# Behavioral Transformations to Increase Noise Immunity in Asynchronous Specifications

Alexander Taubin
The University of Aizu, Japan

Alex Kondratyev
The University of Aizu, Japan

Jordi Cortadella
Univ. Politècnica de Catalunya, Spain

Luciano Lavagno
Politecnico di Torino, Italy

## Abstract

*Noise immunity is becoming one of the most important design parameters for deep-sub-micron (DSM) technologies. Asynchronous circuits seem to be a good candidate to alleviate the problems originated by simultaneous switching noise. However, they are also more sensitive than synchronous ones to spurious signal transitions and delay variations produced by crosstalk noise. This paper addresses the problem of analyzing and synthesizing asynchronous circuits with noise immunity being the main design parameter. The techniques presented in the paper focus on cross talk noise and tackle the problem from the behavioral point of view.*

## 1 Introduction

The technology of interest for research on CAD for integrated circuits aimed at a 10-15 year time frame should assume feature sizes below 100 nm (deep sub-micron or DSM). According to the National Technology Roadmap for Semiconductors (NTRS'97) the prediction for the year 2009 is: technology norm $\leq 0.07\mu m$, 520M transistors per chip, 2500 MHz on-chip clock, area $\geq 620mm^2$ and 8-9 wiring levels [5, 10]. It is now common to point out that for DSM noise immunity is becoming a metric comparable in terms of importance to area, timing and power [5, 9, 25]. Among noise sources one usually considers the following:

- cross coupling capacitances and charge sharing (crosstalk),
- power and ground noise, including simultaneous switching noise,
- alpha particle radiation,
- substrate noise,
- electro-magnetic radiation from external sources.

Immunity against external noise sources (the last three items) can be ensured for DSM by the same means as for VLSI (by means of appropriate packaging and electro-magnetic isolation). Contrary to that, the problem of internal noise immunity becomes much more severe due to the following reasons:

- increasing capacitive coupling (closer packing, nonuniform geometrical scaling, etc.),
- lower noise margins (due to scaling down power supply and threshold voltage),
- higher speed of voltage changes.

Crosstalk and simultaneous switching noise are identified as a major source of problems during DSM layout synthesis.

The absence of a common clock in asynchronous systems somewhat helps to avoid the simultaneous switching noise. Recent investigations [19] show that by a self-timed approach one might reduce not only simultaneous switching noise, but also the high frequency components of noise (400% reduction for peak switching current [19]) and Electro-Magnetic Interference by an order of magnitude with respect to a comparable synchronous design.

However, for crosstalk noise an asynchronous approach gives no immediate advantage in comparison to synchronous circuits. A straightforward way to reduce noise would be to constrain the layout synthesis step, by forbidding the adjacency of noisy wires. The drawback of this approach is that the information on which wires are subject to crosstalk will be available, with current synthesis-based methodologies, only at a later stage (namely after extraction and back-annotation). Crosstalk noise reduction thus requires a potentially large number of iterations between layout and analysis steps, without any guarantee of convergence.

Thus researchers are beginning to advocate a new design flow, in which design errors (in particular due to noise) are identified as early as possible in the flow, and avoided by *constraint propagation to subsequent steps*. Hence one can speak of "design for low noise" in the same way as design for low power or design for testability.

An important step in this direction was made by Kirkpatrick and Sangiovanni-Vincentelli in [11, 12]. They suggested to extract information about the possible sources of noise at the *logic* (gate netlist) level, and use it to guide the layout tools. The goal of this paper is to extend that work to the *asynchronous sequential logic level*, so as to raise the level of abstraction to the asynchronous behavior level. The possible advantages are twofold:

1. we can apply to the reduction of noise local transformations that are only possible at the behavior level (e.g., reshuffling of transitions),

2. we provide a uniform approach for noise analysis and avoidance in both sequential and combinational circuits, while the methods of [11, 12] were mainly targeted for combinational circuits (the analysis of sequential parts was complex and approximate).

These techniques can be combined together with traditional methods for constrained layout synthesis. Behavior transformations make the layout problem much easier, because they partly (and sometimes completely) remove the sources of crosstalk noise.

The novelty of this work with respect to existing approaches for noise avoidance is illustrated in Figure 1.

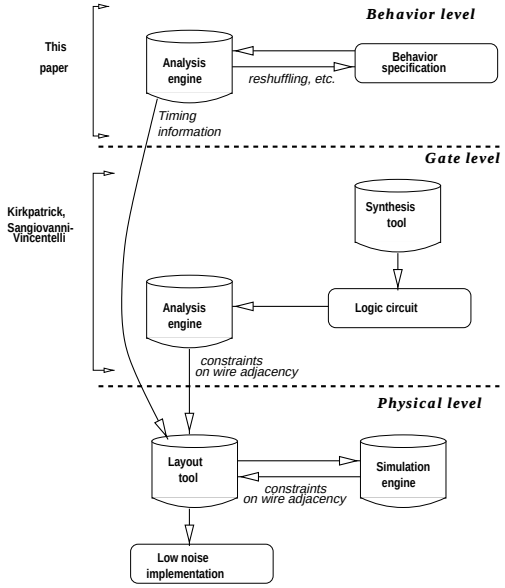


**Figure 1. Techniques for low noise design**

In this paper we concentrate on noise avoidance within control circuits only, for the following two main reasons:

1. Datapaths are very regular, hence during layout the task of noise avoidance within the datapath itself is easier to solve. Moreover, datapath and control can be segregated reasonably well, thus reducing the possibility of crosstalk between these two.

2. In general it is possible to use a behavioral description in which the datapath transitions are specified together with control events, as symbolic events. This extension would allow one to consider also the datapath within the suggested framework.

The paper is organized as follows. Section 2 discusses the source of crosstalk noise and its effects in terms of circuit faults. Section 3 reviews some asynchronous behavior models: State Graphs (SG) and Signal Transition Graphs (STG). Sections 4 and 5 present and extend the concept of digital sensitization to asynchronous sequential circuits [11, 12], and apply it to the analysis of crosstalk noise. Section 6 discusses under which conditions one can use behavioral (SG and STG) models for noise analysis, instead of detailed simulation or conditions developed for combinational logic in previous work. Section 7 sketches some of the algorithms for analysis (noise-isolated pair extraction) and synthesis (aggressor concurrency reduction and reduction of temporal adjacency) based on the previous discussion. Section 8 describes preliminary experimental results that show the promise of our approach. Section 9 concludes the paper and discusses several topics for future research.

## 2 Crosstalk Noise and Circuit Faults

Two main types of faults can be caused by noise [20].

**Definition 2.1 (Noise-Generated Faults)**

1. **Transient fault***: a signal temporarily changes its logical value due to noise.*

2. **Delay fault***: the delay of a* wire or wire segment *(e.g., a segment of wire after a fanout fork) exceeds the specified worst case.*[1]

The source of crosstalk noise lies in the interaction between adjacent wires, due to the coupling capacitance between them (hence it is also called coupling noise). Of course, working at the logic level means that we must talk about the *future possibility* of such interaction, if layout tools do not take appropriate precautions (e.g., shielding, segregation, and so on).

For an arbitrary pair of wires **(a, v)** we will call **a** the *aggressor* and **v** the *victim*, if a transition on **a** might produce a noise-generated fault on (any segment of) **v**. Clearly, the same pair of wires can play opposite roles in different phases of the operation of the circuit, and even at the same time.

Noise analysis at the layout level can be confined to immediate adjacent lines, since they act as shields and protect a "victim" from other "aggressors" [30]. The total crosstalk noise voltage on a wire can be computed as the sum of the individual noise contributions of each of adjacent aggressors.

