Designing Self Test Programs for Embedded DSP Cores

Hani Rizk, Chris Papachristou and Francis Wolff
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract. This paper describes a self test program design technique for embedded DSP cores. The method requires minimal knowledge of the core's internals and minimal insertion of external LFSR hardware, without scan insertions. The test program consists of a small set of instructions which operate iteratively on pseudorandom data generated by the LFSRs to fully test the DSP core components. The method uses instruction-based test metrics and a program template as a blueprint to generate the test program. The self test scheme has been successfully applied on an industrial-strength DSP core and the results compare favorably to other methods using ATPG and pseudorandom BIST.

1 Introduction

Testing of embedded DSP cores in a System-on-Chip (SoC) is very challenging. Embedded cores have limited test data bandwidth from the Automated Test Equipment (ATE) due to limited speed and number of external pins. The area and performance overhead of invasive DFT such as scan may make them prohitive in timing critical applications. Moreover, hard to test components may be embedded within the core. Finally, IP vendors desire to minimize internal knowledge that may be revealed indirectly by test sets supplied with their core.

To address these issues, we have developed a self-test methodology for DSP cores whose basis is to construct a self-test template program using instruction-level controllability and observability metrics on the core's instruction set. The self-test program primarily consists of template instructions and data using minimal additional hardware based on LSFRs. For random pattern resistant faults, non-template instructions and data are provided. Errors are propagated to the output of the core and can be validated directly or indirectly using a MISR.

The idea of utilizing a processor's instruction set to construct self-test programs has been around for a while [1]. Since then, more fully developed approaches have been developed. A functional-based self-test and validation approach is discussed in [2]. Another functional test generation method based on an evolution paradigm is in [3]. The BIST scheme [4] randomizes instruction opcodes, under some restrictions, as well as data fields to produce self-test code. However, no specific methodology for constructing the self-test program is provided and there is difficulty targeting components with poor controllability and observability. The method described in [5] uses deterministic data, however, the test program produced may be too large [6] attempts to decrease the size of self-test programs by adding test instructions to the core, however, it requires modifications to the core's controller, while their results show only 20% test size reduction. The methodology in [7] develops tests for the individual core components; it also may lead to very large test programs. Although, the option of pseudorandom patterns is given, no instruction-level testability metrics are used incurring accessibility problems for some core components. Instruction self-testing was also used in [8], however, they do not provide systematic testability metrics for instructions. A method using ATPG constraint extraction is in [9].

In our work, we use a metrics-based approach to construct the self-test program. We employ instruction-based testability metrics to determine whether an instruction sequence exercises components along the core's datapath sufficiently. We believe that this approach is simple and can be easily implemented once the testability metrics are calculated. Moreover, using ATPG and fault simulation, we can enhance our results regarding test time and random resistant faults, in like manner as in [7].

This paper is organized as follows. Section 2 discusses our self-test methodology, including instruction-based test metrics and introduces a program template as a blueprint for test program generation. Section 3 discusses in detail how we apply our test methodology on an industrial DSP core. The results compare favorably to the use of ATPG and pseudorandom BIST.

2 Self Test Methodology

The self test program consists of a sequence of instructions which, when executed as a loop several times, with different random numbers loaded into the DSP core as data, provides the core with good coverage. The objective is to sufficiently exercise each datapath component and to propagate any errors that result in the component outputs to an observable output of the core. We use instruction-level controllability and observability metrics [10, 11] to determine the component testability.

2.1 The Controllability Metric

The controllability metric is based on randomness of the inputs to a component. The idea of pseudorandom fault testing is that if enough random test vectors are provided to a component then most of the possible faults will be observable at the component's output. The randomness metric is derived from entropy as defined in information theory which measures the average amount of uncertainty regarding the value of a variable. The following equation shows the entropy of a n-bit variable x that can take on values 0 to $2^n - 1$ and p_i is the probability that x equals i.

$$H(X) = H(p_0, p_1, ..., p_{2^n-1}) = -\sum_{0}^{2^n-1} p_i \, log_2 \, p_i$$

The controllability metric, $\mathcal{C}(X)$ quantifies the quality of pseudorandom patterns as they propagate through the embedded components. The entropy of each component's output is measured as a result of applying a pseudorandom uniform distribution to the input of the core. Simulation can be done at the

