Analyzing Asynchronous Reset Glitches during Scan-Test

Master Thesis at NXP Semiconductors

Saul Pennings

Delft University of Technology

TUDelft

Analyzing Asynchronous Reset Glitches during Scan-Test

Master Thesis at NXP Semiconductors

by

Saul Pennings

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday June 12, 2024 at 13:00 AM.

Student number:	4477502	
Thesis committee :	Said Hamdioui	TU Delft
	Chris Verhoeven	TU Delft
	Moritz Fieback	TU Delft
	Ghazanfar Ali	NXP - Semiconductors
Internship supervisor:	Richard Morren	NXP - Semiconductors

June 7, 2024

Abstract

As technology nodes continue to shrink, more challenges arise in the field of Design for Testability (DfT). Sequential Integrated Circuits (IC) with asynchronous (re)set flip-flops are notorious for producing unwanted reset behaviour during scan-test. Typically the scan flip-flops are restricted to a non-reset state while the shift operation is performed. This can be achieved by inserting an independent test signal in parallel with each local reset port. This ensures that the scan flip-flops are loaded with the correct input data during the shift cycle. However, before the capture cycle is initiated, this test signal must be released as it prohibits the reset logic from being tested with stuck-at-faults. When there are multiple cascading resets present in the design this release can cause glitches to occur. Since the reset ports operate asynchronously, these glitches can also trigger a scan flip-flop to reset, thereby changing the input data. As a result, some chips may be tested with corrupted input data, leading to differences in scan patterns. Consequently, these ICs fail during manufacturing tests and are classified as faulty, leading to yield loss.

By adding DfT to the (re)set port of each flip-flop these glitches can be prevented at the cost of additional hardware. This thesis establishes the conditions that lead to the occurrence of asynchronous (re)set glitches during scan-test. A model is proposed to find race conditions that are caused by the de-assertion of the independent test signal. Based on this model a design rule-based algorithm is presented that analyzes glitchy structures based on the number of inversions of the combinational logic between (re)set flip-flops. It accurately identifies which (re)set ports are susceptible to glitches except for circuits where there is reconvergence from the same source. As an extension, a simulation-based algorithm is presented that validates which structures are glitchy at the cost of increased execution time. Using the Electronic Design automation tool (EDA) Tessent exhaustive simulations are performed for each (re)set port to identify which (re)set flip-flops can cause glitches. These algorithms have been tested on two case studies where glitches were observed during asynchronous scan-test. After deploying these algorithms 33% and 65% of the total number of (re)set flip-flops were identified to be glitch-free. This shows that the overhead can be significantly reduced by only adding DfT hardware to the flip-flops that are susceptible to glitches. This work addresses a considerable challenge which is minimizing the cost of DfT while ensuring a robust test sequence.

Preface

I would like to express my gratitude to Moritz Fieback for his guidance and feedback on my master's thesis. I would also like to thank Said Hamdioui and Chris Verhoeven for being part of the thesis committee and reviewing my work.

I am very thankful for the opportunity provided by NXP to do my internship at the Design Enablement team. I want to extend my thanks to Ghazanfar Ali and Richard Morren for their help and guidance during my internship. Their support helped me learn a lot about the wonders of DfT and how to conduct my own research.

Lastly, I am grateful to my parents for their unconditional love and support throughout my studies.

Saul Pennings
Delft, July 2024

Contents

1 Introduction

1.1. Motivation

The semiconductor industry has experienced a consistent evolution over the past few decades, with a continuous increase in the number of transistors that can be accommodated on a single chip. As a result, testing of Integrated Circuits (ICs) has become significantly more complex, posing a challenge for the industry to conduct efficient and effective testing. While the manufacturing cost of transistors has steadily declined over the years, test costs have remained the same, as illustrated in the Figure 1.1. This trend shows that test cost have caught up with production costs; and lately the test costs of chips has been further increasing [1].

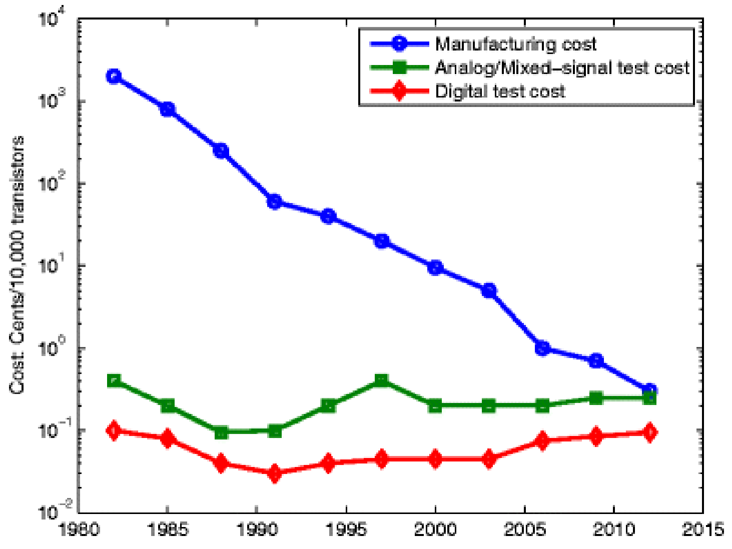


Figure 1.1: Transistor manufacturing cost vs test cost [2]

A malfunctioning chip in most of today's high-growth markets is more than an inconvenience. Chip manufacturers and large-scale computing operators are now aware of the impact of silent data errors [3, 4] in data centres operating on a massive scale. These errors have the potential to cause data loss and require extensive time and engineering efforts to resolve, which can take up several months [5]. The challenges in testing become even greater in safety- and mission-critical applications, such as automotive [6] where the quality standard is 10 defective parts per billion (Dppb) [7]. Thorough testing increases the chances of identifying all failures, but it also increases time and cost.

1.1.1. Importance of yield

A 2015 report [8] identified the effect of common cost-reduction techniques which are summarized in Figure 1.2. The technique that stands out the most is increasing the yield, which leads to a 9% reduction in total manufacturing costs for only a 1% increase in the yield. In the semiconductor industry, yield refers to the number of functional and reliable ICs produced on wafer surfaces divided by the potential number of ICs that can be made. As defined by [8], yield enhancement aims to ensure that semiconductor manufacturing is optimized to produce the maximum number of functional units by identifying, reducing, and avoiding defects and minimizing process variation that could adversely affect product output.

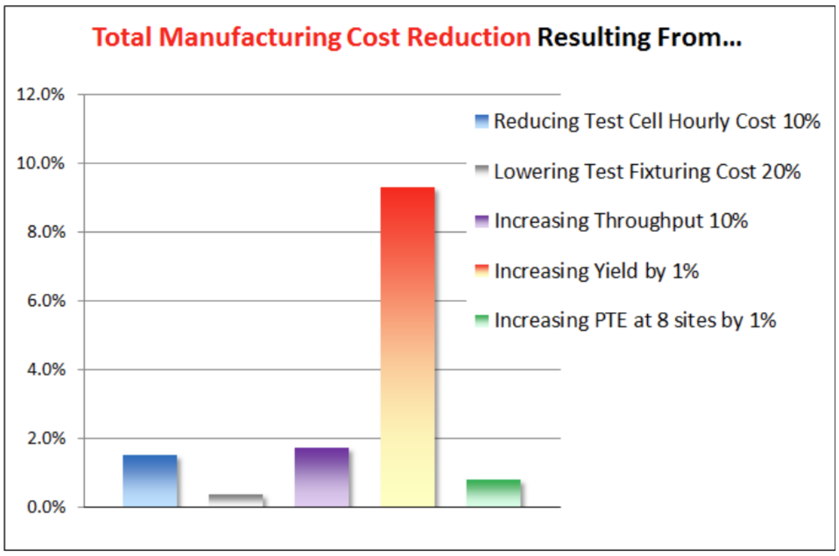


Figure 1.2: Total manufacturing cost reduction vs different cost reduction techniques [8]

Testing chips earlier is crucial due to the increasing complexity of IC designs, which makes it more challenging to detect defects through traditional methods. Additionally, testing at advanced nodes often necessitates specialized equipment and facilities, adding to the financial burden for manufacturers [9]. To realize this designers increasingly integrate more DfT techniques into the IC design phase. DfT aims to enhance the ease and efficiency of testing by incorporating features that facilitate error detection and diagnosis. The most popular DfT technique is scan test [10, 11], which allows the circuit to be put into a test mode. In this test mode, the internal memory elements between the combinational logic can be controlled and observed, making testing easier and faster.

1.1.2. Glitches during asynchronous scan test

The most common internal memory elements are flip-flops. IC designs commonly use flip-flops that can asynchronously reset or set their value to make them independent of any clock pulse. This allows a circuit to be reset almost instantly compared to a synchronous reset which can take multiple clock cycles. When millions of test vectors have to run, a couple of clock cycles can add a significant amount of test time. Asynchronous resets do however pose a challenge during scan tests. An additional signal is added that keeps the flip-flops in their non-reset state preventing them from accidentally resetting when they are put into test mode.

This signal needs to be released in order to test the logic driving the reset. As this logic can account for 2-4% of the total faults of the circuit [12] it must be tested to guarantee high-quality ICs. However, when this signal is released glitches can occur that reset the value of the flip-flops. This causes them to lose their input data which results in a mismatch between their test and expected output. Although these circuits may operate as expected in functional mode, they produce different results during scan tests, which classifies them as faulty and results in yield loss, which is very costly. Unfortunately, these glitches cannot be prevented in state-of-the-art design synthesis automation [13].

1.2. State-of-the-Art

Both [12, 13] present a solution that allows the asynchronous reset logic to be tested without the risk of glitches. This is achieved by adding additional DfT to every reset and set port in the design. However this solution is only necessary when a flip-flop is susceptible to glitches, otherwise it is just redundant logic. To identify which flip-flops are at risk additional analysis is required.

Multiple Electronic Design Automation (EDA) tools exist that can analyze circuits and detect glitchy structures. But the problem with these commercial tools is that vendors do not provide precise documentation of how their tool performs a check and many of them do not apply to contexts such as test-paths [14]. [14] tested several EDA tools and found that these tools were unable to identify glitches in multiple in-house examples. An algorithm is needed that can ensure the robustness of a circuit during asynchronous scan-test.

1.3. Contribution of the Thesis

The objective of this thesis is to find the root cause of glitches occurring during asynchronous scan-test. A model is proposed that identifies which reset and set ports are sensitive to glitches.

This thesis will analyze why these glitches can occur during asynchronous scan-test and what can be done to prevent them. Additionally, an algorithm will be proposed that can analyze a circuit based on the Netlist of an IC. The algorithm will propose which flip-flops require additional DfT to make the design robust against the asynchronous glitches during scan-test.

The main contributions of this thesis are:

- A root cause analysis on glitches occurring during asynchronous scan test. Additionally, the conditions for these glitches are outlined.
- The inversion model that identifies which (re)set ports are sensitive to glitches. This model helps to find the source of where these glitches occur and where they can take effect.
- A design rule check algorithm that can find which (re)set ports are sensitive to glitches. This algorithm checks the structure of the circuit to identify paths on which glitches can occur. This algorithm is suitable for large circuits without reconverging paths.
- A simulation-based algorithm that classifies which (re)set ports are sensitive to glitches. This algorithm is effective at identifying glitchy structures for paths with reconvergence or dependencies.
- Both algorithms have been implemented using the scripting TCL and Tessent. They have been validated in two case studies that are both glitch-sensitive. The results show a reduction of 33% and 71% DfT hardware overhead for asynchronous (re)sets.

1.4. Outline thesis

This thesis is organized in the following chapters.
Chapter 2 covers the background knowledge needed to understand this work. It introduces testing terminologies, fault models and glitches.
Chapter 3 covers the related work and state-of-the-art solutions for asynchronous reset glitches during scan test.
Chapter 4 gives a root cause analysis of glitches during scan test and explains a model to identify the source and destination of those glitches. Two algorithms are proposed to find which (re)set flip-flops are susceptible to glitches during asynchronous scan test.
Chapter 5 details how these algorithms are implemented into one detection algorithm.
Chapter 6 covers all of the experiments conducted and two case studies on which the algorithm was tested.
Chapters 7 and 8 cover the discussion, conclusion and future work.

2
Background

This chapter provides an introduction to standard testing terminology and definitions, primarily from [15, 16]. In addition, some context is given to asynchronous reset design and glitches.

2.1. Defect to Fault

In an IC a *defect* is the unintended difference between the implemented hardware and its intended design. Defects can occur during either the manufacturing process or the operation of the device. Bridging and open defects[17] are examples of some common manufacturing defects caused by undesired extra materials or the lack of sufficient material. An *error* is a wrong output signal produced by a defective system. It is the result which is caused by some defect. A *fault* is a representation of some physical defect at an abstracted function *level*.

2.1.1. Levels of Abstraction

During the design process of ICs hierarchically structured abstraction levels are needed, which allow for different views of the system. As specified by [16], these levels are illustrated in Figure 2.1. Abstraction is used to hide information that is too detailed for a specific step in the process. As you ascend through the hierarchical order the abstraction levels become more detailed and less transparent.

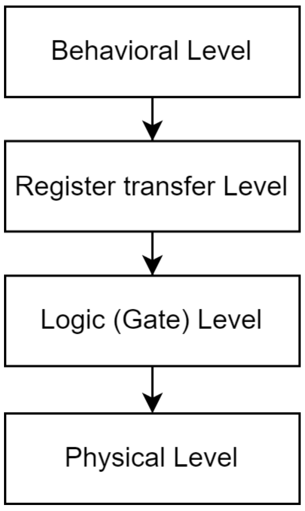


Figure 2.1: Levels of abstraction in IC design

The design process starts with defining the requirements, which leads to a behavioral level description that outlines the system's functionality. This stage specifies the IC requirements, such as speed, power consumption, and costs.

The next step is to describe the design at the Register Transfer Level (RTL), which models a synchronous digital circuit in terms of signal flow between registers and logical operations performed on those signals. In simulation, the circuit can be observed in terms of clock cycles, but there are no timing constraints like delay present yet.

At the logic, or gate level, the functionality of the circuit is described in terms of its structure, based on logic gates and storage elements. A synthesis tool uses a cell library that holds information about the available gates and their parameters based on the technology used. During synthesis the cell library is mapped onto the RTL which produces a netlist.

Finally, at the layout level, the IC is represented in terms of geometric shapes with different colours. These shapes represent the patterns of metal, oxide, and semiconductor layers that form the IC. After verification, the data is translated into an industry-standard format called GDSII, which can be handed over to IC foundries for production.

2.2. Manufacturing tests

The process of manufacturing test, also known as production test, is aimed at verifying whether the design has been accurately manufactured. Unfortunately, the fabrication process is not perfect and can result in defects in chips. By modelling physical defects as fault models, the complexity of defect behaviour analysis is greatly reduced. They allow defects to be modelled as logical faults, which can then be evaluated and tested for.

A fault model is a formal description of how a defect causes faulty behaviour in a circuit. It specifies the potential faulty behaviour and the locations at which it can occur, also known as fault locations. Fault models guide test patterns to generate the required test to excite and propagate faults to primary outputs. Fault simulation simulates the behaviour of a circuit by generating test patterns to excite and propagate faults to the primary inputs. This allows the faults to be observed in the output patterns which can be compared with the results of a fault-free circuit to determine fault detection.

2.2.1. Yield and defect level

During the manufacturing process, some ICs are expected to be faulty due to manufacturing defects. This is represented by the yield, which is defined as:

$$\text{Yield} \; = \frac{\text{Number of functional ICs}}{\text{Total number of ICs fabricated}} \tag{2.1}$$

When ICs are tested, two undesirable situations may occur. The first situation is when a faulty device appears to be a good part, passing the test. These are referred to as escapes which can be costly when they end up in the hands of the customer.

The second situation is when a good device fails the test and appears to be faulty, referred to as yield loss. The ratio of field-rejected parts to all parts passing quality assurance testing is referred to as the defect level or reject rate.

$$\text{Defect level} \; = \frac{\text{Number of faulty ICs passing testing procedure}}{\text{Total number of ICs passing testing procedure}} \tag{2.2}$$

The defect level reflects the overall quality of the ICs testing procedure. This number is generally in the order of defective Parts Per Million (Dppm), but for industries such as automotive, the target is in the order of 10 defective parts per billion (Dppb) [7].

2.2.2. Structural testing

Using fault models, the structural integrity of an IC is tested rather than its functionality. That is because exhaustive functional testing is very costly. Take for example a two-input 32-bit adder where testing every possible input combination would require $2^{32} * 2^{32} = 1.84 * 10^{19}$ test vectors, which is just infeasible.

By using structural testing, the number of test patterns required can be greatly reduced, resulting in improved test efficiency and time savings. It is important to note that structural testing does not guarantee the detection of all possible manufacturing defects, as the tests are based on specific fault models. However, it does provide a quantifiable way to measure how effectively a set of test vectors can detect certain faults, which is known as fault coverage.

2.2.3. Fault coverage

Fault coverage refers to the percentage of faults within an IC's design that can detected and identified by a given test. It is a quantitative measure of how well a given set of test vectors can find faults for a specific fault model. The equation for the fault coverage is defined as:

$$\text{Fault coverage (\%)} = \frac{\text{Number of detected faults}}{\text{Total number of faults}} * 100 \tag{2.3}$$

Achieving a fault coverage of 100% is not always feasible due to the presence of undetectable faults. An undetectable fault means that there is no possible test to differentiate between a fault-free circuit and a faulty circuit containing that specific fault. Fault coverage can be related to yield and defect level by the following equation [18].

$$\text{Defect level} = 1 - \text{yield}^{1-\text{fault coverage}} \tag{2.4}$$

Increasing the fault coverage is very beneficial for the defect level as there is an exponential relationship. Improving the fault coverage is usually less expensive and easier compared to improving the yield as yield enhancements can be costly [16].

2.3. Fault Models

Given the many different physical defects that can occur, there is no single fault model that can capture the every manufacturing defect. Therefore, test generation considers several defects and their respective fault models. Some commonly used fault models are described below.

Stuck-at Fault

The stuck-at-fault model [19] is used at the gate level, which targets physical defects that cause a circuit node to be permanently fixed to a logic high (stuck-at-1) or logical low (stuck-at-0). These defects include scenarios where a signal line is shorted to the power or ground [20]. It models faults at either the input or the output of a logic gate, but not within the circuit of the gate itself.

The stuck-at-test model does not accurately represent actual physical defects but rather abstracts the problem to a logical level, making it very effective for various testing technologies. Test vectors for stuck-at faults can detect many internal defects within logic gates, even if these defects do not correspond directly to simple physical faults like wires shorted to power or ground [21]. When test sets detect the same single-stuck-at fault multiple times very high test coverage can be achieved. [20].

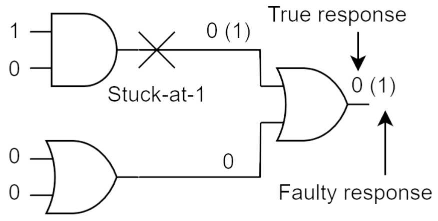


Figure 2.2: Example of single stuck-at fault

Figure 2.2 illustrates an example where there is a stuck-at-1 fault at the output of the AND gate. This means that the connection to the input of the OR gate will always be a 1, regardless of the inputs to the AND gate. As a result, the OR gate will always output a 1. The most used form of this model is the single stuck-at-fault model which assumes that only one line in the circuit is faulty.

Bridging faults

Bridging faults represent a short circuit between two or multiple signals, usually modelled at the gate or transistor level. If a signal is shorted to power or ground the stuck-at fault model can be used; however when there is a short between two signal wires a bridging fault model is required to accurately model the fault [22, 23]. Bridging faults are very common, an industrial study revealed that in a 130-nanometer IC fabrication, bridge defects and open defects make up approximately 58% of all defects [24]. Figure 2.3(a) shows a bridging fault between two shorted signals A and B. Here As and Bs denote the sources of the signals and Ad and Bd denote the destination of those signals.