Transient and delay faults generated by noise are of a different nature and can happen in two different scenarios.

1. In order for a transient fault to occur on a victim line that should normally keep a stable value, a short pulse must be generated on the victim due to transitions on adjacent aggressor lines. Fortunately, the voltage swing of a victim line in response to a transition on a *single* adjacent line is small enough [9, 8] and can be successfully filtered by the rest of the circuit. However, the cumulative effect of two

[1] This case is dangerous even for asynchronous circuits, because it can violate some timing assumptions (e.g., Fundamental Mode or Isochronic Fork) that ensure correct operations of an asynchronous circuit.

lines switching in the same direction might produce a pulse sufficient to be sensed by gates connected to the victim line. Therefore for transition faults we will consider only the situations when two or more aggressors are switching the same direction.

Figure 2 from [9] illustrates this case. The coupling voltage noise on the center wire ($\Delta V_{coup}$) can be expressed as: $\Delta V_{coup} = \Delta V_{sw} * 2C_c/(C_{sub} + 2C_c)$, where $C_c$ is the coupling capacitance between two adjacent wires, $C_{sub}$ is the capacitance between the center wire and the substrate, and $\Delta V_{sw}$ is the switched voltage [9].

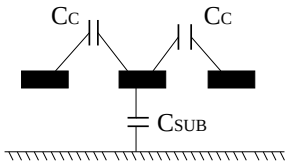


**Figure 2. Simplified cross section of wires**

With the help of this simple model, one can check that even for $0.5\mu m$ CMOS technology (with single minimal spacing) the crosstalk voltage can be larger than the total noise margin ($\Delta V_{coup} = 0.59 \cdot \Delta V_{sw}$). The explanation of this phenomenon lies in nonuniform scaling: lateral dimensions are aggressively scaled, while vertical dimensions are practically unchanged from one generation to the next. This results in the increase of the horizontal coupling component, and a larger swing of the crosstalk voltage.

A known design solution is to space the tracks further apart than the minimum. However, this does not extend to DSM. Even with a line spacing that is twice the minimum, for a 70 nm technology the coupling noise due to two aggressors switching in the same direction, with the victim wire in between, will exceed 30% of $V_{dd}$ [5].

Therefore, solving the problem of crosstalk noise in DSM requires a design methodology that does not tackle it at the layout level only, but at the logic and behavioral levels as well.

2. When considering delay faults due to noise, one should consider a different scenario. A delay fault happens because of an interaction between wires having simultaneous transitions in opposite directions [11, 12]. The coupling between them might induce an additional delay by injecting charge on the more weakly driven wire, which will slow down its switching[2].

Faults generated by noise are dangerous only if they can influence the rest of the circuit. In this sense we can consider *noise generation* and *noise propagation* conditions. In order to increase noise immunity, one can either reduce noise generation or reduce noise propagation. These issues are considered in more detail in Section 4.

[2] The opposite case, making some transition faster when neighboring signals switch in the same direction [8, 32], might also be a problem. For example, it could violate timing assumptions such as isochronic forks. However, it does not need special consideration, since the effects of speeding up some transitions in asynchronous circuits can be modeled as slowing down other transitions.

# 3 Basic notions

## 3.1 Basic definitions about Boolean Functions

An incompletely specified (scalar) Boolean function is a functional mapping $F : B^n \rightarrow \{0, 1, \Leftrightarrow\}$, where $B = \{0, 1\}$ and '$\Leftrightarrow$' is a *don't care* value. The subsets of the domain $B^n$ in which $F$ has a 0, 1 and don't care value are respectively called the *OFF-set*, *ON-set* and *DC-set* of $F$. $F$ is *completely specified* if its DC-set is empty. A point (i.e., binary vector of values) in the domain $B^n$ of a function $F$ (not necessarily in its ON-set) is called a *minterm*.

Let $F(x_1, x_2, \ldots, x_n)$ be a Boolean function of $n$ Boolean variables. The set $X = \{x_1, x_2, \ldots, x_n\}$ is called the *support* of the function $F$. In this paper we shall mostly be using the notion of *true support*, that is defined as follows. A variable $x \in X$ is *essential* for function $F$ (or $F$ is dependent on $x$) if there exist at least two minterms $v1, v2$ different only in the value of $x$, such that $F(v1) \neq F(v2)$. The set of essential variables for a Boolean function $F$ is called the *true support* of $F$.

Let $F(X)$ be a Boolean function with support $X = \{x_1, x_2, \ldots, x_n\}$. The *cofactor* of $F(X)$ with respect to $x_i$ ($\overline{x_i}$) is defined as $F_{x_i} = F(x_1, x_2, \ldots, x_i = 1, \ldots, x_n)$ ($F_{\overline{x_i}} = F(x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$, respectively). The well-known Shannon expansion of a Boolean function $F(X)$ is based on its cofactors: $F(X) = x_i F_{x_i} + \overline{x_i} F_{\overline{x_i}}$.

The existential abstraction of a function $F(X)$ with respect to $x_i$ is defined as $\exists_{x_i} F = F_{x_i} + F_{\overline{x_i}}$. The existential abstraction can be naturally extended to a set of variables. The *Boolean difference*, or *Boolean derivative*, of $F(X)$ with respect to $x_i$ is defined as $\delta F/\delta x_i = F_{x_i} \oplus F_{\overline{x_i}}$.

A *controlling set* for a boolean function $F$ with a true support $\{x_1, \ldots, x_n\}$ is a a set $C$ of values of variables $\{x_1, \ldots, x_i\}$ such that $F(C, x_{i+1}, \ldots, x_n)$ has the same value, for *any* values of the other variables $x_{i+1}, \ldots, x_n$. Otherwise a set of values at $\{x_1, \ldots, x_i\}$ is called *non-controlling*.

The *characteristic function* of a set $S$ of $n$-dimensional boolean vectors is a boolean function $F : B^n \rightarrow B$ such that $s \in S \Leftrightarrow F(s) = 1$.

## 3.2 Behavioral models and Logic Implementability

In this subsection we assume the reader to be familiar with Petri nets, a formalism used to specify concurrent systems. We refer to [23] for a general tutorial on Petri nets and to [15] for a review of applications of Petri nets to asynchronous design.

### 3.2.1 State Graphs

A *State Graph* (SG) is a labeled directed graph whose nodes are called *states*. Each arc of an SG is labeled with an *event*,

that is a rising ($a+$) or falling ($a\Leftrightarrow$) transition of a signal $a$ in the specified circuit. We also allow the notation $a*$ if we are not specific about the direction of the signal transition. The set of signals of an SG is called $X = I \cup O$, where $I$ and $O$ denote the sets of *input* and *output* signals respectively. The behavior of the input signals is determined by the environment, whereas the behavior of the output signals must be implemented by the circuit[3]. We write $s \xrightarrow{a} (s \xrightarrow{a} s')$ if there is an arc from state $s$ (to state $s'$) labeled with $a$.

A labeling function $v : S \rightarrow \{0,1\}^n$ assigns a vector of signal values to each state ($n = |X|$). We will call $v_a(s)$ the value of signal $a$ in state $s$. An SG is *consistent* if:

$$
\begin{aligned}
s \overset{a+}{\Leftrightarrow} s' \quad &\Longrightarrow \quad v_a(s) = 0 \ \wedge \ v_a(s') = 1 \\
s \overset{a-}{\Leftrightarrow} s' \quad &\Longrightarrow \quad v_a(s) = 1 \ \wedge \ v_a(s') = 0 \\
s \overset{b*}{\Leftrightarrow} s' \ \wedge \ a \neq b \quad &\Longrightarrow \quad v_a(s) = v_a(s')
\end{aligned}
$$

### 3.2.2 Signal Transition Graph

A Signal Transition Graph (STG) is a Petri net in which transitions are labeled with the same type of events that we defined for SGs, i.e. rising and falling signal transitions [4].

