Efficient and Effective Test Program Generation for Software-Based Self-Test of Pipelined Processors*

Michiko Inoue Masato Nakazato Shinya Yokoyama Kazuko Kambe
Hideo Fujiwara
Graduate School of Information Science
Nara Institute of Science and Technology

Abstract

This paper presents a method of test program generation for software-based self-test of pipelined processors. We propose a model of pipelined processors and testability measures for registers. We generate a test program efficiently by means of specific behaviors of pipelined processors and combinational ATPG. Experimental results show that the proposed method obtains high fault efficiency.

keywords: *software-based self test, pipeline processor, test program, test program template*

1 Introduction

Today's processors with high performance and rich functionality absolutely require accurate and at-speed testing. Software-based self test (SBST) of processors attracts attention as a testing strategy achieving at-speed testing. In this strategy, a processor enables self-test by running a sequence of instructions called a test program. It does not induce any performance penalty, area overhead or excessive power. A number of approaches [1]-[14] are proposed for software-based self testing.

Pipelining is a key technology to enhance performance of processors, and it is essential for high performance processors. Though several SBST approaches have been proposed and some works [3, 10] use pipelined processors for their evaluation, all previous studies except [5] and [12] do not consider specific behavior to pipelined processors. Singh et. al[11] consider a delay fault testing where they mainly focus on how to activate path delay faults in an SBST approach, and extend it to handle pipelined processors. However, there are no discussion how to generate an entire test program for pipelined processors. A case study of SBST approach for some specific pipelined prooceessor is reported in [5]. The test program are developed through carefull consideration for the target processor, and obtain fault cooverage of more than 95% for stuck-at faults.

This paper proposes an SBST approach for pipelined processors. Our goal is to provide a method of *automatic* generation of test programs that obtain high fault efficiency for *structure* faults, such as stuck-at faults. The consideration for structure faults realizes an accurate testing, and the test program realizes at-speed testing. To generate test program for pipelined processors efficiently and effectively, we utilize specific behaviors of pipelined processors. Though architecture of pipelined processor is complicated to handle pipeline hazards, the data and control flows are simple if no hazards are encountered. We extract such flows as *hazard-free circuits* and use them to generate test program. We also facilitate test program generation by distinguishing pipeline registers and non-pipeline registers. The proposed method efficiently generates test programs with high fault efficiency by using combinational automatic test pattern generation (ATPG).

The rest of the paper is organized as follows. We first propose a model of pipelined processors in Section 2. Section 3 presents a method of test program generation for pipelined processors. The experimental results are shown in Section 4, and Section 5 concludes the paper.

2 Model of pipelined processor

Pipelined processors have complicated architecture to handle concurrent execution of multiple instructions in a pipeline fashion. The execution of one instruction is divided into stages, and different stages run for different instructions concurrently. Pipeline hazards complicate the architecture further since processors detect and resolve several hazards.

In this paper, we propose a simple but general model of pipelined processor (Fig.1). A pipelined processor is com-

*This work is supported in part by Semiconductor Technology Academic Research Center (STARC) and Japan Society for the Promotion of Science (JSPS) under the Grant-in-Aid for Science Research No.15300018.

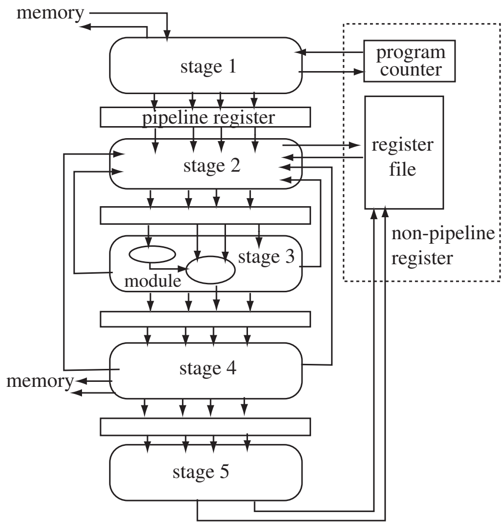

Figure 1. Model of pipelined processor.

posed of registers and combinational modules. Registers are classified into *non-pipeline registers* and *pipeline registers*. Non-pipeline registers appear in an instruction set architecture (ISA) and hold the values unless we explicitly update the values by instructions. A pipelined processor has a series of stages $1, 2, \cdots$, and each stage is composed of one or more combinational modules. Pipeline registers are placed between stages, and their values are updated at every clock. We consider that pipeline registers between stages i and $i+1$ belong to stage i.