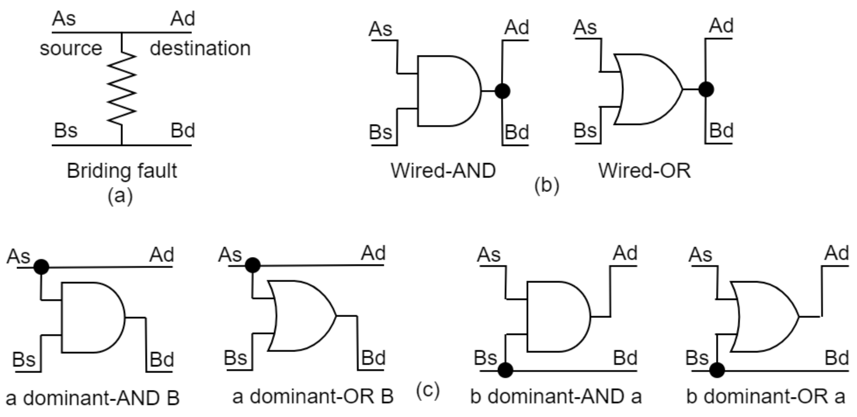


Figure 2.3: Example bridging fault models

The first bridging fault models described the behaviour of shorted lines as a logic AND or OR logic value. This model is referred to as the wired-AND/wired-OR bridging fault model [16]. Figure 2.3(b) shows the two possible faults of the model, where a wired-AND fault represents two shorted signals that will read logic 0 if either signal is driven with a logic 0. The other fault is a wired-OR which means that the shorted signal will read a logic 1 if either signal is driven with a 1.

For CMOS devices the dominant-AND/dominant-OR bridging fault model[25] most realistically models the behaviour of a resistive short [16]. In this model, one signal dominates the logic value of the other shorted values, but only for a specific logic value as depicted in Figure 2.3(c).

Transition Delay Faults

Delay faults can model physical defects at the gate level that cause additional propagation delay in the circuit. Two types of delay fault models are commonly used: the path delay fault model [26] and the gate delay fault model [27, 28]. The path delay fault model is generally considered more effective in modelling physical delay defects. But in practice, it is more difficult to use as circuit designs can have enormous amounts of paths [29]. Therefore the gate delay fault model is more suited for larger designs.

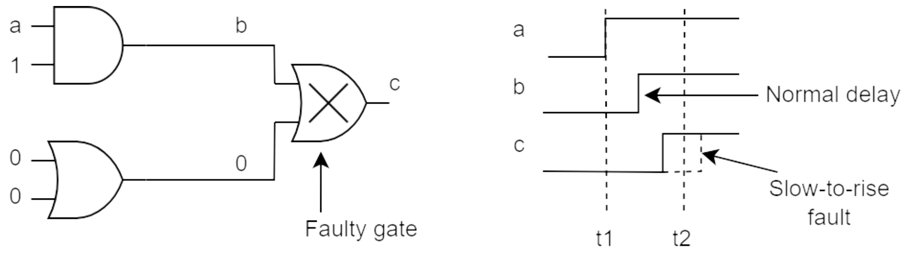


Figure 2.4: Example of a transition fault

The Transition delay fault model [28] is the most common gate delay fault model, which represents delay faults as a slow-to-rise or a slow-to-fall transition through a gate. The model assumes that for a fault-free circuit, every gate has a nominal delay except for one single gate. This gate's delay has increased so that every transition through it will exceed the clock period, even for the shortest path. Figure 2.4 shows an example where the period from t1 till t2 represents one clock cycle. The AND gate has the nominal delay which causes signal b to transition as expected. The second OR gate has a slow-to-rise fault, which causes signal c to rise after t2, which exceeds the clock period. Instead of reading a logic 1 at signal c, the output would be a logic 0.

2.3.1. Single vs multiple-fault model

Single-fault models, as the name suggests, assume that the IC only contains a single fault. This means that each circuit can have n possible fault sites which can experience k different types of faults (typically k = 2). This means that the number of single faults can be given by:

$$\text{Number of single faults} = k * n \tag{2.5}$$

In reality, multiple faults may be present in a circuit which is referred to as a multiple-fault model. For these models each fault site can have k possible faults or be fault-free, so there are (k+1) possibilities. The number of multiple faults is given below, where the "- 1" term represents a fault-free circuit.

$$\text{Number of multiple faults} = (k + 1)^n - 1 \tag{2.6}$$

Compared to the single-fault model, The multiple-fault model can accurately model more physical faults. But this precision comes at the cost of more complex test generation algorithms and impractically longer test times. Studies have demonstrated that achieving high fault coverage under the single-fault assumption correlates with high fault coverage for the multiple-fault model [15]. As a result, test generation and evaluation commonly employ the single-fault assumption for practical purposes.

Two or more faults can lead to identical faulty behaviour across all possible input patterns. These are called equivalent faults, which can be represented by any single fault from the equivalent set. This results in a substantially reduced number of distinct faults to be considered during test generation compared to the equation 2.5. Eliminating equivalent faults from the set of single faults is known as fault collapsing. The application of fault collapsing not only speeds up the test generation process but also contributes to reducing fault simulation times.

2.3.2. Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) is a widely used method that uses fault models to identify input or test sequences to distinguish between correct circuit behaviour and faulty behaviour caused by defects. Many current ATPG tools use a common approach that involves starting from a random set of test patterns. Fault simulation is then used to determine how many potential faults are detected. The results of the fault simulation are used as a guide to generate additional vectors for hard-to-detect faults to achieve the desired or reasonable fault coverage. These test vectors are generated through algorithms like D [30, 31], PODEM [32], and FAN [33].

ATPG works by injecting a fault into a circuit and then using a variety of mechanisms to activate the fault and cause its effect to propagate through the logic so it can be observed at the output. This is achievable for small combinational circuits, but for large sequential circuits, this becomes very difficult. Sequential circuits rely on both their current and previous memory state. Propagating a specific fault through the whole system would require multiple clock cycles and specific control.

2.4. Design for Testability

Design for Testability (DfT) is the general term to describe techniques to allow more comprehensive and cost-effective testing. Typically, DfT involves the incorporation of additional hardware circuits dedicated to testing purposes, which in turn allows for enhanced access to internal circuit elements. By enabling easier control and observation of the local internal state, DfT significantly contributes to the development of test programs and serves as an interface for test applications and diagnostics.

Implementing DfT rules in the development cycle yields numerous benefits, including improved fault coverage, reduced test generation time, potentially shorter test length, reduced test memory and decreased test application time. However, these advantages come with additional costs such as pin overhead, increased area and performance degradation. Nevertheless, the overall cost of the chip is reduced through DfT, making it a cost-effective methodology widely used in the IC industry [16].

For digital circuits, DfT methods include ad hoc and structured methods. Ad hoc DFT methods rely on design experience and may require circuit modification or test-point insertion to improve testability in certain areas. However, for large circuits, ad hoc DfT techniques are discouraged due to being labour-intensive and offering no guarantee of good results from ATPG. On the other hand, structured methods like scan, partial scan, BIST [34, 35], and boundary scan [11] are commonly used.

2.4.1. Scan-test

Scan design is a commonly used technique in DfT that aims to make testing easier by enhancing the observability and control of a circuit. Testing the response of a circuit requires a specific sequence of input vectors as the state of a sequential circuit depends on both the previous state and the current inputs. Scan-test allows a circuit to be transformed into a combinational circuit that relies solely on its current inputs. Regular flip-flops are replaced by scan flip-flops (SFFs), which can be switched into a test mode to create a long shift register. This shift register, also known as a scan chain, acts as a pseudo-primary input through which test data is loaded.

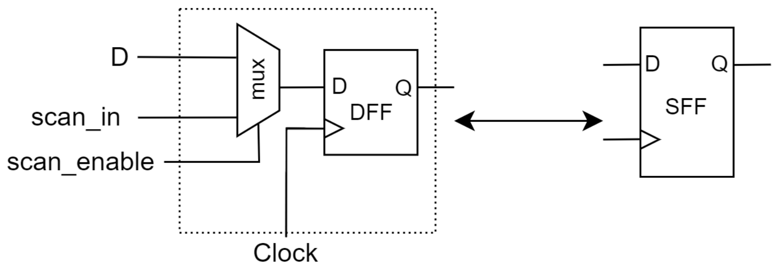


Figure 2.5: Scan flip-flop

Figure 2.5 illustrates how a scan D-type flip-flop (SDFF) is created by adding a multiplexer before the DFF. The $scan\_enable$ signal determines the mode of the SDFF. When the $scan\_enable$ is high, it selects the test mode, using the $scan\_in$ signal as input. When the $scan\_enable$ is low, it operates in the functional mode which takes the signal D coming from the combinational logic as input. In this thesis, each flip-flop is converted into an SDFF with the appropriate test signals. The test signals are not included for illustration purposes, and the SDFF is replaced by the SFF shown on the right in Figure 2.5.

The input of each SFF is connected to the output of another, creating a so-called scan-chain[36] as depicted in Figure 2.6. In this figure, the added test signals are shown in red. The first SFF in this chain is connected to a primary input ($scan\_in$) so input data can be loaded into the chain. The output of the last SFF in the chain is also connected to a primary output ($scan\_out$) so the response of the circuit can be observed.

Scan-test is split into two parts: shift mode and capture mode. During shift mode, the $scan\_enable$ is held high, forcing the SFFs to select the scan-in as their input. This turns the SFFs into a long shift register, loading the scan chain with the desired test data and bringing the entire circuit to a desired state.

During capture mode, the $scan\_enable$ is released, and the circuit is run for one clock cycle. In this way,

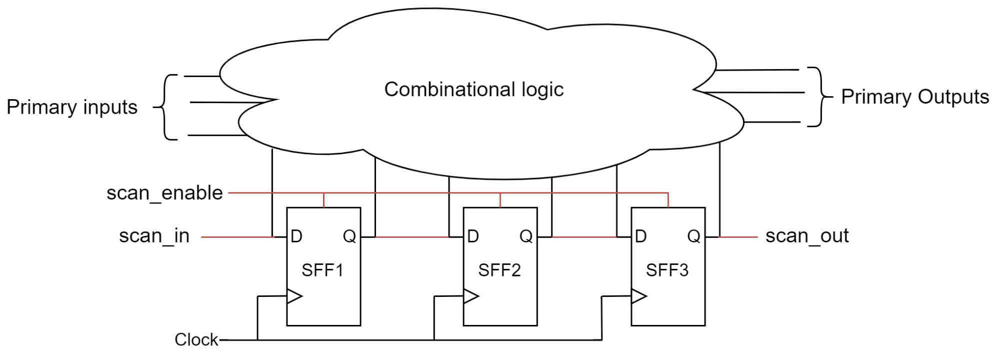


Figure 2.6: Scan chain in sequential circuits

the test data goes through the combinational logic, and the SFFs will update their data accordingly. By applying a shift cycle again, new test data is loaded in while the response of the previous test is shifted to *scan_out*. If the response of the circuit is the same as that of a fault-free circuit, then it passes the test.

2.4.2. Asynchronous scan-test

Circuits containing flip-flops with an asynchronous set and/or reset require additional control during scan-testing. During the shift operation, the flip-flops constantly change their value, resulting in random values being fed into the combinational logic. While the flip-flops are in test mode, they will take the scan-in value, so this is not an issue for the D-input. However, there is also logic feeding into the asynchronous set or reset port that can be randomly triggered. If this occurs during the shift cycle, the test data of that flip-flop would be set or reset, which alters the test data.

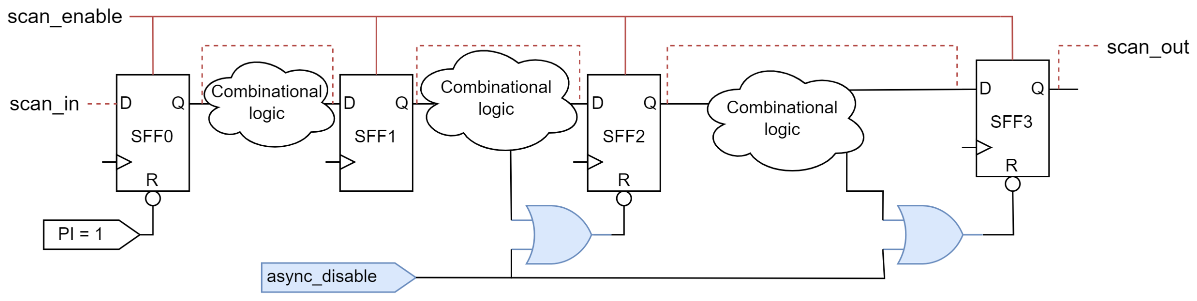


Figure 2.7: Scan chain for flip-flops with asynchronous reset

To prevent this, an *async_disable* signal is added to the design, which is gated with every set or reset port that is not controlled by a primary input. Figure 2.7 gives an example for flip-flops that are active low, so their reset is triggered when a 0 is received. In this figure the test signals added for scan-test are depicted in red and the logic required for asynchronous scan-test is shown in blue. This figure shows how the reset ports of SFF2 and SFF3 are connected to the *async_disable* signal, which is held at logic 1 during the shift operation. Note that the reset of SFF0 is connected to a primary input and thus does not require to be connected to the *async_disable*. This is because primary inputs are also controlled during ATPG, therefore they can be held at logic 1.

If the *async_disable* is kept at 1 during the capture stage, then the logic feeding into the reset ports cannot be tested. Therefore, it must be released before the capture stage is initiated. It is common practice to split up the scan-test into two separate tests: a synchronous scan-test, where the *async_disable* is kept at 1 during both shift and capture, and an asynchronous scan-test where the *async_disable* is kept at 1 during shift and released before the capture cycle to allow the reset logic to be tested.

2.5. Reset signal

For digital ICs the main purpose of the reset signal is to force the device into a known and predefined state before running tests and simulations. Apart from that most applications also require the reset during power-up, when a watchdog timer expires, or for the user (or another IC) to manually restart the system. This reset can be either synchronous or asynchronous depending on the design of the IC.

The reset functionality of an IC serves as a critical aspect of its overall design, influencing both reliability and performance. Essentially, the reset mechanism ensures that the IC begins operating in a known and stable state, vital for correct system initialization before simulations are run. Commonly there are two main approaches for resets: asynchronous and synchronous. This section explains the pros and cons of both resets.

Asynchronous Reset

An asynchronous reset provides an immediate response to the IC without being bound to clock edges. It is implemented by adding a pin to the design of flip-flops which sets the internal value of the flip-flop to a known state. This ensures that the data path is clean as the reset does not add any gates or net delay to the combinational logic [37].

Without the dependency of a clock, the asynchronous reset can be asserted immediately, making it very fast. However, there is a catch. If the reset is de-asserted at or near the edge of the active clock it could cause a flip-flop to go metastable[38]. This happens when a flip-flop's setup and hold time requirements are violated. In a metastable state the output of a flip-flop is unknown, so neither a 1 nor a 0 for an unpredictable amount of time before settling to one value. A solution for this is using a reset synchronizer circuit to ensure that the reset is de-asserted at the appropriate time.

Another disadvantage is false resets that can be triggered due to glitches or noise on the reset signal. Whether this is a serious issue depends on the design and requirements of the system. A possible solution is to add a reset glitch filtering circuit that removes short pulses from the reset input signal [37].

For DfT, designers need to make sure that the asynchronous reset can be directly accessible. During scan-test, the reset must be blocked to prevent flip-flops from accidentally resetting which would result in a loss of test data.

Synchronous Reset

Synchronous reset aligns with specific clock edges, ensuring controlled and predictable timing behaviour within the IC. This synchronization to clock cycles facilitates a more structured approach to resets, simplifying design verification and reducing the likelihood of race conditions.

Flip-flops with a synchronous reset can be synthesized more efficiently as the reset is gated with the logic generating the d-input. This technique does save some area, but only about one or two gates per flip-flop. Nowadays, the die sizes are already huge, so this area saving is not that significant[37]. Additionally, since the reset is just another data input it may introduce additional delay to the data path, which is undesirable. As a synchronous reset only happens at the active edge of a clock, it ensures that the whole circuit is 100% synchronous and filters out small glitches. However, glitches that occur near the active clock cycle can still cause flip-flops to go metastable.

For low-power designs, the requirement of a clock to reset the system can be a disadvantage. For instance, when clock gating is implemented to save dynamic power dissipation. Clock gating is a technique that keeps the clock signal stable, preventing flip-flops from switching state and consuming power. Without a clock, it is impossible to assert a reset, and thus, only an asynchronous reset can be used in that case.

Table 2.1 presented below provides a comprehensive comparison of asynchronous and synchronous resets, focusing on various aspects such as timing characteristics, reliability, and implications for DFT. This information aids in understanding the trade-offs associated with each reset approach in the realm of IC design.

From a design perspective, an asynchronous reset is usually preferred as it allows a circuit to be reset without a clock present. Since it is asynchronous, there is minimal latency, allowing the circuit to be

Aspect	Asynchronous Reset	Synchronous Reset
Implementation	Add additional pin to flip-flop. Requires special care for DfT.	Gate reset signal with d-input of flip-flop.
Timing Characteristics	Provides immediate response. Not tied to clock edges.	May introduce additional gate delay. Only acts on active clock edge.
Reliability & Hazards	Sensitive for glitches & metastability.	Less susceptible to glitches.
Area & Power	Permits clock-gating for less power consumption.	Requires less area due to due to smaller flip-flops.

Table 2.1: Different aspects of using an asynchronous vs synchronous reset

quickly brought to a desired state compared to a synchronous reset which can take multiple clock cycles. It also enables using clock-gating, making it more suitable for low-power designs. For DfT, an asynchronous reset brings some challenges as it needs to be controlled during scan-test. Inserting an independent $async\_disable$ test signal can prevent the flip-flops from resetting ensuring a correct shift phase. However, de-asserting this signal can produce glitches in designs with multiple cascading resets, which is a challenge that must dealt with by the DfT engineers.

2.6. Glitches

Glitches are unintentional and temporary changes in logic values that can occur in digital circuits. They manifest as brief pulses or spikes on digital signals and can potentially cause incorrect circuit behaviour. For synchronous paths, glitches will not impact the functionality as long as the timing path between flip-flops meets the required setup and hold timings. But for asynchronous paths glitches can accidentally trigger other paths resulting in unwanted behaviour.

Glitches can be caused by physical phenomena, such as electromagnetic interference, capacitive coupling, or high-energy particles from cosmic rays, and are referred to as transient faults [39, 40]. Additionally, glitches can occur due to race conditions, which arise when a logic gate combines signals that have travelled through different and unequally timed paths from a single source [41].

A piece of circuitry where race conditions can potentially lead to a glitch is called a hazard. The two common types of hazards are static and dynamic hazards [41]. In a static hazard, a signal momentarily switches while it should have remained at a stable logic value. In dynamic hazards, a signal oscillates while it should have made a clean transition. Figure 2.8 shows the three kinds of hazards that can occur.

- **Static 0-hazard**: Signal temporarily changes to 1 when it should have stayed at 0.
- **Static 1-hazard**: Signal temporarily changes to 0 when it should have stayed at 1.
- **Dynamic hazard**: Signal changes multiple times while it should have transitioned once.

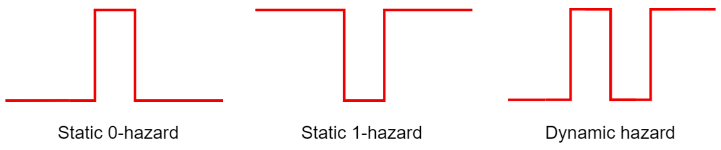


Figure 2.8: Categories of hazards in digital circuits

When a circuit contains a hazard, it means that there is a possibility of a glitch occurring under certain circumstances. These circumstances can be difficult to predict as they depend on various factors, such as the inputs to the circuit, gate delay, and noise. Uncertainties in gate delay [42] are caused by several factors, including manufacturing process variations [43, 44], environmental variations, and device ageing phenomenon [45].

3

Related work

3.1. Problem of asynchronous scan test

While performing scan test with asynchronous logic a separate signal is needed to prevent the flip-flops from accidentally resetting, which is the *async_disable* signal. This signal is kept high during shift mode but must be released during capture mode; otherwise, the asynchronous reset logic cannot be tested [12]. When this signal is released multiple reset sources are enabled at the same time, which can cause glitches.

There is a lack of work in the literature on the topic of asynchronous reset scan-test. However, that does not mean that glitches do not impact the semiconductor industry during these tests. Take for example the DfT challenges that were faced during the design of the ARM Cortex-A15 Microprocessor [46]. In this design, the internal resets are asynchronously asserted and synchronously de-asserted. A test circuit was added to allow at-speed testing of those reset paths. They mention that this signal must remain at a single value during capture cycles to prevent potential reset glitches from occurring. Because the reset is synchronously de-asserted the reset cannot propagate asynchronously from one flip-flop to another, hence for this design glitches will not be observed. But it is with these reset ripple architectures that glitches have a high chance of occurring.

