Singh, Harvinder

DESIGN MODIFICATIONS IN MICROPROCESSORS TO SIMPLIFY THEIR TESTING

The University of Oklahoma PH.D. 1982

THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN MODIFICATIONS IN MICROPROCESSORS TO SIMPLIFY THEIR TESTING

A DISSERTATION

SUBMITTED FOR THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

By

HARVINDER SINGH

Norman, Oklahoma

1982

DESIGN MODIFICATIONS IN MICROPROCESSORS

TO SIMPLIFY THEIR TESTING

APPROVED FOR THE SCHOOL OF ELECTRICAL ENGINEERING

AND COMPUTER SCIENCE

By

Jon G. Bredeson, Chairman

Adel A. Ally

John Y. Cheung

Seun K. Kahng

William L. Kuriger

Andy R. Magid

ABSTRACT

Design modifications in microprocessors are recommended to simplify the task of testing them in the user's environment. The recommended design modifications are two additional instructions that can easily be incorporated into the architecture of any currently manufactured microprocessor. A test method is then given that makes use of these instructions and also utilizes some of the principles of Thatte and Abraham (given in their recently published paper). It is shown that the present test method is an improvement over the test method of Thatte and Abraham in at least three ways: the test generation procedures are simpler to use, the fault model allows the existence of larger number of faults, and the testing time is reduced significantly. To illustrate the application of the method, it is used to generate test sequences for an Intel 8085 microprocessor.

CONTENTS

CHAPTER I

INTRODUCTION

With their low cost and numerous potential applications, microprocessors have revolutionized the design of digital systems. This widespread application of microprocessors has prompted the industry to build more and more sophisticated microprocessor devices on a single chip. Sixteen bit microprocessors are already available in the commercial market and research is being done for building parallel processing devices also on a single chip.

The reliable manufacture of sophisticated microprocessors on the commercial scale requires the existence of satisfactory test methods. The methods currently being used in industry are based on ad hoc techniques. These techniques include testing various modules (such as the arithmetic logic unit, or indexing hardware, etc.) separately, testing each microprocessor instruction separately, or running application programs. Based on these techniques most manufacturers test microprocessors using special programs designed to test their individual products. Their techniques do not rely on any systematic or general test methods that could be extended to all types of microprocessors.

This current state of the art for microprocessor testing points to the need for developing more general test methods that can be extended to microprocessors with all types of architectures. Starting at a conceptual level and knowing the architecture of the microprocessor

one should be able to generate the necessary tests to test the microprocessor. Not only should these methods be able to generate tests for the microprocessors of today, but they must be capable of extension for testing the chips of the future. It is expected that the number of gates on the future Very Large Scale Integrated (VLSI) chips will grow over a million within the next few years.

A major step forward in this direction has been taken by Thatte and Abraham in their recently published paper [1]. The authors describe a method that is suitable for testing microprocessors in a user's environment. Starting at the conceptual level and by utilizing the data that is normally available in manufacturers' catalogs, one follows a systematic procedure which results in a test sequence that is capable of testing the given microprocessor. The test sequence consists of instructions that are executed by the microprocessor under test. The operands used in these instructions are chosen carefully. The response of the microprocessor is monitored by an external tester which compares the observed response to the expected response. Any deviation from the expected response is interpreted as a failure.

The basic strategy used by Thatte and Abraham was to classify the microprocessor functions into various functional categories and then to consider faults in each of the categories. The various microprocessor functions considered are: instruction decoding function, register decoding function, data transfer function, data storage function and data manipulation function. The tests generated by Thatte and Abraham methods assume that the fault can exist in only one of the functions. If more than one microprocessor function is faulty then some of the faults can be masked and may go undetected.

The paper by Thatte and Abraham is a major step forward in systematizing the procedure for testing microprocessors with varied architectures. Their method provides a general framework for future work to develop more refined methods. Their methods can also serve as guidelines for designing microprocessors from the point of view of testability. These methods are especially useful in a user environment because the tests can be generated from the data normally available to the user. However, the methods do suffer from a few drawbacks. As already mentioned, faults are allowed only in one microprocessor function. Some test sequences can be very lengthy and the test generation may involve a detailed knowledge of the microprocessor functions.

The present research effort was originally undertaken to remove the drawbacks of the methods of Thatte and Abraham and develop test generation methods that will be simpler to use, will reduce the testing time and help in the location of faults to a certain degree. Subsequent investigation of the problem showed that some of these improvements can be accomplished by making a very small number of design changes in the current designs of microprocessors. The design changes suggested in this dissertation are two additional instructions that can easily be incorporated into the design of any microprocessor. An improved methodology for the test generation is then given using these instructions and some of the principles of Thatte and Abraham. The test method is applied to an Intel 8085 microprocessor and it is shown that the design changes considerably cut down the testing task by simplifying the test generation procedures and reducing the length of the sequences.

The dissertation is organized in the following manner. Chapter II contains an exhaustive literature review of all the previous work done for testing microprocessors as well as other digital circuits. Chapter III gives the background material relating to the methods of Thatte and Abraham. Chapter IV gives the description of the design changes and the test method. In Chapter V, the test methodology is applied to an Intel 8085 microprocessor in presence of the suggested design changes. Chapter VI contains conclusions and recommendations for further research.

CHAPTER II

LITERATURE REVIEW

2.1 Introduction

The concept of designing digital systems in order to simplify testing is fairly old [2]. However, microprocessors have entered the commercial market more recently, and no effective techniques for designing easily testable microprocessors exist. In fact, not many systematic methods exist to test microprocessors of all types of architectures. Most manufacturers test microprocessors using special programs developed to test their individual products. To gain a proper perspective on how to design them from the point of view of testability, it is useful to be familiar with not only the test methods for microprocessors but also with the test methods for VLSI circuits and the test methods for digital systems in general. A thorough literature search was undertaken to accomplish this goal. The purpose of the following presentation is two-fold: first, to lay down in perspective the various fault detection techniques developed to date for testing digital systems; second, to discuss the related literature that resulted in the development of the approach taken in this dissertation. Various methods for testing digital circuits will be examined.

2.2 Fault Table Method

Fault table method is a simple but useful method of arriving at a minimal number of tests to test an n-input combinational circuit [3].

It is obvious that if the truth table for an n-input combinational circuit is known, the circuit can always be tested by providing all the possible 2^n inputs and comparing the actual output against the output indicated by the truth table. However, if an assumption is made that the given circuit contains only a single stuck fault, the same circuit can be tested by a much simpler test set for most practical circuits [3]. The fault table method makes use of this assumption. Corresponding to the given circuit, a table containing 2^n rows and 2m columns is constructed. Each row corresponds to a possible input to this circuit. Whereas, the columns correspond to stuck-at faults at the m-modes of this circuit. Since each node can have either a stuck-at-0 or a stuck-at-1 fault, the total number of columns is 2m. The table is now completed either row wise or column wise. If a given fault causes an incorrect output to be produced for a given input, a check is placed at the corresponding intersection of the row and column. The minimal test set can now be selected from this table, using one of the methods described in [4].

2.3 One Dimensional Path Sensitizing

Another classical method of fault detection for digital circuits is called One Dimensional Path Sensitizing [5-8]. It will be out of place here to describe the complete method, but a brief description of its application to circuits is given below:

a) A single stuck-at fault is postulated at a known location in the circuit.

b) The postulated fault is propagated to one of the primary outputs via one of the sensitized paths. This is

called the forward trace phase. Implicit in the forward trace phase is the setting up of elemental inputs and outputs determined by their prodecessors and in the limit this determines the primary inputs. This process is termed as the backward trace phase, and the final set of primary inputs constitutes the necessary test configuration for the postulated fault.

The main disadvantage of this method is that the presence of reconvergent fanouts in the given circuit can obscure the determination of the required test configuration. [Reconvergent fanout is defined by Armstrong as [6]: Fanout paths that reconverge again are referred to as "reconverging fanout" paths.] In fact, Schneider [7] has given a network containing reconvergent fanout in which a postulated fault cannot be detected using one dimensional path sensitizing technique.

The problem of reconvergent fanout has been solved in n-dimensional path sensitizing method, which is also known as D-algorithm. D-algorithm is described later in this section.

It is also useful to mention that the one dimensional path sensitizing method works best for combinational circuits although attempts have been made to apply the method to sequential networks. For example, Crook and Blythin [8] have extended the method to include bistable elements along the sensitized path. Galey, Norby, Roth [5] have suggested a way of breaking the feedback loops of sequential circuits to make one dimensional path sensitizing applicable. However, their method does not succeed when the network becomes complicated.

2.4 N-Dimensional Path Sensitizing (D-Algorithm)

A description of N-Dimensional Path Sensitizing can be found in [10-11]. N-Dimensional path sensitizing or D-algorithm as it is commonly called, arose out of Roth's efforts to solve the problem of reconvergent fanout associated with one dimensional path sensitizing [10]. Roth based his D-algorithm on the calculus of D-cubes. A D-cube is a mathematical model of combinational network, not much unlike a truth table, but with an extra symbol D (and its inverse $\bar{D}$). Calculus of D-cubes gives various combinational rules for D-cubes. Similar to the forward trace phase in one dimensional path sensitizing, is the process of finding D-cube of failure. Also, similar to the backward phase trace of one dimensional path sensitizing is the technique applying "consistency operation" to the circuit. The process of finding the D-cube of failure as well as the application of consistency operation is facilitated by the rules of D-calculus.

The D-algorithm as suggested by Roth is applicable only to combinational circuits. Kubo [11] however, has shown that the standard model for sequential networks can be redrawn as a cascaded connection of combinational networks called the developed sequential network. He then has affected a modification to the D-algorithm to produce a Diagnostic Test Sequence (DTS) for the original sequential network.

2.5 State Table Analysis

State Table Analysis is a method that is applicable primarily to sequential circuits. The intention here is to produce a DTS from the original state table of the circuit. The most widely used version of this method is based on a paper by Hennie [12]. His idea was to

derive certain input sequences that can be applied to a sequential circuit in any state and the analysis of the output response identifies the initial starting state. The combinations of these Distinguishing Sequences and certain state transitional sequences form an overall checking sequence that can be applied to the network. Regardless of the initial state, the output sequence follows a predefined pattern if there is no fault.

One major academic problem associated with this technique has been the study of the theoretical bound on the length of the checking sequence. Several authors have succeeded in reducing the previous bounds [13-16].

Another important modification of the method has been due to Kohavi and Lavallee [14]. They have suggested a method for modifying the state table M of a machine which does not possess a distinguishing sequence. The modified state table then possess the distinguishing sequence. This diagnosis requirement can then be incorporated in the machine as a design parameter.

2.6 The Boolean Difference Method

The Boolean difference (or derivative) $\frac{df}{dx}$ of a switching function f(x) with respect to a variable x is defined as:

$$\frac{df}{dx} = f(x) \oplus f(x')$$

The above definition of Boolean derivative has been used by Amar and Condulmari [17] and Sellers, et al.[18] to arrive at required tests for combinational circuits. Its main advantage is that most of the work done in deriving the sequence is a straightforward Boolean

expression minimization and there are excellent algorithms for this. Marinos [19] has extended this technique to sequential networks. Carroll, et al. [20] and Kajitani [21] have also used Boolean difference for arriving at the diagnostic test sequences.

2.7 Testing of Large Scale Integrated (LSI) Circuits

The earlier approaches to develop tests for LSIs attempted to extend the classical techniques described above to LSI testing. For example, Lewis [22] proposed an algorithm based on path sensitizing techniques that produces acceptance tests for certain types of chips. Also, a paper by Jones and Mayes [23] combines the elements of partitioning and State Table approaches to produce an integrated Diagnostic Test Sequence procedure applicable to either combinational or sequential networks. Hillman [24] has also described a programmable test system for LSI chips.

However, the problems of LSI testing are manifold and the paper by De Atley [25] contains an appraisal of these. Extension of classical techniques to LSI testing is clearly not possible. As the complexity of LSI chips grows, their functional testing is receiving more and more attention. Two LSI (or VLSI) chips that are in wide use today are memories and microprocessors. In the remaining part of this section, literature relating to the testing of these two types of VLSI chips is examined.

Many excellent papers have been published relating to the testing of memories [26-36]. Several algorithms to detect different classes of faults are given in [30-36]. A major problem related to fault diagnosis is that related to the pattern sensitive faults [30-32]. The complexity of the problem of attempting to detect unrestricted pattern

sensitive faults in large semi-conducter random access read/write memories has been adequately pointed out in [27]. However, recently, Reddy and Suk [36] have developed a restricted model for a class of pattern sensitive faults.

Perhaps the most complex problem related to the testing of a LSI (and VLSI) chips is the testing of microprocessors. As mentioned earlier, the methods in industry use ad hoc techniques, such as "testing" each instruction for many operands, "exercising" various modules in the microprocessor, or running application programs [37-39]. These techniques do not rely on any systematic or general test method that could be extended to all microprocessors.

As the interest in VLSI chips continues to grow, their testing is receiving increased attention. References [40-46] contain very recent literature published in this area. These range from papers on test equipment to functional level testing of digital chips and their design for testability. For example in [40] a unique diagnostic test chip and apparatus for testing microprocessors is described. In [41], the techniques to model LSI devices at a functional level using General Simulation Program (GSP) are described. Su and Hsieh have given a method for modeling the behavior of digited systems using the register transfer language [42]. However their method is not easily applicable to complex chips such as microprocessors. An approach for high speed testing of circuit boards is given in [43]. The paper presents the design rationale and technical approach used in the development of a high speed functional board test system that is targeted for testing and isolating at the native operating speed of the circuit board under test. The problem of designing microprocessors has been addressed by Buckroyd in [44]. The author assesses the problem and offers general guidelines. However no specific method is described.

A model suitable for testing the control units of digital computers has been proposed by Robach and Saucier [45]. It overcomes some of the difficulties in the classical methods because instead of testing the gates and flip flops separately it considers the control commands and the control states as the basic test elements. It thus uses the concept of functional testing. Their approach, therefore is suitable for developing test methods for microprocessors.

Using the approach of functional testing, a methodology for test generation for a "typical" microprocessor has also been given by Thatte and Abraham [46]. They have later modified their method in a recently published paper [1] to extend the method to microprocessors with all architectures. It is felt that the approach suggested by Thatte and Abraham is an excellent way of testing microprocessors with varied architectures. Their approach is also suitable for testing microprocessors in the user environment. Further, it is not restricted to stuck-at faults only but can also detect certain other faults such as those resulting from metalization, etc. However, all possible types of faults are not covered in their model. Multiple faults may exist in several types of microprocessor functions, but their model assumes that only one function may be faulty at a given time. Furthermore, some of the test sequences can be very lengthy. This not only takes a longer time to test the microprocessor, but also makes the task of generation of these test sequences more cumbersome. This dissertation resulted from the need to extend the work done by Thatte and Abraham. It is shown that the test generation procedures can be systematized and greatly simplified by making very few simple modifications in the current designs of microprocessors. The design modifications suggested are few, simple, easy to implement and are applicable to microprocessors with all architectures. They reduce the test task con-

siderably by simplifying the test generation prodedures, and cutting down the testing time.

CHAPTER III

BACKGROUND

3.1 Introduction

The motivation behind the solution presented in this dissertation becomes obvious once the test methods of Thatte and Abraham are known. Their test methods and the drawbacks in their test methods are briefly discussed in the following sections. However, the discussion presented here is by no means complete. It is provided only to give a proper perspective on the proposed test methodology given later in Chapter IV. For a complete description of their test methods the reader is referred to [42].

3.2 Test Generation Procedures of Thatte and Abraham

Thatte and Abraham [1] have developed methods which are suitable for testing microprocessors in the user environment. From the product information available to the user (such as knowledge of the instruction repertoire of the microprocessor and its various registers), one develops the required tests. These tests actually consist of instruction sequences which are executed by the microprocessors under test. The operands used for these instructions are chosen carefully. The response of the microprocessor under test, is monitored by an external tester, which compares the observed response to the expected response. Any deviation from the expected response is interpreted as a failure.

To generate the required test sequences, Thatte and Abraham have divided the faults in a microprocessor into the following five catagories: Instruction Decoding Faults, Register Decoding Faults, Data Transfer Faults, Data Storage Faults and Data Manipulation Faults. These fault catagories are briefly described below.

a) Instruction Decoding and Faults in the Control Function Faults: These faults refer to the faults in the instruction decoding and control function of the microprocessor. Since this function is carried out by the decoder and control circuitry in the microprocessor, faults in this function will also indicate faults in the instruction control and decoder circuitry. Under these types of faults, incorrect instructions may get executed. For example, if the microprocessor is required to execute an instruction I_j, under a fault instruction I_k may be executed.

b) Register Decoding Faults: These faults refer to the register decoding function in the microprocessor. Since this function is also carried out by the decoding circuitry in the microprocessor, faults in this function will also indicate faults in the register decoding circuitry. Under these faults, data may be loaded in or read out of the wrong registers, in response to the execution of an instruction. For example, if the contents of register R_1 are required to be read, the ANDed or ORed contents of registers R_1 and R_2 may be read instead.

c) Data Transfer Faults: These faults refer to the faults in the microprocessor buses. The lines of a bus may get

connected (bridged) together due to metallization of some other form of coupling. The connected lines cannot transmit different logic values. Thus, the bus can transmit different logic values on the two bridged lines. In the same manner stuck-at faults can cause incorrect data values to be transmitted.

d) Data Storage Faults: These faults refer to the faults in the microprocessor registers. Certain cells in the microprocessor may be stuck-at-1 or stuck-at-0 and this may cause incorrect data to be stored in the registers.

e) Data Manipulation Faults: These refer mainly to the faults in the arithmetic logic unit of the microprocessor. Faults in the interrupt handling hardware, hardware used for incrementing (or decrementing) the stack pointer, etc., are also included in these.

