Hierarchical Test Generation and Design for Testability of ASPPs and ASIPs *

Indradeep Ghosh, Anand Raghunathan and Niraj K. Jha
Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

Abstract

In this paper, we present design for testability (DFT) and hierarchical test generation techniques for facilitating the testing of application-specific programmable processors (ASPPs) and application-specific instruction processors (ASIPs). The method utilizes the register transfer level (RTL) circuit description of an ASPP or ASIP and tries to generate a set of test microcode patterns which can be written into the instruction read-only memory (ROM) of the processor. These lines of microcode dictate a new control/data flow in the circuit and can be used to test modules which are not easily testable. The new control/data flow is used to justify precomputed test sets of a module from the system primary inputs to the module inputs and propagate output responses from the module output to the system primary outputs. If the derived test microcode cannot test all untested modules in the circuit, then test multiplexers are added to the data path to test these modules and thus testability of all modules is guaranteed. This scheme has the advantages of low area and delay overheads (average of 3.1% and 0.4% respectively), high fault coverage (>99.6% for all cases) and at-speed testing. Test generation times are about three orders of magnitude smaller than an efficient gate-level sequential test generator. Only one external test pin is required for the DFT method.

1. Introduction

An ASPP is a programmable architecture which is designed to execute a number of different behaviors [1]. Since application-specific integrated circuits (ASICs) are specifically designed to execute a single behavior, it is difficult to make changes in the later phase of the design cycle. However, a programmable data path can be more easily adapted to a new behavior, so that redesign work is greatly simplified. In ASPPs, a microprogrammed controller is generally used and microcode programs are written into the control memory. There are usually several of these programs in a single control memory of an ASPP where each program caters to a single behavior. External mode inputs are used to load the starting address of a particular program into the program counter and thus switch the ASPP to execute different behaviors. An ASIP is also a programmable data path which lies in between a digital signal processor (DSP) and an ASPP in terms of programmability [2]. Here the data path is more generic and has register files and arithmetic-logic units (ALUs) like a traditional processor. To improve performance, its instruction set is more limited than a DSP and the connectivity in the data path is restricted. So it is tuned to a smaller set of applications which it can perform faster. However, it is not as restricted as an ASPP in terms of programability and is slower than an ASPP if programmed to execute the specific behaviors which an ASPP is tuned to do. Some work has been done on exploiting the properties of ASIPs for testing purposes [3]. However, it neither explains the method of generating test programs nor gives fault coverage results.

In this work we present a method to generate test microcode based on an analysis of the structural RTL implementation of an ASPP or ASIP. This test microcode, when written into the instruction memory of the ASPP or ASIP, dictates a new control/data flow in the circuit. Thus, we try to exploit the inherent programmability of these circuits during testing. This new control/data flow is used to justify and propagate precomputed module-level test sets. The method of using precomputed test sets for acyclic RTL circuits was first presented in [4]. However, we can tackle cyclic RTL circuits as well. In the few cases where this method is unsuccessful in testing the data path completely, we add some test multiplexers to the data path to solve the problem [5]. We use a particular test architecture to test the controller separately. The only assumption we make is that the microcode is encoded in the horizontal format. However, the method can be easily modified to work for designs in which the microcode is in the vertical format. Experimental results on a number of examples show that this method results in a very high fault coverage (>99.6%) and low area (3.1%) and delay (0.4%) overheads. The area overheads would be even lower if the instruction ROM was assumed to be erasable. The test generation time by our method is up to 1500 times faster than an efficient gate-level sequential test generator and the circuits are at-speed testable.

This procedure can be extended to be applicable to any general-purpose programmable processor if it is possible to alter its microcode ROM. If that is not possible, one will have to work with a predefined instruction set, as opposed to microcode directly, whereby the general controllability and observability of the data path is restricted to what the instruction set can provide.

In Sections 2 through 4, we illustrate how our method works through an example ASPP and ASIP. We present the DFT procedure in Section 5 and experimental results in Section 6.

2. Example 1: An ASPP

Figure 1 shows the RTL implementation of an ASPP which we have named ASPP4. This was obtained by simultaneously synthesizing the control/data flow graphs (CDFGs) of *Paulin* and *Tseng* into a single RTL circuit using *Genesis(area)*, a high-level synthesis tool that targets area optimization [6]. Depending on which microcode program in the controller is used, the circuit can emulate the behavior of either *Paulin* or *Tseng*.

First, using the existing microcode programs, we extract the CDFGs of *Paulin* and *Tseng* [7], assuming that the CDFGs of the different behaviors that will execute on the data path are not available to us. We utilize these CDFGs to test as many modules in the data path as possible through symbolic justification of a precomputed test set at the inputs of any operation in the CDFG, which is mapped to the module under test, from the system primary inputs, and by propagating the output response of the operation to a CDFG output. The symbolic justification and propagation paths form a *test environment* for the operation [6]. Since it is symbolic, it is independent of exactly which vectors are present in the precomputed test set.

The CDFG and binding information extracted for *Tseng* from ASPP4 is shown in Figure 2. Constants *c1* and *c2* are the initial

*Acknowledgments: This work was supported by NSF under Grant No. MIP-9319269.