1530-1591/04 $20.00 (c) 2004 IEEE

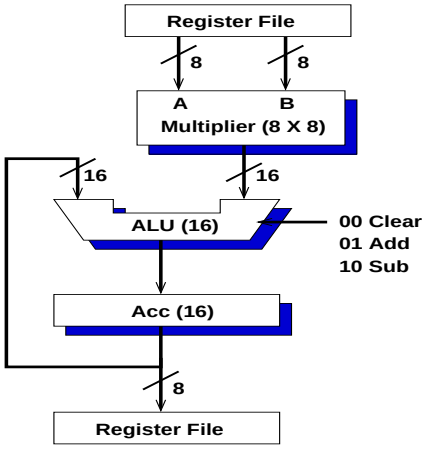

Figure 1. DSP Datapath

Opcode	Mult	Add	Sub	Clear	Acc
Add 0	0.79/0.99	0.79/0.99			0.79/0.99
Add R	0.73/0.99	0.85/0.99			0.85/0.99
Sub 0	0.70/0.99		0.83/0.99		0.83/0.99
Sub R	0.67/0.99		0.86/0.99		0.86/0.99
Mac 0	0.71/0.99	0.86/0.99			0.86/0.99
Mac R	0.73/0.99	0.89/0.99			0.89/0.99
Clr 0	0.64/0.00			0.76/0.99	0.76/0.99
Clr R	0.64/0.00			0.76/0.99	0.76/0.99

Table 1. Controllability/Observability Metrics Table

behavioral level in order to get the probabilities used for these calculations.

$$\mathcal{C}(X) = \frac{H(X)}{H(\frac{1}{2^n}, ..., \frac{1}{2^n})} = \frac{H(X)}{n}$$

$\mathcal{C}(X)$ can range from non-controllability, 0.0, to full controllability, 1.0. Values decreasing towards zero represent more coverage effort required by the random number generator for the component.

Table 1 shows the controllability metric of each component for a simple DSP datapath example in Figure 1. Each column represents each component's mode or behavior due to control bits on the datapath. The component "ALU" has three modes: clear, add, or subtract. The rows of the table represent the different instructions that can be used to exercise the components in the circuit. When running the behavioral simulations to calculate the metrics, we assume that the values which are fed back or stored in datapath registers are either known or random. The metrics for each of the instructions, Table 1, are calculated twice, once assuming that the value previously stored is the constant zero (i.e. denoted by a "0") and the other time with a random value, denoted by "R" in column one. This point is explained later.

2.2 The Observability Metric

The observability metric, $\mathcal{O}(X)$, quantifies the ability of propagating an erroneous value within the datapath to the core's observable output. This can be calculated by the total number of errors observed at the core's output, δ_{core}, divided by the total number of simulations with random errors applied at the output of the component within the datapath, $\delta(X)$.

For an n-bit signal there are $2^n - 1$ possible erroneous values. For components with large output signals it would take too long to perform all $2^n - 1$ possible faulty simulations for each good simulation, so a good heuristic resulted in $2 \times n$. For example, a good circuit was simulated 2000 times. If a component had an output that was n-bits wide, for each of the good simulations, $2 \times n$ faulty simulations were run with a random value replacing the component's actual output value. Thus if 2000 good simulations were run, then $2000 \times 2 \times n$ simulations were run with a random erroneous signal at the n-bit output signal of component X, then the approximate observability of component X would be calculated using the following formula:

$$\mathcal{O}(X) = \frac{\delta_{core}}{\delta(X)} \approx \frac{\delta_{core}}{2000 \times 2 \times n}$$

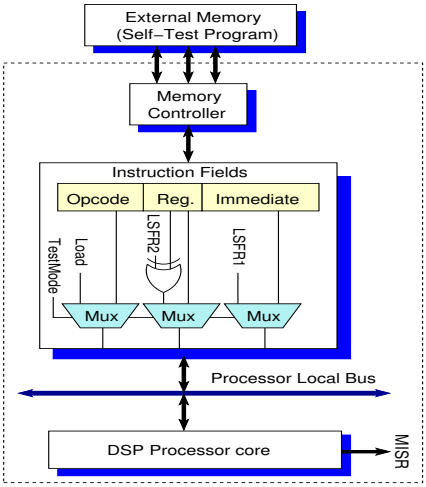

Figure 2. Test Program Template Architecture

Table 1 also shows the observability metric of each component of a simple DSP datapath, Figure 1, ranging from 0.0 to 1.0. An observability of 1.0 means that every time that an erroneous value was used at the components output it resulted in an error at the core's output. An observability of 0.0 means that errors in the components output had no effect on the value output by the core.