Classification of ripple reset glitches

In [47] a couple of clock-gating structures are described that can impact the robustness of scan test. This includes the reset ripple architecture which can suffer from glitches due to reconvergence of an *async_disable* signal.

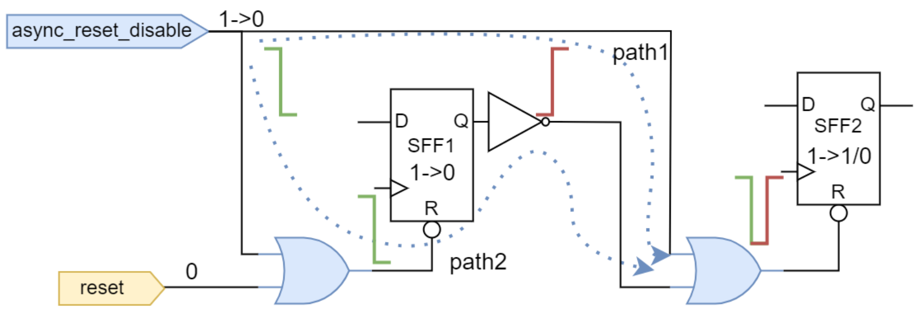


Figure 3.1: Ripple reset structure with a potential glitch

Figure 3.1 shows how glitches can occur when it is released. The signal diverges and takes two paths to reach the OR gate feeding into the reset port of SFF2. One path goes directly to the OR gate, while the other goes through SFF1 and changes polarity in the process. When the signal reconverges there is a static 1-hazard at the output of the OR gate. If a glitch occurs it will trigger the reset of SFF2 changing its value from 1 to 0, while it should remain at 1. However, whether this glitch occurs depends on the propagation delay of the two paths which varies from chip to chip [42].

The variation in propagation delay could result in a difference in scan patterns. Chips that produce a different result during asynchronous scan test would then be classified as faulty, while in functional mode they might work as intended. To prevent these glitches from impacting the yield a solution is needed to make the asynchronous scan test more robust.

3.2. State-of-the-art solutions

This section will cover the state-of-the-art solutions for glitches during asynchronous scan test. The first two solutions propose additional DfT to prevent glitches, followed by Electronic Design Automation tools (EDAs) that could be used and an algorithm based on a Design Rule Check (DRC).

3.2.1. Additional DfT to prevent glitches during scan test

The effect of glitches on the (re)set paths can be mitigated by preventing flip-flops from resetting each other in a cascading fashion. One method to achieve this is through test point insertion, which involves adding extra circuitry to the design [12]. Instead of having one *async_disable* signal that enables all flip-flops at the same time, the signal is split into several unique signals. These signals are released one by one, allowing the glitches to settle before the next flip-flop is reset. To create these unique *async_disable* signals, a test point is added between each cascading reset flip-flop. The *async_disable* is gated with each test point, which consists of an OR gate and another flip-flop. These flip-flops are 1-cold decoded, which means that only one flip-flop holds the value 0 while the rest hold a logic 1.

This solution is appropriate for designs that have only a few cascading flip-flops. However, for structures like the ripple reset architecture, this approach will become impractical. In addition to the extra hardware required for each flip-flop, the test time for these designs will significantly increase. Since the flip-flops are 1-cold decoded, they need to be released one after another for every test vector during ATPG.

In the patent [13] this issue is resolved by adding an OR-AND-OR gate in combination with a flip-flop per local functional reset. This enables the test of the reset logic to be split into a synchronous and asynchronous capture phase. During the synchronous capture phase, the logic that generates the local reset signals is tested as illustrated in Figure 3.2, where the dotted box represents the additional hardware. During this phase, the tcb_*async_disable* signal is held high, which prevents the flip-flops from resetting, just like during the shift cycle. The func_local_rst_an is the local reset path that originates from other reset flip-flops. Normally, this path is blocked by the *async_disable* signal, but now the value is captured by the additional flip-flop. Note that this is the d-input of the flip-flop rather than a (re)set port. As this flip-flop only captures the value during the clock signal, it will not be affected by any glitches that may occur.

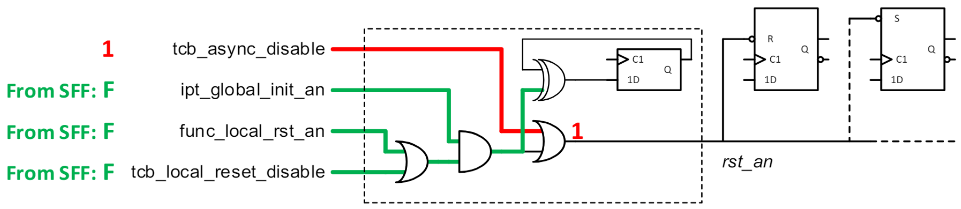


Figure 3.2: OR-AND-OR structure with highlighted signals during synchronous scan mode [13]

During the asynchronous capture phase, the asynchronous behaviour of the reset flip-flops themself is tested, as illustrated in Figure 3.3. The tcb_local_reset_disable is another added test signal blocking the local (re)set path as these could produce glitches. Instead, the ipt_global_init_an is used which is a glitch-free global reset. This divide-and-conquer strategy allows all the logic to be tested with just two tests without the risk of glitches due to the cascading asynchronous reset signals.

Both [12] and [13] allow the asynchronous reset logic to be tested at the expense of additional hardware. However this solution is only necessary when a flip-flop is susceptible to glitches, otherwise it is just

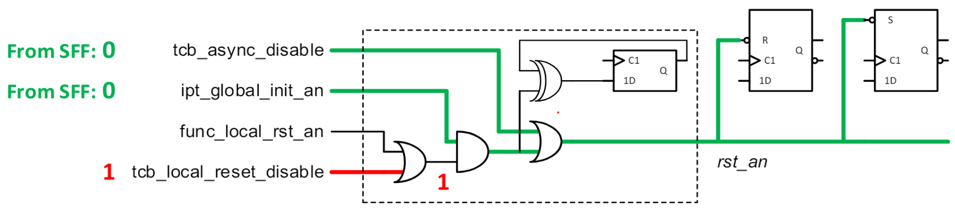


Figure 3.3: OR-AND-OR structure with highlighted signals during asynchronous scan mode [13]

redundant logic. To identify which flip-flops are at risk additional analysis is required.

3.2.2. EDAs performing glitch analysis

Various Electronic Design Automation tools (EDAs) exist that can assist in different steps of the ICs design. Ideally, the glitches are found early on in the design as making changes later on in the design process is more expensive. In [48] the capabilities of several RTL verification tools are explored. The area most closely related to glitches on reset paths is called Reset Domain Crossing errors (RDC). These errors occur when a reset signal crosses from one clock domain to another during a reset event, which can lead to potential errors due to timing mismatches and metastability. Examples of EDAs that identify these errors are Synopsys Spyglass RDC [49], Real Intent Meridian RDC [50] and Siemens Questa RDC [51].

In a research study [14], several EDA tools were evaluated for their effectiveness in detecting glitches in clock-domain-crossing (CDC) circuits, a closely related area to RDC. The study found that these tools were unable to identify glitches in multiple in-house examples. The researchers attributed this to the lack of a comprehensive explanation from vendors about how the algorithms and tools work. They also noted that many CDC checks are specific to certain contexts and do not apply to other scenarios, such as test-paths.

Another issue with these EDA tools is that they often report false errors which need to be manually checked. This can lead to engineers overlooking actual errors, which cause actual errors to be missed [52]. That is why designers cannot rely on these tools alone to find glitches in these areas.

DRC for ripple reset structures

A simple static DRC algorithm is proposed in [47] to find structures where the $async\_disable$ signal reconverges at the gate feeding into a reset port, as illustrated in Figure 3.1. This is done by taking each (re)set port as a starting point and trace back through the combinational gates and (re)set ports. When a fan-in gate has been visited twice then that gate is a reconvergent fan-out point which will be flagged as a DRC violation. This is a very pessimistic approach as reconvergence on the $async\_disable$ signal is not the only requirement for a glitch.

One major disadvantage of this algorithm is that paths are only traversed if they have simulated values. These values depend on the test setup and will not always be present. As a consequence, this algorithm would not check every path and could thus miss some glitchy structures. To ensure the whole circuit is robust a more general approach is needed that checks every path regardless of simulated values. As pointed out in their paper, this static DRC does not take logic dependencies into account, which can lead to false positives.

On top of this algorithm, a dynamic ATPG-based DRC analysis is performed which uses ATPG to check if a solution exists. As ATPG algorithms are not capable of detecting glitches [47], three conditions are checked instead. If one of these conditions is met then it is a potential violation, otherwise it is a false positive.

In total five static DRCs and dynamic ATPG DRCs were performed on 3000 test cases. Each case had a circuit size ranging from 10K to over 30M gates. 49 circuits were flagged with static DRC violations for ripple reset glitches, with an average of 8.6K in those 49 cases. Notable is that the dynamic ATPG DRC did not lead to an improvement as it flagged the same amount of cases. For some cases, the Verilog simulations were checked and they confirmed that there were some glitches asserted for those ATPG patterns.

For the ATPG DRC the amount of violations found is not reported, only the amount of cases that had ATPG DRC violations. Whether or not the ATPG DRC resulted in fewer or the same false positives is not clear from these results.

To prevent glitches found by the DRC violations [47] suggest blocking the reconvergent path. For example, defining the input to the reset logic, $reset$ pin in Figure 3.1, as a clock input and setting it to the off-state. This solution is not feasible when the reset is produced locally from another flip-flop, which is always the case in a ripple reset architecture.

3.2.3. Limitation and shortcomings

To prevent glitches from affecting scan test patterns the most suited solution is to add additional DfT. In most cases, patent [13] would be the best solution as it results in only a doubling in test time of the asynchronous scan test. The advantage of this solution compared to the one proposed by [47] is that the functionality of the design does not need to be altered. Instead, some hardware needs to be inserted at the local (re)set ports.

This hardware is only needed when there is a risk of glitches on the (re)ports, so the additional area can be minimized by an algorithm that can find all potential glitchy structures. Relying on EDAs alone to find these glitchy structures will not be sufficient as [14] found multiple examples where they were unable to do so. Using DRC to find these structures is a valid method, but the algorithm proposed by [47] relies on having simulated values that might not always be present. Apart from that the algorithm is also very pessimistic as it only checks for reconvergence of the $async\_disable$ signal, which is not the only condition for a glitch.

This thesis will conduct a root cause analysis on how glitches can occur during asynchronous scan-test. These findings result in three conditions that must be met for a glitch to occur. A model is proposed that utilizes these conditions to identify the source and destinations of these glitches.

Based on this model two algorithms are proposed that can classify which (re)set flip-flops are susceptible to glitches. One is a design rule check algorithm that investigates the structure of a circuit. This algorithm can identify on which paths glitches can occur. The second algorithm performs local simulation to identify glitchy behaviour between (re)set flip-flops.

4 Analysis and Detection Algorithm

This chapter will analyze the root cause of glitches that appear during asynchronous scan-tests. This analysis will explain what causes these glitches and what the conditions are to consider a (re)set port sensitive to glitches. To better understand the problem, the "inversion model" will be proposed, which encapsulates the conditions of a glitch. Based on this model, a detection algorithm will be proposed that can analyze a circuit given a Netlist. This algorithm is divided into two parts: the inversion propagation check and the inversion simulator. The inversion propagation check is a design rule check based on the inversion model. The inversion simulator performs local simulation and uses these results to identify glitch-sensitive (re)set ports.

4.1. Analysis

After the $async_{/}disable$ is released any flip-flop can be reset which in turn can then reset other flip-flops as well, starting a sort of cascading reset chain. When the first flip-flop in this chain (re)sets it will update the data that it was holding, which is expected behavior during the asynchronous scan-test. Except for the $async_{/}disable$ signal, all other inputs to that (re)set port are static so no glitches can occur.

However, after the first flip-flop in this cascading reset chain is (re)set, the other flip-flops now have at least two sources that change value. That is the reset flip-flops feeding into its (re)set port and the $async_{/}disable$ signal switching to 0. This can lead to race conditions which can then cause glitches to appear at the (re)set port. As stated before, whether or not glitches will be observed depends on the propagation delay of the different paths in the circuit.

4.1.1. Hazards on asynchronous (re)set ports

In this thesis, the asynchronous (re)set ports of flip-flops are all active low. When a 0 is received at the reset port, the value of the flip-flop immediately goes to 0, irrespective of the clock value. Similarly, when the set port is triggered, the value of the flip-flop goes to 1. The analysis will also hold for active high (re)set flip-flops, but in that case the polarity of the glitches and the inversions on paths is inverted.

For asynchronous active low reset ports, only static 1-hazards need to be considered. These cause the reset signal to momentarily drop to 0, triggering a reset. In the case of static 0-hazard, the (re)set signal momentarily goes to 1, but these do not affect the value of the flip-flop as it is already in a (re)set state. Similarly, when the reset signal transitions from 0 to 1 or from 1 to 0, the value of the flip-flop will remain equal to its (re)set value. Therefore they are unaffected by any number of dynamic hazards. This coincides with findings from [47] which only considers static 0-hazards as they use flip-flops with an active high reset.

This work focusses on identifying race hazards caused by the de-assertion of the $async_{/}disable$ signal. In this context, a race hazard implies the potential for a glitch without considering other factors like propagation delay or process variation. While factoring in these elements would provide a more precise representation of the possibility of a glitch, it would also introduce significant complexity to the inversion model.

4.2. Conditions for an asynchronous (re)set glitch

To find the conditions for a glitch Figure 4.1 is used where the $async_{/}disable$ along with the OR gates (blue in the Figure) are inserted at the reset ports of flip-flop 1 and 2. The cloud representing combinational logic could be any combination of logic gates.

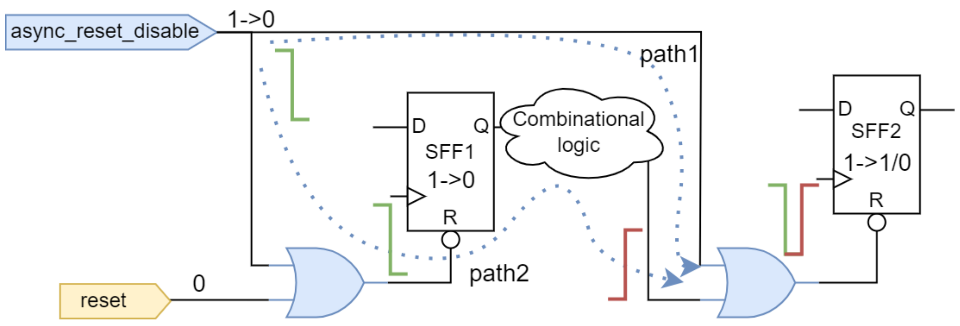


Figure 4.1: Conditions for a static 1-hazard

During shift mode of scan-test the $async_{/}disable$ signal is kept at 1 which prevents the reset ports of flip-flop from going to a logic 0, thus no glitches can happen during this phase. In capture mode the $async_{/}disable$ signal is released and transitions from 1 to 0. For a glitch to occur at the reset port of flip-flop 2 the signal must first go to 0 (green in Figure) which means that the combinational logic feeding into the OR gate must be a 0 as well.

For the signal to go to 1 again (red in Figure), some signal in the circuit needs to transition. As the clock and input ports are kept constant it can only be caused by another flip-flop resetting, which is flip-flop 1 in this case. Note that this reset is triggered by the $async_{/}disable$ signal going to 0. When flip-flop 1 gets reset its output value goes to 0 while for a glitch to occur a 1 is required at the reset port of flip-flop 2. This means that in the combinational logic, an inversion of the signal must take place.

From this analysis, there are three conditions for a glitch on a (re)set port of a flip-flop:

1. An $async_{/}disable$ signal transitions from a 1 to a 0 while all other input signals remain static. Which occurs during the capture event of scan-test.
2. The reset of some flip-flop gets triggered which is connected to the (re)set port of the flip-flop where the glitch is observed.
3. Through the combinational logic in between the two flip-flops the polarity of the signal is inverted.

Note that these glitches do not occur in functional mode as they are initialized by the release of the $async_{/}disable$ signal.

4.2.1. Assumptions asynchronous disable signal

It is important to make some assumptions for this analysis. Firstly, it is assumed that the $async\_disable$ signals and the OR gates are inserted 'correctly'. This means that each local reset feeding into the (re)set ports is connected with an OR gate where the other input connects to the $async\_disable$ signal. If the OR gate is missing at a port then the scan-shift will not operate correctly. On the other hand, if the $async\_disable$ feeds into a random part of the combinational logic then there could be reconvergence on the disable signal itself, which can also lead to glitches.

Secondly, it is assumed that the $async\_disable$ signal itself is glitch-free and makes a clean transition from 1 to 0. This can be achieved by using an external pin as a global reset or by something like a test control block that produces a global reset. Glitches on this signal will not be taken into account in this analysis.

Third, it is assumed that the propagation delay of the $async_{/}disable$ signal is smaller than that of the reset path. Figure 4.2 shows the waveform of two test cases of the signals from Figure 4.1. Let us assume that the reset delay of the flip-flops is t_{reset} and the propagation delay through the combinational logic is t_{comb}.

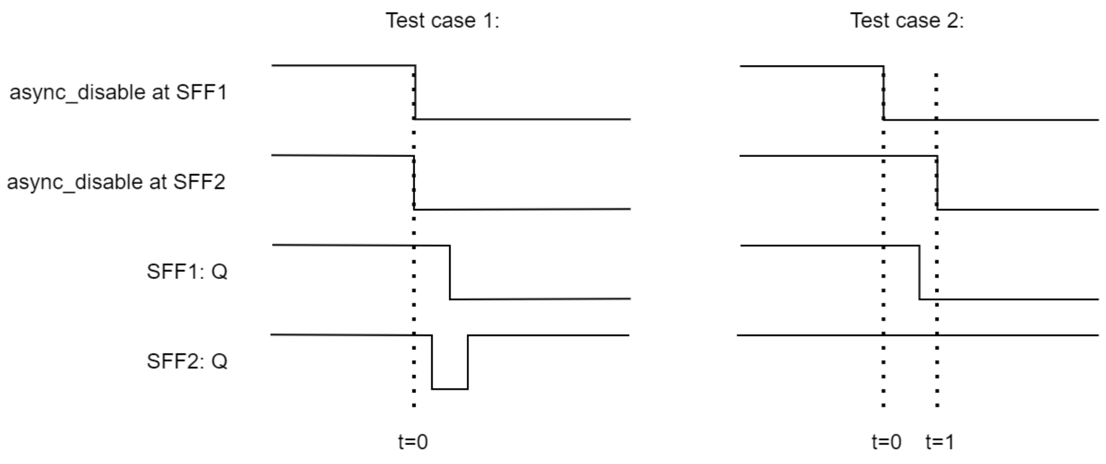


Figure 4.2: Propagation delay on $async_{/}disable$ signal

In test case 1, there is no delay between the release of the $async\_disable$ signal at SFF1 and SFF2. This causes the output of SFF2 to update first after t_{reset}, and then again after $t_{reset} + t_{comb}$, resulting in a glitch. In test case 2 there is some propagation delay on the $async\_disable$ between SFF1 and SFF2, which is called t_{async}. As a result, the $async\_disable$ at SFF2 is released after t = 1. If $t_{async} > t_{reset} + t_{comb}$, the glitch is already settled before the $async\_disable$ signal at SFF2 is released.

In other words, as long as the propagation delay of the $async\_disable$ is larger than the combined delay of the flip-flop resetting and the combinational logic then no glitch will be observed. The objective of this thesis is to improve the circuit's robustness against potential glitches. To achieve this, the worst-case scenario is considered, which is to assume that there is no propagation delay on the $async\_disable$ signal.

Depending on the design there can be one or multiple $async\_disable$ signals. For example, there can be one $async\_disable$ signal for the set ports and another for the reset ports of flip-flops. For this analysis, it does not matter if one or multiple signals are used, its result will be the same.

4.3. Inversion model

This section will introduce the 'inversion model' which helps identify whether a (re)set port is susceptible to glitches. In the first part, some definitions will be established which are highlighted in different colours in Figure 4.3. The model describes the connection between scan flip-flops as paths where the (re)set ports are the destination and the output ports of (re)set scan flip-flops are the sources.

