Chapter 4. Testing Techniques

4.1 Introduction

Testing of a system is a procedure where the system is exercised and its resulting response is analyzed to ascertain whether it produced correct results. If incorrect results are detected, a second goal of the testing procedure is, diagnosis, to identify the faulty module in the system. Diagnosis assumes knowledge of the internal structure of the system under test. The defects which occur in the system could either be caused during manufacture or be caused by wear and tear in the field. Testing is done at various stages in the production of a system, i.e. dies are tested during fabrication, the packaged chips are tested before insertion into the boards, the boards are tested after assembly, and the entire system is tested when complete.

Testing can be broadly divided into three distinct areas :

1. **Design Tests :** These are used to validate that the model under test meets its specifications and will operate properly in the system or, in other words, these tests check the functionality of the model. The effort spent to generate these tests is comparable to the effort spent in the component design itself [3]. System stimuli generated for high level simulation can form the basis for the final system test; however these tests may not be helpful in identifying faulty components.

2. **Manufacturing tests** : These tests are employed during the system assembly to identify defective components, boards and subsystems for replacement.

3. **Diagnostic and maintenance tests** : These are used to isolate faults to units that are replaceable in the field or at the repair shop. Once a system has been developed, it needs to be tested before being shipped to the customer. This test is usually a high-speed test where the

system is checked to ensure that it performs as expected under normal conditions. When the system is in the field, it has to be maintained in operating order, so it is periodically tested to ensure that it has no failures. If the system fails in the field, tests are run to determine the failed part. This type of testing is usually called diagnosis. The development of maintenance strategy involves three steps :

- Tests must be developed to isolate faults to replaceable units.
- A fault dictionary relating test outputs to defective units must be constructed.
- An assessment of tests in identifying all possible failures must be performed.

This thesis mainly focuses on only design and maintenance tests. Manufacturing tests are beyond the scope of the present research work.

Another important facet in the system development process is test evaluation, which refers to determining the effectiveness or quality of a test. Test evaluation is usually done in the context of a fault model, and the quality of a test is measured by the ratio between the number of faults it detects and the number of faults which could possibly exist in the system. This ratio is called fault coverage. Testing methods should be so developed that we achieve a high fault coverage. Test evaluation is carried out via a simulated testing experiment called fault simulation, which computes the response of a circuit in the presence of faults to the test being evaluated. A fault is detected when the response it produces differs from the expected response of the fault-free circuit. Expected response can be generated during testing from a known good copy of the MUT - the so called ***GOLD*** model.

Yet another important aspect of the testing process is test generation. It is the process of determining the stimuli necessary to test a system. Test generation for design verification testing and diagnosis involve a large effort and is often a manual activity.

The subsequent sections discuss examples of memory testing , regression testing and diagnostic testing.

4.2 Memory Testing

4.2.1 Introduction

Semiconductor memory occurs frequently in large scale integration (VLSI) products. They can be divided into two types. One is a read-and-write type and the other is a read-only type of memory. We generally call the read-and-write type a RAM or random-access-memory and the read-only-type a ROM. The RAM stores information in a memory cell array; each bit of information being typically stored in a single transistor cell. The address of each cell is typically provided to the decoders which activate the appropriate select lines.

Test procedures for a RAM can be divided into two broad areas.

- Parametric Testing
- Functional Testing

Parametric testing involves measuring DC parameters such as output levels, power consumption, fan-out capability, leakage current and noise margins. It also involves AC parameters such as dynamic behavior, setup and hold times, access time and recovery times. Testing for these parameters requires detailed knowledge of the chip as well as the availability of test equipment which has the ability to apply the test patterns and to capture the responses at the required speeds and with necessary resolution. This method of testing is beyond the scope of this present research work.

Functional testing involves detection of physical failures which cause the RAM to function incorrectly, e.g., faults in memory cells, address logic, sense amplifiers, write drivers, noise coupling between cells, etc. In the most general sense, a memory can be defined to be functional if it is possible to change every cell from a 0 to a 1 as well as from a 1 to a 0, and to read every cell correctly when it stores a 0 as well as when it stores a 1 independent of the state of the remaining cells.

Unfortunately if we wish to check every cell in the memory for all possible states, the number of operations required would be of the order of 2^n,where n is the number of cells in memory. Therefore a useful functional test of reasonable length would be based on a particular subset of faults which would account for most of the failures associated with memories.

Common failure modes that could exist in a memory are detailed below:

- One or more cells are stuck-at-zero or stuck-at-one. These faults are called ***stuck-at-faults***. In this case a cell will remain stuck at a particular state irrespective of reads or writes to any cell in memory.

- One or more cells fail to undergo a $(0 \rightarrow 1)$ **and/or** $(1 \rightarrow 0)$ transition when the complement of the contents of a memory cell is written to the respective cell. These faults are known as ***transition faults***.