An instruction *covers* a component if the metrics it produces on that component are greater or equal to threshold values for the controllability, $\mathcal{C}_\theta$, and observability, $\mathcal{O}_\theta$, metrics. From experience, good initial choices are $\mathcal{C}_\theta = 0.70$ and $\mathcal{O}_\theta = 0.50$.

2.3 Test Program Template Architecture

As mentioned earlier, the goal of the self-test program is to sufficiently exercise each of the DSP components and to propagate any errors that result in the outputs of these components to an observable output of the core. By using a combination of randomly generated data and specific data, this methodology will generate test vectors that will provide good test coverage of the core. This means providing both a) topological or structural coverage of the core's components and b) good quality controllability and observability metrics. This will hopefully mean that the self test program also provides a high fault coverage for the core. Figure 2 shows the self-test template architecture which modifies the instruction stream between the memory and core. Instructions from memory are treated as templates and various instruction fields are instantiated with pseudorandom data during testing. Instantiated instructions are then passed on to the cache or processor core. Although, special test instructions can be added, they are only visible to the the template architecture and not the processor core itself. The output of the core can then be fed into a response analyzer, e.g. MISR.

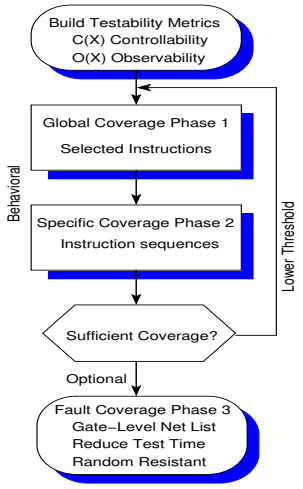

Figure 3. Self-Test Template Generation flow

Several pseudorandom number generators (i.e. LFSRs) are used by the architecture. Load pseudorandom data instructions are created from unused opcodes from the DSP core's architecture which are trapped beforehand by the template architecture. The instructions immediate field is then filled with the LFSR1 data and then the opcode is transformed as a normal load instruction.

When constructing our test program we do not take into consideration the different registers that make up the register file. This is because a register file may contain many registers, and it would take a long test program to cover all of them. We still want to exercise each of the registers that make up the register file, so we add hardware that masks the register addresses used by the test program, exercising a different group of registers each iteration through the test program. Exclusive-ORing the LFSR2 to the instructions register fields is one method which can be used to change the register field instance of the self-test program. This allows reuse of the same program to exercise other registers during datapath coverage by using test loops.

2.4 Test Program Template Generation

Figure 3 shows the program generation flow for choosing the instructions that make up the self-test program. The first step is to construct a "Metrics Table" that provides the controllability and observability metrics produced by each instruction for each of the components on our datapath at the behavioral level. This is followed by two self-test program refinement phases, 1 and 2. If sufficient coverage is not reached, the thresholds can be lowered a limited amount of times. Finally, a third, non-behavioral, phase adds instructions with specific patterns inserted to attack the random pattern resistant cases.

• Global Coverage Phase 1: We begin picking the instruction that covers the most columns in the metrics table, then we delete those columns. We continue with the next instruction until we delete all columns in the table. For example, in Table 1, multiply and add instruction, "Mac R", covers three columns. This instruction is chosen to be part of the self-test program.

Some instructions, such as Load and Out are automatically used as wrappers in our test program. By "wrapper" we mean an instruction sequence that is either used before an instruction to put the core in a certain state, before that instruction is executed, or used after the instruction to ensure that any faults detected by the instruction are propagated to an observable output. If any components are covered by these wrappers we also remove their columns from the metrics table.

• Specific Coverage Phase 2: We target individual components that were not sufficiently exercised in Phase 1. This means that there was no single instruction that was able to provide good enough randomness and observability for the component. First we try to find a sequence of instructions that provides a good randomness for the component. After that, we need to find a sequence of instructions that propagates any errors to an observable output (register). We can use our knowledge of the core's behavior in order to accomplish this. At the end we will have a test program that can be run as a loop multiple times generating test vectors for the core.

• Optional Fault Coverage Phase: We do not require gate-level knowledge of the core, but if it is available, two further enhancements can be made to improve fault coverage and reduce the test time. The first enhancement is to use fault simulation to find which components take longer to get good coverage. We add instructions exercising that component to the test program so that they are executed multiple times per iteration. The second enhancement is to use ATPG to target random resistant faults. ATPG is only used on the specific component that contains these faults so that the ATPG tool will have a better chance of generating good vectors. We add instructions to propagate the given patterns to the component and then to propagate the fault to observable output. These instructions are not within the program loop and are only executed once. While these techniques will increase the length of the test program, nonetheless, they should reduce the number of times we need to loop through the program to obtain a sufficient fault coverage, decreasing the total test time.

