Software-Based Delay Fault Self-Testing of Pipelined 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 a graph theoretic model based systematic approach for the delay fault testing of pipelined processor cores using their instruction sets. The proposed methodology generates the test vectors under the extracted architectural constraints. These test vectors can be applied in functional mode of operation, hence self-test become possible. Self-test in functional mode can also be used for the on-line periodic testing. Our approach uses a graph model to model the behaviour of the pipelined processor which is then used to extract the architectural constraints and to classify the paths in different categories. Test vectors are generated using constrained ATPG under the extracted constraints. Finally, an instruction sequence is generated for the application of generated test vectors by using graph model. The experimental results on a 16 bit five stage pipelined processor (VPRO) demonstrate the effectiveness of our methodology.

1. Introduction:

Today's System-on-a-Chips (SoCs) are mainly built around very high speed processors, in order to meet the increasing consumer demand of high performance and rich functionality with quick turn-around time. Normally, these processor cores are pre-optimized and implemented with deep sub-micron technology. Design reuse reduces time to market and design effort but introduces test difficulties. Moreover, it is no longer sufficient to target conventional stuck at faults, faults such as delay faults and cross talk faults are becoming increasing important. At-speed testing using external tester is almost infeasible solution because of its inherent inaccuracy and cost. Due to the need of design change, possibility of excessive power consumption, and high area and performance overhead, hardware-based self-test (BIST) is also not a feasible solution, particularly when we consider a high performance, low power optimized chip for mobile devices.

Recent research has been directed towards software-based self-test as an effective alternative of hardware-based self-test. Software-based self-test can be applied in the functional mode of operation. Therefore, it is the most suitable choice for testing optimized processor cores embedded deep inside an SoC. Moreover, the same test can also be used for the periodic online testing that contributes to system reliability, as it can be applied to a system in normal mode without affecting the system functionality.

This paper focuses on instruction-based self-testing methodology. Today, pipeline design is the most common design methodology. Therefore, it is very important to target pipelined processor. This paper is the first attempt for the delay fault testing of pipelined processors in functional mode of operation using its instruction set. We use path delay fault model as this the model which is most applicable to the modern high performance processors. A graph theoretical model is developed to model the behaviour of the pipeline

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

processors, and it is used for the test vector generation, and test instruction sequence generation under the architectural constraints.

The paper is organized as follows. Section 2 describes the previous work done in the domain of software-based self-test. Section 3 describes the overview of our methodology. Section 4 and 5 describe the test generation process for datapath and controller. Section 6 describes the instruction sequence generation process. Section 7 presents experimental results to demonstrate the effectiveness our methodology, and finally conclude with section 8.

2. Previous work

A number of software based self-test approaches [1,2,3,4,5], targeting stuck-at faults, have been proposed for unpipelined processors. The approaches proposed by Shen [1] and Batcher [2] are based on instruction randomization, and are more functional in nature. Both these approaches [1,2] give low fault coverage due to high level of abstraction and they generate large code sequence resulting in large test application time. Chen [3] 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. These self-test signatures are expanded into test sets using software LFSR during self-test and applied using a test application program. Due to pseudorandom nature of this methodology self-test code size and test application time are large. Moreover, efficiency of pseudorandom software based methodology depends on internal architecture and bit width. Krantis [4,5] proposed a methodology based on instruction set architecture and RT level description while using deterministic test sets (universal test set) to tests every functional component of the processor for all the operations performed by that component. Deterministic nature of [4,5] approach leads to reduced test code size but these methods find difficult to achieve high fault coverage for complex architectures as functional blocks are targeted. A methodology for pipelined processor based on the test template is presented by Chen [6], which uses statistical regression for function mapping. It may leads to long program size for complex architecture due to statistical nature. Krantis [7] proposed a methodology for pipelined processors based on RTL description and instruction set architecture. This approach uses deterministic test for functional blocks. Pashalis [8] proposed an online periodic test methodology for pipeline processors. Approaches [7,8] are targeting only functional blocks; hence it is difficult to achieve high fault coverage for complex architecture. Though [6,7,8] have considered pipelined processor but the pipelined behaviour is not considered in these, as these are largely focused on the functional blocks. None of the above stated approach target controller explicitly.

A few [9,10,11,12,13] software based self-test approach targeting delay faults for unpipelined processor have been proposed. The approach proposed by Lai [9,10,11], 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. This approach extracts constraints by exhaustively searching all possible instructions and instruction pairs which are applicable in functional mode, and that is not required. The results for the controller are not presented. The scope of this approach is limited to simple processors. An efficient graph theoretical model based approach was proposed by us in [12, 13]. This approach efficiently extract the constraints by using the graph theoretical model (called IE-Graph) for datapath and state transition graph model for controller and generate test vectors for datapath and controller. Constrained combinational ATPG is used for the test vector generation. These test vectors can easily be applied by using instruction set as these are generated under constraints. This model cannot handle pipeline processors. So far, no approach has been proposed for the pipelined processor. Pipelined behaviour of pipelined processor has not been used so far even for stuck-at fault model, which was the motivating factor of this work.

3. Overview of the proposed approach

The objective of this work is to develop a procedure for delay fault testing of pipeline processor core that can be used to generate tests for the functional mode of operation of a processor using its instruction set. Path delay fault model [14] is used in this work.

Unlike unpipelined processor which finishes execution of one instruction before the execution of next, pipelined processor executes multiple instructions simultaneously, which makes its behaviour more complex. Pipeline processor contains several independent units, called stages; and each stage execute concurrently. In order to support the concurrent execution necessary data and control signals are carried along as it progresses in the pipeline stages. Simultaneous execution of multiple instructions can leads to data, control and structural hazards. Data bypassing is a commonly used technique to resolve data hazards; stalling is used for the unresolved hazards. Data flow from the first pipeline stage to the last pipeline stage during the normal execution (without any hazard).