DAC 97, Anaheim, California

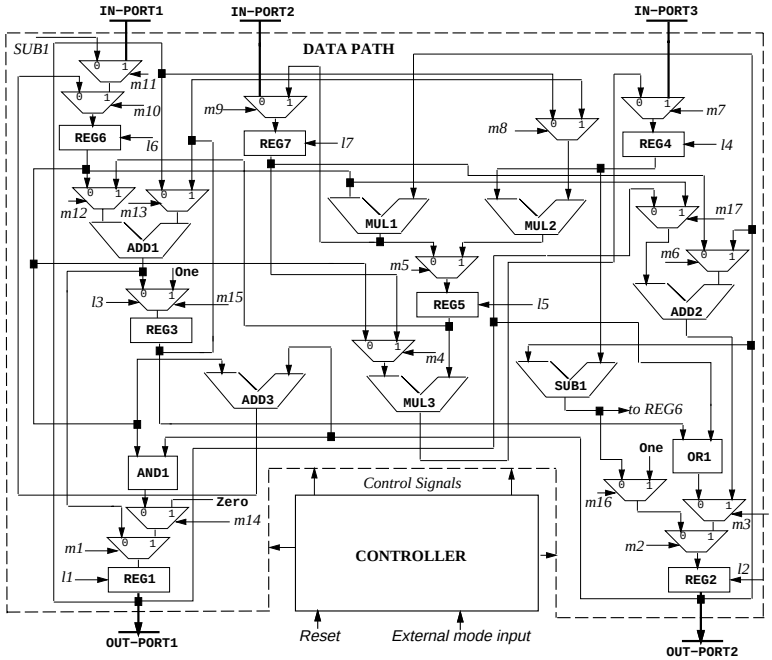


Figure 1: The RTL circuit of ASPP4

values of the loop outputs with their values in brackets. The test environment of operation *+4* is used to test adder ADD2 in the RTL circuit. To test an adder, suppose we need vector x (y) at its left (right) input. We can apply vector y at input *i3* and vector $1 - x$ at input *i2*. However, operation *+1* is also mapped to adder ADD2. Since a fault in a module can affect all operations mapped to it, we need to verify the value of *v1* by observing it at primary output register REG2. The output of *+4*, *v6*, is observable since it is also mapped to REG2. Hence, to test operation *+4*, and thus module ADD2, we require four clock cycles for every vector in the precomputed test set of the adder. Also, the above method is general enough to handle chaining, multicycling, and structural pipelining.

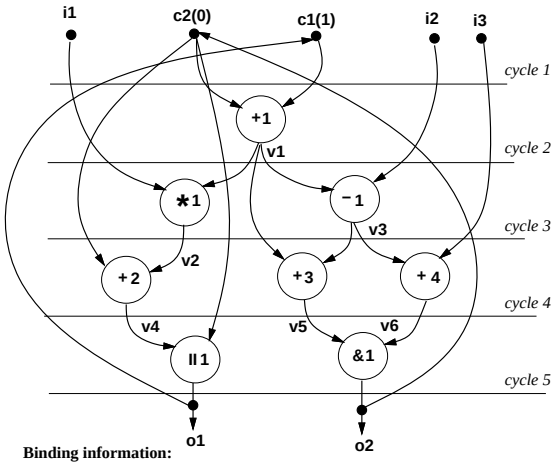
Let n be the number of lines in the main loop body of a microcode program, *e.g.* for *Tseng*, $n = 5$. In general, we extract the behavior of the circuit for at most $2n$ cycles (this limit is flexible but $2n$ was sufficient for our purposes), although, for simplicity, in Figure 2 we show the extracted behavior for only 5 cycles. After extracting all existing behaviors in the circuit, and trying to derive a test environment for at least one operation for every module, if some untested modules still remain, we resort to DFT. In Figure 1, modules ADD3 and OR1 remain untested after the above efforts. To test them, we add some test microcode which dictates a special data/control flow that allows us to find a test environment for these modules in the test mode.

Consider ADD3. Let vector x (y) be required at the left (right) input of ADD3. Consider the following operation sequence:
Cycle 1: Load REG7 with x from IN-PORT2 and REG1 with 0.
Cycle 2: Configure input multiplexer trees of ADD2 so that the output of REG1 (REG7) flows into its left (right) input; load REG2 with the output of ADD2; load REG6 with y from IN-PORT1.
Cycle 3: ADD3 inputs now have vector (x,y). Load REG6 from the output of ADD3; load REG7 with vector 0 from IN-PORT2.
Cycle 4: Configure input multiplexer trees of ADD2 so that the output of REG6 (REG7) flows into its left (right) input; load REG2 with the output of ADD2.

The sequence of operations, which tests module ADD3 with test vector (x,y), can be achieved by adding four extra lines of test microcode to the control memory. This sequence of microcode can be formally obtained by first converting the RTL circuit into a *structural connectivity graph* (SCG). In the SCG, all multiplexers are abstracted out as edges since we can get the required flow through a multiplexer tree using test microcode. The input/output ports, constants, registers and modules are all represented in the graph. The modules have two different sets of edges coming into them comprising of their input ports. Figure 3 shows the SCG obtained from the RTL circuit of ASPP4. The register nodes are replicated to keep the diagram simple and node i represents REGi. The *primary input registers* (registers directly connected to primary input ports) are encircled at the module inputs. The *primary output registers* (registers directly connected to the primary output ports) are encircled with shaded circles at the module outputs. The *constant registers* (registers to which constants may be loaded) are encased in a square. The remaining registers are just black circles. The modules which remain untested without the extra test microcode are shaded.

