TESTING FUNCTIONAL FAULTS IN VLSI*

by

Yinghua Min**
China Academy of Railway Sciences
Beijing, China

Stephen Y.H. Su
Department of Computer Science
State University of New York
Binghamton, New York 13901

ABSTRACT

Functional testing has become increasingly important due to the advent of VLSI technology. This paper presents a systematic procedure for generating tests for detecting functional faults in digital systems described by the register transfer language. Procedures for testing register decoding, instruction decoding, data transfer, data storage and data manipulation function faults in microprocessors are described step-by-step. Examples are given to illustrate the procedures.

I. INTRODUCTION

One of the reasons which makes the testing of VLSI very difficult is that the detailed implementation is unknown to the users of the IC (integrated circuit) chips because most of the IC manufacturers consider the circuit implementation of the IC chips proprietary. Therefore, we need to test VLSI based on whatever information is available to users in the manufacturer's data books and application notes from which the behavior of the IC chip is known. Functional testing deals with the detection and location of faults which change the behavior (function) of a digital system. Increasing attention and interest have been given to this area because of its importance. The objective is to find effecient procedures to generate tests automatically by a computer to test VLSI.

Recently, Su and Hsieh [1] have briefly discussed the prior work in functional testing area and outlined two approaches for testing functional faults in digital systems with the aid of register transfer language (RTL). Using a different approach, The first part of this paper considers the test generation for detecting permanent functional faults in digital systems whose behavior is described by register transfer language (RTL). Section II formally defines the standard statements in RTL by which a digital system including VLSI chips can be uniquely described.

In Section III, a procedure for generating tests for any given functional fault is presented by using the inverse operation of the RTL description of the system under test. Some examples are given to demonstrate the efficiency of the algorithm. In Section IV, test generation procedures for microprocessors are presented. Procedures for testing several types, instead of one type, of functional faults are given to simplify the testing.

II. SOME CONCEPTS AND NOTATIONS

Definition 1: A RTL statement is formally defined as

$$k:\ (T,C)\ R_D \leftarrow f(R_{s1},R_{s2}\ldots,R_{sv}),\ \rightarrow n \qquad (1)$$

where

- k - label for representing the RTL statement,
- T,C - time and conditions for executing the RTL statement,
- $R_D \leftarrow f(R_{s1},\ldots,R_{sv})$ - operation section of the statement,
- R_i - registers or data input or data output,
- R_D - destination register,
- $R_{s1},R_{s2},\ldots,R_{sv}$ - source registers,
- $\rightarrow n$ - jump section of the statement. If n=k+1, this section is omitted.

There are some special cases:

1. When f=∅ (empty), k becomes a jump statement,
2. When f=I (identity), $R_D \leftarrow I(R_{s1})$ means $R_D \leftarrow R_{s1}$, which is a register transfer statement.

Here data input and data output are considered as registers during the data flow. Once the CPU has been built using digital hardware components, the behavior of the CPU of a digital computer can be described by RTL. Any statement in RTL is a special case of the standard statement (1). For instance, for instruction fetch, the following three statements are involved:

* This work is supported by the Division of Mathematical and Computer Science, National Science Foundation under Grant No. MCS 78-24323 and Grant No. 8021262.

** Yinghua Min is now a Visiting Scholar in the Department of Computer Science, State University of New York, Binghamton, New York, 13901.

0420-0098/82/0000/0384$00.75 © 1982 IEEE

OUT ← PC

PC ← PC+1

IR ← IN

For an instruction execution cycle, we may consider for transfer instruction, $f(R_{s1},\ldots,R_{sv}) = R_{s1}$, for manipulation instruction, $R_D \leftarrow f(R_{s1},\ldots,R_{sv})(v\geq 0)$, for branch instruction, $PC \leftarrow f(R_{s1},\ldots,R_{sv})$.

A functional fault refers to the faulty execution of some statements which are called faulty statements.

We shall use k/k' to denote a functional fault where k denotes the fault-free statement, k' denotes the statement executed due to the functional fault. k' may be empty and needs not belong to any RTL statement for describing a given digital system.

Definition 2: A statement is called an input statement if for some i, $1\leq i\leq v$, R_{si} = IN (at least one source is the input bus). A statement is called an output statement if R_D=OUT (the destination is the output bus). A statement is called an I/O statement if it is an input and also an output statement.

Definition 3: The functional fault k/k' is detectable if there exists an executable statement sequence

$$w_1,\ldots,w_p,k,r_1,\ldots,r_q$$

where w_1 is an input statement, r_q is an output statement and all $w_1,\ldots,w_p$ are fault-free statements such that the sequence produced by k'

$$w_1,\ldots,w_p,k',r'_1,\ldots,r'_q$$

has the property $OUT(r_q) \neq OUT(r'_q)$. Statement r_q is called observable point of fault k/k'.

III. TESTING A GIVEN FUNCTIONAL FAULT

Definition 4: An executable sequence of statements

$$k,r_1,\ldots,r_q$$

is called a sensitizing sequence if r_q is an observable point of fault k/k' but points $r_1,\ldots,r_{q-1}$ are not.

Definition 5: Set K_{-1} = {statements whose jump section is "→k"} is called predecessor of k. Obviously, K_{-1} is non-empty, because at least $k-1 \in K_{-1}$, unless k itself is the first statement in the RTL description of a digital system under which k must be an input statement.