It is very difficult to separate datapath and controller part clearly in a pipelined processor as every pipeline stage carries all the data and control signals required by the pipeline stages ahead of that. We are considering the paths in datapath part and control part separately. We consider the data transfer activities between the architectural registers and data and address (memory address and register address) part of the pipeline registers in the datapath. Rest of the paths is considered in the control part. The activities in datapath are controlled by the control signals which are carried forward with the data; thus the function of datapath are constrained by the controller. Hence, only a subset of structurally applicable test vectors may be applied in the functional mode of operation due to the presence of constraints.

A graph theoretic model called pipeline instruction execution graph (PIEG) for the datapath has been developed that is constructed by using instruction set architecture and RT level description. This graph models the complex pipeline behaviour. This model can classify paths as functionally testable, functionally untestable, potentially functionally testable, and potentially functionally untestable. After the classification, it extracts constraints for the potentially functionally testable and potentially functionally untestable paths. First, constraints on the control signals in one or more relevant pipeline stages are extracted and then the constraints on justifiable data in the data registers or pipeline registers under the control constraints are extracted. Potentially functionally untestable paths are further classified as functionally untestable or potentially functionally testable paths. A combinational constrained ATPG is used for the test vector generation for the potentially functionally testable paths.

For testing the controller, the constraints on the legitimate values for a group of control signals are extracted by using RT level description. PIEG is used along with these constraints for further extraction of control and data constraints for target control paths, and their classification. Constrained ATPG is used to generate the test vectors. Finally, an instruction sequence to apply the generated test vectors, is generated by using the knowledge of the control signals of various pipeline stages and the PIEG.

4. Datapath

This part considers the paths that are responsible to transfer data between architectural registers or data and address part of the pipeline registers, which are significant in number. Other paths will be considered in the control part.

Datapath of a pipeline processor can be modeled by PIEG that is based on Instruction Execution Graph (IE-Graph) introduced by us in [12, 13] for non-pipelined processors. PIEG to captures the pipeline behaviour, and is used for the constraint extraction, path classification, and instruction sequence generation.

PIEG can be constructed from the instruction set architecture, and RTL description. This includes architecture registers and data and address part of the pipeline registers. (Note that this does not include the control part of the pipeline registers).

Nodes of the PIEG are: (i) registers, (ii) part of registers which can be independently readable and writeable, (iii) equivalent register (set of registers which behave in the same way with instruction set, like register file), (iv) two special nodes, IN and OUT, which models the external world such as memory and IO devices, (v) data and address (memory address and register address) part of pipeline registers.

A directed edge between two nodes is drawn iff there exists at least one instruction responsible to transfer data (with or without manipulation) over the edge (paths) between the two nodes (registers). Each edge is marked with a tupple [<*instruction set*>, <*stage from, stage to*>, <*delay*>, <*logic type*>]. This tupple signifies that a set of instructions <*instruction set*> is responsible for the data transfer from <*stage from*> stage to <*stage to*> through the logic type <*logic type*> which are generated by an instruction <*distance*> cycle ahead. The instructions which are responsible to latch data in destination register are included in <*instruction set*>.

If a pipeline register is a source node then the pipeline stage ahead of it will be <*stage from*> stage; otherwise, the stage controlling the data transfer activities in the register will be in <*stage from*>. If a pipeline register is a destination node, then the stage just before it (whose data it is latching) will be <*stage to*> stage, otherwise the stage which is controlling the data write activity in the register will be <*stage to*>.
Data transfer activities inside a pipeline stage is modeled by keeping the same <*stage from*> and <*stage to*> and zero distance, and the data transfer activities across the pipeline stages (mainly bypass paths) are modeled by using <*stage from*>, <*stage to*> and appropriate distance to create hazard in order to transfer data over that.
We will use our 16 bit, 5 stage pipelined processor VPRO design, which has most common 24 instructions. A complete PIEG of VPRO is shown in figure 1, and description is given in the appendix.