From the SCG, we compute controllability and observability numbers, called *C-numbers* and *O-numbers*, for each register. The C-number of a register gives an approximate idea as to how many cycles are required to control its value to a symbolic vector from a primary input port, given complete freedom to choose the control signals of the RTL circuit. For example, the C-number of all primary input registers is 1 as they can be loaded from an input port in one cycle. Similarly, the O-number of a register gives an approximate idea as to how many cycles are required to observe the symbolic value of a register at a primary output port, given the above freedom. Hence, the O-number of all primary output registers is 0. We then apply a few simple rules (see Section 5) based on a distance metric to get the C- and O-numbers of the other registers. These are shown at the top of Figure 3.

We next perform RTL test generation on the untested modules. First, we need to define a few terms. The general controllability



Binding information:

Inputs : i1, i2, i3
Outputs : o1, o2
REG1 : c2, o2
REG2 : c1, o1, v1, v6
REG3 : v4
REG4 : i2
REG5 : v2
REG6 : v3, v5, i1
REG7 : i3

ADD1 : +2
ADD2 : +1, +4
ADD3 : +3
*MUL1 : *1*
MUL2 : −
MUL3 : −
SUB1 : −1
OR1 : ||1
AND1 : &1

Figure 2: CDFG for *Tseng* extracted from ASPP4

C_g of an SCG node is the ability to control it to any arbitrary value from the system primary inputs. The observability O_v of an SCG node is the ability to observe any arbitrary value at this node at a system primary output. Similarly, we can define C_1 (controllability to 1), C_0 (controllability to 0), and C_{a1} (controllability to the all-1s vector). The verifiability V of an SCG node is the ability to verify its value by either controlling or observing it. The controllability, observability and verifiability are all Boolean parameters, *i.e.* they only take the values of 1 or 0, depending on whether the variable has the corresponding ability or not [6]. For RTL justification and propagation, we add another field to these parameters which des-

ignates the cycle when the particular property of a node is desired. Hence, $C_g(2)$ of an SCG node means that we need to control that node to an arbitrary value in cycle 2.

We next illustrate how we can use RTL justification and propagation to test module ADD3. At first, we set the requirement of $C_g(\alpha)$ at the left and right inputs of ADD3 and $O_v(\alpha+1)$ at its output. To achieve $C_g(\alpha)$ at the left input, it is sufficient to get $C_g(\alpha)$ at any of the registers feeding this input. Usually, we choose the register with the lowest C-number (because it is easiest to control) and break ties arbitrarily. Here the only choice is REG6, which is a primary input register and hence controllable in one cycle. Only register REG2 exists at the right input. The requirement of $C_g(\alpha)$ at REG2 is propagated up through a module for which REG2 is an output. We can choose either ADD2, SUB1, or OR1. Among these, we first look for a module which has registers with C-number 1 at both its inputs (these registers must be different), or a module which has a register with C-number 1 at one of its inputs and a different constant register at the other input (special constants are required for each type of module, *e.g.* an arbitrary constant for an adder or subtracter; 1 for a multiplier, etc.). Based on this criterion, we can choose either ADD2 or SUB1. Let us choose ADD2 arbitrarily. Then the $C_g(\alpha)$ requirement of REG2 propagates to become the requirements: $C_0(\alpha-1)$ of REG1 and $C_g(\alpha-1)$ of REG7 (this is one of the several choices; in general, we may need to backtrack and consider other choices). $C_g(\alpha-1)$ of REG7 is trivial as it is a primary input register. $C_0(\alpha-1)$ of REG1 is obtainable as it is fed by constant 0. We next need to propagate the $O_v(\alpha+1)$ requirement from REG6 at the output of module ADD3. In general, the register with the lowest O-number is selected. For propagation from REG6, we choose a module, if possible, whose other input has either a required constant register or a register with C-number 1 and whose output has a register with O-number 0. When not possible, of all the modules with REG6 as an input, we add up the minimum C-number at the other input and the minimum O-number at the output and choose the module which has the lowest sum. We avoid propagation through the module under test to prevent the faulty module from interfering with the propagation. For propagation from REG6, we can choose either AND1, ADD1 or ADD2. Let us choose ADD2 arbitrarily. Hence, the $O_v(\alpha+1)$ requirement of REG6 gets propagated as $O_v(\alpha+2)$ of REG2 and $C_0(\alpha+1)$ of REG7 (one of many choices). $O_v(\alpha+2)$ of REG2 and $C_0(\alpha+1)$ of REG7 are trivially satisfied as REG2 and REG7, respectively, are primary output and input registers. We next assign a value of 1 to the lowest symbolic cycle number and compute the rest of the cycles from that. Here we assign the value of 1 to $(\alpha-1)$. Thus, we can obtain a test environment for ADD3.

