

Fault Masking in Synchronous and in Asynchronous Logic – A Comparison

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Wolfgang Ramsl

Matrikelnummer 0526694

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dr. Vorname Familienname

Wien, TT.MM.JJJJ

(Unterschrift Verfasser)

(Unterschrift Betreuung)



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Wolfgang Ramsl
Siedingerstrasse 19, 2631 Sieding

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would first like to thank my thesis advisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger. He consistently allowed this thesis to be my own work, but steered me in the right the direction whenever he thought I needed it. Finally, I must express my very profound gratitude to my parents and to my wife Sandra, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

The topic of this diploma thesis is the investigation of fault masking effects in synchronous and asynchronous logic. A fault is said to be masked if it affects a circuit but never creates an erroneous state and hence stays ineffective. In synchronous logic it is known that faults can be masked on three different levels: (1) Electrical masking: A fault is injected on the electrical level, but it doesn't affect the logical level. The current pulse induced is not large enough to change the boolean value. (2) Logical masking: The fault changes a boolean value but the logical function which is performed on this signal does not take it into account. For example if you look at an AND-Gate (2 inputs), a „false“ logic 1 only propagates if the other input is also 1. This we call implicit logical masking. Explicit logical masking is related to majority voting with replicated functions. (3) Temporal (latching-window) masking: This level deals with the temporal behavior, the fault disturbs a signal but it isn't captured. For example a transient fault between two clock edges in a synchronous circuit has no effect on the storage element as long as its effect has vanished by the next clock event.

While (1) and (2) work similarly in synchronous and asynchronous logic, temporal masking (3) will be different. Instead of the rigid clock in synchronous logic there is a flexible timing driven by completion detection. The consideration of skew effects will be one focus of this thesis, whose general aim is to investigate the masking effects in both theory and practice.

The result of the diploma thesis is expected to be a model which explains the behavior of masking effects on the three different levels. With the help of this model the appearance of faults and the behaviour of masking effects in synchronous and asynchronous logic should be better understood. We will get a (also quantitative) comparison between masking effects in synchronous and asynchronous logic because those effects are already investigated in synchronous logic.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Der Inhalt dieser Diplomarbeit ist die Untersuchung der Fehlermaskierung in synchroner und asynchroner Logik. Ein Fehler wird als maskiert angesehen wenn er sich auf die Schaltung auswirkt und niemals einen fehlerhaften Zustand bewirkt, also inaktiv bleibt. In synchroner Logik unterscheidet man folgende 3 Arten der Maskierung von transienten Fehlern: (1) Electrical masking: Ein Fehler wird auf elektronischer Ebene injiziert, aber er wirkt sich nicht auf der logischen Ebene aus. Der induzierte elektrische Impuls ist nicht stark genug um einen Wert im Speicher zu verändern. (2) Logical masking: Der Fehler ändert einen booleschen Wert aber die logische Funktion welche damit ausgeführt wird, ignoriert diesen. Als Beispiel könnte man hier ein Und-Gatter (2 Eingänge) anführen, eine fehlerhafte „1“ an einem Eingang wirkt sich nur dann aus, wenn am anderen Eingang auch eine „1“ anliegt. Dieses Verhalten nennt man „logische Maskierung“. (3) Temporal (latching-window) masking: Diese Art der Maskierung wird beeinflusst durch das zeitliche Verhalten der Schaltung. Zum Beispiel ein transienter Fehler zwischen zwei Takt-Flanken in synchroner Logik hat keinen Effekt auf das Speicherelement solange der transiente Fehler wieder vor der nächsten Takt-Flanke verschwindet.

Es kann angenommen werden, dass sich (1) und (2) in synchroner und asynchroner Logik gleich verhalten, temporal masking (3) aber anders ist. Statt eines strikten vorgegeben Clock-signals gibt es hier ein flexibles Zeitverhalten welches durch Logik (completion detection) gesteuert wird. Die Betrachtung von „skew“ Effekten ist ein besonders wichtiger Punkt in dieser Diplomarbeit. Ziel ist es, die Maskierungseffekte sowohl in Theorie als auch in Praxis beschreiben zu können.

Es soll in dieser Diplomarbeit ein Modell erzeugt werden, welches die Maskierungseffekte auf den drei unterschiedlichen Ebenen beschreibt. Mit Hilfe dieses Modells können dann das Auftreten von Fehlern und das Verhalten von Maskierungseffekten in synchroner und asynchroner Logik besser verstanden werden. Es wird einen (quantitativen) Vergleich zwischen Maskierungseffekten in synchroner und asynchroner Logik geben, da diese Effekte in synchroner Logik schon untersucht worden sind.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Contents	ix
1 Introduction	1
1.1 Motivation	1
1.2 Aims and Scope	2
1.3 Structure of the Master's Thesis	3
2 Background	5
2.1 Asynchronous logic	5
2.2 Classification and Operating modes	6
2.3 Handshake Protocols	7
2.4 Pipeline and Data flow control	10
2.5 Fault Models and Masking effects	11
3 Comparison of Models	17
3.1 Token and Transition Based Fault Description	17
3.2 State Graph	21
3.3 Signal Transition Graph	22
3.4 Trace Based Description	23
3.5 Probability Model	28
4 Fault Injection Setup	31
4.1 4-phase bundled data	31
4.2 2-phase bundled data	39
5 Simulation	45
5.1 Simulation of 4-phase bundled data pipeline	49
5.2 Simulation of 2-phase bundled data pipeline	55
6 Comparison	61
6.1 Synchronous Pipeline	61
6.2 Masking Effect Results	65
7 Summary	71
	ix

7.1 Summary	71
Bibliography	73
A Code	77

Introduction

The first chapter is an introduction to the diploma thesis to give a motivation why the analysis of robustness and masking effects of synchronous and asynchronous logic is important for the design of logic circuits.

1.1 Motivation

There is a trend for integrated circuits always to get smaller, faster and higher integrated while the costs are getting lower and lower. As the design becomes smaller, the supply voltage can be reduced, which leads to higher power efficiency. All those positive effects are paid for by an increased fault sensitivity, as the amount of charge which is necessary to change the logic value in the circuit, is becoming smaller. Figure 1.1a shows the soft error rate (SER) versus the technology, described as feature size. The y-axis in this graphic is log scaled, so the actual increase is exponential across this range of feature sizes. As a result soft-errors, which are caused by cosmic neutrons, are becoming a major source of errors in modern integrated circuits. It has been shown that compared to memory cells, a smaller critical charge is necessary to change the logic value in a logic cell and since particles with lower energy occur far more often than particles with high energy, this is the reason for a higher soft-error rate, which is shown in figure 1.1b. This figure shows soft-error rate versus technology generation in nanometer.

The soft error rate for SRAM didn't change significantly. The reason for this is the compensation of smaller critical charge by a disproportionate reduction in cell area and an improvement in technology. However, for logic elements and latches the soft error rate grows as predicted by the critical charge reduction.

There are two main styles of hardware architectures- synchronous and asynchronous logic. In contrast to asynchronous logic, in synchronous logic all events are driven by one or more global or distributed timing signals called clocks. Asynchronous logic promises a number of advantages as shown in table 1.1. The two aspects power efficiency and higher robustness were already mentioned before and good modularity of different components means that it can be

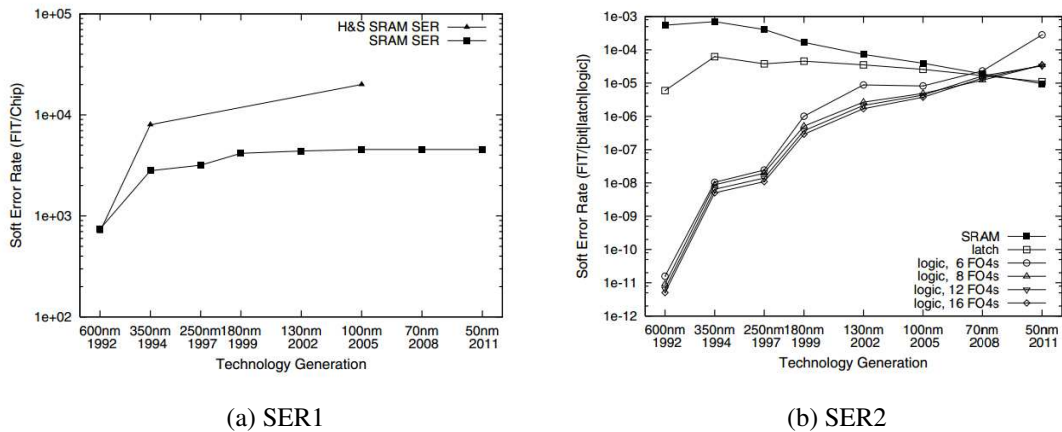


Figure 1.1: SER [21]

easier to connect parts with different asynchronous clocks. Since the handshake mechanism that asynchronous circuits are based on guarantees that the connection is functioning correctly, it autonomously adapts to the slowest device. Transitions only take place when they are needed, which reduces electromagnetic emission and susceptibility as well as dynamic power consumption. On the other hand designing asynchronous circuits is also challenging, because hazards and race conditions must be carefully considered. Also more space for completion detection and handshake is used because it has to be recognized when new data is ready or can be computed by the next stage in a pipeline. The investigation of asynchronous logic with its advantages is becoming more important, to have a comparison to synchronous logic. There is a good reason why synchronous logic is almost exclusively used in today's companies: Tools for developing asynchronous logic are in the minority compared to synchronous tools and often limited to university usage. Although synchronous tools can be used for developing asynchronous logic, their outcome is not as satisfactory, or often they can not produce the right solution. There is also a lack of tools for testing and test vector generation. Asynchronous circuits are used less frequently because their design is more time intensive and there is not so much knowledge about them, so there has to be more investigation on this topic. While there was a lot of research done on fault tolerance in synchronous systems, less attention has been paid to asynchronous circuits. Well established error detection techniques can not be directly applied to asynchronous designs. Nevertheless asynchronous logic can constitute a promising alternative for fault-tolerant systems because of high robustness and reliability.

1.2 Aims and Scope

There is still a lot of discussion whether asynchronous logic is better and there are not so many research papers which investigate this topic in detail. A paper which gives a promising result about the advantages was published by Babak Rahbaran and Andreas Steininger [6]. The robustness of two versions (synchronous and asynchronous) of the same processor was compared

Table 1.1: advantages of asynchronous circuits

advantages	reason
power efficiency	smaller feature size
more robustness	no clock, elastic timing
modularity	handshake, elastic timing
average case performance	inherent speed regulation
no clock skew	no inherent clock signal distribution
reduced electromagnetic emission	no regular switching operation
reliability	no single point of failure

by fault injection. The result was that the asynchronous circuit showed a better performance in the presence of faults.

The main objectives of this diploma thesis are to give a better understanding of asynchronous logic, to show if there are also masking effects as they exist in synchronous logic, and to make a comparison between the two main design styles. A model will be developed to easily recognize the behavior of the circuits and see where and when faults are happening. Further literature can be found in [7] (clock generation scheme), [2] (asynchronous design) and [13] (fault detection).

As a result there will be a (also quantitative) comparison between masking effects in synchronous and asynchronous logic because those effects are already investigated in synchronous logic. With the help of simulations, different numbers of failures will be injected into different asynchronous logic circuits and consequently a hypothesis about the behavior in different fault situations can be proven.

There will be an investigation of transient faults in two sorts of bundled data pipelines (4-phase and 2-phase) without using hardening techniques for reducing errors. The assumption for fault injection will be the single fault model. In the context of this diploma thesis a fault simulator for bundled-data pipelines (4-phase and 2-phase) with three stages will be developed, to show the result of fault injection and compare the simulation results to those computed by the formula of the created fault model.

1.3 Structure of the Master's Thesis

The diploma thesis is structured into six main chapters. The first two chapters discuss the motivation for this thesis and introduce the reader to the main methodologies of asynchronous and synchronous logic. Important vocabulary, used pipeline circuit designs and already existing fault models are explained to build a basis for the understanding of the following chapters.

Chapter 3 deals with the problem of how to get to a suitable model for analyzing faults in asynchronous logic, especially in asynchronous pipelines and shows how to develop a suitable fault model out of already existing ones.

After the fault model has been developed and discussed, the hypothesis derived from this model has to be proven by simulation. Chapter 4 shows the setup and purpose of simulation and discusses the difficulties if fault injection is made in hardware as well.

Chapter 5 reports the results of the simulations and will compare these to the predictions of

the fault model. After we discussed the behaviour of fault injection in asynchronous logic we also take a look into synchronous logic in chapter 6, to make a comparison to the results that we found in the previous chapters. The last chapter of this thesis will give a conclusion and an outlook on promising advantages, which can be further elaborated in future works.

In Appendix A, a short explanation of how to use the code for the fault injection simulation is given. A CD with the source code of the fault injector is attached to this diploma thesis.

Background

This chapter introduces the necessary background to understand the main terms of this diploma thesis. First a short introduction to asynchronous logic and masking effects is provided. Then the most common styles of modeling systems are shown and examples are given.

2.1 Asynchronous logic

To make the design of digital logic easier, we generally assume the existence of a global, discrete time, defined by a clock signal. This timing model of a circuit solves many problems as it is unimportant what happens between two consecutive clock edges since the values which the operation is made on, are only used after they have settled to a stable and valid state. Propagation delays, glitches or hazards do not affect the outcome of a logic function as long as they occur between two consecutive clock ticks. This important property is expressed by the setup and hold-timing of synchronous logic. The setup-time defines the time when a new value has to be ready at the input of a component before the clock edge happens. Similarly the hold time defines how long the value has to be valid after the edge. Synchronous logic follows a discrete time scale, which is the major difference to asynchronous logic.

In asynchronous logic there is no clock and the operations in the circuit are made as soon as the input values are stable and ready, so it is not so easy to make a timing analysis. Synchronization is done by handshaking to perform a sequencing of operations. Handshaking is done by sending request and acknowledge signals. As soon as the sender knows that the receiver is ready, for new data – which is signaled by the acknowledge signal – and the new data is stable on the data lines, the request signal is activated. This signals the receiver to perform its operation and after finishing, it sends the acknowledge signal back to the sender. There are two main styles of asynchronous logic, which differentiate if the sequencing of operation will be done in equally timed steps or after arbitrary delay, determined by the time it takes for the operation to calculate. The two main classifications are:

- bounded delay: constraints on the propagation delay between different components
- unbounded delay: no constraints are made, arbitrary delays in the circuits are allowed

In the bounded delay model we have to know the worst case execution time for the function between sender and receiver in advance, and we have to guarantee that the functions will never take longer. It will always take the same time until the request signal will be sent to the receiver, to signal that new data is available and ready at the input line. In the unbounded delay model we make no assumption about the readiness and how long it will take to transfer a signal from sender to receiver. Unlike with bounded delay, the sender has no information about the timing. The receiver has to recognize if new data is available, stable and ready for input. This can be done by coding - the receiver has some logic in front of its input port to interpret if the data can be used. Two important coding styles to encode validity information will be introduced in the next section, when the four main hand-shake protocols will be introduced.

2.2 Classification and Operating modes

To distinguish the different styles of asynchronous logic we have to make a classification, like it is made in [22]. With the help of the main classification, described in the last chapter, into bounded and unbounded delay we can now make a better differentiation and will discuss the subcategories of those models in more detail.

Within the not bounded delay model two main categories can be distinguished, speed-independent (SI) and delay insensitive circuits (DI). A speed-independent (SI) circuit is a circuit that operates correctly assuming positive, finite but unknown delays in gates and ideal zero-delay wires. DI circuits do not apply any delay restrictions, neither on gates nor on wires (except finiteness). Unfortunately, the class of DI circuits is limited to circuits that only consist of inverters and Muller C-gates. A less restrictive subclass of DI are quasi-delay insensitive circuits which allow for positive but unbounded delays in all elements except in isochronic forks. Such forks assume that delays on different paths are negligible, i.e a transition which starts at the root of the fork will reach the the end of each branch at the same time. With this assumption this class becomes more practical and bigger. If all forks in the QDI circuit are required to be isochronic, the circuit essentially becomes a SI circuit.

Bounded delay models require certain timing assumptions and operation modes - the timing behavior has to be known in advance. As we can see, synchronous circuits can also be assigned to the class of bounded delay models, as all transient states have settled to a steady state before the next clock edge happens. Synchronous and asynchronous circuits which match these classifications are also referred to as self-timed logic [22].

The classification of asynchronous logic according to its delay model is not enough - we also have to define the interaction with the environment ([22]). The two main operation modes are

- Fundamental mode
- Input- Output Mode

In fundamental mode it is not allowed to apply the next input until the circuit has settled to a stable state. For that reason a worst case timing for the circuit has to be calculated because the internal states are not visible to the environment. Fundamental mode circuits always adhere to the bounded delay model. In a classical model it is only allowed to change one input signal at a time but this model can be extended to burst mode, where more than one signal can be changed. But after a burst the environment has again to wait until the circuit is stable before sending the next burst. The environment can not distinguish if the circuit is now in stable state by only looking if the output signals have changed, since they could change again because of internal intermediate states.

In input-output Mode the circuit also has to be stable. But here the environment can apply the next input signal as soon as the circuit has changed its output, so there exists a causal relation between input and output transition. The internal signal as well as the internal states are not observed by the environment, so every transition at the output must be a valid one and no intermediate output transition is allowed. The behavior of such a system can be analyzed by a transition- based model. An example on how to model such system will be given in the next chapter.

2.3 Handshake Protocols

As stated before, two handshake events are necessary to guarantee a synchronized communication and in normal case these events will happen alternately:

- Request event: signals that new data is available
- Acknowledge event: new data can be consumed

Here two main styles have to be distinguished but other protocols can be applied as well:

- Bundled-data protocol: separate line for request
- Delay insensitive data encoding: no separate request line

In the bundled-data protocol the handshake information is explicitly transmitted together with the data - there is an additional request line. This handshake protocol consists of one request signal and one acknowledge signal for N data signals. Therefore the request signal has to be delayed to guarantee that the information data is valid when the request signal arrives at the receiver. Bundled-data circuits follow the bounded delay model and require a positive timing margin between the propagation-delay of the request line and the worst case propagation delay on the data lines. There is no such timing assumption for the acknowledge line required.

In the dual rail protocol there is no separate request line. The information when new data is available at the receiver's input is encoded in the data signals. Therefore the receiver needs a special logic function which interprets the information at the input (completion detector). Dual rail means that for each bit of information which is sent to the receiver two wires are used to encode additional request information.

It is also necessary to decide if information is encoded in the transition or in the state of a signal line (edge or level encoded). This also leads us to the 4 different styles of handshaking methods, which are „4-phase bundled data“, „2-phase bundled data“, „4-phase dual-rail“ and „2-phase dual-rail“.

Two of them are from the bounded delay style - the worst case delay has to be known while designing the circuit - the other two can handle arbitrary delays; no assumptions are made on the timing. In the following subsections the different handshaking styles are explained in detail.

Bundled Data Handshake Protocols

Diagrams which show the connection between sender and receiver and the timing diagrams for the two bundled-data timing styles can be seen in the figures 2.1a and 2.1b. The communications lines are one request and one acknowledge line and for the codewords n data lines.

In 4-phase bundled data the communication action takes place in 4 stages: (1) The sender issues data and sets the request signal to high, (2) the receiver absorbs the signal and sets acknowledge to high, (3) the sender responds by setting request to low (typically showing that data isn't guaranteed to be valid anymore) and (4) receiver acknowledges this by setting acknowledge to low. At this point the sender can start a new transmission.