An STG has an associated SG in which each reachable marking corresponds to a state and each transition between a pair of markings corresponds to an arc labeled with the same event as that labeling the transition.

Although STGs with bounded reachability space and SGs have the same descriptive power, STGs can usually express the same behavior more succinctly.

Figure 3.a depicts an STG with three signals. For simplicity, places with only one input and output transitions are omitted. Figure 3.b shows the corresponding SG, with states labeled with the binary vectors of the signal values. The SG is consistent.

In addition to consistency, *speed independence* and *Complete State Coding* (CSC) are two properties required for an SG to be implementable as a hazard-free asynchronous circuit [13].

## 3.3 Excitation and quiescent regions. Next-state function.

The *excitation region* (ER) of event $a^*$ is the set of states such that $\forall s \in ER(a^*) : s \xrightarrow{a^*}$. The *quiescent region* (QR) of event $a^*$ with excitation region $ER(a^*)$, is a set of states in which $a$ is stable and keeps the same value, i.e. for $ER(a+)$ ($ER(a\Leftrightarrow)$ ), $a$ is equal to 1(0) in $QR(a+)$ ($QR(a\Leftrightarrow)$).

In Figure 3.b, $\mathsf{ER}(x\Leftrightarrow) = \{101, 111\}$ and $\mathsf{QR}(x\Leftrightarrow) = \{001, 011, 010\}$. The symbol $0^*$ ($1^*$) indicates that a rising (falling) transition of the corresponding signal is enabled in that state.

The implementation of an SG as a logic circuit is done through the definition of the *next-state function* for each output signal and binary vector. It is defined as follows:

$$
f_a(z) = \begin{cases} 1 & \text{if } \exists s \in \mathsf{ER}(a+) \cup \mathsf{QR}(a+) \text{ s.t. } v(s) = z \\ 0 & \text{if } \exists s \in \mathsf{ER}(a\Leftrightarrow) \cup \mathsf{QR}(a\Leftrightarrow) \text{ s.t. } v(s) = z \\ \Leftrightarrow & \text{otherwise} \end{cases}
$$

The next-state function $f_a$ is correctly defined when the SG has the CSC property, i.e., when there is no pair of states $(s, s')$ such that $v(s) = v(s')$ and $s \in \mathsf{ER}(a+) \cup \mathsf{QR}(a+)$ and $s' \in \mathsf{ER}(a\Leftrightarrow) \cup \mathsf{QR}(a\Leftrightarrow)$. Note that $f_a$ is an incompletely defined function with a *don't care* (DC) set corresponding to those binary vectors without any associated state in the SG.

In the SG of Figure 3.b, the DC set is empty since all binary vectors have a corresponding state in the SG. As an example, $f(101) = 011$ since signals $x$ and $y$ are enabled in that state. The Karnaugh maps for the next-state functions are depicted in Figure 3.c.

From the next-state functions, a logic circuit can be derived by implementing the boolean equation of each output signal as an atomic complex gate, as shown in Figure 3.d. This is called a *complex gate implementation*. If the original SG is consistent and speed-independent then the corresponding complex gate implementation will be speed-independent as well [22].

# 4 Digital Sensitivity and Noise Avoidance

The digital sensitivity approach [11, 12] suggests a constraint-driven design methodology in which the information extracted from logical or timing correlations between signals are provided to a layout synthesis tool (a constraint-based channel router). In the simplest case this information indicates which sets of wires should not be routed adjacent to each other, because of the *logical* possibility of noise. The latter gives a set of (usually conservative) constraints for a layout tool, that if satisfied guarantee that a circuit does not have noise-generated faults.

From the discussion of Section 2 on the sources of the faults, it follows that noise generation always happens due to *several signals switching at the same time*. Therefore, conditions for noise generation can be checked by analyzing the *concurrency relations* between signal transitions. The information on concurrency is explicitly represented in the behavior models that we consider. In particular, two transitions $a*$ and $b*$ are concurrent ($a^* \,||\, b*$) if they are enabled in the same state of an SG, and firing of one of them cannot disable the other. Formally, in the case when $a* = a+$ and $b* = b+$ e.g. $a+$ concurrent to $b+$ means that there exists state $s$ in an SG such that:

1) $s \xrightarrow{a+} s1 \wedge s1 \xrightarrow{b+}$ and 2) $s \xrightarrow{b+} s2 \wedge s2 \xrightarrow{a+}$.

The following SG-based algorithm identifies sets of wires that might potentially be sources of noise faults. This algorithm formalizes the conditions of occurrence of noise faults which were presented in Section 2.

**Example 4.1** *Let us apply the algorithm in Figure 4 to the example in Figure 3. In the corresponding SG there are only two states from which signals might fire concurrently: 11*0**

[3] In general this may require the creation of new *internal* signals. In this paper we only consider SGs that are already *implementable* in a given standard cell library. See [6] for techniques to ensure gate-level implementability.

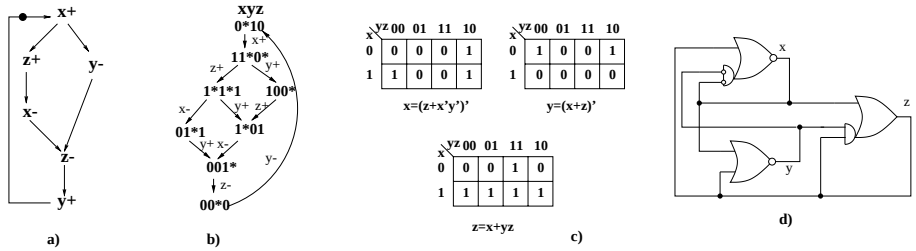


**Figure 3. (a)** STG**, (b)** SG**, (c) next-state functions, (d) complex-gate implementation.**

**Input:** Initial SG $A$
**Output:** Set $Noise$ of potentially noisy wires
**foreach** $s \in A$ **do**
/* Check delay faults */
**foreach** pair $a, b$ s.t.
$v_a(s) = 1^* \wedge v_b(s) = 0^* \wedge a\text{-}||b+$ **do**
$Del\_faults = Del\_faults \cup \{a, b\}$;
/* Check transient faults */
**foreach** triple $a, b, c$ s.t.
$v_a(s) = 1^* \wedge v_b(s) = 1^* \wedge v_c(s) = 1 \wedge a\text{-}||b\text{-}$ **do**
$Trans\_faults = Trans\_faults \cup \{a, c, b\}$;
**foreach** triple $a, b, c$ s.t.
$v_a(s) = 0^* \wedge v_b(s) = 0^* \wedge v_c(s) = 0 \wedge a+||b+$ **do**
$Trans\_faults = Trans\_faults \cup \{a, c, b\}$
**enddo**
$Noise = Trans\_faults \cup Del\_faults$

**Figure 4. Analysis for noise fault**

*and 1*1*1. The former implies the possibility of delay faults for the pair of wires $(y, z)$, while the latter might be the source of a transient fault for the triple $(x, z, y)$, where $z$ is the victim and $x, y$ are the aggressors (for a triple, the order of signals is important; the victim is always placed between the aggressors). Hence in this example $Noise = \{(y, z), (x, z, y)\}$.*

For those signals that successfully pass the check about noise faults (i.e. for those that are not included in the $Noise$ set) we will talk about *logic separation*. Hence, so far the notion of logic separation of signals is based completely on the analysis of the ordering/concurrency relations between transitions in the behavioral (STG) specification. However, two transitions that are concurrent in the specification do not necessarily happen ("fire") concurrently in the circuit. This is because delays of circuit gates and wires may actually cause two signals that are specified as concurrent (e.g., pairs of gates triggered by the same transition) to actually always occur in a specific order. For this reason, the concurrency relation in the specification is a conservative approximation of reality.

