False-Noise Analysis Using Logic Implications

A. GLEBOV, S. GAVRILOV,
Motorola, Inc., Moscow, Russia
D. BLAAUW
University of Michigan, Ann Arbor
and
V. ZOLOTOV
Motorola, Inc., Austin, TX

Cross-coupled noise analysis has become a critical concern in today's VLSI designs. Typically, noise analysis makes the assumption that all aggressing nets can simultaneously switch in the same direction. This creates a worst- case noise pulse on the victim net that often leads to false noise violations. In this article we present a new approach that uses logic implications to identify the maximum set of aggressor nets that can inject noise simultaneously under the logic constraints of the circuit. We propose an approach to efficiently generate logic implications from a transistor-level description and propagate them in the circuit using ROBDD representations. We propose a new method for lateral propagation of implications and also show how tristate gates and high-impedance signal states can be handled using tristate implications. We then show that the problem of finding the worst-case logically feasible noise can be represented as a maximum weighted independent set problem and show how to efficiently solve it. Initially, we restrict our discussion to zero-delay implications, which are valid for glitch-free circuits, and then extend our approach to timed implications. The proposed approaches were implemented in an industrial noise analysis tool and results are shown for a number of industrial test cases. We demonstrate that a significant reduction in the number of noise failures can be obtained from considering the logic implications as proposed in this article, underscoring the need for false-noise analysis.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*VLSI* (*very large scale integration*); B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault Tolerance; B.6.3 [**Logic Design**]: Design Aids

General Terms: Design, Reliability, Verification, Theory, Algorithm, Performance

Additional Key Words and Phrases: VLSI (very large scale integration), noise analysis, circuit logic

1. INTRODUCTION

Advances in process technology have greatly increased the coupling capacitance in VLSI interconnects making it common for as much as 60 to 80% of

Authors' addresses; A. Glebov, S. Gavrilov, V. Zolotov, Motorola, Inc., 7700 W. Parmer Lane, Building C, MD: 78729, Austin, TX 78729; email: vladimir.zolotov@motorola.com.; D. Blaauw, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1301 Beal Ave., Ann Arbor, MI 48109-2122; email: blaauw@umich.edu.

interconnect capacitance to be coupling capacitance to other nets. This trend has led to an increase in the noise injected on a net due to the unanticipated switching of neighboring nets, creating the necessity for noise analysis tools.[1] In noise analysis, the net under consideration is commonly referred to as the *victim net*, and the neighboring nets that inject noise are referred to as *aggressor nets*. A victim net with its associated aggressor nets is referred to as a *noise cluster*. A *functional noise failure* is said to occur when a victim net is in a quiescent state while its aggressor nets switch, creating a noise pulse injected on the victim that could potentially be latched. A *delay noise failure* is said to occur if the victim net transitions at the same time as the aggressor nets, decreasing or increasing the delay of the victim net depending on the direction of the aggressor switching, and potentially creating a timing violation.

Noise analysis tools typically make the assumption that all aggressor nets switch at the same time and in the same direction. Under this assumption, the noise injected from each aggressor combines, creating the maximum possible composite noise pulse on the victim net and yielding a conservative analysis. In practice, however, the timing and logic constraints present in the circuit may prevent all aggressors from switching in the same direction at the worst possible alignment time. Therefore, the noise reported by an analysis that does not account for timing and logic correlations can severely overestimate the actual noise realizable on a victim net and can create a so-called *false noise violation*. This is especially important when the number of aggressors for a victim is high (e.g., 10 or more), as is often the case. In such a situation the combined noise from all aggressors will be very severe, whereas the likelihood of realizing the simultaneous switching for all aggressors is small due to inherent logic and timing correlations.

Industrial noise analysis approaches have exploited timing correlations in circuits to reduce the pessimism of noise analysis by identifying situations where two aggressor nets cannot switch at the same time. A common example of such a situation is when two aggressor nets switch in different clock cycles, or where one switches very early and the other very late in the same clock cycle. To determine when a net can switch, so-called *switching windows* are propagated in the circuit using static timing analysis [Shepard 1998; Levy et al. 2000]. After switching windows are identified for each aggressor, the possibility of overlap between timing windows for a set of aggressors is determined. It is important to note that this approach is local in nature, meaning that the switching windows are identified separately for each aggressor net, making this analysis very efficient. However, this also results in a weakness of this approach, in that it does not identify situations where a pair of aggressor nets can each switch individually at a particular time but cannot both switch at that time due to logic relationships in the circuit. A simple example of this situation is shown in Figure 1(a). Also, the timing window-based approach does not identify cases where nets cannot switch in the same direction, for instance,

[1]Please see Zurada et al. [1989], Shepard et al. [1997], Shepard [1998], Levy et al. [2000], Xue et al. [1994], Vittal et al. [1999], Tohr et al. [1998], Vittal and Marek-Sadowska [1997], and Zhou and Wong [1998].

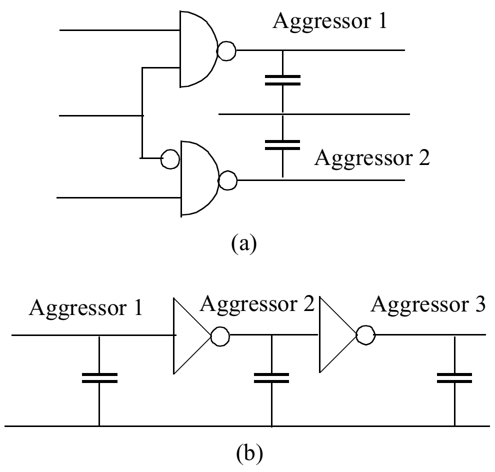


Fig. 1. (a) Logic and (b) timing relationships among aggressors.

when they are connected by an inverter as shown in Figure 1(b). Therefore, the timing window approach may not identify all false noise failures, although it has been shown in practice to be relatively effective [Levy et al. 2000].

In order to identify all false noise failures, both the timing and logic correlations of the circuit must be taken into account. In Chen and Keutzer [1999] it was shown that, in general, this problem can be represented as a search for a worst-case two-vector test using a Boolean constraint optimization problem formulation. In Kirkpatrick and Sangiovanni-Vincentelli [1996], a method based on compatible observability don't care sets was proposed. In Rubio et al. [1997], a method was proposed using a test pattern generation approach. However, all these methods have very high complexity and cannot be applied to large problem sizes. Since noise primarily occurs in top-level routes, it is critical to perform false noise analysis globally for large designs and hence, heuristic methods must be employed.

In this article, we present a new approach for false noise analysis based on the generation and propagation of logic implications between signal pairs. Logic implications [Brown 1990] have been widely used in logic synthesis [Hachtel et al.1988; Kunz and Menon 1994; Bahar et al. 1996; Long et al. 2000] as well as in peak current estimation [Bobba and Hajj 1998], although they have not until now been proposed for false noise analysis. The input to our analysis is a transistor-level description of the circuit although the analysis can also be performed at the gate level, using precharacterization of transistor-level leaf cells. We show how pairwise logic implications can be efficiently generated using ROBDD representations of the DC-connected components in the circuit. The generated pairwise implications are then propagated in the circuit through forward and backward topological traversals. We also propose a new method to generate so-called lateral implications and methods for handling tristate gates and high-impedance signal states using tristate implications.

Given the logic implications between the aggressor nets of a noise cluster, we show that the problem of finding the subset of aggressor nets which induce the

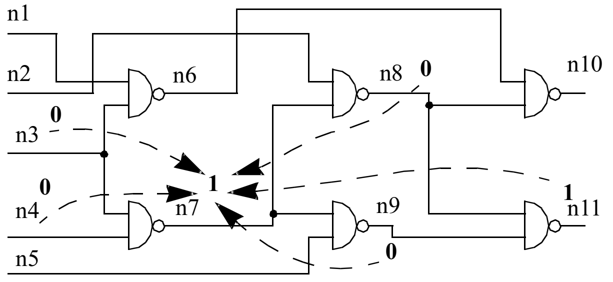


Fig. 2. Example of SLIs in simple circuit.

maximum noise on the victim under the constraints of these logic implications can be represented by a constraint graph. We then show that this problem can be solved by solving the maximum weighted independent set problem for this constraint graph. Although this is an NP-hard problem, the number of aggressors coupling to a victim, and hence the size of the constraint graph, is typically small, allowing for an exact solution of the problem. Since the logic implications only capture pairwise relationships, the overall approach remains heuristic and very efficient, capable of analyzing large designs in a few hours.

