Concurrent Testing in High Level Synthesis

Ravibala Singh John Knight

Dept. of Electronics, Carleton University, Ottawa, Canada

Abstract

For digital circuits synthesized from data-flow graphs, this paper presents a method to test the circuit concurrently with its normal operation. The method tests hardware elements when they are not in use in the data-flow graph. An algorithm for synthesizing the test circuit is presented that starts with the data-flow graph, generating a circuit to cycle test vectors through the idle hardware and produce a signature so as to give a built-in-self-test. By utilizing idle computational time for testing, the method reduces test-time overheads.

1.0 Concurrent testing and applicability of the test method

At the present time, many ICs are tested only by *production tests* at the factory. More recently some circuits have used *built-in self-test*. This makes production testing easier and also allows them to be tested in the field by service personnel during "power up" or even tested remotely over telephone lines. *Concurrent tests* can be run at the same time that the circuit is processing data. This allows testing of circuits that run continuously, as in telephone central offices, air-traffic control systems, or production control systems. Concurrent testing can also give a faster warning if a system develops a fault. A machine tool control, for example, may be changed to a "safe" mode if a fault is detected. Concurrent testing is also a practical way to detect soft errors, such as caused by power glitches, metastability, electromagnetic transients, or leaky dynamic RAMs.

The test method presented here is a *concurrent, built-in self-test* which can be incorporated into the data-path synthesis process. Testing of a circuit is time-shared with the data processing in some manner. Because the testing is concurrent with the data processing, the method can reduce test-time overheads and therefore offers an advantage over scan-based tests that require high test times. Further, the method allows at-speed testing of the circuit. A majority of ASICs are not tested at speed; process variants can cause a circuit to pass a low speed test but fail at-speed in the system.

The test method runs pseudorandom test vectors through the circuit. To avoid storing a large number of comparison vectors some form of *response compression* or *signature* must be used to check if an error has occurred. Common response compression schemes such as *ones counting* or *signature analysis* could be used. These and other compression methods are well covered in chapter 10 of [1]. This paper will assume *signature analysis* is used. Since the signature is generated in parallel with the data, it may give warning of a fault before it affects the data. On the other hand, the warning may come slightly later. We envision using the test on circuits where a complete test would be done with 2^{32} or fewer test vectors and a complete test would be completed every few seconds.

The test method will test mainly the data-path and not the controller. It will test that the instructions are sequenced properly and also test the connections from the controller to each controlled element. However, it will not test control bits, in say a microcode ROM, which control true data as opposed to test data. Methods for concurrently testing these parts of the controller would have to be merged with the method described here.

2.0 Previous work

Concurrent testing has been widely applied to computational units using error-checking codes, to PLAs [1] and systolic arrays [2] using methods that apply to these particular architectures. This paper presents a new and unique concurrent, or *on-line*, test method that specifically

This work was supported in part by the Canadian Government via the Centres of Excellence, MICRONET.

0-8186-5785-5/94 $03.00 © 1994 IEEE

applies to circuits synthesized from data flow graphs. The closest previous work is [3]. However, it does not relate to data-path synthesis; *off-line* testing resources are modified so that they can observe normal circuit inputs and outputs during system operation. The test technique, therefore, depends on a set of test vectors appearing as input data in the course of the circuit's normal operation.

Testability in data-path synthesis has been addressed by many, including [4] [5] [6]; they investigate the incorporation of prevalent *off-line* test methodologies such as scan-paths and BIST in the data-path synthesis process.

3.0 Testing idle operators

As a first step in data-path synthesis, the algorithm is translated into a data-flow graph. The data-flow graph shows the causality of the operations. Then, as was done in Figure 1(a) the operations (three multiplications and a subtraction) can be assigned to definite clock cycles or control-steps, and to definite hardware units (multipliers M1, M2 and subtracter S1). Also storage registers (R1, R2 and R3) can be assigned to hold data between clock cycles. A clock going to all the registers is assumed. The circuit is completely synchronous. The circuit must also have a controller, which is not shown here. The controller controls the data flow in the data-path.The data-flow graph shows explicitly what hardware is used during each clock cycle. This makes it easy to identify, on a time localized basis, when an operator or a register is available for concurrent testing.

