FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Online Test Vector Insertion – A Concurrent Built-In Self-Testing (CBIST) Approach for Asynchronous Logic

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Jürgen Maier

Matrikelnummer 0825749

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

Wien, 12.08.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Online Test Vector Insertion – A Concurrent Built-In Self-Testing (CBIST) Approach for Asynchronous Logic

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Jürgen Maier

Registration Number 0825749

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

Vienna, 12.08.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Jürgen Maier
Eschenweg 1, 2223 Martinsdorf

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

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all I want to thank my advisor professor Andreas Steininger for introducing me to that very interesting and thrilling field of application and his guidance while working on this thesis. I also want to thank my dear colleague Stefan Mödlhamer, who helped to improve the quality of this thesis with his indispensable input. Last but not least I would like to thank Sandra Burin for her patience while checking the thesis for grammar flaws as well as my parents for their never ending support throughout my studies.

Abstract

Testing electronic circuits during their operation in the field is mandatory to ensure correct functionality over a long period of time. To avoid fault accumulation test vectors have to be applied actively to the circuit under test, without disturbing the normal operation. In synchronous circuits this topic has been investigated thoroughly, however for asynchronous ones, a more and more emerging design paradigm due to its superior properties, only few test approaches are available. In this thesis a novel concurrent Built-In Self-Testing (CBIST) approach is presented, that is capable of testing asynchronous logic without interrupting the normal operation at any point in time. For that purpose the rather unproductive NULL-phase of a 4-phase communication protocol is replaced by dedicated TEST values, which are generated and analysed on chip. In detail two 4-phase input streams (user and test data) are combined to a 2-phase one, which is afterwards processed by the circuit under test and then split up into two 4-phase streams again. The units responsible for merging and splitting had to be implemented from scratch due to missing references in literature, both for the bundled data and completion detection communication style and in several versions, differing by their complexity and level of concurrency. The proposed test procedure has the advantage that the test data are independent of the user data and can therefore be defined already at design time. This yields several advantages, for example the possibility to test rather complex structures like cyclic pipelines. As our assessment shows, the price for the test approach in terms of increased hardware effort and additional delay is very moderate, especially for large circuits. For those reasons the proposed test approach is a good alternative if data processing must not be interrupted at all.

Kurzfassung

Um zu gewährleisten, dass eine elektronische Schaltung auch über längere Zeit gemäß ihrer Spezifikation arbeitet, ist es zwingend notwendig, diese im laufenden Betrieb zu testen. Zur Verhinderung von Fehlerakkumulation innerhalb der Schaltung müssen Testvektoren aktiv an die zu testende Einheit weitergegeben werden, ohne natürlich die normale Funktionsweise einzuschränken. Für synchrone Schaltungen wurde dieses Thema schon zur genüge erforscht. Für asynchrone Implementierungen, die sich aufgrund ihrer zahlreichen Vorteile immer stärker verbreiten, sind jedoch relativ wenig Testmethoden vorhanden. In dieser Arbeit wird deshalb ein völlig neuartiges Testverfahren präsentiert, das es ermöglicht, asynchrone Schaltungen zu testen, ohne die normale Funktion auch nur ein einziges Mal zu unterbrechen. Zu diesem Zweck wird die eher unproduktive NULL Phase eines 4-Phasen Kommunikationsprotokolls durch dezidierte Testvektoren ersetzt, die direkt am Chip erzeugt und analysiert werden. Im Detail werden zwei 4-Phasen Eingänge zu einem 2-Phasen Ausgang kombiniert, der anschließend von der zu testenden Schaltung verarbeitet und am Ende wieder in zwei 4-Phasen Ausgänge aufgespalten wird. Die Schaltungen, die das Verschmelzen bzw. Aufspalten übernehmen, mussten komplett neu entwickelt werden, da in der Literatur nichts Vergleichbares gefunden werden konnte. Diese Einheiten wurden für die bundled data als auch für die completion detection Kommunikationsmethode in unterschiedlichen Versionen implementiert, welche sich durch ihre Komplexität und den Grad der Parallelität unterscheiden. Die vorgestellte Testmethode hat den Vorteil, dass die Testdaten komplett unabhängig von den Nutzdaten gewählt werden können, was es möglich macht, diese bereits im Zuge der Entwicklung zu bestimmen. Dies hat etliche Vorteile, so gewährt es zum Beispiel die Möglichkeit, sehr komplexe Strukturen, wie etwa zyklische Pipelines, zu testen. Wie unsere Analysen zeigen, fällt der Preis dieser Testmethode, ausgedrückt in zusätzlicher Hardware und Verzögerungszeit, sehr moderat aus, besonders bei großen Schaltungen. Aus diesem Grund stellt die hier präsentierte Methode eine gute Alternative dar, wenn die laufende Datenverarbeitung auf keinen Fall unterbrochen werden darf.

Contents

CHAPTER 1

Introduction

Over the last few decades a tremendous shrinking of the feature size of electronic circuits has been observed, making it possible to implement more and more logic on the same die size. Together with the trends to higher speed and lower supply voltages [29] this leads to an increased sensitivity to disturbances, making it unreasonable to assume the correct functionality of a unit, once tested after fabrication, throughout its whole lifetime. The fact that more and more critical tasks, i.e. those where an unintended behaviour has severe consequence, are handled by integrated circuits drives the need to assure, that a unit is working along its specification. Some approaches, for example triple modular redundancy (TMR), are designed to tolerate a certain amount of faults. However, especially for long operation times, this yields several disadvantages, because it is commonly known that a TMR exhibits lower reliability compared to a normal approach, as soon as one replicate encounters a permanent fault [14]. For that reason test approaches are required that examine circuits and detect existing faults. A straight forward approach is to monitor input and output values of the circuit under test (CUT) and determine, if the output generated based on the inputs is valid or not. The big disadvantage of these methods is, that the test vectors are determined by the input data making it possible, that some parts of the circuit are not tested for a very long period, enabling fault accumulation [20, 29]. If all of these faults are activated later at once they may exceed the capabilities of the used fault-tolerance approach, which in general utilise the single fault model, i.e. they are able to handle one fault at a time. To prevent fault accumulation actively applying test values to the CUT during its normal operation becomes mandatory.

Asynchronous circuits are receiving increasing attention due to their superior properties compared to their synchronous counterparts [11, 12, 23]. In contrast to the latter ones no central clock is required to control the circuit, but instead the units inside communicate with each other by using handshake protocols. These are used to indicate the succeeding unit that new data are available or to tell the preceding one that the data have been processed and are not required any more. This mechanism makes the asynchronous design style event driven and timing much

more flexible, yielding an improved tolerance to process,voltage and temperature (PVT) variations. The main advantages, however, are that it naturally tackles the current problems with synchronous circuits, namely low skew clock networks and high power dissipation, due to its absence of a clock and the event driven working procedure. One important reason why asynchronous circuits, introduced already several decades ago, are not spread widely is their bad testability which results from the high level of concurrency. The fact that the single components of the CUT coordinate themselves results in a huge amount of possible states the system may be in, yielding a very high test effort.

The goal of this thesis is to tackle the bad testability of asynchronous circuits by implementing a novel online test approach, i.e. one that actively applies test vectors to the circuit under test during its normal operation. To achieve real concurrency the rather unproductive NULL-phase of a 4-phase handshake protocol[1] is utilised for testing. For that purpose the NULL values are replaced at the input of the CUT by dedicated TEST values and the resulting data stream is then processed. At the end the TEST values are removed again and checked for correctness, resulting in a unit, that uses the 4-phase protocol at its interfaces but the 2-phase one inside. In this thesis it is analysed what infrastructure is necessary to implement this test approach for the bundled data and completion detection design style together with a critical reflection on its properties.

Unfortunately the units inserting and removing the TEST values had to be designed from scratch due to missing references in literature. Several designs are presented in this thesis for these units, differing in complexity and level of concurrency. The circuit implementations are derived by designing the desired behaviour in a state transition graph (STG) and afterwards converting it automatically to a net list. To show the correct functionality of the approach a proof-of-concept implementation is used, which also states the basis for overhead considerations in the value and time domain.

This thesis is structured in the following way: In chapter 2 one will find a State-of-the-Art analysis of currently available implementations. In chapter 3 concepts as well as definitions used throughout the thesis are shortly described, followed by an overview of the used methodology in chapter 4. The chapters 5, 6, 7 and 8 then show the concrete implementation in theory, a proof-of-concept implementation, including an area and delay overhead analysis, is presented in chapter 9. Finally a critical reflection on and characterisation of the proposed test approach is carried out in chapter 10 followed by a final conclusion (chapter 11).

[1] An explanation as well as additional information on these topics follow in section 3.

CHAPTER 2

State-of-the-Art

Verifying that an electronic circuit is working according to its specifications is a well researched field in dependable computing. In this chapter existing approaches are presented, whereat two fundamental test methods are distinguished: concurrent checking and (online) testing. If the circuit under test has to be tested in the field it is crucial that no external units are required for testing, meaning that all necessary parts of the test approach have to be integrated on chip, also known as Built-In Self-Test (BIST). As one can imagine these approaches are of growing interest due to the increasing application of electronic devices in mobile applications. It would be unimaginable, if such a unit has to be connected to a separate test device to ensure its correct functionality. Many of the approaches shown in this section can be integrated as BIST approach, yielding more or less hardware overhead.

Concurrent Checking

Test approaches in this group observe the output and input of the CUT during its normal operation and determine, if the calculated result based on the input values is correct. To determine the correctness of the output values some kind of redundancy has to be introduced which however differs from approach to approach. One very favourable property of this procedure is, that it can be carried out completely concurrent to the normal operation, due to the fact that the input and output lines are only observed. In addition the results are available (nearly) in real time, yielding fantastic response times. This properties make these approaches very well suited to detect transient and intermittent faults (see section 3.10), however not for permanent ones. The reason is, that the test values, i.e. the input values, are not controllable, making it possible that certain parts of the circuit are not tested for a long time, implying the possibility of fault accumulation [20, 29].

The above mentioned redundancy can be introduced in many different ways, for example in the space domain. For that purpose either the CUT gets duplicated or a unit carrying out the same

functionality is implemented and connected to the same inputs. For error detection the outputs of both units are compared and if they differ an error has appeared [31]. Careful design of both implementations is advised to assure, that a fault is not present in both units, which would yield it undetectable. A very simple implementation for the additional unit is for example a lookup table realised by a storage element. This method works fine as long as both implementations do not produce identical errors, as outlined in [16], requires however a huge hardware overhead. In addition it is vulnerable to faults on shared resources like the clock signal. Researches have also discovered several shortcomings when used with asynchronous logic [41]. An even simpler approach also presented in [16] uses the additional unit solely to predict the parity of the output lines. This method however relies on the single fault assumption, i.e. that at a time only one fault exists. If a linear system, i.e. one having a linear relationship between input and output, has to be tested, the additional unit can be used to calculate the output analytically, as shown in [32].

A quite different approach uses codes for error detection. In detail the input lines are encoded leading to encoded output lines [20,26]. A fault is detected if the output value is not a valid code word, requiring additional data lines and therefore increased area. In addition the code has to be designed thoroughly, such that it is capable of detecting all modelled faults, which is a very challenging task. However it is possible that certain faults are only detectable by non-code words, making the approach incapable of detecting all faults, since non-code words will never be processed [18].

In addition to checking the logic values on the lines, [20] mentions the possibility to observe reliability indicators such as electric current, temperature, intermediate voltage, output activity and total dose to predict correct behaviour. By checking e.g. the power consumption it is possible to detect faults, that do not alter the output results, but for example increases the drained current [39], which may be a severe failure in modern low power applications. In addition it is possible to implement protocol checkers with this approach, assuring the correct temporal behaviour of a unit, due to the fact that CMOS circuits only consume power when they switch. A necessity for this approach however are current measurement units that are fast and precise enough for the given problem.

(Online) Testing

As mentioned before concurrent checking approaches allow fault accumulation. If even faults in rarely used parts of the CUT have to be detected reliably, as it is the case for units with long operation times, actively applying test vectors becomes mandatory. To detect a fault the output, like with concurrent checking, is investigated, with the difference, that the expected result can be precalculated. This makes it, at least in theory, possible, to check the semantic of the output signal and not just its syntax. Online testing, however, requires to interrupt the normal operation, implying a degradation of the delivered service. Therefore a proper integration of the test procedure yields a very big challenge. To assure efficient and fast error detection online test approaches are measured on the following quality criteria [14]:

- low performance penalty for the application
- high test coverage for a given fault model, determined by the quality and amount of test vectors
- low error detection latency, determined by the period required to apply the whole set of test vectors

Two fundamental properties have to be fulfilled by an online test approach:

non-interference in time domain The normal operation must not be delayed beyond the point that deadlines are missed. This can be assured by either including the test procedure into the schedule, which decreases the response time, or by making the test process preemptive [29]. For example is it imaginable to start the test procedure when the circuit is idle or at fixed points in time, independent of the current workload.

non-interference in value domain The internal state of the circuit must not be altered during the test. More specifically the first user data value that is processed after the test has finished must see the same environment as if it was processed right before the test. For that reason so called "transparent" test approaches are required, which restore the system state after they have been executed. [19] shows such a test approach for RAMs, where several operations are carried out on the data, eventually leading back to the original values (for example two times XOR). Another transparent approach is described in [1], where reconfigurable blocks in an FPGA are tested before configuration.

Most online test approaches use two distinct operation modes: In the normal mode the circuit processes user data and forwards the achieved results, whereas the test circuitry sleeps. Only after the circuit switches to the test mode, in most cases realised by an input line, the inputs of the CUT are redirected (for example by MUXes) to the test vector generator and the outputs to the test response analyser. The purpose of the latter is to compare the received results to the expected ones and indicate an error if they differ. It would of course require lots of memory to store the response to each single test vector, therefore so called compactors are used that generate unique values based on the received results [23, 25, 27]. Concrete implementations can be found for example in [17,21], which mainly differ in the way the test vectors are generated and analysed. Please note, that for certain approaches the test response analyser, i.e. the compactor, can be replaced by a checker [12]. As mentioned testing is only carried out when the circuit is in the test mode. When and how long it enters that state largely depends on the specific application and has to be determined at design time.

A different approach, called scan chain test, uses the internal storage elements to set the CUT to a specific state. Therefore all internal storage elements are connected to a linear chain, making it possible to shift the desired setting into the circuit under test. After the state was set, normal operation is started, but only for a very short amount of time. Afterwards the connected

storage elements are read out in the same fashion as before and the received state is compared to a reference value. This approach makes it possible to test each imaginable system configuration making it a very extensive test method. The downside is, as one can imagine, that it takes quite some time to shift in a new state and afterwards shift out the result. This implies that the normal operation also has to be interrupted for a long time. Furthermore in general additional devices are required to compare the resulting state to the intended one, making it not well suited for online testing. For synchronous circuits the amount of steps the test approach carries out can be controlled very well by the clock signal, which however is not available in asynchronous ones. Therefore the integration in these is a lot harder, but has already been achieved in several approaches [2, 23, 37].

Due to the dramatic shrinking of transistor sizes more and more logic can be placed on a single chip, making it possible to implement general purpose processors in embedded applications. This yields the possibility to shift testing to a higher level of abstraction, i.e. into the software domain, also called Software Based Self Test (SBST) [22]. In this domain complementary test methods are possible: A test procedure especially designed to activate every part of the chip is run from time to time and the generated results are analysed. The main difficulty in this case is the development of that specific test procedure. Of course also non-interference in the time domain has to be assured, which however can be implemented by using the scheduler of the processor [3]. Redundancy can also be introduced in the time domain, for example by double execution. In that case the calculations are carried out twice and the results of both runs are afterwards compared [10]. This makes it possible to detect transient faults that only effect a single computations however permanent ones, altering both, can not be detected. Even more elaborate mechanisms, exploiting the possibilities of processors, are realisable. The approach in [7], for example, uses the debug port to observe the control flow inside the processor. In detail the checker controls which branches are taken and generates checksums on the executed commands to assure the correct functionality.

Unfortunately no completely concurrent test approach, actively applying test vectors, could be found in literature. A very interesting approach, combining properties from concurrent checking and online testing, is called input vector monitoring Concurrent BIST [46]. This method determines a subset of all possible input vectors as test set. As soon as a new input vector is applied to the CUT it is checked if that value is part of the test set and if it has not been processed in the current cycle. If that is the case the output of CUT is compressed in a compactor. After all vectors of the test set have shown up the computed signature is compared to a precalculated value. Due to the fact that the input values are not controllable it might happen, that the set is never received completely, yielding the test procedure unfinished and stuck. To prevent this, the circuit switches to a special test mode if testing was not finished within a certain time interval and the still missing vectors are actively applied to the input. The earliest implementation of this approach has been proposed in [28] and was afterwards extended several times in w-MCBIST [46], SWiM-BIST [44], MICSET [48], R-CBIST [49], exploiting “X” values [43], NEMO [45] and w-CBIST [47], just to name a few. These differ mainly in the way the input vectors are detected as member of the test set. In the first version the input value is compared

to a value generated by a generator unit. Only when that value is detected the next one is generated. In future implementations this mechanism has been improved for example by window based approaches, capable of comparing more vectors at the same time, or by utilising RAMs.

Asynchronous Circuits

Most of the available test procedures were developed for synchronous circuits and afterwards adapted to asynchronous ones. Due to the fact that the two implementations may differ significantly (e.g. handshake lines, dual rail) not all possible faults are detectable by these approaches. For example is it a common mistake to assume that faults on the control lines cause a circuit to halt automatically; in fact that is only true for stuck-at faults under various constraints as pointed out in [30]. Others may for example introduce additional transitions, causing severe malfunctions of the CUT. Therefore protocol checkers have been developed [30, 52], which check the control lines not only in the value but also in the time domain. In the latter case the correct temporal order of the signals is assured, for example that a new value is only indicated after the old one has been acknowledged or that an acknowledge signal is only sent after new data values have been received.

Asynchronous circuits are in general harder to test due to the missing clock signal and the therefore increased concurrency of the single components of a chip. This makes it (nearly) impossible to verify each possible system state in an offline test. Therefore online test approaches, like the one proposed in this thesis, become even more valuable, because they are capable of testing the unit during its normal operation.

CHAPTER 3

Background on Asynchronous Logic

The purpose of this chapter is to give a short overview of the concepts and definitions applied in the thesis. For more detailed information on each topic follow the references stated in the text.

3.1 Asynchronous Circuits

Only a small percentage of today's circuits are asynchronous, despite the fact that they have been invented a long time ago. However nowadays the number of implementations is rising due to their advantages compared to synchronous logic. The following listing gives a short overview over the most important properties of asynchronous logic and the main differences to the synchronous one.

Asynchronous circuits ...

- possess no central clock. The units coordinate themselves using handshake signals, indicating that data are available or have been processed. This mechanism also renders time consuming timing analysis unnecessary.
- are more robust against additional delays introduced for example by environmental changes, e.g. temperature or voltage. They simply lower their working speed but will continue to deliver results, not fail completely as synchronous ones do, when timing violations occur.
- consume less power compared to synchronous circuits because they only work when new data are available. In the synchronous case the clock drives the circuit always, even if nothing has to be done. In addition the clock network itself consumes a lot of power too [1].

[1] Please note that power saving mechanisms like clock gating were introduced in synchronous designs to tackle those problems.

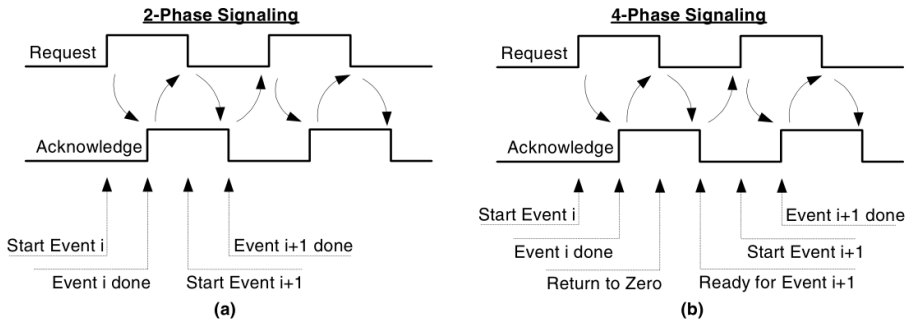


Figure 3.1: (a) 2-Phase and (b) 4-phase Signalling [41]

- have a high level of concurrency making development and testing very difficult. In addition they have to be designed very thoroughly to prevent glitches and spikes.

- lack design tool support because they are not used widely, making development even harder.

3.2 Handshake Styles

The process used to coordinate the data exchange, i.e. the temporal order of sending and receiving data between two units, in asynchronous logic is called handshaking. Two fundamental styles (see also figure 3.1) are used for this purpose:

4-phase

The name yields from the fact that there are four distinct phases in a complete transmission cycle. In the first one the sender indicates new data to the receiver which the latter acknowledges in the second phase. In the third one the sender returns to its base state which is again acknowledged by the receiver in the fourth phase. This method is also called Return To Zero (RTZ) or level signalling.

The 4-phase communication style uses two differing phases [2], i.e. the DATA-phase holding the actual value and the NULL-phase used to separate two consecutive DATA values.

Due to the fact that efficient and simple robust function blocks are harder to build using the 2-phase style, combinational gates are in general realised using the 4-phase communication style [15].

[2]These are not connected to the phases the expression 4-phase refers to.

2-phase

As the name indicates this style needs only two phases for a single transmission. The sender indicates new data in the first phase and the receiver acknowledges it in the second one. This method is also called Non Return to Zero (NRZ) or transition signalling.

When using 2-phase communication DATA values are sent consecutively without any separating value in between. To recognise the end of the old value and the beginning of a new one the data are sent in two alternating phases, which are recognisable by the receiver, similar to the NULL-phase and DATA-phase of the 4-phase protocol.

The fact that only half of the phases are required to send the DATA values makes this style faster than the 4-phase one and also more power efficient. However in general the implementation of efficient function blocks is very difficult [15], which is the reason why it is mainly used for long communication lines connecting two units that internally use the 4-phase style.

3.3 Bundled Data Approach

A configuration is called bundled data if in addition to the data lines separate request and acknowledge line are installed, which are used to indicate that valid data are ready to be processed or that the data have been read, more specifically to carry out the handshake protocol. A detailed description on bundled data is stated in [34, pp. 9-11].

A very important property of bundled data used in this thesis is the necessary delay on the request line, which is required to assure, that the request signal reaches the next storage element only after all input lines to that unit already got stable. The appropriate delay has to be determined at design time by thorough timing analyses and simulations.

The bundled data approach can be seen as an intermediate step between synchronous and DI[3] asynchronous logic. On the one hand it is possible to remove the clock signal however on the other hand thorough timing analysis, which are very time consuming and hard to execute, are still mandatory.

The handshake styles described in section 3.2 can be realised by a differing handling of the request line. In the 2-phase style each transition either from high to low or from low to high indicates new data whereas with the 4-phase style only a high value on the request line has that meaning. Due to the fact that no new data are sent in the NULL-phase, a delay of the request line is not necessary in that particular case. Therefore an asymmetric delay line, that only delays the high value of the request line, may be used in the 4-phase style. Please note that the same logic cloud can be used for both styles; only the pipeline nodes have to be changed due to the differing meanings of the transitions (single edge versus double edge) on the control lines.

[3] More information on that follow in section 3.5

3.4 Completion Detection Approach

The completion detection approach [34, pp. 11-13] uses for each data bit multiple wires, in the following called rails, for transmission. This enables the data themselves to indicate when they are ready to be processed instead of using a separate line for that purpose, which not only renders timing analysis unnecessary but also increases the stability.

Using a 4-phase protocol in conjunction with completion detection yields the dual rail NULL convention logic (NCL), where one rail is named high rail and one low rail. A high value on the high rail indicates that the sent signal is logic high, and a high value on the low rail stands for a logic low value. Both wires carrying a high value is not allowed and never appears if every unit is working correctly. The NULL-phase used to separate two consecutive DATA values is identified by a low value on both rails.

Level encoded dual rail (LEDR), the 2-phase protocol chosen in this thesis, transmits two consecutive DATA values without any value in between. To clearly distinguish between old and new data, two phases are introduced, one where the parity of the two rails is even and one where it is odd. To signal the receiver a new value only the phase has to change, which can be done by changing a single rail. When looking at both rails one can see that one of the lines holds the actual value and the other one is responsible to generate the correct phase. If the DATA value between two succeeding phases is the same just the phase rail has to toggle to signal a new value. If the value changes the value rail itself toggles automatically generating a different phase and therefore indicating a new value.

Besides size and complexity another very important criterion has to be investigated when using the completion detection approach, namely timing assumptions. A detailed explanation follows in section 3.5.

3.5 Timing Assumptions

Timing assumptions express what timings have to be fulfilled to make the circuit work as specified. The fewer the assumptions that have to be made, the more robust the concrete implementation gets. A short summary of already specified assumptions shall be given here, a more detailed description can be found in [34, pp. 9-28].

DI Delay Insensitivity is the least restrictive class allowing arbitrary but finite delays. Only very few circuits are really DI because in general only inverters and Muller-C elements are allowed [11]. In special configuration also other gates may be used without loosing DI characteristics.

A clear sign that DI is not fulfilled is the violation of the *indication principle* ([34, pp. 14-16]) which states, that every change at the input of a gate has to be recognisable also at its output. If, for example, an OR-gate forwards a low value it can be said that both

inputs are low, however if the output is high, it is not possible to distinguish if both of just a single input lines holds a high value.

QDI Quasi Delay Insensitivity is very similar to DI with just one additional restriction. If a signal line forks, i.e. is split up and directed to two different units, the signals on both lines of the fork have to have the same delay, also called *isochronic-fork*. This implies that a signal sent reaches both receiving units at the same time, which may be a very challenging task, as pointed out in [11]. Furthermore the *isochronic-fork* property may be violated during operation due to environmental properties such as a local temperature deviation.

SI A circuit in general consists of several gates that are interconnected and together compute the desired result. However different paths through these logic gates might have a longer delay than others, causing gates to switch at different times, producing glitches or simply incorrect results. A circuit is called Speed Independent if no unintended behaviour is possible as a consequence of gates switching at different points in time. Due to the fact that the tool *Petrify* (section 3.8), used to develop the implementations in this thesis, only generates SI circuits, this timing assumption is fulfilled by all implementations. In addition [11] claims, that QDI and SI are identical for practical purposes, making it possible to assume all designed circuits QDI as well.

Please note that there exist additional, more restrictive, timing assumptions which are however neglected in this listing, because in this thesis only DI, or if that is not possible QDI, circuits are presented.

3.6 Data-Validity Schemes and Channel Types

When using the bundled data design style a separate line, more specifically the request line, indicates that new data are available, however there are different possibilities when the data really are available at the input of the next unit. Furthermore two different channel types can be distinguished, more specifically the push and the pull channel. In the first case the sender starts a transmission by indicating that new data is available, in the latter case the receiver indicates that it is ready to receive new data which causes the sender to transmit them if available. All these approaches are shown and described in detail in [34, pp. 116-117]

In this thesis the early data-validity scheme on a push channel is used. In detail the data lines hold valid data as soon as the request signal reaches the receiver and get invalid as soon as the acknowledge signal is received at the sender. This implies that the receiver has to read and process or store the DATA values before the acknowledge signal is sent.

3.7 Signal Transition Graph

Signal Transition Graphs [11], or short STGs, are a special form of a Petri Net [34, pp. 86-113], which specify the temporal order of transitions on input and output lines of a component. In this thesis these graphs are used to generate concrete circuit implementations using the tool *Petrify* (section 3.8).

In general an STG consists of places, arcs and special nodes, which are used to split or combine paths. Due to the fact that the latter are not used in this thesis they won't be described in more detail.

place Places in an STG represent a transition either at an input or output line. The transition is only allowed to occur, in the following also called to fire, if all its input arcs hold a token. After the place fired the tokens of the incoming arcs are removed and each outgoing arc receives a single token.

arc An arc connects two places and is able to hold one or more tokens. At the beginning an initial marking has to be introduced to start the data processing at the intended position.

The STGs shown in this thesis encode the signals using the naming scheme xYs whereas $x \in \{a, r, p\}$; $Y \in \{U, T, UT\}$ and $s \in \{+, -\}$. The first letter specifies the type of the line, which is either an acknowledge (a), request (r) or a phase detector (p). The letters afterwards determine which data are delivered on the line, i.e. either the 4-phase types Test Data (T), User Data (U) or the combined 2-phase version User/Test Data (UT). Finally a $'+'$ sign at the end indicates a rising edge and a $'-'$ a falling one.

3.8 Petrify

"Petrify is a tool for synthesis of Petri nets and asynchronous controllers." [24] In this thesis it is used to convert STGs to netlists, i.e. into specific circuit implementations. For this thesis version 4.2 compiled 13-Oct-03 at 3:06 PM was used, which can be downloaded at [24]. The used gate library can be found in section B.1.

3.9 Circuit Drawings

Some of the developed circuits were also drawn using the gates shown in table 3.1. In the figures the inputs (labels aXX or rXX) are always displayed at the left side, the outputs (labels aXX or rXX) at the right and signals produced and consumed inside the unit (denoted by everything except aXX or rXX) at any side. Please note that the term input or output declares the data signals, the acknowledge line and, if used, the request line as a whole where an input is recognised by an incoming flow on the data lines and an output by an outgoing one, as it is also used when describing the overall functionality of a unit. The fact that all signals are

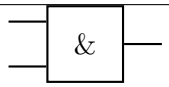
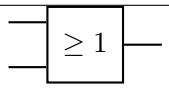
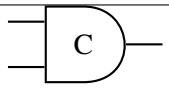
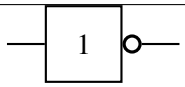
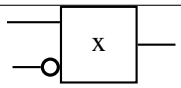
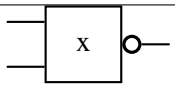
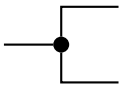
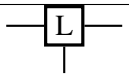
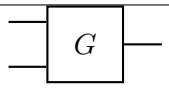
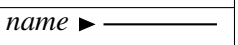
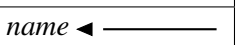
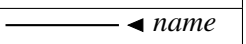
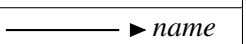
gate	symbol
AND	
OR	
Muller-C	
Negation	
Negation at gate input	
Negation at gate output	
Connection	
Latch	
Gatekeeper	
Line accepting information	
Line providing information	
Line accepting information	
Line providing information	

Table 3.1: logic elements and the corresponding symbols used in graphical circuit representations

referred to by a single term however implies that the acknowledge signal of an input is actually a line providing information to the outside and the acknowledge signal of an output is actually receiving information from the outside. For that reason the signal direction is indicated on the single lines by arrows, as they are shown in the last few lines of table 3.1.

3.10 Fault Types

To classify fault sources it first of all has to be stated what the difference between fault, error and failure is. A failure appears, if the behaviour of a unit deviates from its intended one. The cause of a failure is an error, i.e. an incorrect system state which has been caused by a fault, an unexpected environmental property [13]. Based on these definitions the following classes of faults are distinguished [5]:

1. *permanent faults:* Faults in this category are first of all reproducible meaning that a vector that activates this fault will do this every time. This implies that the fault does not vanish by itself but it has to be removed actively. An example for this class are hardware defects like a broken wire.
2. *transient faults:* Transient faults are only introduced once and for a short time. Due to the increasing miniaturisation of circuits and the steady improvements in speed the importance of this class grows bigger and bigger. The typical example is cosmic radiation which causes voltage spikes on signal lines.
3. *intermittent faults:* Faults in this class are in general present all the time, however they only introduce an error if a specific trigger condition is showing up. Despite the fact that these faults are reproducible, they are in general very hard to detect and may be easily confused with transient ones. A typical example for this class is a software fault, that only produces a wrong result if a specific input value is received.

3.11 Circuit Characteristics

To describe the capabilities of a test circuit, [31] proposes three properties which are also used for analysing the characteristics of the test scheme proposed in this paper in chapter 10.

1. *Latency of Test Completion* (LTC): This property describes how long it takes the test circuit to completely test the circuit during its normal operation, i.e. that all test vectors have been applied to the CUT.
2. *Latency of Fault Detection* (LFD): This attribute represents the time it takes the test circuit to detect a fault after it has occurred. Again it is calculated under the assumption that the CUT is working normally while testing is carried out.
3. *Error Latency* (EL): This indicator determines how long, in average, incorrect values are possible until the corresponding fault is detected. It is calculated as the difference between the *LFD* and the latency of fault manifestation (*LFM*) which indicates how long it takes until a fault generates a failure. The smaller the EL is the better the test circuit works. Please note that it is also possible for the EL to become negative indicating that faults are detected before they generate failures.

CHAPTER 4

Methodology

In a 4-phase protocol the exclusive task of the NULL-phase is to clearly separate two consecutive DATA values. More specifically the circuit does not process any data during that time but literally halts until the next DATA-phase starts. This behaviour introduced the question if it is possible to use the circuit during the NULL-phase for other useful purposes, like in this thesis to test the circuit without actually interfering with the original computations.

The easiest way to accomplish this assignment is to replace the NULL values by additional TEST values, used to actually test the CUT. The communication protocol is converted to a 2-phase type by carrying out this step, because afterwards a TEST value immediately follows a DATA value. For this purpose additional units are required that are capable of merging DATA values and TEST values into a single 2-phase output and another one that splits a single 2-phase input into DATA values and TEST values. At first an extensive literature research was started to find out if units having that particular behaviour already exist. Unfortunately this was not the case because no application could be found that propagates two input values one after the other to the output. The applications that were found always wait for both inputs to hold a DATA value and then both of them are sent at the same time.

For that reason the appropriate units had to be designed completely new, which was carried out by modelling state transition graphs (STGs) of the expected behaviour. These were then transformed into concrete circuit implementations using *Petrify* and finally checked for further simplifications. The top priority in this thesis was to identify the easiest and smallest ways to carry out the above described functionality. It turned out, that it reduces the size of the resulting circuits a lot, if the TEST values are delivered in the same way as the DATA values, i.e. by using a 4-phase communication protocol. This leads to the following desired functionality of the additional units:

Merge This unit only propagates the DATA values of the inputs in an alternating fashion to the output, the NULL values are dropped. If necessary a format conversion [1] is carried out on the data to assure a valid 2-phase communication at the output.

Split This unit propagates the DATA values of the 2-phase input in an alternating fashion to the two 4-phase outputs with optional necessary format conversions. Between any two consecutive DATA values a NULL value has to be inserted to create a valid 4-phase communication protocol.

While developing an efficient implementation of these two units several versions were discovered, differing mainly in their level of concurrency and their hardware requirements. Due to the fact that each of them has its pros and cons no "best" solution could be determined and therefore several versions are described and analysed in this thesis.

After all theoretical preparations were finished, an actual implementation using the proposed test approach was created. A simple three stage pipeline, was chosen as the CUT, which was afterwards extended by the proposed test approach. To verify the correct functionality the implementation was simulated and it was checked, if the output was according to the results achieved with the original pipeline. The internal test procedure was verified by introducing a stuck-at fault and it was observed, if it is detected correctly, i.e. if an error is reported to the outside world.

In addition the area overhead and introduced delay were estimated analytically to achieve generic formulae for an arbitrary amount of computational logic. The received results were then used to calculate characteristic values for the proposed test approach, which were afterwards analysed to determine the properties of the proposed test approach.

[1] If more than one rail per bit is used a conversion is mandatory.

CHAPTER 5

Proposed Solution - Overview

This chapter gives a first introduction to the proposed test approach. At first the concept is developed step by step and afterwards the necessary components are presented as block diagrams. Furthermore the idea of different implementation styles is presented however without showing concrete circuit designs, which is the topic of the following two chapters.

5.1 Introduction

As it was shown in section 3.2 the NULL-phase in a 4-phase protocol has solely the purpose to separate two DATA values, implying that it does not contribute to a computation at all. This fact begs the question, if that particular phase may be used for additional tasks, rather than just separating DATA values. In this thesis it is therefore investigated, if it is possible to implement a self-testing circuit that uses the time available in the NULL-phase to test the actual circuit and to develop appropriate implementations, if there are any.