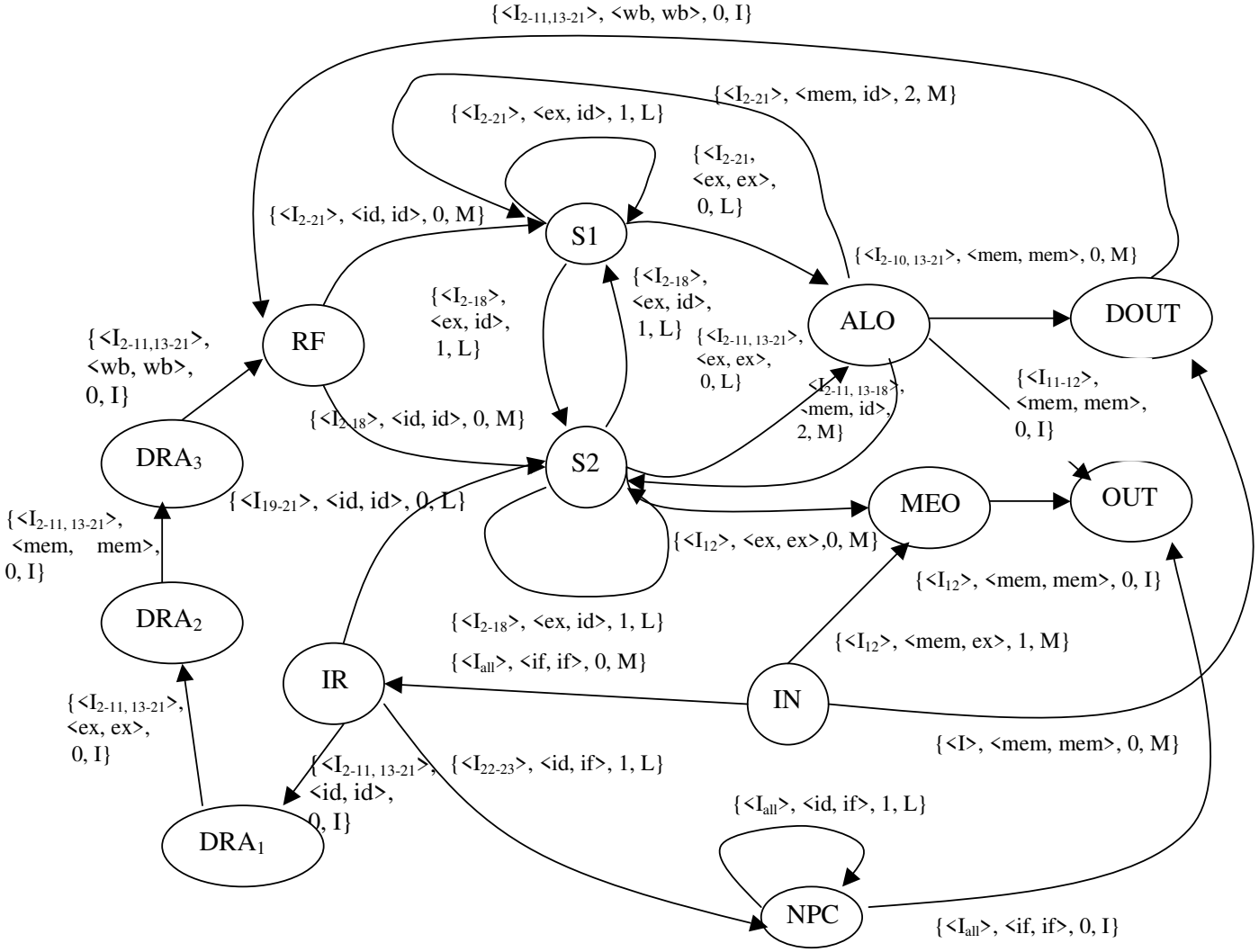


Figure 1: PIEG of VPRO processor

We have observed that many paths directly transfer data to the next stage using simple interconnect or through multiplexers. Keeping this view, logic is classified in three types i) interconnect (I), ii) multiplexers (M), iii) processing logic (L), which makes test generation process simple. This information can be obtained from RTL description even when full structural description is not available.
Instructions, which behave in the same way inside a pipeline stage, are defined as *equivalent instructions*, for that stage. For example ADD and INC are the equivalent instructions in EX stage.
We can use these equivalent instructions to reduce the marked instructions which in turn reduce the constraint extraction and test generation work. We make a table of the equivalent instructions for every stage, which can also be used for the instruction sequence generation.

We assume that any instruction can be followed by any other instruction in a pipeline stage except those instructions, which always need stall after the execution such as unconditional jumps.
We use normal data transfer to test the path except when the bypass paths (paths across the pipeline stages) are being tested. During the test of a path across the pipeline stages, we allow data propagation through normal path (distance = 0) other than the target path, as we are assuming that bypass paths are coming through MUXs, and can easily be disintegrated. Hence, we need not consider those paths during constraint extraction.

A path from register R_i to register R_o, marked with [$<I_{set}>$, $<S_j, S_i>$, d, LT] can be tested by a test instruction sequence IP_1, IP_2, ID_1, ……ID_{d-2} , IS_1, IS_2, where ID_1, ID_2, ……ID_{d-2} are the *(d-2)* dummy instructions which are responsible to transfer data from stage S_j to S_i. Instruction pair IP_1, IP_2 is responsible to create a transition at register R_i and allows it to propagate in S_j stage. Instruction pair IS_1, IS_2 is responsible to propagate the created transition in S_i stage and finally instruction IS_2 latches the result in register R_o. Other instructions ID_1, ……ID_{d-2} are used to provide the proper distance *d* between the instructions, so that path can be excited, propagated and result will be latched. A sequence of d+2 instructions is needed. Note that $IP_2 = IS_1$ if d=1, and $IP_1 = IS_1$ and $IP_2 = IS_2$ if d=0.

Instruction pair (IP_1, IP_2) must be marked on any zero distance (with d = 0) in-edge of R_i and instruction pair (IS_1, IS_2) must be marked on the target path (edge between R_i and R_o). If target path is inside a pipeline stage (d = 0) then we don't need dummy instructions, and are left with an instruction pair (IS_1, IS_2). The instructions IS_1 and IS_2 must be marked on any zero distance in-edge of R_i and at target edge respectively.

Path classification and constraint extraction:

Paths are classified in four categories, which are as follows:
(i) Functionally Testable Path (FTP), ii) Functionally Untestable Paths (FUTP), iii) Potentially Functionally Testable Paths (PFTP), iv) Potentially Functionally Untestable Paths (PFUTP).
There are two types of constraints, i) control constraints, ii) data constraints.
Control constraints are the constraints on control signals, which are responsible to transfer data between two nodes. These are obtained from PIEG. Data constraints are the constraints on justifiable data under the control constraints.