Refinement of concurrency is thus possible, either by considering the implementation (when delay estimation becomes possible) or by making some "reasonable" timing assumptions, which must then be satisfied (e.g., by transistor sizing or constrained routing) by the implementation. If signals are not logically separated, but according to these timing informations we can conclude that, for example, they cannot switch at the same time, we will talk about *temporal separation*.

Temporal separation can be checked in a way similar to the algorithm of Figure 4. In [29, 14] it was shown that timing constraints can be captured at the STG level, by introducing timing arcs that reduce the potential concurrency of signals. From such STG with timing arcs one can generate the SG and apply the same algorithm to it. The resulting $Noise$ set will contain tuples of signals that do not have logical nor temporal separation.

A further refinement of these sets of potentially noisy signals could be obtained by considering the *noise propagation conditions*. If some signals are not separated (logically or temporally) and could generate a crosstalk noise, we would still have no problems if the generated noise faults could not propagate through the circuit. Suppose, for example, that a transient fault on wire $a$ happens at a time when for all fanout gates of $a$ some input different from $a$ has a controlling value. Then the behavior of the fanout of $a$ does not actually depend on the value $a$, and any transient fault affecting $a$ could not propagate within the circuit (the formal conditions of noise propagation are considered in Section 5).

If for a set of signals the conditions for noise generation or propagation are not satisfied, then the set is said to be *isolated from crosstalk noise*. The level of *digital isolation* from crosstalk noise is the percentage of signal pairs for which the circuit is isolated from crosstalk noise (*isolated pairs*), considering each signal as a victim candidate.

According to [12], about 50% of digital isolation is needed to have a reasonable impact on the reduction of circuit area after layout, due to a smaller set of layout constraints.

The main improvement introduced by this work with respect to the contribution of [11, 12] is the consideration of behavioral *sequential* models in the digital sensitivity approach. The STG- and SG-based methods that we suggest can be useful both for efficient derivation of isolated pairs and for increasing the digital isolation through specification transformations.

In Section 8 it will be shown that the combination of behavior transformations together with reasonable timing assumptions allows us to reach about 80% of digital isolation in average.

The methodology for low-noise asynchronous circuit design includes the following steps:

1. **Analysis of noise-critical sets of signals**. This reduces

to checking logical and temporal separation on a behavior specification, and providing the layout tools with constraints on adjacency of critical wires.

2. **Improving logical and temporal separations of signals**. This might be achieved by *optimizations of behavioral specifications* aimed at noise avoidance. These optimizations include

   a) signal reshuffling and concurrency reduction [7] (logical separation) and

   b) timing assumptions [14] (temporal separation).

3. **Logic optimization for low noise**. Logic functions for gates might be chosen according to a cost function evaluating noise. This requires different approaches to minimization (e.g., introducing redundancy in the circuit in order to avoid noise propagation). Similar ideas were exploited in an alternative wiring scheme from [18].

4. **Clustering for reduction of crosstalk noise**. Interconnections could be divided into local and global levels.

   - The global level (top layers of interconnect structure [2]) should include all (or most of) the long wires that could be dangerous from the point of view of crosstalk just because of the long cross couplings.
   - The local level (lower layers of interconnect) should be used only for short distance interconnects, but could still suffer from crosstalk noise because of high density (local lines are usually restricted to lengths less than 3mm, however starting at technologies beyond 0.18 $\mu$m they become vulnerable to crosstalk as well [25]).

   We could potentially exploit this fact and divide large circuits into relatively small "islands", connected with each other by global and vertical interconnects. Crosstalk noise analysis and synthesis could be done separately for each "island", that could be small enough to apply our behavior specification techniques

The techniques presented in the paper mainly concern items 1, 2 and 3 from the list above. Item 4 gives a more systematic approach for the avoidance of noise, but it is still an open area for research.

## 5 Digital sensitivity analysis

The analysis of logical or temporal separation[4] requires one to consider each signal as a potential victim, and identify sets of states in which the signals might be a subject to noise faults.

**Definition 5.1 (Fault Sets)**

1. *The* **transient fault set** *with respect to signal $b$ (denoted by $S_{tr}(b)$) is the set of SG states such that $\forall s \in S_{tr}(b)$ there are two enabled signals $a$ and $c$, which have the same value as $b$.*

2. *The* **delay fault set** *with respect to signal $b$ (denoted by $S_{del}(b)$) is the set of SG states such that $\forall s \in S_{del}(b)$, $b$ is enabled and there exists another signal $a$ enabled in $s$, with the opposite logical value to $b$.*

3. *The* **noise fault set** *with respect to signal $b$ (denoted by $S_n(b)$) is the union of the transient and delay fault sets.*

Note that these three sets are subsets of those (of aggressor/victim signal tuples) computed by the Algorithm in Figure 4. The conditions under which a noise tuple is not actually dangerous differ for transient and delay faults:

- A transient fault appears as a short pulse (hazard) on a wire when it should keep its value stable. Therefore circuit malfunction is avoided if each one of these pulses is either eliminated (by changing the behavioral specification or the logic), or identified as not dangerous (because the involved signals actually do not switch concurrently due to some known timing properties of the circuit), or filtered by the circuit (if its fanout gates are not sensitive to its changes, or due to the inertial nature of gates). In this work, we will use sensitivity analysis to identify potentially dangerous transient faults, and logic (concurrency reduction) as well as timing methods to eliminate those that have been identified as dangerous.

- A delay fault happens when a signal is changing according to the specification, but the delay of its transition is affected by noise. In this case, it is impossible to filter the propagation of the transition, because it is part of the functional specification. Therefore, the fault can be either eliminated or identified as not dangerous (in the same way as in the case of transient faults).

The latter can be done by a *timing analysis* of the fault conditions, and is discussed in Section 6. In the rest of this section we will concentrate on *logic* methods for checking and avoiding noise. They can be naturally illustrated by the consideration of transient faults. The extension of the technique to delay faults is discussed later in Section 6.

**Transient fault propagation.** The formal conditions of transient fault propagation can be formulated in terms of *sensitization* [21].

Sensitization of a gate $g$ with respect to signal $b$ captures the conditions under which the value at a gate output depends on the value of $b$. Formally, when $g$ implements Boolean function $F$, its sensitization with respect to $b$ is: $Sens_F(b) = \frac{dF}{db}$.

The inverse of gate sensitization (observability don't care, ODC) gives conditions under which the value on a wire cannot affect a gate output. In particular, if a transient fault on wire $b$ occurs in a circuit state in which gate $g$ is not sensitized to $b$,

[4] As mentioned in Section 4, temporal separation can be reduced to logic separation by considering the SG obtained from an STG with added timing constraints.

then the fault cannot propagate through the gate. The idea of using gate sensitization for checking noise fault propagation in a circuit was first suggested in [12].

