Synthesis of Robust Delay-Fault Testable Circuits: Practice

Srinivas Devadas
Department of EECS
MIT, Cambridge

Kurt Keutzer
Synopsys
Palo Alto

Abstract

Because of increases in levels of integration, reliability and quality are growing in importance for many integrated circuit designs. Moreover, for highly critical integrated circuits, comprehensive manufacture testing is pre-eminently important. High fault coverages under more comprehensive fault models, such as multiple stuck-at fault and delay fault models, is one way to improve defect coverages of chips produced by highly complex fabrication processes.

The problem of delay-fault testing is difficult for two reasons. Firstly, current design for testability methodologies are known to not result in integrated circuits that are highly robustly path-delay-fault testable. Secondly, the number of possible faults under this model is exponentially greater than for the traditional single stuck-at fault model. Thus a classic test generation/fault simulation approach is often not feasible for gaining delay-fault coverage. In this paper we solve the first problem with a constructive synthesis approach to delay-fault testing, wherein completely path-delay-fault testable circuits are automatically synthesized. We address the second problem by giving optimization and composition procedures that generate the test vectors for the requisite coverage as a by-product of the synthesis process.

In this paper we show how a sophisticated orchestration of combinational synthesis for testability approaches can result in logic-level implementations of large integrated circuit designs that are *completely robustly gate-delay-fault and path-delay-fault testable*. For control portions of VLSI circuits, we use Boolean covering and algebraic factorization procedures that guarantee path-delay-fault testability, starting from a sum-of-products representation of a function. We use *hierarchical composition rules* in the synthesis of regular structures occurring in datapath portions, like parity generators and arithmetic units. Finally, we show how test vectors to detect all path delay faults can be obtained as a by-product of the synthesis process. We have successfully used these techniques on circuits with over 5000 gates. We present preliminary experimental results on a data encryption chip, a small μ-processor and a speech recognition chip.

1 Introduction

Because of increases in levels of integration, reliability and quality are growing in importance for many integrated circuit designs. Moreover, for highly critical integrated circuits, comprehensive manufacture testing is pre-eminently important. High fault coverages under more enhanced fault models, such as multiple stuck-at fault (multifault) and delay fault models, is one way to improve defect coverages of chips produced by highly complex fabrication processes.

Synthesis for testability presents a radically different approach to logic testing in which testability considerations are incorporated into the logic optimization process. In this paper we advocate a synthesis approach to delay-fault testing wherein completely robustly gate-delay-fault and path-delay-fault testable (`RPDFT`) circuits are automatically synthesized, and test vectors are generated as a by-product of the synthesis process.

In this paper we apply recently developed necessary and sufficient conditions for robust path and gate-delay-fault testability [**?**, **?**] to develop synthesis procedures which produce circuits with high levels of testability under these fault models. We show how a sophisticated orchestration of combinational synthesis for testability approaches can result in logic-level implementations of large integrated circuit designs that are *completely* `RPDFT`.

For control portions of VLSI circuits, we develop synthesis procedures based on Boolean covering and algebraic factorization that guarantee path-delay-fault or gate-delay-fault testability, starting from a sum-of-products representation of a function. We show how *hierarchical composition rules* can be used in the synthesis of regular structures occurring in datapath portions, like parity generators and arithmetic units. We analyze area and performance penalties associated with our synthesis techniques. Finally, we show how test vectors to detect all path-delay-faults can be obtained as a by-product of the synthesis process.

In our initial experiments we have successfully used these techniques on circuits with over 5000 gates, and we present preliminary experimental results on a data encryption chip, a small μ-processor and a speech recognition chip.

In Section 2, we address concerns associated with testing in non-traditional fault models. In Section 3, we present some basic definitions and terminology used. In Section 4, we review theoretical results presented in [**?**, **?**] pertaining to delay-fault testability. We also review previous work in the area of synthesis for delay-fault testability.

In Section 5, we focus on the synthesis of two-level sum-of-product implementations that are robustly testable for path and gate delay faults. We present a Boolean covering procedure that optimizes for delay-fault testability, and give experimental results using the covering procedure.

In Section 6, we use the notion of *ENF reducibility* to show that algebraic factorization with a constrained use of the complement retains `RPDFT`, beginning from a two-level or multilevel network. This directly results in an efficient synthesis procedure for multilevel `RPDFT` circuits. We analyze other multilevel logic restructuring operations from the point of view of retaining delay-fault testability. We show how test vectors to detect all delay faults can be obtained as a by-product of the factorization and technology mapping steps and finally present experimental results using the factorization procedures.

In Section 7, we switch focus to the synthesis of functions that do not have manageable sum-of-product representations, in particular, regular structures like arithmetic units and parity generators, which are common in datapath circuits. We prove a hierarchical composition rule for the hierarchical testability of regular structures, and apply it to a simple example. We show how a data encryption chip, a small μ-processor and a word-processing chip for a speech recognition system can be synthesized for complete path-delay-fault testability beginning from a register-transfer-level (RTL) description. In each case, we indicate how test vectors to detect these faults can be obtained as a by-product of synthesis.

2 Problems with Obtaining High Fault Coverage in Enhanced Fault Models

The problem of multiple stuck-at fault (multifault) and robust path-delay-fault testing is difficult for a number of reasons, and despite the long research interest in non-stuck-at fault-models these difficulties have caused some members of the testing community to consider these problems to be of only academic (literally and figuratively) interest. In this section we attempt to summarize commonly stated problems, and insofar as is possible at this stage of our research, address them.

Firstly, current design for testability methodologies are known to result in integrated circuits that are not highly robustly path-delay-fault testable. In this paper we solve the problem of untestable faults with a constructive synthesis approach to delay-fault testing, wherein completely robustly path-delay-fault testable circuits are automatically synthesized.

Another problem is that the number of possible faults under the path delay fault model is exponentially greater than for the traditional single stuck-at fault model. Thus, a classic test generation/fault simulation approach to obtaining test vectors is often not feasible for path delay faults. We address this problem by giving optimization and composition procedures that generate the test vectors for complete coverage as a by-product of the synthesis process. Applying a set of vectors for complete RPDFT is feasible for many sub-circuits, but for some sub-circuits even though we can constructively generate complete test vector sets, the sub-circuit may have too many paths for each to be individually tested. A subset of critical paths may be chosen for testing in these sub-circuits, in addition to comprehensively testing sub-circuit in a weaker fault model as described in Section 7.9.

It may be questioned that if we do not test for all path delay faults what is the use of achieving a fully RPDFT design? As discussed in Section 7.9 RPDFT is such a comprehensive testability requirement that it implies testability in most other common fault models. Furthermore, RPDFT is invariant over a number of area-reducing (see Section 6) and delay-reducing [?] transformations. This means that once a design has been synthesized to be RPDFT a large class of area and timing optimization procedures may be applied to it without diminishing the testability of the final design. If these same transformations were applied to an initial design that was only single-stuck-at fault testable then single-stuck-at fault redundancies could be *introduced* into the final design, because of redundant multiple stuck-at faults in the initial design.

It has been noted that arbitrary input vector pairs, $< v_1, v_2 >$, cannot be applied to a combinational circuit using standard scan path design [?]. Several solutions have been proposed. In [?], an extra latch to each existing latch was added to store v_1, while v_2 was being shifted in. Adding this extra circuitry could significantly increase circuit size relative to normal scan structures. Another solution is to obtain v_2 by shifting v_1 [?]. However, this limits the number of vector pairs that can be applied. In [?], an enhanced latch that allows the application of arbitrary vector pairs was proposed for nMOS technologies. Another alternative is to scan in v_1 from the block under test while propagating v_2 forward from the previous logic block [?]. Unfortunately, this potentially results in a non-robust application, but the fact that the combinational logic is RPDFT means that the sources of test invalidation are considerably restricted.

These scan structures can be significantly more expensive than traditional scan structures. We assume that for some circuit applications quality and reliability considerations will be pre-eminent and therefore the additional cost of the specialized scan structure will be worthwhile. For other circuits even if the path delay faults are not tested robustly, then at least by synthesizing a fully RPDFT design the sources of test invalidation are limited quite significantly.

Ideally, a non-scan circuit that is highly or fully RPDFT should be synthesized and this is one subject of current research.

3 Definitions and Notation

Analysis and synthesis of delay-fault testable circuits requires spanning the fields of logic optimization, testing and timing analysis. In this section we try to provide a minimal amount of terminology from these fields.

3.1 Logic Synthesis

A **Boolean function** F of N variables is a mapping from $B^N = \{0,1\}^N \rightarrow \{0,1\}$. We model B^N as a Binary N-cube. A vertex $v \in B^N$ for which $F(v) = 1$ is a member of the ON-set. If $F(v) = 0$ then v is a member of the OFF-set.

A **literal** is a Boolean variable or its complement. We formally define a **cube** as a set of literals and we interpret the cube as a product of literals. For example $\{a, b, c\}$ is a cube, interpreted as $a \cdot b \cdot c$ which may be abbreviated to abc. A **minterm** is a cube in which every variable in the Boolean function appears. We may interpret the minterm as a **vertex** in the N-cube.

Minterms and cubes may be used to represent the values of a set of input variables: *e.g.* $x\overline{y}z$ is shorthand for $x = 1,\ y = 0$, and $z = 1$. In this way there is a natural correspondence between an input vector or input stimulus, a minterm and a vertex in the N-cube. This correspondence may be extended to cubes where unspecified values in the function are assumed to be undefined values. Thus, if a circuit C has inputs $v,\ w,\ x,\ y,\ z$ then applying the cube $x\overline{y}z$ to C is shorthand for applying $v = X,\ w = X,\ x = 1,\ y = 0$, and $z = 1$. (Here X denotes an unknown value.) We define the distance δ between a cube q and a cube r as the cardinality of the set: $\{l | (l \in q) \wedge (\overline{l} \in r)\}$. For example, the distance between abc and $ab\overline{c}$ is 1; the distance between ab and abc is 0; and the distance between $b\overline{c}def$ and $abcf$ is 1.

Following the historical usage we define a **cover** as a set of cubes and we interpret the cover as a sum-of-products expression. For example $\{\{a, b, c\}, \{d, e, f\}\}$ is a cover, interpreted as $abc + def$. We say that a cube q **covers** a cube (or vertex) r if $q \subseteq r$. If a cube q covers only ON-set vertices of a Boolean function F then q is called an **implicant** of F. A **relatively essential vertex** of a cube q in a cover C is a vertex that is covered by q and is not contained in any other cube in C.

A **Boolean Network** η is a directed-acyclic graph (DAG) such that for each node n_i in η there is an associated cover F_i, and a Boolean variable y_i representing the output of F_i. As there is a one-to-one correspondence between a node and its cover we will use the terms interchangeably. There is a directed edge from n_i to n_j iff F_j explicitly depends on y_i or $\overline{y}_i$. Furthermore, some of the variables in η may be classified as *primary inputs* or *primary outputs*. We may directly implement a Boolean Network η by a combinational circuit $\mathcal{C}$ by replacing each cover F_i in η by a NAND/NAND network and by replacing each edge set fanning out from a node in η by a **net**.

3.2 Testing

A Combinational Logic Circuit is represented as a labelled, directed, acyclic graph (DAG) $G = (V, E)$ with each vertex v labelled with the name of a primitive gate such as AND, OR or NOT, or with the name of a primary input or output. In general, a combinational logic circuit can be created from gates of arbitrary complexity, but to simplify the discussion we assume that the circuit is expressed in terms of primitive gates. There is an edge $< u,\ v >$ in V between two vertices if the output of the gate associated with u is an input to gate v. The members of V that have no fan-in are the only vertices that may be labelled with the name of a primary input. The members of V that have no fan-out are the only vertices that may be labelled with the name of a primary output.

If the output of a gate, g_1, is connected to an input of gate, g_2, g_1 is a **fanin** of g_2. Gate g_2 is a **fanout** of gate g_1.

A **two-level circuit** is a circuit directly implementing a *cover* by implementing each product term by an AND gate, the sum term by an OR gate adding inverters on inputs as needed. We consider only non-trivial two-level circuits in which no AND gate is fed both by a signal and its complement (*e.g.* $a \cdot \overline{a}$).

A single-output combinational logic circuit computes a Boolean function in the obvious way. An N input M output combinational logic circuit computes a vector Boolean function from $B^N \rightarrow B^M$. As we will be dealing primarily with combinational logic circuits in this paper, we will use circuit to mean combinational logic circuit.

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. A circuit has a **single-stuck-at-fault** if there is one stuck-at-fault in the circuit. A circuit has a **multiple-stuck-at-fault** (multifault) if there are one or more stuck-at-faults in the circuit.

A circuit C is **completely single-stuck-at-fault testable** if for each stuck-at fault f there exists an input stimulus t such that the output of the circuit on applying t in the presence of f is different from the output of C without the fault f. **Completely multiple-stuck-at-fault-testable** can be defined similarly.

If a single fault is stuck-at-1 untestable then the edge (net) in the circuit associated with that fault can be replaced by a constant 1. Similarly if a single fault is stuck-at-0 untestable then the edge (net) in the circuit associated with that fault can be replaced by a constant 0.

For a fuller treatment of testing terminology, see [?] [?].

3.3 Timing/Delay Testing

In this section we introduce terminology that will allow us to discuss timing issues as well as fault models for temporal behavior. We will introduce a lot of terminology, but hopefully the resultant clarity regarding the concepts will be worth the price. A **path** in a combinational circuit is an alternating sequence of vertices and edges, $\{g_0, e_0, ..., g_n, e_n, g_{n+1}\}$, where edge e_i, $1 \leq i \leq n$, connects the output of vertex g_i to an input of vertex g_{i+1}. For $1 \leq i \leq n$, g_i is a gate; g_0 is a primary input and g_{n+1} is a primary output. Each e_i is a net.

An **event** is a transition $0 \rightarrow 1$ or $1 \rightarrow 0$ at a gate. Consider a sequence of events, $\{r_0,\ r_1,\ ...,\ r_n\}$ occurring at gates $\{g_0,\ g_1,\ ...,\ g_n\}$ along a path, such that r_i occurs as a result of event r_{i-1}. The event r_0 is said to **propagate** along the path. If there exists an vector pair such that under *appropriate* delays in the circuit, an event could propagate along a path, then the path is said to be **event sensitizable**. If there exists an input vector pair such that under *arbitrary* delays in the circuit, an event propagates along a path, then the path is said to be **robustly event sensitizable**.

A **controlling value** at a gate input is the value that determines the value at the output of the gate independent of the other inputs. For example, 0 is a controlling value for an AND gate. A **non-controlling value** at a gate input is the value which is not a controlling value for the gate. For example, 1 is a non-controlling value for an AND gate. We say a gate g has a **controlled value** if one of its inputs has a controlling value; otherwise, we say g has a **non-controlled value**.

A circuit has a **gate delay fault** iff there is one gate in the circuit such that the output of the circuit is slow to make a $0 \rightarrow 1$ (or $1 \rightarrow 0$) 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** iff there exists a path from a primary input to a primary output via a set of gates and nets such that a primary input event is slow to propagate *along the path* to the primary output.

A **robust test** for a path delay or a gate 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 delay fault in a circuit C is a vector pair $< v_1,\ v_2 >$ 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 $< v_2, v_1 >$ is applied to propagate the opposite event along the path and detect faults corresponding to the $1 \rightarrow 0$ event. Note that a vector sequence is a robust path delay fault test for a path π iff it *robustly event sensitizes* π.

4 Review of Relevant Theory

To explain how individual modules may be synthesized to be `RPDFT`, we first review some theoretical results presented in [?, ?].

Theorem 4.1 *: Let C be a two-level single-output circuit. Let π be a path in C that starts with l in cube q. There exists a delay-fault test for π in C iff there exists a vertex v_2 that is a relatively essential vertex of q and a vertex $v_1 = v_2 - \{l\} \cup \{\bar{l}\}$ such that v_1 is in the OFF-set of C.*

A similar result for transistor stuck-open faults was presented in [?]. This result was generalized to multilevel circuits [?] using the Equivalent Normal Form (ENF) originally developed by Armstrong [?] (*c.f.* Section 6.1).

A multilevel logic synthesis procedure for delay-fault testable circuits was proposed in [?, ?] based on algebraic factorization (without the use of the complement).

Theorem 4.2 *: Let C be a two-level multiple-output circuit that is completely robustly path-delay-fault testable. Let A be a multi-level circuit resulting from an algebraic factorization of C. Then A is robustly path-delay-fault testable.*

In [?, ?] it was shown that algebraic factorization retains multifault testability as well. These results have important applications to multilevel logic synthesis for testability, since state-of-the-art multilevel logic optimizers that target area and performance improvement (*e.g.* MIS [?]) use predominantly algebraic techniques to restructure logic equations. The unate decomposition and "tree" factoring techniques presented in [?, ?] have limited applicability in the area optimization of multilevel logic. The former can *increase* the area of a multilevel circuit.

An example of algebraic factorization dramatically changing the structure of a network, while reducing area and maintaining testability, is the transformation of the sum-of-products logic function $ace + ade + acf + adf + bce + bde + bcf + bdf$ to $(a+b)(c+d)(e+f)$. The number of literals has been reduced from 24 to 6.

5 Synthesis of Two-Level Circuits for Delay-Fault Testability

5.1 Introduction

In Section 4, a necessary and sufficient condition for robust path-delay fault testability of two-level circuits was given. While the conditions named implied primality and irredundancy, we shall show that primality and irredundancy is not a sufficient condition for robust path-delay fault testability of two-level circuits. Thus a two-level minimization procedure particularly tuned to optimize for robust path-delay fault testability is required. We begin with the observation that different prime and irredundant two-level implementations of logic functions can have different path-delay-fault testabilities. Consider the prime and irredundant function shown in Figure 1(a). The literal x in the cube $x.\overline{y}$ is not robustly testable to the output f. A different prime and irredundant implementation of the same logic function is shown in Figure 1(b). This implementation is completely robustly path-delay-fault testable.

