

Fault Detection and Isolation Techniques for Quasi Delay-Insensitive Circuits

Christopher LaFrieda and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca NY 14853, U.S.A.

Abstract

This paper presents a novel circuit fault detection and isolation technique for quasi delay-insensitive asynchronous circuits. We achieve fault isolation by a combination of physical layout and circuit techniques. The asynchronous nature of quasi delay-insensitive circuits combined with layout techniques makes the design tolerant to delay faults. Circuit techniques are used to make sections of the design robust to non-delay faults. The combination of these is an asynchronous defect-tolerant circuit where a large class of faults are tolerated, and the remaining faults can be both detected easily and isolated to a small region of the design.

1. Introduction

Quasi delay-insensitive (QDI) circuits are asynchronous circuits that operate correctly regardless of gate delays in the system. These circuits do not use any clocks, and the function of the clock is replaced by handshaking signals on wires. Instead of using time to implement sequencing (via the clock), QDI circuits use the notion of causality and event-ordering.

QDI systems exhibit many advantages over clocked design. QDI circuits can be used to design complex, highly concurrent systems that are low-power and exhibit average-case behavior [11]. The very nature of QDI circuits—namely, that they are insensitive to gate delays—makes them well-suited for fault-tolerant design, because most delay faults do not cause the circuit to malfunction. However, while there is an enormous body of literature that examines different types of faults for clocked circuits, little attention has been paid to the problem of faults in asynchronous VLSI systems. This paper provides an in-depth examination of faults in the context of QDI circuits, and proposes several methods for fault detection and isolation.

The absence of a global clock means that a faulty asynchronous circuit might exhibit problems that would not nor-

mally arise in a clocked system. In some sense, faults in control signals in asynchronous logic would be analogous to faults in data as well as clock lines in a clocked system. Faulty asynchronous circuits can behave quite differently than faulty synchronous circuits. Asynchronous circuits consist of many state-holding nodes (dynamic nodes with keepers) and handshaking signals. A fault in an asynchronous system can cause computations to occur out-of-sequence; this could violate an event-ordering constraint required for correctness, leading to a circuit failure. A fault might prevent some computation from occurring by preventing a signal transition and causing deadlock.

Fault detection in QDI circuits requires new techniques than those used in synchronous architectures. To illustrate this, consider the concept of introducing redundancy in the system by using error detecting/correcting codes. These codes contain data bits and check bits which are periodically compared using hardware known as checkers. If the data is consistent with the check bits then it is assumed that no error exists. If the check bits are inconsistent with the data, then an error is detected and possibly corrected [5]. Unfortunately, this approach might not work well with asynchronous circuits as many faults can cause deadlock, preventing some data/check bits from appearing on the output. In a clocked system where we assume the clock always works, one can always sample the output of the circuit at the clock edge and use those values. In contrast to this, asynchronous signaling protocols embed data validity in the signal values themselves, leading to invalid state faults or simply deadlock. (Note that a similar problem could arise in a clocked system when part of the clock network has a fault.)

In this paper we present a set of techniques for both detecting and isolating faults in asynchronous QDI circuits. We begin with an overview of both our gate model, QDI circuits, and fault model (Section 2). We examine the effect of the faults on the behavior of asynchronous circuits, and determine the types of failures that can result from faults (Section 3). We introduce techniques that improve the fault detection capability of QDI circuits (Section 4), and examine the implementation of these techniques in an asynchronous

pipeline (Section 5). Finally, we provide an overall summary of the results (Section 6).

Related Work. Previous analysis of faulty/mis-behaving asynchronous circuits has been done using the stuck-at-0/stuck-at-1 fault model [3]. This work examines the effects of stuck-at faults in delay-insensitive, quasi delay-insensitive, and speed independent circuits. Testing QDI circuits, using the stuck-at model, is thoroughly explored in [2]. This testing method classifies a fault as either inhibiting (preventing an action) or stimulating (causing an action), identifies faults that can't be observed, and describes a technique to make all faults observable by adding testing points. A technique to mask transient faults that occur in asynchronous, speed independent, interfaces is described in [14]. This technique employs the use an adjudicator to mask transient faults between a circuit and the environment.

Our work examines a robust class of faults, including process and reliability, and examines the effect of transient faults. Perhaps the most important distinguishing feature of our approach v/s the previous work is that our fault detection techniques can be applied at any granularity—at a single bit level, at the function block level, at the pipeline stage level, etc. depending on the granularity of fault isolation required. However, previous asynchronous fault detection techniques focused on making stuck-at faults externally visible without attempting to isolate them, and the granularity of fault detection could not be controlled.

2. Sources of Faults and Fault Modeling

Faults can occur at any point in an integrated circuit's lifetime. Faults that occur during fabrication are known as process faults. Process faults can directly cause failures or can result in devices with a short lifespan. Faults that cause devices to fail early in their lifetime are known as reliability faults. Since reliability faults behave like delay faults before they fail and like process faults after they fail, we won't consider them directly. Throughout this paper, we make the following distinction between faults and failures. Faults are the physical (electrical) mechanism that may cause a circuit to malfunction. Failures are the actual deviant behaviors that result from faults. For example, let's say a process fault converts a static node into a dynamic one. Such a circuit may cause transient failures because it fails at random time intervals depending on noise, however, the fault itself always exists. Soft-errors such as EM noise, crosstalk, and alpha particle radiation will have a similar effect.

In this section we discuss asynchronous QDI circuits, provide a description of the circuit notation, and explain some of their key properties that make them suitable for fault detection. We continue by discussing sources of faults,

and the way we model the effects of these faults on a QDI circuit in terms of a transformation on the circuit itself.