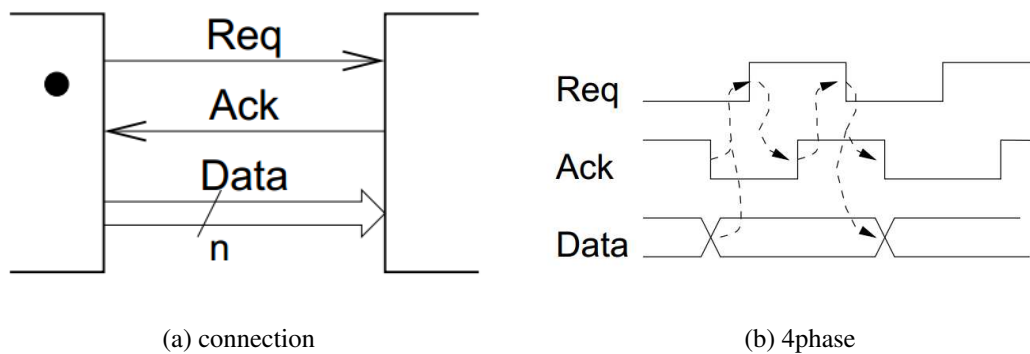


Figure 2.1: 4-phase timing diagram [22]

In contrast to 4-phase bundled data, in 2-phase bundled data the request and acknowledge signal information is encoded in the signal transition. It takes only 2 protocol phases to transmit data, which is generally more complex but doesn't waste the time for reset phases. Some asynchronous designs use the benefits of both protocols by applying the 4-phase protocol for the computation part and the 2-phase protocol for the interconnection parts. In figure 2.2 we can see that information is encoded in the transition and no return-to-zero phase is used. The data communication needs 2 stages: (1) The sender issues a new data word and produces a transition at the request line to signal the receiver a new word is ready at the input lines of the receiver. If the request line was high then it is low now and if the line was low then it is now high. (2) The receiver absorbs the codeword and changes the state of the acknowledge line. The sender can detect a state change of the acknowledge line and will start with the next communication cycle.

It seems that the 2-phase bundled data is less robust against fault injection than the 4-phase protocol because all depends on the correct timing of the transitions but we will discuss this issue in the later chapters.

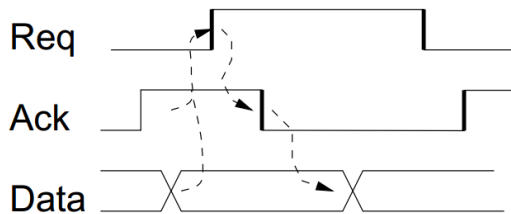


Figure 2.2: 2-phase protocol [22]

Dual-rail Handshake Protocols

As already highlighted, real DI circuits do not have a lot of practical applications. Therefore QDI is applied instead, with the limitation that isochronic forks have to be acceptable- which is often the case in practical designs. Due to unbounded delay it is more flexible and calculation times for functions - for example between pipeline states - can change according to input speed.

The 4-phase dual-rail handshaking 2.3, has the advantage of needing no separate request line. It is comparable to a 4-phase protocol using 2 wires per bit of information d . One wire $d.f$ is used for signaling logic 0 (or false) and another wire $d.t$ is used for signaling logic 1 (or true). When observing a 1-bit channel one will see a sequence of 4-phase handshakes where the participating „request“ signal in any handshake cycle can be either $d.t$ or $d.f$. Viewed together the $\{x.f,x.t\}$ wire pair is a codeword. $\{x.f,x.t\} = \{1,0\}$ and $\{x.f,x.t\} = \{0,1\}$ are valid data (logic 0 and logic 1 respectively) and $\{x.f,x.t\} = \{0,0\}$ represent „NULL“ or „no data“. The codeword $\{x.f,x.t\} = \{1,1\}$ is not used and a transition from one valid codeword to another without a „NULL“ word in between is not allowed.

This leads to a more abstract view of this handshake scheme (Figure 2.3): (1) The sender issues a valid codeword, (2) the receiver acknowledges the reception of the codeword by setting the acknowledge line to high. (3) The sender responds by sending the „NULL“- word, (4) and after reception of the „NULL“-word, the receiver sets the acknowledge line to low. After those 4 steps the next communication starts at the sender by issuing the next valid codeword. In a more abstract way we see a data stream of valid codewords, separated by empty codewords.

The 2-phase dual rail handshaking mechanism will be explained here but will not be part of the simulations with fault injections in pipelines. The 2-phase dual-rail protocol uses also 2 wires for the codewords but the communication is encoded as transition. On an N-bit channel a new codeword is received when exactly one wire in each of the N wire pairs has made a transition and there is no empty codeword. The communication cycle for sending one codeword takes place in two stages: (1) The sender changes one bit of the codeword. (2) The receiver

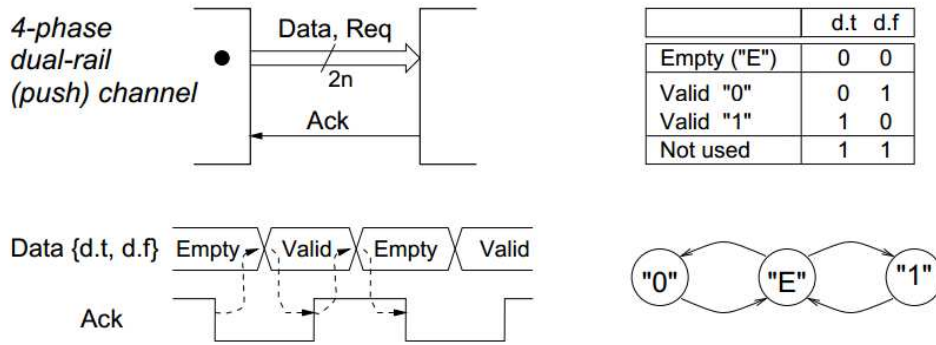


Figure 2.3: Four phase dual rail diagram [22]

detects the valid codeword by completion detection logic, absorbs it and changes the state of the acknowledge line. After those two steps the communication cycle starts again. The completion detection works by differentiation to different phase states, which have to be accessed alternately. More information about 2-phase dual-rail protocols can be found in [22].

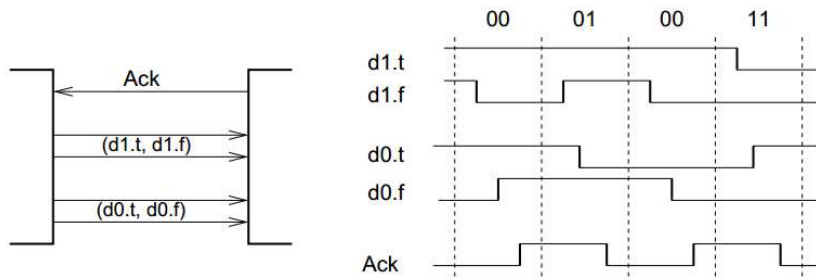


Figure 2.4: FSL [22]

2.4 Pipeline and Data flow control

A fundamental requirement in the design of a digital system is to control the sequence of input data of a logic function unit in such a way that the respective sequence of output data from a logic function unit can orderly be received by the subsequent unit.

A pipeline is a division of operation steps in stages. The operations are made in discrete synchronous steps with a given clock or asynchronous without any clock but with the help of handshaking and completion detection. A picture of a part of a pipeline with 2 stages is given in picture 2.5 .

Data is passed from stage i to stage $i+1$ via the (optional) transition function $f(x)$. The handshake is controlled by the registers, with the help of completion detection or request and acknowledge signals. A more detailed description of the used pipeline architectures of 4- and 2-phase bundled data will be given in chapter 4.

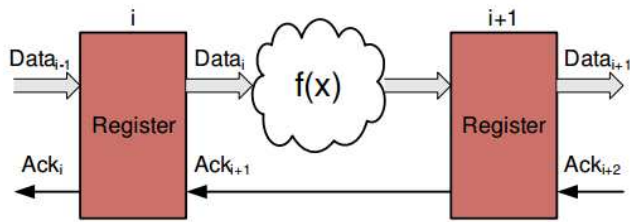


Figure 2.5: 2-stage pipeline model

2.5 Fault Models and Masking effects

Before discussing the different kinds of faults we have to define some frequently used terms related to error detection and fault tolerance. The terms *failure*, *error* and *fault* are understood as established in the Working Group 10.4 (WG10.4) on Dependable Computing and Fault Tolerance of the International Federation for Information Processing.

- A *failure* occurs when the delivered service deviates from the correct system function
- An *error* is a deviation of the internal state of a system from its correct state. An error can - but doesn't necessarily- lead to a failure of the system.
- A *fault* is the adjudged or hypothesized cause of an error.

Examples for faults can be a random break of a wire, or a design fault either in software or hardware.

So a fault generates an error but an error does not always lead to a failure of the system. The produced error will lead to a failure provided the error changes the intended behavior of the system. In other words the error must propagate to the system boundaries to trigger a failure. This causal chain is continued to next higher level. A failure may be regarded as a fault on the higher system level.

The ability of a system to function correctly, i.e. without the occurrence of failures, in the presence of faults is called *fault tolerance*. To ensure fault tolerance, two features can be incorporated into the system: *error detection* and *system recovery*.

Error detection is carried out by checking the results of the transition functions for testing their correctness. Error detection can take place during operation or the system is stopped when checking the function results. Those two operation schemes are called Concurrent Error Detection (CED) and preemptive error detection.

After the error detection is completed the system enters the system recover phase. This phase consists of two stages: *error handling* and *fault handling*. Error handling is the correction of any errors detected during the error detection phase. One error detection mechanism are CRC checksums which are often used in memory or data transfer protocols. Fault handling is the

prevention of existing faults from generating subsequent errors.

The ability of a system to show fault tolerance even in absence of error detection and error recovery mechanisms is called *robustness*.

Since in this thesis we want to investigate the robustness of systems and the mechanisms of error propagation and masking, we will not provide error detection or error recovery mechanisms. The correct functioning of the services will be checked by comparing the output of the system with the output of a correctly functioning system.

A classification of faults can be made according to their persistence, into transient and permanent faults. Also the term *intermittent* fault can be added for faults which occur repetitively but not continuously. [22]

Transient faults may be produced by three effects: High energy cosmic neutrons that interact with the silicon nuclei of semiconductor devices, low energy cosmic or thermal neutrons that interact with insulation layers and alpha particle radiation due to package imperfection. As today's integrated circuits generally use advanced processes with purified materials, high energetic cosmic neutrons are the dominating radiation effect.

Radiation leads to ionization effects in the circuit (neutrons ionize indirectly by interacting with the atomic nuclei) and thus to charge injection into the conduction band of the semiconductor.

Beside radiation, transient faults can also be provoked by electromagnetic interference (EMI) due to external sources or signal integrity problems such as a ground bounce. Transient faults - especially those generated by particle impacts - can be modeled by boolean signals. If the injected charge is high enough a logic transition is generated. At the same time the injected charge is restored by the node's driver, thus the disturbed signal will return to its initial state after the fault duration. Eventually the transient fault manifests as a positive or negative digital pulse on the specified signal.

If a transient fault is injected into a circuit without feedback elements, it will only generate a logic pulse at the output of the circuit. Especially in space engineering, such pulses are called Single Event Transients (SET). In circuits with feedback elements, e.g. latches, a transient fault may be memorized and generate a permanent upset or error, which is also referred to as Single Event Upset (SEU) or simply a soft error.

Permanent faults are typically caused by physical defects, such as fabrication imperfections. Contrary to a transient fault, a permanent fault can not be removed. This fact has to be observed, when both permanent and transient faults can corrupt memory values. Transient faults may result in a soft error which can be restored by updating the memory value but permanent faults can not be corrected. Therefore the effect of a permanent fault is also called a hard error.

Faults can be modeled in different ways: One popular method is the Single Stuck-At Fault (SSAF), which disconnects the circuit node from its surrounding elements and forces the isolated node either to be power supply or ground. The result is either a stuck-at 1 or a stuck-at 0

fault. This model was defined for permanent faults, but can be applied to transient faults as well - the difference lies in the fault duration. A drawback of using this model for transient faults is the inherent activation problem [5].

There is a certain probability that the fault will force the affected signal to its anticipated value. This means that the change of the signal which is induced by the fault is identical to a change that would have taken place at the same time if the system had functioned correctly. In this way the fault has no effect and no error is produced. This possibility has to be taken into account when analyzing systems by fault experiments, otherwise the results can be falsified. An alternative model is the bit-flip model. It is popular because it inverts the logic value of the victim's signal and therefore avoids an activation problem of stuck-at faults. A simple inversion of the fault-free signal is not a good representation of a transient fault because if during the fault duration the signal changes, the bit-flip model will again change the state of the model, further information can be found in [6].

A more realistic representation is done by the pulse model that forces the fault-free signal to the inverse value at the fault occurrence and maintains that state during the complete fault duration. To understand the propagation of signals and errors within the system it is necessary to consider the delays that occur during internal signal transmission. In the past gate delay was the major delay source in integrated circuits but with smaller feature size the delays in wires and interconnections become the major part of delay and determine a circuit's performance. Thus delay faults are gaining more importance, especially for devices with very high quality and reliability requirements.

Another type of fault that is getting more important are open faults, which exist because of disconnections. They isolate the circuit node from its environment. Since the node has no associated driver any more its logic state is controlled by the surrounding environment (e.g. noise). Still another type of fault is a bridging fault. It occurs, when the logic state of a node called victim is controlled by another signal called aggressor.

Not every fault leads necessarily to an error. The possibility of a fault staying dormant and thus invisible is called *masking*. In general there are three main reasons for fault masking:

1. Electrical masking: A fault is injected on the electrical level, but it doesn't affect the logical level. The current pulse induced is not large enough to change the boolean value.
2. Logical masking: The fault changes a boolean value but the logical function which is performed on this signal does not take it into account. For example if you look at an AND-Gate (2 inputs), a „false“ logic 1 only propagates if the other input is also 1. This is called implicit logical masking. Explicit logical masking is related to majority voting with replicated functions.
3. Temporal (latching-window) masking: This level deals with the temporal behavior. The fault disturbs a signal but it isn't captured. For example a transient fault between two clock edges in a synchronous circuit has no effect on the storage element as long as its effect has vanished by the next clock event.

4. Code masking. This can be compared to logical masking, which is mentioned in [5], since the logic function of the receiver does not evaluate the token. Contrary to logical masking not the data value leads to the masking effect but the code phase of the data prevents a fault from becoming active.

Graphic 2.6 shows how masking effects help to mitigate the propagation of faults, it can be seen as an inverse fault tree. It is sufficient for the tree to have at least one valid branch to mask a fault. As an example a transient fault is injected at the input. The temporal masking rejects a fault because the subsequent system node is not ready. The code masking shows a QDI circuit that holds data in code phase p_0 and waits for phase p_1 . A fault corrupts one bit of the previous phase p_0 , which produces inconsistent data that is not processed. The node waits until all codewords are in p_0 , and thus in a stable state. Electrical masking is not further explained here because the experiments in this thesis deal more with logical, code and temporal masking.

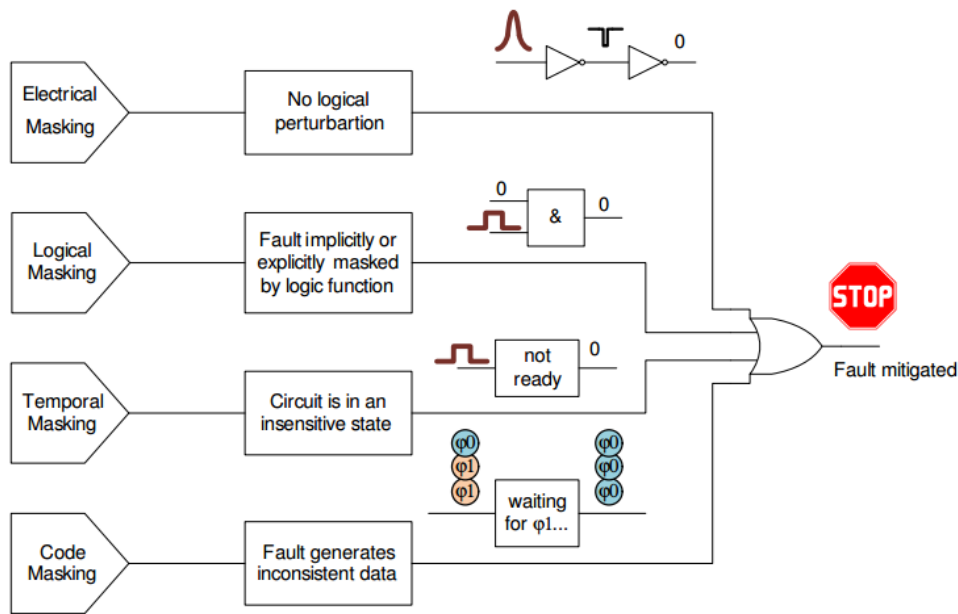


Figure 2.6: Fault mitigation

A fault model summarizes all previously defined boundary conditions: which faults are assumed to occur, the fault classification, the logical model and the applicable masking effects. It is a common approach to limit faults to one single physical system part per time, which is referred to as the single fault model. In single rail logic, such as the common synchronous logic, this restriction means that only one single boolean variable is affected by the fault. To utilize the single fault model in QDI logic one single rail is affected by one fault per time. The single fault model also assumes that consecutive faults are separated in time and not interleaved. Finally the single fault model only limits the fault itself as being a singular event. It

doesn't tell anything about its effect, e.g. it might be possible that a single fault leads to multiple errors that propagate through the system.

Like in synchronous systems only sequential circuits define the progression of states. If a state is not altered then the fault has no effect. An example can be seen in figure 2.6 , the fault will not lead to an error as long as the receiving register doesn't capture the error. This effect is comparable to the latching window masking in a synchronous pipeline.

Finally, the abstraction level of a model has to be defined. It can be from transistor level, register level or even to system level. The abstraction level defines the granularity of a circuit from the fault's perspective. In papers [11] and [18] investigations about probabilistic models in combinational and synchronous logic can be found, which influenced this thesis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Comparison of Models

This chapter deals with the problem of modeling a system where faults are injected, how to build a system which explains what is happening when a fault occurs or if it is possible to mask it and prevent propagation. It should also be possible to explain the behavior of a system in terms of probability, e.g. how many faults are mitigated and if it is possible to recover from fault injection. First we have to discuss the different styles of modeling a system and show the advantages and disadvantages of each procedure. We will build some sort of token-based description which is similar to the description used in [5], but in this thesis we don't use QDI pipelines but bundled data pipelines. Some knowledge about probability theory is needed for building the fault model and will be explained in this chapter. First the most commonly used description for faults in systems are described. The main difference between the transition and token based fault description is the abstraction level.

3.1 Token and Transition Based Fault Description

In a token based model faults are applied on a high level of abstraction, where a token is an atomic unit of the information processed in the system. In [16] a formal analysis is developed for QDI circuits to describe the behavior in the presence of SEUs, a procedure for describing how the different sorts of tokens (valid, invalid, bubble) are corrupted when faults are injected. Quasi delay insensitive circuits (QDI) are asynchronous circuits that operate correctly regardless of gate delays in the system, their delay-insensitive property makes them robust for example against delay faults.

A token is carrying information and is stored in a memory element. Tokens are defined as follows:

- Valid token: A token that has a data value which is correct in the execution process of the circuit. We have to distinguish a data token that has a normal value and data tokens which are corrupted. Those erroneous tokens have „wrong“ data values but are actually still valid.