The prime and irredundant network of Figure 1(b) satisfies the conditions given by Theorem 4.1, whereas the prime and irredundant network of Figure 1(a) does not. Thus, selecting an appropriate set of primes can enhance the path-delay-fault testability of a logic function. In the following sections, we will describe exact and heuristic covering algorithms that find a two-level implementation of logic maximizing path-delay-fault testability. There are some similarities between our heuristic covering procedure and a heuristic covering procedure for transistor stuck-open fault testability presented in [?].

5.2 Exact Covering

In the well-known Quine-McCluskey procedure for two-level Boolean minimization, primes are generated from the specification of a Boolean function and a covering step is performed wherein a minimum set of primes that form a cover for the function is selected. In our procedure the prime implicant generation step is identical, and we use a variant of the branch-and-bound strategy for the covering step. During covering we select a minimum set of primes attempting to *maximize the number of* *relatively essential vertices in the primes that are distance-1 from the OFF-set* of the function.

The procedure below receives as input the prime implicant table, T, corresponding to a single-output logic function. It searches for the most testable, minimum-product-term realization of a Boolean function.

1. Delete primes in T that are dominated by other primes, *i.e.* if a prime p covers all the minterms covered by another prime q in the current table T, delete prime p. Similarly, delete any minterm which is covered by a set of primes that is a superset of the set of primes covering any other minterm. Detect essential primes by checking to see if any minterm is covered by a single prime. Add these essential primes to the selected set. Repeat until no new essential elements are detected.

2. If the size of the selected set *plus* the lower bound calculated using a maximal independent set heuristic [**?**] exceeds the best solution thus far, return from this level of recursion. Else if the size of the selected set *plus* the lower bound equals the best solution thus far, then if the number of untestable literals/paths [1] is greater than that of the best solution, return from this level of recursion. If there are no elements left to be covered, declare the selected set as the best solution recorded thus far.

3. Heuristically select a branching prime. Find the essential vertices of all the primes in the selected set. This is done by using the disjoint SHARP operation implemented in ESPRESSO [**?**]. Search for an unselected prime that minimally impacts (*i.e.* does not destroy) these essential vertices and which itself contains essential vertices that are distance-1 from the OFF-set in a maximum number of constituent literals. The current implicant table T (the table with the selected primes and covered minterms removed) gives us the information regarding the essential vertices of the next prime that we select.

4. Add this prime to the selected set and recur for the sub-table resulting from deleting the prime and all minterms that are covered by this prime. Then, recur for the sub-table resulting from deleting this prime without adding if to the selected set.

In the above procedure, we have explored all possible minimum covers of a function, and selected the minimum cover with the least number of untestable literals/paths. The procedure can easily be

[1] A literal in a cover has satisfy the conditions of Theorem 4.1 to be path-delay-fault testable.

generalized to the multiple-output case. In the multiple-output case, we have two possible alternatives for generating primes.

- Generate primes for the multiple-output function and perform covering as described above. In this case, a minimum-product-term solution will result, but since literals in the selected multiple-output primes may not be prime for *every* output, a large number of literals/paths may be untestable.

- Generate primes for each of the single-output functions and merge primes with identical input parts. Thus, the primality and irredundancy requirements for robust path-delay-fault testability of a multiple-output function (*c.f.* Theorem 4.1) will be satisfied for any selected cover. The covering procedure can be used to find a minimum-product-term implementation, *under this smaller set of primes*, that is maximally path-delay-fault testable. The testability of the implementation produced by this procedure is typically significantly higher than by the previous procedure, but the implementation may require a larger number of product terms.

5.3 Heuristic Covering to Maximize Testability

The exact covering algorithm described above is viable for medium-sized functions with upto 15 inputs. However, for larger functions, the number of primes in the function may be too large, or more often, the branch-and-bound step, which explores all minimum covers, may require too much CPU time. Hence, an efficient, heuristic strategy is desirable.

We have developed a heuristic strategy that (unlike the exact covering algorithm) targets the delay testability of the selected prime and irredundant cover as a primary objective and the number of product terms in the cover as a secondary objective. The delay testability of the cover is, as before, measured as the inverse number of untestable literals/paths in the cover. The number of primes considered are a subset of the total number of primes, if the latter are too numerous. The input to the procedure is the prime implicant table, T, corresponding to the logic function.

1. Same as Step 1 of the exact procedure, except dominated primes in T are *not* deleted, since we are primarily concerned with testability, rather than product term count.

2. If there are no elements left to be covered, quit. Else, select a prime which covers at least one uncovered element, and which when added to the current selected set, results in the minimal number of untestable literals/paths in the selected set (including the prime). In the case of two

primes resulting in the same number of untestable literals/paths, select the prime which covers more uncovered elements.

The above procedure represents a fast, greedy strategy to maximize testability without regard to product term count.

5.4 Maximizing Gate-Delay-Fault Testability

While robust gate-delay-fault testability (`RGDFT`) is a much less stringent testability criterion than `RPDFT`, there exist logic functions that are not completely `RGDFT` in any two-level realization. The example below is due to Touati and Watanabe [?]. Gate 5 is not robustly testable, since the corresponding cube does not contain a relatively essential vertex distance-1 from the OFF-set.

Hence, a covering procedure to maximize `RGDFT` is required. The necessary and sufficient condition given in Section 4 for `RGDFT` in a two-level network is similar to the `RPDFT` criterion. Maximizing `RGDFT` entails maximizing the number of cubes in a cover that have relatively essential vertices that are distance-1 in some literal from the OFF-set (rather than in each literal as in the case of `RPDFT`). The covering procedure described can be used for maximizing `RGDFT` as well.

5.5 Experimental Results

In this section, we present experimental results using the covering procedures described in the previous sections. We chose several examples from the MCNC Benchmark set, whose statistics are summarized in Table 1.

In Table 2, we present results obtained using the covering procedure. We compare the number of product terms and the testability of prime and irredundant covers using three different strategies. First, each of the functions were optimized using the multiple-output logic minimizer ESPRESSO [?]. Next, each of the outputs in the multiple-output functions were optimized separately. A larger number of product terms are required since each literal in a cube is required to be prime in each output in which the cube appears; however, path-delay-fault testability is drastically improved. Finally, the heuristic covering procedure of Section 5.3 was used to obtain a maximally testable realization. The covering procedure not only produces circuits with smaller areas than in the single-output minimized case, it also produces circuits with higher testabilities in many cases. An appropriate selection of primes satisfies the necessary conditions for path-delay-fault testability of more paths than in the circuits produced by ESPRESSO.

Results on RGDFT are not given, since most functions we have encountered, including all of the functions above, are 100% RGDFT in their minimum sum-of-products form as produced by ESPRESSO (*c.f.* Table 9).

6 Synthesis of Multilevel Circuits for Delay-Fault Testability

6.1 Equivalent Normal Form

We use an alternative representation of a multilevel circuit developed by Armstrong [?] called the *equivalent normal form* or ENF. The ENF of a circuit is a two-level representation in which each path is represented by a cube with each literal in the cube annotated by information regarding its path from the primary input of the circuit to the primary output. We will define ENF more formally below.

Given a combinational circuit C with primary inputs $i_1, \ldots, i_m$, outputs $o_1, \ldots, o_n$ and labelled gates $g_1, \ldots, g_p$, we can construct the ENF for each output o_k of C by traversing the circuit forward from the primary inputs and visiting each gate in topological order. Each variable in an ENF expression is simply a primary input of C, such as i_j, and has a label consisting of a set of gates denoting the path this input has taken to reach the output. The ENF for a primary input i_j is simply i_j. The intermediate ENF expression corresponding to the output of an AND gate g with inputs a_α, b_β is a sum-of-products expression over tagged literals, $E_g = a_{\alpha \cup g} \cdot b_{\beta \cup g}$. The expression E_g is created by reducing the product to sum-of-products form using DeMorgan's laws and distributivity without making any Boolean reductions such as $(a + \overline{a} \equiv 1)$, $(a + a \equiv a)$, $(a \cdot a \equiv a)$ or $(a \cdot \overline{a} \equiv 0)$. Similarly, the intermediate ENF expression corresponding to the output of an OR gate h with inputs a_α, b_β is a sum-of-products expression over tagged literals, $E_h = a_{\alpha \cup h} + b_{\beta \cup h}$. As before, no Boolean reductions are made in reducing E_h to sum-of-products form. The intermediate ENF expression corresponding to the output of an inverter j with the single input a_α is a sum-of-products expression over tagged literals, $E_j = \overline{a}_{\alpha \cup j}$ reduced to sum-of-products form as above. The ENF expression for other gates can be defined in a similar way.

In Figure 3 we a circuit based on Figure 2 in [?]. The final ENF expression for primary output M is:

$$M = e_{\{5,9\}} \cdot b_{\{5,9\}} \cdot \overline{a}_{\{1,4,5,9\}} + b_{\{5,9\}} \cdot \overline{c}_{\{2,3,4,5,9\}} \cdot d_{\{3,4,5,9\}} \cdot e_{\{5,9\}} + a_{\{1,4,6,8,9\}} \cdot c_{\{2,3,4,6,8,9\}} \cdot \overline{b}_{\{7,8,9\}} + a_{\{1,4,6,8,9\}} \cdot c_{\{2,3,4,6,8,9\}} \cdot \overline{d}_{\{7,8,9\}} + a_{\{1,4,6,8,9\}} \cdot \overline{d}_{\{3,4,6,8,9\}} \cdot \overline{b}_{\{7,8,9\}} + a_{\{1,4,6,8,9\}} \cdot \overline{d}_{\{3,4,6,8,9\}} \cdot \overline{d}_{\{7,8,9\}}.$$

6.2 ENF Reducibility

We say that two cubes, c and d, are **syntactically identical**, denoted $c \equiv_s d$ iff they are equal as sets of literals. We say that two sum-of-products expressions, C and D, are **syntactically identical**, denoted $C \equiv_s D$, iff they are equal as sets of cubes. An ENF expression E, considered as an untagged sum-of-products expression is denoted $NOTAGS(E)$. We say that two ENF expressions, E and F, are **syntactically identical up to a renaming of tags**, denoted $E \geq_t F$, iff $NOTAGS(E) \equiv_s NOTAGS(F)$ and there is a many-to-one mapping from the tagged literals of E to the tagged literals of F. We will call this property of ENF equivalence up to a many-to-one mapping of tags **ENF reducibility**.

What we are attempting to capture in the notion of ENF reducibility between circuits C and D, *i.e.* $C \geq_t D$, is the notion of a syntactic equivalence between the unreduced collapsed networks of C and D together with the notion that there is a many-to-one mapping from the paths (tags) of C to the paths (tags) of D.

For example, consider the following ENF expressions: $E = a_{\{1,3\}} \cdot b_{\{1,3\}} + a_{\{2,3\}} \cdot c_{\{1,3\}}$ corresponding to $a \cdot b + a \cdot c$ and $F = a_{\{2\}} \cdot b_{\{1,2\}} + a_{\{2\}} \cdot c_{\{1,2\}}$ corresponding to $a \cdot (b + c)$. $NOTAGS(E) \equiv_s NOTAGS(F)$ and there is a simple many-to-one mapping from the tags of E to the tags of F (remember that a function or mapping is just a set of ordered pairs): $< \{a, 1, 3\}, \{a, 2\} >, < \{a, 2, 3\}, \{a, 2\} >, < \{b, 1, 3\}, \{b, 1, 2\} >, < \{c, 1, 3\}, \{c, 1, 2\} >$. Thus $E \geq_t F$.

Given a circuit C with ENF expression E, and a tagged literal l_π, the **cube complex associated with** l_π is the set of all cubes q such that $l_\pi \in q$ and there is no other tagged literal $l_\rho \in q$.

In [?, ?] a necessary and sufficient condition for a path to be robustly testable in a multilevel circuit was given based on the ENF of the circuit. This theorem is reproduced below.

Theorem 6.1 *: Let C be a multilevel single-output circuit with ENF E. Let π be a path in C and let the tagged literal associated with π be l_π. Let the cube complex of l_π be L and let D be those cubes not containing l_π. There exists a delay-fault test for π in C iff*

1. *there exists a vertex v_2 such that $L(v_2) = 1$ and v_2 is not covered by any cube in D and*
2. *there exists a vertex $v_1 = v_2 - \{l\} \cup \{\bar{l}\}$ such that v_1 is in the OFF-set of C and*
3. *for every cube d in D there exists some literal m in both v_1 and v_2 such that $d_m(v_1) = 0$ and $d_m(v_2) = 0$.*

We now prove the main theorem of this section.

Theorem 6.2 *: Let C be a single-output circuit with ENF expression E_C. Let M be a single-output circuit with ENF expression E_M. If $E_C \geq_t E_M$ and if a vector set V robustly detects all path delay faults in C then V robustly detects all path delay faults in M.*

Proof: Let π be a path in M. There is a corresponding tagged literal l_π in E_M. Corresponding to l_π in E_M there is a non-empty set of corresponding tagged literals $\{l_{\pi_1}, \ldots, l_{\pi_m}\}$ in E_C. Let the cube complex of l_{π_i} in E_C be L and let D be those cubes in E_C not containing any l_{π_i}. By Theorem 5.4 of [?] there exists a robust path-delay-fault test $< v_1,\ v_2 >$ for each π_i which has the following properties:

1. vertex v_2 is such that $L(v_2) = 1$ and v_2 is not covered by any cube in D and
2. vertex $v_1 = v_2 - \{l\} \cup \{\overline{l}\}$ and v_1 is in the OFF-set of C and
3. for every cube d in D there exists some literal m in both v_1 and v_2 such that $d_m(v_1) = 0$ and $d_m(v_2) = 0$.

Let the cube complex of l_π in E_M be L_M and let D_M be those cubes in E_M not containing l_π. Because L_M contains all the cubes in E_M that correspond to those cubes containing l_{π_i} in E_C, $NOTAGS(L) \subseteq NOTAGS(L_M)$, so $L_M(v_2) = 1$ and v_2 is not covered by any cube in D_M. As the OFF-set of E_C and E_M is exactly the same, v_1 is in the OFF-set of E_M. Finally, just as $NOTAGS(L) \subseteq NOTAGS(L_M)$, similarly $NOTAGS(D) \supseteq NOTAGS(D_M)$. Thus, for every cube d in D_M there exists some literal m in both v_1 and v_2 such that $d_m(v_1) = 0$ and $d_m(v_2) = 0$. Thus, by Theorem 5.4 of [?] the vector pair $< v_1,\ v_2 >$ is a robust path-delay-fault test for π in M. Thus, for every path π in M there exists a non-empty set of associated paths P in C such that any vector pair for `RPDF` testing any $\pi_i \in P$ is a `RPDF` test for π. ∎

While this proof shows that ENF reducibility retains `RPDFT` this same argument may also be used to show that algebraic factorization can improve the `RPDFT` of a circuit. For example consider $C = a\overline{b} + \overline{b}c + b\overline{c}$ and its algebraic factorization $M = (a+c)\overline{b} + b\overline{c}$. The path associated with literal $\overline{b}$ in cube $a\overline{b}$ of C, call it π_1, is not `RPDFT`, but the path associated with $\overline{b}$ in cube $\overline{b}c$ of C, call it π_2, is `RPDFT`. After C is factored into M there is a many-to-one reduction from paths π_1 and π_2 to a single path π associated with literal $\overline{b}$ in the factor $(a+c)\overline{b}$, and the testability of π_2 alone is sufficient to ensure the testability of π. As a result M is completely `RPDFT` while C is not.

It should also be noted that ENF reducibility as defined above does not imply the preservation of `RGDFT` (*c.f.* Section 6.7). The conditions for `RGDFT` are weak, and require a stronger form of reducibility.

6.3 ENF Reducibility Preserving Transformations

Now that we have a property, ENF reducibility, that ensures retaining `RPDFT` and its associated vector set, our task is reduced to finding a class of operations that retain ENF reducibility. A first step in this direction was presented in [**?**, **?**] where the operations of cube extraction, kernel extraction and algebraic resubstitution without complement were shown to retain `RPDFT`. The arguments used there appealed implicitly to the property of ENF reducibility.

In this section we show that the operation of constrained *algebraic resubstitution with complement* also retains ENF reducibility. Regarding the other operations used in area optimization, we will consider elimination in Section 6.5.1, Boolean resubstitution in Section 6.5.2 and simplification in Section 6.5.4. We will show that these operations not only do not preserve ENF reducibility, but may be shown to introduce paths that are not robustly path-delay-fault testable. Thus with this paper we will have shown that among the operations commonly used for area optimization in MIS [**?**], all may be proved to retain robust path-delay-fault testability or may explicitly be shown not to.

6.4 Algebraic Resubstitution with Complement

We now explain how to constrain algebraic resubstitution with complement in such a way that ENF reducibility is retained.

Lemma 6.1 *: Let η be a Boolean network of N nodes and for one node k in η let the cover of F_k be G. Let η' be a Boolean network of $N+1$ nodes obtained by pushing an inverter out from F_k. In other words, for $1 \leq i < N+1$, $i \neq k$, $F'_i = F_i$. Let the new node be $F'_{N+1} = \overline{y_k}$ and let $F'_k = \overline{G}$.*

If when the node F'_k in η' is collapsed into node F'_{N+1} the resulting cover is syntactically identical to the node F_k in η, then the ENF expression of η is syntactically identical up to a renaming of tags to the ENF expression of η'.