The initial formulation presented in this article uses zero-delay implications that are valid only during the stationary state of the circuit before and after all transitions occur. Hence, this formulation for false noise analysis is conservative only for glitch-free circuits, obtained, for instance, through special transistor sizing approaches [Wroblewski et al. 2000]. In the last section of the article we show how our analysis can be extended for timed implications that are valid at all points during the operation of a circuit. The proposed approaches were implemented and used in an industrial noise analysis tool called ClariNet. Results are presented for a number of industrial test cases. It is shown that the total number of noise failures is reduced by up to 47% using our proposed approach.

The remainder of the article is organized as follows. Section 2 discusses the generation and propagation of logic implications, including generation logic implications for tristate logic. Section 3 shows how to use logic implications in false noise avoidance. Section 4 presents extensions of the algorithm for timed implications. Section 5 presents results, and in Section 6 we draw our conclusions.

2. COMPUTING LOGIC IMPLICATIONS

We use the following notation for simple logic implications (SLI) between two circuit nodes a and b.

$(a = V_a) \rightarrow (b = V_b)$, where $V_a, V_b \in \{0, 1\}$, meaning that if node a is at logic value V_a the resulting value on node b will be V_b. Figure 2 shows a small example circuit where $n3 = 0$ implies that node $n7 = 1$. In total, this example circuit has 26 nontrivial SLIs, where a trivial SLI is an implication such as $(a = V_a) \rightarrow (a = V_a)$. Similar to Bobba and Hajj [1998], we store the implications for a node in one of four implication lists: H^a_H, H^a_L, L^a_H, L^a_L, where implication

$(b=1) \rightarrow (a=0)$ would belong to implication list L_H^a at node a; that is, $b \in L_H^a$. In Figure 2, for example, the H_L^{n7} implication list at node $n7$ is $\{n3, n4, n8, n9\}$ and the implication list H_H^{n7} is $\{n11\}$.

The SLI generation algorithm consists of two steps: SLIs are generated as explained in Sections 2.1 and 2.2, and SLI are propagated through the circuit using the basic operations of list union, list intersection, and contrapositive law as explained in Section 2.3. Finally, in Section 2.4 we show how to extend our analysis for tristate gates.

2.1 Generation of SLIs for Simple Gates

We first consider how to generate the initial SLIs for a gate with inputs a_i and output x. We start our discussion with some general properties about SLIs from gate input nodes to gate output nodes and vice versa.

Property 1: *The implication $(a=V_a) \rightarrow (x=V_x)$ is equivalent to implication $(x=\bar{V}_x) \rightarrow (a=\bar{V}_a)$ due to the contrapositive law, and we consider both implications as a single implication at the input a_i.*

Property 2: *The presence of an SLI $(a_i=V_i) \rightarrow (x=V_x)$ at input a_i of a gate means that this input is a controlling input with controlling value V_i.*

Property 3: *Since a gate input a_i can take one of two logic values, there can be no more than two SLIs at a_i. The presence of two SLIs at a gate input implies that the gate is one of the trivial cases:*

(1) if $(a=V_a) \rightarrow (x=V_x)$ and $(a=\bar{V}_a) \rightarrow (x=V_x)$ then x is Boolean constant;
(2) if $(a=V_a) \rightarrow (x=V_x)$ and $(a=\bar{V}_a) \rightarrow (x=\bar{V}_x)$ or $(a=V_a) \rightarrow (x=\bar{V}_x)$ and $(a=\bar{V}_a) \rightarrow (x=V_x)$, then either $x=a$ or $x=\bar{a}$ and x has no dependence on other gate inputs.

It follows that for any nontrivial multi-input logic gate each input has at most one SLI. If a gate has multiple SLIs at its inputs, all these SLIs must have the same value of V_x, as stated in the following lemma.

Lemma 1. *Consider a gate G with inputs a_i, where $i=1,\ldots,n$ and output x, implementing a Boolean function. If there are SLIs $(a_i=V_i) \rightarrow (x=V_x)$ at inputs a_i then all these SLIs must have the same value of V_x.*

Proof. Consider SLIs $(a=V_a) \rightarrow (x=V_x)$, $(b=V_b) \rightarrow (x=\bar{V}_x)$ at inputs a and b and consider the input combination: $a=V_a$, $b=V_b$. In this case, the first SLI implies that $x=V_x$ while at the same time the second SLI implies that $x=\bar{V}_x$, which is clearly a conflict. □

Based on Properties 1 to 3 and Lemma 1, we can now examine how the Boolean function of a gate is defined by its input SLIs.

Theorem 1. *Let the gate G with inputs $a_i, i=1,\ldots,n$ and output x, implement a Boolean function. The set of nontrivial SLI $(a=V_a) \rightarrow (x=V_x)$ for inputs*

$a_1, \ldots, a_m$, where $1 \le m \le n$, is equivalent to the definition of its Boolean function as

$$p_x = p_{a1} \vee \ldots \vee P_{am} \vee \bar{p}_{a1} \wedge \ldots \wedge \bar{p}_{am} \wedge f(a_{m+1}, \ldots, a_n), \tag{1}$$

where

—*$p_x = x$ if $V_x = 1$ or $p_x = \bar{x}$ if $V_x = 0$;*
—*$p_{ai} = a_i$ if $V_i = 1$ or $p_{ai} = \bar{a}_i$ if $V_i = 0$;*
—*f is a Boolean function of variables $a_{m+1}, \ldots, a_n$.*

PROOF. Let $p_x = f_0(a_1, \ldots, a_n)$. Applying the Shannon expansion with respect to p_{a1}, while accounting for the first SLI, gives

$$p_x = p_{a1} \vee \bar{p}_{a1} \wedge f_1(a_2, \ldots, a_n), \tag{2}$$

where function f_1 has the same SLIs at its inputs as f_0, except the first SLI. Now suppose that for certain $k < m$,

$$p_x = p_{a1} \vee \ldots \vee p_{ak} \vee \bar{p}_{a1} \wedge \ldots \wedge \bar{p}_{ak} \wedge f_k(a_{k+1}, \ldots, a_n). \tag{3}$$

Then, substituting Shannon expansion with respect to $p_{a,k+1}$ for f_k, and accounting for

$$p_{a1} \vee \ldots \vee p_{ak} \vee \bar{p}_{a1} \wedge \ldots \wedge \bar{p}_{ak} \wedge p_{a,k+1} = p_{ak} \vee \ldots \vee p_{ak} \vee p_{a,k+1} \tag{4}$$

we obtain:

$$p_x = p_{a1} \vee \ldots \vee p_{a,k+1} \vee \bar{p}_{a1} \wedge \ldots \wedge \bar{p}_{a,k+1} \wedge f_{k+1}(a_{k+2}, \ldots, a_n). \tag{5}$$

Finally, setting $f_m = f$, we obtain (1).

Conversely, the set of SLIs of Theorem 1 can be easily derived from Equation (1). □

Since any Boolean function of one variable has two SLIs, we can conclude that f in (1) must have at least two variables, that is, $n - m > 1$. Therefore, we have the following.

COROLLARY 1. *For an n-input gate G implementing a Boolean function, specifying SLIs between its inputs and output is equivalent to specifying the Boolean function, if and only if G has exactly one SLI at every input.*

It is easy to see from (1) that gates for which all inputs have an SLI to the gate output are either an n-input AND or OR gate (with arbitrary inversions at inputs and output). We can therefore state the following corollary.

COROLLARY 2. *If a circuit consists of AND, OR, and INVERTER gates, then the logic function of the circuit is completely specified by the full set of SLIs in the circuit.*

If the circuit contains complex gates such as AOIs, OAIs, XORs, and XNORS, then the generated SLIs in the circuit will contain incomplete information about the circuit's logic function. For an AO22 gate, for example, the full set of SLIs consists only of trivial implications and, therefore, contains no information about the logic function of the gate. However, if we decompose the AO22 gate into two AND gates and one OR gate and construct SLIs with the use of the new internal nodes, then the SLIs of the AND/OR gates will completely define the logic function of the AOI gate. Therefore, for effective generation of SLIs, the circuit must be decomposed into the gates listed in Corollary 2.

2.2 Generation of SLIs for Complex Gates

To generate initial SLIs for a circuit containing complex multi-input gates, we first represent the circuit as a network of ROBDDs [Bryant 1986], where each ROBDD represents a single DC-connected component (DCCC) in the circuit. We then propose the following algorithm to generate an SLI for each DCCC directly from its ROBDD without explicitly decomposing it into AND, OR, and INVERTER gates. We define intermediate variables f_i for each vertex v_i in the ROBDD representing its Boolean function. Each intermediate variable will have four associated SLI lists as discussed in the previous section. For the root vertex, no intermediate variable is needed, since it corresponds to the output node of the gate.

We visit each nonterminal vertex in the ROBDD in topological order, starting from the bottom and working up toward the root vertex. At each vertex v_i with controlling variable c_i, we define the Boolean function of the intermediate variable f_i in terms of the intermediate variables of its child vertices and the controlling variable c_i and then create the SLIs associated with this function. As we visit vertices of the ROBDD, we may encounter one of the following possible situations.

