# Program-Based Delay Fault Self-Testing of Processor Cores

Virendra Singh* [1,3], Michiko Inoue[1], Kewal K Saluja[2], and Hideo Fujiwara[1]

[1]Nara Institute of Science & Technology, Ikoma, Nara 630-0192, Japan
{virend-s, kounoe, fujiwara}@is.aist-nara.ac.jp

[2]University of Wisconsin – Madison, U.S.A
saluja@ece.wisc.edu

[3]Central Electronics Engineering Research Institute, Pilani, India
viren@ceeri.ernet.in

***Abstract***

*This paper proposes an efficient methodology of delay fault testing of processor cores using its instruction set. These test vectors can be applied in the functional mode of operation, hence, self-testing of processor core becomes possible. A delay fault will affect the circuit functionality only when it can be activated in functional mode. There are some paths, which are never excited in the functional mode of operation; hence these are functionally untestable paths. The proposed approach uses a graph theoretic model (represented as an Instruction Execution Graph) of the datapath and a finite state machine model of the controller for the elimination of functionally untestable paths at the early stage without looking into the circuit details and extraction of constraints for the the paths that can potentially be tested. Path delay fault model is used. The experimental results on Parwan processor demonstrate the effectiveness of our method.*

## 1. Introduction

The importance of at-speed delay fault testing has soared in recent years to keep pace with the increasing speed and decreasing feature size of processors as aggressive design methodologies of processors use giga-hertz clock and deep sub-micron technology. At-speed testing using external tester is not an economical viable scheme. Structural self-testing methodology such as Built In Self Test (BIST) has been considered as an alternative. However, the following properties of BIST, 1) excessive area and performance overhead 2) operation in orthogonal test mode 3) requirement to make circuit BIST ready beforehand, 4) possibility of excessive power consumption in test mode, and 5) resolution of various clocking issues, make it unsuitable for the testing of processor cores especially when a optimized processor core is embedded deep inside a System-on-a-Chip (SoC). Once a core is embedded then it is also difficult to access the core for test application. Functional self-testing resolves these issues as it is operating in functional mode of operation by using its instructions. Functional self-test can easily be applied on a processor core, embedded deep inside an SoC. This paper focuses on functional self-testing of processor cores.

A number of software based self-test approaches [3,4,5,6,7,8,9], targeting stuck-at faults, have been proposed. The approaches proposed by Shen and Abraham [3] and Batcher and Papachristou [4] are based on instruction randomization. Both these approaches [3,4] give low fault coverage due to high level of abstraction and they generate large code sequence resulting in large test application time. Chen and Dey [5] used the concept of self-test signature in which they generate structural tests in the form of self-test signatures for functional modules by taking constraints into consideration. Due to pseudorandom nature of this methodology self-test code size and test application time are large. Paschalis et. al. [6] use self-test routines for functional modules based on deterministic test sets for testing datapath of a processor. Similarly, Krantis et. al. [7,8] proposed a methodology to tests every functional component of the processor for all the operations performed by that component using deterministic test sets. Deterministic nature of these [6,7,8] approaches lead to reduced test code size but these methods find difficult to achieve high fault coverage for complex architectures. Li Chen et al [9] proposed a scalable methodology based on test program templates which uses statistical regression method for function mapping. Approaches [5,6,7,8,9] do not explicitly consider the controller.

A software based self-test approach targeting delay faults was proposed by Lai et. al [10,11,12]. This approach, first classifies a path to be functionally testable or untestable. The authors argue that delay defects on the functionally untestable paths will not cause any chip failure. Path classification is performed by extracting a set of constraints for the datapath logic and the controller. In constraint extraction procedure for datapath, all instruction pairs are enumerated and for each instruction pair all possible vector pairs that can be applied to the datapath are derived symbolically. This requires a substantial effort to analyze all the instructions and all possible pairs of instructions even though it is not necessary to analyze all the pairs as shown in this paper. For controller, constraints in terms of legitimate bit patterns in registers and correlation between control signals and transition in registers are extracted. Path classification procedure in controller uses sequential path classification methodology i.e., in order to classify a path it propagate the transition forward till Primary Output (PO) and backward till Primary Input (PI) in multiple time frames under the constraints. Method proposed in this paper extract the constraints first on state transition, which eliminate the need of consideration of multiple time frames. After classification of paths, constrained Automatic Test Pattern Generator (ATPG) is used to generate the test patterns for testable paths. Results on controller are not reported in [10,11,12]. Lai and Cheng [13] proposed an approach for delay fault testing of an SoC using its own embedded processor instructions.

