False-Noise Analysis using Logic Implications

Alexey Glebov, Sergey Gavrilov, David Blaauw*, Supamas Sirichotiyakul, Chanhee Oh**, Vladimir Zolotov****

MicroStyle - Moscow, Russia, *University of Michigan (blaauw@umich.edu), **Motorola Inc. Austin, TX,

1 Abstract

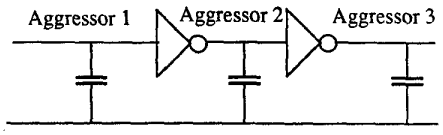
Cross-coupled noise analysis has become a critical concern in today's VLSI designs. Typically, noise analysis makes an assumption that all aggressing nets can simultaneously switch in the same direction. This creates a worst-case noise pulse on the victim net that often leads to false noise violations. In this paper, 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 and a newly proposed laterial propagation method. 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 paper, underscoring the need for false-noise analysis.

2 Introduction

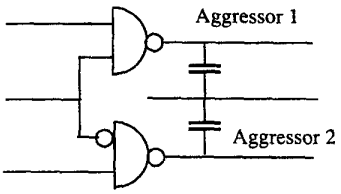
Advances in process technology have greatly increased the coupling capacitance in VLSI interconnects making it common for as much as 60-80% of interconnect capacitance to be coupling capacitance to other nets. This trend has lead to an increase in the noise injected on a net due to the unanticipated switching of neighboring nets, creating the necessity for noise analysis tools [1], [2], [15], [17]. In noise analysis, the net under consideration is commonly referred to as the *victim net*, while 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 [1], [15], [17]. 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 some cases, it may be necessary to shift the alignment times of the aggressor transitions by a predetermined amount, to account for the difference in aggressor driver and interconnect delays.) 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, while the likelihood of realizing the simultaneous switching scenario 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 [1], [2], [15]. 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. Also the timing window based approach does not identify cases where nets cannot switch in the same direction, for instance when they are connected by an inverter. Therefore, this approach may not identify all false noise failures, although it has been shown in practice to be relatively effective [1].



a) timing correlations between aggressors



b) logic correlations between aggressors

Figure 1. Logic relationships between aggressors

In order to identify all false noise failures, both the timing and logic correlations of the circuit must be taken into account. In [2], it was shown that in general, this problem can be represented as a

0-7803-7247-6/01/$10.00 © 2001 IEEE

search for a worst-case 2-vector test using a Boolean Constraint Optimization problem formulation. In [3], a method based on compatible observability don't care sets was proposed. In [16], a method is 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 chip-level routes, it is critical to perform false noise analysis at this level in large designs, and hence heuristic methods must be employed.

In this paper, we present a new approach for false noise analysis based on the generation and propagation of logic implications between signal pairs [4], [18]. Logic implications have been widely used in logic synthesis [5-8] as well as in peak current estimation [9], 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. 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, and we propose a new method to generate so-called lateral 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 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 circuits in few hours.

The initial formulation presented in this paper uses zero-delay implications which 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 [10]. In the last section of this paper, we show how our analysis can be extended for timed implications which 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%. The remainder of this paper is organized as follows: Section 2 discusses the generation and propagation of logic implications. 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.

3 Computing logic implications

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

$(a=V_a)\text{->}(b=V_b)$, 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 non-trivial SLIs, where a trivial SLI is an implication such as $(a=V_a)\text{->}(a=V_a)$. Similar to [9], 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 $(b=1)\text{->}(a=0)$ would belong to implication list L^a_H at node a, i.e. $b \in L_H{}^a$. In Figure 2, for example, the H^{n7}_L implication list at node $n7$ is $\{n3, n4, n8, n9\}$ and the implication list H^{n7}_H is $\{n11\}$.

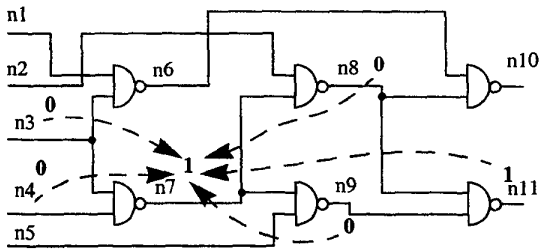


Figure 2. Example of SLIs in simple circuit.