In the second part, it will be explained how these destinations can be evaluated for the possibility of glitches.

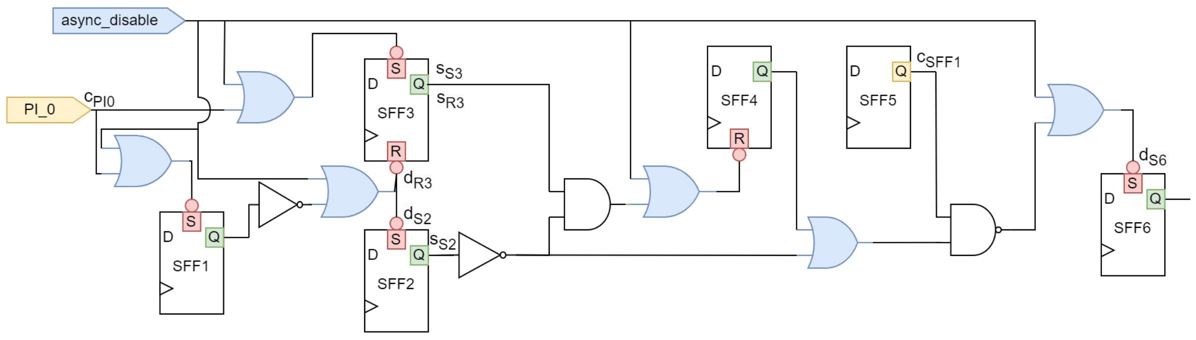


Figure 4.3: Example of the inversion model

4.3.1. Definitions

Destinations (Red)

Every reset and set port of a scan flip-flop is called a destination, which is the point where a glitch will take effect. One flip-flop can have both a set and a reset and can therefore have at most two destinations.

- d_S = Set port of a flip-flop.
- d_R = Reset port of a flip-flop.
- Ex: d_{S2}, d_{R3}

Sources (Green)

Every output port Q of a scan flip-flop with a set and/or reset is called a source. The source is coupled to the (re)set port of that flip-flop as triggering a reset or set will give a different value at port Q. This means that a flip-flop with both a set and a reset will have two different sources that are connected at the same point.

- s_S = Output port Q of a flip-flop with a set port.
- s_R = Output port Q of a flip-flop with a reset port.
- Ex: s_{S2}, s_{S3}, s_{R3}

Control signals (Yellow)

The output of every scan flip-flop without a (re)set is called a control signal. These are signals that do not change their values during the capture event of asynchronous scan-test. Primary inputs that are ATPG controllable are also control signals as they are static as well.

- c_{SFF} = Output of a scan flip-flop without a set or reset port.
- c_{PI} = Primary inputs that are ATPG controllable.
- Ex: c_{SFF1}, c_{PI0}

Async disable signals (Blue)

The $async_/disable$ signal and related logic are represented in blue.

Paths

Path $p(x, y)$ is the connection from a source (x) to a destination (y) through some combinational logic, for example, $p(s_{S2}, d_{S6})$. Each path is either inverting or non-inverting which is calculated based on the inverting logic gates that that are in that path. A path can have one of three values: 1 if it is inverting, 0 if it is non-inverting and X if the path is potentially inverting.

- Inverting path = A path with an odd number of inversions = value 1.
- Non-inverting path = A path with an even number of inversions = value 0.
- Potentially inverting = A path with an undetermined number of inversions = x

$D_y = \{x1, x2, ...\}$ is a collection of all sources that have the destination y. For example destination d_{S6} has the sources s_{S2} and s_{R4} thus: $D_{S6} = \{s_{S2}, s_{R4}\}$

4.3.2. Inversion on paths

Before determining when a path is inverting let's first look at how a glitch itself will propagate through it. Whether or not a glitch propagates through some logic gate will depend on the other input values of that gate. Figure 4.4 illustrates how glitches propagate through each primitive logic gate. For an AND/OR gate, a glitch only propagates when its other input is a 1/0 respectively and it keeps the same polarity.

It can be assumed that the other input values of each logic gate are set in such a way that a glitch will propagate. If the input values are not set correctly, the glitch will not show up at the destination. This reasoning also applies to NAND/NOR gates, but in those cases, the polarity of the output is inverted. On the other hand, an inverter, having only one input, will always propagate a glitch and change its polarity.

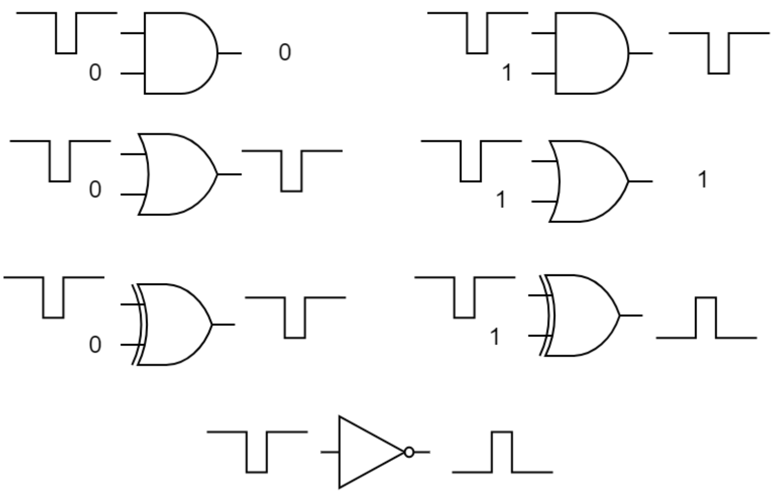


Figure 4.4: Glitches propagating through different primitive logic gates

Apart from logic gates, a source itself can also be inverting which are the flip-flops with a set source, so s_S. Flip-flops are active low (re)set, which means that when the set receives a 0, the output is set to 1, essentially inverting the signal as well. Note that if the flip-flops in the design are active high, then the flip-flops with a reset source are inverting.

The sum of every inverting logic gate and inverting source is taken to determine if a path is inverting (odd number of inversions) or non-inverting (even number of inversions). For example, consider the path $p(s_{S2}, d_{S6})$ in Figure 4.3. The set source, inverter, and NAND gate are all inverting, adding up to 3 inversions in total. Hence, $p(s_{S2}, d_{S6}) = 1$ is an inverting path.

XOR/XNOR gates

Figure 4.4 shows that an XOR will propagate a glitch regardless of its input. On top of that it inverts the polarity of the glitch if the other input reads a 1. The opposite is true for an XNOR gate which will invert and propagate a glitch when the other input reads a 0. When there is at least one XOR or XNOR gate in a path then its value is set to X, this indicates that the path is potentially inverting.

4.3.3. Glitch-sensitive flip-flops

To determine whether a destination is susceptible to glitches, it must meet the three conditions outlined in Section 4.2. The first condition is that the $async_{/}disable$ signal should be 0, which is always the case during asynchronous scan tests.

The second condition is that the (re)set of another flip-flop, called the source, should trigger the (re)set port of the flip-flop where the glitch is observed, called the destination. This can happen when a destination is connected to at least one source.

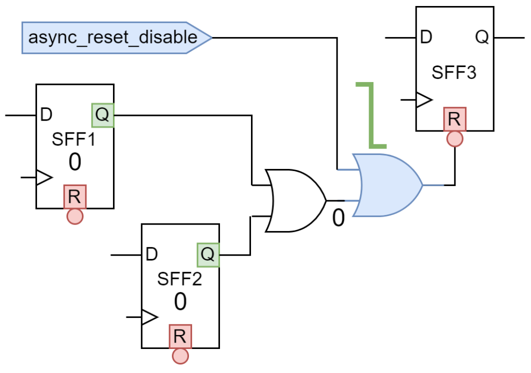
The third condition is that the polarity of the signal between the two flip-flops should be inverted. This is the case when a path between the source and destination is inverting. Thus, for a destination y, at least one path from collection D_y must be inverting.

Since the $async_{/}disable$ signal is always 0 during asynchronous scan tests, it can be left out of the analysis. Whether or not a flip-flop is glitch-sensitive will depend on the other two conditions. So, if a destination has a source and its path is inverting, then that destination is considered glitch-sensitive. This means that at that destination, in other words, the (re)set port, there is a static 1-hazard.

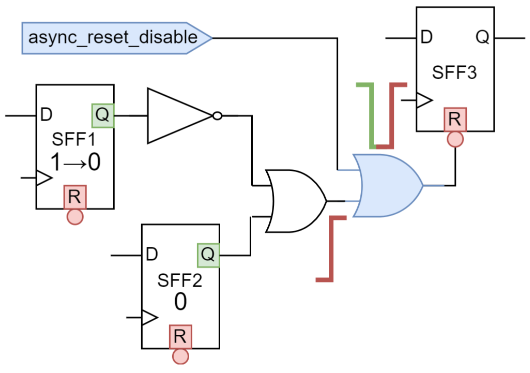
4.3.4. Single source transition model

According to this model, every glitch results from a single flip-flop being set or reset. This might sound counterintuitive as the output of a logic gate depends on both inputs. Even if two flip-flops are involved in initiating a glitch, it can always be traced back to a single flip-flop with an inverting path. This is because a glitch will only propagate through a gate if a transition at the input results in a transition at the output, see Figure 4.4

Figure 4.5a shows a destination with 2 reset sources. After the release of the $async_{|}disable$ a glitch can only occur if the output of the white OR gate is first 0 and then transitions to 1. Its output can only be 0 if both sources hold the value 0 and in that case, triggering either reset will not change anything in the circuit.



(a) Multisource without inversion



(b) Multisource with inversion

Figure 4.5: Single source transition

For the OR gate to transition from 0 to 1 at least one of the paths needs to be inverted. In that case, a reset can be triggered to generate a glitch, illustrated in Figure 4.5b. It could also be that both paths are inverted in which case both flip-flops must be 1 initially, otherwise no glitch can be created. Resetting one flip-flop will then result in a glitch, but afterwards triggering the other reset will not change anything in the circuit as the OR gate will remain at 1, thus it can no longer propagate a glitch.

4.4. Detection Algorithm

This section will explain how these glitch-sensitive destinations can automatically be detected. This is split up into two different algorithms. The first is called 'inversion propagation check' and the second one 'inversion simulator'. The inversion check performs a DRC that identifies which destination and sources there are and calculates which paths are inverting. The inversion simulator performs a local simulation for each destination and based on those results determines which sources are inverting.

4.4.1. Inversion propagation check

The inversion propagation check algorithm will be explained using the inversion model, defined in Section 4.3. Figure 4.6 shows the example circuit that will used to illustrate how the algorithm operates in which only the signals of interest are shown. The left-out signals are all of the D inputs, which depend on the circuit's functionality, and all of the signals needed to put the flip-flops into scan mode.

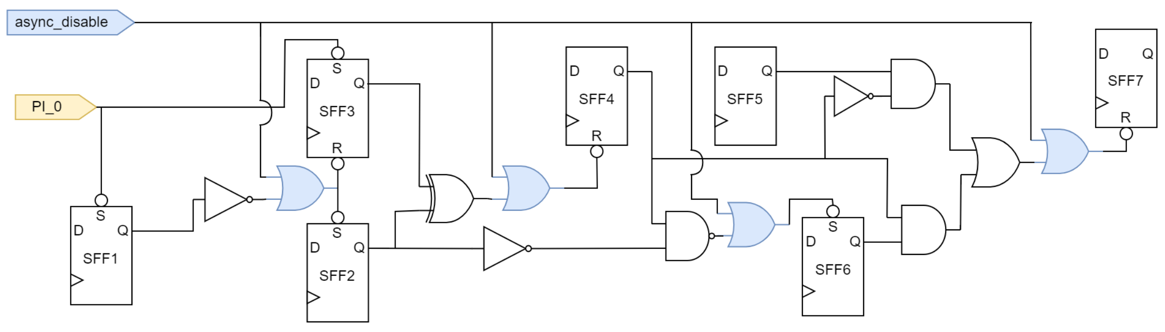


Figure 4.6: Example circuit used as input for detection algorithm

Identify destinations and sources

The first step of the algorithm is to identify every destination and source that is in the circuit. This is done by checking every flip-flop in the scan-chain and classifying each (re)set port as a destination. The output Q of each (re)set flip-flop is set as a source, where each source is linked to either a set or reset. In case a flip-flop has both a set and reset then two separate sources are created. Figure 4.7 shows the circuit after this step of the algorithm.

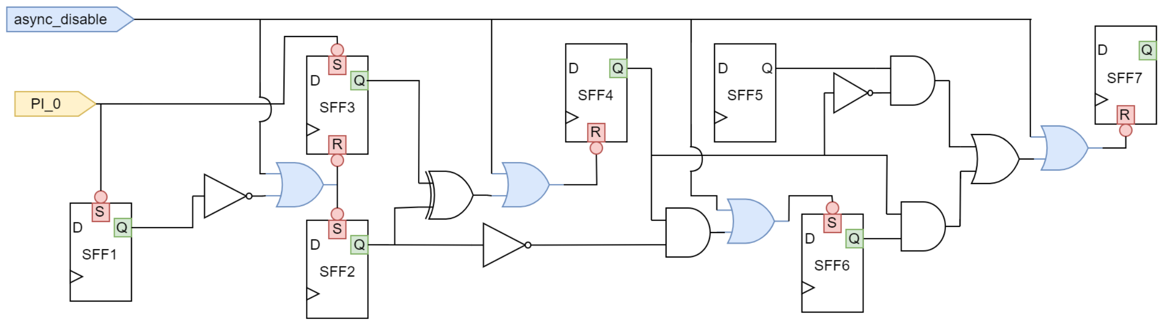


Figure 4.7: Example circuit after identifying destinations and sources

Trace back for each destination

Every path leading to a destination is traced backwards until it ends either at a primary input or at the output of a scan flip-flop. If the endpoint is a source, then the path is saved. For each saved path, we calculate the number of inversions based on the combinational gates on the path and the type of source.

Take SFF6 as an example, the destination d_{S6} is connected to two sources: s_{S2} and s_{R4}. Path $p(s_{S2}, d_{S6})$ consists of the following gates: OR, NAND, INV. This means there are 2 inverting gates (NAND,INV) on this path plus one inversion due to source s_{S2}. This gives a total of 3 inversions, which is odd, thus $p(s_{S2}, d_{S6}) = 1$ as it is inverting. The same calculation can be done for $p(s_{R4}, d_{S6})$ which is a reset source, no inversion, with OR plus NAND gate, so $p(s_{R4}, d_{S6}) = 1$ as well.

The path $p(s_{S1}, d_{S2}) = 0$ as it consists of a set source and an inverter, which is an even number of inversions, thus this is an example of a non-inverting path. Path $p(s_{S2}, d_{R4})$ contains an XOR gate so this path gets the value X indicating that it is potentially inverting, which will be further explored by the inversion simulator algorithm.

On the combinational logic between d_{R7} and s_{R4} there is an example of reconvergence. It will depend on the implementation of the algorithm whether the path with or without the inverter will be picked. Thus this path also needs additional exploration to determine if $p(s_{R4}, d_{R7})$ is inverting or not. For now let us assume this path is set to be non-inverting

Identifying glitch-sensitive scan flip-flops

After every path for each destination is compiled the algorithm can identify which scan flip-flops are glitch-sensitive. These are the destinations that have at least one source with an inverting path or potentially inverting path. The other destinations are seen as safe in the context of glitches during asynchronous scan tests. By combining all of the calculated paths the following collection for each destination can be made:

- $D_{S2} = \{s_{S1}\} = \{0\}$
- $D_{R3} = \{s_{S1}\} = \{0\}$
- $D_{R4} = \{s_{S2}, s_{S3}, s_{R3}\} = \{X, X, X\}$
- $D_{S6} = \{s_{S2}, s_{R4}\} = \{0, 0\}$
- $D_{R7} = \{s_{R4}, s_{S6}\} = \{0, 1\}$

The number in brackets denotes the sources that have a path that is: non-inverting (0), inverting (1), or potentially inverting (X). Since at least one of the sources for d_{R7} is inverting, this scan flip-flop is considered to be sensitive to glitches. For now, d_{R4} is also considered to be glitch-sensitive since it has at

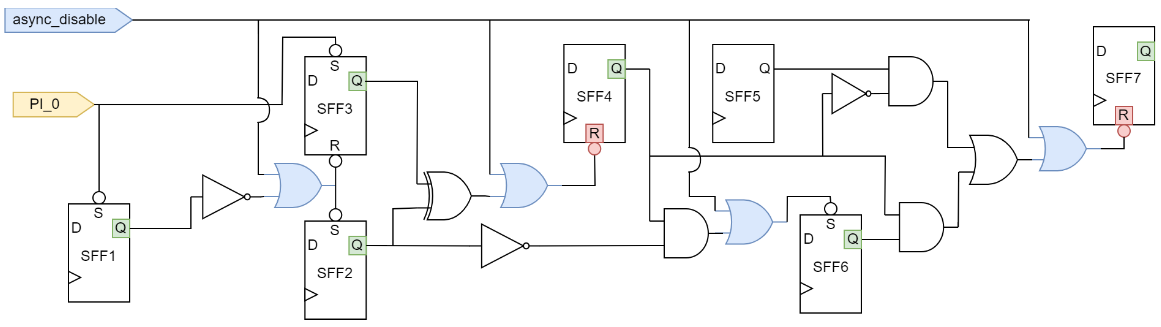


Figure 4.8: Glitch-sensitive scan flip-flops (in red) after inversion propagation check is performed

least one potentially inverting path. Figure 4.8 highlights only the destinations of such glitch-sensitive scan flip-flops in red.

4.4.2. Inversion simulator

The inversion simulator algorithm will further investigate which scan flip-flops are glitch-sensitive by performing local simulations. As input, it takes the destinations that are classified as glitch-sensitive by the inversion propagation check. For each destination, all scan flip-flops connected to it are identified. Using these as inputs the algorithm simulates every possible combination of input values and records the resulting value at the destination. Table 4.1 shows the simulated values for d_{R7} as an example. If one (re)set port has N inputs the algorithm needs to simulate 2^N values, which means it has an exponential time complexity [53].

$$\text{Time complexity for one (re)set port} = O(2^N) \tag{4.1}$$

This means that the executing time of the algorithm grows quadratically with the size of the input. For circuits that have a large fan-in on the reset logic the number of inputs N might become too large (>8) which will drastically increase the execution time of the algorithm.

Cycle	SFF4	SFF5	SFF6	Sim Result
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Table 4.1: Simulation results for d_{R7}

Determine inverting sources from simulation

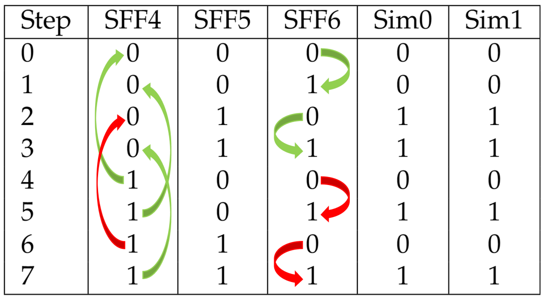
The algorithm performs an exhaustive search by checking every possible input transition that can be made. Based on the simulation results, if a source exhibits inverting behaviour for at least one input transition, then it is classified as inverting.

Now, let's consider the scenario where an input is also a reset source. If only this input transitions from a 1 to a 0, while the other inputs remain the same, then it mimics the behavior when that source is reset. If the simulation result changes from a 0 to a 1 for that input transition, then it indicates that a glitch could occur for those input values. This is the same behaviour as shown in Figure 4.1.

A reset source is considered non-inverting only when a transition from 1 to 0 does not cause a transition from 0 to 1 at the destination. On the other hand, set sources transition from 0 to 1 when their set port is triggered. Therefore, a set source is non-inverting only when a transition from 0 to 1 does not result in a transition from 0 to 1 at the destination. To verify this, the algorithm examines each input and checks the simulation results when only that input is changed.

A reset source is considered non-inverting when no transition from 1 to 0 leads to a transition from 0 to 1 at the destination. On the other hand, set sources transition from 0 to 1 when their set port is triggered. Therefore, for a set source the same applies when the input transitions from 0 to 1 instead of from 1 to 0.

Table 4.9 displays how the algorithm checks each input, where red arrows indicate inverting transitions and green arrows indicate non-inverting transitions.