- Empty token: This token is used by the four-phase protocol scheme when returning to zero. There is only one token which is {00}.
- The forbidden token is used for data values which are not used by the communication protocol. For example in a dual rail scheme the valid tokens are {10} and {01}, the forbidden token would be {11}.
- Bubble: A bubble indicates that a buffer is free and ready to be written with a new data value, the data value which was in this memory is already stored in the next memory stage. It still contains a copy of the previously stored and acknowledged data.

Soft errors can lead to token vanishing, token generation, bubble vanishing, bubble generation and token corruption. The major advantage of this approach is the simplicity of the model. A QDI circuit stage is composed of a computational logic block (function block) that processes the input, and a memory block (buffer) that memorizes data and implements the handshaking protocol. A three stage pipeline using dual rail, return to zero protocol, is investigated in figures 3.1a and 3.1b. When using this protocol, the asynchronous data path processes a stream of alternating valid tokens, empty tokens and bubbles (empty memory). In this basic model the data flow is controlled by two rules: Token rule: a memory may receive and store a new token valid or empty from its predecessor if and only if it has a bubble of the respective opposite type, i.e., empty or valid. Bubble rule: a memory becomes empty (bubble) if and only if its successor has received and stored the token that it was holding. The three stages are described in a row vector and in each state all previously defined soft error representations in this token based system are applied.

As described in chapter 2, for the 4 phase dual rail model, the encoding for each signal allows to express four possible states. 1,0 ,unused and NULL. The representation in the token model is:

- V1,V2,F: Valid token V1 and valid but not correct in this state (V2) and a forbidden token F (11)
- BV1,BV2,BI: Bubble (Buffer ready to be written)
- I: Empty token I (NULL) , for signal return to zero

Figure 3.1b shows the state enumeration list, it contains a representation of all reachable valid states for a 3-stage state machine. The movement of the valid token in the ring can be seen. Each state has only one successor because only one state transition rule can be applied in each stage. The state transition rules are determined according to the communication protocol. To investigate the behavior of fault injections, for each stage fault injection is performed separately. So it is possible to find faults which can be detected and also fault which can not be detected.

An example is shown in Figure 3.1d. If faults are injected in state 3, three possible outcomes related to the transition rules, are f_1 , f_2 or f_3 , which can be seen in Figure 3.1c. All transition rules can be seen in Figure 3.2 and set of rules the performs a fault injection is shown in Figure 3.3. When we continue the fault injection for example in state f_2 , the number of the next possible states is two (f_{2_1} and f_{2_2}). If a fault state can be detected, the state machine stops, but more dangerous are the cases where the injected fault keeps remaining. With the help of this model all faults can be found and it can be determined when they are reached.

In this example the state machine was simple but if it gets more complex there are far more stages to investigate which leads to rapid state explosion and that prevents us from analyzing large circuits.

Further explanations about the token based model can be found in [16].

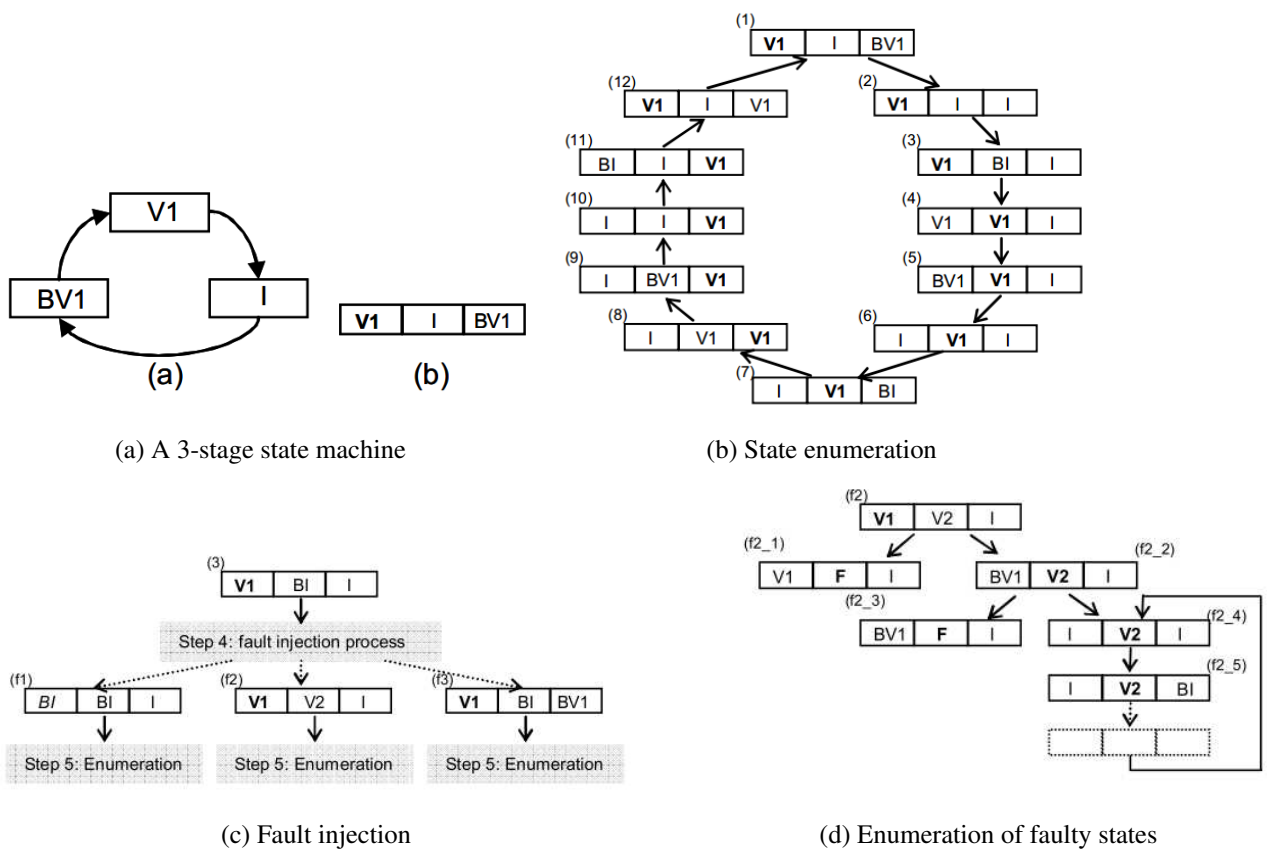


Figure 3.1: Token model [16]

Token based description efficiently explains the end effect of transient faults, such as a soft error, or SEU. This approach eases a formal investigation of the effect of SEUs but it doesn't allow a detailed investigation in the presence of transient faults as it disregards this primary cause of an error. A fault is only recognized if it is stored in the memory element of a stage and it will not appear if it is masked. The following fault description model will be on a much higher detail

$\{V1, BI\} \rightarrow \{V1, V1\}$	$\{V2, BI\} \rightarrow \{V2, V2\}$
$\{V1, V1\} \rightarrow \{BV1, V1\}$	$\{V2, V2\} \rightarrow \{BV2, V2\}$
$\{V1, V2\} \rightarrow \{BV1, V2\}, \{V1, F\}$	$\{V2, V1\} \rightarrow \{BV2, V1\}, \{V2, F\}$
$\{V1, BV1\} \rightarrow \{BV1, BV1\}$	$\{V2, BV2\} \rightarrow \{BV2, BV2\}$
$\{V1, BV2\} \rightarrow \{BV1, BV2\}$	$\{V2, BV1\} \rightarrow \{BV2, BV1\}$
$\{BV1, V2\} \rightarrow \{BV1, F\}$	$\{BV2, V1\} \rightarrow \{BV2, F\}$
$\{BV1, I\} \rightarrow \{V1, I\}$	$\{BV2, I\} \rightarrow \{V2, I\}$
$\{BV1, BI\} \rightarrow \{BV1, V1\}, \{V1, BI\}$	$\{BV2, BI\} \rightarrow \{BV2, V2\}, \{V2, BI\}$
$\{BI, V1\} \rightarrow \{I, VI\}$	$\{I, BV1\} \rightarrow \{I, VI\}$
$\{BI, V2\} \rightarrow \{I, V2\}$	$\{I, BV2\} \rightarrow \{I, V2\}$
$\{BI, BV1\} \rightarrow \{BI, I\}, \{I, BV1\}$	$\{I, I\} \rightarrow \{BI, I\}, \{I, BV1\}$
$\{BI, BV2\} \rightarrow \{BI, I\}, \{I, BV2\}$	$\{I, BI\} \rightarrow \{BI, I\}, \{I, BV2\}$

Figure 3.2: Transition rules [16]

(1) $V1 \rightarrow BI$	(4) $I \rightarrow BV2$
(2) $BI \rightarrow V2$	(5) $V1 \rightarrow F$
(3) $BV1 \rightarrow I$	

Figure 3.3: Fault injection rules [16]

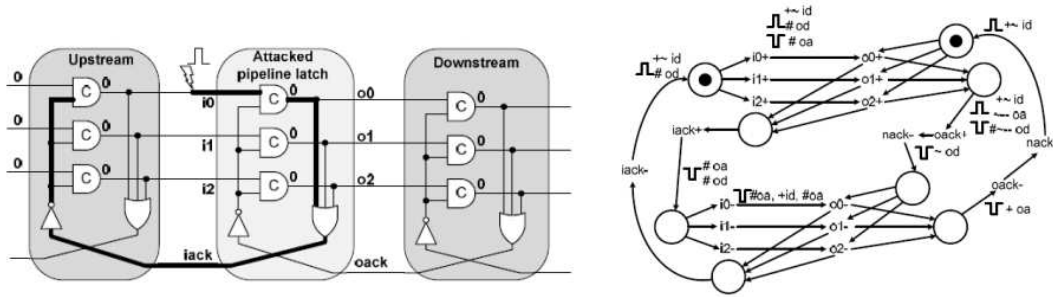
level, to investigate every injected transient fault.

In a transition based model, transient faults are directly applied to internal or external signals. This description is on a lower abstraction level than the token based fault description. Figure 3.4a shows a 1-of-3 QDI pipeline where a transient fault is applied to one input of the latch in the middle. The effects are visualized with a STG and tabulated in Figure 3.4b. This method investigates the impact of a transient fault at an arbitrary state and an arbitrary signal in the systems. It seems to be the most simple way for investigating fault injection into a system.

The possible effects of glitches in the circuit as described in [3] are:

- none: the circuit ignores the glitch.
- temporary lockout: the circuit experiences a short delay but it is still functioning.
- symbol loss: the spacing between successive symbols is lost.
- additional symbol injection: the glitch introduces a whole new 4-phase cycle.
- symbol corruption: resulting in an illegal symbol where the glitch causes a 2-of-n symbol to be created in the 1-of-n or 3-of-n symbol in the 2-of-n case.

This fault description is on a high level of detail so it involves a large computation time even at moderately complex circuits and the state space that has to be covered grows rapidly if all scenarios have to be covered.



(a) 1-of-3 four phase (RTZ) QDI pipeline.

Glitch	Location	Expected next activity	Effect possible
+	ack	New 1-of-n code	Temporary lockout
+	ack	Ack assertion	Symbol loss (race through)
+	code-wire	New 1-of-n code (same wire)	Additional symbol
+	code-wire	New 1-of-n code (different wire)	Additional symbol, Illegal symbol
+	code-wire	Ack assertion	Illegal symbol (2-of-n)
+	code-wire	Ack deassertion	Additional symbol, Illegal symbol
-	ack	code rtz	Temporary lockout
-	ack	Ack rtz	Illegal symbol (race through)
-	code-wire	code rtz (0-of-n)	Additional symbol
-	code-wire	Ack assertion	No effect
-	code-wire	Ack deassertion	Additional symbol

(b) Hazard descriptions and resulting effects.

Figure 3.4: State Transition graph analysis [3]

The transition based model seems to be more suitable to describe the behavior of circuits in the presence of transient faults compared to the token based model. The behavior of masking effects can be modeled in this model because it doesn't only show the end effect of an injected fault, which has been stored in the memory of a pipeline stage.

3.2 State Graph

A state graph describes the temporal behavior of a circuit implementation, where each node of the graph is a unique state of the system and each arc describes a transition from one state to another. Each state is defined by a binary encoded vector, which contains all input and output signals but can also contain internal signals to guarantee a unique state assignment. This is important because each state of the graph has to correspond to one unique marking in the representation as STG. Contrary to an STG, an SG expands all possible state transition sequences, thus an SG is in general more complex than its corresponding STG. An STG can be seen in 3.5, the edges are labeled with the same markings and only one transition is allowed to occur per time. If two events (a,b) should be modeled as concurrent, both orderings (a,b) and (b,a) have to

be included in this system description. For fault injection the description as state graph is used to have a picture, how many states exists for the investigated circuit because this information will be needed to write a program which measures the temporal behavior. The reason for the usage of a state graph will become more clear after the procedure for the simulation and comparing of different traces was described.

3.3 Signal Transition Graph

An STG is a variant of a Petri Net, that models the causal relations between the signal transitions of a system. A petri net is used for modeling concurrent systems. It is a directed graph with nodes and arcs where the nodes are places or transitions. Places can be marked with tokens and if a transition is enabled (called „fired“), then all inputs must have tokens. The result of firing is that the tokens from the input places are removed and added to each reachable output place. The dashed arrows in STGs indicate orderings, which must be maintained by the environment, the solid arrows represent orderings, which the circuit itself must ensure.

STGs form a restricted subclass of petri nets. Transitions are always signal transitions and simple places with only a single input and a single output are omitted. Places (arcs) represent causal relationships between signal transitions. Markings of an STG represent circuit states. Single places are omitted and we can see arcs containing places.

The properties of an STG are:

1. Input-free choice: The selection among alternatives must only be controlled by mutually exclusive inputs.
2. 1-bounded: There must never be more than one token in a place.
3. Liveness: The STG must be free from deadlocks.

To describe a speed-independent system, the circuits has to contain the following characteristics:

4. Consistent state assignment: The transitions of a signal must strictly alternate between + and - in any execution of the STG.
5. Persistency: If a signal transition is enabled it must take place, i.e. it must not be disabled by another signal transition. The STG specification of the circuit must guarantee persistency of internal signals (state variables) and output signals, whereas it is up to the environment to guarantee persistency of the input signals.

To be able to synthesize a circuit implementation the following is required:

6. CSC (complete state coding), to be synthesizable into a circuit

As we have discussed and introduced the information about state graphs and state transition graphs we can now start to develop and describe a good way how to model a suitable system in which it is easier to calculate the probability of faults. To describe system behavior it is an advantage to start with a signal flow diagram which shows for each signal in the system the temporal behavior. With it we can get to an STG and with the help of an STG it is easy to construct a state graph by expanding all possible traces and show all possible states. SGs will be the main part of the system behavior description. Figure 3.5 shows this design procedure for a C-Element.

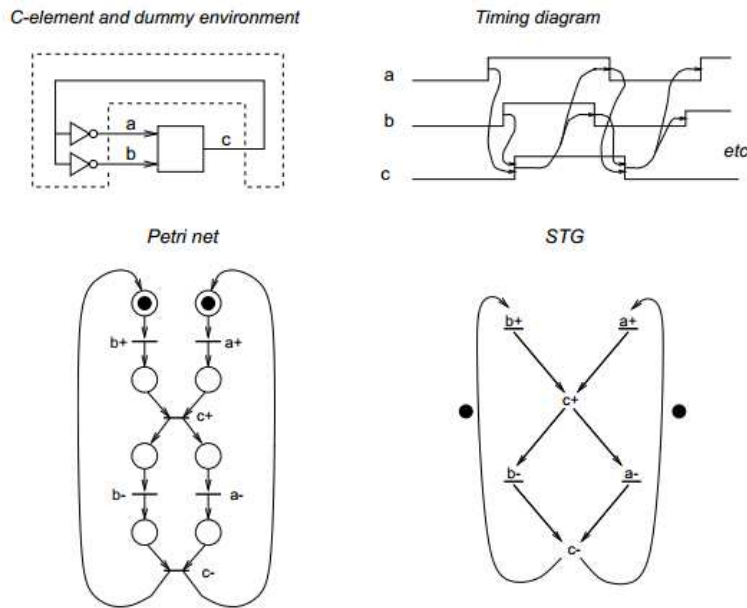


Figure 3.5: STGs [22]

For further explanations about STGs and SGs the reader can look into [22].

3.4 Trace Based Description

The trace based model as described in [5], describes concurrent computation using formal language theory.

Definition: A directed trace structure describes the behavior of a circuit C by a triple $P = \langle I; O; T \rangle$, where I is the set of input transitions, O is the set of output transitions and T is the trace set of C . The trace set $T \subseteq (I \cup O)^*$ comprises all finite-length sequences of input and output transitions. Each *trace* in the trace set T defines one possible finite sequence of such input and output transitions.

Example: A two-input Muller-C gate, one of the basic building blocks of SI and (Q)DI circuits, can be described by the following directed trace structure $\langle I, O, T \rangle$: $I = \{a, b\}$,

$O = \{c\}$, $T = \{\epsilon, a, b, ab, ba, abc, bac, abca, \dots\}$, where ϵ describes the empty trace. The output of a Muller-C gate will transition to logic 1/0 only if both inputs are logic 1/0, otherwise it will maintain its current state.

So we can conclude that it is possible to describe the behavior of a system as a trace or sequence of states. For this thesis we need to find a way to describe transient faults and to somehow calculate the sum and probability of events when transient faults can happen. In the trace based description transient faults can be described as two consecutive transitions on the same signal. It forms a single fault trace $t_x = \{xx\}$ (transient fault on signal x).

In my fault model the following assumptions are made to construct a model for transient faults:

- There are only single faults.

Effects of transient faults are distinguished by the time they appear at:

- premature firing: The fault is injected before the expected transition takes place. The injected pulse could trigger a function earlier than it should be.
- delayed firing: A fault occurs at the same time with the expected transition and the fault prolongs or delays the expected transition.
- late firing: The fault occurs after the expected transition.

In this thesis a similar description of a system is used as in [5]. Both perspectives can be transformed into one another, so the explanations used in transitions based description can also be used in the following model.

Now we want to describe the trace of a Muller C-element in a different way. An example how a trace can be transformed, to use concrete values for the signals, can be seen in the following list. The signals are a , b and c and the transition sequence is shown here:

1. $\{a\} \Leftrightarrow 000, 100$
2. $\{b\} \Leftrightarrow 000, 010$
3. $\{a, b\} \Leftrightarrow 000, 100, 110$
4. $\{b, a\} \Leftrightarrow 000, 010, 110$
5. $\{a, b, c\} \Leftrightarrow 000, 100, 110, 111$

On the left side we see the order in which the signals are changed in the states and on the right side the sequence of states is shown. Each state consists of the three signals a , b and c .

Each line describes one of the elements in the trace based model. The description is more compact and it can directly be used for simulation. It is a straightforward approach to investigate

the behavior of a circuit. The output of the simulation can directly be applied to this model and used for calculation with the formula which will be described in the following subchapter.

If we want to describe a fault in this systems we have to find out if a transient is masked or not, which we call a „bad“ or „not bad“ signal. It is a bad signal if a transient fault takes effect and the behavior of the circuits differs from the fault free behavior of the system. If the transient doesn't take any effect on the system than it is a „not bad“ signal. This definition will be used later when we investigate the behavior of bundled data pipeline in the presence of transient faults.