Let there be an edge between nodes R_i and R_o, marked with [$<I_{set}>$, $<S_j, S_i>$, d, LT]. Constraints are extracted as follows:
(1) when logic type is interconnect 'I':
These paths are generally used to carry forward data for the next stage and are always with d = 0. R_o has only one edge, and that is from Ri. Instruction sequence IS_1, IS_2 is needed to test. Any instruction marked on the zero distance (d = 0) in edge of register R_i, can be used as IS_1, and any instruction marked in the target edge can be used as IS_2. These two instructions give the constraint on the control signals in S_i stage. R_o has no other in-edge; hence, it will not observe any data constraint

These paths always get an instruction pair to test, and can be tested as the interconnect test. Therefore, these paths are classified as FTP.

(2) when logic type is multiplexer 'M':

These paths pass through a set of MUXs. These paths behave as interconnect if control signals are properly assigned. Therefore, under the control constraints (proper assignment of MUX select signals), data constraints are not applicable, as other paths to R_o will automatically be disintegrated with the proper assignment of MUXs control signals. These paths can be tested as interconnect test.

Path classification and control constraint extraction under different distance conditions:

(a) When d = 0 (Normal flow inside a pipeline stage):

These paths will transfer the data in the same stage. Therefore an instruction pair IS_1, IS_2 is needed to test these paths. Any zero distance (d = 0) instruction marked on the in-edge of R_i can be used as IS_1, and any instruction marked on the target edge can be used as IS_2. Instruction pair IS_1, IS_2 gives the constraints on the control signals in S_i stage. These paths always find a sequence of instructions without any data constraints, hence classified as FTP.

(b) When d=1 (data flow across the pipeline stages):

These are the paths from an adjacent stage. We need an instruction sequence of three instructions (IP_1, IS_1, IS_2).

If there is a self-loop, then we cannot launch a transition; hence, paths are classified as FUTP. Otherwise, any instruction marked at the zero distance in-edge of node R_i can be used as IP_1, any pair of instructions marked on the target edge can be used as IS_1, IS_2. Therefore, the instruction sequence IP_1, IS_1 gives the control constraints for the control signals of stage S_j, and the instruction sequence IS_1, IS_2 gives the constraints on the control signals in stage S_i. Hence, these paths are classified as FTP.

(c) when d>1 (data flow across the pipeline stages):

These paths are also responsible for the data flow across the pipeline stages. We need a sequence of (d+2) instructions (IP_1, IP_2, ID_1, ……ID_{d-2}, IS_1, IS_2) for testing. Instructions IP_1, IP_2 must be marked on any of zero distance in-edge of R_i, and IS_1, IS_2 must be marked on the target edge. Therefore, the instruction pair IP_1, IP_2 gives the control constraints on the control signals of stage S_j, and the instruction pair IS_1, IS_2 gives the constraints on the control signals in stage S_i. These paths are also classified as FTP.

(3) when logic type is processing logic 'L':

This includes the paths which are passing through the combinational logic. Let an edge between two registers R_i and R_o be marked with [$<I_{set1}>$, $<S_j, S_i>$, d, L>. Following edges and registers must be considered: i) All the in-edges to R_o with distance d and logic type 'L' (having some instructions common with I_{set1}), ii) all the in-edges to R_o with zero distance, logic type 'L', and have some instruction common with I_{set1}, iii) all the zero distance (d = 0) in-edges to R_i.

All those registers which have out-edge to R_o (with distance d (same as distance of target path), logic type 'L', and some instruction common with the target path) provide the data constraints for the propagation of created transition in S_j stage. All those registers which have out-edge to R_o (with zero distance, logic type 'L', and have some instructions common with the target path) provide data constraints for the propagation of the created transition in S_i stage. Figure 2 shows the edges and nodes which are needed to be considered. Note that $I_{set1} \cap I_{set2} \neq \phi$, Iset1 $\cap$ $I_{set3} \neq \phi$.

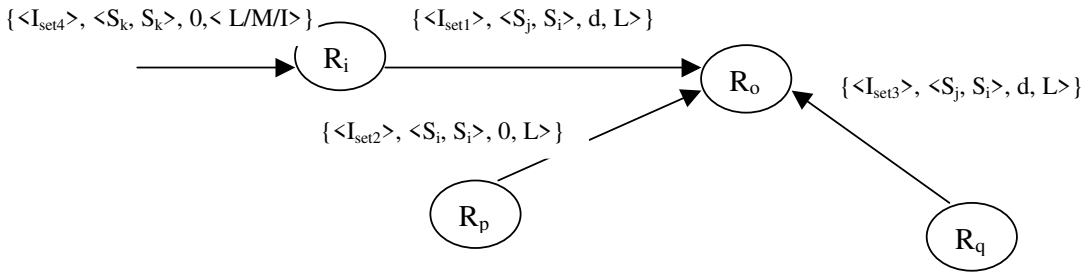


Figure 2: Edge consideration for the constraint extraction

We consider three different distance cases separately:

(i) when d = 0 (Normal flow inside a pipeline stage):

This includes the paths in a single stage. We need an instruction pair (IS_1, IS_2) for testing. IS_1 can be any instruction among the instructions marked on the in-edge of R_i, and IS_2 can be any instruction among the instructions marked on the target edge. An instruction pair (IS_1, IS_2) gives the constraint on control signal in S_i stage. We have to find out the data constraints on all those registers which have zero distance in-edge to R_o with logic type 'L' using PIEG and RTL description. Let R_o has an in-edge from register R_p which is marked with common instructions with the target edge.

(a) If selected IS_2 is not common with the other in-edges, then there is no constraint.