Proof: In the construction of the ENF as described in Section 6.1 the node F'_k in η' will be collapsed forward into F'_{N+1}. If the cover resulting from this collapsing is syntactically identical to the node F_k in η, then the intermediate ENF's will also be syntactically identical up to a renaming of tags. The only difference between the ENF's will be that paths through node F'_k in η' will have an additional tag for the inverter at node F'_{N+1} that paths through node F_k in η will not. As η and η' are identical in every other respect, the ENF of η is syntactically identical up to a renaming of tags to the ENF of η'. ■

Theorem 6.3 : *Let C be a single-output circuit with ENF expression E_C. Let M be C after the resubstitution of a single complemented algebraic factor as constrained by Lemma 6.1. Then $E_C \geq_t E_M$.*

Proof: Let M_1 be C after moving an inverter up in C. From Lemma 6.1 we know that $E_C \geq_t E_{M_1}$. Let M be M_1 after a single instance of algebraic resubstitution without complement. The arguments in Theorem 4.2 (from [?, ?]) show that $E_{M_1} \geq_t E_M$. It is easy to show that the relation $\geq_t$ is transitive, so $E_C \geq_t E_M$. ■

For example, consider the sum-of-products expression $C = a\overline{b}d\overline{e} + cd\overline{e} + a\overline{b}f + cf + \overline{a}\,\overline{c}e + \overline{a}\,\overline{c}g + b\overline{c}e + b\overline{c}g$. Let $B = (c + a\overline{b})(d\overline{e} + f) + (\overline{a}\,\overline{c} + b\overline{c})(e + g)$. Let F_k be $(\overline{a}\,\overline{c} + b\overline{c})$. Moving an inverter up gives : $F'_k = (a\,\overline{b} + c)$ and $F'_{N+1} = \overline{y_k}$. The node F'_k can now be algebraically resubstituted without complement into the other instance of $(a\,\overline{b}+c)$ in B. The resulting circuit, call it A, is given in Figure 4. The ENF of A is $E_A = a_{\{3,4,10,11\}} \cdot \overline{b}_{\{1,3,4,10,11\}} \cdot d_{\{8,9,10,11\}} \cdot \overline{e}_{\{2,8,9,10,11\}} + c_{\{4,10,11\}} \cdot d_{\{8,9,10,11\}} \cdot \overline{e}_{\{2,8,9,10,11\}} + a_{\{3,4,10,11\}} \cdot \overline{b}_{\{1,3,4,0,11\}} \cdot f_{\{9,10,11\}} + c_{\{4,10,11\}} \cdot f_{\{9,10,11\}} + \overline{a}_{\{1,3,4,6,7,11\}} \cdot \overline{c}_{\{4,6,7,11\}} \cdot e_{\{5,7,11\}} + \overline{a}_{\{3,4,6,7,11\}} \cdot \overline{c}_{\{4,6,7,11\}} \cdot g_{\{5,7,11\}} + b_{\{1,3,4,6,7,11\}} \cdot \overline{c}_{\{4,6,7,11\}} \cdot e_{\{5,7,11\}} + b_{\{1,3,4,6,7,11\}} \cdot \overline{c}_{\{4,6,7,11\}} \cdot g_{\{5,7,11\}}$.

Note that the ENF of the circuit A is syntactically identical to the sum-of-products expression C if the tags on the literals are ignored.

The necessity of the constraint on algebraic resubstitution with complement can also be seen in this example. Starting over again with B, if we chose $F_k = (c + a\overline{b})$ then the complement of F_k can also be expressed as $F'_k = (\overline{a}\,\overline{c} + b\overline{c})$. The node F'_k can now be algebraically resubstituted (without complement) into the other instance of $(\overline{a}\,\overline{c} + b\overline{c})$. This choice of $F'_k = (\overline{a}\,\overline{c} + b\overline{c})$ does not meet the requirements of Lemma 6.1, nor does it result in a circuit whose ENF is syntactically identical to C. It is interesting to note that this resubstitution does result in a circuit that is still `RPDFT`; however as there are more paths in the resubstituted newtork than in the original the `RPDF` test vector set for the original network will not robustly detect all path-delay-faults of the resulting resubstituted network. Thus, while `RPDFT` is retained in this example the vector set for complete coverage is not (Below we will see an example where resubstitution with complement may result in a circuit that is not `RPDFT`).

Corollary 6.1 : *Given a completely robustly path-delay-fault testable network, algebraic factorization with a constrained use of the inverse as defined in Lemma 6.1, produces a completely robustly* *path-delay-fault testable network.*

Proof: Follows directly from Theorem 6.2 and Theorem 6.3. ■

It should be noted that ENF reducibility is a sufficiency condition for retaining RPDFT, and not a necessary one. For instance, a XOR gate $a\overline{b} + \overline{a}b$ is RPDFT. Adding an inverter to the output of the XOR gate does not affect RPDFT, but the ENF of the resulting circuit $\overline{a\overline{b} + \overline{a}b}$ is more complex than the original ENF.

It can be shown that algebraic factorization without the constraint embodied by Lemma 6.1 may result in a circuit which is not RPDFT. This phenomenon is simply related to the fact that if the constraint is not satisfied, the number of paths may increase after factorization. In Figure 5(a), we have a multilevel circuit that is completely RPDFT. Algebraic factorization using the complement, produces the circuit of Figure 5(b), where the path $\{c', 2, 6, 0, 7, 10\}$ is not robustly testable. The path $\{c, 5, 7, 10\}$ in Figure 5(a) has split into two paths $\{c', 2, 6, 0, 7, 10\}$ and $\{c', 3, 6, 0, 7, 10\}$ in Figure 5(b). The increase in the number of paths has resulted in the non-retainment of RPDFT.

In our experiments with algebraic factorization with the complement using the program MIS (*c.f.* Section 6.9), we did not constrain the algebraic factorization, and RPDFT was retained, due to the fact that the number of paths typically did not increase.

6.5 Operations Which Do Not Preserve ENF Reducibility

While the previous sections showed that variations of algebraic factorization resulted in circuit with ENF reducibility and therefore retained the delay-fault testability of the original circuit, in this section we briefly note other transformations which do not retain ENF reducibility.

6.5.1 Eliminating a Node

Eliminating a node by pushing it into its fanout nodes can destroy ENF reducibility and RPDFT. An example is the circuit $(a + c)\overline{b} + b\overline{c}$, that is completely RPDFT. The ENF of this circuit given an appropriate ordering of the gates is $a_{\{1,2,4\}} \cdot \overline{b}_{\{2,4\}} + \overline{b}_{\{2,4\}} \cdot c_{\{1,2,4\}} + b_{\{3,4\}} \cdot \overline{c}_{\{3,4\}}$. Eliminating the node $a + c$ results in a circuit with ENF $a_{\{1,4\}} \cdot \overline{b}_{\{1,4\}} + \overline{b}_{\{2,4\}} \cdot c_{\{2,4\}} + b_{\{3,4\}} \cdot \overline{c}_{\{3,4\}}$. ENF reducibility is not preserved across elimination. Furthermore, the literal $\overline{b}$ in $a_{\{1,4\}} \cdot \overline{b}_{\{1,4\}}$ is not robustly path-delay-fault testable.

However, if one begins from a two-level network that is RPDFT, eliminating a node after a sequence of factorizations, will *not* destroy testability. This is because the conditions for the two-level cover to be fully testable are the most restrictive, and ENF reducibility against the initial two-level network is preserved. Alternatively, a different sequence of factorizations can be used to obtain factored-eliminated network.

6.5.2 Boolean Resubstitution

Figure 6 shows the two-level implementation of a function $f = a \cdot b + \overline{a} \cdot \overline{c} \cdot d + a \cdot c \cdot d + \overline{a} \cdot \overline{b} \cdot c \cdot \overline{d}$ which is fully RPDFT. Choose $g = c \cdot \overline{d} + \overline{c} \cdot d$ as a Boolean factor. After performing Boolean division, f can be written as $f = (c \cdot \overline{d} + \overline{c} \cdot d) \cdot (\overline{a} \cdot \overline{b}) + a \cdot b + \overline{a} \cdot \overline{c} \cdot d + a \cdot c \cdot d$. This implementation of f is shown in Figure 7. The ENF for this implementation is given by $f = a_{\{8,12\}} \cdot d_{\{8,12\}} \cdot c_{\{8,12\}} + \overline{a}_{\{1,11,12\}} \cdot c_{\{5,7,11,12\}} \cdot \overline{d}_{\{4,5,7,11,12\}} \cdot \overline{b}_{\{2,11,12\}} + \overline{a}_{\{1,11,12\}} \cdot d_{\{6,7,11,12\}} \cdot \overline{c}_{\{3,6,7,11,12\}} \cdot \overline{b}_{\{2,11,12\}} + \overline{a}_{\{1,10,12\}} \cdot \overline{c}_{\{3,10,12\}} \cdot d_{\{10,12\}} + a_{\{9,12\}} \cdot b_{\{9,12\}}$. Gate 6 in this implementation is not robustly testable for delay faults since the only cube in the ENF that contains gate 6 is $\overline{a} \cdot \overline{b} \cdot \overline{c} \cdot d$ which is covered by the cube $\overline{a} \cdot \overline{c} \cdot d$ which is also in the ENF. This Boolean factorization not only destroys the ENF reducibility, but this Boolean factorization is not RGDFT (and therefore not RPDFT) testable.

6.5.3 Phase Assignment

Phase assignment is capable of modifying the number of inverters in a path from inputs to outputs. This changes the polarity of a variable in the ENF and therefore ENF reducibility is lost in this process.

6.5.4 Simplification

Modifying the cover of any node in a Boolean network necessarily changes the ENF expression and therefore does not retain ENF reducibility. Furthermore, different covers of the same Boolean function have differing path-delay-fault testabilities so modifying the cover may also diminish the delay-fault testability.

6.6 Technology Mapping

So far our results treated technology independent optimizations only. In this section we state some theorems that show that common technology mapping procedures such as those used in DAGON [?] or MIS [?] retain delay-fault testability.

In tree-covering technology mapping each fanout-free subnetwork (tree) t of a combinational circuit C is replaced by a logically equivalent and single-fault testable tree t' in M. [2] Furthermore, the fanout points of M are exactly the fanout points of C. For each input i in in a fanout-free combinational circuit C there is a unique path π from that input to the output of that circuit. If M is a logically equivalent tree-covering technology mapped version of C then for each path π in C there is a **corresponding** path ρ in M.

Lemma 6.2 *: Let C be a* NAND *gate and* INVERTER *implementation of a circuit and let M be a tree-covering technology mapped version of C. A vector sequence $< v_1, v_2 >$ event sensitizes a path π in C iff $< v_1, v_2 >$ event sensitizes the corresponding path ρ in M.*

Proof: We again make the observation is that the ENF expression for C, call it E_C, is the same, up to renaming of tags, as the ENF expression for M, call it E_M. Moreover, for each path ρ in M there is a corresponding path π in C (not a many-to-one correspondence as in ENF reducibility), and therefore for each path tagged literal l_ρ in E_M there is a corresponding tagged literal l_π in E_C. Thus there exists a vector v which tests for l_π precisely when v tests for l_ρ and similarly vector sequence $< v_1, v_2 >$ event sensitizes π precisely when $< v_1, v_2 >$ event sensitizes ρ. ■

This lemma may be used to show:

Theorem 6.4 *: Delay-fault Testability of Tree-Mapped Networks:*
Let C be a NAND *gate and* INVERTER *implementation of a circuit and let M be a tree-covering technology mapped version of C. Then, if C was completely robustly gate/path delay-fault testable then M is also.*

Proof: Given a one-to-one correspondence between the paths of C and M, robust path-delay-fault testability is maintained. In the case of C being robustly gate-delay-fault testable (but not robustly path-delay-fault testable), in order to test a gate in C, multiple paths may have to be sensitized. The very same paths can be simultaneously sensitized in M to test the corresponding mapped gate in M. ■

Tree-mapping based technology mapping, as introduced in [?], maps a NAND and INVERTER representation of a circuit into a library of gates (such as a standard-cell library) by decomposing a circuit into trees and then mapping each tree into a logically equivalent tree built from library elements. Thus, by the theorem the tree-mapping approach to technology mapping as used in [?] and in [?] [?] retains delay-fault testability.

[2] Strictly speaking, tree-covering may be applied in such a way that a tree t in the input circuit is replaced by a non-tree structure after covering. In this discussion we are explicitly disallowing any such coverings.

6.7 Preserving Robust Gate-Delay-Fault Testability

We wish to construct a procedure which synthesizes a RGDFT multi-level circuit from a RGDFT two-level circuit. Because path-delay-fault testability was such a strong property, it was possible to retain it using unconstrained algebraic factorization. However, a constrained algebraic factorization procedure is required to retain RGDFT in a multi-level circuit.

Consider the two-level circuit shown in Figure 8(a). All four gates are robustly testable for gate delay faults. However, when the algebraic factor ab is extracted from the cube abc, the circuit is no longer RGDFT. The algebraically factored circuit is shown in Figure 8(b). In this circuit, gate 7 is untestable. In the original circuit, the only path through gate 3 which was testable was that originating at literal c. Thus there is no way to test the new gate corresponding to the factor ab when it is extracted from the cube abc in the original cover. A constrained factorization would be required in order to preserve testability of the gate delay faults in the circuit.

A more common scenario corresponds to algebraic factorization *increasing* gate-delay-fault testability as illustrated by following example. Consider a function whose sum-of-products expression is $f = a\overline{b}+\overline{a}b+c\overline{d}+\overline{c}d+\overline{a}d$. The AND gate associated with the cube $\overline{a}d$ is not robustly testable. However, if the circuit is implemented as the algebraically factored expression $f = a\overline{b} + c\overline{d} + \overline{c}d + \overline{a}(b + d)$, it will be completely robustly testable for gate delay faults.

The essence of an algebraic factorization procedure that retains RGDFT of the initial multi-level circuit is to ensure that in each cube of each factor (cube or kernel) there is at least one literal with the properties of literal l in Theorem 6.1. Of course if all literals in the initial two-level circuit have this property, as in the case of a RPDFT two-level circuit, then no constraints are required. The theorem below gives a succinct sufficiency condition for the retainment of RGDFT, beginning from a two-level circuit where a path through each gate is robustly testable.

Theorem 6.5 *: Given a two-level network C, where at least one path through each* AND *gate is robustly testable, let E_C be the ENF of circuit C. Let A be a multilevel circuit resulting from an algebraic factorization of C, with ENF E_A. If the tagged literals in E_C, which satisfy the conditions of Theorem 6.1 correspond to a set of tagged literals/paths in E_A/A that cover all the gates in A, then A is completely robustly gate-delay-fault testable.*

Proof: By Theorem 6.2 and 6.3, there is a one-to-one correspondence between the literals of E_A and E_C. Thus, if a tagged literal in E_C satisfies the conditions of Theorem 6.1, then the corresponding tagged literal in E_A does also. Since the set of paths in A, that correspond to tagged literals in E_A

satisfying the conditions of Theorem 6.1 cover all the gates in A, robustly testing these paths will robustly test all gate delay faults in A. ■

It is possible as illustrated in Figure 8 that the robustly testable paths in the two-level circuit do not produce corresponding paths which cover all the gates after algebraic factorization. Constraints have to be imposed on cube and kernel extraction in order to ensure that this condition is satisfied. Making this check after each cube or kernel extraction step during multilevel logic optimization will ensure that RGDFT is maintained.

6.8 Vector Generation

The typical environment for testing is a automatic test pattern generation phase followed by a fault simulation phase to see what the coverage of the vectors generated so far is. One of the advantages of synthesis for testability approaches is that test vectors can be generated automatically as a by-product of the synthesis process.

In the covering procedures of Sections 5.2 and 5.3, we "know" all the essential vertices and distance-1 OFF-set members for every cube in any current cover. These correspond to vector pairs $< v_1, v_2 >$ that test whether a $0 \rightarrow 1$ event along a path is slow. The sequence of vectors $< v_2, v_1 >$ test whether a $1 \rightarrow 0$ event is slow along a path. In Section 6 we showed how for each path ρ in a multilevel circuit obtained via algebraic factorization from a two-level network there exists a corresponding path π in the original two-level network L such that ρ is sensitized exactly when π is sensitized. Thus, if $< v_1, v_2 >$ is a robust path-delay-fault test for π in L then $< v_1, v_2 >$ is a robust path-delay-fault test for ρ in U. Finally, if M is a tree-covering based technology-mapped version of U, then the results of Section 6.6 show that for each path σ in M there is a corresponding path ρ in U such that if $< v_1, v_2 >$ is a robust path-delay-fault test for ρ in U then $< v_1, v_2 >$ is a robust path-delay-fault test for σ in M. Thus, for every path σ in M there is a corresponding path π in L such that the path-delay-fault test for π covers σ. This means that a complete test set for L, call it $S(L)$ is a complete set for M.

One might argue that while $S(L)$ covers all faults in M, that a smaller set could be applied, and thus fault simulation is still desirable. For a single-output function, as each vector pair in $S(L)$ event sensitizes exactly one path there is no possibility for accidental path-delay-fault detection.

The sufficiency condition for preserving RGDFT given in Section 6.7 ensures that the test vectors that detect all gate delay faults in a two-level network detect all gate delay faults in a algebraically-factored and technology-mapped multilevel network, precisely by the arguments above.

6.9 Experimental Results

In this section, experimental results are presented demonstrating the testability preservation and enhancement procedures described in the preceding sections. In particular, results are presented for the algebraic factorization techniques that use the inverse. Results were obtained from 32 examples, 29 being examples from the MCNC Benchmark set, with the remaining three being other examples chosen to illustrate specific points of interest.

In Table 9, statistics are presented for the 32 examples, including the number of inputs and outputs of each function. The number of product terms and the level of testability for gate delay faults and path delay faults are also presented, both for a multiple-output minimized two-level cover, and for a single-output minimized two-level cover. For the multiple-output minimized cover, the prime and irredundant cover was obtained by using the multiple-output logic minimizer ESPRESSO [?]. The single-output minimized cover was obtained by using the same program, except that each output was optimized separately.

For the single-output optimized cover, all 32 examples had 100 percent testability for gate delay faults. While it was shown in Section 5.4 that a two-level prime and irredundant cover for a single-output function is not necessarily completely `RGDFT`, these results show that a wide range of examples are completely `RGDFT`, when a single-output optimization is performed. The multiple-output optimized covers had an average `RGDFT` of 98 percent.

In Table 4, comparisons of the number of literals in different optimized multilevel implementations of the examples are given. Results are given for three cases. In the first case, *alg*, results were obtained by performing algebraic factorization without using the complement on the single-output optimized two-level cover of the function. The results for the second case, *alg-i*, were obtained by performing algebraic factorization using both the factor and its inverse on the single-output optimized two-level cover of the function. In the final case, *bool*, the results were obtained by performing an unconstrained Boolean logic optimization on both the single-output and multiple-output minimized two-level cover of the function, and picking the smaller of the two literal counts. All three sets of results were obtained by using a version of the program MIS [?].