For example when we look at a Muller C-gate which has two input signals a and b and one output signal c then a transient fault on c is always a bad signal. The signals a or b are only bad if this circuit is sensitive, which means that a transient fault on an input signal is propagated to the output. It depends on the current value of the signal c but lets say that signal c is 0, then if a is 0 and b is 1, a transient fault injection on a, will not be masked and will change the circuit's behavior. If we have the situation that c is 0 and the two other signals a and b are 0 then a transient fault injection on a will not propagate, it will be masked. So we see that the same signal is not always good or bad and it also depends on the temporal behavior, the time when the transient fault is injected. In a pipeline where the temporal behavior of the stages is described by a state graph, in each state of the state graph we can distinguish if a signal is good or bad, by comparing a correct trace to a trace with an injected fault. The concrete procedure how to do this and what we have to consider when making this simulation will be discussed in the next chapter, when we know more about the used pipeline architectures.

A possibility to locate all possible fault locations was described in [5] with the help of a simple fault set. A simple fault set T_y^{xx} describes all traces that are obtained when the single fault trace t_{xx} is merged into a code phase trace t_y : $T_y^{xx} = \{t | t = t_{xx} \cup t_y\}$. The size of T_y^{xx} is calculated as follows: There are n signals that build the trace. A first faulty transition can be placed before any of the n expected transitions as after the last one, which leaves n + 1 possible locations. The second must be placed after the first one. If the first fault transition is set to the beginning of the trace, there are n + 1 possible locations for the second one. If the first location is set as second transition, there exist only n possible locations for the second one, etc. In case the first fault is placed after the last expected transition, only one possibility remains to place the second transition.

Eventually, all different configurations are summed, which yields the formula.

$$|T_y^{xx}| = (n + 1) + n + (n - 1) + \dots + 1 = \frac{(n + 1)(n + 2)}{2} \quad (3.1)$$

Let's assume we have a Muller-C element with the input vector $I = \langle a, b \rangle = \langle 1, 2 \rangle$ and the input trace $t = \{1\}$. Now a transient fault on signal 2 is added, which will extend t to the fault set $T^{22} = \{221, 212, 122\}$. The set contains 3 traces, which could also be found by applying formula 3.1: $\frac{(2)(3)}{2} = 3$

In [5] tokens are used to describe the set of transitions, we also want to reuse these definitions. The notation is T (# of additional excited rails):

- $T(-1)$: one expected transition inhibited. That class will be inherently masked by all QDI circuits as it prevents the code phase completion. However, it requires redundant gates in the function to inhibit an output transition.
- $T(+0)$: the expected number of rails is excited. Although the class describes legal tokens, it also contains token errors if unexpected rails are excited.
- $T(+1)$: one additional rail is excited. That class is not delay-insensitive anymore. Although unexpected rail transitions are involved, not all members of this class lead to a token error.

We can use token classes as described in [5] but in this thesis $T(+1)$ means that there is a signal change, which is not contained in the normal trace, while $T(-1)$ describes the case when a transition is inhibited and $T(0)$ describes changes which are also in the normal trace (transient fault is injected on a signal where a state transition will happen anyhow). This way of describing faults is more general. The Token description in [5] was used for a dual rail model and not for a bundled data protocol where the code words are encoded on one signal line and not on two signal lines. For example if two signal lines are used for code words then $T(0)$ means that there are not more rails excited than the expected ones. The token classes describe all types of single bit faults in a single-bit QDI signal. These definitions can also be used when describing transient fault injection in bundled data pipelines. As long as the fault only effects one signal line, the token class description is exhaustive. The potential effects of transient faults in this token class model can be seen in Figure 3.6.

When the Token model in [5] is compared to the model as it is used in this thesis, we can conclude the following:

- $T_c(+1)$: There is an unexpected signal change because of the transient fault.
- $T_c(-1)$: The signal change will happen later than expected. The transient fault happens at the same time when the normal state transition would have happened. The signal is changed to the opposite value, so the real signal change, can only be seen after the transient fault.
- $T_c(0)$: There will be no additional signal change and we can not see any difference. The transient fault doesn't affect the circuit, it doesn't change the signal to a different value.

As we know the behavior of a Muller-C element, its output changes only if both inputs have the same value. Whether a fault will be propagated depends on the input state and on the current state of the circuits output when the transient fault is injected. For example if signal a is high and when a fault is injected on signal b , it doesn't take effect and doesn't change the output c . It can also happen the other way around. All possible situations where an injected fault propagates to the output c are also shown in 3.7. Another example: input a and output c are low and input b is high. If a transient fault is injected at a , the output c of the Muller-C element will be high and

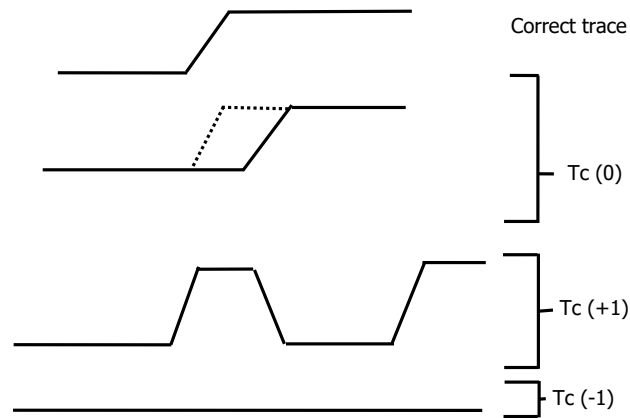


Figure 3.6: Token classes

will hold this state also after recovery of the transient fault.

Based on the description with the help of the state trace and the knowledge of the delays and bad signals in each state, we can build a probability model which describes the system's behavior in the presence of transient faults. If we inject n transient faults one at a time, we can get to a value which describes how robust the circuit is, how many transient faults will be masked. As explained before masking describes the effect when transient faults don't have any effect on the outcome of the circuit's functions.

We can find the bad signals by simulation or going through all possible traces in the circuit. For simulation we can start to inject in each state on each signal line a fault and then compare it to the fault free trace of the circuit. There are three possibilities:

1. correct: The circuit's trace doesn't differ from the correct one, except the two transitions, where the fault was injected. Masking was successful.
2. bad: The circuit's behavior is different at the time of fault injection and after the fault disappears.
3. stabilized: The circuit's trace will become the normal trace again after some time and looks like there was no transient fault injected.

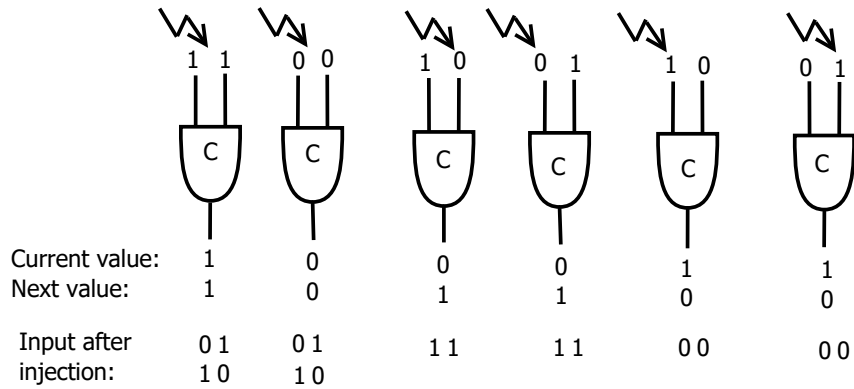


Figure 3.7: Muller-C element , Fault injection

3.5 Probability Model

We will now develop a probability model to describe the likelihood of errors leading to failures. In every state there is the chance, that an error is masked by affecting a signal whose change has no effect on the system behavior.

In every state there is a fixed number of signals which we will refer to as „bad signals“ which are sensitive to errors. If a state is affected by exactly one error the probability of the error leading to a failure is given by formula:

$$P(A|C_i) = \frac{b_i}{B_i} \tag{3.2}$$

- b_i . . . Number of bad signals in state i
- B_i . . . Number of signals in state i
- A . . . A failure occurs
- C_i . . . an error occurs in state i

If we assume that errors are evenly distributed over time, and we inject exactly one error into the system, the probability of the error being injected into state i is :

$$P(C_i) = \frac{t_i}{T} \quad (3.3)$$

- t_i ... duration of state i
- T ... duration of pipeline cycle

Thus we get for the probability $P(A)$ that a single error which is injected into the system leads to a failure:

$$f = P(A) = \sum_{i=1}^N P(A|C_i) \cdot P(C_i) = \sum_{i=1}^N \frac{b_i}{B_i} \cdot \frac{t_i}{T} \quad (3.4)$$

- N ... Number of pipeline states

If we further assume that all cycles in a sequence of n cycles are statistically independent, the number of failures X is a binomial random variable with success probability f as calculated above.

This leads to:

$$E = n \cdot f \quad (3.5)$$

$$\sigma = \sqrt{n \cdot f \cdot (1 - f)} \quad (3.6)$$

The formula can also be written in a different way which shows the mean value:

$$E(X) = \sum_{i=1}^N X \cdot \frac{b_i \cdot t_i}{B_i \cdot T} \quad (3.7)$$



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Fault Injection Setup

This chapter deals with the structure of 4- and 2-phase bundled data pipelines and their modeling when we use them as target pipelines for fault injection. We want to find formulas to describe the temporal behavior, to estimate how long each state in the bundled data pipeline takes. With the help of this information we can use the probability formula explained in the previous chapter, to derive a fault probability model for concrete examples.

4.1 4-phase bundled data

To investigate the masking effect of an asynchronous pipeline structure, we start with the simulation of a 4-phase bundled data pipeline. To understand the behavior and to be able to code this circuit in VHDL, we have to look at the 4-phase latch control circuit in more detail. We will not introduce a logic transfer function between two stages of the pipeline, to make it easier for simulation and it is sufficient to recognize the causes of faults. [8]

The variant of a 4-phase control circuit which was used for this diploma thesis is presented in [8]. Only the simplest controller has been investigated, basically a Muller pipeline. To understand the behavior in more detail we have to start with a STG (state transition graph) of its input and output signals and to describe one stage of the control logic, as shown in Figures 4.1a and 4.1b.

The control logic for a micro pipeline register must support the handshake protocol on both its input and output ports. It is therefore defined in terms of input and output requests (R_{in} , R_{out}), acknowledge signals (A_{in} , A_{out}), and its internal latching function.

An event on R_{in} signals the availability of new data, and the register issues an event on A_{in} , to indicate to the source of the data that it has been captured and the data at the input can now be changed. The control logic also issues an event on R_{out} to indicate, that the latch's output is now valid and will be held stable until an event on A_{out} signals that it has been accepted by the next stage in the asynchronous pipeline. In addition to the handshake signals, the latch circuit also has an internal control signal (L_t) which causes the data latches to be open (transparent) when low and closed when high.

In four-phase signaling it can be chosen, which edge of each handshake signal is active and takes the place of the event while the other edge is inactive and is part of the „recovery“ phase, during which the control circuit prepares for the next cycle. We choose the rising edge to be active in every case.

It can be seen in Figure 4.1a that the circuit has to follow some behavioral properties:

- R_{out+} indicates the availability of output data and must therefore follow R_{in+} .
- When input data is available (signaled by R_{in+}) the latch closes (L_t+) and stores the new value. Later the input is acknowledged (A_{out+}) and the latch opens again (L_t-).
- The latch must alternate between open and closed.

It can be seen that Figure 4.1b is an implementation of the STG (4.1a). While this circuit operates correctly, it has some undesirable properties. Most notable of these is, when several such latches are formed into a FIFO, at most alternate stages can be occupied at any time. This is because A_{out} must be low (and therefore the next latch empty) before L_t can go high (and this latch becomes full).

Certain assumptions are built into this circuit construction. It is expected that there will be several bits of latched data, so L_t must have reasonable drive buffering. The path from R_{in} to A_{in} reflects the need for the latch to close before input data may be removed, and to consider this the buffer delay is built into this path. The buffer delay is not built into the path between R_{in} and R_{out} , since there is no need for the latch to close before R_{out} is signaled, as long as the data has propagated through the latches. The C-gate delay must be no shorter than the latch data-in to data-out delay for the correct functioning of the circuit.

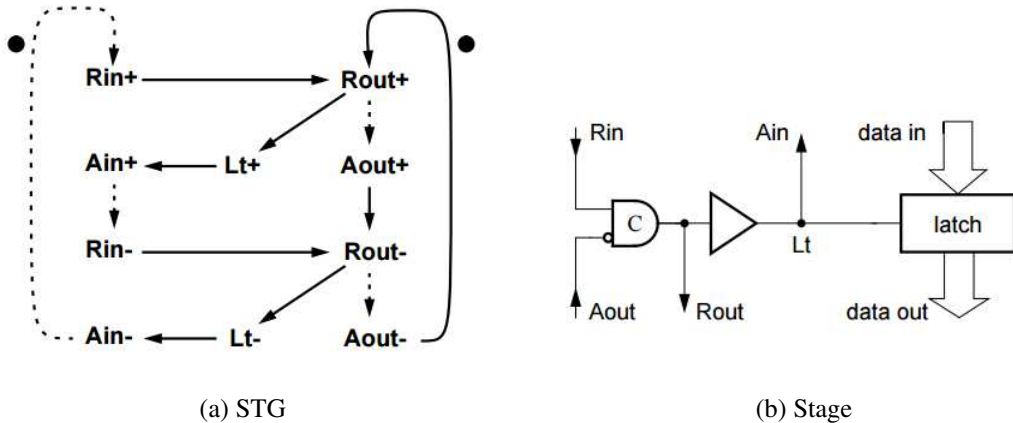


Figure 4.1: 4-phase bundled data pipeline. [22]

After discussing the properties and behavior of one stage we assemble three stages to a FIFO pipeline. With the help of Figure 4.2, the sequential steps of the control circuits can be investigated. We can see a 4-phase pipeline structure with three stages, which is used for the fault

injection experiment. It is built out of 4 logic components which are connected by wires. The main components are:

- Muller C-element: logic component which remembers its last state, it only changes its output if both inputs have the same logic value. It has a similar behavior as an and-gate.
- Delay element Δ : To control the relative timing in the path delays after the fork right at at output of the C-element, for example to assure that an event on A_{in} happens after an event on R_{out} . Moving A_{in} to before R_{in} can be accomplished by a negative delay Δ .
- Delay element $D2$: This delay controls the relative timing of the fork towards A_{in} and L_t .
- Latch: A D-latch is used as memory element.

The main difference to the circuit which is described in [8] is that there is a delay element $D2$ on the path from signal A_{in} to L_t . This delay element was introduced to make a distinction between the signals A_{in} and L_t in the trace-based model (the trace which is produced by the simulation).

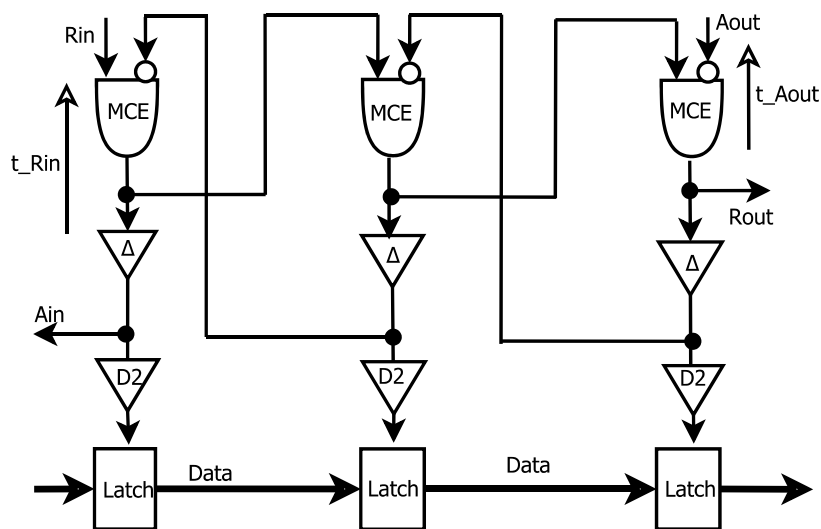


Figure 4.2: 3-stage, 4-phase-pipeline

The next step is now to get from the circuit and the STG description of the circuit to the state graph, to show in which states a specific stage (we consider the stage in the middle) can be. It can be seen in figure 4.3 that the upper and lower part of figure is in some way symmetric. There is a phase where new data („new value“) is available and the part where the return to zero takes place. Those two phases always happen alternately. Every state in the SG shows the current state (signal level) of the five signals in each stage. The five signals which are already described above are R_{in} , R_{out} , A_{in} , A_{out} and L_t .

The two main observations are:

- The diagram forms a cycle with some cycle time.
- The path which is chosen in different cycles need not be the same. It depends on the input to output behavior of the pipeline.

The following is an explanation of what happens in the different conditions. It can be seen in the SG that every condition happens twice and symmetrically in the diagram. Which condition is chosen, depends on the circuit parameters (e.g. delay time). How parameters have to be chosen to get to a specific condition will be shown later.

To keep the formulas simple, each kind of circuit component has been assigned the same delay value. For example all Muller C-elements used in the circuit have the same gate delay time.

First we will look at the time difference between transition of R_{in} and the transition of A_{out} , which can be adjusted by the environment which influences if the token is consumed faster from the output or written to the input of the 4-phase bundled data pipeline. The two possible paths are available from 12 to 1 and 5 to 8:

- 12 - 13 - 1: $R_{in} \uparrow$ is faster than $A_{out} \downarrow$, $A \downarrow \rightarrow R \uparrow > 0$
- 12 - 0 - 1: $A_{out} \downarrow$ is faster than $R_{in} \uparrow$, $R \uparrow \rightarrow A \downarrow > 0$
- 5 - 6 - 8: $A_{out} \uparrow$ is faster than $R_{in} \downarrow$, $R \downarrow \rightarrow A \uparrow > 0$
- 5 - 7 - 8: $R_{in} \downarrow$ is faster than $A_{out} \uparrow$, $A \uparrow \rightarrow R \downarrow > 0$

The next conditional ways can be seen at the paths 2 to 5 and the symmetric path 9 to 12. The difference between those two paths exists because of the delay parameter $D2$, which can be seen in Figure 4.2. If L_t changes faster than A_{in} , the circuit has only one possible path because for positive delay $D2$, A_{in} always happens before L_t . For completeness of the state description of the circuit, the paths 2-3-5 and 9-11-12 were not deleted they will be needed later for the trace based description, to see if a trace behaves according to the rules. The explanations for the different paths are shown below:

- 2 - 3 - 5: L_t rises \uparrow faster than A_{in}

3. $T_1B_2T_3$

The requirements for a pipeline with iterative computation are: 1. at least one bubble and 2. at least one value and one empty token. It is a subset of all 8 states of an elastic pipeline, in which no requirement exists.

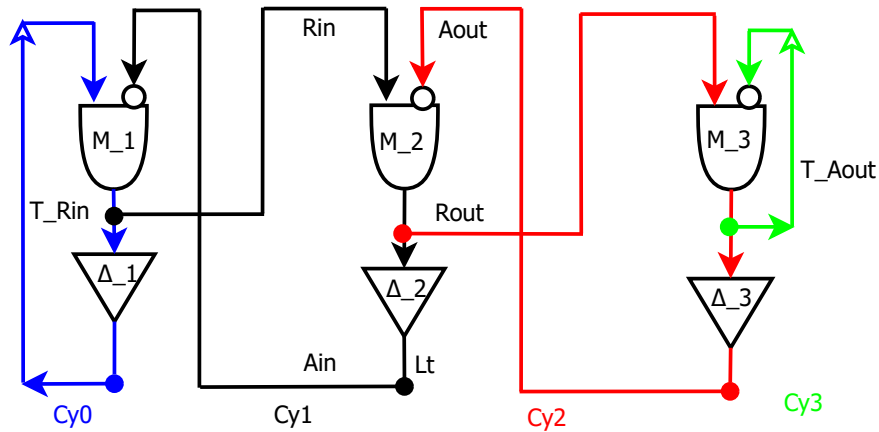


Figure 4.4: 4-phase-pipeline cycles

To be able to express the temporal behavior of this circuit in a more formal way, we have to divide the circuit into 4 cycles. The time for each cycle is the summation of the circuits component delay times.

