Testing DSP Cores Based on Self-Test Programs[†]

Wei Zhao

Rockwell Semiconductor Systems
4311 Jamboree Road
Newport Beach, CA 92660

Chris Papachristou

Computer Engineering Department
Case Western Reserve University
Cleveland, OH 44106

Abstract

This paper presents a new method for the testing of the datapath of DSP cores based on self-test program. During the test, random patterns are loaded into the core, exercise different components of the core, and then are loaded out of the core for observation under the control of the self-test programs. We propose a systematic approach to generate the self-test program based on two metrics. One is the structural coverage and the other is the testability metric. Experimental results show the self-test program obtained by this approach can reach very high fault coverage in programmable core testing.

1. Introduction

1.1 Motivation

Design with cores has become popular recently. A core is a highly complex block that is fully defined, predictable and reusable in multiple design environments and different target technologies. Designers do not have to know the internal RTL structure of the cores, except their functions. Some advanced design tools such as hardware/software codesign tools and retargetable compilers can map part of the system behaviors onto these cores. Currently, cores such as RISC processors, bus interface controllers, image compression functions, floating-point processors are available in design libraries.

For design reuse, usually cores are programmable, for example DSP microprocessors. Testing these cores in different design environments is a new research topic. Not only because testing of programmable processors is much different from that of ASICs, but also because there is a need for a methodology which can be used in different situations. DSP cores are usually embedded in a system-on-chip together with with other components, including other cores. The fact that cores are usually embedded makes their testing more difficult due to accessibility problems. Our work is to develop a general methodology for testing DSP programmable cores.

Unlike microprocessors which are normally used in general purpose designs, embedded DSP cores are usually employed in specialized designs and with different technologies by different end users. Testing embedded DSP cores needs coordination with testing other components in a system-on-chip. In most situations, the process to propagate the faults in DSP cores is a tedious procedure. So a good DSP core testing scheme should have simple requirements on its neighboring components, and moreover it should be general and easy for integration.

1.2 Related Work

Functional testing of processors has a long history [ChMc76] [ThAb80] [BrAb84] [ShenSu88] [Tal89] [vGoor92]. However, functional testing achieves low fault coverage compared to structural testing because it does not consider the RTL structure and it is not based on a gate-level fault model (s-a-fault). More recently, some researchers began to combine functional testing with some information from the RTL structure to assemble self-test programs [LePa94] [LePa92] [Krug91] [BiMa95]. But all these techniques are not based on an accurate structural level testability analysis, and their approaches are more appropriate to microprocessors rather than embedded cores.

There are several reasons that make testing of embedded cores more difficult than testing of microprocessors:

1. When the chip designers license the core from another company, they usually do not have access to the internal structure of the core. This is because the core company wants to protect its intellectual property. This makes core testing very difficult. Thus the testing process can only rely on the core's behavioral level information and its brief architecture description. Also because of the intellectual property reason, the final chip designers are not able to modify the core to implement conventional testing or incorporate any DFT.
2. As there are other heterogeneous cores on the chip, the final chip testing should be coordinated with testing of different cores. Conventional testing schemes such as scan need to consider the testing of different components together. Usually this is very difficult. A self-test scheme is very attractive here because it has minimum requirement on its neighboring components.

1.3 Our Work

To solve the above new challenges in embedded core testing, we propose a general methodology for the testing of DSP programmable cores based on behavioral and structural level testability analysis. We develop self-test program consisting of several sections, each targeting different part of the core under test, not from the point of view of functions, but structure. supports peripheral random pattern generators and signature registers. The test program guarantees that the input random patterns are efficiently used in the core, meaning that they can be propagated as much as possible. Also it guarantees that the responses from the cores have good enough observability.

The main contributions of this work is:

1. A testing scheme that needs only the core's instruction set level information and brief architecture information. No gate level information necessary.
2. The work is based on instruction level testability analysis.
3. Our approach is based on pseudorandom BIST methodology which can reach high fault coverage but under the control of a self-test program.

2. Overview of Core Testing

Figure 1 shows the overall testing scheme which does not require any Design for Test (DFT) features of the DSP core under test. A a random pattern generator, LFSR, can be placed at the boundary of the core to provide pseudorandom test patterns. This LFSR can also be used to test other RTL components on the chip. A MISR can also be placed outside the core for the responses analysis. During testing, random patterns generated by the LFSR can be loaded into the core to exercise different part of the core, just as if the core accessed the external data. The responses are fed to the MISR for analysis. This testing scheme imposes less requirements on

[†]This work was partially supported by the Semiconductor Research Corporation (SRC) under Contract No. DJ-527.