For detecting the functional fault k/k', we have to find an executable statement sequence $w_1,\ldots,w_p,k,r_1,\ldots,r_q$ where k is a faulty statement and r_q is an output statement from which we can observe different outputs for fault-free and faulty systems. The task involves two steps. The first step is sensitization which finds the conditions for executing the sequence $k,r_1,\ldots,r_q$. If the conditions are satisfied, the system will sequentially execute $k,r_1,\ldots,r_q$ so that $OUT(r_q) \neq OUT(r'_q)$. These conditions constitute the constraint package C_k. In the next step, we shall obtain the conditions in C_k by applying the input sequence.

The second step is justification which finds the fault-free sequence $w_1,w_2,\ldots,w_p$ where w_1 is an input statement such that the conditions in C_k are satisfied after executing the sequence $w_1,w_2,\ldots,w_p$. Each statement in sequence $w_1,\ldots,w_p$ has to be fault-free, otherwise the fault in k may be masked. The procedure given below is for generating the sequence. If such a sequence does not exist, the fault will be undetectable. Of course before w_1 is executed some conditions for the constraint package C_{w_1} have to be satisfied. Generally speaking, we cannot find the $w_1,\ldots,w_p$ and C_{w_1} from $k,r_1,\ldots,r_q$ and C_k by one step. We have to use the inverse operation step by step, from k to w_p, from w_p to w_{p-1} and so on until w_1 is reached. In the first step of the inverse operation, we construct C_{w_p} to guarantee that the conditions in C_k will be satisfied after w_p is executed. So C_{w_p} includes not only the conditions needed for executing w_p, but also the conditions from which the transformation w_p will transform to the conditions in C_k. This inverse operation process continues until the input statement w_1 and C_{w_1} are found. During the inverse operation process, some special cases may occur. For example, for statement j, there may exist several statements, predecessors of j, which precede j, we can choose anyone. The set of all predecessors of j is J_{-1} and j_{-1} is a member of J_{-1}. The constraints for the chosen predecessor j_{-1} is $C_{j_{-1}}$. If there exists a contradiction in $C_{j_{-1}}$, take another statement in J_{-1} and try again. If all retries fail, we can choose another faulty statement as k and repeat the whole process since, in general, for any given functional fault, there are several faulty statements in the RTL description. If this cannot be done, then the functional fault k/k' is undetectable. The formal description of this idea is as follows:

Definition 6: Let (R_D) denote the content of R_D. If right after executing the faulty statement k, $(R_D)=s_d$ for the fault-free system and $(R_D)\neq s_d$ for the faulty system, then s_d is called sensitive data for the k/k' fault.

Definition 7: If f is a transformation from the set of conditions $C_{j_{-1}}$ to the set C_j, then f^{-1}, the inverse transformation of f, is the transformation

from C_j to $C_{j_{-1}}$ such that $f \cdot f^{-1} = I$ (identity). $f^{-1}(C_j)$ denotes the set of conditions through the inverse transformation f^{-1} of the conditions in C_j.

For example, suppose f=SHIFT RIGHT, C_j={Q=01, A=0101}. The f^{-1} = SHIFT LEFT, $C_{j_{-1}} = f^{-1}(C_j)$={Q=1x, A =101x}. Because shifting Q=1x to the right one bit produces Q=01, shifting right 101x produces 0101, that is $f \cdot f^{-1}(C_j) = C_j$. As another example, if f is addition, then f^{-1} is subtraction.

Definition 8: For statement j in sequence $w_1,\ldots,w_p,k$, the constraint package C_j is recursively defined as follows:

1. C_k is a set of conditions to guarantee that the sequence $k,r_1,\ldots,r_q$ is a sensitizing sequence.

2. For any statement j in the sequence $w_1,\ldots,w_p,k$, if j_{-1}: (T,C) $R_D \leftarrow f(R_{s1},\ldots,R_{sv}), \rightarrow n$ then we have

$$C \cup [(R_D)=f(R_{s1},\ldots,R_{sv})] \cup f^{-1}(C_j) \subseteq C_{j_{-1}},$$

$$j_{-1} \in \{J_{-1}\}$$

where $f^{-1}(C_j)$ denotes the inverse transformation of all constraints in C_j for statement j_{-1}.

Procedure for Test Generation for Detecting a Given Functional Fault

Step 1 - Search: In the RTL description of the system, find a statement k which cannot be executed correctly under the functional fault.

Step 2 - Propagation: Find a sensitizing sequence for the fault k/k' and construct the constraint package C_k based on Definition 8.

Step 3 - Justification: Find the fault-free sequence $w_1,w_2,\ldots,w_p$ and C_w, by sequentially applying the inverse operations.

From j=k to an input statement, execute the following statements:

(a) If $||J_{-1}||>1$ then temp←j, where $||J_{-1}||$ denotes the number of elements in J_{-1}.

(b) If $||J_{-1}||=0$, go to Step 1 to search another k.

(c) Take $j_{-1} \in \{J_{-1}\}$, $\{J_{-1}\} \leftarrow \{J_{-1}\} - j_{-1}$ (Remove j_{-1}, an element of J_{-1}, from J_{-1}).

(d) $C_{j_{-1}} \leftarrow C \cup [(R_D)=f(R_{s1},\ldots,R_{sv})] \cup f^{-1}(C_j)$

(e) Solve the equations in $C_{j_{-1}}$

(f) If there exist contradictions in $C_{j_{-1}}$, then j←temp else j←j_{-1}.

(g) If j=k go to step 3(b).

(h) If j is a faulty statement other than k then k←j, return to step 2.

Step 4: Solve the final constraint equations in C_{IN} to find out the input patterns for detecting the functional fault. If this cannot be done, let k ← IN, go to Step 3.