2.1. QDI Circuits

QDI circuits are implemented as a network of gates, where each gate consists of a pull-up network implemented with p-transistors, and a pull-down network implemented with n-transistors. Logically, we can think of a gate as corresponding to two Boolean predicates: G^+ , the condition that causes its output ν to be connected to the power supply (V_{DD} , interpreted as the logic "true" or 1 value in any Boolean expression), and G^- , the condition that causes its output ν to be connected to ground (GND , interpreted as the logic "false" or 0 value in any Boolean expression). We denote this gate using the *production rule* (PRS) notation [9] as follows:

$$\begin{aligned} G^+ &\mapsto \nu\uparrow \\ G^- &\mapsto \nu\downarrow \end{aligned}$$

Using this notation, a two-input NAND gate would be specified as follows:

$$\begin{aligned} \neg a \vee \neg b &\mapsto out\uparrow \\ a \wedge b &\mapsto out\downarrow \end{aligned}$$

where " \wedge " denotes the Boolean AND, " \vee " denotes OR, and " \neg " denotes logical negation. A restriction on production rules is that both G^+ and G^- must never be true at the same time, because this would result in a short-circuit. This condition is known as non-interference.

If G^+ and G^- are complements of each other, then the gate output is always connected to a power supply. This corresponds to a conventional static CMOS gate and is referred to as a combinational gate. If there is a state when both G^+ and G^- are false, then in this state the output does not change. If this occurs, then the gate is said to be *state-holding*. State-holding gates always contain a staticizer (a.k.a. a keeper) on their output to prevent the gate output from changing due to leakage or noise.

A fork in a circuit corresponds to an output of a gate being used as the input to more than one gate. Each connection from a gate output to a gate input is referred to as a branch of the fork. We say that a branch of the fork is *isochronic* if we must make a delay assumption about the relative delay of the branch of the fork relative to the other branches of the same fork (a detailed technical discussion can be found in [10, 7]).

A circuit is said to be QDI if it operates correctly regardless of the delays of the gates or wires that implement it, with the exception of wires that implement isochronic branches [10, 7]. It has been established that for an asynchronous circuit to be hazard-free under the QDI model, every signal transition must be acknowledged by a transition

on the output of some gate it is connected to [10]. In particular, this means that a signal cannot make a $0 \rightarrow 1 \rightarrow 0$ transition without an intervening transition on the output of some gate that it is connected to. This condition can be translated into a semantic check on the production rules [7], and this check can be used to determine if a circuit is QDI.

A commonly occurring gate in QDI circuits is the two-input C-element. A C-element is state-holding, and is described by the following production rules:

$$\begin{aligned} a \wedge b &\mapsto c \uparrow \\ \neg a \wedge \neg b &\mapsto c \downarrow \end{aligned}$$

A C-element could also be inverting, in which case the $c \uparrow$ and $c \downarrow$ transitions are interchanged. This can be generalized to an n -input C-element by and-ing additional terms to both the pull-up and pull-down.

This acknowledgment property translates to the following result [2]: a stuck-at-0/stuck-at-1 fault on the output of any gate in a QDI circuits results in deadlock. This is intuitively obvious from the description of acknowledgment described above. However, other faults can cause more subtle errors in QDI circuits. We discuss these situations in the next section, providing production-rule models for each category of fault.

There is a well-established synthesis method for QDI asynchronous circuits [9]. Production rules can be generated that guarantee both non-interference and hazard-free behavior by “compiling” a description of the asynchronous computation expressed in a programming notation. The notation, called “handshaking expansions,” (HSE) describes the sequence of waits and actions that must be performed by the asynchronous circuit. For instance, the sequence $*[L \uparrow; [\neg L^e]; L \downarrow; [L^e]]$ can be read as follows: repeat the following sequence forever (the outer $*[..]$): set L high; then wait for L^e to be low; then set L low; then wait for L^e to be high. The “;” here denotes sequential actions, while a “,” would denote parallel actions. This sequence describes a four-phase handshake protocol on a pair of wires L and L^e .

2.2. Process Faults

Process faults are those faults which occur during fabrication. Process faults can be either global or local in nature. Global disturbances, such as mask misalignment, will more than likely damage an entire wafer. Local disturbances, however, usually only causes damage to a small number of devices. Local disturbances result from contaminants introduced during the various process steps. Contaminants will result in extra or missing material depending on the contaminant size, location, and the processing step in which it’s introduced [4].

Figure 1 shows a few examples of process faults. Fault (a) is a short in the metal one layer caused by a contaminant

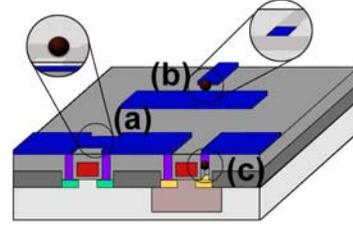


Figure 1. Process faults occurring in a cmos n-well process: (a) is a metal one short, (b) is a metal one open, and (c) is an open via.

that was introduced before metal layer one was etched, but after photoresist was applied. The contaminant prevented a section of photoresist from being exposed, which protected the underlying metal from being etched. Fault (b) is an open in metal one that is caused by the presence of a contaminant before metal one deposition. This contaminant creates a raise in the metal one material that is deposited on top of it. When resist is spun on, it fails to cover this raised portion and more metal one is etched than should be. Fault (c) is an open in a diffusion contact via. A contaminant blocked, or partially blocked, the via and prevented deposited metal from making contact with the diffusion.