(1) Both sons of vertex v are terminal vertices. In this case, we do not need to introduce an intermediate variable, since the Boolean function of v is entirely defined by its controlling variable c. If the high-son of v is 1, $v = c$ and if the high-son of v is 0, $v = c$.

(2) Vertex v, controlled by variable c, has one child vertex x that is a terminal vertex and one child vertex y that is a nonterminal vertex with intermediate variable w. Then the Boolean function of the intermediate variable f of vertex v will be defined by one of the following four cases.
   - —If x is 0 and is the low-son of v then $f = c \wedge w$.
   - —If x is 1 and is the low-son of v then $f = \bar{c} \vee w$.
   - —If x is 0 and is the high-son of v then $f = \bar{c} \wedge w$.
   - —If x is 1 and is the high-son of v then $f = c \vee w$.

(3) Both child vertices of vertex v, controlled by variable c, are nonterminal vertices. Suppose that the high-son vertex of v has intermediate variable a and the low-son vertex of v has intermediate variable b. In this case, we introduce two additional intermediate variables x and y, where $x = c \wedge a$ and $y = \bar{c} \wedge b$ Then the intermediate variable f of vertex v will be defined by the Boolean function $f = x \wedge y$.

As internal variables are defined during the traversal of the ROBDD, SLI lists are created for each variable. Since each intermediate variable is expressed as either a simple AND or OR function of its input variables, the complex gate will be completely defined by generated SLIs per Corollary 2.

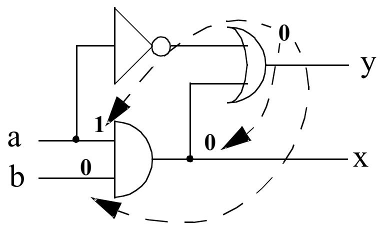


Fig. 3. Circuit with possible lateral SLI propagation.

2.3 Propagation of SLIs

After initial SLIs are generated for each gate in the circuit, we propagate SLIs through the circuit using the basic operations of list union, list intersection, and contrapositive law. For example, let a two-input AND gate be considered with inputs a, b and output x. If we have an implication list L_H (L_L) at nodes a and b then the implication list L_H^x (L_L^x) at output is calculated as the union of lists L_H^a and L_H^b (L_L^a and L_L^b). Similarly, list H_H^x (H_L^x) is calculated as the intersection of H_H^a and H_H^b (H_L^a and H_L^b). Accordingly, rules for implication propagation can be generated for OR gates and inverters. Once an SLI is obtained at a gate output, the reverse SLI is added by applying the contrapositive law:

$$\text{if } (a = v_a) \rightarrow (b = v_b) \quad \text{then} \quad (b = \bar{v}_b) \rightarrow (a = \bar{v}_a).$$

Therefore, we visit each gate in topological order and propagate the SLI lists at the input of the gates to the output. Since complex gates are implicitly decomposed into simple AND and OR gates, SLIs will propagate across complex gates without loss of information. In order to generate all possible implications in a circuit multiple forward propagation passes through the circuit with contrapositive law application may be required.

In addition to these so-called direct implication propagations, we propose so-called *lateral* SLI propagations. which allows us to find indirect implications, which are known to be particularly useful in logic optimization [Bahar et al. 1996; Long et al. 2000]. Again, let us consider the two-input AND gate with inputs a, b and output x. When we perform the list intersection between H_L^a and H_L^b, we exploit the gate implication $(a = 1 \wedge b = 1) \rightarrow (x = 1)$ to obtain the implication list at the output $H_L^x = H_L^a \cap H_L^b$. However, we can also use the equivalent gate implication $(a = 1 \wedge x = 0) \rightarrow (b = 0)$ which will result in the following implication list at node b, $L_L^b = H_L^a \cap L_L^x$ and $L_H^b = H_L^a \cap L_H^x$. We call this operation a lateral propagation of SLIs. Note that both the lateral and direct propagation of SLIs can be trivially extended to n-input AND and OR gates.

To illustrate the fact that lateral propagation cannot be obtained through direct propagation, we consider the simple example in Figure 3. In this example, we obtain two implication lists $H_L^a = \{y\}$ and $L_L^x = \{y\}$ through application of the contrapositive law and direct propagation across the inverter and OR gate. Due to lateral propagation, we therefore obtain the following implication at node b: $L_L^b = \{y\}$; that is, $(y = 0) \rightarrow (b = 0)$. It is clear that this SLI cannot be obtained by means of repeated direct propagation only.

```
1.Initialize trivial SLIs;
2. Repeat the following steps until convergence
   {
     2.1Repeat the following steps until convergence
          {
           For every gate in topological order
           perform forward SLI propagation
           with application of contrapositive law.
          }
     2.2 For every gate in reverse topological order
         perform lateral SLI propagation with
         application of transitive and contrapositive laws.
   }
```

Fig. 4. SLI propagation algorithm.

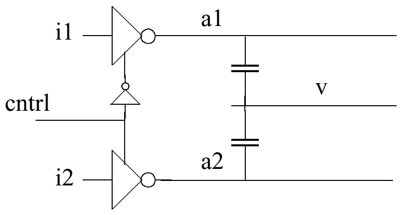


Fig. 5. Example circuit with tristate logic.

Therefore, the overall proposed SLI propagation algorithm consists of the following stages. First, we perform multiple direct propagations with application of the contrapositive law until convergence. Then, we perform multiple passes of lateral propagation with application of the contrapositive law until convergence. Each pass of lateral propagation is followed by one or more passes of direct propagation. The algorithm is shown in Figure 4. The transitive propagation can be applied either in forward or reverse topological order with reverse order yielding faster convergence in practice.

2.4 Generating SLIs for Circuits with Tristate Gates

Up to this point, we have assumed that all gates implement a Boolean function. In practice, however, it is very common to have tristate gates, where the output of the gate can take a high-impedance state. A simple approach is to ignore this high impedance state in false noise analysis. However, as shown in the example in Figure 5, some false noise failures may remain undetected under this simplifying assumption. In the example, two tristate drivers are shown, where one of the two drivers is always in a high-impedance state, as is common in bus structures. In this circuit, the two aggressor nets $a1$ and $a2$ have no logic correlation based on a strict Boolean analysis of the circuit. However, it is clear that only one of the two aggressors can switch simultaneously. In this section, we therefore show how to extend the proposed logic implications to handle tristate gates.

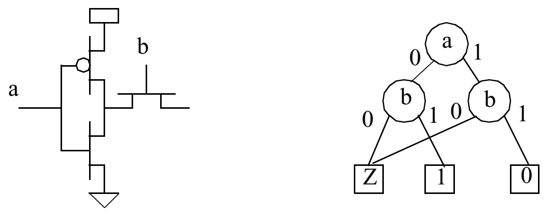


Fig. 6. Example of tristate gate and its MTBDD.

A tristate gate has, in addition to the logic 0 and logic 1 states, a high-impedance state (Z-state), when both the pull-up and pull-down networks are nonconducting. Since the inputs of a tristate gate can be in one of two states (logic 0 or logic 1), we can describe the tristate gate with a multiterminal BDD (MTBDD)[Meinel and Teobald 1998], using an additional terminal to represent the Z-state. A simple example of a tristate gate and its MTBDD is shown in Figure 6. Note that each tristate gate corresponds to a DCCC in the circuit.

The MTBDD is constructed directly from the DCCC netlist using the following steps.

(1) Order all inputs and initialize the state of each transistor as unknown.
(2) Create a root vertex.
(3) Recursively traverse all created vertices, and for every visited vertex enable transistors controlled by the corresponding variable with value 0 (for low-son) and 1 (for high-son). After this, check whether the son of the current vertex is one of the terminal vertices. If not, then create a new son vertex and make it current. If both vertices of the current vertex exist, then transistors controlled by corresponding variables are disabled, and its parent vertex is made current.
(4) After the traversal is completed and all nonterminal vertices have a low- and high-son, the MTBDD is reduced using repeated isomorphic subgraph elimination.

In Figure 7(a), an ideal unidirectional pass transistor (UPT) and its MTBDD representation is shown which we refer to as an *elementary tristate gate*. We also show an example of a more complex tristate gate and its MTBDD representation in Figure (b).

In order to efficiently generate SLIs for a tristate gate we propose the following transformation, referred to as the *pass-transformation*. In this transformation, a tristate gate is represented with two ordinary binary gates controlling an ideal undirectional pass-transistor, as shown in Figure 8. Signals a and b are calculated such that the output x is the same as the output of the tristate gate, meaning that the transformation does not alter the behavior of the tristate gate in any way. Signal a represents the logic value of the tristate gate when it is not in a high-impedance state and signal b represents those input states when the gate is in a high-impedance state. We define the pass-transformation as follows.