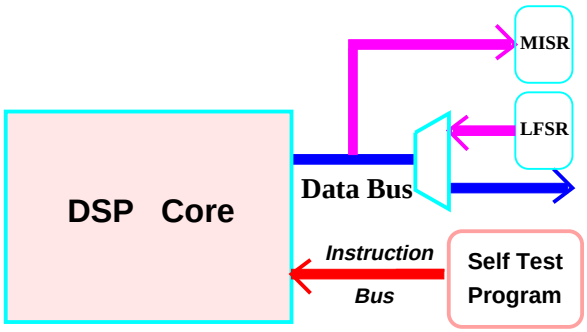

Figure 1: Overview of Testing Scheme

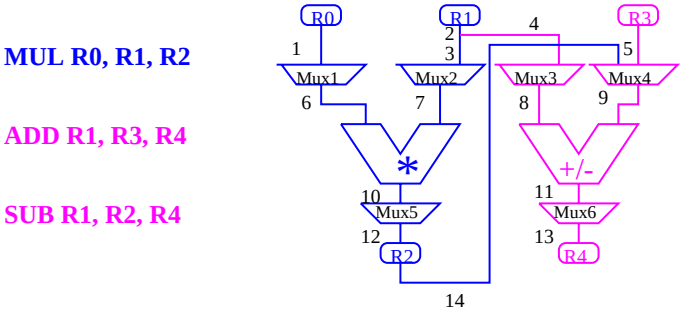

Figure 2: Instructions and Their RTL Components

its neighboring circuits, thus making it easy for system designers to work out overall testing strategy for the whole system-on-chip. Testing cost is reduced because our scheme does not require much overhead and can be reused in many designs.

DSP cores usually have a Harvard architecture with separate data bus and instruction bus, separate data memory and instruction memory, [PattHenn96]. The data memory and instruction memory can be further classified as on-core memory and off-core memory. In our discussion to follow, we assume the core has the following inputs and outputs:
a) Data bus : Input/Output. b) Instruction Bus : Input.
Under pseudorandom test, random test patterns can be fed into data input port rather than instruction port. Because random patterns in instruction – i.e. "random op-codes" – will make the core's behavior very unpredictable. Some special patterns may also lead the core to a so-called dead state which will make the subsequent testing meaningless. In our scheme, as shown in Figure 1, random patterns are only fed into the data bus port and the carefully assembled self-test program is fed to the instruction port to help the random patterns in datapath reach high fault coverage.

We assume that the testing of the data memory can be done using functional testing as well as BIST_RAM test technology, [Niko92]. Our approach is focused, but not limited to, the testing of the core's datapath. Any results generated by the test programs can indicate the faults not only within datapath, but also the controller, buses and other components. The key issue for testing cores is the self-test programs. From our discussion above, the test programs will direct random patterns that are applied to the core primary inputs to different parts of the core. After testing these parts, the result will be directed to the primary outputs for analysis. In this paper we will focus on the systematic synthesis of the core self-test programs.

3. Structural Coverage of Self-Testing

3.1 Structural Coverage

This notion indicates how many RTL components are covered by the self-test program. Structural information of the core is considered at the behavioral level, which means the self-test program can be measured not only from the functions it performs, but also from the number of structural components it uses. This is a basic difference between our work and other works in microprocessor-based testing.

In ASIC testing, structural coverage is not a problem. Most ASICs are designed using high-level synthesis tools, starting from a behavioral level description of the ASIC. For instance, in [PaCa95], they proposed a test synthesis approach to test the ASICs according to their behavioral descriptions. Because the ASIC is synthesized according to its behavioral description, using this behavior to drive the test can almost use all the RTL components of the ASIC, thus a very high fault coverage can be reached, [RoyAbr90]. But in DSP core testing, there are many behaviors which can be mapped onto this core by changing the DSP programs. Thus there will be bias or compromise if certain application behavior is selected to drive the core during testing. The following example shows the idea.

Suppose that we have a datapath as shown in Figure 2, there are three instructions, addition, subtraction and multiplication. The RTL components of the datapath are the ALU, multiplier, the registers, multiplexers and connecting wires. Obviously, addition can only exercise the 2-function ALU, the registers (R1, R2, R3), multiplexers (MUX3(part), MUX4 (part)) and the relevant connecting wires. Multiplication can only exercise the multiplier, R0, R1, and MUX1 (part) and MUX2 (part), and some connecting wires. During testing, if we only execute one of the above instructions, there will always be some RTL components which can not be tested. So in testing, we need a program which will cover all RTL components. In this special example, all these three instructions should be included in the self-test program.

Exhaustively enumerating all instructions is infeasible and unnecessary. First, exhaustive enumeration all instructions will lead to a very long test program which means a very long test session. Even for a simple DSP core which has 16 bit width instruction, enumerating all instructions will lead to a program with thousands of lines of instructions. Actually this lengthy program is not necessary. From the example in Figure 2, we know that both instructions will use R2 and its connecting wire, so there are some components that can be tested by more than one instruction. If a component is tested by one instruction, it is not necessary to have another instruction to test it again.

Based on the above analysis and observations, we know that a structural coverage metric should make the self-test program use as few as possible instructions to cover as many as possible RTL components of the core under test. This will make the self-test program efficient and effective. The metric summarizes the structural level information at behavioral level. The self-test program is not only based on function, but also based on structure.

Structural Coverage of a Program: The percentage of the RTL components in a core under test that are exercised by the instructions so that the faults within these components can be propagated to the core observable output port.

3.2 Reservation Table

We use a tabular data structure to bookkeep the structural coverage of a program.