Next we illustrate the test environment generation for module OR1. For this, we need $C_g(\alpha)$ of REG3, $C_g(\alpha)$ of REG1, and $O_v(\alpha+1)$ of REG2. $O_v(\alpha+1)$ of REG2 is trivially satisfied (primary output register). $C_g(\alpha)$ of REG3 gets propagated as $C_g(\alpha-1)$ of REG6 and $C_0(\alpha-1)$ of REG1 through ADD1. $C_g(\alpha)$ of REG1 gets propagated as $C_g(\alpha-1)$ of REG6 and $C_0(\alpha-1)$ of REG1 through ADD1. This leads to a conflict in values desired in REG6 in cycle $(\alpha-1)$ (since this register cannot be controlled to two arbitrary values at the same time), as well as a conflict in the operation desired out of module ADD1 in cycle $(\alpha-1)$. Hence, we relax the requirement of $C_g(\alpha)$ of REG1 to $C_g(\alpha+1)$ of REG1. This means we also need $C_g(\alpha+1)$ of REG3 since the two parts of the test vector of ADD3 have to be applied simultaneously. However, once

C–number		Register
1	–	4, 6, 7
2	–	1, 2, 3, 5

O–number		Register
0	–	1, 2
1	–	3, 4, 5, 6, 7

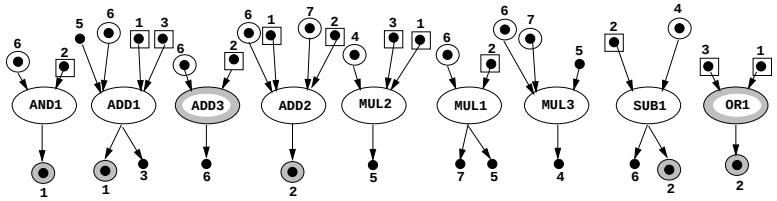


Figure 3: The SCG for ASPP4

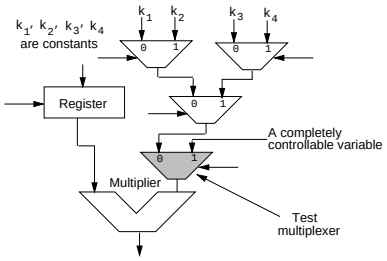


Figure 4: Case showing the necessity of a test multiplexer

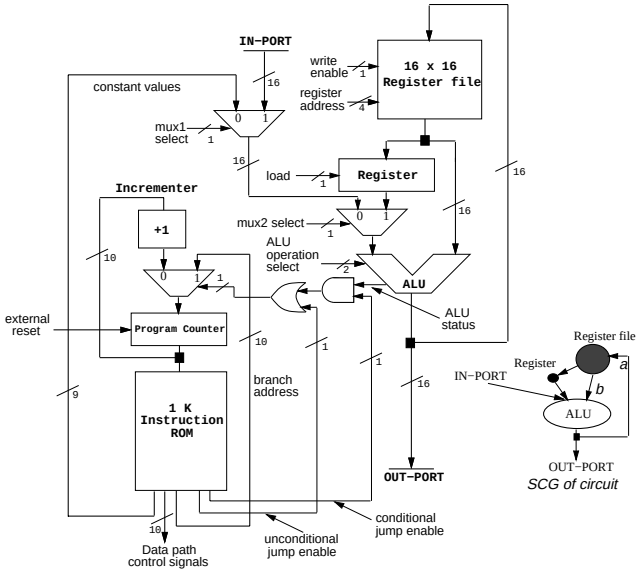


Figure 5: The RTL circuit of SimpleCPU and its SCG

$C_g(\alpha)$ of REG3 is satisfied, the value may be frozen in that register for one more cycle by just making its load signal 0. Hence, we now have: $C_g(\alpha+1)$ of REG3 $\rightarrow$ $C_g(\alpha)$ of REG3 $\rightarrow$ $C_g(\alpha-1)$ of REG6 and $C_0(\alpha-1)$ of REG1 (through module ADD1), and $C_g(\alpha+1)$ of REG1 $\rightarrow$ $C_g(\alpha)$ of REG6 and $C_0(\alpha)$ of REG1 (through module ADD1). The observability requirement of ADD3 translates to $O_v(\alpha+2)$ of REG2 which is still trivially satisfied. Now we just need to set $(\alpha-1)$ to 1 and calculate all the cycle numbers. Thus, we have obtained a correct test environment for OR1.

At each step, we need to specify the multiplexer select signals according to the path required for the flow of the test data. There can be conflicts during the justification and propagation process (*e.g.* we might require conflicting values in a register at the same cycle). In that case, we need to backtrack and explore other options. We may also relax the requirements if a test environment is not found initially as explained above. Thus, $C_g(r)$ requirement may be relaxed to $C_g(r+1)$. Finally, we have the option of adding a test multiplexer to solve the problem, as discussed later. In our experiments, we did not target registers and multiplexers specifically for test microcode generation, as the system-level test set derived from module test sets was generally sufficient to detect faults in registers and multiplexers. This is because registers and multiplexers only need four vectors for complete stuck-at fault detection. However, we can also target them for test environment generation exactly as above.

For each cycle of a test environment we need one line of test microcode. Hence, if we directly concatenate the test microcodes of ADD3 and OR1, we need eight lines. However, we can do better. Suppose we need vector $x(y)$ at the left(right) input of ADD3 and for OR1 correspondingly $r(s)$. Consider the following sequence:

Cycle 1: Load REG6 with vector r; load REG1 with constant 0; load REG7 with vector y.

Cycle 2: Configure input multiplexers of ADD2 (ADD1) to add the content of REG1 with that of REG7 (REG6); load REG3 (REG2) from ADD1 (ADD2); load REG6 with either vector *x* or vector *s* (it is enough to load either vector because the same test program can be used to sequentially test each untested module; when testing ADD3 (OR1) we load *x*(*s*)).
Cycle 3: Configure the input multiplexer of ADD1 to add contents of REG1 and REG6; load REG1 (REG6) from ADD1 (ADD3).
Cycle 4: Configure input multiplexers of ADD1 to add contents of REG1 and REG6; load REG2 (REG1) from OR1 (ADD1) (though REG1 may not contain 0 in cycle 3, the fault effect from REG6 will always be propagated through ADD1).