Definition 1. Given a tristate gate T, the *pass-transformation* of T consists of a pair of binary gates, P and Q with the same set of inputs, followed by a

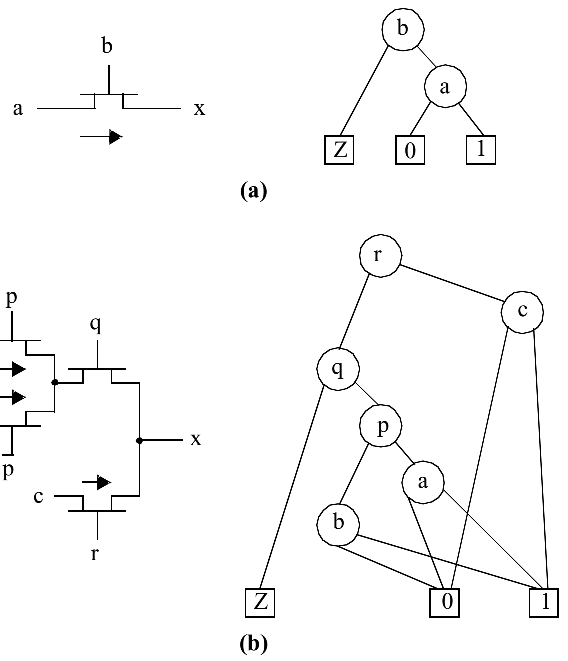


Fig. 7. Elementary and complex tristate gates and their MTBDD representations.

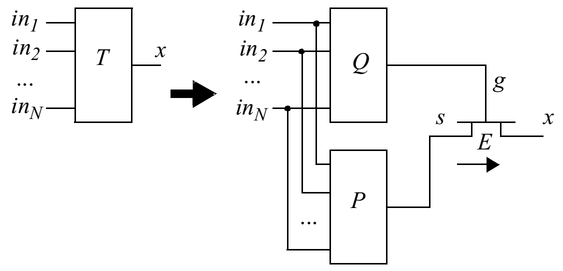


Fig. 8. Pass-transformation of tristate gate.

unidirectional pass-transistor E, such that:

—the output of P is connected to the input s of E;

—the output of Q is connected to the input g of E;

—if the output of T is Z then the output of Q is 0 and the output of P is either 0 or 1;

—if the output of T is 0 then the output of Q is 1 and the output of P is 0;

—if the output of T is 1 then the output of Q is 1 and the output of P is 1.

Since our purpose is the calculation of logic implications, it is not necessary to explicitly construct gates P, Q and only their ROBDDrepresentations are

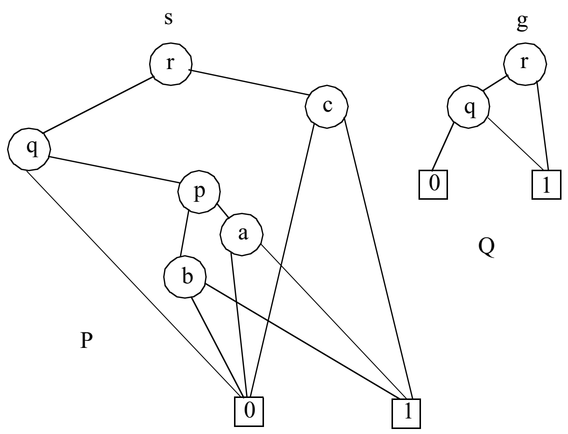


Fig. 9. Result of pass-transformation applied to tristate gate from Figure 7(b).

Table I. Logic Function of Ideal Unidirectional Pass Transistor

G	S	X
0	0	Z
0	1	Z
1	0	0
1	1	1

formed. Given the MTBDD representation T of a tristate gate, the ROBDD for gates P and Q can be easily constructed as follows.

—To construct the ROBDD for Q we transform the MTBDD T into an ROBDD by changing each terminal vertex Z in T to 0 and by merging terminal vertices 0 and 1 in T to 1. This is followed by OBDD reduction.

—To construct the ROBDD for P we transform the MTBDD T into an ROBDD by merging terminal vertices Z in T into 0. This is followed by OBDD reduction.

The construction of the ROBDDs P and Q is illustrated in Figure 9 for the MTBDD shown in Figure 7(b).

After we have applied the pass-transform on all MTBDDs, we generate SLIs for the tristate gates. We first consider binary SLIs (B-SLIs) and then discuss tristate SLIs (T-SLIs). The P and Q functions obtained after the pass-transformation are binary ROBDDs and, we can use the same forward and lateral SLI propagation as described in Section 2.3 for their constituent AND, OR, and INV gates. For the ideal unidirectional pass transistor (UPT), we derive its propagation rules from its logic function, which is shown in truth table form in Table I. Since the first two entries result in a Z-state, they are not useful for binary SLIs. Therefore, we obtain the following two Boolean implications for the UPT gate: $s \wedge g \rightarrow x$ and $\bar{s} \wedge g \rightarrow \bar{x}$. We now propagate SLIs using list intersection and list union operations based on the Boolean implications of the gate, as follows.

$$H_H^x = H_H^s \cap H_H^g$$
$$L_H^x = L_H^s \cap H_H^g,$$

with similar rules for the H_L^x and L_L^x lists. Also, we perform lateral propagation of B-SLIs through the UPT gate based on the derived gate implications $x \wedge g \rightarrow s, \bar{x} \wedge g \rightarrow \bar{s}$, as follows.

$$H_H^s = H_H^x \cap H_H^g$$
$$L_H^s = L_H^x \cap H_H^g,$$

with similar rules for the H_L^s and L_L^s lists. Note that contrapositive and transitive laws are applicable to all B-SLIs, including B-SLIs obtained by propagation through the UPT since they are fundamental logic laws. It is thus clear that using the pass-transform, generation and propagation of binary SLIs for tristate gates are quite similar to those for Boolean gates.

We now discuss the generation of tristate SLI. As explained in further detail in Section 3, false noise analysis using SLIs is based on detecting that simultaneous switching of two nodes is prohibited by the presence of an SLI between these two nodes. However, a tristate implication, such as $(a = 1) \rightarrow (b = Z)$ will not provide useful information for false noise avoidance, since the Z-state is a unknown binary value and could be interpreted as either logic 0 or logic 1. For false noise analysis, we therefore introduce a new signal state Y which represents an unknown *stable* Boolean state, which is stored on the capacitance of the wire. In other words, a signal is in a Y-state if it cannot switch (in a glitch-free circuit). We can therefore propagate T-SLIs of the type $(a = V) \rightarrow (b = Y)$, and $(a = Y) \rightarrow (b = Y)$, where V is from {0, 1}, and a, b are circuit nodes. We define the propagation and generation of T-SLIs for unidirectional pass-transistors, INV, AND, and OR gates as follows, where p is an arbitrary circuit node and V is from {0, 1}.

UPT (unidirectional pass transistor), s,g,x—source, gate, drain, respectively:

—if $(p = V) \rightarrow (g = 0)$ then $(p = V) \rightarrow (x = Y)$;
—if $(p = V) \rightarrow (s = Y)$ and $(p = V) \rightarrow (g = 1)$ then $(p = V) \rightarrow (x = Y)$.

INV (inverter), a—input, x—output:

—if $(p = V) \rightarrow (a = Y)$ then $(p = V) \rightarrow (x = Y)$;
—if $(p = Y) \rightarrow (a = Y)$ then $(p = Y) \rightarrow (x = Y)$.

Simple two-input gate (AND, OR, ...), a,b—inputs, x—output:

—if $(p = V) \rightarrow (a = Y)$ and $(p = V) \rightarrow (b = \text{non\_controlling\_value})$ then $(p = V) \rightarrow (x = Y)$;
—if $(p = V) \rightarrow (a = Y)$ and $(p = V) \rightarrow (b = Y)$ then $(p = Y) \rightarrow (x = Y)$;
—if $(p = Y) \rightarrow (a = Y)$ and $(p = Y) \rightarrow (b = Y)$ then $(p = Y) \rightarrow (x = Y)$.

A tristate gate in a high-impedance state will have an output node that is in the Y-state. However, it should be emphasized that the meanings of the Z-state and the Y-state are different, and that the Y-state can propagate, whereas the Z-state cannot. For instance, if the input of an inverter is in the Y-state, its output is also in the Y-state, and it cannot be in the Z-state. Based on the above gate implications, we can again define the propagation of T-SLI through tristate and binary gates using the union and intersection operations of SLI lists. We can also again apply the contrapositive law. For instance, applying the contrapositive law to implication $(p = 0) \rightarrow (x = Y)$ will result in the implication $(x = \bar{Y}) \rightarrow (p = 1)$.

