VLSI Design
1993, Vol. 1, No. 1, pp. 45–60
Reprints available directly from the publisher
Photocopying permitted by license only

Analysis and Design of Regular Structures for Robust Dynamic Fault Testability

MICHAEL J. BRYAN and SRINIVAS DEVADAS
Department of EECS, MIT Cambridge, Massachusetts, USA

KURT KEUTZER
Synopsys, Mountain View, California, USA

Recent methods of synthesizing logic that is fully and robustly testable for dynamic faults, namely path delay, transistor stuck-open and gate delay faults, rely almost exclusively on flattening given logic expressions into sum-of-products form, minimizing the cover to obtain a fully dynamic-fault testable two-level representation of the functions, and performing structural transformations to resynthesize the circuit into a multilevel network, while also maintaining full dynamic-fault testability. While this technique will work well for random or control logic, it is not practical for many regular structures.

To deal with the synthesis of regular structures for dynamic-fault testability, we present a method that involves the development of a library of cells for these regular structures such that the cells are all fully path-delay-fault, transistor stuck-open fault or gate-delay-fault testable. These cells can then be utilized whenever one of these standard functions is encountered.

We analyze various regular structures such as adders, arithmetic logic units, comparators, multipliers, and parity generators to determine if they are testable for dynamic faults, or how they can be modified to be testable for dynamic faults while still maintaining good area and performance characteristics. In addition to minimizing the area and delay, another key consideration is to get designs which can be scaled to an arbitrary number of bits while still maintaining complete testability. In each case, the emphasis is on obtaining circuits which are fully path-delay-fault testable. In the process of design modification to produce fully robustly testable structures, we have derived a number of new composition rules that allow cascading individual modules while maintaining robust testability under dynamic fault models.

Key Words: *Synthesis for testability; Dynamic faults; Regular structures; Composition rules*

Recent methods of synthesizing logic that is fully testable for dynamic faults, namely path delay, transistor stuck-open and gate delay faults (e.g. Kundu and Reddy, and Kundu et al. [6, 7]), rely almost exclusively on flattening given logic expressions into sum-of-products form, minimizing the cover to obtain a fully dynamic-fault testable two-level representation of the functions, and performing structural transformations to resynthesize the circuit into a multilevel network, while also maintaining full dynamic-fault testability. While this technique will work well for random or control logic, it is not practical for many regular structures.

There are two major problems with applying these synthesis techniques to regular structures. First, for many of these type of circuits, the number of product terms in the flattened structure becomes prohibitive. Consider a binary adder as an example. In an adder, the number of product terms grows exponentially with the number of bits. For an N-bit adder, the most significant bit of the sum output has $2^{N+2} - 4$ product terms in its flattened representation. Another example is a parity generator. An N-bit parity generator has 2^{N-1} product terms in the flattened representation of the circuit. As a result of this exponential growth in the number of product terms, it can quickly become prohibitive in terms of both the CPU time required and the memory requirements to flatten even relatively small regular structures such as adders and parity generators.

A second problem is that in flattening the original logic expressions, the structure that the designer has created in the overall architecture is lost. This can result in a number of problems if the optimization

algorithms cannot synthesize an implementation which has similar area or performance characteristics, which is often the case for regular structures. For example, often a bit-slice approach works best for many data path structures. The basic building block can be optimized and laid out, and then the overall circuity constructed by simply replicating this one block many times. Once the structure is flattened, all of the information about the original structure is lost which may not be recoverable by synthesis procedures.

As a result of the problems identified with trying to flatten regular structures and then synthesizing a dynamic-fault testable implementation, it is desirable to develop an alternative method for dealing with such circuits. In Devadas and Keutzer [4] a simple composition rule for robustly path-delay-fault testable circuits was developed that allowed for the development of a robustly path-delay fault testable ripple-carry-adder and a parity generator. A number of new composition rules are developed here in order to develop a library of cells for these regular structures which are all fully dynamic-fault testable which can then be utilized whenever one of these standard functions is encountered.

The following sections use and extend the theory of Devadas and Keutzer [3, 4] to create a library of regular structures such as a variety of adders, arithmetic logic units, comparators, multipliers, and parity generators. The theory is used to determine if common structures are testable for dynamic faults, or *how they can be modified to be testable for dynamic faults while still maintaining good area and performance characteristics*. In addition to minimizing area and delay, another key consideration is to get designs which can be scaled to an arbitrary number of bits while still maintaining complete testability for delay faults. In each section, the emphasis is on obtaining circuits which are fully robustly path-delay-fault testable. This implies that they are fully robustly testable for gate-delay-faults and stuck-open faults as well [3]. When this is not achievable without significant area or speed penalties, methods of obtaining transistor stuck-open-fault testable and robustly gate-delay-fault testable circuits are analyzed.

PREVIOUS WORK

Because of the importance of regular arithmetic and logical structures in digital design, a number of researchers have sought to develop testing methods for these structures, but most of the attention has apparently been focussed on functional testing (for a survey see Abramovici et al. [1] chapter 8). These functional tests are useful in detecting faults that affect the logical behavior of circuits, but they do not address the temporal behavior of the circuit. Furthermore, many implementations of regular structures which are completely testable functionally, may be poorly testable for faults such as path-delay-faults, even when the functional vectors can be applied at speed.

In light of this, if there is interest in detecting delay faults then attention must be given to the precise logic-gate-level implementation of regular logic structures. The testability of transistor-level design of arithmetic regular structures is considered in Montoye [8]. Unfortunately, delay defects are not explicitly considered there and the results are tied to particular transistor structures in an *n*MOS custom design methodology.

DEFINITIONS

A gate has an input/output **stuck-at-1** if the logical value associated with the input/output is 1 independently of the value presented at the input. **Stuck-at-0** can be defined similarly. If a fault is stuck-at-1 untestable then the input net or gate in the circuit associated with that fault can be replaced by a constant 1. Similarly for stuck-at-0.

A circuit has a **gate delay fault** if there is one gate in the circuit such that the output of the circuit is slow to make a $0 \rightarrow 1$ (or $1 \rightarrow 0$) transition even when one or more of the gate's inputs change values. Each single gate delay fault is assumed to be so catastrophic as to cause a delay along any path to any output.

A gate in a CMOS circuit has a **transistor stuck-open** fault if there is one transistor in the gate that is permanently non-conducting.

A circuit has a **path delay fault** if there exists a path from a primary input to a primary output via a set of gates and interconnecting nets such that a primary input event is slow to propagate *along the path* to the primary output.

Necessary and sufficient conditions for path-delay-fault testability were given in Devadas and Keutzer [3]. In order for a gate delay fault to be detected, it is sufficient for a path through the gate to be robustly testable. Similarly, in order for a stuck-open fault at the input of a gate to be detected, it is sufficient for a path through that input to be robustly testable.

A **robust test** for a dynamic (gate delay, transistor stuck-open, path delay) fault is one which is valid

under arbitrary delays and is therefore not invalidated by hazards or races.

Throughout this paper we will assume that a robust test for a dynamic fault in a circuit C is a vector pair $\langle v_1, v_2 \rangle$ such that $C(v_1) = 0$ and $C(v_2) = 1$. Let the expected transition time on the vector pair be τ. The application of the vector pair is as follows: Vector v_1 is applied to C and the values on nets are allowed to settle for an arbitrary amount of time. Vector v_2 is then applied to C. At time τ the output of C is sampled; if the value is 1 then no fault is detected, otherwise a fault is detected. Next, the vector pair $\langle v_2, v_1 \rangle$ is applied to propagate the opposite event along the path and detect faults corresponding to the $1 \rightarrow 0$ event.

ADDERS

There are numerous types of adder designs which provide various performance and area tradeoffs. We have analyzed various adder structures to determine if they are testable for the various classes of dynamic faults, or how they can be made to be testable for these classes of faults. Since data path structures often form the critical path in a system, it is important that these paths be testable for delay. We will not describe the analysis in its entirety (the reader is referred to Bryan [2]) but will simply present a novel design of a carry bypass adder which is fully testable for path delay faults.

The composition rule below was proved in Devadas and Keutzer [4].