2.5 Experimental Environment

We start with a VHDL description of our core and use the Synopsys Design Compiler to get a gate-level Verilog netlist. We then construct a self-test program using our strategy. Our scheme assumes that we only have a high level description of the core and its instruction set. A Perl script was used to generate test patterns that would be generated if the test program was looped through a given number of times. It also inserts the pseudorandom data patterns which were generated by simulating the LFSR. These test patterns and the core's gate level netlist are fed into the Synopsys fault simulator, Tetramax, to find the fault coverage of our testing scheme. The Perl script also outputs a VHDL testbench which is used to simulate the execution of our test program on the core using the Synopsys VHDL Simulator. This is done for verification purposes to ensure that the the model used for fault simulation behaves correctly. We use other Perl scripts to compute the testability metrics. The scripts modify behavioral VHDL code, simulate the modified code using Synopsys' VHDL Simulator, and use the results to compute randomness and observability.

3 Industry-based DSP core

This section discusses a step-by-step implementation of our test methodology on an industry-based pipelined DSP core. First we describe the core in some detail, then go over the steps of our testing scheme and finally we provide results and comparisons.

3.1 The DSP Datapath

Our DSP core, shown in Fig. 6, is designed as a four stage pipelined RISC-based Load/Store processor. The core uses a 17 bit instruction as input and has an 8 bit output. We use a fixed 17-bit instruction format, shown in Fig. 4, consisting of a 5-bit opcode field, and a destination register field. Depending on the opcode, there is a data field (bits 11-4), or two source register fields. The core contains a register file made of sixteen 8-bit registers and a functional unit that includes a complex MAC datapath. Since a pipeline architecture was used, read after write hazards had to be considered. To take care of the hazards a temporary (forwarding) register, was used. The

16–12 Opcode	11–8 Reg A	7–4 Reg B	3–0 Dest Reg	Format 1
16–12 Opcode	Value to be Loaded		3–0 Dest Reg	Format 2
16–12 Opcode	11–8 XXXX	7–4 Source Reg	3–0 XXXX	Format 3
16–12 00010	11–8 XXXX	7–4 Source Reg	3–0 Dest Reg	Format 4

Figure 4. Instruction set

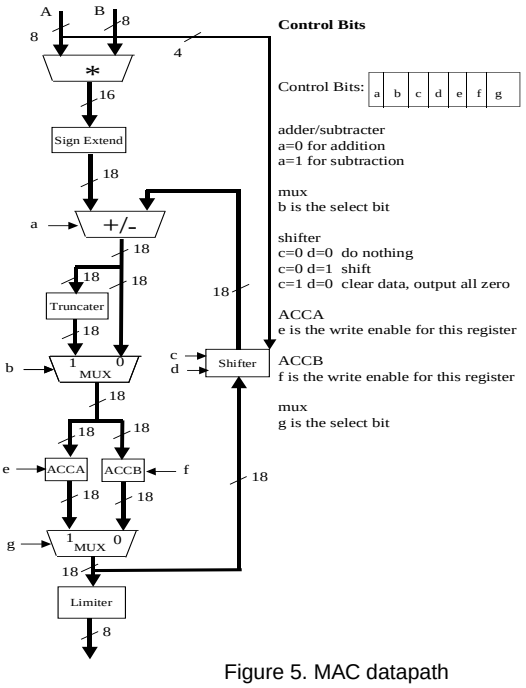

Figure 5. MAC datapath

third stage of the core contains a buffer whose output is used by instructions such as "Ld" and "Out". For most other instructions the results of the MAC datapath (Figure 5) are used. The input to the MAC consists of seven control bits and two 8-bit operands which come from the specified registers in the register file; their values are fetched in the second stage. The control bits are the output of a decoder in the second stage. The MAC datapath in Fig. 5 outputs an 8 bit result to the *M*acReg which is connected to the multiplexer MUX7. The inputs and outputs of the MAC use 8-bit fixed point integers formatted with four bits to the left and four to the right of the decimal point. The MAC contains an 8-bit multiplier that outputs a sign extended product to 18 bits. The result of the adder/subtracter is then written into one of two 18-bit accumulators, $AccA$ and $AccB$. The contents of the accumulators are fed into a shifter whose output is fed back into the adder/subtracter. The MAC also contains a truncater, which truncates the data to the right of the decimal point. The arithmetic shifter is controlled by bits c and d;the direction and amount of shift is determined by the four bit signed integer from the A input. The limiter clips the maximum positive and negative values of the 18-bit input integer producing an 8-bit output integer.