For the above data-flow graph, the operators and registers which are idle during the control steps are shown in Figure 1(b). The operators can have pseudorandom test vectors run through them in these particular control steps without interfering with the normal flow of data. Figure 2 shows the data-flow graph (DFG) and a completed test data-flow graph (TDFG) which runs in parallel with the data processing. The TDFG must duplicate every path in the data flow graph. For example, the path from R1 into M1 connected as a squarer in step C-2 in the DFG is duplicated in steps C-1 and C-4 in the TDFG. As another example, the subtracter deposits its output into R2 during C-4 in the DFG and C-2 in the TDFG. The testing checks each operator, each register and each bus-multiplexer path. Note that the original register allocation is changed when the test path is added. One has to add new registers to carry the test results.

A pseudorandom number generator and a signature collector are put in at convenient points in the circuit. In this case they were both put in C-step 2. Data in the test data-flow graph is collected and compressed in the signature collector. The circuit is assumed to be in an infinite loop. The data-path calculation is repeated generating a new value of **D** every 4th C-step. The test program is in a much

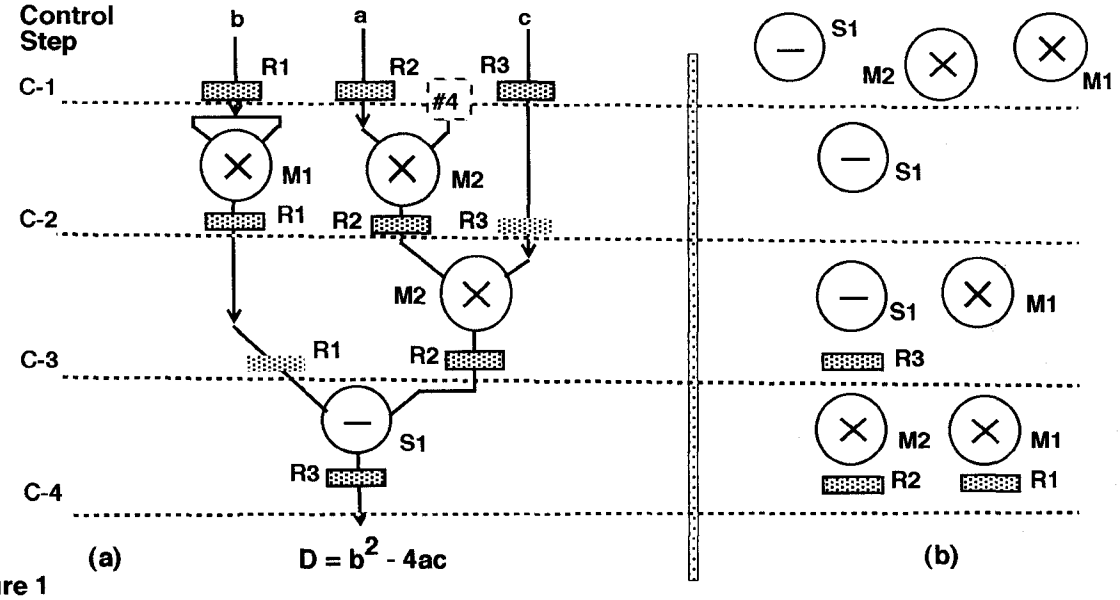


Figure 1

(a) The data-flow graph of a circuit to calculate the discriminant of a quadratic equation.

(b) The operators and registers unused in a given control-step.