**Definition 5.2 (Digital sensitivity)** *Digital sensitivity of a gate $g$ to noise faults in a wire $b$ ($DS_g(b)$) is the conjunction of the characteristic function of the Noise fault set with the gate $b$ sensitization $DS(b) = S_n(b) \wedge Sens_g(b)$[5]. Digital sensitivity of a circuit to noise faults on a wire $b$ ($DS(b)$) is obtained by the union of the digital sensitivity conditions of all the gates in the fanout of $b$.*

If $DS(b)$ is empty, then the circuit is insensitive to faults occurring on a victim wire $b$. Hence all the aggressor-victim tuples that could generate these faults are actually *isolated from crosstalk noise*.

**Example 5.1** *For the example in Figure3.b, the sets of wires that may be the sources of crosstalk noise are: $Noise = \{(y,z),(x,z,y)\}$. The only state in which a transient fault may occur is $S_{tr}(z)$= {1*1*1}. Let us check, by only using the sensitivity conditions for wire $z$, whether the transient fault can propagate through the circuit.*

*The functions of the individual gates in this example are:*

$$x = \overline{z + \overline{xy}} = \overline{z}(x+y) \quad y = \overline{x+z} = \overline{x}\overline{z} \quad z = x + yz$$

*From them we can calculate the corresponding sensitivity functions:*

$$Sens_x(z) = x_z \oplus x_{\overline{z}} = 0 \oplus (x+y) = x+y$$
$$Sens_y(z) = y_z \oplus y_{\overline{z}} = 0 \oplus \overline{x} = \overline{x}$$
$$Sens_z(z) = z_z \oplus z_{\overline{z}} = (x+y) \oplus x = \overline{x}y$$
$$Sens(z) = Sens_x(z) + Sens_y(z) + Sens_z(z) = 1$$

*The circuit is sensitive to a fault on wire $z$ in every SG state. Hence the considered transient fault may indeed propagate through the circuit.*

*The results of propagation analysis hence* in this case *cannot reduce the $Noise$ set. We have to assume that both tuples $(y,z)$ and $(x,z,y)$ might be a source of noise faults.*

*There are two possibilities for solving the remaining noise faults after sensitivity (and timing) analysis:*

1. *to use legal behavior transformations, such as concurrency reduction, to avoid noise sources, as discussed in the rest of this paper, and*
2. *to use a constraint-driven layout tool, avoiding to put wires $z$ and $y$ adjacent to each other.*

## 6 Noise Analysis using STG and SG

### 6.1 Transient faults

When discussing digital sensitivity (Section 5), we made several simplifications that need to be justified.

[5]If the meaning is clear from the context, we will liberally identify the characteristic function of a set and the set itself.

**Simplification 1.** The suggested method treats transient faults *one at a time*, assuming that faults are not influencing each other.

We neglect the effects of fault interference, because the probability of two glitches arriving at the same time to the inputs of a gate seems to be rather small. This is similar to the assumptions that are traditionally made in testing, when considering single fault models versus multiple fault models [1].

**Simplification 2.** Sensitivity analysis in general assumes that sensitivity conditions are calculated when an input set is applied to the circuit, and then the circuit has enough time to stabilize. This is like the standard *fundamental mode* assumption. However, circuits synthesized from an STG operate in I/O mode, which means that inputs might change while part of the circuit is still reacting to a previous input set.

Therefore, we need to discuss the assumptions under which this "static" view is applicable to a circuit operating in I/O mode.

Let us start from an example, and consider the STG in Figure 5.a. One can derive the implementation of its signals $k$ and $e$ shown in Figure 5.c.

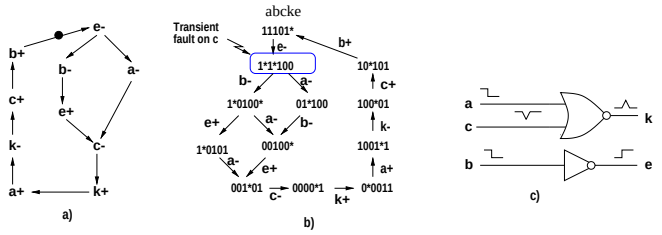


**Figure 5. A transient fault and its propagation**

In the corresponding SG (Figure 5.b) the only state which might be the cause of a transient fault is 1*1*100. This fault produces a pulse on victim wire $c$ and the propagation of this pulse could be checked by a sensitivity analysis of the NOR gate $k$ with respect to $c$: $Sens_k(c) = \overline{a} \Rightarrow DS_k(c) = S_{tr}(c) \cdot Sens_k(c) = abc\overline{k}\overline{e} \cdot \overline{a} = 0$

From $DS_k(c) = 0$ one can conclude that the transient fault on wire $c$ is not propagating through gate $k$ because it is blocked by the controlling value 1 on wire $a$. However, in its own turn signal $a$ is switching in state 1*1*100. Therefore, the propagation of a transient fault may depend on the relative propagation speed of the fault and some other transition (in this case $a\Leftrightarrow$ propagating to the inputs of gate $k$). If we denote by $wire\_delay(a,k)$ and $wire\_delay(c,k)$ the delays of the wires between gates $a$ and $k$ and $c$ and $k$ respectively, then the blocking of a transient fault on wire $c$ requires $wire\_delay(a,k) > wire\_delay(c,k)$[6]. This condition may be difficult to check, because of a number of reasons. It requires precise control over/analysis of wire delays, the inequalities could be inconsistent for different transient faults, etc. Therefore, our current sensitivity analysis might be *too optimistic* by not considering propagation delays.

[6]In fact, instead of delay $wire\_delay(c,k)$ we should consider the delay from the source of noise in wire $c$ to the input of gate $k$. However, since that point is not known until after layout, we conservatively use $wire\_delay(c,k)$.

The example in Figure 5 shows that signals which are enabled in a state that causes a transient fault are poor candidates for blocking the fault propagation. This leads to the notion of *static sensitivity* with respect to a state (the word "static" indicates that sensitivity conditions are stable at least for the considered state and its immediate successors).

**Definition 6.1 (Static sensitivity)** *Let an SG state $s$ be the source of a noise fault in wire $b$ and let $En(s)$ denote the set of signals enabled in $s$. The static sensitivity of gate $g$ to the noise fault in state $s$ ($SS_g(b,s)$) is given by the product of the characteristic function for $s$ and the existential abstraction of the sensitivity function with respect to signals in $En(s)$: $SS_g(b,s) = s \cdot \exists_{a \in En(s)} Sens_g(b)$.*

Static sensitivity assumes that checking for transient fault propagation is performed for each state separately. When state $s$ 1) is the source of a transient fault in a victim wire $b$ due to aggressors $a$ and $c$ and 2) $SS_g(b,s) \neq \emptyset$ the corresponding noise triple $(a,b,c)$ is considered to be dangerous.

**Example 6.1** *Consider the SG in Figure 5.b. In state 1*1*100 the set of enabled signals is $\{a,b\}$. $Sens_k(c) = \overline{a} \Rightarrow \exists_{x \in \{a,b\}} Sens_k(c) = 1$. Moreover, $SS_k(c, 1\text{*}1\text{*}100) = abc\overline{k}\overline{e} \cdot 1 \neq \emptyset$, the transient fault on c may propagate through gate $k$, and hence noise triple $\{a,c,b\}$ should be considered as dangerous.*

The idea behind the application of static sensitivity analysis instead of digital sensitivity (Definition 5.2) is to extend its applicability to *asynchronous sequential circuits*, by avoiding unrealistic (or difficult to implementation) timing assumptions about wire delays at the fanins of a gate. Let us consider Figure 6 and derive the timing assumptions behind static sensitivity. Suppose that in state $s$ a transient fault is generated at input $c$ of gate $Y$, and that the fault does not propagate through $Y$ according to the static sensitivity conditions.