1. $Cy0 = T_Rin + M_1 + \Delta_1$
2. $Cy1 = M_1 + M_2 + \Delta_2$
3. $Cy2 = M_2 + M_3 + \Delta_3$
4. $Cy3 = M_3 + T_Aout$

With the help of those explanations and descriptions of cycles we conclude the following inequalities:

$$\begin{aligned}
 & T_1 T_2 B_3 : \\
 \text{condition } T_1 \text{ and } T_2 : & C_{y1} > C_{y0} \\
 \text{condition } B_3 : & C_{y3} < C_{y2}
 \end{aligned}$$

$$\begin{aligned}
 & B_1 T_2 T_3 : \\
 \text{condition } T_2 \text{ and } T_3 : & C_{y3} > C_{y2} \\
 \text{condition } B_1 : & C_{y1} < C_{y0}
 \end{aligned}$$

$$\begin{aligned}
 & T_1 B_2 T_3 : \\
 \text{condition } T_1 \text{ and } T_3 : & C_{y0} < C_{y1} \\
 \text{condition } B_2 : & C_{y3} > C_{y2}
 \end{aligned}$$

After those estimations about input to output timing behavior we have to find the exact duration for each state in which the circuit can be. The best way would be to calculate the duration of each state by a formula with the help of the circuit parameters, e.g. delay of the Muller gate or delay of the delay line Δ . In the simulation some typical duration values for the circuit parameters were chosen and can be seen in the following list:

- Delay Muller C-gate (M) ... 3 ns.
- Delay of the Delay line (Δ) ... 13 ns.
- Delay for closing the latch (T_{D2}) ... 1 ns
- Delay at output T_{Aout} ... depends on environment.
- Delay at input T_{Rin} ... depends on environment.

In Table 4.1 all formulas for each state are noted. If the delay of each circuit parameter in the circuit is known, the duration for each state can be calculated and no simulation is necessary. As we can see there are four different columns which show the four different paths of the SG, which were explained before.

Condition 1 (**COND 1**) means L_t changes faster than A_{in} and Condition 2 (**COND 2**) means A_{out} is faster than R_{in} . If a condition is true or not is shown by N or Y .

Table 4.1: Duration of the stages of a 4-phase bundled data pipeline

COND 1 ($L_t < A_{in}$)	Y		N	
COND 2 ($A_{out} < R_{in}$)	Y	N	Y	N
0	$T_{Rin} - M$	-	$T_{Rin} - M$	-
1	M	M	M	M
2	Δ	Δ	Δ	Δ
3	T_{D2}	T_{D2}	-	-
4	-	-	T_{D2}	T_{D2}
5	$M - T_{D2}$	$M - T_{D2}$	$M - T_{D2}$	$M - T_{D2}$
6	$T_{Rin} - M$	-	$T_{Rin} - M$	-
7	-	$T_{Aout} - M - \Delta$	-	$T_{Aout} - M - \Delta$
8	M	M	M	M
9	Δ	Δ	Δ	Δ
10	-	-	T_{D2}	T_{D2}
11	T_{D2}	T_{D2}	-	-
12	$M - T_{D2}$	$M - T_{D2}$	$M - T_{D2}$	$M - T_{D2}$
13	-	$T_{Aout} - M - \Delta$	-	$T_{Aout} - M - \Delta$

Now a more detailed explanation for the formulas in Table 4.1 will be given and we start with the first column where **COND 1** and **COND 2** are true. Each entry shows the duration of each state in the state graph with the used circuit elements. Some of the delays are given by the circuits requirements to be a correctly functioning pipeline. For example if we look at the duration of the states 1 or 8, it is clear that the time between the change of the signals is the gate delay of the Muller C-element. Another example is the time between the signal change of R_{out} and A_{in} , the time, which is given by the delay element Δ . More interesting is the time between the states 0 and 1 and 12 and 0 (The duration of the mirrored states are also calculated in the same way).

Because of symmetry we can find a formula which expresses the input behaviour. Later we want to find out how we have to adjust the two parameters T_{Rin} and T_{Aout} to define if input is faster than output or the opposite and with this formula it is possible to see which condition in the table was chosen.

- $M_2 + M_3 + \Delta_3 = Cy_2$.
- $M_1 + M_2 + \Delta_2 = Cy_1$
- $M_1 + \Delta_1 + T_{Rin} = Cy_0$
- $Cy_0 - Cy_1 = T_{Rin} - M$

The formulas to calculate the duration between the states 12 and 0 are:

- $M_2 + M_3 + \Delta_3 = Cy_2$.

- $M2 + T_{D2} + \Delta_2 = x$
- $Cy_2 - x = M - T_{D2}$

x is the delay of the second stage.

Now we will look at the case where **Cond1** is true and **Cond2** is false, the other two possibilities for the conditions are mirrored and can be found by the same formulas. The reason is that there is a delay element between those two signals but for completeness it is shown in table 4.1.

For the duration between the states 13 and 1 we get the following formulas:

- $M_2 + M_3 + \Delta_3 = Cy_2.$
- $M_3 + T_{Aout} = Cy_3$
- $Cy_3 - Cy_2 = T_{Aout} - M - \Delta_3$

The input T_{Rin} to output T_{Aout} delay can be explained in more detail in the following way: For the correct functionality of the circuit T_{Rin} must have the duration time which is at least M , it behaves like a predecessor stage of stage 1. For T_{Aout} the behavior is similar but a successor stage of stage 3 is simulated, thus the duration time must be at least $M + \Delta$. The formula which describes if input is faster than output is $(T_{Rin} - M) - (T_{Aout} - M - \Delta)$: if the result is positive then the input is faster than the output.

- if $(T_{Rin} - M) - (T_{Aout} - M - \Delta) > 0$ then $A \leftarrow R < R \leftarrow A$ (COND2 is false)
- if $(T_{Aout} - M - \Delta) - (T_{Rin} - M) > 0$ then $A \leftarrow R > R \leftarrow A$ (COND2 is true)

4.2 2-phase bundled data

For fault injection the second experiment was done with a 2-phase bundled data pipeline. Before describing the fault injection procedure, we start by describing the main blocks of the pipeline, which was used for this thesis. The circuit, which was used in this thesis, can also be found in [17].

The main building blocks are:

- delay element: This delay element is not used in [17] but we will explain this later.
- xnor-gate: It has two inputs and its output is 1, when both inputs have the same value, otherwise its output is 0.
- a level sensitive latch: It is transparent when the enable signal Lt is 1, otherwise it is not transparent and saves the last forwarded value. It is not driven by a clock-signal, but it immediately opens and closes the forwarding gate when the enable signal Lt changes.

In contrast to the Muller control, the feedback loop in the circuit is broken by a state-holding element - the latch.

Figure 4.5 shows the architecture of a 2-phase bundled data pipeline with 3 stages. For simplicity we omitted the computation functions between the stages to describe the main functionality of the control circuit. We will use the same signal names as for the 4-phase bundled data pipeline.

As described in chapter 2 the 2-phase bundled data pipeline works with a handshaking protocol, which is divided into 2 phases. If the sender wants to send new data, it produces a transition on the request line. In our circuit this line is called „Rin“. The signal value of „Rin“ is only forwarded if the latch is transparent, this only happens if the signal Lt is true (xnor gate's output has to be 1). Both inputs of the xnor gate had the same value but now the signal from Rin changes one of the inputs of the xnor gate and the output of the xnor gate becomes 0. The latch will not be transparent anymore, therefore it closes. The signal value of „Lt“ is also used for the Latch „D-Latch“, which stores the data for each pipeline stage. As data is already stored „Rin“ is forwarded to the next pipeline stage as „Rout“, to signal the next stage that new data is available at input.

In the second stage of the 2-phase handshaking protocol the receiver absorbs the codeword and changes the signal state of the acknowledge line, in our example „Ain“. Compared to the circuit design in [17], a delay element for „ack“ was added to give the circuit enough time to absorb the new data from the sender. If the signal change of „Aout“ is received at the sender's xnor gate both inputs have the same value and the communication cycle starts again at phase one.

As we have compared the behavior of this circuit to the 2-phase protocol description and see that it behaves correctly, we have to look closer at the stage in the middle of the pipeline, as we have done in the sub chapter before, when we discussed the behavior of the 3-stage 4-phase bundled data pipeline.

In Figure 4.6 we see the state graph of the 2-phase bundled data pipeline, which describes the communication cycle states of the second stage in the 3-stage pipeline. As this pipeline protocol doesn't forward information simply by the state of the signals –“1“ and “0“– but it is driven by signal transitions, it can be concluded that it has more states and more possible paths in the state graph.

Now the different paths are described in more detail. First the path at 2 to 5 is described, it behaves the same as the mirrored path from 11 to 14:

- 2 - 3 - 5: $Lt \downarrow$ faster than $Ain \uparrow$ (xnor faster than T_D)
- 2 - 4 - 5: $Ain \uparrow$ faster than $Lt \downarrow$

There are also different paths between 5 to 10, which can be distinguished by the delay of the circuit elements. Here the important circuit parameters are $Aout$, Rin and Lt . We see that there is only one possible trace when the state 8 is reached but at state 6 there are two possibilities. In state 8 there is only one possibility because of the circuit definition. The output Lt of the xnor element can only change if both inputs $Aout$ and $Rout$ have the same value, so it is not possible that Lt changes in the transition from state 8 to 9.

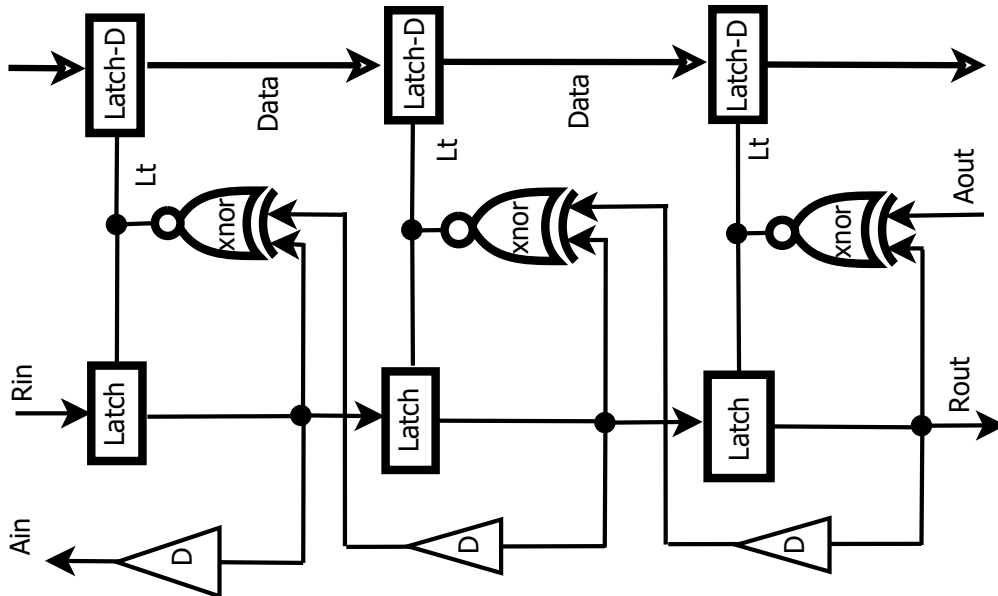


Figure 4.5: 3-stage, 2-phase-pipeline

- 5 - 6 - 7 - 10: $Aout \uparrow$ faster than $Rin \downarrow$ and $Lt \uparrow$ faster than $Rin \downarrow$
- 5 - 8 - 9 - 10: $Rin \downarrow$ faster than $Aout \uparrow$
- 5 - 6 - 9 - 10: $Aout \uparrow$ faster than $Rin \downarrow$ and $Rin \downarrow$ faster than $Lt \uparrow$

As we have described the different cycle formulas of the 4-phase bundled data pipeline state graph, we will do the same for the 2-phase bundled data pipeline. In the following list we describe the 4 cycles for the three stage pipeline. Now we also find the formula for describing T_{Rin} and T_{Aout} - the input and output delays. With the help of those two parameters we can control whether the pipeline is faster at input or faster at output.

We also describe all cycles shown in 4.7 as formulas. T_{Rin} and T_{Aout} are the user controlled parameters to try different input and output scenarios and are the only parameters which will be changed during fault injection simulation, all other parameters like the Muller C-element gate delay will stay the same. With the help of those formulas it is possible to explain the input to output behaviour of the pipeline:

Rin ,Ain ,Rout ,Aout ,Lt

-----2-phase-----

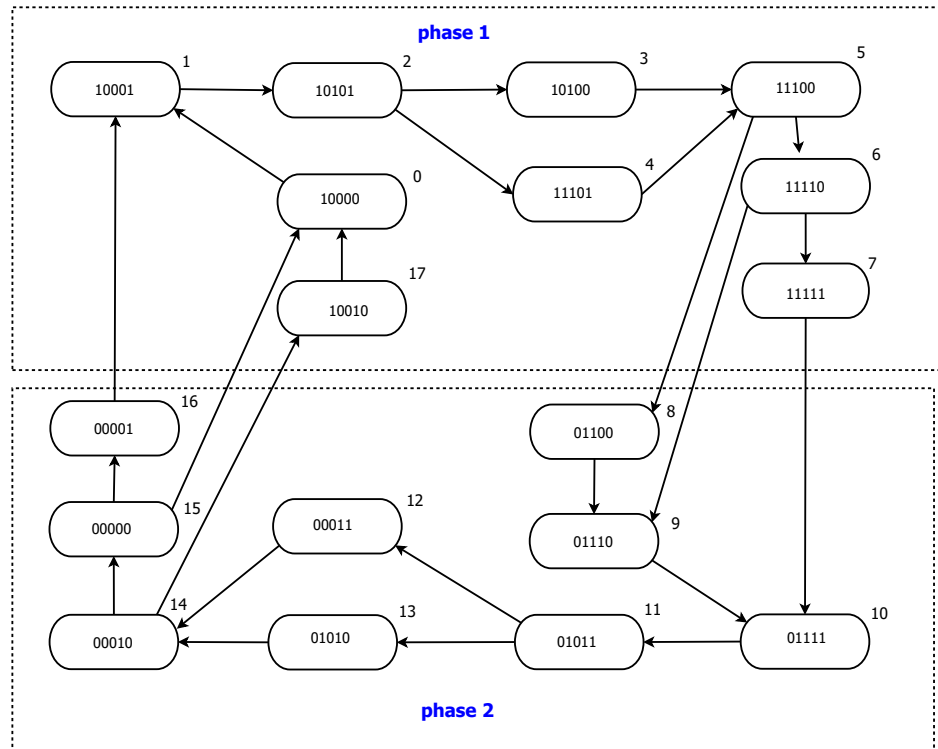


Figure 4.6: SG of 2-phase-pipeline stage

1. $T_{Rin} > T_{xnor} + T_{D2}$
2. $Cy0 = T_{xnor} + T_{D2} + \Delta + T_{Rin}$
3. $Cy1 = 2 \cdot T_{xnor} + 2 \cdot T_{D2} + \Delta$
4. $Cy2 = 2 \cdot T_{xnor} + 2 \cdot T_{D2} + \Delta$
5. $Cy3 = T_{xnor} + T_{D2} + \Delta + T_{Aout}$
6. $T_{Aout} > T_{xnor} + T_{D2}$

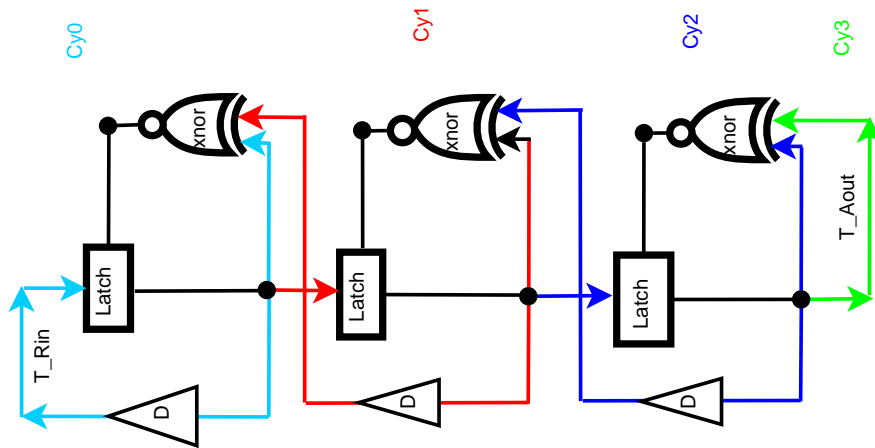


Figure 4.7: 2-phase-pipeline cycles

We described the pipeline with cycles, now we start to look at duration for each stage in the pipeline a little bit closer. The three conditions which show which path it will be taken, are described in the following list. The delay parameter Lt for the xnor element is given by the circuit but the delay of the output and input signals of the pipeline is given by the environment and can be changed for the experiments.

1. COND1: $Lt \uparrow$ is faster than $Ain \downarrow$
2. COND2: $Rin \uparrow$ is faster than $Aout \downarrow$
3. COND3: $Rin \downarrow$ is faster than $Lt \uparrow$

Two paths of the circuit are not possible, those are the following two traces starting at stage 5:

1. COND1 is true, COND2 is true and COND3 is false
2. COND1 is false, COND2 is true and COND3 is false

As we see that Rin changes faster than $Aout$, the successor stage of 5 is 8. In stage 8 the signals $Aout$ and $Rout$ are different, therefore the signal value of Lt cannot change in the successor stage, there is only one path possible. This is the reason why above two traces are not possible.