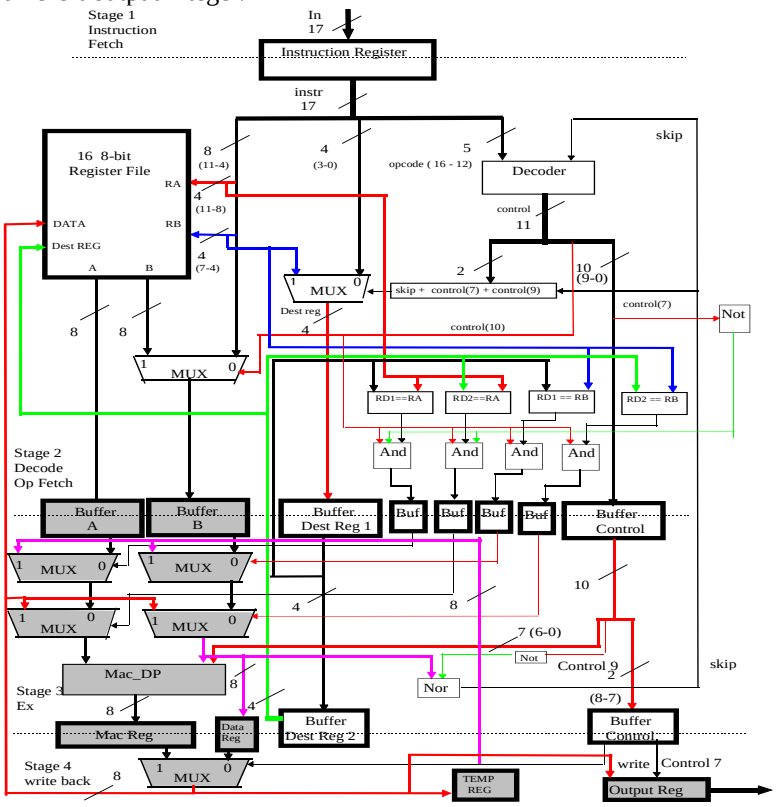

Figure 6. Four-stage pipeline core design

3.2 DSP Testability Metrics

Table 2 shows a subset of DSP instructions (i.e. rows) for controllability and observability metrics. A subset of the columns are also shown which represent each of the shaded components of Fig. 6. For example, the shifter has two control bits and therefore requires four columns. The first row beneath the column names contains the number of faults of the component.

• **DSP controllability issues**. To calculate the controllability of a component's inputs we get their probability distribution through the simulation of the core's behavior while executing an instruction sequence. We found that we needed more than 2000 iterations to get good results for the randomness of components with large inputs. To avoid long simulations, we wrote a perl script which operates on the core's behavioral description and produces a C++ program that would simulate the circuit behavior and calculate the controllability metrics much faster. Nonetheless, some components have 2 input ports each 18-bit wide, or a total of 36 input lines, thus taking too long to simulate 2^{36} iterations. However, for most of these 2 input port components we were able to assure that their 2 input port signals are statistically independent. Recall that if two events X and Y are independent then their entropies are related as $H(X,Y) = H(X) + H(Y)$. Thus, for n-bit input ports we have: $\mathcal{C}(X,Y) = (1/2n)(\mathcal{C}(X) + \mathcal{C}(Y))$. This assumption allowed us to find the controllability of most components in reasonable time. However, sometimes input independence can not be assured, e.g. in case of reconvergent fanouts as shown in Fig. 5 for the adder output merging into the MUX_b. Generally, in cases like this, we need to perform longer simulations to compute the controllability metrics.