Example 1: Fig. 1 illustrates the RTL description of a module for multiplication. Following the above procedure, the test generation for detecting SC stuck-at-0 is given below.

```
EXTERNAL INPUT: XS, X   /*MULTIPLICAND*/
                YS,Y    /*MULTIPLIER*/
                START   /*MULTIPLICATION COMMAND*/
EXTERNAL OUTPUT:  Q(0.. 1), A(0.. 1), AS
INTERNAL REG'S:   M,BS,B,QS,E,SC
  /*E=overflow bit, SC=sequence counter*/
(0) M←START   /*strobe multiplication command*/
(1) (M=0), → 0
(2) B←X, BS←XS, Q←Y, QS←YS
(3) AS←QS ⊕ BS  /*calculate the sign bit*/
(4) A←00, E←0, SC←0
(5) (Q(1)=0),→7
(6) E∘A←A+B     /*partial sum of the product*/
(7) SHR E∘A∘Q   /*shift right 1 bit*/
(8) SC←SC-1     /*decrement sequence counter by 1*/
(9) (SC≠0), → 5
(10) M←0, → 1   /*end of multiplication*/
```

Fig. 1 - RTL Description of Multiplication System

Step 1: From Fig. 1 it is easy to see that if SC stuck-at-0, the statement 8 will not be executed correctly. So we choose k=8.

Step 2: After 8 is executed, SC=1 for the fault-free system, SC=0 for the faulty system, so 1 is the sensitive data and before 8 is executed, SC=0. SC=0 is then a condition belonging to C_8. Among several choices, we take a sensitizing sequence 8→9→5→7. Suppose (E∘A∘Q) = $x_1x_2x_3x_4x_5$ where "∘" denotes concatenation, 5→7 implies x_5=0. After 7 is executed, we have (E∘A∘Q) = 0 $x_1x_2x_3x_4$ for the fault-free machine. But, for the faulty machine, the following sequence is taken: 8→9→10→1→0, so (E∘A∘Q) will not be changed. Thus we obtain the constraint package C_8={SC=0, x_5=0, $x_1x_2x_3x_4x_5 \neq$ 0 $x_1x_2x_3x_4$, (E∘A∘Q) = $x_1x_2x_3x_4x_5$}.

Step 3. The only way to reach statement 8 is from 7, so we have $\{8_{-1}\}$ = 7. Let us construct C_7 from C_8. Under this inverse transformation, since executing a shift right operation always inserts a 0 into the left-hand bit, x_1=0 and (E∘A∘Q) = $x_2x_3x_4x_5x_6$, thus we obtain C_7 = {SC=0, x_5=0, x_1=0, $x_1x_2x_3x_4x_5 \neq 0\ x_1x_2x_3x_4$, (E∘A∘Q) = $x_2x_3x_4x_5x_6$}. Continuing the backward tracing process, we find that statement 7 can come from 5 or 6, so $\{7_{-1}\}$ = {5,6}. First, let us choose 7_{-1}=5. 5→7 is possible only when x_6=0. Considering 5_{-1}=4 we have ($x_2x_3x_4$)= 000 and C_4={SC=0,x_5=0,x_1=0,x_6=0,($x_2x_3x_4$)=000,

$x_1x_2x_3x_4x_5 \neq 0\ x_1x_2x_3x_4\}$. Obviously, there is a contradiction in C_4. Therefore, we have to take another choice 7_{-1}=6 and repeat the process. The constraints for the whole process are given below.

Step 1: If SC s-a-0, then statement 8 cannot be executed correctly.

Step 2: 8 SC=0 /*SC=1 sensitive data*/
9 $(E \circ A \circ Q) = x_1x_2x_3x_4x_5$
/*9→10→1→0 for faulty system*/
5 x_5=0
/*we have two choices 5→6 or 5→7, choose 7*/
7 $(E \circ A \circ Q) = 0\ x_1x_2x_3x_4 \neq x_1x_2x_3x_4x_5$
/*output statement*/

Step 3: 8_{-1}=7, x_1=0
$(E \circ A \circ Q) = x_2x_3x_4x_5x_6$
/*f^{-1} is SHL*/
7_{-1}=5, x_6=0
/*$\{7_{-1}\}$ = {5,6}, choose 5*/
5_{-1}=4, $(x_2x_3x_4)$ = 000
$0\ x_1x_2x_3x_4 \neq x_1x_2x_3x_4x_5$
/*impossible, choose 6*/

7_{-1}=6, A+B=$(x_2x_3x_4)$

6_{-1}=5, x_6=1
5_{-1}=4, E=0, A=0 Let B=b_0b_1

hence x_2=0, B=(x_3x_4) = (b_0b_1)

Step 4: Equation $0\ x_1x_2x_3x_4 \neq x_1x_2x_3x_4x_5$
$000\ b_0b_1 \neq 00\ b_0b_1\ 0$

$0\ b_0b_1 \neq b_0b_1\ 0$

$b_0 \neq 0$ or $b_0 \neq b_1$ or $b_1 \neq 0$
thus X = 01 or 10 or 11
and Y = x_5x_6 =01
The input patterns are
(01,01), (10,01), (11,01).

We obtain the following input patterns for detecting SC s-a-0.

X=01 or 10 or 11, Y=01

If in Step 2, 6 is chosen instead of 7, the following test patterns will be obtained:

(01,10), (10,10), (11,10), (00,10),
(00,11), (10,11), (11,11)

Therefore, we get all possible test patterns for detecting SC s-a-0 by using this procedure.