RTL Component and its Space: A core's RTL structure can be divided into some basic components, each component either is used completely or not at all by an instruction. All these components constitute a space called RTL component space of the core.

To illustrate this concept we will use the example in Figure 2. According to the above definition, a component is either used or not used by an instruction. Table 1 lists all the basic RTL components in the RTL structure of Figure 2 in the

Instruction	*	+ -	Connection Wire													
			1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R0, R1, R2	✓		✓	✓	✓			✓	✓			✓		✓		
ADD R1, R3, R4		✓				✓	✓			✓	✓		✓		✓	
SUB R1, R2, R4		✓				✓				✓	✓		✓		✓	✓

Instruction	R0	R1	R2	R3	R4	MUXs						Structure Coverage
						1	2	3	4	5	6	
MUL R0, R1, R2	✓	✓	✓			✓	✓			✓		52%
ADD R1, R3, R4		✓		✓	✓			✓	✓		✓	48%
SUB R1, R2, R4		✓	✓		✓			✓	✓		✓	48%

Table 1: Instructions, Their Reservation Table and Structural Coverage

first row, which includes functional units, registers, multiplexers and connecting wires. In the second, third, and fourth row, all the RTL components that are used by addition and multiplication instruction are listed. We calculate the structural coverage of each instruction and the whole self-test program as follows. Suppose:

S	The RTL components space of the core under test
s_i	The RTL component set used by instruction i
n	The total number of instructions in a self-test program
SC	The structural coverage of a self-test program
SC_i	The structural coverage of instruction i

The structural coverage of a self-test program SC can be calculated by the following:

$$SC = \frac{\cup_{i=1}^{n} \mid s_i \mid *100}{S}\%$$

The above definition can also be extended to calculate the structure coverage of one instruction as following:

$$SC_i = \frac{\mid s_i \mid *100}{S}\%$$

From the above, the structural coverage for the addition and multiplication instructions in Figure 1 are 48%, 52%, respectively And the self-test program which includes these two instructions has a structural coverage 96%.

Sometimes, because of the complexity of the core architecture, it is not so straight forward to calculate the structural coverage. For example, in CISC microprocessor cores, some instructions are implemented by a rather complex microinstruction stream. Thus, analysis of structural coverage should be further extended to the microinstruction or microoperation level. Another difficult situation is the involvement of the controller and other auxiliary RTL components. In the above discussion, we define the structural coverage by counting the RTL components that are used by an instruction. There are ambiguities. For example, every instruction will use the Program Counter(PC), but actually, the random patterns are not applied to PC. So PC is not randomly tested by this instruction. This tells us that we need to further classify the term "used by". Here, we only count those RTL components that are exercised by random patterns in one instruction.

We use an example to show the previous idea in some detail. Figure 2 shows an DSP core architecture. In Figure 3, on the left side box, we have an instruction flow for this architecture. In the middle is the $CDFG$ (Control Data Flow Graph) representing a fragment of this instruction flow. On the right side, we have the corresponding microinstruction flow graph (MIFG) Each node in MIFG represents a microinstruction, the edges represent the dependences among the microinstructions. PI and PO are the primary inputs and outputs.

In Figure 4, we have the $MIFG$ on the left side. The bold line shows the path from PI to PO. Here this path is like a sensitized path in gate level stuck at fault testing. The RTL components used by the instructions along this path are randomly

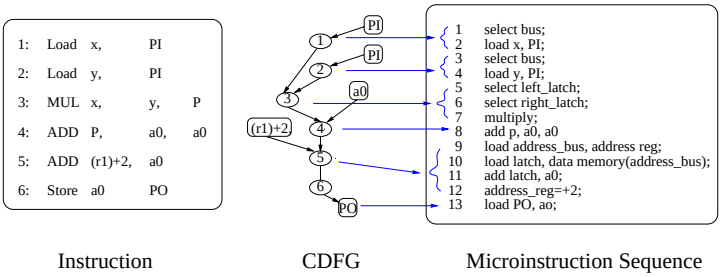

Figure 3: Instructions and Their Microinstructions

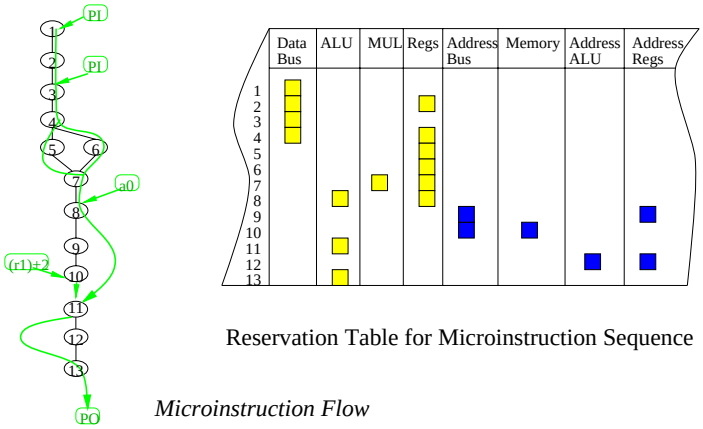

Figure 4: Testing Path and Reservation Table

tested by the random patterns. Not all microinstructions are on this path. Only those microinstructions that process the random patterns from PI to PO are on this path. So not all the RTL components that are used by the instructions of the self test program are being tested by the current test program. We use the $MIFG$ to distinguish these differences between all the RTL components that are used, and the RTL components that are tested by a self test program. $MIFG$ is similar to the SDG used in [LePa94]. The difference is that we only annotate the structural information along the path through which the random patterns flow. We actually keep the annotation information in a reservation table as shown on the right hand side of Figure 4. The boxes indicate which RTL components are used by this self test program in which micro steps. Only the light gray boxes are tested by this self-test program.