To test a circuit, the TEST values have to use the same path as the DATA values. To achieve this, the data input has to be blocked while the test vector is processed, because otherwise both may interfere. Due to that fact the core idea of the proposed approach is to replace the NULL values by TEST values, and afterwards feed the modified input stream into the circuit under test (CUT). At the output of the CUT the test vectors are replaced again by NULL values to achieve a correct 4-phase protocol. In parallel the processed test vectors, which were replaced by NULL values, are checked against precalculated ones. If no differences are detected one may assume that the application related calculations in the CUT were carried out without failures as well and the results can be assumed to be correct, however if the test result was not identical with the stored value a failure occurred and the circuit indicates this to the outside world.

By replacing the NULL-phase of a 4-phase protocol with another DATA-phase the protocol is converted to a 2-phase one because in that case DATA values succeed one another without

any value in between. For the purpose of *merging* User Data and Test Data a unit has to be developed, that has two separate inputs, one for User Data and one for Test Data, and one output for the combination of both, using the 2-phase communication protocol. In a similar fashion also another unit is needed that *splits* the 2-phase User/Test Data stream up into Test Data and User Data, at which the User Data output has to use a valid 4-phase protocol, to assure correct functionality at the output of the CUT.

5.2 Approach

As starting point of all considerations a linear asynchronous Muller pipeline (i.e. without loops) using a 4-phase protocol like in figure 5.1 is chosen. Within that structure no registers are allowed, that store values from the previous computation for the next one, like the status register in an arithmetic logic unit (ALU). Altogether no interactions between different computation stages or succeeding values are allowed, as it is the case in state machines calculating their actual output based on their internal state and the input or when loops in the data path are used. This very restrictive model will be used to develop and exploit the fundamentals of the approach, however in chapter 8 it is investigated if some of the stated restrictions may be lifted under certain conditions.

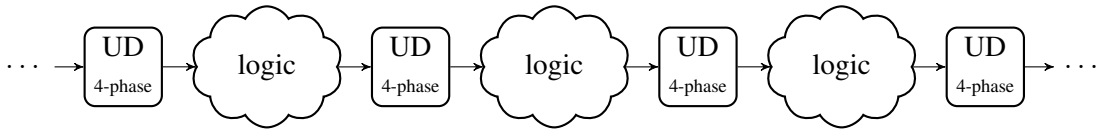


Figure 5.1: original Muller pipeline structure used as basis of considerations, multi-stage possible, 4-phase communication protocol

The nodes in the picture represent a controller with integrated latches, i.e. a complete pipeline unit. A subset of these, namely the ones that have to be tested, are then chosen, forming the Circuit Under Test (CUT) (figure 5.2[1]). After the CUT was formed it is transformed to a self-testing circuit, by integrating additional units[2]. Please note that it does not matter if the additional units are placed before or after the logic cloud. The only difference, of course, is that the logic will be tested if it is placed inside the CUT and not if it is outside.

One of the additional units, as already mentioned, is responsible for merging the inputs. Figure 5.3 shows its inputs and output over time. Please note that the Test Data input was also chosen as 4-phase input, resulting in simpler circuits. Due to the fact that this building block converts two 4-phase inputs into one 2-phase output it is named 4-to-2 phase merge and gets described in detail in section 5.5.

[1] In this figure the logic clouds are not shown for better readability.
[2] Details on this procedure follow in section 5.3 and 5.4

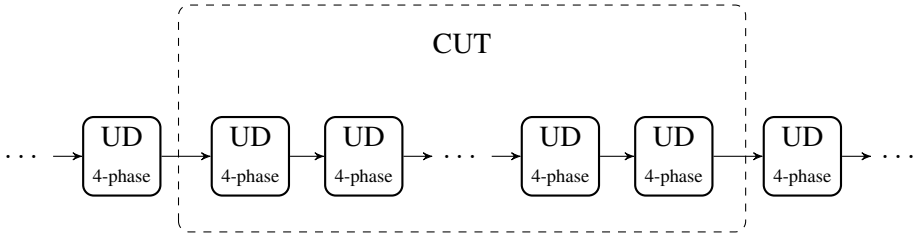


Figure 5.2: original Muller pipeline structure used as basis of considerations, CUT already determined, logic clouds are dropped for better readability

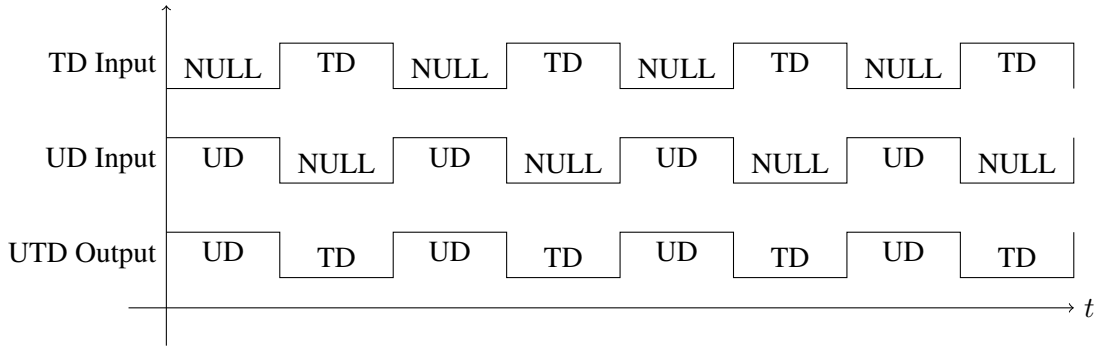


Figure 5.3: Input and Output of the 4-to-2 Phase Merge unit over time

To remove the TEST values from the data stream at the end of the CUT, more specifically to split User Data and Test Data, another additional unit has to be integrated. It is exactly the opposite of the one before, i.e. taking a 2-phase User/Test Data input and splitting it up into a 4-phase Test Data output and a 4-phase User Data output. The latter one has to use the 4-phase protocol, whereas this property is not mandatory for the Test Data output, however as before the 4-phase style was chosen to simplify the resulting circuit implementation. Figure 5.4 shows the temporal progress of the input and the outputs of this particular unit. Due to the fact that it splits a 2-phase input up into two 4-phase outputs this building block is named 2-to-4 phase split and gets described in detail in section 5.5.

The test vectors fed to the 4-to-2 phase merge unit are produced by a test vector generator and the ones delivered from the 2-to-4 phase split unit are analysed by a test response analyser. Due to the fact that these units can be implemented in many different ways they are described in detail in section 5.7, where also the selection and generation of appropriate test vectors is discussed.

Two implementation styles, describing how the conversion from the Muller pipeline to a self-testing circuit is carried out, have been developed, namely *Complete CUT Testing* (CCUTT) and *Single Stage Testing* (SST). These will be described in more detail in the following sections.

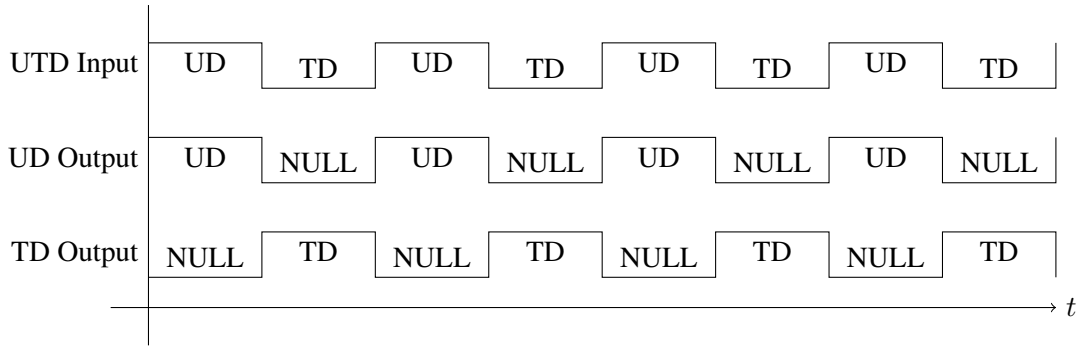


Figure 5.4: Input and Output of the 2-to-4 Phase Split unit over time

5.3 Complete CUT Testing

This design method is the most straight forward approach. The NULL values of the User Data input are replaced by TEST values at the beginning of the CUT and removed at the end. On the processed Test Data error checking is carried out to determine if failures have occurred.

In detail the 4-to-2 phase merge unit is placed right at the entry of the CUT, or more specifically in front of the first pipeline node inside the CUT. The resulting 2-phase data flow is then injected into the original pipeline structure, which has to be adopted to the 2-phase communication protocol first. If the bundled data method is used this adaption only concerns the controller nodes, if completion detection is used solely the logic in between has to be adapted. At the end of the CUT Test Data and User Data are finally split up again in the 2-to-4 phase split unit. The split up Test Data are afterwards checked (details see section 5.7) and the outside world is informed through a signal line if the received result matched the expected result.

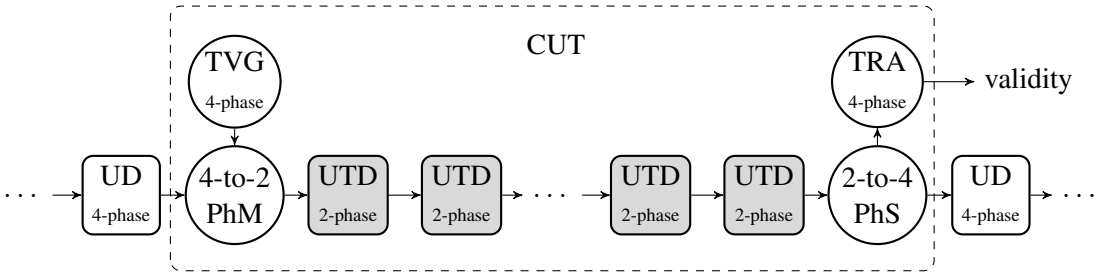


Figure 5.5: complete CUT Testing Approach, Schematic View

Figure 5.5 shows the approach, which is very similar to the original pipeline structure (Figure 5.2). Altered nodes, i.e. those transformed from 4-phase to 2-phase, are represented as coloured nodes, newly introduced ones, in detail the 4-to-2 phase merge (PhM) and 2-to-4 phase split (PhS) unit as well as the test vector generator (TVG) and test response analyser (TRA) are

displayed as nodes with a circular shape. As mentioned before, also the logic clouds, that are not shown in this figure, may have to be converted, depending on the used communication protocol.

When comparing the data interfaces of the CUT to the ones in the original pipeline (figure 5.2) one can see, that they are the same. This means that the test procedure is carried out transparent to the outside application, which is a necessary condition to test only parts of a longer pipeline. Furthermore this property implies that one can not tell if testing is carried out inside the unit by just looking at the data interface.

5.4 Single Stage Testing

Another possibility to verify the correct behaviour of the CUT is to test each pipeline stage individually. In contrast to the Complete CUT Testing style described before only the logic clouds between the pipeline nodes are tested in this approach.

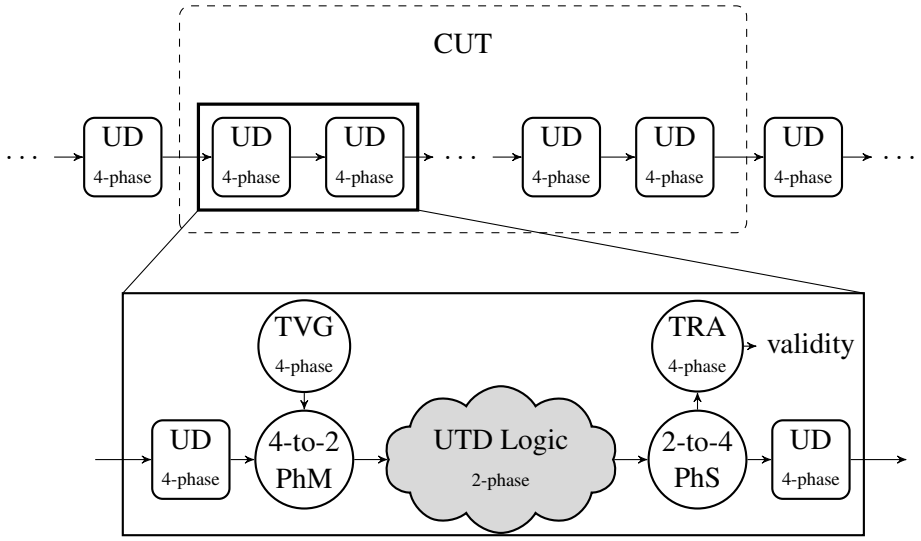


Figure 5.6: Single Stage Testing Approach, Schematic View, only shown for first stage, in real implementation test circuit has to be integrated between every two consecutive pipeline nodes

For that purpose all parts of the test circuit (test vector generator, test response analyser, 4-to-2 phase merge, 2-to-4 phase split) have to be implemented between every two consecutive pipeline nodes inside the CUT. Figure 5.6 shows this at the example of the first stage, using again a circular shape for newly integrated nodes. The logic cloud in between is coloured because it has to be converted from 4-phase to 2-phase when the completion detection design style is used, whereas no conversion at all has to be carried out when bundled data is chosen. Test Data and User Data are merged right after one pipeline controller, sent through the logic, which was adopted to 2-phase, and split up in front of the following controller node. The split up Test Data are checked for errors and the result is presented on an output line, in the figure named validity,

which is afterwards connected to the validity lines of all other stages in a way, that a mismatch is indicated to the outside application as soon as a single stage failed.

Advantages:

- The testing works transparent to the pipeline nodes i.e. they do not have to be modified, because the conversion to a 2-phase protocol is carried out after the output of the first node and the conversion back to a 4-phase one before the input of the next one. Therefore only the logic has to be converted to 2-phase. If the bundled data approach is used even this task can be dropped because in that case the logic for 2-phase and 4-phase is the same. The only task that might remain is to replace asymmetric delay lines by symmetric ones.
- By introducing a test circuit into each single pipeline stage, it is possible to test every one of them separately. Therefore the specific properties of each stage can be tested individually and not combined with others as in the previous case. This will in general reduce the effort of finding the right test vectors and may also reduce the effort of designing and implementing the TVG and TRA.

Disadvantages:

- The TVG and TRA design might get easier, however for each single stage one is needed. With an increasing stage count the effort may raise to a level that the overall design and implementation time exceeds that of a complex one.
- Errors in the stage controllers themselves are not detectable by this approach. That is the reason why an additional test procedure is necessary, for example a parity check.

5.5 Transformation Blocks

While searching for literature for this thesis many different transformation circuits were found; some convert a 2-phase into a 4-phase protocol and others transform between protocols of the same phase type [15]. Even conversions between bundled data protocols and completion detection ones have already been introduced [8, 9]. However no publications could be found describing an approach for combining two 4-phase inputs to one 2-phase output or reverse. That is the reason why these building blocks had to be designed completely new in this thesis.

The 4-to-2 phase merge as well as the 2-to-4 phase split block can be integrated as a switch or as a pipeline node. The first method forwards either the User Data or the Test Data input but does not store any data, yielding the advantage, that very small implementations are possible. If the latter method is used an improved decoupling of in- and output can be achieved, resulting in an increased working speed at still very little circuit complexity. The big disadvantage of this

method is the additional latch, which can be seen as another failure source, making it more probable for an error to occur. At least these additional failures are detectable by the test procedure proposed in this thesis, as long as the latch is placed at the combined User/Test Data line.

All latches introduced are defined to become transparent, i.e. propagate the value at their input to their output, whenever the enable input is high and store their actual value if the enable input is low. In the first case the latch is also called open, in the latter one closed throughout the thesis.

As it was described in the sections 3.2, 3.3 and 3.4 the 2-phase protocol sends the DATA values in two alternating phases. The fact that always exactly one 2-phase channel and two 4-phase channels, served in an alternating fashion, are integrated in one transformation block, makes it possible to assign each 4-phase channel to a specific DATA-phase of the User/Test Data line. This yields the advantage, in contrast to other approaches like in [8], that phase relations can be integrated hard wired and need not be calculated dynamically. This reduces the complexity of the circuit and also increases the speed. In this thesis the User Data channel is assigned to phase 1 i.e. when the request signal is high using the bundled data approach and odd parity when using the completion detection approach. In contrast the Test Data channel is assigned to phase 0, i.e. low request signal respectively even parity. Of course all following considerations are still correct when this assignment is interchanged i.e. User Data to phase 0 and Test Data to phase 1 however then some of the presented circuits have to be altered as well.

Due to the fact that the implementations for the bundled data and the completion detection approach differ a lot, detailed implementations are described in the chapters 6 (BD) and 7 (CD). The implementations are designed using STGs at which the starting configuration of User Data, Test Data and User/Test Data signals are defined as:

- The first action to come is a rising transition on the input (either User Data or User/Test Data)
- The Test Data Input/Output has already delivered its NULL-phase / acknowledged its DATA-phase at startup.

The main guideline when designing the test circuit was to keep it as simple and small as possible. One of the reasons for this choice was to keep the amount of necessary, additional hardware very low. Furthermore this thesis is supposed to investigate if such an approach is possible at all, so the driving force was to find any, not the best or fastest, implementation. That is the reason why additional complexity for increased speed was avoided.

4-to-2 Phase Merge Unit

As mentioned in section 5.2 the 4-to-2 phase merge unit combines two 4-phase inputs, i.e. User Data (UD) and Test Data (TD), to a single 2-phase output, i.e. User/Test Data (UTD). This

is achieved by replacing the NULL-phase of one input signal by the DATA-phase of the other one and only propagating the combined data stream. The User Data values are thereby always assigned to phase 1 and the Test Data ones to phase 0. A time diagram of this process was already shown in figure 5.3. In reality the input signals naturally will not be aligned as perfectly as it is shown in that picture, but with a specific offset to each other. The point in time when the signals are propagated to the output in these cases depends on the design of the merge unit. Several different implementations are possible, differing in complexity and their level of concurrency. More details on these follow in section 6.2 for the bundled data approach and in section 7.2 for the completion detection approach.

In the STGs describing the behaviour of the single implementations the request and acknowledge line of the User Data input are named *aU* respectively *rU*, those for the Test Data input *aT* and *rT* and the ones for the output User/Test Data *aUT* and *rUT*, according to the naming guidelines presented in section 3.7.

2-to-4 Phase Split Unit

As already described in section 5.2 the 2-to-4 phase split unit splits one 2-phase input, i.e. User/Test Data (UTD), into two 4-phase outputs, i.e. User Data (UD) and Test Data (TD). Therefore this block propagates all the values of phase 1 to the User Data output and the values of phase 0 to the Test Data output. In addition the unit has to insert NULL values between two consecutive DATA values on each output to create a valid 4-phase protocol. A time diagram of this process was already shown in figure 5.4.

As with the 4-to-2 phase merge unit the output behaviour of the 2-to-4 phase split unit is determined by the used implementations. Again several differing designs are possible, which are explained in detail in section 6.5 for the bundled data approach and in section 7.4 for the completion detection approach.

The nomenclature used for the signals is the same as with the 4-to-2 phase merge unit, with the only difference that input and output are interchanged, i.e. that User Data and Test Data are in this case output signals and User/Test Data the input signal.

5.6 Implementation Styles

Specific solutions are presented for the bundled data (see Section 6) and completion detection (see Section 7) protocol. The separate study of these two is reasonable, because the circuit implementations differ significantly. Nevertheless both have in common that the solutions are working in the same manner and that optimisations are possible in a similar way. For each approach different implementations are presented and shall be listed shortly here. For more details head to the above mentioned sections. Please note that this listing is by far not complete, i.e. there exist several additional implementations, however for this thesis only the simplest solutions were chosen.

Basic Implementation: A tight coupling between User Data and Test Data is introduced, forcing both 4-phase channels to be in differing phases (one in its DATA-phase and one in its NULL-phase) before the data value is propagated. In addition both 4-phase channels are requested/acknowledged at the same time, i.e. as the signal of the 2-phase one arrives. These couplings result in area efficient implementations with very little complexity. When used in the 4-to-2 phase merge unit it can be compared to the Join unit presented in [34, pp. 31-32, 58-60], with the difference that both inputs have to have diverse phases for the data to be propagated. Furthermore it has to be noted that unlike the Join unit this implementation only propagates values from one input at a time to the output and not from both. The 2-to-4 phase split unit is very much alike the Fork unit introduced in [34, pp. 31-32, 58-60], just differing by the fact that the request line of one output is negated and of course the implicit conversion from 2-phase to 4-phase.

The fact that the operation halts until both 4-phase lines have changed their state also implies, that the faster phase has to wait for the slower one. To tackle this problem some enhancements can be applied to increase the speed for certain timing constellation, presented in the following paragraphs.

Early NULL-phase: The considerations of this method are based on the fact, that the NULL-phase is a pure spacer, so no information is acquired by receiving it or required to create it. Therefore it is possible to acknowledge/start the NULL-phase much earlier than the DATA-phase.

Figure 5.7 shows the concept on the 4-to-2 phase merge unit (bundled data implementation) with its two 4-phase inputs at the top and the 2-phase output at the bottom. One can observe that the DATA-phase arrives later than the NULL-phase, i.e. $T_{DATA} > T_{NULL}$. This situation can be improved by acknowledging the NULL-phase already at time t_{req}, giving the following DATA values more time to reach the 4-to-2 phase merge unit. In bundled data systems the constellation that DATA values are slower than NULL values is very probable if asymmetric delay lines are used, i.e. ones that only delay the request signal. In all other protocols it is less probable and the differences are not that big however it can not be precluded.

In the 2-to-4 phase split unit the implementation is very similar. In detail the NULL-phase at the appropriate output gets initiated (request signal sent) at the same instant the input is acknowledged. In contrast to the 4-to-2 phase merge unit in this case the NULL values get additional time. As a consequence this implementation just yields an improvement if the DATA-phase is faster than the NULL-phase, which however is far less probable than the reverse case.

Early DATA-phase: Another possibility to increase the speed is to focus on the DATA-phase of the 4-phase inputs/outputs. Due to the fact that this phase holds all the information required it is not necessary to also wait for NULL values, because they are dropped anyway. Therefore in this method the units continue their operation as soon as new data are available.

Figure 5.8 shows the concept on the 4-to-2 phase merge unit (bundled data implementation).

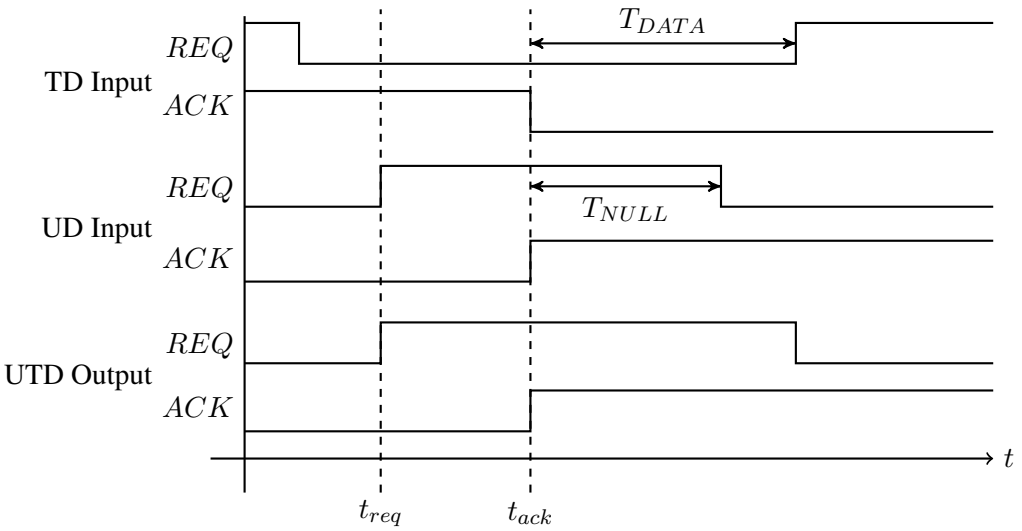


Figure 5.7: Early NULL-phase shown for the 4-to-2 phase merge unit in bundled data implementation

In this case the DATA values show up before the NULL values (T_{DATA}<T_{NULL}), however the output is not requested until both of them have shown up, unnecessarily, because the NULL values are dropped anyway. Therefore it is possible to forward the new data, i.e. to request the output, already at time t_{req}.

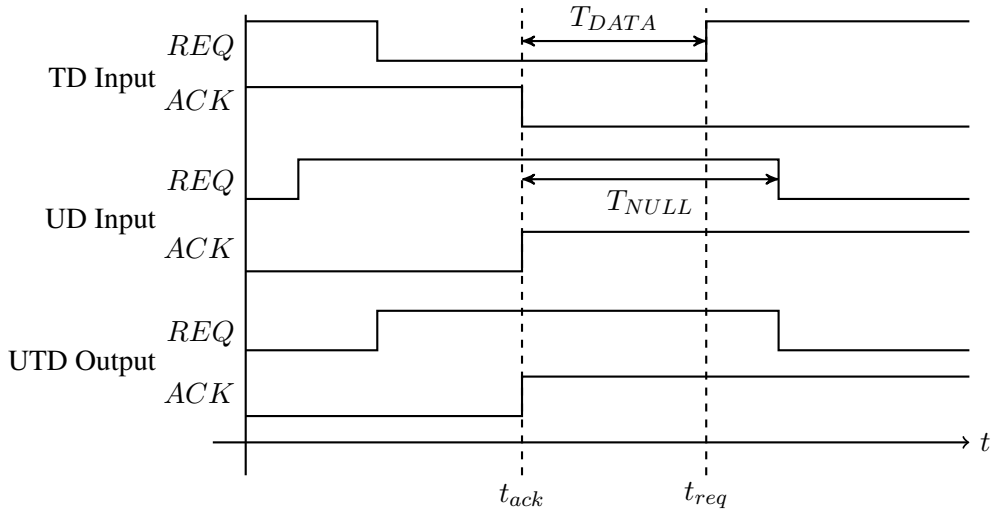


Figure 5.8: Early DATA-phase shown for the 4-to-2 phase merge unit in bundled data implementation

When used in the 2-to-4 phase split unit this design style is realised by acknowledging the

input as soon as the output that received the data has sent the acknowledge signal, even if the other one has not sent the acknowledge signal for its NULL-phase. As before this just yields an advantage if the DATA-phase is faster than the NULL-phase. Otherwise this design style falls back to the speed of the Basic Implementation.

Additional Latch: The last improvement shown in this thesis adds an additional latch to the User/Test Data input or output, making it possible to decouple in- and output even further. As the latch is applied the 4-to-2 phase merge respectively 2-to-4 phase split unit no longer is a simple switch propagating either User Data or Test Data but instead forms a complete pipeline node with internal storage. Due to the latch the speed is increased with only a moderate increase in circuit complexity, however the additional storage also can be seen as a supplementary error source resulting in an increased error probability. Nevertheless these failures can be recognised by the proposed test approach, because both User Data and Test Data pass the same latch.

5.7 Test Vector Generation and Response Analysis

The proposed test approach uses test vectors to check the correct functionality of the CUT. Due to the fact that it was designed as built-in self-testing approach the generation as well as the analysis of these has to be carried out on chip too. Several solutions have been proposed in the past fulfilling this task, which will be discussed in the following subsections.

To simplify the implementation of the transformation blocks (section 5.5) the test vector generator as well as analyser were designed as 4-phase components, i.e. they deliver and accept test vectors using the 4-phase communication protocol.

Test Vectors

The most challenging task when actually integrating the proposed test circuit is the determination of the used test vectors. These depend solely on the circuit structure and have to detect each modelled fault that might show up. In detail this means that for each fault at least one test vector has to exist that is altered by that particular fault. In that context it also does not matter if the test values are valid input vectors at all, because it might happen, for example, that the circuit is working according to its specification with the valid input values but actually has a stuck-on error that sources current constantly.

With an increasing number of input lines it will get too cumbersome to use every possible input vector as test vector. Therefore an intelligent selection of all possible vectors has to be carried out to reduce the test effort however without reducing the error coverage. For that purpose automated test pattern generators (ATPG) have been proposed (e.g. in [33, 42]) that analyse a circuit and generate a fitting test set.

Test Vector Generation

As soon as an appropriate test set has been found, it somehow has to be provided to the test input. The easiest way is of course to store the values in a (read only) memory and apply one vector after the other to the input. This however requires lots of memory on the chip, which is very expensive and therefore only suitable for a small amount of test vectors. In general a small circuit that generates one test vector after the other is a better solution. A simple example is a counter, starting at the lowest value and counting to the highest one. With increasing data width this will however take more and more time to finish and is therefore not preferable. An alternative is a linear feedback shift register (LFSR) as used in [23]. The challenge is to find a fitting LFSR that generates only the prior selected vectors and maybe in addition even in a specific order. In the past several approaches have been proposed to synthesise a LFSR from a given test set [40, 51]. Also cellular automata, as shown in [27] may be used for this purpose.

Test Response Analysis

The analysis of the test vectors has the task to determine if the received result is according to the expected output. The straight forward method is again to save every input with its corresponding output value in a storage element and compare the result as soon as it is available. Unfortunately this method requires lots of memory and is therefore expensive and not applicable in real world applications. For that reason so called compactors are used having the purpose to generate a unique value from the received results, which then can be used to determine if an error occurred. Many different properties have to be considered as it is described in [50]. Possible implementations are feedback shift registers [23], cellular automata [27] or rotate carry adders [25] just to name a few.

CHAPTER 6

Proposed Solution - Bundled Data

In this chapter concrete implementations using the bundled data design style, which was introduced in section 3.3, are discussed. Please note that a thorough timing analysis is required if one of the proposed designs is actually implemented, to assure that the timing requirements described in the text are fulfilled.

6.1 4-to-2 Phase Merge

A general description of this building block was already given in section 5.5. At this point more specific details about this unit are presented, when it is implemented using the bundled data design style.

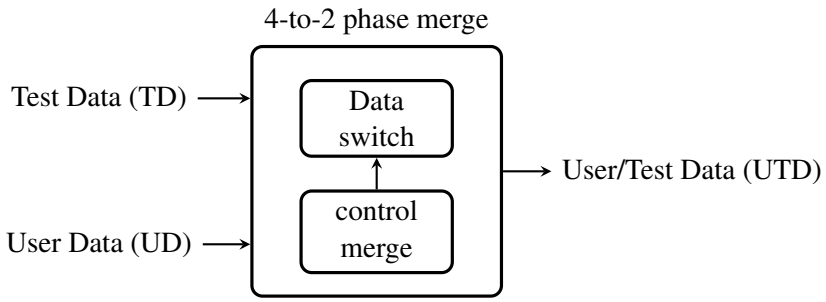


Figure 6.1: 4-to-2 Phase Merge - bundled data

In the bundled data communication style the handshake protocol is carried out using two control lines. The request line is used to indicate new data and the acknowledge line to tell the sender, that the receiver has processed them. The data themselves do not contain any control information at all making it possible to clearly separate data and control signal processing into two different blocks, more specifically the data switch block and the control merge block. Figure

6.1 shows these blocks as well as their interconnections. The data switch block is, as described before, responsible for handling the data lines, the control merge block coordinates the control signals. The arrow between these two blocks shows the fact, that the control merge unit tells the data switch block how to handle the incoming data. It further has the duty to assure, that the temporal order of control signals at the output is according to the 2-phase protocol.

In contrast to the data switch unit, which has a single implementation, many different ones exist for the control merge block. A few chosen designs are presented in section 6.2.

Data Switch

The data switch block has the purpose to handle the data lines (see also section 6.1). More specifically it works, as its name indicates, as a switch either forwarding the Test or the User Data to the output, steered by the control merge block. If the bundled data style is used, just a simple MUX unit (as shown in figure 6.2) is necessary. The control signal of the MUX is connected to the request signal line of the output (rUT). In that case the User Data, connected to the input of the MUX labelled with 1, is forwarded when the output request line is high, i.e. in the phase that was assigned to the User Data in section 5.5. If the control signal however is low, i.e. the output is in phase 0, then the Test Data input is propagated.

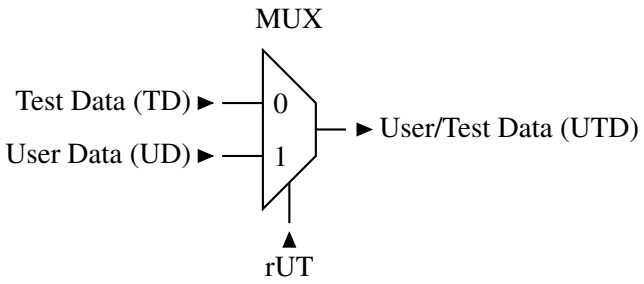


Figure 6.2: Data switch - bundled data

Due to the additional delay introduced by this MUX it is possible that the request signal arrives at the next stage before the data do. This has to be checked at design time and compensated by additional delay elements at the rUT signal line if necessary. It is important, that only the branch connected to the next stage is delayed and not the one controlling the MUX, because otherwise the difference between the arrival of the data at the next stage and the instant the request signal shows up stays the same.

For some control merge implementations it is necessary to upgrade this block by additional latches, converting it from a simple switch to a complete control node. These concepts, however, will be introduced as soon as they are used.

Control Merge

The control merge block has the task to coordinate the control signals of input and output as well as to control the data switch block, i.e. which data input has to be forwarded onto the data lines. Due to the fact that the control signal of the MUX is directly connected to the rUT signal (see section 6.1), the data switch control part becomes trivial.

The control merge block has to handle three input signals (rU,rT,aUT) as well as three output signals (aU,aT,rUT) [1]. The internal connections between them , i.e. the way the control merge block is built, determines the speed and the degree of concurrency of the 4-to-2 phase merge unit. This will be the topic of the subsequent section.

[1] For the signal naming convention refer to section 3.7.

6.2 Implementations of 4-to-2 Phase Merge

Basic Implementation

The STG implementing the functionality described in section 5.6 is shown in figure 6.3. It can be seen very easily that User Data and Test Data get acknowledged, using the signals aU and aT, not before the User/Test Data at the output were acknowledged (signal aUT) and furthermore that new data are only forwarded to the output, recognisable by a change of signal rUT, after both inputs have indicated a new phase, by changing the value on their request line. Please note that here, and also in the following, the terms input and output only describe the combination of the data, acknowledge and request lines, where the direction is determined by the data flow, as already mentioned in section 3.9. Therefore the acknowledge line of an input signal provides and the one of an output signal accepts information.

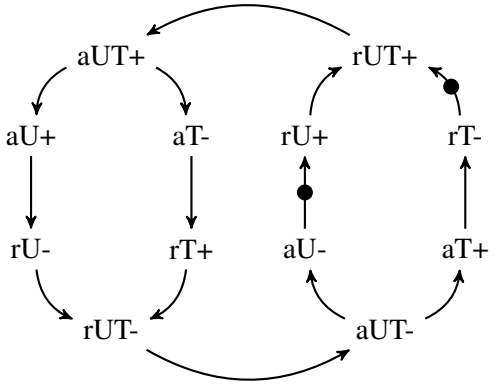


Figure 6.3: STG, Basic Implementation, bundled data, 4-to-2 phase merge

The circuit implementation of the STG is shown in figure 6.4. The tight coupling of the input signals made the circuit very small as it is the goal of this thesis. The drawback is the strict coupling of Test Data and User Data which compromises performance. Therefore some extensions to that design are presented in the following.

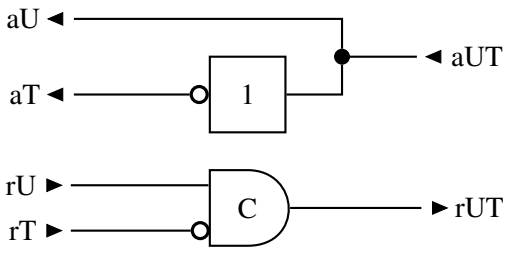


Figure 6.4: Circuit, Basic Implementation, bundled data, 4-to-2 phase merge

Early NULL-phase

The Early NULL-phase functionality is achieved by the STG shown in figure 6.5. The only difference compared to the Basic Implementation is, that the input currently in its NULL-phase (either TD or UD) gets acknowledged at the same time the data of the other input is forwarded to the output i.e. the circuit does not wait until the acknowledge signal of the receiver shows up. One can also say that the NULL-phase is acknowledged as soon as the request signal at the output changes. Starting from the STG used in the Basic Implementation this results in an additional edge from $rUT+$ to $aT-$ replacing the one coming from $aUT+$ and a newly introduced edge from $rUT-$ to $aU-$ is replacing the one coming from $aUT-$.