In the next example, we shall show how a functional fault can be detected by using instruction sequence.

Example 2: Let us find an instruction sequence to test the stuck-at-0 faults in the adder of INTEL 8080 microprocessor. We shall use the ADD M instruction to detect the stuck-at-0 faults in the adder and MOV M,A to move the sum to memory so that the faulty data can be observed in the bus between the accumulator A and memory M. The RTL statements for describing the micro-operations for ADD M and MOV M,A are given in Statements 1 to 7 and Statements 8 to 13 in Fig. 2, respectively.

```
1  (M1,T1)  OUT←PC /*micro-operations for ADD M
                   starts here*/
2  (M1,T2)  PC←PC +1
3  (M1,T3)  IR←IN  /*instruction fetch*/
4  (M1,T4)  ACT←A  /*temporary accumulator(ACT)*/
5  (M2,T1)  OUT←(H)(L) /*sending address*/
6  (M2,T3)  TMP←((H)(L)) /*reading from memory
                   specified by address (H)(L)
                   into temporary register*/
7  (M3,T2)  A←(TMP)+(ACT)  /*addition*/
8  (M1,T1)  OUT←PC /*micro-operations for MOV M,A
                   starts here */
9  (M1,T2)  PC←PC+1
10 (M1,T3)  IR←IN  /*instruction fetch*/
11 (M1,T4)  TMP←(A)
12 (M2,T1)  OUT←(H)(L)  /*sending address*/
13 (M2,T3)  OUT←(TMP)   /*sending content*/
```

Figure 2

Step 1: If some bits of the adder stuck-at-0, the addition statement will be executed incorrectly. As shown in Fig. 2, the addition statement appears during state T2 of the cycle M3 of instruction ADD M which is the 7th statement in the RTL description, so we take k=7.

Step 2: We have got the faulty statement k=7. Now we have to find a sensitizing sequence to let us observe the wrong data from the data bus. One way is to use instruction MOV M,A to sensitize the faults in Statement 7 after executing ADD M. So we obtain 7 to 13 in Fig. 2 as the sensitizing sequence. In order to test any bits of the adder stuck-at-0, let the result of addition for the good machine (sensitive data) be 11...1.

Step 3: Justification: we load the accumulator A with a number, then we add a number in memory to A such that the sum is 11...1. When we store the sum into memory through the bus, faults can be observed from the bus lines.

7_{-1} (TMP)+(ACT)=11...1
/*we must have IN→TMP, IN→ACT*/
6_{-1} ((H)(L))=(TMP)
∴ ((H)(L))+(ACT)=11...1 /*IN→HL→TMP*/
4_{-1} (ACT)=(A)
∴ ((H)(L))+(A)=11...1
/*we have to use an instruction to load A from external source, we choose LDA addr*/

Step 4: To satisfy the last equation, we choose the following instruction sequence:

```
LDA  addr
ADD  M
MOV  M,A
```

satisfying constraint ((H)(L))+(A)=11...1. Many values of ((H)(L)) and (A) can satisfy the constraint. For example, ((H)(L))=11...1 and (A)=00...0 will detect and locate any bits of the adder stuck-at-0.

Theorem 1: The proecedure mentioned above is capable of generating tests for testing a detectable functional fault in digital systems described by RTL.

Proof: In Step 1, the existence of k is obvious, otherwise k/k' will not be a functional fault in the digital system. But k need not be unique. The test sequence

$$w_1,\ldots,w_p,k,r_1,\ldots,r_q$$

obtained from the procedure under the input patterns satisfying the conditions in constraint package C_{w_1} will detect the fault k/k' by observing the output at r_q. If there is no test sequence for all faulty RTL statements, then these statements are redundant. In this case, for all paths through any faulty statement, the fault-free system and the faulty system are just the same.

In Step 2, there exists at least a sequence from k to an output statement unless k is an isolated (trap) state in the system or k/k' is undetectable. (Interruption is not considered in this paper).

So far we have obtained the conditions which guarantee the path from k to r_q is sensitized. In Step 3, beginning with k, after k_{-1} is found, we take new conditions $C \cup (R_D)=f(R_{s1},\ldots,R_{sv})$ and together with $f^{-1}(C_k)$, the inverse mapping of C_k under the transformation f to form C_{k-1}. If the constraint package $C_{k_{-1}}$ is consistent, the procedure continues until an input statement (w_1) is found to provide the input sequence satisfying the conditions in C_{IN}. Under the input sequence, the sequence $w_1,\ldots,w_p,k,r_1,\ldots,r_q$ will detect the fault k/k'. This process is always possible unless either C_{IN} is unsolvable which means the fault is undetectable or there exists an infinite loop which is one of the four cases shown in Fig. 3.

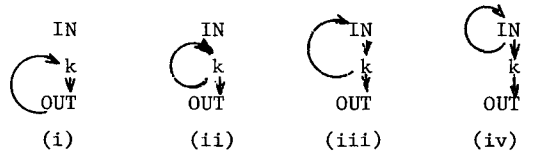

Figure 3

(i) The infinite loop includes k and OUT but excludes IN. In this case, changing the input pattern will not change the output. Hence the fault k/k' is undetectable.

(ii) The fault is again undetectable.

(iii) The infinite loop includes IN and k. If the input statement makes it possible to find the input patterns for detecting the fault, our goal is reached. Otherwise, the conditions in C_{w_1} are not completely satisfied, the fault is still undetectable.