An instruction is executed as follows. If the instruction does not encounter any pipeline hazard, data and control of the instruction go through stages $1, 2, \cdots$ in this order. We call such a flow *a hazard-free flow*. However, once some hazard is detected, data or control of one instruction might go to stage of another instruction to resolve the hazard. Moreover, some modules have inputs from different stages to detect hazards. In this model we do not explicitly distinguish control and data flows, which enables to handle all the module uniformly.

3 Test Program Generation

3.1 Ideas

Our test program generation method has two distinctive ideas: (1) use of hazard-free circuits, and (2) differentiation of pipeline and non-pipeline registers.

Architecture of pipelined processors is complicated by mechanism for detection and resolution of hazards. For this purpose, there are many connections between different pipeline stages, and they make test generation difficult. To generate a test program for a fault in some stage, we should activate the fault in the stage, justify adequate patterns of inputs of the stage, and propagate an error of the faults from outputs of the stage to primary outputs. Our test program generation method uses a *hazard-free circuit* to facilitate the above justification and propagation. The hazard-free circuit is a combinational circuit which represents a behavior of an instruction when no hazards are encountered.

In a processor, some modules may accept data or control flow from other stages. Such flows are used to resolve hazards, and they bring the same values when there are no hazards. For example, consider the following two instructions.

ADD $R1$ $R2$ $R3$
ADD $R4$ $R1$ $R5$

If two instructions are executed consecutively, the result of the first ADD is forwarded to the second ADD not through the final destination $R1$. However, if there are enough gaps between them, such a hazard does not occur, and the second ADD uses the value of $R1$. However, from the view of controllability, these two cases have the same controllability for pipeline or non-pipeline registers relevant to the second ADD. In justification or propagation processes, the controllability or observability of registers or signals should be cared. However, we do not need to care the exact paths to realize such a testability. From this observation, we consider only hazard-free data and control flows for the justification or propagation.

The second idea is to differentiate pipeline and non-pipeline registers. A pipelined processor has two types of registers. The pipeline registers are updated at every clock, while the non-pipeline registers hold the values unless they are explicitly updated by some instructions. The property of the pipeline registers is very suitable to consider time expansion model to generate test program, and enables us to use combinational ATPG. The property of the non-pipeline registers enable us to consider non-pipeline registers separately. Our strategy first select a few instructions to activate a fault using values of primary inputs and non-pipeline registers, and propagate an error to primary outputs or non-pipeline registers. We call such instructions *test instructions*. We generate a whole test program by concatenating the test instructions and the instructions to handle non-pipeline registers used in the test instructions.

3.2 Overview

Figure 2 shows an overview of the proposed method. We extract a hazard-free circuit for each instruction. We then generate a set of test instruction sequences for each combinational module. The test instruction sequence for a module delivers the test patterns to the module to activate

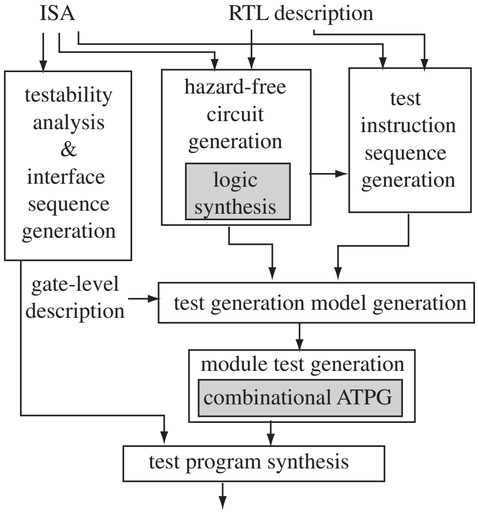

Figure 2. Proposed test program generation method.