We use *static* and *dynamic* reservation tables in our approach. The static one is used by each individual instruction. Actually, once the core's architecture is decided, the table can be decided. As shown in Table 1, one row is the static reservation table for one instruction. It indicates which RTL components can be randomly tested by using this instruction. For some instructions with variations, there will be more than one entry in this table to indicate this difference. This table is statically decided.

The second table is dynamically built by the self-test program assembler during the assembly process. It is a run time table maintained by the assembler during run time. Each column in dynamic table is an RTL component in the core, each entry of this table is an instruction. Each row in the dynamic table corresponds to one instruction or microinstruction as shown in Figure 4. With the development of the instructions in the self-test program, more and more RTL components will be tested by the random patterns, and the structural coverage will reach its highest. The structural coverage of a self-test program can be easily calculated from this table by accumulating the RTL components test by random patterns. The

dynamic reservation table is used as a guide in the process of making the following two decisions:

1. Instruction to be added to the self-test program.
2. Where to stop the self-test program.

By using the reservation table, the intellectual property of the core design company can be protected. The static reservation table can be generated by the core company and shipped to the designer. Our approach can help the designers generate a self-test program without accessing the internal RTL structure of the core. And the result it can reach is very close to that of structural level testing approaches which depend on a complete knowledge of the gate level netlist of the core.

The dynamic reservation table enables us not to assemble one fixed self-test program for one core. Because different design may have different design considerations and different test requirements; leaving this to the final designers will provide them more flexibility. On the other hand, many cores are now parameterized, helping different designers who may need different configurations; this forces us to leave the testing decision, retargetable self-test programs, to the final designers.

4. Testability Metrics

As we discussed in the previous section, we need to exercise as many as possible RTL components to reach high fault coverage. This leads us to the first type of metrics for self-test program which is structural coverage. But a self-test program with high structural coverage may not be able to detect the faults and may not be able to propagate the faults perfectly. This leads to the second type of metrics for self-test program which concerns testability quality.

In our early works, we have developed two testability metrics, randomness and transparency, to evaluate the controllability and observability, respectively, of signals embedded within a behavior. Specifically, *randomness* is a controllability metric that quantifies the quality of pseudorandom patterns as they propagate through embedded modules. Also, *transparency* is an observability metric which quantifies the sensitivity of embedded modules to erroneous value propagation to an observable point. Reference [PaCa95] provides a complete discussion about the basic concepts of these metrics and their application to test synthesis of ASICs in the behavioral domain. These two metrics have wide applications to analyze behavioral level signals in terms of their controllability and observability properties. In our previous research, we used them to analyze the behavioral descriptions of the design, usually provided in VHDL. In this work, we apply them to analyze the variables of a self-test program.

Recall that in section two, we discussed our overall testing scheme. The self-test program will first load the random patterns from the primary inputs to the registers, then it will carry out some computations to exercise some parts of the core under test, and then it will route the results to the primary output for observation. Figure 5 continues to use the example that we discussed in previous section to show the idea. The self-test program is listed on the left hand side, its corresponding data-flow graph(DFG) is shown on the right hand side. We use exactly the same testability metrics introduced in [PaCa95] to analyze this section of the program. The results are back-annotated on the DFG. We assume that the input data have the maximum randomness because they are coming from a LFSR, which can have perfect randomness if proper seeds are given in practice. In Figure 5, R0,R1,R3, have perfect randomness and transparency because they are all LFSRs. R2's randomness is 0.9621. Its transparency has two values. One indicates its left input and another indicates its right input, and the numbers are 0.8720 for the left, 0.8764 for the right. As both values are not 1, so the faults with its left and right inputs can not be propagated to the output which is R4. About the details of the computation and definitions of these metrics, interesting readers may refer to [PaCa95].

From this example, we can see that even if a self-test program covers all the RTL components, it may still lead to low