- Two or more cells are accessed during a Read/Write operation. These faults are called ***multiple access faults***. During a READ operation, at some address, if more than one cell is accessed then the output is the AND or OR of the contents of the cells accessed. The logic operation of AND or OR is fixed for a given RAM by the technology and the details of its logic circuitry.

- Two or more cells are coupled. In this case a ($0 \rightarrow 1$ or $1 \rightarrow 0$) transition in a cell, due to a write operation changes the contents of not only the present cell but also another cell in the memory. These faults are called ***coupling faults***.

4.2.2 Methodology

Generally, the memory module could have one or more of the faults listed in section 4.2.1. The testing procedure should effectively test for all of the faults. The methodology adopted to illustrate memory testing is detailed below :

1. First, several faulty architectures of the memory module are developed by embedding one or more of the faults listed in section 4.2.1.

2. A test bench for testing the faulty architectures is then developed. The memory component is tested with several test patterns and its output response is compared with a GOLD model response in order to evaluate the correctness of the outputs. An output report is generated which records the status of each memory cell as it is compared with the expected response. If the cell is faulty, an ***"ERROR"*** is reported, else a ***"PASS"*** is reported.

3. The test patterns need to be carefully chosen while testing the memory component, since not all faults can be detected with a particular set of test patterns. Hence an efficient method of testing would be to apply more than one set of test patterns to ensure that the component is functional.

4. Lastly, a configuration declaration for selecting the different architectures is developed. It is used to bind the entities to the instances of the components declared in the test bench and also allows the user to fill in the values of the generic constants.

4.2.3 Testing Procedure

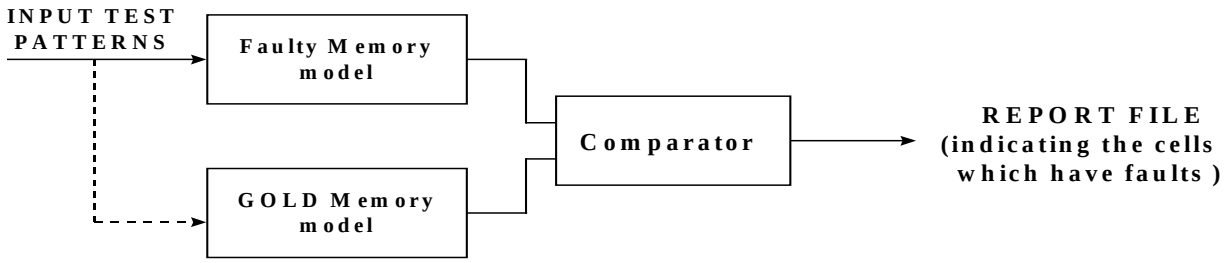

Figure 4.1 Test setup for Memory testing

In order to demonstrate the above methodology, five different faulty architectures of the memory component used in the Sobel Edge detector were created, with a single fault introduced in four of the architectures and multiple faults in the fifth architecture. The four different faults introduced in the memory architecture are the ones listed in section 4.2.1.

Figure 4.1 shows the test setup for Memory testing. A set of input test vectors are applied to the memory and its response is compared with the response of the GOLD model. The comparator then generates a report, which records the status of all the memory cells.

Let us consider the memory model shown in Figure 4.2. Figure 4.3 shows the entity declaration. The first generic constant, ***MEM_OUT_DELAY***, specifies the time after which the memory outputs are transferred to the window processor. The two generic constants ***NUM_COLS*** and ***N*** specify the number of columns in the image file and the length of the Y address in bits respectively. The ports of the memory module include ***CLOCK*** and ***RWB*** for the system clock input and for READ/WRITE signals respectively. Also included are four signals ***X_ADDR1, X_ADDR2, X_ADDR3, Y_ADDR*** for addressing the various cells in the memory and three output signals ***MEM_OUT1, MEM_OUT2*** and ***MEM_OUT3*** which are used to sent out data to the window processor.

The behavioral architecture for the faulty memory module with multiple faults is shown in Figure 4.4. An array type called ***MEMORY_ARRAY*** is defined, which specifies the size of the internal memory. Internal memory has three rows and the same number of columns as the input image.