The SLI generation algorithm consists of two steps: First, SLIs are generated as explained in Sections 2.1 and 2.2. Second, SLI are propagated through the circuit using the basic operations of list union, list intersection, and contra-positive law as explained in Section 2.3

3.1 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 $(a=V_a)\text{->}(x=V_x)$ is equivalent to implication $(x=\overline{V}_x)\text{->}(a=\overline{V}_a)$ due to the contra-positive law, and we consider both implications as a single implication at the input a_i.

Property 2: The presence of an SLI $(a_i=V_i)\text{->}(x=V_x)$ 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 two trivial cases:

1. If $(a=V_a)\text{->}(x=V_x)$ and $(a=\overline{V}_a)\text{->}(x=V_x)$ then x is Boolean constant.
2. If $(a=V_a)\text{->}(x=V_x)$ and $(a=\overline{V}_a)\text{->}(x=\overline{V}_x)$, then either $x=a$ or $x=\overline{a}$ and x has no dependence on other gate inputs.

It follows that for any non-trivial 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, $i=1,...,n$ and output x, implementing a Boolean function. If there are SLIs $(a_i=V_i)\text{->}(x=V_x)$ at inputs a_i then all these SLIs must have the same value of V_x.

Proof.

Consider SLIs $(a=V_a)\text{->}(x=V_x)$, $(b=V_b)\text{->}(x=\overline{V}_x)$ at inputs a and b and consider the following 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 $x=\overline{V}_x$, which is clearly a conflict.

Based on properties 1-3 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, $i=1,...,n$ and output x, implement a Boolean function. The set of non-trivial SLI $(a=V_a)\text{->}(x=V_x)$ for

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

$$p_x=p_{a1}+...+p_{am} + \bar{p}_{a1}\&...\&\bar{p}_{am}\&f(a_{m+1},...,a_n) \quad (1)$$

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 $a_{m+1},..., a_n$.

Proof.

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

$$p_x = p_{a1} + \bar{p}_{a1}\&f_1(a_2,...,a_3), \quad (2)$$

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

$$p_x = p_{a1} + ... + p_{ak} + \bar{p}_{a1}\&...\&\bar{p}_{ak}\&f_k(a_{k+1},...,a_n) \quad (3)$$

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

$$p_{a1} + ... + p_{ak} + \bar{p}_{a1}\&...\&\bar{p}_{ak}\&p_{a,k+1} = p_{a1} + ... + p_{ak} + p_{a,k+1} \quad (4)$$

we obtain:

$$p_x = p_{a1} + ... + p_{a,k+1} + \bar{p}_{a1}\&...\&\bar{p}_{ak+1}\&f_{k+1}(a_{k+2},...,a_n) \quad (5)$$

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

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

Since any Boolean function of one variable has two SLIs, we can conclude that f in (1) must have at least two variables, i.e. $n-m>1$. 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 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 logic function of the gate. However, if we decompose the AO22 gate into two AND gates and one OR gate and construct SLIs with use of the new internal nodes then, these SLIs will completely define the logic function of the gate. Therefore, to obtain effective generation of SLIs, the circuit must be decomposed into the gates listed in Corollary 2.

3.2 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 [11], where each ROBDD represents a single DC-connected component (DCCC) in the circuit. We then propose the following algorithm to generate 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 4 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 non-terminal 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 $f(x_i)$ in terms of the intermediate variable 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 three possible situations:

1. Both sons of vertex v are terminal vertices. In these 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*, $v=\bar{c}$.
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 non-terminal 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 low-son of v then $f=c\&w$.
   - if x is 1 and is low-son of v then $f=\bar{c}+w$.
   - if x is 0 and is high-son of v then $f=\bar{c}\&w$.
   - if x is 1 and is high-son of v then $f=c+w$.
3. Both child vertices of vertex v, controlled by variable c are non-terminal 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 the case, we introduce two additional intermediate variables x and y, where $x=c\&a$ and $y=\bar{c}\&b$. Then intermediate variable f of vertex v will be defined by the Boolean function $f=x+y$.

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.