This sequence of four operations can be used to test both OR1 and ADD3. Hence, only four lines of test microcode are enough. This leads us to the issue of test microcode compaction. The compaction technique we use here is similar to those used for test compaction in sequential test generation. Typically, the test microcode obtained for an untested module has many don't-cares for unspecified control signals. Using this initial microcode as a constraint, test compaction techniques [8]-[10] are used to derive test environments of other untested modules. The compacted program is encased in a loop to allow multiple runs with different test vectors by adding an unconditional branch from the end to the beginning of the program. The test architecture given ahead lets us easily exit the loop when testing of the module is done. In the normal mode, the test program is totally bypassed, as explained later.

Sometimes, it may so happen that no amount of test microcode can test a particular module. Consider the situation in Figure 4 where one port of a multiplier is fed by multiple constants only. In such cases, we use the procedure from [5] to add a test multiplexer at the input of the multiplier as shown.

3. Example 2: An ASIP

Figure 5 shows the RTL circuit of an ASIP named SimpleCPU, taken from [3] and modified slightly. We can derive test environments for this circuit also. Consider the register file. From the SCG of the circuit, the objective is $C_g(\alpha)$ of node *a* and $O_v(\alpha + 1)$ of node *b*. $C_g(\alpha)$ of node *a* is transformed to $C_g(\alpha)$ of the IN-PORT and $C_0(\alpha)$ of node *b* (assuming we are using the adder of the ALU). $C_g(\alpha)$ at the IN-PORT is trivially satisfied. $C_0(\alpha)$ of node *b* is transformed to $C_0(\alpha - 1)$ of node *a* which is in turn transformed to $C_0(\alpha - 1)$ at the IN-PORT (assuming we are using the multiplier of the ALU). This is trivially satisfiable. $O_v(\alpha + 1)$ of node *b* is transformed to $C_0(\alpha + 1)$ of the IN-PORT through the adder of the ALU and $O_v(\alpha + 1)$ of node *a* (again trivially satisfiable). Setting $(\alpha - 1)$ to 1 completes the test environment for the register file.

The above method is repeated for all the modules of the data path and encasing of each test microcode program in a loop is done as before. We also need to add some more microcode for testing the controller and the glue logic between the data path and controller. This is explained next.

4. Testing the controller

Figure 6 shows the test architecture used to test the controller of SimpleCPU where all the added DFT hardware is shaded grey. Application of this test architecture to other ASPPs/ASIPs simply involves small changes in the width of the *test configuration register* (TCR), the output multiplexer, *etc*. We assume one extra *Test pin* is available. TCR stores bits needed to control various signals in the test mode [5]. When it is reset, the normal functionality of the circuit is restored. In the control ROM, assuming it is non-erasable, both the test and normal programs co-exist, as shown. We place an unconditional jump statement at address 0 to the beginning of the normal program. Hence, resetting the program counter (PC) starts the normal program. PC's content is frozen when the *Test* signal is asserted to load TCR. The three bits of TCR are connected to the lower-order three bits of the IN-PORT. The ALU status signal from the data path is made directly observable at the primary output by setting the TCR bits appropriately. This is done while testing the not-equal-to comparator in the ALU.

We reserve the last two addresses of the ROM for testing the multiplexer feeding the PC. The test microcode written in these locations is shown in the figure. First, we set the branch inhibit signal and reset the PC. Then we observe the contents of the ROM exhaustively at OUT-PORT by stepping through all the addresses. At each step, we need to freeze the PC in order to observe the second half of the ROM output at OUT-PORT. This tests the incrementer and the PC exhaustively. For testing the multiplexer feeding the PC, we feed the following test set, which if fed to each bit-slice of the multiplexer (assuming an AND-OR two-level implementation), detects all single stuck-at faults in it: {(1,0,0), (0,1,1), (0,1,0), (1,0,1)}. Test vector (1,0,0) (corresponding to the 0-input, 1-input, select) is already applied to each bit-slice of the 10-bit multiplexer when the PC is incremented from 1022 to 1023. On reaching address 1023, we reset the branch inhibit signal and let the PC jump back to 1023. This enables each bit-slice of the multiplexer to get (0,1,1). Next, we again set the branch inhibit signal to let PC's content become 0. This enables each bit-slice of the multiplexer to get (0,1,0). Incrementing the PC by 1 next lets it point to the test program and reset the branch inhibit signal. This lets the test program take control and test the data path. At the end of the test program we add one more jump instruction to address 1022. When the PC is incremented from 1022 to 1023, each bit-slice of the multiplexer gets (1,0,1).