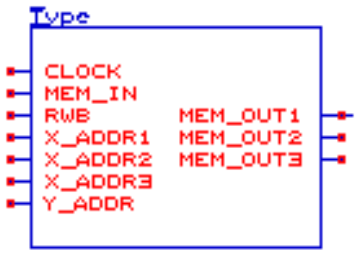

Figure 4.2 Memory module

Next we have a process which declares a variable ***MEM*** of the array type and ***FAULT CELL*** which is of data type std_logic_vector(7 downto 0). At the rising edge of the clock, if the ***RWB*** signal is low, then the input image data is loaded into the memory cells. Various faults have been introduced in many of the memory cells, hence values different from the true input image data may be written into the memory cells. Once the ***RWB*** signal goes high, then data is read from the memory. Again some faults have been introduced which cause the wrong outputs to be sent out to the window processor.

```
 entity MEMORY is
  generic(MEM_OUT_DELAY: TIME;                    -- Memory output delay
          NUM_COLS: NATURAL;                      -- number of columns of the input image file
          N: INTEGER);                            -- length of the Y Address vector
  port ( CLOCK: in STD_LOGIC:='0';                -- system CLOCK
          RWB: in STD_LOGIC:='0';                 -- READ/(NOT)WRITE signal
       MEM_IN : in PIXEL:="00000000";             -- input PIXEL
       X_ADDR1: in STD_LOGIC_VECTOR(1 downto 0):="01";    -- X address 1
       X_ADDR2: in STD_LOGIC_VECTOR(1 downto 0):="01";    -- X address 2
       X_ADDR3: in STD_LOGIC_VECTOR(1 downto 0):="01";    -- X address 3
       Y_ADDR:  in STD_LOGIC_VECTOR(N-1 downto 0):=INT_TO_STDLOGICN(1,N);-- Y address
       MEM_OUT1: out PIXEL:="00000000";           -- Memory unit output 1
       MEM_out2: out PIXEL:="00000000";           -- Memory unit output 2
       MEM_out3: out PIXEL:="00000000");          -- Memory unit output 3
end MEMORY;
```

Figure 4.3 Entity declaration of memory module

```
 architecture MULT_F of MEMORY is
type MEMORY_ARRAY is array(1 to 3, 1 to NUM_COLS) of PIXEL;
begin
  process
    variable MEM:MEMORY_ARRAY;
    variable FAULT_CELL:PIXEL:="00000000";
begin
-------------- WRITE operation ------------------------------------------------
          wait until rising_edge(CLOCK);
          if RWB='0' then
--------------------------------------------------------------------------------
--INTRODUCING STUCK_AT_ZERO FAULT--
          if X_ADDR1="11" and Y_ADDR=INT_TO_STDLOGICN(2,N) then
                    FAULT_CELL:= MEM_IN;
          FAULT_CELL(1):='0';
                    MEM(STDLOGIC_TO_INT(X_ADDR1),STDLOGIC_TO_INT(Y_ADDR)):= FAULT_CELL;

--INTRODUCING CELL INTERACTION FAULTS--

          elsif
               X_ADDR1="01" and Y_ADDR=INT_TO_STDLOGICN(4,N) then
            MEM(1,3):=MEM_IN;
            MEM(1,4):=MEM_IN;
 --INTRODUCING STUCK_AT_ONE FAULT--
          elsif
                    X_ADDR1="10" and Y_ADDR=INT_TO_STDLOGICN(4,N) then
                    FAULT_CELL:= MEM_IN;
           FAULT_CELL(2):='1';
                    MEM(STDLOGIC_TO_INT(X_ADDR1),STDLOGIC_TO_INT(Y_ADDR)):= FAULT_CELL;
--------------------------------------------------------------------------------------------------------------------------------
      elsif not(((X_ADDR1="01") and (Y_ADDR=INT_TO_STDLOGICN(4,N))) or
```

```
            ((X_ADDR1="11") and (Y_ADDR=INT_TO_STDLOGICN(2,N))) or
            ((X_ADDR1="01") and (Y_ADDR=INT_TO_STDLOGICN(3,N))) or
            ((X_ADDR1="10") and (Y_ADDR=INT_TO_STDLOGICN(4,N)))) then
          MEM(STDLOGIC_TO_INT(X_ADDR1),STDLOGIC_TO_INT(Y_ADDR)) := MEM_IN;
    end if;
end if;

-------------- READ operation-------------------------------------------------------------------------------------------
wait until falling_edge(CLOCK);
if RWB='1' then

-------INDIVIDUAL ADDRESS ACCESS FAULTS-----------------------------------------------------------------------------
if X_ADDR1="01" and Y_ADDR=INT_TO_STDLOGICN(2,N) then
        MEM_OUT1 <= MEM(STDLOGIC_TO_INT(X_ADDR1),STDLOGIC_TO_INT(Y_ADDR)) and
                        MEM(STDLOGIC_TO_INT(X_ADDR1)+1,STDLOGIC_TO_INT(Y_ADDR)+1)
        after MEM_OUT_DELAY;
        else
                MEM_OUT1 <= MEM(STDLOGIC_TO_INT(X_ADDR1),STDLOGIC_TO_INT(Y_ADDR))
                        after MEM_OUT_DELAY;
end if;

        MEM_OUT2 <= MEM(STDLOGIC_TO_INT(X_ADDR2),STDLOGIC_TO_INT(Y_ADDR))
                                after MEM_OUT_DELAY;
        MEM_OUT3 <= MEM(STDLOGIC_TO_INT(X_ADDR3),STDLOGIC_TO_INT(Y_ADDR))
                                after MEM_OUT_DELAY;
    end if;
end process;
end MULT_F;
```

Figure 4.4 Behavioral architecture of the faulty memory model.