Step	SFF4	SFF5	SFF6	Sim0	Sim1
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	1	1
3	0	1	1	1	1
4	1	0	0	0	0
5	1	0	1	1	1
6	1	1	0	0	0
7	1	1	1	1	1

Figure 4.9: Simulation results checked by algorithm

Take s_{R4} as an example, in Table 4.9 from cycle 6 to 2 and from 7 to 3 only input SFF4, the source, is changed from value 1 to 0. The simulation result changes from 0 to 1 for these input values, thus s_{R4} is inverting.

Output after inversion simulator

After performing both algorithms the following collection of destinations is made:

- $D_{S2} = \{s_{S1}\} = \{0\}$
- $D_{R3} = \{s_{S1}\} = \{0\}$
- $D_{R4} = \{s_{S2}, s_{S3}, s_{R3}\} = \{1, 1, 1\}$
- $D_{S6} = \{s_{S2}, s_{R4}\} = \{0, 0\}$
- $D_{R7} = \{s_{R4}, s_{S6}\} = \{1, 1\}$

The values for d_{R4} were updated by the inversion simulator as each of its sources was inverting. This is quite trivial as the logic feeding into this destination was only an XOR gate. However, in more complex circuits, this may not always be the case. It is worth noting that path $p(s_{S4}, d_{S7})$ was set to non-inverting by the inversion propagation check, while the inversion simulator deemed it as an inverting path. As mentioned earlier, the inversion propagation check does not work reliably for circuits with recovering paths, which is why both algorithms are used to ensure that false negatives are filtered out.

5 Implementation detection algorithm

This chapter covers how the inversion propagation check and inversion simulator are implemented in a detection algorithm. The exact requirements for the detection algorithm and how it was integrated into the design flow are confidential and have thus been left out of the thesis. Instead, the general design flow of a chip is described to explain at what stage the algorithm could be implemented. The remaining sections explain how the algorithm works using a flow chart and elaborate some important functions.

5.1. Integration Detection algorithm

Asynchronous glitches during scan design are observed through differences in ATPG simulations, which occurs near the tape out of the chip. This is very late in the design cycle, making changes to the design very costly. Hence, a detection algorithm is needed to identify these glitches early in the design phase.

From the chips design requirements functional Register Transfer Level (RTL) code is created, which describes the functionality of a circuit. During synthesis, library cells are mapped onto the RTL which produces a gate-level netlist. This netlist describes the components and structure of the circuit. At this stage, DfT such as scan-chains, can be added to the netlist. After everything is added the layout of the netlist is finalized after which it is ready for tape-out.

The algorithms require the structure of the circuit to calculate inverting paths. For this reason, RTL cannot be used, as only after synthesis the structure is determined. Additionally, scan-chains should already be inserted as this makes it easier to identify each flip-flop. The layout does not need to be finalized yet for the algorithm to work so that it can be used on either the pre- or post-layout netlist depending on the application. Running it on the pre-layout would be advised as the algorithm highlights which scan flip-flops require additional DfT, which should be inserted before finalizing the layout.

5.1.1. Coding languages

To analyse the circuit the EDA Tessent tool from Siemens is used. It is specifically designed to address the challenges of testing complex digital integrated circuits (ICs) at different stages of the IC lifecycle including design and verification. Some key aspects of the tool are high-quality ATPG, fault simulation and analysis, and support for various DfT techniques such as scan test. Many functions are included to analyze paths in a circuit, gather information on the scan-chain, and perform simulations.

In combination Tcl, short for "Tool Command Language", is used which is a flexible scripting language. Tcl is an interpreted language, which means its executed line by line without needing compilation. It is a basic but flexible language which makes it very suited for automating tasks.

5.2. Detection algorithm code

This section explains from a high-level view what the detection algorithm does. It will be described using the flowchart in Figure 5.1. The inversion propagation check and inversion simulator will be discussed in their separate subsection which is also referenced in the flowchart. The functionality of some important Tessent functions will be explained, these are italicized.

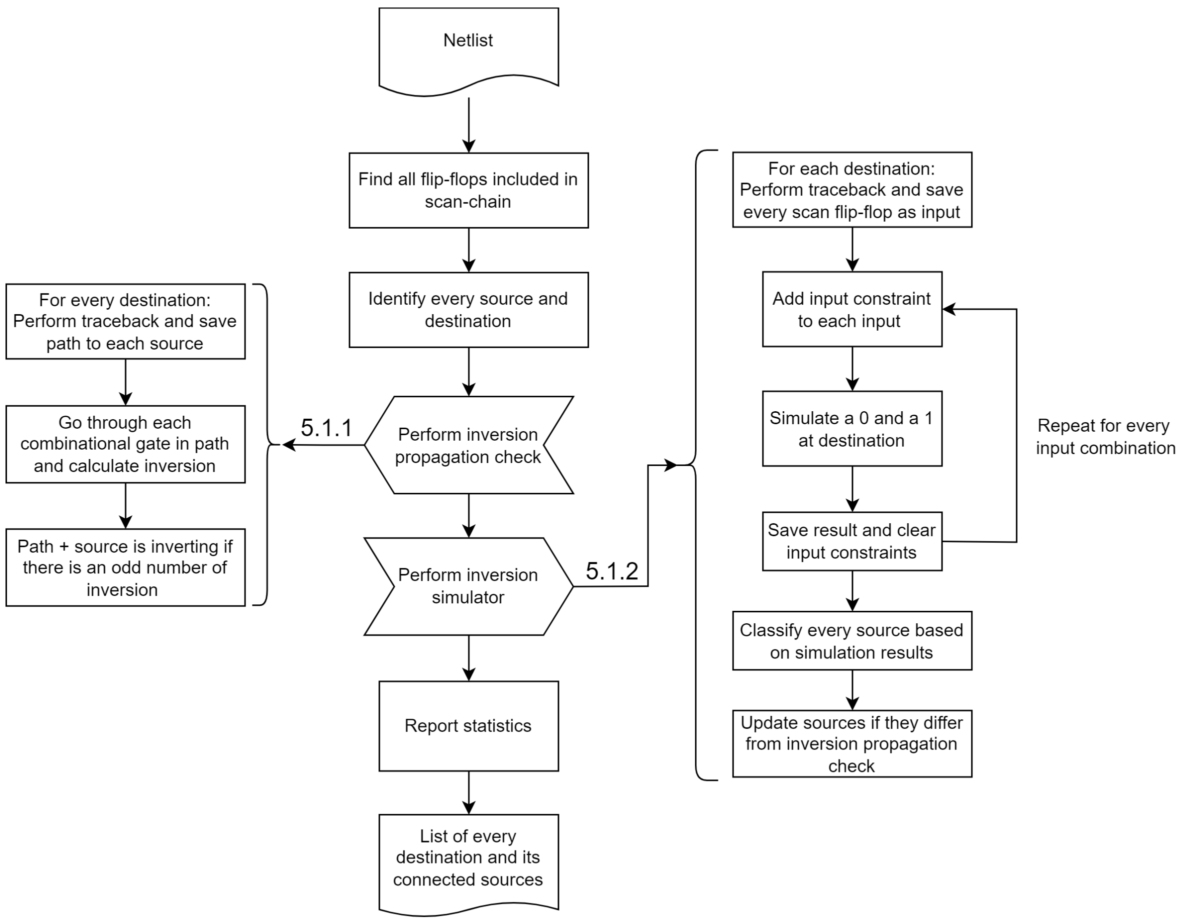

Netlist
Find all flip-flops included in scan-chain
Identify every source and destination
Perform inversion propagation check
Perform inversion simulator
Report statistics
List of every destination and its connected sources
5.1.1
For every destination: Perform traceback and save path to each source
Go through each combinational gate in path and calculate inversion
Path + source is inverting if there is an odd number of inversion
5.1.2
For each destination: Perform traceback and save every scan flip-flop as input
Add input constraint to each input
Simulate a 0 and a 1 at destination
Save result and clear input constraints
Repeat for every input combination
Classify every source based on simulation results
Update sources if they differ from inversion propagation check

Figure 5.1: Flowchart of detection algorithm

After Tessent loads in the netlist the function *set_system_mode analysis* is run to allow design analysis, test pattern generation and simulation. Then Tessent report every flip-flop that is part of a scan-chain (there could be multiple scan-chains in a design) and finds every flip-flop that has a set or reset port.

Before every (re)set port there should be an OR gate inserted which is gated with the asynchronous reset disable signal. The other input to this OR gate is set as a destination (d_{Sx}, d_{Rx}), which prevents Tessent from traversing to the logic generating the asynchronous reset disable. The output Q of every (re)set flip-flop is set as a source (s_{Sy}, s_{Ry}).

Using the destinations and sources as an input the inversion propagation check is performed which generates a list with every path, both inverting and non-inverting. The inversion simulator then uses these paths as input to simulate and classify each source. Some statistics are reported such as the total amount of flip-flops, the number of sources and destinations, and how many paths were reclassified by the inversion simulator. As an output two text files are generated. One that includes: every destination, which source is connected to it and whether it is inverting or not, this is illustrated in the Appendix Figure A.1. The other text file only contains the destinations that have at least one inverting source, as these destinations require additional DfT.

5.2.1. Inversion propagation check

The function of the inversion propagation check is described on the left in Figure 5.1. For each destination, the Tessent function $trace\_flat\_model$ is used, this traces back through the combinational logic until it encounters a source or an input.

The parameter 'direction' is set to backward as Tessent traverses the logic from the destination back to the source. The parameter 'controllability' is set to connected which ensures that Tessent will traverse any path that is connected to the destination. The optional parameter 'store_path_to_tagged_objects' saves the traversed path which includes each primitive combinational gate that is in between the source and destination.

Another function calculates if the traversed path is inverting or not. If the traversed path has an odd number of inverting gates then the source is inverting (1) and if it is even then it is non-inverting (0). If an XOR or XNOR gate is in the path then it is set to potentially inverting (X). For each destination, a list is stored which contains which source it is connected to, its inverting value, and the path length. The path length is the amount of gates that is between the destination and the source.

5.2.2. Inversion Simulator

The function of the inversion simulator is described on the right in Figure 5.1. To explain this part of the code the example from Figure 4.8 will be used where destination d_{R7} is being checked. For each destination, Tessent again performs a trace back but this time stores every scan flip-flop that it encounters and sets them as inputs, resulting in Table 5.1. The left-hand column gives the simulation step of the simulation which is created using a for loop from 0 to 2^{inputs}

Step	SFF4	SFF5	SFF6	Sim0	Sim1
0	0	0	0		
1	0	0	1		
2	0	1	0		
3	0	1	1		
4	1	0	0		
5	1	0	1		
6	1	1	0		
7	1	1	1		

Table 5.1: Simulation inputs for d_{R7}

A snippet of this part of the code is shown in Listing 5.1. Each input is given an index from 0 to the number of inputs, which means the following: SFF6 (0), SFF5 (1), SFF4 (3). When the step is represented in binary we get the corresponding input values. At each step another for loop goes through each index and performs an AND operation on the step and 2^{index}. In other words, if the simulation step is 3 this represents 011 in binary and 2^{index} is: SFF6 (001), SFF5 (010), SFF4 (100). So for step 3 SFF6 and SFF5 are put to 1 while SFF4 is put to 0.

Listing 5.1: TCL code: Simulating input value

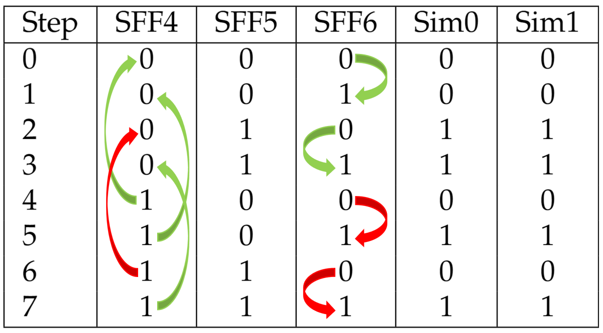
```
for {set step 0} {$step < [expr 2 ** $length]} {incr step} {
           for {set index 0} {$index < $length} {incr index} {
               if {[expr $step & 2 ** $index] != 0} {
                   set bit c1} else {set bit c0
                   }
               add_cell_constraints [get_gate_pins [lindex $sources_connected $index]] $bit
               }
               report_test_stimulus -set $dest_input 1 > $file2
               set sim_1_output($step) [ReadSim_1 $file2]

               report_test_stimulus -set $dest_input 0 > $file2
               set sim_0_output($step) [ReadSim_0 $file2]

               delete_cell_constraints -All
           }
       }
```

At the start of each step, the value of each scan flip-flop is set using the Tessent function *add_cell_consraints*. After each input is loaded its response is tested using the Tessent function *report_test_stimulus*. This function allows a specific node in the circuit, the destination, to be set to either a 1 or 0. Tessent then attempts to simulate that value, and if successful, it reports which scan flip-flops were used to do so. This function is run for both a 1 and a 0 with the same input constraints. These constraints are then deleted before performing the same code on the next destination.

The column Sim0 in Table 5.2 displays the results when a 0 is simulated at d_{R7}. A result of 0 indicates a successful simulation, while a result of 1 indicates that the simulator could not simulate a 0. The same applies to Sim1, where a result of 1 means the simulation was successful, and 0 means it could not simulate a 1.

Classifying sources

Once all simulations are run, the algorithm checks both simulation values using the same two for loops shown in Listing 5.1. If only a set source changes its input value from 0 to 1, and this results in the simulation value also changing from 0 to 1, then that source is identified as inverting. Similarly, when a reset source changes from 1 to 0, and this results in the simulation value changing from 0 to 1, it is identified as inverting. Figure 5.2 illustrates how this checking is done, where the green arrows indicate a safe transition, and the red arrows indicate an inverting one.



Step	SFF4	SFF5	SFF6	Sim0	Sim1
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	1	1
3	0	1	1	1	1
4	1	0	0	0	0
5	1	0	1	1	1
6	1	1	0	0	0
7	1	1	1	1	1

Figure 5.2: Simulation results from inversion simulator

Utilizing Tessent simulator

The reason for computing both Sim0 and Sim1 is related to how the Tessent simulator operates. The function name *add_cell_constraints* suggests that scan-cells are restricted to a specific value. However, during simulation, these values can be overwritten by triggering the (re)set port of a scan-cell. This gives the simulator significant control as it can modify the input data. By simulating both a 0 and a 1 the reset behaviour and the true response of the circuit will be captured.

Take Figure 5.3 as an example, which is an AND gate with two reset sources. Performing both simulators for d_{R3} results in the table illustrated in Table 5.2.

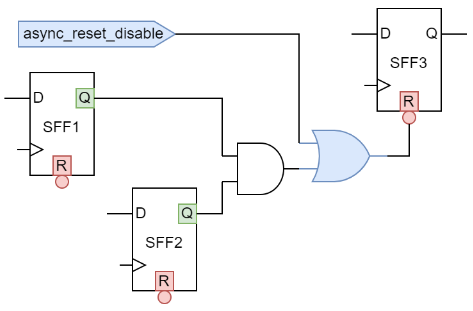


Figure 5.3: AND gate with two reset sources

For Sim0 the simulator always gives a 0, while a 1 is expected when the input values are SFF1 (1) SFF2 (1). But remember that the simulator will try to simulate a 0 at d_{R3}, which is possible by triggering either one of the reset ports. However, when simulating a 1 this is not possible and the true response is found.

Step	SFF1	SFF2	Sim0	Sim1
0	0	0	0	0
1	0	1	0	0
2	1	0	0	0
3	1	1	0	1

Table 5.2: Simulation results for AND gate

Because the simulator can also control the (re)set ports of the inputs the simulation values will sometimes differ and thus both need to be checked. If this could be prevented in Tessent then only one simulation would be necessary, but this is not possible in the current state.

6 Experiments and results

This chapter covers the results of all the experiments conducted. The first part shows very basic structures to showcase the capabilities and limitations of both algorithms. An overview of the results is given at the end of this section.

In the second part some functional structures are shown where there is reconvergence from the same source. These experiments highlight where the capabilities of the inversion simulator.

The third part covers two case studies on which the algorithms were tested. In these case studies asynchronous reset glitches have been observed. In the last section benchmarks are performed on both algorithms.

6.1. Experimental setup: Basic structures

The RTL of the basic structures has been written in Verilog. The RTL has been compiled using Genus Synthesis Solution using the library cells specified by NXP-Semiconductors. These are the intellectual property of NXP Semiconductors and will not be discussed.

The structures have been build without taking their functionality into account. During the optimization phase of the synthesizer, these structures could be altered or even completely removed. To prevent this the combinational logic has been written as a netlist using the gates defined in the library of NXP. In the synthesis script set_dont_touch was specified for each gate preventing the synthesis tool from optimizing the logic.

The first set of experiments is a set and reset source directly connected to a destination, which represents the simplest configuration possible. This experiment is then repeated with an inverter placed between the source and destination. These experiments demonstrate how the simulator functions for sources with both an inverting and non-inverting path.

The second set of experiments shows how both algorithms deal with different logic gates. These are the primitive 2-input logic gates: OR, NOR, AND, NAND, XOR and XNOR. There are already buffers present in between the combinational logic which have been placed by the synthesizer. So these primitives gates are indirectly tested as well although they do not affect either algorithm.

In the third set the output of a source is connected to the (re)set port of another and both outputs converge through a 2-input logic gate. In these experiments, there are dependencies within the circuit which results in different behaviour. The fourth set looks at reconvergence coming from the same source.

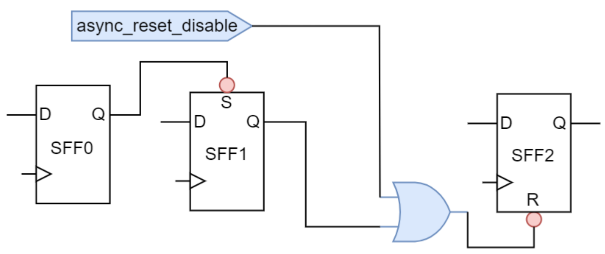
The fourth set shows some structures where there is reconvergence from the same source. These experiments These structures require at least two logic gates, meaning that at least $8 * 8 = 64$ basic structures are possible to create. Testing them all would be very time-consuming, so instead some structures are hand-picked to show the capabilities of the algorithm.

These structures have been simulated using a simulator in Tessent. As explained before, this simulator essentially has too much control as it can alter the input data set by the inversion simulator algorithm. By triggering the (re)set ports of these flip-flops the test data is overwritten which can mess up the simulation data. For the inversion simulator to work properly this should be avoided. Additional experiments are run where the (re)set ports of the sources are kept at logic value 1. This prevents Tessent from altering the input data to show how the algorithm would perform if the simulator was working as intended.

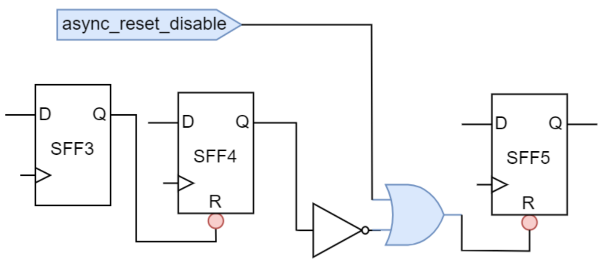
The top of table shows which inputs are used. Sources which have their (re)set port fixed to logic 1 are indicated by a *. The sources classified as inverting by the inversion simulator are highlighted in bold in each table. Any sources that are incorrectly identified will be explored.

6.1.1. Directly connected sources

Figure 6.1a shows the simplest structure that can be built, which is a source directly connected to a destination. Figure 6.1b shows a source connected to a destination through an inverter. Both structures have been tested for both a set and reset source as shown in Table 6.1 and 6.2.



(a) Test: Directly connected source without inverter



(b) Test: Directly connected source with inverter

Input	**Set**		Reset	
SFF1	Sim0	Sim1	Sim0	Sim1
0	0	1	0	0
1	1	1	0	1

Table 6.1: Simulator values without inverter

Input	Set		**Reset**	
SFF3	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1
1	0	0	0	1

Table 6.2: Simulator values with inverter

Both algorithms classify s_{S1} and s_{R4} as inverting, which is the expected result. Looking at table 6.1 for the set source, there are only 1s simulated for Sim1. Similar behaviour is observed when the input is a reset source but with inverted polarity. That is because the simulator can control the input by putting a 0 on SFF0 which would trigger the set of SFF1 and overwrite the input value. When multiple sources are used the simulator has a lot of control and alters the input data.