1:	LHI	R1 *x1*	interface justification instruction sequence
2:	ADD.I	R2 R1 *x2*	
3:	LHI	R3 *x3*	
4:	ADD.I	R4 R3 *x4*	
5:	LHI	R5 *x5*	
6:	ADD.I	R6 R5 *x6*	
7:	LHI	R7 *x7*	
8:	ADD.I	R8 R7 *x8*	
9:	LHI	R9 *x9*	
10:	ADD.I	R10 R9 *x10*	
11:	LHI	R11 *x11*	
12:	ADD.I	R12 R11 *x12*	
13:	LW	R10 R2 R4	test instruction sequence
14:	SUB	R12 R6 R8	
15:	ADD	R13 R10 R12	
16:	LHI	R14 *x13*	interface observation instruction sequence
17:	ADD.I	R15 R14 *x14*	
18:	LHI	R16 *x15*	
19:	ADD.I	R17 R16 *x16*	
20:	SW	R15(R17) R13	

Figure 3. Test program.

faults in the module. To obtain the values of operands of the test instructions and the values of non-pipeline registers used in the test instructions, we generate a test generation model for each test instruction sequence, and apply combinational ATPG to it. The test generation model is composed of the stage including the module under test and its surrounding circuit which transfer the values of operands and non-pipeline registers used in the test instructions into test patterns to the module, and propagates the test response of the module to primary outputs or non-pipeline registers. We generate such a surrounding circuit using hazard-free circuits for the test instructions.

We also analyze testability (controllability and observability) of non-pipeline registers and obtain interface instruction sequences which are sequences of instructions to justify the values of non-pipeline registers from primary inputs (interface justification instruction sequence) or to propagate the values of non-pipeline registers to primary outputs (interface observation instruction sequence).

Finally, a test program is synthesized by concatenating the interface justification instruction sequences for the non-pipeline registers used in the test instruction sequence, the test instruction sequence, and the interface observation instruction sequence for a non-pipeline register where the test response is captured by the test instruction sequence.

In the proposed method, some processes can be aided by commercial CAD tools written in shade boxes, and hence, it facilitates automation of the whole system.

Figure 3 shows an example for test program for forwarding control unit (FCU) in Fig.4, where $x1, x2, \cdots, x16$ are immediate values. FCU is a controller to decide which data is adopted among a hazard-free data flow and forwarded data flows from different stages. FCU has inputs from several stages to detect hazards and resolve them. Assume that FCU is in stage i and FCU has inputs from stage i, $i+1$ and $i+2$ and outputs to stage $i+1$. The instructions 13-15 compose a test instruction sequence. These three instructions generates the values of inputs of stage i from stages $i+2$, $i+1$ and i, and instruction 15 (ADD) propagates the values of outputs of stage i to non-pipeline registers. The first 12 instructions compose an interface justification instruction sequence to set the values of R2, R4, R6, R8, R10 and R12. The last instruction SW stores the value of R13 into the memory, where R15(R17) specifies the address where the value of R13 are stored and they are set by instructions 16-19.

To generate test program for FCU, we generate a set of test instruction sequences, and also generate hazard-free circuits for the instructions. Figure 5(a) shows a test instruction sequence LW, SUB, ADD, where each instruction has three operands: addresses of one destination register and two source registers. We generate a test generation model from the test instruction sequence(Fig.5(b)). The test generation model is a combinational circuit which has inputs corresponding the 9 operands and non-pipeline registers. This test generation model is generated by combining the stage including FCU and hazard-free circuits for instructions LW, SUB and ADD. We can obtain the values of operands (Fig.5(c)) and the values of non-pipeline registers used in the test instruction.

We finally concatenate the test instructions with speci-

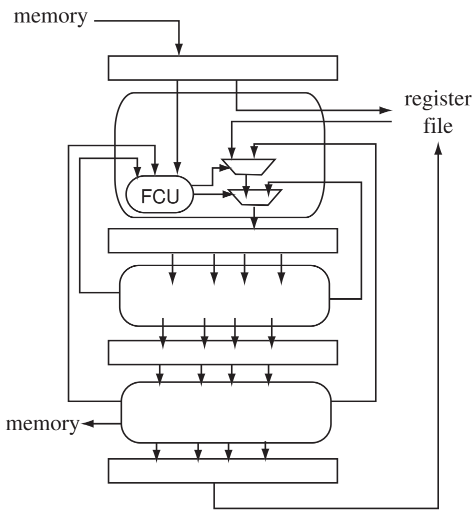

Figure 4. Forwarding control unit (FCU).