In the example circuit shown in Figure 5, the following two T-SLIs are generated relating $a2$ to $a1$ using the proposed approach: $(a2 = \bar{Y}) \rightarrow (a1 = Y)$ and $(a1 = \bar{Y})(a2 = Y)$. These implications therefore indicate that if either aggressor $a1$ or $a2$ is switching (i.e., is not in a stable state), the other aggressor is in a stable unknown Boolean state and cannot inject noise simultaneously with the switching aggressor.

3. FALSE NOISE ANALYSIS USING SLIs

After SLIs are generated in the circuit, we apply them in our false noise analysis. For each victim net, a set of aggressor nets that inject coupled noise on the victim net is identified, where each aggressor can potentially contribute a different amount of injected noise. A victim net and its associated aggressor nets are referred to as a *noise cluster*. Among the set of aggressors in a noise cluster, we intend to find the subset of aggressors with a maximal sum of injected noise, such that the logic constraints represented by SLIs between a pair of aggressors and between an aggressor and the victim net are satisfied. We refer to this problem as the *maximum realizable noise* problem and the set of aggressors responsible for the maximal realizable noise as the *maximal realizable aggressor set*. Note that each noise cluster can be analyzed individually since the global logic relationships present in the circuit are already represented by pairwise SLIs between the nets in the noise cluster. The maximum realizable noise problem is therefore defined with the information:

(1) a single victim node V,
(2) a set of aggressor nodes A_i that inject noise w_i, $i = 1, \ldots, n$ on the victim net V, and
(3) a noise type t, $t \in \{$*LowR, LowF, HighR, HighF, RiseR, RiseF, FallR, FallF*$\}$.

The first four noise types correspond to functional noise where the victim net is either at a stable low state (*LowR* and*LowF*) or a stable high state (*HighR* and*HighF*), while the aggressor nets are either rising (*LowR* and*HighR*) or falling (*LowF* and*HighF*). The second set of four noise types corresponds to delay noise where the victim net is either rising (*RiseR* and *RiseF*) or falling (*FallR* and *FallF*) while the aggressor nets are again rising (*RiseR* and *FallR*) or falling (*RiseF* and *FallF*).

The false noise analysis algorithm now consists of several basic steps. First we compute the SLIs in the circuit, as was explained in Section 2. Second, we represent the logic constraints between the aggressors for a particular noise type using a constraint graph, as presented in Section 3.1. Finally, we find the maximum realizable noise by solving the Maximum Weight Independent Set problem for the constraint graph as presented in Section 3.2.

3.1 Forming the Constraint Graph

A constraint graph is an undirected graph $G=(V,E,w)$ of vertex set $V=\{v_1,\ldots,v_n\}$, edge set $E=\{(u,v):u,v\in V,u\neq v\}$, and a vertex weighting function w such that $w(u)\geq 0,\forall(u\in V)$. The vertices represent the aggressor nets of a noise cluster and the weight of a vertex is the amount of noise injected by the associated aggressor net on the victim net. We form a separate constraint graph for each noise type. An edge exists between two vertices in the constraint graph if the two associated aggressors cannot simultaneously switch and inject noise on the victim net.

For each particular noise type, we first determine the initial and final states of the victim nets V_v^i and V_v^f and the initial and final states of the aggressor net V_a^i and V_a^f. For instance, for noise type *LowR*, $V_v^i=0$, $V_v^f=0$, $V_a^i=0$, $V_a^f=1$ and for noise type *RiseF*, $V_v^i=0$, $V_v^f=1$, $V_a^i=1$, $V_a^f=0$. We then determine which aggressor nets can have a transition that is logically compatible with the initial and final victim states for this particular noise type. If, for a victim/aggressor pair (v,a_i), either of the following two B-SLIs exist, $(v=V_v^i)\rightarrow(a_i=\bar{V}_a^i)$ or $(v=V_v^f)\rightarrow(a_i=\bar{V}_a^f)$, then the aggressor net is not compatible with the victim net for this noise type and is not included in the constraint graph. For instance, if the noise type is *RiseF*, the victim is switching from low to high, and the aggressor must switch from high to low. Therefore, the presence of implication $(v=0)\rightarrow(a_i=0)$ would prohibit aggressor a_i from switching and injecting noise on net v, since it would already be in its final state at the start of the victim transition. Similarly, the implication $(v=1)\rightarrow(a_i=1)$ would prohibit the aggressor a_i from switching since it would be in its initial state at the end of the victim transition. In this case, a_i would not be included in the constraint graph for victim net v under noise type *RiseF*.

For T-SLIs we consider functional and delay noise separately. For functional noise (*LowR, LowF, HighR, HighF)*, the victim is in a stable state and $V_v^i=V_v^f$. Therefore the presence of the implication $(v=V_v^i)\rightarrow(a_i=Y)$ between victim v and aggressor a_i will eliminate the aggressor from participating in the constraint graph, since the aggressor cannot switch while the victim is in its stable state. For delay noise (*RiseR*, *RiseF*, *FallR*, *FallF*), the victim transitions. The presence of the implication: $(v=Y)\rightarrow(a_i=\bar{Y})$ therefore means that the aggressor net will be in a stable Boolean state and cannot switch when the victim transitions, and aggressor a_i cannot contribute noise to the victim in this case and is not included in the constraint graph.

After the vertices of the constraint graph have been identified, we determine which edges exist in the graph. We examine each pair of vertices

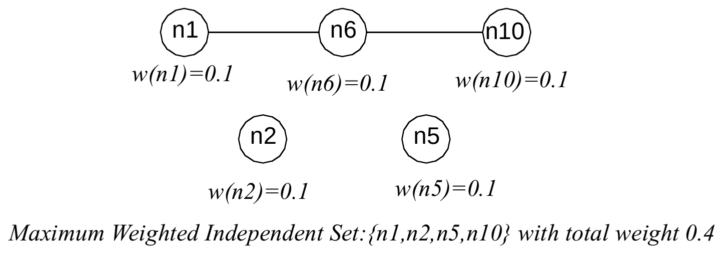

n1
n6
n10
w(n1)=0.1
w(n6)=0.1
w(n10)=0.1
n2
n5
w(n2)=0.1
w(n5)=0.1
Maximum Weighted Independent Set:{n1,n2,n5,n10} with total weight 0.4

Fig. 10. Example of the constraint graph.

(v_i, v_j), $i \neq j$, v_i, $v_j \in V$. Again, we can determine if v_i and v_j can both switch in the required direction by searching for an SLI that renders their transitions logically incompatible. In this case, if we find either of the two B-SLIs $(a_i = V_a^i) \rightarrow (a_j = \bar{V}_a^i)$, $(a_i = V_a^f) \rightarrow (a_j = \bar{V}_a^f)$, or the T-SLI $(a_i = \bar{Y}) \rightarrow (a_j = Y)$, a_i and a_j cannot both participate in the maximal realizable aggressor set and an edge (v_i, v_j) is created in the constraint graph.

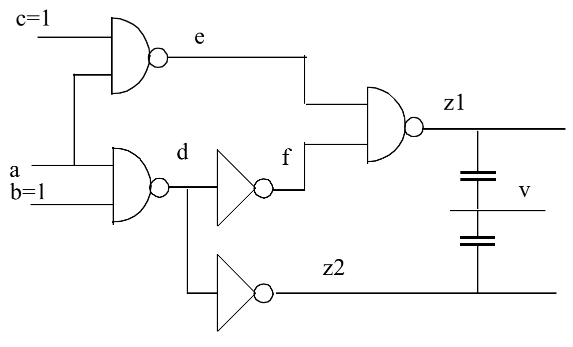
3.2 Solving MWIS Problem

Given the constraint graph constructed according to Section 3.1, we can find the maximal realizable aggressor set and the associated maximal realizable noise by solving the *maximum weighted independent set* (MWIS) problem for the constraint graph. Consider the constraint graph $G = (V, E, w)$ and the global weighting function $W(K) = \sum_{\in} w(u)$, for $K \subseteq V$. An independent set S is a subset, $S \subseteq V$, such that for any $u, v \in S; (u, v) \notin E$. The maximum independent set is the independent set S, such that $W(S)$ is maximum.