The formulas of the variables used in table 4.2 are:

1. $T_{delay} : \Delta - (T_{xnor} + T_{D2})$
2. $T_{z1} : T_{D2}$
3. $T_{y1} : T_{Rin} - (T_{D2} + T_{xnor})$
4. $T_{z2} : T_{Aout} - (\Delta + T_{xnor})$

Table 4.2: Duration of the stages of a 2-phase bundled data pipeline

COND1	Y				N			
COND2	Y		N		Y		N	
COND3	Y	N	Y	N	Y	N	Y	N
0	T_{xnor}	-	T_{xnor}	-	T_{xnor}	-	T_{xnor}	-
1	T_{D2}	-	T_{D2}	T_{D2}	T_{D2}	-	T_{D2}	T_{D2}
2	T_{delay}	-	T_{delay}	T_{delay}	T_{xnor}	-	T_{xnor}	T_{xnor}
3	-	-	-	-	T_{delay}	-	T_{delay}	T_{delay}
4	T_{xnor}	-	T_{xnor}	T_{xnor}	-	-	-	-
5	T_{y2}	-	T_{z1}	T_{z1}	T_{y2}	-	T_{z1}	T_{z1}
6	-	-	T_{y1}	T_{xnor}	-	-	T_{y1}	T_{xnor}
7	-	-	-	T_{y1}	-	-	-	T_{y1}
8	T_{z2}	-	-	-	T_{z2}	-	-	-
9	T_{xnor}	-	T_{xnor}	-	T_{xnor}	-	T_{xnor}	-
10	T_{D2}	-	T_{D2}	T_{D2}	T_{D2}	-	T_{D2}	T_{D2}
11	T_{delay}	-	T_{delay}	T_{delay}	T_{xnor}	-	T_{xnor}	T_{xnor}
12	T_{xnor}	-	T_{xnor}	T_{xnor}	-	-	-	-
13	-	-	-	-	T_{delay}	-	T_{delay}	T_{delay}
14	T_{y2}	-	T_{z1}	T_{z1}	T_{y2}	-	T_{z1}	T_{z1}
15	-	-	T_{y1}	T_{xnor}	-	-	T_{y1}	T_{xnor}
16	-	-	-	T_{y1}	-	-	-	T_{y1}
17	T_{z2}	-	-	-	T_{z2}	-	-	-

5. $T_{y2} : T_{D2} + T_{xnor}$

Now the input to output delay is explained in more detail, we will use the same procedure as for the 4-phase bundled data pipeline to find a formula which describes the correct behavior of input to output. For the correct behavior of the circuit the delay of the predecessor stage of stage 1 has to be larger than $T_{xnor} + T_{D2}$ and for the successor stage of stage 3 we conclude that the duration must be larger than $T_{xnor} + T_{D2} + \Delta$. In the following chapter we will use this formula and try different values for T_{Rin} and T_{Aout} , to control the delay between input to output and find out how much influence the different setting have for the fault masking effect of the pipeline circuits.

- if $(T_y - T_{xnor} - T_{D2}) - (T_{Rin} - T_{xnor} - T_{D2} - \Delta) > 0$ then $A \leftarrow R < R \leftarrow A$ (COND2 is true)
- if $(T_{Rin} - T_{xnor} - T_{D2} - \Delta) - (t_y - T_{xnor} - T_{D2}) > 0$ then $A \leftarrow R > R \leftarrow A$ (COND2 is false)

It is also possible to control COND1 and COND3 and change the behaviour, but this would not describe a functioning pipeline. For example the output of the latch would not be stable for the next stage if COND3 is true.

Simulation

In the following chapter we will discuss the building blocks of the simulator which will be used to quantitatively verify the results which we got in the last chapter per theory. We want to show here that the formulas about the timing analysis and error probability can be proven by fault simulation. First we will discuss what is important for fault simulation and show some similar programs but we will recognize that the most important parts are the same.

Fault simulation is used as a process of measuring the quality of the probability model, which was developed to measure the masking effects. The following equation shows the quality of the developed model: $T = \frac{\text{calculated masking probability}}{\text{simulated masking probability}}$. This formula shows the discrepancy between the simulation and using the formulas, which will be shown later in more detail.

Some differences in the approach result from differences in basic assumptions about the circuit being evaluated. When simplifying assumptions are made, it is possible to take advantage of those assumptions to produce a faster product, but one that will not function correctly when they do not hold. Hence the user must understand the capabilities and limitations of the tool that he or she chooses to use in order to obtain maximum benefit from it.

The paper used as guidance for writing the simulator is [19], and it shows how to develop a suitable simulator for QDI circuits and what are the advantages and disadvantages of different designs.

The simulator which was modified according to the requirements, contains the following main parts:

- Fault injector: This part is common in all fault simulators and generally has the function of applying the fault to certain parts of the circuit. Where it is injected, is identified by the Fault Place Identify, which is a subcomponent of the Fault injector. There are two different procedures to decide where to inject a fault, per random process or in a methodical way and both have their advantages and disadvantages. They will be used in different steps of the simulation process.
- DUT: Device under test is the component which has to be tested.

- Monitor: Some output which shows the difference between a correct simulation trace and a trace where faults have been injected and were not masked.

The simulator which will be used in this thesis is slightly different to the one presented in [19]. A random delay generator can be used to increase the ability of asynchronous circuit testing, to simulate the circuit with various delays. In [19] the simulator was designed for QDI circuits, according to the features of these circuits, random delays must be within certain boundaries. The simulator designed for this thesis does not include a Random Delay Generator, the delays of the circuit parts are fixed, only input to output delay can be changed. Of course the pipeline architecture can be changed in VHDL to have different gate delays of the elements, but those parameters will not be changed dynamically. The assertion for our simulator is that the trace of a correct test run without fault injection has the same sequence as a test run with injected faults. In Figure 5.1, a description of the simulator on a high abstraction level is given.

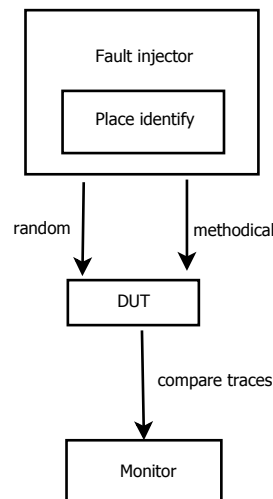


Figure 5.1: Fault simulator

Setting up fault injection for synchronous target is a well- researched topic. In contrast, little is known about the specific issues that arise in context with fault injection into an asynchronous target [6]. The analysis can be structured in two major topics. 1) the actual fault injection process including trigger and 2) the subsequent observation of the fault effect on the device under test, covering capturing and comparison of data. In both topics, synchronization is an issue of interest. The usual way to do this is by comparison with a well-known good reference, in other words the

Table 5.1: Calculation of state duration

State	<i>trace1</i>	<i>timestamp</i>	duration
0	0000	100	200
1	0010	300	

device is operated in lock step with a fault free duplicate, that serves as a Golden Node. Due to the adaptive timing behavior of asynchronous circuits, it would be not useful since not required due to elastic timing, to lock operation of a duplicate, this rules out the use of a Golden Node. For our simulation and also used in this thesis, it is more appropriate to use the Golden run concept ([6]). However a cycle-by-cycle comparison based on a rigid time schedule makes no sense for an asynchronous circuit as the name implies. It is much more natural to compare the sequence of events (traces) observed in our device under test to a reference sequence, without alignment to a time scale. So we record the device under test behavior in the fault-free case as a list of events, and relate the sequence of events observed during the actual fault injection to this reference on an entry-by-entry basis.

There are two methods to decide in the fault injector where to inject the transient fault:

- Methodical faults: A strategic way of fault injection, selection of specific signal lines and circuit elements for injection.
- Random faults: Selection of fault injection positions is done randomly.

In our experiments we first use the methodical way to find out the sensitive signals for each state, to find the last missing parameteres of the probability formula. It has to be found out how many signals in a state are sensitive to transient signals and will not be masked in a later step. This will be done for each signal in each state. After the probability formula for calculation is complete, the random procedure is used for evaluating the masking probability formula.

Another important point is observation time and here we see two different possibilities: 1) Inject the fault and reset the target system after each single fault injection or 2) proceed with the next fault injection without a reset of the target, as long as no failure has been observed after a given timeout. It allows the „late“ observation of faults, which means a fault that has not become activated within its associated slot may show up in a later slot. For the simulation done in this thesis method 1 was used, for the fault injection experiments to be statistically independent.

The first step of the fault simulation is the description of the fault free trace:

1. Generation of a fault free trace, with time stamps for each state transition.
2. Calculation of the duration for each state of the state graph with the help of a Python script.

An example is given in Table 5.1, where the duration of state 0 can be seen. It is calculated in the following way that t_1 (time stamp of state 0) is subtracted from t_2 (time stamp of state 2).

The second part is the injection of faults and comparing the produced traces to the fault free trace:

Table 5.2: Traces

Line	<i>trace1</i>	<i>trace2</i>	Compare <i>trace3</i>
0	0000	0000	0000
1	0010	0100	1000
2	0110	0000	
3	0100	1000	

1. Generation of a .do script with injected faults at the signals of interest and producing a set of faulty traces.
2. Comparing the faulty traces to the correct ones.
3. If the injected faults are masked then the error counter will not be increased but if the error propagates and brings the circuit into an error state then the error counter will be increased.

It is important how we can recognize that a transient fault in a faulty trace could be masked. The easiest way is to go to the first line where we found a difference to the fault free trace and delete this line and the following one. If the two traces are completely the same after the deletion of those two lines, we can conclude that the fault is masked and will not propagate. The table 5.2 shows an example. There are two faulty traces, *trace1* and *trace2* where transient faults are injected and a trace (*trace3*) of a correct functioning circuit without fault injection, which is used for comparison. A fault was injected at line 1 and if it is a transient fault which will not propagate, it will disappear in line 2 which is the next state. If we delete the lines 1 and 2 we should see the same trace as for *trace3*. For *trace2* this is possible but for *trace1* the injected fault propagates and introduces more than only 2 new states, in line 2 the circuit is in the state 0110 which totally differs from the compare trace.

For the simulation a VHDL description was written to describe the behavior of the 4-phase bundled data pipeline.

The program and program language which were used for the simulator, used for this diploma thesis are Modelsim and Python. Modelsim is a graphical user interface for performing behavior simulations of VHDL architecture descriptions with the help of a test bench which describe the environment in which the described circuit is embedded. It has also a command line interface for starting simulations and producing output, this feature was used here to automate the fault injection simulation. The programming language Python was used to perform the calculation, write the .do- scripts, automate the behavior simulation, and copy the trace files when comparing the different traces. The transient faults were injected first methodically and in the second simulation step with a random generator at one of the five possible signals at the second stage of the 4-phase bundled data pipeline.

In Figure 5.2 the main steps of the fault simulation can be seen:

1. Methodical fault injection: Inject to each signal in each state a transient fault and see if it will propagate, to find out the sensitive signals.

Table 5.3: Timing trace

STATE	<i>transition timestamp</i>	<i>duration</i>
0	2000	100
1	2100	

2. Random fault injection: Inject a transient fault to a location in the circuit, the decision where to inject the fault is made randomly.
3. Compare traces: Compare the fault free trace to the trace were the transient fault was injected.
4. Use equation: Calculate the Error probability with the help of the formula and not by random injection of transient faults.
5. Calculate state duration: Calculate the duration of each state.
6. Error probability: Compare the result of the random fault injection with the calculated value of the error probability.

Another important requirement for calculation of the fault masking probability is to know the duration for each state. The step for computation of the duration time is shown as example in the following table 5.3. In the simulation the time is stored when a signal changes, a transition from one state to another occurs. This is also done for the next state transition to get from state 0 to state 1. The difference between those two states, in our example $2100 - 2000 = 100$, is the duration time of state 0.

As we have discussed the main building blocks and the functionality of the fault simulator which was used for this thesis, we can look in more detail on the results, the quantitative representation of the fault model which was developed in chapter 2. We will see that the fault model fits to the results of the fault simulation only with some minor deviations.

5.1 Simulation of 4-phase bundled data pipeline

First we will start discussing the four-phase bundled data pipeline and investigate the 2 cases of the pipeline which we have described in the last chapter. The two parameters T_{Rout} and T_{Rin} are the only values which will be changed in the simulation, they are controlled by the environment. All other parameters stay the same only input (T_{Rin}) to output (T_{Rout}) behavior will be changed.

Case 1: Input delay is smaller than output delay

Table 5.4 shows the timing behavior of all three stages of a 4-phase bundled data pipeline. For each state there is a column and the different lines S0, S1 and S2 represent the three stages.

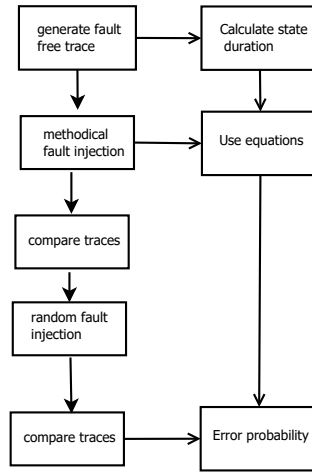


Figure 5.2: sequence diagram

Table 5.4: Duration time for: T_{Rin} : 4 ns and T_{Aout} : 18 ns

S	0	1	2	4	5	6	7	8	9	10	12	13
S0	0	3000	13000	1000	3000	1000	1000	3000	13000	1000	2000	1000
S1	3000	3000	13000	1000	2000	1000	2000	3000	13000	1000	2000	2000
S2	6000	3000	13000	1000	2000		2000	3000	13000	1000	2000	2000

We can see that the values for the duration time for each state in stage S1, which we get from the simulation are the same values which we get when we use the formulas from Table 4.1, in Chapter 3. For the circuit elements we use the same values as in the previous chapter: $M = 3$ ns, $\Delta = 13$ ns and $T_{D2} = 1$ ns. For the input T_{Rin} to be faster than the output T_{Aout} the values $T_{Rin} = 4$ ns and $T_{Aout} = 18$ ns are chosen. We can conclude that $4 - 3 < 18 - 13 - 3$ and therefore input is faster than the output. We also see some states which are not possible if the circuit is in the steady state because here also the state transitions which appear at the beginning of the simulation are in the trace file. Those lines are not deleted because of completeness. For example in the steady state of the circuit, the state 0 is not possible if the input delay is shorter than the output delay of the circuit but at the beginning of the simulation all signals are 0, therefore this state is only possible for one time and for all stages. The delay of state 0 at the different stages is exactly the delay of Muller-C element, which fits to our circuit description.

In the following two Tables 5.5 and 5.6 the delays of the outputs (T_{Aout}) are larger, to find out the delay behavior of the different states. We see that only the resulting delay of the two states 7 and 13 and their mirrored states have changed when we look at the second stage of the pipeline. The delay of the states where the output delay (T_{Aout}) is included in the formula

Table 5.5: Duration time for: T_{Rin} : 4 ns and T_{Aout} : 38 ns

S	0	1	2	4	5	6	7	8	9	10	12	13
S0	1000	3000	13000	1000	3000	1000	21000	3000	13000	1000	2000	21000
S1	3000	3000	13000	1000	2000	1000	22000	3000	13000	1000	2000	22000
S2	6000	3000	13000	1000	2000		22000	3000	13000	1000	2000	22000

Table 5.6: Duration time for: T_{Rin} : 13 ns and T_{Aout} : 38 ns

S	0	1	2	4	5	6	7	8	9	10	12	13
S0	10000	3000	13000	1000	3000	1000	12000	3000	13000	1000	2000	12000
S1	3000	3000	13000	1000	2000	1000	22000	3000	13000	1000	2000	22000
S2	6000	3000	13000	1000	2000		22000	3000	13000	1000	2000	22000

changes. As an example the calculation of the duration for the two states 7 and 1 in Table 5.6 are shown:

- 1: $M = 3$ ns
- 7: $T_{Aout} - M - \Delta = 38 - 3 - 13 = 22$ ns

Another interesting finding is that some states in the first stage of the pipeline have different durations compared to stage zero and stage two. The states 7 and 13 are different because the input of the pipeline is faster than the output and the formula for these states are calculated in the following way: $(T_{Aout} - \Delta - M) - (T_{Rin} - 5) = 12000$.

Calculation of fault masking probability

Now the fault masking probability will be calculated with the help of the formulas which we found in the previous chapter. First we will start with calculation of the duration times for each state. In the first column of Table 5.7 the duration times for each state can be seen, the second column is the percentage of the period time. The second important parameter for calculation are the number of sensitive signals for each state and a value for the percentage, as shown in column three and four in Table 5.7. In the last line the probabilities are summarized for the calculation of the mean value to be able to use the following as formula: $E(x) = p * x$. The variable x is the number of transient faults which are injected to the pipeline and p is the summarized value, in our case the value is 0,20952381.

In Table 5.7 the calculation of the mean value and all its calculation steps are given for the situation where input is faster than output. The mean value shows the probability that a transient fault which is injected to the inputs or outputs of the second stage of the 3-stage 4 phase bundled data pipeline, is not masked and forwarded to the next stage, thus producing an error in the pipeline. One important thing is how we get to the number of the sensitive signals (signals in a state, which can lead to an error if a transient fault is injected). One way is by analyzing

Table 5.7: Results of Fault simulation

S	duration		sensitive signals		probability
0	0	0	0	0	0
1	3000	0,071428571	1	0,2	0,014285714
2	13000	0,30952381	1	0,2	0,061904762
3	0	0	0	0	0
4	1000	0,023809524	1	0,2	0,004761905
5	3000	0,071428571	1	0,2	0,014285714
6	0	0	0	0	0
7	1000	0,023809524	2	0,4	0,00952381
8	3000	0,071428571	1	0,2	0,014285714
9	13000	0,30952381	1	0,2	0,061904762
10	1000	0,023809524	1	0,2	0,004761905
11	0	0	0	0	0
12	3000	0,071428571	2	0,2	0,014285714
13	1000	0,023809524	2	0,4	0,00952381
Σ	1000				0,20952381

the circuit states theoretically, the other is a simulation for each state of the state machine and injection at each simulation run on each signal of the state one transient fault. By comparing the outcome of each simulation to the compare trace it is possible to find out if an injected fault on a signal in a specific state will propagate and therefore produce a faulty trace. This simulation to find out the bad signal has to be done for different input to output timings because the behavior of the second stage of the 4-phase bundled data pipeline depends on the timing behavior of the predecessor and successor stages of this pipeline. If the duration of a stage gets longer more transient faults could propagate.

In the simulation we injected 500 faults and reset the simulation after each fault injection. For this simulation, seen in Figure 5.7 we get an estimate which is calculated in the following way: $E(500) = 500 \cdot 0,20952381$. It shows the expected value of how many transient faults will propagate and not be masked.

Comparison between calculation and simulation

We are also interested in comparing the values which we got from the simulation to calculated values. The first row in 5.8 is a parameter (DIFF), showing the input to output delay, it represents the speed of the pipeline, how fast the pipeline processes data, beginning at stage zero to stage two. The unit of this parameter is milliseconds and we started with nearly maximum speed and then we made the pipeline slower. We got the following result:

The number of masked faults gets larger the faster the pipeline is driven, as we can see in Table 5.8. Here we also see the difference between the values which are calculated with the formulas

Table 5.8: Comparison between simulation and calculation

DIFF	1	2	5	10	20	30	40	50
SIM	105	109	120	141	193	194	195	196
CALC	105	109	120	141	193	194	195	196

Table 5.9: Duration time for: T_{Rin} : 5 ns and T_{Aout} : 17 ns

S	0	1	2	4	5	6	8	9	10	12
ST0	2000	3000	13000	1000	2000	2000	3000	13000	1000	2000
ST1	2000	3000	13000	1000	2000	2000	3000	13000	1000	2000
ST2	1000	3000	13000	1000	3000	1000	3000	13000	1000	3000

Table 5.10: Duration time for: T_{Rin} : 25 ns and T_{Aout} : 17 ns

S	0	1	2	4	5	6	8	9	10	12
ST0	22000	3000	13000	1000	2000	22000	3000	13000	1000	2000
ST1	22000	3000	13000	1000	2000	22000	3000	13000	1000	2000
ST2	21000	3000	13000	1000	3000	21000	3000	13000	1000	3000

and the values which we can get from the simulation. The values fit together so the probability model seems to be reliable.

Case 2: Input delay is larger than output delay

Now we also want to investigate the opposite case where the output delay is faster than the input delay at stage one ($T_{Rin} - M > T_{Aout} - \Delta - M$). It means that data in the pipeline is faster processed than new data is available. First we want to find out if the simulated duration of the different states are the same as the calculated one in Table 4.1. The results in the Tables 5.9, 5.10 and 5.11 fits exactly to our expectation.