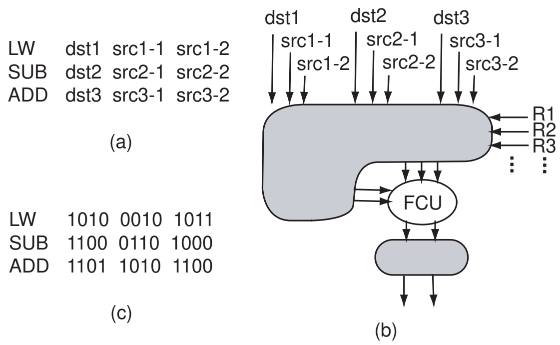

Figure 5. Module test generation: (a) test instruction sequence, (b)test generation model, (c)obtained operands.

fied operands and the interface justification and observation instruction sequences which are generated by the testability analysis of non-pipeline registers.

3.3 Hazard-free circuit

One of our distinctive idea is to use a hazard-free circuit. The hazard-free circuit for an instruction is a combinational circuit to represent data and control flows of the instruction when no hazard is encountered.

The hazard-free flow for an instruction is realized when enough number of NOP instructions precede the instructions. We generate a hazard-free circuit of an instruction from the RTL description as follows. We delete non-pipeline registers and treat their inputs and outputs as primary outputs and primary inputs. If there exist signal lines between different stages, cut them and treat them as primary inputs and primary outputs. Replace all the pipeline registers with signal lines. The resulting circuit is a kind of time expansion model. We them assign the value corresponding to the instruction to the corresponding primary input and the values corresponding NOP to the primary input corresponding to the input from other stages, and apply logic synthesis to the circuit. We eliminate parts of circuits which do not connect with any primary outputs, and obtain a hazard-free circuit. The obtained hazard-free circuit is a compact combinational circuit since many signals are reduced to constants by fixing an instruction.

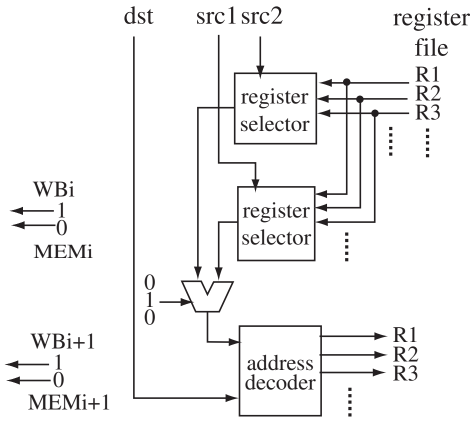

Figure 6. hazard-free circuit.

Figure 6 shows an example of a hazard-free circuit for instruction ADD, where some signals such as *alu_code*, WB(write-back) and MEM(memory operation) are fixed to constants. The register file is decomposed into the address decoder, a set of registers and the register selectors. In the hazard-free circuit, the address decoder and the register selectors appeare in different stages.

For each instruction, we also generate hazard-free circuits for consecutive stages 1 to i ($i = 1, 2, \cdots N_s$) and j to N_s ($j = 1, 2, \cdots N_s$) where N_s is the number of stages. We use these circuits not only to generate a test generation model but also to analyze the testability of pipeline registers.

3.4 Test instructions

We generate a set of test instruction sequences for a module under test (MUT). We first introduce *adjacent registers* for MUT. Adjacent registers of a module M are registers that have paths to M directly or only through combinational modules. We consider two kinds of adjacent registers: *input adjacent registers* and *output adjacent registers*. Figure 7 shows an example of adjacent registers.

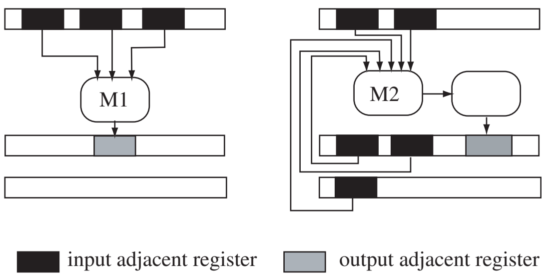

Figure 7. Adjacent registers.