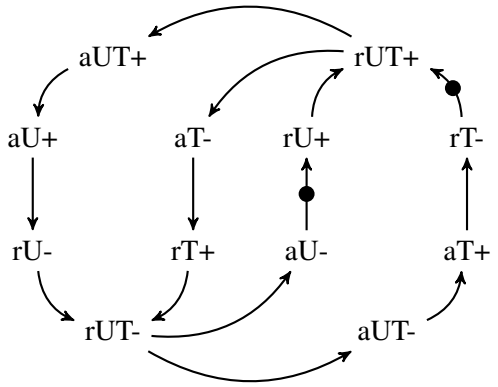


Figure 6.5: STG, Early NULL-phase, bundled data, 4-to-2 phase merge

The resulting circuit differs from the Basic Implementation only by an AND and an OR gate (see figure 6.6), which are used to early acknowledge the NULL-phase. If the line rUT is set to a high value the NOR gate gets low automatically acknowledging the NULL-phase on the Test Data input. Similarly the AND gate forwards a low value as soon as the output request signal rUT is set to a low value, acknowledging the NULL-phase on the User Data input.

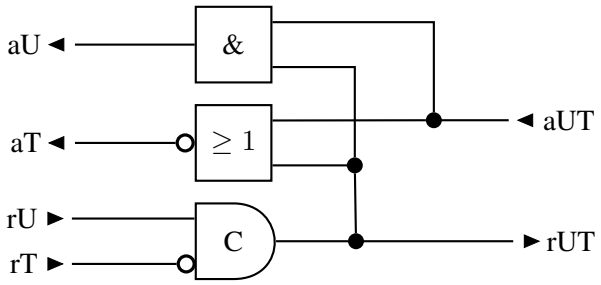


Figure 6.6: Circuit, Early NULL-phase, bundled data, 4-to-2 phase merge

Early DATA-phase

The Early DATA-phase functionality is realised by the STG shown in figure 6.7, which differs significantly from the ones presented in the previous subsections. Important to see is, that the signal transition $rUT+$ depends solely on $rU+$ and $rUT-$ solely on $rT+$, i.e. that the status of the input which has to contribute the NULL values has no effect at all. However Test Data and User Data still get acknowledged at the same time, implying that a speed improvement is only achieved if the DATA-phase is faster than the NULL-phase. Since this is not a common phenomenon this implementation may not come in hand in an actual circuit.

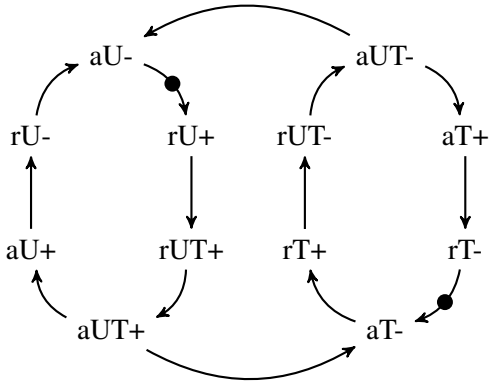


Figure 6.7: STG, Early DATA-phase, bundled data, 4-to-2 phase merge

The increased decoupling of the inputs also yields a big increase in complexity. For that reason the resulting circuit is not drawn here, however the corresponding *Petrify* output is shown in listing B.2.

Additional Latch

For this implementation style an additional latch has to be integrated, in detail right after the data switch block, as it can be seen in figure 6.8. Of course for each single data line one latch has to be installed.

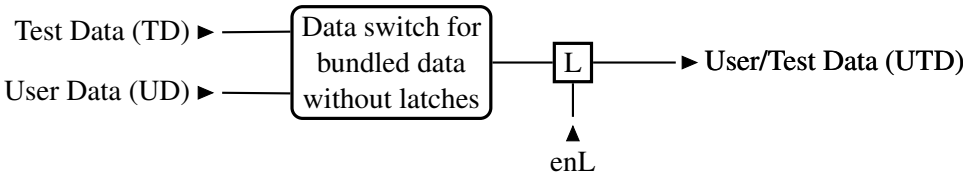


Figure 6.8: Data switch with latch at output, bundled data, 4-to-2 phase merge

In the STG (figure 6.9) one can see very clearly, that the User Data and Test Data inputs as well as the User/Test Data output are working concurrently. As soon as both inputs have changed their phase and the acknowledge signal at the output was received, the DATA values of the input

currently in the DATA-phase are stored in the latches, the inputs are acknowledged and a request is initiated at the output. After that the circuit waits for both inputs to change their phase and the output to acknowledge the values to start over again. Please note that the latch control, i.e. the circuit telling the latch if it has to store the actual value or to be transparent, is developed separately and was therefore neglected in the STG. More details concerning that topic follow in the next paragraphs.

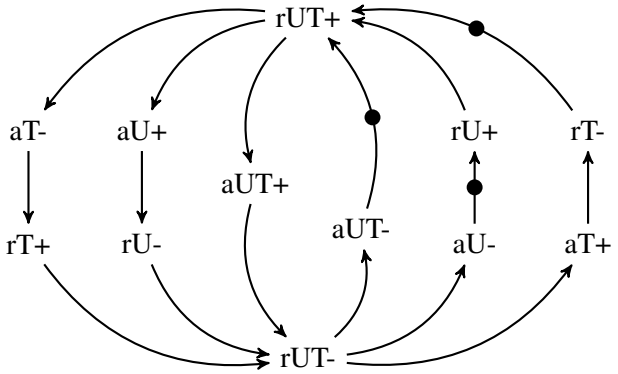


Figure 6.9: STG, Additional Latch, bundled data, 4-to-2 phase merge

Despite the fact that the implementation is working highly concurrent the circuit stays at low complexity, as can be seen in the circuit implementation shown in figure 6.10. The two Muller-C gates are used to assure, that both input request lines as well as the output acknowledge line have changed their value before a new request is initiated. In addition the input acknowledge lines are directly connected to the output request line, automatically acknowledging both inputs when a new value is indicated at the output.

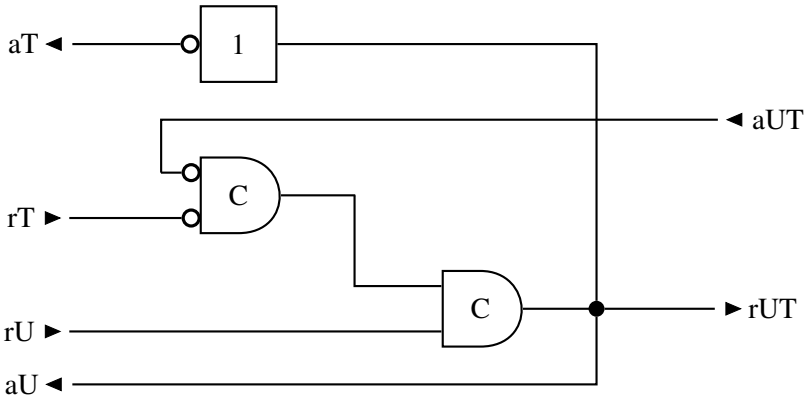


Figure 6.10: Circuit, Additional Latch, bundled data, 4-to-2 phase merge

In addition a circuit, that controls the latches, i.e. tells them when to store the actual value and when to get transparent, has to be introduced. More specific they have to be closed as soon as the request signal is raised at the output and can be opened again when the output acknowledge

signal arrives. This means that the latches have to store their values when the signals rUT and aUT are different and have to be transparent when both hold the same value. For that reason the enable signal enL in figure 6.8 can be defined as

$$enL = (rUT \equiv aUT)$$

If a common latch is used the designer has to assure that the data are stored before the new ones arrive. This depends on the speed of the equality gate, the latch itself and the connections between them. As an alternative it is possible to replace the latch by a capture-pass event-controlled storage element as described in [35] and [34, pp. 9-28]. In that case the rUT signal line has to be connected to the capture input and the aUT line to the pass input. Connected in that way the functionality is equal to the above described latch-based version however the effort for the timing analysis is reduced.

As already pointed out several times, timing analyses are mandatory if the bundled data design style is used. With this implementation the designer has to additionally take care that the latches are working correctly, i.e. that no setup and hold time violations occur. Consider the following case: The inputs have already changed their phases, just the acknowledge signal at the output is still missing, causing the latches to still hold their actual value. As soon as the acknowledge signal arrives the latches get transparent, however at the same time the output request signal is set again causing them to return to the storing state right away. Depending on the timing the latches may not get transparent at all or not long enough to store the new data correctly. Therefore delay elements may be necessary to guarantee correct behaviour of the circuit.

Furthermore the data propagation within the unit has to be investigated very carefully. The fact that the output request line also controls the MUX implies that it will take some time until the data arrive at the latches. If the control signals of the latches are too fast they may be already storing their actual values by the time the data arrive.

It is also possible to merge the control circuits for latch and control lines. Listing B.3 shows the *Petrify* output for an STG closing the latch right after rUT was changed and opening it up again after the acknowledge signal of the output was arrived.

6.3 Enhancements

The implementations presented above were designed under the perspective of simplicity. Of course there exist a lot more different designs, each with its individual properties and advantages. Nevertheless the development of an optimal controller was not the goal of this thesis and was therefore intentionally left open for future research.

In the following some further improvements and improved designs are presented to show, that more complex solutions are possible, but they neither have been further investigated nor optimised. It is probable that much more compact and simple solutions exist. Please note that these enhancements are just presented once with the 4-to-2 phase merge unit using the bundled data design style but are in general applicable to all other units and styles as well, of course with appropriate modifications.

Latches at Input

One example of such an enhancement is to add latches in front of the data switch block as it can be seen in figure 6.11. Please note that the signal coordinating the latches, named *enL* in the figure, is also used to control the MUX, though in negated form. This means that the latch at a particular input holds its value if the data of that input are propagated to the output, and is transparent otherwise.

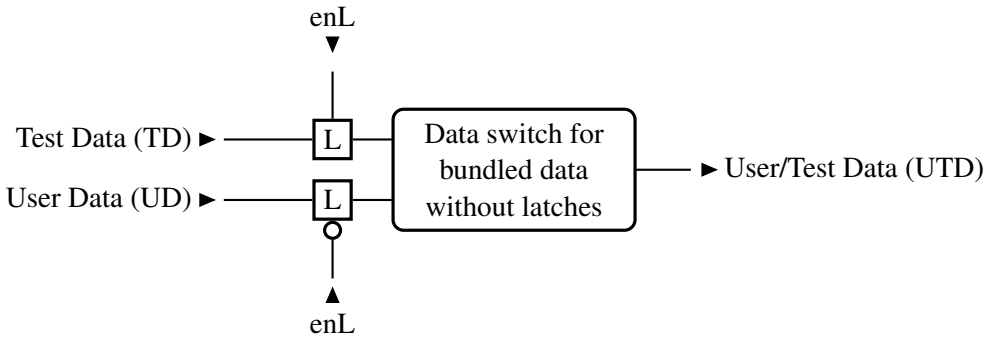


Figure 6.11: Data switch with latches - bundled data

The benefit of this design is, that the input data can be acknowledged before the acknowledge signal at the output has been received. An STG using that design style can be seen in figure 6.12, the resulting netlist generated by *Petrify* in listing B.4. The additional latches however represent a supplementary fault source, which are not testable using the presented test procedure. To cover these an additional one has to be introduced.

Decoupled Controller

The STG of a completely decoupled controller design is shown in figure 6.13. There, two nearly independent loops can be identified, one handling the User Data input (left side of the STG) and

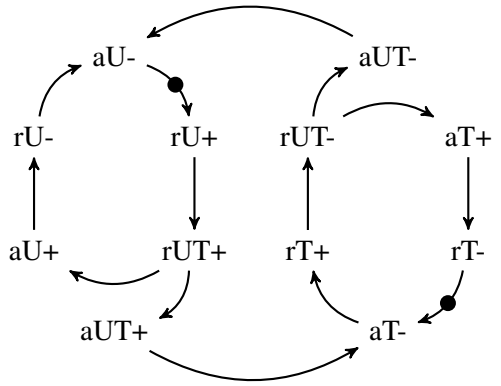


Figure 6.12: STG, Latches at Input, bundled data, 4-to-2 phase merge

one the Test Data input (right side of the STG). These are only connected by the control signals of the output, that assure alternating sending of User Data and Test Data. All other operations like requesting and acknowledging the NULL-phase are carried out completely independently. In contrast to other presented implementations the speed of the circuit depends on the overall time needed to complete both the NULL-phase and the DATA-phase, whereat the individual length of each phase has no effect, as it was the case with previous designs. This implementation is achieved when trying to combine the Early NULL-phase and Early DATA-phase approach.

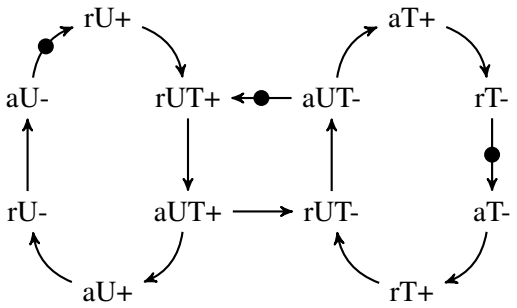


Figure 6.13: STG, Decoupled Controller, bundled data, 4-to-2 phase merge

Unfortunately the high concurrency leads to a very complex implementation, which can be seen in the *Petrify* output in listing B.5.

Decoupled Controller with latches at input

At last a combination of latches at the input together with a decoupled controller is presented. In this implementation the inputs are acknowledged as soon as the data were forwarded to the output and therefore stored in the latches, which further increases the speed. The corresponding STG is shown in figure 6.14.

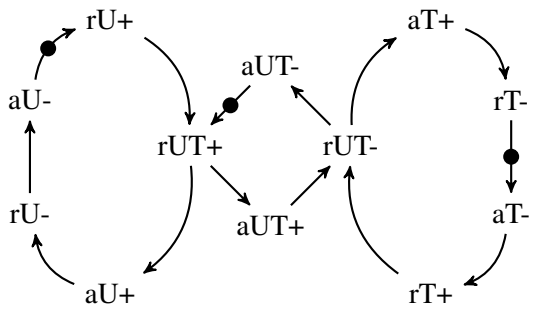


Figure 6.14: STG, Decoupled Controller with latches, bundled data, 4-to-2 phase merge

By introducing latches the gate count of the control circuit is slightly reduced compared to the decoupled controller without latches (see listing B.6), however the overall count is increased. Furthermore these latches represent not only additional error sources, but also require a supplementary error detection method, as already described earlier.

6.4 2-to-4 Phase Split

The schematics of the 2-to-4 phase split unit (figure 6.15) shows two blocks, the data fork one, responsible for handling the data lines, and the control split one, used to coordinate the control lines. One difference compared to the 4-to-2 phase merge unit however is the missing connection from the control split unit to the data fork block, which will be explained in the next subsection.

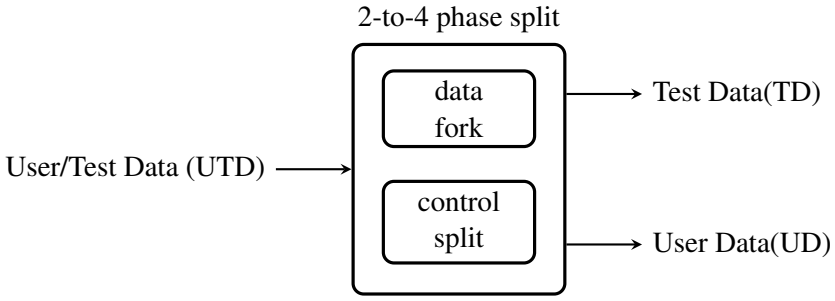


Figure 6.15: 2-to-4 phase split - bundled data

Data Fork

The data fork unit is used for data handling, more specific to forward the input data to the correct output. In this case, i.e. by using the bundled data style, it is possible to implement this building block as a simple fork, using the control lines to indicate which unit is supposed to read the data. This is only possible because a device using the 4-phase protocol does not read or write the data lines in its NULL-phase, making it possible to put arbitrary data there. Of course one could also integrate a DEMUX as counterpart to the MUX used in the 4-to-2 phase merge block but in this case it is not necessary.

If an increased level of concurrency is required, it is possible, as before, to add latches either on the in- or the output. These concepts however will be described in detail as soon as they are used.

Control Split

The control split block serves the purpose to coordinate the control signals in a way, that on both outputs a correct 4-phase protocol is delivered. As mentioned before the data path is a simple fork, implying that the whole complexity lies within the control split block. Furthermore the timing of the signal is important too, since both outputs get the same data and have to know when they have to read the lines. As before this block can be implemented in various shapes, differing in speed, size and level of concurrency. Concrete implementations will be discussed in the following section.

6.5 Implementations of 2-to-4 Phase Split

Due to the characteristics of the bundled data approach the implementations of the 2-to-4 phase split unit are sometimes just the opposite of the 4-to-2 phase merge implementations described in section 6.2. Therefore the circuits presented here differ only slightly from their counterparts shown earlier. Nevertheless there are some functional differences, that are pointed out in the description of the designs, that have to be considered.

Basic Implementation

The Basic Implementation of the 2-to-4 phase split unit is nearly the same as the one for the 4-to-2 phase merge, which can be seen best in the STG (figure 6.16). As soon as a request is received at the input it is forwarded to the User Data output and in its negated form to the Test Data one. After the acknowledge signal has been received on both outputs the input acknowledge signal is sent.

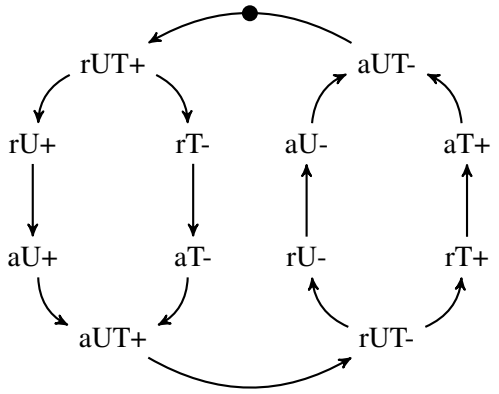


Figure 6.16: STG, Basic Implementation, bundled data, 2-to-4 phase split

This tight coupling yields a very small and simple circuit which is shown in figure 6.17. It can be seen very clearly that the request signal at the input is forwarded without delay to both outputs, whereas the acknowledge signal is only propagated if both, the User Data and the Test Data output, have acknowledged their current phases.

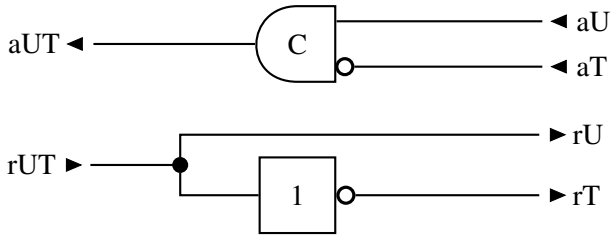


Figure 6.17: Circuit, Basic Implementation, bundled data, 2-to-4 phase split

Early NULL-phase

The STG for this implementation can be seen in figure 6.18. It is very similar to the Basic Implementation, just with an additional edge from $aUT-$ to $rT-$ replacing the one coming from $rUT+$ and an additional edge between $aUT+$ and $rU-$ replacing the one coming from $rUT-$. The first change has the effect that the Test Data NULL-phase is requested right after the input was acknowledged instead of waiting until a new request was received at the input. The second change introduces the same effect at the User Data output.

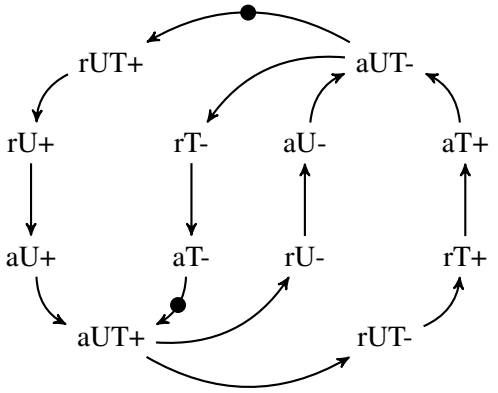


Figure 6.18: STG, Early NULL-phase, bundled data, 2-to-4 phase split

The resulting circuit implementation can be seen in figure 6.19. Compared to the Basic Implementation circuit only two AND gates are added, which are responsible to generate the early NULL-phase requests. If aUT rises from low to high the AND gate at the TD output (signal rT) gets ready to propagate the next incoming request to the consecutive stage. At the same time the AND gate at the UD output (signal rU) forwards a low value, automatically starting the NULL-phase at that output. If aUT eventually drops back to a low value the same procedure occurs, with the difference that rU and rT are interchanged.

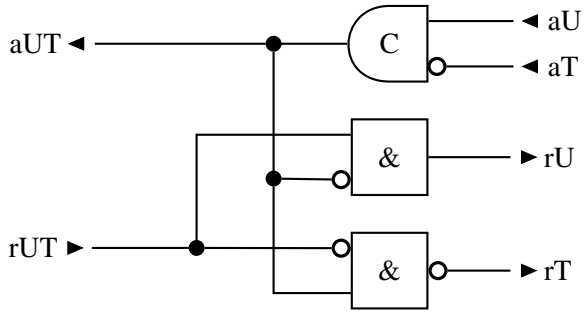


Figure 6.19: Circuit, Early NULL-phase, bundled data, 2-to-4 phase split

Early DATA-phase

The corresponding STG describing this implementation using the bundled data style can be seen in figure 6.20. It differs significantly from the ones presented in the previous implementations, the main difference however is, that $aUT+$ solely depends on $aU+$ and $aUT-$ solely on $aT+$. This assures that the acknowledge signal at the input is sent right after the acknowledge signal was received at the output that received the most recent data.

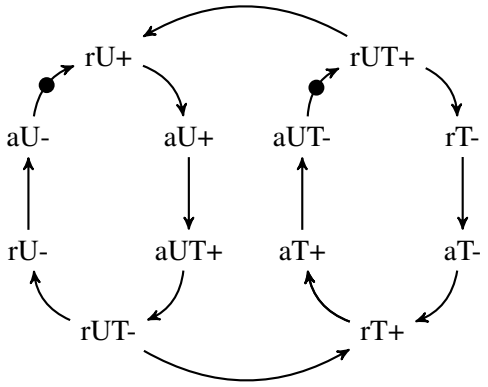


Figure 6.20: STG, Early DATA-phase, bundled data, 2-to-4 phase split

This implementation, as well as the one before, are yielding an improvement if the NULL-phase is slower than the DATA-phase, however for the opposite case no simple implementation was found. Just assume for a moment that such a method would exist. To improve the speed under the specified properties the DATA values would have to be forwarded to the appropriate output before the NULL values are propagated to the other one, giving the DATA values more time to travel to the following stage and to be acknowledged than the NULL values. However the earliest possible moment the NULL-phase may be started is the moment the acknowledge signal is sent to the User/Test Data input, whereas the first possible point in time the DATA-phase can be launched is the moment the request signal of the User/Test Data input arrives. Since the first always occurs earlier in time than the second one that kind of method is not realisable.

That is the reason why two methods speeding up the circumstance, that is less probable to appear, exist and none optimising the more probable one. When comparing the circuit implementation of these methods (listing B.7) one can see that the Early NULL-phase method is superior to the Early DATA-phase one concerning area by far. In addition differing working speeds are imaginable depending on the run time deviations between the two phases.

Additional Latch

For this implementation an additional latch has to be introduced into the data fork unit, as it is shown in figure 6.21. It is installed on the input path making it possible to acknowledge the input value right away. Please note that for each single data line a latch has to be installed.

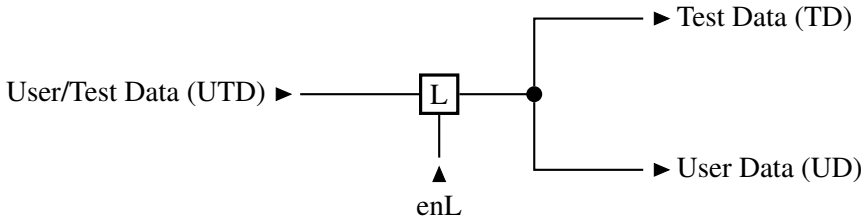


Figure 6.21: data fork, Additional Latch, bundled data, 2-to-4 phase split

The corresponding STG is shown in figure 6.22. It can be seen that in- and output signals are completely decoupled. More specific the input gets acknowledged as soon as both outputs have acknowledged their current phase. This further implies that the slowest phase determines the working speed of the circuit.

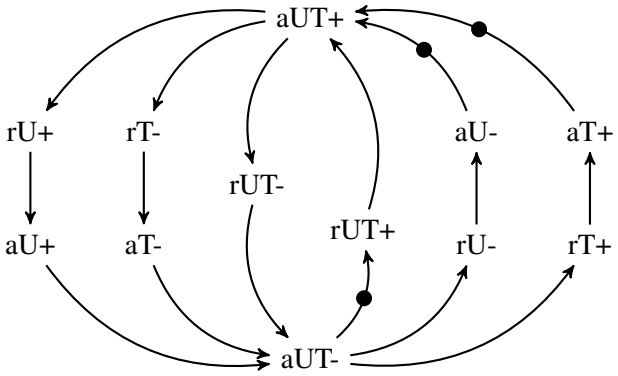


Figure 6.22: STG, Additional Latch, bundled data, 2-to-4 phase split

An additional challenge in this implementation is the latch control. As soon as a new data arrive at the input and the circuit is ready to propagate it, the latches have to store them until the corresponding output (either TD or UD) sends the acknowledge signal, which causes the latches to get transparent again, ready to store the next input data. This behaviour is achieved when defining the signal enL controlling the latches as

$$enL = (aU \wedge aUT) \vee (aT \wedge \neg aUT)$$

Due to the definition in section 5.1 the latches get transparent if the enable signal is high, i.e. if the data delivered by the input got acknowledged by the appropriate output and stores its actual value, if the enable signal is low.

As always when using a bundled data approach a timing analysis is important to prove the correct behaviour of the circuit, whereas the propagation delay of the signal controlling the latch has to be investigated extremely thoroughly. Due to the fact that enL is calculated from the acknowledge value aUT it is possible that new data arrive before the old one have been stored in the latch. As an alternative again the latch could be replaced by a capture-pass event-controlled storage element as described in [35] and [34, pp. 9-28]. The capture input has to be connected to the aUT signal and the pass to a Muller-C gate having aU and $\neg aT$ as inputs.

Due to the fact that still a very tight coupling between the two phases is used, the circuit is very simple, as it can be seen in figure 6.23. The Muller-C gates again assure that a new value is ready at the input and that both outputs have changed their phases before acknowledging the next input value and at the same time requesting the outputs. The improved decoupling of input and output is paid with an additional latch, not only introducing a supplementary error source but also transforming the 2-to-4 phase split unit from a simple switch to a node with internal storage.

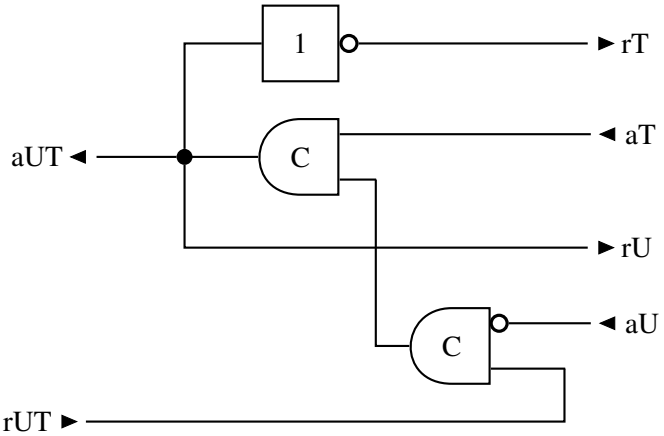


Figure 6.23: Circuit, Additional Latch, bundled data, 2-to-4 phase split

Please note that it is possible to integrate the logic calculating the latch control signal enL into the circuit handling the request and acknowledge signals. For that purpose the STG shown above has to be extended such that the latch is ordered to store its actual value right after the input was acknowledged and to get transparent again after the DATA-phase got acknowledged. The *Petrify* output of an STG altered in that way is shown in listing B.8.

CHAPTER 7

Proposed Solution - Completion Detection

In this chapter the proposed approach is implemented using the completion detection communication method, which was already described in section 3.4. At first a closer look is taken at the 4-to-2 phase merge unit and afterwards at the 2-to-4 phase split one. Again different implementations, each with its individual advantages and disadvantages, are shown.

When using the completion detection style timing constraints (see section 3.5) have to be considered to determine under which restrictions the circuit is working correctly. These constraints of course have to be checked in a timing analysis at design time, which is the reason why it is tried in this thesis to achieve DI implementations wherever possible. Unfortunately this is very hard so most of the circuits represented here are only QDI. Nevertheless this already simplifies integration a lot due to its rather little restrictions.

7.1 4-to-2 Phase Merge

At the beginning the block combining Test Data and User Data is discussed, i.e. the 4-to-2 phase merge unit. In contrast to the bundled data approach this unit does not have a fixed size but instead consists of two basic components, namely the data format conversion and control merge unit, which have to be extended by additional units if an improved level of concurrency has to be achieved. The first one gets more complex compared to the bundled data implementation, due to the fact that the data format has to be changed between 4-phase and 2-phase and because the data implicitly hold the information when they are ready to be processed.

The basic components (solid lines) as well as some possible additional ones (dashed and dotted lines) can be seen in figure 7.1. The arrows in the upper row represent the data lines whereas

the ones to and from the control merge block represent control information (e.g. acknowledge signals, phase information, steering signals).

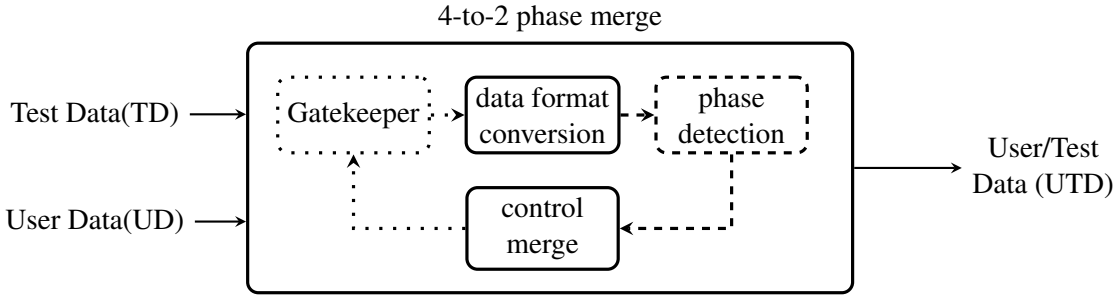


Figure 7.1: 4-to-2 Phase Merge unit using completion detection communication style, basic components in solid lines, additional components in dashed and dotted lines

Data Format Conversion

As mentioned before, the amount of units processing the data signals is variable and scales with the desired concurrency and speed. Although the approaches differ significantly all of them have to have a data format conversion block that transforms the data from NCL to LEDR encoding, which is necessary due to the differing rail usage in both schemes (see section 3.4).

An implementation of this data format conversion unit, which has to be introduced on each single input line separately, can be seen in figure 7.2. On the right side the 2-phase User/Test Data output is shown, where the value rail (VR) holds the actual value and the phase rail (PR) is responsible to generate the correct phase. On the left side the User Data (UD,bottom) and Test Data (TD,top) input are shown each with its high (HR) and low (LR) rail. Due to the limitations of the 4-phase protocol NCL only one rail of an input can be set to a high value at a time (compare section 3.4). Therefore exactly three of the four OR gates propagate a high value if both inputs are in their DATA-phase, two if only one input is in its DATA-phase and none if both are in their NULL-phase.

The unique property of this circuit is, that it just changes the value at its output when exactly two of the four OR-gates propagate a high value, i.e. when one input is in its DATA-phase and the other one in its NULL-phase. This is the case because both inputs of a Muller-C gate have to have the same value to set the output, implying that if both OR-gates at the input of one Muller-C element forward a high value the inputs differ and the current value is stored. Only after one of them switches to low the Muller-C gate opens for the new data. This implies a very tight coupling between the inputs because changes are only propagated if both inputs are in different phases (NULL-phase and DATA-phase).

Unfortunately this circuit is not DI by design. The reason is that when both inputs are in their DATA-phase a single OR gate receives a high value on both its inputs. This violates

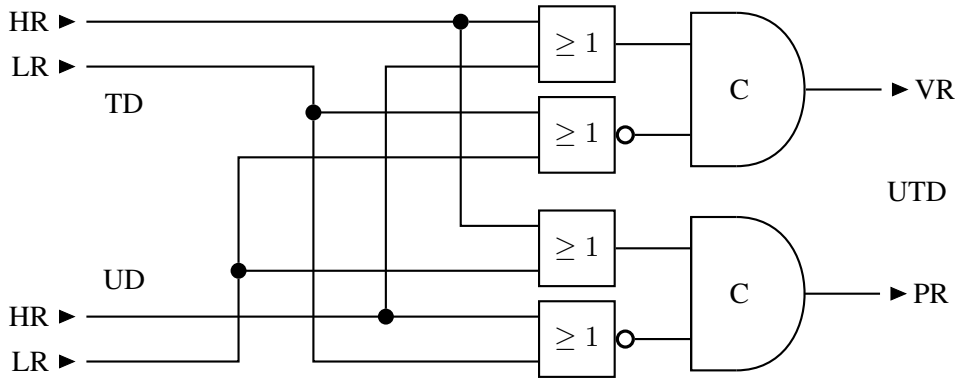


Figure 7.2: Data Format Conversion unit, conversion from NCL to LEDR

the indication principle because it can not be assured that the value on both input lines has reached the gate when it starts forwarding a high value. Based on this considerations it is easy to construct a situation where the actions of the unit deviate from the desired behaviour. The reader is encouraged to sketch shortly such a setting.

Control Merge

The control merge block is again responsible for managing the control signals, however it is a lot smaller compared to the bundled data approach because only acknowledge lines have to be coordinated. The request lines, responsible to indicate that new data are available, were eliminated, because the data themselves determine when they are complete and ready to be processed. However it has to be noted that the complexity of this block starts to rise when optional blocks are added, as it is shown later in this chapter.

Phase Detection

This building block is optional and is used to determine the actual phase the data are in. In detail its output is set to a high value if all inputs are in phase 1, i.e. if they all have an odd parity and to a low value if all are in phase 0, i.e. having an even parity. When single inputs are in differing phases the unit keeps its current output until one of the two above described configurations appears again.

Gatekeeper

The Gatekeeper block is used to block inputs from propagating into the unit or to the outside and is an optional building block. Connected to this unit are input lines and the same amount of output ones as well as a control line. If the latter one holds a high value the data are propagated from the input to the output and if it holds a low value, low values are put on all output lines. It is possible to implement these units by simple AND gates having the control signal and an input line as inputs.

7.2 Implementations of 4-to-2 Phase Merge

In this section concrete implementations for the 4-to-2 phase merge block are shown, following the listing presented in section 5.6, starting with the simplest one. Afterwards the more concurrent versions are presented and discussed in detail.

Basic Implementation

As mentioned in section 5.6 in this implementation style the DATA values of one input are only propagated if the other input is in its NULL-phase. Due to the fact that this is exactly the behaviour of the data format conversion unit introduced in the last section (figure 7.2) nothing more is necessary to process the data lines.

Since the data propagation is coordinated by the data format conversion block the only thing left to do for the control merge block is to coordinate the acknowledge signals, which is a fairly easy task, as the STG shows (see figure 7.3). As soon as the output acknowledges the data the signal is further propagated to the inputs.