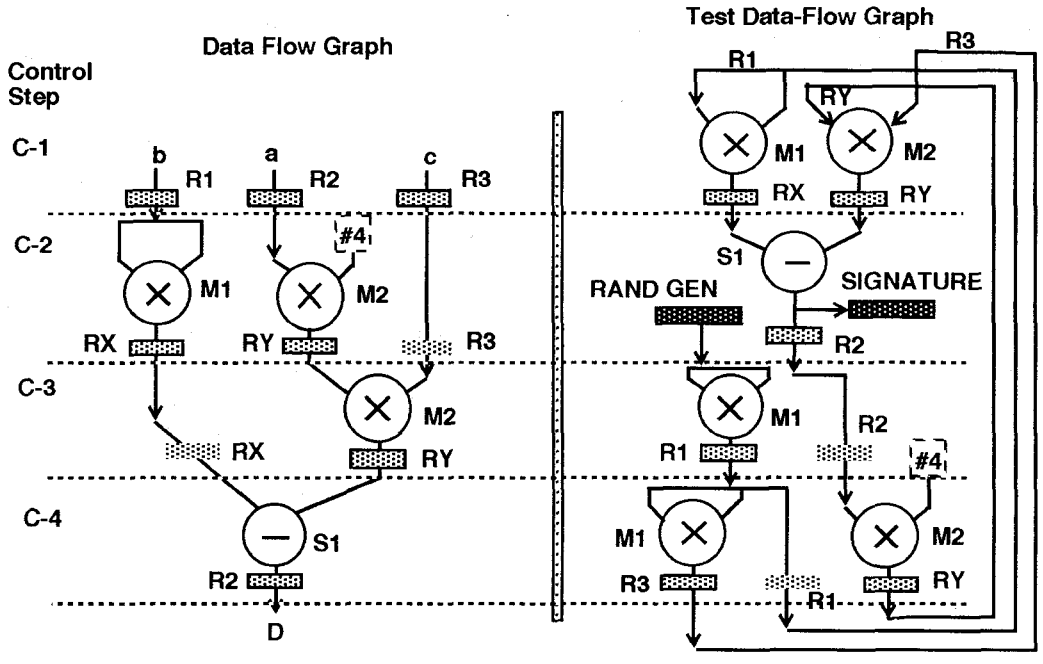


Figure 2

The data-flow graph and the test data-flow graph for the 'discriminant' example.

larger loop. For a 16 bit generator clocked every four C-steps, it would loop approximately 2^{16} times before checking the signature. Typically, the pseudorandom generator can also act as a loop counter. At the completion of its sequence it could initiate hardware action to check the signature, clear necessary registers and restart the test loop.

Concurrent testing requires some extra registers and multiplexers but requires no increase in the number of operators. For the above example DFG, concurrent testing adds two registers and two MUXs over the original circuit [7]. The *self test* also requires a pseudo-random number generator, a signature collector and a way to check that the signature is correct. It would be, however, premature to draw conclusions about the extra overhead required for the test since the example circuit is very small and hardly typical. But, it is clear that the saving is in checking the operators and the expense is in the increased number of registers and MUXs.

3.1 Test model and completeness of the test

The test, as mentioned above, can be described as a concurrent built-in self test. It is a non-exhaustive, random-pattern functional test based on high level not gate level models of the data-path modules. No assumption is made regarding the fault model of the modules. The pseudo-random test patterns exercise the function of the data-path modules which include the set of functional blocks and the set of interconnections between the functional blocks. Because the testing can be run in real time, this test method has an advantage over scan-based tests both in terms of test-times and the ability to test the circuit at-speed.

We are currently investigating the fault coverage of these tests. Our preliminary simulations of some example data paths have been encouraging. Our opinion is as follows:

- It might be unlikely that the tests will have the fault coverage expected for production testing without additions, like test-point insertion, to improve fault coverage. However, unlike production testing which is targeted only at permanent faults, concurrent testing targets both permanent as well as transient faults. Siewiorek has estimated that even in the best of environments, transient errors occur an order of magnitude more often than permanent faults [8].

- Hewlett Packard has used signature analysis for field testing for several years. They seem to have found the method useful, and the fault coverage they experience should be similar to that of this proposal [9].

- Certain test-flow connections disturb the randomness of the test vectors as is explained below.

If equal inputs are placed on an adder, the least significant output bit will always be zero. Equal inputs on a multiplier make the least-but-one significant bit zero. Multiplying a random number by a constant, like "4" in Figure 2, means all the results in the output register, RY, must be multiples of 4, and the two least significant bits will be 00. RY will subsequently test multiplier M2 with vectors ending in 00. These test vectors will still end in 00 after passing through M2 and being applied to test subtracter S1. In general, if an immediate multiplicative constant contains a factor 2^k, the last k bits in the product will be zero. Addition or subtraction of a constant, whether it contains factors of 2^k or not, does not restrict the sum or difference. Thus addition and subtraction will pass on as good test vectors as they receive. However, division will usually make high-order quotient bits zero and the output of a divider could pass on poor test vectors.

To combat this so called *entropy* [10] reduction in the test vectors, one can re-randomize the data in the TDFG by injecting new test vectors at multiple points in the graph.