We then determine the number of test instructions and gaps between them. A gap between two instructions means the number of instructions separating them in a test instruction sequence. For module $M1$ in Fig.7, one instruction can apply a test pattern from all the input adjacent registers, and can capture the test response in some output adjacent registers. On the other hand, for module $M2$ in Fig.7, we need three consecutive instructions to apply a test pattern from the input adjacent registers placed at three different stages. Since the third instruction also captures the test response, we need three consecutive test instructions. If we need some gaps in a test instruction sequence, we insert adequate number of NOP instructions.

Consider that an MUT is in stage i and there are input adjacent registers in stages i, $i + 1$ and $i + 2$. In this case, we need three consecutive instructions as a test instruction sequence. To select each of the three instructions, we first group instructions for each of stages i, $i + 1$ and $i + 2$. We check the signals corresponding to the input adjacent registers in the hazard-free circuits, and extract their controllability. In the hazard-free circuits, some signals are fixed to some constants, and some signals are connected with some primary inputs. We consider signals connected with some primary inputs have general controllability, while signals fixed to some constants have constant controllability to the constants. We analyze such controllability for each instruction. Then, if two or more instructions have the same controllability for all the signals corresponding to the input adjacent registers in one stage, we group those instructions together for the stage. For example, consider the instructions which execute some arithmetic or logical operations in an execution stage and write the results into the register file. Such instructions behave in the same way in later stages such as a memory stage or a write-back stage, and they are grouped together for these stages.

Let N_i, N_{i+1} and N_{i+2} be the numbers of groups of instructions for stages i, $i + 1$ and $i + 2$, respectively. We select one instruction from each group for each stage, and generate possible combinations of the selected instructions as test instruction sequences. In this case, we generate $N_i \times N_{i+1} \times N_{i+2}$ test instruction sequences.

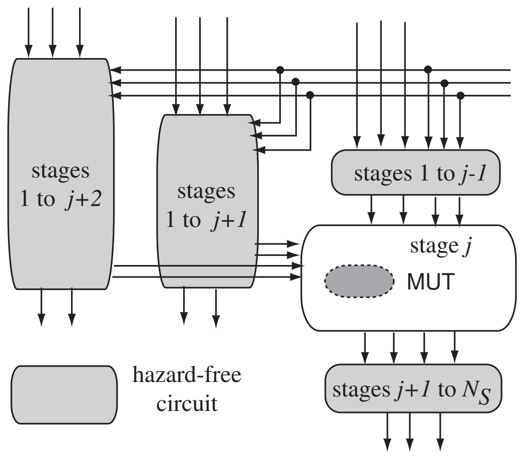

Figure 8. Test generation model.

3.5 Test generation model and module test generation

We generate a test generation model for each test instruction sequence, and apply combinational ATPG. The test generation model is composed of a stage including MUT and a surrounding circuit to deliver test pattern to the stage from primary inputs or non-pipeline registers and propagate errors from the stage to primary outputs or non-pipeline registers. For such surrounding circuit, we use hazard-free circuits for the test instruction.

Consider a test instruction sequence I_{i-2}, I_{i-1}, I_i where MUT is supposed to be activated in stage j of instruction I_i, and there are input adjacent registers in stages $j+2, j+1, j$ of I_{i-2}, I_{i-1}, I_i, respectively, and there are output adjacent registers in stage $j + 1$ of I_i. The test generation model is composed of stage j and hazard-free circuits for stages 1 to $j + 2$ of I_{i-2}, stages 1 to $j + 1$ of I_{i-1}, stages 1 to $j - 1$ and stages $j + 1$ to N_s for I_i (Fig.8). The stage j has inputs from stage $j + 1$, $j + 2$, and they are fed from stage $j + 1$, $j + 2$ of hazard-free circuits for I_{i-1}, I_{i-2}, respectively. The primary inputs corresponding to the same non-pipeline registers are shared among the hazard-free circuits for I_{i-1}, I_{i-2}, and I_i.

We apply combinational ATPG to the test generation model, and obtain the values of operands and the values of non-pipeline registers used in the test instructions. We call this process *module test generation*.

3.6 Testability analysis

We analyze testability (controllability and observability) of non-pipeline registers and generate interface instruction sequences. The interface instruction sequence propagates values from primary inputs to non-pipeline registers (interface justification instruction sequence) and from non-pipeline registers to primary outputs (interface observation instruction sequence).

The testability is analyzed using only ISA, since it specifies how to update the values of non-pipeline registers. We consider two kinds of testability: local and global. The local testability considers only one instruction, while the global testability considers a sequence of instructions.