3.2.1 Notation and Test Generation Procedures for Instruction Decoding Faults

The test model of Thatte and Abraham assumes that the Instruction Decoder is realized without any reconvergent fanout. Further, the decoder is allowed to have only a single stuck-at-fault. Under the presence of such faults, when a microprocessor is required to execute a valid instruction I_j, any one of the following can happen.

(i) Instead of the instruction I_j, some other instruction I_k is executed. This fault is denoted by $f(I_j/I_k)$.

(ii) In addition to instruction I_j, some other instruction I_k is also activated. This fault is denoted by $f(I_j/I_j + I_k)$.

(iii) No instruction is executed. This fault is denoted by $f(I_j/\phi)$.

Now assume that the tests are required to be generated for a fault denoted by $f(I_j/I_k)$ and the instruction I_j is such that it transfers the contents from its source register $S(I_j)$ to its destination register $D(I_j)$. Also assume that instruction I_k is a transfer instruction which transfers the contents of its source register $S(I_k)$ into its destination register $D(I_k)$. The procedure described below generates the required test sequences.

Procedure

(i) Generate instructions for storing operand 1 (operand 2 may be 000...000 or 111...111 or 101...010 or any other predetermined operand) in $S(I_j)$.

(ii) Generate instructions for storing operand 2 (operand 2 is also a predetermined operand such that operand 1 $\neq$ operand 2) in $S(I_k)$.

(iii) I_j is executed.

(iv) Generate instructions to read $D(I_j)$. $D(I_j)$ is read. Expected output is operand 1.

(v) I_k is executed.

(vi) Generate instructions to read $D(I_k)$. $D(I_k)$ is read. Expected output is operand 2.

It is clear that if the output in step (iv) is operand 2 or in step (vi) is operand 1, fault (or faults) will be indicated.

The procedure above was discussed by the way of an example. Detailed procedures and their proofs are given in [42]. However,

it should be clear that test procedures for detecting faults such as $f(I_j/I_j + I_k)$ and $f(I_j/\phi)$ are very similar.

3.2.2 Fault Masking of Instruction Decoding Faults and the Order of Test Application

In order to ensure that a given microprocessor has been tested for all the possible Instruction Decoding Faults, tests must be applied in a specific order. If not, some Instruction Decoding Faults can go undetected. The need to avoid fault masking also makes some test sequences more complicated.

Thatte and Abraham have approached the problem of fault masking by dividing the total microprocessor instructions into three categories: Class T instructions, Class M instructions and Class B instructions. Basically, Class T instructions are those instructions that carry out some sort of data transfer within the microprocessor; Class M instructions are the instructions which perform the function of data manipulation in a microprocessor; whereas Class B refers to branch instructions. Further, a labeling algorithm assigns appropriate labels to all the instructions in the microprocessor (regardless of their class). In general, a label of an instruction refers to the ease with which data can be transferred from the destination register (of the instruction in question) to the outside world. Instructions that directly transfer data to the devices (such as memory) in the outside world are assigned to label 1. Instructions that call for the execution of one or more additional instructions to transfer the data from their destination registers to the outside world are assigned higher labels. As an example consider the instruction ADD R, also of the Intel 8085. This instruction adds the contents of the register R to the Accumulator and

stores the results in the Accumulator, i.e., the destination register of this instruction is the Accumulator. Transferring the contents of the Accumulator to the memory requires the execution of one additional instruction. ADD R is therefore assigned a label 2.

In order to detect instruction decoding faults in a microprocessor, the instructions are classified and labeled in the above manner. Next, each instruction I_i must be tested for faults $f(I_i/I_j)$, $f(I_i/I_j + I_k)$ and $f(I_i/\phi)$. The total task of testing these faults is divided into subtasks. Each subtask consists of testing a portion of the faults depending upon the labeling and classification of I_i and I_j. The number of subtasks into which the instruction decoding faults are divided is a function of the distribution of the labels within the instruction repertoire of the microprocessor. Corresponding to each subtask, there exists a well-defined test procedure. The subtasks are carried out in an order that depends upon the classification and labeling of faults to which subtask is applicable. The execution of tasks in the specified order ensures that fault masking will not occur.

Given below is an example of application of one of the procedures of Thatte and Abraham to a hypothetical microprocessor. The example will illustrate two important points: first, that the order of application of various test sequences is important if fault masking is to be avoided; second, the test generation procedures can become complicated while attempting to avoid fault masking.

Example 3.2.1: Consider a microprocessor with the following instructions (among others) in its instruction repertoire.

I_1 -- Complement the Accumulator

I_2 -- Subtract the contents of the memory location LOC X (next to the one storing the OP CODE of Instruction I_2) from the contents of the Accumulator and store the result in the Accumulator.

I_3 -- Store Accumulator in memory using direct addressing.

Assume that tests are to be generated for the fault $f(I_1/I_2)$. It is clear that both I_1 and I_2 are class M instructions whereas I_3 is a class T instruction. Also, I_3 must be assigned a label 1 whereas I_1 and I_2 must both be assigned a label 2. An appropriate test procedure selected from [42], consists of the following steps.

Step 1: Store proper operands in $S(I_1)$ and $S(I_2)$ such that when I_1 is executed RESULT 1 is produced and when I_2 is executed RESULT 2 is produced in the Accumulator, and RESULT 1 $\neq$ RESULT 2. (Note that in this case $S(I_1)$ is the Accumulator and $S(I_2)$ = LOC X in the memory.)

Step 2: Read out the Accumulator by executing I_3 to ensure that the Accumulator does in fact contain the operands that were to be stored in it in Step 1.

Step 3: Repeat Step 2.

Step 4: Execute I_1.

Step 5: Read Accumulator by executing I_3.

The above procedure detects the faults of the type $f(I_1/I_2)$ provided faults of the type $f(I_3/I_q)$ do not exist in the given microprocessor. (I_q is an instruction in the microprocessor repertoire) The procedure therefore assumes that the microprocessor has already been tested for $f(I_3/I_q)$.

Step 2 is necessary to ensure that the Accumulator contains the proper operand to be stored in it in Step 1, otherwise I_2 could produce RESULT 1 instead of RESULT 2 and fault masking could occur.

Step 3 is necessary to ensure against fault masking due to faults such as $f(I_3/I_q + I_q)$ where D(I) = Accumulator. In presence of this

fault, the contents of the Accumulator can change after the execution of I_3 (in Step 2), they will not change after the second execution of I_3 (in Step 3) either.

In step 4, I_1 is executed. If $f(I_1/I_2)$ is present, RESULT 2 will be produced in the Accumulator in Step 4, and it will be detected in Step 5.

3.2.3 Faults in the Register Decoding Function

Register decoding refers to the task of decoding the address of a register which may be stored as a specific bit pattern in the instructions involving that register. Under such a fault, all the instructions of the microprocessor which involve a register R_i will now manipulate the contents of set of registers $\{R_1, R_2, \ldots, R_j\}$. This set:

(i) May include only R_i. In this case, no fault is indicated.

(ii) May be empty. This fault is denoted by $f_d\{R_i\} = (\phi)$.

(iii) May consist of set $\{R_1, R_2, \ldots, R_j\}$ which may or may not include R_i. This fault is denoted by $f_d(R_i) = \{R_1, R_2, \ldots, R_j\}$.

Thatte and Abraham again give detailed methods with proofs for the detection of these faults. An example will be sufficient here. Consider a fault denoted by $f_d(R_2) = (R_1, R_2)$. To detect the fault, R_1 is written with operand 1 and subsequent to that R_2 is written with operand 2. R_1 and R_2 are now read. Under the fault, register R_1 will read operand 2 instead of operand 1.

Although this scheme looks simple at first, the writing and reading of operands into various registers of a microprocessor may not be so straightforward. Consider a register R_j in a microprocessor which saves

the contents of the address of a program sequence when a 'jump to subroutine' instruction is executed. Also let

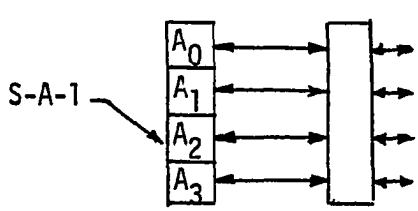
I_j = jump to subroutine instruction

LOC = address of I_j in the main program

When I_j is executed, the updated contents (LOC + 1) of the program counter will be saved in R_j. Hence, to load a given operand 1 (say) in R_j the address LOC of I_j in the main program must be chosen such that LOC + 1 = operand 1.

3.2.4 Test Generation Procedure for Data Storage Faults

These faults can be easily detected by writing in and reading out various data in the microprocessor registers. Consider the 4-bit register shown in Figure 3.1. One of its cells is stuck-at-1. Suppose a data value of 1100 is written into the faulty register, on reading out the same register a value of 1110 would be read out, as is also indicated in the same figure. This also illustrates the principle that data storage faults can be detected by writing and reading different values of data into the microprocessor registers. This principle is used to arrive at the design modifications given in Chapter IV.



Write	Read
1	1
1	1
0	1
0	0

Figure 3.1 A Four-Bit Register with One Cell Stuck-at-1.

3.2.5 Test Generation for Data Transfer Faults

Data transfer faults can be detected in a manner similar to the data storage faults. Figure 3.2 shows two lines of a 4-line bus, coupled (bridged) together. As shown in the figure, if a data value of 1100 is transmitted at the left end of the line then a value of 1110 will be received at the right hand end. This again shows that faults in the microprocessor buses can be detected by transmitting properly chosen values of data over the faulty bus. This principle was again used to arrive at a design modification given in Chapter IV.

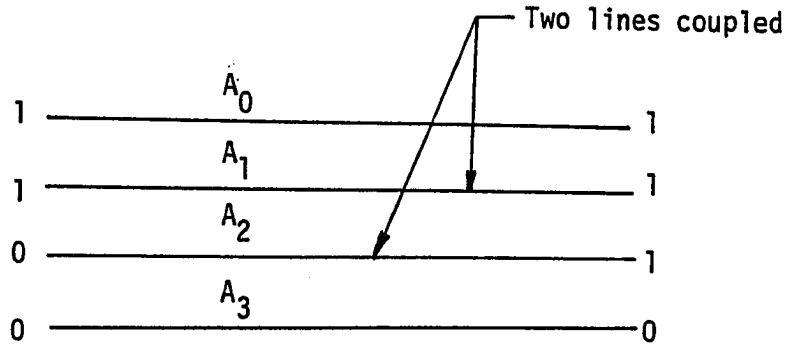


Figure 3.2 A Four-Line Bus Having Two of its Lines Coupled.

3.2.6 Test Generation for Data Manipulation Faults

If a logic level description of functional units is available, test sets can be generated for them using classical fault detection algorithms based on stuck-at-fault model.

3.3 Length of the Test Sequences

Thatte and Abraham have obtained estimates for the lengths of test sequences required to test various types of microprocessor faults. For example, it can be shown that the number of instructions in the test sequence to detect the register decoding faults is of the order of n_R^3 (where n_R denotes the total number of registers in the microprocessor). The length of the test sequences to detect Data Storage or Data Transfer Faults depends upon the widths of the microprocessor buses and registers; whereas the length of the sequences to detect data manipulation functions depends upon the nature of the operation performed by "o" in instructions of the type "R_io R_j" → "R_k".

It is of special interest to obtain an estimate of the length of test sequences to detect Instruction Decoding Faults since it will be shown that these test sequences constitute a major portion of the total test set. Assume that a given microprocessor is to be tested for all faults of the type:

$$f(I_i/I_j) \quad \text{for} \quad 1 \leq i \leq n_I$$
$$1 \leq j \leq n_I$$

where

n_I = total number of instructions in the instruction repertoire of the microprocessor under test.

It is obvious that the possible number of faults of the type $f(I_i/I_j)$ in a given microprocessor is n_I^2. The total number of instructions that would be needed to test a given fault $f(I_i/I_j)$ will depend upon the classification and labels of I_i and I_j. (These can be generated using methods given in [47]). It may be concluded that the length of

the instruction sequence to test for all possible Instruction Decoding Faults must be of the order of n_I^2. In the same way, it can also be shown that the length of the instruction sequence to detect all faults of the type $f(I_i/I_i + I_j)$ must also be of the order of n_I^2.

For today's microprocessor n_R typically ranges from 4 to 32, while n_I ranges from 30 to 512. Therefore usually the test sequences to detect Instruction Decoding Faults form the dominant portion of the complete test sequence for the microprocessor under test.

3.4 Limitations of the Test Methods of Thatte and Abraham

Some limitations of the Thatte-Abraham methods are given below.

1. Multiple faults are not considered. In their test model it is assumed that only one function can be faulty at a time, although any number of faults may exist in the faulty function. This is a serious limitation since in practice faults occur at random and can exist in more than one function.
2. Methods of Thatte and Abraham do not locate the faults. By their methods it is not possible to tell which component in the microprocessor is faulty.
3. Errors in the control sequence are not considered.
4. Some test sequences generated by the methods of Thatte and Abraham can be very lengthy. Consider for example, the Intel 8085 microprocessor. The total number of instructions in its instruction repertoire is 248. The order of the test sequences to locate Instruction Decoding Faults must be of the order of $(248)^2$. This is a large number and the task of generating the test sequence will be quite cumbersome.

3.5 On the Order of Test Sequences to Test the Microprocessor for Instruction Decoding Faults

It is of interest to compare the number of instructions required to test a microprocessor by the methods of Thatte and Abraham to the number of instructions required when using the present method. It was shown earlier that the number of instructions required to test the microprocessor for instruction decoding faults is of the order of n_I^2 - when the methods of Thatte and Abraham are used. It is shown below that this number is reduced to the order of n_I by the present method.

To test for instruction decoding faults in each group of EPI instructions, each instruction by an MVMI instruction and followed by an MVMO instruction, i.e. the total number of instructions required to test instruction decoding faults in the i^{th} group of EPI instructions is $3n_{Gi}$. It follows that the total number of instructions to test for all possible faults in the m EPI instruction groups is:

$$= 3n_{G1} + 3n_{G2} + 3n_{G3} + \quad 3n_{Gm}$$

$$= 3(n_{G1} + n_{G2} + \quad n_{Gm})$$

$$= 3n_I$$

where n_I = total number of instructions in the instruction repertoire of the microprocessor

i.e. the number of instructions is now of the order of n_I.

CHAPTER IV

DESIGN MODIFICATIONS, FAULT MODEL AND THE TEST METHODOLOGY

4.1 Introduction

As was discussed in the previous chapter, the test sequences generated by the methods of Thatte and Abraham can be lengthy because of two reasons. First, some registers in most commercially available microprocessors cannot be read out (or read in) explicitly. These registers can only be read out (or read in) implicitly by using sets of instructions rather than only one instruction. There is another reason why the test sequences generated by Thatte and Abraham are lengthy. In order to test faults of the type $f(I_j/I_k)$ or $f(I_j/I_j + I_k)$, test sequences must be carefully generated in order to avoid fault masking (the concept of fault masking was described in section 3.2.2). As a result the total number of instructions to test the above mentioned faults by the methods of Thatte and Abraham is of the order of n_I^2 (where n_I = total number of instructions in the given microprocessor). This is a large number and since the set of instructions to test the instruction decoding faults forms a large portion of the total test set, the total test set must also be large.

It follows that the number of instructions in the testing sequence can be reduced significantly by reducing the instructions to test the instruction decoding faults and by providing means to read in or write out microprocessor registers explicitly. Both these objectives can be accomplished by introducing two new instructions in the currently manufactured

microprocessors. We will term these instructions as MOVE MULTIPLE instructions. A description of these instructions is given below.

4.2 MOVE MULTIPLE Instructions and Their Application to Simplify the Testing of Microprocessors

Most currently manufactured microprocessors do not use all the possible available bit patterns in the instruction register to implement the instructions. For example, in the Intel 8085 microprocessor only 246 bit patterns out of the available 256 bit pattern combinations are used to implement distinct instructions. Two of the remaining 10 bit pattern combinations can be used to implement the MOVE MULTIPLE instructions. Even in microprocessors that use all the bit pattern combinations, we can still introduce the MOVE MULTIPLE instructions by removing instructions that do not have extensive use.

Consider the Intel 8085. The bit patterns corresponding to the following hexadecimal numbers are unused: 08, 10, 18, 28, 38, CB, D9, DD, ED, FD. Two of these bit patterns 08 and 10 can be used to implement the instructions whose description is given below.