The most common model used for process faults is the stuck-at fault model [1]. The stuck-at model is attractive because it considers faults at the gate level, rather than the transistor level, which makes test pattern generation easy. However, the stuck-at fault model doesn’t model bridging faults, open faults or transistor level faults well. Table 1 shows the resulting production rules for some stuck-at faults. Since many gates in asynchronous circuits are C-elements, it is common for any stuck input to result in a stuck output.

Fault Class	Original PRS	Resulting PRS
Output Stuck-at-0	$G^+ \mapsto \nu \uparrow$ $G^- \mapsto \nu \downarrow$	$\neg V_{DD} \mapsto \nu \uparrow$ $V_{DD} \mapsto \nu \downarrow$
Output Stuck-at-1	$G^+ \mapsto \nu \uparrow$ $G^- \mapsto \nu \downarrow$	$\neg GND \mapsto \nu \uparrow$ $GND \mapsto \nu \downarrow$
Input Stuck-at-0	$\neg x \wedge G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$	$G^+ \mapsto \nu \uparrow$ $GND \mapsto \nu \downarrow$
Input Stuck-at-1	$\neg x \wedge G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$	$\neg V_{DD} \mapsto \nu \uparrow$ $G^- \mapsto \nu \downarrow$

Table 1. Mapping of stuck-at process faults to resulting PRS.

Table 2 contains some classes of faults that might not be detected using the stuck-at fault model. Bridging faults may result in production rules that are interfering (their pull-up network and pull-down network are both active). The result-

ing voltage level will depend on the driving strength of each network and the resistance of the bridge itself. When the pull-up or pull-down networks are active exclusively, this node will be logically high or low respectively. The stuck-on fault may cause interference if G^+ and G^- can be simultaneously true. Without having to wait for $\neg x$, this production rule may also result in a premature firing of ν . The first PRS for the stuck-open fault is essentially the same as the last stuck-at example in Table 1. The second PRS, however, may have turned a non-state-holding node into a state-holding node.

Fault Class	Original PRS	Resulting PRS
Bridging ($\nu_0 \leftrightarrow \nu_1$)	$G_0^+ \mapsto \nu_0 \uparrow$ $G_0^- \mapsto \nu_0 \downarrow$ $G_1^+ \mapsto \nu_1 \uparrow$ $G_1^- \mapsto \nu_1 \downarrow$	$G_0^+ \vee G_1^+ \mapsto \nu_0, \nu_1 \uparrow$ $G_0^- \vee G_1^- \mapsto \nu_0, \nu_1 \downarrow$
Stuck-On ($\neg x$)	$\neg x \wedge G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$	$G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$
Stuck-Open ($\neg x$)	$\neg x \wedge G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$	$\neg V_{DD} \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$
Stuck-Open ($\neg x$)	$\neg x \vee G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$	$G^+ \mapsto \nu \uparrow$ $x \wedge G^- \mapsto \nu \downarrow$
State-Holding ($G^+ \neq \overline{G^-}$)	$G^+ \mapsto \nu \uparrow$ $G^- \mapsto \nu \downarrow$	$G^+ \vee$ $(\overline{G^+} \wedge \overline{G^-} \wedge \neg \zeta(t)) \mapsto \nu \uparrow$ $G^- \vee$ $(\overline{G^-} \wedge \overline{G^+} \wedge \zeta(t)) \mapsto \nu \downarrow$

Table 2. Mapping of non-stuck-at process faults to resulting PRS.

State-holding faults occur for two reasons. One, a non-state-holding node becomes state-holding in the presence of a fault. Two, a fault occurs in the feedback circuit that is maintaining the charge on a state-holding node. It is difficult to predict the exact behavior of a state holding node since it will be particularly sensitive to noise. We will make no assumptions about the time it takes for such a node to dissipate charge. Instead, we assume that when this node is state-holding it may be driven high or low by some arbitrary unit function $\zeta(t)$.

2.3. Transient Faults

Transient faults are those that might occur at some time t , but are not stable in the sense they might not occur at other times. Two major causes of transient faults are crosstalk and radiation. Crosstalk is a mechanism by which switching wires (aggressors) can induce a voltage on other wires (victims). Crosstalk is the result of both capacitive coupling and inductive coupling. Although crosstalk is somewhat of a design issue, newer technologies have upwards of six metal layers and it can be difficult to guarantee that crosstalk faults won't occur. In-

ductive coupling is particularly challenging since it decays logarithmically with wire spacing [12], rather than linearly like capacitive coupling does. Transient faults due to radiation occur when a particle strikes some region of a device and creates a track of electron-hole pairs. If these pairs collect at a p-n junction, then there will be a resulting current pulse.

When considering transient faults, without knowing the exact geometry of a circuit, we have to consider the possibility that any node in our circuit can be affected. Every production rule in the system will be of the form:

$$G_n^+ \vee \zeta_{n+}(t) \mapsto \nu_n \uparrow$$

$$G_n^- \vee \zeta_{n-}(t) \mapsto \nu_n \downarrow$$

2.4. Delay Faults and Isochronic Branches

If a fault causes a circuit to exceed the circuit's timing specifications, but doesn't affect the circuit's logical function, it is said to be a delay fault. Delay faults can occur due to the aforementioned sources. Some examples of sources of delay faults are partial shorts, partial opens, and induced currents in the opposite direction of switching.

In QDI circuits, delay faults will only cause a logical fault if they occur on an isochronic branch, since it is the only place timing assumptions are permissible. All other locations of delay faults will only change the performance of the circuit, but not affect its correctness.