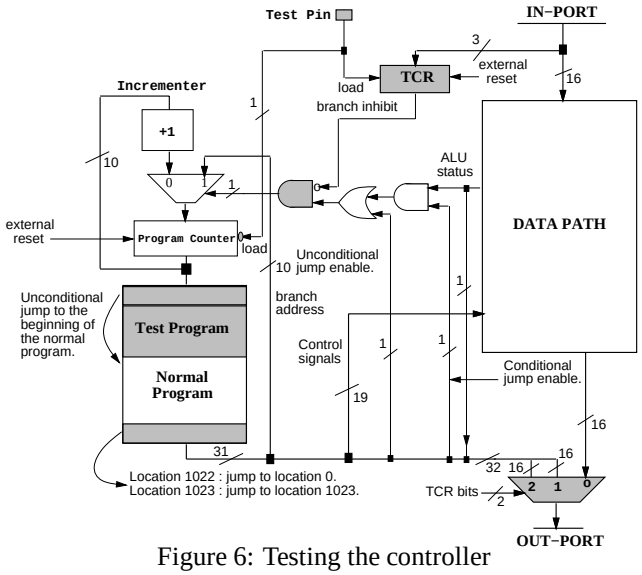


Figure 6: Testing the controller

The glue logic between the data path and controller need four vectors to test. These translate into four different actions - executing an unconditional jump, executing a conditional jump, not executing a conditional jump, and testing if a jump takes place when the ALU status is 1 but both the conditional jump and the unconditional jump are disabled. These microinstructions are included in the test program. Fault propagation through the glue logic results in branches being taken when they are not supposed to be and *vice versa*. The microinstructions located in the faulty address and the fault-free address are typically different. Even if these two microinstructions are the same, subsequent increments in the PC definitely results in the microinstructions becoming different in the faulty and fault-free cases. Thus, the observability of these fault effects is trivial. The extra AND gate (shaded) also gets tested in the above process. The output multiplexer and TCR are not targeted explicitly for testing. However, through fault simulation at the logic level, we found that they are easily and completely tested with the first few vectors of the system-level test set.

5. The DFT procedure

In this section, the hierarchical testability analysis and DFT procedures are formalized. At first, all the C- and O-numbers are computed. The C-numbers are computed first. The C-number of a register is an indication of the register's distance from a system primary input. The procedure starts with the primary input registers whose C-numbers are, by definition, 1. C-numbers of

all other registers are initially made ∞. It then successively obtains the C-numbers of the other registers by using procedure *Update_C_numbers*. This procedure has special cases for propagating C-numbers from the inputs of a particular type of module to its outputs. The different cases are required because to propagate C-numbers across different types of modules, different requirements must be met. For example, for propagating the C-number from the left input of a multiplier to its output, the required constant at the right input is assumed to be 1, whereas for an adder any constant suffices. It is possible that at the end of this procedure many registers with C-number of ∞ remain. In such cases, we identify modules which prevent propagation of C-numbers. Such a module has a finite C-number at one of its input registers and a C-number of ∞ at some of its output registers. We add a test multiplexer with an appropriate constant input to the other input port of that module. Then we compute the C-numbers, and iterate, if necessary, until each module has at least one register with a finite C-number at each of its input ports.

Next, we compute the O-numbers. The O-number of a register is an indication of its distance from the primary output ports. The main procedure starts by assigning an O-number of 0 to all primary output registers and ∞ to all other registers. It then successively obtains the O-numbers of all registers using the procedures *Update_right_input_O_numbers* and *Update_left_input_O_numbers*. If the calculated O-number of a register in an iteration is smaller than its existing O-number then this new value becomes the O-number of the register. The main procedure terminates when the O-numbers of all registers stabilize. The above two procedures try to propagate O-numbers from a register at the output of a module to the registers at its corresponding input ports using some specific rules. For example, if a necessary constant value is available at the left input of a module (*e.g.* 1 for a multiplier) then the smallest O-number among all registers connected to the output of the module is incremented by 1 and the value assigned as the O-number of all the registers connected to the right input of the module. The rationale behind this method is that if the output of the module is observable at a primary output in α cycles, then we can use the constant at its left input to observe any value at its right input in $\alpha + 1$ cycles. We use these C- and O-numbers to guide our search for generating test environments.

Figure 7 shows the main procedure for RTL testability analysis (RTA). The procedure takes as input the SCG of the RTL circuit, the untested component list, the C- and O-numbers previously computed, and a library of precomputed module test sets. It then generates the test environment for each untested module, obtains the corresponding test microcode, and derives the system-level test set to be applied while executing the test microcode. The initial objectives for justification and propagation are set up using the lowest C- and O-numbers at the inputs and output of the untested module in procedure *Setup_initial_objectives* which returns a CVO (controllability, observability, and verifiability) list. This has CVO objectives for each cycle that need to be met to get the test environment for the untested module. *RTA_justify_&_propagate* routine does the justification and propagation of the initial CVO objectives until they are satisfied. Also, there are some new CVO transformations which are valid here. One of them is *relaxation* which was explained earlier. This is valid because we can freeze the value of a register for an indefinite number of cycles once its objective is achieved. The C- and O-numbers are used to speed up the search.

```
Procedure RTA(SCG R, Untested Component list, C- and O-numbers, test_set library)
{
  for each module M in untested_component list {
    Initialize C, V, O lists ;
    Initialize microcode values ;
    Setup_initial_objectives(M, R, CVO lists, Microcode_vals, C- and O-numbers) ;
    Test_environment
        = RTA_justify_&_propagate(R, CVO lists, Microcode_vals, C- and O-numbers) ;
    if (Test_environment ≠ ϕ)
      Generate_test_vectors(Test_environment, test set for M) ;
    else
      Append M to Untestable_list;
    return (Untestable_list);
  }
```