MVMI (MOVE MULTIPLE IN)

This is a three byte instruction that inputs the contents of the consecutive memory locations of the external memory, into the various microprocessor registers. The registers involved are: all the work registers A through L, the stack pointer (SP), the program counter (PC), status register (SR), and the instruction register (IR). The contents of the memory location pointed to by the address byte of MVMI are transferred to Register

A (accumulator). The contents transferred into register B are from a memory location having an address LOC + 1 (LOC = address stored in the address byte of the instruction MVMI). Contents transferred in registers C, D, E, H, L, SP, PC, SR and IR are from locations with addresses LOC + 2, LOC + 3, LOC + 5, ..., LOC + 12.

OP Code	Operand
MVMI	dbl. addr.

0 0 0 0 1 0 0 0
low-order addr.
high-order addr.

Cycles: 16
States: 49
Addressing: Indirect
Flags: None

It is anticipated that the execution of the above instruction on Intel 8085 will require 49 clock periods. This is arrived at using the following considerations. In Intel 8085, every machine cycle involving memory reference must take 3 clock periods, except for the instruction fetch cycle. Instruction fetch takes 4 clock periods to complete. Since MVMI requires 16 memory reference operations, it can be executed in $4 + 15 \times 3 = 49$ clock periods.

It is also clear that this instruction will involve 16 machine cycles, one for each memory reference operation.

Example 4.2.1

Assume that at a certain stage in the program execution, the microprocessor registers and the external memory have the stored contents as shown in Figure 4.1.

Memory Word	Address of the Memory Word (Hexadecimal)
9C	40
0A	41
30	42
40	43
31	44
BD	45
F1	46
10	47
2B	48
C3	49
B1	50
AC	51
AB	52

(a)

ACCUMULATOR 2A	
INSTRUCTION REGISTER 92	
STATUS REGISTER 95	

B D5	C OF
D FF	E 01
H 5C	L 36
STACK POINTER 3CCB	
PROGRAM COUNTER 9C89	

(b)

Figure 4.1 Contents of Various Microprocessor Registers and Memory Locations Before the Move Multiple Instructions are Executed.

We execute the 3-byte MVMI instruction with contents in its three bytes as given below:

MVMI	08
low-order addr.	00
high-order addr.	04

After the execution of this instruction, the contents stored in various registers will be as follows:

ACCUMULATOR 9C	
INSTRUCTION REGISTER AB	
STATUS REGISTER AC	

B 0A	C 30
D 40	E 31
H BD	L F1
STACK POINTER 102B	
PROGRAM COUNTER C3B1	

MVMO (MOVE MULTIPLE OUT)

This instruction performs the function opposite to that of MVMI. It transfers the contents of various microprocessor registers into consecutive memory locations in the external memory. The microprocessor registers involved are: all the microprocessor registers A through E plus the status register. The contents of the accumulator A are transferred to the memory location addressed by the address byte of MVMO. Contents of registers B, C, D, E, H, L, PC, SR and IR are transferred to locations LOC, LOC + 1, ..., LOC + 12. (LOC is the address stored in the address bytes of the MVMO instruction.)

OP Code	Operand
MVMO	dbl.addr.

0 0 0 0 1 0 1 0
low-order addr.
high-order addr.

Cycles: 16
States: 49
Addressing: Indirect
Flags: None

As in the case of the instruction MVMI, we anticipate that MVMO will also require 49 clock periods. This is a reasonable execution time to assume, since the instruction requires 16 memory reference operations. Also, we assume that the instruction has 16 machine cycles, one cycle for each memory reference operation.

Example 4.2.2

Again assume that the microprocessor registers and the external memory have the contents shown in Figures 4.1 (a) and (b) of Example 4.2.1. Assume further that the instruction MVMO is now executed, with the three instruction bytes having the following contents:

MVMO	10
low order addr	00
high order addr	04

After the execution of MVMO, the external memory will have the following contents stored in it:

Memory Word	Address of the Memory Word (Hexadecimal)
2A	40
D5	41
0F	42
FF	43
01	44
5C	45
36	46
3C	47
CB	48
9C	49
89	50
95	51
92	52

4.4.2 Use of the 'MOVE MULTIPLE' Instructions to Detect Data Storage and Data Transfer Faults

Since data storage faults refer to the stuck-at faults within the microprocessor registers, these faults can be easily detected by writing in and reading out various data in the microprocessor registers. Similarly, data transfer faults can be detected by transmitting different values of data over the microprocessor bus. It follows that if the microprocessor

has only one bus then the faults in that bus can be detected by transmitting all possible values of data over that bus. Also, the faults in the various microprocessor registers can be detected by writing in and reading out all the possible values of data in each of these registers. Therefore, the following test procedure tests all the Data Transfer and Data Storage faults.

Test Algorithm 4.2.1

1. Let K = 0
2. MVMI LOC 1/ Execute this instruction with the binary value of K stored in each of the memory locations LOC 1 through LOC L + 12/
3. MVMO LOC 2.
4. Increment K.
5. If K = 2^8, Stop
6. If K < 2^8, Go to 1.

Since the procedure above reads in and writes out every possible bit pattern in each register of the microprocessor, it is obvious that it tests all the data storage faults. Also, since all the possible bit patterns are also transmitted over the bus, the procedure also tests all the data transfer faults. It is assumed that the 'MOVE MULTIPLE' instructions that appear in the above test procedure have been checked and found to be fault free.

4.2.2 Use of the 'MOVE MULTIPLE' Instructions to Detect Instruction Decoding Faults

In general, any fault of the type $f(I_1/I_j)$ can be detected by executing the following test sequence and monitoring the signals produced at the external microprocessor pins.

Test Sequence 4.2.1

1. MVMI LOC/ Memory locations LOC through LOC + 12 contain data that is chosen, using the methods explained in Appendix A./
2. I_1
3. MVMO LOC

It is assumed that data input by the above MVMI instruction can be chosen such that no two instructions operating on this data produce identical results in the microprocessor registers. It is easy to see why the above procedure detects all faults of the type $f(I_i/I_j)$. If instead of instruction I_i any other instruction I_j is executed in Step 2, it will produce different results in microprocessor registers than those that would be produced by I_i. This data is output in Step 3 of the above sequence by instruction MVMO and stored in memory location LOC 1 through LOC 1 + 12. When this data is compared to the reference expected output data, any fault of the type $f(I_i/I_j)$ would be detected.

This principle is used later in Section 4.4.1 to develop a comprehensive test generation procedure for instruction decoding faults.

4.3 Fault Model

Before we describe complete testing procedures used in testing the microprocessors it is necessary to develop a fault model for microprocessors. The goal is to provide a sound theoretical footing for our fault detection techniques. Further, if the fault detection methods are to be useful to microprocessor users, the fault model should be applicable to various commercially available microprocessors and yet be independent of their implementation details. The design of microprocessors does not easily lend itself to any theoretical analysis. The features found in

commercially available microprocessors have evolved to fulfill the needs of the applications for which microprocessors are used. This makes the task of developing a comprehensive fault model very complicated. It is clear that generalizations and assumptions would have to be made. Two types of assumptions are necessary to develop the fault model: 1) assumptions regarding the architectures of the microprocessors, and 2) assumptions regarding the types of faults to be allowed in these microprocessors. The assumptions should fit the real world as closely as possible and should still simplify the testing.

4.3.1 Assumptions and Definitions for Developing the Fault Model

Assumption 4.3.1: To execute a valid instruction, every microprocessor must put out signals at its external pins.

Since a microprocessor is a general purpose chip it is difficult to form hard and fast rules as to what features are common to all. However, it is reasonable to say that every microprocessor is a programmable device that has its future states determined by a set of instructions stored in the external memory. In order to access these instructions or in order to store or retrieve data from the external memory or in order to interface with the external logic, a microprocessor must output signals on its pins which either indicate the state of the microprocessor or its response to an instruction execution. If a known sequence of instructions is stored in the external memory of a microprocessor then the response of the microprocessor to these instructions can be monitored by an external tester to provide clues to any faults within the microprocessor.

Consider a STA (STORE ACCUMULATOR DIRECT) of an 8085 microprocessor. This instruction transfers the contents of the accumulator to an external

memory locations whose address is a part of the instruction. Since this register is located anywhere in the 64K memory space that the 8085 can directly address, 16 bits are required for the address. Thus the STA instruction contains 3 bytes: a 1-byte OP code and a 2-byte address. The instruction is stored in the memory as follows:

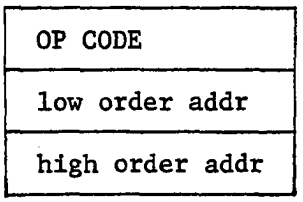


Figure 4.2 The Three Byte STA Instruction.

Three machine cycles are required to fetch this instruction. OPCODE FETCH transfers the OP code from memory to the instruction register. The 2-byte address is then transferred, 1 byte at at time, from memory into a temporary storage within the microprocessor. This calls for two MEMORY READ machine cycles. When the entire instruction is in the microprocessor it gets executed. Execution of this instruction entails data transfer from the microprocessor to the memory.

The CPU timing for the STA instruction is illustrated in Figure 4.3. Each machine cycle is divided by the system clock into a number of state transitions, or T states, which correspond to the period between two negative going transitions of that clock. Each machine cycle for this instruction consists of 3 or 4 states as is also shown in Figure 4.3. Each state

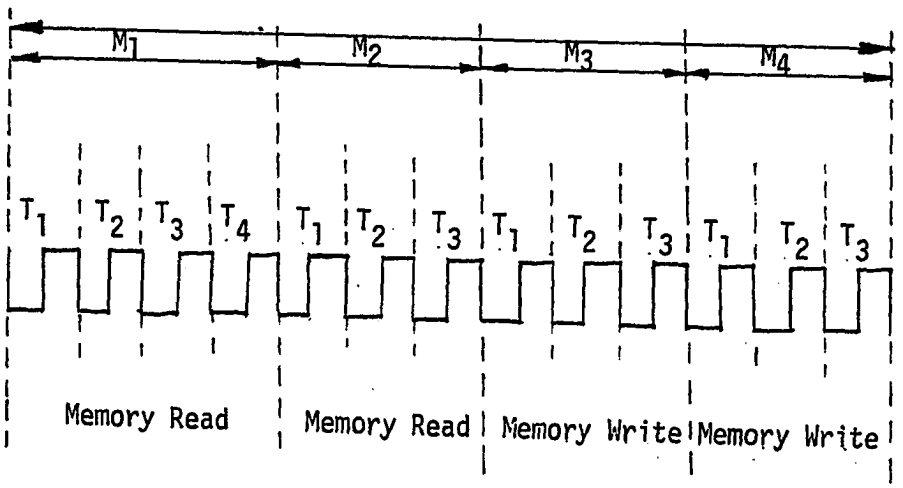


Figure 4.3 CPU Timing for the STA Instruction of an 8085 Microprocessor.

is one clock period in duration. The instruction STA has a total of 13 states.

The timing diagram for an OPCODE FETCH machine cycle is shown in Figure 4.3. As is shown in the figure, at the beginning of state T, the IO/$\bar{M}$, S1, and S0 status and control signal indicate the type of machine cycle which has been initiated. For the OPCODE FETCH, IO/$\bar{M}$ = 0, S1 = 1, and S0 = 1. This status information remains constant for the duration of the machine cycle. The 16-bit address A0-A15 of the memory location containing the OP code is obtained from the program counter and placed in the address and address/data latches. The high order byte of the address appears on the address bus, A8-A15, and remains constant until the end of state T_3. During the state T_4 the data on the address bus is unspecified. The low order byte of the address is placed on the address/data bus at the beginning of T_1. This data, however, remains valid only until the beginning of state T_2 at which time the address/data bus is floated. The address latch enable, ALE, clocks an external register which latches the low order address byte on its falling edge.

During the state T, the $\overline{RD}$ control signal goes low, and the OP code to be fetched is placed on the data bus by the addressed memory location. On the rising edge of the RD control signal in T_3, the OP code obtained from the memory is transferred to the microprocessor's instruction register. During T_4 the 8085 decodes the instruction.

The next machine cycle for this instruction is a MEMORY READ cycle. MEMORY READ cycle is otherwise similar to the OPCODE FETCH cycle except that only the states T_1 to T_3 are used. Also, status signals corresponding to memory read are output at the status pins (i.e., S_1 = 1, S_2 = 0). Also, the address output at the address pins is the address of the

previous byte fetched + 1.

The next machine cycle for the STA instruction is again a MEMORY READ cycle in which the high order byte from the memory is transferred to the microprocessor. The only difference between this MEMORY READ cycle and the previous cycle is that contents output at the address are the previous value + 1.

The next cycle is the MEMORY WRITE cycle and is similar to the previous MEMORY READ cycles except that the WR control, instead of the RD goes low during T_2.

It is thus seen that the execution of STA instruction is accompanied by appearance of predetermined signals at specified time intervals. These signals can be monitored to give indication of the faults within the microprocessor. Any deviation in the appearance of the output at the microprocessor pin indicates a fault.

Similarly, the microprocessor pins can be monitored in response to other instruction execution cycles to detect faults.

Assumption 4.3.2: Any detectable fault within a microprocessor must either result in incorrect data being stored in various microprocessor registers (including registers like program counter, stack pointer, etc.) or must result in incorrect signals being output at various external microprocessor pins.

Most faults in a microprocessor are detectable faults. That is, most faults if present in a microprocessor will alter the normal functioning of a microprocessor. For example, two coupled lines in a microprocessor bus may cause erroneous data to be stored in that register. However, there can exist faults within a microprocessor that do not alter the functioning of the microprocessor in any way. We will term these

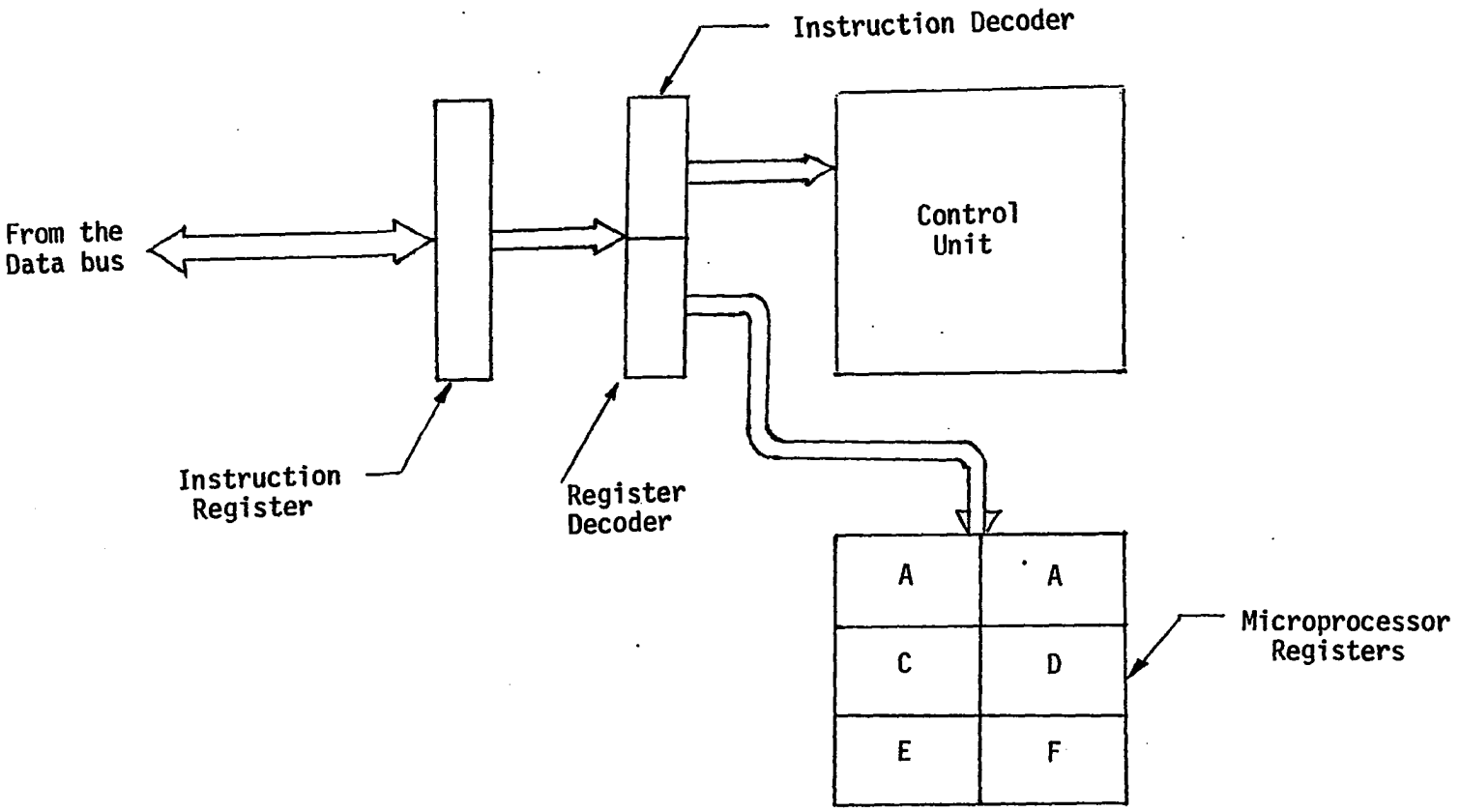


Figure 4.4 Schematic Diagram Showing the Partial Circuitry to Implement Instructions Involved Registers A Through E.

faults as undetectable faults. We will give an example of an undetectable fault.