(iv) The infinite loop includes IN but excludes k and OUT, the case is similar to (iii). QED

IV. TEST GENERATION PROCEDURES FOR MICROPROCESSORS

Microprocessors are widely used components in digital systems. In Section III we have developed a formal procedure to find input patterns for detecting a given function fault in digital systems. But it will not be very efficient to detect all function faults in a microprocessor exhaustively. Therefore, we need test generation procedures to test microprocessors. For instance, once we detect that the microprocessor cannot execute multiplication instruction correctly, we can use the procedure in Section III to detect every given function fault in RTL description for multiplication.

Two criteria can be used for judging the effectiveness of a test procedure. One is the fault coverage, another is the efficiency, including the time needed for test generation and testing. The more faults a test procedure covers and the higher efficiency it has, the better it will be. Thatte and Abraham, in their significant contribution [2], gave eight procedures for detecting four types of function faults. For the register decoding function fault, [2] combines READ and WRITE decoding faults into one. In this paper, to increase fault coverage, we test WRITE and READ decoding faults separately. On the other hand, for improving the efficiency we use one procedure to test several types of function faults to shorten the computing time. As a matter of fact, when we test register decoding function, correct execution of $1\rightarrow R$ means that the instruction decoding for this instruction, the data transfer and storage for data 1 and the register decoding for writing R are all correct. There are some other differences between [2] and this paper. We check three types of instruction decoding faults simultaneously instead of individually as shown in Procedure 2-4 of [2]. We present a cover matrix to guarantee the consideration of all possible data transfer paths from one register to another and between ALU and registers. Shrink the matrix each time after a chain is formed until all register transfers are covered.

In this section, we present three procedures to test five types of faults. Here we restrict ourselves to detect single permanent functional fault. For example, R_1/R_2 is a single register decoding functioned fault: which uses R_2 whenever R_1 appears in a RTL statement. We also consider single data transfer function faults, for example, fault $R_1\rightarrow R_2/\bar{R}_1\rightarrow R_2$, means $\bar{R}_1$ instead of R_1 is transferred into R_2.

4.1 Testing Register Decoding and Data Storage Faults

Definition 9: $R=\{R_1,\ldots,R_n\}$ is called the explicit register set if and only if $R_i (1\le i\le n)$ can be a source register or the destination register of at least one instruction.

Definition 10: WRITE $(R_i) = \{I^i_1,\ldots,I^i_{s_i}\}$ is the shortest instruction sequence for writing register R_i from IN. READ (R_i) can similarly be defined.

Here IN and OUT are the input and output data buses for the CPU. Notice that the sequence WRITE (R_i) and READ (R_i) may not be unique. The corresponding destination register sequence is denoted by

$\{WRITE\ (R_i)\}=\{R_D(I^i_1),\ldots,R_D(I^i_{s_i})\}$, where

$R_D(I^i_{s_i}) = R_i$

The writing distance is

$||WRITE\ (R_i)|| = s_i$

Similarly define

$READ(R_i) = \{I^i_1, I^i_2, \ldots, I^i_{t_i}\}$

$\{READ(R_i)\}=\{R_D(I^i_1),\ldots,R_D(I^i_{t_i})\}$where

$R_D(I^i_{t_i}) = OUT$

$||READ(R_i)|| = t_i$

The following lemmaes are obvious, hence their proofs are omitted.

Lemma 1: The registers in $\{WRITE\ (r_i)\}$ are different from each other. The same argument holds for $\{READ(R_i)\}$.

Lemma 2: If register R is in the ith position of a writing (reading) sequence, it will be in the ith position of any other writing (reading) sequence.

Lemma 3: $\{WRITE(R_i)\}\subset\{WRITE(R_j)\}$ only if $\{WRITE(R_i)\}$ is a subsequence of $\{WRITE(R_j)\}$.

In order to write (read) all registers without disturbing the others, we write into (read out from) registers in the order of decreasing (increasing) writing (reading) distance. The following sequences are obtained

Writing order = $R_1^w, R_2^w, \ldots, R_n^w$

Reading order = $R_1^r, R_2^r, \ldots, R_n^r$

The register sequence for writing R_i^w should not go thourgh R_j^w for all $j<i$. The register sequence for reading R_i^r should not go through R_j^r for all $j>i$.

Notice that for any register R, there exists at least a writing sequence $\{WRITE(R_i)\}$ such that $R\in\{WRITE(R_i\}$. From Lemma 1-3, we can partition all registers into various levels according to their position in writing sequence, as shown in Fig. 4. We may have register transfer statements in the RTL description performing transfer at the same level or from a higher level to a lower level or from J^{th} level to $(J+i)^{th}$ level. But it is impossible to transfer from J^{th} level to $(J+i)^{th}$ level ($i\geq 2$) (otherwise the $(J+i)^{th}$ level registers should belong to $(J+1)^{th}$ level.) However, it is possible to transfer from $(J+i)^{th}$ level to J^{th} level for $i\geq 1$. Furthermore, depending upon the RTL description, it is not necessary to have transfer statement from any register to any other register. From now on, we order $R_1,\ldots,R_n$ according to their level as $R^I_1,\ldots,R^I_{n1},R^{II}_1,\ldots,R^{II}_{n2},\ldots,R^N_1,\ldots,R^N_{nN}$.