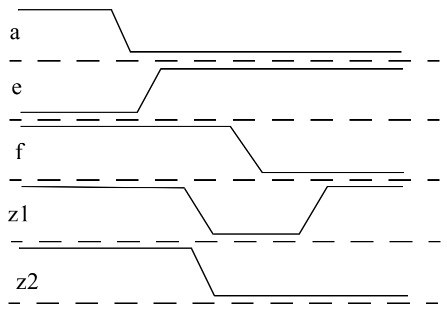
For a general constraint graph, the MWIS problem is known to be NP-complete [Garey and Johnson 1979]. However, in our problem formulation, we have the advantage that the number of significant aggressors in a noise cluster is typically small (<15). Therefore the MWIS problem for the associate constraint graph can in most cases be solved exactly by exhaustive enumeration of all independent sets. For larger graphs, we use the heuristic algorithm of Loukakis and Tsouros [1983]. As a simple example of our approach, again let the circuit in Figure 1 be considered. Let $n7$ be the victim node with *LowR* noise, and let all 10 other nodes be potential aggressors, each aggressor net contributing the same injected noise. As shown in Figure 10, the resulting constraint graph consists of the $\{n1, n2, n5, n6, n10\}$ aggressors' vertices and two edges $(n1, n6)$ and $(n6, n10)$. The final set of maximal realizable aggressor nets is $\{n1, n2, n5, n10\}$.

4. EXTENSION TO TIMED SLIs

Until now, we have considered false noise analysis based on zero-delay implications. These implications are only valid when the circuit has reached a stable state, that is, at the beginning and end of a clock cycle. However, when the circuit is in transition, it is possible that two aggressor nets can switch simultaneously, even though their zero-delay SLIs would indicate that such a transition is impossible. This occurs when there are glitches in the circuit,



(a)



(b)

Fig. 11. Noise injection by glitches.

as shown in the simple example in Figure 11(a). In this example, nodes $z1$ and $z2$ are aggressor nodes of victim v. Using the contrapositive law and direct SLI propagation, the following implication will be computed under the zero-delay model, $(z2 = 0) \rightarrow (z1 = 1)$, which will disallow both aggressors from switching in the same direction, which is correct when we consider the final transition of these nets. However, if we switch signal a low, while setting inputs b and c high, signal $z1$ glitches, as shown in Figure 11(b), and can inject noise simultaneously with aggressor $z2$. Therefore, zero-delay implications will yield a conservative false noise analysis only if the circuit in question is glitch-free. In this section, we therefore show how our zero-delay implication can be extended with delay information to obtain so-called *timed implications* that can be used for false noise analysis in circuits that have glitching signals. For simplicity, we restrict our discussion to binary SLIs, although the proposed methods are also applicable to the discussed tristate SLIs.

4.1 Basic Definitions

In Bobba and Hajj [1998], timed SLIs are proposed using the formulation:

$$(a(t) = V_a) \rightarrow (b(t + T) = V_b). \tag{6}$$

Here a transition of net a to value V_a implies that net b will be at value V_b after some fixed time interval T. This model is applicable if all gates have a constant delay and we refer to these SLIs as *fixed delay* SLIs. In practice, however, the delay of a gate varies due to process variation and state dependence. Therefore, fixed delay SLIs cannot be used in such cases and we propose the following two types of timed logic implications.

Definition 2. An *exclusive timed SLI* (*or E-SLI*) is the relation between signals a, b:

$$(\forall(t \in [t_1, t_2])(a(t) = V_a) \rightarrow (\forall(t \in [t_1 + T_1, t_2 + T_2])(b(t) = V_b). \qquad (7)$$

An E-SLI reflects the situation where the presence of a stable value of signal a during the entire time interval $[t_1, t_2]$ guarantees the stable value of signal b during the entire time interval $[t_1 + T_1, t_2 + T_2]$. The SLI is said to be exclusive, since other values for signal a and b are not permitted during the respective time intervals. We use the following short notation to denote an E-SLI:

$$(a = 1) \rightarrow (b = 0)(T_1, T_2). \qquad (8)$$

Definition 3. An *inclusive timed SLI* (*or I-SLI*) is the relation between signals a, b:

$$(\exists(t \in [t_1, t_2])(a(t) = V_a) \rightarrow (\exists(t \in [t_1 + T_1, t_2 + T_2])(b(t) = V_b). \qquad (9)$$

An I-SLI implies that if signal a is at value V_a at least once in the time interval $[t_1, t_2]$, signal b will be at value V_b at least once in the time interval $[t_1 + T_1, t_2 + T_2]$. Since the I-SLI allows for other signal values to exist during the respective time intervals, it is said to be inclusive. We use the following short notation to denote an I-SLI.

$$(a = 1) \Rightarrow\!\!\gg (b = 0)(T_1, T_2). \qquad (10)$$

We can see Equations (7) and (9) are only meaningful if $t_2 - t_1 \geq \max(0, T_1 - T_2)$. Also, note that zero-delay SLIs and fixed delay SLIs (6) are special cases of E-SLIs and I-SLIs.

We now introduce the following two useful definitions.

Definition 4. An E-SLI or I-SLI is said to be *expanding* if $T_2 > T_1$ and is said to be *contracting* if $T_1 > T_2$.

An E-SLI or I-SLI is *neutral* if it is both nonexpanding and noncontracting (i.e., $T_1 = T_2$). Clearly, zero-delay SLIs and the fixed delay SLIs (6) are neutral.

We now examine a logic gate with input a and output b and zero-delay SLI $(a = 1) \rightarrow (b = 1)$. Also, assume that the rise and fall, minimum and maximum delays of the gate are $T^R_{\min}, T^R_{\max}, T^F_{\min}, T^F_{\max}$. A rising transition of signal a may be accompanied by a rising transition of b with a time shift lying between $T^R_{\min}, T^R_{\max}$ as shown in Figure 12. Similarly, a falling transition of a may be accompanied by a falling transition of b with a time shift lying between $T^F_{\min}, T^F_{\max}$. We can see from Figure 12 that when signal a is at a stable high value during the entire interval $[t_1, t_2]$, b will be guaranteed to be at a stable high value for the entire time interval $[t_1 + T^R_{\max}, t_2 + T^F_{\min}]$, excluding the shaded areas in Figure 12. Therefore, we can formulate the following exclusive-SLI.

$$(a = 1) \rightarrow (b = 1)(T^R_{\max}, T^F_{\min}).$$

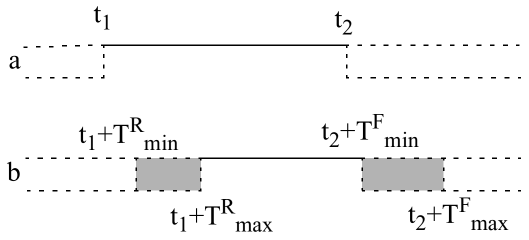


Fig. 12. Timed SLI $a \rightarrow b(T^R_{\max} T^F_{\min})$.

Similarly, we can see that the presence of a high value for signal a during at least one point in time interval $[t1, t2]$ implies that for signal b a high value will exist for at least one time point in the interval $[t_1 + T^R_{\min}, t_2 + T^F \max]$, including the shaded areas in Figure 12. We can therefore formulate the following inclusive-SLI.

$$(a = 1) \Rightarrow\!\!\!\gg (b = 1)(T^R_{\min}, T^F_{\max}).$$

For typical gates, the time intervals of I-SLIs will expand as we propagate them through the circuit, and the time intervals of E-SLIs will shrink.

Below, we now show the contrapositive and transitive laws for timed SLIs.

$$\text{If } (a = v_a) \rightarrow (b = v_b)(T_1, T_2) \quad \text{then} \quad (b = \bar{v}_b) \Rightarrow\!\!\!\gg (a = \bar{v}_a)(-T_1, -T_2). \tag{11}$$

$$\text{If } (a = v_a) \Rightarrow\!\!\!\gg (b = v_b)(T_1, T_2) \quad \text{then} \quad (b = \bar{v}_b) \rightarrow (a = \bar{v}_a)(-T_1, -T_2). \tag{12}$$

One can see that the result of applying the contrapositive law to an E-SLI results in an I-SLI, and vice versa. This motivates the need for having two types of timed SLIs. Although only E-SLIs are directly used for false noise elimination, the creation and propagation of I-SLIs allows us to generate more E-SLIs through the application of the contrapositive law. For example, in the situation shown in Figure 12 we can obtain a second E-SLI $(b = 0) \rightarrow (a = 0)(-T^R_{\min}, -T^F_{\max})$, through the application of the contrapositive law to the I-SLI $(a = 1) \Rightarrow\!\!\!\gg (b = 1)(T^R_{\min}, T^T_{\max})$. In general, an E-SLI is always accompanied by an associated I-SLI, which results in a reverse E-SLI by applying the contrapositive law. In addition to the contrapositive law, we can also extend the transitive law to time SLIs as shown below.

$$\begin{aligned} &\text{If } (a = v_a) \rightarrow (b = v_b)(T_1, T_2) \quad \text{and} \quad (b = v_b) \rightarrow (c = v_c)(T_3, T_4) \\ &\quad \text{then} \quad (a = v_a) \rightarrow (c = v_c)(T_1 + T_3, T_2 + T_4). \end{aligned} \tag{13}$$

$$\begin{aligned} &\text{If } (a = v_a) \Rightarrow\!\!\!\gg (b = v_b)(T_1, T_2) \quad \text{and} \quad (b = v_b) \Rightarrow\!\!\!\gg (c = v_c)(T_3, T_4) \\ &\quad \text{then} \quad (a = v_a) \Rightarrow\!\!\!\gg (c = v_c)(T_1 + T_3, T_2 + T_4). \end{aligned} \tag{14}$$

4.2 Propagation of Timed SLIs

Similarly to zero-delay SLIs, we store timed SLIs for a circuit node x using four implication lists: $L^x_L, H^x_H, L^x_H, H^x_L$. Every implication list is a list of E-SLIs, where each list entry contains both the implicating node and its associated time shifts T_1 and T_2. For instance, the implication list

$$L^x_H = \{a(1, 2), a(-4, -2), b(0, 1)\} \tag{15}$$

describes the following set of E-SLIs at x: $(a=1)\rightarrow(x=0)(1,2), (a=1)\rightarrow(x=0)(-4,-2), (b=1)\rightarrow(x=0)(0,1)$.