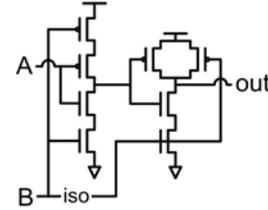


Figure 2. An example of an isochronic branch (state holding element not shown).

An isochronic branch occurs when a wire forks to multiple gates, and at least one of those gates doesn't acknowledge a transition on that wire. An example of an isochronic branch is shown in Figure 2. Initially, both A and B are high, then A goes low. Since one input of the NAND gate is already low, when B goes low there is no change in out (it's still high). The problem occurs when there is a large delay in the isochronic branch (labeled iso). When B goes low, the output of the C-element (the left circuit) goes high. If signal B at the input of the NAND gate is still high, then its output will go low when it shouldn't have. We will discuss a method to avoid these faults in Section 4.

3. Failures in QDI Circuits

QDI systems are collections of communicating hardware components known as processes [9]. A hardware process communicates with other hardware processes via synchronization and data channels [9]. Our approach to failure detection focuses on the impact of faults on the behavior of channels in the asynchronous system. This approach has several advantages: (i) focusing on channels allows us to ignore the problems that might occur in complex production rules internal to a process; (ii) channels that operate correctly have a well-defined behavior that is consistent throughout the asynchronous design; (iii) there is a class of *slack elastic* systems whose correct operation only depends on correct channel behavior, and such systems encompass entire microprocessors [8]; (iv) channels occur at the bit-level granularity, as well as the function block or pipeline granularity in high-performance asynchronous systems [11]; (v) transient faults can also be treated as errors on channels, thereby leading to a uniform treatment of the different fault categories. For simplicity, we will assume that channels use a standard four-phase return-to-zero handshake protocol, and that data is encoded using dual-rail codes.

Data values and/or synchronization actions that transfer control and/or data from one hardware process to another are referred to as tokens. A token is a data item that propagates through a pipeline, and that can be passed from one process to another via a communication channel.

3.1. Deadlock

Most faults, especially stuck-at faults, will cause asynchronous circuits to deadlock [3][2]. Whenever a fault inhibits a transition on a handshaking wire, then deadlock will occur. Consider the half buffer circuit in Figure 3, where

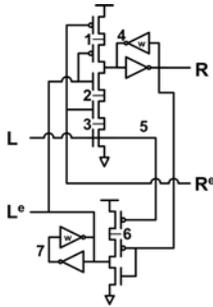


Figure 3. Precharge half buffer circuit (PCHB).

shake protocols on the pairs (L, L^e) and (R, R^e) . In terms of handshaking expansion notation, the operation of this circuit can be described as the following sequence [6]:

$$* [[R^e \wedge L]; R\uparrow; L^e\downarrow; [\neg R^e]; R\downarrow; [\neg L]; L^e\uparrow]$$

Such circuits are the basis for highly pipelined asynchronous QDI designs [6], so examining this circuit is instructive. A stuck-at fault on L , L^e , R , or R^e will halt the buffer and the surrounding environment. If an open fault occurs on node 1, then node 4 will be inhibited from making an up-transition and R will be stuck-at-1. An open fault on nodes 2 or 3 will cause R to be stuck-at-0 and similarly, an open fault at node 5 or 6 will cause L^e to be stuck-at-0. We can assume that open faults on these nodes will result in stuck-at faults because their outputs are sticized (their charge is held by the weak transistors in the inverter labeled “w”).

We can determine the resulting values of each synchronization channel, when a particular fault causes deadlock, by examining the HSE (handshaking expansion) of the process. Annotating the states of this buffer process with the values of signals $\{L, L^e, R, R^e\}$, we have:

$$\begin{aligned} & * [\{x, 1, 0, x\} [R^e \wedge L]; \{1, 1, 0, 1\} R\uparrow; \\ & \quad \{1, 1, 1, x\} L^e\downarrow; \{x, 0, 1, x\} [\neg R^e]; \\ & \quad \{x, 0, 1, 0\} R\downarrow; \{x, 0, 0, x\} [\neg L]; \\ & \quad \{0, 0, 0, x\} L^e\uparrow] \end{aligned}$$

The states in which the process will halt can be determined by starting from the beginning of the HSE and stepping through to the furthest state that can be reached when a transition is inhibited. Assuming that the process is examined at $t=\infty$, all the variables will be stable. The values of the channels for each stuck-at fault are shown in Table 3. When there is a stuck-at-1 fault on R , it can halt in two different states. The second state in which R stuck-at-1 halts can occur after reset, since we can’t make an assumption on how long it takes the environment to perform $R^e\uparrow$.

Variable	Stuck-At-0	Stuck-At-1
L	$\{0,1,0,1\}$	$\{1,0,0,1\}$
L^e	$\{0,0,0,1\}$	$\{1,1,1,0\}$
R	$\{1,1,0,1\}$	$\{1,1,1,0\}$ $\{0,0,1,0\}$
R^e	$\{1,1,0,0\}$	$\{0,0,1,1\}$

Table 3. Resulting states, $\{L, L^e, R, R^e\}$, when a PCHB deadlocks.

3.2. Synchronization Failure

If a handshake on a synchronization channel begins or ends prematurely, then the process and its environment ex-

the environment communicates with the circuit using hand-

perience synchronization failure. When a process is receiving a synchronization signal, if $L^e \downarrow$ fires before a handshake is complete, then the environment may stop sending the signal before the process has received it. If $R \uparrow$ fires when the process has not received a synchronization signal, then the process may send a synchronization signal when it shouldn't have.