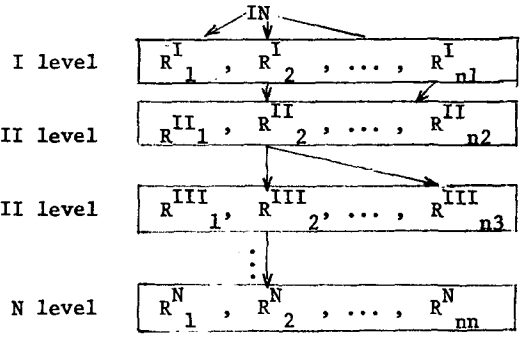

Figure 4

In order to test register decoding function fault, we need to write all registers with different data and read out respectively to check the contents of the registers. To test register storage fault, we need to write every register with a specified data to check if there exist any stuck-at faults and bridging faults [7,8] in some bits of the registers. The procedure is given below.

Procedure 1: Testing register decoding and data storage faults

Step 1: Write all registers according to the order $R^w_1,R^w_2,\ldots R^w_n$ with data 1,2,...,n respectively.

Read R^w_1

Next, write all registers with 1,2,...,n and read R^w_2.

Repeat above process for $R^w_3,\ldots,R^w_n$ until R^w_n is read out.

Step 2: Write all registers according to the order $R^w_1,R^w_2,\ldots,R^w_n$ with data 1,2,...,n respectively. (This is necessary because in the reading process in Step 1, the contents of some registers may be destroyed.)

Step 3: Write R^r_1 with data $\overline{1}$ (the complement of 0...01). Read all registers according to the order $R^r_1,R^r_2,\ldots,R^r_n$. Next write R^r_2 with data $\overline{2}$ and read all registers. Repeat above process for $R^r_3,\ldots,R^r_n$ until R^r_n is written with $\overline{n}$.

Step 4: Repeat Steps 1-3 with data $\overline{1},\overline{2},\ldots,\overline{n}$.

Step 5: Apply 1...10...0 to all bits of all registers. For n registers of m bits each, we have mn bits for each pattern.

Repeat this step for the following data:

$$D=\begin{Bmatrix} 11.....1\ 00.....0\ 11.....1\ 00.....0 \\ 11.100.0\ 11.100.0\ 11.100.0\ 11.100.0 \\ \quad\quad\quad\quad\ \ldots\quad\quad\quad\ \ldots \\ 1010.....1010.....1010 \end{Bmatrix}$$

Theorem 2: Procedure 1 detects all single register decoding, data storage function faults.

Proof: Let us consider the following three cases:

(i) WRITE register decoding fault R_D/R'_D.

Case 1: If R'_D is a register other than R_D and $R_D=R^W_i, R'_D=R^W_j$ $(i\neq j)$, we only consider the i<j case since the i>j case is similar. At Step 1 of Procedure 1, since i<j, writing R^W_i is before writing R^W_j. But due to the fault R^W_i/R^W_j, i will be written into R^W_j instead of R^W_i. When j is written into R^W_j, it erases its previous content. Therefore, "write all registers" results in

(R^W_i) = the original content,

(R^W_j) = j

Next let us consider two conditions when "read R_i" is executed.

Condition 1: If reading register decoding is correct, "read R^W_i" will obtain the original content of R^W_i.

(a) The original content is different from i, the fault is detected.

(b) The original content is i. The fault is detected when we apply $\bar{i}$ at Step 4 of Procedure 1.

Condition 2: If reading register decoding is wrong, when read R^W_i" is executed, the content of R^W_j which is j will be read. Hence, the fault is detected.

Case 2: If $R'_D=\phi$ (empty) then nothing is written into R_D and after "write R_D", R_D keeps its original content. This is the same as Condition 1 of Case 1.

Case 3: $R'_D=R_D\cap R_{D1}$. This means that in the process of writing R_D, another register R_{D1} is also simultaneously written with the same information. Suppose $R_D=R^W_i$, $R_{D1}=R^W_j$, there are two conditions to be considered.

Condition 1: i<j.

At Step 1 of Procedure 1, $R^W_j\leftarrow j$ is after $R^W_i\leftarrow i$ and thus the fault is masked. Therefore, if READ decoding is correct, the fault cannot be detected by Procedure 1. However, it can be detected by Procedure 2 when the data transfer is tested. If READ decoding is also wrong, "read R^W_i" will read both i and j simultaneously and hence produces the data other than i and hence the fault can be detected.

Condition 2: i>j

At Step 1 of Procedure 1, $R^W_j\leftarrow j$ before $R^W_i\leftarrow i$ but due to the fault, i is written into both R^W_i and R^W_j which destroys the original content of R^W_j. The fault is detected when "Read R^W_j" since (R^W_j) becomes i instead of j.

(ii) READ register decoding fault R_D/R'_D

The proof is similar to (i) hence it is omitted.

(iii) Data storage fault $(R_D)/(R_D)'$

Procedure 1, writing and reading R_D with data i and $\bar{i}$ will detect stuck-at faults in any bits, writing and reading R_D with data given by D in Step 5 will detect any bridging faults among register bits.

4.2 Testing Data Transfer and Instruction Decoding Faults

After Procedure 1, we have covered the data transfer paths in {WRITE(R_i)} and {READ(R_i)} $(1\le i\le n)$, but we have not covered all data transfer paths among registers and paths from ALU (arithmetic logic unit) to registers. In this subsection, we will give some procedures for detecting all data transfer and M-class (manipulation class) instruction decoding and manipulation function faults.