4.0 Building the test data-flow graph algorithmically

Figure 3(a) shows the data-path for a more complex algorithm. It was created to illustrate how a TDFG can be generated. The operations in the DFG are scheduled into C-steps before the test-path is considered. The registers in the data-path have initially been put in so that each result is stored in a separate register. These registers will later be merged to give a much smaller set of registers. The start of the test path is shown on the right in Figure 3(a). The registers are given small Roman letters and are split i.e. an output register like *e* will eventually be merged with one or more input registers like *g*. The output registers are drawn as triangles as a reminder that they cannot be merged i.e. *e* and *f* must be separate physical registers. However, rectangular input registers like *a* and *c* could merge as could *a* and *d*. Even *a* and *b* could merge provided one was willing to tolerate the loss of randomness in the test vectors. This is explained in Section 3.1. In the data-path, the results of a new cycle of computation are placed in R15 and R16 every six C-steps. The test path, however, could be cyclic i.e. the output in *r* could be fed back into one or more of the input test registers *a, b,* c or *d*.

4.1 Constructing the register-merge graph

To build the test data-flow graph one starts by listing all the transfer paths in the data-flow graph; this is the list of paths that must be tested. There are four types of transfer paths:

- Transfers from a register into an operator's left input.
- Transfers from a register into an operator's right input
- Transfers from an operator into a register.
- Transfers from a register into another register.

Rather than test a complete operation i.e. a complete transfer path from an input register through the operator to the output register, we test the transfers associated with the operation, individually, as two separate transfers: a transfer from the input register into the operator and a transfer from the operator into an output register. Figure 3(b) illustrates the procedure for the example data-flow graph. For each transfer, the test registers that can take part in testing the transfer are shown alongside the data-path registers participating in the transfer. Figure 3(c) explains the construction of the *register-merge graph*. The register-merge graph is a graphical representation of the relations between the registers in the data-path and those in the test-path. For both the register-to-operator and operator-to-register transfers, the registers (nodes) which connect to the same pin on the same operator, are joined by an edge in the register-merge graph. A register-to-register transfer is represented by an edge between the nodes corresponding to the two registers.

Besides the necessity of merging registers to duplicate all transfer paths of the data-path in the test-path, one may also want to merge registers to reduce circuit area. A heuristic for register merging was developed by Midwinter [11]. The heuristic orders the mergers on the basis of an estimated saving of buses and MUXs. After extensive experimentation, she obtained the largest area saving when the registers were merged in the following order:

1. Transfers that have the *same source and same destination.* An examples is R4 and R11; both registers store the transfer out from the adder, $+_1$, as well as the transfer into the left input of multiplier, $\mathbf{x_1}$.
2. Transfers with the *same source.* An example is R8 and R15 which both receive data out from the multiplier, $\mathbf{x_2}$.
3. Transfers with the *same destination.* An example is R1 and R10. Both send data into the left input of adder, $+_1$.