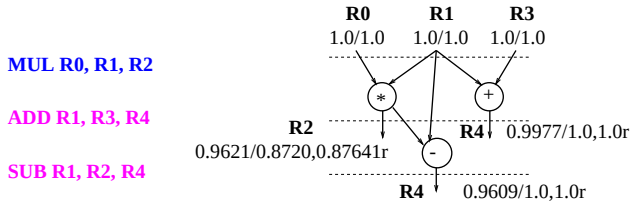

Figure 5: Testability Metrics of Self-Test Program

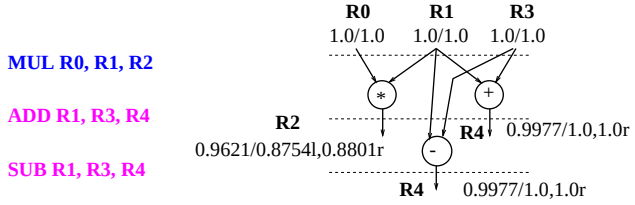

Figure 6: Improved Self-Test Program Metrics

fault coverage because of the characteristics of some computations, like multiplications and some bit operations. They can either generate results with very low randomness or block the faults propagation. Our previous research in [PaCa95], suggested some behavioral level transformation to improve these metrics. In this work, there is no room for redesigning the core to improve testability. An adaptation of our previous research is to carefully organize the self-test program to improve testability of the core. From this point of view, a self-test program is a core's test behaviors, expressing the behavior of the core in test mode.

Figure 6 shows a better version of the above self-test program. Table 2 lists each variable's testability metrics. The table abides by the following two general rules which are used in our systematical assembling of self-test program:

1. The inputs variables to each instruction should have the best randomness (controllability).
2. If the output variables generated by some instruction have low observability, they should be sent out for observation.

The first rule is similar to the controllable point insertion and the second rule is similar to observable point insertion in [PaCa95]. More details of this techniques can be found in the same paper. This testability analysis is done "on-the-fly" with the assembling of instructions in the self-test program. Whenever a new instruction is put into the self-test program during assembling, the testability analysis will be invoked and the results are used to make further assembling decision, regarding the next instruction. More details follow next.

Metrics	R_0	R_1	R_2	R_3	R_4	R_5
Controllability	1.0	1.0	0.96	1.0	0.99	0.96
Observability (Left)	1.0	1.0	0.87	1.0	1.0	1.0
Observability (Right)	1.0	1.0	0.87	1.0	1.0	1.0

Table 2: Testability Metrics

5. Generation of Self-Test Program

We use heuristics to assemble a self-test program. These heuristics will guarantee that the self-test program has both high structural coverage and good testability metrics.

5.1 Organization of Self-Test Program

We do not need to build the test program from scratch. Actually there is a **template** for self-test program. Each template

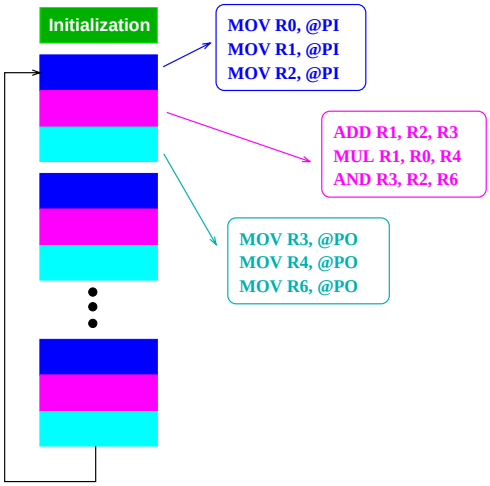

Figure 7: Templates in Self-test Program

can be divided into three consecutive sections according to different functions as shown below:

1. **LoadIn Section** Instruction(s) loading the random patterns from PI to where we want them to test the core.
2. **Test Behavior** Instructions to test different parts of the core. This is designed to achieve high structural coverage and high testability metrics.
3. **LoadOut Section** Instruction(s) moving the results to PO for observation and analysis.

The first section is composed of data transfer instructions. Some destinations, usually registers and memory addresses are necessary for this section of self-test program. The third section is also composed of data transfer instructions. The destinations are usually the PO buses.

The second section, which is called the **test behavior**, is the key section of the self-test program that we will focus on.

Figure 7 shows the template idea. The whole self-test program is composed of many of these instantiations of templates. Each instantiation targets certain part of the core. Each instantiation has good testability and all the instantiations together cover as many RTL components as possible.

5.2 Classification of Instructions

We classify the instructions into several groups as a preparation for this assembling process. The purpose of this classification is to help the assembler to pick up instructions which target different parts of the core under test. We also give different weights to different instruction groups so that the assembler will pick up instructions first from the group with highest weight. To make the assembling process dynamic so that the assembled instruction can also guide the selection of next instruction, these weights can be adjusted after instructions are selected from each group. This mechanism will help with assembler uses as few instructions as possible to cover as many RTL components as possible.

We classify the instructions according to the principles:

1. Based on the major RTL components that will be exercised. For example, ADDITION and SUBTRACTION are all implemented by the ALU. So we put them in one group.
2. The number of common RTL components that the instructions will use. For example, AND and OR instructions will mostly use the same RTL components.

The first principle is good for complex datapath-dominated cores because there are more than more functional units. This principle can make the classification very simple, effective and easy to use. The second principle is more generous. Actually it can be automatically generated if the static reservation table of each instruction is available. The *distance* between each instruction can be defined as the Hamming distance of the vector of RTL components usage of each instruction. For the three instructions in Table 1, we will have the following distance between them:

$$D_{mul,add} = 25 \quad D_{add,sub} = 3 \quad D_{mul,sub} = 23$$

Accordingly, the addition and subtraction will be in one group. Multiplication will be in another group. So if the self-test program first pick addition, it will avoid to pick up subtraction next because it has less chance to increase greatly in structural coverage because it is in the same group as addition.