(b) If selected IS_2 is common with another edge from R_p then

(i) If selected IS_1 is also marked on zero distance in-edge of R_p then there is no constraint.

(ii) If selected IS_1 is not marked on any of zero distance in-edge of R_p, then R_p must have constant value across two time frames (under IS_1 and IS_2).

(ii) when d = 1 (data flow across pipeline stages):

An instruction sequence of three instructions (IP_1, IS_1, IS_2) is needed to test these paths. Any pair of instructions marked on the target edge can be IS_1 and IS_2, and any instruction marked on the zero distance in-edge of R_i can be IP_1. Therefore, the instruction sequence IP_1, IS_1 gives the control constraints on the control signals of stage S_j, and the instruction sequence IS_1, IS_2 gives the constraints on the control signals in stage S_i.

(a) when there is a self loop ($R_i = R_o$):

1. Paths from bit *i* to bit *i* :

   These paths can be functionally testable only when there is an odd inversion parity exists in the path, i.e, when odd number of gates which can invert the logic (NOT, NOR etc.) exists, otherwise, these paths are functionally untestable. These paths are declared as *PFTUP*. Many paths of such kind exist in the circuit, such as paths in the pass logic of ALU, paths in shifter, paths in logic operation block of ALU etc.

   This imposes data constraints on all those registers which have out-edge to the destination register R_o with unity distance, or zero distance and marked with some common instruction with the target edge. Constraints must be extracted for the register which has unity distance edge under IP_1, and IS_1 instructions, and for registers which have zero distance edge during IS_1, IS_2 instructions, as explained in d = 0 case.

2. Paths from bit *i* and *j*:

   These paths are PFTP and data constraints can be obtained as stated above.

(b) when there is no self loop:

These paths are classified as PFTP and data constraints can also be obtained in same way as in self-loop case.

(iii) when d > 1 (data flow across pipeline stages):

These paths need (d+2) consecutive instructions (IP_1, IP_2, ID_1, ……ID_{d-2} , IS_1, IS_2) to test, which consists of (d-2) dummy delay instructions to excite these paths. IP_1 and IP_2 can be any instruction marked at any zero distance in-edge of R_i, and IS_1 and IS_2 can be any instruction marked at target edge. Instruction pair (IP_1, IP_2) gives the control constraints on the control signals of S_j stage, and instruction pair (IS_1, IS_2) gives the constraint on the control signals of S_i stage.

Instruction pair (IP_1, IP_2) imposes constraints on those registers which have out-edge to R_o with distance d, logic type 'L' and some instructions common with the marked edge. Instruction pair (IS_1, IS_2) imposes data constraints to those registers which have zero distance out-edge of logic type 'L' to register R_o with some instruction common with marked edge. These paths are classified as PFTP, and data constraints can be obtained as explained in d = 0 case.

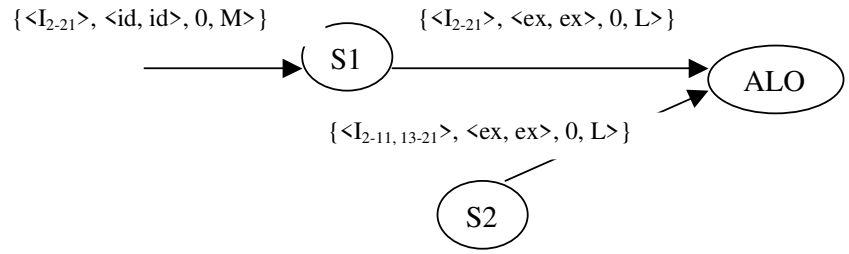


Figure 3: Edge consideration for the paths from S1 to ALO in VPRO

Example 1: Paths from the S1 to ALO in VPRO processor
Paths from S1 to ALO are marked with {<$I_{2\text{-}18}$>, <ex, ex>, 0, L} as shown in figure 3. This indicates that any instruction from I_2 to I_{21} can transfer data on the target path. We consider only normal data flow for the paths other than the target paths. Register R_i has only one zero distance edge marked with the same tuple {<$I_{2\text{-}21}$>, <ex, ex>, 0, L}. Therefore, IS_1 can be any instruction from I_2 to I_{21}, and IS_2 can also be any instruction from I_2 to $I_{21.}$ We can extract constraints on the control signals (alu control, and comparator control). We obtain total 8 constraints. Register S2 also has an out-edge with ALO and marked with {<$I_{2\text{-}11,\ 13\text{-}21}$>, <ex, ex>, 0, L>}, which provide the data constraints. Since register S1 do not have any instruction marked at the in-edge which is not marked at the zero distance in-edge of S2. Therefore, these paths do not have any data constraint.

Test vector generation:
Constraint ATPG is used to generate the test vectors after the extraction of constraints and path classification. Inversion parity test program is used to further classify PFUTP paths, and further classified as FUTP or PFTP. Constraint ATPG is used for test vector generation for all the PFTP paths by using extracted constraints.

5. Controller

This section deals with the paths that contribute to control signals. In order to execute an instruction, the instruction is decoded by the decode unit (in decode stage) and dispatch control signals along with the required data for the pipeline stages ahead. Therefore, each pipeline stage does have control signals that are not structured in nature but most of the time form a small group. In our approach, small grouping is used to find constraints. Therefore, we need to extract two types of constraints: i) constraints on the legitimate value of the group of control signals, ii) constraints on inter group signals in a pipeline stage.

(i) Constraints on the legitimacy of signals: Control signals generally form a group of small number of signals, and every possible value is not valid. Therefore we need to extract all the legitimate values. Test patterns must be generated under the legitimate values. For example, comparator control (*comp_ctrl*) signals in VPRO and DLX-PII are grouped in a group of 3 bits, and legitimate values are <0XX, 10X, and 110>.