6.1.2. 2-input logic gates

This subsection outlines the test cases conducted for each 2-input logic gate: the OR, NOR, AND, NAND, XOR, and XNOR gate. Figure 6.2 depicts the structure tested for an OR gate with a set and reset source as input. The same tests were conducted for two reset sources and two set sources which all show different simulation results. The sources indicated by a * have their (re)set port tied to 1. This prevents the simulator from resetting the input values, which leads to the actual response of the circuit.

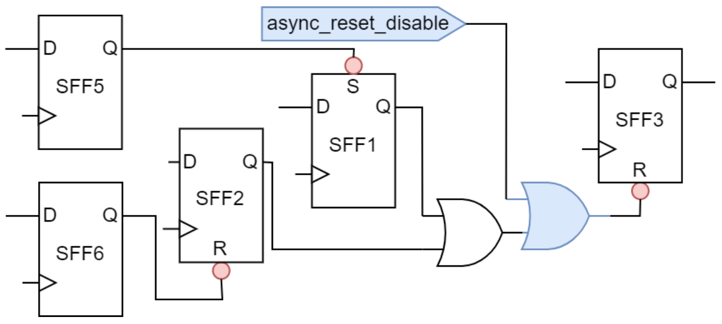


Figure 6.2: Test: 2-input OR gate

2-input OR gate

Both algorithms classify each set source as inverting. When the input is set (0) and reset (1) the simulation result is 0 which deviates from the rest. In this case, the reset is triggered which gives a 0 as response instead of 1. However, this has no impact on the classification of the sources.

	Inputs:	**Set***	Reset*	**Set**	Reset	Reset	Reset	**Set**	**Set**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	1	1	1
1	0	0	0	1	1	0	1	1	1
1	1	1	1	1	1	0	1	1	1

Table 6.3: Simulator results 2-input OR gate

2-input NOR gate

The propagation check algorithm classifies each reset source as inverting. So does the inversion simulator, except when the inputs are both a set and reset. When the input is set (1) and reset (0) the simulation result is 0 instead of a 1. As a result, the reset is not identified as an inverting source.

	Inputs	Set*	**Reset***	Set	Reset	**Reset**	**Reset**	Set	Set
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	0	1	1	1	0	1
0	1	0	0	0	1	0	1	0	0
1	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	1	0	0

Table 6.4: Simulator results 2-input NOR gate

2-input AND gate

Both algorithms classify each set source as inverting. When the input is set (1) and reset (0) the simulation result is 0 instead of a 1. However, this has no impact on the classification of the sources.

	Inputs	**Set***	Reset*	**Set**	Reset	Reset	Reset	**Set**	**Set**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	1	1	1	0	1	0	1	1	1

Table 6.5: Simulator results 2-input AND gate

2-input NAND gate

Both algorithms classify each reset source as inverting. When the input is set (0) and reset (1) the simulation result is 0 instead of a 1. However, this has no impact on the classification of the sources.

	Inputs	Set*	**Reset***	Set	**Reset**	**Reset**	**Reset**	Set	Set
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	0	1
0	1	1	1	0	1	1	1	0	1
1	0	1	1	1	1	1	1	0	1
1	1	0	0	0	1	0	1	0	0

Table 6.6: Simulator results 2-input NAND gate

2-input XOR gate

The inversion propagation check classifies each source as potentially inverting. For every source combination 1 simulation results deviates: Set (0) reset (1) should be 1, reset (1) reset (1) should be 0 and set (0) set (0) should be 0. The inversion simulator correctly identifies each source as inverting, except when the input is 2 reset sources. In that case, both sources are incorrectly identified as non-inverting.

	Inputs	**Set***	**Reset***	**Set**	Reset	Reset	**Reset**	**Set**	**Set**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	1	0	0	0	1
0	1	1	1	0	1	0	1	0	1
1	0	1	1	1	1	0	1	0	1
1	1	0	0	0	1	0	1	0	0

Table 6.7: Simulator results 2-input XOR gate

2-input XNOR gate

The inversion propagation check classifies each source as potentially inverting. For every source combination 1 simulation results deviates: Set (0) reset (1) should be 0, reset (1) reset (1) should be 1 and set (0) set (0) should be 1. The inversion simulator correctly identifies each source as inverting, except when the input is a set and reset source. In that case, both sources are incorrectly identified as non-inverting.

	Inputs	**Set***	**Reset***	Set	Reset	**Reset**	**Reset**	**Set**	**Set**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	0	1	1	1	0	1
0	1	0	0	0	1	0	1	0	1
1	0	0	0	0	0	0	1	0	1
1	1	1	1	0	1	0	1	1	1

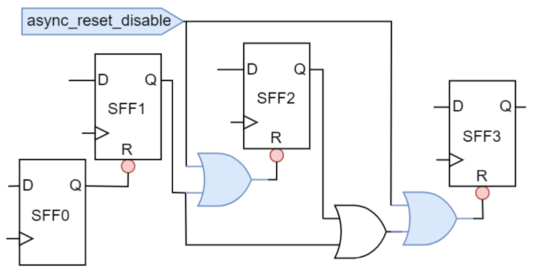
Table 6.8: Simulator results 2-input XNOR gate

6.1.3. Controlling sources

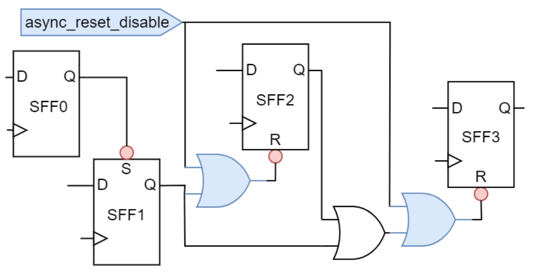
This section examines structures involving the reconvergence of a signal through a 2-input logic gate. The signal originates from a source that controls another source's (re)set port. For each logic gate two different setups are tested. One where the reset port of a flip-flop is controlled and a second where the set port of a flip-flop is controlled. Both setups will be discussed separately. These experiments will show how the simulator deals with cascading (re)set flip-flops as it can lead to a different response.

Figure 6.3a illustrates the reset source s_{R2} which is controlled by another reset source s_{R1}. Their output signal converges through the OR gate and feeds into the destination of interest: d_{R3}. Figure 6.3b illustrates a similar structure but in this case s_{R2} is controlled by s_{S1}.

Figures 6.4a and 6.4b illustrate a similar setup, but in this case the controlled source is a set source, s_{S6}, and the destination of interest is d_{R7}.



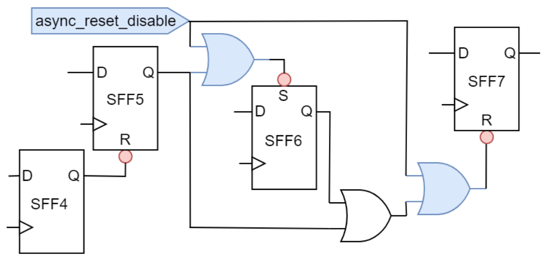
(a) By reset flip-flop



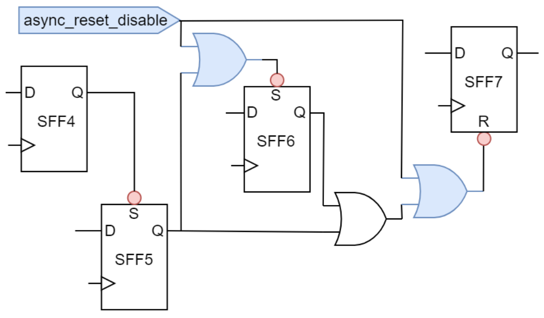
(b) By set flip-flop

Figure 6.3: Tests: Reset controlled

To prevent the simulator from altering the input data only the reset of the controlling source (SFF1/SFF5) needs to be tied to logic 1. The controlled source (SFF2/SFF6) is not changed as its reset behaviour should be part of the structure.



(a) By reset flip-flop



(b) By set flip-flop

Figure 6.4: Tests: Set controlled

Controlled reset source with OR gate

The inversion simulator correctly classifies the controlling set* and set as inverting sources.

	Inputs	Reset*	Reset	Reset	Reset	**Set***	Reset	**Set**	Reset
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	1
1	0	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1

Table 6.9: Simulator results for controlled reset source with OR gate

Controlled set source with OR gate

The inversion simulator classifies the controlled set source as inverting, but only when the controlling source is has a reset. At first glance it looks like there is a simulation error for this setup when the inputs are reset (1) set (0) = 0. But as these are asynchronous signals simulating a 0 is possible. When the reset is triggered the input values become reset (0) and set (0) for a short duration before the set source updates to set (1) again. This intermediate value will be captured if the propagation delay from SFF5 to SFF7 is shorter than the propagation delay from SFF6 to SFF7, which is likely as resetting a flip-flop takes longer compared to updating just 1 gate.

Note that the same behaviour would be observed if any of the other controlling sources switches from 0 to 1. This does not show up in the simulation results as that transition cannot take place in this context. In a real circuit, this structure should be avoided as it will also be very glitchy in functional mode.

	Inputs	Reset*	Set	Reset	**Set**	Set*	Set	Set	Set
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1
1	0	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Table 6.10: Simulator results for controlled set source with OR gate

Controlled reset source with NOR gate

The inversion simulator classifies no source as inverting when it has the set* and reset source. However, this is a mistake as a glitch can occur when the input is SFF1 (0) SFF2 (1) as this would result in a 0 at d_{R3} before the $async\_reset\_disable$ is released. After it is released the reset of SFF2 is triggered which results in a 1 at d_{R3}, thus a potential glitch. For the structure with Reset* and reset, also misidentifies the reset source as non-inverting. However, d_{R3} would still be considered glitch-sensitive as the Reset* source is classified as inverting.

	Inputs	**Reset***	Reset	**Reset**	**Reset**	Set*	Reset	Set	**Reset**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	0
1	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0

Table 6.11: Simulator results for controlled reset source with NOR gate

Controlled set source with NOR gate

The inversion simulator classifies the reset controlling source as inverting. The input reset (1) set (0) is similar to the set controlled OR gate. For this case the simulator finds the intermediate value when the reset source transitions from 1 to 0. This results in a glitch at SFF7 which will transition from 0 to 1 to 0. Note that this is a static 0-hazards which does not impact the test data, so the reset source is incorrectly identified.

	Inputs	Normal	Set	**Reset**	Set	Set*	Set	Set	Set
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0

Table 6.12: Simulator results for controlled set source with NOR gate

Controlled reset source with AND gate

The inversion simulator correctly identifies the set source as inverting.

	Inputs	Reset*	Reset	Reset	Reset	**Set***	Reset	**Set**	Reset
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	1	1	1	1

Table 6.13: Simulator results for controlled reset source with AND gate

Controlled set source with AND gate

The inversion simulator correctly identifies every set source as inverting. The simulator for both a set as controlling and controlled source gives some deviating outputs. However, these have no impact on the result as the sources are both correctly identified.

	Inputs	Reset*	**Set**	Reset	**Set**	**Set***	**Set**	**Set**	**Set**
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	1	1	1	1

Table 6.14: Simulator results for controlled set source with AND gate

Controlled reset source with NAND gate

The inversion simulator correctly identifies every reset source as inverting.

	Inputs	**Reset***	**Reset**	**Reset**	**Reset**	Set*	**Reset**	Set	**Reset**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0

Table 6.15: Simulator results for controlled reset source with NAND gate

Controlled set source with NAND gate

The inversion simulator correctly identifies the reset source as inverting. However, when both sources are a set source the algorithm incorrectly identifies them as inverting as well. This is because the simulator can trigger the set port of SFF5 which overwrites the test data. The different simulation values result in an incorrect classification.

	Inputs	**Reset***	Set	**Reset**	Set	Set*	Set	**Set**	**Set**
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0

Table 6.16: Simulator results for controlled set source with NAND gate

Controlled reset source with XOR gate

The inversion simulator correctly identifies every source as inverting, except for one. When both sources are a reset, the controlled source is missed. As the other source is identified it would still result in SFF7 being classified as glitch-sensitive. So for the end result, this has no effect.

	Inputs	**Reset***	**Reset**	**Reset**	Reset	**Set***	**Reset**	**Set**	**Reset**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	1	1
1	0	1	1	0	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0

Table 6.17: Simulator results for controlled reset source with XOR gate

Controlled set source with XOR gate

For any set controlled source this structure is already glitchy in functional mode. Take the following transition: SFF5 (1) SFF6 (0) = 1 to SFF5 (0) SFF6 (0) = 0. As this input triggers the set port it again transitions to SFF5 (0) SFF6 (1) = 1.

When the structure is a Set* and set source, the inversion algorithm incorrectly identifies neither source. This would result in d_{R7} being classified as glitch-free, which is a mistake. For the structures with a reset and set port only the reset is identified. But as long as one source is identified d_{R7} will be classified as glitch-sensitive.

	Inputs	**Reset***	Set	**Reset**	Set	Set*	Set	**Set**	**Set**
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	0	1
1	0	1	1	0	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0

Table 6.18: Simulator results for controlled set source with XOR gate

Controlled reset source with XNOR gate

For any set controlled source this structure is already glitchy in functional mode. Take the following transition: SFF5 (1) SFF6 (1) = 1 to SFF5 (0) SFF6 (1) = 0. As this input triggers the reset port it again transitions to SFF5 (0) SFF6 (0) = 1

When the structure is a Set* and set source, the inversion algorithm incorrectly identifies neither source. This would result in d_{R7} being classified as glitch-free, which is a mistake. For the structures with a reset and set port only the reset is identified. But as long as one source is identified d_{R7} will be classified as glitch-sensitive.

	Inputs	**Reset***	Reset	**Reset**	Reset	Set*	Reset	**Set**	**Reset**
SFF1	SFF2	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0	0	0
1	1	1	1	0	1	1	1	1	1

Table 6.19: Simulator results for controlled reset source with XNOR gate

Controlled set source with XNOR gate

The inversion simulator correctly identifies every set source as inverting.

	Inputs	Reset*	**Set**	Reset	**Set**	**Set***	**Set**	**Set**	**Set**
SFF5	SFF6	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1	Sim0	Sim1
0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	0	0	0
1	1	1	1	0	1	1	1	1	1

Table 6.20: Simulator results for controlled set source with XNOR gate

6.2. Results basic structures

The following section provides an overview of the results. Each table displays the true value, indicating whether the source should be classified as inverting (1) or non-inverting (0). The inversion propagation check can also output an X, indicating potential inverting. A cell is color-coded to indicate when it is misidentified by the algorithm. Yellow indicates that one source is missed, but the connected destination would still be classified correctly. Red indicates that all sources are missed, leading to an incorrect identification of the source.

The experiments where the (re)set ports of sources were tied to logic 1 are shown separately. These are to show how the inversion simulator algorithm would work if the Tessent simulator worked as intended.

Directly connected sources

Table 6.21 shows the results for the directly connected sources. These are all correctly identified by both algorithms.

Experiment	True value	Propagation check	Simulator
Normal: Reset / Set	1 / 0	1 / 0	1 / 0
Inverter: Reset/ Set	1 / 0	1 / 0	1 / 0

Table 6.21: Overview experiments: Directly connected sources

2-input logic gates

In Table 6.22, the results for the 2-input logic gates are presented. The inversion propagation check correctly identifies each source, except for structures with an XOR/XNOR, where the algorithm classifies the sources as potentially inverting.

The inversion simulator accurately identifies most sources, except for the NOR, XOR, and XNOR, where some structures were misidentified. This is due to the simulator having too much control, which triggers the set or reset of the sources, altering the input data.

Experiment	True value	Propagation check	Simulator
OR: Set / Reset	1 / 0	1 / 0	1 / 0
OR: Reset / Reset	0 / 0	0 / 0	0 / 0
OR: Set / Set	1 / 1	1 / 1	1 / 1
NOR: Set / Reset	0 / 1	0 / 1	0 / 0
NOR: Reset / Reset	1 / 1	1 / 1	1 / 1
NOR: Set / Set	0 / 0	0 / 0	0 / 0
AND: Set / Reset	1 / 0	1 / 0	1 / 0
AND: Reset / Reset	0 / 0	0 / 0	0 / 0
AND: Set / Set	1 / 1	1 / 1	1 / 1
NAND: Set / Reset	0 / 1	0 / 1	0 / 0
NAND: Reset / Reset	1 / 1	1 / 1	1 / 1
NAND: Set / Set	0 / 0	0 / 0	0 / 0
XOR: Set / Reset	1 / 1	X / X	1 / 0
XOR: Reset / Reset	1 / 1	X / X	0 / 1
XOR: Set / Set	1 / 1	X / X	1 / 1
XNOR: Set / Reset	1 / 1	X / X	0 / 0
XNOR: Reset / Reset	1 / 1	X / X	1 / 1
XNOR: Set / Set	1 / 1	X / X	1 / 1

Table 6.22: Overview experiments: 2-input gates

Table 6.23 shows the results when the (re)set sources are tied to 1. In this case, all of the simulation values were as expected and the inversion simulator correctly identifies every source.

Tied flip-flops	True value	Propagation check	Simulator
OR: "Set / Reset*	1 / 0	1 / 0	1 / 0
NOR: "Set / Reset*	0 / 1	0 / 1	0 / 1
AND: "Set / Reset*	1 / 0	1 / 0	1 / 0
NAND: "Set / Reset*	0 / 1	0 / 1	0 / 0
XOR: "Set / Reset*	1 / 1	X / X	1 / 1
XNOR: "Set / Reset*	1 / 1	X / X	1 / 1

Table 6.23: Overview experiments: 2-input gates with tied (re)set ports

Controlling sources

Table 6.24 shows the results for the controlling gates when normal (re)set flip-flops are used. Apart from the XOR/XNORS, the inversion propagation check classified only 1 false positive. That is for the NOR with reset/set source as these sources would not produce a glitch. As a result a destination would be incorrectly classified as glitch-sensitive. Apart from that, only 1 source is misidentified, which is the OR with set/set source. This would not have an impact on the overall result as the destination is correctly classified.

The inversion simulator produced 1 false negative and 2 false positives for the (re)set flip-flops. So it performed worse compared to the inversion propagation check. For the XOR/XNOR structures 3 sources were misidentified, but the destinations are still classified correctly.

Experiment	True value	Propagation check	Simulator
OR: Reset / Reset	0 / 0	0 / 0	0 / 0
OR: Set / Reset	1 / 0	1 / 0	1 / 0
OR: Reset / Set	0 / 1	0 / 1	0 / 1
OR: Set / Set	0 / 1	1 / 1	0 / 0
NOR: Reset / Reset	1 / 1	1 / 1	1 / 1
NOR: Set / Reset	0 / 1	0 / 1	0 / 1
NOR: Reset / Set	0 / 0	1 / 0	1 / 0
NOR: Set / Set	0 / 0	0 / 0	0 / 0
AND: Reset / Reset	0 / 0	0 / 0	0 / 0
AND: Set / Reset	1 / 0	1 / 0	1 / 0
AND: Reset / Set	0 / 1	0 / 1	0 / 1
AND: Set / Set	1 / 1	1 / 1	1 / 1
NAND: Reset / Reset	1 / 1	1 / 1	1 / 1
NAND: Set / Reset	0 / 1	0 / 1	0 / 1
NAND: Reset / Set	1 / 0	1 / 0	1 / 0
NAND: Set / Set	0 / 0	0 / 0	1 / 1
XOR: Reset / Reset	1 / 1	X / X	1 / 0
XOR: Set / Reset	1 / 1	X / X	1 / 1
XOR: Reset / Set	1 / 1	X / X	1 / 0
XOR: Set / Set	1 / 1	X / X	1 / 1
XNOR: Reset / Reset	1 / 1	X / X	1 / 0
XNOR: Set / Reset	1 / 1	X / X	1 / 1
XNOR: Reset / Set	0 / 1	X / X	0 / 1
XNOR: Set / Set	1 / 1	X / X	1 / 1

Table 6.24: Overview experiments: Controlling sources

Table 6.25 shows the results when the (re)set port of the controlling source is tied to 1. The inversion simulator performs slightly better for the structures without an XOR/XNOR as it only produced 1 false negative. This however is still worse than the inversion propagation check. For the XOR/XNOR structures there are two false negatives, which is significantly worse compared to the sources that were not tied.