Consider the schematic diagram shown in Figure 4.4. It shows circuitry to implement some of the instructions that involve the work registers A through E of a hypothetical microprocessor. Suppose this microprocessor has an instruction with the following MNEMOMIC

MOV r, M

Assume further that the OPCODE that realizes this instruction is given by:

0	1	1	1	0	S	S	S

where the values of SSS depend upon which register is involved. Depending upon the value of the last 3 bits in this OPCODE, the appropriate register gets decoded and its contents are stored in memory. Assume further that the binary digits that correspond to each of the registers A through F in Figure 2 are as follows:

A	0 0 0
B	0 0 1
C	0 1 0
D	0 1 1
E	1 0 0
F	1 1 1

This scheme implies that the OPCODE 0 1 1 1 0 0 1 0 transfers the contents of register C to memory, whereas the OPCODE 0 1 1 1 0 1 0 0 transfers the contents of E to memory, and so on.

Assuming that due to a stuck-at fault or faults in the module marked as the register decoder, whenever register C is required to be decoded, register E gets decoded and vice versa. If we further assume that this fault is found in every instruction that the microprocessor executes, and

that the registers C and E are otherwise identical in every respect, the fault will not be detected by executing any instruction. Hence, it is possible for stuck-at faults to be present in a microprocessor but still go undetected. Our testing procedure will only concentrate on detecting the detectable faults.

Further, all the detectable faults in a microprocessor must result in either an incorrect execution of a given instruction or an undesired response to activation of pins. We illustrate below how a detectable fault may result in the storage of incorrect data in various microprocessor registers. Consider the hardware shown in Figure 4.5. The figure shows hardware capable of executing at least two instructions, I_1 and I_2. Assume that I_2 is an instruction that transfers the contents of register A into register C, whereas I_1 transfers the contents of register B in register C.

The figure also illustrates 3 registers, A, B, and C, connected to the system bus via buffers B_A, B_B, B_C. Whenever the lines C_0, C_2, and C_4 are enabled, the contents of the register are output to the system bus, whereas activating the lines C_1, C_3 and C_5 results in data from the system bus being input to the microprocessor registers. Thus the execution of I_1 calls for activating control outputs C_0 and C_5, and the data is clocked onto C. Similarly the execution of I_2 calls for the activation of the signals C_2 and C_5, and the data is clocked into register C. Assume that under a fault in the control unit or the instruction decoder, whenever instruction I_1 is executed, an additional line C_2 is also activated. It is clear that the activation of this additional line will cause the ORed contents of registers B and C to be stored in register C. Thus it is seen how certain faults in microprocessors can result in incorrect data being

stored in various microprocessor registers. Further, we note that this fault will be detected if the contents registers A and B satisfy the inequality:

Contents of A + Contents of B $\neq$ Contents of A

Assumption 4.3.3: If due to a fault in the instruction decoder of a micro processor an incorrect instruction gets executed, and if the incorrect instruction that is executed:

(i) has different instruction timing,

or, (ii) has different number of bytes,

or,(iii) has same number of bytes and execution time, but produces different signals at the external pins,

then this fault can be detected solely by monitoring the response at the external pins (i.e. the incorrect data stored in the various registers need not be examined).

First, it will be explained how an incorrect instruction can get executed in a faulty microprocessor. Consider the schematic diagram of Figure 4.5. The diagram shows that the output from the instruction decoder is applied to the control unit. Assume that the decoder develops a stuck at fault such that whenever the contents corresponding to instruction I_1 are stored in the instruction register the output line corresponding to I_2 gets activated instead. The presence of such a fault will cause I_2 to be executed whenever I_1 is called for.

Further assume that a faulty 8085 microprocessor has a fault in its instruction decoder such that instruction I_2 = XTHL gets executed instead of I_1 = MOV reg 1, reg 2. The documentation for 8085 shows that the instruction I_1 = MOV reg 1, reg 2 has only 4 states, whereas I_2 = XTHL has 16 states. Therefore, this fault will be readily detected by monitoring

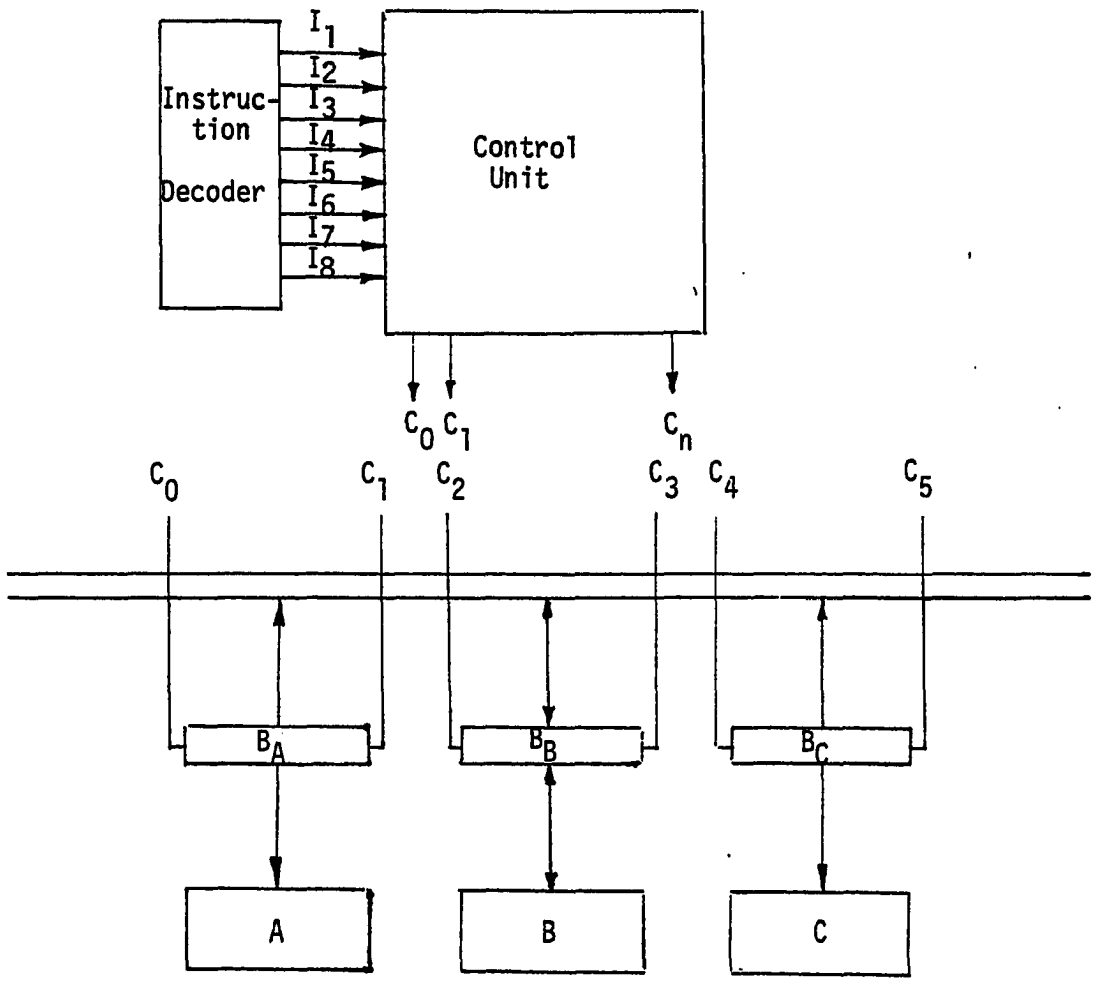


Figure 4.5 Hardware to implement instructions I_1 and I_2.

the external pins of the microprocessor since under this fault the address of the next instruction in the program sequence will appear on the address pins after an interval of 16 states instead of 4 states (OPCODE Timing).

Thus, if due to faults in the instruction decoder, the microprocessor executes incorrect instructions then if the incorrect instruction so executed has different instruction timing, then the fault can be determined solely by monitoring the response at the external pins. In order to facilitate the presentation of the remainder of the theory we introduce two new definitions.

Definition 4.3.1:

External Pin Distinguishable (EPD) Instructions:

If the signals produced at the external microprocessor pins in response to the execution of two instructions I_1 and I_2 are different regardless of data, I_1 and I_2 are said to be external pin distinguishable.

Example 4.3.2

Consider the following two instructions I_1 and I_2 in Motorola MC6800 and having the following descriptions.

	Operation	MNEMONIC	Byte	Execution time (states)
I_1	Complement Accumulator A	COM A	1	2
I_2	Decrement Stack Pointer	DES	1	4

These instructions are EPD instructions because of their different execution times. If I_1 appears in a sequence followed by other instructions, then the next instruction in sequence will be fetched later if I_2 was being executed instead of I_1.

Definition 4.3.2:

External Pin Indistinguishable (EPI) Instructions:

If the signals produced at the external microprocessor pins in response to execution of two instructions I_1 and I_2 are the same, regardless of data, I_1 and I_2 are said to be external pin indistinguishable.

Example 4.3.2

Consider the following two instructions I_1 and I_2 in Intel 8085 having the following descriptions.

I_1

ACI ADD IMMEDIATE WITH CARRY

No. of Bytes: 2
No. of States: 7

I_2

ADI ADD IMMEDIATE

No. of Bytes: 2
No. of States: 7

Once the OP code for ACI or ADI has been fetched, it is impossible to tell which of the two instructions has been executed from the signals produced at the external pins alone. Hence, these instructions are EPI instructions.

Assumption 4.3.4: Only the following type of instruction decoding faults are allowed:

(i) $f(I_i/\phi)$

(ii) $f(I_i/I_k)$. I_j and I_k can be either EPD or EPI instructions

(iii) $f(I_i/I_i+ I_j)$, $f(I_i/I_i + I_j + I_k)$, $f(I_i/I_i + I_j + I_k + \ldots I_n)$ etc. Instructions I_j, I_k, ..., I_n do not belong to the same group of EPI insturctions to which I_i belongs.

(iv) Any other faults in the instruction decoding or control sequence that can alter the microprocessor signals put out at the external pins.

Assumption 4.3.4 implies that data values in the test sequence can be chosen to detect the faults indicated by the above assumption. In real world situations, this data will usually not mask most other types of instruction decoding faults. Therefore, the assumption does not severely limit the capacity of the fault detection procedures.

However, it should be noted that the fault model presented here allows a much larger number of faults than those allowed by the fault model of Thatte and Abraham. Thatte and Abraham methods allow the existence of only one stuck at fault in the instruction decoder. This results in decoding faults of the type $f(I/\phi)$, $f(I_i/I_j)$ and $f(I_i/I_i + I_j)$. Clearly the present model is much broader in scope.

4.4 Test Methodology

Having given the description of the two new instructions and developed an adequate fault model we are now in a position to develop a comprehensive test methodology for testing microprocessors. As we shall see, a large part of the present test method is comprised of testing the microprocessor for the instruction decoding faults. Therefore, before giving the complete test method, we give below the method for testing the microprocessor for instruction decoding faults.

4.4.1 Comprehensive Test Method for Testing Instruction Decoding Faults

The testing for the Instruction Decoding faults can be simplified by making use of the MOVE MULTIPLE instructions and by using a tester that can monitor the signals put out by the microprocessor at its external pins. Consider, for example, a hypothetical microprocessor that puts out different signals at its pins for different instructions in its instruction set such

that the signals put out by no two instructions are the same. This microprocessor can be tested for all possible instruction decoding faults by monitoring the external signals alone. To test this processor for instruction decoding faults each instruction in its instruction set will have to be executed once and the signals at the external pins monitored. If an incorrect instruction is being executed it will be detected by the signals put out at the external pins.

Unfortunately, currently available microprocessors contain sets of instructions that put out identical signals at the external pins. Instruction decoding faults for these instructions cannot be detected by the above described method. However, the test strategy that was described in section 4.4.2 can still be employed. We give below a testing procedure that is applicable to currently available microprocessors. The procedure will be developed to test instruction decoding faults covered by the Assumption 4.3.4. For convenience, we restate this assumption below:

Assumption 4.3.4: Only following type of instruction decoding faults are allowed.

(i) $f(I_i/\phi)$

(ii) $f(I_j/I_k)$. I_j and I_k can either be EPD instructions or EPI instructions.

(iii) $f(I_i/I_i + I_j)$, $f(I_i/I_i + I_j + I_k)$, $f(I_i/I_i + I_j + I_k, \ldots, I_n)$, etc. Instructions I_j, I_k, ... I_n do not belong to the same group of EPI instructions to which I_i belongs.

(iv) Any other faults in the instructions decoding or the control sequence that can alter the microprocessor signals put out at the external pins.

The testing begins by dividing the total microprocessor instruction set into groups of EPI instructions. In order to test instruction decoding faults

in a given EPI instruction group $\{I_1, I_2, \ldots, I_n\}$ the microprocessor under test executes the following sequence:

1. MVMI $addr_1$
2. I_1
3. MVMO $addr_2$
4. MVMI $addr_3$
5. I_2
6. MVMO $addr_4$

x. MOVMI $addr_{(2n-1)}$

y. I_n

z. MOVMO $addr_{(2n)}$

$addr_1, addr_3, \ldots, addr_{(2n-1)}$ = addresses of the memory location in which operands are stored.

$addr_2, addr_4, \ldots, addr_{(2n)}$ = addresses of the memory locations where output operands from MVMO instructions may be stored.

while the response of the microprocessor at its external pins is monitored. It is clear that if any faults covered by Assumption 4.3.4 (ii) and (iii) are present they will be detected by monitoring the external signals whereas any faults covered by Assumption 4.3.4 (i) and (ii) will be detected when the data output by the MVMO instructions in the above sequence is compared with the 'expected output data'. If this test procedure if followed for all the groups of EPI instructions in the microprocessor instruction set, all the instruction decoding faults covered by our fault model can be tested.

4.4.2 Complete Testing Procedure

Complete testing of the microprocessor requires that tests be applied to test all the microprocessor functions, namely, instruction

decoding,register decoding, data storage, data transfer and data manipulation functions. It should be noted that in the test algorithm presented below, the faults in the register decoding function are considered to be a part of the instruction decoding faults. In practice, the information about a particular reigster involved in a given instruction is contained in OP CODE for that instruction itself. The treatment of register decoding faults separate from the instruction decoding faults therefore seems unnecessary. The following procedure is recommended for complete testing.

Test Procedure 4.4.1

Step 1.

Check the given microprocessor for all the possible data transfer and data storage faults using test algorithm . . , while monitoring the signal put out by the external pins. If no fault is indicated in either the expected output or the external signals, the microprocessor is assumed to have no data transfer or data storage faults or faults in the execution of MVMI or MVMO instructions.

Step 2.

Divide the total microprocessor functions into groups of External Pin Indistinguishable (EPI) instructions. This can be done by referring to the instruction set description in the User's Manual for the given microprocessor (for groups of EPI instructions for the Intel 8085 microprocessor, see Appendix A.).

Step 3.

In each group of EPI instructions, the microprocessor is tested for all faults covered by Assumption 4.3.4. The operands needed for the MVMI instructions are determined using the methods

given in Chapter V. The external pins are monitored during the entire test period while the testing is carried out using the hardware fixture described in Section 4.6. Any deviation from the expected behavior either in terms of the operands output by MVMO instructions or the signals output at the external pins can be interpreted as fault.

Step 4.

Sequences to test the Data Manipulation Faults must next be generated. Assuming that a logic level description of the functional units is available, data to test the units can be generated using classical fault detection methods using stuck-at fault models. The classical fault detection methods include such methods as the State Table or Boolean Difference method, etc. (The literature related to these methods was surveyed in Chapter III.) Once the binary test data is obtained in this manner it can be input to the various functional units using a sequence of instructions of class T. Similarly, the result produced by a functional unit can be read out using a sequence of instructions of class T.