	Multiplier :0	Shifter 00 :1	Shifter 01 :2	Shifter 10 :3	Shifter 11 :4	Add / Sub 0 :5	Add / Sub 1 :6	Acc A 0 :7
	2162	2028				700		404
load	0.99,0.00	0.18,0.00				0.35,0.00		0.71,0.00
	0.99,0.00	0.99,0.00				0.85,0.00		0.99,0.00
mpy	0.99,0.71 X			0.18,0.70		0.35,0.69		0.71,0.00
Mpyshift								
Mpyshiftmac								
Out	1,0	.18,0				.36,0		0.71,0.00
Outr	1,0	1,0	.			.83,0		1,0
Mac+	0.99,0.71 X	0.18,0.69				0.35,0.69		0.71,0.00
Mac+R	0.99,0.14	0.99,0.51 X				0.85,0.51 X		0.99,0.00
Mac-	0.99,0.70 X	0.18,0.69					0.35,0.69	0.71,0.00
Mac-R	0.99,0.14 X	0.99,0.51 X					0.85,0.51 X	0.99,0.00
Mact+	0.99,0.70 X	0.18,0.69				0.35,0.69		0.35,0.00
Mact+R	0.99,0.14	0.99,0.51 X				0.85,0.51 X		0.55,0.00
Mact-	0.99,0.70 X	0.18,0.68					0.35,0.69	0.35,0.00
Mact-R	0.99,0.14	0.99,0.51 X					0.85,0.52 X	0.55,0.00
MpyR	0.99,0.71 X			0.99,0.70 X		0.35,0.70		0.71,0.00
mpyt	0.99,0.70 X			0.18,0.69		0.35,0.69		0.35,0.00
MpytR	0.99,0.69 X			0.99,0.68 X		0.35,0.68		0.35,0.00
shift	0.99,0.71 X		0.18,0.70			0.35,0.70		0.71,0.00

Table 2. Controllability/Observability Metrics Table

• **DSP observability issues**. To describe the sequential behavior of an instruction sequence or a feedback loop, we unroll the loop. For example, the loop around the MAC datapath in Fig. 5 demonstrates the unrolling of the MAC datapath. Thus the MUX component becomes unrolled as two separate components: $MUX_g^{Limiter}$ before the limiter, and $MUX_g^{Shifter}$ before the shifter. When calculating the metrics, we treat these different instances of the components as if they were actually different components. When deciding if an instruction sufficiently covers the MUX, we would check the testability metrics provided for both $MUX_g^{Limiter}$ and $MUX_g^{Shifter}$. If either of these metrics values are good enough, then we say that the component has been sufficiently covered. Note, we calculate the metrics for each instruction twice, once assuming that the accumulators contain a zero value and another time assuming that they contain a random value denoted by an "R". To get the core into the assumed state we have to precede the instruction in the test program with a group of other instructions. The first four instructions in the table are used to set the core to an expected state. The Load instruction is used to load the pseudorandomly generated data into the registers which are used as operands for the other instructions.

We chose $\mathcal{C}_\theta$, 0.70 to be our controllability threshold and $\mathcal{O}_\theta$, 0.50 to be our observability threshold. If both of these thresholds are met then we mark, "X", the column as being covered.

3.3 Self Test Phases

• **Phase 1**. In this phase we choose from the instructions listed in the metrics table to construct the self-test program. Since each test sequence begins with the Load instruction several columns (not shown in Table 2) are covered and removed from the table. Our covering algorithm now selects the next instruction that covers the most columns, here being MpyR, covering eleven. We continue choosing instructions to be added to our self-test program in this manner until none of the remaining columns are marked as being sufficiently covered by any of the instructions. Table 3 is derived at the end of Phase 1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
																			18,23					22,19		23,24,27		19,25,26						
load																X							X(M7)	X(M7)	X(temp)	X(temp)					X	X
mpy	X								A		B						X	X	X(M7)	X(M7)		X	X(M7)	X(M7)	X(temp)	X(temp)	X(temp)	X(temp)			X		X
Mpyshift												B	A																				
Mpyshiftmac									A		B							X															
Out																	X						X(M7)	X(M7)	X(temp)	X(temp)						X	X
MactA-R		X					X						A																				
MpyRB	X			X							B	B						X	X(M7)	X(M7)		X	X(M7)	X(M7)	X(temp)	X(temp)	X(temp)	X(temp)			X		X
ShiftRB			X			X					B	B						X		X(M7)							X(temp)	X(temp)			X		

Table 3. Instructions chosen at the end of Phase 1

• **Phase 2** . In this phase we target the individual components that were not sufficiently covered in Phase 1, specifically, columns 4, 7, 9, 13, 14, 20, 28, 29, not all shown in Table 2. This means there was no single instruction that was able to provide randomness and observability to these component above the thresholds. Our approach is based on several observations.

a) Improve metrics by using a sequence of instructions. First we use instructions that provide sufficient randomness for the component's input and then we try to propagate the component's results to an observable output.

b) Eliminate columns whose control bits are not set by any instruction. Thus for column 4, representing the shifter with control bits set to "11", there are no instructions that set its control bits to "11", meaning we can discard this column.