When the (re)set ports are tied the simulator can only generate the response for a static input. However, when the simulator has full control over the (re)set ports, it can also generate intermediate values by resetting the controlling source. For a brief moment, the input at the logic gate will already be updated, but the controlled source is not. These are functional glitches which are being simulated. But when the (re)set ports are tied these functional glitches do not show up, hence exactly the structures with functional glitches are incorrectly identified.

Normal flip-flops	True value	Propagation check	Simulator
OR: Reset* / Reset	0 / 0	0 / 0	0 / 0
OR: Set* / Reset	1 / 0	1 / 0	1 / 0
OR: Reset* /Set	0 / 1	0 / 1	0 / 1
OR: Set* /Set	0 / 1	1 / 1	0 / 0
NOR: Reset* / Reset	1 / 1	1 / 1	1 / 0
NOR: Set* / Reset	0 / 1	0 / 1	0 / 0
NOR: Reset* /Set	0 / 0	1 / 0	0 / 0
NOR: Set* /Set	0 / 0	0 / 0	0 / 0
AND: Reset* / Reset	0 / 0	0 / 0	0 / 0
AND: Set* / Reset	1 / 0	1 / 0	1 / 0
AND: Reset* /Set	1 / 1	1 / 1	1 / 1
AND: Set* / Set	1 / 1	1 / 1	1 / 1
NAND: Reset* / Reset	1 / 1	1 / 1	1 / 1
NAND: Set* / Reset	0 / 1	0 / 1	0 / 1
NAND: Reset* /Set	1 / 0	1 / 0	1 / 0
NAND: Set* / Set	0 / 0	0 / 0	0 / 0
XOR: Reset* / Reset	1 / 1	X / X	1 / 1
XOR: Set* / Reset	1 / 1	X / X	1 / 1
XOR: Reset* /Set	1 / 1	X / X	1 / 0
XOR: Set* / Set	1 / 1	X / X	0 / 0
XNOR: Reset* / Reset	1 / 1	X / X	1 / 0
XNOR: Set* / Reset	1 / 1	X / X	0 / 0
XNOR: Reset* /Set	0 / 1	X / X	0 / 1
XNOR: Set* / Set	1 / 1	X / X	1 / 1

Table 6.25: Overview experiments: Controlling sources with tied (re)set ports

6.3. Functional structures with reconvergence

This section will cover some structures with reconvergence through combinational logic from the same source. Unlike the previous experiments, these structures have a specific functionality which can be easily verified.

Multiplexer build from primitive logic gates

Figure 6.5 illustrates a multiplexer (MUX) build from NOR and OR gates. There is reconvergence for SFF3 as it can take two different paths to SFF4. The inversion propagation check picks one of the two paths at random and calculates the inversion. In this example, it chose the bottom path, from which it classifies s_{R3} as non-inverting.

The output from the inversion simulator is shown in Table 6.26 where sim1 corresponds with the truth table of a MUX. For sim0 when the input is: SFF3(1) SFF2(0) SFF1(1) the simulator resets SFF3 which gives a 0 as a response. This does not affect affect the classification.

In this example the inversion propagation check incorrectly identified s_{R3} and as this is the only source of d_{R4} it would be classified glitch-free. The inversion simulator correctly identifies the source and corrects this mistake.

Figure 6.6 also illustrates a MUX structure but built using NAND gates. In this experiment, SFF1 and SFF2 are replaced by set sources, which allows the simulator to set the input values to one.

The output of the inversion simulator is shown in Table 6.27 which does not correspond with a normal MUX. Based on these values the inversion simulator identifies s_{S1},s_{S2} as inverting and s_{R3} as non-inverting. In the previous example it was shown that s_{R3} is inverting as well, so this is missed due to the incorrect simulation values. The destination would still be classified correctly as at least one source is identified as inverting.

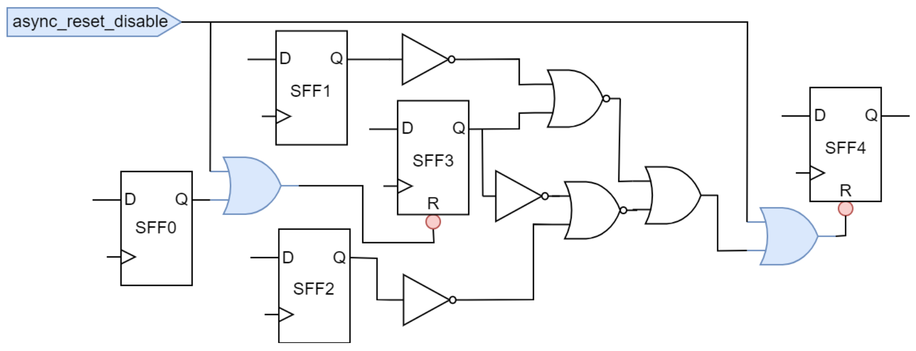


Figure 6.5: Experiment: MUX with one reset source

SFF3	SFF2	SFF1	sim0	sim1
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Table 6.26: Simulation results MUX with one reset source

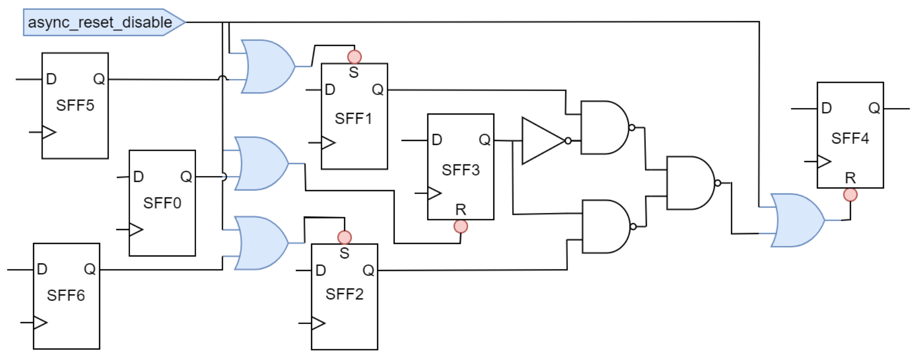


Figure 6.6: Experiment: MUX with multiple sources

The same experiment was repeated but with the (re)set ports tied to 1. The results of the simulator are shown in Table which does correspond with the truth table of a MUX. The inversion algorithm correctly identifies s_{S1},s_{S2} and s_{R3} as inverting. These experiments show that the (re)set ports must indeed be tied to 1 to prevent the Tessent simulator from altering the input data.

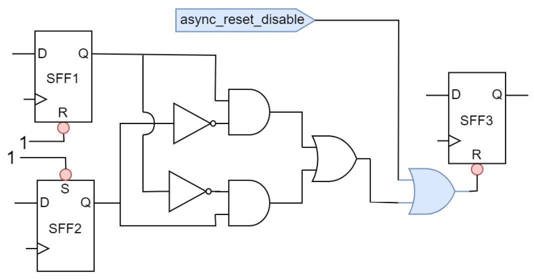
SFF3	SFF2	SFF1	sim0	sim1
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

Table 6.27: Simulation results MUX with multiple sources

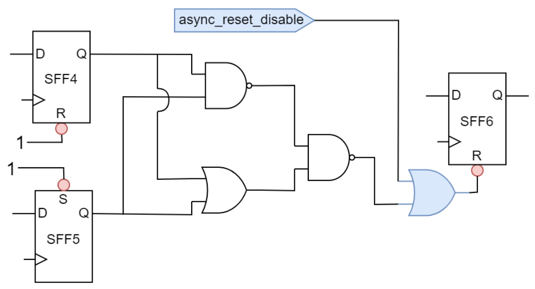
SFF3	SFF2	SFF1	sim0	sim1
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

Table 6.28: Simulation results MUX with tied (re)set sources

XOR and XNOR build from primitive logic gates
Figure 6.7a and 6.7b show a structure that functions as an XOR and XNOR respectively. These results are easily verified as they should be the same as for the 2-input XOR and XNOR gate. The (re)set sources are all tied to 1 to ensure correct simulation values.



(a) Experiment: Reconvergence XOR structure



(b) Experiment: Reconvergence XNOR structure

The inversion propagation check identified the sources s_{R1} and s_{S5} as inverting for the XOR structure. From previous experiments we know that each source is inverting, so s_{S2} and s_{R4} were missed in this case.

The Tables below show the results from the simulator. Based on these the inversion simulator correctly identifies every source as inverting.

SFF1	SFF2	sim0	sim1
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0

Table 6.29: Simulation results XOR structure

SFF5	SFF6	sim0	sim1
0	0	1	1
0	1	0	1
1	0	0	1
1	1	1	1

Table 6.30: Simulation results XNOR structure

6.4. Case study 1

This section discusses a circuit that experienced glitches during the design phase. The circuit contains many repetitive structures comprised of 189 flip-flops, including 100 flip-flops with a reset and 14 flip-flops with a set. Multiple (re)set flip-flops share a local reset, which are 17 in total.

A large part of the structure consists of cascading reset sources as illustrated in Figure 6.8. As there is no inversion between the sources, the algorithm classifies them as glitch-free.

Figure 6.9a displays one of the other structures present in the circuit. This consists of 2 AND gates and three sources. Due to the set source s_{S6} a glitch could occur at destination d_{S7}. Figure 6.9b is almost identical but with an additional OR gate inserted after SFF8. This OR gate is connected with the $async\_reset\_disable$ signal, which was incorrectly inserted into the combinational logic. The inversion propagation correctly identifies the sources s_{S6} and s_{S9} as inverting.

Out of the 114 destinations, the inversion propagation check classified 33 as glitch-sensitive and 81 as glitch-free. As a result only 6 out of the 17 local resets are classified as glitch-sensitive, which means

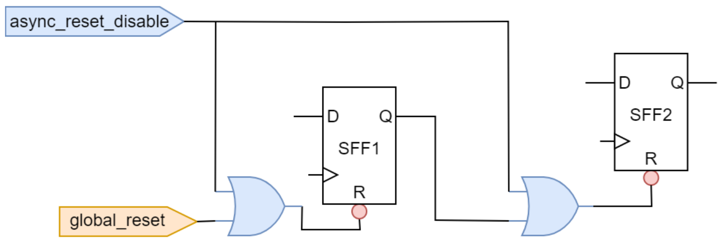
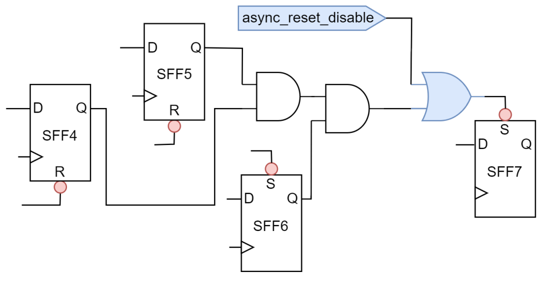
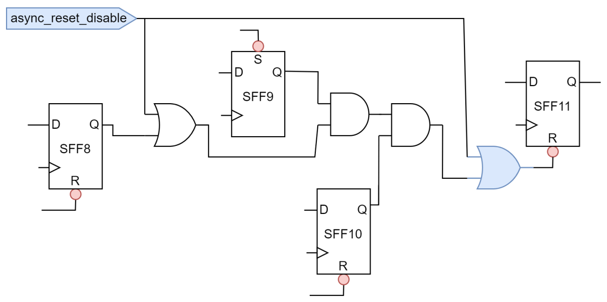


Figure 6.8: Real glitchy circuit: Non-inverting cascading resets



(a) Case study 1: Glitch-sensitive structure



(b) Case study 1: Incorrectly inserted *async_reset_disable* signal

that 65% less DfT hardware for asynchronous resets is required. Every destination was also evaluated by the inversion simulator. For Figure 6.9a the simulator results are displayed in Table 6.31. For these structures, the inversion simulator identifies the same sources as inverting, so it did not lead to any improvements.

The inversion simulator reports errors for ten destinations. These are all from the structure illustrated in Figure 6.9b. The error is caused by the incorrectly inserted *async_reset_disable* which causes the simulator to crash. Nevertheless, as the inversion propagation check accurately classified these destinations, this has no impact on the overall results.

SFF4	SFF5	SFF6	sim0	sim1
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 6.31: Simulation results for Figure 6.9a

6.5. Case study 2

This section discusses another case study which is the circuit shown in Figure 6.10. This schematic shows a glitch at d_{R9} when the *async_reset_disable* is released and the reset of s_{R1} is triggered. In functional mode, resetting SFF1 will cause every scan flip-flop in this schematic to be (re)set.

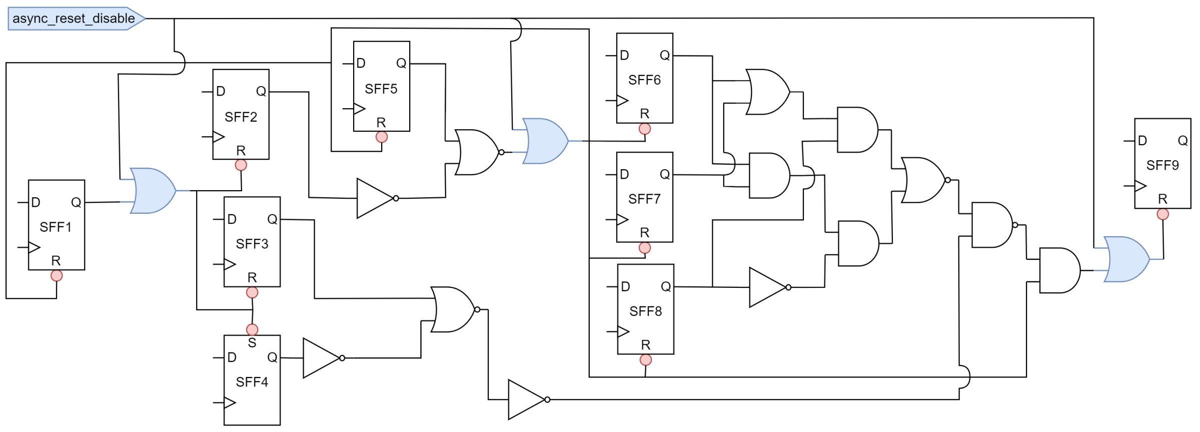


Figure 6.10: Part of a circuit where glitch was observed at SFF9

The inversion propagation check identified 4 inverting sources and classified 6 destinations as glitch-sensitive. Figure 6.11 illustrates which sources are inverting (Q in green) and which destinations are glitch-sensitive (R in red). The sources s_{R6},s_{R7} and s_{R8} all have paths with reconvergence, but only s_{R8} has both an inverting and non-inverting path. The inversion propagation check picked the inverting path and identified s_{R8} as inverting.

Running the inversion simulator for d_{R9} results in a table with 128 input values. Only the part with non-zero values of this simulation is shown in the Appendix: Table A.1. For most input values the simulation results in 0. This is due to the last AND gate feeding into d_{R9} as this will only produce a 1 when the input value is: SFF2 (1) and SFF5 (0). The same input to this AND gate also triggers the reset ports of s_{R6},s_{R7} and s_{R8}. So even if resetting s_{R8} would produce a glitch, it could not propagate through the AND gate as the other input is 0.

The inversion simulator correctly identifies this source as non-inverting, which is the only difference compared to the inversion propagation check. This does not change anything regarding the classification of the destinations. Out of the 9 (re)set sources both algorithms classify 6 as inverting and thus leads to a reduction of 33% for the DfT hardware overhead for asynchronous (re)sets.

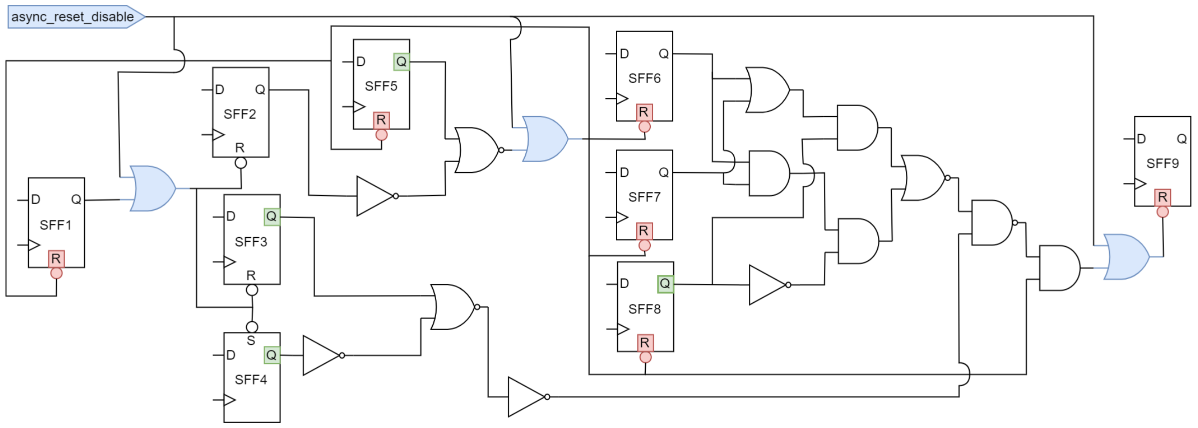


Figure 6.11: Sources and destinations identified by inversion propagation check

6.6. Evaluation of the execution time for the proposed algorithms

Benchmarks have been conducted on two case studies to compare the execution time of both algorithms. The reset synchronizer circuit contains 189 flip-flops, out of which 114 have a set/reset. This circuit is part of a much larger system that contains 1457 flip-flops, with 95 having a set/reset. The function of this larger integrated circuit (IC) is more complex, and due to confidentiality reasons, we cannot discuss its function.

The inversion propagation check is performed for every set/reset flip-flop in the circuit. Running the inversion simulator is optional and will only compute for destinations with at least one source (both inverting and non-inverting). Moreover, the inversion simulator can be set to run for specific destinations if needed.

For case study 1 the inversion simulator was executed in 0.250s and the inversion simulator was run on 98 which took 3.235s. These results are also listed in Table 6.32. In contrast, the second case study which has approximately 13 times more flip-flops, took the inversion simulator 3.850s, demonstrating that the execution time scales linearly with the number of flip-flops. The inversion simulator was only run for the schematic with an observed glitch, which has 9 destinations, and it took 14.217s.

	Inversion propagation check	Inversion simulator
Case study 1	0.250 s	2.935 s
Case study 2	3.253 s	10.217 s

Table 6.32

The reason why the inversion simulator relatively takes so much longer for case study 1 is because there is one destination (d_R9) which has 7 inputs. That means $2^7 = 128$ input values need to be simulated. While the reset synchronizer circuit has destinations with at most $2^3 = 8$ input values. This shows that the inversion simulator has indeed a time complexity of $O(2^N)$, where N is the maximum number of inputs to a destination.

To illustrate how the inversion simulator scales an additional experiment was run. In this case, simple schematics were made with increasing inputs to the destinations. Figure 6.12 shows how the executing time of the inversion simulator increases exponentially with the number of inputs.

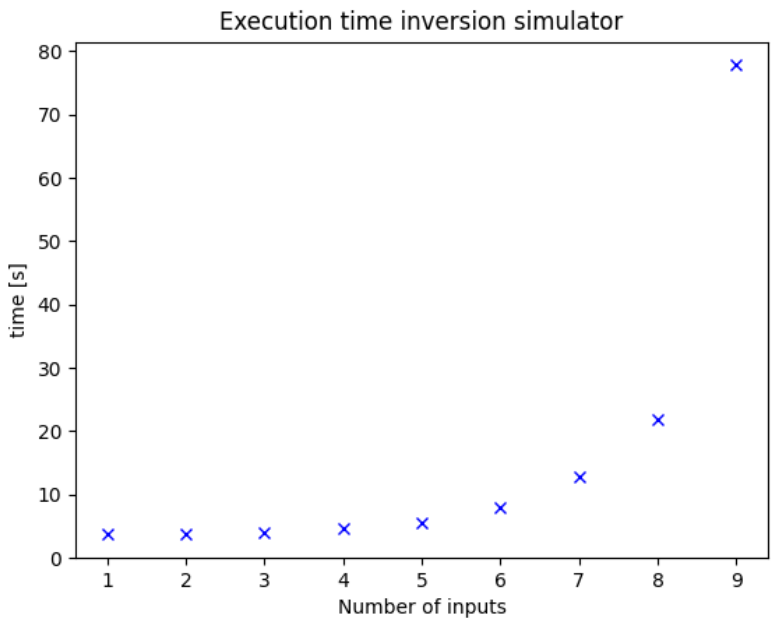


Figure 6.12: Execution time with inversion simulator

7 Discussion

7.1. Experiments

The initial structures were constructed to demonstrate the capabilities and limitations of each algorithm without considering functionality. The structure of some of the controlled sources should never be put into an actual design. Such as the controlled set source with a NOR gate which will always output a 0. As these structures should be avoided it is not as important that the algorithm incorrectly identifies them. Additional experiments should be conducted of which the functionality is taken into account.