The test bench model for testing the memory component has been developed on similar lines as that developed for the sobel edge detector shown in Section 3.2.2.4. The entire source code for the test bench module can be found in Appendix B.

Different sets of test vectors were used to detect all the possible faults in the model. In order to test the memory model we used test patterns of all 0's or all 1's or a combination of both to detect all the possible faults. The report generated with an input pattern of all 0's is shown in Figure 4.5. By carefully observing the output response, we see that with this input test pattern, only one of the faults i.e.; stuck-at-1 fault was detected. Hence we tested the model again with an input combination shown in Figure 4.6.

MEMORY CELL: 1,1 PASS
MEMORY CELL: 1,2 PASS
MEMORY CELL: 1,3 PASS
MEMORY CELL: 1,4 PASS
MEMORY CELL: 2,1 PASS
MEMORY CELL: 2,2 PASS
MEMORY CELL: 2,3 PASS
MEMORY CELL: 2,4 ERROR
MEMORY CELL: 3,1 PASS
MEMORY CELL: 3,2 PASS
MEMORY CELL: 3,3 PASS
MEMORY CELL: 3,4 PASS

Figure 4.5 Error report generated with a test pattern of all 0's in the input image.

The report generated with this pattern is shown in Figure 4.7. It is observed that with this pattern, the remaining faults have been detected. Hence we find that in order to detect all the faults in a component, we may need to apply several test patterns to the component.

0 255 0 255
0 255 0 255
0 255 0 255
0 255 0 255
0 255 0 255
0 255 0 255

Figure 4.6 Input Test Pattern - Combination of 1's and 0's.

MEMORY CELL: 1,1 PASS
MEMORY CELL: 1,2 ERROR
MEMORY CELL: 1,3 ERROR
MEMORY CELL: 1,4 PASS
MEMORY CELL: 2,1 PASS
MEMORY CELL: 2,2 PASS
MEMORY CELL: 2,3 PASS
MEMORY CELL: 2,4 PASS
MEMORY CELL: 3,1 PASS
MEMORY CELL: 3,2 ERROR
MEMORY CELL: 3,3 PASS
MEMORY CELL: 3,4 PASS

Figure 4.7 Error report generated with a test pattern shown in Figure 4.6

A configuration declaration was used to select the different faulty memory architectures. A portion of the source code to select the multiple faulty memory module is shown in Figure 4.8.

```
 library STRUC_STD;
use STRUC_STD.IMAGE_PROCESSING.all;
use STRUC_STD.all;

configuration CONFIG_B of TB_CONFIG is
for TEST_BENCH
```

```
for con:TEST1 use entity WORK.TEST(BENCH)
generic map("test_im.dat","test_mg.dat","errf_f3.dat",10 ns,6,4,3);

        for BENCH

        for P1:CLOCK_GENERATOR1 use entity BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
        generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
        end for;

        for P2:MEMORY_PROCESSOR1 use entity STRUC_STD.MEMORY_PROCESSOR(STRUCTURE)
        generic map (NUM_ROWS=>6,NUM_COLS=>4,MEM_OUT_DELAY=>10 ns,N=>3);

      for STRUCTURE

            for ADDR:ADDR_GEN1 use entity STRUC_STD.ADDR_GEN(BEHAVIOR)
            generic map(ADDR_A_DELAY=>2 ns, RWB_A_DELAY=>10 ns,
                  NUM_ROWS=>6,NUM_COLS=>4,N=>3)
            port map(START,CLOCK,X_ADDR1,X_ADDR2,X_ADDR3,Y_ADDR,RWB);
            end for;

            for MEM: MEMORY1 use entity STRUC_STD.MEMORY(MULT_F)
            generic map(MEM_OUT_DELAY=>10 ns, NUM_COLS=>4,N=>3)
            port map(CLOCK,RWB,MEM_IN,X_ADDR1,X_ADDR2,X_ADDR3,
            Y_ADDR,MEM_OUT1, MEM_OUT2, MEM_OUT3);
            end for;
      end for;
      end for;
 end for;
end for;
end for;
end;
```

Figure 4.8 Configuration declaration for the multiple fault memory model.

This configuration declaration is similar to the one explained in Section 3.2.2.6. Configuration specifications are used to select the Clock generator*(BEH_INT.CLOCK_GENERATOR (BEHAVIOR))*, memory processor*(STRUC_STD.MEMORY_PROCESSOR(STRUCTURE))*, address generator*(STRUC_STD.ADDR_GEN(BEHAVIOR))* and memory *STRUC_STD.MEMORY(MULT_F)*.