The results in Table 4 show that the number of literals is not significantly impacted by restricting oneself to using purely algebraic techniques. The unconstrained multilevel Boolean optimization gave better results than the multilevel algebraic optimizations in 15 of the 32 examples, with the average reduction in the number of literals being 8 percent. Using the algebraic factor and its inverse obtained an average decrease of 14 percent in the number of literals when compared to using algebraic factorization without the inverse.

Our literal count comparisons are based on using the `script.bn` sequence for Boolean optimization in the MIS system and the `script.alg` sequence for algebraic optimization. The use of other scripts in MIS or the use of predominantly Boolean transformation-based logic optimizers such as BOLD [?] can produce smaller literal counts for some examples in the unconstrained optimization case. The algebraic script can be tuned to produce better results for certain examples as well. Our focus here is in illustrating that literal counts are not significantly impacted using algebraic techniques, rather than in comparing optimization methods.

As shown in Section 6.3 algebraic factorization (without using the inverse or with a constrained use of the inverse) of a fully testable two-level function will preserve RPDFT. In general, RGDFT will also be maintained. A constraint involved with using the inverse was shown as a necessary condition in Theorem 6.3 to satisfy ENF reducibility and is therefore a sufficiency condition for the retainment of RPDFT. It was shown via the example of Figure 5 that an unconstrained use of the inverse could destroy RPDFT. In our experiments with the *alg-i* script, we did not constrain the use of the inverse. We found that in a large fraction of the cases (notably in the non-arithmetic functions), ENF reducibility was satisfied after the *alg-i* script. In examples such as add4 and par8, ENF reducibility was not satisfied, but RPDFT was retained.

It was also shown in Section 6.5 that unconstrained Boolean operations do not necessarily preserve testability. Table 5 presents the level of fault coverage for the multilevel implementations for which the number of literals was given in Table 4. The percentage fault coverage is given for gate delay and path delay faults. The testability results for the multilevel implementations were obtained using an ENF testability analysis program.

The results of Table 5 show that the two different algebraic factorization based optimizations preserved the level of RPDFT. In the ten examples in which the two-level cover was 100 percent testable for path delay faults, the multilevel cover is also 100 percent testable. In the other 22 examples in which the two-level covers were not fully testable, the level of RPDFT *increased* by an average of 3 percent. This phenomenon was illustrated in reverse via an example in Section 6.5.1. Starting from a sum-of-product cover that is not completely RPDFT, factoring produces a completely RPDFT multilevel circuit. The level of RPDFT for the algebraically factored implementations was significantly better in most cases than the level of RPDFT for the unconstrained Boolean optimized implementation. The average RPDFT for the Boolean optimized implementations was 34 percent less than the RPDFT level for the algebraically factored implementations.

Examples add4, par8, and pla4 are the three examples which are not part of the MCNC Benchmark set. The first two were included to illustrate types of examples in which the decrease in the number of literals can be very significant when using both an algebraic factor and its inverse, as opposed to using just the factor. The example add4 is a 4-bit adder with carry input, and the example par8 is an 8-bit parity generator. Due to the presence of exclusive OR type structures in these examples, using the inverse of a factor can result in significant improvements to the number of literals. For add4, the number of literals decreases by 40 percent, while for par8 the number of literals decreases by 45 percent. The example pla4 is a large (31 inputs, 90 outputs) PLA from an industrial example which was included to demonstrate that the techniques presented in this report are viable for large examples.

While a constrained algebraic factorization is required, in principle, to retain complete RGDFT, typically optimized two-level covers have a large number of paths that are robustly testable, and unconstrained algebraic factorization more or less maintains the level of RGDFT as shown in Table 5. A notable exception is sym10 whose RPDFT level is rather low. Thus, an unconstrained algebraic factorization results in a RGDFT level of 87 percent beginning from a 100 percent RGDFT two-level cover. Boolean factorization can substantially lower RGDFT as illustrated by the results.

If one is interested in RGDFT alone, a multiple-output minimized cover can be used as a starting point for algebraic factorization, provided the testability of the cover is high. For instance, in the case of pla4 and bc0, the multiple-output minimized cover is almost as testable as the single-output minimized cover. The freedom in using the multiple-output cover as a starting point, allows for more efficient synthesis. *No* area penalty is incurred and $>$ 99 percent RGDFT is achieved for the examples pla4 and bc0, if one begins algebraic factorization from a multiple-output minimized cover.

7 Synthesis of Integrated Circuit Designs for Delay-Fault Testability

7.1 Cascading Combinational Logic Blocks

So far the results of this paper have been developed for individual modules of combinational logic that can be flattened to two-level circuits. In this section we consider how these results can be practically applied in an environment in which flattenable combinational logic modules are cascaded with other, perhaps unflattenable, combinational logic modules. Experience has shown that if a combinational logic module is the result of the synthesis of a finite state machine (FSM) description, then it can be flattened to a two-level circuit. On the other hand some have argued that most register-bounded combinational blocks, even in control portions of designs, are not flattenable [?]. How can these two points be consistent? A code fragment describing a portion of a hardware system is shown below. This description would typically be synthesized into a combinational logic block by a system such as CONES [?].

```
switch(state){
   ...
   case state_22:
     if(packet_odd_parity)
         ...
   case state_30:
     if(add_overflow)
         ...
}
```

In this control module the two lines `packet_odd_parity` and `add_overflow` are treated as primary inputs, but in the final design the `packet_even_parity` signal is the output of a 32-bit parity generator module and the `add_overflow` is the final carry-out output of a 16-bit adder module. Both of these signals are the outputs of combinational logic modules and will feed the combinational logic module associated with the FSM. If the hierarchy is removed and the register-bounded combinational net-list resulting from the adder and parity generator cascading into the controller is examined, it will be found that the resulting combinational logic block is *not flattenable* to a two-level circuit. This is because of the explosion in product terms due to flattening the parity generator and adder. Thus the synthesis procedures developed so far cannot be directly applied because the single combinational logic block is not flattenable to two levels.

What is the impact of this on the testability of the resulting circuit? Let us recall that when optimizing for circuit area, the savings are clearly compositional. In other words the savings in optimizing over k blocks is the sum of the savings for each individual block. The same is true to a lesser extent for timing optimization. As long as a timing optimization system is able to take in variable arrival times for inputs, two cascading combinational logic blocks can be optimized independently. Unfortunately, in the context of circuit testing, hierarchy is even more difficult to manage and *the composition of two completely testable circuits may be poorly testable* [?].

Our solution to this problem is to develop sets of *composition rules* that allow us to prove conditions such that when one `RPDFT` circuit module feeds another `RPDFT` circuit module, then the resulting cascade is also `RPDFT`. We begin with a modest undertaking in this regard, but one that may be applied to cover a significant number of the cases of cascading combinational logic modules in control-dominated designs that may not be flattened to two-level circuits. We first give a composition rule that gives a sufficiency condition for the result of a composition of two `RPDFT` circuits to be `RPDFT`. We then apply this theorem to show that some common non-flattenable circuits are robustly path-delay-fault testable. We then apply the theorem in another way to show how cascading circuits consisting of both flattenable and unflattenable modules, such as those indicated by the FSM description above can also be proven to be robustly path-delay-fault testable.

Theorem 7.1 *: If each of C_1, C_2 is a robustly path-delay-fault testable combinational circuit and if a circuit C is created by connecting one output o_j of C_1 to one input i_k of C_2 in such a way that the input variables of C_1 and C_2 are disjoint, then C is also robustly path-delay-fault testable.*

Proof: Let σ be a path in C. Suppose that σ was also a path in C_2. Let i_m be the input of C_2 associated with σ. (Note that $i_m \neq i_k$.) By the multiple-output generalization of Theorem 6.1 there exists a vector sequence $< x_1, x_2 >$ that is a robust path delay fault test for σ in C_2 such that only the value of input variable i_m changes in $< x_1, x_2 >$. Now consider the value of input i_k in C_2 in vector x_1 (it will be the same in x_2). Without loss of generality assume that the value is 1. Let w_1 be a vector over the input variables to C_1 such that the output $o_j = 1$. Let $v_1 = \{x_1 - \{i_k\}\} \cup w_1$ and let $v_2 = \{x_2 - \{i_k\}\} \cup w_1$. The vector v_1 provides the same non-controlling values along the side-inputs of σ as did x_1. Thus, the transition caused by the change in input i_m in vector v_2 propagates along σ just as in x_2. Thus $< v_1, v_2 >$ is a robust path-delay-fault test for σ.

Suppose that σ is a path in C_1. In other words, σ is a path from an input of C_1 to an output o_p of C_1 such that $o_p \neq o_j$ and thus σ is not affected by the cascading of C_1 and C_2. Thus the original vector sequence $< v_1, v_2 >$ for σ in C_1 may be applied to give a robust path-delay-fault vector sequence $< v_1, v_2 >$ for σ in C.

Suppose that σ is not a path in C_1 or a path in C_2. Then the path σ must consist of a path, call it π beginning at some input i_n of C_1 and passing out output o_j of C_1, entering a path, call it ρ, beginning at what was formerly input i_k of C_2 and out of some output o_p of C_2. Our approach will be to build a vector sequence $< v_1, v_2 >$ for testing σ by using the sequence $< x_1, x_2 >$ used

to test for π and the sequence $< w_1, w_2 >$ used to test for ρ. By the multiple output generalization of Theorem 6.1 there exists a vector sequence $< w_1, w_2 >$ that is a robust path delay fault test for ρ in C_2 such that only the value of input variable i_k changes in $< x_1, x_2 >$. Suppose that the value of i_k in x_1 is 0 and the value of i_k in x_2 is 1. By Theorem 6.1, there exists a vector sequence $< x_1, x_2 >$ that is a robust path delay fault test for π in C_1 such that only the value of input variable i_n changes in $< x_1, x_2 >$. Let $v_1 = x_1 \cup \{w_1 - \{i_k\}\}$ and let $v_2 = x_2 \cup \{w_1 - \{i_k\}\}$. The vector w_1 supplied non-controlling values along side-inputs of ρ in C_2; thus, the vector $\{w_1 - \{i_k\}\}$ does the same. The vector x_1 supplied non-controlling values along side-inputs of π in C_1 resulting in a 0 on the output of C_1; thus, the vector $v_1 = x_1 \cup \{w_1 - \{i_k\}\}$ propagates a 0 along σ. The vector x_2 supplied non-controlling values along side-inputs of π in C_1 resulting in a 1 on the output of C_1; thus, the vector $v_2 = x_2 \cup \{w_1 - \{i_k\}\}$ propagates a $0 \rightarrow 1$ event along σ to the output of C.
In the alternative case that the value of i_k in x_1 is 1 and the value of i_k in x_2 is 0, then the vector pair $< v_1, v_2 >$ tests for a $1 \rightarrow 0$ event along σ to the output of C and the vector pair $< v_2, v_1 >$ may be applied to test for a $0 \rightarrow 1$ event. ■

Note that in Theorem 7.1 we assume complete RPDFT of each of C_1 and C_2. This is very important. If a single path starting with input i_k of C_2 is not robustly path-delay-fault testable then *all* the paths in the sub-circuit C_1 may be untestable in C. This is an instance where the unate-decomposition techniques of [?] [?] may be used to augment algebraic factorization to ensure a completely RPDFT circuit.

7.2 Regular Structures

We now show how Theorem 7.1 may be applied to prove the RPDFT of regular structures.

7.2.1 Parity Trees

We wish to show that parity trees are RPDFT. We begin by showing that a single two-input XOR gate is RPDFT. Let a and b be the primary inputs to a XOR gate built from primitive gates as in the dotted box in Figure 9. The ENF expression for the XOR gate is $x = \overline{a_{\{1,3,5\}}} \cdot b_{\{3,5\}} + a_{\{4,5\}} \cdot \overline{b_{\{2,4,5\}}}$. The reader need not be concerned over the gate selection, any topologically equivalent XOR structure would have the same ENF. It is easy to see that this expression satisfies the conditions of Theorem 6.1 and thus a single XOR gate is RPDFT. For example, consider the path associated with the tagged literal $\overline{a_{\{1,3,5\}}}$. The relatively essential vertex associated with the cubes containing $\overline{a_{\{1,3,5\}}}$ is $v_2 = (a = 0, b = 1)$. A member of the OFF-set of the XOR function distance-1 from v_2 is $v_1 = (a = 1, b = 1))$. Thus vector pair $<(v_1 = (a = 1, b = 1)), (v_2 = (a = 0, b = 1))>$ is a robust delay-fault test for the path from primary input a through gates $\{1, 3, 5\}$ to the output.

We may now use Theorem 7.1 to show that an parity tree of arbitrary size is RPDFT.

Theorem 7.2 *: If P is a parity tree built from* XOR *gates that are logically and topologically equivalent to those in the dotted box of Figure 9, then P is robustly path-delay-fault testable.*

Proof (by induction on the number of bits): Let $S(k)$ be the statement that if P is a parity-tree of k bits built from XOR gates are logically and topologically equivalent to those in the dotted box of Figure 9, then P is robustly path-delay-fault testable.

Basis $k = 1$. By the arguments given above a two-bit XOR gate is robustly path-delay-fault testable.

Suppose $S(k)$ for $k \leq n$. Consider $S(n)$. By the induction hypothesis a parity tree of $n-1$ bits is robustly path-delay-fault testable. Let us call such an $n-1$ bit parity tree C_2 with gate inputs $i_1 \ldots i_{n-1}$ and primary input variables $v_1 \ldots v_{n-1}$. Let us call a single XOR gate C_1. C_1 is also robustly path-delay-fault testable. Let us connect C_1 to the input i_{n-1} in C_2 and connect the input variable v_{n-1} to one input of C_2 and a new variable v_n with the other input of C_2. As the inputs of C_1 and C_2 are disjoint, and each of the circuits are robustly path-delay-fault testable, then by Theorem 7.1 the resulting circuit is also robustly path-delay-fault testable. Note that it doesn't matter which of the $n-1$ inputs we connect to so we an always build the parity tree in a balanced way. ■

7.2.2 Ripple-carry Adders

We proceed to show that a full adder is RPDFT. Figure 9 shows the labeled gate network for a full adder. In Section 7.2.1 we showed that a parity tree was RPDFT and so we argue that the sum (S) output of a full adder is also RPDFT. The ENF expression for the *carry-out* signal, CO, is: $CO = \overline{a_{\{1,3,5,7,8\}}} \cdot b_{\{3,5,7,8\}} \cdot c_{\{7,8\}} + a_{\{4,5,7,8\}} \cdot \overline{b_{\{2,4,5,7,8\}}} \cdot c_{\{7,8\}} + a_{\{6,8\}} \cdot b_{\{6,8\}}$. As before any topologically equivalent gate network would have an equivalent ENF. It may be shown by inspection that this expression satisfies the conditions of Theorem 6.1 and thus a single full adder is RPDFT. For example, consider the path associated with the tagged literal $\overline{a_{\{1,3,5,7,8\}}}$. The relatively essential vertex associated with the cubes containing $\overline{a_{\{1,3,5,7,8\}}}$ is $v_2 = (a = 0, b = 1, c = 1)$. A member of the OFF-set of the *carry-out* function distance-1 from v_2 is $v_1 = (a = 1, b = 1, c = 1))$. Thus vector pair $<(v_1 = (a = 1, b = 1, c = 1)), (v_2 = (a = 0, b = 1, c = 1))>$ is a robust delay-fault test for the path from primary input a through gates $\{1, 3, 5, 7, 8\}$ to the output.

Having shown a full adder is `RPDFT` we continue to show that a ripple-carry adder is `RPDFT`. An informal inductive argument is as follows.

Theorem 7.3 *: If A is a ripple-carry adder built from full adders that are logically and topologically equivalent to that in Figure 9 then A is robustly path-delay-fault testable.*

Proof (by induction on the number of bits): Let $P(k)$ be the statement that if A is a ripple-carry adder of k bits built from full adders that are logically and topologically equivalent to that in Figure 9 then A is robustly path-delay-fault testable.
Basis $k = 1$. A one-bit ripple-carry adder and by the arguments above a full adder is robustly path-delay-fault testable.

Suppose $P(k)$ for $k \leq n$. Consider $P(n)$. By the induction hypothesis the first $n - 1$ stages of the ripple-carry adder are robustly path-delay-fault testable. Call this sub-circuit C_1. Consider the full adder associated with the n-th stage in a ripple-carry adder. Call this sub-circuit C_2. The inputs to C_2 are two primary inputs a_n and b_n, and CO_{n-1}, the carry-out from C_1. Each of C_1 and C_2 is robustly path-delay-fault testable, their primary inputs are disjoint and one output of C_1 feeds one input of C_2. Therefore by Theorem 7.1 the n-stage ripple-carry adder is also robustly path-delay-fault testable. ■

7.3 Hierarchical Circuits

Theorem 7.1 was used to show that certain regular structures are `RPDFT`. We now employ Theorem 7.1 to show how a FSM which takes as inputs the output of a parity generator and the carry-out of an adder can be shown to be `RPDFT`.

The first step is to synthesize a `RPDFT` FSM module according to the techniques described in Section 6 treating the `packet_odd_parity` and `add_overflow` signals as primary inputs. The parity generator module has already been shown to be individually `RPDFT` so Theorem 7.1 may be invoked once to show that the parity generator and the FSM description (with the `add_overflow` signal treated as a primary input). The ripple-carry adder module has been shown to be individually `RPDFT` so Theorem 7.1 may be invoked a second time to show that the parity generator, the FSM description and the ripple-carry adder module, all form a `RPDFT` combinational logic circuit.

7.4 Obtaining Test Vectors By Composition

In Theorem 7.1 we gave a constructive proof that is essentially an algorithm for deriving the test vectors for each path in the resulting circuit C, from the path-delay-fault test vectors for each path in C_1 and C_2. Thus, we have a procedure for building a complete set of vectors as well.