We continue this process in this manner choosing instruction sequences to cover the rest of columns, thus producing the final test program. Fig. 7 shows the assembled binary, instructions and component coverage of a portion of the test program.

```
Bit Code          Symbolic Code     Columns covered and comments

00111XXXXXXXX0001   ld rnd,R1
00101XXXXXXXX0000   ld rnd,R0         // 15,31,32,22
01100000000010010   MPYB R0,R1,R2     // 0,10,17,19,21,25,26,30 randomize accb
00001000000100000   out R2            //
00000011100000011   ld "01110000",R3
10000001111110100   SHIFTB R3,R4      // ShiftB7
00000000000010101   ld "00000001",R5
00101XXXXXXXX0110   ld rnd,R6
11100011001010111   MACB+ R6,R5,R7
00001000001100000   out R6             //29,16 Phase2 gets temp_reg
00111XXXXXXXX1000   ld rnd,R8
00101XXXXXXXX1001   ld rnd,R9          // end randomization seq
01100100010011010   MPYB R8,R9,R10     // 3,18,23,24,27 MPYBR
00001000010100000   out R10
01100000000010010   MPYB R0,R1,R2      // randomize accb
10000001111110100   SHIFTB R3,R4      // ShiftB7
11100011001010111   MACB+ R6,R5,R7     // End randomization seq
10000100011111010   SHIFTB R8,R15,R10  // 2,5 ShiftBR
10001001111110100   SHIFTA R3,R4        // 7 Phase2 Observe ACCA
00001000001000000   out R4
00001000010100000   out R10
01101000000010010   MPYA R0,R1,R2      // 8 randomize acca
00001000000100000   out R2
10001001111110100   SHIFTA R3,R4       //13 ShiftA7
00001000001000000   out R4             // Phase2
11101011001010111   MACA+ R6,R5,R7     // end randomization seq
10000001111110100   SHIFTB R3,R4       // 9 Phase2 Observe ACCB
00001000001000000   out R4
11011100010011011   MACTA- R8,R9,R11   // 1,6,12  MACTA-R
00001000010110000   out R11
00000000100001100   ld "00010000",R12
01100000011001101   MPYB R0,R12,R13    // 28  Phase2
00001000011010000   out R13            // 20
00001000000000000   out R0             // 16 Output random value
```

Figure 7. Self Test Program after Phase 2

• **Fault Coverage Produced by Our Scheme**. We use Synopsys's ATPG and fault simulation tool, Tetramax, to determine the fault coverage. The test program shown in Figure 7 is used as input for a Perl script which simulates the behavior of the LFSRs and generates the test vectors that would be fed to the core. We choose to loop through the self-test program 6000 times. The program contains 34 instructions so looping through it 6000 times would generate $34 \times 6000 = 204,000$ test vectors. If we assume that the core has a 500Mhz clock, then the total test time would be 0.408ms. The results show that our testing scheme provides 98.14% fault coverage and a test coverage of 98.33% on the core.

3.4 Test Phase 3: Enhancements

We will discuss three enhancements that can be implemented after phases 1 and 2.

• **Control bit constraints**. First, for each component on the datapath that has control bits we use Tetramax to get the possible fault coverage on that component with its control bits constrained. For example, the shifter has two control bits. When we constrain the bits so that they cannot have a value of "11" we find that only 3 faults cannot be detected and we get 99.86% fault coverage on the shifter. If we constrain the control bits so that they cannot be "00," 59 faults go undetected and the fault coverage is 97.21%. If the control bits are not allowed to be

"01," 1829 faults go undetected and we only get 13.4% fault coverage for the shifter. If the control bits are not allowed to be "10," only 1 fault goes undetected and the shifter has a fault coverage of 99.95%. Finally, if we constrain the control bits so that they can only be "00" and "01", 5 faults go undetected and we get a 99.76% fault coverage for the shifter. From the above results we see that it is not really important to exercise the shifter when it's control bits have values of "11" and "10", so we can discard the columns representing the shifter with these control bit values from the metrics table (columns 3 and 4).

• **Increasing component execution frequency**. It may take longer to test some components than others. Through fault simulation we are able to find out how many test vectors it takes for sufficient fault coverage to be achieved on the different components. Speeding up the time it takes for the larger components to achieve good fault coverage may have a noticable effect on the required test length for the entire core. Executing the instructions that exercise the shifter and adder more than once per iteration in our program, causes the fault coverage to rise more rapidly, allowing us to shorten our test time. In fact, we only need to use the first 27,346 test vectors generated by the enhanced program in order to detect more faults than were detected when we used the original program to generate 204,000 test vectors.