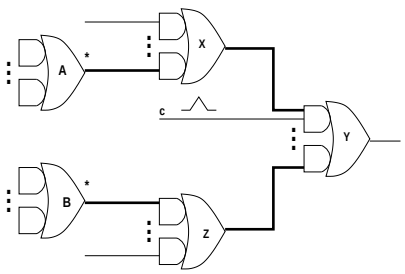


**Figure 6. Static sensitivity analysis**

The latter means that some the inputs of gate $Y$ ($fanin(Y)$) have logical values that block propagation of a pulse on $c$ to the output of $Y$. Moreover, due to Definition 6.1, these blocking signals are stable in state $s$. The blocking signals may, however, change their values when the gates in $fanin(Y)$ will make transitions (e.g., gates $X, Z, \ldots$). These transitions in their own turn should be caused by changes of signals in $fanin(X)$, $fanin(Z), \ldots$ (e.g., $A, B$).

Let us estimate the earliest time when blocking signals can change their values after the circuit starts from state $s$. In the worst case scenario (from the timing point of view) the signals causing transitions in $X, Z, \ldots$ are enabled in $s$ (otherwise, the blocking signals at $fanin(Y)$ will change even later). Then their propagation paths are shown in bold in Figure 6. If we denote by $min\_wire(fanin(V))$ the minimum delay among the wires in $fanin(V)$ and by $min\_gate(fanin(V))$ the minimum delay of gates in $fanin(V)$, then the minimum delay of these paths would be: $min_{V \in fanin(Y)}(min\_wire(fanin(V))) + min\_gate(fanin(Y)) + min\_wire(fanin(Y))$.

From here we can derive the worst case timing assumption that limits the applicability of static sensitivity:

$$min_{V \in fanin(Y)}(min\_wire(fanin(V))) + min\_gate(fanin(Y)) + min\_wire(fanin(Y)) > wire\_delay(c,Y) \quad (T1)$$

This means that a transient fault in wire $c$ will not propagate through gate $Y$ if the wire delay of $(c,Y)$ is less than *the time needed to change the value of the static sensitivity function*.

Notice that:

1. Timing assumption (T1) can always be satisfied in an implementation by delay padding of the gates in $fanin(Y)$. Delay padding of a gate cannot affect the validity of other timing assumptions (due to consideration of states different from $s$), because the right hand side (T1) contains only a wire delay, not a gate delay. This gives simple proof of convergence of the delay padding procedure. The requirement (T1) is very similar to the conditions of hazard free implementation from [17], where various delay padding algorithms are considered in detail.

2. However, (T1) is very difficult to satisfy by gate insertion or transistor sizing if wire delays dominate over gate delays. In this case, only relatively expensive constrained routing, with the goal of minimizing wire delay skew [27], can be used. Hence it becomes desirable to refine (T1) and make it less conservative, if possible.

This refinement of timing assumptions can be performed using the following procedure *Refine_assumptions*:

1. Start from state $s$ that is the cause of a transient fault on wire $c$ and such that $SS_g(c,s) = 0$.

2. Traverse the SG up to the states $s' \in S1$ in which $SS_g(c,s') \neq 0$.

3. For each $s' \in S1$ estimate the path delay for reaching $s'$ from $s$ ($del\_path(s,s')$) as the sum of wire and gate delays (for non-concurrent transitions).

4. Choose the minimal path delay: $del\_path\_min = min_{s' \in S1}(del\_path(s,s'))$

5. For each gate $y$ in the fanout of $c$ add a timing constraint $del\_path\_min < wire\_delay(c,y)$

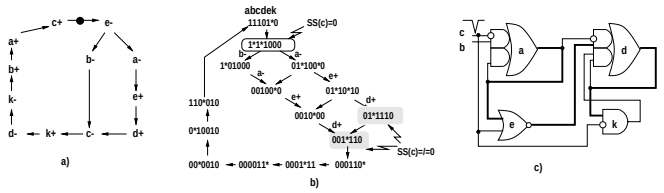


**Figure 7. Example of static sensitivity analysis**

**Example 6.2** *Let us perform the static sensitivity analysis for state 1*1*1000 in the SG in Figure 7.b, which causes a transient fault on wire c. Consider the propagation of this fault through gate k.*

$Sens_k(c) = d$. *Signal d is not enabled in 1*1*1000, therefore* $SS_k(c, 1*1*1000) = d \cdot abc\overline{d}\overline{e}\overline{k} = 0$.

*A conservative estimation, according to T1, of the timing assumptions under which the static sensitivity analysis is valid is:* $min\_wire(fanin(d)) + gate\_delay(d) + wire\_delay(d,k) > wire\_delay(c,k)$

*Let us apply procedure* Refine_assumptions *to get less conservative timing constraints. In the procedure, traversal of the SG from state 1*1*1000 will stop by reaching states 011*100 and 001*110, in which* $SS_k(c, 011*100)$ *and* $SS_k(c, 001*110)$ *have a non-zero value. These states are reached from 1*1*1000 by sequences of transitions* $\sigma_1 = a- \; e+ \; d+$ *and* $\sigma_2 = b- \; a- \; e+ \; d+$ *respectively. By leaving in* $\sigma_1, \sigma_2$ *only non-concurrent transitions, we may conclude that the static sensitivity will become non-zero only after the sequential firing of* $a\Leftrightarrow, e+$ *and* $d+$. *This sequence corresponds to the bold path in the circuit in Figure 7.c. By considering the delay of this path we thus arrive to a less conservative timing assumption, that is:* $wire\_delay(a,e) + gate\_delay(e) + wire\_delay(e,d) + gate\_delay(d) + wire\_delay(d,k) > wire\_delay(c,k)$.

The traversal in procedure *Refine_assumptions* can be pruned by additional timing assumptions. For example, in some applications we could assume that the circuit environment is *relatively slow*. If the environment reaction is slower than the skew among wire delays, then we can stop our traversal earlier, by forbidding the firing of inputs.

## 6.2 Delay faults

In the previous section we discussed the conditions under which transient faults are filtered by the circuit. They are based on static sensitivity analysis together with timing assumptions which justify the analysis. This technique can also be extended for the analysis of delay faults. We briefly sketch how to apply it.

Delay faults produce changes in the temporal behavior of some events. These changes may be dangerous if some timing constraints are assumed for the correct behavior of the circuit. Unfortunately, any circuit not in the class of delay-insensitive circuits is vulnerable to delay faults.

Given that speed-independent circuits are correct under any gate delay, delay faults may only affect wires with isochronic forks [28]. If a wire $v$ has a symmetric isochronic fork, a delay fault with $v*$ as a victim is always dangerous, no matter which branch of the fork it affects. In case of asymmetric isochronic forks, delay faults are only dangerous if they affect the fast branch to become slower (or vice versa – the slow branch to become faster).

A detailed analysis of different cases for delay faults is provided in [26]. Here we will give only a summary of it.

1. There are exceptional cases when delay faults can be neglected

   (E1) Delay fault happens at a wire which has no fork

   (E2) If a transition $v*$ in STG triggers only transitions of input signals then under a hypothesis of "slow environment" any delay fault on the wires originating at gate $v$ and feeding back to a circuit are safe. (The latter corresponds to an asymmetric isochronic fork by wire $v$, where its fast branch is a branch which goes to environment)

2. For checking the propagation of delay faults through branches of isochronic forks one can apply an analysis based on sensitization. (This analysis is similar to the case of transient faults with the only difference that in it we should consider a digital sensitization instead of the more complicated static sensitization.)