3.3 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 contra-positive law. For example, let a 2-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^x_H (L^x_L) at output is calculated as the union of lists L^a_H and L^b_H (L^a_L and L^b_L). 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 contra-positive law:

$$\text{if } (a=v_a)\text{->}(b=v_b) \text{ then } (b=\bar{v}_b)\text{->}(a=\bar{v}_a)$$

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 contra-positive law application may be required.

In addition to these so-called *direct* implication propagations, we propose to use so called *lateral* SLI propagations. This allows us to find indirect implications, that are known to be particularly useful in logic optimization [7],[8].

Again, let us consider the 2-input AND gate with inputs *a,b* and output *x*. When we perform the list intersection between $H^a{}_L$ and $H^b{}_L$, we exploit the gate implication *(a=1 & b=1)->(x=1)* to obtain the implication list at the output $H^x{}_L = H^a{}_L \cap H^b{}_L$. However, we can also use the equivalent implication *(a=1 & x=0)->(b=0)* which will result in the following implication list at node *b*: $L^b{}_L = H^a{}_L \cap L^x{}_L$ and $L^b{}_H = H^a{}_H \cap L^x{}_H$. 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 following simple example in Figure 3.

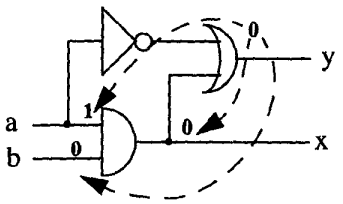


Figure 3. Circuit with possible lateral SLI propagation

In this example we obtain, through application of the contra-positive law and direct propagation across the inverter and OR gate two implication lists: $H^a{}_L = \{y\}$ and $L^x{}_L = \{y\}$. Due to lateral propagation, we therefore obtain the following implication at node *b*: $L^b{}_L = \{y\}$, i.e. *(y=0)->(b=0)*. It is clear that this SLI cannot be obtained by means of repeated direct propagation only.

Therefore, the overall proposed SLI propagation algorithm consists of the following stages. First, we perform multiple direct propagations with application of the contra-positive law until convergence. Then, we perform multiple passes of lateral propagation with application of the contra-positive law, until convergence. Each pass of lateral propagation is followed by one or more passes of direct propagation including application of the contra-positive law. 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.

```
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 contra-positive law.
          }
     2.2 For every gate in reverse topological order
         perform lateral SLI propagation with
         application of transitive and contra-positive laws.
   }
```

Figure 4. SLI propagation algorithm

4 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 is identified that inject coupled noise on the victim net, where each aggressor can potentially contribute a different amount of injected noise. A victim net and its associated aggressor nets is 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 the aggressor nets 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 following information:

1. a single victim node *V*
2. a set of aggressor nodes A_i that inject noise w_i $(i=1,...,n)$ on the victim net *V*
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 rising (*LowR* and *HighR*) or falling (*LowF* and *HighF*). The second four noise types correspond 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 three 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.

4.1 Forming the Constraint Graph

A constraint graph is an undirected graph $G=(V,E,w)$ of vertex set $V = \{v_1,...v_n\}$, 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 while the weight of a vertex is the amount of injected noise 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 state of the victim nets V_v^i and V_v^f and the initial and final state 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 state for this particular noise type. If for a victim / aggressor pair (v, a_i), either of the following two SLIs exist: $(v=V_v^i)->(a_i=\overline{V}_a^i)$ or $(v=V_v^f)->(a_i=\overline{V}_a^f)$ 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, while the aggressor must switch from high to low. Therefore, the presence of implication *(v=0)->(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)->(a_i=1)* would prohibit the aggressor a_i from switching since it would be already 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*.

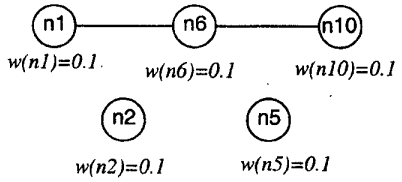
After the vertices of the constraint graph have been identified, we determine which edges exist in the graph. We examine each pair of vertices (v_i, v_j), $i \neq j$, $v_i, v_j \in V$. Again, we can determine if v_i and v_j can both switch in the required direction by searching for an SLI that renders their transitions logically incompatible. In this case, if we find either of the following two SLIs: $(a_i=V_a^i)\text{->}(a_j=\overline{V}_a^i)$ or $(a_i=V_a^f)\text{->}(a_j=\overline{V}_a^f)$, v_i and v_j cannot both participate in the maximal realizable aggressor set and an edge (v_i, v_j) is created in the constraint graph.