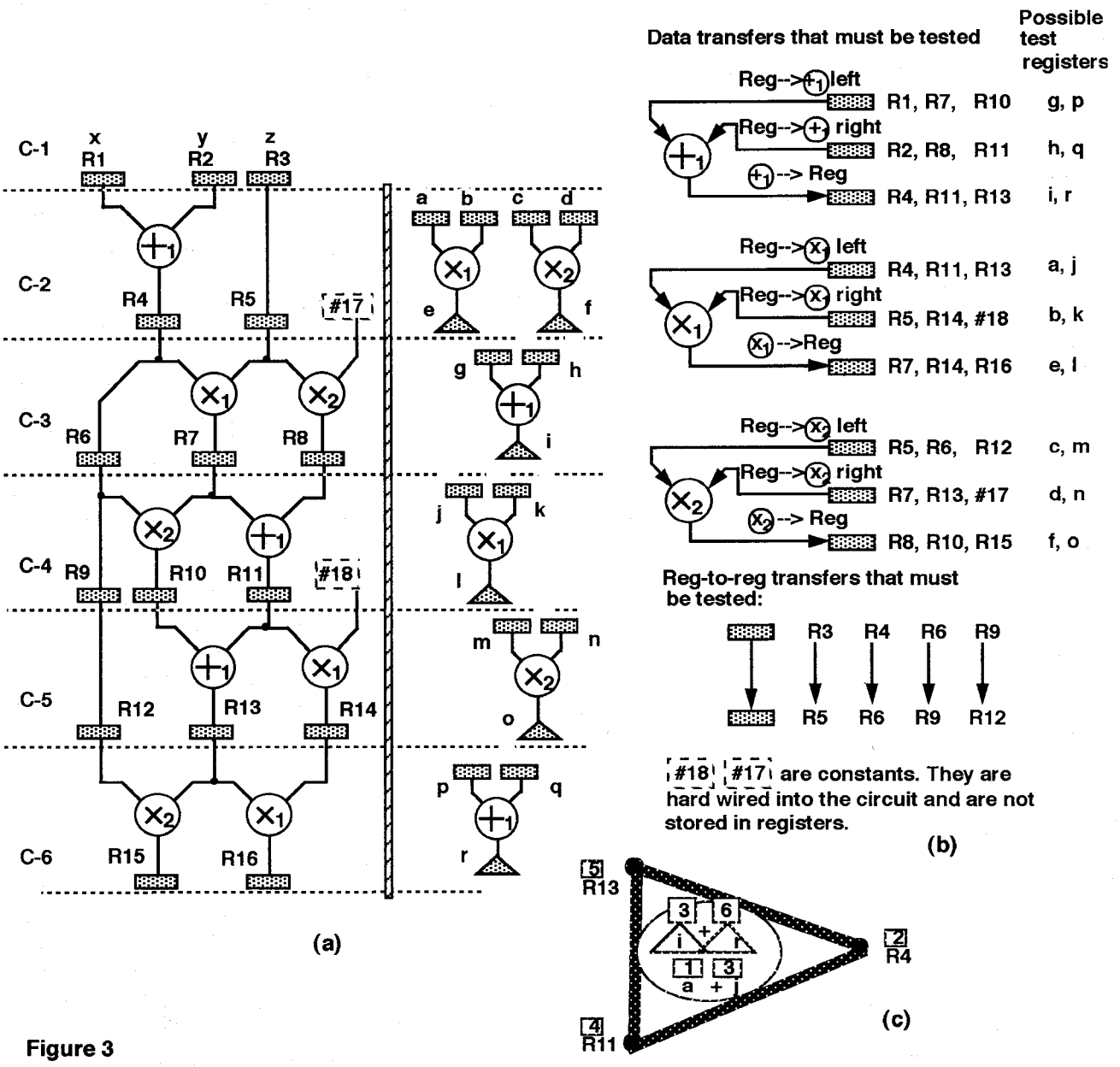


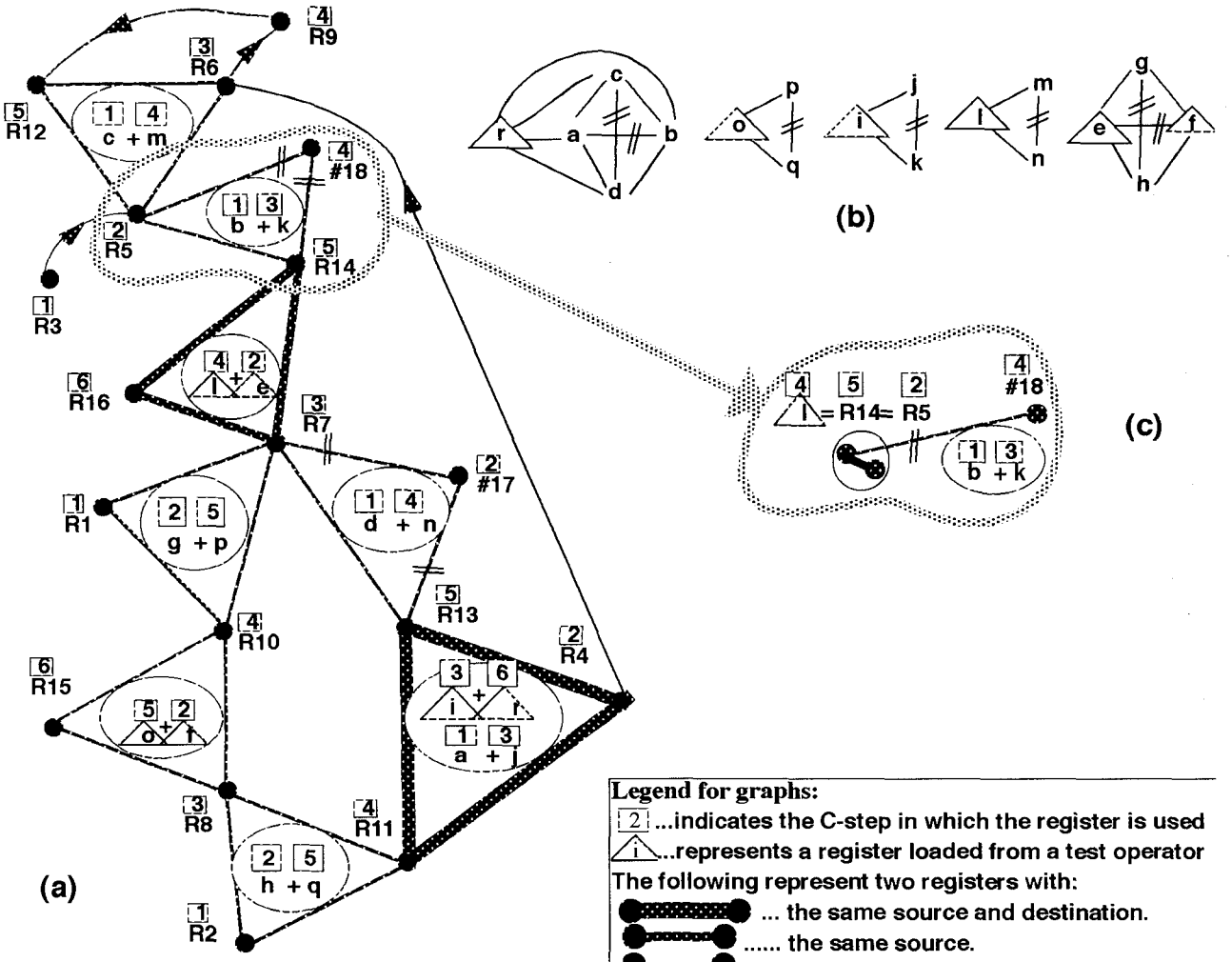
Figure 3

(a) The example data-flow graph on the left with the idle operators in each control step shown on right

(b) All the transfers that must be tested.

(c) Constructing the register-merge graph.

Consider the transfer from the adder output into registers R4, R11 and R13. In the test path the adder has an output into registers *i* and *r*. Thus registers R4, R11 and R13 must merge with *i* and *r*. Since three registers must merge with two, this means that at least one merger must take place between R4, R11 and R13. The triangle in the lower right symbolizes this. Showing registers *i* and *r* in an ellipse touching all three sides of the triangle, indicates that all three registers must, in some way, merge with *i* and *r*. In this case the same R4, R11 and R13 are also part of a transfer into x_1. The possible test registers for this transfer *a* and *j*, must also merge with R4, R11 and R13. Thus they also go inside the ellipse. The number in the square over the top of the register label is the C-step in which the transfer takes place. Placing the C-steps beside the register conveniently shows which mergers cannot be done. The graph is completed to include all the transfer paths in the data-flow graph.



Legend for graphs:

2 ...indicates the C-step in which the register is used

i ...represents a register loaded from a test operator

The following represent two registers with:

... the same source and destination.

..... the same source.

..... the same destination.

..... represents register-to-register transfer

..... represents two registers which cannot be merged, and still test all the data paths.

Figure 4

(a) The complete register-merge graph.

(b) Desirable test register mergers.

(c) The register-merging process.

4. Transfers between the same operators but with the source and destination interchanged. An example is R8 and R13 which store the transfers from x_2 to $+_1$ and $+_1$ to x_2 respectively.
5. Transfers with nothing in common.

Register-to-register transfers must be added to this list. Currently we place them about 3.5, slightly after *same-destination* transfers. The ordering of the mergers is represented on the register-merge graph by the thickness of the line that depicts the merger; the thickest line shows the most beneficial merger. The above ranking for mergers, however, is only considered after the mergers necessary to satisfy testability constraints.