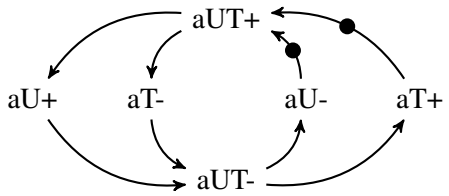


Figure 7.3: STG, Basic Implementation, 4-to-2 phase merge, completion detection

This direct propagation yields a very simple implementation in the form of a direct connection and one inverter as it can be seen in figure 7.4. Due to the fact that only one inverter was used the control merge implementation is DI [34, pp. 9-28]. Because of the timing restrictions of the data format conversion block, the unit as a whole however is again just QDI.

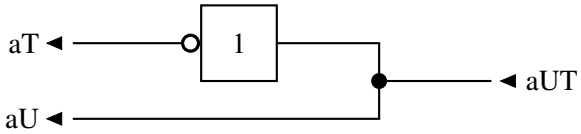


Figure 7.4: Circuit, Basic Implementation, 4-to-2 phase merge, completion detection

Early NULL-phase

To implement the Early NULL-phase behaviour an additional unit is necessary, namely a phase detection (see section 7.1), which is installed after the data format conversion block. The data processing part of this 4-to-2 phase merge implementation therefore consists of the data format conversion block and the phase detection unit. A sooner acknowledgement of the input currently in its NULL-phase furthermore requires, that the output of the data format conversion logic does not change when both inputs carry DATA values because otherwise the arrival of the new data might disturb the actual transmission, which might not be finished at that time. However this property is true for the format conversion unit as already shown earlier.

The corresponding STG is shown in figure 7.5, where the characteristics can be observed very easily. As soon as new data have passed the data format conversion unit they are detected by the phase detector (signal pUT) and at the same time the NULL-phase at the appropriate input gets acknowledged. The DATA-phase on the other one however gets acknowledged not before the acknowledge signal at the output has been received.

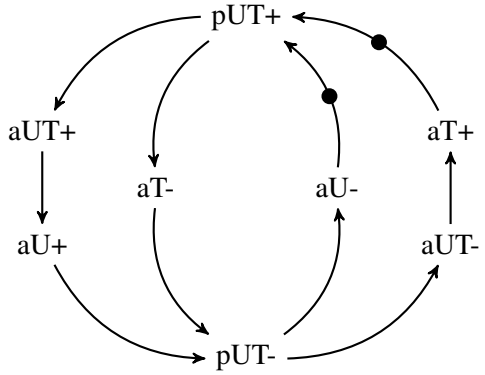


Figure 7.5: STG, Early NULL-phase, 4-to-2 phase merge, completion detection

The complexity of the control merge block increases due to the additional control signal of the phase detection unit, which can be observed in the circuit implementation shown in figure 7.6. The AND and NOR gate are used to send the acknowledge signals as soon as the phase detector indicates a phase change by changing its output value. At the beginning the aUT and pUT signals are low. As soon as pUT rises to a high value the NOR gate sets its output to low, i.e. acknowledging the NULL-phase. The AND gate does not react at all, it just switches when aUT finally gets high. When pUT returns to low the AND gate immediately issues a low value which acknowledges the NULL-phase and the NOR gate reacts after aUT got low.

Unfortunately the implementation is just QDI due to the violation of the *indication principle* explained in section 3.5. With unfortunate delays it may happen, that phases are acknowledged too early or that additional phases are inserted. Again the reader is encourage to sketch these situations.

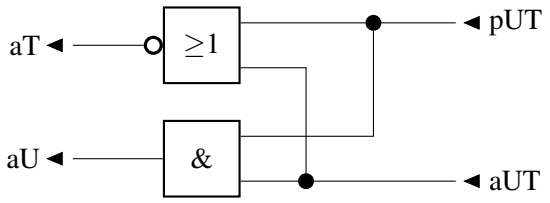


Figure 7.6: Circuit, Early NULL-phase, control merge, completion detection

Early DATA-phase

In this implementation new data have to be propagated to the output as soon as they are ready at the input, independent of the state of the other input. Unfortunately the data format conversion does only propagate data if the inputs are in differing phases, so the NULL-phase has to be imposed on the corresponding input if this style is desired. For that purpose the Gatekeeper units can be used due to the fact that the NULL-phase is defined as a low value on both rails, which is exactly the output if a Gatekeeper blocks the signals.

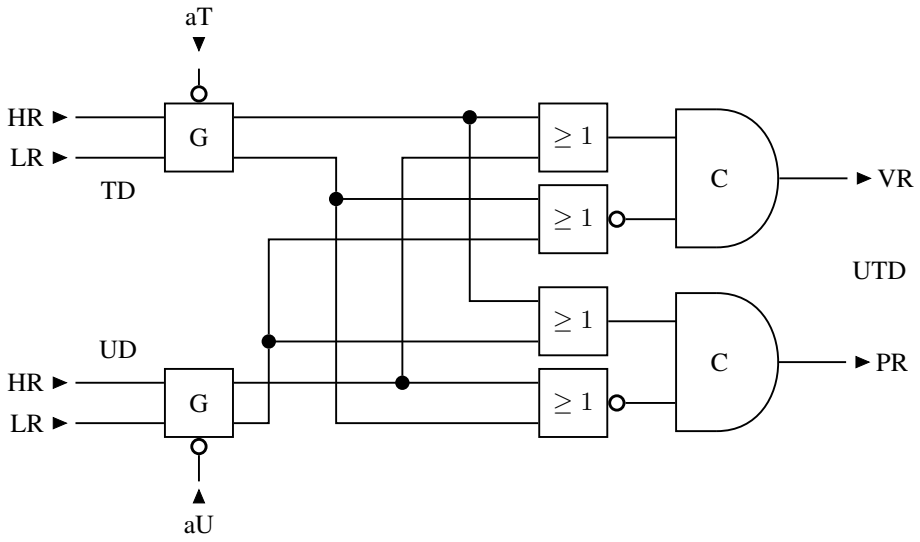


Figure 7.7: data format conversion, Early DATA-phase, completion detection

In the data format conversion block these Gatekeepers are integrated in the data path of the User Data and Test Data input as shown in figure 7.7, where they are represented by a building block containing the letter G. The control lines are drawn above (below) and are connected to the inversion of aU in the one case and to the inversion of aT in the other one. In detail the Gatekeeper blocks the input lines if aT respectively aU gets high and propagates them if aT (aU) gets low. This implies that as soon as the DATA-phase is acknowledged on one line, the data lines are blocked by the Gatekeeper giving the other input the opportunity to instantly start sending its own data. When an input is currently in its NULL-phase the Gatekeeper forwards

the input lines as soon as the NULL-phase was acknowledged. This is allowed due to the fact that the output of the format conversion unit does not change if new data arrive while other data are processed.

By adding these Gatekeepers the timing requirements have to be reconsidered. Due to the fact that they do not alter the data directly but only block them at some points in time, the timing requirements stated earlier for the data format conversion block, i.e. QDI, is also required in this implementation. More specific this implies that the acknowledge signals reach the Gatekeepers at the same time as the actual node. If the delay of the NULL-phase is low, the speed of this design approaches the one of the Basic Implementation, providing that the Gatekeeper circuit does not introduce an additional delay.

Unfortunately additional phase detection units have to be added on both input lines, used to detect new data on the line. These are mounted right before the data format conversion unit and prevent an error that may appear if the phases are acknowledged too early. Assume the case that the UD input just delivered its DATA-phaseand that the time it takes the following NULL-phase to reach the data format conversion unit is huge. Right after the output acknowledge signal was received the Gatekeeper at the UD input blocks the input lines, i.e. propagates the NULL-phase. If the TD input already got new DATA values they are propagated and eventually acknowledged. As that acknowledge signal arrives, the Gatekeeper at the UD input gets transparent, propagating the data of the previous DATA-phase because the following NULL-phase has not reached the input yet. For that reason it has to be checked if the input has changed its phase before a new acknowledge signal is sent.

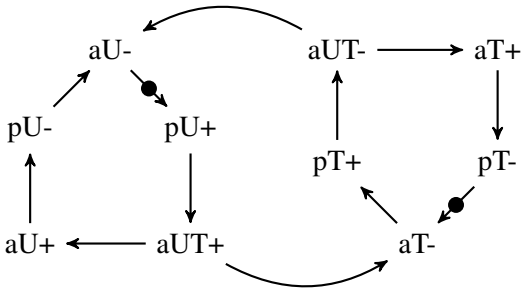


Figure 7.8: STG, Early DATA-phase, 4-to-2 phase merge, completion detection

The corresponding STG is shown in figure 7.8. The most important thing to see is, that the transition $aU-$, i.e. the acknowledgement of the NULL-phase and therefore the opening of the Gatekeeper at the User Data input, does depend on the phase detection signal $pU-$ i.e. that NULL values are present at the input. This prevents the case that the Gatekeeper opens and propagates old values to the output. The same observations can be made at the Test Data input. The *Petrify* output, showing only a slight increase in complexity, can be seen in listing B.9.

Additional Latch

As proposed in section 5.6 input and output can be further decoupled by introducing a latch at the output line. For the latch control circuit however an additional phase detection unit is necessary, which has to be integrated right after the latch, as it is shown in figure 7.9.

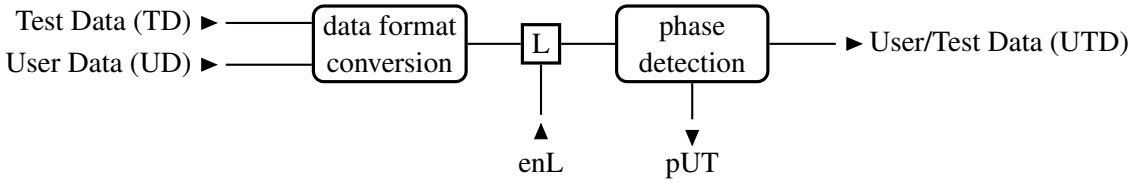


Figure 7.9: data path of Additional Latch, 2-to-4 phase split, completion detection

The phase detection unit in this implementation is used to detect new data at the output which is used to control the latch as well as the acknowledgement mechanism of the inputs, which can also be seen in the STG (figure 7.10). As soon as both inputs have changed their phases, the new DATA values are forwarded to the latch which, if transparent, propagates it to the output and also to the phase detection unit. A change on the output of the latter indicates the beginning of a new phase at the 4-to-2 phase merge output resulting in an acknowledgement of the inputs as well as the command to close the latch. More details on the circuit responsible for the latter follow in the next paragraphs.

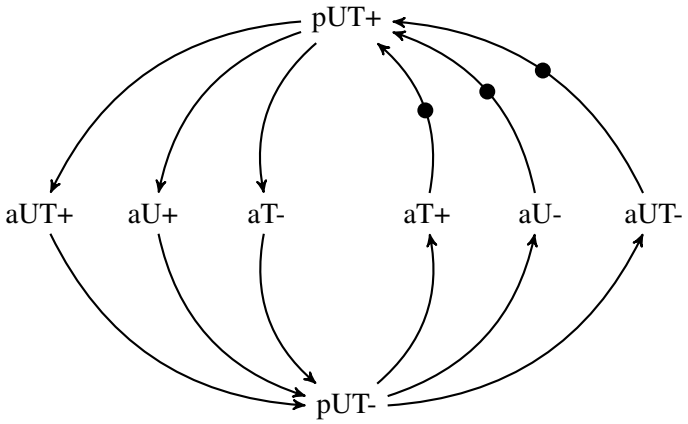


Figure 7.10: STG, Additional Latch, 4-to-2 phase merge, completion detection

Please note that the signal aUT was solely introduced in the STG to indicate that a new output value may only appear after the acknowledge signal of the old output value was received (see also the latch control). For the design of the control circuit handling the input acknowledge lines it is not needed and is therefore automatically dropped by *Petrify*.

The final circuit implementation is shown in figure 7.11. It can be seen that the acknowledge

signals are directly connected to the output of the phase detection unit, in the Test Data case with an additional negation. In the same figure the latch control implementation (output enL) is shown. This output gets high, i.e. the latch gets transparent, if the phase that is currently present at the output already got acknowledged.

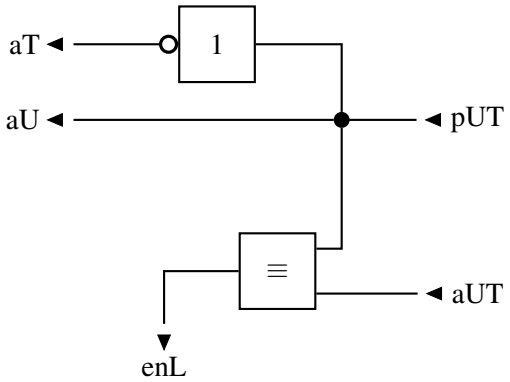


Figure 7.11: Circuit, Additional Latch, 4-to-2 phase merge, completion detection

Please note that race conditions appear if such an implementation is used. Even if QDI is assumed it only demands that the pUT signal reaches the equivalence gate and the previous pipeline stage at the same time. If the first one is very slow it may happen that the signal for the latch to close shows up too late, forwarding the new DATA values before the old had been acknowledged. A more robust implementation can be achieved by replacing the latch by a capture-pass event-controlled storage element as described in [35] and [34, pp. 9-28], where the capture input has to be connected to pUT and the pass input to aUT. In that case QDI property assures that the pUT signal reaches the storage element and the previous pipeline stage at the same time. Please note that even this does not completely guarantee correct behaviour, however the chance for an error is very small and can therefore be accepted. Nevertheless to assure correct functionality it has to be checked at design time if the capture-pass storage element stores its current value faster than it takes the DATA values to propagate from the previous pipeline stage to the storage element, which can be carried out using a thorough timing analysis.

An alternative approach is to include the latch control already in the STG and use *Petrify* to synthesise the whole unit. For test purposes this has also been carried out, however the latch control did also depend on the acknowledge signals, as it was the case with the first implementation. This means that even in this case QDI is not enough to ensure correct behaviour. For that reason the usage of an ordinary latch is only possible in combination with a timing analysis.

7.3 2-to-4 Phase Split

The 2-to-4 phase split unit divides the 2-phase input data stream into two independent 4-phase output data streams. The layout of this block (figure 7.12) is very similar to the 4-to-2 phase merge block described in section 7.1. Again the mandatory blocks, data format conversion and control split, can be extended by additional units like a phase detector or Gatekeepers, if an increased concurrency has to be achieved. As before, the control flow (arrows from and to the control split unit) does not directly reach the data format conversion block, i.e. it is not necessary to control it in any sense. The data flow in the picture is represented by the arrows from and to the data format conversion block.

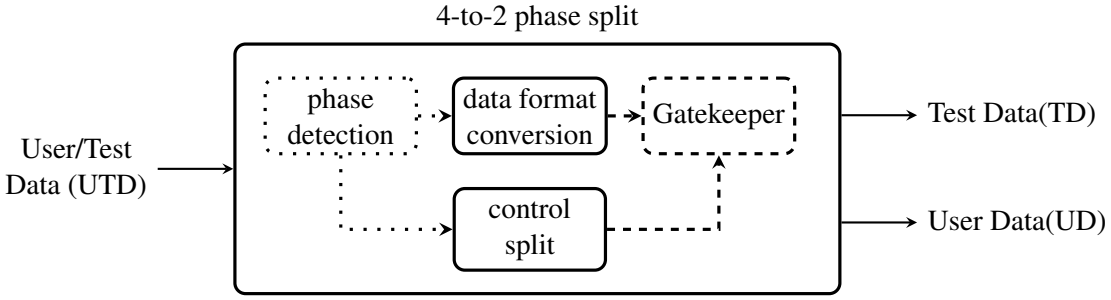


Figure 7.12: 2-to-4 Phase Split Unit

Data Format Conversion

The data format conversion block is used to convert the input data, which are delivered in the 2-phase LEDR format, to an equivalent representation in the NCL format. The general layout of this unit can be seen in figure 7.13. Two independent format conversion (FC) units, both working on the same input data, are used to realise the conversion. Each FC block thereby handles exactly one of the two phases of the input signal and blocks the other one. Furthermore it introduces an additional NULL-phase to create a valid 4-phase signal at the output.

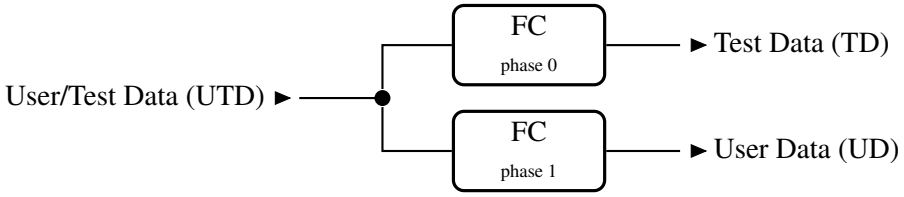


Figure 7.13: Format Conversion from 2 to 4 phase

Please note that this approach is only possible because User Data and Test Data are always assigned to the same phase (as described in section 5.5). According to the definitions stated there the block handling phase 1 is connected to the UD output and the one handling phase 0 to

the TD output. If the reverse mapping is required i.e. phase 1 to TD and phase 0 to UD also the FC blocks have to be interchanged.

The circuit implementation for the FC block handling phase 1, i.e. odd parity, is shown in figure 7.14. In the picture the input rails are named value rail, holding the actual value (high or low), and phase rail, which is responsible to generate the correct parity for the actual phase.

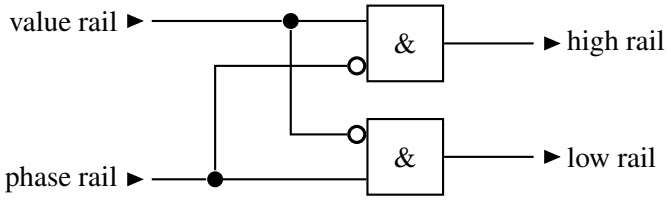


Figure 7.14: Circuit, Format Conversion, Phase 1, 2-to-4 phase split, completion detection

If the input rails of the FC phase 1 block have an odd parity, which implies that exactly one of the input lines is high and the other one low, the input value is transformed into the corresponding NCL value, otherwise the NULL value is sent, because in that case both AND gates dispense a low value. This can also be seen in the truth table presented in table 7.1.

value rail	**phase rail**	**high rail**	**low rail**
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

Table 7.1: Truth Table, Format Conversion, Phase 1, 2-to-4 phase split, completion detection

The circuit implementation of phase 0 is similar to the one of phase 1 and is shown in figure 7.15. The only differences are the input negations of the AND gates, causing this unit to only propagate values, having the same logical level on both rails, or more specific if the parity is even. All others are blocked and the NULL value is sent due to the fact, that both AND gates evaluate to a low value. More details shows the truth table in table 7.2.

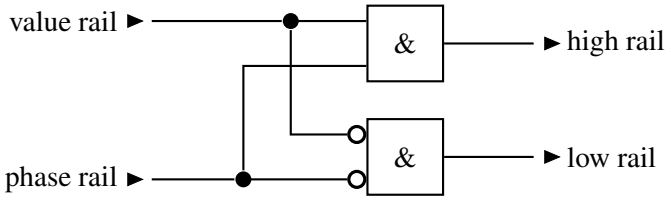


Figure 7.15: Format Conversion, Phase 0, 2-to-4 phase split, completion detection

The truth tables show very clearly, that the two FC blocks are dual i.e. that the values

value rail	phase rail	high rail	low rail
0	0	0	1
0	1	0	0
1	0	0	0
1	1	1	0

Table 7.2: Truth Table, Format Conversion, Phase 0, 2-to-4 phase split, completion detection

propagated by the one handling phase 0 are blocked by the one handling phase 1 and vice versa. Furthermore in a 2-phase communication protocol the next value is achieved from the current one by changing a single rail, which also switches the parity. This implies that the FC unit that sent a DATA value before sends the NULL value afterwards and the one propagating the NULL value before forwards the DATA value afterwards. For that reasons it is indeed possible to attach both of them to the same input, leading to an alternating propagation of the DATA values as well as a valid 4-phase communication protocol at the User Data and Test Data output.

In a similar fashion as done in section 7.1 it can be shown that the implementation is only QDI, due to the fact that the AND gates violate the *indication principle*. In detail an input change is not always recognisable when the output is low. Therefore it can happen that a low input line masks the other, heavily delayed, leading to massive disturbances and failures. Again the reader is encouraged to design a situation leading to that undesired behaviour.

Control Split

Due to the fact that the request signals, used to indicate that the data on the lines are ready to be read are already integrated in the data themselves, this building block just has to handle the acknowledge lines in its Basic Implementation. However if an increased concurrency is needed, additional units like a phase detector have to be integrated. These generate additional control signals that also have to be integrated into the control split block, which results in more complex designs. Concrete implementations for this unit are presented in the following section.

7.4 Implementations of 2-to-4 Phase Split

Basic Implementation

In the Basic Implementation the single unit on the data path is the data format conversion block, converting the 2-phase input automatically into two 4-phase outputs. By looking at the format conversion (FC) implementations for phase 0 (figure 7.15) and phase 1 (figure 7.14) one can see, that each input value only causes a single AND gate to forward a high value in both units combined. This further implies that exactly one output line has a high value at a time which is equal to the fact, that one output is in its DATA-phase and the other one in its NULL-phase .

As already mentioned earlier the only thing left to do for the control split block is to handle the acknowledge signals, or more specifically to acknowledge the input as soon as the acknowledge signal has been received at the User Data and Test Data output. The corresponding STG can be seen in figure 7.16.

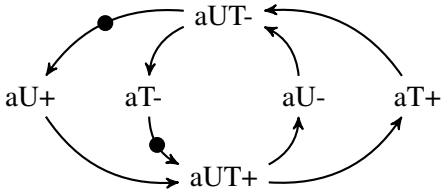


Figure 7.16: STG, Basic Implementation, control split, completion detection

The circuit generated by *Petrify* is shown in figure 7.17. Despite the fact that the control split unit is DI, because it only consists of a single Muller-C gate, the whole implementation is just QDI, due to the data format conversion block, which was shown to be QDI (see section 7.3).

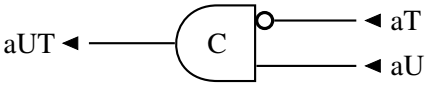


Figure 7.17: Circuit, Basic Implementation, control split, completion detection

Early NULL-phase

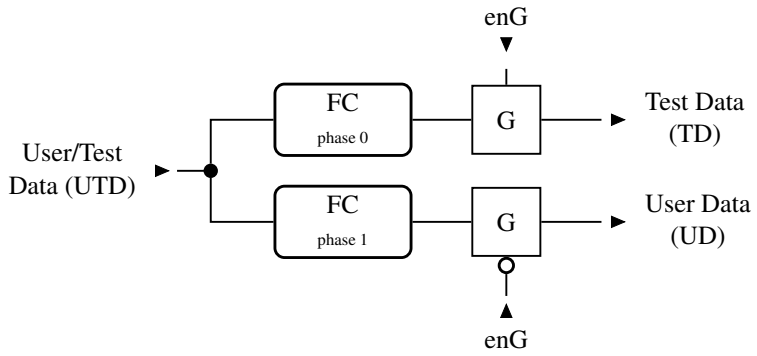


Figure 7.18: Data Path, Early NULL-phase, 2-to-4 phase split, completion detection

To implement the design idea presented in section 5.6 additional units, more specifically two Gatekeepers, have to be introduced into the data path. The complete layout is shown in figure 7.18, where the Gatekeepers, denoted by the letter G, are introduced at the Test Data and User Data output, making it possible to prevent data from propagating. The signal enG is the control line for the Gatekeepers telling them to block or to propagate data. In the first case these units forward a low value on both rails, which is equal to the NULL-phase, making it possible to start the NULL-phase at the User Data output by setting the enable signal enG to a low value and on the Test Data output by changing enG to a high value.

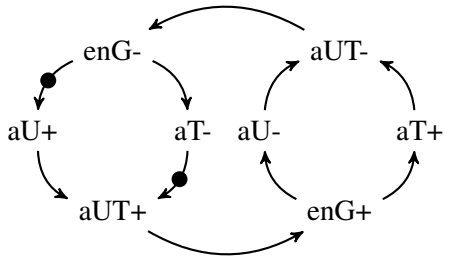


Figure 7.19: STG, Early NULL-phase, 2-to-4 phase split, completion detection

The corresponding STG can be seen in figure 7.19. In the starting position the enable signal enG is low, implying that the Gatekeeper at the Test Data outputs blocks incoming data and the one at the User Data output forwards them. Therefore the acknowledge line aU eventually gets a high value, automatically acknowledging the input and changing enG to a high value. This causes the Gatekeeper at the User Data output to start blocking, which starts the NULL-phase due to the fact that both lines are set to a low value. This gives the NULL values more time to propagate to the next pipeline stage. By closing the Gatekeeper at the UD output the one at the TD output gets opened, which however has at that point in time no effect because the format conversion block handling phase 0 forwards a NULL value anyway. Only after the data at the input have changed they are forwarded to the Test Data output and get eventually acknowledged,

starting the cycle over again.

The resulting circuit implementation is shown in figure 7.20. It can be seen that the enable signal is equal to the acknowledge signal for the input, yielding a very simple circuit. The Muller-C gate is again used to assure that both outputs are in differing phases.

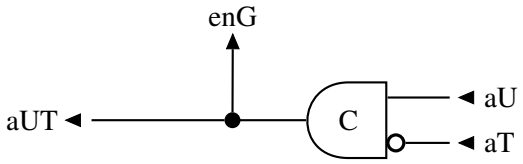


Figure 7.20: Circuit, Early NULL-phase, control split, completion detection

To assure correct functionality it has to be taken care that the control signals arrive at the Gatekeepers before the next data of that line show up. If that is not the case it might happen that the DATA-phase is aborted before it was acknowledged and maybe in addition the introduced NULL-phase might end early starting an extra DATA-phase with the same value again. For that reason QDI is required, causing the control signal to reach the Gatekeepers at the same time as the preceding pipeline stage.

Early DATA-phase

In the Early DATA-phase approach the input is acknowledged as soon as the output currently in its DATA-phase has sent the acknowledge signal, making it possible for the input to send the next DATA values earlier. If only the data format conversion unit would be used on the data path, as it was done for example in the Basic Implementation, the new data would be automatically propagated to both outputs, implying that data are sent to the output currently in its NULL-phase before the acknowledge signal was received. To prevent this incorrect handling of the communication protocol the new data have to be stalled until the receiver at that output indicates, that it is ready to accept new data. This behaviour can be realised by a Gatekeeper unit at each output which however requires an additional phase detection unit on the input to know when to block which output. The resulting data path scheme, can be seen in figure 7.21.

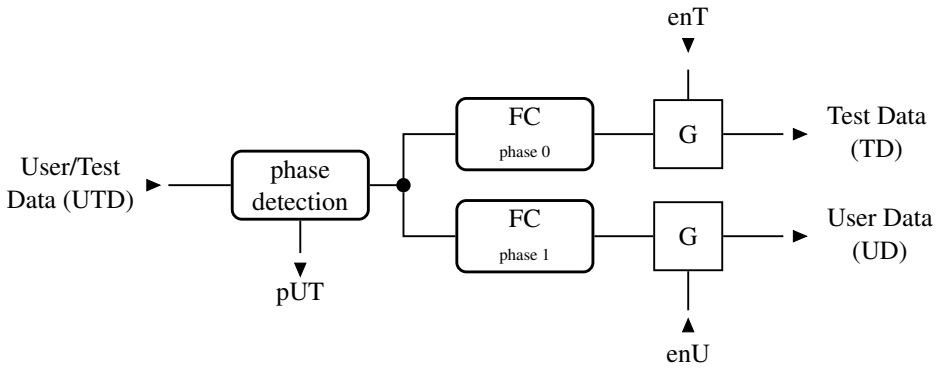


Figure 7.21: Data Path, Early DATA-phase, 2-to-4 phase split, completion detection

As mentioned before the Gatekeepers at the outputs are used to prevent data propagation if the NULL-phase has not yet been completed. For that purpose they start blocking as soon as the phase resulting in a NULL-phase at that specific output shows up at the input, in particular if phase 0 is currently processed then the Gatekeeper at the UD output closes and if phase 1 is worked on the one at the TD output closes. It is not necessary to start blocking the input so early because the data format conversion block would convert it to the NULL value anyway, however in that way larger delays are tolerable. For example, if the delay to the Gatekeepers is too big, they just start blocking after the next DATA-phase was propagated to their input. This results in a propagation of DATA values despite the NULL-phase was not finished yet, a termination of the DATA-phase before it was acknowledged and maybe even the sending of the same DATA value twice, if the blocking sequence of the Gatekeeper is shorter than the input signal. A more robust version would be to close these gates at the same time the input is acknowledged, i.e. a combination with the Early NULL-phase approach, which is however not further pursued here.

In contrast to the data path the control split unit differs heavily to the Early NULL-phase design which can be seen best in the STG shown in figure 7.22. As soon as new data arrive at the input one Gatekeeper starts blocking and the other one starts propagating if the NULL-phase has been completed. One very important thing to see is, that the signal $aUT+$ solely depends

on $aU+$ and $aUT-$ solely on $aT+$.

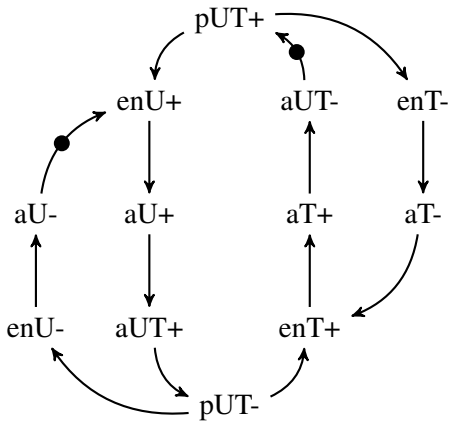


Figure 7.22: STG, Early DATA-phase, control split, completion detection

As already mentioned above there are some race conditions that have to be checked when the design is integrated, more specifically that the Gatekeepers start blocking before new data for their output arrive. However it was tried to give these units as much time as possible such that a violation of the timings is very improbable.

The results received from *Petrify* can be observed in listing B.10. There, an increased complexity is noticeable, resulting from the increased concurrency.

Additional Latch

By introducing an additional latch at the input a better decoupling can be achieved. In detail the value of the input is stored, making it possible to acknowledge it right away. Please note that a latch has to be introduced on each single input rail, more specifically two times as many as with the bundled data approach. The rest of the circuit remains unchanged.

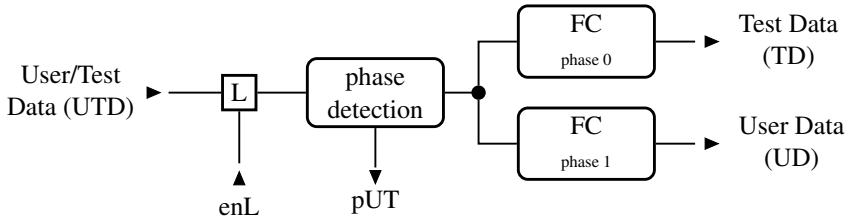


Figure 7.23: Data Path, Additional Latch, 2-to-4 phase split, completion detection

The improved data path can be seen in figure 7.23. The building block labelled with L thereby represents the latch, which is transparent if the control signal enL is high and closed if enL is low.

As soon as the phase detection unit indicates new data the latches are closed, causing them to hold their actual values and at the same time the input gets acknowledged. Afterwards the circuit waits until on both outputs the acknowledge signal has been received and afterwards opens the latches at the input. Only after that has happened a new phase can be identified by the phase detection unit, because just in that case new data are possible to pass the latches, starting the cycle all over again.

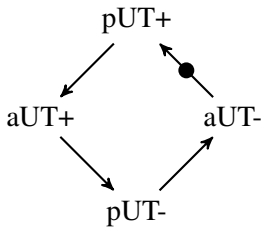


Figure 7.24: STG, Additional Latch, 2-to-4 phase split, completion detection

It is possible to design the latch control and the acknowledgement of the input in different circuits. The STG for the latter one is shown in figure 7.24 and is very simple. As soon as the phase detector announces a new phase, which implies that the latches are transparent, the input is acknowledged. At the same time the latches have to be closed, i.e. they have to store their current value, until both outputs have acknowledged their phases. This represents a very tight coupling and may be loosened however it yields a very simple circuit implementation, which can be seen in figure 7.25. There in addition the circuit of the STG shown before is also integrated, as a simple connection from pUT to aUT. For the latch control the acknowledge signals of the

outputs have to be connected using a Muller C gate to assure that both outputs got acknowledged. The result of that gate is then connected together with the signal pUT to an equality gate, whose output is set to a high value if both inputs have the same logic value. At startup the equality gate forwards a high value, causing the latches to be transparent. As soon as a new data phase is detected, pUT switches causing enL to change to low, i.e. store the values in the latches, until both outputs have acknowledged their phases. If that happens the Muller-C gate switches its output value causing the latches to get transparent again giving the phase detector the chance to detect a new data phase.

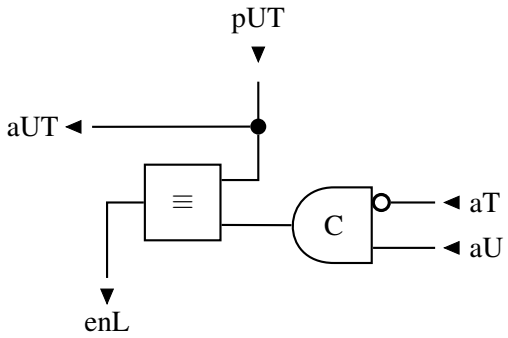


Figure 7.25: Circuit, Additional Latch, 2-to-4 phase split, completion detection

This implementation has the big disadvantage that the timing requirements are very complicated. Even QDI is not enough because this only implies that pUT arrives at the prior pipeline node and the equality gate at the same time. If the latter is very slow it might happen that new data already arrive before the latches are closed. To make the circuit more robust the simple latch can be replaced by a capture-pass event-controlled storage element as described in [35] and [34, pp. 9-28]. The capture input has to be connected to pUT and the pass input to the output of the Muller-C gate. However also with this specific unit QDI is not enough to assure correct behaviour because this only states, that the enable signal reaches the storage element at the same instant as the previous pipeline stage. However in general the time it takes to store a value is far less than it takes to put new data on the line plus the time it takes these values to propagate to the input of the 2-to-4 phase split unit.

Similar to the 4-to-2 phase merge unit the latch control for an ordinary latch may be integrated into the STG, unfortunately again without yielding advantages. For that reason the usage of a simple latch in this implementation is only reasonable in combination with a thorough timing analysis carried out at design time to assure, that the actual values of the data lines are stored before the new values show up.

CHAPTER 8

Proposed Solution - Extensions

In section 5.2 some restrictions were introduced to ease the development of the test approach. In this chapter it is investigated if these restrictions may be lifted or at least softened without changing the overall functionality.

8.1 Cyclic Pipeline

The first restriction affected the pipeline structure, more specifically no loops inside the CUT were allowed. However in real life applications feed back or feed forward loops are very well imaginable so in this section it is studied, if the proposed approach is also applicable to cyclic pipelines.

In general, verifying a result depending on the previous input values, is very hard. However if the test approach described in this thesis is used, all test vectors as well as their order are known a priori and for that reason the results can also be computed a priori, even if cyclic pipelines are used. One major problem with this pipeline structure is, that faulty values are fed back or forward into the loop before they were checked, making it possible that they introduce periodic failures each time the incorrect value interchanges with a correct one. Moreover it can't be told if the fault is still present or if the present failures are just introduced by the incorrect feed back values. The source of these problems is the fact that the history, e.g. the number of values that have been used for computing particular data may become infinitely in cyclic pipelines. For that reason they have to be cleared after a fault was detected by applying input vectors to the CUT, that reset the internal registers to the initial state. This also has to be carried out for the test vectors. Another possibility is to activate the reset signal locally for the specific block if possible. Alternatively it would be possible to prove for the given settings that a cyclic failure based on the Test Data and the expected User Data is not possible at all.