Our methodology uses graph theoretic model of datapath (represented by Instruction Execution Graph) and finite state machine model of the controller to eliminate the functionally untestable paths at the early stage without considering circuit details. This eliminates a substantial number of functionally untestable faults. It is also used to extract the constraints for the paths classified as potentially

* This work was carried out while author was with NAIST, Japan.

testable paths. These extracted constraints are used for test generation. As the vectors are generated under constraints, instruction(s) to apply the test vectors can always be found. However, the method proposed in [10,11,12] considers circuit details for path classification and multiple time frames for the controller.

The paper is organized as follows. Section 2 describes the overview of our work and definitions used. Sections 3 and 4 describe testing methodology of datapath and controller respectively. Section 5 discusses test instruction sequence generation and the experimental results, and finally the paper concludes with section 6.

## 2. Overview of proposed work

Our methodology considers datapath and controller separately as both of these have different design characteristics. The activities in the datapath are controlled by the controller, thus the function of the datapath and inputs to the datapah are constrained by the controller. Hence, only a subset of structurally applicable test vectors may be applied in functional mode of operation. Similarly, the controller is also constrained by state transitions and signals from the datapath, thus restricting the tests that can be applied to the controller during functional mode of a processor.

We model datapath by an Instruction Execution Graph (IE-Graph) that can be constructed from the instruction set architecture and RT level description of the processor. In our formulation of the test problem IE-Graph is used to classify the paths as *functionally untestable* (FUTP) or *potentially functionally testable paths* (PFTP), and to extract the constraints imposed on the datapath for PFTP paths. First, constraints on the control signals that can be applied on the paths between a pair of registers in consecutive cycles are extracted. Next, constraint on justifiable data inputs (registers) are extracted. Following these, a combinational constrained ATPG is used to generate test vectors under the extracted constraints. Thus, in this approach only those vectors are generated that can be applied functionally. Further, the search space is significantly small as only those states are used during test generation which can cause data transfer to take place on a path between a pair of registers.

For testing the controller, the constraints are extracted in the form of state transitions from its RT level description. These constraints also include the values of status signals in the status register and instruction code in the instruction register of the processor. After extracting the constraints, test generation is performed in two-phases. The first phase is the preprocessing stage during which all paths are classified as *FUTP or PFTP*. The second phase is the combinational constrained ATPG phase during which tests are generated for the paths classified as PFTP during the first phase. As the vectors must be generated with constraints on the states and inputs to the controller (contents of the instruction register and status register), the number of time frames that are required for sequential test generation are reduced. In the final phase test instructions are generated using the knowledge of the control signals and contents of the instruction register. Justification and observation instruction sequence generation processes are based on heuristics which minimize the number of instructions and/or the test application time.

Throughout this paper the following concepts and notation will be used.

Definition 1: A *path* [16] is defined as an ordered set of gates $\{g_0, g_1, \ldots, g_n\}$, where $g_0$ is a primary input or output from a FF, and $g_n$ is a primary output or input to a FF. Output of a gate $g_i$ is an input to gate $g_{i+1}$ $(0<i<n-1)$.

Definition 2: A path is (enhanced-scan or standard-scan) *structurally testable* [10] if there exists a structural test for the path, which can be applied through the (enhanced or standard) full-scan chain.

Definition 3: A path is *functionally testable* [10] if there exists a functional test for that path, otherwise the path is functionally untestable.

A functional two-pattern test does not exist to test a path implies that there does not exist an instruction or an instruction sequence to apply the required test in functional mode of operation. Clearly, functionally untestable paths are never activated in normal (functional) operational mode and we need not target these paths in our approach.

We use the following notation to represent signal values.

c: represents a value that does not change in two consecutive timeframes, i.e, it represents a stable 0 or a stable 1 value in two time frames.
x: represents a bit that can be assigned either a logic 0 or a logic 1 value at will.
d: represents bit which is not cared by state transition. It is the same as x, except that legitimate bit pattern in the register has to be justified.
R: represents rising transition.
F: represents falling transition.
A constraint can be represented by a vector pair, say P, and the elements of P can be 0, 1, x, c, or d.