7.5 The Global Strategy Toward Synthesis of Entire Chips that are Robustly Path-Delay-Fault Testable

In this section, we describe our global strategy toward the synthesis of integrated circuit designs for high testability, and particularly for synthesis of circuits that are RPDFT. Our approach is hierarchical. Firstly, we assume a full scan structure capable of delivering a delay-fault vector *pair* as discussed in Section 2. We then synthesize and compose RPDFT combinational modules, while retaining full delay-fault testability according to composition rules one of which was described in Section 7.1. Individual modules may be synthesized to be RPDFT in a number of ways. Irregular modules, such as control blocks, and other small combinational blocks, can typically be flattened to two-levels and synthesized into RPDFT multilevel circuits using the procedures of Section 6. Full-adder and multiplier cells, which form the atomic elements of arithmetic structures such as adders and multipliers, can be individually crafted to be RPDFT. Then composition rules may be applied to build complete RPDFT arithmetic structures. Regular logic structures, such as parity trees, can be built up in a similar way from atomically RPDFT modules, such as XOR gates.

Ripple-carry adders, parity trees, subtractors etc all obey the composition topology summarized by Theorem 7.1. More general composition rules that maintain testability, for regular structures such as serial and parallel comparators, adder-incrementers and parallel multipliers, under the path delay, transistor stuck-open and gate delay fault models were developed in [?].

7.6 A Data Encryption Chip

Based on the results discussed in the previous sections, we now proceed with the principal aim of the paper: the synthesis of entire chips that are RPDFT. The first circuit design we consider is that of a Data Encryption Chip. Our discussion of the chip, and alternative architectures, relies heavily on a chapter from [?] and our principal contribution rests on the synthesis of a version that is RPDFT.

7.6.1 Introduction

Digital encryption is the process of transforming a block of binary data (*plain-text*) into a different representation (*cipher-text*) under the control of a *key*. Decryption is the reverse process – given the cipher-text and key, derive the plain-text. The DES algorithm is one means of digital encryption. It operates on a 64-bit plain-text block using a 48-bit key and produces a 64-bit cipher-text block.

The DES algorithm was published as a Federal Information Processing Standard in January 1977 [?]. DES is in wide use now for low security applications. Many satellite communications (*e.g.* cable TV) are encrypted using the DES algorithm. Sun Microsystems is using DES as the basis for their secure ether-net communication product. DES is also used as the security mechanism for passwords in Berkeley Unix [?].

A public domain C program is available for the DES algorithm. The program performs approximately 200 block encryptions per second on a DEC MicroVax-II which corresponds to an encryption rate of 13,000 bits per second. This speed is practical for low-speed communication channels, but is not suitable for disk-controller interfaces or ether-net interfaces which operate at millions of bits per second. Therefore, special-purpose integrated circuits are used in these applications. The automatic synthesis of such a circuit for minimal area was presented in [?]. Here, we are concerned with synthesis for complete RPDFT.

7.6.2 The DES Algorithm

In Figure 10 the topology of the interconnected logic blocks in a single stage of the DES algorithm is shown.

The 64-bit data value PI_1 is permuted with the initial permutation IP. A multiplexor MUX is used to select either the data value or the output from the previous stage PS. The data value is split into two 32-bit fields, L and R. The L field is permuted using the E block and the result XORed with the key input, namely, PI_2. The XOR result is scrambled using the SC and P blocks, and the result XORed with the R field to produce the new R field for the next iteration. The L field for the next iteration is the old R field.

7.6.3 Design Alternatives

There are several design alternatives for the DES algorithm enumerated in [?]. We will concentrate on the synthesis of a single stage of the DES algorithm. Sixteen such stages can be cascaded in a combinational fashion or pipelined with registers in between each stage or control logic can be used to cycle the data through a single stage sixteen times. DES is an interesting example for synthesis for combinational testability, since it represents a large piece of non-flattenable combinational logic. Collapsing even one stage of the DES design results in a PLA with over 40,000 product terms [?]. The combinational part of the DES design has 176 input signals (48-bit key plus 64-bit present state lines, which are the outputs of the flip-flops plain-text block) and 64 output signals (64-bit cipher-text block). A fully combinational design would have sixteen copies of the basic stage and the computation time would be sixteen times the delay of one stage. No registers are used in a fully combinational design.

7.6.4 Synthesis Results

The IP, MUX, E, SC and P blocks can all be individually flattened into a sum-of-products representation and synthesized to be fully RPDFT via algebraic factorization. However, the composition of these blocks will not necessarily be fully RPDFT.

Inspecting the DES description, we see that the E block, which is a permuter, expands its 32 inputs to 48 outputs. This means that if there is a transition in one input to the E block there may be multiple transitions at its outputs (This is not the case with the IP and P blocks). We have to flatten the E block, the set of XOR gates fed by the E block and the $PI2$ lines, the SC and P blocks all together, in order to ensure full RPDFT. There may be multiple transitions at the outputs of this composite E, XOR, SC, P block, due to a single $PI1$, $PI2$ or $start$ input change, but these outputs do not reconverge. Algebraically-factoring the composite block and other blocks retains testability, and testability is maintained after composing the E, XOR, SC, P block with the remaining blocks.

The program misII [?] was used to flatten, as well as to factor, the logic.

Area penalties in this combinational synthesis for testability procedure may arise from the fact that logic partitioning is required – we cannot share logic between, for instance, the IP and XOR blocks. Also, after flattening, the IP, E, SC and P blocks have to be optimized using strictly algebraic techniques. However, experimental results indicate that eliminating Boolean minimization techniques does not significantly impact the area of the synthesized DES, as was illustrated with other examples in Section 6.9.

Statistics for the fully testable DES obtained via testability-driven synthesis and a circuit obtained via unconstrained synthesis are given in Table 6. The CPU time corresponds to the time required for both test generation and synthesis in hours on a VAX 11/8650.

A dramatic increase in testability has been achieved while incurring a small area penalty in the combinational logic.

Attempting to generate robust tests for all paths in the DES implementation obtained via unconstrained synthesis would require huge amounts of CPU time. To ensure that we did not overstate the benefits of our approach we used an optimistic estimate, using simple topological analysis and a ENF-based path-delay-fault testing program, for the fault coverage for the circuit obtained via unconstrained synthesis; the path delay fault coverage for that version may be much worse. The circuit was broken up into subcircuits, and robust path-delay-fault coverages were obtained for each of the subcircuits. An optimistic estimate of the fault coverage of the entire circuit was derived by analyzing the interconnection of these subcircuits. RPDFT has very stringent topological requirements, and, in general, composing modules that have even a small number of paths that are not robustly testable typically results in a circuit that is very untestable.

The test vectors were derived using information about the essential vertices of two-level networks and via composition as described in Section 7.4. No attempt to compact the test vectors was made. Typically, path-delay-fault test sets cannot be significantly compacted, since only a single path to any particular output is sensitized. Compaction can only occur if we can merge delay-fault tests for different paths beginning from the same input and ending at different outputs. While the test set is large, it is *very* comprehensive.

7.7 A Simple Microprocessor

7.7.1 Introduction

In this section, we will describe the synthesis of a fully testable PDP-8-like μ-processor from a RTL description. The description we are using has been modified from Chapter 4 of [?]. The PDP-8 is a classical example of a datapath/controller circuit embodying many of the interesting aspects of general-purpose computers.

The PDP-8 description we used has 14 primary inputs, [3] 24 primary outputs, and 51 flip-flops. For an indication of logic complexity see Section 7.7.4. It contains control logic as well as an arithmetic unit capable of performing addition, subtraction, Boolean AND etc. We chose to implement the multiplication operation as well.

The topology of the combinational interconnect blocks that are separated by registers in the PDP-8

[3] We assumed an off-chip memory.

is shown in Figure 11. The arithmetic unit contains the large fraction of logic gates.

7.7.2 Flattenable Logic

Other than the arithmetic unit in the PDP-8 all the remaining logic, can be flattened into sum-of-products form and can be synthesized as a fully RPDFT multilevel circuit.

7.7.3 The Arithmetic Logic Unit and Decode

The arithmetic logic unit (ALU) in the PDP-8 is capable of addition, subtraction and Boolean operations. Each of these operator blocks can be separately synthesized to be completely RPDFT. Given that the ALU performs several functions, an ALUDECODE block is required to specify what operation, if any, is to be performed during a particular instruction. Unfortunately, the ALUDECODE and the ALU do *not* form an interconnection which satisfies the composition rule described in Section 7.1.

In order to synthesize a fully testable ALU and ALUDECODE block, one option is to use the configuration shown in Figure 12. In this approach each of the ALU operations is realized separately and the result is chosen based on the ALUDECODE signals. In this case, we can fix the decoder signals to the appropriate values to test paths from the output of any operator. Paths beginning from the decoder signals can be tested by producing the appropriate values for the results from the different operators. Note that the logic in the different operations *can* be shared; we can perform an algebraic factorization of the configuration shown in Figure 12 while maintaining RPDFT. However, the optimum configuration of an adder-subtractor, for example, involves a Boolean resubstitution of the ALUDECODE signals. Hence the structure of Figure 12 involves an area penalty, though it allows the sharing of logic across operations.

The incrementer is essentially a chain of half-adders. This incrementer chain obeys the composition rule, as shown in Figure 13, and can be synthesized to be completely testable.

Another option in the case of decoding logic, that typically has a smaller area penalty is to enhance controllability by adding extra flip-flops. Typically, the ALUDECODE signals are outputs of a FSM controller. One can always synthesize a FSM controller (that is fully RPDFT) such that the outputs of the FSM are outputs of a subset of the state flip-flops. Say, we have a ALU capable of performing addition and subtraction based on a 0/1 decode output signal from the FSM controller. Consider a 8-bit ALU with two 4-bit stages. Each of these stages receives the decode signal as input, and therefore the ALU is not path-delay-fault testable. However, if we simply duplicate the decode output signal of the FSM such that we have two flip-flops in the FSM (that are driven by the same input and therefore always assert the same output) and use the different outputs of these flip-flops as inputs to the two 4-bit stages, we have full `RPDFT`. This is because we have effectively no common input between the two stages, since we can control the values stored in the scan flip-flops independently. The above can be generalized to multiple decode signals and multiple ALU stages. We chose this option in the synthesis of our μprocessor.

We also implemented a multiply operation, common in more complex μ-processors. Multiplication can be performed either in a serial fashion by repeated addition or using a parallel multiplier. The latter is the more interesting case, since the combinational logic is much larger. The topology of a 2×2 parallel multiplier is shown in Figure 14.

In Figure 15, we have the basic cell we use for the multiplier. The multiplier does not obey the composition rule as given by Theorem 7.1, but obeys a more general one considered in [**?**]. For instance, consider the input `a[0]` in the 2×2 multiplier of Figure 14. It passes through the multiplier cell `2` and arrives at input `z` of cell `3`. It is also directly connected to input `x` of cell `3`. In order to preclude multiple transitions at the input to the $+$ block of cell `3`, we require `b[1]` to be set to 0, when testing for paths beginning from `a[0]` and passing through cell `2`. This implies that the output of the AND gate in cell `3` is a 0. We need to test all paths passing through input `z` of cell `3`. To do this we require values of both 0 and 1 at the input of the XOR gate that is fed by input `z`. We assume that the XOR gate is implemented by two inverters, two AND gates and an OR gate.

We can achieve this by placing an appropriate value at input `y`, though the output of `w.x` is 0. Note that if, in fact, we had used a different basic cell, where the XOR of `w.x` and `z` was computed before XORing with `y`, this would not be the case, and the multiplier would not be fully `RPDFT`.

The 2×2 multiplier of Figure 14 can be generalized to a $n \times 2$ multiplier while maintaining path-delay-fault testability. Unfortunately, generalization to a $n \times n$ multiplier is not possible. Typically, for large n, multipliers are either pipelined after every 2 adder arrays or the output of a $n \times 2$ array is fed back to itself to save hardware. This implies that there is a set of flip-flops between each $n \times 2$ array, ensuring path-delay-fault testability. The topology of Figure 14 has to be augmented with a final set of adders to actually obtain the multiplicand. This does not affect `RPDFT` in the $n \times 2$ case.

The adder circuitry of Figure 15 was used for the addition and subtraction operations in the ALU as well.

7.7.4 Synthesis Results

The glue, control and decode logic (including multiplexors) can be be flattened into a sum-of-products representation and synthesized to be fully RPDFT via algebraic factorization. The arithmetic unit in the PDP-8 was separated from the initial description and synthesized independently. A 8×2 multiplier was synthesized to be fully RPDFT. Multiplying two 8-bit quantities thus requires 5 clock cycles; $8/2 = 4$ clock cycles to obtain the partial products and a final clock cycle to add them.

The interactions between the multiplexor logic, control logic and the ALU are simple and an inspection of Figure 11 shows that the interconnection obeys the composition rule. Thus, the entire circuit is completely path-delay-fault testable.

Statistics for the fully testable PDP-8 obtained via testability-driven synthesis and a circuit obtained via unconstrained synthesis are given in Table 7.

As before, the fault coverage for the circuit obtained via unconstrained synthesis is an optimistic figure, arrived at using topological analysis and a delay-fault testing program (as described in the previous section). The test vectors were derived using information about the relatively essential vertices of two-level networks and via composition as described in Section 7.4. Again, a dramatic increase in testability has been achieved. The area penalty incurred corresponding to the increase in the number of literals (transistors), is primarily due to the required increase in the number of flip-flops in the ALU and the use of strictly algebraic operations while optimizing the glue and control logic.

While the PDP-8 is a small μ-processor, the synthesis techniques we have described scale quite well, and it is relatively easy to synthesize a 32-bit μ-processor with a similar architecture to be fully testable. In the next section, we deal with a larger Application-Specific Integrated Circuit (ASIC) example.

7.8 The Viterbi Processor

7.8.1 Introduction

It is widely accepted that Hidden Markov Models (HMM) are currently the most efficient technique to model speech for use in automatic speech recognition [?]. The states of a HMM speech process correspond to generic speech sounds (*e.g.* a phoneme or part of a phoneme). The task that has to be performed during speech recognition is to find, for a given HMM, the state sequence that most likely could have produced the speech that has to be recognized. A very efficient search algorithm for the most likely state sequence is the *Viterbi* algorithm.

In [?], a hardware design capable of performing the Viterbi equation for up to 50,000 states in real time was presented. In Figure 16, a block diagram of the Viterbi processor is shown. The hierarchical composition and interconnection of the various blocks are of primary interest. The processor has up to 11 levels of pipelining. The basic blocks are 8 datapaths, 3 bidirectional memories, and a control unit with data stationary control architecture. The chip was fabricated through MOSIS using a $2\mu m$ CMOS technology. Including 204 pads, it has a die size of $11.6 \times 9.8\ mm^2$ with 25,000 transistors.

Here, we are concerned primarily with the synthesis of the combinational logic in the processor (including the arithmetic units in the datapaths). Logic between the registers is synthesized to be fully `RPDFT`.

7.8.2 The Datapaths

The eight datapaths in the Viterbi processor have varying complexities. In Figure 17, the topology of the datapath *predcom* which is itself composed of three smaller datapaths is shown. As can be seen, the interconnection topology conforms to the composition rule and hence synthesizing a fully testable comparator suffices to ensure full `RPDFT` for *predcom*.

In Figure 18 the topology of the datapath *outmindp* is shown. The datapath has a comparator that feeds a multiplexor – for example, the comparator $prob7 > prob8$ that controls the multiplexor *bestmux*. This topology implements the maximum operation and is *not* robustly path-delay-fault testable. We can however synthesize a fully testable maximum function by composing modified subtractor cells (*c.f.* Section 7.8.5).

The other datapaths are very similar and can also be synthesized to be fully testable.

7.8.3 The Controller

The controller in the Viterbi processor is a 16-state finite state machine with pipeline registers that produce delayed control outputs. In fact, all the control output signals but one, namely, *stall* are outputs of registers. The signal *stall* shown in Figure 16 is both a primary output of the chip and feeds the *minpla* block which is a small piece of combinational logic. The outputs of the *minpla* block are latched with registers.

7.8.4 Putting Them Together

As mentioned previously, other than the signal *stall* the control outputs are all latched. This implies an effective decoupling between the datapaths and control for testability purposes. There is however, quite a lot of interaction between the datapaths. For example datapaths *predcom* drives datapath *outmindp* in Figure 16. Looking at Figures 17 and 18, we note that the driving signals from datapath *predcom* are all outputs of registers. This implies we can compose the datapaths with impunity. The other interactions between the remaining datapaths are also handled quite easily.

7.8.5 Synthesis Results

There are mild constraints on the synthesis of the different datapaths and controllers which to a large extent were followed in the manual design [?] as well.

All the datapaths have to be synthesized separately, as in the manual design. The adders are carry lookahead adders with 4 bits of lookahead. Since each adder is 12 bits wide, three such modules are required. A 4-bit carry lookahead adder can be synthesized to be fully `RPDFT` with minimal area overhead in the combinational portion. The finite state machine controller was specified as a State Transition Graph (with 16 states). The control logic was flattened and algebraically-factored to produce a fully `RPDFT` circuit. Algebraic factorization in this case involved no area penalty. Some comparator cells (*e.g.* [?]) are not fully testable when hierarchically composed. Alternate designs that are fully `RPDFT` are presented in [?].

We chose to perform $A > B$ simply by checking $A - B > 0$ (Similarly for $A < B$). This can be done by implementing a subtractor and inspecting the final borrow bit. A subtractor, like an adder, can be composed in a bit-sliced design for full `RPDFT`. A maximum function can be based on this design as well. The subtractor cell in the comparator does not have to implement the sum output – we are only interested in the final borrow. However, for the maximum function, we have an output for each cell i representing the i-th bit of $MAX(A,\ B)$.

Thus the entire chip can be synthesized to be fully path-delay-fault testable.

The adder and comparator circuits used in the manual design [?] are *not* fully path-delay-fault testable. This implies that the manual design has *very* low robust path-delay-fault testability ($<$ 20%). Testability can currently be increased to 100% with $\approx$ 10% area overhead in the combinational logic. Over 46 million paths exist in the synthesized processor.