In the following subsections some more specific forms of cyclic pipelines are investigated as well as some restrictions that have to be taken care of. For the purpose of visualisation the notation from [34, chapter 3] is used, which represents latches as boxes with double vertical lines whereas the logic in between is neglected. If the label in a box is surrounded by a circle, the latch holds a valid value and if not the value has already been consumed by the unit afterwards, i.e. the latch holds a so called bubble. To indicate a NULL value the label E is used and for a DATA value the label V. In this thesis the usage is extended to the 2-phase protocols, whereas the label U indicates User Data belonging to phase 1 and T Test Data belonging to phase 0.

Feed Back Loop

In a feed back loop the value of a later stage of the pipeline is used in an earlier one. The schematic view for the synchronous case is shown in figure 8.1. The quadratic blocks in the picture represent a pipeline stage, i.e. the data move ahead one stage at each clock tick. The data lines are split up at the start point of the loop, marked with a black dot, fed back to a previous pipeline stage and integrated in the logic, shown as black circle. Please note that the minimum length of the feedback path is 1 and not like the picture might suggest 2 or more.

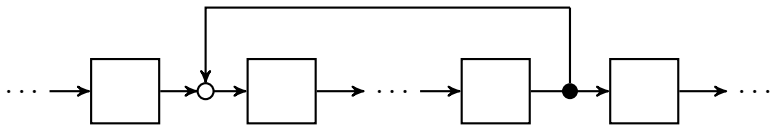


Figure 8.1: Feed Back Loop, Synchronous Pipeline

This concept has to be adopted to asynchronous logic to check if it is also testable with the test procedure presented in this thesis. For that purpose it is first converted to work for a 4-phase communication protocol and afterwards adapted to a 2-phase one. This intermediate step is reasonable, as it will be shown later, because the 4-phase implementation is very similar to the final one.

The main problem when using feed back loops in a 4-phase protocol are the alternating DATA-phase and NULL-phase. It has to be taken care that DATA values only interact with other DATA values and not with NULL values. To assure this property it is necessary to integrate two additional nodes in the feed back loop for each pipeline stage that has to be passed. This means that if the loop jumps back n pipeline stages, $2 * n$ additional nodes are required. Figure 8.2 shows the 4-phase version of the synchronous pipeline implementation. The fed back data as well as the data of the pipeline are joined using the Join unit also presented in [34, chapter 3]. This building block requires that both inputs are in the same phase, i.e. either DATA-phase or NULL-phase, before it propagates the input values to the output. To prevent a deadlock right at the start, the nodes on the feed back path have to be initialised, starting at the first every other with a DATA value and all others with a NULL value. The initial DATA values have to be chosen carefully, in the best case they are neutral elements to the operation that is carried out on them, e.g. 0 if the operation is an addition or 1 if it is a multiplication.

The first data token on the pipeline joins the first data token in the feed back loop and propagates through the whole pipeline until it reaches the fork unit. Its current value is afterwards inserted at the end of the loop, if the last node on the feed back path already contains a bubble. The following NULL value is handled in the same way. After the next $n-1$ DATA and NULL values the first inserted value is ready to be joined with the value that appeared n steps later, which is exactly the same functionality as in the synchronous case.

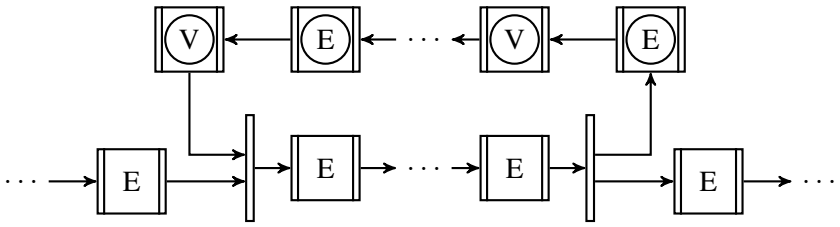


Figure 8.2: Feed Back Loop, Asynchronous Pipeline, 4-phase

As already mentioned the structure presented for the case of a 4-phase protocol is very much alike the one that is used for the test approach presented here, despite the fact that inside the CUT a 2-phase protocol is used. In general the phases in a 2-phase communication protocol are allowed and, after a slight conversion, well able to interact, however in the CUT one phase only contains User Data and the other one only Test Data. For that reason they must not interact at all because otherwise the functionality of the whole unit would be destroyed. Due to the fact that the same restriction was introduced for the 4-phase implementation, the structure can be adopted as it is with just minor changes. At first the labels are changed from V to U indicating DATA values, and from E to T displaying TEST values. Furthermore the Join and Fork units also have to be replaced by their counterparts capable of the 2-phase protocol. The complete feed back loop can be seen in figure 8.3.

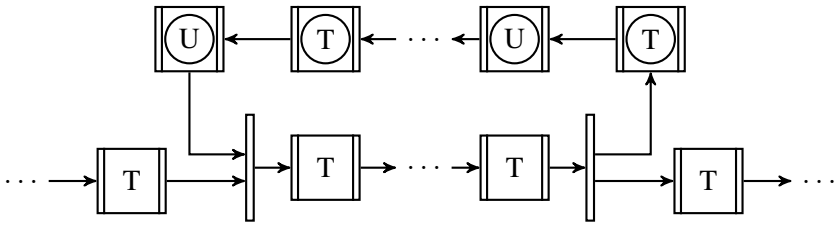


Figure 8.3: Feed Back Loop, Asynchronous Pipeline, 2-phase

Please note that the presented pipeline structure is the minimal feasible feed back path. To improve the performance, it is of course possible to add latches in the feedback path, as is was described in [34, chapter 3]. Nevertheless the count of initialised nodes has to stay the same because they define how long the DATA values are delayed.

Feed Forward Loop

In a feed-forward loop the actual value of a specific stage is used also in a stage further ahead in the pipeline as the original one. Figure 8.4 shows this case schematically for the synchronous case. At the point marked with a black dot, the data lines fork and one part is forwarded to another stage, marked with a black circle. As already mentioned in the description of the feed back loop the boxes represent a pipeline node such that the data propagate to the following node at each clock tick. It has to be noted, as also mentioned when describing the feed back loop, that the minimal count of stages that are passed by the loop is one and not two or more like the figure might suggest.

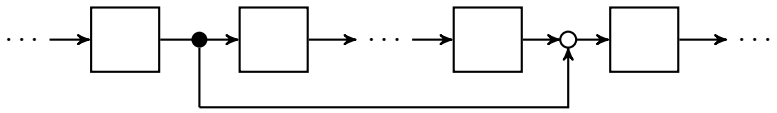


Figure 8.4: Feed Forward Loop, Synchronous Pipeline

When adopting this concept to a 4-phase communication protocol it again has to be taken care that the correct phases interact i.e. DATA-phase with DATA-phase and NULL-phase with NULL-phase. If the structure of the circuit stays the same as in the synchronous case severe problems occur. Assume that the data should be forwarded right to the following stage. However at the time one stage processes the DATA-phase the following one already handles the NULL-phase of the previous DATA values. This would make it impossible to create a feed-forward loop over an odd amount of stages.

To counteract this behaviour, every synchronous stage controller that is bypassed by the data on the loop has to be replaced by two asynchronous controllers, as it is shown in figure 8.5. Due to the fact that the 4-phase implementation as well as the one for the actual test approach only differ by labels, the picture already shows the final design. Assume that n stages are bypassed by the loop then the n synchronous controllers have to be replaced by $2 * n$ asynchronous ones, capable of handling 2-phase communication. Starting at the last one every other has to be initialised with a User Data token and all others with Test Data one. This is necessary to assure on the one hand the correct delay of the values and on the other hand to prevent deadlocks, that would appear if the Join unit at the end of the loop does not get a pair of the same phase at its input.

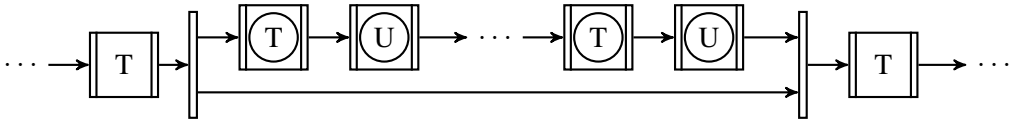


Figure 8.5: Feed Forward Loop, Asynchronous Pipeline, 2-phase

At the time the first DATA values arrive at the last stage in front of the loop, they are immediately forwarded to the Join unit at the end. Due to the fact that both inputs of the Join unit are in the same phase (User Data) the values are forwarded to the next stage and eventually

acknowledged. After all elements of the pipeline in the loop have progressed to the next node, the first one gets empty, implying that it now is ready to accept the value that is stored in the previous stage. Afterwards the data are propagated through the two times n stages and finally interact with the correct data. Please note that the speed can be significantly increased if additional nodes are introduced into the pipeline inside the loop, however there always have to be two times n initially filled stages present. Please note that twice the amount of nodes does not automatically mean that the hardware overhead is 100 %. Due to the fact that an asynchronous latch is one half of a synchronous flip flop.

A possible extension to the above presented pipeline structures in the synchronous elements is to add additional controller nodes on the feed-back/forward loop, used to delay the values for one clock tick per introduced unit. This behaviour can be realised in the asynchronous case too, by simply adding two additional stage controllers for each delay controller node in the synchronous pipeline, which have to be filled with tokens.

Computational Loop

Besides feed-back and feed-forward loops also more general circuits are testable using the proposed test scheme however with restrictions. In the following, circuits that accept a value at the input and compute the result in an arbitrary amount of steps are investigated. An example circuit can be seen in figure 8.6, which calculates the greatest common divisor (GCD) of two numbers. Please note that the implementation in the picture is based on 4-phase logic so it has to be converted to a 2-phase one to be testable by the test scheme proposed in this thesis. To achieve that, the NULL-phase has to be replaced by an additional DATA-phase and the control circuits have to be adopted to 2-phase logic.

In the adopted configuration the circuit accepts one User Data and one Test Data at the input and afterwards starts the computations. After these are finished the results are propagated to the output and new inputs are read. As long as the Test Data and the User Data are processed in the same amount of computation steps no problems occur, because they both get read in in the same circle and the result is also presented at the output in the same computation cycle. However if User Data and Test Data are not computed in the same amount of steps, as it can happen easily when calculating the GCD, serious problems may appear. Assume for example that it takes five iterations to process the User Data value and three for the Test Data one. At the beginning the DATA value of phase 1 is accepted at the input and afterwards the one of phase 0. The stage in front of the GCD will then propagate the next User Data value to the input. In the meantime the calculations start until phase 0 is finished. In that case the circuit tries to read the next Test Data value from the input where however only a User Data value is available. This causes the circuit to deadlock because the Merge unit at the entry only works if the values at its inputs are in the same phase.

As this little example shows only circuits delivering the result in a fixed amount of computation steps are testable by the proposed test approach. If it is desired that also circuits are tested not fulfilling this property, then the Test Data would have to be adopted to the actually processed

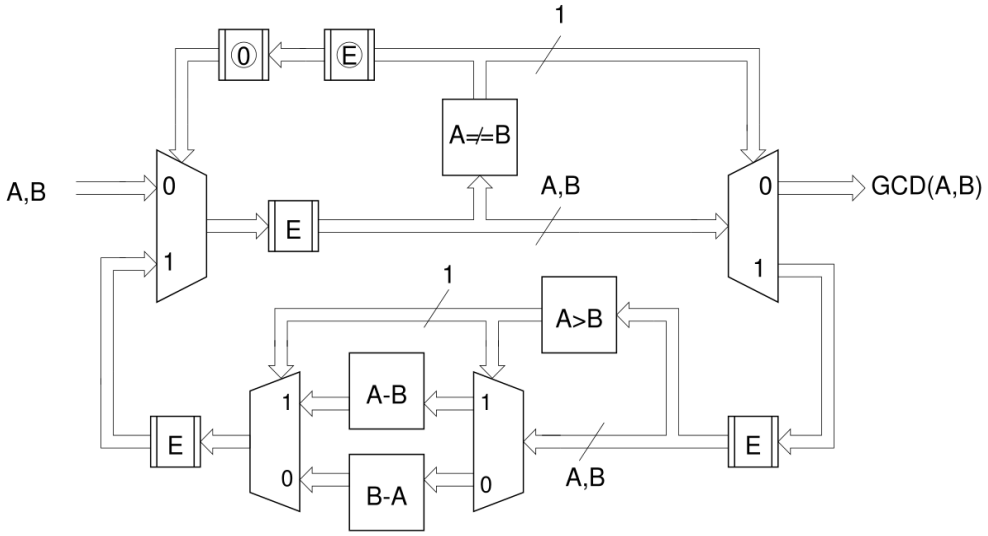


Figure 8.6: circuit calculating the greatest common divider of two numbers [34, p.39]

User Data, destroying however the foundation of the proposed test scheme because in that case the applied test vectors are no longer known a priori and therefore the whole test approach is not applicable at all.

Restrictions

In addition to the already stated conditions for the usage of loops, one more rule has to be introduced, namely that a loop has to start and end inside the CUT. This means, that it is not allowed for a loop to start inside the CUT and end outside as well as to start outside and end inside. The most obvious reason for this is, that the communication protocol used within the CUT does not match the one at the outside (4-phase vs. 2-phase).

8.2 Storage Elements

Registers

The second restriction introduced in section 5.2 forbids registers, used to store parts of the result such that it can be used in the next computation. Due to the fact that User Data and Test Data are processed in an alternating fashion this behaviour is not allowed because otherwise they would start to interact which has to be prevented under all circumstances. More specific a value stored in e.g. phase 0 has to be kept hidden while the stage works in phase 1 and is only allowed to be used if another value of phase 0 is processed.

There are two possibilities to actually include the register concept. The first one is to use a feed back loop only over that particular stage. In that case the values are not stored in the control node but travel around the stage and arrive at the input at the same time new data show up. Nevertheless this method requires lots of hardware because at least two additional nodes have to be introduced on the feed back loop. The second possibility is to use an actual register in the node however duplicated i.e. one for phase 0 and one for phase 1. The pipeline stage has to detect the phase the circuit is currently in and use the appropriate register.

The huge disadvantage of this method is that the User Data registers are not testable by the proposed approach, making it necessary to add an additional test mechanism observing the registers, like a parity checker for example.

State Machines

Similarly to registers also state machines, i.e. circuits that compute their output based on their current internal state and the input, are very problematic units when using the proposed test approach. If the internal 2-phase User/Test Data stream would be connected to the input of that kind of unit, both would interfere causing miscalculations.

One solution for this problem is to duplicate the state machine, whereas one duplicate is assigned to phase 0 and the other one to phase 1, and furthermore to implement a control logic forwarding the data to the appropriate unit, depending on the actual phase. Another approach would be to just double the registers storing the internal storage which has the advantage of less hardware overhead. However as mentioned before the hardware only used for User Data is not tested by the proposed test approach, demanding additional test methods.

CHAPTER 9

Proof-of-concept

In this chapter a concrete implementation of the theoretical concepts described in the previous chapters is presented, which is furthermore used to show the correct functionality of the stated test approach. In addition the introduced delay and the additional required area is calculated analytically to estimate the overhead.

9.1 Introduction

To verify the proposed test approach a simple pipeline structure with three stages (see figure 9.1) was chosen as CUT. The squares in the figure represent controller nodes which are connected directly to each other, i.e. without any logic in between. Therefore this implementation does not compute anything but only propagates input values without modifying them. This pipeline design is despite its simplicity suitable for the desired purpose, because mainly the merging and splitting of the data had to be investigated. Furthermore this approach was chosen to make the results better readable and easier to understand. The feed back path in the figure is drawn dashed, meaning that it is not part of the original pipeline and is only added at the end to show the correct functionality when using a cyclic pipeline structure.

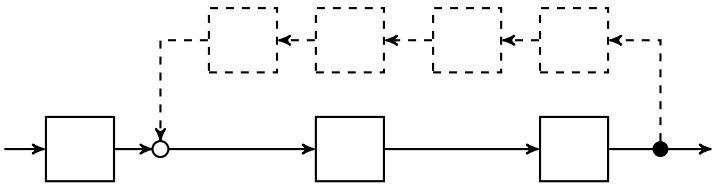


Figure 9.1: Pipeline Schematic used for the implementation, pipeline nodes squares, optional feed back path dashed

For each design style, i.e. bundled data and completion detection, three different versions have been implemented. The first one only consists of the three pipeline stages and is used to verify the correct functionality of the original pipeline. In the second version the devices used to test the CUT are added to realise a CCUTT approach, more specific a 4-to-2 phase merge unit and a test vector generator are installed in front of the first pipeline node and a 2-to-4 phase split unit as well as a test vector analyser right after the last one. In the third version finally the feedback path is added to the pipeline structure. In this case the 2-to-4 phase split unit is mounted after the point feeding the data into the feedback path. The 4-to-2 phase merge and 2-to-4 phase split unit were implemented in their Basic Implementations, saving the other implementation styles for future research. Also the TVG as well as the TRA were kept as simple as possible, by implementing them as mere lists applying one element after the other to the input respectively comparing the output to the next element in its list.

The data path itself was implemented by eight rails, resulting in eight information bits when using bundled data and four when using completion detection. The control lines also had to be adapted depending on which communication style was used, as it has been described earlier.

The implementation was written in VHDL, synthesised in *Quartus®II 32-bit Version 12.1 Build 177 11/07/2012 SJ Web Edition* and afterwards simulated using a Post-Layout simulation in *ModelSim-Altera® Starter Edition 10.1d*, which both are available for download at [6].

Please note that the main purpose of this implementation was to prove, that the proposed testing scheme is working according to its specifications in an actual implementation. This means that no time was invested to optimise the VHDL code in terms of speed and area, which is also one of the reason why the overheads were calculated analytically and not taken from the implementation. Nevertheless the usage of such a minimal test circuit is a valid proof-of-concept implementation due to the following reasons:

- It can be shown that testing is carried out completely concurrent, i.e. without interrupting the normal operation.
- It can be observed very well if the 4-to-2 phase merge and 2-to-4 phase split units combine and separate the 4-phase respectively 2-phase data streams as desired.
- The test approach works independently of the CUT making it possible to replace the simple pipeline by any circuit fulfilling the criteria stated earlier without changing the overall functionality.
- It can be proven, that feedback loops work for a feedback count of two, which can be easily extended to n without effects to the rest of the implementations.
- SST can be achieved from the shown implementation by replacing the CUT by a single combinational cloud. Again this does not change the overall functionality only the expected test vectors have to be altered according to the introduced logic.

9.2 Implementation

Bundled Data

As mentioned earlier no logic clouds between the pipeline nodes were implemented, implying that the data are forwarded as they are from one pipeline stage to the following. This is, however, not true for the last version, i.e. the one with the feed-back loop. In that case the lower four bits are taken from the feed back path and the other ones from the previous stage. If logic clouds would have been used they would not have to be altered when adding the test procedure because one property of the bundled data design style is, that it is possible to use the same logic units in the 4-phase and 2-phase style.

The dataflow of the implementation can be observed in figure 9.2 and 9.3, whereat in the latter one a loop was introduced into the linear pipeline. The signals in the figure are arranged according to the order they are processed, starting with the input at the top and ending with the output at the bottom. At the very top the input interface, i.e. the input data ($data\_In$) and the request (req_In) and acknowledge line (ack_In), are shown. Afterwards the output of the test vector generator (TVG) is presented, which is combined with the data input inside the 4-to-2 phase merge unit to the signal PhM_Out. That one is afterwards processed by stage 1, resulting in signal $stage1\_Out$, which is fed into stage 2, leading to signal $stage2\_Out$. The output of the last stage is the same as the input of the 2-to-4 phase split unit, which is shown in line PhS_In. The 2-to-4 phase split unit afterwards splits User Data and Test Data resulting in the data streams that are described by TRA (TD) and $data\_Out$ (UD). For the latter also the request (req_Out) and acknowledge (ack_Out) line are shown. Please note that the request and acknowledge signals inside the CUT are not shown in the picture to improve the readability.

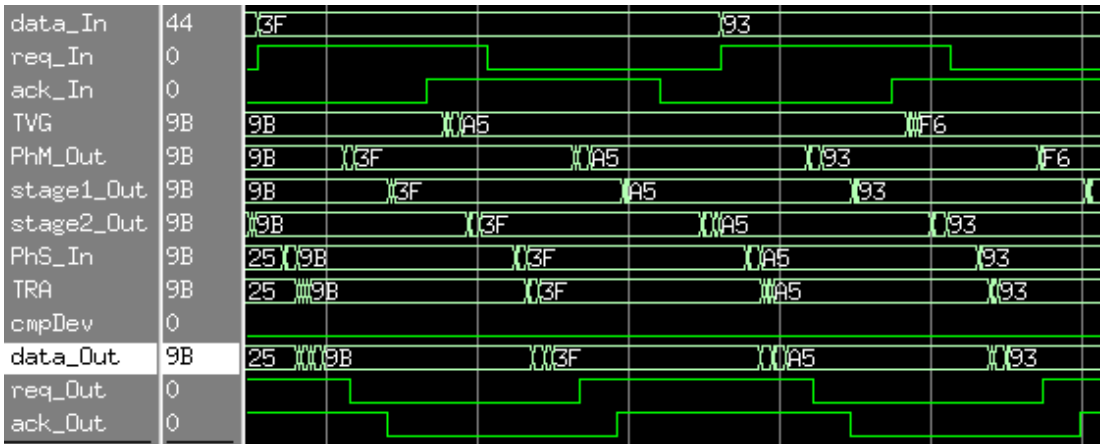


Figure 9.2: Simulation of bundled data implementation without loop

Figure 9.3 shows the implementation with a loop in the pipeline structure. The behaviour is very similar to the one described above with the difference that the output of the feedback loop,

denoted as the signal $feedback\_Out$ in the picture, is combined with the output of stage 1 and fed together into stage 2. This can be seen best when comparing the signals $stage1\_Out$ and $stage2\_Out$ where the lower four bits are replaced by the value 2 if it is a DATA value and by 7 if it is a TEST value.

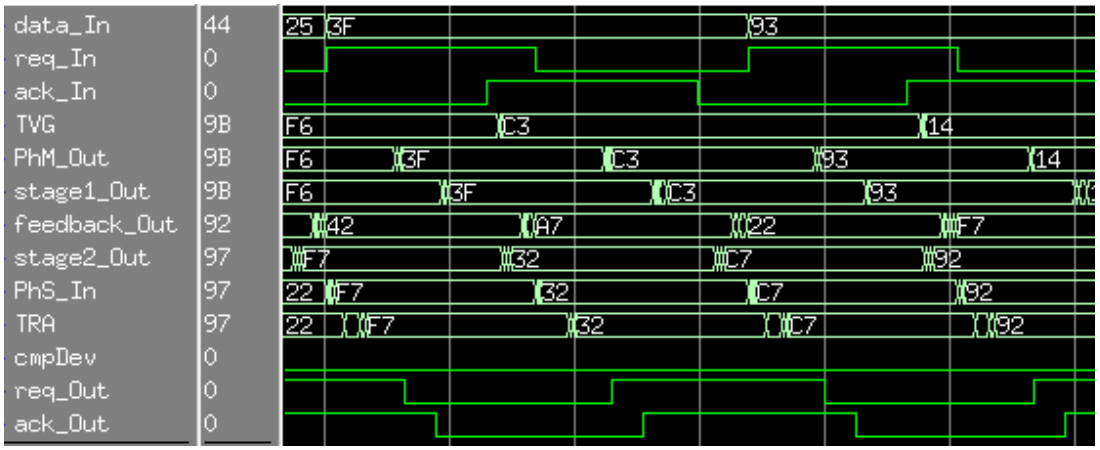


Figure 9.3: Simulation of bundled data implementation with loop

Completion Detection

Like in the bundled data implementation no logic gates were used between the pipeline stages, causing the data to be propagated without changes. Solely when a feed back loop is used the upper two bits (four rails) are taken from the feed back path and the lower two from the previous stage. If logic gates are used they have to be adapted to work with the new data encoding scheme, which in general increases the size of the gates.

The operation within the CUT is shown in the figures 9.4 and 9.5. In the first case a linear pipeline was used, in the latter one an additional feedback loop was introduced. The overall functionality is the same as with bundled data, described already earlier, with the difference, that the data encoding is changed for the Test Data in the merge and split unit (e.g. $5A \rightarrow 0F$), whereas the User Data are not changed at all. This results from the fact that the former are assigned to the phase having an even parity and the latter to the odd parity phase.

9.3 Fault Detection

To verify that faults are detected correctly a stuck-at fault at bit 3 was introduced in the bundled data implementation, causing it to be low all the time. The effects of the introduced fault can be observed in figure 9.6. While the first few values (UD: 44 and 25; TD: 14) are propagated without modification the TEST value $9B$ is converted to 93 which is detected by the test approach

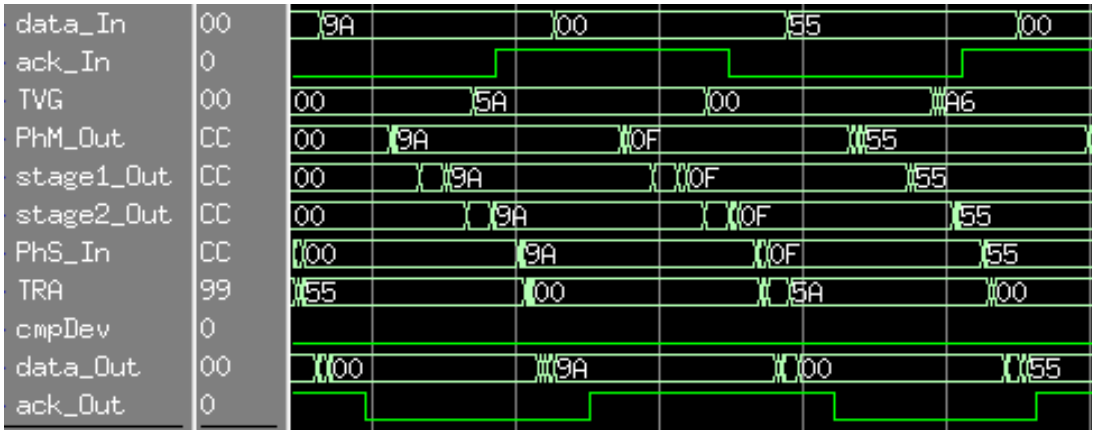


Figure 9.4: Simulation of completion detection implementation without loop

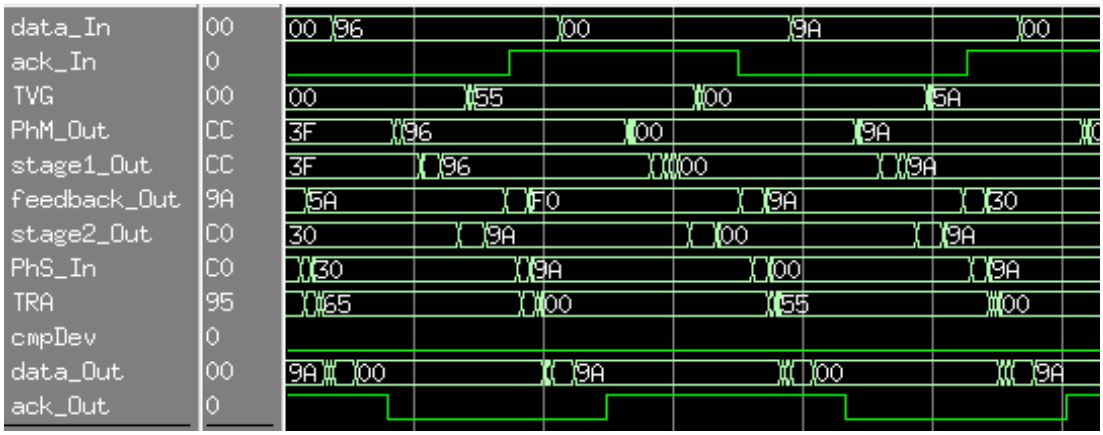


Figure 9.5: Simulation of completion detection implementation with loop

and the detected failure is reported to the outside using a high value on the signal *cmpDev*. Please note that also the DATA value $3F$ following afterwards is changed to 37.

9.4 Area Overhead

The introduction of the test circuitry requires additional area on the chip. For that reason calculations are carried out in this section to estimate the overhead. Despite the fact that the whole circuit was integrated on an FPGA the calculations are carried out analytically and are presented in transistor counts. This method was preferred because the amount of units required may differ from design tool to design tool due to different optimisation algorithms. Another advantage of

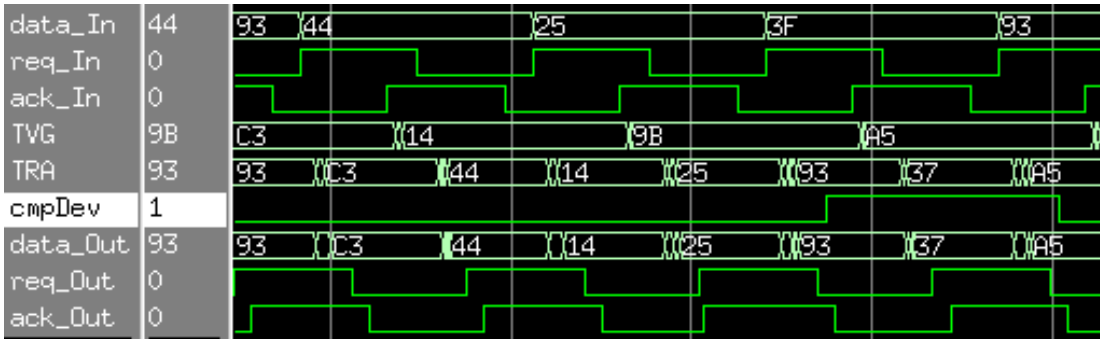


Figure 9.6: Simulation of bundled data implementation with introduced fault

the chosen method is, that an upper bound is received, which may be improved significantly by an optimised implementation.

Table 9.1 lists the transistor count of the gates used in the calculations. Please note that the overhead is only computed for the basic implementations of the test infrastructure. The more elaborate versions are left for future research.

gate	**transistor count**
negation	2
transmission gate	2
NOR, NAND	4
OR, AND	6
XOR	10
MUX, Latch	12
Muller-C	12

Table 9.1: Area requirements for standard building blocks in transistors

Bundled Data

Figure 9.7 shows the layout of a Muller pipeline as it was proposed in [35]. One can see that each pipeline node consists of a latch per data line, a Muller-C element and a negation, overall accumulating to $14 + 12 * k$ transistors whereat k represents the amount of data lines. When assuming n pipeline stages and also considering the combinational logic between the stages, represented in average transistor count per pipeline stage ($comb$), the overall transistor count sums up to

$$native = n * (14 + k * 12 + comb)$$

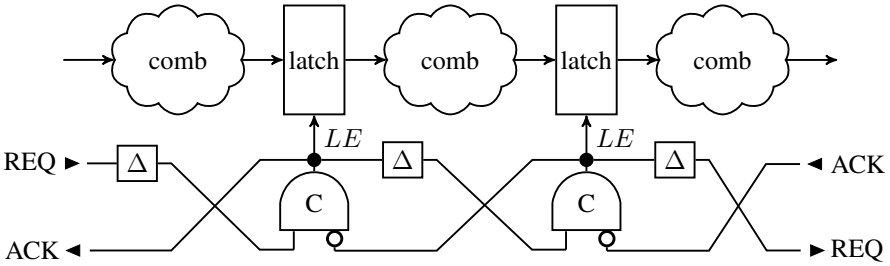


Figure 9.7: Muller Pipeline as proposed in [35]

When introducing the test approach the latches inside the pipeline nodes have to be replaced by capture and pass storage elements proposed by Sutherland [35]. For the calculations the implementation shown in figure 9.8 consisting of 5 inverters, 6 transmissions gates and 2 additional inverters for generating the negated control signals for the transmission gates was chosen, resulting in an overall transistor count of 26. It is possible to integrate this unit on far less space, however in the calculations this pessimistic estimation is used to achieve a safe upper bound of what to expect.

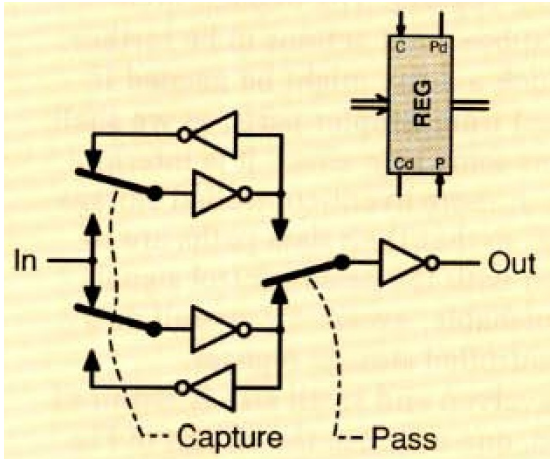


Figure 9.8: Implementation of a Capture and Pass Storage Element [35]

Additionally the 4-to-2 phase merge as well as the 2-to-4 phase split units in their basic implementation have to be integrated, whereat the first one consists of two negations (4 transistors (T)), one Muller-C gate (12 T) and one MUX/data line (DL) (12 T) summing up to $16T + 12T/DL$. The latter one simply consists of two negations (4 transistors (T)) and one Muller-C gate (12 T) resulting in overall 16 T. Assuming n pipeline stages and k data lines the overall transistor count for the bundled data test implementation accumulates to

$$test = n * (14 + k * 26 + comb) + 32 + k * 12$$

The overhead therefore results to

$$\begin{aligned} overhead &= \frac{test - native}{native} \\ &= \frac{14 * k * n + 12 * k + 32}{12 * k * n + 14 * n + n * comb} \end{aligned}$$

By assuming no logic between the stages (worst case assumption) and large values for k (> 32) the expression can be further simplified to

$$\begin{aligned} overhead &= \frac{14 * k}{12 * k + 14} + \frac{1}{n} * \frac{12 * k + 32}{12 * k + 14} \\ &\approx \frac{14}{12} + \frac{1}{n} * \frac{12}{12} \\ &\approx 117 + \frac{100}{n}\% \end{aligned}$$

introducing an error of plus 4 to 5 % for $k = 32$ and around 2 % for $k = 64$.

If the pipeline consists solely of a single stage the overhead results to 217 %, however with an increasing length this value decreases linearly towards 117 %. Please keep in mind, that this model represents the worst case with no combinational logic at all between the stages, a situation that will never occur in reality. By adding l combinational logic transistors per data line (DL), i.e. $comb = l * k$, the formula stated above can be rewritten to

$$\begin{aligned} overhead &= \frac{14 * k}{12 * k + 14 + l * k} + \frac{1}{n} * \frac{12 * k + 32}{12 * k + 14 + l * k} \\ &\approx \frac{14}{12 + l} + \frac{1}{n} * \frac{12}{12 + l} \end{aligned}$$

which is shown in figure 9.9. The x-axis represents the size of the logic and the y-axis the area overhead in %. The calculations were carried out for a data line count of $k = 64$ and a pipeline length of $n \in \{1, 2, 5, 10, 50\}$. As it can be seen very clearly the area overhead drops very quickly as soon as combinational logic is added. When looking at the worst case scenario, i.e. when only a single pipeline stage exists ($n = 1$), then a logic of 12T/DL/stage already halves the relative overhead. When more and more logic is added the overhead of course approaches 0 %.