Definition 4: A Constraint P is said to *cover* a constraint Q if P = Q or Q can be obtained from P by assigning 0 and 1 values to x's in P.

## 3. Datapath

In this section, we consider paths relevant to data transfer between registers in the datapath. The other paths are treated in the next section. The paths, which are going through the logic in the controller, are considered in the next section, even if they start from and end at some registers in the datapath.

Datapath is modeled by an IE-Graph. This is based on the concept of S-Graph proposed in [1, 2]. However, unlike S-Graph, the IE-Graph contains information about data transfer activities associated with an instruction as well as the state during which a given action takes place. IE-Graph is constructed from the instruction set architecture and register transfer level description and includes architecture registers of the datapath.

Nodes of the IE-Graph are (i) registers, (ii) two special nodes, IN and OUT, which model external world such as memory and I/O devices, (iii) part of registers which can be independently readable and writable, and (iv) equivalent registers (set of registers which behave in the same way with instruction set, as defined by [2]), such as registers in a register file. A directed edge between two nodes is drawn iff there exists at least one instruction which is responsible to transfer data (with or without manipulation) over the paths between two nodes (registers). Each edge is marked with a set of [state, instruction(s)] pairs, which are responsible for the data transfer between the pair of nodes.

Partial IE-Graph of Parwan Processor [15] is shown in Figure 1. Complete graph is given in [17]. Parwan Processor is an accumulator based 8-bit processor with 12-bit address bus. It has 17 instructions, listed in Table 1, and it supports both direct and indirect addressing modes.

Table 1. Instruction set of Parwan processor

| 1. LDA | 4. SUB | 7. JSR | 10.BRA_Z | 13.CLA | 16.ASL |
|---|---|---|---|---|---|
| 2. AND | 5. JMP | 8. BRA_V | 11.BRA_N | 14.CMA | 17.ASR |
| 3. ADD | 6. STA | 9. BRA_C | 12. NOP | 15.CMC | |

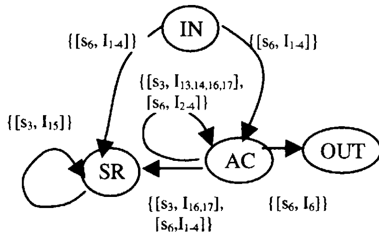


Fig 1. Partial IE-Graph of Parwan processor

Test vector generation process uses instruction set architecture, RT level description, and gate level netlist. It is a two-step process. The first step is constraint extraction process and the second step is test vector generation process.

### 3.1 Extraction of constraints

There are two types of constraints imposed on the datapath by the controller. (i) Control constraints, and (ii) Data constraints. Control constraints are the constraints on control signals, which are responsible to transfer data between two registers. These constraints are obtained from IE-Graph and RT level description. Data constraints are the constraints on the justifiable data in the registers under control constraints, which can be obtained using RT level description.

Definition 5: Let there be an edge from node $R_i$ to $R_o$, marked with $[s_l, I_p]$. The marked state $s_l$ is defined as a *latching state* for the paths represented by that edge.
Data transfer activity from register $R_i$ to $R_o$ takes place in state $s_l$ during the execution of instruction $I_p$ and register $R_o$ will be latched. Hence, state $s_l$ is defined as a latching state.

*Lemma* 1: Let $<V_1, V_2>$ be a test vector pair for a path from register $R_i$ to a register $R_o$, where test vector $V_1$ is followed by $V_2$ and the edge between these registers is marked with a set of state-instruction pairs $\{[s_l, I_p]\}$. This vector pair can be a test vector pair in functional mode only if there exists at least one state-instruction pair $[s_l, I_p] \in \{[s_l, I_p]\}$, such that

(i) vector $V_2$ can be applied in the latching state $s_l$ of the instruction $I_p$, and
(ii) vector $V_1$ can be applied in the state just before the latching state $s_l$ of the instruction $I_p$.