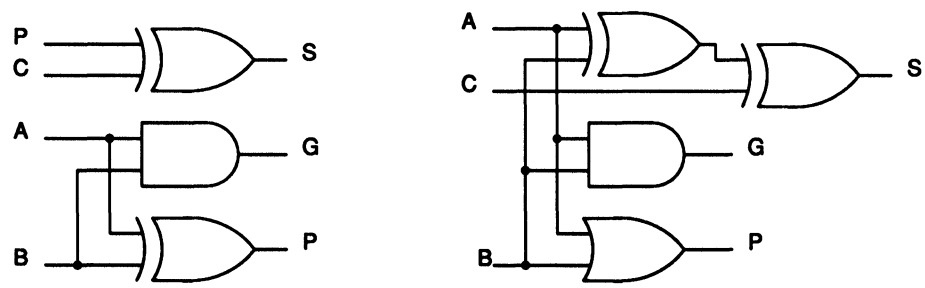
Composition Rule: *Given a set of robustly path-delay-fault testable circuits, $C_1, C_2, \cdots C_N$, if for $1 \leq i < N$ a single output of C_i, namely l_i feeds C_{i+1} and for $1 \leq i, j \leq N$ when $i \neq j$ the inputs of C_i and C_j are disjoint, then the composition is fully robustly path-delay-fault testable.*

All of the circuit implementations in this paper are given in terms of AND gates, OR gates, and inverting buffers, as well as exclusive OR gates which are modeled using these three types of gates. In some of the cases the inverters are not explicitly represented in the logic diagrams, but are implied by inversion bubbles on the inputs or outputs of the other gates. In the actual models these inverters must be explicitly represented in order to accurately represent all possible gate delay faults. Describing the circuits in this manner provides a good method to easily analyze whether the implementation is testable for robust path-delay-faults as well as the other classes of dynamic faults. The paths in an actual transistor-level implementation in a particular technology and design style (such as static CMOS) will have a one-to-one correspondence to the paths in the gate-level implementations used in this paper.

The carry lookahead and bypass adders create both the propagate (P) and generate (G) terms for each bit. The propagate term is asserted whenever a carry input would propagate through the adder section based on the values of the operands. Thus for a 1-bit section, $P = A \oplus B$. The generate term is asserted whenever a carry is generated by an adder section based on only the values of the operands (i.e. regardless of the value of the carry input). Thus for a 1-bit section, $G = A \cdot B$. The carry output which is created from the propagate and generate terms is defined as $CO = G + P \cdot CI$. In Figure 1(a), the standard logic for generating the propagate, generate, and sum outputs for each bit is shown. An alternate representation is shown in Figure 1(b).

Carry Bypass Adder

The carry bypass adder is just an offshoot of the carry lookahead adder. It becomes very inefficient to extend the carry generation scheme for lookahead ad-



(a) Conventional Implementation (b) Alternate Implementation

FIGURE 1 Propagate/generate logic implementation.

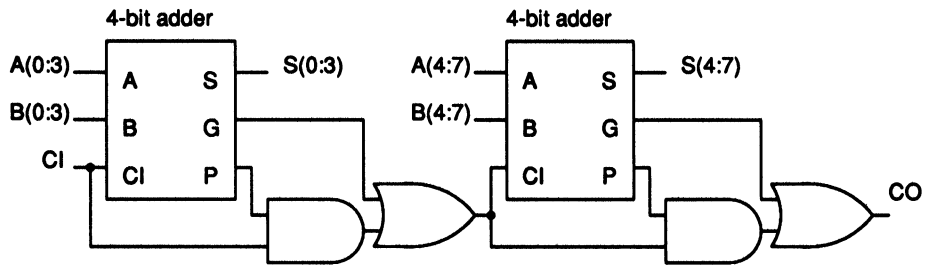


FIGURE 2 Carry bypass adder implementation.

ders to very large bit-widths. Typically, the carry signals are only calculated in this manner for up to 4-bit sections. By creating a propagate and generate signal for each 4-bit section, the carry signal can be bypassed through each stage as shown in Figure 2. The logic for the cumulative propagate and generate signal for each 4-bit stage can be expressed in terms of the propagate and generate signals for each bit, where $P = P0 \cdot P1 \cdot P2 \cdot P3$ and $G = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$. An implementation is shown in Figure 3. Using a carry bypass scheme allows each N-bit stage (4-bit in this case) to be identical, and thus makes layout simpler since only one stage needs to be laid out, and then an arbitrary number of these blocks can be interconnected to form a larger adder.

The carry bypass generation is an algebraic factorization of the carry signal. This circuit is thus fully testable assuming that the individual propagate signals for each bit are implemented as $P = A + B$, i.e. the alternate implementation of Figure 1(b).[1] If the propagate signal is implemented as $P = A \oplus B$, then the circuit will not be fully testable.

For a 28-bit carry select adder composed of 7 4-bit stages, the longest path for a carry select adder would be 10 stages of logic (each stage being two levels). For the carry bypass adder, the worst case delay for the carry output signal would be 8 stages of logic—7 carry bypass chains and the propagate/generate logic in the first stage. The worst case sum output would go through 10 stages of logic—6 up to the carry input of the last stage, and 4 in the last stage of the adder to generate the sums (assuming a ripple adder). By adjusting the stage size, the performance of the carry bypass adder can be made to exceed that of the carry select adder. However, the significant advantage of the carry bypass adder is that it does not need to generate two different sums at each stage and then multiplex the outputs, creating a large savings in area.

Summary of Testability of Adders

It was shown [2] that all four types of adders analyzed can be made to be fully testable for all three classes of dynamic faults. It was also shown that the carry select adder becomes inferior to the carry bypass adder and has no significant advantages which would result in it being chosen over the carry bypass adder. While versions of the carry lookahead and carry bypass adder were shown that are fully testable, the most commonly used implementation of these adders is not fully testable. The testable version adds a small amount of logic to the common implementation (1 OR gate per adder bit), but retains the same performance characteristics or improves the performance slightly.

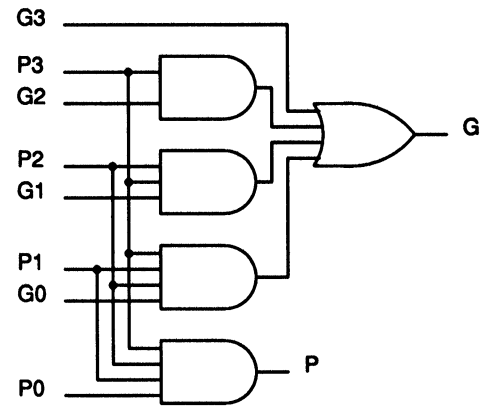


FIGURE 3 Cumulative propagate and generate implementation.

[1]Note that the commonly used implementation of the carry bypass adder has a single stuck-at fault redundancy. Removing this stuck-at fault redundancy results in a circuit that is considerably slower than the original circuit. In Keutzer et al. [5], an implementation of a carry bypass adder that was fully testable for stuck-at faults with equal or better performance than the redundant adder was given. However, the circuit of Keutzer et al. [5] was not fully robustly path-delay-fault testable. The circuit presented here is completely robustly path-delay-fault testable, and maintains the performance of the original circuit.

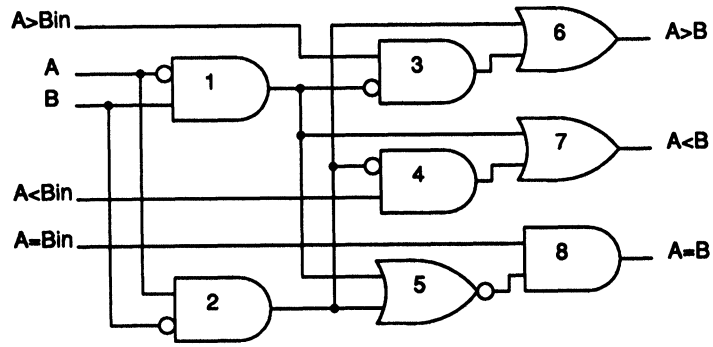


FIGURE 4 1-bit extensible comparator implementation.

COMPARATORS

Binary magnitude comparators are another type of regular structure. It is desirable to have a comparator which is fully testable for dynamic faults that can also be scaled to an arbitrary number of bits. Performance is also an important cost function when evaluating alternate comparator designs. In this section the testability of some typical comparator implementations and methods to cascade comparators are analyzed.