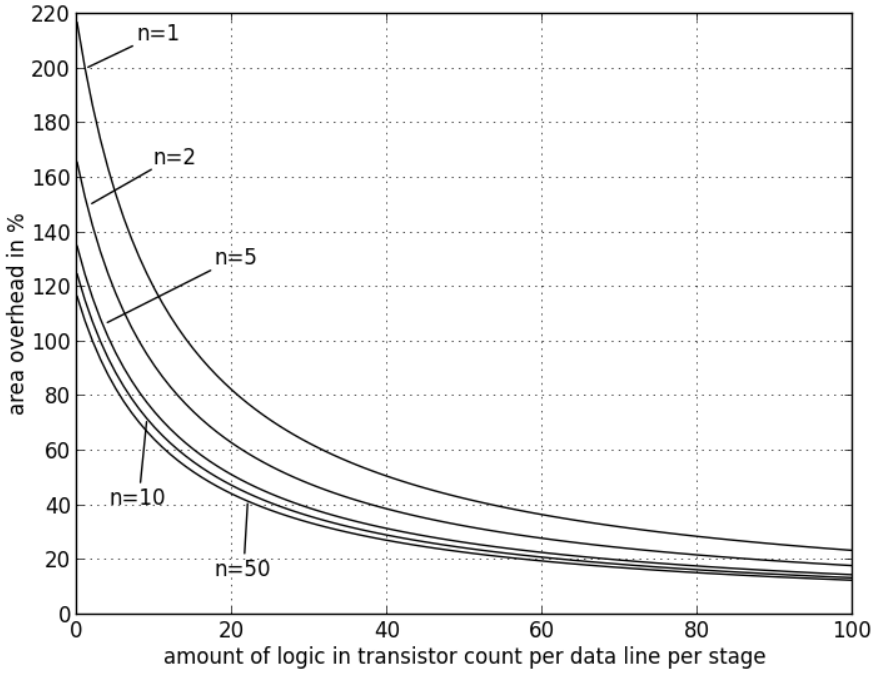


Figure 9.9: area overhead in % in dependence of amount of logic and number of pipeline stages (n), $k = 64$ data lines, bundled data

Completion Detection

The native implementation using the completion detection design style uses also one negation (2 T) and one Muller-C gate (12 T) to control the storage elements within one pipeline stage, however for each data line in addition one OR (6 T) and several Muller-C gates (12 T) are required to implement the completion detection mechanism. In detail both rails of each signal line are connected to the same OR-gate and the outputs of these are then concentrated using a tree of Muller-C elements. This tree requires $k - 1$ such units, where k again denotes the amount of data lines. This tree forwards a high value only when all OR-gates forward a high value, indicating that all lines hold a data value. For a low value the same statements can be made, whereas this indicates that all lines hold a NULL value. Furthermore for each single DL consisting of two rails two capture pass storage elements (26 T) have to be installed, because in contrast to the bundled data approach even the NULL values have to be stored to guarantee a frictionless operation.

In the following calculations the logic was dropped because the additional overhead is not exactly known, due to missing numbers in literature. Therefore an educated guess had to be made which will be discussed separately at the end of this section. Without logic gates the

overall transistor count for the native approach having n pipeline stages results to

$$native = n * (14 + k * 6 + 12 * (k - 1) + 2 * k * 26)$$

When adding the test circuit the communication method is changed from two to four phase yielding several consequences. The first one is that now XOR units are required instead of the OR gates for the completion detection mechanism, resulting in four additional transistors per DL. In addition the 4-to-2 phase merge (2T + 44 T/DL: 2 OR, 2 NOR, 2 Muller-C, 1 negation) and 2-to-4 phase split (14T + 22T/DL, 3 NOR, 1 NAND, 3 negations, 1 Muller-C) have to be inserted, whereat simple optimisations such as combining OR and negation to NOR have been carried out.

By summing up the single components the overall transistor count for the test implementations results to

$$test = n * (14 + k * 6 + 12 * (k - 1) + 2 * k * 26 + 4 * k) + 66 * k + 16$$

The overhead is then calculated to

$$\begin{aligned} overhead &= \frac{test - native}{native} \\ &= \frac{4 * k * n + 66 * k + 16}{70 * k * n + 2 * n} \end{aligned}$$

For large k this simplifies to

$$\begin{aligned} overhead &= \frac{4 * k}{70 * k + 2} + \frac{1}{n} * \frac{66 * k + 16}{70 * k + 2} \\ &\approx \frac{4}{70} + \frac{1}{n} * \frac{66}{70} \end{aligned}$$

This means that for long pipelines the overhead approaches a value of about 6 % however only because the logic was completely neglected earlier. Unfortunately no concrete values of the additional necessary overhead when converting 4-phase logic to 2-phase one could be found in the literature so it is roughly estimated by 100%, which is assumed to be a very pessimistic guess. By adding the combinational logic ($comb$) to the formulae from above and assuming the amount of logic to be l transistors/DL/stage the following formula for the overhead is derived:

$$
\begin{aligned}
overhead &= \frac{test + (1 + 100\%) * n * comb - (native + n * comb)}{native + n * comb} \\
&= \frac{4 * k * n + 66 * k + 16 + n * comb}{70 * k * n + 2 * n + n * comb} \\
&= \frac{4 * k}{70 * k + 2 + l * k} + \frac{1}{n} * \frac{66 * k + 16 + l * k * n}{70 * k + 2 + l * k} \\
&\approx \frac{4}{70 + l} + \frac{1}{n} * \frac{66 + l * n}{70 + l}
\end{aligned}
$$

The last simplification was again based on the assumption of many data lines, i.e. a large k.

Figure 9.10 shows the overhead graphically. One can see that it is very low when no combinational logic is present due to the already very high area requirements of completion detection logic. However as soon as computational blocks are added they start to dominate the overhead directing it toward the boundary value of 100 %. It has to be stated again that the area overhead was just a guess and seems to strongly depend on the specific circuit, which may cause the overhead to drop significantly.

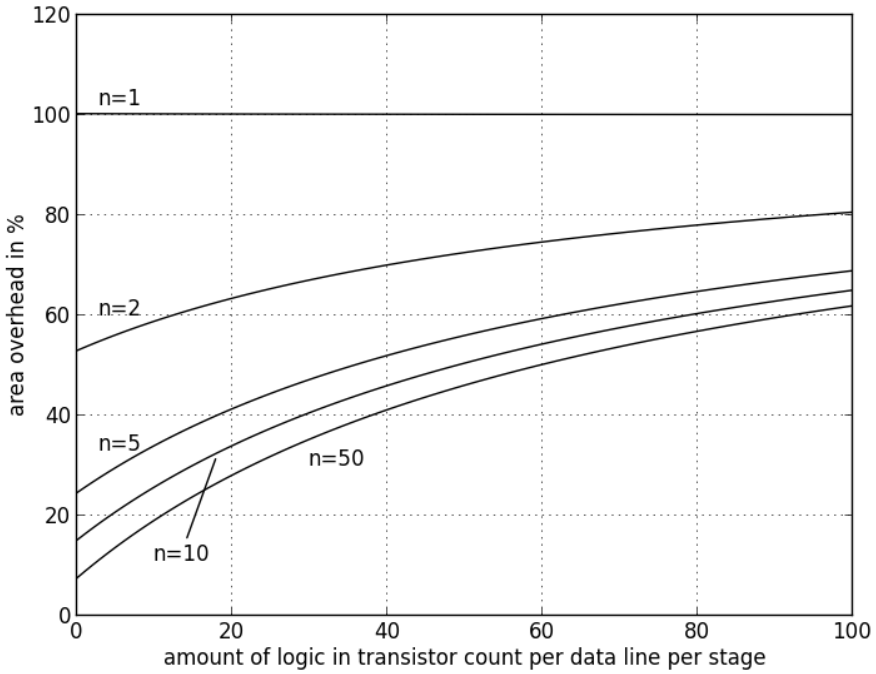


Figure 9.10: Area overhead in % in dependence of amount of logic and number of pipeline stages (n), $k = 64$ data lines, completion detection

Summary

Table 9.2 summarises the results achieved for the area overhead introduced by the test approach for bundled data and completion detection. The first four rows show the hardware effort in transistors for the original pipeline implementation (column *native*) and the additional transistor count required to implement the test approach (column *test*). The last two then present generic formulae for the overhead for a pipeline of length n in % once without and once with combinational logic.

	BD		**CD**	
	native	**test**	**native**	**test**
merge unit	-	$16 + 12/DL$	-	$2 + 44/DL$
combinational	-	-	-	$\approx$*2
pipeline node	$14 + 12/DL$	$14/DL$	$2 + 70/DL$	$4/DL$
split unit	-	16	-	$14 + 22/DL$
n stages (%) without comb.	$117 + \frac{100}{n}$		$5.7 + \frac{94}{n}$	
n stages (%) with comb.	$100 * \left(\frac{14}{12+l} + \frac{1}{n} * \frac{12}{12+l}\right)$		$100 * \left(\frac{4}{70+l} + \frac{1}{n} * \frac{66+l*n}{70+l}\right)$	

Table 9.2: Area overhead estimated by transistor count, l represents combinational logic in transistors/DL/stage

9.5 Additional Delay

By adding the test infrastructure to the original circuit and converting it from 4-phase to 2-phase an additional delay is introduced. In this section analytic calculations are carried out to achieve a rough estimation of what has to be expected. Please note that again only the basic implementation of the test circuit is analysed, the more elaborate ones are left for future research.

The introduced delay was determined in multiples of the unit delay using the gate delays stated in table 9.3. These were determined by observing the longest path in concrete implementations of the single units. Of course more elaborate methods, like logical effort presented in [36], do exist to calculate the delay. For a rough estimation the chosen approach however is sufficient.

gate	**delay**
negation	1
transmission gate	1
NOR, NAND	1
OR, AND	2
Muller-C	2
XOR	3
MUX, Latch	3
capture-pass register	4
wires	0

Table 9.3: Delay of used building blocks in multiples of the inverter delay

It has to be stated that for the calculations it was assumed, that the TEST values are a lot faster than the DATA values, such that the latter are not slowed down. For that reason the elements, not passed by DATA values, have been neglected in the following considerations.

Bundled Data

When looking at the pipeline structure shown before (9.7) one can see, that the request signal has to pass the delay element Δ to reach the Muller-C element and the acknowledge line of the succeeding pipeline stage a negation. Only after both have reached the Muller-C gate it switches making the latch transparent, which again only works with a certain delay. By adding the delay values for the Muller-C element, the inverter and the latch the initial delay for n pipeline stages sums up to

$$native = n * (2 + 1 + 3) + n * \Delta = 6 * n + n * \Delta$$

When introducing the test approach the latches inside the nodes have to be replaced by capture-pass registers adding one unit delay. In addition the 4-to-2 phase merge (one Muller-C element+ MUX) and 2-to-4 phase split unit (one Muller-C element) have to be considered. Please note that the inverters in both units have been dropped due to the fact that these are solely present for the TEST values and are, as stated above, not considered here. In total the delay for the test implementation sums up to

$$test = native + 7 + n = 7 + 7 * n + n * \Delta$$

The overhead in this case results to

$$\begin{aligned} overhead &= \frac{test - native}{native} \\ &= \frac{n + 7}{6 * n + \Delta * n} \\ &= \frac{1}{6 + \Delta} + \frac{1}{n} * \frac{7}{6 + \Delta} \end{aligned}$$

If no combinational logic is assumed at all the worst case overhead is achieved as

$$overhead = \frac{1}{6} + \frac{1}{n} * \frac{7}{6} = \left(17 + \frac{117}{n}\right)\%$$

As with the area overhead also the additional delay has a high overhead for short pipelines and no combinational logic. However as soon as some of the latter is added the overhead again quickly approaches 0 % as shown in figure 9.11.

Completion Detection

The calculation of the additional delay for circuits using the completion detection approach is somewhat harder due to the fact, as already mentioned when evaluating the area overhead, that no concrete numbers could be found, stating how much a gate changes when it is converted from 4-phase to 2-phase. Therefore the combinational logic is neglected in the following considerations. Just at the end an educated guess is made to estimate the additional delay and the following overhead.

The completion detection signals used to control the capture-pass registers inside one pipeline node are generated by an OR gate followed by a tree of Muller-C gates with a maximum depth of $\lceil log_2(k) \rceil$ with k equal to the number of data lines. Afterwards the signal, as before, has to

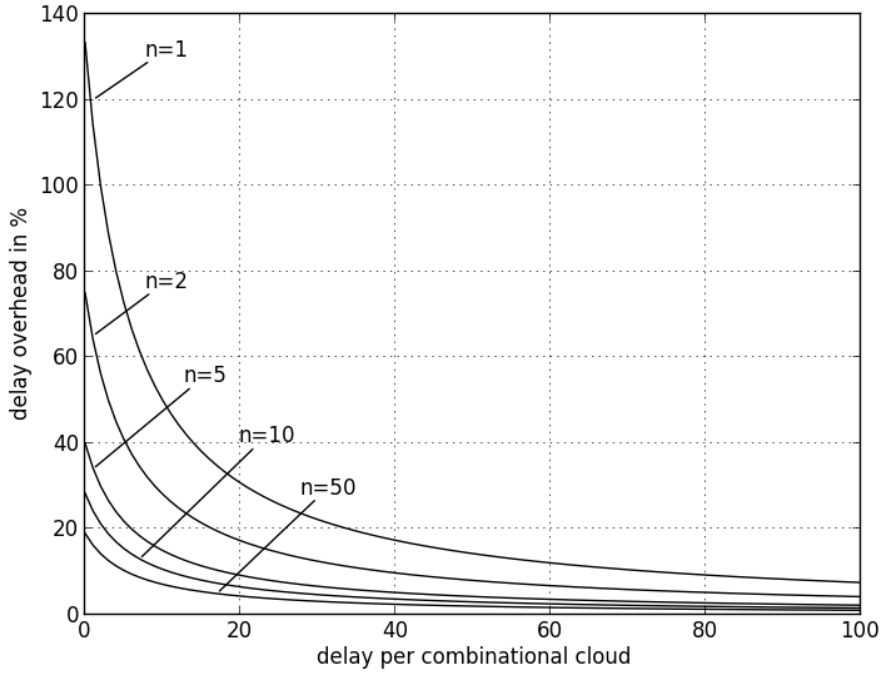


Figure 9.11: delay overhead in % in dependence of delay of single combinational cloud and number of pipeline stages (n), bundled data

pass another Muller-C element with negation until it reaches the storage element which again needs some time to react. The overall delay then sums up to

$$native = n * (2 + 2 * \lceil log_2(k) \rceil + 2 + 1 + 4) = n * (9 + 2 * \lceil log_2(k) \rceil)$$

When adapting the circuit to the test approach the OR gates inside the pipeline nodes have to be replaced by XOR gates, increasing the delay by one unit delay. In the 4-to-2 phase merge unit only the data format conversion part adds delay in the amount of one OR and one Muller-C gate on its longest path. In the split unit at most one AND gate and one Muller-C element have to be passed, whereat again the negation on the test path has been dropped. Overall the delay of the test approach accumulates to

$$test = n + 8 + native = n * (10 + 2 * \lceil log_2(k) \rceil) + 8$$

The overhead can be computed to

$$
\begin{aligned}
overhead &= \frac{test - native}{native} \\
&= \frac{n+8}{n*(9+2*\lceil log_2(k) \rceil)} \\
&= \frac{1}{9+2*\lceil log_2(k) \rceil} + \frac{1}{n} * \frac{8}{9+2*\lceil log_2(k) \rceil}
\end{aligned}
$$

which is initially already very low (42.9 % for $k = 64$ and $n = 1$) and further drops with increasing pipeline length. The reasons are the already very high delay of the native implementation and, of course, that the combinational logic was neglected. As mentioned earlier an educated guess had to be made for the introduced delay when converting the logic, which was set to plus 100 %. As before this value is believed to be a very pessimistic one so actual implementations might work out far better than calculated here. By adding the logic in the form of delay Δ per pipeline stage the overhead results to

$$
\begin{aligned}
overhead &= \frac{test + (1+100\%) * logic - (native + logic)}{native + logic} \\
&= \frac{n+8+\Delta*n}{n*(9+2*\lceil log_2(k) \rceil + \Delta)} \\
&= \frac{1+\Delta}{9+2*\lceil log_2(k) \rceil + \Delta} + \frac{1}{n} * \frac{8}{9+2*\lceil log_2(k) \rceil + \Delta}
\end{aligned}
$$

A graphical illustration of the results is shown in figure 9.12. It can be seen that an increased amount of pipeline stages also reduces the overall overhead. However when adding combinational logic it quickly starts to dominate and the overhead approaches the boundary value of 100 %.

Summary

The results for bundled data and completion detection are summarised in table 9.4. As before the first four rows show the delay for the original (column $native$) implementation and the additional delay when the test approach is included (column $test$). The last two rows state analytical formulae once with and once without considering combinational logic. The variable Δ represents the average delay of the logic per stage and k the amount of data lines.

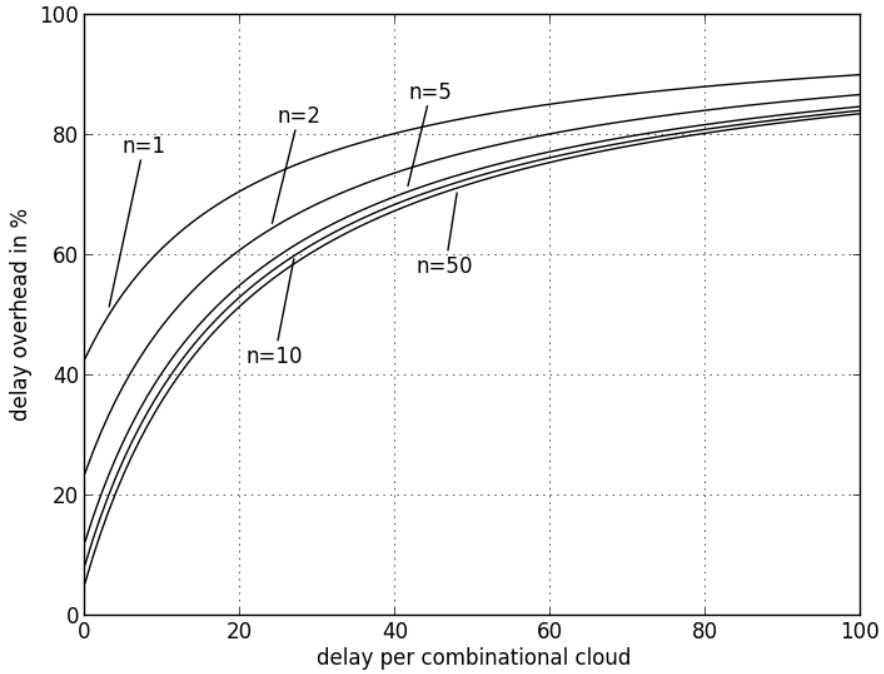


Figure 9.12: delay overhead in % in dependence of delay of single combinational cloud and number of pipeline stages (n), $k = 64$, completion detection

	BD		**CD**	
	native	**test**	**native**	**test**
merge unit	-	5	-	4
combinational	-	-	-	$\approx$*$1.5 - 2$
pipeline node	6	1	$9 + 2 * \lceil log_2(k) \rceil$	1
split unit	-	2	-	4
n stages (%) without comb.	$17 + \frac{117}{n}$		$100 * \left(\frac{n+8}{n*(9+2*\lceil log_2(k) \rceil)} \right)$	
n stages (%) with comb.	$100 * \left(\frac{1}{6+\Delta} + \frac{1}{n} * \frac{7}{6+\Delta} \right)$		$100 * \left(\frac{n+n*\Delta+8}{n*(9+2*\lceil log_2(k) \rceil+\Delta)} \right)$	

Table 9.4: delay overhead estimated by unit delay, Δ represents combinational logic delay per stage, k stands for the amount of data lines

CHAPTER 10

Critical Reflection

In this chapter the proposed test approach is investigated thoroughly to determine its advantages and disadvantages. For that purpose at first some general considerations are carried out and later several characteristic values are computed, to determine the achieved test properties.

10.1 Analysis

The proposed test approach is, as the title of this thesis indicates, truly CBIST, meaning that the test circuit checks a CUT while it is working without interrupting its normal operation at any point in time. This means that it is not necessary to switch to a specific test mode even once, like it may happen when using for example the input vector monitoring approach. Furthermore the test procedure is carried out completely transparent to the outside world i.e. that an outside observer is not able to tell if testing is carried out or not by just looking at the data interfaces. With a little extra effort the proposed test approach can also be used as off-line test method e.g. for initial testing after fabrication. The only thing that has to be done is to feed dummy DATA values into the CUT, due to the fact that testing is only carried out after user data have been processed.

To test a circuit using the proposed approach several additional components, with 4-to-2 phase merge and 2-to-4 phase split leading the way, have to be implemented which implies a higher area requirement compared to the original circuit. The actual implementation size of these units largely depends on the chosen implementation style, which may differ between the two units. It is possible, for example, to implement the 4-to-2 phase merge unit using the Basic Implementation and the 2-to-4 phase split one using the Early NULL-phase. Such configurations are useful, if the properties of the interfaces differ, requiring different implementations to enhance the working speed at input and output. Furthermore the TVG and TRA have to be implemented, whose individual size again depends on the implementation method. The smallest

results for large test sets will probably deliver a Linear Feedback Shift Register (LFSR) or an automaton, for small sets a memory holding the actual values may be an alternative.

When applying the test approach to an existing circuit, utilising the 4-phase communication style, the circuit itself has to be transformed first. When using bundled data a conversion of the pipeline nodes to support the 2-phase protocol has to be carried out as well as a conversion of asymmetric delay lines to symmetric ones, however the logic can be left untouched. When completion detection is used only the logic gates have to be adapted to the new data format because the pipeline controllers are the same for both communication protocol types. Unfortunately no concrete comparison between 2-phase and 4-phase implementations of logic gates could be found in the literature so the overhead can not be determined exactly. Overall it has to be noted that the hardware actually being tested is not the same as the one that originally was supposed to be investigated, because of the conversion of single components and the addition of new ones. This may lead to some extra faults in the altered CUT that would not have been possible in the original one, but at least these are recognisable.

A general estimation of the delay introduced by the proposed test approach is, similar to the area overhead, very difficult. At least in theory it does not decrease the overall working speed provided that the Test Data input is much faster than the User Data one, because in that case the UD are propagated as soon as they show up, i.e. at the original speed. In specific situations some implementation styles may even increase the working speed compared to the original pipeline, because delays are compensated. In real world implementations however the speed of the circuit will always be slower than presumed in theory as it can also be seen in the proof-of-concept implementation in section 9. The actual delay again depends on the implementation itself, in detail on the routing of the signals, the working speed of the individual parts, etc.

Area and time overhead also largely depend on whether Single Stage Testing (SST) or Complete CUT Testing (CCUTT) is used. Assume that a CUT having n pipeline stages has to be tested. When using SST n TVGs, TRAs, 4-to-2 phase merge and 2-to-4 phase split units have to be introduced, whereas with CCUTT only one of each is required. Despite the fact that the individual test vector units may get easier to design when using SST, their sum will in general exceed the requirements for the one used with CCUTT. Therefore the usage of SST is not advisable because faster and smaller test approaches exist for bare logic testing, like codes. However it has to be noted, as also mentioned in [4], that pipelines are approaching their worst case performance the longer they get and therefore such structures may not be found in actual implementations.

The conversion of (parts of) the original circuit is one of the most challenging tasks that has to be carried out when implementing the proposed test approach. The conversion from a 4-phase to a 2-phase style does not only result in improved area requirements due to the lower efficiency, as mentioned in section 3.2, but also restricts the amount of testable implementations. As pointed out in chapter 8 some computational circuits can only be tested by such TEST values that lead to a result in the same amount of computation steps as the DATA value currently processed. This however implies that the test vectors are correlated to the input and can not be chosen freely

making it impossible to determine the result a priori. Therefore the proposed test approach may only be used to test computational circuits computing their result in a fixed amount of steps independent of the input value. Cyclic pipelines, in contrast, are fully supported by the proposed test approach. In such cases, i.e. when feed back or feed forward loops are present, it is not possible to predict the output solely based on the actual input but also the internal values have to be considered. Due to the fact that the order of the TEST values is fixed, even in such situations the correct results can be computed a priori, making it possible to test this kind of circuits very well, which is not possible with other approaches. However it has to be noted that the complexity of the TRA unit might rise because in general more reference values have to be stored in that case.

In this thesis LEDR and FSL were chosen as representatives for the completion detection communication protocol. To increase the speed and reduce overhead the developed units were adapted specifically for these protocols. Please note that other data encodings, like *M-of-N* [38], have not been investigated at all. This implies that the validity of the whole test approach has to be shown if other encoding schemes are used. A generic expression for all available schemes could not be derived due to the fact, that the format conversion units are unique for each combination of 4-phase and 2-phase encoding.

The fact that separate TEST values are used, yields the advantage that the test vectors can be arranged by the designer at design time implying that they do not depend in any way on the input data, as it is the case with the input vector monitoring approach. This also means that no changes to explicit test modes are necessary but instead the testing can be carried out completely concurrent. Furthermore the knowledge of the exact order of the test vectors makes it possible to regulate the frequency of signature checking in the TRA for the desired purpose. In detail it is possible to check each output value of course with a corresponding memory effort or to check only once per test cycle. Each possible value in between is also realisable even not equidistant sections are imaginable. More specifically the signature can be compared after the first x values and then after the next y values. This makes it possible to adapt this parameter for the actual implementation depending on the required *EL* and available die space.

The introduction of the separate TEST values however yields the disadvantage that the User Data are not checked themselves. Instead the results of TD validations are used to estimate the correctness of the actual User Data. This circumstance leads to problems when transient fault with a duration shorter than a single data transmission are assumed. Due to the fact that one test vector is sent per user data vector the probability is 50 % that the transient fault hits the test phase, resulting in an error detection despite the fact, that the User Data were correct all the time. In contrast no failure is reported if the fault hits the User Data phase although a wrong value was propagated. A correct functionality only shows up if the fault hits both phases. In these cases the input vector monitoring approach yields some advantages because it compares the actual UD output to precalculated values. However it has to be noted, that in general not all input vectors are in the test set, therefore errors on these values are not detectable at all. Intermittent failures can be handled better with the proposed approach, due to the fact that repeating application

of test vectors for a specific intermittent fault lead to a sufficiently high probability that it is not present, as outlined in [31]. Overall the proposed test approach is best suited to detect permanent faults – which, after all, is the genuine dedication of a test – with very limited capabilities for transient and intermittent ones. It also has to be noted that only faults on the data path are detected, meaning that violations of the communication protocol, current draining gates, etc. are not detectable in a systematic manner and require a separate testing method.

The fact that the correct result of the test vector operations is known a priori yields the big advantage that even several faults altering the output simultaneously are detectable, as long as they do not cancel each other. This leads to an improvement compared to code checking where the data may be altered to form a valid code word which however is not according to the expected result. In that case the code checking approach would not detect an error because only the code property can be checked and not the actual result. Of course this may also happen in the proposed test approach (although with far lower probability), more specific when a compactor is used in the TRA and several failures lead to the expected signature. This could only be avoided for sure if each result is precalculated and checked as it shows up at the output, also yielding a fantastic *LFD* as well as an *EL*. Due to the fact that this would require lots of memory on chip, it is only conceivable for circuits which are testable with very few different test vectors.

As mentioned in section 1 asynchronous circuits only operate when new data are available. The fact that a TEST value is only processed after a DATA value has been, implies, that testing is also just carried out when new data are available. When nothing has to be done for a longer period the circuit halts, rising the possibility for multiple faults. For that reason single fault models may be problematic with asynchronous logic, which therefore were not used in this thesis.

10.2 Circuit Characteristics

In the following some characteristic values of the proposed test approach will be evaluated. A detailed description of each single value can be found in section 3.11.

Latency of Test Completion (LTC): In the proposed test approach the test vectors are applied in a fixed order by the test circuit which is known a priori. Due to the fact that asynchronous logic is used no exact test time can be specified, because testing is only carried out after DATA values have arrived. For that reason the *LTC* is expressed in a number of input DATA values that have to arrive, causing the test circuit to check the complete CUT. This is the case if each modelled fault has been addressed by at least one input test vector, which is assumed in the following as the whole test set. In detail this means, that each test vector is altered by exactly one fault, i.e. that the whole test set is necessary to check the whole CUT yielding

$$LTC = \#test\ vectors = n$$

It can be seen easily that the *LTC* can be optimised by reducing the test set. More important, however, is the fact, that it is constant during operation, which was found problematic with the input vector monitoring approach [47].

Latency of Fault Detection (LFD): The time it takes to detect a fault largely depends on the frequency the signature computed in the TRA is compared to precomputed values. For that reason the value C describes in the following, how often per cycle, i.e. while n test vectors are applied to the input, the signature is checked. It is assumed that each test vector is important to detect a specific fault and that faults appear equally probable at any instant in time, yielding the received *LFD* to represent the worst case performance.

In addition it is assumed that checking is carried out in equal time intervals, splitting the set of n test vectors into C blocks of length $BC = n/C$. This is shown in figure 10.1 for $n = 9$, $C = 3$ and $BC = n/C = 3$, where C_1, C_2 and C_3 mark the spots when the signature at the TRA is checked. In the following p denotes the position of a test vector inside a block starting at 0. Taking the example from the figure for test vector 8 p results to 1, for 6 to 2 and so forth.

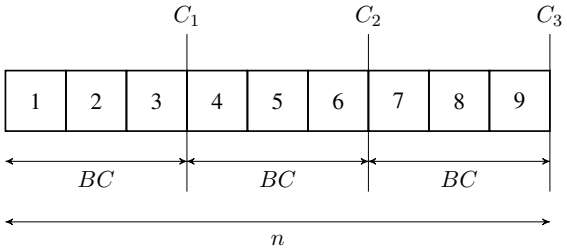


Figure 10.1: test set structure for $n = 9$ and $C = 3$ resulting to $BC = n/C = 3$

The *LFD* for the whole test set can be computed by determining it for a single block because it was assumed, that each fault is detected by a single test vector. In detail this implies that the test vectors with the same p value have the same fault detection latency, yielding the same *LFD* for each single block. By averaging again the original value is received so calculating the *LFD* of a single block is indeed valid. For more elaborate test sets, i.e. multiple test vectors for frequent faults, this argumentation is not valid any more.

The signature in the TRA is checked always at the end of a block, therefore the *LFD* varies depending on the position of the corresponding test vector within the block. Furthermore it depends on the point in time the fault appears in relation to the moment the test vector, used to detect that particular fault, is applied to the CUT. Figure 10.2 shows the best and worst case for test vector number 2 (test set as shown in figure 10.1). In the first case the fault appears right before the corresponding test vector is applied to the CUT, marked by $t_{f2,BC}$ in the picture. Due to the fault the signature computed in the TRA deviates from the expected one after test vector number 2 has been processed. This is detected the next time the signature is checked, i.e. at point $t_{f2,DBC}$. Expressed mathematically p has to be subtracted from BC to achieve the number of steps it takes to detect the fault. In the shown case this would be $BC - p = 3 - 1 = 2$

steps which is exactly $t_{f2,DBC} - t_{f2,BC}$. In the latter case the fault shows up right after the corresponding test vector ($t_{f2,WC}$). In that case the signature is not altered until test vector 2 is applied to the CUT again, due to the assumption, that only a single vector is altered by a single fault. Therefore the fault is detected at point $t_{f2,DWC}$, i.e. the next time the signature is checked after test vector 2 has been processed again. The number of steps it takes to detect the fault in the worst case thus is achieved by subtracting p from $BC-1$ and adding n for the additional time it takes to reach the next test cycle. Overall the latency in the worst case sums up to $BC-1-p+n$ which is in the toy example $3-1-1+9=10$ and also equal to $t_{f2,DWC} - t_{f2,DBC}$.

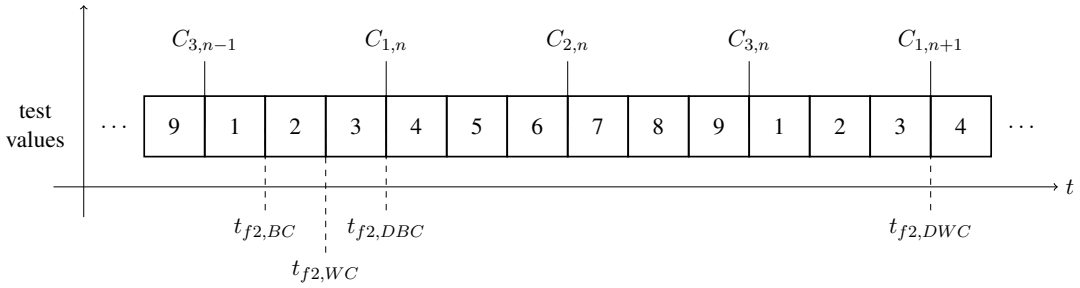


Figure 10.2: detection latency of a fault detected by test vector 2 for best and worst case

Please note that $t_{f2,BC}$ and $t_{f2,WC}$ are in reality not directly at the borders but somewhere in between. The actual position however depends on the location of the faulty node inside the circuit. For simplicity reasons and because the introduced shift is expected to be rather small compared to n these considerations have been neglected in these evaluations.

The latency in the worst case and best case for the first and last element of one block are shown in table 10.1. The average case is achieved by the arithmetic mean, because the values between the best and worst case scale linearly.

position	worst case	best case	average case
first element	$BC-1+n$	BC	$(n-1)/2+BC$
last element	n	1	$(n+1)/2$

Table 10.1: *LFD* for test vectors at first and last position within one block

The average case latencies for the elements within the block can again be achieved by linear interpolation between the first and last element. To achieve an average value for the whole block, i.e. the *LFD*, again only the arithmetic mean has to be created. Therefore the *LFD* results, with the usage of $BC = n/C$, to

$$LFD(n,C) = \frac{\frac{n-1}{2} + \frac{n}{C} + \frac{n+1}{2}}{2} = \frac{n}{2} + \frac{n}{2C}$$

If C is set to one, more specifically that the TRA compares the computed signature only once per cycle, a value of

$$LFD(n, 1) = n$$

is achieved, i.e. in average it takes one whole test cycle to detect a fault. This seems plausible because it takes in average $n/2$ cycles until the appropriate test vector shows up and additional $n/2$ cycles until the signature is checked, resulting also to the above achieved n cycles.

Error Latency (EL): For the calculation of the error latency the latency of fault manifestation (*LFM*), defined as the time it takes a fault to lead to an undesired behaviour at the output with a certain probability, was used. The *LFM* largely depends on the amount of possible input values and their probabilities to appear. In several papers an equal distribution was assumed which however does not describe a real world implementation properly. Nevertheless this assumption is also used in the calculations shown later, to achieve comparable results. Please keep in mind that the calculated value might differ significantly from the one received for an actual implementation.

In contrast to the *LTC* the *LFM* has two arguments, namely the fault f and the probability α that f manifests in an erroneous value at the output. Furthermore let $m(f)$ be the number of differing inputs that are affected by f and N the overall number of inputs. Based on these definitions the probability α, in dependence of the number of applied input vectors denoted as L, can be described as

$$\alpha = 1 - \left(1 - \frac{m(f)}{N}\right)^L$$

The interesting variable for the calculation of the *LFM* is L, i.e. how many input vectors have to be applied that a specific fault results in a failure at the output with a certain probability. By transforming the equation to the form $L = \ldots$ and replacing L by $LFM(f, \alpha)$ the following formula is achieved:

$$LFM(f, \alpha) = \frac{log_e(1 - \alpha)}{log_e\left(1 - \frac{m(f)}{N}\right)}$$

Finally the error latency is received by calculating the difference of the *LFD* , i.e. the time it takes to detect a fault and the *LFM* , i.e. how long it takes until a fault manifests as undesired behaviour. Therefore the *EL*, after plugging in the values for the *LFD* and *LFM*, results to:

$$EL(f,\alpha,n,C) = LFD(n,C) - LFM(f,\alpha) = \frac{n}{2} + \frac{n}{2C} - \frac{log_e(1-\alpha)}{log_e\left(1-\frac{m(f)}{N}\right)}$$

As it can be seen the *EL* can be reduced by decreasing the amount of TEST values that are necessary to test the whole CUT, denoted by variable n, or to check the signature in the TRA more often, i.e. by increasing value C. Another possibility is to reduce the amount of input vectors that are affected by a single fault, i.e. decreasing $m(f)$.

The resulting *EL* can be interpreted as the amount of input vectors that are applied to the input while erroneous results are produced at the output. Due to the characteristics of an asynchronous circuit, i.e. that it just works when data are available, it is not possible to determine the latency characteristics in actual time values. Note that the *EL* can also become negative which represents the case that the fault is detected before it causes a single error at the output.