If an implication list contains two entries with the same node but with different time shifts: $a(T_1, T_2)$, $a(T_3, T_4)$, we can merge these two entries into a single entry in the following two cases.

—Both SLIs are noncontracting and the intervals $[T_1, T_2]$ and $[T_3, T_4]$ intersect. In this case, the two entries can be merged into a single entry $a(T_5, T_6)$, where $[T_5, T_6]$ is the union of intervals $[T_1, T_2]$ and $[T_3, T_4]$.

—Both SLIs are contracting and interval $[T_1, T_2]$ is contained in interval $[T_3, T_4]$; then the first SLI can be deleted, since it follows from the second SLI. (A similar operation can be performed when $[T_1, T_2]$ contains $[T_3, T_4]$.)

During propagation of timed SLIs, at each node all possible SLIs in the SLI lists are merged.

The propagation of SLIs across a logic gate from the gate inputs to the gate output is performed using two basic operations: *timed union* of two implication lists and *timed intersection* of two implication lists, as explained in more detail below.

Timed Union. The timed union of two implication lists L_1, L_2 is denoted as $L = L_1\mathrm{U}_tL_2$, and is obtained as follows. First, we compose a new list L containing all entries of L_1 and L_2 shifted with use of proper gate delays. Then, we perform all possible merges of entries (as described above).

Timed Intersection. To formulate the timed intersection operation, we consider an AND gate with inputs a, b and output x, and the two timed implications: $(p=1)\rightarrow(a=1)(T_1^a, T_2^a)$ and $(p=1)\rightarrow(b=1)\,(T_1^b, T_2^b)$. For time interval $[t_1, t_2]$, if $p([t_1,t_2])=1$ then $a([R_1^a, R_2^a])=1$ and $b([R_1^b, R_2^b])=1$, where $R_1^a=t_1+T_1^a$, $R_2^a=t_2+T_2^a$, and $R_1^b=t_1+T_1^b$, $R_2^b=t_2+T_2^b$. We can take $[t_1,t_2]$ such that intervals $[R_1^a, R_2^a]$ and $[R_1^b, R_2^b]$ intersect. In this case $x=1$ within time interval $[Q_1, Q_2]$, where $Q_1=\max((R_1^a+D_{a,\max}^r),(R_1^b+D_{b,\max}^r))$, $Q_2=\min((R_2^a+D_{a,\min}^f), (R_2^b+D_{b,\min}^f))$,where $D_{a,\max}^r$, $D_{a,\min}^f$, $D_{b,\max}^r$, $D_{b,\min}^f$, are the minimum and maximum, rising and falling gate delays from input a or b to output x. So, we obtain the following E-SLI at x.

$$\begin{aligned}(p=1)\rightarrow(x=1)\big(&\max\big(\big(T_1^a+D_{a,\max}^r\big), \big(T_1^b+D_{b,\max}^r\big)\big),\\ &\min\big(\big(T_2^a+D_{a,\min}^f\big), \big(T_2^b+D_{b,\min}^f\big)\big)\big).\end{aligned} \tag{16}$$

Based on this consideration we can formulate the timed intersection rule: The timed intersection of two implication lists L_1, L_2 is denoted as $L = L_1\mathrm{I}_tL_2$, and is obtained as follows. First, we initialize L with an empty list. Then, for every pair of entries $a(T_1^a, T_2^a)$ from L_1 and $b(T_1^b, T_2^b)$ from L_2, such that $a=b$ (the same node), we add a new entry to L using a formula like (16) with the proper gate delays. Finally, we perform all possible merges of entries in L.

Clearly, this rule is valid also for lateral propagation of E-SLIs. Therefore, the algorithm for E-SLI generation and propagation is similar to that for zero-delay SLIs except for the additional timing information. Also, timed-SLIs are propagated through complex gates using their ROBDD representations

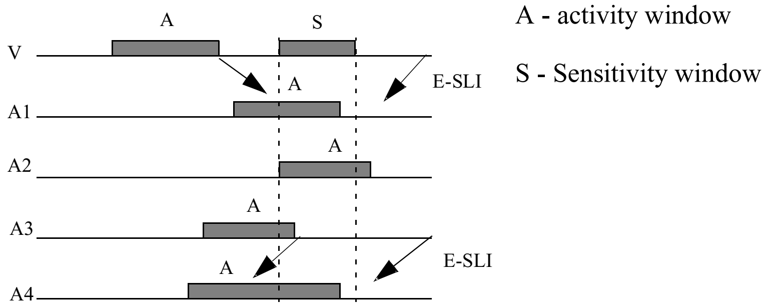


Fig. 13. Constraint graph generation using timing windows and E-SLIs.

directly, without actual decomposition of the complex gates into simple gates.

4.3 Use of Timed SLIs for False Noise Analysis

Timed SLIs can be used in the elimination of false noise from a circuit. Unlike zero-delay SLIs, a timed SLI will result in a conservative analysis that is valid even for circuits with glitches. False noise analysis with timed SLIs is similar to that with zero-delay SLIs in that we formulate the problem using a constraint graph and solve it using the MWIS problem. First, a timing analysis of the circuit is performed to obtain the timing windows of the signals. For the aggressor nets, so-called *activity* windows are constructed, indicating the possible time interval when an aggressor net can switch. For the victim net, a so-called *sensitivity* window [Levy et al. 2000] is also constructed, indicating the time interval when a coupled noise event could have an impact on the logic operation of the circuit. The use of E-SLIs combined with the timing windows is illustrated in Figure 13. A cluster of four aggressors ($A1$ to $A4$) and one victim (V) is shown. For the victim both the activity and sensitivity windows are shown, and for the aggressors only the activity windows are shown. We consider the presence of two E-SLIs, $(V=1) \rightarrow (A1=1)(T1, T2)$ and $(A3=0) \rightarrow (A4=1)(T3, T4)$. The arrows in Figure 13 show timing shifts for each E-SLI. We consider a *HighF* noise which requires that during the sensitivity interval of V, the victim net is logic 1 while the aggressors switch from $1 \rightarrow 0$. However, the E-SLI between V and $A1$ prevents $A1$ from switching from $1 \rightarrow 0$ during the sensitivity interval since the victim is in a static logic 1 state after its activity interval and hence, $A1$ is not included in the constraint graph. The constraint graph therefore consists of three vertices, $A2$, $A3$, and $A4$. Also, due to the E-SLI between $A3$ and $A4$, only one of these two aggressors can switch from $1 \rightarrow 0$, resulting in an edge between $A3$ and $A4$. The false noise analysis is now performed by solving the MWIS problem, as discussed in Section 3.2.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

The presented algorithms were implemented in an industrial noise analysis tool called ClariNet [Levy et al. 2000]. The system was designed using a separate

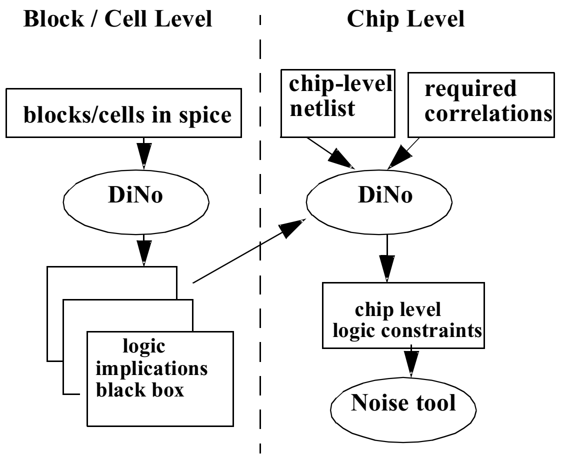


Fig. 14. Block diagram of SLI-based noise analysis algorithms.