Ripple Comparator

An N-bit comparator can be constructed by simply connecting N 1-bit extensible comparators in series. An implementation of a 1-bit extensible comparator is shown in Figure 4. It has five inputs, 2 of which are the operands A and B, and the remaining three are the results of the comparison of the less significant bits. The three outputs indicate whether A is greater than, less than, or equal to B. The implementation shown in Figure 4 is fully testable for dynamic faults. It has 14 gates, 25 links, and 15 paths, all of which are robustly testable.

The 1-bit extensible comparator shown in Figure 4 can be cascaded together to form an N-bit comparator by simply connecting the comparison outputs of the ith stage to the comparison inputs of the i + 1th stage. A 3-bit ripple comparator is shown in Figure 5. Note that the first stage has been simplified by using the fact that $A > B_{in} = 0$, $A < B_{in} = 0$, and $A = B_{in} = 1$. The ripple comparator is fully testable for path delay faults since each stage is fully testable and the inputs that affect each output of a given stage are fully controllable. Note that the entire input set for a given stage is not fully controllable, since the inputs $A > B_{in}$, $A < B_{in}$, and $A = B_{in}$ for a given stage can only take on the values ⟨100⟩, ⟨010⟩, or ⟨001⟩. However, the output $A > B$ of the ith stage only depends on the inputs A_i, B_i, and $A > B_{i-1}$. Thus for $A > B_i$ to be testable for path delay faults only requires that A_i, B_i, and $A > B_{i-1}$ be independently controllable and that the stage itself be fully testable. $A > B_{i-1}$ is independent of A_i and B_i, so $A > B$ is fully testable. Similar arguments can be made for the outputs $A < B$ and $A = B$.

Figure 5 above indicates a more general composition rule than the rule given in Devadas and Keutzer [4].

Composition Rule: *Consider a set of robustly path-delay-fault testable circuits, C_1, C_2, $\cdots$ C_N, where for $1 \leq i < N$ outputs l_{i1}, $\cdots$ l_{iP} from C_i feed C_{i+1}, each*

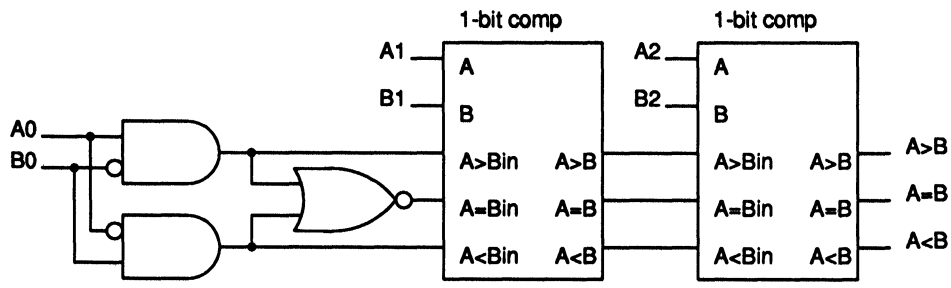


FIGURE 5 3-bit ripple comparator implementation.

C_i block receives another set of inputs, namely I_i such that for $1 \leq i, j \leq N$ when $i \neq j$, I_i and I_j have no common inputs. If, for $1 \leq i \leq N$, $1 \leq k \leq P$, l_{ik} only depends on I_i and l_{i-1k}, then the composition is fully robustly path-delay-fault testable.

Each circuit C_i can be broken up into P parallel circuits, each receiving I_i and l_{i-1k}. Then, given Composition Rule on Carry Bypass Adder, we have the above result.

Parallel Comparator

While an arbitrary size ripple comparator can be constructed, delays become prohibitive when performing comparisons of large operands. An alternate method of performing comparisons of operands with a large number of bits is to construct a parallel comparator. A parallel comparator of two N-bit operands can be constructed out of N/M M-bit extensible comparators. An implementation of an 8-bit comparator constructed out of 4 2-bit extensible comparators is shown in Figure 6. In this commonly used implementation, the comparator with the least significant bits of the operands is connected in the typical manner. However, the comparators that receive the other bits of the operands can accept one additional bit, with the operand A_i being connected to the $A > B_{in}$ input, and the operand B_i being connected to the $A < B_{in}$ input. The $A = B_{in}$ input is tied low. In addition to the comparators which receive the primary inputs, one additional comparator is required. This comparator receives as inputs the $A > B$ and $A < B$ outputs of the other comparators, with the result of the comparison of the most significant input bits being connected to the most significant bits of the final comparator.

The parallel comparator implementation in Figure 6 is not fully testable for dynamic faults. The untestability occurs since the A and B inputs of the final comparator are not fully controllable.

Alternate Parallel Comparator

Figure 7 is an alternate method for implementing a parallel comparator. The comparator in Figure 7 is a 6-bit comparator, and is composed of 3 initial stages each of which does a 2-bit comparison, and then an output stage which performs the final comparison. This implementation is fully testable for path delay faults. It has a total of 85 gates, 187 links, and 144 paths, all of which are robustly testable.

Figure 8 shows the detailed implementation of the initial 2-bit stage used in the parallel comparator of Figure 7. There are 4 outputs from each of the comparators in the input stage. These outputs are $A > B$, $A < B$, EQA, and EQB. Output $A > B$ is asserted whenever A is greater than B, and $A < B$ is asserted whenever A is less than B for the bits of the operands which are inputs to the particular stage. Output EQA is the minimal expression obtained from using $A = B$ as the ON-set, and $A > B$ as the DC-set. Thus

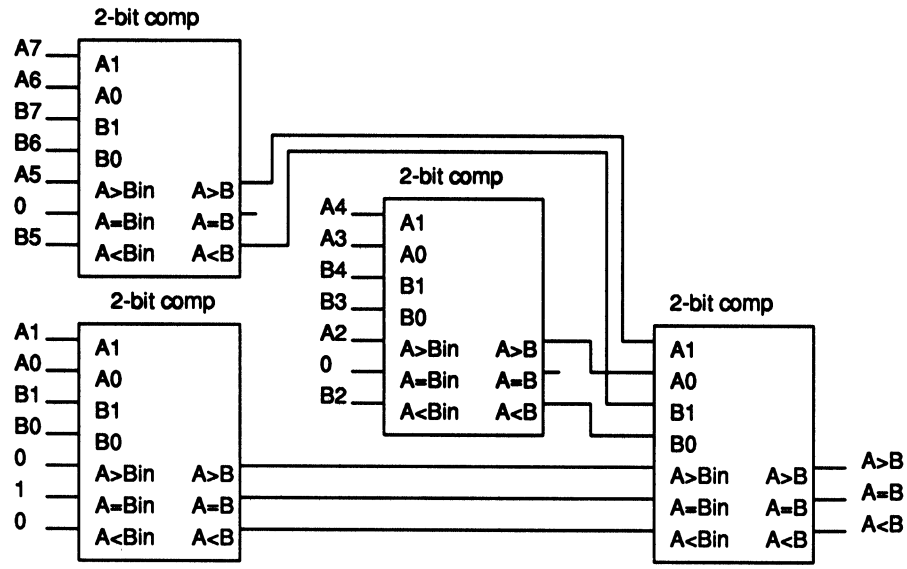


FIGURE 6 8-bit parallel comparator implementation.

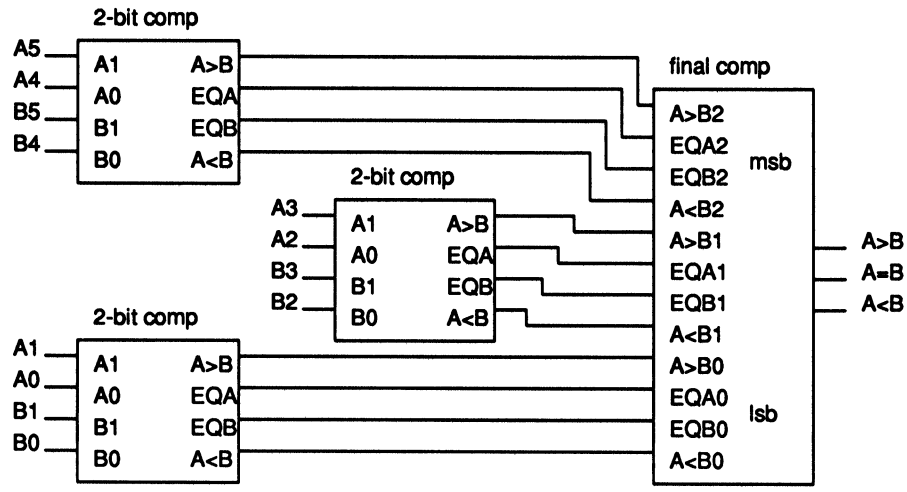


FIGURE 7 Alternate parallel comparator implementation.

whenever EQA is asserted, A is either greater than or equal to B, but $EQA \neq A \geq B$. Likewise, output EQB is the minimal expression obtained from using $A = B$ as the ON-set, and $A < B$ as the DC-set. Thus whenever EQB is asserted, A is either less than or equal to B.