Table 10.2 shows some calculations for the above mentioned property values, where a 16 bit input was chosen and all input vectors may appear, implying that $N = 2^{16}$. In that calculation the overall test vector count n was set to 2^{12} and C to 1.

$m(f)$	α	LTC = LFD	LFM(f,α)	EL(f,α)
2^4	0.5	2^{12}	2838.78	1257.22
	0.7		4930.87	−834.87
	0.9		9430.24	−5334.24
2^6	0.5	2^{12}	709.44	3386.56
	0.7		1232.27	2864.73
	0.9		2356.70	1738.30
2^8	0.5	2^{12}	177.10	3918.90
	0.7		307.62	3788.39
	0.9		588.31	3507.69
2^{10}	0.5	2^{12}	44.01	4051.99
	0.7		76.45	4019.55
	0.9		146.21	3949.79

Table 10.2: Calculation results for *LFD*, *LFM*, *LTC* and *EL* for $N = 2^{16}$ and $n = 2^{12}$.

Notice that the obtained values of LTC, LFD and EL are excellent in comparison with other test approaches, since a new test vector can be applied with every data word. In the Concurrent Checking approach one has to wait until the input data stream happens to produce the complete desired set of test vectors, while in the case of a switching between test and operational mode, one has to find a trade-off between the duty cycle among those. In both cases one will typically end up with much worse characteristics.

CHAPTER 11

Conclusion

In this thesis a novel test approach, especially designed for asynchronous circuits, has been presented. It exploits the unique properties of handshake protocols to process test data in parallel to the normal operation without interrupting it at any point in time. More specifically the rather unproductive NULL-phase of a 4-phase protocol is replaced by user defined TEST values and then fed into the circuit under test. In that way the tightest possible interleaving (every other value is a TEST value) as well as complete independence between user and test data is achieved. This makes it possible to fully determine the TEST values as well as the expected results not only in the value but also in the time domain at design time. Please note, that in contrast to similar approaches also stated with the title BIST, the test approach proposed in this thesis actively applies test vectors to the inputs of the CUT .

In addition implementation methods have been developed, i.e. how a given circuit has to be altered to be tested using the proposed test approach. During this process additional necessary units have been identified and designed from scratch due to missing references in literature. Several implementations have been achieved differing in their level of complexity and concurrency, for the bundled data as well as the completion detection communication style. At last it was investigated what criteria a CUT has to fulfil to be testable, which turned out to be rather relaxed, because even cyclic pipelines are allowed.

Finally a proof-of-concept implementation, developed on an FPGA, demonstrated the correct functionality of the proposed test approach. It was shown that the newly designed units work indeed as specified and the test approach detects faults reliably. The introduced overheads have been analysed analytically and turned out to be rather moderate. In the case of bundled data in the worst case an overhead of around 200 % is achieved which however quickly approaches 0 % when logic is added to the initially empty pipeline. For completion detection lower values are received in the beginning (around 30 %) which however approaches the boundary value of 100 % with additional logic. The reason for that increase is, that the overhead for converting com-

pletion detection gates from 4-phase to 2-phase was estimated with plus 100 % due to missing numbers in literature.

The additional delay introduced by the test approach also turned out to be very moderate. In the case of bundled data a value of 134 % for short pipelines and without considering combinational logic is achieved. This value however quickly converges to 0 % when logic is added. Unfortunately completion detection circuits do not show this favourable behaviour due to the increase of logic when converting it from 4-phase to 2-phase. Due to missing numbers in literature an educated an very pessimistic guess of an increase of 100 % was chosen. The very low overhead for pipelines without combinational logic is quickly dominated by the logic, raising the overhead to the boundary value of 100 %. Please note that this estimation, as well as the one for the additional area, was chosen pessimistically and real values may be way below, which however is a topic for future research.

One general conclusion from the implementation and analyses is that the approach is well suited for bundled data circuits, where the overheads in terms of area and speed scale quite nicely and amount to reasonable values for typical parameters (pipeline stages, logic gates per stage), and where testing is essential. Completion detection circuits, in contrast, exhibit a worse scaling behaviour and are hence less suited for the proposed approach. In addition, their inherent protection by coding makes them hardly prone to undetected permanent faults.

The analysis of characteristic values for the test approach furthermore revealed very favourable properties. First of all, the duration required to test the whole CUT is constant all of the time. In addition it is achievable to detect faults in average before they result in an undesired behaviour at the output. Another big advantage is, that the characteristic values can, to a certain degree, be tweaked by the designer to allow for example faster fault detection.

In this thesis only the basic structure of the new test approach was described, leaving plenty of work for future research. Still missing is an investigation under which conditions the proposed implementations for the 4-to-2 phase merge and 2-to-4 phase split unit indeed increase the speed and by what factor compared to e.g. the Basic Implementation. Another interesting topic might be the development of even faster and maybe even smaller implementations than the presented ones as well as the creation of a completely DI design, or a proof that it is impossible at all. It is also possible to extend the approach to other completion detection communication protocols like *M-of-N* or to show that these are not supported.

The key results of this thesis were accepted at the *17th IEEE Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)* and the submission received the best paper award. The corresponding paper [14] can be found in Appendix C.

At last the following list summarises advantages, disadvantages and unique properties of the proposed test approach:

- Data processing and testing are carried out in parallel i.e. it is not necessary to change to a specific test mode which interrupts normal data processing at any point in time.
- The test approach is transparent to the world outside the CUT. In detail a communication partner can not determine if testing is carried by just observing the interfaces.
- Through the replacement of the NULL-phase the protocol type is changed from a 4-phase to a 2-phase one. This implies that the control logic, when using bundled data, and the data logic, when using completion detection, has to be altered, which results in an increased effort and increased costs.
- The presented implementations only work correctly if the early data-validity scheme on a push channel is used.
- Despite the fact that the test approach was designed as online method it can also be used for initial tests in the factory. The only thing that has to be done is to apply proper request and acknowledge signals to the interfaces.
- The approach works well for permanent faults, which are the target of a test. Transient and intermittent faults may be covered, but may as well lead to false positives and false negatives, unless they span at least two computational cycles. This is because the test only checks the integrity of the Test Data, while User Data are passed through essentially without protection.
- Due to the fact that the correct result of each test vector is known the value of the output can be checked directly, not just if it justifies some properties as it is the case when using coding. This makes it possible to even detect multiple failures as soon as they do not cancel each other.
- For the purpose of the proposed approach, the original target circuit needs to be modified and extended. As a consequence, it may exhibit different and additional faults compared to the original.
- It is only possible to detect faults on the data path, all others, for example on the control lines, are not detectable at all.
- Based on the implementation the detection of an error has more or less delay. This largely depends on the frequency the signature generated by the test vector analyser is compared to the precomputed values. More frequent checking decreases the response time however increases the hardware requirements, because more reference values have to be stored. Here a good compromise between the *LFD* and the area overhead has to be found by the designer.
- The test vectors in this approach can be chosen freely, making the testing independent of the actual User Data that is applied to it. The determination of an optimal test set, i.e. that is as small as possible but is capable of detecting any modelled fault, however is a very challenging task. Furthermore the size of the test set directly influences the *LTC*, which is

in this approach constant and equal to the size of the test set. This of course implies that the CUT can be checked more often in a given time interval when the set gets smaller.

- In the test approach it is not defined what has to be done when the behaviour of the CUT is not according to the expected one, meaning that the error handling as well as possible recovery steps have to be carried out by the outer application. Several methods are possible, which again depend on the functionality of the CUT as well as on the criticality of the results provided by that unit.

- Due to the fact that asynchronous circuits are used, computations might not be carried out on a regular basis as with synchronous logic. This implies that also testing is not carried out regularly because a test vector is just processed after a User Data one had been, meaning that if no User Data are delivered for a period of time also testing is not taking place for the same amount. Therefore Mean Time Between Upset (MTBU) calculations are harder to carry out than in synchronous designs because it may happen that the period between two test cycles is very long, making the single fault assumption unreasonable. This may lead to problems because most methods, for example the coding approach, might have problems with multiple faults.

- In certain cases it is possible to achieve a negative *EL*, meaning that the fault is detected by the test circuit before it manifests in a single failure within the User Data. This however requires a high test effort or only a little amount of input vectors that are altered by a fault.

- It is possible to test cyclic pipeline structures, in detail feed forward and feed back loops, because the test values that are interacting are known at design time. When using these structures it may happen that errors cycle within the CUT, periodically leading to failures. For that reason it is necessary to clear the whole CUT after an error has been detected, for example by resetting the circuit, or to prove, that it is impossible for an error to cycle.

- Unfortunately not every circuit structure is testable by the proposed test approach, more specific computational circuits whose number of computation steps depend on the input data. Furthermore circuits with internal storage like registers or state machines have to be altered to become testable.

APPENDIX A

Glossary

BD Bundled Data

CBIST Concurrent Built-In Self-Testing

CD Completion Detection

CCUTT Complete CUT Testing

CUT Circuit Under Test

DI Delay Insensitive

DL Data Line

EL Error Latency

FC Format Conversion

LEDR Level Encoded Dual Rail

LFD Latency of Fault Detection

LFM Latency of Fault Manifestation

LTC Latency of Test Completion

NCL Null Convention Logic

NRZ No Return to Zero

PhM Phase Merge

PhS Phase Split

QDI Quasi Delay Insensitive

RTZ Return To Zero

STG State Transition Graph

SST Single Stage Testing

T Transitors

TD Test Data

TRA Test Response Analyser

TVG Test Vector Generator

UD User Data

UTD User/Test Data

APPENDIX B

Code

B.1 Petrify Library

```
1  # ---- COMBINATIONAL GATES
2
3  GATE   "inv:combinational"    16   O=!A;
4  PIN * INV 1 999 1 .2 1 .2
5
6
7  GATE   "and2:combinational"     32   O=A*B;
8  PIN * NONINV 1 999 1 .2 1 .2
9
10 GATE   "and2_1:combinational"     32   O=A*!B;
11 PIN * NONINV 1 999 1 .2 1 .2
12
13 GATE     "nand2:combinational"            24       O=!(A*B);
14 PIN      * INV 1 999 1 .2 1 .2
15
16 GATE     "nand2_1:combinational"          24       O=!(A*!B);
17 PIN      * INV 1 999 1 .2 1 .2
18
19
20
21
22 GATE   "and3:combinational"     40   O=A*B*C;
23 PIN * NONINV 1 999 1 .2 1 .2
24
25 GATE     "nand3:combinational"            32       O=!(A*B*C);
26 PIN      * INV 1 999 1 .2 1 .2
27
28 GATE   "and4:combinational"     48   O=A*B*C*D;
29 PIN * NONINV 1 999 1 .2 1 .2
30
31 GATE     "nand4:combinational"            40       O=!(A*B*C*D);
```

```
32 PIN       * INV 1 999 1 .2 1 .2
33
34
35 GATE  "or2:combinational"    32  O=A+B;
36 PIN * NONINV 1 999 1 .2 1 .2
37
38 GATE  "or2_1:combinational"    32  O=A+!B;
39 PIN * NONINV 1 999 1 .2 1 .2
40
41 GATE     "nor2:combinational"              24        O=!(A+B);
42 PIN       * INV 1 999 1 .2 1 .2
43
44 GATE  "or3:combinational"    40  O=A+B+C;
45 PIN * NONINV 1 999 1 .2 1 .2
46
47 GATE     "nor3:combinational"              32        O=!(A+B+C);
48 PIN       * INV 1 999 1 .2 1 .2
49
50 GATE  "or4:combinational"    48  O=A+B+C+D;
51 PIN * NONINV 1 999 1 .2 1 .2
52
53 GATE     "nor4:combinational"              40        O=!(A+B+C+D);
54 PIN       * INV 1 999 1 .2 1 .2
55
56
57 GATE  "aoi22:combinational"    40  O=!(A*B+C*D);
58 PIN * INV 1 999 1 .2 1 .2
59
60 GATE  "aoi12:combinational"    32  O=!(A+B*C);
61 PIN * INV 1 999 1 .2 1 .2
62
63
64 GATE  "oai22:combinational"    40  O=!((A+B)*(C+D));
65 PIN * INV 1 999 1 .2 1 .2
66
67 GATE  "oai12:combinational"    32  O=!(A*(B+C));
68 PIN * INV 1 999 1 .2 1 .2
69
70
71 GATE  "ao22:combinational"     56  O=A*B+C*D;
72 PIN * NONINV 1 999 1 .2 1 .2
73
74 # The following functions are not used. They are binate and
75 # there is no guarantee of being hazard-free. Use them at
76 # your own risk
77
78 #GATE "xor:combinational"    40  O=!(A*B+!A*!B);
79 #PIN   * UNKNOWN 1 999 1 .2 1 .2
80
81 #GATE "xorbar:combinational"     48  O=A*B+!A*!B;
82 #PIN   * UNKNOWN 1 999 1 .2 1 .2
83
84 #GATE "mux2:combinational"     48  O=D1*SEL+D2*!SEL;
```

```
85 #PIN   D1 NONINV 1 999 1 .2 1 .2
86 #PIN   D2 NONINV 1 999 1 .2 1 .2
87 #PIN   SEL UNKNOWN 1 999 1 .2 1 .2
88
89 GATE   "const1:combinational"     8 O=CONST1;
90 GATE   "const0:combinational"     8 O=CONST0;
91
92 # ---- ASYNCH LATCHES
93
94 # Pure delay
95 LATCH    "delay:asynch"  0    Q=D;
96 PIN D    NONINV 1 999 0.00001 0.00001 0.00001 0.00001
97 SEQ Q ANY ASYNCH
98
99 # Inverter
100 LATCH    "delay_inv:asynch"  16    Q=!D;
101 PIN D    NONINV 1 999 0.00001 0.00001 0.00001 0.00001
102 SEQ Q ANY ASYNCH
103
104 # Cross-coupled NAND (SR latch)
105 LATCH "sr_nand:asynch"  40  Q=!S+R*Q_NEXT;
106 PIN S INV 1 999 1 .2 1 .2
107 PIN R NONINV 1 999 1 .2 1 .2
108 SEQ Q Q_NEXT ASYNCH
109
110 # Cross-coupled NOR (SR latch)
111 LATCH "sr_nor:asynch"   40  Q=S+!R*Q_NEXT;
112 PIN S NONINV 1 999 1 .2 1 .2
113 PIN R INV 1 999 1 .2 1 .2
114 SEQ Q Q_NEXT ASYNCH
115
116 # Cross-coupled NAND (RS latch)
117 LATCH "rs_nand:asynch"  40  Q=!R+S*Q_NEXT;
118 PIN S INV 1 999 1 .2 1 .2
119 PIN R NONINV 1 999 1 .2 1 .2
120 SEQ Q Q_NEXT ASYNCH
121
122 # Cross-coupled NOR (RS latch)
123 LATCH "rs_nor:asynch"   40  Q=!R*Q_NEXT+!R*S;
124 PIN S NONINV 1 999 1 .2 1 .2
125 PIN R INV 1 999 1 .2 1 .2
126 SEQ Q Q_NEXT ASYNCH
127
128
129 # C-element
130 LATCH "c_element1:asynch" 40  C = !A*B+(!A+B)*C_NEXT;
131 PIN A NONINV 1 999 1 .2 1 .2
132 PIN B NONINV 1 999 1 .2 1 .2
133 SEQ C C_NEXT ASYNCH
134
135 # C-element
136 LATCH "c_element0:asynch" 40  C = A*B+(A+B)*C_NEXT;
137 PIN A NONINV 1 999 1 .2 1 .2
```

```
138 PIN B NONINV 1 999 1 .2 1 .2
139 SEQ C C_NEXT ASYNCH
140
141 # C-element
142 LATCH "c_element2:asynch" 40  C = !A*!B+(!A+!B)*C_NEXT;
143 PIN A NONINV 1 999 1 .2 1 .2
144 PIN B NONINV 1 999 1 .2 1 .2
145 SEQ C C_NEXT ASYNCH
146
147
148 # Gated Latch
149 LATCH "gated_latch0:asynch" 40  Q=D*G+Q_NEXT*(!G+D);
150 PIN D NONINV 1 999 1 .2 1 .2
151 PIN G UNKNOWN 1 999 1 .2 1 .2
152 SEQ Q Q_NEXT ASYNCH
153
154 # Gated Latch
155 LATCH "gated_latch1:asynch" 40  Q=D*!G+Q_NEXT*G;
156 PIN D NONINV 1 999 1 .2 1 .2
157 PIN G UNKNOWN 1 999 1 .2 1 .2
158 SEQ Q Q_NEXT ASYNCH
```

Listing B.1: Petrify Library

B.2 Petrify Netlists

```
module FourToTwoPhase_net (
    rU,
    rT,
    aUT,
    aU,
    aT,
    rUT
);

input rU;
input rT;
input aUT;

output aU;
output aT;
output rUT;


// Functions mapped into library gates:
// ----------------------------------------

nand2_1:combinational _U0 (.A(csc0), .B(aUT), .O(aU));
nand2_1:combinational _U1 (.A(aUT), .B(csc0), .O(aT));
oai12:combinational _U2 (.A(csc0), .B(rU), .C(aUT), .O(_2_));
// This inverter should have a short delay
inv:combinational _U3 (.A(aUT), .O(_3_));
oai12:combinational _U4 (.A(_2_), .B(rT), .C(_3_), .O(rUT));
// This inverter should have a short delay
inv:combinational _U5 (.A(csc0), .O(_5_));
aoi12:combinational _U6 (.A(aUT), .B(_5_), .C(rU), .O(_6_));
aoi12:combinational _U7 (.A(_6_), .B(rT), .C(csc0), .O(_7_));
inv:combinational _U8 (.A(_7_), .O(csc0));

// signal values at the initial state:
//     !rU !rT !aUT !aU aT _2_ _3_ !rUT !_5_ _6_ !_7_ csc0
endmodule
```

Listing B.2: Petrify netlist, Early DATA-phase, bundled data, 4-to-2 phase merge

```
module FourToTwoPhase_net (
    rU,
    rT,
    aUT,
    aU,
    aT,
    rUT,
    enL
);

input rU;
input rT;
input aUT;
```

```
14
15 output aU;
16 output aT;
17 output rUT;
18 output enL;
19
20
21 // Functions mapped into library gates:
22 // ----------------------------------------
23
24 buf _U0 (aU,rUT);
25 // This inverter should have a short delay
26 inv:combinational _U1 (.A(rUT), .O(aT));
27 // This inverter should have a short delay
28 inv:combinational _U2 (.A(rU), .O(_1_));
29 nand4:combinational _U3 (.A(enL), .B(aUT), .C(rT), .D(_1_), .O(_2_));
30 // This inverter should have a short delay
31 inv:combinational _U4 (.A(rU), .O(_3_));
32 // This inverter should have a short delay
33 inv:combinational _U5 (.A(enL), .O(_4_));
34 nor4:combinational _U6 (.A(aUT), .B(rT), .C(_3_), .D(_4_), .O(_5_));
35 c_element0:asynch _U7 (.A(_2_), .B(_5_), .C(rUT));
36 nor2:combinational _U8 (.A(aUT), .B(rUT), .O(_7_));
37 aoi12:combinational _U9 (.A(_7_), .B(aUT), .C(rUT), .O(_8_));
38 inv:combinational _U10 (.A(_8_), .O(enL));
39
40 // signal values at the initial state:
41 //     !aU !rU !rT !aUT aT _1_ _2_ _3_ !_4_ !_5_ !rUT _7_ !_8_ enL
42 endmodule
```

Listing B.3: Petrify netlist, Additional Latch, bundled data, 4-to-2 phase merge

```
1 module FourToTwoPhase_net (
2     rU,
3     rT,
4     aUT,
5     aU,
6     aT,
7     rUT
8 );
9
10 input rU;
11 input rT;
12 input aUT;
13
14 output aU;
15 output aT;
16 output rUT;
17
18
19 // Functions mapped into library gates:
20 // ----------------------------------------
21
22 or2:combinational _U0 (.A(rUT), .B(csc0), .O(aU));
```

```
23 nand2:combinational _U1 (.A(rUT), .B(csc0), .O(aT));
24 nand2:combinational _U2 (.A(rT), .B(csc0), .O(_2_));
25 // This inverter should have a short delay
26 inv:combinational _U3 (.A(csc0), .O(_3_));
27 aoi22:combinational _U4 (.A(_3_), .B(rU), .C(_2_), .D(rUT), .O(_4_));
28 inv:combinational _U5 (.A(_4_), .O(rUT));
29 // This inverter should have a short delay
30 inv:combinational _U6 (.A(rT), .O(_6_));
31 oai12:combinational _U7 (.A(aUT), .B(csc0), .C(_6_), .O(_7_));
32 sr_nand:asynch _U8 (.S(_7_), .R(rU), .Q(csc0));
33
34 // signal values at the initial state:
35 //     !rU !rT !aUT !aU aT _2_ _3_ _4_ !rUT _6_ _7_ !csc0
36 endmodule
```

Listing B.4: Petrify netlist, Latches at Input, bundled data, 4-to-2 phase merge

```
1  module FourToTwoPhase_net (
2      rU,
3      rT,
4      aUT,
5      aU,
6      aT,
7      rUT
8  );
9
10 input rU;
11 input rT;
12 input aUT;
13
14 output aU;
15 output aT;
16 output rUT;
17
18
19 // Functions mapped into library gates:
20 // ----------------------------------------
21
22 and2:combinational _U0 (.A(csc0), .B(csc1), .O(aU));
23 // This inverter should have a short delay
24 inv:combinational _U1 (.A(rT), .O(_1_));
25 // This inverter should have a short delay
26 inv:combinational _U2 (.A(aT), .O(_2_));
27 oai22:combinational _U3 (.A(csc2), .B(csc3), .C(_1_), .D(_2_), .O(aT));
28 // This inverter should have a short delay
29 inv:combinational _U4 (.A(csc2), .O(_4_));
30 nand4:combinational _U5 (.A(_4_), .B(rT), .C(csc1), .D(csc3), .O(_5_));
31 // This inverter should have a short delay
32 inv:combinational _U6 (.A(csc1), .O(_6_));
33 aoi22:combinational _U7 (.A(_6_), .B(csc0), .C(_5_), .D(aUT), .O(_7_));
34 inv:combinational _U8 (.A(_7_), .O(rUT));
35 nor2:combinational _U9 (.A(csc1), .B(csc3), .O(_9_));
36 c_element0:asynch _U10 (.A(rU), .B(_9_), .C(csc0));
37 aoi12:combinational _U11 (.A(aUT), .B(csc0), .C(csc1), .O(_11_));
```

```
38 inv:combinational _U12 (.A(_11_), .O(csc1));
39 sr_nor:asynch _U13 (.S(aT), .R(csc3), .Q(csc2));
40 nand2_1:combinational _U14 (.A(csc2), .B(aT), .O(_14_));
41 c_element1:asynch _U15 (.A(_14_), .B(aUT), .C(csc3));
42
43 // signal values at the initial state:
44 //     !rU rT !aUT !aU !_1_ _2_ !aT !_4_ _5_ _6_ _7_ !rUT _9_ !csc0 _11_ !
       csc1 csc2 !_14_ !csc3
45 endmodule
```

Listing B.5: Petrify netlist, Decoupled Controller, bundled data, 4-to-2 phase merge

```
 1 module FourToTwoPhase_net (
 2     rU,
 3     rT,
 4     aUT,
 5     aU,
 6     aT,
 7     rUT
 8 );
 9
10 input rU;
11 input rT;
12 input aUT;
13
14 output aU;
15 output aT;
16 output rUT;
17
18
19 // Functions mapped into library gates:
20 // ----------------------------------------
21
22 oai12:combinational _U0 (.A(csc0), .B(rUT), .C(csc2), .O(_0_));
23 inv:combinational _U1 (.A(_0_), .O(aU));
24 // This inverter should have a short delay
25 inv:combinational _U2 (.A(csc1), .O(_2_));
26 aoi12:combinational _U3 (.A(_2_), .B(rUT), .C(csc2), .O(aT));
27 nand3:combinational _U4 (.A(aUT), .B(csc2), .C(csc1), .O(_4_));
28 // This inverter should have a short delay
29 inv:combinational _U5 (.A(csc2), .O(_5_));
30 aoi22:combinational _U6 (.A(_5_), .B(csc0), .C(_4_), .D(rUT), .O(_6_));
31 inv:combinational _U7 (.A(_6_), .O(rUT));
32 nor3:combinational _U8 (.A(aUT), .B(rUT), .C(csc2), .O(_8_));
33 c_element0:asynch _U9 (.A(rU), .B(_8_), .C(csc0));
34 nand2:combinational _U10 (.A(rUT), .B(csc2), .O(_10_));
35 c_element1:asynch _U11 (.A(_10_), .B(rT), .C(csc1));
36 // This inverter should have a short delay
37 inv:combinational _U12 (.A(csc1), .O(_12_));
38 oai12:combinational _U13 (.A(rUT), .B(_12_), .C(csc2), .O(_13_));
39 sr_nand:asynch _U14 (.S(_13_), .R(csc0), .Q(csc2));
40
41 // signal values at the initial state:
```

```
42 //      !rU rT !aUT _0_ !aU _2_ !aT _4_ _5_ _6_ !rUT _8_ !csc0 _10_ !csc1 _12_
       _13_ !csc2
43 endmodule
```

Listing B.6: Petrify netlist, Decoupled Controller with latches, bundled data, 4-to-2 phase merge

```
1  module TwoToFourPhase_net (
2      rUT,
3      aT,
4      aU,
5      rT,
6      rU,
7      aUT
8  );
9
10 input rUT;
11 input aT;
12 input aU;
13
14 output rT;
15 output rU;
16 output aUT;
17
18
19 // Functions mapped into library gates:
20 // ----------------------------------------
21
22 nor2:combinational _U0 (.A(rUT), .B(csc0), .O(rT));
23 and2:combinational _U1 (.A(rUT), .B(csc0), .O(rU));
24 // This inverter should have a short delay
25 inv:combinational _U2 (.A(rUT), .O(_2_));
26 oai12:combinational _U3 (.A(csc0), .B(aU), .C(_2_), .O(_3_));
27 oai12:combinational _U4 (.A(_3_), .B(aT), .C(rUT), .O(aUT));
28 // This inverter should have a short delay
29 inv:combinational _U5 (.A(aU), .O(_5_));
30 oai12:combinational _U6 (.A(rUT), .B(_5_), .C(csc0), .O(_6_));
31 sr_nand:asynch _U7 (.S(_6_), .R(aT), .Q(csc0));
32
33 // signal values at the initial state:
34 //      !rUT aT !aU rT !rU _2_ _3_ !aUT _5_ _6_ !csc0
35 endmodule
```

Listing B.7: Petrify netlist, Early DATA-phase, bundled data, 2-to-4 phase split

```
1  module FourToTwoPhase_net (
2      rUT,
3      aT,
4      aU,
5      aUT,
6      rT,
7      rU,
8      enL
9  );
10
```

```
11 input rUT;
12 input aT;
13 input aU;
14
15 output aUT;
16 output rT;
17 output rU;
18 output enL;
19
20
21 // Functions mapped into library gates:
22 // ----------------------------------------
23
24 // This inverter should have a short delay
25 inv:combinational _U0 (.A(enL), .O(_0_));
26 // This inverter should have a short delay
27 inv:combinational _U1 (.A(aU), .O(_1_));
28 nor4:combinational _U2 (.A(_0_), .B(_1_), .C(rUT), .D(aT), .O(_2_));
29 // This inverter should have a short delay
30 inv:combinational _U3 (.A(aU), .O(_3_));
31 nand4:combinational _U4 (.A(_3_), .B(rUT), .C(aT), .D(enL), .O(_4_));
32 c_element2:asynch _U5 (.A(_2_), .B(_4_), .C(aUT));
33 // This inverter should have a short delay
34 inv:combinational _U6 (.A(aT), .O(_6_));
35 aoi12:combinational _U7 (.A(aUT), .B(enL), .C(_6_), .O(rT));
36 // This inverter should have a short delay
37 inv:combinational _U8 (.A(enL), .O(_8_));
38 oai12:combinational _U9 (.A(aUT), .B(aU), .C(_8_), .O(_9_));
39 inv:combinational _U10 (.A(_9_), .O(rU));
40 // This inverter should have a short delay
41 inv:combinational _U11 (.A(aUT), .O(_11_));
42 aoi22:combinational _U12 (.A(_11_), .B(aT), .C(aUT), .D(aU), .O(_12_));
43 inv:combinational _U13 (.A(_12_), .O(enL));
44
45 // signal values at the initial state:
46 //     !rUT aT !aU !_0_ _1_ !_2_ _3_ _4_ !aUT !_6_ rT !_8_ _9_ !rU _11_ !_12_
       enL
47 endmodule
```

Listing B.8: Petrify netlist, Additional Latch, bundled data, 2-to-4 phase split

```
1 module FourToTwoPhase_net (
2     aUT,
3     pU,
4     pT,
5     aU,
6     aT
7 );
8
9 input aUT;
10 input pU;
11 input pT;
12
13 output aU;
```

```
14 output aT;
15
16
17 // Functions mapped into library gates:
18 // ----------------------------------------
19
20 or2:combinational _U0 (.A(aUT), .B(csc0), .O(aU));
21 nand2:combinational _U1 (.A(aUT), .B(csc0), .O(aT));
22 inv:combinational _U2 (.A(pT), .O(_2_));
23 oai12:combinational _U3 (.A(aUT), .B(_2_), .C(csc0), .O(_3_));
24 sr_nand:asynch _U4 (.S(_3_), .R(pU), .Q(csc0));
25
26 // signal values at the initial state:
27 //      !aUT !pU pT !aU aT !_2_ _3_ !csc0
28 endmodule
```

Listing B.9: Petrify netlist, Early DATA-phase, completion detection, 4-to-2 phase merge

```
1 module TwoToFourPhase_net (
2     pUT,
3     aT,
4     aU,
5     aUT,
6     enU,
7     enT
8 );
9
10 input pUT;
11 input aT;
12 input aU;
13
14 output aUT;
15 output enU;
16 output enT;
17
18
19 // Functions mapped into library gates:
20 // ----------------------------------------
21
22 // This inverter should have a short delay
23 inv:combinational _U0 (.A(aU), .O(_0_));
24 nor2:combinational _U1 (.A(pUT), .B(aT), .O(_1_));
25 // This inverter should have a short delay
26 inv:combinational _U2 (.A(_1_), .O(_2_));
27 aoi22:combinational _U3 (.A(csc0), .B(_2_), .C(pUT), .D(_0_), .O(aUT));
28 and2_1:combinational _U4 (.A(pUT), .B(csc0), .O(enU));
29 and2_1:combinational _U5 (.A(csc0), .B(pUT), .O(enT));
30 // This inverter should have a short delay
31 inv:combinational _U6 (.A(pUT), .O(_6_));
32 oai22:combinational _U7 (.A(_6_), .B(aU), .C(_1_), .D(csc0), .O(_7_));
33 inv:combinational _U8 (.A(_7_), .O(csc0));
34
35 // signal values at the initial state:
36 //      !pUT aT !aU _0_ !_1_ _2_ !aUT !enU enT _6_ !_7_ csc0
```

```
37 endmodule
```

Listing B.10: Petrify netlist, Early DATA-phase, completion detection, 2-to-4 phase split

APPENDIX C

Paper

In this section the paper submitted to the *17th IEEE Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, which was awarded with the best paper award, is presented. It is also already available for download from the *IEEE* database.

Online Test Vector Insertion: A Concurrent Built-In Self-Testing (CBIST) Approach for Asynchronous Logic

Jürgen Maier and Andreas Steininger
Institute of Computer Engineering
Vienna University of Technology
Email: {juergen.maier, andreas.steininger}@tuwien.ac.at

***Abstract*—Complementing concurrent checking with online testing is crucial for preventing fault accumulation in fault-tolerant systems with long mission times. While implementing a non-intrusive online test is cumbersome in a synchronous environment, this task becomes even more challenging in asynchronous designs. The latter receive increasing attention, mainly due to their elastic timing behaviour; however the issues related with their testing remain a key obstacle for their wide adoption.**

In this paper we present a novel approach for testing of asynchronous circuits that leverages the redundancy present in the conventional 4-phase protocol for implementing a fully transparent and fully concurrent test procedure. The key idea is to use the protocol's unproductive NULL phase for processing test vectors, thus effectively interleaving the incoming 4-phase data stream with a test data stream in a 2-phase fashion. We present implementation templates for the fundamental building blocks required and give a proof-of-concept by an example application that also serves as a platform for evaluating the overheads of our solution which turn out to be moderate.

I. Introduction

Throughout the last decades we have witnessed a tremendous shrinking in the feature sizes of VLSI chips, paired with an increase of complexity. While, without doubt, these trends have been the key to the rapidly increasing performance, they also cause an increasing rate of faults per chip. In the face of extremely high transistor counts and small critical charges it is unrealistic to assume that a chip, once tested and put into operation, will perform its operation without further experiencing transient faults or permanent defects. Consequently, fault-tolerance provisions, e.g. based on concurrent checking or replication and masking, have been devised to cope with those faults and defects. However, all these approaches are based on assumptions about the multiplicity of faults – typically the single fault assumption – and they will fail when these are exceeded. While it is often sufficiently improbable that multiple faults coincide, the potential of fault accumulation is sometimes overlooked: A permanent fault that is tolerated within a fault-tolerance concept still uses up its fault-tolerance capacity, thus making the system vulnerable to the next fault that may occur, unless the first fault is properly removed. It is, e.g., well understood that a TMR architecture exhibits lower reliability than a simplex architecture, once one of the replica is affected by a permanent fault. This becomes particularly cumbersome for systems with long mission times. Therefore it is crucial, in addition to masking, to detect the existence of a fault, diagnose and remove it. The identification of faults may be non-trivial, especially when faults in rarely used resources must be considered that may remain undetected by concurrent checking approaches for a long time. This is where on-line testing becomes mandatory [1].

978-1-4799-4558-0/14/$31.00

Asynchronous design is receiving increasing attention since it naturally avoids some of the most serious problems currently faced by synchronous designs, such as the need for low-skew clock distribution, insufficient tolerance to process, temperature and voltage (PVT) variations, and high power dissipation. Instead of a global clock it employs local handshaking to coordinate the activities, which makes operation demand driven and timing much more flexible. One of the main reasons why asynchronous design, although being around for several decades, has still not been widely adopted is the difficulty of testing – in the absence of a clock that the tester can use to control the test procedure, even their off-line test requires considerable efforts. In contrast, the approach we propose here naturally leverages the redundancy already present in the asynchronous 4-phase protocol for introducing test patterns into the data stream in a transparent fashion and fully concurrent with the ongoing operation. The key idea is to build components that present a conventional 4-phase interface to the outside, but internally operate with a 2-phase protocol, which allows test vectors to be inserted between any pair of regular data words, namely during the NULL phase of the external protocol. At the component's output the results pertaining to the regular data stream are presented to the outside, again in a 4-phase fashion, while the test results are internally conveyed to a response analysis block.

The paper is structured as follows: After a review of related work we will present the fundamental concepts of the considered asynchronous design styles in Section III. Section IV will be devoted to presenting our approach in detail. A proof-of-concept implementation will be given and evaluated in Section V. Finally we will conclude the paper in Section VI.

II. Requirements and Related Work

Concurrent checking is a well researched field in dependable computing. Its key principle is to employ some form of redundancy (hardware replication [2], coding [3], repeated execution of a calculation, etc.) to allow checking whether the result of a computation is correct. While this approach

works fine for transient faults, it is not suitable for detecting permanent faults that may reside in a resource that is not exercised by the ongoing operation. Several of these dormant faults may accumulate over time and, once activated together, exceed the capabilities of the checking scheme. In order to safely unveil these faults one cannot simply rely on the ongoing operation to exercise the resources – a test is needed here that actively applies a well selected set of stimuli, independent of what is seen through normal system operation. This is another heavily researched area, however most approaches were developed for synchronous circuits, which sometimes leads to dissatisfying results when used on asynchronous ones.