Consider, once again, the PCHB of Figure 3. An invalid synchronization signal may be sent if the production rule for $R \uparrow$ fires in any of the following states (which correspond to the labels in the previous HSE):

$$* [\textcircled{1} [R^e \wedge L]; R \uparrow; L^e \downarrow; \neg R^e]; R \downarrow; \textcircled{2} [\neg L]; \textcircled{3} L^e \uparrow]$$

The production rules for $R \uparrow$ are:

$$\begin{aligned} \neg R^e \wedge \neg L^e &\mapsto \overline{R} \uparrow \\ R^e \wedge L^e \wedge L &\mapsto \overline{R} \downarrow \\ \neg \overline{R} &\mapsto R \uparrow \\ \overline{R} &\mapsto R \downarrow \end{aligned}$$

The resulting PRS for faults that cause a premature firing of $R \uparrow$ are shown in Table 4. Any variable that can transition in states $\textcircled{1}$, $\textcircled{2}$, or $\textcircled{3}$ may cause $R \uparrow$ to fire if it's bridged to R and makes an upward transition or bridged to \overline{R} and makes a downward transition. If the nmos transis-

Fault	State	Resulting PRS
Bridging($L^e \leftrightarrow R$)	$\textcircled{2}$	$G_{L^e}^+ \vee \neg \overline{R} \mapsto R \uparrow$
Bridging($R^e \leftrightarrow R$)	$\textcircled{1}, \textcircled{2}, \textcircled{3}$	$G_{R^e}^+ \vee \neg \overline{R} \mapsto R \uparrow$
Stuck-On(L)	$\textcircled{1}$	$R^e \wedge L^e \mapsto \overline{R} \downarrow$
State-Holding	$\textcircled{1}$	$(R^e \wedge L^e \wedge L) \vee (R^e \wedge L^e \wedge \overline{L} \wedge \zeta(t)) \mapsto \overline{R} \downarrow$
Transient	$\textcircled{1}, \textcircled{2}, \textcircled{3}$	$(R^e \wedge L^e \wedge L) \vee \zeta_{\overline{R}-}(t) \mapsto \overline{R} \downarrow$
	$\textcircled{1}, \textcircled{2}, \textcircled{3}$	$\neg \overline{R} \vee (\neg \zeta_{R+}(t)) \mapsto R \uparrow$

Table 4. PRS for synchronization failures on PCHB.

tor with L as its input is stuck-on, then the resulting circuit will constantly send synchronization signals to the environment since the buffer no longer needs to wait for $L \uparrow$. During state $\textcircled{1}$, \overline{R} might only be driven by its staticizer and is therefore vulnerable to noise and power dissipation if its state-holding element is faulty.

3.3. Token Generation and Token Consumption

Similar to synchronization failures, processes that receive and send data may generate and consume tokens (data) in the presence of a fault. Figure 4 is a template for a computation block based on the half-buffer design previously discussed [6]. Circuits based on this template

were extensively used in the design of a high-performance asynchronous microprocessor [11]. Validity and neutrality

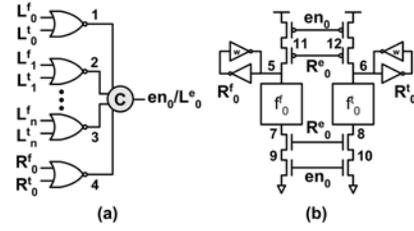


Figure 4. Template of a precharge function unit.

of the inputs and the output are check in (a) and the result of $f_0(L_0, L_1, \dots, L_n)$ is computed and sent in (b). f_0^f and f_0^t are the pull-down networks that determine R_0^f and R_0^t , respectively. During fault-free operation, the inputs, $(L_0^f, L_0^t, \dots, L_n^f, L_n^t)$, and outputs, (R_0^f, R_0^t) , start low and the enable signal is high. When the inputs arrive, only one of R_0^f or R_0^t will go low and the corresponding output will go high (assuming that R_0^e has arrived). After an output goes high and all the inputs have arrived, the enable signal will go low and the environment will set the inputs low. If R_0^e has gone low than R_0^f and R_0^t will follow and the enable goes high, returning the circuit to the initial state.

When a fault occurs in this process, the resulting circuit may deadlock, consume a token, or generate a token. Let's assume the circuit is in the state after $R_0^f \uparrow$ fired. If a fault causes $R_0^f \downarrow$ to fire after enable goes low and before the environment has received the data (0), then this token will get consumed and the circuit will return to the initial state. The following token that enters this process may be evaluated and take the place of the initial token.

If a fault results in $R_0^f \uparrow$ and $R_0^t \uparrow$ firing prematurely, such as a transient or state-holding fault at node 5 or 6, then a token will be generated. In addition to generating a logical 1 or a logical 0, there is a third kind of token that can be generated. Imagine that a fault causes $R_0^f \uparrow$ to fire just as $R_0^t \uparrow$ begins to fire. Such a token is considered illegal, however, many circuits will receive and process this type of token. The template circuit of Figure 4 makes no distinction between valid and invalid token and will receive and process both.

4. Design for Detection

We have shown that faults in asynchronous circuits can result in deadlock, synchronization failure, token generation, and token consumption. Of these failure types, dead-

lock is the most desirable because it will prevent an action from occurring rather than perform an incorrect action. Deadlock is also the easiest failure to detect since we only need to check if a circuit is making forward progress after some minimum time t . Our approach to improving detection of the remaining failure types is twofold: One, prevent all failures from performing incorrect actions. Two, make all failures result in deadlock. Also, if a fault does not lead to an observable failure, then for all practical purposes we can continue to operate as if there was in fact no fault.