As before we also can find some states which behave a little bit different compared to stage two of the pipeline, we can see those states in stage two because the output delay T_{Aout} is shorter than the input delay T_{Rin} . The states which are influenced by this situation are the following two states six and zero. The calculation for the duration of the states zero and six in 5.11 is the following:

$$(T_{Rin} - M) - (T_{Aout} - \Delta - M) = (25000 - 3000) - (26000 - 13000 - 3000) = 22000 - 10000 = 12000$$

Calculation of fault masking probability

The same procedure, as before, will be used to compute the probability value for the masking effects in the situation where the output delay is shorter than the input delay and the results are the following:

The calculation of the mean value and the values for each state can be seen in table 5.12. If

Table 5.11: Duration time for: T_{Rin} : 25 ns and T_{Aout} : 26 ns

S	0	1	2	4	5	6	8	9	10	12
ST0	22000	3000	13000	1000	2000	22000	3000	13000	1000	2000
ST1	22000	3000	13000	1000	2000	22000	3000	13000	1000	2000
ST2	12000	3000	13000	1000	3000	12000	3000	13000	1000	12000

Table 5.12: Fault simulation

S	duration		sensitive signals		probability
0	0	0	0	0	0
1	3000	0,071428571	1	0,2	0,014285714
2	13000	0,30952381	1	0,2	0,061904762
3	0	0	0	0	0
4	1000	0,023809524	1	0,2	0,004761905
5	3000	0,071428571	1	0,2	0,014285714
6	0	0	0	0	0
7	1000	0,023809524	2	0,4	0,00952381
8	3000	0,071428571	1	0,2	0,014285714
9	13000	0,30952381	1	0,2	0,061904762
10	1000	0,023809524	1	0,2	0,004761905
11	0	0	0	0	0
12	3000	0,071428571	2	0,2	0,014285714
13	1000	0,023809524	2	0,4	0,00952381
Σ	42000				0,20952381

input is faster than output, it has more impact on masking if we make the duration of the output larger. The expected value for 500 injected faults was calculated in the following way as before: $E(500) = 500 \cdot 0,20952381$. Of course if the delay of the input is changed then also the fault probability will change but only to a specific limit. The smaller the difference between input and output delay gets, the larger the number of masked transient fault increases in a 4-phase bundled data pipeline. Here as we can see the output delay defines the limit of masked faults.

Why the masking effect is better when the pipeline is driven very fast is because of the masking behavior of the Muller C-elements, the only element in this circuit where the masking effect exactly takes place at its input. In paper [23] this behavior was investigated and described also quantitatively. A theoretical analysis indicated the two modes of operation of Muller C-element, namely transparent mode and hold mode indeed these account for a different sensitivity of single-event fault. In the Muller pipeline the Muller C-elements assume the more robust transparent mode only transiently when propagating an input transition, while in the quiescent state hold mode is dominating. This suggests that the Muller pipeline is less sensitive when operated close to its maximum frequency. We got the same results in our fault injection experiments.

Table 5.13: Comparison between simulation and calculation

DIFF	1	2	5	10	20	30	40	50
SIM	105	105	105	156	156	156	156	156
CALC	105	105	105	156	156	156	156	156

5.2 Simulation of 2-phase bundled data pipeline

The next subchapter will be about the simulation of the 2-phase bundled data pipeline and comparing the calculated values to the results of the simulation. For this thesis a VHDL description was written for the simulation of the pipeline and with the help of these programs, Modelsim and some Python scripts, the results can be evaluated. Our 2-phase bundled data pipeline is a little bit more difficult for timing analysis. In the 4-phase bundled data pipeline we found that there is only one circuit element in the circuit which is responsible for the masking effect – the Muller C-element. Here, in the 2-phase bundled data pipeline we can also find a comparable element, which produces masking effects, the xnor gate. The xnor gate is only 1 at the output, if both inputs have the same values, otherwise its output is 0. For this circuit we will again make a simulation with three pipeline stages. For timing analysis, in the simulation we will wait until the circuit is in the steady state then it will only follow one path in the state graph. As before in the 4-phase bundled data pipeline, the injection of transient faults will only be done on the signals of the second stage, in the middle of the pipeline.

Case 1: Input delay is smaller than output delay

First we will start measuring the duration of each state for each stage of the pipeline. As an example, in Figure 5.14, we see the states 0 to 17 for the 3 stages (ST1, ST2 and ST3). The first experiment with the condition that the input is faster than the output. Two simulation runs are combined in this table, to see what happens if we change the output delay, the first column shows the three stages of the same pipeline but with different input to output delays. On the left side the input delay is smaller compared to the simulated values of the stages on the right side of table 5.14.

Following values are chosen for the two simulations in Figure 5.14:

1. on the left side: $A \rightarrow R$: 3 ns and $R \rightarrow A$: 20 ns
2. on the right side: $A \rightarrow R$: 3 ns and $R \rightarrow A$: 30 ns

Some of the states are not reachable if we chose the specific timing, as we have seen in the last chapter, so we always have the same path in the state graph. As the circuit has a faster input than output, the first stage of the 3 stage bundled pipeline has a different timing than the other two stages. For example when we look at state 14 in stage 0, the state's duration is longer than the duration of the same state in the stages ST1 and ST2. The simulation of the state durations fits exactly to the calculation of the formulas for duration of the second stage of a three stage pipeline. The gate delays of the circuit elements are:

Table 5.14: Duration for each state of the 2-phase bundled data pipeline

	simulation run 1				simulation run 2		
STATE	ST0	ST1	ST2		ST0	ST1	ST2
0	130	130	130		130	130	130
1	2000	2000	2000		2000	2000	2000
2	130	130	130		130	130	130
3	12870	12870	12870		12870	12870	12870
5	3000	2130	2130		3000	2130	2130
8	4000	4870	4870		14000	14870	14870
9	130	130	130		130	130	130
10	2000	2000	2000		2000	2000	2000
11	130	130	130		130	130	130
13	12870	12870	12870		12870	12870	12870
14	3000	2130	2130		3000	2130	2130
17	4000	4870	4870		14000	14870	14870

- $T_{xnor} \dots 130$ ps
- $T_{D2} \dots 2$ ns
- $\Delta \dots 15$ ns

To prove the correctness of the formulas in table 4.2, we can use those concrete values for the variables. The values for T_{Rin} (input delay) and T_{Aout} (output delay) are chosen for each experiment separately, it depends if we want to have faster input or output. The formula for calculating the difference between input and output was already discussed in the previous chapter.

For the left side of table 5.14 we can do the following calculation for input to output behaviour and show that here $A \rightarrow R$ is smaller than $R \rightarrow A$:

- $T_{xnor} + T_{D2} = 2130$
- $\Delta = 15000$
- $A \rightarrow R : 3 \text{ ns} , 3000 - 2130 = 870$
- $R \rightarrow A : 20 \text{ ns} , 20000 - 2130 - 15000 = 2870$

$A \rightarrow R$ means the input delay, in our case the delay between A_{in} and the signal change of Req at the first stage of the pipeline. $R \rightarrow A$ is the opposite, the output delay, the delay between the change of $Rout$ and the time when the input value of signal $Aout$ is different.

Table 5.15: Fault simulation results

S	duration		sensitive signals		probability
0	130	0,002937189	3	0,6	0,001762314
1	2000	0,045187528	4	0,8	0,036150023
2	130	0,002937189	3	0,6	0,001762314
3	12870	0,290781744	3	0,6	0,174469047
4	0	0	0	0	0
5	2130	0,048124718	3	0,6	0,028874831
6	0	0	0	0	0
7	0	0	0	0	0
8	4870	0,110031631	4	0,8	0,088025305
9	130	0,002937189	3	0,6	0,001762314
10	2000	0,045187528	4	0,8	0,036150023
11	130	0,002937189	5	0,6	0,001762314
12	0	0	0	0	0
13	12870	0,290781744	3	0,6	0,174469047
14	2130	0,048124718	3	0,6	0,028874831
15	0	0	0	0	0
16	0	0	0	0	0
17	4870	0,110031631	4	0,8	0,088025305
Σ	44260				0,662087664

Calculation of fault masking probability

In Table 5.15 we have the table for the calculation of the expected value. Now we have more states compared to the state graph of the 4-phase bundled data pipeline. It is interesting that in a 2-phase bundled data pipeline, there are more bad signals in the stages. The calculation for the expected value was done in the same way as before, for these experiments 500 faults were injected and the expected value can be calculated in the following way: $E(500) = 500 \cdot 0,662087664$. If the output delay is made larger and the input delay is kept the same then there will be less transient fault masking. It is comparable to the behavior of the Muller C-gate, the circuit is more sensitive to faults if the circuit is operated at lower frequency.

Case 2: Output delay is smaller than input delay

We also want to investigate the case when the output is faster than the input. The delay which was chosen for the two timing analysis in Figure 5.16 are the following:

1. on the left side: A \rightarrow R: 3 ns and R \rightarrow A: 15 ns
2. on the right side: A \rightarrow R: 8 ns and R \rightarrow A: 18 ns

Table 5.16: state duration

STATE	simulation run 1				simulation run 2		
	ST0	ST1	ST2		ST0	ST1	ST2
1	2000	2000	2000		2000	2000	2000
2	130	130	130		130	130	130
3	12870	12870	12870		12870	12870	12870
5	2000	2000	2000		2000	2000	5000
6	130	130	130		130	130	130
7	870	870	870		5870	5870	2870
10	2000	2000	2000		2000	2000	2000
11	130	130	130		130	130	130
13	12870	12870	12870		12870	12870	12870
14	2000	2000	2000		2000	2000	5000
15	130	130	130		130	130	130
16	870	870	870		5870	5870	2870

If we fulfill the following rules, $T_{Aout} \geq T_{xnor} + T_{D2} + \Delta$ for output duration and $T_{Rin} \geq T_{xnor} + T_{D2}$ for input duration, we can distinguish if input is faster than output by calculating $diff = T_{Rin} - T_{Aout}$. This is necessary to find out which condition in table 4.2 should be used to calculate the state durations.

Calculation of fault masking probability and comparison

The result of this simulation to calculate the expected value as explained before, can be seen in table 5.17 and here the expected value is: $E(500) = 500 \cdot 0,670695652$.

If we use the formulas of the last chapter to calculate the fault probability of the 2-phase bundled data pipeline and compare it to the simulation data, we can see that those values are similar only with minor deviation. The results of this comparison can be seen in table 5.18. The difference between input to output delay was changed and always 500 faults have been injected. In the first line we can see the difference of output to input delay which was calculated in the following way: $diff = (T_{Rin} - T_{xnor} - T_{D2}) - (T_{Aout} - T_{xnor} - T_{D2} - \Delta)$. Our results also show that if the difference gets larger the difference between calculated to simulated values becomes smaller. If the difference is smaller, the pipeline is driven faster and the circuit becomes more robust to injected transient faults. The behaviour of the xnor-gate is comparable to the behaviour of the Muller C-gate. As a result of those simulations we found out that the 4-phase bundled data pipeline is more robust to transient faults than the 2-phase bundled data pipeline. There are more states which are able to produce faulty states, running the pipeline at higher speed (less difference between output to input delay) gives a better performance but is not comparable to the 4-phase type.

Table 5.17: Fault simulation results

S	duration		sensitive signals		probability
0	0		0	0	0
1	2000	0,043478261	4	0,8	0,034782609
2	130	0,002826087	4	0,8	0,00226087
3	12870	0,279782609	3	0,6	0,167869565
4	0	0	0	0	0
5	2000	0,043478261	3	0,6	0,026086957
6	130	0,002826087	4	0,8	0,00226087
7	5870	0,127608696	4	0,8	0,102086957
8	0	0	0	0	0
9	0	0	0	0	0
10	2000	0,043478261	4	0,8	0,034782609
11	130	0,002826087	4	0,8	0,00226087
12	0	0	0	0	0
13	12870	0,279782609	3	0,6	0,167869565
14	2000	0,043478261	3	0,6	0,026086957
15	130	0,002826087	4	0,8	0,00226087
16	5870	0,127608696	4	0,8	0,102086957
17	0	0	0	0	0
Σ	46000				0,670695652

Table 5.18: Comparison between simulation and calculation

DIFF	-12	-2	7	10
SIM	335	322	331	352
CALC	335	318	329	352



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Comparison




In this chapter a short comparison between the masking effects of asynchronous and synchronous logic will be given. It also includes a summary of the results of this diploma thesis and the findings will be discussed in more detail. With the help of the calculations which are explained in this thesis further investigations can be done about whether asynchronous logic behaves more robust when transient faults are injected.

6.1 Synchronous Pipeline

In this thesis the main topic was to show the behaviour of asynchronous pipelines, but now we also want to introduce a simple example for a synchronous design, to be able to compare the masking effects. A picture of such a simple design of a synchronous pipeline can be seen in Figure 6.1. We use again a three stage pipeline but here the propagation steps are not controlled by the signal values and some kind of completion detector but only by a clock signal. The circuit elements which are controlled by this clock signal and handle the propagation of data values, are D-flip-flops. D-flip-flops are circuits that have two stable states and can be used to store state information. It would also be possible to use a double edged logic, with means a inverter at the clock input of the second stage of the pipeline, which would have the advantages of using a higher speed. The double edged logic wasn't chosen because of the following disadvantages:

- An asymmetrical clock duty cycle can cause setup and hold violations.
- It is difficult to determine critical signal paths.
- Test methodologies such as scan-path insertion are difficult, as they rely on all flip-flops being activated on the same clock edge. If scan insertion is required in a circuit with double-edged clocking, multiplexers must be inserted in the clock lines to change to single-edged clocking in test mode.

Table 6.1: States of the D-flip-flop

E	D	Q	\bar{Q}
0	0	hold	hold
0	1	hold	hold
1	0	hold	hold
1	1	hold	hold
	0	hold	hold
	1	1	0
	0	0	1

Double edged logic is described in more detail in paper [1].

A D-flip-flop has only one data input: the „D“ input. Activating the D input sets the circuit, and deactivating the D input resets the circuit. Of course, this only happens if there is a rising edge on the clock input (E) as well. Otherwise, the output(s) will be saved, unresponsive to the state of the D input. This behavior is summarized in table 6.1. This table shows the two inputs D and E as well as the output signal Q and its inverse signal \bar{Q} .

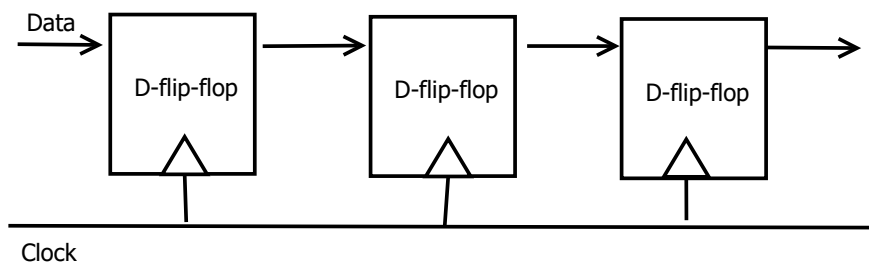


Figure 6.1: 3-stage, synchronous pipeline

As we can see in Figure 6.1, the enable inputs of the D-flip-flops are controlled by the clock signal. This signal is a single point of failure, if a transient fault is injected on this signal line, the fault can propagate immediately or data is captured at a D-flip-flop [20]. On the other side if a transient fault is injected on the D-signal at the input of the flip-flop then it will only propagate if the flip-flop is open and ready to receive new data, in all other cases the flip-flop will ignore the transient fault and there will be no fault propagation. This is the latching window masking. To show the behaviour of this pipeline architecture in more detail, we have to create the state diagram, shown in Figure 6.2. In each state we can see the value of each signal and the state transition if one signal changes its value. We will show a formula and behaviour of the second stage of a three stage pipeline as we have done it for asynchronous logic, to be able to compare those two architecture styles in the presence of injected transient faults. First we have to describe the state durations with the help of formulas, this is why we need to start with a state diagram. A state contains the following signal values, which have the same name as in Figure 6.1:

- CLK : The clock signal.
- D : Input value of the data signal.
- Q : Output value of the data signal at the flip-flop.
- WIN : Time window of the flip flop (setup + hold time).

After creating the state diagram of the 3-stage synchronous pipeline we want to find the duration of each state to describe the temporal behaviour in more detail. This is necessary to find a formula which describes the probability of fault masking for this kind of circuit. To find the duration of each state in the diagram, the fault simulator can be used, which we created for the asynchronous circuits. To be able to use this simulator, the pipeline architecture and the testbench have to be changed appropriately in the VHDL code. The code which was used for this simulation can also be found on the attached CD and a description how to use the simulator is give in Appendix A.

With the help of the simulator and the state diagram it is possible to describe the duration of each state as shown in table 6.2. To read the table the following parameters are described:

- T_{clk} : Period of the CLK signal.
- t_{setup} : Setup time of the latching window.
- t_{hold} : Hold time of the latching window.
- t_{CO} : Rising edge to clock output delay.
- t_{PD} : Time for output to appear after input is applied.

The left column „duration change“ shows the duration of each state if after each clock period the input signal D changes. The second column shows the durations if the input signal doesn't change after one clock period. Per simulation it is now also possible to find out the signal values in each state which are sensitive to transient faults. First we want to explain the fault

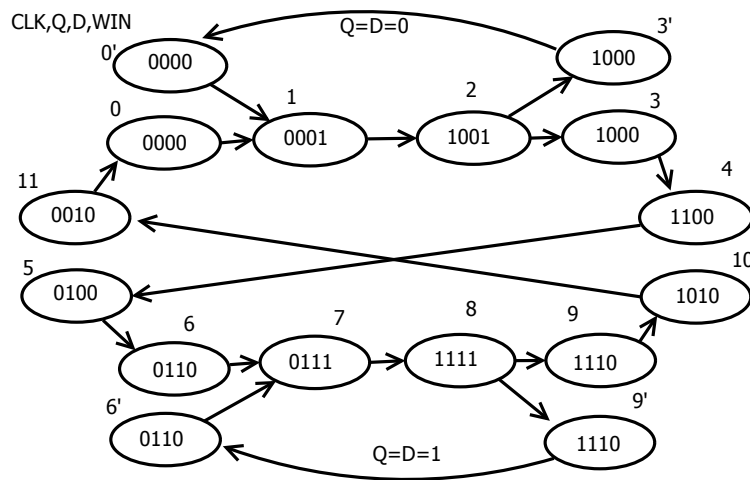


Figure 6.2: state diagram

propagation with a formula as we now found out the duration for each state. We can identify the circuit elements which are sensitive to fault theoretically or by the fault simulator.

As we can see in table 6.2, there are seven states which are influenced by the clock signal (0,4,5,6,7,10,11), the other signals are defined by the circuit parameters, for example the delay of the flip-flop. The circuit parameters influence the optimal clock frequency of the synchronous pipeline, to get more masking effects. Next step is to make some fault injection simulation with different clock frequencies for our synchronous pipeline implementation, we will show this in the next sub chapter. With the help of the state diagram and information about the duration of the pipeline in the different states we can find the sensitive signals in the synchronous pipeline. For example transient faults on the signal *CLK* will always lead to fault propagation.

If we look at the synchronous pipeline and check for the element which helps to mitigate the propagation of transient fault we can conclude the following. The only circuit element which is responsible for masking effects in this synchronous pipeline is the D-flip-flop when it is in the blocked state. Every transient fault will propagate if this circuit opens and stores the faulty information. Our previous analysis has shown that in asynchronous logic a higher speed for the pipeline results in more masking effects, therefore less fault propagation. The next step is now to compare this behaviour to the masking effects in synchronous logic. In the asynchronous pipeline design the masking behaviour was controlled by the delays of input and output, in synchronous logic the speed is controlled by a clock signal and this signal is responsible for opening and closing all state holding elements.