4.2 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* $W(K) = \sum_{u \in K} w(u)$, 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 [12]. 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 [13].

As simple example of our approach, let again the circuit in Figure 2 be considered. Let *n7* be 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 5, 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}*.



Maximum Weighted Independent Set:{n1,n2,n5,n10} with total weight 0.4

Figure 5. Example of the constraint graph

5 Extension to timed SLIs

Up to this point, we have only considered false noise analysis based on zero delay implications. These implications are only valid when the circuit has reached a stable state, i.e. 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 switching is impossible. This occurs when there are glitches in the circuit, as shown in the simple example in Figure 6a. In this example, nodes *z1* and *z2* are aggressor nodes of victim *v*. Using the contra-positive law and direct SLI propagation, the following implication will be computed under the zero delay model: *(z2=0)->(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 6b, and can inject noise simultaneously with aggressor *z2*.

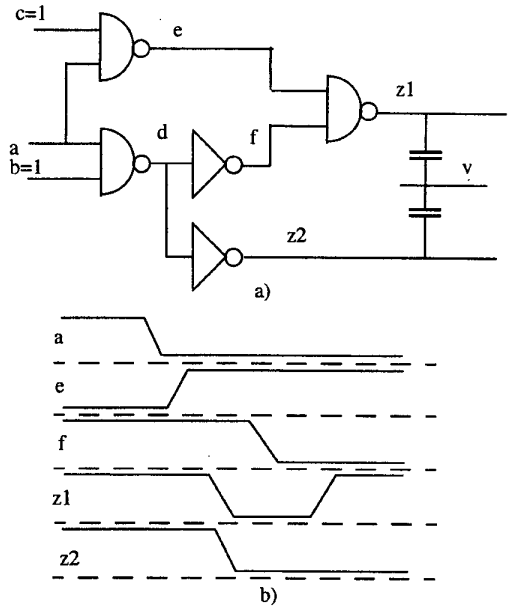


Figure 6. Noise injection by glitches

Therefore, zero-delay implication 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. In [9] simple timed SLIs are proposed using the following formulation:

$$(a(t)=V_a) \text{->} (b(t+T)=V_b) \qquad (6)$$

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 1. An exclusive timed SLI (or E-SLI) is the following relations between signals *a, b*:

$$(\forall(t \in [t_1, t_2])\, a(t)=V_a) \text{->}$$
$$(\forall(t \in [t_1 + T_1, t_2 + T_2])\, b(t)=V_b) \qquad (7)$$

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:

$$(a=1) \text{->} (b=0)\,(T_1,T_2) \qquad (8)$$

Definition 2. An inclusive timed SLI (or I-SLI) is the following relations between signals *a, b*;

$$\exists(t \in [t_1, t_2])\, a(t)=V_a) \text{->} \qquad (9)$$

$(\exists(t \in [t_1 + T_1, t_2 + T_2])\ b(t)=V_b)$

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:

$$(a=1) =>> (b=0)\ (T_1,T_2) \qquad (10)$$

We can see that (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 3. An E-SLI or I-SLI is said to be expanding if $T_2>T_1$. An E-SLI or I-SLI is said to be contracting if $T_1>T_2$. An E-SLI or I-SLI is *neutral* if it is both non-expanding and non-contracting (i.e. $T_1=T_2$). Clearly, zero delay SLIs and the fixed delay SLIs (6) are neutral.

Let us now examine a logic gate with input a and output b and zero delay SLI $(a=1)->(b=1)$. 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 7. Similarly, 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 7 that when signal a is at a stable high value during the entire interval $[t1,t2]$, 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 7. Therefore, we can formulate the following exclusive-SLI:

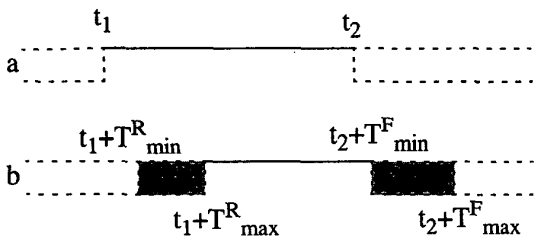


Figure 7. Timed SLI $a->b\ (T^R_{max}, T^F_{min})$.