One does not want the register-merge graph to indicate all mergable register combinations, or even the alternative of all non-mergable combinations. These grow rapidly. By showing only desirable mergers, the size of the register-merge graph is reduced from $O(c^2f^2)$ to $O(c^2f)$. Here f is the number of functional units in the data-flow graph and c is the number of C-steps. For the data-flow graph in Figure 3, $f = 3$ and $c = 6$. Note that the number of registers, or transfers, in the data-flow graph is $O(cf)$.

4.2 Reduction of the register-merge graph

The completed register-merge graph shown in Figure 4(a) may now be reduced. Reducing the register merge-

graph involves successively merging registers (nodes in the graph). Registers used in the same C-step in the DFG cannot merge. In the TDFG, two test output registers also cannot be merged if they are in the same C-step. However, test registers used as inputs to operators in the same C-step, can with some restrictions be merged. Such a merger may reduce the register count, however, in a C-step, merging of two input test registers to the same operator should be avoided because such a merger results in identical test vectors being applied at the two inputs of an operator. This can affect the fault coverage of the testing as explained in Section 3.1. Desirable and forbidden mergers for the test registers are shown by the small graphs in Figure 4(b).

The primary objective of the register merging is to satisfy the constraints set out by testability considerations; the secondary objective is to reduce the number of registers, buses or MUXs that will be required. Testability constraints or restrictions on the mergers are generated because we would like the TDFG to duplicate every path in the DFG. These restrictions might forbid some mergers or could force other mergers. Register-merging proceeds on the basis of an ordering of mergers. This ordering can be translated into the following set of ranked objectives:

1. Do any mergers on the graph that are forced.
2. Expand the influence of these forced mergers to adjacent nodes. Such an expansion may force other mergers, or may add some further restrictions.
3. If there are no forced mergers, merge the registers which reduce the MUX and bus area. These are the edges with the thickest lines.
4. If there is no other guide, choose the merger that causes the least restriction on other possible mergers.
5. Finally, check any remaining registers for potential mergers that might result in reducing the total number of registers required.

Restrictions or constraints on other mergers, that arise as a result of a particular register merge operation, are propagated outward on the graph from the merger and stop when no further restrictions are propagated. Restrictions include reduction of choice of test cycles that a merger forces on other transfers. This provides the data for objective 4. However if the choice of test cycles is reduced to zero for any transfer, the merger is recorded as impossible. Because of limitations on the length of the paper we cannot fully describe the register-mergers. However, the