It must also be noted that although some faults in the data manipulation function appear to resemble the faults in the instruction decoding and control sequence, and vice versa, the set of faults in one function is not a subset or superset of the faults in the other function. Test sequences to generate each function therefore must be generated separately. Consider for example, a hypothetical microprocessor having a fault in its instruction decoding circuitry such that the "Add" instruction for this

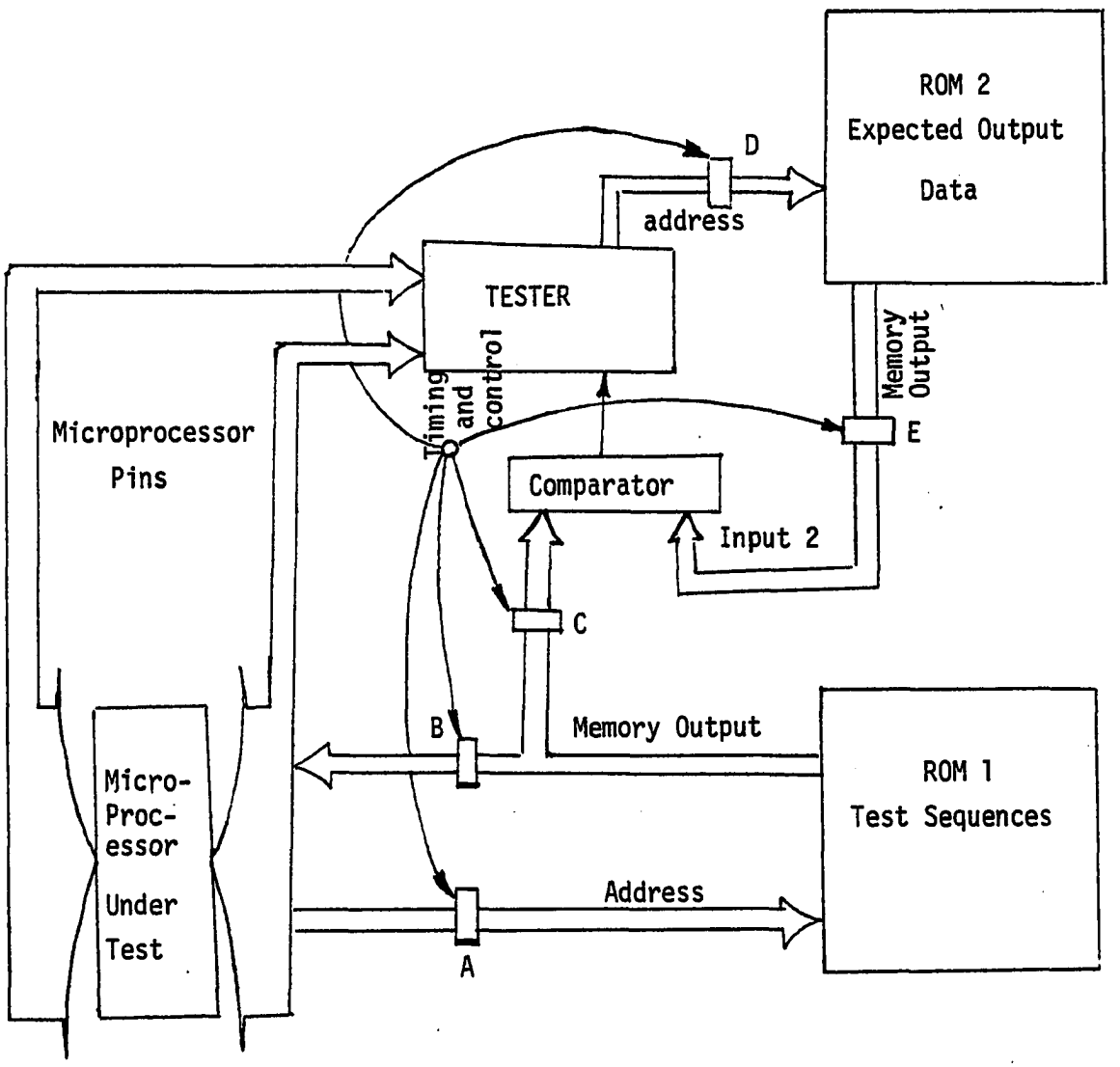


Figure 4.6 A Set Up for Testing Microprocessors.

microprocessor gets decoded as a 'Subtract' instruction. It is also possible that another fault in the data manipulation function causes certain data values to be added instead of subtracted. It is clear that faults such as these can easily go undetected when testing the microprocessor for the instruction decoding faults. However, the proper execution of the tests for the Data Manipulation Function will detect this fault.

4.5 Hardware Test Fixture

The test algorithm given above can be used to test the microprocessors with the setup shown in the schematic of Figure 4.6. The pertinent components of this setup are the tester, microprocessor under test, two Read Only Memories (ROMs), and a comparator. The tester contains the control and the timing logic necessary to carry out the testing. In ROM 1 are stored the test sequences necessary to carry out the testing (these sequences were mentioned in Section 4.5). ROM 2 contains the data (expected output data) that will be output by a fault-free microprocessor in response to the test sequences. The comparator circuit contains circuitry capable of comparing the output from the microprocessor under test to the expected output stored in ROM 2. The various components in the circuit are synchronized by an external clock (not shown).

The pins of various components are interconnected as shown in Figure 4.6. All the pins of the microprocessor under test are monitored by the tester. The address pins of the microprocessor are connected to the address pins of the ROM 1. The address of ROM 2 is supplied by the tester. The outputs from both the ROMs are applied to the comparator. The output from the comparator in turn goes back to the tester. The flow

of data is controlled by various buffers A, B, C, D, E connected at the points shown in the figure. The buffers can either allow the flow of data or restrict it. They are controlled by control and timing signals generated by the tester. As was mentioned in the test algorithm, the testing is divided into two parts: 1) testing for Data Storage and Data Transfer Faults, and 2) testing for Instruction Decoding Faults.

As was mentioned in the test algorithm, the data storage and data transfer faults may be detected by a sequence of the type:

1. MVMI $addr_1$
2. MVMO $addr_2$
3. MVMO $addr_3$
4. MVMI $addr_4$
5. MVMO $addr_5$
6. MVMO $addr_6$

.

.

.

x. MVMI $addr_{(n-2)}$

y. MVMO $addr_{(n-1)}$

z. MVMO $addr_n$

where $addr_1$, $addr_4$, ..., $addr_{(n-2)}$ etc. denote the initial addresses for the blocks of operands needed for MVMI instructions that are stored in the memory ROM 1. Also addresses $addr_2$, $addr_3$, $addr_5$, ..., $addr_{(n-1)}$, $addr_{(n-2)}$ are dummy addresses stored in the second and third bytes of the MVMO instructions. They can have any values, and as we will shortly see, their values are not critical to the testing.

The testing begins by the tester issuing a reset signal to the microprocessor under test. Simultaneously, the buffers A and B are also probed by the tester. The microprocessor responds to the activation of the reset signal by fetching an instruction from the ROM 1 and executing it. This must be the first instruction in the above noted sequence. i.e. MVMI $addr_1$ is executed. The operands needed for its execution are retrieved from addresses $addr_1$ through $addr_7$ in the ROM 1. After the execution of MVMI $addr_1$ is complete, the next instruction in the sequence, i.e.,the the instruction MVMO $addr_2$, is fetched and executed. While MVMO $addr_2$ is being executed by the microprocessor the tester strobes buffer A into cutoff state while strobing buffers B and C into states that allow data to flow in the directions indicated by arrows (Figure 4.6). While the MVMO $addr_2$ is being executed, the tester also applies the expected output from the ROM 2 to one of the comparator inputs. It accomplishes this by strobing tristate buffers into proper states, applying appropriate address to ROM 2 and issuing a read signal to ROM 2. The system is so synchronized that at those instants when a byte output by the MVMO $addr_2$ appears at the input of the comparator, a corresponding byte of 'expected output' is applied at the input 2 of the comparator by the tester. It should also be noted that while MVMO is being executed, the buffer A remains in the cutoff state. This implies that the addresses appearing at the address bus of the microprocessor during the execution of MVMO $addr_2$ are not used to store data in any memory locations. However, they must be monitored by the tester throughout the testing.

Whenever the input 1 of the comparator matches the input 2 of the comparator, the output from the comparator is a 0. This is also clear

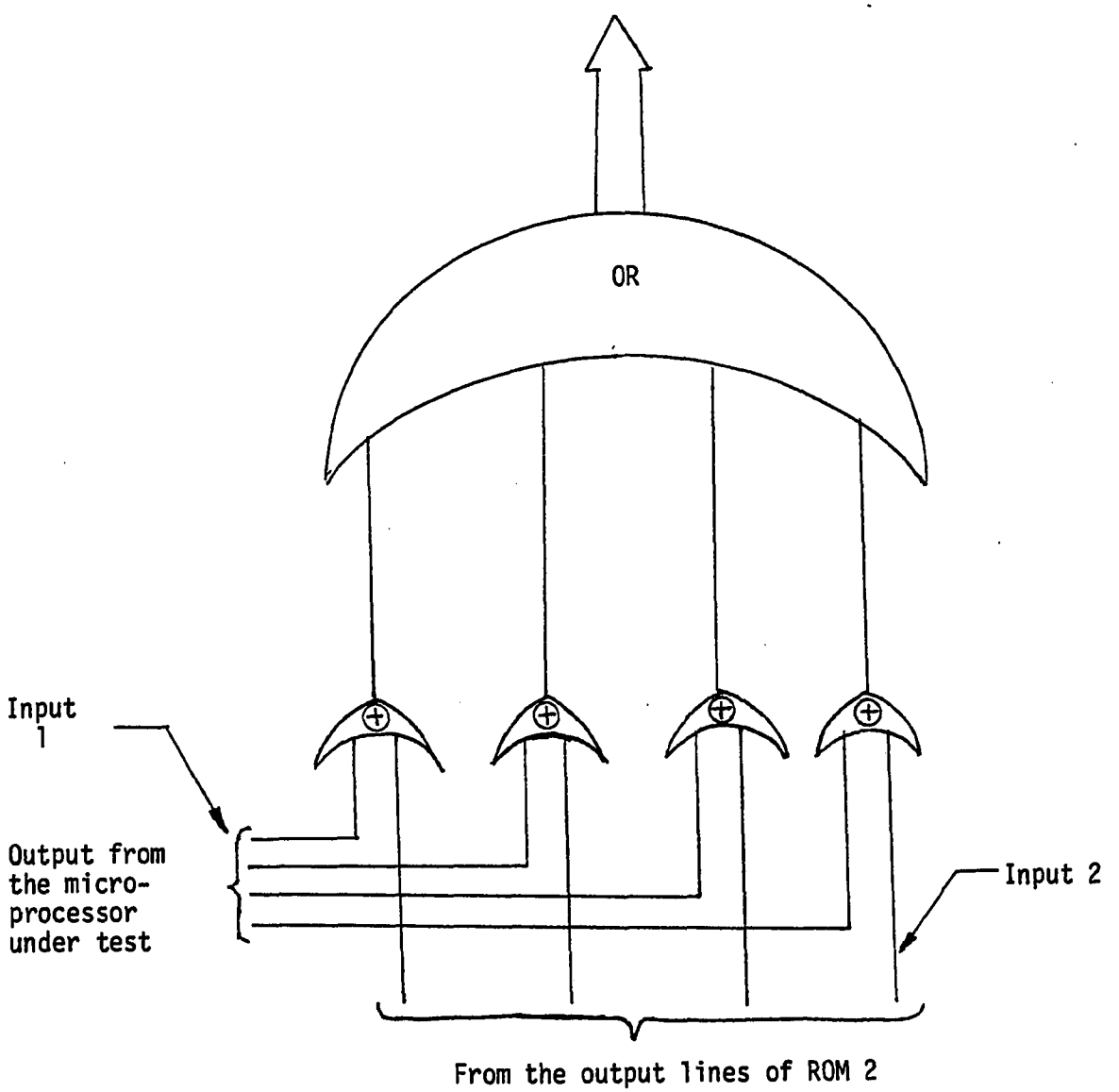


Figure 4.7 Circuit Diagram for a Comparator.

from circuit of the comparator as shown in Figure 4.7. The outputs from the array of EXCLUSIVE OR gates will be zeros if all the corresponding bits in inputs 1 and 2 have same values (i.e., they are both binary 1s or binary 0s). However, if there is at least one pair with non-matching values, then the output of the OR gate must be a binary 1. This value is fed back into the tester and can be used to stop the testing whenever such a condition occurs.

If the data output by the instruction MVMO $addr_2$ matches the expected output from ROM 2, the testing continues. The next instruction in the test sequence, i.e., instruction MVMO $addr_3$, is fetched and executed in the same way. If still no fault is indicated, MVMI $addr_4$ is fetched and executed in a manner similar to MVMI $addr_1$. Subsequent instructions are fetched and executed in a similar manner, until the test sequence is exhausted.

As mentioned earlier the test sequence to test the faults in the instruction decoder can be written down as follows:

1. MVMI $addr_1$
2. I_1
3. MVMO addr
4. MVMI $addr_2$
5. I_2
6. MVMO addr

.

.

.

MVMI addr

MVMO addr

where

$I_1, I_2, \ldots, I_n$ = Instructions in the instruction set of the microprocessor

$addr_1, \ldots, addr_n$ = initial address values for the blocks of operands needed for the MVMI instructions that are stored in the memory ROM 1.

addr = dummy address values stored in the 2nd and 3rd bytes of the MVMI instructions.

The value addr appearing in the second and third bytes of the MVMI instructions is only a dummy value and as we shall shortly see, it does not alter the testing in any way. The above sequence is stored immediately after the test sequence 1. Thus, if no fault in the microprocessor is found while executing sequence 1, the testing automatically continues with the first instruction in sequence 2.

All instructions in the microprocessor instruction set $\{I_1, I_2, \ldots, I_n\}$ must belong to either of the two categories: 1) those instructions that are external pin distinguishable when compared to any other instruction in the microprocessor, or 2) those instructions that are external pin indistinguishable when compared to at least one other instruction in the microprocessor.

If there is a fault in the instruction decoder such that the fault is exhibited whenever I_i belonging to the former category is executed, then the fault can be detected simply by monitoring the response at external microprocessor pins. However, if I_i belongs to the latter category, then the data output by MVMO addr instruction following the I_i must also be compared to the expected output data. Otherwise, the testing procedure for the instruction decoding faults is similar to the testing procedure for the data transfer and data storage faults.

CHAPTER V

CONCLUDING REMARKS AND SUGGESTIONS FOR FURTHER RESEARCH

5.1 Concluding Remarks

The purpose of this research has been to simplify the task of testing microprocessors in a user's environment by suggesting simple design modifications in currently available microprocessors and developing an appropriate test method. Although the fault model was developed under a stated set of assumptions, the model is broad enough to cover most types of faults that can occur in real world situations. There is a slight chance that two faults in different microprocessor functions may mask each other in such a way that they may go undetected by the present method. For example, it is possible for a fault to exist in the data transfer function and for another to exist in the instruction decoding function in such a way that both of them mask each other and may go undetected. However, it is felt that the chance of this occurring is extremely small and usually one of these faults will be detected when the microprocessor is tested for either of the functions. Since the method is meant to be used in a user's environment only, it should not cause severe problems if such faults do go undetected on rare occasions.

It is also felt that the test method suggested here is an improvement over the methods of Thatte and Abraham in at least three ways. (a) First, the tests are easier to generate, (b) second, the fault model is

less restricted and covers a larger number of faults, (c) third, the testing time is considerably reduced.

(a) Ease of Test Generation. The test method given here is simpler to use than the methods of Thatte and Abraham for the detection of Data Transfer, Data Storage or Instruction Decoding Faults. For example, if the methods of Thatte and Abraham are used to detect Data Transfer or Data Storage faults then sequences must be carefully generated for storing appropriate data in various microprocessor registers. Since some microprocessor registers cannot be read in explicitly, this task can be cumbersome. However, by using the MOVE MULTIPLE instructions Data Transfer and Data Storage faults are detected very simply by using the procedures given in Section 4.2.1.

The detection of the Instruction Decoding faults by the methods of Thatte and Abraham requires labelling and classification of the various microprocessor instructions before the tests can be generated. However, the method given in Section 4.4.1 is straightforward and can be applied once the instructions are grouped in groups of External Pin Indistinguishable Instructions.

(b) Fault Model. The fault model assumed by Thatte and Abraham is very restricted since it allows for the existence of only those faults that are caused by a single stuck at fault in the instruction decoder. However, the fault model presented here allows for the detection of all those faults that either alter the signals output at the external microprocessor pins or

cause incorrect data to be stored in the microprocessor registers.

(c) Testing Time. The testing time for microprocessors is considerably reduced by using the method described here. This is so because the present method considerably reduces the length of the test sequences to detect the instruction decoding faults, which form the major portion of the total test sequence. This reduction in the testing time is best illustrated by the test sequences for the Intel 8085 microprocessor.

We first evaluate the time taken to test the Intel 8085 microprocessor using the method of this thesis. This can be done very simply by counting the sequences that are generated in Appendix A. The total test sequence for detecting the Instruction Decoding faults consists of 248 MVMI instructions, 248 MVMO instructions and every instruction in the instruction repertoire of the microprocessor executed at least once. A count of this test sequence yields a total time of clock periods.

It is not possible to calculate the exact time required to test the Intel 8085 microprocessor by the methods of Thatte and Abraham without actually generating the test sequences. However, the minimum time that would be needed can be estimated. This is done using the following reasoning. The total number if IDFs of the type $f(I_j/I_k)$ and $f(I_j/I_j + I_k)$ in a given microprocessor is $2n_I^2$. The length of the test sequence to test each fault depends upon the nature of I_i and I_j. This sequence may be long or short. If we look at the methods of Thatte and Abraham closely, one of the shortest test sequence needed is to test faults of the type $f(I_j/I_k)$ in which $l(I_j) = 1$ (read as label of I_j is 1) and $l(I_k) \geq 2$. In the Intel 8085, the instruction MOV B,C (MOVE Register) has a label 2, whereas MOVE

M,D (Move to Memory) has a label 1. Therefore, in order to detect f(MOVE M,D/MOV B,C) the following procedure adapted from [42] is applicable.