4.1. Synchronization Channels

Unlike data, synchronization signals are usually sent individually from one process to another. In order to make synchronization failures more detectable, redundancy must be added to each synchronization channel. We also add an extra constraint that redundant signals should be produced in an inverted fashion with respect to the original signal. This constraint ensures that the resulting circuit is tolerant of a fault that drives both signals high or low simultaneously. A single crosstalk fault may have this affect since the wires carrying these signals are parallel to one another and may be long. Although adding redundant signals will result in at least twice the area, redundant synchronization channels are feasible because synchronization channels are a small fraction of the overall design when compared to data channels.

In order to prevent synchronization failures, a synchronization signal and its redundant counterpart must travel together throughout the circuit. To achieve this, each process receiving a synchronization signal must wait for both signals to arrive before performing any action. Figure 5 shows

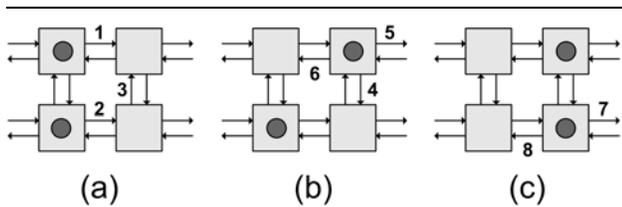


Figure 5. Redundant synchronization channels: (a) both signals arrive, (b) top signal can proceed, (c) bottom signal can proceed.

how redundant synchronization signals travel between processes with the numbered wires indicating the order in which transitions occur. In addition to a left and right set of handshaking signals, a center set of handshaking signals (denoted C and C^e) is introduced to ensure that the process waits for both synchronization signals.

The HSE for a redundant buffer is as follows (note that there are two R and L channels, one for the top and one for the bottom):

HSE for the top portion is:

$$* [[R_T^e \wedge L_T \wedge C]; C^e \downarrow; R_T \uparrow; L_T^e \downarrow; \\ [\neg R_T^e \wedge \neg L_T \wedge \neg C]; C^e \uparrow; R_T \downarrow; L_T^e \uparrow]$$

HSE for the bottom portion is:

$$* [[R_B^e \wedge L_B \wedge C^e]; C \uparrow; [\neg C^e]; R_B \uparrow; L_B^e \downarrow; \\ [\neg R_B^e \wedge \neg L_B]; C \downarrow; (R_B \downarrow; L_B^e \uparrow)]$$

This buffer takes 39 transistors to implement, with the added constraint of inverting the bottom signals. This is more than twice that of the PCHB of Figure 3, which only requires 18 transistors. This buffer is also somewhat slower than the PCHB with a forward latency of 4 transitions and a backward latency of 5 transitions.

Table 5 shows some failures that can occur when the redundant buffer is waiting for data or receiving data. As expected, this buffer may still cause synchronization failures when faults cause both $R_T \uparrow$ and $R_B \uparrow$ or both $L_T \uparrow$ and $L_B \uparrow$ to fire simultaneously. However, an interesting property of

State	Premature Firing	Resulting Failure
waiting for $[L_T \wedge L_B]$	$R_T \uparrow$ or $R_B \uparrow$	none or deadlock
	$C \uparrow$ or $C^e \downarrow$	none or deadlock
	$R_T \uparrow$ and $R_B \uparrow$	synch failure
receiving $(R_T \uparrow, R_B \downarrow)$	$R_B \downarrow$	none, then deadlock
	$L_B^e \downarrow$	deadlock

Table 5. Failures resulting from premature firings of the redundant synchronization buffer.

this buffer is that it may tolerate some premature firings, rather than deadlock. For instance, suppose $C \uparrow$ fires prematurely when the buffer is waiting for L_T and L_B . When L_T arrives, $C^e \downarrow$ will fire allowing both $R_T \uparrow$ and $R_B \uparrow$ to fire. When $R_B \downarrow$ fires prematurely during the receive state shown in the table, a fault induced synchronization signal may take the place of one of the pair. In this case, the actions that result are still correct since at least one synchronization signal is correct. This system will eventually deadlock because the signal that was replaced is trapped in the buffer.

4.2. Data Channels

The two types of failures, other than deadlock, that have been shown to occur in data channels are token generation and token consumption. One could attempt to use the same solution used for synchronization channels, however, due to the vast number of data channels, it would be impractical to make each channel redundant. To detect token gener-

ation, we will exploit the following properties of data channels: (i) adjacent bits of a data word complete parallel computations in roughly the same time; (ii) a fault might cause an illegal transition on an output wire, but the correct channel will eventually go high if given enough time. This can be detected because the net result will be an illegal state.

In the template of a precharge function block shown in Figure 4, the true-rail and false-rail of each bit of data is generated by separate pieces of logic. Legal states are 00, 01, and 10, so a 11 indicates a failure. If a fault causes the wrong rail of R to go high, then the process receiving that bit may lower R^e causing the function block to reset. However, if R^e is prevented from going low, then the correct rail will eventually go high as well, assuming that the inputs to the function unit were not faulty. If this were always the case, then we could detect these faults by checking lines for invalid tokens.