Figure 9 shows the detailed implementation of the final comparison stage used in the parallel comparator of Figure 7. This logic takes as inputs the 4 outputs of each of the input stages and generates the final outputs of the comparator. The output $A > B$ is asserted whenever input $A > B_i$ and inputs $EQA_{i+1:n-1}$ are asserted, n being the number of input comparator stages, numbered from 0 to $n - 1$. The EQA terms can be considered the equivalent of the propagate terms in the carry lookahead adder. In order for an assertion of the $A > B$ output of a given input stage to cause an assertion of the $A > B$ output of the overall circuit, all of the EQA terms of the more significant input stages must be asserted. The $A < B$ output is defined similarly, with it depending on the $A < B$ and EQB inputs. The $A = B$ output is asserted whenever all of the EQA and EQB inputs are asserted. Figure 7 gives rise to the following composition rule.

Composition Rule: *Given a set of individually robustly path-delay-fault testable circuits,* C_1, C_2, $\cdots$ C_N

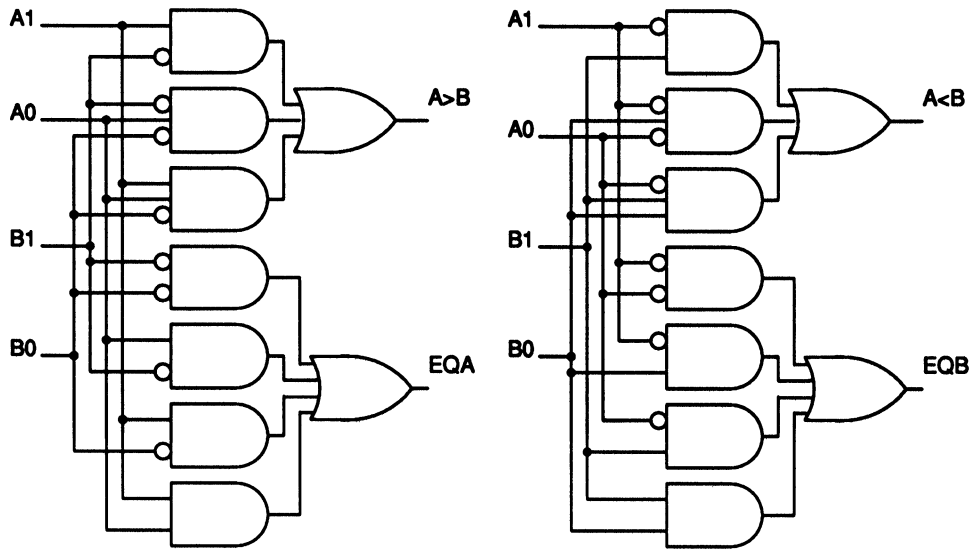


FIGURE 8 Alternate 2-bit comparator initial stage.

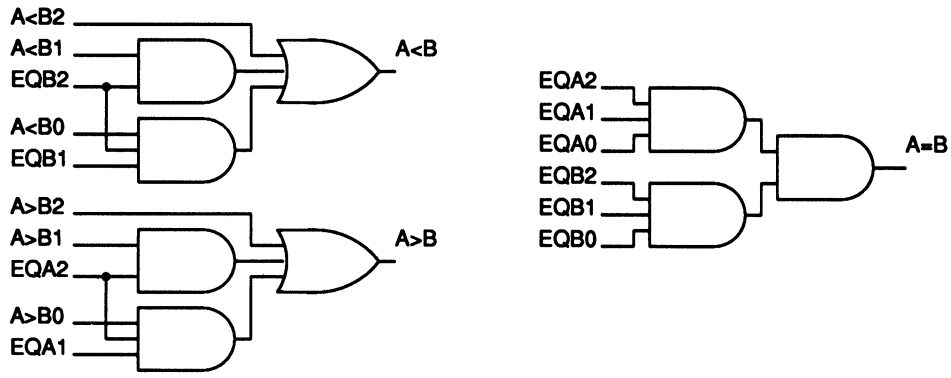


FIGURE 9 Alternate comparator final stage.

and D such that C_1, C_2, $\cdots$ C_N feed D, if for $1 \leq i, j \leq N$ when $i \neq j$ the inputs of C_i and C_j are disjoint, for $1 \leq i \leq N$ outputs l_{i1}, $\cdots$ l_{iP} from C_i feed D, and the side-inputs on the paths in D beginning from any l_{ij} can be controlled to 0/1 by l_{km}, $k \neq i$, $1 \leq m \leq P$, then the composition is fully robustly path-delay-fault testable.

Since each C_i is fully testable, a transition through any path can be propagated to an output l_{ij}. A transition on l_{ij} can be propagated through paths in D, given that the appropriate values can be produced by the *other* C_j blocks, that have disjoint inputs from C_i.

There are two major disadvantages with this alternate parallel comparator. These disadvantages are the extra area required to implement the circuit, and the problems which occur when scaling it to a larger number of bits. For small comparators, the size is comparable to that of the other comparators presented. For a 3-bit comparator, the ripple comparator, parallel comparator constructed out of ripple comparators, and the alternate parallel comparator would each have 48 literals in their multilevel cover. However, for an 8-bit comparator, the size of the first two would scale linearly to 128 literals, while the alternate comparator would increase to approximately 220 literals. The testability of the alternate comparator can be maintained for larger designs, but the area penalty becomes increasingly worse. Thus the design does not scale well. From a performance standpoint, the alternate parallel comparator is faster than the original parallel comparator presented. For the case of an 8-bit comparator, the longest path in the original comparator would be through 8 gates, while the longest path in the alternate comparator will be through 4 gates. This advantage in speed will gradually be lost for larger comparators.

From a functional standpoint, an equivalent comparator to that shown in Figure 7 could be constructed by generating a single $A = B$ output from each input comparator stage. This signal could then be used in place of the EQA and EQB terms in the final comparison, with $A > B$ only being asserted when inputs $A > B_i$ and $A = B_{i+1:n-1}$ are all asserted. However, this implementation is not fully testable. All of the 70 gates and 172 links are robustly testable, but only 144 out of 192 paths are robustly testable. It can be shown that all 48 untestable paths correspond to stuck-at fault redundancies, and if they are all removed from this comparator implementation, the circuit of Figure 7 would result. However, this design does not scale well to generate arbitrarily large comparators, and in general the design is inferior to the others presented in this section.

It is of interest to note that when the comparator "propagate" signals EQA and EQB are defined to be simply $A = B$, this is equivalent to the case when the propagate signal for the carry lookahead adder was defined to be $P = A \oplus B$. Both of these cases define the propagate signal to allow a signal from a lower stage to propagate through only when the signal can propagate through and when the signal is not generated at the current stage. Both of these cases are also not fully testable for path delay faults. However, when the propagate signal is defined to allow a signal from a lower stage to propagate through even if the signal is generated at the current stage, the circuits both become fully testable.