Note that every instruction is a sequence of state transitions and the latching state(s) in this sequence for a register pair is well defined. However, if the latching state happens to be the very first state of an instruction then the last state of every instruction needs to be considered as the state that immediately precedes the latching state.

During the latching state $s_l$, data transfer (with or without manipulation) from register $R_i$ to $R_o$ takes place and the result is latched in register $R_o$. Therefore, we can apply the second vector only in the latching state (say $s_l$) and the first vector must be applied in a state just before the latching state (say $s_j$). Two consecutive states $s_j$ and $s_l$ give the control constraints, and control signals in these states during the execution of instruction(s) marked with the latching state are obtained from RT level description. Constraints on the states during which we can apply the test vectors $<V_1, V_2>$ take care of justification of the control signals in the functional mode of testing. Data constraints in the form of justifiable data in the input register of the register pair and other registers required for the execution of marked instruction are obtained from RT level description.

*Lemma* 2: Paths from register $R_i$ to $R_o$ are *functionally untestable* if the following conditions exist,

1. $R_i$ is not an IN node, and
2. $R_i$ has no incoming edge marked with the state just before the latching state ($s_l$) of the instruction $I_p$ for any $[s_l, I_p]$ marked on the edge ($R_i$, $R_o$).

If conditions stated in Lemma 2 exist then transition cannot be launched from register $R_i$. Hence, the paths between a register pair $R_i$ and $R_o$ are FUTP. Otherwise, these paths are classified as PFTP. We need to extract the data constraints for the potentially functionally testable paths. Covering relation, defined in section 2, helps reduce the number of constraints.

Example 1. Constraints on paths between AC and AC in Parwan processor
The edge between nodes AC and AC is marked with $\{[s_3, (I_{13, 14}, I_{16-17})], [s_6, I_{2-4}]\}$, as shown in Figure 1. AC is neither an IN node nor it has any incoming edge which is marked with just previous state of its latching state $s_3$ or $s_6$. Therefore, using Lemma 2 we can conclude that paths from AC to AC are functionally untestable.

Example 2: Paths from IN to AC
The edge between nodes IN and AC is marked with $[s_6, I_{1-4}]$, as shown in Figure 1. These paths are PFTP in accordance with Lemma 2, as input node is an IN node, and the latching state for these paths is $s_6$. Therefore, control constraints are the control signals generated in state $s_4$ or $s_5$ followed by $s_6$ for the instructions $I_1$, $I_2$, $I_3$ or $I_4$. This is obtained from IE-Graph and RT level description. Data constraints can be obtained in the state $s_4$ followed by state $s_6$ or state $s_5$ followed by $s_6$, for the instructions $I_1$, $I_2$, $I_3$ and $I_4$. Data constraints for the instruction $I_3$ are shown in Table 2. Here we assume that when input to a combinational logic is in high impedance state then it can hold the logic value that is applied before the high impedance state. Parwan processor uses tristate buses which are responsible for the constraints on IN node.

Table 2. Data constraints for the paths between IN and AC

| State | $I_3$ (ADD) | | | |
|---|---|---|---|---|
| | ALU ctrl | SHU ctrl | IN | AC (other i/p) |
| $s_4$ | 000 | 00 | xxxx_xxxx | xxxx_xxxx |
| $s_6$ | 101 | 00 | xxxx_xxxx | cccc_cccc |

For instruction $I_3$, both control constraints, $s_4$ followed by $s_6$ and $s_5$ followed by $s_6$ are identical. Hence, using the covering relation one of these two constraints can be eliminated. All other constraints are extracted similarly.

### 3.2 Test vector generation procedure

Constrained ATPG is used to generate the test vectors for the PFTP paths under the extracted constraints. Path lists between a register pair and their corresponding

constraints are provided as inputs to an ATPG along with gate level netlist and it returns the test vectors for the testable path.