3. An analysis of delay faults propagation shows that under a conservative timing assumptions (similar to (T1)) *no delay fault* propagates through a circuit. Better (than (T1)) timing assumptions might be obtained via Procedure $Refine\_assumptions$.

Based on analysis of noise faults generation and propagation we can suggest two extreme approaches for checking a noise isolation.

- *Optimistic approach*

  We assume that any timing constraint which is derived from a sensitization analysis for noise faults might be satisfied in an implementation. Hence transient faults which are not sensitized and **all** delay faults are neglected. The remaining faults are reported to layout tool as dangerous ones.

- *Conservative approach*

  No timing assumptions are used for separating faults on observable and not. Under this conservative view all transient faults and all delay faults (excluding exceptional cases (E1) and (E2)) are assumed to be dangerous.

In Section 8 we show the results of noise isolation only for optimistic and conservative approaches. However in practice it might be reasonable to use an intermediate solution which lies between these two extreme cases. For example we could assume the existence of a small set of timing constraints (either given a priori or produced by timing analysis) and only part of faults is filtered due to these constraints.

## 7 Implementation issues

The methods proposed in this paper have been implemented in the tool **petrify**. For preliminary experiments, the methods have been applied only to the synthesis of speed-independent circuits.

Two different aspects have been considered in the implementation: analysis and synthesis.

**Analysis**

Analysis has been reduced to the calculation of those wires subject to noise faults, according to the algorithm presented in Figure 4. The analysis has been enhanced by [26]:

- calculating the static sensitization of wires to report only those transient faults that may be propagated (see definition 6.1) and
- by doing a conservative estimation of the wire forks of the circuit, according to the case study presented in section 6.2. The estimation is conservative in the sense that no timing assumptions are done on the delay of the gates.

**Synthesis**

A method for synthesis aiming at the reduction of noise faults has also been implemented. The method is based on behavioral transformations of the specification to reduce the concurrency events that may act as aggressors. The algorithm consists of the following steps:

1. Derive a complex-gate implementation of the circuit
2. Analyze the circuit, as explained above, and derive the set of noise faults.
3. Create a list of "noisy" pairs of events with one pair $(a*, b*)$ for each noise fault produced by $a* \parallel b*$.
4. Transform the specification aiming at reducing the concurrency of the noisy pairs of events.
5. Synthesize the circuit.

The algorithm has been integrated in the phase of concurrency reduction of `petrify` [7]. The cost function has been biased towards reducing the concurrency of those pairs of events that are "noisy".

## 8 Experimental results

We made experiments with a known set of asynchronous benchmarks. The results are presented in table 1.

The experiment was organized as follows.

- We implemented all the circuits by atomic complex gates, since the library is not crucial for the problem of noise avoidance.
- We used the crosstalk analysis algorithm to discover dangerous (noisy) concurrent states, isolated pairs and noisy wire configurations for both conservative and optimistic assumptions.
- We performed concurrency reduction for all dangerous concurrent states and used the crosstalk analysis algorithm (for almost all examples one reduction was enough).

The results of the first synthesis run (the circuit $CIR_{ini}$) are presented in columns 2-4 by showing the number of states $n_s$ in SG ($G$), the circuit area $Ar$ (literals in factored form), and the percentage of concurrent states $p_c$ ($p_c = \frac{|S_{con}|}{n_s}$, where $S_{con}$ is a set of concurrent states and $|SET|$ denotes the cardinality of set $SET$). The percentage of concurrent states is taken as a relative measure of performance, since concurrency is an important source of performance gains for asynchronous circuits[7].

The results of the application of the crosstalk analysis algorithm to this circuit are presented in column 5 - the digital isolation percentage - $isol$.

The results of the application of the concurrency reduction algorithm under conservative delay assumption are presented in columns 6-8 ("without sensitivity analysis"). The results of the application of the concurrency reduction algorithm under optimistic assumption ("with timing assumption") are presented in columns 9-12. From these *preliminary* experiments we can conclude that crosstalk noise avoidance using behavioral specifications is a very promising technique for asynchronous control circuits, because:

- it can effectively extract all isolated and noisy wire configurations from the behavioral specification, and thus supply a set of constraints that can completely avoid crosstalk noise to the layout tools.
- it simplifies the task of constrained layout generation by using noise avoidance techniques at behavioral and logic level.

These experiments showed that about 80% of noise isolation can be reached in average by behavioral transformations. Previous work [7] has confirmed that restricted behavioral transformations (concurrency reduction e.g.) allow the designer to save up to 20% of area under a small degradation of performance (below 5%). This gives a ground that, in the suggested methodology targeted for "low noise", the quality of implementation is not significantly degraded. In fact, the area of the circuits after noise avoidance has decreased in all our benchmarks.

## 9 Conclusion

[7] This is rather coarse measure of performance to get a qualitative picture only. The better way for performance estimation under the concurrency reduction was given in [7]

| name | noisy synthesis | | | | without sensitivity analysis | | | | with timing assumptions | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n_s$ | $Ar$ | $p_c$ % | $isol$ % | $n_s$ | $Ar$ | $p_c$ % | $isol$ % | $n_s$ | $Ar$ | $p_c$ % | $isol$ % |
| chu133 | 24 | 13 | 42 | 24 | 14 | 12 | 0 | 100 | 19 | 13 | 26 | 100 |
| chu150 | 26 | 11 | 46 | 47 | 16 | 9 | 12 | 93 | 18 | 11 | 22 | 100 |
| master-read | 2844 | 41 | 99 | 0 | 52 | 22 | 46 | 30 | 96 | 32 | 71 | 31 |
| mmu | 370 | 32 | 94 | 2 | 24 | 19 | 8 | 96 | 44 | 19 | 50 | 100 |
| mp-forward-pkt | 22 | 15 | 27 | 75 | 20 | 15 | 20 | 75 | 22 | 15 | 27 | 86 |
| mr1 | 336 | 35 | 92 | 3 | 26 | 25 | 0 | 100 | 33 | 28 | 21 | 100 |
| nak-pa | 58 | 18 | 66 | 38 | 20 | 12 | 0 | 100 | 20 | 12 | 0 | 100 |
| nowick | 20 | 13 | 20 | 20 | 19 | 12 | 16 | 40 | 19 | 12 | 16 | 47 |
| ram-read-sbuf | 39 | 22 | 44 | 33 | 24 | 16 | 8 | 96 | 29 | 17 | 24 | 100 |
| sbuf-ram-write | 64 | 21 | 62 | 9 | 33 | 21 | 27 | 53 | 33 | 20 | 27 | 53 |
| sbuf-send-ctl | 18 | 15 | 11 | 57 | 16 | 12 | 0 | 100 | 16 | 12 | 0 | 100 |
| trimos-send | 336 | 30 | 95 | 0 | 30 | 17 | 40 | 36 | 37 | 23 | 51 | 36 |
| vbe5b | 24 | 11 | 50 | 40 | 12 | 3 | 0 | 100 | 14 | 6 | 14 | 100 |
| vbe6a | 192 | 34 | 90 | 0 | 20 | 18 | 0 | 100 | 22 | 24 | 9 | 100 |
| vbe10b | 256 | 39 | 91 | 0 | 22 | 19 | 0 | 100 | 27 | 30 | 19 | 100 |
| wrdtab | 216 | 35 | 89 | 0 | 39 | 18 | 38 | 7 | 60 | 22 | 60 | 7 |
| sbuf-read-ctl | 19 | 15 | 16 | 89 | 17 | 13 | 6 | 96 | 19 | 15 | 16 | 100 |
| mr0 | 584 | 54 | 96 | 0 | 28 | 23 | 0 | 100 | 44 | 33 | 36 | 100 |
| pe-send-ifc | 111 | 42 | 52 | 9 | 59 | 42 | 8 | 53 | 63 | 48 | 14 | 53 |
| converta | 18 | 16 | 22 | 50 | 14 | 12 | 0 | 100 | 16 | 15 | 12 | 100 |
| ebergen | 18 | 12 | 22 | 40 | 14 | 12 | 0 | 100 | 14 | 12 | 0 | 100 |
| half | 14 | 9 | 43 | 17 | 8 | 2 | 0 | 100 | 12 | 6 | 33 | 100 |
| hazard | 12 | 8 | 17 | 50 | 12 | 8 | 17 | 50 | 12 | 8 | 17 | 67 |
| seq4 | 28 | 25 | 29 | 49 | 20 | 14 | 0 | 100 | 23 | 21 | 13 | 100 |
| muller5 | 64 | 21 | 81 | 0 | 16 | 8 | 25 | 73 | 20 | 9 | 40 | 100 |
| muller10 | 2048 | 46 | 99 | 0 | 31 | 13 | 29 | 84 | 40 | 14 | 45 | 100 |
| par4 | 1307 | 32 | 98 | 5 | 28 | 25 | 0 | 100 | 30 | 27 | 7 | 100 |
| vme-read | 147 | 21 | 83 | 2 | 31 | 16 | 19 | 56 | 34 | 19 | 26 | 62 |
| vme-write | 265 | 23 | 90 | 2 | 38 | 16 | 29 | 48 | 47 | 17 | 43 | 52 |
| total | 9273 | 709 | 61 | 23 | 703 | 454 | 12 | 79 | 883 | 540 | 25 | 83 |