Generic values have been supplied for all of the constants at the top level, namely (WORK.TEST(BENCH)). File names have been specified for the input image(test_i4.dat), GOLD model(test_gm.dat) and report file(errf_f3.dat). Also the size of the input image(*NUM_ROWS=>6, NUM_COLS=>4)* and READ/WRITE delay (10 ns), address generator delay(2 ns), memory output delay(10 ns) and length of Y Address(4) values have been supplied. Likewise for the Clock generator HI_TIME(75 ns) and LO_TIME(25 ns) values have been specified.

Figure 4.9 shows a table which summarizes the results for testing of the faulty memory module shown in Figure 4.4 for different test patterns.

Test Results :

As seen in the faulty memory module shown in Figure 4.4, faults were embedded in cells (1,2), (1,3), (2,4) and (3,2).The table shown below summarizes the test results.

INPUT TEST PATTERN	TEST RESULT
PATTERN OF ALL 0's	Only the fault in cell (2,4) is detected
PATTERN OF ALL 1's	Only the fault in cell (3,2) is detected
PATTERN SHOWN IN FIGURE 4.6	Faults in cells (1,2),(1,3) and (3,2) are detected

Figure 4.9 Test results for Memory Testing

Hence from the test results we observe that we need more than one test pattern to detect all the faults embedded in the memory module.

4.3 Regression Testing

4.3.1 Introduction

Regression testing is an expensive but necessary maintenance activity wherein we replace a certain component with a modified version and test the system to ensure that the new component does not adversely affect the overall system performance. It is a process of validating modified components of the system and ensuring that no new errors are introduced into the previously tested system. A regression test selection technique chooses, from an existing test set, tests that are deemed necessary to validate the system with the modified component. After changes are made to a previously tested system, a goal of regression testing is to test the system with the modifications while maintaining the same testing coverage as achieved with the previous version of the system. An important difference between regression testing and development testing is that during regression testing an established set of tests may be reused.

4.3.2 Methodology

In this section the methodology adopted for regression testing is explained. The main advantage of regression testing is that it facilitates model-year upgrades. During the process of design development and testing, we often would like to test the system in the following scenarios:

- When a particular sub-module at the behavioral level is replaced with its modified version at a lower level say RTL or gate level.

- When two different architectures for a sub-module are developed, then we need to ensure that the system generates the same response with both of the sub-modules.
- When a particular sub-module of a different data type say std_logic is inserted into the system with integer data types.
- When a component is upgraded to a newer technology. For example a 100 MHz computer may be replaced by a 200 MHz computer.

In order to test for the above mentioned scenarios in a simple and efficient way, a method of testing is required where we have to simply replace the modified sub-component in the system, and simulate the system without modifying any of the other components in the system.

Again this can be achieved by the use of a configuration declaration, where we can simply replace the entity, architecture and library name in the configuration specification statement for the modified component and simulate the whole system with the same set of test vectors and test bench module as used before. The output response of the simulation can then be checked against the expected response to ensure the correctness of the results.

4.3.3 Testing Procedure

The methodology explained in Section 4.3.2 can be illustrated by considering the following scenario.

Two memory components were developed for the Sobel edge detector. The first was a custom memory module designed specifically for the Sobel edge detector (ASIC design) and the second was a memory module built from commercially available components (COTS design).

The Sobel Edge detector was first tested with the ASIC memory module and its results were verified. The next step was to test the MUT with the COTS memory module and see how this new model would affect the overall performance of the system. Figure 4.10 shows a portion of the source code of the configuration declaration for the Sobel edge detector. This configuration is similar to the one shown in Figure 3.12.

```
configuration CONFIG_CODE3 of TB_CONFIG is

    for MEM:MEMORY1 use entity STRUC_STD.MEMORY(STANDARD)
    generic map(MEM_OUT_DELAY=>MEM_OUT_DELAY,
                NUM_COLS=> NUM_COLS, N=>N)
    port map(CLOCK,RWB,MEM_IN,X_ADDR1, X_ADDR2, X_ADDR3,
            Y_ADDR, MEM_OUT1, MEM_OUT2, MEM_OUT3);
    end for;
```

Figure 4.10 Configuration declaration for Regression testing

In the configuration specification statement for the memory system shown in bold, we replace the architecture name of the ASIC memory module with that for the new COTS memory module and simulate the system with the same test set and same test bench module.

The output response of the new memory architecture is verified against the expected response. Figure 4.11 shows the simulation report which was generated for both the tests. Since the same report was generated for the tests, it was verified that the new model (one built with COTS components) did not affect the performance of the system.