Bypass Comparator

The disadvantage of the parallel comparator of Figure 7 over that of Figure 6 is that the comparator of Figure 7 requires two different cell types to be de-

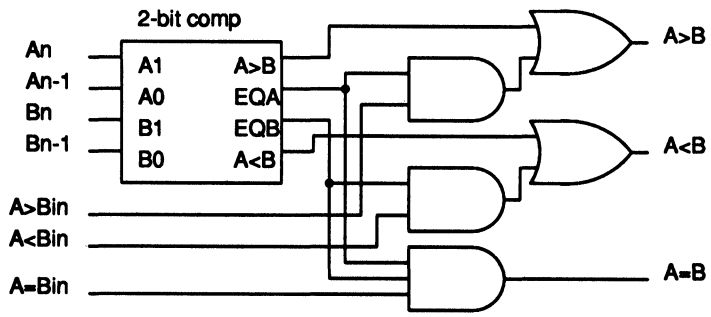


FIGURE 10 Bypass comparator.

signed and laid out. In addition, it does not scale well, and requires more area to implement. A comparator which is fully testable for dynamic faults but which only requires a single cell type is shown in Figure 10. This comparator is similar to the carry bypass adder (see appropriate section). Each stage is composed of a comparator which generates the $A > B$, $A < B$, EQA, and EQB signals such as that shown in Figure 8. The stage also receives the $A > B$, $A = B$, and $A < B$ signals from the preceding stage, and adds some additional logic to generate $A > B$, $A < B$, and $A = B$ signals which reflect the total result of the comparison to that point. Thus no final output stage is needed. The disadvantage is slower performance. In the case of the N-bit comparator constructed out of stages of M-bits each, the longest path would be through one M-bit comparator, and $N/M - 1$ stages of bypass logic. This implementation is, however, much faster than the ripple comparator of said named section. This design does require more area than the comparators which were not fully testable, but the area is less than that of the comparator of the previous section. An 8-bit comparator would have 195 literals in the multilevel cover for the circuit. This is about 50% more than for the ripple comparator, but since the area for this design scales linearly with the number of bits of the comparator, the area penalty will not grow any worse for larger designs.

Comparator Tree Structure

In the preceding sections, three fully testable comparators were presented. Each of these designs has some weaknesses associated with it. The ripple comparator has a delay which grows linearly with the number of bits in the comparator. It does, however, have a compact area. The alternate parallel comparator has a delay which is largely independent of the size of the comparator. However, it has a large area penalty which becomes increasingly worse for large comparators. The bypass comparator has a delay which also grows linearly with the number of bits, although it is faster by a constant factor than the ripple comparator. This increase in performance results in a 50% increase in area over the ripple comparator. In this section, another comparator is presented which has good performance and area characteristics, but which has somewhat reduced testability.

If full robust path-delay-fault testability is not essential, a comparator can be constructed which is fully gate-delay and stuck-open fault testable, has an area approximately 30% less than that of the ripple comparator, and which has performance approaching that of the parallel comparator. A block diagram of this comparator is shown in Figure 11. It is implemented using a binary tree structure. The input cells determine if a given bit of the operand A is greater than or less than the corresponding bit of the operand B. The subsequent stages compare sets of two $A > B$ and $A < B$ signals from the previous level and generate a single pair of $A > B$ and $A < B$ signals. This continues until a final result is achieved. The basic cells for this comparator are shown in Figure 12.

The structure of Figure 11 inspires the following rule. The rule can be used repeatedly to show that the comparator tree is robustly stuck-open and gate-delay-fault testable.

Composition Rule: *Given a robustly stuck-open fault (gate delay fault) testable circuit, C_1, and a robustly path-delay-fault testable circuit C_2 such that outputs $l_1, \cdots l_P$ from C_1 feed C_2, if a transition can be robustly propagated through every link (gate) in C_1 to some output of C_1, namely l_j, while holding the l_k, $k \neq j$ at constant values that allow the robust propagation of this event through some path in C_2, then the composition is fully stuck-open fault (gate-delay-fault) testable.*

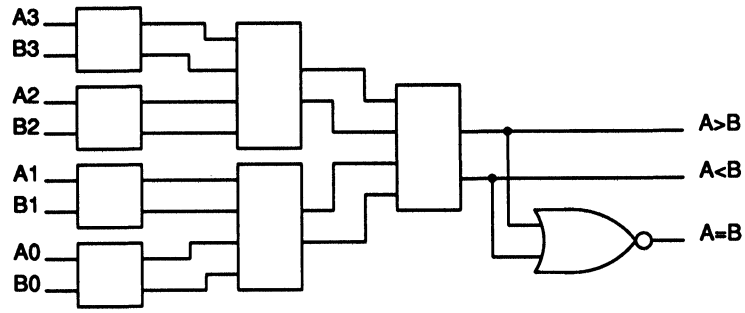


FIGURE 11 4-bit comparator tree.

Holding the l_k, $k \neq j$ to constant values is necessary to ensure that the transition on l_j can be propagated robustly through C_2. Stuck-open or gate delay faults in C_2 can be detected by robustly propagating an event along a path that passes through the appropriate links or gates.

The delay through this comparator is proportional to $\log_2 N$, N being the number of bits being compared. It is also possible to implement a similar comparator using a scheme in which each subsequent stage combines the results of three stages at the previous level, making the delay proportional to $\log_3 N$. The area saving in this binary tree comparator design is a result of not generating an $A = B$ signal at every level in the design. This signal is generated with a single NOR gate at the output of the circuit, and thus a 30% area improvement can be obtained over the ripple comparator for an 8-bit comparator (90 literals versus 128 literals). This same percentage improvement in area is also maintained for larger comparators. Thus this design scales well in terms of area, performance, and testability. The disadvantage is that it is not fully testable for the most restrictive class of delay faults, namely robust path-delay faults.

ARITHMETIC LOGIC UNITS

Arithmetic logic units are another type of regular structure for which it is highly desirable to be fully testable for dynamic faults since they often comprise the critical path within the design. Typically an ALU contains a high degree of shared logic and thus is generally not fully testable. We refer the reader to Bryan [2] for the analysis and design of adder/incrementer circuits and adder/substractor circuits for dynamic fault testability. We summarize the composition rules that are useful in designing such structures.

We do not restrict the subcircuits to have disjoint inputs as in the Composition Rule dealing with Carry Bypass Adder.

Composition Rule: *Given a set of robustly path-delay-fault testable circuits, C_1, C_2, $\cdots$ C_N such that for $1 \leq i < N$ a single output l_i of C_i feeds C_{i+1} and common inputs I_1, $\cdots$ I_P that feed all the C_i, if the remaining inputs to C_i can damp a transition on any input I_j before convergence with paths from l_i, then the composition is fully stuck-open fault and gate-delay-fault testable.*

Stuck-open faults corresponding to I_j links are propagated to l_i or the outputs of some C_i. In the former case, the transition can be propagated through C_{i+1} robustly, since the transition on I_j never reconverges.

We also give a condition different from the Composition Rule on Arithmetic Logic Units that maintains testability.

Composition Rule: *Given a set of robustly path-delay-fault testable circuits, C_1, C_2, $\cdots$ C_N such that*

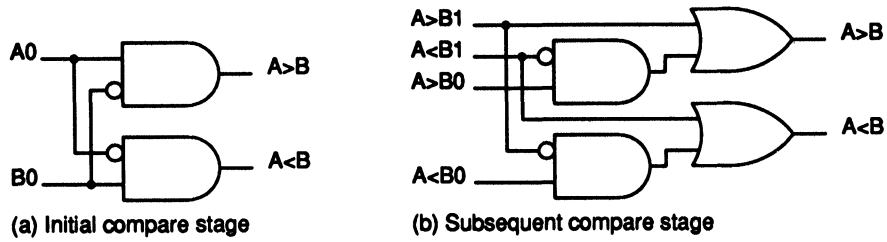


FIGURE 12 Comparator tree basic cells.

for $1 \leq i < N$ a single output l_i of C_i feeds C_{i+1}, and common inputs $I_1, \cdots I_P$ that feed all the C_i, if for each C_i (1) all links from inputs I_j in C_i are testable at some output of C_i not equal to l_i and (2) a path through other links in C_i is robustly testable up to the point of output l_i or some other output of C_i by placing constant values on the I_j, then the composition is fully gate-delay-fault and stuck-open fault testable.

Links connected to any I_j of a C_i block are tested at outputs that do not feed other blocks, without reconvergence occurring between I_j transitions. Other links do not require I_j transitions and can be tested through l_i (further propagation is required) or at the outputs of C_i.