**Table 1. Experimental results of crosstalk noise avoidance using behavioral specifications**

This paper has only begun the investigation of noise avoidance techniques in asynchronous control circuit design. In particular, we consider the following topics to be very promising areas for future research.

- Incorporate the noise avoidance technique into the synthesis process from the very beginning. For example, one could use better the don't care information for the fanout of the victim wire to increase its isolation. Also, concurrency reduction could be integrated with timing constraints.

- Noise avoidance using behavioral specifications could be better incorporated into a complete design flow. At higher levels of abstraction, it could be very useful to integrate it with clustering in hierarchical synthesis ([24, 2]). At lower levels, direct interaction with some constrained layout tools like [3, 11] or [31] could help both the layout tool and the logic synthesis tool by exchanging information on the relative ease of solving some noise cases by layout or logic.

- Most steps of our algorithms (except for the current version of logic synthesis) could be implemented using only STG-based methods (STG unfolding analysis [16, 15]), that are often faster than SG-based methods and cope better with the state explosion problem. These techniques, together with the hierarchical approaches mentioned above, would allow us to tackle realistic design problems.

[30] pointed out that the digital sensitivity approach has also some problems:

- In case the victim wire v is coupled to many aggressors, a capacity charge sharing model must be used to find the amplitude of the noise pulse. The corresponding physical effect can only be conservatively broken down into pairs of potential noise contributions.

- The problem of wire ordering to minimize total noise is NP-complete.

  Our approach, that considers also *sequential* information helps solving both problems, by reducing the number of pairs that must be considered,

## Acknowledgments

This work has been partially funded by CICYT TIC 98-0410 and ACiD-WG (ESPRIT 21949). We are grateful to Michael Kishinevsky from Intel (Strategic CAD Labs) for many useful discussions and critics. We thank anonymous reviewers for useful comments on the draft of this paper.

## References


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

[2] M. Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, (3):80–87, 1998.

[3] U. Choudhury and A. L. Sangiovanni-Vincentelli. Constraint-based channel routing for analog and mixed analog/digital circuits. In *Proceedings of the International Conference on Computer-Aided Design*, 1990.

[4] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[5] J. Cong. Deep submicron layout and coupling to logic synthesis (invited talk). In *International Workshop on Logic and Architecture Synthesis (IWLAS'97)*, 1997.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. In *IEEE/ACM Int. Conference on Computer Aided Design*, pages 220–227, San Jose, USA, Nov. 1997.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic synthesis and optimization of partially specified asynchronous systems. In *Proceedings of the Design Automation Conference*, 1999. (accepted).

[8] F. Dartu and L. T. Pileggi. Calculating worst-case gate delays due to dominant capacitance coupling. In *Proceedings of the Design Automation Conference*, June 1997.

[9] L. Gal. On-chip cross talk – the new signal integrity challenge. In *IEEE 1995 Custom Integrated Circuits Conference*, pages 251–254, 1995.

[10] R. Ginosar and S. Rotem. 1997 sia technology roadmap. implications to asynchronous design. In *ACiD-WG Workshop on Specification models and languages and Technology effects on asynchronous design*, 1998.

[11] D. A. Kirkpatrick and A. L. Sangiovanni-Vincentelli. Techniques for crosstalk avoidance in the physical design of high-performance digital systems. In *Proceedings of the International Conference on Computer-Aided Design*, 1994.

[12] D. A. Kirkpatrick and A. L. Sangiovanni-Vincentelli. Digital sensitivity: Predicting signal interaction using functional analysis. In *Proceedings of the International Conference on Computer-Aided Design*, 1996.

[13] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-Timed Design*. John Wiley and Sons Ltd., 1994.

[14] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 324–331, November 1998.

[15] A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, and L. Lavagno. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems, and Computers*, 8(1):67–118, 1998.

[16] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, 1998.

[17] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[18] T.-C. Lee and J. Cong. The new line in ic design. *IEEE Spectrum*, (3):52–58, 1997.

[19] W. A. Lien, P. Day, C. Faransworth, D. L. Jackson, J. Liu, and N. Paver. Noise in a self-timed and synchronous implementation of a dsp. Unpublished, White Paper, Cogency Technology Inc., http://www.cogency.com/noise.html, 1997.

[20] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[21] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[22] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[23] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[24] K. L. Shepard, V. Narayanan, P. C. Elmendorf, and G. Zheng. Global harmony: Coupled noise analysis for full-chip rc interconnect networks. In *Proceedings of the International Conference on Computer-Aided Design*, 1997.

[25] D. Sylvester, C. Hu, O. S. Nakagawa, and S.-Y. Oh. Interconnect scaling: Signal integrity and performance in future high-speed CMOS designs. In *Proc. of VLSI Symposium on Technology*, pages 42–43, 1998.

[26] A. Taubin, A. Kondratyev, J. Cortadella, and L. Lavagno. Noise avoidance methods in asynchronous design using behavior specifications. Technical Report 98-2-005, University of Aizu, Japan, Oct. 1998.

[27] R.-S. Tsay. An exact zero-skew clock routing algorithm. *IEEE Transactions on Computer-Aided Design*, 12:242–249, 1993.

[28] K. van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[29] P. Vanbekbergen, G. Goossens, and B. Lin. Modeling and synthesis of timed asynchronous circuits. In *Proceedings of the European Design Automation Conference (EURO-DAC)*, pages 460–465, Sept. 1994.

[30] A. Vittal and M. Marek-Sadowska. Crosstalk reduction for VLSI. *IEEE Transactions on Computer-Aided Design*, (3):290–298, 1997.

[31] T. Xue, E. Kuh, and D. Wang. Post global routing crosstalk synthesis. *IEEE Transactions on Computer-Aided Design*, (12):1418–1430, Dec. 1997.

[32] G. Yee, R. Chandra, V. Ganesan, and C. Sechen. Wire delay in the presence of crosstalk. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, Dec. 1997.