• **Random Resistant Patterns**. Some components may contain random resistant faults, which still may not be detected after looping through the test program a reasonable amount of times. If the tester wants to increase fault coverage he can find which components contain undetected faults through fault simulation. ATPG is used specifically on that component to find which test patterns are needed to detect these faults and instructions are added that will place these specific patterns on the component's inputs and propagate any errors on the component's output to an observable output. These instructions are also stored in memory, but unlike the other instructions that made up the self-test program these instructions are only executed once.

One disadvantage of this method is that it can take up a lot of memory. It took 21 lines to test the adder with just one pattern given by ATPG. If we wanted to detect every single fault we would have to go through this process for the other ATPG patterns on the adder and then for other patterns on each of the components that did not achieve 100% test coverage after looping through our original self-test program. It may also be very hard, to figure out how to use the instruction set to get some of the ATPG patterns to the target component and to propagate any errors to an observable output [9].

3.5 Discussion

For comparison purposes, we generated test patterns with the Tetramax ATPG tool. The test only gave us an 8.51% fault coverage. Because our core is a relatively complex circuit, it is just too hard for the ATPG tool to determine good sequential test patterns. ATPG tools work on the gate level, so they do not have a high level description. For example, the simple fact that we need to use the output instruction in order to output the value of a register is very useful when trying to make faults observable, but a gate-level ATPG tool does not have the benefit of this information.

We also compared our scheme to a regular pseudorandom BIST. For this method a 17 bit wide LFSR generates the 17 bit test vectors. We decided to generate all 131,071 test vectors that could be generated by the LFSR. The reason is that the LFSR does not take into account the core's present state or the core's behavior while generating test vectors.

The experimental results for our testing scheme provided good fault coverage. Running the original self-test program for 6000 iterations we got a fault coverage of 98.14%. Running the enhanced self-test program for 6000 iterations we got a fault coverage of 98.42%. As mentioned before the real benefit of the enhancements was the shortening the total test time, while still achieving high fault coverage.

4 Conclusion

We have developed a method for testing embedded DSP cores based on self test programs. In order to demonstrate our method we used an industry-based design of a DSP pipelined core. Our experimental results showed that our approach can achieve very high fault coverage within a reasonable test time.

Our test scheme requires no modifications to the embedded core itself which is important in high performance DSP designs. Moreover, only the core's instruction set and higher-level behavioral information is needed to construct the self-test program. We use controllability and observability metrics to evaluate how well the different instructions in the instruction set exercise the core components. These metrics do not require gate-level knowledge of the core, but if it is available our proposed enhancements to the test program improve fault coverage and reduce test time.

References

[1] J. Hayes, E.J. McCluskey, "Testability Considerations in Microprocessor-Based Design," *IEEE Computer*, March 1980.

[2] J. Shen, J. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," *I*ntern. Test Conf. (ITC-1998), Oct. 1998.

[3] F. Corno, G. Cumani, M. S. Reorda, G. Squillero, "Fully automatic test program generation for microprocessor cores," *Design Automation and Test in Europe (DATE-2003)*, March 2003.

[4] Batcher, K.; Papachristou, C. "Instruction Randomization Self Test for Processor Cores," *IEEE VLSI Test Symposium*, 1999.

[5] Lai, W-C., Krstic A., Cheng K-T., "Test Program Synthesis for Path Delay Faults in Microprocessor Cores," *Internat. Test Conference (ITC-2000)*, pp. 1080-1089, 2000.

[6] Lai, W., Cheng, K-T. Cheng, "Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip," *Design Automation Conf. (DAC-2001)*, 2001.

[7] Chen L, Dey S., "deFUSE: a Deterministic Functional Self-Test Methodology for Processors," *IEEE, VLSI Test Symposium*, pp.255-262, 2000.

[8] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, Y. Zorian, "Low-cost software-based self-testing of RISC processor cores," *Design Automation and Test in Europe (DATE-2003)*, March 2003.

[9] Chen L., Ravi S., Raghunathan A., Dey S., "A scalable software-based self-test methodology for programmable processors," *Design Automation Conf. (DAC-2003)*, June 2003.

[10] Ravikumar, C. P., Saund G. S., Agrawal N., "A STAFAN-Like Functional Testability Measure for Register-Level Circuits," *IEEE Fourth Asian Test Symposium*, pp. 192-198, 1995.

[11] Corno F., Prinetto P., Sonza M., "Testability Analysis and ATPG on Behavioral RT-Level VHDL," *Internat. Test Conference (ITC-97)*, pp. 753-759, 1997.