In all of the experiments, there was only one structure for which the algorithm incorrectly classified a destination. This was the controlled set source with NOR gate which produced a false positive. Granted that this structure has no real functionality, it perhaps should be overlooked. In addition, a false positive would only result in slightly more DfT than necessary, while a false negative could result in a glitchy circuit. So a false positive does not have a huge impact.

The algorithm does fall short when there are paths with reconvergence. In that case, Tessent picks one path for which the inversion is calculated, which could result in sources being classified incorrectly. The algorithm cannot tell if it encountered reconvergence which is a big downside.

When there are XOR/XNOR gates or there is reconvergence present the inversion simulator correctly identifies most sources. However, there are instances where this algorithm produces false negatives. This is only the case when there are functional glitches present, so these should already be detected during functional verification.

These functional glitches could be detected as well if the simulator is allowed to (re)set the input flip-flops. In the current implementation this alters the input data which corrupts the simulator results. An improvement could be made where the input data is adjusted accordingly through feedback between the algorithm and simulator. This would be very difficult to realize in Tessent as the simulator does not clearly provide if a reset has been triggered.

7.1.1. Case studies

The algorithm was only tested on two case studies that suffered from glitches. The first case study only contained relatively simple structures with at most 3 sources per destination and no reconverging paths. It did contain a design mistake as the $async\_reset\_disable$ signal was incorrectly inserted into the combinational logic (Figure 6.9b. While the algorithm was not intended to find such mistakes, the errors reported by the inversion simulator for these specific structures indirectly led to their discovery.

The second case study has more complex structure which includes reconvergence. The inversion propagation check inaccurately identified a source as inverting, which was subsequently corrected by the inversion simulator. This demonstrates how the algorithms are meant to work together.

7.1.2. Tessent simulator

The results from the experiments show that the simulator from Tessent does not always produce accurate results. When the (re)set ports of the inputs are tied to logic 1 the simulator behaves as expected for the 2-input logic gate. However, for the controlling sources, the structures with functional glitches are missed. An argument could be made that these glitches should already be taken care of in the verification step of the design.

For the case studies none of the sources needed to be tied to 0. This is not necessary for the reset synchronizer circuit as it contains either directly connected sources or structures with AND gates. As is shown in the results for the basic structures, these types of structures work fine without tying the (re)set ports.

The schematic (Figure 6.10) on the other hand does contain structures for which the simulator produced errors, such as the NOR gate with set/reset port. However, the functionality of this schematic creates dependencies between flip-flops. The dependencies limit the control that Tessent has and as a result the simulator computes as intended.

7.2. Robustness

The algorithm detects logic hazards caused by de-asserting the $async\_disable$ signal. To ensure that the asynchronous scan-test is robust the worst-case scenario is considered where it is assumed that a hazard can result in a glitch. In reality, this depends on the propagation delay of the different paths traversed by the $async\_disable$ signal. The accuracy of the algorithm could be improved by modeling the propagation delay of those paths. This would make the algorithm a lot more complex as the propagation delay is dependent on the structure of the gates, manufacturing process variations and environmental variations.

Even if the algorithm classifies a (re)set flip-flop as glitch-free, it is still possible for a glitch to appear due to transient faults. These faults can be caused by physical phenomena, such as electromagnetic interference, capacitive coupling, or high-energy particles from cosmic rays. Preventing these kinds of glitches requires separate models and dedicated solutions.

7.3. Performance

The first case study comprises of 114 (re)set flip-flops where multiple flip-flops share a local reset. Both algorithms classified 6 out of 17 local resets to be glitch-sensitive, which means 65% less DfT hardware for asynchronous (re)sets is required.

The circuit contains 6232 gates, including the OR gates for the $async\_disable$, and 189 scan flip-flops. As a scan flip-flop can be built with 14 logic gates [15] the total number of gates is $6232 + 189 * 14 = 8878$. The DfT for each local reset adds another 3 gates and one scan flip-flop, so 17 gates. Inserting the DfT at every local (re)set gives $8878 + 17 * 17 = 9167$ gates. After running the algorithm only 6 local (re)sets require DfT so the total number of gates becomes $8878 + 6 * 17 = 8980$. So after the algorithm, the total number of gates can be reduced by $1 - \frac{8980}{9167} = 2.0\%$

The same calculations can be done for the second case study. This circuit contains 16 gates, 9 scan flip-flops and 3 local resets. The total number of gates including the DfT is $16 + 9 * 14 + 3 * 17 = 193$. After running the algorithm 2 of the local resets are classified as glitch-sensitive, so the total number of gates is $16 + 9 * 14 + 2 * 17 = 176$ which results in a reduction of $1 - \frac{176}{193} = 8.8\%$.

A reduction in the total number of gates provides an indication of potential area savings. Determining the exact amount of area requires the post-layout netlist of the circuits before and after the algorithm. This information was not obtainable with the tools provided and is thus recommended for future work.

To accurately determine the performance of the algorithm it must be tested on more circuit designs. Asynchronous (re)set glitches only show up during scan test when the circuit contains a lot of ripple reset structures. During this work it was not possible to obtain additional circuits with these specific structures.

7.4. Integration

The entire algorithm could easily be integrated into any workflow as it only needs a scan-inserted netlist. The current implementation outputs a text file that contains every destination with at least one inverting source, which are the flip-flops that require additional DfT. Another algorithm can be built to automatically insert the DfT at these flip-flops.

These algorithms can also be used to verify the Intellectual Property of outside vendors. As the algorithm only requires a scan-inserted netlist it can indicate if there are (re)set flip-flops that are not robust against asynchronous reset glitches during scan-test.

7.5. Scalability

The main problem with the inversion simulator algorithm is its time complexity of $O(2^N)$[53], where N is the maximum number of inputs to a destination. This means that t, the execution time per destination, can scale quite rapidly for complex circuits. This is not a big drawback for less complex circuits where N is small such as in case study 1: N = 7, t = 10s and in case study 2: N = 3, t = 3s. However, circuits with a large fan-in on the reset logic can have a larger N in which case a maximum N of about 8 (t = 21s) is recommended.

One major improvement would be to compute at each local reset instead of each (re)set port. Both algorithms calculate for every (re)set port (destination) whether it is inverting or not, however as each local reset has the same source this calculation would only need to be done once. For example, the reset synchronizer requires only 17 simulations instead of 98.

The algorithm only needs to be run once or twice during the design cycle of a chip. Synthesis and simulation of large designs can already take multiple hours or even days. So even if the simulations take a while, it might not add that much overhead.

Alternatively, only the inversion propagation check can be computed which adds minimal overhead. This would give the same results for circuits where there is no reconvergence and no XOR/XNOR gates present. In the experiments conducted every structure with an XOR/XNOR was inverting. That is because these gates will always flip their output when one of their inputs changes value, which makes a glitch likely. Perhaps instead of classifying paths with an XOR/XNOR as potentially inverting, they could just always be set to inverting. It is also possible to run to inversion simulator just for paths with an XOR/XNOR to check if further optimization is possible at the cost of a longer execution time.

8 Conclusion and future work

8.1. Conclusion

This thesis conducted a comprehensive analysis of glitches that occur during asynchronous scan tests. The study investigated the conditions under which these glitches occur and proposed an inversion model to identify which (re)set flip-flops are sensitive to these glitches.

Two separate algorithms are proposed that can classify which (re)set flip-flops are glitch-sensitive. The inversion propagation check, a design rule-based algorithm, identifies glitch-sensitive (re)set flip-flops based on inverting paths. The inversion simulator, which is a simulation-based algorithm, is an extension that finds the source of a glitch even when there is reconvergence and dependencies on paths.

The experiments show that the inversion propagation check is the most accurate in identifying glitchy structures. It only resulted in one false positive in all of the conducted experiments which is quite promising. The inversion simulator can find the true inverting sources while there is reconvergence in the circuit but at the cost of simulation overhead.

Both algorithms have been combined in a detection algorithm which has been tested on two circuits where glitches were observed. Applying the algorithm resulted in 33% and 65% of the flip-flops being classified as safe against glitches. As a result, the total number of gates can be reduced by 8.8% and 2.0% respectively.

8.2. Future work

The following items are suggested for future work:

- Additional tests on functional circuits should be conducted. From these the performance of both algorithms can be measured.
- The inversion propagation check does not work for recovering paths. To improve the algorithm another design rule check could be implemented to identify recovering paths. In that case a designer would know that for those paths the inversion propagation check might lead to incorrect values.
- The inversion simulator can take any scan flip-flop as an input for the local simulation. However, the values of primary inputs can not be constrained in the same way. This is a limitation from Tessent, but perhaps a workaround can be made or another simulator can be used.
- Instead of performing both algorithms on each (re)set port it could be done on each local reset instead. This gives the same results while reducing the computation time.
- Improve the inversion simulator to also detect functional glitches. This would require another simulation tool which provides feedback about which flip-flops have been reset.
- A feature that could be added to the detection algorithm is a verification for the async reset disable. This could notify designers when it is incorrectly inserted into the circuit.

Bibliography

[1] E. Sperling, “Customizing ic test to improve yield and reliability,” 2023. [Online]. Available: https://semiengineering.com/customizing-ic-test-improve-yield-and-reliability/

[2] S. Sindia and V. Agrawal, “Defect level constrained optimization of analog and radio frequency specification tests,” *Journal of Electronic Testing*, vol. 31, 12 2015.

[3] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, “Silent data corruptions at scale,” 2021. [Online]. Available: https://arxiv.org/abs/2102.11245v1

[4] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, “Detecting silent data corruptions in the wild,” 2022. [Online]. Available: https://arxiv.org/abs/2203.08989v1

[5] A. Singh, S. Chakravarty, G. Papadimitriou, and D. Gizopoulos, “Silent data errors: Sources, detection, and modeling,” *Proceedings of the IEEE VLSI Test Symposium*, vol. 2023-April, 2023.

[6] A. Ahmad, “Automotive semiconductor industry - trends, safety and security challenges,” *ICRITO 2020 - IEEE 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)*, pp. 1373–1377, 2020.

[7] A. Meixner, “The race to zero defects in auto ics,” 2020. [Online]. Available: https://semiengineering.com/the-race-to-zero-defects-in-auto-ics/

[8] “International technology roadmap for semiconductors - test and test equipment,” 2015. [Online]. Available: https://www.semiconductors.org/wp-content/uploads/2018/06/0_2015-ITRS-2.0-Executive-Report-1.pdf

[9] E. Sperling, “Test costs spiking,” 2020. [Online]. Available: https://semiengineering.com/test-costs-spiking/

[10] J. Savir and S. Patil, “Scan-based transition test,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1232–1241, 1993.

[11] “Ieee standard for test access port and boundary-scan architecture,” *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, 2013.

[12] “Robust asynchronous-reset architecture for scan coverage,” *EDN - Voice of the Engineer*, 2015. [Online]. Available: https://www.edn.com/robust-asynchronous-reset-architecture-for-scan-coverage/

[13] T. Waayers, J. C. Meirlevede, P.-H. Pugliesi-Conti, and V. Chalendard, “Testing of asynchronous reset logic,” 2019. [Online]. Available: https://patents.google.com/patent/US11301607B2/en?oq=11%2c301%2c607

[14] Y. Peng, I. W. Jones, and M. R. Greenstreet, “Finding glitches using formal methods,” *Proceedings - International Symposium on Asynchronous Circuits and Systems*, vol. 2016-October, pp. 45–46, 2016.

[15] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[16] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann Publishers Inc., 2006.

[17] R. R. Montañés, P. Volf, and J. P. D. Gyvez, “Resistance characterization for weak open defects,” *IEEE Design and Test of Computers*, vol. 19, pp. 18–25, 2002.

[18] T. W. Williams and N. C. Brown, "Defect level as a function of fault coverage," *IEEE Transactions on Computers*, vol. C-30, pp. 987–988, 1981.

[19] R. Dekker, F. Beenker, and L. Thijssen, "A realistic fault model and test algorithms for static random access memories," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 567–572, 1990.

[20] E. J. McCluskey and C. W. Tseng, "Stuck-fault tests vs. actual defects," *IEEE International Test Conference (TC)*, pp. 336–343, 2000.

[21] J. H. Patel, "Stuck-at fault: A fault model for the next millennium," *IEEE International Test Conference (TC)*, p. 1166, 1998.

[22] S. D. Millman, E. J. McCluskey, and J. M. Acken, "Diagnosing cmos bridging faults with stuck-at fault dictionaries," *Digest of Papers - International Test Conference*, pp. 860–870, 1990.

[23] J. A. Acken and S. D. Millman, "Accurate modeling and simulation of bridging faults," *Proceedings of the Custom Integrated Circuits Conference*, 1991.

[24] B. Kruseman, A. Majhi, C. Hora, S. Eichenberger, and J. Meirlevede, "Systematic defects in deep sub-micron technologies," *Proceedings - International Test Conference*, pp. 290–299, 2004.

[25] J. M. Emmert, C. E. Stroud, and J. R. Bailey, "New bridging fault model for more accurate fault behavior," *AUTOTESTCON (Proceedings)*, pp. 481–485, 2000.

[26] G. L. Smith, "Model for delay faults based upon paths." vol. 85, 1985, pp. 342–349.

[27] C. J. Lin and S. M. Reddy, "On delay fault testing in logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, pp. 694–703, 1987.

[28] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition fault simulation," *IEEE Design and Test of Computers*, vol. 4, pp. 32–38, 1987.

[29] Y. Shao, I. Pomeranz, and S. M. Reddy, "On generating high quality tests for transition faults," *Proceedings of the Asian Test Symposium*, vol. 2002-January, pp. 1–8, 2002.

[30] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, 2010.

[31] T. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 567–580, 1967.

[32] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, vol. C-30, pp. 215–222, 1981.

[33] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Transactions on Computers*, vol. C-32, pp. 1137–1144, 1983.

[34] A. Steininger, "Testing and built-in self-test – a survey," *Journal of Systems Architecture*, vol. 46, pp. 721–747, 2000.

[35] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design and Test of Computers*, vol. 2, pp. 21–28, 1985.

[36] J. Aerts and E. J. Marinissen, "Scan chain design for test time reduction in core-based ics," *IEEE International Test Conference (TC)*, pp. 448–457, 1998.

[37] C. E. Cummings, D. Mills, and S. Golson, "Asynchronous & synchronous reset design techniques-part deux," *SNUG Boston*, vol. 9, 2003.

[38] P. Srivastava, *Introduction to Asynchronous Circuit Design*. Springer International Publishing, 2022, pp. 1–13.

[39] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, pp. 30–39, 2005.

[40] N. Miskov-Zivanov and D. Marculescu, "Multiple transient faults in combinational and sequential circuits: A systematic approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1614–1627, 2010.

[41] M. Kasim, V. Gupta, and M. Jebin, "Methodology for detecting glitch on clock, reset and cdc path." IEEE, 2020, pp. 300–304. [Online]. Available: https://ieeexplore.ieee.org/document/9138064/

[42] C. P. Garcia and T. H. Both, "An analytical gate delay variability model for low-power applications under the process variations effects," *Journal of Integrated Circuits and Systems*, vol. 18, pp. 1–9, 2023. [Online]. Available: https://jics.org.br/ojs/index.php/JICS/article/view/654

[43] H. Mahmoodi, S. Mukhopadhyay, and K. Roy, "Estimation of delay variations due to random-dopant fluctuations in nanoscale cmos circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 1787–1795, 2005.

[44] P. Verma, A. K. Sharma, V. S. Pandey, A. Noor, and A. Tanwar, "Estimation of leakage power and delay in cmos circuits using parametric variation," *Perspectives in Science*, vol. 8, pp. 760–763, 2016.

[45] K. G. Verma, B. K. Kaushik, and R. Singh, "Propagation delay variation due to process induced threshold voltage variation," *Communications in Computer and Information Science*, vol. 101, pp. 520–524, 2010. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-15766-0_87

[46] T. McLaurin, F. Frederick, and R. Slobodnik, "The dft challenges and solutions for the arm® cortex™-a15 microprocessor," *Proceedings - International Test Conference*, 2012.

[47] K. H. Tsai and S. Sheng, "Design rule check on the clock gating logic for testability and beyond," *Proceedings - International Test Conference*, 2013.

[48] A. Yadav, P. Jindal, and D. Basappa, "Study and analysis of rtl verification tool," *2020 IEEE Students' Conference on Engineering and Systems, SCES 2020*, 2020.

[49] "Synopsys spyglass rdc - datasheet." [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-rdc.html

[50] "Real intent - meridian rdc." [Online]. Available: https://www.realintent.com/reset-domain-crossing-meridian-rdc/

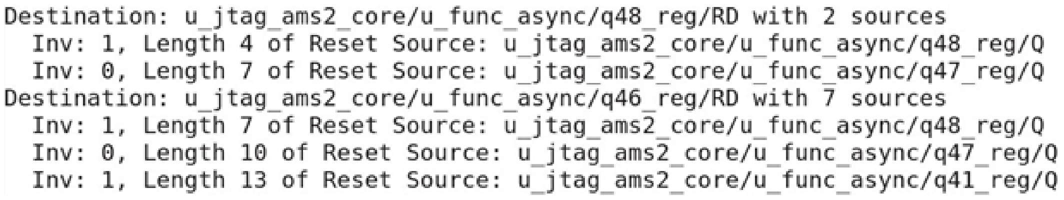
[51] "Siemens questa rdc verification -datasheet." [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/design-solutions/reset-domain-crossing/?pk_vid=263901564ccee18d359da1216c2e1ee217143177697f471d

[52] "Real intent gets 3rd best of 2018," 2019. [Online]. Available: https://www.deepchip.com/items/dac18-03.html

[53] G. J. Woeginger, "Space and time complexity of exact algorithms: Some open problems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3162, pp. 281–290, 2004. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-28639-4_25

A Appendix

A.1. Implementation



```
Destination: u_jtag_ams2_core/u_func_async/q48_reg/RD with 2 sources
  Inv: 1, Length 4 of Reset Source: u_jtag_ams2_core/u_func_async/q48_reg/Q
  Inv: 0, Length 7 of Reset Source: u_jtag_ams2_core/u_func_async/q47_reg/Q
Destination: u_jtag_ams2_core/u_func_async/q46_reg/RD with 7 sources
  Inv: 1, Length 7 of Reset Source: u_jtag_ams2_core/u_func_async/q48_reg/Q
  Inv: 0, Length 10 of Reset Source: u_jtag_ams2_core/u_func_async/q47_reg/Q
  Inv: 1, Length 13 of Reset Source: u_jtag_ams2_core/u_func_async/q41_reg/Q
```

Figure A.1: A part of the text file output

A.2. Experiments

SFF2	SFF5	SFF8	SFF6	SFF7	SFF4	SFF3	sim0	sim1
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	1
1	0	0	0	0	1	1	0	0
1	0	0	0	1	0	0	0	0
1	0	0	0	1	0	1	0	0
1	0	0	0	1	1	0	0	1
1	0	0	0	1	1	1	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	1	0	0
1	0	0	1	0	1	0	0	1
1	0	0	1	0	1	1	0	0
1	0	0	1	1	0	0	0	1
1	0	0	1	1	0	1	0	1
1	0	0	1	1	1	0	0	1
1	0	0	1	1	1	1	0	1
1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0
1	0	1	0	0	1	0	0	1
1	0	1	0	0	1	1	0	0
1	0	1	0	1	0	0	0	1
1	0	1	0	1	0	1	0	1
1	0	1	0	1	1	0	0	1
1	0	1	0	1	1	1	0	1
1	0	1	1	0	0	0	0	1
1	0	1	1	0	0	1	0	1
1	0	1	1	0	1	0	0	1
1	0	1	1	0	1	1	0	1
1	0	1	1	1	0	0	0	1
1	0	1	1	1	0	1	0	1
1	0	1	1	1	1	0	0	1
1	0	1	1	1	1	1	0	1

Table A.1: A part of the simulation for SFF9