Figure 6 shows a technique to ensure that the correct rail in each bit of data will be given enough time to fire. In this diagram, there are n bits of data traveling through k

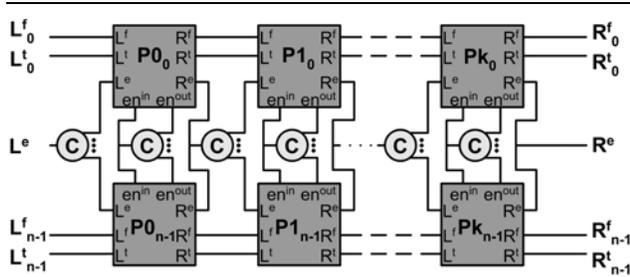


Figure 6. A pipeline of k full buffer processes that read and acknowledge data in groups.

columns of full-buffer processes. Each process in a column performs the same function on the data. The crucial property of this design is that the enables of each column are performed together, due to the insertion of C-elements. This results in each column reading and acknowledging data as a group.

Suppose a fault occurs in process $P0_1$ that causes its true rail to go high. Process $P1_1$ will lower its L^e , but the intervening C-element will prevent R^e in $P0_1$ from going low as well. R^e will only go low once the slowest bit of data in column 0 has been received by column 1. Since the processes in each column are the same, this would give the false rail enough time to go high if it was supposed to. If the fault caused an error, then the token will be invalid, if not, no error occurred.

Fortunately, this technique will result in deadlock when token consumption occurs at the bit level. Consider the case

where each process of column 0 has a token and each process column 1 has a token. In this case, the grouped enables of each column is low and the grouped R^e input to each column is high. Suppose the token in $P1_0$ is consumed by a transient fault, and the token in $P0_0$ begins to move into the $P1_0$ process. The bit L^e output of $P1_0$ will go low, however, the corresponding R^e input of $P0_0$ will remain high due to the intervening C-element. Consequently, the enable of $P1_0$ will remain low indefinitely and as a result the enable of this entire column will be prevented from going high. The configuration of Figure 6 introduces the possibility of token deletion at the group level if a transient fault occurs on the output of the intervening C-element. To remedy this, the C-elements that group the enables are implemented in a redundant fashion wherein each enable input has a dedicated 3-input C-element. The C-element that generates the R^e_i of a column has L^e_{i-1} , L^e_i , and L^e_{i+1} of the following column as inputs.

An important feature of the modified circuit is that all the C-elements that were introduced have non-isochronic input and output branches. This means that any permanent fault on a single wire on those inputs and outputs will result in deadlock as well.

4.3. Isochronic Delay Faults

It was shown in Figure 2 that a failure may occur if there is a greater delay on an isochronic branch than on the non-isochronic branches. A routing solution to prevent these failures for some subset of isochronic forks is shown in Figure 7. When there is some non-isochronic branch in a fork that has an isochronic branch, the solution shown can be used to prevent delay faults from causing an isochronic fork to fail. Rather than forking, we can route a signal through

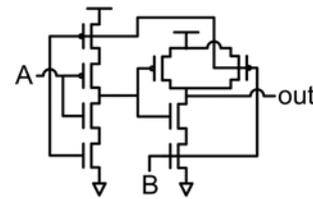


Figure 7. Routing solution to delay faults on isochronic branches.

the polysilicon of the gate that had the isochronic branch as an input, then to the gates with the non-isochronic branch input. A similar technique is used in [13]. Such a solution is practical since isochronic branches are kept local to a process, i.e., they don't exist on input or output channels. Now a fault that caused a delay on an isochronic branch will cause

an equal delay on the non-isochronic branches. Assuming that circuits follow the above routing scheme, delay faults will result in slower, but correct circuits.

5. Datapath Design Issues

There are two main caveats with the proposed detection techniques for data channels. One, a circuit is needed to detect invalid tokens in a datapath. Two, not all processes will produce an invalid token when they receive one. These issues are addressed in this section.

5.1. Invalid Token Detector

Token generating failures will produce invalid tokens, however, these invalid tokens need to be detected at some point in the datapath and deadlock the system. Since invalid tokens use an illegal encoding, both true and false rails are high. This situation can be detected by the following production rules:

$$\begin{aligned} \neg reset &\mapsto Ck \uparrow \\ L^f \wedge L^t &\mapsto Ck \downarrow \end{aligned}$$

where L^t and L^f are the two rails of a dual-rail code that (by definition) has an illegal “11” state. The *reset* signal is used to initialize the *Ck* check signal. As a final point, we can use this signal to deadlock any handshake in the pipeline. For example, *Ck* (and a complement generated using an inverter) can be used to block any of the C-elements shown in Figure 6 by modifying their input signal as follows. Instead of using input signal *s*, the C-element should use *ns* where *ns* is generated by the production rules $Ck \wedge s \mapsto ns \uparrow$ and $Ck \wedge \neg s \mapsto ns \downarrow$ (referred to as the “switch” gate [9]). Now when $Ck \downarrow$ occurs, the next stage will be blocked permanently, thereby deadlocking the system.

This checker uses a timing assumption, but it is one that is very liberal. It assumes that the delay through the next pipeline stage and the C-element shown is larger than the time it takes for the *Ck* signal to transition. Satisfying this assumption is a trivial task in practice. A more interesting property of this checker circuit is that it also identifies *where* the fault occurs, and prevents it from propagating beyond the next pipeline stage and corrupting the state of the rest of the system.