Once the distance is decided, many existing clustering algorithms can be used to cluster them into different groups. In real practice, we assign weights to each RTL component in the static reservation table, so the distance between different instructions can be a weighted Hamming distance.

5.3 Weights to Instructions and Instruction Groups

We assign weights to each instruction so that we can more accurately link each instruction with the number of RTL components that it can exercise and then number of faults it covers. By establishing this link, our assembler can know which instruction is more important than others.

We still use the previous example to illustrate the idea. Figure 2 shows a partial datapath. Here we consider two instructions, one is addition and another is multiplication. The static reservation table is shown in Table 1. For all the RTL components, if we treat them equally, obviously, it is not fair. Because some components, such as multiplier, have more potential faults than the ALU(adder/subtracter). So for different instructions, because of their mapping to different RTL components, we assign different weights to them, according to the number of potential faults that these RTL components have.

5.4 Heuristics in Instruction Operand Fileds

After the random patterns are loaded into the on-core memory or register file, these data can be used to test the core. As for the address, either the register address or the data memory address can fill the corresponding instruction fields as the sources of the instructions. And the results of the instruction can also be stored in the register file or data memory by specifying the destination field in instructions.

The unused data loaded from LFSR is called "fresh" data. We hope that all operations can use the "fresh" data as much as possible. The loading of the data takes a lot of CPU time, i.e. testing time. So we try to load less data and use the load data as efficiently as possible. In the course of the assembling, we maintain a table for all the memory elements, to indicate each element's testability metrics. The current instruction will avoid using the "old" data whose testability metrics are not good. They will always try to use the "fresh" data. If there is no enough data for use, some instruction will be inserted to load more fresh data into core's memory elements.

Figure 8 shows an example developed from the one in Figure 6. Figure 8 shows the variables' mapping onto the core's memory elements. After the three instructions, the shaded memory elements are already tested, the variable in R2 is not good in testability metrics, so the later instructions should not use this variable. R2 variable needs to be loaded out and a new "fresh" data needs to be loaded in it. In addition, the further instructions will try to use the unshaded memory elements. This will guide the selection of operands fileds in assembling of instructions.

5.5 Randomness in Instruction Operand Fields

We also keep a randomness mechanism to decide the instruction's operand fields. The purpose is not only to test the datapath, but also to test the controller, memory element, the relevant connections and so on. But the random number filled to the operand field in this way should be under control and

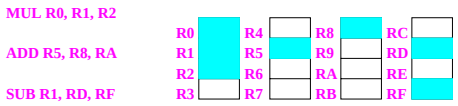

Figure 8: Heuristics to Guide the Selection of Operands

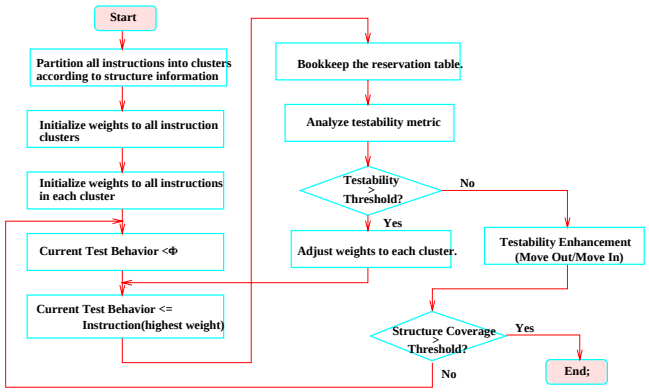

Figure 9: Self-Test Program Assembling Procedure

valid. For each instruction, there will be a randomness space for each field. Any random number should be in this space. Also if there are several fields for random number, their combinations should also be in a bigger valid space. Here we try not to generate an instruction which will lead the core to unstable or uncontrollable states.

5.6 The Heuristic Assembly Algorithm

Figure 9 shows the flow chart of this assembly procedure. From this flow chart, we can see that the two metrics for the self-test program are guaranteed. If the structural coverage is not met, the procedure will continue to select instructions to cover more RTL components. In each assembling template, once the testability metrics become bad, the procedure will stop to assemble new instructions but to insert some other instructions, for example, to add LaodIn and LoadOut sections, to improve the testability metrics of the program. The testability metrics controls the inner most loop, and the structural coverage controls the whole assembling procedure. The final program generated by this procedure will have both high structural coverage and high testability metrics.

6. Experiments

6.1 Experimental Environment

Figure 10 shows the experimental environment. There are several major tools that are involved in this experiment. The Self-Test Program Assembler(SPA) is the procedure which will generate a program in its assembly code. SYNTEST [HPCN92] is used for testability analysis [PaCa95], COMPASS [Comp94] is used for core architecture compilation to generate the gate level netlist and Gentest from AT&T for fault simulation.

There are mainly two flows in this experiments as shown by the arrow line in the Figure 10. One is the hardware flow. The other is the software flow. The hardware flow starts with the VHDL description of the architecture of the core under test. This is fed to COMPASS's ASIC synthesizer [Comp94] to get netlist file and then gate level VHDL descriptions.