(ii) Constraints on inter group signals: It is not only sufficient to consider the legitimate value for a group of signals but also need to consider the legitimacy of the inter group signals in a pipeline stage, as all the possible combination are not valid. We extract these in terms of instructions, i.e., map to the instruction which can generate the particular combination and all possible combinations are extracted. For example, in VPRO when ALU ctrl (*alu_ctrl*) signal is 0000 the comparator control (*comp_ctrl*) signal must be 000. Here onwards we will discuss how we can use these constraints for the test generation.

The part of a pipeline register, which carries the control signals is call control register. There may be paths between control register (CR) to control register, control register to data register (DR), or data register (such

as IR) to control register. The paths between CR-to-CR are used to carry the control signals for the pipeline stages ahead. These paths usually connected directly and these can always be tested as a byproduct of the test of CR to DR paths, or we can exclusively consider these and test as interconnect test. Paths from CR to DR are the paths which are passing through the combinational logic.
We construct a table which shows the transition on some bit in CR with instructions after exclusion of equivalent instructions.

Let there be a path between a bit *i* of control register C_i, and data register R_o. Constraints can be extracted in the following manner:

(i) when control register C_i and data register are in the same stage:
It needs an instruction sequence of two instructions (IS_1, IS_2). All those instruction pairs that can produce a transition at bit *i* and those are also marked on the in-edge of the register R_o can be the test instructions (IS_1, IS_2). All the data registers that have zero distance out-edge to R_o (has some common instruction with the selected potential instruction pairs) are needed to check for data constraints. Data constraints can be obtained in the same way as we obtain for datapath.

(ii) when control register C_i and data register R_o are in different stages:
Register R_o must have an edge from a register R_i that lies in the same stage in which C_i lies. This edge gives us the distance (say *d*), and we need (*d+2*) instruction sequence (IP_1, IP_2, ID_1 ……I_D_{d-2}, IS_1, IS_2) to apply test vector.
   (a) when d = 1:
   All those instruction pairs that can produce the transition and the second instruction also marked on the unity distance (d = 1) in-edge of register R_o, can act as IP_1, IS_1. Data constraints on the registers which have out-edge to R_o from the same sage (with distance d), can be obtained in the same way as it was shown in datapath. All those instructions which are marked on the above stated edge to R_o can also act as IS_2, and constraints on all those registers which have zero distance out-edge to R_o must be obtained under the instruction pair (IS_1, IS_2).
   (b) when d > 1:
   All those instructions which can produce the transition at bit *i* of C_i can act as IP_1, IP_2, and constraints on those registers which have out-edge with R_o (with distance d) must be considered under IP_1, and IP_2. All those instructions which are marked on the in-edge of R_o (with distance = d) can act as IS_1, IS_2, and data constraints on those registers which have zero distance out-edge to R_o must be considered under the control constraints of IS_1, IS_2 instructions. These data constraints can be obtained in the same as in datapath.

Test vector generation:
Constrained ATPG is used to generate the test vector for all the potentially functionally testable paths under the extracted constraints. ATPG is given with a PFTP with the extracted constraints, and ATPG will return the test vectors if paths are functionally testable.

6. Test instruction sequence generation

The generated test vector pairs as explained above are assigned to control signals and registers. A sequence of instruction is needed to apply these test vectors. This process needs following three steps

(i) Test instruction sequence generation
(ii) Justification instruction sequence generation
(iii) Observation instruction sequence generation

Test instruction sequence generation step, generates a sequence of instructions which is responsible to launch the transition, propagate the launched transition, and latches the result provided that desired data are available in the appropriate registers. These data are made available by the justification instruction sequence. Finally, the result must be transferred to memory by a sequence of instruction, called observation sequence.

(i) Test instruction sequence generation:

We have generated test vectors between two registers, under extracted constraints, and the information regarding the registers is available to us. We can use this information and PIEG to generate test instructions, which make this process simpler. It is clear from the earlier discussion that if an edge between registers R_i and R_o is marked [<I_{set}>, <S_j, S_i>, d, LT], then we need an instruction sequence (IP_1, IP_2, ID_1, ……ID_{d-2} , IS_1, IS_2) to apply the test vectors provided that test vectors are available in desired registers. Instructions IP_1 (when d > 0) and IP_2 (when d > 1) are decided by the control signals of the stage Sj, and instructions IS_1 and IS_2 are decided by the control signals of S_i stage. If there are more than one potential candidates for these instructions then we must select easy to observe instruction (such as STORE) for IS_2, and easy to justify instruction for the rest. Once IP_1, IP_2, IS_1, and IS_2 instructions are decided, we have to fill the rest of the instruction by NOP instructions which can be later on replaced by the justification instruction for IS_1 and IS_2 that can reduce the number of instructions.

Instruction sequence for the paths in the controller can also be generated in the same manner.

(ii) Justification instructions:

In order to generate test instruction sequence we assume that desired data are available in appropriate registers. We need to generate a sequence of instructions to justify data in the registers.

We cannot directly justify data in the pipeline registers. Therefore, we map back the data to either memory or architecture registers where we can justify easily. In order to do this, we construct a table that maps every pipeline data/address register to justifiable register or memory and to output register or memory when we execute some particular instruction. A table for VPRO is shown in part. Mapped input and output are shown in the table.