Figure 7: Main procedure for RTL testability analysis

Table 1: Circuit size and DFT hardware statistics

Circuit	bw	# lits	# ffs	# test muxs	# lines test µcode	CPU (sec)
ASPP1	16	56740	214	1	7	51
ASPP2	16	46251	229	2	8	65
ASPP3	16	50435	310	0	9	7
ASPP4	16	20145	117	0	8	8
ASPP5	16	29603	343	0	13	12
SimpleCPU	16	23806	282	0	185	2
TMS32010	16	25902	314	0	202	3

At each step, depending on the CVO transformations employed, some control signal values get fixed. This information is kept in a microcode table which is updated in parallel with the CVO lists. There might be conflicts during the search due to conflicting value requirements at a register in a particular cycle or in the microcode requirement also. For example, we might require two different configurations of the select signals of a multiplexer tree at a particular cycle. In such cases, we backtrack and undo the CVO transformations as well as the microcode values resulting from previous decisions. If *RTA_justify_&_propagate* fails to generate the test environment of a particular module, the module is appended to an untestable list and later test multiplexers are added to the data path to test these modules [5].

We have assumed that the microcode is encoded in a horizontal format here. However, the above procedure can be extended to the vertical format. This means that because of dependencies among certain control signals, once the value of one is fixed, the values of certain other signals also get fixed. This can be taken care of in procedure *RTA_justify_&_propagate*, by fixing dependent signal values when we fix the microcode values due to a chosen CVO transformation. For further justification and propagation, these signals are taken into account to check for conflicts. The procedures for computing the C- and O-numbers will need to be changed slightly. In those cases, once we use an edge in the SCG, some edges will disappear from the SCG due to implications from the signals corresponding to the selected edge. Further details of the above procedures are available in [11].

6. Experimental results

We present experimental results obtained by applying our testing scheme to five example ASPPs and two ASIPs. The ASPPs were generated by Genesis(area) [6], that targets area optimization. Out of these examples, ASPP4 is as shown in Section 2. ASPP1 can execute the behavior of different algorithms for discrete cosine transform (DCT) namely, *Dct_lee*, *Dct_dif*, and *Dct_wang*. ASPP2 can execute the behaviors of *Pr1* and *Pr2* - two different algorithms that perform rotation-based DCT. ASPP3 can execute the behaviors of three filters - *Elliptic* (fifth-order elliptic wave filter), *Chemical* and *Dist* (IIR filters used in the industry). ASPP5 can execute the behaviors of *Bandpass*, *Lowpass* and *Bandstop* filters. The ASIP *SimpleCPU* has been discussed in Section 3. The *TMS32010* design was taken from [12] and is a multiplexer-based implementation of the architecture from Texas Instruments. The data paths were optimized at the logic level using SIS [13]. They were then technology-mapped with the *stdcell2_2.genlib* library. The instruction ROMs were generated using the ALLIANCE synthesis system [14] and scaled down to the data path technology to get the complete area and delay figures of the circuits.

Table 1 shows the size and DFT hardware statistics of the circuits obtained. In Columns 2, 3 and 4 the bit-width, literal-count,

Table 2: DFT hardware placement overheads

Circuit	*Area*			*Delay*		
	Orig.	*Mod.*	*Ovhd.* (%)	*Orig.* (ns)	*Mod.* (ns)	*Ovhd.* (%)
ASPP1	607297	615851	1.4	119.6	119.8	0.2
ASPP2	529076	540712	2.2	122.3	122.5	0.2
ASPP3	588628	604964	2.8	118.1	118.2	0.1
ASPP4	212732	215576	1.3	116.2	116.3	0.1
ASPP5	333627	348976	4.6	118.1	118.2	0.1
SimpleCPU	253419	268475	5.9	135.1	136.4	1.0
TMS32010	284126	300605	5.8	85.2	86.4	1.4

Table 3: Testability results

Circuit	*HITEC*				*Our Method*	
	Orig. Ckt.		*Mod. Ckt.*		*Mod. Ckt.*	
	FCov. (%)	*TGen.* (sec)	*FCov.* (%)	*TGen.* (sec)	*FCov.* (%)	*TGen.* (sec)
ASPP1	20.51	188711	76.82	67581	99.89	57
ASPP2	75.51	334313	87.65	79550	99.82	52
ASPP3	15.62	163413	80.90	69807	99.69	76
ASPP4	93.45	20925	99.45	19786	99.92	24
ASPP5	53.24	120381	87.72	75836	99.61	96
SimpleCPU	77.82	42654	98.93	20354	99.67	13
TMS32010	73.56	68976	99.25	21892	99.72	16

and number of flip-flops of the circuits are given, respectively. The number of test multiplexers used in the data path of the circuit is given in Column 5. This number does not include the multiplexers used for testing the controller. Column 6 shows the number of lines of test microcode required to test the circuits. If the instruction ROM is erasable, then these lines may be erased once the circuit is tested. In that case, the overheads will be much lower. In Column 7, the CPU time required for the DFT hardware addition process is given. All CPU times are measured on a SPARCstation 20 with 256 MB of memory. From Column 5 it is clear that the test microcode is usually successful in eliminating the need of test multiplexers from the data path except for some unavoidable cases, as discussed in Section 2.