Procedures to extract the constraints and test generation is given in Figure 2. This procedure systematically extracts the constraints using IE-Graph and uses constrained ATPG to generate the test vectors.

```
Constraint Extraction Procedure
1. Constraint path pair set W = Φ
2. for nodes R_i (i = 1 to n) {  // there are  n nodes in IE-Graph //
3.      for each edge (R_i, R_j) (j = 1 to m) {
                  // there are m edges from node R_i //
4.          if paths are PFTP then {   // (using Lemma 2) //
5.           P_ij = Set of all paths between R_i and R_j
6.          C_ij = Set of constraints for the paths from
      node R_i to node R_j
7.          W = W ∪ {[ C_ij, P_ij ]}
8.           }
9.      }
10.}
Test Generation Procedure
   Constrained ATPG  process
   Input : Constraint path pair set W, Gate level net list
   Output : Set of testable path  with their test vector pairs
```

Fig 2. Constraint extraction and Test generation Procedures

## 4. Controller

Controller is a sequential circuit that is normally implemented as Mealy type or Moore type finite state machine. In this section, we treat all the paths that go through the logic in the controller. Test vectors applicable in functional mode of operation to the controller are restricted by the state transitions. If we never find a sequence of valid state transitions which could create a transition and propagate it along a path then that path is a functionally untestable path, even though that may be structurally testable. Therefore, we extract constraints on state transitions prior to test generation.

### 4.1 Extraction of constraints

Change of state of controller is determined by the values in registers (IR and SR), inputs and present state. Input from registers IR and SR (i.e., registers other than the present state register (PSR)) are treated as Constrained Primary Input (CPI). Therefore, we need to extract two types of constraints (i) constraints on state transition, and (ii) constraint on legitimate values in IR and SR registers, as these are treated as constrained primary input.

(i) Constraints on state transitions

Constraints on state transition can be extracted by extracting possible valid state transition under legitimate values in IR, SR and input, by using instruction set architecture and RT level description. We show it using Parwan processor as an example. Table 3 shows a part of the state transition table of Parwan processor.

The Table 3 shows that when present state is $s_1$ then next state will be either $s_1$ or $s_2$ depending on the value of input, and independent of values in IR and SR registers. During these state transitions ($s_1$ to $s_1$ or $s_2$) register IR and SR can have any legitimate value in the present state ($s_1$) and must have the same values in next state ($s_1$ or $s_2$). Hence, we cannot launch transition from IR and SR during these state transitions. When present state is $s_2$ then next state is always $s_3$. IR can have any legitimate value in present state ($s_2$) as well as in next state ($s_3$). Therefore, transition can be launched from IR during the state transition $s_2$ to $s_3$.

Table 3. State transition table of Parwan processor (partial)

| PS | NS | IR (PS) | IR (NS) | SR (PS) | SR (NS) | In (PS) {Intrpt.} |
|---|---|---|---|---|---|---|
| $s_1$ | $s_1$ | dddd_dddd | cccc_cccc | dddd | cccc | 1 |
| | $s_2$ | dddd_dddd | cccc_cccc | dddd | cccc | 0 |
| $s_2$ | $s_3$ | dddd_dddd | 0xxx_xxxx | dddd | cccc | d |
| | | | 100x_xxxx | dddd | cccc | d |
| | | | 101x_xxxx | dddd | cccc | d |
| | | | 110x_xxxx | dddd | cccc | d |
| | | | 1110_0000 | dddd | cccc | d |
| | | | 1110_0001 | dddd | cccc | d |
| | | | 1110_0010 | dddd | cccc | d |
| | | | 1110_0100 | dddd | cccc | d |
| | | | 1110_1000 | dddd | cccc | d |
| | | | 1110_1001 | dddd | cccc | d |

(ii) Constraints on legitimate values in IR and SR register (registers other than the present state register):

A set of legitimate values in the registers other than the present state register can be obtained from its instruction set architecture and RT level description. For example, IR of Parwan processor can have some of the legitimate bit patterns which are specified as {IR, <0xxx_xxxx, 10xx_xxxx, 110x_xxxx, 1111_0100, 1111_0010, 1111_0001, 1110_0000, 1110_0001, 1110_0010, 1110_0100, 1110_1000, 1110_1001>}.

### 4.2 Test generation process

Test generation process is a two-phase process which uses extracted constraints. The first phase is the preprocessing phase, which classifies paths as PFTP or FUTP. Functionally untestable paths are removed from the path list. The second phase generates the test vectors for PFTP paths if these are functionally testable under the extracted constraints.

*Phase* 1: Preprocessing

Preprocessing classifies a path as PFTP or FUTP by using state transition diagram and gate level implementation.