7.9 Synthesis for Testability Under Other Fault Models

Robust path-delay-fault testability is the most stringent fault model that is currently known, and it implies transistor stuck-open fault and gate-delay-fault testability. For this reason, we have concentrated on the synthesis of completely RPDFT circuits in this paper.

Restrictions on the synthesis procedures can be relaxed if one is only interested in say, transistor stuck-open or gate-delay-fault test coverage. Composition rules for regular structures that are less restrictive than the topology shown in Section 7.1, but which maintain robust stuck-open and gate-delay-fault testability were presented in [?]. Some restrictions on logic optimization procedures presented here can be removed, in the case of targeting stuck-open and gate-delay-fault testability. For instance, one can algebraically factor a multiple-output minimized, rather than single-output minimized cover.

The test vector sets to detect transistor stuck-open and gate delay faults are typically substantially smaller in size than the path-delay-fault test set. Given a robustly path-delay-fault testable circuit, path selection procedures (*e.g.* [?]) can be used to obtain a minimal set of test vectors which detect faults under the less restrictive dynamic and static fault models. In the Viterbi processor for instance, the set of test vectors to detect all transistor stuck-open faults, gate delay faults and single stuck-at faults is less than $10,000$.

8 Conclusions

Traditional design for testability approaches do not typically result in circuits that are highly RPDFT. Also, test sets providing high fault coverage for path delay faults and multifaults are not expected from traditional automatic test-pattern generation procedures. Based on an understanding of the sources of untestability in logic circuits, we have developed and applied synthesis procedures that can be used to produce large, yet fully RPDFT, integrated circuits.

In [?, ?] a set of necessary and sufficient conditions for robust path-delay-fault testability and robust gate-delay-fault testability was developed. In this paper we have used these results to develop synthesis and optimization procedures for robustly delay-fault testable two-level and multilevel circuits. Our synthesis procedure involves a novel orchestration of Boolean covering, unate decomposition, algebraic factorization, and tree-covering-based technology mapping. Experimental results presented in this paper indicate that these synthesis procedures can be used to dramatically increase path-delay and gate-delay-fault testability. This synthesis approach results in much more area efficient circuits than the unate decomposition procedure described in [?], although it can benefit from the use of that procedure as a "subroutine". In our synthesis procedures a high degree of testability does not come at a significant area penalty, in fact our results indicate that these synthesis procedures may produce circuits that are actually smaller than those used by more typical logic optimization methods.

Our procedure relies on an underlying theory to synthesize circuits that are guaranteed to be highly/fully robust delay-fault testable. In contrast, Roy *et al.* in [?] take a more *post facto* analytical approach to the problem requiring time consuming path analysis of an existing circuit. When untestable paths are found in a sub-circuit the only recourse is to flatten the sub-circuit to two levels, minimize, and use unate decomposition as in [?]. Our approach allows us to exploit the fact that algebraic factorization procedures retain delay-fault testability to synthesize more area efficient circuits.

In this paper we also developed results for cascaded combinational logic circuits that cannot be flattened. We developed a sufficiency condition for composing `RPDFT` circuits and applied this to proving the `RPDFT` of regular structures such as parity trees and adders. We showed how these regular structures can be cascaded with other combinational logic modules while retaining `RPDFT`. In our initial experiments, we were able to synthesize a data encryption chip, a small μ-processor and a speech recognition chip to be completely `RPDFT`. Further, test vectors were obtained as a by-product of the synthesis process.

Two significant problems still remain before complete robust coverage for path delay faults can be considered practical for any large class of circuits. The first requirement is to reduce the size of the unmanageably large test sets that guarantee complete `RPDFT`. One solution is to use path selection procedures to choose a set of paths that cover each gate and each net in the circuit at least once as described in Section 7.9. Applying the `RPDFT` test set for even this dramatically reduced set of paths covers all gate delay faults and all transistor stuck-open faults.

A second problem is to find a more economic scan structure for robustly applying the path-delay-fault tests. One alternative here is to apply the `RPDFT` vectors non-robustly using a standard scan structure.

Because of the invariance and compositionality of `RPDFT`, we argue that `RPDFT` is a desirable circuit-testability property even if a set of test vectors for complete path-delay-fault coverage cannot be robustly applied. We further claim that even non-robustly applying delay tests for a minimal set of net and gate covering paths, which is quite feasible given current scan structures and required vector set sizes, would bring practical circuits to an unprecedented degree of defect coverage.

9 Acknowledgements

Thanks to Michael Bryan for help with the synthesis experiments. Thanks to Dave Gregory and Rick Rudell whose criticisms led to the undertaking of the work described here. Thanks to Vishwani Agrawal, Vijay Iyengar and Tom Williams whose conversations forced us to sharpen our motivation for working on robust path-delay-fault testability. This work was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

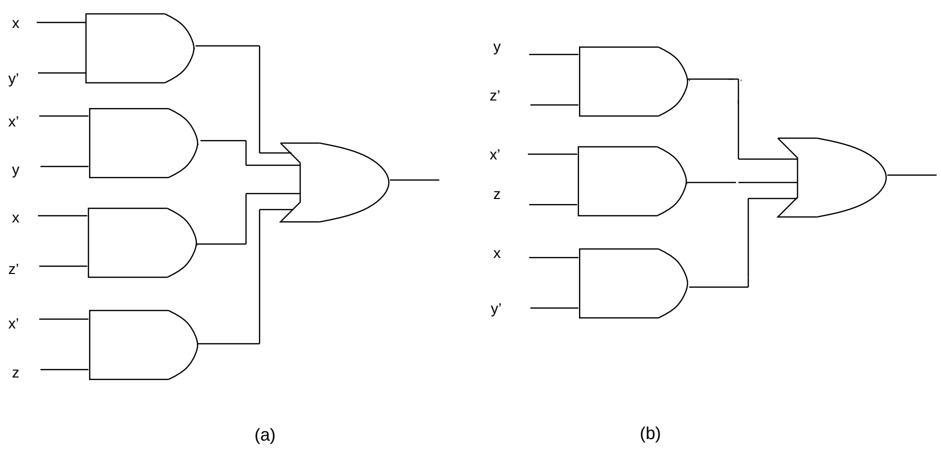

Figure 1: Different Prime and Irredundant implementations of a logic function

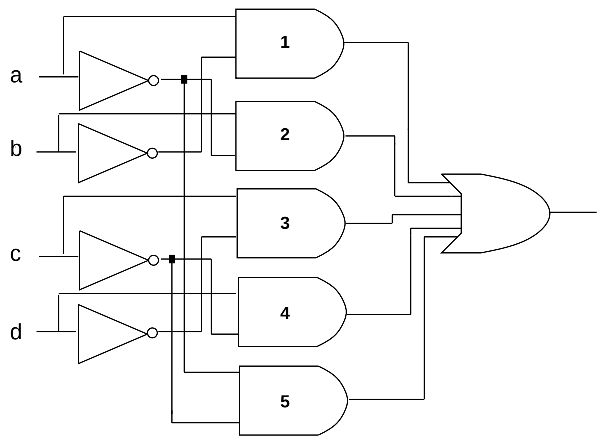

Figure 2: A Prime and Irredundant Two-Level Cover that is Not Robustly Gate Delay Fault Testable

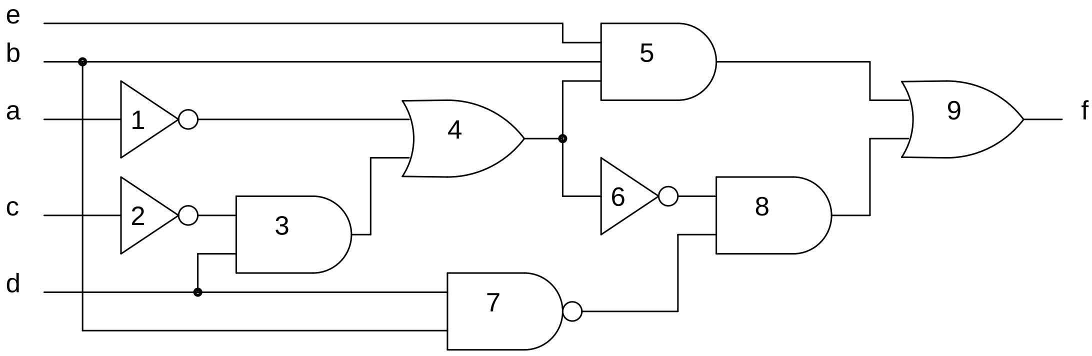

Figure 3: Example multilevel circuit to illustrate the ENF

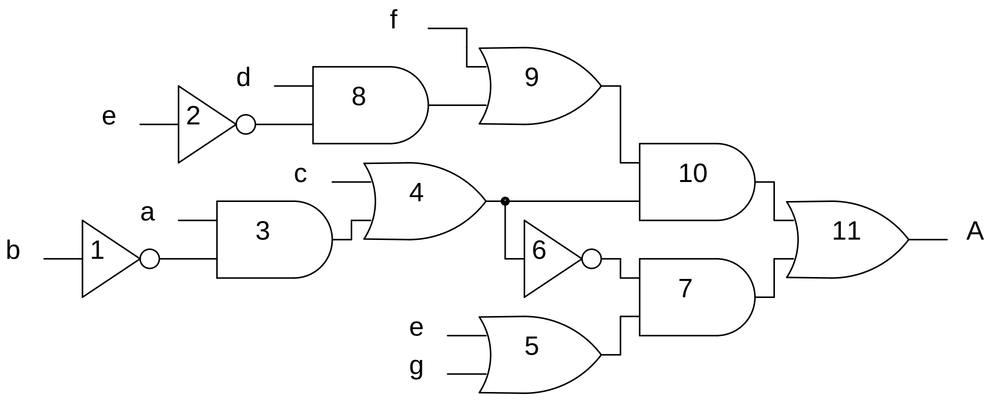

Figure 4: Algebraic factorization example using inverse

EX	#inp	#out	#init. prods
rd53	5	3	31
rd73	7	3	147
life	9	1	140
dekoder	4	7	10
ex5	8	6	200
lin.rom	7	36	128
luc	8	27	27
m2	8	16	96
m3	8	16	128
m4	8	16	256
max128	7	24	128
max512	9	6	512

Table 1: Statistics of PLA Benchmark Examples

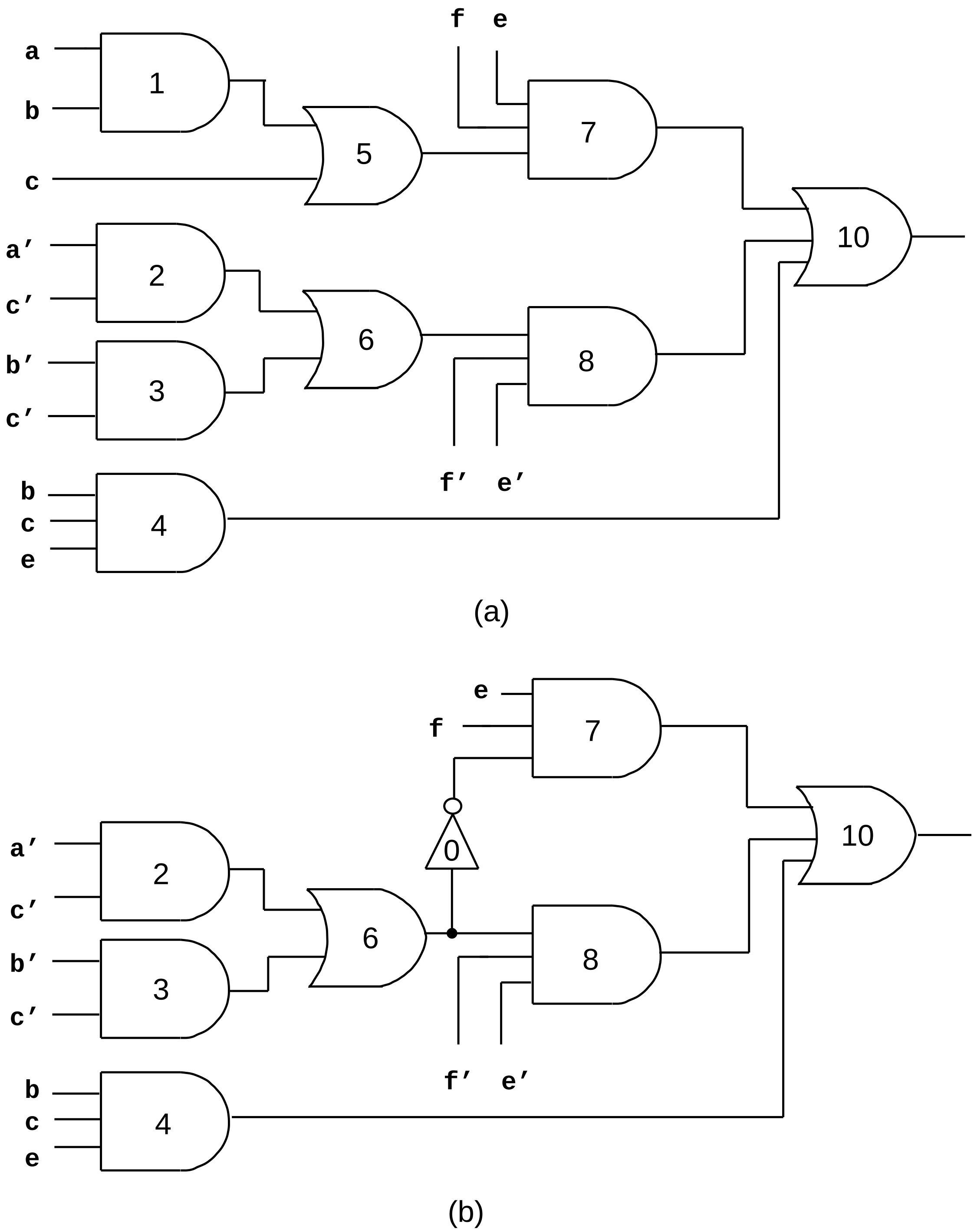

Figure 5: Algebraic Factorization Using the Complement Does Not Preserve Testability