Procedure 5.1

Step 1: Store proper operand in S(MOV M,D) such that when MOV M,D is executed, the expected output 'data' is different from the quiescent value on the data or address bus.

Step 2: Execute MOV M,D

Using the above procedure the test sequence is generated as:

Sequence 5.1

```
LXI  rp
MOV  D,M
INX  H
MOV  M,D
```

On consulting the "MCS-80/85 Family User's Manual" one concludes that the above test sequence can be executed in 60454 clock periods. Assuming that the sequences to detect other instruction decoding faults take no longer to execute the total test sequence can be completed in $2 \times (248)^2 \times 29 = 3567232$ clock periods. We therefore conclude the testing time to test the Intel 8085 microprocessor for instruction decoding faults is reduced to about 1/60th of the Thatte-Abraham method time by the present method.

5.2 Future Research

Although the method presented is adequate when used in the user's environment, the fault model presented herein can be expanded to allow greater number of faults. For example, in Section 4.3 it was mentioned

that our fault model allows those faults that either cause incorrect signals to be output at the microprocessor pins or cause incorrect data to be stored in various microprocessor registers. However, it is possible for two faults to be simultaneously present in the microprocessor and yet have correct signals to be output at the microprocessor pins and correct values to be stored in various registers. For example, there can be a fault present in the instruction decoder and another in the ALU in a way so that the two faults mask the effect of each other. It needs to be investigated as to what design changes or data can enhance the chances of detection of such faults.

REFERENCES

[1] Thatte, S. M., and J. A. Abraham. "Test Generation for Microprocessor", IEEE Transactions on Computers, Vol. C-29, pp. 429-441, June 1980.

[2] Mailing, K., and E. L. Allen. "A Computer Organization and Programming System for Automated Maintenance", IEEE Transactions on Electronic Computers, Vol. EC-12, 1963, pp. 887-895.

[3] Bruer, M. A., and A. D. Friedman. Diagnosis and Reliable Design of Digital Systems. New York: Computer Science Press, 1976.

[4] Kohavi, Zvi. Switching and Finite Automata Theory. New York: McGraw Hill, 1970.

[5] Galey, J. M., R. E. Norby and R. P. Roth. "Techniques for the Diagnosis of Switching Circuit Failures", IEEE Transactions on Communications and Electronics, Vol. 83, No. 74, 1964, pp. 95-110.

[6] Armstrong, D. B. "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets", IEEE Trans. on Electronic Computers, Vol. EC-15, 1966, pp. 66-73.

[7] Schneider, P. R. "On the Necessity to Examine D-Chains in Diagnostic Test Generation -- An Example", IBM Journal, Jan. 1967, p. 114.

[8] Crook, K. J. and J. Blythin. "A Computer Controlled Tester for Logic Networks and a Method for Synthesizing Test Patterns", Joint Conference on Automatic Test Systems, IERE Conference Proceedings, No. 17, April 1970, pp. 187-200.

[9] Seshu, S. "On An Improved Diagnosis Program", IEEE Transactions on Electronic Computers, Vol. EC-14, 1965, pp. 69-76.

[10] Roth, J. P. "Diagnosis of Automata Failures: A Calculus and a Method", IBM Journal R & D, Vol. 10, 1966, pp. 278-291.

[11] Kubo, H. "A Procedure for Generating Test Sequences to Detect Sequential Circuit Failures", NEC R & D, No. 12, 1968. pp. 69-78.

[12] Hennie, F. C. "Fault Detecting Experiments for Sequential Circuits", 1964, Proc. of the Fifth Annual Switching Theory and Logical Design Symposium, S-164, pp. 95-110.

[13] Kime, C. R. "An Organization for Checking Experiments on Sequential Circuits", IEEE Transactions on Electronic Computers, Vol. EC-15, 1966, pp. 113-115.

[14] Kohavi, Z., and P. Lavallec. "Design of Sequential Machines with Fault Detection Capability", IEEE Transactions on Electronic Computers, Vol. EC-16, 1967, pp. 473-484.

[15] Kohavi, I., and Z. Kohavi. "Variable Length Distinguishing Sequences and Their Application to the Design of Fault-Detection Experiments", IEEE Transactions on Computers, Vol. C-17, 1968, pp. 792-795.

[16] Gonenc, G. "A Method for the Design of Fault Detection Experiments", IEEE Computer Group Repository, R-69-134.

[17] Amar, V., and V. Condulmari. "Diagnosis of Large Combinational Networks", IEEE Trans. on Electronic Computers, Vol. EC-16, 1967, pp. 675-680.

[18] Sellers, F. F. Jr., M. Y. Hsia and L. W. Bearnson. "Analyzing Errors with the Boolean Difference", IEEE Trans. on Computers, Vol. C-17, 1968, pp. 676-683.

[19] Marinox, P. N. "A Method of Deriving Minimal Complete Sets of Test Input Sequences Using Boolean Difference", IEEE Computer Group Repository, R-70-22.

[20] Carroll, A. B., M. Kato, Y. Koga and K. Naemura. "A Method of Diagnostic Test Generation", Proc. AFIPS, SJCC, 1969, pp. 221-228.

[21] Kajitani, K., Y. Tezuka and Y. Kasahara. "Diagnosis of Multiple Faults in Combinational Circuits", Electronics and Communications in Japan, Vol. 52-C, 1969, pp. 123-131.

[22] Lewis, R. S. Jr. "An Approach to Test Pattern Generation for Synchronous Circuits", Ph.D. Thesis, Southern Methodist University, 1967.

[23] Jones, P. R. and C. H. Mays. "Automatic Test Generation Methods for Large Scale Integration Logic", IEEE Journal of Solid-State Circuits, Vol. SC-2, 1967, pp. 221-226.

[24] Hillman, L. "An Automatic Dynamic Digital Logic Circuit Test System", Computer Design, Vol. 8, Aug. 1969, pp. 58-62.

[25] De Atley, E. "LSI Testing is a Large Scale Headache!", Electronic Design, 16, Aug. 2nd, 1969, pp. 24-34.

[26] Brown, J. R. Jr. "Pattern Sensitivity in MOS Memories", Dig. Symp. Testing to Integrate Semiconductor Memories into Computer Mainframes, Oct. 1972, pp. 33-46.

[27] Hayes, J. P. "Detection of Pattern-Sensitive Faults in Random Access Memories", IEEE Trans. Comput., Vol. C-34, Feb. 1975, pp. 150-157.

[28] Boonstra, D., P. Lock, A. Lambrechtse, W. Cees and R. H. W. Salters. "A 4096-b One-Transistor Per Bit Random-Access Memory With Internal Timing and Low Dissipation", IEEE J. Solid-State Circuits, Vol. SC-8, Oct. 1973, pp. 305-310.

[29] Abbott, R. A., W. M. Regitz and J. A. Kapr. "A 4K MOS Dynamic Random-Access-Memory", IEEE J. Solid-State Circuits, Vol. SC-8, Oct. 1973, pp. 292-298.

[30] Thatte, S. M. "Fault Diagnosis of Semiconductor Random Access Memories", Coordinated Sci. Lab. Rep. R-769, May 1977.

[31] Thatte, S. M. and J. A. Abraham. "Testing of Semiconductor Random Access Memories", Proc. 7th Annual Int. Conf. Fault-Tolerant Computing, IEEE Comp. Soc., June 1977, pp. 81-87.

[32] Nair, R., S. M. Thatte, and J. A. Abraham. "Efficient Algorithms for Testing Semiconductor Random-Access Memories", IEEE Trans. Comput., Vol. C-26, June 1978, pp. 572-576.

[33] Nair, R. "Comments on 'An Optimal Algorithm for Testing Stuck-At Faults in Random-Access Memories'," IEEE Trans. Comput., to be published.

[34] Knaizuk, J. Jr., and C. R. P. Hartmann. "An Optimal Algorithm for Testing Stuck-At Faults in Random-Access Memories", IEEE Trans. Comput., Vol. C-25, Nov. 1977, pp. 1141-1144.

[35] Cocking, J. "RAM Test Patterns and Test Strategy", Dig. Papers, 1975, Symp. Semiconductor Memory Testing, IEEE Comput. Soc., Oct. 1975, pp. 1-8.

[36] Suk, D. S. and S. M. Reddy. "Test Procedures for a Class of Pattern Sensitive Faults in Semiconductor Random Access Memories", IEEE Transactions on Computers, Vol. C-29, June 1980, pp. 419-429.

[37] Chiang, A. C. L. and R. McCaskill. "Two New Approaches to Simplify Testing of Microprocessors", Electronics, Jan. 22, 1976, pp. 100-105.

[38] Fee, W. G. Turotiral LSI Testing, 2nd, ed., IEEE Comput. Soc., 1978, IEEE Catalog EHO-122-2.

[39] Sridhar, T. and J. P. Hayes. "Testing Bit Sliced Microprocessors", Proceedings 9th Int. Conf. Fault Tolerant Computing, Madison, WI, IEEE Comput. Soc., June 1979, pp. 211-218.

[40] Kenish, J.C. "Test Equipment for Microprocessors" IBM Tech. Disclosure Bull. (USA), Vol 24, no. 3, P. 1676 (Aug. 1981)

[41] Puthenpurayil, V. and J.R. Armstrong," Functional Level Modelling of LSI devices" Proceedings of the Fourteenth Southeastern Symposium on System Theory", Blacksburg, VA, 15-16 April 1982.

[42] Su, S.Y.H and Hsieh, Yu.I." Testing Functional Faults in Digital Systems Described by Register Transfer Language" 1981 International Test Conference. Testing in the 1980's, Philadelphia, PA, 27-29 Oct. 1981.

[43] Sacher, E. "High-speed Functional Testing of Microprocessor Based Circuit Board" 1981 International Test Conference. Testing in the 1980's. Philedelphia, PA, 27-29 Oct. 1981.

[44] Buckroyd, A. "Designing Microprocessors for Testability" Electron. Technol. (GB) Vol. 16, no- 3 P. 48-51 (March 1982)

[45] Robach, C. and G. Saucier. "Diversified Test Methods for Local Control Units", IEEE Trans. Comput., Vol. C-24, May 1975, P. 3-10.

[46] Thatte, S. M. and J. A. Abraham. "A Methodology for Functional Level Testing", Proc. 8th Int. Conf. Fault-Tolerant Computing, Toulouse, France: IEEE Comput. Soc., June 1978, P. 90-95.

[47] Thatte, S. M. "Test Generation for Microprocessors", Coordinated Sci. Lab., Univ. of Illinois, Urbana, IL, Rep. R-842, May 1979.

APPENDIX A

APPLICATION OF THE TEST METHOD TO THE INTEL 8085 MICROPROCESSOR

A.1 Introduction

In this section, the test methodology of Chapter IV is applied to the Intel 8085 microprocessor. The methodology is followed step by step to generate the test sequences for the microprocessor. The instruction names and other notation relating to the 8085 microprocessor are taken from the 'MCS-80/85 Family User's Manual' -- a document published by Intel. The testing has been divided into three parts: the testing of the Instruction Decoding faults, testing of Data Storage and Data Transfer faults and the testing of Data Manipulation faults.

A.2 Detection of Instruction Decoding Faults

A.2.1 Grouping the Total Instruction Set into EPI Instructions

The task before us is to detect all faults covered under Assumption 4.3.4 in the instruction set of the Intel 8085 microprocessor. This requires grouping the total instruction set of the Intel 8085 microprocessor into groups containing External Pin Indistinguishable instructions. It is obvious the instructions that have different number of bytes or take different number of clock periods for their execution, must be External Pin Distinguishable. In addition, the instruction timing of the 8085 microprocessor is divided into 'Machine Cycles'. Instructions with different Machine Cycles are also External Pin Distinguishable. Consider for example,

MC1						MC2	MC3	MC4	MC5
T_1	T_2	T_3	T_4	T_5	T_6	$T_1T_2T_3$	$T_1T_2T_3$	$T_1T_2T_3$	$T_1T_2T_3$

Figure A.1 Machine Cycles for the 8085 Microprocessor.

the timing of a typical instruction cycle of 8085 microprocessor in Figure 5.1. As shown in that figure, the machine cycle of the 8085 consists of three to six clock periods. The first machine cycle (known as the instruction fetch cycle) has four or six clock periods. Subsequent machine cycles have three clock periods only. This is illustrated in Figure A.1 by shading the appropriate areas. When MC is shaded, the entire Machine Cycle is optional. When T is shaded the clock period is optional within its machine cycle. The microprocessor signals the start of a new cycle to the outside world by pulsing the ALE pin (see MCS-80/85 Family User's Manual for description of the ALE pin and other microprocessor pins) high during the first clock period of every Machine Cycle. In addition, the pins S_0 and S_1 are both output high during an "instruction fetch" machine cycle. It follows that the signals put out on pins ALE, S_0 and S_1, can be monitored to get an indication as to which microprocessor cycle is being executed.

Complete information about the number of cycles, number of clock periods and number of machine cycles for each instruction in the instruction set of the 8085 microprocessor is available in the MCS-80/85 Family User's Manual. Using this information, the total number of instructions in the instruction set of the 8085 can be divided into following groups of External Pin Indistinguishable instructions.

Group I: Instructions with 2 bytes, 7 clock periods and 2 cycles.

ACI	data	ADD IMMEDIATE WITH CARRY
ACI	data	ADD IMMEDIATE
ANI	data	ADD IMMEDIATE WITH ACCUMULATOR
CPI	data	COMPARE IMMEDIATE

MVI	r,data	MOVE IMMEDIATE
ORI	data	INCLUSIVE OR IMMEDIATE
XRI	data	EXCLUSIVE OR IMMEDIATE
SBI	data	SUBTRACT IMMEDIATE WITH BORROW
STAX	rp	STORE ACCUMULATOR INDIRECT
SUI	data	SUBTRACT IMMEDIATE
XRI	data	EXCLUSIVE OR IMMEDIATE

Group II: Instructions with 1 byte, 10 clock periods and 3 cycles.

This group can further be divided into five subgroups.

Subgroup A: In the following subgroup of instructions the first cycle in an instruction fetch cycle, the second cycle is a memory read cycle and the third cycle is also a memory read cycle.

POP	rp	POP
RET		RETURN FROM SUBROUTINE

Subgroup B: In the following subgroup of instructions, the first cycle is an instruction fetch cycle, the second cycle is a memory read cycle and the third cycle is a memory write cycle.

INR	M	INCREMENT MEMORY
DCR	M	DECREMENT MEMORY

Subgroup C: In the following subgroup of instructions the first cycle is an instruction fetch cycle and the second and third cycles are bus idle cycles.

DAD	rp	DOUBLE REGISTER ADD

Group III: Instructions with 1 byte, 7 clock periods and 2 cycles.

This group can further be divided into two subgroups.

Subgroup A: In the following subgroup of instructions the first cycle is an instruction fetch cycle whereas the second cycle is a memory read cycle.

ANA	M	LOGICAL AND WITH ACCUMULATOR
CMP	M	COMPARE WITH ACCUMULATOR
LDAX	rp	LOAD ACCUMULATOR INDIRECT
MOV	r,M	MOVE
ADC	M	ADD MEMORY WITH CARRY
STAX	rp	STORE ACCUMULATOR INDIRECT
SBB	M	SUBTRACT REGISTER WITH BORROW
ORA	M	INCLUSIVE OR WITH ACCUMULATOR
SUB	M	SUBTRACT
SUI	data	SUBTRACT
XRA	M	EXCLUSIVE OR WITH ACCUMULATOR

Subgroup B: In the following instruction, the first cycle is an instruction fetch cycle whereas the second cycle is a memory write cycle.

MOV	M,r	MOVE

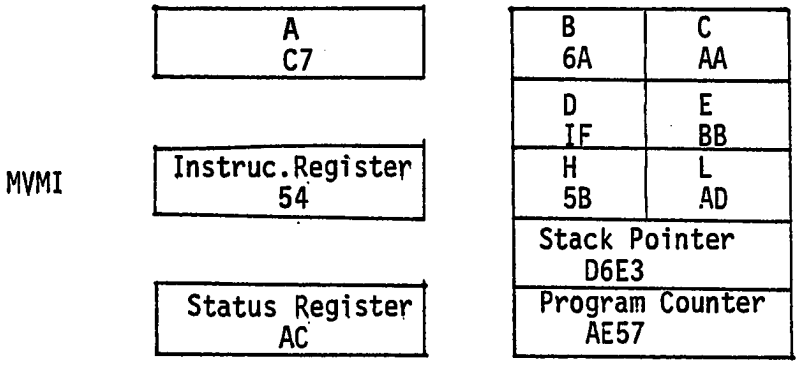
Group IV: Instructions with 1 byte, 4 clock periods and 1 machine cycle.