5.2. Error Propagating Processes

An error in the output of one pipeline stage might be filtered out by the next stage, depending on the computation being performed. Sometimes errors will propagate through a pipeline. A process that produces an invalid token output when it receives an invalid token input is said to be error propagating. Consider the pull-down networks for the

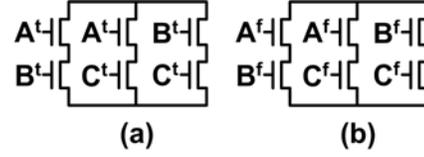


Figure 8. The pull-down network for an adder carry: (a) true rail, (b) false rail.

carry out of an adder in Figure 8. Results from invalid inputs into the carry function are shown in Table 6. For two cases of invalid inputs, the carry circuit generated valid outputs. In this case, the outputs are correct since replacing the invalid token with either valid token will yield the same result. Processes that are not always error propagating may

A		B		C		Carry	
T	F	T	F	T	F	T	F
1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	0
1	1	0	1	0	1	0	1
1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1

Table 6. Output tokens resulting from a carry process with invalid input tokens.

prevent an invalid token from being detected and they must be made self-checking by the method of invalid token detection discussed in the previous subsection.

6. Experimental Results

Experimental results are obtained by applying our detection/isolation technique to two 64-bit 3-stage pipelines. The first pipeline contains a 64-bit buffer and the second contains a 64-bit AND-function unit, each of which receive input from a bit generator and send output to a bit bucket. Initially, each stage generates a single enable signal, therefore we partition each stage into smaller segments where each segment has an enable signal. The detection/isolation technique is applied to this set of pipelines and simulated in HSPICE using TSMC 0.18 micron technology. The results are shown in Tables 7 and 8. We report the transistor count, average power consumption, and period (total number of transitions per cycle) for the original pipelines and the modified pipelines. The number of partitions is denoted by *N*. The numbers that appear in parenthesis after the period is the period we would obtain by performing bubble reshuffling after adding the redundant C-elements.

The performance penalty of applying our detection/isolation method is fixed at 4 transitions per cycle (2

N	T.-Count		Avg. P.(mW)		Period	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
1	3324	N/A	6.03	N/A	16	N/A
2	3336	3384	7.56	6.41	16	20(18)
4	3296	3392	10.1	8.23	12	16
8	3328	3520	12.9	10.6	12	16
16	3264	3646	16.1	13.6	12	16(14)
32	3456	4224	20.1	18.7	12	16(14)
64	2116	4352	20.7	25.4	8	12

Table 7. Experimental results for a 64-bit buffer.

when bubble reshuffling is possible). The number of additional transistors required increases with the number of partitions because each partition requires a dedicated redundant C-element. Since we may freely choose the number of partitions in this method, we can make a tradeoff between the additional hardware required and the desired period. For example, we may choose a 64-bit AND-function unit with 16 partitions and pay an increase in hardware of 7%, but maintain the original period of 16 (the original case being 1 partition).

N	T.-Count		Avg. P.(mW)		Period	
	Orig.	Mod.	Orig.	Mod.	Orig.	Mod.
1	5116	N/A	7.85	N/A	16	N/A
2	5132	5180	9.44	7.71	16	20
4	5104	5200	11.9	10.0	16	20(18)
8	5168	5360	14.9	13.7	16	20(18)
16	5056	5440	18.7	17.2	12	16
32	5312	6080	23.6	22.6	12	16
64	4864	6400	24.6	28.7	12	16(14)

Table 8. Experimental results for a 64-bit AND-function unit.

7. Summary

We presented a detailed description of different faults and their effect on asynchronous QDI circuits. These circuits, by their very nature, are highly tolerant of any delay fault. Other faults such as stuck-at, stuck-open/closed, bridging faults, and transient faults and their impact on asynchronous circuits was presented first at the gate level, and then the gate faults were translated to failure in communication channels that occur at the interfaces of asynchronous components. In particular, errors in terms of deadlock, synchronization failure, token generation and token consumption were identified. Two modifications to conventional QDI circuits were described that dealt with pure synchronization channels and data channels respectively. These modifications translated errors into circuit deadlock,

thereby making the errors visible and preventing them from propagating far away from their origin. Layout techniques to mitigate delay faults in some isochronic forks were also presented. Finally, methods to translate invalid tokens into deadlock along with detecting where the fault occurred were also described.

References

- [1] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000.
- [2] P. J. Hazewindus. *Testing Delay-Insensitive Circuits*. PhD thesis, California Institute of Technology, Pasadena, California, 1996.
- [3] H. Hulgaard, S. M. Burns, and G. Borriello. Testing asynchronous circuits: a survey. *Integr. VLSI J.*, 19(3):111–131, 1995.
- [4] J. B. Khare and W. Maly. *From Contamination To Defects, Faults and Yield Loss*. Kluwer Academic Publishers, 1996.
- [5] P. K. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann Publishers, 2001.
- [6] A. M. Lines. Pipelined asynchronous circuits. Master’s thesis, California Institute of Technology, Pasadena, California, 1996.
- [7] R. Manohar and A. J. Martin. Quasi-delay-insensitive circuits are Turing complete. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, 1996.
- [8] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proceedings of the Mathematics of Program Construction*, pages 272–285. Springer-Verlag, 1998.
- [9] A. J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed Computing*, 1:226–234, 1986.
- [10] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 302–311, 1990.
- [11] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous mips r3000 microprocessor. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI ’97)*, page 164. IEEE Computer Society, 1997.
- [12] Y. Massoud, S. Majors, J. Kawa, T. Bustami, D. MacMillen, and J. White. Managing on-chip inductive effects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(6):789–797, 2002.
- [13] V. I. Varshavsky. Circuits insensitive to delays in transistors and wires. Technical report, Helsinki University of Technology, November 1989.
- [14] A. Yakovlev. Structural technique for fault-masking in asynchronous interfaces. In *IEE Proceedings E - Computers and Digital Techniques*, pages 81–91. IEEE Computer Society, 1993.