Assertion ERROR at 100 NS in design unit TEST(BENCH) from process
*/TB_CONFIG/CON/MEMORY1: "**array read in**"*

Assertion ERROR at 10600 NS in design unit TEST(BENCH) from process
*/TB_CONFIG/CON/MEMORY1: "**magnitude and direction outputs written to file**"*

Assertion ERROR at 10700 NS in design unit TEST(BENCH) from process
*/TB_CONFIG/CON/MEMORY1: "**MAGNITUDE VALUES MATCH -- PASS**"*

Assertion ERROR at 10700 NS in design unit TEST(BENCH) from process
*/TB_CONFIG/CON/MEMORY1: "**DIRECTION VALUES MATCH -- PASS**"*

Figure 4.11 Simulation results with the custom memory module.

4.4 Diagnostic Testing

4.4.1 Introduction :

A system fails when its observed behavior is different from its expected behavior. If the system produces incorrect results, the cause of the observed error(s) must be diagnosed. Diagnosis consists of locating the faulty sub-components in a structural model of a system. The degree of accuracy to which faults can be located is referred to as diagnostic resolution. No testing experiment can distinguish among functionally equivalent faults. The partition of all the possible faults into distinct sets of functionally equivalent faults defines the maximum fault resolution of the system. The fault resolution of a test sequence reflects its capability of distinguishing among faults, and it is bounded by the maximum fault resolution. A test sequence that achieves the maximal fault resolution is said to be a complete fault-location test. Replacing the MUT often consists of substituting one of its replaceable units identified as containing some faults by a good unit. Hence most often we are interested in locating the fault rather than in the accurate identification of the fault inside a module.

4.4.2 Methodology for Diagnostic testing

When a system is tested, its resulting response is analyzed to ascertain whether it behaved correctly. If incorrect behavior is observed, then the second step is to ***diagnose*** or locate the cause of the misbehavior. This section illustrates the methodology adopted for diagnosis.

The MUT is first tested with an input test pattern and its output response compared with the expected response. If the output response does not match with the expected response, then we know that the MUT is faulty. Now either one or more of the components of the MUT could be faulty. In order to locate the faulty components, we need to apply several test patterns, until we observe an output response pattern typical of a particular faulty component.

Once the faulty component is located, it can be replaced with a functional, error-free one. The MUT is then tested again. If the output of the MUT matches with the expected response, then we can conclude that the MUT is functional, else we need to determine the next component to be replaced. In order to do this, we apply several test patterns, until we can again observe an output typical of any of the remaining faulty components.

After replacing the next faulty component with an error-free one, we again test the MUT and repeat the procedure detailed above until the outputs match with the expected response.