The software flow starts with our assembling procedure, the resultant assembly code of the self-test program or that of the normal application programs as shown in Figure 10, is then fed to the core's assembler to get the binary code. The VHDL gate level description of the core and the self-test program's binary code are finally fed to Gentest for fault simulation. To ensure that the binary code is correct and the fault simulator works

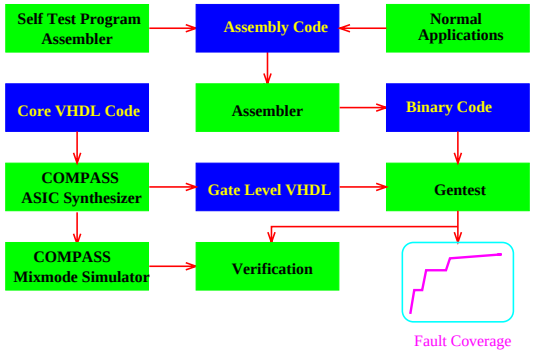

Figure 10: Experimental Test Environment

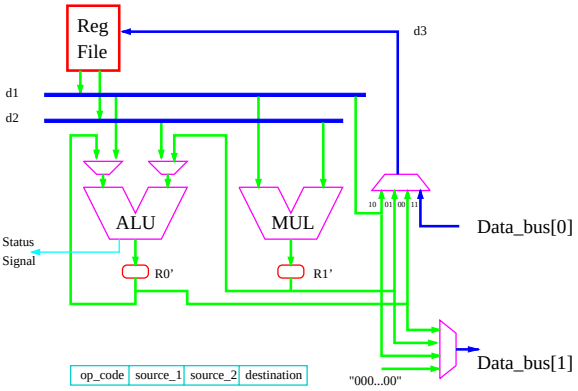

Figure 11: DSP Core Architecture

properly, a verification procedure is introduced to compare the simulation results from the fault simulator and the COMPASS simulator. Some software to link different tools together are developed in our lab to assist this experiment.

6.2 Experimental Core

We implemented a DSP core to test our approach. It has 19 instructions with some DSP features, such as multiplication and accumulation. Its architecture and instruction set are shown in Figure 11 and Figure 12, respectively. Although the instruction set is small, it is fully functional. Branch is implemented after a compare instruction. The following word has the branch taken address and the second following word has the branch not taken address. Register read, operation and write back to register will take two clock cycles. There are totally 24444 transisitors in the core's datapath generated by COMPASS ASIC synthesizer.

6.3 Analysis of Results

We compared the self-test program generated by our approach with those normal application programs. By normal applications, we means that during the random testing, normal application programs, for example, FFT and FIR filter, are fed into the core to control the core's behavior. The results are shown in Table 3. We can see that the normal application programs, because of their low structural coverage and low testability, have ivery low fault coverage. This is simply because not only there are a lot of RTL components that they do not exercise but also because even they exercise some RTL components, the faults within those RTL components cannot be propagated to primary output for observation because of the bad testability of the programs.

We also compared our approach with ATPG methods, one is from [SaSA94] and another is Gentest from AT&T, to test

0000	+ s1 + s2 => des	1000	= s1 = s2 => status
0001	- s1 - s2 => des	1001	/= s1 /= s2 => status
0010	and s1 and s2 => des	1010	> s1 > s2 => status
0011	or s1 or s2 => des	1011	< s1 < s2 => status
0100	xor s1 xor s2 => des	1100	* s1 * s2 => des
0101	not not s1 => des	1101	+ * s1*s2=>R1';
0110	shl s1 <<(s2) => des		R0'+R1'=>R0'
0111	shr s1 >>(s2) => des		R0'=>des
1110	MOR s1 => des ex: MOR 2 15 3	1111	MOV BUS => des ex: MOR 2 15 3
1110	MOR s1 => Output Port ex: MOR 2 0 15	1110	MOR ALU => Output Port ex: MOR 15 2 15 MOR MUL => Output Port ex: MOR 15 3 15

Figure 12: Instruction Set of DSP Core

Program	Structure Coverage	Testability		Fault Coverage
		Controllability (average/min)	Observability (average/min)	
Arfilter	72.12%	0.97404348 0.95682120	0.55724556 0.0	72.93%
Bandpass	72.12%	0.94723958 0.0	0.93888458 0.0	77.72%
Biquad	68.27%	0.96204983 0.91321445	0.67403408 0.0	74.49%
Bpfilter	70.19%	0.97434439 0.81129462	0.55653020 0.0	75.57%
Convolution	65.38%	0.98735248 0.98724031	0.93742645 0.75375003	65.34%
FFT	60.58%	0.95853872 0.94886202	0.39941249 0.0	74.22%
HAL	62.50%	0.95507135 0.91424042	0.51253748 0.0	73.67%
Wave	75.96%	0.96574649 0.96546084	0.48326498 0.0	74.79%
ATPG (Gentest)	N/A	N/A	N/A	89.70%
ATPG (CRIS94)	N/A	N/A	N/A	86.55%
Test Program	97.12%	1.0/1.0	1.0/1.0	94.15%