PARALLEL MULTIPLIERS

A binary multiplier can be implemented as an adder along with some additional logic. As a result, multiplier circuits have a number of the same properties that adders exhibit. However, it is much more difficult to design a multiplier which is fully testable for dynamic faults due to the large amount of reconvergence present in a multiplier. In this section the testability of parallel multipliers is analyzed, both for pipelined and non-pipelined designs.

To perform a parallel multiply, the bitwise AND of each pairwise combination of the two operands is performed, and then the appropriate AND outputs are added together to form the product outputs. The sum of the indices of the operands which are ANDed together determines which product term the bitwise AND contributes to. For example, $X0Y0$ contributes to $P0$, while $X2Y2$ contributes to $P4$. These relationships between the multiplier partial products are shown in Figure 13 for a 4-bit by 4-bit multiply. $X_{0:3}$ and $Y_{0:3}$ are the two operands to be multiplied, and $P_{0:7}$ is the product.

A parallel multiplier of n bits by m bits can be constructed to be a $n \times m$ array of cells which compute the partial products and perform the sums. Each of these cells computes the partial product of the operand bits which intersect at that cell, and then adds that value to the incoming sum passed to the cell and generates an outgoing sum and carry. There are two typical ways in which this multiplier array is constructed. One method uses rows of carry propagate adders, the other uses rows of carry save adders. We will analyze the carry save multiplier here.

Carry Save Multiplier

Figure 14 is a block diagram of a 4-bit by 4-bit parallel multiplier using carry save adder rows, with operands X and Y, and product P. Figure 15(a) shows the pin assignments for the basic cell labeled A-ADD in Figure 14, while Figure 15(b) is a gate-level implementation of the cell. This cell is simply a full adder with two inputs $A1$ and $A2$ which are ANDed together to form one of the operands. In the multiplier array, the outgoing sums (S) are passed to the incoming sum (B) of the cell that is below and to the right of the current cell. If the current cell is on the right edge of the array then the outgoing sum is one of the final product outputs. The carry output is passed to the carry input of the cell immediately below the current cell. The carry inputs of the top row of cells are tied to a logic low, and the incoming sums of the top row and left column are also tied to logic low.

By implementing the multiplier array using carry save adder rows, an additional set of full adders needs to be added after the bottom row to propagate the final carry between the most significant product bits. In the multiplier shown in Figure 14, this adder is represented by the three cells labeled ADD at the bottom of the array. Each cell is a 1-bit full adder, with the two operands entering at the top, the carry

				$X3$	$X2$	$X1$	$X0$
				$Y3$	$Y2$	$Y1$	$Y0$
				$X3 \cdot Y0$	$X2 \cdot Y0$	$X1 \cdot Y0$	$X0 \cdot Y0$
			$X3 \cdot Y1$	$X2 \cdot Y1$	$X1 \cdot Y1$	$X0 \cdot Y1$	
		$X3 \cdot Y2$	$X2 \cdot Y2$	$X1 \cdot Y2$	$X0 \cdot Y2$		
	$X3 \cdot Y3$	$X2 \cdot Y3$	$X1 \cdot Y3$	$X0 \cdot Y3$			
$P7$	$P6$	$P5$	$P4$	$P3$	$P2$	$P1$	$P0$

FIGURE 13 Multiplier partial products.

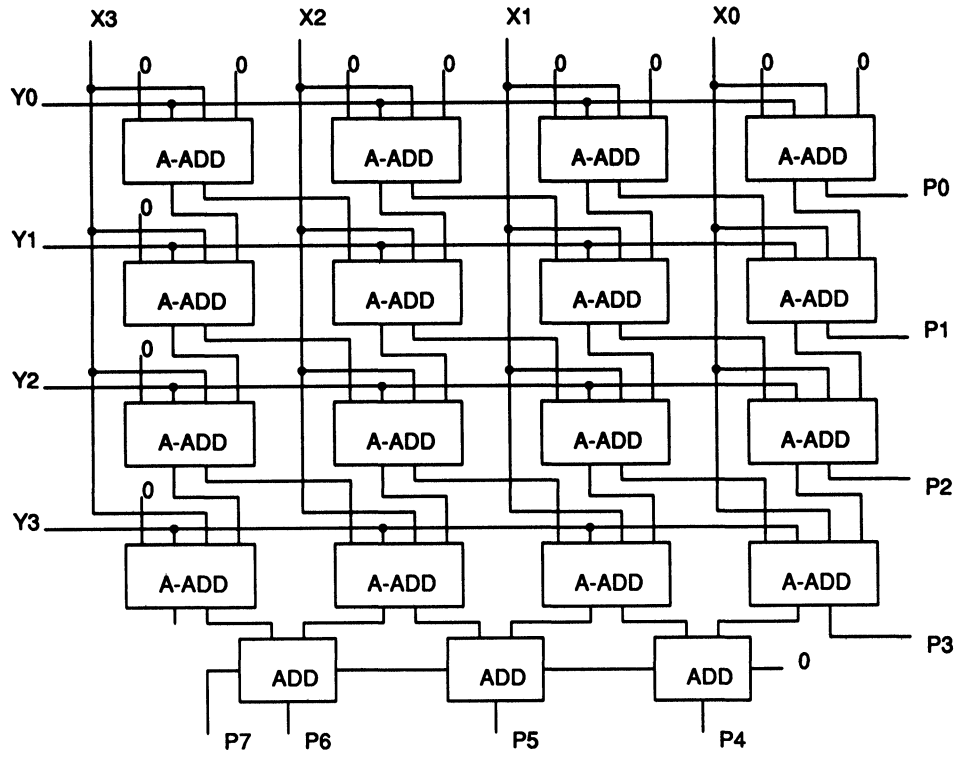
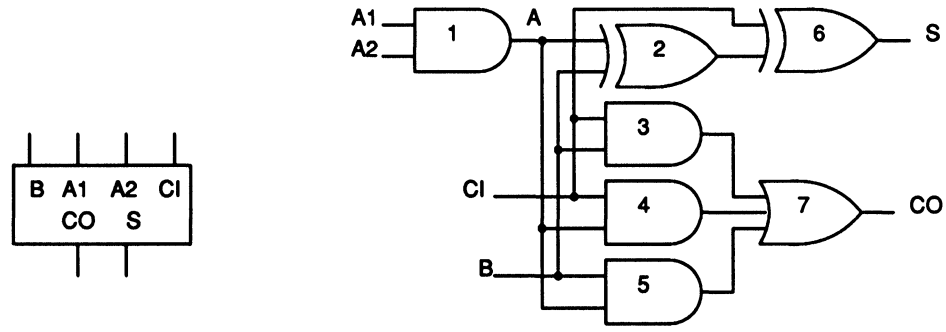


FIGURE 14 Carry save parallel multiplier implementation.

input on the right, and the carry output leaving on the left and the sum on the bottom.

While each cell in the multiplier array can be identical with some of the inputs tied to logic low as explained in the preceding paragraphs, typically some optimization is performed. Optimizing the circuit can improve area, performance, and testability. Each of the cells in the top row and the left column can be replaced simply with AND gates since the carry input and sum input for each of these cells are both always low. Likewise, since the second row receives only a sum input and no carry input from the first row, these cells can also be simplifed.

The multiplier shown in Figure 14 is not fully testable for dynamic faults due to the large amount of reconvergence. Consider output $P2$ as an example. In the A-ADD cell that generates $P2$, there is reconvergence of inputs $X0$, $X1$, $Y0$, and $Y1$. This reconvergence results in only 24 out of the 48 paths passing through output $P2$ being robustly testable. Figure 16 shows the combined logic for output $P2$. Gates 1–3 generate the incoming sum to the cell,



(a) A-ADD cell (b) A-ADD cell implementation

FIGURE 15 Carry save multiplier basic cell.