Definition 11: The (n+1) x (n+1) matrix

$A_i =$

	OUT	R_1	R_2	...	R_n
IN	0	a^i_{01}		...	a^i_{0n}
R_1	a^i_{10}	a^i_{11}		...	a^i_{1n}
R_2					
⋮					
R_n	a^i_{n0}	a^i_{n1}		...	a^i_{nn}

is called the i^{th} step transfer cover matrix where

$$a^i_{0j}=\begin{cases}1 & \text{IN}\mapsto R_j \text{ exists but has not been covered after the } i^{th} \text{ step test.}\\ d & \text{IN}\mapsto R_j \text{ exists but has been covered after the } i^{th} \text{ step test.}\\ 0 & \text{IN}\mapsto R_j \text{ does not exist.}\end{cases}$$

$$a^i_{j0}=\begin{cases}1 & R_j\to\text{OUT exists but has not been covered after the } i^{th} \text{ step test.}\\ d & R_j\to\text{OUT exists but has been covered after the } i^{th} \text{ step test.}\\ 0 & R_j\to\text{OUT does not exist.}\end{cases}$$

$$a^i_{kj}=\begin{cases}1 & R_k\to R_j \text{ exists but has not been covered after the } i^{th} \text{ step test.}\\ d & R_k\to R_j \text{ exists but has been covered after the } i^{th} \text{ step test.}\\ 0 & R_k\to R_j \text{ does not exist.}\end{cases}$$

Obviously, A_0 can be produced based on the instruction set. $^0a^0_{ij}=1$ if and only if there exists an

instruction performing $R_i \rightarrow R_j$. After Procedure 1, $a^1_{ij}=d$ if and only if $R_i \rightarrow R_j$ is included in some {WRITE(R_s)} or {READ(R_s)}.

Example:

$$A_0 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

The register transfer relations corresponding to matrix A_0 is shown in Figure 5.

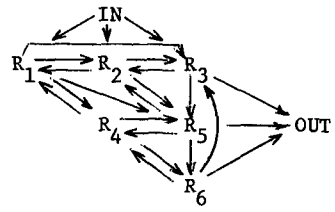

Figure 5

Suppose

{WRITE(R_1)}={R_1} , {READ(R_1)}={R_1,R_3},
{WRITE(R_2)}={R_2} , {READ(R_2)}={R_2,R_5},
{WRITE(R_3)}={R_3} , {READ(R_3)}={R_3},
{WRITE(R_4)}={R_1,R_4}, {READ(R_4)}={R_4,R_6}
{WRITE(R_5)}={R_3,R_5}, {READ(R_5)}={R_5},
{WRITE(R_6)}={R_1,R_5,R_6}and{READ(R_6)}={R_6},

then

$$A_1 = \begin{pmatrix} 0 & d & d & d & 0 & 0 & 0 \\ 0 & 0 & 1 & d & d & d & 0 \\ 0 & 1 & 0 & 1 & 0 & d & 0 \\ d & 0 & 1 & 0 & 0 & d & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & d \\ d & 0 & 1 & 0 & 1 & 0 & d \\ d & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Definition 12: A sequence of all registers

$$C^i = \{R^i_1, R^i_2, \ldots, R^i_n\}$$

is called a chain if and only if there exist the following data transfer paths:

$$IN \rightarrow R^i_1 \rightarrow R^i_2 \rightarrow \ldots \rightarrow R^i_n \rightarrow OUT$$

The set of chains $\{C^1, C^2, \ldots, C^m\}$ is called a complete chain set if and only if any register transfer path specified by one instruction is included in at least one chain in the set.

Example: As shown above, from A_1, we see $a^1_{12}=a^1_{21}=1$, $a^1_{23}=a^1_{32}=1$, $a^1_{41}=1$, $a^1_{52}=1$, $a^1_{45}=a^1_{54}=1$, $a^1_{63}=1$ and $a^1_{64}=1$.

The complete chain set of A_1 is

$C^1=\{R_3, R_2, R_1, R_4, R_5, R_6\}$
$C^2=\{R_2, R_5, R_6, R_4, R_1, R_3\}$
$C^3=\{R_2, R_1, R_5, R_4, R_6, R_3\}$
$C^4=\{R_3, R_5, R_2, R_1, R_4, R_6\}$
$C^5=\{R_1, R_2, R_3, R_5, R_4, R_6\}$

Note that the d's in matrix A are "don't cares". To form a complete chain we can use these d entries. After we get the complete chain set of A_1, we can test all data transfer paths between any two registers.

Our testing scheme is illustrated in Fig. 6 with the step-by-step procedure given below.

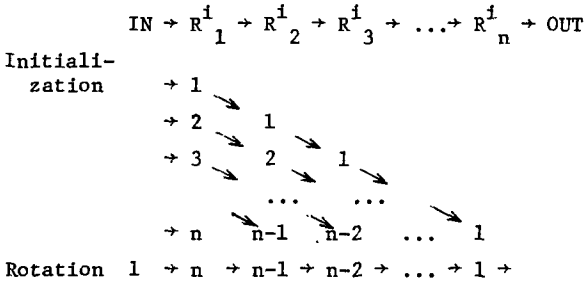

Figure 6

Procedure 2: Testing data transfer, M-class instruction decoding faults

For all chains in the complete chain set of A_1, for example $C^i=\{R^i_1, R^i_2, \ldots, R^i_n\}$, do the following steps:

Step 1: Initialization /*write C^i with data {n,n-1,...,3,2,1}*/

Write all registers in C^i with data {n,n-1,...,3,2,1} according to the order $R^w_1, R^w_2, \ldots, R^w_n$ by using the following:

1) write R^i_1 with data 1.

Read all registers according to the order $R^r_1, R^r_2, \ldots R^r_n$ to check to see if $(R^i_1)=1$ and the other registers keep their original contents.