Table 3: Comparison of Experimental Results

the core. Our approach generates better results. It is because ATPG treats all the inputs equally, no matter they are data inputs or instruction inputs. It is very difficult for ATPG programs to find out the right combinations in instruction inputs and data inputs that can propagate certain faults. In this case, there are 16 bit data input and 16 bit instruction input, the search space is 2^{32} which is huge. Obviously, ATPG neglects the information provided by the instruction set treating them the same as the data input. In addition, there are some faults which need a sequence of instructions to set up certain bits. These faults are regarded as sequential faults which are undetectable by ATPG. But in our testing scheme, the self-test program makes use of the behavioral level information of the chip, so the searching space is pruned according to this behavioral level information. Also instructions has certain inherent sequence which can set up certain bits so that some sequential faults can be detected.

6.4 In Depth Study

We also carried out more experiments to show that a self-test program is necessary. In the above experiments, we just used one application program during the test and compared its results with that of self-test program. We noticed that several normal application programs maybe concatenate together so that a lengthy program can be obtained. It will have better structural coverage. Table 4 shows the results. Still, they are quite far behind from that of the self-test program approach.

The program called **comb1** is a concatenation of the eight programs listed in Table 3 in alphabetic sequence. The program **comb2** is in reverse order of comb1, comb3 is in a random order of these application programs.

Program	Structure Coverage	Testability		Fault Coverage
		Controllability	Observability	
Comb1	79.81%	0.97 0	0.71 0	79.88%
Comb2	79.81%	0.97 0	0.71 0	79.87%
Comb3	79.81%	0.97 0	0.71 0	79.87%

Table 4: Results of In Depth Study

7. Conclusion and Future Work

In this paper, we proposed an approach to systematically assemble a self-test program for random test of embedded DSP cores. This random testing scheme is very general and easy to use in different design situations. In this testing scheme, as the end users of the core do not have to know the internal structure of the core, it can protect the intellectual properties of the core designer. The key issue of the testing scheme is the self-testing program. Our approach, based on two metrics, structural coverage and testability metrics, can efficiently and effectively move the random patterns to different parts of the core and propagate them to the output port for observation. The experimental results show that our approach can reach very high fault coverage which the other approaches and methodology can not reach.

References

[BiMa95] U. Bieker, P. Marwedel, "Retargetable Self-Test Program Generation Using Constraint Logic Programming", 32nd Design Automation Conference (DAC-95) , pp. 605-611, 1995.

[BrAb84] D. Brahme, J. Abraham, "Functional Testing of Microprocessors", IEEE Trans. on Computers, Vol. C-33, No.6, 1984.

[ChMc76] A. Chiang, R. McCaskill, "Two New Approaches Simplify Testing of Microprocessors", Electronics, Vol. 49, No.2, pp.100-105, Jan. 1976.

[ShenSu88] L. Shen and S.Y.H. Su, "A Functional Testing Method for Microprocessors," IEEE trans. Computers, C-37, No. 10, pp. 1288-1293, Oct. 1988.

[Tal89] E. Talkan, et al, "Microprocessors Functional Testing Techniques," IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, CAD-8, No. 3, pp. 316-318, 1989.

[vGoor92] A.J. van de Goor and Th. J. Verhallen, "Functional Testing of Current Microprocessors," Intern. Test Conference (ITC-92), pp. 684-695, Sept. 1992.

[Comp94] COMPASS Design Automation, Inc., "VHDL for the ASIC Synthesizer User Guide", COMPASS Design Automation, Inc., Aug. 1994.

[HPCN92] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test", European Design Automation Conference (Euro-DAC-92), Sept. 1992.

[Krug91] G. Krüger, "A Tool for Hierarchical Test generation", IEEE Trans. Computer-aided Design of Integrated Circuits and Systems, Vol. 10, April, 1991.

[LePa94] J. Lee, J. Patel, "Architectural Level Test Generation for Microprocessors", IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, Vol. 13, No.10, pp.1288-1300, Oct. 1994.

[LePa92] J. Lee, J. Patel, "An instruction sequence assembling methodology for testing microprocessors," Internat. Test Conference (ITC-92), pp. 49-58, Sept. 1992.

[PaCa95] C. Papachristou, J. Carletta, "Test Synthesis in the Behavioral Domain," Int'l Test Conf. (ITC-95), Oct. 1995.

[PattHenn96] D. Patterson and J. Hennessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Francisco, 1996.

[Niko92] M. Nicolaidis, "Transparent BIST for RAMs," Intern. Test Conf. (ITC-92), pp. 598-607, Sept. 1992.

[RoyAbr90] K. Roy and J. Abraham, "High Level Test Generation Using Data Flow Descriptions," European Design Automation Conf. (EDAC-90), pp. 480-484, Feb. 1990.

[SaSA94] D. Saab, Y. Saab, J. Abraham, "Iterative [Simulation-Based Genetics+Deterministic Techniques]=Co mplete ATPG", ICCAD'94, pp. 40-43.

[ThAb80] S. Thatte, J. Abraham, "Test Generation for Microprocessors", IEEE Trans. on Computers, Vol. C-29, No.6, 1980.