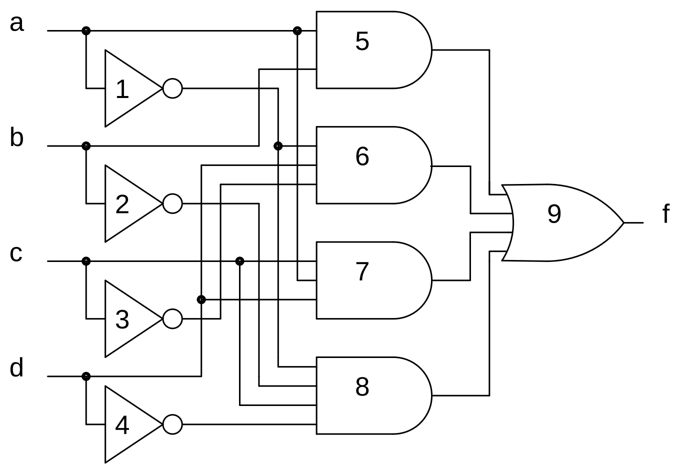

Figure 6: Two-level fully path-delay-fault testable circuit

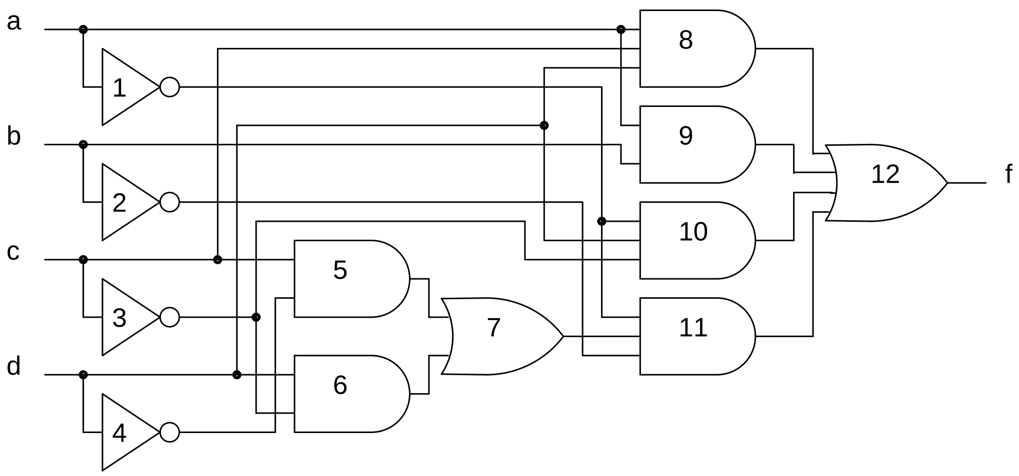

Figure 7: Untestable circuit resulting from Boolean factorization

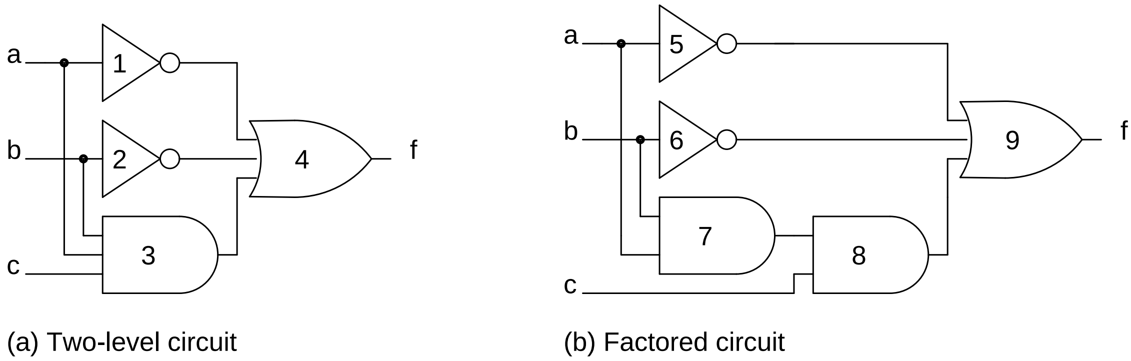

Figure 8: Algebraic factorization lowering gate-delay-fault testability

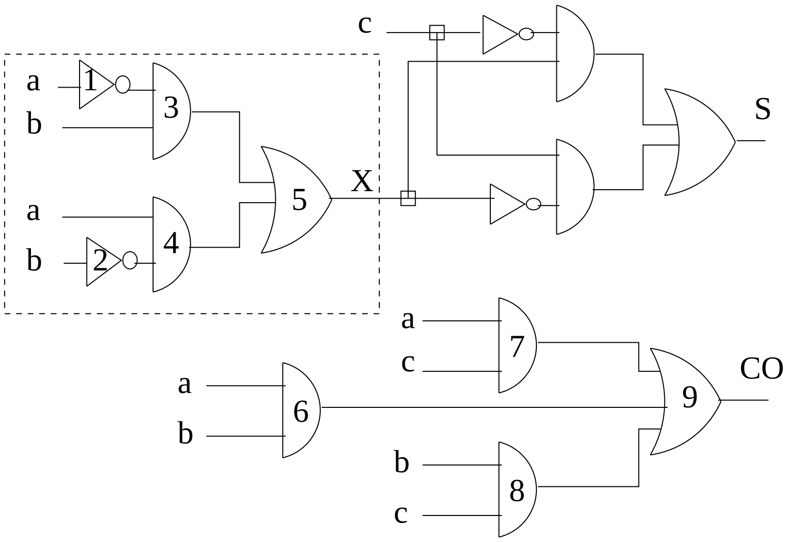

Figure 9: Full Adder

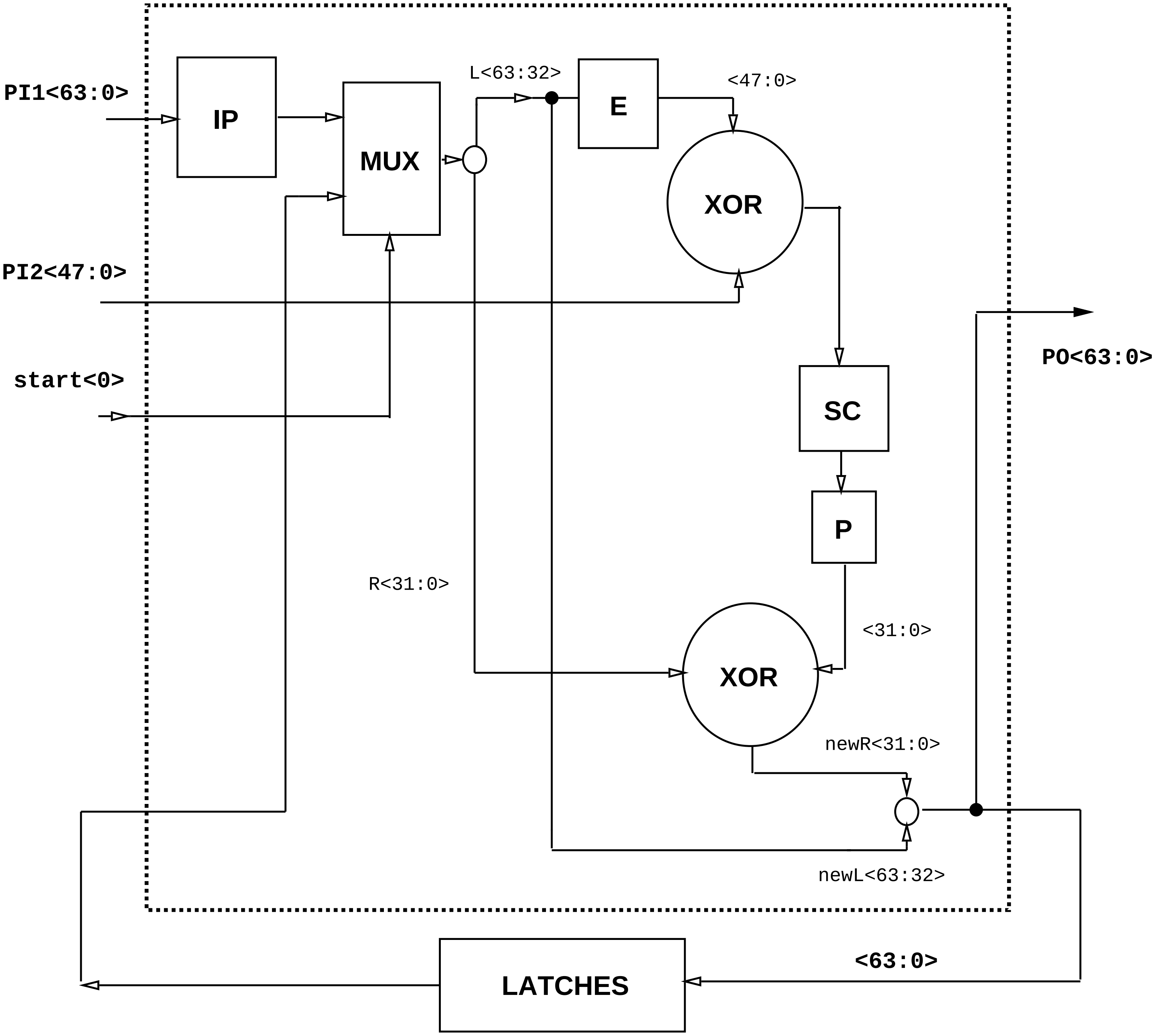

Figure 10: One Stage of the DES Algorithm

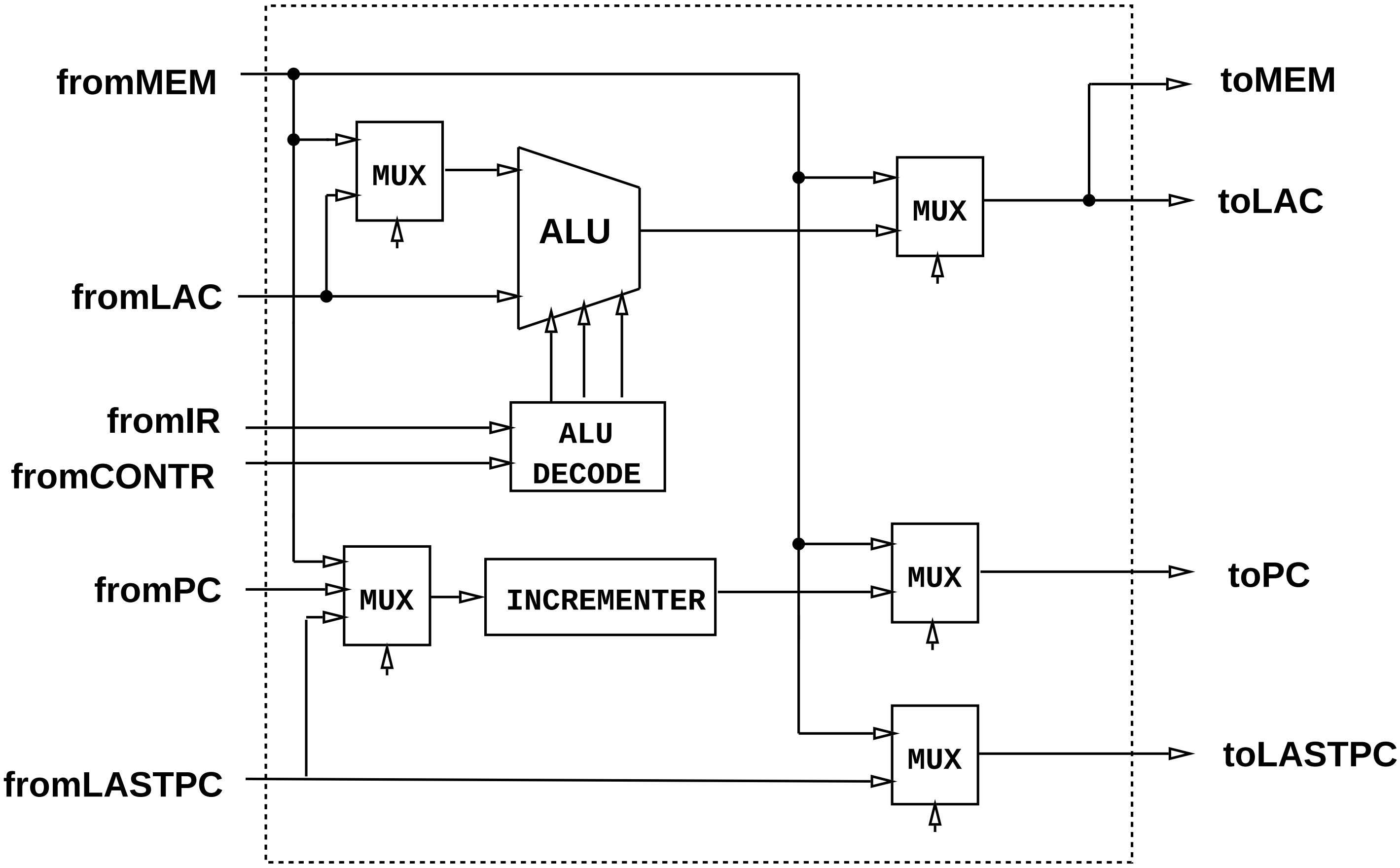

Figure 11: PDP-8 Datapath

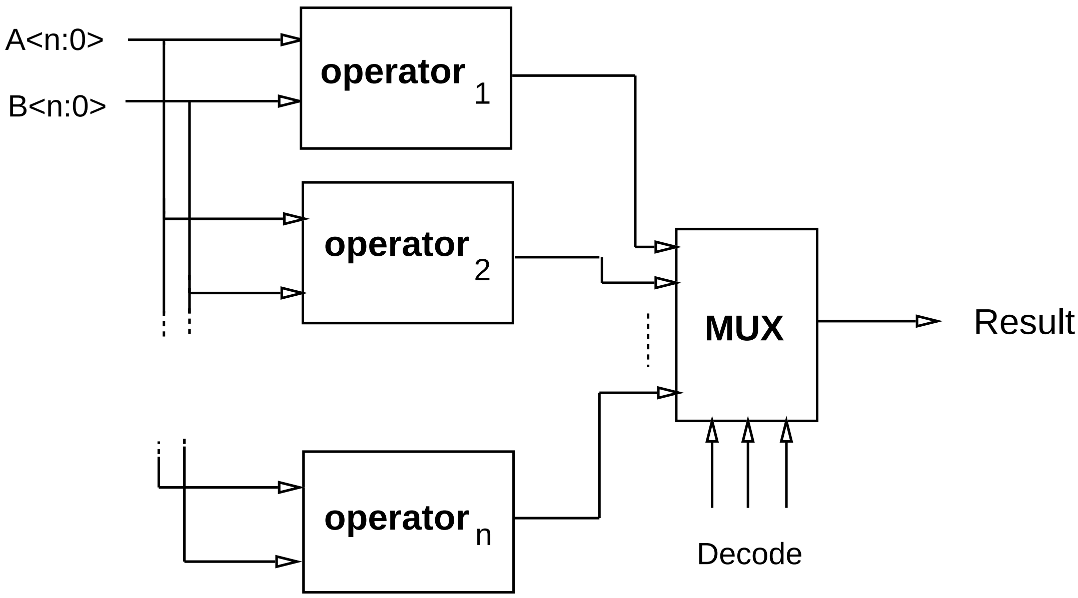

Figure 12: Arithmetic Unit and Decoder

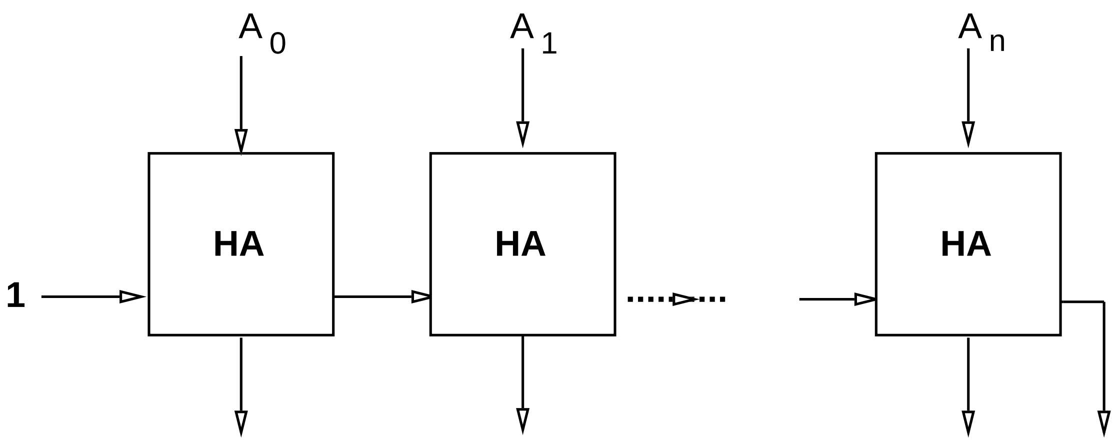

Figure 13: An Incrementer

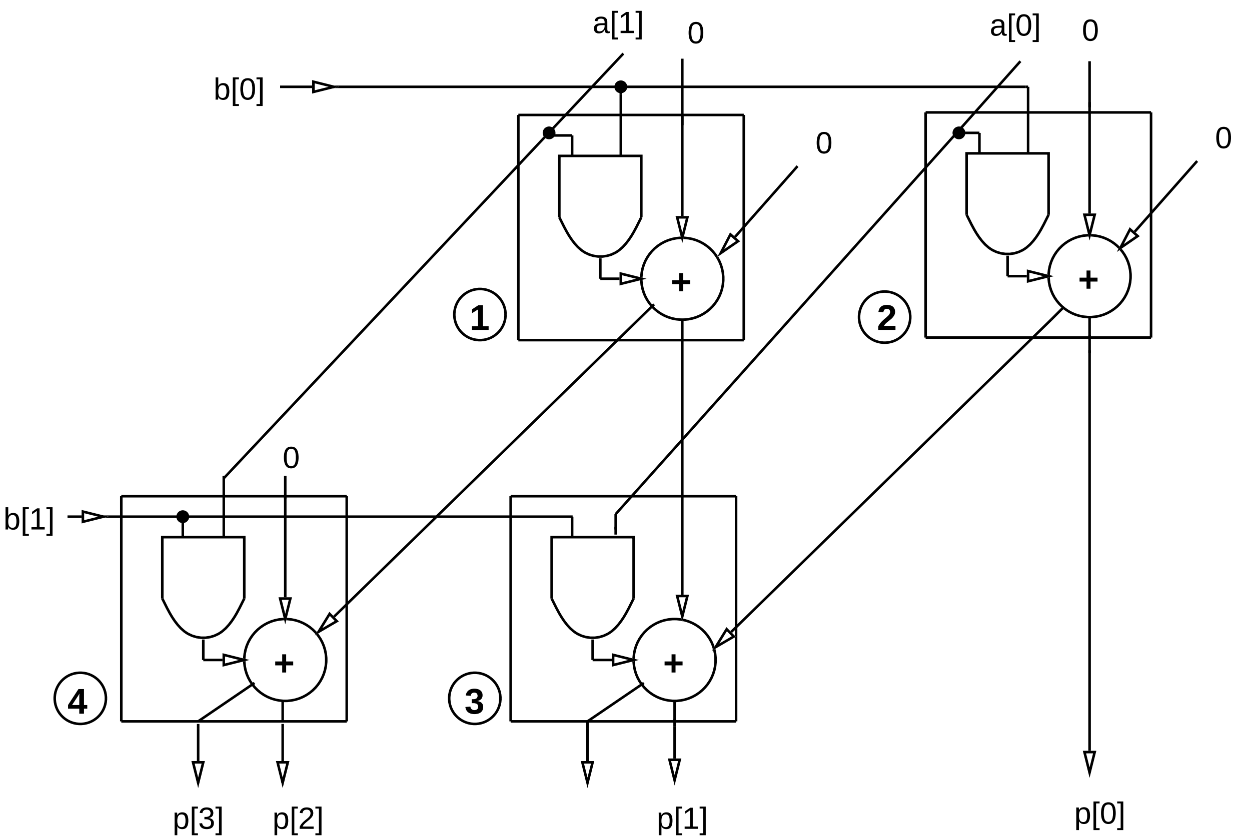

Figure 14: A 2 x 2 Parallel Multiplier

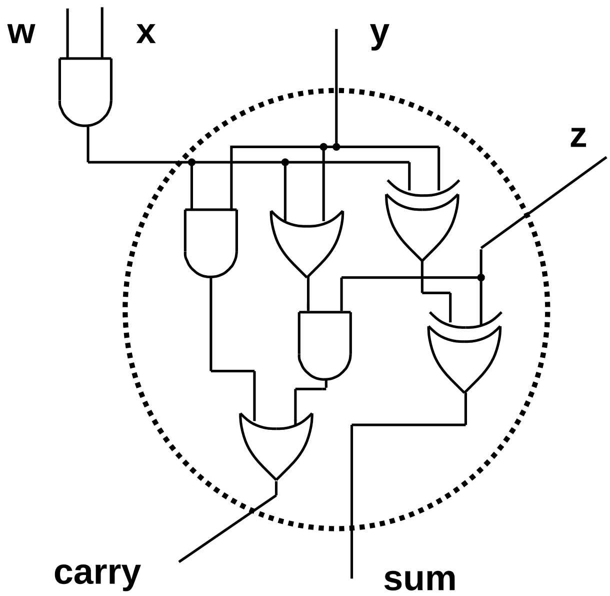

Figure 15: Basic Cell for a Multiplier

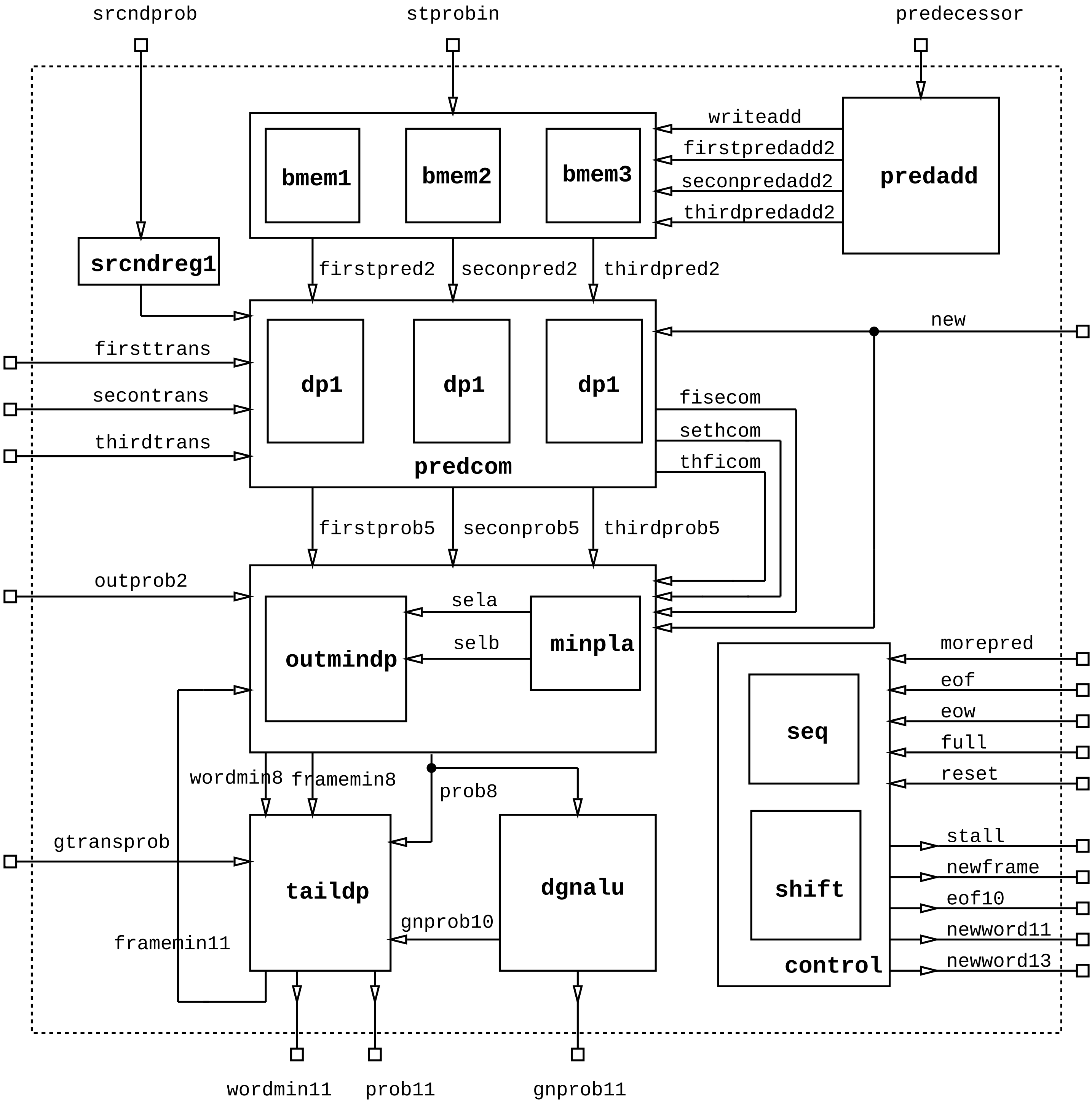

Figure 16: The Viterbi Processor

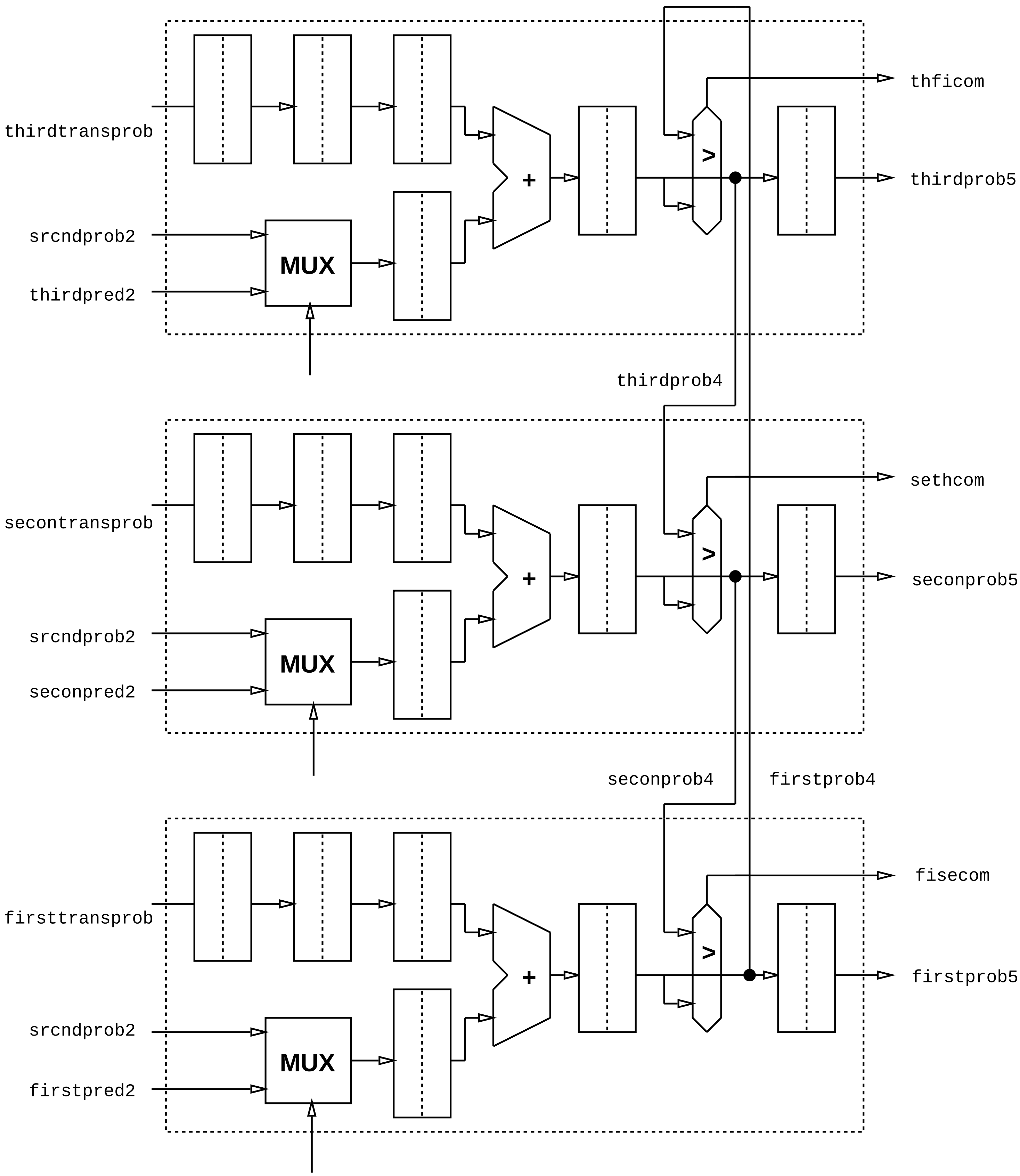

Figure 17: The predcom Datapath

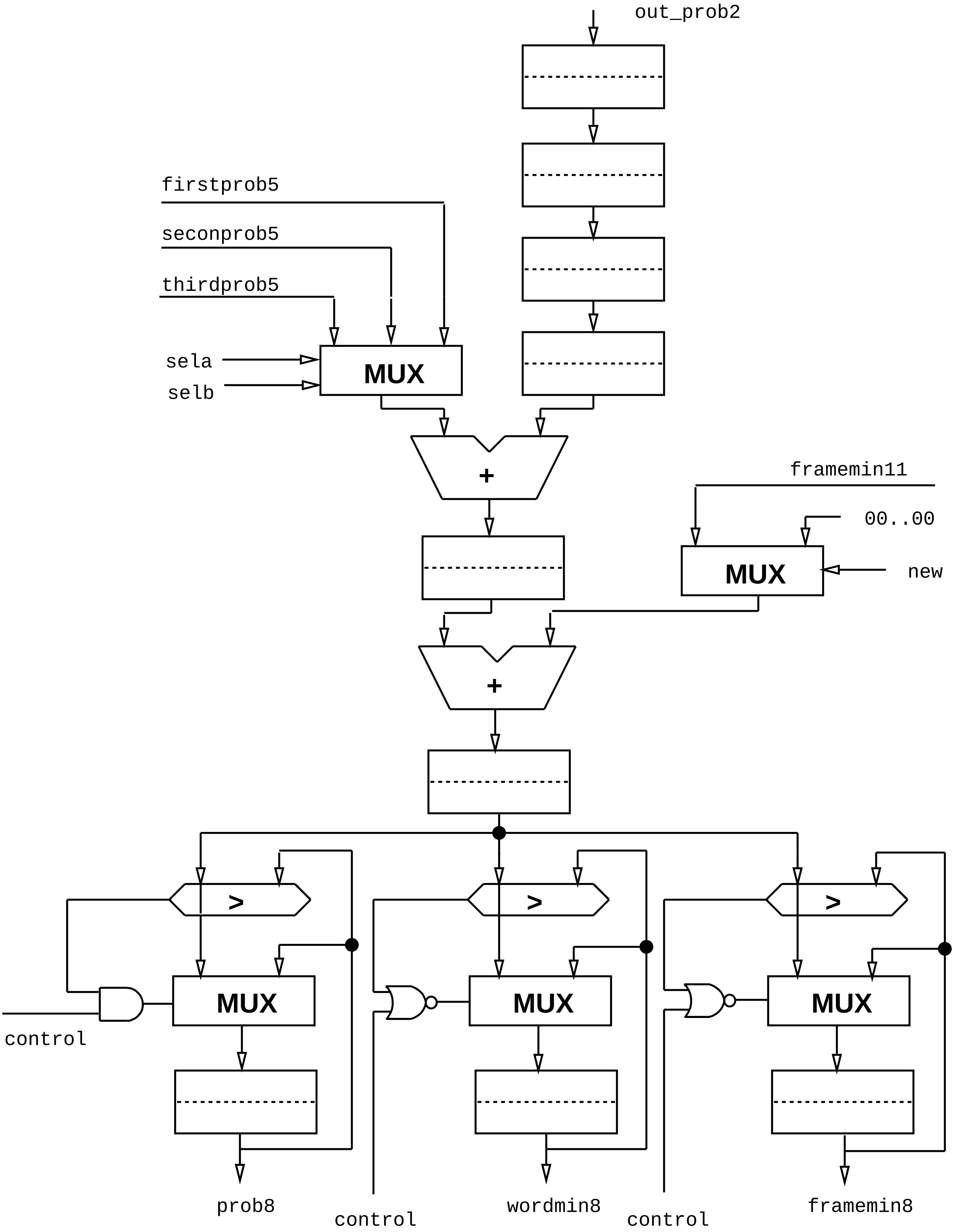

Figure 18: The outmindp Datapath

EX	MULTI-OP		SINGLE-OP		COVER	
	#prod	fcov.	#prod	fcov.	#prod	fcov.
rd53	31	83.75	31	100.0	31	100.0
rd73	127	88.39	141	94.76	141	95.23
life	84	79.16	84	79.16	84	91.00
dekoder	9	72.88	25	97.61	15	97.91
ex5	74	15.21	304	89.92	176	89.92
lin.rom	128	33.00	456	93.79	387	93.79
luc	26	66.10	158	98.45	63	98.45
m2	47	44.66	106	95.15	91	95.42
m3	62	41.94	133	94.60	120	95.14
m4	101	36.65	211	93.31	189	93.31
max128	78	31.39	194	90.22	170	90.22
max512	133	73.45	167	91.08	164	91.45

Table 2: Results using Covering Procedure

example	in	out	multiple-output			single-output		
			pro	gate	path	pro	gate	path
Z5xp1	7	10	65	98.5	70.9	74	100.0	98.3
Z9sym	9	1	86	100.0	95.3	86	100.0	95.3
add4	9	5	135	100.0	100.0	135	100.0	100.0
add6	12	7	355	100.0	100.0	355	100.0	100.0
addm4	9	8	200	96.2	88.0	213	100.0	95.9
adr4	8	5	75	100.0	100.0	75	100.0	100.0
al2	16	47	66	100.0	91.9	89	100.0	99.5
bc0	26	11	179	96.7	72.5	487	100.0	92.3
bcd.div3	4	4	9	100.0	88.6	11	100.0	100.0
dist	8	5	123	95.2	83.3	152	100.0	92.3
f51m	8	8	77	96.3	94.7	76	100.0	96.6
l8err	8	8	52	94.5	86.1	58	100.0	92.4
life	9	1	84	100.0	79.2	84	100.0	79.2
log8mod	8	5	38	97.7	84.4	52	100.0	92.8
m181	15	9	42	96.7	82.9	53	100.0	95.2
m2	8	16	47	100.0	45.9	106	100.0	95.2
m3	8	16	62	99.1	41.9	133	100.0	94.6
m4	8	16	105	97.0	40.5	211	100.0	93.3
max512	9	6	133	94.1	73.5	167	100.0	91.1
mlp4	8	8	128	98.8	89.0	144	100.0	97.1
par8	8	1	128	100.0	100.0	128	100.0	100.0
pla4	31	90	147	99.9	84.8	695	100.0	98.6
radd	8	5	75	100.0	100.0	75	100.0	100.0
rckl	32	7	32	100.0	61.1	97	100.0	100.0
rd53	5	3	31	85.1	83.8	31	100.0	100.0
rd73	7	3	127	90.0	88.4	141	100.0	94.8
root	8	5	57	89.7	71.8	71	100.0	90.2