Table 1 Register mapping

Register	I_1	I_2	I_3	I_4	I_{12}
S1	-----	RF/RF	RF/RF	RF/RF	RF/mem
S2	------	RF/RF	RF/RF	RF/RF	RF/mem
ALO	-----	RF/RF	RF/RF	RF/RF	RF/mem
MEO	-----	-------	-------	------	RF/mem

We use this table to find the register/ memory where we need to justify data. Simple justification instructions are used. A special routine for special register is used.

(iii) Observation instructions:

Result from the register R_o must be transferred to memory. The instruction IS_2 transfers data to some output register or memory, and that information we can get from the table stated above. If it is transferred to memory then we don't need any observation instruction, otherwise we need to transfer data from register file to memory using STORE instruction. Special routine is used for the data transfer from the special register.

7. Experimental Results

VPRO processor has been synthesized using 2345 gates and 268 sequential elements. A complete PIEG of the processor is constructed by using instruction set architecture and RT level description. PIEG is used for the constraint extraction and the path classification. Constrained ATPG is developed, as commercially available ATPG do not handle the required constraints. Test vectors are generated under these constraints by using constrained ATPG. Some paths from the register S1 to register S1 and S2 to S2, and some paths from NPC to NPC are identified as potentially functionally untestable paths and later on classified as functionally untestable faults. Results for the Non Robust (NR) [14] tests are shown in the table 2. The results show that small number (about 14%) of paths are functionally testable. However, we have achieved 100% fault efficiency in the test generation. Present results show that we can achieve fault coverage better than the non-pipelined processor.

Table 2 Results for VPRO processor (NR test)

	Datapath	Controller
Total paths	112,752	98,786
Number of faults	225,504	197,572
Fault declared untestable in first phase	1,274	---
Functionally NR testable faults	32,134	27, 512
Functionally NR untestable paths	193,370	170,060
Fault coverage	14.2	13.9

8. Conclusion:

A systematic approach for the delay fault testing of a pipelined processor cores using their instruction set has been presented. A graph theoretical model has been developed to model the complex pipeline behaviour. This model can efficiently extract the constraints under which a processor can be tested. This model also assists the test instruction sequence generation process. Some paths can be declared as functionally untestable paths at the early stage. We would like to extend this model for the more complex processors such as super-scalar architecture in future.

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

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

[3]. 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.

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

[5]. 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.

[6]. Li Chen, S. Ravi, A. Raghunath, and S. Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", Proc. of the Design Automation Conference 2003, ACM Press, pp. 548-553.

[7] N.Krantis, G.Xenoulis, A.Paschalis, D.Gizopolous, Y.Zorian, "Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores", Proc. of International Test Conference, 2003, pp 431-440.

[8] A. Paschalis, D. Gizopoulos, " Effective Software-Based Self_Test Strategies for On-Line periodic Testing of Embedded Processors", Proc. of Design and Test in Europe 2004, pp 578-583.

[9]. 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.

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

[11].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 2001, ACM Press, NY, 2001, pp. 59-64.

[12] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Instruction-Based Delay Fault Testing of Processor Cores", Proc. of the International Conference on VLSI Design 2004, pp 933-938.

[13] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Delay Fault Testing of Processor Cores in Functional Mode", IEICE Trans. on Information & Systems. *To appear*

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

[15] J.L. Hennesy, and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, 1996.

Appendix: VPRO processor description

VPRO is a 16 bit, 5 stage pipelined RISC processor. It has 24 most common instructions. It uses load/store architecture. It consists of 8 general purpose 16 bit registers.

Instruction set:

1. NOP	7. MOV	13. SEQ	19. MVI
2. ADD	8. SLL	14. SNE	20. INC
3. SUB	9. SRL	15. SLT	21. DEC
4. AND	10. SRA	16. SLE	22. BEQ
5. OR	11. LOAD	17. SGT	23. BNE
6. XOR	12. STORE	18. SGE	24. JUMP

Instruction set architecture

It consists of 3 type of instruction

1. Register- Register type instructions
2. Immediate Instruction
3. Jump Instruction

Register-Register instruction:

2	3	3	3	3	2
OP	Rs1	Rs2	Rd	Fn2	Fn1

Immediate Instruction

2	3	8	1	2
OP	Rs1	Immediate 8 bit	Unused	Fn1

Jump Instructions

2	14
OP	Immediate 14 bit offset

Where

OP = Opcode (00 for R-R instructions, 01 for Immediate instructions, 10 for branch instructions, and 11 for unconditional jump)