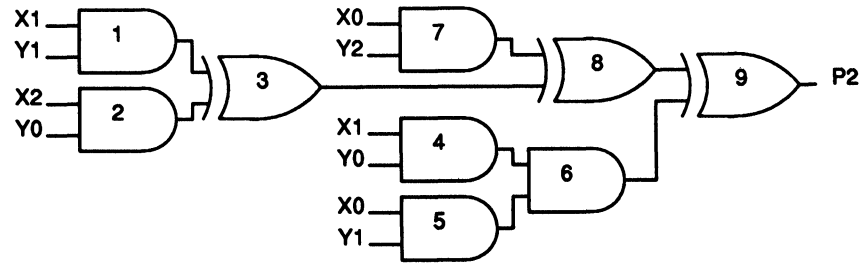


FIGURE 16 Combined logic for $P2$ output of carry save multiplier.

gates 4–6 generate the carry input to the cell, and gates 7–9 are in the A-ADD cell that outputs $P2$. To illustrate the untestability caused by reconvergence, consider the two paths originating at input $X1$ and passing through gates 4, 6, and 9. In order to propagate a transition on input $X1$ to the output of gate 6 requires that $Y0 = 1$, $X0 = 1$, and $Y1 = 1$. When these three inputs are all high, a transition on input $X1$ will also propagate through gates 1, 3, and 8, and a blockage will occur at the input of gate 9. Thus these two paths from input $X1$ to output $P2$ through gates 4, 6, and 9 are not robustly testable. Testability analysis using our program indicates that similar reconvergence causes an additional 22 paths through the circuit of Figure 16 to be untestable.

Modifications to the carry save parallel multiplier could be made to achieve a fully robustly path-delay-fault testable circuit. However, doing so would result in losing the regular structure of the design, and an arbitrarily large multiplier could not be constructed simply by connecting up a number of basic building block cells. A large area penalty would also occur since much of the sharing of logic would also be lost. An alternate approach to building a fully testable multiplier by using pipelining which maintains the regular structure is described in the following paragraphs.

A pipelined version of the 4-bit by 4-bit carry save multiplier is shown in Figure 17. This multiplier incorporates some optimizations, and has three types of basic cells. The cell A-ADD is the same as was shown in Figure 15, the AND cell is simply a two input AND gate, and the A-ADD0 cel is a AND gate and simplified adder (using the fact that $CI = 0$). Typically large multipliers are implemented in a pipelined manner to increase the total throughput. Pipelining the multiplier also increases the testability by reducing the amount of reconvergence in each combinational logic block between sets of registers. In the case of the multiplier in Figure 17 in which each section is composed of two rows, reconvergence is limited to only two signals in each of the A-ADD cells at the bottom of each stage. There is no reconvergence in the first stage of the multiplier, or in the final adder stage.

The reconvergence of the signals in the A-ADD cells at the bottom of each stage does result in some untestable paths. Consider output $P3$ as an example. In the A-ADD cell that generates $P3$, there is reconvergence of inputs $X0$ and $Y2$. This reconvergence results in 2 of the 128 paths passing through this cell being untestable. All 80 of the paths through output $P3$ are robustly testable, but 2 of the 48 paths through the carry output of this cell are not robustly testable. Figure 18 shows the combined logic for the outputs of this cell. Gates 1–3 generate the incoming sum to the cell, gates 4–8 generate the carry input to the cell, and gates 7–15 are in the A-ADD cell that outputs $P3$. The two untestable paths are given by the $\{X0, 4, 6, 8, 12, 15\}$ and $\{X0, 4, 7, 8, 12, 15\}$. These paths are blocked at gate 12 since the other input to that gate is dependent on $X0$ and cannot be forced to a non-controlling value (high) when $X0$ transitions.

In order to eliminate the blockage described in the preceding paragraph, the logic for the carry output can be restructured. In the A-ADD cell (Figure 15), the carry output is implemented in sum of products form, $CO = A \cdot B + A \cdot CI + B \cdot CI$. Since the untestable paths originate on the carry input, the circuit can be made fully testable by implementing the carry output by factoring out the carry input. That is, implement the carry output as $CO = A \cdot B + CI \cdot (A + B)$. This modification to the basic carry save multiplier cell is shown in Figure 19.

Figure 20 shows the combined logic for the outputs of the cell containing output $P3$ when the cell in Figure 19 is used in place of the A-ADD cell. Using this implementation makes both outputs of this cell fully testable for dynamic faults. It also makes the entire pipelined multiplier shown in Figure 17 fully testable. It should be noted that the modified A-ADD cell only needs to be used in the second row of each stage to make it fully testable. However,

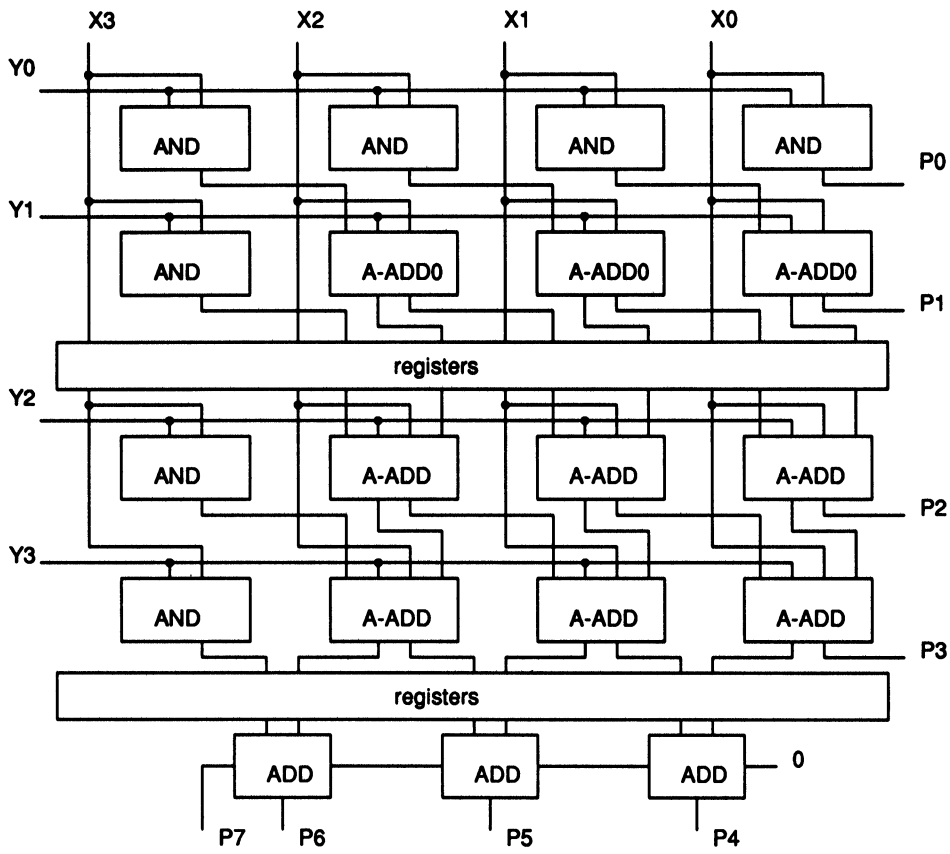


FIGURE 17 Pipelined carry save parallel multiplier implementation.

using it in both rows does not destroy the testability, and keeps the cells uniform.

The problem with the approach used in this section to generate a fully testable pipelined multiplier is that each stage in the multiplier cannot be scaled to an arbitrary depth. Each stage can be arbitrarily wide, but can be at most 2 bits deep. A fully testable multiplier of arbitrary size can of course be created out of 1 and 2 bit stages. The approach used in the case of the ALU circuits could be utilized by duplicating some or all of the latches generating the X and Y inputs to the multiplier array. The optimal approach to utilizing this approach remains to be investigated.

The cell of Figure 19 can be used to produce an $n \times m$ multiplier (that is not necessarily pipelined) that is completely gate-delay-fault and stuck-open fault

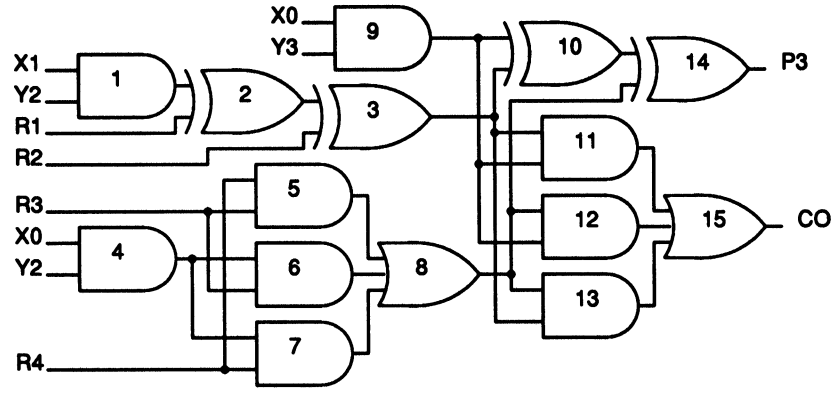


FIGURE 18 Combined logic for $P3$ output of pipelined carry save multiplier.