We also introduce *equivalent* registers to efficiently analyze testability. A set of registers are equivalent if they behave in the same way for all instructions. We consider that registers in a register file are equivalent, and they are treated as a representative RF, or RF_1, RF_2, etc. if we want to distinguish them. These notations are mapped to actual registers when we generate a test program.

3.6.1 Controllability

The local controllability of a register for an instruction is represented by the general controllability C_g (arbitrary value can be justified), the constant controllability C_c (some constant value can be justified), the dependent controllability C_d (justifiable values are depended on non-pipeline registers), and the no-controllability NC. We extract the local controllability from ISA.

For example, a register RF in a register file has the following local controllability for instructions ADD.I and LHI. Let $2N$ be the bit width of data.

ADD.I
    $RF[N : 2N-1]$: $C_d(RF[N : 2N-1])$
    $RF[0 : N-1]$: C_g
LHI
    $RF[N : 2N-1]$: C_g
    $RF[0 : N-1]$: $C_c(00\cdots 0)$

Local controllability is expressed by the controllability of non-pipeline registers as well as the general or constant controllability. We then extend the local controllability of a register to the global controllability that represents how the register is controllable from the outside of a processor. In other words, if a register has general global controllability, there exists an instruction sequence that can justify arbitrary value to the register. We analyze the global controllability while constructing such a sequence.

We obtain global controllability of non-pipeline registers from local controllability of non-pipeline registers. A non-pipeline register holds its value unless it is updated explicitly by some instruction. This property enables efficient analysis for global controllability. Global controllability is analyzed by repeating reduction of dependency. If local controllability of some register R depends on other registers $R1$ and $R2$ for some instruction i, global controllability of R can be reduced to global controllability of $R1$ and $R2$. In this case, we can analyze $R1$ and $R2$ independently since we can justify values of both non-pipeline registers independently.

```
global_controllability(R){
  unresolved := {R};
  instructions := null sequence;
  while unresolved is not empty {
    get ur from unresolved;
    i := an instruction
      with minimum number of Cc or NC bits
      among maximum number of Cg bits;
    R1,R2,... : =registers some bits of ur depend on;
    insert R1,R2,... to unresolved ;
    append i at the beginning of instructions;
  }
}
```

Figure 9. Global controllability.

Figure 9 shows a procedure to analyze global controllability. In this procedure, a register can be a part of register like $R[a : b]$. If some register has different local controllability for different bits, we divide the register according to the local controllability. When global controllability of some register R is reduced to other registers, some bits in R may have C_g or C_c. We say that such bits are resolved in the reduction. The procedure derives a sequence of instructions that resolves all the bits, and if all the resolved bits have general controllability, the target register has global general controllability.

In the proposed procedure, RF is first reduced to $RF[N : 2N-1]$ by ADD.I, and then it is resolved by LHI. In this case, we find that RF has global general controllability and we can justify any value $x_0 \cdots x_{N-1} x_N \cdots x_{2N-1}$ of $RF2$ by the following sequence of instructions.

LHI $RF1\ x_N \cdots x_{2N-1}$
ADD.I $RF2\ RF1\ x_0 \cdots x_{N-1}$

We later use such a sequence for an interface justification instruction sequence. Though a non-pipeline register usually has global general controllability, it might remain C_c for some bits for the obtained sequence. In this case, we backtrack to the last selection and generate another sequence by selecting the next best instruction. If the obtained sequence has different patterns for bits with C_c compared to the sequences already obtained, we also record it as an interface justification instruction sequence of the register. However, this strategy might continue to select instructions infinitely. To avoid this problem, if selection of an instruction induces a loop of reduction, we do not select it. For example, we do not select an instruction that reduce controllability of R to R itself since the selection induces a self loop of reduction. We repeat this analysis until we obtain global general controllability or all the possible selections are tried.

3.6.2 Observability

We then extract local observability of registers for each instruction. The local observability is also extracted from ISA. The Local observability is represented by O_g(general observability), O_d(dependent observability), and NO(no observability) together with a set of non-pipeline registers called *support registers*. A support register is a register to be controlled in order to observe the target register.