There are three types of paths in a controller

1. PSR to PSR
2. Primary input or constraint primary input (registers IR and SR) to present state register (PSR)
3. PI, CPI, or PSR to a register in datapath.

Paths from PSR to PSR are only responsible for sequential behavior of the controller circuit. For preprocessing, we construct a table that shows transition on bits in PSR and other registers with state transitions. Table 4 shows transition on bits in PSR with state transitions for Parwan processor when states are binary encoded.

Table 4. Transition on bits in PSR with state transition (Parwan Processor)

| bit | | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ | $s_8$ | $s_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $b_3$ | R | | | | $s_9$ | | | | | |
| | F | | | | | | | | | $s_1$ |
| $b_2$ | R | | | | $s_5, s_6, s_7$ | | | | | |
| | F | | | | | | $s_1, s_2$ | | $s_1$ | |
| $b_1$ | R | | $s_3$ | | | | | | | |
| | F | | | | $s_5, s_6, s_9$ | | | | | |
| $b_0$ | R | $s_2$ | | $s_4$ | | $s_6$ | | $s_8$ | | |
| | F | | $s_3$ | | $s_5, s_7, s_9$ | | $s_1$ | | $s_1$ | |

This table shows that there can be rising transition on bit $b_3$ only when there is a state transition from state $s_4$ to $s_9$. There can be falling transition on $b_3$ only when there is a state transition from state $s_9$ to $s_1$.

*Lemma* 3: Paths between bit i in register $R_1$ and bit j in register $R_2$ (registers $R_1$ and $R_2$ need not be different) in controller circuit are *functionally untestable* paths for a transition (rising or falling) if

(i) there does not exist a valid state transition $s_m$ to $s_n$ to launch the transition at bit i, or

(ii) there does not exist a state transition $s_p$ to $s_q$ which can receive the launched transition or its inverse (receive falling transition when rising is launched) at bit j, such that $s_n = s_p$.

A path is functionally testable if we can create a transition and propagate its effect along the path. If the conditions stated in Lemma 3 exist then we either cannot launch a transition or cannot propagate the created transition. Hence, the paths between bit i in register $R_1$ and bit j in register $R_2$ are FUTPs. Otherwise, these paths are classified as PFTP as transition can be created and may be propagated if values in other registers are justifiable. We also get precise constraints under which these paths can be tested using state transition table.

1. **PSR to PSR path classification** (Paths from controller to controller):
A path between bit i and bit j in PSR can be classified as follows for rising transition, using Lemma 3. We consider 3 consecutive time frames as shown in following table. Activities at bit i and j in PSR and required state transitions are listed in Table 5. These paths are PFTP iff either $s_m$ to $s_n$ to $s_r$, or $s_m$ to $s_p$ to $s_q$ state transition sequence exists.

Table 5. Activities at bit i and j in PSR

| Time frame | k | k+1 | k+2 |
|---|---|---|---|
| bit i | 0 | 1 | x |
| bit j | x | 0 (1) | 1 (0) |
| state | $s_m$ | $s_n(s_p)$ | $s_r(s_q)$ |

*Example* 3: PSR to PSR paths classification in Parwan processor when states are binary encoded. Table 3 shows transition on bits in PSR with state transitions. Paths from $b_2$ to $b_1$ (for rising transition) are classified as FUTP because no one state transition sequence exists to test these paths, where as paths from $b_2$ to $b_1$ (for falling transition) are classified as a PFTP because a state transition sequence $s_6$ to $s_2$ to $s_3$ exists. State transition sequence $s_6$ to $s_2$ to $s_3$ is an exact constraint for these paths ($b_2$ to $b_1$, falling transition) under which these can be a tested if other values are justifiable. Similarly, we can find out all PFTPs, which are the potential candidates for the next phase.

2. **Paths from PI or CPI to bit i in PSR classification**(Paths from input or datapath to controller):

(i) Paths from PI to bit i in PSR are classified as PFTP, iff there exists a state sequence ($s_m$ to $s_n$), which can receive a transition at bit i.