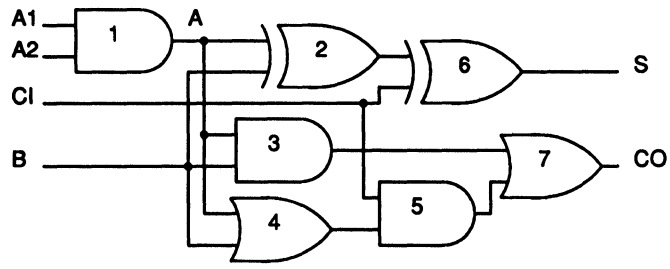


FIGURE 19 Modified carry save multiplier cell.

testable. Figure 14 suggests the following two-dimensional composition rule.

Composition Rule: *Given a set of robustly path-delay-fault testable circuits,* C_{11}, $\cdots$ C_{N1} $\cdots$ C_{1M}, $\cdots$ C_{NM} *such that for* $1 \leq i < N$, $1 \leq j < M$, *a single output* l_{ij} *of* C_{ij} *feeds* C_{i+1j}, *a single output* m_{ij} *of* C_{ij} *feeds* C_{i+1j+1}, *there exists a common input* X_x *to each* C_{xj}, $1 \leq j \leq M$ *and a common input* Y_y *to each* C_{iy}, $1 \leq i \leq N$, *and a transition on* X_i/Y_j *can be damped by* Y_j/X_i *before convergence with inputs from previous cells, then the composition is fully stuck-open fault and gate-delay-fault testable.*

Transitions through links and gates in C_{ij} are first propagated out to l_{ij} or m_{ij}. In the former case, if the transition began from X_i, it can be robustly propagated down the array to an output, without ever encountering any other transition, since the Y_k, $k > j$ can damp the X_i transition. If the transition began from Y_j, it can be propagated downward as before. In the latter case, the transition is propagated to the C_{i+1j+1} block, and then downward without encountering either X_i or Y_j transitions.

For a discussion of the testability of carry propagate multipliers, see Bryan [2].

CONCLUSIONS

Few commonly used designs for regular structures are completely robustly testable for path and gate delay or transistor stuck-open faults. However, in a vast majority of the cases, we have developed modified designs with good area and performance characteristics that are scalable to an arbitrary number of bits, and which are completely path-delay-fault testable. In some cases where complete robust path-delay-fault testability is not achievable, we showed that completely stuck-open or gate-delay-fault testable circuits could be designed. In summary, we have, in Bryan [2]:

- shown that minor perturbations in existing designs of ripple and carry lookahead adders that are not fully path-delay-fault testable can result in completely testable circuits. These circuits retain testability when extended to any number of bits.

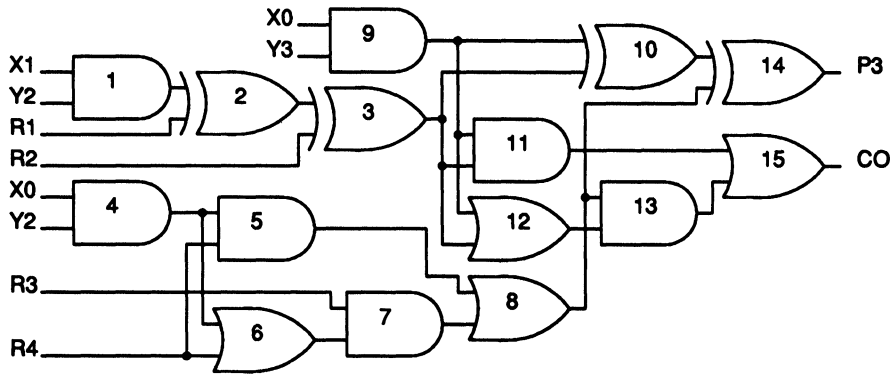


FIGURE 20 Modified logic for *P*3 output of pipelined carry save multiplier.

- shown that a carry select adder of up to 4 bits can be designed to be completely path-delay-fault testable with negligible overhead and 4-bit blocks can be replicated to form arbitrarily large, completely testable adders.
- shown that a carry bypass adder can be made fully path-delay-fault testable and extensible for any number of bits with negligible area and no performance overhead.
- designed a ripple comparator that is completely path-delay-fault testable and extensible to an arbitrary number of bits.
- developed two parallel comparator designs, the first of which is completely path-delay-fault testable, has negligible performance overhead, but a significant area overhead. The second has comparable area and performance characteristics to the traditional parallel comparator design, and is not completely path-delay-fault testable, but is fully gate-delay-fault and stuck-open fault testable.
- analyzed various realizations of parity generators and ALUs for dynamic fault testability.
- designed a completely path-delay-fault testable $n \times 2$ parallel multiplier, for arbitrary n, and a completely gate-delay-fault and stuck-open fault testable $n \times m$ parallel multiplier, for arbitrary n and m.

In the process of design modification to produce fully testable structures, we have derived a number of new composition rules that maintain robust testability in dynamic fault models. These composition rules can be used to analyze and design other regular structures for robust dynamic testability and to compose regular structures with control sections to create register-bounded subcircuits that are robustly testable for all dynamic faults.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825, and in part by a NSF Young Investigator Award with matching funds from IBM and Mitsubishi Corp.

References

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

[2] M.J. Bryan, Synthesis Procedures to Preserve Testability of Multilevel Combinational Logic Circuits, M.S. thesis, Massachusetts Institute of Technology, June 1990.

[3] S. Devadas and K. Keutzer, "Necessary and Sufficient Conditions for Robust Delay-Fault-Testability of Logic Circuits," *Sixth MIT Conference on Advanced Research on VLSI,* 221–238, April 1990.

[4] S. Devadas and K. Keutzer, "Synthesis and Optimization Procedures for Robustly Delay-Fault Testable Logic Circuits," *Proceedings of the 27th Design Automation Conference,* 221–227, June 1990.

[5] K. Keutzer, S. Malik, and A. Saldanha, "Is Redundancy Necessary to Reduce Delay?" *IEEE Transactions on Computer-Aided Design,* 427–435, April 1991.

[6] S. Kundu and S.M. Reddy, "On the Design of Robust Testable CMOS Combinational Logic Circuits," *Proceedings of the Fault Tolerant Computing Symposium,* 220–225, 1988.

[7] S. Kundu, S.M. Reddy, and N.K. Jha, "On the Design of Robust Multiple Fault Testable CMOS Combinational Logic Circuits," *Proceedings of the Int'l Conference on Computer-Aided Design,* 240–243, November 1988.

[8] R. Montoye, Optimization and Testing of nMOS Arithmetic Structures, Ph.D. thesis, University of Illinois, 1983.

Biographies

MICHAEL BRYAN received a B.S. degree in Electrical Engineering from Oregon State University and an M.S. degree in Electrical Engineering from the Massachusetts Institute of Technology. He worked for Hughes Aircraft Company, and is currently employed by Intel Corporation. His research interests include synthesis of fully testable circuits.

SRINIVAS DEVADAS received a B. Tech in Electrical Engineering from the Indian Institute of Technology, Madras, and an M.S. and Ph.D. in Electrical Engineering from the University of California, Berkeley. He is an Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge, and held the Analog Devices Career Development Chair of Electrical Engineering. Dr. Devadas research interests are in the area of Computer-Aided Design of VLSI circuits and systems, with emphasis on logic sysnthesis, formal verification, test generation, synthesis for testability and automated layout techniques. He is a member of IEEE and ACM.

KURT KEUTZER received his B.S. degree in Mathematics from Maharishi International University, and his M.S. and Ph.D. degrees in Computer Science from Indiana University. He is employed by AT&T Bell Laboratories, where he has worked to apply various computer-science disciplines to practical problems in computer-aided design. Dr. Keutzer is an Associate Editor of IEEE Transactions on Computer-Aided Design of Integrated Circuits and serves on a number of technical problem committees.