ADC r	ADD WITH CARRY
ADD r	ADD
ANA r	LOGICAL AND WITH ACCUMULATOR
CMA	COMPLEMENT ACCUMULATOR

CMC	COMPLEMENT CARRY
CMP r	COMPARE WITH ACCUMULATOR
DAA	DECIMAL ADJUST ACCUMULATOR
DCR r	RECREMENT CONTENTS OF REGISTER
INR r	INCREMENT REGISTER
MOV r1,r2	MOVE
NOP	NO OPERATION
ORA r	INCLUSIVE OR WITH ACCUMULATOR
RAL	ROTATE LEFT THROUGH CARRY
RAR	ROTATE RIGHT THROUGH CARRY
RLC	ROTATE ACCUMULATOR LEFT
RRC	ROTATE ACCUMULATOR RIGHT
SBB r	SUBTRACT WITH BORROW
STC	SET CARRY
SUB r	SUBTRACT
XCHG	EXCHANGE
XRA r	EXCLUSIVE OR MEMORY

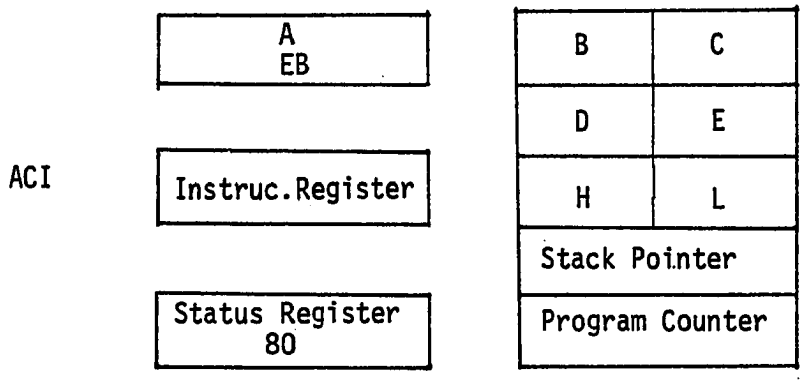
Group V: Instructions with 3 bytes, 9 (18) clock periods, and 2 (5 cycles). The first listed unconditional CALL instruction of this group always takes 18 clock periods and 5 cycles to execute. The remaining instruction of the group are conditional CALL instructions. If a conditional CALL is executed it takes the same time as the unconditional CALL; otherwise, it is completed in 9 clock periods and 5 cycles .

CALL	UNCONDITIONAL CALL
CC	CALL IF CARRY



MVMI

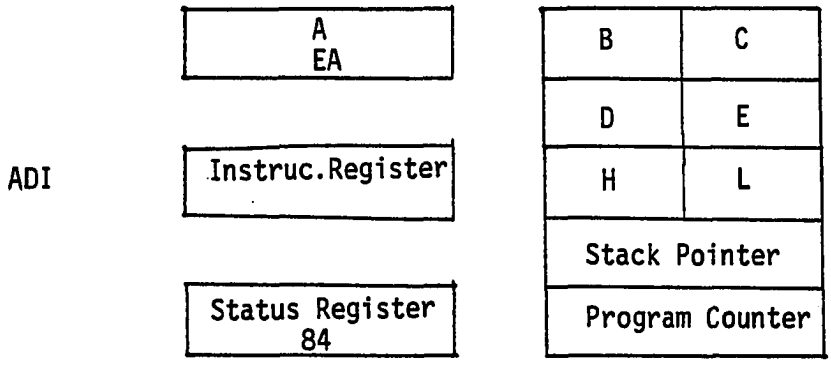
(a)



ACI

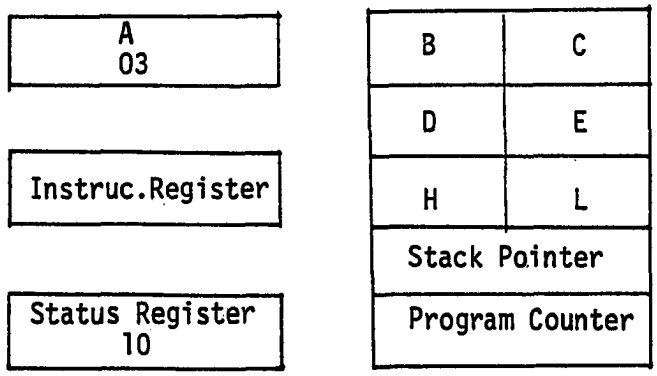
(b)

Figure A.2 Stored Values in the Microprocessor Registers After Various Instructions of Group I Are Executed.



ADI

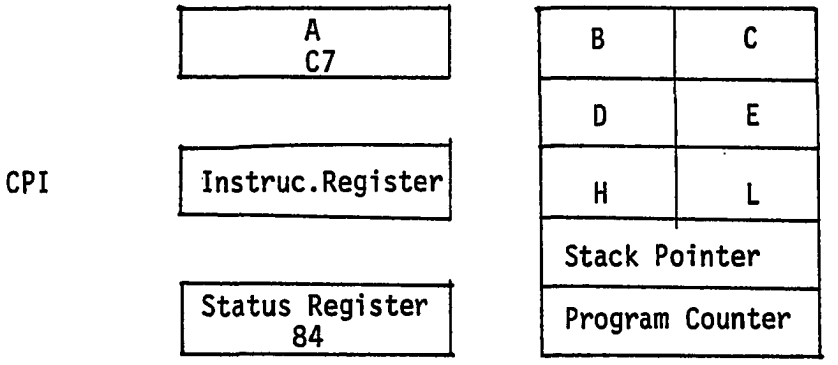
(c)



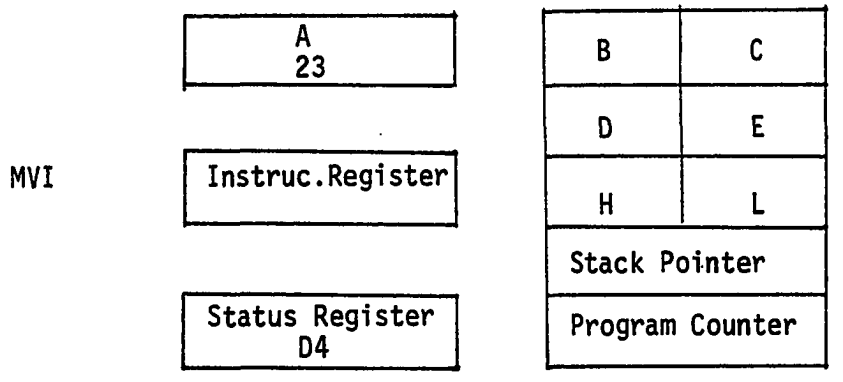
ANI

(d)

Figure A.2 (Cont'd) Stored Values in the Microprocessor Registers After Various Instructions of Group I are Executed.

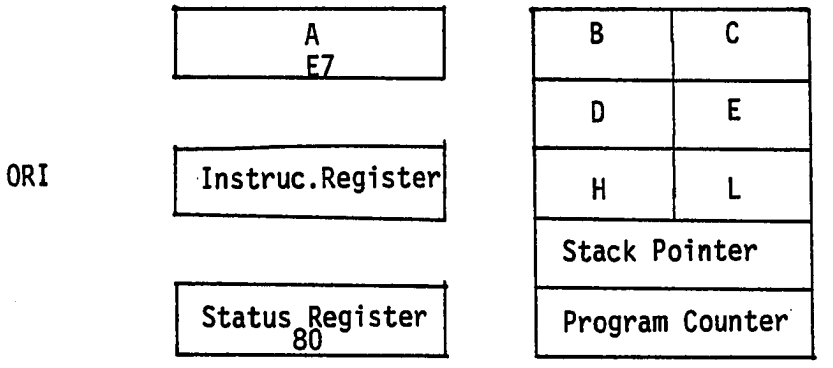


(e)

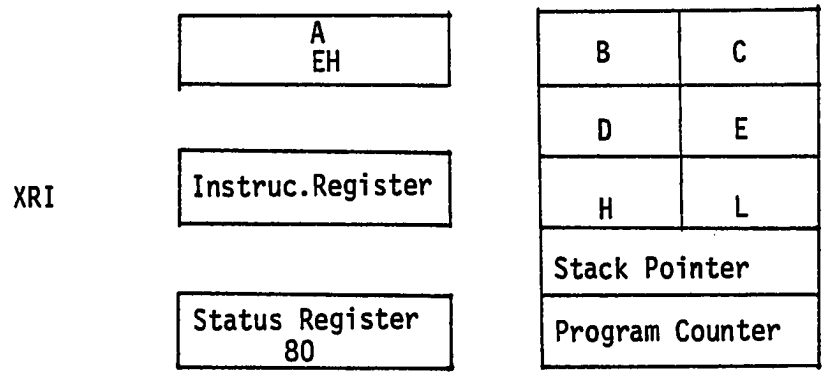


(f)

Figure A.2 (Cont'd.) Stored Values in the Microprocessor Registers After Various Instructions of Group I are Executed.



(f)



(h)

Figure A.2 (Cont'd.) Stored Values in the Microprocessor Registers After Various Instructions of Group I are Executed.

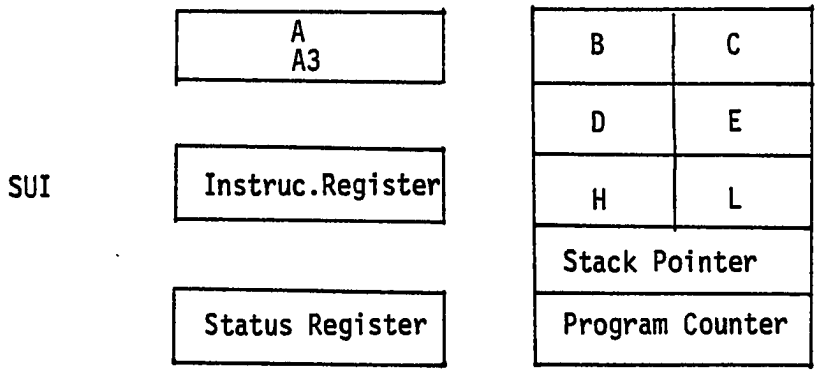


Figure A.2 (Cont'd.) Stored Values in the Microprocessor Registers After Various Instructions of Group I are Executed.

Sequence A.2.1

1. MVMI $addr_1$
2. ACI 23H
3. MVMO $addr_2$

Note: It is assumed that the memory space pointed by

$addr_1$	contains	C7H
$addr_1$ + 1	contains	6AH
$addr_1$ + 2	contains	AAH
$addr_1$ + 3	contains	1FH
$addr_1$ + 4	contains	BBH
$addr_1$ + 5	contains	5BH
$addr_1$ + 6	contains	ADH
$addr_1$ + 7	contains	D6H
$addr_1$ + 8	contains	E3H
$addr_1$ + 9	contains	AEH
$addr_1$ + 10	contains	S7H
$addr_1$ + 11	contains	ACH
$addr_1$ + 12	contains	S4H

To show that the data used in the above MVMI instruction does not mask the above mentioned faults (such data would be termed as fault sensitive), use will be made of the schematics of Figures A.2 (a) through (i). These schematics represent the registers of 8085 as described in the MCS-80/85 Family User's Manual. In Figure A.2 (a) are shown the various values that will be stored in the various microprocessor registers after MVMI of step 1 is executed. The actual balues are written

Group VII: Instructions with 1 byte, 6 clock periods and 1 cycle.

DCX	DECREMENT REGISTER PAIR
INX	INCREMENT REGISTER PAIR
PCHL	MOVE H & L TO PROGRAM COUNTER
SPHL	MOVE H & L TO THE STACK POINTER

Group VIII: Instructions with 1 byte, 6 (12) clock periods and (1) 3 cyles.

RC	RETURN IF CARRY
RM	RETURN IF MINUS
RNC	RETURN IF NO CARRY
RNZ	RETURN IF NOT ZERO
RP	RETURN IF POSITIVE
RPE	RETURN IF PARITY EVEN
RPO	RETURN IF PARITY ODD
RZ	RETURN IF ZERO

The next step is to generate test sequences to detect faults of the type $f(I_j/\phi)$ and $f(I_j/I_k)$ in each of the groups of EPI instruction. These faults can be detected by executing the MOVE MULTIPLE instructions as was mentioned in Chapter IV. The data for the MOVE MULTIPLE instructions must be chosen appropriately.

A.2.2 Detection of Faults in Group 1

It is easiest to illustrate the generation of test sequences for this group of instructions by an example. Assume that it is required to generate tests for instructions of the type $f(\text{ACI data}/I_k)$ and $f(\text{ACI data}/\phi)$ where I_k is an instruction (other than ACI) belonging to Group I. It is claimed that the following sequence when executed with appropriate data detected all such faults.

Sequence A.2.2

```
MVMI addr1
ADI  23H
MVMO addr2
MVMI addr1
ANI  23H
MVMO addr2
MVMI addr1
CPI  23H
MVMO addr2
MVMI addr1
MVI  23H
MVMO addr2
MVMI addr1
ORI  23H
MVMO addr2
MVMI addr1
XRI  23H
MVMO addr2
MVMI addr1
SUI  23H
MVMO addr1
MVMI addr1
STAX B
MVMO addr2
MVMI addr1
STAX D
MVMO addr1
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

```
SBI  23H
MVMO addr2
MVMI addr1
XRI  23H
MVMI addr1
```

Faults of the type $f(I/\phi)$ are also detected by the above sequence. If such a fault is present for any instruction I_j belonging to Group I, no response will be produced at the pins when I_j is executed during instructions 2, 5, 8, 11, 14, 17 and 20 of the above sequence. This condition is easily detected by the external monitor.

A.2.3 Detection of Instruction Decoding Faults in Group II

The instruction sequence to detect faults in this group can be generated in the same way as that for the instructions in Group I. However, we need to write down separate test sequences for each of the subgroups.

Sequence A.2.3 (for Subgroup A)

```
MVMI addr1
POP  B
MVMO addr2
MVMI addr1
POP  D
MVMO addr2
MVMI addr1
POP  H
MVMO addr2
MVMI addr1
POP  PSW
```

into the registers; whereas the mnemonic MVMI is written to the left of the schematic.

Figure A.2 (b) represents the values that will be stored in various microprocessor registers after ACI of step 2 is executed. The registers that are shown blank are assumed to contain the data that was stored in them by the preceding MVMI instruction, i.e., they contain the same stored values as in Figure A.2 (a).

Figure A.2 (c) shows data values that would be stored in the various microprocessor registers if the instruction ADI were executed in step 2 of sequence A.2.1 instead of ACI. This is denoted by writing MVMI/ADI to the left of this schematic. Once again the registers shown blank are assumed to contain the data values that were stored in them by the preceding MVMI instruction, i.e., the data values shown in Figure A.2 (a).

Figures A.2 (d) through A.2 (e) are completed in a similar manner to Figures A.2 (a) and (b). The figures are a convenient way of determining if the data used in the MVMI instruction in step 1 of sequence A.2.1 was fault sensitive. If the data in any of these Figures A.2 (b) through (e) is identical, the corresponding fault will be masked.

However, the examination of Figure A.2 shows that the data used in the MVMI instruction of sequence A.2.1 was indeed fault sensitive. If a decoding fault of the type $f(I_j/I_k)$ is present, it will be detected when the MVMO of sequence A.2.1 reads out the data from all registers in step 3.

The same reasoning can be used to detect all the possible instruction decoding faults covered by Assumption 4.3.4 in Group 1 of the EPI instructions. The following sequence will detect the remaining faults.

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

```
MVMO addr2
MVMI addr1
RET
MVMO addr2
```

Sequence A.2.4 (for Subgroup B)

```
MVMI addr1
INR  M
MVMO addr2
MVMI addr1
DCR  M
MVMO addr2
```

Sequence A.2.5 (for Subgroup C)

```
MVMI addr1
DAD  B
MVMO addr2
MVMI addr1
DAD  D
MVMO addr2
MVMI addr1
DAD  H
MVMO addr2
MVMI addr1
DAD  SP
MVMO addr2
```

The operands used in these instructions were determined to be fault sensitive. Schematics such as those given in Figure A.2 were used to determine the fault sensitiveness. However, those schematics are not presented here to keep this discussion brief.

A.2.4 Decoding Faults in Group III

The instruction sequence to detect faults in this group is generated in exactly the same way as that for the instructions in Group II. This sequence can be written down as:

Sequence A.2.6

(for Subgroup A)

```
MVMI addr1
ANA  M
MVMO addr2
MVMI addr1
CMP  M
MVMO addr2
MVMI addr1
MOV  B,M
MVMO addr2
MVMI addr1
MOV  C,M
MVMO addr2
MVMI addr1
MOV  D,M
MVMO addr2
MVMI addr1
MOV  E,M
MVMI addr1
MVMO addr2
MOV  H,M
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

```
MVMI addr1
MOV  L,M
MVMO addr2
MVMI addr1
ORA  M
MVMO addr2
MVMI addr1
ADC  M
MVMO addr2
MVMI addr1
SUB  M
MVMO addr2
MVMI addr1
SUI  M
MVMO addr2
MVMI addr1
XRA  M
MVMO addr2
MVMI addr1
STAX B
MVMO addr2
MVMI addr1
STAX D
MVMO addr2
MVMI addr1
SBB  M
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Comments: In the above sequence, a data value of 24H is stored in the memory location designated by the H & L registers (i.e., in memory location of 5BADH).

Sequence A.2.7

(for Subgroup B)

```
MVMI addr1
MOV  M,A
MVMO addr2
MVMI addr1
MOV  M,R
MVMO addr2
MVMI addr1
MOV  M,C
MVMO addr2
MVMI addr1
MOV  M,D
MVMO addr2
MVMI addr1
MOV  M,E
MVMO addr2
MVMI addr1
MOV  M,H
MVMO addr2
MVMI addr1
MOV  M,L
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

A.2.5 Detection of Decoding Faults in Group IV

The test sequence to detect faults in this group is generated in the same way as that for the instructions in Groups I, II, and III.