Table 6.2: Duration of the stages of a synchronous pipeline

state	duration change	duration no change
0	$\frac{T_{clk}}{2} - t_{setup}$	
1	t_{setup}	
2	t_{hold}	
3	$t_{CO} - t_{hold}$	
4	$\frac{T_{clk}}{2} - t_{CO}$	
5	$t_{CO} + t_{PD} - \frac{T_{clk}}{2}$	
6	$\frac{T_{clk}}{2} - (T_{delay_length} + T_{in})$	
7	$T_{clk} - (t_{setup} - (t_{CO} + t_{PD}))$	
8	t_{hold}	
9	$t_{CO} - t_{hold}$	
10	$\frac{T_{clk}}{2} - t_{CO}$	
11	$t_{CO} + t_{PD} - \frac{T_{clk}}{2}$	
0'		$\frac{T_{clk}}{2} - t_{setup}$
3'		$\frac{T_{clk}}{2} - t_{hold}$
6'		$\frac{T_{clk}}{2} - t_{setup}$
9'		$\frac{T_{clk}}{2} - t_{hold}$

6.2 Masking Effect Results

First we want to show the types of fault injection in synchronous logic. Then we will compare how the masking effects change when the pipeline is driven with a high rate, to the behaviour in asynchronous logic.

As already mentioned before there are two different categories of faults in synchronous pipelines:

- Faults injected on control signal.
- Faults injected on data line, which was not investigated in this thesis with bundled data pipelines but will be discussed here in more detail.

In synchronous logic the control signal is the clock signal, and if transient faults are injected on this signal it will immediately result in an error. The control signals of asynchronous logic have more similarities to data signals in synchronous logic. This is also a main difference when comparing the masking behaviour of synchronous and asynchronous pipeline.

Transient faults injected on data lines are only propagated if the state holding elements are open and ready to store the input data which is provided to their inputs. In asynchronous logic this type has a similar behavior for the control signals, when the values are stored by the C-element. For a D-flip-flop in order to store data and propagate data, the setup and hold times have to be met. The setup time is the minimum time before which data must be stable before clock transition and hold time is the minimum time for which data must be stable after active

clock transition. Set up and hold is related to input signal and clock edge, not to the output. If the input meets these requirement then we will get a valid output at this element. If a transient fault is injected in a way that those times are not met, then it will also result in a faulty state. A timing diagram which shows the setup and hold time for a D-flip-flop can be seen in Figure 6.3.

Simulation

To show the behavior of a synchronous pipeline when transient faults are injected, a three stage pipeline with D-flip-flops as state holding elements was used (6.1), which was described before. We used different clock rates to get to some result to be able to compare the masking behavior to asynchronous logic. The following steps are describing the simulation of transient fault injection:

1. Start simulation.
2. Inject fault on the data signal at the input of the second D-flip-flop in state four or nine.
3. Check if this fault injection results in a faulty state and error propagation.
4. Save the result of this test run and reset the environment.
5. Start again at point 2 or stop and go to the next point if this simulation was already executed 500 times (500 faults were injected).
6. Change clock speed and start at point 1.

Table 6.3 shows how many faults are propagated at different frequencies, the parameter T_{win} has a fix value of 10 ns. The results of this simulation can be seen in Figure 6.4, which shows the fault propagation for different clock rates. The key result of this simulation is, if the synchronous pipeline was driven with a higher clock rate, less fault masking was happening and more transient faults are propagated.

As we can see if the pipeline is driven faster more faults will propagate, which is completely different to the behaviour in asynchronous logic, where we got the opposite result. The optimal frequency for the synchronous pipeline always depends on the circuit parameters (e.g.: computation delays) and is not as flexible as asynchronous designs where different delays for the stages can exist.

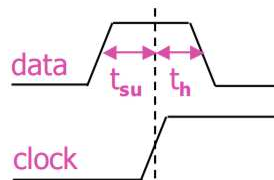


Figure 6.3: Setup- and hold-Time of a D-flip-flop

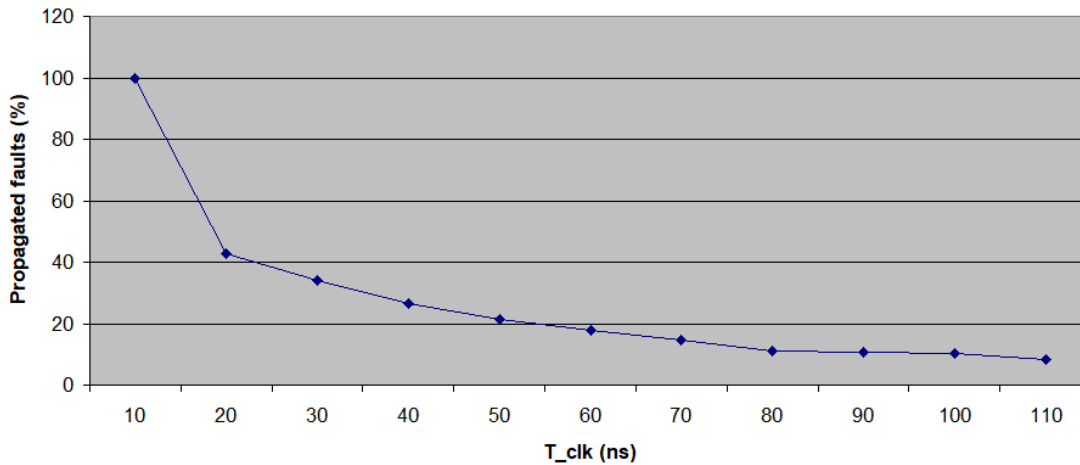


Figure 6.4: Fault propagation in a 3 stage-synchronous pipeline

Table 6.3: Results of simulation

T_{clk} (ns)	propagated faults
10	500
20	214
30	170
40	132
50	106
60	89
70	74
80	56
90	54
100	51
110	42

Theory

Paper [14], which is about error probability in synchronous digital circuits due to power supply noise, shows similar results. There are two ways to achieve less error propagation, improving the power supply voltage quality (increasing cost) and increasing the clock period for a better safety margin (decreasing performance). In this thesis a formula for the second reason for less fault propagation is shown in the following:

$$P_{fail} = \frac{T_{win}}{T_{clk}} = \frac{t_{setup} + t_{hold}}{T_{clk}} \quad (6.1)$$

In Figure 6.6 we can see the result, for fault propagation, of this formula when we use different frequencies. The parameter T_{win} and all different states of the pipeline are shown in

Figure 6.5. T_{win} is the part of the period T_{clk} where the flip flop is sensitive to transient faults. For our computation the parameter T_{win} has the fixed value of 10 ns. We can see that the computation with the help of the formula fits to the result of the simulation.

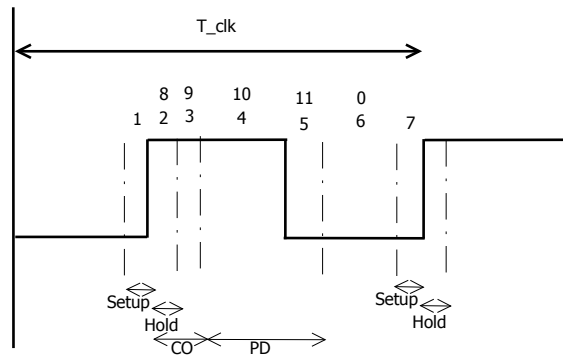


Figure 6.5: T_{win}

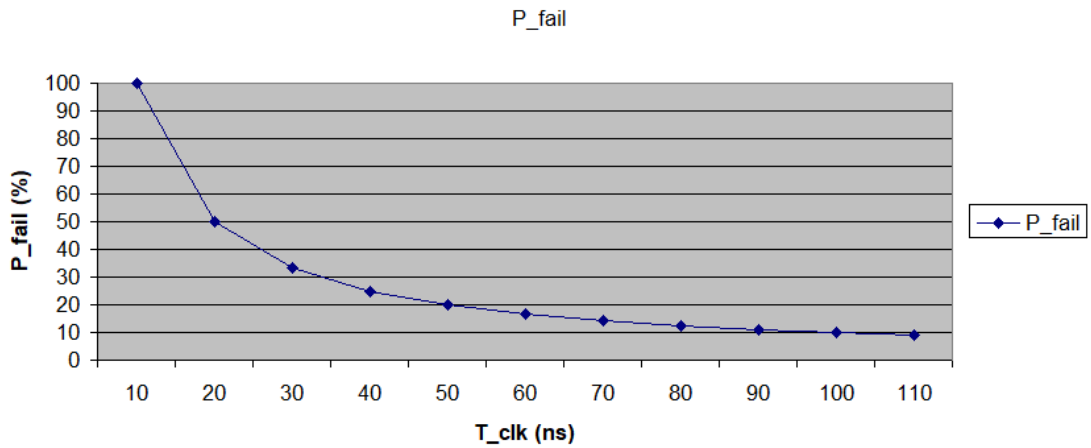


Figure 6.6: P_{fail}

For this pipeline setup in synchronous logic two different fault situations can be distinguished. First situation is when the transient fault is injected and the setup and hold times are not violated, the other case is when those times are not met. Both cases will result in fault propagation. Only if the transient fault is injected outside of the time window of opening the D-flip-flop for new data, masking will happen. So we can see that there is only one situation

where fault propagation is prohibited, dependent on the window size T_{win} . Compared to asynchronous logic it is more prone to faults as for example the signal CLK is always sensitive to transient faults, whereas in asynchronous logic we don't have one signal where fault injection always leads to fault propagation. Of course the fault propagation was already investigated in different papers and solutions for this topic have been found.

To make synchronous logic more robust logic functions and element are duplicated and voters are added at the end of the circuit to define the output value by majority voting [1].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Summary

7.1 Summary

This chapter provides a brief summary of the work that is contained in this thesis and shows the main findings.

We started with describing the 3 different fault masking effects, (1) Electrical masking, (2) Logical Masking, (3) Temporal masking.

After describing different already existing fault models, a new description for the behaviour of asynchronous logic was built. First we started with analyzing the the different states of a 3-stage asynchronous bundled data pipeline. After building a probability model we were able to quantitatively describe the masking effects when transient faults are injected. We continued to do the same for a 2-phase bundled data pipeline. To verify the values which we found in a mathematical way, we built a simulation setup which could be used for both pipeline types. The result of the simulation was that the probability model fits exactly to the simulated pipeline circuits. Changing circuit parameters results in different outcomes in the masking behaviour. With the fault model which is described in this thesis it is now possible to find out if the parameters of the circuit and the timing behaviour of the environment are chosen appropriately.

The last part in this thesis was a short comparison between the masking effects in synchronous and asynchronous logic. A synchronous pipeline was simulated and we could find out that more transient faults are masked if the pipeline is driven with lower clock frequency. So we can see that it behaves exactly in the opposite way as asynchronous logic where faster cycle times result in higher fault masking. Asynchronous logic has more cases in which fault masking is happening compared to synchronous logic, which might be taken as an indication that asynchronous logic is more robust to transient fault injection than synchronous logic.

The fault injection simulator which was developed for this thesis can help to analyze the behaviour of pipeline circuits in more detail and can be extended to be used for other circuits. In Appendix A a short explanation how to use the fault simulator is given and the main steps of the

simulation process are described.

This thesis gives promising results to show that asynchronous circuits can be used in an environment where high clock frequency is needed and transient faults are present, as also shown in publications [9] and [12]. With the help of hardening techniques in asynchronous logic even more fault propagation can be prohibited [15].

Bibliography

- [1] Mohit Arora. *The Art of Hardware Architecture*. Springer, 2012.
- [2] John Bainbridge and Steve Furber. Chain: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.
- [3] William John Bainbridge and Sean James Salisbury. Glitch Sensitivity and Defense of Quasi Delay-Insensitive Network-on-Chip Links. *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 35–44, 2009.
- [4] Todd A. DeLong, Barry W. Johnson, and Joseph A. Profeta. A fault injection technique for VHDL behavioral-level models. *IEEE Design and Test of Computers*, 13(4):24–33, 1996.
- [5] Werner Friesenbichler. *Effects and Mitigation of Transient Faults in Quasi Delay-Insensitive Logic*. PhD thesis, TU Wien, 2011.
- [6] Werner Friesenbichler, Thomas Panhofer, and Andreas Steininger. A deterministic approach for hardware fault injection in asynchronous QDI logic. *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 6(4):317–322, 2010.
- [7] Gottfried Fuchs, Matthias Függer, and Andreas Steininger. On the threat of metastability in an asynchronous fault-tolerant clock generation scheme. *Proceedings - International Symposium on Asynchronous Circuits and Systems*, (809456):127–136, 2009.
- [8] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2):247–253, 1996.
- [9] Wonjin Jang. Soft-error robustness in QDI circuits. *Workshop on System Effects of Logic Software*, 2005.
- [10] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: the MEFISTO tool. *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, 2:66–75, 1994.
- [11] Jung Sub Kim, Chrysostomos Nicopoulos, N Vijaykrishnan, Yuan Xie, and Emanuele Lattanzi. A Probabilistic Model for Soft-Error Rate Estimation in Combinational Logic. *Proceedings of the International Workshop on Probabilistic Analysis Techniques for Realtime and Embedded Systems*, 2004.

- [12] Weidong Kuang, Enjun Xiao, Casto Manuel Ibarra, and Peiyi Zhao. Design asynchronous circuits for soft error tolerance. *Proceedings 2007 IEEE International Conference on Integrated Circuit Design and Technology, ICICDT*, pages 221–225, 2007.
- [13] C LaFrieda and R Manohar. Fault detection and isolation techniques for quasi delay-insensitive circuits. *Dependable Systems and Networks, 2004 International Conference on*, pages 41–50, 2004.
- [14] Ferran Martorell, Marc Pons, Antonio Rubio, and Francesc Moll. Error probability in synchronous digital circuits due to power supply noise. *2007 International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, pages 170–175, 2007.
- [15] Y. Monnet, M. Renaudin, and R. Leveugle. Designing Resistant Circuits against Malicious Faults Injection Using Asynchronous Logic. *IEEE Transactions on Computers*, 55(9):1104–1115, 2006.
- [16] Y. Monnet, M. Renaudin, and R. Leveugle. Formal analysis of quasi delay insensitive circuits behavior in the presence of SEUs. *Proceedings - IOLTS 2007 13th IEEE International On-Line Testing Symposium*, (Iolts):113–118, 2007.
- [17] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. *2010 IEEE Symposium on Asynchronous Circuits and Systems*, pages 3–14, 2010.
- [18] Irith Pomeranz and Sudhakar M Reddy. On Application of Output Masking to Undetectable Faults in Synchronous Sequential Circuits with Design-for-Testability Logic. IEEE, 2003.
- [19] Amir Mohammad Rahmani, Ali Asghar Salehpour, Masoud Zamani, Siamak Mohammadi, and Hossein Pedram. An efficient fault simulator for QDI asynchronous circuits. *Proceedings - 2008 4th Southern Conference on Programmable Logic, SPL*, pages 99–104, 2008.
- [20] Emre Salman, Ali Dasdan, Feroze Taraporevala, Kayhan Küçükçakar, and Eby G. Friedman. Exploiting setup - Hold-time interdependence in static timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1114–1125, 2007.
- [21] Premkishore Shivakumar, M Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *Proceedings International Conference on Dependable Systems and Networks*, 2002.
- [22] Jens Sparsø. *Principles of asynchronous circuit design - A systems Perspective*. Kluwer Academic Publishers, 2001.
- [23] Andreas Steininger, Varadan S. Veeravalli, Dan Alexandrescu, Enrico Costenaro, and Lorena Anghel. Exploring the state dependent SET sensitivity of asynchronous logic - The muller-pipeline example. *2014 32nd IEEE International Conference on Computer Design, ICCD 2014*, 30:61–67, 2014.

- [24] P. C. Ward and J. R. Armstrong. Behavioral fault simulation in vhdl. In *Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC '90*, pages 587–593, New York, NY, USA, 1990. ACM.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX **A**

Code

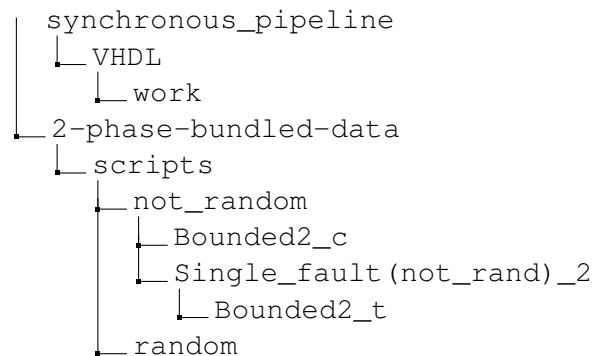
Introduction

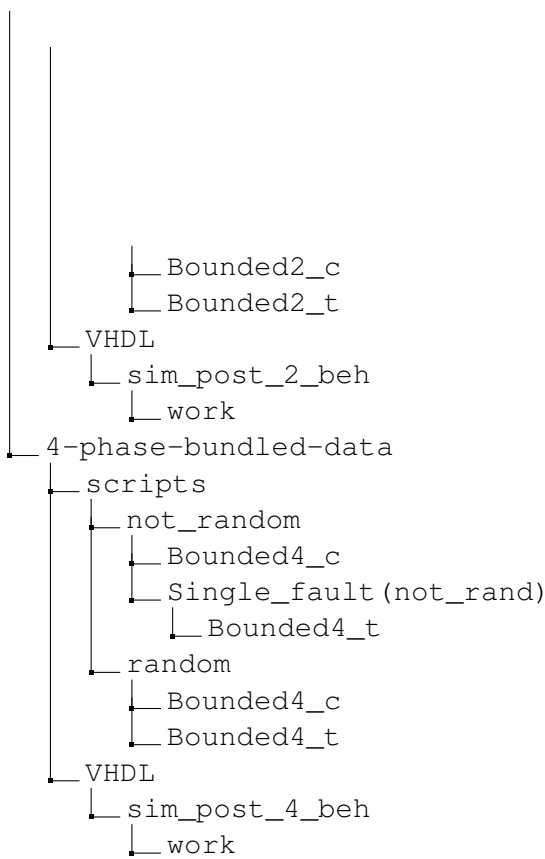
The following chapter gives an overview about the fault injection simulator, how to use it and which programs have to be installed. As a guidance for creating my behavior simulations in VHDL I found the following papers about the theory of fault injection in more detail, [10], [4] and [24].

The following list describes the environment and programs which need to be installed to use the simulator:

- Windows 7
- Python (2.7)
- Altera Modelsim
- Windows commandline

File structure





How To

The simulator is divided into three major folders, which can be run separately. The folders **2-phase-bundled-data** and **4-phase-bundled-data** contain the whole test setup for the 2-phase and 4-phase bundled data pipeline implementation. The folder **synchronous_pipeline** contains the synchronous pipeline fault injection test setup.

For the asynchronous pipeline implementations also some .bat files are created which help to start and execute the simulation in a more automated way. For the synchronous pipeline Modelsim has to be started and the working directory has to be changed to the folder where the folder work is located, then the simulation can be started. There are always two folders **VHDL** and **scripts** for each pipeline structure. **VHDL** contains all the circuit and test bench implementation, changes of circuit parameters have to be done inside here. In folder **scripts** the python code and batch scripts are stored for automated test execution.