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

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 exists 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 7. We can therefore formulate the following inclusive-SLI:

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

For typical gates, the time intervals of I-SLIs will shrink as we propagate them through the circuit, while the time intervals of E-SLIs will expand. It is clear that forward and lateral propagation rules and contra-positive laws can be formulated for timed E-SLIs and I-SLIs analogous to that for zero-delay SLIs. Also, we can use the same implicit decomposition for ROBDD for complex gates and formulate the constraint graph discussed in Section 3.1 for E-SLIs and I-SLI. Therefore, we can use the results of timing analysis, combined with zero-delay SLIs, to calculate timed E-SLIs and I-SLIs in the circuit. Timed SLIs can then be used to perform a conservative false noise analysis for circuits with glitches.

6 Implementation and experimental results

The presented algorithms were implemented in an industrial noise analysis tool called *ClariNet* [1]. The system was architected using a separate logic analysis engine call DiNo, which currently generates the zero delay 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 pre-characterizes 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 8 illustrates both the block and chip level analysis methodology.

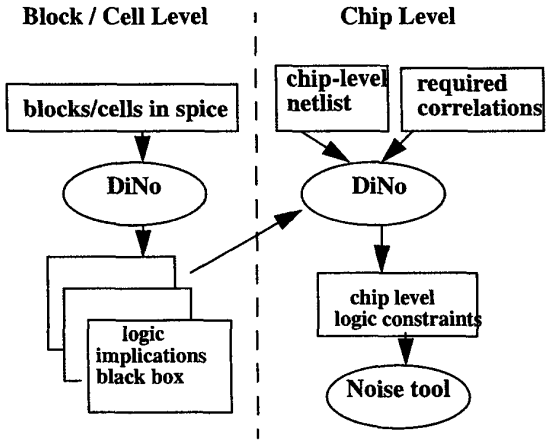


Figure 8. Block diagram of SLI based noise analysis algorithms.

In Table 1, we show the number of generated SLI for a number of circuits using the proposed SLI generation and propagation approach. The first two circuits are ISCAS benchmark circuits [14], while the remaining circuits are industrial circuits synthesized using a commercial synthesis tool. The third and fourth column show the number of generated SLIs using only direct propagation and using both direct and lateral propagation respectively. In the fifth column the percentage increase in the number of generated SLI due to the use of lateral propagation is recorded. The final column shows the number of SLI as a percentage of the total number of node pairs.

The results in Table 1 demonstrate the effectiveness of the lateral SLI propagation proposed in this paper, which increased the number generated SLI on average by 38%. The total number of SLIs, as a percentage of the number of nodes pairs ranges from 5 - 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 circuit, as shown in Table 2. Circuit *plldriver* and *cntrl* are small control blocks. Circuit *xbar* is a small crossbar switch, circuit *rot8*, *rot16* and *rot32* are 8, 16 and 32 bit shifters, circuit *adder32* is a 32 bit adder, circuit *control* is a large control block. 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 column 5 shows the number of failures with false noise analysis as presented in this paper. Note that the num-

circuit	#nets	#SLIs w/o laterial	#SLIs with laterial	% laterial	#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	559	27218	32802	21	10

Table 1. Results of testing SLI propagation algorithm

circuit	#nets	#SLIs computed	#failures	#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
control	46168	10910	11603	10452	10

Table 2. Results of SLI based noise analysis on block level.

ber of failures can exceed the number of nodes, since there are several noise types for each net. The final column shows the percentage decrease in the number 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. The test results show that for the *control* circuit the reduction obtained from using false noise analysis is smaller than for the other blocks. This is due to the fact that it is a large random block constructed using standard place and route tools. Therefore, the likelihood that a significant number of the aggressor nets in a noise cluster have a close logic distance is small. We therefore found our analysis to be especially effective for structures where routing is highly controlled, such as bus structures, and the neighboring nets have a small logic distance. Fortunately, these are often also the interconnects with the most severe noise problems. Besides reducing the number of noise failures, SLIs reduce the noise value of the nets, that remain failures. For example for the circuit *control* the proposed approach reduces the noise estimation for 3500 nets.