For example, we can obtain the following observability from instructions SW (store word) and MFC0 (move the value of EPC to $R31$). To observe the value of RF by SW, we need to control two registers in a register file to specify the address where the value is stored. The support registers are represented as $+\{RF\}$.

SW
- RF: $O_g + \{RF_1, RF_2\}$
- $R31$: $O_g + \{RF_1, RF_2\}$

MFC0
- EPC: $O_d(R31)$

We then extend the local observability of non-pipeline registers to the global observability. In the case of observability, we do not need to find an instruction sequence to observe all the bits of target registers. If some registers have different observability for different bits, we generate instruction sequences for each part of bits according to the observability. Global observability are analyzed using an observability graph. An observability graph is a directed graph where nodes are the non-pipeline registers and a special node PO, and an arc $(R1, R2)$ exists if $R1$ has dependent observability for $R2$, or $R1$ has global observability and $R2$ is PO. Each arc is labeled by instructions and support registers. If a register has different local observability for different bits, the corresponding arc is also labeled by such information. Figure 10 is an example of observability graph. A register RF in a register file can be observed by an instruction SW by controlling two registers in a register file where the two registers specify the address of the stored word.

We analyze global observability of registers by finding paths to PO. We can observe the target registers with instructions along with the path and justification instruction sequences for support registers. We call such a sequence *interface observation instruction sequence*. For example, RF has the following observation instruction sequence that stores the value of RF at the memory address $x_0 \cdots x_{2N-1} + y_0 \cdots y_{2N-1}$

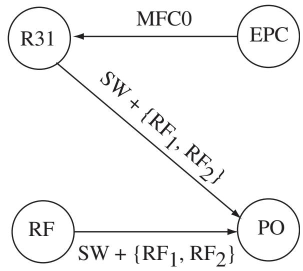

Figure 10. Observability graph.

LHI	$RF1\ x_0 \cdots x_{N-1}$
ADD.I	$RF2\ RF1\ x_N \cdots x_{2N-1}$
LHI	$RF3\ y_0 \cdots y_{N-1}$
ADD.I	$RF4\ RF3\ y_N \cdots y_{2N-1}$
SW	$RF2(RF4)\ RF$

3.7 Test program synthesis

Finally, a test program is synthesized from the test instruction sequence and necessary interface instruction sequences. In module test generation, we obtain the values of non-pipeline registers used in the test instructions. We generate the interface justification instruction sequences for such registers. We also generate the interface observation instruction sequences for non-pipeline registers that capture the test response. Consider the test instruction sequence in Fig5(c). We generate the interface justification interface sequences for 6 non-pipeline registers used as the source registers in three test instructions, and the interface observation instruction sequence for R13 where the test response is captured. We can generate a test program in Fig.3 by concatenating the interface justification instruction sequences, the test instruction sequence, the interface observation instruction sequence.

4 Experiments

We evaluated the proposed method using a pipelined processor DLX_N that is based on DLX processor[15]. The processor has standard 20 instructions excluding unsigned operations. We applied the proposed method to two typical modules ALU and FCU, where ALU is connected with one stage and need one test instruction while FCU is connected

Table 1. Results on ALU and FCU of DLX_N.

module	# faults	# test instruction sequences	# test patterns	test program length	# detected faults (MTG)	# detected faults (TP)	# identified redundant faults	fault coverage (%)	fault efficiency (%)
ALU	7026	20	160	1152	6994	6708	32	95.47	95.92
FCU	754	160	84	1263	648	648	81	85.94	96.68

with four stages and needs four test instructions. The experiments used a logic synthesis tool Design Compiler (Synopsys) and test generation tool TestGen (Synopsys).

Table1 shows the results for ALU and FCU. The target faults are stuck-at faults. In the table, "# detected faults (MTG)" and "# detected faults (TP)" represent the numbers of detected faults by the module test generation and by the synthesized test program, respectively.