Table II. Results of Testing SLI Propagation Algorithm

Circuit	Number of Nets	#SLIs Without Lateral	#SLIs with Lateral	% Lateral	Number SLIs/ Pair(%)
cnt_0	83	1196	1466	23	21
cnt_1	87	1222	1516	24	20
cnt_ones	97	1976	2248	14	24
cnt_zeros	99	1812	2098	16	21
c432	248	7826	20210	158	33
cla1	333	5136	5672	10	5
testckt	474	82572	86444	5	38
c1355	59	27218	32802	21	10

logic analysis engine called DiNo, which generates the SLIs for the circuit. First, the noise analysis tool performs the analysis without logic information. If a victim fails, the noise tool will request the SLIs for the nets belonging to the noise cluster of the failing victim net and form the constraint graph to determine the maximum feasible noise.

The analysis can be performed both at the block- and chip-level. At the block-level, the tool directly operates on the transistor-level description of the circuit. At the chip-level, DiNo first precharacterizes each gate in the library with a so-called *logic implication black box*. These black boxes are then used in the chip-level generation of SLIs to allow for increased efficiency. Figure 14 illustrates the chip level analysis methodology.

In Table II, we show the number of generated SLIs for a number of circuits using the proposed SLI generation and propagation approach. The first two circuits are ISCAS benchmark circuits [Brglez and Fujiwara 1985], and the remaining circuits are industrial circuits synthesized using a commercial synthesis tool. The third and fourth columns show the number of generated SLIs using only direct propagation and using both direct and lateral propagation, respectively. In the fifth column the percentage of increase in the number of

Table III. Results of SLI-Based Noise Analysis on Block Level

Circuit	Number of Nets	Number of SLIs Computed	Number of Failures	Number Failures with DiNo	% Reduction
plldriver	35	188	59	31	47
cntrl	47	112	73	63	14
srot8	308	24380	401	358	11
xbar	433	2336	384	314	18
srot16	622	55976	975	841	14
adder32	1168	195902	112	81	28
srot32	1588	451422	2005	1417	29
proc	46168	10910	11603	10452	10

generated SLIs due to the use of lateral propagation is recorded. The final column shows the number of SLIs as a percentage of the total number of node pairs. The results in Table II demonstrate the effectiveness of the lateral SLI propagation proposed in this article, which increased the number of generated of SLIs on average by 38%. The total number of SLIs, as a percentage of the number of node pairs ranges from 5 to 38%, revealing significant dependence on the structure of the circuit. On average, the algorithm generated SLIs for 21% of all node pairs.

The false noise analysis was used on a number of industrial circuits, as shown in Table III. Circuits *plldriver* and *cntrl* are small control blocks. Circuit *xbar* is a small crossbar switch, circuit *rot*8*,* *rot*16, and *rot*32 are 8, 16, and 32-bit shifters, circuit *adder*32 is a 32-bit adder, circuit *proc* is a microprocessor core. The second column shows the number of top level nets analyzed for noise. The fourth column shows the number of noise failures without false noise analysis, and the fifth column shows the number of failures with false noise analysis as presented in this article. Note that the number of failures can exceed the number of nodes, since there are several noise types for each net. The final column shows the percentage of decrease in the number of noise failures due to the use of false noise analysis. On average, a decrease of 27% is obtained over all test cases, which significantly reduced the task of fixing noise failures for the designers. Besides reducing the number of noise failures, SLIs also reduce the noise value of the nets that remain failures. For example, for the circuit *proc* the proposed approach significantly reduces the noise level for 3500 nets.

6. CONCLUSIONS

In this article, we presented a new approach for false noise analysis. We proposed the use of logic implications for eliminating aggressor nets that cannot simultaneously switch. We first showed how simple logic implications can be effectively generated and propagated in the circuit. We proved that SLIs only fully represent the logic function of a circuit if it consists of simple NAND, NOR, and INV gates. We then showed how SLIs can be generated for complex gates using implicit decomposition of their ROBDD representation. We also introduced a new, so-called, lateral propagation method to increase the number of obtained SLIs in a circuit and methods for handling tristate circuits. Using the

SLIs we showed how the false noise problem can be formulated as a constraint graph and solved as a maximum weighted independent set problem. Finally, we showed how the proposed SLIs can be extended to timed implications for conservative false noise analysis in nonglitch-free circuits. The presented algorithms were implemented and used on industrial circuits. The results showed a reduction of 27% in the number of failures on average, underscoring the importance of false noise analysis.

REFERENCES

Bahar, R. I., Burns, M., Hachtel, G. D., et al. 1996. Symbolic computation of logic implications for technology-dependent low-power synthesis. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

Bobba, S. and Hajj, I. N. 1998. Estimation of maximum current envelope for power bus analysis and design. In *Proceedings of the International Symposium on Physical Design*.

Brglez, F. and Fujiwara, H. 1985. A neutral netlist of 10 combinatorial benchmark circuits. In *Proceedings of the IEEE ISCAS*, IEEE Press, Piscataway, N.J., 695–698

Brown, F. M. 1990. *Boolean Reasoning*. Kluwer Academic, Hingham, Mass.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35*, 677–691.

Chen, P. and Keutzer, K. 1999. Towards tyrue crosstalk noise analysis. In *ACM/IEEE Proceedings of the International Conference on Computer-Aided Design*, 132–137.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

Hachtel, G., Jacoby, R., Moceyunas, P., and Morrison, C. 1998. Performance enhancements in BOLD using implications. In *ACM/IEEE Proceedings of the International Conference on Computer-Aided Design*, 94–97.

Kirkpatrick, D. A. and Sangiovanni-Vincentelli, A. L. 1996. Digital sensitivity: Predicting signal interaction using functional analysis. In *ACM/IEEE Proceedings of the International Conference on Computer-Aided Design*, 536–541.

Kunz, W. and Menon, P. R. 1994. Multi-level logic optimization by implication analysis. In *ACM/IEEE Proceedings of the International Conference on Computer-Aided Design*, 6–13.

Levy, R., Blaauw, D., Braca, G., Dasgupta, A., Grinshpon, A., Oh, C., Orshav, B., Sirichotiyakul, S., and Zolotov, V. 2000. Clarinet: A noise analysis tool for deep submicron design. In *Proceedings of the IEEE/ACM Design Automation Conference* (June), 233–238.

Long, W., Wu, Y. L., and Bian, J. 2000. IBAW: An implication-tree based alternative-wiring logic transformation algorithm. In *Proceedings of the Asian Pacific Design Automation Conference*, 415–422.

Loukakis, E. and Tsouros, C. 1983. An algorithm for the maximum internally stable set in a weighted graph. *Int. J. Comput. Math. 13*, 117–129.

Meinel, C. and Teobald, T. 1998. *Algorithms and Data Structures in VLSI Design*, Springer-Verlag, New York.

Rubio, A., Itazaki, N., Xu, X., and Kinoshita, K. 1997. An approach to the analysis and detection of crosstalk faults in digital VLSI circuits. *IEEE Trans. Comput. Aid. Des. 13*, 3.

Shepard, K. L. 1998. Design methodologies for noise in digital integrated circuits. In *Proceedings of the ACM/IEEE Design Automation Conference*, 94–99.

Shepard, K. L., Narayanan, V., Elemendorf, P. C., and Zheng, G. 1997. Global harmony: Coupled noise analysis for full-chip RC interconnect networks. In *ACM/IEEE Proceedings of the International Conference on Computer-Aided Design*, 139–146.

Tohr, T. S., Alt, H., Hetzel, A., and Koehl, K. 1998. Analysis, reduction and avoidance of crosstalk on VLSI chips. In *Proceedings of the International Symposium on Physical Design*, 211–218.

Vittal, A. and Marek-Sadowska, M. 1997. Crosstalk reduction for VLSI. *IEEE Trans. Comput. Aid. Des. 16*, 3 (March), 290–298.

Vittal, A., Chen, L. H., Marek-Sadowska, M., Wang, K.-P., and Yang, S. 1999. Crosstalk in VLSI Interconnections. *IEEE Trans. Comput. Aid. Des. Integ. Circ. Syst. 18,* 12 (Dec.), 1817–1824.

WROBLEWSKI, A., SCHIMPFLE, C. V., AND NOSSEK, J. A. 2000. Automated transistor sizing algorithm for minimizing spurious switching activities in CMOS circuits. In *Proceedings of the ISCAS*, 291–294.

XUE, T., KUH, E. S., AND WANG, D. 1994. Post global routing crosstalk risk estimation and reduction. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 616–619.

ZHOU, H. AND WONG, D. F. 1998. Global routing with crosstalk contraints. In *Proceedings of the IEEE/ACM Design Automation Conference*, 374–377.

ZURADA, J. M., JOO, Y. S., AND BELL, S. V. 1989. Dynamic noise margins of MOS logic gates. In *Proceedings of IEEE ISCAS*, 1153–1156.

Received June 2001; revised May 2002