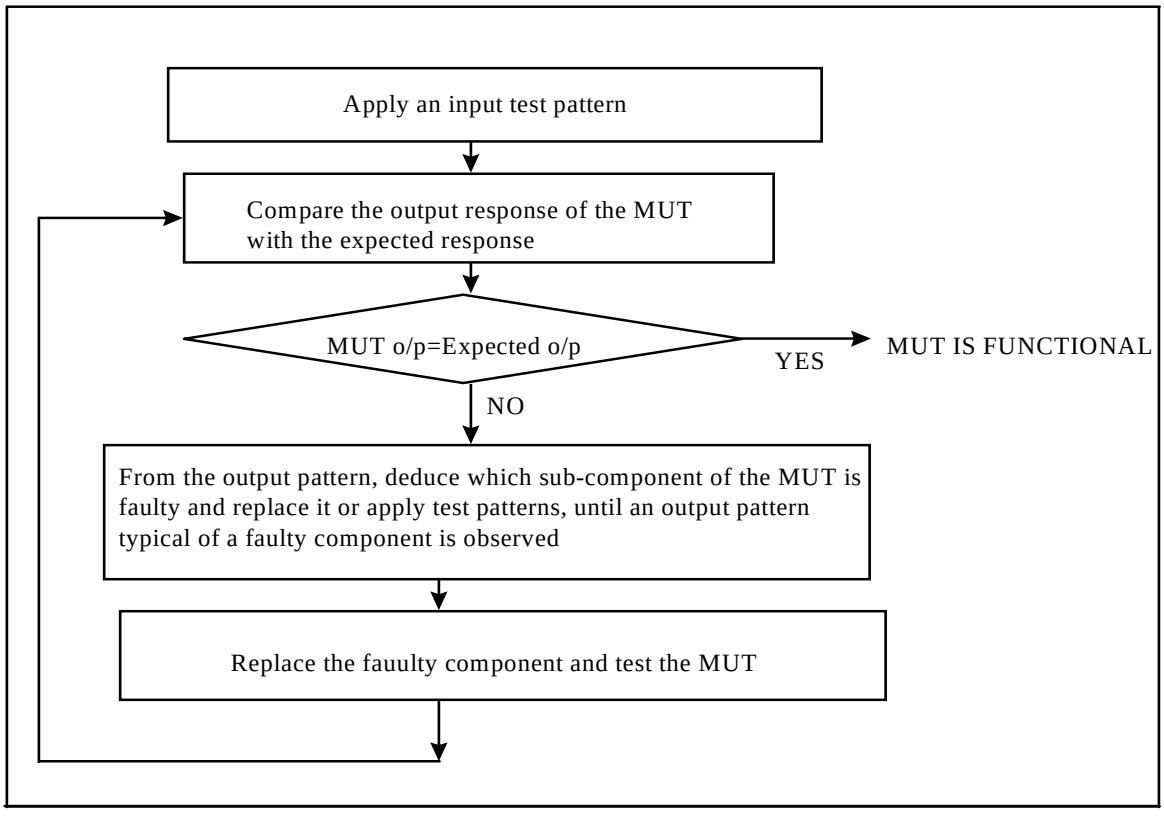

Figure 4.12 Flowchart for diagnostic testing

The most difficult task in this method of testing is to determine the input test patterns to be applied to the MUT. Not all test patterns are useful for diagnosis. Hence it is imperative that we test the MUT with several test patterns before we conclude that it is functional. A flowchart for diagnostic testing is shown in Figure 4.12.

4.4.3 Testing Procedure

In order to illustrate the methodology in Section 4.4.2, let us consider the test setup for diagnostic testing of the Sobel edge detector as shown in Figure 4.13.

Figure 4.13 Test Setup for Diagnostic Testing

Figure 4.13 shows the MUT(Sobel edge detector) which consists of three components namely the memory system, window processor and magnitude processor. Faults were embedded in the sub-components of the memory processor and the window processor. The source code for the faulty architectures can be found in Appendix B.

The testing procedure is now illustrated for diagnostic testing.

Step 1: The pattern shown in Figure 4.14 was first applied to the MUT.

```
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 255 255
```

Figure 4.14 Input Image Pattern

With an error-free system or the GOLD model, the output response for the input pattern shown in Figure 4.14 is determined to be of the form shown in Figure 4.15 for a threshold value between "000" and "3F8" and for a threshold between "3F9" and "FFF", the output response would be as shown in Figure 4.16.

0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	255	255	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255

Figure 4.15 Expected Response for threshold value set between "000" and "3F8".

0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255

Figure 4.16 Expected Response for threshold value set between "3F9" and "FFF".

The actual output pattern of the MUT observed with a threshold of “002” is shown in Figure 4.17.

0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
0	0	0	0	0	0	0	0	0	0	255	255	255	255	255	255	255	255	255	255

Figure 4.17 Output Response of the MUT for threshold value set at “002”.

This response when compared with the expected response in Figure 4.15 was found to be incorrect. Hence we know that the MUT is faulty, but at this point of the test, it is not possible to determine which of the sub-components of the MUT are faulty. To locate the faulty components some further tests must be performed on the MUT. The threshold was increased in steps until at the value of “00F”, the output pattern shown in Figure 4.18 was observed.

After testing the MUT with several test patterns, it was found that such a test pattern (highlighted in bold) was possible only if the memory component was faulty. If this form of response was not observed even after increasing the threshold to a very high value, it was found

that the incorrect behavior of the MUT is only due to faults in the window or magnitude processor or both of these components.

0	0	**0**	0	**0**	0	0	0	0	0	255	255	255	**255**	255	255	255	255	255	255
0	255	**0**	255	**255**	255	255	255	255	255	255	255	255	**0**	255	255	255	255	255	255
0	255	**255**	255	**255**	255	255	255	255	255	255	255	255	**255**	255	255	255	255	255	255
0	255	**255**	255	**0**	255	255	255	255	255	255	255	255	**255**	255	255	255	255	255	255
0	255	**0**	255	**255**	255	255	255	255	255	255	255	255	**0**	255	255	255	255	255	255
0	255	**255**	255	**255**	255	255	255	255	255	255	255	255	**255**	255	255	255	255	255	255
0	255	**255**	255	**0**	255	255	255	255	255	255	255	255	**255**	255	255	255	255	255	255
0	255	**0**	255	**255**	255	255	255	255	255	255	255	255	**0**	255	255	255	255	255	255
0	255	**255**	255	**255**	255	255	255	255	255	255	255	255	**255**	255	255	255	255	255	255
0	0	**0**	0	**0**	0	0	0	0	0	255	255	255	**255**	255	255	255	255	255	255

Figure 4.18 Output Response of the MUT for the threshold value set at "00F".

Step 2: Once the memory component was found to be faulty, it was replaced with an error-free module and the MUT was tested again with the threshold set at "00F". The output response was the same as that in Figure 4.17. Since the output response still did not match with the expected response in Figure 4.15, we have to replace either the window or magnitude processor. Deciding which of the two to replace is a difficult task, since faults in both sub-components produced similar output response patterns between a certain range of threshold values.