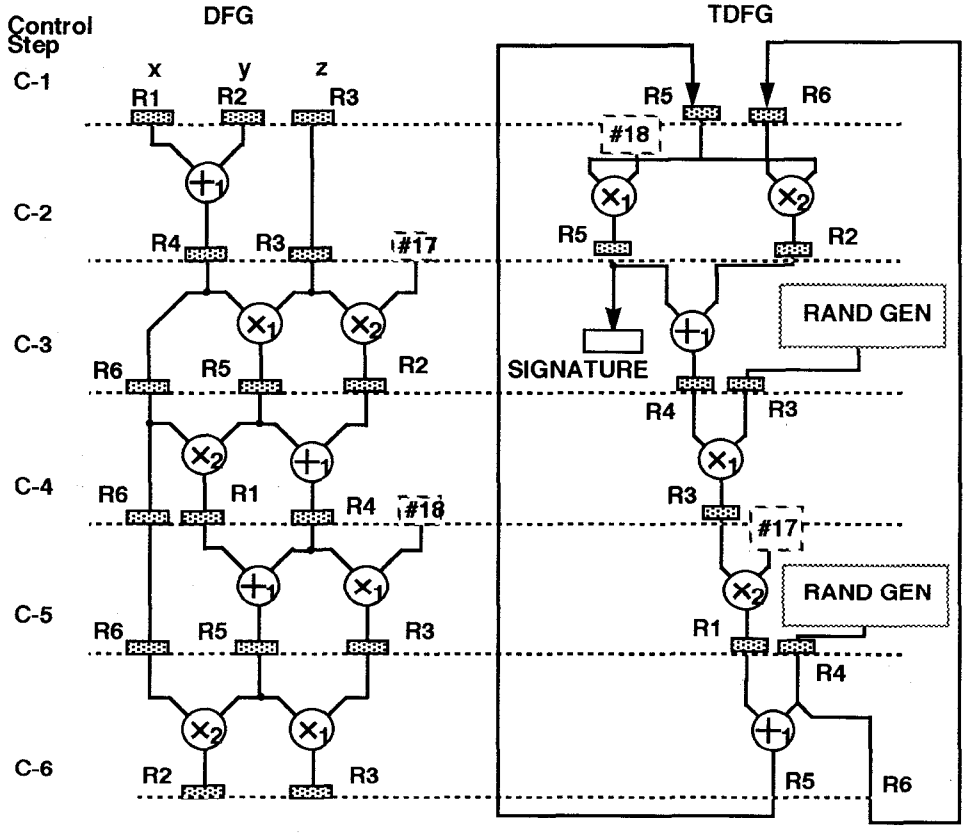


Figure 5 **The data-flow graph and completed test data-flow graph.**

merging process is illustrated in Figure 4(c). There, one notes that the immediate constant, #17 cannot merge with either registers R5 or R14. Since two of R5, #18 and R14 must merge, R5 must be merged with R14. As a result of this forced merger, R5 which must be tested by one of *e* or *l*, cannot be merged with *e* because R5 = R14 is used in control-step 2. This forces the merger of R5 with *l*. Each merger adds further incompatibility with adjacent nodes and may result in more forced mergers. The data-flow graph and the final test data-flow graph are shown in Figure 5. Pseudorandom numbers were injected into R3 and R4 which had no "natural" inputs. The signature can be collected at any convenient point. The operator hardware has not increased; testing adds three registers and doubles the number of MUXs.

Because of constraints on the length of the paper the procedure for building the test-flow graph as well as for reducing the graph could not be fully described. The authors can be contacted for details of the procedures.

5.0 Summary

The advantages of the test method are:

1. Testing can be done concurrently while the circuit is running.
2. It can detect transient, or soft errors.
3. It tests the circuit at-speed.
4. It does not require an increase in the area of large arithmetic elements.

The disadvantages are:

1. The test coverage will not be as high as with test methods that apply test vectors directly to operator inputs. We are examining ways to improve test coverage in concurrent testing.
2. There is an increase in the number of registers and the bus/MUX area. The controller area will increase since most of the control is for registers and MUXs.
3. The method depends on using slack time on the operators. A pipelined system would reduce this slack time, and might be hard to schedule.

6.0 References

[1] M. Abramovici, M. A. Breuer, A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, 1990.

[2] J. A. Abraham and P. Banerjee, "Fault Tolerant Techniques for Systolic Arrays," *IEEE Computer*, pp. 65-74, July 1987.

[3] K. K. Saluja, R. Sharma and C. R. Kime, "Concurrent Comparative Testing using BIST Resources", *IEEE International Conference on Computer-Aided Design*, pp. 336-339, 1987.

[4] T. Lee, N. K. Jha and W. H. Wolf, "Behavioral Synthesis of Highly Testable Data Paths under the Non-Scan and Partial Scan Environments", *Proc. 30th Design Automation Conference*, pp. 292-297, 1993.

[5] C. A. Papachristou, S. Chiu and H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs", *Proc. 28th Design Automation Conference*, pp. 378-384, 1991.

[6] C. H. Gebotys and M. I. Elmasry, "Integration of Algorithmic VLSI Synthesis with Testability Incorporation", *IEEE J. Solid-State Circuits*, pp. 409-416, April 1989.

[7] R. Singh, M. Townsend and J. P. Knight, "Concurrent Testing of Digital ASICs Synthesized from Data-Flow Graphs", *Proc. 6th Workshop on New Directions for Testing*, pp. 87-95, May 1992.

[8] D. P. Siewiorek and R. S. Swarz, The Theory and Practice of Reliable Design, Digital Press, Bedford, Mass, 1982.

[9] A. Y. Chan, "Easy-to-use Signature Analysis Accurately Troubleshoots Complex Logic Circuits", *Hewlett-Packard Journal*, vol. 28, pp. 9-14, May1977.

[10] K. Thearling and J. Abraham, "An Easily Computed Functional Level Testability Measure", *Proc. International Test Conference*, pp 381-390, 1989.

[11] J. Midwinter, "Improving Interconnect and Register Allocation for the Behavioral Synthesis of Digital Circuits", M. Eng. Thesis, Carleton University, Ottawa, Canada, March 1988.