Sequence A.2.8

	MVMI $addr_1$
	ADC A
	MVMO $addr_2$
	MVMI $addr_1$
	ADC B
	MVMO $addr_2$
	MVMI $addr_1$
	ADC C
	MVMO $addr_2$
	MVMI $addr_1$
	ADC D
	MVMO $addr_2$
	MVMI $addr_1$
	ADC E
	MVMO $addr_2$
	MVMI $addr_1$
	ADC H
	MVMO $addr_2$
	MVMI $addr_1$
	ADC L
	MVMO $addr_2$
	MVMI $addr_1$

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
ADD  A
MVMO addr2
MVMI addr1
ADD  B
MVMO addr2
MVMI addr1
ADD  C
MVMO addr2
MVMI addr1
ADD  D
MVMO addr2
MVMI addr1
ADD  E
MVMO addr2
MVMI addr1
ADD  H
MVMO addr2
MVMI addr1
ADD  L
MVMO addr2
MVMI addr1
ANA  A
MVMO addr2
MVMI addr1
ANA  B
MVMO addr2
MVMI addr1
ANA  C
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMO addr2
MVMI addr1
ANA  D
MVMO addr2
MVMI addr1
ANA  E
MVMO addr2
MVMI addr1
ANA  H
MVMO addr2
MVMI addr1
ANA  L
MVMO addr2
CMA
MVMI addr1
CMC
MVMO addr2
MVMI addr1
CMP  A
MVMO addr2
MVMI addr1
CMP  B
MVMO addr2
MVMI addr1
CMP  C
MVMO addr2
MVMI addr1
CMP  D
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMI addr1
CMP  E
MVMO addr2
MVMI addr1
CMP  H
MVMO addr2
MVMI addr1
CMP  L
MVMO addr2
MVMI addr1
XRA  B
MVMO addr2
MVMI addr1
XRA  C
MVMO addr2
MVMI addr1
XRA  D
MVMO addr2
MVMI addr1
XRA  E
MVMO addr2
MVMI addr1
XRA  H
MVMO addr2
MVMI addr1
XRA  L
MVMO addr2
MVMI addr1
XRA  A
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMO addr2
DAA
MVMO addr2
MVMI addr1
DCR  A
MVMO addr2
MVMI addr1
DCR  B
MVMO addr2
MVMI addr1
DCR  C
MVMO addr2
MVMI addr1
DCR  D
MVMO addr2
MVMI addr1
DCR  E
MVMO addr2
MVMI addr1
DCR  H
MVMO addr2
MVMI addr1
DCR  L
MVMO addr2
MVMI addr1
INR  A
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMI addr1
INR  B
MVMO addr2
MVMI addr1
INR  C
MVMO addr2
MVMI addr1
INR  D
MVMO addr2
MVMI addr1
INR  E
MVMO addr2
MVMI addr1
INR  H
MVMO addr2
MVMI addr1
INR  L
MVMO addr2
MVMI addr1
ORA  A
MVMO addr2
MVMI addr1
ORA  B
MVMO addr2
MVMI addr1
ORA  C
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMO addr2
MVMI addr1
ORA  D
MVMO addr2
MVMI addr1
ORA  E
MVMO addr2
MVMI addr1
ORA  H
MVMO addr2
MVMI addr1
ORA  L
MVMO addr2
MVMI addr1
RAL
MVMO addr2
MVMI addr1
RAR
MVMO addr2
MVMI addr1
RLC
MVMO addr2
MVMI addr1
RRC
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
SBB  B
MVMO addr2
MVMI addr1
SBB  C
MVMO addr2
MVMI addr1
SBB  D
MVMO addr2
MVMI addr1
SBB  E
MVMO addr2
MVMI addr1
SBB  H
MVMO addr2
MVMI addr1
SBB  L
MVMO addr2
MVMI addr1
SBB  A
MVMO addr2
MVMI addr1
STC
MVMO addr2
MVMI addr1
SUB  A
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMI addr1
SUB  B
MVMO addr2
MVMI addr1
SUB  C
MVMO addr2
MVMI addr1
SUB  D
MVMO addr2
MVMI addr1
SUB  E
MVMO addr2
MVMI addr1
SUB  H
MVMO addr2
MVMI addr1
SUB  L
MVMO addr2
MVMI addr1
MOV  A,A
MVMO addr2
MVMI addr1
MOV  A,B
MVMO addr2
MVMI addr1
MOV  A,C
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMO addr2
MVMI addr1
MOV  A,D
MVMO addr2
MVMI addr1
MOV  A,E
MVMO addr2
MVMI addr1
MOV  A,H
MVMO addr2
MVMI addr1
MOV  A,L
MVMO addr2
MVMI addr1
MOV  B,A
MVMO addr2
MVMI addr1
MOV  B,B
MVMO addr2
MVMI addr1
MOV  B,C
MVMO addr2
MVMI addr1
MOV  B,D
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MOV  B,E
MVMO addr2
MVMI addr1
MOV  B,H
MVMO addr2
MVMI addr1
MOV  B,L
MVMO addr2
MVMI addr1
MOV  C,A
MVMO addr2
MVMI addr1
MOV  C,B
MVMO addr2
MVMI addr1
MOV  C,C
MVMO addr2
MVMI addr1
MOV  C,D
MVMO addr2
MVMI addr1
MOV  C,E
MVMO addr2
MVMI addr1
MOV  C,H
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMI addr1
MOV  C,L
MVMO addr2
MVMI addr1
MOV  D, A
MVMO addr2
MVMI addr1
MOV  D,B
MVMO addr2
MVMI addr1
MOV  D,E
MVMO addr2
MVMI addr1
MOV  D,H
MVMO addr2
MVMI addr1
MOV  D,L
MVMO addr2
MVMI addr1
MOV  E,A
MVMO addr2
MVMI addr1
MOV  E,B
MVMO addr2
MVMI addr1
MOV  E,C
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MVMO addr2
MVMI addr1
MOV  E,D
MVMO addr2
MVMI addr1
MOV  E,E
MVMO addr2
MVMI addr1
MOV  H,A
MVMO addr2
MVMI addr1
MOV  H,B
MVMO addr2
MVMI addr1
MOV  H,C
MVMO addr2
MVMI addr1
MOV  H,D
MVMO addr2
MVMI addr1
MOV  H,E
MVMO addr2
MVMI addr1
MOV  H,H
MVMO addr2
MVMI addr1
MOV  L,A
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8

(Continued)

```
MOV  L,B
MVMO addr2
MVMI addr1
MOV  L,C
MVMO addr2
MVMI addr1
MOV  L,D
MVMO addr2
MVMI addr1
MOV  L,E
MVMO addr2
MVMI addr1
MOV  L,H
MVMO addr2
MVMI addr1
MOV  L,L
MVMO addr2
MVMI addr1
XRA  A
MVMO addr2
MVMI addr1
XRA  B
MVMO addr2
MVMI addr1
XRA  C
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

Sequence A.2.8 (Continued)

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1.

```
XRA  D
MVMO addr2
MVMI addr1
XRA  E
MVMO addr2
MVMI addr1
XRA  H
MVMO addr2
MVMI addr1
XRA  L
MVMO addr2
MVMI addr1
NOP
MVMO addr2
MVMI addr1
XCHG
MVMO addr2
```

A.2.6 Detection of Decoding Faults in Group V

The testing of the Instruction Decoding Faults in this group is simplified by the fact that all the instructions in this group are conditional CALL instructions. Whether a CALL is executed or not depends upon the condition of the various bits in the status register of the microprocessor. For example, in response to instruction CC, a CALL is executed if and only if the carry bit of the status register is 1. To test decoding faults in this group, we execute the following sequence:

Sequence A.2.9

```
MVMI addr1
CC
MVMO addr2
MVMI addr1 ; change the 12th data byte to
             56H to complement the carry bit.
CC           Data in the remaining byte is
             the same.
MVMO addr2
MVMI addr1
CM
MVMO addr2
MVMI addr1 ; change the 12th data byte to
             D7H to complement the sign bit.
CM
MVMO addr2
MVMI addr1
CNC
MVMO addr2
MVMI addr1
CNZ
MVMO addr2
MVMI addr1 ; change the 12th data byte to
             17H to complement the zero bit.
CNZ
MVMO addr2
MVMI addr1
CP
MVMO addr2
MVMI addr1 ; change the value of the sign bit
             in the 12th data byte.
CP
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1. (except for the changes noted.)

Sequence A.2.9

(Continued)

```
MVMO addr2
MVMI addr1
CPE
MVMO addr2
MVMI addr1 ; change parity of bits by chang-
CPE          ing 1st, 3rd, or 5th bits.  Note
             that these bits of the flag regis-
             ter are unassigned
MVMO addr2
MVMI addr1
CPO
MVMO addr2
MVMI addr1 ; change parity of bits in the
             12th byte
CPO
MVMO addr2
MVMI addr1
CZ
MVMO addr2
MVMI addr1 ; change the zero bit in the 12th
             byte
CZ
MVMO addr2
MVMI addr1
CNC
MVMO addr2
MVMI addr1 ; change the carry bit in the 12th
             byte
CNC
MVMO addr2
```

COMMENTS: It is assumed that each MVMI instruction in this sequence is executed with the same operands as the first MVMI instruction of A.2.1. (except for the changes noted.)

It can be shown that the sequence above detects all the instruction decoding faults in Group V. Consider for example, instructions 1 through 6 of the above sequence. If no fault of the type f(CC/I) is present, then a CALL would be executed in step 2; whereas no CALL would be executed in step 5. However, if for example a faultf(CC/CPO) exists, a CALL would not be executed in either step 2 or step 5 of the above sequence. (Since the parity of the bits stored in the accumulator is now even.) This deviation from the fault free behavior would be read out by the MVMO instruction in step 6. However, even if the parity of the bits stored in the accumulator was odd, the fault would still be detected because CALL instruction would be executed, both in step 2 and 5 of the above sequence. The same reasoning may be used to show that the sequence above detects all faults of the type $f(I_j/I_k)$ provided I_j and I_k both belong to Group V.

A.2.7 Detection of Decoding Faults in Group VI

All the instructions in this group (except the last two) are conditional JUMP instructions. The nature of these instructions is very similar to the instructions of Group V, where all the instructions were conditional CALL's. Using the reasoning of Group V, it can be shown that the following sequence detects all the instruction decoding faults in Group VI.

Sequence A.2.10

```
MVMI addr1
JC
MVMO addr2
MVMI addr1 ; change the carry bit in the
             12th data byte
JC
MVMO addr2
```

Sequence A.2.10

(Continued)

```
MVMI addr1
JM
MVMO addr2
MVMI addr1 ; change sign bit in the 12th
             data byte
JM
MVMO addr2
MVMI addr1
JNC
MVMO addr2
MVMI addr1 ; change carry bit in the 12th
             data byte
JNC
MVMO addr2
MVMI addr1
JNZ
MVMO addr2
MVMI addr1 ; change zero bit in the 12th
             data byte
JNZ
MVMO addr2
MVMI addr1
JP
MVMO addr2
MVMI addr1 ; change parity in the 12th data
             byte
JP
MVMO addr2
MVMI addr1
JPE
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of this sequence A.2.1. (except for the changes noted.)

Sequence A.2.10

(Continued)

```
MVMO addr2
MVMI addr1 ; change parity of the 12th data
             byte
JPE
MVMO addr2
MVMI addr1
JPO
MVMO addr2
MVMI addr1 ; change parity of the bits in
             the 12th data byte
JPO
MVMO addr2
MVMI addr1
JZ
MVMO addr2
MVMI addr1 ; change zero bit in the 12th
             data byte
JZ
MVMO addr2
MVMI addr1
LXI, B
MVMO addr2
MVMI addr1
LXI, D
MVMO addr2
MVMI addr1
LXI, SP
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of this sequence A.2.1. (except for the changes noted.)

Sequence A.2.10 (Continued)

```
JMP
MVMO addr2
MVMI addr1 ; complement all bits of the flag
             register
JMP
MVMO addr 2
```

A.2.8 Detection of Decoding Faults in Group VII

The test sequence to detect faults for the first four instructions in this group can be generated using the method that was used for the instructions in Group I. The rest of the instructions in this group are conditional RETURN instructions. The test sequence to detect faults for conditional RETURN instructions is generated in the same manner as for the instructions in Group IV. The test sequence may be written as:

Sequence A.2.11

```
MVMI addr1
DCX  B
MVMO addr2
MVMI addr1
DCX  D
MVMO addr2
MVMI addr1
DCX  H
MVMO addr2
MVMI addr1
DCX  SP
MVMO addr2
MVMI addr1
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of sequence A.2.1. (except for the changes noted.)

Sequence A.2.11 (Continued)

```
INX  B
MVMO addr2
MVMI addr1
INX  D
MVMO addr2
MVMI addr1
INX  H
MVMO addr2
MVMI addr1
INX  SP
MVMO addr2
MVMI addr1
PCHL
MVMO addr2
MVMI addr1
SPHL
MVMO addr2
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of sequence A.2.1. (except for the changes noted.)

A.2.9 Detection of Decoding Faults in Group VIII

Since all the instructions in this group are conditional RETURN instructions, the procedure for test generation is very similar to that for Group V., where all the instructions were conditional CALL's. Therefore, the test sequence is:

Sequence A.2.12

```
MVMI addr1
RC
MVMO addr2
```

Sequence A.2.12

(Continued)

```
MVMI addr1; change carry bit in the 12th
           byte
RC
MVMO addr2
MVMI addr1
RM
MVMO addr2
MVMI addr1; change sign bit of the 12th
           data byte
RM
MVMO addr2
MVMI addr1
RNC
MVMO addr2
MVMI addr1; change carry bit of the 12th
           data byte
RNC
MVMO addr2
MVMI addr1
RNZ
MVMO addr2
MVMI addr1; change zero bit of the 12th
           data byte
RNZ
MVMO addr2
MVMI addr1
RP
MVMO addr2
MVMI addr1; change parity of the 12th data
           byte
RP
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of sequence A.2.1. (except for the changes noted.)

Sequence A.2.12 (Continued)

```
MVMO addr2
MVMI addr1
RPE
MVMO addr2
MVMI addr1 ; change parity of the 12th data byte
RPE
MVMO addr2
MVMI addr1
RPO
MVMO addr2
MVMI addr1 ; change parity of the 12th data byte
RPO
MVMO addr2
```

COMMENTS: It is assumed that all the MVMI instruction in the sequence is executed with the same operands as the first instruction of sequence A.2.1. (except for the changes noted.)

It will be noted that to generate the above test sequences, instructions RIM, SIM, EI, DI and HLT were not considered. In order to detect faults in these instructions the program exeuction in the microprocessor must be interrupted. It is easy to see how the decoding faults for these instructions can be detected. As an example, consider the instruction SIM (Set Interrupt Masks). The execution of the SIM instruction uses the contents of the accumulator (which must be previously loaded) to set or clear the various interrupts on the 8085 microprocessor. In order to detect the faults covered by the Assumption 4.3.3 (i) and (ii) in instruction SIM (SIM belongs to group of the EPI instructions), only the signals put out at the external pins in response to SIM need to be monitored. However, the

faults covered by Assumption 4.3.4 can be detected only by executing SIM, and observing the response of the microprocessor to various interrupts.

A.3 Detection of Data Transfer and Data Storage Faults

Data Storage and Data Transfer faults in the 8085 are detected using the Algorithm 4.2.1 given in Chapter IV. Since it has also been shown how the MOVE MULTIPLE instructions used in this algorithm can be implemented on an 8085 microprocessor, it is not considered necessary to repeat the algorithm here. However, it may be pointed out that these faults can also be detected using the procedures of Thatte and Abraham.

A.4 Detection of Data Manipulation Faults

A detailed description of the various functional units of the 8085 microprocessor such as the ALU, or the interrupt handling hardware is needed to generate the tests for the Data Manipulation Function. During the course of the present research, several efforts were made to obtain a gate level description of the 8085 microprocessor from the Intel Corporation and from other companies that manufacture this microprocessor. The gate level diagrams for the microprocessor are regarded as classified information by the companies and therefore could not be obtained. Actual tests for testing the Data Manipulation Faults for the 8085 microprocessor therefore cannot be given here. However, these can be detected using the technique stated in Algorithm 4.4.2.