Table 2 shows the area and delay overheads for our DFT scheme. In Column 2 the original technology- mapped area of the circuits is given. This a relative figure and hence has no units. Columns 3 and 4 respectively show the area and incurred overhead after DFT hardware addition. Columns 5, 6, and 7 show the corresponding delay numbers (clock cycle time). The average area overhead was 3.1% and the average delay overhead was 0.4%. The delay overheads were negligible because the test multiplexers, if any, were all placed in non-critical paths and most of the overhead was due to the small increase in the instruction ROM delay because of an increase in its size due to the test microcode. The test microcode is substantially larger for ASIPs because we do not assume that any microcode used in the normal mode is available to us for ASIPs, unlike ASPPs, since the applications mapped to ASIPs are not *a priori* determined, whereas for ASPPs they are. Also, the address lines of the register files in the ASIPs demand controllability which can only be provided by adding more test instructions to the instruction ROM.

In Table 3, testability results for the circuits are presented. We compare the performance of our method against HITEC [15], an efficient gate-level sequential test generator. The testability results are just for the data paths of the circuits as HITEC cannot handle the testing of ROMs. However, our scheme did test the controller exhaustively. Column 2 shows the fault coverage obtained by running HITEC on the original data paths using the default controllers. For the ASIPs we placed a *fast Fourier transform* algorithm as the original program into the instruction ROM. Column 3 shows the CPU time required by HITEC to obtain the given fault coverage. In Columns 4 and 5 the corresponding results are given when HITEC is run on the circuits modified by our DFT hardware. The corresponding results obtained by running our hierarchical test generation method on the modified circuits are shown in Columns 6 and 7, respectively. Our fault coverage numbers were obtained by fault simulating our system-level test sets on the modified circuit using PROOFS [16]. It can be seen that very high fault coverage are obtainable by our method while the CPU time required is about three orders of magnitude lower than that of HITEC for the modified circuits. Our DFT modifications also aided HITEC in its test generation as is clear from the results.

7. Conclusions

In this paper, we introduced an efficient and practical DFT scheme that can be applied to programmable data paths like ASPPs and ASIPs. Our method exploits the inherent programmability of these types of circuits and produces test microcode through an RTL testability analysis. The test microcode helps test otherwise untested modules by dictating a new data/control flow which is used to justify precomputed test sets of the modules from the system primary inputs to the module inputs and propagate test responses from the module outputs to the system primary outputs. When this technique is insufficient to test some modules, some test multiplexers are added to the data path as a last resort. However, this scheme is guaranteed to test all modules in the data path. We also devised a scheme to test the controller of these circuits completely. The advantages of this technique are (i) low area overheads and negligible delay overheads, (ii) high fault coverage, (iii) about three orders of magnitude smaller test generation time compared to gate-level sequential test generation, and (iv) at-speed testability.

References

[1] W. Zhao and C.A. Papachristou, "An evolution programming approach on multiple behaviors for the design of application specific programmable processors," in *Proc. European Design and Test Conf.*, pp. 144-150, Feb. 1996.

[2] I.J. Huang and A. Despain,"Synthesis of application specific instruction sets," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 663-675, June 1995.

[3] U. Bieker and P. Marwedel, "Retargetable self-test program generation using constrained logic programming," in *Proc. Design Automation Conf.*, pp. 605-611, June 1995.

[4] B.T. Murray and J.P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 594-603, June 1990.

[5] I. Ghosh, A. Raghunathan, and N.K. Jha, "Design for hierarchical testability of RTL circuits obtained by behavioral synthesis," in *Proc. Int. Conf. Computer Design*, pp. 173-179, Oct. 1995.

[6] S. Bhatia and N.K. Jha, "Genesis: A behavioral synthesis system for hierarchical testability," in *Proc. European Design and Test Conf.*, pp. 272-276, Feb. 1994.

[7] I. Ghosh, A. Raghunathan, and N.K. Jha, "A design for testability technique for RTL circuits using control/data flow extraction," in *Proc. Int. Conf. Computer-Aided Design*, pp. 329-336, Nov. 1996.

[8] P. Goel and B.C. Rosales, "PODEM-X: An automatic test generation system for VLSI logic structures," in *Proc. Design Automation Conf.*, pp. 260-268, June 1981.

[9] I. Pomeranz, L.N. Reddy, and S.M. Reddy, "COMPACTEST: A method to generate compact test set for combinational circuits," *IEEE Trans. Computer-Aided Design*, pp. 1040-1049, July 1993.

[10] A. Raghunathan and S. T. Chakradhar, "Dynamic test sequence compaction for sequential circuits," in *Proc. Int. Conf. VLSI design*, pp. 170-173, Jan. 1996.

[11] I. Ghosh, A. Raghunathan, and N.K. Jha, "Hierarchical test generation and design for testability for ASPPs and ASIPs," *Technical Report CE-J96-001, EE Dept., Princeton University*, Oct, 1996.

[12] C. Monahan and F. Brewer, "Concurrent analysis techniques for data path timing optimization," in *Proc. Design Automation Conf.*, pp. 47-50, June 1996.

[13] E. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, pp. 328-333, Oct. 1992.

[14] A. Greiner and F. Pecheux, "ALLIANCE: A complete set of CAD tools for teaching VLSI design," *Technical Report: Laboratoire MASI/CAO-VLSI, Institut de Programmation, Universite Pierre et Marie Curie, Paris*, 1993.

[15] T.M. Niermann and J.H. Patel. "HITEC: A test generation package for sequential circuits," in *Proc. European Design Automation Conf.*, pp. 214-218, Feb. 1991.

[16] T.M. Niermann, W.T. Cheng, and J.H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," in *Proc. Design Automation Conf.*, pp. 535-540, June 1990.