7 Conclusions

In this paper, we presented a new approach for false noise analysis. We propose the use of logic implications for eliminating aggressor nets that cannot simultaneously switch. We first show how simple logic implications can be effectively generated and propagated in the circuit. We prove that SLIs fully represent the logic function of a circuit only if it consists of simple NAND, NOR, and INV gates. We then show how SLIs can be generated for complex gates using implicit decomposition of their ROBDD representation. We also introduce a new, so-called, *lateral* propagation method to increase the number of obtained SLIs in a circuit. Using the SLIs we show how the false noise problem can be formulated as a constraint graph and solved as a maximum weighted independent set problem. Finally, we show how the proposed SLIs can be extended to timed implications for conservative false noise analysis in non-glitch-free circuits. The presented algorithms were implemented and used on industrial circuits. The results show a reduction of 27% in the number of failures on average, underscoring the importance of false noise analysis.

8 References

[1] R.Levy, D.Blaauw, G.Braca, et.al. "ClariNet: A noise analysis tool for deep submicron design", DAC-2000, pp.233-238.

[2] P.Chen, K.Keutzer. "Towards True Crosstalk Noise Analysis", ICCAD-99, pp.132-137.

[3] D.A.Kirkpatrick, A.L.Sangiovanni-Vincentelli. "Digital Sensitivity: Predicting Signal Interaction using Functional Analysis", ICCAD-96, pp.536-541.

[4] F.M.Brown. "Boolean reasoning", Kluwer Academic Publishers, 1990.

[5] G.Hachtel, R.Jacoby, P.Moceyunas, C.Morrison. "Performance Enhancements in BOLD using Implications", ICCAD-88, pp.94-97.

[6] W.Kunz, P.R.Menon. "Multi-Level Logic Optimization by Implication Analysis", ICCAD-94, pp.6-13.

[7] R.I.Bahar, M.Burns, G.D.Hachtel, et.al. "Symbolic Computation of Logic Implications for Technology-Dependent Low-Power Synthesis", ISPLED-96.

[8] W.Long, Y.L.Wu, J.Bian. "IBAW: An Implication-Tree Based Alternative-Wiring Logic Transformation Algorithm", ASPDAC-2000, pp.415-422.

[9] S.Bobba, I.N.Hajj. "Estimation of maximum current envelope for power bus analysis and design", Int. Symp. on Phys. Des., 1998.

[10] A.Wroblewski, C.V.Schimpfle, J.A.Nossek. "Automated Transistor Sizing Algorithm for Minimizing Spurious Switching Activities in CMOS Circuits", ISCAS-2000, pp.291-294.

[11] R.E.Bryant. "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Computers, 1986, v.35, pp.677-691.

[12] M.R.Garey, D.S.Johnson. "Computers and Intractability, A Guide to the Theory of NP-Completeness", Freeman, 1979.

[13] E.Loukakis, C.Tsouros. "An Algorithm for the Maximum Internally Stable Set in a Weighted Graph", Intern. J. Computer Math., 1983, v.13, pp.117-129.

[14] F. Brglez, H.Fujiwara, A Neutral Netlist of 10 Combinatorial Benchmark Circuits", Proc. IEEE ISCAS, IEEE Press, Piscataway, N.J., 1985, pp.695-698

[15] K.L. Shepard, V. Narayanan, P.C.Elementary and G.Zheng, "Global Harmony: Coupled noise analysis for full-chip RC interconnect networks", Proc. ICCAD ,1997 pp. 139-146

[16] A. Rubio, N. Itazaki, X. Xu, and K. Kinoshita, "An Approach to the Analysis and Detection of Crosstalk Faults in Digital VLSI Circuits", IEEE Trans. on CAD, Vol. 13, No. 3, 1997

[17] Shepard K.L. "Design methodologies for noise in digital integrated circuits", Proc., DAC, 1998, pp. 94-99

[18] Tafertshower P., Ganz A., Antreich K.J. "IGRAINE - An Implication GRaph-bAsed engINE for Fast Implication, Justification, Propagation", IEE Trams. on CAD, Vol. 19, No 8, 2000, pp. 907-927