(ii) Paths from CPI to bit i in PSR are classified as PFTP, iff there exist a valid state transition ($s_m$ to $s_n$) to create a transition at CPI (register IR or SR) and there exists a valid state transition ($s_n$ to $s_p$) at bit i of PSR (according to Lemma 2)

3. **Paths from CPI or PSR to a register in datapath** (Paths from controller to datapath) are classified as PFTP, iff

(i) there exists a state sequence ($s_m$ to $s_n$) which can launch a transition at bit i in PSR or CPI, and

(ii) the register in the datapath, where these paths terminate, has an incoming edge, marked with state $s_n$ in IE-Graph and state $s_m$ and $s_n$ are two consecutive states of the marked instruction $I_p$.

*Phase* 2: Test generation
A constrained combinational ATPG is used to generate the test vectors for the paths, which are, classified as potentially functionally testable paths under the extracted constraints. ATPG is given with a set of PFTP along with their respective constraints. ATPG will return the test vectors if a path is testable under constraints.

This approach extracts the constraints in the form of state transitions and classifies the paths as functionally untestable or potentially functionally testable. Functionally untestable paths are removed from the path list. It uses combinational constraint ATPG to generate test vectors. Therefore, we need not to consider multiple time frames for all the paths like a sequential ATPG, as sequential behavior is taken care by the state transition in our approach, which in turn reduces the complexity of test generation.

## 5. Test instruction sequence generation and experimental results

The generated test vector pairs as explained in the preceding section are assigned to control signals and registers. Control signals and value(s) in IR in two consecutive time frames give the test instruction(s). Data in registers and in memory, which will be used by the test instruction, must be justified, using justification instruction. The result from the output register must be transferred to memory using observation instructions. For example, vector pairs are (V1 = {ALU ctrl=000, SHU ctrl=00, AC=48H, IN=24H}, V2 = {ALU ctrl=111, SHU ctrl=00, AC=48H, IN=04}). This shows that the test instruction will be SUB instruction and the value at x24H must be 04H. Value in AC must be 48H which can be justified by using LDA mem[1], and the result from AC is transferred using STA mem[2]. So, we need three instructions LDA mem[1], SUB mem[3], and STA mem[2]. Heuristics can be used for instruction justification process in order to reduce the test program size or test application time.

We have applied our methodology to Parwan processor [15]. The synthesized version of the Parwan processor contains 888 gates and 53 flip-flops. An IE-Graph is constructed and functionally untestable paths have been detected. Paths from AC to AC, AC to SR, AC to Out, SR to SR, SR to Out and PC to PC are found to be untestable. We extract constraints for rest of the paths using IE-Graph and RT level description. Similarly, a state transition table has been constructed which shows the constraints on controller. We generated test patterns for Non Robust (NR) [16] and Functional Sensitizable (FS)[16] paths under the extracted constraints manually, as no one commercially available ATPG handles our constraints. Here we consider, the paths which are starting from some register in datapath (e.g., IR or SR) going through the controller and terminating at some register in datapath, as a part of the controller. Results are shown in the table 6. Test instructions are generated manually.

The results show that 39% of functionally untestable paths in data path and 32% functionally untestable paths in controller are eliminated during the first phase without using circuit details.

Table 6: Results of Parwan processor

| | Datapath | | Controller | |
|---|---|---|---|---|
| | NR | FS | NR | FS |
| Total Path | 5,012 | 5,012 | 48,484 | 48,484 |
| # Faults | 10,024 | 10,024 | 96,968 | 96,968 |
| # Faults declared untestable in first phase* | 3,902 | 3,902 | 31,836 | 31,836 |
| # Functionally testable paths | 1,218 | 1,218 | 2,344 | 3,110 |
| Percentage of functionally testable paths | 12.1 | 12.1 | 2.4 | 3.2 |

* These paths are declared functionally untestable using RT level description and instruction set architecture only.

Table 7: Comparison with earlier work

| | | Percentage of functionally testable paths | |
|---|---|---|---|
| | | Our work | Lai et al [11]'s work |
| Datapath | NR | 12.1 | 3.7 |
| | FS | 12.1 | -- |
| Controller | NR | 2.4 | -- |
| | FS | 3.2 | --- |

The results also show that our methodology generates the test patterns for more number of faults in datapath as compared to Lai & Cheng [11] because we are considering at microinstruction level during the extraction of constraints for the potentially testable paths. Note that [11] is using different synthesized version of Parwan processor with 168 sequential elements in order to separate out controller and datapath to make it better testable and reduction of number of paths, where as we are using original Parwan processor. The results for the controller are not shown in Lai & Cheng [11]. This approach can be extended for pipelined architecture by considering pipeline registers in IE-Graph.