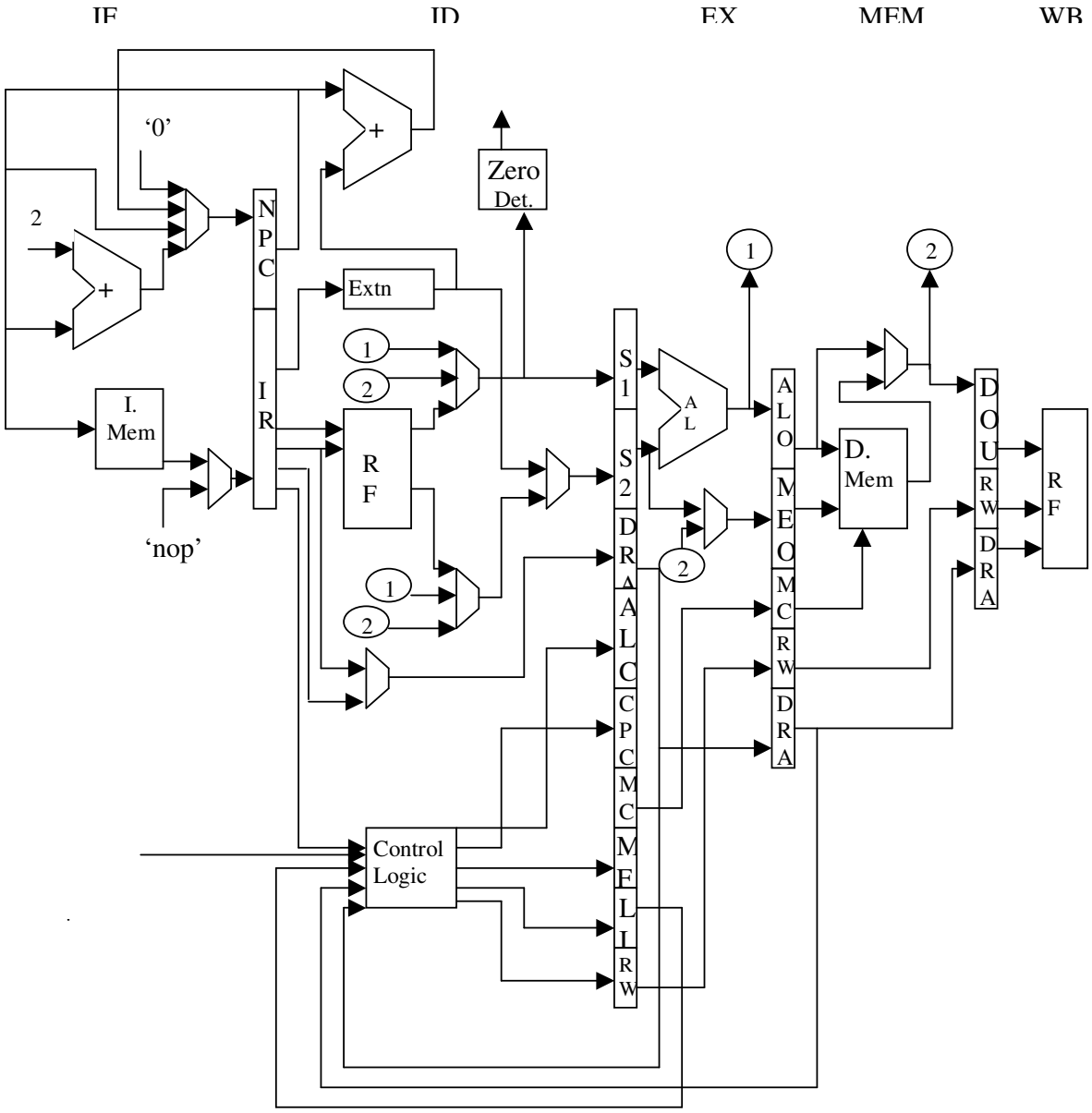
Rs1 = Source operand 1 regiter address, Rs2 = Source operand 2 register address

Rd = Destination register address

Immediate 8 bit = Immediate 8 bit data, Immediate 14 bit = Immediate 14 bit offset data

Instructions

Instructions	Operation	Op	Fn1	Fn2	Type
NOP	No operation	00	00	000	R
ADD Rd, Rs1, Rs2	[Rd] <= [Rs1] + [Rs2]	00	00	001	R
SUB Rd, Rs1, Rs2	[Rd] <= [Rs1] - [Rs2]	00	00	011	R
AND Rd, Rs1, Rs2	[Rd] <= [Rs1] and [Rs2]	00	00	100	R
OR Rd, Rs1, Rs2	[Rd] <= [Rs1] or [Rs2]	00	00	101	R
XOR Rd, Rs1, Rs2	[Rd] <= [Rs1] xor [Rs2]	00	00	110	R
MOV Rd, Rs1	[Rd] <= [Rs1]	00	00	111	R
SLL Rd, Rs1, Rs2	[Rd] <= [Rs1] sll [Rs2]	00	01	000	R
SRL Rd, Rs1, Rs2	[Rd] <= [Rs1] srl [Rs2]	00	01	001	R
SRA Rd, Rs1, Rs2	[Rd] <= [Rs1] sra [Rs2]	00	01	010	R
LOAD Rd, Rs1	[Rd] <= mem[Rs1]	00	10	000	R
STORE Rd, Rs1	Mem [Rs1] <= [Rd]	00	10	001	R
SEQ Rd, Rs1, Rs2	[Rd]<= 1 if Rs1 = Rs2 else 0	00	11	001	R
SNE Rd, Rs1, Rs2	[Rd] <= 1 if Rs1 ≠ Rs2 else 0	00	11	010	R
SLT Rd, Rs1, Rs2	[Rd]<= 1 if Rs1 < Rs2 else 0	00	11	011	R
SLE Rd, Rs1, Rs2	[Rd] <= 1 if Rs1 ≤ Rs2 else 0	00	11	100	R
SGT Rd, Rs1, Rs2	[Rd] <= 1 if Rs1 > Rs2 else 0	00	11	101	R

SGE Rd, Rs1, Rs2	[Rd] <= 1 if Rs1 ≥ Rs2 else 1	00	11	110	R
MVI Rd, #imm8	[Rd] <= [Imm8]	01	00	-----	I
INC Rd, #imm8	[Rd] <= [Rd] + [Imm8]	01	10	----	I
DEC Rd, #imm8	[Rd] <= [Rd] - [Imm8]	01	11	----	I
BEQ Rs1, #imm8	PC <= PC+ [Imm8] if Rs1 = 0	10	00	----	I
BNE Rs1, #imm8	PC <= PC+ [Imm8] if Rs1 ≠ 0	10	00	----	I
JUMP #imm14	PC <= PC+ [Imm14]	11	----	----	J

STRUCTURAL ORGANIZATION: