</u>	<u>Function</u>
0	Low connects console 8251 to the console terminal; High wraps serial output to serial input.
1-3	Console 8251 baud rate control (see below).
4	Controls the 'heartbeat' LED: low = ON; high = OFF.
5-7	Down-line load 8251 baud rate control (see below).

Assuming the 8251's are programmed for 16x operation, the baud rate control is defined as follows:

Bit:	3	2	1	--	console
	7	6	5	--	down-line load

	0	0	0		Do NOT use
	0	0	1		19200 baud
	0	1	0		9600 baud
	0	1	1		4800 baud
	1	0	0		2400 baud
	1	0	1		1200 baud
	1	1	0		600 baud
	1	1	1		300 baud

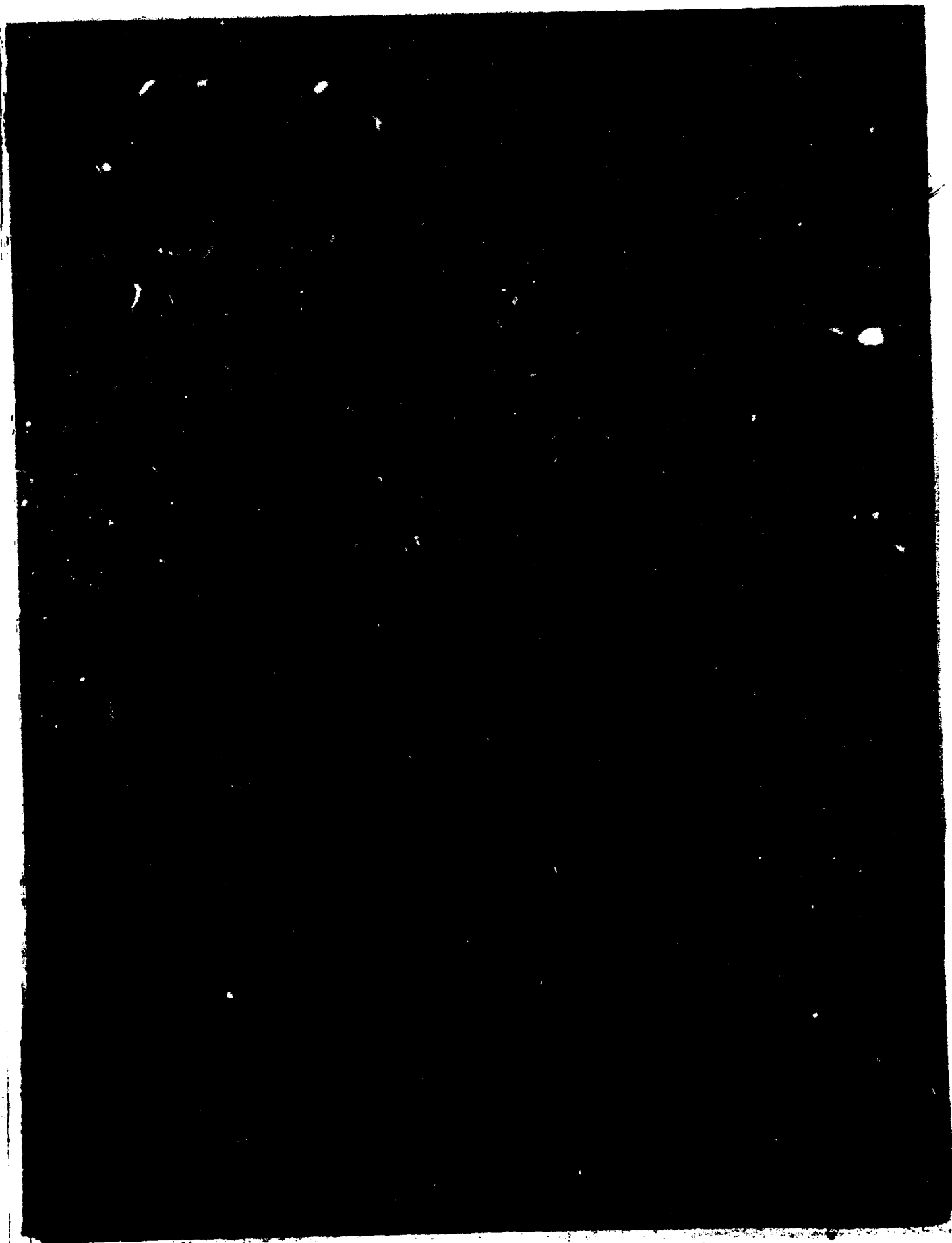
The CNTL2 8-bit latch (IC 17) is responsible for controlling the 8255 wraparound self-test logic, the 8253 Counters 0 & 1 gate inputs, and for enabling/disabling the hardware timeout. Its bits have the following functions:

<u>Bit</u>	<u>Function</u>
0	8253 Gate 0 (Timer 0)
1	8253 Gate 1 (Timer 1)
2	High allows hardware timeout; Low forces the timeout.
3	High disables hardware timeout; Low enables it.
4	Low enables 8255 Port A wraparound logic; High disables it (& tri-states Port A).
5	Low sets 8255 Port A wraparound INTO Port A; High sets wraparound OUT OF Port A.
6	Low enables 8255 Port C wraparound logic; High disables it (& tri-states Port C).
7	Low sets 8255 Port C wraparound INTO Port C; High sets wraparound OUT OF Port C.

Note that Bit 5 is meaningless if Bit 4 is 1, and Bit 7 is meaningless if Bit 6 is 1.

To enable a hardware timeout (Timer 1), Bits 2 and 3 should be 1 and 0, respectively. To indicate an error (fault detected), Bits 2 and 3 should both be set to 0; this forces a 'timeout' and turns off the ERR-bar LED.

The system's 7-segment display is connected to the 8255's Port B; if Port B is configured as an output, then Port B is used to drive the display. When Port B is configured as an input, Port A or Port C may be used to drive the display via the wraparound logic. Note that low bits light segments, while high bits turn segments off.



DATE
LMED
8