After performing several tests, it was found that 80% of the time, when such an output was observed, the window processor was found to be faulty. Hence the first choice was to replace the window processor with an error-free component and then run the tests again.

Step 3: After replacing the window processor, the same input pattern was applied to the MUT. This time the outputs of the MUT matched with the expected response shown in Figure 4.15. In order to ensure that the system worked perfectly, the MUT was tested further at higher values of the threshold signal. It was found that the output response of the MUT matched with the expected response even at threshold values set between "3F9" and "FFF" as shown in Figure 4.16. Hence it was concluded, that the MUT functioned correctly.

Diagnostic testing is a method by which we can easily locate a faulty sub-component of the MUT by observing the output test pattern. This is done by observing the output response pattern. Some output response patterns are distinct to faults in certain components. In this case, it is easy to identify the faulty sub-component. This was the case of the memory component in the Sobel edge detector which produced a distinct pattern if it was faulty. On the other hand, this method of testing could prove to be extremely cumbersome, since sometimes even after applying several test patterns, it is impossible to find the faulty sub-component. This was the case with the window processor and magnitude and direction processor, both of which produced similar outputs if they were faulty. In this case we need to replace the component which is most likely to be faulty. Hence we can conclude that the diagnostic testing method is not a deterministic method of testing. Figure 4.19 shows a flow chart diagram for testing the Sobel edge detector.

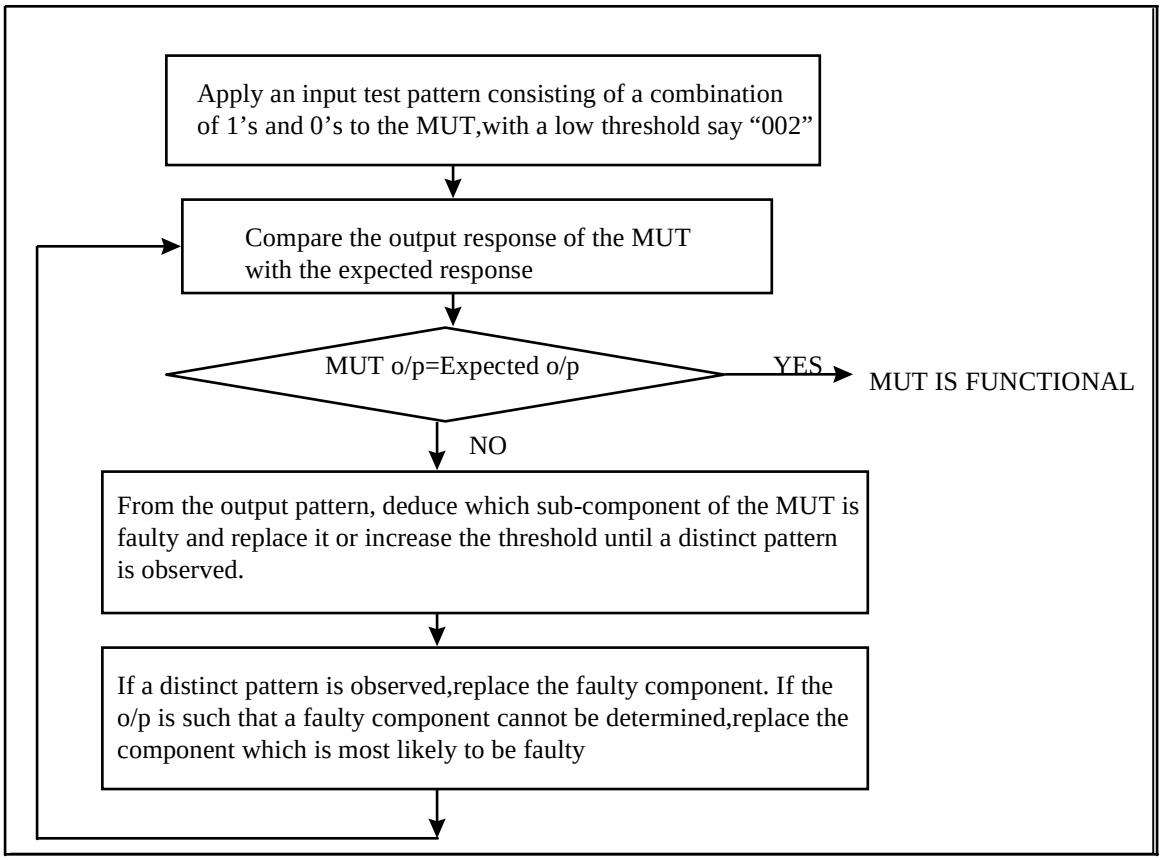

Figure 4.19 Flow chart for Diagnostic testing