## 6. Conclusion

A systematic approach for the delay fault testing of processor core using its instruction set has been presented in this paper. A graph theoretic model for data path has been developed. This model is used with the RT level description to eliminate the functionally untestable paths at the early stage and extraction of constraints. Controller is modeled as a finite state machine and constraints on state transitions are extracted. This will eliminate the need of multiple time frame consideration for the test generation, as we extracted the constraints on state transitions, hence reduces the test generation complexity. Our experimental results show that our test generation process can efficiently generate the test vector for functionally testable paths which can be applied by test instructions. Our future work includes automation of the proposed method, extension of this approach for complex architectures, and application of some efficient heuristics for the generation of justification instruction, which can minimize the test size or test application time.

## Acknowledgement

This work was supported in part by Semiconductor Technology Academic Research Center (STARC) under the Research Project and in part by Japan Society for the Promotion of Science (JSPS) under Grants-in-Aid for Scientific Research B (2) (No. 15300018).

## References

[1]. S.M. Thatte and J.A. Abraham, "Test generation for Microprocessors", IEEE Trans. on Computers, Vol. C-29, No.6, June 1980, pp. 429-441.

[2]. D. Brahme and J.A. Abraham, "Functional Testing of Microprocessors", IEEE Trans. on Computers, vol. 33, No. 6, June 1984, pp. 475-484.

[3]. J. Shen and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation", in Proc. of the International Test Conference 1998, pp. 990-999.

[4]. K. Batcher and C. Papachristou, "Instruction Randomization Self Test for Processor Cores" in Proc. of the VLSI Test Symposium 1999, pp. 34-40.

[5]. Li Chen, and Sujit Dey, "Software-Based Self-Testing Methodology for Processor Cores", IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 20, No.3, March 2001, pp. 369-380.

[6]. A. Paschalis, D. Gizopoulos, N. Krantis, M. Psarakis, and Y. Zorian, "Deterministic Software-Based Self-Testing of Embedded Processor Cores", Design Automation & Test in Europe 2001, Munich, Germany, March 2001, pp 92-96.

[7]. N. Krantis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Instruction-Based Self-Testing of Processor Cores", in Proc. of the VLSI Test Symposium 2002, pp 223-228.

[8]. N. Krantis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-Based Self-Testing of Processor Cores", Journal of Electronic Testing: Theory and Application (JETTA) 19, 2003, pp 103-112.

[9]. Li Chen, S. Ravi, A. Raghunath, and S. Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", Proc. Design Automation Conference (DAC 03), ACM Press, pp. 548-553.

[10].W.-C. Lai, A. Krstic, and K.-T. Cheng, "On Testing the Path Delay Faults of a Microprocessor Using its Instruction Set", Proc. of the VLSI Test Symposium 2000, pp. 15-20.

[11].W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test Program Synthesis for Path Delay Faults in Microprocessor Cores", in Proc. of International Test Conference 2000, pp 1080-1089.

[12].W.-C. Lai, A.Krstic, and K.-T. Cheng, "Functionally Testable Path Delay Faults on a Microprocessor", IEEE Design & Test of Computers, Oct-Dec 2000, pp 6-14.

[13] W.-C. Lai, and K.-T. Cheng, "Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip", Proc. of the Design Automation Conference (DAC 01), ACM Press, NY, 2001, pp. 59-64.

[14].A. Krstic, Li Chen, W.-C. Lai, K.-T. Cheng, and Sujit Dey, " Embedded Software-Based Self-Test for Programmable Core-Based Designs", IEEE Design & Test of Computers, July-August 2002, pp. 18-27.

[15].Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, New York, 1997.

[16].A. Krstic and K.-T. Cheng, Delay fault testing for VLSI circuits, Kluwer Academic Publishers, 1998.

[17].V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Software-Based Delay Fault Testing of Processor Cores", NAIST technical report NAIST-IS-TR2003006, May 2003. http://isw3.aist-nara.ac.jp/Contents/Research-en/Research-en.html