sqr6	6	12	49	98.5	81.0	58	100.0	92.8
sym10	10	1	210	100.0	70.6	210	100.0	70.6
t3	12	8	33	99.5	99.1	33	100.0	99.1
x1dn	27	6	110	100.0	100.0	110	100.0	100.0
z4	7	4	59	100.0	100.0	59	100.0	100.0

Table 3: Testability and area comparisons of two-level implementations

example	alg	alg-i	bool	example	alg	alg-i	bool
Z5xp1	169	152	154	m3	349	339	383
Z9sym	238	238	213	m4	530	514	604
add4	90	56	76	max512	481	468	467
add6	126	96	97	mlp4	377	353	384
addm4	507	476	501	par8	52	28	28
adr4	74	48	59	pla4	1462	1444	1079
al2	156	155	155	radd	74	48	57
bc0	1178	1132	769	rckl	214	195	197
bcd.div3	24	24	24	rd53	67	59	36
f51m	184	167	120	rd73	165	137	65
dist	423	413	345	root	184	169	158
l8err	178	170	135	sqr6	152	148	150
life	81	80	73	sym10	310	317	348
log8mod	155	152	120	t3	100	98	86
m181	100	97	92	x1dn	102	101	89
m2	266	253	290	z4	66	42	51

Table 4: Literal comparisons of multilevel implementations

example	algebraic		algebraic-inv		Boolean	
	gate	path	gate	path	gate	path
Z5xp1	100.0	98.9	99.3	98.9	90.1	49.7
Z9sym	100.0	95.9	100.0	95.9	99.2	97.7
add4	100.0	100.0	100.0	100.0	84.4	74.3
add6	100.0	100.0	100.0	100.0	72.3	40.3
addm4	99.5	97.6	100.0	97.3	80.5	64.9
adr4	100.0	100.0	100.0	100.0	76.4	56.5
al2	100.0	99.6	100.0	99.6	89.4	78.5
bc0	98.9	94.8	99.3	94.8	88.8	74.0
bcd.div3	100.0	100.0	100.0	100.0	83.6	65.1
dist	99.0	95.3	98.8	94.5	56.2	33.5
f51m	99.3	97.7	99.3	97.6	54.8	24.1
l8err	97.3	93.9	98.1	95.3	95.4	80.9
life	98.4	88.1	97.2	89.1	96.5	87.1
log8mod	98.3	95.9	98.4	95.6	70.3	23.1
m181	100.0	100.0	100.0	100.0	91.4	76.7
m2	100.0	96.7	100.0	96.4	78.3	43.8
m3	99.6	96.8	99.3	95.7	67.7	40.7
m4	99.2	95.1	98.4	94.7	73.2	41.7
max512	98.6	93.2	97.9	93.3	93.6	72.4
mlp4	100.0	97.0	99.7	98.2	99.4	91.2
par8	100.0	100.0	100.0	100.0	100.0	100.0
pla4	99.7	99.1	99.8	99.0	97.6	78.4
radd	100.0	100.0	100.0	100.0	88.3	44.0
rckl	100.0	100.0	100.0	100.0	87.5	61.4
rd53	100.0	100.0	100.0	100.0	76.4	33.2
rd73	100.0	94.8	100.0	93.4	92.7	79.5
root	99.3	94.1	99.3	94.1	92.1	78.2
sqr6	100.0	95.3	100.0	95.3	70.6	43.7
sym10	87.0	79.6	89.1	77.6	80.0	71.1
t3	100.0	100.0	100.0	100.0	75.8	48.1
x1dn	100.0	100.0	100.0	100.0	100.0	100.0
z4	100.0	100.0	100.0	100.0	100.0	83.4

Table 5: Testability comparisons of multilevel implementations

UNCONSTRAINED			TESTABLE			
#lit	#path	fault cov.	#lit	#path	fault cov.	CPU time
2944	90,112	$< 25\%$	3102	98,465	100%	2.2h

Table 6: Results for the Data Encryption Standard Chip

UNCONSTRAINED			TESTABLE			
#lit	#path	fault cov.	#lit	#path	fault cov.	CPU time
1542	541,023	$< 15\%$	1706	661,012	100%	3.1h

Table 7: Results for the PDP-8 Microprocessor

UNCONSTRAINED			TESTABLE			
#lit	#path	fault cov.	#lit	#path	fault cov.	CPU time
1941	45×10^6	$< 25\%$	2215	46×10^6	100%	2.2h

Table 8: Results for the Viterbi Processor