We have argued above that actively applying test stimuli is desired and characteristic for testing. At the same time these stimuli deliberately change the state of the system under test, which interferes with the ongoing operation, and hence seems to make testing and regular operation mutually exclusive. Methods for online testing must fulfill two conditions: (value domain) non-interference with the system state perceived by the application and (time domain) no degradation of system performance beyond the point where deadlines are missed. This can be achieved by either interleaving phases of test and normal operation in a carefully controlled way, or by devising special test methods that remain transparent for the ongoing operation [4].

The key quality criteria of an online test are

- low performance penalty for the application
- high test coverage for a given fault model; this is determined by the quality and amount of test vectors
- low error detection latency; this is determined by the period required to apply the whole set of test vectors

We could not find approaches for a truly transparent test of asynchronous logic in the literature. The available methods either interrupt the ongoing operation [5] or simply check the output without actively applying test vectors [6]. An interesting combination of these two models is called input vector monitoring in [7]. Here a list of desired test vectors is determined as a subset of all possible inputs during operation. When one of these vectors is encountered during normal operation, the corresponding output is checked against a known reference, and the vector marked successful in the list. The test cycle completes as soon as all vectors in the list have been marked. Variations of this scheme have been proposed that differ in how strictly the sequence within the list must be kept; some even enter a dedicated test mode to apply vectors that are still missing after a timeout.

The approach we propose here is specifically designed for asynchronous logic. It provides a tight interleaving of test and ongoing operation and exploits specific protocol properties to largely eliminate performance penalties. It can be used with any arbitrary set of test vectors, whose generation can be carried out by standard methods from literature.

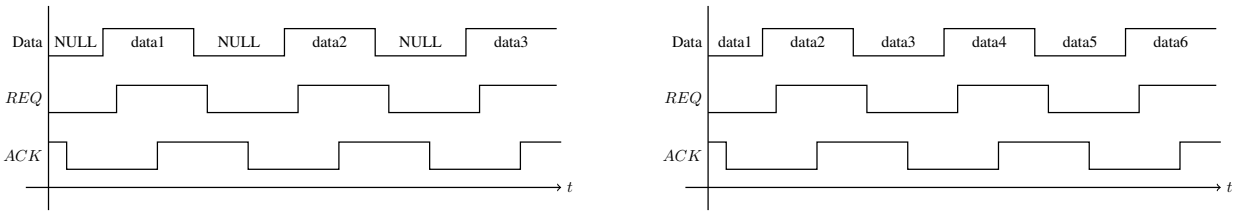
III. Background

In synchronous systems all activities, specifically data exchange, are coordinated by a global clock. Asynchronous design, in contrast, employs explicit handshaking between communicating partners [8]: The sender indicates the validity of the data provided by means of a request (REQ), while the receiver indicates their reception by means of an acknowledge (ACK) signal. This closed-loop principle is the root of the elastic timing behaviour of asynchronous designs. Depending on the specific interpretation associated with the transitions on REQ and ACK two protocols can be distinguished: In the *4-phase protocol* (see fig. 1(a)) the sender indicates data validity by activating REQ, to which the receiver responds by activating ACK as soon as it has captured these data. This is followed by a return-to-zero phase, in which sender and receiver deactivate REQ and ACK, respectively. In the *2-phase protocol* (see fig. 1(b)) that unproductive return-to-zero (RTZ) phase is avoided, and the falling transitions of REQ and ACK already guide the transfer of the next data item. This halves the number of control transitions per data transfer, which makes the 2-phase protocol the preferred choice when data needs to be transferred in an energy-efficient way. The 4-phase protocol, on the other hand, allows a more efficient implementation of logic functions and registers, and is hence usually employed for computation-centric blocks.

The indication of data validity via REQ faces a fundamental race condition: The activation of REQ must be perceived by the receiver only *after* data has actually become valid. The two principles used to ensure this pertain to different timing models of the circuit and have substantially different implementation complexity. In the bounded delay model a delay element Δ is artificially inserted into the REQ signal path that is chosen large enough to accommodate for all potential delays, including combinational functions, that the data may experience on its travel from sender to receiver. Obviously this necessitates a timing analysis and worst case assumptions, just like in the synchronous case. We will further refer to this approach as *bundled data (BD)*, since it uses one REQ for the complete bundle of data. In contrast, the delay insensitive[1] approach uses a more elaborate coding for the data that allows the receiver to evaluate, by means of a so-called completion detector, when a received data item is valid. In this way no explicit REQ line is required any more, thus avoiding the race condition. The advantage of this solution is its ability to accommodate arbitrary delays on the data path without the need for worst case assumptions, its drawback is the necessity of data encoding (typically two signal rails per data bit are required). We will refer to this approach as *completion detection (CD)*. In its 4-phase version two successive data items are separated by a so-called *NULL spacer* that establishes the RTZ phase. In the 2-phase version the coding itself allows a separation of successive data items.

Like in the synchronous case a fundamental structure for a data processing unit is a pipeline, in which register stages separate complex logic operations into smaller ones. The classical pattern in the asynchronous domain is the *Muller pipeline* shown in fig.2. Its constituent component is the Muller C-Element, whose function is as follows: When both inputs match, the same value is reflected on the output; otherwise the output retains its last value. In the 4-phase operation that we will consider in the following, the latches in the datapath are transparent when LE is active, and opaque otherwise.

[1]For the sake of simplicity we disregard the notion of quasi-delay insensitivity here, for a more detailed discussion see e.g. [8]



(a) 4-phase protocol

(b) 2-phase protocol

Fig. 1. Comparison of the two different handshake methods based on the indication of new data.

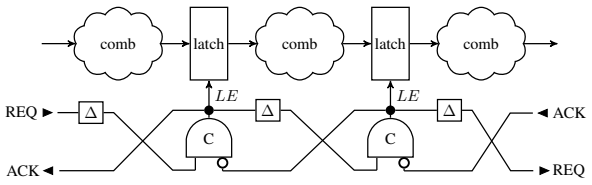


Fig. 2. Fundamental structure of an asynchronous pipeline

IV. Proposed Approach

When comparing the data streams in fig. 1 one can realise that the 2-phase protocol works like a 4-phase protocol with extra data items being conveyed during the RTZ phase. In the CD case this can be understood as replacing the unproductive NULL spacer by productive data[2]. So when processing an incoming 4-phase data stream in a 2-phase function module, we obtain the freedom to insert a data items of our choice in place of the NULL spacers, (ideally) without loss of performance. The key idea of our approach is to use this freedom for inserting a stream of test data items into the original user data stream. Notice that, although we obtain an extremely tight interleaving between ongoing operation and test, this approach is completely decoupled from and transparent to the application, and allows choosing the test data freely.

Figure 3(a) illustrates the basic architecture of our proposed approach. At the input of the device under test (DUT) we place a *4-to-2 phase merge element* (4-to2 PhM) that joins the 4-phase user data stream (UD) with the 4-phase test data stream (TD) into a single 2-phase data stream (UTD). Of course, a source for the test vectors is required here, which is considered part of the self-testing module. In the following we will, however, not go into detail about which test vectors to actually select, these can be freely derived in accordance with the needs of the given DUT by means of the available test pattern generation techniques [9]. Here we will only be concerned with inserting a given test vector into the data stream and extracting the respective response later on.

The DUT now has to process the 2-phase data stream, so its design has to be converted from the original 4-phase protocol to 2-phase. This renders it more complex, which can somehow be considered the price for the online testing property. The DUT's 2-phase output stream finally needs to be separated into the test responses and the results pertaining to the application input data, which are both again 4-phase. This task is performed by a *2-to-4 phase split element* (2-to-4 PhS). The test responses can be analysed (compressed with a multiple-input shift register, e.g., and compared with a stored reference) inside the self-testing function block, while the application data stream is passed on to the actual output where it naturally appears as the 4-phase stream of results that one would expect in response to the original 4-phase input data stream. So from the outside the self-testing DUT behaves like a regular 4-phase logic block.

Interestingly, the approach allows an arbitrary choice of the DUT size: One extreme case would be to consider every single pipeline stage a separate DUT and equip it with all the required infrastructure at its input and output (fig. 3(b)). The other extreme would be to regard the complete design as the DUT(fig. 3(a)), thus trading controllability and observability for lower implementation overheads.

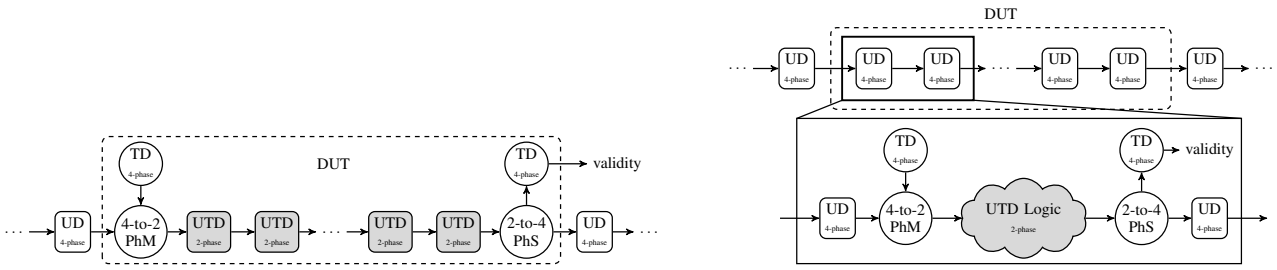
In the following we will focus on the description of the required merge and split elements, since they are fundamental for our approach, and we could not find suitable implementation patterns in the literature – only 2-phase/4-phase conversion of a single data stream [10] has been considered, or splitting and merging of datastreams following the same protocol [11], [12].

A. Merge and split for the bundled data approach

It is possible to compose the merge unit from two nearly independent blocks, one for handling the data bus and one for the control lines. The data handling block boils down to a multiplexer (MUX) that selects between forwarding the user data and the test data. In contrast to other approaches in the literature [11] we have a strict alternation between the two inputs and hence a fixed association between input source and state of REQ in the 2-phase protocol on the output side. This allows to hardwire the MUX's select input to the output REQ, yielding low circuit complexity. In particular we chose to associate user data with $REQ = 1$ and test data with $REQ = 0$. According to the bounded delay model an appropriate delay needs to be added before conveying the REQ signal downstream, to compensate the data delay caused by the MUX.

For the output REQ, termed rUT in fig. 4, we want a rising edge when (a) the REQ of the user data (rU) rises, indicating new user data are available, and (b) the REQ of the test data (rT) falls, indicating the test vector generator is in its RTZ phase – whichever happens last. The same is true for the falling edge of rUT, with the input transitions from rU and rT inverted. The Muller C-element shown in fig. 4(a) (top) serves exactly this purpose. The ACK signal coming from the 2-phase function unit, termed aUT can be simply conveyed as the ACK to the user input (as aU), and after inversion to the test input (aT).

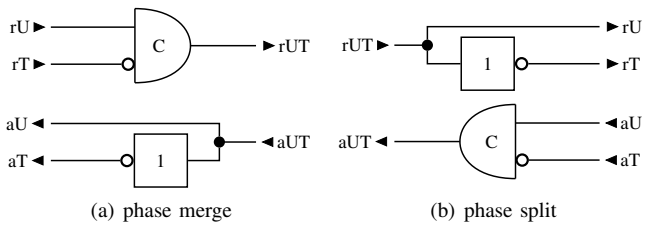
[2]For the BD approach we assume early data validity [8], which is the most common approach anyway.



(a) Complete DUT tested at once

(b) Each stage inside the DUT tested separately

Fig. 3. Principle of the proposed approach showing two different test granularities. Components that need to be adapted for the approach are shaded, additionally required components are shown as circles.



(a) phase merge

(b) phase split

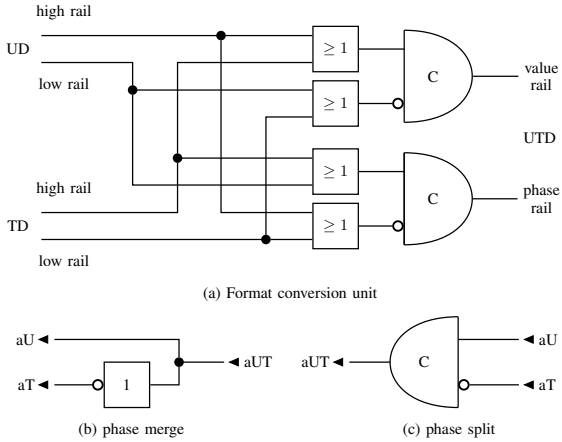
Fig. 4. Handshake signals generation for the bundled data implementation

For the *2-to-4 phase split* element the data handling unit becomes trivial, namely just a set of wire forks: Since the data may assume any arbitrary value during the RTZ phase, all incoming data are directly forwarded to both outputs at the same time. It is up to the REQ signals to indicate which of the outputs is intended to receive the respective data word. Recall that the merge unit associated user data with $rUT = 1$. Therefore we need to activate rU (and deactivate rT) at the split unit output when the $rUT = 1$ is seen at its input, and set $rT = 1$ (and deactivate rU) otherwise. The simple circuit shown in fig. 4(b) (top) does this job and ensures that rU and rT are activated in a mutually exclusive fashion. Merging the ACK responses aU and aT from the 4-phase outputs to a common 2-phase ACK, namely aUT, follows the same pattern as outlined for the REQ signals in the merge unit. Not surprisingly, a Muller C-element with one inverted input, as shown in fig. 4(b) (bottom), does the job.

B. Merge and split for completion detection approach

From the available options for implementing the CD approach we chose NCL as the 4-phase protocol and LEDR as the 2-phase one. As these protocols use different data representations, a bit-level conversion becomes necessary in the merge and split unit. Table I shows the required mapping (per data bit). In the 2-phase protocol we have 2 rails per bit, one value rail (val) and one phase rail (phs). On the 4-phase side we have again 2 rails per bit, this time a one-hot code with one rail indicating high (hi) and one low (lo). For the merge unit we need to convert from 4-phase to 2-phase (right-to-left in the table). Notice that in the 4-phase representation only a single rail is high at a time in each of the four valid states.

The circuit shown in fig. 5(a) identifies these states and maps them to the respective LEDR code. As the two 4-phase inputs (user data and test data) originate in different sources, we cannot avoid invalid intermediate input patterns (i.e. such with 2 rails or no rail at high). This is why we use Muller



(a) Format conversion unit

(b) phase merge

(c) phase split

Fig. 5. Completion detection format conversion from NCL to LEDR

C-elements to retain the valid previous outputs during those phases. The ACK can be treated in the same way as in the BD merge.

In the split unit the format has to be transformed back to the 4-phase protocol (left-to-right in Table I). Notice in the table how the alternation of test data and user data in the 2-phase stream leads to a natural insertion of the required NULL spacers into the 4-phase data streams. The required circuit can be easily derived and is not shown here. A purely combinational (glitch-free) implementation without Muller C-elements is sufficient here, since the 2-phase input does not exhibit invalid intermediate states.

Finally, the generation of the ACK signal is again realised by connecting the incoming ACK lines to a Muller-C element with the test ACK in its negated form.

TABLE I. TRUTH TABLE, FORMAT CONVERSION

2-phase UTD			4-phase UD			4-phase TD		
val	phs	int	hi	lo	int	hi	lo	int
0	0	LO(TD)	0	0	NULL	0	1	LO
0	1	LO(UD)	0	1	LO	0	0	NULL
1	0	HI(UD)	1	0	HI	0	0	NULL
1	1	HI(TD)	0	0	NULL	1	0	HI

C. Enhancements

So far we have presented the basic implementations of the blocks handling the control signals. It is possible to increase their speed at the cost of increased complexity and thus increased area overhead. In the case of the merge element it

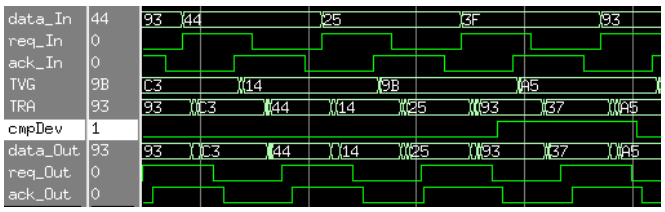


Fig. 6. Post-layout simulation with detection of a stuck-at fault

is possible to acknowledge the NULL phase earlier, namely as soon as the data of the other input are propagated, giving the data values more time to travel through the logic. This yields advantages if NULL values are much faster than data values, which is the case when asymmetric delay lines are used. Another possibility is to propagate new data as soon as they show up, no matter if the other input has already delivered its NULL spacer or not, of course only after the ACK was received from the succeeding stage. Furthermore, introducing a latch at the output of the merge unit makes it possible to acknowledge the inputs right away, resulting in a further decoupling of in- and output.

For the split unit it is possible, to start the NULL phase at the output that received the last data as soon as the input gets acknowledged. Another alternative is to acknowledge the input as soon as the output that recently received the data has acknowledged them, without the necessity of the other one having acknowledged its NULL phase. In addition a latch may be implemented at the input making it possible to acknowledge the input stream right away. For a more detailed and generic treatment of this topic see [13].

V. Evaluation

We verified our online test approach for a three-stage Muller-pipeline. To keep the focus on the newly designed units, we did not introduce combinational functions between the pipeline registers; that would, however, be easy to add in a next step. More specifically we augmented the pipeline by a test vector generator, a response analyzer, and, most importantly, by our proposed merge and split units.

After synthesizing the VHDL design we carried out a post-layout simulation, whose result can be seen in fig. 6. The topmost three traces show the input signals to the DUT and the four traces at the bottom the output signals. TVG represents the output of the test vector generator and TRA the input of the test response analyser. The highlighted signal *cmpDev* gets high as soon as TVG and TRA mismatch, i.e. a fault is detected. Note how the values from $data\_In$ and TVG are processed in an alternating fashion and show up at the output with some delay corresponding to their propagation time through the pipeline.

To validate the self-testing capability of our approach we introduced a stuck-at-0 fault on bit 3 in our design. This fault is activated by the test vector $9B$ which is transformed to 93 (as well as the data vector $3F$ being transformed to 37). As soon as the TRA recognizes 93 it raises *cmpDev*, as intended.

The area overhead and performance penalty introduced by our approach depend on many implementation parameters and are hard to estimate in general. We therefore decided to give an analytic estimation here that still allows to judge the influence of some choices, rather than presenting specific quantitative area and timing data from the synthesized design.

For the *area overhead* we compare the transistor count of the original pipeline with that of the enhancements required for the online testing feature. We do not include the TVG and the TRA in our analysis for two reasons: (1) The need for these units is common to all test approaches, and (2) depending on the specific demands the complexity of these units varies by orders of magnitude. In general, when mapping gates to transistor counts, we did not assume highly optimized cell designs, but we applied simplifications in the overall circuit when they were obvious (like reducing inverter count). The results of our analysis are shown in table II.

The first row analyses the bundled data case. In the first line the transistor count (unit "T" for "transistors") for a stage (register plus control) of the original pipeline is given; in case the transistor count is proportional to the number of data bits "/DL" indicates "per data line". Line 2 gives the *overheads* for the online test. The columns correspond to the individual function blocks (merge and split), with the rightmost column giving the overhead in $\%$ depending on the number n of stages, for large data width and without combinational logic. For the latches the overhead for conversion into a capture/pass latch according to [14] is accounted for as well[3]. In the BD approach the combinational function block (if any) remains unchanged. As there are substantially different ways of implementing the REQ delay required in the BD approach, and furthermore the size largely varies with the required delay value, we did not include it here. This means that the initial size of the pipeline is underestimated here (making the relative overheads seemingly higher), and that the extra delay to compensate for the MUX introduction is not accounted for in the overheads.

The bottom row in table II shows the respective numbers for the CD approach. Here the conversion of the logic is far more complicated because each single gate has to be replaced. Unfortunately no concrete numbers could be found in the literature: That is why we make the pessimistic assumption that the transistor count will duplicate when moving from 4-phase to 2-phase[4]. Furthermore, merge and split blocks need to be added, as well as the completion detector modified.

TABLE II. Area overhead estimated by transistor count

		merge	comb.	pipel. node	split	n stages (%)
BD	**native**	-	-	$14 + 12/DL$	-	$117 + \frac{100}{n}$
	test	$16 + 12/DL$		$14/DL$	16	
CD	**native**	-	-	$2 + 70/DL$	-	$5.7 + \frac{94}{n}$
	test	$2 + 44/DL$	$\approx$*2	$4/DL$	$14 + 22/DL$	

As shown in Table II the overhead for the BD approach is $117 + 100/n\%$. For a test-per-stage approach ($n = 1$) this yields 217%, while for a large number n of pipeline stages between a single pair of merge/split elements, this value drops towards 117%. With large logic function blocks this relative overhead, however, quickly approaches 0%: Consider a combinational block of 12T/DL/pipeline stage; just this approximately halves the overhead.

[3] In contrast to the implementation proposed in [14] we did not account 8T per switch but rather 4T (transmission gate).

[4] In a very simple example that we used for a first comparison, we could build an XOR for NCL with about 70T, while its counterpart in LEDR required 100T, yielding an overhead of less than 50%.

For the CD approach with its more complex native pipeline stages the relative overhead is much lower. However, as, according to our pessimistic estimation, converting the logic function blocks roughly duplicates their transistor count, the situation does not improve with large function blocks.

For estimating the *performance penalty* we identify the additional delays introduced by the test infrastructure. To attain a generic view we consider gate delays (measured in inverter delays ID of the respective technology) and assume zero wire delays. The results are summarized in table III (k = number of data lines). They show the accumulated values for forward and backward (ACK) path. In all cases we assume that TVG and TRA operate fast enough to perform the handshaking without extra delays.

The numbers for the BD approach are shown in the first row, with the first line referring to the native implementation and the second one to the overhead for the online test infrastructure. The introduction of the merge and split units causes a delay of 5 and 2 ID, respectively. Relative to the stage delay this represents a penalty of more than 100%. However, as the number of stages between merge and split, as well as the delays of the logic function blocks (not considered here) grow, the relative penalty quickly approaches 0%. Similarly, the extra 1 ID for the register stage becomes negligible in case of complex combinational function blocks with high delays.

For the CD approach the picture is again initially better (penalty below 100%), as the native pipeline stage has more delay. The problematic point, however, is, once more, the complexity increase when transforming the combinational logic from 4-phase operation to 2-phase. The related performance penalty strongly depends on the specific circuit; we roughly estimate it as $50...100\%$. Unfortunately, this number does not scale down with the number of stages or with the initial complexity of the combinational logic, as in the BD approach.

TABLE III. PERFORMANCE PENALTY IN GATE DELAYS

		merge	comb.	pipel. node	split	n stages (%)
BD	**native**	-	-	6	-	$17 + \frac{117}{n}$
	test	5	-	1	2	
CD	**native**	-	-	$9 + \lceil log_2(k) \rceil * 2$	-	$\frac{n+8}{n(2*\lceil log_2(k) \rceil + 9)}$
	test	4	$\approx *1.5 - 2$	1	4	

VI. CONCLUSION

We have proposed to exploit the, normally unproductive, RTZ phase or NULL spacers of the asynchronous 4-phase protocols for conveying test vectors. While this can be done fully transparent and concurrent to the ongoing application, a new test vector can be applied after every single data word, which yields the tightest possible interleaving between test and operation, and hence an excellent detection latency. Test vectors can be freely chosen, independent from the user data, to optimize test coverage versus test period. We have identified the required infrastructure blocks for this approach and illustrated their basic implementation. In a case study we have proven the feasibility of the approach.

For the BD approach the area overheads can, according to our estimations, go up to 200% under the most pessimistic assumptions. Fortunately they approach 0% quickly with increasing number of stages and complexity of the combinational logic. The performance penalty can be close to 150%, with the same favorable trends. So in practical cases the overheads will be very moderate. The CD approach exhibits lower relative overheads in the worst case scenarios, simply because the native implementation is more complex already. However, for the conversion of the combinational logic from 4-phase to 2-phase it is difficult to estimate the incident penalties. Under our pessimistic assumptions the overheads for this conversion dominate, and therefore we cannot attain the favorable scaling as seen with the BD approach.

However, even in the worst cases the observed overheads are still competitive with those of typical fault-tolerance methods like TMR, duplication or time redundancy, given the superior performance: In contrast to these fault-tolerance approaches that are based on concurrent checking, our online test detects permanent faults in the hardware, which are hard to unveil otherwise, with the best attainable (namely cycle-wise) interleaving between application and test.

Future work will be devoted to increasing the concurrency within the merge and split modules, as already sketched in this paper. This should aid in further reducing the performance penalty. Furthermore, it will be interesting to study the properties of the approach in more complex settings.

REFERENCES

[1] C. Scherrer and A. Steininger, "Dealing with dormant faults in an embedded fault-tolerant computer system," *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 512–522, 2003.

[2] T. Verdel and Y. Makris, "Duplication-based concurrent error detection in asynchronous circuits: Shortcomings and remedies," in *DFT*. IEEE Computer Society, 2002, pp. 345–353.

[3] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[4] M. Nicolaidis and Y. Zorian, "On-line testing for vlsi - a compendium of approaches," *J. Electron. Test.*, vol. 12, no. 1-2, pp. 7–20, Feb. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1008244815697

[5] D. Koppad and A. Efthymiou, "Bist for strongly-indicating asynchronous circuits," in *Very Large Scale Integration (VLSI-SoC), 2009 17th IFIP International Conference on*, Oct 2009, pp. 215–218.

[6] N. Minas, M. Marshall, G. Russell, and A. Yakovlev, "Fpga implementation of an asynchronous processor with both online and offline testing capabilities," in *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, April 2008, pp. 128–137.

[7] K. K. Saluja, R. Sharma, and C. R. Kime, "A concurrent testing technique for digital circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 7, no. 12, pp. 1250–1260, 1988.

[8] J. Sparso, S. B. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer, 2001.

[9] M. Bushnell, V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer, 2000.

[10] A. Mitra, W. F. McLaughlin, and S. M. Nowick, "Efficient asynchronous protocol converters for two-phase delay-insensitive global communication," in *ASYNC*, 2007, pp. 186–195.

[11] M. Ferringer, "Conversion and interfacing techniques for asynchronous circuits," in *Design and Diagnostics of Electronic Circuits & Systems*, 2011, pp. 11–16.

[12] ——, "Conversion of two- to four-phase delay-insensitive asynchronous circuits," in *EUROCON*. IEEE, 2011, pp. 1–4.

[13] R. Najvirt, S. Naqvi, and A. Steininger, "Classifying virtual channel access control schemes for asynchronous nocs," in *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, 2013, pp. 115–123.

[14] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989. [Online]. Available: http://doi.acm.org/10.1145/63526.63532

Bibliography

[1] M.S. Abdelfattah, L. Bauer, C. Braun, M.E. Imhof, M.A Kochte, Hongyan Zhang, J. Henkel, and H. Wunderlich. Transparent structural online test for reconfigurable systems, June 2012.

[2] Vladimir Castro Alves, Felipe M. G. França, and Edson do Prado Granja. A bist scheme for asynchronous logic. In *Asian Test Symposium*, pages 27–32. IEEE Computer Society, 1998.

[3] N. Bartzoudis, V. Tantsios, and K. McDonald-Maier. Dynamic scheduling of test routines for efficient online self-testing of embedded microprocessors. In *On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International*, pages 185–187, July 2008.

[4] Erik Brunvand, Steven M. Nowick, and Kenneth Y. Yun. Practical advances in asynchronous design and in asynchronous/synchronous interfaces. In *DAC*, pages 104–109, 1999.

[5] C. Constantinescu. Impact of deep submicron technology on dependability of vlsi circuits. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 205–209, 2002.

[6] Altera Corporation, Publisher of Quartus II and ModelSim-Altera. `http://www.altera.com/`. Accessed: 2013-11-06.

[7] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A Lindoso, and L. Entrena. Exploiting the debug interface to support on-line test of control flow errors. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 98–103, July 2013.

[8] Markus Ferringer. Conversion and interfacing techniques for asynchronous circuits. In *Design and Diagnostics of Electronic Circuits & Systems*, pages 11–16, 2011. talk: 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2011), Cottbus, Germany; 2011-04-13 – 2011-04-15.

[9] Markus Ferringer. Conversion of two- to four-phase delay-insensitive asynchronous circuits. In *EUROCON*, pages 1–4. IEEE, 2011.

[10] M. Grosso, M.S. Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil, and L. Entrena. An on-line fault detection technique based on embedded debug features. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 167–172, July 2010.

[11] Scott Hauck. Asynchronous design methodologies: An overview. In *PROCEEDINGS OF THE IEEE*, pages 69–93, 1995.

[12] D. Koppad and A. Efthymiou. Bist for strongly-indicating asynchronous circuits. In *Very Large Scale Integration (VLSI-SoC), 2009 17th IFIP International Conference on*, pages 215–218, Oct. 2009.

[13] J.C. Laprie. Dependability: Basic concepts and terminology. In J.C. Laprie, editor, *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*, pages 3–245. Springer Vienna, 1992.

[14] Jürgen Maier and Steininger Andreas. Online test vector insertion – a concurrent built-in self-testing (cbist) approach for asynchronous logic. In *Design and Diagnostics of Electronic Circuits & Systems*, pages 33–38, 2014. talk: 17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2014), Warsaw, Poland; 2014-04-23 – 2014-04-25.

[15] Amitava Mitra, William F. McLaughlin, and Steven M. Nowick. Efficient asynchronous protocol converters for two-phase delay-insensitive global communication. In *ASYNC*, pages 186–195. IEEE Computer Society, 2007.

[16] S Mitra and E.J. McCluskey. Which concurrent error detection scheme to choose ? In *Test Conference, 2000. Proceedings. International*, pages 985–994, 2000.

[17] S.K. Mohideen and J. RajaPaul Perinbam. Design of built in self test asynchronous micropipeline using double edge triggered d flip flop. In *INDICON, 2005 Annual IEEE*, pages 322–326, Dec. 2005.

[18] M. Nicolaidis. Self-exercising checkers for unified built-in self-test (ubist). *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(3):203–218, Mar 1989.

[19] M. Nicolaidis. Theory of transparent bist for rams. *Computers, IEEE Transactions on*, 45(10):1141–1156, Oct 1996.

[20] M. Nicolaidis and Y. Zorian. On-line testing for vlsi - a compendium of approaches. *J. Electron. Test.*, 12(1-2):7–20, February 1998.

[21] O. Novak and H. Nosek. Test-per-clock testing of the circuits with scan. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 90–92, 2001.

[22] A Paschalis and D. Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(1):88–99, Jan 2005.

[23] O. A. Petlin and Stephen B. Furber. Built-in self-testing of micropipelines. In *ASYNC*, pages 22–29. IEEE Computer Society, 1997.

[24] Petrify: a tool for synthesis of Petri nets and asynchronous circuits. `http://www.lsi.upc.edu/~jordicf/petrify/`. Accessed: 2013-06-23.

[25] Janusz Rajski and Jerzy Tyszer. Test responses compaction in accumulators with rotate carry adders. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(4):531–539, 1993.

[26] M. Richter and M. Goessel. Concurrent checking with split-parity codes. In *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, pages 159–163, June 2009.

[27] Marly Roncken, Ken S. Stevens, Rajesh Pendurkar, Shai Rotem, and Parimal Pal Chaudhuri. Ca-bist for asynchronous circuits: A case study on the rappid asynchronous instruction length decoder. In *ASYNC*, pages 62–72. IEEE Computer Society, 2000.

[28] Kewal K. Saluja, Rajiv Sharma, and Charles R. Kime. A concurrent testing technique for digital circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(12):1250–1260, 1988.

[29] Christoph Scherrer and Andreas Steininger. Dealing with dormant faults in an embedded fault-tolerant computer system. *IEEE Transactions on Reliability*, 52(4):512–522, 2003.

[30] D. Shang, A. Bystrov, A. Yakovlev, and D. Koppad. On-line testing of globally asynchronous circuits. In *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, pages 135–140, July 2005.

[31] R. Sharma and K.K. Saluja. An implementation and analysis of a concurrent built-in self-test technique. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 164–169, Jun 1988.

[32] E. Simeu and A Abdelhay. A robust fault detection scheme for concurrent testing of linear digital systems. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 209–214, 2001.

[33] Amardeep Singh. Quantum search algorithm for automated test pattern generation in vlsi testing. In Hamid R. Arabnia and Laurence Tianruo Yang, editors, *VLSI*, pages 217–223. CSREA Press, 2003.

[34] J. Sparsø. Asynchronous circuit design - a tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*. Kluwer Academic Publishers, Boston / Dordrecht / London, dec 2001.

[35] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[36] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[37] Frank J. te Beest. *Full scan testing of handshake circuits*. PhD thesis, University of Twente, 2003.

[38] W. B. Toms, David A. Edwards, and Andrew Bardsley. Synthesising heterogeneously encoded systems. In *ASYNC*, pages 138–149. IEEE Computer Society, 2006.

[39] Qiao Tong. Built-in current self-testing scheme (bicst) for cmos logic circuits. In *VLSI Test Symposium, 1992. '10th Anniversary. Design, Test and Application: ASICs and Systems-on-a-Chip', Digest of Papers., 1992 IEEE*, pages 304–308, April 1992.

[40] P. Udaya. Euclid's algorithm and lfsr synthesis. In *Information Theory, 2000. Proceedings. IEEE International Symposium on*, pages 420–, 2000.

[41] Thomas Verdel and Yiorgos Makris. Duplication-based concurrent error detection in asynchronous circuits: Shortcomings and remedies. In *DFT*, pages 345–353. IEEE Computer Society, 2002.

[42] Taavi Viilukas, Jaan Raik, Maksim Jenihhin, Raimund Ubar, and Anna Krivenko. Constraint-based test pattern generation at the register-transfer level. In Elena Gramatová, Zdenek Kotásek, Andreas Steininger, Heinrich Theodor Vierhaus, and Horst Zimmermann, editors, *DDECS*, pages 352–357. IEEE, 2010.

[43] I. Voyiatzis, D. Gizopoulos, and A. Paschalis. An input vector monitoring concurrent bist scheme exploiting "x" values. In *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, pages 206–207, Jun 2009.

[44] I. Voyiatzis, Th. Haniotakis, C. Efstathiou, and H. Antonopoulou. A concurrent bist architecture on monitoring square windows. In *International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, 2010.

[45] Ioannis Voyiatzis. Input vector monitoring on line concurrent bist based on multilevel decoding logic. In Wolfgang Rosenstiel and Lothar Thiele, editors, *DATE*, pages 1251–1256. IEEE, 2012.

[46] Ioannis Voyiatzis and Constantin Halatsis. A low-cost concurrent bist scheme for increased dependability. *IEEE Trans. Dependable Sec. Comput.*, 2(2):150–156, 2005.

[47] Ioannis Voyiatzis, Dimitris Nikolos, Antonis M. Paschalis, Constantinos Halatsis, and Th. Haniotakis. An efficient comparative concurrent built-in self-test technique. In *Asian Test Symposium*, pages 309–315. IEEE Computer Society, 1995.

[48] Ioannis Voyiatzis, Antonis M. Paschalis, Dimitris Gizopoulos, Constantin Halatsis, Frosso S. Makri, and Miltiadis Hatzimihail. An input vector monitoring concurrent bist architecture based on a precomputed test set. *IEEE Trans. Computers*, 57(8):1012–1022, 2008.

[49] Ioannis Voyiatzis, Antonis M. Paschalis, Dimitris Nikolos, and Constantinos Halatsis. R-cbist: an effective ram-based input vector monitoring concurrent bist technique. In *ITC*, pages 918–925. IEEE Computer Society, 1998.

[50] Peter Wohl, John A. Waicukauski, and Thomas W. Williams. Design of compactors for signature-analyzers in built-in self-test. In *ITC*, pages 54–63. IEEE Computer Society, 2001.

[51] Xin Yuan and C.-I.H. Chen. Automated synthesis of a multiple-sequence test generator using 2-d lfsr. In *ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International*, pages 75–79, 1998.

[52] S. Zeidler, A. Bystrov, M. Krstic, and R. Kraemer. On-line testing of bundled-data asynchronous handshake protocols. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pages 261–267, July 2010.