For ALU, there are some faults detected in module test generation but not detected by the test program. That might be because we do not consider the effect of faults in justification or observation phases. On the other hand, the faults detected in the module test generation for FCU are also detected by the test program. However, there are some faults not detected in both. That is because FCU uses 10 bits to identify the current instruction though there are only 20 instructions. To identify faults that are not detected by any test program, we applied ATPG for the stage including MUT under constraint on some control signals. The values of control signals such as OP_CODE (operation code), FUNC_CODE(function code) are fixed by instructions. Therefore, the values that can be justified by instructions are restricted for the signals. We can treat such restriction as constraint for ATPG. As a result, we identified some faults not to be detected by any test program, and report the numbers at the column "# identified redundant faults". Fault coverage and fault efficiency are calculated from "# detected faults (TP)" and "# identified redundant faults". We obtain high fault coverage for ALU and high fault efficiency for both ALU and FCU. That is, we can detect most testable faults by test program.

5 Conclusion

We presented a method for test program generation for software-based self-test targeting structure faults for pipelined processors. The method efficiently utilizes specific behaviors of pipelined processors, and achieves high fault efficiency. Test program can be applied in the normal function mode of processors, and therefore, it realizes at-speed testing. Moreover, the proposed method does not modify processors, and hence it does not induce any performance penalty, area overhead or excessive power.

One advantage of the proposed method is that it can generate a test program without detailed information for modules. We only require a module to be a combinational circuit placed in one pipeline stage. We need information on connections between modules, but does not need the function or structure of the module. Moreover, we can treat one stage as one module. This property is desirable when there is no structured information on the inside of a stage in an RTL description.

However, there remain some faults that are not detected or identified to be redundant. Our future works include two approaches: to identify more redundant faults and to detect more faults. Since a processor is a sequential circuit, it seems very difficult to obtain complete fault efficiency without any design-for-testability. Therefore, one of our approach is to detect more faults in aid of design-for-testability.

References

[1] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. 17th VLSI Test Symposium*, pp. 34–40, 1999.

[2] D. Brahme and J. A. Abraham, "Functional testing of microprocessors," *IEEE Trans. on Computers*, vol. 33, pp. 475–485, June 1984.

[3] L. Chen, S. Ravi, A. Raghunath, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. 40th Design Automation Conference*, pp. 548–553, 2003.

[4] F. Corno, C. Cumani, M. S. Reorda, and G. Squillero, "Fully automatic test program generation for microprocessor cores," in *Proc. Design, Automation & Test in Europe, 2003*, pp. 1006–1011, 2003.

[5] M. Hatzimihail, G. Xenoulis, A. Paschalis, M. Psarakis, and D. Gizopoulos, "Software-based self-test for pipelined processors: A case study," in *Proc. 20th International Symposium on Defect and Fault Torelance in VLSI Systems*, pp. 535–543, 2005.

[6] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test program synthesis for path delay faults in microprocessor cores," in *Proc. International Test Conference 2000*, pp. 1080–1089, 2000.

[7] K. Kambe, M. Inoue, and H. Fujiwara, "Efficient template generation for instruction-based selt-test of processor cores," in *Proc. 13th Asian Test Symposium*, pp. 152–157, 2004.

[8] K. Kambe, T. Iwagaki, M. Inoue, and H. Fujiwara, "Efficient constraint extraction for template-based processor self-test generation," in *Proc. 14th Asian Test Symposium*, pp. 444–447, 2005.

[9] N. Krantis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-based self-testing of processor cores," *Journal of Electronic Testing: Theory and Application*, vol. 19, pp. 103–112, 2003.

[10] N. Krantis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Application and analysis of rt-level software-based self-testing for embedded processor cores," in *Proc. International Test Conference 2003s*, pp. 431–440, 2003.

[11] V. Singh, M. Inoue, K. Saluja, and H. Fujiwara, "Instruction-based delay fault testing of processor cores," in *Proc. International Conference on VLSI Design 2004*, pp. 933–938, 2004.

[12] V. Singh, M. Inoue, K. Saluja, and H. Fujiwara, "Instruction-based delay fault self-testing of pipelined processor cores," in *Proc. International Symposium on Circuits and Systems*, pp. 5686–5689, 2005.

[13] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," in *Proc. International Test Conference 1998*, pp. 990–999, 1998.

[14] C. H.-P. Wen, L.-C. Wang, K.-T. Cheng, K. Yang, W.-T. Liu, and J.-J. Chen, "On a software-based self-test methhodology and its application," in *Proc. 23rd VLSI Test Symposium*, pp. 107–103, 2005.

[15] J. H. Hennesy and D. A. Patterson, *Computer Architecture: A Quantative Approach*. Morgan Kaufmann Publishers, 1996.