2) $R^i_1 \leftarrow 1$, $R^i_2 \leftarrow R^i_1$, $R^i_1 \leftarrow 2$.

Read all registers according to the order $R^r_1, R^r_2, \ldots, R^r_n$.

3) $R^i_1 \leftarrow 1$, $R^i_2 \leftarrow R^i_1$, $R^i_1 \leftarrow 2$, $R^i_3 \leftarrow R^i_2$, $R^i_2 \leftarrow R^i_1$, $R^i_1 \leftarrow 3$ and read all registers.

Continue the process until $\{R^i_1, R^i_2 \ldots R^i_n\}=\{n,n-1,\ldots,1\}$. In each substep, the data expected in the chain are shift right to the next register and R^i_1 is written with a new data, then all registers are read according to the order $R^r_1, R^r_2, \ldots, R^r_n$. Note that in the above process,

we read all registers at each substep so that we can check to see if the contents of any other registers are changed when $R^i_j \rightarrow R^i_{j+1}$. The purpose is to remedy the deficiency of Procedure 1 mentioned in the proof of Theorem 2, Item (i), Case 3, Condition 1.

Step 2: Rotation for transfer: Read R^i_n, circulating (rotating) shift the data n,n-1,...,1 to the right one register. From the output we can observe the data sequence 1,2,3,...,n.

Step 3: Repeat Steps 1-2 using the following patterns:

$$\overline{n},\overline{n-1},\ldots,\overline{3},\overline{2},\overline{1}$$

Step 4: Apply 1...10...0 to all bits of all registers. For n registers of m bits each, we have mn bits for each pattern. Repeat above steps for the following data given by D in Step 5 of Procedure 1.

Step 5: Initialization: This step is the same as Step 1 without reading.

Step 6: Rotation for manipulation: For every manipulation in M-class instructions, take the first group of registers in C^i as the source registers, the next register as the destination register, execute the instruction.

Take the addition as an example, the following operations are performed.

$$R^i_1 + R^i_2 \rightarrow R^i_3$$
$$R^i_2 + R^i_3 \rightarrow R^i_4$$
$$R^i_3 + R^i_4 \rightarrow R^i_5$$
$$\ldots \quad \ldots \quad \ldots$$
$$R^i_{n-2} + R^i_{n-1} \rightarrow R^i_n$$

and then rotate to output.

Theorem 3: Procedure 2 is capable of detecting all data transfer and T-class (transfer), M-class (manipulation) instruction decoding function faults if the complete chain set exists.

4.3 Testing data manipulation faults

We now present Procedure 3 for M-class instructions which includes all kinds of data manipulation statements in RTL description to detect data manipulation function faults by using different operands.

Procedure 3: Detecting data manipulation function faults

For every M-class instruction, do the following:

Step 1: Write all registers with 1,2,...,n according to the order of $R^W_1,R^W_2,\ldots,R^W_n$.

Step 2: Take any specified registers as source registers, take another as destination register, select proper operands, write them in. Here "proper operands" means that they can easily cause the wrong result of the operation. The selection is difficult if the logical level description of the functional unit is not available. In this case, more input patterns are needed.

Step 3: Execute the instruction to be tested.

Step 4: Read all registers according to the order $R^r_1,R^r_2,\ldots,R^r_n$. By analyzing contents of R_i ($1 \le i \le n$) obtained in Step 4, we may detect the data manipulation function faults if it occurs.

V. CONCLUSION

Since the behavior instead of the detailed implementation of a VLSI chip is known to the users of the IC chips, RTL (Register Transfer Language) seems to be an effective tool for describing the behavior of a digital system. From the results in this paper, it is interesting to know that once the behavior of a digital system is described by RTL, similar ideas used in testing generation for stuck-at faults can be pushed up to RTL level with each RTL statement as a "component" of the system. For a given functional fault, this paper presents a procedure to generate all possible test patterns. For a new product of microprocessor, this paper presents three procedures to generate test for detecting five types of function faults. All procedures given in this paper can be programmed (Due to the page limitation, the proof for Theorem 3 has been omitted.)

REFERENCES

[1] S.Y.H. Su and Y.I. Hsieh; "Testing Functional Faults in Digital Systems Described by Register Transfer Language," Digest of papers, 1981 Test Conference, pp. 447-457. Revised version published in Journal of Digital Systems, June 1982.

[2] S.M. Thatte and J.A. Abraham, "Test Generation for Microprocessors," IEEE Transactions on Computers, pp. 429-441, June 1980.

[3] J.A. Abraham and S.M. Thatte, "Fault Coverage of Test Programs for a Microprocessor," Digest of Papers, 1979 Test Conference, pp. 18-22.

[4] S.Y.H. Su, "A Survey of Computer Hardware Description Languages in the U.S.A.," Computer, Dec. 1974, pp. 45-51.

[5] Intel 8080 Microprocessor Systems User's Manual, Intel Corporation, Santa Clara, California, September 1975.

[6] S.M. Thatte and J.A. Abraham, "A Methodology for Functional Level Testing of Microprocessors," Digest of Papers FTCS-8, 1978, pp. 90-95.

[7] M. Karpovsky and S.Y.H. Su, "Detecting Bridging and Stuck-at Faults at Input and Output Pins of Standard Digital Components," Proceedings of 17th Design Automation Conference, pp. 494-505, June 1980.

[8] M. Karpovsky and S.Y.H. Su, "Detection and Location of Input and Feedback Bridging Faults Among Input and Output Lines," IEEE Transactions on Computers, pp. 523-527, June 1980.